

Broadview
www.broadview.com.cn

案例的选择与评估
类模板特化
标准库及Qt对字符串的处理
国际化与区域文化
C++的iostream

Qt的流
隐式共享与d-pointer技术
Qt容器与迭代器
多线程与可重入
信号与槽
Graphics/View框架

Model/View框架
Qt中的命令模式
Qt中的抽象工厂模式
Qt中的观察者模式
Qt的元对象系统
智能指针

Qt中的 C++技术

· 张波 编著 ·

国内首本深入剖析Qt设计理念及源代码的书籍

- 研读最优秀的C++开源项目，学习如何使用C++
- 以**80个**形象、鲜活的例子诠释面向对象设计思想
- 系统阐述了**8种**设计模式
- 分析比较了Qt与**C++标准库**的设计理念
- 全面覆盖C++的各种**语言特性**

让我们从这本书开始，一起体验C++处理复杂问题的强大及其精妙吧！

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

内 容 简 介

本书剖析了开源开发框架 Qt 中的 C++ 技术, 给读者提供一个优秀的案例, 以学习 C++ 语言以及面向对象设计技术。该书讨论了以下内容: 类模板特化技术; 分析比较了 C++ 标准库、Qt 对字符串、数据输入/输出的处理思路; 隐式共享与 d-pointer 技术; 函子及其在 QTL (Qt Template Library) 中的应用, QTL 是如何使用模板特化技术优化 QList 性能的; 如何在 C++ 程序中嵌入汇编代码, 实现一个原子操作, 以很小的开销实现线程间通信; 信号与槽机制; Graphics/View 框架等。

软件学院或者计算机学院的学生, 可将本书作为课程“C++ 程序设计”或者“面向对象软件设计”的参考书; 上述课程的教师, 可将本书的内容融入他们的主讲或者试验环节, 作为相关实训课程的教材; 软件行业的开发者, 可将本书作为深入学习 C++ 设计与编程技术的案例教材。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目 (CIP) 数据

Qt 中的 C++ 技术 / 张波编著. —北京: 电子工业出版社, 2012.7
ISBN 978-7-121-17159-8

I. ①Q… II. ①张… III. ①软件工具—程序设计 ②C 语言—程序设计
IV. ①TP311.56 ②TP312

中国版本图书馆 CIP 数据核字 (2012) 第 106582 号

责任编辑: 孙学瑛

特约编辑: 顾慧芳

印 刷: 北京中新伟业印刷有限公司
装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 19.25 字数: 444 千字

印 次: 2012 年 7 月第 1 次印刷

印 数: 4000 册 定价: 55.00 元 (含光盘 1 张)

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。

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

服务热线: (010) 88258888。

据 2011 年 12 月 Tiobe 网站 (www.tiobe.com) 的排名, 最流行的前 5 个编程语言依次是 Java、C、C++、C#以及 Objective-C。排名的依据是熟练使用一种语言的人数, 与该语言相关的课程数量以及支持该语言的第三方供应商的数量。自 2001 年这个排名标准诞生以来, C++几乎总是处于第 3 名。虽然 Perl、Visual Basic 以及 PHP 也曾占据过这个位置, 但是它们只能在这个位置上维持几个月。

软件行业的骨架是由 C++ (及其兄弟 C) 搭建起来的。

- 三大桌面操作系统 Windows, MacOS 以及 Chrome OS 使用了 C++。
- 运行在苹果公司 iPhone, iPod, iTouch 以及 iPad 上的操作系统, Windows Mobile 和 Symbian OS 使用了 C++。
- 在关系数据库管理系统方面, 主流的产品 Oracle Database, MySQL, IBM DB2, Microsoft SQL Server, IBM Informix, SAP DB/MaxDB 都使用了 C++。
- Web 浏览器方面, 依据网络分析公司 StatCounter 2011 年 11 月发布的数据, 全球用户数量最多的前 5 款浏览器依次为微软的 IE, Google Chrome, Mozilla Firefox, Safari 以及 Opera, 它们全部使用了 C++。
- 流行的办公套件 Microsoft Office, Sun Open Office, Corel Office 也都使用了 C++, 其中 Corel Office 在开发过程中曾使用过 Java, 但由于速度太慢最终转回 C/C++。
- 甚至, 出乎一般人的预料, C++也被用来开发网站。Google 的网站采用 C++ (及汇编), eBay 和 Amazon 采用 C++与 Java, Facebook 采用 LAMP (Linux+Apache+Mysql+PHP) 外加 C++。

C++的语言特性

C++的地位是由其鲜明的语言特性决定的。

(1) 它兼容 C, 意味着 C++项目可以复用过去 40 年来积累的 C 函数库以及 C 源代码。

(2) 对复杂系统的抽象表达能力。C++和其他面向对象编程语言一样, 使用“类”封装底层的数据和相关的操作, 使用“继承”描述基类和派生类的共性, 使用“多态性”描述不同子类的差异。一个复杂的系统总会被表示为一组具有清晰逻辑结构的类。为了解决复杂软件系统中名字冲突的问题, C++引入了函数名重载 (function overloading)、名字空间 (namespace) 机制。

(3) 执行速度快。C++和其他面向对象编程语言不同, 并不追求形式上的简单。为了提高程序执行速度, C++不惜将自己变得更加复杂。模板技术以及基于该技术的泛型编程思想

在编译阶段处理多态性问题，使得一个类模板或者函数模板能够处理各种类型。而其他编程语言采用虚函数与多态性解决类似的问题，相比之下，C++模板的执行速度更快。

(4) C++遵循“用时付费”原则。例如，有些编程语言将垃圾回收（*garbage collection*）作为语言内在的功能，无论一个程序是否需要，该机制在程序运行期间总会占用一定的计算资源。对于C++，即使演化到最新的C++0x11标准，它仍然坚持不纳入垃圾回收机制，因为它认为需要该功能的程序可以采用第三方库，而不需要该功能的程序不应花费任何额外计算资源。C++程序被编译为可执行代码后可以直接运行，不像其他一些语言那样需要一个虚拟机来解释执行。在对速度要求极高的场合，我们可以在C++程序中嵌入汇编代码。这些语言特性都能提高C++程序的执行速度。

本书特点

然而，也正因为C++的这些语言特性，使其成为所有面向对象语言中最复杂的一个。许多学习者抱怨C++语言“难学难精”。为了降低学习难度，提高学习效率，C++学习者至少应该阅读两种类型的教材：

- 讲述“C++是什么”的教材，如 *Thinking in C++* 以及 *C++ Primer* 等。
- 讲述“如何使用C++”的教材，如 *Effective C++* 以及 *The C++ Programming Language*（每章的 *Advice* 部分与 *Part IV*）。

阅读第一种类型的教材时，学习者通常不会遇到什么障碍，然而在阅读第二种类型的教材时，学习者有时会觉得其中一些C++编程经验听起来有道理，但是在编程实践中并不一定能够灵活运用它们。有这种感觉并不奇怪，因为这些编程经验是C++高手们从几十年的编程实践中总结出来的，要真正理解它们，需要读者也能以某种方式重演这些经验背后的编程实践。

一种最有效的方式就是选择一个优秀的C++案例，研读它、模仿它，也就是说，C++学习者还应该阅读第三种类型的教材：案例教材。

目前有如下两种类型的案例教材。

(1) 偏重介绍如何使用某种开发框架（如MFC）的教材。学习者看完这些教材，的确可以编写出一些具有图形界面的程序，但是无法体会如何使用C++语言的特性来解决复杂的设计问题、性能问题。更值得注意的是，这些教材中的案例往往都是为了书籍的出版而特意设计的，并非来源于实际软件项目。其系统设计、代码质量未经同行审阅，可能会误导学习者。

(2) 深度剖析C++标准模板库（STL）的教材。由于STL本身精巧的设计，阅读这类教材的确可以帮助学习者掌握C++（尤其是模板技术）的精髓，但是这类教材抽象而深奥，令大多数学习者望而却步。

读者希望临摹如下这样的案例。

(1) 案例具有良好的设计、高质量的代码，已被成功地广泛使用。

(2) 案例不要涉及太多、太深其他领域的知识。

(3) 案例可以比较复杂（以展现 C++对复杂问题的处理能力）但是不要过于抽象，这样，读者可以通过形象、鲜活、具体的例子，体会 C++语言特性的应用。

(4) 案例应该全面覆盖 C++的各种语言特性。

寻找这样一个案例并非易事。在众多的开源项目中，我们最终选择了 Qt。Qt 是一个跨平台的 C++开发框架，它包含一个功能丰富的 C++类库以及一套简便易用的集成开发工具。Qt 所支持的平台不但包括 Linux，Windows 以及 Mac OS X 等主流桌面操作系统，还包括诸如 Symbian，Maemo 以及 MeeGo 这样的嵌入式操作系统。

使用 Qt 编写的 C++程序具有良好的跨平台特性，程序员几乎无须更改源代码，所编写的应用程序即可运行在各种操作系统中，这能大幅度缩短开发周期、降低开发成本。Qt 的 C++类库是完全面向对象的，该类库不但功能强大，而且设计精良、方便易用。这些优点使得 Qt 被 Adobe®，Boeing®，Google®，IBM，Motorola®，NASA，Skype®等大型机构以及众多的中小公司采用。

Qt 类库非常复杂，仅本书剖析的两个子模块 QtCore 以及 QtGui 就有大约 22 万行代码，其复杂程度足以展现 C++对复杂问题的处理能力。该类库不但全面覆盖了 C++的各种语言特性，还用到了 MVC（Model-View-Control）框架、隐式共享、信号与槽、命令模式、抽象工厂模式、观察者模式等精妙的设计技巧，是一个值得学习者临摹的案例。

本书内容

本书共 18 章。第 1 章讲述为什么会从众多的开源 C++项目中选择 Qt。读者可以借鉴其中的方法选择其他 C++案例，或者在学习其他编程语言时，使用其中的方法选择对应的案例。而且，读者还可以使用其中的工具 CppDepend 剖析其他软件的结构与质量。这一章还介绍了本书对术语、UML 类图方面的约定。在阅读后续章节前，读者应该首先阅读这一章。

本书不但剖析 Qt 的源代码，有的章节还涉及修改 Qt 的源代码，此时需要重新编译整个 Qt 库。第 2 章简要介绍 Qt，并讲述如何在 Visual Studio 2010 开发环境下安装、编译 Qt 库。Qt 库多处用到了类模板特化技术。考虑到一般的 C++教科书不会详细讲解这个话题，故第 3 章阐述该技术的概念和基本应用，第 6 章及第 9 章用到了该技术。

绝大多数 C++程序都会涉及字符串的处理，因而在 Qt 提供的所有功能中，我们首先在第 4 章讲述 Qt 对字符串的处理思路。和 C++标准库不同，Qt 不再区分单字节字符串、宽字节字符串，而是使用类 QString 表示所有类型的字符串，每个字符用类 QChar 表示，至少占用 2 个字节。这一章分析比较了 C++标准库、Qt 对字符串的两种处理思路。

数据的输入输出也是绝大多数 C++程序需要使用的功能。第 5 章～第 7 章分析比较 C++

标准库、Qt 对数据输入/输出的不同处理思路。Qt 的流框架功能丰富，易于使用但是可扩展性差。而 C++ 标准库的流框架具有良好的可扩展性，但是其本身的功能偏弱，不便于使用。

如果一个类的某些对象的数据成员具有完全相同的取值，我们可以令它们共享一块内存以节省空间。只有当程序需要修改其中某个对象时，我们再为其分配新的内存。这种技术被称为隐式共享，在 Qt 库中被广泛使用。第 8 章介绍该技术，并剖析了 QString 的部分源代码以演示该技术的具体实现。除此之外，这一章还介绍了 d-pointer 技术，该技术将一个类的所有数据成员分离出来，定义在另外一个类中，并在原先的类中定义一个指针，指向另外一个类。这种技术能维持 Qt 库的二进制数据兼容性、提高 Qt 库的编译速度。

起初开发 Qt 时，C++ 的标准模板库 STL 尚未成为业界标准，部分编译器根本没有配备 STL。为了实现良好的跨平台特性，Qt 的设计者干脆开发了一个类似于 STL 的模块，将其集成在 Qt 软件体系中，本文将其称为 QTL (Qt Template Library)。Qt 其他模块大量使用了 QTL，使其成为 Qt 软件体系的基石。正如设计精良的 STL 能够引来人们为其出书立著一样，QTL 的设计也有可圈可点的特点。第 9 章介绍函数的概念及其在 QTL 中的应用，以及 QTL 是如何使用模板特化技术优化 QList 性能的。

目前的 C++ 标准并不支持并发处理，但是并发处理的确可以提高程序性能、改善用户体验。因此，Qt 封装了不同操作系统的细节，向程序员提供了一组 and 平台无关的类来处理并发问题。第 10 章介绍如何使用 QThread 创建一个线程，如何使用互斥体 QMutex，信号量 QSemaphore 及条件量 QWaitCondition 实现线程同步。以上这些技术运行速度慢、开销大。这一章以多线程环境下 singleton 模式的实现为例，讨论了如何在 C++ 程序中嵌入汇编代码，实现一个原子操作，并巧妙地利用这个原子操作，以很小的开销、非常快的速度同步多个线程。

Qt 使用信号与槽机制进行对象间的通信。第 11 章阐述 Qt 为什么使用信号与槽机制而不是传统的回调函数进行对象间的通信。信号与槽机制大幅降低了各对象之间的耦合度，各对象的代码可以被独立地开发、测试，也更容易被复用。这种机制是 Qt 开发框架区别于其他框架的典型特征。

Qt 的 Graphics/View 框架被用来存放、显示二维图形元素，处理那些对图形元素进行操作的交互命令。第 12 章介绍 Qt 图形系统的基本知识，Graphics/View 框架如何处理图形元素，程序员如何定义新的派生类，以配合该框架实现动画效果。通过学习本章，读者将体会到虚函数的作用：一个开发框架实现基本、通用的功能。通过重载开发框架提供的虚函数，应用程序的代码能够 and 框架中的代码协同工作，达到复用框架代码的目的。

模型-视图-控制器架构 (Model-View-Control structure, MVC) 是一种流行的软件体系架构，它将数据及其显示分离开来，使用模型存取数据，使用视图显示数据，使用控制器处理用户的交互命令。第 13 章详细阐述 Qt 如何实现这个框架。该框架涉及 Qt 库的 36 个类，结构复杂、设计精良。这一章的篇幅最长，读者将从本章体会到一个复杂面向对象系统背后的设计理念。

Qt 库用到多种设计模式。第 14 章~第 16 章阐述 Qt 如何实现命令模式（command pattern）、抽象工厂模式（abstract factory pattern）以及观察者模式（observer pattern）。对于每一种模式，我们先简要介绍它的一般定义，再讨论 Qt 使用哪些类/机制实现这种模式，最后给出一个例子演示该模式的作用。

Qt 库的另外一个显著特征是构建了一个比标准 C++ 的运行时类型信息 RTTI（Run-time Type Information）系统功能更强大的元对象系统（meta-object system）。第 17 章回顾了标准 C++ 中的 RTTI 系统，讨论了它的局限性，阐述了 Qt 的元对象系统以及如何使用该系统来获取对象的运行时信息。

C++ 的指针是一把双刃剑：一方面，它可被用来构建复杂的数据结构。但是另一方面，对指针的错误使用将导致内存泄漏以及野指针等问题。第 18 章介绍 Qt 提供的一些智能指针。这些智能指针对普通 C++ 指针进行封装，能够有效解决上述问题。

本书虽然侧重于剖析 Qt，但不是系统讲授如何使用 Qt 开发 GUI 应用程序。然而，本书的确详细阐述了 Qt 的一些基本组成部分，比如输入/输出流、隐式共享、信号与槽、Graphics/View 框架、Model/View 框架等，理解这些组成部分可以帮助读者尽快掌握 Qt 的使用。

本书读者对象

本书面向以下读者：软件学院或者计算机学院的学生，将本书作为学习“C++ 程序设计”或者“面向对象软件设计”课程的参考书；上述课程的教师，将本书的内容融入他们的主讲或者试验环节；软件行业的开发者，与本书一起领略 Qt 的精妙。

笔者由衷地感谢与敬佩 Qt 的创始人 Haavard Nord 以及 Eirik Chambe-Eng。正是他们为 C++ 世界带来了如此精彩的一个开发框架，并且无私地公开了整个框架的源代码，允许成千上万的 C++ 程序员以 GPL 授权方式自由地使用、修改、发布这个库。另外，本书源于南开大学以及南开大学软件学院资助的一个教学改革项目，在此一并表示感谢。最后，感谢董颖、于洋等人对本书草稿的校对。

虽然笔者付出了近两年时间剖析 Qt、撰写此书，但由于 C++ 以及 Qt 的复杂，本书未能涵盖 Qt 的所有技术亮点。对本书的批评与建议请发送至 zhangbo_nk@163.com。关于本书的勘误、讨论以及其他信息，请访问笔者个人博客 blog.csdn.net/zhangbo_nk。

张 波

2012 年 3 月 1 日于南开大学

第 1 章 案例的选择与评估	1
1.1 案例的初步选择	1
1.2 案例的定量评估	3
1.3 其他案例	5
1.4 基本约定	6
1.5 关于类图的约定	8
第 2 章 Qt 概述	11
2.1 Qt 版权	13
2.2 Qt 库的编译	14
2.3 开发环境的设置	16
2.4 主控台的输入与输出	18
2.5 Qt 风格的编程规范	19
2.6 与 Qt 及 C++ 相关的文献资源	21
第 3 章 类模板特化	24
3.1 类模板特化	24
3.2 Traits 技术	27
3.3 类型分类 (Type Classification) 技术	28
3.4 降低代码膨胀	30
第 4 章 标准库及 Qt 对字符串的处理	32
4.1 字符及其编码	32
4.2 标准库的类模板 <code>basic_string</code>	34
4.3 Qt 的类 <code>QString</code>	37
第 5 章 国际化与区域文化	41
5.1 区域文化	41
5.2 <code>facet</code>	44
5.3 类 <code>locale</code> 的实现	50
5.4 类模板 <code>facet</code> 的实现	52

5.5 派生新的 facet 类	53
第 6 章 C++的 iostream	56
6.1 C 语言的 scanf/printf 函数组	56
6.2 iostream 的总体结构	57
6.3 字符特征的描述	61
6.4 模板特化后的总体结构	64
6.5 文件流	65
6.6 字符串流	71
6.7 流缓冲区	73
6.8 二进制文件的处理	76
6.9 用户自定义类型的输入和输出	77
第 7 章 Qt 的流	79
7.1 文件系统及底层文件操作	80
7.2 类 QTextStream	81
7.3 类 QDataStream	83
7.4 类 QLocale	86
7.5 iostream 和 Qt 流类的比较	87
第 8 章 隐式共享与 d-pointer 技术	88
8.1 隐式共享	88
8.2 d-pointer 在隐式共享中的应用	90
8.3 二进制代码兼容	92
8.4 d-pointer 模式的实现	96
8.5 QObject 中的 d-pointer	98
第 9 章 Qt 容器与迭代器	101
9.1 QTL 概述	101
9.2 QTL 容器和 QDataStream 的无缝连接	107
9.3 类型分类技术在 QList 中的应用	109
9.4 函子的应用——相关词词典	112
第 10 章 多线程与可重入	115
10.1 创建一个线程	116
10.2 线程间同步	116

10.3	线程安全与可重入	121
10.4	多线程环境下的 singleton 模式	122
第 11 章	信号与槽 (Signals and Slots)	134
11.1	对象树 (QObject Tree)	134
11.2	信号与槽机制	135
11.3	信号与槽的应用例子	140
第 12 章	Graphics/View 框架	145
12.1	Qt 图形系统介绍	145
12.2	Graphics/View 框架	146
12.3	例子——相撞的老鼠	147
第 13 章	Model/View 框架	152
13.1	Model/View 框架总体架构	154
13.2	模型 (Models)	157
13.3	视图 (Views)	179
13.4	选择操作	180
13.5	委托 (Delegates)	185
13.6	代理模型 (Proxy Models)	188
13.7	便利视图类	199
第 14 章	Qt 中的命令模式	206
14.1	Qt 的 Undo Framework	206
14.2	使用 Undo Framework 的一个例子	211
第 15 章	Qt 中的抽象工厂模式	215
15.1	抽象工厂模式简介	215
15.2	QTextCodec 及其子类的定义	217
15.3	界面风格	220
第 16 章	Qt 中的观察者模式	235
16.1	事件处理机制	236
16.2	事件滤波器	237
16.3	一个简单的例子——图像浏览器	239
16.4	一个有趣的例子——鼠标手势	240

第 17 章 Qt 的元对象系统	252
17.1 C++ RTTI (Run-time Type Information)	252
17.2 Qt 的元对象系统	258
第 18 章 智能指针	263
18.1 QPointer	263
18.2 QSharedPointer	275
参考文献	282
索引	284

第 1 章 案例的选择与评估	1
第 2 章 Qt 概述	11
代码段 2-1, 使用 Qt 进行主控台输入与输出, 取自 z:\examples\qt_console\main.cpp	19
第 3 章 类模板特化	24
代码段 3-1, 类模板 Stack, 引自 z:\examples\template_specialization	24
代码段 3-2, 完全特化的类模板 Stack, 引自 examples\template_specialization	25
代码段 3-3, 部分特化的类模板 Stack, 引自 examples\template_specialization	25
代码段 3-4, 使用 traits 技术封装 float 及 double 类型的特征, 取自 z:\examples\float_traits\main.cpp	27
代码段 3-5, 类型分类技术, 取自 z:\examples\type_classification\main.cpp	29
代码段 3-6, 应用类模板特化降低代码膨胀, 摘自 examples\reduce_code_bloat	30
代码段 3-7, 应用类模板特化降低代码膨胀, 摘自 z:\examples\reduce_code_bloat	31
第 4 章 标准库及 Qt 对字符串的处理	32
代码段 4-1, 类模板 basic_string 的构造函数	35
代码段 4-2, 类模板 basic_string 常用构造函数的使用, 取自 z:\examples\basic_string_demo\main.cpp	36
代码段 4-3, basic_string 对象和字符串的比较	36
代码段 4-4, 字符串的不同存放方式, 摘自 z:\examples\qstring_demo\main.cpp	39
代码段 4-5, QString 的字符编码转换功能, 摘自 z:\examples\qstring_merit\main.cpp	40
第 5 章 国际化与区域文化	41
代码段 5-1, 类模板 time_get 的使用方法, 取自 z:\examples\locale_time_get\main.cpp	47
代码段 5-2, 类模板 time_put 的成员函数 put 的用法, 取自 z:\examples\locale_time_ put\main.cpp	48
代码段 5-3, 类模板 codecvt 的成员函数 in 的功能, 取自 z:\examples\locale_codecvt\ main.cpp	50
代码段 5-4, 类 locale 以及 facet 的实现框架, 取自 VS 2010 安装目录 crt\src\locale	51
代码段 5-5, 类 locale::id 的作用	53
代码段 5-6, 创建新的 facet 子类, 取自 z:\examples\locale_unit\main.cpp	54

第 6 章 C++的 iostream	56
代码段 6-1, 应用类模板 char_traits 实现大小写不敏感的字符串类 ci_string, 取自 z:\examples\ci_string\ci_string\main.cpp	63
代码段 6-2, 对文件流进行读取以及写入操作, 取自 z:\examples\ fstream_demo\main.cpp	68
代码段 6-3, 具有多种格式设置的流, 取自 Z:\examples\share_streambuf	74
代码段 6-4, 流缓冲区的复制, z:\examples\copy_streambuf\main.cpp	75
代码段 6-5, 二进制文件的读取, 摘自 z:\examples\process_binary\main.cpp	77
代码段 6-6, 直接操作二进制文件对应的流缓冲区, 摘自 z:\examples\ process_binary\main.cpp	77
代码段 6-7, 用户自定义类型的输入和输出, 摘自 z:\examples\ overloaded_io\main.cpp	78
第 7 章 Qt 的流	79
代码段 7-1, 使用 QFileInfo 以及 QDir 获取各驱动器下的子目录名, 取自 z:\examples\QDir_demo\main.cpp	81
代码段 7-2, 使用 QFile 操作一个文件, 取自 z:\examples\QFile_read\main.cpp	81
代码段 7-3, 使用 QTextCodec 转换编码方案, 摘自 z:\examples\ QTextStream_demo\main.cpp	82
代码段 7-4, 使用 QDataStream 输出二进制数据, 摘自 z:\examples\ QDataStream_usage\main.cpp	84
代码段 7-5, 使用 QDataStream 读取二进制数据, 摘自 z:\examples\ QDataStream_usage\main.cpp	84
代码段 7-6, 重载运算符以使 QDataStream 支持新的数据类型, 取自 z:\examples\QDataStream_demo\main.cpp	85
代码段 7-7, QLocale 的使用, 摘自 z:\examples\qlocale_set\main.cpp	86
第 8 章 隐式共享与 d-pointer 技术	88
代码段 8-1, 采用隐式共享技术的 QString::toCaseFolded(), 取自 src\ corelib\tools\qstring.cpp	89
代码段 8-2, QString 的复制构造函数, 摘自 src\corelib\tools\qstring.h	90
代码段 8-3, 类 Matrix 的传统定义方式, 摘自 z:\examples\d_pointer\matrix\main.cpp	90
代码段 8-4, 应用 d-pointer 模式的类 Matrix, 摘自 z:\examples\d_pointer\matrix_with_d_pointer\main.cpp	91
代码段 8-5, Qt 4.5 版本中类 QLocale 的定义, 摘自 S:\corelib\tools\qlocale.h	93
代码段 8-6, Qt 4.5 版本中类 QLocalePrivate 的定义, 摘自 S:\corelib\tools\qlocale_p.h	93

代码段 8-7, 修改成员函数 toString()加入 Qt 库的版本信息, 摘自 S:\corelib\tools\qlocale.cpp	94
代码段 8-8, 使用类 QLocale 的 Qt 应用程序, 摘自 z:\examples\d_ pointer\test_QLocale\main.cpp	95
代码段 8-9, 实现 d-pointer 模式的一个例子	96
代码段 8-10, Qt 中与 d-pointer 模式相关的宏, 摘自 src\corelib\global\qglobal.h	97
代码段 8-11, 宏 Q_DECLARE_PRIVATE 展开后的结果	97
代码段 8-12, QObject 及 QObjectData 的定义, 摘自 S:\corelib\kernel\qobject.h	98
代码段 8-13, 类 QObject 及 QObjectPrivate 对 d-pointer 的使用	99
代码段 8-14, QWidget 继承了 QObject 的 d-pointer 模式	99
第 9 章 Qt 容器与迭代器	101
代码段 9-1, STL 风格以及 Java 风格的迭代器, 摘自 z:\examples\ QList_change_value\main.cpp	103
代码段 9-2, foreach 的使用格式, 取自 z:\examples\foreach_demo\main.cpp	104
代码段 9-3, 类模板 qLess, 摘自 src\corelib\tools\qalgorithms.h	106
代码段 9-4, 使用 QDataStream 保存/读取 QMap 对象, 摘自 z:\examples\english_pron\main.cpp	109
代码段 9-5, QListData 的成员函数 remove(), 摘自 src\corelib\tools\qlistdata.cpp	110
代码段 9-6, QTypeInfo 的定义, 摘自 src\corelib\global\qglobal.h	110
代码段 9-7, QList 的数据结构, 摘自 src\corelib\tools\qlist.h	111
代码段 9-8, 向 QList 中添加元素, 摘自 src\corelib\tools\qlist.h	111
代码段 9-9, 使用 qSort 对容器排序, 摘自 z:\examples\qtl_related_words\main.cpp	113
代码段 9-10, 函子 indirectCompare, 摘自 z:\examples\qtl_thesaurus\main.cpp	113
第 10 章 多线程与可重入	115
代码段 10-1, 在 Qt 中创建多线程, 摘自 Z:\examples\simple_thread\main.cpp	116
代码段 10-2, 互斥体 QMutex 的使用, 取自 Z:\examples\qmutex\main.cpp	117
代码段 10-3, 使用互斥体的一个简单方法	118
代码段 10-4, 用信号量来管理循环缓冲区, 摘自 z:\examples\qsemaphore\main.cpp	119
代码段 10-5, 使用条件量管理循环缓冲区, 摘自 z:\examples\qwaitcondition\main.cpp	120
代码段 10-6, 以传统方式实现 singleton 模式	122
代码段 10-7, 在堆中创建全局对象	123
代码段 10-8, 简化的成员函数 instance()	124
代码段 10-9, 通过静态局部对象来定义 singleton 对象	124
代码段 10-10, 静态局部对象的初始化	125

代码段 10-11, C++的判断/赋值操作无法锁定共享资源	125
代码段 10-12, QBasicAtomicPointer 的定义, 摘自 S:\corelib\thread\qbasicatomic.h	126
代码段 10-13, QBasicAtomicPointer 在 Windows/Intel 平台上的实现, 摘自 S:\corelib\arch\qatomic_windows.h	126
代码段 10-14, 类模板 QGlobalStatic 的定义, 摘自 S:\corelib\global\qglobal.h	129
代码段 10-15, 类模板 QGlobalStaticDeleter 的定义, 摘自 S:\corelib\global\qglobal.h	129
代码段 10-16, 宏 Q_GLOBAL_STATIC 的定义, 摘自 S:\corelib\global\qglobal.h	130
代码段 10-17, 单线程环境下宏 Q_GLOBAL_STATIC 的定义, 取自 S:\corelib\global\qglobal.h	131
代码段 10-18, 直接返回一个指向 singleton 对象的指针	132
代码段 10-19, 使用一个已经析构的 singleton 对象, 摘自 z:\examples\use_destructed_singleton\main.cpp	132
第 11 章 信号与槽 (Signals and Slots)	134
代码段 11-1, QObject 对象的定义顺序, 摘自 z:\examples\ QObject_destruction_order\main.cpp	135
代码段 11-2, 信号与槽的定义, 摘自 Z:\examples\signals_slots_ demo\signals_slots_declare.h	137
代码段 11-3, 信号与槽的绑定, 摘自 z:\examples\signals_slots_demo\main.cpp	137
代码段 11-4, 类 FindDialog 的定义, 摘自 z:\examples\find_dialog\find_dialog.h	140
代码段 11-5, 类 FindDialog 的构造函数, 取自 z:\examples\find_dialog\find_dialog.cpp	141
代码段 11-6, 类 FindDialog 的构造函数 (续), 摘自 z:\examples\find_dialog\find_dialog.cpp	142
代码段 11-7, 类 FindDialog 的槽函数及析构函数, 摘自 z:\examples\find_dialog\find_dialog.cpp	143
第 12 章 Graphics/View 框架	145
代码段 12-1, 类 Mouse 的定义, 取自 z:\examples\collidingmice\mouse.h	147
代码段 12-2, 类 Mouse 的部分成员函数, 取自 z:\examples\collidingmice\mouse.cpp	148
代码段 12-3, 类 Mouse 的成员函数 advance(), 取自 z:\examples\collidingmice\mouse.cpp	149
代码段 12-4, 项目 collidingmice 的主函数, 取自 z:\examples\collidingmice\main.cpp	150

第 13 章 Model/View 框架..... 152

代码段 13-1, 类 TreeModel 的声明, 取自 z:\examples\mvc\binary_tree\treemodel.h.....	166
代码段 13-2, 类 TreeModel 的实现, 取自 z:\examples\mvc\binary_tree\treemodel.cpp.....	166
代码段 13-3, 类 TreeModel 的实现 (续), 取自 z:\examples\mvc\binary_tree\treemodel.cpp.....	167
代码段 13-4, 满二叉树例子的主函数, 取自 z:\examples\mvc\binary_tree\main.cpp.....	168
代码段 13-5, 能够处理更多角色的模型类, 取自 z:\examples\mvc\binary_tree_more_role\treemodel.cpp.....	169
代码段 13-6, 显示自身发生变化的数据项, 取自 z:\examples\mvc\binary_tree_changing_data\treemodel.cpp.....	170
代码段 13-7, 更改数据集的标头, 取自 z:\examples\mvc\ binary_tree_header\treemodel.cpp.....	172
代码段 13-8, 编辑满二叉树的叶节点, 取自 z:\examples\mvc\binary_ tree_editable\treemodel.cpp.....	173
代码段 13-9, 重载 QAbstractListModel 的虚函数以显示、编辑一个列表, 取自 z:\examples\mvc\QAbstractListModel_demo\ListModel.cpp.....	174
代码段 13-10, 使用 QStandardItemModel 处理列表, 取自 z:\examples\mvc\QStandardItemModel_demo\main.cpp.....	176
代码段 13-11, 使用 QStandardItemModel 处理表格, 取自 z:\examples\mvc\QStandardItemModel_demo\main.cpp.....	176
代码段 13-12, 使用 QStandardItemModel 处理树, 取自 z:\examples\mvc\QStandardItemModel_demo\main.cpp.....	177
代码段 13-13, 类 QStringListModel 的使用, 取自 z:\examples\mvc\QStringListModel_demo\main.cpp.....	178
代码段 13-14, 便利模型类 QFileSystemModel 的用法, 取自 z:\examples\mvc\file_system\main.cpp.....	179
代码段 13-15, 用 QColumnView 对象显示本地文件系统, 取自 z:\examples\mvc\QColumnView_demo\main.cpp.....	180
代码段 13-16, 类 MainWindow 的声明, 取自 z:\examples\mvc\selection_monitoring\mainwindow.h.....	183
代码段 13-17, 类 MainWindow 实现, 取自 z:\examples\mvc\selection_monitoring\mainwindow.cpp.....	183
代码段 13-18, 同步两个视图对象中的选择信息, 取自 z:\examples\mvc\sync_selection\main.cpp.....	185
代码段 13-19, 例子 SpinBox 的主函数, 取自 z:\examples\mvc\ spinboxdelegate\main.cpp.....	187

代码段 13-20, 类 SpinBoxDelegate 的实现, 取自 z:\examples\mvc\ spinboxdelegate\delegate.cpp	187
代码段 13-21, 代理模型索引的创建, 取自 z:\examples\mvc\revertProxyModel\revertProxyModel.cpp	192
代码段 13-22, 代理模型 RevertProxyModel 的其他 2 个接口函数, 取自 z:\examples\mvc\revertProxyModel\revertProxyModel.h	193
代码段 13-23, 代理模型 RevertProxyModel 的接口函数 parent(), 取自 z:\examples\mvc\revertProxyModel\revertProxyModel.cpp	194
代码段 13-24, 接口函数 data()的实现, 取自 q:\src\gui\itemviews\ qabstractproxymodel.cpp	194
代码段 13-25, 创建源模型, 取自 z:\examples\mvc\basicsortfiltermodel\main.cpp	197
代码段 13-26, 类 Window 的构造函数, 取自 z:\examples\mvc\ basicsortfiltermodel>window.cpp	197
代码段 13-27, 令代理模型指向源模型, 取自 z:\examples\mvc\ basicsortfiltermodel>window.cpp	198
代码段 13-28, 代理模型对源模型数据项的过滤、排序, 取自 z:\examples\mvc\basicsortfiltermodel>window.cpp	198
代码段 13-29, 向 QListWidget 中添加数据项, 取自 z:\examples\mvc\QListWidget_demo\main.cpp	200
代码段 13-30, 新闻的表示, z:\examples\mvc\item_roles\NewsDialog.cpp	200
代码段 13-31, 设置 QListWidgetItem 所表示数据项中的数据子项, 取自 z:\examples\mvc\item_roles\NewsDialog.cpp	201
代码段 13-32, GDP 数据的表示, 取自 z:\examples\mvc\ QTableWidget_demo\main.cpp	202
代码段 13-33, 类 QTableWidget 的使用, 取自 z:\examples\mvc\ QTableWidget_demo\main.cpp	203
代码段 13-34, 书籍目录的表示, 取自 z:\examples\mvc\ QTreeWidget_demo\main.cpp	204
代码段 13-35, 构建 QTreeWidget 中的树状模型, 取自 z:\examples\mvc\QTreeWidget_demo\main.cpp	205
第 14 章 Qt 中的命令模式	206
代码段 14-1, QUndoCommand 的部分定义	208
代码段 14-2, 类 QUndoCommand 部分成员函数的实现, 摘自 S:\gui\util\qundostack.cpp	209
代码段 14-3, QUndoStack 基本功能部分的定义, 取自 S:\gui\util\qundostack.h	210
代码段 14-4, QUndoStackPrivate 的部分定义, 取自 S:\gui\util\qundostack_p.h	211

代码段 14-5, 类 MoveCommand 的定义, 取自 z:\examples\undoframework\commands.h	212
代码段 14-6, 类 MoveCommand 的实现, 取自 z:\examples\undoframework\commands.cpp	213
第 15 章 Qt 中的抽象工厂模式	215
代码段 15-1, 依据全局变量创建不同风格的界面元素	216
代码段 15-2, 抽象工厂模式的使用	217
代码段 15-3, Latin1 到 Unicode 的转换, 取自 z:\examples\factory_pattern\main.cpp	217
代码段 15-4, 类 QTextCodec 的部分定义, 取自 S:\corelib\codecs\qtextcodec.h	218
代码段 15-5, QTextCodec 部分成员函数的实现, 取自 S:\corelib\codecs\qtextcodec.cpp	219
代码段 15-6, 类 QLatin1Codec 的定义与实现	219
代码段 15-7, 类 WidgetGallery 的成员函数 changeStyle(), 取自 z:\examples\styles\widgetgallery.cpp	223
代码段 15-8, 圆角矩形绘制路径的绘制, 取自 z:\examples\ styles\norwegianwoodstyle.cpp	225
代码段 15-9, 基本元素的绘制, 取自 z:\examples\styles\norwegianwoodstyle.cpp	226
代码段 15-10, 基本元素的绘制 (续), 取自 z:\examples\styles\ norwegianwoodstyle.cpp	228
代码段 15-11, NorwegianWoodStyle 的成员函数 drawControl, 取自 z:\examples\styles\norwegianwoodstyle.cpp	229
代码段 15-12, 更改控件属性的成员函数 polish, 取自 z:\examples\ styles\norwegianwoodstyle.cpp	231
代码段 15-13, 更改控件尺寸的成员函数 pixelMetric, 取自 z:\examples\styles\norwegianwoodstyle.cpp	231
代码段 15-14, 设置与风格相关的一些属性, 取自 z:\examples\ styles\norwegianwoodstyle.cpp	232
代码段 15-15, 更改应用程序调色板的成员函数 polish, 取自 z:\examples\styles\norwegianwoodstyle.cpp	232
代码段 15-16, 设置填充图像的私有成员函数 setTexture, 取自 z:\examples\styles\norwegianwoodstyle.cpp	233
第 16 章 Qt 中的观察者模式	235
代码段 16-1, 设置 QScrollArea 为另一控件的观察者, 取自 Q:\src\gui\widgets\qscrollarea.cpp	238
代码段 16-2, 将一个 QScrollArea 对象设置为一个 QLabel 对象的观察者, 取自 z:\examples\imageviewer\imageviewer.cpp	239

代码段 16-3, 类 ImageViewer 中更改图像显示比例的函数, 取自 z:\examples\imageviewer\imageviewer.cpp	240
代码段 16-4, 鼠标手势及其回调函数的定义, 取自 Z:\examples\mouse_gesture\mousegesturerecognizer.h	244
代码段 16-5, 类 MouseGestureRecognizer 的定义, 取自 Z:\examples\mouse_gesture\mousegesturerecognizer.h	245
代码段 16-6, 对鼠标轨迹进行识别的核心算法, 取自 Z:\examples\mouse_gesture\mousegesturerecognizer.cpp	246
代码段 16-7, 类 MouseGesture 的定义, 取自 Z:\examples\mouse_ gesture\MouseGesture.h	247
代码段 16-8, 类 GestureCallbackToSignal 的定义, 取自 Z:\examples\mouse_gesture\MouseGestureFilter.cpp	247
代码段 16-9, 类 MouseGestureFilter 的定义	248
代码段 16-10, 鼠标手势的添加与删除, 取自 Z:\examples\mouse_gesture\MouseGestureFilter.cpp	249
代码段 16-11, 事件滤波器及相关函数, 取自 Z:\examples\mouse_gesture\MouseGestureFilter.cpp	249
代码段 16-12, 类 MainWindow 的定义, 取自 Z:\examples\mouse_ gesture\mainwindow.h	250
代码段 16-13, 鼠标手势例子的主函数, 取自 Z:\examples\mouse_gesture\main.cpp	251
第 17 章 Qt 的元对象系统	252
代码段 17-1, 类 type_info 的声明	254
代码段 17-2, typeid 的操作数可以为基本类型、非多态类及多态类, 取自 z:\examples\typeid\main.cpp	255
代码段 17-3, 判断一个 QObject 派生类的对象是否“具有”某个类型, 取自 z:\examples\QMetaObject_demo\main.cpp	260
代码段 17-4, 获取 QObject 派生类对象的类型信息, 取自 z:\examples\QMetaObject_demo1\main.cpp	260
代码段 17-5, 获取 QObject 派生类对象的数据, 取自 z:\examples\introspect_ qobject\main.cpp	261
代码段 17-6, QVariant 的使用, 取自 z:\examples\qvariant\main.cpp	262
代码段 17-7, QVariant 支持二进制输入/输出, 摘自 z:\examples\qvariant\main.cpp	262
第 18 章 智能指针	263
代码段 18-1, QPointer 的功能, 取自 z:\examples\qpointer_demo\main.cpp	264
代码段 18-2, 对一个对象施加 delete 运算符, 取自 z:\examples\delete_ object\main.cpp	264

代码段 18-3, 具有淡入显示效果的类 <code>FaderWidget</code>	266
代码段 18-4, 使用 <code>QPointer</code> 来判断一个 <code>QFaderWidget</code> 控件是否存在	267
代码段 18-5, <code>QWidget</code> 对 <code>QPointer</code> 的使用, 取自 <code>S:\gui\kernel\qwidget.cpp</code>	269
代码段 18-6, 遍历 <code>QMultiHash</code> 中具有相同关键字的元素, 取自 <code>z:\examples\QMultiHash_demo\main.cpp</code>	271
代码段 18-7, 使用信号量 <code>QReadWriteLock</code> 锁定某个资源以进行写入操作	271
代码段 18-8, 使用信号量 <code>QWriteLock</code> 锁定某个资源以进行写入操作	272
代码段 18-9, 与 <code>QPointer</code> 相关的类型与函数, 取自 <code>S:\corelib\kernel\qobject.cpp</code>	273
代码段 18-10, <code>QPointer</code> 的定义, 摘自 <code>S:\corelib\kernel\qpointer.h</code>	273
代码段 18-11, <code>QMetaObject</code> 中的相关代码, 取自 <code>S:\corelib\kernel\qobject.cpp</code>	274
代码段 18-12, <code>QObject</code> 析构函数中与 <code>QPointer</code> 相关的代码, 摘自 <code>S:\corelib\kernel\qobject.cpp</code>	275
代码段 18-13, 使用类模板 <code>QSharedDataPointer</code> 实现隐式共享, 取自 <code>z:\examples\QShareDataPointer_demo\main.cpp</code>	276
代码段 18-14, 关于常量型成员函数的约定, 取自 <code>z:\examples\select_ const\main.cpp</code>	278
代码段 18-15, 类 <code>QSharedData</code> 的定义, 取自 <code>Q:\src\corelib\tools\qshareddata.h</code>	279
代码段 18-16, 类模板 <code>QSharedDataPointer</code> 的定义 (待续), 取自 <code>Q:\src\corelib\tools\qshareddata.h</code>	279
代码段 18-17, 类模板 <code>QSharedDataPointer</code> 的定义 (待续), 取自 <code>Q:\src\corelib\tools\qshareddata.h</code>	280
代码段 18-18, 类模板 <code>QSharedDataPointer</code> 的定义 (续), 取自 <code>Q:\src\corelib\tools\qshareddata.h</code>	280
参考文献	282
索引	284

案例的选择与评估

C++语言的学习者往往希望有一个源代码公开、质量上乘的 C++项目作为学习的范本来临摹。这种案例最理想的来源是知名软件企业使用 C++开发的产品，比如微软公司开发的 Windows 操作系统、Office 办公组件，苹果公司开发的 MacOS 操作系统，Google 公司开发的 Chrome OS 操作系统或者 Adobe 公司的 Photoshop。但显然，这些公司出于商业利益不会公开这些产品的源代码。幸运的是，软件开源运动为 C++学习者提供了成千上万个可供借鉴的项目，我们可以在这些项目中遴选出质量高、文档完整、适合 C++学习者的案例。

由于 C++开源项目数量众多，我们分两个步骤进行遴选。

(1) 初步筛选阶段，我们主要依据一个开源项目受关注的程度、该项目是否涉及太多的专业知识来选择。这个阶段我们选择了 8 个开源项目。

(2) 定量评估阶段，我们使用 CppDepend 工具，评估每个开源项目的代码规模以及代码质量，最终选择了 Qt。本章 1.1、1.2 节详细描述这个过程，读者可以借鉴其中的方法选择其他 C++案例，或者在学习其他编程语言时，使用其中的方法选择对应的案例。而且，读者还可以使用这两节讨论的工具 CppDepend 剖析其他软件的结构与质量。

C++开源社区还有其他优秀的项目，1.3 节简要介绍了其中的 Boost 库以及 KDE 开发框架。Boost 库大量使用了模板技术，某些技术对 C++初学者来说过于深奥。而 KDE 是在 Qt 基础上构建的一个开发框架。在掌握本书的内容后，读者可以继续研读 KDE 或者 Boost，以进一步提升 C++语言的应用能力。

本章 1.4 节介绍本书对术语等的约定，1.5 节专门讨论在 UML 类图方面的约定。这两节和本章主旨没有关联，但是将它们放在书的前言部分又显得太长，因而只好将它们放在此处。

1.1 案例的初步选择

我们从以下网站搜索开源项目：

(1) Sourceforge (sourceforge.net)，这是最著名、历史最悠久、规模最大的开源项目管理网站。

(2) Google code (code.google.com)，它的访问速度快，是开源项目管理网站的后起之秀。

(3) C++创始人 Bjarne Stroustrup 的个人网站 www2.research.att.com/~bs/applications.html，其中罗列了一些优秀的开源 C++项目。

(4) 开源中国社区 (www.oschina.net/project/lang/21/c)。

世界上使用 C++ 编写的开源项目很多,比如截至 2011 年 12 月仅 Sourceforge 上就有 6450 个 C++ 项目。如何在这么多的项目选择一个优秀案例呢? 我们分两个步骤来选择:

第一, 初步筛选。依据一些定性的指标, 比如一个开源项目被用户关注的程度、该项目是否涉及太多的专业领域知识等, 选择少量项目。

第二, 定量评估。依据代码规模、注释内容比例、类的内聚性等指标, 对各个项目进行定量地评估, 选择最优者作为本书剖析的对象。

初步筛选阶段, 我们采用了以下指标。

① **关注度。**一个开源项目的关注度高, 说明该软件实用、在同类软件中性能更好。由于得到用户的关注, 该项目的开发小组就能够持之以恒地对软件进行改进, 该项目的源代码质量就会更高。我们依据每周用户下载次数对开源项目进行排序, 忽略那些下载次数少的项目。

② **涉及其他专业领域的程度。**由于 C++ 课程往往是软件专业学生的基础课程, 当他们学习 C++ 语言时, 对其他专业领域了解不多。如果将那些涉及较深专业知识的项目 (比如电路板自动布线系统、编译系统等) 选为案例, 学生将花费很多时间去学习和 C++ 不相关的知识, 这将降低学习效率。

依据以上指标, 我们选择了 8 个 C++ 开源项目。读者可从前文所述的开源网站下载这些项目的源代码以及安装程序, 然后将这些项目安装在自己的机器上, 体验它们的功能。以下是除了 Qt 以外的其他 7 个项目的简要介绍。

Celestia。以三维方式显示宇宙间 10 万颗星座的位置、形状信息。对于那些已有影像数据的星球, 该软件能显示这些星球的表面图像, 令用户感觉似乎是驾驶一艘宇宙飞船在星际间畅游。自 2001 年该软件被免费发布以来, 已有 3 百万次下载, 并被广泛地用在家庭、学校、政府机关等场所。该软件的官方网站为 www.shatters.net。安装该软件之后, 读者可以执行“帮助/系统演示”命令, 体验星际旅游的乐趣。

K3DSurf。能够以非常漂亮的三维形式绘制多元数学函数的表面。

WinDirStat。能够计算并显示本地文件系统中各个子目录的字节数, 也能够依据文件的类型 (比如临时文件、MP3 文件等) 对文件进行分类, 并显示每个类别的字节数。计算机用户可以使用该软件察看各子目录、各种类型文件占用磁盘空间的情况, 删除那些不再使用的文件, 以释放出更多的磁盘空间。

Source-Navigator。能够分析 C、C++、Java 等语言的源程序中各种部件 (如函数、类) 之间的关系, 并以图形化的方式显示这些关系, 还允许用户查询各个部件之间的关系, 比如某个函数调用了哪些函数, 该函数又被哪些函数调用。开发者可以使用该软件来查看、理解一个复杂软件的总体结构。

Notepad++。是运行在 Windows 操作系统之上的一款文本、源代码编辑软件。除了具有同类软件的常见功能之外，它支持 Unicode，允许用户使用鼠标滚轮即可放大/缩小字体，支持多达 52 种编程语言、脚本语言以及标记语言。

WinMerge。能够比较两个文本文件的差异，也能比较两个文件夹中哪些文件存在差异。程序开发者可以使用该软件比较一个源程序的多个版本，以快速澄清各个版本的差异。

CppCheck。能够检测 C++ 程序中的逻辑错误（而不是语法错误），比如，一个类在重载“=”运算符时，应该返回 this 指针所指对象的引用。

1.2 案例的定量评估

我们依据以下指标对 8 个 C++ 开源项目进行量化评估。

1. 学生兴趣。从教育心理学角度，如果学习者对一个开源项目感兴趣，他就会积极主动地探询 C++ 的语言特性是如何被应用在他所关注的项目中的，这可以大幅提高学习效率。为了得到定量的数据，我们对南开大学软件学院 114 名一年级本科生进行了问卷调查。具体地说，我们演示了这 8 个开源项目的功能，让每个学生独立地填写一个调查表。每个学生对一个项目感兴趣的程度用数字 1~5 表示，5 表示最感兴趣，1 表示最不感兴趣，感兴趣程度从 1 到 5 逐渐过渡。最后我们计算所有学生对于一个项目兴趣程度的平均值。

2. 代码规模。规模太大的开源项目将大幅增加学习者的学习难度和学习周期，规模太小的项目将无法展示 C++ 语言解决复杂问题的能力，我们需要在两者之间折中。我们依据源代码行数（Line of Code, LOC）以及源代码中的类型数量来评测一个项目的规模。

3. 代码质量。精确地评估一个软件系统的代码质量是比较困难的，这需要专家仔细阅读软件的代码以及文档，评估其设计是否精良，是否具有良好的可扩展性、可移植性，源代码是否严格遵循某种编码规范，评估其运行时的性能、健壮程度等。8 个 C++ 开源项目的代码总量约为 53 万行，进行人工评估显然是不现实的。我们选择以下两个可以量化的指标来评估一个项目的代码质量。

（1）代码中注释部分的比例。适当比例的注释可以提高代码的可读性，同时也能体现编程人员的仔细与严谨。

（2）内聚性（cohesion）。所谓内聚性，是指一个类的成员变量和成员函数之间的耦合程度。较高的内聚性往往意味着较高的代码质量。有多种度量内聚性的方法，我们采用 LCOM-HS（Lack of Cohesion of Methods，提出者为 Henderson-Sellers）度量，其取值范围为 0~2，越大表示内聚性越差。

4. C++ 特性的应用。有的开源项目虽然宣称是使用 C++ 语言开发的，但是大部分代码是 C 语言编写的，只用到“封装”这样简单的 C++ 特性。为了评判一个开源项目是否大量使用了 C++ 特性，我们选择了以下三个指标。

(1) 名字空间 (namespace) 的个数。在一个中、大型软件项目中, 合理使用名字空间可以有效避免名字冲突, 提高软件系统的模块化程度。

(2) 继承的个数。类的继承是面向对象编程思想的典型特征, 是实现多态性的必要条件。

(3) 模板 (template) 的个数。除了使用面向对象思想, 现代 C++项目还大量使用模板技术, 以实现泛型编程 (generic programming) 的思想。

以下工具可以定量地评估一个 C++项目的规模与质量。

(1) SourceAudit (www.frontendart.com), 由 FrontEndART 公司开发, 该工具甚至还可以分析出 C++程序中是否应用了设计模式。

(2) Telelogic 公司的 logiscope, 对一个软件系统的可维护性、可重用性、可测试性、可读性等进行评估。该公司于 2008 年被 IBM 收购, 该产品演化为 Rational Software Analyzer。

(3) CppDepend (www.cppdepend.com), 能够对 C++程序进行 60 多个指标的测量, 其中有些是关于代码结构的 (如类及名字空间的数量), 有些是关于代码质量的 (比如程序注释比例, 内聚性, 项目稳定度等)。该工具还可以直观地显示程序模块、类、函数之间的依赖性。它将被分析的源代码当作数据库来处理, 允许用户使用一种代码查询语言 (Code Query Language, CQL) 对源代码做各种分析。

由于 CppDepend 小巧 (约 8.6M 字节)、灵活 (支持代码查询语言)、被允许在学术机构中免费使用, 我们选择了该工具。它生成的分析报告只含有部分评估指标, 我们应该使用代码查询语言获得其他评估指标。关于该软件的使用 (尤其是代码查询语言的规范), 请参考其官方网站, 本文不再赘述。

8 个 C++开源项目的定量化评估结果如表 1-1 所示。由于整个 Qt 库的规模太大, 我们仅选择了其核心模块 QtCore 以及 QtGui 作为分析对象。表中, LOC 表示代码行数 (Line Of Code), Types 表示类型的数量, Comm 表示注释行与总代码行的比值, LCOM 表示内聚性较差的类在所有类型中的比例, NSpace 表示名字空间的数量, Templates 和 Inherits 分别表示模板与继承的数量。

表 1-1 8 个 C++开源项目的定量化评估结果

序号	软件名称		代码规模		代码质量		C++技术的应用			兴趣
			LOC	Types	Comm	LCOM	NSpace	Templates	Inherits	
1	Qt	QtCore	39,812	838	49%	0.8%	10	278	219	4.2
		QtGui	181,066	1,714	36%	2%	11	14	548	
		合计	220,878	2552	43%	1.8%	21	292	767	
2	K3DSurf		14,026	28	20%	25%	2	0	0	3.7
3	Notepad++		68,849	415	13%	2.4%	1	2	115	3.4
4	Celestia		49,777	628	13%	0.8%	7	31	222	3.2

续表

序号	软件名称	代码规模		代码质量		C++技术的应用			兴趣
		LOC	Types	Comm	LCOM	NSpace	Templates	Inherits	
5	CppCheck	19,280	127	20%	0.8%	31	0	51	3.2
6	SourceNav	77,502	655	16%	0.2%	1	0	115	2.9
7	WinDirStat	7,259	111	12%	2.7%	17	2	0	2.6
8	Winmerge	69,449	582	26%	1%	20	14	57	2.3

学生们最感兴趣的是 Qt，这归因于该软件包的示例程序所展现的强大的图形/图像处理能力。其次，学生们感兴趣的是 K3DSurf，这归因于该软件所展现的精美的数学函数曲面。Qt、Celestia 和 Winmerge 都大量使用了名字空间、模板以及继承，表明它们都能成为 C++ 语言的案例。由于学生们对 Winmerge 不感兴趣，我们应该在 Qt 和 Celestia 两者中选择一个。考虑到 Qt 不但可以作为 C++ 语言的案例，同时也是一款功能强大的跨平台 C++ 类库，研究该类库不但可以学习 C++ 语言特性，还可以使用它来开发桌面应用程序，因此，我们最终选择了 Qt。

1.3 其他案例

除了 Qt 之外，以下 C++ 项目也可作为学习 C++ 语言的优秀案例。

Boost。由 80 多个开源的 C++ 子库组成。这些子库所针对的应用领域很广，即涉及通用领域（比如智能指针子库），也涉及众多的具体领域（比如封装不同操作系统文件系统差异的 FileSystem 子库）。出于性能、灵活性方面的考虑，该库大量使用了模板技术。

以下是该库中功能较强的一些子库：

（1）文本处理方面，处理正则表达式的 Regex，处理 Unicode 编码的 Locale。

（2）容器/数据结构方面，表示并操作循环缓冲区的 Circular Buffer，表示并处理多个二进制位的 Dynamic Bitset，表示各种类型图像数据的 GIL（Generic Image Library），表示并处理图的 Graph。

（3）并行处理方面，支持多线程的 Thread，支持分布式并行处理的 MPI（Message Passing Interface），支持线程间通信与同步的 Interprocess。

（4）数学方面，表达并处理几何图形的 Geometry，处理基本线形代数问题的 uBLAS（Basic Linear Algebra Library）。

（5）其他方面，对程序进行调试和单元测试的 Test，允许 C++ 和 Python 交互操作的 Python 子库。

在开源网站 SourceForge 上，Boost 库每发布一个新版本，就会有 10 万次下载。一些著名的商业软件使用了该库，比如绘图工具 Adobe Photoshop CS2，Adobe Indesign，反病毒软件 McAfee Managed VirusScan 等。Boost 库中的一些子库也被纳入了 C++ 11 标准。

Boost 库具有高质量的源代码。Herb Sutter 以及 Andrei Alexandrescu 在 *C++ Coding Standards* ^[1]中评价该库为“世界上设计最精良、质量最优秀的 C++库之一”。Scott Meyers 在 *Effective C++* ^[2]中将掌握 Boost 库作为 C++编程准则之一，也即“第 55 项：熟悉 Boost”。

KDE。从终端用户角度看，KDE（Kool Desktop Environment）是一个能够运行在 UNIX/Linux 等平台上的一个桌面环境。它具有漂亮的外观，用户能够像使用本地文件一样使用网络上的文件，支持 60 多种自然语言，还包含了诸多应用程序（比如办公应用套件 KOffice）。而从开发者角度看，KDE 是一个构建在 Qt 基础之上、提供了更多功能的应用程序框架。该框架的源代码行数多达 6 百万行（不包括 Qt），具有良好的代码质量。

1.4 基本约定

书中的源代码。除了直接引用 Qt 的源代码，本书还创建了一些独立的 Qt 应用程序作为例子。读者可以在随书光盘找到这些例子的完整代码。在本书正文中，为简明起见，我们往往只给出这些例子的主要代码，省略了那些与被讨论话题关系不密切的部分。对于预处理命令，我们只是简单地省略；对于其他语句，我们用省略号“……”表示省略。另外，Qt 的源代码使用了较多的预处理命令（比如 `Q_CORE_EXPORT`）。如果这些命令和被讨论话题没有关联，正文将省略它们。有些情况下，我们会将 Qt 源代码中的一些宏替换为它们所表示的具体内容，以增加程序的可读性。例如，在本书对应的开发环境中，宏 `Q_INLINE_TEMPLATE` 表示 `inline`。本书正文将采用后者，使程序更易读。

将目录映射为盘符。本书假设 Qt 被安装在 Windows 的 `d:\qtsdk` 目录下。供 Visual Studio 2010 使用的二进制库、头文件等存放在目录 `d:\qtsdk\desktop\qt\4.8.1\msvc2010` 下。由于本书多个地方需要访问该目录，为方便起见，我们使用 Windows 的命令

```
subst d:\qt\vc q:
```

将这个目录映射为 Q 盘。另外，本书使用盘符“Z:”表示随书光盘的位置。您可以将随书光盘的内容复制到您的本地文件系统中的某个文件夹中，使用 `subst` 命令将其映射为“Z:”盘。类似地，您也可以使用该命令将您计算机上的 Qt 安装目录映射为 Q 盘。这样，本书所有对“Q:”以及“Z:”的引用也会适用于您的计算机。

交互操作的表示。本书会讨论对 Windows 操作系统或者 Visual Studio 2008 开发环境的具体操作。有些教科书会将操作过程中屏幕上出现的菜单、对话框等插入正文，指导读者如何进行操作。这种方式虽然直观，但会占用太多的篇幅。本书采用更加简洁的文字描述方式。以修改 Windows 操作系统的环境变量为例，本书的描述为“显示桌面\我的电脑\右键\属性\高级\环境变量\系统变量\PATH”，表示单击 Windows 操作系统的“显示桌面”按钮，找到图标“我的电脑”，按鼠标右键，在弹出的菜单中选择“属性”项，在弹出的对话框中单击“高级”标签，单击“环境变量”按钮，在弹出的对话框中的“系统变量”部分，找到“PATH”进行修改。

支持的平台。本书使用 Windows XP/Visual Studio 2008 编写所有的例子，尚未在其他平台上测试它们。

函数名。正文引用一个函数名时，会在其后加上“()”，比如 `printf()`，以将其和其他标识符（比如类名）区分开来。函数参数全部被省略，不出现在括号中。

基类的对象。设 B 是一个基类，D 是它的派生类。除非特别说明，“B 的对象”既可以指 B 这个类本身的对象，也可以指派生类 D 的对象。即使 B 是一个抽象基类（因而不可能定义 B 本身的对象），我们也使用这个术语来表示其派生类的对象。

术语。同一个英文术语在不同的中文文献中会有不同的翻译，本书对常见术语的翻译如表 1-2 所示。某些英文术语在中文文献中很少出现（比如 `trait`），本书将直接使用这些英文术语，而不是试图给出一个难以被广泛接受的翻译。

表 1-2 英文/中文术语对照表

英文	中文	英文	中文
base class	基类	function template	函数模板
blocked thread	被阻塞线程	garbage collection	垃圾回收
class template	类模板 ^①	implicit sharing	隐式共享
concurrency	并发	inheritance	继承
constructor	构造函数	initialization	初始化
container	容器	instantiate	实例化
copy constructor	复制构造函数	iterator	迭代器
copy-on-write	写时复制	layout	布局
data member	数据成员	namespace	名字空间
derived class	派生类	operator	运算符
destructor	析构函数	preprocessing directives	预处理命令
disabled	禁用	reentrant	可重入
enabled	使能	reference	引用
factor (function object)	函子（函数对象） ^②	template specialization	类模板特化
friend	友元	thread	线程
function overloading	函数重载	type safe	类型安全
function signature	函数原型	wide character	宽字符

注①：本书将带有模板参数的类（比如 C++ 标准模板库中的 `vector`）称为类模板（class template）。我们不使用术语“模板类（template class）”，因为有的文献使用这个术语来表示本书中的类模板，而有的文献则使用这个术语来表示一个类模板实例化后生成的一个具体类（比如 `vector<double>`）。类似地，我们将带有模板参数的函数（比如 C++ 标准库中的 `sort`）称为函数模板（function template），不使用模板函数（template function）这样的术语。

注②：有的文献将函子（factor）称为函数对象（function object）。考虑到 factor 实际上是一个类而不是一个类的对象，本书不采用函数对象的称谓。

1.5 关于类图的约定

本书使用 UML (Unified Modeling Language) 描述类之间的关系。虽然市面上多款 UML 建模与绘制工具都宣称它们遵循 UML 规范,但是这些工具绘制出来的类图存在着细微的差别。本书采用 MagicDraw 绘制所有类图,下面我们以一个框图编辑器为例介绍该软件的一些约定。

如图 1-1 所示,类 `Diagram` 表示一个框图,类 `Shape` 表示抽象意义的图形元素,类 `Circle` 和 `Rectangle` 表示圆形、矩形的图形元素,它们是 `Shape` 的子类。UML 使用尾部位三角形的箭头来表示类之间的继承关系。

一个类被表示为一个最多具有 3 行的矩形框,第 1 行是类的名字,第 2 行罗列该类的数据成员,第 3 行罗列该类的成员函数,第 2 行、第 3 行可以被单独或者全部隐藏起来。每个数据成员(或者成员函数)前的“-”表示该成员是该类的私有成员(private),而“+”表示公有成员(public),“#”表示受保护成员(protected)。在数据成员那一行,冒号左边的是数据成员的名字,右边的是数据成员的类型,这部分可被省略。例如,类 `Diagram` 含有数据成员 `elements`,表示一个框图含有哪些图形元素。它还含有数据成员 `fileName`,表示以什么文件名保存该框图的信息。以上两个数据成员的类型分别为 `vector<Shape>` 以及 `string`。

图 1-1 使用 MagicDraw 绘制的类图

如果一个成员函数为纯虚函数,它的名字被显示为斜体,对应的类成为抽象类,类名也被显示为斜体。例如,类 `Shape` 表示抽象意义上的几何图形,其成员函数 `display()` 无法显示一个抽象的几何图形,因而被定义为一个纯虚函数。在 UML 类图中,它们的名字都被显示为斜体。

图 1-1 中,一端为实心菱形的箭头表示类之间的 Composition 关系。这种关系表示一个类的对象由另外一个类的对象组成。简单地说,就是表示“部分”与“全部”的关系。本例中的这个箭头表示一个框图(类 `Diagram` 的对象)“包含”多个几何图形(类 `Shape` 的对象,但实际上是 `Circle` 或者 `Rectangle` 的对象,因为 `Shape` 是一个抽象类)。箭头正上方的字符串“contains”表示这两个类的关系。箭头左右两侧的数字揭示了相关对象之间的数量对应关系:给定类 `Diagram` 的一个对象(表示一个框图),会有类 `Shape` 的 0 到多个对象(实际上仍然是 `Circle` 或者 `Rectangle` 的对象)被包含在这个框图中;给定 `Shape` 的一个对象,只会有一个框图包含这个图形元素。

图 1-1 中，给定类 **Diagram** 的一个对象，通过其数据成员 **elements** 可以访问该框图所包含的任意一个图形元素。然而，给定 **Shape** 的一个对象，我们却无法简单地依据其数据成员找到它所属的框图对象。UML 将这种属性称为可达性（**navigability**）。单向的可达性用箭头表示，如图 1-1 所示，而双向的可达性则用不带箭头的实线表示。

类之间有多种关系，上面的 **Aggregation** 关系只是其中的一种。在这些关系中，有两种关系和 **Aggregation** 关系存在着逻辑上的联系：**Association** 关系以及 **Composition** 关系。在绘制类图时，要注意它们之间的联系与区别。

Association，表示两个类之间存在着语义上的关联。两个类是相互独立的，不一定有包含关系。UML 用一个实线来表示这种关系，比如图 1-2 中类 **Venue** 和 **Event** 之间的关系：当在某个地方举办某个活动的时候，这两个类的对象会有语义上的联系，但是并不存在谁包含谁的关系。

Aggregation，如前文所述，表示“部分”与“全部”的关系，一个类的对象会包含另外一个类的对象，但是，可以有多个包含者包含同一个对象。当包含者被析构时，被包含者不一定被析构。UML 用一端为空心菱形的实线来表示这种关系，比如图 1-2 中类 **Team** 和 **Person** 之间的关系。一个团队包含多个人，而一个人可以隶属于多个团队。当一个团队被解散时，团队成员依旧存在。

Composition，也表示“部分”与“全部”的关系，但是其中的“部分”只能够隶属于最多一个“全部”。当“全部”被删除时，其中的“部分”也必须被删除。UML 用一端为实心菱形的一条实线来表示这种关系，比如前文所述的类 **Diagram** 和 **Shape** 之间的关系。一个框图可以包含一个或者多个图形元素，但是一个图形元素只能隶属于一个框图。当该框图被保存、关闭后，所有的图形元素将被析构。有的文献也将这种关系称为 **composition aggregation**。

图 1-2 类之间的几种关联关系

这几种关系之间的逻辑联系是显而易见的：按照 **association**，**aggregation**，**composition** 的次序，约束条件越来越多，因此，**aggregation** 是 **association** 的一种特例，而 **composition** 又是 **aggregation** 的一种特例。但是这几个概念之间也的确存在着区别。并非所有的 **association** 都是 **aggregation**，比如类 **Venue** 和 **Event** 之间存在着关联关系，但是根本就不存在包含关系。另外，并非所有的 **aggregation** 都是 **composition**，比如类 **Team** 和 **Person** 的关系。

在绘制 UML 类图的时候，我们应该使用尽量精确的关联关系来标注两个类的关系。也就是说，如果两个类之间存在 **composition** 的关系，我们就不要使用 **aggregation** 或者 **association** 来标注。如果两个类之间存在 **aggregation** 的关系，我们就不要使用 **association** 来标注。只有

这样，所绘制的类图才能够精确地揭示相关对象之间的关联关系。

例如，为了描述一个手机的机械结构，我们可以使用图 1-3 左侧的类图。一个手机含有多个零件（part），比如 LCD 显示屏、键盘按钮、电池等。一些零件可以组成一个组件（assembly），比如由电阻、电容、芯片等组成的射频信号发射模块。而较小的组件和一些零件可以组成更大的组件，比如由射频信号发射模块、充电模块、音频处理模块等组成的手机主板。由于零件和组件都是更大组件的一个组成部分，我们用基类 component 来描述它们的共同属性。类 Assembly 和 Component 之间应该是 composition 的关联关系，这是由于某一个 component 只可能隶属于一个 assembly，对某一个 assembly 的删除操作必然导致其中所有 component 的删除。

仅仅使用 association 关系来描述类 Assembly 和 Component 之间的关系是不精确的。图 1-3 右侧的类图存在错误情形：

① 设有一个 assembly 的对象 a，它可以包含一个 component 的对象。由于 assembly 是 component 的子类，被包含的对象可以是对象 a，此时，就出现了对象 a 包含其自身的错误情形。

② 设有两个 assembly 的对象 a 和 b。a 可以包含一个 component 的对象。由于 b 可以被看做一个 component 的对象，因而 a 可以包含 b。而同时，b 也可以包含一个 component 的对象，比如 a，因而 b 可以包含 a。这样，就出现了 a 和 b 相互包含的错误情形。

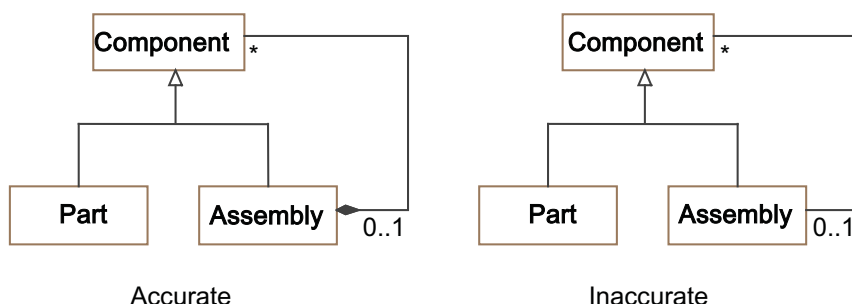


图 1-3 描述手机机械结构的两个类图

当我们使用图 1-3 左侧的 composition 关联关系来描述类 Assembly 和 Component 之间的关系时，就不会出现上述两种错误的情形。这是由于 composition 是 aggregation 关系的一种，而 UML 约定 aggregation 关系必须具备以下两个性质：

（1）反自反（anti-symmetry），即一个对象不能够和自身具有 aggregation 关系。

（2）传递（transitivity）。如果对象 a 是 b 的一个部分，而 b 是 c 的一个部分，则 a 一定是 c 的一个部分。依据反自反性质，错误情形①可以被排除。在错误情形②中，a 包含 b，b 包含 a，依据传递性质，会推导出 a 包含 a 自身，这违背了反自反性质，所以错误情形②也可以被排除。可见，使用精确的关联关系可以正确地描述软件系统中各个类之间的关系。

本节只是对 UML 类图的简要介绍。如果读者需要学习 UML，可以参考文献[3]。如果需要了解一个 UML 术语的严格定义，可以参阅 UML 官方网站 www.uml.org 上的语言规范。

Qt 概述

Qt（发单词“Cute”的音）是一个跨平台的 C++ 开发框架，它包含一个功能丰富的 C++ 类库以及一套简便易用的集成开发工具。Qt 所支持的平台不但包括 Linux，Windows 以及 Mac OS X 等主流桌面操作系统，还包括诸如 Symbian，Maemo 以及 MeeGo 这样的嵌入式操作系统。使用 Qt 编写的 C++ 程序具有良好的跨平台特性，程序员几乎无须更改源代码，所编写的应用程序即可运行在各种操作系统中，这能大幅度缩短开发周期、降低开发成本。Qt 的 C++ 类库是完全面向对象的，经过精心的设计，该类库不但功能强大，而且方便易用。这些优点使得 Qt 被 Adobe，Boeing，Google，IBM，Motorola，NASA，Skype 等大型机构以及众多的中小公司采用。

1. Qt 的历史

回顾 Qt 二十余年的发展历史，我们可以学习是哪些因素促成了 Qt 的成功。Qt 的创始人是 Haavard Nord 和 Eirik Chambe-Eng，二人后来分别成为 Trolltech 公司的首席执行官和总裁。1988 年，受一个瑞典公司的委托 Haavard 开始开发一个 C++ 图形库。1990 年夏季，二人共同开发一个处理超声波图像的数据库系统时，需要一个能够运行在 UNIX，Macintosh 以及 Windows 上的跨平台 C++ 图形库。一天，两人在公园长椅上享受阳光浴时，Haavard 说：我们需要一个面向对象的图形显示系统”，之后的讨论促成了 Qt 的诞生。这是 Qt 成功的首要因素：源于实际需求。

1991 年—1993 年，二人设计并实现了 Qt 库的图形核心库，一组控件以及“信号与槽”机制。1994 年，二人创建了后来的 Trolltech 公司，并与 1995 年 5 月公开发布了 Qt 0.90 版。自发布之日起，Qt 就提供了商业授权和开源软件授权两种方式。发布之后的 10 个月中，没有任何人购买 Qt 的商业授权。直到 1996 年 3 月，欧洲航天局终于购买了 Qt 的 10 份商业授权，Qt 才得以逐步壮大。从这一阶段的历史，我们可以看出 Qt 成功的另外两个因素：开发团队精良的技术（比如提出并实现了“信号与槽”机制）；欧洲人对知识产权的尊重（Qt 创始人能够放心地发布 Qt 的源代码，而欧洲航天局在能够看到 Qt 所有源代码的条件下仍然购买 Qt 的商业授权）。

1997 年，KDE 项目的组织者 Matthias Ettrich 决定使用 Qt 构建 KDE，使 Qt 实际成为 Linux 上开发 C++ 图形程序的标准库。2001 年，Qt 3.0 发布，它的源代码已经超过 50 万行。2005 年，Qt 4.0 发布，包含 500 多个类，9000 个函数。2008 年，Nokia 收购了 Trolltech 公司，将 Qt 作为该公司移动设备的主要开发平台。

2. Qt 库的功能

虽然 Qt 库起初只是一个 C++ 图形库，但是经过多年的演化，它已经成为一个功能丰富

的通用 C++ 类库。它集成了数据库、OpenGL、多媒体、脚本、XML、正则表达式、WebKit 等模块等，其内核部分也加入了进程间通信、多线程等模块。

作为一个成熟的 GUI 框架，它定义了多种功能丰富的控件，实现了事件处理机制，可以实现普通菜单、上下文相关菜单、拖曳、可停靠工具栏等功能。Qt 发明了“信号与槽”机制，各控件利用这一机制发送、处理消息，大幅降低了各控件的耦合度。其他 GUI 框架常常使用回调函数来实现控件之间的通信，相比之下，信号与槽机制更加安全。Qt 库提供的 Graphics/View 框架以及 Model/View 框架可令程序员编写少量的代码，即可显示、编辑应用程序所要处理的数据。

整个 Qt 库支持 Unicode 编码，因而一个 Qt 应用程序可以轻易地同时显示英文、中文、日文、俄文等多种 Unicode 编码所支持的语言。Qt 软件包还提供了诸如 Qt Linguist 这样的工具，便于程序员开发国际化软件产品。

Qt 库的数据库模块内含以下数据库管理系统的驱动软件：Oracle, Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySQL, Borland Interbase, SQLite, 以及其他支持 ODBC (Open Database Connectivity) 接口的数据库管理系统。这意味着可以使用 Qt 访问各种平台上的多种数据库管理系统。除了使用 SQL 语句直接访问数据库外，Qt 还提供了一些与数据库密切相关的控件，以简化数据的访问过程。

Qt 库的 XML 模块包含了能够读取、解析、处理 XML 文档的类。该模块支持 SAX (Simple API for XML) 接口以及 DOM (Document Object Model) 规范。该模块易用、强大、功能完备。Qt 库还允许应用程序使用正则表达式搜索、解析文档，或者在一个数据集中筛选符合某种条件的数据项。

Qt 库集成了浏览器引擎 WebKit，该引擎能够从服务器下载、解析、渲染、显示网页。由于该引擎执行速度快、运行稳定，已被用于 Safari、Google Chrome 等浏览器中。利用这个引擎，Qt 应用程序可以在其界面中显示服务器网页。不但如此，Qt 令 WebKit 引擎将网页的结构以及其中一些对象的细节呈现给 Qt 应用程序，使得程序中的其他控件可以直接和这些对象进行交互。这意味着 Qt 应用程序在显示网页时可以将一些 Qt 控件嵌入到网页中，也可以使用“信号与槽”机制，使得网页中某些对象的状态发生变化时，能够触发本地程序中某些控件的槽函数。

3. Qt 的应用

作为一个成熟的 C++ 开发框架，Qt 已成为数以万计商业和开源应用程序的基础。在桌面应用领域，下面这些重量级的软件都用到了 Qt：（1）Maya，是 Autodesk 公司出品的顶级三维动画制作软件。2011 版开始使用 Qt 进行开发。（2）Mathematica，能够进行数学领域的数值和符号计算，是世界上最流行的数学软件之一。起初使用 Motif 开发，但是效率较低，难以移植到多个平台。为了能够在短时间内开发出能够运行在多个平台的版本，开发团队切换到了 Qt。另外，通过使用 Qt 库中最新的控件，该软件的界面也更加符合时代潮流。（3）Google Earth，能够显示地球上任意一个角落的 3D 地形图，部分地区的精度高达 0.6 米。其

图像来源于卫星照片、航空照相以及地面摄像。(4) Photoshop Elements, 是 Adobe 公司出品的图像编辑软件, 比专业版的 Photoshop 功能稍弱但价格相对很低。(5) Skype, 国际上非常流行的网络电话软件。Qt 被用来开发其客户端部分, 使其能够运行在各种桌面环境中。(6) KDE, 如前文所述, 是一个能够运行在 Linux、UNIX、FreeBSD 等操作系统上的图形工作环境, 也包含一个应用程序开发框架。(7) KOffice, 类似于微软 Office 的一套办公软件。由于使用了 Qt, 该软件可以运行在 Linux, Windows, Mac OS X 等操作系统上。(8) VirtualBox, 是运行在 x86 上的一款虚拟机软件, 目前属于 Oracle 公司。运行该软件的操作系统被称为主操作系统。该软件运行时, 向用户呈现一个虚拟的计算机, 用户可以在这个虚拟计算机上安装一个从操作系统。

Qt 也被一些知名厂商用来开发移动设备中的软件。三星用它来开发数字相框产品 SPF-105V, 中兴用它来开发智能手机 ZTE U980。Qt 也为 Symbian、Maemo 以及 MeeGo 等操作系统提供了优秀的 C++ 应用程序开发框架。

对 Qt 感兴趣的读者可能会关心一个问题: “如果我使用 Qt 来开发商用软件, 是否需要向 Nokia 公司付费?”。本章 2.1 节将讨论 Qt 的版权问题。读者在阅读本书时, 常常希望能够运行、修改书中的例子, 这需要在读者本地机器上安装、配置 Qt 的开发环境。由于 Qt 是一个跨平台软件, 其安装、配置过程稍微有些复杂, 2.2~2.3 节介绍这个过程。为简单起见, 本书举的一些例子只是运行在主控台模式下, 不需要创建任何图形元素, 2.4 节介绍如何创建一个能够输入/输出 Qt 类型的主控台程序。

良好的编程规范(比如变量的命名等)是确保软件质量的一个重要条件, 2.5 节介绍 Qt 开发团队定义的编程规范。本书并不试图全面介绍 Qt 框架, 读者在阅读本书时可以参考 2.6 节罗列的 Qt/C++ 文献。

2.1 Qt 版权

Qt 的最近几个版本提供 3 种授权方式: GPL 协议、LGPL 协议以及商业授权。其中, GPL 协议、LGPL 协议是开源社区中广泛使用的两种协议。作为法律条文, 最精确的阐述是该条文本身, 任何解读都可能是不全面、不准确的, 下面我们仅从 Qt 用户的角度介绍这两种协议。最后, 我们讨论如何在这 3 种协议中进行选择。关于 GPL、LGPL 协议的官方文本以及详细的解读, 请参 www.gnu.org/licenses。

GPL (General Public License) 协议并不是为了保护软件作者的利益, 而是为了鼓励软件开发者相互共享各自的成果。该协议允许软件的用户享有以下权利: 能够得到软件的源代码; 修改软件, 或者将软件的一部分用在用户自己开发的软件中; 能够发行软件的复本, 用户和原作者均可收费, 即使一个软件的用户通常并不是该软件的作者。同时, 该协议要求用户履行以下义务: 一旦该用户所开发的一个软件用到了其他 GPL 软件, 新开发的软件也必须遵循 GPL 协议, 也就是说, 新软件的用户也享有上述权利, 这要求新软件的开发者在发布该软件时也必须发布源代码。

对于 GPL 协议下的软件, 由于软件的用户也有权以收费或者免费的方式发布该软件,

该软件的原作者实际上不能指望通过出售该软件本身赢利。然而，由于原作者对该软件最熟悉，能够提供强有力的技术支持，他们可以通过技术支持的方式获利，比如 Redhat 公司。

LGPL (Lesser GPL) 协议一般适用于类库。该协议允许类库的用户享有以下权利：如果用户没有修改类库，而且是以动态链接的方式使用类库，用户在发布自己开发的软件时，可以不发布源代码。但是，如果用户修改了受 LGPL 协议保护的类库，或者在自己开发的软件中使用了类库中的源代码，则新开发的软件也必须遵循 LGPL 协议，也就是说，在发布新软件时，必须发布类库中被修改的源代码、新软件中相关部分的源代码。

Qt 的商业授权方式允许 Qt 用户享有以下权利：能够对 Qt 进行任意的修改，以满足特定的需求，比如使 Qt 能够运行在某种智能手机平台上。用户可以不公布这些修改；能够在至多两个工作日内，答复用户任何类型的技术问题，不限制问题的总数量。关于这种授权方式的细节，可参见 www.digia.com。

GPL 或者 LGPL 授权的 Qt 是免费的，而商业授权的 Qt 则要求用户支付一定的费用。对于 Qt 用户，如果他希望对开源社区作贡献，可以选择 GPL 授权方式，但他在发布自己的软件时必须附上源代码。如果他在开发中不需要修改 Qt 库，也不需要任何技术支持，可以选择 LGPL 授权方式。如果他需要修改 Qt 库或者需要良好的技术支持，应该选择商业授权方式。

2.2 Qt 库的编译

开发工具的选择。虽然目前支持 Qt 开发的工具较多，比如跨平台的 Qt Creator、QDevelop、Eclipse 等，但是在 Windows 平台上，经过笔者的评测，微软的 Visual Studio 仍然是最优选择。为了支持开发者使用 Visual Studio 开发 Qt 应用程序，Qt 提供了一个插件“Visual Studio Add-in”。安装、执行该插件后，开发者在 Visual Studio 中能方便地创建 Qt 应用程序的项目文件（扩展名为“.vcproj”），也能把 Qt 项目文件（扩展名为“.pro”）转换为 Visual Studio 的项目文件。Visual Studio 有多个版本，本书采用了中文版 Visual Studio 2010（以下简称 VS 2010）。

Qt 版本的选择。Qt 总被不断地更新。本书采用的是 2012 年 4 月 11 日发布的“Qt SDK Version 1.2.1”。该软件包不但包含了 Qt 库 4.8.1，还包括了其他开发工具，比如 Qt Creator 等。Qt 的官方网站 qt.nokia.com/downloads 提供了该软件包的两种安装方式：（1）用户下载一个字节数较少的“安装程序”，用户运行该安装程序，选择所要安装的子模块，该程序再从 Nokia 服务器下载所选择的子模块并将它们安装到用户的本地机器中。（2）用户下载一个包含所有子模块的安装程序，用户在本地机器中运行该程序，选择所要的子模块，完成后续的安装操作。本书采用了第二种方式，所要下载的安装程序“qtsdk-offline-win-x86-v1_2_1.exe”接近 1.8G 字节。

Qt 开发环境的配置。老版本的 Qt 没有直接提供在 Visual Studio 环境下编程所需要的库文件，需要用户使用 Visual Studio 重新编译 Qt 的源代码以生成所需要的库文件。这个过程不但费时，而且可能出错。而 Qt SDK Version 1.2.1 直接提供了该库文件，简化了安装过程。我们可以按照以下步骤配置 Qt 的开发环境。

❶ 安装 VS 2010，比如到默认的 C:\Program Files\Microsoft Visual Studio 9.0 目录下。

❷ 获取并安装 Qt 软件包。截止到 2011 年 2 月 22 日，Windows 平台上最新版的 Qt 软件包为 2010 年 5 月发布的 qt-sdk-win-opensource-2010.05.exe，其中 Qt 库的版本为 4.7.1。读者可以从 Qt 的官方网站下载该软件包。该软件包运行时需要用户输入目标路径。本书设该路径为 d:\qt\4.7.1。

安装完毕后，Qt 4.7.1 的库文件存放在 D:\qt\4.7.1\lib 目录下，头文件存放在 D:\qt\4.7.1\include 目录下，可执行程序（比如 Qt Assistant）以及动态链接库（扩展名为 DLL）存放在 D:\qt\4.7.1\bin 目录下。Qt 4.7.1 的库文件可以供 VS 2010 的 C++ 程序直接使用，没有必要像老版本那样需要使用 VS 2010 的编译器对 Qt 的源代码重新进行编译。因此，进行到这个步骤，我们可以直接在 VS 2010 中编写 Qt 应用程序了。当然，为了能够更方便地在 VS 2010 中开发 Qt 应用程序，我们也可以使用 2.3 节介绍的工具 Qt Visual Studio Add-in。

由于 Qt 4.7.1 软件包已经包含了 VS 2010 编译生成的库文件，一般情况下我们不需要重新编译 Qt 源代码。然而，在某些场合下我们需要重新编译 Qt 库。例如本书 8.3 节在讨论 Qt 应用程序的二进制代码兼容问题时就需要重新编译 Qt 库。这是一项耗时的工作，具体的步骤如下。

❸ 复制 Qt 软件包。本书 8.3 节会本着试验的目的修改 Qt 的源代码。完成试验后我们应该复原 Qt 的源代码。Qt 4.7.1 的所有文件都存放在目录 d:\qt\4.7.1 下。为了能够精确复原 Qt 的源代码，我们令该目录为原始目录，不会对其进行任何修改。做实验时，该目录下的内容会被复制到一个工作目录，本书设为 D:\qt\vc，其中的内容可以被修改。当我们想复原 Qt 源代码时，只需要将原始目录中的文件复制到工作目录即可。由于原始目录中含有大约 3.6 万个文件，这个复制操作需要运行一段时间。我们可以使用 Windows 的资源管理器来完成这个复制操作，也可以使用命令

```
xcopy d:\qt\4.7.1d:\qt\vc/s
```

的方式来完成，这种方式可以显示复制的过程。

❹ 配置适用于 VS 2010 的环境变量。对 Qt 软件包进行编译时，需要以命令行方式使用 VS 2010 的编译器，这需要配置一些环境变量。用一个文本编辑软件打开文件 C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools\vsvars32.bat，使环境变量 PATH 包含 d:\qt\vc、INCLUDE 包含 d:\qt\vc\include、LIB 包含 d:\qt\vc\lib。保存该批处理文件后，在命令行窗口运行该批处理文件。随书光盘 Z:\misc\vsvars32.bat 是一个修改后的文件，读者可以参考该文件进行修改。读者也可以直接运行该文件，跳过以上编辑过程。为了验证是否对上述环境变量做了正确的设置，可以在命令行窗口运行 set 命令，并检查屏幕的输出。

❺ 生成供 VS 2010 编译器使用的配置文件。在命令行窗口中，将当前目录切换到 d:\qt\vc，运行：

```
configure-platform win32-msvc2008
```

命令将生成一个配置文件，供步骤 6 中的编译程序 nmake 使用。Qt 库由 corelib, GUI, network

等多个模块组成，上述命令表示即将编译所有模块。如果开发者只需要使用部分模块，可以使用下面的命令略过其他模块的编译，以节省编译时间：

```
configure -no-sql-sqlite -no-qt3support -platform win32-msvc2008 -no-libtiff  
-no-dbus -no-phonon -no-phonon-backend -no-webkit
```

该命令表示不编译 `sql-sqlite`、`qt3support`、`libtiff`、`dbus`、`phonon`、`phonon-backend` 及 `webkit` 模块。`configure` 命令运行时会询问 Qt 的版本是商业版还是免费开源版，选择免费开源版。该命令大约运行 10 分钟。

6 运行 `nmake` 进行编译。在命令行窗口中，将当前目录切换到 `d:\qt\vc`，运行命令 `nmake`，对 Qt 的源代码进行编译。该命令运行 2~3 个小时。为了验证 Qt 库是否已经被正确编译，用户可以查看目录 `D:\qt\vc\lib`。如果该目录包含 `QtCore4.lib`、`QtGui4.lib` 等库文件，而且这些库文件的更新时间正好为运行 `nmake` 命令的时间，则表示 Qt 库已被成功编译。编译完毕后 `d:\qt\vc` 目录占用了大约 10GB 的空间，其中大部分是 `.obj` 文件。用户可以在该目录下运行命令 “`del *.obj /s`” 删除这些文件，以节省磁盘空间。缩减后的目录大约占用 2.6GB 空间。

2.3 开发环境的设置

在 VS 2010 中开发 Qt 应用程序需要做以下设置。

1. 安装插件 Qt Visual Studio Add-in。为了便于在 VS 2010 环境下使用 Qt 库，可以从 qt.nokia.com/download 下载、安装插件 Qt Visual Studio Add-in。安装时选用默认的安装参数即可。安装完毕后，运行 VS 2010，会有一个名为 Qt 的菜单项。执行其中的 Qt Options\Qt Versions\Add，在 Version Name 域输入任意名字比如 QT 4.8.1，在 Path 域中输入 `d:\qtsdk\desktop\qt\4.8.1\msvc2010`。至此，该插件和 VS 2010 无缝地集成在一起了。

2. 指定 Qt 库的头文件位置以及库文件。设我们已经按照 1.4 节所述将目录 “`d:\qtsdk\desktop\qt\4.8.1\msvc2010`” 映射为盘符 “`q:\`”，则所有 Qt 库的头文件都存放在 `q:\include` 目录下。Qt 库包含多个子模块，每个子模块的头文件存放在一个单独的子目录下。例如，`q:\include\QtCore` 子目录存放 Qt 核心模块的所有头文件，`q:\include\QtGui` 子目录存放模型界面子模块的所有头文件，`q:\include\QtWebKit` 存放 Web 开发子模块的所有头文件。在每个子目录下，所有头文件会被包含到一个“汇总”头文件中。例如，`q:\include\QtGui` 目录下有一个名为 “QtGui” 的头文件，它包含了这个目录下的所有其他头文件。通过包含这个汇总头文件，我们就可以间接地包含这个子目录下的所有其他头文件，不必再写一大堆的 `include` 语句，使得我们的程序看起来更加简洁。

Qt 应用程序通常以相对路径而不是绝对路径的方式包含 Qt 库的头文件，比如使用 “`#include <QtGui>`” 而不是使用 “`#include <q:\include\QtGui>`”。当编译器看到这个预处理命令时，需要知道去哪个目录寻找所包含的头文件。

对于 VS 2010，具体操作步骤如下。首先创建一个项目（或者打开一个已经存在的项目），然后执行 “项目\属性\通用属性\VC++ 目录”，在 “包含目录” 中添加：

“q:\include;q:\include\QtCore;q:\include\QtGui”。这样，我们就可以在 C++ 程序中通过“#include <QtGui/QColor>”或者更简单的“#include <QColor>”方式，包含子目录 QtGui 下的头文件 QColor。如果读者用到了 Qt 库的其他子模块，也应该将它们的路径添加到“包含目录”中。

可供 Qt 应用程序链接的 Qt 库实际上是由多个库文件组成的，这些库文件都存放在 q:\lib 目录下，每个子模块对应两个库文件，其中一个的扩展名为“.lib”，供静态链接模式下使用，另外一个的扩展名为“.DLL”，供动态链接模式下使用。

一个 Qt 应用程序必定会调用某些库文件中的某些函数，VS 2010 链接器需要知道到哪个目录去寻找这些库文件，因此我们需要做以下配置。执行“项目\属性\通用属性\VC++目录”，在“库目录”中添加“q:\lib”。再执行“项目\属性\通用属性\连接器\输入”，在其中的“附加依赖项”中加入库文件的名称。例如，如果一个 Qt 应用程序只用到 QtCore 以及 QtGui 模块，而且这个应用程序以动态链接方式和 Qt 库相链接，那么所要加入的库文件应该为“qtmaind.lib QtGuid4.lib QtCored4.lib”。

以动态方式链接的 Qt 应用程序在运行的时候需要使用 q:\bin 目录下的动态链接库(DLL 文件)，所以用户应该将该目录加入到 Windows 系统的环境变量 PATH 中。具体来说，执行显示桌面\我的电脑\右键\属性\高级\环境变量\系统变量\PATH，在其末尾加上 q:\bin。

3. 编写测试程序。安装完毕 Qt Visual Studio Add-in，我们可以尝试在 VS 2010 环境下编写一个小测试程序。运行文件\新建\项目，在弹出的对话框中选择 Qt4 Projects。有多种项目类型可供选择，本书仅涉及其中两个。

(1) Qt Concole Application，即主控台程序。这种类型的程序不使用 Qt GUI 模块的任何类或者函数，运行时没有图形界面，只有一个能够显示文字的窗口，允许用户通过键盘输入一些信息，程序将运行结果以文字形式显示在该窗口中。该窗口通常被称为命令行窗口。

(2) Qt Application，即一般的图形界面程序。这种类型的程序使用 Qt GUI 模块的函数，显示一个具有图形界面的窗口。以上两种类型相互排斥，不要试图创建一个既具有命令行窗口又具有图形界面窗口的程序。如果一个程序使用了 Qt GUI 模块的任何类或者函数，就只能创建图形界面程序。2.4 节将介绍如何创建主控台程序，本节仅介绍如何创建图形界面程序。

在“名称”域输入 hello，在“位置”域输入一个目录名，以存放 Qt 插件自动生成的源文件、头文件。单击“确认”按钮后会启动一个创建项目的向导，其中有一页罗列了 Qt 的多个软件模块，供程序员选择，以和应用程序链接。我们暂时选择默认配置即可。项目创建完毕后，运行“生成\生成解决方案”对项目进行编译。成功后，执行“调试\开始执行（不调试）”，一个标题为“hello”的空白窗口弹出，表明该应用程序运行成功。下面我们来看看 Qt 是如何实现这个简单程序的。

从 VS 2010 的解决方案浏览器可以看到 Qt 插件为我们创建了源文件（source files）hello.cpp, main.cpp, 头文件（header files）hello.h、资源文件（resource files）hello.qrc，以及若干生成文件（generated files）。目前，我们不必弄清楚每个文件、每一行的含义，我们只关心其中部分内容。hello.h 的主要内容如下。

```

#include <QtGui/QMainWindow>           ①
#include "ui_hello.h"
class hello : public QMainWindow       ②
{
    Q_OBJECT
public:
    hello(QWidget *parent = 0, Qt::WFlags flags = 0);
    ~hello();
private:
    Ui::helloClass ui;
};

```

由于该头文件使用了类 `QMainWindow`，所以行①包含了头文件 `QMainWindow`。Qt 的每个类均对应着一个与该类同名的头文件。行②从类 `QMainWindow` 派生新类 `hello`。类 `QMainWindow` 实现了一个应用程序主窗口的功能。它能够在操作系统的图形环境中显示自己，接收鼠标、键盘等消息进行处理。菜单、滚动条等部件常常作为子对象，被添加到该类的对象中。

用户在声明类 `hello` 时用到了宏 `Q_OBJECT`，应该将声明该类的头文件 `hello.h` 加入到 VS 2010 项目的“Header Files”部分，否则将产生编译错误，具体原因我们将在 17.2 中阐述。

项目中 `main.cpp` 的内容如下。

```

#include "hello.h"
#include <QtGui/QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);       ③
    hello w;                           ④
    w.show();                          ⑤
    return a.exec();                  ⑥
}

```

行③定义了一个 `QApplication` 对象，用于管理程序命令行等信息。行④定义一个 `hello` 对象，行⑤显示该对象。行⑥启动 Qt 的事件处理机制，处理用户的鼠标输入、键盘输入等事件。

4. Qt 文档。Qt 软件包提供了丰富的文档，介绍 Qt 库中各个类的用法、Qt 开发环境（比如 Qt Creator）的用法等内容。这些文档存放在目录 `d:\qtsdk\documentation` 下。为了阅读这些文档，读者应该运行“Qt 助手”，也就是 `q:\bin` 目录下的 `assistant.exe`，然后执行“编辑\首选项\文档\添加”，选择 `d:\qtsdk\documentation` 目录下的某个文件，比如介绍 Qt 库用法的 `qt.qch`，以阅读该文件内容。

2.4 主控台的输入与输出

本书一些例子需要从键盘接收输入，并向屏幕输出文本信息。如果输入/输出数据是 C++ 的基本类型，我们仍然可以使用 C++ 标准库定义的 `cin` 与 `cout`。如果输入/输出数据是 Qt 库中某些类的对象，我们就不能使用标准库中的 `cin` 与 `cout`，因为 Qt 为了支持 Unicode，设计了自己的输入/输出架构。

第 7 章将详细阐述 Qt 的输入/输出流框架，本节简要介绍如何使用该框架。类似于 C 语言，Qt 也将键盘、屏幕看作特殊的文件。为了对文件进行输入/输出操作，Qt 定义了类 QFile 以及 QTextStream。QFile 是对 C 语言文件操作函数的简单封装，实现底层的文件操作。而 QTextStream 实现二进制数据与 Unicode 文本数据之间的转换。

对文件操作时，一个 QTextStream 对象和一个 QFile 对象绑定。当需要输出程序中的二进制数据时，QTextStream 将二进制数据转换为 Unicode 文本格式，再由 QFile 将数据写入文本文件。读取数据时，QFile 从文本文件读取数据，由 QTextStream 完成文本格式到二进制格式的转换。如果该文本文件恰好是 C++ 中预先定义的全局变量 FILE * stdin，则从键盘接收数据；如果该文本文件是 stdout，则向屏幕输出。

我们也可以使用 Qt 提供的类 QDebug 来输出数据，该类支持 C++ 基本数据类型、Qt 中的大部分数据类型。代码段 2-1 演示了以上两种方式。行①、②中的对象虽然与 C++ 标准库中的 cin, cout 同名，但二者的类型是 QTextStream，因而可以对一个类型为 QString 的对象进行操作。行③的函数 qDebug() 返回一个全局默认的 QDebug 对象，该对象支持 QRect 类型对象的输出。

代码段 2-1，使用 Qt 进行主控台输入与输出，取自 z:\examples\qt_console\main.cpp

```
QTextStream cin(stdin, QIODevice::ReadOnly);           ①
QTextStream cout(stdout, QIODevice::WriteOnly);        ②
int main(int argc, char *argv[])
{
    QString str;
    cin >> str;
    cout << "the string is " << str << endl;
    qDebug() << "Qt types " << QRect(0,0,10,10) << endl;    ③
}
```

创建上述主控台应用程序的步骤如下。在 VS 2010 中执行 File\New\Project，在 Project Types 中选择 Qt4 Projects，在 Templates 中选择 Qt Console Application。在 Name 域中输入一个项目名称比如 qt_console，在 Location 域输入一个已经存在的目录名。不选“Create directory for solution”选择框，以默认配置执行后续步骤，即可创建一个主控台应用程序。

关于选择框“Create directory for solution”的细节如下：VS 2010 的一个 solution 可以包含多个 projects，每个 project 可以有自己独立的子目录。每个 solution 对应着一个描述文件，描述该 solution 包含哪些 projects，而每个 project 也会对应着一个描述文件，描述该 project 包含哪些 C++ 源文件、头文件等。如果一个 solution 只含有一个 project，可以不选 Create directory for solution 选项，此时该 solution 以及该 project 共享一个目录，二者的描述文件都被存放在该目录下。如果选择了这个选项，VS 2010 会为二者各自创建一个目录，将二者的描述文件放在各自的目录下。

2.5 Qt 风格的编程规范

笔者的一位朋友曾在北京一家电信软件公司担任技术总监。当我问他每天都在忙碌什么时，他半开玩笑的回答“我们公司的软件系统经过这么多年的开发，体系架构、性能等都已

经比较成熟，并不需要大幅度的重构。我每天做的工作也就是修改函数、参数的名字，使程序更加易读”。的确，在传统的大学课程中，算法、系统架构、设计等会被强调，而诸如函数命名这样的编程规范常被忽略。良好的编程规范可以大幅提高一个程序的可读性、可维护性。

软件开发领域并没有一个公认的、统一的编程规范，不同的软件开发项目或者组织会采用不同的编程规范，如 Google 公司采用参考文献[4]作为该公司所有开源项目的编程规范。Qt 的开发团队采用参考文献[5]作为编程规范。该规范使得 Qt 的 API 更容易被程序员理解和使用，缩短了开发人员的学习周期，是 Qt 得以广泛流行的原因之一。学习并遵循该规范，可以使我们开发的 Qt 应用程序也更易读、更易维护。下面我们介绍该规范的主要内容。

1. API 的设计原则。(1) 精炼。软件系统应该尽量少定义类，每个类中应该尽量少定义成员。(2) 清晰而简单的语义。用户应该能够轻易地使用一个 API 来解决那些常见的任务。对于不常见、复杂的任务，用户也能通过该 API 来实现，但是这不应该成为 API 设计的重点。另外，API 不能过度追求通用性，它应该首先被用来解决具体的、常见的问题。(3) 直观。使得那些没有阅读过一个 API 文档的程序员也能够读懂使用该 API 开发出来的程序。(4) 容易记忆。应该使用精确、统一的命名规则来对类、函数、参数等进行命名。

2. 可读性比精炼更重要。例如，语句：

```
QSlider *slider = new QSlider(12, 18, 3, 13, Qt::Vertical, 0, "volume");
```

虽然精炼但是可读性差。类 QSlider 的 API 应该被改为：

```
QSlider *slider = new QSlider(Qt::Vertical);
slider->setRange(12, 18);
slider->setPageStep(3);
slider->setValue(13);
slider->setObjectName("volume");
```

又比如，当我们调用一个控件的成员函数 `repaint()` 重新绘制该控件时，有时我们希望先擦除其背景再绘制，有时却又希望直接绘制以提高速度。一种做法是令该函数包含一个布尔类型的参数，表示是否需要擦除背景。这种做法虽然只用了一个函数来处理两种情形，看起来比较简洁，但是可能导致下面的代码：

```
widget->repaint(false);
```

API 的初级用户可能将其理解为“不重新绘制控件”。因此，为了提高代码的可读性，更好的一个做法是用两个函数来实现上述两种情形：成员函数 `repaint()` 擦除背景再绘制控件，而成员函数 `repaintWithoutErasing()` 只绘制控件、不擦除背景。

3. 相似的类应该具有相似的接口。例如，Qt 中的 `QTextStream` 负责以文本形式输入或者输出程序中的数据，而 `QDataStream` 负责以二进制方式输入或者输出。由于二者具有相似的功能，它们都重载了运算符“<<”以及“>>”，使得熟悉其中一个类的用户能够迅速学会另外一个类的使用方法，缩短了学习周期。

4. 命名原则。Qt 推荐使用以下命名原则。

- 不要使用缩写。即使对于“previous→prev”这样广为接受的缩写形式，Qt 也不予采纳，因为这要求程序员记忆哪些单词被缩写、哪些没有被缩写。
- 如果一个标识符含有多个单词，则第 2 个单词之后所有单词的首字母应该大写，比如“installSceneEventFilter”。有的编程规范建议使用下画线分割各个单词，但这会增加标识符的长度。
- 类的名字以大写字母开始。其中绝大多数以字母 Q 开始，表示该类是 Qt 软件包中定义的类，此时，第二个字母也应该是大写的。而且，类的名字应该是一个名词，比如 QFileSystemWatcher。
- 函数名字以小写字母开始，应该是一个动词或者含有动词的短语，比如 collidesWithItem()。
- 全局常量名字中的字母都应该大写，比如 Q_BIG_ENDIAN。枚举常量的名字应该含有枚举类型的信息。这是由于枚举常量在 C++ 程序中可被直接使用，如果其名字过于简单，可能会导致歧义。例如，设有：

```
enum CaseSensitivity { Insensitive, Sensitive };
```

由于有上下文，枚举常量 `Insensitive` 及 `Sensitive` 在被定义时并没有任何歧义。然而，当它们在程序其他地方被引用时，程序的读者难以理解它们的含义。因此，应该在它们的名字中嵌入枚举类型的信息，比如

```
enum CaseSensitivity { CaseInsensitive, CaseSensitive };
```

- 如果某个数据成员具有布尔类型，读取该数据的成员函数应该以“is”打头，比如 `isEmpty()`、`isMovingEnabled()`。然而，如果该数据成员的名字是复数形式的，则不需要加任何前缀，比如 `scrollBarsEnabled()`。

2.6 与 Qt 及 C++ 相关的文献资源

1. 与 Qt 相关的文献资源。developer.qt.nokia.com/books 列举了有关 Qt 的书籍。初始学习阶段，读者可以阅读书籍 *Foundations of Qt Development*^[6]，*An Introduction to Design Patterns in C++ with Qt4*^[7] 以及 *C++ GUI Programming with Qt 4*^[8]。也可以在 Qt 官方网站 qt.nokia.com 上观看 Qt 专家的讲座录像、Qt 培训资料、电子杂志“Qt Quarterly”。在开发过程中，可以参考 Qt 联机文档，查看类的使用方法。如果想学习深层的编程技巧，可以参考 *Advanced Qt Programming*^[9]。该书的作者正是负责撰写 Qt 联机文档的 Mark Summerfield。Qt 的创始人之一 Eirik Chambe-Eng 为该书写了序言。

英文社区中，网站 blog.qt.nokia.com 对 Qt 进行一般性的讨论，网站 labs.qt.nokia.com 的 blog 部分含有一些技术水准较高的文章。developer.qt.nokia.com/forums 讨论与 Qt 相关的技术问题。中文社区中，网站 www.cuteqt.com，www.qtcn.org 以及 qt.csdn.net 讨论 Qt 的使用与开发。

开发 Qt 应用程序时，开发者可以使用 Qt 软件包中的 Qt Assistant 阅读 Qt 的联机文档。

该文档介绍类库中各个类的功能，类之间的继承关系，编程工具 Qt Designer、Qt Linguist、QMake 的使用等。这些文档的格式是 Qt 特有的，程序员只能使用 Qt Assistant 来查看。阅读文档内容时，用户可以使用 Ctrl+鼠标滚轮更改字体大小，还可以在多个标签页中显示具有不同主题的内容。

2. 与 C++ 相关的文献资源。关于 C++ 的书籍、论文、论坛非常多，即使那些被 C++ 程序员奉为经典的书籍也多达十几本。以下是笔者推荐的一些书籍、网站。其中大多数文献都有对应的中文版，本文不再一一列举。对于具有一定英文基础的读者，笔者建议直接阅读英文原版文献，因为同一个 C++ 概念在不同的中文版图书中会有不同的翻译，会增加阅读、交流难度。对于 C++ 的初学者，如果他以前学习过 C 语言，可以选择下面的 *Thinking in C++* 一书，否则，可以选择下面的 *C++ Primer* 一书。

C++ Primer^[10]，详细、系统地阐述了 C++ 语言的各种语言特性，文笔流畅，通俗易懂。该书涵盖了 C 语言的内容，因而适用于那些从未接触过 C 语言的读者。该书的作者之一 Stanley B. Lippman 曾和 C++ 创始人 Bjarne Stroustrup 在 Bell 实验室一起工作，实现了 cfront。另外一个作者 Josée Lajoie 是 IBM 加拿大实验室 C++ 编译器项目组成员，自 1990 年起成为 C++ 标准委员会的成员，曾担任该委员会的副主席。虽然他们对 C++ 语言有着深刻的理解，但是他们在本书中却常常使用一些通俗易懂的小例子，来讲述为什么要使用某个语言特性以及如何使用它。

Thinking in C++^[11]，假设读者已经掌握了 C 语言。在批判 C 语言缺点的过程中，逐步引入 C++ 的各个语言特性，借此证明这些语言特性的作用。书中例子短小精炼，仅用来说明某个语言特性，不涉及任何实际问题。该书分为两卷，第一卷介绍基本的 C++ 语言特性，文笔流畅而严谨，第二卷介绍 C++ 标准库、多继承、运行时类型信息等内容，虽然仍然保持着第一卷通俗易懂的风格，但是极少数内容似乎写得比较仓促。由于国内许多大学将 C 语言作为一门独立的课程，那些掌握了 C 语言的学生可以将该书作为学习 C++ 语言的入门教材。

以上两本教材主要讨论了“C++ 是什么”。要想在编程实践中“正确、有效地使用 C++”，可以阅读下面的 *Effective C++* 一书以及本书。*Effective C++*（以及其姊妹篇 *More Effective C++*）将这些知识总结为 85 条编程准则，而本书结合 Qt 这一具体开源项目，以具体而鲜活的形式诠释这些知识。

Effective C++^[2]，文笔流畅、精炼、幽默。随着 C++ 的演化，作者出版了 1992、1998、2005 年三个版本。这组书籍讨论如何更加有效地利用 C++ 的各种语言特性，以设计出更好的面向对象程序。作者以 50 余个短小、具体、易于记忆的准则，总结了 C++ 程序员应该做什么（以及原因）、不应该做什么（以及原因）。其中一些准则和设计策略相关，比如，是使用继承还是使用模板，是使用函数名重载还是使用默认参数，一个类的成员函数何时应该被定义为虚函数，何时应该被定义为普通函数。另外一些准则和某个具体的语言特性相关，比如，不要像 C 语言那样在函数体的开始部分定义所有变量，而是应该尽量推迟对象的定义。该书的作者还于 1996 年出版了另外一本风格相似的书 *More Effective C++*，给出了一些更深层次的准则，比如如何提升程序的性能，如何定义智能指针等。

对于那些具有一定开发经验、想要全面、准确地把握 C++ 语言的程序员，可以参考下面的文献。

The C++ Programming Language^[12]，由“C++之父”Bjarne Stroustrup 撰写，具有 1986、1991、1997、2000 年四个版本，讨论如何使用 C++ 的语言特性设计、编写更好的面向对象程序，该书最后一部分甚至专门从整体的软件开发与设计这一角度讨论 C++ 的各种语言特性。该书系统、准确，书中的观点是作者几十年软件开发实践中宝贵经验的结晶，值得 C++ 程序员仔细品味、借鉴。但是，作者有意将该书的读者定位为具有一定开发经验的 C++ 程序员，加之文笔稍显晦涩，学术味重，使得该书不适合初学者。

要想开发一个高质量的软件系统，仅了解 C++ 语言本身是不够的，开发人员还应该从软件工程这一更高的角度来看待软件开发。虽然这一领域的理论、文献更加繁多，但是笔者建议读者至少要系统地阅读下面这本教材。

Design patterns : elements of reusable object-oriented software^[13]，设计模式领域的经典之作。针对软件开发中经常出现的一些问题，该书以类图的形式给出每个问题的解决思路，阐述类和类之间的协作关系，讨论这种设计对系统的可复用性、性能或者其他方面的影响，并给出 C++ 或者 Smalltalk 语言的具体实现。学习这些设计模式，读者一方面可以直接将它们应用在自己的设计中，另一方面可以揣摩如何利用 C++ 语言的各种特性来解决实际设计问题。这些设计模式是该书作者们从大量设计优良的软件系统中总结而来的，因而在一个规模稍大、质量较高的软件系统中，我们可以找到这些模式的具体应用。Qt 就是一个例子，本书将阐述 Qt 是如何使用其中一些设计模式来解决具体问题的。

关于 C++ 的网络资源也非常丰富。导航网站 www.robertnz.net/cpp_site.html 含有多个链接，指向与 C++ 相关的文献、库、编程规范、新闻组等。随书光盘“Z:\misc\Sites of interest to C++ users.mht”保存了该网站的内容，供读者参考。另外一个类似的导航网站是 C++ FAQ (www.parashift.com/c++-faq-lite)。

Model/View 框架

具有图形用户界面的应用程序常需要使用一些控件来显示程序中的数据，或者接收用户输入的数据。一种设计策略是令控件既负责存储程序中的数据，又负责显示或者编辑这些数据。这种策略虽然简单、直观，但是难以复用、灵活性差。例如，设有两个控件，一个负责显示一系列的字符串，另外一个负责显示一系列整数。虽然这两个控件在外观、处理用户交互命令等方面有诸多相似之处，但是由于它们所含数据的类型不同，两者无法复用对方的代码。还比如，设一个控件所要处理的数据量很大，用户需要使用多个控件以查看不同部分的数据。由于在这种设计策略下，数据被嵌入在控件内部，我们只能创建多个相同类型的控件，每个被用来显示不同部分的数据。但是这会导致大量数据的重复存放。

模型-视图-控制器架构（Model-View-Control structure, MVC）可以很好地解决这一问题。该架构将数据及其显示分离开来，使用模型存储、访问数据，使用视图显示数据，使用控制器处理用户的交互命令。无论采用什么数据结构存放数据，模型总会提供一个统一的数据访问接口，我们将其称为“模型访问接口”。使用该接口，一个视图可被用来显示不同模型中的数据，如图 13-1 左半部分所示，这提高了视图的可复用性。当需要查看大量数据的不同部分时，可以采用图 13-1 右半部分的结构，令多个视图通过模型访问接口，访问不同部分的数据并显示。数据虽然被显示在多个不同的视图中，但是程序并没有重复存放这些数据。当视图 A、B、C 是不同类的对象时，它们将以不同的方式显示模型中的数据，这往往是用户所期待的。

图 13-1 MVC 架构的优势

标准 MVC 架构中，控制器处理用户的交互命令，视图仅负责数据的显示。Qt 中的 Model/View 框架实现了标准 MVC 架构的功能，但是它的体系结构和标准 MVC 有些差别。如图 13-2 所示，Qt 的 Model/View 框架由模型、视图、委托（delegate）组成。其中，模型负责存储与访问数据，这和 MVC 中的一致。数据的显示被分为两个部分：每个数据项的绘制由委托完成，其他的绘制工作由视图完成。对用户交互命令的处理也被分为两个部分：对每个数据项的编辑命令由委托完成，其他的交互命令由视图完成。或者说，视图负责总体的绘制与交互，委托仅负责单个数据项的绘制与交互。这样的职责划分和 MVC 中的并不一致。

图 13-2 Model/View 框架的总体结构

具体地说，Model/View 框架中的数据集（dataset）由若干个数据项（item）组成，每个数据项除了存放应用程序所要显示的数据之外，还存放着框架中其他部分需要访问的数据，比如这个数据项的显示颜色。无论数据集以什么数据结构存放，模型（model）总是将它看作一棵树，每个树节点就是一个数据项。具有列表或者表格结构的数据集被看作只含最顶层节点、不含任何子节点的树。“模型”提供统一的编程接口，供视图（view）、委托（delegate）或者应用程序本身获取这棵树的结构信息以及每个节点的数据。

“视图”负责绘制总体外观并处理用户的交互命令，但它并不具体负责每个数据项的绘制与编辑。对于具有列表或者表格结构的数据集，视图计算各数据项的位置，绘制每列上方或者每行左侧的标头（header）。对于具有树状层次结构的数据集，视图还绘制树状的层次结构。视图负责处理用户的浏览、选择等交互命令。例如，用户使用键盘或者鼠标选择一些数据项时，视图能够记录哪些数据项被选择。

“委托”负责绘制每个数据项，并创建“编辑器”控件对数据项进行编辑。默认情况下，Model/View 框架会创建一个委托对象，负责绘制与编辑各种类型的数据项。程序员也可以创建新的委托对象，以截然不同的方式对某些数据项进行绘制与编辑，提高了框架的灵活性，这是引入“委托”的主要目的。当用户希望编辑某个数据项时，委托对象并不会亲自处理这个任务，它会依据被编辑数据项的数据类型，创建一个编辑器控件，并对目标数据项进行编辑。编辑完成后，委托对象负责将编辑器中的数据写回模型。

图 13-3 中的例子具体演示了框架各个部分的分工。数据集含有 4 行、2 列数据项，视图对象计算每个数据项位置，绘制水平、垂直方向的标头。委托对象负责绘制每个数据项。当用户双击一个数据项时，一个编辑器控件负责对其进行编辑。

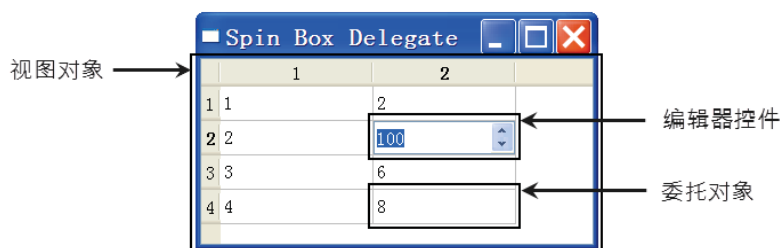


图 13-3 使用 Model/View 框架的一个例子

负责实现模型、视图、委托功能的类分别被称为模型类、视图类、委托类，这些类的对象分别被称为模型对象、视图对象、委托对象。Qt 的 Model/View 框架比较复杂，涉及 36 个类。本章 13.1 节介绍该框架的总体结构，主要类之间的协作关系以及各个类的主要职责。为

了使用 Model/View 框架，程序员应该将数据集封装在一个模型类中，并实现模型访问接口，以供视图类或者委托类访问。13.2 节介绍了这个实现过程。这一节是理解 Model/View 框架的关键。

视图类通过模型访问接口访问应用程序的数据，与数据的存储、访问细节没有任何关系，因而，在最简单的情形下，程序员只需创建一个视图对象，该对象即可显示模型中的数据集。当然，程序员也可以调用视图对象的成员函数，控制视图的外观（比如标头文字等）。13.3 节介绍了视图类的功能与使用方法。

用户在浏览一个视图时，常会选择某些数据项，以对它们做进一步的操作（比如删除、拖曳等）。视图对象能够处理与选择操作相关的交互命令，记录哪些数据项被选择，并利用 Qt 的信号与槽机制通知应用程序数据项的选择状态发生了变化。应用程序所要做的是读取数据项的选择状态并做进一步处理。13.4 节将讨论这部分内容。

大多数情况下，程序员只需利用 13.2～13.4 节讨论的知识即可顺利地使用 Model/View 框架，无需关心该框架中的“委托”，这是由于每个视图对象会指向一个默认的委托对象，后者能完成常见数据项的绘制与编辑任务。然而，当程序员需要彻底地控制数据项的绘制，或者需要使用新的编辑器来编辑某些数据项时，就需要创建新的委托类，13.5 讨论这一话题。

某些情况下，我们希望对模型中的数据集做一些临时处理（比如排序、筛选）之后，再送给视图对象显示，但是我们又不希望在物理上修改数据集。Model/View 框架中的代理模型（proxy model）可以完成这个任务，13.6 节讨论相关的类。

在前面这些小节的讨论中，为了使用 Model/View 框架，我们至少需要创建一个模型对象、一个视图对象。如果数据集很简单，我们也可以使用 13.7 节讨论的“便利视图类”对数据集进行处理。便利视图类在其内部定义了模型对象，并提供了简洁、易用的接口来操作模型中的数据集。同时，作为视图类，它们也能显示模型中的数据集。因此，我们只需定义一个便利视图类的对象，即可使用 Model/View 框架。

13.1 Model/View 框架总体架构

Model/View 框架中，所有模型类具有共同的抽象基类 `QAbstractItemModel`，所有视图类具有共同的抽象基类 `QAbstractItemView`，所有委托类具有共同的抽象基类 `QAbstractItemDelegate`。这些类之间的协作关系如图 13-4 所示。

图 13-4 模型类、视图类、委托类的协作

Model/View 框架涉及的类如图 13-5 所示。初看起来这个框架非常复杂，但是由于所有的模型类、视图类、委托类都遵循上述协作关系，我们其实只需关注 `QAbstractItemModel` 的各个子类的区别、`QAbstractItemView` 的各个子类的区别以及 `QAbstractItemDelegate` 的各个子类的区别。

图 13-5 Model/View 框架涉及的类

(1) **模型类**。作为所有模型类的抽象基类，`QAbstractItemModel` 定义了模型访问接口。这个接口是一组纯虚函数，负责访问数据集中的数据项。该类将数据集看作一棵树，但是该类并没有定义任何数据成员来存放数据集。类似地，除了具有灰色背景的类 `QStandardItemModel` 外，其他模型类也都如此。由于 `QAbstractItemModel` 是一个抽象基类，我们并不能直接定义该类的对象。使用模型类的一个简单做法是了解 `QAbstractItemModel` 各个派生类的功能，寻找一个与应用程序数据集特征最接近的派生类，定义该类的对象或者派生该类的子类。如果 `QAbstractItemModel` 的所有派生类都无法满足我们的要求（比如性能方面的），我们才需要直接派生 `QAbstractItemModel` 的子类，但这需要实现模型访问接口中的几个纯虚

函数，编程工作量较大。

`QAbstractListModel` 负责处理具有列表结构的数据集。它已经实现了部分模型访问接口，我们只需派生该类的子类，实现模型访问接口中的其他虚函数，因而比直接派生 `QAbstractItemModel` 的子类要简单。如果应用程序只是企图显示、编辑一些字符串，可以使用 `QAbstractListModel` 的子类 `QStringListModel`。程序员所需做的只是将这些字符串写入一个 `QStringList` 对象中，然后将这个对象传递给一个 `QStringListModel` 对象。后者实现了整个模型访问接口，因而程序员不需要重载它的任何成员函数。

`QAbstractTableModel` 负责处理具有表格结构的数据集。它已经实现了部分模型访问接口，我们只需派生该类的子类，实现模型访问接口中的其他虚函数。它的派生类 `QSqlQueryModel`，`QSqlTableModel`，`QSqlRelationalTableModel` 和关系型数据库相关，限于篇幅，本书不予讨论。

`QStandardItemModel` 负责处理具有树状层次结构的数据集。当然，由于列表、表格也可以被看作特殊的树，该类实际上也可以处理具有列表、表格结构的数据集。和前面几个模型类不同，该类在其内部定义了一个数据结构，用来存放树状结构的数据集。它使用类 `QStandardItem` 表示一个树节点，`QStandardItem` 定义了一些数据成员，用来存放数据集的数据。另外，它还定义了一些指针，每个指针指向另外一个 `QStandardItem` 对象，形成当前节点的子节点。通过这种方式，`QStandardItemModel` 可以存储具有任意多层的树状数据集。

`QFileSystemModel` 专门负责处理本地文件系统。它实现了整个模型访问接口，其他视图类通过该类可以立即显示本地文件系统。

和前面的模型类都不同，`QAbstractProxyModel` 并不直接处理数据集，它对其他模型中的数据项进行一些处理，再将处理结果呈现给框架中的其他对象。该类的子类 `QSortFilterProxyModel` 能够做排序、筛选处理。

(2) 视图类。作为所有视图类的抽象基类，`QAbstractItemView` 负责绘制总体外观并处理用户的交互命令。由于它是一个抽象类，我们不能直接定义该类的对象，而是应该在该类的派生类中，选择一个和数据集拓扑结构相似的类，定义其对象来显示数据集。`QListView` 适合显示具有列表结构的数据集，`QTableView` 适合显示表格结构的数据集，`QTreeView` 以及 `QColumnView` 适合显示树状结构的数据集。`QHeaderView` 仅被用来显示标头部分，与数据项无关。

以上几个视图类实现普通视图的功能，在它们的内部没有定义任何模型对象。使用这些视图类时，程序员必须自行构造模型对象。为了方便对 Model/View 框架的使用，Qt 还提供了一组便利视图类。这些类在其内部定义了模型对象，并定义了一组简洁、易用的成员函数来操作模型对象中的数据项。其中，`QListWidget` 存储并显示一个列表，`QTableWidget` 存储并显示一个表格，`QTreeWidget` 存储并显示一棵树。在图中这些类的背景被显示为灰色，表示它们包含模型对象。这些视图类分别使用类 `QListWidgetItem`，`QTableWidget` 以及 `QTreeWidgetItem` 表示模型对象中的数据项。

对于 `QListWidget` 表示的列表或者 `QTableWidget` 表示的表格，依据其行数、列数即可遍历整个数据集。对于 `QTreeWidget` 表示的树状数据集，遍历所有树节点并非易事。为此，Model/View 框架提供了迭代器 `QTreeWidgetIterator`，对 `QTreeWidget` 表示的树进行前序遍历。

(3) 委托类。抽象基类 `QAbstractItemDelegate` 及其子类负责视图中每个数据项的绘制与编辑。视图类的内部定义了一个指针，指向一个委托对象。当视图需要绘制每个数据项时，会将该数据项的屏幕位置等信息传递给委托对象进行绘制。`QAbstractItemDelegate` 的子类 `QItemDelegate` 总是以一种默认的风格绘制数据项，而它的另外一个子类 `QStyledItemDelegate` 使用应用程序当前的界面风格进行绘制，使得整个界面具有协调一致的风格，因而我们应该尽可能地使用后者。当用户需要编辑某个数据项时，委托对象会将目标数据项的类型传递给类 `QItemEditorFactory`，后者负责创建一个能够编辑该类型的编辑器控件，对目标数据项进行编辑。编辑完毕后，委托对象调用模型访问接口，将编辑结果写回模型。

(4) 索引。在视图类或者委托类看来，模型类中的数据集总是一棵树，该树可能具有多层树节点。由于模型的访问者对数据集的存放方式一无所知，为了表示要访问哪个节点，访问者会通知模型类目标节点的父节点，以及在这个父节点所包含的所有子节点中，目标节点所在的行号、列号。模型类收到这些信息后，在数据集中找到目标节点，创建一个 `QModelIndex` 对象，逻辑上指向这个节点，这个 `QModelIndex` 对象就被称为目标节点的索引。模型的访问者此后就可以使用这个索引来访问目标节点，提高了访问效率。另外，`QModelIndex` 内部还定义了一个指针，指向它所属的模型对象。这样，给定一个索引，不再需要任何其他信息，就可以唯一确定一个模型对象中的一个数据项。

当数据集发生变化时，比如部分数据项被删除或者有新的数据项加入，`QModelIndex` 表示的索引可能失效，也就是不再指向原先所指的数据项。而 `QPersistentModelIndex` 所表示的索引却总是指向某一个数据项，不会随着数据集的变化而失效，除非它所指的数据项被删除。

(5) 选择。视图类允许用户选择一些数据项。在最复杂情况下，用户可以选择多个父节点所包含的多个子节点。以从上到下、从左到右的阅读顺序，每个父节点下被选的多个子节点可能是不连续的。我们将连续的被选子节点形成的集合称为“选择块”。一个 `QItemSelectionRange` 对象记录一个选择块的范围信息，它使用数据成员 `tl` (top-left) 表示选择块左上角数据项的索引，使用数据成员 `br` (bottom-right) 表示选择块右下角数据项的索引。一个 `QItemSelection` 对象包含多个 `QItemSelectionRange` 对象，表示多个选择块的信息。

视图类使用 `QItemSelectionModel` 记录选择信息，该类的数据成员 `currentSelection` 所指的 `QItemSelection` 对象表示用户最近一次所做的选择，而数据成员 `ranges` 所指的 `QItemSelection` 对象表示最近一次之前所选的选择块。

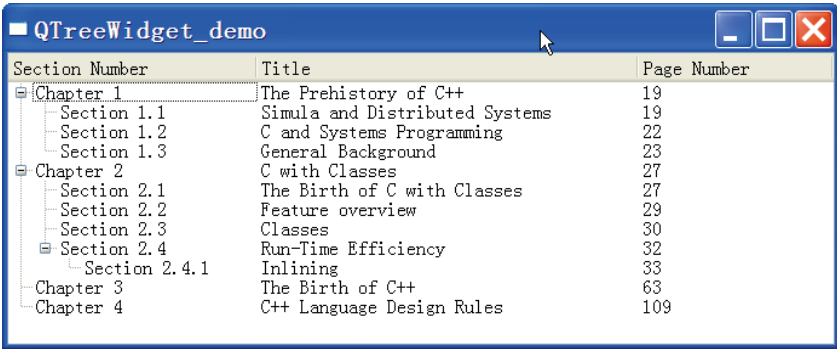
13.2 模型 (Models)

依据数据项之间的关系，模型具有如图 13-6 所示的三种拓扑结构：(1) 列表模型 (List Model)。各数据项相互独立，组成一个线形的序列。比如，一个班级所有学生的姓名就组成一个列表，每个数据项就是一名学生的姓名。(2) 表格模型 (Table Model)。数据项可被划

分为若干行、若干列。比如，一个班级所有学生的成绩信息可被表示为一个表格，其中第一列表示学生姓名，第二列表示学生学号，第三列表示英语课程的成绩，第四列表示数学课程的成绩。每一行表示一个学生的成绩信息。(3) 树模型 (Tree Model)。这种模型具有树状层次结构，每个数据项可以包含若干行、若干列的子数据项，而其中每个子数据项又可以包含更深层的子数据项。

图 13-6 模型的三种拓扑结构

一个树模型的例子如图 13-7 所示。该例子显示的是 C++ 创始人 Bjarne Stroustrup 所著书籍 *The Design and Evolution of C++* 的部分目录。不可见根具有 4 行、3 列子节点，每行表示书中某一章的目录信息，其中第 1 列是该章的编号，第 2 列是该章的题目，第 3 列是该章的起始页码。每行的第一个数据项又含有若干行、若干列子数据项。比如，图中的 “Chapter 1” 含有 “Section 1.1”，“Section 1.2”，“Section 1.3” 所在行的 9 个子数据项。



Section Number	Title	Page Number
Chapter 1	The Prehistory of C++	19
Section 1.1	Simula and Distributed Systems	19
Section 1.2	C and Systems Programming	22
Section 1.3	General Background	23
Chapter 2	C with Classes	27
Section 2.1	The Birth of C with Classes	27
Section 2.2	Feature overview	29
Section 2.3	Classes	30
Section 2.4	Run-Time Efficiency	32
Section 2.4.1	Inlining	33
Chapter 3	The Birth of C++	63
Chapter 4	C++ Language Design Rules	109

图 13-7 具有树状结构的模型

为了将以上三种结构统一起来，Model/View 框架引入了 “不可见根” 的概念。对于列表模型，所有行的数据项都是这个不可见根的子节点。对于表格模型，所有行、所有列的数据项都是这个不可见根的子节点。对于树模型，位于最顶层的那些数据项（可能包含若干行、若干列）是这个不可见根的子节点。引入这个概念后，列表模型、表格模型都可以被看作树模型——仅含有最顶层节点，不含任何子节点的树模型。

Qt 在设计模型类的基类 `QAbstractItemModel` 时，只处理树状结构的模型，并没有单独

考虑列表、表格模型。视图类的基类 `QAbstractItemView` 在访问一个模型中的数据项时，也认为模型具有树状结构。`Model/View` 框架中的其他类，比如只能存放具有列表结构数据项的模型类 `QAbstractListModel` 以及只能以列表方式显示数据项的视图类 `QListView`，都是把列表或者表格看作树的一种特殊形式，它们并没有将列表或者表格当作独立的对象来处理。

使用 `Model/View` 框架时，我们首先需要将所要显示的数据存放在数据项中。除了这些数据外，Qt 令数据项存放更多的数据，比如其中一些数据表示显示数据项时所用的字体、颜色、背景颜色等，视图对象在显示一个数据项时，会读取这些数据。一个数据项所包含的这些数据被称为“数据子项”，它们所起的作用被称为“角色”。13.2.1 节将阐述各个角色对应数据子项的作用。

对于列表或者表格结构的模型，视图对象可以简单地使用行号、列号来访问模型中的某个数据项。但是对于树状结构的模型，仅使用行号、列号无法精确地定位一个数据项。`Model/View` 框架使用“索引”来定位一个数据项，13.2.2 节阐述如何为一个数据项创建索引。

理解了“角色”以及“索引”这两个基本的概念之后，我们就可以尝试使用 `Model/View` 框架了。总体上，我们可以使用以下三种方式来使用该框架。

1. 直接使用便利视图类，比如 13.7 节讨论的 `QListWidget`，`QTableWidget` 或者 `QTreeWidget`。这些视图类在其内部定义了模型对象，程序员只需简单地调用几个函数，就可以建立起数据项之间的拓扑关系，并将所要显示的数据写入到数据项中。

2. 使用便利模型类来存放应用程序的数据，再使用一个视图对象显示这些数据。13.2.5 节讨论了部分便利模型类，比如 `QStringListModel`、`QFileSystemModel`。最简单的情形下，程序员只需调用一两个函数，就可以创建一个完整的模型。

3. 从已有的模型类派生新模型类，将应用程序所要显示的数据定义在这个派生类中。实现基类 `QAbstractItemModel` 中定义的接口，将模型中的数据项呈现给其他视图对象。

虽然前两种方式简单、易于使用，但是无助于我们理解模型类和 `Model/View` 框架中其他类的协作关系。因此，我们将在 13.2.3 节首先阐述上述第 3 种使用方式。这一节是理解 `Model/View` 框架工作原理的钥匙。在读者深刻理解了模型类和其他类的协作关系之后，13.2.4 以及 13.2.5 节阐述上述第 2 种使用方式。

13.2.1 角色与数据子项

模型中的一个数据项会存放多个数据子项，其中一些是应用程序本身需要处理的数据，另外一些是 `Model/View` 框架中其他部分（比如视图对象、委托对象等）需要处理的。我们将一个数据子项所起的作用称为它的“角色”（role），因而一个数据项可被看作多个『角色，数据子项』对组成的集合。

例如，图 13-8 中的程序显示 2011 年世界 10 大新闻。左侧列表显示这些新闻的标题以及图标。用户单击其中一个后，程序在右上方显示该新闻的图片，在右下方以英文显示该新

闻。如果用户将鼠标停留在列表中某个数据项的区域内，程序弹出一个提示框，显示该新闻中文版的文字。

我们使用一个具有列表结构的模型来表示这 10 条新闻，模型中的每个数据项表示一条新闻。新闻的标题（比如图中的“墨西哥海底雕塑”）是一个数据子项，对应的角色为 DisplayRole，视图对象总是显示这个数据子项。新闻的图标是一个数据子项，对应的角色为 DecorationRole。视图对象可能显示这个数据子项，也可能忽略它而只显示 DisplayRole 对应的数据子项。提示框中的中文文字是一个数据子项，对应的角色为 ToolTipRole。只要用户将鼠标停留在一个数据项的区域内，视图对象就会显示这个数据子项。以上角色对应的数据子项和 Model/View 框架密切相关，框架中的其他对象比如视图对象、委托对象等会读取这些数据子项并做相应的处理。应用程序还可以在角色 UserRole 对应的数据子项中存放一些与具体应用相关的数据，例如，本例将一条新闻的英文文字内容存放在这个数据子项中。当用户单击某个数据子项时，应用程序读取这个数据子项，将其中的文字显示在界面右下角的位置。



图 13-8 一个数据项含有不同角色的数据子项

通用的角色如表 13-1 所示，所有的角色名字都被定义在名字空间 Qt 中。所有数据子项都具有类型 QVariant，但是 QVariant 中所存放的数据的类型会因角色的不同而不同。例如，对于角色 WhatsThisRole，这个类型为 QString，表示所要显示的文字，而对于角色 SizeHintRole，这个类型为 QSize，表示数据项的尺寸。当我们向数据子项写入数据时，应该遵循这些约定。

表 13-1 通用的角色以及对应数据子项的功能与类型

常量	对应的数据子项	QVariant 中数据的类型
Qt::DisplayRole	代表一个数据项的文字	QString 或者数值类型
Qt::DecorationRole	代表一个数据项的图标	QColor, QIcon 或 QPixmap
Qt::EditRole	对一个数据项进行编辑时所要处理的数据	QString 或其他

续表

常量	对应的数据子项	QVariant 中数据的类型
Qt::ToolTipRole	鼠标停留在一个数据项区域内时，提示框中所要显示的文字	QString
Qt::StatusTipRole	鼠标停留在一个数据项区域内时，状态栏所要显示的文字	QString
Qt::WhatsThisRole	用户对一个数据项执行“What's This?”命令时所要显示的文字	QString
Qt::SizeHintRole	一个数据项所占据的屏幕尺寸	QSize

和数据项外观相关的角色如表 13-2 所示。通过设置这些数据子项，应用程序可以控制数据项的外观。某些数据项被用来存放布尔类型的值，视图对象在显示 DisplayRole 对应的数据子项的同时，还可以显示一个复选框。角色 CheckStateRole 对应的数据子项表示这个复选框的状态。

表 13-2 和数据项外观相关的角色

常量	对应的数据子项	QVariant 中数据的类型
Qt::FontRole	显示 DisplayRole 对应数据子项时所用的字体	QFont
Qt::TextAlignmentRole	显示 DisplayRole 对应数据子项时所用的文字对齐方式	Qt::AlignmentFlag
Qt::BackgroundRole	显示 DisplayRole 对应数据子项时所用的背景刷子	QBrush
Qt::BackgroundColorRole	过时的角色，应该使用 BackgroundRole	--
Qt::ForegroundColorRole	显示 DisplayRole 对应数据子项时所用的背景刷子	QBrush
Qt::TextColorRole	过时的角色，应该使用 ForegroundColorRole	--
Qt::CheckStateRole	一个数据项的复选状态	Qt::CheckState

13.2.2 索引

在 Qt 的模型/视图框架中，模型类的使用者（比如视图类、委托类）每次只能访问数据集中的一个数据项。这些类总是通过一个索引来指定将要访问哪个数据项。为简单起见，本节仅以视图类为例，来说明如何使用该索引访问数据集中的数据项。Qt 使用类 QModelIndex 表示这种索引，该类含有以下一些重要信息。

1. 行号、列号。被索引数据项的父数据项可能含有多行、多列的子数据项，类 QModelIndex 在其内部定义了一个行号、一个列号，表明被索引数据项在第几行、第几列。如果数据集是一个列表，这个行号可以唯一确定一个数据项在列表中的位置。如果数据集是一个表格，一个行号以及一个列号可以唯一确定一个数据项的位置。

2. 一个指针。如果数据集是一个树或者其他更复杂的数据结构，仅仅靠行号、列号无法确定一个数据项的位置。类 `QModelIndex` 在其内部定义了一个指针，直接指向数据集中的 一个数据项。有的情况下，模型类使用一个容器（比如数组或者 `QVector`）存放数据项，通过指定一个整数类型的容器下标，也可以唯一确定一个数据项的位置，比使用指针更加方便。对于这种情形，`QModelIndex` 将这个指针变量当作一个整数变量来使用，并没有另外定义一个整数变量，以节省内存空间。我们将这个整数称为“内部 ID”（Internal ID）。为简单起见，本节只讨论该变量被用作指针的情形。

一个问题出现了：如果数据集是一个列表或者表格，视图类设置 `QModelIndex` 的行号、列号，即可通知模型类，它要访问哪个数据项。但是，如果数据集是一个树或者其他复杂的数据结构，仅仅设定行号、列号无法确定视图类要访问数据集中的哪个数据项。由于视图类根本不知道模型类采用了什么样子的底层数据结构存放这些数据项，它却又无法设置 `QModelIndex` 中的指针，使得视图类无法精确、直接地设定一个索引。

为了解决这个问题，视图类和模型类约定了一个合作协议。

（1）无论数据集是一个列表、表格、树或者其他数据结构，模型类在内部定义了一个不可见根节点（invisible root），作为列表、表格中所有数据项的父节点，或者树的根节点的父节点。这个不可见根节点并不存放数据集中的任何数据，但是对数据集任何数据项的访问都始于对这个不可见根节点的访问。Qt 约定：一个无效索引（invalid index）指向这个不可见根节点。所谓无效索引，是指 `QModelIndex` 的一个特殊对象，它没有指向数据集中任何一个数据项。类 `QModelIndex` 的无参构造函数返回这样一个对象。

（2）当视图类需要访问列表、表格中的数据项，或者树的根节点时，视图类会将目标数据项的行号、列号、父节点的索引传递给模型类，要求模型类为目标数据项创建一个索引。由于这种情况下目标数据项的父节点是不可见根节点，所以父节点的索引一定是一个无效索引。

如果数据集是列表或者表格，由于视图类传递过来的行号、列号信息已经能够唯一确定目标数据项的位置，模型类只需创建一个 `QModelIndex` 对象，并将传来的行号、列号写入该对象即可，不用设置该对象的指针域。

如果数据集是一棵树，模型类创建一个 `QModelIndex` 对象，将传来的行号、列号写入该对象。树可能含有多个根节点，模型类依据该行号、列号确定目标数据项的位置，并设置 `QModelIndex` 的指针域，令其指向这个目标数据项。

如果数据集是列表或者表格，视图类利用模型类返回的索引，能够访问数据集中的任意一个数据项，两者通过 `QModelIndex` 可以进行畅通无阻的信息交换。如果数据集是树，本步骤只是为树的根节点建立了索引，还需要继续下面的步骤，为其他树节点建立索引。

（3）一棵树可能具有多个根节点，每个根节点可能含有多个子节点。为了求取某个子节点的索引，视图类把该子节点的父节点的索引，该子节点的行号、列号传递给模型类。例如，为了求取图 13-9 中节点 B 的索引，视图类会将 A 的索引，B 的行号（等于 1）、列号（由于

A 只有一列子节点，因而此值为 0）传递给模型类，请求创建 B 的索引。模型类接收到这些信息后，会依据 A 的索引（图中的 QModelIndex 对象）中存放的指针，迅速找到节点 A。再依据 B 的行号、列号确定 B 的位置，然后创建一个 QModelIndex 对象，将 B 的行号、列号、内存地址写入该对象。模型类将 B 的完整索引返回给视图类，视图类利用这个索引完成其他操作，比如获取 B 中的数据，或者继续求取 B 节点的子节点（C、D、E）的索引。沿着从父节点到子节点的方向，不断重复这个过程，可以求取每个树节点的索引。

图 13-9 创建树节点的索引

（4）有些情况下，视图类将一个子节点的索引传递给模型类，请求模型类创建该子节点的父节点的索引。例如，视图类将图 13-9 中子节点 D 的索引传递给模型类，请求模型类创建 B 的索引。模型类依据 D 的索引可以轻易地找到 D。由于是模型类负责整棵树的存放，因而它也可以轻易地找到 D 的父节点 B，创建一个 QModelIndex 对象，令其指针域指向 B。然而，工作尚未做完，模型类还要求解 B 的行号、列号。它会找到 B 的父节点 A，察看 A 含有哪些子节点，并计算 B 的行号、列号，然后将这些信息填入 B 的索引对象中。至此，B 的完整索引才被创建完毕。

13.2.3 派生新模型类

选择合适的基类：QAbstractItemModel 最灵活、复杂。QAbstractListModel and QAbstractTableModel 简单，已经实现了一些功能，取决于数据本身的结构。

1. 最小模型访问接口

类 QAbstractItemModel 定义了模型访问接口，模型类的使用者（比如视图类、委托类）使用这个接口访问数据集。类 QAbstractItemModel 是一个抽象类，模型类的使用者无法直接使用这个类，它们必须使用这个类的派生类来访问数据集。本节介绍如何派生新的模型类。为简单起见，我们令模型类的使用者为视图类。

类 QAbstractItemModel 定义的这个接口包含多个函数，其中 5 个函数被定义为该类的纯虚函数，意味着任何派生类必须实现这 5 个函数，视图类才能和这个派生类协同工作，访问

其中的数据集。这 5 个函数是视图类和模型类之间的最小接口。下面我们分别介绍这 5 个函数的功能。派生类的实现者应该理解并精确实现每个函数的功能。

当视图类企图访问某个数据项时，会调用下面的函数请求模型类创建一个索引，指向目标数据项：

```
QModelIndex index ( int row, int column, const QModelIndex & parent =  
                    QModelIndex() ) const = 0
```

其中 **parent** 是目标数据项父节点的索引。这个父节点可能含有多行、多列子节点，参数 **row**、**column** 表示目标数据项所在的行号、列号。如果 **parent** 为无效索引，则表示目标数据项的父节点为类 **QAbstractItemModel** 内部定义的那个不可见根节点。

当视图类企图访问一个数据项的父节点时，会调用下面的函数请求模型类创建一个索引，指向该父节点：

```
QModelIndex parent ( const QModelIndex & index ) const = 0
```

其中 **index** 是子节点的索引，该函数的返回值应该是父节点的索引。

当视图类企图访问一个数据项的子节点时，会调用下面的两个函数，向模型类咨询这个父节点含有多少行、多少列子节点：

```
int rowCount ( const QModelIndex & parent = QModelIndex() ) const = 0  
int columnCount ( const QModelIndex & parent = QModelIndex() ) const = 0
```

类似地，当 **parent** 为无效索引时，这两个函数返回不可见根节点含有多少行、多少列子节点。

派生类一旦实现了以上 4 个函数，视图类可以获得数据集中任何一个数据项的索引。具体地说，如果数据集是一个列表，视图类调用 **rowCount()**（令参数 **parent** 为无效索引），获得列表中数据项的个数。由于列表的列数被约定为 1，所以无需调用 **columnCount()**。视图类再调用 **index()** 即可获得每个数据项的索引。如果数据集是一个表格，视图类调用 **rowCount()** 以及 **columnCount()**（令参数 **parent** 为无效索引），获得表格含有多少行、多少列数据项，再调用 **index()** 获得每个数据项的索引。

如果数据集是一棵树，情况会复杂些。虽然一般情况下一棵树只具有一个根，每个父节点只含有 1 列子节点，但是我们此处考虑最复杂的情况：一棵树可以具有多行、多列的根节点，每个父节点也可以含有多行、多列的子节点。视图类调用 **rowCount()** 以及 **columnCount()**（令参数 **parent** 为无效索引），获得该树含有多少行、多少列根节点，再调用 **index()** 函数（令参数 **parent** 为无效索引），获得每个根节点的索引。将其中每个根节点作为父节点，调用 **rowCount()** 以及 **columnCount()**，获得每个根节点含有多少行、多少列子节点，再调用 **index()** 函数，获得每个子节点的索引。依此类推，可以逐级获得更深层次节点的索引，直到获得所有节点的索引。给定一个节点，如果视图类想获得其父节点的索引，可以直接调用函数 **parent()**。

视图类一旦获得某个数据项的索引，它就可以调用下面的函数读取某个角色对应的数据子项：

```
QVariant data ( const QModelIndex & index,  
                int role = Qt::DisplayRole ) const = 0
```

其中 `index` 是目标数据项的索引，而 `role` 是表示某个角色的枚举常量。正是通过这个函数，派生类将其管理的数据呈现给视图类。

2. 实现最小接口的例子——满二叉树

如果数据集是一个列表或者表格，虽然我们可以按照前一节所述，直接从 `QAbstractItemModel` 派生新模型类并重载 5 个纯虚函数，但是，从 Qt 提供的类 `QAbstractListModel` 或者 `QAbstractTableModel` 派生新模型类的效率更高，因为这两个类已经实现了 `QAbstractItemModel` 的部分虚函数。13.7 节将介绍如何从这两个类派生新的模型类，以处理具有列表或者表格结构的数据集。

本节介绍如何从 `QAbstractItemModel` 派生新模型类，以处理具有树状层次结构的数据集。虽然 Qt 提供的类 `QStandardItemModel` 能够存放、访问这样的数据集，但是本节没有使用这个类。通过演示如何从 `QAbstractItemModel` 派生新模型类，我们可以深入理解一个模型类应该如何与 Model/View 框架中的其他类协同工作，这有利于我们理解后续小节将要讨论的其他模型类。

我们以一个满二叉树的例子，来介绍如何从 `QAbstractItemModel` 派生新模型类。所谓满二叉树 (full binary tree)，如图 13-10 所示，以通俗的语言来描述，是指除了最底层的节点（被称为叶子节点）之外，每个高层节点都具有 2 个子节点。图中二叉树的每个节点存放着一个整数，每个父节点中的数是它的两个子节点中数的和。

有多种数据结构可被用来存放一棵满二叉树，本例采用如图 13-10 所示的数组来存放。我们按照从上到下、从左到右的顺序，把各节点中的整数存放在数组中。给定一个数组元素，对其下标做简单的运算，即可求出其父节点、两个子节点的位置。具体地说，设数组下标始于 0，如果一个数组元素的下标为 i ，其父节点的下标为 $(i-1)/2$ （此处“/”表示整数除法，即只保留商），其左子节点的下标为 $2*i+1$ ，右子节点的下标为 $2*i+2$ 。例如，图中节点“11”、“5”、“6”的数组下标分别为 5、11、12，满足上述公式。

依据这些公式，我们可以轻易地推出以下结论：

(1) 所有左子节点的下标为奇数，所有右子节点的下标为偶数。

(2) 模型类以若干行、若干列的形式存放一个父节点的子节点。对于二叉树，模型类会先存放其左子节点，再存放其右子节点。因而左子节点对应的行号为 0，右子节点对应的行号为 1。给定一个节点的数组下标 i ，该节点对应的行号为 $(i+1) \% 2$ 。

我们从 `QAbstractItemModel` 派生一个模型类 `TreeModel` 表示这棵满二叉树，该类的声明如代码段 13-1 所示。满二叉树将被存放在该类的成员数组 `numbers` 中。类 `QAbstractItem`

Model 具有多个虚函数，而 TreeModel 仅仅重载了其中 5 个必须重载的纯虚函数。

图 13-10 使用数组存放一棵满二叉树

代码段 13-1, 类 `TreeModel` 的声明, 取自 `z:\examples\mvc\binary_tree\treemodel.h`

```
class TreeModel : public QAbstractItemModel
{
    Q_OBJECT
public:
    TreeModel();
    int rowCount ( const QModelIndex & parent ) const;
    int columnCount ( const QModelIndex & parent ) const;
    QModelIndex index ( int row, int column, const QModelIndex & parent
        = QModelIndex() ) const;
    QModelIndex parent ( const QModelIndex & index ) const;
    QVariant data ( const QModelIndex & index, int role ) const;
private:
    enum {N=15};
    int numbers[N];
};
```

类 `TreeModel` 的实现如代码段 13-2 与代码段 13-3 所示。其构造函数将图 13-10 中的满二叉树存放在成员数组 `numbers` 中。一般情况下, 模型类中的某个父节点可以包含若干行、若干列子节点, 但是对于本例的满二叉树, 一个父节点最多只含有 2 行、1 列子节点。因而, 成员函数 `columnCount()` 总是返回 1。成员函数 `rowCount()` 应该返回一个父节点所含子节点的个数。当父节点的索引是一个无效索引时, 表示该父节点是模型类的不可见根节点; 函数返回 1, 表示该节点只含有一个子节点, 也就是满二叉树的根节点。如果该父节点是满二叉树的非叶节点, 函数返回 2, 表示非叶节点总是含有 2 个子节点。否则, 该父节点是一个叶子节点, 不含任何子节点, 函数返回 0。

代码段 13-2, 类 `TreeModel` 的实现, 取自 `z:\examples\mvc\binary_tree\treemodel.cpp`

```
TreeModel::TreeModel()
{
    int values[N]={36,10,26,3,7, 11,15, 1,2,3,4,5,6,7,8};
    for (int i=0; i<N; i++)
        numbers[i] = values[i];
}
int TreeModel::rowCount ( const QModelIndex & parent ) const
{
    if ( ! parent.isValid() )
        return 1;
    if (parent.internalId() < N/2 )
        return 2;
    return 0;
}
int TreeModel::columnCount ( const QModelIndex & parent ) const
```

```
{
    return 1;
}
```

成员函数 `index()` 返回一个数据项的索引。参数 `parent` 是该数据项父节点的索引。其父节点可能含有多个子节点，参数 `row` 和 `column` 表示该子数据项所在的行号、列号。该函数调用 `QAbstractItemModel` 的成员函数 `createIndex()` 创建一个 `QModelIndex` 对象作为这个子数据项的索引。`createIndex()` 函数的原型如下。

```
QModelIndex QAbstractItemModel::createIndex ( int row, int column, void
* ptr = 0 ) const
QModelIndex QAbstractItemModel::createIndex ( int row, int column, quint32
id ) const
```

该函数将新创建的 `QModelIndex` 对象内部的行号，列号，指针（或者内部 ID）设置为参数中的 `row`、`column`、`ptr`（或者 `id`）。本例调用的是第 2 个 `createIndex()` 函数，内部 ID 表示一个数据项在成员数组中的下标。该下标显然能够确定一个数据项的存储位置，所创建的 `QModelIndex` 对象当然也就是一个数据项的完整索引。

对于本例，如果父节点为模型类的不可见根节点，子数据项一定是二叉树的根节点。它在成员数组中的下标为 0，而其行号、列号由 `index()` 函数的参数 `row`，`column` 指定（应该都为 0），所以我们在行①创建该子数据项的索引。

如果父节点不是模型类的不可见根节点，它就一定是二叉树的一个非叶节点。行②取得它在成员数组中的位置，做一些简单的计算，即可求出子数据项在成员数组中的位置，最后调用 `createIndex()` 创建该数据项的索引。

代码段 13-3，类 `TreeModel` 的实现（续），取自
z:\examples\mvc\binary_tree\treemodel.cpp

```
QModelIndex TreeModel::index ( int row, int column, const QModelIndex &
parent ) const
{
    if ( ! parent.isValid() )
        return createIndex(row, column, 0);           ①
    int parent_idx = parent.internalId();               ②
    int idx = parent_idx * 2 + ( row + 1 );
    return createIndex(row, column, idx );
}
QModelIndex TreeModel::parent ( const QModelIndex & index ) const
{
    if (index.internalId() == 0)                        ③
        return QModelIndex();                          ④
    int parent_idx = (index.internalId() - 1 )/2;
    return createIndex( (parent_idx+1) % 2, 0, parent_idx ); ⑤
}
QVariant TreeModel::data ( const QModelIndex & index, int role ) const
{
    switch (role) {
    case Qt::DisplayRole:
        int value = numbers[ index.internalId() ];
        return QVariant( value );
    }
    return QVariant();
}
```

成员函数 `parent()` 返回一个子数据项父节点的索引。参数 `index` 是这个子数据项的索引。如果这个索引的内部 ID 为 0（行③），则这个子数据项是二叉树的根，其父节点是模型类的不可见根，行④调用 `QModelIndex` 的无参构造函数，创建并返回一个无效索引。如果子数据项不是二叉树的根，行④后的代码做一些简单的计算，求取父节点在成员数组中的位置，创建并返回其索引。这个子数据项的父节点可能是该子数据项的“爷”节点的左子节点或者右子节点，对应的行号可能会是 0 或者 1。行⑤依据该子数据项的父节点在成员数组中的下标确定这个行号。

函数 `data()` 获取一个数据项中的数据。函数参数 `index` 是该数据项的索引，`role` 是所要获取的数据子项的角色。本例只处理角色 `Qt::DisplayRole`，该函数取得该数据项在成员数组中的下标，返回该数据项的数据，也就是二叉树节点中的整数，供视图类显示。对于其他角色，该函数调用 `QVariant` 的无参构造函数，构造一个表示“无效值”的 `QVariant` 对象，并返回给视图类，表明该模型类无法提供与角色 `role` 对应的数据子项，视图类自己应该为角色 `role` 指定一个默认值。比如，如果角色为 `Qt::BackgroundRole`，视图类将会采用默认的白色作为显示数据项时的背景色。

重载了上述 5 个纯虚函数的类 `TreeModel` 就能够和 `Model/View` 框架中其他类协同工作了。由于这个模型类所表示的数据集是一棵树，我们采用类 `QTreeView` 来显示它的数据集，如代码段 13-4 所示。主函数只是简单地调用视图类的成员函数 `setModel()`，建立起模型类和视图类两者之间的关联关系，两者就可以通过上述 5 个纯虚函数组成的最小接口进行信息交互。

代码段 13-4，满二叉树例子的主函数，取自 `z:\examples\mvc\binary_tree\main.cpp`

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TreeModel treeModel;
    QTreeView treeView(0);
    treeView.setModel( & treeModel );
    treeView.show();
    return app.exec();
}
```

本例的模型类 `TreeModel` 只是提供了最基本的功能：当一个视图类需要显示数据集中的数据时，该类向视图类报告数据集的构造，并向视图类返回指定数据项中的数据，以使视图类能够“显示”模型类中的数据集。`Model/View` 框架所具有的功能远比这丰富。在下面几节中，我们将向类 `TreeModel` 中添加新的功能，以展示 `Model/View` 框架的能力。

3. 处理更多的角色

数据集的每个数据项包含多个『角色，数据子项』对，其中一些数据子项存放应用程序的数据，比如二叉树例子中每个数节点中的整数，而另外一些数据子项和这个数据项的显示相关，比如背景色、文字字体。当视图类需要显示某个数据项时，会查询这个数据项所包含的所有『角色，数据子项』对。上一节的模型类 `TreeModel` 仅向视图类报告 `Qt::DisplayRole` 角色对应的数据子项。本节扩展 `TreeModel` 的功能，使其能够向视图类报告更多角色对应的

数据子项，以控制数据项的外观。我们的目标是使用红色、26 磅字体显示叶节点。对于其他非叶节点，采用黑色、20 磅显示。

如代码段 13-5 所示，我们只需修改模型类的 `data()` 函数即可达到这个目标。函数参数 `index` 指向被查询的数据项，`role` 是所要查询的角色。由于我们需要更改某些节点的显示颜色，`TreeModel` 必须处理角色 `Qt::ForegroundRole`（行①）。`index` 的成员函数 `internalId()` 返回 `QModelIndex` 对象的内部 ID。本例中，这个内部 ID 表示当前数据项在模型类的成员数组的位置。行②依据位置判断被查询数据项是否为二叉树的叶节点。如果是，则返回一个红色刷子。否则，表示当前数据项不是叶节点，行④构造并返回一个无效 `QVariant` 对象，视图类将采用默认的黑色显示当前数据项。类似地，如果当前数据项是叶节点，行③后的代码返回 26 磅字体，否则，行④返回 20 磅字体，视图类将使用返回的字体显示当前数据项。

代码段 13-5，能够处理更多角色的模型类，取自
z:\examples\mvc\binary_tree_more_role\treemodel.cpp

```
QVariant TreeModel::data ( const QModelIndex & index, int role ) const
{
    switch (role) {
        case Qt::DisplayRole:{
            int value = numbers[ index.internalId() ];
            return QVariant( value );
        }
        case Qt::ForegroundRole:                                ①
            if ( index.internalId() >= N/2 )                    ②
                return QBrush(Qt::red);
            break;
        case Qt::FontRole:                                      ③
            QFont font;
            if ( index.internalId() >= N/2 )
                font.setPointSize(26);
            else
                font.setPointSize(20);
            return font;
        }
        return QVariant();                                     ④
    }
}
```

4. 数据变化时通知视图类

数据集中的数据项可能自发地发生变化。例如，当一个应用程序从网络上下载文件时，数据集中的某个数据项被用来表示已经下载的字节数。每当一些字节被成功下载，这个数据项会被更新。如果某个视图类负责显示这个数据项，模型类应该通知该视图类这个变化，以使视图类更新这个数据项的显示。

模型类使用 Qt 的信号与槽机制通知视图类。每当模型类中的某个或者某些数据项发生变化时，模型类发出一个 `dataChanged()` 信号，该信号的函数参数表明了哪些数据项发生了变化，其原型为：

```
void QAbstractItemModel::dataChanged ( const QModelIndex & topLeft,
                                         const QModelIndex & bottomRight )
```


当只有一个数据项发生变更时，上述两个函数参数都指向这个数据项。当有多个数据项发生变化，而且这些数据项具有相同的父节点时，`topLeft` 是最左上角那个数据项的索引，而 `bottomRight` 是最右下角那个数据项的索引。例如，当数据集是一个表格时，所有数据项都具有一个共同的父节点（即模型类的不可见根节点）。如果按照从上到下、从左到右的顺序看，有多个连续的数据项发生了变化，函数参数 `topLeft` 以及 `bottomRight` 可以表示哪些数据项发生了变化。

当我们调用某个视图类的 `setModel()` 函数，令其显示一个模型类中的数据集时，该函数会将 `dataChanged()` 信号与视图类中的某个槽函数绑定。一旦模型类发出 `dataChanged()` 信号，视图类中的槽函数就会被调用，该槽函数就会查询 `topLeft` 与 `bottomRight` 之间数据项的取值，并重新显示这些数据项。

如果一个父节点下的多个不连续的子节点发生变化，模型类应该将这些子节点分为多个小的区域，其中每个区域中的子节点是连续的。然后，发出多个 `dataChanged()` 信号，每个信号只涉及其中一个小区域中的子节点。如果发生变化的节点根本不属于同一个父节点，模型类必须发出多个 `dataChanged()` 信号，其中每个信号所涉及的节点必须具有相同的父节点。

为了模拟“数据项自发地发生变化”的情形，我们令图 13-11 中二叉树的最后一个节点（含有数字“8”的那个）每 1 秒钟增加 1，增加到 60 时归零，也就是令这个节点表示当前时间的秒数部分。我们还有意地令这个问题更加复杂：当这个节点发生变化时，从这个节点到二叉树根的所有节点都应该被更新，以满足“父节点中的数字等于子节点中的数字之和”这个规则。也就是说，图中含有数字“15”、“26”、“36”的节点都应该被更新。

图 13-11 二叉树的部分节点自发发生变化

为了及时显示这些自发发生变化的节点，我们扩展了 13.2.3 节 2. 中的代码，得到代码段 13-6。模型类 `TreeModel` 的构造函数创建了一个 `QTimer` 对象，每秒发出一个 `timeout()` 信号，触发该类的槽函数 `timerHit()`。在这个槽函数中，行①后的代码更新二叉树节点中的值，行②后的代码创建这些节点的索引，行③后的代码触发 `dataChanged()` 信号，通知视图类重新显示这些节点的值。

代码段 13-6，显示自身发生变化的数据项，取自
z:\examples\mvc\binary_tree_changing_data\treemodel.cpp

```
TreeModel::TreeModel()  
{
```

```

.....
timer = new QTimer(this);
timer->setInterval(1000);
connect(timer, SIGNAL(timeout()), this, SLOT(timerHit()));
timer->start();
}
void TreeModel::timerHit()
{
    numbers[14] = ( numbers[14] + 1 ) % 60;           ①
    numbers[6] = numbers[14] + numbers[13];
    numbers[2] = numbers[6] + numbers[5];
    numbers[0] = numbers[2] + numbers[1];
    QModelIndex idx_14= createIndex(1,0,14);          ②
    QModelIndex idx_6  = createIndex(1,0,6);
    QModelIndex idx_2  = createIndex(1,0,2);
    QModelIndex idx_0   = createIndex(0,0,0);
    emit dataChanged(idx_14, idx_14);                 ③
    emit dataChanged(idx_6,  idx_6 );
    emit dataChanged(idx_2,  idx_2 );
    emit dataChanged(idx_0,  idx_0 );
}

```

5. 更改标头

模型类的不可见根可能具有多行、多列子节点，其中每行、每列都具有一个标头(header)。当模型类中的数据项太多时，视图类无法在一个窗口显示出所有数据项，用户需要使用视图类提供的水平或者垂直滚动条进行察看。视图类会在每行左侧、每列顶部显示该行或者该列的标头，以使用户了解窗口所显示的数据属于哪行、哪列。默认情况下，模型类把每行的行号加 1，作为该行的标头，把每列的列号加 1，作为该列的标头。之所以加 1，是由于模型类的行号或者列号始于 0，而用户习惯从 1 开始对行或列进行编号。有的情况下，我们希望将标头设置为一些容易记忆、容易理解的字符串。

例如，满二叉树的例子中，模型类 `TreeModel` 的不可见根只有 1 个子节点（即二叉树的根节点），所以默认情况下，水平方向上只含有唯一的标头“1”，如图 13-12 所示。我们希望将其更改为更容易理解的字符串“Full Binary Tree”。这可以通过重载 `QAbstractItemModel` 的虚函数 `headerData()`来实现，如代码段 13-7 所示。

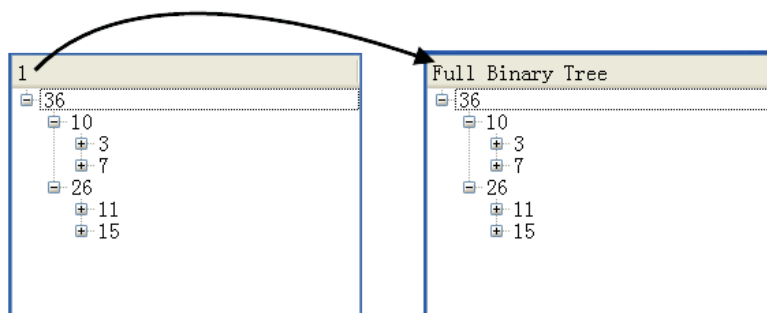


图 13-12 更改标头

视图类在显示标头之前，会调用模型类的这个 `headerData()`函数，向模型类查询每个标头对应的文字。函数参数 `orientation` 表示查询的是水平方向还是垂直方向的标头。当为水平方向时，`section` 表示列号，当为垂直方向时，`section` 表示行号。对于本例，我们仅设置水平

方向第 0 列的标头为“Full Binary Tree”。对于其他情形，比如垂直方向第 0 行，该函数返回一个表示无效取值的 `QVariant` 对象，导致视图类在对应的位置不显示任何标头。

代码段 13-7，更改数据集的标头，取自
`z:\examples\mvc\binary_tree_header\treemodel.cpp`

```
QVariant TreeModel::headerData(int section, Qt::Orientation orientation,
int role) const
{
    if (role == Qt::DisplayRole && section==0
        && orientation == Qt::Horizontal )
        return QString("Full Binary Tree");
    return QVariant();
}
```

6. 编辑数据项

Qt 的 Model/View 框架不但可以将数据集中的数据呈现给用户，它还允许用户编辑每个数据项。并非所有数据项都是可以编辑的，模型类负责确定哪些是可编辑的，哪些是不可编辑的。当用户双击某个数据项试图对其进行编辑时，视图类会调用模型类的成员函数 `flags()`，查询目标数据项是否可编辑。如果是，视图类会调用模型类的 `data()` 函数，获取目标数据项中角色 `Qt::EditRole` 对应的数据子项，送给一个编辑器控件进行编辑。用户完成编辑后，Qt 的 Model/View 框架会调用模型类的 `setData()` 函数，把编辑器中的编辑结果传递给模型类，令模型类修改目标数据项中的数据。

模型类在修改完毕目标数据项的值之后，应该触发一个 `dataChanged()` 信号。Qt 的 Model/View 框架会将这个信号发送给所有与这个模型类绑定的视图对象，以重新绘制目标数据项。最后这个步骤有两个作用。

(1) 如前文所述，委托类负责绘制一个数据项。当用户编辑一个数据项时，编辑器控件会采用不同于委托类的方式显示一个数据项。例如，设目标数据项中存放着一个整数，视图类只会简单地显示这个整数，而编辑器控件使用一个编辑框显示、编辑这个整数，同时还在编辑框的右侧显示一个上箭头、一个下箭头，用户按这两个箭头也可以更改整数的值。当用户完成一个数据项的编辑时，编辑器控件会被关闭，委托类必须重新绘制该数据项。

(2) 如果有多个视图对象与一个模型类对象绑定，某个数据项被更改后，必须通过所有的视图对象，以重新绘制这个数据项，否则，只有被编辑的那个视图显示的是正确的值，其他视图显示的都是较老的值。

当视图类查询一个目标数据项中角色 `Qt::EditRole` 对应的数据子项时，模型类将这个数据子项以类型 `QVariant` 返回。Qt 的 Model/View 框架依据这个 `QVariant` 内部存放的数据的类型，选择适当的编辑器控件对其中的数据进行编辑，程序员不必定义新的编辑器控件。当然，如果程序员不喜欢默认的编辑器控件，当然也可以创建新的编辑器控件。本节演示如何使用默认的编辑器控件对前文所述的满二叉树进行编辑。

我们约定用户只可以编辑满二叉树的叶节点，不可以编辑任何非叶节点。当某个叶节点的值变化时，该叶节点到根节点路径上所有节点的值都应该被更改，以满足约束条件“父节

点的值等于两个子节点值之和”。

如代码段 13-8 所示，模型类 `TreeModel` 重载 `QAbstractItemModel` 的虚函数 `flags()`，向视图类返回一个标志变量，表明某个数据项是否可编辑。当一个数据项是叶节点时，返回的标志变量包含 `Qt::ItemIsEditable`，表明该数据项是可编辑的。如果一个数据项是可编辑的，Qt 的 Model/View 框架会调用 `TreeModel` 的 `data()` 函数，以角色 `Qt::EditRole` 查询对应的数据子项，行①后的代码返回该数据项中存放的整数。Model/View 框架显示一个适当的编辑器，以允许用户对这个整数进行编辑。

代码段 13-8，编辑满二叉树的叶节点，取自
z:\examples\mvc\binary_tree_editable\treemodel.cpp

```
Qt::ItemFlags TreeModel::flags(const QModelIndex & index) const
{
    if ( index.internalId() < N/2 )
        return Qt::ItemIsSelectable | Qt::ItemIsEnabled ;
    else
        return Qt::ItemIsSelectable | Qt::ItemIsEnabled |
            Qt::ItemIsEditable;
}

QVariant TreeModel::data ( const QModelIndex & index, int role ) const
{
    int value;
    switch (role) {
    case Qt::DisplayRole:
        value = numbers[ index.internalId() ];
        return QVariant( value );
    case Qt::EditRole:           ①
        value = numbers[ index.internalId() ];
        return QVariant( value );
    }
    return QVariant();
}

bool TreeModel::setData(const QModelIndex & index, const QVariant & value,
int role)
{
    if (role != Qt::EditRole) return true;
    int id = index.internalId();
    while (1) {           ②
        if ( id >= N/2 )
            numbers[id] = value.toInt();
        else
            numbers[id] = numbers[2 * id + 1] + numbers[ 2 * id + 2];

        QModelIndex idx = createIndex(1,0,id);
        emit dataChanged(idx,idx);    ③
        if ( id == 0 ) break;
        id = ( id - 1 ) /2;
    };
    return true;
}
```

编辑完成后，Model/View 框架会调用 `TreeModel` 的 `setData()` 函数。类型为 `QModelIndex` 的函数参数表示目标数据项的索引，类型为 `QVariant` 的函数参数存放编辑结果，函数参数 `role` 表示要修改目标数据项哪个角色对应的数据子项。程序员在 `QAbstractItemModel` 的派生类中重载这个 `setData()` 函数时，应该返回一个布尔类型的值。返回 `true`，表示该函数成功地修改

了目标数据项的值，返回 `false` 则表示未能成功修改。本例总是返回 `true`。

行②的 `while` 循环语句沿着被编辑叶节点到根节点的路径，逐个修改满二叉树节点中的整数值。每修改一个，在行③触发一个 `dataChanged` 信号，通知视图对象重新绘制被修改的节点。

7. 从 `QAbstractListModel` 及 `QAbstractTableModel` 派生新模型类

如果数据集是一个列表，我们没有必要直接从 `QAbstractItemModel` 派生新模型类，而是应该从该类的子类 `QAbstractListModel` 派生新模型类，因为后者已经替我们实现了部分功能，使得我们只需要做少量的工作，就可以实现一个新模型类。`QAbstractListModel` 是一个抽象类，使用该类的唯一方式是派生新模型类，程序员并不能直接使用该类。

具体地说，新模型类只需重载 `QAbstractListModel` 的 `rowCount()` 以及 `data()` 函数，前者返回列表中元素的个数，后者返回与某个数据项中指定角色对应的数据子项。如果应用程序需要修改列表中的元素，新模型类还应该重载 `flags()` 以及 `setData()` 函数，前者返回一个标志变量，表示模型中的数据集是可编辑的，委托对象就会编辑列表中的数据项。编辑完毕后，就会调用 `setData()` 函数，将数据写回模型。

例如，设新模型类 `ListModel` 内部定义了一个数组 `int numbers[N]`，其中常量 `N` 表示数组元素的个数。为了显示、编辑这个列表，该模型类应该如代码段 13-9 所示那样重载 `QAbstractListModel` 的虚函数。由于列表中数据项的个数是固定的，函数 `rowCount()` 只是简单地返回这个数值。函数 `data()` 中，类型为 `QModelIndex` 的函数参数是列表中某个数据项的索引。由于数据集是一个列表，这个索引内部存放的指针（或内部 ID）、列号没有任何作用，该索引内部存放的行号就可以唯一地确定列表中的某个数据项。函数 `flags()` 返回的标志变量包含 `Qt::ItemIsEditable`，表明用户可以编辑列表中的数据项。用户完成编辑后，新模型类的 `setData()` 函数会被调用，类型为 `QModelIndex` 的函数参数是被编辑数据项的索引，该索引内部存放的行号表示目标数据项在列表中的序号。

代码段 13-9，重载 `QAbstractListModel` 的虚函数以显示、编辑一个列表，取自 `z:\examples\mvc\QAbstractListModel_demo\ListModel.cpp`

```
ListModel::ListModel(QObject *parent) :QAbstractListModel(parent)
{
    for (int i=0;i<N;i++)
        numbers[i] = i;
}
int ListModel::rowCount(const QModelIndex & /*parent*/) const
{
    return N;
}
QVariant ListModel::data(const QModelIndex &index, int role) const
{
    if (role == Qt::DisplayRole) {
        int idx = index.row();
        return QVariant( numbers[idx] );
    }
    return QVariant();
}
Qt::ItemFlags ListModel::flags(const QModelIndex & index) const
```



```

{
    return Qt::ItemIsSelectable | Qt::ItemIsEnabled | Qt::ItemIsEditable;
}
bool ListModel::setData(const QModelIndex & index, const QVariant & value,
int role)
{
    if (role != Qt::EditRole) return true;
    int idx = index.row();
    numbers[idx] = value.toInt();
    return true;
}

```

如果数据集是一个表格，我们也没有必要直接从 `QAbstractItemModel` 派生新模型类，而是应该从该类的子类 `QAbstractTableModel` 派生新模型类。具体步骤与从 `QAbstractListModel` 派生新模型类的步骤类似，读者可以参考随书光盘 `z:\examples\mvc\QAbstractTableModel_demo` 下的例子。

13.2.4 QStandardItemModel

类 `QAbstractItemModel`，`QAbstractListModel`，`QAbstractTableModel` 不保存数据，用户需要从这些类派生出子类，并在子类中定义某种数据结构来保存数据。与此不同，类 `QStandardItemModel` 负责保存数据，每个数据项被表示为类 `QStandardItem` 的对象。我们首先阐述如何使用类 `QStandardItem` 保存一个数据项，再阐述如何使用类 `QStandardItemModel` 将这些数据项组织起来，形成列表、表格或者树，以供其他视图类显示。

如前文所述，一个数据项由若干个『角色，数据子项』对组成。类 `QStandardItem` 负责保存、访问这些数据。该类的内部定义了一个类型为 `QVector` 的容器，每个容器元素本质上存放一个『角色，数据子项』对。

由于各个角色对应的数据子项可能具有不同的类型，Qt 使用 `QVariant` 来存放每个数据子项。当用户希望将一些数据存放在一个 `QStandardItem` 对象中时，可以调用其成员函数：

```
void setData ( const QVariant & value, int role)
```

将『role, value』对存入。当用户希望读取该对象中的数据时，可以调用另外一个成员函数：

```
QVariant data ( int role = ) const
```

读取角色 `role` 对应的数据子项。

以上两个函数是 `QStandardItem` 的核心。有了这两个函数，我们就可以访问该类所表示数据项的任何一个『角色，数据子项』对。然而，对于一些常用角色，该类提供了更加简洁、容易记忆的成员函数。例如，当一个数据项被显示在视图中时，它往往包含一些文字、一个图标，还可能包含一个复选框。角色 `Qt::BackgroundRole` 控制显示背景，`Qt::FontRole` 控制文字字体，`Qt::ForegroundRole` 控制文字颜色，`Qt::CheckStateRole` 控制复选框的状态。该类提供的一组成员函数可以方便地访问这些常用角色对应的数据子项。成员函数 `setBackground()`、`background()` 分别设置/返回背景刷子。函数 `setFont()`、`font()` 分别设置/返回文字字体。函数

setForeground()、foreground()分别设置/返回字体颜色。函数 setCheckState()、checkState()分别设置/返回复选框状态。

类 QStandardItemModel 将类 QStandardItem 表示的数据项组织起来，形成列表、表格、树甚至更复杂的数据结构。该类提供了一组成员函数，向这些数据结构添加新的数据项，更改已经存在的数据项，或者删除已有的数据项。另一方面，作为一个模型类，它实现了 QAbstractItemModel 定义的接口函数，以使其他视图类能够访问模型中的数据项。

如果数据集被表示为一个列表，我们可以调用类 QStandardItemModel 的成员函数 appendRow()向列表中添加一个数据项，使用 item()读取一个数据项，如代码段 13-10 所示。行①获取模型最顶层的根节点，行②创建一个 QStandardItem 对象，表示一个数据项，行③将该数据项作为根节点的子节点添加到列表中。行②的构造函数在内部调用该类的 setData()函数，将行②的 QString 对象作为 Qt::DisplayRole 对应的数据子项存入新构造的对象。由于数据集本身是一个列表，所以我们使用 QListView 显示该数据集，读者可以运行该例子查看显示结果。

代码段 13-10，使用 QStandardItemModel 处理列表，取自
z:\examples\mvc\QStandardItemModel_demo\main.cpp

```
QStandardItemModel listModel;  
QStandardItem *rootItem = listModel.invisibleRootItem();  
for (int row = 0; row < 4; ++row) {  
    QStandardItem *item = new QStandardItem(QString("%0").arg(row) );  
    rootItem->appendRow( item );  
}  
QListView listView;  
listView.setModel ( & listModel );
```

如果数据集被表示为一个表格，可以调用类 QStandardItemModel 的成员函数 setItem()设定表格中的某个数据项，如代码段 13-11 所示。由于这个代码段中的数据集是一个表格，所以使用 QTableView 显示该数据集。

代码段 13-11，使用 QStandardItemModel 处理表格，取自
z:\examples\mvc\QStandardItemModel_demo\main.cpp

```
QStandardItemModel tableModel(4, 4);  
for (int row = 0; row < 4; ++row) {  
    for (int column = 0; column < 4; ++column) {  
        QStandardItem *item = new QStandardItem(  
            QString("%0,%1").arg(row).arg(column));  
        tableModel.setItem(row, column, item);  
    }  
}  
QTableView tableView;  
tableView.setModel( & tableModel );
```

如果数据集被表示为一个树，可以调用类 QStandardItemModel 的成员函数 appendRow()向某个树节点添加子节点。通过多次调用该函数，可以构建一棵复杂的树。代码段 13-12 构建一棵简单的树：最顶层的根节点有一个文字内容为“0”的子节点，该子节点有一个文字内容为“1”的子节点。依此类推，“1”子节点有一个“2”子节点，“2”子节点有一个“3”子节点，形成一棵深度为 4 的树。这棵树的每个节点都没有兄弟节点（具有相同父节点的多

个节点被相互称为兄弟节点)，感兴趣的读者可以修改这段代码，以使其中某些节点具有兄弟节点。由于数据集是一棵树，我们使用 QTreeView 显示它。

代码段 13-12，使用 QStandardItemModel 处理树，取自
z:\examples\mvc\QStandardItemModel_demo\main.cpp

```
QStandardItemModel treeModel;
QStandardItem *parentItem = treeModel.invisibleRootItem();
for (int i = 0; i < 4; ++i) {
    QStandardItem *item = new QStandardItem(QString("%0").arg(i));
    parentItem->appendRow(item);
    parentItem = item;
}
QTreeView treeView;
treeView.setModel( & treeModel );
```

类 QStandardItemModel 之所以能够表示列表、表格、树甚至更复杂的数据结构，得益于类 QStandardItem 在其内部定义了一个类型为 QVector<QStandardItem*>的容器，可以将每个容器元素所指的 QStandardItem 对象设定为子对象。表现在如图 13-13 所示的类图上，类 QStandardItem 和自身具有“children”关系。一个类和自身发生关联，在 UML 中被称为自关联（self association）。类 QStandardItemModel 定义了一个名为 root 的数据成员，逻辑上是一个指向 QStandardItem 对象的指针。这个对象可以设定多个 QStandardItem 的对象作为自己的子对象，而其中每个子对象又可以包含其他的子对象。依此类推，这棵树可以具有任意深度，每个父对象可以包含任意多个子对象。

图 13-13 类 QStandardItem 使用自关联以形成层次结构

很自然地，QStandardItemModel 可以使用 QStandardItem 表示具有树状数据结构的数据集，如图 13-14 所示。图中的每个小方框表示类 QStandardItem 的一个对象。如果小方框的边线为虚，相应的 QStandardItem 对象并不表示数据集中的任何数据，仅被用来表示某种数据结构。如果小方框的边线为实，相应的 QStandardItem 对象就表示数据集中的一个数据项。在右侧的图中，QStandardItemModel 的数据成员 root 所指的 object 表示一个不可见的根，而数据集的根（图中结点 G）被表示为这个不可见根的一个子节点。

图 13-14 类 QStandardItem 使用自关联形成列表、表格与树

列表被看作一个特殊的树：不可见根具有若干个子节点，每个子节点表示列表中的一个数据项，不再包含任何子节点，如该图左侧所示。而表格的表示方式反而麻烦一些。不可见根含有若干子节点（图中 A，B，C），这些子节点并不表示数据集中的任何数据项。第 i 个子节点会包含若干子节点（比如图中 D，E，F），这些子节点才表示表格第 i 行的数据项。

使用 QStandardItemModel 表示数据集具有以下优点：该类使用 QStandardItem 存放数据项，用户不必定义任何数据结构来存放数据项；QStandardItem 使用自关联关系，能够表达列表、表格、树甚至更复杂的数据结构，能够涵盖各种各样的数据集；QStandardItem 本身存放着多个『角色，数据子项』，视图类、委托类或者其他用户定义的类能够方便地依据角色访问各个数据子项。

然而，这种表示方法也有局限性：当数据集中的数据项很多时，施加在数据集上的某些操作的执行效率会很低。比如，设数据集是一个 1 万行、20 列的表格，其中第 10 列存放的是浮点数。如果我们想计算这一列的平均值，按照图 13-14，这需要遍历所有行，取得第 10 列的 QStandardItem 对象，再依据角色 “Qt::DisplayRole” 取得对应的数据子项。由于这个数据子项的类型为 QString，还需要将其转换为浮点数，最后求所有浮点数的平均值。这些操作会耗费较长的时间。

因此，对于数据量不是很大、对性能要求不是很高的场合，我们可以使用类 QStandardItemModel 来表示一个数据集。否则，用户应该从 QAbstractItemModel、QAbstractListModel 或者 QAbstractTableModel 派生新类，自行管理数据集的存放与访问。

13.2.5 便利模型类

对于一些特殊的数据集，Qt 的 Model/View 框架提供了专门的模型类来处理它们。这些模型类虽然也是 QAbstractItemModel 的派生类，但是它们的使用方式非常简单，根本不需要程序员派生任何新的模型类，因此，我们将它们称为便利模型类（convenience model class）。

Qt 提供了以下便利模型类：QStringListModel，处理数据项为字符串的列表；QFileSystemModel，处理具有树状层次结构的本地文件系统； QSqlQueryModel 及其子类 QSqlTableModel、QSqlRelationalTableModel，负责使用 SQL 语句访问数据库。由于最后这几

个类涉及数据库操作，比较复杂，本节不讨论它们。下面我们仅阐述 `QStringListModel` 以及 `QFileSystemModel` 的用法。

类 `QStringListModel` 处理数据项为字符串的列表，其使用方式如代码段 13-13 所示。程序员不需要派生任何新模型类，只需要像行①那样，将一个 `QStringList` 对象作为参数，构造一个 `QStringListModel` 对象。由于新对象已经重载了必要的虚函数，实现了 `QAbstractItemModel` 定义的接口，因而立即可以和 `QListView` 对象协同工作。

代码段 13-13，类 `QStringListModel` 的使用，取自
z:\examples\mvc\QStringListModel_demo\main.cpp

```
QStringList list;  
list << "One" << "Two" << "Three" << "Four" << "Five";  
QStringListModel listModel(list);           ①  
QListView listView;  
listView.setModel( &listModel );  
listView.show();
```

类 `QFileSystemModel` 负责处理具有树状层次结构的本地文件系统，其使用方式如代码段 13-14 所示。运行该例子，本地文件系统的信息会被显示在一个窗口中，所显示的信息包括文件的大小、类型、日期信息。每个目录可以被展开/折叠。

当该类的对象刚刚被创建时（行①），新对象并不会立即开始扫描本地文件系统。只有当该类的 `setRootPath()` 函数被调用时（行②），新对象才会启动一个独立的线程扫描整个文件系统，视图类才能够获得并显示文件系统的信息。`setRootPath()` 函数的另外一个功能是创建一个监视器（是类 `QFileSystemWatcher` 的对象），对该函数参数所指定的路径进行监视。一旦该路径下的文件或者目录被删除、更改或者该路径下出现了新的文件、子目录，该监视器会通知模型类，模型类会通知视图类以更新显示。

代码段 13-14，便利模型类 `QFileSystemModel` 的用法，取自
z:\examples\mvc\file_system\main.cpp

```
QFileSystemModel model;           ①  
model.setRootPath("C:/Documents and Settings"); ②  
QTreeView treeView;  
treeView.setModel(&model);  
treeView.show();
```

13.3 视图 (Views)

在 Qt 的 Model/View 框架中，视图类在总体上负责绘制视图，处理用户的交互命令。对于含有较多数据项的视图，用户可以使用滚动条浏览到所有的数据项。对于具有树状层级结构的数据集，视图对象可以折叠/展开其中每个父节点。用户可以选择某个或者某些数据项，显示一个上下文相关的菜单或者进行拖曳操作。所有这些交互命令都由视图对象完成。

然而，视图类并不负责每个数据项的绘制以及编辑工作，这些工作将被“委托”给委托类，13.5 节将讨论相关内容。每个视图对象被创建时，都会指向一个默认的委托对象。这个委托对象将负责绘制视图对象中的数据项。当用户需要编辑其中某个数据项时，这个委托对

象会创建一个合适的编辑器，负责编辑目标数据项。编辑完成后，委托对象会将编辑结果写回模型的目标数据项中。由于这个默认委托对象的存在，程序员在使用一个视图对象时，可以不提供任何委托对象，方便了视图对象的使用。

Model/View 框架提供了以下视图类。

(1) `QAbstractItemView`，是所有视图类的抽象基类。当 Model/View 框架中其他视图类无法满足我们的要求时，可以派生该类的子类。

(2) `QListView`、`QTableView`、`QTreeView`、`QColumnView`，分别以列表，表格，树，多级列表的形式显示模型中的数据集。13.2 节已经讨论了前三种视图类的使用，本节将介绍 `QColumnView` 的使用。

(3) `QHeaderView`，和前面几个视图类不同，这个视图类只负责显示视图的标头部分。它调用模型类的 `headerData()` 函数，读取标头文字内容并将其显示在每列的上部或者每行的左侧。

对于具有树状层次结构的数据集，我们既可以像 13.2.3 节那样使用 `QTreeView` 来显示，也可以使用 Model/View 框架提供的另外一个视图类 `QColumnView` 来显示。`QColumnView` 使用一个列表显示树中某个父节点的子节点。起始时，`QColumnView` 对象只将树的最顶层节点显示在一个列表中。当用户单击其中一个节点时，视图对象在已有列表的右侧创建一个列表，显示被单击节点的所有子节点。依次类推，新列表不断被创建，其中第 i 个列表显示第 $i-1$ 个列表中被单击节点的所有子节点。

例如，使用一个 `QColumnView` 对象显示本地文件系统的情形如图 13-15 所示。第一列显示驱动器名称（其中有些是使用 Windows 命令 `subst` 生成的虚拟驱动器），当用户选择“D:\”时，该驱动器下的目录、文件被显示在第二列。当用户选择其中的“Qt”时，该目录下的子目录被显示在第三列。当目录层次很深时，会创建很多个列表。当视图对象无法在整个窗口中显示所有列表时，会在窗口下方显示一个水平滚动条，以使用户能够浏览所有列表。

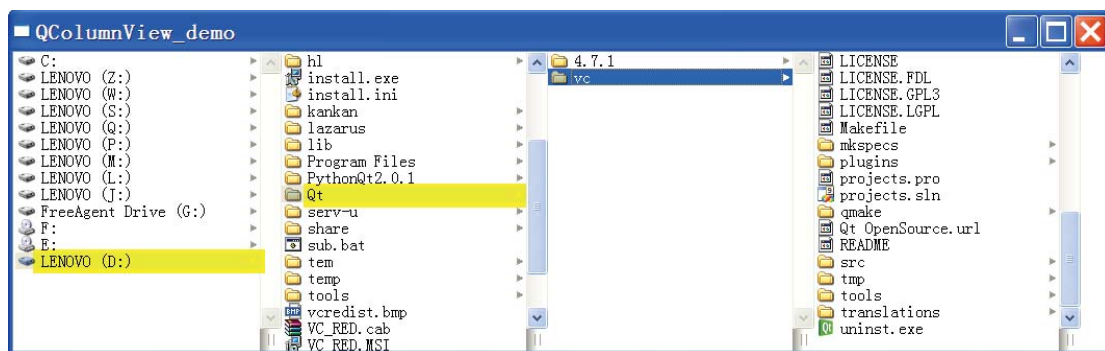


图 13-15 用 `QColumnView` 对象显示本地文件系统

虽然看起来 `QColumnView` 的功能很强大，但是与其他视图类相似，它的使用方式却很简单，如代码段 13-15 所示。程序员并不需要告诉一个 `QColumnView` 对象需要显示多少个列表，它在运行时会查询模型对象。只要用户选择的数据项具有子节点，这个视图对象就会创建新的列表。

代码段 13-15, 用 QColumnView 对象显示本地文件系统, 取自
z:\examples\mvc\QColumnView_demo\main.cpp

```
QFileSystemModel model;  
model.setRootPath("C:/");  
QColumnView *cview = new QColumnView;  
cview->setModel(&model);
```

13.4 选择操作

视图类允许用户使用键盘或者鼠标选择某个或者某些数据项。视图类提供三种选择模式:

(1) 单选模式。用户最多只能选择一个数据项。

(2) 多选模式。用户可以选择多个数据项。图 13-16 显示了一个具有 8 行、4 列的数据集。按照从上到下、从左到右的阅读顺序, 某些被选中的数据项是相邻的, 比如图中 (2, 2) 到 (3, 3) 之间的 6 个数据项, 它们形成一个“选择块”。选择块可以包含多个数据项, 也可以只包含一个数据项, 比如图中的 (8, 4)。多选模式下, 一个视图可能包含多个选择块, 这些选择块互不相邻。

(3) 扩展模式。一个视图只允许有一个选择块, 用户只能在这个选择块的末尾选择新的数据项以扩展现有的选择块, 不可以选择一个和当前选择块不相邻的数据项。

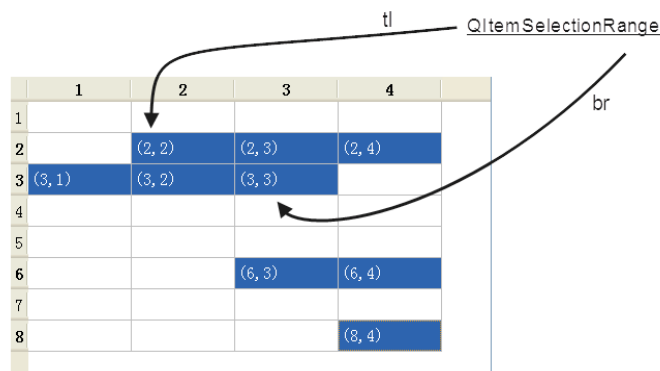


图 13-16 多选模式与 QItemSelectionRange 的作用

Qt 使用类 QItemSelectionRange 描述每个选择块, 其数据成员 tl 是选择块中最左上角数据项的索引, br 是最右下角数据项的索引。该类的成员函数 indexes() 返回该选择块中所有数据项的索引。如图 13-17 所示 (是图 13-5 “Qt 的 Model/View 框架” 的一部分), 数据成员 tl、br 的类型为 QPersistentModelIndex, 意味着即使相关的数据集发生了变化, 比如其中部分数据项被删除或者新增添了数据项, tl、br 仍然分别指向选择块的最左上角、最右下角数据项。

如果有多个 QPersistentModelIndex 类型的索引指向同一个数据集中的若干个数据项, 每当这个数据集发生任何更改, 就需要维护所有这些索引, 代价较大。一般场合下, Model/View 框架中的其他类并不需要使用这种类型的索引, 类型为 QModelIndex 的索引就能够满足它们的要求。因此, 类 QPersistentModelIndex 重载了一个类型转换运算符, 将一个 QPersistentModelIndex 对象转换为一个 QModelIndex 对象。QItemSelectionRange 的成员函数 indexes() 就调用了这个函数, 以 QList<QModelIndex> 类型返回一个选择块中所有数据项的索引。

图 13-17 Model/View 框架中与数据项选择相关的类

类 `QItemSelection` 描述由多个选择块组成的一个集合。在多选模式下，用户最近一次选择的数据项信息被记录在一个 `QItemSelection` 对象中。这些信息被称为“当前选择”（`current selection`），类 `QItemSelectionModel` 使用一个名为“`currentSelection`”的指针指向这个对象。在最近一次选择操作之前，视图对象已有的选择信息也被表示为一个 `QItemSelection` 对象，这些信息被称为“已有选择”，类 `QItemSelectionModel` 使用一个名为“`ranges`”的指针指向这个对象。

本书所讨论的 Qt 版本虽然在 `QItemSelectionModel` 的实现层面上做了“当前选择”与“已有选择”的划分，但是 `QItemSelectionModel` 并没有将这种划分呈现给使用该类的用户。例如，该类的公有成员函数 `selectedIndexes()` 合并这两个部分，将合并结果返回给调用者，调用者无法区分哪些数据项属于“当前选择”，哪些数据项属于“已有选择”。

类 `QItemSelectionModel` 除了管理“当前选择”与“已有选择”之外，还负责监视选择信息的变更。一旦有变更，会触发诸如 `selectionChanged()` 这样的信号，通知视图对象及时更新显示。每个视图对象的内部定义了一个指针，指向一个 `QItemSelectionModel` 对象，以记录该视图所显示的数据项的选择状态信息。当多个视图对象显示同一个数据集中的数据项时，这些视图对象可以指向同一个 `QItemSelectionModel` 对象，使得用户在某个视图所做的选择操作会被立即同步到其他视图中。

类 `QItemSelectionModel` 并不处理与选择操作相关的键盘或者鼠标事件，这由视图对象负责。当用户执行了某些选择操作，比如用鼠标圈选了一些数据项，视图对象会判断这个操作涉及哪些数据项，然后通知该视图对象所指的 `QItemSelectionModel` 对象执行相应的选择（或者取消选择）操作。`QItemSelectionModel` 对象更改这些数据项的选择状态，触发一个 `selectionChanged()` 信号，通知指向这个对象的所有视图对象更新显示。

13.4.1 获取与监视选择范围

视图类将多个数据项呈现给用户后，用户常常会选择其中一些数据项，并要求应用程序对这些数据项施加某种操作。这种情形下，应用程序应该能够查询到哪些数据项被选中了。

每个视图对象包含一个 `QItemSelectionModel` 对象，调用后者的 `selectedIndexes()` 成员函数，可以返回上述信息。

有些情形下，应用程序需要监视用户的选择操作。每当用户选择或者取消选择某些数据项时，应用程序立即会执行某些动作。例如，用户在使用 Windows 操作系统的资源管理器时，屏幕下方的状态栏会显示被选中文件的个数、总字节数。每当用户选择或者取消选择某些文件，状态栏中的数据会立即更新。每个视图对象所包含的 `QItemSelectionModel` 对象负责监视用户的选择操作。每当用户选择或者取消选择某些数据项，该对象会触发一个 `selectionChanged()` 信号。应用程序把这个信号与某个槽函数绑定，就可以及时处理用户的选择或者取消选择操作。

我们举一个具体的例子来说明如何读取与监视用户所做的选择或者取消选择操作。如图 13-18 所示，该例子显示一个 8 行、4 列的表格。初始时所有数据项内容为空。每当用户选择某个或者某些数据项，每个数据项的行号、列号会被显示在该数据项所在位置。每当用户取消选择某个或者某些数据项，这些数据项中的内容会恢复为空。窗口的标题显示有多少个数据项被选中，每当用户执行一个选择或者取消选择操作，这个标题会被立即更新。

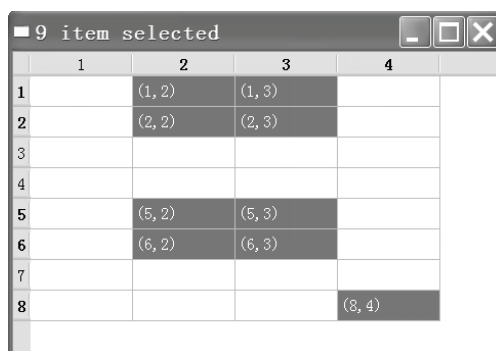


图 13-18 读取与监视选择范围

例子中的窗口由类 `MainWindow` 实现，其声明如代码段 13-16 所示。类型为 `QStandardItemModel` 的成员负责管理例子中 8 行、4 列的数据项，类型为 `QTableView` 的成员负责显示这些数据项。该类并没有定义一个 `QItemSelectionModel` 对象，因为将来创建 `QTableView` 对象时，必定会有这样一个对象被创建。该类定义了一个槽函数 `updateSelection()`，该槽函数将与 `QItemSelectionModel` 对象的 `selectionChanged()` 信号绑定，以处理用户的选择或者取消选择操作。

代码段 13-16，类 `MainWindow` 的声明，取自
z:\examples\mvc\selection_monitoring\mainwindow.h

```
class MainWindow : public QMainWindow{
    .....
private slots:
    void updateSelection(const QItemSelection &selected, const
        QItemSelection &deselected);
private:
    QStandardItemModel * model;
    QTableView * tableView;
};
```

类 `MainWindow` 的实现如代码段 13-17 所示。其构造函数将 `QItemSelectionModel` 对象的 `selectionChanged()` 信号与类 `MainWindow` 的槽函数 `updateSelection()` 绑定。一旦用户选择或者取消选择了一些数据项，该槽函数会被执行。该函数的参数 `selected` 表示用户刚刚选择了哪些数据项，参数 `deselected` 表示用户刚刚取消选择了哪些数据项。调用这两个参数的成员函数 `indexes()`（行①、③），可以获得对应数据项的索引的列表，该列表的类型为 `QModelIndexList`，等价于 `QList<QModelIndex>`。由于是一个 `QList`，所以行②、④可以使用 `foreach` 关键字遍历列表。

代码段 13-17，类 `MainWindow` 实现，取自
z:\examples\mvc\selection_monitoring\mainwindow.cpp

```
MainWindow::MainWindow(QWidget *parent, Qt::WFlags flags)
: QMainWindow(parent, flags)
{
    model      = new QStandardItemModel(8, 4, this);
    tableView = new QTableView(this);
    tableView->setModel(model);
    QItemSelectionModel * selectionModel = tableView->selectionModel();
    connect(selectionModel, SIGNAL( selectionChanged(QItemSelection,
        QItemSelection) ),
            this,          SLOT ( updateSelection (QItemSelection,
        QItemSelection) ) );
    resize(600,400);
    setCentralWidget(tableView);
}

void MainWindow::updateSelection(const QItemSelection &selected, const
QItemSelection &deselected)
{
    QModelIndexList items = selected.indexes();           ①
    QModelIndex index;
    foreach (index, items) {                             ②
        QString text = QString("(%1,%2)")
            .arg( index.row() + 1 ).arg( index.column() + 1 );
        model->setData(index, text);
    }
    items = deselected.indexes();                       ③
    foreach (index, items)                               ④
        model->setData(index, "");

    QItemSelectionModel * selectionModel = tableView->selectionModel();
    QModelIndexList whole_list = selectionModel->selectedIndexes(); ⑤
    setWindowTitle( QString("%1 item selected")
        .arg( whole_list.size() ) );
}
```

以上调用的类 `QItemSelection` 的成员函数 `indexes()` 只能返回用户刚刚选择或者取消选择的数据项的索引。为了返回所有被选择数据项的索引，行⑤调用视图对象所包含的 `QItemSelectionModel` 对象的 `selectedIndexes()` 函数。

13.4.2 选择信息的同步

Qt 的 Model/View 框架允许多个视图对象显示同一个模型类对象中的数据，这样，用户可以在不同的视图对象中查看数据集中不同部分的数据项。在这种情形下，用户往往希望

在某个视图对象中选择了一些数据项之后，其他视图对象也应该把这些数据项显示为选中状态。也就是说，无论用户在哪个视图对象中执行了选择或者取消选择操作，其他视图对象也应该立即更新数据项的选择状态，以确保所有视图中同一个数据项的选择状态是一致的，这被称为选择信息的同步。

Qt 的 Model/View 框架没有把数据项的选择状态直接存放在视图对象中，而是使用独立的类 `QItemSelectionModel` 专门维护数据项的选择信息，这种设计很容易实现选择信息的同步：我们只需要令多个视图对象共享同一个 `QItemSelectionModel` 对象，即可轻易地实现选择信息的同步。

如代码段 13-18 所示，行①、②调用两个视图对象的 `setModel()` 函数，令它们显示同一个模型类对象中的数据项。每当一个视图对象的 `setModel()` 函数被调用时，会有一个 `QItemSelectionModel` 对象被创建。而且，这个 `QItemSelectionModel` 对象的信号 `selectionChanged()` 会与视图对象的同名槽函数 `selectionChanged()` 绑定。一旦选择信息发生变化，视图对象会及时检测到这个变化并做相应的处理。

行③调用函数 `selectionModel()` 获取其中一个视图对象 `v2` 中的 `QItemSelectionModel` 对象，再调用另外一个视图对象 `v1` 的 `setSelectionModel()` 函数，令 `v1` 也使用这个 `QItemSelectionModel` 对象。当 `setSelectionModel()` 函数被调用时，`QItemSelectionModel` 对象的信号 `selectionChanged()` 会和 `v1` 的同名槽函数 `selectionChanged()` 绑定。得益于 Qt 的信号与槽机制，同一个信号可以和多个槽函数绑定，因此，一旦选择信息发生变化，同一个信号可以触发两个视图对象中的槽函数，使得两个视图对象都可以更新显示，以反映最新的选择信息。

代码段 13-18，同步两个视图对象中的选择信息，取自
z:\examples\mvc\sync_selection\main.cpp。

```
QStringList numbers;  
numbers << "One" << "Two" << "Three" << "Four" << "Five";  
QStringListModel model(numbers);  
QListView v1,v2;  
v1.setModel( &model );           ①  
v2.setModel( &model );           ②  
v1.setSelectionModel( v2.selectionModel() );           ③
```

每个视图对象在内部使用一个指针指向一个 `QItemSelectionModel` 对象。有的读者疑惑：行③令 `v1` 中的这个指针指向 `v2` 对应的 `QItemSelectionModel` 对象，那么 `v1` 中原先这个指针所指的 `QItemSelectionModel` 对象似乎成为一个“孤立对象”，也就是说，不再有指针指向它，这是否会导致内存泄露？是否应该在行③之前先析构这个“孤立对象”？

得益于 `QObject` 的良好设计，以上问题不会发生。当应用程序创建 `QObject` 或其派生类的对象时，如果指定了这个对象的父对象，那么应用程序不必显式地析构这个对象。当父对象被析构时，这个对象会被自动析构。本例中的所有对象都是 `QObject` 对象。行①调用 `setModel()` 函数时会创建一个 `QItemSelectionModel` 对象，其父对象被设置为视图对象 `v1`。因而，当整个程序退出时会析构 `v1`，届时就会析构 `v1` 对应的那个“孤立对象”。

13.5 委托 (Delegates)

Qt 的 Model/View 框架中，委托类负责显示数据项，创建与管理编辑器对象，以对数据项进行编辑。具体地说，在显示一个模型对象中的数据项时，虽然视图对象负责绘制总体结构（比如树状层次结构、标头等），但是它不会负责绘制每个数据项。每个视图对象总会指向一个默认的委托对象，视图对象把绘制数据项的任务“委托”给这个默认的委托对象。

除此之外，当用户双击某个数据项，希望对其进行编辑时，视图对象并不会亲自处理这个任务，它会请求委托对象创建一个编辑器。委托对象依据被编辑数据项的数据类型，创建并返回一个编辑器。这个编辑器将负责目标数据项的编辑任务。编辑完成后，委托对象负责将编辑器中的数据写回模型对象的目标数据项之中。

由于一个视图对象中所显示的数据项具有多种类型，视图对象可能需要为每一个数据项创建一个独立的编辑器。视图对象在其内部定义了一个容器保存这些编辑器，以便用户第二次编辑某个数据项时，能在这个容器中找到并直接使用对应的编辑器，不用为其再创建一个。

对于大多数应用程序，默认的委托对象能够胜任数据项的绘制任务，它所创建的编辑器也能胜任数据项的编辑任务，这些应用程序无需创建新类型的委托对象。而且，这个默认的委托对象会调用模型类的 `data()` 函数，读取一个数据项中角色 `FontRole`、`TextAlignmentRole`、`TextColorRole` 以及 `BackgroundColorRole` 对应的数据子项，以控制数据项的外观，因而提供了一定程度的灵活性。

然而，当我们需要彻底地控制数据项的绘制，或者需要使用新的编辑器来编辑某些数据项时，就需要创建新类型的委托对象。Model/View 框架提供了三个委托类：抽象基类 `QAbstractItemDelegate` 定义了所有委托类的接口，子类 `QStyledItemDelegate` 使用应用程序当前的界面风格（参见 15.3 节）绘制数据项的外观，另外一个子类 `QItemDelegate` 仅适用固定的风格绘制数据项外观。由于后者显示的风格很可能和应用程序中其他控件的风格不一致，本书采用的 Qt 版本不推荐使用它，我们也应该从 `QStyledItemDelegate` 派生新的委托类。

通过调用一个视图对象的 `setItemDelegate()` 函数，我们可以令这个视图对象指向一个新委托类的对象。新委托类可以重载成员函数 `paint()` 以及 `sizeHint()`，以新的方式绘制数据项。它也可以重载成员函数 `createEditor()`，创建新类型的编辑器对数据项进行编辑。下面，我们举一个具体例子来说明如何定义新的委托类。

这是 Qt 软件包中的一个例子，我们对其做了细微的改动，存放在目录 `z:\examples\mvc\spinboxdelegate` 下。该例子的主界面如图 13-19 所示，它以表格方式显示一些数字。用户双击其中某个数据项，可以对该数据项进行编辑。默认情况下，Model/View 框架使用一个简单的编辑框对其中的数据进行编辑。本例采用一个 `SpinBox` 控件对数据进行编辑，该控件包含一个编辑框以及两个小箭头。用户单击上（下）箭头时，编辑框中的数字将被增加（减少）。

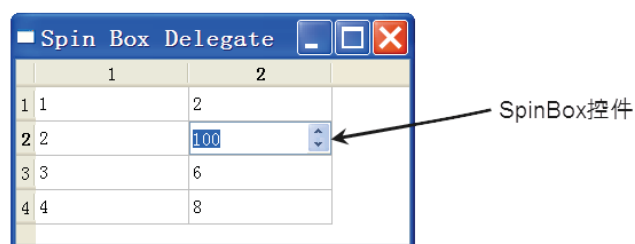


图 13-19 例子 SpinBox 的主界面

该例子中各个类之间的协作关系如图 13-20 所示。程序使用 `QStandardItemModel` 存放表格中的数据项，用 `QTableView` 显示这些数据项。程序派生了 `QStyledItemDelegate` 的子类 `SpinBoxDelegate`，该类被设置为 `QTableView` 的委托，负责每个数据项的显示。当用户双击某个数据项时，`QTableView` 调用 `SpinBoxDelegate` 的成员函数 `createEditor()`，请求创建一个能够编辑这个数据项的编辑器。本例中，`SpinBoxDelegate` 创建一个 `QSpinBox` 对象作为目标数据项的编辑器，返回给 `QTableView`。`QTableView` 将返回的编辑器登记到内部定义的一个容器中，以便用户第二次编辑同一个目标数据项时，直接使用这个已经登记的编辑器，不用再创建一个新的编辑器。

图 13-20 例子 SpinBox 中各个类之间的协作关系

编辑器被创建之后，委托类的成员函数 `setEditorData()` 会被调用，对编辑器所要编辑的数据设置一个初始值。具体地说，委托类依据目标数据项的索引，调用模型类的成员函数 `data()`，读取目标数据项的值，再调用编辑器（本例中的 `QSpinBox`）的成员函数 `setValue()`，设置编辑器的数据。用户完成编辑后，委托类调用 `QSpinBox` 的成员函数 `value()`，读取编辑器的数据，再调用模型类的成员函数 `setData()`，将数据写回模型类。

模型对象、视图对象、委托对象是在主函数中被创建的，如代码段 13-19 所示。行①调用视图对象的成员函数 `setItemDelegate()`，令该视图对象不再使用默认的委托对象，而是使用类 `SpinBoxDelegate` 的对象。

代码段 13-19，例子 SpinBox 的主函数，取自
z:\examples\mvc\spinboxdelegate\main.cpp

```
int main(int argc, char *argv[])
{
    .....
    QStandardItemModel model(4, 2);
    QTableView tableView;
    tableView.setModel(&model);
    SpinBoxDelegate spinbox dg;
    tableView.setItemDelegate(&spinbox_dg);           ①
    for (int row = 0; row < 4; ++row) {
        for (int column = 0; column < 2; ++column) {
            QModelIndex index = model.index(row, column, QModelIndex());
            model.setData(index, QVariant((row+1) * (column+1)));
        }
    }
    .....
}
```

类 `SpinBoxDelegate` 的实现如代码段 13-20 所示。其成员函数 `createEditor()` 创建一个编辑器，类型为 `QModelIndex` 的函数参数是将被编辑的数据项的索引。一般情况下，该函数应该读取目标数据项中角色 `EditorRole` 对应的数据子项，并依据这个数据子项的类型决定创建什么类型的编辑器。对于本例，由于所有数据项都是数值类型，该函数无须访问目标数据项，总是返回一个能够编辑数值的 `QSpinBox` 对象。

代码段 13-20，类 SpinBoxDelegate 的实现，取自
z:\examples\mvc\spinboxdelegate\delegate.cpp

```
QWidget *SpinBoxDelegate::createEditor(QWidget *parent, const
QStyleOptionViewItem & /* option */,
                                const QModelIndex & /* index */) const
{
    QSpinBox *editor = new QSpinBox(parent);
```

```

        editor->setMinimum(0);
        editor->setMaximum(100);
        return editor;
    }
    void SpinBoxDelegate::setEditorData(QWidget *editor, const QModelIndex
    &index) const
    {
        int value = index.model()->data(index, Qt::EditRole).toInt();
        QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
        spinBox->setValue(value);
    }
    void SpinBoxDelegate::setModelData(QWidget *editor, QAbstractItemModel
    *model,
                                   const QModelIndex &index) const
    {
        QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
        spinBox->interpretText();
        int value = spinBox->value();
        model->setData(index, value, Qt::EditRole);
    }
    void SpinBoxDelegate::updateEditorGeometry(QWidget *editor, const
    QStyleOptionViewItem &option,
                                   const QModelIndex & /*
    index */) const
    {
        editor->setGeometry(option.rect);    ①
    }

```

如前文所述，成员函数 `setEditorData()` 读取目标数据项的数据，初始化编辑器，而成员函数 `setModelData()` 读取编辑器中的值，将其写回目标数据项中。当用单击并拖曳图 13-19 中两列标头之间的分隔线时，可以更改数据项的外观尺寸。视图对象会将被编辑数据项的尺寸信息封装在 `QStyleOptionViewItem` 中，通知委托对象更改与这个数据项关联的编辑器的外观尺寸。代码段 13-20 中的成员函数 `updateEditorGeometry()` 将被调用。行①令编辑器占据目标数据项所占的整个区域。

13.6 代理模型（Proxy Models）

有些情况下，应用程序需要对模型中的数据集做一些处理之后，再交给视图对象显示。例如，一位人事部经理在使用一个电子邮件处理程序时，需要在过去几年接收的成千上万封求职信中，寻找一份曾经被拒、但是目前急需的人才的求职信。由于他面对的求职者太多，所以只是记得这封求职信的接收时间大约是 2010 年秋季。对发送者姓名、邮件主题也只有些模糊的记忆。这位用户想让这个邮件处理程序筛选出 2010 年 9 月到 2010 年 12 月期间的邮件，显示在两个视图中。在其中一个视图，用户依据发送者姓名进行排序，以快速过滤掉一些发送者（比如已在公司就职的员工）。在另外一个视图，用户可以输入其他筛选条件，筛选主题中含有某个单词的邮件。

筛选 2010 年 9~12 月期间邮件的这个操作由模型或者视图对象来完成都不合适。如果由模型来完成，这个筛选操作会影响所有与这个模型关联的视图对象，使得所有视图只能看到上述期间的邮件，无法看到其他时间段的邮件。如果由视图对象来完成，每个视图对象都需要执行这个操作，耗费时间，操作结果无法被其他视图对象共享。另外，在 `Model/View`

框架中，视图对象的主要职责是显示而非处理数据。

由于以上原因，Qt 使用一个单独的代理模型 (proxy model) 负责对源模型 (source model) 中的数据项进行处理，再将处理结果呈现给视图对象。代理模型具有双重职责：首先，它重载了 `QAbstractItemModel` 定义的模型接口，因而其他视图对象可将其看作一个普通模型；另外，它在内部定义了一个指针，指向一个源模型，以对该模型中的数据项进行处理。代理模型并不直接处理源模型中的数据项（本节称它们为“源数据项”），而是将它们的索引映射为代理模型的索引，得到一些“虚拟数据项”。和这个代理模型关联的其他视图对象显示的正是这些虚拟数据项。所谓“虚拟”，是指代理模型并不实际存储这些数据项，而是通过虚拟数据项索引与源数据项索引之间的映射关系，逻辑上拥有一些数据项。采用这种方式，可以避免重复存放源模型的数据集，节省空间，也会提高程序运行速度。

代理模型向外界呈现一个处理过的数据集，但是并不真正地更改源模型中的数据集。其他视图对象仍然可以直接访问源模型中的原始数据集。另外，当被看作一个普通模型时，代理模型可以和多个视图对象关联，令这些视图对象共享代理模型中的虚拟数据集，而不用每个视图对象都要对数据处理一次。

Qt 的 Model/View 框架定义了抽象基类 `QAbstractProxyModel` 来表示代理模型。直接派生该类的子类，我们可以对源数据项实现任何类型的处理。这种方式虽然足够灵活，但需要我们处理许多实现细节，13.6.1 节详细阐述了这种方式。如果我们只需做排序或者筛选操作，可以使用 Model/View 框架定义的类 `QSortFilterProxyModel`，该类能够将源模型某一列的数据作为关键字，对源模型的所有行进行排序。该类还允许程序员设定一个正则表达式，该类可以将这个表达式施加到源模型的某一列，对源模型的所有行进行筛选操作，13.6.2 节给出了一个具体例子。

13.6.1 派生 `QAbstractProxyModel` 的子类

`QAbstractProxyModel` 是一个抽象类，程序员不能直接定义该类的对象，而是应该派生该类的子类，实现对其他模型的代理。该类在其内部定义了一个指针，指向被代理的模型。我们将派生的子类称为“代理模型” (proxy model)，而把被代理的模型称为“源模型” (source model)。成员函数 `setSourceModel()` 设置代理模型中的这个指针，指向一个源模型，而成员函数 `sourceModel()` 返回这个指针。

对源模型的代理本质上就是建立代理模型的索引与源模型的索引之间的映射关系。`QAbstractProxyModel` 定义了 2 个成员函数表达这种映射关系。成员函数 `mapToSource()` 将代理模型的索引映射为源模型的索引，而成员函数 `mapFromSource()` 将源模型的索引映射为代理模型的索引。这两个函数是纯虚函数，`QAbstractProxyModel` 派生类必须实现它们。

定义 `QAbstractProxyModel` 的派生类时，需要完成以下两项任务。

(1) 重载纯虚函数 `mapToSource()` 以及 `mapFromSource()`，定义代理模型索引与源模型索引之间的映射关系。

(2)实现 `QAbstractItemModel` 的最小接口。由于 `QAbstractProxyModel` 是 `QAbstractItemModel` 的子类,从前者派生出来的代理模型就必须实现所有模型都应该实现的最小接口,即 `index()`, `parent()`, `rowCount()`, `columnCount()`以及 `data()`。

在定义普通模型时,我们十分清楚采用什么样的数据结构来存放数据集,也十分清楚如何定义 `QModelIndex` 中的内部指针(或者内部 ID)以令其指向某个数据项。然而,在定义代理模型时,由于这个模型中的数据集只是逻辑上存在的,或者说是虚拟的,在定义这个模型对应的索引以及上述成员函数时,情况会变得复杂、抽象。接下来我们以一个具体的例子来说明如何定义代理模型。

我们将要定义的代理模型 `RevertProxyModel` 能够对任意一个源模型做如下映射:给定源模型中的任意一个父节点,设该节点有 N 行子节点,则源模型中第 i 行子节点被映射为代理模型中该父节点的第 $N-1-i$ 行子节点。也就是说,代理模型颠倒源模型中子节点的行顺序。例如,设源模型为 13.2.3 节 3.中的一棵满二叉树,如图 13-21 左侧所示。经代理模型 `RevertProxyModel` 映射后的满二叉树如该图右侧所示,每个父节点所拥有的子节点的顺序都被颠倒。

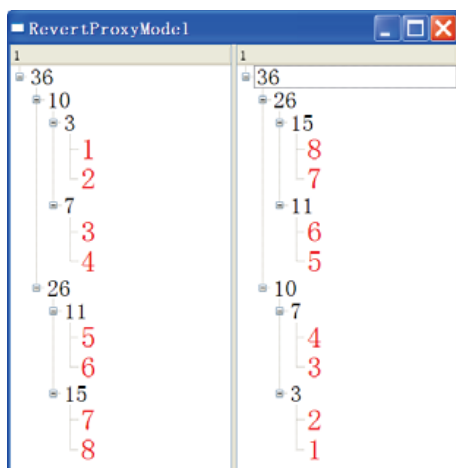


图 13-21 源模型以及代理模型中的满二叉树

如图 13-22 所示,从视图对象的角度来看,代理模型和普通模型没有什么区别。视图对象会调用接口函数 `index()`, `parent()`, `rowCount()`, `columnCount()`以及 `data()`来访问代理模型中的数据集。如 13.2.2 节所述,在这些函数中, `index()`函数最重要,它负责为模型中的某个数据项创建一个完整的索引。所谓“完整”是指视图对象仅使用这个索引,就能够访问模型中的目标数据项,不再需要任何其他额外信息。一旦视图对象获得某个目标数据项的索引,它就可以调用其他接口函数轻易地获得目标数据项的其他信息。因此,设计一个代理模型的首要任务是定义它所使用的索引。

代理模型以及源模型都使用类 `QModelIndex` 表示索引。给定一个 `QModelIndex` 对象,只有当我们知道它是哪个模型对应的索引时,才能够确定它所指的目标数据项。通常情况下,单独的一个 `QModelIndex` 对象不能唯一地确定一个目标数据项。

每个 `QModelIndex` 对象具有 3 个主要的数据成员:行号、列号、内部指针(或内部 ID)。对于源模型,行号、列号表示目标数据项在其父节点所包含的所有子节点中的位置,而内部

指针（或内部 ID）指向目标数据项（物理地或逻辑地）。对于代理模型，如图 13-22 中的虚线框所示，虽然从视图对象的角度来看，代理模型的某个父节点 PP（表示 Proxy-model Parent）也含有若干个数据项，但是这些数据项只是源模型中对应数据项的映射，代理模型根本不会在物理上存放这些数据项，因而我们将它们称为“虚拟数据项”。源模型中的一个数据项和其对应的虚拟数据项很可能具有不同的行号、列号。例如，设图 13-22 左下角实线方框中的 4 行、4 列是源模型中父节点 SP（表示 Source-model Parent）的数据项。经过 RevertProxyModel 的映射后，这些数据项会被映射为虚线方框中的虚拟数据项。数据项 SC（表示 Source-model Child）在源模型中的位置是第 2 行、第 1 列（行号和列号总是从 0 开始的）。映射后，对应的虚拟数据项 PC（表示 Proxy-model Child）位于第 1 行、第 1 列。因此，对于代理模型，QModelIndex 中的行号、列号表示一个虚拟数据项在代理模型中的行号、列号，而不是源模型中的行号、列号。类似地，在本节中，只要所讨论的内容与模型相关，我们应该明了这些内容是针对代理模型的，还是针对源模型的。



图 13-22 例子 RevertProxyModel 中代理模型类对索引的映射过程

如何定义代理模型对应 QModelIndex 对象的内部指针（或内部 ID）是本例最重要的部分。由于代理模型只是在逻辑上包含了一些数据项，它根本不会在物理上存放这些虚拟数据项，所以我们无法令这个内部指针（或内部 ID）指向某个虚拟数据项。但是，给定一个虚拟数据项，源模型中存在一个数据项与之对应，将其称为“源数据项”。我们可以令上述指针指向这个源数据项的索引。

例如，在图 13-22 中，设源模型父节点 SP 具有 N 行数据项（图中 N 为 4）。代理模型第 i 行、第 j 列虚拟数据项对应的索引如图中右上角所示，QModelIndex 对象的行号、列号就是 i、j。这个虚拟数据项对应的源数据项在源模型中的行号为 N-1-i、列号为 j，它的索引如图中右下角灰色矩形所示。代理模型索引中的内部 ID 逻辑上指向源数据项的这个索引。本例中，我们使用一个类型为 QVector 的容器 vector 存放所有源数据项的索引，上述内部 ID 是源数据项索引在这个容器中的序号。本节将这个容器称为“索引池”。

显然，给定一个虚拟数据项，只有当索引池中存在着对应源数据项的索引时，我们才能够为这个虚拟数据项构建一个完整的索引。当视图对象需要访问代理模型中的某个虚拟数据项 PC 时，会首先调用代理模型的成员函数 index()，请求代理模型为这个虚拟数据项构建一

个索引，index()函数的原型如下：

```
QModelIndex RevertProxyModel::index(int row, int column, const QModelIndex & proxy_parent) const
```

其中 proxy_parent 是目标虚拟数据项的父节点 PP 的索引。这个父节点可能包含多个子节点，row 和 column 是目标虚拟数据项的位置。

为了完成这个任务，我们首先要求解代理模型中的这个父节点所对应的源模型中的那个节点 SP。再求解源模型中 SP 所包含子节点的行数 N，那么目标虚拟数据项 PC 就与源模型中 SP 节点的第 N-1-row 行、第 column 列的那个子节点 SC 对应。调用源模型的 index() 函数就可以轻易地求出子节点 SC 的索引。将这个索引添加到索引池，并令目标虚拟数据项的内部 ID 指向这个新索引，就可以建立目标虚拟数据项 PC 的完整索引。

这个求解过程看起来很容易，但是稍加分析就会发现一个问题：我们如何求解 PP 对应的 SP。有的读者也许会回答：PP 的索引（上述函数参数 proxy_parent）的内部 ID 应该指向索引池中 SP 的索引，依靠 SP 的索引我们不就可以找到 SP 吗？可这里的问题是：此时 SP 的索引是否已被添加到了索引池？

一种可能的做法是：在代理模型响应视图对象的第一个请求之前，首先构造一个囊括源模型中所有数据项索引的索引池。这个方案虽然能够解决刚才的问题，但是却存在着性能问题：如果源模型是一个数据项很多、层次很深的树（比如一个庞大的文件系统），那么建立这个索引池既费时间又费空间。

实际上，我们所担心的问题根本不会发生。也就是说，我们可以确保上述 SP 的索引已经存在于索引池，原因如下：依照 13.2.2 节所述的协议，视图对象总会从代理模型的不可见根开始，以逐层深入的方式，请求代理模型构建虚拟数据项的索引。代理模型接收到这些请求后，也会以逐层深入的方式，请求源模型构建相关源数据项的索引，并将这些索引添加到索引池。按照这样的方式，当视图对象请求代理模型构建 PC 的索引时，SP 的索引已被添加到索引池。

有了上述设计，我们就可以编写出代理模型 RevertProxyModel 的成员函数 index()，如代码段 13-21 所示。该函数调用了其他两个成员函数 mapToSource() 以及 register_index()。给定代理模型中一个虚拟数据项的索引，函数 mapToSource() 负责返回对应源数据项的索引。代理模型以及源模型都使用无效索引作为它们的不可见根节点的索引，因而如果给定虚拟数据项的索引是一个无效索引，该函数也返回一个无效索引，表示代理模型的不可见根被映射为源模型的不可见根。如果给定的索引是一个普通索引，行①求取 QModelIndex 对象的内部 ID，并在索引池中查找并返回对应源数据项的索引。给定一个源数据项的索引，成员函数 register_index() 判断索引池是否已有这个索引。如果没有，则将其添加到索引池中。无论哪种情况，函数返回该索引在索引池中的位置。

代码段 13-21，代理模型索引的创建，取自
z:\examples\mvc\revertProxyModel\revertProxyModel.cpp

```
QModelIndex RevertProxyModel::mapToSource ( const QModelIndex & proxy_index ) const
```

```

{
    if (!proxy_index.isValid())
        return QModelIndex();
    int pos = proxy_index.internalId();           ①
    return vector[ pos ];
}
int RevertProxyModel::register_index(const QModelIndex & source_index)
const
{
    int pos = vector.indexOf( source_index );
    if ( pos == -1 ){
        vector.push_back( source_index );
        pos = vector.size() - 1;
    }
    return pos;
}
QModelIndex RevertProxyModel::index(int row, int column, const QModelIndex & proxy_parent) const
{
    QModelIndex source_parent = mapToSource( proxy_parent );
    int rowCount = sourceModel()->rowCount( source_parent );
    QModelIndex source_index = sourceModel()->index( ②
        rowCount - 1 - row, column, source_parent );
    int pos = register_index( source_index );
    return createIndex( row, column, pos );
}

```

成员函数 `index()` 负责为一个虚拟数据项 PC 构建索引。该数据项在代理模型中的位置由该函数的 3 个参数确定：`proxy_parent` 是其父节点 PP 的索引。这个父节点可能包含多个子节点，`row` 和 `column` 表示 PC 的位置。

函数 `index()` 首先调用 `mapToSource()`，求解 PP 所对应的源数据项 SP 的索引。再调用源模型的 `rowCount()` 函数求解 SP 所包含子节点的行数 `rowCount`。与 PC 对应的那个源数据项 SC 就应该是 SP 的第 `rowCount-1-row` 行 第 `column` 列的那个节点。行②调用源模型的 `index()` 函数，求取 SC 的索引，并调用 `register_index` 函数将其登记到索引池。最后，也是最重要的一步，调用基类 `QAbstractItemModel` 的成员函数 `createIndex()`，构造一个 `QModelIndex` 对象，令其内部 ID 指向 SC 的索引。至此，一个指向目标虚拟数据项 PC 的完整索引构造完毕。

代理模型 `RevertProxyModel` 的成员函数 `rowCount()`，`columnCount()` 分别负责求取代理模型中某个虚拟数据项 PP 所含子节点的行数，列数。对于本例，PP 和对应 SP 所含的行数、列数相等，因而这两个函数的实现比较简单，如代码段 13-22 所示。

代码段 13-22，代理模型 `RevertProxyModel` 的其他 2 个接口函数，取自
z:\examples\mvc\revertProxyModel\revertProxyModel.h

```

int RevertProxyModel::rowCount(const QModelIndex & proxy_parent) const
{
    QModelIndex source_parent = mapToSource(proxy_parent);
    int rowCount = sourceModel()->rowCount( source_parent );
    return rowCount;
}
int RevertProxyModel::columnCount(const QModelIndex & proxy_parent) const
{
    QModelIndex source_parent = mapToSource(proxy_parent);
    return sourceModel()->columnCount( source_parent );
}

```


RevertProxyModel 的成员函数 parent() 负责求取代理模型中某个虚拟数据项 PC 的父节点 PP 的索引。该函数的实现如代码段 13-23 所示。代理模型只利用索引池存放虚拟数据项和源数据项的映射关系，它根本就没有存放各个虚拟数据项之间的父子关系信息。因此，我们必须将这个目标虚拟数据项 PC 映射到源模型中，求取其父节点 SP，再将其映射回代理模型。代码段中的成员函数 mapFromSource() 完成最后这个映射操作。给定一个源数据项的索引，该函数将其添加到索引池中（如果此前尚未添加），创建一个属于代理模型的索引，令其内部 ID 指向新添加的索引，如行③所示。

代码段 13-23，代理模型 RevertProxyModel 的接口函数 parent()，取自
z:\examples\mvc\revertProxyModel\revertProxyModel.cpp

```
QModelIndex RevertProxyModel::mapFromSource ( const QModelIndex &
source_index ) const
{
    if (!source_index.isValid())
        return QModelIndex();
    int rowCount = sourceModel()->rowCount( source_index );
    int pos = register_index( source_index );
    return createIndex( rowCount -1 - source_index.row(),      ③
                        source_index.column(), pos );
}
QModelIndex RevertProxyModel::parent(const QModelIndex & proxy_child)
const
{
    QModelIndex source_child = mapToSource( proxy_child );
    QModelIndex source_parent = sourceModel()->parent( source_child );
    return mapFromSource( source_parent );
}
```

由于所有的代理模型都应该定义成员函数 mapToSource() 以及 mapFromSource()，负责虚拟数据项索引与源数据项索引之间的映射，所以 QAbstractProxyModel 将这两个函数定义为纯虚函数，要求所有派生类必须重载它们。

本例中的 RevertProxyModel 并没有重载 QAbstractItemModel 的接口函数 data()，这是由于 QAbstractProxyModel 已经实现了该函数，如代码段 13-24 所示。Qt 在实现类 QAbstractProxyModel 时用到了第 8 章中的 d-pointer 技术，该代码段中的“d->model”表示该类所指的源模型。行④将虚拟数据项的索引映射到源模型中，调用源模型的 data() 函数，读取并返回指定角色对应的数据子项。

代码段 13-24，接口函数 data() 的实现，取自
q:\src\gui\itemviews\qabstractproxymodel.cpp

```
QVariant QAbstractProxyModel::data(const QModelIndex &proxyIndex, int
role) const
{
    Q_D(const QAbstractProxyModel);
    return d->model->data( mapToSource( proxyIndex ), role );      ④
}
```

类 RevertProxyModel 的数据成员 vector 表示索引池，该成员的声明为：

```
mutable QVector<QModelIndex> vector;
```

被声明为 mutable 的原因如下：类 QAbstractItemModel 中将 5 个接口函数定义为常量函数。

总体上，这个做法是合理的，因为对于一般的模型，这 5 个接口函数 `index()`、`data()`、`parent()`、`rowCount()` 以及 `columnCount()` 只会查询模型中数据集的一些信息，即不会修改数据集中的数据，也不需要修改模型中的其他数据成员。但是，对于本例中的代理模型 `RevertProxyModel`，部分接口函数比如 `index()` 会修改 `vector`。由于这些接口函数在基类 `QAbstractItemModel` 中已经被定义为常量函数，`RevertProxyModel` 也只能将它们定义为常量函数。为了使其中一些常量函数仍然能够修改 `vector`，我们必须将 `vector` 定义为 `mutable`。

另外，随书光盘中本例的源代码包含一些调试语句，能够向一个日志文件输出各个接口函数被调用的顺序以及各函数的执行过程。读者可以浏览生成的日志文件，印证本节所讲述的内容。

13.6.2 QSortFilterProxyModel 的应用

前一节介绍了如何直接派生 `QAbstractProxyModel` 的子类，以将源模型中的数据集映射为另外一个虚拟的数据集。这种方法虽然足够灵活，但是需要程序员实现两个数据集索引的映射，并且需要重载多个接口函数。如果这种映射只是对源模型中的数据项进行过滤以及排序，我们可以使用 Qt 提供的代理模型 `QSortFilterProxyModel`。

该类采用了与前一节类似的思路，实现代理模型与源模型索引之间的映射，重载多个接口函数以使其他视图对象能够访问代理模型中的虚拟数据集。在实现索引之间的映射时，该类考虑了过滤与排序两种操作。关于过滤操作，该类的成员函数 `setFilterRegExp()` 设置一个过滤条件，成员函数 `setFilterKeyColumn()` 将这个过滤条件施加到源模型的某一列，使得只有部分行的源数据项被映射到代理模型，其他行的数据项被过滤出局。关于排序操作，该类实现了抽象基类 `QAbstractItemModel` 定义的接口函数 `sort()`，也即

```
void QSortFilterProxyModel::sort(int column, Qt::SortOrder order)
```

在这个函数中，该类构建源数据项索引与虚拟数据项索引之间的映射关系，以使得映射后的数据项按照第 `column` 列排为升序或者降序（由参数 `order` 指定）。

使用 `QSortFilterProxyModel` 时，程序员需要调用与过滤相关的成员函数，设置过滤条件。但是，程序员并不需要直接调用与排序相关的成员函数，因为视图对象知道所有模型都提供上述接口函数 `sort()`，因而当用户单击视图某列标头要求排序时，视图对象会自动调用 `QSortFilterProxyModel` 的 `sort()` 函数。

还有另外一种方法对源模型中的数据项进行排序：令源模型自己实现这个 `sort()` 函数，然后令一个视图对象直接显示这个源模型。当用户需要排序时，视图对象会直接调用源模型的 `sort()` 函数对数据项进行排序。然而，这种方法会更改源数据项的物理存放位置，不像代理模型的 `sort()` 函数那样，只是对虚拟数据项进行排序。

下面我们将使用 Qt 提供的一个例子来演示 `QSortFilterProxyModel` 的作用。该例子能够对一组电子邮件的摘要信息进行过滤、排序。这个例子的输出如图 13-23 所示。每条电子邮件的摘要信息包括邮件主题（`subject`）、发送者（`sender`）以及发送时间（`date`）。这些数据项

组成源模型的数据集，被显示在窗口上方。窗口中间部分显示代理模型中的数据项，用户单击某列的标头，视图将依据该列中的数据，将代理模型中的各行数据排为升序或者降序。连续地单击将在升序和降序之间切换。用户可以使用三种格式输入过滤条件：正则表达式、通配符或者固定的字符串，其中正则表达式的表达能力最强，本节仅讲述这种格式。关于正则表达式，读者可参考文献[25]及[26]。界面下方和过滤相关，用户在第一行输入过滤条件，在第二行选择过滤条件的格式，在第三行选择这个过滤条件施加于哪列。图中用户的选择表示：对“Sender”列施加一个正则表达式“Andy|Grace”，表示只要源模型某行的“Sender”列出现字符串“Andy”或者“Grace”，该行就会被代理模型接收。

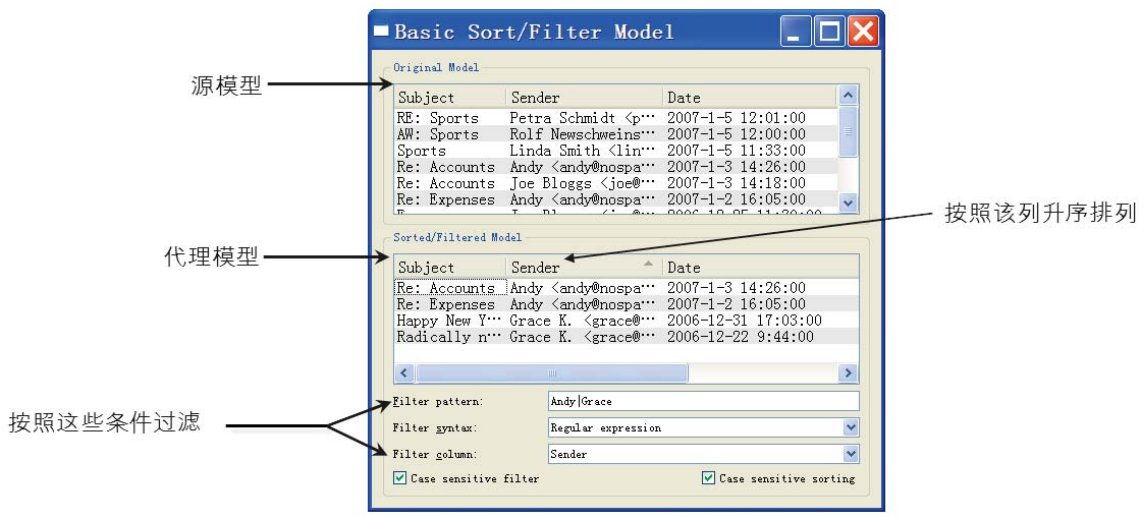


图 13-23 类 QSortFilterProxyModel 的应用

本例中的主要类如图 13-24 所示。该例的主函数 main() 创建一个 QStandardItemModel 对象，作为 QSortFilterProxyModel 的源模型。QWidget 的子类 Window 是应用程序的主界面，其构造函数创建一个 QSortFilterProxyModel 对象、两个 QTreeView 视图对象，其中一个视图对象被用来显示 QSortFilterProxyModel 中的虚拟数据项，另外一个被用来显示源模型中的数据项。

图 13-24 例子 basicSortFilterModel 的类图

创建源模型的函数如代码段 13-25 所示，主要是创建一个 QStandardItemModel 对象，并向其中添加若干行数据。行①调用的函数 insertRow() 具有原型：

```
bool QStandardItemModel::insertRow ( int row,
                                     const QModelIndex & parent = QModelIndex() )
```

表示在 `parent` 所指的父节点的第 `row` 行子节点之前插入一个新行。由于本例的源模型是一个简单的表格，所有数据项的父节点都是这个不可见根，因而行①第二个实参取默认值 `QModelIndex()`，也即无效索引，表示在不可见根的第 `row` 行之前插入一个新行。类似地，行②调用的 `index()` 函数本来具有三个参数，但是第三个参数取默认值 `QModelIndex()`，表示返回不可见根下某个数据项的索引。

代码段 13-25，创建源模型，取自 `z:\examples\mvc\basicsortfiltermodel\main.cpp`

```
void addMail(QAbstractItemModel *model, const QString &subject,
            const QString &sender, const QDateTime &date)
{
    model->insertRow(0);                                ①
    model->setData(model->index(0, 0), subject);          ②
    model->setData(model->index(0, 1), sender);
    model->setData(model->index(0, 2), date);
}
QAbstractItemModel *createMailModel(QObject *parent)
{
    QStandardItemModel *model = new QStandardItemModel(0, 3, parent);
    model->setHeaderData(0, Qt::Horizontal, QObject::tr("Subject"));
    model->setHeaderData(1, Qt::Horizontal, QObject::tr("Sender"));
    model->setHeaderData(2, Qt::Horizontal, QObject::tr("Date"));
    addMail(model, "Happy New Year!", "Grace K. <grace@software-inc.com>",
            QDateTime(QDate(2006, 12, 31), QTime(17, 03)));
    addMail(model, "Radically new concept", "Grace K. <grace@software-inc.com>",
            QDateTime(QDate(2006, 12, 22), QTime(9, 44)));
    .....
    return model;
}
```

类 `Window` 的构造函数如代码段 13-26 所示，我们只列出了和代理模型相关的部分，省略了创建图 13-23 界面中各个控件的语句。这个构造函数创建一个代理模型以及两个视图对象，令其中一个视图对象指向刚刚创建的代理模型。默认情况下，视图对象不响应用户的排序请求。为了使能这个功能，行①调用了视图对象的 `setSortingEnabled()` 函数。行②后的 `connect` 语句将各个控件的信号与类 `Window` 的槽函数相连。一旦用户输入或者更改了过滤、排序条件，类 `Window` 的相应槽函数将被调用，代理模型将对源模型中的数据项重新进行过滤、排序。

代码段 13-26，类 `Window` 的构造函数，取自
`z:\examples\mvc\basicsortfiltermodel>window.cpp`

```
Window::Window()
{
    proxyModel = new QSortFilterProxyModel;             .....
    sourceView = new QTreeView;                         .....
    proxyView = new QTreeView;                         .....
    proxyView->setModel(proxyModel);
    proxyView->setSortingEnabled(true);                 ①

    connect(filterPatternLineEdit, SIGNAL(textChanged(QString)), ②
            this, SLOT(filterRegExpChanged()));
    connect(filterSyntaxComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(filterRegExpChanged()));
    connect(filterColumnComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(filterColumnChanged()));
}
```

```

connect(filterCaseSensitivityCheckBox, SIGNAL(toggled(bool)),
        this, SLOT(filterRegExpChanged()));
connect(sortCaseSensitivityCheckBox, SIGNAL(toggled(bool)),
        this, SLOT(sortChanged()));
proxyView->sortByColumn(1, Qt::AscendingOrder); .....
}

```

该例子的 `main()` 函数在构造完毕 `Window` 对象后，调用该类的 `setSourceModel()` 函数，令新创建的代理模型指向代码段 13-24 中创建的源模型，如代码段 13-27 所示。

代码段 13-27，令代理模型指向源模型，取自
z:\examples\mvc\basicsortfiltermodel\window.cpp

```

void Window::setSourceModel(QAbstractItemModel *model)
{
    proxyModel->setSourceModel(model);
    sourceView->setModel(model);
}

```

当用户更改了过滤或者排序条件时，代码段 13-28 所定义类 `Window` 的槽函数会被调用。具体来说，当过滤条件变化时，槽函数 `filterRegExpChanged()` 会被调用。行①的控件 `filterSyntaxComboBox` 是类 `QComboBox` 的对象，表示过滤条件的格式。`QComboBox` 负责管理一个组合框。它在屏幕上显示一个输入框以及一个按钮（里面通常显示一个下箭头）。当用户单击按钮时，该控件弹出一个含有若干数据项的列表。用户选择其中一个之后，列表消失，被选择的数据项显示在输入框中。`QComboBox` 使用 `Model/View` 框架存放、显示数据。默认情况下，它采用 `QStandardItemModel` 存放列表中的数据项，采用 `QListView` 显示其中的数据项。虽然我们可以使用模型的接口函数来访问 `QStandardItemModel` 中的数据项，但是 `QComboBox` 提供了更加方便的访问方式。我们使用以下方式，向控件 `filterSyntaxComboBox` 添加一个数据项，

```

filterSyntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);

```

其中的字符串“Regular expression”被存放在角色 `DisplayRole` 对应的数据子项中，而枚举常量 `QRegExp::RegExp`（表示正则表达式格式）被存放在角色 `UserRole` 对应的数据子项中。行①调用 `QComboBox` 的成员函数 `currentIndex()` 返回组合框中第几个数据项被选择，再调用其成员函数 `itemData()` 返回该数据项中角色 `UserRole` 对应的数据子项，也就是表示过滤条件格式的枚举常量。

行②构造一个正则表达式对象，代理模型在行③使用该对象对源模型中的数据项进行过滤。在槽函数 `filterColumnChanged()` 中，行④设置过滤条件施加到源模型的哪个列。

代码段 13-28，代理模型对源模型数据项的过滤、排序，取自
z:\examples\mvc\basicsortfiltermodel\window.cpp

```

void Window::filterRegExpChanged()
{
    QRegExp::PatternSyntax syntax =
        QRegExp::PatternSyntax(filterSyntaxComboBox->itemData( ①
            filterSyntaxComboBox->currentIndex()).toInt());
    Qt::CaseSensitivity caseSensitivity =
        filterCaseSensitivityCheckBox->isChecked() ? Qt::CaseSensitive
                                                    : Qt::CaseInsensitive;
}

```



```

        QRegExp regExp(filterPatternLineEdit->text(), caseSensitivity, syntax);②
        proxyModel->setFilterRegExp(regExp); ③
    }
    void Window::filterColumnChanged()
    {
        proxyModel->setFilterKeyColumn(filterColumnComboBox->currentIndex());④
    }
    void Window::sortChanged()
    {
        proxyModel->setSortCaseSensitivity(
            sortCaseSensitivityCheckBox->isChecked() ? Qt::CaseSensitive
                                                    : Qt::Case-
                                                    Insensitive);
    }

```

13.7 便利视图类

本章前面几个节在使用 Model/View 框架时，模型对象和视图对象是相互独立的。我们可以直接定义一个模型对象（比如类 `QStandardItemModel` 或者其他便利模型类的对象），或者派生 `QAbstractListModel`、`QAbstractTableModel` 的子类，再定义子类的对象。然后，我们调用视图对象的成员函数 `setModel()`，将一个视图对象和一个模型对象关联起来。

为了方便对 Model/View 框架的使用，Qt 还提供了一组便利视图类。这些类在其内部定义了模型对象，并定义了一组简洁、易用的成员函数来操作模型对象中的数据项。同时，作为 `QAbstractItemView` 的子类，这些类能够显示内部的模型。由于这些类仅被用来显示内部的模型对象，它们的成员函数 `setModel()` 被声明为私有的，意味着我们无法使用这些类来显示外部的模型对象。另外，由于这些类内部的模型对象是私有的，其他视图对象也无法使用 `QAbstractItemModel` 定义的最小接口来访问它们，意味着我们无法令多个视图对象显示同一个模型。因此，便利视图类虽然易用，但不够灵活。

Qt 提供了三个便利视图类。类 `QListWidget` 负责保存、显示具有列表结构的模型，类 `QTableWidget` 负责表格结构的模型，类 `QTreeWidget` 负责树状结构的模型。接下来的几节分别阐述这几个类的使用方式。

13.7.1 QListWidget

类 `QListWidget` 在其内部定义了一个具有列表结构的模型，并以列表形式显示其中的数据项。列表中的每个数据项被表示为类 `QListWidgetItem` 的一个对象。在列表视图中所要显示的数据被存放在每个数据项的角色 `DisplayRole` 对应的数据子项中。这个数据子项的类型为 `QVariant`，其中可以存放类型为 `QString` 的字符串，也可以存放类型为 `int` 或者 `double` 的数值。我们可以调用 `QListWidget` 的成员函数 `addItem()` 向列表添加一个数据项。

如果列表中所要处理的数据是字符串类型，`QListWidget` 提供了更简单的添加方式，如代码段 13-29 行①所示。`QListWidget` 的成员函数 `addItems()` 将一个 `QStringList` 对象所含的全部元素添加到内部模型中，该函数将这些数据项中角色 `DisplayRole` 对应的数据子项设置为 `QString` 类型的，因而行②的排序操作会将“100”、“101”以及“102”排在“98”的前面。

代码段 13-29, 向 QListWidget 中添加数据项, 取自
z:\examples\mvc\QListWidget_demo\main.cpp

```
QListWidget * left = new QListWidget();
QStringList list;
list << "98" << "99" << "100" << "101" << "102";
left->addItem( list );           ①
left->sortItems();               ②

QListWidget * right = new QListWidget();
for ( int i=98; i<103; i++) {
    QListWidgetItem * item = new QListWidgetItem;           ③
    item->setData(Qt::DisplayRole, i);                       ④
    right->addItem(item);
}
right->sortItems();           ⑤
```

为了能够在数据项中存入数值类型的数据, 我们可以像行③那样先创建一个 `QListWidgetItem` 对象, 然后调用其 `setData()` 函数, 显式地设置该数据项中角色 `DisplayRole` 对应的数据子项的值。该函数的第二个参数是 `QVariant` 类型, 本例中的实参“i”会被转换为一个 `QVariant` 对象, 该对象会纪录所存放的数据为整数类型。行⑤对列表中的数据项进行排序时, 会将它们看作整数。

与普通模型中的数据项一样, `QListWidgetItem` 所表示的数据项也包含多个『角色, 数据子项』对。前面的例子仅演示和如何修改 `DisplayRole` 对应的数据子项, 下面我们再举一个较复杂的例子, 来演示如何更改其他数据子项, 以使得 `QListWidget` 显示出来的列表具有更漂亮的外观、更强大的功能。

这个例子运行时的界面如 13.2.1 节图 13-8 所示, 它在对话框的左侧显示 2011 年世界上 10 大新闻的标题以及图标。用户单击其中一个后, 程序在右侧的上方显示该新闻的图片, 下方显示英文版的新闻文字。如果用户将鼠标停留在列表中某个数据项的区域内, 程序会显示中文版的新闻文字。

新闻的信息被存放在一个全局数组中, 如代码段 13-30 所示。数组中每个元素的类型为结构体 `News_Info`, 其成员变量都是指向字符串的指针, 分别指向新闻标题、新闻图片的文件名、中文版新闻文字、英文版新闻文字。

代码段 13-30, 新闻的表示, z:\examples\mvc\item_roles\NewsDialog.cpp

```
typedef struct {
    char * news_title, * image_filename;
    char * news_in_Chinese, * news_in_English;
} News_Info;
News_Info news_2011[10] = {
    {"日本海啸", "images/Japan_Tsunami.jpg",
     "3月14日, 日本岩手县大槌町, 一艘船被冲到屋顶, 四周被无数碎片包围.....",
     "A tsunami-tossed boat rests on top of a building amid a sea of debris in Otsuchi....."},
    .....
}
```

程序定义类 `NewsDialog` 负责显示该例子的主界面, 它是 `QDialog` 的子类。该类定义了以下私有数据成员: `QListWidget * listWidget`, 所指对象表示界面左侧的列表; `QLabel *`

image，所指对象负责显示界面右上方的新闻图片；QLabel * news，所指对象负责显示界面右下方的英文版新闻文字。得益于 QLabel 的设计，该类既可以显示一个图像，也可以显示一段文字。

在 NewsDialog 的构造函数中我们创建以上对象，如代码段 13-31 所示。行①创建一个列表视图，接下来的几行设置该视图对象的显示属性。行②创建一个 QListWidgetItem 对象，表示列表中的一个数据项，接下来的几行设置该数据项中各角色对应的数据子项。由于本例的一些字符串含有中文字符，而这些字符在笔者所用的 VS 2010 开发环境中是以 GB 2312 标准进行编码的，所以本例子需要在 main() 函数中调用：

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("GB2312")) ;
```

设置对应的编码方案，然后在行②～③的几行中调用 QObject::tr() 函数，将本例中的字符串转换为 Unicode 编码的字符串。在这几行中，角色 DisplayRole 对应着数据项的文字，DecorationRole 对应着数据项的图标，TextAlignmentRole 对应着数据项文字的对齐方式，ToolTipRole 对应着鼠标停留在数据项区域中时所显示的文字信息。而角色 UserRole 对应的数据子项却是由应用程序自行决定的，本例利用这个数据子项来存放英文版的新闻文字。

代码段 13-31，设置 QListWidgetItem 所表示数据项中的数据子项，取自
z:\examples\mvc\item_roles\NewsDialog.cpp

```
NewsDialog::NewsDialog()
{
    listWidget = new QListWidget(this);           ①
    listWidget->setViewMode(QListView::IconMode);
    listWidget->setIconSize(QSize(120, 80));
    listWidget->setMovement(QListView::Static);
    listWidget->setFixedWidth(340);
    listWidget->setSpacing(12);

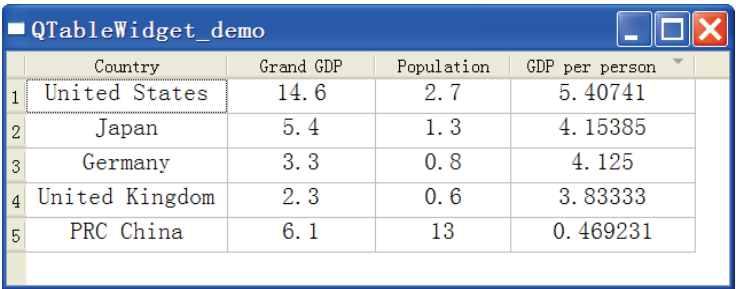
    for ( int i=0; i<10; i++) {
        News_Info news = news_2011 [i];
        QListWidgetItem * item = new QListWidgetItem(listWidget); ②
        item->setData(Qt::DisplayRole,    QObject::tr(news.news title) );
        item->setData(Qt::DecorationRole, QIcon(news.image filename) );
        item->setData(Qt::TextAlignmentRole, Qt::AlignHCenter );
        item->setData(Qt::ToolTipRole, QString( QObject::tr(news.
            news_in_Chinese) ) );
        item->setData(Qt::UserRole,  QString( news.news_in_English) ); ③
    }
    connect(listWidget,           ④
        SIGNAL(currentItemChanged(QListWidgetItem *, QListWidgetItem *)),
        this, SLOT(changeNews(QListWidgetItem *, QListWidgetItem *)));
    image = new QLabel(this);
    news = new QLabel(this);
    .....
}
void NewsDialog::changeNews(QListWidgetItem *current, QListWidgetItem
*previous)      ⑤
{
    int row = listWidget->row( current );
    image->setPixmap( QPixmap( news_2011[row].image_filename ) );
    news->setText( current->data(Qt::UserRole).toString() );      ⑥
}
```

行④将使得用户单击列表中某个数据项时,调用类 NewsDialog 的槽函数 changeNews(行⑤),该函数的参数 current 表示用户单击的数据项,而 previous 表示单击之前的当前数据项。行⑥读取数据项中角色 UserRole 对应的数据子项,也就是此前写入的英文版新闻文字,显示在界面右下角的 QLabel 对象中。

类 QTableWidgetItem 在其内部定义了一个具有表格结构的模型,并能够以表格的样子显示这个模型。表格中的每个数据项被表示为类 QTableWidgetItem 的对象。在表格视图中所要显示的数据被存放在每个数据项的角色 DisplayRole 对应的数据子项中。这个数据子项的类型为 QVariant,其中可以存放类型为 QString 的字符串,也可以存放类型为 int 或者 double 的数值。我们可以调用该类的成员函数 setData(),将所要显示的数据写入角色 DisplayRole 对应的数据子项中。类 QTableWidgetItem 的成员函数 setItem()可以将一个数据项设定到指定的行、列。

为了更加方便地控制数据项的外观, QTableWidgetItem 提供了一组函数,用于设置数据项的背景、字体、文字对齐方式等与外观相关的属性。这些函数实际上调用了该类的成员函数 setData(),通过修改数据项中各个数据子项的值,来影响数据项的外观。

我们举一个例子来演示 QTableWidgetItem 的功能。这个例子的输出如图 13-25 所示,表格中的每一行显示一个国家的名称、这个国家 2010 年 GDP 总量值(单位为万亿美元),这个国家 2010 年的总人口(单位为亿),这个国家 2010 年人均 GDP(单位为万美元/每人)。



	Country	Grand GDP	Population	GDP per person
1	United States	14.6	2.7	5.40741
2	Japan	5.4	1.3	4.15385
3	Germany	3.3	0.8	4.125
4	United Kingdom	2.3	0.6	3.83333
5	PRC China	6.1	13	0.469231

图 13-25 使用 QTableWidgetItem 显示一个表格

每个国家的 GDP 信息存放在结构体 GDP_FACTS 中,成员 Country 指向一个国家的名称,GDP 表示 GDP 总值,Population 表示总人口。我们无须在这个结构体中定义每个国家的人均 GDP,因为这个数据可以从已有的成员计算出来。GDP 数据的表示如代码段 13-32 所示。

代码段 13-32, GDP 数据的表示,取自 z:\examples\mvc\QTableWidgetItem_demo\main.cpp

```
typedef struct {
    char * country;
    double GDP, population;
}GDP_FACTS;
GDP_FACTS GDP facts[] = {
    {"United States", 14.6, 2.7 },
    {"PRC China", 6.1, 13.0 },
    {"Japan", 5.4, 1.3 },
    {"Germany", 3.3, 0.8 },
    {"United Kingdom", 2.3, 0.6 }
};
```

使用 QTableWidgetItem 显示部分国家 GDP 数据的过程如代码段 13-33 所示。行①创建一个

QTableWidget 对象，行②调用其成员函数 setHorizontalHeaderLabels()设置每列的标头。行③创建一个 QTableWidgetItem 对象表示表格中的一个数据项。该行下面的几行代码设置该数据项的文字对齐方式以及字体。行④~⑤设置某一行 4 个列的数据项，将角色 DisplayRole 对应的数据子项设置为该列对应的值。其中成员函数 setData()的第二个函数参数是 QVariant 类型，行④构造的 QVariant 对象存放一个 QString 值，其他几行构造的 QVariant 对象存放 double 类型的值。当用户单击视图对象某列的标头要求排序时，QTableWidget 会依据该列数据项中角色 DisplayRole 对应数据子项的类型进行排序。创建完毕这些数据项后，行⑥调用 QTableWidget 的成员函数 setItem()将它们添加到 QTableWidget 的内部模型中。

代码段 13-33，类 QTableWidget 的使用，取自
z:\examples\mvc\QTableWidget_demo\main.cpp

```
int main(int argc, char *argv[])
{
    .....
    const int rows=5, columns=4;
    QTableWidget widget(rows, columns);           ①
    QStringList list;
    list << "Country" << "Grand GDP" << "Population" << "GDP per person";
    widget.setHorizontalHeaderLabels(list);       ②

    for (int row=0; row<rows; row++) {
        QTableWidgetItem * items[columns];
        for (int j=0; j<columns; j++) {
            items[j] = new QTableWidgetItem();   ③
            items[j]->setTextAlignment( Qt::AlignHCenter);
            QFont font;  font.setPointSize(16);
            items[j]->setFont( font );
        }
        GDP_FACTS * p = & GDP_facts[row];
        items[0]->setData(Qt::DisplayRole, p->country );      ④
        items[1]->setData(Qt::DisplayRole, p->GDP );
        items[2]->setData(Qt::DisplayRole, p->population );
        items[3]->setData(Qt::DisplayRole, p->GDP / p->population ); ⑤
        for (int j=0; j<columns; j++)
            widget.setItem(row, j, items[j] );              ⑥
    }
    widget.setSortingEnabled(true );
    .....
}
```

13.7.2 QTreeWidget

类 QTreeWidget 在其内部定义了一个具有树状层次结构的模型，并以树状外观显示这个模型。虽然显示出来的外观和 QTreeView 的类似，但是程序员并不需要自己创建模型对象，因而使用起来更加方便。使用 QTreeWidget 时，模型中的数据项被表示为 QTreeWidgetItem。向模型添加一个数据项时，程序员只须创建一个 QTreeWidgetItem 对象并指定其父节点，QTreeWidget 即可记录并维护所有节点的父子关系，形成一个完整的树状模型。

为了使得该类更容易被使用，Qt 甚至定义了一个类 QTreeWidgetItemIterator，以对该类中的树进行前序遍历。所谓前序遍历，是指先访问某个父节点本身，然后再依次对其所含的子树进行前序遍历。关于树以及树的遍历，请参考文献[27]。

下面我们举一例子来说明 QTreeWidget 的用法。该例子曾在 13.2 节提及，其输出如图 13-26 所示，显示的是 C++ 创始人 Bjarne Stroustrup 所著书籍 *The Design and Evolution of C++* 的部分目录。

Section Number	Title	Page Number
Chapter 1	The Prehistory of C++	19
Section 1.1	Simula and Distributed Systems	19
Section 1.2	C and Systems Programming	22
Section 1.3	General Background	23
Chapter 2	C with Classes	27
Section 2.1	The Birth of C with Classes	27
Section 2.2	Feature overview	29
Section 2.3	Classes	30
Section 2.4	Run-Time Efficiency	32
Section 2.4.1	Inlining	33
Chapter 3	The Birth of C++	63
Chapter 4	C++ Language Design Rules	109

图 13-26 使用 QTreeWidget 显示具有树状结构的数据集

我们使用代码段 13-34 中的结构体 SectionInfo 表示目录中每个章节的信息。该结构体含有四个类型为字符指针的成员。这些指针所指的字符串依次表示：章节号（比如“Section 2.4.1”），章节题目（比如“Inlining”，在书中的页码（比如“33”），其父节点的章节号（比如“Section 2.4”）。所有章节的信息存放在数组 directory 中。

代码段 13-34，书籍目录的表示，取自 z:\examples\mvc\QTreeWidget_demo\main.cpp

```
typedef struct {
    char * sect_id, * title, * page num;
    char * parent_id;
} SectionInfo;
SectionInfo directory[] = {
    {"Chapter 1", "The Prehistory of C++", "19", ""},
    {"Chapter 2", "C with Classes", "27", ""},
    .....,
    {"Section 1.1", "Simula and Distributed Systems", "19",
     "Chapter 1"},
    {"Section 1.2", "C and Systems Programming", "22",
     "Chapter 1"},
    .....,
    {"Section 2.4", "Run-Time Efficiency", "32",
     "Chapter 2"},
    {"Section 2.4.1", "Inlining", "33",
     "Section 2.4"},
};
```

分析这些章节信息并构造一个树状模形的过程如代码段 13-35 所示。行①、②设置一个 QTreeWidget 视图对象的列数以及每列的标头。对于存放在结构体 SectionInfo 中每条章节信息，如果父节点的章节号为空字符串，这条信息表示的是书中某“章”的信息，应该作为模型的最顶层节点存放。行③创建一个 QTreeWidgetItem 对象时，令其父节点为 QTreeWidget 视图对象，就表达了这样的意图。

对于一个章所含的小节，由于在创建其对应的 QTreeWidgetItem 对象时需要指定其父节点，我们首先要在树状模形中寻找其父节点。一个小节的父节点可能表示一个“章”（比如“Section 1.1”的），也可能表示另外一个更高层的小节（比如“Section 2.4.1”的）。表示该小

节的结构体 `SectionInfo` 只保存了其父节点的章节号信息，因此我们需要遍历整棵树，依据其父节点的章节号信息确定其父节点的位置。行④定义了一个类型为 `QTreeWidgetItemIterator` 的迭代器，它能够对树状模型进行前序遍历。在遍历过程中，我们判断迭代器所指当前节点的章节号是否是我们正在寻找的。找到后，我们创建一个 `QTreeWidgetItem` 对象，令其父节点为刚刚找到的那个节点。

代码段 13-35，构建 `QTreeWidget` 中的树状模型，取自
z:\examples\mvc\QTreeWidget_demo\main.cpp

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTreeWidget * treeWidget = new QTreeWidget();
    treeWidget->setColumnCount(3); ①
    QStringList headers;
    headers << "Section Number" << "Title" << "Page Number";
    treeWidget->setHeaderLabels(headers); ②

    for (int i=0; i<sizeof(directory)/sizeof(directory[0]); i++) {
        SectionInfo info = directory[i];
        QTreeWidgetItem * item=NULL;
        if (strcmp(info.parent_id, "")==0 ){
            item = new QTreeWidgetItem(treeWidget); ③
        }else{
            QString parent_id(info.parent_id);
            QTreeWidgetItemIterator it (treeWidget); ④
            while ( (*it)->text(0) != parent_id)
                ++it;
            item = new QTreeWidgetItem( *it ); ⑤
        }
        if ( item) {
            item->setText(0, info.sect_id);
            item->setText(1, info.title );
            item->setText(2, info.page num);
        }
    }
    treeWidget->resize(400,200);
    treeWidget->show();
    return app.exec();
}
```

从这个例子可以看出，和 `QTreeView` 相比，`QTreeWidget` 的用法更加简单。我们用 `QTreeWidgetItem` 存放数据项的信息。创建该类的每个对象时，恰当地指定其父节点，然后将数据项的信息写入该对象即可。

Qt中的C++技术

- ◎运行在苹果公司iPhone、iPod、iTouch以及iPad上的操作系统
- ◎Windows Mobile和Symbian OS
- ◎关系数据库管理系统Oracle Database, MySQL, IBM DB2, Microsoft SQL Server, IBM Informix, SAP DB/MaxDB
- ◎微软的IE、Google Chrome、Mozilla Firefox、Safari以及Opera
- ◎流行的办公套件Microsoft Office、Sun Open Office、Corel Office
- ◎Google、eBay、Amazon、Facebook

均多多少少地在使用C++，它一直处于Tiobe网站（www.tiobe.com）排名的第三，应用如此广泛的一门语言，难道真的“难学难精”？

本书帮你通过Qt，这样一个跨平台的C++开发框架，学会学精C++！

- 为什么会从众多的开源C++项目中选择Qt
- Qt对字符串的处理思路
- 比较C++标准库、Qt对数据输入/输出的不同处理思路
- 介绍隐式共享技术，并剖析了QString的部分源代码以演示该技术的具体实现
- 介绍函数的概念及其在QTL中的应用，以及QTL是如何使用模板特化技术优化QList性能的
- 如何使用QThread创建一个线程，如何使用互斥体QMutex
- Qt为什么使用信号与槽机制而不是传统的回调函数进行对象间的通信
- Qt图形系统的基本知识，Graphics/View框架如何处理图形元素
- Qt如何实现模型-视图-控制器架构（Model-View-Control structure，MVC）
- Qt如何实现命令模式、抽象工厂模式以及观察者模式
- Qt提供的一些智能指针，能有效地解决内存泄露以及野指针等问题

本书面向以下读者：

- 软件学院或者计算机学院的学生，将本书作为学习“C++程序设计”或者“面向对象软件设计”课程的参考书；
- 上述课程的教师，将本书的内容融入他们的主讲或者试验环节；
- 软件行业的开发者，与本书一起领略Qt的精妙。

上架建议：C++

ISBN 978-7-121-17159-8



9 787121 171598 >

定价：55.00元（含光盘1张）



责任编辑：孙学瑛
封面设计：侯士卿