

全新打造的**Qt**爱好者社区强力支持！
短期即可入门及提高的优秀**Qt**系列书籍！

Qt应用编程系列丛书

内部培训使用，不要外传！！！

霍亚飞 编著
吴迪 白建平 董世明 审校

Qt Creator 快速入门

- 全新， 基于最新的Qt及Qt Creator编写， 包含Qt Quick！
- 经典， 基于经典的Qt网络博客编写， 可无限更新！
- 基础， 对每个知识点详尽讲解，并均设计了示例程序！
- 系统， 与《Qt及Qt Quick开发实战精解》配套， 理论结合实际！



北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

Qt 应用编程系列丛书

Qt Creator 快速入门

霍亚飞 编著

吴迪 白建平 董世明 审校

北京航空航天大学出版社

内 容 简 介

本书是基于 Qt Creator 集成开发环境的入门书籍, 详细介绍了 Qt Creator 开发环境的使用和 Qt 基本知识点的应用。本书内容主要包括 Qt 的基本应用, 以及 Qt 在图形动画、影音媒体、数据处理和网络通信方面的应用内容。

本书的内容全面、实用, 讲解通俗易懂, 适合没有 Qt 编程基础、有 Qt 编程基础但是没有形成知识框架以及想学习 Qt 某一方面应用的读者。对于想进一步学习 Qt 开发实例或者 Qt Quick 的读者, 可以学习《Qt 及 Qt Quick 开发实战精解》一书。

图书在版编目(CIP)数据

Qt Creator 快速入门 / 霍亚飞编著. — 北京 : 北京航空航天大学出版社, 2012. 5

ISBN 978 - 7 - 5124 - 0783 - 1

I. ①Q… II. ①霍… III. ①软件工具—程序设计
IV. ①TP311. 56

中国版本图书馆 CIP 数据核字(2012)第 063232 号

版权所有, 侵权必究。

Qt Creator 快速入门

霍亚飞 编著

吴 迪 白建平 董世明 审校

责任编辑 董立娟

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191) <http://www.buaapress.com.cn>

发行部电话:(010)82317024 传真:(010)82328026

读者信箱: emsbook@gmail.com 邮购电话:(010)82316936

涿州市新华印刷有限公司印装 各地书店经销

*

开本: 710×1 000 1/16 印张: 30.25 字数: 681 千字

2012 年 5 月第 1 版 2012 年 5 月第 1 次印刷 印数: 4 000 册

ISBN 978 - 7 - 5124 - 0783 - 1 定价: 59.00 元

丛书前言

本系列丛书的原型是网络上的一系列博客教程。2009年,当作者开始使用推出不久的Qt开发环境Qt Creator时,由于该软件还不成熟,所以经常出现一些问题,但是网络上却几乎找不到任何的相关内容可以参考。于是作者便将自己的学习经验和心得进行整理,以yafeilinux为网名在博客中推出了一系列Qt和Qt Creator相关教程。这些教程浅显易懂,使用的描述语言很自然,而且图文并茂,每一个知识点都设计了一个可以运行的示例程序。这些特点都很符合读者的需求,所以这个系列教程得到了众多网友的肯定和称赞,这也给了作者很大的信心继续写下去。2010年,作者接到北航出版社的邀请,于是有了这个系列丛书的编写。

在这一年里,作者创建了www.yafeilinux.com网站,希望可以有更多的人一起参与到Qt的普及工作中。在该网站创建后,作者不仅将系列教程扩展到了Qt基础、2D绘图、数据库操作和网络通信等几个部分,而且还推出了多个专题教程和开源软件。其中,《Qt串口通信专题教程》在网络上广为流传,几乎成为了学习Qt串口编程的必备教程;而开源软件中的“音乐播放器”,在推出不到一个月的时间里就有了上万次的下载量。现在网站的访问量已经超过百万,同时在作者的邮箱中,一年里收到了两千多封请教和交流的邮件,而QQ交流群更是建一个满一个。这些都让作者看到了Qt爱好者热情和Qt发展的良好趋势,也让作者下定决心要写出一本好书。2011年3月,Qt 4.7.2和Qt Creator 2.1.0正式发布,该版本中Qt Creator第一次正式使用了中文界面,也是第一次正式支持Qt Quick编程。作者感觉是时候开始正式写作了,有了这一年来和众多网友的交流,以及作者在Qt各个应用方面的编程积累,已经有了足够的信心来写出一部经典作品。

直至今日书已成型,10个月的写作包含了太多的心酸与喜悦,最终呈现给读者的是包括了《Qt Creator快速入门》和《Qt及Qt Quick开发实战精解》的一个系列丛书。之所以同时推出两本,是为了给不同学习阶段的读者提供灵活的选择。尽管本套书籍已经包含了Qt中最常用的大部分内容,不过Qt实在太庞大了,所以还是有很多内容没有写进去,这些会在www.yafeilinux.com网站中得到补充。一本书的厚度是有限的,但是网络中的内容却可以是无限的,网上的系列教程,作者还会一直写下去的。

Qt简介

Qt是一个跨平台应用程序和UI开发框架。使用Qt只须一次性开发应用程序,无须重新编写源代码,便可跨不同桌面和嵌入式操作系统部署这些应用程序。Qt

Software 的前身为创始于 1994 年的 Trolltech(奇趣科技),Trolltech 于 2008 年 6 月被 Nokia 收购,加速了其跨平台开发战略。Qt Creator 是全新的跨平台 Qt IDE,可单独使用,也可与 Qt 库和开发工具组成一套完整的 SDK。其中包括:高级 C++ 代码编辑器、项目和生成管理工具、集成的上下文相关的帮助系统、图形化调试器、代码管理和浏览工具。Qt Quick 是在 Qt 4.7 中被引进的一种高级用户界面技术,开发人员和设计人员可用它协同创建动画触摸式用户界面和应用程序。

丛书特色

本系列丛书作为全面介绍 Qt、Qt Creator 和 Qt Quick 的入门级教材,也是市面上第一套详细介绍 Qt Creator 和 Qt Quick 的教材。与其他相关书籍最大的不同之处还在于,本套书籍是基于网络博客教程的。综合来说,本套书籍主要具有以下特色:

- 最新。基于最新的 Qt 4.7.2 和 Qt Creator 2.1.0 版本进行编写,在该版本中首次正式集成了 Qt Quick。
- 最全。书中的内容包含了 Qt 基础、图形动画、多媒体、数据库、网络通信、WebKit 以及 Qt Quick 等所有基本的应用内容。
- 无限更新。本套书籍对应的网络教程是无限更新的,而且读者可以通过邮件、QQ 群或者论坛和作者零距离接触。
- 按应用分类。本套书籍中将同一应用的相关章节放在了一起作为一个篇章,例如图形动画篇中就是讲解了所有与绘图和动画相关的知识,这样组织更有利于读者在程序开发时进行选择性参考学习。
- 知识点讲解与大型实例设计分离。本套书籍将知识的讲解和综合实例的设计分为两部分。这样读者便不会因为要在很长的代码中学习某个知识点而一头雾水,但每一个知识点都设计了一个很简单的示例程序,让读者可以实际看到该知识点的应用效果;而设计大型实例时,将主要的精力放在程序框架和功能实现上,不再细讲其中的知识点,这样更有利于读者学习设计综合性程序。
- 程序设计分步实现。本套书籍不像其他书籍那样将一个完整示例程序的所有代码全部列出并进行讲解,而是分步来设计该程序,从创建项目开始,一个功能模块一个功能模块地分步添加,这样可以让读者更好地了解程序的设计过程。
- 关键字提示。本套书籍的编写主要基于 Qt 参考文档,而且所讲解的知识点也只是 Qt 参考文档中的部分内容,读者在学习时一定要多参考 Qt 帮助文档。在本套书籍讲解的所有的知识点和示例程序中,都很明显地标出了其在 Qt 帮助中对应的关键字,这样可以让读者对书中的内容有迹可循。

丛书结构

本系列丛书一共分为两本:《Qt Creator 快速入门》和《Qt 及 Qt Quick 开发实战精解》。《Qt Creator 快速入门》中主要进行知识点的详细讲解,根据应用的不同分为了 5

个篇：基本应用、图形动画、影音媒体、数据处理和网络通信；《Qt 及 Qt Quick 开发实战精解》中主要讲解了 5 个综合实例以及 Qt Quick 的内容，其中这 5 个实例程序分别对应了《Qt Creator 快速入门》的 5 个篇，也就是说每一个应用实例都是对应篇知识点的综合应用。本系列丛书的结构如下所示：



使用本系列丛书

对于没有任何 Qt 编程经验的读者，建议先学习《Qt Creator 快速入门》。该书对 Qt 基本内容以及 Qt Creator 开发环境进行了详细讲解，而且每一个知识点都设计了一个可以独立运行的示例程序，非常适合初学者入门使用。对于有一定 Qt 编程经验，并且想学习 Qt 综合实例编写的读者，推荐使用《Qt 及 Qt Quick 开发实战精解》。该书对于不同的应用领域分别设计了一个实用的应用程序，而且每一个实例都应用了众多的知识点，非常适合没有实际开发经验的读者；如果读者想学习全新的 Qt Quick 技术，那么《Qt 及 Qt Quick 开发实战精解》也是很好的选择，该书的 Qt Quick 部分详细介绍了 Qt Quick 的方方面面。

当然，本系列丛书是一个相互联系的整体，建议读者最好可以在学习《Qt Creator 快速入门》的同时学习《Qt 及 Qt Quick 开发实战精解》，每学完一篇内容就去学习对应的实例程序，这样可以达到更好的效果。

在学习过程中要多动手，尽量自己按照步骤编写代码，只有在遇到无法解决的问题时再去参考本系列丛书提供的源代码。每当学习一个知识点时，书中都会给出 Qt 帮助中的关键字，建议参考 Qt 的帮助文档，看英文原文是怎么描述的。不要害怕去看那些英文文档，因为在网上的不可能找到所有文档的中文翻译，而且有些翻译也偏离了原意。学会看参考文档是入门 Qt 编程的重要一步，不要说自己英文不好，其实跟自己的英文水平关系不大，只要坚持，掌握了一些英文术语和关键词以后，阅读英文文档是不成问题的。

Qt 版本说明

本系列丛书是基于 Windows 下的 Qt 4.7.2 和 Qt Creator 2.1.0 版本的，这两个

版本是在本系列丛书开始编写时的最新版本。为了避免读者使用不同的操作系统而产生不必要的问题,本系列丛书采用了最常用的 Windows XP 系统。并且只讲解了 Qt 桌面编程,并没有涉及移动平台开发和嵌入式开发的内容,如果要进行移动平台(Symbian 和 MeeGo Harmattan 系统)软件开发,那么可以下载 Qt SDK,它集成了 Qt 桌面、移动平台的开发库以及 Qt Creator 开发环境。

这里要向对 Qt 版本不是很了解的读者说明一下,对于 Qt 程序开发,无论是在 Windows 系统下进行开发,还是在 Linux 系统下进行开发;无论是进行桌面程序开发,还是进行移动平台或者嵌入式平台的开发,只需要编写一次代码,然后分别进行编译就可以了,这就是 Qt 最大的特点,即所谓的“一次编写,随处编译”。也就是说,读者只需要学习书中 Qt 的基本内容,然后编写代码,使用 Qt 不同的版本进行编译即可。

学习本系列丛书时,推荐使用指定的 Qt 和 Qt Creator 版本,因为对于初学者来说,任何微小的差异都可能导致错误的理解。当然,也可以使用其他版本,在作者的网站上有不同平台的各个版本的使用教程。

致 谢

感谢北京航空航天大学出版社的编辑,是他们的邀请和支持,才让我坚定信心要写一本与众不同的经典作品。

感谢那些关注和爱好 Qt 的朋友们,是他们的支持和帮助,才让作者一步一步走下来。其中一些朋友还参与了本系列丛书的审校和代码审核工作,具体的审校分工是:解放军装甲兵工程学院的吴迪(wd007)完成了初审、Qt 中文论坛管理员白建平(XChinuX)完成了二审、上海大学计算机工程与科学学院的董世明完成了终审;参加代码审核的朋友有:程梁(豆子)、刘柏燊(紫侠)、么贺文(mehewen)、胡峰(古月行云)、周慧宗、杨凡(云帆)和黄金金等,他们分别完成了本系列丛书代码在 Windwos XP 系统、Windows 7 系统和 Ubuntu Linux 系统下的代码审核工作。是众多朋友的认真工作,才使得本系列丛书可以较早出版。

由于作者技术水平有限,经验也很欠缺,再加上 Qt 的内容繁多,并且没有统一的中文术语参考,所以书中难免有各种错误理解和代码设计缺陷,恳请读者批评指正。读者可以到作者的网站: www.yafeilinux.com 下载本系列丛书的源码,还可以到 Qt 爱好者社区(www.qter.org)在本系列丛书的专版进行讨论交流,查看与本系列丛书对应的不断更新的系列教程。

霍亚飞

2012 年 2 月于北京

前言

本书主要讲解 Qt Creator 开发环境的使用和 Qt 基本知识点的应用,适合没有 Qt 编程基础、有 Qt 编程基础但是没有形成知识框架以及想学习 Qt 中某一方面应用的读者阅读。因为书中的内容比较浅显,而且讲解很详细,所以读者可以根据自己的情况选择性学习。

书中对每个小的知识点都进行了详细讲解,并且均设计了一个简单的示例程序来帮助读者理解学习。不仅如此,在介绍知识点的同时还标明了该知识点在 Qt 帮助文档中的关键字,使得读者可以很容易地找到相关知识点的出处。本书内容共包含了 20 章,并根据应用内容的不同又分为了 5 个篇:

- 基本应用篇:包括第 1~9 章。该篇讲解 Qt 最基本的内容,包含了对 Qt Creator 开发环境的详细介绍和 Qt 编程中最基本的术语、概念和部件的使用方法等内容。
- 图形动画篇:包括第 10~12 章。该篇讲解 Qt 绘图与动画的内容,包含了 2D 绘图、3D 绘图、图形视图、状态机和动画等内容。
- 影音媒体篇:包括第 13~14 章。该篇讲解 Qt 多媒体应用的内容,包含了音频/视频播放、底层控制和 Phonon 多媒体框架等内容。
- 数据处理篇:包括第 15~17 章。该篇讲解 Qt 数据存储与显示的内容,包含了文件目录操作、模型/视图、数据库和 XML 等内容。
- 网络通信篇:包括第 18~20 章。该篇讲解 Qt 网络与通信的内容,包含了网络各协议编程、进程、线程和 WebKit 等内容。

《Qt 及 Qt Quick 开发实战精解》一书中对应本书的每一个篇都设计了一个综合的应用程序,它们是众多知识点的综合应用。读者学习完本书的一篇内容后,建议在《Qt 及 Qt Quick 开发实战精解》中学习对应的实例程序,可以达到更好的效果。

开始学习本书前,读者最好有一定的 C++ 语言基础;如果没有任何的编程基础,那么可以在学习本书的同时学习 C++ 语言,这样理论结合应用,可以达到更好的效果。

霍亚飞

2012 年 2 月于北京



录

基本应用篇

第1章 Qt Creator 简介	2
1.1 Qt Creator 的下载与安装	2
1.1.1 下载软件	2
1.1.2 安装软件	3
1.2 Qt Creator 环境介绍	3
1.2.1 运行一个示例程序	6
1.2.2 帮助模式	8
1.3 Qt 工具简介	9
1.3.1 Qt Assistant(Qt 助手)	9
1.3.2 Qt Designer(Qt 设计师)	10
1.3.3 Qt Examples and Demos(Qt 演示程序与示例)	10
1.3.4 Qt Linguist(Qt 语言家)	10
1.4 小结	11
第2章 Hello World	12
2.1 编写 Hello World 程序	12
2.1.1 新建 Qt Gui 应用	12
2.1.2 文件说明与界面设计	14
2.2 程序的运行与发布	16
2.2.1 程序的运行	16
2.2.2 程序的发布	19
2.2.3 设置应用程序图标	20
2.3 helloworld 程序源码与编译过程详解	22
2.3.1 纯代码编写程序与命令行编译	22
2.3.2 使用.ui 文件	27
2.3.3 自定义 C++ 类	31
2.3.4 使用 Qt 设计师界面类	34
2.4 项目模式和项目文件介绍	34
2.4.1 项目模式	34



2.4.2 项目文件	35
2.4.3 关于本书源码的使用	36
2.5 小结	37
第3章 窗口部件	38
3.1 基础窗口部件 QWidget	39
3.1.1 窗口、子部件以及窗口类型	39
3.1.2 窗口几何布局	42
3.1.3 程序调试	42
3.2 对话框 QDialog	45
3.2.1 模态和非模态对话框	46
3.2.2 多窗口切换	47
3.2.3 标准对话框	51
3.3 其他窗口部件	59
3.3.1 QFrame 类族	59
3.3.2 按钮部件	63
3.3.3 行编辑器	65
3.3.4 数值设定框	67
3.3.5 滑块部件	68
3.4 小结	69
第4章 布局管理	70
4.1 布局管理系统	70
4.1.1 布局管理器	71
4.1.2 设置部件大小	74
4.1.3 可扩展窗口	77
4.1.4 分裂器	78
4.2 设置伙伴	78
4.3 设置 Tab 键顺序	79
4.4 小结	80
第5章 应用程序主窗口	81
5.1 主窗口框架	81
5.1.1 菜单栏和工具栏	82
5.1.2 中心部件	86
5.1.3 Dock 部件	87
5.1.4 状态栏	88
5.1.5 自定义菜单	89
5.2 富文本处理	92
5.2.1 富文本文档结构	92

5.2.2 文本块	93
5.2.3 表格、列表与图片	97
5.2.4 查找功能	99
5.2.5 语法高亮与 HTML	100
5.3 拖放操作	102
5.3.1 使用拖放打开文件	102
5.3.2 自定义拖放操作	103
5.4 打印文档	107
5.5 小结	109
第6章 事件系统	110
6.1 Qt 中的事件	110
6.1.1 事件的处理	111
6.1.2 事件的传递	111
6.2 鼠标事件和滚轮事件	115
6.3 键盘事件	117
6.4 定时器事件与随机数	120
6.5 事件过滤器与事件的发送	123
6.6 小结	125
第7章 Qt 对象模型与容器类	126
7.1 对象模型	126
7.1.1 信号和槽	127
7.1.2 属性系统	132
7.1.3 对象树与拥有权	135
7.1.4 元对象系统	137
7.2 容器类	138
7.2.1 Qt 的容器类简介	138
7.2.2 遍历容器	143
7.2.3 通用算法	150
7.2.4 QString	152
7.2.5 QByteArray 和 QVariant	157
7.3 正则表达式	159
7.3.1 正则表达式简介	160
7.3.2 正则表达式组成元素	162
7.3.3 文本捕获	165
7.4 小结	166
第8章 界面外观	167
8.1 Qt 风格	167

8.1.1 使用不同风格预览程序	168
8.1.2 使用不同风格运行程序	168
8.1.3 调色板	169
8.2 Qt 样式表	170
8.2.1 概述	170
8.2.2 Qt 样式表语法	172
8.2.3 自定义部件外观与换肤	174
8.3 特殊效果窗体	178
8.3.1 不规则窗体	178
8.3.2 透明窗体	179
8.4 小结	181
第 9 章 国际化、帮助系统和 Qt 插件	182
9.1 国际化	182
9.1.1 使用 Qt Linguist 翻译应用程序	183
9.1.2 程序翻译中的相关问题	188
9.2 帮助系统	191
9.2.1 简单的帮助提示	191
9.2.2 定制 Qt Assistant	191
9.3 创建 Qt 插件	199
9.3.1 在设计模式提升窗口部件	199
9.3.2 创建应用程序插件	200
9.3.3 创建 Qt Designer 自定义部件	205
9.4 小结	206

图形动画篇

第 10 章 2D 绘图	208
10.1 基本绘制和填充	208
10.1.1 基本图形的绘制和填充	208
10.1.2 渐变填充	213
10.2 坐标系统	216
10.2.1 抗锯齿渲染	216
10.2.2 坐标变换	219
10.3 其他绘制	226
10.3.1 绘制文字	226
10.3.2 绘制路径	228
10.3.3 绘制图像	230
10.3.4 复合模式	236

10.4 双缓冲绘图	237
10.5 绘图中的其他问题	240
10.5.1 重绘事件	240
10.5.2 剪切	241
10.5.3 读取和写入图像	241
10.5.4 播放 gif 动画	241
10.5.5 渲染 SVG 文件	241
10.6 小结	242
第 11 章 图形视图、动画和状态机框架	243
11.1 图形视图框架的结构	243
11.1.1 场景	244
11.1.2 视图	245
11.1.3 图形项	247
11.2 图形视图框架的坐标系统和事件处理	249
11.2.1 坐标系统	249
11.2.2 事件处理与传播	254
11.3 图形视图框架的其他特性	258
11.3.1 图形效果	258
11.3.2 动画、碰撞检测和图形项组	260
11.3.3 打印和使用 OpenGL 进行渲染	263
11.3.4 窗口部件、布局和内嵌部件	264
11.4 动画框架	266
11.4.1 实现属性动画	267
11.4.2 使用缓和曲线	268
11.4.3 动画组	268
11.4.4 在图形视图框架中使用动画	270
11.5 状态机框架	272
11.5.1 创建状态机	272
11.5.2 在状态机中使用动画	274
11.5.3 状态机框架的其他特性	275
11.6 小结	282
第 12 章 3D 绘图	283
12.1 使用 OpenGL 绘制图形	283
12.2 设置颜色	286
12.3 实现 3D 图形	286
12.4 使用纹理贴图	288
12.5 在 3D 场景中绘制 2D 图形	290

12.6 小结.....	292
--------------	-----

影音媒体篇

第 13 章 Qt 多媒体应用	294
13.1 使用 QSound 播放声音	294
13.2 使用 QMovie 播放动画	296
13.3 多媒体的底层控制	299
13.4 小结	303
第 14 章 Phonon 多媒体框架	304
14.1 Phonon 多媒体框架的架构	304
14.1.1 Phonon 媒体图中的节点	305
14.1.2 播放后端	306
14.2 播放音频	306
14.2.1 实现简单的音频播放	306
14.2.2 创建音频流媒体图	307
14.2.3 使用音频效果	307
14.3 播放视频	308
14.3.1 实现简单的视频播放	308
14.3.2 创建播放视频的媒体图	309
14.3.3 控制视频播放	309
14.4 小结	315

数据处理篇

第 15 章 文件、目录和输入/输出	317
15.1 文件和目录	317
15.1.1 输入/输出设备	317
15.1.2 文件操作	318
15.1.3 目录操作	321
15.2 文本流和数据流	324
15.2.1 使用文本流读/写文本文件	324
15.2.2 使用数据流读/写二进制数据	325
15.3 其他相关类	328
15.3.1 应用程序设置	328
15.3.2 统一资源定位符	328
15.3.3 Qt 资源	329
15.3.4 缓冲区	329
15.4 小结	330

第 16 章 模型/视图编程	331
16.1 模型/视图架构	331
16.1.1 组成部分	332
16.1.2 简单的例子	333
16.2 模型类	334
16.2.1 基本概念	334
16.2.2 创建新的模型	338
16.3 视图类	344
16.3.1 基本概念	344
16.3.2 处理项目选择	345
16.4 委托类	350
16.4.1 基本概念	350
16.4.2 自定义委托	351
16.5 项目视图的便捷类	353
16.5.1 QListWidget	354
16.5.2 QTreeWidget	355
16.5.3 QTableWidget	356
16.5.4 共同特性	357
16.6 在项目视图中启用拖放	357
16.6.1 在便捷类中启用拖放	357
16.6.2 在模型/视图类中启用拖放	359
16.7 其他内容	362
16.7.1 代理模型	362
16.7.2 数据—窗口映射器	363
16.8 小 结	365
第 17 章 数据库和 XML	366
17.1 数据库	366
17.1.1 连接到数据库	367
17.1.2 执行 SQL 语句	372
17.1.3 使用 SQL 模型类	376
17.2 XML	384
17.2.1 DOM	384
17.2.2 SAX	394
17.2.3 XML 流	398
17.3 小 结	402

网络通信篇

第 18 章 网络编程	404
18.1 HTTP	404
18.2 FTP	408
18.3 获取网络接口信息	416
18.4 UDP	419
18.5 TCP	422
18.6 小结	436
第 19 章 进程和线程	437
19.1 进程	437
19.1.1 运行一个进程	437
19.1.2 进程间通信	440
19.2 线程	444
19.2.1 使用 QThread 启动线程	444
19.2.2 同步线程	447
19.2.3 可重入与线程安全	450
19.2.4 线程和 QObject	452
19.3 小结	454
第 20 章 WebKit	455
20.1 QtWebKit 模块	455
20.2 基于 QtWebKit 的网页浏览器	456
20.2.1 显示一个网页	456
20.2.2 显示网站图标	459
20.2.3 显示历史记录	461
20.2.4 链接跳转和查找功能	463
20.3 小结	464
参考文献	465

基本应用篇

- 第 1 章 Qt Creator 简介
- 第 2 章 Hello World
- 第 3 章 窗口部件
- 第 4 章 布局管理
- 第 5 章 应用程序主窗口
- 第 6 章 事件系统
- 第 7 章 Qt 对象模型与容器类
- 第 8 章 界面外观
- 第 9 章 国际化、帮助系统和 Qt 插件

第 1 章

Qt Creator 简介

Qt Creator 是一个跨平台的、完整的 Qt 集成开发环境，其中包括了高级 C++ 代码编辑器、项目和生成管理工具、集成的上下文相关的帮助系统、图形化调试器、代码管理和浏览工具等。这一章先对 Qt Creator 的下载安装和界面环境进行简单介绍，然后打开并运行一个 Qt 示例程序来让读者了解 Qt Creator 的基本使用方法，其中会重点介绍帮助模式的使用。这一章中没有涉及代码的编写。

1.1 Qt Creator 的下载与安装

下面从 Qt 和 Qt Creator 的下载与安装讲起，正式带读者开始 Qt 的学习之旅。需要说明一下，这里的主要开发平台是 Windows 桌面平台，所以下面主要讲解 Windows 版本的 Qt Creator。

1.1.1 下载软件

为了避免由于开发环境的版本差异而产生不必要的问题，推荐在学习本书前下载和本书相同的软件版本。这里采用了 Qt 和 Qt Creator 分别下载和安装的方式，这样可以随意选择其版本。

■ Qt 使用 4.7.2 版本：

从 <http://get.qt.nokia.com/qt/source/> 下载 qt-winopensource-4.7.2-mingw.exe 文件。

■ Qt Creator 使用 2.1.0 版本：

从 <http://get.qt.nokia.com/qtcreator/> 下载 qt-creator-win-opensource-2.1.0.exe 文件。

1.1.2 安装软件

先安装 Qt Creator。双击运行 qt-creator-win-opensource-2.1.0.exe,然后按照默认设置安装即可(注意:如果要改变安装目录,那么安装路径中不能有中文)。这里需要说明的是,组件选择(Choose Component)界面有一项 Post mortem debugging,其中的描述表明它将把 Qt Creator 作为 post mortem debugger,如图 1-1 所示。就是说,如果程序运行崩溃时,可以调用 Qt Creator 进行调试。一般情况下该项选择与否不会影响以后的编程。

当 Qt Creator 安装完成后,双击运行 qt-win-opensource-4.7.2-mingw.exe 安装 Qt 框架。整个过程也是按默认设置即可。在 MinGW 安装(MinGW Installation)界面需要指定 MinGW 的路径,它在刚才安装的 Qt Creator 目录下,如果上面使用的是默认路径,那么应该是 C:\Qt\qtcreator-2.1.0\mingw,如图 1-2 所示。

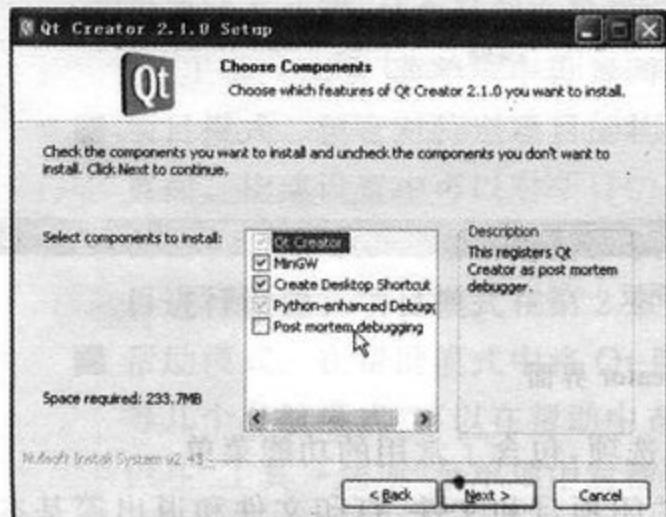


图 1-1 组件选择界面

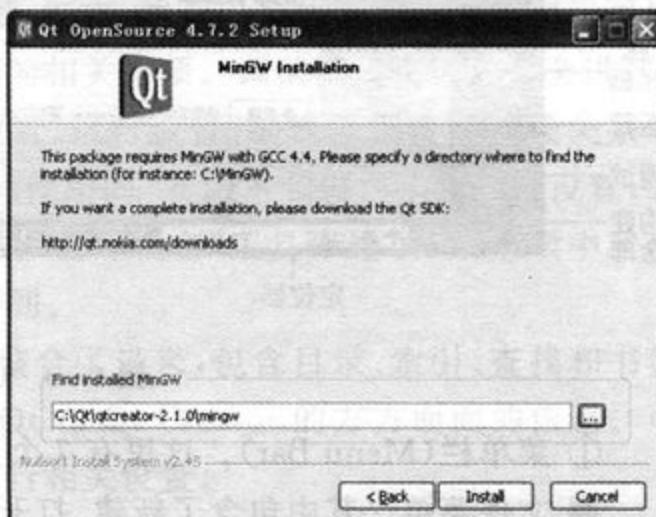


图 1-2 MinGW 安装界面



提示 MinGW 即 Minimalist GNU For Windows,是将 GNU 开发工具移植到 Win32 平台下的产物,是一套 Windows 上的 GNU 工具集。用其开发的程序不需要额外的第三方 DLL 支持就可以直接在 Windows 下运行。更多内容请查看 <http://www.mingw.org>。

1.2 Qt Creator 环境介绍

下面先简单介绍 Qt Creator 的界面组成,然后演示一个示例程序,并简单介绍 Qt Creator 的环境。

打开 Qt Creator,界面如图 1-3 所示。它主要由主窗口区、菜单栏、模式选择器、常用按钮、定位器和输出面板等部分组成,简单介绍如下:

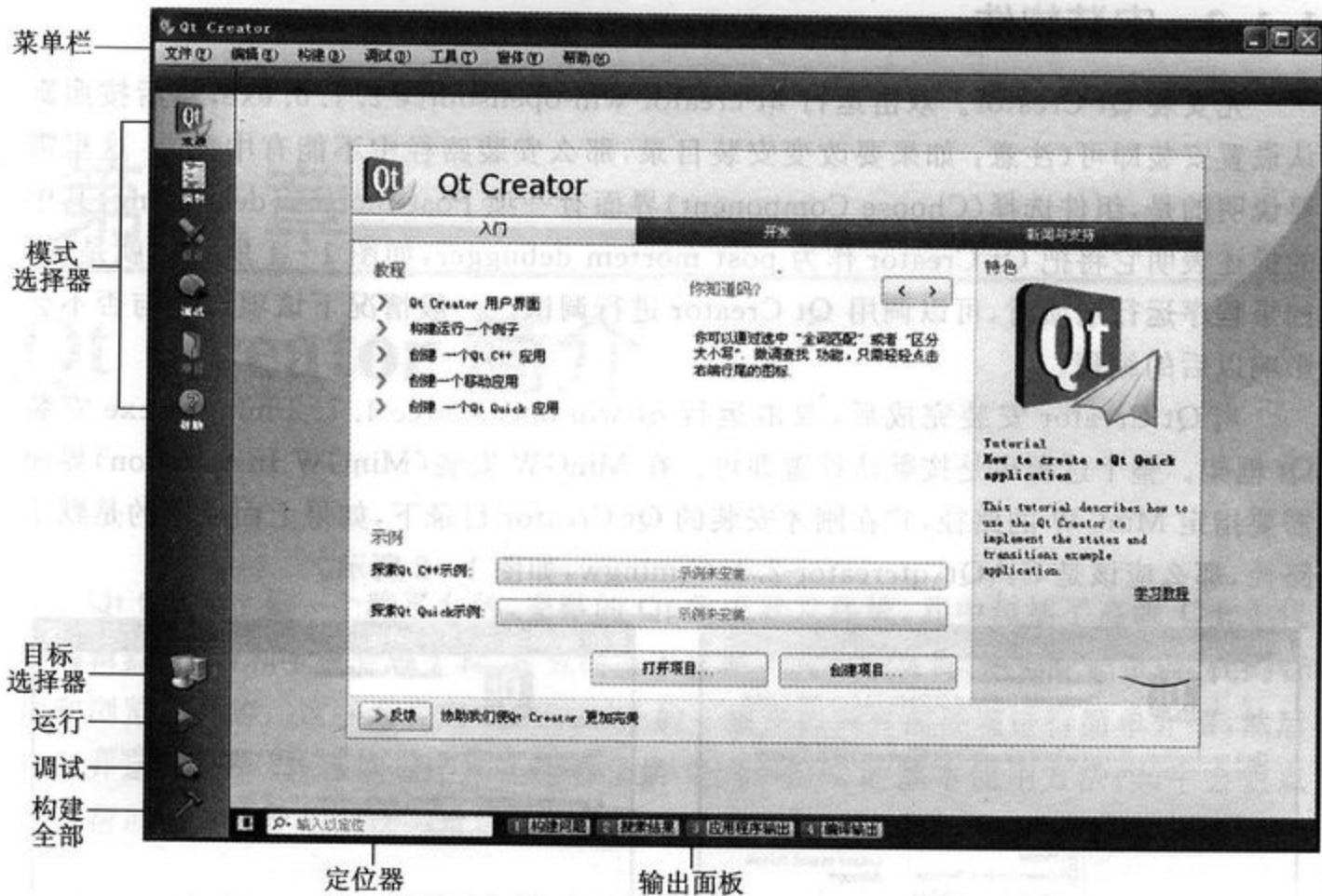


图 1-3 Qt Creator 界面

① 菜单栏(Menu Bar)。这里有 7 个菜单选项,包含了常用的功能菜单。

- 文件菜单。其中包含了新建、打开和关闭项目和文件、打印文件和退出等基本功能菜单。
- 编辑菜单。这里有撤销、剪切、复制和查找等常用功能菜单,在高级菜单中还有标示空白符、折叠代码、改变字体大小和使用 vim 风格编辑等功能菜单。
- 构建菜单。包含构建和运行项目等相关的菜单。
- 调试菜单。包含调试程序等相关的功能菜单。
- 工具菜单。这里提供了快速定位菜单、版本控制工具菜单和界面编辑器菜单等。这里的选项菜单中包含了 Qt Creator 各个方面的设置选项:环境设置、快捷键设置、编辑器设置、帮助设置、Qt 版本设置、Qt 设计师设置和版本控制设置等。
- 窗体菜单。这里包含了设置窗口布局的一些菜单,如全屏显示和隐藏边栏等。
- 帮助菜单。包含 Qt 帮助、Qt Creator 版本信息和插件管理等菜单。

② 模式选择器(Mode Selector)。Qt Creator 包含欢迎、编辑、设计、调试、项目和帮助 6 个模式,各个模式完成不同的功能,也可以使用快捷键来更换模式,它们对应的快捷键依次是 Ctrl+数字 1~6。

- 欢迎模式。图 1-3 就是欢迎模式。这里主要提供了一些功能的快捷入口,如

打开帮助教程、打开示例程序、打开项目、新建项目、快速打开以前的项目和会话、联网查看 Qt Labs 网站的新闻和打开 Qt 相关网站的链接。这里还可以快捷反馈 Qt Creator 的使用问题。

- 编辑模式。这里主要用来查看和编辑程序代码，管理项目文件。Qt Creator 中的编辑器具有关键字特殊颜色显示、代码自动补全、声明定义间快捷切换、函数原型提示、F1 键快速打开相关帮助和全项目中进行查找等功能。也可以在“工具→选项”菜单项中对编辑器进行设置。
- 设计模式。这里整合了 Qt 设计师的功能。可以在这里设计图形界面，进行部件属性设置、信号和槽设置、布局设置等操作。如果是在 Qt Quick 项目中，还可以激活 Quick 设计器，那是全新的设计器界面。可以在“工具→选项”菜单项中对设计师进行设置。设计模式在第 2 章会讲到。
- 调试模式。Qt Creator 默认使用 Gdb 进行调试，支持设置断点、单步调试和远程调试等功能，包含局部变量和监视器、断点、线程以及快照等查看窗口。可以在“工具→选项”菜单项中设置调试器的相关选项。调试模式在第 3 章会讲到。
- 项目模式。包含对特定项目的构建设置、运行设置、编辑器设置和依赖关系等页面。构建设置中可以对项目的版本、使用的 Qt 版本和编译步骤进行设置；编辑器设置中可以设置文件的默认编码。也可以在“工具→选项”菜单项中对项目进行设置。项目模式在第 2 章会讲到。
- 帮助模式。在帮助模式中将 Qt 助手整合了进来，包含目录、索引、查找和书签等几个导航模式，可以在帮助中查看 Qt 和 Qt Creator 的方方面面的信息。可以在“工具→选项”菜单项中对帮助进行相关设置。

③ 常用按钮。包含了目标选择器(Target selector)、运行按钮(Run)、调试按钮(Debug)和构建全部按钮(Build all)4 个图标。目标选择器用来选择要构建哪个平台的项目，这对于多个 Qt 库的项目很有用。这里还可以选择编译项目的 debug 版本或是 release 版本。运行按钮可以实现项目的构建和运行；调试按钮可以进入调试模式，开始调试程序；构建全部按钮可以构建所有打开的项目。

④ 定位器(Locator)。在 Qt Creator 中可以使用定位器来快速定位项目、文件、类、方法、帮助文档以及文件系统。可以使用过滤器来更加准确地定位要查找的结果，可以在“工具→选项”菜单项中设置定位器的相关选项。定位器在第 4 章会讲到。

⑤ 输出面板(Output panes)。这里包含了构建问题、搜索结果、应用程序输出和编译输出 4 个选项，它们分别对应一个输出窗口。构建问题窗口显示程序编译时的错误和警告信息；搜索结果窗口显示执行了搜索操作后的结果信息；应用程序输出窗口显示在应用程序运行过程中输出的所有信息；编译输出窗口显示程序编译过程输出的相关信息。

在图 1-3 中还可以看到现在的示例程序尚不可以使用，那是因为现在的 Qt Creator 还没有和 Qt 库进行关联。下面便来进行这个操作。

1.2.1 运行一个示例程序

现在将 Qt Creator 与 Qt 库进行连接。选择“工具→选项”菜单项，然后选择 Qt4 项。因为这里是分别下载安装的 Qt 和 Qt Creator，它们并没有自动关联，需要读者手动设置。单击右上方的图标，输入“版本名称”为“4.7.2”。然后选择“qmake 路径”，如果前面选择的是默认安装路径，则是“C:\Qt\4.7.2\bin\qmake.exe”。后面是 MinGw 目录的路径，这里是“C:\Qt\qtcreator-2.1.0\mingw”。对于“调试助手”一项，可以单击后面的“重新构建”按钮，这样 Qt Creator 会自动重新构建调试助手（关于调试助手，可以在帮助索引中查看 debugging helpers 关键字），这可能需要几分钟时间。最终如图 1-4 所示。完成后，按下“确定”按钮关闭对话框。此时又回到 Qt Creator 的欢迎模式。

设定 Qt 库以后，示例程序已经可以选择。可以看到，所有示例程序分为 Qt C++ 和 Qt Quick 两类，而各类中又根据功能的不同分为多个小类，在这里几乎可以找到 Qt 支持的所有功能的演示程序。下面选择一个 C++ 示例，例如“Animation Framework”中的“Animated Tiles”示例，单击它则弹出目标选择对话框，因为现在只有一个桌面版本的 Qt 4.7.2，所以单击“完成”按钮即可，如图 1-5 所示。

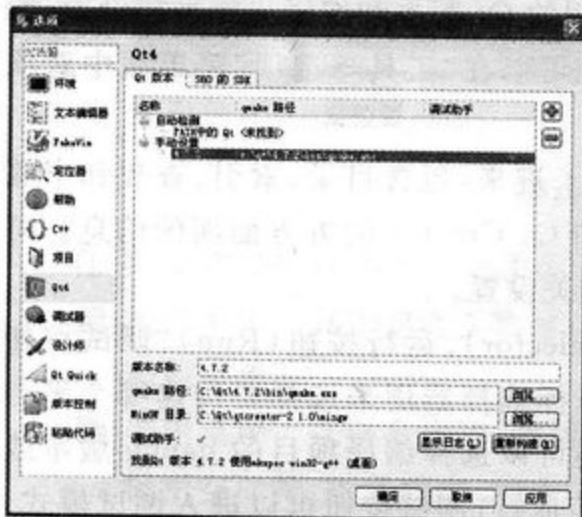


图 1-4 Qt 版本设置

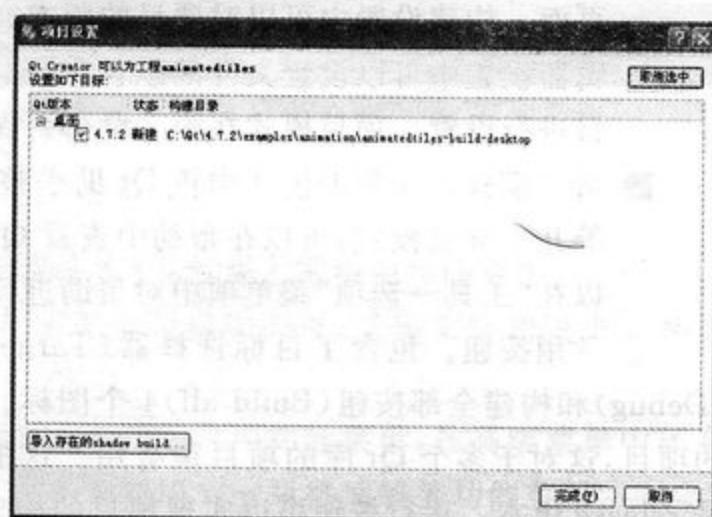


图 1-5 项目设置对话框

这时便进入了编辑模式。每当打开一个示例程序，Qt Creator 便会自动打开该程序的项目文件，然后进入编辑模式，并且打开该示例的帮助文件。可以在项目文件列表中查看该示例的源代码。现在单击左下角的运行按钮，程序便开始编译运行，在下面的“应用程序输出”栏会显示程序的运行信息和调试输出信息，如图 1-6 所示。

这里需要提醒读者的是，最好不要在示例程序中直接进行修改。如果想按照自己的想法更改，那么应该先对项目文件夹进行备份。这里可以使用编辑模式中的快捷视图来打开该程序的项目目录。在左上方的边栏中选择“文件系统”视图，如图 1-7 所示。这时便会显示项目目录的文件列表，在该目录文件名 animatedtiles 上右击，在弹出的菜单中选择“在 Explorer 中显示”，如图 1-8 所示，这样就会在新窗口中打开该项目目录了，可以先将该目录进行备份。

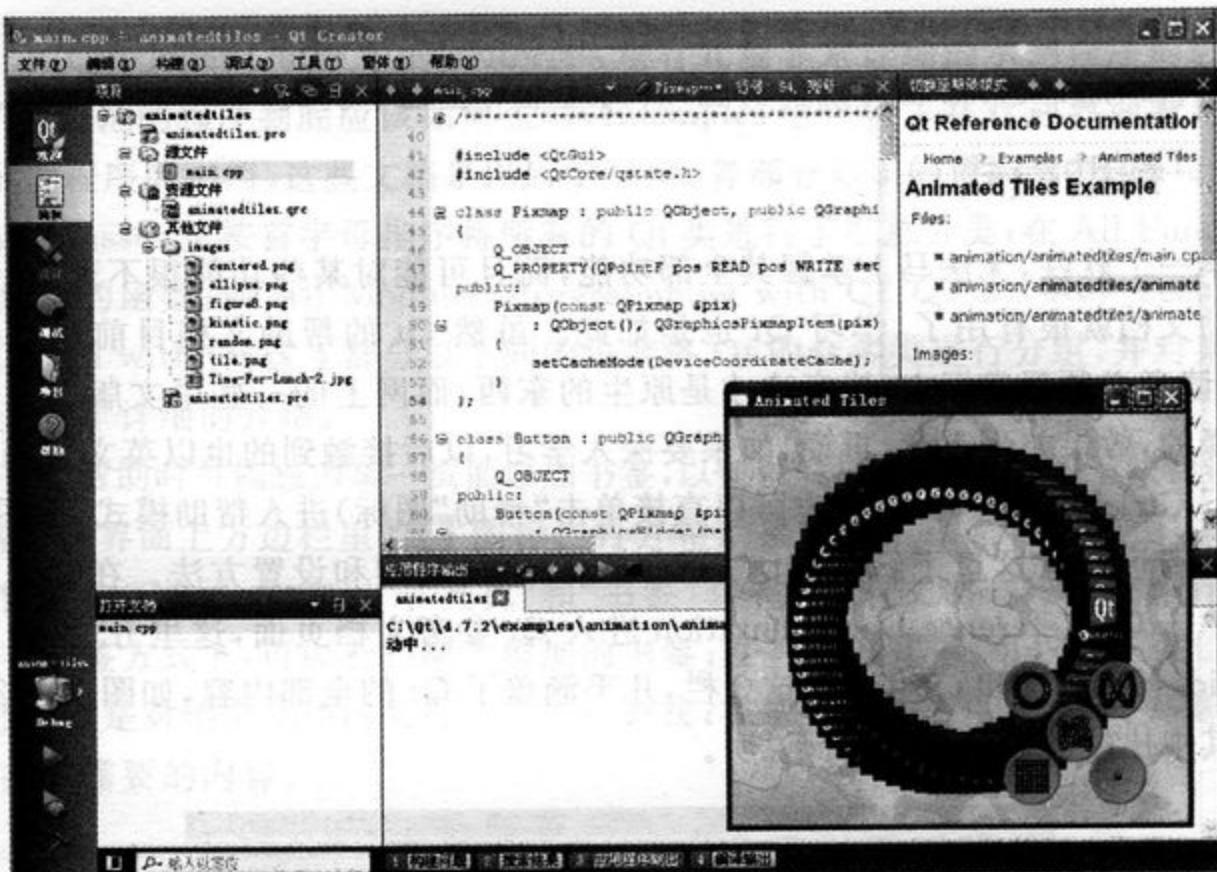


图 1-6 编辑模式

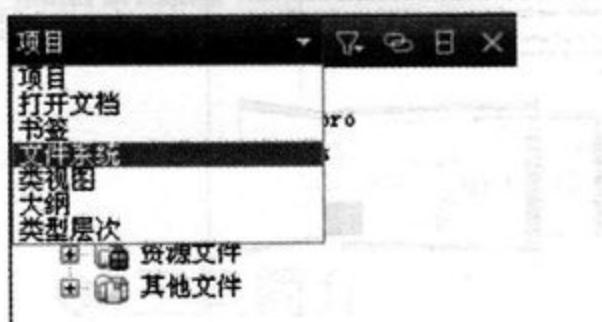


图 1-7 选择文件系统视图

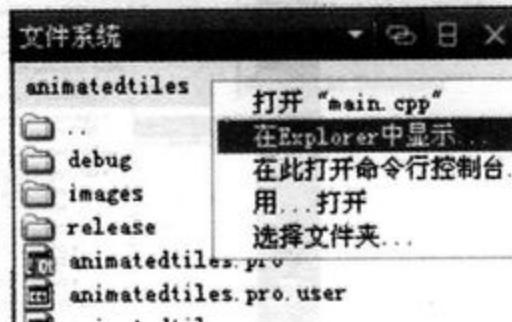


图 1-8 打开工程目录

这里还有一个“**大纲**”视图,可以使用它显示该文件中的所有类、函数和变量,并且可以快速定位,如图 1-9 所示。不过必须保证“与编辑器同步”图标是被选中的,这样选择列表中的一个函数才会自动在编辑器中定位。其实,也可以使用编辑器上方的“选择符号”下拉框来定位文件中的函数和变量,如图 1-10 所示。这个功能对于浏览或者编辑代码都很有用,希望读者掌握。还有一个功能就是在打开了多个文件后,可以

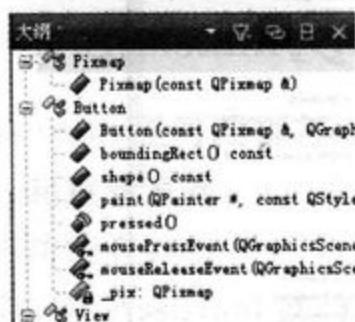


图 1-9 大纲视图

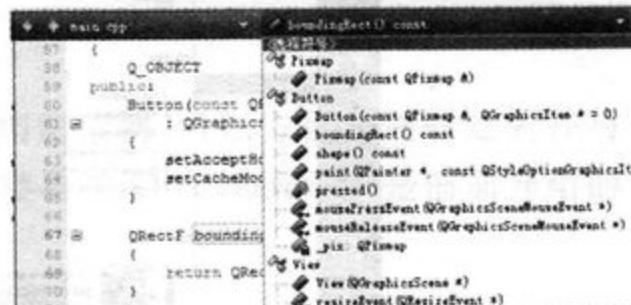


图 1-10 选择符号

在打开文档列表中进行切换，更方便的是使用 **Ctrl+Tab** 快捷键，可以在多个打开的文档间切换。当然，编辑器还有很多功能，后面的章节会逐个讲到。

1.2.2 帮助模式

初学一个软件，无法马上掌握其全部功能，而且可能对某些功能很不理解，这时软件的帮助文档就很有用了，学习 Qt 也是如此。虽然 Qt 的帮助文档目前还是全英文的，但是读者必须要掌握它，毕竟这才是原生的东西，而网上的一些中文版本是广大爱好者翻译的，效果差强人意，再说，如果要深入学习，以后接触到的也以英文文档居多。

按下 **Ctrl+6** 组合键（当然也可以直接单击“帮助”图标）进入帮助模式，如图 1-11 所示。读者可以从这里了解到 Qt Creator 更详细的使用和设置方法。在左上方的目录栏中单击 Qt Reference Documentation 进入 Qt 参考文档页面，这里分为 Qt Developer Guide、Qt API 和 Qt Tools 这 3 栏，几乎涵盖了 Qt 的全部内容，如图 1-12 所示。下面对其中比较重要的内容进行说明。

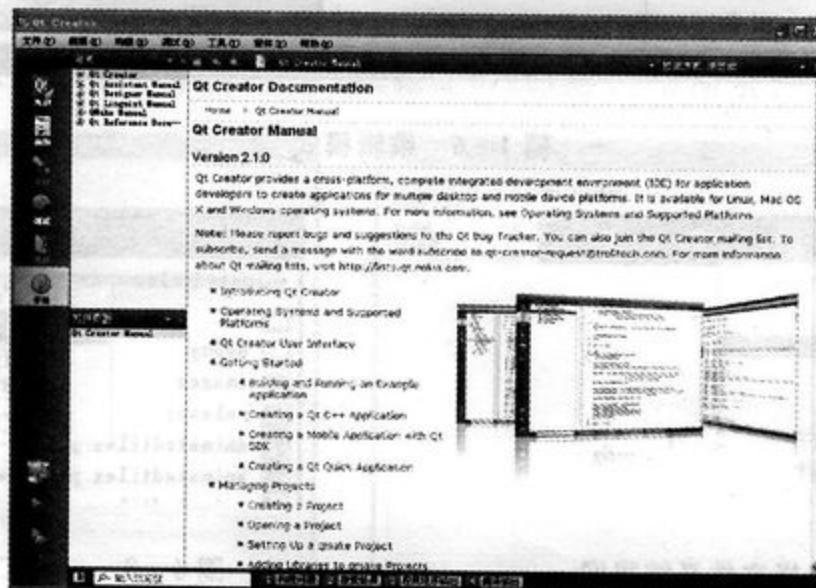


图 1-11 帮助模式

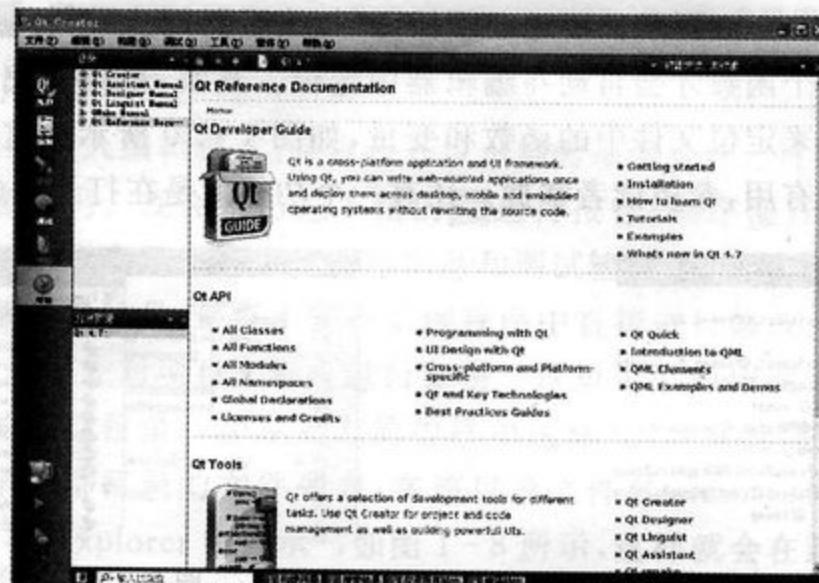


图 1-12 Qt 参考文档

在 Qt Developer Guide 中的 **Tutorials** 里有几篇非常详细的示例程序的教程可以参考,了解规范的程序到底应该如何去写;**Examples** 里包含了在欢迎界面看到的所有 C++示例程序的文档,这些文档是以后学习 Qt 各部分知识的重要参考;在 **Qt API** 一栏中,All Classes 里按首字母排序将所有的 Qt 类进行了汇总分类;在 All Functions 里列举了所有的函数;而 All Modules、Programming with Qt、Qt and Key Technologies 和 UI Design with Qt 这 4 部分以不同的角度将 Qt 的众多类进行分组,并对某一方面的应用进行了详细的介绍。

在查看帮助时可能想为某一页面添加书签,以便以后再看,则可以按下快捷键 Ctrl+W,或者单击界面上方边栏里的 图标。打开帮助模式时默认是 **目录视图**,其实帮助的工具窗口中还提供了“索引”、“查找”和“书签”3 种方式对文档进行导航,如图 1-13 所示。在书签方式下,可以看到刚才添加的书签;在查找方式下,可以输入关键字进行全文检索,就是对整个文档的所有文章进行查找;在索引方式下,只要输入关键字,那么就可以找到需要的内容。

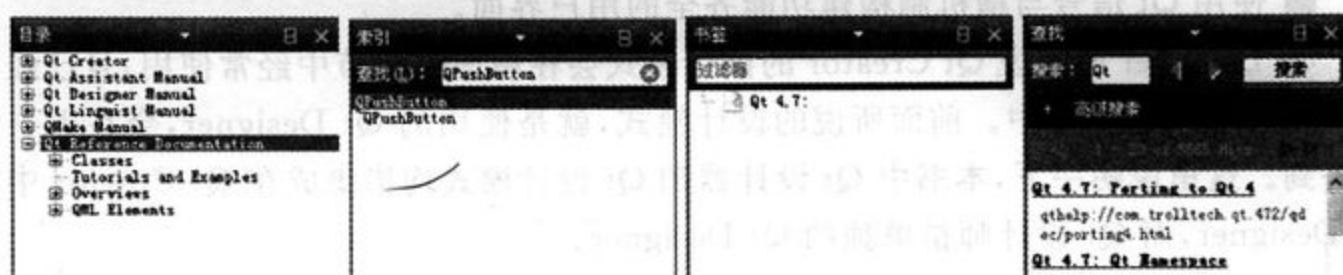


图 1-13 帮助导航模式

1.3 Qt 工具简介

前面安装的 Qt 4.7.2 中包含了几个很有用的工具,分别是 Qt Assistant(Qt 助手)、Qt Designer(Qt 设计师)、Qt Examples and Demos(Qt 演示程序与示例)和 Qt Linguist(Qt 语言家)。可以从开始菜单启动它们;当然也可以在安装目录下找到它们,笔者这里的路径是 C:\Qt\4.7.2\bin。因为前三者已经被整合到了 Qt Creator 中,而 Qt 语言家会在第 9 章中详细讲解,所以这里不会深入讲解,现在提及只是想让读者知道有这些工具,更多的相关内容可以在帮助索引中搜索它们的英文关键字。

1.3.1 Qt Assistant(Qt 助手)

Qt Assistant 是可配置且可重新发布的文档阅读器,可以方便地进行定制并与 Qt 应用程序一起重新发布。Qt Assistant 已经被整合进 Qt Creator,就是前面介绍的 Qt 帮助。它的功能有:

- 定制 Qt Assistant 并与应用程序一起重新发布。
- 快速查找关键词、全文本搜索、生成索引和书签。
- 同时为多个帮助文档集合建立索引并进行搜索。

- 在本地存放文档或在应用程序中提供在线帮助。

关于 Qt Assistant 的定制和重新发布,会在第 9 章中讲到。

1.3.2 Qt Designer(Qt 设计师)

Qt Designer 是强大的跨平台 GUI 布局和格式构建器。由于使用了与应用程序中将要使用的相同部件,可以使用屏幕上的格式快速设计、创建部件以及对话框。使用 Qt Designer 创建的界面样式功能齐全并可以预览,这样就可确保其外观完全符合要求。功能和优势有:

- 使用拖放功能快速设计用户界面。
- 定制部件或从标准部件库中选择部件。
- 以本地外观快速预览格式。
- 通过界面原型生成 C++ 或 Java 代码。
- 将 Qt Designer 与 Visual Studio 或 Eclipse IDE 配合使用。
- 使用 Qt 信号与槽机制构建功能齐全的用户界面。

Qt Designer 或者说 Qt Creator 的设计模式会在后面的章节中经常使用,且已经被整合到了 Qt Creator 中。前面所说的设计模式,就是使用的 Qt Designer,会在下一章中讲到。这里说明一下,本书中 Qt 设计器和 Qt 设计模式均指集成在 Qt Creator 中的 Qt Designer,而 Qt 设计师指单独的 Qt Designer。

1.3.3 Qt Examples and Demos(Qt 演示程序与示例)

这里列举了 Qt 自带的所有示例程序和演示程序。其中,示例程序就是前面讲到的 Qt Creator 欢迎模式中显示的示例程序,而演示程序 Demos 是一些比较综合的程序,它们在第一栏 Demonstrations 中。这里可以直接运行所有程序,例如选中一个具体示例,那么右边就会出现该示例的简单介绍和程序界面截图,单击 Launch 按钮就可以运行该程序。还可以单击 Documentation 按钮在 Qt Assistant 中打开这个程序的帮助文档。一般就是这样查看示例程序的,学习某方面的知识时,可以先查看一下相关的示例,然后在 Qt Assistant 中打开它们的帮助文档来学习,这也是学习 Qt 的一个很重要的方法!

1.3.4 Qt Linguist(Qt 语言家)

Qt Linguist 提供了一套加速应用程序翻译和国际化的工具。Qt 使用单一的源码树和单一的应用程序二进制包就可同时支持多个语言和书写系统,主要功能有:

- 收集所有 UI 文本并通过简单的应用程序提供给翻译人员。
- 语言和字体感知外观。
- 通过智能的合并工具快速为现有应用程序增加新的语言。
- Unicode 编码支持世界上大多数字母。
- 在运行时可切换从左向右或从右向左的语言。

■ 在一个文档中混合多种语言。

可以使用 Qt Linguist 使应用程序支持多种语言,这个将会在后面的第 9 章中具体介绍。

1.4 小结

本章简单介绍了 Qt Creator 的下载、安装以及 Qt 示例程序的运行。最重要的是要掌握 Qt 帮助的使用,因为在后面的章节里几乎每个知识点都要使用 Qt 的帮助索引来查找关键字。目的不仅是要读者掌握一个知识,更多的是要告诉读者一种学习方法,告诉这个知识点应该怎样学习,所以使用帮助就显得格外重要了。

第 2 章

Hello World

这章将从一个 Hello World 程序讲起,先讲述一个 GUI(Graphical User Interface,图形用户界面)项目的创建、运行和发布的过程;然后再将整个项目分解,从单一的主函数文件,到使用图形界面.ui 文件,再到自定义 C++ 类和 Qt 图形界面类,一步一步分析解释每行代码,并从命令行进行编译运行,让读者清楚地看到 Qt Creator 创建、管理、编译和运行项目的内部实现。学完本章,读者能够掌握 Qt 项目建立、编译、运行和发布的整个过程。

2.1 编写 Hello World 程序

Hello World 程序就是让应用程序显示“Hello World”字符串。这是最简单的应用,但却包含了一个应用程序的基本要素,所以一般使用它来演示程序的创建过程。本节要讲的就是在 Qt Creator 中创建一个图形用户界面的项目,来生成一个可以显示“Hello World”字符串的程序。

2.1.1 新建 Qt Gui 应用

(项目源码路径: src\02\2-1\helloworld)首先运行 Qt Creator,然后通过下面的步骤创建 Qt Gui 项目:

第一步,选择项目模板。选择“文件→新建文件或工程”菜单项(也可以直接按下 Ctrl+N 快捷键,或者单击欢迎模式中的“创建项目”按钮),在选择模板页面选择 Qt C++ 项目中的“Qt Gui 应用”一项,然后单击“选择”按钮,如图 2-1 所示。

第二步,输入项目信息。在“项目介绍和位置”页面输入项目的名称为 helloworld,然后单击创建

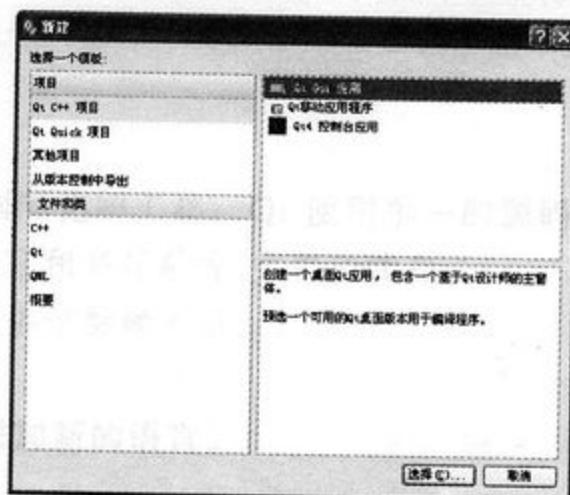


图 2-1 选择模板

路径右边的“浏览”按钮，在F盘中新建文件夹，命名为“2-1”，然后单击“确定”即可，如图2-2所示。单击“下一步”进入下个页面。（注意：项目名和路径中都不能出现中文。）

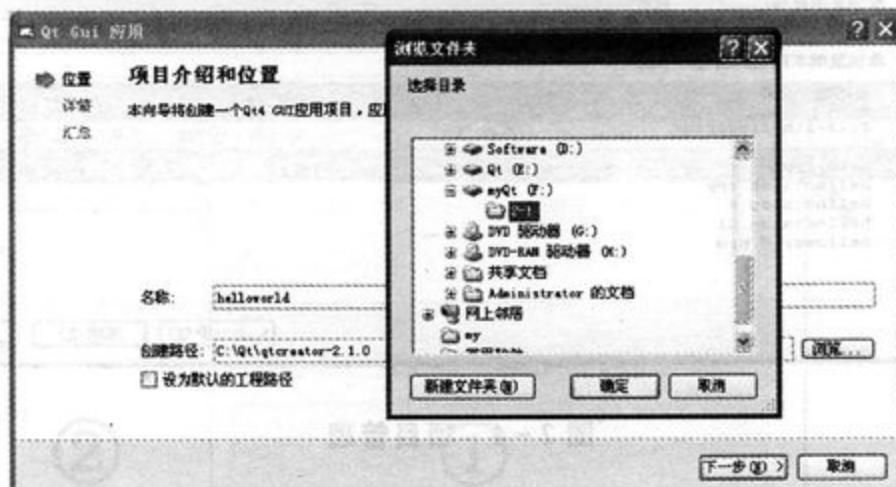


图2-2 项目介绍与位置

第三步，输入类信息。在“类信息”页面中创建一个自定义类。这里设定类名为 **HelloDialog**，基类选择 **QDialog**，表明该类继承自 **QDialog** 类，使用这个类可以生成一个对话框界面。这时下面的头文件、源文件和界面文件都会自动生成，保持默认即可，如图2-3所示，然后单击“下一步”。

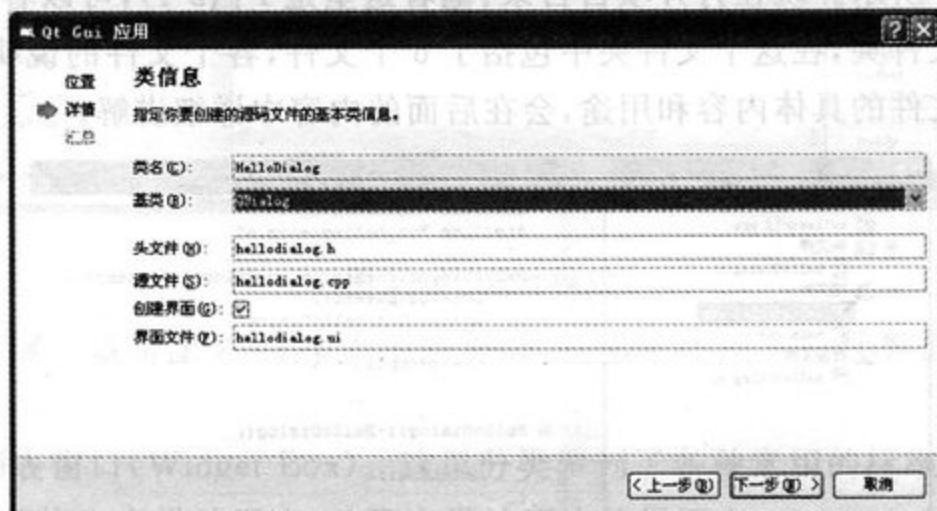


图2-3 类信息

第四步，设置项目管理。在这里可以看到这个项目的汇总信息，还可以使用版本控制系统，这个项目不会涉及，所以可以直接单击“下一步”。如图2-4所示。

第五步，完成项目设置。这个在上一章中已经提到过了，因为这里只有一个Windows桌面版本的Qt 4.7.2，所以默认选择它就可以了。然后单击“完成”按钮完成项目的创建。

注意：本书中将“工程”、“项目”和“应用”作为同义词，都指的是某个项目；“目录”与“文件夹”也作为同义词。与本小节内容相对照的内容可以在帮助索引中查看 **Creating a Qt C++ Application** 关键字。

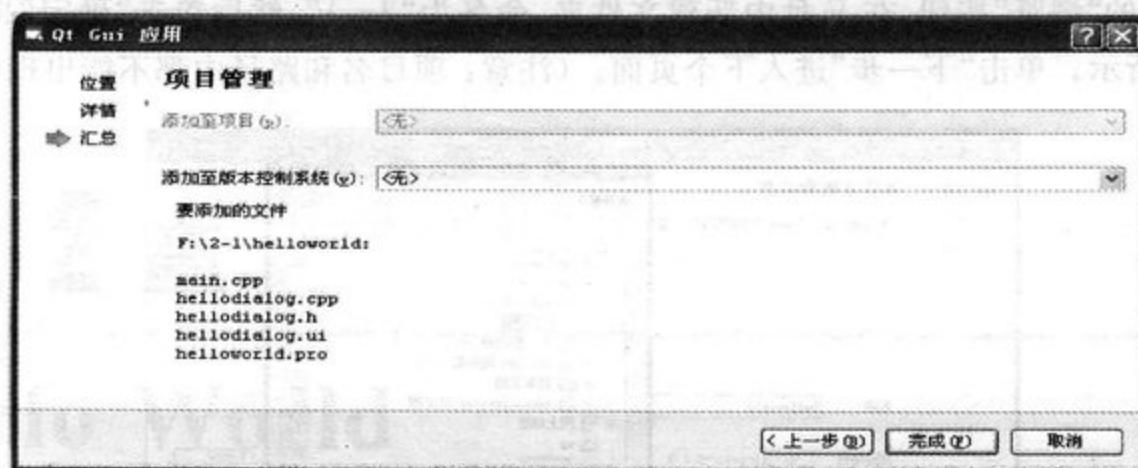


图 2-4 项目管理

2.1.2 文件说明与界面设计

项目建立完成后会直接进入编辑模式。界面的右边是编辑器，可以阅读和编辑代码。如果觉得**字体太小**，可以使用快捷键 **Ctrl+“+”**(即同时按下 Ctrl 键和十号键)来放大字体，使用 **Ctrl+“-”**(减号)来缩小字体，或者使用 **Ctrl 键 + 鼠标滚轮**，使用 **Ctrl + 0(数字)**可以使字体**还原到默认大小**。再来看左边侧边栏，其中罗列了项目中的所有文件，如图 2-5 所示。现在打开项目目录(笔者这里是 F:\2-1)，可以看到现在只有一个 helloworld 文件夹，在这个文件夹中包括了 6 个文件，各个文件的说明如表 2-1 所列。关于这些文件的具体内容和用途，会在后面的内容中详细讲解。

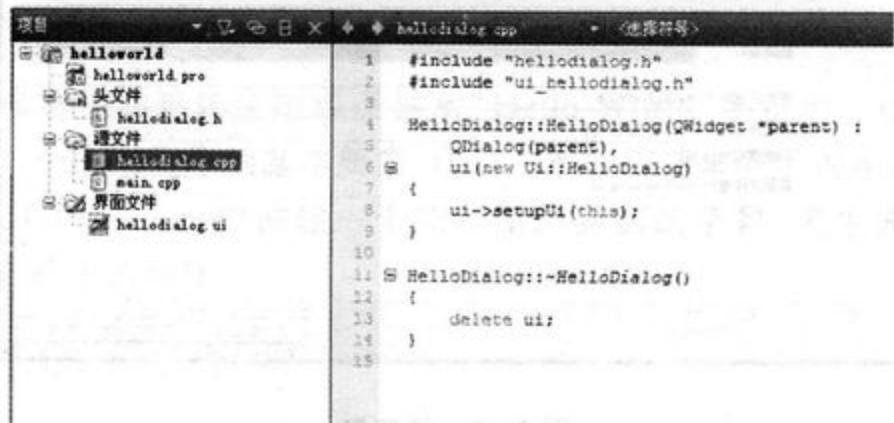


图 2-5 编辑模式

表 2-1 项目目录中各个文件说明

文件	说明
helloworld.pro	该文件是项目文件，其中包含了项目相关信息
helloworld.pro.user	该文件中包含了与用户有关的项目信息
hellodialog.h	该文件是新建的 HelloDialog 类的头文件
hellodialog.cpp	该文件是新建的 HelloDialog 类的源文件
main.cpp	该文件中包含了 main() 主函数
hellodialog.ui	该文件是设计师设计的界面对应的界面文件

在 Qt Creator 的编辑模式下双击项目文件列表中界面文件分类下的 hellodialog.ui 文件, 这时便进入了设计模式, 如图 2-6 所示。可以看到设计模式由以下几部分构成:

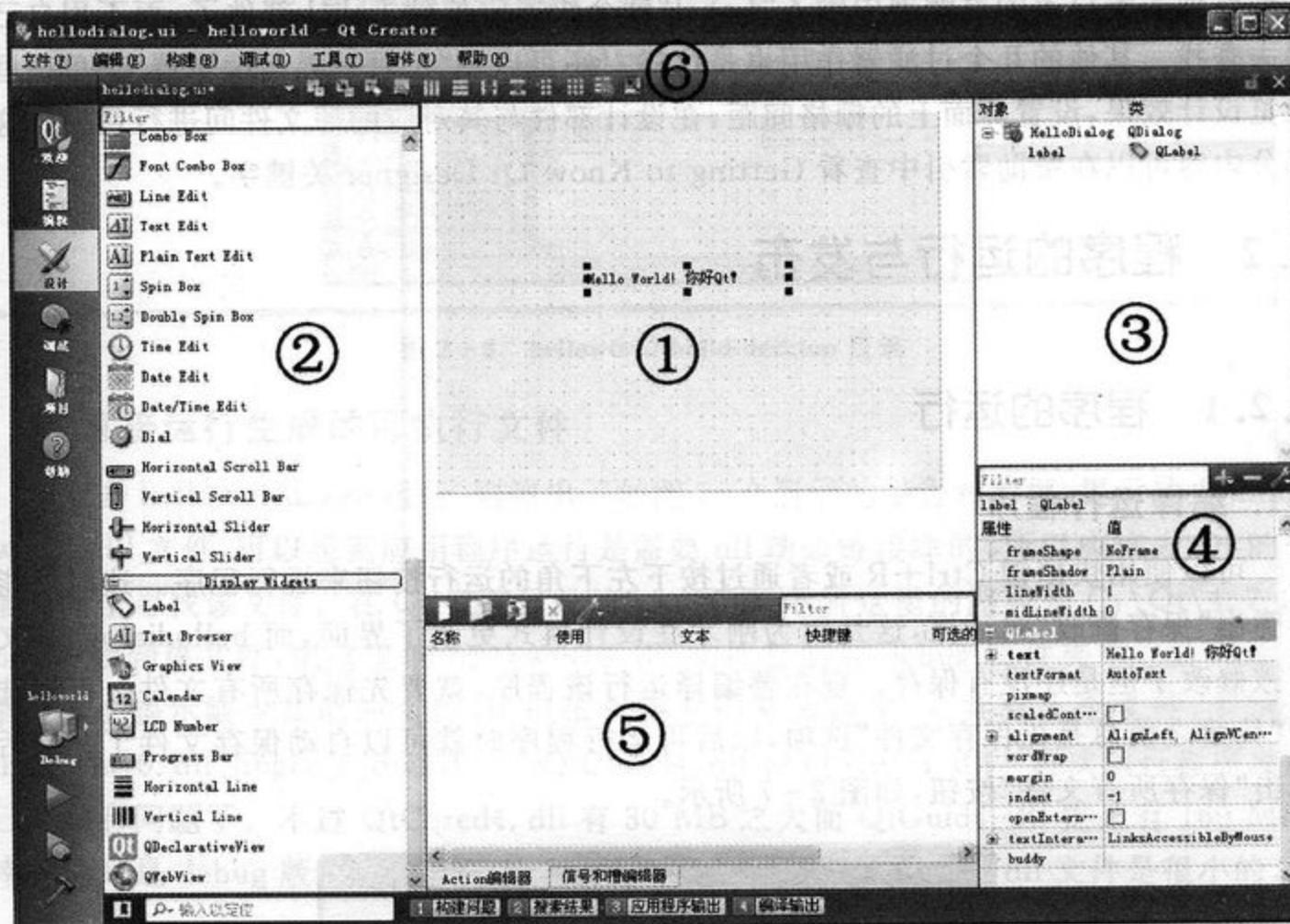


图 2-6 设计模式

① 主设计区。就是图 2-6 中的中间部分, 这里主要用来设计界面以及编辑各个部件的属性。

② 部件列表窗口(Widget Box)。这里分类罗列了各种常用的标准部件, 可以使用鼠标将这些部件拖入主设计区中, 放到主设计区中的界面上。

③ 对象查看器(Object Inspector)。这里列出了界面上所有部件的对象名称和父类, 而且以树形结构显示了各个部件的所属关系。可以在这里单击对象来选中该部件。

④ 属性编辑器(Property Editor)。这里显示了各个部件的常用属性信息, 可以在这里更改部件的一些属性, 如大小、位置等。这些属性按照从祖先继承的属性、从父类继承的属性和自己的属性的顺序进行了分类。

⑤ 动作(Action)编辑器与信号/槽编辑器。在这里可以对相应的对象内容进行编辑。因为现在还没有涉及这些内容, 所以放到以后使用时再介绍。

⑥ 常用功能图标。单击最上面的侧边栏中的前 4 个图标可以进入相应的模式, 分别是窗口部件编辑模式(这是默认模式)、信号/槽编辑模式、伙伴编辑模式和 Tab 顺序编辑模式。后面的几个图标用来实现添加布局管理器以及调整大小等功能。

下面从部件列表中找到 Label(标签)部件,然后按着鼠标左键将它拖到主设计区的界面上,再双击它进入编辑状态后输入“Hello World! 你好 Qt!”字符串。这里顺便说一下,Qt Creator 的设计模式中有几个过滤器,就是写着“Filter”的行输入框。例如在部件列表窗口上的过滤器中输入“Label”就会快速定位到 Label 部件了,而不用自己再去查找。其他的几个过滤器作用也是这样。还可以在“工具→界面编辑器”菜单项里预览设计效果,设置界面上的栅格间距,在设计部件与其对应的源文件间进行切换。这部分内容可以在帮助索引中查看 [Getting to Know Qt Designer 关键字](#)。

2.2 程序的运行与发布

2.2.1 程序的运行

1. 编译运行程序

可以使用快捷键 **Ctrl+R** 或者通过按下左下角的运行按钮来运行程序。这时可能会弹出“保存修改”对话框,这是因为刚才在设计模式更改了界面,而 `hellodialog.ui` 文件被修改了但是还没有保存。现在要编译运行该程序,就要先保存所有文件。可以选中“构建之前总是先保存文件”选项,以后再运行程序时就可以自动保存文件了。然后单击“保存所有文件”按钮,如图 2-7 所示。

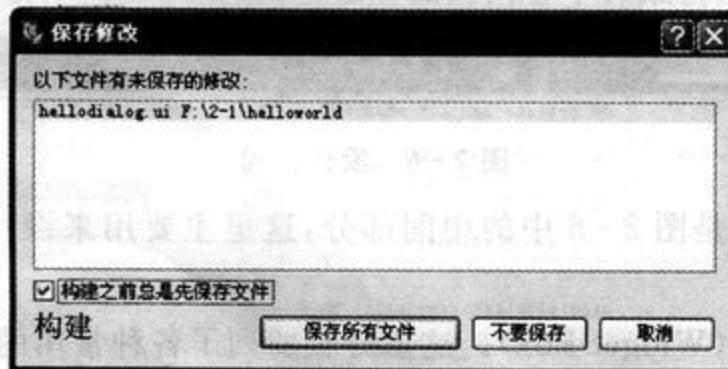


图 2-7 保存修改对话框

2. 查看构建项目生成的文件

下面再看一下现在项目目录中的文件可以发现,`F:\2-1` 目录下又多了一个 `helloworld-build-desktop` 文件夹,这是默认的构建目录。也就是说,现在 Qt Creator 将项目源文件和编译生成的文件进行了分类存放,`helloworld` 文件夹中是项目源文件,而现在这个文件夹存放的是编译后生成的文件。进入该文件夹可以看到这里有 3 个 Makefile 文件和一个 `ui_hellodialog.h` 文件,还有两个目录 `debug` 和 `release`,如图 2-8 所示。现在 `release` 文件夹是空的,进入 `debug` 文件夹,这里有 3 个 `.o` 文件和一个 `.cpp` 文件,它们是编译时生成的中间文件,可以不必管它,而剩下的一个 `helloworld.exe` 文件便是生成的可执行文件。



图 2-8 helloworld-build-desktop 目录

3. 直接运行生成的可执行文件

双击 helloworld.exe 运行，则弹出了如图 2-9 所示的警告对话框，提示缺少 mingwm10.dll 文件，可以想到应用程序运行是需要 dll 动态链接库的，所以应该去 Qt 的安装目录下寻找该文件。在 Qt 安装目录的 bin 目录（笔者这里的路径是 C:\Qt\4.7.2\bin）中找到该文件，把这里的 mingwm10.dll 文件复制到 debug 文件夹中。这时运行程序又会提示缺少其他的文件，可以依次将它们复制过来，一共有 4 个文件，分别是 mingwm10.dll、libgcc_s_dw2-1.dll、QtCore4.dll 和 QtGuid4.dll。再次运行程序发现已经没有问题了。不过 QtCore4.dll 有 30 MB 之大而 QtGuid4.dll 更是有 160 MB，幸好这只是 debug 版，后面讲解的程序发布时的 release 版需要的 dll 文件是很小的。

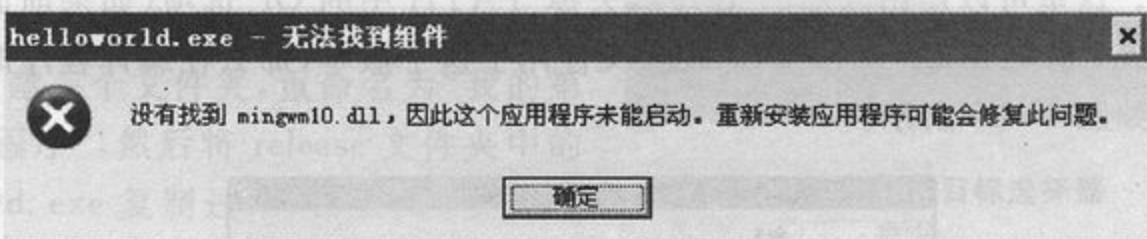


图 2-9 警告框

那么有没有别的办法，不需要移动 dll 文件就可以直接运行程序呢？其实可以直接将 C:\Qt\4.7.2\bin 目录加入到系统 Path 环境变量中去，这样程序运行时就可以自动找到 bin 目录中的 dll 文件了。具体做法是右击“我的电脑”，在弹出的菜单上选择“属性”，然后在弹出的系统属性对话框中选择“高级”一项，如图 2-10 所示。单击“环境变量”按钮进入环境变量设置界面。在“系统变量”栏中找到 Path 变量，单击“编辑”弹出编辑系统变量对话框。在变量值的最后添上“;C:\Qt\4.7.2\bin”（注意前面有一个英文的分号），然后确定即可，如图 2-11 所示。

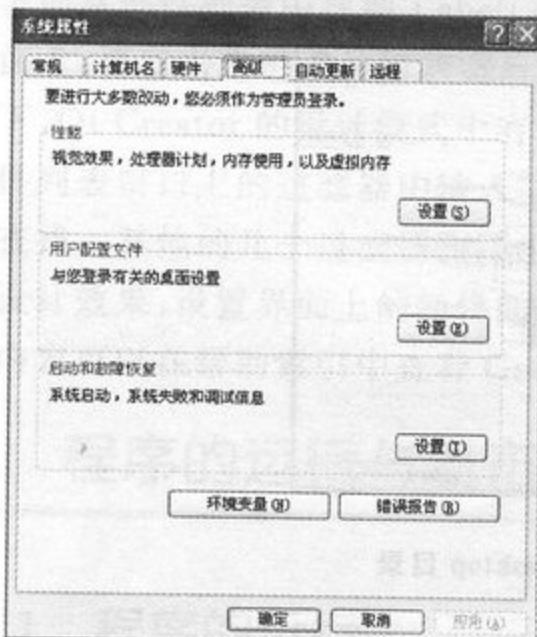


图 2-10 系统属性对话框



图 2-11 编辑系统变量对话框

4. 添加 PATH 环境变量

现在删除那些 dll 文件，再次运行 helloworld.exe 文件，发现已经可以正常运行了。而这时重新启动 Qt Creator，再到“工具→选项”菜单项中查看 Qt 4 一项，可以发现这时 Qt Creator 已经自动检测到了 PATH 中的 Qt 版本，如图 2-12 所示。当然，这个与前面手动设置的是同一个版本，也可以在这里为它设置一下 MinGW 目录，这样以后再新建项目时，会出现两个 Qt 版本的选择，因为两个版本是完全相同的，所以随便选择一个即可。这里可以还用以前的 4.7.2，去掉“PATH 中的 Qt”选项（如果前面没有给 path 中的 Qt 指定 MinGW 目录，而现在又使用了这个版本，那么在编译程序时就会出错）即可，如图 2-13 所示。

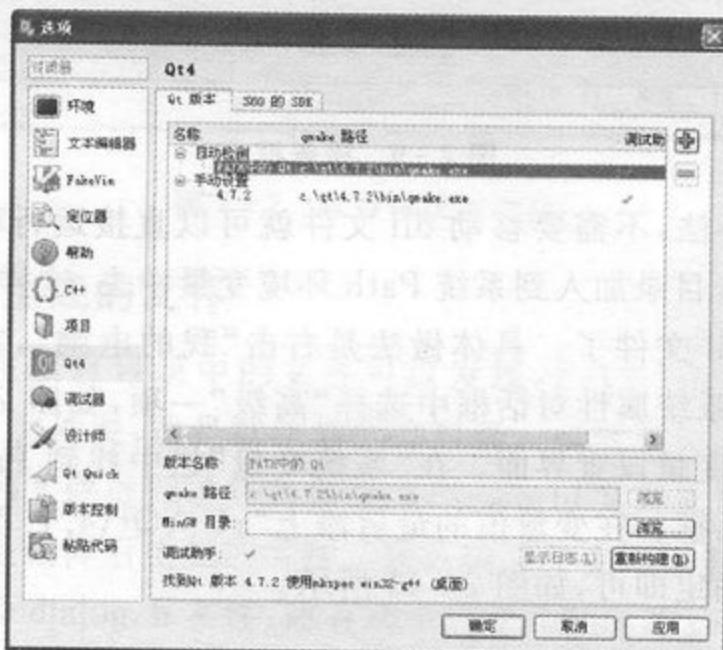


图 2-12 Qt 选项对话框

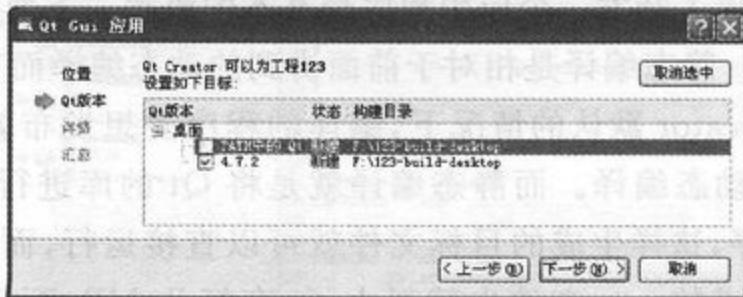


图 2-13 Qt 版本选择对话框

2.2.2 程序的发布

现在程序已经编译完成,那么怎样来发布它,让它在别人的计算机上也能运行呢?前面生成的 debug 版本的程序依赖的 dll 文件很大,那是因为 debug 版本的程序中包含了调试信息,可以用来调试。而真正要发布程序时,要使用 release 版本。下面回到 Qt Creator 中对 helloworld 程序进行 release 版本的编译。在左下角的目标选择器 (Target selector) 中将构建目标设置为 Release(如图 2-14 所示),然后单击运行图标。编译完成之后再看工程目录中的 release 目录中,已经生成了 helloworld.exe 文件。可以看一下它的大小,只有 60 多 KB,而前面的 debug 版的 helloworld.exe 却有 700 多 KB,相差很大。如果前面已经添加了 Path 系统环境变量,那么现在就可以直接双击运行该程序。如果要使现在的 Release 版本的程序可以在别人的计算机上运行(当然,对方计算机也要是 Windows 平台),那么还是需要将几个 dll 文件与其一起发布。可以在桌面上新建一个文件夹,重命名为“我的第一个 Qt 程序”,然后将 release 文件夹中的 helloworld.exe 复制过来,再去 Qt 安装目录的 bin 目录(笔者这里是 C:\Qt\4.7.2\bin)中将 mingwm10.dll、libgcc_s_dw2-1.dll、QtCore4.dll 和 QtGui4.dll 这 4 个文件(注意不是 QtCore4.dll 和 QtGui4.dll,它们是 debug 版本的库文件)复制过来。现在整个文件夹一共有 12 MB,如果使用 WinRAR 等打包压缩软件对它进行压缩,就只有 4 MB 了,已经到达了可以接受的程度,这时就可以将压缩包发布出去了。

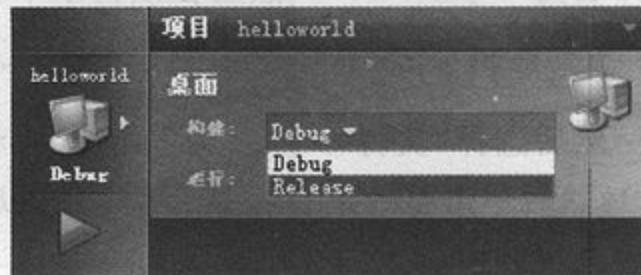


图 2-14 目标选择器

提示

若程序中使用了 png 以外格式的图片,在发布程序时就要将 Qt 安装目录下 plugins 目录中的 imageformats 文件夹复制到发布程序文件夹中,其中只要保留自己用到的文件格式的 dll 文件即可。例如用到了 gif 文件,那么只需要保留 qgif4.dll。而如果程序中使用了其他的模块,比如 Phonon,那么就要将 plugins 目录中的 phonon_backend 文件夹复制过来。

到这里就已经完成了开发一个应用程序最基本的流程。下面再来看一个经常被提到的概念：静态编译。静态编译是相对于前面讲到的动态编译而言的。因为就像前面看到的一样，在 Qt Creator 默认的情况下，编译的程序要想发布就需要包含 dll 文件，这种编译方式被称为动态编译。而静态编译就是将 Qt 的库进行重新编译，用静态编译的 Qt 库来链接程序，这样生成的目标文件就可以直接运行，而不再需要 dll 文件的支持了。不过这样生成的 exe 文件也就很大了，有好几 MB，而且静态编译缺乏灵活性，也不能够部署插件。从前面的介绍可以看到，其实发布程序时带几个 dll 文件并不是很复杂的事情，而且如果要同时发布多个应用程序还可以共用 dll 文件，所以使用默认的方式就可以了。想了解更多 Qt 发布的知识和静态编译的方法，可以在 Qt Creator 帮助的索引方式下查看 Deploying Qt Applications 关键字，Windows 平台发布程序对应的关键字是 Deploying an Application on Windows。

2.2.3 设置应用程序图标

在程序发布时，也想使 exe 文件可以有一个漂亮的图标。在 Qt Creator 的帮助索引中查找 Setting the Application Icon 关键字，这里列出了在 Windows 上设置应用程序图标的方法，步骤如下：

第一步，创建.ico 文件。将 ico 图标文件复制到工程文件夹的 helloworld 目录中，重命名为“myico.ico”。然后在该目录中右击，新建文本文档，并输入一行代码：

```
IDI_ICON1 ICON DISCARDABLE "myico.ico"
```

然后选择“文件→另存为”菜单项，将该文件命名为 myico.rc（注意文件后缀为 .rc），然后单击“保存”。完成后可以将以前的“新建文本文档”删除。最后 helloworld 文件夹中的内容如图 2-15 所示。

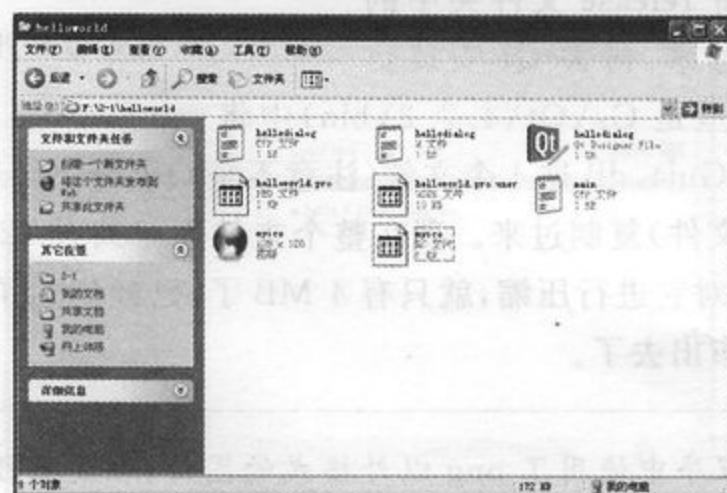


图 2-15 helloworld 目录

第二步，修改项目文件。在 Qt Creator 中的编辑模式双击 helloworld.pro 文件，在最后面添加下面一行代码，如图 2-16 所示。

```
RC_FILE += myico.rc
```



```

helloWorld
├── helloWorld.pro
├── 头文件
│   └── helldialog.h
├── 源文件
│   ├── helldialog.cpp
│   └── main.cpp
└── 界面文件
    └── helldialog.ui

#-
#
# Project created by QtCreator 2011-02-28T17:47:16
#
#
QT      += core gui
TARGET = helloWorld
TEMPLATE = app
SOURCES += main.cpp \
           helldialog.cpp
HEADERS += helldialog.h
FORMS   += helldialog.ui
RC_FILE += myico.ico

```

图 2-16 编辑工程文件

第三步,运行程序。可以看到窗口的左上角的图标已经更换了,如图 2-17 所示。然后查看一下 release 文件夹中的文件,可以看到现在 exe 文件已经更换了新的图标,如图 2-18 所示。

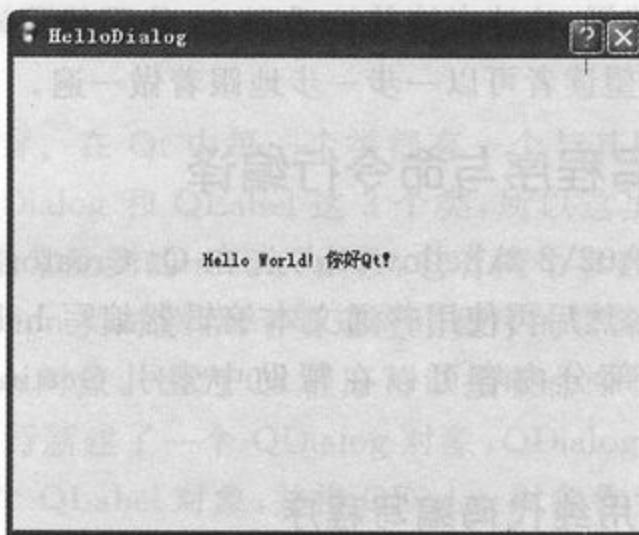


图 2-17 更换了图标的程序运行界面

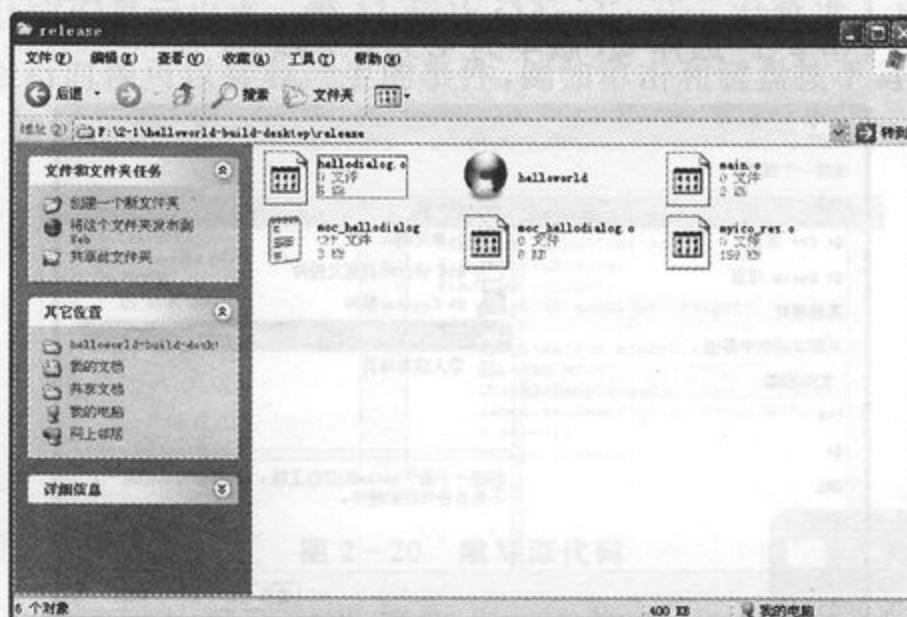


图 2-18 更换了图标后的 release 目录

现在只需要将这里的 helloworld.exe 复制到程序发布文件夹中, 就可以得到一个完整而且漂亮的的应用程序。对于整个流程而言, 最后这步“设置应用程序图标”并不是必须的。这就是程序从创建到发布的整个过程。

2.3 helloworld 程序源码与编译过程详解

前面建立了最简单的 helloworld 项目, 可以看到, 在 Qt Creator 中只需要很简单的几步就可以创建出一个图形用户界面的程序, Qt Creator 已经做好了绝大多数的工作。但是生成的项目目录中的各个文件都是什么? 它们有什么作用? 相互之间有什么联系? 还有 Qt 程序到底是怎么编译运行的? 解决这些问题对于学习 Qt 编程至关重要。

下面将分步骤地使用多种方法来创建 helloworld 程序, 从最开始的纯代码编写, 到使用 .ui 文件, 再到添加自定义类, 最后还原到 2.1 节建立的 helloworld 程序。其中还会使用命令行进行代码编译, 让读者清楚地看到 Qt 代码的编译过程。再次说明一下, 这一节的内容很重要, 希望读者可以一步一步地跟着做一遍。

2.3.1 纯代码编写程序与命令行编译

(项目源码路径: src\02\2-2\helloworld) 先在 Qt Creator 中使用纯代码编写 helloworld 程序并编译运行, 然后再使用普通文本编辑器编写 helloworld 程序, 并在命令行中编译运行。对应该部分内容可以在帮助中索引 Getting Started Programming with Qt 关键字。

1. 在 Qt Creator 中用纯代码编写程序

第一步, 新建空项目。打开 Qt Creator, 并新建项目, 选择“其他项目”中的“空的 Qt 项目”。如图 2-19 所示。然后将项目命名为“helloworld”, 设置路径为 F 盘中新建的 2-2 文件夹即“F:\2-2”。选择 Qt 版本为 4.7.2 一项。

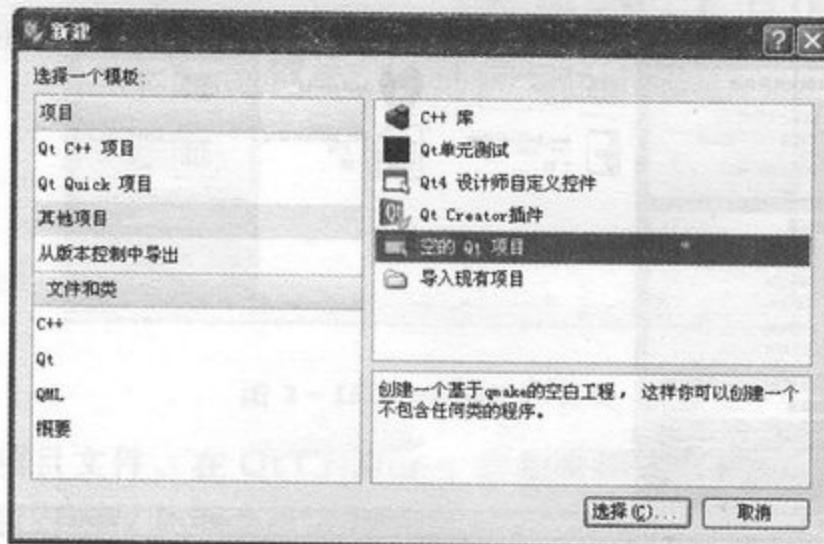


图 2-19 新建文件

第二步,往项目中添加 main.cpp 文件。在项目文件列表中的工程文件夹 helloworld 上右击,选择“添加新文件”一项,然后选择 C++ 源文件,名称改为“main.cpp”,路径就是默认的项目目录,后面的选项保持默认即可。

第三,编写源代码。向新建的 main.cpp 文件中添加如下代码:

```

1 #include < QApplication>
2 #include < QDialog>
3 #include < QLabel>
4 int main(int argc, char * argv[])
5 {
6     QApplication a(argc, argv);
7     QDialog w;
8     QLabel label(&w);
9     label.setText("Hello World! 你好 Qt!");
10    w.show();
11    return a.exec();
12 }
```

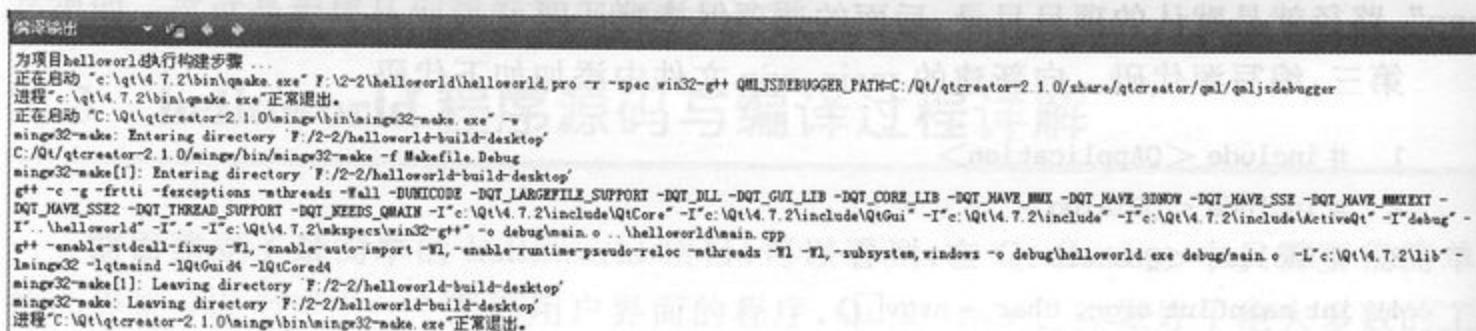
前 3 行是头文件包含。在 Qt 中每一个类都有一个与其同名的头文件,因为后面用到了 QApplication、QDialog 和 QLabel 这 3 个类,所以这里要包含这些类的定义。第 4 行就是在 C++ 中最常见到的 main() 函数,它有两个参数,用来接收命令行参数。第 6 行新建了 QApplication 类对象,用于管理应用程序的资源,任何一个 Qt GUI 程序都要有一个 QApplication 对象。因为 Qt 程序可以接收命令行参数,所以它需要 argc 和 argv 两个参数。第 7 行新建了一个 QDialog 对象,QDialog 类用来实现一个对话框界面。第 8 行新建了一个 QLabel 对象,并将 QDialog 对象作为参数,表明了对话框是它的父窗口,也就是说这个标签放在对话框窗口中。第 9 行给标签设置要显示的字符。第 10 行让对话框显示出来。在默认情况下,新建的可视部件对象都是不可见的,要使用 show() 函数让它们显示出来。第 11 行让 QApplication 对象进入事件循环,这样当 Qt 应用程序在运行时便可以接收产生的事件,例如单击和键盘按下等事件。这一点可以参考一下第 6 章的相关内容。编写好代码后,如图 2-20 所示。

```

1 // 项目
2 // helloworld
3 //   helloworld.pro
4 //   资源文件
5 //     main.cpp
6
7 // main.cpp
8
9 // 1 main.cpp
10 // 2
11 // 3
12 // 4
13 // 5
14 // 6
15 // 7
16 // 8
17 // 9
18 // 10
19 // 11
20 // 12
21 // 13
22 // 14
23 // 15
24 // 16
25 // 17
26 // 18
27 // 19
28 // 20
29 // 21
30 // 22
31 // 23
32 // 24
33 // 25
34 // 26
35 // 27
36 // 28
37 // 29
38 // 30
39 // 31
40 // 32
41 // 33
42 // 34
43 // 35
44 // 36
45 // 37
46 // 38
47 // 39
48 // 30
49 // 31
50 // 32
51 // 33
52 // 34
53 // 35
54 // 36
55 // 37
56 // 38
57 // 39
58 // 30
59 // 31
60 // 32
61 // 33
62 // 34
63 // 35
64 // 36
65 // 37
66 // 38
67 // 39
68 // 30
69 // 31
70 // 32
71 // 33
72 // 34
73 // 35
74 // 36
75 // 37
76 // 38
77 // 39
78 // 30
79 // 31
80 // 32
81 // 33
82 // 34
83 // 35
84 // 36
85 // 37
86 // 38
87 // 39
88 // 30
89 // 31
90 // 32
91 // 33
92 // 34
93 // 35
94 // 36
95 // 37
96 // 38
97 // 39
98 // 30
99 // 31
100 // 32
101 // 33
102 // 34
103 // 35
104 // 36
105 // 37
106 // 38
107 // 39
108 // 30
109 // 31
110 // 32
111 // 33
112 // 34
113 // 35
114 // 36
115 // 37
116 // 38
117 // 39
118 // 30
119 // 31
120 // 32
121 // 33
122 // 34
123 // 35
124 // 36
125 // 37
126 // 38
127 // 39
128 // 30
129 // 31
130 // 32
131 // 33
132 // 34
133 // 35
134 // 36
135 // 37
136 // 38
137 // 39
138 // 30
139 // 31
140 // 32
141 // 33
142 // 34
143 // 35
144 // 36
145 // 37
146 // 38
147 // 39
148 // 30
149 // 31
150 // 32
151 // 33
152 // 34
153 // 35
154 // 36
155 // 37
156 // 38
157 // 39
158 // 30
159 // 31
160 // 32
161 // 33
162 // 34
163 // 35
164 // 36
165 // 37
166 // 38
167 // 39
168 // 30
169 // 31
170 // 32
171 // 33
172 // 34
173 // 35
174 // 36
175 // 37
176 // 38
177 // 39
178 // 30
179 // 31
180 // 32
181 // 33
182 // 34
183 // 35
184 // 36
185 // 37
186 // 38
187 // 39
188 // 30
189 // 31
190 // 32
191 // 33
192 // 34
193 // 35
194 // 36
195 // 37
196 // 38
197 // 39
198 // 30
199 // 31
200 // 32
201 // 33
202 // 34
203 // 35
204 // 36
205 // 37
206 // 38
207 // 39
208 // 30
209 // 31
210 // 32
211 // 33
212 // 34
213 // 35
214 // 36
215 // 37
216 // 38
217 // 39
218 // 30
219 // 31
220 // 32
221 // 33
222 // 34
223 // 35
224 // 36
225 // 37
226 // 38
227 // 39
228 // 30
229 // 31
230 // 32
231 // 33
232 // 34
233 // 35
234 // 36
235 // 37
236 // 38
237 // 39
238 // 30
239 // 31
240 // 32
241 // 33
242 // 34
243 // 35
244 // 36
245 // 37
246 // 38
247 // 39
248 // 30
249 // 31
250 // 32
251 // 33
252 // 34
253 // 35
254 // 36
255 // 37
256 // 38
257 // 39
258 // 30
259 // 31
260 // 32
261 // 33
262 // 34
263 // 35
264 // 36
265 // 37
266 // 38
267 // 39
268 // 30
269 // 31
270 // 32
271 // 33
272 // 34
273 // 35
274 // 36
275 // 37
276 // 38
277 // 39
278 // 30
279 // 31
280 // 32
281 // 33
282 // 34
283 // 35
284 // 36
285 // 37
286 // 38
287 // 39
288 // 30
289 // 31
290 // 32
291 // 33
292 // 34
293 // 35
294 // 36
295 // 37
296 // 38
297 // 39
298 // 30
299 // 31
300 // 32
301 // 33
302 // 34
303 // 35
304 // 36
305 // 37
306 // 38
307 // 39
308 // 30
309 // 31
310 // 32
311 // 33
312 // 34
313 // 35
314 // 36
315 // 37
316 // 38
317 // 39
318 // 30
319 // 31
320 // 32
321 // 33
322 // 34
323 // 35
324 // 36
325 // 37
326 // 38
327 // 39
328 // 30
329 // 31
330 // 32
331 // 33
332 // 34
333 // 35
334 // 36
335 // 37
336 // 38
337 // 39
338 // 30
339 // 31
340 // 32
341 // 33
342 // 34
343 // 35
344 // 36
345 // 37
346 // 38
347 // 39
348 // 30
349 // 31
350 // 32
351 // 33
352 // 34
353 // 35
354 // 36
355 // 37
356 // 38
357 // 39
358 // 30
359 // 31
360 // 32
361 // 33
362 // 34
363 // 35
364 // 36
365 // 37
366 // 38
367 // 39
368 // 30
369 // 31
370 // 32
371 // 33
372 // 34
373 // 35
374 // 36
375 // 37
376 // 38
377 // 39
378 // 30
379 // 31
380 // 32
381 // 33
382 // 34
383 // 35
384 // 36
385 // 37
386 // 38
387 // 39
388 // 30
389 // 31
390 // 32
391 // 33
392 // 34
393 // 35
394 // 36
395 // 37
396 // 38
397 // 39
398 // 30
399 // 31
400 // 32
401 // 33
402 // 34
403 // 35
404 // 36
405 // 37
406 // 38
407 // 39
408 // 30
409 // 31
410 // 32
411 // 33
412 // 34
413 // 35
414 // 36
415 // 37
416 // 38
417 // 39
418 // 30
419 // 31
420 // 32
421 // 33
422 // 34
423 // 35
424 // 36
425 // 37
426 // 38
427 // 39
428 // 30
429 // 31
430 // 32
431 // 33
432 // 34
433 // 35
434 // 36
435 // 37
436 // 38
437 // 39
438 // 30
439 // 31
440 // 32
441 // 33
442 // 34
443 // 35
444 // 36
445 // 37
446 // 38
447 // 39
448 // 30
449 // 31
450 // 32
451 // 33
452 // 34
453 // 35
454 // 36
455 // 37
456 // 38
457 // 39
458 // 30
459 // 31
460 // 32
461 // 33
462 // 34
463 // 35
464 // 36
465 // 37
466 // 38
467 // 39
468 // 30
469 // 31
470 // 32
471 // 33
472 // 34
473 // 35
474 // 36
475 // 37
476 // 38
477 // 39
478 // 30
479 // 31
480 // 32
481 // 33
482 // 34
483 // 35
484 // 36
485 // 37
486 // 38
487 // 39
488 // 30
489 // 31
490 // 32
491 // 33
492 // 34
493 // 35
494 // 36
495 // 37
496 // 38
497 // 39
498 // 30
499 // 31
500 // 32
501 // 33
502 // 34
503 // 35
504 // 36
505 // 37
506 // 38
507 // 39
508 // 30
509 // 31
510 // 32
511 // 33
512 // 34
513 // 35
514 // 36
515 // 37
516 // 38
517 // 39
518 // 30
519 // 31
520 // 32
521 // 33
522 // 34
523 // 35
524 // 36
525 // 37
526 // 38
527 // 39
528 // 30
529 // 31
530 // 32
531 // 33
532 // 34
533 // 35
534 // 36
535 // 37
536 // 38
537 // 39
538 // 30
539 // 31
540 // 32
541 // 33
542 // 34
543 // 35
544 // 36
545 // 37
546 // 38
547 // 39
548 // 30
549 // 31
550 // 32
551 // 33
552 // 34
553 // 35
554 // 36
555 // 37
556 // 38
557 // 39
558 // 30
559 // 31
560 // 32
561 // 33
562 // 34
563 // 35
564 // 36
565 // 37
566 // 38
567 // 39
568 // 30
569 // 31
570 // 32
571 // 33
572 // 34
573 // 35
574 // 36
575 // 37
576 // 38
577 // 39
578 // 30
579 // 31
580 // 32
581 // 33
582 // 34
583 // 35
584 // 36
585 // 37
586 // 38
587 // 39
588 // 30
589 // 31
590 // 32
591 // 33
592 // 34
593 // 35
594 // 36
595 // 37
596 // 38
597 // 39
598 // 30
599 // 31
600 // 32
601 // 33
602 // 34
603 // 35
604 // 36
605 // 37
606 // 38
607 // 39
608 // 30
609 // 31
610 // 32
611 // 33
612 // 34
613 // 35
614 // 36
615 // 37
616 // 38
617 // 39
618 // 30
619 // 31
620 // 32
621 // 33
622 // 34
623 // 35
624 // 36
625 // 37
626 // 38
627 // 39
628 // 30
629 // 31
630 // 32
631 // 33
632 // 34
633 // 35
634 // 36
635 // 37
636 // 38
637 // 39
638 // 30
639 // 31
640 // 32
641 // 33
642 // 34
643 // 35
644 // 36
645 // 37
646 // 38
647 // 39
648 // 30
649 // 31
650 // 32
651 // 33
652 // 34
653 // 35
654 // 36
655 // 37
656 // 38
657 // 39
658 // 30
659 // 31
660 // 32
661 // 33
662 // 34
663 // 35
664 // 36
665 // 37
666 // 38
667 // 39
668 // 30
669 // 31
670 // 32
671 // 33
672 // 34
673 // 35
674 // 36
675 // 37
676 // 38
677 // 39
678 // 30
679 // 31
680 // 32
681 // 33
682 // 34
683 // 35
684 // 36
685 // 37
686 // 38
687 // 39
688 // 30
689 // 31
690 // 32
691 // 33
692 // 34
693 // 35
694 // 36
695 // 37
696 // 38
697 // 39
698 // 30
699 // 31
700 // 32
701 // 33
702 // 34
703 // 35
704 // 36
705 // 37
706 // 38
707 // 39
708 // 30
709 // 31
710 // 32
711 // 33
712 // 34
713 // 35
714 // 36
715 // 37
716 // 38
717 // 39
718 // 30
719 // 31
720 // 32
721 // 33
722 // 34
723 // 35
724 // 36
725 // 37
726 // 38
727 // 39
728 // 30
729 // 31
730 // 32
731 // 33
732 // 34
733 // 35
734 // 36
735 // 37
736 // 38
737 // 39
738 // 30
739 // 31
740 // 32
741 // 33
742 // 34
743 // 35
744 // 36
745 // 37
746 // 38
747 // 39
748 // 30
749 // 31
750 // 32
751 // 33
752 // 34
753 // 35
754 // 36
755 // 37
756 // 38
757 // 39
758 // 30
759 // 31
760 // 32
761 // 33
762 // 34
763 // 35
764 // 36
765 // 37
766 // 38
767 // 39
768 // 30
769 // 31
770 // 32
771 // 33
772 // 34
773 // 35
774 // 36
775 // 37
776 // 38
777 // 39
778 // 30
779 // 31
780 // 32
781 // 33
782 // 34
783 // 35
784 // 36
785 // 37
786 // 38
787 // 39
788 // 30
789 // 31
790 // 32
791 // 33
792 // 34
793 // 35
794 // 36
795 // 37
796 // 38
797 // 39
798 // 30
799 // 31
800 // 32
801 // 33
802 // 34
803 // 35
804 // 36
805 // 37
806 // 38
807 // 39
808 // 30
809 // 31
810 // 32
811 // 33
812 // 34
813 // 35
814 // 36
815 // 37
816 // 38
817 // 39
818 // 30
819 // 31
820 // 32
821 // 33
822 // 34
823 // 35
824 // 36
825 // 37
826 // 38
827 // 39
828 // 30
829 // 31
830 // 32
831 // 33
832 // 34
833 // 35
834 // 36
835 // 37
836 // 38
837 // 39
838 // 30
839 // 31
840 // 32
841 // 33
842 // 34
843 // 35
844 // 36
845 // 37
846 // 38
847 // 39
848 // 30
849 // 31
850 // 32
851 // 33
852 // 34
853 // 35
854 // 36
855 // 37
856 // 38
857 // 39
858 // 30
859 // 31
860 // 32
861 // 33
862 // 34
863 // 35
864 // 36
865 // 37
866 // 38
867 // 39
868 // 30
869 // 31
870 // 32
871 // 33
872 // 34
873 // 35
874 // 36
875 // 37
876 // 38
877 // 39
878 // 30
879 // 31
880 // 32
881 // 33
882 // 34
883 // 35
884 // 36
885 // 37
886 // 38
887 // 39
888 // 30
889 // 31
890 // 32
891 // 33
892 // 34
893 // 35
894 // 36
895 // 37
896 // 38
897 // 39
898 // 30
899 // 31
900 // 32
901 // 33
902 // 34
903 // 35
904 // 36
905 // 37
906 // 38
907 // 39
908 // 30
909 // 31
910 // 32
911 // 33
912 // 34
913 // 35
914 // 36
915 // 37
916 // 38
917 // 39
918 // 30
919 // 31
920 // 32
921 // 33
922 // 34
923 // 35
924 // 36
925 // 37
926 // 38
927 // 39
928 // 30
929 // 31
930 // 32
931 // 33
932 // 34
933 // 35
934 // 36
935 // 37
936 // 38
937 // 39
938 // 30
939 // 31
940 // 32
941 // 33
942 // 34
943 // 35
944 // 36
945 // 37
946 // 38
947 // 39
948 // 30
949 // 31
950 // 32
951 // 33
952 // 34
953 // 35
954 // 36
955 // 37
956 // 38
957 // 39
958 // 30
959 // 31
960 // 32
961 // 33
962 // 34
963 // 35
964 // 36
965 // 37
966 // 38
967 // 39
968 // 30
969 // 31
970 // 32
971 // 33
972 // 34
973 // 35
974 // 36
975 // 37
976 // 38
977 // 39
978 // 30
979 // 31
980 // 32
981 // 33
982 // 34
983 // 35
984 // 36
985 // 37
986 // 38
987 // 39
988 // 30
989 // 31
990 // 32
991 // 33
992 // 34
993 // 35
994 // 36
995 // 37
996 // 38
997 // 39
998 // 30
999 // 31
1000 // 32
1001 // 33
1002 // 34
1003 // 35
1004 // 36
1005 // 37
1006 // 38
1007 // 39
1008 // 30
1009 // 31
1010 // 32
1011 // 33
1012 // 34
1013 // 35
1014 // 36
1015 // 37
1016 // 38
1017 // 39
1018 // 30
1019 // 31
1020 // 32
1021 // 33
1022 // 34
1023 // 35
1024 // 36
1025 // 37
1026 // 38
1027 // 39
1028 // 30
1029 // 31
1030 // 32
1031 // 33
1032 // 34
1033 // 35
1034 // 36
1035 // 37
1036 // 38
1037 // 39
1038 // 30
1039 // 31
1040 // 32
1041 // 33
1042 // 34
1043 // 35
1044 // 36
1045 // 37
1046 // 38
1047 // 39
1048 // 30
1049 // 31
1050 // 32
1051 // 33
1052 // 34
1053 // 35
1054 // 36
1055 // 37
1056 // 38
1057 // 39
1058 // 30
1059 // 31
1060 // 32
1061 // 33
1062 // 34
1063 // 35
1064 // 36
1065 // 37
1066 // 38
1067 // 39
1068 // 30
1069 // 31
1070 // 32
1071 // 33
1072 // 34
1073 // 35
1074 // 36
1075 // 37
1076 // 38
1077 // 39
1078 // 30
1079 // 31
1080 // 32
1081 // 33
1082 // 34
1083 // 35
1084 // 36
1085 // 37
1086 // 38
1087 // 39
1088 // 30
1089 // 31
1090 // 32
1091 // 33
1092 // 34
1093 // 35
1094 // 36
1095 // 37
1096 // 38
1097 // 39
1098 // 30
1099 // 31
1100 // 32
1101 // 33
1102 // 34
1103 // 35
1104 // 36
1105 // 37
1106 // 38
1107 // 39
1108 // 30
1109 // 31
1110 // 32
1111 // 33
1112 // 34
1113 // 35
1114 // 36
1115 // 37
1116 // 38
1117 // 39
1118 // 30
1119 // 31
1120 // 32
1121 // 33
1122 // 34
1123 // 35
1124 // 36
1125 // 37
1126 // 38
1127 // 39
1128 // 30
1129 // 31
1130 // 32
1131 // 33
1132 // 34
1133 // 35
1134 // 36
1135 // 37
1136 // 38
1137 // 39
1138 // 30
1139 // 31
1140 // 32
1141 // 33
1142 // 34
1143 // 35
1144 // 36
1145 // 37
1146 // 38
1147 // 39
1148 // 30
1149 // 31
1150 // 32
1151 // 33
1152 // 34
1153 // 35
1154 // 36
1155 // 37
1156 // 38
1157 // 39
1158 // 30
1159 // 31
1160 // 32
1161 // 33
1162 // 34
1163 // 35
1164 // 36
1165 // 37
1166 // 38
1167 // 39
1168 // 30
1169 // 31
1170 // 32
1171 // 33
1172 // 34
1173 // 35
1174 // 36
1175 // 37
1176 // 38
1177 // 39
1178 // 30
1179 // 31
1180 // 32
1181 // 33
1182 // 34
1183 // 35
1184 // 36
1185 // 37
1186 // 38
1187 // 39
1188 // 30
1189 // 31
1190 // 32
1191 // 33
1192 // 34
1193 // 35
1194 // 36
1195 // 37
1196 // 38
1197 // 39
1198 // 30
1199 // 31
1200 // 32
1201 // 33
1202 // 34
1203 // 35
1204 // 36
1205 // 37
1206 // 38
1207 // 39
1208 // 30
1209 // 31
1210 // 32
1211 // 33
1212 // 34
1213 // 35
1214 // 36
1215 // 37
1216 // 38
1217 // 39
1218 // 30
1219 // 31
1220 // 32
1221 // 33
1222 // 34
1223 // 35
1224 // 36
1225 // 37
1226 // 38
1227 // 39
1228 // 30
1229 // 31
1230 // 32
1231 // 33
1232 // 34
1233 // 35
1234 // 36
1235 // 37
1236 // 38
1237 // 39
1238 // 30
1239 // 31
1240 // 32
1241 // 33
1242 // 34
1243 // 35
1244 // 36
1245 // 37
1246 // 38
1247 // 39
1248 // 30
1249 // 31
1250 // 32
1251 // 33
1252 // 34
1253 // 35
1254 // 36
1255 // 37
1256 // 38
1257 // 39
1258 // 30

```

所示。再看运行的程序，发现英文显示是正常的，但是中文却是乱码，而且窗口太小，下面更改代码来显示中文。



```
编译输出
正在启动 "c:\qt\4.7.2\bin\qmake.exe" F:/2-2/helloworld\helloworld.pro -r -spec win32-g+ QMLJSDEBUGGER_PATH=C:/Qt/qtcreator-2.1.0/share/qtcreator/qml/qmljsdebugger
进程:c:\qt\4.7.2\bin\qmake.exe正常退出。
正在启动 "C:\Qt\Qtcreator-2.1.0\mingw\bin\mingw32-make.exe" -w
mingw32-make: Entering directory 'F:/2-2/helloworld-build-desktop'
C:/Qt/qtcreator-2.1.0\mingw\bin\mingw32-make -f Makefile.Debug
mingw32-make[1]: Entering directory 'F:/2-2/helloworld-build-desktop'
g++ -c -g -fPIC -fexceptions -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT -DQT_DLL -DQT_GUI_LIB -DQT_CORE_LIB -DQT_HAVE_MMX -DQT_HAVE_3DNOW -DQT_HAVE_SSE -DQT_HAVE_MMXEXT -
DQT_HAVE_SSE2 -DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN -I"c:\Qt\4.7.2\include\QtCore" -I"c:\Qt\4.7.2\include\QtGui" -I"c:\Qt\4.7.2\include\ActiveQt" -I"debug" -
I".\helloworld" -I".\.." -I"c:\Qt\4.7.2\mkspecs\win32-g++" -o debug\main.o ..\helloworld\main.cpp
g++ -enable-stdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo-reloc -mthreads -Wl -Wl,-subsystem,windows -o debug\helloworld.exe debug\main.o -L"c:\Qt\4.7.2\lib" -
Lmingw32 -LQt\Guid4 -LQtCored4
mingw32-make[1]: Leaving directory 'F:/2-2/helloworld-build-desktop'
mingw32-make: Leaving directory 'F:/2-2/helloworld-build-desktop'
进程:C:\Qt\Qtcreator-2.1.0\mingw\bin\mingw32-make.exe"正常退出。
```

图 2-22 编译输出信息

第五，设置显示中文。更改代码如下：

```
1 # include < QApplication >
2 # include < QDialog >
3 # include < QLabel >
4 # include < QTextCodec >
5 int main( int argc, char * argv[] )
6 {
7     QApplication a( argc, argv );
8     QTextCodec::setCodecForTr( QTextCodec::codecForLocale() );
9     QDialog w;
10    QLabel label( &w );
11    label.setText( QObject::tr("Hello World! 你好 Qt!") );
12    w.show();
13    return a.exec();
14 }
```

在第 4 行添加了 QTextCodec 类的头文件包含。QTextCodec 类提供了文本编码的转换功能。第 8 行使用了 QTextCodec 类中的静态函数 setCodecForTr()，用来设置 QObject::tr() 函数所要使用的字符集，就像第 11 行中所看到的那样，tr() 函数可以使用指定的字符集来对文本编码进行转换。这里使用了 QTextCodec::codecForLocale()，它返回了系统指定的字符集。当然也可以自己指定字符集，比如使用“GB18030”，则将第 8 行改为：

```
QTextCodec::setCodecForTr( QTextCodec::codecForName("gb18030") );
```

为了能够显示中文，需要设置字符集，然后使用 QObject::tr() 函数将字符串进行编码转换。其实 tr() 函数还可以实现多语言支持，这个在第 9 章国际化部分将会讲到。需要说明的是，setCodecForTr() 函数最好的放置位置就是像程序中这样，放在 main() 函数中的 QApplication 对象下面。Qt 程序中所有要显示到界面上的字符串最好都使用 tr() 函数括起来，而对于不是要显示到界面上的字符串中如果包含了中文，可以使用 QString() 进行编码转换，这需要在主函数中添加如下代码进行设置：

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
```

下面介绍两个实用功能：第一个是**代码自动补全功能**（可以在帮助索引中查看 Using the Editor 关键字）。在编辑器中敲入代码时可以发现打完开头几个字母后就会出现相关的列表选项，这些选项都是以这些字母开头的。现在要说明的是，如果要输入一个很长的字符，比如 codecForLocale，而这里又有很多选项，那么就可以直接输入 cFL 这 3 个字母（就是 codecForLocale 中首字母加其中的**大写字母**）来快速定位它，然后按下 Enter 按键就可以完成输入，如图 2-23 所示。列表中各个图标都代表一种数据类型，如图 2-24 所示。也可以使用 Ctrl+空格键来强制代码补全，需要注意它可能与使用的输入法的快捷键冲突。

```
QApplication a(argc, argv);
QTextCodec::setCodecForTr(QTextCodec::cFL)
QDialog w;
QLabel label(&w);
label.setText(QObject::tr("Hello Wor"))
w.show();
return a.exec();
```

图 2-23 自动补全功能

第二个是**快速查看帮助**。将鼠标指针放到一个类名或函数上，便会出现一个提示框显示其简单的介绍，而这时按下 F1 键就可以在编辑器右边快速打开其帮助文档。可以单击左上角的“切换至帮助模式”进入帮助模式。

下面运行程序即可以看到，中文已经可以正常显示了。如图 2-25 所示。

第六，设置窗口大小。添加代码如下：

```
⑨ QDialog w;
⑩ w.resize(400, 300);
⑪ QLabel label(&w);
⑫ label.move(120, 120);
```

如果想改变对话框的大小，可以使用 QDialog 类中的函数来实现。在第 9 行代码下面另起一行，输入“w.”（注：w 后面输入一个“.”），这时会弹出 QDialog 类中所有成员的列表，可以使用键盘的向下方向键↓ 来浏览列表，根据字面意思，这里选定了 resize() 函数。这时按下 Enter 键，代码便自动补全，并且显示出了 resize() 函数的原型。它有两个重载形式，用键盘方向键来查看另外的形式，这里的“int w,int h”应该就是宽和高了，如图 2-26 所示。所以写出

Icon	Description
类	A class
枚举类型	An enum
枚举类型值	An enumerator
函数	A function
私有函数	A private function
受保护函数	A protected function
变量	A variable
私有变量	A private variable
受保护变量	A protected variable
信号	A signal
槽	A slot
私有槽	A private slot
受保护槽	A protected slot
C++关键字	A C++ keyword
C++代码段	A C++ code snippet
QML元素	A QML element
QML代码段	A QML code snippet
宏	A macro
命名空间	A namespace

图 2-24 类型列表

了第 10 行代码,设置对话框宽为 400,高为 300(它们的单位是像素)。还要说明的是,编写代码时所有的符号都要用输入法中的英文半角(当然,中文字符串中的除外)。然后在第 12 行代码中设置了 label 在对话框中的位置,默认的,对话框的左上角是(0, 0)点。运行程序,显示结果已经和 2.1 节中一样了。

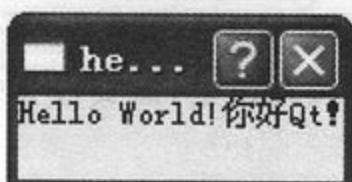


图 2-25 正常显示中文
的运行结果

```
QTextCodec::setCodecForTr(QTextCodec::  
QDialog w ▲ 1/2 ▼ void resize(const QSize &)  
w.resize();  
  
QTextCodec::setCodecForTr(QTextCodec::  
QDialog w ▲ 2/2 ▼ void resize(int w, int h)  
w.resize();
```

图 2-26 显示函数原型

2. 在命令行编译程序

前面在 Qt Creator 中使用纯代码实现了 2.1 节中的 helloworld 程序。下面不使用 Qt Creator,而是在其他的编辑器(如 Windows 的记事本)中编写源码,然后再到命令行去编译运行该程序。

第一步,新建工程目录。在 Qt 的安装目录(笔者这里是 C:\Qt)中新建文件夹 helloworld,然后在其中新建文本文档,将 Qt Creator 中 main.cpp 文件中的所有内容复制过来,并将文件另存为 main.cpp。完成后打开开始菜单中 Qt 安装目录下的命令提示符程序 Qt 4.7.2 Command Prompt,如图 2-27 所示。

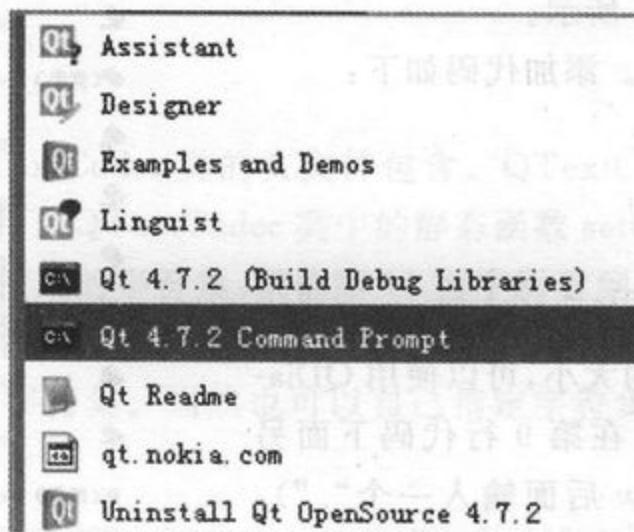


图 2-27 打开 Command Prompt

第二步,使用命令编译程序。在 Command Prompt 中已经配置好了编译环境。现在的默认路径在 C:\Qt\4.7.2 下,输入命令“cd ..”后按下回车键则跳转到上一级目录,即 C:\Qt 下。然后再输入命令“cd helloworld”跳转到新建的 helloworld 目录中。然后再输入“qmake -project”命令来生成 pro 工程文件,这时可以看到在 helloworld 目录中已经有了 helloworld.pro 文件了。下面接着输入 qmake 命令来生成用于编译的

Makefile 文件。这时在 helloworld 目录中出现了 Makefile 文件和 debug 目录与 release 目录,当然这两个目录现在是空的。接下来输入 **make** 命令来编译程序,编译完成后会在 debug 目录中出现 helloworld.exe 文件。看一下编译输出的信息,如图 2-28 所示,这里的信息与前面在 Qt Creator 中的编译信息(如图 2-22 所示)是一致的。

```

Qt 4.7.2 Command Prompt

C:\Qt\helloworld>qmake -project
C:\Qt\helloworld>qmake
C:\Qt\helloworld>make
mingw32-make[1]: Entering directory 'C:/Qt/helloworld'
g++ -c -g -fexceptions -fthreads -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT -DQT_DLL -DQT_GUI_LIB -DQT_CORE_LIB -DQT_HAVE_MMX -DQT_HAVE_3DNOW -DQT_HAVE_SSE -DQT_HAVE_MMXEXT -DQT_HAVE_SSE2 -DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN -I"../4.7.2/include/QtCore" -I"../4.7.2/include/QtGui" -I"../4.7.2/include" -I"../4.7.2/include\ActiveQt" -I"debug" -I"../4.7.2/mkspecs/win32-g+" -o debug/main.o main.cpp
g++ -enable-stdcall-fixup -fPIC -enable-auto-import -fPIC -enable-runtime-pseudo-reloc -fthreads -fPIC -fsubsystem=windows -o debug/helloworld.exe debug/main.o -L"e:/Qt/4.7.2/lib" -lmingw32 -lqtmaind -lQtGuid4 -lQtCored4
mingw32-make[1]: Leaving directory 'C:/Qt/helloworld'

```

图 2-28 命令行编译输出信息

这里对 Qt 程序的编译过程做一个简单的补充。上面使用的 qmake 是 Qt 提供的一个编译工具,它可以生成与平台无关的. pro 文件,然后利用该文件生成与平台相关的 Makefile 文件。Makefile 文件中包含了要创建的目标文件或可执行文件、创建目标文件所依赖的文件和创建每个目标文件时需要运行的命令等信息。最后使用 make 命令来完成自动编译,make 就是通过读入 Makefile 文件的内容来执行编译工作的。使用 make 命令时会为每一个源文件生成一个对应的. o 目标文件,最后将这些目标文件进行链接来生成最终的可执行文件。

第三步,运行程序。在命令行接着输入 cd debug 命令,跳转到 debug 目录下,然后再输入“helloworld.exe”,按下回车键就会运行 helloworld 程序了。

这就是 Qt 程序编辑、编译和运行的整个过程,可以看到 Qt Creator 是将工程目录管理、源代码编辑和程序编译运行等功能集合在了一起,这也就是 IDE(集成开发环境)的含义。

2.3.2 使用.ui 文件

(项目源码路径: src\02\2-3\helloworld)这里先在 Qt Creator 中往前面的 helloworld 项目里添加. ui 文件,使用. ui 文件来生成界面。然后再使用命令行来编译. ui 文件和整个项目。对应这部分内容也可以在帮助索引中查看 Using a Designer UI File in Your Application 关键字。

1. 使用.ui 界面文件

第一步,添加. ui 文件。先按照 2.3.1 小节添加 main.cpp 文件那样向工程中继续添加文件。在模板中选择 Qt 中的“Qt 设计师界面”,如图 2-29 所示。在选择界面模

板时选择 Dialog without Buttons 项，如图 2-30 所示。再单击“下一步”，将文件名称改为“hellodialog.ui”。

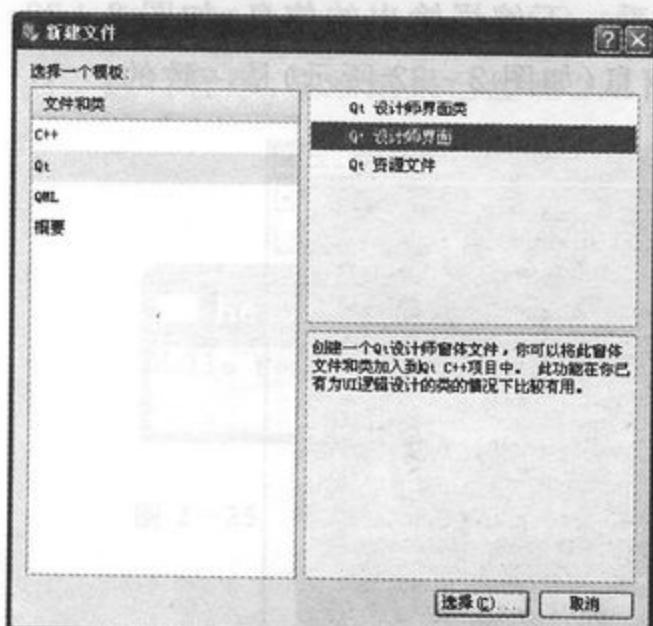


图 2-29 添加.ui 文件

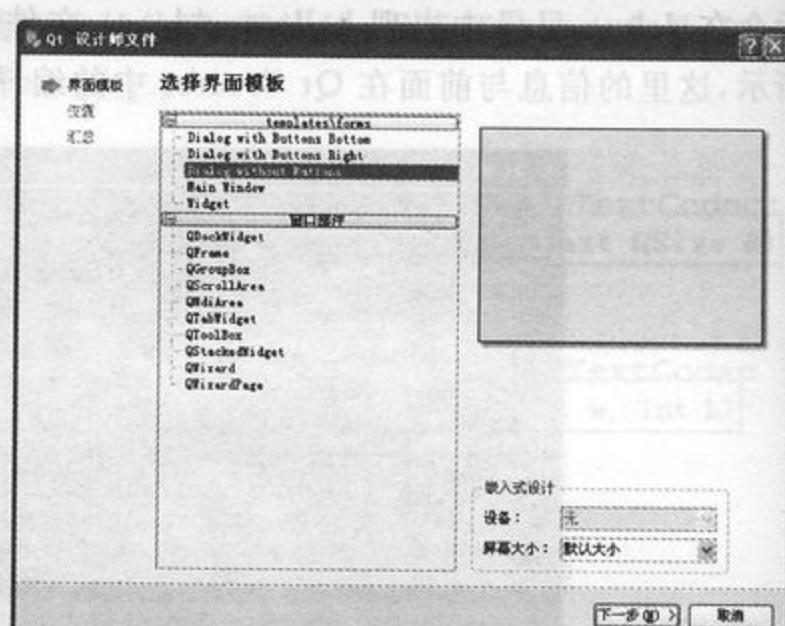


图 2-30 选择界面模板

第二步，设计界面。生成好文件后便进入了设计模式，在界面上添加一个 Label 部件，并且更改其显示内容为“Hello World! 你好 Qt!”。然后在右侧的属性栏中更改其坐标位置为“X: 120, Y: 120”。这样就与那行代码“label.move(120,120);”起到了相同的作用。然后在右上角的类列表中选择 QDialog 类对象，并且在下面的属性中更改它的对象名 objectName 为“HelloDialog”。

第三步，生成 ui 头文件。这时按下 Ctrl+S 快捷键保存修改，然后按下 Ctrl+2 快捷键回到编辑模式，那么就会看到 .ui 文件的内容了，它是一个 XML 文件，里面是界面部件的相关信息。使用 **Ctrl+Shift+B 快捷键** 或者左下角的 图标来 **编译工程**。然后单击项目文件列表上面的过滤视图 图标，去掉“隐藏生成的文件”一项，这时就可以看到由 .ui 文件生成的 **ui_hellodialog.h** 头文件了。下面来看一下这个头文件中的具体内容。

```

1  / ****
2  * * Form generated from reading UI file hellodialog.ui
3  * *
4  * * Created: Wed Mar 2 15:05:49 2011
5  * *           by: Qt User Interface Compiler version 4.7.1
6  * *
7  * * WARNING! All changes made in this file will be lost when recompiling UI file!
8  ****
9
10 #ifndef UI_HELLODIALOG_H
11 #define UI_HELLODIALOG_H
12
13 #include <QtCore/QVariant>

```

```

14 #include <QtGui/QAction>
15 #include <QtGui/QApplication>
16 #include <QtGui/QButtonGroup>
17 #include <QtGui/QDialog>
18 #include <QtGui/QHeaderView>
19 #include <QtGui.QLabel>
20
21 QT_BEGIN_NAMESPACE
22
23 class Ui_HelloDialog
24 {
25 public:
26     QLabel *label;
27
28     void setupUi(QDialog *HelloDialog)
29     {
30         if (HelloDialog->objectName().isEmpty())
31             HelloDialog->setObjectName(QString::fromUtf8("HelloDialog"));
32         HelloDialog->resize(400, 300);
33         label = new QLabel(HelloDialog);
34         label->setObjectName(QString::fromUtf8("label"));
35         label->setGeometry(QRect(120, 120, 121, 16));
36
37         retranslateUi(HelloDialog);
38
39         QMetaObject::connectSlotsByName(HelloDialog);
40     }//setupUi
41
42     void retranslateUi(QDialog *HelloDialog)
43     {
44         HelloDialog->setWindowTitle(QApplication::translate("HelloDialog", "Di-
45         alog", 0, QApplication::UnicodeUTF8));
46         label->setText(QApplication::translate("HelloDialog", "HelloWorld! \u344
47         \u275\240\345\275Qt\357\274\201", 0, QApplication::UnicodeUTF8));
48     }//retranslateUi
49
50 namespace Ui {
51     class HelloDialog: public Ui_HelloDialog {};
52 } //namespace Ui
53
54 QT_END_NAMESPACE

```

55

56 #endif //UI_HELLODIALOG_H

其中,第 1~8 行是注释信息。第 10、11 行,还有最后的第 56 行是预处理指令,能够防止对这个头文件的多重包含。第 13~19 行包含了几个类的头文件。第 21 行和最后的第 54 行是 Qt 的命名空间开始和结束宏。第 23 行定义了一个 Ui_HelloDialog 类,该类名就是前面更改的对话框类对象的名称前添加了“Ui_”字符,这是默认的名字。第 26 行定义了一个 QLabel 类对象的指针。这个就是对话框窗口添加的 Label 部件。第 28 行 **setupUi()** 函数用来生成界面,因为当时选择模板时使用的是对话框,所以现在这个函数的参数是 QDialog 类型的,后面会看到这个函数的用法。第 30 和 31 行设置了对话框的对象名称。第 32 行设置了窗口的大小,与之前在 2.3.1 小节中使用代码来设置是一样的。第 33~35 行在对话框上创建了标签对象,并设置了它的对象名称、大小和位置。第 37 行调用了 **retranslateUi()** 函数,这个函数在第 42~46 行进行了定义,实现了对窗口里的字符串进行编码转换的功能。第 39 行调用了 QMetaObject 类的 **connectSlotsByName()** 静态函数,使得窗口中的部件可以实现按对象名进行信号和槽的关联,如 void **on_button1_clicked()**,这个在第 7 章信号和槽部分还会讲到。第 50~52 行定义了命名空间 **Ui**,其中定义了一个 HelloDialog 类,继承自 **Ui_HelloDialog** 类。

可以看到,Qt 中使用 ui 文件生成了相应的头文件,其中代码的作用与 2.3.1 小节中纯代码编写程序中代码的作用是相同的。使用 Qt 设计师可以直观地看到设计的界面,而且省去了编写界面代码的过程。这里要说明的是,使用 Qt 设计师设计界面和全部自己用代码生成界面,效果是相同的。在以后的章节中主要的界面都会使用 Qt 设计师来实现,如果想自己用代码来实现,一种可以取经的方式是参考它的 ui 头文件,查看具体的代码实现。

第四步,更改 main.cpp 文件。将 main.cpp 文件中的内容更改如下:

```

1 #include "ui_helldialog.h"
2 int main(int argc, char *argv[])
3 {
4     QApplication a(argc, argv);
5     QDialog w;
6     Ui::HelloDialog ui;
7     ui.setupUi(&w);
8     w.show();
9     return a.exec();
10 }
```

第 1 行代码是头文件包含。因为在 ui_helldialog.h 中已经包含了其他类的定义,所以这里只需要包含这个文件就可以了。对于头文件的包含,使用“<>”时,系统会到默认目录(编译器及环境变量、工程文件所定义的头文件寻找目录,包括 Qt 安装的 include 目录,即“C:\Qt\4.7.2\include”)查找要包含的文件,这是标准方式;用双引号

时,系统先到用户当前目录(即项目目录)中查找要包含的文件,找不到再按标准方式查找。因为 ui_hellodialog.h 文件在我们自己的项目目录中,所以使用了双引号包含。第 6 行代码使用命名空间 Ui 中的 HelloDialog 类定义了一个 ui 对象。在第 7 行中使用了 setupUi() 函数,并将对话框类对象作为参数,这样就可以将设计好的界面应用到对象 w 所表示的对话框上了。

第五步,运行程序,便会看到与以前相同的对话框窗口了。

2. 在命令行编译 ui 文件和程序

第一步,新建工程目录。在 C:\Qt 目录中新建文件夹“helloworld_2”,然后将上面的项目文件夹 helloworld 目录下的 hellodialog.ui 和 main.cpp 两个文件复制过来。

第二步,编译 ui 文件。打开 Command Prompt,然后输入 cd C:\Qt\helloworld_2 命令进入 helloworld_2 文件夹中。再使用 uic 编译工具,从 ui 文件生成头文件。具体命令是:

```
uic -o ui_hellodialog.h hellodialog.ui
```

就像前面看到的那样,ui 文件生成的默认头文件的名称是“ui_”加 ui 文件的名称。这时在 helloworld_2 目录中已经生成了相应的头文件。

第三步,编译运行程序。依次输入如下命令:

```
qmake -project  
qmake  
make  
cd debug  
helloworld_2.exe
```

这样就完成了整个编译运行过程。可以看到 ui 文件是使用 uic 编译工具来编译的,这里的 Qt 程序通过调用相应的头文件来使用 ui 界面文件。

2.3.3 自定义 C++ 类

(项目源码路径: src\02\2-4\helloworld)这一小节首先新建空工程并且建立自己定义的一个 C++ 类,然后再使用上一小节的 ui 文件。

第一步,新建空的 Qt 项目。项目名称为 helloworld,路径为“F:\2-4”。

第二步,添加文件。向项目中添加新文件,选择 C++ 类。类名为 HelloDialog,基类为 QDialog。添加完成后再往项目中添加 main.cpp 文件。

第三步,编写源码。在 main.cpp 中添加如下代码:

```
1 #include < QApplication >  
2 #include "hellodialog.h"  
3 int main(int argc, char * argv[]){  
4     QApplication a(argc, argv);
```

```

6     HelloDialog w;
7     w.show();
8     return a.exec();
9 }

```

这里在第 2 行添加了新建的 HelloDialog 类的头文件,然后在第 6 行定义了一个该类的对象。这时运行程序,它会显示一个空白的对话框。

第四步,添加 ui 文件。将上一小节建立的 hellodialog.ui 文件复制到项目目录下(笔者这里的路径是“F:\2-4\helloworld”),然后在 Qt Creator 中的项目文件列表中的项目文件夹上右击,在弹出的菜单中选择“添加现有文件”,然后在弹出的对话框中选择 helloworld.ui 文件,将其添加到项目中。

第五步,更改 C++ 类文件。这次不在 main() 函数中使用 ui 文件,而是在新建立的 C++ 类中使用。先在头文件 hellodialog.h 中添加如下代码:

```

1 #ifndef HELLODIALOG_H
2 #define HELLODIALOG_H
3
4 #include <QDialog>
5
6 namespace Ui{
7     class HelloDialog; //我们
8 } //内容
9
10 class HelloDialog : public QDialog
11 {
12     Q_OBJECT
13 public:
14     explicit HelloDialog(QWidget *parent = 0); //新添
15
16 signals:
17
18 public slots:
19
20 private:
21     Ui::HelloDialog *ui; //新添内容
22 };
23
24 #endif //HELLODIALOG_H

```

第 1、2 和 24 行是预处理指令,避免该头文件多重包含。第 6~8 行是新添加的内容,定义了命名空间 Ui,并在其中前置声明了 HelloDialog 类,这个类就是在 ui_hellodialog.h 文件中看到的那个类。因为它与新定义的类同名,所以使用了 Ui 命名空间。而前置声明是为了加快编译速度,也可以避免在一个头文件中随意包含其他头文件而

产生错误。因为这里只使用了该类对象的指针，如第 21 行定义了该类对象的指针，这并不需要该类的完整定义，所以可以使用前置声明。这样就不用在这里添加 ui_helldialog.h 的头文件包含，而可以将其放到 helldialog.cpp 文件中进行。第 10 行是新定义的 HelloDialog 类，继承自 QDialog 类。第 12 行定义了 Q_OBJECT 宏，扩展了普通 C++ 类的功能，比如下一章要讲的信号和槽功能。必须在类定义的最开始处来定义这个宏。第 14 行是显式构造函数，参数是用来指定父窗口的，默认是没有父窗口的。第 16 行是要定义的信号，第 18 行是要定义的公共槽，这里没有定义。

提示 关于程序中涉及的一些 C++ 知识，本书中不会详细解释，有疑问可以查阅相关资料。这里推荐《C++ Primer 中文版（第 4 版）》。比如这里提到的预处理指令、前置声明和显式构造函数等内容分别在该书的第 60、374 和 394 页有相应的解释。

然后在 helldialog.cpp 文件中添加代码：

```
1 #include "helldialog.h"
2 #include "ui_helldialog.h"
3 HelloDialog::HelloDialog(QWidget *parent) :
4     QDialog(parent)
5 {
6     ui = new Ui::HelloDialog;
7     ui->setupUi(this);
8 }
```

第 2 行添加了 ui 头文件，因为在 helldialog.h 文件中只使用了前置声明，所以头文件在这里添加。第 6 行创建 Ui::HelloDialog 对象。第 7 行设置 setupUi() 函数的参数为 this，表示为现在这个类所代表的对话框创建界面。也可以将 ui 的创建代码放到构造函数首部，代码如下：

```
3 HelloDialog::HelloDialog(QWidget *parent) :
4     QDialog(parent),
5     ui(new Ui::HelloDialog)
6 {
7     ui->setupUi(this);
8 }
```

这样与前面的代码效果是相同的。现在已经写出了和 2.1 节中使用 Qt Creator 创建的 helloworld 项目中相同的文件和代码。此时，可以再运行程序。需要说明的是，这里使用 ui 文件的方法是 **单继承**，还有一种多继承的方法，就是将 HelloDialog 类同时继承自 QDialog 类和 Ui::HelloDialog 类，这样在写程序时就可以直接使用界面上的部件而不用添加 ui 指针的定义了。不过，现在 Qt Creator 默认生成的文件都是使用单继

承方式,所以这里只讲述了这种方式,这个可以在“选项”菜单的“设计师”分类中更改。也可以使用 QUiLoader 来动态加载 ui 文件,不过这种方式并不常用,这里也就不再介绍。想了解这些知识,可以参考 Qt 帮助文档的 Using a Designer UI File in Your Application 文档。

2.3.4 使用 Qt 设计师界面类

(项目源码路径: src\02\2-5\helloworld)再次新建空项目,名称仍为 helloworld,路径为“F:\2-5”。完成后向该项目中添加新文件,模板选择 Qt 中的“Qt 设计师界面类”。界面模板依然选择 Dialog without Buttons 一项,类名为 HelloDialog。完成后在设计模式往窗口上添加一个 Label,更改显示文本为“Hello World! 你好 Qt!”。然后再往项目中添加 main.cpp 文件,并更改其内容如下:

```

1 #include <QApplication>
2 #include "hellodialog.h"
3 int main(int argc, char *argv[])
4 {
5     QApplication a(argc, argv);
6     HelloDialog w;
7     w.show();
8     return a.exec();
9 }
```

现在可以运行程序,不过,还要说明一下,如果在建立这个项目时,没有关闭上一个项目,那么现在 Qt Creator 的项目文件列表中应该有两个项目,可以在这个项目的项目文件夹上右击,在弹出的菜单中选择“运行”,则本次运行该项目。也可以选择第一项“设置为启动项目”,那么就可以按下运行按钮或者使用 Ctrl+R 快捷键来运行该程序了。

这里将上一小节的内容进行了简化,因为 Qt 设计师界面类就是上一小节的 C++ 类和 ui 文件的结合,它将这两个文件一起生成了,而不用再一个一个地添加。

到这里就已经把 Qt Creator 自动生成 Qt GUI 项目进行了分解再综合,一步一步地讲解了整个项目的组成和构建过程,可以看到 Qt Creator 做了很多事情,但是读者最好也学会自己建立空项目,然后依次往里面添加各个文件,这种方式更灵活。

2.4 项目模式和项目文件介绍

2.4.1 项目模式

按下快捷键 Ctrl+5,或者单击“项目”图标,都可以进入项目模式。如果现在没有打开任何工程,项目模式是不可用的。项目模式分为“构建设置”、“运行设置”、“编辑器

设置”和“依赖关系”几个页面。如果当前打开了多个工程，那么最上面会分别列出这些工程，比如这里打开了两个 helloworld 工程，所以上面有两个 helloworld 选项，可以选择自己要设置的项目。

在构建设置页面可以设置要构建的版本，如 Debug 版或是 Release 版本，还可以设置所使用的 Qt 版本。这里有一个 Shadow build 选项，作用是将项目的源码和编译生成的文件分别存放，就像在 2.2.1 小节讲到的，helloworld 项目经编译后会生成 helloworld-build-desktop 文件夹，里面放着编译生成的所有文件。将编译输出与源代码分别存放是个很好的习惯，尤其是在使用多个 Qt 版本进行编译时更是如此。Shadow build 选项默认是选中的。如果想让源码和编译生成的文件放在一个目录下，比如前面在命令行编译时那样，那么也可以将这个选项去掉。“构建步骤”、“清除步骤”和“构建环境变量”等选项，前面都已经提及过相关内容了，如果对编译命令不是很清楚，这里保持默认就可以了，不用修改。

在运行设置中可以设置程序的运行参数和环境变量等；在编辑器设置中可以设置默认的文件编码；在依赖关系中，如果同时打开了多个项目，可以设置它们之间的依赖关系。这些选项一般都不需要更改，这里不再过多介绍。

2.4.2 项目文件

下面来看一下 2.1 节中建立的 helloworld 项目的 helloworld.pro 文件的内容：

```

1  # -----
2  #
3  # Project created by QtCreator 2011-02-28T17:47:18.
4  #
5  # -----
6
7 QT      + = core gui
8
9 TARGET = helloworld
10 TEMPLATE = app
11
12
13 SOURCES += main.cpp \
14          hellodialog.cpp
15
16 HEADERS  + = hellodialog.h
17
18 FORMS    + = hellodialog.ui
19
20 RC_FILE  + = myico.rc

```

第 1~5 行是注释信息，说明这个文件生成的时间。第 7 行表明了这个项目使用的

模块。core 模块包含了 Qt 非图形用户界面的核心功能，其他所有模块都依赖于这个模块；而 gui 模块扩展了 core 模块的图形界面功能。也就是说，如果不需要设计图形界面的程序，那么只需要 core 模块就可以了，但是如果涉及图形界面，那么就必须包含 gui 模块。其实所谓的模块，就是很多相关类的集合，比如所有与图形界面有关的类都在 gui 模块中，读者可以在 Qt 帮助中查看 QtCore Module 和 QtGui Module 关键字。第 9 行是生成的目标文件的名称，就是生成的 exe 文件的名字，默认的是工程的名字，当然也可以在这里改为别的名字。第 10 行使用 app 模板，表明这是个应用程序。第 13、16 和 18 行分别是工程中包含的源文件、头文件和界面文件。第 20 行就是添加的应用程序图标的文件。这里这些文件都使用了相对路径，因为都在项目目录中，所以只写了文件名。

这里还要提一下那个在项目文件夹中与 .pro 文件一起生成的 .pro.user 文件，它包含了本地构建信息，包含 Qt 版本和构建目录等。可以用记事本或者写字板将这个文件打开查看其内容。因为读者的系统环境都不太一样，Qt 的安装与设置也不尽相同，所以如果要将自己的源码公开，那么最好认真检查一下项目设置，以确认是否要包含这个 user 文件，一般的桌面程序是不需要包含的。而如果要打开别人的项目文件，但里面包含了 user 文件，那么这时 Qt Creator 会弹出如图 2-31 所示的提示窗口，这时应该选择“否”，然后就可以选择自己的 Qt 版本了。但是如果选择了“是”，那么打开的程序可能无法编译，这时就需要打开项目模式，然后在构建设置中选择自己的 Qt 版本。当然，最简单的方法就是在打开项目之前仔细检查一遍 .pro.user 文件，以确认是否影响我们编译运行项目，是否可使用自动建立的 .pro.user 文件。如果确认没有影响（一般桌面程序都没有影响），可以先将其删除，因为它以后还会被自动建立的。

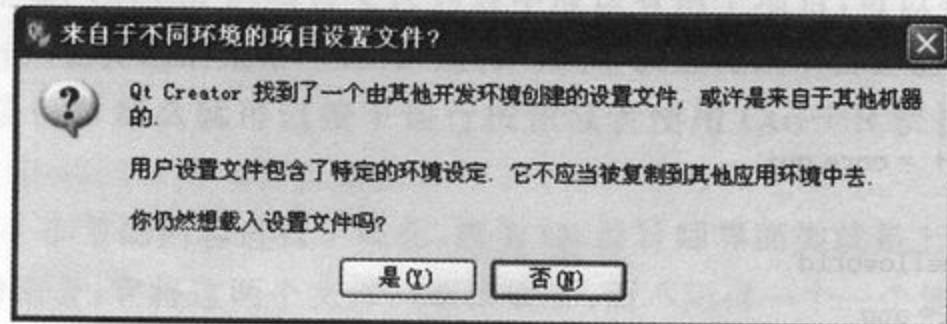


图 2-31 不同项目提示对话框

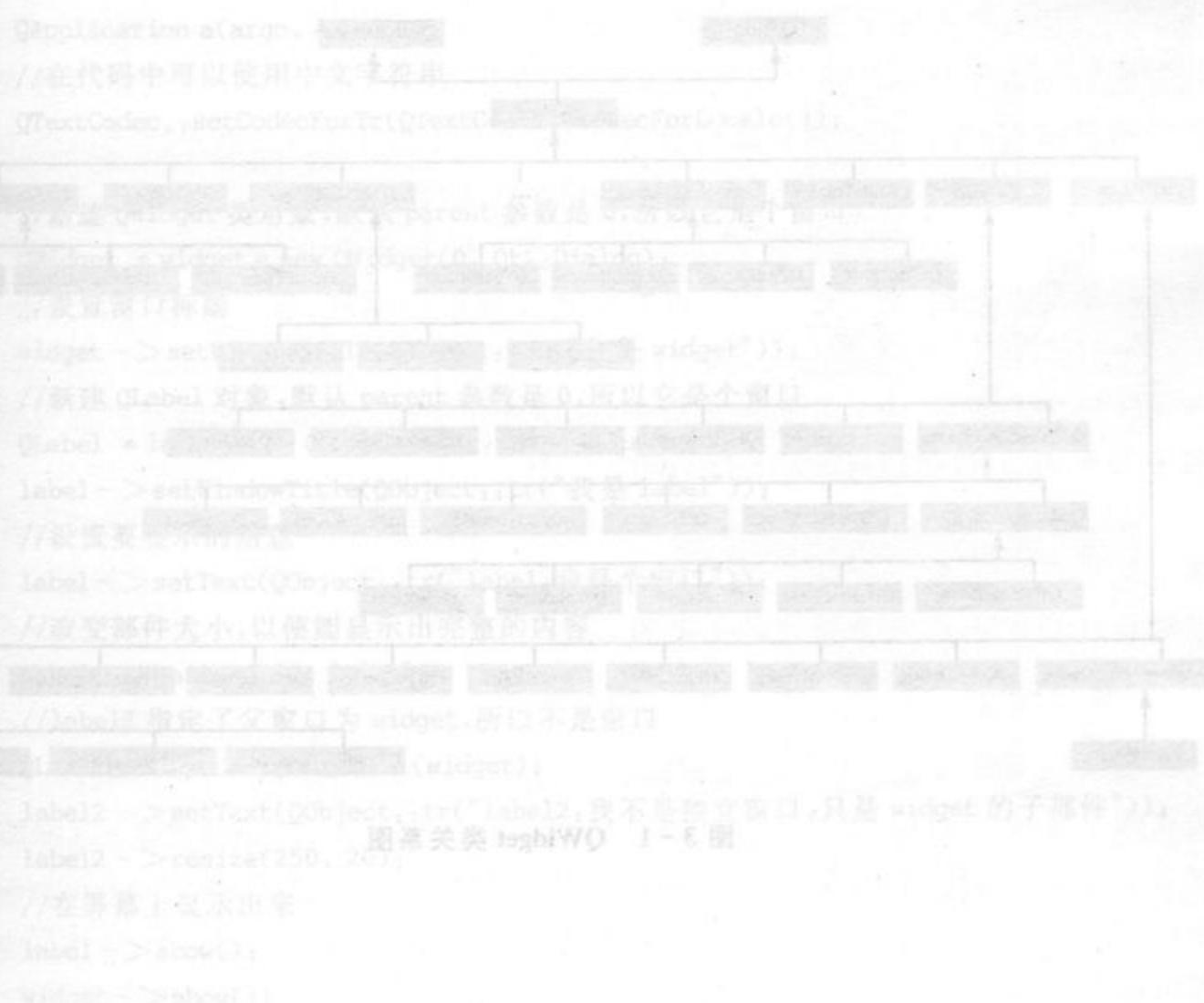
2.4.3 关于本书源码的使用

这本书的编写过程中，在每个程序开始都明确注明了项目源码的路径，因为编写代码过程中难免出现这样或那样的问题，所以最好的办法就是先下载本书的源码，然后出错了再和下载的源码对比，找出错误原因。而且，以后的章节中由于程序源码可能过长，或者有些代码重复出现，我们就会使用省略部分代码的方法，这样下载源码就更加必要了。可以到网站 www.yafeilinux.com 下载本书源码。所有的源码都放在了 src 文件夹中，然后可以根据书中的提示来找到对应的源码目录。

找到对应的源码时,建议将这个例程的整个源码目录复制出去,但路径中一定不要有中文。然后可以使用 Qt Creator 的“文件→打开文件或工程”菜单项打开目录中的 .pro 工程文件,也可以直接将 .pro 工程文件拖入 Qt Creator 中打开。当要关闭一个项目时,可以使用“文件→关闭项目”菜单项来关闭,对于已经打开的文件可以使用关闭文件菜单来关闭。

2.5 小结

这一章虽然只是讲了一个最简单的 Hello World 程序,不过讲解了 Qt 项目从建立到编译运行,再到发布的全过程。其中还讲解了整个项目的组成与编译过程。而穿插在这些内容之中的是 Qt Creator 的一些基本操作和使用流程,比如建立各种应用、添加各种文件、查看帮助、设计界面和更改属性等。这一章是后面所有章节的基础。



第3章

窗口部件

从这一章开始正式接触 Qt 的窗口部件。在第 2 章曾看到 Qt Creator 提供的默认基类只有 QMainWindow、QWidget 和 QDialog 这 3 种。是的,这 3 种窗体也是以后用的最多的,QMainWindow 是带有菜单栏和工具栏的主窗口类,QDialog 是各种对话框的基类,而它们全部继承自 QWidget。不仅如此,其实所有的窗口部件都继承自 QWidget,如图 3-1 所示。这一章会讲解 QWidget、QDialog 和其他一些常用部件类,而 QMainWindow 将在第 5 章讲解。

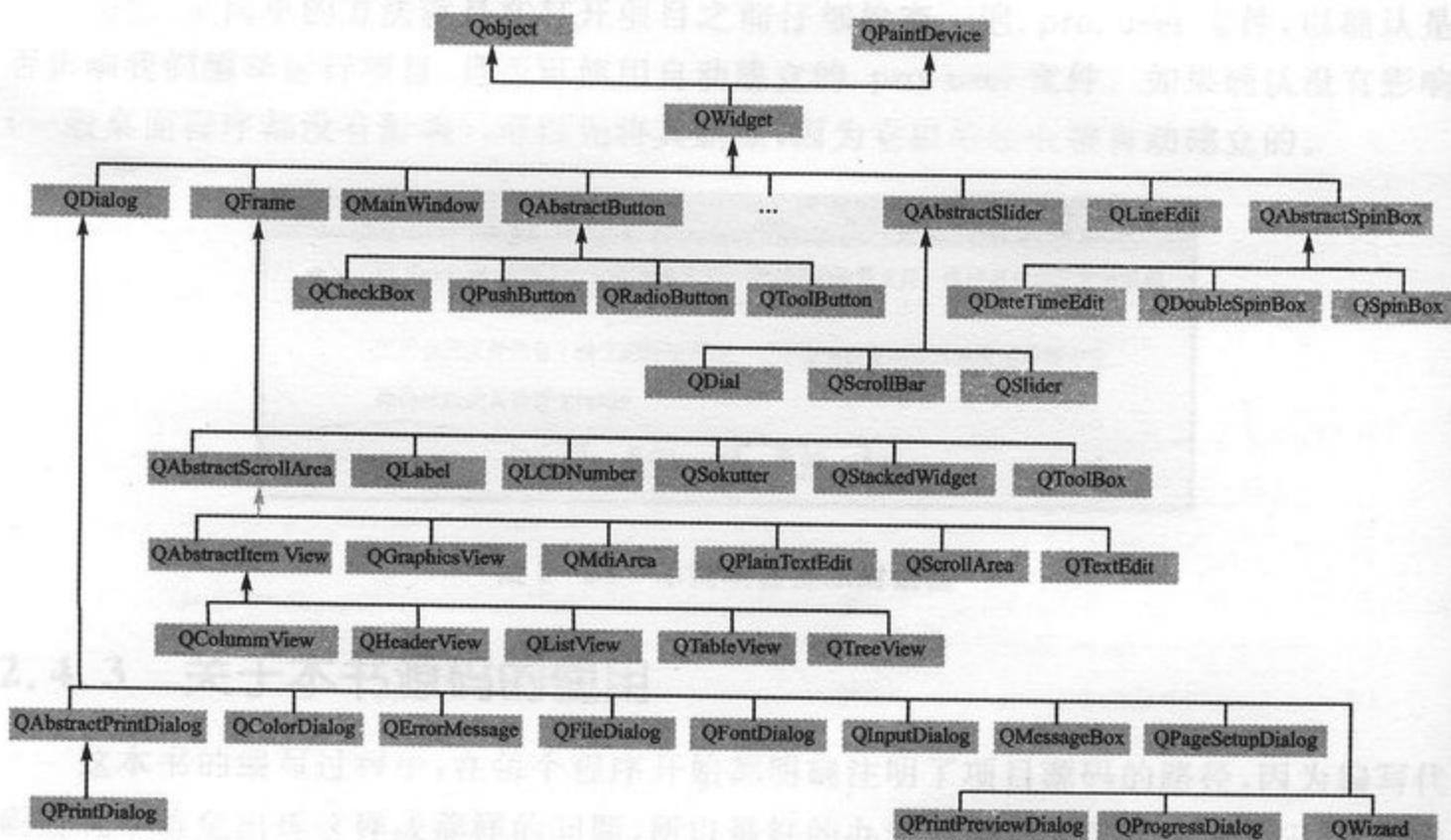


图 3-1 QWidget 类关系图

3.1 基础窗口部件 QWidget

QWidget 类是所有用户界面对象的基类,被称为基础窗口部件。在图 3-1 中可以看到, QWidget 继承自 QObject 类和 QPaintDevice 类,其中 QObject 类是所有支持 Qt 对象模型(Qt Object Model)的基类,QPaintDevice 类是所有可以绘制的对象的基类。这一节先讲解 Qt 窗口部件的概念和窗口类型,然后再讲解 Qt 窗口的几何布局,最后讲解 Qt 程序调试方面的内容。

3.1.1 窗口、子部件以及窗口类型

1. 窗口与子部件

先来看一个例子(项目源码路径:src\03\3-1\myWidget1)。打开 Qt Creator,新建空的 Qt 项目,项目名称为 myWidget1。然后往项目中添加 C++ 源文件“main.cpp”,并添加以下代码:

```
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    //在代码中可以使用中文字符串
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    //新建 QWidget 类对象,默认 parent 参数是 0,所以它是个窗口
    QWidget *widget = new QWidget(0, Qt::Dialog);
    //设置窗口标题
    widget->setWindowTitle(QObject::tr("我是 widget"));

    //新建 QLabel 对象,默认 parent 参数是 0,所以它是个窗口
    QLabel *label = new QLabel(0, Qt::SplashScreen);
    label->setWindowTitle(QObject::tr("我是 label"));

    //设置要显示的信息
    label->setText(QObject::tr("label:我是个窗口"));
    //改变部件大小,以便能显示出完整的内容
    label->resize(180, 20);
    //label2 指定了父窗口为 widget,所以不是窗口
    QLabel *label2 = new QLabel(widget);
    label2->setText(QObject::tr("label2:我不是独立窗口,只是 widget 的子部件"));
    label2->resize(250, 20);
    //在屏幕上显示出来
    label->show();
    widget->show();
}
```

```

int ret = a.exec();
delete label;
delete widget;
return ret;
}

```

这里包含了头文件 `#include <QtGui>`, 因为下面所有要用到的类, 如 `QApplication`、`QWidget` 等都包含在 `QtGui` 模块中。需要说明, 一般的原则是要包含尽可能少的头文件。程序中定义了一个 `QWidget` 类对象的指针 `widget` 和两个 `QLabel` 对象指针 `label` 与 `label2`, 其中 `label` 没有父窗口, 而 `label2` 在 `widget` 中, `widget` 是其父窗口。(注意: 这里使用 `new` 操作符为 `label2` 分配了空间, 但是并没有使用 `delete` 进行释放, 这是因为在 Qt 中销毁父对象的时候会自动销毁子对象, 这里 `label2` 指定了 `parent` 为 `widget`, 所以在 `delete widget` 时会自动销毁作为 `widget` 子对象的 `label2`。在第 7 章对象树部分会对这个问题进行详细讲解。)然后运行程序, 效果如图 3-2 所示。



图 3-2 两个窗口运行效果

窗口部件(Widget)这里简称部件, 是 Qt 中建立用户界面的主要元素。像主窗口、对话框、标签、还有以后要介绍到的按钮、文本输入框等都是窗口部件。这些部件可以接收用户输入, 显示数据和状态信息, 并且在屏幕上绘制自己;有些也可以作为一个容器, 来放置其他部件。Qt 中把没有嵌入到其他部件中的部件称为窗口, 一般窗口都有边框和标题栏, 就像程序中的 `widget` 和 `label` 一样。 `QMainWindow` 和大量的 `QDialog` 子类是最一般的窗口类型。窗口就是没有父部件的部件, 所以又称为顶级部件(top-level widget)。与其相对的是非窗口部件, 又称为子部件(child widget)。在 Qt 中大部分部件用作子部件, 嵌入在别的窗口中, 例如程序中的 `label2`。这部分内容可以查看关键字 `QWidget` 和 `Widgets and Layout`。

`QWidget` 提供了绘制自己和处理用户输入事件的基本功能, Qt 提供的所有界面元素不是 `QWidget` 的子类就是与 `QWidget` 的子类相关联。要设计自己的窗口部件, 可以继承自 `QWidget` 或者是它的子类等。

2. 窗口类型

前面讲到窗口一般都有边框和标题栏, 其实这也不是必需的。 `QWidget` 的构造函数有两个参数: “ `QWidget * parent=0`”和“ `Qt::WindowFlags f=0`”, 前面的 `parent` 就是指父窗口部件, 默认值为 0, 表明没有父窗口;而后面的 `f` 参数是 `Qt::WindowFlags` 类型的, 是一个枚举类型, 分为窗口类型(`WindowType`)和窗口标志(`WindowFlags`)。前者可以定义窗口的类型, 比如 `f=0` 表明使用了 `Qt::Widget` 一项, 这是 `QWidget` 的默

认类型,这种类型的部件如果有父窗口,那么它就是子部件,否则就是独立的窗口。其中还有很多其他类型,下面使用其中的 Qt::Dialog 和 Qt::SplashScreen 更改程序中的新建对象的那两行代码:

```
QWidget * widget = new QWidget(0, Qt::Dialog);
QLabel * label = new QLabel(0, Qt::SplashScreen);
```

这时运行程序,效果如图 3-3 所示。

可以看到,更改窗口类型后,窗口的样式发生了改变,一个是对话框类型,一个是欢迎窗口类型。而窗口标志的作用是更改窗口的标题栏和边框,而且可以和窗口类型进行位或操作。下面再次更改那两行代码:

```
QWidget * widget = new QWidget(0, Qt::Dialog | Qt::FramelessWindowHint);
QLabel * label = new QLabel(0, Qt::SplashScreen | Qt::WindowStaysOnTopHint);
```

Qt::FramelessWindowHint 用来产生一个没有边框的窗口,而 Qt::WindowStaysOnTopHint 用来使该窗口停留在所有其他窗口上面,如图 3-4 所示。虽然单击了 Qt Creator,但是只有 widget 窗口隐藏到了后面,而 label 窗口依然在最上面。这里还要提示一点,现在两个窗口都没有了标题栏,那么怎么关闭它们呢? 其实可以在应用程序输出栏中按下那个红色的按钮,来强行关闭程序,如图 3-5 所示。

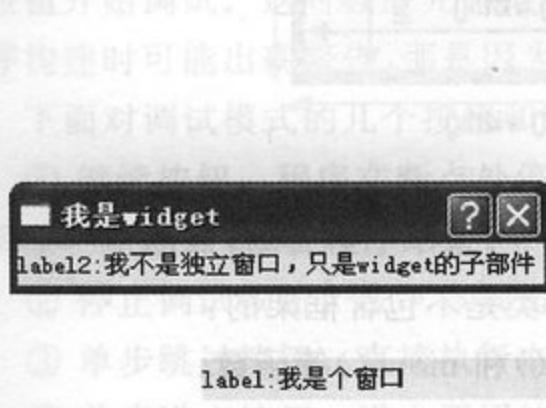


图 3-3 更改窗口类型后的运行效果

这里只举了两个简单的例子来演示 `f` 参数的使用,

如果还想看其他几个值的效果,可以自己更改程序。在 Qt 的演示程序中也有一个 Window Flags 程序演示了所有窗口类型和标志。可以在 Qt Examples and Demos 工具中的 Widgets 分类中找到,或者直接在 Qt Creator 的欢迎模式中打开。QWidget 中还有一个 `setWindowState()` 函数可以用来设置窗口的状态,比如最大化 Qt::WindowMaximized、最小化 Qt::WindowMinimized 和全屏显示 Qt::WindowFullScreen 等,这个函数的默认设置为 Qt::WindowNoState。



图 3-4 更改窗口标志后的运行效果



图 3-5 关闭运行程序的按钮

3.1.2 窗口几何布局

对于一个窗口，往往要设置它的大小和运行时出现的位置，这就是本小节要说的窗口几何布局。在上面的例子中应该已经看到了，widget 默认的大小就是它所包含的子部件 label2 的大小，而 widget 和 label 出现时在窗口上的位置也是不确定的。对于窗口的大小和位置，根据是否包含边框和标题栏两种情况，要用不同的函数来获取。可以在帮助索引中查看 Window and Dialog Widgets 关键字，文档中显示了窗口的几何布局图，如图 3-6 所示。

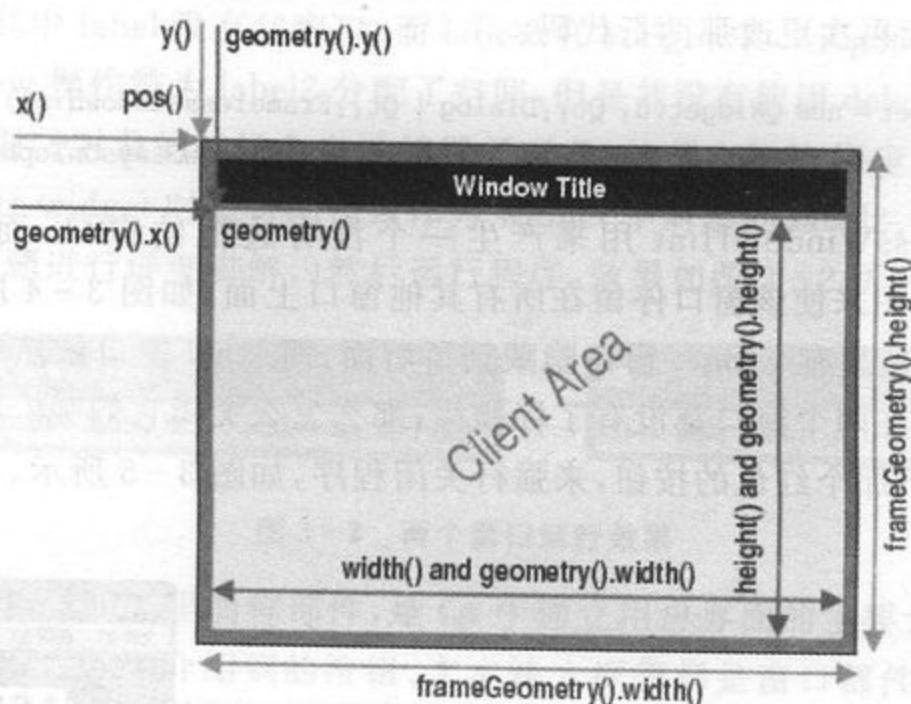


图 3-6 窗口几何布局

这里的函数分为两类，一类是包含框架的，另一类是不包含框架的：

- 包含框架：`x()`、`y()`、`frameGeometry()`、`pos()` 和 `move()` 等函数；
- 不包含框架：`geometry()`、`width()`、`height()`、`rect()` 和 `size()` 等函数。

3.1.3 程序调试

下面会在讲解窗口几何布局的几个函数的同时，讲解程序调试方面的内容。这部分内容可以在帮助索引中查看 Interacting with the Debugger 和 Debugging the Example Application 关键字。（项目源码路径：src\03\3-2\myWidget2）

1. 设置断点

在前面程序的基础上进行更改，将主函数内容更改如下：

```
#include <QApplication>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.show();
    return app.exec();
}
```

```

    {
        Application a(argc, argv);

        QWidget widget;
        int x = widget.x();
        int y = widget.y();
        QRect geometry = widget.geometry();
        QRect frame = widget.frameGeometry();

        return a.exec();
    }
}

```

开始调试程序之前可以先看一下这些函数的介绍。将光标定位到函数上，按下 F1 键，打开函数的帮助文档。可以看到 `x()`、`y()` 分别返回部件的位置坐标的 `x`、`y` 值，它们的默认值为 0。而 `geometry()` 和 `frameGeometry()` 函数分别返回没有边框和包含边框的窗口框架矩形的值，其返回值是 `QRect` 类型的，就是一个矩形，它的形式是(位置坐标, 大小信息)，也就是(`x, y, 宽, 高`)。

下面在“`int x=widget.x();`”一行代码的标号前面单击来设置断点。只要在那个断点上再单击一下就可以取消断点了。设置好断点后便可以按下 F5 或者左下角的调试按钮开始调试。这时程序先进行构建再进入调试模式，这个过程可能需要一些时间。程序构建时可能出现警告，那是因为我们定义了变量却没有使用造成的，不用管它。

下面对调试模式的几个按钮和窗口进行简单介绍：

① 继续按钮。程序在断点处停了下来，按下继续按钮后，程序便会正常运行，直到遇到下一个断点，或者程序结束。

② 停止调试按钮。按下该按钮后结束调试。

③ 单步跳过按钮。直接执行本行代码，然后指向下一行代码。

④ 单步进入按钮。进入调用的函数内部。

⑤ 单步跳出按钮。当进入函数内部时，跳出该函数，一般与单步进入配合使用。

⑥ 显示源码对应的汇编指令，并可以单步调试。

⑦ 堆栈视图。这里显示了从程序开始到断点处所有嵌套调用的函数所在的源文件名和行号。

⑧ 其他视图。这里有局部变量和监视器视图，用来显示局部变量和它们的类型及数值；断点视图用来显示所有的断点，以及添加或者删除断点；线程视图用来显示所有的线程和现在所在的线程；快照视图用来管理快照，快照可以保存当前的调试状态。

2. 单步调试

先单击“单步进入”按钮，或者按下 F11，则程序跳转到 `QWidget` 类的 `x()` 函数的源码处，下面直接单击“单步跳出”按钮回到原来的断点处。然后一直单击“单步跳过”按钮，单步执行程序并查看局部变量和监视器视图中相应变量值的变化情况。等执行到

最后一行代码“`return a.exec();`”时，单击“停止调试”按钮结束调试。补充说明一下，程序调试过程中可以进入到 Qt 类的源码中，还有一个很简单的方法也可以实现这个功能，就是在编辑器中将鼠标光标定位到一个类名或者函数上，然后按下 F2 键，或者右击，选择“跟踪光标位置的符号”，则编辑器就会跳转到其源码处。

从变量监视器中可以看到 `x`、`y`、`geometry` 和 `frame` 这 4 个变量初始值都是一个随机未知数。调试完成后，`x`、`y` 的值均为 0，这是默认值。而 `geometry` 的值为 `640x480+0+0`，`frame` 的值为 `639x479+0+0`。现在对这些值还不是很清楚，不过，为什么 `x`、`y` 的值会是 0 呢？我们可能会想到，应该是窗口没有显示的原因，那么就更改代码，让窗口先显示出来再看这些值。在“`QWidget widget;`”一行代码后添加一行代码：

```
widget.show();
```

再次调试程序，这时会发现窗口只显示了一个标题栏，先不管它，继续在 Qt Creator 中单击“单步跳过”按钮。将程序运行到最后一行代码“`return a.exec();`”时再次单击“单步跳过”按钮后程序窗口终于显示出来了。这是因为只有程序进入主事件循环后才能接收事件，而 `show()` 函数会触发显示事件，所以只有在完成 `a.exec()` 函数调用进入消息循环后才能正常显示。这次看到几个变量的值都有了变化，但是仍不清楚这些值的含义。

因为使用调试器调试要等待一段时间，而且步骤麻烦，对于初学者来说，如果按错了按钮，还很容易出错。下面介绍一个更简单的调试方法。

3. 使用 `qDebug()` 函数

程序调试过程中很常用的是 `qDebug()` 函数，它可以将调试信息直接输出到控制台，当然，Qt Creator 中是输出到应用程序输出栏。更改上面的程序：

```
#include <QApplication>
#include <QWidget>
#include <QDebug>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QWidget widget;
    widget.resize(400, 300);           //设置窗口大小
    widget.move(200, 100);            //设置窗口位置
    widget.show();
    int x = widget.x();
    qDebug("x: %d", x);             //输出 x 的值
    int y = widget.y();
    qDebug("y: %d", y);
    QRect geometry = widget.geometry();
```

```

QRect frame = widget.frameGeometry();
qDebug() << "geometry: " << geometry << "frame: " << frame;
return a.exec();
}

```

要使用 qDebug() 函数,就要添加 #include <QDebug>头文件。这里使用了两种输出方法,一种是直接将字符串当作参数传给 qDebug() 函数,例如上面使用这种方法输出 x 和 y 的值。另一种方法是使用输出流的方式一次输出多个值,它们的类型可以不同,例如程序中输出 geometry 和 frame 的值。需要说明的是,如果只使用第一种方法,那么是不需要添加<QDebug>头文件的;而第二种方法就必须添加这个头文件。第一种方法很麻烦,所以经常使用的是第二种方法。程序中还添加了设置窗口大小和位置的代码。下面运行程序,在应用程序输出窗口可以看到输出信息,如图 3-7 所示。

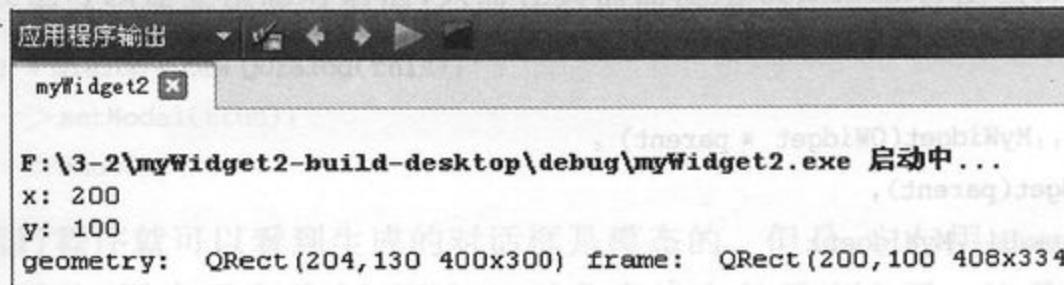


图 3-7 程序输出信息

从输出信息中可以清楚地看到几个函数的含义。其实使用 qDebug() 函数的第二种方法时还可以让输出自动换行,下面来看一下其他几个函数的用法。在“return a.exec();”一行代码前添加如下代码:

```

qDebug() << "pos:" << widget.pos() << endl << "rect:" << widget.rect()
    << endl << "size:" << widget.size() << endl << "width:"
    << widget.width() << endl << "height:" << widget.height();

```

“endl”是起换行作用的。根据程序的输出结果,可以很明了地看到这些函数的作用。其中, pos() 函数返回窗口的位置,是一个坐标值,上面的 x()、y() 函数返回的就是它的 x、y 坐标值; rect() 函数返回不包含边框的窗口内部矩形,在窗口内部,左上角是 (0,0) 点; size() 函数返回不包含边框的窗口大小信息; width() 和 height() 函数分别返回窗口内部的宽和高。从数据可以看到,前面使用的调整窗口大小的 resize() 函数是设置的不包含边框的窗口大小。

到这里, QWidget 的内容就告一段落,其他特性会在后面的章节中涉及。

3.2 对话框 QDialog

这一节主要讲述对话框类,先从对话框的介绍讲起,然后讲述两种不同类型的对话框,再讲解一个有多个窗口组成并且窗口间可以相互切换的程序,最后介绍 Qt 提供的

几个标准对话框。其中还会涉及信号和槽的初步知识。对应本节的内容，可以在帮助索引中查看 QDialog 和 Dialog Windows 关键字。

3.2.1 模态和非模态对话框

QDialog 类是所有对话框窗口类的基类。对话框窗口是一个经常用来完成一个短小任务或者和用户进行简单交互的顶层窗口。按照运行对话框时是否还可以和该程序的其他窗口进行交互，对话框常被分为两类，模态的(modal)和非模态的(modeless)。关于这两个概念，下面先看一个例子。

(项目源码路径：src\03\3-3\myDialog1)新建 Qt Gui 应用，项目名称为 myDialog1，类名为 MyWidget，基类选择 QWidget。然后在 mywidget.cpp 文件中添加代码：

```
#include "mywidget.h"
#include "ui_mywidget.h"
#include <QDialog>

MyWidget::MyWidget(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::MyWidget)
{
    ui->setupUi(this);
    QDialog dialog(this);
    dialog.show();
}
```

这里在 MyWidget 类的构造函数中定义了一个 QDialog 类对象，还指定了 dialog 的父窗口为 MyWidget 类对象，就是那个 this 参数，最后调用它的 show() 函数让其显示。但是这时运行程序会发现一个窗口一闪而过，然后就只显示 MyWidget 类对象的窗口了。为什么会这样呢？因为在在一个函数中定义的变量，等这个函数执行结束后，它就会自动释放。也就是说，这里的 dialog 对象只在这个构造函数中有用，等这个构造函数执行完了，dialog 也就消失了。为了不让 dialog 消失，可以将 QDialog 对象的创建代码更改如下：

```
QDialog * dialog = new QDialog(this);
dialog->show();
```

这次使用了 QDialog 对象的指针，并使用 new 运算符开辟了内存空间，这时再运行程序就已经可以正常显示了。这里要说明的是，我们说定义一个对象是指“QDialog dialog;”这样的方式，而像“QDialog * dialog;”不能称为定义了一个对象，而应该说成定义了一个指向 QDialog 类对象的指针变量。后面我们也会把“QDialog * dialog;”说成是定义了一个 QDialog 对象。再补充一点，这里为 dialog 对象指明了父窗口，所以就没有必要使用 delete 来释放该对象了。

其实,不用指针也可以让对话框显示出来,可以将代码更改如下:

```
QDialog dialog(this);
dialog.exec();
```

这时运行程序就会发现对话框弹出来了,但是 MyWidget 类对象的窗口并没有出来;关闭这个对话框后,MyWidget 类对象的窗口才弹出来。这个与上面的那个对话框的效果不同,称它为模态对话框,而上面那种对话框称为非模态对话框。

模态对话框就是没有关闭它之前,不能再与同一个应用程序的其他窗口进行交互,比如新建项目时弹出的对话框。而对于非模态对话框,既可以与它交互,也可以与同一程序中的其他窗口交互,例如 Microsoft Word 中的查找替换对话框。就像前面看到的,要想使一个对话框成为模态对话框,只需要调用它的 exec() 函数,而要使其成为非模态对话框,可以使用 new 操作来创建,然后使用 show() 函数来显示。其实使用 show() 函数也可以建立模态对话框,只需在其前面使用 setModal() 函数即可。例如:

```
QDialog * dialog = new QDialog(this);
dialog->setModal(true);
dialog->show();
```

现在运行程序就可以看到生成的对话框是模态的。但是,它与用 exec() 函数时的效果是不一样的,因为现在的 MyWidget 对象窗口也显示出来了。这是因为调用完 show() 函数后会立即将控制权交给调用者,那么程序可以继续往下执行。而调用 exec() 函数却不同,它只有当对话框被关闭时才会返回。与 setModal() 函数相似的还有一个 setWindowModality() 函数,它有一个参数来设置模态对话框要阻塞的窗口类型,可以是 Qt::NonModal(不阻塞任何窗口,就是非模态)、Qt::WindowModal(阻塞它的父窗口和所有祖先窗口以及它们的子窗口)、Qt::ApplicationModal(阻塞整个应用程序的所有窗口)三者之一。而 setModal() 函数默认设置的是 Qt::ApplicationModal。

3.2.2 多窗口切换

下面来讲述一个由多个窗口组成且各窗口之间可以切换的程序,首先讲述一下信号和槽的初步知识。

1. 认识信号和槽

在 Qt 中使用信号和槽机制来完成对象之间的协同操作。简单来说,信号和槽都是函数,比如单击窗口上的一个按钮想要弹出一个对话框,那么就可以将这个按钮的单击信号和定义的槽关联起来,在这个槽中可以创建一个对话框并且显示。这样单击这个按钮时就会发射信号,进而执行槽来显示一个对话框。下面来看一个例子。

(项目源码路径: src\03\3-4\myDialog1)还在前面的程序中进行更改。双击 mywidget.ui 文件,在设计模式中往界面添加一个 Label 和 Push Button,在属性栏中将 Push Button 的 objectName 改为 showChildButton,然后更改 Label 的显示文本为“我是主界面!”,更改按钮的显示文本为“显示子窗口”。然后回到编辑模式中,打开 my-

widget.h 文件，并且在 MyWidget 类声明的最后写上槽声明：

```
public slots:
    void showChildDialog();
```

这里自定义了一个槽，槽必须声明为 slots，这里使用了 public slots，表明这个槽可以在类外被调用。现在要到源文件中编写这个槽的实现代码，这里 Qt Creator 设计了一个快速添加声明的方法：单击 showChildDialog() 槽，然后同时按下 Alt+Enter 键，则弹出如图 3-8 所示的“在 mywidget.cpp 添加声明”选项，再次按回车键 Enter，编辑器便会转到 mywidget.cpp 文件中，并且已经创建了 showChildDialog() 槽的定义，只需要在其中添加代码即可。这种方法也适用于先在源文件中添加定义，然后自动在头文件中添加声明的情况。

```
private:
    Ui::MyWidget *ui;
public slots:
    void showChildDialog();
```

在mywidget.cpp添加声明

```
#endif // MYWIDGET_H
```

图 3-8 自动添加声明

在 mywidget.cpp 文件中将 showChildDialog() 槽的实现更改如下：

```
void MyWidget::showChildDialog()
{
    QDialog * dialog = new QDialog(this);
    dialog->show();
}
```

在这里新建了对话框，并让其显示。然后再更改 MyWidget 类的构造函数如下：

```
MyWidget::MyWidget(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::MyWidget)
{
    ui->setupUi(this);
    connect(ui->showChildButton, SIGNAL(clicked()),
            this, SLOT(showChildDialog()));
```

这里使用了 connect() 函数将按钮的单击信号 clicked() 与新建的槽进行关联。clicked() 信号在 QPushButton 类中进行了定义，而 connect() 是 QObject 类中的函数，因为类继承自 QObject，所以可以直接使用。这个函数中的 4 个参数分别是：发送信号的对象、发送的信号、接收信号的对象和要执行的槽，而信号和槽要分别使用 SIGNAL() 和 SLOT() 宏括起来。这时运行程序，然后单击主界面上的按钮就会弹出一个对

话框。

其实,对于信号和槽的关联还有一种方法,叫做自动关联。前面那种方法叫做手动关联。自动关联就是将关联函数整合到槽命名中,但必须使用 Qt 部件已经提供的信号。比如前面的槽可以重命名为 `on_showChildButton_clicked()`,就是由字符“on”和发射信号的部件对象名还有信号名组成。这样就可以去掉那个 `connect()` 关联函数了,具体做法在下面介绍。

(项目源码路径: `src\03\3-5\myDialog1`)在 `MyWidget` 类的构造函数中删去 `connect()` 函数调用,然后更改 `showChildDialog()` 槽的名字,在 Qt Creator 中提供了一个快捷方式来更改所有该函数出现的地方,不再需要逐一更改函数名。先在 `showChildDialog` 上右击,在弹出的菜单中选择“Refactor→重命名光标位置的符号”,或者直接使用 `Ctrl+Shift+R` 快捷键,然后在出现的替换栏中输入 `on_showChildButton_clicked`,再单击“替换”就可以了。这时源文件和头文件中相应的函数名都进行了更改。现在运行程序,和前面的效果是一样的。

对于这两种关联方式,后一种只适用于 Qt 部件已经定义的信号,但是它很简便。后面还会看到,用 Qt 设计器直接生成的槽就是使用这种方式。不过,对于不是在 Qt 设计器中往界面上添加的部件,我们就要在调用 `setupUi()` 函数前定义该部件,而且还要使用 `setObjectName()` 函数指定部件的对象名,这样才可以使用自动关联。而如果是自己写的信号,就只有使用手动关联方式了。关于信号和槽,在第 7 章还会深入讲解。

2. 自定义对话框

关于自定义对话框,其实前面的第一个 `helloworld` 程序中就已经实现了。现在再定义一个自己的对话框,给它添加按钮,并且在 Qt 设计器中设计信号和槽,然后实现与主界面的切换。(项目源码路径: `src\03\3-6\myDialog1`)

第一步,添加自定义对话框类。依然在前面的项目中进行更改。首先向该项目中添加 Qt 设计器界面类。界面模板选择 `Dialog without Buttons`,类名改为 `MyDialog`。然后在设计模式向窗口添加两个 `Push Button`,并且分别更改其显示文本为“进入主界面”和“退出程序”。

第二步,设计信号和槽。这里使用设计器来实现“退出程序”按钮的信号和槽的关联。单击设计器上方的“编辑信号/槽”图标,或者按下快捷键 F4,便进入了部件的信号和槽的编辑模式。在“退出程序”按钮上按住鼠标左键,然后拖动到窗口界面上,这时松开鼠标左键。在弹出的配置连接对话框中选择“显示从 QWidget 继承的信号和槽”选项,然后在左边的 `QPushButton` 栏中选择信号 `clicked()`,在右边的 `QDialog` 栏中选择对应的槽 `close()`,完成后单击“确定”,如图 3-9 所示(还可以单击“编辑”按钮添加自定义的槽,不过这还需要在 `MyDialog` 类中实现该槽)。这时“退出程序”按钮的单击信号就和对话框的关闭操作槽进行了关联。要想取消这个关联,只需在信号和槽编辑模式中选择这个关联,当它变为红色时,按下 Delete 键或者右击,选择“删除”。也可以

在下面的信号和槽编辑器中看到设置好的关联，当然，直接在这里建立关联也是可以的，它与鼠标选择部件进行关联是等效的。设置好关联后按下 F3 键，或者单击“编辑控件”图标，回到部件编辑模式。关于设计器中信号和槽的详细使用，可以在帮助索引中查看 Qt Designer's Signals and Slots Editing Mode 关键字。

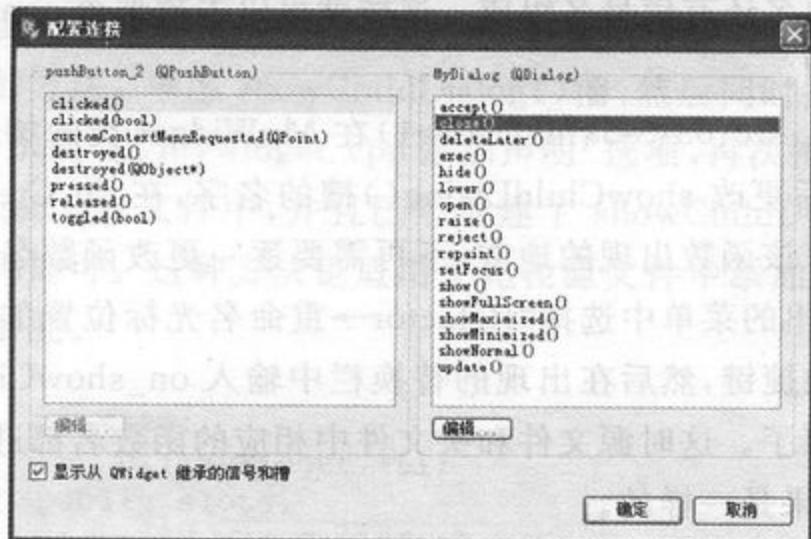


图 3-9 选择信号和槽

现在设置“进入主界面”按钮的信号和槽的关联。在该按钮上右击，在弹出的菜单上选择“转到槽”，然后在弹出的对话框中选择 clicked() 信号，并单击“确定”，便会进入代码编辑模式，并且定位到自动生成的 on_pushButton_clicked() 槽中。在其中添加代码：

```
void MyDialog::on_pushButton_clicked()
{
    accept();
}
```

这个 accept() 函数是 QDialog 类中的一个槽。一个使用 exec() 函数实现的模态对话框执行了这个槽就会隐藏这个模态对话框，并返回 QDialog::Accepted 值，这里就是要使用这个值来判断是哪个按钮被按下了。与其对应的还有一个 reject() 槽，它可以返回一个 QDialog::Rejected 值。其实，前面的“退出程序”按钮也可以关联这个槽。

前面讲述了两种关联信号和槽的方法，第一种是直接在设计器中进行，这个更适合在设计器中的部件间进行。第二种方法是在设计器中直接进入相关信号的槽，这个与手写函数是一样的，它用的就是自动关联，这样也会自动添加该槽的声明，我们只需更改其实现内容就可以了。以后的章节中，如果在设计器中添加的部件要使用信号和槽，那么都会使用第二种方法。

3. 在主界面中使用自定义的对话框

更改 main.cpp 函数内容如下：

```
#include <QtGui/QApplication>
#include "mywidget.h"
```

```

#include "mydialog.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MyWidget w;
    MyDialog dialog; //新建 MyDialog 类对象
    if(dialog.exec() == QDialog::Accepted){ //判断 dialog 执行结果
        w.show(); //如果是按下了“进入主界面”按钮，则显示主界面
        return a.exec(); //程序正常运行
    }
    else return 0; //否则，退出程序
}

```

在主函数中建立了 MyDialog 对象，然后判断其 exec() 函数的返回值，如果是按下了“进入主界面”按钮，那么返回值应该是 QDialog::Accepted，显示主界面，并且正常执行程序；如果不是，则直接退出程序。

现在运行程序可以发现已经实现了从登录对话框到主界面，再从主界面显示一个对话框的应用了。再来实现可以从主界面重新进入登录界面的功能。双击 mywidget.ui 文件，在设计模式中再向界面上添加两个 Push Button，分别更改它们的显示文本为“重新登录”和“退出”。然后使用信号和槽模式，将“退出”按钮的 clicked() 信号和 MyWidget 界面的 close() 槽关联。完成后再进入“重新登录”按钮的 clicked() 信号的槽。并更改如下：

```

void MyWidget::on_pushButton_clicked()
{
    //先关闭主界面，其实它是隐藏起来了，并没有真正退出。然后新建 MyDialog 对象
    close();
    MyDialog dlg;
    //如果按下了“进入主窗口”按钮，则再次显示主界面
    //否则，因为现在已经没有显示的界面了，所以程序将退出
    if(dlg.exec() == QDialog::Accepted) show();
}

```

需要说明的是那个 close() 槽，它不一定使程序退出，只有当只剩下最后一个主界面（就是没有父窗口的界面）时调用 close() 槽，程序才会退出；其他情况下界面只是隐藏起来了，并没有被销毁。

3.2.3 标准对话框

Qt 提供了一些常用的对话框类型，全部继承自 QDialog 类，并增加了自己的特色功能，比如获取颜色、显示特定信息等。下面将简单讲解这些对话框。也可以在帮助索引中查看 Standard Dialogs 关键字，也可以直接查看相关类的类名。

(项目源码路径: src\03\3-7\myDialog2)这里新建一个项目,首先新建 Qt Gui 应用,项目名称为 myDialog2,类名改为 MyWidget,基类选择 QWidget。双击 mywidget.ui 文件进入设计模式,在界面上添加一些按钮,如图 3-10 所示。然后回到编辑模式,在 main.cpp 中添加支持中文的代码。

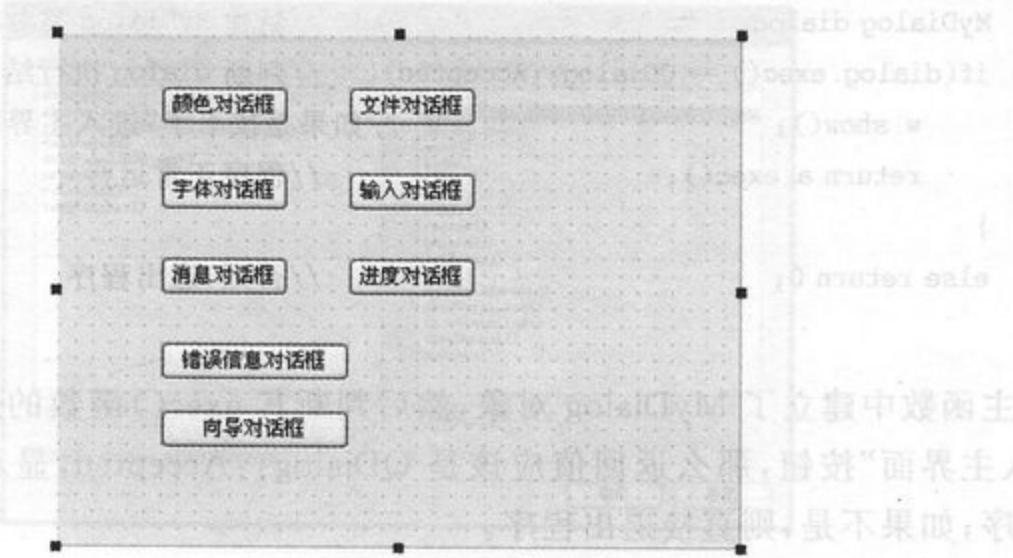


图 3-10 添加完按钮的界面

1. 颜色对话框

颜色对话框类 QColorDialog 提供了一个可以获取指定颜色的对话框部件。下面创建一个颜色对话框。先在 mywidget.cpp 文件中添加 #include <QDebug> 和 #include <QColorDialog> 头文件,然后从设计模式进入“颜色对话框”按钮的 clicked() 单击信号槽。更改如下:

```
void MyWidget::on_pushButton_clicked()
{
    QColor color = QColorDialog::getColor(Qt::red, this, tr("颜色对话框"));
    qDebug() << "color: " << color;
}
```

这里使用了 QColorDialog 的静态函数 getColor() 来获取颜色,它的 3 个参数的作用分别是:设置初始颜色、父窗口和对话框标题。这里的 Qt::red 是 Qt 预定义的颜色对象,可以直接单击该字符串,然后按下 F1 查看其快捷帮助,或者在帮助索引中查找 Qt::GlobalColor 关键字,从而查看到所有的预定义颜色列表。getColor() 函数返回的是一个 QColor 类型数据。现在运行程序,然后单击“颜色对话框”按钮,显示出如图 3-11 所示的颜色对话框。如果不选择颜色直接单击 OK 键,那么输出信息应该是 QColor (ARGB 1, 1, 0, 0),这里的 4 个数值分别代表透明度(alpha)、红色(red)、绿色(green)和蓝色(blue)。它们的数值都是从 0.0~1.0,有效数字为 6 位。对于 alpha 来说,1.0 表示完全不透明,这是默认值,而 0.0 表示完全透明。对于三基色红、绿、蓝的数值,还可以使用 0~255 来表示,颜色对话框中就是使用这种方法。其中 0 表示颜色最浅,255

表示颜色最深。在 0~255 与 0.0~1.0 之间可以通过简单的数学运算来对应，其中 0 对应 0.0, 255 对应 1.0。在颜色对话框中还可以添加对 alpha 的设置，就是在 getColor() 函数中再使用最后一个参数：

```
QColorDialog::getColor(Qt::red, this, tr("颜色对话框"), QColorDialog::ShowAlphaChannel);
```

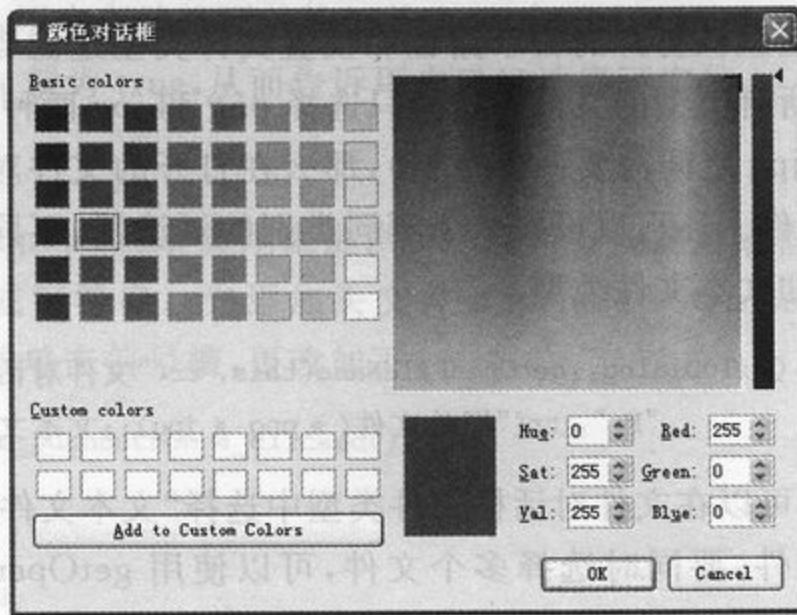


图 3-11 颜色对话框

这里的 QColorDialog::ShowAlphaChannel 就是用来显示 alpha 的设置内容的。

前面使用了 QColorDialog 类的静态函数来直接显示颜色对话框，好处是不用创建对象。但是如果想要更灵活地设置时，可以先创建对象，然后进行各项设置：

```
void MyWidget::on_pushButton_clicked()
{
    QColorDialog dialog(Qt::red, this); // 创建对象
    dialog.setOption(QColorDialog::ShowAlphaChannel); // 显示 alpha 选项
    dialog.exec(); // 以模态方式运行对话框
    QColor color = dialog.currentColor(); // 获取当前颜色
    qDebug() << "color:" << color; // 输出颜色信息
}
```

这样的代码与前面的实现效果是等效的。

2. 文件对话框

文件对话框类 QFileDialog 提供了一个允许用户选择文件或文件夹的对话框。继续添加 #include <QFileDialog> 头文件，然后进入“文件对话框”按钮的单击信号槽，并更改如下：

```
void MyWidget::on_pushButton_2_clicked()
{
    QString fileName = QFileDialog::getOpenFileName(this, tr("文件对话框"),
                                                    "F:", tr("图片文件(*.png *.jpg)"));
```

```
qDebug() << "fileName:" << fileName;
}
```

这里使用了 QFileDialog 类中的 getOpenFileName() 函数来获取选择的文件名, 这个函数会以模态方式运行一个文件对话框, 然后选择一个文件, 单击“打开”按钮后, 这个函数便可以返回选择的文件的文件名。它的 4 个参数的作用分别是: 指定父窗口、设置对话框标题、指定默认打开的目录路径和设置文件类型过滤器。如果不指定文件过滤器, 默认是选择所有类型的文件。这里只选择 png 和 jpg 两种格式的图片文件(注意代码中 * png 和 * jpg 之间需要一个空格), 那么在打开的文件对话框中只能显示目录下这两种格式的文件。还可以设置多个不同类别的过滤器, 不同类别间使用两个分号“;;”隔开, 例如添加文本文件类型:

```
QString fileName = QFileDialog::getOpenFileName(this, tr("文件对话框"),
    "F:", tr("图片文件(*.png *.jpg);;文本文件(*.txt)"));
```

再次运行程序就可以在文件对话框文件类型中选择“文本文件”类型了。前面这个程序只能选择单个文件, 要同时选择多个文件, 可以使用 getOpenFileNames() 函数, 例如:

```
void MyWidget::on_pushButton_2_clicked()
{
    QStringList fileNames = QFileDialog::getOpenFileNames(this, tr("文件对话框"),
        "F:", tr("图片文件(*.png *.jpg)"));
    qDebug() << "fileNames:" << fileNames;
}
```

这时运行程序就可以同时选择多个图片文件了, 多个文件名存放在 QStringList 类型变量中。也可以不使用这些静态函数, 而是建立对话框对象来操作。除了上面的两个函数, QFileDialog 类还提供了 getSaveFileName() 函数来实现保存文件对话框和文件另存为对话框, 还有 getExistingDirectory() 函数来获取一个已存在的文件夹路径。它们的用法与上面的例子类似, 这里就不再举例。

3. 字体对话框

字体对话框 QFontDialog 类提供了一个可以选择字体的对话框部件。先添加 #include <QFontDialog> 头文件, 然后转到“字体对话框”按钮的单击信号槽, 更改如下:

```
void MyWidget::on_pushButton_3_clicked()
{
    //ok 用于标记是否单击了 OK 按钮。然后获得选择的字体
    bool ok;
    QFont font = QFontDialog::getFont(&ok, this);
    //如果单击 OK 按钮, 那么让“字体对话框”按钮使用新字体
```

```

//如果单击 Cancel 按钮,那么输出信息
if (ok) ui->pushButton_3->setFont(font);
else qDebug() << tr("没有选择字体!");
}

```

这里使用了 QFileDialog 类的 getFont() 静态函数来获取选择的字体。第一个参数是 bool 类型变量,用来存放按下的按钮状态,比如在打开的字体对话框中单击了 OK 按钮,那么这里的 ok 就为 true,从而告诉用户已经选择了字体。

4. 输入对话框

输入对话框 QInputDialog 类用来提供一个简单方便的对话框,从而从用户那里获取一个单一的数值或字符串。先添加头文件 #include <QInputDialog>,然后进入“输入对话框”按钮的单击信号槽,更改如下:

```

void MyWidget::on_pushButton_4_clicked()
{
    bool ok;
    //获取字符串
    QString string = QInputDialog::getText(this, tr("输入字符串对话框"),
                                             tr("请输入用户名:"), QLineEdit::Normal, tr("admin"), &ok);
    if(ok) qDebug() << "string:" << string;
    //获取整数
    int value1 = QInputDialog::getInt(this, tr("输入整数对话框"),
                                       tr("请输入 -1000 到 1000 之间的数值"), 100, -1000, 1000, 10, &ok);
    if(ok) qDebug() << "value1:" << value1;
    //获取浮点数
    double value2 = QInputDialog::getDouble(this, tr("输入浮点数对话框"),
                                             tr("请输入 -1000 到 1000 之间的数值"), 0.00, -1000, 1000, 2, &ok);
    if(ok) qDebug() << "value2:" << value2;
    QStringList items;
    items << tr("条目 1") << tr("条目 2");
    //获取条目
    QString item = QInputDialog::getItem(this, tr("输入条目对话框"),
                                         tr("请选择一个条目"), items, 0, true, &ok);
    if(ok) qDebug() << "item:" << item;
}

```

这里一共创建了 4 个不同类型的输入对话框。getText() 函数可以提供一个可输入字符串的对话框,参数的作用分别是:指定父窗口、设置窗口标题、设置对话框中的标签的显示文本、设置输入的字符串的显示模式(例如密码可以显示成小黑点,这里选择了显示用户输入的实际内容)、设置输入框中的默认字符串和设置获取按下按钮信息的 bool 变量。getInt() 函数可以提供一个输入整型数值的对话框,其中的参数 100 表示默认的数值是 100, -1 000 表示可输入的最小值是 -1 000, 1 000 表示可输入的最大值是 1 000。getDouble() 函数可以提供一个输入浮点型数值的对话框,其中的参数 0.00 表示默认的数值是 0.00, -1000 和 1000 分别表示可输入的最小值和最大值。

大值是 1 000,10 表示使用箭头按钮,数值每次变化 10。getDouble() 函数可以提供一个输入浮点型数值的对话框,其中的参数 2 表示小数的位数为 2。getItem() 函数提供一个可以输入一个条目的对话框,需要先给它提供一些条目,例如这里定义的 QStringList 类型的 items。它的参数 0 表示默认显示列表中的第 0 个条目(0 就是第一个);然后是参数 true,设置条目是否可以被更改,true 就是可以被更改。这里使用了这些方便的静态函数,也可以自己定义对象,然后使用相关的函数进行设置。

5. 消息对话框

消息对话框 QMessageBox 类提供了一个模态的对话框用来通知用户一些信息,或者向用户提出一个问题并且获取答案。先添加头文件 #include <QMessageBox>,然后进入“消息对话框”按钮的单击信号槽中,添加如下代码:

```
void MyWidget::on_pushButton_5_clicked()
{
    //问题对话框
    int ret1 = QMessageBox::question(this, tr("问题对话框"),
                                     tr("你了解 Qt 吗?"), QMessageBox::Yes, QMessageBox::No);
    if(ret1 == QMessageBox::Yes) qDebug() << tr("问题!");
    //提示对话框
    int ret2 = QMessageBox::information(this, tr("提示对话框"),
                                       tr("这是 Qt 书籍!"), QMessageBox::Ok);
    if(ret2 == QMessageBox::Ok) qDebug() << tr("提示!");
    //警告对话框
    int ret3 = QMessageBox::warning(this, tr("警告对话框"),
                                   tr("不能提前结束!"), QMessageBox::Abort);
    if(ret3 == QMessageBox::Abort) qDebug() << tr("警告!");
    //错误对话框
    int ret4 = QMessageBox::critical(this, tr("严重错误对话框"),
                                    tr("发现一个严重错误! 现在要关闭所有文件!"), QMessageBox::YesAll);
    if(ret4 == QMessageBox::YesAll) qDebug() << tr("错误");
    //关于对话框
    QMessageBox::about(this, tr("关于对话框"),
                      tr("yafeilinux.com 致力于 Qt 及 Qt Creator 的普及工作!"));
}
```

这里创建了 4 个不同类型的消息对话框,分别拥有不同的图标及提示音(这个是操作系统设置的),参数分别是父窗口、标题栏、显示信息和拥有的按钮。这里使用的按钮都是 QMessageBox 类提供的标准按钮。这几个静态函数的返回值就是那些标准的按钮,它们是枚举类型,可以使用这个返回值来判断用户按下了哪个按钮。about() 函数没有返回值,因为它默认只有一个按钮,与其相似的还有一个 aboutQt() 函数,用来显示现在使用的 Qt 版本等相关信息。如果想使用自己的图标和自定义按钮,那么可以创建 QMessageBox 类对象,然后使用相关函数进行操作。

6. 进度对话框

进度对话框 QProgressDialog 对一个耗时较长的操作进度提供了反馈。还是先添加 #include <QProgressDialog> 头文件,然后进入“进度对话框”按钮的单击信号槽,更改如下:

```
void MyWidget::on_pushButton_6_clicked()
{
    QProgressDialog dialog(tr("文件复制进度"), tr("取消"), 0, 50000, this);
    dialog.setWindowTitle(tr("进度对话框")); //设置窗口标题
    dialog.setWindowModality(Qt::WindowModal); //将对话框设置为模态
    dialog.show();
    for(int i = 0; i < 50000; i++) { //演示复制进度
        dialog.setValue(i); //设置进度条的当前值
        QApplication::processEvents(); //避免界面冻结
        if(dialog.wasCanceled()) break; //按下取消按钮则中断
    }
    dialog.setValue(50000); //这样才能显示 100 %,因为 for 循环中少加了一个数
    qDebug() << tr("复制结束!");
}
```

这里创建了一个 QProgressDialog 类对象 dialog,构造函数的参数分别为对话框的标签内容、取消按钮的显示文本、最小值、最大值和父窗口。然后将其设置为了模态对话框并显示。后面的 for() 循环语句模拟了文件复制过程,使用 setValue() 函数使进度条向前推进。为了避免长时间的操作而使用户界面冻结,必须不断调用 QApplication 类的静态函数 processEvents(),可以将它放在 for() 循环语句中。然后使用 QProgressDialog 的 wasCanceled() 函数来判断用户是否单击了“取消”按钮,如果是,则中断复制过程。这里使用了模态对话框,其实 QProgressDialog 还可以实现非模态对话框,不过它需要定时器等的帮助。

7. 错误信息对话框

错误信息对话框 QErrorMessage 类提供了一个显示错误信息的对话框。首先添加头文件 #include <QErrorMessage>,然后转到“错误信息对话框”按钮的单击信号槽添加代码:

```
void MyWidget::on_pushButton_7_clicked()
{
    QErrorMessage * dialog = new QErrorMessage(this);
    dialog->setWindowTitle(tr("错误信息对话框"));
    dialog->showMessage(tr("这里是出错信息!"));
}
```

这里新建了一个 QErrorMessage 对话框，并且调用它的 showMessage() 函数来显示错误信息，调用这个函数时对话框会以非模态的形式显示出来。在错误信息对话框中默认有一个 Show this message again 复选框，可以选择以后是否还要显示相同错误信息。

8. 向导对话框

向导对话框 QWizard 类提供了一个设计向导界面的框架。对于向导对话框读者应该已经很熟悉了，比如安装软件时的向导和创建项目时的向导。QWizard 之所以被称为框架，是因为它具有设计一个向导的全部功能函数，可以使用它来实现想要的效果。Qt 演示程序中的 Dialogs 分类下有 Trivial Wizard、License Wizard 和 Class Wizard 这 3 个示例程序，可以参考一下。

打开 mywidget.h 文件，然后添加头文件 #include <QWizard>，在 MyWidget 类的声明中添加 private 类型函数声明：

```
private:
    Ui::MyWidget * ui;
    QWizardPage * createPage1();           //新添加
    QWizardPage * createPage2();           //新添加
    QWizardPage * createPage3();           //新添加
```

这里声明了 3 个返回值为 QWizardPage 类对象的指针的函数，用来生成 3 个向导页面。然后在 mywidget.cpp 文件中对这 3 个函数进行定义：

```
QWizardPage * MyWidget::createPage1()      //向导页面 1
{
    QWizardPage * page = new QWizardPage;
    page ->setTitle(tr("介绍"));
    return page;
}

QWizardPage * MyWidget::createPage2()      //向导页面 2
{
    QWizardPage * page = new QWizardPage;
    page ->setTitle(tr("用户选择信息"));
    return page;
}

QWizardPage * MyWidget::createPage3()      //向导页面 3
{
    QWizardPage * page = new QWizardPage;
    page ->setTitle(tr("结束"));
    return page;
}
```

在各个函数中分别新建了向导页面，并且设置了标题。下面进入“向导对话框”按

钮的单击信号槽中,更改如下:

```
void MyWidget::on_pushButton_8_clicked()
{
    QWizard wizard(this);
    wizard.setWindowTitle(tr("向导对话框"));
    wizard.addPage(createPage1());           //添加向导页面
    wizard.addPage(createPage2());
    wizard.addPage(createPage3());
    wizard.exec();
}
```

这里新建了 `QWizard` 类对象,然后使用 `addPage()` 函数为其添加了 3 个页面,这里的参数是 `QWizardPage` 类型的指针,所以直接使用了生成向导页面函数。运行程序可以看到,向导页面出现的顺序和添加向导页面的顺序是一致的。

上面程序中的向导页面是线性的,而且什么内容也没有添加。如果想设计自己的向导页面,或添加图片和自定义按钮,或设置向导页面顺序等,那么就需要再多了解 `QWizard` 类和 `QWizardPage` 类的成员函数了。

到这里,Qt 提供的几个标准对话框就基本讲完了。其实还有 3 个与打印有关的标准对话框类 `QPageSetupDialog`(页面设置对话框)、`QPrintDialog`(打印对话框)和 `QPrintPreviewDialog`(打印预览对话框)这里暂时没有介绍,它们将在第 5 章使用时再统一讲解。关于这些对话框的使用,Qt 提供了一个示例程序 Standard Dialogs,它在 Dialogs 分类中。

3.3 其他窗口部件

下面介绍其他一些常用的窗口部件,这些部件都是图 3-1 中显示的,继承自或者间接继承自 `QWidget` 类。

3.3.1 QFrame 类族

`QFrame` 类是带有边框的部件的基类,它的子类有最为常用的标签部件 `QLabel`,另外还有 `QLCDNumber`、`QSplitter`、`QStackedWidget`、`QToolBox` 和 `QAbstractScrollArea` 类。`QAbstractScrollArea` 类是所有带有滚动区域的部件类的抽象基类,这里需要说明,在 Qt 中凡是带有 Abstract 字样的类都是抽象基类。对于抽象基类,我们是不能直接使用的,但是可以继承该类实现自己的类,或者使用它提供的子类。`QAbstractScrollArea` 的子类中有最常用的文本编辑器类 `QTextEdit` 类和各种项目视图类。

带边框部件最主要的特点就是可以有一个明显的边界框架。`QFrame` 类的主要功能也就是用来实现不同的边框效果,这主要是由边框形状(Shape)和边框阴影(Shadow)组合来形成的。`QFrame` 类中定义的主要边框形状如表 3-1 所列,边框阴影如

表 3-2 所列。其中, `lineWidth` 是边框边界的线的宽度;而 `midLineWidth` 是在边框中额外插入的一条线的宽度,作用是形成 3D 效果,并且只在 Box、Hline 和 VLine 表现为凸起或者凹陷时有用。QFrame 的这些元素组合成的所有边框效果,如图 3-12 所示。

表 3-1 QFrame 类边框形状的取值

常量	描述
<code>QFrame::NoFrame</code>	<code>QFrame</code> 什么也不绘制
<code>QFrame::Box</code>	<code>QFrame</code> 在它的内容四周绘制一个边框
<code>QFrame::Panel</code>	<code>QFrame</code> 绘制一个面板,使得内容表现为凸起或者凹陷
<code>QFrame::StyledPanel</code>	绘制一个矩形面板,它的效果依赖于当前的 GUI 样式,可以凸起或凹陷
<code>QFrame::HLine</code>	<code>QFrame</code> 绘制一条水平线,没有任何框架(可以作为分离器)
<code>QFrame::VLine</code>	<code>QFrame</code> 绘制一条垂直线,没有任何框架(可以作为分离器)
<code>QFrame::WinPanel</code>	绘制一个类似于 Windows 2000 中的矩形面板,可以凸起或者凹陷

表 3-2 QFrame 类边框阴影的取值

常量	描述
<code>QFrame::Plain</code>	边框和内容没有 3D 效果,与四周界面在同一水平面上
<code>QFrame::Raised</code>	边框和内容表现为凸起,具有 3D 效果
<code>QFrame::Sunken</code>	边框和内容表现为凹陷,具有 3D 效果

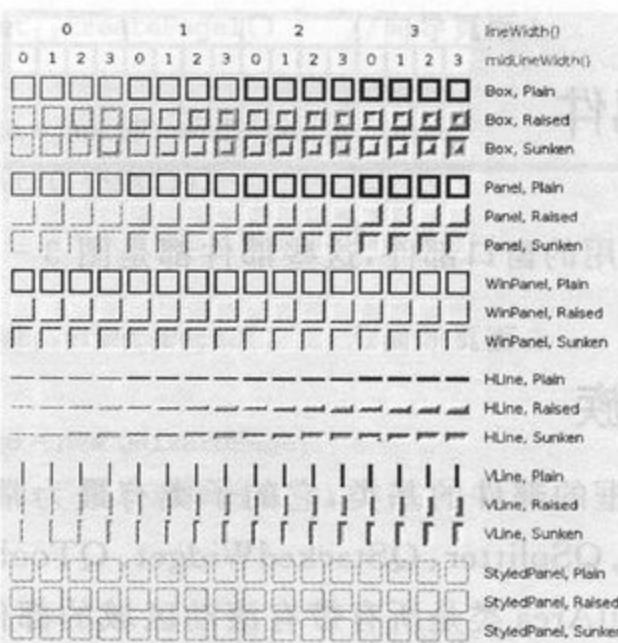


图 3-12 QFrame 类边框效果

下面在程序中演示具体的效果。(项目源码路径: `src\03\3-8\myFrame`)新建 Qt Gui 应用,项目名称为 `myFrame`,类名 `MyWidget`,选择 `QWidget` 为基类。打开 `mywidget.ui` 文件,在 Qt 设计器中从部件列表里拖一个 Frame 到界面上,然后在右下方的属性栏中更改其 `frameShape` 为 `Box`, `frameShadow` 为 `Sunken`, `lineWidth` 为 5, `midLine-`

Width 为 10。完成后 Frame 的效果如图 3-13 所示。在属性栏中设置部件的属性，这和在源码中用代码实现是等效的，也可以直接在 mywidget.cpp 文件中的 MyWidget 构造函数里使用如下代码来代替：

```
ui->frame->setFrameShape(QFrame::Box);
ui->frame->setFrameShadow(QFrame::Sunken);
//以上两个函数可以使用 setFrameStyle(QFrame::Box | QFrame::Sunken)代替
ui->frame->setLineWidth(5);
ui->frame->setMidLineWidth(10);
```

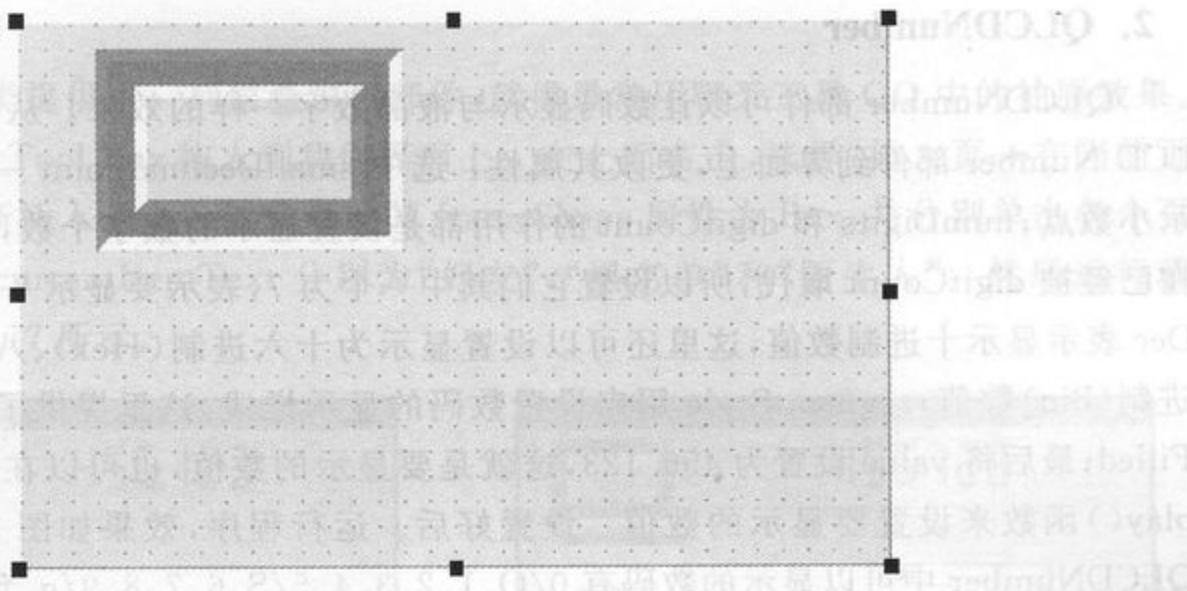


图 3-13 设置 Frame 边框

因为下面要讲的部件大都是 Qt 的标准部件，所以大多会在 Qt 设计器中直接设置其属性。能在属性栏中设置的属性，其类中就一定有相关的函数可以使用代码来实现，只要根据名字在类的参考文档中查找就可以了。

对于 QFrame 的子类，它们都继承了 QFrame 的边框设置功能，所以下面对子类的介绍中就不再涉及这方面的内容了，而是讲解各个子类的独有特性。

1. QLabel

标签 QLabel 部件用来显示文本或者图片。在设计器中往界面拖入一个 Label，然后将其拖大，并在属性栏中设置其对齐方式为 alignment 属性，水平的改为 AlignHCenter，垂直的改为 AlignVCenter，这样 QLabel 中的文本就会显示在正中间。也可以设置为其他方式查看效果。属性栏中还有一个 wordWrap，选中它就可以实现文本的自动换行。下面看一下怎么在标签中使用图片。首先在 mywidget.cpp 文件中添加头文件 #include <QPixmap>，然后在 MyWidget 类的构造函数中添加一行代码：

```
ui->label->setPixmap(QPixmap("F:/logo.png"));
```

这样就可以在标签中显示 F 盘中的“logo.png”图片了。显示图片时使用了图片的绝对路径“F:/logo.png”，这样需要指明盘符。也可以使用相对路径，比如将 logo.png 图片放到项目目录的 myFrame-build-desktop 文件夹中，那么只需要使用相对路径“lo-

go.png”就可以了。最好的方法是使用资源管理器，将图片放到程序中，这个会在第5章讲述。在 QLabel 中还可以显示 gif 动态图片，在 mywidget.cpp 中添加头文件 #include <QMovie>，然后在 myWidget 的构造函数中继续添加代码：

```
QMovie * movie = new QMovie("F:/donghua.gif");
ui->label->setMovie(movie); //在标签中添加动画
movie->start(); //开始播放
```

这时运行程序，效果如图 3-14 所示。可以看到，新添加的图片会遮盖以前的图片。

2. QLCDNumber

QLCDNumber 部件可以让数码显示与液晶数字一样的效果。从部件栏中拖一个 LCD Number 部件到界面上，更改其属性：选中 smallDecimalPoint 一项，这样可以显示小数点；numDigits 和 digitCount 的作用都是设置显示的数字个数，而 numDigits 现在已经被 digitCount 取代，所以设置它们其中一个为 7，表示要显示 7 个数字；mode 选 Dec 表示显示十进制数值，这里还可以设置显示为十六进制（Hex）、八进制（Oct）和二进制（Bin）数值；segmentStyle 用来设置数码的显示样式，这里提供了 3 种样式，选择 Filled；最后将 value 设置为 456.123，这就是要显示的数值，也可以在代码中使用 display() 函数来设置要显示的数值。设置好后。运行程序，效果如图 3-15 所示。在 QLCDNumber 中可以显示的数码有 0/O、1、2、3、4、5/S、6、7、8、9/g、负号、小数点、A、B、C、D、E、F、h、H、L、o、P、r、u、U、Y、冒号、度符号（输入时使用单引号来代替）和空格。



图 3-14 QLabel 中使用 gif 图片

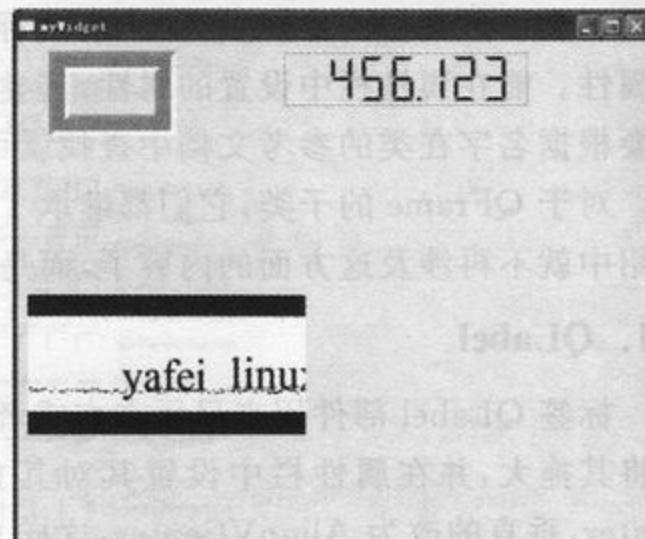


图 3-15 显示液晶数字效果

3. QStackedWidget

QStackedWidget 类提供了一个部件栈，可以有多个界面（称为页面），每个界面可以拥有自己的部件，不过每次只能显示一个界面。对于这个部件，需要使用 QComboBox 或者 QListWidget 来选择它的各个页面。在设计模式中向界面拖入一个 List Widget 和一个 Stacked Widget。在 List Widget 上右击，选择“编辑项目”菜单，然后在编辑

列表窗口部件对话框中按下左下角的加号添加两项，并更该名称为“第一页”和“第二页”。然后在 Stacked Widget 上拖一个 Label，更改文本为“第一页”，然后再单击 Stacked Widget 右上角的小箭头进入下一页，再拖入一个标签，更改文本为“第二页”，然后再将 Stacked Widget 部件的 frameShape 属性更改为 StyledPanel。最后在信号和槽设计模式，将 listWidget 部件的 currentRowChanged() 信号和 stackedWidget 的 setCurrentIndex() 槽关联。设置完成后运行程序，效果如图 3-16 所示。可以看到，现在可以单击 listWidget 中的项目来选择 stackedWidget 的页面了。也可以在设计模式中在 stackedWidget 上右击来添加新的页面。

4. QToolBox

QToolBox 类提供了一列层叠窗口部件，就像最常用聊天工具 QQ 中的抽屉效果。从部件栏中选择 Tool Box 拖入到我们界面上。在上面右击，选择“插入页→在当前页之后”菜单项来新插入一页。然后更改其 frameShape 属性为 Box，并分别单击各个页的标签，更改其 currentItemText 分别为“好友”，“黑名单”和“陌生人”。然后运行程序，效果如图 3-17 所示。

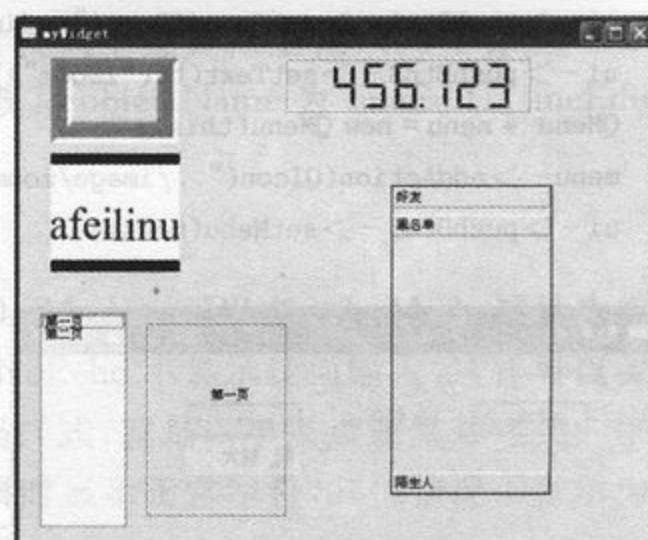
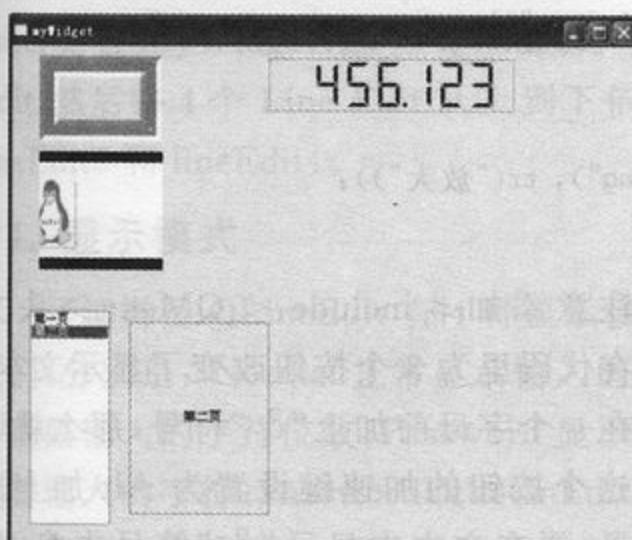


图 3-16 QStackedWidget 部件运行效果

图 3-17 QToolBox 部件运行效果

3.3.2 按钮部件

QAbstractButton 类是按钮部件的抽象基类，提供了按钮的通用功能。它的子类包括复选框 QCheckBox、标准按钮 QPushButton、单选框按钮 QRadioButton 和工具按钮 QToolButton。关于这一节，可以参考示例程序 Group Box Example，它在 Widgets 分类中。

(项目源码路径：src\03\3-9\myButton)新建 Qt Gui 应用，项目名称 myButton，类名 MyWidget，基类选择 QWidget。项目建立好之后，在 main.cpp 中使用 setCodecForTr() 函数保证编写代码时可以使用中文。然后在项目文件夹中新建 image 文件夹，并且放入几张图片，供下面编写程序时使用。

1. QPushButton

QPushButton 部件提供了一个标准按钮。在新建的项目中打开 mywidget.ui 文件, 拖入 3 个 Push Button 到界面上, 将它们的 objectName 依次更改为 pushBtn1、pushBtn2 和 pushBtn3。选中 pushBtn1 的 checkable 属性, 使得它可以拥有“选中”和“未选中”两种状态; 再选中 pushBtn2 的 flat 属性, 这样可以不显示该按钮的边框。然后进入 pushBtn1 的 toggled(bool) 信号的槽, 更改如下:

```
void MyWidget::on_pushButton_toggled(bool checked) //按钮是否处于被按下状态
{
    qDebug() << tr("按钮是否按下:") << checked;
}
```

注意添加 `#include <QDebug>` 头文件。当 pushBtn1 处于按下状态的时候 checked 为 true, 否则为 false。下面在 MyWidget 类的构造函数中添加代码:

```
ui->pushBtn1->setText(tr("&nihao")); //这样便指定了 Alt + N 为加速键
ui->pushBtn2->setText(tr("帮助(&H)"));
ui->pushBtn2->setIcon(QIcon("../image/help.png"));
ui->pushBtn3->setText(tr("z&oom"));
QMenu * menu = new QMenu(this);
menu->addAction(QIcon("../image/zoom-in.png"), tr("放大"));
ui->pushBtn3->setMenu(menu);
```



图 3-18 标准按钮运行效果

注意添加 `#include <QMenu>` 头文件。在代码里为 3 个按钮改变了显示文本, 这里在一个字母前加上“&”符号, 那么就可以将这个按钮的加速键设置为 Alt 加上这个字母; 要在文本中显示“&”符号本身, 那么可以使用“&&”。也可以使用 `setIcon()` 函数来给按钮添加图标, 这里图片文件使用了相对路径(也可以在设计模式通过更改 icon 属性来实现)。对于 pushBtn3, 我们为其添加了下拉菜单, 当然, 现在这个菜单什么功能也没实现。运行程序, 效果如图 3-18 所示。

2. QCheckBox、QRadioButton 和 QGroupBox

对于调查表之类的应用, 往往提供多个选项供选择, 有些是可以选择多项的, 有些只能选择其中一项。复选框 `QCheckBox` 类提供了同时选择多项的功能, 而 `QRadioButton` 提供了只能选择一项的功能, 一般要把一组按钮放到一个 `QGroupBox` 中来进行管理。

下面在界面上拖入两个 Group Box, 将标题分别改为“复选框”和“单选框”。然后往复选框中拖入 3 个 Check Box, 分别更改显示内容为“跑步”、“踢球”和“游泳”。再往单选框中拖入 3 个 Radio Button, 分别更改其内容为“很好”、“一般”和“不好”。这里还可以选中 Check Box 的 tristate 属性, 让它拥有不改变状态、选中状态和未选中状态 3 种状态。对于选择按钮后的操作, 可以关联 stateChanged() 信号和我们的槽, 也可以使用 isChecked() 函数查看一个按钮是否被选中。除了 Group Box, 还可以使用 QButtonGroup 类来管理按钮。

3.3.3 行编辑器

行编辑器 QLineEdit 部件是一个单行的文本编辑器, 允许用户输入和编辑单行的纯文本内容, 而且提供了一系列有用的功能, 包括撤销与恢复、剪切和拖放等操作。其中剪切复制等功能是行编辑自带的, 不用自己编码实现。这里主要讲解行编辑器的其他几个特殊用法。对应这部分内容, 可以查看 Qt 的示例程序 Line Edits, 它在 Widgets 分类中。

(项目源码路径: src\03\3-10\myLineEdit) 新建 Qt Gui 应用, 项目名称 myLineEdit, 类名 MyWidget, 基类 QWidget。在设计模式中往界面上拖入几个标签和 Line Edit, 然后将 4 个 Line Edit 从上到下依次更改其 objectName 为 lineEdit1、lineEdit2、lineEdit3 和 lineEdit4。

1. 显示模式

行编辑器 QLineEdit 有 4 种显示模式 (echoMode), 可以在 echoMode 属性中更改它们, 分别是: Normal 正常显示输入的信息; NoEcho 不显示任何输入, 这样可以保证不泄露输入的字符位数; Password 显示为密码样式, 就是以小黑点或星号之类的字符代替输入的字符; PasswordEchoOnEdit 在编辑时显示正常字符, 其他情况下显示为密码样式。这里设置 lineEdit1 的 echoMode 为 Password。

2. 输入掩码

QLineEdit 提供了输入掩码 (inputMask) 来限制输入的内容。可以使用一些特殊的字符来设置输入的格式和内容, 这些字符中有的起限制作用且必须要输入一个字符, 而有的只是起限制作用, 但可以不输入字符而是以空格代替。先来看一下这些特殊字符的含义, 如表 3-3 所列。

表 3-3 QLineEdit 掩码字符

字符(必须输入)	字符(可留空)	含 义
A	a	只能输入 A~Z, a~z
N	n	只能输入 A~Z, a~z, 0~9
X	x	可以输入任意字符
9	0	只能输入 0~9

续表 3-3

字符(必须输入)	字符(可留空)	含义
D	d	只能输入 1~9
	#	只能输入加号(+)，减号(-)，0~9
H	h	只能输入十六进制字符，A~F, a~f, 0~9
B	b	只能输入二进制字符，0、1
>		后面的字母字符自动转换为大写
<		后面的字母字符自动转换为小写
!		停止字母字符的大小写转换
\		将该表中的特殊字符正常显示用作分隔符

下面将 `lineEdit2` 的 `inputMask` 属性设置为“>AA-90-bb-! aa\#H; *”，含义为：“>”号表明后面输入的字母自动转为大写；“AA”表明开始必须输入两个字母，因为有前面的“>”号的作用，所以输入的这两个字母自动变为大写；“-”号为分隔符，直接显示，该位不可输入；“9”表示必须输入一个数字；“0”表示输入一个数字，或者留空；“bb”表示这两位可以留空，或者输入两个二进制字符，即 0 或 1；“!”表明停止大小写转换，就是在最开始的“>”号不再起作用；“aa”表示可以留空，或者输入两个字母；“\#”表示将“#”号作为分隔符，因为“#”号在这里有特殊含义，所以前面要加上“\”号；“H”表明必须输入一个十六进制的字符；“; *”表示用“*”号来填充空格。也可以使用 `setInputMask()` 函数在代码中来设置输入掩码。

在 `lineEdit2` 上右击，然后转到它的 `returnPressed()` 回车键按下信号的槽中。更改如下：

```
void MyWidget::on_lineEdit2_returnPressed() //回车键按下信号的槽
{
    ui->lineEdit3->setFocus(); //让 lineEdit3 获得焦点
    qDebug() << ui->lineEdit2->text(); //输出 lineEdit2 的内容
    qDebug() << ui->lineEdit2->displayText(); //输出 lineEdit2 显示的内容
}
```

注意要添加 `#include <QDebug>` 头文件。这里先让下一个行编辑器获得焦点，然后输出了 `lineEdit2` 的内容和显示出来的内容，它们有时是不一样的，编程时更多的是使用 `text()` 函数来获取它的内容。这时运行程序，输入完所需内容后按下回车键即可，如果没有输入完那些必须要输入的字符，那么按下回车键是不起作用的。

3. 输入验证

在 `QLineEdit` 中还可以使用验证器(`validator`)来对输入进行约束。它使用起来很简单，在 `mywidget.cpp` 文件中的 `MyWidget` 类的构造函数中添加代码：

```
//新建验证器，指定范围为 100~999
```

```

QValidator * validator = new QIntValidator(100, 999, this);
//在行编辑器中使用验证器
ui->lineEdit3->setValidator(validator);

```

在代码中为 `lineEdit3` 添加了验证器,那么它现在只能输入 100~999 之间的数字。现在再进入 `lineEdit3` 的回车键按下信号的槽,输出 `lineEdit3` 的内容。然后运行程序会发现,其他的字符无法输入,而输入小于 100 的数字,那么按下调用键也是没有效果的。

在 `QValidator` 中还提供了 `QDoubleValidator`,可以用来设置浮点数。如果想设置更一般的字符约束,就要使用正则表达式了。这个在第 7 章会讲到,这里举一个简单的例子:

```

QRegExp rx("-? \d{1,3}");
QValidator * validator = new QRegExpValidator(rx, this);

```

这样就可以实现在开始输入“-”号或者不输入,然后输入 1~3 个数字的限制。

4. 自动补全

在 `QLineEdit` 中也提供了强大的自动补全功能,这是利用 `QCompleter` 类实现的。在 `MyWidget` 类的构造函数中继续添加代码:

```

QStringList wordList;
wordList << "Qt" << "Qt Creator" << tr("你好");
QCompleter * completer = new QCompleter(wordList, this); //新建自动完成器
completer->setCaseSensitivity(Qt::CaseInsensitive); //设置大小写不敏感
ui->lineEdit4->setCompleter(completer);

```

这里要添加 `#include <QCompleter>` 头文件,因为代码中使用了中文,所以还要在 `main.cpp` 文件中使用 `setCodecForTr()` 函数。关于 `QCompleter` 的使用,可以参考 Qt 的示例程序 `Completer`,它在 Tools 分类下。

3.3.4 数值设定框

`QAbstractSpinBox` 类是一个抽象基类,提供了一个数值设定框和一个行编辑器来显示设定值。它有 3 个子类 `QDateTimeEdit`、`QSpinBox` 和 `QDoubleSpinBox` 分别用来完成日期时间、整数和浮点数的设定。关于这一小节,可以查看 Widgets 分类下的 Spin Boxes 示例程序。

(项目源码路径: `src\03\3-11\mySpinBox`) 新建 Qt Gui 应用,项目名称 `mySpinBox`,类名 `MyWidget`,基类为 `QWidget`。

1. QDateTimeEdit

`QDateTimeEdit` 类提供了一个可以编辑日期和时间的部件。从部件栏中分别拖入 Time Edit、Date Edit 和 Date/Time Edit 到界面上,然后设置 `timeEdit` 的 `display-`

Format 为“h:mm:ssA”这样就可以使用 12 小时制来显示。选中 dateEdit 的 calendarPopup 属性就可以使用弹出的日历部件来设置日期了。然后在 MyWidget 类的构造函数中添加代码：

```
//设置时间为现在的系统时间
ui->dateTimeEdit->setDateTime(QDateTime::currentDateTime());
//设置时间的显示格式
ui->dateTimeEdit->setDisplayFormat(tr("yyyy 年 MM 月 dd 日 ddd HH 时 mm 分 ss 秒"));
```

这里使用代码设置了 dateTimeEdit 中的日期和时间。y 表示年；M 表示月；d 表示日，而 ddd 表示星期；H 表示小时，使用 24 小时制显示，而 h 也表示小时，如果最后有 AM 或者 PM 的，则是 12 小时制显示，否则使用 24 小时制；m 表示分；s 表示秒；还有一个 z 可以用来表示毫秒。更多的格式可以参考 QDateTime 类。因为这里使用了中文，所以要在主函数中添加相应的代码。可以使用该部件的 text() 函数来获取设置的值，它返回 QString 类型的字符串；也可以使用 dateTime() 函数，它返回的是 QDateTime 类型数据。

2. QSpinBox 和 QDoubleSpinBox

QSpinBox 用来设置整数，QDoubleSpinBox 用来设置浮点数，这两个部件在前面的输入对话框中已经接触过了。从部件栏中找到 Spin Box 和 Double Spin Box 并将它们拖入到界面上，可以在属性栏中看到 spinBox 的属性有：后缀 suffix 属性，可以设置为“%”，这样就可以显示百分数了；前缀 prefix 属性，比如表示金钱时前面有“¥”字符；最小值 minimum 属性，设置其最小值；最大值 maximum 属性设置其最大值；单步值 singleStep 属性设置每次增加的数值，默认为 1；value 为现在显示的数值。而 doubleSpinBox 又增加了一个小数位数 decimals 属性，用来设置小数点后面的位数。可以在程序中使用 value() 函数来获取设置的数值。

3.3.5 滑块部件

QAbstractSlider 类提供了一个区间内的整数值，有一个滑块，可以定位到一个整数区间的任意值。这个类是一个抽象基类，有 3 个子类 QScrollBar、QSlider 和 QDial，其中，滚动条 QScrollBar 更多用在 QScrollArea 类中来实现滚动区域；而 QSlider 是最常见的音量控制或多媒体播放进度等滑块；QDial 是一个刻度表盘。关于这些部件，可以参考 Widgets 分类下的 Sliders 示例程序。

(项目源码路径：src\03\3-12\mySlider)新建 Qt Gui 应用，项目名称 mySlider，类名 MyWidget，基类 QWidget。然后从部件栏中分别将 Dial、Horizontal Scroll Bar 和 Vertical Scroll Bar、Horizontal Slider 以及 Vertical Slider 等部件拖入到界面上。

先看两个 Scroll Bar 的属性：maximum、minimum 分别用来设置最大、最小值；singleStep 属性是每步的步长，默认是 1，就是按下方向键后其数值增加或者减少 1；pageStep 是每页的步长，默认是 10，就是按下 PageUp 或者 PageDown 按钮后，其数值

增加或者减少 10; value 与 sliderPosition 是当前值; tracking 是设置是否跟踪, 默认认为是, 就是在拖动滑块时, 每移动一个刻度, 都会发射 valueChanged() 信号。而如果选择否, 则只有拖动滑块释放时才发射该信号; orientation 是设置部件的方向, 有水平和垂直两种选择; invertedAppearance 属性是设置滑块所在的位置, 比如默认滑块开始在最左端, 选中这个属性后, 滑块默认就会在最右端。invertedControls 是设置反向控制。再来看两个 Slider, 它们有自己的两个属性 tickPosition 和 tickInterval。前者用来设置显示刻度的位置, 默认是不显示刻度; 后者是设置刻度的间隔。Dial 有自己的属性 wrapping, 用来设置是否首尾相连, 默认开始与结束是分开的; 属性 notchTarget 用来设置刻度之间的间隔; 属性 notchesVisible 用来设置是否显示刻度。

再往界面上拖入一个 Spin Box, 进入信号和槽编辑界面, 将刻度表盘部件 dial 的 sliderMoved(int) 信号分别与其他各个部件的 setValue(int) 槽相连接, 如图 3-19 所示。设置完成后运行程序, 然后使用鼠标拖动刻度盘部件的滑块, 可以看到其他所有的部件都跟着变化了, 如图 3-20 所示。

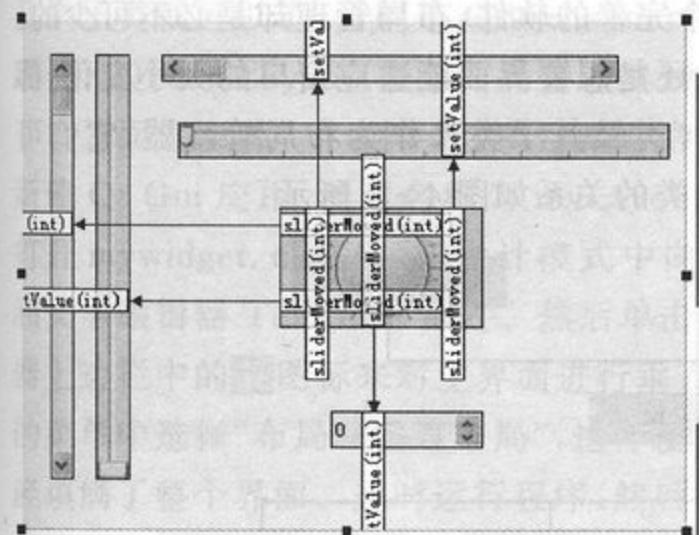


图 3-19 滑块部件间设置信号和槽

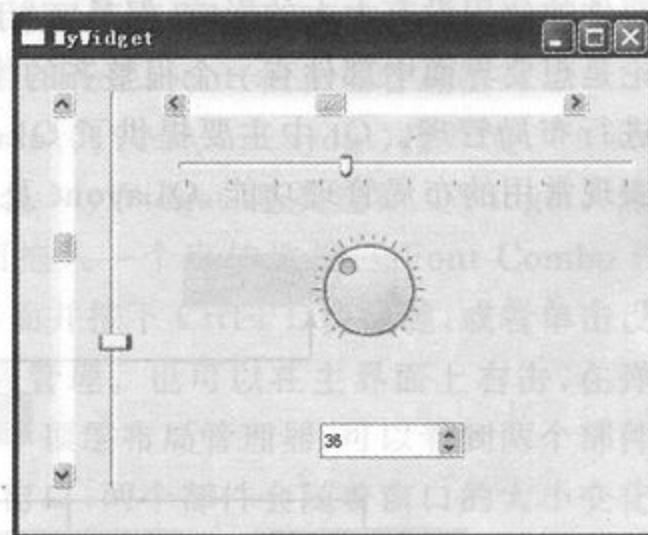


图 3-20 滑块部件运行效果

3.4 小结

本章介绍了常用窗口部件的使用方法, 其中还涉及了程序调试、信号和槽等知识。学习完本章, 读者没有必要把所有讲到的部件都熟练掌握, 只要心中有个印象, 并且大概了解各个部件可以实现的功能即可, 以后使用时可以再回头来参考学习。

最重要的是掌握程序的创建流程和各个部件类之间的相互关系, 而且要掌握信号、槽以及 qDebug() 函数, 这是以后 Qt 编程中经常要用到的。

第 4 章

布局管理

第 3 章讲述了一些窗口部件,当时往界面上拖放部件时都是随意放置的,这对于学习部件的使用没有太大的影响,但是,对于一个完善的软件,布局管理却是必不可少的。无论是想要界面中部件有一个很整齐的排列,还是想要界面能适应窗口的大小变化,都要进行布局管理。Qt 中主要提供了 QLayout 类及其子类来作为布局管理器,它们可以实现常用的布局管理功能,QLayout 及其子类的关系如图 4-1 所示。

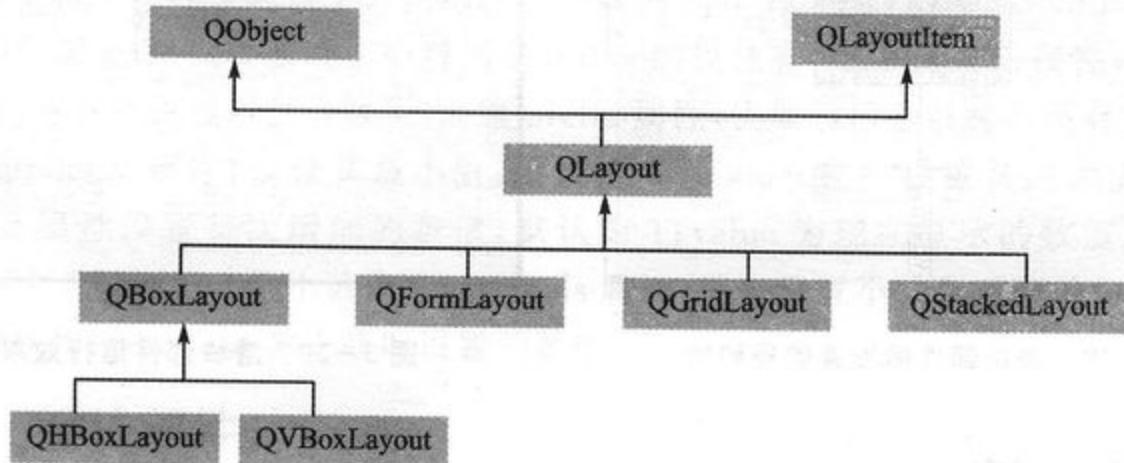


图 4-1 QLayout 类关系图

4.1 布局管理系统

Qt 的布局管理系统提供了简单而强大的机制来自动排列一个窗口中的部件,确保它们有效地使用空间。Qt 包含了一组布局管理类来描述怎样在应用程序的用户界面中对部件进行布局,比如 QLayout 的几个子类,这里将它们称作布局管理器。所有 QWidget 类的子类的实例(对象)都可以使用布局管理器来管理位于其中的子部件, QWidget::setLayout() 函数可以在一个部件上应用布局管理器。一旦一个部件上设置了布局管理器,那么它会完成以下几种任务:

- 定位子部件；
- 感知窗口默认大小；
- 感知窗口最小大小；
- 改变大小处理；
- 当内容改变时自动更新：
 - 字体大小，文本或子部件的其他内容随之改变；
 - 隐藏或显示子部件；
 - 移除一个子部件。

4.1.1 布局管理器

QLayout 类是布局管理器的基类，是一个抽象基类，继承自 QObject 和 QLayoutItem 类，而 QLayoutItem 类提供了一个供 QLayout 操作的抽象项目。QLayout 和 QLayoutItem 都是在设计自己的布局管理器时才使用的，一般只需要使用 QLayout 的几个子类就可以了，分别是 QVBoxLayout(基本布局管理器)、QGridLayout(栅格布局管理器)、QFormLayout(表单布局管理器)和 QStackedLayout(栈布局管理器)。

下面先来看一个例子。(项目源码路径：src\04\4-1\myLayout)打开 Qt Creator，新建 Qt Gui 应用，项目名称为 myLayout，类名 MyWidget，基类选择 QWidget。然后打开 mywidget.ui 文件，在设计模式中向界面拖入一个字体选择框 Font Combo Box 和文本编辑器 Text Edit 部件。然后单击主界面并按下 Ctrl+L 快捷键，或者单击设计器上边栏中的图标来对主界面进行垂直布局管理。也可以在主界面上右击，在弹出的菜单中选择“布局→垂直布局”，这样便设置了顶层布局管理器，可以看到两个部件已经填满了整个界面。这时运行程序，然后拉伸窗口，两个部件会随着窗口的大小变化而变化，如图 4-2 所示。这就是布局管理器在起作用。

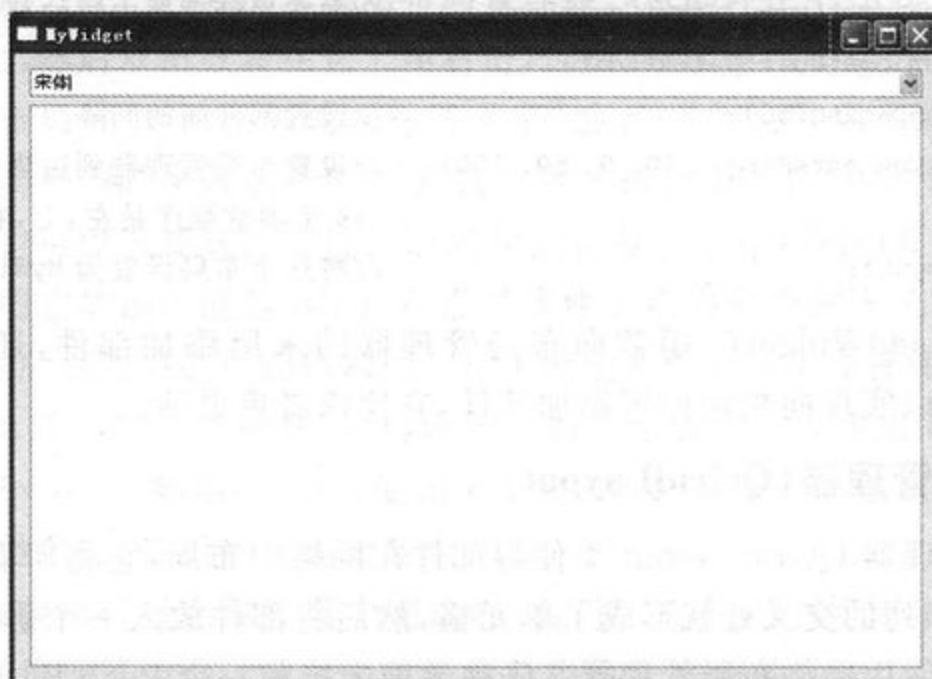


图 4-2 垂直布局管理器运行效果

1. 基本布局管理器(QBoxLayout)

基本布局管理器 QBoxLayout 类可以使子部件在水平方向或者垂直方向排成一列, 将所有的空间分成一行盒子, 然后将每个部件放入一个盒子中。它有两个子类 QHBoxLayout 水平布局管理器和 QVBoxLayout 垂直布局管理器, 编程中经常用到。再回到程序中看看布局管理器的属性。先单击主界面, 查看它的属性栏, 最后面的部分是其使用的布局管理器的属性, 如表 4-1 所列。

表 4-1 布局管理器常用属性说明

属性	说明
layoutName	现在所使用的布局管理器的名称
layoutLeftMargin	设置布局管理器到界面左边界的距离
layoutTopMargin	设置布局管理器到界面上边界的距离
layoutRightMargin	设置布局管理器到界面右边界的距离
layoutBottomMargin	设置布局管理器到界面下边界的距离
layoutSpacing	布局管理器中各个子部件间的距离
layoutStretch	伸缩因子
layoutSizeConstraint	设置大小约束条件

下面打破已有布局, 使用代码实现水平布局。在界面上右击, 然后在右键菜单中选择“打破布局”, 或者单击设计器上边栏中的打破布局图标。在 mywidget.cpp 文件中添加头文件 #include <QHBoxLayout>, 并在 MyWidget 类的构造函数中添加如下代码:

```
QHBoxLayout * layout = new QHBoxLayout;           //新建水平布局管理器
layout ->addWidget(ui ->fontComboBox);         //向布局管理器中添加部件
layout ->addWidget(ui ->textEdit);
layout ->setSpacing(50);                          //设置部件间的间隔
layout ->setContentsMargins(0, 0, 50, 100);      //设置布局管理器到边界的距离
                                                    //4个参数顺序是左,上,右,下
setLayout(layout);                             //将这个布局设置为 MyWidget 类的布局
```

这里使用了 addWidget() 函数向布局管理器的末尾添加部件, 还有一个 insertWidget() 函数可以实现向指定位置添加部件, 它比前者更灵活。

2. 栅格布局管理器(QGridLayout)

栅格布局管理器 QGridLayout 类使得部件在网格中布局, 它将所有的空间分隔成一些行和列, 行和列的交叉处就形成了单元格, 然后将部件放入一个确定的单元格中。下面编写代码来使用栅格布局管理器。先往界面上拖放一个 Push Button, 然后在 mywidget.cpp 中添加头文件 #include <QGridLayout>, 再使用/* */注释掉上面添加的关于水平布局管理器的代码, 最后再添加如下代码:

```

QGridLayout * layout = new QGridLayout;
//添加部件,从第 0 行 0 列开始,占据 1 行 2 列
layout->addWidget(ui->fontComboBox, 0, 0, 1, 2);
//添加部件,从第 0 行 2 列开始,占据 1 行 1 列
layout->addWidget(ui->pushButton, 0, 2, 1, 1);
//添加部件,从第 1 行 0 列开始,占据 1 行 3 列
layout->addWidget(ui->textEdit, 1, 0, 1, 3);
setLayout(layout);

```

这里主要是设置了部件在栅格布局管理器中的位置,将 fontComboBox 部件设置为占据 1 行 2 列,而 pushButton 部件占据 1 行 1 列,这主要是为了将 fontComboBox 部件和 pushButton 部件的长度设置为 2 : 1。而这样一来,TextEdit 部件要想占满剩下的空间,就要使它的跨度为 3 列。这里需要说明,当部件加入到一个布局管理器中,然后这个布局管理器再放到一个窗口部件上时,这个布局管理器以及它包含的所有部件都会把该窗口部件自动重新定义为自己的父对象(parent),所以在创建布局管理器和其中的部件时不用指定父部件。此外,也可以直接在设计模式使用前面讲过的方法来使用栅格布局管理器。

3. 表单布局管理器(QFormLayout)

表单布局管理器 QFormLayout 类用来管理表格的输入部件及其相关的标签,将它的子部件分为两列,左边是一些标签,右边是一些输入部件,比如行编辑器或者数字选择框等。其实如果只是起到这样的布局作用,那么用 QGridLayout 就完全可以做到了,之所以添加 QFormLayout 类,是因为它有独特的功能。下面看一个例子。

先将前面在 MyWidget 类的构造函数中自己添加的代码全部注释掉,然后进入设计模式,这里使用另外一种方法来使用布局管理器。从部件栏中找到 Form Layout,将其拖入到界面上,然后双击或者在它上面右击,选择“添加窗体布局行”菜单。然后在弹出的“添加表单布局行”对话框中输入标签文字“姓名(&N)”,则下面自动填写了“标签名称”、“字段类型”和“字段名称”等,并且设置了伙伴关系。这里使用了 QLineEdit 行编辑器,当然也可以选择其他部件。而填写的标签文字中(&N)要注意括号必须是英语半角的,表明它的加速键是 Alt+N,设置伙伴关系表示当按下 Alt+N 时,光标会自动跳转到标签后面对应的行编辑器中。按下确定键便会在布局管理器中添加一个标签和一个行编辑器。按照这种方法,再添加 3 行: 性别(&S), 使用 QComboBox; 年龄(&A), 使用 QSpinBox; 邮箱(&M), 使用 QLineEdit。可以按下加速键 Alt+N, 光标就可以定位到“姓名”标签后的行编辑器中。

上面的添加表单行是在设计器中完成的,也可以在代码中使用 addRow() 函数来完成。表单布局管理器为设计填写表单的窗口提供了方便的功能,其实还有一些实用的特性,这个放到下一节再讲。表单管理器也可以像普通管理器一样使用,但是,如果不是为了设计这样的表单,一般会使用栅格布局管理器。

4. 综合使用布局管理器

前面讲到了 3 种布局管理器,真正使用时很多时候是将它们综合起来应用的。现在将前面的界面再进行设计:按下 Ctrl 键同时选中界面上的字体选择框 fontComboBox 和按钮 pushButton,然后按下 Ctrl+H 快捷键将它们放入一个水平布局管理器中(也可以从部件栏中拖入一个 Horizontal Layout,然后再将这两个部件放进去,效果是一样的)。然后再从部件栏中拖入一个 Vertical Spacer 垂直分隔符,它们是用来在部件间产生间隔的,将它放在表单布局管理器与水平布局管理器之间,如图 4-3 所示。最后单击主界面,然后按下 Ctrl+L 快捷键,让整个界面处于一个垂直布局管理器中。这时可以在右上角的对象列表中选择分隔符 Spacer,然后在属性栏中设置它的高度为 100,效果如图 4-4 所示。这时运行程序,则可以看到分隔符是不显示的。

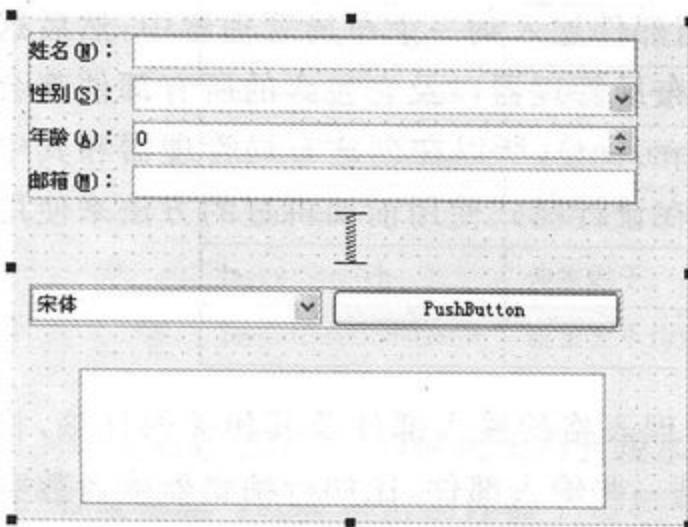


图 4-3 使用分隔符

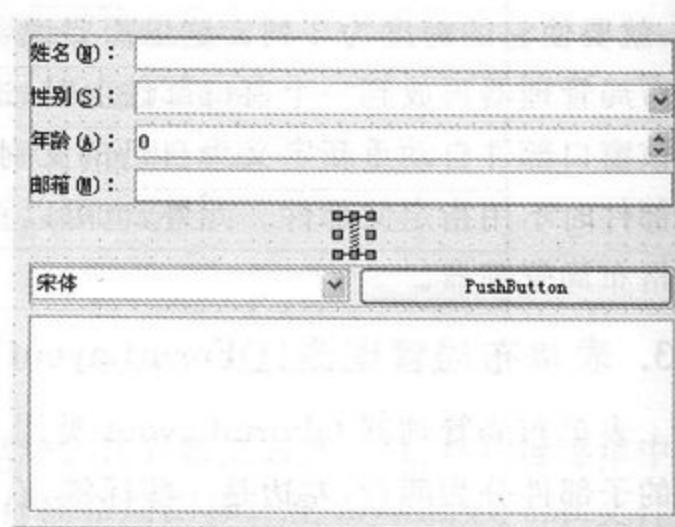


图 4-4 设置分隔符高度

这里综合使用了表单布局管理器、水平布局管理器和垂直布局管理器,其中垂直布局管理器是顶级布局管理器,因为它是主界面的布局,其他两个布局管理器都包含在它里面。如果要使用代码来实现将一个子布局管理器放入一个父布局管理器之中,可以使用父布局管理器的 addLayout() 函数。

4.1.2 设置部件大小

讲解之前要先了解两个概念:大小提示(sizeHint)和最小大小提示(minimumSizeHint)。凡是继承自 QWidget 的类都有这两个属性,其中 sizeHint 属性保存了部件的建议大小,对于不同的部件,默认拥有不同的 sizeHint;而 minimumSizeHint 保存了一个建议的最小大小。可以在程序中使用 sizeHint() 函数来获取 sizeHint 的值,使用 minimumSizeHint() 函数来获取 minimumSizeHint 的值,需要说明的是,如果使用 minimumSize() 函数设置了部件的最小大小,那么最小大小提示将会被忽略。这两个属性在使用布局时起到了很重要的作用。

下面再来看一下大小策略(sizePolicy)属性,它也是 QWidget 类的属性。这个属性保存了部件的默认布局行为,在水平和垂直两个方向分别起作用,控制着部件在布局管理器中的大小变化行为。sizePolicy 属性的所有取值如表 4-2 所列。

表 4-2 QSizePolicy 类大小策略的取值

常量	描述
QSizePolicy::Fixed	只能使用 sizeHint() 提供的值,无法伸缩
QSizePolicy::Minimum	sizeHint() 提供的大小是最小的,部件可以被拉伸
QSizePolicy::Maximum	sizeHint() 提供的是最大大小,部件可以被压缩
QSizePolicy::Preferred	sizeHint() 提供的大小是最佳大小,部件可以被压缩或拉伸
QSizePolicy::Expanding	sizeHint() 提供的是合适的大小,部件可以被压缩,不过它更倾向于被拉伸来获得更多的空间
QSizePolicy::MinimumExpanding	sizeHint() 提供的大小是最小的,部件倾向于被拉伸来获取更多的空间
QSizePolicy::Ignored	sizeHint() 的值被忽略,部件将尽可能的被拉伸来获取更多的空间

可以看到,大小策略与 sizeHint() 的值息息相关。对于布局管理器来说,大小策略对于布局效果也起到了很重要的作用。下面来看一下它们的效果。我们还在前面的程序中进行操作。单击界面上那个分隔符 Spacer,当时将其属性中的 sizeHint 的高度设置为了 100,可是实际上界面上的分隔符的高度并没有到达 100。这时可以看到它的 sizeType 属性设置为了 Expanding,将它更改为 Fixed,这样界面上的分隔符马上增高了,现在它的实际高度才是 sizeHint 的高度值。

下面再来了解一下伸缩因子(stretch factor)的概念。在前面讲垂直布局管理器时曾提到过它,其实它是用来设置部件间比例的。界面上的字体选择框和一个按钮处于一个水平布局管理器中,现在想让它们的宽度比例为 2:1,那么就可以单击对象栏中的 horizontalLayout 水平布局管理器对象,然后在它的属性栏中将 layoutStretch 属性设置为“2,1”即可。如果要在代码中进行设置,可以在使用布局管理器的 addWidget() 函数添加部件的同时,在第二个参数中指定伸缩因子。

现在再来看一下表单布局管理器中的一些属性。单击对象栏中的 formLayout 查看它的属性栏,如表 4-3 所列。对于 layoutFieldGrowthPolicy 属性,这里选择 ExpandingFieldsGrow 选项,这样性别和年龄两个输入框就没有那么宽了,更符合美观要求。然后将界面中的“邮箱”标签更改为“邮箱地址”,在 layoutLabelAlignment 属性中选择 AlignRight。

表 4-3 表单布局管理器相关属性说明

属性	说 明	值	说 明	
layoutFieldGrowthPolicy	指定部件的大小变化方式	AllNonFixedFieldsGrow	所有的部件都被拉伸,这是默认值	
		FieldsStayAtSizeHint	所有的部件都使用 sizeHint() 提供的大小	
		ExpandingFieldsGrow	大小策略为 Expanding 的部件会被拉伸	
layoutRowWrapPolicy	设置是否换行,如果需要换行,则是将输入部件放到相应的标签下面	DontWrapRows	不换行,这是默认值	
		WrapLongRows	将较长的行进行换行	
		WrapAllRows	将所有行都换行,这样所有的输入部件都会放置在相应的标签下面	
layoutLabelAlignment	设置标签的对其方式,分为水平方向和垂直方向	水平方向	AlignLeft	左对齐
			AlignRight	右对齐
			AlignHCenter	水平居中对齐
			AlignJustify	两端对齐
		垂直方向	AlignTop	向上对齐
			AlignBottom	向下对齐
			AlignVCenter	垂直居中对齐
layoutFormAlignment	设置部件在表单中的对齐方式	同 layoutLabelAlignment 属性	同 layoutLabelAlignment 属性	

下面来看一下 QWidget 类及其子类部件的设置大小的相关属性。单击主界面,看一下其属性栏,在最开始便是几个与大小有关的属性,如图 4-5 所示。



图 4-5 QWidget 大小属性

这里的高度与宽度属性是现在界面的大小; sizePolicy 属性可以设置大小策略以及伸缩因子; minimumSize 属性用来设置最小大小,改为 200×150; maximumSize 属性设置最大大小,设置为 500×350; sizeIncrement 属性和 baseSize 属性是设置窗口改变大小的,一般不用设置。

layoutSizeConstraint 属性是用来约束窗口的大小的,也就是说,这个只对顶级布局管理器有用,因为它只对窗口有用,对其他子部件没有效果。它的几个值及其含义如表 4-4 所列。这个属性的默认值是

SetDefaultConstraint。可以将界面的顶级布局管理器设置为 SetFixedSize，这样运行程序时可以看到窗口就无法再变化大小了。

表 4-4 QLayout 类的大小约束属性的取值

常量	描述
QLayout::SetDefaultConstraint	主窗口大小设置为 minimumSize() 的值,除非该部件已经有一个最小大小
QLayout::SetFixedSize	主窗口大小设置为 sizeHint() 的值,它无法改变大小
QLayout::SetMinimumSize	主窗口的最小大小设置为 minimumSize() 的值,它无法再缩小
QLayout::SetMaximumSize	主窗口的最大大小设置为 maximumSize() 的值,它无法再放大
QLayout::SetMinAndMaxSize	主窗口的最小大小设置为 minimumSize() 的值,最大大小设置为 maximumSize() 的值
QLayout::SetNoConstraint	部件不被约束

4.1.3 可扩展窗口

一个窗口可能有很多选项是扩充的,只有在必要的时候才显示出来,这时就可以使用一个按钮,用来隐藏或者显示多余的内容,就是所谓的可扩展窗口。要实现可扩展窗口,就要得力于布局管理器的特性,那就是当子部件隐藏时,布局管理器自动缩小,当子部件重新显示时,布局管理器再次放大。下面看一个具体的例子。

依然在前面的程序中进行更改。首先将界面上 pushButton 的显示文本更改为“显示可扩展窗口”,然后在其属性栏中选中 checkable 选项。然后转到它的 toggled(bool) 信号的槽,更改如下:

```
void MyWidget::on_pushButton_toggled(bool checked) //显隐窗口按钮
{
    ui->textEdit->setVisible(checked);           //设置文本编辑器的显示和隐藏
    if(checked) ui->pushButton->setText(tr("隐藏可扩展窗口"));
    else ui->pushButton->setText(tr("显示可扩展窗口"));
}
```

使用按钮的按下与否两种状态来设置文本编辑器是否显示,并且相应更改按钮的文本。为了让文本编辑器一开始是隐藏的,还要在 MyWidget 类的构造函数中添加一行代码:

```
ui->textEdit->hide(); //让文本编辑器隐藏,也可以使用 setVisible(false) 函数
```

运行程序。可扩展窗口隐藏时效果如图 4-6 所示,可扩展窗口显示时如图 4-7 所示。也可以参考 Qt 自带的示例程序 Extension Dialog,它在 Dialogs 分类中。



图 4-6 可扩展窗口隐藏效果

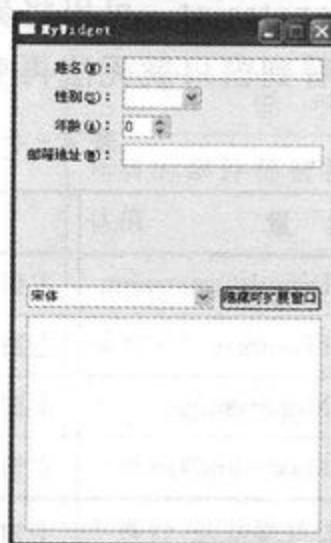


图 4-7 可扩展窗口显示效果

4.1.4 分裂器

分裂器 QSplitter 类提供了一个分裂器部件,和 QVBoxLayout 很相似,可以完成布局管理器的功能,但是包含在它里面的部件,默认是可以随着分裂器的大小变化而进行相应大小变化的。比如一个按钮放在布局管理器中,它的垂直方向默认是不会被拉伸的,但是放到分裂器中就可以被拉伸。还有一个不同就是,布局管理器是继承自 QObject 类的,而分裂器却是继承自 QFrame 类,QFrame 类又是继承自 QWidget 类,也就是说,分裂器拥有 QWidget 类的特性,它是可见的,而且可以像 QFrame 一样设置边框。下面看一个例子。

(项目源码路径: src\04\4-2\mySplitter)新建 Qt Gui 应用,项目名称为 mySplitter,类名 MyWidget,基类选择 QWidget。建好项目后打开 mywidget.ui 文件,然后往上面拖入 4 个 Push Button,同时选中这 4 个按钮然后右击,选择“布局→使用分裂器水平布局”菜单项,将这 4 个按钮放到一个分裂器中。将分裂器拉大点,并在属性栏中设置其 frameShape 为 Box, frameShadow 为 Raised, lineWidth 为 5。此时运行程序,效果如图 4-8 所示。

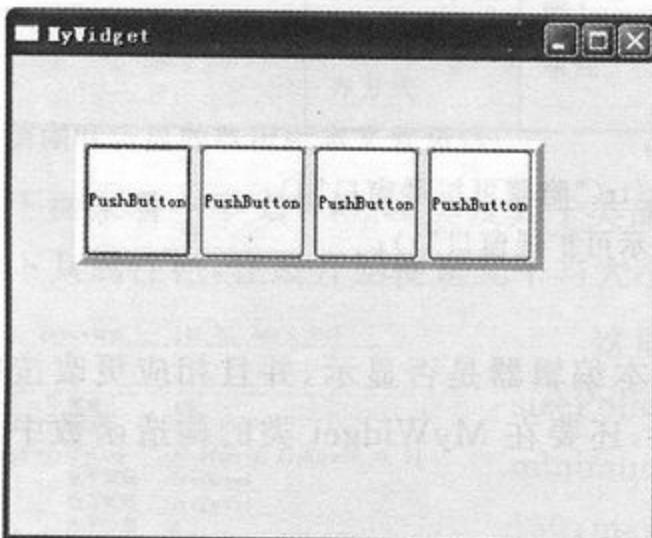


图 4-8 分裂器运行效果

4.2 设置伙伴

讲述表单布局管理器时提到了设置一个标签和一个部件的伙伴关系。其实,伙伴(buddy)是在 QLabel 类中提出的一个概念。因为一个标签经常用作一个交互式部件

的说明,就像在讲表单布局管理器时看到的那样,一个 `lineEdit` 部件前面有一个标签说明这个 `lineEdit` 的作用。为了方便定位,`QLabel` 提供了一个有用的机制,那就是提供了助记符来设置键盘焦点到对应的部件上,这个部件就叫这个 `QLabel` 的伙伴。其中助记符就是我们所说的加速键。使用英文标签时,在字符串的一个字母前面添加“`&`”符号,那么就可以指定这个标签的加速键是 `Alt` 加上这个字母,而对于中文,需要在小括号中指定加速键字母,这个前面已经见过多次了。Qt 设计器中也提供了伙伴设计模式,下面看一个例子。

(项目源码路径: `src\04\4-3\myBuddy`)新建 Qt Gui 应用,项目名称为 `myBuddy`,类名 `MyWidget`,基类选择 `QWidget`。建好项目后打开 `mywidget.ui` 文件,然后往界面上拖放 4 个标签 `Label`,再在标签后面依次放上 `PushButton`、`CheckBox`、`LineEdit` 和 `SpinBox`。然后将 `PushButton` 前面的标签的文本改为“`&Button:`”; `CheckBox` 前面的标签文本改为“`C&heckBox:`”; `LineEdit` 前面的标签的文本改为“`行编辑器 (&L):`”; `SpinBox` 前面的标签文本改为“`数字选择框 (&N):`”。单击下设计器顶部栏中的编辑伙伴图标 进入伙伴设计模式,分别将各个标签与它们后面的部件连起来,如图 4-9 所示。然后按下 `F3` 回到正常编辑模式,可以看到所有的 `&` 符号都没有了,只是在字母下面多了一个横杠,表示这个标签的加速键就是 `Alt` 加这个字母。

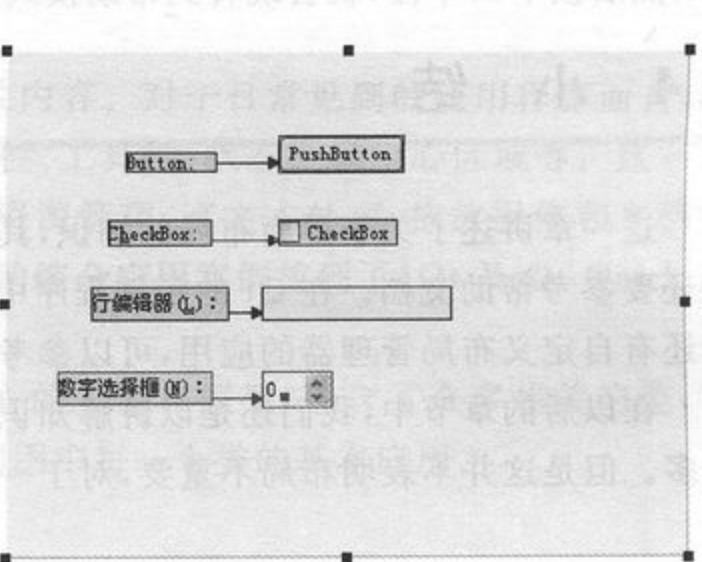


图 4-9 编辑伙伴模式

现在可以运行程序,使用加速键测试效果。如果要在代码中设置伙伴关系,只需要使用 `QLabel` 的 `setBuddy()` 函数就可以了。对于本小节内容可以在帮助索引中查看 Qt Designer's Buddy Editing Mode 关键字。

4.3 设置 Tab 键顺序

对于一个应用程序,我们有时总希望使用 Tab 键来将焦点从一个部件移动到下一个部件。在设计模式,设计器提供了 Tab 键的设置功能。上面程序的设计模式中,单击上边栏的编辑 Tab 顺序 按钮进入编辑 Tab 键顺序模式,这时已经显示出了各个部件的 Tab 键顺序,只需要单击这些数字即可以更改。需要说明,当程序启动时,焦点会在 Tab 键顺序为 1 的部件上。这里进行的设置等价于在 `MyWidget` 类的构造函数中使用如下代码:

```

setTabOrder(ui->lineEdit,ui->spinBox);      //lineEdit 在 spinBox 前面
setTabOrder(ui->spinBox,ui->pushButton);      //spinBox 在 pushButton 前面
  
```

```
setTabOrder(ui->pushButton, ui->checkBox); //pushButton 在 checkBox 前面
```

关于在设计器中设置 Tab 键顺序,读者也可以在帮助索引中查看 Qt Designer's Tab Order Editing Mode 关键字。

最后再介绍一下 Qt Creator 中定位器的应用。第 1 章中介绍 Qt Creator 时已经提到了定位器,它位于主界面的左下方。使用定位器可以很方便地打开指定文件、定位到文档的指定行、打开一个特定的帮助文档、进行项目中函数的查找等。更多的功能可以在帮助索引中查看 Searching With the Locator 关键字。

下面举两个简单的例子。按下 Ctrl+K 快捷键打开定位器,这时输入“l 8”(英文字母 l 和一个空格,然后是数字 8),然后按回车键,那么就会跳转到编辑模式的当前打开文档的第 8 行。再按下 Ctrl+K 快捷键,输入“? qpu”,这时已经查找到了 QPushButton,然后按回车键,就会跳转到帮助模式中,并打开 QPushButton 类的帮助文档。

4.4 小结

这一章讲述了关于界面布局的知识,其实只是提到了最基本的应用,更多的布局知识还要参考帮助文档。在 Qt 的示例程序中 Layouts 分类下是几个关于布局的例子,其中还有自定义布局管理器的应用,可以参考一下。

在以后的章节中,我们还是以讲解知识点为主,所以对于界面的布局操作不会涉及太多。但是这并不表明布局不重要,对于一个软件,良好的布局是必须的。

第 5 章

应用程序主窗口

这一章开始接触应用程序主窗口的相关内容。对于日常见到的应用程序而言，许多都是基于主窗口的，主窗口中包含了菜单栏、工具栏、状态栏和中心区域等。这一章会详细介绍主窗口的每一个部分，还会涉及资源管理、富文本处理、拖放操作和文档打印等相关内容。重点是讲解知识点，而相关的综合应用实例放到了《Qt 及 Qt Quick 开发实战精解》一书中。

Qt 中提供了以 QMainWindow 类为核心的主窗口框架，包含了众多相关的类，它们的继承关系如图 5-1 所示，本章会讲解到图中每一个类的基本应用。

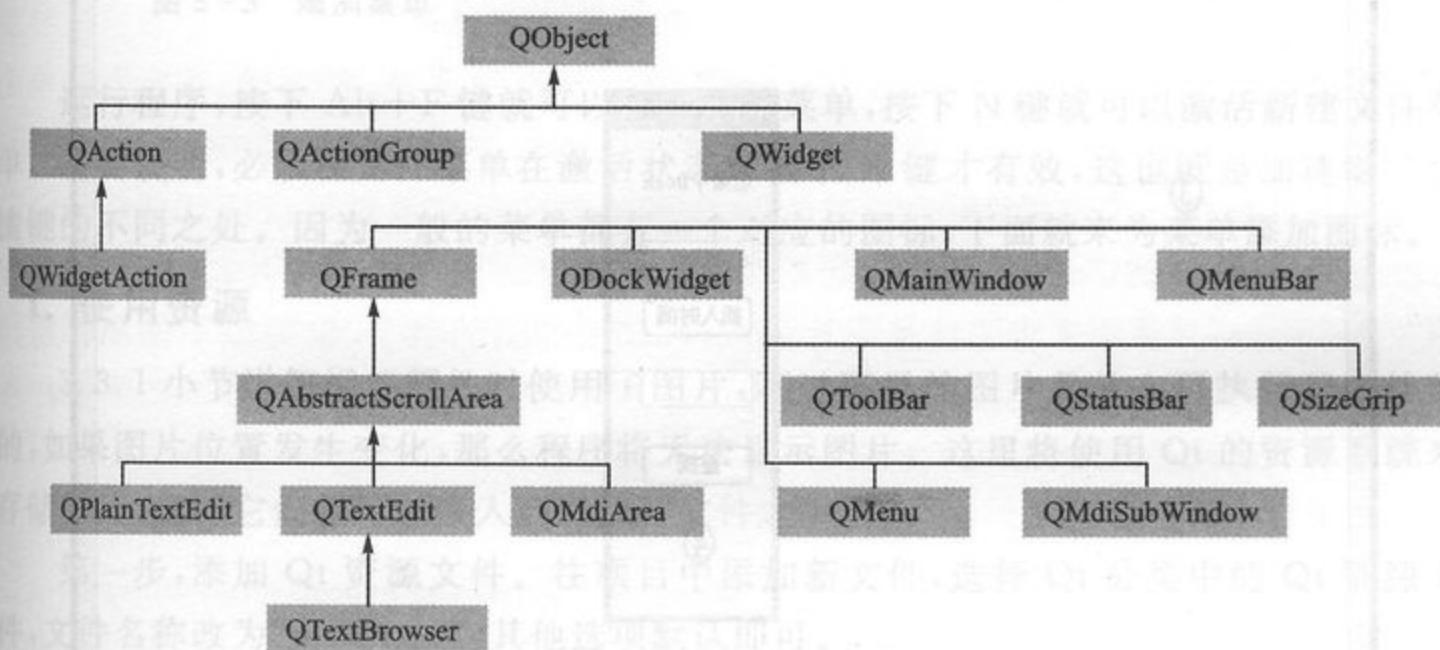


图 5-1 主窗口相关类关系图

5.1 主窗口框架

主窗口为建立应用程序用户界面提供了一个框架，Qt 提供了 QMainWindow 和其他一些相关的类共同主窗口的管理。QMainWindow 类拥有自己的布局，如图 5-2 所

示,它包含以下组件:

① 菜单栏(QMenuBar)。菜单栏包含了一个下拉菜单项的列表,这些菜单项由 QAction 动作类实现。菜单栏位于主窗口的顶部,一个主窗口只能有一个菜单栏。

② 工具栏(QToolBar)。工具栏一般用于显示一些常用的菜单项目,也可以插入其他窗口部件,并且工具栏是可以移动的。一个主窗口可以拥有多个工具栏。

③ 中心部件(Central Widget)。在主窗口的中心区域可以放入一个窗口部件作为中心部件,是应用程序的主要功能实现区域。一个主窗口只能拥有一个中心部件。

④ Dock 部件(QDockWidget)。Dock 部件常被称为停靠窗口,因为可以停靠在中心部件的四周。它用来放置一些部件来实现一些功能,就像个工具箱。一个主窗口可以拥有多个 Dock 部件。

⑤ 状态栏(QStatusBar)。状态栏用于显示程序的一些状态信息,在主窗口的最底部。一个主窗口只能拥有一个状态栏。

本节知识可以在帮助索引中查看 Application Main Window 关键字,其中列出了所有与创建主窗口应用程序相关的类,而查看 The Qt 4 Main Window Classes 关键字,可以看到主窗口相关类的应用介绍和主窗口类在 Qt 4 中相对于在 Qt 3 中的更改。

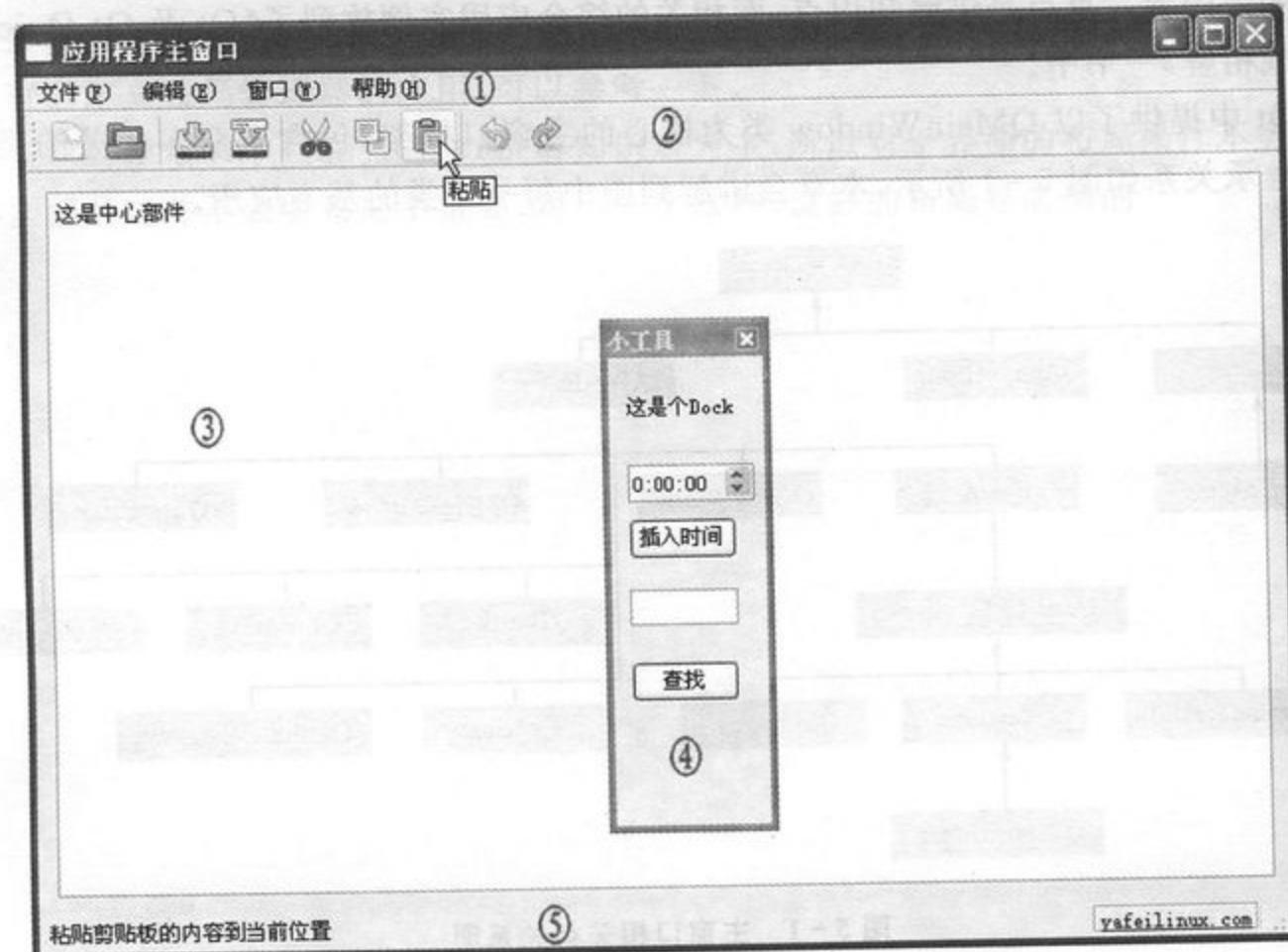


图 5-2 应用程序主窗口界面

5.1.1 菜单栏和工具栏

下面先来看一个例子。(项目源码路径: src\05\5-1\myMainWindow)新建 Qt

Gui应用，项目名称 myMainWindow，类名默认为 MainWindow，基类默认为 QMainWindow 不做改动。建立好项目后，在文件列表中双击 mainwindow.ui 文件进入设计模式，这时在设计区域出现的便是主窗口界面。下面来添加菜单。双击左上角的“在这里输入”，修改为“文件(&F)”，这里要使用英文半角的括号，“&F”被称为加速键，表明程序运行时，可以按下 Alt+F 键来激活该菜单。修改完成后，按回车键，并在弹出的下拉菜单中将第一项改为“新建文件(&N)”并按回车键，效果如图 5-3 所示。这时可以看到在下面的 Action 编辑器中已经有了“新建文件”动作，如图 5-4 所示。单击该动作，将其拖入菜单栏下面的工具栏中，如图 5-5 所示。关于在设计器中创建主窗口，也可以在帮助索引中查看 Creating Main Windows in Qt Designer 关键字。

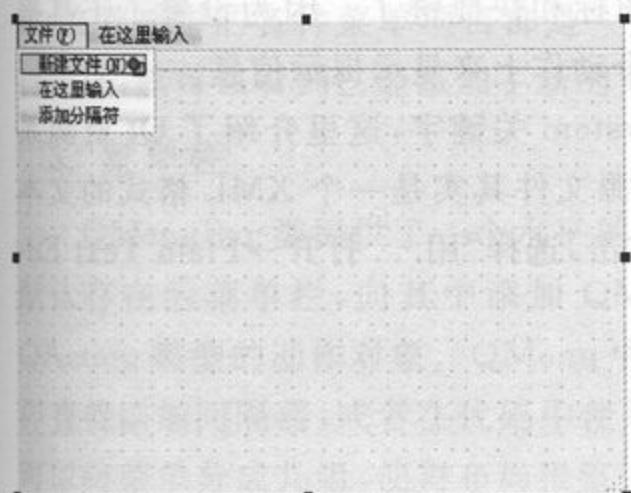


图 5-3 添加菜单

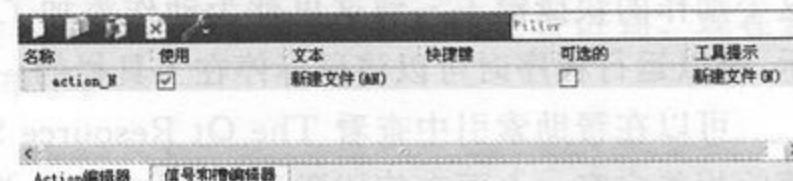


图 5-4 Action 编辑器

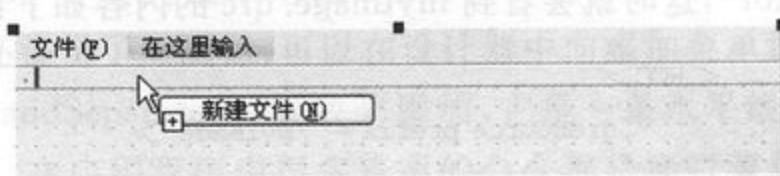


图 5-5 向工具栏中拖入动作

运行程序，按下 Alt+F 键就可以打开文件菜单，按下 N 键就可以激活新建文件菜单。需要说明，必须在文件菜单在激活状态时按下 N 键才有效，这也就是加速键与快捷键的不同之处。因为一般的菜单都有一个对应的图标，下面就来为菜单添加图标。

1. 使用资源

3.3.1 小节讲解标签部件时使用了图片，不过那里的图片是放在可执行程序外部的，如果图片位置发生变化，那么程序将无法显示图片。这里将使用 Qt 的资源系统来存储图片，这样它们就可以嵌入到可执行文件之中了。

第一步，添加 Qt 资源文件。往项目中添加新文件，选择 Qt 分类中的 Qt 资源文件，文件名称改为“myImage”，其他选项默认即可。

第二步，添加资源。建立好资源文件后会默认进入资源管理界面，就是新建的 myImage.qrc 文件中。现在先到项目文件夹 myMainWindow 中新建一个名为 images 的文件夹，并在其中放入两张图标图片，比如这里放入了一个 new.png 和一个“open.png”图片。（注意：Qt 的资源系统要求资源文件必须放在与 qrc 文件同级或子级目录下，如果放在其他地方，添加资源时会提示将文件复制到有效的位置。）

回到 Qt Creator 中，在资源管理界面上单击“添加”按钮，选择“添加前缀”，然后将属性栏中的前缀改为“/myImage”，再单击“添加”按钮选择“添加文件”，在弹出的对话框

中进入到前面新建的 images 文件夹，选中那两张图片单击“打开”即可。这时 myImage.qrc 文件中就出现了添加的图片列表。最后按下 Ctrl+S 快捷键保存对文件的修改（注意：这一点很重要，如果没有保存，在下面使用图片时将看不到图片）。

第三步，使用图片。先使用 Ctrl+Tab 快捷键转到 mainwindow.ui 文件，回到设计模式。在 Action 编辑器中双击“新建文件”动作，这时会弹出编辑动作对话框。在其中将对象名称改为“action_New”，工具提示改为“新建文件”，然后单击图标后面的按钮进入选择资源界面。第一次进入该界面还没有显示可用的资源，需要单击左上角的重新加载绿色箭头图标，这时图片资源就显示出来了。这里选择 new.png 图片，然后单击“确定”按钮。最后在快捷键后面的输入栏上单击并按下 Ctrl+N，就可以将它设为这个动作的快捷键了。到这里就为动作添加了图标和快捷键了。因为设置了工具提示，所以运行程序时可以将鼠标停在工具栏的一个动作上来显示提示信息。

可以在帮助索引中查看 The Qt Resource System 关键字，这里介绍了 Qt 资源系统相关内容。上面在使用资源时添加的 qrc 资源文件其实是一个 XML 格式的文本文件，现在进入编辑模式，在 myImage.qrc 文件右击，选择“用... 打开→Plain Text Editor”，这时就会看到 myImage.qrc 的内容如下：

```
<RCC>
    <qresource prefix="/myImage">
        <file>images/new.png</file>
        <file>images/open.png</file>
    </qresource>
</RCC>
```

这里指明了文件类型为 RCC，表明是 Qt 资源文件。然后是资源前缀，下面罗列了添加的图片路径。如果在编写代码时使用 new.png 图片，那么就可以将其路径指定为“:/myImage/images/new.png”，这样就表明在使用资源文件中的图片，其中添加的前缀“/myImage”只是用来表明这是图片资源，可以改为别的名字，也可以不使用。前面在 Resource Editor 中进行图片添加操作，这种方式比较方便明了，也可以按照这种格式使用手写代码来添加图片。

往项目中添加资源文件时会自动往工程文件 myMainWindow.pro 中添加代码：

```
RESOURCES += \myImage.qrc
```

这表明项目中使用了资源文件 myImage.qrc。在前面的过程中这是自动生成的，但是如果是自己添加的已有资源文件，要想在项目中使用，就要手动添加这行代码。其实资源可以是任意类型的，不只是图片文件，不过，资源只能是只读的，不能对其进行修改。而且，编译时会对加入的资源自动压缩，这也就是为什么有时生成的 release 版本可执行文件比添加进去的资源还要小的原因。

2. 编写代码方式添加菜单

上面在设计器中添加了文件菜单，然后添加了新建文件子菜单，其实这些都可以使

用代码来实现,下面使用代码来添加一个菜单。首先在 main.cpp 文件中添加代码使其可以支持中文,然后在 mainwindow.cpp 文件的 MainWindow 类构造函数中添加代码:

```
QMenu * editMenu = ui ->menuBar ->addMenu(tr("编辑(&E)")); //添加编辑菜单
 QAction * action_Open = editMenu ->addAction(QIcon(":/myImage/images/open.png"), tr(
    "打开文件(&O)")); //添加打开菜单
 action_Open ->setShortcut(QKeySequence("Ctrl + O")); //设置快捷键
 ui ->mainToolBar ->addAction(action_Open); //在工具栏中添加动作
```

这里使用 ui->menuBar 来获取了 QMainWindow 的菜单栏,使用 ui->mainToolBar 来获取了工具栏,然后分别使用相应的函数来添加菜单和动作,就像前面提到过的,在菜单中的各种菜单项目都是一个 QAction 类对象,这个后面还会讲到。现在运行程序,就可以看到已经添加了新的菜单了。

3. 菜单栏

QMenuBar 类提供了一个水平的菜单栏,在 QMainWindow 中可以直接获取它的默认存在的菜单栏,向其中添加 QMenu 类型的菜单对象,然后向弹出菜单中添加 QAction 类型的动作对象。QMenu 中还提供了间隔器,可以在设计器中向添加菜单那样直接添加间隔器,或者在代码中使用 addSeparator() 函数来添加,它是一条水平线,可以将菜单分成几组,使得布局很整齐。在应用程序中很多普通的命令都是通过菜单来实现的,而我们也希望能将这些菜单命令放到工具栏中,以方便使用。QAction 就是这样一种命令动作,可以同时放在菜单和工具栏中。一个 QAction 动作包含了图标、菜单显示文本、快捷键、状态栏显示文本、“What's This?”显示文本及工具提示文本。这些都可以在构建 QAction 类对象时在构造函数中指定。另外还可以设置 QAction 的 checkable 属性,如果指定这个动作的 checkable 为 true,那么当选中这个菜单时就会在它的前面显示“√”之类的表示选中状态的符号;如果该菜单有图标,那么就会用线框将图标围住,用来表示该动作被选中了。

下面再介绍一个动作组 QActionGroup 类。它可以包含一组动作 QAction,支持这组动作中是否只能有一个动作处于选中状态,这对于互斥型动作很有用。在上面程序的 MainWindow 类构造函数中继续添加如下代码:

```
QActionGroup * group = new QActionGroup(this); //建立动作组
 QAction * action_L = group ->addAction(tr("左对齐(&L)")); //向动作组中添加动作
 action_L ->setCheckable(true); //设置动作 checkable 属性为 true
 QAction * action_R = group ->addAction(tr("右对齐(&R)"));
 action_R ->setCheckable(true);
 QAction * action_C = group ->addAction(tr("居中(&C)"));
 action_C ->setCheckable(true);
 action_L ->setChecked(true); //最后指定 action_L 为选中状态
 editMenu ->addSeparator(); //向菜单中添加间隔器
 editMenu ->addAction(action_L); //向菜单中添加动作
```

```
editMenu ->addAction(action_R);
editMenu ->addAction(action_C);
```

这里让“左对齐”、“右对齐”和“居中”3个动作处于一个动作组中，然后设置“左对齐”动作为默认选中状态。

4. 工具栏

工具栏 QToolBar 类提供了一个包含了一组控件的可以移动的面板。在上面已经看到可以将 QAction 对象添加到工具栏中，它默认只是显示一个动作的图标，这个可以在 QToolBar 的属性栏中更改。在设计器中查看 QToolBar 的属性栏，其中 toolButtonStyle 属性设置图标和相应文本的显示及其相对位置等；movable 属性设置状态栏是否可以移动；allowedArea 设置允许停靠的位置；iconSize 属性设置图标的大小；floatable 属性设置是否可以悬浮。

工具栏中除了可以添加动作外，还可以添加其他的窗口部件，下面来看一个例子，在前面的程序中的 mainwindow.cpp 文件中添加头文件：

```
#include <QToolButton>
#include <QSpinBox>
```

然后在构造函数中继续添加如下代码：

```
QToolButton * toolBtn = new QToolButton(this); //创建 QToolButton
toolBtn ->setText(tr("颜色"));
QMenu * colorMenu = new QMenu(this); //创建一个菜单
colorMenu ->addAction(tr("红色"));
colorMenu ->addAction(tr("绿色"));
toolBtn ->setMenu(colorMenu); //添加菜单
toolBtn ->setPopupMode(QToolButton::MenuButtonPopup); //设置弹出模式
ui ->mainToolBar ->addWidget(toolBtn); //向工具栏添加 QToolButton 按钮

QSpinBox * spinBox = new QSpinBox(this); //创建 QSpinBox
ui ->mainToolBar ->addWidget(spinBox); //向工具栏添加 QSpinBox 部件
```

这里创建了一个 QToolButton 类对象，并为它添加了一个弹出菜单，这里设置了弹出方式是在按钮旁边有一个向下的小箭头，可以按下这个箭头弹出菜单，而默认的弹出方式是按下按钮一段时间才弹出菜单。最后将它添加到了工具栏中。下面又在工具栏中添加了一个 QSpinBox 部件，可以看到往工具栏中添加部件可以使用 addWidget() 函数。

这里还要再说明一下 QToolButton 类。其实，往工具栏中添加一个 QAction 类对象时就会自动创建了一个 QToolButton，所以说工具栏上的动作就是一个 QToolButton，这就是为什么属性栏中会有 toolButtonStyle 属性的原因了。

5.1.2 中心部件

在主窗口的中心区域可以放置一个中心部件，它一般是一个编辑器或者浏览器。

这里支持单文档部件,也支持多文档部件。一般的,我们会在这里放置一个部件,然后使用布局管理器使其充满整个中心区域,并可以随着窗口的大小变化而改变大小。下面在前面的程序中添加中心部件。在设计模式中,往中心区域拖入一个 Text Edit,然后单击界面,按下 Ctrl+G 使其处于一个栅格布局中。现在可以运行程序查看效果。QTextEdit 是一个高级的 WYSIWYG(所见即所得)浏览器和编辑器,支持富文本的处理,为用户提供了强大的文本编辑功能。而与 QTextEdit 对应的是 QPlainTextEdit 类,它提供了一个纯文本编辑器,这个类与 QTextEdit 类的很多功能都很相似,只不过无法处理富文本。还有一个 QTextBrowser 类,是一个富文本浏览器,可以看作 QTextEdit 的只读模式。因为这 3 个类的用法大同小异,所以在以后的内容中只讲解 QTextEdit 类。

中心区域还可以使用多文档部件。Qt 中的 QMdiArea 部件就是用来提供一个可以显示 MDI (Multiple Document Interface) 多文档界面的区域,它代替了以前的 QWorkspace 类,用来有效地管理多个窗口。QMdiArea 中的子窗口由 QMdiSubWindow 类提供,这个类有自己的布局,包含一个标题栏和一个中心区域,可以向它的中心区域添加部件。

下面更改前面的程序,在设计模式将前面添加的 Text Edit 部件删除,然后拖入一个 MdiArea 部件。在 Action 编辑器中的“新建文件”动作上右击,在弹出的菜单中选择“转到槽...”,然后在弹出的对话框中选择 triggered() 触发信号,单击“确定”按钮后便会转到 mainwindow.cpp 文件中的该信号的槽的定义处,更改如下:

```
void MainWindow::on_action_New_triggered()
{
    //新建文本编辑器部件
    QTextEdit * edit = new QTextEdit(this);
    //使用 QMdiArea 类的 addSubWindow() 函数创建子窗口,以文本编辑器为中心部件
    QMdiSubWindow * child = ui->mdiArea->addSubWindow(edit);
    child->setWindowTitle(tr("多文档编辑器子窗口"));
    child->show();
}
```

这里需要添加 #include <QTextEdit> 和 #include <QMdiSubWindow> 头文件。在新建文件菜单动作的触发信号槽 on_action_New_triggered() 中创建了多文档区域的子窗口。这时运行程序,然后按下工具栏上的新建文件动作图标,每按一次,就会生成一个子窗口。

5.1.3 Dock 部件

QDockWidget 类提供了这样一个部件,它可以停靠在 QMainWindow 中,也可以悬浮起来作为桌面顶级窗口,称它为 Dock 部件或者停靠窗口。Dock 部件一般用于存放一些其他部件来实现特殊功能,就像一个工具箱。它在主窗口中可以停靠在中心部

件的四周,也可以悬浮起来,被拖动到任意的地方,还可以被关闭或隐藏起来。一个 Dock 部件包含一个标题栏和一个内容区域,可以向 Dock 部件中放入任何部件。

在设计模式中向中心区域拖入一个 Dock Widget 部件,然后再向 Dock 中随意拖入几个部件,比如这里拖入一个 Push Button 和 Font Combo Box。然后在 dockWidget 的属性栏中更改其 windowTitle 为“工具箱”,另外还可以设置它的 features 属性,包含是否可以关闭、移动和悬浮等;还有 allowedArea 属性,用来设置可以停靠的区域。现在运行程序,效果如图 5-6 所示,然后拖动它,使其悬浮起来,如图 5-7 所示。

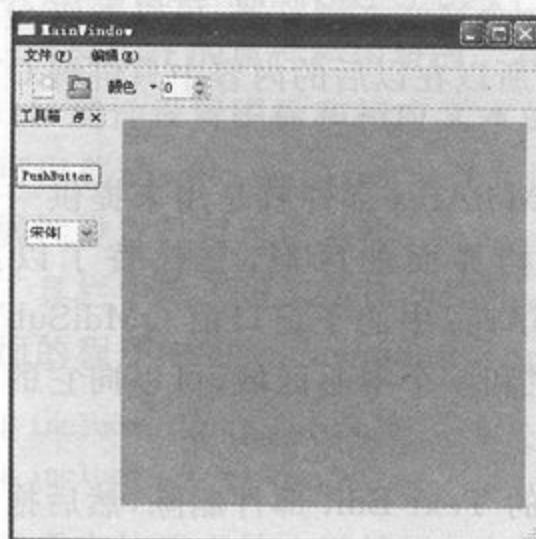


图 5-6 Dock 部件运行效果

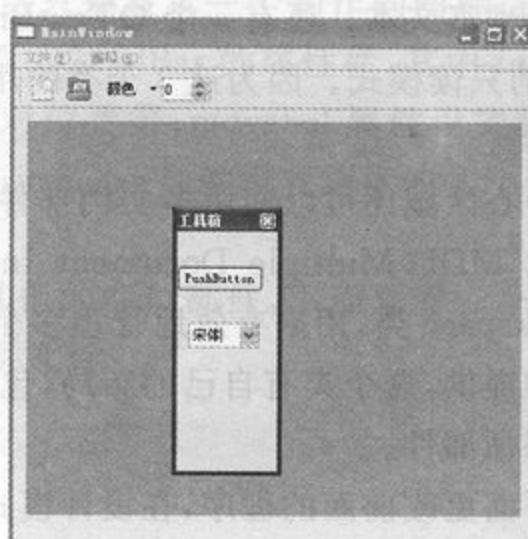


图 5-7 Dock 部件浮动效果

下面在文件菜单中添加“显示 Dock”菜单,然后在 Action 编辑器中进入“显示 Dock”菜单动作的触发信号 triggered() 的槽函数,更改如下:

```
void MainWindow::on_action_Dock_triggered()
{
    ui->dockWidget->show();
}
```

运行程序时关闭了 Dock 部件后,按下该菜单,就可以重新显示 Dock 了。

5.1.4 状态栏

QStatusBar 类提供了一个水平条,用来显示状态信息。QMainWindow 中默认提供了一个状态栏。状态信息可以被分为 3 类:临时信息,如一般的提示信息;正常信息,如显示页数和行号;永久信息,如显示版本号或者日期。可以使用 showMessage() 函数来显示一个临时消息,它会出现在状态栏的最左边。一般用 addWidget() 函数添加一个 QLabel 到状态栏上用于显示正常信息,它会生成到状态栏的最左边,可能会被临时消息所掩盖。如果要显示永久信息,要使用 addPermanentWidget() 函数来添加一个如 QLabel 一样的可以显示信息的部件,它会生成在状态栏的最右端,不会被临时消息所掩盖。

在状态栏的最右端还有一个 QSizeGrip 部件,用来调整窗口的大小,可以使用 set-

SizeGripEnabled()函数来禁用它。

因为在目前的设计器中还不支持直接向状态栏中拖放部件，所以需要使用代码来生成。在mainwindow.cpp文件中的构造函数里继续添加代码：

```
//显示临时消息,显示2000毫秒即2秒钟
ui->statusBar->showMessage(tr("欢迎使用多文档编辑器"), 2000);
//创建标签,设置标签样式并显示信息,然后将其以永久部件的形式添加到状态栏
QLabel *permanent = new QLabel(this);
permanent->setFrameStyle(QFrame::Box | QFrame::Sunken);
permanent->setText("www.yafeilinux.com");
ui->statusBar->addPermanentWidget(permanent);
```

此时运行程序可以发现“欢迎使用多文档编辑器”字符串在显示一会儿后就自动消失了，而“www.yafeilinux.com”一直显示在状态栏最右端。

到这里，主窗口的几个主要组成部分就讲完了。可以看到，一个 QMainWindow 类中默认提供了一个菜单栏、一个工具栏、一个中心区域和一个状态栏，而 Dock 部件是需要自己添加的。其实在设计模式中相关部件上右击就可以删除菜单栏、工具栏和状态栏，也可以添加新的工具栏，当然这些操作也可以使用代码实现。

5.1.5 自定义菜单

前面已经看到，可以在工具栏中添加任意的部件，那么菜单中呢，是否也可以使用其他部件呢？当然可以，Qt 中的 QWidgetAction 类就提供了这样的功能。为了实现自定义菜单，需要新建一个类，它继承自 QWidgetAction 类，并且在其中重新实现 createWidget() 函数。下面的例子中实现了这样一个菜单：它包含一个标签和一个行编辑器，可以在行编辑器中输入字符串，然后按下回车键，就可以自动将字符串输入到中心部件文本编辑器中。

(项目源码路径：src\05\5-2\myAction)新建 Qt Gui 应用，项目名称 myAction，类名默认为 MainWindow，基类默认为 QMainWindow 不做改动。建好项目后往项目中添加新文件，模板选择 C++ 类，类名 MyAction，基类为 QWidgetAction，类型信息选择继承自 QObject，因为 QWidgetAction 类继承自 QAction 类，而 QAction 类又继承自 QObject 类，所以这里选择了“继承自 QObject”。

在 myaction.h 文件中添加代码，完成后 myaction.h 文件的部分内容如下：

```
#include <QWidgetAction>
class QLineEdit; //前置声明
class MyAction : public QWidgetAction
{
    Q_OBJECT
public:
    explicit MyAction(QObject *parent = 0);
protected:
```

```

//声明函数,该函数是 QWidgetAction 类中的虚函数
QWidget * createWidget(QWidget * parent);

signals:
    //新建信号,用于在按下回车键时,将行编辑器中的内容发射出去
    void getText(const QString &string);

private slots:
    //新建槽,它用来与行编辑器的按下回车键信号关联
    void sendText();

private:
    //定义行编辑器对象的指针
    QLineEdit * lineEdit;
};


```

这里对手动添加的内容都添加了注释。下面再到 myaction.cpp 添加代码。先添加头文件包含：

```

#include <QLineEdit>
#include <QSplitter>
#include <QLabel>

```

然后在 MyAction 类的构造函数中添加如下代码：

```

lineEdit = new QLineEdit;
//将行编辑器的按下回车键信号与我们的发送文本槽函数关联
connect(lineEdit, SIGNAL(returnPressed()), this, SLOT(sendText()));

```

再添加 createWidget() 函数的定义：

```

QWidget * MyAction::createWidget(QWidget * parent) //创建部件
{
    //这里使用 inherits() 函数判断父部件是否是菜单或者工具栏
    //如果是,则创建该父部件的子部件,并且返回子部件
    //如果不是,则直接返回 0
    if(parent -> inherits("QMenu") || parent -> inherits("QToolBar")){
        QSplitter * splitter = new QSplitter(parent);
        QLabel * label = new QLabel;
        label -> setText(tr("插入文本："));
        splitter -> addWidget(label);
        splitter -> addWidget(lineEdit);
        return splitter;
    }
    return 0;
}

```

当使用该类的对象并将其添加到一个部件上时,就会自动调用 createWidget() 函数,这里先判断这个部件是否是一个菜单或者工具栏,如果不是,直接返回 0,不处理。

如果是，就以该部件为父窗口，创建了一个分裂器，并在其中添加一个标签和我们的行编辑器，最后将这个分裂器返回。

下面是 sendText() 函数的定义：

```
void MyAction::sendText()
{
    emit getText(lineEdit->text()); //发射信号，将行编辑器中的内容发射出去
    lineEdit->clear();           //清空行编辑器中的内容
}
```

每当在行编辑器中输入文本并按下回车键时就会激发 returnPressed() 信号，这时就会调用 sendText() 槽，这里发射了自定义的 getText() 信号，并将行编辑器中的内容清空。

下面双击 mainwindow. ui 文件进入设计模式，向中心区域拖入一个 Text Edit 部件，并使用 Ctrl+G 使其处于一个栅格布局中。然后进入 mainwindow. h 文件中添加一个私有槽的声明：

```
private slots:
    void setText(const QString &string); //向编辑器中添加文本
```

然后进入 mainwindow. cpp 文件中对该函数进行定义：

```
void MainWindow::setText(const QString &string) //插入文本
{
    ui->textEdit->setText(string);           //将获取的文本添加到编辑器中
}
```

然后在 mainwindow. cpp 中添加头文件 #include "myaction. h"，并在 MainWindow 类的构造函数中添加如下代码：

```
//添加菜单并且加入 action
MyAction * action = new MyAction;
QMenu * editMenu = ui->menuBar->addMenu(tr("编辑(&E)"));
editMenu->addAction(action);
//将 action 的 getText() 信号和这里的 setText() 槽进行关联
connect(action, SIGNAL(getText(QString)), this, SLOT(setText(QString)));
```

因为代码中使用了中文，所以还要在 main. cpp 文件中添加相应的代码。现在运行程序，并且在编辑菜单中单击自定义的菜单动作，然后输入字符并按下回车键。当然，也可以将这个 action 添加到工具栏中。

这个例子中设计了自己的信号和槽，整个过程是这样的：在行编辑器中输入文本然后按下回车键，这时行编辑就会发射 returnPressed() 信号，而这时就调用了我们的 sendText() 槽，sendText() 槽中又发射了 getText() 信号，它包含了行编辑器中的文本，接着又会调用 setText() 槽，在 setText() 槽中将 getText() 信号发来的文本输入到

文本编辑器中,这样就完成了按下回车键将行编辑器中的文本输入到中心部件的文本编辑器中的操作。其实,如果所有部件都在一个类中,就可以直接关联行编辑器的 `returnPressed()` 信号到我们的槽中,然后进行操作。但是,这里是在 `MyAction` 和 `MainWindow` 两个类之间进行数据传输,所以使用了自定义信号和槽。可以看到,如果能很好地掌握信号和槽的应用,那么实现几个类之间的相互操作是很简单的。

5.2 富文本处理

前面提到 `QTextEdit` 支持富文本的处理,什么是富文本呢?富文本(Rich Text)或者富文本格式,简单来说就是在文档中可以使用多种格式,比如字体颜色、图片和表格等。它是与纯文本(Plain Text)相对而言的,比如 Windows 上的记事本就是纯文本编辑器,而 Word 就是富文本编辑器。Qt 提供了对富文本处理的支持,可以在帮助中查看 Rich Text Processing 关键字,那里详细讲解了富文本处理的相关内容。

5.2.1 富文本文档结构

Qt 对富文本的处理分为了编辑操作和只读操作两种方式。编辑操作使用基于光标的一些接口函数,更好地模拟了用户的编辑操作,更加容易理解,而且不会丢失底层的文档框架;而对于文档结构的概览,使用了只读的分层次接口函数,有利于文档的检索和输出。可见,对于文档的读取和编辑要使用不同方面的两组接口。文档的光标主要基于 `QTextCursor` 类,而文档的框架主要基于 `QTextDocument` 类。一个富文本文档的结构分为几种元素来表示,分别是框架(`QTextFrame`)、文本块(`QTextBlock`)、表格(`QTextTable`)和列表(`QTextList`)。而每种元素的格式又使用相应的 `format` 类来表示,分别是框架格式(`QTextFrameFormat`)、文本块格式(`QTextBlockFormat`)、表格格式(`QTextTableFormat`)和列表格式(`QTextListFormat`),这些格式一般在编辑文档时使用,所以常和 `QTextCursor` 类配合使用。`QTextEdit` 类就是一个富文本编辑器,所以在构建 `QTextEdit` 类的对象时就已经构建了一个 `QTextDocument` 类对象和一个 `QTextCursor` 类对象,只须调用它们进行相应地操作即可,如图 5-8 所示。

一个空的文档包含了一个根框架(Root frame),这个根框架又包含了一个空的文本块(Block)。框架将一个文档分为多个部分,在根框架里可以再添加文本块、子框架和表格等,一个文档的结构如图 5-9 所示。下面先来看一下框架的实际应用。

(项目源码路径: `src\05\5-3\myRichText`)新建 Qt Gui 应用,项目名称为 `myRichText`,类名默认为 `MainWindow`,基类默认为 `QMainWindow`。建立好项目后,在设计模式向中心区域拖入一个 Text Edit 部件。然后到 `mainwindow.cpp` 文件中,先添加头文件 `#include <QTextFrame>`,再在 `MainWindow` 类的构造函数中添加如下代码:

```
QTextDocument * document = ui->textEdit->document(); // 获取文档对象
QTextFrame * rootFrame = document->rootFrame();           // 获取根框架
QTextFrameFormat format;                                     // 创建框架格式
```

```

format.setBorderBrush(Qt::red); //边界颜色
format.setBorder(3); //边界宽度
rootFrame ->setFrameFormat(format); //框架使用格式

```

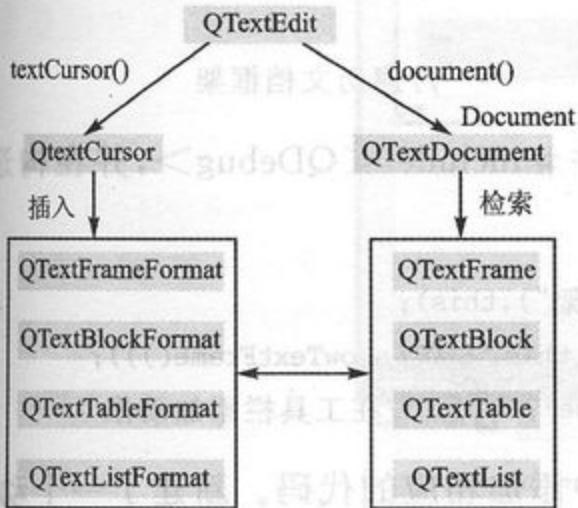


图 5-8 富文本元素

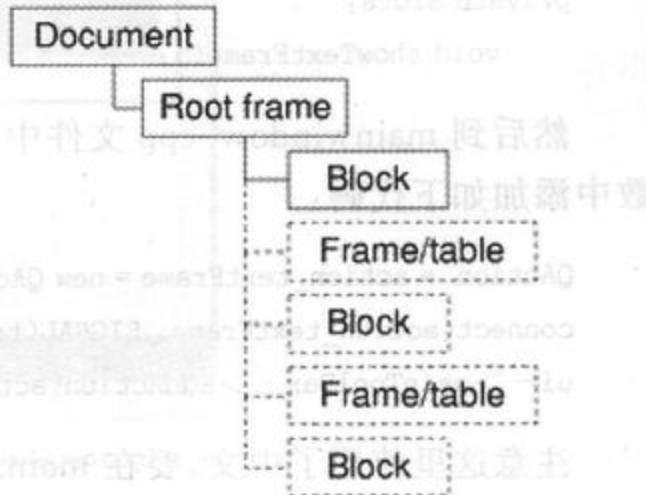


图 5-9 文档结构图

在构造函数中获取了编辑器中的文档对象,然后获取了文档的根框架,并且重新设置了框架的格式。现在运行程序会发现只能在红色的边框中进行输入,这里还可以使用 setHeight() 和 setWidth() 函数来固定框架的高度和宽度。

下面再添加代码,使用光标类对象,在根框架中再添加一个子框架。

```

QTextFrameFormat frameFormat;
frameFormat.setBackground(Qt::lightGray); //设置背景颜色
frameFormat.setMargin(10); //设置边距
frameFormat.setPadding(5); //设置填衬
frameFormat.setBorder(2);
frameFormat.setBorderStyle(QTextFrameFormat::BorderStyle_Dotted); //设置边框样式
QTextCursor cursor = ui->textEdit->textCursor(); //获取光标
cursor.insertFrame(frameFormat); //在光标处插入框架

```

这里又建立了一个框架格式,然后获取了编辑器的光标对象,并使用这个框架格式插入了一个新的框架。这里为框架格式设置了边白,分为边界内与本身内容间的空白,即填衬(Padding),和边界外与其他内容间的空白,即边距(Margin)。框架边界的样式有实线、点线等。

5.2.2 文本块

文本块 QTextBlock 类为文本文档 QTextDocument 提供了一个文本片段(QTextFragment)的容器。一个文本块可以看作一个段落,但是不能使用回车换行,因为一个回车换行就表示创建一个新的文本块。QTextBlock 提供了只读接口,是前面提到的文档分层次接口的一部分,如果 QTextFrame 看作一层,那么其中的 QTextBlock 就是另一层。文本块的格式由 QTextBlockFormat 类来处理,主要涉及对齐方式、文本块四周

的边白、缩进等内容。而文本块中的文本内容的格式,比如字体大小、加粗、下划线等内容,则由 QTextCharFormat 类来设置。下面从例子中去理解这些内容。

继续在前面的项目中添加代码。在 mainwindow.h 文件中添加私有槽函数的声明:

```
private slots:  
    void showTextFrame(); //遍历文档框架
```

然后到 mainwindow.cpp 文件中添加头文件 #include <QDebug>,并在构造函数中添加如下代码:

```
QAction * action_textFrame = new QAction(tr("框架"),this);  
connect(action_textFrame, SIGNAL(triggered()), this, SLOT(showTextFrame()));  
ui ->mainToolBar ->addAction(action_textFrame); //在工具栏添加动作
```

注意这里使用了中文,要在 main.cpp 文件中添加相应的代码。新建了一个动作,然后将它的触发信号和 showTextFrame() 槽关联,最后将它加入到了工具栏中。下面是 showTextFrame() 槽的实现:

```
void MainWindow::showTextFrame() //遍历框架  
{  
    QTextDocument * document = ui ->textEdit ->document();  
    QTextFrame * frame = document ->rootFrame();  
    QTextFrame::iterator it; //建立 QTextFrame 类的迭代器  
    for (it = frame ->begin(); ! (it.atEnd()); + + it) {  
        QTextFrame * childFrame = it.currentFrame(); //获取当前框架的指针  
        QTextBlock childBlock = it.currentBlock(); //获取当前文本块  
        if (childFrame)  
            qDebug() << "frame";  
        else if (childBlock.isValid())  
            qDebug() << "block:" << childBlock.text();  
    }  
}
```

在这个函数中获取了文档的根框架,然后使用它的迭代器 iterator 来遍历根框架中的所有子框架和文本块。在循环语句中先使用 QTextFrame 类的 begin() 函数使 iterator 指向根框架最开始的元素,然后使用 iterator 的 atEnd() 函数判断是否已经到达了根框架的最后一个元素。这里如果出现子框架,就输出一个框架的提示,如果出现文本块,便输出文本块提示和文本块的内容。现在运行程序,然后在编辑器中输入一些内容,按下工具栏中的“框架”动作,查看一下输出栏中的信息,如图 5-10 所示。

可以看到,这里只能输出根框架中的文本块和子框架,而子框架中的文本块却无法遍历到。其实还可以使用其他方法来遍历文档的所有文本块。下面在 mainwindow.h 文件中继续添加私有槽 private slots 声明:

```
void showTextBlock(); //遍历所有文本块
```

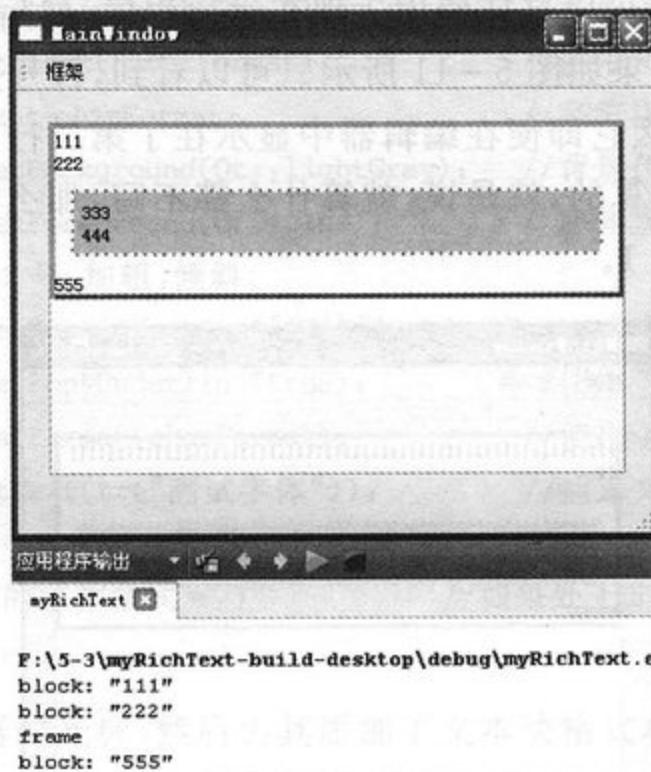


图 5-10 遍历框架运行效果

然后到 mainwindow.cpp 文件中的构造函数里继续添加如下代码：

```
QAction * action_textBlock = new QAction(tr("文本块"),this);
connect(action_textBlock,SIGNAL(triggered()),this,SLOT(showTextBlock()));
ui ->mainToolBar ->addAction(action_textBlock);
```

下面是 showTextBlock()槽的定义：

```
void MainWindow::showTextBlock() //遍历文本块
{
    QTextDocument * document = ui ->textEdit ->document();
    QTextBlock block = document ->firstBlock(); //获取文档的第一个文本块
    for (int i = 0; i < document ->blockCount(); i + +) {
        qDebug() << tr("文本块 %1, 文本块首行行号为: %2, 长度为: %3, 内容为: ")
            .arg(i).arg(block.firstLineNumber()).arg(block.length())
            << block.text();
        block = block.next(); //获取下一个文本块
    }
}
```

这里使用了 QTextDocument 类的 firstBlock() 函数来获取文档的第一个文本块，而 blockCount() 函数可以获取文档中所有文本块的个数，这样便可以使用循环语句来遍历所有文本块。对于每一个文本块都输出了它的编号、第一行的行号、长度和内容，然后使用 QTextBlock 的 next() 函数来获取下一个文本块。这里需要说明的是，在 tr() 函数中使用了“1%”等位置标记，然后在后面使用 arg() 添加了变量作为参数，这样这些参数就会代替前面字符串中的“1%”显示出来，字符串中有几个“%”号，后面就应该有几个 arg() 与其对应。这个 arg() 是 QString 类中的函数，因为 tr() 函数返回

QString 类对象,所以这里可以这样使用。现在运行程序,然后在编辑器中添加一些内容,按下“文本块”动作,效果如图 5-11 所示。可以看到,行号是从 0 开始标记的,而且如果不使用回车换行,那么它即便在编辑器中显示在了第二行,其实还是在一行里。文本块的长度是从 1 开始计算的,就是说,就算什么都不写,那么文本块的长度也是 1,所以长度会比实际字符数多 1。

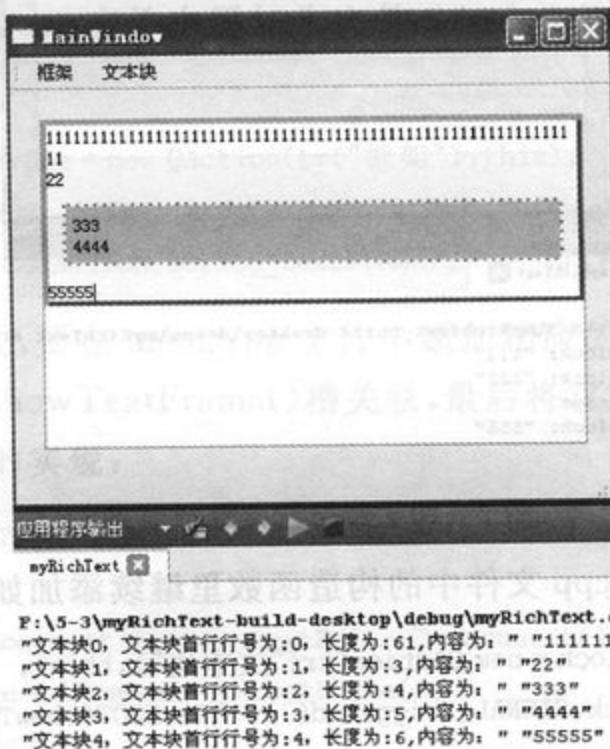


图 5-11 遍历文本块运行效果

下面再来看看怎样来编辑文本块及其内容的格式。前面讲到,对于编辑操作是使用基于光标的函数接口,我们来介绍一下几个常用的编辑操作。在 mainwindow.h 文件中添加私有槽 private slots 声明:

```
void setTextFont(bool checked); //设置字体格式
```

然后在 mainwindow.cpp 文件的构造函数中继续添加代码:

```
QAction * action_font = new QAction(tr("字体"),this);
action_font ->setCheckable(true); //设置动作可以被选中
connect(action_font,SIGNAL(toggled(bool)),this,SLOT(setTextFont(bool)));
ui ->mainToolBar ->addAction(action_font);
```

这里创建了一个动作,并设置它可以被选中,然后关联它的切换信号到我们的槽上。当动作的选中和取消选中状态切换时会触发切换信号 toggled(bool),当处于选中状态时参数 bool 值为 true。下面是 setTextFont()槽的定义:

```
void MainWindow::setTextFont(bool checked) //设置字体格式
{
    if(checked){ //如果处于选中状态
        QTextCursor cursor = ui ->textEdit ->textCursor();
        QTextBlockFormat blockFormat; //文本块格式
```

```

blockFormat.setAlignment(Qt::AlignCenter); //水平居中
cursor.insertBlock(blockFormat); //使用文本块格式
QTextCharFormat charFormat; //字符格式
charFormat.setBackground(Qt::lightGray); //背景色
charFormat.setForeground(Qt::blue); //字体颜色
//使用宋体,12号,加粗,倾斜
charFormat.setFont(QFont(tr("宋体"), 12, QFont::Bold, true));
charFormat.setFontUnderline(true); //使用下划线
cursor.setCharFormat(charFormat); //使用字符格式
cursor.insertText(tr("测试字体")); //插入文本
}
else{ /*恢复默认的字体格式 */ } //如果处于非选中状态,可以进行其他操作
}

```

这里先获得了编辑器的光标,然后为其添加了文本块格式和字符格式,文本块格式主要设置对齐方式、缩进等格式,而字符格式主要设置字体、颜色、下划线等格式。最后使用光标插入了一个测试文字。下面运行程序,按下“字体”动作,效果如图 5-12 所示。

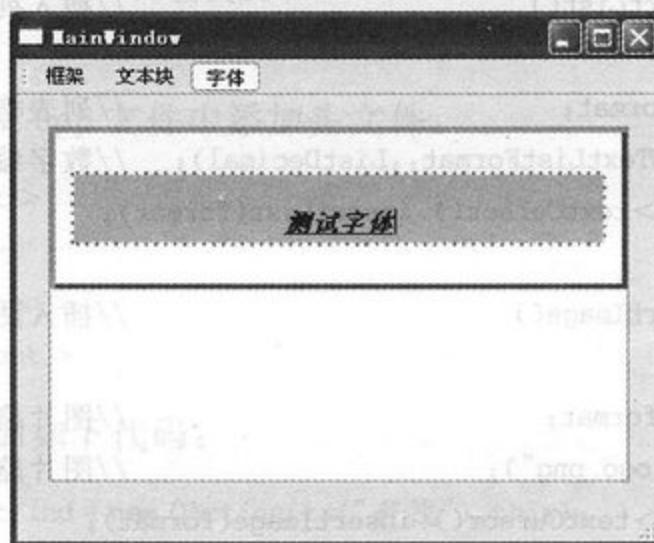


图 5-12 设置字体格式运行效果

5.2.3 表格、列表与图片

现在来看一下怎样在编辑器中插入表格、列表和图片。

(项目源码路径: src\05\5-4\myRichText)在前面的程序中继续添加代码。在 mainwindow.h 文件中添加私有槽 private slots 声明:

```

void insertTable(); //插入表格
void insertList(); //插入列表
void insertImage(); //插入图片

```

然后到 mainwindow.cpp 文件中的构造函数中继续添加代码:

```

QAction *action_textTable = new QAction(tr("表格"), this);
QAction *action_textList = new QAction(tr("列表"), this);

```

```

QAction * action_textImage = new QAction(tr("图片"),this);
connect(action_textTable, SIGNAL(triggered()), this, SLOT(insertTable()));
connect(action_textList, SIGNAL(triggered()), this, SLOT(insertList()));
connect(action_textImage, SIGNAL(triggered()), this, SLOT(insertImage()));
ui ->mainToolBar ->addAction(action_textTable);
ui ->mainToolBar ->addAction(action_textList);
ui ->mainToolBar ->addAction(action_textImage);

```

这里新建了 3 个动作，并将它们添加到工具栏中，下面是几个槽的定义：

```

void MainWindow::insertTable()                                //插入表格
{
    QTextCursor cursor = ui ->textEdit ->textCursor();
    QTextTableFormat format;                                 //表格格式
    format.setCellSpacing(2);                               //表格外边白
    format.setCellPadding(10);                             //表格内边白
    cursor.insertTable(2, 2, format);                     //插入 2 行 2 列表格
}

void MainWindow::insertList()                                //插入列表
{
    QTextListFormat format;                               //列表格式
    format.setStyle(QTextListFormat::ListDecimal);      //数字编号
    ui ->textEdit ->textCursor().insertList(format);
}

void MainWindow::insertImage()                                //插入图片
{
    QTextImageFormat format;                            //图片格式
    format.setName("logo.png");                      //图片路径
    ui ->textEdit ->textCursor().insertImage(format);
}

```

对于表格和列表，也可以使用 `QTextFrame::iterator` 来遍历它们，可以在帮助中参考 Rich Text Document Structure 关键字。表格对应的是 `QTextTable` 类，还提供了 `cellAt()` 函数来获取指定的单元格；`insertColumns()` 函数来插入列；`insertRows()` 函数来插入行；`mergeCells()` 函数来合并单元格；`splitCell()` 函数来拆分单元格。对于一个单元格，其对应的类是 `QTextTableCell`，而其格式对应的类是 `QTextTableCellFormat` 类。列表对应的类是 `QTextList`，它提供了 `count()` 函数来获取列表中项目的个数；`item()` 函数获取指定的项目的文本块；`removeItem()` 函数来删除一个项目。对于列表编号，这里使用了数字编号，更多的选项可以查看 `QTextListFormat::Style` 关键字。对于图片，可以使用 `QTextImageFormat` 类的 `setHeight()` 和 `setWidth()` 函数来设置图片的高度和宽度，这可能会将图片进行拉伸或压缩而变形。因为图片没有对应的类，所以只能使用图片格式类，程序中使用了 `setName()` 函数来指定图片，这里需要往项目文件夹的相应 `build-desktop` 文件夹中放入一张图片，当然，也可以将图片放到任何地方。

或者使用资源管理器，只需要将程序中的路径改一下。

5.2.4 查找功能

其实像字体格式设置等操作完全可以在 QTextEdit 类中直接进行。QTextEdit 类提供了很多方便的函数，比如常用的复制、粘贴操作，撤销、恢复操作，放大、缩小操作等。关于这些，这里不再介绍，因为它们很简单，只须调用一个函数即可。下面简单介绍一下文本查找功能，它使用的是 QTextEdit 类的 find() 函数。

(项目源码路径：src\05\5-5\myRichText) 仍然在前面的程序中添加代码。在 mainwindow.h 文件中添加类的前置声明：

```
class QLineEdit;
```

然后添加一个私有 private 对象指针定义：

```
QLineEdit *lineEdit;
```

再添加两个私有槽 private slots 声明：

```
void textFind();           //查找文本
void findNext();           //查找下一个
```

然后到 mainwindow.cpp 文件中添加头文件：

```
#include <QLineEdit>
#include <QDialog>
#include <QPushButton>
#include <QVBoxLayout>
```

再在构造函数中添加如下代码：

```
QAction *action_textFind = new QAction(tr("查找"),this);
connect(action_textFind,SIGNAL(triggered()),this,SLOT(textFind()));
ui ->mainToolBar ->addAction(action_textFind);
```

然后是两个函数的定义：

```
void MainWindow::textFind()           //查找文本
{
    QDialog *dlg = new QDialog(this);   //创建对话框
    lineEdit = new QLineEdit(dlg);      //创建行编辑器
    QPushButton *btn = new QPushButton(dlg); //创建按钮
    btn ->setText(tr("查找下一个"));
    connect(btn,SIGNAL(clicked()),this,SLOT(findNext())); //关联信号和槽
    QVBoxLayout *layout = new QVBoxLayout;           //创建垂直布局管理器
    layout ->addWidget(lineEdit);                  //添加部件
    layout ->addWidget(btn);                     //添加部件
    dlg ->setLayout(layout);                    //在对话框中使用布局管理器
```

```

    dlg ->show();
}

void MainWindow::findNext() //查找下一个
{
    QString string = lineEdit ->text();
    //使用查找函数查找指定字符串,查找方式为向后查找
    bool isfind = ui ->textEdit ->find(string, QTextDocument::FindBackward);
    if(isfind){ //如果查找成功,输出字符串所在行和列的编号
        qDebug() << tr("行号: %1 列号: %2")
            .arg(ui ->textEdit ->textCursor().blockNumber())
            .arg(ui ->textEdit ->textCursor().columnNumber());
    }
}

```

这两个函数都是很简单的操作,就是创建了一个查找对话框,然后使用了 QTextEdit 类的 find() 函数进行查找。这里使用了选项 QTextDocument::FindBackward 表示向后查找,默认的是向前查找。另外还有 QTextDocument::FindCaseSensitively 表示不区分大小写和 QTextDocument::FindWholeWords 表示匹配整个单词。其实, QTextEdit 中的 find() 函数只是为了方便使用而设计的,更多的查找功能可以使用 QTextDocument 类的 find() 函数,它有几种形式可以选择,其中还可以使用正则表达式。查找到相应字符时便输出它们所在的行号和列号,它们都是从 0 开始的。

5.2.5 语法高亮与 HTML

在使用 Qt Creator 编辑代码时可以发现,输入关键字时会显示不同的颜色,这就是所谓的语法高亮。在 Qt 的富文本处理中提供了 QSyntaxHighlighter 类来实现语法高亮。为了实现这个功能,需要创建 QSyntaxHighlighter 类的子类,然后重新实现 highlightBlock() 函数,使用时直接将 QTextDocument 类对象指针作为其父部件指针,这样就可以自动调用 highlightBlock() 函数了。

(项目源码路径: src\05\5-6\myRichText)首先往前面的项目中添加新文件,模板选择 C++ 类,类名为 MySyntaxHighlighter,基类为 QSyntaxHighlighter,类型信息选择“继承自 QObject”。然后在 mysyntaxhighlighter.h 文件中将构造函数声明改为:

```
explicit MySyntaxHighlighter(QTextDocument * parent = 0);
```

然后再添加一个函数声明:

```
protected:
    void highlightBlock(const QString &text); //必须重新实现该函数
```

现在到 mysyntaxhighlighter.cpp 文件中,先更改构造函数为:

```
MySyntaxHighlighter::MySyntaxHighlighter(QTextDocument * parent) :
    QSyntaxHighlighter(parent)
```

```
{
}
```

然后对 highlightBlock() 函数进行定义：

```
void MySyntaxHighlighter::highlightBlock(const QString &text) //高亮文本块
{
    QTextCharFormat myFormat; //字符格式
    myFormat.setFontWeight(QFont::Bold);
    myFormat.setForeground(Qt::green);
    QString pattern = "\bchar\b"; //要匹配的字符,这里是“char”单词
    QRegExp expression(pattern); //创建正则表达式
    int index = text.indexOf(expression); //从位置 0 开始匹配字符串
    //如果匹配成功,那么返回值为字符串的起始位置,它大于或等于 0
    while (index >= 0) {
        int length = expression.matchedLength(); //要匹配字符串的长度
        setFormat(index, length, myFormat); //对要匹配的字符串设置格式
        index = text.indexOf(expression, index + length); //继续匹配
    }
}
```

这里主要使用了正则表达式和 QString 类的 indexOf() 函数来进行字符串的匹配,如果匹配成功,则使用 QSyntaxHighlighter 类的 setFormat() 函数来设置字符格式。下面使用这个自定义的类。

在 mainwindow.h 文件中添加类的前置声明：

```
class MySyntaxHighlighter;
```

然后再添加私有对象指针定义：

```
MySyntaxHighlighter * highlighter;
```

到 mainwindow.cpp 文件中添加头文件：

```
#include "mysyntaxhighlighter.h"
```

然后在构造函数的最后添加一行代码：

```
highlighter = new MySyntaxHighlighter(ui->textEdit->document());
```

这里创建了 MySyntaxHighlighter 类的对象,并且使用编辑器的文档对象指针作为其参数,这样,每当编辑器中的文本改变时都会调用 highlightBlock() 函数来设置语法高亮。

关于语法高亮,可以查看 Syntax Highlighter Example 示例程序,它在 Rich Text 分类下。富文本处理中还提供了对 HTML 子集的支持,可以在 QLabel 或者 QTextEdit 添加文本时使用 HTML 标签或者 CSS 属性,具体内容可以在帮助中参考 Supported HTML Subset 关键字,下面举一个最简单的例子。

在 mainwindow.cpp 文件中的构造函数的最后添加下面一行代码：

```
ui ->textEdit ->append(tr("<h1><font color = red>使用 HTML</font></h1>"));
```

这里往编辑器中添加了文本，并且使用了 HTML 标签。

前面讲到了在编辑器中使用语法高亮，那么可能会想到在编辑代码时另一个非常有用的功能，就是自动补全。Qt 中提供了 QCompleter 类来实现自动补全，可以使它来实现编辑器中的自动补全功能，这个可以参考示例程序 Custom Completer，它在 Tools 分类下。关于富文本处理的内容我们就讲到这里。富文本处理涉及的东西很多，要学好就要多动手去编写程序。在帮助文档中的 Advanced Rich Text Processing 里还提供了一个处理大文档的方法，可以查看这个关键字。对于这部分的示例程序，可以查看 Rich Text 分类下的几个程序，也可以查看一下 Text Edit 程序，这个例子是个综合的富文本编辑器。

5.3 拖放操作

对于一个实用的应用程序，不仅希望能从文件菜单中打开一个文件，更希望通过拖动直接将桌面上的文件拖入程序界面上来打开，就像可以将.pro 文件拖入 Creator 中来打开整个项目一样。Qt 中提供了强大的拖放机制，可以在帮助中查找 Drag and Drop 关键字来了解。拖放操作分为拖动(Drag)和放下(Drop)两种操作。数据拖动时会被存储为 MIME(Multipurpose Internet Mail Extensions)类型，在 Qt 使用 QMimeData 类来表示 MIME 类型的数据，并使用 QDrag 类来完成数据的转换，而整个拖放操作都是在几个鼠标事件和拖放事件中完成的。

5.3.1 使用拖放打开文件

下面来看一个很简单的例子，就是将桌面上的 txt 文本文件拖入程序打开。(项目源码路径：src\05\5-7\myDragDrop)新建 Qt Gui 应用，项目名称改为 myDragDrop，类名和基类保持 MainWindow 和 QMainWindow 不变。建立完项目后，往界面上拖入一个 Text Edit 部件。然后在 mainwindow.h 文件中添加函数声明：

```
protected:  
    void dragEnterEvent(QDragEnterEvent * event); // 拖动进入事件  
    void dropEvent(QDropEvent * event); // 放下事件
```

然后到 mainwindow.cpp 文件中添加头文件：

```
#include <QDragEnterEvent>  
#include <QUrl>  
#include <QFile>  
#include <QTextStream>
```

最后对两个事件处理函数进行定义：

```

void MainWindow::dragEnterEvent(QDragEnterEvent * event) //拖动进入事件
{
    if(event ->mimeData() ->hasUrls()) //数据中是否包含 URL
        event ->acceptProposedAction(); //如果是则接收动作
    else event ->ignore(); //否则忽略该事件
}

void MainWindow::dropEvent(QDropEvent * event) //放下事件
{
    const QMimeData * mimeData = event ->mimeData(); //获取 MIME 数据
    if(mimeData ->hasUrls()){ //如果数据中包含 URL
        QList<QUrl> urlList = mimeData ->urls(); //获取 URL 列表
        //将其中第一个 URL 表示为本地文件路径
        QString fileName = urlList.at(0).toLocalFile();
        if(!fileName.isEmpty()){ //如果文件路径不为空
            QFile file(fileName); //建立 QFile 对象并且以只读方式打开该文件
            if(!file.open(QIODevice::ReadOnly)) return;
            QTextStream in(&file); //建立文本流对象
            ui ->textEdit ->setText(in.readAll()); //将文件中所有内容读入编辑器
        }
    }
}

```

当鼠标拖拽一个数据进入主窗口时,就会触发 dragEventEvent()事件处理函数,获取其中的 MIME 数据,然后查看它是否包含 URL 路径,因为拖入的文本文件实际上就是拖入了它的路径,这就是 event->mimeData()->hasUrls()实现的功能。如果有这样的数据就接收它,否则就忽略该事件。QMimeData 类中提供了几个函数来方便地处理常见的 MIME 数据,如表 5-1 所列。当松开鼠标左键,将数据放入主窗口时就会触发 dropEvent()事件处理函数,这里获取了 MIME 数据中的 URL 列表,因为拖入的只有一个文件,所以获取了列表中的第一个条目,并使用 toLocalFile()函数将它转换为本地文件路径。然后使用 QFile 和 QTextStream 将文件中的数据读入编辑器中。这时先运行程序,然后从桌面上将一个文本文件拖入程序中。

表 5-1 常用 MIME 类型数据处理函数

测试函数	获取函数	设置函数	MIME 类型
hasText()	text()	setText()	text/plain
hasHtml()	html()	setHtml()	text/html
hasUrls()	urls()	setUrls()	text/uri-list
hasImage()	imageData()	setImageData()	image/*
hasColor()	colorData()	setColorData()	application/x-color

5.3.2 自定义拖放操作

下面再来看一个在窗口中拖动图片的例子,就是在窗口中有一个图片,可以随意拖动

它。这里需要使用到自定义的 MIME 类型。(项目源码路径: src\05\5-8\imageDragDrop)新建 Qt Gui 应用,项目名称改为 imageDragDrop,类名和基类保持 MainWindow 和 QMainWindow 不变。在 mainwindow.h 文件中对几个事件处理函数进行声明:

```
protected:
    void mousePressEvent(QMouseEvent * event);           //鼠标按下事件
    void dragEnterEvent(QDragEnterEvent * event);        //拖动进入事件
    void dragMoveEvent(QDragMoveEvent * event);          //拖动事件
    void dropEvent(QDropEvent * event);                  //放下事件
```

然后到 mainwindow.cpp 文件中添加头文件:

```
# include <QLabel>
# include <QMouseEvent>
# include <QDragEnterEvent>
# include <QDragMoveEvent>
# include <QDropEvent>
# include <QPainter>
```

然后在构造函数中添加如下代码:

```
setAcceptDrops(true);                                //设置窗口部件可以接收拖入
QLabel * label = new QLabel(this);                  //创建标签
QPixmap pix("../yafeilinux.png");
label ->setPixmap(pix);                           //添加图片
label ->resize(pix.size());                      //设置标签大小为图片的大小
label ->move(100,100);
label ->setAttribute(Qt::WA_DeleteOnClose);       //当窗口关闭时销毁图片
```

这里必须先设置部件,使其可以接受拖放操作,因为在上面的例子中 QTextEdit 默认可以接受拖放操作,所以不用设置,但是其他部件默认是不可以接受拖放操作的。然后创建一个标签,并且为它添加了一张图片,这里将图片放入了项目文件夹下。下面是几个事件处理函数的定义:

```
void MainWindow::mousePressEvent(QMouseEvent * event)      //鼠标按下事件
{
    //第一步: 获取图片
    //将鼠标指针所在位置的部件强制转换为 QLabel 类型
    QLabel * child = static_cast<QLabel*>(childAt(event->pos()));
    if(! child->inherits("QLabel")) return; //如果部件不是 QLabel 则直接返回
    QPixmap pixmap = * child->pixmap();           //获取 QLabel 中的图片

    //第二步: 自定义 MIME 类型
    QByteArray itemData;                          //创建字节数组
    QDataStream dataStream(&itemData, QIODevice::WriteOnly); //创建数据流
    //将图片信息,位置信息输入到字节数组中
```

```

dataStream << pixmap << QPoint(event->pos() - child->pos()); //将图片和位置信息写入 QByteArray

//第三步：将数据放入 QMimeData 中
QMimeData * mimeData = new QMimeData(); //创建 QMimeData 用来存放要移动的数据
//将字节数组放入 QMimeData 中,这里的 MIME 类型是我们自己定义的
mimeData->setData("myimage/png", itemData);

//第四步：将 QMimeData 数据放入 QDrag 中
QDrag * drag = new QDrag(this); //创建 QDrag,它用来移动数据
drag->setMimeData(mimeData);
drag->setPixmap(pixmap); //在移动过程中显示图片,若不设置则默认显示一个小矩形
drag->setHotSpot(event->pos() - child->pos()); //拖动时鼠标指针的位置不变

//第五步：给原图片添加阴影
QPixmap tempPixmap = pixmap; //使原图片添加阴影
QPainter painter; //创建 QPainter,用来绘制 QPixmap
painter.begin(&tempPixmap);
//在图片的外接矩形中添加一层透明的淡黑色形成阴影效果
painter.fillRect(pixmap.rect(), QColor(127, 127, 127, 127));
painter.end();
child->setPixmap(tempPixmap); //在移动图片过程中,让原图片添加一层黑色阴影

//第六步：执行拖放操作
if (drag->exec(Qt::CopyAction | Qt::MoveAction, Qt::CopyAction)
    == Qt::MoveAction) //设置拖放可以是移动和复制操作,默认是复制操作
    child->close(); //如果是移动操作,那么拖放完成后关闭原标签
else {
    child->show(); //如果是复制操作,那么拖放完成后显示标签
    child->setPixmap(pixmap); //显示原图片,不再使用阴影
}
}

```

鼠标按下时会触发鼠标按下事件进而执行其处理函数,这里进行了一系列操作,就像程序中注释所描述的那样,大体上可以分为 6 步。第一步,先获取鼠标指针所在处的部件的指针,将它强制转换为 QLabel 类型的指针,然后使用 inherits() 函数判断它是否是 QLabel 类型,如果不是则直接返回,不再进行下面的操作。第二步,因为不仅要在拖动的数据中包含图片数据,还要包含它的位置信息,所以需要使用自定义的 MIME 类型。这里使用了 QByteArray 字节数组来存放图片数据和位置数据。然后使用 QDataStream 类将数据写入数组中。其中,位置信息是当前鼠标指针的坐标减去图片左上角的坐标而得到的差值。第三步,创建了 QMimeData 类对象指针,使用了自定义的 MIME 类型“myimage/png”,将字节数组放入 QMimeData 中。第四步,为了移动数据,必须创建 QDrag 类对象,然后为其添加 QMimeData 数据。这里为了在移动过程

中一直显示图片，需要使用 `setPixmap()` 函数为其设置图片。然后使用 `setHotSpot()` 函数指定了鼠标在图片上单击的位置，这是相对于图片左上角的位置；如果不设定这个，那么在拖动图片过程中，指针会位于图片的左上角。第五步，在移动图片过程中我们希望原来的图片有所改变来表明它正在被操作，所以为其添加了一层阴影。第六步，执行拖动操作，这需要使用 `QDrag` 类的 `exec()` 函数，它不会影响主事件循环，所以这时界面不会被冻结。这个函数可以设定所支持的放下动作和默认的放下动作，比如这里设置了支持复制动作 `Qt::CopyAction` 和移动动作 `Qt::MoveAction`，并设置默认的动作是复制。这就是说我们拖动图片，可以是移动它，也可以是进行复制，而默认的是复制操作，比如使用 `acceptProposedAction()` 函数时就是使用默认的操作。当图片被放下后 `exec()` 函数就会返回操作类型，这个返回值由下面要讲到的 `dropEvent()` 函数中的设置决定。我们这里判断到底进行了什么操作，如果是移动操作，那么就删除原来的图片；如果是复制操作，就恢复原来的图片。

```
void MainWindow::dragEnterEvent(QDragEnterEvent * event) // 拖动进入事件
{
    // 如果有我们定义的 MIME 类型数据，则进行移动操作
    if (event->mimeData()->hasFormat("myimage/png")) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::dragMoveEvent(QDragMoveEvent * event) // 拖动事件
{
    if (event->mimeData()->hasFormat("myimage/png")) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    } else {
        event->ignore();
    }
}
```

在这两个事件处理函数中，先判断拖动的数据中是否有我们自定义的 MIME 类型的数据，如果有，则执行移动动作 `Qt::MoveAction`。

```
void MainWindow::dropEvent(QDropEvent * event) // 放下事件
{
    if (event->mimeData()->hasFormat("myimage/png")) {
        QByteArray itemData = event->mimeData()->data("myimage/png");
        QDataStream dataStream(&itemData, QIODevice::ReadOnly);
        QPixmap pixmap;
```

```

QPoint offset;
//使用数据流将字节数组中的数据读入到 QPixmap 和 QPoint 变量中
dataStream >> pixmap >> offset;
//新建标签,为其添加图片,并根据图片大小设置标签的大小
QLabel *newLabel = new QLabel(this);
newLabel ->setPixmap(pixmap);
newLabel ->resize(pixmap.size());
//让图片移动到放下的位置,不设置的话,图片会默认显示在(0,0)点即窗口左上角
newLabel ->move(event ->pos() - offset);
newLabel ->show();
newLabel ->setAttribute(Qt::WA_DeleteOnClose);
event ->setDropAction(Qt::MoveAction);
event ->accept();
} else {
    event ->ignore();
}
}
}

```

在放下事件中,使用字节数组获取了拖放的数据,然后将其中的图片数据和位置数据读取到两个变量中,并使用它们来设置新建的标签。

这个例子是对图片进行移动,如果想对图片进行复制,只需要将 dragEnterEvent()、dragMoveEvent() 和 dropEvent() 这 3 个函数中的 event->setDropAction() 函数中的参数改为 Qt::CopyAction 即可。对于拖放操作的其他应用,可以在帮助中查看 Drag and Drop 关键字中的相关内容,比如根据移动的距离来判断是否开始一个拖放操作,还有剪贴板 QClipboard 类,不过,因为编辑器中的剪切、粘贴等功能都提供了现成的函数,所以这个类很少用到。关于这部分内容,也可以参考 Drag and Drop 分类中的几个示例程序。

5.4 打印文档

Qt 中提供了方便的打印文档功能,只需要使用一个 QPrinter 类和一个打印对话框 QPrintDialog 类就可以完成文档的打印操作。在这一节将简单介绍一下打印文档、打印预览和生产 PDF 文档等操作,也可以在帮助中查看 Printing with Qt 关键字。

(项目源码路径: src\05\5-9\myPrint) 新建 Qt Gui 应用,项目名称改为 myPrint,类名和基类保持 MainWindow 和 QMainWindow 不变。然后在设计模式向界面上拖入一个 Text Edit,再到 mainwindow.h 文件中进行几个槽的声明:

```

private slots:
void doPrint();
void doPrintPreview();
void printPreview(QPrinter * printer);

```

```
void createPdf();
```

然后到 mainwindow.cpp 文件中添加头文件：

```
#include <QPrinter>
#include <QPrintDialog>
#include <QPrintPreviewDialog>
#include <QFileDialog>
#include <QFileInfo>
```

在构造函数中定义几个动作：

```
QAction *action_print = new QAction(tr("打印"), this);
QAction *action_printPreview = new QAction(tr("打印预览"), this);
QAction *action_pdf = new QAction(tr("生成 pdf"), this);
connect(action_print, SIGNAL(triggered()), this, SLOT(doPrint()));
connect(action_printPreview, SIGNAL(triggered()), this, SLOT(doPrintPreview()));
connect(action_pdf, SIGNAL(triggered()), this, SLOT(createPdf()));
ui->mainToolBar->addAction(action_print);
ui->mainToolBar->addAction(action_printPreview);
ui->mainToolBar->addAction(action_pdf);
```

然后添加那几个槽的定义：

```
void MainWindow::doPrint() //打印
{
    QPrinter printer; //创建打印机对象
    QPrintDialog dlg(&printer, this); //创建打印对话框
    //如果编辑器中有选中区域，则打印选中区域
    if (ui->textEdit->textCursor().hasSelection())
        dlg.addEnabledOption(QAbstractPrintDialog::PrintSelection);
    if (dlg.exec() == QDialog::Accepted) //如果在对话框中按下了打印按钮
        ui->textEdit->print(&printer); //则执行打印操作
}
```

这里先建立了 QPrinter 类对象，它代表了一个打印设备。然后创建了一个打印对话框，如果编辑器中有选中区域则打印该区域，否则打印整个页面。

```
void MainWindow::doPrintPreview() //打印预览
{
    QPrinter printer;
    QPrintPreviewDialog preview(&printer, this); //创建打印预览对话框
    //当要生成预览页面时，发射 paintRequested() 信号
    connect (&preview, SIGNAL(paintRequested(QPrinter *)), this, SLOT(printPreview(QPrinter *)));
    preview.exec();
```

```

}

void MainWindow::printPreview(QPrinter * printer)
{
    ui->textEdit->print(printer);
}

```

这里主要使用打印预览对话框来进行打印预览,这里要关联它的 paintRequested() 信号到我们的槽上,在槽中调用编辑器的打印函数,并以传来的 QPrinter 类对象指针为参数。

```

void MainWindow::createPdf() //生成 PDF 文件
{
    QString fileName = QFileDialog::getSaveFileName(this, tr("导出 PDF 文件"),
                                                    QString(), " *.pdf");

    if (!fileName.isEmpty()) {
        if (QFileInfo(fileName).suffix().isEmpty())
            fileName.append(".pdf"); //如果文件后缀为空,则默认使用.pdf

        QPrinter printer;
        printer.setOutputFormat(QPrinter::PdfFormat); //指定输出格式为 pdf
        printer.setOutputFileName(fileName);
        ui->textEdit->print(&printer);
    }
}

```

在生成 PDF 文档的槽中,使用文件对话框来获取要保存文件的路径,如果文件名没有指定后缀则为其添加“.pdf”后缀。然后为 QPrinter 对象指定输出格式和文件路径,这样就可以将文档打印成 PDF 格式了。

现在运行程序,如果没有安装打印机,那么弹出的打印对话框将无法使用打印操作,打印预览对话框也看不到真实的效果。

5.5 小结

通过学习这一章,读者要掌握主窗口各个部件的使用,能够自行开发出一个简单的基于 QMainWindow 的程序。另外,还涉及了资源文件的使用以及信号和槽的设计,这些都是开发 Qt 程序的基础。因为富文本处理是一个庞大的体系,这里不可能讲到每一个细节,所以还需要结合帮助文档多加练习。后面的拖放操作和打印文档等内容,也可以等到使用时再去学习。

学习完这一章,读者可以去看《Qt 及 Qt Quick 开发实战精解》中的多文档编辑器实例,那个程序比较综合,如果可以很好地完成,那么说明 Qt 已经入门了。

第6章

事件系统

第5章讲解拖放操作时曾提到了拖放事件,这一章将讲解Qt中的事件系统。在Qt中,事件作为一个对象,继承自QEvent类,常见的有键盘事件QKeyEvent、鼠标事件QMouseEvent和定时器事件QTimerEvent等,与QEvent类的继承关系如图6-1所示。本章中会详细讲解这3个常见的事件,还会涉及事件过滤器、自定义事件和随机数的知识。关于本章的相关内容,可以在Qt帮助中查看The Event System关键字。

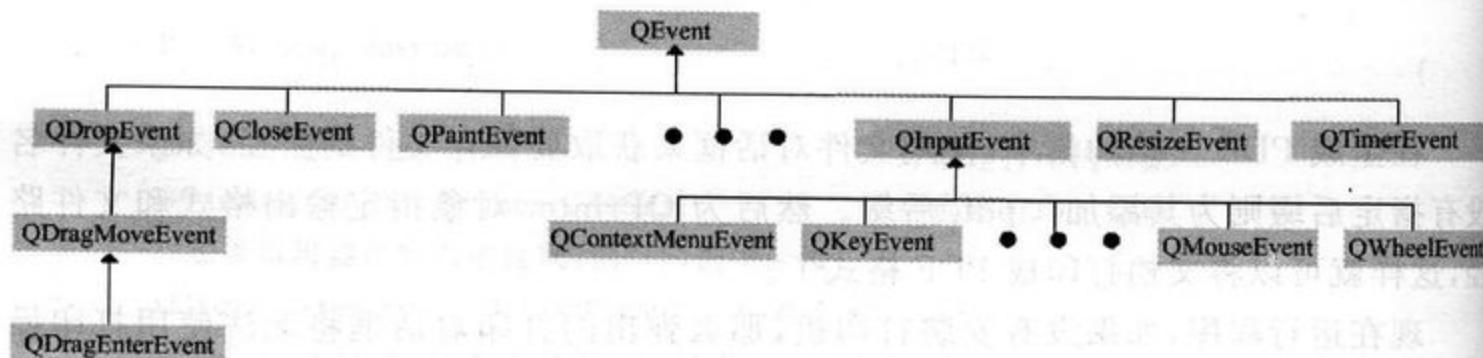


图 6-1 QEvent 类关系图

6.1 Qt 中的事件

事件是对各种应用程序需要知道的由应用程序内部或者外部产生的事情或者动作的通称。在Qt中使用一个对象来表示一个事件,继承自QEvent类。需要说明的是,事件与信号并不相同,比如单击一下界面上的按钮,那么就会产生鼠标事件QMouseEvent(不是按钮产生的),而因为按钮被按下了,所以它会发出clicked()单击信号(是按钮产生的)。这里一般只关心按钮的单击信号,而不用考虑鼠标事件,但是如果要设计一个按钮,或者当单击按钮时让它产生别的效果,那么就要关心鼠标事件了。可以看到,事件与信号是两个不同层面的东西,发出者不同,作用也不同。在Qt中,任何QObject子类实例都可以接收和处理事件。

对于最常用的一些事件,比如上下文菜单事件 QContextMenuEvent 和关闭事件 QCloseEvent,在《Qt 及 Qt Quick 开发实战精解》中的多文档编辑器实例小节有它们具体的应用实例;拖放事件在上一章中已经讲了;对于绘制事件 QPaintEvent,将会在第 10 章 2D 绘图部分经常用到。

6.1.1 事件的处理

一个事件由一个特定的 QEvent 子类来表示,但是有时一个事件又包含多个事件类型,比如鼠标事件又可以分为鼠标按下、双击和移动等多种操作。这些事件类型都由 QEvent 类的枚举型 QEvent::Type 来表示,其中包含了一百多种事件类型,可以在 QEvent 类的帮助文档中查看。虽然 QEvent 的子类可以表示一个事件,但是却不能用来处理事件,那么应该怎样来处理一个事件呢?在 QApplication 类的 notify() 函数的帮助文档处给出了 5 种处理事件的方法:

方法一:重新实现部件的 paintEvent()、mousePressEvent() 等事件处理函数。这是最常用的一种方法,不过它只能用来处理特定部件的特定事件。例如前一章实现拖放操作,就是用的这种方法。

方法二:重新实现 notify() 函数。这个函数功能强大,提供了完全的控制,可以在事件过滤器得到事件之前就获得它们。但是,它一次只能处理一个事件。

方法三:向 QApplication 对象上安装事件过滤器。因为一个程序只有一个 QApplication 对象,所以这样实现的功能与使用 notify() 函数是相同的,优点是可以同时处理多个事件。

方法四:重新实现 event() 函数。QObject 类的 event() 函数可以在事件到达默认的事件处理函数之前获得该事件。

方法五:在对象上安装事件过滤器。使用事件过滤器可以在一个界面类中同时处理不同子部件的不同事件。

在实际编程中,最常用的是方法一,其次是方法五。因为方法二需要继承自 QApplication 类;而方法三要使用一个全局的事件过滤器,这将减缓事件的传递,所以,虽然这两种方法功能很强大,但是却很少被用到。

6.1.2 事件的传递

在第 2 章讲解 helloworld 程序代码时就曾提到过,在每个程序的 main() 函数的最后都会调用 QApplication 类的 exec() 函数,它会使 Qt 应用程序进入事件循环,这样就可以使应用程序在运行时接收发生的各种事件。一旦有事件发生,Qt 便会构建一个相应的 QEvent 子类的对象来表示,然后将它传递给相应的 QObject 对象或其子对象。下面通过例子来看一下 Qt 中的事件传递过程。

(项目源码路径: src\06\6-1\myEvent)新建 Qt Gui 应用,项目名称为 myEvent,基类选择 QWidget,然后类名保持 Widget 不变。建立完成后向项目中添加新文件,模板选择 C++ 类,类名为 MyLineEdit,基类为 QLineEdit,类型信息选择“继承自 QWidget”。

get”。然后在 mylineedit.h 文件中添加函数声明：

```
protected:  
    void keyPressEvent(QKeyEvent * event);
```

再到 mylineedit.cpp 文件中添加头文件：

```
# include <QKeyEvent>  
# include <QDebug>
```

然后是事件处理函数的定义：

```
void MyLineEdit::keyPressEvent(QKeyEvent * event) //键盘按下事件  
{  
    qDebug() << tr("MyLineEdit 键盘按下事件");  
}
```

下面进入 widget.h 文件中，添加类前置声明：

```
class MyLineEdit;
```

然后添加函数声明：

```
protected:  
    void keyPressEvent(QKeyEvent * event);
```

再添加一个 private 对象指针定义

```
MyLineEdit * lineEdit;
```

然后进入 widget.cpp 文件中，添加头文件：

```
# include "mylineedit.h"  
# include <QKeyEvent>  
# include <QDebug>
```

在 Widget 类的构造函数中添加代码：

```
lineEdit = new MyLineEdit(this);  
lineEdit ->move(100,100);
```

然后是事件处理函数的定义：

```
void Widget::keyPressEvent(QKeyEvent * event)  
{  
    qDebug() << tr("Widget 键盘按下事件");  
}
```

因为程序中使用了中文，所以还要在 main.cpp 文件中添加必要的代码。在这里自定义了一个 MyLineEdit 类，继承自 QLineEdit 类，然后在 Widget 界面中添加了一个 MyLineEdit 部件。要注意，这里既实现了 MyLineEdit 类的键盘按下事件处理函数，也

实现了 Widget 类的键盘按下事件处理函数。现在运行程序,这时光标焦点在行编辑器中,随便在键盘上按下一个按键,比如按下 A 键,这时在 Qt Creator 的应用程序输出栏中只会出现“MyLineEdit 键盘按下事件”,说明这时只执行了 MyLineEdit 类中的 keyPressEvent() 函数。

下面到 mylineedit.cpp 文件中的 keyPressEvent() 函数中添加如下一行代码,让它忽略掉这个事件:

```
event -> ignore(); //忽略该事件
```

这时再运行程序,按下 A 键,那么在以前输出的基础上又输出了“Widget 键盘按下事件”,说明这时也执行了 Widget 类中的 keyPressEvent() 函数。但是现在出现了一个问题,就是行编辑器中无法输入任何字符,为了让它可以正常工作,还需要在 mylineedit.cpp 文件中的 keyPressEvent() 函数中添加一行代码,整个函数定义如下:

```
void MyLineEdit::keyPressEvent(QKeyEvent * event) //键盘按下事件
{
    qDebug() << tr("MyLineEdit 键盘按下事件");
    QLineEdit::keyPressEvent(event); //执行 QLineEdit 类的默认事件处理
    event -> ignore(); //忽略该事件
}
```

这里调用了 MyLineEdit 父类 QLineEdit 的 keyPressEvent() 函数来实现行编辑器的默认操作。这里一定要注意代码的顺序,ignore() 函数要在最后调用。

从这个例子中可以看到,事件是先传递给指定窗口部件的,这里确切地说应该是先传递给获得焦点的窗口部件的。但是如果该部件忽略掉该事件,那么这个事件就会传递给这个部件的父部件。在重新实现事件处理函数时,一般要调用父类的相应事件处理函数来实现默认操作。下面将这个例子再进行改进,看一下事件过滤器等其他方法获取事件的顺序。

(项目源码路径: src\06\6-2\myEvent) 在 mylineedit.h 文件中添加 public 函数声明:

```
bool event(QEvent * event);
```

然后在 mylineedit.cpp 文件中对该函数进行定义:

```
bool MyLineEdit::event(QEvent * event) //事件
{
    if(event -> type() == QEvent::KeyPress)
        qDebug() << tr("MyLineEdit 的 event() 函数");
    return QLineEdit::event(event); //执行 QLineEdit 类 event() 函数的默认操作
}
```

在 MyLineEdit 的 event() 函数中使用了 QEvent 的 type() 函数来获取事件的类型,如果是键盘按下事件 QEvent::KeyPress,则输出信息。因为 event() 函数具有 bool

型的返回值,所以在该函数的最后要使用 return 语句,这里一般是返回父类的 event() 函数的操作结果。下面进入 widget.h 文件中进行 public 函数的声明:

```
bool eventFilter(QObject *obj, QEvent *event);
```

然后到 widget.cpp 文件中的构造函数的最后添上一行代码:

```
lineEdit->installEventFilter(this); //在 Widget 上为 lineEdit 安装事件过滤器
```

下面是事件过滤器函数的定义:

```
bool Widget::eventFilter(QObject *obj, QEvent *event) //事件过滤器
{
    if(obj == lineEdit){ //如果是 lineEdit 部件上的事件
        if(event->type() == QEvent::KeyPress)
            qDebug() << tr("Widget 的事件过滤器");
    }
    return QWidget::eventFilter(obj, event);
}
```

在事件过滤器中,先判断该事件的对象是不是 lineEdit,如果是,再判断事件类型。最后返回了 QWidget 类默认的事件过滤器的执行结果。现在可以运行程序,然后按一下键盘上的任意键,比如这里按下 A 键,执行效果如图 6-2 所示。

可以看到,事件的传递顺序是这样的:先是事件过滤器,然后是该部件的 event() 函数,最后是该部件的事件处理函数,如图 6-3 所示。这里还要注意, event() 函数和事件处理函数,是在该部件内进行重新定义的,而事件过滤器却是在该部件的父部件中进行定义的。

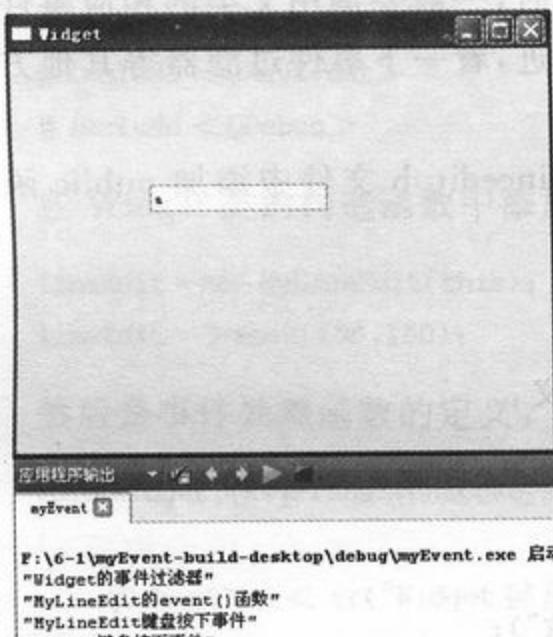


图 6-2 各个函数获取事件顺序运行效果

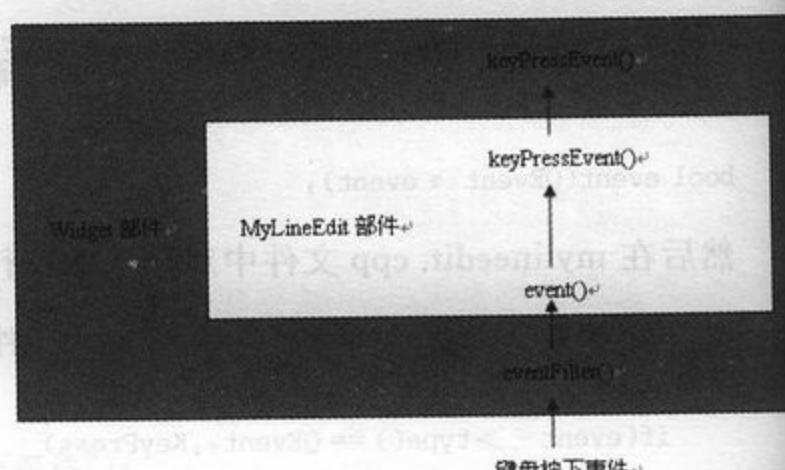


图 6-3 事件传递顺序示意图

6.2 鼠标事件和滚轮事件

QMouseEvent 类用来表示一个鼠标事件,当在窗口部件中按下鼠标或者移动鼠标指针时,都会产生鼠标事件。利用 QMouseEvent 类可以获知鼠标是哪个键按下了、鼠标指针的当前位置等信息。通常是重定义部件的鼠标事件处理函数来进行一些自定义的操作。QWheelEvent 类用来表示鼠标滚轮事件,在这个类中主要是获取滚轮移动的方向和距离。下面来看一个实际的例子,这个例子要实现的效果是:可以在界面上按着鼠标左键来拖动窗口,双击来使其全屏,按着鼠标右键则使指针变为一个自定义的图片,而使用滚轮则可以放大或者缩小编辑器中的内容。

(项目源码路径: src\06\6-3\myMouseEvent)新建 Qt Gui 应用,项目名称为 myMouseEvent,基类选择 QWidget,然后类名保持 Widget 不变。在设计模式中向界面上拖入一个 Text Edit。然后在 widget.h 文件中进行 protected 函数声明:

```
protected:
    void mousePressEvent(QMouseEvent * event);
    void mouseReleaseEvent(QMouseEvent * event);
    void mouseDoubleClickEvent(QMouseEvent * event);
    void mouseMoveEvent(QMouseEvent * event);
    void wheelEvent(QWheelEvent * event);
```

再在 private 中定义一个位置变量:

```
QPoint offset; //用来储存鼠标指针位置与窗口位置的差值
```

然后到 widget.cpp 文件中,添加头文件 #include <QMouseEvent>,并在构造函数中添加代码:

```
QCursor cursor; //创建光标对象
cursor.setShape(Qt::OpenHandCursor); //设置光标形状
setCursor(cursor); //使用光标
```

这几行代码可以使鼠标指针进入窗口后便改为小手掌形状,Qt 中提供了常用的鼠标指针的形状,可以在帮助中查看 Qt::CursorShape 关键字。下面是几个事件处理函数的定义:

```
void Widget::mousePressEvent(QMouseEvent * event) //鼠标按下事件
{
    if(event->button() == Qt::LeftButton){ //如果是鼠标左键按下
        QCursor cursor;
        cursor.setShape(Qt::ClosedHandCursor);
        QApplication::setOverrideCursor(cursor); //使鼠标指针暂时改变形状
        offset = event->globalPos() - pos(); //获取指针位置和窗口位置的差值
    }
}
```

```
else if(event ->button() == Qt::RightButton){ //如果是鼠标右键按下
    QCursor cursor(QPixmap("../yafeilinux.png"));
    QApplication::setOverrideCursor(cursor); //使用自定义的图片作为鼠标指针
}
}
```

在鼠标按下事件处理函数中,先判断是哪个按键按下,如果是鼠标左键,那么就更改指针的形状,并且存储当前指针位置与窗口位置的差值。这里使用了 globalPos() 函数来获取鼠标指针的位置,这个位置是指针在桌面上的位置,因为窗口的位置就是指它在桌面上的位置。另外,还可以使用 QMouseEvent 类的 pos() 函数获取鼠标指针在窗口中的位置。如果是鼠标右键按下,那么就将指针显示为我们自己的图片。

```
void Widget::mouseMoveEvent(QMouseEvent * event) //鼠标移动事件
{
    if(event ->buttons() & Qt::LeftButton){ //这里必须使用 buttons()
        QPoint temp;
        temp = event ->globalPos() - offset;
        move(temp); //使用鼠标指针当前位置减去差值,就得到了窗口应该移动的位置
    }
}
```

在鼠标移动事件处理函数中,先判断是否是鼠标左键按下,如果是,那么就使用前面获取的差值来重新设置窗口的位置。因为鼠标移动时会检测所有按下的键,而这时使用 QMouseEvent 的 button() 函数无法获取哪个按键被按下,只能使用 buttons() 函数,所以这里使用 buttons() 和 Qt::LeftButton 进行按位与的方法来判断是否是鼠标左键按下。

```
void Widget::mouseReleaseEvent(QMouseEvent * event) //鼠标释放事件
{
    QApplication::restoreOverrideCursor(); //恢复鼠标指针形状
}
```

在鼠标释放函数中进行了恢复鼠标形状的操作,这里使用的 restoreOverrideCursor() 函数要和前面的 setOverrideCursor() 函数配合使用。

```
void Widget::mouseDoubleClickEvent(QMouseEvent * event) //鼠标双击事件
{
    if(event ->button() == Qt::LeftButton){ //如果是鼠标左键按下
        if(windowState() != Qt::FullScreen) //如果现在不是全屏
            setWindowState(Qt::FullScreen); //将窗口设置为全屏
        else setWindowState(Qt::WindowNoState); //否则恢复以前的大小
    }
}
```

在鼠标双击事件处理函数中使用 setWindowState() 函数来使窗口处于全屏状态或

者恢复以前的大小。

```
void Widget::wheelEvent(QWheelEvent * event)           //滚轮事件
{
    if(event->delta() > 0){                           //当滚轮远离使用者时
        ui->textEdit->zoomIn();                      //进行放大
    }else{                                              //当滚轮向使用者方向旋转时
        ui->textEdit->zoomOut();                     //进行缩小
    }
}
```

在滚轮事件处理函数中使用 QWheelEvent 类的 delta() 函数获取了滚轮移动的距离,每当滚轮旋转一下,默认的是 15° ,这时 delta() 函数就会返回 15×8 即整数 120。当滚轮向远离使用者的方向旋转时,返回正值;当向着靠近使用者的方向旋转时,返回负值。这样便可以利用这个函数的返回值来判断滚轮的移动方向,从而进行编辑器中内容的放大或者缩小操作。

程序中使用了图片,所以还要往项目文件夹中添加一张图片。这里还要说明一下,默认的是按下鼠标按键时移动鼠标,鼠标移动事件才会产生;如果想不按鼠标按键,也可以获取鼠标移动事件,那么就要在构造函数中添加下面一行代码:

```
setMouseTracking(true);          //设置鼠标跟踪
```

这样便会开启窗口部件的鼠标跟踪功能。

6.3 键盘事件

QKeyEvent 类用来描述一个键盘事件。当键盘按键被按下或者被释放时,键盘事件便会被发送给拥有键盘输入焦点的部件。QKeyEvent 的 key() 函数可以获取具体的按键,对于 Qt 中给定的所有按键,可以在帮助中查看 Qt::Key 关键字。需要特别说明的是,回车键在这里是 Qt::Key_Return;键盘上的一些修饰键,比如 Ctrl 和 Shift 等,这里需要使用 QKeyEvent 的 modifiers() 函数来获取,可以在帮助中使用 Qt::KeyboardModifier 关键字来查看所有的修饰键。下面通过例子来看一下它们具体的应用。

(项目源码路径: src\06\6-4\myKeyEvent) 新建 Qt Gui 应用,项目名称为 myKeyEvent,基类选择 QWidget,类名保持 Widget 不变。完成后在 widget.h 文件中添加函数声明:

```
protected:
    void keyPressEvent(QKeyEvent * event);
    void keyReleaseEvent(QKeyEvent * event);
```

再到 widget.cpp 文件中,添加头文件 #include <QKeyEvent>,然后是两个函数的定义:

```

void Widget::keyPressEvent(QKeyEvent * event)           //键盘按下事件
{
    if(event->modifiers() == Qt::ControlModifier){ //是否按下 Ctrl 键
        if(event->key() == Qt::Key_M)                //是否按下 M 键
            setWindowState(Qt::WindowMaximized);      //窗口最大化
    }
    else QWidget::keyPressEvent(event);
}

void Widget::keyReleaseEvent(QKeyEvent * event)         //按键释放事件
{
    //其他操作
}

```

这里使用了 Ctrl+M 键来使窗口最大化，在键盘按下事件处理函数中，先检测 Ctrl 键是否按下，如果是，那么再检测 M 键是否按下。

在上面的例子中，可以同时按下 Ctrl 键和 M 键来实现一定的操作，那么可不可以按下两个不同的普通按键来实现一定的操作呢？现在将上面的程序进行更改。

(项目源码路径：src\06\6-5\myKeyEvent) 在设计模式中向界面上拖放一个 Horizontal Line 部件，在属性栏中将它的 X、Y 坐标分别设置为 50、100，然后再拖入一个 Vertical Line 部件，将其 X、Y 坐标分别设置为 100、20，然后再拖入一个 Push Button，设置其 X、Y 坐标为 120、120，然后再更改其显示内容为“请按方向键”。在 widget.cpp 文件中先添加头文件 #include <QDebug>，然后在构造函数中添加一行代码：

```
setFocus();                                         //使主界面获得焦点
```

然后将两个事件处理函数的内容更改如下：

```

void Widget::keyPressEvent(QKeyEvent * event)           //键盘按下事件
{
    if(event->key() == Qt::Key_Up){                   //如果是向上方向键
        qDebug() << "press:" << event->isAutoRepeat(); //是否自动重复
    }
}

void Widget::keyReleaseEvent(QKeyEvent * event)         //按键释放事件
{
    if(event->key() == Qt::Key_Up){
        qDebug() << "release:" << event->isAutoRepeat();
        qDebug() << "up";
    }
}

```

这里在键盘按下事件处理函数和按键释放处理函数中分别输出了向上方向键是否自动重复的信息。运行程序，按一下键盘上的向上方向键便松开，然后再一直按着向上方向键不放开，查看 Qt Creator 应用程序输出栏中的信息。可以看到，如果只是按了

一下按键，那么便不会自动重复，但是，如果一直按着这个按键，那么它就会自动重复。所以要想实现两个普通按键同时按下，就要避免按键的自动重复。下面来实现这样的效果：按下向上方向键按钮上移，按下向左方向键，按钮左移，如果在按下向左方向键的同时又按下了向上方向键，那么按钮便向左上方移动。

首先在 `widget.h` 文件中定义两个 `private` 变量：

```
bool keyUp; //向上方向键按下的标志
bool keyLeft; //向左方向键按下的标志
bool move; //是否完成了一次移动
```

然后在 `widget.cpp` 文件中的构造函数里对这两个变量进行初始化：

```
keyUp = false; //初始化变量
keyLeft = false;
move = false;
```

然后将两个事件处理函数的内容更改如下：

```
void Widget::keyPressEvent(QKeyEvent * event) //键盘按下事件
{
    if (event ->key() == Qt::Key_Up) {
        if (event ->isAutoRepeat()) return; //按键重复时不做处理
        keyUp = true; //标记向上方向键已经按下
    }
    else if (event ->key() == Qt::Key_Left) {
        if (event ->isAutoRepeat()) return;
        keyLeft = true;
    }
}

void Widget::keyReleaseEvent(QKeyEvent * event) //按键释放事件
{
    if (event ->key() == Qt::Key_Up) {
        if (event ->isAutoRepeat()) return;
        keyUp = false; //释放按键后将标志设置为 false
        if (move) { //如果已经完成了移动
            move = false; //设置标志为 false
            return; //直接返回
        }
        if (keyLeft) { //如果向左方向键已经按下且没有释放
            ui ->pushButton ->move(30, 80); //斜移
            move = true; //标记已经移动
        } else { //否则直接上移
            ui ->pushButton ->move(120, 80);
        }
    }
}
```

```

else if (event ->key() == Qt::Key_Left) {
    if (event ->isAutoRepeat()) return;
    keyLeft = false;
    if (move) {
        move = false;
        return;
    }
    if (keyUp) {
        ui ->pushButton ->move(30,80);
        move = true;
    } else {
        ui ->pushButton ->move(30,120);
    }
}
else if (event ->key() == Qt::Key_Down) {
    ui ->pushButton ->move(120,120); //使用向下方向键来还原按钮的位置
}
}

```

这里先在键盘按下事件处理函数中对向上方向键和向左方向键是否按下做了标记，并且当它们自动重复时不做任何处理。然后在按键释放事件处理函数中分别对这两个按键的释放做了处理。大体过程是这样的：当按下向左方向键时，在键盘按下事件处理函数中便会标记 keyLeft 为真，此时若又按下了向上方向键，那么 keyUp 也标记为真。先放开向上方向键，在按键释放事件处理函数中会标记 keyUp 为假，因为此时 keyLeft 为真，所以进行斜移，并且将已经移动标志 move 标记为真。此时再释放向左方向键，在按键释放事件处理函数中会标记 keyLeft 为假，因为已经进行了斜移操作，move 此时为真，所以这里不再进行操作，将 move 标记为假。这样就完成了整个斜移操作，而且所有的标志又恢复到了操作前的状态。这个程序只是提供一种思路，并不是实现这种操作的最好办法，因为这里按键的自动重复功能被忽略了。

6.4 定时器事件与随机数

QTimerEvent 类用来描述一个定时器事件。对于一个 QObject 的子类，只需要使用 int QObject::startTimer (int interval) 函数来开启一个定时器，这个函数需要输入一个以毫秒为单位的整数作为参数来表明设定的时间，它返回一个整型编号来代表这个定时器。当定时器溢出时就可以在 timerEvent() 函数中获取该定时器的编号来进行相关操作。

其实编程中更多的是使用 QTimer 类来实现一个定时器，它提供了更高层次的编程接口，比如可以使用信号和槽，还可以设置只运行一次的定时器。所以在以后的章节中，如果使用定时器，那么一般都是使用的 QTimer 类。关于定时器的介绍，可以在帮

助中查看一下 Timers 关键字。

关于随机数,在 Qt 中是使用 qrand()和 qsrand()两个函数实现的。下面在具体的程序中来讲解这些知识点。

(项目源码路径: src\06\6-6\myTimerEvent)新建 Qt Gui 应用,将项目名称更改为 myTimerEvent,基类选择 QWidget,然后类名保持 Widget 不变。完成后首先在 widget.h 文件中添加函数声明:

```
protected:  
void timerEvent(QTimerEvent * event);
```

然后再定义几个 private 私有变量:

```
int id1, id2, id3;
```

在 widget.cpp 文件中添加头文件 #include <QDebug>,然后在构造函数中添加代码:

```
id1 = startTimer(1000); //开启一个 1 秒定时器,返回其 ID  
id2 = startTimer(2000);  
id3 = startTimer(3000);
```

因为 startTimer()函数的参数是以毫秒为单位的,这里使用 1000,所以是 1 秒,程序中获取了各个定时器的编号。下面是定时器事件处理函数的定义:

```
void Widget::timerEvent(QTimerEvent * event)  
{  
    if (event->timerId() == id1) { //判断是哪个定时器  
        qDebug() << "timer1";  
    }  
    else if (event->timerId() == id2) {  
        qDebug() << "timer2";  
    }  
    else {  
        qDebug() << "timer3";  
    }  
}
```

这里使用 QTimerEvent 的 timerId()函数来获取定时器的编号,然后判断是哪一个定时器并分别进行不同的操作。现在运行程序,并看一下应用程序输出栏中的信息。下面来使用 QTimer 类实现一个简单的电子表。

(项目源码路径: src\06\6-7\myTimerEvent)继续在上面的程序中添加内容。先在设计模式中往界面上添加一个 LCD Number 部件。再到 widget.h 文件中添加私有槽声明:

```
private slots:  
    void timerUpdate();
```

在 widget.cpp 文件中添加头文件：

```
#include <QTimer>  
#include <QTime>
```

然后在构造函数中继续添加代码：

```
QTimer *timer = new QTimer(this);           // 创建一个新的定时器  
// 关联定时器的溢出信号到槽上  
connect(timer, SIGNAL(timeout()), this, SLOT(timerUpdate()));  
timer->start(1000);                      // 设置溢出时间为 1 秒，并启动定时器
```

下面是定时器溢出信号槽函数的定义：

```
void Widget::timerUpdate()                  // 定时器溢出处理  
{  
    QTime time = QTime::currentTime();      // 获取当前时间  
    QString text = time.toString("hh:mm"); // 转换为字符串  
    if((time.second() % 2) == 0) text[2] = ' '; // 每隔一秒就将“：“显示为空格  
    ui->lcdNumber->display(text);  
}
```

这里在构造函数中开启了一个 1 秒的定时器，当它溢出时就会发射 timeout() 信号，这时就会执行定时器溢出处理函数。在槽里获取了当前的时间，并且将它转换为可以显示的字符串，然后使用 QTime 类的 second() 函数获取秒的值，再将它与 2 进行取余操作，如果为 0 就让时与分之间的“：“变为空格，这样便实现了每隔一秒闪烁一下的效果。如果想停止一个定时器，可以调用它的 stop() 函数。

下面再来看一下随机数的使用。首先在 widget.cpp 文件中的构造函数里添加一行代码：

```
qrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
```

然后在 timerUpdate() 函数里面添加如下代码：

```
int rand = qrand() % 300;                // 产生 300 以内的正整数  
ui->lcdNumber->move(rand, rand);
```

在使用 qrand() 函数产生随机数之前，一般要使用 qrand() 函数为其设置初值，如果不设置初值，那么每次运行程序，qrand() 都会产生相同的一组随机数。为了每次运行程序时，都可以产生不同的随机数，要使用 qrand() 设置一个不同的初值。这里使用了 QTime 类的 secsTo() 函数，它表示两个时间点之间所包含的秒数，比如代码中就是指从零点整到当前时间所经过的秒数。当使用 qrand() 要获取一个范围内的数值时，一般是让它与一个整数取余，比如这里与 300 取余，就会使所有生成的数值在 0~

299 之间(包含 0 和 299)。这时运行程序,可以看到 LCD Number 部件每隔一秒便移动一个随机的位置。

在 QTimer 类中还有一个 singleShot() 函数来开启一个只运行一次的定时器,下面使用这个函数,让程序运行 10 秒后自动关闭。在 widget.cpp 文件中的构造函数里添加如下一行代码:

```
QTimer::singleShot(10000, this, SLOT(close()));
```

这里将时间设置为 10 秒,当溢出时便调用窗口部件的 close() 函数来关闭窗口。可以运行一下程序,等待 10 秒,程序会自动退出。

6.5 事件过滤器与事件的发送

Qt 提供了事件过滤器来在一个部件中监控其他多个部件的事件。事件过滤器与其他部件不同,它不是一个类,只是由两个函数组成的一种操作,用来完成一个部件对其他部件的事件的监视。这两个函数分别是 installEventFilter() 和 eventFilter(),都是 QObject 类中的函数。下面通过具体的例子来讲解。

(项目源码路径: src\06\6-8\myEventFilter)新建 Qt Gui 应用,将项目名称更改为 myEventFilter,基类选择 QWidget,类名保持 Widget 不变。完成后在设计模式中向界面上拖入一个 Text Edit 和一个 Spin Box。在 widget.h 文件中添加 public 函数声明:

```
bool eventFilter(QObject *obj, QEvent *event);
```

然后在 widget.cpp 文件中添加头文件:

```
#include <QKeyEvent>
#include <QWheelEvent>
```

在构造函数中添加代码:

```
ui->textEdit->installEventFilter(this); //为编辑器部件在本窗口上安装事件过滤器
ui->spinBox->installEventFilter(this);
```

要对一个部件使用事件过滤器,那么就要先使用该部件的 installEventFilter() 函数为该部件安装事件过滤器,这个函数的参数表明了监视对象。比如这里就是为 textEdit 部件和 spinBox 部件安装了事件过滤器,其参数 this 表明要在本部件即 Widget 中监视 textEdit 和 spinBox 的事件。这样,就需要重新实现 Widget 类的 eventFilter() 函数,在其中截获并处理两个子部件的事件。

```
bool Widget::eventFilter(QObject *obj, QEvent *event) //事件过滤器
{
    if (obj == ui->textEdit) { //判断部件
        if (event->type() == QEvent::Wheel) { //判断事件
            //将 event 强制转换为发生的事件的类型
        }
    }
}
```

```

QWheelEvent * wheelEvent = static_cast<QWheelEvent*>(event);
if (wheelEvent->delta() > 0) ui->textEdit->zoomIn();
else ui->textEdit->zoomOut();
return true; //该事件已经被处理
} else {
    return false; //如果是其他事件,可以进行进一步的处理
}
}

else if (obj == ui->spinBox) {
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent * keyEvent = static_cast<QKeyEvent*>(event);
        if (keyEvent->key() == Qt::Key_Space) {
            ui->spinBox->setValue(0);
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}

else return QWidget::eventFilter(obj, event);
}

```

在这个事件过滤器中先判断部件的类型,然后再判断事件的类型,如果是我们需要的事件就将其进行强制类型转换,然后进行相应的处理。这里需要说明,如果要对一个特定的事件进行处理,而且不希望它在后面的传递过程中再被处理,那么就返回 true,否则返回 false。这个函数中实现了在 textEdit 部件中使用滚轮进行内容的放大或缩小,在 spinBox 部件中使用空格来使数值设置为 0。

可以看到,使用事件过滤器可以很容易地处理多个部件的多个事件,如果不使用它,我们就得分别子类化各个部件,然后重新实现它们对应的各个事件处理函数,那样就会很麻烦了。

Qt 也提供了发送一个事件的功能,由 QApplication 类的 sendEvent() 函数或者 postEvent() 函数来实现。这两个函数的主要区别是: sendEvent() 会立即处理给定的事件,而 postEvent() 则会将事件放到等待调度队列中,当下一次 Qt 的主事件循环运行时才会处理它。这两个函数还有其他一些区别,比如 sendEvent() 中的 QEvent 对象参数在事件发送完成后无法自动删除,所以需要在栈上创建 QEvent 对象;而 postEvent() 中的 QEvent 对象参数必须在堆上进行创建(例如使用 new),当事件被发送后事件队列会自动删除它。

下面在 widget.cpp 文件中的构造函数里添加代码来向 spinBox 部件发送一个向上方向键被按下的事件:

```
QKeyEvent myEvent(QEvent::KeyPress, Qt::Key_Up, Qt::NoModifier);
qApp ->sendEvent(ui ->spinBox, &myEvent); //发送键盘事件到 spinBox 部件
```

这里使用了 sendEvent() 函数, 其中 QKeyEvent 类对象是在栈上创建的。其中的 qApp 是 QApplication 对象的全局指针, 每一个应用程序中只能使用一个 QApplication 对象, 这里等价于使用 QApplication::sendEvent()。现在运行程序可以发现 spinBox 部件中初始值变为了 1, 这说明已经在这个部件上按下了向上方向键。Qt 演示程序中的 Input Panel 程序就使用了事件的发送功能, 它在 Tools 分类下。

在 Qt 中还可以使用自定义的事件, 这个需要继承 QEvent 类, 可以查看帮助中 The Event System 关键字的相关内容。

6.6 小结

这一章主要讲解了 Qt 中事件的应用, 读者要掌握基本事件的处理方法, 包括重新实现事件处理函数和使用事件过滤器。这一章还涉及了定时器和随机数的知识, 它们在实现一些特殊效果以及动画、游戏中会经常使用到, 所以也希望读者掌握。

第7章

Qt 对象模型与容器类

这一章将学习 Qt 中的一些核心机制,它们是构成 Qt 的基础,包括对象模型、信号和槽、属性系统、对象树与拥有权、元对象系统等。这一章的后半部分将学习容器类(Container Classes)的相关内容,还会涉及 QString、QByteArray、QVariant 和正则表达式的使用等相关内容。

7.1 对象模型

标准 C++ 对象模型可以在运行时非常有效地支持对象范式(object paradigm),但是它的静态特性在一些问题领域中不够灵活。图形用户界面编程不仅需要运行时的高效性,还需要高度的灵活性。为此,Qt 在标准 C++ 对象模型的基础上添加了一些特性,形成了自己的对象模型。这些特性有:

- 一个强大的无缝对象通信机制——信号和槽(signals and slots);
- 可查询和可设计的对象属性系统(object properties);
- 强大的事件和事件过滤器(events and event filters);
- 通过上下文进行国际化的字符串翻译机制(string translation for internationalization);
- 完善的定时器(timers)驱动,使得可以在一个事件驱动的 GUI 中处理多个任务;
- 分层结构的、可查询的对象树(object trees),它使用一种很自然的方式来组织对象拥有权(object ownership);
- 守卫指针即 QPointer,它在引用对象被销毁时自动将其设置为 0;
- 动态的对象转换机制(dynamic cast)。

Qt 的这些特性都是在遵循标准 C++ 规范内实现的,使用这些特性都必须继承自 QObject 类。其中对象通信机制和动态属性系统还需要元对象系统(Meta-Object System)的支持。关于对象模型,可以在帮助中查看 Object Model 关键字。

7.1.1 信号和槽

前面的章节中已经多次用到过信号和槽了,这一小节主要讲解理论知识,也可以在帮助中查看 Signals & Slots 关键字。

1. 信号和槽机制

信号和槽用于两个对象之间的通信,信号和槽机制是 Qt 的核心特征,也是 Qt 不同于其他开发框架的最突出特征。在 GUI 编程中,当改变了一个部件时,总希望其他部件也能了解到该变化。更一般来说,我们希望任何对象都可以和其他对象进行通信。例如,如果用户单击了关闭按钮,则希望可以执行窗口的 close() 函数来关闭窗口。为了实现对象间的通信,一些工具包中使用了回调(callback)机制,而在 Qt 中使用了信号和槽来进行对象间的通信。当一个特殊的事情发生时便可以发射一个信号,比如按钮被单击;而槽就是一个函数,它在信号发射后被调用来响应这个信号。Qt 的部件类中已经定义了一些信号和槽,但是更多的做法是子类化这个部件,然后添加自己的信号和槽来实现想要的功能。

提示

回调就是指向函数的指针,把这个回调函数指针传递给要被处理的函数,那么就可以在这个函数被处理时在适当的地方调用这个回调函数。

回调机制主要有两个缺陷:第一,不是类型安全的(type-safe),不能保证在调用回调函数时可以使用正确的参数;第二,是强耦合的。处理函数必须知道调用哪个回调函数。

前面使用过的信号和槽的关联都是一个信号对应一个槽。其实,一个信号可以关联到多个槽上,多个信号也可以关联到同一个槽上,甚至,一个信号还可以关联到另一个信号上,如图 7-1 所示。如果存在多个槽与某个信号相关联,那么,当这个信号被发射时,这些槽将会一个接一个地执行,但是执行的顺序是随机的,无法指定它们的执行顺序。

下面通过一个简单的例子来进一步讲解信号和槽的相关知识。这个例子实现的效果是:在主界面中创建一个对话框,在这个对话框中可以输入数值,当单击“确定”按钮时关闭对话框并且将输入的数值通过信号发射出去,而在主界面中接收该信号并且显示数值。程序运行效果如图 7-2 所示。

(项目源码路径: src\07\7-1\mySignalSlot)新建 Qt Gui 应用,项目名称为 mySignalSlot,基类选择 QWidget,然后类名保持 Widget 不变。项目建立完成后,向项目中添加新文件,模板选择 Qt 分类中的“Qt 设计师界面类”,界面模板选择 Dialog without Buttons,类名为 MyDialog。完成后首先在 mydialog.h 文件中添加代码来声明一个信号:

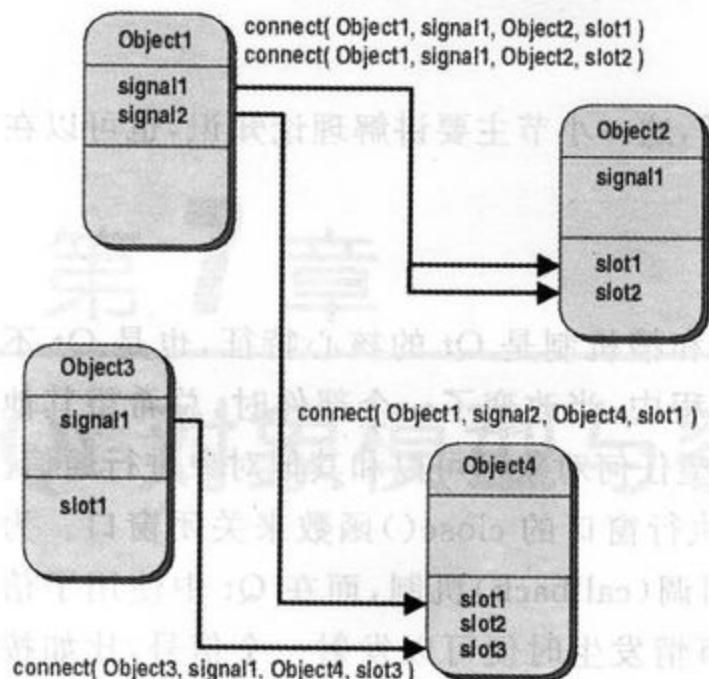


图 7-1 对象间信号和槽的关联图

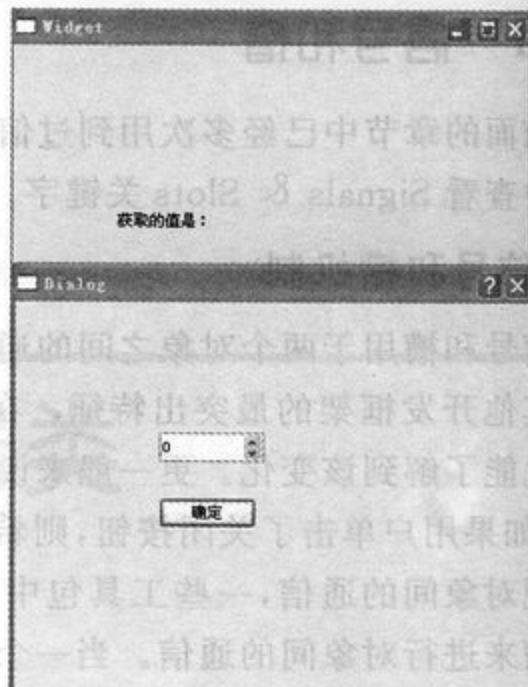


图 7-2 设计信号和槽运行效果

```
signals:
```

```
void dlgReturn(int); //自定义的信号
```

声明一个信号要使用 `signals` 关键字, 在 `signals` 前面不能使用 `public`、`private` 和 `protected` 等限定符, 因为只有定义该信号的类及其子类才可以发射该信号。而且信号只用声明, 不需要也不能对它进行定义实现。还要注意, 信号没有返回值, 只能是 `void` 类型的。因为只有 `QObject` 类及其子类派生的类才能使用信号和槽机制, 这里的 `MyDialog` 类继承自 `QDialog` 类, `QDialog` 类又继承自 `QWidget` 类, `QWidget` 类是 `QObject` 类的子类, 所以这里可以使用信号和槽。不过, 使用信号和槽还必须在类声明的最开始处添加 `Q_OBJECT` 宏, 在这个程序中, 类的声明是自动生成的, 已经添加了这个宏。

在 `mydialog.ui` 对应的界面中添加一个 `Spin Box` 部件和 `Push Button` 部件, 将 `pushButton` 的显示文本改为“确定”。然后转到 `pushButton` 的单击信号 `clicked()` 槽, 更改如下:

```
void MyDialog::on_pushButton_clicked() //确定按钮
{
    int value = ui->spinBox->value(); //获取输入的数值
    emit dlgReturn(value); //发射信号
    close(); //关闭对话框
}
```

单击“确定”按钮便获取 `spinBox` 部件中的数值, 然后使用自定义的信号将其作为参数发射出去。发射一个信号要使用 `emit` 关键字, 例如程序中发射了 `dlgReturn()` 信号。

然后到 `widget.h` 文件中添加自定义槽的声明:

```
private slots:
    void showValue(int value);
```

声明一个槽需要使用 slots 关键字。一个槽可以是 private、public 或者 protected 类型的，槽也可以声明为虚函数，这与普通的成员函数是一样的，也可以像调用一个普通函数一样来调用槽。槽的最大特点就是可以和信号关联。

下面打开 widget.ui 文件，向界面上拖入一个 Label 部件，更改其文本为“获取的值是：”。然后进入 widget.cpp 文件中添加头文件 #include "mydialog.h"，再在构造函数中添加代码：

```
MyDialog * dlg = new MyDialog(this);
//将对话框中的自定义信号与主界面中的自定义槽进行关联
connect(dlg, SIGNAL(dlgReturn(int)), this, SLOT(showValue(int)));
dlg->show();
```

这里创建了一个 MyDialog，并且使用 Widget 作为父部件。然后将 MyDialog 类的 dlgReturn() 信号与 Widget 类的 showValue() 槽进行关联。信号和槽进行关联使用的是 QObject 类的 connect() 函数，这个函数的原型如下：

```
bool QObject::connect ( const QObject * sender, const char * signal, const QObject * receiver, const char * method, Qt::ConnectionType type = Qt::AutoConnection )
```

它的第一个参数为发送信号的对象，例如这里的 dlg；第二个参数是要发送的信号，这里是 SIGNAL(dlgReturn(int))；第三个参数是接收信号的对象，这里是 this，表明是本部件，即 Widget，当这个参数为 this 时，也可以将这个参数省略掉，因为 connect() 函数还有另外一个重载形式，该参数默认为 this；第四个参数是要执行的槽，这里是 SLOT(showValue(int))。对于信号和槽，必须使用 SIGNAL() 和 SLOT() 宏，它们可以将其参数转化为 const char * 类型。connect() 函数的返回值为 bool 类型，当关联成功时返回 true。还要注意，在调用这个函数时信号和槽的参数只能有类型，不能有变量，例如写成 SLOT(showValue(int value)) 是不对的。对于信号和槽的参数问题，基本原则是信号中的参数类型要和槽中的参数类型相对应，而且信号中的参数可以多于槽中的参数，但是不能反过来；如果信号中有多余的参数，那么它们将被忽略。connect() 函数的最后一个参数表明了关联的方式，默认值是 Qt::AutoConnection，这里还有其他几个选择，具体功能如表 7-1 所列。编程中一般使用默认值，例如这里，在 MyDialog 类中使用 emit 发射了信号之后就会执行槽，只有等槽执行完了才会执行 emit 语句后面的代码。也可以将这个参数改为 Qt::QueuedConnection，这样在执行完 emit 语句后便立即执行其后面的代码，而不管槽是否已经执行。不再使用这个关联时，还可以使用 disconnect() 函数来断开关联。

表 7-1 信号和槽关联类型表

常量	描述
Qt::AutoConnection	如果信号和槽在不同的线程中, 同 Qt::QueuedConnection; 如果信号和槽在同一个线程中, 同 Qt::DirectConnection
Qt::DirectConnection	发射完信号后立即执行槽, 只有槽执行完成返回后, 发射信号处后面的代码才可以执行
Qt::QueuedConnection	接收部件所在线程的事件循环返回后再执行槽, 无论槽执行与否, 发射信号处后面的代码都会立即执行
Qt::BlockingQueuedConnection	类似 Qt::QueuedConnection, 只能用在信号和槽在不同的线程的情况下
Qt::UniqueConnection	类似 Qt::AutoConnection, 但是两个对象间的相同的信号和槽只能有唯一的关联
Qt::AutoCompatConnection	类似 Qt::AutoConnection, 它是 Qt 3 中的默认类型

下面是自定义槽的实现, 这里只是简单地将参数传递来的数值显示在标签上。因为这里使用了中文, 所以还要在 main.cpp 文件中添加相关代码。

```
void Widget::showValue(int value)           // 自定义槽
{
    ui->label->setText(tr("获取的值是: %1").arg(value));
}
```

这个程序自定义了信号和槽, 可以看到它们的使用是很简单的, 只需要对它们进行关联, 然后在适当的时候发射信号就行。下面列举一下使用信号和槽应该注意的几点:

- 需要继承自 QObject 或其子类;
- 在类声明的最开始添加 Q_OBJECT 宏;
- 槽中的参数类型要和信号的参数的类型相对应, 且不能比信号的参数多;
- 信号只用声明, 没有定义, 且返回值为 void 类型。

2. 信号和槽的自动关联

信号和槽还有一种自动关联方式, 在第 2 章已经提到过了。例如上面程序中在设计模式直接生成的“确定”按钮的单击信号的槽, 就是使用的这种方式: on_pushButtonClicked()由 on、部件的 objectName 和信号 3 部分组成, 中间用下划线隔开。这样组织的名称的槽就可以直接和信号关联, 而不用再使用 connect() 函数。不过使用这种方式还要进行其他设置, 而前面之所以可以直接使用, 是因为程序中默认已经设置了。可以回头看一下第 2 章讲解的 ui_helloworld.h 文件的内容, 其中的 connectSlotsByName() 函数就是用来支持信号和槽的自动关联的, 它是使用对象名(objectName)来实现的。下面来看一个简单的例子。

(项目源码路径: src\07\7-2\mySignalSlot2) 新建 Qt Gui 应用, 项目名称为

mySignalSlot2，基类选择 QWidget，然后类名保持 Widget 不变。完成后先在 widget.h 文件中进行函数声明：

```
private slots:
    void on_myButton_clicked();
```

这里自定义了一个槽，它使用自动关联。然后在 widget.cpp 文件中添加头文件 #include <QPushButton>，再将构造函数的内容更改如下：

```
Widget::Widget(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::Widget)

{
    QPushButton * button = new QPushButton(this); // 创建按钮
    button->setObjectName("myButton");           // 指定按钮的对象名
    ui->setupUi(this);                          // 要在定义了部件以后再调用这个函数
}
```

因为 setupUi() 函数中调用了 connectSlotsByName() 函数，所以要使用自动关联的部件的定义都要放在 setupUi() 函数之前，还必须使用 setObjectName() 函数指定它们的 objectName，只有这样才能正常使用自动关联。下面是槽的定义：

```
void Widget::on_myButton_clicked()           // 使用自动关联
{
    close();
}
```

这里进行了关闭部件的操作。对于槽的函数名，中间要使用前面指定的 objectName，这里是 myButton。运行程序，单击按钮，发现可以正常关闭窗口。

可以看到，如果要使用信号和槽的自动关联，就必须在 connectSlotsByName() 函数之前进行部件的定义，而且还要指定部件的 objectName，并且自动关联中必须使用 Qt 中已经定义的信号，而不能是自定义的信号。鉴于这些约束，虽然自动关联形式上很简单，但是实际编写代码时却很少使用。而且，在定义一个部件时，很希望明确地使用 connect() 函数来对其进行信号和槽的关联，这样当别人看到这个部件定义时，就可以知道和它相关的信号和槽的关联了，而使用自动关联却没有这么明了。

3. 信号和槽的高级应用

有时希望获得信号发送者的信息，Qt 提供了 QObject::sender() 函数来返回发送该信号的对象的指针。但是如果多个信号关联到了同一个槽上，而在该槽中需要对每一个信号进行不同的处理，使用上面的方法就很麻烦了。对于这种情况，便可以使用 QSignalMapper 类。QSignalMapper 可以叫作信号映射器，可以实现对多个相同部件的相同信号进行映射，为其添加字符串或者数值参数，然后再发射出去。对于这个类的使用，可以参考《Qt 及 Qt Quick 开发实战精解》的 1.3.3 小节。Qt 演示程序中 Tools

分类下的 Input Panel 示例程序也使用了这个类,也可以作为参考。

信号和槽机制的特色和优越性:

- 信号和槽机制是类型安全的,相关联的信号和槽的参数必须匹配;
- 信号和槽是松耦合的,信号发送者不知道也不知道接受者的信息;
- 信号和槽可以使用任意类型的任意数量的参数。

虽然信号和槽机制提供了高度的灵活性,但就其性能而言,还是慢于回调机制的。当然,这点性能差异通常在一个应用程序中是很难体现出来的。

7.1.2 属性系统

Qt 提供了强大的基于元对象系统的属性系统,可以在能够运行 Qt 的平台上支持任意的标准 C++ 编译器。要声明一个属性,那么该类必须继承自 QObject 类,而且要在声明前使用 Q_PROPERTY() 宏:

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

其中 type 表示属性的类型,它可以是 QVariant 支持的类型或者是用户自定义的类型。而如果是枚举类型,还需要使用 Q_ENUMS() 宏在元对象系统中进行注册,这样以后才可以使用 QObject::setProperty() 函数来使用该属性。name 就是属性的名称。READ 后面是读取该属性的函数,这个函数是必须有的,而后面带有“[]”号的选项表示这些函数是可选的。一个属性类似于一个数据成员,不过它添加了一些可以通过元对象系统访问的附加功能:

- 一个读(READ)访问函数。该函数是必须有的,用来读取属性的取值。这个函数一般是 const 类型的,返回值类型必须是该属性的类型,或者是该属性类型的指针或者引用。
- 一个可选的写(WRITE)访问函数,用来设置属性的值。这个函数必须只有一个参数,而且它的返回值必须为空 void。
- 一个可选的重置(RESET)函数,用来将属性恢复到一个默认的值。这个函数不能有参数,而且返回值必须为空 void。
- 一个可选的通知(NOTIFY)信号。如果使用该选项,那么每当属性的值改变时都要发射一个指定的信号。

- 可选的 DESIGNABLE 表明这个属性在 GUI 设计器(例如 Qt Designer)的属性编辑器中是否可见。大多数属性的该值为 true, 即可见。
- 可选的 SCRIPTABLE 表明这个属性是否可以被脚本引擎(scripting engine)访问, 默认值为 true。
- 可选的 STORED 表明是否在当对象的状态被存储时也必须存储这个属性的值, 大部分属性的该值为 true。
- 可选的 USER 表明这个属性是否被设计为该类的面向用户或者用户可编辑的属性。一般, 每一个类中只有一个 USER 属性, 默认值为 false。
- 可选的 CONSTANT 表明这个属性的值是一个常量。对于给定的一个对象实例, 每一次使用常量属性的 READ 方法都必须返回相同的值, 但对于不同的实例, 这个常量可以不同。一个常量属性不可以有 WRITE 方法和 NOTIFY 信号。
- 可选的 FINAL 表明这个属性不能被派生类重写。

其中的 READ、WRITE 和 RESET 函数可以被继承, 也可以是虚的(virtual), 当在多继承时, 它们必须继承自第一个父类。关于这一节的内容, 可以在帮助中参考 The Property System 关键字, 下面来看一个具体的例子。

(项目源码路径: src\07\7-3\myProperty)新建 Qt Gui 应用, 项目名称为 myProperty, 基类选择 QWidget, 类名保持 Widget 不变。当建立好项目后, 向其中添加新文件, 模板选择 C++ 类, 类名为 MyClass, 基类选择 QObject, 类型信息选择“继承自 QObject”。添加完新文件后, 到 MyClass.h 文件中更改 MyClass 类的定义如下:

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString userName READ getUserName WRITE setUserName
               NOTIFY userNameChanged); //注册属性 userName
public:
    explicit MyClass(QObject *parent = 0);
    QString getUserName() const           //实现 READ 读函数
    { return m_userName; }
    void setUserName(QString userName)   //实现 WRITE 写函数
    {
        m_userName = userName;
        emit userNameChanged(userName); //当属性值改变时发射该信号
    }
signals:
    void userNameChanged(QString);      //声明 NOTIFY 通知消息
private:
    QString m_userName;                //私有变量, 存放 userName 属性的值
};
```

这里使用 Q_PROPERTY() 宏向元对象系统注册了属性 `userName`, 然后声明了几个相应的函数, 因为读/写函数都很简单, 所以在声明时直接进行了定义。下面来看一下在类外对这个属性的使用。

首先在 `widget.h` 文件中添加一个私有槽的声明:

```
private slots:
    void userChanged(QString);
```

然后到 `widget.cpp` 文件中, 先添加头文件:

```
#include "myclass.h"
#include <QDebug>
```

然后在构造函数中添加代码:

```
MyClass * my = new MyClass(this); // 创建 MyClass 类实例
connect(my, SIGNAL(userNameChanged(QString)), this, SLOT(userChanged(QString)));
my ->setUserName("yafei"); // 设置属性的值
qDebug() << "userName:" << my ->getUserName(); // 输出属性的值
// 使用 QObject 类的 setProperty() 函数设置属性的值
my ->setProperty("userName", "linux");
// 输出属性的值, 这里使用了 QObject 类的 property() 函数, 它返回值类型为 QVariant
qDebug() << "userName:" << my ->property("userName").toString();
```

这里创建了 `MyClass` 类的实例, 然后进行了 `userName` 属性的写入与读取, 这里有两种方法: 一种是直接调用该属性的相关函数; 另一种是使用 `QObject` 类的 `setProperty()` 函数和 `property()` 函数, 使用这两个函数要指定属性名。`property()` 函数的返回值类型为 `QVariant`, 可以使用这个类的 `toString()` 函数将其转换为 `QString` 类型的数据。下面是处理属性值变化的槽的定义:

```
void Widget::userChanged(QString userName)
{
    qDebug() << "user changed:" << userName;
}
```

这里只是简单地将 `userName` 输出。

使用 `QObject` 类的 `setProperty()` 函数还可以设置动态属性, 只需要将属性名设置为一个类中没有的属性即可。比如在 `MyClass` 类外为其添加动态属性 `myValue`, 可以在 `widget.cpp` 文件中构造函数的最后面添加如下代码:

```
my ->setProperty("myValue", 10); // 动态属性, 只对该实例有效
qDebug() << "myValue:" << my ->property("myValue").toInt();
```

需要说明, 这样添加的动态属性只对实例 `my` 有效, 对于 `MyClass` 类的其他对象没有作用。

这里使用 Q_PROPERTY() 宏向元对象系统注册了属性 `userName`, 然后声明了几个相应的函数, 因为读/写函数都很简单, 所以在声明时直接进行了定义。下面来看一下在类外对这个属性的使用。

首先在 `widget.h` 文件中添加一个私有槽的声明:

```
private slots:
    void userChanged(QString);
```

然后到 `widget.cpp` 文件中, 先添加头文件:

```
#include "myclass.h"
#include <QDebug>
```

然后在构造函数中添加代码:

```
MyClass * my = new MyClass(this); // 创建 MyClass 类实例
connect(my, SIGNAL(userNameChanged(QString)), this, SLOT(userChanged(QString)));
my->setUserName("yafei"); // 设置属性的值
qDebug() << "userName:" << my->getUserName(); // 输出属性的值
// 使用 QObject 类的 setProperty() 函数设置属性的值
my->setProperty("userName", "linux");
// 输出属性的值, 这里使用了 QObject 类的 property() 函数, 它返回值类型为 QVariant
qDebug() << "userName:" << my->property("userName").toString();
```

这里创建了 `MyClass` 类的实例, 然后进行了 `userName` 属性的写入与读取, 这里有两种方法: 一种是直接调用该属性的相关函数; 另一种是使用 `QObject` 类的 `setProperty()` 函数和 `property()` 函数, 使用这两个函数要指定属性名。`property()` 函数的返回值类型为 `QVariant`, 可以使用这个类的 `toString()` 函数将其转换为 `QString` 类型的数据。下面是处理属性值变化的槽的定义:

```
void Widget::userChanged(QString userName)
{
    qDebug() << "user changed:" << userName;
}
```

这里只是简单地将 `userName` 输出。

使用 `QObject` 类的 `setProperty()` 函数还可以设置动态属性, 只需要将属性名设置为一个类中没有的属性即可。比如在 `MyClass` 类外为其添加动态属性 `myValue`, 可以在 `widget.cpp` 文件中构造函数的最后面添加如下代码:

```
my->setProperty("myValue", 10); // 动态属性, 只对该实例有效
qDebug() << "myValue:" << my->property("myValue").toInt();
```

需要说明, 这样添加的动态属性只对实例 `my` 有效, 对于 `MyClass` 类的其他对象没有作用。

7.1.3 对象树与拥有权

Qt 中使用对象树(object tree)来组织和管理所有的 QObject 类及其子类的对象。当创建一个 QObject 时,如果使用了其他的对象作为其父对象(parent),那么这个 QObject 就会被添加到父对象的 children() 列表中,这样当父对象被销毁时,这个 QObject 也会被销毁。实践表明,这个机制非常适合于管理 GUI 对象。例如,一个 QShortcut(键盘快捷键)对象是相应窗口的一个子对象,所以当用户关闭了这个窗口时,这个快捷键也可以被销毁。

QWidget 作为能够在屏幕上显示的所有部件的基类,扩展了对象间的父子关系。一个子对象一般也就是一个子部件,因为它们要显示在父部件的区域之中。例如,当关闭一个消息对话框(message box)后要销毁它时,消息对话框中的按钮和标签也会被销毁,这也正是我们所希望的,因为按钮和标签是消息对话框的子部件。当然,也可以自己来销毁一个子对象。关于这一部分内容,可以在帮助索引中查看 Object Trees & Ownership 关键字。

在第 3 章中讲解第一个例子时,曾经提到了使用 new 来创建一个部件,但是却没有使用 delete 来进行释放的问题。这里再来研究一下这个问题。

(项目源码路径: src\07\7-4\myOwnership) 新建 Qt Gui 应用,项目名称为 myOwnership,基类选择 QWidget,然后类名保持 Widget 不变。完成后向项目中添加新文件,模板选择 C++ 类,类名为 MyButton,基类为 QPushButton,类型信息选择“继承自 QWidget”。添加完文件后在 mybutton.h 文件中添加析构函数的声明:

```
~MyButton();
```

然后到 mybutton.cpp 文件中添加头文件 #include <QDebug> 并定义析构函数:

```
MyButton::~MyButton()
```

```
{
```

```
    qDebug() << "delete button";
```

```
}
```

这样当 MyButton 的对象被销毁时,就会输出相应的信息。这里定义析构函数,只是为了更清楚地看到部件的销毁过程,其实一般在构建新类时不需要实现析构函数。下面在 widget.cpp 文件中进行更改,添加头文件:

```
#include "mybutton.h"
#include <QDebug>
```

在构造函数中添加代码:

```
MyButton *button = new MyButton(this); // 创建按钮部件,指定 widget 为父部件
button->setText(tr("button"));
```

更改析构函数：

```
Widget::~Widget()
{
    delete ui;
    qDebug() << "delete widget";
}
```

Widget 类的析构函数中默认已经有了销毁 ui 的语句,这里又添加了输出语句。当 Widget 窗口被销毁时,将输出信息。下面运行程序,然后关闭窗口,在 Qt Creator 的应用程序输出栏中的输出信息为:

```
delete widget
delete button
```

可以看到,当关闭窗口后,因为该窗口是顶层窗口,所以应用程序要销毁该窗口部件(如果不是顶层窗口,那么关闭时只是隐藏,不会被销毁),而当窗口部件销毁时会自动销毁其子部件。这也就是为什么在 Qt 中经常只看到 new 操作而看不到 delete 操作的原因。再来看一下 main.cpp 文件,其中 Widget 对象是建立在栈上的:

```
Widget w;
w.show();
```

这样对于对象 w,在关闭程序时会自动销毁。而对于 Widget 中的部件,如果是在堆上创建(使用 new 操作符),那么只要指定 Widget 为其父窗口就可以了,也不需要进行 delete 操作。整个应用程序关闭时会销毁 w 对象,而此时又会自动销毁它的所有子部件,这些都是 Qt 的对象树所完成的。

所以,对于规范的 Qt 程序,我们要在 main() 函数中将主窗口部件创建在栈上,例如“Widget w;”,而不要在堆上进行创建(使用 new 操作符)。对于其他窗口部件,可以使用 new 操作符在堆上进行创建,不过一定要指定其父部件,这样就不需要再使用 delete 操作符来销毁该对象了。

还有一种重定义父部件(reparented)的情况,例如,将一个包含其他部件的布局管理器应用到窗口上,那么该布局管理器和其中的所有部件都会自动将它们的父部件转换为该窗口部件。在 widget.cpp 文件中添加头文件 #include <QHBoxLayout>,然后在构造函数中继续添加代码:

```
MyButton *button2 = new MyButton;
MyButton *button3 = new MyButton;
QHBoxLayout *layout = new QHBoxLayout;
layout ->addWidget(button2);
layout ->addWidget(button3);
setLayout(layout); //在该窗口中使用布局管理器
```

这里创建了两个 MyButton 和一个水平布局管理器,但是并没有指定它们的父部件,现在各个部件的拥有权(ownership)不是很清楚。但是当使用布局管理器来管理这两个按钮,并且在窗口中使用这个布局管理器后,这两个按钮和水平布局管理器都将重定义父部件而成为窗口 Widget 的子部件。可以使用 children() 函数来获取一个部件的所有子部件的列表,例如在构造函数中再添加如下代码:

```
qDebug() << children(); //输出所有子部件的列表
```

这时可以运行程序查看应用程序输出栏中的信息,然后根据自己的想法更改程序来进一步体会 Qt 中对象树的概念。

7.1.4 元对象系统

Qt 中的元对象系统(Meta-Object System)提供了对象间通信的信号和槽机制、运行时类型信息和动态属性系统。元对象系统是基于以下 3 个条件的:

- 该类必须继承自 QObject 类;
- 必须在类的私有声明区声明 Q_OBJECT 宏(在类定义时,如果没有指定 public 或者 private,则默认为 private);
- 元对象编译器 Meta-Object Compiler(moc),为 QObject 的子类实现元对象特性提供必要的代码。

其中,moc 工具读取一个 C++ 源文件,如果它发现一个或者多个类的声明中包含有 Q_OBJECT 宏,便会另外创建一个 C++ 源文件(就是在项目目录中的 debug 目录下看到的以 moc 开头的 C++ 源文件),其中包含了为每一个类生成的元对象代码。这些产生的源文件或者被包含进类的源文件中,或者和类的实现同时进行编译和链接。

元对象系统主要是为了实现信号和槽机制才被引入的,不过除了信号和槽机制以外,元对象系统还提供了其他一些特性:

- QObject::metaObject() 函数可以返回一个类的元对象,它是 QMetaObject 类的对象;
- QMetaObject::className() 可以在运行时以字符串形式返回类名,而不需要 C++ 编辑器原生的运行时类型信息(RTTI)的支持;
- QObject::inherits() 函数返回一个对象是否是 QObject 继承树上一个类的实例的信息;
- QObject::tr() 和 QObject::trUtf8() 进行字符串翻译来实现国际化;
- QObject::setProperty() 和 QObject::property() 通过名字来动态设置或者获取对象属性;
- QMetaObject::newInstance() 构造该类的一个新实例。

除了这些特性,还可以使用 qobject_cast() 函数来对 QObject 类进行动态类型转换,这个函数的功能类似于标准 C++ 中的 dynamic_cast() 函数,但它不再需要 RTTI

的支持。这个函数尝试将它的参数转换为尖括号中的类型的指针,如果是正确的类型则返回一个非零的指针,如果类型不兼容则返回 0。例如:

```
QObject * obj = new MyWidget;
QWidget * widget = qobject_cast<QWidget *>(obj);
```

这个函数已经在多个章节中用到过了,这里也不再进行过多讲述。信号和槽机制是 Qt 的核心内容,而信号和槽机制必须依赖于元对象系统,所以它是 Qt 中很关键的内容。这里只是说明了它的一些应用,关于它的具体实现机制,这里不再讲述,而对于一本入门教材,那些知识就显得太过深入了。作为初学者,上面讲述的知识显得有些枯燥,读者也没有必要一次就把它搞得很明白,心中有个大概的印象就行了,可以等以后有了一定的基础之后再来学习。关于元对象系统的知识,可以在 Qt 中查看 The Meta-Object System 关键字。

7.2 容器类

Qt 库提供了一组通用的基于模板的容器类(container classes)。这些容器类可以用来存储指定类型的项目(items),例如,如果需要一个 QString 类型的可变大小的数据组,那么可以使用 QVector。在《C++ Primer》一书中作者就强力推荐使用 vector 类型和迭代器来取代一般的数组,除非 vector 无法达到必要的速度要求时才使用数组。与 STL(Standard Template Library,C++ 的标准模板库)中的容器类相比,Qt 中的这些容器类更轻量,更安全,更容易使用。如果不熟悉 STL 或者更喜欢使用 Qt way 来进行编程,那么就可以使用这些容器类来代替 STL 的类。对应本节内容可以在帮助中参考 Container Classes Introduction 关键字。

7.2.1 Qt 的容器类简介

Qt 提供了一些顺序容器: QList、QLinkedList、 QVector、QStack 和 QQueue。因为这些容器中的数据都是一个接一个线性存储的,所以称为顺序容器。对于大多数应用程序而言,使用最多的,而且最好用的是 QList,虽然它是作为一个数组列表,但是它在头部和尾部进行添加操作是很快速的。如果需要使用一个链表,那么可以使用 QLinkedList;如果希望数据项可以占用连续的内存空间,可以使用 QVector。而 QStack 和 QQueue 作为便捷类,分别提供了后进先出(LIFO)和先进先出(FIFO)语义。

Qt 还提供了一些关联容器: QMap、QMultiMap、QHash、QMultiHash 和 QSet。因为这些容器存储的是<键,值>对,比如 QMap<Key, T>,所以称为关联容器。其中“Multi”容器用来方便支持一个键多个值的情况。表 7-2 对常用的容器类进行了介绍。

表 7-2 常用容器类简介表

类	简介
QList<T>	这是目前最常用的容器类。它存储了给定类型值的一个列表,而这些值可以通过索引访问。在内部,QList 使用数组来实现,以确保进行快速的基于索引的访问。可以使用 QList::append() 和 QList::prepend() 在列表的两端添加项目,也可以使用 QList::insert() 在列表的中间插入项目。相对于其他容器类,为了扩展到尽可能少的可执行代码,QList 被高度优化了。常用的 QStringList 继承自 QList<QString>
QLinkedList<T>	除了使用迭代器而不使用整数索引进行项目访问外,它基本与 QList 相同。当在一个很大的列表的中间插入项目时,QLinkedList 比 QList 提供了更好的性能,而且它拥有更好的迭代器语义(当迭代器指向 QLinkedList 的一个项目后,只要这个项目还存在,那么迭代器就依然有效;而当迭代器指向 QList 中的一个项目后,如果 QList 进行了插入或者删除操作,那么这个迭代器就无效了)
QVector<T>	它在内存的相邻位置存储给定类型的值的一个数组。在 vector 的前面或者中间插入项目是非常缓慢的,因为这样可能导致大量的项目要在内存中移动一个位置
QStack<T>	它是 QVector 的一个便捷子类,提供了后进先出(LIFO)语义。它添加了 push()、pop() 和 top() 等函数
QQueue<T>	它是 QList 的一个便捷子类,提供了先进先出(FIFO)语义。它添加了 enqueue()、dequeue() 和 head() 等函数
QSet<T>	它提供了一个快速查询单值的数学集
QMap<Key, T>	它提供了一个字典(关联数组),将 Key 类型的键值映射到 T 类型的值上。一般每一个键关联一个单一的值。 QMap 使用键顺序来存储它的数据;如果不关心存储顺序,那么可以使用 QHash 来代替它,因为 QHash 更快速
QMultiMap<Key, T>	它是 QMap 的一个便捷类,提供了实现多值映射的方便的接口函数,例如一个键可以关联多个值
QHash<Key, T>	它与 QMap 拥有基本相同的接口,但是它的查找速度更快。QHash 的数据是以任意的顺序存储的
QMultiHash<Key, T>	它是 QHash 的一个便捷类,提供了实现多值散列的方便的接口函数

下面分别对最为常用的 QList 和 QMap 进行介绍,而对于其他几个容器,读者可以参照着进行操作,因为它们的接口函数是很相似的,当然也可以参考帮助手册。

(项目源码路径: src\07\7-5\myContainers)新建 Qt 4 控制台应用,项目名称为 myContainers。这里只是为了演示容器类的使用,所以没有使用图形界面,这样只需要建立控制台程序就可以了。将 main.cpp 文件更改如下:

```
#include <QtCore/QCoreApplication>
#include <QList>
#include <QDebug>
```

```

int main(int argc, char * argv[])
{
    QApplication a(argc, argv);
    QList<QString> list;
    list << "aa" << "bb" << "cc";      //插入项目
    if(list[1] == "bb") list[1] = "ab";
    list.replace(2,"bc");           //将“cc”换为“bc”
    qDebug() << "the list is: ";      //输出整个列表
    for(int i = 0; i<list.size(); + + i){
        qDebug() << list.at(i);      //现在列表为 aa ab bc
    }
    list.append("dd");              //在列表尾部添加
    list.prepend("mm");            //在列表头部添加
    QString str = list.takeAt(2);   //从列表中删除第 3 个项目，并获取它
    qDebug() << "at(2) item is: " << str;
    qDebug() << "the list is: ";
    for(int i = 0; i<list.size(); + + i)
    {
        qDebug() << list.at(i);      //现在列表为 mm aa bc dd
    }
    list.insert(2,"mm");           //在位置 2 插入项目
    list.swap(1,3);               //交换项目 1 和项目 3
    qDebug() << "the list is: ";
    for(int i = 0; i<list.size(); + + i)
    {
        qDebug() << list.at(i);      //现在列表为 mm bc mm aa dd
    }
    qDebug() << "contains mm ?" << list.contains("mm"); //列表中是否包含“mm”
    qDebug() << "the mm count: " << list.count("mm");   //包含“mm”的个数
    //第一个“mm”的位置，默认从位置 0 开始往前查找，返回第一个匹配的项目的位置
    qDebug() << "the first mm index: " << list.indexOf("mm");
    //第二个“mm”的位置，我们指定从位置 1 开始往前查找
    qDebug() << "the second mm index: " << list.indexOf("mm",1);
    return a.exec();
}

```

QList 是一个模板类，提供了一个列表。QList<T>实际上是一个 T 类型项目的指针数组，所以支持基于索引的访问，而且当项目的数目小于 1 000 时，可以实现在列表中间进行快速的插入操作。QList 提供了很多方便的接口函数来操作列表中的项目，例如插入操作 insert()、替换操作 replace()、移除操作 removeAt()、移动操作 move()、交换操作 swap()、在表尾添加项目 append()、在表头添加项目 prepend()、移除第一个项目 removeFirst()、移除最后一个项目 removeLast()、从列表中移除一项并获取

这个项目 takeAt() 及相应的 takeFirst() 和 takeLast()、获取一个项目的索引 indexOf()、判断是否含有相应的项目 contains() 以及获取一个项目出现的次数 count() 等。对于 QList，可以使用“<<”操作符来向列表中插入项目，也可以使用“[]”操作符通过索引来访问一个项目，其中项目是从 0 开始编号的。不过，对于只读的访问，另一种方法是使用 at() 函数，比“[]”操作符要快很多。上面的程序中使用了一些常用的函数，很简单所以没有分开讲。读者可以不用一下子把整个程序都写出来再运行，而是写一部分就运行一次并查看结果。

(项目源码路径：src\07\7-6\myContainers2) 新建 Qt 4 控制台应用，项目名称为 myContainers2。下面来看一下 QMap 的具体应用，将 main.cpp 文件更改如下：

```
#include <QtCore/QCoreApplication>
#include <QMap>
#include <QMultiMap>
#include <QDebug>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QMap<QString, int> map;
    map["one"] = 1; //向 map 中插入("one",1)
    map["three"] = 3;
    map.insert("seven", 7); //使用 insert() 函数进行插入
    //获取键的值，使用“[ ]”操作符时，如果 map 中没有该键，那么会自动插入
    int value1 = map["six"];
    qDebug() << "value1:" << value1;
    qDebug() << "contains six ?" << map.contains("six");
    //使用 value() 函数获取键的值，这样当键不存在时不会自动插入
    int value2 = map.value("five");
    qDebug() << "value2:" << value2;
    qDebug() << "contains five ?" << map.contains("five");
    //当键不存在时，value() 默认返回 0，这里可以设定该值，比如这里设置为 9
    int value3 = map.value("nine", 9);
    qDebug() << "value3:" << value3;
    //map 默认是一个键对应一个值，如果重新给该键设置了值，那么以前的会被擦除
    map.insert("ten", 10);
    map.insert("ten", 100);
    qDebug() << "ten:" << map.value("ten");
    //可以使用 insertMulti() 函数来实现一键多值，然后使用 values() 函数来获取值的列表
```

```

map.insertMulti("two", 2);
map.insertMulti("two", 4);
QList<int> values = map.values("two");
qDebug() << "two: " << values;

//也可以使用 QMultiMap 类来实现一键多值
QMultiMap<QString, int> map1, map2, map3;
map1.insert("values", 1);
map1.insert("values", 2);
map2.insert("values", 3);
//可以进行相加,这样 map3 的“values”键将包含 3,2,1 三个值
map3 = map1 + map2;
QList<int> myValues = map3.values("values");
qDebug() << "the values are: ";
for (int i = 0; i < myValues.size(); ++i) {
    qDebug() << myValues.at(i);
}
return a.exec();
}

```

QMap 类是一个容器类,提供了一个基于跳跃列表的字典(a skip-list-based dictionary)。 QMap<Key, T> 是 Qt 的通用容器类之一,它存储(键,值)对并提供了与键相关的值的快速查找。 QMap 中提供了很多方便的接口函数,例如插入操作 insert()、获取值 value()、是否包含一个键 contains()、删除一个键 remove()、删除一个键并获取该键对应的值 take()、清空操作 clear()、插入一键多值 insertMulti() 等。可以使用“[]”操作符插入一个键值对或者获取一个键的值,不过当使用该操作符获取一个不存在的键的值时,会默认向 map 中插入该键;为了避免这个情况,可以使用 value() 函数来获取键的值。当使用 value() 函数时,如果指定的键不存在,那么默认会返回 0,可以在使用该函数时提供参数来更改这个默认返回的值。 QMap 默认是一个键对应一个值的,但是也可以使用 insertMulti() 进行一键多值的插入,对于一键多值的情况,更方便的是使用 QMap 的子类 QMultiMap。

容器也可以嵌套使用,例如 QMap<QString, QList<int> >, 这里键的类型是 QString, 而值的类型是 QList<int>, 需要注意,在后面的“>>”符号之间要有一个空格,不然编译器会将它当作“>>”操作符对待。在各种容器中存储的值类型可以是任何的可赋值的数据类型,该类型需要有一个默认的构造函数、一个拷贝构造函数和一个赋值操作运算符,像基本的类型 double, 指针类型、Qt 的数据类型(如 QString、QDate、QTime)等。但是 QObject 以及 QObject 的子类都不能存储在容器中,不过,可以存储这些类的指针,例如 QList<QWidget * >。也可以自定义这样的数据类型,具体方法可以参考 Container Classes Introduction 文档中的相关内容。

7.2.2 遍历容器

遍历一个容器可以使用迭代器(iterators)来完成,迭代器提供了一个统一的方法来访问容器中的项目。Qt的容器类提供了两种类型的迭代器:Java风格迭代器和STL风格迭代器。如果只是想按顺序遍历一个容器中的项目,那么还可以使用Qt的foreach关键字。

1. Java风格迭代器

Java风格迭代器在使用时比STL风格迭代器要方便很多,但是在性能上稍微弱于后者。每一个容器类都有两个Java风格迭代器数据类型:一个提供只读访问,另一个提供读/写访问,如表7-3所列。

表7-3 Java风格迭代器

容器	只读迭代器	读/写迭代器
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

下面将以QList和 QMap为例来进行讲解,而QLinkedList、 QVector 和 QSet与QList的迭代器拥有极其相似的接口;类似的,QHash与QMap的迭代器拥有相同的接口。

(项目源码路径:src\07\7-7\myIterators)新建Qt4控制台应用,项目名称为myIterators。然后将main.cpp文件更改如下:

```
#include <QtCore/QCoreApplication>
#include <QList>
#include <QListIterator>
#include <QMutableListIterator>
#include <QDebug>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QList<QString> list;
```

```

list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list);      //创建列表的只读迭代器,将 list 作为参数
qDebug() << "the forward is :";
while (i.hasNext())                  //正向遍历列表,结果为 A,B,C,D
    qDebug() << i.next();
qDebug() << "the backward is :";
while (i.hasPrevious())              //反向遍历列表,结果为 D,C,B,A
    qDebug() << i.previous();
return a.exec();
}

```

这里先创建了一个 QList 列表 list,然后使用 list 作为参数创建了一个列表的只读迭代器。这时,迭代器指向列表第一个项目的前面(这里是指向项目“A”的前面)。然后使用 hasNext() 函数来检查在该迭代器后面是否还有项目,如果还有项目,那么使用 next() 来跳过这个项目,next() 函数会返回它所跳过的项目。当正向遍历结束后,迭代器会指向列表最后一个项目的后面,这时可以使用 hasPrevious() 和 previous() 来进行反向遍历。可以看到,Java 风格迭代器是指向项目之间的,而不是直接指向项目。所以,迭代器或者指向容器的最前面,或者指向两个项目之间,或者指向容器的最后面,如图 7-3 所示。

在上面程序中的 previous() 函数和 next() 函数的效果如图 7-4 所示。QListIterator 还有一些常用的 API,如表 7-4 所列。

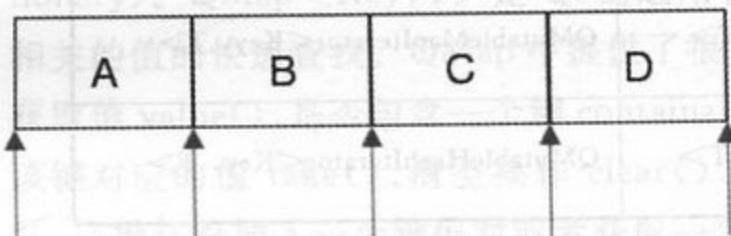


图 7-3 Java 风格迭代器的有效位置

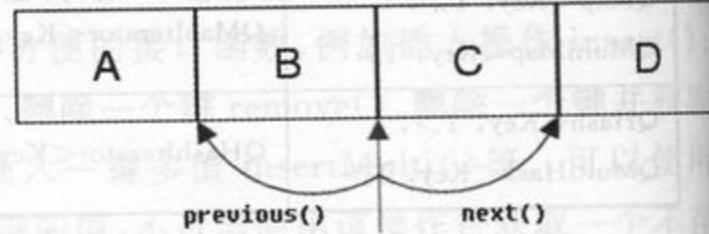


图 7-4 Java 风格迭代器的函数效果

表 7-4 QListIterator 常用 API

函 数	行 为
toFront()	将迭代器移动到列表的最前面(第一个项目之前)
toBack()	将迭代器移动到列表的最后面(最后一个项目之后)
hasNext()	如果迭代器没有到达列表的最后面,那么返回 true
next()	返回下一个项目,并且使迭代器前移一个位置
peekNext()	返回下一个项目,但是不移动迭代器
hasPrevious()	如果迭代器没有到达列表的最前面,那么返回 true
previous()	返回前一个项目,并且使迭代器往回移动一个位置
peekPrevious()	返回前一个项目,但是不移动迭代器

QListIterator 没有提供向列表中插入或者删除项目的函数,要完成这些功能就必须使用 QMutableListIterator。这个类增加了 insert() 函数来完成插入操作; remove() 函数完成删除操作; setValue() 函数完成设置值操作。在前面的程序中“return a.exec();”前添加如下代码:

```

QMutableListIterator<QString> j(list);
j.toBack();                                //返回列表尾部
while (j.hasPrevious()) {
    QString str = j.previous();
    if(str == "B") j.remove();               //删除项目“B”
}
j.insert("Q");                            //在列表最前面添加项目“Q”
j.toBack();
if(j.hasPrevious()) j.previous() = "N";     //直接赋值
j.previous();
j.setValue("M");                          //使用 setValue()进行赋值
j.toFront();
qDebug() << "the forward is :";
while (j.hasNext())                      //正向遍历列表,结果为 Q,A,M,N
    qDebug() << j.next();

```

可以使用 remove() 函数来删除上一次跳过的项目,使用 insert() 函数在迭代器指向的位置插入一个项目,这时迭代器会位于添加的项目之后,比如这里添加“Q”后,迭代器指向“Q”和“A”之间。使用 QMutableListIterator 类的 next() 和 previous() 等函数时会返回列表中项目的一个非 const 引用,所以可以直接对其赋值,当然也可以使用 setValue() 函数进行赋值,这个函数是对上一次跳过的项目进行赋值的。除了上面讲到的这些函数外,还有 findNext() 和 findPrevious() 函数可以用来实现项目的查找。现在运行一下程序,运行结果已经在上面的程序中注释出来了。

与 QListIterator 类似,QMapIterator 提供了 toFront()、toBack()、hasNext()、next()、peekNext()、hasPrevious()、previous() 和 peekPrevious() 等函数。可以在 next()、peekNext()、previous() 和 peekPrevious() 等函数返回的对象上分别使用 key() 和 value() 函数来获取键和值。

(项目源码路径: src\07\7-8\myIterators2) 新建 Qt 4 控制台应用,项目名称为 myIterators2。然后将 main.cpp 文件更改如下:

```

#include <QtCore/QCoreApplication>
#include <QMapIterator>
#include <QMutableMapIterator>
#include <QDebug>

int main(int argc, char *argv[])

```

```

QCoreApplication a(argc, argv);

 QMap<QString, QString> map;
 map.insert("Paris", "France");
 map.insert("Guatemala City", "Guatemala");
 map.insert("Mexico City", "Mexico");
 map.insert("Moscow", "Russia");

 QMapIterator<QString,QString> i(map);
 while(i.hasNext()) {
     i.next();
     qDebug() << i.key() << ":" << i.value();
 }

 if(i.findPrevious("Mexico")) qDebug() << "find Mexico"; //向前查找键的值
 QMutableMapIterator<QString, QString> j(map);
 while(j.hasNext()) {
     if (j.next().key().endsWith("City")) //endsWith()是QString类的函数
         j.remove(); //删除含有"City"结尾的键的项目
 }
 while(j.hasPrevious()) {
     j.previous(); //现在的键值对为(Paris,France),(Moscow,Russia)
     qDebug() << j.key() << ":" << j.value();
 }
 return a.exec();
}

```

这里在 QMap 中存储了一些(首都,国家)键值对,然后删除了包含以“City”字符串结尾的键的项目。对于 QMap 的遍历,可以先使用 next() 函数,然后再使用 key() 和 value() 来获取键和值的信息。因为这里的很多函数与前面的用法相似,这里就不再过多讲解。现在运行一下程序,从遍历结果可以看到 QMap 是按照键的顺序来存储数据的,比如这里是按照键的字母顺序排列的。

2. STL 风格迭代器

STL 风格迭代器兼容 Qt 和 STL 的通用算法(generic algorithms),而且在速度上进行了优化。对于每一个容器类,都有两个 STL 风格迭代器类型:一个提供了只读访问,另一个提供了读/写访问,如表 7-5 所列。因为只读迭代器比读/写迭代器要快很多,所以应尽可能使用只读迭代器。

表 7-5 STL 风格迭代器

容 器	只读迭代器	读/写迭代器
QList<T>, QQueue<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QVector<T>, QStack<T>	QVector<T>::const_iterator	QVector<T>::iterator
QSet<T>	QSet<T>::const_iterator	QSet<T>::iterator
QMap<Key, T>, QMultiMap<Key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QHash<Key, T>, QMultiHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator

下面仍然以 QList 和 QMap 为例来进行相关内容的讲解。(项目源码路径: src\07\7-9\myIterators3)新建 Qt 4 控制台应用,项目名称为 myIterators3。然后将 main.cpp 文件更改如下:

```
#include <QtCore/QCoreApplication>
#include <QList>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<QString> list;
    list << "A" << "B" << "C" << "D";
    QList<QString>::iterator i; // 使用读/写迭代器
    qDebug() << "the forward is :";
    for (i = list.begin(); i != list.end(); ++i) {
        *i = (*i).toLower(); // 使用 QString 的 toLower() 函数转换为小写
        qDebug() << *i; // 结果为 a,b,c,d
    }
    qDebug() << "the backward is :";
    while (i != list.begin()) {
        --i;
        qDebug() << *i; // 结果为 d,c,b,a
    }
    QList<QString>::const_iterator j; // 使用只读迭代器
    qDebug() << "the forward is :";
    for (j = list.constBegin(); j != list.constEnd(); ++j)
```

```

    qDebug() << * j; //结果为 a,b,c,d
    return a.exec();
}

```

STL 风格迭代器的 API 模仿了数组的指针,例如,使用“`++`”操作符来向后移动迭代器使其指向下一个项目;使用“`*`”操作符返回迭代器指向的项目等。需要说明的是,不同于 Java 风格迭代器,STL 风格迭代器是直接指向项目的。其中一个容器的 `begin()` 函数返回了一个指向该容器中第一个项目的迭代器,`end()` 函数也返回一个迭代器,但是这个迭代器指向该容器的最后一个项目的下一个假想的虚项目,`end()` 标志着一个无效的位置,当列表为空时,`begin()` 函数等价于 `end()` 函数。STL 迭代器的有效位置如图 7-5 所示。

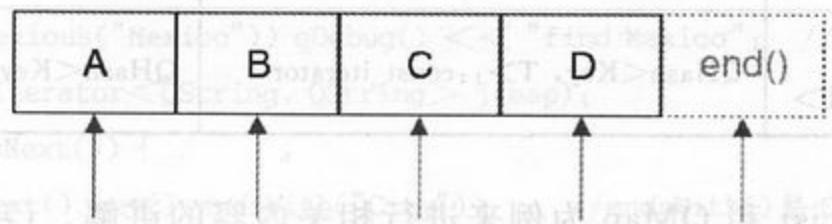


图 7-5 STL 风格迭代器有效位置

在 STL 风格迭代器中“`++`”和“`--`”操作符即可以作为前缀(`++i`,`--i`)操作符,也可以作为后缀(`i++`,`i--`)操作符。当作为前缀时会先修改迭代器,然后返回修改后的迭代器的一个引用;当作为后缀时,在修改迭代器以前会对其进行复制,然后返回这个复制。如果在表达式中不会对返回值进行处理,那么最好使用前缀操作符(`++i`,`--i`),这样会更快一些。对于非 `const` 迭代器类型,使用一元操作符“`*`”获得的返回值可以用在赋值运算符的左侧。STL 风格迭代器的常用 API 如表 7-6 所列。

表 7-6 STL 风格迭代器常用 API

表达式	行 为
<code>* i</code>	返回当前项目
<code>++i</code>	前移迭代器到下一个项目
<code>i += n</code>	使迭代器前移 n 个项目
<code>--i</code>	使迭代器往回移动一个项目
<code>i -= n</code>	使迭代器往回移动 n 个项目
<code>i - j</code>	返回迭代器 i 和迭代器 j 之间的项目的数目

对于 QMap,可以使用“`*`”操作符来返回一个项目,然后使用 `key()` 和 `value()` 来分别获取键和值。下面在前面的程序中先添加头文件 `#include <QMap>`,然后在 `main()` 函数的“`return a.exec();`”一行代码前添加如下代码:

```

QMap<QString, int> map;
map.insert("one",1);

```

```

map.insert("two",2);
map.insert("three",3);
QMap<QString, int>::const_iterator p;
qDebug() << "the forward is :";
for (p = map.constBegin(); p != map.constEnd(); ++p)
    qDebug() << p.key() << ":" << p.value(); //结果为(one,1),(three,3),(two,2)

```

这里创建了一个 QMap，然后使用 STL 风格的只读迭代器对其进行遍历，输出了其中所有项目的键和值。

3. foreach 关键字

foreach 是 Qt 向 C++ 语言中添加的一个用来进行容器的顺序遍历的关键字，使用预处理器来进行实施。下面来看一下具体应用的例子。

(项目源码路径：src\07\7-10\myForeach)新建 Qt 4 控制台应用，项目名称为 myForeach。然后将 main.cpp 文件更改如下：

```

#include <QtCore/QCoreApplication>
#include <QList>
#include <QMap>
#include <QMultiMap>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<QString> list;
    list.insert(0, "A");
    list.insert(1, "B");
    list.insert(2, "C");
    qDebug() << "the list is :";
    foreach (QString str, list) {           //从 list 中获取每一项
        qDebug() << str;                  //结果为 A,B,C
    }

    QMap<QString, int> map;
    map.insert("first", 1);
    map.insert("second", 2);
    map.insert("third", 3);
    qDebug() << endl << "the map is :";
    foreach (QString str, map.keys())       //从 map 中获取每一个键
        qDebug() << str << ":" << map.value(str);
    //输出键和对应的值，结果为(first,1),(second,2),(third,3)

    QMultiMap<QString, int> map2;
    map2.insert("first", 1);

```

```

map2.insert("first", 2);
map2.insert("first", 3);
map2.insert("second", 2);
qDebug() << endl << "the map2 is :";
QList<QString> keys = map2.uniqueKeys(); //返回所有键的列表
foreach (QString str, keys) { //遍历所有的键
    foreach (int i, map2.values(str)) //遍历键中所有的值
        qDebug() << str << ":" << i;
}//结果为(first,3),(first,2),(first,1),(second,2)
return a.exec();
}

```

上面的程序使用 `foreach` 关键字分别遍历了 `QList`、 `QMap` 和 `QMultiMap`，很简单所以不再讲解。需要说明的是，在 `foreach` 循环中也可以使用 `break` 和 `continue` 语句。可以看到，使用 `foreach` 关键字进行容器的遍历是很简单的，当不愿意使用迭代器时，那么就可以使用 `foreach` 来代替。

关于容器的相关内容就讲到这里，如果还希望了解各个容器类的时间复杂度、增长策略等内容，那么可以参考 Qt 帮助中 Container Classes Introduction 关键字的相关内容。

7.2.3 通用算法

在 `<QtAlgorithms>` 头文件中，Qt 提供了一些全局的模板函数，这些函数是可以使用在容器上的十分常用的算法。可以在任何提供了 STL 风格迭代器的容器类上使用这些算法，包括 `QList`、`QLinkedList`、 `QVector`、 `QMap` 和 `QHash`。如果在目标平台上可以使用 STL，那么可以使用 STL 的算法来代替 Qt 的这些算法，因为 STL 提供了更多的算法，而 Qt 只提供了其中最重要的一些算法。下面对其中几个常用的算法进行演示，可以在帮助索引中查看 `Generic Algorithms` 关键字来了解其他的算法。

(项目源码路径：src\07\7-11\myAlgorithms)新建 Qt4 控制台应用，项目名称为 myAlgorithms。然后将 `main.cpp` 文件更改如下：

```

#include <QtCore/QCoreApplication>
#include <QTextCodec>
#include <QVector>
#include <QStringList>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
}

```

```

QStringList list;
list << "one" << "two" << "three";

qDebug() << QObject::tr("qCopy()算法：");
QVector<QString> vect(3);
//将 list 中所有项目复制到 vect 中
qCopy(list.begin(), list.end(), vect.begin());
qDebug() << vect; //结果为 one,two,three

qDebug() << endl << QObject::tr("qEqual()算法：");
//从 list 的开始到结束的所有项目与 vect 的开始及其后面的等数量的项目进行比较
//全部相同则返回 true
bool ret1 = qEqual(list.begin(), list.end(), vect.begin());
qDebug() << "equal: " << ret1; //结果为 true

qDebug() << endl << QObject::tr("qFind()算法：");
//从 list 中查找"two",返回第一个对应的值的迭代器,如果没有找到则返回 end()
QList<QString>::iterator i = qFind(list.begin(), list.end(), "two");
qDebug() << *i; //结果为"two"

qDebug() << endl << QObject::tr("qFill()算法：");
//将 list 中的所有项目填充为"eleven"
qFill(list.begin(), list.end(), "eleven");
qDebug() << list; //结果 eleven,eleven,eleven

QList<int> list1;
list1 << 3 << 3 << 6 << 6 << 6 << 8;

qDebug() << endl << QObject::tr("qCount()算法：");
int countOf6 = 0;
qCount(list1.begin(), list1.end(), 6, countOf6); //查找 6 的个数
qDebug() << "countOf6: " << countOf6; //结果为 3

qDebug() << endl << QObject::tr("qLowerBound()算法：");
//返回第一个出现 5 的位置,如果没有 5,则返回 5 应该在的位置
//list1 被查找的范围中的项目必须是升序
QList<int>::iterator j = qLowerBound(list1.begin(), list1.end(), 5);
list1.insert(j, 5);
qDebug() << list1; //结果 3,3,5,6,6,6,8

QList<int> list2;
list2 << 33 << 12 << 68 << 6 << 12;

```

```

qDebug() << endl << QObject::tr("qSort()算法：");
//使用快速排序算法对 list2 进行升序排序，排序后两个 12 的位置不确定
qSort(list2.begin(), list2.end());
qDebug() << list2; //结果 6,12,12,33,68

qDebug() << endl << QObject::tr("qStableSort()算法：");
//使用一种稳定排序算法对 list2 进行升序排序
//排序前在前面的 12 排序后依然在前面
qStableSort(list2.begin(), list2.end());
qDebug() << list2; //结果 6,12,12,33,68

qDebug() << endl << QObject::tr("qGreater()算法：");
//可以在 qSort() 算法中使其反向排序
qSort(list2.begin(), list2.end(), qGreater<int>());
qDebug() << list2; //结果 68,33,12,12,6

qDebug() << endl << QObject::tr("qSwap()算法：");
double pi = 3.14;
double e = 2.71;
qSwap(pi, e); //交换 pi 和 e 的值
qDebug() << "pi:" << pi << "e:" << e; //结果 pi = 2.71, e = 3.14

return a.exec();
}

```

这些算法在一些数据处理等操作中是很有用的，应该对它们有一定的了解，以后遇到相关的问题时要能够想到使用它们。现在可以运行程序，运行结果已经注释在上面的程序中了。<QtGlobal>头文件中也提供了一些函数来实现一些经常使用的功能，比如 qAbs() 来获取绝对值、qBound() 来获取数值边界、qMax() 返回两个数中的最大值、qMin() 返回两个数中的最小值、qRound() 返回一个浮点数接近的整数值、还有以前讲到的与随机数相关的 qrand() 和 qsrand() 等。

7.2.4 QString

QString 类提供了一个 Unicode(Unicode 是一种支持大部分文字系统的国际字符编码标准)字符串。其实在第一个 Hello World 程序就用到了它，而几乎所有的程序都会使用到它，所以有必要更多地了解。QString 存储了一串 QChar，而 QChar 提供了一个 16 位的 Unicode 字符。在后台，QString 使用隐式共享(implicit sharing)来减少内存使用和避免不必要的数据拷贝，有助于减少存储 16 位字符的固有开销。对应于这一节的内容，可以参考 QString 类的帮助文档。

1. 隐式共享

隐式共享(Implicit Sharing)又称为写时复制(copy-on-write)。Qt 中很多 C++ 类

使用隐式数据共享来尽可能地提高资源使用率和减少复制操作。使用隐式共享类作为参数传递是既安全又有效的,因为只有一个指向该数据的指针被传递了,只有当函数向它写入时才会复制该数据。这里根据下面的几行代码进行讲解:

```
QPixmap p1, p2;
p1.load("image.bmp");
p2 = p1; //p1 与 p2 共享数据
QPainter paint;
paint.begin(&p2); //p2 被修改
paint.drawText(0,50, "Hi");
paint.end();
```

一个共享类由一个指向一个共享数据块的指针和数据组成,在共享数据块中包含了一个引用计数。当一个共享对象被建立时,会设置引用计数为 1,例如这里 QPixmap 类是一个隐式共享类,开始时 p1 和 p2 的引用计数都为 1。每当有新的对象引用了共享数据时引用计数都会递增,而当有对象不再引用这个共享数据时引用计数就会递减,当引用计数为 0 时,这个共享数据就会被销毁掉。例如这里执行了“p2=p1;”语句后,p2 便与 p1 共享同一个数据,这时 p1 的引用计数为 2,而 p2 的引用计数为 0,所以 p2 以前指向的数据结构将会被销毁掉。当处理共享对象时,有两种复制对象的方法:深拷贝(deep copy)和浅拷贝(shallow copy)。深拷贝意味着复制一个对象,而浅拷贝则是复制一个引用(仅仅是一个指向共享数据块的指针)。一个深拷贝是非常昂贵的,需要消耗很多的内存和 CPU 资源;而浅拷贝则非常快速,因为它只需要设置一个指针和增加引用计数的值。当隐式共享类使用“=”操作符时就是使用浅拷贝,如上面的“p2=p1;”语句。但是当一个对象被修改时,就必须进行一次深拷贝,比如上面程序中“paint.begin(&p2);”语句要对 p2 进行修改,这时就要对数据进行深拷贝,使 p2 和 p1 指向不同的数据结构,然后将 p1 的引用计数设为 1,p2 的引用计数也设为 1。

共享的好处是程序不需要进行不必要的数据复制,可以减少数据的拷贝和使用更少的内存,对象也可以很容易地被分配,或者作为参数被传递,或者从函数被返回。隐式共享在后台进行,在实际编程中不必去关注它。Qt 中主要的隐式共享类有: QByteArray、QCursor、QFont、QPixmap、QString、QUrl、QVariant 和所有的容器类等,要看所有的隐式共享类和隐式共享的其他内容,可以在帮助索引中查看 Implicit Sharing 关键字。

2. 编辑操作

(项目源码路径: src\07\7-12\myString)新建 Qt 4 控制台应用,项目名称为 myString。然后将 main.cpp 文件更改如下:

```
#include <QtCore/QCoreApplication>
#include <QTextCodec>
#include <QDebug>
#include <QStringList>
```

```

int main(int argc, char * argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    qDebug() << QOBJECT::tr("以下是编辑字符串操作:") << endl;

    QString str = "hello";
    qDebug() << QOBJECT::tr("字符串大小:") << str.size(); // 大小为 5
    str[0] = QChar('H'); // 将第一个字符换为 'H'
    qDebug() << QOBJECT::tr("第一个字符:") << str[0]; // 结果为 'H'
    str.append(" Qt"); // 向字符串后添加 "Qt"
    str.replace(1, 4, "i"); // 将第 1 个字符开始的后面 4 个字符替换为字符串 "i"
    str.insert(2, " my"); // 在第 2 个字符后插入 " my"
    qDebug() << QOBJECT::tr("str 为:") << str; // 结果为 Hi my Qt
    str = str + "!!!"; // 将两个字符串组合
    qDebug() << QOBJECT::tr("str 为:") << str; // 结果为 Hi my Qt!!!
}

str = " hi\r\n Qt! \n ";
qDebug() << QOBJECT::tr("str 为:") << str;
QString str1 = str.trimmed(); // 除去字符串两端的空白字符
qDebug() << QOBJECT::tr("str1 为:") << str1;
QString str2 = str.simplified(); // 除去字符串两端和中间多余的空白字符
qDebug() << QOBJECT::tr("str2 为:") << str2; // 结果为 hi Qt!

str = "hi,,my,,Qt";
// 从字符串中有","的地方将其分为多个子字符串
// QString::SkipEmptyParts 表示跳过空的条目
QStringList list = str.split(", ", QString::SkipEmptyParts);
qDebug() << QOBJECT::tr("str 拆分后为:") << list; // 结果为 hi,my,Qt
str = list.join(" "); // 将各个子字符串组合为一个字符串, 中间用 " " 隔开
qDebug() << QOBJECT::tr("list 组合后为:") << str; // 结果为 hi my Qt

qDebug() << QString().isNull(); // 结果为 true
qDebug() << QString().isEmpty(); // 结果为 true
qDebug() << QString("").isNull(); // 结果为 false
qDebug() << QString("").isEmpty(); // 结果为 true
return a.exec();
}

```

QString 中提供了多个方便的函数来操作字符串, 例如 append() 和 prepend() 分别实现了在字符串后面和前面添加字符串或者字符; replace() 替换指定位置的多个字符; insert() 在指定位置添加字符串或者字符; remove() 在指定位置移除多个字符; trimmed() 除去字符串两端的空白字符, 这包括 '\t'、'\n'、'\v'、'\f'、'\r' 和 ' ' ; sim-

plified()不仅除去字符串两端的空白字符,还将字符串中间的空白字符序列替换为一个空格;split()可以将一个字符串分割为多个子字符串的列表等。对于一个字符串,也可以使用“[]”操作符来获取或者修改其中的一个字符,还可以使用“+”操作符来组合两个字符串。在QString类中一个null字符串和一个空字符串并不是完全一样的。一个null字符串是使用QString的默认构造函数或者在构造函数中传递了0来初始化的字符串;而一个空字符串是指大小为0的字符串。一般null字符串都是空字符串,但一个空字符串不一定是一个null字符串,在实际编程中一般使用isEmpty()来判断一个字符串是否为空。

3. 查询操作

继续在上面的程序中进行更改。在“return a.exec();”一行代码前添加如下代码:

```
qDebug() << endl << QObject::tr("以下是在字符串中进行查询的操作:") << endl;
str = "yafeilinux";
qDebug() << QObject::tr("字符串为:") << str;
//执行下面一行代码后,结果为 linux
qDebug() << QObject::tr("包含右侧 5 个字符的子字符串:") << str.right(5);
//执行下面一行代码后,结果为 yafei
qDebug() << QObject::tr("包含左侧 5 个字符的子字符串:") << str.left(5);
//执行下面一行代码后,结果为 fei
qDebug() << QObject::tr("包含第 2 个字符以后 3 个字符的子字符串:") << str.mid(2,3);
qDebug() << QObject::tr("fei的位置:") << str.indexOf("fei"); //结果为 2
qDebug() << QObject::tr("str 的第 0 个字符:") << str.at(0); //结果为 y
qDebug() << QObject::tr("str 中 i 字符的个数:") << str.count('i'); //结果为 2
//执行下面一行代码后,结果为 true
qDebug() << QObject::tr("str 是否以"ya"开始?") << str.startsWith("ya");
//执行下面一行代码后,结果为 true
qDebug() << QObject::tr("str 是否以"linux"结尾?") << str.endsWith("linux");
//执行下面一行代码后,结果为 true
qDebug() << QObject::tr("str 是否包含"lin"字符串?") << str.contains("lin");
QString temp = "hello";
if(temp > str) qDebug() << temp; //两字符串进行比较,结果为 yafeilinux
else qDebug() << str;
```

在QString中还提供了right()、left()和mid()函数分别来提取一个字符串的最右面,最左面和中间的含有多个字符的子字符串;也可以使用indexOf()函数来获取一个字符或者子字符串在该字符串中的位置;使用at()函数可以获取一个指定位置的字符,它比“[]”操作符要快很多,因为它不会引起深拷贝;可以使用contains()函数来判断该字符串是否包含一个指定的字符或者字符串;可以使用count()来获得字符串中一个字符或者子字符串出现的次数;而使用startsWith()和endsWith()函数可以用来判断该字符串是否是以一个字符或者字符串开始或者结束的;对于两个字符串的比较,可以使用“>”和“<=”等操作符,也可以使用compare()函数。

4. 转换操作

在前面的程序中继续添加如下代码：

```

qDebug() << endl << QObject::tr("以下是字符串的转换操作:") << endl;
str = "100";
qDebug() << QObject::tr("字符串转换为整数:") << str.toInt(); //结果为 100
int num = 45;
qDebug() << QObject::tr("整数转换为字符串:") << QString::number(num); //结果为"45"
str = "FF";
bool ok;
int hex = str.toInt(&ok,16);
//结果为 ok: true 255
qDebug() << "ok: " << ok << QObject::tr("转换为十六进制:") << hex;
num = 26;
qDebug() << QObject::tr("使用十六进制将整数转换为字符串:")
    << QString::number(num,16); //结果为 1a
str = "123.456";
qDebug() << QObject::tr("字符串转换为浮点型:") << str.toFloat(); //结果为 123.456
str = "abc";
qDebug() << QObject::tr("转换为大写:") << str.toUpper(); //结果为 ABC
str = "ABC";
qDebug() << QObject::tr("转换为小写:") << str.toLowerCase(); //结果为 abc
int age = 25;
QString name = "yafei";
//name 代替 %1, age 代替 %2
str = QString("name is %1, age is %2").arg(name).arg(age);
//结果为 name is yafei, age is 25
qDebug() << QObject::tr("更改后的 str 为:") << str;
str = "%1 %2";
qDebug() << str.arg("%1f", "hello"); //结果为 %1f hello
qDebug() << str.arg("%1f").arg("hello"); //结果为 hellof %1
str = QString("ni %1").arg("hi", 5, '*');
qDebug() << QObject::tr("设置字段宽度为 5, 使用*填充:") << str; //结果为 ni * * * hi
qreal value = 123.456;
str = QString("number: %1").arg(value, 0, f, 2);
qDebug() << QObject::tr("设置小数点位数为两位:") << str; //结果为 "number:123.45"
//执行下面一行代码, 结果为 number:123.45 不会显示引号
qDebug() << QObject::tr("将 str 转换为 const char * :") << qPrintable(str);

```

QString 中的 `toInt()`、`toDouble()` 等函数可以很方便地将字符串转换为整型或者 `double` 型数据, 当转换成功后, 它们的第一个 `bool` 型参数会为 `true`; 使用静态函数 `number()` 可以将数值转换为字符串, 这里还可以指定要转换为哪种进制; 使用 `toLower()` 和 `toUpper()` 函数可以分别返回字符串小写和大写形式的副本; 而 `arg()` 函数中的

参数可以取代字符串中相应的“%1”等标记，在字符串中可以使用的标记在1~99之间，arg()函数会从最小的数字开始对应，比如QString("%5,%2,%7").arg("a").arg("b")，那么“a”会代替“%2”，“b”会代替“%5”，而“%7”会直接显示。arg()的一种重载形式是arg(const QString & a1, const QString & a2)，与使用str.arg(a1).arg(a2)是相同的，不过当参数a1中含有“%1”等标记时，两者的效果是不同的，这个可以在上面的程序中看到。该函数的另一种重载形式为arg(const QString & a, int fieldWidth=0, const QChar & fillChar=QLatin1Char(' '))，这里可以设定字段宽度，如果第一个参数a的宽度小于fieldWidth的值，那么就可以使用第三个参数设置的字符来进行填充。这里的fieldWidth如果为正值，那么文本是右对齐的，比如上面程序中的结果为“ni * * * hi”。而如果为负值，那么文本是左对齐的，例如将上面的程序中的fieldWidth改为-5，那么结果就应该是“nih * * *”。arg()还有一种重载形式arg(double a, int fieldWidth=0, char format='g', int precision=-1, const QChar & fillChar=QLatin1Char(' '))，它的第一个参数是double类型的，后面的format和precision分别可以指定其类型和精度。可用的format如表7-7所列。

表7-7 QString类函数参数格式

格 式	含 义
e	格式如：[-]9.9e[+ -]999
E	格式如：[-]9.9E[+ -]999
f	格式如：[-]9.9
g	使用e或者f格式，选择其中最精简的
G	使用E或者f格式，选择其中最精简的

对于‘e’、‘E’和‘f’格式，精度precision表示小数点后面的位数；而对于‘g’和‘G’格式，精度表示有效数字的最大位数。arg()是一个非常有用的函数，前面的章节中已经多次用到了它，要在一个字符串中使用变量，那么使用arg()是一种很好的解决办法。还有一个qPrintable()函数，它不是QString中的函数，但是它可以将字符串转换为const char*类型，输出一个字符串的时候，两边总会有引号，为了显示更清晰，可以使用这个函数将引号去掉。在QString中还提供了toAscii()、toLatin1()、toUtf8()和toLocal8Bit()等函数，可以使用一种编码将字符串转换为QByteArray类型。

7.2.5 QByteArray 和 QVariant

QByteArray类提供了一个字节数组，可以用来存储原始字节（包括‘\0’）和传统的以‘\0’结尾的8位字符串。使用QByteArray比使用const char*要方便很多，在后台，它总是保证数据以一个‘\0’结尾，而且使用隐式共享来减少内存的使用和避免不必要的数据拷贝。但是除了当需要存储原始二进制数据或者对内存保护要求很高（如在嵌入式Linux上）时，一般都推荐使用QString，因为QString是存储16位的Unicode

字符,使得在应用程序中更容易存储非 ASCII 和非 Latin - 1 字符,而且 QString 全部使用的是 Qt 的 API。

QByteArray 类拥有和 QString 类相似的接口函数,比如上一节讲到的 QString 的那些函数,除了 arg()以外,在 QByteArray 中都有相同的用法。

QVariant 类像是最常见的 Qt 的数据类型的一个共用体(union),一个 QVariant 对象在一个时间只保存一个单一类型的单一值(有些类型可能是多值的,比如字符串列表)。可以使用 toT()(T 代表一种数据类型)函数来将 QVariant 对象转换为 T 类型,并且获取它的值。这里 toT() 函数会复制以前的 QVariant 对象,然后对其进行转换,所以以前的 QVariant 对象并不会改变。QVariant 是 Qt 中一个很重要的类,比如前面讲解属性系统时提到的 QObject::property() 返回的就是 QVariant 类型的对象。

(项目源码路径: src\07\7-13\myVariant)新建 Qt Gui 应用,项目名称为 myVariant,基类选择 QWidget,类名保持 Widget 不变。建好项目后,我们在 widget.cpp 文件中添加头文件 #include <QDebug>,然后在构造函数中添加如下代码:

```

QVariant v1(15);
qDebug() << v1.toInt();                                //结果为 15

QVariant v2(12.3);
qDebug() << v2.toFloat();                               //结果为 12.3

QVariant v3("nihao");
qDebug() << v3.toString();                             //结果为"nihao"

QColor color = QColor(Qt::red);

QVariant v4 = color;
qDebug() << v4.type();                                 //结果为 QVariant::QColor
qDebug() << v4.value<QColor>();                      //结果为 QColor(ARGB 1,1,0,0)

QString str = "hello";
QVariant v5 = str;
qDebug() << v5.canConvert(QVariant::Int);           //结果为 true
qDebug() << v5.toString();                            //结果为"hello"
qDebug() << v5.convert(QVariant::Int);               //结果为 false
qDebug() << v5.toString();                            //转换失败,v5 被清空,结果为"0"

```

QVariant 类的 toInt() 函数返回 int 类型的值,toFloat() 函数返回 float 类型的值。但是,因为 QVariant 是 QtCore 库的一部分,所以没有提供对 QtGui 中定义的数据类型(例如 QColor、QImage 等)进行转换的函数,也就是说,这里没有 toColor() 这样的函数。不过,可以使用 QVariant::value() 函数或者 QVariantValue() 模板函数来完成这样的转换,例如上面程序中对 QColor 类型的转换。要了解 QVariant 可以包含的所有类型以及这些类型在 QVariant 类中对应的 toT() 函数,可以查看 QVariant 类的参考文档。对于一个类型是否可以转换为一个特殊的类型,可以使用 canConvert() 函数来判断;如果可以转换,则该函数返回 true。也可以使用 convert() 函数来将一个类型转换为其他不同的类型,如果转换成功则返回 true;如果无法进行转换,variant 对象将会

被清空，并且返回 false。需要说明，对于同一种转换，canConvert()和 convert()函数并不一定返回同样的结果，这通常是因为 canConvert()只报告 QVariant 进行两个类型之间转换的能力。也就是说，如果提供了合适的数据时，这两个类型间可以进行转换，但是，如果提供的数据不合适，那么转换就会失败，这样 convert()的返回值就与 canConvert()不同了。例如上面程序中的 QString 类型的字符串 str，当 str 中只有数字字符时，它可以转换为 int 类型，比如 str=“123”，因为它有这个能力，所以 canConvert()返回为 true。但是，现在 str 中包含了非数字字符，真正进行转换时会失败，所以 convert()返回为 false。使用 canConvert()函数返回为 true 的数据类型组合如表 7-8 所列。

表 7-8 可以自动进行的类型转换

类型	自动转换到
Bool	Char、Double、Int、LongLong、String、UInt、ULongLong
ByteArray	Double、Int、LongLong、String、UInt、ULongLong
Char	Bool、Int、UInt、LongLong、ULongLong
Color	String
Date	DateTime、String
DateTime	Date、String、Time
Double	Bool、Int、LongLong、String、UInt、ULongLong
Font	String
Int	Bool、Char、Double、LongLong、String、UInt、ULongLong
KeySequence	Int、String
List	QStringList(如果列表中的项目可以转换为字符串)
LongLong	Bool、ByteArray、Char、Double、Int、String、UInt、ULongLong
Point	PointF
Rect	RectF
String	Bool、ByteArray、Char、Color、Date、DateTime、Double、Font、Int、KeySequence、LongLong、QStringList、Time、UInt、ULongLong
QStringList	List、String(如果列表中只包含一个项目)
Time	String
UInt	Bool、Char、Double、Int、LongLong、String、ULongLong
ULongLong	Bool、Char、Double、Int、LongLong、String、UInt

7.3 正则表达式

在前面的章节中已经接触过了正则表达式，如 3.3.3 小节中的行编辑器进行输入验证和 5.2.5 小节中实现编辑器的语法高亮。正则表达式 (regular expression) 就是在

一个文本中匹配子字符串的一种模式(pattern)，可以简写为“regexp”。一个 regexp 主要应用在以下几个方面：

- 验证。一个 regexp 可以测试一个子字符串是否符合一些标准。例如，是一个整数或者不包含任何空格等。
- 搜索。一个 regexp 提供了比简单的子字符串匹配更强大的模式匹配。例如，匹配单词 mail 或者 letter，而不匹配单词 email 或者 letterbox。
- 查找和替换。一个 regexp 可以使用一个不同的字符串替换一个字符串中所有要替换的子字符串。例如，使用 Mail 来替换一个字符串中所有的 M 字符，但是如果 M 字符后面有 ail 时不进行替换。
- 字符串分割。一个 regexp 可以识别在哪里进行字符串分割。例如，分割制表符隔离的字符串。

Qt 中的 QRegExp 类实现了使用正则表达式进行模式匹配。QRegExp 是以 Perl 的正则表达式语言为蓝本的，它完全支持 Unicode。QRegExp 中的语法规则可以使用 setPatternSyntax() 函数来更改。

7.3.1 正则表达式简介

Regexps 由表达式(expressions)、量词(quantifiers)和断言(assertions)组成。最简单的一个表达式就是一个字符，例如 x 和 5。而一组字符可以使用方括号括起来，例如 [ABC] 将会匹配一个 A 或者一个 B 或者一个 C，这个也可以简写为 [A-C]，这样我们要匹配所有的英文大写字母，就可以使用 [A-Z]。

一个量词指定了必须要匹配的表达式出现的次数。例如， $x\{1,1\}$ 意味着必须匹配且只能匹配一个字符 x，而 $x\{1,5\}$ 意味着匹配一列字符 x，其中至少要包含一个字符 x，但是最多包含 5 个字符 x。

现在假设要使用一个 regexp 来匹配 0~99 之间的整数。因为至少要有一个数字，所以使用表达式 [0-9]{1,1} 开始，它匹配一个单一的数字一次。要匹配 0~99，可以想到将表达式最多出现的次数设置为 2，即 [0-9]{1,2}。现在这个 regexp 已经可以满足假设的需要了，不过，它也会匹配出现在字符串中间的整数。如果想匹配的整数是整个字符串，那么就需要使用断言“^”和“\$”，当 ^ 在 regexp 中作为第一个字符时，意味着这个 regexp 必须从字符串的开始进行匹配；当 \$ 在 regexp 中作为最后一个字符时，意味着 regexp 必须匹配到字符串的结尾。所以，最终的 regexp 为 ^[0-9]{1,2} \$。

一般可以使用一些特殊的符号来表示一些常见的字符组和量词。例如，[0-9] 可以使用 \d 来替代。而对于只出现一次的量词 {1,1}，可以使用表达式本身代替，例如 x{1,1} 等价于 x。所以要匹配 0~99，就可以写为 ^\d{1,2} \$ 或者 ^\d\d{0,1} \$。而 {0,1} 表示字符是可选的，就是只出现一次或者不出现，它可以使用 ? 来代替，这样 regexp 就可以写为 ^\d\d? \$，它意味着从字符串的开始，匹配一个数字，紧接着是 0 个或 1 个数字，再后面就是字符串的结尾。

现在写一个 regexp 来匹配单词“mail”或者“letter”其中的一个，但是不要匹配那些

包含这些单词的单词，比如“email”和“letterbox”。要匹配“mail”，regexp 可以写成 `m{1,1}a{1,1}i{1,1}l{1,1}`，因为 `{1,1}` 可以省略，所以又可以简写成 `mail`。下面就可以使用竖线“|”来包含另外一个单词，这里“|”表示“或”的意思。为了避免 regexp 匹配多余的单词，必须让它从单词的边界进行匹配。首先，将 regexp 用括号括起来，即 `(mail|letter)`。括号将表达式组合在一起，可以在一个更复杂的 regexp 中作为一个组件来使用，这样也可以方便检测到底是哪一个单词被匹配了。为了避免匹配的开始和结束都在单词的边界上，我们要将 regexp 包含在 `\b` 单词边界断言中，即 `\b(mail|letter)\b`。这个 `\b` 断言在 regexp 中匹配一个位置，而不是一个字符，一个单词的边界是任何的非单词字符，如一个空格、新行或者一个字符串的开始或者结束。

想使用一个单词，例如“Mail”，替换一个字符串中的字符 M，但是当字符 M 的后面是“ail”的话就不再替换。这样可以使用 `(?! E)` 断言，例如这里 regexp 应该写成 `M(?! Mail)`。

如果想统计“Eric”和“Eirik”在字符串中出现的次数，可以使用 `\b(Eric|Eirik)\b` 或者 `\bEi? ri[ck]\b`。这里需要使用单词边界断言‘\b’来避免匹配那些包含了这些名字的单词。

下面就在实际的程序中演示这些例子。先说明一下，因为在 C++ 中“\”也是转义字符，所以在 regexp 中使用它时，需要再转义一次，比如使用“\d”就应该写成“\\d”；而如果要使用“\”本身，那么就要写成“\\\\”。

（项目源码路径：src\07\7-14\myRegexp）新建 Qt Gui 应用，项目名称为 myRegexp，基类选择 QWidget，类名保持 Widget 不变。建好项目后，首在 widget.cpp 文件中添加头文件 `#include <QDebug>`。然后在构造函数中添加如下代码：

```

QRegExp rx("^\d\d? $"); //两个字符都必须为数字,第二个字符可以没有
qDebug() << rx.indexIn("a1"); //结果为 -1,不是数字开头
qDebug() << rx.indexIn("5"); //结果为 0
qDebug() << rx.indexIn("5b"); //结果为 -1,第二个字符不是数字
qDebug() << rx.indexIn("12"); //结果为 0
qDebug() << rx.indexIn("123"); //结果为 -1,超过了两个字符
qDebug() << "*****"; //输出分割符,为了显示清晰
rx.setPattern("\b(mail|letter)\b"); //匹配 mail 或者 letter 单词
qDebug() << rx.indexIn("emailletter"); //结果为 -1,mail 不是一个单词
qDebug() << rx.indexIn("my mail"); //返回 3
qDebug() << rx.indexIn("my email letter"); //返回 9
qDebug() << "*****";
rx.setPattern("M(?! ail)"); //匹配字符 M,其后面不能跟有 ail 字符
QString str1 = "this is M";
str1.replace(rx,"Mail"); //使用"Mail"替换匹配到的字符
qDebug() << "str1: " << str1; //结果为 this is Mail
QString str2 = "my M, your Ms, his Mail";
str2.replace(rx,"Mail");

```

```

qDebug() << "str2: " << str2;           //结果为 my Mail,your Mails,his Mail
qDebug() << "*****";
QString str3 = "One Eric another Eirik, and an Ericsson. "
"How many Eiriks, Eric?";           //一个字符串如果一行写不完,换行后两行都需要加双引号
QRegExp rx2("\bEi? ri[ck]\b");        //匹配 Eric 或者 Eirik
int pos = 0;
int count = 0;
while (pos >= 0) {
    pos = rx2.indexIn(str3, pos);
    if (pos >= 0) {
        ++pos;                         //从匹配的字符的下一个字符开始匹配
        ++count;                        //匹配到的数目加 1
    }
}
qDebug() << "count: " << count;        //结果为 3

```

这里使用了 QRegExp 的 indexIn() 函数, 它从指定的位置开始向后对字符串进行匹配, 默认是从字符串开始进行匹配。如果匹配成功, 返回第一个匹配到的位置的索引, 如果没有匹配到则返回 -1。setPattern() 函数用来输入一个 regexp。而 QString 的 replace() 函数可以使用给定的 regexp 和替换字符串来进行字符串的替换。

7.3.2 正则表达式组成元素

前面已经提到过一个正则表达式 regexp 由表达式、量词和断言组成, 其中的表达式可以是各种字符和字符组, 而一些常用的字符集可以使用一些缩写来表示, 如表 7-9 所列。

表 7-9 正则表达式中的字符和字符集缩写

元 素	含 义
c	一个字符代表它本身, 除非这个字符有特殊的 regexp 含义。例如, c 匹配字符 c
\c	跟在反斜杠后面的字符匹配字符本身, 但是本表中下面指定的这些字符除外。例如, 要匹配一个字符串的开头, 使用 \^
\a	匹配 ASCII 的振铃(BEL, 0x07)
\f	匹配 ASCII 的换页(FF, 0x0C)
\n	匹配 ASCII 的换行(LF, 0x0A)
\r	匹配 ASCII 的回车(CR, 0x0D)
\t	匹配 ASCII 的水平制表符(HT, 0x09)
\v	匹配 ASCII 的垂直制表符(VT, 0x0B)
\xhhhh	匹配 Unicode 字符对应的十六进制数 hhhh(在 0x0000~0xFFFF 之间)
\ooo	匹配八进制的 ASCII/Latin1 字符 ooo(在 0~0377 之间)

续表 7-9

元素	含义
.(点)	匹配任意字符(包括新行)
\d	匹配一个数字
\D	匹配一个非数字
\s	匹配一个空白字符,包括‘\t’、‘\n’、‘\v’、‘\f’、‘\r’和‘ ’
\S	匹配一个非空白字符
\w	匹配一个单词字符,包括任意一个字母或数字或下划线,即 A~Z, a~z, 0~9, _ 中任意一个
\W	匹配一个非单词字符
\n	第 n 个反向引用。例如,\1,\2 等

对于字符集还有两个特殊的符号“~”和“-”，其中“~”在方括号的开始可以表示相反的意思，例如[~abc]表示匹配任何字符，但是不匹配‘a’或‘b’或‘c’。而“-”可以表示一个范围的字符，例如[W-Z]表示匹配‘W’或者‘X’或者‘Y’或者‘Z’。

默认的，一个表达式将自动量化为{1,1}，就是说它应该出现一次。表 7-10 列出了量词的使用情况，其中 E 代表一个表达式，一个表达式可以是一个字符，或者一个字符集的缩写，或者在方括号中的一个字符集，或者在括号中的一个表达式。

表 7-10 正则表达式中的量词

量词	含义
E?	匹配 0 次或者 1 次, 表明 E 是可选的, E? 等价于 E{0,1}
E+	匹配 1 次或者多次, E+ 等价于 E{1,}, 例如, 0+ 匹配“0”、“00”、“000”等
E*	匹配 0 次或者多次, 等价于 E{0,}
E{n}	匹配 n 次, 等价于 E{n,n}, 例如, x{5} 等价于 x{5,5}, 也等价于 xxxxx
E{n,}	匹配至少 n 次
E{,m}	匹配至多 m 次, 等价于 E{0,m}
E{n,m}	匹配至少 n 次, 至多 m 次

在使用量词时要注意, tag+ 表示匹配一个‘t’跟着一个‘a’然后跟着至少一个‘g’。而(tag)+ 表示匹配“tag”至少一次。还要说明的是, 量词一般是贪婪的(greedy), 它会尽可能多地去匹配可以匹配的文本, 例如, 0+ 匹配它发现的第一个 0 以及其随后所有的连续的 0, 当应用到字符串“20005”时, 它会匹配其中的 3 个 0。要使量词变得非贪婪(non-greedy), 可以使用 setMinimal(true), 这样在上面的例子中就会只匹配第一个 0。

断言在 regexp 中作出一些有关文本的声明, 它们不匹配任何字符。正则表达式中的断言如表 7-11 所列, 其中 E 代表一个表达式。

表 7-11 正则表达式中的断言

断言	含义
'	标志着字符串的开始。如果要匹配'就要使用\\'
\$	标志着字符串的结尾。如果要匹配\$就要使用\\\$
\b	一个单词的边界
\B	一个非单词的边界,当\b为false时它为true
(? =E)	表达式后面紧跟着E才匹配。例如,const(? =\s+char)匹配“const”且其后必须有“char”
(?! E)	表达式后面没有紧跟着E才匹配。例如,const(?! \s+char)匹配“const”但其后不能有“char”

QRegExp 类还支持通配符(Wildcard)匹配。很多命令 shell(例如 bash 和 cmd.exe)都支持文件通配符(file globbing),可以使用通配符来识别一组文件。QRegExp 的 setPatternSyntax()函数就是用来在 regexp 和通配符之间进行切换的。通配符匹配要比 regexp 简单很多,它只有 4 个特点,如表 7-12 所列。

表 7-12 通配符

字符	含义
c	任意一个字符,表示字符本身
?	匹配任意一个字符,类似于 regexp 中的“.”
*	匹配 0 个或者多个任意的字符,类似于 regexp 中的“.*”
[...]	在方括号中的字符集,与 regexp 中的类似

例如要匹配所有的 txt 类型的文件,那么可以在前面的程序中添加如下代码来实现:

```
QRegExp rx3("*.txt");
rx3.setPatternSyntax(QRegExp::Wildcard);
qDebug() << rx3.exactMatch("README.txt");           //结果为 true
qDebug() << rx3.exactMatch("welcome.txt.bak");    //结果为 false
```

除了通配符以外,QRegExp 还支持其他一些语法,如表 7-13 所列。这些语法都可以使用 setPatternSyntax()函数来切换。

表 7-13 正则表达式语法

常量	描述
QRegExp::RegExp	类似于 Perl 的模式匹配语法,这个是默认语法
QRegExp::RegExp2	类似于 RegExp,不过是一种贪婪匹配语法
QRegExp::Wildcard	一种简单的模式匹配语法
QRegExp::WildcardUnix	与 Widcard 类似,但是使用 Unix shell 的行为
QRegExp::FixedString	使用字符原意,不使用任何的转义字符
QRegExp::W3CXmlSchema11	在 W3C XML Schema 1.1 规范中定义的一种正则表达式

7.3.3 文本捕获

在 regexp 中使用括号可以使一些元素组合在一起,这样既可以对它们进行量化,也可以捕获它们。例如,使用表达式 mail|letter 来匹配一个字符串,则知道了有一个单词被匹配了,但是却不能知道具体是哪一个,使用括号就可以捕获被匹配的那个单词,比如使用 (mail|letter) 来匹配字符串“I Sent you some email”,这样就可以使用 cap() 或者 capturedTexts() 函数来提取匹配的字符。

还可以在 regexp 中使用捕获到的文本,为了表示捕获到的文本,我们使用反向引用 \n,其中 n 从 1 开始编号,比如 \1 就表示前面第一个捕获到的文本。例如,使用 \b (\w+) \W+ \1 \b 在一个字符串中查询重复出现的单词,这意味着先匹配一个单词边界,随后是一个或者多个单词字符,随后是一个或者多个非单词字符,随后是与前面第一个括号中相同的文本,随后是单词边界。

如果使用括号仅仅是为了组合元素而不是为了捕获文本,那么可以使用非捕获语法,例如 (? : green | blue)。非捕获括号由“(?:”开始,由“)”结束。使用非捕获括号比使用捕获括号更高效,因为 regexp 引擎只需做较少的工作。

下面来看一个文本捕获的例子,在前面的程序中继续添加如下代码:

```
QRegExp rx4("(\\d+)");
QString str4 = "Offsets: 12 14 99 231 7";
QStringList list;
int pos2 = 0;
while ((pos2 = rx4.indexIn(str4, pos2)) != -1) {
    list << rx4.cap(1); //第一个捕获到的文本
    pos2 += rx4.matchedLength(); //上一个匹配的字符串的长度
}
qDebug() << list; //结果 12,14,99,231,7

QRegExp rxlen("(\\d+)(?:\\s*)(cm|inch)");
int pos3 = rxlen.indexIn("Length: 189cm");
if (pos3 > -1) {
    QString value = rxlen.cap(1); //结果为 189
    QString unit = rxlen.cap(2); //结果为 cm
    QString string = rxlen.cap(0); //结果为 189cm
    qDebug() << value << unit << string;
}

QRegExp rx5("\\b(\\w+)\\W+\\1\\b");
rx5.setCaseSensitivity(Qt::CaseInsensitive); //设置不区分大小写
qDebug() << rx5.indexIn("Hello--hello"); //结果为 0
qDebug() << rx5.cap(1); //结果为 Hello
```

```

QRegExp rx6("\b 你好\b");           //匹配中文
qDebug() << rx6.indexIn("你好");      //结果为 0
qDebug() << rx6.cap(0); //整个字符串完全匹配,使用 cap(0)捕获,结果为“你好”

```

在使用 `cap()` 函数时,其参数 0 表示完全匹配,会返回整个的匹配结果。如果想要返回在括号中的子表达式的匹配结果,那么就要从 1 开始编号,为 1 时表示最左边的括号中的第一个匹配到的结果,具体的例子可以参见上面的代码。当要在表达式中使用中文时,需要在 `main.cpp` 文件中添加头文件 `#include <QTextCodec>`,然后再 `main()` 函数中添加如下一行代码:

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
```

3.3.3 小节讲解行编辑器的输入验证时使用了验证器 `QValidator` 类,当时也提到了 `QRegExpValidator` 类,它是基于正则表达式的验证器。`QRegExpValidator` 使用一个 `regexp` 来决定一个输入的字符串是否是可以接受的(Acceptable)或者无效的(Invalid)或者在两者之间的(Intermediate)。

正则表达式对于初学者来说是很复杂的一部分内容,不过,也是很重要的内容。其实把一些基本的语法规则掌握了以后,照着去写一个 `regexp` 也不是很困难的事情。在《Qt 及 Qt Quick 开发实战精解》中的音乐播放器实例的 3.4.2 小节中进行歌词文件的解析,就是使用的正则表达式,可以参考一下。关于正则表达式的内容,可以查看 `QRegExp` 的参考文档。Qt 的 Regular Expressions 示例程序是关于正则表达式的,在 Tools 分类下,也可以参考一下。

7.4 小结

这一章介绍了 Qt 的一些核心内容,比如信号和槽、元对象系统等;也学习了容器类及其相关的 `QString`、`QByteArray` 和 `QVariant` 等类;还学习了正则表达式的相关知识。之所以将这些知识放到同一章中进行讲解,是因为它们都是理论性比较强,而且很枯燥。不过,这些知识都是非常重要的,只有掌握了 Qt 的核心内容才能在 Qt 编程过程中游刃有余;而容器类和正则表达式是实现一些强大的功能时经常用到的,而且它们不是 Qt 自身的,所以学习了这些知识,以后在别处照样可以使用。

第8章

界面外观

一个完善的应用程序不仅应该有实用的功能,还要有一个漂亮的外观,这样才能使应用程序更加友好,更加吸引用户。作为一个跨平台的 UI 开发框架,Qt 提供了强大而灵活的界面外观设计机制。这一章将学习在 Qt 中设计应用程序外观的相关知识,会对 Qt 风格 QStyle 和调色板 QPalette 进行简单介绍,然后再对 Qt 样式表(Qt Style Sheets)进行重点讲解,最后还会涉及了不规则窗体和透明窗体的实现方法。

8.1 Qt 风格

Qt 中的各种风格是一组继承自 QStyle 的类。QStyle 类是一个抽象基类,封装了一个 GUI 的外观,Qt 的内建(built-in)部件使用它来执行几乎所有的绘制工作,以确保它们看起来可以像各个平台上的本地部件一样。一些风格已经内置在了 Qt 中,例如 Windows 风格和 Motif 风格;而有些风格只在特定的平台上才有效,例如 Windows XP 风格、Windows Vista 风格和 Mac OS X 风格。Qt 提供的风格类如表 8-1 所列。

表 8-1 Qt 提供的风格类

类名	介绍
QCDEStyle	CDE(Common Desktop Environment)风格
QCleanlooksStyle	类似于 GNOME 中的 Clearlook 风格
QGtkStyle	GTK+风格
QMotifStyle	Motif 风格
QMacStyle	Mac OS X 风格
QPlastiqueStyle	类似于 KDE 中的 Plastik 风格
QWindowsStyle	微软 Windows 风格
QWindowsVistaStyle	微软 Windows Vista 风格
QWindowsXPStyle	微软 Windows XP 风格

在使用 Qt Creator 设计模式设计界面时,可以使用 Qt 提供的各种风格进行预览,当然也可以使用特定的风格来运行程序。下面来看具体的例子。

8.1.1 使用不同风格预览程序

(项目源码路径: src\08\8-1\myStyle)新建 Qt Gui 应用,项目名称为 myStyle,类名为 MainWindow,基类保持 QMainWindow 不变。建立完项目后,单击 mainwindow.ui 文件进入设计模式,向界面上拖入一个 Label、Push Button、Spin Box、Line Edit 和 Progress Bar。

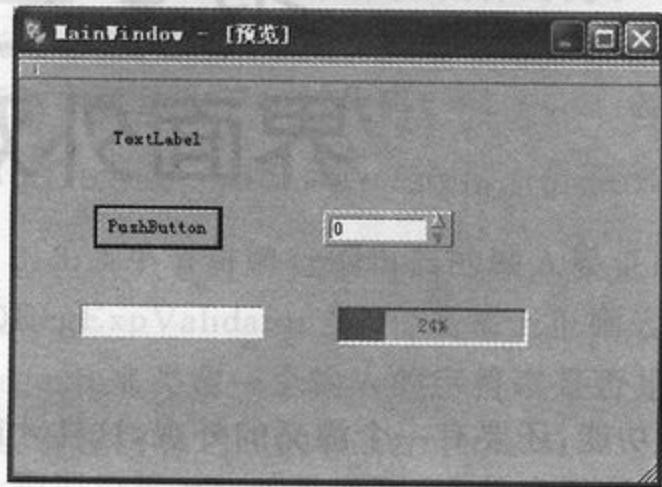


图 8-1 CDE 风格预览效果

然后选择“工具→界面编辑器→预览于”菜单项,这里列出了现在可用的几种风格,这里选择“CDE 风格”,预览效果如图 8-1 所示。也可以使用其他几种风格进行预览。

8.1.2 使用不同风格运行程序

如果想使用不同的风格来运行程序,那么只需要调用 QApplication 的 setStyle() 函数指定要使用的风格即可。现在打开 main.cpp 文件,然后添加 #include <QMotifStyle> 头文件包含,并在 main() 函数的“QApplication a(argc, argv);”一行代码后添加如下一行代码:

```
a.setStyle(new QMotifStyle);
```

这时运行程序,便会使用 Motif 风格。如果不想在程序中指定风格,而是想在运行程序时再指定,那么就可以在使用命令行运行程序时通过添加参数来指定,比如要使用 Motif 风格,则可以使用“-style motif”参数。而如果不想整个应用程序都使用相同的风格,那么可以调用部件的 setStyle() 函数来指定该部件的风格。进入 mainwindow.cpp 文件,先添加头文件 #include <QWindowsXPStyle>,然后在构造函数中添加如下一行代码:

```
ui->progressBar->setStyle(new QWindowsXPStyle);
```

这时再次运行程序,其中的进度条部件就会使用 Windows XP 的风格了,效果如图 8-2 所示。另外,还可以使用 QStyleFactory::keys() 函数来获取当前系统所支持的风格。

除了 Qt 中提供的这些风格外,也可以自定义风格,一般的做法是子类化 Qt 的风格类,或者子类化 QStyle 类。关于这个内容这里不再讲述,有兴趣的读者可以查看 Styles 示例程序,它在 Widgets 分类下。关于 Qt 风格更多的内容,可以查看 Styles and Style Aware Widgets 关键字。

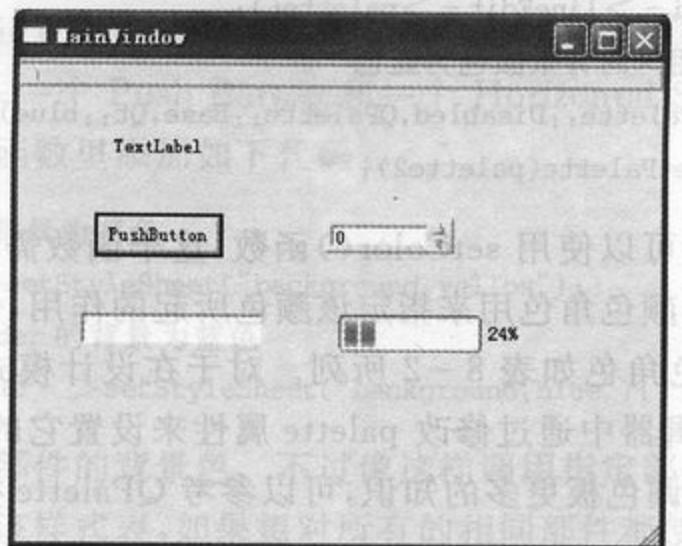


图 8-2 更改部件的风格运行效果

8.1.3 调色板

调色板 QPalette 类包含了部件各种状态的颜色组。一个调色板包含 3 种状态：激活(Active)、失效(Disabled)和非激活(Inactive)。Qt 中的所有部件都包含一个调色板，并且使用各自的调色板来绘制它们自身，这样可以使用户界面更容易配置，也更容易保持一致。调色板中的颜色组包括：

- 激活颜色组 QPalette::Active，用于获得键盘焦点的窗口；
- 非激活颜色组 QPalette::Inactive，用于其他的窗口；
- 失效颜色组 QPalette::Disabled，用于由于一些原因而不可用的部件(不是窗口)。

要改变一个应用程序的调色板，可以先使用 QApplication::palette() 函数来获取其调色板，然后对其进行更改，最后再使用 QApplication::setPalette() 函数来使用该调色板。更改了应用程序的调色板，会影响到该程序的所有窗口部件。如果要改变一个部件的调色板，可以调用该部件的 palette() 和 setPalette() 函数，这样只会影响该部件及其子部件。下面来看一个例子。

仍然在前面的程序中进行更改。在 mainwindow.cpp 文件中添加头文件 #include <QPalette>，然后在构造函数中继续添加如下代码：

```
// 获取 pushButton 的调色板
QPalette palette1 = ui->pushButton->palette();
// 设置按钮文本颜色为红色
palette1.setColor(QPalette::ButtonText, Qt::red);
// 设置按钮背景色为绿色
palette1.setColor(QPalette::Button, Qt::green);
// pushButton 使用修改后的调色板
ui->pushButton->setPalette(palette1);
// 设置 lineEdit 不可用
ui->lineEdit->setDisabled(true);
```

```

QPalette palette2 = ui->lineEdit->palette();
//设置行编辑器不可用时的背景颜色为蓝色
palette2.setColor(QPalette::Disabled,QPalette::Base,Qt::blue);
ui->lineEdit->setPalette(palette2);

```

设置调色板颜色时可以使用 `setColor()` 函数,这个函数需要指定颜色角色(Color Role)。在 `QPalette` 中,颜色角色用来指定该颜色所起的作用,例如是背景颜色或者是文本颜色等,主要的颜色角色如表 8-2 所列。对于在设计模式中添加到界面上的部件,也可以在其属性编辑器中通过修改 `palette` 属性来设置它的调色板,这样还可以预览修改后的效果。对于调色板更多的知识,可以参考 `QPalette` 类的帮助文档。

表 8-2 `QPalette` 类主要的颜色角色

常量	描述
<code>QPalette::Window</code>	一个一般的背景颜色
<code>QPalette::WindowText</code>	一个一般的前景颜色
<code>QPalette::Base</code>	主要作为输入部件(如 <code>QLineEdit</code>)的背景色,也可用作 <code>QComboBox</code> 的下拉列表的背景色或者 <code>QToolBar</code> 的手柄颜色
<code>QPalette::AlternateBase</code>	在交替行颜色的视图中作为交替背景色
<code>QPalette::ToolTipBase</code>	作为 <code>QToolTip</code> 和 <code>QWhatsThis</code> 的背景色
<code>QPalette::ToolTipText</code>	作为 <code>QToolTip</code> 和 <code>QWhatsThis</code> 的前景色
<code>QPalette::Text</code>	和 <code>Base</code> 一起使用,作为前景色
<code>QPalette::Button</code>	按钮部件背景色
<code>QPalette::ButtonText</code>	按钮部件前景色

8.2 Qt 样式表

Qt 样式表是一个可以自定义部件外观的十分强大的机制。Qt 样式表的概念、术语和语法都受到了 HTML 的层叠样式表(Cascading Style Sheets,CSS)的启发,不过与 CSS 不同的是,Qt 样式表应用于部件的世界。

8.2.1 概述

样式表使用文本描述,可以使用 `QApplication::setStyleSheet()` 函数将其设置到整个应用程序上,也可以使用 `QWidget::setStyleSheet()` 函数将其设置到一个指定的部件(还有它的子部件)上。如果在不同的级别都设置了样式表,那么 Qt 会使用所有有效的样式表,这被称为样式表的层叠。下面来看一个简单的例子。

1. 使用代码设置样式表

(项目源码路径: `src\08\8-2\myStyleSheets`)新建 Qt Gui 应用,项目名称为 `my`

StyleSheets，类名为 MainWindow，基类保持 QMainWindow 不变。建立好项目后进入设计模式，向界面上拖入一个 Push Button 和一个 Horizontal Slider，然后在 mainwindow.cpp 文件中的构造函数里添加如下代码：

```
//设置 pushButton 的背景为黄色
ui->pushButton->setStyleSheet("background:yellow");
//设置 horizontalSlider 的背景为蓝色
ui->horizontalSlider->setStyleSheet("background:blue");
```

这样便设置了两个部件的背景色。不过像这样调用指定部件的 setStyleSheet() 函数只会对这个部件应用该样式表，如果想对所有的相同部件都使用相同的样式表，那么可以在它们的父部件上设置样式表。因为这里两个部件都在 MainWindow 上，所以可以为 MainWindow 设置样式表。先注释掉上面的两行代码，然后添加如下代码：

```
setStyleSheet("QPushbutton{background:yellow}QSlider{background:blue}");
```

这样，以后再往主窗口上添加的所有 QPushbutton 部件和 QSlider 部件的背景色都会改为这里指定的颜色。除了使用代码来设置样式表外，也可以在设计模式中为添加到界面上的部件设置样式表，这样更加直观。

2. 在设计模式中设置样式表

先注释掉上面添加的代码，然后进入设计模式。在界面上右击，在弹出的菜单中选择“改变样式表”，这时会出现编辑样式表对话框，在其中输入如下代码：

```
QPushbutton{
```

```
}
```

注意光标留在第一个大括号后。然后单击上面“添加颜色”选项后面的下拉箭头，在弹出的列表中选择“background-color”一项，如图 8-3 所示。这时会弹出选择颜色对话框，随便选择一个颜色，然后单击“确定按钮”，这时会自动添加代码：

```
QPushbutton{
    background-color: rgb(0, 85, 255);
}
```

根据选择的颜色不同，rgb() 中参数的数值也会不同。可以看到，在这里设置样式表不仅很便捷而且很直观，不仅可以设置颜色，还可以使用图片，使用渐变颜色或者更改字体。相似的，可以再设置 QSlider 的背景色，添加完成后，如图 8-4 所示。在设计模式，有时无法正常显示设置好的样式表效果，不过运行程序后会正常显示的。这里是在 MainWindow 界面上设置了样式表，当然，也可以按照这种方法在指定的部件上添加样式表。

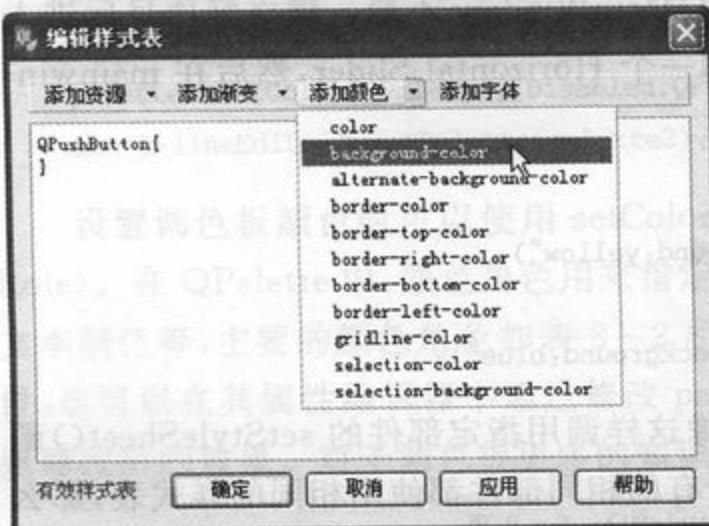


图 8-3 在设计模式编辑样式表

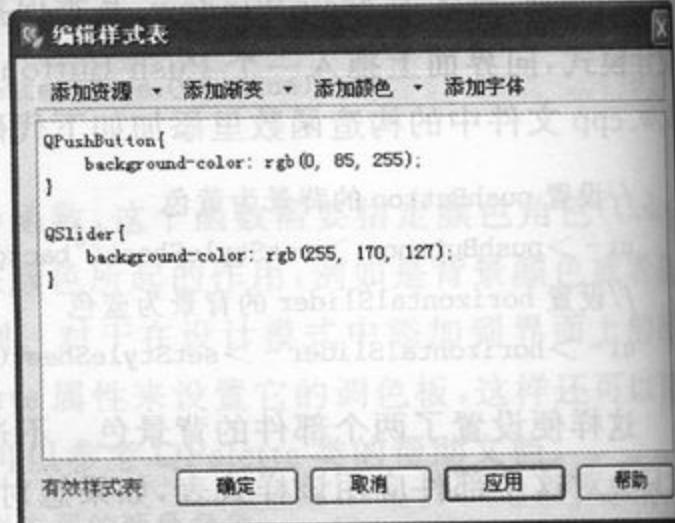


图 8-4 添加样式表代码

对于自定义样式，样式表要比调色板强大很多。例如，可以通过设置 QPalette::Button 角色为红色来获得一个红色的按钮，但是，这并不能保证在所有风格中都可以正常工作，因为它会受到不同平台的准则和本地主题引擎所限制。不过，样式表就不受这些限制，样式表可以执行所有那些单独使用调色板很困难或者无法执行的自定义操作。样式表应用在当前的部件风格之上，这意味着应用程序的外观会尽可能本地化。此外，样式表可以用来给应用程序提供一个独特的外观，而不用去子类化 QStyle，这样就可以很容易地实现现在大多数应用程序中所拥有的换肤功能。

8.2.2 Qt 样式表语法

Qt 样式表的术语和语法规则与 HTML CSS 基本相同，下面从几个方面来进行讲解。

1. 样式规则

样式表包含了一系列的样式规则，一个样式规则由一个选择符(selector)和声明(declaration)组成。选择符指定了受该规则影响的部件，声明指定了这个部件上要设置的属性。例如：

```
QPushButton{color:red}
```

在这个样式规则中，QPushButton 是选择符，{color:red} 是声明，而 color 是属性，red 是值。这个规则指定了 QPushButton 和它的子类应该使用红色作为它们的前景色。Qt 样式表中一般不区分大小写，例如 color、Color、COLOR 和 COloR 表示相同的属性。只有类名，对象名和 Qt 属性名是区分大小写的。一些选择符可以指定相同的声明，只需要使用逗号隔开，例如：

```
QPushButton, QLineEdit, QComboBox{color:red}
```

一个样式规则的声明部分是一些属性：值对组成的列表，它们包含在大括号中，使用分号隔开。例如：

```
QPushButton{color:red;background-color:white}
```

可以在 Qt Style Sheets Reference 关键字对应的文档中的 List of Properties 一项中查看 Qt 样式表所支持的所有属性。

2. 选择符类型

Qt 样式表支持在 CSS2 中定义的所有选择符。表 8-3 列出了最常用的选择符类型。

表 8-3 常用的选择符类型

选择符	示例	说 明
通用选择符	*	匹配所有部件
类型选择符	QPushButton	匹配所有 QPushButton 实例和它的所有子类
属性选择符	QPushButton[flat="false"]	匹配 QPushButton 的属性 flat 为 false 的实例
类选择符	.QPushButton	匹配所有 QPushButton 实例,但不包含它的子类
ID 选择符	QPushButton#okButton	匹配所有 QPushButton 中以 okButton 为对象名的实例
后代选择符	QDialog QPushButton	匹配所有 QPushButton 实例,它们必须是 QDialog 的子孙部件
孩子选择符	QDialog>QPushButton	匹配所有 QPushButton 实例,它们必须是 QDialog 的直接子部件

3. 子控件

一些复杂的部件修改样式可能需要访问它们的子控件 (Sub-Controls), 例如 QComboBox 的下拉按钮, 还有 QSpinBox 的向上和向下的箭头等。选择符可以包含子控件来对部件的特定子控件应用规则, 例如:

```
QComboBox::drop-down{image:url(dropdown.png)}
```

这样的规则可以改变所有 QComboBox 部件的下拉按钮的样式。Qt Style Sheets Reference 关键字对应的帮助文档的 List of Stylable Widgets 一项列出了所有可以使用样式表来自定义样式的 Qt 部件, List of Sub-Controls 一项中列出了所有可用的子控件。

4. 伪状态

选择符可以包含伪状态 (Pseudo-States) 来限制规则在部件的指定的状态上应用。伪状态出现在选择符之后, 用冒号隔离, 例如:

```
QPushButton:hover{color:white}
```

这个规则表明当鼠标悬停在一个 QPushButton 部件上时才被应用。伪状态可以使用感叹号来表示否定, 例如要当鼠标没有悬停在一个 QRadioButton 上时才应用规

则,那么这个规则可以写为:

```
QRadioButton:! hover{color:red}
```

伪状态还可以多个连用,达到逻辑与效果,例如当鼠标悬停在一个被选中的 QCheckBox 部件上时才应用规则,那么这个规则可以写为:

```
QCheckBox:hover:checked{color:white}
```

如果有需要,也可以使用逗号来表示逻辑或操作,例如:

```
QCheckBox:hover,QCheckBox:checked{color:white}
```

在 Qt Style Sheets Reference 关键字对应的帮助文档的 List of Pseudo-States 项中列出了 Qt 支持的所有的伪状态。

5. 冲突解决

当几个样式规则对相同的属性指定了不同的值时就会产生冲突。例如:

```
QPushButton#okButton { color: gray }
QPushButton { color: red }
```

这样 okButton 的 color 属性便产生了冲突。解决这个冲突的原则是:特殊的选择符优先。在这里,因为 QPushButton#okButton 一般代表一个单一的对象,而不是一个类所有的实例,所以它比 QPushButton 更特殊,那么这时便会使用第一个规则,okButton 的文本颜色为灰色。相似的,有伪状态比没有伪状态优先。如果两个选择符的特殊性相同,则后面出现的比前面的优先。Qt 样式表使用 CSS2 规范来确定规则的特殊性。

6. 层叠

样式表可以设置在 QApplication 或者父部件上或者子部件上。部件有效的样式表是通过部件祖先的样式表和 QApplication 上的样式表合并得到的。当发生冲突时,部件自己的样式表优先于任何继承的样式表,同样,父部件的样式表优先于祖先的样式表。

8.2.3 自定义部件外观与换肤

1. 盒子模型 (The Box Model)

当使用样式表时,每一个部件都看作是拥有 4 个同心矩形的盒子,如图 8-5 所示。这 4 个矩形分别是内容(content)、填衬(padding)、边框(border)和边距(margin)。边距、边框宽度和填衬等属性的默认值都是 0,这样 4 个矩形恰好重合。

可以使用 background-image 属性来为部件指定一个背景,默认的,background-image 只在边框以内的区域进行绘制,这个可以使用 background-clip 属性来进行更改。可以使用 background-repeat 和 background-origin 来控制背景图片的重复方式以及

原点。

一个 background-image 无法随着部件的大小来自动缩放,如果想要背景随着部件的大小变化,那就必须使用 border-image。如果同时指定了 background-image 和 border-image,那么 border-image 会绘制在 background-image 之上。

此外,image 属性可以用来在 border-image 之上绘制一个图片。如果使用 image 指定的图片的大小与部件的大小不匹配,那么它不会平铺或者拉伸。

图片的对齐方式可以使用 image-position 属性来设置。

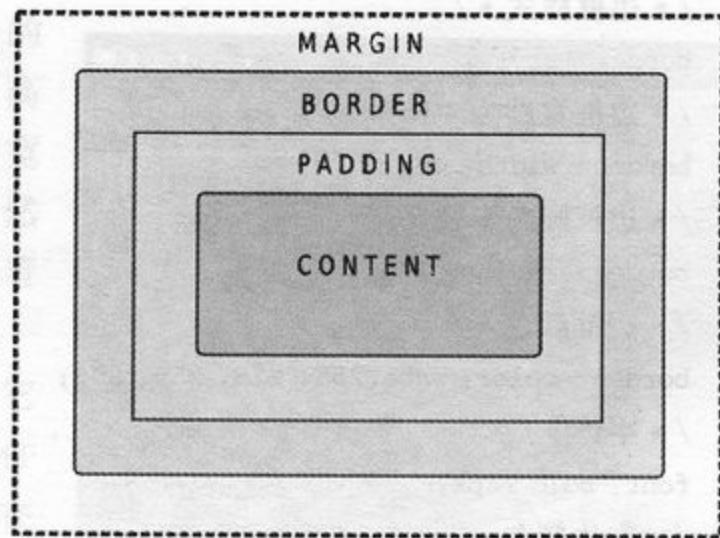


图 8-5 盒子模型示意图

2. 自定义部件外观

下面继续在 8.2.1 小节的程序中进行更改。首先向项目目录中添加几张图片(可以下载源文件来获取这里使用的图片),然后再向项目中添加一个 Qt 资源文件,名称为“myResource”。建立完成后,先添加前缀“/image”,然后将项目目录中的几个图片添加进来,最后按下 Ctrl+S 进行保存。然后进入设计模式,再次打开主界面的编辑样式表对话框,先清空以前的代码,再添加如下代码:

```
/***** 主界面背景 *****/
QMainWindow{
}
```

这里可以将光标放到第一个大括号后,然后单击“添加资源”选项后面的下拉箭头,选择“background-image”,在弹出的选择资源对话框中选择一张背景图片(注意:第一次打开资源对话框,可能无法显示资源,需要按下左上角的“重新加载”图标),这样便可以自动添加使用图片的代码。然后再更改 QPushButton 和 QSlider 的样式代码,最终的代码为:

```
/***** 主界面背景 *****/
QMainWindow{
/* 背景图片 */
background-image: url(:/image/beijing01.png);
}

/***** 按钮部件 *****/
QPushButton{
/* 背景色 */
background-color: rgba(100, 225, 100, 30);
```

```

/* 边框样式 */
border-style: outset;
/* 边框宽度为 4 像素 */
border-width: 4px;
/* 边框圆角半径 */
border-radius: 10px;
/* 边框颜色 */
border-color: rgba(255, 225, 255, 30);
/* 字体 */
font: bold 14px;
/* 字体颜色 */
color:rgba(0, 0, 0, 100);
/* 填衬 */
padding: 6px;
}

/* 鼠标悬停在按钮上时 */
QPushButton:hover{
background-color:rgba(100,255,100, 100);
border-color: rgba(255, 225, 255, 200);
color:rgba(0, 0, 0, 200);
}

/* 按钮被按下时 */
QPushButton:pressed {
background-color:rgba(100,255,100, 200);
border-color: rgba(255, 225, 255, 30);
border-style: inset;
color:rgba(0, 0, 0, 100);
}

***** 滑块部件 *****
/* 水平滑块的手柄 */
QSlider::handle:horizontal {
image: url(:/image/sliderHandle.png);
}

/* 水平滑块手柄以前的部分 */
QSlider::sub-page:horizontal {
/* 边框图片 */
border-image: url(:/image/slider.png);
}

```

下面回到设计模式，将界面上的 QPushButton 部件的大小更改为宽 120 高 40，将 horizontalSlider 部件的大小更改为宽 280 高 6。现在运行程序，拖动滑块手柄，然后按下按钮，效果如图 8-6 所示。可以看到，当一个部件获得鼠标焦点后就会显示一个虚线的边框，这很影响美观，我们要将它去掉。这个可以在设计模式中将两个部件的 focusPolicy 属性选择为 NoFocus，或者在代码中调用两个部件的 setFocusPolicy() 函数，将其参数设置为 Qt::NoFocus。现在再次运行程序，已经没有那个虚线框了。

3. 实现换肤功能

Qt 样式表可以存放在一个以.qss 为后缀的文件中，这样就可以在程序中调用不同的.qss 文件来实现换肤的功能。下面先在前面的程序中添加新文件，模板选择“概要”分类中的“文本文件”，名称为“my.qss”，建立完成后，将前面在主界面的编辑样式表对话框中的内容全部剪切到这个文件中（注意：要将编辑样式表对话框中的内容清空）。然后按下 Ctrl+S 保存该文件。下面再向项目中添加一个“my1.qss”文件，然后在其中编写另外一个样式表，最后保存该文件。

打开 myResource.qrc 资源文件，再添加一个“/qss”前缀（添加这个前缀只是为了将文件区分开），然后添加资源文件，选择项目目录下新添加的“my.qss”和“my1.qss”文件。最后按下 Ctrl+S 保存修改。

下面在 mainwindow.cpp 文件中添加头文件 #include <QFile>，然后在构造函数中添加代码：

```
QFile file(":/qss/my.qss");
//只读方式打开该文件
file.open(QFile::ReadOnly);
//读取文件全部内容，使用 tr() 函数将其转换为 QString 类型
QString styleSheet = tr(file.readAll());
//为 QApplication 设置样式表
qApp ->setStyleSheet(styleSheet);
```

这里读取了 Qt 样式表文件中的内容，然后为应用程序设置了样式表。下面再进入设计模式，将 pushButton 的文本更改为“换肤”，然后转到它的单击信号对应的槽中，更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QFile file(":/qss/my1.qss");
```

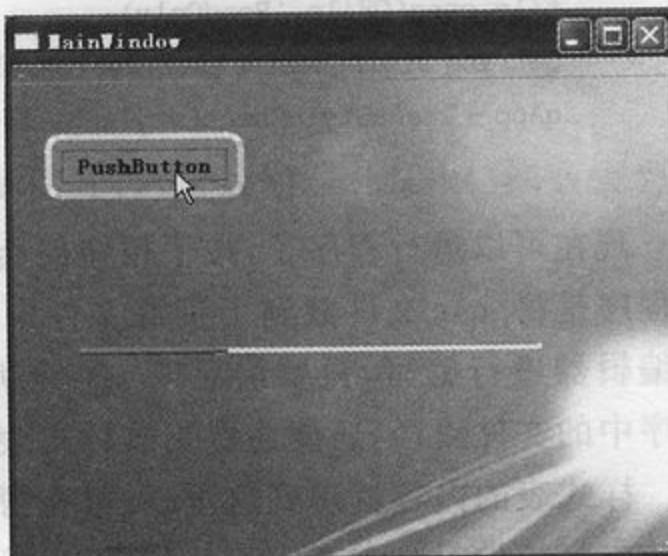


图 8-6 自定义部件外观运行效果

```

    file.open(QFile::ReadOnly);
    QString styleSheet = tr(file.readAll());
    qApp ->setStyleSheet(styleSheet);
}

```

现在可以运行程序了,按下按钮便会更改界面的外观,这样就实现了换肤功能。这个程序是将.qss文件放到了资源文件中,其实也可以放在程序外,可以使用任意的文本编辑器进行编写,只要最后以.qss为后缀保存即可。如果放在了程序外,就要更改程序中的文件路径,还要注意更改样式表中使用的图片路径。

样式表的内容就讲到这里,可以在帮助中参考 Qt Style Sheets 关键字来进行更多相关内容的学习。在 Qt Style Sheets Examples 关键字对应的文档中列举了很多常用部件的一些样式表应用范例,可以作为参考。关于样式表的使用,Qt 中还提供了一个 Style Sheet 示例程序,它在 Widgets 分类下。

8.3 特殊效果窗体

8.3.1 不规则窗体

使用样式表可以实现矩形、圆形等规则形状的部件,不过,有时想设计一个不规则形状的部件或者窗口,以使得应用程序的外观更加个性化。Qt 中提供了部件遮罩(mask)来实现不规则窗体。

(项目源码路径:src\08\8-3\myMask)新建 Qt Gui 应用,项目名称为 myMask,基类选择 QWidget,类名保持 Widget 不变。建立好项目后向项目目录中放一张背景透明的 png 图片(笔者这里是 yafeilinux.png),然后再向项目中添加一个 Qt 资源文件,建立好后先添加前缀“/image”,然后再将 png 图片添加进来并保存更改。下面进入 widget.h 文件,声明两个事件处理函数:

```

protected:
    void paintEvent(QPaintEvent * );
    void mousePressEvent(QMouseEvent * );

```

然后到 widget.cpp 文件中,先添加头文件包含:

```

#include <QPixmap>
#include <QBitmap>
#include <QPainter>

```

再在构造函数中添加如下代码:

```

QPixmap pix;
//加载图片
pix.load(":/image/yafeilinux.png");
//设置窗口大小为图片大小

```

```
resize(pix.size());
//为窗口设置遮罩
setMask(pix.mask());
```

这里使用 QPixmap 类加载了资源文件中的图片,然后调用 setMask() 函数来为窗口设置遮罩。下面是两个事件处理函数的定义:

```
void Widget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    //从窗口左上角开始绘制图片
    painter.drawPixmap(0,0,QPixmap(":/image/yafeilinux.png"));
}

void Widget::mousePressEvent(QMouseEvent *)
{
    //关闭窗口
    close();
}
```

这里必须在 paintEvent() 函数中将图片绘制在窗口上,这样运行程序时才可以正常显示图片。在鼠标按下事件中,只是进行了简单的关闭窗口操作。也可以使用第 6 章的相关知识来实现鼠标拖动窗口移动的功能。现在运行程序,效果如图 8-7 所示。

这个程序使用了一张图片来设置遮罩,其实还可以使用 QRegion 设置一个区域来作为遮罩,这个就不再讲解了。



图 8-7 不规则窗体运行效果

8.3.2 透明窗体

在讲解样式表的时候已经看到,如果想实现窗体内容部件的透明效果,只需在设置其背景色时指定 alpha 值即可,例如:

```
QPushButton{background-color:rgba(255, 255, 255, 100)}
```

其中 rgba() 中的 a 就是指 alpha, 取值为 0~255, 取值为 0 时完全透明, 取值为 255 时完全不透明。这里 a 的值为 100, 这样会出现半透明的效果, 因为前面的 r(红)、g(绿)、b(蓝) 的值均为 255, 所以是白色, 这样最终的效果是按钮的背景为半透明的白色。

部件的透明效果可以使用这种方式来设置,但是,作为顶级部件的窗口却无法使用这种方式来实现透明效果。不过,可以使用其他两种方法来实现透明效果。

(项目源码路径: src\08\8-4\myTranslucent) 新建 Qt Gui 应用, 项目名称为 myTranslucent, 基类选择 QWidget, 类名保持 Widget 不变。建好项目后, 在设计模式向界面上拖入一个 Label、Push Button 和 Progress Bar, 然后在 widget.cpp 文件中的

构造函数里添加一行代码：

```
//设置窗口的不透明度为 0.3
setWindowOpacity(0.3);
```



图 8-8 窗体半透明效果

使用 `setWindowOpacity()` 函数就可以实现窗口的透明效果，参数取值范围为 0.0~1.0，当取值为 0.0 时完全透明，取值为 1.0 时完全不透明。这时运行程序，效果如图 8-8 所示。可以看到，这样实现的效果是整个应用程序界面都是半透明的，如果不想让窗口中的部件透明，那该怎么实现呢？下面来看另一种方法。

先将构造函数中的 `setWindowOpacity()` 函数调用注释掉，然后再添加下面两行代码：

```
setWindowFlags(Qt::FramelessWindowHint);
setAttribute(Qt::WA_TranslucentBackground);
```

这里使用了 `setAttribute()` 函数指定窗口的 `Qt::WA_TranslucentBackground` 属性，它可以使窗体背景透明，而其中的部件不受影响。不过在 Windows 下，还要使用 `setWindowFlags()` 函数指定 `Qt::FramelessWindowHint` 标志，这样才能实现透明效果。运行程序，效果如图 8-9 所示。读者会发现，窗口没有了标题栏，这时要想关闭窗口，就要使用 Qt Creator 的应用程序输出栏上的红色按钮来强行关闭程序。这样实现的效果是背景完全透明的，要是还想实现半透明效果，可以使用重绘事件。

先在 `widget.h` 文件中声明 `paintEvent()` 函数：

```
protected:
    void paintEvent(QPaintEvent *);
```

然后到 `widget.cpp` 文件中添加头文件 `#include <QPainter>`，再进行 `paintEvent()` 函数的定义：

```
void Widget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.fillRect(rect(), QColor(255, 255, 255, 100));
}
```

这里先使用 `rect()` 函数获取窗口的内部矩形，它不包含任何边框。然后使用半透明的白色对这个矩形进行填充，最终的效果如图 8-10 所示。对于 `fillRect()` 函数，可以指定任意的一个区域，所以可以实现窗体的部分区域全部透明，部分区域半透明或者不透明的效果。

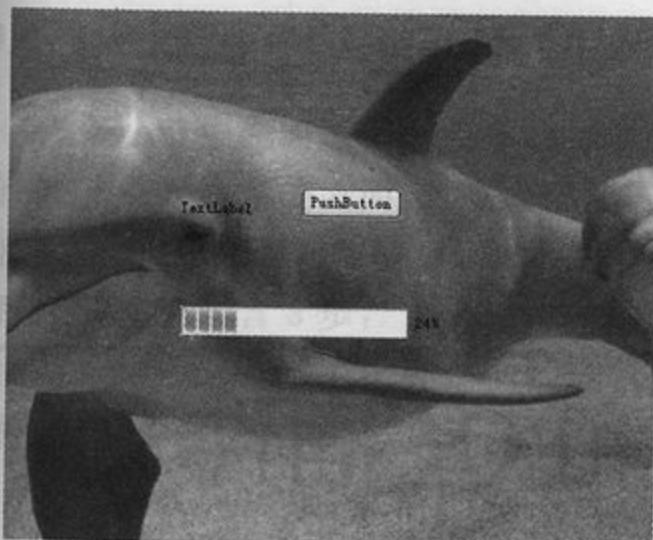


图 8-9 背景完全透明运行效果

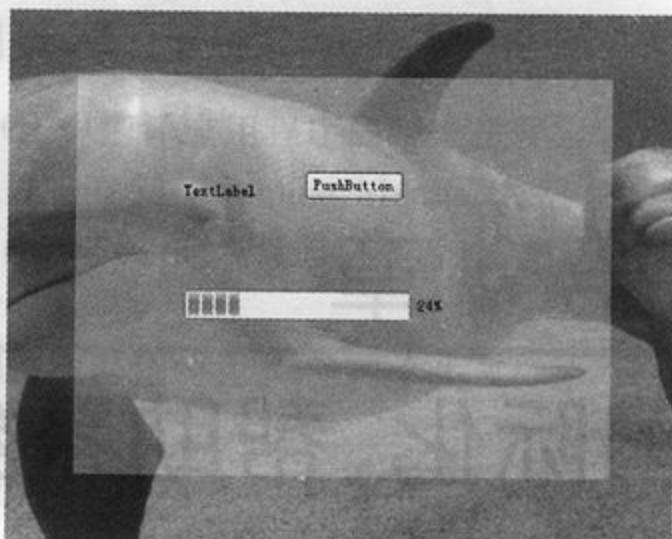


图 8-10 背景半透明运行效果

使用第一种方法会使整个应用程序都成为半透明效果；而使用第二种方法，可以实现只是顶层窗口的背景透明，不过，它没有了标题栏和边框，需要手动为其添加一个标题栏。其实，对于一个个性化的窗体界面，没有标题栏也许正是想要的结果。

另外，使用第 11 章讲到的图形效果也可以实现部件的透明效果，而且使用它还可以实现模糊、阴影和染色等特殊效果。

8.4 小结

学习完这一章，读者要掌握最基本的更改部件样式的方法，而且应该可以实现一些简单的界面效果。重点是 Qt 样式表，但是因为外观设计应用到了很多美学方面的知识，而且涉及很多 CSS 的应用，这里并没有深入讲解。如果想设计出非常漂亮的界面，那么还是需要有相关的知识基础的。

第 9 章

国际化、帮助系统和 Qt 插件

这一章是基本应用篇的最后一章,介绍 Qt 的国际化、帮助系统和创建插件等方面的内容。

9.1 国际化

国际化的英文表述为 Internationalization,通常简写为 I18N(首尾字母加中间的字符数),一个应用程序的国际化就是使该应用程序可以让其他国家的用户使用的过程。Qt 支持现在使用的大多数语言,特别是:

- 所有东亚语言(汉语、日语和朝鲜语);
- 所有西方语言(使用拉丁字母);
- 阿拉伯语;
- 西里尔语言(俄语和乌克兰语等);
- 希腊语;
- 希伯来语;
- 泰语和老挝语;
- 所有在 Unicode 5.1 中不需要特殊处理的脚本。

在 Qt 中,所有的输入部件和文本绘制方式对 Qt 所支持的所有语言都提供了内置的支持。Qt 内置的字体引擎可以在同一时间正确而且精细地绘制不同的文本,这些文本可以包含来自众多不同书写系统的字符。相关知识可以在帮助中查看 Internationalization with Qt 关键字。

Qt 对把应用程序翻译为本地语言提供了很好的支持,可以使用 Qt Linguist 工具很容易地完成应用程序的翻译工作,这个工具在第 1 章就已经介绍过了,这里进一步介绍。

9.1.1 使用 Qt Linguist 翻译应用程序

这一小节中先通过一个简单的例子介绍 Qt 中翻译应用程序的整个过程,然后再介绍其中需要注意的方面,对应这部分内容可以查看 Qt Linguist Manual 关键字。在 Qt 中编写代码时要对需要显示的字符串调用 `tr()` 函数,完成代码编写后对这个应用程序的翻译主要包含 3 步:

① 运行 `lupdate` 工具从 C++ 源代码中提取要翻译的文本,这时会生成一个 `.ts` 文件,这个文件是 XML 格式的。

② 在 Qt Linguist 中打开 `.ts` 文件,并完成翻译工作。

③ 运行 `lrelease` 工具从 `.ts` 文件中获得 `.qm` 文件,它是一个二进制文件。这里的 `.ts` 文件是供翻译人员使用的,而在程序运行时只需要使用 `.qm` 文件,这两个文件都是与平台无关的。

下面通过一个简单的例子来介绍整个翻译过程,该例子实现了将一个英文版本的应用程序翻译为简体中文版本。(项目源码路径: `src\09\9-1\myI18N`。)

第一步,编写源码。新建 Qt Gui 应用,项目名称为 `myI18N`,类名为 `MainWindow`,基类保持 `QMainWindow` 不变。建立完项目后,单击 `mainwindow.ui` 文件进入设计模式,先添加一个“&File”菜单,再为其添加一个“&New”子菜单并设置快捷键为 `Ctrl+N`,然后往界面上拖入一个 Push Button。再使用代码添加几个标签,打开 `mainwindow.cpp` 文件,添加头文件 `#include <QLabel>`,然后在构造函数中添加代码:

```
QLabel * label = new QLabel(this);
label->setText(tr("hello Qt!"));
label->move(100,50);

QLabel * label2 = new QLabel(this);
label2->setText(tr("password","mainwindow"));
label2->move(100,80);

QLabel * label3 = new QLabel(this);
int id = 123;
QString name = "yafei";
label3->setText(tr("ID is %1,Name is %2").arg(id).arg(name));
label3->resize(150,12);
label3->move(100,120);
```

这里向界面上添加了 3 个标签,因为这 3 个标签中的内容都是用户可见的,所以需要调用 `tr()` 函数。在 `label2` 中调用 `tr()` 函数时,还使用了第二个参数,其实 `tr()` 函数一共有 3 个参数,它的原型如下:

```
QString QObject::tr ( const char * sourceText, const char * disambiguation = 0, int n = -1 ) [static]
```

第一个参数 sourceText 就是要显示的字符串, tr() 函数会返回 sourceText 的译文;第二个参数 disambiguation 是消除歧义字符串,比如这里的 password,如果一个程序中需要输入多个不同的密码,那么在没有上下文的情况下,就很难确定这个 password 到底指哪个密码。这个参数一般使用类名或者部件名,比如这里使用了 mainwindow,就说明这个 password 是在 mainwindow 上的;第三个参数 n 表明是否使用了复数,因为英文单词中复数一般要在单词末尾加“s”,比如“1 message”,复数时为“messages”。遇到这种情况,就可以使用这个参数,它可以根据数值来判断是否需要添加“s”,例如:

```
int n = messages.count();
showMessage(tr("%n message(s) saved", "", n));
```

关于 tr() 函数 3 个参数更多的用法介绍,可以在帮助中查看 Writing Source Code for Translation 关键字。

第二步,更改项目文件。要在项目文件中指定生成的. ts 文件,每一种翻译语言对应一个. ts 文件。打开 myI18N.pro 文件,在最后面添加如下一行代码:

```
TRANSLATIONS = myI18N_zh_CN.ts
```

这表明后面生成的. ts 文件的文件名为“myI18N_zh_CN.ts”,对于. ts 的名称可以随意编写,不过一般是以区域代码来结尾,这样可以更好地区分,例如这里使用了“zh_CN”来表示简体中文。最后按下 Ctrl+S 保存该文件(这个很重要,不然无法进行下面的操作)。

第三步,使用 lupdate 生成. ts 文件。要进行翻译工作时,先要使用 lupdate 工具来提取源代码中的翻译文本,生成. ts 文件。在项目文件列表的 myI18N.pro 文件上右击,在弹出的菜单中选择“在此打开命令行控制台”。这时打开的命令行控制台中已经自动切换到了项目目录下,这时输入下面一行代码,并按回车键:

```
lupdate myI18N.pro
```

注意,这里要确保已经将 Qt 安装目录的 bin 目录的路径添加到了系统 PATH 环境变量中。如果还没有设置 PATH 环境变量,可以按照 2.2.1 小节的方法设置,然后重启 Qt Creator,再进行该步骤。最后的结果表明更新了“myI18N_zh_CN.ts”文件,发现了 8 个源文本,其中有 8 条新的翻译和 0 条已经存在的翻译。这就是说可以对程序代码进行更改,然后多次运行 lupdate,而只需要翻译新添加的内容。可以在项目目录中使用写字板打开这个. ts 文件,可以看到它是 XML 格式的,其中记录了字符串的位置和是否已经被翻译等信息。

第四步,使用 Qt Linguist 完成翻译。这一步一般是翻译人员来做的,就是在 Qt Linguist 中打开. ts 文件,然后对字符串逐个进行翻译。在系统的开始菜单中启动 Linguist(也可以直接在命令行输入 linguist 或者在 Qt 安装目录的 tools 目录下找到并启动它),然后单击界面左上角的 open 图标,在弹出的文件对话框中进入项目目录,打开

my18N_zh_CN文件,这时整个界面如图9-1所示。Qt Linguist窗口主要由以下几部分组成:

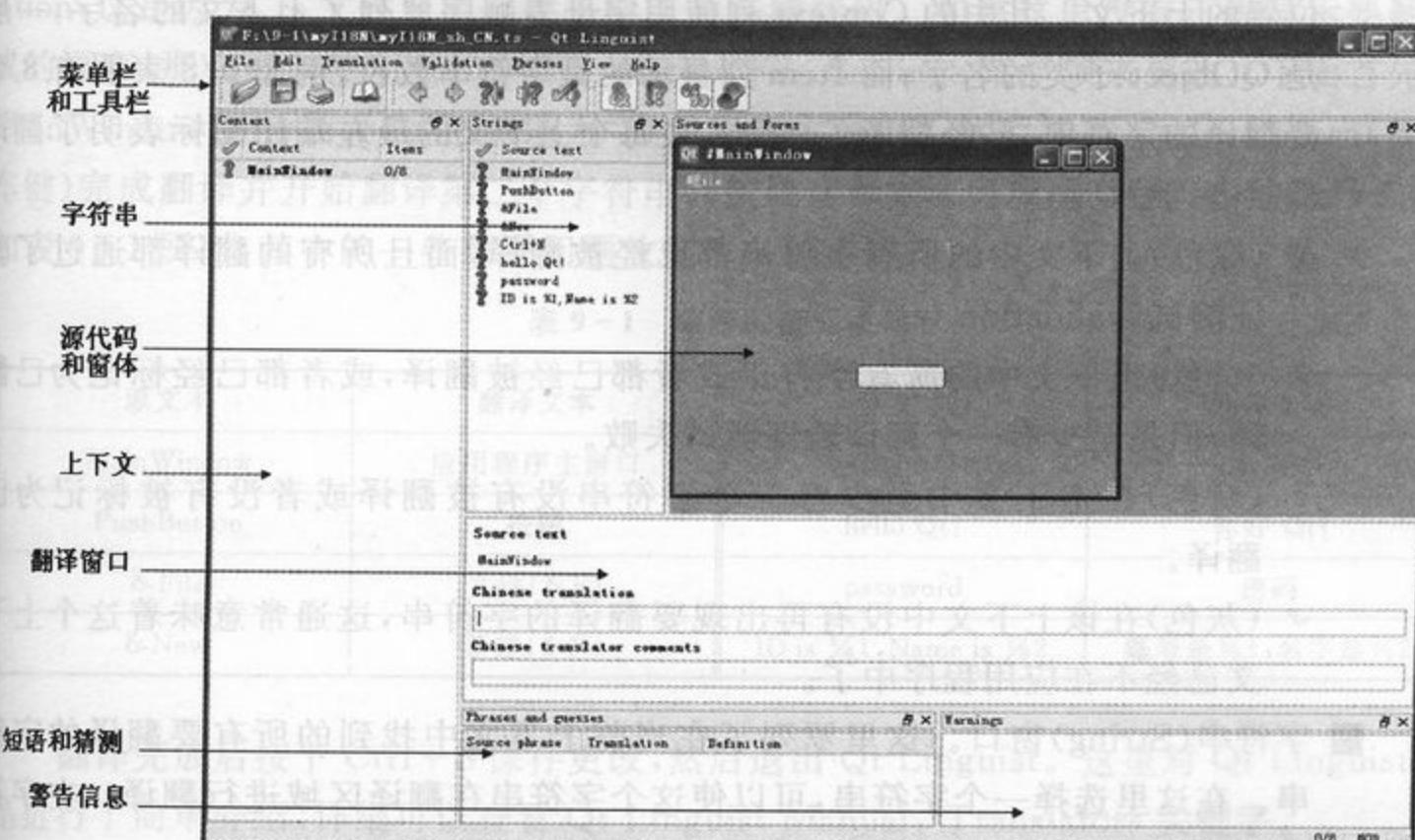


图9-1 Qt Linguist界面

■ 菜单栏和工具栏。在菜单栏中列出了Qt Linguist的所有功能选项,而工具栏中列出了常用的一些功能,其中后面9个图标的功能为:

- ◆ 在字符串列表中移动到前一个条目。
- ◆ 在字符串列表中移动到下一个条目。
- ◆ 在字符串列表中移动到前一个没有完成翻译的条目。
- ◆ 在字符串列表中移动到下一个没有完成翻译的条目。
- ◆ 标记当前条目为完成翻译状态,然后移动到下一个没有完成翻译的条目。
- ◆ 打开或关闭加速键(accelerator)验证(validation)。打开加速键验证可以验证加速键是否被翻译,例如字符串中包含“&”符号,但是翻译中没有包含“&”符号,则验证失败。
- ◆ 打开或关闭短语结束标点符号验证。打开短语标点符号验证可以验证翻译中是否使用了和字符串中相同的标点来结尾。
- ◆ 打开或关闭短语参考(phrase book)验证。打开短语参考验证可以验证翻译是否和短语参考中的翻译相同。在翻译相似的程序时,若希望将常用的翻译记录下来,以便以后使用,就可以使用短语参考。可以通过Phrases→New Phrase Book菜单项来创建一个新的短语参考,然后再翻译字符串时使用Ctrl+T将这个字符串及其翻译放入短语参考中。
- ◆ 打开或关闭占位符(place marker)验证。打开占位符验证可以验证翻译中

是否使用了和字符串中相同的占位符,例如%1、%2等。

■ 上下文(Context)窗口。这里是一个上下文列表,罗列了要翻译的字符串所在位置的上下文。其中的 Context 列使用字母表顺序罗列了上下文的名字,一般是 QObject 子类的名字;而 Item 列显示的是字符串数目,例如 0/8 表明有 8 个要翻译的字符串,已经翻译了 0 个。在每个上下文的最左端用图标表明了翻译的状态,含义是:

- (绿色)上下文中的所有字符串都已经被翻译,而且所有的翻译都通过了验证测试(validation test)。
- ✓ (黄色)上下文中的所有字符串或者都已经被翻译,或者都已经标记为已翻译,但是至少有一个翻译验证测试失败。
- ? (黄色)在上下文中至少有一个字符串没有被翻译或者没有被标记为已翻译。
- ✓ (灰色)在该上下文中没有再出现要翻译的字符串,这通常意味着这个上下文已经不在应用程序中了。

■ 字符串(String)窗口。这里罗列了在当前上下文中找到的所有要翻译的字符串。在这里选择一个字符串,可以使这个字符串在翻译区域进行翻译。在字符串左边使用图标表明了字符串的状态,它们的含义是:

- ✓ (绿色)源字符串已经翻译(可能为空),或者用户已经接受翻译,而且翻译通过了所有验证测试。
- ✓ (黄色)用户已经接受了翻译,但是翻译没有通过所有的验证测试。
- ? (黄色)字符串已经拥有了一个通过了所有验证测试的非空翻译,但是用户还没有接受该翻译。
- ? (棕色)字符串还没有翻译。
- ! (红色)字符串拥有一个翻译,但是这个翻译没有通过所有的验证测试。
- ✓ (灰色)字符串已经过时,它已经不在该上下文中。

■ 源代码和窗体(Sources and Forms)窗口。如果包含有要翻译字符串的源文件在 Qt Linguist 中可用,那么这个窗口会显示当前字符串在源文件中的上下文。

■ 翻译区域(The Translation Area)。在字符串列表中选择的字符串会出现在翻译区域的最顶端的“Source Text”下面;如果在使用 tr() 函数时设置了第二个参数消除歧义注释,那么这里还会在“Developer comments”下出现该注释;后面的“translation”中可以输入翻译文本,如果文本中包含空格,会使用“.”显示;最后面的“translator comments”中可以填写翻译注释文本。

■ 短语和猜测(Phrases and Guesses)窗口。如果字符串列表中的当前字符串出现在了已经加载的短语参考中,那么当前字符串和它在短语参考中的翻译会被罗列在这个窗口。在这里可以双击翻译文本,这样翻译文本就会复制到翻译区域。

■ 警告信息(Warnings)窗口。如果输入的当前字符串的翻译没有通过开启的验

证测试，那么在这里会显示失败信息。

下面来翻译程序。在翻译区域可以看到现在已经是是要翻译成中文 Chinese translation，这是因为.ts 文件名中包含了中文的区域代码。如果这里没有正确显示要翻译成的语言，那么可以使用 Edit→Translation File Settings 菜单项来更改。下面首先对 MainWindow 进行翻译，这里翻译为“应用程序主窗口”，然后按下 Ctrl+Return(即回车键)完成翻译并开始翻译第二个字符串。按照这种方法完成所有字符串的翻译工作，如表 9-1 所列，其中的一些翻译问题放到下一节再讲。

表 9-1 程序的翻译文本

原文本	翻译文本	原文本	翻译文本
MainWindow	应用程序主窗口	Ctrl+N	Ctrl+N
PushButton	按钮	hello Qt!	你好 Qt!
&File	文件(&F)	password	密码
&New	新建(&N)	ID is %1, Name is %2	账号是%1,名字是%2

翻译完成后按下 Ctrl+S 保存更改，然后退出 Qt Linguist。这里对 Qt Linguist 只是进行了简单介绍，详细可以查看 Qt Linguist Manual: Translators 关键字。

第五步，使用 lrelease 生成.qm 文件。在命令行输入如下一行代码，并按下回车键：

```
lrelease myI18N.pro
```

输出信息如图 9-2 所示，这表明已经生成了.qm 文件。另外在 Qt Linguist 中也可以使用 File→Reelase 和 File→Release As 这两个菜单项来生成当前已打开的.ts 文件对应的.qm 文件。

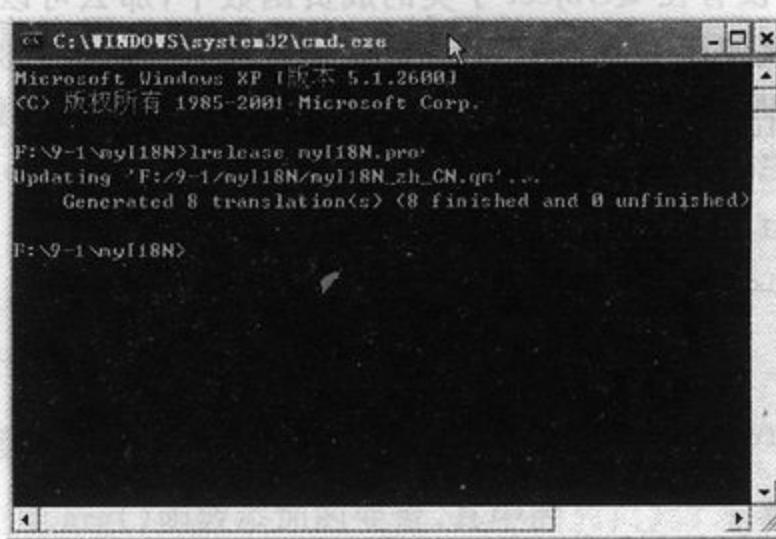


图 9-2 在命令行运行 lrelease 工具

第六步，使用.qm 文件。下面在项目中添加代码使用.qm 文件来更改界面的语言。进入 main.cpp 文件，添加头文件 #include <QTranslator>，然后在“QApplica-

tion a(argc, argv);”代码下添加如下代码：

```
QTranslator translator;
translator.load("../myI18N/myI18N_zh_CN.qm");
a.installTranslator(&translator);
```

这里先加载了. qm 文件(使用了相对路径),然后为 QApplication 对象安装了翻译。这里还要说明一点,有时可能因为部件的大小问题使得翻译后的文本无法完全显示,较好地解决方法就是使用布局管理器。

9.1.2 程序翻译中的相关问题

1. 对所有用户可见的文本使用 QString

因为 QString 内部使用了 Unicode 5.1 编码,世界上所有的语言都可以使用熟悉的文本处理操作来处理。而且,因为所有的 Qt 函数都使用 QString 作为参数来向用户呈现文本内容,所以没有 char * 到 QString 的转换开销。

2. 对所有文字文本使用 tr() 函数

无论什么时候使用要呈现给用户的文本,都要使用 Object::tr() 函数进行处理。例如:

```
LoginWidget::LoginWidget()
{
    QLabel * label = new QLabel(tr("Password:"));
    ...
}
```

如果引用的文本没有在 QObject 子类的成员函数中,那么可以使用一个合适的类的 tr() 函数,或者直接使用 QCoreApplication::translate() 函数。例如:

```
void some_global_function(LoginWidget * logwid)
{
    QLabel * label = new QLabel(
        LoginWidget::tr("Password:"), logwid);
}

void same_global_function(LoginWidget * logwid)
{
    QLabel * label = new QLabel(
        qApp ->translate("LoginWidget", "Password:"), logwid);
}
```

如果要在不同的函数中使用要翻译的文本,那么可以使用 QT_TR_NOOP() 宏和 QT_TRANSLATE_NOOP() 宏,它们仅仅对该文本进行标记来方便 lupdate 工具进行

提取。使用 QT_TR_NOOP()的例子：

```
QString FriendlyConversation::greeting(int type)
{
    static const char * greeting_strings[] = {
        QT_TR_NOOP("Hello"),
        QT_TR_NOOP("Goodbye")
    };
    return tr(greeting_strings[type]);
}
```

使用 QT_TRANSLATE_NOOP()的例子：

```
static const char * greeting_strings[] = {
    QT_TRANSLATE_NOOP("FriendlyConversation", "Hello"),
    QT_TRANSLATE_NOOP("FriendlyConversation", "Goodbye")
};

QString FriendlyConversation::greeting(int type)
{
    return tr(greeting_strings[type]);
}

QString global_greeting(int type)
{
    return qApp ->translate("FriendlyConversation",
                           greeting_strings[type]);
}
```

3. 对加速键的值使用 QKeySequence()函数

类似于 Ctrl+Q 或者 Alt+F 等加速键的值也需要翻译。如果使用了硬编码的 Qt::CTRL+Qt::Key_Q 作为退出操作的快捷键，那么翻译将无法覆盖它。正确的习惯用法为：

```
exitAct = new QAction(tr("E&xit"), this);
exitAct ->setShortcuts(QKeySequence::Quit);
```

4. 对动态文本使用 QString::arg()函数

对于字符串中使用 arg()函数添加的变量，其中的 %1、%2 等参数的顺序在翻译时可以改变，它们对应的值不会改变。

5. 支持编码

QTextCodec 类以及 QTextStream 中的设施使得 Qt 可以很容易支持大量用户数据的输入和输出编码。当一个应用程序启动时，机器的语言环境便会决定当处理 8 位

数据(例如字体选择、文本显示、8位文本 I/O 和字符输入等)时使用的 8 位编码。

应用程序有时也需要使用默认的本地 8 位编码以外的编码,例如使用 ISO 8859-5 编码,代码如下:

```
QString string = ...; //一些 Unicode 文本
QTextCodec *codec = QTextCodec::codecForName("ISO 8859 - 5");
QByteArray encodedString = codec ->fromUnicode(string);
```

如果要将 Unicode 转换为本地 8 位编码,那么可以使用 `QString::toLocal8Bit()` 函数;还有一个 `QString::toUtf8()` 函数可以返回使用 8 位 UTF-8 编码的文本。要将其他编码转换为 Unicode 编码,那么可以使用 `QTextCodec` 类的 `toUnicode()` 函数、`QString::fromUtf8()` 和 `QString::fromLocal8Bit()` 等。

使用 `QTextCodec::codecForLocale()` 返回的编码是最重要的一个编码,因为这个是用户与他人或者其他应用程序进行通信最有可能使用的编码。

6. 翻译非 Qt 类

如果要使一个类中的字符串支持国际化,那么该类或者继承自 `QObject` 类或者使用 `Q_OBJECT` 宏。而对于非 Qt 类,如果要支持翻译,需要在类定义的开始使用 `Q_DECLARE_TR_FUNCTIONS()` 宏,例如:

```
class MyClass
{
    Q_DECLARE_TR_FUNCTIONS(MyClass)

public:
    MyClass();
    ...
};
```

这样就可以在该类中使用 `tr()` 函数了。

7. 自动判断语言环境

如果在一个程序中提供了多种语言选择,最好的方法就是在程序启动时判断本地的语言环境,然后加载对应的 `.qm` 文件。可以使用 `QLocale::system().name()` 来获取本地的语言环境,它会返回 `QString` 类型的“语言_国家”格式的字符串,其中的语言用两个小写字母表示,符合 ISO 639 编码;国家使用两个大写字母表示,符合 ISO 3166 国家编码。例如中国简体中文的表示为“zh_CN”。可以使用这个返回值来调用不同的文件,使应用程序自动使用相应的语言。

在国际化中还有本地化、在应用程序运行时动态进行语言更改等内容,这里就不再涉及。感兴趣的读者可以在帮助中参考 Internationalization with Qt 关键字。Qt 示例程序的 Qt Linguist 分类中提供了几个关于国际化的例子,在 Tools 分类中还有一个

Internationalization 的例子，也可以参考一下。

9.2 帮助系统

一个完善的应用程序应该提供尽可能丰富的帮助信息。在 Qt 中可以使用工具提示、状态提示以及“What’s This”等简单的帮助提示，也可以使用 Qt Assistant 来提供强大的在线帮助。

9.2.1 简单的帮助提示

第 5 章已经讲到了工具提示和状态提示，这里简单介绍“What’s This”帮助提示。运行一个对话框窗口时会看到在标题栏中有一个“?”图标，按下它就会进入“What’s This”模式，这时如果哪个部件设置了“What’s This”帮助提示，那么当鼠标移动到它上面时就会弹出一个悬浮的文本框显示相应的帮助提示。下面来看一个具体例子。

(项目源码路径：src\09\9-2\myWhatsThis)新建 Qt Gui 应用，项目名称为 myWhatsThis，类名为 MainWindow，基类保持 QMainWindow 不变。建立完项目后，单击 mainwindow.ui 文件进入设计模式。在界面上右击，在弹出的菜单中选择“改变‘这是什么’”项，则弹出“编辑这是什么”对话框，可以在这里输入文本或者添加图片来设置“What’s This”帮助提示。这里输入“这是主窗口”，然后将文本改为红色，最后单击“确定”按钮关闭该对话框。现在运行程序，按下 Shift+F1 键就可以显示提示信息了。

有时想添加一个“?”图标来进入“What’s This”模式，这可以通过在代码中使用 QWhatsThis 类来实现。现在进入 mainwindow.cpp 文件中，先添加头文件 #include <QWhatsThis>，然后在构造函数中添加如下代码：

```
QAction * action = QWhatsThis::createAction(this);
ui ->mainToolBar ->addAction(action);
```

这里使用了 QWhatsThis 类的 createAction() 函数来创建了一个“What’s This”图标，然后将它添加到工具栏中。运行程序，按下“What’s This”图标，并在主界面上单击就可以显示提示信息了。另外，QWhatsThis 类还提供了 enterWhatsThisMode() 来进入“What’s This”模式。要为一个部件提供“What’s This”提示，也可以在代码中通过调用该部件的 setWhatsThis() 函数来实现。

要提供详细的功能和使用帮助，则需要使用 HTML 格式的帮助文本。在程序中可以通过调用 Web 浏览器或者使用 QTextBrowser 来管理和应用这些 HTML 文件。不过，Qt 提供了更加强大的工具，那就是 Qt Assistant，它支持索引和全文检索，而且可以为多个应用程序同时提供帮助，可以通过定制 Qt Assistant 来实现强大的在线帮助系统。

9.2.2 定制 Qt Assistant

为了将 Qt Assistant 定制为我们自己的应用程序的帮助浏览器，需要先进行一些

准备工作,主要是生成一些文件,最后再在程序中启动 Qt Assistant。主要的步骤如下:

- ① 创建 HTML 格式的帮助文档;
- ② 创建 Qt 帮助项目(Qt help project). qhp 文件,该文件是 XML 格式的,用来组织文档,并且使它们可以在 Qt Assistant 中使用;
- ③ 生成 Qt 压缩帮助(Qt compressed help). qch 文件,该文件由. qhp 文件生成,是二进制文件;
- ④ 创建 Qt 帮助集合项目(Qt help collection project). qhcp 文件,该文件是 XML 格式的,用来生成下面的. qhc 文件;
- ⑤ 生成 Qt 帮助集合(Qt help collection). qhc 文件,该文件是二进制文件,可以使 Qt Assistant 只显示一个应用程序的帮助文档,也可以定制 Qt Assistant 的外观和一些功能;
- ⑥ 在程序中启动 Qt Assistant。

下面通过一个具体例子来讲解整个过程。这里还在前一节的程序的基础上进行更改。

第一步,创建 HTML 格式的帮助文档。可以通过各种编辑器例如 Microsoft Word 来编辑要使用的文档,最后保存为 HTML 格式的文件,例如这里创建了 5 个 HTML 文件。然后在项目目录中新建文件夹,命名为 documentation,再将这些 HTML 文件放入其中。再在 documentation 文件夹中新建一个 images 文件夹,往里面复制一个图标图片,以后将作为 Qt Assistant 的图标,例如这里使用了 yafeilinux.png 图片。

第二步,创建. qhp 文件。首先在 documentation 文件夹中创建一个文本文件,然后进行编辑,最后另存为 myHelp. qhp,注意后缀为“. qhp”。文件的内容如下:

```
<? xml version = "1.0" encoding = "GB2312"? >
<QtHelpProject version = "1.0">
<namespace>yafeilinux. myHelp</namespace>
<virtualFolder>doc</virtualFolder>
<filterSection>
  <toc>
    <section title = "我的帮助" ref = "./index.html">
      <section title = "关于我们" ref = "./aboutUs.html">
        <section title = "关于 yafeilinux" ref = "./about_yafeilinux.html"></section>
        <section title = "关于 Qt Creator 系列教程" ref = "./about_QtCreator.html">
      </section>
    </toc>
    <section title = "加入我们" ref = "./joinUs.html"></section>
  </section>
</filterSection>
```

```

<keywords>
    <keyword name = "关于" ref = "./aboutUs.html"/>
    <keyword name = "yafeilinux" ref = "./about_yafeilinux.html"/>
    <keyword name = "Qt Creator" ref = "./about_QtCreator.html"/>
</keywords>
<files>
    <file>about_QtCreator.html</file>
    <file>aboutUs.html</file>
    <file>about_yafeilinux.html</file>
    <file>index.html</file>
    <file>joinUs.html</file>
    <file>images/* .png</file>
</files>
</filterSection>
</QtHelpProject>

```

这个.qhp 文件是 XML 格式的。第一行是 XML 序言,这里指定了编码 encoding 为 GB2312,这样就可以使用中文了,如果只想使用英文,那么编码一般是 UTF-8;第二行指定了 QtHelpProject 版本为 1.0;第三行指定了命名空间 namespace,每一个.qhp 文件的命名空间都必须是唯一的,命名空间会成为 Qt Assistant 中的页面的 URL 的第一部分,这个在后面的内容中会涉及;第四行指定了一个虚拟文件夹 virtualFolder,这个文件夹并不需要创建,只用来区分文件;再下面的过滤器部分 filterSection 标签包含了目录表、索引和所有文档文件的列表。过滤器部分可以设置过滤器属性,这样以后可以在 Qt Assistant 中通过过滤器来设置文档的显示与否,不过,因为这里只有一个文档,所以不需要 Qt Assistant 的过滤器功能,这里也就不需要设置过滤器属性;在目录表 toc(table of contents)标签中创建了所有 HTML 文件的目录,指定了它们的标题和对应的路径,这里设定的目录表为:

■ 我的帮助

■ 关于我们

■ 关于 yafeilinux

■ 关于 Qt Creator 系列教程

■ 加入我们

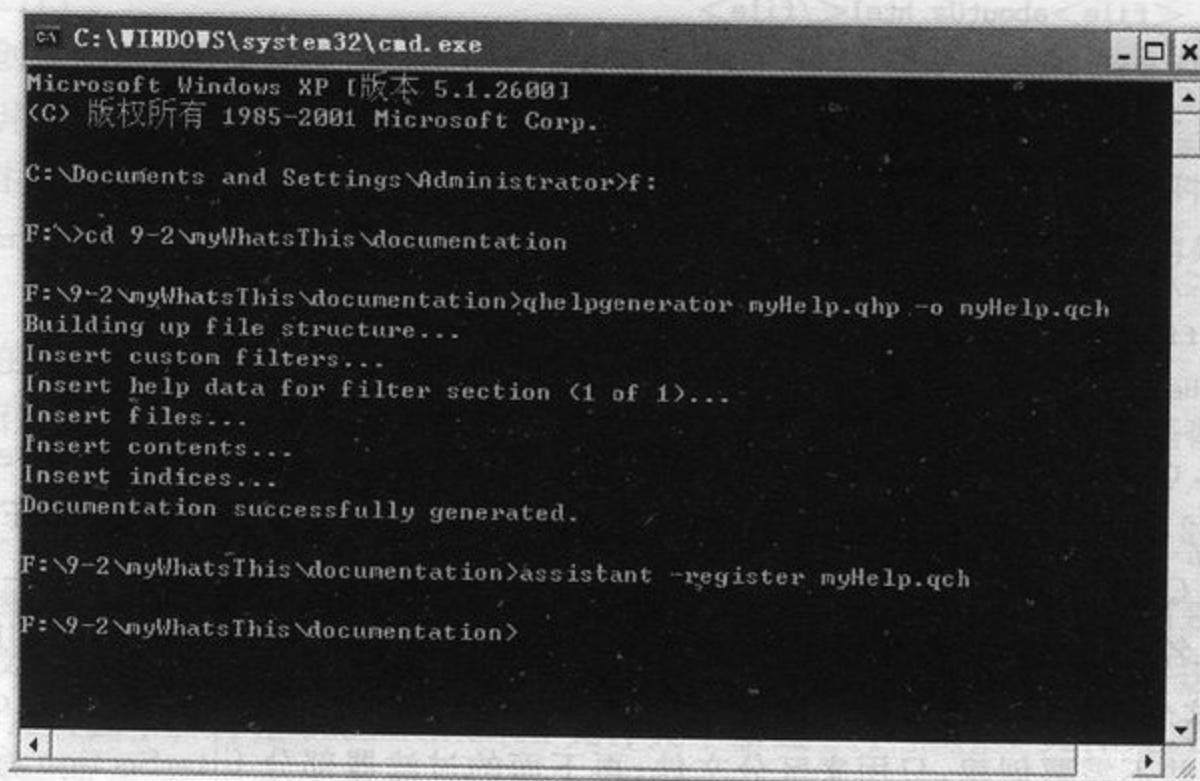
然后是 keywords 标签,它指定了所有索引的关键字和对应的文件,这些关键字会显示在 Qt Assistant 的索引页面;files 标签中列出了所有的文件,也包含图片文件。

第三步,生成.qch 文件。这里为了测试创建的文件是否可用,可以先生成.qch 文件,然后在 Qt Assistant 中注册它。这样运行 Qt Assistant 就会看到添加的文档了。不过,这一步不是必须的。打开命令行控制台,然后使用 cd 命令跳转到项目目录的 documentation 目录中,分别输入下面的命令后按下回车:

```
qhelpgenerator myHelp.qhp -o myHelp.qch
```

```
assistant -register myHelp.qch
```

要保证命令可以正常运行,前提是已经将 Qt 安装目录的 bin 目录的路径添加到了系统的 PATH 环境变量中。命令运行结果如图 9-3 所示。注册成功时,会显示“Documentation successfully registered”提示对话框。这时在开始菜单中启动 Qt Assistant(或者直接在命令行输入 assistant 来启动 Qt Assistant),可以发现已经出现了我们的 HTML 文档,如图 9-4 所示。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>f:
F:>cd 9-2\myWhatsThis\documentation

F:\9-2\myWhatsThis\documentation>qhelpgenerator myHelp.qhp -o myHelp.qch
Building up file structure...
Insert custom filters...
Insert help data for filter section <1 of 1>...
Insert files...
Insert contents...
Insert indices...
Documentation successfully generated.

F:\9-2\myWhatsThis\documentation>assistant -register myHelp.qch
F:\9-2\myWhatsThis\documentation>
```

图 9-3 在命令行生成 qch 文件

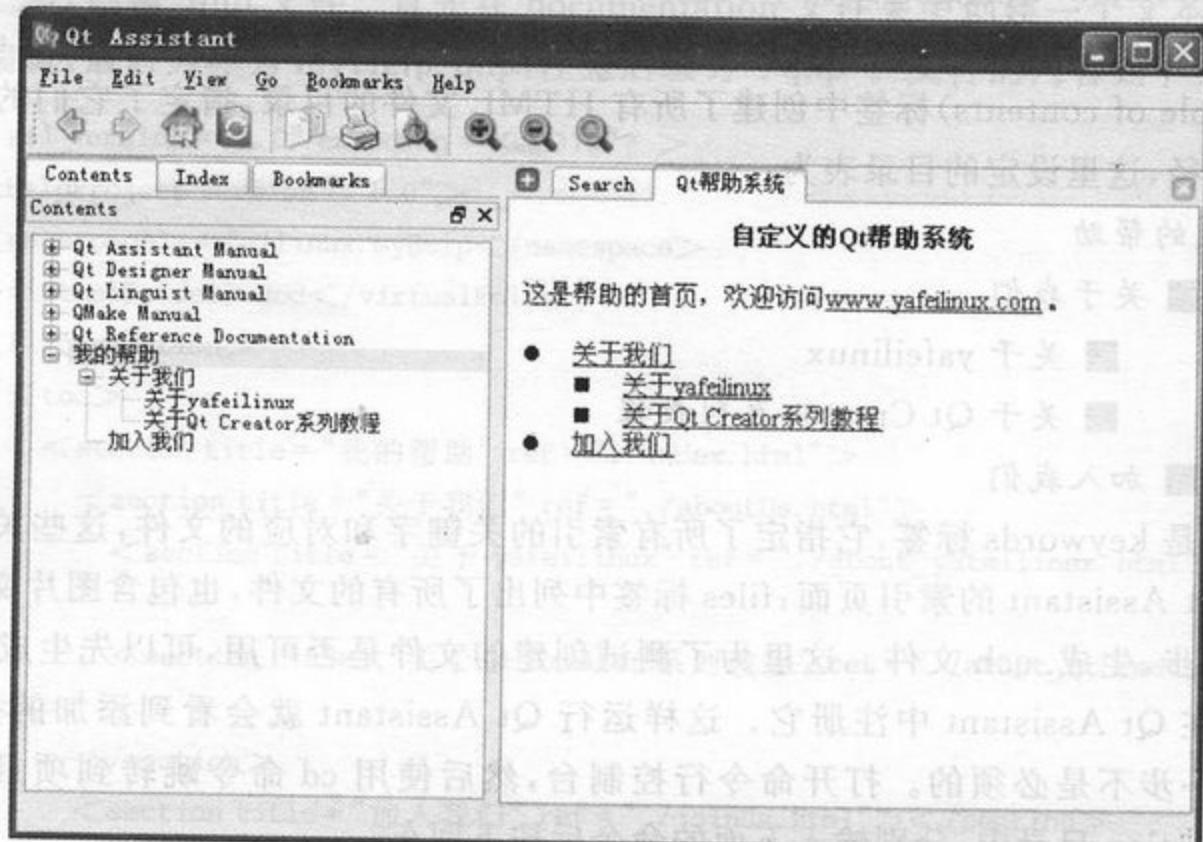


图 9-4 Qt Assistant 运行效果

第四步，创建.qhcp文件。要想使Qt Assistant只显示我们自己的帮助文档的最简单方法就是生成帮助集合文件即.qhc文件，要生成.qhc文件，首先要创建.qhcp文件。在documentation文件夹中新建文本文档对其进行编辑，最后另存为“myHelp.qhcp”，注意后缀为“.qhcp”。这里还要创建一个名为“about.txt”的文本文件，在其中输入一些该帮助的说明信息，作为Qt Assistant的About菜单的显示内容。myHelp.qhcp文件的内容如下：

```

<? xml version = "1.0" encoding = "GB2312"? >
<QHelpCollectionProject version = "1.0">
<assistant>
    <title>我的帮助系统</title>
    <applicationIcon>images/yafeilinux.png</applicationIcon>
    <cacheDirectory>cache/myHelp</cacheDirectory>
    <homePage>qthelp://yafeilinux.myHelp/doc/index.html</homePage>
    <startPage>qthelp://yafeilinux.myHelp/doc/index.html</startPage>
    <aboutMenuText>
        <text>关于该帮助</text>
    </aboutMenuText>
    <aboutDialog>
        <file>./about.txt</file>
        <icon>images/yafeilinux.png</icon>
    </aboutDialog>
    <enableDocumentationManager>false</enableDocumentationManager>
    <enableAddressBar>false</enableAddressBar>
    <enableFilterFunctionality>false</enableFilterFunctionality>
</assistant>
<docFiles>
    <generate>
        <file>
            <input>myHelp.qhp</input>
            <output>myHelp.qch</output>
        </file>
    </generate>
    <register>
        <file>myHelp.qch</file>
    </register>
</docFiles>
</QHelpCollectionProject>

```

assistant标签中对Qt Assistant的外观和功能进行了定制，其中设置了标题、图标、缓存目录、主页、起始页、About菜单文本、关于对话框的内容和图标等，还关闭了一些没有用的功能。缓存目录cacheDirectory是进行全文检索等操作时缓存文件要存放的位置。对于主页homePage和起始页startPage，这里使用了第二步中提到的Qt As-

sistant 页面的 URL,这个 URL 由“qthelp://”开始,然后是在. qhp 文件中设置的命名空间,然后是虚拟文件夹,最后是具体的 HTML 文件名。因为 Qt Assistant 可以添加或者删除文档来为多个应用程序提供帮助,但是这里只是为一个应用程序提供帮助,并且不希望删除我们的文档,所以禁用了文档管理器 documentation manager;因为这里的文档集很小,而且只有一个过滤器部分,所以也关闭了地址栏 address bar 和过滤器功能 filter functionality。

虽然第三步已经生成了. qch 文件并且在 Qt Assistant 中进行了注册,但那只是为了测试文件是否可用,其实完全可以跳过第三步,因为这里 docFiles 标签中就完成了第三步的操作。不过与第三步不同的是,第三步是在默认的集合文件中注册的,而这里是在我们自己的集合文件中注册的。

第五步,生成. qhc 文件。在命令行输入如下命令:

```
qcollectiongenerator myHelp.qhcp -o myHelp.qhc
```

为了测试我们定制的 Qt Assistant,可以在输入如下命令:

```
assistant -collectionFile myHelp.qhc
```

这里在运行 Qt Assistant 时指定了集合文件为我们自己的. qhc 文件,所以运行后只会显示我们自己的 HTML 文档。可以看到,现在 Qt Assistant 的图标也更改了,打开“Help→关于该帮助”菜单项,这里是前面添加的 about.txt 文件的内容。

第六步,在程序中启动 Qt Assistant。这里先要将 Qt 安装目录的 bin 目录中的 assistant.exe 程序复制到我们项目目录的 documentation 目录中。然后在上一节的程序中进行更改。为了启动 Qt Assistant,先要创建了一个 Assistant 类。首先向项目中添加新文件,模板选择 C++ 类,类名为 Assistant,基类不填写,类型信息选择无。然后将 assistant.h 文件更改如下:

```
#ifndef ASSISTANT_H
#define ASSISTANT_H
#include <QtCore/QString>
class QProcess;

class Assistant
{
public:
    Assistant();
    ~Assistant();
    void showDocumentation(const QString &file);
private:
    bool startAssistant();
    QProcess * proc;
};


```

```
#endif //ASSISTANT_H
```

在 Assistant 类中主要是使用 QProcess 类创建一个进程来启动,下面是 assistant.cpp 文件的内容:

```
#include <QtCore/QByteArray>
#include <QtCore/QProcess>
#include <QtGui/QMessageBox>
#include "assistant.h"

Assistant::Assistant()
: proc(0)
{
}

Assistant::~Assistant()
{
    if (proc && proc->state() == QProcess::Running) {
        //试图终止进程
        proc->terminate();
        proc->waitForFinished(3000);
    }
    //销毁 proc
    delete proc;
}

//显示文档
void Assistant::showDocumentation(const QString &page)
{
    if (!startAssistant())
        return;
    QByteArray ba("SetSource ");
    ba.append("qthelp://yafeilinux.myHelp/doc/");
    proc->write(ba + page.toLocal8Bit() + '\n');
}

//启动 Qt Assistant
bool Assistant::startAssistant()
{
    //如果没有创建进程,则新创建一个
    if (!proc)
        proc = new QProcess();
    //如果进程没有运行,则运行 assistant,并添加参数
}
```

```

if (proc->state() != QProcess::Running) {
    QString app = QLatin1String("../myWhatsThis/documentation/assistant.exe");
    QStringList args;
    args << QLatin1String(" - collectionFile")
        << QLatin1String("../myWhatsThis/documentation/myHelp.qhc");
    proc->start(app, args);
    if (!proc->waitForStarted()) {
        QMessageBox::critical(0, QObject::tr("my help"),
            QObject::tr("Unable to launch Qt Assistant (%1)").arg(app));
        return false;
    }
}
return true;
}

```

在 startAssistant() 函数中使用 QProcess 创建了一个进程来启动 Qt Assistant，这里使用了命令行参数来使用帮助集合文件，对于 assistant.exe 和 myHelp.qhc 都使用了相对地址；在 showDocumentation() 函数中可以指定具体的页面作为参数来使 Qt Assistant 显示指定的页面；在析构函数中，如果进程还在运行，则终止进程，最后销毁了进程指针。

下面来使用 Assistant 类来启动 Qt Assistant。在 mainwindow.h 文件中先添加前置声明：

```
class Assistant;
```

再添加 private 对象指针声明：

```
Assistant *assistant;
```

然后添加一个私有槽：

```
private slots:
```

```
void startAssistant();
```

现在到 mainwindow.cpp 文件中进行更改。添加头文件包含 #include "assistant.h"，然后在构造函数中添加如下代码：

```

QAction *help = new QAction("help", this);
ui->mainToolBar->addAction(help);
connect(help, SIGNAL(triggered()), this, SLOT(startAssistant()));

//创建 Assistant 对象
assistant = new Assistant;

```

这里创建了一个 help 动作，并将它添加到了工具栏中，可以使用该动作启动 Qt Assistant。下面是 startAssistant() 槽的定义：

```

void MainWindow::startAssistant()
{
    //单击 help 按钮,运行 Qt Assistant,显示 index.html 页面
    assistant->showDocumentation("index.html");
}

```

最后在析构函数中销毁 `assistant` 指针,即在 `MainWindow::~MainWindow()` 函数中添加如下代码:

```

//销毁 assistant
delete assistant;

```

现在运行程序,单击工具栏上的 help 动作就可以启动 Qt Assistant 了。这里还要提示一下,如果要发布该程序,那么需要将 documentation 目录复制到发布目录中,这时运行程序,还会提示缺少一些 dll 文件,那么就可以根据提示在 Qt 安装目录的 bin 目录中将相应的 dll 文件复制过来。

对应这个例子,可以在帮助中查看 Simple Text Viewer Example 关键字,也可以在 Qt 安装目录 examples 下的 help 目录中找到。关于 Qt Assistant 的定制,可以查看 Using Qt Assistant as a Custom Help Viewer 关键字。在 The Qt Help Framework 关键字对应的文档中还讲解了使用 QHelpEngine 的 API 将帮助内容直接嵌入到应用程序中。

9.3 创建 Qt 插件

Qt 插件(Qt Plugin)就是一个共享库(dll 文件),可以使用它来进行功能的扩展。Qt 中提供了两种 API 来创建插件:

- 用来扩展 Qt 本身的高级 API,如自定义数据库驱动、图片格式、文本编码和自定义风格等;
- 用来扩展 Qt 应用程序的低级 API。

如果要写一个插件来扩展 Qt 本身,那么可以子类化合适的插件基类,然后重写一些函数并添加一个宏。可以通过在帮助中查看 How to Create Qt Plugins 关键字来了解本节的内容,在这里还可以查看 Qt 提供的插件基类。Qt 中提供了一个风格插件的例子,用来扩展 Qt 风格,可以在帮助中查看 Style Plugin Example 关键字。这一节主要讲解一个创建 Qt 应用程序的插件的例子,以及创建 Qt Designer 自定义部件的方法。

9.3.1 在设计模式提升窗口部件

讲解创建插件以前先介绍如何在设计模式提升窗口部件。一般的,使用代码生成的部件无法直接在设计器中使用,但是可以通过建立成设计器插件来实现,不过,更简单的方法是使用提升窗口部件的做法,这样可以将设计器中的部件指定为自定义类的

实例。下面来看一个具体例子。

(项目源码路径: src\09\9-3\myButton)新建 Qt Gui 应用,项目名称为 myButton,类名为 MainWindow,基类保持 QMainWindow 不变。建立好项目后向其中添加新文件,模板选择 C++ 类,类名为 MyButton,基类为 QPushButton,类型信息选择“继承自 QWidget”。添加文件完成后进入 mybutton.h 文件,添加一个 public 函数定义:

```
QString getName(){return "My Button!"}
```

然后打开 mainwindow.ui 文件,在设计模式中向界面上放入一个 Push Button,并在 pushButton 上右击,在弹出的菜单中选择“提升的窗口部件”。在弹出的对话框中将提升的类名称改为 MyButton,头文件会自动生成为“mybutton.h”,这时单击右边的添加按钮就会在提升的类列表中进行显示,单击“提升”按钮退出对话框。

现在界面上的 PushButton 部件已经是 MyButton 类的实例了,可以调用它调用 MyButton 的 getName() 函数。在 mainwindow.cpp 文件的构造函数中添加如下代码:

```
QString str = ui ->pushButton ->getName();
ui ->pushButton ->setText(str);
```

通过这种方式就可以在设计模式使用代码生成的自定义类。

9.3.2 创建应用程序插件

创建插件要先创建接口,接口就是一个类,只包含纯虚函数。插件类要继承自该接口。插件类存储在一个共享库中,因此可以在应用程序运行时加载。创建一个插件包括以下几步:

- ① 定义一个插件类,它需要同时继承自 QObject 类和该插件所提供的功能对应的接口类;
- ② 使用 Q_INTERFACES() 宏在 Qt 的元对象系统中注册该接口;
- ③ 使用 Q_EXPORT_PLUGIN2() 宏导出该插件;
- ④ 使用合适的.pro 文件构建该插件。

使一个应用程序可以通过插件进行扩展要进行以下几步:

- ① 定义一组接口(只有纯虚函数的抽象类);
- ② 使用 Q_DECLARE_INTERFACE() 宏在 Qt 的元对象系统中注册该接口;
- ③ 在应用程序中使用 QPluginLoader 来加载插件;
- ④ 使用 qobject_cast() 来测试插件是否实现了给定的接口。

下面通过创建一个过滤字符串中出现的第一个数字的插件来讲解应用程序的插件创建过程。这里需要创建两个项目,一个项目用来生成插件即 dll 文件;另一个项目是一个测试程序,用来使用插件。因为这两个项目中有共用的文件,所以这里将它们放到一个目录中。(项目源码路径: src\09\9-4\myPlugin。)

1. 创建插件

第一步,创建插件类。新建空的 Qt 项目,项目名称为 plugin,在选择路径时指定到

一个新建的 myPlugin 目录中。建立好项目后向其中添加一个 C++ 类,类名为 RegExpPlugin,基类保持为空,类型信息选择“无”。

第二步,定义插件类。将 regexpplugin.h 文件中的内容更改如下:

```
#ifndef REGEXPPLUGIN_H
#define REGEXPPLUGIN_H

#include <QObject>
#include "regexpinterface.h"

class RegExpPlugin : public QObject, RegExpInterface
{
    Q_OBJECT
    Q_INTERFACES(RegExpInterface)

public:
    QString regexp(const QString &message);
};

#endif
```

为了使这个类作为一个插件,它需要同时继承自 QObject 和 RegExpInterface。RegExpInterface 是接口类,用来指明插件要实现的功能,这个类在 regexpinterface.h 文件中(这个文件在后面的测试程序项目中)定义。这里还需要使用 Q_INTERFACES() 宏将这个接口注册到 Qt 的元对象系统中,告知 Qt 这个类实现了哪个接口。最后还声明了一个 regexp() 函数,它是在 RegExpInterface 中定义的一个纯虚函数。这里通过重写它来实现该插件具体的功能,就是将字符串中的第一个数字提取出来并返回。

第三步,导出插件。将 regexpplugin.cpp 文件中的内容更改如下:

```
#include "regexpplugin.h"
#include <QRegExp>
#include <QtPlugin>

QString RegExpPlugin::regexp(const QString &message)
{
    QRegExp rx("\\d+");
    rx.indexIn(message);
    QString str = rx.cap(0);
    return str;
}

Q_EXPORT_PLUGIN2(regexpplugin, RegExpPlugin);
```

这里最后使用了 Q_EXPORT_PLUGIN2() 导出了该插件,其中 regexpplugin 为

插件名, RegExpPlugin 为插件类名。

第四步,更改项目文件。打开 plugin.pro 文件,将其内容更改如下:

```
TEMPLATE          = lib
CONFIG           += plugin
INCLUDEPATH      += ../regexpwindow
HEADERS          = regexpplugin.h
SOURCES          = regexpplugin.cpp
TARGET            = regexpplugin
DESTDIR          = ../plugins
```

这里使用了 TEMPLATE=lib 表明该项目要构建库文件,而不是像以前那样的可执行文件;使用 CONFIG+=plugin 告知 qmake 要创建一个插件;因为项目中使用了 regexpwindow 目录中的 regexpinterface.h 文件,所以这里将该目录的路径添加到了 INCLUDEPATH 中;TARGET 指定了生产的 dll 文件的名字,它需要和上一步中的 Q_EXPORT_PLUGIN2() 指定的名字相同;最后使用 DESTDIR 指定了生成的 dll 文件所在的目录。

因为这个项目中使用了 regexpinterface.h 文件,而这个文件在另一个项目中,所以现在还无法构建该项目。

2. 使用插件扩展应用程序

第一步,新建 Qt Gui 应用。项目名称为 regexpwindow,选择路径时仍选择前面建立的 myPlugin 目录。基类选择 QWidget,类名保持 Widget 不变。建立完成后,向该项目中添加新文件,模板选择 C++ 头文件,名称为“regexpinterface.h”。

第二步,定义接口。将 regexpinterface.h 文件的内容更改如下:

```
#ifndef REGEXPINTERFACE_H
#define REGEXPINTERFACE_H

#include <QString>

class RegExpInterface
{
public:
    virtual ~RegExpInterface() {}
    virtual QString regexp(const QString &message) = 0;
};

Q_DECLARE_INTERFACE(RegExpInterface,
                    "yafeilinux.RegExpInterface/1.0");
#endif
```

接口类中定义了插件要实现的函数,比如这里定义了 regexp() 函数,可以看到在

前面的 RegExpPlugin 类中已经实现了该函数。在这个类中只能包含纯虚函数。最后使用 Q_DECLARE_INTERFACE() 宏在 Qt 元对象系统中注册了该接口，其中第二个参数是一个字符串，用来确保这个接口与其他接口不会相同。

第三步，加载插件。单击 widget.ui 文件进入设计模式，设计的界面如图 9-5 所示。将其中显示“无”字的 Label 的 objectName 属性更改为 labelNum。然后进入 widget.h 文件，先添加头文件 #include "regexpinterface.h"，然后在 private 部分定义一个接口对象指针，再声明一个加载插件函数：

```
RegExpInterface * regexpInterface;
bool loadPlugin();
```

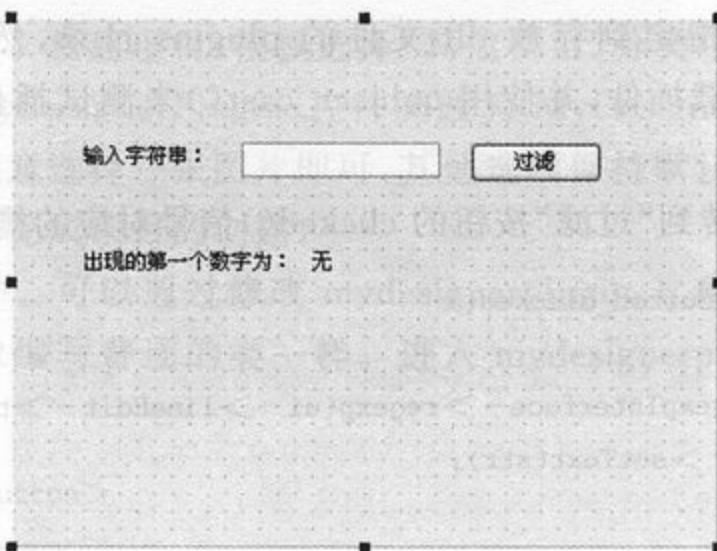


图 9-5 设计界面

现在到 widget.cpp 文件中先添加头文件包含：

```
#include <QPluginLoader>
#include <QMessageBox>
#include <QDir>
```

然后在构造函数中调用加载插件函数，如果加载失败则进行警告：

```
if (!loadPlugin()) { //如果无法加载插件
    QMessageBox::information(this, "Error", "Could not load the plugin");
    ui->lineEdit->setEnabled(false);
    ui->pushButton->setEnabled(false);
}
```

下面是加载插件函数的定义：

```
bool Widget::loadPlugin()
{
    QDir pluginsDir("../plugins");
    //遍历插件目录
```

```

foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
    QPluginLoader pluginLoader(pluginsDir.absoluteFilePath(fileName));
    QObject * plugin = pluginLoader.instance();
    if (plugin) {
        regexpInterface = qobject_cast<RegexpInterface *>(plugin);
        if (regexpInterface)
            return true;
    }
}
return false;
}

```

这里使用 QDir 类指定到存放 dll 文件的 plugins 目录,然后遍历该目录,使用 QPluginLoader 类来加载插件,并使用 qobject_cast() 来测试插件是否实现了 RegexpInterface 接口。

最后到设计模式,转到“过滤”按钮的 clicked() 信号对应的槽,更改如下:

```

void Widget::on_pushButton_clicked()
{
    QString str = regexpInterface ->regexp(ui ->lineEdit ->text());
    ui ->labelNum ->setText(str);
}

```

这里就是使用了 regexpInterface 接口的 regexp() 函数来获取lineEdit 部件中输入的字符串中第一个出现的数字,然后在 labelNum 中显示出来。

第四步,运行程序。先构建 plugin 项目,在编辑模式左侧项目列表中的 plugin 目录上右击,选择“构建项目‘plugin’”。当构建完成后,在 myPlugin 目录中会生成 plugins 目录,里面包含了生成的 dll 文件。然后运行 regexpwindow 项目,输入一些字符串,查看运行结果,如图 9-6 所示。

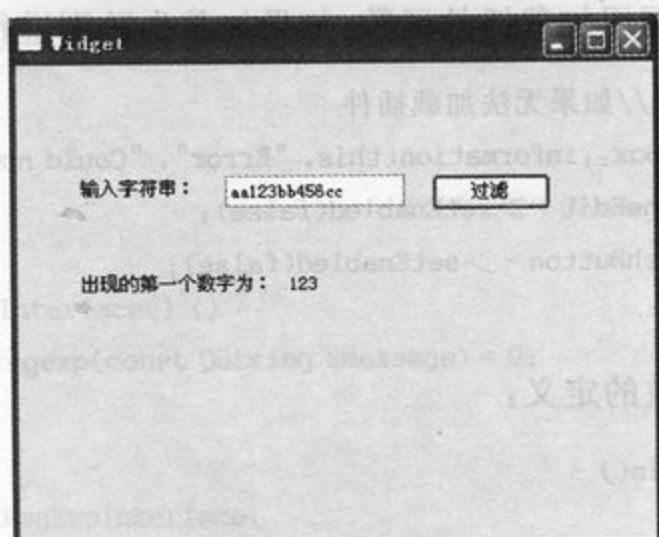


图 9-6 使用插件运行效果

到这里，创建插件并在应用程序中使用插件的整个过程就介绍完了。这个例子是基于 Qt 的 Echo Plugin Example 示例程序的，可以参考。Qt 中还有一个 Plug and Paint 演示程序，在 Tools 分类中，这是一个比较综合的使用插件扩展应用程序的例子。

9.3.3 创建 Qt Designer 自定义部件

Qt Designer 的基于插件的架构使得它可以使用用户设计的或者第三方提供的自定义部件，就像使用标准的 Qt 部件一样。自定义部件中的所有特性在 Qt Designer 中都是可用的，这包含了部件属性、信号和槽等。下面通过例子来看一下在 Qt Creator 中创建 Qt Designer 自定义部件的过程。（项目源码路径：src\09\9-5\myDesignerPlugin。）

第一步，创建项目。新建项目，模板选择“其他项目”分类中的“Qt4 设计师自定义控件”；项目名称为 myDesignerPlugin；控件类改为 MyDesignerPlugin，然后在右侧指定图标文件的路径，随意选择一张图片即可，其他选项保持默认；后面步骤全部保持默认，单击“下一步”直到完成项目的创建。

第二步，更改部件。可以通过修改 mydesignerplugin.h 和 mydesignerplugin.cpp 文件来修改部件，就像编写普通的类一样。进入 mydesignerplugin.cpp 文件，添加头文件：

```
#include <QPushButton>
#include <QHBoxLayout>
```

然后在构造函数中添加如下代码：

```
QPushButton *button1 = new QPushButton(this);
QPushButton *button2 = new QPushButton(this);
button1 ->setText("hello");
button2 ->setText("Qt!");
QHBoxLayout *layout = new QHBoxLayout;
layout ->addWidget(button1);
layout ->addWidget(button2);
setLayout(layout);
```

这里需要在 Qt Creator 左下角的目标选择器中将构建选项选择为 Release，因为只有 Release 版本的插件在 Qt Designer 中才可以使用。按下 Ctrl+Shift+B 构建项目，完成后可以看到在项目目录 build-desktop 目录中的 release 目录中已经生成了相应的 dll 文件。

第三步，在 Qt Designer 中使用插件。为了使 Qt Designer 可以自动检测到插件，需要将生成的 dll 文件复制到 Qt 安装目录的 plugins 目录下的 designer 目录中，笔者这里的路径为 C:\Qt\4.7.2\plugins\designer。这时在开始菜单中运行 Qt Designer，发现已经可以使用自定义的部件了，效果如图 9-7 所示。

这里需要说明一下，现在生成的插件只能在 Qt Designer 中使用，却不能在 Qt

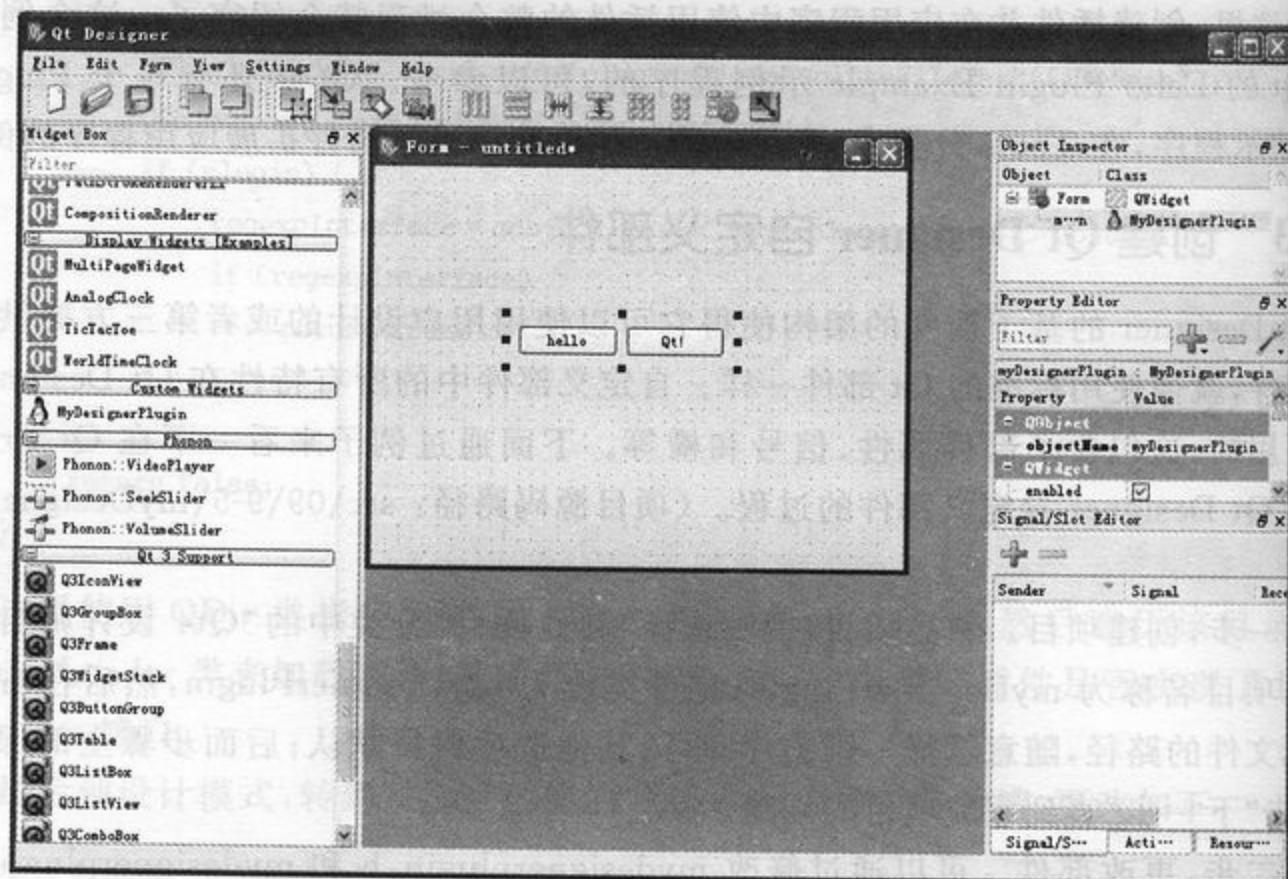


图 9-7 在 Qt Designer 中使用自定义部件

Creator 中的设计模式中使用, Adding Qt Designer Plugins 关键字对应的文档中提到了这一点。这是因为现在使用的 Windows 版本的 Qt Creator 是使用 Microsoft Visual Studio 的编译器生成的, 而 Qt Creator 中编译项目使用的是 MinGW/g++ 编译器, 它们的 Build Keys(包含了体系结构、操作系统, 编译器等信息, 具体可以参见 Deploying Plugins 关键字)不同, 所以生成的插件无法在 Qt Creator 中使用。

关于创建 Qt Designer 自定义部件的更多的内容, 可以在帮助中查看 Creating Custom Widgets for Qt Designer 关键字。Qt 还提供了几个创建 Qt Designer 插件的例子, 比如 Custom Widget Plugin, 它们在 Qt Designer 分类中。

9.4 小结

这一章介绍了 Qt 中 3 个比较重要的内容, 分别是 Qt 的国际化、自定义 Qt Assistant 和自定义 Qt 插件。这 3 部分内容看似复杂, 其实很简单, 因为这里并没有涉及太多的技术问题, 而只是一些流程, 以后使用时按照步骤进行操作就可以了。

这一章是第一篇的最后一章, 也就是说, 到这里 Qt 基本应用部分就讲完了。从下一章开始, 我们将接触一些比较专业的知识, 它们分别代表了一个应用方面, 需要掌握更加具体的专业信息。

(项目源码路径: `src\第10章\2DDrawing`)新建 Qt GUI 应用, 项目名为 myDrawing, 基类选择 QWidget, 类名为 MyWidget。建立完成后, 在 `widget.cpp` 文件中重写事件处理函数:

```
protected:
    void paintEvent(QPaintEvent *event)
```

第 10 章

图形动画篇

► 第 10 章 2D 绘图

► 第 11 章 图形视图、动画和状态机框架

► 第 12 章 3D 绘图

不再吓唬钱本基 1.01

这两种方式都可以完成绘制, 无论使用哪种方式, 都要指定设备对象, 否则就无法进行绘制。第二行代码使用 `drawLine()` 函数绘制了一条线段, 这里使用了该函数的一个参数表示线段的起始点是水平坐标, 其基础类会自动将垂直坐标转换为 `QPainter` 中的纵坐标。从这个例子可以看出, `QPainter` 的绘图功能非常强大, 它可以绘制出任何你想要的图形。本文将主要以 QPainter 为例, 通过前面对讲解其操作过程和除了绘制简单的线条以外, `QPainter` 还能绘制各种复杂的图形。关于 `QPainter` 的更多知识, 请参阅 `Qt API 参考手册` 中的 `QPainter` 一章。

第 10 章

2D 绘图

Qt 中提供了强大的 2D 绘图系统, 可以使用相同的 API 在屏幕和绘图设备上进行绘制, 主要基于 QPainter、QPaintDevice 和 QPaintEngine 这 3 个类。其中, QPainter 用来执行绘图操作; QPaintDevice 提供绘图设备, 是一个二维空间的抽象, 可以使用 QPainter 在其上进行绘制; QPaintEngine 提供了一些接口, 可以用于 QPainter 在不同的设备上进行绘制。它们三者的关系如图 10-1 所示。

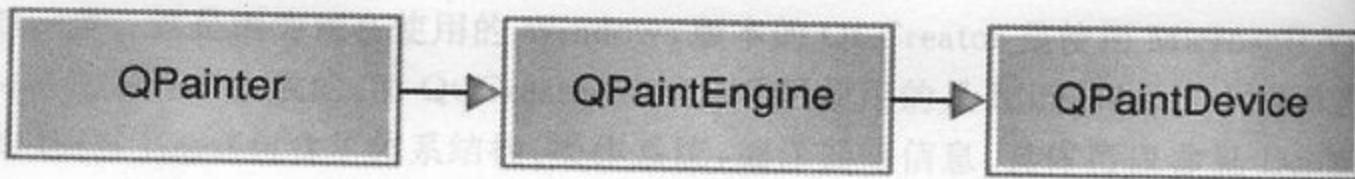


图 10-1 QPainter、QPaintEngine 和 QPaintDevice 关系图

这一章中我们将讲解与 Qt 2D 绘图相关的一些知识, 包括基本的绘制和填充、Qt 坐标系统等。对应本章的内容, 可以在帮助中查看 Paint System 关键字。

10.1 基本绘制和填充

绘图系统中由 QPainter 来完成具体的绘制操作, 提供了大量高度优化的函数来完成 GUI 编程所需要的大部分绘制工作。QPainter 可以绘制一切想要的图形, 从最简单的一条直线到其他任何复杂的图形, 还可以用来绘制文本和图片。QPainter 可以在继承自 QPaintDevice 类的任何对象上进行绘制操作。

QPainter 一般在一个部件重绘事件(Paint Event)的处理函数 paintEvent() 中绘制, 首先要创建 QPainter 对象, 再进行图形的绘制, 最后销毁 QPainter 对象。

10.1.1 基本图形的绘制和填充

QPainter 中提供了一些方便的函数来绘制常用的图形, 而且还可以设置线条、边框的画笔以及进行填充的画刷。

(项目源码路径: src\10\10-1\myDrawing) 新建 Qt Gui 应用, 项目名称为 myDrawing, 基类选择 QWidget, 类名为 Widget。建立完成后, 在 widget.h 文件中声明重绘事件处理函数:

```
protected:  
    void paintEvent(QPaintEvent * event);
```

然后到 widget.cpp 文件中添加头文件 #include <QPainter>。

1. 绘制图形

先在 widget.cpp 文件中对 paintEvent() 函数进行如下定义:

```
void Widget::paintEvent(QPaintEvent * event)  
{  
    QPainter painter(this);  
    painter.drawLine(QPoint(0, 0), QPoint(100, 100));  
}
```

这里先创建了一个 QPainter 对象, 使用了 QPainter::QPainter (QPaintDevice * device) 构造函数, 并指定了 this 为绘图设备, 即表明在该部件上进行绘制。使用这个构造函数创建的对象会立即开始在设备上绘制, 自动调用 begin() 函数, 然后在 QPainter 的析构函数中调用 end() 函数结束绘制。如果在构建 QPainter 对象时不想指定绘制设备, 那么可以使用不带参数的构造函数, 然后使用 QPainter::begin (QPaintDevice * device) 在开始绘制时指定绘制设备, 等绘制完成后再调用 end() 函数结束绘制。上面函数中的代码等价于:

```
QPainter painter;  
painter.begin(this);  
painter.drawLine(QPoint(0, 0), QPoint(100, 100));  
painter.end();
```

这两种方式都可以完成绘制, 无论使用哪种方式, 都要指定绘图设备, 否则将无法进行绘制。第二行代码使用 drawLine() 函数绘制了一条线段, 这里使用了该函数的一种重载形式 QPainter::drawLine (const QPoint & p1, const QPoint & p2), 其中 p1 和 p2 分别是线段的起点和终点。这里的 QPoint(0, 0) 就是窗口的原点, 默认是窗口的左上角(不包含标题栏)。

除了绘制简单的线条以外, QPainter 中还提供了一些绘制其他常用图形的函数, 其中最常用的几个如表 10-1 所列。

表 10-1 QPainter 中常用图形绘制函数介绍

函 数	功 能	函 数	功 能
drawArc()	绘制圆弧	drawPoint()	绘制点
drawChord()	绘制弦	drawPolygon()	绘制多边形
drawConvexPolygon()	绘制凸多边形	drawPolyline()	绘制折线
drawEllipse()	绘制椭圆	drawRect()	绘制矩形
drawLine()	绘制线条	drawRoundedRect()	绘制圆角矩形
drawPie()	绘制扇形		

2. 使用画笔

在 paintEvent() 函数中继续添加如下代码：

```
// 创建画笔
QPen pen(Qt::green, 5, Qt::DotLine, Qt::RoundCap, Qt::RoundJoin);
// 使用画笔
painter.setPen(pen);
QRectF rectangle(70.0, 40.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;
// 绘制圆弧
painter.drawArc(rectangle, startAngle, spanAngle);
```

QPen 类为 QPainter 提供了画笔来绘制线条和形状的轮廓, 这里使用的构造函数为 QPen::QPen (const QBrush & brush, qreal width, Qt::PenStyle style=Qt::SolidLine, Qt::PenCapStyle cap=Qt::SquareCap, Qt::PenJoinStyle join=Qt::BevelJoin), 几个参数依次为画笔使用的画刷、线宽、画笔风格、画笔端点风格和画笔连接风格, 也可以分别使用 setBrush()、setWidth()、setStyle()、setCapStyle() 和 setJoinStyle() 等函数来设置。其中画刷可以为画笔提供颜色; 线宽的默认值为 0(宽度为 1 个像素); 画笔风格有实线、点线等, Qt 中提供的画笔风格及其效果如图 10-2 所示, 它还有一个 Qt::NoPen 值, 表示不进行线条或边框的绘制。还可以使用 setDashPattern() 函数来自定义一个画笔风格。

画笔端点风格定义了怎样进行线条端点的绘制, 这些风格对宽度为 0 的线条没有作用。

最后的画笔连接风格定义了怎样绘制两个线条的连接, 同样, 风格对于宽度为 0 的线条没有作用, 可以把很宽的线条看作一个矩形来理解这些风格。Qt 中提供了 Path Stroking 演示程序, 可以显示画笔属性的各种组合效果, 可以在帮助中查询该关键字。

创建完画笔后, 使用了 setPen() 来为 painter 设置画笔, 然后使用画笔绘制了一个圆弧。绘制圆弧函数的一种重载形式为 QPainter::drawArc (const QRectF & rectan-

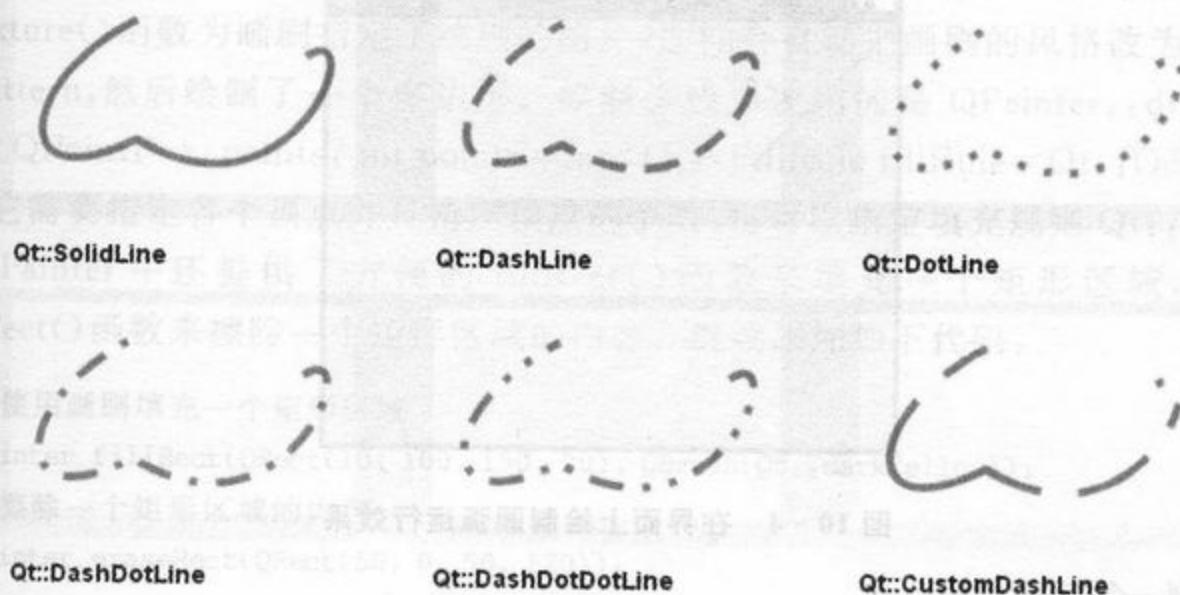


图 10-2 画笔风格

gle, int startAngle, int spanAngle),这里的3个参数分别对应于需要指定弧线所在的矩形、起始角度和跨越角度,如图10-3所示。QRectF::QRectF(qreal x, qreal y, qreal width, qreal height)可以使用浮点数为参数来确定一个矩形,它需要指定左上角的坐标(x,y)、宽width和高height。如果只想使用整数来确定一个矩形,那么可以使用QRect类。这里角度的数值为实际度数乘以16,在时钟表盘中,0度指向3时的位置,角度数值为正则表示逆时针旋转,角度数值为负则表示顺时针旋转,整个一圈的数值为5760(即 360×16)。现在运行程序,效果如图10-4所示。

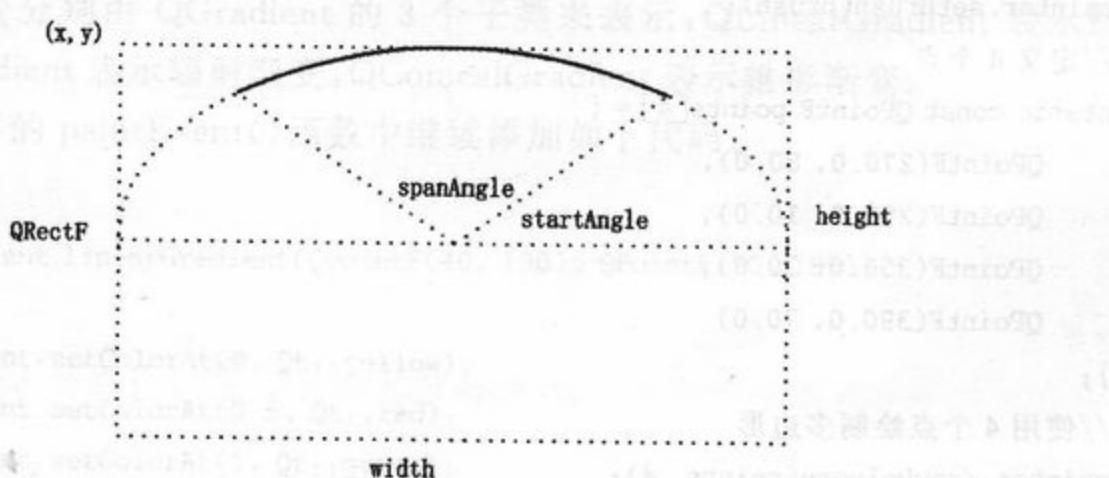


图 10-3 绘制圆弧示意图

3. 使用画刷

在paintEvent()函数中继续添加如下代码:

```
//重新设置画笔
pen.setWidth(1);
pen.setStyle(Qt::SolidLine);
painter.setPen(pen);
```

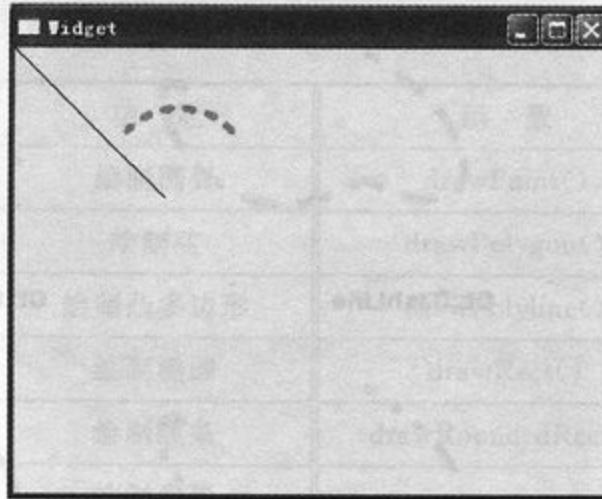


图 10-4 在界面上绘制圆弧运行效果

```

//绘制一个矩形
painter.drawRect(160, 20, 50, 40);
//创建画刷
QBrush brush(QColor(0, 0, 255), Qt::Dense4Pattern);
//使用画刷
painter.setBrush(brush);
//绘制椭圆
painter.drawEllipse(220, 20, 50, 50);
//设置纹理
brush.setTexture(QPixmap("../myDrawing/yafeilinux.png"));
//重新使用画刷
painter.setBrush(brush);
//定义 4 个点
static const QPointF points[4] = {
    QPointF(270.0, 80.0),
    QPointF(290.0, 10.0),
    QPointF(350.0, 30.0),
    QPointF(390.0, 70.0)
};
//使用 4 个点绘制多边形
painter.drawPolygon(points, 4);

```

QBrush 类提供了画刷来填充图形,一个画刷使用它的颜色和风格(例如它的填充模式)来定义。Qt 中使用的颜色一般都由 QColor 类来表示,支持 RGB、HSV 和 CMYK 等颜色模型。QColor 还支持基于 alpha 的轮廓和填充(实现透明效果),而且 QColor 类与平台和设备无关(颜色使用 QColormap 类向硬件进行映射)。Qt 中还提供了 20 种预定义的颜色,比如以前经常使用的 Qt::red 等,可以查看 Qt::GlobalColor 关键字来了解。填充模式使用 Qt::BrushStyle 枚举变量来定义,包含了基本模式填充、渐变填充和纹理填充。

上面程序先绘制了一个矩形,这里没有指定画刷,那么将不会对矩形的内部进行填

充;然后使用 `Qt::Dense4Pattern` 风格定义了一个画刷并绘制了一个椭圆;最后使用 `setTexture()` 函数为画刷指定了纹理的图片,这样会自动把画刷的风格改为 `Qt::TexturePattern`,然后绘制了一个多边形。绘制多边形使用的是 `QPainter::drawPolygon(const QPointF * points, int pointCount, Qt::FillRule fillRule=Qt::OddEvenFill)` 函数,它需要指定各个顶点并且指定顶点的个数,也可以指定填充规则 `Qt::FillRule`。

`QPainter` 中还提供了方便的 `fillRect()` 函数来填充一个矩形区域,也提供了 `eraseRect()` 函数来擦除一个矩形区域的内容。继续添加如下代码:

```
//使用画刷填充一个矩形区域
painter.fillRect(QRect(10, 100, 150, 20), QBrush(Qt::darkYellow));
//擦除一个矩形区域的内容
painter.eraseRect(QRect(50, 0, 50, 120));
```

关于绘制和填充,可以查看 `Drawing and Filling` 关键字,`Qt` 中还提供了一个 `Basic Drawing` 的例子来演示画笔和画刷使用的方法,它在 `Painting` 分类中。

10.1.2 渐变填充

在前一节提到了在画刷中可以使用渐变填充。`QGradient` 类就是用来和 `QBrush` 一起指定渐变填充的。`Qt` 现在支持 3 种类型的渐变填充:

- 线性渐变(`linear gradient`)在开始点和结束点之间插入颜色;
- 辐射渐变(`radial gradient`)在焦点和环绕它的圆环间插入颜色;
- 锥形渐变(`Conical`)在圆心周围插入颜色。

这 3 种渐变分别由 `QGradient` 的 3 个子类来表示,`QLinearGradient` 表示线性渐变,`QRadialGradient` 表示辐射渐变,`QConicalGradient` 表示锥形渐变。

在前面程序的 `paintEvent()` 函数中继续添加如下代码:

```
//线性渐变
QLinearGradient linearGradient(QPointF(40, 190), QPointF(70, 190));
//插入颜色
linearGradient.setColorAt(0, Qt::yellow);
linearGradient.setColorAt(0.5, Qt::red);
linearGradient.setColorAt(1, Qt::green);
//指定渐变区域以外的区域的扩散方式
linearGradient.setSpread(QGradient::RepeatSpread);
//使用渐变作为画刷
painter.setBrush(linearGradient);
painter.drawRect(10, 170, 90, 40);

//辐射渐变
QRadialGradient radialGradient(QPointF(200, 190), 50, QPointF(275, 200));
radialGradient.setColorAt(0, QColor(255, 255, 100, 150));
radialGradient.setColorAt(1, QColor(0, 0, 0, 50));
```

```

painter.setBrush(radialGradient);
painter.drawEllipse(QPointF(200, 190), 50, 50);

//锥形渐变
QConicalGradient conicalGradient(QPointF(350, 190), 60);
conicalGradient.setColorAt(0.2, Qt::cyan);
conicalGradient.setColorAt(0.9, Qt::black);
painter.setBrush(conicalGradient);
painter.drawEllipse(QPointF(350, 190), 50, 50);

//画笔使用线性渐变来绘制直线和文字
painter.setPen(QPen(linearGradient, 2));
painter.drawLine(0, 280, 100, 280);
painter.drawText(150, 280, tr("helloQt!"));

```

1. 线性渐变

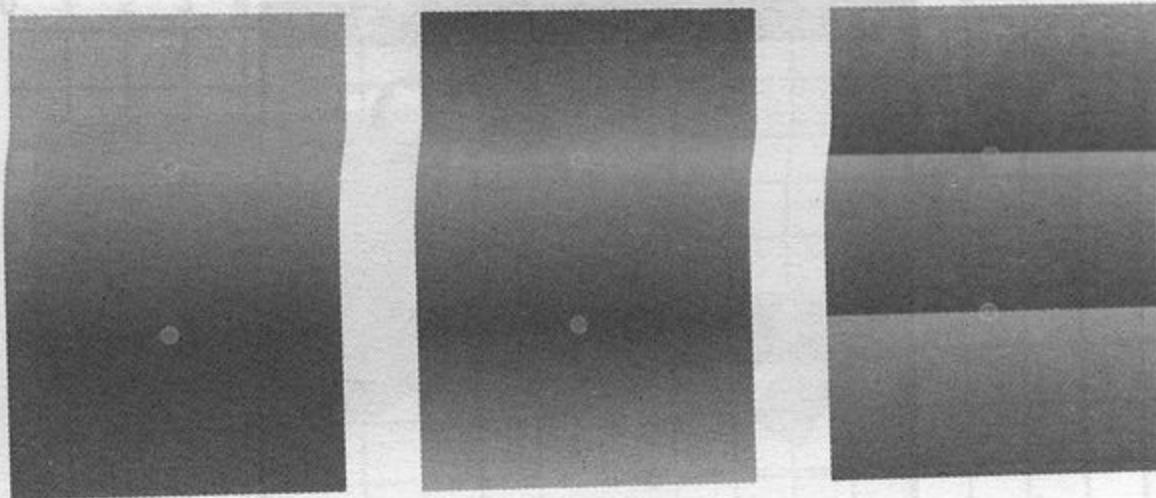
线性渐变 `QLinearGradient::QLinearGradient (const QPointF & start, const QPointF & finalStop)` 需要指定开始点 `start` 和结束点 `finalStop`, 然后将开始点和结束点之间的区域进行等分, 开始点的位置为 0.0, 结束点的位置为 1.0, 它们之间的位置按照距离比例进行设定, 然后使用 `QGradient::setColorAt (qreal position, const QColor & color)` 函数在指定的位置 `position` 插入指定的颜色 `color`, 当然, 这里的 `position` 的值要在 0~1 之间。

这里还可以使用 `setSpread()` 函数来设置填充的扩散方式, 即指明在指定区域以外的区域怎样进行填充。扩散方式由 `QGradient::Spread` 枚举变量定义, 它一共有 3 个值, 分别是 `QGradient::PadSpread`, 使用最接近的颜色进行填充, 这是默认值, 如果不使用 `setSpread()` 指定扩散方式, 那么就会默认使用这种方式; `QGradient::RepeatSpread` 在渐变区域以外的区域重复渐变; `QGradient::ReflectSpread` 在渐变区域以外将反射渐变。在线性渐变中这 3 种扩散方式的效果如图 10-5 所示。要使用渐变填充, 可以直接在 `setBrush()` 中使用, 这时画刷风格会自动设置为相对应的渐变填充。

2. 辐射渐变

辐射渐变 `QRadialGradient::QRadialGradient (const QPointF & center, qreal radius, const QPointF & focalPoint)` 需要指定圆心 `center` 和半径 `radius`, 这样就确定了一个圆, 然后再指定一个焦点 `focalPoint`。焦点的位置为 0, 圆环的位置为 1, 然后在焦点和圆环间插入颜色。辐射渐变也可以使用 `setSpread()` 函数设置渐变区域以外的区域的扩散方式, 3 种扩散方式的效果如图 10-6 所示。

程序中设置颜色时使用了 `QColor::QColor (int r, int g, int b, int a=255)`, 其中参数 `r,g,b` 为三基色, 分别是红(red)、绿(green)和蓝(blue)。它们的取值都在 0~255 之间, 例如 `QColor(255, 0, 0)` 表示红色, `QColor(255, 255, 0)` 表示黄色, `QColor(255,`

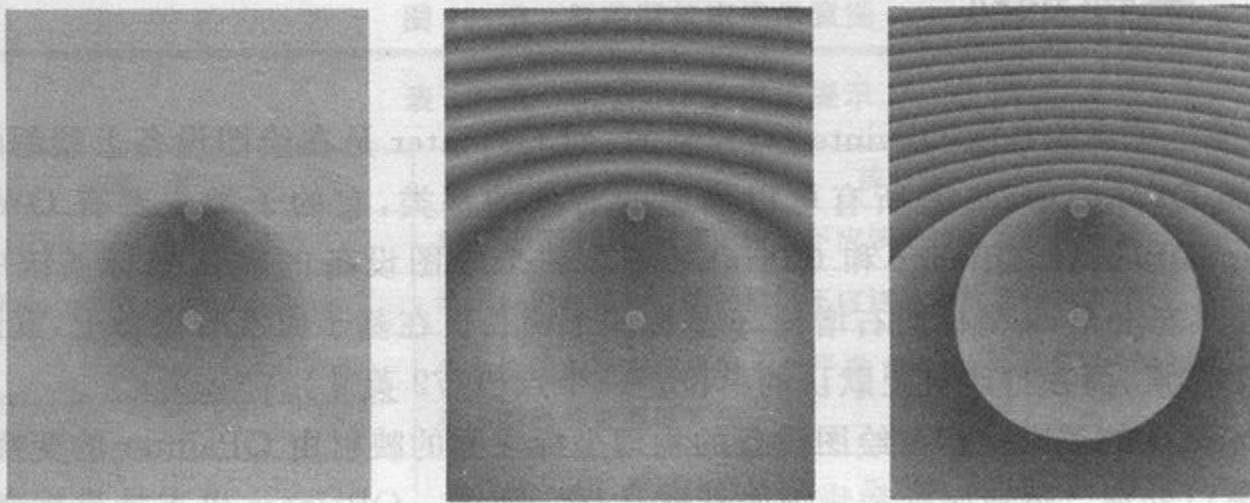


(a) PadSpread(default)

(b) ReflectSpread

(c) RepeatSpread

图 10-5 线性渐变的 3 种扩散效果



(a) PadSpread(default)

(b) ReflectSpread

(c) RepeatSpread

图 10-6 辐射渐变的 3 种扩散方式

255, 255) 表示白色, QColor(0, 0, 0) 表示黑色; 而 a 表示 alpha 通道, 用来设置透明度, 取值也在 0~255 之间, 0 表示完全透明, 255 表示完全不透明。更多的颜色的知识, 可以参考 QColor 类的帮助文档。

3. 锥形渐变

锥形渐变 QConicalGradient::QConicalGradient (const QPointF & center, qreal angle) 需要指定中心点 center 和一个角度 angle(其值在 0~360 之间), 然后沿逆时针从给定的角度开始环绕中心点插入颜色。这里给定的角度沿逆时针方向开始的位置为 0, 旋转一圈后为 1。setSpread() 函数对于锥形渐变没有效果。

另外, 如果为画笔设置了渐变颜色, 那么可以绘制出渐变颜色的线条和轮廓, 还可以绘制出渐变颜色的文字。现在运行程序, 效果如图 10-7 所示。Qt 中提供了一个 Gradients 演示程序, 它在 Demonstrations 分类中, 在这个程序中可以设置任意的渐变填充效果, 可以参考一下。

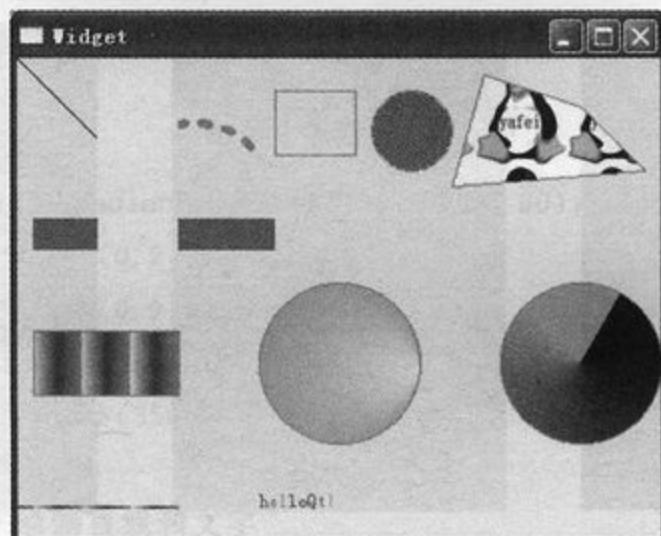


图 10-7 渐变填充运行效果

10.2 坐标系统

Qt 的坐标系统是由 QPainter 类控制的,而 QPainter 是在绘图设备上绘制的。绘图设备类 QPaintDevice 是所有可以绘制的对象的基类,它的子类主要有 QWidget、QPixmap、QPicture、QImage 和 QPrinter 等。一个绘图设备的默认坐标系统中原点(0, 0)在其左上角,x 坐标向右增长,y 坐标向下增长。在基于像素的设备上,默认的单位是一个像素,而在打印机上默认的单位是一个点(1/72 英寸)。

QPainter 的逻辑坐标与绘图设备的物理坐标之间的映射由 QPainter 的变换矩阵、视口和窗口处理。逻辑坐标和物理坐标默认是一致的。QPainter 也支持坐标变换(例如旋转和缩放)。对应本节的内容可以在帮助中查看 Coordinate System 关键字。

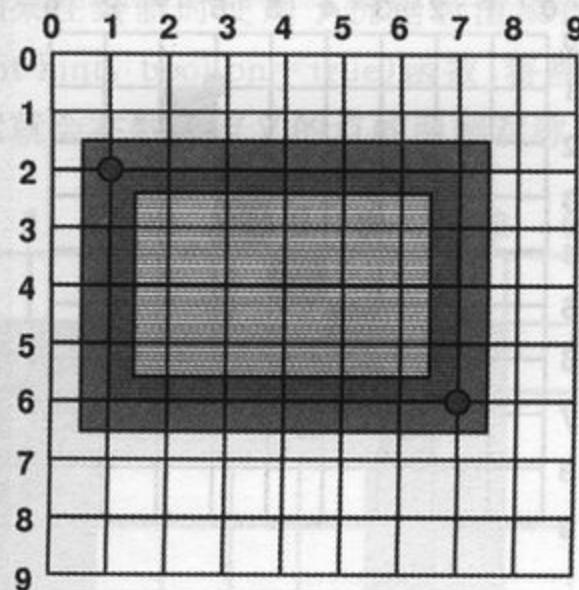
10.2.1 抗锯齿渲染

1. 逻辑表示

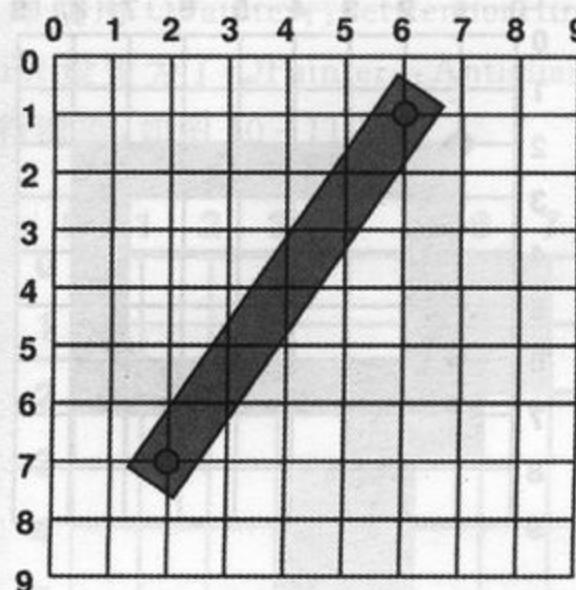
一个图形的大小(宽和高)总与其数学模型相对应,图 10-8 是忽略其渲染时使用的画笔的宽度时的样子。

2. 抗锯齿绘图

抗锯齿(Anti-aliased)又称为反锯齿或者反走样,就是对图像的边缘进行平滑处理,使其看起来更加柔和流畅的一种技术。QPainter 进行绘制时可以使用 QPainter::RenderHint 渲染提示来指定是否要使用抗锯齿功能,渲染提示的取值如表 10-2 所列。



(a) QRect(1,2,6,4)



(b) QLine(2,7,6,1)

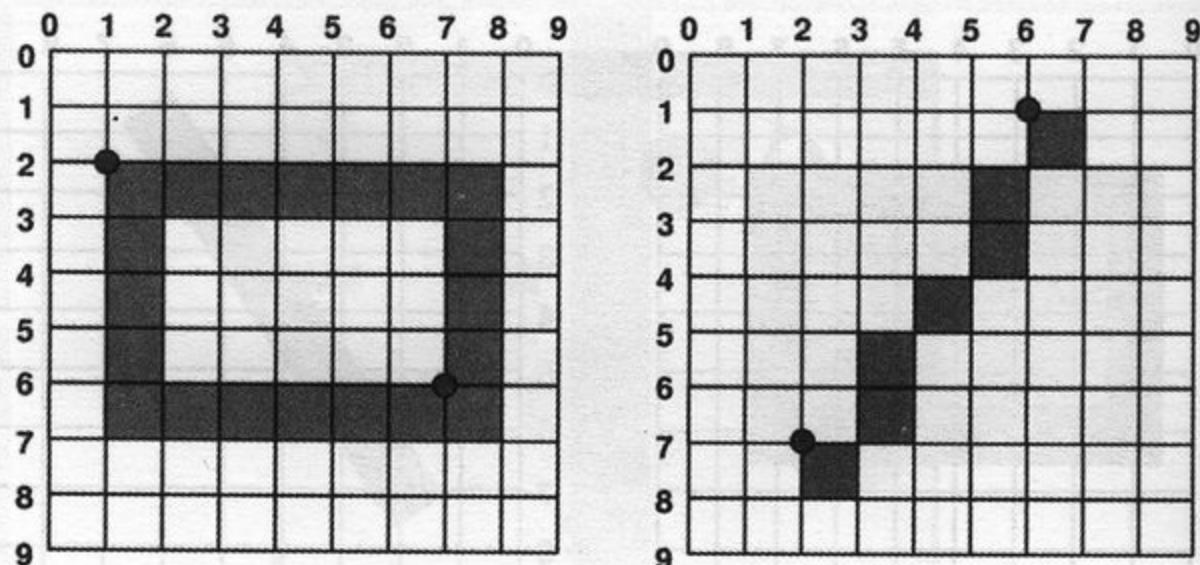
图 10-8 忽略画笔宽度示意图

表 10-2 QPainter 的渲染提示

常量	描述
QPainter::Antialiasing	指示绘图引擎在可能的情况下应该进行边缘的抗锯齿
QPainter::TextAntialiasing	指示绘图引擎在可能的情况下应该绘制抗锯齿的文字
QPainter::SmoothPixmapTransform	指示绘图引擎应该使用一个平滑 pixmap 转换算法(例如双线性插值)而不是最邻近插值算法
QPainter::HighQualityAntialiasing	一个 OpenGL 使用的渲染提示,指定绘图引擎应该使用 fragment programs 和 offscreen rendering 来进行抗锯齿
QPainter::NonCosmeticDefaultPen	绘图引擎应该将宽度为 0 的画笔看作是一个宽度为 1 的非装饰笔

默认情况下,绘制会产生锯齿,并且使用这样的规则进行绘制:当使用宽度为一个像素的画笔进行渲染时,像素会在数学定义的点右边和下边进行渲染,如图 10-9 所示。当使用一个拥有偶数像素的画笔进行渲染时,像素会在数学定义的点的周围对称渲染;而当使用一个拥有奇数像素的画笔进行渲染时,像素会被渲染到数学定义的点的右边和下边,如图 10-10 所示。

矩形可以用 QRect 类来表示,但是由于历史的原因,QRect::right() 和 QRect::bottom() 函数的返回值会偏离矩形真实的右下角。使用 QRect 的 right() 函数返回 left() + width() - 1,而 bottom() 函数返回 top() + height() - 1。建议使用 QRectF 来代替 QRect, QRectF 类在一个使用了浮点数精度的坐标平面中定义了一个矩形,QRectF::right() 和 QRectF::bottom() 会返回真实的右下角坐标。当然,也可以使用 QRect 类,应用 x() + width() 和 y() + height() 来确定右下角的坐标,而不要使用 right() 和 bottom() 函数。

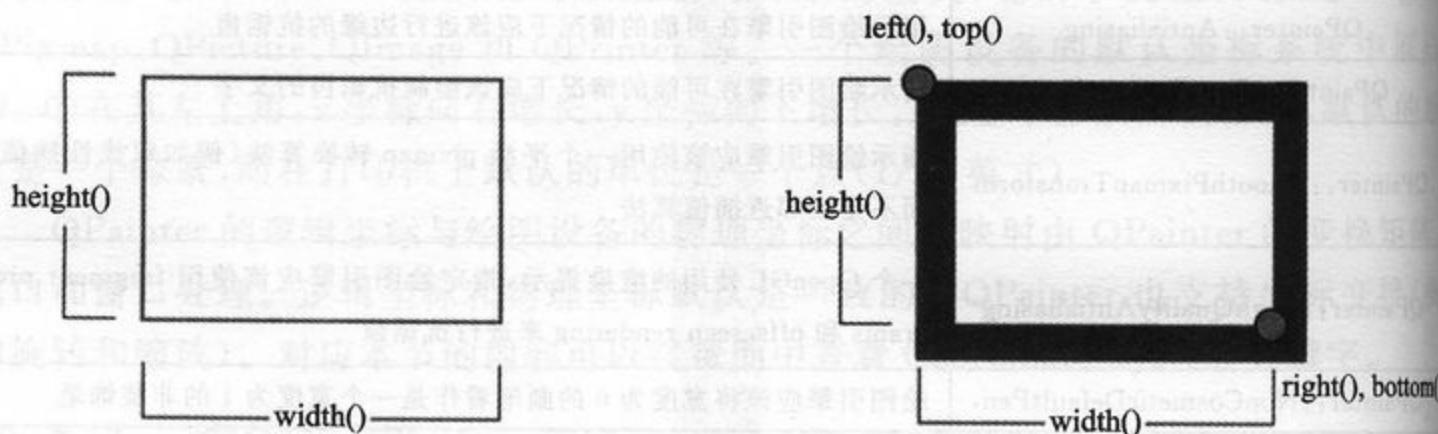


```
QPainter painter(this);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```

```
QPainter painter(this);
painter.setPen(Qt::darkGreen);
painter.drawLine(2, 7, 6, 1);
```

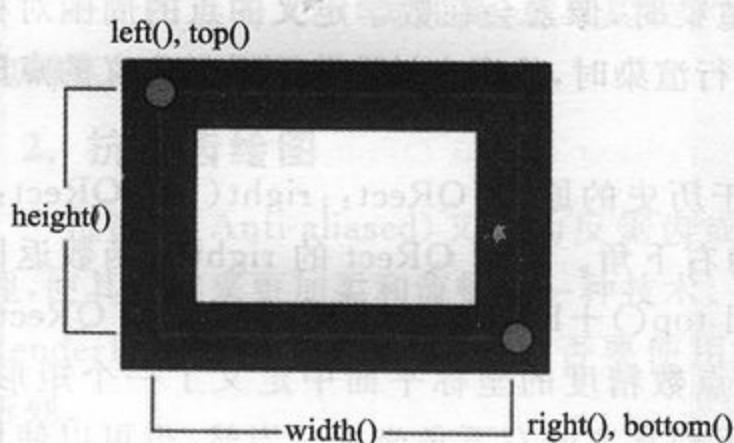
图 10-9 像素渲染规则示意图

QRectF

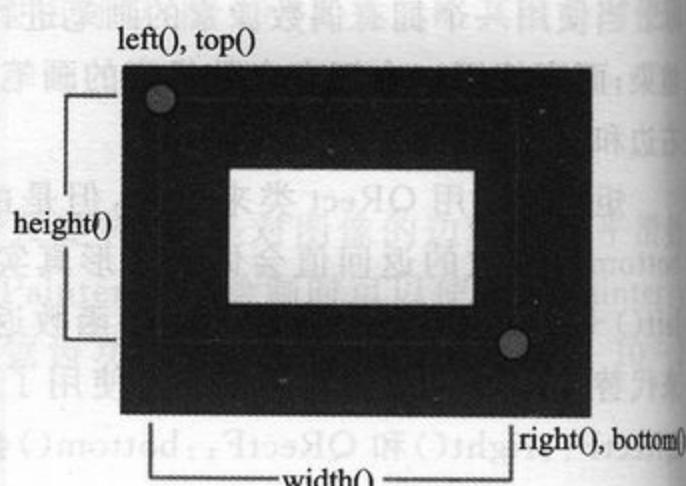


(a) 逻辑表示

(b) 宽度为一个像素的画笔



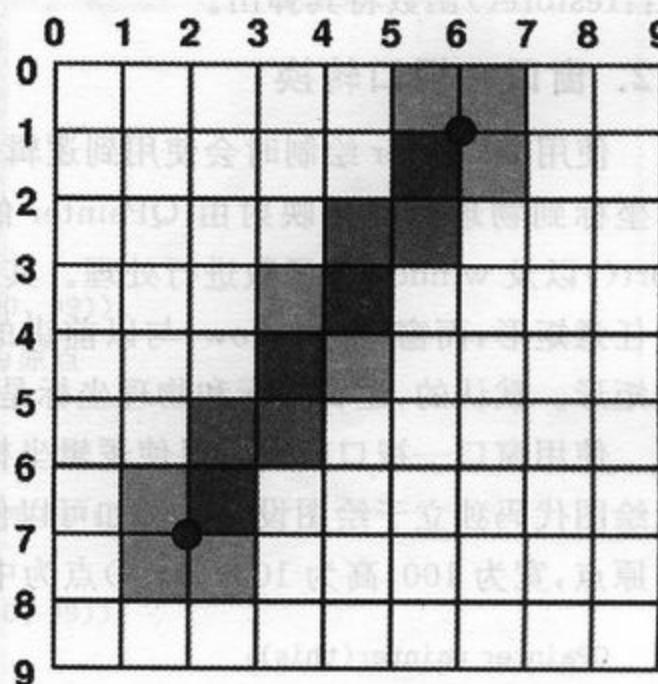
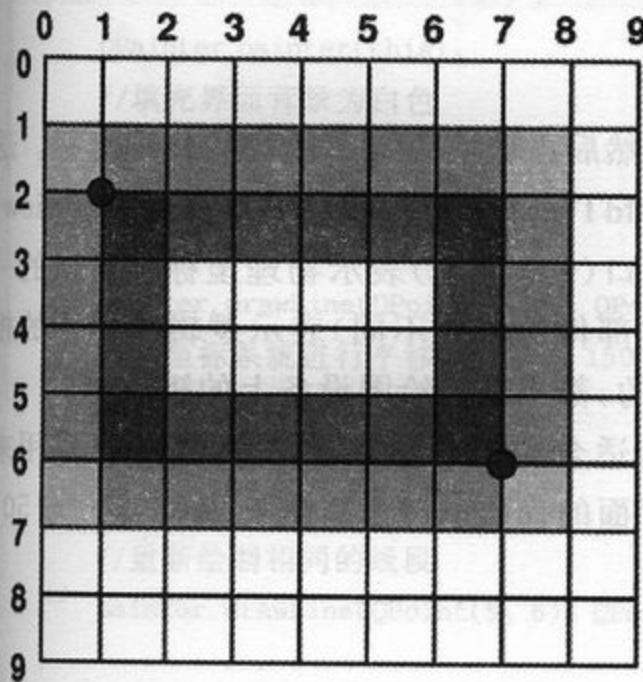
(c) 宽度为两个像素的画笔



(d) 宽度为3个像素的画笔

图 10-10 不同宽度画笔渲染示意图

如果在绘制时使用了抗锯齿渲染提示,即使用 `QPainter::setRenderHint(RenderHint hint, bool on=true)` 函数,将参数 `hint` 设置为了 `QPainter::Antialiasing`。那么像素就会在数学定义的点的两侧对称的进行渲染,如图 10-11 所示。



```
QPainter painter(this);
painter.setRenderHint(
    QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```

```
QPainter painter(this);
painter.setRenderHint(
    QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawLine(2, 7, 6, 1);
```

图 10-11 抗锯齿渲染示意图

10.2.2 坐标变换

1. 基本变换

默认的, `QPainter` 在相关设备的坐标系统上进行操作,但是完全支持仿射(affine)坐标变换。绘图时可以使用 `QPainter::scale()` 函数缩放坐标系统,使用 `QPainter::rotate()` 函数顺时针旋转坐标系统,使用 `QPainter::translate()` 函数平移坐标系统,可以使用 `QPainter::shear()` 围绕原点来扭曲坐标系统。

坐标系统的 2D 变换由 `QTransform` 类实现,可以使用前面提到的那些便捷函数进行坐标系统变换,当然也可以通过 `QTransform` 类实现;而且 `QTransform` 类对象可以存储多个变换操作,当同样的变换要多次使用时,建议使用 `QTransform` 类对象。坐标系统的变换是通过变换矩阵实现的,可以在平面上变换一个点到另一个点。进行所有变换操作的变换矩阵都可以使用 `QPainter::worldTransform()` 函数获得,如果要设置一个变换矩阵,可以使用 `QPainter::setWorldTransform()` 函数,这两个函数也可以分别使用 `QPainter::transform()` 和 `QPainter::setTransform()` 函数来代替。

在进行变换操作时,可能需要多次改变坐标系统再恢复,这样就显得很乱,而且很容易出现操作错误。这时可以使用 `QPainter::save()` 函数来保存 `QPainter` 的变换矩阵,它会把变换矩阵保存到一个内部栈中,然后在需要恢复变换矩阵时再使用 `QPainter::restore()` 函数将其弹出。

2. 窗口—视口转换

使用 `QPainter` 绘制时会使用到逻辑坐标,然后再转换为绘图设备的物理坐标。逻辑坐标到物理坐标的映射由 `QPainter` 的 `worldTransform()` 函数、`QPainter` 的 `viewport()` 以及 `window()` 函数进行处理。其中,视口(`viewport`)表示物理坐标下指定的一个任意矩形,而窗口(`window`,与以前讲的窗口部件的概念不同)表示逻辑坐标下的相同矩形。默认的,逻辑坐标和物理坐标是重合的,都相当于绘图设备上的矩形。

使用窗口—视口转换可以使逻辑坐标系统适合应用的要求,这个机制也可以用来让绘图代码独立于绘图设备。例如可以使用下面的代码来使逻辑坐标以 $(-50, -50)$ 为原点,宽为 100,高为 100, $(0, 0)$ 点为中心:

```
QPainter painter(this);
painter.setWindow(QRect(-50, -50, 100, 100));
```

现在逻辑坐标 $(-50, -50)$ 对应绘图设备的物理坐标 $(0, 0)$ 点,这样就可以独立于绘图设备,使绘图代码在指定的逻辑坐标上进行操作了。当设置窗口或者视口矩形时,实际上是执行了坐标的一个线性变换,窗口的 4 个角会映射到视口对应的 4 个角,反之亦然。因此,一个很好的办法是让视口和窗口维持相同的宽高比来防止变形:

```
int side = qMin(width(), height());
int x = (width() - side / 2);
int y = (height() - side / 2);
painter.setViewport(x, y, side, side);
```

如果设置了逻辑坐标系统为一个正方形,那么也需要使用 `QPainter::setViewport()` 函数设置视口为正方形,例如这里将视口设置为适合绘图设备矩形的最大矩形。在设置窗口或视口时考虑到绘图设备的大小,就可以使绘图代码独立于绘图设备。

窗口—视口转换仅仅是线性变换,不会执行裁剪操作,这就意味着如果绘制范围超出了当前设置的窗口,那么仍然会使用相同的线性代数方法将绘制变换到视口上。在绘制过程中是先使用坐标矩阵进行变换,再使用窗口—视口转换。

前面讲到的知识可能不是很容易理解,下面通过实际的程序来进一步理解这些知识点。(项目源码路径: `src\10\10-2\myTransformation`)新建 Qt Gui 应用,项目名称为 `myTransformation`,基类选择 `QWidget`,类名为 `Widget`。建立完成后,在 `widget.h` 文件中声明重绘事件处理函数:

```
protected:
void paintEvent(QPaintEvent * event);
```

然后到 `widget.cpp` 文件中添加头文件 `#include <QPainter>`。下面是 `paintEvent()` 函数的定义：

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    //填充界面背景为白色
    painter.fillRect(rect(), Qt::white);
    painter.setPen(QPen(Qt::red, 11));
    //绘制一条线段
    painter.drawLine(QPoint(5, 6), QPoint(100, 99));
    //将坐标系统进行平移,使(200, 150)点作为原点
    painter.translate(200, 150);
    //开启抗锯齿
    painter.setRenderHint(QPainter::Antialiasing);
    //重新绘制相同的线段
    painter.drawLine(QPoint(5, 6), QPoint(100, 99));
}
```

这里先绘制了一条线段,然后使用 `translate()` 函数改变了坐标原点并重新绘制了前面的线段,该函数的两个参数分别为水平方向和垂直方向的偏移值。因为现在的坐标原点已经改变,也就是说会以(200, 150)作为新的原点(0, 0),所以两条线段并不会重合。而且在绘制第二条线段时使用了抗锯齿,所以可以看出它比第一条线段要平滑许多。在程序中,要想将坐标原点再还原回去,可以进行反向平移,即使用 `translate(-200, -150)`。运行程序,效果如图 10-12 所示。

下面继续在 `paintEvent()` 函数中添加如下代码:

```
//保存 painter 的状态
painter.save();
//将坐标系统旋转 90 度
painter.rotate(90);
painter.setPen(Qt::cyan);
//重新绘制相同的线段
painter.drawLine(QPoint(5, 6), QPoint(100, 99));
//恢复 painter 的状态
painter.restore();
```

这里先使用 `save()` 函数保存了 `painter` 的当前状态,然后将坐标系统旋转并绘制了同以前一样的线段,不过,因为坐标系统已经旋转了,所以这条线段也不会和前面的线段重合。这里的 `rotate()` 函数会以原点为中心进行旋转,参数为旋转的角度,正数为顺时针旋转,负数为逆时针旋转。最后使用 `restore()` 函数恢复了 `painter` 以前的状态,就是恢复到了旋转以前的坐标系统和画笔颜色。运行程序,效果如图 10-13 所示。

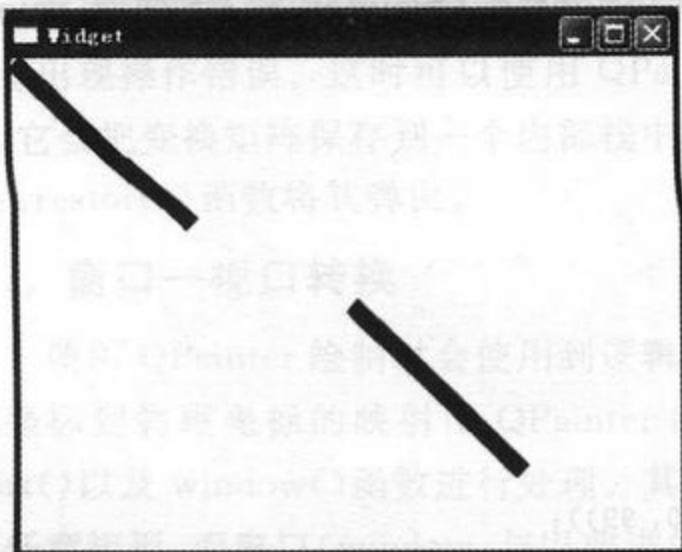


图 10-12 平移坐标系统运行效果

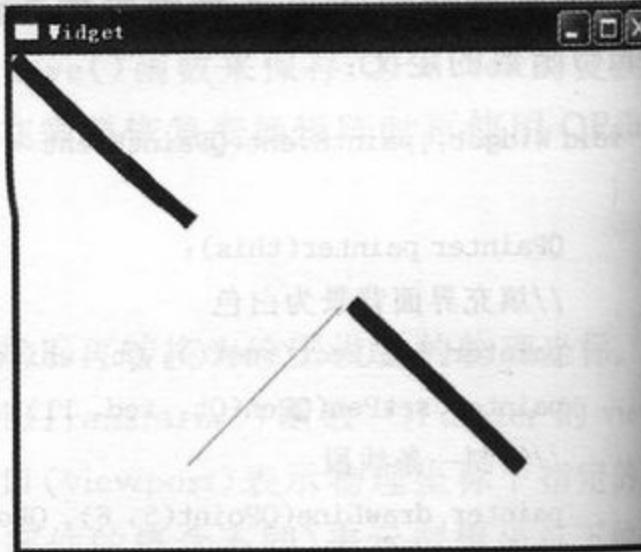


图 10-13 旋转坐标系统运行效果

下面继续添加代码：

```

painter.setBrush(Qt::darkGreen);
//绘制一个矩形
painter.drawRect(-50, -50, 100, 50);
painter.save();
//将坐标系统进行缩放
painter.scale(0.5, 0.4);
painter.setBrush(Qt::yellow);
//重新绘制相同的矩形
painter.drawRect(-50, -50, 100, 50);
painter.restore();

```

这里先绘制了一个矩形，然后将坐标系统进行缩放并绘制了相同的矩形，因为坐标系统已经改变，所以两个矩形不会重合。这里 scale() 函数的两个参数分别为水平方向和垂直方向缩放的倍数。运行程序，效果如图 10-14 所示。

继续在 paintEvent() 函数中添加代码：

```

painter.setPen(Qt::blue);
painter.setBrush(Qt::darkYellow);
//绘制一个椭圆
painter.drawEllipse(QRect(60, -100, 50, 50));
//将坐标系统进行扭曲
painter.shear(1.5, -0.7);
painter.setBrush(Qt::darkGray);
//重新绘制相同的椭圆
painter.drawEllipse(QRect(60, -100, 50, 50));

```

这里先绘制了一个椭圆，然后将坐标系统进行扭曲并绘制了相同的椭圆，因为坐标系统已经改变，所以两个椭圆不会重合。这里 shear() 函数的两个参数分别为水平方向和垂直方向的扭曲值，当其值为 0 时表示不进行扭曲。运行程序，效果如图 10-15

所示。

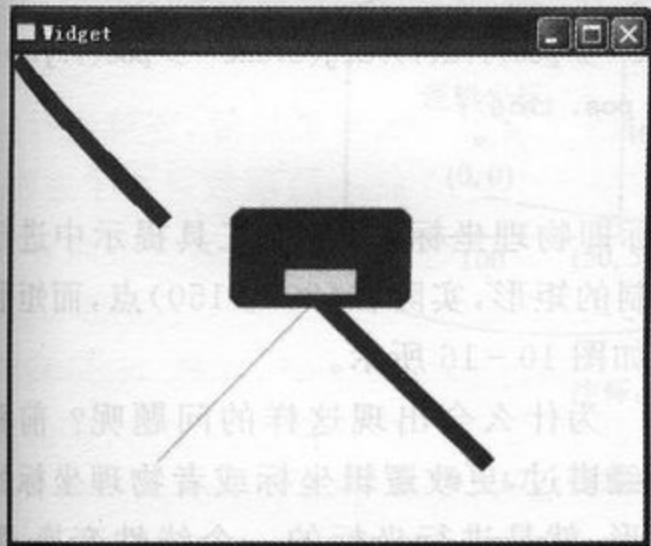


图 10-14 缩放坐标系统运行效果

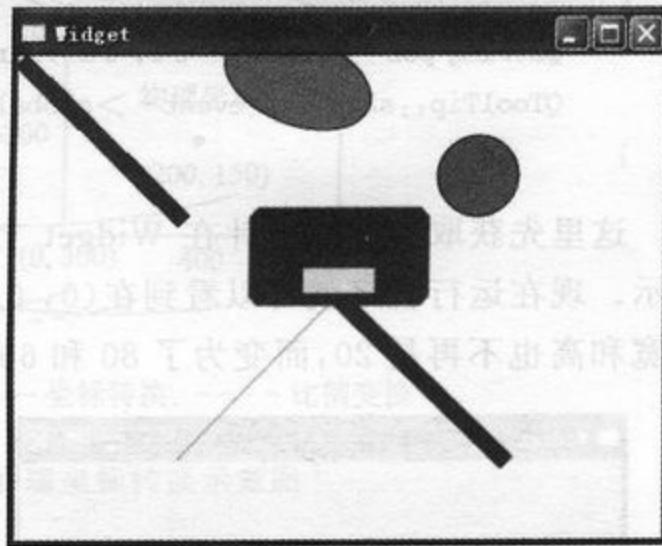


图 10-15 扭曲坐标系统运行效果

(项目源码路径: src\10\10-3\myTransformation)下面来看一下窗口—视口转换的内容,先将前面 paintEvent() 函数中的所有内容都删除,然后更改如下:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    painter.setWindow(-50, -50, 100, 100);
    painter.setBrush(Qt::green);
    painter.drawRect(0, 0, 20, 20);
}
```

这里先使用 setWindow() 函数将逻辑坐标矩形设置为以 (-50, -50) 为起点, 宽 100, 高 100。这样逻辑坐标的 (-50, -50) 点就会对应物理坐标的 (0, 0) 点, 因为这里是在 this 即 Widget 部件上进行绘图, 所以 Widget 就是绘图设备, 也就是说, 现在逻辑坐标的 (-50, -50) 点对应界面上的左上角的 (0, 0) 点。而且, 因为逻辑坐标矩形宽为 100, 高为 100, 所以界面的宽度和高度都会被 100 等分。下面在界面上显示出物理坐标, 帮助理解。在 widget.h 文件的 protected 域中声明鼠标移动事件处理函数:

```
void mouseMoveEvent(QMouseEvent * event);
```

然后在 widget.cpp 文件中添加头文件:

```
#include <QToolTip>
#include <QMouseEvent>
```

再在构造函数中添加如下一行代码, 保证不用按下鼠标按键也能触发鼠标移动事件:

```
setMouseTracking(true);
```

最后定义鼠标移动事件处理函数:

```

void Widget::mouseMoveEvent(QMouseEvent * event)
{
    QString pos = QString(" %1, %2").arg(event->pos().x()).arg(event->pos().y());
    QToolTip::showText(event->globalPos(), pos, this);
}

```

这里先获取了鼠标指针在 Widget 上的坐标即物理坐标,然后在工具提示中进行显示。现在运行程序就可以看到在(0, 0)点绘制的矩形,实际在(200, 150)点,而矩形的宽和高也不再是 20,而变为了 80 和 60,效果如图 10-16 所示。

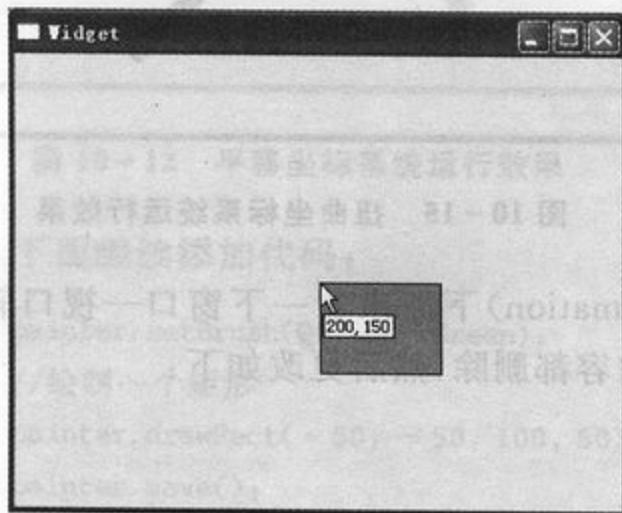


图 10-16 改变逻辑坐标运行效果

为什么会出现这样的问题呢?前面已经讲过,更改逻辑坐标或者物理坐标的矩形,就是进行坐标的一个线性变换,逻辑坐标矩形的 4 个角会映射到对应物理坐标矩形的 4 个角。而现在 Widget 部件的大小为宽 400,高 300,所以物理坐标对应的矩形就是(0, 0, 400, 300)。这样按比例对应,就是在水平方向,逻辑坐标的 1 个单位对应物理坐标的 4 个单位;在垂直方向,逻辑坐标的 1 个单位,对应物理坐标的 3 个单位,如图 10-17 所示。所以逻辑坐标中的宽 20 高 20 的矩形在物理坐标中就是宽 80 高 60 的矩形。

可以看到设置的矩形已经发生了变形,由设置的正方形变成了一个长方形。为了防止变形,需要将视口的宽和高的对应比例设置为相同值,因为逻辑坐标的矩形设置为了一个正方形,所以视口即物理坐标矩形也应该设置为一个正方形,更改 paintEvent() 函数如下:

```

void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    int side = qMin(width(), height());
    int x = (width() / 2);
    int y = (height() / 2);
    //设置视口
    painter.setViewport(x, y, side, side);
    painter.setWindow(0, 0, 100, 100);
    painter.setBrush(Qt::green);
    painter.drawRect(0, 0, 20, 20);
}

```

这样绘制出来的矩形就是正方形了。下面再来看将定时器和 2D 绘图相结合实现简单动画的应用。

(项目源码路径: src\10\10-4\myTransformation)继续在前面的程序中进行更改。

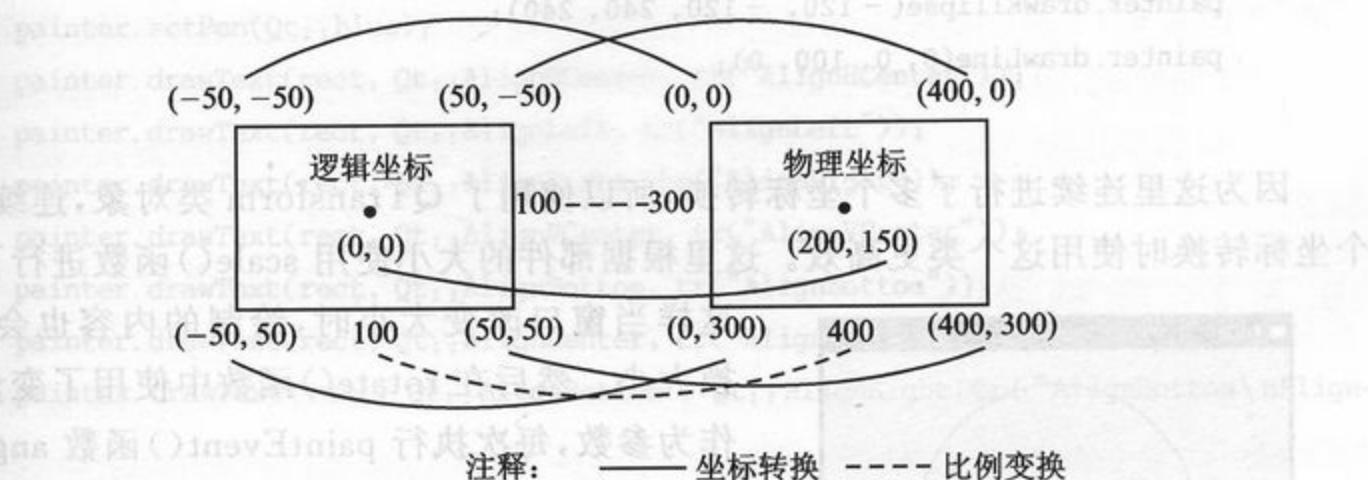


图 10-17 逻辑坐标与物理坐标转换示意图

首先在 `widget.h` 文件中添加前置声明：

```
class QTimer;
```

然后添加两个私有变量的声明：

```
QTimer * timer;
int angle;
```

再进入 `widget.cpp` 文件中，添加头文件 `#include <QTimer>`，并在构造函数中添加代码：

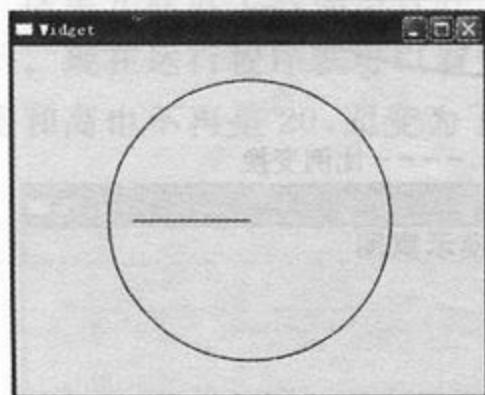
```
QTimer * timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(update()));
timer->start(1000);
angle = 0;
```

这里创建了一个定时器，并将定时器的溢出信号关联到了 Widget 部件的 `update()` 槽上，然后开启了一个 1 秒的定时器，这样，每过 1 秒钟都会执行一次 `paintEvent()` 函数。下面将 `paintEvent()` 函数更改如下：

```
void Widget::paintEvent(QPaintEvent * event)
{
    angle += 10;
    if(angle == 360)
        angle = 0;
    int side = qMin(width(), height());
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    QTransform transform;
    transform.translate(width()/2, height()/2);
    transform.scale(side/300.0, side/300.0);
    transform.rotate(angle);
    painter.setWorldTransform(transform);
```

```
painter.drawEllipse(-120, -120, 240, 240);
painter.drawLine(0, 0, 100, 0);
}
```

因为这里连续进行了多个坐标转换,所以使用了 `QTransform` 类对象,连续进行多个坐标转换时使用这个类更高效。这里根据部件的大小使用 `scale()` 函数进行了缩放,



这样当窗口改变大小时,绘制的内容也会跟着变换大小。然后在 `rotate()` 函数中使用了变量 `angle` 作为参数,每次执行 `paintEvent()` 函数 `angle` 都增加 10° ,这样就会旋转一个不同的角度,当其值为 360 时将它重置为 0。运行程序,效果如图 10-18 所示,程序实现了一个每隔 1 秒走动一下的表针动画。

关于坐标系统的应用,Qt 中提供了 `Analog Clock` 程序和 `Transformations` 程序,分别在 `Widgets` 和 `Painting` 分类中。另外还有一个 `Affine Transformations` 演示程序,可以参考一下。

10.3 其他绘制

10.3.1 绘制文字

除了绘制图形以外,还可以使用 `QPainter::drawText()` 函数来绘制文字,也可以使用 `QPainter::setFont()` 设置文字所使用的字体,使用 `QPainter::fontInfo()` 函数可以获取字体的信息,它返回 `QFontInfo` 类对象。在绘制文字时会默认使用抗锯齿。

(项目源码路径: `src\10\10-5\myDrawing2`) 新建 Qt Gui 应用,项目名称为 `myDrawing2`,基类选择 `QWidget`,类名为 `Widget`。建立完成后,在 `widget.h` 文件中声明重绘事件处理函数:

```
protected:
void paintEvent(QPaintEvent * event);
```

然后到 `widget.cpp` 文件中添加头文件 `#include <QPainter>`。下面是 `paintEvent()` 函数的定义:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    QRectF rect(10.0, 10.0, 380.0, 280.0);
    painter.setPen(Qt::red);
    painter.drawRect(rect);
```

```

painter.setPen(Qt::blue);
painter.drawText(rect, Qt::AlignHCenter, tr("AlignHCenter"));
painter.drawText(rect, Qt::AlignLeft, tr("AlignLeft"));
painter.drawText(rect, Qt::AlignRight, tr("AlignRight"));
painter.drawText(rect, Qt::AlignVCenter, tr("AlignVCenter"));
painter.drawText(rect, Qt::AlignBottom, tr("AlignBottom"));
painter.drawText(rect, Qt::AlignCenter, tr("AlignCenter"));
painter.drawText(rect, Qt::AlignBottom | Qt::AlignRight, tr("AlignBottom\nAlignRight"));
}

```

这里使用了绘制文本函数的一种重载形式 `QPainter::drawText (const QRectF & rectangle, int flags, const QString & text, QRectF * boundingRect=0)`, 第一个参数指定了绘制文字所在的矩形;第二个参数指定了文字在矩形中的对齐方式,由 `Qt::AlignmentFlag` 枚举变量进行定义,不同对齐方式也可以使用“|”操作符同时使用,这里还可以使用 `Qt::TextFlag` 定义的其他一些标志,比如自动换行等;第三个参数就是所要绘制的文字,这里可以使用“\n”来实现换行;第四个参数一般不用设置。如果绘制的文字和它的布局不用经常改动,那么也可以使用 `drawStaticText()` 函数,它更高效。

下面在 `paintEvent()` 函数中继续添加如下代码:

```

QFont font("宋体", 15, QFont::Bold, true);
//设置下划线
font.setUnderline(true);
//设置上划线
font.setOverline(true);
//设置字母大小写
font.setCapitalization(QFont::SmallCaps);
//设置字符间的间距
font.setLetterSpacing(QFont::AbsoluteSpacing, 10);
//使用字体
painter.setFont(font);
painter.setPen(Qt::green);
painter.drawText(120, 80, tr("yafeilinux"));

painter.translate(100, 100);
painter.rotate(90);
painter.drawText(0, 0, tr("helloqt"));

```

这里创建了 `QFont` 字体对象,使用的构造函数为 `QFont::QFont (const QString & family, int pointSize=-1, int weight=-1, bool italic=false)`,第一个参数设置字体的 `family` 属性,这里使用的字体族为宋体,可以使用 `QFontDatabase` 类来获取所支持的所有字体;第二个参数是点大小,默认大小为 12;第三个参数为 `weight` 属性,这

里使用了粗体；最后一个属性设置是否使用斜体。然后我们又使用了其他几个函数来设置字体的格式，最后调用 `setFont()` 函数来使用该字体，并使用 `drawText()` 函数的另一种重载形式在点(120, 80)绘制了文字。后面又将坐标系统平移并旋转，然后再次绘制了文字。

10.3.2 绘制路径

如果要绘制一个复杂的图形，尤其是要重复绘制这样的图形，那么可以使用 `QPainterPath` 类对象，然后使用 `QPainter::drawPath()` 来进行绘制。`QPainterPath` 类为绘制操作提供了一个容器，可以用来创建图形并且进行重复使用。一个绘图路径就是由多个矩形、椭圆、线条或者曲线等组成的对象，一个路径可以是封闭的，例如矩形和椭圆；也可以是非封闭的，例如线条和曲线。

1. 组成一个路径

(项目源码路径：src\10\10-6\myDrawing2)现在来绘制一个路径，将上面程序中的 `paintEvent()` 函数中的内容删除，然后更改如下：

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    QPainterPath path;
    //移动当前点到点(50, 250)
    path.moveTo(50, 250);
    //从当前点即(50, 250)绘制一条直线到点(50, 230),完成后当前点更改为(50, 230)
    path.lineTo(50, 230);
    //从当前点和点(120, 60)之间绘制一条三次贝塞尔曲线
    path.cubicTo(QPointF(105, 40), QPointF(115, 80), QPointF(120, 60));
    path.lineTo(130, 130);
    //向路径中添加一个椭圆
    path.addEllipse(QPoint(130, 130), 30, 30);

    painter.setPen(Qt::darkYellow);
    //绘制路径
    painter.drawPath(path);

    //平移坐标系统后重新绘制路径
    path.translate(200,0);
    painter.setPen(Qt::darkBlue);
    painter.drawPath(path);
}
```

创建一个 `QPainterPath` 对象后就会以坐标原点为当前点进行绘制，可以随时使用 `moveTo()` 函数改变当前点，比如程序中移动到了点(50, 250)，那么下次就会从该点开

始进行绘制；可以使用 `lineTo()`、`arcTo()`、`cubicTo()` 和 `quadTo()` 等函数将直线或者曲线添加到路径中，其中 `QPainterPath::cubicTo (const QPointF & c1, const QPointF & c2, const QPointF & endPoint)` 函数可以在当前点和 `endPoint` 点之间添加一个三次贝塞尔曲线，其中的 `c1` 和 `c2` 是控制点，如图 10-19 所示。`quadTo()` 函数可以绘制一个二次贝塞尔曲线；可以使用 `addEllipse()`、`addPath()`、`addRect()`、`addRegion()`、`addText()` 和 `addPolygon()` 来向路径中添加一些图形或者文字，它们都从当前点开始进行绘制，绘制完成后以结束点作为新的当前点，这些图形都是由一组直线或者曲线组成的。例如矩形就是顺时针添加的一组直线，绘制完成后当前点在矩形的左上角；而椭圆由一组顺时针曲线组成，开始点和结束点都在 0° 处（3点钟的位置）。另外还可以使用 `addPath()` 来添加其他的路径，这样会从本路径的当前点和要添加路径的第一个组件间添加一条直线。可以使用 `currentPosition()` 函数获取当前点，使用 `moveTo()` 函数改变当前点；当组建好路径后可以使用 `drawPath()` 函数来绘制路径，这里使用 `translate()` 函数将路径平移后又重新绘制了路径，可以看到这样就可以重复绘制复杂的图形了，这也是 `QPainterPath` 的主要作用。运行程序，效果如图 10-20 所示。

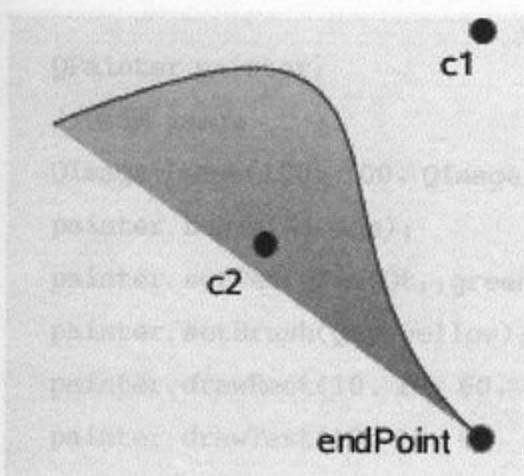


图 10-19 绘制三次贝塞尔曲线示意图

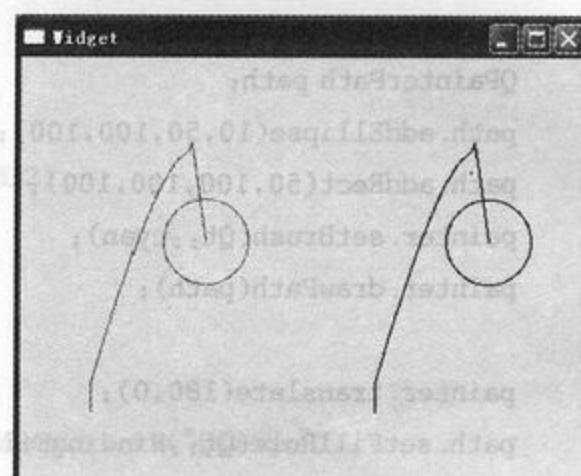


图 10-20 绘制路径运行效果

2. 填充规则

前面在绘制多边形时就提到了填充规则 `Qt::FillRule`，在填充路径时也要使用填充规则，这里一共有两个填充规则：`Qt::OddEvenFill` 和 `Qt::WindingFill`。其中，`Qt::OddEvenFill` 使用的是奇偶填充规则，具体来说就是：如果要判断一个点是否在图形中，那么可以从该点向图形外引一条水平线，如果该水平线与图形的交点的个数为奇数，那么该点就在图形中。这个规则是默认值；而 `Qt::WindingFill` 使用的是非零弯曲规则，具体来说就是：如果要判断一个点是否在图形中，那么可以从该点向图形外引一条水平线，那么如果该水平线与图形的边线相交，这个边线是顺时针绘制的，就记为 1，是逆时针绘制的就记为 -1，然后将所有数值相加，如果结果不为 0，那么该点就在图形中。图 10-21 是这两种规则的示意图，对于 `Qt::OddEvenFill` 规则，第一个交点记为 1，第二个交点记为 2；对于 `Qt::WindingFill` 规则，因为椭圆和矩形都是以顺时针进行绘制的，所以各个交点对应的边都使用 1 来代表。

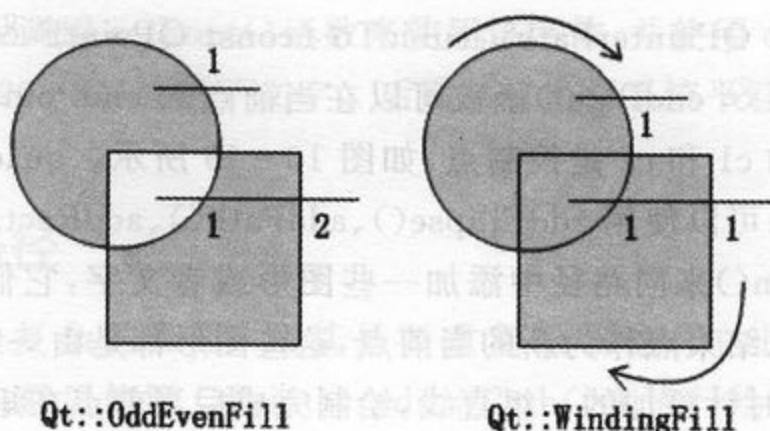


图 10-21 填充规则示意图

(项目源码路径: src\10\10-7\myDrawing2) 将 paintEvent() 函数中以前的内容清空, 然后更改如下:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    QPainterPath path;
    path.addEllipse(10,50,100,100);
    path.addRect(50,100,100,100);
    painter.setBrush(Qt::cyan);
    painter.drawPath(path);

    painter.translate(180,0);
    path.setFillRule(Qt::WindingFill);
    painter.drawPath(path);
}
```

这里先绘制了一个包含了相交的椭圆和矩形的路径, 因为没有显式指定填充规则, 则默认会使用 Qt::OddEvenFill 规则。然后将路径进行平移, 重新使用 Qt::WindingFill 规则绘制了该路径。

另外可以使用 QPainter::fillPath() 来填充一个路径; QPainter::strokePath() 函数来绘制路径的轮廓; QPainterPath::elementAt() 来获取路径中的一个元素; QPainterPath::elementCount() 获取路径中元素的个数; QPainterPath::contains() 函数来判断一个点是否在路径中; 还可以使用 QPainterPath::toFillPolygon() 函数将路径转换为一个多边形。这部分内容可以参考 Painter Paths 示例程序, 它在 Painting 分类中, 另外 Qt 还提供了一个 Vector Deformation 演示程序。

10.3.3 绘制图像

Qt 提供了 4 个类来处理图像数据: QImage、QPixmap、QBitmap 和 QPicture, 都是常用的绘图设备。其中 QImage 主要用来进行 I/O 处理, 对 I/O 处理操作进行了优化,

而且也可以用来直接访问和操作像素；QPixmap 主要用来在屏幕上显示图像，它对在屏幕上显示图像进行了优化；QBitmap 是 QPixmap 的子类，是一个便捷类，用来处理颜色深度为 1 的图像，即只能显示黑白两种颜色；QPicture 用来记录并重演 QPainter 命令。下面来看一下在这几个绘图设备上绘制图形的效果。

(项目源码路径：src\10\10-8\myDrawing3) 新建 Qt Gui 应用，项目名称为 myDrawing3，基类选择 QWidget，类名为 Widget。建立完成后，在 widget.h 文件中声明重绘事件处理函数，然后到 widget.cpp 文件中添加头文件：

```
#include <QPainter>
#include <QImage>
#include <QPixmap>
#include <QBitmap>
#include <QPicture>
```

下面是 paintEvent() 函数的定义：

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter;
    //绘制 image
    QImage image(100, 100, QImage::Format_ARGB32);
    painter.begin(&image);
    painter.setPen(QPen(Qt::green, 3));
    painter.setBrush(Qt::yellow);
    painter.drawRect(10, 10, 60, 60);
    painter.drawText(10, 10, 60, 60, Qt::AlignCenter, tr("QImage"));
    painter.setBrush(QColor(0, 0, 0, 100));
    painter.drawRect(50, 50, 40, 40);
    painter.end();
    //绘制 pixmap
    QPixmap pix(100, 100);
    painter.begin(&pix);
    painter.setPen(QPen(Qt::green, 3));
    painter.setBrush(Qt::yellow);
    painter.drawRect(10, 10, 60, 60);
    painter.drawText(10, 10, 60, 60, Qt::AlignCenter, tr("QPixmap"));
    painter.setBrush(QColor(0, 0, 0, 100));
    painter.drawRect(50, 50, 40, 40);
    painter.end();
    //绘制 bitmap
    QBitmap bit(100, 100);
    painter.begin(&bit);
    painter.setPen(QPen(Qt::green, 3));
```

```

painter.setBrush(Qt::yellow);
painter.drawRect(10, 10, 60, 60);
painter.drawText(10, 10, 60, 60, Qt::AlignCenter, tr("QBitmap"));
painter.setBrush(QColor(0, 0, 0, 100));
painter.drawRect(50, 50, 40, 40);
painter.end();
//绘制 picture
QPicture picture;
painter.begin(&picture);
painter.setPen(QPen(Qt::green, 3));
painter.setBrush(Qt::yellow);
painter.drawRect(10, 10, 60, 60);
painter.drawText(10, 10, 60, 60, Qt::AlignCenter, tr("QPicture"));
painter.setBrush(QColor(0, 0, 0, 100));
painter.drawRect(50, 50, 40, 40);
painter.end();
//在 widget 部件上进行绘制
painter.begin(this);
painter.drawImage(50, 20, image);
painter.drawPixmap(200, 20, pix);
painter.drawPixmap(50, 170, bit);
painter.drawPicture(200, 170, picture);
}

```

这里分别在 4 个绘图设备上绘制了两个相交的正方形,较小的正方形使用了透明的黑色进行填充,在较大的正方形的中间绘制了文字。在定义 QImage、QPixmap 和 QBitmap 对象时均指定了它们的大小,即宽和高均为 100。而且,各个绘图设备都有自己的坐标系统,它们的左上角为原点。在进行绘制时,因为所绘制的图形没有占完设置的大小,也没有设置背景填充色,所以背景应该为透明的。这里还要看到,在各个不同的绘图设备上进行绘制时,都使用了 begin() 函数来指定设备,等绘制完成后再使用 end() 函数来结束绘制。最后,将这 4 张图像绘制到了窗口界面上。运行程序,效果如图 10-22 所示。可以看到,QPixmap 的透明背景显示为黑色,QBitmap 只能显示路径的轮廓。

1. QImage

QImage 类提供了一个与硬件无关的图像表示方法,可以直接访问像素数据,也可以作为绘图设备。因为 QImage 是 QPainterDevice 的子类,所以 QPainter 可以直接在 QImage 对象上进行绘制。当在 QImage 上使用 QPainter 时,绘制操作会在当前 GUI 线程以外的其他线程中执行。QImage 支持的图像格式如表 10-3 所列,它们包含了单色、8 位、32 位和 alpha 混合格式图像。QImage 提供了获取图像各种信息的相关函数,还提供了一些转换图像的函数。QImage 使用了隐式数据共享,所以可以进行值传递。

QImage 对象还可以使用数据流或者进行比较。

表 10-3 Qt 支持的图像格式

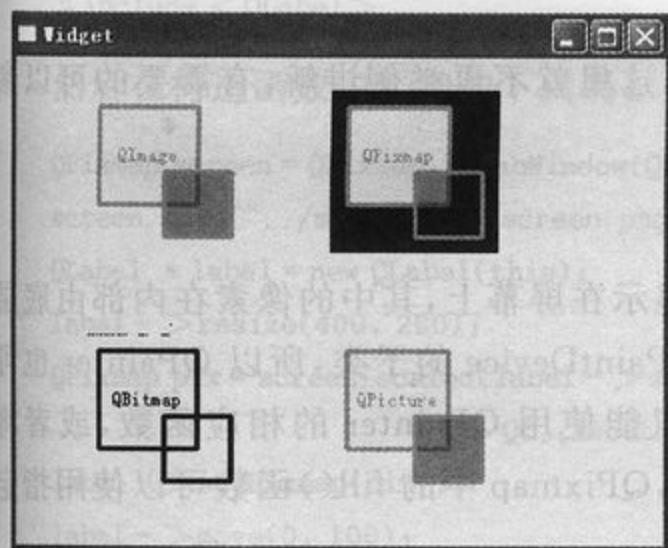


图 10-22 不同绘图设备绘图效果

格 式	Qt 的支持
BMP	读/写
GIF	读
JPG	读/写
JPEG	读/写
PNG	读/写
PBM	读
PGM	读
PPM	读/写
TIFF	读/写
XBM	读/写
XPM	读/写

(项目源码路径: src\10\10-9\myDrawing3)删除前面程序中 paintEvent() 函数里的内容,然后将其更改如下:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    QImage image;
    //加载一张图片
    image.load("../myDrawing3/image.png");
    //输出图片的一些信息
    qDebug() << image.size() << image.format() << image.depth();
    //在界面上绘制图片
    painter.drawImage(QPoint(10, 10), image);
    //获取镜像图片
    QImage mirror = image.mirrored();
    //将图片进行扭曲
    QTransform transform;
    transform.shear(0.2, 0);
    QImage image2 = mirror.transformed(transform);
    painter.drawImage(QPoint(10, 160), image2);
    //将镜像图片保存到文件
    image2.save("../myDrawing3/mirror.png");
}
```

因为使用了 qDebug() 函数,所以还要先添加头文件 #include <QDebug>。这里先为 QImage 对象加载了一张图片,然后输出了图片的一些信息,并将图片绘制到了界面上。然后使用 QImage::mirrored (bool horizontal=false, bool vertical=true) 函数

获取了该图片的镜像图片,默认返回的是垂直方向的镜像,也可以设置为水平方向的镜像。使用 transformed() 函数可以将图片进行各种坐标变换,最后使用了 save() 函数将图片存储到文件中。

QImage 类还提供了强大的操作像素功能,这里就不再举例讲解,有需要的可以参考 QImage 类的帮助文档。

2. QPixmap

QPixmap 可以作为一个绘图设备将图像显示在屏幕上,其中的像素在内部由底层的窗口系统来进行管理。因为 QPixmap 是 QPaintDevice 的子类,所以 QPainter 也可以直接在它上面进行绘制。要想访问像素,只能使用 QPainter 的相应函数,或者将 QPixmap 转换为 QImage。而与 QImage 不同,QPixmap 中的 fill() 函数可以使用指定的颜色初始化整个 pixmap 图像。

可以使用 toImage() 和 fromImage() 函数在 QImage 和 QPixmap 之间进行转换,通常情况下,QImage 类用来加载一个图像文件,随意操纵图像数据,然后将 QImage 对象转换为 QPixmap 类型再显示到屏幕上。当然,如果不需要对图像进行操作,那么也可以直接使用 QPixmap 来加载图像文件。与 QImage 不同,QPixmap 依赖于具体的硬件。QPixmap 类也是使用隐式数据共享,可以作为值进行传递。

QPixmap 可以很容易地使用 QLabel 或 QAbstractButton 的子类(比如 QPushButton)来显示在屏幕上。QLabel 拥有一个 pixmap 属性,而 QAbstractButton 拥有一个 icon 属性。还可以使用 grabWidget() 和 grabWindow() 等静态函数来实现截屏功能,使用 mask() 等函数实现遮罩效果。

(项目源码路径: src\10\10-10\myDrawing3)删除前面程序中 paintEvent() 函数里的内容,然后将其更改如下:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load("../myDrawing3/yafeilinux.png");
    painter.drawPixmap(0, 0, pix.width(), pix.height(), pix);
    painter.setBrush(QColor(255, 255, 255, 100));
    painter.drawRect(0, 0, pix.width(), pix.height());
    painter.drawPixmap(100, 0, pix.width(), pix.height(), pix);
    painter.setBrush(QColor(0, 0, 255, 100));
    painter.drawRect(100, 0, pix.width(), pix.height());
}
```

这里使用 QPixmap 先将同一图片并排绘制了两次,然后分别在其上面又绘制了一个使用不同的透明颜色填充的矩形,这样就可以使图像显示出不同的颜色,这使用到了下一节要讲到的复合模式的应用。

下面来实现截取屏幕的功能。在 widget.cpp 文件中再添加头文件：

```
#include <QDesktopWidget>
#include <QLabel>
```

然后在构造函数中添加如下代码：

```
QPixmap screen = QPixmap::grabWindow(QApplication::desktop() ->winId());
screen.save("../myDrawing3/screen.png");
QLabel * label = new QLabel(this);
label->resize(400, 200);
QPixmap pix = screen.scaled(label->size(), Qt::KeepAspectRatio,
                             Qt::SmoothTransformation);
label->setPixmap(pix);
label->move(0, 100);
```

使用 QPixmap::grabWindow (WId window, int x=0, int y=0, int width=-1, int height=-1) 静态函数可以截取屏幕的内容到一个 QPixmap 中，这里要指定窗口系统标识符 (The window system identifier, Wid)，还有要截取屏幕的内容所在的矩形，默认是截取整个屏幕的内容。除了截取屏幕，还可以使用 grabWidget() 来截取窗口部件上的内容。然后将截取到的图像显示在一个标签中，为了显示整张图像，这里将其进行了缩放，使用了函数 QPixmap::scaled (const QSize & size, Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio, Qt::TransformationMode transformMode=Qt::FastTransformation)，这个函数需要指定缩放后图片的大小 size，还要指定宽高比 Qt::AspectRatioMode 和转换模式 Qt::TransformationMode。这里的宽高比一共有 3 种取值，如表 10-4 所列。而转换模式默认是快速转换 Qt::FastTransformation，还有一种就是程序中使用的平滑转换 Qt::SmoothTransformation。关于截屏功能，可以参考 Screenshot 示例程序，它在 Desktop 分类中。

表 10-4 图像宽高比取值

常量	描述
Qt::IgnoreAspectRatio	可以自由缩放，不保持宽高比
Qt::KeepAspectRatio	在给定矩形中尽量放大，保持宽高比
Qt::KeepAspectRatioByExpanding	在给定的矩形外尽量缩小，保持宽高比

3. QPicture

QPicture 是一个可以记录和重演 QPainter 命令的绘图设备。QPicture 可以使用一个平台无关的格式 (.pic 格式) 将绘图命令序列化到 I/O 设备中，所有可以绘制在 QWidget 部件或者 QPixmap 上的内容，都可以保存在 QPicture 中。QPicture 与分辨率无关，在不同设备上的显示效果都是一样的。要记录 QPainter 命令，可以像如下代码这样进行：

```

QPicture picture;
QPainter painter;
painter.begin(&picture);
painter.drawEllipse(10,20, 80,70);
painter.end();
picture.save("drawing.pic");

```

要重演 QPainter 命令,可以像如下代码这样进行:

```

QPicture picture;
picture.load("drawing.pic");
QPainter painter;
painter.begin(&myImage);
painter.drawPicture(0, 0, picture);
painter.end();

```

10.3.4 复合模式

QPainter 提供了复合模式(Composition Modes)来定义如何完成数字图像的复合,即如何将源图像的像素和目标图像的像素合并。QPainter 提供的常用复合模式及其效果如图 10-23 所示,所有的复合模式可以在 QPainter 的帮助文档中进行查看。其中最普通的类型是 SourceOver(通常被称为 alpha 混合),就是正在绘制的源像素混合在已经绘制的目标像素上,源像素的 alpha 分量定义了它的透明度,这样源图像就会以透明效果在目标图像上进行显示。若绘图设备是 QImage,图像的格式一定要指定为 Format_ARGB32Premultiplied 或者 Format_ARGB32,不然复合模式就不会产生任何效果。当设置了复合模式,它就会应用到所有的绘图操作中,例如画笔、画刷、渐变和 pixmap/image 绘制等。

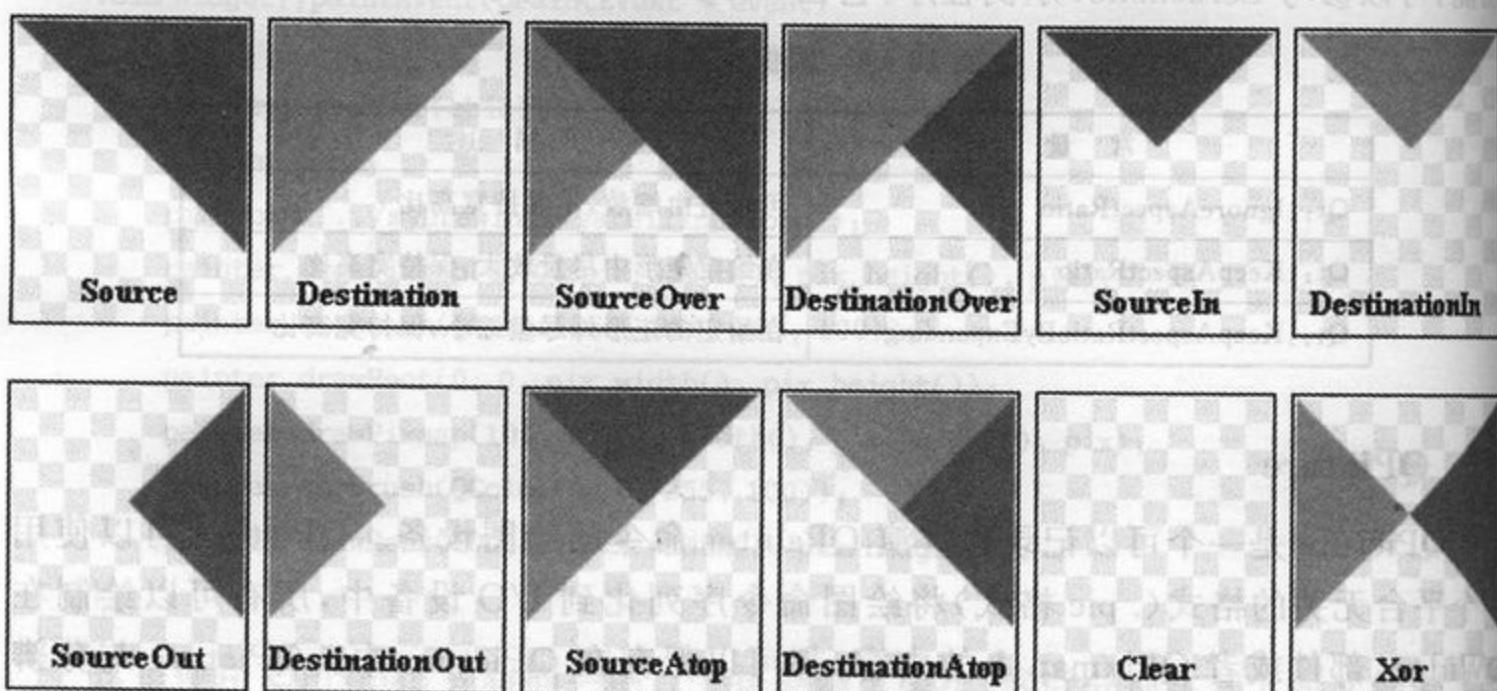


图 10-23 常用复合模式示意图

(项目源码路径: src\10\10-11\myComposition)新建 Qt Gui 应用,项目名称为 myComposition,基类选择 QWidget,类名为 Widget。建立完成后,在 widget.h 文件中声明重绘事件处理函数,然后到 widget.cpp 文件中添加头文件 #include <QPainter>,并定义 paintEvent() 函数如下:

```
void Widget::paintEvent(QPaintEvent * event)
{
    QPainter painter;
    QImage image(400, 300, QImage::Format_ARGB32_Premultiplied);
    painter.begin(&image);
    painter.setBrush(Qt::green);
    painter.drawRect(100, 50, 200, 200);
    painter.setBrush(QColor(0, 0, 255, 150));
    painter.drawRect(50, 0, 100, 100);
    painter.setCompositionMode(QPainter::CompositionMode_SourceIn);
    painter.drawRect(250, 0, 100, 100);
    painter.setCompositionMode(QPainter::CompositionMode_DestinationOver);
    painter.drawRect(50, 200, 100, 100);
    painter.setCompositionMode(QPainter::CompositionMode_Xor);
    painter.drawRect(250, 200, 100, 100);
    painter.end();
    painter.begin(this);
    painter.drawImage(0, 0, image);
}
```

这里先在 QImage 上绘制了一个矩形,然后又在这个矩形的 4 个角分别绘制了 4 个小矩形,每个小矩形都使用了不同的复合模式,并且使用了半透明的颜色进行填充。第一个小矩形没有明确指定复合模式,它默认使用的是 SourceOver 模式。关于复合模式的使用,可以参考 Painting 分类中的 Image Composition 示例程序,还可以看一下 Composition Modes 演示程序。

10.4 双缓冲绘图

所谓双缓冲(double-buffers)绘图,就是在进行绘制时先将所有内容都绘制到一个绘图设备(如 QPixmap)上,然后再将整个图像绘制到部件上显示出来。使用双缓冲绘图可以避免显示时的闪烁现象。从 Qt 4.0 开始,QWidget 部件的所有绘制都自动使用了双缓冲,所以一般没有必要在 paintEvent() 函数中使用双缓冲代码来避免闪烁。

虽然在一般的绘图中无需手动使用双缓冲绘图,不过要想实现一些绘图效果,还是要借助于双缓冲的概念。在下面的程序中,我们来实现使用鼠标在界面上绘制一个任意大小的矩形的例子。这里需要两张画布,它们都是 QPixmap 实例,其中一个 tempPix 用来作为临时缓冲区,当鼠标正在拖动矩形进行绘制时,将内容先绘制到 temp-

Pix 上,然后将 tempPix 绘制到界面上;而另一个 pix 作为缓冲区,用来保存已经完成的绘制。当松开鼠标完成矩形的绘制后,则将 tempPix 的内容复制到 pix 上。为了绘制时不显示拖影,那么在移动鼠标过程中,每绘制一次,都要在刚开始绘制这个矩形的图像上进行绘制,所以需要在每次绘制 tempPix 之前,先将 pix 的内容复制到 tempPix 上。

(项目源码路径: src\10\10-12\myDoubleBuffers)新建 Qt Gui 应用,项目名称为 myDoubleBuffers,基类选择 QWidget,类名为 Widget。建立完成后,在 widget.h 文件中添加如下内容:

```
protected:  
    void mousePressEvent(QMouseEvent * event);  
    void mouseMoveEvent(QMouseEvent * event);  
    void mouseReleaseEvent(QMouseEvent * event);  
    void paintEvent(QPaintEvent * event);  
  
private:  
    Ui::Widget * ui;  
    //缓冲区  
    QPixmap pix;  
    //临时缓冲区  
    QPixmap tempPix;  
    QPoint startPoint;  
    QPoint endPoint;  
    //是否正在绘图的标志  
    bool isDrawing;
```

然后到 widget.cpp 文件中,先添加头文件:

```
# include <QMouseEvent>  
# include <QPainter>
```

再在构造函数中对一些变量进行初始化:

```
pix = QPixmap(400,300);  
pix.fill(Qt::white);  
tempPix = pix;  
isDrawing = false;
```

下面是几个鼠标事件处理函数的定义:

```
void Widget::mousePressEvent(QMouseEvent * event)  
{  
    if(event->button() == Qt::LeftButton) {  
        //当鼠标左键按下时获取当前位置作为矩形的开始点  
        startPoint = event->pos();  
        //标记正在绘图
```

```

isDrawing = true;
}

void Widget::mouseMoveEvent(QMouseEvent * event)
{
    if(event->buttons() & Qt::LeftButton) {
        //当按着鼠标左键进行移动时,获取当前位置作为结束点,绘制矩形
        endPoint = event->pos();
        //将缓冲区的内容复制到临时缓冲区,这样进行动态绘制时,
        //每次都是在缓冲区图像的基上进行绘制,就不会产生拖影现象了
        tempPix = pix;
        //更新显示
        update();
    }
}

void Widget::mouseReleaseEvent(QMouseEvent * event)
{
    if(event->button() == Qt::LeftButton) {
        //当鼠标左键松开时,获取当前位置为结束点,完成矩形绘制
        endPoint = event->pos();
        //标记已经结束绘图
        isDrawing = false;
        update();
    }
}

```

这里在鼠标按下事件处理函数中获取了要绘制矩形的左上角的位置,然后标记正在绘制矩形;在鼠标移动事件处理函数中获取了要绘制矩形的右下角的位置,然后动态绘制矩形,这里为了不会绘制出一堆小矩形而产生所谓的拖影现象,就要在绘制临时缓冲区前,将缓冲区的内容复制到临时缓冲区中。这样每次都是在缓冲区图像的基础上进行绘制的,所以不会产生拖影现象。最后在鼠标按键释放事件处理函数中,获取矩形的右下角坐标,标记已经结束绘制。下面是重绘事件处理函数的定义:

```

void Widget::paintEvent(QPaintEvent * event)
{
    int x = startPoint.x();
    int y = startPoint.y();
    int width = endPoint.x() - x;
    int height = endPoint.y() - y;
    QPainter painter;
    painter.begin(&tempPix);
    painter.drawRect(x, y, width, height);
    painter.end();
}

```

```

painter.begin(this);
painter.drawPixmap(0, 0, tempPix);
//如果已经完成了绘制,那么更新缓冲区
if(! isDrawing)
    pix = tempPix;
}

```

这里先在临时缓冲区中进行绘图,然后将其绘制到界面上。最后判断是否已经完成了绘制,如果是,则将临时缓冲区中的内容复制到缓冲区中,这样就完成了整个矩形的绘制。这个例子中的关键是 pix 和 tempPix 的相互复制,如果想将这个程序进行扩展,可以查看一下网站上的涂鸦板程序。

与这个例子很相似的一个应用是橡皮筋线,就是我们在 Windows 桌面上拖动鼠标出现的橡皮筋选择框。Qt 中提供了 QRubberBand 类来实现橡皮筋线,使用它只需要在几个鼠标事件处理函数中进行设置即可,具体应用可以查看该类的帮助文档。

10.5 绘图中的其他问题

10.5.1 重绘事件

前面讲到的所有绘制操作都是在重绘事件处理函数 paintEvent() 中完成的,是 QWidget 类中定义的函数。一个重绘事件用来重绘一个部件的全部或者部分区域,下面几个原因中的任意一个都会发生重绘事件:

- repaint() 函数或者 update() 函数被调用;
- 被隐藏的部件现在被重新显示;
- 其他一些原因。

大部分部件可以简单地重绘它们的全部界面,但是一些绘制比较慢的部件需要进行优化而只绘制需要的区域(可以使用 QPaintEvent::region() 来获取该区域),这种速度上的优化不会影响结果。Qt 也会通过合并多个重绘事件为一个事件来加快绘制,当 update() 函数被调用多次,或者窗口系统发送了多个重绘事件,那么 Qt 就会合并这些事件成为一个事件,而这个事件拥有最大的需要重绘的区域。update() 函数不会立即进行重绘,要等到 Qt 返回主事件循环后才会进行,所以多次调用 update() 函数一般只会引起一次 paintEvent() 函数调用。而调用 repaint() 函数会立即调用 paintEvent() 函数来重绘部件,只有在必须立即进行重绘操作的情况下(比如在动画中),才使用 repaint() 函数。update() 允许 Qt 优化速度和减少闪烁,但是 repaint() 函数不支持这样的优化,所以建议一般情况下尽可能使用 update() 函数。还要说明一下,在程序开始运行时就会自动发送重绘事件而调用 paintEvent() 函数。另外,不要在 paintEvent() 函数中调用 update() 或者 repaint() 函数。

当重绘事件发生时,要更新的区域一般会被擦除,然后在部件的背景上进行绘制。

部件的背景一般可以使用 `setBackgroundRole()` 来指定, 然后使用 `setAutoFillBackground(true)` 来启用指定的颜色。例如使界面显示比较深的颜色, 可以在部件的构造函数中添加如下代码:

```
setBackgroundRole(QPalette::Dark);
setAutoFillBackground(true);
```

10.5.2 剪切

`QPainter` 可以剪切任何的绘制操作, 可以剪切一个矩形、一个区域或者一个路径中的内容, 这分别可以使用 `setClipRect()`、`setClipRegion()` 和 `setClipPath()` 函数来实现。剪切会在 `QPainter` 的逻辑坐标系统中进行。下面的代码实现了剪切一个矩形中的文字:

```
QPainter painter(this);
painter.setClipRect(10, 0, 20, 10);
painter.drawText(10, 10, tr("yafeilinux"));
```

10.5.3 读取和写入图像

要读取图像, 最普通的方法是使用 `QImage` 或者 `QPixmap` 的构造函数, 或者调用 `QImage::load()` 和 `QPixmap::load()` 函数。而在 Qt 中还有一个 `QImageReader` 类, 该类提供了一个格式无关的接口来从文件或者其他设备中读取图像。`QImageReader` 类可以在读取图像时提供更多的控制, 例如, 可以使用 `setScaledSize()` 函数将图像以指定的大小来读取, 还可以使用 `setClipRect()` 来只读取图像的一个区域。依赖于图像格式底层的支持, `QImageReader` 的这些操作可以节省内存和加快图像的读取。另外, Qt 还提供了 `QImageWriter` 类来存储图像, 它支持设置图像格式的特定选项, 比如伽玛等级、压缩等级和品质等。当然, 如果不需要设置这些选项, 那么可以直接使用 `QImage::save()` 和 `QPixmap::save()` 函数。

10.5.4 播放 gif 动画

`QMovie` 类是使用 `QImageReader` 来播放动画的便捷类, 使用它可以播放不带声音的简单的动画, 它支持 gif 和 mng 文件格式。这个类提供了很方便的函数来进行动画的开始、暂停和停止等操作。可以参考该类的帮助文档, 也可以查看一下 Movie Player 示例程序, 它在 Widgets 分类中。

10.5.5 渲染 SVG 文件

可缩放矢量图形(Scalable Vector Graphics, SVG)是一个使用 XML 来描述二维图形和图形应用程序的语言。在 Qt 中可以使用 `QSvgWidget` 类来很容易地加载一个 SVG 文件, 而使用 `QSvgRenderer` 类在 `QSvgWidget` 中进行 SVG 文件的渲染。这两个

类的使用很简单，这里就不再讲述。可以参考一下 SVG Generator 和 SVG Viewer 例程，它们都在 Painting 分类中。

10.6 小结

Qt 的绘图系统是一个复杂、庞大的系统,不过,一些简单的应用还是容易实现的。希望读者能很好地掌握本章的内容,因为这些知识的用途很广泛,而且也是学习下一章的基础。

第 11 章

图形视图、动画和状态机框架

Qt 提供了图形视图框架(Graphics View Framework)、动画框架(The Animation Framework)和状态机框架(The State Machine Framework)来实现更加高级的图形和动画应用。使用这些框架可以快速设计出动态 GUI 应用程序和各种动画、游戏程序。

《Qt 及 Qt Quick 开发实战精解》中的方块游戏实例就是应用本章的知识设计出的一个经典的俄罗斯方块游戏，学习完本章的内容后就可以去完成这个实例，从而进一步熟悉这些知识的应用。

11.1 图形视图框架的结构

在前一章讲的 2D 绘图中已经可以绘制出各种图形，并且进行简单的控制。不过，如果要绘制成千上万的相同或者不同的图形，并且对它们进行控制，比如可以拖动这些图形、检测它们的位置以及判断它们是否相互碰撞等，这样使用以前的方法就很难完成了。这时可以使用 Qt 提供的图形视图框架来进行设计。

图形视图框架提供了一个基于图形项的模型视图编程方法，它主要由场景、视图和图形项 3 部分组成，这 3 部分分别由 QGraphicsScene、QGraphicsView 和 QGraphicsItem 这 3 个类来表示。多个视图可以查看一个场景，场景中包含各种各样几何形状的图形项。图形视图框架在 Qt 4.2 中被引入，用来代替以前的 QCanvas 类组。

图形视图框架可以管理数量庞大的自定义 2D 图形项，并且可以与它们进行交互。使用视图部件可以使这些图形项可视化，视图还支持缩放和旋转。框架中包含了一个事件传播构架，提供了和场景中的图形项进行精确的双精度交互的能力，图形项可以处理键盘事件，鼠标的按下、移动、释放和双击事件，还可以跟踪鼠标的移动。图形视图框架使用一个 BSP(Binary Space Partitioning)树来快速发现图形项，也正是因为如此，它可以实时显示一个巨大的场景，甚至包含上百万个图形项。对应本节的内容，可以在帮助中查看 Graphics View Framework 关键字。

11.1.1 场景

QGraphicsScene 提供了图形视图框架中的场景,场景拥有以下功能:

- 提供了用于管理大量图形项的快速接口;
- 传播事件给每一个图形项;
- 管理图形项的状态,例如选择和焦点处理;
- 提供无变换的渲染功能,主要用于打印。

场景是图形项 QGraphicsItem 对象的容器。可以调用 QGraphicsScene::addItem() 函数将图形项添加到场景中,然后调用众多的图形项发现函数之一来检索添加的图形项。QGraphicsScene::items() 函数和其他几个重载函数可以返回符合条件的所有图形项,这些图形项不是与指定的点、矩形、多边形或者矢量路径相交,就是包含在它们之中。QGraphicsScene::itemAt() 函数返回指定点的最上面的图形项。所有的图形项发现函数返回的图形项都是使用递减顺序(例如第一个返回的图形项在最上面,最后返回的图形项在最下面)。如果要从场景中删除一个图形项,可以使用 QGraphicsScene::removeItem() 函数。下面先来看一个最简单的例子。

(项目源码路径: src\11\11-1\myScene)新建空的 Qt 项目,项目名称为 myScene,完成后向其中添加一个新的 C++ 源文件,名称为 main.cpp。添加完成后在 main.cpp 文件中添加如下代码:

```
#include < QApplication >
#include < QGraphicsScene >
#include < QGraphicsRectItem >
#include < QDebug >

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    //新建场景
    QGraphicsScene scene;
    //创建矩形图形项
    QGraphicsRectItem * item = new QGraphicsRectItem(0, 0, 100, 100);
    //将图形项添加到场景中
    scene.addItem(item);
    //输出(50, 50)点处的图形项
    qDebug() << scene.itemAt(50, 50);
    return app.exec();
}
```

这里先创建了一个场景,然后创建了一个矩形图形项,并且将该图形项添加到了场景中。然后使用 itemAt() 函数来返回指定坐标处最顶层的图形项,这里返回的就是刚才添加的矩形图形项。现在可以运行程序,不过因为还没有设置视图,所以不会出现任

何图形界面。这时可以在应用程序输出栏中看到输出的项目的信息,要关闭运行的程序,可以按下应用程序输出栏上的红色按钮,然后强行关闭应用程序。

QGraphicsScene 的事件传播构架可以将场景事件传递给图形项,也可以管理图形项之间事件的传播。例如,如果场景在一个特定的点接收到了一个鼠标按下事件,那么场景就会将这个事件传递给该点的图形项。

QGraphicsScene 也用来管理图形项的状态,如图形项的选择和焦点等。可以通过向 QGraphicsScene::setSelectionArea() 函数中传递一个任意的形状来选择场景中指定的图形项。如果要获取当前选取的所有图形项的列表,可以使用 QGraphicsScene::selectedItems() 函数。另外可以调用 QGraphicsScene::setFocusItem() 或者 QGraphicsScene::setFocus() 函数来为一个图形项设置焦点,调用 QGraphicsScene::focusItem() 函数来获取当前获得焦点的图形项。

QGraphicsScene 也可以使用 QGraphicsScene::render() 函数将场景中的一部分渲染到一个绘图设备上。这里讲到的这些函数会在后面的内容中看到它们的应用。

11.1.2 视图

QGraphicsView 提供了视图部件,它用来使场景中的内容可视化。可以连接多个视图到同一个场景来为相同的数据集提供多个视口。视图部件是一个可滚动的区域,提供了一个滚动条来浏览大的场景,可以使用 setDragMode() 函数以 QGraphicsView::ScrollHandDrag 为参数来使光标变为手掌形状,从而可以拖动场景。如果设置 setDragMode() 的参数为 QGraphicsView::RubberBandDrag,那么可以在视图上使用鼠标拖出橡皮筋框来选择图形项。默认的 QGraphicsView 提供了一个 QWidget 作为视口部件,如果要使用 OpenGL 进行渲染,可以调用 QGraphicsView::setViewport() 设置 QGLWidget 作为视口。QGraphicsView 会获取视口部件的拥有权(ownership)。

在前面的程序中先添加头文件 #include <QGraphicsView>,然后在主函数中“return app.exec();”一行代码前继续添加如下代码:

```
//为场景创建视图
QGraphicsView view(&scene);
//设置场景的前景色
view.setForegroundBrush(QColor(255, 255, 255, 100));
//设置场景的背景图片
view.setBackgroundBrush(QPixmap("../myScene/background.png"));
view.resize(400, 300);
view.show();
```

这里新建了视图部件,并指定了要可视化的场景。然后为该视图设置了场景前景色和背景图片。一个场景分为 3 层: 图形项层(ItemLayer)、前景层(ForegroundLayer)和背景层(BackgroundLayer)。场景的绘制总是从背景层开始,然后是图形项层,最后是前景层。前景层和背景层都可以使用 QBrush 进行填充,比如使用渐变和贴图等。

这里的前景色设置为半透明的白色,当然也可以设置为其他的填充。这里要提示一下,其实使用好前景色可以实现很多特殊的效果,比如使用半透明的黑色便可以实现夜幕降临的效果。代码中使用了 `QGraphicsView` 类中的函数来设置了场景中的背景和前景,其实也可以使用 `QGraphicsScene` 中的同名函数来实现,不过它们的效果并不完全一样。如果使用 `QGraphicsScene` 对象设置了场景背景或者前景,那么对所有关联了该场景的视图都有效,而 `QGraphicsView` 对象设置的场景的背景或者前景,只对它本身对应的视图有效。可以在里面的代码后面再添加如下代码:

```
QGraphicsView view2(&scene);
view2.resize(400, 300);
view2.show();
```

这时运行程序会出现两个视图,但是第二个视图中的背景是白色的。然后将前面使用 `view` 对象设置背景和前景的代码更改为:

```
scene.setForegroundBrush(QColor(255, 255, 255, 100));
scene.setBackgroundBrush(QPixmap("../myScene/background.png"));
```

这时再运行程序,可以发现两个视图的背景和前景都一样了。当然,使用视图对象来设置场景背景的好处是可以在多个视图中使用不同的背景和前景来实现特定的效果。

最后设置了视图的大小,并调用 `show()` 函数来显示视图。现在前一小节场景中的内容就可以在图形界面中显示出来了,运行程序,效果如图 11-1 所示。可以看到矩形图形项和背景图片都是在视图中间部分进行绘制的,这个问题会在坐标系统部分详细讲解。

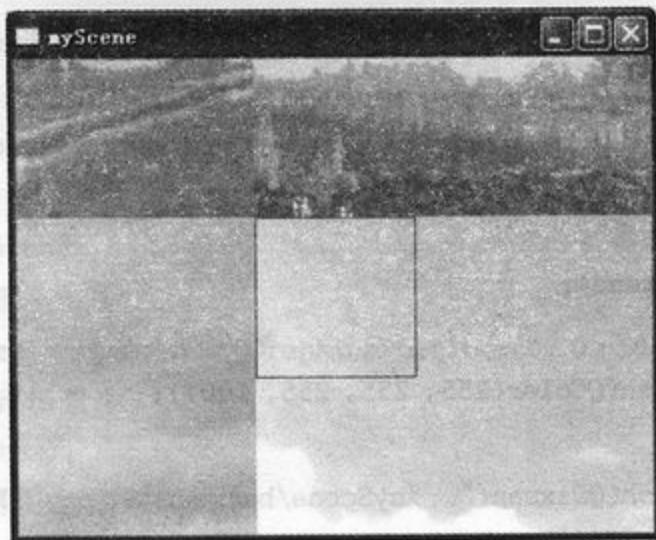


图 11-1 视图运行效果

视图从键盘或者鼠标接收输入事件,然后会在发送这些事件到可视化的场景之前将它们翻译为场景事件(将坐标转换为合适的场景坐标)。另外,使用视图的变换矩阵函数 `QGraphicsView::transform()`,可以使用视图来变换场景的坐标系统,这样便可以实现例如缩放和旋转等高级的导航功能。

11.1.3 图形项

QGraphicsItem 是场景中图形项的基类。图形视图框架为典型的形状提供了标准的图形项，例如矩形（QGraphicsRectItem）、椭圆（QGraphicsEllipseItem）和文本项（QGraphicsTextItem）。不过，只有当编写自定义的图形项时才能发挥 QGraphicsItem 的强大功能。QGraphicsItem 主要支持如下功能：

- 鼠标按下、移动、释放、双击、悬停、滚轮和右键菜单事件；
- 键盘输入焦点和键盘事件；
- 拖放事件；
- 分组，使用 QGraphicsItemGroup 通过 parent-child 关系来实现；
- 碰撞检测。

除此之外，图形项还可以存储自定义的数据，可以使用 setData() 进行数据存储，然后使用 data() 获取其中的数据。下面自定义图形项。（项目源码路径：src\11\11-2\myScene）在前面的程序中添加新文件，模板选择 C++ 类，类名为 MyItem，基类为 QGraphicsItem，类型信息选择“无”。添加完成后，在 myitem.h 文件中添加两个函数的声明：

```
class MyItem : public QGraphicsItem
{
public:
    MyItem();
    QRectF boundingRect() const;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
               QWidget *widget);
};
```

再到 myitem.cpp 文件中添加头文件 #include <QPainter>，然后定义添加的两个函数：

```
QRectF MyItem::boundingRect() const
{
    qreal penWidth = 1;
    return QRectF(0 - penWidth / 2, 0 - penWidth / 2,
                  20 + penWidth, 20 + penWidth);
}

void MyItem::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                   QWidget *widget)
{
    painter->setBrush(Qt::red);
    painter->drawRect(0, 0, 20, 20);
```

要实现自定义的图形项,那么首先要创建一个 `QGraphicsItem` 的子类,然后重新实现它的两个纯虚公共函数: `boundingRect()` 和 `paint()`,前者用来返回要绘制图形项的矩形区域,后者用来执行实际的绘图操作。其中, `boundingRect()` 函数将图形项的外部边界定义为一个矩形,所有的绘图操作都必须限制在图形项的边界矩形之中。而且, `QGraphicsView` 要使用这个矩形来剔除那些不可见的图形项,还要使用它来确定当绘制交叉项目时哪些区域需要进行重新构建。另外, `QGraphicsItem` 的碰撞检测机制也需要使用到这个边界矩形。如果图形绘制了一个轮廓,那么在边界矩形中包含一半画笔的宽度是很重要的,尽管对于抗锯齿绘图并不需要这些补偿。对于绘图函数 `paint()`,它的原型如下:

```
void QGraphicsItem::paint (QPainter * painter, const QStyleOptionGraphicsItem * option, QWidget * widget = 0)
```

这个函数一般会被 `QGraphicsView` 调用,用来在本地坐标中绘制图形项中的内容。其中 `painter` 参数用来进行一般的绘图操作,这与前一章中的绘图操作是一样的; `option` 参数为图形项提供了一个风格选项; `widget` 参数是可选的,如果提供了该参数,那么它会指向那个要在其上进行绘图的部件,否则默认为 0,表明使用缓冲绘图。`painter` 的画笔的宽度默认为 0,它的画笔被初始化为绘图设备调色板的 `QPalette::Text` 画刷,而 `painter` 的画刷被初始化为 `QPalette::Window`。

一定要保证所有的绘图都在 `boundingRect()` 的边界之中。特别是当 `QPainter` 使用了指定的 `QPen` 来渲染图形的边界轮廓时,绘制的图形的边界线的一半会在外面,一半会在里面(例如,使用了宽度为两个单位的画笔,就必须在 `boundingRect()` 里绘制一个单位的边界线)。这也是在 `boundingRect()` 中要包含半个画笔宽度的原因。`QGraphicsItem` 不支持使用宽度非零的装饰笔。

下面来使用自定义的图形项。在 `main.cpp` 文件中先添加头文件 `#include "myitem.h"`,然后将以前的图形项的创建代码:

```
QGraphicsRectItem * item = new QGraphicsRectItem(0, 0, 100, 100);
```

更改为:

```
MyItem * item = new MyItem;
```

这时运行程序,效果如图 11-2 所示。可以看到,自定义的红色小方块出现在了视图的正中间,背景图片的位置也有所变化,这些问题都会在后面的坐标系统中讲到。如果只想添加简单的图形项,那么也可以直接使用图形视图框架提供的 8 种标准图形项,它们的效果如图 11-3 所示。

视图从键盘或者鼠标接收输入事件,然后会将这些事件映射到可视化的场景中,将它们翻译为场景事件(将坐标转换为合适的场景坐标)。另外,使用视图的变换功能。`QGraphicsView::transform()` 可以使用高得非常精确的坐标系统,从而可以实现例如缩放和旋转等高级的导航功能。

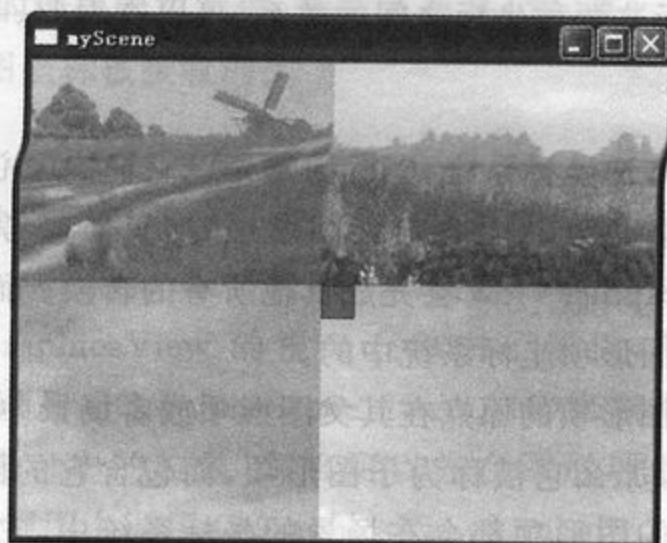


图 11-2 自定义图形项运行效果

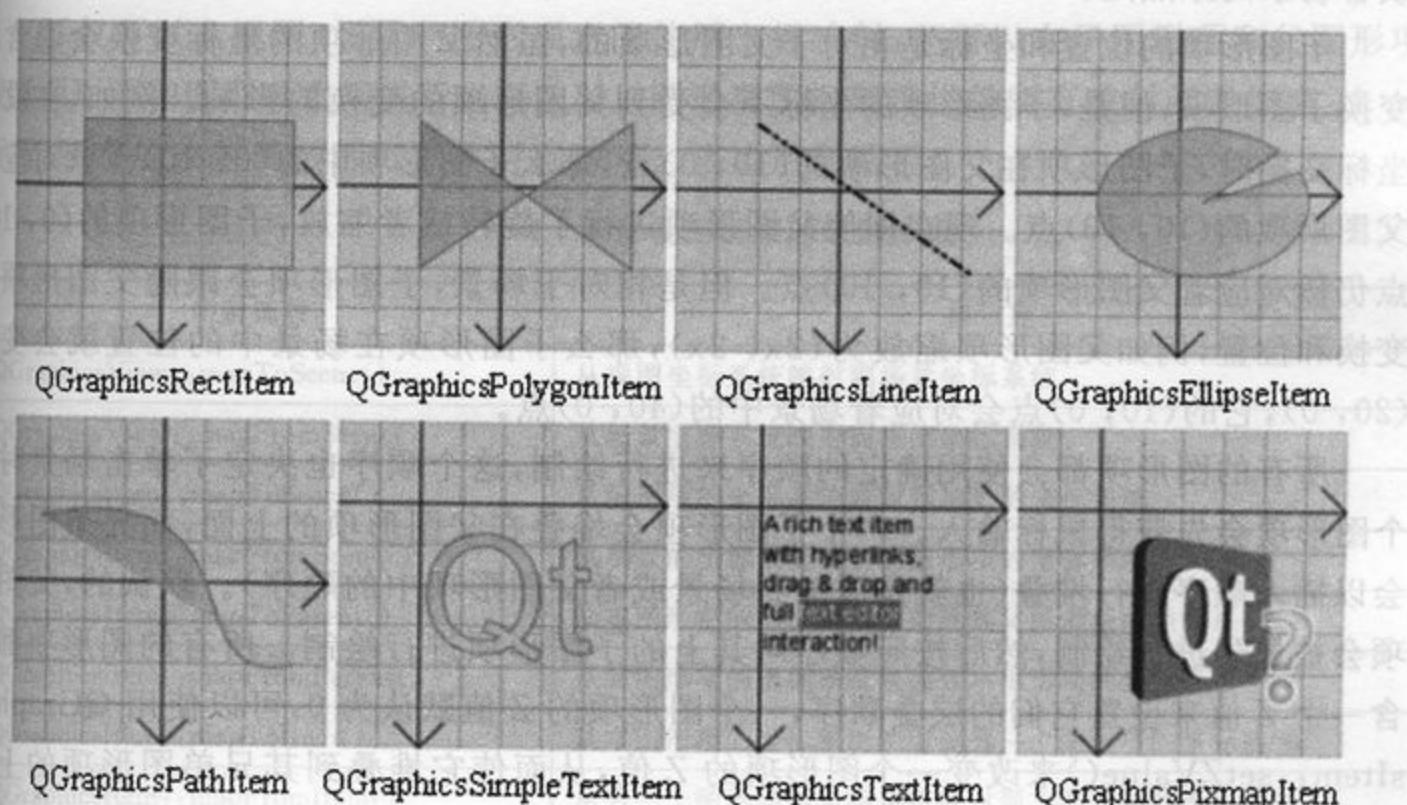


图 11-3 图形视图框架提供的标准图形项

11.2 图形视图框架的坐标系统和事件处理

11.2.1 坐标系统

图形视图框架是基于笛卡尔坐标系统的，一个图形项在场景中的位置和几何形状由 x 坐标和 y 坐标来表示。当使用一个没有变换的视图来观察场景时，场景中的一个单元代表屏幕上的一个像素。在图形视图框架中有 3 个有效的坐标系统：图形项坐标、场景坐标和视图坐标。为了方便应用，图形视图框架中提供了一些便捷函数来完成 3 个坐标系统之间的映射。当进行绘图时，场景坐标对应 QPainter 的逻辑坐标，视图

坐标对应设备坐标。关于这两种坐标之间的关系,可以查看前一章的坐标系统部分。

1. 图形项坐标

图形项使用自己的本地坐标系统,坐标通常是以它们的中心为原点(0, 0),而这也是所有变换的中心。当要创建一个自定义图形项时,只需要考虑图形项的坐标系统,QGraphicsScene 和 QGraphicsView 会完成其他所有的转换。而且,一个图形项的边界矩形和图形形状都是在图形项坐标系统中的。

图形项的位置是指图形项的原点在其父图形项或者场景中的位置。如果一个图形项在另一个图形项之中,那么它被称为子图形项,而包含它的图形项称为它的父图形项。所有没有父图形项的图形项都会在场景的坐标系统中,它们被称为顶层图形项,可以使用 setPos() 函数来指定图形项的位置,如果没有指定,它默认会出现在父图形项或者场景的原点处。

子图形项的位置和坐标是相对于父图形项的,虽然父图形项的坐标变换会隐含的变换子图形项,但是,子图形项的坐标不会受到父图形项的变换的影响。例如,在没有坐标变换时,子图形项在父图形项的(10, 0)点,那么子图形项中的(0, 10)点就对应了父图形项的(10, 10)点。现在即使父图形项进行了旋转或者缩放,子图形项的(0, 10)点仍然对应着父图形项的(10, 10)点。但是相对于场景,子图形项会跟随父图形项的变换和位置,例如父图形项缩放为(2x, 2x),那么子图形项在场景中的位置就会变为(20, 0),它的(10, 0)点会对应着场景中的(40, 0)点。

所有的图形项都会使用确定的顺序来进行绘制,这个顺序也决定了单击场景时哪个图形项会先获得鼠标输入。一个子图形项会堆叠在父图形项的上面,而兄弟图形项会以插入顺序进行堆叠(也就是添加到场景或者父图形项中的顺序)。默认的,父图形项会被最先进行绘制,然后按照顺序对其上的子图形项进行绘制。所有的图形项都包含一个 Z 值来设置它们的层叠顺序,一个图形项的 Z 值默认为 0,可以使用 QGraphicsItem::setZValue() 来改变一个图形项的 Z 值,从而使它堆叠到其兄弟图形项的上面(使用较大的 Z 值时)或者下面(使用较小的 Z 值时)。

2. 场景坐标

场景坐标是所有图形项的基础坐标系统。场景坐标系统描述了每一个顶层图形项的位置,也形成了所有从视图传到场景上的事件的基础。场景坐标的原点在场景的中心,x 和 y 坐标分别向右和向下增大。每一个在场景中的图形项除了拥有一个图形项的本地坐标和边界矩形外,还都拥有一个场景坐标(QGraphicsItem::scenePos())和一个场景中的边界矩形(QGraphicsItem::sceneBoundingRect())。场景坐标用来描述图形项在场景坐标系统中的位置,而图形项的场景边界矩形构成了 QGraphicsScene 怎样来判断场景中的哪些区域被更改了的基础。

3. 视图坐标

视图的坐标就是部件的坐标。视图坐标的每一个单位对应一个像素,原点(0, 0)

总在 QGraphicsView 的视口的左上角,而右下角是(宽,高)。所有的鼠标事件和拖放事件最初都是使用视图坐标被接收的。

4. 坐标映射

当处理场景中的图形项时,将坐标或者一个任意的形状从场景映射到图形项、或者从一个图形项映射到另一个图形项、或者从视图映射到场景,这些坐标变换都是非常有用的。例如,当在 QGraphicsView 的视口上单击了鼠标,便可以调用 QGraphicsView::mapToScene()以及 QGraphicsScene::itemAt()来获取光标下的图形项;如果要获取一个图形项在视口中的位置,那么可以先在图形项上调用 QGraphicsItem::mapToScene(),然后在视图上调用 QGraphicsView::mapFromScene();如果要获取在视图的一个椭圆形中包含的图形项,可以先传递一个 QPainterPath 对象作为参数给 mapToScene()函数,然后传递映射后的路径给 QGraphicsScene::items()函数。

不仅可以在视图、场景和图形项之间使用坐标映射,还可以在子图形项和父图形项或者图形项和图形项之间进行坐标映射。图形视图框架提供的所有映射函数如表 11-1 所列,所有的映射函数都可以映射点、矩形、多边形和路径。

表 11-1 图形视图框架的映射函数

映射函数	描述
QGraphicsView::mapToScene()	从视图坐标系统映射到场景坐标系统
QGraphicsView::mapFromScene()	从场景坐标系统映射到视图坐标系统
QGraphicsItem::mapToScene()	从图形项的坐标系统映射到场景的坐标系统
QGraphicsItem::mapFromScene()	从场景的坐标系统映射到图形项的坐标系统
QGraphicsItem::mapToParent()	从本图形项的坐标系统映射到其父图形项的坐标系统
QGraphicsItem::mapFromParent()	从父图形项的坐标系统映射到本图形项的坐标系统
QGraphicsItem::mapToItem()	从本图形项的坐标系统映射到另一个图形项的坐标系统
QGraphicsItem::mapFromItem()	从另一个图形项的坐标系统映射到本图形项的坐标系统

下面通过例子来进一步学习图形视图框架的坐标系统。(项目源码路径: src\11\11-3\myScene)在前面的程序中添加新文件,模板选择 C++ 类,类名为 MyView,基类为 QGraphicsView,类型信息选择“继承自 QWidget”。完成后在 myview.h 文件中添加函数声明:

```
protected:  
void mousePressEvent(QMouseEvent * event);
```

然后到 myview.cpp 文件中,添加头文件:

```
# include <QMouseEvent>  
# include <QGraphicsItem>  
# include <QDebug>
```

然后是鼠标按下事件处理函数的定义：

```
void MyView::mousePressEvent(QMouseEvent * event)
{
    //分别获取鼠标点击处在视图、场景和图形项中的坐标，并输出
    QPoint viewPos = event->pos();
    qDebug() << "viewPos: " << viewPos;
    QPointF scenePos = mapToScene(viewPos);
    qDebug() << "scenePos: " << scenePos;
    QGraphicsItem * item = scene()->itemAt(scenePos);
    if (item) {
        QPointF itemPos = item->mapFromScene(scenePos);
        qDebug() << "itemPos: " << itemPos;
    }
}
```

这里先使用鼠标事件对象 `event` 来获取了鼠标单击位置在视图中的坐标, 然后使用映射函数将这个坐标转换为了场景中的坐标, 并使用 `scene()` 函数获取视图当前的场景的指针, 然后使用 `QGraphicsScene::itemAt()` 函数获取了场景中该坐标处的图形项, 如果这里有图形项, 那么便输出该点在图形项坐标系统中的坐标。

下面到 `main.cpp` 文件中, 先添加头文件 `#include "myview.h"`, 然后更改主函数的内容为：

```
int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QGraphicsScene scene;
    MyItem * item = new MyItem;
    scene.addItem(item);
    item->setPos(10, 10);
    QGraphicsRectItem * rectItem = scene.addRect(QRect(0, 0, 100, 100),
                                                QPen(Qt::blue), QBrush(Qt::green));
    rectItem->setPos(20, 20);
    MyView view;
    view.setScene(&scene);
    view.setForegroundBrush(QColor(255, 255, 255, 100));
    view.setBackgroundBrush(QPixmap("../myScene/background.png"));
    view.resize(400, 300);
    view.show();
    return app.exec();
}
```

这里先向场景中添加一个 `MyItem` 图形项, 然后又使用 `QGraphicsScene::addRect()` 函数添加了矩形图形项, 并分别设置了它们在场景中的位置。然后使用自定义的视图

类 MyView 创建了视图。因为 item 在 rectItem 之前添加到场景中，所以 rectItem 会在 item 之上进行绘制。现在在视图上单击，然后查看应用程序输出栏中的输出数据，分别点击视图的左上角，背景图片的交点处，小正方形的左上角以及大正方形的左上角。通过数据可以发现，视图的左上角是视图的原点，背景图片的交点处是场景的原点，而两个正方形的左上角分别是它们图形项坐标的原点。如果想将 item 移动到 rectItem 之上，那么可以在创建 item 的代码之后添加如下一行代码：

```
item -> setZValue(1);
```

这样就可以让 item 显示在 rectItem 之上了。其实还可以将 item 作为 rectItem 的子图形项，这样 item 就会在 rectItem 的坐标系统上进行绘制，也就是不用使用 setZValue() 函数，item 也是默认的显示在 rectItem 之上的。先注释掉添加的 setZValue() 的代码，然后在创建 rectItem 的代码的后面添加如下代码：

```
item -> setParentItem(rectItem);
rectItem -> rotate(45);
```

这里将 rectItem 设置为了 item 的父项，然后将 rectItem 进行了旋转。可以看到，rectItem 会在自己的坐标系统中进行旋转，并且是以原点为中心进行旋转的。虽然 item 也进行了旋转，但是它在 rectItem 中的相对位置却没有改变。可以在视图上单击来查看一下输出的坐标信息。

下面再来看为什么场景背景图片会随着图形项的不同而改变位置？其实场景背景图片的位置变化也就是场景的位置的变化，默认的，如果场景中没有添加任何图形项，那么场景的中心（默认的是原点）会和视图的中心重合。如果添加了图形项，那么视图就会以图形项的中心为中心来显示场景。所以就像前面看到的，图形项的大小或者位置变化了，视口的位置也就变化了，这样看起来好像是背景图片的位置发生了变化。其实，场景还有一个很重要的属性就是场景矩形，它是场景的边界矩形。场景矩形定义了场景的范围，主要用于 QGraphicsView 来判断视图默认的滚动区域，当视图小于场景矩形时，就会自动生成水平和垂直的滚动条来显示更大的区域。另外场景矩形也用于 QGraphicsScene 来管理图形项索引。可以使用 QGraphicsScene::setSceneRect() 来设置场景矩形，如果没有设置，那么 sceneRect() 会返回一个包含了自从场景创建以来添加的所有图形项的最大边界矩形（这个矩形会随着图形项的添加或者移动而不断增长，但是永远不会缩小），所以操作一个较大的场景时，总应该设置一个场景矩形。

设置了场景矩形，就可以指定视图显示的场景区域了。比如将场景的原点显示在视图的左上角，那么可以在创建场景的代码下面添加如下一行代码：

```
scene.setSceneRect(0, 0, 400, 300);
```

如果当场景很大时，还可以使用 QGraphicsView 类中的 centerOn() 函数来设置场景中的一个点或者一个图形项作为视图的显示中心。

11.2.2 事件处理与传播

图形视图框架中的事件都是首先由视图进行接收,然后传递给场景,再由场景传递给相应的图形项。而对于键盘事件,它会传递给获得焦点的图形项,可以使用 QGraphicsScene 类的 setFocusItem() 函数或者图形项自身调用 setFocus() 函数来将其设置为焦点图形项。默认的,如果场景没有获得焦点,那么所有的键盘事件都会被丢弃。如果调用了场景的 setFocus() 函数或者场景中的一个图形项获得了焦点,那么场景也会自动获得焦点。如果场景丢失了焦点(例如调用了 clearFocus() 函数),然而它的一个图形项获得有焦点,那么场景就会保存这个图形项的焦点信息,当场景重新获得焦点后,就会确保最后一个焦点项目重新获得焦点。

对于鼠标悬停效果,QGraphicsScene 会调度悬停事件。如果一个图形项可以接收悬停事件,那么当鼠标进入它的区域之中时,它就会收到一个 GraphicsSceneHoverEnter 事件。如果鼠标继续在图形项的区域之中进行移动,那么 QGraphicsScene 就会向该图形项发送 GraphicsSceneHoverMove 事件。当鼠标离开图形项的区域时,它将会收到一个 GraphicsSceneHoverLeave 事件。图形项默认是无法接收悬停事件的,可以使用 QGraphicsItem 类的 setAcceptHoverEvents() 函数使图形项可以接收悬停事件。

所有的鼠标事件都会传递到当前鼠标抓取的图形项,一个图形项如果可以接收鼠标事件(默认可以)而且鼠标在它的上面被按下,那么它就会成为场景的鼠标抓取的图形项。

下面来看一个例子。(项目源码路径: src\11\11-4\myView)新建空的 Qt 项目,名称为 myView。完成后先按照源码路径为 11-2 那样向本项目中添加一个 MyItem 自定义图形项,然后在 myitem.h 文件中声明 boundingRect() 和 paint() 两个纯虚函数,再声明并定义一个公共的用于设置图形项填充色的函数:

```
void setColor(const QColor &color) { brushColor = color; }
```

然后定义私有变量 brushColor,用于保存图形项的填充色:

```
private:  
    QColor brushColor;
```

下面到 myitem.cpp 文件中,先添加头文件 #include <QPainter>,然后在构造函数中初始化变量:

```
MyItem::MyItem()  
{  
    brushColor = Qt::red;  
}
```

这样图形项默认的填充色就是红色了。下面是那两个纯虚函数的定义:

```

QRectF MyItem::boundingRect() const
{
    qreal adjust = 0.5;
    return QRectF(-10 - adjust, -10 - adjust,
                  20 + adjust, 20 + adjust);
}

void MyItem::paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
                   QWidget * widget)
{
    if (hasFocus()) {
        painter->setPen(QPen(QColor(255, 255, 255, 200)));
    } else {
        painter->setPen(QPen(QColor(100, 100, 100, 100)));
    }
    painter->setBrush(brushColor);
    painter->drawRect(-10, -10, 20, 20);
}

```

在这里根据图形项是否获得焦点 hasFocus() 来使用不同颜色绘制图形项的轮廓。这个会在后面使用到。而使用了变量作为画刷的颜色，就可以动态指定图形项的填充色了。

完成了自定义图形项类的添加后，再按照源码路径为 11-3 那样添加一个 MyView 自定义视图类。添加完成后，先在 myview.h 中添加键盘按下事件处理函数的声明：

```

protected:
    void keyPressEvent(QKeyEvent * event);

```

然后到 myview.cpp 文件中添加头文件 #include <QKeyEvent>，然后定义函数：

```

void MyView::keyPressEvent(QKeyEvent * event)
{
    switch (event->key())
    {
        case Qt::Key_Plus :
            scale(1.2, 1.2);
            break;
        case Qt::Key_Minus :
            scale(1 / 1.2, 1 / 1.2);
            break;
        case Qt::Key_Right :
            rotate(30);
            break;
    }
}

```

```
QGraphicsView::keyPressEvent(event);
```

}

这里使用不同的按键来实现视图的缩放和旋转等操作。一定要注意，在视图的事件处理函数的最后一一定要调用 QGraphicsView 类的 keyPressEvent() 函数，不然在场景或者图形项中就无法再接收到该事件了。

最后添加 main.cpp 文件，并且将其内容更改如下：

```
#include < QApplication >
#include "myitem.h"
#include "myview.h"
#include < QTime >

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
    QGraphicsScene scene;
    scene.setSceneRect(-200, -150, 400, 300);
    for (int i = 0; i < 5; ++i) {
        MyItem * item = new MyItem;
        item->setColor(QColor(qrand() % 256, qrand() % 256, qrand() % 256));
        item->setPos(i * 50 - 90, -50);
        scene.addItem(item);
    }
    MyView view;
    view.setScene(&scene);
    view.setBackgroundBrush(QPixmap("../myView/background.png"));
    view.show();
    return app.exec();
}
```

这里在场景中添加了 5 个图形项，分别为它们设置了随机颜色，然后将它们排成一行。可以使用键盘上的“+”和“-”键来放大和缩小视图，也可以使用向右方向键“→”来旋转视图。

再来看一下其他事件的应用。先在 myitem.h 文件中添加一些事件处理函数的声明：

```
protected:
void keyPressEvent(QKeyEvent * event);
void mousePressEvent(QGraphicsSceneMouseEvent * event);
void hoverEnterEvent(QGraphicsSceneHoverEvent * event);
void contextMenuEvent(QGraphicsSceneContextMenuEvent * event);
```

然后到 myitem.cpp 文件中添加头文件：

```
#include <QCursor>
#include <QKeyEvent>
#include <QGraphicsSceneHoverEvent>
#include <QGraphicsSceneContextMenuEvent>
#include <QMenu>
```

再在 MyItem 构造函数中添加如下代码：

```
setFlag(QGraphicsItem::ItemIsFocusable);
setFlag(QGraphicsItem::ItemIsMovable);
setAcceptHoverEvents(true);
```

这里的 setFlag() 函数可以开启图形项的一些特殊功能,比如要想使用键盘控制图形项,则必须使图形项可以获得焦点,所以要先设置 ItemIsFocusable 标志;而如果想使用鼠标来拖动图形项进行移动,那么就必须先设置 ItemIsMovable 标志。这些标志也可以在创建图形项时进行设置,更多的标志可以参见 QGraphicsItem 类的帮助文档。为了使图形项支持悬停事件,需要调用 setAcceptHoverEvents(true) 来进行设置。下面是事件处理函数的定义:

```
//鼠标按下事件处理函数,设置被点击的图形项获得焦点,并改变光标外观
void MyItem::mousePressEvent(QGraphicsSceneMouseEvent * event)
{
    setFocus();
    setCursor(Qt::ClosedHandCursor);
}

//键盘按下事件处理函数,判断是否是向下方向键,如果是,则向下移动图形项
void MyItem::keyPressEvent(QKeyEvent * event)
{
    if (event->key() == Qt::Key_Down)
        moveBy(0, 10);
}

//悬停事件处理函数,设置光标外观和提示
void MyItem::hoverEnterEvent(QGraphicsSceneHoverEvent * event)
{
    setCursor(Qt::OpenHandCursor);
    setToolTip("I am item");
}

//右键菜单事件处理函数,为图形项添加一个右键菜单
void MyItem::contextMenuEvent(QGraphicsSceneContextMenuEvent * event)
{
    QMenu menu;
    QAction * moveAction = menu.addAction("move back");
    QAction * selectedAction = menu.exec(event->screenPos());
```

```

if (selectedAction == moveAction) {
    setPos(0, 0);
}
}

```

这里首先在鼠标按下事件处理函数中,为鼠标单击的图形项设置了焦点,这样当按下键盘时该图形项就会接收到按键事件。如果按下了键盘的向下方向键,那么获得焦点的图形项就会向下移动,这里使用了 moveBy(qreal dx, qreal dy) 函数,它用来进行相对移动,就是相对于当前位置在水平方向移动 dx,在垂直方向移动 dy。在进行项目移动时,经常使用到该函数。然后是右键菜单事件,在一个图形项上右击,就会弹出一个菜单,如果选中该菜单,那么图形项会移动到场景原点。现在运行程序,将鼠标光标移动到一个图形项上,可以看到光标外观改变了,而且出现了工具提示。可以使用鼠标拖动图形项,在一个图形项上右击,还可以看到弹出的右键菜单;也可以选中一个图形项,然后使用键盘来移动它。

图形视图框架还可以处理一些其他的事件,比如拖放事件,这个可以参考第 6 章的相关知识来学习,也可以参考一下 Drag and Drop Robot 示例程序,它在 Graphics View 分类中,这个分类中还有一个 Diagram Scene 示例程序,是一个使用图形视图框架设计的绘图程序,也可以参考。

11.3 图形视图框架的其他特性

11.3.1 图形效果

图形效果(graphics effect)是 Qt 4.6 添加的一个新的特色功能,QGraphicsEffect 类是所有图形效果的基类。使用图形效果来改变元素的外观是通过在源对象(如一个图形项)和目标设备(如视图的视口)之间挂接了渲染管道和一些操作来实现的。图形效果可以实施在任何一个图形项或者非顶层窗口的任何一个窗口部件上,只需先创建一个图形效果对象,然后调用 setGraphicsEffect() 函数来使用这个图形效果即可。如果想停止使用该效果,那么可以调用 setEnabled(false)。Qt 提供了 4 种标准的效果,如表 11-2 所列。也可以自定义效果,这需要创建 QGraphicsEffect 的子类,可以查看该类的帮助文档来了解更多的相关内容。

表 11-2 Qt 标准图形效果

图形效果类	介绍
QGraphicsBlurEffect	该类提供了一个模糊效果,该效果一般用来减少源对象细节的显示。可以使用 setBlurRadius() 函数来修改细节等级,默认的模糊半径是 5 像素;还可以使用 setBlurHints() 来指定模糊怎样来执行

续表 11-2

图形效果类	介绍
QGraphicsColorizeEffect	该类提供了一个染色效果,该效果用来为源对象进行染色。可以使用 setColor()函数修改颜色,默认是浅蓝色 QColor(0, 0, 192);还可以使用 setStrength()来修改效果的强度,强度在 0.0~1.0 之间,默认为 1.0
QGraphicsDropShadowEffect	该类提供了一个阴影效果,该效果可以为源对象提供一个阴影。可以使用 setColor()来修改阴影的颜色,默认是透明的黑灰色 QColor(63, 63, 63, 180);可以使用 setOffset()来改变阴影的偏移值,默认为右下方 8 像素;还可以使用 setBlurRadius()来改变阴影的模糊半径,其默认值为 1
QGraphicsOpacityEffect	该类提供了一个透明效果,该效果可以使源对象透明。可以使用 setOpacity()函数来修改透明度,其值在 0.0~1.0 之间,0.0 表示完全透明,1.0 表示完全不透明

(项目源码路径: src\11\11-5\myView)继续在前面程序的基础上进行更改,先在 myitem.cpp 文件中添加头文件 #include <QGraphicsEffect>,然后更改 keyPressEvent() 函数如下:

```

void MyItem::keyPressEvent(QKeyEvent * event)
{
    switch (event->key())
    {
        case Qt::Key_1 :
            QGraphicsBlurEffect * blurEffect = new QGraphicsBlurEffect;
            blurEffect ->setBlurHints(QGraphicsBlurEffect::QualityHint);
            blurEffect ->setBlurRadius(8);
            setGraphicsEffect(blurEffect);
            break;
    }
    case Qt::Key_2 :
        QGraphicsColorizeEffect * colorizeEffect = new QGraphicsColorizeEffect;
        colorizeEffect ->setColor(Qt::white);
        colorizeEffect ->setStrength(0.6);
        setGraphicsEffect(colorizeEffect);
        break;
    }
    case Qt::Key_3 :
        QGraphicsDropShadowEffect * dropShadowEffect = new QGraphicsDropShadowEffect;
        dropShadowEffect ->setColor(QColor(63, 63, 63, 100));
        dropShadowEffect ->setBlurRadius(2);
        dropShadowEffect ->setOffset(10);
        setGraphicsEffect(dropShadowEffect);
    }
}

```

```

    break;
}

case Qt::Key_4 :
    QGraphicsOpacityEffect * opacityEffect = new QGraphicsOpacityEffect;
    opacityEffect ->setOpacity(0.4);
    setGraphicsEffect(opacityEffect);
    break;
}

case Qt::Key_5 :
    graphicsEffect() ->setEnabled(false);
    break;
}
}

```

这里分别使用不同的按键来实现了不同的图形效果,现在运行程序,然后分别选中一个图形项设置为不同的图形效果。数字键 5 可以取消图形项的图形效果。

11.3.2 动画、碰撞检测和图形项组

1. 动 画

图形视图框架支持几种级别的动画。以前可以使用 `QGraphicsItemAnimation` 类很容易地实现图形项的动画效果,不过该类现在已经过时,不再讲解。现在主要是通过动画框架来实现动画效果。另外的方法是创建一个继承自 `QObject` 和 `QGraphicsItem` 的自定义图形项,然后创建它自己的定时器来实现动画,这个这里也不再讲解。第三种方法是使用 `QGraphicsScene::advance()` 来推进场景,下面我们来看一下它的应用。

(项目源码路径: `src\11\11-6\myView`)继续在前面程序的基础上进行修改。首先在 `myitem.h` 文件中的 `public` 部分添加函数声明:

```
void advance(int phase);
```

然后到 `myitem.cpp` 文件中进行该函数的定义:

```

void MyItem::advance(int phase)
{
    //在第一个阶段不进行处理
    if (!phase)
        return;

    //图形项向不同方向随机移动
    int value = qrand() % 100;
    if (value < 25) {
        rotate(45);
        moveBy(qrand() % 10, qrand() % 10);
    }
}

```

```

} else if (value < 50) {
    rotate(-45);
    moveBy(-qrand() % 10, -qrand() % 10);
} else if (value < 75) {
    rotate(30);
    moveBy(-qrand() % 10, qrand() % 10);
} else {
    rotate(-30);
    moveBy(qrand() % 10, -qrand() % 10);
}
}

```

调用场景的 advance() 函数就会自动调用场景中所有图形项的 advance() 函数, 而且图形项的 advance() 函数会被分为两个阶段调用两次。第一次 phase 为 0, 告知所有的图形项场景将要改变; 第二次 phase 为 1, 在这时才进行具体的操作, 这里就是让图形项在不同的方向上移动一个随机的数值。下面到 main.cpp 文件中, 先添加头文件 #include <QTimer>, 然后在主函数的最后 return 语句前添加如下代码:

```

QTimer timer;
QObject::connect(&timer, SIGNAL(timeout()), &scene, SLOT(advance()));
timer.start(300);

```

这里创建了一个定时器, 当定时器溢出时会调用场景的 advance() 函数。

2. 碰撞检测

图形视图框架提供了图形项之间的碰撞检测, 碰撞检测可以使用两种方法来实现:

① 重新实现 QGraphicsItem::shape() 函数来返回图形项的准确的形状, 然后使用默认的 collidesWithItem() 函数通过两个图形项形状之间的交集来判断是否发生碰撞。如果图形项的形状很复杂, 那么进行这个操作是非常耗时的。如果没有重新实现 shape() 函数, 那么它默认会调用 boundingRect() 函数返回一个简单的矩形。

② 重新实现 collidesWithItem() 函数来提供一个自定义的图形项碰撞算法。

可以使用 QGraphicsItem 类中的 collidesWithItem() 函数来判断是否与指定的图形项进行了碰撞; 使用 collidesWithPath() 来判断是否与指定的路径碰撞; 使用 collidingItems() 来获取与该图形项碰撞的所有图形项的列表; 也可以调用 QGraphicsScene 类的 collidingItems()。这几个函数都有一个 Qt::ItemSelectionMode 参数来指定怎样进行图形项的选取, 它一共有 4 个值, 如表 11-3 所列, 其中 Qt::IntersectsItemShape 是默认值。

表 11-3 图形项选取模式

常量	描述
Qt::ContainsItemShape	选取只有形状完全包含在选择区域之中的图形项
Qt::IntersectsItemShape	选取形状完全包含在选择区域之中或者与区域的边界相交的图形项
Qt::ContainsItemBoundingRect	选取只有边界矩形完全包含在选择区域之中的图形项
Qt::IntersectsItemBoundingRect	选取边界矩形完全包含在选择区域之中或者与区域的边界相交的图形项

下面继续在前面的程序中添加代码。首先在 myitem.h 文件的 public 部分进行函数声明：

```
QPainterPath shape();
```

然后到 myitem.cpp 文件中定义该函数：

```
QPainterPath MyItem::shape()
{
    QPainterPath path;
    path.addRect(-10, -10, 20, 20);
    return path;
}
```

这里只是简单地返回了图形项对应的矩形，然后将 paint() 函数中以前用来判断是否获得焦点的 if 语句的判断条件更改如下：

```
if(hasFocus() || !collidingItems().isEmpty())
```

这样就可以在图形项与其他图形项碰撞时使其轮廓线变为白色了。

关于 advance() 函数和碰撞检测的使用，可以参考 Graphics View 分类中的 Colliding Mice 示例程序。

3. 图形项组

QGraphicsItemGroup 图形项组为图形项提供了一个容器，它可以将多个图形项组合在一起而将它本身以及它所有的子图形项看作一个独立的图形项。与父图形项不同，图形项组中的所有图形项都是平等的，例如可以通过拖动其中任意一个来将它们一起进行移动。而如果只想将一个图形项存储在另一个图形项之中，那么可以使用 setParentItem() 来为其设置父图形项。下面仍然在前面的程序中添加代码。在 main() 函数的 return 语句前添加如下代码：

```
MyItem * item1 = new MyItem;
item1->setColor(Qt::blue);
MyItem * item2 = new MyItem;
item2->setColor(Qt::green);
QGraphicsItemGroup * group = new QGraphicsItemGroup;
```

```

group -> addToGroup(item1);
group -> addToGroup(item2);
group -> setFlag(QGraphicsItem::ItemIsMovable);
item2 -> setPos(30, 0);
scene.addItem(group);

```

这里创建了两个图形项和一个图形项组，然后将两个图形项加入到图形项组中。这样只需要将图形项组添加到场景中，那么两个图形项也就自动添加到场景中了。运行程序，可以通过鼠标拖动其中一个图形项来一起移动两个图形项。除了手动创建图形项组，常用的方法还有使用场景对象直接创建图形项组，并将指定的图形项添加到其中，例如：

```
QGraphicsItemGroup * group = scene -> createItemGroup(scene -> selectedItems());
```

这个一般用来选取场景中的图形项，可以让 QGraphicsView 类的对象通过调用 setDragMode(QGraphicsView::RubberBandDrag) 函数来使鼠标可以在视图上拖出橡皮筋框来选择图形项，注意如果要使图形项可以被选择，还要使用 setFlag() 指定它们的 ItemIsSelectable 标志。如果要从图形项组中删除一个图形项，可以调用 removeFromGroup() 函数，还可以调用 QGraphicsScene::destroyItemGroup() 来销毁整个图形项组，这两个函数都不会销毁图形项组中的图形项，而会将它们移动到父图形项组或者场景中。

11.3.3 打印和使用 OpenGL 进行渲染

1. 打印

图形视图框架提供渲染函数 QGraphicsScene::render() 和 QGraphicsView::render() 来完成打印功能。这两个函数提供了相同的 API，可以在绘图设备上绘制场景或者视图的全部或者部分内容。两者的不同之处就是一个在场景坐标上进行操作而另一个在视图坐标上。QGraphicsScene::render() 经常用来打印没有变换的场景，比如几何数据和文本文档等；而 QGraphicsView::render() 函数适合用来实现屏幕快照。要在打印机上进行打印，可以使用如下代码：

```

QPrinter printer;
if (QPrintDialog(&printer).exec() == QDialog::Accepted) {
    QPainter painter(&printer);
    painter.setRenderHint(QPainter::Antialiasing);
    scene.render(&painter);
}

```

下面来实现屏幕快照功能。（项目源码路径：src\11\11-7\myView）继续在前面程序的基础上添加代码。首先在 main.cpp 文件中添加头文件：

```
#include <QPainter>
```

```
#include <QPixmap>
```

然后在 main() 函数的最后 return 语句之前添加如下代码：

```
QPixmap pixmap(400, 300);
QPainter painter(&pixmap);
painter.setRenderHint(QPainter::Antialiasing);
view.render(&painter);
painter.end();
pixmap.save("view.png");
```

此时运行程序，可以在项目目录的 build - desktop 子目录中发现生成的 view.png 图片。

2. 使用 OpenGL 进行渲染

使用 OpenGL 进行渲染，可以使用 QGraphicsView::setViewport() 来将 QGLWidget 作为 QGraphicsView 的视口。如果想 OpenGL 进行抗锯齿，可以使用采样缓冲区 (sample buffer)。

继续在前面的程序中添加代码。首先在项目文件 myView.pro 中添加如下一行代码：

```
QT += opengl
```

这样在程序中才可以使用 OpenGL 相关的类。然后到 main.cpp 文件中添加头文件 #include <QGLWidget>，并在 main() 主函数中创建 MyView 对象的代码后添加如下一行代码：

```
view.setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers)));
```

这样就可以使用 OpenGL 进行渲染了。对于 OpenGL 可以查看相关资料。

11.3.4 窗口部件、布局和内嵌部件

从 Qt 4.4 开始通过 QGraphicsWidget 类引入了支持几何和布局的图形项。图形部件 QGraphicsWidget 与 QWidget 很相似，但是与 QWidget 不同，它不是继承自 QPaintDevice，而是 QGraphicsItem。通过它可以实现一个拥有事件、信号和槽、大小提示和策略的完整部件，还可以使用 QGraphicsLinearLayout 和 QGraphicsGridLayout 来实现部件的布局。

QGraphicsWidget 继承自 QGraphicsObject 和 QGraphicsLayoutItem，而 QGraphicsObject 继承自 QObject 和 QGraphicsItem，所以 QGraphicsWidget 既拥有以前窗口部件的一些特性也拥有图形项的一些特性。图形视图框架提供了对任意的窗口部件嵌入场景的无缝支持，这是通过 QGraphicsWidget 的子类 QGraphicsProxyWidget 实现的。可以使用 QGraphicsScene 类的 addWidget() 函数方便地将任何一个窗口部件嵌入到场景中，这也可以通过创建 QGraphicsProxyWidget 类的实例来实现。

(项目源码路径: src\11\11-8\myWidgetItem)新建空的 Qt 项目,名称为 myWidgetItem,完成后向其中添加新文件“main.cpp”,并在其中添加如下代码:

```
#include <QApplication>
#include <QGraphicsScene>
#include <QGraphicsView>
#include <QGraphicsWidget>
#include <QTextEdit>
#include <QPushButton>
#include <QGraphicsProxyWidget>
#include <QGraphicsLinearLayout>
#include <QObject>

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QGraphicsScene scene;
    // 创建部件，并关联它们的信号和槽
    QTextEdit * edit = new QTextEdit;
    QPushButton * button = new QPushButton("clear");
    QObject::connect(button, SIGNAL(clicked()), edit, SLOT(clear()));
    // 将部件添加到场景中
    QGraphicsWidget * editText = scene.addWidget(edit);
    QGraphicsWidget * pushButton = scene.addWidget(button);
    // 将部件添加到布局管理器中
    QGraphicsLinearLayout * layout = new QGraphicsLinearLayout;
    layout ->addItem(editText);
    layout ->addItem(pushButton);
    // 创建图形部件，设置其为一个顶层窗口，然后在其上应用布局
    QGraphicsWidget * form = new QGraphicsWidget;
    form ->setWindowFlags(Qt::Window);
    form ->setWindowTitle("Widget Item");
    form ->setLayout(layout);
    // 将图形部件进行扭曲，然后添加到场景中
    form ->shear(2, -0.5);
    scene.addItem(form);
    QGraphicsView view(&scene);
    view.show();
    return app.exec();
}
```

可以看到嵌入窗口部件结合了以前窗口部件的应用和现在图形项的应用,可以实现一些特殊的效果。QGraphicsWidget 还有一个 QGraphicsWebView 子类,可以直接

将网页内容集成到视图中,可以参考一下该类的帮助文档。Qt 中提供了一个 Pad Navigator Example 的示例程序,它在 Graphics View 分类中,还有一个 Embedded Dialogs 演示程序,都是关于这部分内容的,也可以参考一下。

图形视图框架是一个庞大且功能十分强大的体系,可以看到这些内容很有趣,可以实现很多图形和动画效果,所有学习这部分内容不会感到很吃力。Qt 中还提供了一个 40 000 Chips 演示程序,它使用了图形视图框架来管理大量的图形项,可以作为参考。

11.4 动画框架

动画框架是 Kinetic 项目的一部分,目的是提供一种简单的方法来创建平滑的具有动画效果的 GUI 界面。该框架是通过控制 Qt 的属性来实现动画的,可以应用在窗口部件和其他 QObject 对象上,也可以应用在图形视图框架中。动画框架在 Qt 4.6 中被引入。对该部分内容,可以在帮助中查看 The Animation Framework 关键字。

动画框架中主要的类及其关系如图 11-4 所示。其中基类 QAbstractAnimation 和它的两个子类 QVariantAnimation、QAnimationGroup 构成了动画框架的基础。这里的 QAbstractAnimation 是所有动画类的祖先,定义了一些所有动画类都共享的功能函数,比如动画的开始、停止和暂停等,它也可以接收时间变化的通知,通过继承这个类,可以创建自定义的动画类。

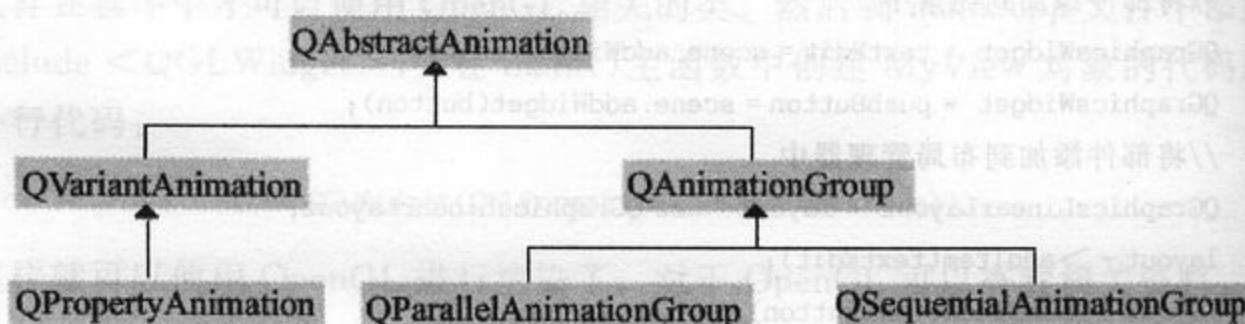


图 11-4 动画框架中主要类的关系图

动画框架中提供了 QPropertyAnimation 类,继承自 QVariantAnimation,用来执行 Qt 属性的动画。这个类使用缓和曲线(easing curve)来对属性进行插值。如果要对一个值使用动画就可以创建继承自 QObject 的类,然后在类中将该值定义为一个属性。当然属性动画为已经存在的窗口部件以及其他 QObject 子类提供了非常灵活的动画控制。Qt 现在支持的可以进行插值的 QVariant 类型有 int、double、float、QLine、QLineF、QPoint、QPointF、QSize、Q.SizeF、QRect、QRectF 和 QColor 等。如果要实现复杂的动画,可以通过动画组 QAnimationGroup 类实现,它的功能是作为其他动画类的容器,一个动画组中还可以包含另外的动画组。

动画框架也被设计作为状态机框架的一部分,将两者结合使用可以实现更为强大的功能。

11.4.1 实现属性动画

前面已经讲到 QPropertyAnimation 类可以对 Qt 属性进行插值,如果要对一个值进行动画,那么就要使用这个类,而它的父类 QVariantAnimation 是一个抽象类,无法直接使用。使用 Qt 属性来进行动画的最主要原因就是这样可以为已经存在的 Qt API 中的类提供灵活的动画。可以在 QWidget 类的帮助文档中查看它所有的属性,当然,并不是所有的属性都可以设置动画,必须是前面讲到的 Qt 支持的 QVariant 类型。下面来看一个例子。

(项目源码路径: src\11\11-9\myAnimation)新建空的 Qt 项目,名称为 myAnimation,完成后向其中添加新文件“main.cpp”,并在其中添加如下代码:

```
#include <QApplication>
#include <QPushButton>
#include <QPropertyAnimation>

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QPushButton button("Animated Button");
    button.show();
    QPropertyAnimation animation(&button, "geometry");
    animation.setDuration(10000);
    animation.setStartValue(QRect(0, 0, 100, 30));
    animation.setEndValue(QRect(250, 250, 200, 60));
    animation.start();
    return app.exec();
}
```

这里创建了一个按钮部件并让其显示,然后为按钮部件的 geometry 属性创建了动画,并使用 setDuration() 函数指定了动画的持续时间为 10000 毫秒即 10 秒,然后使用函数 setStartValue() 和 setEndValue() 分别设置了动画开始时和结束时 geometry 属性的值,最后调用 start() 函数来开始动画。这样就实现了按钮部件在 10 秒内从屏幕的(0, 0)点移动到(250, 250)点,与此同时由宽 100 高 30 的大小变为宽 200 高 60 的大小的动画。除了设置属性开始和结束的值以外,还可以调用 setKeyValueAt(qreal step, const QVariant &value) 函数在动画中间为属性设置值,其中 step 取值在 0.0~1.0 之间,0.0 表示开始位置,1.0 表示结束位置,而 value 为属性的值。将程序中调用 setStartValue() 和 setEndValue() 两个函数的代码更改为:

```
animation.setKeyValueAt(0, QRect(0, 0, 100, 30));
animation.setKeyValueAt(0.8, QRect(250, 250, 200, 60));
animation.setKeyValueAt(1, QRect(0, 0, 100, 30));
```

这样就实现了在 8 秒内按钮部件由(0, 0)点移动到(250, 250)点，并变化大小，然后在后 2 秒内又回到原点并且恢复原来大小的动画。

在动画中可以使用 pause() 来暂停动画；使用 resume() 来恢复暂停状态；使用 stop() 来停止动画；可以使用 setDirection() 函数设置动画的方向，这里可以设置为两个方向，默认是 QAbstractAnimation::Forward，动画的当前时间随着时间而递增，即从开始位置到结束位置；还有一个 QAbstractAnimation::Backward，动画的当前时间随着时间而递减，即从结束位置到开始位置；还可以使用 setLoopCount() 函数来设置动画的重复次数，默认为 1，表示执行一次，如果设置为 0，那么动画不会执行，如果设置为 -1，那么在调用 stop() 函数停止动画之前，它会一直持续。

11.4.2 使用缓和曲线

在前面程序的运行效果中可以看到，按钮部件的运动过程都是线性的，即匀速运动。除了在动画中添加更多的关键点，还可以使用缓和曲线，缓和曲线描述了怎样来控制 0 和 1 之间的插值的速度的功能，这样就可以在不改变插值的情况下控制动画的速度。

(项目源码路径：src\11\11-10\myAnimation) 将前面程序中的中间部分代码更改如下：

```
animation.setDuration(2000);
animation.setStartValue(QRect(250, 0, 100, 30));
animation.setEndValue(QRect(250, 300, 100, 30));
animation.setEasingCurve QEasingCurve::OutBounce);
```

这里使用了 QEasingCurve::OutBounce 缓和曲线，此时运行程序会发现它会使按钮部件就像从开始位置掉落到结束位置的皮球一样出现弹跳效果。在 QEasingCurve 类中提供了四十多种缓和曲线，而且还可以自定义缓和曲线。Qt 中还提供了一个 Easing Curves 的示例程序，可以演示所有缓和曲线的效果，它在 Animation Framework 分类中。

11.4.3 动画组

在一个应用中经常会包含多个动画，例如要同时移动多个图形项或者让它们一个接一个的串行移动。通过使用 QAnimationGroup 类可以实现复杂的动画，它的两个子类 QSequentialAnimationGroup 和 QParallelAnimationGroup 分别提供了串行动画组和并行动画组。

下面先来看一个串行动画组的例子。(项目源码路径：src\11\11-11\myAnimation) 先添加头文件 #include <QSequentialAnimationGroup>，再将主函数的中间部分内容更改如下：

```
QPushButton button("Animated Button");
button.show();
```

```

//按钮部件的动画 1
QPropertyAnimation * animation1 = new QPropertyAnimation(&button, "geometry");
animation1 ->setDuration(2000);
animation1 ->setStartValue(QRect(250, 0, 100, 30));
animation1 ->setEndValue(QRect(250, 300, 100, 30));
animation1 ->setEasingCurve(QEasingCurve::OutBounce);
//按钮部件的动画 2
QPropertyAnimation * animation2 = new QPropertyAnimation(&button, "geometry");
animation2 ->setDuration(1000);
animation2 ->setStartValue(QRect(250, 300, 100, 30));
animation2 ->setEndValue(QRect(250, 300, 200, 60));
//串行动画组
QSequentialAnimationGroup group;
group.addAnimation(animation1);
group.addAnimation(animation2);
group.start();

```

此时运行程序就会先执行动画 1, 等执行完动画 1 之后才执行动画 2, 动画的执行顺序与加入动画组的顺序是一致的。

下面再来看一个并行动画组的例子。(项目源码路径: src\11\11-12\myAnimation)先添加头文件 #include <QParallelAnimationGroup>, 再将主函数的中间部分内容更改如下:

```

QPushButton button1("Animated Button");
button1.show();
QPushButton button2("Animated Button2");
button2.show();
//按钮部件 1 的动画
QPropertyAnimation * animation1 = new QPropertyAnimation(&button1, "geometry");
animation1 ->setDuration(2000);
animation1 ->setStartValue(QRect(250, 0, 100, 30));
animation1 ->setEndValue(QRect(250, 300, 100, 30));
animation1 ->setEasingCurve(QEasingCurve::OutBounce);
//按钮部件 2 的动画
QPropertyAnimation * animation2 = new QPropertyAnimation(&button2, "geometry");
animation2 ->setDuration(2000);
animation2 ->setStartValue(QRect(400, 300, 100, 30));
animation2 ->setEndValue(QRect(400, 300, 200, 60));
//并行动画组
QParallelAnimationGroup group;
group.addAnimation(animation1);
group.addAnimation(animation2);
group.start();

```

现在运行程序可以看到两个按钮部件的动画是同时进行的。另外,使用动画组还有一个好处是,可以将它看作一个独立的动画,从而进行暂停、停止或者添加到其他动画组等操作。

11.4.4 在图形视图框架中使用动画

要对 QGraphicsItem 使用动画,也可以使用 QPropertyAnimation 类。但是, QGraphicsItem 并不是继承自 QObject 类,所以直接继承自 QGraphicsItem 的图形项并不能直接使用 QPropertyAnimation 类来创建动画。Qt 4.6 中提供了一个 QGraphicsItem 的子类 QGraphicsObject,它继承自 QObject 和 QGraphicsItem,这个类为所有需要使用信号和槽以及属性的图形项提供了一个基类,通过创建这个类的子类就可以使用属性动画了。QGraphicsObject 还提供了多个常用的属性,比如位置 pos、透明度 opacity、旋转 rotation 和缩放 scale 等,这些都可以直接用来设置动画。

(项目源码路径: src\11\11-13\myItemAnimation)新建空的 Qt 项目,名称设置为 myItemAnimation。完成后添加新的 C++ 类,类名 MyItem,基类 QGraphicsObject,类型信息选择“无”。完成后更改 myitem.h 文件中 MyItem 类的定义为:

```
class MyItem : public QGraphicsObject
{
public:
    MyItem(QGraphicsItem * parent = 0);
    QRectF boundingRect() const;
    void paint(QPainter * painter,
               const QStyleOptionGraphicsItem * option, QWidget * widget);
};
```

然后到 myitem.cpp 文件中,更改其内容如下:

```
#include "myitem.h"
#include <QPainter>

MyItem::MyItem(QGraphicsItem * parent) :
    QGraphicsObject(parent)
{
}
QRectF MyItem::boundingRect() const
{
    return QRectF(-10 - 0.5, -10 - 0.5, 20 + 1, 20 + 1);
}
void MyItem::paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
                   QWidget * widget)
{
    painter->drawRect(-10, -10, 20, 20);
```

最后添加新的 main.cpp 文件，并更改其内容如下：

```
#include <QApplication>
#include <QGraphicsScene>
#include <QGraphicsView>
#include "myitem.h"
#include <QPropertyAnimation>

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QGraphicsScene scene;
    scene.setSceneRect(-200, -150, 400, 300);
    MyItem * item = new MyItem;
    scene.addItem(item);
    QGraphicsView view;
    view.setScene(&scene);
    view.show();
    //为图形项的 rotation 属性创建动画
    QPropertyAnimation * animation = new QPropertyAnimation(item, "rotation");
    animation->setDuration(2000);
    animation->setStartValue(0);
    animation->setEndValue(360);
    animation->start(QAbstractAnimation::DeleteWhenStopped);

    return app.exec();
}
```

现在运行程序，图形项已经可以自动旋转了。当然这个动画效果也可以使用前面讲到的其他知识来实现，不过可以看到，使用动画框架是非常简单的，而且如果要实现更加复杂的动画，那么动画框架的优势就显而易见了。使用动画对象时，如果是创建对象的指针，而且执行一遍以后就不再使用，那么可以在 start() 函数中指定 DeleteWhenStopped 删除策略，这样当动画执行结束后便会自动销毁该动画对象。

除了继承 QGraphicsObject 类以外，当然也可以同时继承 Object 和 QGraphicsItem 类来实现自己的图形项，不过要注意 QObject 必须是第一个继承的类，这是元对象系统的要求。另外还可以继承自 QGraphicsWidget 类，这个类已经是 QObject 的子类了。如果要使用一个自定义的属性，那么就要先声明该属性，这个可以查看第 7 章的相关内容。Qt 还提供了一个 Animated Tiles 的示例程序，它在 Animation Framework 分类中，可以参考一下。

11.5 状态机框架

状态机框架提供了一些类来创建和执行状态图(state graphs),状态图为一个系统如何对外界激励进行反应提供了一个图形化模型,该模型是通过定义一些系统可能进入的状态以及系统怎样从一个状态切换到另一个状态来实现的。事件驱动的系统(比如 Qt 应用程序)的一个关键特性就是它的行为不总是仅仅依赖于前一个或者当前的事件,而且也依赖于将要执行的事件。通过使用状态图,这些信息会非常容易进行表达。

状态机框架提供了一个 API 和一个执行模型来有效的将状态图的元素和语义嵌入到 Qt 应用程序中。该框架与 Qt 的元对象系统是紧密结合的,例如状态间的切换可以由信号来触发。Qt 的事件系统用来驱动状态机。在状态机框架中的状态图是分层的,状态可以嵌套在其他状态中,在状态机的一个有效配置中的所有状态都拥有一个共同的祖先。状态机框架在 Qt 4.6 中被引入。对应本节内容,可以参考 The State Machine Framework 关键字。

11.5.1 创建状态机

下面先来看一个最简单的应用:假定状态机被一个 QPushButton 控制,包含 3 个状态:s1、s2 和 s3,其中 s1 是初始状态。当单击按钮时,状态机切换到另一个状态。图 11-5 是该状态机的状态图。

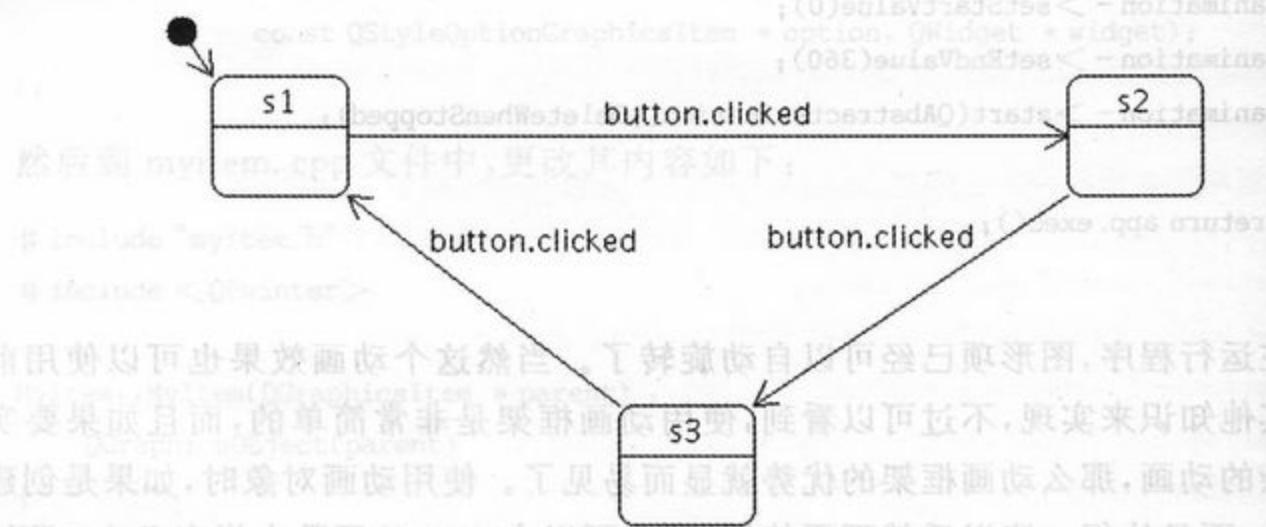


图 11-5 最简单的状态机的状态图

下面通过编写代码来看一下该状态机在程序中的实现。(项目源码路径:src\11\11-14\myStateMachine)新建空的 Qt 项目,名称为 myStateMachine,完成后向其中添加新的“main.cpp”文件,并更改其中内容为:

```

#include <QApplication>
#include <QPushButton>
#include <QState>
  
```

```
#include <QStateMachine>
int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QPushButton button("State Machine");
    // 创建状态机和3个状态，并将3个状态添加到状态机中
    QStateMachine machine;
    QState * s1 = new QState(&machine);
    QState * s2 = new QState(&machine);
    QState * s3 = new QState(&machine);
    // 为按钮部件的geometry属性分配一个值，当进入该状态时会设置该值
    s1->assignProperty(&button, "geometry", QRect(100, 100, 100, 50));
    s2->assignProperty(&button, "geometry", QRect(300, 100, 100, 50));
    s3->assignProperty(&button, "geometry", QRect(200, 200, 100, 50));
    // 使用按钮部件的单击信号来完成3个状态的切换
    s1->addTransition(&button, SIGNAL(clicked()), s2);
    s2->addTransition(&button, SIGNAL(clicked()), s3);
    s3->addTransition(&button, SIGNAL(clicked()), s1);
    // 设置状态机的初始状态并启动状态机
    machine.setInitialState(s1);
    machine.start();
    button.show();
    return app.exec();
}
```

要使用一个状态机需要先创建该状态机和使用到的状态，可以像程序中那样在创建状态时直接将其添加到状态机中，也可以使用 `QStateMachine::addState()` 来添加状态；创建完状态后要使用 `assignProperty()` 函数来为 `QObject` 对象的一个属性分配一个值，这样在进入该状态时就可以为 `QObject` 对象的这个属性设置该值；然后要使用 `addTransition()` 函数来完成一个状态到另一个状态的切换，可以关联 `QObject` 对象的一个信号来触发切换；最后要为状态机设置初始状态并启动状态机，这样当状态机启动时就会自动进入初始状态。状态机是异步执行的，它会成为应用程序事件循环的一部分。现在大家可以运行程序，然后单击按钮，查看状态机的运行效果。

当状态机进入一个状态时会发射 `QState::entered()` 信号，而退出一个状态时会发射 `QState::exited()` 信号。可以关联这两个信号来完成一些操作。例如在进入 `s3` 状态时将按钮最小化，那么可以在程序中调用 `setInitialState()` 函数的代码前添加如下代码：

```
QObject::connect(s3, SIGNAL(entered()), &button, SLOT(showMinimized()));
```

这里定义的3个状态间的切换是循环的，状态机也永远不会停止，如果想让状态机

完成一个状态后就停止,那么可以设置这个状态为 QFinalState 对象,将它加入状态图中,等切换到该状态时状态机就会发射 finished() 信号并停止。

11.5.2 在状态机中使用动画

如果将状态机中的 API 和 Qt 中的动画 API 相关联,那么就可以使分配到状态上的属性自动实现动画效果。在前面的程序中先添加头文件:

```
#include <QSignalTransition>
#include <QPropertyAnimation>
```

然后将进行状态切换的代码更改如下:

```
QSignalTransition * transition1 = s1 ->addTransition(&button,
                                                       SIGNAL(clicked()), s2);
QSignalTransition * transition2 = s2 ->addTransition(&button,
                                                       SIGNAL(clicked()), s3);
QSignalTransition * transition3 = s3 ->addTransition(&button,
                                                       SIGNAL(clicked()), s1);
QPropertyAnimation * animation = new QPropertyAnimation(&button, "geometry");
transition1 ->addAnimation(animation);
transition2 ->addAnimation(animation);
transition3 ->addAnimation(animation);
```

这样就可以在状态切换时使用动画效果了。在属性上添加动画,意味着当进入一个状态时分配的属性将无法立即生效,而是在进入时开始播放动画,然后以平滑的动画来达到属性分配的值。这里无需为动画设置开始和结束的值,它们会被隐含的进行设置,开始值就是开始播放动画时属性的当前值,结束值就是状态分配的属性的值。

1. 默认动画

如果想对一个属性指定一个动画,从而使所有的切换都默认使用这个动画,那么可以在状态机中使用默认动画。例如可以将前面程序中 3 个调用 addAnimation() 函数的代码使用下面一行代码来代替:

```
machine.addDefaultAnimation(animation);
```

要注意如果为一个属性明确指定了动画,那么它会优先于该属性的任何的默认动画。

2. 检测状态中的所有属性都已经被设置

首先来看一下如下代码:

```
QMessageBox * messageBox = new QMessageBox(mainWindow);
messageBox -> addButton(QMessageBox::Ok);
messageBox -> setText("Button geometry has been set!");
messageBox -> setIcon(QMessageBox::Information);
```

```

QState * s1 = new QState();
QState * s2 = new QState();
s2 ->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));
connect(s2, SIGNAL(entered()), messageBox, SLOT(exec()));
s1 ->addTransition(button, SIGNAL(clicked()), s2);

```

当按钮被单击时,状态机便会进入状态 s2,这时会设置按钮的 geometry 属性,然后弹出一个提示框来告诉用户 geometry 属性已经改变。在正常情况下,没有使用动画时,将会按照期望的操作进行。然而,如果在 s1 向 s2 切换时对 geometry 属性使用了动画,那么动画将会在进入 s2 时启动,而 geometry 属性不会在动画结束前达到指定的值。在这种情况下,提示框会在 geometry 属性获得指定的值之前弹出来,这就不是我们想要的结果了。

为了确保直到 geometry 属性获得最终的值以后提示框才会弹出来,可以使用状态的 propertiesAssigned() 信号,该信号会在属性被分配到最终的值时被发射,而无论使用了动画与否。再将上面的处理状态的几行代码更改如下:

```

QState * s1 = new QState();
QState * s2 = new QState();
s2 ->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));
QState * s3 = new QState();
connect(s3, SIGNAL(entered()), messageBox, SLOT(exec()));
s1 ->addTransition(button, SIGNAL(clicked()), s2);
s2 ->addTransition(s2, SIGNAL(propertiesAssigned()), s3);

```

这样当按钮被单击时,状态机会进入 s2,它会留在 s2 直到 geometry 属性获得了最终的值,然后切换到 s3。当进入 s3 后再弹出提示框。如果进入到 s2 的切换对 geometry 属性使用了动画,那么状态机会一直留在 s2 直到动画播放结束;如果没有使用动画,它会简单地设置属性的值,然后立即进入 s3。

3. 如果在动画结束前退出状态会发生什么

如果一个状态在动画结束前退出了,那么状态机的行为会依赖于切换的目标状态。如果目标状态明确的为该属性分配了一个值,那么该属性就会使用目标状态设置的这个值。如果目标状态没有为该属性分配任何值,这样会有两种选择:默认的,该属性会被分配切换时离开的那个状态所定义的值;但是如果设置了全局恢复策略,那么,恢复策略指定的值优先。

11.5.3 状态机框架的其他特性

1. 为状态分组来共享切换

假设要使用一个退出按钮在任何时候都可以退出应用程序,那么可以创建一个 QFinalState 最终状态,然后让它作为切换的目标状态,并且将切换关联到退出按钮的

单击信号上。这样虽然可以将最终状态和 s1、s2 以及 s3 状态分别进行切换，不过这样看起来很乱，而且如果以后再添加新的状态还要记得让它和最终状态进行切换。其实可以将 s1、s2 和 s3 进行分组来到达相同的效果。这就是创建一个新的状态，然后将这 3 个状态作为新状态的子状态，相对于子状态而言，前面直接添加到状态机中的状态都可以看做是顶级状态。新的状态机如图 11-6 所示。

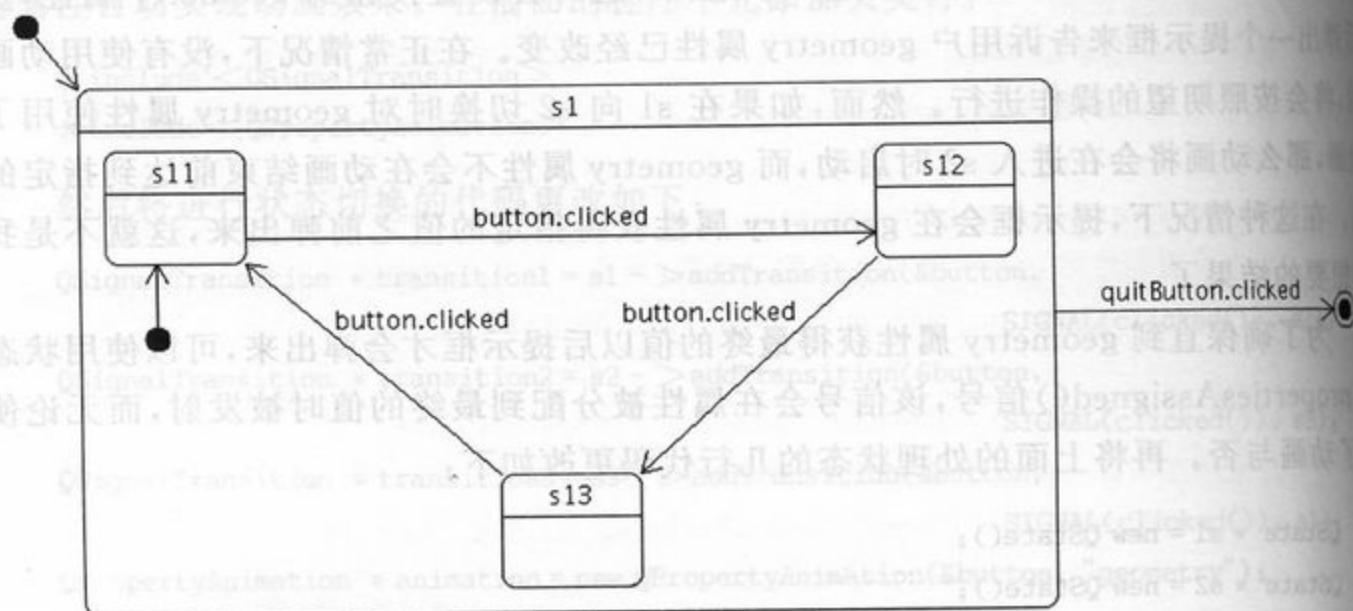


图 11-6 将状态进行分组后的状态机

这里将那 3 个状态分别重命名为 s11、s12 和 s13 来表明它们是新的顶层状态 s1 的子状态，子状态会隐含的继承它们父状态的切换，这意味着现在只需要从 s1 到最终状态 s2 添加一个切换即可，而新添加到 s1 中的状态都会自动继承这个切换。

(项目源码路径：src\11\11-15\myStateMachine)下面首先在前面的程序中添加头文件 #include <QFinalState>，然后更改主函数中的内容，更改后主要代码如下：

```

.... //省略了部分代码
QState * s1 = new QState(&machine);
QState * s11 = new QState(s1);
QState * s12 = new QState(s1);
QState * s13 = new QState(s1);
s1 ->setInitialState(s11);
.... //省略了部分代码
QFinalState * s2 = new QFinalState(&machine);
s1 ->addTransition(&quitButton, SIGNAL(clicked()), s2);
QObject::connect(&machine, SIGNAL(finished()), qApp, SLOT(quit()));
.... //省略了部分代码
  
```

这里在创建子状态时要指定父状态，而且还要指定初始子状态。当按下退出按钮时会切换到 s2 状态，为了可以退出应用程序，需要将状态机的 finished() 信号关联到 quit() 槽上。这里省略了部分代码，读者可以下载源码进行查看。

子状态也可以覆盖继承的切换，比如要在 s12 状态时忽略退出按钮，可以添加如

下一行代码：

```
s12 -> addTransition(quitButton, SIGNAL(clicked()), s12);
```

一个切换的目标状态可以是任意的状态，比如目标状态可以和源状态不在状态层次结构的同一个层中。

2. 使用历史状态来保存或者恢复当前状态

假设要在前面的例子中添加一个“中断”机制，按下一个按钮后可以让状态机执行一些无关的工作，而完成后又可以恢复到以前的状态，这可以通过使用历史状态 QHistoryState 来完成。历史状态是一个伪状态，代表了当父状态退出时所在的那个子状态。历史状态应创建为一个状态的子状态，这个状态就是我们要记录的当前子状态的父状态。当状态机在运行时检测到该状态的存在，就会在这个父状态退出时自动记录当前的子状态，切换到历史状态实际上就是切换到状态机先前保存的子状态。添加了中断机制的状态机如图 11-7 所示。

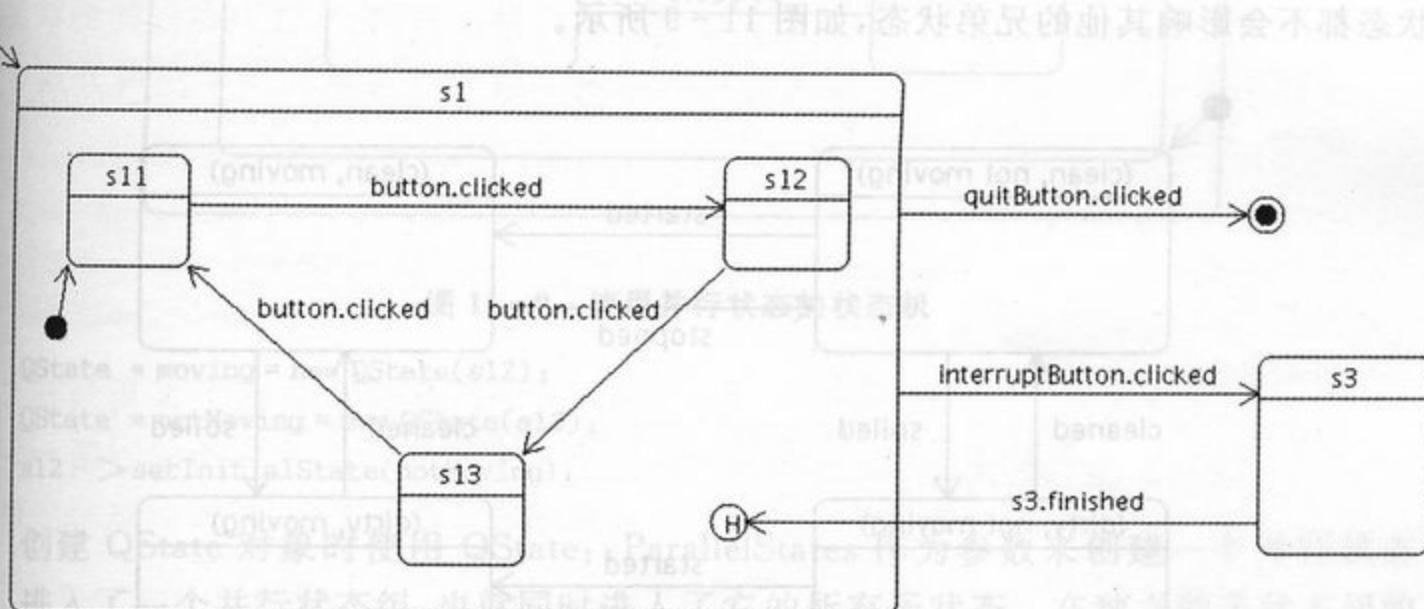


图 11-7 添加了中断机制的状态机

(项目源码路径：src\11\11-16\myStateMachine) 首先在前面的程序中添加头文件：

```
#include <QHistoryState>
#include <QMessageBox>
```

然后在主函数中添加如下代码：

```
QPushButton interruptButton("interrupt");
interruptButton.show();
QHistoryState * s1h = new QHistoryState(s1);
QState * s3 = new QState(&machine);
QMessageBox mbox;
mbox.addButton(QMessageBox::Ok);
```

```
mbox.setText("Interrupted!");
mbox.setIcon(QMessageBox::Information);
QObject::connect(s3, SIGNAL(entered()), &mbox, SLOT(exec()));
s3 ->addTransition(s1h);
s1 ->addTransition(&interruptButton, SIGNAL(clicked()), s3);
```

这里当进入 s3 状态时只是简单的显示一个提示框,然后通过历史状态立即返回到先前的子状态。

3. 使用并行状态来避免组合爆炸

假定在一个单一的状态机中包含了一个汽车的一组互斥的属性,例如 clean 对 dirty, moving 对 not moving。可以使用 4 个互斥的状态和 8 个切换来表示所有可能出现的组合,如图 11-8 所示。但是如果再添加第三个属性(比如 Red 对 Blue),总的状态数就会翻倍变为 8,而如果再添加第四个属性,那么状态总数就会变为 16。使用并行状态,就可以使状态的总数线性增长而不是指数增长,而且向并行状态中添加或者移除状态都不会影响其他的兄弟状态,如图 11-9 所示。

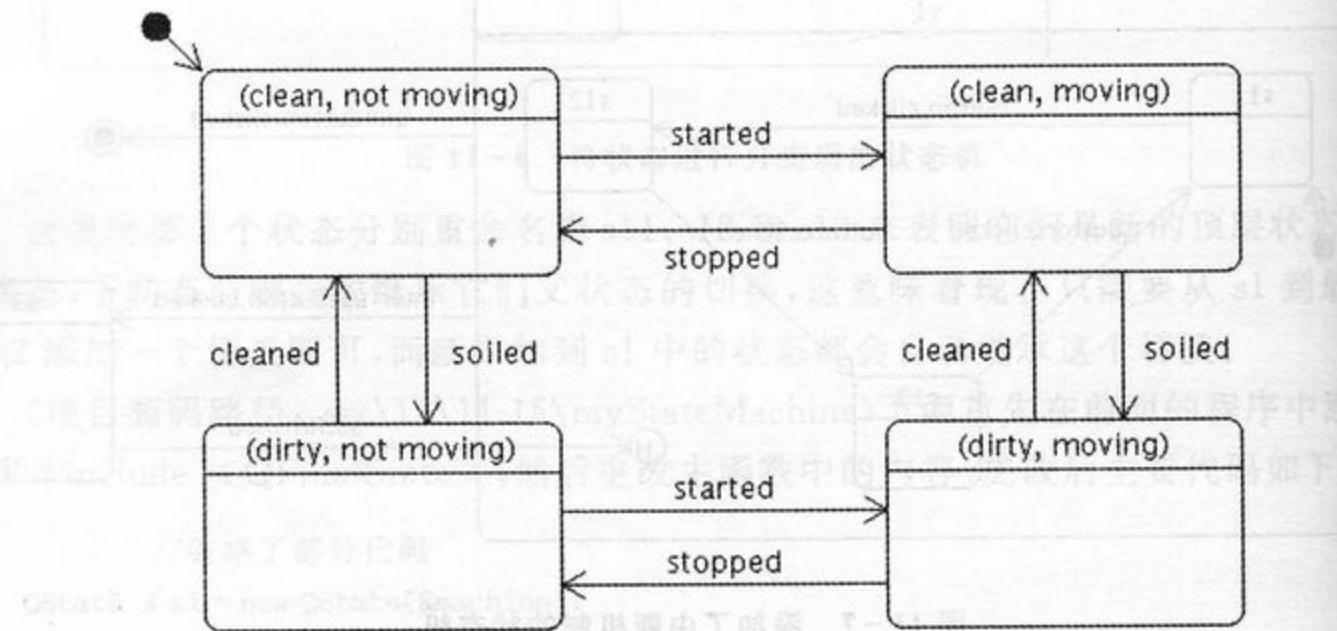


图 11-8 两对互斥属性的状态机

(项目源码路径: src\11\11-17\myStateMachine) 将前面的代码进行更改,其中的部分代码如下:

```
QState * s1 = new QState(QState::ParallelStates);

QState * s11 = new QState(s1);
QState * clean = new QState(s11);
QState * dirty = new QState(s11);
s11 ->setInitialState(clean);

QState * s12 = new QState(s1);
```

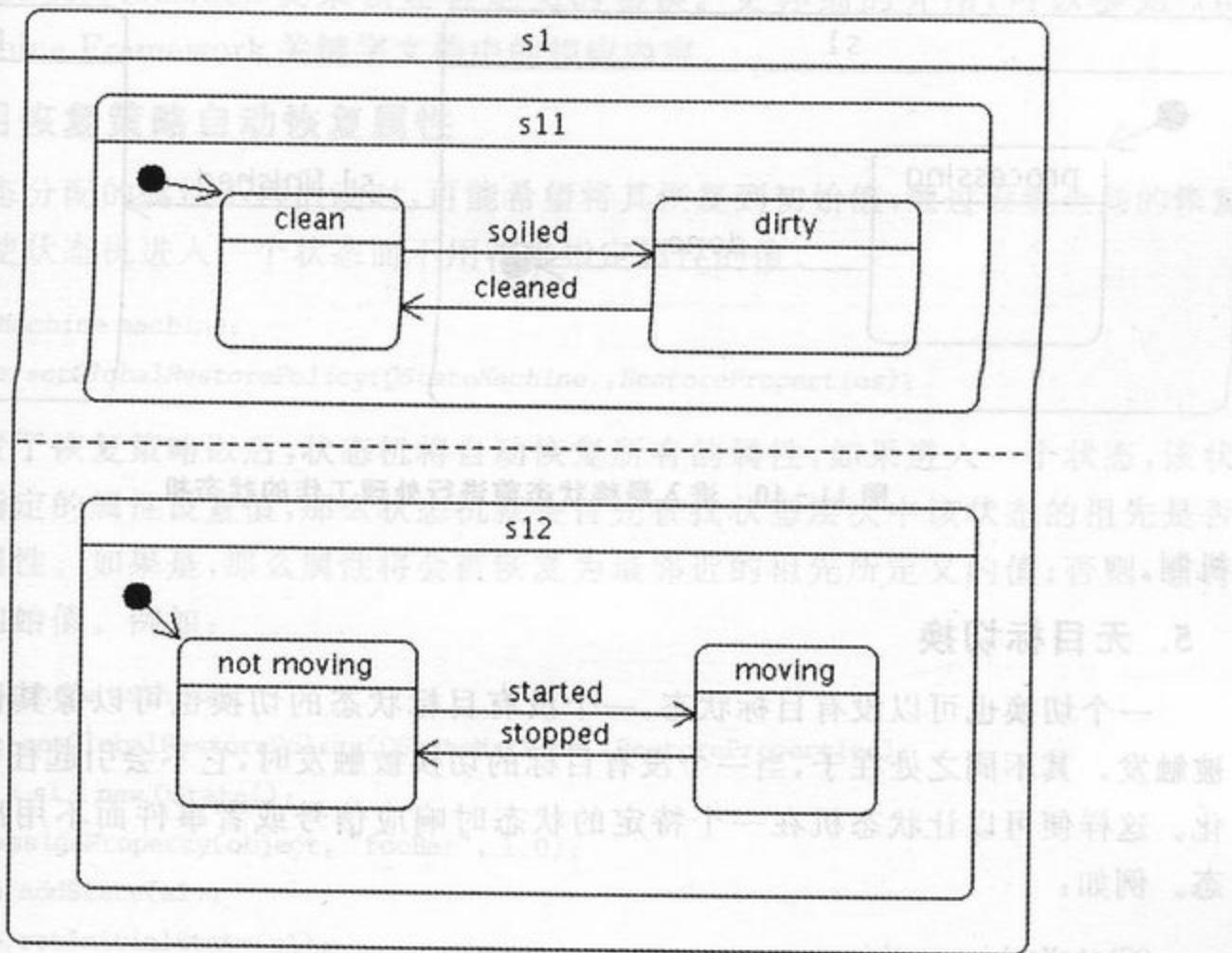


图 11-9 使用并行状态的状态机

```

QState * moving = new QState(s12);
QState * notMoving = new QState(s12);
s12 ->setInitialState(notMoving);

```

创建 QState 对象时使用 QState::ParallelStates 作为参数来创建一个并行状态组。进入了一个并行状态组，也就同时进入了它的所有子状态。在独立的子状态间的切换操作可以正常进行。然而任何一个子状态都可以通过切换而退出父状态，这时将退出父状态以及它所有的子状态。

4. 检测复合状态的结束信号

一个子状态可以是一个最终状态，当进入了一个最终子状态时，其父状态就会发射 QState::finished() 信号。图 11-10 显示了一个复合状态 s1 在进入最终状态前进行了一些处理工作。

当进入 s1 的最终状态时，s1 会自动发射 finished() 信号，可以使用信号切换来使这个事件触发一个状态变化：

```
s1 ->addTransition(s1, SIGNAL(finished()), s2);
```

如果想隐藏一个复合状态的内部细节，那么使用复合状态的最终状态是非常有效的。对于外界来说需要做的只是进入这个状态，然后等该状态完成工作后获得一个通知。当要创建一个复杂的（深嵌套的）状态机时，这将是一个非常强大的抽象和封装。

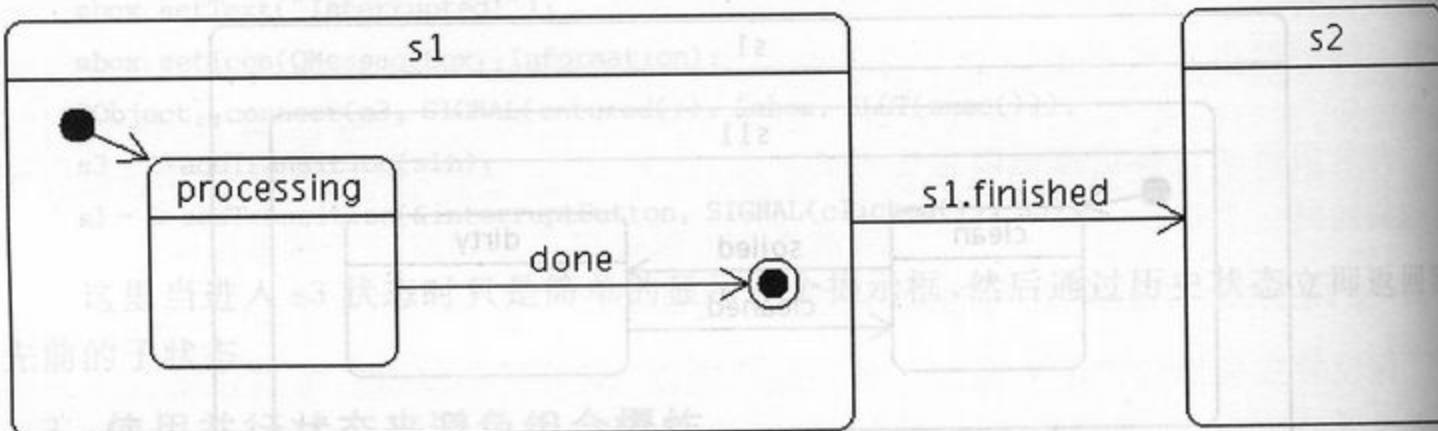


图 11-10 进入最终状态前进行处理工作的状态机

机制。

5. 无目标切换

一个切换也可以没有目标状态,一个没有目标状态的切换也可以像其他切换那样被触发。其不同之处在于,当一个没有目标的切换被触发时,它不会引起任何的状态变化。这样便可以让状态机在一个特定的状态时响应信号或者事件而不用离开这个状态。例如:

```

QStateMachine machine;
QState * s1 = new QState(&machine);
QPushButton button;
QSignalTransition * trans = new QSignalTransition(&button, SIGNAL(clicked()));
s1 ->addTransition(trans);

QMessageBox msgBox;
msgBox.setText("The button was clicked; carry on.");
QObject::connect(trans, SIGNAL(triggered()), &msgBox, SLOT(exec()));

machine.setInitialState(s1);
  
```

每当按下按钮时都会显示提示框,但是状态机仍然会留在它当前的状态。但是如果将目标状态显式的设置为 s1,那么每次 s1 都会退出然后重新进入(例如每次都会发射 entered() 和 exited() 信号)。

6. 事件、切换和守护

QStateMachine 在它自己的事件循环中运行。对于信号切换 (QSignalTransition 对象),当状态机截获相应的信号时,QStateMachine 自动发送 QStateMachine::SignalEvent 事件给它自己;相似的,对于 QObject 事件切换 (QEventTransition 对象),QStateMachine 将会发送 QStateMachine::WrappedEvent 事件。另外还可以使用 QStateMachine::postEvent() 发送自定义的事件给状态机,发送自定义事件时还需要

继承 QAbstractTransition 类来创建自定义的切换。更详细的介绍,可以参见 The State Machine Framework 关键字文档中的相应内容。

7. 使用恢复策略自动恢复属性

当状态分配的属性不再活动时,可能希望将其恢复到初始值,通过设置全局的恢复策略可以使状态机进入一个状态而不用明确指定属性的值。

```
QStateMachine machine;
machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);
```

当设置了恢复策略以后,状态机将自动恢复所有的属性,如果进入一个状态,该状态没有为指定的属性设置值,那么状态机就会首先查找状态层次中该状态的祖先是否定义了该属性。如果是,那么属性将会被恢复为最邻近的祖先所定义的值;否则,它将会恢复到初始值。例如:

```
QStateMachine machine;
machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);
QState * s1 = new QState();
s1->assignProperty(object, "fooBar", 1.0);
machine.addState(s1);
machine.setInitialState(s1);
QState * s2 = new QState();
machine.addState(s2);
```

这里设置了恢复策略 QStateMachine::RestoreProperties,假定在状态机开始时 fooBar 属性的值为 0.0,这样当在状态 s1 时,该属性的值为 1.0,而在 s2 时因为没有明确指定该属性的值,它便会隐含地恢复为 0.0。下面再来看一个例子:

```
QStateMachine machine;
machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);
QState * s1 = new QState();
s1->assignProperty(object, "fooBar", 1.0);
machine.addState(s1);
machine.setInitialState(s1);
QState * s2 = new QState(s1);
s2->assignProperty(object, "fooBar", 2.0);
s1->setInitialState(s2);
QState * s3 = new QState(s1);
```

这里 s1 拥有 s2 和 s3 两个子状态,当进入 s2 时,属性 fooBar 的值为 2.0,而当进入 s3 时,因为没有定义该属性的值,但是 s1 定义了该属性的值为 1.0,所以 s3 中该属性的值也为 1.0。

对于状态机的应用,可以参考一下 State Machine 中的几个示例程序。而要将图形视图框架、动画框架和状态机框架综合起来应用,可以参考一下 Sub-attaq 演示程序,

当然也可以参考一下《Qt 及 Qt Quick 开发实战精解》中的方块游戏。

11.6 小结

学习完本章应该掌握图形视图框架、动画框架和状态机框架的基本应用，会使用它们来创建动态的 GUI 程序。虽然这 3 个框架都很庞大，但都是一些很有趣的应用，学习起来并不枯燥。学习完本章后就可以看一下《Qt 及 Qt Quick 开发实战精解》中的方块游戏。

3. 无目标切换

一个矩阵也可以没有目标状态，一个没有目标状态的矩阵在处理时将忽略所有被触发，其不同之处在于，当一个矩阵从队列被触发时，它不会处理该矩阵的状态变化，这样便可以在任意取一个特定的状态时直接从矩阵中取出不用关心这个状态。例如：

```
QState *state = new QState();
state->addTransition(transition);
state->addTransition(transition2);
state->addTransition(transition3);
```

4. 条件、切换和守护

矩阵不存储时断点是不能实现，但是状态机中可以在矩阵中插入一个分支语句将矩阵模式的设置为 0，那么每次执行都会退出矩阵进入一个分支（通过 `switch` 和 `case` 语句）。

第 12 章

3D 绘图

OpenGL 是一个跨平台的用来渲染 3D 图形的标准 API。Qt 中提供了 QtOpenGL 模块,从而很轻松地实现了在 Qt 应用程序中使用 OpenGL,这主要是在 QGLWidget 类中完成的。要使用 QtOpenGL 模块,需要在项目文件中添加代码“QT += opengl”。

本章不会对 OpenGL 的专业知识进行过多的讲解,只会涉及在 Qt 应用程序中进行 3D 绘图的一些最基本应用。可以通过参考 OpenGL 分类中提供的示例程序和 Boxes 演示程序来学习更多的内容。

12.1 使用 OpenGL 绘制图形

QGLWidget 类是一个用来渲染 OpenGL 图形的部件,提供了在 Qt 应用程序中显示 OpenGL 图形的功能。这个类使用起来很简单,只需要继承该类,然后像使用其他 QWidget 部件一样来使用它。QGLWidget 提供了 3 个方便的虚函数,可以在子类中通过重新实现它们来执行典型的 OpenGL 任务:

- `initializeGL()`: 设置 OpenGL 渲染环境,定义显示列表等。该函数只在第一次调用 `resizeGL()` 或 `paintGL()` 前被调用一次。
- `resizeGL()`: 设置 OpenGL 的视口、投影等。每次部件改变大小时都会调用该函数。
- `paintGL()`: 渲染 OpenGL 场景。每当部件需要更新时都会调用该函数。

下面通过一个简单的程序来看一下怎样在 QGLWidget 中使用 OpenGL 来绘制图形。也可以参考一下 Qt 中的 Hello GL 示例程序,它在 OpenGL 分类中。

(项目源码路径: `src\12\12-1\myOpenGL`) 新建空的 Qt 项目,项目名称为 `myOpenGL`,然后往项目中添加新的 C++ 类,类名为 `MyGLWidget`,基类为 `QGLWidget`,类型信息选择“继承自 QWidget”。添加完成后,打开项目文件 `myOpenGL.pro`,添加一行代码:

QT += opengl

然后保存该文件。下面打开 myglwidget.h 文件，添加函数声明：

```
protected:
    void initializeGL();
    void resizeGL(int w, int h);
    void paintGL();
```

再到 myglwidget.cpp 文件中添加这 3 个函数的定义：

```
void MyGLWidget::initializeGL()
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_SMOOTH);
    glClearDepth(1.0);
    glEnable(GL_DEPTH_TEST);
}
```

glClearColor() 函数用来设置清除屏幕时使用的颜色，4 个参数分别用来设置红、绿、蓝颜色分量和 Alpha 值，它们的取值范围都是 0.0~1.0，这里 4 个参数都为 0，表示纯黑色。然后设置了阴影平滑（smooth shading），这样可以使色彩和光照更加精细。最后设置了深度缓存和启用深度测试，用来记录图形在屏幕内的深度值。

```
void MyGLWidget::resizeGL(int w, int h)
{
    glViewport(0, 0, (GLint)w, (GLint)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat)w/(GLfloat)h, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

glViewport() 函数用来设置视口的大小。使用 glMatrixMode() 设置了投影矩阵，投影矩阵用来为场景增加透视，后面使用了 glLoadIdentity() 重置投影矩阵，这样可以将投影矩阵恢复到初始状态。gluPerspective() 用来设置透视投影矩阵，这里设置视角为 45°，纵横比为窗口的纵横比，最近的位置为 0.1，最远的位置为 100，这两个值是场景中所能绘制的深度的临界值。可以想象，离我们眼睛比较近的东西看起来比较大，而比较远的东西看起来就比较小。最后设置并重置了模型视图矩阵。

```
void MyGLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    // 绘制三角形
```

```

glTranslatef(-2.0, 0.0, -6.0);
glBegin(GL_TRIANGLES);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(-1.0, -1.0, 0.0);
glVertex3f(1.0, -1.0, 0.0);
glEnd();
//绘制四边形
glTranslatef(4.0, 0.0, 0.0);
glBegin(GL_QUADS);
glVertex3f(-1.0, 1.0, 0.0);
glVertex3f(1.0, 1.0, 0.0);
glVertex3f(1.0, -1.0, 0.0);
glVertex3f(-1.0, -1.0, 0.0);
glEnd();
}

```

色题置好

在开始绘制以前,先要使用 glClear()清除屏幕和深度缓存。然后重置了模型视图矩阵,这样便将当前点移动到了窗口的中心,现在窗口中心即为坐标原点,X轴从左到右,Y轴从下到上,Z轴从里到外。完成这两步以后就可以进行图形的绘制了,在图形绘制开始时,一般会使用 glTranslatef()来移动坐标原点,它是相对于当前点来移动的,比如这里先将坐标原点左移 2.0,向里移 6.0,然后绘制了三角形(TRIANGLES)。绘制从 glBegin()开始,到 glEnd()结束,使用 glVertex3f()来设置各个顶点的坐标,顶点的绘制顺序可以是顺时针,也可以是逆时针。要注意逆时针绘制出来的是正面,而顺时针绘制出来的是反面,这一点在后面的纹理贴图部分会显示出来。当绘制完三角形以后,又将原点相对于当前点向右移动了 4.0,然后绘制了一个四边形(QUADS)。

最后再向项目中添加 main.cpp 文件,更改内容如下:

```

#include <QApplication>
#include "myglwidget.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyGLWidget w;
    w.resize(400, 300);
    w.show();
    return app.exec();
}

```

现在可以运行程序查看效果。QGLWidget 类中还提供了 renderText() 函数来方便进行文本的绘制,只需要在 paintGL() 函数的最后调用该函数,并在参数中指定坐

标、文本和字体就可以了；还有一个 renderPixmap() 函数可以返回指定矩形的截图，然后可以使用 QPixmap 类的 save() 函数来保存该截图。

12.2 设置颜色

可以使用 glColor3f() 函数来设置绘制时使用的颜色，3 个参数用来指定 RGB(红绿蓝)颜色分量，取值范围为 0.0~1.0。可以在绘制一个顶点时指定使用的颜色，如果后面不再设置其他颜色，那么所有的顶点都将使用同样的颜色。

(项目源码路径：src\12\12-2\myOpenGL)下面将图形的绘制代码更改如下：

```
glTranslatef(-2.0, 0.0, -6.0);
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glColor3f(0.0, 1.0, 0.0);
glVertex3f(-1.0, -1.0, 0.0);
glColor3f(0.0, 0.0, 1.0);
glVertex3f(1.0, -1.0, 0.0);
glEnd();

glTranslatef(4.0, 0.0, 0.0);
glBegin(GL_QUADS);
glColor3f(1.0, 1.0, 0.0);
glVertex3f(-1.0, 1.0, 0.0);
glVertex3f(1.0, 1.0, 0.0);
glVertex3f(1.0, -1.0, 0.0);
glVertex3f(-1.0, -1.0, 0.0);
glEnd();
```

这时运行程序，可以看到三角形的 3 个角分别是红色、绿色和蓝色，而正方形是纯黄色的。

12.3 实现 3D 图形

前面的程序中只绘制了一个面，这样看不出什么三维效果，这一节通过绘制多个面来实现效果明显的三维图形。

(项目源码路径：src\12\12-3\myOpenGL)将程序中绘制图形的代码更改为：

```
glTranslatef(0.0, 0.0, -6.0);
glRotatef(45, 0.0, 1.0, 0.0);
glBegin(GL_QUADS);
//上面
```

```

glColor3f(1.0, 0.0, 0.0);
glVertex3f(1.0, 1.0, 1.0);
glVertex3f(1.0, 1.0, -1.0);
glVertex3f(-1.0, 1.0, -1.0);
glVertex3f(-1.0, 1.0, 1.0);
//下面
glColor3f(0.0, 1.0, 0.0);
glVertex3f(1.0, -1.0, 1.0);
glVertex3f(1.0, -1.0, -1.0);
glVertex3f(-1.0, -1.0, -1.0);
glVertex3f(-1.0, -1.0, 1.0);
//前面
glColor3f(0.0, 0.0, 1.0);
glVertex3f(1.0, 1.0, 1.0);
glVertex3f(-1.0, 1.0, 1.0);
glVertex3f(-1.0, -1.0, 1.0);
glVertex3f(1.0, -1.0, 1.0);
glEnd();

```

这里使用了 `glRotatef(angle, x, y, z)` 函数来对坐标轴进行旋转, 第一个参数是旋转的角度, 后面 3 个参数用来确定旋转轴向量。旋转轴经过原点, 指向 (x, y, z) 点。图形的旋转满足右手定则, 就是用右手握住旋转轴, 大拇指指向旋转轴所指的方向, 然后其他 4 个手指所指的方向就是要旋转的方向。再往下面, 分别绘制了 3 个正方形作为正方体的 3 个面, 还可以再补充上其他几个面。

下面修改程序, 使用键盘来控制图形进行旋转。首先在 `myglwidget.h` 文件中添加键盘按下事件处理函数的声明:

```
void keyPressEvent(QKeyEvent *);
```

然后添加几个变量的定义:

```
private:
    GLfloat translate, xRot, yRot, zRot;
```

下面到 `myglwidget.cpp` 文件中添加头文件 `#include <QKeyEvent>`, 然后在构造函数中初始化变量:

```
translate = -6.0;
xRot = yRot = zRot = 0.0;
```

再到 `paintGL()` 函数中将 `glTranslatef()` 函数和 `glRotatef()` 函数的调用更改为:

```
glTranslatef(0.0, 0.0, translate);
glRotatef(xRot, 1.0, 0.0, 0.0);
glRotatef(yRot, 0.0, 1.0, 0.0);
glRotatef(zRot, 0.0, 0.0, 1.0);
```

最后添加键盘按下事件处理函数的定义：

```
void MyGLWidget::keyPressEvent(QKeyEvent * event)
{
    switch (event->key())
    {
        case Qt::Key_Up :
            xRot += 10;
            break;
        case Qt::Key_Left :
            yRot += 10;
            break;
        case Qt::Key_Right :
            zRot += 10;
            break;
        case Qt::Key_Down :
            translate -= 1;
            if (translate <= -20)
                translate = -1;
            break;
    }
    updateGL();
}
QGLWidget::keyPressEvent(event);
}
```

这里先分别设置了使用几个方向键来控制图形在各个轴的旋转角度和原点的深度，然后使用了 updateGL() 函数来更新绘制。现在运行程序，使用键盘方向键对图形进行控制。

12.4 使用纹理贴图

前面的程序中生成了正方体的 3 个面，还可以使用图片来作为这 3 个面的纹理贴图，这样可以达到更加真实的效果。对该部分内容，可以参考一下 Textures 示例程序，它也在 OpenGL 分类中。

(项目源码路径：src\12\12-4\myOpenGL)首先在 myglwidget.h 文件中添加一个私有变量的声明：

```
GLuint textures[3];
```

这个数组变量用来存储纹理。然后到 myglwidget.cpp 文件中在 initializeGL() 函数中添加如下代码：

```
textures[0] = bindTexture(QPixmap("../myOpenGL/side1.png"));
textures[1] = bindTexture(QPixmap("../myOpenGL/side2.png"));
```

```
textures[2] = bindTexture(QPixmap("../myOpenGL/side3.png"));
glEnable(GL_TEXTURE_2D);
```

这里先使用 QGLWidget 的 bindTexture() 函数生成并绑定了 3 个 2D 纹理，然后使用 glEnable() 开启了 2D 纹理贴图功能。下面再到 paintGL() 函数中将绘图代码更改如下：

```
//上面
glBindTexture(GL_TEXTURE_2D, textures[2]);
glBegin(GL_QUADS);
glTexCoord2f(1.0, 0.0);
glVertex3f(1.0, 1.0, 1.0);
glTexCoord2f(1.0, 1.0);
glVertex3f(1.0, 1.0, -1.0);
glTexCoord2f(0.0, 1.0);
glVertex3f(-1.0, 1.0, -1.0);
glTexCoord2f(0.0, 0.0);
glVertex3f(-1.0, 1.0, 1.0);
glEnd();

//下面
glBindTexture(GL_TEXTURE_2D, textures[1]);
glBegin(GL_QUADS);
glTexCoord2f(1.0, 0.0);
glVertex3f(1.0, -1.0, 1.0);
glTexCoord2f(1.0, 1.0);
glVertex3f(1.0, -1.0, -1.0);
glTexCoord2f(0.0, 1.0);
glVertex3f(-1.0, -1.0, -1.0);
glTexCoord2f(0.0, 0.0);
glVertex3f(-1.0, -1.0, 1.0);
glEnd();

//前面
glBindTexture(GL_TEXTURE_2D, textures[0]);
glBegin(GL_QUADS);
glTexCoord2f(1.0, 1.0);
glVertex3f(1.0, 1.0, 1.0);
glTexCoord2f(0.0, 1.0);
glVertex3f(-1.0, 1.0, 1.0);
glTexCoord2f(0.0, 0.0);
glVertex3f(-1.0, -1.0, 1.0);
glTexCoord2f(1.0, 0.0);
glVertex3f(1.0, -1.0, 1.0);
glEnd();
```



绘制图形时要使用纹理,就要在开始绘图前使用 `glBindTexture()` 函数来绑定纹理。而如果要更改使用的纹理,就必须再次使用 `glBegin()` 来开始一个新的绘图。`glTexCoord2f()` 用来设置纹理的坐标,应该在每次绘制顶点前调用它。该函数的第一个参数是 X 坐标,0.0 表示纹理的左侧,0.5 表示纹理的中点,1.0 表示纹理的右侧;第二个参数是 Y 坐标,0.0 表示纹理的底部,0.5 表示纹理的中点,1.0 表示纹理的顶部。需要做的就是将纹理的 4 个顶点正确的对应到 4 边形的 4 个顶点上。现在运行程序,效果如图 12-1 所示。可以看到,下面那张图片从外面看显示的是 2 的背面,这是因为绘制下面那张图片时是从上面向下看进行逆时针绘制的,这样看到的是下面那个面的背面,所以会出现这样的问题,更改一下顶点的绘制顺序就可以了。

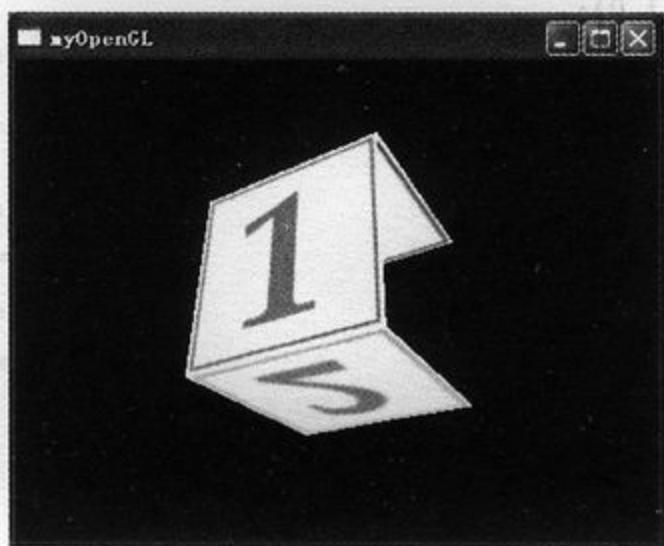


图 12-1 使用纹理贴图运行效果

12.5 在 3D 场景中绘制 2D 图形

使用 `QGLWidget` 可以实现 3D 绘图,因为 `QGLWidget` 也是一个 `QWidget` 部件,所以也可以进行一般的 2D 绘图,这样就可以在 3D 场景中绘制 2D 图形了。对该部分内容,读者可以参考一下 Qt 中的 Overpainting 示例程序。

(项目源码路径: `src\12\12-5\myOpenGL`)首先在 `myglwidget.h` 文件中声明两个函数:

```
void paintEvent(QPaintEvent * );
void setupViewport(int w, int h);
```

然后到 `myglwidget.cpp` 文件中,首先在构造函数中关闭自动填充背景,即添加如下代码:

```
setAutoFillBackground(false);
```

然后添加 `setupViewport()` 函数的定义,就是将前面 `resizeGL()` 函数中的内容全部剪切到这个函数中,然后在 `resizeGL()` 函数中调用 `setupViewport()` 函数:

```
setupViewport(w, h);
```

下面再到 keyPressEvent() 函数中将以前调用的 updateGL() 函数更改为 update() 函数。最后添加 paintEvent() 函数的定义：

```
void MyGLWidget::paintEvent(QPaintEvent * )
{
    //在当前窗口中进行 OpenGL 的绘制
    makeCurrent();
    //将模型视图矩阵压入堆栈
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    //以下是以前 initializeGL() 函数中的全部内容
    ..... //将 initializeGL() 函数中的内容全部剪切到这里

    //该函数中是以前 resizeGL() 函数中的全部内容
    setupViewport(width(), height());
    //以下是以前 paintGL() 函数中的全部内容
    ..... //将 paintGL() 函数中的内容全部剪切到这里

    //关闭启用的功能并弹出模型视图矩阵
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_TEXTURE_2D);
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();

    //下面是 2D 绘图的内容
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    painter.setBrush(Qt::red);
    painter.drawRect(0, 0, 100, 100);
    painter.end();
}
```

现在运行程序，效果如图 12-2 所示。

绘制图形时要使用

档。而把背景设置为黑色的函数是 glClearColor(), 它用来设置颜色, 其参数是 X 坐标, 0.0 表示红色, 0.0 表示绿色, 0.0 表示蓝色, 1.0 表示将纹理的 4 个顶点都设为白色, 如图 12-1 所示。可是下面那张图片时从后面看的, 所以会出现这样的问题。

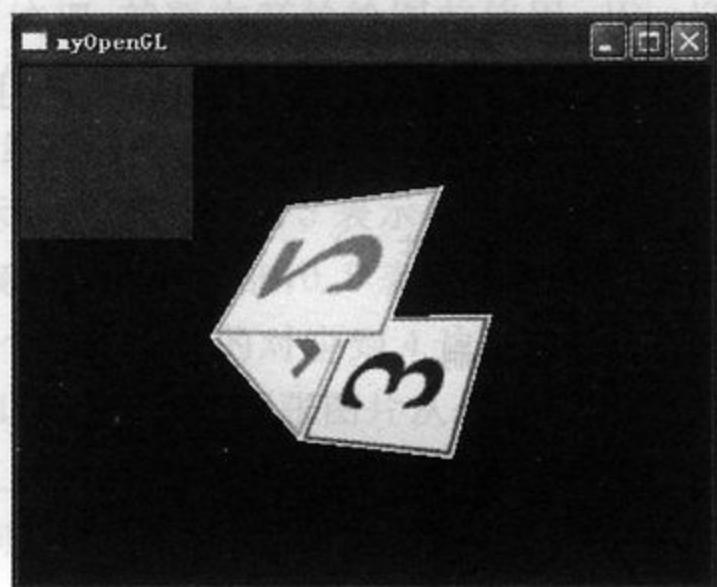


图 12-2 在 3D 场景中进行 2D 绘图

12.6 小结

本章简单介绍了怎样在 Qt 应用程序中使用 OpenGL 来绘制 3D 图形, 并讲解了其中最典型的应用。而真正要进行 3D 图形开发还是要学习 OpenGL 的专业知识, 为了使没有基础的初学者容易理解, 尽量避免了涉及太多的 OpenGL 的专业术语。

12.5 在 3D 场景中绘制 2D 图形

使用 QGLWidget 可以实现 3D 绘图, 因为 QGLWidget 也是 QWidget 的子类, 所以也可以进行一般的 2D 绘图, 这样就可以在 3D 场景中绘制 2D 图形了。有关内容, 读者可以参考一下 Qt 中的 Overpainting 示例程序。

该示例程序的路径是: `Qt\Qt4.8.1\Examples\QtOpenGL\Overpainting`, 先在 `QGLWidget::paintEvent()` 函数中调用 `QGLWidget::clear(GL_COLOR_BUFFER_BIT)` 清除画面, 然后调用 `QGLWidget::beginGL()`。

然后调用 `myOpenGL.cpp` 文件中, 首先在 `setupViewports()` 函数中写以下代码:

```
void setupViewports(QGLWidget *w) {
    w->clear(GL_COLOR_BUFFER_BIT);
```

然后读取 `setupViewports()` 函数的定义, 就是将前面 `resizeGL()` 函数中的代码剪切到这个函数中, 然后在 `resizeGL()` 函数中调用 `setupViewport()` 函数。

影音媒体篇

影音媒体篇

再到 mainwindow.cpp 文件中, 将构造函数里添加的调用 play() 函数的代码更

- 第13章 Qt多媒体应用

- 第 14 章 Phonon 多媒体框架

第 13 章

Qt 多媒体应用

Qt 对于音频视频的播放和控制等多媒体应用提供了强大的支持。要想使计算机发出响声,最简单的方法是调用 `QApplication::beep()` 静态函数;而对于简单的音频播放,可以使用 `QSound` 类;对于简单的动画播放,可以使用 `QMovie` 类;要想对音频视频实现更多的控制,可以使用 Phonon 多媒体框架;而对于音频视频底层的控制,可以使用 `QtMultimedia` 模块。关于 Phonon 多媒体框架的内容放在下一章再讲,这一章主要讲解其他几个方面的应用。

13.1 使用 `QSound` 播放声音

`QSound` 类提供了对平台音频设备的访问,提供了 GUI 应用程序中最常用的音频操作:异步播放一个声音文件。可以使用 `QSound::isAvailable()` 静态函数来判断在平台上是否存在相应的音频设备,`QSound` 在各平台上使用的音频设备如表 13-1 所列。

表 13-1 `QSound` 在各平台上使用的音频设备

平 台	音频设备
Microsoft Windows	使用底层的多媒体系统;仅支持 WAVE 格式的音频文件
X11	使用网络音频系统,支持 WAVE 和 AU 格式的音频文件
Mac OS X	使用 NSSound,可以支持 NSSound 支持的所有格式
Qt for Embedded Linux	使用内置的混合声音服务器,直接访问 /dev/dsp 目录,仅支持 WAVE 格式
Symbian	使用 CmndaAudioPlayerUtility,可以支持 Symbian 系统支持的所有格式

下面通过例子来看一下 `QSound` 的应用。(项目源码路径: `src\13\13-1\mySound`)新建 Qt Gui 应用,名称为 `mySound`,类名 `MainWindow` 和基类 `QMainWindow` 保持默认即可。完成后在 `mainwindow.cpp` 文件中添加头文件 `#include <QSound>`,然

后在构造函数中添加如下一行代码：

```
QSound::play("../mySound/sound.wav");
```

这时运行程序就可以播放指定的音频文件了，注意这里将音频文件放在了项目目录中。因为现在 QSound 并不支持资源文件，所以音频文件必须要放在程序外面。除了简单使用静态函数进行播放外，也可以先构建一个 QSound 对象，然后再调用 play() 槽进行播放，可以使用 stop() 槽来停止声音的播放，还可以使用 setLoops() 函数设置播放重复的次数，如果设置为 -1 表示无限循环。

先到 mainwindow.h 文件中添加前置声明 class QSound；然后声明一个私有对象：

```
QSound * sound;
```

再到 mainwindow.cpp 文件中，将构造函数里添加的调用 play() 函数的代码更改为：

```
sound = new QSound("../mySound/sound.wav", this);
```

然后双击 mainwindow.ui 文件进入设计模式，向界面上添加两个 Push Button 和一个 Spin Box，并将两个按钮的文本分别改为“播放”和“停止”。然后更改 Spin Box 的属性，将最小值 minimum 设置为 -1，将当前值 value 设置为 1。最后分别转到两个按钮的 clicked() 槽和 Spin Box 的 valueChanged(int) 槽，更改它们的内容如下：

```
void MainWindow::on_pushButton_clicked()
{
    //播放按钮
    sound->play();
}

void MainWindow::on_pushButton_2_clicked()
{
    //停止按钮
    sound->stop();
}

void MainWindow::on_spinBox_valueChanged( int value )
{
    sound->setLoops(value);
}
```

现在运行程序可以设置播放的次数，然后使用开始按钮进行播放，使用停止按钮来停止播放了。要说明一下，在 Windows 平台上，如果设置了循环次数，那么 stop() 函数无法立即停止播放，需要完成当前的循环才可以停止播放。

使用 QSound 可以实现一些简短的声音的播放，使用它来实现单击按钮或者其他事件的音效是很好的选择。根据平台音频设备的不同，如果使用 QSound 同时播放多个音频文件，那么后面播放的声音或者会与前面播放的声音进行混合，或者会停止前面播放的声音，在 Windows 平台上，新播放的声音会停止前面播放的声音。如果要播放其他格式的音频文件，或者需要对播放进行更多的控制，那么就需要使用 Phonon 多媒

体框架来实现了。

13.2 使用 QMovie 播放动画

前面已经多次提到过 `QMovie` 类, `QMovie` 类是一个使用 `QImageReader` 来播放动画的便捷类。该类用来显示没有声音的简单动画, 主要支持 GIF 和 MNG 格式的文件, 其支持的格式可以使用 `QMovie::supportedFormats()` 静态函数来获取。要播放一个动画, 只需要先创建一个 `QMovie` 对象, 并为其指定要播放的动画文件, 然后将 `QMovie` 对象传递给 `QLabel::setMovie()` 函数, 最后调用 `start()` 函数来开始播放动画, 例如:

```
QLabel label;
QMovie * movie = new QMovie("animations/fire.gif");
label.setMovie(movie);
movie->start();
```

还可以使用 `setPaused(true)` 来暂停动画的播放, 然后使用 `setPaused(false)` 来恢复播放; 使用 `stop()` 函数可以停止动画的播放。`QMovie` 一共有 3 个状态, 如表 13-2 所列, 每当状态改变时都会发射 `stateChanged()` 信号, 一般可以关联这个信号来改变播放、暂停等按钮的状态。

表 13-2 `QMovie` 的不同状态

常量	描述
<code>QMovie::NotRunning</code>	动画未执行。这是 <code>QMovie</code> 的初始状态, 如果调用了 <code>stop()</code> 函数或者动画已经结束都会进入该状态
<code>QMovie::Paused</code>	动画被暂停。当调用 <code>setPaused(true)</code> 函数后会进入该状态, 会保持当前的帧号, 当调用了 <code>setPaused(false)</code> 函数后会继续播放下一帧
<code>QMovie::Running</code>	动画正在播放

可以使用 `frameCount()` 函数来获取当前动画总的帧数; `currentFrameNumber()` 函数可以返回当前帧的序列号, 动画第一个帧的序列号为 0; 如果动画播放到了一个新的帧, `QMovie` 会发射 `updated()` 信号, 这时可以使用 `currentImage()` 或者 `currentPixmap()` 函数来获取当前帧的一个副本。还可以使用 `setCacheMode()` 来设置 `QMovie` 的缓存模式, 这里有两个选项: `QMovie::CacheNone` 和 `QMovie::CacheAll`, 前者是默认选项, 不缓冲任何帧; 后者是缓存所有的帧。如果指定了 `QMovie::CacheAll` 选项, 那么就可以使用 `jumpToFrame()` 来跳转到指定的帧了。另外, 还可以使用 `setSpeed()` 来设置动画的播放速度, 该速度是以原始速度的百分比来衡量的, 默认的速度为 100%。下面通过具体的例子来看一下这些功能的应用。

(项目源码路径: src\13\13-2\myMovie)新建 Qt Gui 应用,名称 myMovie,类名为 MainWindow 和基类 QMainWindow 都保持默认即可。完成后先在设计模式设计如图 13-1 所示的界面。其中要设置水平滑块部件 Horizontal Slider 的 tickPosition 属性为 TicksBelow,然后设置间隔 tickInterval 的数值为 10;选中“暂停”按钮的 checkable 属性;设置 Spin Box 的后缀 suffix 属性为“%”,最大值 maximum 为 999,当前值 value 为 100。

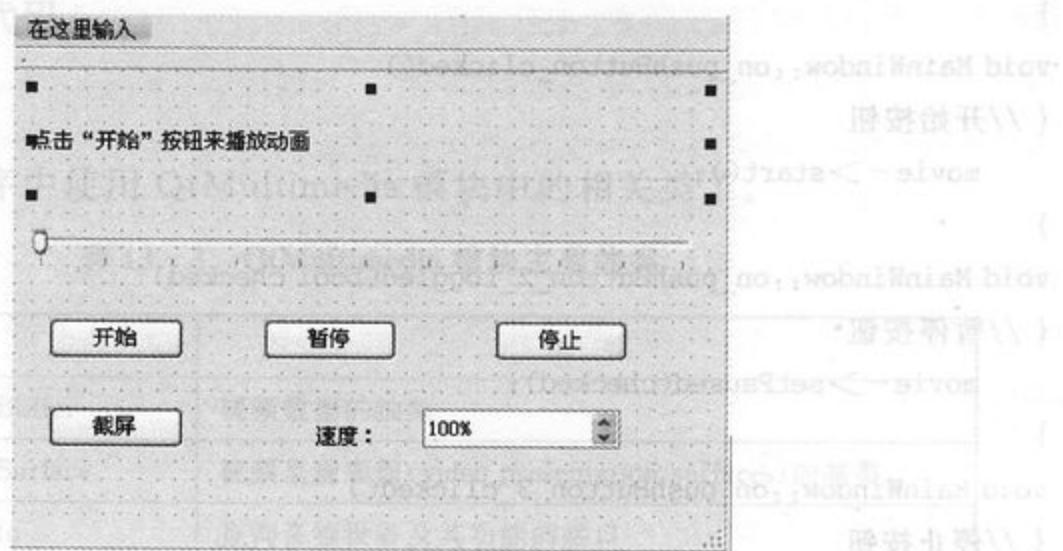


图 13-1 设计动画播放界面

然后到 mainwindow.h 文件中添加前置声明 class QMovie;再添加私有对象声明:

```
QMovie * movie;
```

到 mainwindow.cpp 文件中添加头文件 #include <QMovie>,然后在构造函数中添加如下代码:

```
//设置标签的对齐方式为居中对齐、自动填充背景为暗色
ui->label->setAlignment(Qt::AlignCenter);
ui->label->setBackgroundRole(QPalette::Dark);
ui->label->setAutoFillBackground(true);

movie = new QMovie(this);
movie->setFileName("../myMovie/movie.gif");
//设置缓存模式
movie->setCacheMode(QMovie::CacheAll);
//设置动画大小为标签的大小
QSize size = ui->label->size();
movie->setScaledSize(size);
ui->label->setMovie(movie);

//设置水平滑块的最大最小值,然后当动画播放时自动更改滑块的值
ui->horizontalSlider->setMinimum(0);
ui->horizontalSlider->setMaximum(movie->frameCount());
```

```
connect(movie, SIGNAL(frameChanged(int)),
       ui->horizontalSlider, SLOT(setValue(int)));
```

然后分别从设计模式进入相应部件的相应信号的槽,更改如下:

```
void MainWindow::on_horizontalSlider_valueChanged(int value)
{ //播放进度
    movie->jumpToFrame(value);
}

void MainWindow::on_pushButton_clicked()
{ //开始按钮
    movie->start();
}

void MainWindow::on_pushButton_2_toggled(bool checked)
{ //暂停按钮
    movie->setPaused(checked);
}

void MainWindow::on_pushButton_3_clicked()
{ //停止按钮
    movie->stop();
}

void MainWindow::on_pushButton_4_clicked()
{ //截屏按钮
    int id = movie->currentFrameNumber();
    QPixmap pix = movie->currentPixmap();
    pix.save(QString("../myMovie/%1.png").arg(id));
}

void MainWindow::on_spinBox_valueChanged(int value)
{ //播放速度
    movie->setSpeed(value);
}
```

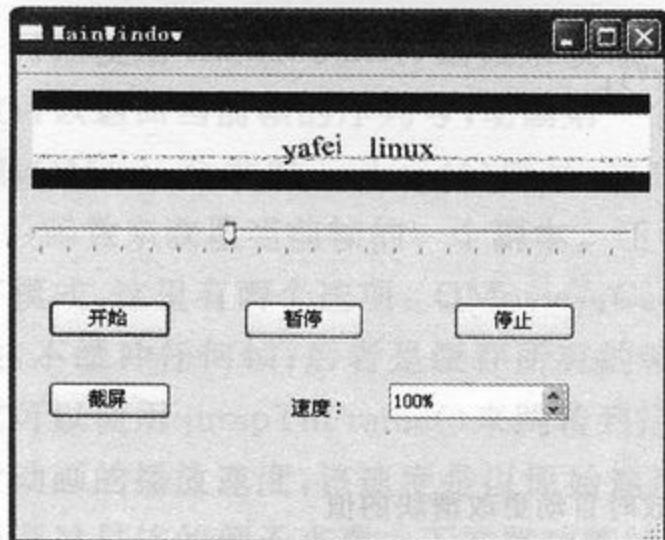


图 13-2 播放动画效果

现在运行程序,效果如图 13-2 所示。关于 QMovie 类的应用,可以参考一下 Movie Player 示例程序,它在 Widgets 分类下。QMovie 类只能播放和控制简单的动画,如果要显示视频和其他媒体内容,那么还是需要使用 Phonon 多媒体框架的。

13.3 多媒体的底层控制

在 Qt 4.6 中新加入了 QtMultimedia 模块来提供一些底层的多媒体功能,比如音频的采集和回放、频谱分析、操作视频帧等。该模块主要由 8 个类组成,这些类及其功能介绍如表 13-3 所列。如果在程序中要使用 QtMultimedia 模块,需要先在.pro 项目文件中添加如下一行代码:

```
QT += multimedia
```

这样就可以在程序中使用 QtMultimedia 模块中的相关类了。

表 13-3 QtMultimedia 模块主要的类

类	介绍
QAbstractVideoBuffer	视频数据的抽象
QAbstractVideoSurface	视频呈现表面(video presentation surfaces)的基类
QAudioDeviceInfo	查询音频设备及其功能的接口
QAudioFormat	存储音频参数信息
QAudioInput	从音频输入设备接收音频数据的接口
QAudioOutput	从音频输出设备输出音频的接口
QVideoFrame	代表一个视频数据帧
QVideoSurfaceFormat	指定一个视频呈现表面的流格式

在 QtMultimedia 模块中与音频控制相关的类有 QAudioFormat、QAudioDeviceInfo、QAudioInput 和 QAudioOutput。其中, QAudioFormat 类用来存储音频参数信息, 音频格式指定了一个音频流中的数据怎样被排列, 可以在音频流上使用 codec() 来指定编码。除了编码, QAudioFormat 还包含了频率、声道数量、样本大小、样本类型和字节顺序等参数, 如表 13-4 所列。

表 13-4 QAudioFormat 类中的音频参数

参数	描述
采样率	每秒音频数据的样本个数, 单位为赫兹(Hz)
声道数量	音频声道数量(通常是单声道或者立体声)
样本大小	每一个样本中存储的数据量(通常是 8 或者 16 位)
样本类型	样本的数值表示(通常是有符号整数、无符号整数或者浮点数)
字节顺序	样本的字节顺序(通常是小端、大端)

QAudioDeviceInfo 类提供了一个用来查询音频设备及其功能的接口, 可以用来查询音频设备, 例如系统中当前可用的声卡或者 USB 耳机等, 音频设备是否可用依赖于

平台和安装的音频插件。还可以查询每个设备所支持的音频格式,各种参数信息可以通过调用 `supportedByteOrders()`、`supportedChannelCounts()`、`supportedCodecs()`、`supportedSampleRates()`、`supportedSampleSizes()` 和 `supportedSampleTypes()` 等函数来获取。如果要使用一个指定的格式,可以使用 `isFormatSupported()` 函数来检查设备是否支持它,还可以使用 `nearestFormat()` 来获取一个与指定格式最接近的可用的格式。`QAudioDeviceInfo` 被 Qt 用来构建与 `QAudioInput` 和 `QAudioOutput` 等音频设备进行通信的类,使用静态函数 `defaultInputDevice()`、`defaultOutputDevice()` 和 `availableDevices()` 可以获得所有可用的设备的一个列表。

`QAudioInput` 类提供了一个用来从音频输入设备接收音频数据的接口,可以为系统默认的音频输入设备构建一个音频输入,也可以使用指定的 `QAudioDeviceInfo` 来创建一个 `QAudioInput`,当创建一个音频输入时,还要指定用于录制的 `QAudioFormat`。在任何时间点,`QAudioInput` 都会在活跃、暂停、停止或空闲 4 种状态的其中一个状态,这些状态使用 `QAudio::State` 枚举变量来指定。可以通过直接调用 `suspend()`、`resume()`、`stop()`、`reset()` 和 `start()` 来改变状态,可以使用 `state()` 来获取当前所在的状态,在状态变化时 `QAudioInput` 还会发射 `stateChanged()` 信号。`QAudioInput` 提供了几种方法来测量自从录制开始所经历的时间,`processedUSecs()` 函数返回音频流写入的长度,单位为微妙,它不包含音频输入中暂停或者空闲的时间;`elapsedUSecs()` 函数返回自从调用 `start()` 函数以后所经过的时间,包含所有状态所用的时间。如果发生了错误,可以使用 `error()` 来获取错误的原因,如果遇到了错误,那么 `QAudioInput` 将会进入停止状态。

`QAudioOutput` 类提供了一个用来向音频设备发送音频数据的接口,该类中的函数与 `QAudioInput` 类中的函数及其用法是相似的。如果要使用滑块来显示播放进度,可以先使用 `setNotifyInterval()` 来设置时间间隔,然后 `QAudioOutput` 会在指定的时间间隔发射 `notify()` 信号,通过关联这个信号来更新滑块的位置。

下面通过一个音频录制与播放的例子来进一步熟悉这些类的应用。(项目源码路径: `src\13\13-3\myRecord`)新建 Qt Gui 应用,名称 `myRecord`,类名为 `MainWindow` 和基类 `QMainWindow` 都保持默认即可。完成后先进入设计模式,向界面上拖入两个 `Push Button` 按钮部件和一个 `Label` 部件,再把两个按钮的显示文本分别改为“开始录制”和“开始播放”。然后进入项目文件 `myRecord.pro`,添加如下一行代码:

```
QT += multimedia
```

完成后按下 `Ctrl+S` 保存该文件。下面再进入 `mainwindow.h` 文件,首先添加头文件和类前置声明:

```
#include <QFile>
#include <QAudio>
class QAudioInput;
class QAudioOutput;
```

再添加私有槽声明：

```
private slots:
    void stopRecording();
    void finishedPlaying(QAudio::State state);
```

然后定义几个私有对象：

```
QFile file;
QAudioInput * audioInput;
QAudioOutput * audioOutput;
```

下面到 mainwindow.cpp 文件中，先添加头文件：

```
#include <QAudioFormat>
#include <QAudioDeviceInfo>
#include <QAudioInput>
#include <QAudioOutput>
#include <QTimer>
```

然后从设计模式分别进入到两个按钮的单击信号对应的槽，更改如下：

```
//开始录制按钮，将音频录制到指定的文件中
void MainWindow::on_pushButton_clicked()
{
    file.setFileName("test.raw");
    file.open(QIODevice::WriteOnly | QIODevice::Truncate);
    //设置音频格式，包括采样率、声道数量、样本大小、编码、字节顺序和样本类型等
    QAudioFormat format;
    format.setFrequency(8000);
    format.setChannels(1);
    format.setSampleSize(8);
    format.setCodec("audio/pcm");
    format.setByteOrder(QAudioFormat::LittleEndian);
    format.setSampleType(QAudioFormat::UnSignedInt);
    //获取默认的音频输入设备，判断是否支持指定的格式，如果不支持则使用一个邻近的格式
    QAudioDeviceInfo info = QAudioDeviceInfo::defaultInputDevice();
    if (!info.isFormatSupported(format)) {
        format = info.nearestFormat(format);
    }
    //设置定时器实现录制 10 秒后自动停止
    QTimer::singleShot(10000, this, SLOT(stopRecording()));
    audioInput = new QAudioInput(format, this);
    audioInput->start(&file);
    ui->label->setText(tr("正在录制……"));
}
```

```
//开始播放按钮,播放指定的文件
void MainWindow::on_pushButton_2_clicked()
{
    file.setFileName("test.raw");
    file.open(QIODevice::ReadOnly);
    QAudioFormat format;
    format.setFrequency(8000);
    format.setChannels(1);
    format.setSampleSize(8);
    format.setCodec("audio/pcm");
    format.setByteOrder(QAudioFormat::LittleEndian);
    format.setSampleType(QAudioFormat::UnSignedInt);
    QAudioDeviceInfo info(QAudioDeviceInfo::defaultOutputDevice());
    if (!info.isFormatSupported(format)) {
        return;
    }
    audioOutput = new QAudioOutput(format, this);
    connect (audioOutput, SIGNAL(stateChanged(QAudio::State)), SLOT(finishedPlaying
        (QAudio::State)));
    audioOutput->start(&file);
    ui->label->setText(tr("正在播放……"));
}
```

下面再定义其他两个槽如下：

```
//停止录制
void MainWindow::stopRecording()
{
    audioInput->stop();
    ui->label->setText(tr("停止录制！"));
    file.close();
    delete audioInput;
}

//停止播放
void MainWindow::finishedPlaying(QAudio::State state)
{
    if (state == QAudio::IdleState) {
        audioOutput->stop();
        ui->label->setText(tr("停止播放"));
        file.close();
        delete audioOutput;
    }
}
```

因为代码中使用了中文，所以还要在主函数中添加相应的代码。运行程序，连接好麦克风后按下录制按钮就可以实现录音了，录制完毕后可以按下播放按钮进行播放。

对该部分的内容，在Qt中提供了Audio Devices、Audio Output和Audio Input几个示例程序，都在Multimedia分类中；还有一个Spectrum Analyzer频谱分析演示程序，该程序比较综合，可以参考一下。

QtMultimedia模块除了可以对音频进行处理以外，还可以对视频进行处理，这主要是通过QAbstractVideoSurface、QVideoSurface、QVideoFrame和QAbstractVideoBuffer几个类实现的。在Qt中对于这几个类的使用也提供了两个示例程序，可以通过Video Graphics Item Example和Video Widget Example关键字在帮助中查看相关内容，这里就不再对这几个类的使用进行介绍了。

13.4 小结

这一章讲述了简单的音频和动画的播放，它们在一些场合还是很有用的。对于音频和视频的底层控制，可以使用QtMultimedia模块，不过如果只是想播放音频和视频，最便捷的方法还是使用下一章要讲述的Phonon多媒体框架。

本书通过分析和研究中文字体设计的基本原理，文中不仅展示了各种字体设计的案例，还深入探讨了字体设计的基本原则。希望读者能够通过本书掌握字体设计的基本知识，并能够在今后的字体设计工作中运用所学的知识。

第 14 章

Phonon 多媒体框架

Phonon 是一个跨平台的多媒体框架，利用它可以在 Qt 应用程序中使用音频和视频等多媒体内容。Qt 使用 Phonon 多媒体框架来播放常见的多媒体格式，既可以播放文件，也可以通过网络播放媒体流。要使用 Phonon 中的类进行编程，需要在.pro 项目文件中添加代码“QT+=phonon”。发布软件时，需要把 Qt 安装目录中 plugins 目录下的 phonon_backend 文件夹与应用程序放在同一目录中一起发布。

本章中只讲解 Phonon 中一些主要内容的应用，对应可参考《Qt 及 Qt Quick 开发实战精解》中的音乐播放器实例，那里会应用到 Phonon 中在本章没有涉及的一些知识点。对应本章的内容，可以在帮助中查看 Phonon multimedia framework 关键字。

14.1 Phonon 多媒体框架的架构

Phonon 包含 3 个基本的概念：媒体对象（Media Objects）、汇点（Sinks）和路径（Paths）。媒体对象用来管理媒体源，例如一个音乐文件，提供开始、停止和暂停等简单的播放控制功能；汇点用来从 Phonon 中输出媒体，例如在部件上渲染视频或者将音频传送到声卡；路径用来连接 Phonon 对象，例如连接一个媒体对象和一个汇点，这样形成的关系图在 Phonon 中被称为媒体图（media graph）。图 14-1 是一个音频流的媒体图，图 14-2 是一个播放带有声音的视频文件的媒体图。

媒体的播放从媒体对象开始，可以对播放进行简单的控制，然后媒体对象将媒体流发送到通过路径与其连接的汇点上，最后汇点使用底层的多媒体设备完成媒体流的播放。

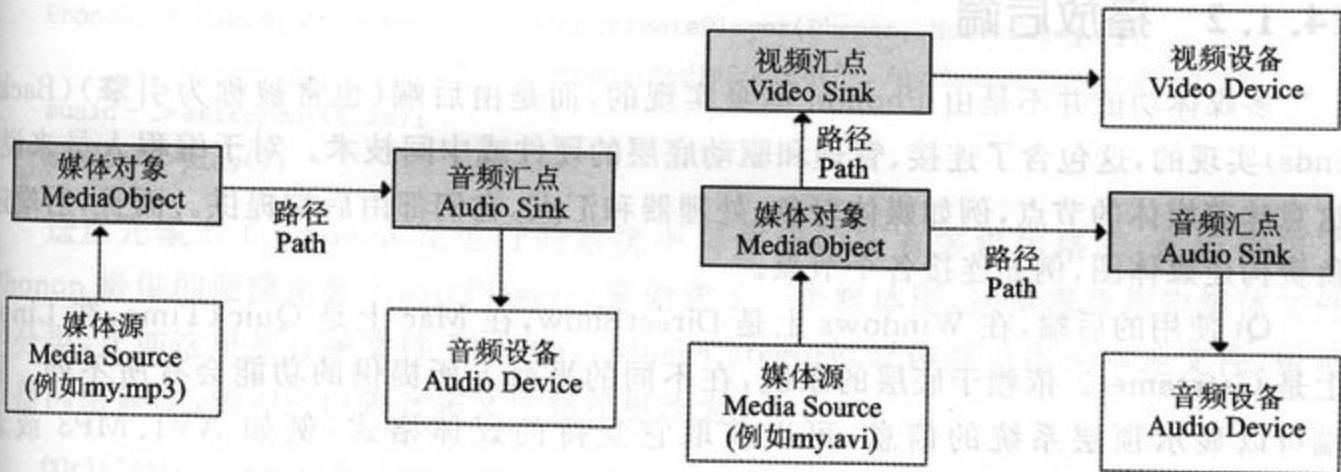


图 14-1 音频流媒体图

图 14-2 播放带有声音的视频的媒体图

14.1.1 Phonon 媒体图中的节点

1. 媒体对象

媒体对象(Media Objects)是 MediaObject 类的一个实例,用来开始、暂停和停止媒体流的播放,可以将它看作一个简单的多媒体播放器。多媒体数据由媒体源提供,媒体源是一个单独的对象,是 MediaSource 的实例,媒体源并不是媒体图中的一部分。媒体源为媒体对象提供原始的数据,媒体数据可以从一个文件中读取,也可以从网络上的媒体流读取,媒体对象可以解析媒体源中的内容。

作为媒体对象的一个补充,Phonon 还提供了一个 MediaController 类,提供了对于给定媒体可选的控制功能,比如 DVD 文件的章节、菜单和标题等都由 MediaController 来管理。

2. 汇点

汇点(Sinks)是一个可以从媒体图中输出媒体的节点,通常是一个渲染设备。在 Phonon 媒体图中汇点的输入来自媒体对象。汇点可以对媒体进行基本的控制,对于一个音频汇点,它代表一个虚拟的音频设备,可以对音量进行控制,也可以设置静音;对于一个视频汇点,比如 VideoWidget,可以在一个 QWidget 部件上渲染视频,并且可以改变亮度、色调,还可以进行视频缩放。

3. 处理器

Phonon 不允许直接操纵媒体流,例如,当把媒体流给定到媒体对象后,不可以在编程中改变媒体流中的字节。不过可以通过处理器节点来实现这个功能,在媒体图中,处理器节点位于媒体对象和对应的汇点之间的路径上。在 Phonon 中,处理器由 Effect 类表示。在渲染过程中插入一个处理器后,它就会作为媒体图的一部分而一直有效,停止时需要删除该处理器。

14.1.2 播放后端

多媒体功能并不是由 Phonon 本身实现的,而是由后端(也常被称为引擎)(Backends)实现的,这包含了连接、管理和驱动底层的硬件或中间技术。对于编程人员来说,这意味着媒体的节点,例如媒体对象、处理器和汇点,它们都由后端提供。而且,后端还负责构建媒体图,例如连接各个节点。

Qt 使用的后端,在 Windows 上是 DirectShow,在 Mac 上是 QuickTime,在 Linux 上是 Gstreamer。依赖于底层的系统,在不同的平台上所提供的功能会有所不同。后端可以显示顶层系统的信息,可以获取它支持的媒体格式,例如 AVI、MP3 或者 OGG 等。

由于各个后端的不同,可能无法设计出支持所有后端的 API,因此,应用程序需要检查当前后端所提供的功能。可以调用 availableMimeTypes() 和 isMimeTypeAvailable() 函数来查询后端可以为哪些 MIME 类型产生节点;当后端的功能发生改变时就会发射 Notifier::capabilitiesChanged() 信号,如果可用的音频设备发生了改变,那么会发射 Notifier::availableAudioOutputDevicesChanged() 信号;要查询实际的音频设备,可以使用 availableAudioOutputDevices(),如果要查询有关单个设备的信息,可以检查它们的 name();视频播放的汇点并没有可选的设备,为了方便,VideoWidget 既作为媒体图中的一个节点,也作为渲染视频输出的部件。要查询各种可用的视频格式,可以使用 isMimeTypeAvailable()。

14.2 播放音频

使用 Phonon 来播放音频,通过调用便捷函数(就是指这个函数使用起来很方便)只需要两行代码就可以完成。当然,如果需要更灵活的控制,也可以手动创建媒体图,然后获取音频的各种信息和进行各种控制。

14.2.1 实现简单的音频播放

(项目源码路径: src\14\14-1\myPhonon1) 新建 Qt Gui 应用,名称为 myPhonon1,类名 MainWindow 和基类 QMainWindow 保持默认即可。完成后,在项目文件 myPhonon1.pro 中添加如下一行代码:

```
QT      + = phonon
```

然后保存该文件。再到 mainwindow.cpp 文件中添加头文件:

```
#include <phonon>
#include <QDebug>
#include <QUrl>
```

并在构造函数中添加如下代码:

```

qDebug() << Phonon::BackendCapabilities::availableMimeTypes();
Phonon::MediaObject * music = Phonon::createPlayer(Phonon::MusicCategory,
    Phonon::MediaSource("../myPhonon1/mysong.mp3"));
music->setParent(this);
music->play();

```

这里先输出了 Phonon 在运行的系统中所支持的所有多媒体格式,然后使用了 Phonon 提供的便捷函数 createPlayer() 来创建了一个媒体图,这里需要指定媒体文件的分类,比如这里是音频文件 Phonon::MusicCategory,媒体源可以是本地文件,也可以是网络媒体,例如可以将音乐文件路径更改为:

```
QUrl("http://www.yafeilinux.com/wp-content/uploads/2010/04/mysong.mp3")
```

createPlayer() 函数返回一个媒体对象,最后调用 play() 函数来播放音频即可。现在运行程序,便可以播放指定的歌曲了。可以看到使用 Phonon 来播放音频是很容易的,也可以使用 MediaObject 类提供的其他函数来实现更多的控制。

14.2.2 创建音频流媒体图

要获得更灵活地控制,可以手动创建自己的媒体图。当播放音频时,需要创建一个媒体对象,然后将它连接到一个音频输出节点,该节点由 AudioOutput 类提供,用来将音频输出到声卡。

(项目源码路径: src\14\14-2\myPhonon1) 将前面在构造函数中添加的代码更改如下:

```

Phonon::MediaObject * mediaObject = new Phonon::MediaObject(this);
mediaObject->setCurrentSource(Phonon::MediaSource("../myPhonon1/mysong.mp3"));
Phonon::AudioOutput * audioOutput = new Phonon::AudioOutput(Phonon::MusicCategory,
    this);
Phonon::Path path = Phonon::createPath(mediaObject, audioOutput);
mediaObject->play();

```

AudioOutput 类用来向音频输出设备发送数据,音频输出需要使用 createPath() 函数连接到媒体对象上。AudioOutput 提供了函数来控制音量的大小,还可以设置静音,而且 Phonon 还使用 AudioOutput 提供了一个音量控制部件 VolumeSlider 类,它是 QWidget 的子类。另外,Phonon 中也提供了播放进度滑块 SeekSlider,该类也是 QWidget 的子类,它需要连接到媒体对象上。

14.2.3 使用音频效果

在播放音频时,可以在路径上插入 Effect 来实现一些音频效果。由于后端的不同,各个平台所支持的音频效果是不同的,可以先获取后端支持的音频效果,然后再使用。

在前面程序的构造函数中继续添加如下代码：

```
//获取可用的音频效果的描述
QList<Phonon::EffectDescription> effectDescriptions =
    Phonon::BackendCapabilities::availableAudioEffects();
qDebug() << effectDescriptions;
Phonon::EffectDescription effectDescription = effectDescriptions.at(5);
//使用指定的音频效果的描述来创建音频效果
Phonon::Effect * effect = new Phonon::Effect(effectDescription);
//在路径中插入音频效果
path.insertEffect(effect);
//创建效果部件,它可以用来更改效果中的参数
Phonon::EffectWidget * effectWidget = new Phonon::EffectWidget(effect);
effectWidget->show();
```

可以使用 availableAudioEffects() 函数来获取可用的音频效果,它返回的是音频效果的描述 EffectDescription 的列表,EffectDescription 用来描述一个音频效果,可以使用它来创建一个相应的音频效果。笔者这里是在 Windows Xp 系统中,上面程序中使用的第 5 个(从 0 开始)音频特效是“Echo”即回音效果。最后使用 path 的 insertEffect() 函数来插入音频效果。对于音频效果的各个参数,可以使用 parameters() 函数获取,然后使用 EffectParameter 类来设置。不过 Phonon 中提供了 EffectWidget 便捷类来管理音频效果的参数,该类是 QWidget 的子类。运行程序,可以发现音乐播放时产生了回音效果。

《Qt 及 Qt Quick 开发实战精解》中介绍了一个音乐播放器的实例,那里会涉及更多的音频播放的应用,所以这里就不再对这部分内容进行讲解了。读者可以查看 Phonon 分类下的 Music Player 示例程序,还可以通过 Capabilities Example 关键字查看另一个示例程序,这个程序可以检索系统中支持的 MIME 类型、音频设备和音频效果。

14.3 播放视频

在 Phonon 中可以使用和播放音频相似的方法来播放视频,同样可以使用便捷方式很容易地实现视频的播放,也可以手动创建媒体图来对视频播放进行更加灵活的控制。

14.3.1 实现简单的视频播放

(项目源码路径: src\14\14-3\myPhonon2)像前面程序 14-1 那样新建项目 myPhonon2。在 mainwindow.cpp 文件中添加头文件 #include <phonon>后,再在构造函数中添加如下代码:

```
Phonon::VideoPlayer * player = new Phonon::VideoPlayer(Phonon::VideoCategory, this);
player ->resize(400, 300);
player ->play(Phonon::MediaSource("../myPhonon2/myVideo.WMV"));
```

可以使用 VideoPlayer 便捷类来实现视频的播放,该类继承自 QWidget,可以像一个窗口部件一样来使用。可以使用 VideoPlayer 类的 play() 函数开始视频的播放,还可以使用 pause() 来暂停播放,stop() 来停止播放,除此之外,该类还可以设置音量大小和播放位置。

14.3.2 创建播放视频的媒体图

播放视频可以使用 VideoWidget 类,该部件可以自动选择可用的设备来播放视频。VideoWidget 并不会播放媒体流中的音频,如果要播放视频中的音频,那么就要创建一个 AudioOutput 节点。

(项目源码路径: src\14\14-4\myPhonon2) 把前面程序构造函数中添加的代码更改如下:

```
Phonon::MediaObject * mediaObject = new Phonon::MediaObject(this);
Phonon::VideoWidget * videoWidget = new Phonon::VideoWidget(this);
Phonon::createPath(mediaObject, videoWidget);
Phonon::AudioOutput * audioOutput = new Phonon::AudioOutput(Phonon::VideoCategory,
                                                               this);
Phonon::createPath(mediaObject, audioOutput);
mediaObject ->setCurrentSource(Phonon::MediaSource("../myPhonon2/myVideo.WMV"));
videoWidget ->resize(400, 300);
mediaObject ->play();
```

这里创建了两个路径,分别用于视频流和音频流,就像图 14-2 所示的那样。创建 VideoWidget 时不需要指定类别,它会自动指定为 VideoCategory 类别,我们需要做的是确保 AudioOutput 也在同一个类别中。

14.3.3 控制视频播放

在前面已经看到,VideoWidget 类提供了一个用来显示视频的部件,该类可以在一个 QWidget 上渲染媒体流中的视频。VideoWidget 提供了一些操作视频流的功能,比如可以改变亮度 brightness()、改变色调 hue()、改变饱和度 saturation() 和改变对比度 contrast()。视频显示大小的改变会被自动处理,也可以使用 aspectRatio 和 scaleMode 属性来控制视频改变大小的方式,这两个属性的取值分别如表 14-1 和表 14-2 所列。默认的,VideoWidget 会使用视频流自身的宽高比。可以使用 setFullScreen() 来进入全屏模式,还可以使用 snapshot() 来获取屏幕快照。

表 14-1 VideoWidget 的宽高比选项

常量	描述
Phonon::VideoWidget::AspectRatioAuto	使用媒体文件自身的宽高比
Phonon::VideoWidget::AspectRatioWidget	使视频的宽高比适应部件的大小
Phonon::VideoWidget::AspectRatio4_3	使宽高比为 4 : 3
Phonon::VideoWidget::AspectRatio16_9	使宽高比为 16 : 9

表 14-2 VideoWidget 的缩放模式选项

常量	描述
Phonon::VideoWidget::FitInView	填充视图时保持宽高比
Phonon::VideoWidget::ScaleAndCrop	视频将被缩放

下面来看一个视频播放器的例子。(项目源码路径: src\14\14-5\myPhonon3)新建 Qt Gui 应用,项目名称为 myPhonon3,基类选择 QWidget,类名为 Widget。完成后先到项目文件 myPhonon.pro 中添加代码:

```
QT      + = phonon
```

然后保存该文件。再到 widget.h 文件中,先添加头文件和类的前置声明:

```
#include <phonon>
class QMenu;
class QAction;
```

再添加几个私有槽声明:

```
private slots:
    void createContextMenu();
    void showContextMenu(const QPoint &pos);
    void aspectChanged(QAction * action);
    void scaleChanged(QAction * action);
```

然后添加两个私有变量:

```
Phonon::VideoWidget * videoWidget;
QMenu * mainMenu;
```

下面进入设计模式,向界面上拖入一个 Frame 部件,然后向 Frame 中拖入 4 个 Label 和 4 个 Horizontal Slider,把 4 个 Horizontal Slider 均进行属性修改: 最小值 minimum 属性改为 -10,最大值 maximum 属性改为 10,tickPosition 属性选择 TicksBelow,tickInterval 属性设置为 10。然后再把 4 个 Label 的显示文本分别改为“亮度:”、“色调:”、“饱和度:”和“对比度:”。最后对 Frame 使用栅格布局,效果如图 14-3 所示。

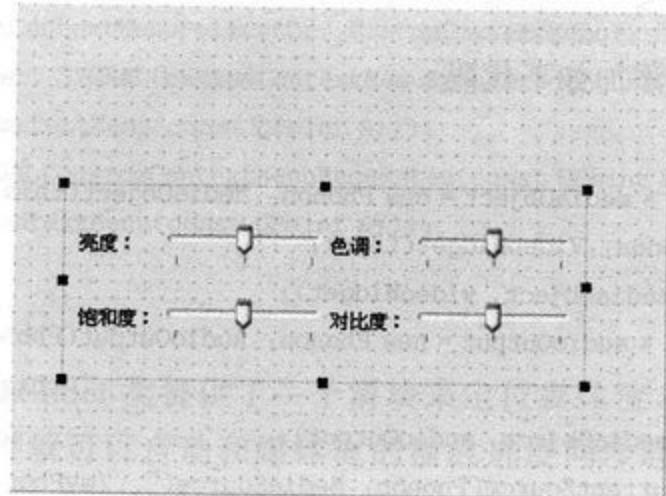


图 14-3 设计界面效果

下面分别进入 4 个 Horizontal Slider 部件 valueChanged(int) 信号对应的槽，更改如下：

```
//更改亮度
void Widget::on_horizontalSlider_valueChanged(int value)
{
    videoWidget->setBrightness(value / 10.0);
}

//更改色调
void Widget::on_horizontalSlider_2_valueChanged(int value)
{
    videoWidget->setHue(value / 10.0);
}

//更改饱和度
void Widget::on_horizontalSlider_3_valueChanged(int value)
{
    videoWidget->setSaturation(value / 10.0);
}

//更改对比度
void Widget::on_horizontalSlider_4_valueChanged(int value)
{
    videoWidget->setContrast(value / 10.0);
}
```

亮度 brightness、色调 hue、饱和度 saturation 和对比度 contrast 这 4 个属性的值都是浮点型的，取值范围在 -1~1 之间，默认值为 0。这里使用了 4 个滑块来设置这 4 个属性的值。下面在 widget.cpp 文件中，添加头文件：

```
#include <QMenu>
#include <QAction>
#include <QVBoxLayout>
```

```
# include <QToolBar>
```

然后在构造函数中添加如下代码：

```
//创建媒体图
```

```
Phonon::MediaObject * mediaObject = new Phonon::MediaObject(this);
videoWidget = new Phonon::VideoWidget(this);
Phonon::createPath(mediaObject, videoWidget);
Phonon::AudioOutput * audioOutput = new Phonon::AudioOutput(Phonon::VideoCategory,
    this);
Phonon::createPath(mediaObject, audioOutput);
mediaObject ->setCurrentSource(Phonon::MediaSource("../myPhonon3/myVideo.WMV"));
```

```
//创建播放进度滑块
```

```
Phonon::SeekSlider * seekSlider = new Phonon::SeekSlider(mediaObject, this);
```

```
//创建工具栏,包含了播放、暂停和停止动作,以及控制音量滑块
```

```
QToolBar * toolBar = new QToolBar(this);
QAction * playAction = new QAction(style() -> standardIcon(QStyle::SP_MediaPlay), tr("播放"), this);
connect(playAction, SIGNAL(triggered()), mediaObject, SLOT(play()));
QAction * pauseAction = new QAction(style() -> standardIcon(QStyle::SP_MediaPause), tr("暂停"), this);
connect(pauseAction, SIGNAL(triggered()), mediaObject, SLOT(pause()));
QAction * stopAction = new QAction(style() -> standardIcon(QStyle::SP_MediaStop), tr("停止"), this);
connect(stopAction, SIGNAL(triggered()), mediaObject, SLOT(stop()));
Phonon::VolumeSlider * volumeSlider = new Phonon::VolumeSlider(audioOutput, this);
volumeSlider -> setSizePolicy(QSizePolicy::Maximum, QSizePolicy::Maximum);
toolBar ->addAction(playAction);
toolBar ->addAction(pauseAction);
toolBar ->addAction(stopAction);
toolBar ->addWidget(volumeSlider);
```

```
//创建布局管理器,将各个部件都添加到布局管理器中
```

```
QVBoxLayout * mainLayout = new QVBoxLayout;
videoWidget ->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
mainLayout ->addWidget(videoWidget);
mainLayout ->addWidget(seekSlider);
mainLayout ->addWidget(toolBar);
mainLayout ->addWidget(ui ->frame);
setLayout(mainLayout);
```

```
//设置 Widget 和 VideoWidget 都使用自定义上下文菜单
```

```

setContextMenuPolicy(Qt::CustomContextMenu);
videoWidget ->setContextMenuPolicy(Qt::CustomContextMenu);
connect (videoWidget, SIGNAL(customContextMenuRequested(const QPoint &)),
         SLOT(showContextMenu(const QPoint &)));
connect (this, SIGNAL(customContextMenuRequested(const QPoint &)),
         SLOT(showContextMenu(const QPoint &)));
//创建上下文菜单
createContextMenu();

```

Phonon 中的 SeekSlider 类提供了一个滑块来定位媒体流,只要与媒体对象 MediaObject 连接后,该部件就可以自动和媒体流的播放进度同步起来,不需要手动进行信号和槽的关联。Phonon 中的 VolumeSlider 部件提供了一个用来控制音频输出设备音量的滑块,该滑块还会默认显示一个可以设置静音的图标,这个图标可以使用 setMuteVisible() 来移除。为了设置视频显示的大小,这里使用函数创建了上下文菜单,下面添加该函数的定义:

```

void Widget::createContextMenu()
{
    mainMenu = new QMenu(this);

    //创建“宽高比”子菜单
    QMenu * aspectMenu = mainMenu ->addMenu(tr("宽高比"));
    QActionGroup * aspectGroup = new QActionGroup(aspectMenu);
    connect(aspectGroup, SIGNAL(triggered(QAction * )), this,
            SLOT(aspectChanged(QAction * )));
    aspectGroup ->setExclusive(true);
    QAction * aspectActionAuto = aspectMenu ->addAction(tr("自动"));
    aspectActionAuto ->setCheckable(true);
    aspectActionAuto ->setChecked(true);
    aspectGroup ->addAction(aspectActionAuto);
    QAction * aspectActionScale = aspectMenu ->addAction(tr("缩放"));
    aspectActionScale ->setCheckable(true);
    aspectGroup ->addAction(aspectActionScale);
    QAction * aspectAction16_9 = aspectMenu ->addAction(tr("16:9"));
    aspectAction16_9 ->setCheckable(true);
    aspectGroup ->addAction(aspectAction16_9);
    QAction * aspectAction4_3 = aspectMenu ->addAction(tr("4:3"));
    aspectAction4_3 ->setCheckable(true);
    aspectGroup ->addAction(aspectAction4_3);

    //创建“缩放模式”子菜单
    QMenu * scaleMenu = mainMenu ->addMenu(tr("缩放模式"));
    QActionGroup * scaleGroup = new QActionGroup(scaleMenu);

```

```

connect(scaleGroup, SIGNAL(triggered(QAction * )), this,
        SLOT(scaleChanged(QAction * )));
scaleGroup ->setExclusive(true);
QAction * scaleActionFit = scaleMenu ->addAction(tr("保持宽高比"));
scaleActionFit ->setCheckable(true);
scaleActionFit ->setChecked(true);
scaleGroup ->addAction(scaleActionFit);
QAction * scaleActionCrop = scaleMenu ->addAction(tr("缩放和裁剪"));
scaleActionCrop ->setCheckable(true);
scaleGroup ->addAction(scaleActionCrop);

//创建“全屏”动作
QAction * fullScreenAction = mainMenu ->addAction(tr("全屏"));
fullScreenAction ->setCheckable(true);
connect(fullScreenAction, SIGNAL(toggled(bool)), videoWidget,
        SLOT(setFullScreen(bool)));
}

```

添加这个函数的目的只是为了简化构造函数，下面添加这个函数中调用到的几个槽的定义：

```

//显示上下文菜单
void Widget::showContextMenu(const QPoint &pos)
{
    mainMenu ->popup(videoWidget ->isFullScreen() ? pos : mapToGlobal(pos));
}

//设置宽高比
void Widget::aspectChanged(QAction * action)
{
    if (action ->text() == tr("16:9"))
        videoWidget ->setAspectRatio(Phonon::VideoWidget::AspectRatio16_9);
    else if (action ->text() == tr("缩放"))
        videoWidget ->setAspectRatio(Phonon::VideoWidget::AspectRatioWidget);
    else if (action ->text() == tr("4:3"))
        videoWidget ->setAspectRatio(Phonon::VideoWidget::AspectRatio4_3);
    else
        videoWidget ->setAspectRatio(Phonon::VideoWidget::AspectRatioAuto);
}

//设置缩放模式
void Widget::scaleChanged(QAction * action)
{
    if (action ->text() == tr("缩放和裁剪"))
        videoWidget ->setScaleMode(Phonon::VideoWidget::ScaleAndCrop);
    else

```

```
videoWidget ->setScaleMode(Phonon::VideoWidget::FitInView);  
}
```

此时运行程序就可以实现视频播放的一些控制功能了。Qt 中还提供了一个多媒体播放器 Media Player 演示程序，可以参考一下。

14.4 小结

本章简要介绍了 Phonon 多媒体框架的架构以及一些基本的应用。可以看到，Phonon 中的类并不像以前 Qt 中的那些类一样使用字母 Q 开头，其实 Phonon 并不是 Qt 原生的东西，关于 Phonon 更多的介绍可以查看其他资料。《Qt 及 Qt Quick 开发实战精解》中的音乐播放器实例中设计了一个功能比较完善的音频播放器程序，涉及了 Phonon 中更多知识的应用，读者可以参考。

数据处理篇

➤ 第 15 章 文件、目录和输入/输出

➤ 第 16 章 模型/视图编程

➤ 第 17 章 数据库和 XML

第 15 章

文件、目录和输入 / 输出

应用程序中经常需要对设备或者文件进行读取或写入,也经常会对本地文件系统中的文件或者目录进行操作。Qt 将这些相关操作的类罗列在了一起,可以在 Qt 帮助 Input/Output and Networking 关键字对应的文档中查看这些类的列表。本章对其中一些常用类的用法进行了简单介绍。

15.1 文件和目录

15.1.1 输入 / 输出设备

QIODevice 类是 Qt 中所有 I/O 设备的基础接口类,为诸如 QFile、QBuffer 和 QTcpSocket 等支持读/写数据块的设备提供了一个抽象接口。QIODevice 类是抽象的,无法被实例化,一般是使用它所定义的接口来提供设备无关的 I/O 功能。

访问一个设备以前,需要使用 open() 函数打开该设备,而且必须指定正确的打开模式。QIODevice 中所有的打开模式由 QIODevice::OpenMode 枚举变量定义,其取值如表 15-1 所列,其中的一些值可以使用按位或符号“|”来同时使用。打开设备后可以使用 write() 或者 putChar() 来进行写入,使用 read()、readLine() 或者 readAll() 进行读取,最后使用 close() 关闭设备。

表 15-1 QIODevice 中的打开模式

常量	描述
QIODevice::NotOpen	设备没有打开
QIODevice::ReadOnly	设备以只读方式打开,这时无法写入
QIODevice::WriteOnly	设备以只写方式打开,这时无法读取
QIODevice::ReadWrite	设备以读写方式打开
QIODevice::Append	设备以附加模式打开,所有的数据都将写入到文件的末尾

常量	描述
QIODevice::Truncate	如果可能,设备在打开前会被截断,设备先前的所有内容都将丢失
QIODevice::Text	当读取时,行结尾终止符会被转换为‘\n’;当写入时,行结尾终止符会被转换为本地编码,例如在 Win32 上是‘\r\n’
QIODevice::Unbuffered	绕过设备所有的缓冲区

QIODevice 会区别两种类型的设备:随机存取设备和顺序存储设备。

- 随机存取设备支持使用 seek() 函数来定位到任意的位置。文件中的当前位置可以使用 pos() 函数来获取。这样的设备有 QFile、QBuffer 和 QTemporaryFile 等。
- 顺序存储设备不支持定位到任意的位置,数据必须一次性读取。pos() 和 size() 等函数无法在操作顺序设备时使用。这样的设备有 QTcpSocket、QUdpSocket 和 QProcess 等。

可以在程序中使用 isSequential() 函数来判断设备的类型。这里提到的 QTcpSocket 和 QUdpSocket 类会在第 18 章重点讲解,QProcess 类会在第 19 章讲解,所以在这一章就不再对这些类以及顺序设备的操作进行过多讲解。

通过子类化 QIODevice 可以为自己的 I/O 设备提供相同的接口,要子类化 QIODevice 只需要重新实现 readData() 和 writeData() 两个函数。QIODevice 的一些子类,例如 QFile 和 QTcpSocket,都使用了内存缓冲区进行数据的中间存储,这样减少了设备的访问次数,使得 getChar() 和 putChar() 等函数非常快速,而且可以在内存缓冲区上进行操作而不用直接在设备上进行操作。但是,一些特定的 I/O 操作,使用缓冲区却无法很好地工作,这时就可以在调用 open() 函数打开设备时使用 QIODevice::Unbuffered 模式来绕过所有的缓冲区。

15.1.2 文件操作

1. 文件 QFile

QFile 类提供了一个用于读/写文件的接口,是一个可以用来读/写文本文件、二进制文件和 Qt 资源的 I/O 设备。QFile 可以单独使用,也可以和 QTextStream 或者 QDataStream 一起使用,这样会更方便。

一般在构建 QFile 对象时便指定文件名,当然也可以使用 setFileName() 在其他任何时间进行设置。无论在哪种操作系统上,文件名路径中的文件分隔符都需要使用‘/’符号。可以使用 exists() 来检查文件是否存在,remove() 来删除一个文件。更多与文件系统相关的高级操作在 QFileinfo 和 QDir 类中提供,这两个类会在后面的内容中讲到。

一个文件可以使用 open() 来打开,使用 close() 来关闭,使用 flush() 来刷新。文件

的数据读/写一般使用 QDataStream 或者 QTextStream 来完成,不过也可以使用继承自 QIODevice 类的一些函数,比如 read()、readLine()、readAll() 和 write(),还有一次只操作一个字符的 getChar()、putChar() 和 ungetChar() 等函数。可以使用 size() 函数来获取文件的大小,使用 seek() 来定位到文件的任意位置,使用 pos() 来获取当前的位置,使用 atEnd() 来判断是否到达了文件的末尾。

2. 文件信息 QFileinfo

QFileinfo 类提供了与系统无关的文件信息,包括文件的名称和在文件系统中的位置(路径),文件的访问权限以及它是否是一个目录或者符号链接等。QFileinfo 也可以获取文件的大小和最近一次修改/读取的时间等,还可以获取 Qt 资源的相关信息。

QFileinfo 可以使用相对(relative)路径或者绝对(absolute)路径来指向一个文件,使用 isRelative() 函数可以判断一个 QFileinfo 对象使用的是相对路径还是绝对路径,还可以使用 makeAbsolute() 来将一个相对路径转换为绝对路径。QFileinfo 指向的文件可以在 QFileinfo 对象构建时设置,或者在以后使用 setFile() 来设置。可以使用 exists() 来查看文件是否存在,使用 size() 可以获取文件的大小。文件的类型可以使用 isFile()、isDir() 和 isSymLink() 来获取, symLinkTarget() 函数可以返回符号链接指向的文件的名称。

可以分别使用 path() 和 fileName() 来获取文件的路径和文件名,还可以使用 baseName() 来获取文件名中的基本名称,使用 suffix() 来获取文件名的后缀,使用 completeSuffix() 来获取复合后缀。文件的日期可以使用 created()、lastModified() 和 lastRead() 来返回;访问权限可以使用 isReadable()、isWritable() 和 isExecutable() 来获取;文件的所有权可以使用 owner()、ownerId()、group() 和 groupId() 来获取;还可以使用 permission() 函数将文件的访问权限和所有权一次性读取出来。

(项目源码路径: src\15\15-1\myFile)新建 Qt4 控制台应用,名称为 myFile, 创建完成后将 main.cpp 文件的内容更改为:

```
#include <QtCore/QCoreApplication>
#include <QFileinfo>
#include <QTextCodec>
#include <QStringList>
#include <QDateTime>
#include <QDebug>

int main(int argc, char * argv[])
{
    QCoreApplication a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    //以只写方式打开,如果文件不存在,那么会创建该文件
    QFile file("myfile.txt");
}
```

```

if (! file.open(QIODevice::WriteOnly | QIODevice::Text))
    qDebug() << file.errorString();
file.write("helloQt! \nyafeilinux");
file.close();

//获取文件信息
QFileInfo info(file);
qDebug() << QObject::tr("绝对路径: ") << info.absoluteFilePath() << endl
    << QObject::tr("文件名: ") << info.fileName() << endl
    << QObject::tr("基本名称: ") << info.baseName() << endl
    << QObject::tr("后缀: ") << info.suffix() << endl
    << QObject::tr("创建时间: ") << info.created() << endl
    << QObject::tr("大小: ") << info.size();

//以只读方式打开
if (! file.open(QIODevice::ReadOnly | QIODevice::Text))
    qDebug() << file.errorString();
qDebug() << QObject::tr("文件内容: ") << endl << file.readAll();
qDebug() << QObject::tr("当前位置: ") << file.pos();
file.seek(0);
QByteArray array;
array = file.read(5);
qDebug() << QObject::tr("前 5 个字符: ") << array
    << QObject::tr("当前位置: ") << file.pos();
file.seek(15);
array = file.read(5);
qDebug() << QObject::tr("第 16-20 个字符: ") << array;
file.close();

return a.exec();
}

```

这里先使用 `QIODevice::WriteOnly` 只写模式和 `QIODevice::Text` 文本模式将文件打开, 当使用写入模式打开文件时, 如果文件不存在, 那么就会自动创建一个; `QIODevice::Text` 模式可以在写入时将‘\n’转换为 Windows 上的‘\r\n’, 比如这里写入的字符串“helloQt! \nyafeilinux”, 在使用 `QIODevice::Text` 时大小为 20, 如果不使用则大小为 19, 而且可以看到, 只有在应用 `QIODevice::Text` 时, 使用 Windows 的记事本打开生成的 `myfile.txt` 文件时才会将“helloQt!”和“yafeilinux”分两行进行显示。下面使用 `QFileInfo` 来获取了文件的一些信息, 然后使用只读模式再次打开文件。这里先使用 `readAll()` 函数读取了文件的所有内容, 这时将会处于文件的末尾, 这里使用了 `seek(0)` 来定位到文件的开始, 0 位置在第一个字符的前面。对应每一个 `open()` 函数, 一定要在操作完文件后使用 `close()` 函数将文件关闭。

3. 临时文件 QTemporaryFile

QTemporaryFile 类是一个用来操作临时文件的 I/O 设备, 可以安全地创建一个唯一的临时文件。当调用 open() 函数时便会创建一个临时文件, 临时文件的文件名可以保证是唯一的, 当销毁 QTemporaryFile 对象时, 该文件会被自动删除掉。在调用 open() 函数时, 默认会使用 QIODevice::ReadWrite 模式, 可以像下面的代码这样来使用 QTemporaryFile 类:

```
QTemporaryFile file;
if (file.open()) {
    //在这里对临时文件进行操作
}
```

在调用了 close() 函数后重新打开 QTemporaryFile 是安全的, 只要 QTemporaryFile 的对象没有被销毁, 那么唯一的临时文件就会一直存在而且由 QTemporaryFile 内部保持打开。临时文件默认会生成在系统的临时目录里, 这个目录的路径可以使用 QDir::tempPath() 来获取。

15.1.3 目录操作

1. 目录 QDir

QDir 类用来访问目录结构及其内容, 可以操作路径名、访问路径和文件的相关信息以及操作底层的文件系统, 还可以访问 Qt 的资源系统。Qt 使用“/”作为通用的目录分隔符和 URLs 的目录分隔符, 如果使用“/”作为目录分隔符, Qt 会自动转换路径来符合底层的操作系统。QDir 可以使用相对路径或者绝对路径来指向一个文件, 使用绝对路径例如:

```
QDir("/home/user/Documents")
QDir("C:/Documents and Settings")
```

在 Windows 上, 当使用第二个例子中的路径访问文件时, 会将其转换为“C:\Documents and Settings”。下面是一个相对路径的例子:

```
QDir("images/landscape.png")
```

可以使用 isRelative() 和 isAbsolute() 来判断一个 QDir 是否使用了相对路径或者绝对路径, 还可以使用 makeAbsolute() 来将一个相对路径转换为绝对路径。一个目录的路径可以使用 path() 函数获取, 使用 setPath() 函数可以设置新的路径, 使用 absolutePath() 函数可以获取绝对路径。目录名可以使用 dirName() 函数获取, 这通常返回绝对路径中的最后一个元素, 然而如果 QDir 代表当前目录, 那么会返回“.”。目录的路径也可以使用 cd() 和 cdUp() 函数来改变, 当使用一个存在的目录的名字来调用 cd() 后, QDir 对象就会转换到指定的目录; 而 cdUp() 会跳转到父目录, cdUp() 与 cd(..) 是等效的。可以使用 mkdir() 来创建目录, 使用 rename() 来进行重命名, 使用

`rmdir()`来删除目录。可以使用 `exists()` 函数来测试指定的目录是否存在, 使用 `isReadable()` 和 `isRoot()` 等函数来测试目录的属性。使用 `refresh()` 函数可以重新读取目录的数据。

目录中包含很多条目, 如文件、目录和符号链接等。一个目录中的条目数目可以使用 `count()` 来返回, 所有条目的名称列表可以使用 `entryList()` 来获取, 如果需要每一个条目的信息, 可以使用 `entryInfoList()` 函数来获取一个 `QFileInfo` 对象的列表。可以使用 `filePath()` 和 `absoluteFilePath()` 来获取一个目录中的文件和目录的路径, `filePath()` 会返回指定文件或目录与当前 `QDir` 对象所在路径的相对路径, `absoluteFilePath()` 返回绝对路径。文件可以使用 `remove()` 函数来移除, 但是目录只能使用 `rmdir()` 函数来移除。可以应用一个名称过滤器 (name filters) 来使用通配符 (wildcards) 指定一个模式进行文件名的匹配, 一个属性过滤器可以选取条目的属性并且可以区分文件和目录, 还可以设定排序顺序。名称过滤器就是一个字符串列表, 可以使用 `setNameFilters()` 函数来设置, 例如下面的代码在 `QDir` 上使用了 3 个名称过滤器来确保只有以通常用于 C++ 源文件的扩展名来结尾的文件才会被列出:

```
QStringList filters;
filters << " *.cpp" << " *.cxx" << " *.cc";
dir.setNameFilters(filters);
```

属性过滤器由按位或组合在一起的过滤器组成, 可以使用 `setFilter()` 来设置。排序顺序使用 `setSorting()` 来设置, 它需要指定按位或组合在一起的排序标志 `QDir::SortFlags`, 所有的排序标志可以在 `QDir` 的帮助文档中进行查看, 包含了按名称、按时间、按大小等排序方式。可以使用 `match()` 函数来测试一个文件名是否匹配一个过滤器。当设置好过滤器和排序标志后, 就可以调用 `entryList()` 或者 `entryInfoList()` 来获取指定条件的条目了。

要访问一些常见的目录, 可以使用一些静态函数来完成, 它们可以返回 `QDir` 对象或者 `QString` 类型的绝对路径, 这些函数如表 15-2 所列。

表 15-2 `QDir` 中常用目录的获取函数

返回类型为 <code>QDir</code>	返回类型为 <code>QString</code>	返回值
<code>current()</code>	<code>currentPath()</code>	应用程序的工作目录
<code>home()</code>	<code>homePath()</code>	用户的 home 目录
<code>root()</code>	<code>rootPath()</code>	root 根目录
<code>temp()</code>	<code>tempPath()</code>	系统存放临时文件的目录

使用 `setCurrent()` 静态函数也可以设置应用程序的工作目录。如果要查找包含应用程序可执行文件的目录, 可以使用 `QCoreApplication::applicationDirPath()` 函数。使用 `drives()` 函数可以返回系统的根目录, 在 Windows 上会返回 `QFileInfo` 对象的列表, 其中包含了“C:/”和“D:/”等, 在其他操作系统上会返回只包含根目录(例如“/”)的

一个列表。路径中包含“.”元素表示路径中的当前目录，“..”元素表示父目录，可以使用 canonicalPath() 函数来返回一个规范的路径，该路径中不包含符号链接和冗余的“.”和“..”元素。cleanPath() 函数可以移除路径中冗余的“/”和“.”或者“..”，例如“./local”将变为“local”，“local/.. /bin”将变为“bin”，“/local/usr/.. /bin”将变为“/local/bin”等。静态函数 toNativeSeparators() 可以将路径中的“/”分隔符转换为适合底层操作系统的分隔符，如 toNativeSeparators("C:/winnt/system32") 返回“C:\winnt\sys-tem32”。

2. 文件系统监视器 QFileSystemWatcher

QFileSystemWatcher 类提供了一个用来监控文件和目录修改的接口，通过监视一个指定路径的列表来监控文件系统中文件和目录的改变。调用 addPath() 来监视一个指定的文件或者目录，多个路径可以使用 addPaths() 函数来添加，现有的路径可以使用 removePath() 和 removePaths() 函数来移除。QFileSystemWatcher 会检测每一个添加到它上面的路径，添加到其上的文件的路径可以使用 files() 来获取，目录的路径可以使用 directories() 函数来获取。

当文件被修改、重命名或者移除后，会发射 fileChanged() 信号，相似的，当目录或者它的内容被修改或者移除后，会发射 directoryChanged() 信号。需要注意的是，当文件被重命名或者移除后，或者当目录被移除后，QFileSystemWatcher 都会停止监视。

(项目源码路径：src\15\15-2\myDir) 新建 Qt Gui 应用，项目名称为 myDir，基类选择 QMainWindow，类名为 MainWindow。完成后首先到设计模式，往界面上拖入一个 List Widget 部件。然后到 mainwindow.h 文件中添加头文件 #include <QFileSystemWatcher>，再添加一个私有对象的定义：

```
QFileSystemWatcher myWatcher;
```

然后添加一个私有槽声明：

```
private slots:  
    void showMessage(const QString &path);
```

下面到 mainwindow.cpp 文件中添加头文件 #include <QDir>，然后在构造函数中添加如下代码：

```
// 将监视器的信号和自定义的槽进行关联  
connect(&myWatcher, SIGNAL(directoryChanged(QString)),  
        this, SLOT(showMessage(QString)));  
connect(&myWatcher, SIGNAL(fileChanged(QString)),  
        this, SLOT(showMessage(QString)));  
// 显示出当前目录下的所有.h 文件  
QDir myDir(QDir::currentPath());  
myDir.setNameFilters(QStringList("*.h"));  
ui->listWidget->addItem(myDir.absolutePath() + tr("目录下的.h文件有:"));
```

```

ui->listWidget->addItems(myDir.entryList());
//创建目录，并将其加入到监视器中
myDir.mkdir("mydir");
myDir.cd("mydir");
ui->listWidget->addItem(tr("监视的目录:") + myDir.absolutePath());
myWatcher.addPath(myDir.absolutePath());
//创建文件，并将其加入到监视器中
QFile file(myDir.absolutePath() + "/myfile.txt");
if (file.open(QIODevice::WriteOnly)) {
    QFileInfo info(file);
    ui->listWidget->addItem(tr("监视的文件:") + info.absoluteFilePath());
    myWatcher.addPath(info.absoluteFilePath());
    file.close();
}

```

最后添加 showMessage() 函数的定义：

```

//显示文件或目录改变信息
void MainWindow::showMessage(const QString &path)
{
    QDir dir(QDir::currentPath() + "/mydir");
    //如果是目录发生了改变
    if (path == dir.absolutePath()) {
        ui->listWidget->addItem(dir.dirName() + tr("目录发生改变:"));
        ui->listWidget->addItems(dir.entryList());
    } else { //如果是文件发生了改变
        ui->listWidget->addItem(path + tr("文件发生改变!"));
    }
}

```

因为代码中使用了中文，所以还要在主函数中添加相应的代码。现在运行程序，然后到生成的 mydir 目录中执行新建文件夹、新建文件、修改 myfile.txt 文件的内容等操作，查看输出的结果。

15.2 文本流和数据流

15.2.1 使用文本流读/写文本文件

QTextStream 类提供了一个方便的接口来读/写文本，可以在 QIODevice、QByteArray 和 QString 上进行操作。使用 QTextStream 的流操作符，可以方便地读/写单词、行和数字。对于生成文本，QTextStream 对字段填充、对齐和数字格式提供了格式选项支持。例如：

```

QFile data("output.txt");
if (data.open(QFile::WriteOnly | QFile::Truncate)) {
    QTextStream out(&data);
    //写入 "Result: 3.14      2.7      "
    out << "Result: " << qSetFieldWidth(10) << left << 3.14 << 2.7;
}

```

QTextStream 提供的所有格式选项可以在该类的帮助文档中进行查看。除了使用 QTextStream 类的构造函数来设置设备外,还可以使用 setDevice() 或者 setString() 来设置 QTextStream 要操作的设备或者字符串。可以使用 seek() 来定位到一个指定位置,使用 atEnd() 判断是否还有可以读取的数据。如果调用了 flush() 函数, QTextStream 会清空写缓冲中的所有数据,并且调用设备的 flush() 函数。

在内部, QTextStream 使用了一个基于 Unicode 的缓冲区, QTextStream 使用 QTextCodec 来自动支持不同的字符集。默认的, 使用 QTextCodec::codecForLocale() 返回的编码来进行读写, 也可以使用 setCodec() 函数来设置编码。

使用 QTextStream 来读取文本文件一般使用 3 种方式:

① 调用 readLine() 或者 readAll() 进行一块接着一块的读取, 例如下面代码实现了对文本文件进行分行读取:

```

QFile file("in.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;
QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    //下面可以对读取的一行字符串进行处理
}

```

② 一个单词接着一个单词。QTextStream 支持到 QString、QByteArray 和 char * 缓冲区的流, 单词由空格分开, 而且可以自动跳过前导空格。

③ 一个字符接着一个字符, 使用 QChar 或者 char 类型的流。这种方式经常在解析文件、使用独立的字符编码和行结束语义时方便输入处理。可以通过调用 skipWhiteSpace() 来跳过空格。

默认的, 当从文本流中读取数字时, QTextStream 会自动检测数字的基数表示。例如, 如果数字以“0x”开头, 它将被假定为十六进制形式; 如果以数字 1~9 开头, 那么它将被假定为十进制形式等。也可以使用 dec 等流操作符或者 setIntegerBase() 来设置整数基数, 从而停止自动检测。

在 Qt 中提供了一个 Find Files Dialog 示例程序, 其中应用到了 QTextStream、 QFile 和 QDir 等类, 它在 Dialogs 分类中可以参考。

15.2.2 使用数据流读/写二进制数据

QDataStream 类实现了将 QIODevice 的二进制数据串行化。一个数据流就是一

个二进制编码信息流,完全独立于主机的操作系统、CPU 和字节顺序。数据流也可以读/写未编码的原始二进制数据。QDataStream 类可以实现 C++ 基本数据类型的串行化,比如 char、short、int 和 char * 等。而串行化更复杂的数据,是通过将数据分解为基本的数据类型来完成的。

将二进制数据写入到数据流中:

```
 QFile file("file.dat");
file.open(QIODevice::WriteOnly);
//要将串行化后的数据输入到 file 中
QDataStream out(&file);
//串行化字符串
out << QString("the answer is");
//串行化整数
out << (qint32)42;
```

从数据流中读取二进制数据:

```
 QFile file("file.dat");
file.open(QIODevice::ReadOnly);
//从 file 中读取串行化的数据
QDataStream in(&file);
QString str;
qint32 a;
//提取“the answer is”和 42
in >> str >> a;
```

写到数据流中的每一个条目都是使用一个预定义的格式写入的,这个格式依赖于条目的类型。支持的 Qt 类型包括 QBrush、QColor、QDateTime、 QFont、QPixmap、QString、QVariant 和很多其他格式,Qt 帮助的 Serializing Qt Data Types 关键字对应的文档中列出了支持数据流的所有 Qt 类型的完整列表。

从 Qt 1.0 开始便有了相应的 QDataStream 的二进制格式,而且会继续发展来反映 Qt 中的变化。当输入或输出复杂数据类型时,确保使用相同的数据流版本(version())来进行读取和写入是非常重要的。对于基本的 C++ 数据类型没有这个要求。如果既要实现向前兼容,又要实现向后兼容,可以在应用程序中对数据流的版本号进行硬编码:

```
 stream.setVersion(QDataStream::Qt_4_0);
```

如果要使用一个新的二进制数据格式,例如在自己应用程序中创建的一个文档的文件格式,这就需要在数据流的前面写入一个简短的数据头,它包含了一个 magic number(幻数或魔术,用来标志文件格式的常数)和一个版本号。例如:

```
 QFile file("file.xxx");
file.open(QIODevice::WriteOnly);
```

```

QDataStream out(&file);

//写入幻数和版本号
out << (quint32)0xA0B0C0D0;
out << (qint32)123;

out.setVersion(QDataStream::Qt_4_0);

//写入数据
out << lots_of_interesting_data;

```

这里幻数可以是一个自定义数字,它是 32 位的。下面来读取该数据流:

```

 QFile file("file.xxx");
file.open(QIODevice::ReadOnly);
QDataStream in(&file);

//读取幻数
quint32 magic;
in >> magic;
if (magic != 0xA0B0C0D0)
    return XXX_BAD_FILE_FORMAT;
//读取版本
qint32 version;
in >> version;
if (version < 100)
    return XXX_BAD_FILE_TOO_OLD;
if (version > 123)
    return XXX_BAD_FILE_TOO_NEW;
if (version <= 110)
    in.setVersion(QDataStream::Qt_3_2);
else
    in.setVersion(QDataStream::Qt_4_0);
//读取数据
in >> lots_of_interesting_data;
if (version >= 120)
    in >> data_new_in_XXX_version_1_2;
in >> other_interesting_data;

```

这里就是根据幻数和版本号来判断文件格式是否正确,应该使用哪种数据流版本。代码中的 XXX_BAD_FILE_FORMAT 等可以是宏定义的一些字符串。

可以在串行化数据时选择使用哪种字节顺序,默认的设置是大端(MSB 在前),如果将它改为小端会破坏可移植性(除非读取时也改变为小端)。也可以直接向数据流写入和读取原始二进制数据,可以使用 readRawData() 将数据从数据流中读取到一个预

先分配的 `char *` 中,同样,可以使用 `writeRawData()`将数据写入到数据流。注意,需要自己完成对数据的编码和解码。

15.3 其他相关类

15.3.1 应用程序设置

`QSettings` 类提供了持久的与平台无关的应用程序设置。用户通常期望应用程序可以记住它们的设置,比如窗口大小和位置等。这些信息在 Windows 上一般存储在系统注册表中,在 Mac OS X 上存储在 XML 偏好文件中,在 Unix 系统中,大多数应用程序使用 INI 文本文件。`QSettings` 是对这些技术的一个抽象,可以使用一种可移植的方式来保存和恢复应用程序的设置,也支持自定义存储类型。`QSettings` 的 API 是基于 `QVariant` 的,可以用来保存大多数的基于值的类型,如 `QString`、`QRect` 和 `QImage` 等。

关于 `QSettings` 的更多内容,可以参考该类的帮助文档。在《Qt 及 Qt Quick 开发实战精解》的 1.4.1 小节使用了该类,可以参考;Qt 中提供了一个 `Settings Editor` 示例程序,在 Tools 分类中,也可以参考一下。

15.3.2 统一资源定位符

1. 统一资源定位符 `QUrl`

`QUrl` 类提供了一个方便的接口来操作 URLs,URL 是 Uniform Resource Locator 的缩写,被称为统一资源定位符或者网页网址。一个 URL 的标准格式如下:

```
protocol://hostname[:port]/path/?query#fragment
```

其中, `protocol` 用来指定传输协议,比如 `http` 和 `ftp` 等;`hostname` 用来指定存放资源的服务器的域名系统主机名或者 IP 地址,在主机名前面还可以包含连接到服务器所需要的用户名和密码(`username:password`);`port` 用来指定端口号,可选,省略时使用默认的端口号,比如 `http` 的默认端口号是 80;`path` 用来指定主机上的目录或者文件地址,路径中可以使用“/”分隔符;`query` 用来设置查询参数,可选,参数间使用“&”符号隔开;`fragment` 用来指定网络资源中的片断。例如下面的 URL:

```
http://www.yafeilinux.com/?tag=afeilinux
```

用来查询在 `www.yafeilinux.com` 上的 `tag` 标签为 `yafeilinux` 的网页。`QUrl` 可以解析和构建编码或者未编码格式的 URLs,它也支持国际化域名(IDNs)。

可以在构造函数中传递一个 `QString` 来初始化 `QUrl`,或者使用 `setUrl()` 和 `setEncodedUrl()`。URLs 可以被表示为两种格式:编码和未编码。未编码的格式适合向用户展示,而编码格式一般用于发送到 web 服务器。一个 URL 也可以一部分一部分地

构造,可以使用 `setScheme()` 来设置协议;使用 `setUserName()` 来设置用户名;使用 `setPassword()` 来设置密码;使用 `setHost()` 来设置主机;使用 `setPort()` 来设置端口;使用 `setPath()` 来设置路径;使用 `setEncodedQuery()` 来设置查询字符串;使用 `setFragment()` 来设置片断。还可以使用一些方便的函数来设置,如 `setAuthority()` 可以一次性设置用户名、密码、主机和端口;`setUserInfo()` 可以一次性设置用户名和密码。

2. 统一资源定位符信息 QUrlInfo

`QUrlInfo` 类存储了 URLs 的相关信息。URLs 可以被检索的信息包括 `name()` 文件名、`permissions()` 权限、`owner()` 拥有者、`group()` 组、`size()` 大小、`lastModified()` 最后修改日期、`lastRead()` 最后被读取的日期、`isDir()` 是否是目录、`isFile()` 是否是文件、`isSymLink()` 是否是符号链接、`isWritable()` 是否可写、`isReadable()` 是否可读、`isExecutable()` 是否可执行。

关于 `QUrl` 和 `QUrlInfo` 类的使用,可以在帮助中参考 `FTP Example` 关键字对应的示例程序。

15.3.3 Qt 资源

`QResource` 类提供了接口来直接读取资源文件,用来表示一组数据的对象,该组数据涉及了一个单一的资源实体。`QResource` 可以使用原始的格式来直接访问字节,这种直接访问允许不使用缓冲拷贝或间接来读取数据。`QResource` 背后的数据和它的孩子通常被编译成一个应用程序库,也可以在运行时加载一个资源,当在运行时进行加载时,资源文件会作为一个很大的数据集进行加载,然后通过引用到资源树分块进行输出。

`QResource` 也可以使用一个绝对路径进行加载,绝对路径可以使用文件系统的表示法,以一个“/”字符开始,或者使用资源表示法,以“:”字符开始。`QResource` 代表的文件将会被备份,这些数据也可以使用 `qCompress()` 来进行压缩,对应的使用 `qUncompress()` 来解压缩。

一个资源可以留在一个应用程序的二进制文件外面,等运行需要它时再使用 `registerResource()` 来加载,传递给 `registerResource()` 的资源文件必须是 `rcc` 生成的二进制资源。关于二进制资源的内容可以在 Qt 帮助中参考 `The Qt Resource System` 关键字。

15.3.4 缓冲区

`QBuffer` 类为 `QByteArray` 提供了一个 `QIODevice` 接口,它允许使用 `QIODevice` 接口来访问 `QByteArray`,这里 `QByteArray` 被视为一个标准的随机访问的文件。默认的,当创建一个 `QBuffer` 时会自动在内部创建一个 `QByteArray` 缓冲区,可以直接调用 `buffer()` 来访问这个缓冲区,也可以调用 `setBuffer()` 来使用现有的 `QByteArray`,或者向 `QBuffer` 的构造函数中传递一个 `QByteArray`。

调用 open() 函数来打开一个缓冲区, 然后使用 write() 或者 putChar() 对缓冲区进行写入, 使用 read()、readLine()、readAll() 或者 getChar() 来读取它。size() 函数可以返回缓冲区的当前大小, 可以使用 seek() 函数来定位到缓冲区的一个指定位置, 最后结束访问缓冲区时, 要调用 close() 函数。下面的代码片段展示了使用 QDataStream 和 QBuffer 来向 QByteArray 写入数据:

```
QByteArray byteArray;
QBuffer buffer(&byteArray);
buffer.open(QIODevice::WriteOnly);
QDataStream out(&buffer);
out << QApplication::palette();
```

下面是怎样从 QByteArray 中读取数据:

```
QPalette palette;
QBuffer buffer(&byteArray);
buffer.open(QIODevice::ReadOnly);
QDataStream in(&buffer);
in >> palette;
```

当有新的数据到达了缓冲区, QBuffer 会发射 readyRead() 信号, 通过关联这个信号, 可以使用 QBuffer 来存储临时的数据, 而后再对它们进行处理。

可以在 Qt 的帮助中查看 Shared Memory Example 示例程序, 其中应用到了 QBuffer 类。

15.4 小结

本章对 Qt 中的输入输出以及文件和目录的操作进行了简单的介绍, 这些内容不是很复杂, 不过在应用程序中经常要用到。读者应该掌握一些常见的操作, 这样在编程过程中就有可能会很容易解决一些比较棘手的问题。

第 16 章

模型 / 视图编程

应用程序中往往要存储大量的数据，并对它们进行处理，然后可以通过各种形式显示给用户，用户需要时还可以对数据进行编辑。Qt 中的模型/视图架构就是用来实现大量数据的存储、处理及其显示的。Qt 4 中引入了一组新的项视图类，它们使用一个模型/视图架构来管理数据和它们展示给用户的方式之间的关系。这种架构引入的功能分离思想为开发者定制项目的显示提供了高度的灵活性，而且还提供了一个标准的模型接口来允许很大范围的数据源使用已经存在的项目视图。对应本章内容，可以在帮助中查看 Model/View Programming 关键字。

16.1 模型 / 视图架构

MVC(Model-View-Controller)是一种起源于 Smalltalk 的设计模式，经常用于创建用户界面。MVC 包含了 3 个组件：模型(Model)是应用对象，用来表示数据；视图(View)是模型的用户界面，用来显示数据；控制(Controller)定义了用户界面对用户输入的反应方式。在 MVC 之前，用户界面设计都是将这 3 种组件集成在一起，MVC 将它们分离开，从而提高了灵活性和重用性。

将视图和控制两种组件结合起来就形成了模型/视图架构。这同样将数据的存储和数据向用户的展示进行了分离，但提供了更为简单的框架。数据和界面进行分离，使得相同的数据在多个不同的视图中显示成为可能，而且还可以创建新的视图，而不需要改变底层的数据框架。为了对用户输入进行灵活处理，还引入了委托(Delegate，也被称为代理)的概念，使用它可以定制数据的渲染和编辑方式。模型/视图的整体架构如图 16-1 所示。其

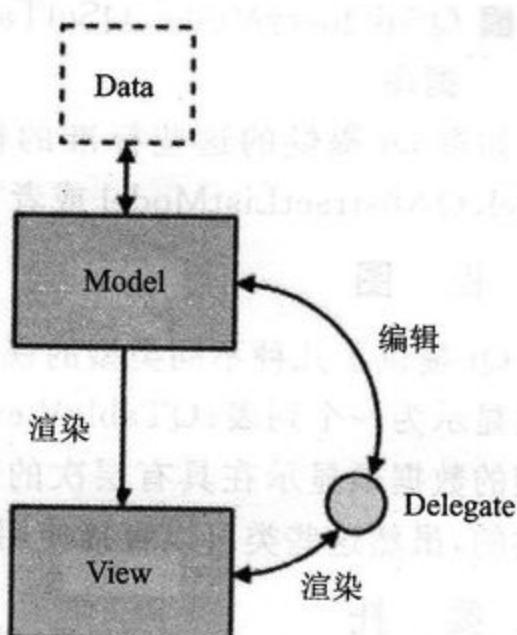


图 16-1 模型 / 视图架构

中,模型与数据源进行通信,为架构中的其他组件提供了接口。视图从模型中获得模型索引(Model Index),模型索引用来表示数据项。在标准的视图中,委托渲染数据项,当编辑项目时,委托使用模型索引直接与模型进行通信。

16.1.1 组成部分

大体上,模型/视图架构中的众多类可以分为3组:模型、视图和委托。其中每一个组件都使用了一个抽象基类来定义,提供了一些通用接口和一些功能的默认实施。模型、视图、委托之间使用信号和槽来实现通信:

- 当数据源的数据发生改变时,模型发出信号告知视图;
- 当用户与显示的项目交互时,视图发出信号来提供交互信息;
- 当编辑项目时,委托发出信号,告知模型和视图编辑器的状态。

1. 模型

所有的模型都基于 `QAbstractItemModel` 类,这个类定义了一个接口,可以供视图和委托来访问数据。数据本身并不是一定要存储在模型中,也可以存储在一个数据结构、一个独立的类、文件、数据库或者应用程序的其他的一些组件中。

`QAbstractItemModel` 为数据提供了一个十分灵活的接口来处理各种视图,这些视图可以将数据表现为表格(table)、列表(list)和树(tree)等形式。然而,当要实现一个新的模型时,如果它基于列表或者表格的数据结构,那么可以使用 `QAbstractListModel` 和 `QAbstractTableModel` 类,因为它们为一些常见的功能提供了默认的实现。这些类都可以被子类化来提供模型,从而支持特殊类型的列表和表格。

Qt 提供了一些现成的模型来处理数据项:

- `QStringListModel` 用来存储一个简单的 `QString` 项目列表;
- `QStandardItemModel` 管理复杂的树型结构数据项,每一个数据项可以包含任意的数据;
- `QFileSystemModel` 提供了本地文件系统中文件和目录的信息;
- `QSqlQueryModel`、 `QSqlTableModel` 和 `QSqlRelationalTableModel` 用来访问数据库。

如果 Qt 提供的这些标准的模型无法满足需要,还可以子类化 `QAbstractItemModel`、`QAbstractListModel` 或者 `QAbstractTableModel` 来创建自定义的模型。

2. 视图

Qt 提供了几种不同类型的视图,它们都完全实现了各自的功能:`QListView` 将数据项显示为一个列表;`QTableView` 将模型中的数据显示在一个表格中;`QTreeView` 将模型的数据项显示在具有层次的列表中。这些类都是基于 `QAbstractItemView` 抽象基类的,虽然这些类可以直接使用,它们也可以被子类化来提供定制的视图。

3. 委托

在模型/视图框架中,`QAbstractItemDelegate` 是委托的抽象基类。从 Qt 4.4 开

始，默认的委托实现由 QStyledItemDelegate 类提供，这也被用作 Qt 标准视图的默认委托。然而，QStyledItemDelegate 和 QItemDelegate 是相互独立的，只能选择其一来为视图中的项目绘制和提供编辑器。它们的主要不同就是，QStyledItemDelegate 使用当前的样式来绘制它的项目，因此，当要实现自定义的委托或者要和 Qt 样式表一起应用时，建议使用 QStyledItemDelegate 作为基类。

16.1.2 简单的例子

前面讲述了模型/视图架构的整体框架，也涉及了模型、视图和委托等概念。模型/视图框架中的内容很繁杂，为了让读者更好地理解，在进一步讲解之前，先来看一个简单的例子。

Qt 中的 QFileSystemModel 类提供了一个保持文件系统信息的模型，它并不包含任何的数据项目，而是代表了本地文件系统中的文件和目录。QFileSystemModel 可以和 QListView 或者 QTreeView 一起使用来显示一个目录中内容。下面的例子中将分别使用树型和列表两种视图来显示同一个模型的数据。（项目源码路径：src\16\16-1\modelView1）新建空的 Qt 项目，项目名称为 modelView1，完成后往项目中添加新的 main.cpp 文件，并更改其内容如下：

```
#include < QApplication >
#include <QFileSystemModel>
#include <QTreeView>
#include <QListView>

int main( int argc, char * argv[] )
{
    QApplication app( argc, argv );

    // 创建文件系统模型
    QFileSystemModel model;

    // 指定要监视的目录
    model.setRootPath( QDir::currentPath() );

    // 创建树型视图
    QTreeView tree;

    // 为视图指定模型
    tree.setModel( &model );

    // 指定根索引
    tree.setRootIndex( model.index( QDir::currentPath() ) );
}
```

```

//创建列表视图
QListView list;
list.setModel(&model);
list.setRootIndex(model.index(QDir::currentPath()));
tree.show();
list.show();

return app.exec();
}

```

这里先创建了 QFileSystemModel 文件系统模型,然后为其指定了要监视的目录,这样该模型便可以表示相应的文件和目录。后面分别创建了树型视图和列表视图,并为它们指定了模型和根索引。可以看到,视图类与其他的窗口部件类的使用是相同的。可以看到,现在已经可以显示当前目录中的内容了。这个程序中并没有指定委托,关于委托的使用将在本章后面的内容中讲到。

16.2 模型类

在模型/视图架构中,模型提供了一个标准的接口供视图和委托来访问数据。在 Qt 中,这个标准的接口使用 QAbstractItemModel 类来定义。无论数据项是怎样存储在何种底层数据结构中,QAbstractItemModel 的子类都会以层次结构来表示数据,这个结构中包含了数据项表。视图按照这种约定来访问模型中的数据项,但是这不会影响数据的显示,视图可以使用任何形式将数据显示出来。当模型中的数据发生变化时,模型会通过信号和槽机制告知与其相关联的视图。

16.2.1 基本概念

常见的 3 种模型分别是列表模型、表格模型和树模型,如图 16-2 所示。

1. 模型索引

为了确保数据的表示与数据的获取相分离,Qt 引入了模型索引的概念。每一块可以通过模型获取的数据都使用一个模型索引来表示,视图和委托都使用这些索引来请求数据项并显示。这样,只有模型需要知道怎样获取数据,被模型管理的数据类型可以广泛定义。模型索引包含一个指针,指向创建它们的模型,当使用多个模型时可以避免混淆。

模型索引由 QModelIndex 类提供,它是对一块数据的临时引用,可以用来检索或者修改模型中的数据。因为模型随时可能对内部的结构进行重新组织,这样模型索引可能失效,所以不需要也不应该存储模型索引。如果需要对一块数据进行长时间的引用,必须使用 QPersistentModelIndex 创建模型索引。如果要获得一个数据项的模型索引,必须指定模型的 3 个属性:行号、列号和父项的模型索引,例如:

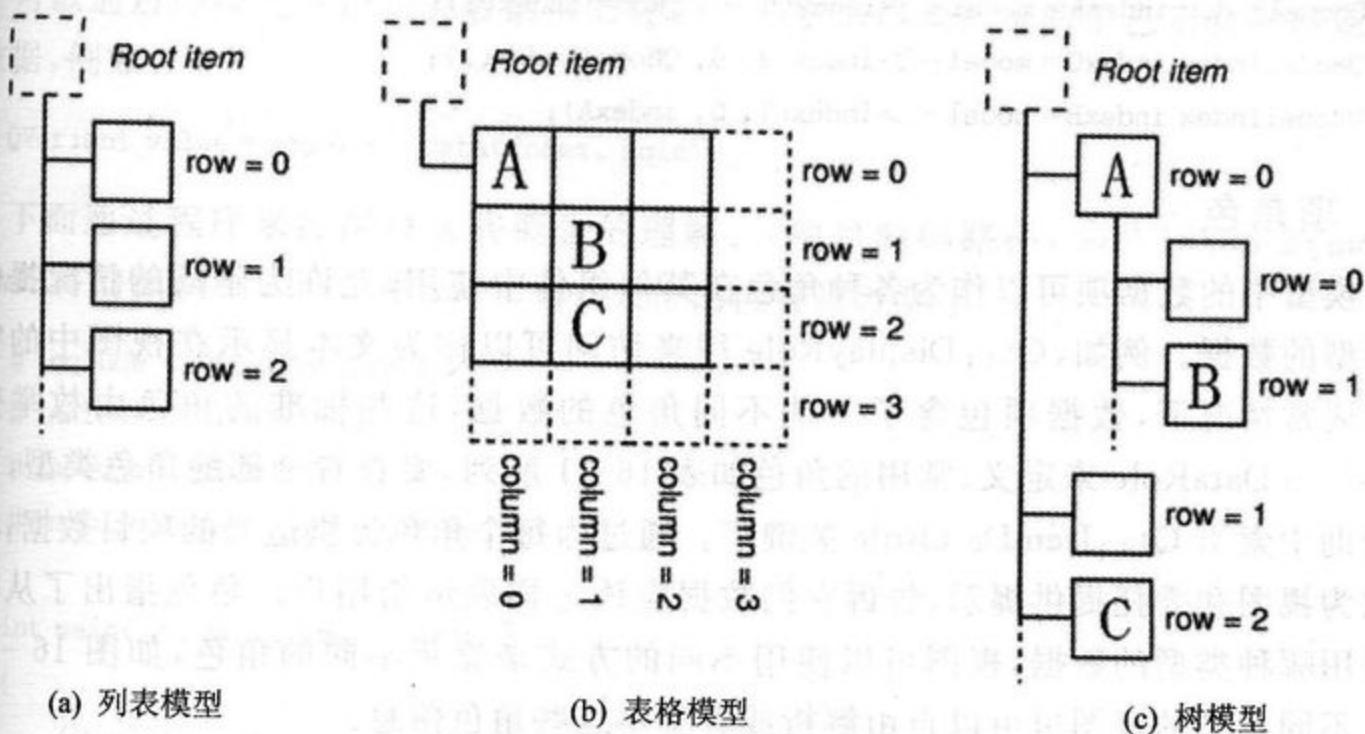


图 16-2 常见的 3 种模型的示意图

```
QModelIndex index = model -> index(row, column, parent);
```

其中, row、column 和 parent 分别代表了这 3 个属性。

2. 行和列

在最基本的形式中,一个模型可以通过把它看作一个简单的表格来访问,这时每个数据项可以使用行号和列号来定位。但这并不意味着在底层的数据块是存储在数组结构中的,使用行号和列号只是一种约定,以确保各组件间可以相互通信。

行号和列号都是从 0 开始的,在图 16-2 中可以看到,列表模型和表格模型的所有数据项都是以根项(Root item)为父项的,这些数据项都可以称为顶层数据项(Top level item),在获取这些数据项的索引时,父项的模型索引可以用 QModelIndex() 表示。例如图 16-2 中的 Table Model 中的 A、B、C 这 3 项的模型索引可以用如下代码获取:

```
QModelIndex indexA = model -> index(0, 0, QModelIndex());
QModelIndex indexB = model -> index(1, 1, QModelIndex());
QModelIndex indexC = model -> index(2, 1, QModelIndex());
```

3. 父项

前面讲述的类似于表格的接口对于在使用表格或者列表时是非常理想的,但是,像树视图一样的结构需要模型提供一个更加灵活的接口,因为每一个数据项都可能成为其他数据项表格的父项,一个树视图中的顶层数据项也可能包含其他的数据项列表。当为模型项请求一个索引时,必须提供该数据项父项的一些信息。前面讲到,顶层数据项可以使用 QModelIndex() 作为父项索引,但是在树模型中,如果一个数据项不是顶层数据项,那么就要指定它的父项索引。例如图 16-2 中的 Tree Model 中 A、B、C 这 3 项的模型索引可以使用如下代码获得:

```

QModelIndex indexA = model ->index(0, 0, QModelIndex());
QModelIndex indexC = model ->index(2, 0, QModelIndex());
QModelIndex indexB = model ->index(1, 0, indexA);

```

4. 项角色

模型中的数据项可以作为各种角色在其他组件中使用,允许为不同的情况提供不同类型的数据。例如,Qt::DisplayRole 用来访问可以作为文本显示在视图中的字符串。通常情况下,数据项包含了一些不同角色的数据,这些标准的角色由枚举变量 Qt::ItemDataRole 来定义,常用的角色如表 16-1 所列,要查看全部的角色类型,可以在帮助中索引 Qt::ItemDataRole 关键字。通过为每个角色提供适当的项目数据,模型可以为视图和委托提供提示,告诉它们数据应该怎样展示给用户。角色指出了从模型中引用哪种类型的数据,视图可以使用不同的方式来显示不同的角色,如图 16-3 所示。不同类型的视图也可以自由解析或者忽略这些角色信息。

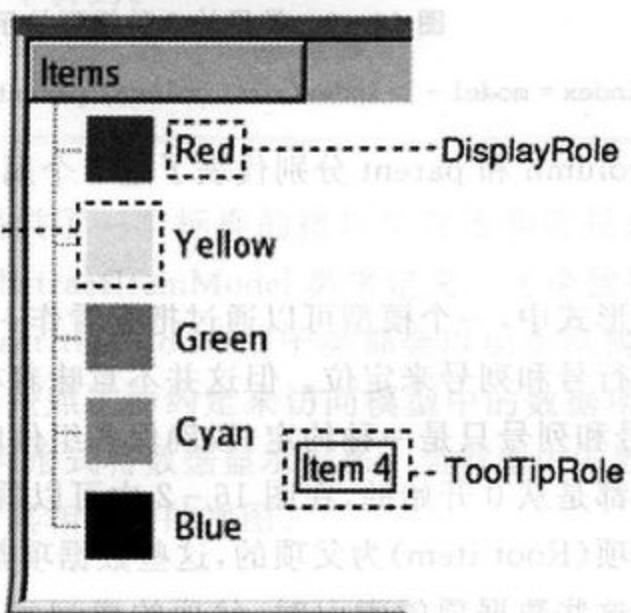


图 16-3 项角色示意图

表 16-1 常用的角色类型

常量	描述
Qt::DisplayRole	数据被渲染为文本(数据为 QString 类型)
Qt::DecorationRole	数据被渲染为图标等装饰(数据为 QColor、QIcon 或者 QPixmap 类型)
Qt::EditRole	数据可以在编辑器中进行编辑(数据为 QString 类型)
Qt::ToolTipRole	数据显示在数据项的工具提示中(数据为 QString 类型)
Qt::StatusTipRole	数据显示在状态栏中(数据为 QString 类型)
Qt::WhatsThisRole	数据显示在数据项的“What’s This?”模式下(数据为 QString 类型)
Qt::SizeHintRole	数据项的大小提示,将会应用到视图(数据为 QSize 类型)

可以通过向模型指定相关数据项对应的模型索引以及特定的角色来获取需要类型的数据,例如:

```
QVariant value = model->data(index, role);
```

下面通过程序来加深对这些概念的理解。(项目源码路径: src\16\16-2\modelView1)将前面例程 16-1 中 main.cpp 文件的内容更改如下:

```
#include < QApplication >
#include < QTreeView >
#include < QDebug >
#include < QStandardItemModel >

int main(int argc, char * argv[])
{
    QApplication app(argc, argv);

    //创建标准项模型
    QStandardItemModel model;

    //获取模型的根项(Root Item),根项是不可见的
    QStandardItem * parentItem = model.invisibleRootItem();

    //创建标准项 item0,并设置显示文本,图标和工具提示
    QStandardItem * item0 = new QStandardItem;
    item0->setText("A");
    QPixmap pixmap0(50,50);
    pixmap0.fill("red");
    item0->setIcon(QIcon(pixmap0));
    item0->setToolTip("indexA");

    //将创建的标准项作为根项的子项
    parentItem->appendRow(item0);

    //将创建的标准项作为新的父项
    parentItem = item0;

    //创建新的标准项,它将作为 item0 的子项
    QStandardItem * item1 = new QStandardItem;
    item1->setText("B");
    QPixmap pixmap1(50,50);
    pixmap1.fill("blue");
    item1->setIcon(QIcon(pixmap1));
    item1->setToolTip("indexB");
```

```

parentItem ->appendRow(item1);

//创建新的标准项,这里使用了另一种方法来设置文本、图标和工具提示
QStandardItem * item2 = new QStandardItem;
QPixmap pixmap2(50,50);
pixmap2.fill("green");
item2 ->setData("C", Qt::EditRole);
item2 ->setData("indexC", Qt::ToolTipRole);
item2 ->setData(QIcon(pixmap2), Qt::DecorationRole);
parentItem ->appendRow(item2);

//在树视图中显示模型
QTreeView view;
view.setModel(&model);
view.show();

//获取 item0 的索引并输出 item0 的子项数目,然后输出了 item1 的显示文本和工具提示
 QModelIndex indexA = model.index(0, 0, QModelIndex());
qDebug() << "indexA row count: " << model.rowCount(indexA);
QModelIndex indexB = model.index(0, 0, indexA);
qDebug() << "indexB text: " << model.data(indexB, Qt::EditRole).toString();
qDebug() << "indexB toolTip: "
      << model.data(indexB, Qt::ToolTipRole).toString();
return app.exec();
}

```

这里使用了标准项模型 QStandardItemModel,该类提供了一个通用的模型来存储自定义的数据。QStandardItemModel 中的项由 QStandardItem 类提供,该类为项目的创建提供了很多方便的函数,例如设置图标的 setIcon() 函数等。当然,也可以不使用这些函数,而是使用 setData() 函数,并且指定项角色,如程序中创建 item2 就是使用的这种方法。在程序中可以看到,获取模型的大小可以使用 rowCount() 和 columnCount() 等函数;可以使用模型索引来访问模型中的项目,但是需要指定其行号、列号和父模型索引;当要访问顶层项目时,父模型索引可以使用 QModelIndex() 来表示;如果项目包含不同角色的数据,那么获取数据时要指定相应的项角色。

16.2.2 创建新的模型

前面已经讲述了 QFileListModel 和 QStandardItemModel 两个模型的使用,在这一小节中将创建一个新的模型来探索模型/视图架构的基本原则。当要为已经存在的数据结构创建一个新的模型,需要考虑使用哪种类型的模型来为数据提供接口。如果数据结构可以表示为项目列表或者表格,那么可以子类化 QAbstractListModel 或者 QAbstractTableModel,因为它们为很多功能提供了非常合适的默认实现。但是,如果

底层数据结构只能表示为具有层次的树结构,那么就需要子类化 QAbstractItemModel。本小节先创建一个基于字符串列表的只读模型,以 QAbstractListModel 作为基类,然后再为其添加编辑、插入行和删除行功能。

1. 创建只读模型

(项目源码路径: src\16\16-3\myModel)首先创建空的 Qt 项目,项目名称为 myModel。完成后往项目中添加新的 C++ 类,类名为 QStringListModel,基类为 QAbstractListModel,类型信息为“继承自 QObject”。完成后,在 stringlistmodel.h 文件中将类的定义更改为:

```
#include <QAbstractListModel>
#include <QStringList>

class QStringListModel : public QAbstractListModel
{
    Q_OBJECT
public:
    QStringListModel(const QStringList &strings, QObject *parent = 0)
        : QAbstractListModel(parent), QStringList(strings) {}
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const;
private:
    QStringList QStringList;
};
```

这里实现的模型是一个简单的、非层次结构的、只读的数据模型,基于标准的 QStringListModel 类。该模型使用了一个 QStringList 作为内部的数据源,这是因为 QAbstractItemModel 本身是不存储任何数据的,它仅仅提供了一些接口来供视图访问数据。在模型类的定义中,除了构造函数以外,只需要实现两个函数: rowCount() 和 data(),前者返回模型的行数,后者返回指定的模型索引的数据项。这里还实现了 headerData() 函数,它可以在树和表格视图的表头显示一些内容。注意,因为现在的模型是非层次结构的,所以不需要考虑父子关系。但是,如果模型是层次结构的,那么还需要实现 index() 和 parent() 函数。下面到 stringlistmodel.cpp 文件里面添加几个函数的实现:

```
#include "stringlistmodel.h"
int QStringListModel::rowCount(const QModelIndex &parent) const
{
    return QStringList.count();
```

因为这个模型是非层次结构的,可以忽略掉模型索引对应的父项目,所以这里只需要简单的返回字符串列表中的字符串个数即可。默认的,继承自 QAbstractListModel 的模型只包含一列,所以这里不需要实现 columnCount() 函数。

```
QVariant StringListModel::data(const QModelIndex &index, int role) const
{
    if (! index.isValid())
        return QVariant();

    if (index.row() >= stringList.size())
        return QVariant();

    if (role == Qt::DisplayRole)
        return stringList.at(index.row());
    else
        return QVariant();
}
```

对于视图中的项目,我们想要显示字符串列表中的字符串,这个函数就是用来返回对应索引参数的数据项的。当提供的索引是有效的,行号在字符串列表的大小范围之内,而且需要的角色是我们支持的角色之一时,返回一个有效的 QVariant。

```
QVariant StringListModel::headerData ( int section, Qt::Orientation orientation, int
role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    if (orientation == Qt::Horizontal)
        return QString("Column %1").arg(section);
    else
        return QString("Row %1").arg(section);
}
```

像 QTreeView 和 QTableView 等一些视图,在显示项目数据的同时还会显示表头。这里想在带有表头的视图中,可以在标头中显示行号和列号。并不是所有的视图都会显示标头,一些视图会隐藏它们。不过,还是建议实现 headerData() 函数来提供数据的相关信息。

现在一个只读的数据模型类就创建完成了,为了测试该模型是否可以正常使用,往项目中添加新的 main.cpp 文件,并更改内容如下:

```
#include <QApplication>
#include "stringlistmodel.h"
#include <QListView>
```

```
#include <QTableView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QStringList list;
    list << "a" << "b" << "c";
    QStringListModel model(list);

    QListView listView;
    listView.setModel(&model);
    listView.show();

    QTableView tableView;
    tableView.setModel(&model);
    tableView.show();

    return app.exec();
}
```

这里创建了 QStringListModel 模型，并为其指定了一个字符串列表来提供数据，然后分别在两个不同类型的视图中进行显示。运行程序，效果如图 16-4 所示。

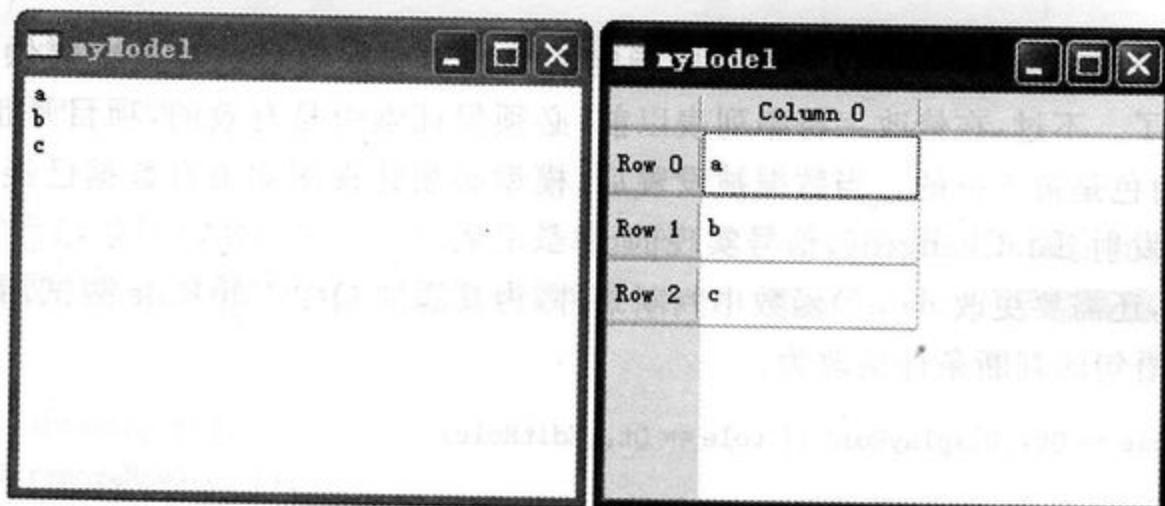


图 16-4 创建新的模型运行效果

2. 添加编辑功能

为了使模型可以编辑，需要更改 `data()` 函数，然后实现另外两个函数：`flags()` 和 `setData()`。（项目源码路径：src\16\16-4\myModel）首先在 `stringlistmodel.h` 文件中添加两个函数的声明：

```
Qt::ItemFlags flags(const QModelIndex &index) const;
```

```
bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole);
```

然后到 stringlistmodel.cpp 文件中添加这两个函数的实现代码：

```
Qt::ItemFlags StringListModel::flags(const QModelIndex &index) const
{
    if (!index.isValid())
        return Qt::ItemIsEnabled;
    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
}
```

委托在创建编辑器以前会检测项目是否是可编辑的，模型必须让委托知道它的项目是可编辑的，这里为模型中的每一个项目返回一个正确的标识来达到这个目的。注意，我们并不需要知道委托是怎样执行真正的编辑操作的，而只需要为委托向模型中设置数据提供一条途径，这个是通过 setData() 函数实现的：

```
bool StringListModel::setData(const QModelIndex &index, const QVariant &value, int role)
{
    if (index.isValid() && role == Qt::EditRole) {
        stringList.replace(index.row(), value.toString());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}
```

在这个模型中，字符串列表里对应指定的模型索引的项目被参数中提供的 value 值替换掉了。不过，在修改字符串列表以前，必须保证索引是有效的，项目是正确的类型，而且角色是被支持的。当数据被设置后，模型必须让视图知道有数据已经改变了，这是通过发射 dataChanged() 信号实现的。

最后，还需要更改 data() 函数中判断条件，为其添加 Qt::EditRole 测试，就是将第三个 if() 语句的判断条件更改为：

```
if (role == Qt::DisplayRole || role == Qt::EditRole)
```

现在运行程序，可以发现已经能够更改编辑数据了。

3. 插入和删除行

要想实现在模型中插入和删除行，需要重新实现 insertRows() 和 removeRows() 两个函数。（项目源码路径：src\16\16-5\myModel）首先在 stringlistmodel.h 文件中添加这两个函数的声明：

```
bool insertRows(int position, int rows, const QModelIndex &index = QModelIndex());
bool removeRows(int position, int rows, const QModelIndex &index = QModelIndex());
```

然后到 stringlistmodel.cpp 文件中添加这两个函数的实现代码：

```
bool QStringListModel::insertRows(int position, int rows, const QModelIndex &parent)
{
    beginInsertRows(QModelIndex(), position, position + rows - 1);
    for (int row = 0; row < rows; ++row) {
        QStringList.insert(position, "");
    }
    endInsertRows();
    return true;
}
```

因为模型中的行对应着列表中的字符串，这个函数就是在指定位置的前面添加了指定数量的空字符串。父索引是用来决定在模型的什么地方添加行的，因为在这里只有单一的顶层字符串列表，所以只需要向列表中添加空的字符串。模型首先要调用 `beginInsertRows()` 函数来告知其他组件指定的行将要发生改变，这个函数指定了将要插入的第一个和最后一个新行的行号，以及它们父项的模型索引。当改变完字符串以后，调用了 `endInsertRows()` 函数来完成操作，而且告知其他组件该模型的大小发生了变化。

```
bool QStringListModel::removeRows(int position, int rows, const QModelIndex &parent)
{
    beginRemoveRows(QModelIndex(), position, position + rows - 1);
    for (int row = 0; row < rows; ++row) {
        QStringList.removeAt(position);
    }
    endRemoveRows();
    return true;
}
```

删除行的操作与前面插入行的操作是相似的，这里不再介绍。下面在 `main.cpp` 文件的主函数中添加代码来测试这两个函数，在 `return app.exec()` 一行代码前添加如下代码：

```
model.insertRows(3, 2);
model.removeRows(0, 1);
```

这样便在模型最后添加两个空数据项，并删除了模型的第一个数据项，现在可以运行程序查看效果。

可以看到，在模型类中就是将各种操作转换为对具体数据源的操作，从而对外提供了一个统一的接口供用户使用。如果还想设计一个更复杂的模型，可以参考 Qt 自带的 Simple Tree Model 示例程序。

16.3 视图类

16.3.1 基本概念

在模型/视图架构中,视图包含了模型中的数据项并将它们呈现给用户,而数据的表示方法可能与底层用于存储数据项的数据结构完全不同。这种内容与表现的分离之所以能够实现,是因为使用了 `QAbstractItemModel` 提供的一个标准模型接口、一个标准视图接口以及使用了模型索引提供的一种通用的方法来表示数据。视图通常管理从模型获取数据的整体布局,它们可以自己渲染独立的数据项,也可以使用委托来处理渲染和编辑。

除了呈现数据,视图还处理项目间的导航,以及项目选择的某些方面,表 16-2 和表 16-3 分别罗列了视图中的选择行为(`QAbstractItemView::SelectionBehavior`)和选择模式(`QAbstractItemView::SelectionMode`),我们会在后面的内容中看到它们的应用。视图也实现了一些基本的用户接口特性,比如上下文菜单和拖放等。视图可以为项目提供默认的编辑实现,当然也可以和委托一起来提供一个自定义的编辑器。不指定模型也可以构造一个视图,但是在视图显示有用的信息以前,必须为其提供一个模型。

表 16-2 视图类的选择行为

常量	描述
<code>QAbstractItemView::SelectItems</code>	选择单个项目
<code>QAbstractItemView::SelectRows</code>	只选择行
<code>QAbstractItemView::SelectColumns</code>	只选择列

表 16-3 视图类的选择模式

常量	描述
<code>QAbstractItemView::SingleSelection</code>	当用户选择一个项目,则所有已经选择的项目将成为未选择状态,而且用户无法在已经选择的项目上单击来取消选择
<code>QAbstractItemView::ContiguousSelection</code>	如果用户在单击一个项目的同时按着 Shift 键,则所有在当前项目和单击项目之间的项目都将被选择或者取消选择,这依赖于被单击项目的状态
<code>QAbstractItemView::ExtendedSelection</code>	具有 ContiguousSelection 的特性,而且还可以按着 Ctrl 键进行不连续的选择
<code>QAbstractItemView::MultiSelection</code>	用户选择一个项目时不影响其他已经选择的项目
<code>QAbstractItemView::NoSelection</code>	项目无法被选择

对于一些视图,例如 QTableView 和 QTreeView,在显示项目的同时还可以显示表头。这是通过 QHeaderView 类实现的,它们使用 QAbstractItemModel::headerData() 函数从模型中获取数据,然后一般使用一个标签来显示表头信息。可以通过子类化 QHeaderView 类来设置标签的显示。

Qt 中已经提供了 QListView、QTableView 和 QTreeView 这 3 个现成的视图,不过都是使用规范的格式显示数据。如果想要实现条形图或者饼状图等特殊显示方式就要重新实现视图类了,这将在《Qt 及 Qt Quick 开发实战精解》中的数据管理系统实例中介绍。

16.3.2 处理项目选择

在模型/视图架构中对项目的选择提供了非常方便的处理方法。视图中被选择的信息存储在一个 QItemSelectionModel 实例中,这样被选择的项目模型索引便保持在一个独立的模型中,与所有的视图都是独立的。当在一个模型上设置多个视图时,就可以实现在多个视图之间共享选择。

选择由选择范围指定,只需要记录每一个选择范围开始和结束的模型索引即可,非连续的选择可以使用多个选择范围来描述。选择可以看作是在选择模型中保存的一个模型索引集合,最近的项目选择被称为当前选择。

1. 当前项目和被选择的项目

视图中总是有一个当前项目和一个被选择的项目,两者是两个独立的状态。在同一时间,一个项目可以既是当前项目,同时也是被选择的项目。视图负责确保总是有一个项目作为当前项目来实现键盘导航。当前项目和被选择的项目的区别如表 16-4 所列。

表 16-4 当前项目和被选择的项目的区别

当前项目	被选择的项目
只能有一个当前项目	可以有多个被选择的项目
使用键盘导航键或者鼠标按键可以改变当前项目	项目是否处于被选择状态,取决于几个预先定义好的模式,例如单项选择、多重选择等
如果按下 F2 键或者双击都可以编辑当前项目	当前项目可以通过指定一个范围来一起被使用
当前项目会显示焦点矩形	被选择的项目会使用选择矩形来表示

当操作选择时,可以将 QItemSelectionModel 看作一个项目模型中所有项目的选择状态的一个记录。一旦设置了一个选择模型,所有的项目集合都可以被选择、取消选择或者切换选择状态,而不需要知道哪一个项目已经被选择了。所有被选择项目的索引都可以被随时进行检索,其他的组件也可以通过信号和槽机制来获取选择模型的改变信息。

2. 使用选择模型

标准的视图类中提供了默认的选择模型,可以在大多数的应用中直接使用。属于一个视图的选择模型可以使用这个视图的 selectionModel() 函数获得,而且还可以在多个视图之间使用 setSelectionModel() 函数来共享该选择模型,所以一般是不需要重新构建一个选择模型的。下面通过例子来看一下选择模型的使用。

(项目源码路径: src\16\16-6\ mySelection)新建 Qt Gui 应用,项目名称为 mySelection,类名和基类保持 MainWindow 和 QMainWindow 不变。完成后先在 mainwindow.h 文件中添加类的前置声明:

```
class QTableView;
```

然后再添加一个私有对象定义:

```
QTableView * tableView;
```

下面到 mainwindow.cpp 文件中添加头文件:

```
# include <QStandardItemModel>
# include <QTableView>
# include <QDebug>
```

然后在构造函数中添加如下内容:

```
QStandardItemModel * model = new QStandardItemModel(7, 4, this);
for (int row = 0; row < 7; + + row) {
    for (int column = 0; column < 4; + + column) {
        QStandardItem * item = new QStandardItem(QString(" %1").arg(row * 4 + column));
        model - >setItem(row, column, item);
    }
}
tableView = new QTableView;
tableView - >setModel(model);
setCentralWidget(tableView);

//获取视图的项目选择模型
QItemSelectionModel * selectionModel = tableView - >selectionModel();

//定义左上角和右下角的索引,然后使用这两个索引创建选择
 QModelIndex topLeft;
 QModelIndex bottomRight;
topLeft = model - >index(1, 1, QModelIndex());
bottomRight = model - >index(5, 2, QModelIndex());
QItemSelection selection(topLeft, bottomRight);

//使用指定的选择模式来选择项目
```

```
selectionModel ->select(selection, QItemSelectionModel::Select);
```

这里先获取了视图的选择模型;要使用选择模型来选择视图中的项目,那么就必须指定 QItemSelection 和选择模式 QItemSelectionModel::SelectionFlag。QItemSelection 是一个项目选择块,需要指定它的左上角和右下角的项目的索引;而选择模式是选择模型更新时的方式,它是一个枚举变量,在 QItemSelectionModel 类中被定义,可以在帮助中查看它的值。这里使用的 QItemSelectionModel::Select 表明所有指定的索引都将被选择,它还有其他的一些值,比如 QItemSelectionModel::Toggle,它会将指定索引的当前状态切换为相反的状态,如果以前项目没有被选择,那么现在会被选择;而如果项目已经被选择了,那么现在会取消选择。SelectionFlag 的一些值还可以使用位或“|”运算符来联合使用,比如使用 QItemSelectionModel::Select | QItemSelectionModel::Rows 可以选中指定选择的项目所在的所有行的项目。运行程序,效果如图 16-5 所示。

下面向程序中添加代码来看一下 QItemSelectionModel::Toggle 的效果,并讲解一下当前项目的相关内容。(项目源码路径: src\16\16-7\ mySelection)先在 mainwindow.h 文件中添加两个槽的声明:

```
public slots:  
    void getCurrentItemData();  
    void toggleSelection();
```

然后到 mainwindow.cpp 文件中,在构造函数中向主窗口工具栏中添加两个动作图标:

```
ui ->mainToolBar ->addAction(tr("当前项目"), this, SLOT(getCurrentItemData()));  
ui ->mainToolBar ->addAction(tr("切换选择"), this, SLOT(toggleSelection()));
```

然后添加两个函数的定义:

```
//输出当前项目的内容  
void MainWindow::getCurrentItemData()  
{  
    qDebug() << tr("当前项目的内容: ")  
        << tableView ->selectionModel() ->currentIndex().data().toString();  
}  
  
//切换选择的项目  
void MainWindow::toggleSelection()  
{
```

图 16-5 项目选择运行效果

```

QModelIndex topLeft = tableView->model()->index(0, 0, QModelIndex());
QModelIndex bottomRight = tableView->model()->index(tableView->model()->
rowCount(QModelIndex()) - 1, tableView->model()->columnCount(QModelIndex()) - 1, QModelIndex());
QItemSelection curSelection(topLeft, bottomRight);
tableView->selectionModel()->select(curSelection, QItemSelectionModel::Toggle);
}

```

在切换选择的项目时,将 QItemSelection 指定为了视图中所有的项目,而选择模式使用了 QItemSelectionModel::Toggle。因为代码中使用了中文,所以还要在 main.cpp 文件中添加相关处理代码。下面运行程序,先单击“当前项目”图标,那么会输出“0”,表明当前项目默认为第一个项目,这个从第一个项目拥有蚂蚁线就可以看出它是

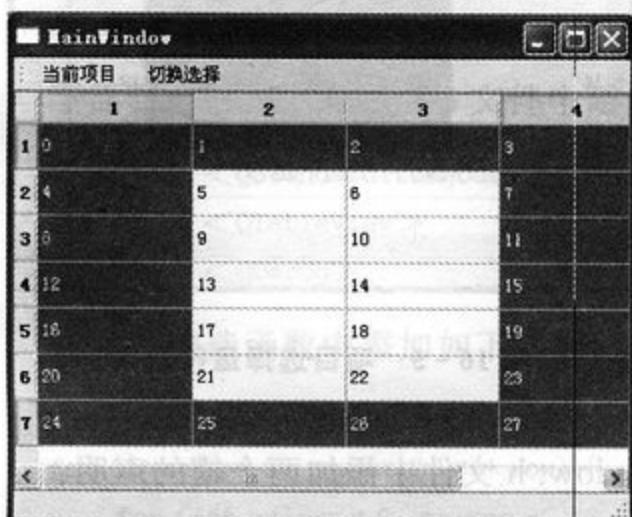


图 16-6 切换选择项目运行效果

当前项。然后单击“切换选择”图标,会发现视图中所有的项目的选择状态都变成了相反的状态,效果如图 16-6 所示。视图类中也提供了几个比较方便的函数来选择,比如,selectAll()选择全部项目,selectColumn()选择指定的一列项目,selectColumns()选择指定的多列项目,selectRow()选择指定的一行项目,selectRows()选择指定的多行项目等。

要获取选择模型中的模型索引,可以使用 selectedIndexes() 函数,它会返回一个模型索引的列表,然后遍历这个列表即可。当

选择模型中选择的项目改变时,会发射相关信息,下面在程序中再次添加代码,来看一下选择模型中信号的使用和对选择的项目的处理。(项目源码路径: src\16\16-8\mySelection)在 mainwindow.h 文件中添加类的前置声明:

```

class QItemSelection;
class QModelIndex;

```

然后再添加两个槽的声明:

```

void updateSelection(const QItemSelection &selected, const QItemSelection &deselected);
void changeCurrent(const QModelIndex &current, const QModelIndex &previous);

```

到 mainwindow.cpp 文件中,在构造函数中添加信号和槽的关联:

```

connect(selectionModel, SIGNAL(selectionChanged(QItemSelection, QItemSelection)), this,
SLOT(updateSelection(QItemSelection, QItemSelection)));
connect(selectionModel, SIGNAL(currentChanged(QModelIndex, QModelIndex)), this, SLOT(
(changeCurrent(QModelIndex, QModelIndex)));

```

这里分别关联了选择模型的选择改变信号 selectionChanged() 和当前项改变信号 currentChanged()。前者在选择的项目改变时发射，会包含新选择的项目 selected 和先前选择的项目 deselected；后者在当前项改变时发射，包含了新的当前项的索引和先前的当前项的索引。下面添加槽的实现：

```
//更新选择
void MainWindow::updateSelection(const QItemSelection &selected, const QItemSelection &deselected)
{
    QModelIndex index;
    QModelIndexList list = selected.indexes();

    //为现在选择的项目填充值
    foreach (index, list) {
        QString text = QString("( %1, %2)").arg(index.row()).arg(index.column());
        tableView->model()->setData(index, text);
    }

    list = deselected.indexes();

    //清空上一次选择的项目的内容
    foreach (index, list) {
        tableView->model()->setData(index, "");
    }
}
```

这里使用 indexes() 来获取了所有选择的项目的索引，然后重新为它们进行赋值。

```
//改变当前项目
void MainWindow::changeCurrent(const QModelIndex &current, const QModelIndex &previous)
{
    qDebug() << tr("move( %1, %2) to ( %3, %4)")
        .arg(previous.row()).arg(previous.column())
        .arg(current.row()).arg(current.column());
}
```

当前项改变时，输出它的位置信息。现在运行程序，可以使用鼠标来改变选择的项目和当前项，看下运行效果。

当多个视图显示同一个模型的数据时，只要使用 setSelectionModel() 函数为它们设置相同的选择模型，那么就可以共享选择。继续在前面程序中添加代码，在 mainwindow.h 文件中添加私有对象定义：

```
QTableView *tableView2;
```

然后到 mainwindow.cpp 文件，在构造函数中继续添加如下代码：

```
tableView2 = new QTableView;
tableView2 ->setWindowTitle("tableView2");
tableView2 ->resize(400, 300);
tableView2 ->setModel(model);
tableView2 ->setSelectionModel(selectionModel);
tableView2 ->show();
```

注意在析构函数中添加如下一行代码

```
delete tableView2;
```

这样在程序运行结束时可以释放 tableView2。运行程序，效果如图 16-7 所示。可以看到，当改变一个视图中选择的项目后，另一个视图中选择的项目会跟随着进行相同的变化。

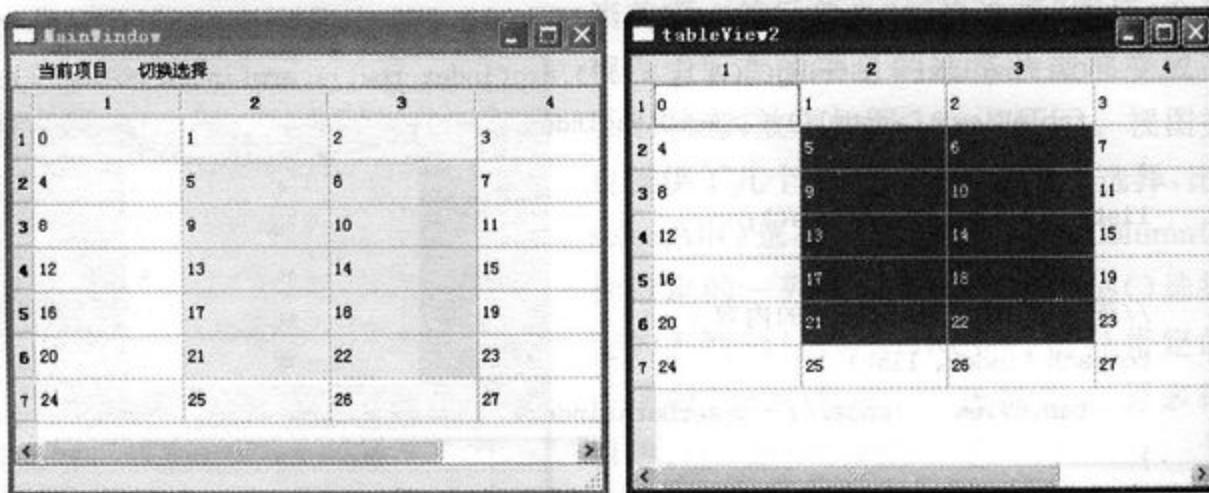


图 16-7 多个视图共享选择

16.4 委托类

16.4.1 基本概念

与 Model-View-Controller 模式不同，模型/视图结构中没有包含一个完全分离的组件来处理与用户的交互。一般地，视图用来将模型中的数据展示给用户，也用来处理用户的输入，这个在前面的程序中已经看到了。为了获得更高的灵活性，交互可以由委托来执行。这些组件提供了输入功能，而且也负责渲染一些视图中的个别项目。控制委托的标准接口在 `QAbstractItemDelegate` 类中定义。

委托通过实现 `paint()` 和 `sizeHint()` 函数来使它们可以渲染自身的内容。然而，简单的基于部件的委托可以通过子类化 `QItemDelegate` 来实现，而不需要使用 `QAbstractItemDelegate`，这样可以使用这些函数的默认实现。委托的编辑器可以通过两种方式来实现，一种是使用部件来管理编辑过程，另一种是直接处理事件。后面会通过一个例子来讲解第一种方式，也可以参考一下 Qt 提供的 Spin Box Delegate 的示例程序。

如果想要继承 QAbstractItemDelegate 来实现自定义的渲染操作,那么可以参考一下 Pixelator 示例程序。另外,还可以使用 QStyledItemDelegate 作为基类,这样可以自定义数据的显示,这个可以参考 Star Delegate 示例程序。这些示例程序都在 Item Views 分类中。

Qt 中的标准视图都使用 QItemDelegate 的实例来提供编辑功能,这种委托接口的默认实现为 QListview、QTableView 和 QTreeView 等标准视图的每一个项目提供了普通风格的渲染。标准视图中的默认委托会处理所有的标准角色,具体的内容可以在 QItemDelegate 类的帮助文档中查看。可以使用 itemDelegate() 函数获取一个视图中使用的委托,使用 setItemDelegate() 函数可以为一个视图安装一个自定义委托。

16.4.2 自定义委托

下面通过一个例子来讲解如何使用现成的部件来自定义委托。这里的委托使用了 QSpinBox 来提供编辑功能,主要用于显示整数的模型。(项目源码路径: src\16\16-9\mySelection)在前面源码路径为 16-8 的基础上继续添加代码。向项目中添加新的 C++ 类,类名为 SpinBoxDelegate,基类为 QItemDelegate,类型信息选择“继承者 QObject”。完成后先在 spinboxdelegate.h 文件中添加几个公有函数的声明:

```
QWidget * createEditor(QWidget * parent, const QStyleOptionViewItem &option, const QModelIndex &index) const;
void setEditorData(QWidget * editor, const QModelIndex &index) const;
void setModelData(QWidget * editor, QAbstractItemModel * model, const QModelIndex &index) const;
void updateEditorGeometry(QWidget * editor, const QStyleOptionViewItem &option, const QModelIndex &index) const;
```

这里的委托继承自 QItemDelegate,这样不需要编写自定义的显示函数,不过,还是必须要提供几个函数来管理编辑器部件。可以看到,在构造委托时并没用设置编辑器部件,只有在需要编辑器部件时才创建它。下面到 spinboxdelegate.cpp 文件中,先添加一个头文件包含: #include <QSpinBox>,再添加这几个函数的定义:

```
// 创建编辑器
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;
}
```

当视图需要一个编辑器时,它会告知委托来为被修改的项目提供一个编辑器部件。这里的 createEditor() 函数为委托设置一个合适的部件提供了所需要的一切。在这个

函数中，并不需要为编辑器部件保持一个指针，因为视图会负责在不再需要该编辑器时销毁它。

```
//为编辑器设置数据
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);
}
```

委托必须将模型中的数据复制到编辑器中，这里已经知道了编辑器部件是一个 QSpinBox，但是，也可能需要为模型中不同类型的数据提供不同的编辑器，基于这个原因，要在访问部件的成员函数以前将它转换为合适的类型。

```
//将数据写入到模型
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);
}
```

当用户完成了对 QSpinBox 部件中数据的编辑时，视图会通过调用 setModelData() 函数来告知委托将编辑好的数据存储到模型中。这里调用了 interpretText() 函数来确保获得的是 QSpinBox 中最近更新的数值。标准的 QItemDelegate 类会在完成编辑后发射 closeEditor() 信号来告知视图，视图确保编辑器部件被关闭和销毁。而这里只是提供了简单的编辑功能，并不需要发射这个信号。

```
//更新编辑器几何布局
void SpinBoxDelegate::updateEditorGeometry(QWidget * editor, const QStyleOptionViewItem
&option, const QModelIndex &/ * index */ ) const
{
    editor ->setGeometry(option.rect);
}
```

委托有责任来管理编辑器的几何布局，必须在创建编辑器以及视图中项目的大小或位置改变时设置它的几何布局，视图使用了一个 QStyleOptionViewItem 对象提供了所有需要的几何布局信息。这里只使用了项目的矩形作为编辑器的几何布局，而对于更复杂的编辑器部件，可能需要将这个矩形进行分割。

下面来使用自定义的委托。到 mainwindow.cpp 文件中，先添加头文件包含：#include "spinboxdelegate.h"，然后在构造函数的最后面添加如下代码：

```
SpinBoxDelegate * delegate = new SpinBoxDelegate(this);
tableView ->setItemDelegate(delegate);
```

一个视图可以通过调用 `setItemDelegate()` 函数来设置一个自定义的委托。下面运行程序，效果如图 16-8 所示，可以看到使用自定义委托和使用默认委托的不同。

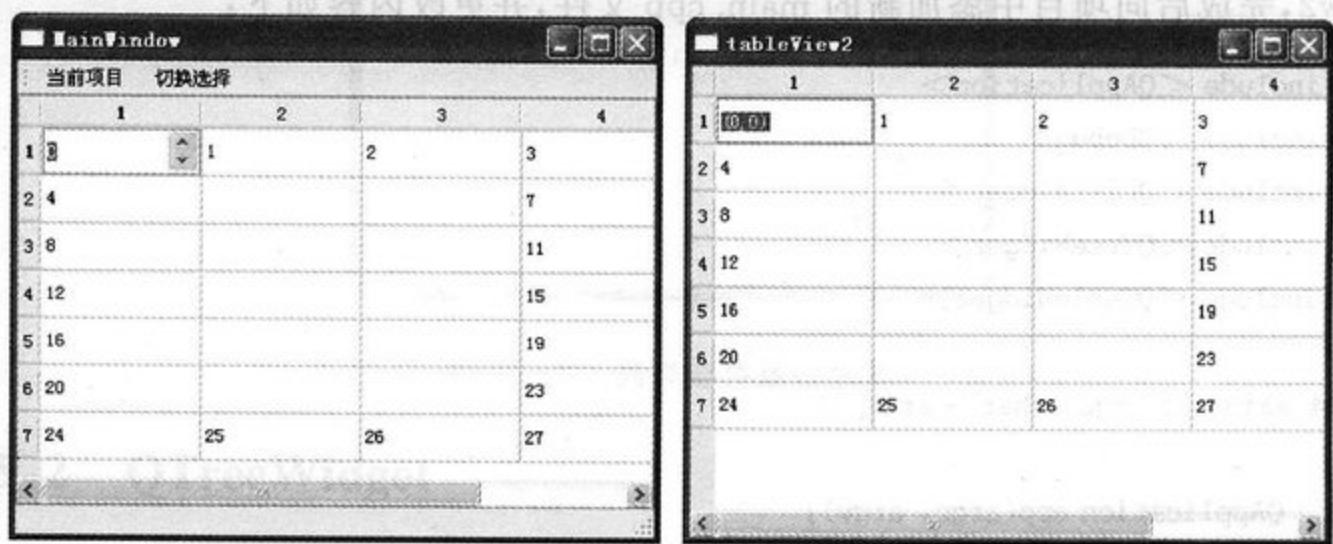


图 16-8 使用自定义委托运行效果

编辑完成后，委托应该为其他组件提供提示，告知它们编辑操作的结果，提供提示也有利于后续的编辑操作。这个可以通过在发射 `closeEditor()` 信号时使用合适的提示来实现，它们会被在构造编辑器时安装的默认的 `QItemDelegate` 事件过滤器捕获。可以通过调整编辑器的行为来使得它更加友好。对于 `QItemDelegate` 提供的默认的事件过滤器，如果用户在 `spinbox` 编辑器中按下回车键，那么委托就会向模型提交数值然后关闭编辑器。可以通过在 `spinbox` 上安装自己的事件过滤器来改变这个行为，并提供编辑提示来满足需要。例如，可以在发射 `closeEditor()` 时使用 `EditNextItem` 提示来实现在视图中自动编辑下一个项目。

另一种不需要使用事件过滤器的方式是提供自己的编辑器部件，例如子类化 `QSpinBox`。这种方式可以让我们对编辑器的行为提供更多的控制，不过它是以编写更多的代码为代价的。一般地，如果需要自定义一个标准的 Qt 编辑器部件的行为，在委托中安装一个事件过滤的方式更加简便。

16.5 项目视图的便捷类

Qt4 中也引进了一些标准部件来提供经典的基于项的容器部件，它们底层是通过模型/视图框架实现的。这些部件分别是：`QListWidget` 提供了一个项目列表，`QTreeWidget` 显示了一个多层次的树结构，`QTableWidget` 提供了一个以项目作为单元的表格。它们每一个类都继承了 `QAbstractItemView` 类的行为。这些类之所以被称为便捷类，是因为它们使用起来比较简单，适合于少量的数据的存储和显示。因为它们没有将视图和模型分离，所以没有视图类灵活，不能和任意的模型一起使用，一般建议使用模型/视图的方式来处理数据。也不建议子类化这些部件，它们的存在仅是为了和 Qt3

中的等价类提供一个相似的接口。

16.5.1 QListWidget

(项目源码路径: src\16\16-10\modelView2)新建空的 Qt 项目,项目名称为 modelView2,完成后向项目中添加新的 main.cpp 文件,并更改内容如下:

```
#include <QApplication>
#include <QDebug>
#include <QListWidget>
#include <QTreeWidget>
#include <QTableWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QListWidget listWidget;
    //一种添加项目的简便方法
    new QListWidgetItem("a", &listWidget);

    //添加项目的另一种方法,这样还可以进行各种设置
    QListWidgetItem *listWidgetItem = new QListWidgetItem;
    listWidgetItem->setText("b");
    listWidgetItem->setIcon(QIcon("../modelView2/yafeilinux.png"));
    listWidgetItem->setToolTip("this is b!");
    listWidget.insertItem(1, listWidgetItem);

    //设置排序为倒序
    listWidget.sortItems(Qt::DescendingOrder);

    //显示列表部件
    listWidget.show();
    return app.exec();
}
```

单层的项目列表一般使用 QListWidget 和 QListWidgetItem 来显示,一个列表部件可以像一般的窗口部件那样进行创建。可以在创建 QListWidgetItem 时将它直接添加到已经创建的列表部件中,也可以使用 QListWidget 类的 insertItem() 函数来添加。列表中的每一个项目都可以显示一个文本标签和一个图标,还可以为其设置工具提示、状态提示和“What’s This?”提示。默认的,列表中的项目会根据添加的顺序来排序,也可以使用 sortItems() 函数对项目排序,比如程序中使用的 Qt::DescendingOrder 是按字母降序排序、Qt::AscendingOrder 是按字母升序进行排序。运行程序,效果如图 16-9 所示。

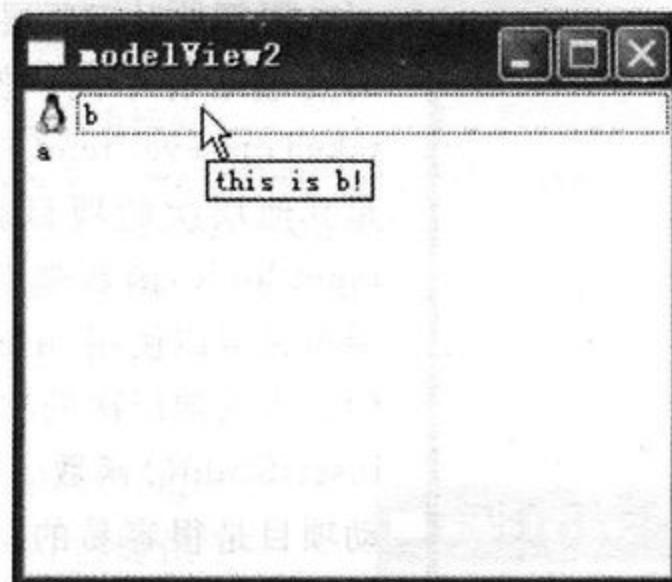


图 16-9 列表部件运行效果

16.5.2 QTreeWidget

在上面的程序中继续添加代码：

```
QTreeWidget treeWidget;

//必须设置列数
treeWidget.setColumnCount(2);

//设置标头
QStringList headers;
headers << "name" << "year";
treeWidget.setHeaderLabels(headers);

//添加项目
QTreeWidgetItem * grade1 = new QTreeWidgetItem(&treeWidget);
grade1->setText(0, "Grade1");
QTreeWidgetItem * student = new QTreeWidgetItem(grade1);
student->setText(0, "Tom");
student->setText(1, "1986");
QTreeWidgetItem * grade2 = new QTreeWidgetItem(&treeWidget, grade1);
grade2->setText(0, "Grade2");
treeWidget.show();
```

树或者项目的层次列表由 QTreeWidget 和 QTreeWidgetItem 类提供，在树部件中的每一个项目都可以有它自己的子项目，而且可以显示多列的信息。向树部件中添加项目以前，必须先使用 setColumnCount() 函数设置列的个数，比如程序中设置了两列，然后还为这两列提供了标头。树部件中的顶层项目使用树部件作为父部件来创建，它们可以使用任意的顺序被插入，也可以构建项目时指定它的前一个项目，比如程序中创建 grade2 时就指定了 grade1 为它的前一个项目。运行程序，效果如图 16-10 所示。

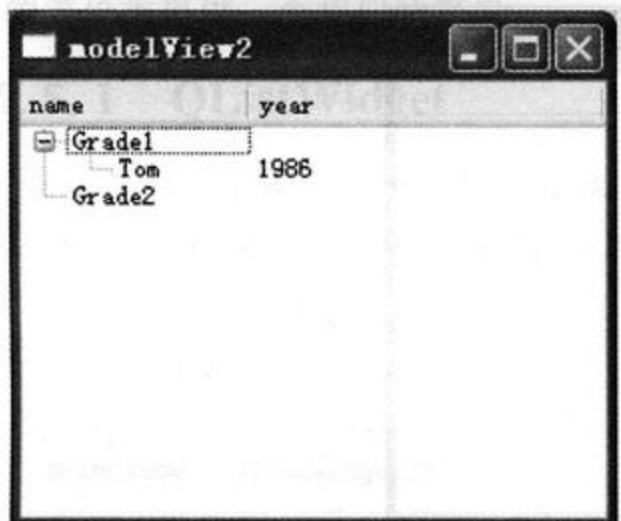


图 16-10 树部件运行效果

树部件对于顶层项目和更深层次的项目的处理略有不同。例如可以使用树部件的 `takeTopLevelItem()` 函数来删除顶层项目,但是其他层次的项目就要调用它们父项目的 `takeChild()` 函数来删除;在树部件中插入顶层项目可以使用 `insertTopLevelItem()` 函数,但插入其他层次的项目就要使用其父项目的 `insertChild()` 函数。在顶层和其他层之间移动项目是很容易的,只需要检查该项目是否为顶层项目,这个可以使用 `parent()` 函数获得。例如,可以使用下面的代码来删除当前的项目:

```
//先获取当前项目的父项目
QTreeWidgetItem * parent = currentItem ->parent();
int index;
//如果当前项目有父项目,则使用其父项目删除当前项目,否则使用树部件删除当前项目
if (parent) {
    index = parent ->indexOfChild(treeWidget ->currentItem());
    delete parent ->takeChild(index);
} else {
    index = treeWidget ->indexOfTopLevelItem(treeWidget ->currentItem());
    delete treeWidget ->takeTopLevelItem(index);
}
```

可以使用相同的方法在当前项目之后添加新的项目,例如:

```
QTreeWidgetItem * parent = currentItem ->parent();
QTreeWidgetItem * newItem;
if (parent)
    newItem = new QTreeWidgetItem(parent, treeWidget ->currentItem());
else
    newItem = new QTreeWidgetItem(treeWidget, treeWidget ->currentItem());
```

16.5.3 QTableWidget

继续向前面的程序中添加代码:

```
//创建表格部件,同时指定行数和列数
QTableWidget tableWidget(3, 2);

//创建表格项目,并插入到指定单元
QTableWidgetItem * tableWidgetItem = new QTableWidgetItem("qt");
```

```

tableWidget.setItem(1, 1, tableWidgetItem);

//创建表格项目，并将它们作为标头
QTableWidgetItem * headerV = new QTableWidgetItem("first");
tableWidget.setVerticalHeaderItem(0, headerV);
QTableWidgetItem * headerH = new QTableWidgetItem("ID");
tableWidget.setHorizontalHeaderItem(0, headerH);
tableWidget.show();

```

项目表格使用 QTableWidget 和 QTableWidgetItem 来构建，它提供了一个包含表头和项目的可滚动表格部件。表格一般在构造时就指定它的行数和列数，项目可以在表格外先构建，然后再添加到表格中指定的位置，而且表格项目还可以作为水平或者垂直标头。运行程序，效果如图 16-11 所示。

16.5.4 共同特性

这 3 个便捷类都使用了相同的接口提供了一些基于项的特色功能。例如，有时候在项目视图部件中需要隐藏一些项目（而不是删除它们），这可以使用 QListWidgetItem 类和 QTreeWidgetItem 类提供的 setHidden() 函数，判断一个项目是否隐藏，可以使用相应的 isHidden() 函数；可以使用 3 个便捷类的 selectedItems() 函数来获取选择的项目，它会返回一个相关项目的列表；还可以使用 findItems() 函数来查找项目，它也会返回一个相关项目的列表。

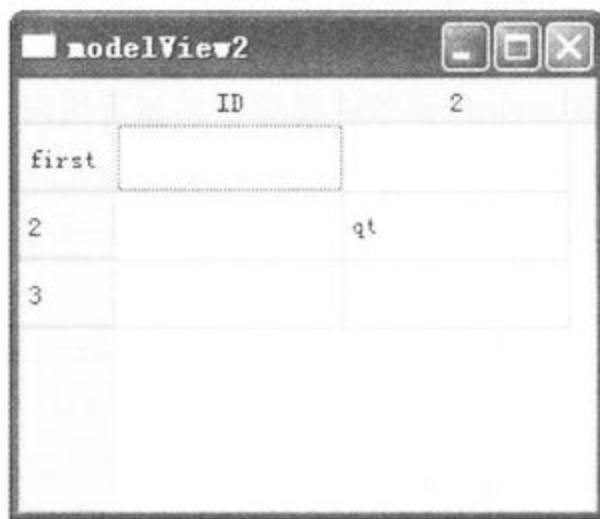


图 16-11 表格部件运行效果

16.6 在项目视图中启用拖放

Qt 的拖放应用可以被模型/视图框架完全支持，列表、表格和树中的项目可以在视图中被拖拽，数据可以作为 MIME 编码的数据被导入和导出。标准视图可以自动支持内部的拖放，这样可以用来改变项目的排列顺序。默认的，视图的拖放功能并没有被启用，如果要进行项目的拖动，需要进行一些属性的设置。如果要在一个新的模型中启用拖放功能，那么还要重新实现一些函数。

16.6.1 在便捷类中启用拖放

在 QListWidget、QTableWidget 和 QTreeWidget 中的每一种类型的项目都默认配置了一组不同的标志。例如，每一个 QListWidgetItem 和 QTreeWidgetItem 被初始化为可用的、可检查的、可选择的，也可以用作拖放操作的源；而每一个 QTableWidget-

Item 可以被编辑和用作拖放操作的目标。尽管所有的标准项目都有一个或者两个标志来设置拖放,但是,一般还是需要在视图中设置一些属性来使它启用对拖放操作的内建支持:

- 启用项目拖拽,要将视图的 dragEnable 属性设置为 true;
- 要允许用户将内部或者外部的项目放入视图中,需要设置视图的 viewport() 的 acceptDrops 属性为 true;
- 要显示现在用户拖拽的项目将要被放置的位置,需要设置 showDropIndicator 属性。

(项目源码路径: src\16\16-11\modelView2)仍然在前面的例程的基础上进行更改。在 main() 函数中继续添加如下代码:

```
//设置选择模式为单选
listWidget.setSelectionMode(QAbstractItemView::SingleSelection);

//启用拖动
listWidget.setDragEnabled(true);

//设置接受拖放
listWidget.viewport() ->setAcceptDrops(true);

//设置显示将要被放置的位置
listWidget.setDropIndicatorShown(true);

//设置拖放模式为移动项目,如果不设置,默认为复制项目
listWidget.setDragDropMode(QAbstractItemView::InternalMove);
```

现在运行程序,效果如图 16-12 所示。可以看到,拖拽项目到一个合适的位置时会显示出一条短线,表明项目可以放置在该位置,这就是 showDropIndicator 属性的作用。

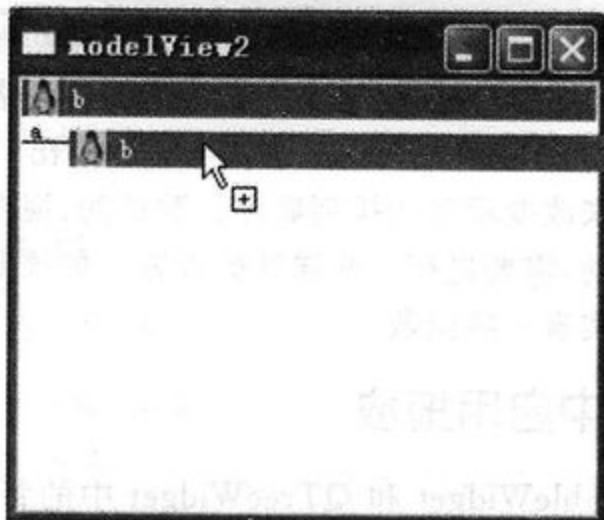


图 16-12 拖放操作运行效果

16.6.2 在模型/视图类中启用拖放

(项目源码路径：src\16\16-12\myModel)在源码路径为 16-5 创建的自定义模型的程序中进行更改。在视图中启用拖放功能与前面在便捷类中的设置是相似的，main()函数中添加如下代码：

```
listView.setSelectionMode(QAbstractItemView::ExtendedSelection);
listView.setDragEnabled(true);
listView.setAcceptDrops(true);
listView.setDropIndicatorShown(true);
```

因为视图中显示的数据是由模型控制的，所以也要为使用的模型提供拖放操作支持。这需要添加必要的函数。先在 stringlistmodel.h 文件中添加如下函数的声明。

```
Qt::DropActions supportedDropActions() const;
QStringList mimeTypes() const;
QMimeData * mimeData(const QModelIndexList &indexes) const;
bool dropMimeData(const QMimeData * data, Qt::DropAction action, int row, int column,
const QModelIndex &parent);
```

下面到 stringlistmodel.cpp 文件中，先添加头文件 #include <QMimeData>，后分别实现前面添加的几个函数。

```
//设置支持放入动作
Qt::DropActions StringListModel::supportedDropActions() const
{
    return Qt::CopyAction | Qt::MoveAction;
}
```

这里设置了支持使用拖放进行复制和移动两种操作。尽管这里可以使用任何 Qt::DropActions 的值，不过，这也需要模型实现一些函数来进行支持。比如，要允许使用 Qt::MoveAction，那么模型就必须实现 removeRows() 函数。

```
//设置在拖放操作中导出的条目的数据的编码类型
QStringList StringListModel::mimeTypes() const
{
    QStringList types;
    // "application/vnd.text.list" 是自定义的类型，在后面的函数中要保持一致
    types << "application/vnd.text.list";
    return types;
}
```

拖放操作中的数据项被从模型中导出时，它们要被编码为合适的格式来对应一个或者多个 MIME 类型。这里自定义了一个类型，它仅支持纯文本类型。

```
//将拖放的数据放入 QMimeData 中
```

```
QMimeData * QStringListModel::mimeData(const QModelIndexList &indexes) const
{
    QMimeData * mimeData = new QMimeData();
    QByteArray encodedData;
    QDataStream stream(&encodedData, QIODevice::WriteOnly);
    foreach (const QModelIndex &index, indexes) {
        if (index.isValid()) {
            QString text = data(index, Qt::DisplayRole).toString();
            stream << text;
        }
    }
    //将数据放入 QMimeData 中
    mimeData->setData("application/vnd.text.list", encodedData);
    return mimeData;
}
```

拖放操作之前需要将数据放入一个 QMimeData 类型的对象中, 这里就是使用自定义的格式将所有要拖拽的数据都放入了一个 QMimeData 对象中。

```
//将拖放的数据放入模型中
bool QStringListModel::dropMimeData(const QMimeData * data, Qt::DropAction action, int
row, int column, const QModelIndex &parent)
{
    //如果放入动作是 Qt::IgnoreAction, 那么返回 true
    if (action == Qt::IgnoreAction)
        return true;
    //如果数据的格式不是指定的格式, 那么返回 false
    if (! data->hasFormat("application/vnd.text.list"))
        return false;
    //因为这里是列表, 只用一列, 所以列大于 0 时返回 false
    if (column > 0)
        return false;
    //设置开始插入的行
    int beginRow;
    if (row != -1)
        beginRow = row;
    else if (parent.isValid())
        beginRow = parent.row();
    else
        beginRow = rowCount(QModelIndex());
    //将数据从 QMimeData 中读取出来, 然后插入到模型中
    QByteArray encodedData = data->data("application/vnd.text.list");
    QDataStream stream(&encodedData, QIODevice::ReadOnly);
```

```

QStringList newItems;
int rows = 0;
while (! stream.atEnd()) {
    QString text;
    stream >> text;
    newItems << text;
    ++rows;
}
insertRows(beginRow, rows, QModelIndex());
foreach (const QString &text, newItems) {
    QModelIndex idx = index(beginRow, 0, QModelIndex());
    setData(idx, text);
    beginRow++;
}
return true;
}

```

任何给定的模型处理放入数据的方式都依赖于它们的类型(列表、表格或者树)和它们的内容向用户展现的方式。一般地,应该使用最适合模型底层数据存储的方式来容纳放入的数据。不同类型的模型会使用不同的方式来处理放入的数据。列表和表格模型只提供了一个平面结构来存储数据项,其结果是,它们可能会在当数据放入一个视图中的已经存在的项目时插入新的行(和列),或者使用提供的数据来覆盖已经存在项目的内容。树模型一般会在它们的底层数据存储中添加包含新的数据的子项目。

在这里,我们先判断放入的数据是否为前面自定义的类型,如果不是则不接受;因为这里实现的是列表模型,所以只有一列,如果列数大于0,那么也不接受。对于将要插入模型的数据,会根据它是否放在了一个已经存在的项目上而进行不同的处理。这里,只允许在已经存在的项目之间、或者第一个项目之前,或者最后一个项目之后进行放入。在一个层次的模型中,当一个放入发生在一个项目上时,最好是在模型中插入一个新的项目作为该项目的孩子,因为这里的模型只有一层,所以没有使用这种方式。然后是对导入的数据进行编码,再将它插入到模型中。

最后还要将以前写的flags()函数的内容更改如下:

```

Qt::ItemFlags QStringListModel::flags(const QModelIndex &index) const
{
    //如果该索引无效,那么只支持放入操作
    if (! index.isValid())
        return Qt::ItemIsEnabled | Qt::ItemIsDropEnabled;

    //如果该索引有效,那么既支持拖拽操作,也支持放入操作
    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable
        | Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled;
}

```

模型通过 flags() 函数提供合适的标志来向视图表明哪些项目可以被拖拽,哪些项目可以接受放入。关于这部分内容,也可以参考 Item Views 分类中的 Puzzle 示例程序,它是一个拼图游戏。

16.7 其他内容

16.7.1 代理模型

代理模型可以将一个模型中的数据进行排序或者过滤,然后提供给视图进行显示。Qt 中提供了 QSortFilterProxyModel 作为标准的代理模型来完成模型中数据的排序和过滤,要使用一个代理模型,则只需要为其设置源模型,然后在视图中使用该代理模型即可。下面通过一个例子来讲解。

(项目源码路径: src\16\16-13\myProxyModel) 新建 Qt Gui 应用,项目名称为 myProxyModel,类名为 MainWindow,基类为 QMainWindow。完成后进入 mainwindow.ui 文件,向界面中拖入一个 List View、Line Edit 和 Push Button 部件,并将 Push Button 部件的显示文本改为“过滤”。下面先进入 mainwindow.h 文件中,添加类的前置声明:

```
class QSortFilterProxyModel;
```

然后再添加一个私有对象定义:

```
QSortFilterProxyModel * filterModel;
```

下面到 mainwindow.cpp 文件中,先添加头文件包含:

```
#include <QStringListModel>
#include <QSortFilterProxyModel>
```

再在构造函数中添加如下代码:

```
QStringList list;
list << "yafei" << "yafeilinux" << "Qt" << "Qt Creator";
QStringListModel * listModel = new QStringListModel(list, this);
filterModel = new QSortFilterProxyModel(this);
```

```
//为代理模型添加源模型
filterModel ->setSourceModel(listModel);
```

```
//在视图中使用代理模型
ui ->listView ->setModel(filterModel);
```

下面进入“过滤”按钮的单击信号槽,更改如下:

```
void MainWindow::on_pushButton_clicked()
```

```

    QRegExp rx(ui->lineEdit->text());
    filterModel->setFilterRegExp(rx);
}

```

这里将行编辑器中的文本作为正则表达式的内容,然后使用该正则表达式作为代理模型的过滤器。这样每当条件改变时,都会自动更新视图的显示。现在运行程序,效果如图 16-13 所示。对于 QSortFilterProxyModel 的使用,可以参考 Basic Sort/Filter Model 和 Address Book 示例程序;如果想自定义代理模型,可以参考 Custom Sort/Filter Model 示例程序。它们都在 Item Views 分类中。

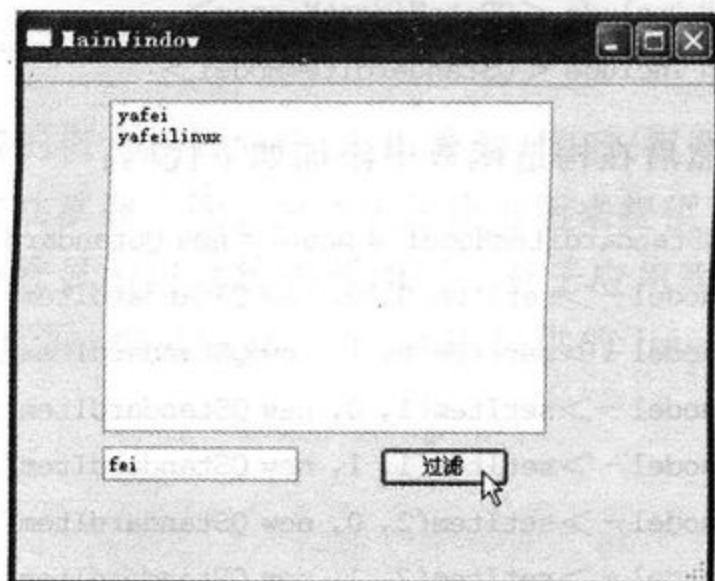


图 16-13 代理模型运行效果

16.7.2 数据—窗口映射器

数据—窗口映射器 QDataWidgetMapper 类在数据模型的一个区域和一个窗口部件间提供了一个映射,这样就可以实现在一个窗口部件上显示和编辑一个模型中的一行数据。下面来看一个例子。

(项目源码路径: src\16\16-14\myMapper) 新建 Qt Gui 应用,项目名称为 myMapper,类名为 MainWindow,基类为 QMainWindow。完成后进入 mainwindow.ui 文件,往界面上拖入 Label、Line Edit 和 Push Button 等部件,最终效果如图 16-14 所示。然后进入 mainwindow.h 文件,先添加类的前置声明:

```
class QDataWidgetMapper;
```

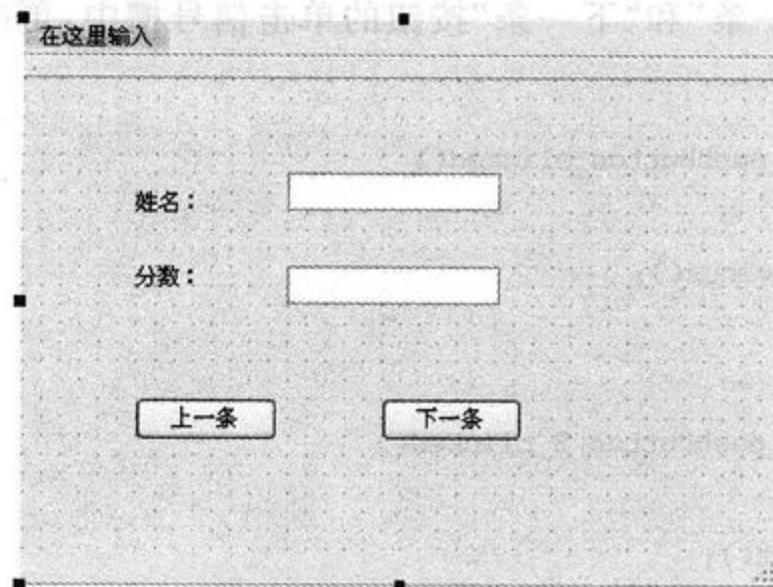


图 16-14 数据—窗口映射器

然后再添加一个私有对象定义：

```
QDataWidgetMapper * mapper;
```

下面再到 mainwindow.cpp 文件中，添加头文本包含：

```
# include <QDataWidgetMapper>
# include <QStandardItemModel>
```

然后在构造函数中添加如下代码：

```
QStandardItemModel * model = new QStandardItemModel(3, 2, this);
model ->setItem(0, 0, new QStandardItem("xiaoming"));
model ->setItem(0, 1, new QStandardItem("90"));
model ->setItem(1, 0, new QStandardItem("xiaogang"));
model ->setItem(1, 1, new QStandardItem("75"));
model ->setItem(2, 0, new QStandardItem("xiaohong"));
model ->setItem(2, 1, new QStandardItem("80"));
mapper = new QDataWidgetMapper(this);

//设置模型
mapper ->setModel(model);

//设置窗口部件和模型中的列的映射
mapper ->addMapping(ui ->lineEdit, 0);
mapper ->addMapping(ui ->lineEdit_2, 1);

//显示模型中的第一行
mapper ->toFirst();
```

这里创建了一个 *QDataWidgetMapper* 实例，然后为其设置了关联的模型，并设置了窗口部件和模型中对应列的映射，最后使用 *toFirst()* 函数来显示模型中第一行的数据。下面分别进入“上一条”和“下一条”按钮的单击信号槽中，更改如下：

```
//上一条按钮
void MainWindow::on_pushButton_clicked()
{
    mapper ->toPrevious();
}

//下一条按钮
void MainWindow::on_pushButton_2_clicked()
{
    mapper ->toNext();
}
```

这里分别使用了 *toPrevious()* 函数和 *toNext()* 函数来显示模型中上一行和下一行

的数据。还有一个 toLast() 函数可以显示模型中最后一行的数据。关于 QDataWidgetMapper 类的使用,也可以参考 Simple Widget Mapper 和 Combo Widget Mapper 示例程序,它们在 Item Views 分类中。

16.8 小结

本章详细讲解了模型/视图架构中众多的概念及其应用,可以看到,模型/视图框架是一个很复杂的知识框架,初学时很难一次性掌握。读者只要记住模型用来提供数据,视图用来显示数据,委托用来提供项目的特殊显示以及编辑器即可。对于应用程序编程,本章的内容还是非常重要的,学习完本章后,也可以看一下 Qt 中提供的 Interview 演示程序。

第 17 章

数据库和 XML

本章将讲解数据库和 XML 的相关内容。在学习数据库相关内容前,建议读者掌握一些基本的 SQL 知识,应该可以看懂基本的 SELECT、INSERT、UPDATE 和 DELETE 等语句,但这并不是必须的,因为 Qt 中提供了不需要 SQL 知识就可以浏览和编辑数据库的接口。在学习 XML 部分前,也建议读者先对 XML 有一个大概的了解。

17.1 数据库

Qt 中的 QSql 模块提供了对数据库的支持,该模块中的众多类基本上可以分为 3 层,如表 17-1 所列。

表 17-1 QSql 模块的类分层

用户接口层	QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel
SQL 接口层	QSqlDatabase、QSqlQuery、QSqlError、QSqlField、QSqlIndex 和 QSqlRecord
驱动层	QSqlDriver、QSqlDriverCreator<T>、QSqlDriverCreatorBase、QSqlDriverPlugin 和 QSqlResult

其中,驱动层为具体的数据库和 SQL 接口层之间提供了底层的桥梁;SQL 接口层提供了对数据库的访问,其中的 QSqlDatabase 类用来创建连接,QSqlQuery 类可以使用 SQL 语句来实现与数据库交互,其他几个类对该层提供了支持;用户接口层的几个类实现了将数据库中的数据链接到窗口部件上,这些类是使用前一章的模型/视图框架实现的,它们是更高层次的抽象,即便不熟悉 SQL 也可以操作数据库。如果要使用 QSql 模块中的这些类,需要在项目文件(.pro 文件)中添加“QT += sql”这一行代码。对应数据库部分的内容,可以在帮助中查看 SQL Programming 关键字。

17.1.1 连接到数据库

1. SQL 数据库驱动

QtSql 模块使用数据库驱动来和不同的数据库接口进行通信。Qt 的 SQL 模型接口是独立于数据库的，所以所有数据库特定的代码都包含在了这些驱动中。Qt 默认支持一些驱动，也可以添加其他驱动，Qt 中包含的驱动如表 17-2 所列。

表 17-2 Qt 中包含的数据库驱动

驱动名称	数据库
QDB2	IBM DB2(7.1 版或者以上版本)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity(ODBC)-微软 SQL Server 和其他 ODBC 兼容数据库
QPSQL	PostgreSQL(7.3 版本或者更高)
QSQLITE2	SQLite 版本 2
QSQLITE	SQLite 版本 3
QTDS	Sybase Adaptive Server 注：从 Qt 4.7 开始已经过时

需要说明的是，由于 GPL 许可证的兼容性问题，并不是这里列出的所有插件都提供了 Qt 的开源版本。下面通过程序来查看 Qt 中可用的数据库插件。（项目源码路径：src\17\17-1\databaseDriver）新建空的 Qt 项目，项目名称为 databaseDriver，完成后往项目中添加新的 main.cpp 文件。下面先在 databaseDriver.pro 文件中添加如下一行代码：

```
QT += sql
```

完成后按下 Ctrl+S 快捷键保存该文件，然后将 main.cpp 文件的内容更改如下：

```
#include < QApplication >
#include < QSqlDatabase >
#include < QDebug >
#include < QStringList >

int main( int argc, char * argv[] )
{
    QApplication a( argc, argv );
    qDebug() << "Available drivers:" ;
    QStringList drivers = QSqlDatabase::drivers();
    foreach( QString driver, drivers )
        qDebug() << driver;
```

```

    return a.exec();
}

```

这里使用了 QSqlDatabase 类的静态函数 drivers() 获取了可用的驱动的列表，然后将它们遍历输出。运行程序可以看到输出的结果为：“QSQLITE”、“QODBC3”和“QODBC”。表明现在仅支持这 3 个驱动。其实，也可以在 Qt 安装目录下的 plugins/sqldrivers 文件夹中看到所有的驱动插件文件。这里要重点提一下 SQLite 数据库，它是一款轻型的文件型数据库，主要应用于嵌入式领域，支持跨平台，而且 Qt 对它提供了很好的默认支持，所以在本章后面的内容将使用该数据库为例子来讲解。关于数据库驱动的更多内容，可以参考 SQL Database Drivers 关键字对应的帮助文档，这里还列出了编译驱动器插件和编写自定义的数据库驱动的方法。

2. 创建数据库连接

要想使用 QSqlQuery 或者 QSqlQueryModel 来访问数据库，那么先要创建并打开一个或者多个数据库连接。数据库连接使用连接名来定义，而不是使用数据库名，可以向相同的数据库创建多个连接。QSqlDatabase 也支持默认连接的概念，默认连接就是一个没有命名的连接。在使用 QSqlQuery 或者 QSqlQueryModel 的成员函数时需要指定一个连接名作为参数，如果没有指定，那么就会使用默认连接。如果在应用程序中只需要有一个数据库连接，那么使用默认连接是很方便的。

创建一个连接就是创建了一个 QSqlDatabase 类的实例，而直到该连接被打开以前，它都是没有被使用的。下面的代码片段显示了怎样创建一个默认的连接，然后打开它：

```

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setHostName("bigblue");
db.setDatabaseName("flightdb");
db.setUserName("acarlson");
db.setPassword("1uTbSbAs");
bool ok = db.open();

```

第一行创建了一个连接对象，最后一行打开该连接以便使用。当创建了连接后，还初始化了一些连接信息，包括数据库名、主机名、用户名和密码等。这里连接到主机 bigblue 上的名称为 flightdb 的 SQLite 数据库。在 addDatabase() 函数中的“QSQLITE”参数指定了该连接使用的数据库驱动。因为这里并没有指定 addDatabase() 函数的第二个参数即连接名，所以这样建立的是默认连接。

下面通过一个例子来具体看一下数据库连接的建立过程。（项目源码路径：src\17\17-2\databaseDriver）在前面的项目中添加新的 C++ 头文件，名称为 connection.h，完成后将其内容更改为：

```

#ifndef CONNECTION_H
#define CONNECTION_H

```

```

#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(":memory:");
    if (!db.open()) {
        QMessageBox::critical(0, "Cannot open database",
            "Unable to establish a database connection.", QMessageBox::Cancel);
        return false;
    }
    QSqlQuery query;
    query.exec("create table student (id int primary key,"
               "name varchar(20))");
    query.exec("insert into student values(0, 'LiMing')");
    query.exec("insert into student values(1, 'LiuTao')");
    query.exec("insert into student values(2, 'WangHong')");
    return true;
}
#endif // CONNECTION_H

```

这个头文件添加了一个建立连接的函数,使用这个头文件的目的就是要简化主函数中的内容。这里先创建了一个 SQLite 数据库的默认连接,设置数据库名称时使用了“:memory:”,表明这是建立在内存中的数据库,也就是说该数据库只在程序运行期间有效,等程序运行结束时就会将其销毁。当然,也可以将其改为一个具体的数据库名称,比如“my. db”,这样就会在项目目录中创建该数据库文件了。下面使用 open() 函数将数据库打开,如果打开失败,则弹出提示对话框。最后使用 QSqlQuery 创建了一个 student 表,并插入了包含 id 和 name 两个字段的 3 条记录,如表 17-3 所列。其中,id 字段是 int 类型的,“primary key”表明该字段是主键,它不能为空,而且不能有重复的值;而 name 字段是 varchar 类型的,并且不大于 20 个字符。这里使用的 SQL 语句都要包含在双引号中,如果一行写不完,那么分行后,每一行都要使用两个双引号引起来。关于 QSqlQuery 的用法,将会在下一节讲到。

下面到 main. cpp 文件中,先添加头文件包含:

```
#include "connection.h"
#include <QVariant>
```

表 17-3 创建的 student 表

id	name
0	LiMing
1	LiuTao
2	WangHong

然后再更改主函数的内容为：

```
int main(int argc, char * argv[])
{
    QApplication a(argc, argv);

    //创建数据库连接
    if (! createConnection()) return 1;

    //使用 QSqlQuery 查询整张表
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
        qDebug() << query.value(0).toInt() << query.value(1).toString();
    }
    return a.exec();
}
```

这里调用了 `createConnection()` 函数来创建数据库连接，然后使用 `QSqlQuery` 查询整张表并将其所有内容进行了输出。现在运行程序，可以在应用程序输出栏中看到 `student` 表格中的内容。这个例子中使用了默认连接，下面再更改程序，看一下同时建立多个连接的情况。

(项目源码路径：src\17\17-3\databaseDriver)首先将 `connection.h` 文件中的创建连接的 `createConnection()` 函数的内容更改如下：

```
static bool createConnection()
{
    //创建一个数据库连接,使用“connection1”为连接名
    QSqlDatabase db1 = QSqlDatabase::addDatabase("QSQLITE", "connection1");
    db1.setDatabaseName("my1.db");
    if (! db1.open())
    {
        QMessageBox::critical(0, "Cannot open database1",
            "Unable to establish a database connection.", QMessageBox::Cancel);
        return false;
    }

    //这里要指定连接
    QSqlQuery query1(db1);
    query1.exec("create table student (id int primary key, "
               "name varchar(20))");
    query1.exec("insert into student values(0, 'LiMing')");
    query1.exec("insert into student values(1, 'LiuTao')");
    query1.exec("insert into student values(2, 'WangHong')");
```

```

//创建另一个数据库连接,要使用不同的连接名,这里是“connection2”
QSqlDatabase db2 = QSqlDatabase::addDatabase("QSQLITE", "connection2");
db2.setDatabaseName("my2.db");
if (!db2.open()) {
    QMessageBox::critical(0, "Cannot open database1",
        "Unable to establish a database connection.", QMessageBox::Cancel);
    return false;
}

//这里要指定连接
QSqlQuery query2(db2);
query2.exec("create table student (id int primary key, "
    "name varchar(20))");
query2.exec("insert into student values(10, LiQiang)");
query2.exec("insert into student values(11, MaLiang)");
query2.exec("insert into student values(12, ZhangBin)");
return true;
}

```

这里分别使用了 connection1 和 connection2 为连接名创建了两个连接,这两个连接分别设置了数据库名为“my1.db”和“my2.db”,它们是两个数据库文件。当存在多个连接时,使用 QSqlQuery 就要指定使用的是哪个连接,这样才能在正确的数据库上进行操作。使用两个连接分别创立了两个 student 表,但是其中的记录的内容是不同的。下面到 main.cpp 文件中分别输出这两个表中的内容,将主函数的内容更改如下:

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    //创建数据库连接
    if (!createConnection()) return 1;

    //使用 QSqlQuery 查询连接 1 的整张表,先要使用连接名获取该连接
    QSqlDatabase db1 = QSqlDatabase::database("connection1");
    QSqlQuery query1(db1);
    qDebug() << "connection1:";

    query1.exec("select * from student");
    while(query1.next())
    {
        qDebug() << query1.value(0).toInt() << query1.value(1).toString();
    }
}

```

```

// 使用 QSqlQuery 查询连接 2 的整张表
QSqlDatabase db2 = QSqlDatabase::database("connection2");
QSqlQuery query2(db2);
qDebug() << "connection2:";
query2.exec("select * from student");
while(query2.next())
{
    qDebug() << query2.value(0).toInt() << query2.value(1).toString();
}
return a.exec();
}

```

这里主要是使用了 QSqlDatabase 的 database() 静态函数, 通过指定连接名来获取相应的数据库连接, 然后在 QSqlQuery 中使用该连接进行数据库的查询操作。现在运行程序, 就可以输出两个表中的内容了, 而在项目目录中也可以看到生成的“my1.db”和“my2.db”两个数据库文件。

17.1.2 执行 SQL 语句

1. 执行一个查询

QSqlQuery 类提供了一个接口, 用于执行 SQL 语句和浏览查询的结果集。要执行一个 SQL 语句, 只需要简单地创建一个 QSqlQuery 对象, 然后调用 QSqlQuery::exec() 函数即可, 例如:

```

QSqlQuery query;
query.exec("select * from student");

```

在 QSqlQuery 的构造函数中可以接受一个可选的 QSqlDatabase 对象来指定使用的是哪一个数据库连接, 当没有指定连接时, 就是使用默认连接。如果发生了错误, 那么 exec() 函数会返回 false, 可以使用 QSqlQuery::lastError() 来获取错误信息。

2. 浏览结果集

QSqlQuery 提供了对结果集的访问, 可以一次访问一条记录。当执行完 exec() 函数后, QSqlQuery 的内部指针会位于第一条记录前面的位置。必须调用一次 QSqlQuery::next() 函数来使其前进到第一条记录, 然后可以重复使用 next() 函数来访问其他的记录, 直到该函数的返回值为 false, 例如可以使用以下代码来遍历一个结果集:

```

while(query.next())
{
    qDebug() << query.value(0).toInt() << query.value(1).toString();
}

```

其中, QSqlQuery::value() 函数可以返回当前记录的一个字段值。比如 value(0) 就是第一个字段的值, 各个字段从 0 开始编号。该函数返回一个 QVariant, 不同的数据类型会自动映射为 Qt 中最接近的相应类型, 这里的.toInt() 和.toString() 就是将 QVariant 转换为 int 和 QString 类型。在 Data Types for Qt-supported Database Systems 关键字对应的帮助文档中列出了所有的数据库数据类型在 Qt 中的对应类型, 需要时可以参考一下。

在 QSqlQuery 类中提供了多个函数来实现在结果集中进行定位, 比如 next() 定位到下一条记录、previous() 定位到前一条记录、first() 定位的第一条记录、last() 定位到最后一条记录、seek(n) 定位到第 n 条记录。如果只需要使用 next() 和 seek() 来遍历结果集, 那么可以在调用 exec() 函数以前调用 setForwardOnly(true), 这样可以显著加快在结果集上的查询速度。当前行的索引可以使用 at() 返回; record() 函数可以返回当前指向的记录; 如果数据库支持, 那么可以使用 size() 来返回结果集中的总行数。要判断是否一个数据库驱动支持一个给定的特性, 可以使用 QSqlDriver::hasFeature() 函数。下面通过例子来看一下这些函数的使用。

(项目源码路径: src\17\17-4\databaseDriver) 在源码路径为 17-3 的例程的基础上进行更改, 先在 main.cpp 文件中添加如下头文件包含:

```
#include <QSqlDriver>
#include <QSqlRecord>
#include <QSqlField>
```

然后在主函数中继续添加如下代码:

```
int numRows;

//先判断该数据库驱动是否支持 QuerySize 特性, 如果支持, 则可以使用 size() 函数
//如果不支持, 那么就使用其他方法来获取总行数
if (db2.driver() ->hasFeature(QSqlDriver::QuerySize)) {
    qDebug() << "has feature: query size";
    numRows = query2.size();
} else {
    qDebug() << "no feature: query size";
    query2.last();
    numRows = query2.at() + 1;
}
qDebug() << "row number: " << numRows;

//指向索引为 1 的记录, 即第二条记录
query2.seek(1);

//返回当前索引值
qDebug() << "current index: " << query2.at();
```

```
//获取当前行的记录  
QSqlRecord record = query2.record();  
  
//获取记录中“id”和“name”两个字段的值  
int id = record.value("id").toInt();  
QString name = record.value("name").toString();  
qDebug() << "id: " << id << "name: " << name;  
  
//获取索引为 1 的字段,即第二个字段  
QSqlField field = record.field(1);  
  
//输出字段名和字段值,结果为“name”和“MaLiang”  
qDebug() << "second field: " << field.name()  
    << "field value: " << field.value().toString();
```

使用 QSqlQuery 中的 record() 函数可以返回当前指向的记录,一条记录由 QSqlRecord 来表示,可以使用 QSqlRecord 中提供的相关函数对一条记录进行操作。而 QSqlRecord 中的 field() 函数可以返回当前记录的一个字段,它由 QSqlField 来表示,可以使用 QSqlField 中提供的相关函数来对一个字段进行操作。现在可以运行程序,查看输出结果。

3. 插入、更新和删除记录

使用 QSqlQuery 可以执行任意的 SQL 语句,下面在前面的程序中再添加代码来看一下怎样插入、更新和删除记录。这里还会涉及数值绑定的内容,使用它就可以在 SQL 语句中使用变量了。

(项目源码路径: src\17\17-5\databaseDriver)在主函数中继续添加代码:

```
query2.exec("insert into student (id, name) values (100, ChenYun");
```

这样就在连接 2 的 student 表中重新插入了一条记录。如果想在同一时间插入多条记录,一个有效的方法就是将查询语句和真实的值分离,这可以使用占位符来完成。Qt 支持两种占位符:名称绑定和位置绑定。例如,使用名称绑定,上面这条代码就等价于下面的代码片段:

```
query2.prepare("insert into student (id, name) values (:id, :name)");  
int idValue = 100;  
QString nameValue = "ChenYun";  
query2.bindValue(":id", idValue);  
query2.bindValue(":name", nameValue);  
query2.exec();
```

如果使用位置绑定,那么就等价于下面的代码片段:

```

query2.prepare("insert into student (id, name) values (?, ?)");
int idValue = 100;
QString nameValue = "ChenYun";
query2.addBindValue(idValue);
query2.addBindValue(nameValue);
query2.exec();

```

可以看到,使用这两种方法来绑定值都是很方便的,只需要注意使用的格式即可。当要插入多条记录时,只需要调用 QSqlQuery::prepare()一次,然后使用多次 bindValue()或者 addBindValue()函数来绑定需要的数据,最后调用一次 exec()函数就可以了。其实,进行多条数据插入时,还可以使用批处理进行,向程序中继续添加如下代码:

```

query2.prepare("insert into student (id, name) values (?, ?)");
QVariantList ids;
ids << 20 << 21 << 22;
query2.addBindValue(ids);
QVariantList names;
names << "xiaoming" << "xiaoliang" << "xiaogang";
query2.addBindValue(names);
if(! query2.execBatch()) qDebug() << query2.lastError();

```

这里先使用了占位符,不过每一个字段值都绑定了一个列表,最后只要调用 execBatch()函数即可,如果出现错误,可以使用 lastError()函数返回错误信息。注意这里还要添加头文件包含:

```
# include <QSqlError>
```

对于记录的更新和删除,它们和插入操作是相似的,并且也可以使用占位符。继续向主函数中添加代码:

```

//更新
query2.exec("update student set name = xiaohong where id = 20");
//删除
query2.exec("delete from student where id = 21");

```

可以使用前面的方法对整个 student 表进行遍历输出,来查看一下数据的更改。

4. 事 务

事务可以保证一个复杂操作的原子性,就是对于一个数据库操作序列,这些操作要么全部做完,要么一条也不做,它是一个不可分割的工作单位。在 Qt 中,如果底层的数据库引擎支持事务,那么 QSqlDriver::hasFeature(QSqlDriver::Transactions)会返回 true。可以使用 QSqlDatabase::transaction()来启动一个事务,然后编写一些希望在事务中执行的 SQL 语句,最后调用 QSqlDatabase::commit()或者 QSqlDatabase::

rollback()。当使用事务时必须在创建查询以前就开始事务,例如:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM employee WHERE name = 'Torild Halvor森'");
if (query.next()) {
    int employeeId = query.value(0).toInt();
    query.exec("INSERT INTO project (id, name, ownerid) "
               "VALUES (201, 'Manhattan Project', "
               + QString::number(employeeId) + ')');
}
QSqlDatabase::database().commit();
```

17.1.3 使用 SQL 模型类

除了 QSqlQuery,Qt 还提供了 3 个更高级的类来访问数据库,分别是 QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel。这 3 个类都是从 QAbstractTableModel 派生来的,可以很容易地实现将数据库中的数据在 QListView 和 QTableView 等项视图类中显示。使用这些类的另一个好处是,这样可以使编写的代码很容易地适应其他数据源。例如,如果开始使用了 QSqlTableModel,而后来要改为使用 XML 文件来存储数据,这样需要做的仅是更换一个数据模型。

1. SQL 查询模型

QSqlQueryModel 提供了一个基于 SQL 查询的只读模型。下面来看一个例子。(项目源码路径: src\17\17-6\sqlModel)新建 Qt Gui 应用,项目名称为 sqlModel,类名为 MainWindow,基类选择 QMainWindow。完成后,在 sqlModel.pro 文件中添加一行代码: QT += sql,然后保存该文件。下面再往项目中添加新的 C++ 头文件,名称为“connection.h”,完成后在其中添加数据库连接函数的定义:

```
#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("my.db");
    if (!db.open()) {
        QMessageBox::critical(0, "Cannot open database",
                             "Unable to establish a database connection.", QMessageBox::Cancel);
        return false;
    }
    QSqlQuery query;
```

```

//创建 student 表
query.exec("create table student (id int primary key, "
           "name varchar, course int)");
query.exec("insert into student values(1, '李强', 11)");
query.exec("insert into student values(2, '马亮', 11)");
query.exec("insert into student values(3, '孙红', 12)");

//创建 course 表
query.exec("create table course (id int primary key, "
           "name varchar, teacher varchar)");
query.exec("insert into course values(10, '数学', '王老师')");
query.exec("insert into course values(11, '英语', '张老师')");
query.exec("insert into course values(12, '计算机', '白老师')");

return true;
}

```

这里使用默认数据库连接创建了 student 和 course 两张表, 这里数据中的内容使用了中文, 所以要使用 `QString()` 将 SQL 语句包含起来转换编码。下面我们再到 main.cpp 文件中, 先添加头文件包含:

```
#include "connection.h"
#include <QTextCodec>
```

然后在主函数中第一行创建 QApplication 对象的代码下面添加如下代码:

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
if (!createConnection()) return 1;
```

注意设置编码的两行代码要写在前面, 不然数据库中的中文会出现乱码。下面到 mainwindow.cpp 文件中, 先添加头文件包含:

```
#include <QSqlQueryModel>
#include <QSqlTableModel>
#include <QSqlRelationalTableModel>
#include <QTableView>
#include <QDebug>
#include <QMessageBox>
#include <QSqlError>
```

然后在构造函数中添加如下代码:

```
QSqlQueryModel *model = new QSqlQueryModel(this);
```

```

model ->setQuery("select * from student");
model ->setHeaderData(0, Qt::Horizontal, tr("学号"));
model ->setHeaderData(1, Qt::Horizontal, tr("姓名"));
model ->setHeaderData(2, Qt::Horizontal, tr("课程"));

QTableView * view = new QTableView(this);
view ->setModel(model);

setCentralWidget(view);

```

这里先创建了 QSqlQueryModel 对象,然后使用 setQuery() 来执行 SQL 语句进行查询整张 student 表,并使用 setHeaderData() 来设置显示的表头。后面创建了视图,并将 QSqlQueryModel 对象作为其要显示的模型。运行程序,效果如图 17-1 所示。这里要注意,其实 QSqlQueryModel 中存储的是执行完 setQuery() 函数后的结果集,所以视图中显示的是结果集的内容。 QSqlQueryModel 中还提供了 columnCount() 返回一条记录中字段的个数;rowCount() 返回结果集中记录的条数;record() 返回第 n 条记录;index() 返回指定记录的指定字段的索引;clear() 可以清空模型中的结果集。也可以使用它提供的 query() 函数来获取 QSqlQuery 对象,这样就可以使用上一节讲到的 QSqlQuery 的相关内容来操作数据库了。还要注意一点就是,如果又使用 setQuery() 进行了新的查询,比如进行了插入操作,这时要想视图中可以显示操作后的结果,那么就必须再次查询整张表,也就是要同时执行下面两行代码:

```

model ->setQuery(QString("insert into student values(5,薛静, 10)"));
model ->setQuery("select * from student");

```

学生信息		
学号	姓名	课程
1 1	李强	11
2 2	马亮	11
3 3	孙红	12

图 17-1 SQL 查询模型运行效果

2. SQL 表格模型

QSqlTableModel 提供了一个一次只能操作一个 SQL 表的读/写模型,它是 QSqlQuery 的更高层次的替代品,可以浏览和修改独立的 SQL 表,并且只需编写很少

的代码,而且不需要了解 SQL 语法。该模型默认是可读可写的,如果想让其成为只读模型,那么可以从视图进行设置,例如:

```
view ->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

下面通过一个例子来使用该模型对数据库表进行各种操作。(项目源码路径:src\17\17-7\sqlModel)还在前面程序的基础上进行更改。先打开 mainwindow.ui 文件,向窗口上拖入 Label、Push Button、Line Edit 和 Table View 等部件,最终效果如图 17-2 所示。下面到 mainwindow.h 文件中,添加类的前置声明:

```
class QSqlTableModel;
```

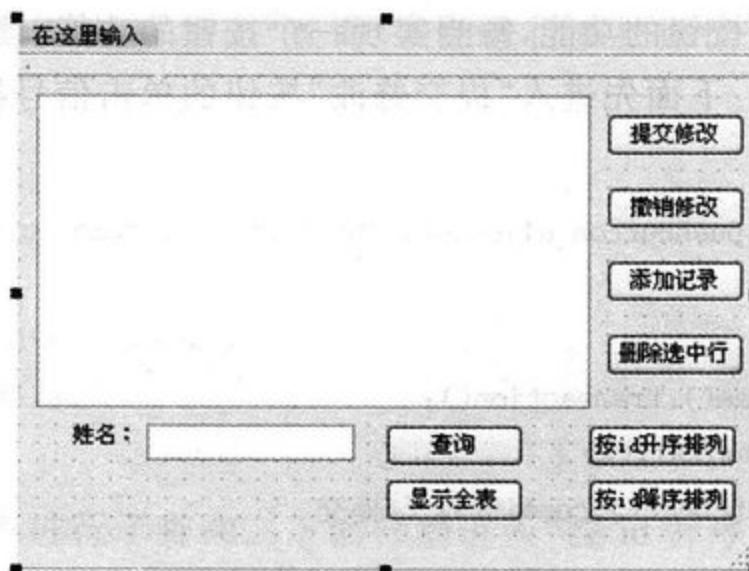


图 17-2 SQL 表格模型设计效果

然后再定义一个私有对象:

```
QSqlTableModel * model;
```

下面到 mainwindow.cpp 文件中,先将构造函数中在 17-6 中添加的代码删掉,然后再添加如下代码:

```
model = new QSqlTableModel(this);
model ->setTable("student");
model ->select();

//设置编辑策略
model ->setEditStrategy(QSqlTableModel::OnManualSubmit);
ui ->tableView ->setModel(model);
```

这里创建一个 QSqlTableModel 后,只须使用 setTable() 来为其指定数据库表,然后使用 select() 函数进行查询,调用这两个函数就等价于执行了“select * from student”这个 SQL 语句。这里还可以使用 setFilter() 来指定查询时的条件,在后面会看到这个函数的使用。在使用该模型以前,一般还要设置其编辑策略,它由 QSqlTableModel::EditStrategy 枚举变量定义,一共有 3 个值,如表 17-4 所列。用来说明当

数据库中的值被编辑后,什么情况下被提交修改。现在可以运行程序,在窗口中会显示 student 表的内容。

表 17-4 SQL 表格模型的编辑策略

常量	描述
QSqlTableModel::OnFieldChange	所有对模型的改变都会立即应用到数据库
QSqlTableModel::OnRowChange	对一条记录的改变会在用户选择另一条记录时被应用
QSqlTableModel::OnManualSubmit	所有的改变都会在模型中进行缓存,直到调用 submitAll() 或者 revertAll() 函数

下面逐个实现那些按钮的功能,每当实现一个按钮的功能,读者都可以运行一下程序,测试该按钮的效果。下面先进入“提交修改”按钮的单击信号槽,添加如下代码:

```
//提交修改按钮
void MainWindow::on_pushButton_clicked()
{
    //开始事务操作
    model->database().transaction();
    if (model->submitAll()) {
        model->database().commit(); //提交
    } else {
        model->database().rollback(); //回滚
        QMessageBox::warning(this, tr("tableModel"),
                             tr("数据库错误: %1").arg(model->lastError().text()));
    }
}
```

这里使用了事务操作,如果可以使用 submitAll() 将模型中的修改向数据库提交成功,那么执行 commit(), 否则进行回滚 rollback(), 并提示错误信息。下面进入“撤销修改”按钮的单击信号槽,添加代码:

```
//撤销修改按钮
void MainWindow::on_pushButton_2_clicked()
{
    model->revertAll();
}
```

这里只是简单调用了 revertAll() 函数将模型中的修改进行恢复。现在可以运行程序,然后修改表格中的内容,如果单击“撤销修改”,那么所有的修改都会被恢复。但是如果先按下了“提交修改”,那么数据已经提交到了数据库,再单击“撤销修改”也无法恢复了。下面再进入“查询”按钮的单击信号槽中,添加如下代码:

```
//查询按钮,进行筛选
```

```

void MainWindow::on_pushButton_7_clicked()
{
    QString name = ui->lineEdit->text();

    //根据姓名进行筛选,一定要使用单引号
    model->setFilter(QString("name = '%1'").arg(name));
    model->select();
}

```

这里使用了 setFilter() 函数来进行数据筛选, 注意, 因为姓名是中文的, 所以筛选的字符串必须使用 tr() 包含, 而且“%1”必须用单引号括起来。现在运行程序就可以在行编辑器中输入一个姓名, 然后单击“查询”按钮进行查找操作了。下面进入“显示全表”按钮的单击信号槽:

```

//显示全表按钮
void MainWindow::on_pushButton_8_clicked()
{
    model->setTable("student");
    model->select();
}

```

这里再次对整张表进行了查询。下面分别进入“按 id 升序排序”和“按 id 降序排序”按钮的单击信号槽, 更改如下:

```

//按 id 升序排列按钮
void MainWindow::on_pushButton_5_clicked()
{
    //id 字段, 即第 0 列, 升序排列
    model->setSort(0, Qt::AscendingOrder);
    model->select();
}

//按 id 降序排列按钮
void MainWindow::on_pushButton_6_clicked()
{
    model->setSort(0, Qt::DescendingOrder);
    model->select();
}

```

这里使用了 setSort() 函数来对指定的字段列进行排序。下面再进入“删除选中行”按钮的单击信号的槽, 更改如下:

```

//删除选中行按钮
void MainWindow::on_pushButton_4_clicked()
{
    //获取选中的行

```

```

int curRow = ui ->tableView ->currentIndex().row();

//删除该行
model ->removeRow(curRow);
int ok = QMessageBox::warning(this, tr("删除当前行!"),
    tr("你确定删除当前行吗?"), QMessageBox::Yes, QMessageBox::No);
if(ok == QMessageBox::No)
{ //如果不删除,则撤销
    model ->revertAll();
} else { //否则提交,在数据库中删除该行
    model ->submitAll();
}
}

```

这里先获取了当前行的行号,然后调用 removeRow() 来删除该行,这时该行的最前面会显示“!”号。删除行时会弹出一个对话框,提示是否确定要删除该行,如果确定删除,那么就执行 submitAll() 函数进行提交修改,否则执行 revertAll() 函数进行恢复。最后进入“添加记录”按钮单击信号的槽中,进行插入操作:

```

//添加记录按钮
void MainWindow::on_pushButton_3_clicked()
{
    //获得表的行数
    int rowNum = model ->rowCount();
    int id = 10;

    //添加一行
    model ->insertRow(rowNum);
    model ->setData(model ->index(rowNum, 0), id);

    //可以直接提交
    //model ->submitAll();
}

```

这里实现了在表的最后添加一条新的记录,因为 id 为主键,所以必须为其提供一个 id 值。使用 insertRow() 可以插入一行,使用 setData() 可以为一个字段设置值。这里可以调用 submitAll() 直接提交修改,如果没有提交修改,那么新添加的行的前面会显示“*”号,这样可以使用“提交修改”按钮来确认添加该行,或者使用“撤销修改”来取消添加该行。到这里整个程序就设计完毕了,可以运行程序查看效果。

3. SQL 关系表格模型

QSqlRelationalTableModel 继承自 QSqlTableModel,并且对其进行了扩展,提供了对外键的支持。一个外键就是表中的一个字段和其他表中的主键字段之间的一对一的映

射。例如,student 表中的 course 字段对应的是 course 表中的 id 字段,那么就称字段 course 是一个外键。因为这里的 course 字段的值是一些数字,这样显示很不友好,使用关系表格模型,就可以将它显示为 course 表中的 name 字段的值。下面来看一个例子。

(项目源码路径: src\17\17-8\sqlModel)在源码路径为 17-6 建立的例程的基础上修改。在 mainwindow.cpp 文件中,先删除在其中添加到构造函数中的代码,然后再添加如下代码:

```
QSqlRelationalTableModel * model = new QSqlRelationalTableModel(this);
model ->setTable("student");
model ->setRelation(2, QSqlRelation("course", "id", "name"));
model ->select();

QTableView * view = new QTableView(this);
view ->setModel(model);
setCentralWidget(view);
```

这里的 setRelation() 函数用来在两个表之间创建一个关系,其中参数“2”表示 student 表中的编号为 2 的列,即第三个字段 course 是一个外键,它映射到了 course 表中的 id 字段,而视图需要向用户显示 course 表中的 name 字段的值。运行程序,效果如图 17-3 所示。

Qt 中还提供了一个 QSqlRelationalDelegate 委托类,它可以为 QSqlRelationalTableModel 显示和编辑数据。这个委托为一个外键提供了一个 QComboBox 部件来显示所有可选的数据,这样就显得更加人性化了。使用这个委托很简单,先在 mainwindow.cpp 文件中添加头文件 #include <QSqlRelationalDelegate>,然后继续在构造函数中添加如下一行代码:

```
view ->setItemDelegate(new QSqlRelationalDelegate(view));
```

下面运行程序,效果如图 17-4 所示。

1	1	李强	英语
2	2	马亮	英语
3	3	孙红	计算机

图 17-3 SQL 关系表格模型运行效果

1	1	李强	英语
2	2	马亮	英语
3	3	孙红	<div style="border: 1px solid black; padding: 2px; display: inline-block;"> 数字 英语 计算机 </div>

图 17-4 使用关系委托运行效果

可以根据自己的需要来选择使用哪个模型。如果熟悉 SQL 语法,又不需要将所有的数据都显示出来,那么只需要使用 QSqlQuery 就可以了。对于 QSqlTableModel,它主要是用来显示一个单独表格的,而 QSqlQueryModel 可以用来显示任意一个结果集,如果想显示任意一个结果集,而且想使其可读/写,那么建议子类化 QSqlQueryModel,然后重新实现 flags() 和 setData() 函数。这部分内容可以查看 Presenting Data in a Table View 关键字对应的帮助文档,也可以参考 Query Model 示例程序。因为这 3 个模型都是基于模型/视图框架的,所以前一章讲的内容在这里都可以使用,例如可以使用 QDataWidgetMapper 等。关于数据库部分的应用,还可以参考一下 SQL 分类中的几个示例程序。

17.2 XML

XML(Extensible Markup Language,可扩展标记语言)是一种类似于 HTML 的标记语言,设计目的是传输数据,而不是显示数据。XML 的标签没有被预定义,用户需要在使用时自定义。XML 是 W3C(万维网联盟)的推荐标准。相对于数据库表格的二维表示,XML 使用的树形结构更能表现出数据的包含关系,作为一种文本文件格式,XML 简单明了的特性使得它在信息存储和描述领域非常流行。

Qt 中提供了 QtXml 模块来进行 XML 文档的处理,这里主要提供了 3 种解析方法:DOM 方法,可以进行读/写;SAX 方法,可以进行读取;基于流的方法,分别使用 QXmlStreamReader 和 QXmlStreamWriter 进行读取和写入。要在项目中使用 QtXml 模块,还需要在项目文件(.pro 文件)中添加“QT += xml”一行代码。

另外,Qt 中还提供了更高级的 QtXmlPatterns 来进行 XML 数据的查询和操作,它支持使用 XQuery 1.0 和 XPath 2.0,关于这个模块的使用可以在帮助中参考 A Short Path to XQuery 和 XQuery 关键字,还可以查看一下 XML Patterns 分类下的几个示例程序。Qt 中的 QtSvg 模块提供了 QSvgRenderer 和 QSvgGenerator 类来对 SVG(一种基于 XML 的文件格式)进行读/写,关于这些类的使用,可以参考其帮助文档,还可以查看一下 SVG Generator Example 和 SVG Viewer Example 示例程序。对应本节内容,可以在帮助中查看 XML Processing 关键字。

17.2.1 DOM

1. 使用 DOM 读取 XML 文档

DOM(Document Object Model,文档对象模型),是 W3C 的推荐标准,提供了一个接口来访问和改变一个 XML 文件的内容和结构,可以将 XML 文档表示为一个存储在内存中具有层次的树视图。文档本身由 QDomDocument 对象来表示,而文档树中所有的 DOM 节点都是 QDomNode 类的子类。

先来看一个标准的 XML 文档:

```
<? xml version = "1.0" encoding = "UTF - 8"? >
<library>
    <book id = "01">
        <title>Qt</title>
        <author>shiming</author>
    </book>
    <book id = "02">
        <title>Linux</title>
        <author>yafei</author>
    </book>
</library>
```

每个 XML 文档都由 XML 说明(或者称为 XML 序言)开始,它是对 XML 文档处理的环境和要求的说明,比如这里的<? xml version = "1.0" encoding = "UTF - 8"? >,其中 xml version = "1.0",表明使用的 XML 版本号,这里字母是区分大小写的;encoding = "UTF - 8"是使用的编码,指出文档是使用何种字符集建立的,默认值为 Unicode 编码。Qt 中使用 QDomProcessingInstruction 类来表示 XML 说明。XML 文档内容由多个元素组成,一个元素由起始标签<标签名>、终止标签</标签名>以及两个标签之间的内容组成,而文档中第一个元素被称为根元素,比如这里的<library></library>,XML 文档必须有且只有一个根元素。元素的名称是区分大小写的,元素还可以嵌套,比如这里的 library、book、title 和 author 等都是元素。元素对应 QDomElement 类。元素可以包含属性,用来描述元素的相关信息,属性名和属性值在元素的起始标签中给出,格式为<元素名 属性名 = "属性值">,如<book id = "01">,属性值必须在单引号或者双引号中。属性对应 QDomAttr 类。在元素中可以包含子元素,也可以只包含文本内容,比如这里的<title>Qt</title>中的 Qt 就是文本内容,文本内容由 QDomText 类表示。在 Qt 中,所有的 DOM 节点,比如这里的说明、元素、属性和文本等,都使用 QDomNode 来表示,然后使用对应的 isProcessingInstruction()、isElement()、isAttr() 和 isText() 等函数来判断是否是该类型的元素,如果是,那么就可以使用 toProcessingInstruction()、toElement()、toAttr() 和 toText() 等函数转换为具体的节点类型。

这里对 XML 文档格式进行了一个简单的介绍,只是为了让没有 XML 知识的读者可以快速学习本节的内容。如果要应用 XML,还是有必要了解一下它的基本语法内容的,这个可以参考其他的书籍或者网络内容(例如: <http://www.w3school.com.cn/x.asp>)。下面就来使用 Qt 中的 DOM 类读取一个 XML 文档。

(项目源码路径: src\17\17-9\myDOM1)新建 Qt4 控制台应用,名称为 myDOM1。完成后在 myDOM1.pro 文件中添加如下一行代码:

```
QT += xml
```

保存该文件。然后再回到新建的项目目录中,新建记事本文本文档,然后将前面介绍

的标准 XML 文档编辑进来, 最后以“my. xml”为文件名保存, 注意后缀要更改为“. xml”。下面到 main. cpp 文件中, 将其内容更改为:

```
# include <QtCore/QCoreApplication>
# include <QtXml>
```

```
int main(int argc, char * argv[])
{
    QCoreApplication a(argc, argv);
```

```
    //新建 QDomDocument 类对象, 它代表一个 XML 文档
```

```
    QDomDocument doc;
    QFile file("../myDOM1/my.xml");
    if (!file.open(QIODevice::ReadOnly)) return 0;
```

```
    //将文件内容读到 doc 中
```

```
    if (!doc.setContent(&file)) {
        file.close();
        return 0;
    }
```

```
    //关闭文件
```

```
    file.close();
```

```
    //获得 doc 的第一个结点, 即 XML 说明
```

```
    QDomNode firstNode = doc.firstChild();
```

```
    //输出 XML 说明, nodeName() 为“xml”, nodeValue() 为版本和编码信息
```

```
    qDebug() << qPrintable(firstNode.nodeName())
        << qPrintable(firstNode.nodeValue());
```

```
    //返回根元素
```

```
    QDomElement docElem = doc.documentElement();
```

```
    //返回根节点的第一个子结点
```

```
    QDomNode n = docElem.firstChild();
```

```
    //如果结点不为空, 则转到下一个节点
```

```
    while(!n.isNull())
    {
```

```
        //如果结点是元素
```

```
        if (n.isElement())
        {
```

```

//将其转换为元素
QDomElement e = n.toElement();

//返回元素标记和 id 属性的值
qDebug() << qPrintable(e.tagName())
    << qPrintable(e.attribute("id"));

//获得元素 e 的所有子结点的列表
QDomNodeList list = e.childNodes();

//遍历该列表
for(int i = 0; i < list.count(); i++)
{
    QDomNode node = list.at(i);
    if(node.isElement())
        qDebug() << " " << qPrintable(node.toElement().tagName())
            << qPrintable(node.toElement().text());
}

//转到下一个兄弟结点
n = n.nextSibling();
}

return a.exec();
}

```

这里先创建了一个 QDomDocument 类对象,用来代表整个 XML 文档。QDomDocument 类提供了对文档数据最基本的访问。然后使用 QFile 类打开了指定的 XML 文件,使用 QDomDocument 类的 setContent() 函数来设置整个文档的内容,它会将 XML 文档的内容解析为一个 DOM 树,并保存在内存中,所以完成后就可以使用 close() 函数把文件关闭了。QDomDocument 类也是 QDomNode 的子类,使用 firstChild() 函数可以获取它的第一个子节点,这里就是 XML 说明。使用 documentElement() 函数可以获得根节点,这也是访问 XML 文档的入口,它返回的是一个 QDomElement 类对象,因为这个对象也是 QDomNode 的子类,所以后面就可以使用 QDomNode 类提供的一些函数来遍历整个文档了,比如 firstChild() 获得第一个子节点, lastChild() 获得最后一个节点, childNodes() 获得该节点的所有孩子节点的一个列表, nextSibling() 获得下一个兄弟节点, previousSibling() 获得前一个兄弟节点。对于一个元素节点,可以使用 tagName() 来获取标签名,使用 attribute() 来获取指定的属性的值,使用 text() 来获取其中的文本内容。现在可以运行程序,查看输出结果,如图 17-5 所示。

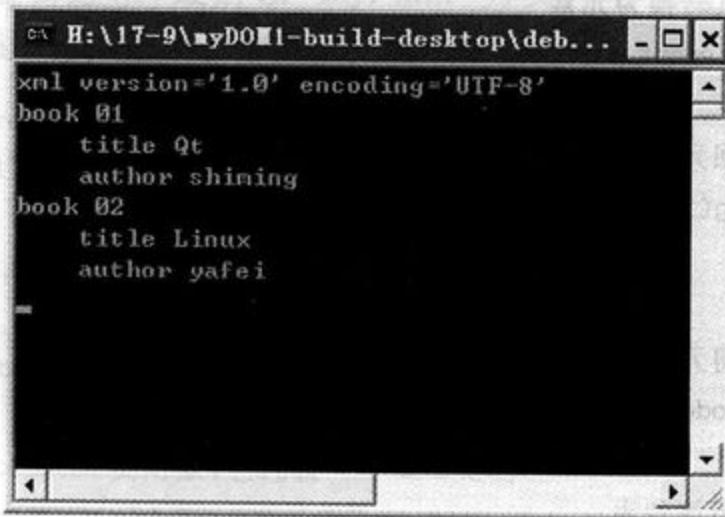


图 17-5 使用 DOM 读取 XML 文件运行效果

2. 使用 DOM 创建和操作 XML 文档

(项目源码路径: src\17\17-10\myDOM2)新建 Qt Gui 应用,名称为 myDOM2,类名为 MainWindow,基类为 QMainWindow。完成后还是先在项目文件 myDOM2.pro 中添加“QT += xml”来导入 QDom 模块。然后打开 mainwindow.ui 文件,往其中拖入 Push Button、List Widget、Label 和 Line Edit 等部件来设计界面,最终效果如图 17-6 所示。下面进入 main.cpp 文件,添加头文件 #include <QTextCodec>,然后在 main() 函数中添加如下一行代码:

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
```

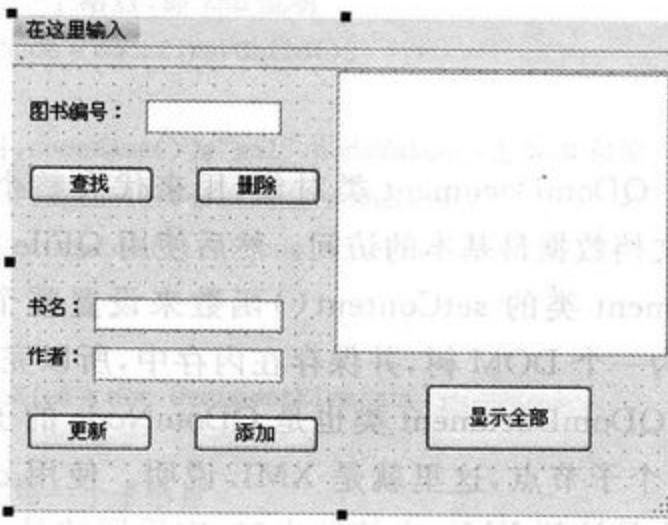


图 17-6 使用 DOM 操作 XML 文档设计界面

这样就可以在后面的代码中使用 QString() 来为中文进行编码了。下面到 mainwindow.cpp 文件中先添加头文件:

```
#include <QtXml>
#include <QFile>
```

然后在构造函数中添加代码来生成 XML 文件:

```

QDomDocument doc;

//添加处理指令即 XML 说明
QDomProcessingInstruction instruction;
instruction = doc.createProcessingInstruction("xml","version = \"1.0\" encoding = \"UTF
-8\"");
doc.appendChild(instruction);

//添加根元素
QDomElement root = doc.createElement(QString("书库"));
doc.appendChild(root);

//添加第一个图书元素及其子元素
QDomElement book = doc.createElement(QString("图书"));
QDomAttr id = doc.createAttribute(QString("编号"));
QDomElement title = doc.createElement(QString("书名"));
QDomElement author = doc.createElement(QString("作者"));
QDomText text;
id.setValue(QString("1"));
book.setAttributeNode(id);
text = doc.createTextNode(QString("Qt"));
title.appendChild(text);
text = doc.createTextNode(QString("shiming"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);

//添加第二个图书元素及其子元素
book = doc.createElement(QString("图书"));
id = doc.createAttribute(QString("编号"));
title = doc.createElement(QString("书名"));
author = doc.createElement(QString("作者"));
id.setValue(QString("2"));
book.setAttributeNode(id);
text = doc.createTextNode(QString("Linux"));
title.appendChild(text);
text = doc.createTextNode(QString("yafei"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);

```

```
QFile file("my.xml");
if(! file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
QTextStream out(&file);

//将文档保存到文件,4 为子元素缩进字符数
doc.save(out, 4);
file.close();
```

这里先使用 QDomDocument 类在内存中生成了一棵 DOM 树,然后调用 save() 函数利用 QTextStream 文本流将 DOM 树保存在了文件中。生成 DOM 树时主要使用了 createElement() 等函数来生成各种节点,然后使用 appendChild() 将各个节点依次追加进去。现在运行程序就可以在项目生成的目录中查看到创建的 my.xml 文件了,可以直接打开,这样默认会在浏览器中打开,也可以使用记事本等编辑器将其打开,当然,还可以将其拖入 Qt Creator 中,使用 Qt Creator 将其打开。下面来输出整个文档的内容。

在设计模式,转到“显示全部”按钮的单击信号的槽中,更改代码如下:

```
void MainWindow::on_pushButton_5_clicked()
{
    //先清空显示
    ui->listWidget->clear();
    QFile file("my.xml");
    if (! file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (! doc.setContent(&file))
    {
        file.close();
        return ;
    }
    file.close();
    QDomElement docElem = doc.documentElement();
    QDomNode n = docElem.firstChild();
    while(! n.isNull())
    {
        if (n.isElement())
        {
            QDomElement e = n.toElement();
            ui->listWidget->addItem(e.tagName() + e.attribute(QString("编号")));
            QDomNodeList list = e.childNodes();
            for (int i = 0; i < list.count(); i++)
            {
                QDomNode node = list.at(i);
                if(node.isElement())
```

```

        ui->listWidget->addItem(" " + node.toElement().tagName()
                                + " : " + node.toElement().text());
    }
}
n = n.nextSibling();
}
}

```

这里的代码就是前面读取 XML 文档时的代码。现在运行程序，然后单击“显示全部”按钮，则会在列表部件中显示出文档中的所有内容。下面来实现向文档中添加一个元素。转到“添加”按钮的单击信号的槽中，添加如下代码：

```

void MainWindow::on_pushButton_4_clicked()
{
    //先清空显示，然后显示“无法添加！”，这样如果添加失败则会显示“无法添加！”
    ui->listWidget->clear();
    ui->listWidget->addItem(QString("无法添加！"));
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
        file.close();
        return;
    }
    file.close();
    QDomElement root = doc.documentElement();
    QDomElement book = doc.createElement(QString("图书"));
    QDomAttr id = doc.createAttribute(QString("编号"));
    QDomElement title = doc.createElement(QString("书名"));
    QDomElement author = doc.createElement(QString("作者"));
    QDomText text;

    //我们获得了最后一个孩子结点的编号，然后加 1，便是新的编号
    QString num = root.lastChild().toElement().attribute(QString("编号"));
    int count = num.toInt() + 1;
    id.setValue(QString::number(count));
    book.setAttributeNode(id);
    text = doc.createTextNode(ui->lineEdit_2->text());
    title.appendChild(text);
    text = doc.createTextNode(ui->lineEdit_3->text());
    author.appendChild(text);
    book.appendChild(title);
    book.appendChild(author);
}

```

```
root.appendChild(book);
if(! file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
QTextStream out(&file);
doc.save(out, 4);
file.close();

//最后更改显示为“添加成功!”
ui->listWidget->clear();
ui->listWidget->addItem(QString("添加成功!"));
}
```

向文档中添加元素的过程：先使用只读方式打开 xml 文件，然后将其解析为内存中的 DOM 树并关闭文件，再向 DOM 树中添加元素，最后使用只写方式打开 xml 文件，将 DOM 树写入到文件并关闭文件。现在运行程序，在书名和作者行编辑器中输入内容，然后按下“添加按钮”，最后按下“显示全部”按钮，就可以看到添加后的内容了。

因为查找、删除和更新内容都是对指定元素进行的，所以它们可以在一个函数中实现。下面先在 mainwindow.h 文件中添加一个 public 函数声明：

```
void doXml(const QString operate);
```

然后到 mainwindow.cpp 文件添加该函数的定义：

```
void MainWindow::doXml(const QString operate)
{
    ui->listWidget->clear();
    ui->listWidget->addItem(QString("没有找到相关内容!"));
    QFile file("my.xml");
    if (! file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (! doc.setContent(&file))
    {
        file.close();
        return ;
    }
    file.close();

    //以标签名进行查找
    QDomNodeList list = doc.elementsByTagName(QString("图书"));
    for(int i = 0; i<list.count(); i++)
    {
        QDomElement e = list.at(i).toElement();
        if(e.attribute(QString("编号")) == ui->lineEdit->text())
        { //如果元素的“编号”属性值与我们所查的相同
            if (operate == "delete")
            {
                //从 DOM 树中移除该元素
                list.removeAt(i);
                doc.documentElement().removeChild(list.item(i));
                doc.documentElement().appendChild(list.item(i));
                file.setDevice(&doc);
                file.open(QIODevice::WriteOnly);
                doc.save(&file, 4);
                file.close();
                ui->listWidget->clear();
                ui->listWidget->addItem(QString("删除成功!"));
            }
        }
    }
}
```

```

//如果是删除操作
QDomElement root = doc.documentElement();

//从根节点上删除该节点
root.removeChild(list.at(i));
QFile file("my.xml");
if(! file.open(QIODevice::WriteOnly | QIODevice::Truncate))
    return ;
QTextStream out(&file);
doc.save(out,4);
file.close();
ui->listWidget->clear();
ui->listWidget->addItem(QString("删除成功!"));
} else if (operate == "update") {
    //如果是更新操作
    QDomNodeList child = list.at(i).childNodes();
    //将它子节点的首个子节点(就是文本节点)的内容更新
    child.at(0).toElement().firstChild()
        .setNodeValue(ui->lineEdit_2->text());
    child.at(1).toElement().firstChild()
        .setNodeValue(ui->lineEdit_3->text());
    QFile file("my.xml");
    if(! file.open(QIODevice::WriteOnly | QIODevice::Truncate))
        return ;
    QTextStream out(&file);
    doc.save(out,4);
    file.close();
    ui->listWidget->clear();
    ui->listWidget->addItem(QString("更新成功!"));
} else if (operate == "find") {
    //如果是查找操作
    ui->listWidget->clear();
    ui->listWidget->addItem(e.tagName() + e.attribute(QString("编
号")));
    QDomNodeList list = e.childNodes();
    for(int i = 0; i < list.count(); i++)
    {
        QDomNode node = list.at(i);
        if(node.isElement())
            ui->listWidget->addItem(" " +
                + node.toElement().tagName() + ":" +
                + node.toElement().text());
    }
}

```

这里先使用 `elementsByTagName()` 来获取了所有图书元素的列表，然后使用指定的 `id` 编号来获取要操作的图书元素，后面分为 3 种情况来处理。如果是删除操作，那么就使用 `removeChild()` 函数来删除该元素并保存到文件；如果是更新操作，那么就使用 `setNodeValue()` 来为其设置新的值并保存到文件；如果是查找操作，就将该元素的内容显示出来。下面分别进入“查找”按钮、“删除”按钮和“更新”按钮的单击信号槽中，更改如下：

```
//查找按钮
void MainWindow::on_pushButton_clicked()
{
    doXml("find");
}

//删除按钮
void MainWindow::on_pushButton_2_clicked()
{
    doXml("delete");
}

//更新按钮
void MainWindow::on_pushButton_3_clicked()
{
    doXml("update");
}
```

下面运行程序，在图书编号行编辑器中输入图书的编号，然后进行查找、删除和更新等操作，查看一下效果。通过这个例子可以看到使用 DOM 可以很方便地进行 XML 文档的随机访问，这也是它最大的优点。关于 DOM 的使用，还可以参考一下 Qt 提供的 DOM Bookmarks example 示例程序。

17.2.2 SAX

SAX(Simple API for XML)为 XML 解析器提供了一个基于事件的标准接口。在 Qt 中支持 SAX2,但是并不支持 Java 接口中的 SAX1。在前面讲解的 DOM 方法需要在一个树结构中读入和存储整个 XML 文档,这样会耗费大量内存,不过使用它来操作文档结构是很容易的。如果不需要对文档进行操作,而只需要读取整个 XML 文档,那么使用 SAX 方法就 very 高效了。SAX2 接口是一个事件驱动机制,用来为用户提供文档信息。这里的事件是由解析器发出的,例如它遇到了一个开始标签或者一个结束标签等。下面来看一个例子:

```
<quote>A quotation.</quote>
```

当解析上面这行文档时会触发 3 个事件：

- ① 遇到开始标签(<quote>),这时会调用 startElement()事件处理函数;
- ② 发现字符数据“A quotation.”,这时会调用 characters()事件处理函数;
- ③ 一个结束标签被解析(</quote>),这时会调用 endElement()事件处理函数。

每当发生一个事件时,都可以在相应的事件处理函数中进行操作来完成对文档自定义解析。比如可以在 startElement()中获得元素名和属性,在 characters()中获元素中的文本,在 endElement()中进行一些结束读取该元素时想要进行的操作。这些事件处理函数都可以通过继承 QXmlDefaultHandler 类来重新实现,Qt 中提供的事件处理类如表 17-5 所列,它们都继承自 QXmlDefaultHandler 类。

表 17-5 QtXml 模块中的事件处理类

处理类	描述
QXmlContentHandler	报告与文档内容相关的事件(例如起始标签或字符)
QXmlDTDHandler	报告与 DTD 相关的事件
QXmlErrorHandler	报告在解析过程中发生的错误或警告
QXmlEntityResolver	报告解析过程中的外部实体,允许用户解析外部实体
QXmlDeclHandler	报告 XML 数据的声明内容
QXmlLexicalHandler	报告与文档的词法结构相关的事件

(项目源码路径: src\17\17-11\mySAX)新建空的 Qt 项目,项目名称为 mySAX。完成后向项目中添加新的 C++ 类,类名为 MySAX,基类为 QXmlDefaultHandler,头文件选择“无”。完成后,先在 mySAX.pro 文件中添加“QT+=xml”,然后保存该文件。下面到 mysax.h 文件中,将 MySAX 类的定义修改为:

```
#include <QXmlDefaultHandler>
class QListWidget;

class MySAX : public QXmlDefaultHandler
{
public:
    MySAX();
    ~MySAX();
    bool readFile(const QString &fileName);
protected:
    bool startElement (const QString &namespaceURI, const QString &localName, const
    QString &qName, const QXmlAttributes &atts);
    bool endElement(const QString &namespaceURI, const QString &localName, const QString
    &qName);
    bool characters(const QString &ch);
```

```

    bool fatalError(const QDomParseException &exception);
private:
    QListWidget *list;
    QString currentText;
};

```

这里主要是重新声明了 `QXmlDefaultHandler` 类的 `startElement()`、`endElement()`、`characters()` 和 `fatalError()` 几个函数, `readFile()` 函数用来读入 XML 文件, `QListWidget` 部件用来显示解析后的 XML 文档内容, `currentText` 字符串变量用于暂存字符数据。下面到 `mysax.cpp` 文件中, 将其内容修改为:

```

#include "mysax.h"
#include <QtXml>
#include <QListWidget>

MySAX::MySAX()
{
    list = new QListWidget;
    list->show();
}

MySAX::~MySAX()
{
    delete list;
}

bool MySAX::readFile(const QString &fileName)
{
    QFile file(fileName);

    //读取文件内容
    QDomInputSource inputSource(&file);

    //建立 QDomSimpleReader 对象
    QDomSimpleReader reader;

    //设置内容处理器
    reader.setContentHandler(this);

    //设置错误处理器
    reader.setErrorHandler(this);

    //解析文件

```

```

    return reader.parse(inputSource);
}

//已经解析完一个元素的起始标签
bool MySAX::startElement (const QString &namespaceURI, const QString &localName,
QString &qName, const QDomAttributes &atts)
{
    if (qName == "library")
        list ->addItem(qName);
    else if (qName == "book")
        list ->addItem("      " + qName + atts.value("id"));
    return true;
}

//已经解析完一块字符数据
bool MySAX::characters(const QString &ch)
{
    currentText = ch;
    return true;
}

//已经解析完一个元素的结束标签
bool MySAX::endElement (const QString &namespaceURI, const QString &localName,
QString &qName)
{
    if (qName == "title" || qName == "author")
        list ->addItem("      " + qName + " : " + currentText);
    return true;
}

//错误处理
bool MySAX::fatalError(const QDomParseException &exception)
{
    qDebug() << exception.message();
    return false;
}

```

这里添加了几个函数的定义。readFile()函数中设置了文件的解析过程。Qt 提供了一个简单的 XML 解析器 QXmlSimpleReader, 它是基于 SAX 的。该解析器需 QXmlInputSource 为其提供数据, QXmlInputSource 会使用相应的编码来读取 XML 文档的数据。解析之前还需要使用 setContentHandler() 来设置事件处理器, 使用 setErrorHandler() 来设置错误处理器, 它们的参数使用了 this, 表明使用本类作为处理器, 也就是在解析过程中出现的各种事件都会使用本类的 startElement() 等事件处理器。

函数来进行处理,而出现错误时会使用本类的 `fatalError()` 函数来处理。最后,调用了 `parse()` 函数来进行解析,该函数会在解析成功时返回 `true`,否则返回 `false`。在后面的几个事件处理函数中,就是简单地将数据显示在 `QListWidget` 中。

下面再向项目中添加 `main.cpp` 文件,并更改其内容如下:

```
#include "mysax.h"
#include < QApplication >

int main( int argc, char * argv[] )
{
    QApplication app( argc, argv );
    MySAX sax;
    sax.readFile( "../mySAX/my.xml" );
    return app.exec();
}
```

现在,把在前面源码路径为 17-9 的例程中创建的 `my.xml` 文件复制到现在的项目目录中,然后运行程序,就可以显示出该文档的内容了。可以看到使用 SAX 方法来解析 XML 文档比使用 DOM 方法要清晰很多,更重要的是它的效率要高很多,不过 SAX 方法只适用于读取 XML 文档。对于 SAX 的使用,也可以参考一下 Qt 自带的 SAX Bookmarks example 示例程序。

17.2.3 XML 流

从 Qt 4.3 开始引入了两个新的类来读取和写入 XML 文档: `QXmlStreamReader` 和 `QXmlStreamWriter`。`QXmlStreamReader` 类提供了一个快速的解析器通过一个简单的流 API 来读取格式良好的 XML 文档,它是作为 Qt 的 SAX 解析器替代品的身份出现的,因为它比 SAX 解析器更快更方便。`QXmlStreamReader` 可以从 `QIODevice` 或者 `QByteArray` 中读取数据。流读取器的基本原理就是将 XML 文档报告为一个记号 (`tokens`) 流,这一点与 SAX 相似,不同之处在于 XML 记号被报告的方式。在 SAX 中,应用程序必须提供处理器 (回调函数) 来从解析器获得所谓的 XML 事件;而对于 `QXmlStreamReader`,是应用程序代码自身来驱动循环,需要的时候可以从读取器中一个接一个地拉出记号。这个是通过调用 `readNext()` 函数实现的,它可以读取下一个记号,然后返回一个记号类型,它由枚举变量 `QXmlStreamReader::TokenType` 定义,其所有取值如表 17-6 所列。然后可以使用 `isStartElement()` 和 `text()` 等函数来判断这个记号是否包含需要的信息。使用这种主动拉取记号方式的最大好处就是可以构建递归解析器,也就是可以在不同的函数或者类中来处理 XML 文档中的不同记号。

表 17-6 在 QXmlStreamReader 中的记号类型

常量	描述
QXmlStreamReader::NoToken	没有读到任何内容
QXmlStreamReader::Invalid	发生了一个错误，在 error() 和 errorString() 中报告
QXmlStreamReader::StartDocument	在 documentVersion() 中报告 XML 版本号，在 documentEncoding() 中指定文档的编码
QXmlStreamReader::EndDocument	报告文档结束
QXmlStreamReader::StartElement	使用 namespaceURI() 和 name() 来报告元素开始，可以使用 attributes() 来获取属性
QXmlStreamReader::EndElement	使用 namespaceURI() 和 name() 来报告元素结束
QXmlStreamReader::Characters	使用 text() 来报告字符，如果字符是空白，那么 isWhitespace() 返回 true，如果字符源于 CDATA 部分，那么 isCDATA() 返回 true
QXmlStreamReader::Comment	使用 text() 报告一个注释
QXmlStreamReader::DTD	使用 text() 来报告一个 DTD，符号声明在 notationDeclarations() 中，实体声明在 entityDeclarations() 中，具体的 DTD 声明通过 dtdName()、dtdPublicId() 和 dtdSystemId() 来报告
QXmlStreamReader::EntityReference	报告一个无法解析的实体引用，引用的名字由 name() 获取，text() 可以获取替换文本
QXmlStreamReader::ProcessingInstruction	使用 processingInstructionTarget() 和 processingInstructionData() 来报告一个处理指令

下面来看一个使用 QXmlStreamReader 解析 XML 文档的例子。（项目源码路径：src\17\17-12\myXmlStream）新建 Qt4 控制台应用，名称为 myXmlStream，完成后向 myXmlStream.pro 文件中添加“QT += xml”这一行代码，然后保存该文件。下面到 main.cpp 文件中，将其代码更改为：

```
#include <QtCore/QCoreApplication>
#include <QFile>
#include <QXmlStreamReader>
#include <QXmlStreamWriter>
#include <QDebug>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QFile file("../myXmlStream/my.xml");
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
```

```
qDebug() << "Error: cannot open file";
return 1;
}

QXmlStreamReader reader;

//设置文件,这时会将流设置为初始状态
reader.setDevice(&file);

//如果没有读到文档结尾,而且没有出现错误
while (! reader.atEnd()) {
    //读取下一个记号,它返回记号的类型
    QXmlStreamReader::TokenType type = reader.readNext();

    //下面便根据记号的类型来进行不同的输出
    if (type == QXmlStreamReader::StartDocument)
        qDebug() << reader.documentEncoding() << reader.documentVersion();
    if (type == QXmlStreamReader::StartElement) {
        qDebug() << "<" << reader.name() << ">";
        if (reader.attributes().hasAttribute("id"))
            qDebug() << reader.attributes().value("id");
    }
    if (type == QXmlStreamReader::EndElement)
        qDebug() << "</>" << reader.name() << ">";
    if (type == QXmlStreamReader::Characters && ! reader.isWhitespace())
        qDebug() << reader.text();
}

//如果读取过程中出现错误,那么输出错误信息
if (reader.hasError()) {
    qDebug() << "error: " << reader.errorString();
}

file.close();

return a.exec();
}
```

可以看到流读取器就在一个循环中通过使用 `readNext()` 来不断读取记号, 这里可以对不同的记号和不同的内容进行不同的处理, 既可以在本函数中进行, 也可以在其他函数或者其他类中进行。下面将前面源码路径为 17-9 的例程中创建的 `my.xml` 文件复制到该项目目录中, 然后运行程序查看效果。

与 QXmlStreamReader 对应的是 QXmlStreamWriter, 它通过一个简单的流 API 提供了一个 XML 写入器。QXmlStreamWriter 的使用是十分简单的, 只需要调用相应记号的写入函数来写入相关数据即可。下面通过一个例子来进行讲解。(项目源码路径: src\17\17-13\myXmlStream) 将前面主函数的内容更改如下:

```

int main(int argc, char * argv[])
{
    QCoreApplication a(argc, argv);

    QFile file("../myXmlStream/my2.xml");
    if (!file.open(QFile::WriteOnly | QFile::Text))
    {
        qDebug() << "Error: cannot open file";
        return 1;
    }

    QXmlStreamWriter stream(&file);
    stream.setAutoFormatting(true);
    stream.writeStartDocument();
    stream.writeStartElement("bookmark");
    stream.writeAttribute("href", "http://qt.nokia.com/");
    stream.writeTextElement("title", "Qt Home");
    stream.writeEndElement();
    stream.writeEndDocument();

    file.close();

    qDebug() << "write finished!";

    return a.exec();
}

```

这里使用了 setAutoFormatting(true) 函数来自动设置格式, 这样会自动换行和添加缩进。然后使用了 writeStartDocument(), 该函数会自动添加首行的 XML 说明(即<? xml version="1.0" encoding="UTF-8"?>), 添加元素可以使用 writeStartElement(), 不过, 这里要注意, 一定要在元素的属性、文本等添加完成后, 使用 writeTextElement() 来关闭前一个打开的元素。在最后使用 writeEndDocument() 来完成文档的写入。现在可以运行程序了, 这时会在项目目录中生成一个 XML 文档。对于 QXmlStreamReader 和 QXmlStreamWriter 的使用, 还可以参考 QXmlStream Bookmarks Example 示例程序。

17.3 小结

数据库和 XML 在很多程序中经常用到，它们的使用也总是和数据的显示联系起来，所以学习好前面一章的知识也是很重要的，这两章密不可分。这一章只是讲解了数据库和 XML 最简单的应用，要深入研究，还需要去学习相关专业知识。在《Qt 及 Qt Quick 开发实战精解》中的数据管理系统实例综合应用了数据库和 XML 的知识，学习完本章可以接着学习该实例。

通过本章的学习，读者应该能够掌握如何使用 SQLite 和 XML 来完成一些基础的数据操作。通过本章的实践，读者应该已经对 Qt Creator 的界面有了初步的了解，并且能够熟练地使用 Qt Designer 来设计界面上的控件。通过本章的练习，读者应该已经掌握了如何使用 QSqlQuery 和 QDomDocument 来操作数据库和 XML 文档。通过本章的实验，读者应该已经能够将所学的知识应用到实际的项目开发中去，从而提高自己的编程水平。

操作。还有一个连接到子线程的线程可以执行一个或者多个任务，从而让线程池能够处理更多的任务。

在通过线程池时，线程池的线程会将任务分派给不同的线程，线程名分别为 WorkerThread、WorkerThread2、WorkerThread3、WorkerThread4、WorkerThread5、WorkerThread6、WorkerThread7、WorkerThread8、WorkerThread9、WorkerThread10。如果线程池中没有足够的线程，那么线程池会添加新的线程。也就是说，线程池的线程数是固定的，而线程数的多少由线程池的线程数决定。

Brewster，而且是大量的线程，所以线程池的线程数不能设置得过大，否则会导致系统资源耗尽，从而导致系统崩溃。

网络通信篇

➤ 第 18 章 网络编程

➤ 第 19 章 进程和线程

➤ 第 20 章 WebKit



第 18 章

网络编程

Qt 提供了 QtNetwork 模块来进行网络编程。该模块提供了诸如 QFtp 等类来实现特定的应用层协议；有较低层次的类，例如 QTcpSocket、QTcpServer 和 QUdpSocket 等来表示低层的网络概念；还有高层次的类，例如 QNetworkRequest、QNetworkReply 和 QNetworkAccessManager 使用相同的协议来执行网络操作；也提供了 QNetworkConfiguration、QNetworkConfigurationManager 和 QNetworkSession 等类来实现负载管理。如果要使用 QtNetwork 模块中的类，需要在项目文件中添加“QT+=network”一行代码。对应本章的内容，可以在帮助中参考 Network Programming 关键字。《Qt 及 Qt Quick 开发实战精解》中的局域网聊天工具实例是一个使用网络模块的综合应用实例程序。

18.1 HTTP

HTTP(HyperText Transfer Protocol, 超文本传输协议)是一个客户端和服务器端请求和应答的标准。在 Qt 的网络模块中提供了网络访问接口来实现 HTTP 编程。网络访问接口是执行一般网络操作的类集合，该接口在特定的操作和使用的协议(例如，通过 HTTP 进行获取和发送数据)上提供了一个抽象层，只为外部暴露出了类、函数和信号。网络请求由 QNetworkRequest 类来表示，也作为与请求有关的信息(例如，任何头信息和使用加密)的容器。在创建请求对象时指定的 URL 决定了请求使用的协议，目前支持 HTTP、FTP 和本地文件 URLs 的上传和下载。QNetworkAccessManager 类用来协调网络操作，每当一个请求创建后，该类用来调度它，并发射信号来报告进度。该类还协调 cookies 的使用、身份验证请求及其代理的使用等。对于网络请求的应答使用 QNetworkReply 类表示，它会在请求被完成调度时由 QNetworkAccessManager 来创建。QNetworkReply 提供的信号可以用来单独监视每一个应答，当然也可以使用 QNetworkAccessManager 的信号来实现。因为 QNetworkReply 是 QIODevice 的子类，应答可以使用同步或者异步的方式来处理，例如阻塞或者非阻塞的

操作。每一个应用程序或者库文件都可以创建一个或者多个 `QNetworkAccessManager` 实例来处理网络通信。

(项目源码路径: `src\18\18-1\myHTTP`)新建 Qt Gui 应用,名称为 myHTTP,名为 MainWindow,基类保持 QMainWindow 不变。完成后先在 `myHTTP.pro` 文件添加代码“`QT+=network`”,并保存该文件。进入设计模式,往界面上拖入一个 `TextBrowser`,然后进入 `mainwindow.h` 文件,先添加类的前置声明:

```
class QNetworkReply;
class QNetworkAccessManager;
```

然后添加一个私有对象定义:

```
QNetworkAccessManager * manager;
```

下面再添加一个私有槽的声明:

```
private slots:
    void replyFinished(QNetworkReply *);
```

现在到 `mainwindow.cpp` 文件中,先添加头文件包含:

```
#include <QtNetwork>
#include <QTextCodec>
```

然后在构造函数中添加如下代码:

```
manager = new QNetworkAccessManager(this);
connect(manager, SIGNAL(finished(QNetworkReply *)), this, SLOT(replyFinished(QNetworkReply *)));
manager->get(QNetworkRequest(QUrl("http://www.baidu.com")));
```

这里先创建了一个 `QNetworkAccessManager` 类的实例,它用来发送网络请求接收应答。然后关联了管理器的 `finished()` 信号和自定义的槽,每当网络应答结束都会发射这个信号。最后使用了 `get()` 函数来发送一个网络请求,网络请求使用 `QNetworkRequest` 类表示, `get()` 函数返回一个 `QNetworkReply` 对象。除了 `get()` 函数,管理器还提供了发送 HTTP POST 请求的 `post()` 函数。下面添加槽的定义:

```
void MainWindow::replyFinished(QNetworkReply * reply)
{
    QTextCodec * codec = QTextCodec::codecForLocale();
    QString all = codec->toUnicode(reply->readAll());
    ui->textBrowser->setText(all);
    reply->deleteLater();
}
```

因为 `QNetworkReply` 继承自 `QIODevice` 类,所以可以像操作一般的 I/O 设备一样来操作该类。这里使用了 `readAll()` 函数来读取所有的应答数据,为了正常显示

文, 使用了 QTextCodec 类来转换编码。在完成数据的读取后, 需要使用 deleteLater() 来删除 reply 对象。现在运行程序, 效果如图 18-1 所示。

下面将前面的程序进行扩展, 实现一般文件的下载, 并且显示下载进度。(项目源码路径: src\18\18-2\myHTTP)先删除界面上的 Text Browser, 然后拖入 Label、Line Edit、Progress Bar 和 Push Button 等部件, 最终效果如图 18-2 所示。下面到 mainwindow.h 文件中, 先添加头文件和类的前置声明:

```
# include <QUrl>
class QFile;
```



图 18-1 使用 HTTP 下载网页运行效果

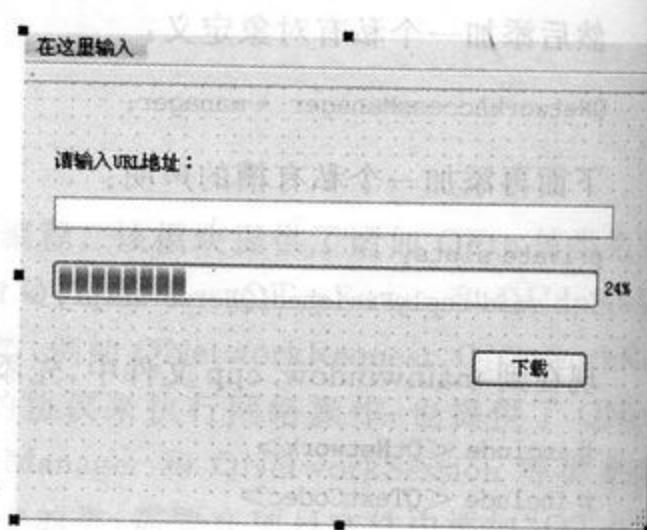


图 18-2 使用 HTTP 下载文件界面设计效果

然后将前面的 replyFinished(QNetworkReply *)槽声明删除掉, 并添加如下私有槽声明:

```
void httpFinished();
void httpReadyRead();
void updateDataReadProgress(qint64, qint64);
```

然后再添加一个 public 函数声明:

```
void startRequest(QUrl url);
```

最后添加几个私有对象定义:

```
QNetworkReply * reply;
QUrl url;
QFile * file;
```

下面到 mainwindow.cpp 文件中, 将前面在构造函数中添加的内容删除, 然后添加如下代码:

```
manager = new QNetworkAccessManager(this);
ui ->progressBar ->hide();
```

这里开始将进度条隐藏了，这样在没有下载文件时是不显示进度条的。下面将前面程序中添加的 replyFinished() 函数的定义删除，然后添加新函数的定义。先添加网络请求函数的实现：

```
void MainWindow::startRequest(QUrl url)
{
    reply = manager ->get(QNetworkRequest(url));
    connect(reply, SIGNAL(readyRead()), this, SLOT(httpReadyRead()));

    connect(reply, SIGNAL(downloadProgress(qint64, qint64)), this, SLOT(updateDataReadProgress(qint64, qint64)));

    connect(reply, SIGNAL(finished()), this, SLOT(httpFinished()));
}
```

这里使用了 get() 函数来发送网络请求，然后进行了 QNetworkReply 对象的几个信号和自定义槽的关联。其中 readyRead() 信号继承自 QIODevice 类，每当有新的数据可以读取时，都会发射该信号；每当网络请求的下载进度更新时都会发射 downloadProgress() 信号，用来更新进度条；每当应答处理结束时，都会发射 finished() 信号，该信号与前面程序中 QNetworkAccessManager 类的 finished() 信号作用相同，只不过发送者不同，参数也不同而已。下面添加几个槽的定义。

```
void MainWindow::httpReadyRead()
{
    if (file) file ->write(reply ->readAll());
}
```

这里先判断是否创建了文件，如果是，则读取返回的所有数据，然后写入到文件。该文件是在后面的“下载”按钮单击信号槽中创建并打开的。

```
void MainWindow::updateDataReadProgress(qint64 bytesRead, qint64 totalBytes)
{
    ui ->progressBar ->setMaximum(totalBytes);
    ui ->progressBar ->setValue(bytesRead);
}
```

这里设置了一下进度条的最大值和当前值。

```
void MainWindow::httpFinished()
{
    ui ->progressBar ->hide();
    file ->flush();
    file ->close();
    reply ->deleteLater();
    reply = 0;
```

```
    delete file;
    file = 0;
}
```

当完成下载后,重新隐藏进度条,然后删除 reply 和 file 对象。下面到设计模式,进入“下载”按钮的单击信号的槽,然后添加如下代码:

```
void MainWindow::on_pushButton_clicked()
{
    url = ui->lineEdit->text();
    QFileinfo info(url.path());
    QString fileName(info.fileName());
    if (fileName.isEmpty()) fileName = "index.html";
    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly))
    {
        qDebug() << "file open error";
        delete file;
        file = 0;
        return;
    }
    startRequest(url);
    ui->progressBar->setValue(0);
    ui->progressBar->show();
}
```

这里使用要下载的文件名创建了本地文件,然后使用输入的 url 进行了网络请求,并显示进度条。现在可以运行程序了,可以输入一个网络文件地址然后单击“下载”按钮将其下载到本地,比如下载华军软件园上的劳拉方块游戏,可以使用如下 URL 地址: <http://zzidc.onlinedown.net:82/down/laolafangkuajin.rar>。对应这个例子,也可以参考 Qt 自带的 HTTP Example 示例程序,它的实现更加完善,还可以参考 Google Suggest Example 示例程序。

18.2 FTP

FTP(File Transfer Protocol,文件传输协议)是一个主要用于浏览远程目录和传输文件的协议。FTP 使用两个网络连接,一个用来发送命令,另一个用来输出数据。FTP 协议有一个状态,并且需要客户端在传输文件之前发送一些命令。FTP 客户端建立一个连接,并在整个会话期间一直保持打开。在每个会话期间,可以发生多个传输。

编写 FTP 应用,建议使用 QNetworkAccessManager 和 QNetworkReply,这些类提供了更加简单和强大的接口。不过,Qt 还提供了一个 QFtp 类,这个类对 FTP 提供了一个直接的接口,允许在网络请求上进行更多的控制。QFtp 类提供了一个支持

FTP的客户端，它有以下特点：

- 非阻塞行为。QFtp是异步的，可以安排一系列的命令，而这些命令等到控制权返回到Qt的事件循环后再执行。
- 命令ID。每一个命令都有一个唯一的ID号，可以使用它来跟随命令的执行过程。例如，当每一个命令被执行时QFtp都会使用该命令ID发射commandStarted()和commandFinished()信号。
- 数据传输的进度指示。当有数据传输时QFtp会发射信号(QFtp::dataTransferProgress()、QNetworkReply::downloadProgress()和QNetworkReply::uploadProgress())，可以关联这些信号到QProgressBar::setProgress()或者QProgressDialog::setProgress()上。
- QIODevice支持。该类支持对QIODevice进行方便的上传和下载操作。

这里主要有两种使用QFtp的方法。最一般的方式是保持跟踪命令的ID号，通过关联到合适的信号来跟踪每一条命令的执行；另一种方式是一次安排所有的命令，然后只关联done()信号，当所有被安排的命令都执行后发射该信号。第一种方式需要更多的编程工作，但是这样可以对每一个单独的命令拥有更多的控制，还可以在前一个命令结果的基础上执行一个新的命令，并且可以为用户提供详细的反馈。

在Qt中提供的FTP Example示例程序演示了怎样来编写一个FTP客户端，如果要编写自己的FTP（或者HTTP）服务器，需要使用更底层的类QTcpSocket和QTcpServer。下面通过例子来看一下QFtp类的具体使用。

（项目源码路径：src\18\18-3\myFTP）新建Qt Gui应用，名称为myFTP，类名为MainWindow，基类保持 QMainWindow 不变。完成后先在 myFTP.pro 文件中添加代码“QT += network”，然后保存该文件。进入设计模式，往界面上拖入一个Text Browser 和一个Label。下面到 mainwindow.h 文件中，添加类的前置声明：

```
class QFtp;
```

然后添加一个私有对象定义：

```
QFtp * ftp;
```

再添加私有槽的声明：

```
private slots:
    void ftpCommandStarted(int);
    void ftpCommandFinished(int, bool);
```

下面到 mainwindow.cpp 文件中，先添加头文件 #include <QFtp>，然后在构造函数中添加如下代码：

```
ftp = new QFtp(this);
ftp->connectToHost("ftp.qt.nokia.com");
ftp->login();
ftp->cd("qt");
```

```
ftp ->get("INSTALL");
ftp ->close();
connect(ftp, SIGNAL(commandStarted(int)), this, SLOT(ftpCommandStarted(int)));
connect(ftp, SIGNAL(commandFinished(int, bool)), this, SLOT(ftpCommandFinished(int, bool)));
```

QFtp 中提供了多个函数来完成常用的操作命令, 分别是: connectToHost() 使用指定的端口号连接到 FTP 服务器主机, 默认端口号是 21; login() 使用指定的用户名和密码登陆到 FTP 服务器; close() 关闭到 FTP 服务器的连接; list() 列出 FTP 服务器上指定的目录的内容, 默认列出当前目录的内容; cd() 改变服务器的工作目录到指定的目录; get() 从服务器下载指定的文件; put() 从 IO 设备中读取数据, 然后写入到服务器上指定的文件; remove() 从服务器上删除指定的文件; mkdir() 在服务器上创建指定的目录; rmdir() 从服务器上删除指定的目录; rename() 为服务器上的文件重命名; rawCommand() 发送原始的 FTP 命令到 FTP 服务器。这些命令函数都会返回一个唯一的 ID 号, 可以在后面使用这些 ID 号来区别不同的命令。这里我们登录到了 Qt 官网的 FTP 服务器, 然后跳转到了 qt 目录中, 下载了 INSTALL 文件。每当开始执行一个命令时, 都会发射 commandStarted() 信号, 当命令执行结束时, 会发射 commandFinished() 信号, 可以关联这两个信号来完成一些相关的操作。下面添加两个槽的定义:

```
void MainWindow::ftpCommandStarted(int)
{
    int id = ftp ->currentCommand();
    switch (id)
    {
        case QFtp::ConnectToHost :
            ui ->label ->setText(tr("正在连接到服务器…"));
            break;
        case QFtp::Login :
            ui ->label ->setText(tr("正在登录…"));
            break;
        case QFtp::Get :
            ui ->label ->setText(tr("正在下载…"));
            break;
        case QFtp::Close :
            ui ->label ->setText(tr("正在关闭连接…"));
    }
}

void MainWindow::ftpCommandFinished(int, bool error)
{
    if(ftp ->currentCommand() == QFtp::ConnectToHost) {
        if (error)
```

```
ui ->label ->setText(tr("连接服务器出现错误: %1").arg(ftp ->errorString
())));
else
    ui ->label ->setText(tr("连接到服务器成功"));
} else if (ftp ->currentCommand() == QFtp::Login) {
    if(error)
        ui ->label ->setText(tr("登录出现错误: %1").arg(ftp ->errorString
()));
    else
        ui ->label ->setText(tr("登录成功"));
} else if (ftp ->currentCommand() == QFtp::Get) {
    if(error)
        ui ->label ->setText(tr("下载出现错误: %1").arg(ftp ->errorString
()));
    else {
        ui ->label ->setText(tr("已经完成下载"));
        ui ->textBrowser ->setText(ftp ->readAll());
    }
} else if (ftp ->currentCommand() == QFtp::Close) {
    ui ->label ->setText(tr("已经关闭连接"));
}
}
```

这里可以使用信号传递过来的 int 参数(即当前命令的 ID)和前面执行命令时返回的 ID 进行比较来判断是哪一个命令执行了。其实也可以使用 currentCommand() 来获取当前正在执行的命令类型,然后进行判断,这样就可以不用保存以前命令的 ID 号了。currentCommand() 返回的是 QFtp::Command 枚举变量值,该枚举变量的取值对应了前面介绍的命令函数,可以在 QFtp 类的帮助文档中查看。在这两个函数中根据不同的命令,输出了一些不同的提示信息,在 QFtp::Get 命令结束时读取了所有获取的数据,并进行了显示。因为代码中使用了中文,还需要在 main.cpp 文件中添加相应代码。现在运行程序,效果如图 18-3 所示。

下面将前面的程序进行扩展，使其可以浏览并下载 FTP 服务器上所有的文件。（项目源码路径：src\18\18-4\myFTP）进入设计模式，删除 Text Browser，然后加入几个 Label、Line Edit、Push Button 部件。

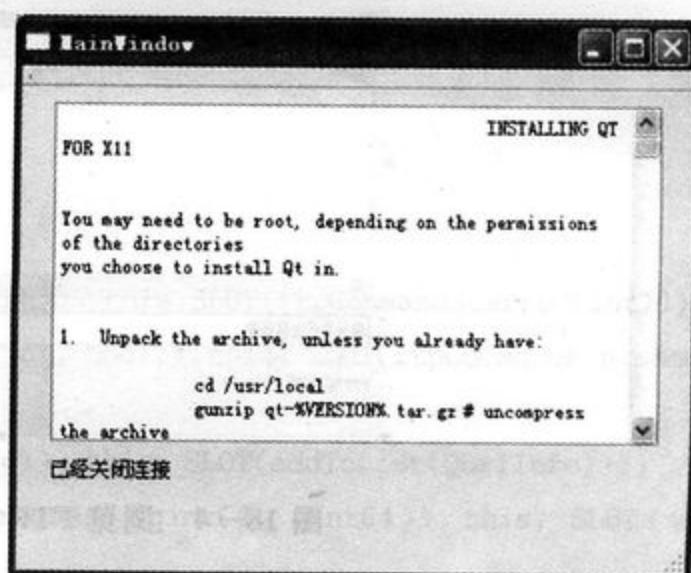


图 18-3 使用 FTP 下载文件运行效果

一个 Tree Widget 及一个 Progress Bar 部件，并对其中几个部件的属性做如下更改：

① 将“FTP 服务器”标签后的 Line Edit 的 objectName 属性改为 ftpServerLineEdit，其 text 属性改为“ftp.qt.nokia.com”。

② 将“用户名”标签后的 Line Edit 的 objectName 属性改为 userNameLineEdit，其 text 属性改为 anonymous，将其 toolTip 属性改为“默认用户名请使用：anonymous，此时密码任意”。

③ 将“密码”标签后的 Line Edit 的 objectName 属性改为 passWordLineEdit，其 text 属性改为 123456，将其 echoMode 属性选择为 Password。

④ 将“连接”按钮的 objectName 属性改为 connectButton。

⑤ 将“返回上一级目录”按钮的 objectName 属性改为 cdToParentButton。

⑥ 将“下载”按钮的 objectName 属性改为 downloadButton。

⑦ 将 Tree Widget 的 objectName 属性改为 fileList，然后在 Tree Widget 部件上右击，选择“编辑项目”菜单，添加“文件”、“大小”、“拥有者”、“所属组”和“修改日期”等列。

界面设计完成后，效果如图 18-4 所示。下面到 mainwindow.h 文件中，先添加类的前置声明和头文件包含：

```
#include <QHash>
class QFile;
class QUrlInfo;
class QTreeWidgetItem;
```

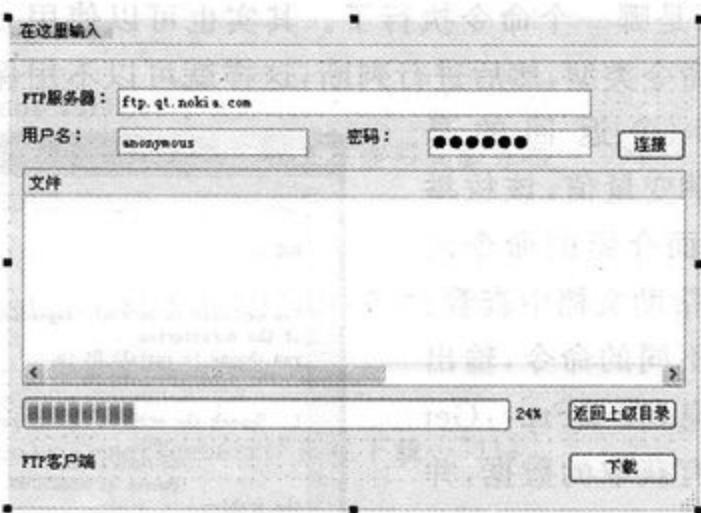


图 18-4 浏览 FTP 服务器界面设计效果

然后添加私有对象定义：

```
//用来存储一个路径是否为目录的信息
QHash<QString, bool> isDirectory;
```

```
//用来存储现在的路径
QString currentPath;
```

```
//用来表示下载的文件
```

```
QFile * file;
```

再添加几个私有槽声明：

```
//更新进度条
```

```
void updateDataTransferProgress(qint64, qint64);
```

```
//将服务器上的文件添加到 Tree Widget 部件中
```

```
void addToList(const QUrlInfo &urlInfo);
```

```
//双击一个目录时显示其内容
```

```
void processItem(QTreeWidgetItem * , int);
```

下面到 mainwindow.cpp 文件中，首先添加头文件包含：

```
# include <QFile>
```

```
# include <QTreeWidgetItem>
```

然后将前面程序中在构造函数中添加的代码删除，并添加如下代码：

```
ui ->progressBar ->setValue(0);
```

```
connect(ui ->fileList, SIGNAL(itemActivated(QTreeWidgetItem * , int)), this, SLOT(processItem(QTreeWidgetItem * , int)));
```

这里关联了列表部件的 itemActivated() 信号，当单击或者双击列表条目时，会发射该信号。下面先进入“连接”按钮的单击信号槽，将其更改如下：

```
void MainWindow::on_connectButton_clicked()
```

```
{
```

```
    ui ->fileList ->clear();
```

```
    currentPath.clear();
```

```
    isDirectory.clear();
```

```
    ftp = new QFtp(this);
```

```
    connect(ftp, SIGNAL(commandStarted(int)), this, SLOT(ftpCommandStarted(int)));
```

```
    connect(ftp, SIGNAL(commandFinished(int, bool)), this, SLOT(ftpCommandFinished(int, bool)));
```

```
    connect(ftp, SIGNAL(listInfo(QUrlInfo)), this, SLOT(addToList(QUrlInfo)));
```

```
    connect(ftp, SIGNAL(dataTransferProgress (qint64, qint64)), this, SLOT (updateDataTransferProgress(qint64, qint64)));
```

```
    QString ftpServer = ui ->ftpServerLineEdit ->text();
```

```
    QString userName = ui ->userNameLineEdit ->text();
```

```
    QString passWord = ui ->passWordLineEdit ->text();
```

```
    ftp ->connectToHost(ftpServer, 21);
```

```
    ftp ->login(userName, passWord);
```

```
}
```

这里主要是创建了 QFtp 对象，然后关联了几个信号和槽，最后根据提供的服务器地址和用户名密码登录。其中 listInfo() 信号在执行完 list() 命令时发射。下面将前面程序中的 ftpCommandFinished() 槽的内容更改如下：

```
void MainWindow::ftpCommandFinished(int, bool error)
{
    if(ftp->currentCommand() == QFtp::ConnectToHost) {
        if(error)
            ui->label->setText(tr("连接服务器出现错误: %1").arg(ftp->errorString
                ()));
        else ui->label->setText(tr("连接到服务器成功"));
    } else if(ftp->currentCommand() == QFtp::Login) {
        if(error)
            ui->label->setText(tr("登录出现错误: %1").arg(ftp->errorString
                ()));
        else {
            ui->label->setText(tr("登录成功"));
            ftp->list();
        }
    } else if(ftp->currentCommand() == QFtp::Get) {
        if(error)
            ui->label->setText(tr("下载出现错误: %1").arg(ftp->errorString
                ()));
        else {
            ui->label->setText(tr("已经完成下载"));
            file->close();
        }
        ui->downloadButton->setEnabled(true);
    } else if(ftp->currentCommand() == QFtp::List) {
        if(isDirectory.isEmpty())
        {
            ui->fileList->addTopLevelItem(
                new QTreeWidgetItem QStringList() << tr("<empty>"));
            ui->fileList->setEnabled(false);
            ui->label->setText(tr("该目录为空"));
        }
    } else if(ftp->currentCommand() == QFtp::Close) {
        ui->label->setText(tr("已经关闭连接"));
    }
}
```

这里在 QFtp::Login 命令结束时调用了 list() 函数来显示目录中的内容。然后添加了 QFtp::List 命令的判断，如果目录为空，则进行提示。在 QFtp::Get 命令结束

时,一定要注意关闭文件,不然下载文件会出现错误。下面添加 addToList()槽的定义:

```
void MainWindow::addToList(const QUrlInfo &urlInfo)
{
    QTreeWidgetItem * item = new QTreeWidgetItem;
    item->setText(0, urlInfo.name());
    item->setText(1, QString::number(urlInfo.size()));
    item->setText(2, urlInfo.owner());
    item->setText(3, urlInfo.group());
    item->setText(4, urlInfo.lastModified().toString("MMM dd yyyy"));
    QPixmap pixmap(urlInfo.isDir() ? "../myFTP/dir.png" : "../myFTP/file.png");
    item->setIcon(0, pixmap);
    isDirectory[urlInfo.name()] = urlInfo.isDir();
    ui->fileList->addTopLevelItem(item);
    if (!ui->fileList->currentItem()) {
        ui->fileList->setCurrentItem(ui->fileList->topLevelItem(0));
        ui->fileList->setEnabled(true);
    }
}
```

这里主要是将目录中的内容显示在列表部件中,并且使用了 isDirectory 容器存储了文件是否是目录的信息,这个会在 processItem()槽中用到,因为如果是目录就需要将其打开。这里还为目录和普通文件分别添加了图标,读者需要在项目目录中添加两张图片。

```
void MainWindow::processItem(QTreeWidgetItem * item, int)
{
    QString name = item->text(0);
    //如果这个文件是个目录,则打开
    if (isDirectory.value(name)) {
        ui->fileList->clear();
        isDirectory.clear();
        currentPath += "/";
        currentPath += name;
        ftp->cd(name);
        ftp->list();
        ui->cdToParentButton->setEnabled(true);
    }
}
```

这里在双击列表中的目录时,重新调用 list()函数来显示新的目录的内容。下面添加“返回上级目录”按钮的单击信号槽的定义:

```
void MainWindow::on_cdToParentButton_clicked()
```

```
{  
    ui ->fileList ->clear();  
    isDirectory.clear();  
    currentPath = currentPath.left(currentPath.lastIndexOf('/));  
    if (currentPath.isEmpty()) {  
        ui ->cdToParentButton ->setEnabled(false);  
        ftp ->cd("/");  
    } else {  
        ftp ->cd(currentPath);  
    }  
    ftp ->list();  
}
```

这里对现在的路径进行处理，然后重新调用 list() 函数来显示上级目录的内容。下面添加“下载”按钮的单击信号槽的定义：

```
void MainWindow::on_downloadButton_clicked()  
{  
    QString fileName = ui ->fileList ->currentItem() ->text(0);  
    file = new QFile(fileName);  
    if (!file ->open(QIODevice::WriteOnly)) {  
        delete file;  
        return;  
    }  
    ui ->downloadButton ->setEnabled(false);  
    ftp ->get(ui ->fileList ->currentItem() ->text(0), file);  
}
```

这里根据要下载的文件的名称创建了本地文件，然后使用 get() 函数进行文件的下载。最后添加更新进度条槽的实现：

```
void MainWindow::updateDataTransferProgress(qint64 readBytes, qint64 totalBytes)  
{  
    ui ->progressBar ->setMaximum(totalBytes);  
    ui ->progressBar ->setValue(readBytes);  
}
```

现在可以运行程序了，先单击“连接”按钮，等待登录成功，显示出文件列表以后，就可以下载任意的文件了。对应这个例子，可以参考 Qt 的 FTP Example 示例程序，该程序的功能更加完善。

18.3 获取网络接口信息

在进行 TCP/UDP 编程时，需要先将连接的主机名解析为 IP 地址，这个操作一般

使用 DNS(Domain Name Service, 域名服务)协议执行。IP(Internet Protocol, 互联网协议)是计算机网络相互连接进行通信时使用的协议, 规定了计算机在互联网上进行通信时应当遵循的规则, 有 IPV4 和 IPV6 两个版本。而 IP 地址就是给每一个连接在互联网上的主机分配的一个唯一的地址, IP 协议使用这个地址来进行主机之间的信息传递。

QtNetwork 模块中的 QHostInfo 类提供了静态函数可以进行主机名的查找, 它使用了操作系统提供的查找机制来获取与一个主机名关联的 IP 地址, 或者获取与一个 IP 地址关联的主机名。这个类提供了两个便捷的静态函数: lookupHost() 异步进行工作, 每当找到主机时都会发射信号; fromName() 会阻塞并返回一个 QHostInfo 对象。

(项目源码路径: src\18\18-5\myIP)新建 Qt Gui 应用, 项目名称为 myIP, 类名为 MainWindow, 基类选择 QMainWindow。完成后先在 myIP.pro 文件中添加“QT+=network”一行代码, 并保存该文件。然后进入 mainwindow.cpp 文件, 添加头文件包含:

```
#include <QtNetwork>
#include <QDebug>
```

然后在构造函数中添加如下代码:

```
QString localHostName = QHostInfo::localHostName();
QHostInfo info = QHostInfo::fromName(localHostName);
qDebug() << "localHostName: " << localHostName << endl
      << "IP Address: " << info.addresses();
```

这里先使用 localHostName() 静态函数获取了本地主机名称, 也就是自己的计算机名称, 这个在桌面的“我的电脑”图标上右击查看其属性, 然后在计算机名页面可以看到。然后使用 fromName() 函数, 根据主机名获取了 QHostInfo 对象, 使用 QHostInfo 类的 addresses() 函数可以获取与主机名相关的 IP 地址的列表, 它返回的是一个 QHostAddress 对象的列表。QHostAddress 类代表了一个 IP 地址。从一个主机名获取的 IP 地址可能不止一个, 不过其中第一个一般是本机设定的 IP 地址, 这个可以在网上邻居的属性中查看本地连接的属性, 然后在“Internet 协议(TCP/IP)”一项中手动指定。现在可以运行程序, 笔者这里出现了两个 IP 地址: 192.168.1.10 和 121.27.241.210, 前者就是笔者自己指定的 IP 地址, 而后者是进行 ADSL 上网拨号时动态生成的。在有些系统上还可能出现 IPv4 和 IPv6 两种地址, 要获取其中的 IPv4 地址, 可以对 IP 地址列表进行遍历, 下面继续添加代码:

```
foreach(QHostAddress address, info.addresses())
{
    if(address.protocol() == QAbstractSocket::IPv4Protocol)
        qDebug() << address.toString();
}
```

除了使用 `fromName()` 来获取 IP 地址, 还可以使用 `lookupHost()` 函数, 它需要指定一个主机名, 一个 `QObject` 指针和一个槽。该函数可以执行名称查找, 当完成后会调用指定的 `QObject` 对象的槽, 查找工作是在其他线程中进行的, 即异步进行的。

下面先到 `mainwindow.h` 文件中添加类的前置声明:

```
class QHostInfo;
```

然后添加一个私有槽声明:

```
private slots:  
    void lookedUp(const QHostInfo &host);
```

下面到 `mainwindow.cpp` 文件中添加该槽的定义:

```
void MainWindow::lookedUp(const QHostInfo &host)
```

```
{  
    if (host.error() != QHostInfo::NoError) {  
        qDebug() << "Lookup failed:" << host.errorString();  
        return;  
    }  
    foreach (const QHostAddress &address, host.addresses())  
        qDebug() << "Found address:" << address.toString();  
}
```

这里先判断是否出错, 如果发生错误就输出错误信息并返回。如果没有问题, 则遍历返回的 IP 列表并输出。下面在构造函数中添加如下一行代码:

```
QHostInfo::lookupHost("www.baidu.com", this, SLOT(lookedUp(QHostInfo)));
```

这里对百度网站的 IP 地址进行了查找, 查找完成后便会调用自定义的 `lookedUp()` 槽。现在可以运行程序, 查看一下输出的结果。`lookupHost()` 函数返回一个整型 ID 值, 可以调用 `abortHostLookup()` 函数, 通过这个 ID 值来终止查找。另外, 还可以将这个函数中的第一个参数设置为一个 IP 地址来查找该 IP 地址对应的域名。

在网络模块中还提供了 `QNetworkInterface` 类来获取主机的 IP 地址列表和网络接口信息。`QNetworkInterface` 类代表了运行当前程序的主机的网络接口。下面通过代码来看一下该类的使用。继续在构造函数中添加如下代码:

```
// 获取所有网络接口的列表  
QList<QNetworkInterface> list = QNetworkInterface::allInterfaces();  
  
// 遍历每一个网络接口  
foreach (QNetworkInterface interface, list)  
{  
    // 接口名称  
    qDebug() << "Name: " << interface.name();
```

```

//硬件地址
qDebug() << "HardwareAddress: " << interface.hardwareAddress();

//获取 IP 地址条目列表,每个条目包含一个 IP 地址,一个子网掩码和一个广播地址
QList<QNetworkAddressEntry> entryList = interface.addressEntries();

//遍历每一个 IP 地址条目
foreach (QNetworkAddressEntry entry, entryList)
{
    //IP 地址
    qDebug() << "IP Address: " << entry.ip().toString();

    //子网掩码
    qDebug() << "Netmask: " << entry.netmask().toString();

    //广播地址
    qDebug() << "Broadcast: " << entry.broadcast().toString();
}

```

这里先使用 allInterfaces() 函数获取主机上所有网络接口的列表,然后对这个列表进行了遍历。使用 name() 函数可以获取网络接口的名称,在 Unix 系统上,这个字符串包含了接口的类型和可选的序列号,例如“eth0”、“lo”或者“pcn0”;在 Windows 系统上,它是一个不能被用户改变的内部编号。使用 hardwareAddress() 函数可以获取底层的硬件地址。函数 addressEntries() 可以返回一个 QNetworkAddressEntry 对象的列表,QNetworkAddressEntry 类保存了一个网络接口支持的 IP 地址,以及与该 IP 地址相关的子网掩码和广播地址。如果只需用获取 IP 地址,那么可以使用 QNetworkInterface 类提供的 allAddresses() 函数,它返回了所有 IP 地址的列表。现在可以运行程序,查看输出的结果。

18.4 UDP

UDP(User Datagram Protocol, 用户数据报协议)是一个轻量级的、不可靠的、面向数据报的、无连接的协议,用于可靠性不是非常重要的情况下,例如,一个服务器报告一天的时间可以选择 UDP。如果一个包含一天的时间的数据报丢失了,那么客户端可以简单地发送另外一个请求。UDP一般分为发送端和接收端,如图 18-5 所示。

QUDpSocket 类用来发送和接收 UDP 数据报,继承自 QAbstractSocket。这里的 Socket 就是所谓的“套接字”,简单来说,套接字就是一个 IP 地址加一个 port 端口号。其中 IP 地址指定了网络中的一台主机,而端口号指定了该主机上的一个网络程序,这样使用套接字就可以实现网络上两台主机上的两个应用程序之间的通信。

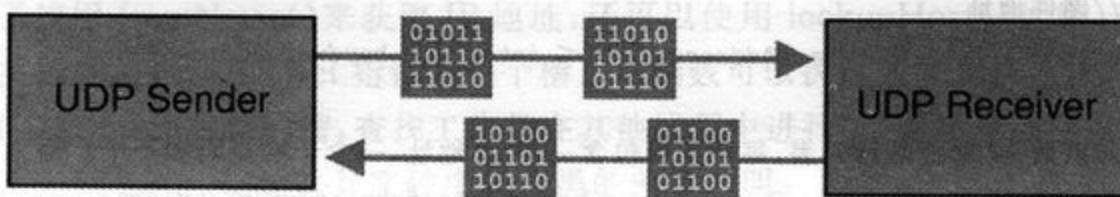


图 18-5 UDP 数据传输示意图

QUDpSocket 支持 IPv4 广播。广播一般用来实现网络发现协议,例如,查找网络上哪个主机拥有最多的硬盘空间。一个主机向网络中广播一个数据报,然后所有其他的主机都接收这个数据报。每一个主机接收到一个请求,然后发送应答给发送端,告知其当前的可用磁盘空间。发送端一直等待,直到它接收到所有主机的答复,然后可以选择拥有最多空闲空间的服务器来存储数据。要广播一个数据报,只需简单发送它到一个特殊的地址 QHostAddress::Broadcast(即 255.255.255.255),或者是本地网络的广播地址。下面通过一个例子来讲解一下怎样进行 UDP 编程。

先编写发送端程序。(项目源码路径: src\18\18-6\udpSender)新建 Qt Gui 应用,项目名称为 udpSender,类名为 Sender,基类选择 QDialog,完成后向 udpSender.pro 文件中添加“QT+=network”一行代码,并保存该文件。然后到 sender.h 文件中,添加类的前置声明:

```
class QUdpSocket;
```

再添加一个私有对象定义:

```
QUdpSocket * sender;
```

下面到 sender.ui 文件中,向界面上拖入一个 Push Button 按钮部件,将其显示文本更改为“进行广播”。然后转到其单击信号槽,更改如下:

```
void Sender::on_pushButton_clicked()
{
    QByteArray datagram = "hello world!";
    sender->writeDatagram(datagram.data(), datagram.size(), QHostAddress::Broadcast,
                           45454);
}
```

这里调用了 writeDatagram() 函数来发送数据报,该函数的原型为:

```
qint64 QUdpSocket::writeDatagram (const char * data, qint64 size, const QHostAddress &
address, quint16 port)
```

它会发送 size 大小的数据报 data 到地址为 address 的主机的 port 端口,并返回成功发送的字节数,如果发送失败则返回 -1。数据报总是作为一整块写入的,它的最大大小根据平台的不同而不同。如果数据报过大,这个函数将会返回 -1,而且 error() 函数会返回 DatagramTooLargeError 错误信息。一般不建议发送大于 512 字节的数据报,即便被发送成功,它们很可能是在到达最终目的地以前,在 IP 层被分割了。这里使

用了枚举变量 QHostAddress::Broadcast 来表示广播地址, 它等价于 QHostAddress (“255.255.255.255”)。对于端口号, 它是可以随意指定的, 但是一般建议使用 1 024 以上的端口号, 因为 1 024 以下的端口号通常用于保留端口号(例如 FTP 使用的 21 端口), 端口号最大为 65 535。需要注意的是, 这里使用端口号 45454, 那么在接收端也要使用这个端口号。

下面在 sender.cpp 文件中添加头文件 #include <QtNetwork>, 然后在构造函数中添加如下一行代码:

```
sender = new QUdpSocket(this);
```

现在可以先运行程序。下面来添加接收端程序。

(项目源码路径: src\18\18-6\udpReceiver)新建 Qt Gui 应用, 项目名称为 udpReceiver, 类名为 Receiver, 基类选择 QDialog, 完成后向 udpReceiver.pro 文件中添加 “QT+=network”一行代码, 然后保存该文件。然后到 receiver.h 文件中, 添加类的前置声明:

```
class QUdpSocket;
```

再添加一个私有对象定义:

```
QUdpSocket * receiver;
```

然后添加一个私有槽声明:

```
private slots:  
    void processPendingDatagram();
```

下面到设计模式, 向界面上拖入一个 Label, 将其显示文本更改为“等待接收数据!”。然后进入 receiver.cpp 文件中, 添加头文件 #include <QtNetwork>, 然后在构造函数中添加如下代码:

```
receiver = new QUdpSocket(this);  
receiver ->bind(45454, QUdpSocket::ShareAddress);  
connect(receiver, SIGNAL(readyRead()), this, SLOT(processPendingDatagram()));
```

这里先使用 bind() 函数来绑定了 IP 地址和端口号, 程序中使用的 bind() 函数的一个重载形式, 它不需要指定 IP 地址, 默认支持所有的 IPv4 的 IP 地址; 其中指定的端口号就是前面发送端使用的端口号; 最后的一个参数是绑定模式, 程序中使用 QUdpSocket::ShareAddress 表明允许其他的服务器绑定到相同的地址和端口上。每当有数据报到来时, QUdpSocket 都会发射 readyRead() 信号, 这样就可以在自定义的槽中读取数据了。下面添加该槽函数的实现代码:

```
void Receiver::processPendingDatagram()  
{  
    // 拥有等待的数据报  
    while(receiver ->hasPendingDatagrams())
```

```

    {
        QByteArray datagram;

        //让 datagram 的大小为等待处理的数据报的大小,这样才能接收到完整的数据
        datagram.resize(receiver->pendingDatagramSize());

        //接收数据报,将其存放到 datagram 中
        receiver->readDatagram(datagram.data(), datagram.size());
        ui->label->setText(datagram);
    }
}

```

这里使用了 `hasPendingDatagrams()` 来判断是否还有等待读取的数据报,如果有,就将其内容读取到自定义的变量中,然后显示出来。可以使用 `pendingDatagramSize()` 来获取当前数据报的大小,然后使用 `readDatagram()` 函数接收不大于指定大小的数据报,并将其存储到 `QByteArray` 变量中。

下面先运行 `udpReceiver` 程序,在 Windows 系统中可能出现“Windows 安全警报”对话框,选择“解除阻止”即可。然后运行 `udpSender` 程序,并单击“进行广播”按钮,现在在 `udpReceiver` 界面上就可以显示出接收到的数据报信息了。关于 UDP 编程,还可以参考 Qt 自带的 Broadcast Sender 和 Broadcast Receiver 示例程序。

18.5 TCP

TCP(Transmission Control Protocol, 传输控制协议)是一个用于数据传输的低层网络协议,多个互联网协议(包括 HTTP 和 FTP)都是基于 TCP 协议的。TCP 是一个面向数据流和连接的可靠传输协议。QTcpSocket 类为 TCP 提供了一个接口,继承自 QAbstractSocket。可以使用 QTcpSocket 来实现 POP3、SMTP 和 NNTP 等标准的网络协议,也可以实现自定义的网络协议。与 QUDpSocket 传输的数据报不同,QTcpSocket 传输的是连续的数据流,尤其适合于连续的数据传输。TCP 编程一般分为客户端和服务器端,也就是所谓的 C/S(Client/Server)模型,如图 18-6 所示。

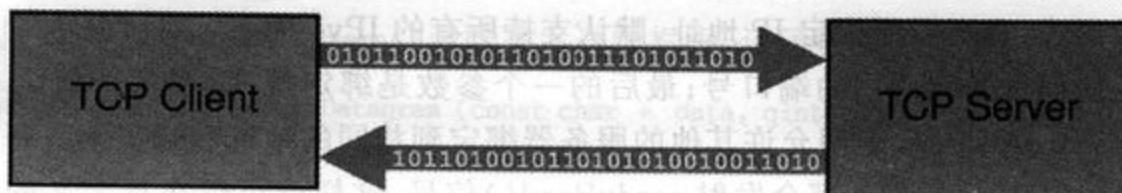


图 18-6 TCP 数据传输示意图

在任何数据传输之前,必须建立一个 TCP 连接到远程的主机和端口上。一旦连接被建立,peer(对使用 TCP 协议连接在一起的主机的通称)的 IP 地址和端口可以分别使用 `QTcpSocket::peerAddress()` 和 `QTcpSocket::peerPort()` 来获取。在任何时间,

peer 都可以关闭连接,这样数据传输就会立即停止。QTcpSocket 异步工作的,通过发射信号来报告状态改变和错误信息,就像前面介绍的 QNetworkAccessManager 和 QFtp 一样。它依靠事件循环来检测到来的数据,并且自动刷新输出的数据。可以使用 QTcpSocket::write() 来写入数据,使用 QTcpSocket::read() 来读取数据。QTcpSocket 代表了两个独立的数据流:一个用来读取,另一个用来写入。因为 QTcpSocket 继承自 QIODevice,所以可以使用 QTextStream 和 QDataStream。从一个 QTcpSocket 中读取数据前,必须先调用 QTcpSocket::bytesAvailable() 函数来确保已经有足够的数据可用。

如果要处理到来的 TCP 连接(例如在一个服务器应用程序中),可以使用 QTcpServer 类。调用 QTcpServer::listen() 来设置服务器,然后关联 QTcpServer::newConnection() 信号,每当有客户端连接时都会发射该信号。在槽中,调用 QTcpServer::nextPendingConnection() 来接收这个连接,然后使用该函数返回的 QTcpSocket 对象与客户端进行通信。

尽管 QTcpSocket 中的大多数函数都是异步工作的,其实也可以使用 QTcpSocket 来实现同步工作(例如,阻塞)。要实现阻塞行为,可以调用 QTcpSocket 的以 waitFor 开头的函数,它们会挂起调用的线程,直到一个信号被发射。例如,在调用了 QTcpSocket::connectToHost() 非阻塞函数后,调用 QTcpSocket::waitForConnected() 来阻塞线程,直到 connected() 信号被发射。使用同步函数,编写代码会更简单,最主要缺点是在 waitFor 函数阻塞时事件将不再被处理,如果在一个 GUI 线程中,那么用户界面可能会被冻结。所以,一般建议只在非 GUI 线程中才使用同步套接字。

为了让读者更加容易地入门 TCP 编程,这里先讲解一个简单的例子,其实现的功能是:服务器一直监听一个端口,一旦有客户端连接请求,便建立连接,并向客户端发送一个字符串,然后客户端接收该字符串并显示出来。

先编写服务器端程序。(项目源码路径: src\18\18-7\tcpServer)新建 Qt Gui 应用,项目名称为 tcpServer,类名为 Server,基类选择 QDialog,完成后向 tcpServer.pro 文件中添加“QT+=network”一行代码,并保存该文件。然后进入 server.ui 文件中,向界面上拖入一个 Label 部件,并更改其显示文本为“等待连接”。下面进入 server.h 文件,先添加类的前置声明:

```
class QTcpServer;
```

然后添加私有对象定义:

```
QTcpServer *tcpServer;
```

最后添加一个私有槽声明:

```
private slots:  
    void sendMessage();
```

下面转到 server.cpp 文件中,先添加头文件 #include <QtNetwork>,然后在构

造函数中添加如下代码：

```
tcpServer = new QTcpServer(this);
//使用了 IPv4 的本地主机地址,等价于 QHostAddress("127.0.0.1")
if (! tcpServer ->listen(QHostAddress::LocalHost, 6666)) {
    qDebug() << tcpServer ->errorString();
    close();
}
connect(tcpServer, SIGNAL(newConnection()), this, SLOT(sendMessage()));
```

这里调用了 QTcpServer 类的 listen() 函数来监听到来的连接,这里监听了本地主机的 6666 端口,这样可以实现客户端和服务器端在同一台计算机上运行并通信,也可以换成其他地址。一旦有客户端连接到服务器就会发射 newConnection() 信号。下面添加发送信息槽的实现代码:

```
void Server::sendMessage()
{
    //用于暂存要发送的数据
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);

    //设置数据流的版本,客户端和服务器端使用的版本要相同
    out.setVersion(QDataStream::Qt_4_0);
    out << (quint16)0;
    out << tr("hello TCP!!!");
    out.device() ->seek(0);
    out << (quint16)(block.size() - sizeof(quint16));

    //获取已经建立的连接的套接字
    QTcpSocket * clientConnection = tcpServer ->nextPendingConnection();
    connect(clientConnection, SIGNAL(disconnected()),
            clientConnection, SLOT(deleteLater()));
    clientConnection ->write(block);
    clientConnection ->disconnectFromHost();

    //发送数据成功后,显示提示
    ui ->label ->setText("send message successful!!!");
}
```

这里使用了 QByteArray 对象来暂存要发送的数据,然后使用数据流将要发送的数据写入 QByteArray 对象中。这里需要设置数据流的版本,而且客户端在接收数据时要使用相同的版本。为了在接收数据时可以接收到完整的数据,发送数据时一定要在最开始写入实际数据的大小信息,该大小信息占用两个字节,可以使用(quint16)0 来表示。因为在写入数据以前,我们可能不知道实际数据的大小,所以要先在数据块的最

前端空留两个字节的位置,以便以后填写数据的大小信息,这就是 `out << (quint16)0` 的作用。然后输入实际的数据,这里就是一个字符串,注意字符串要使用 `tr()` 进行编码,否则在接收端可能无法正常显示。输入完实际的数据以后,使用 `(quint16)(block.size() - sizeof(quint16))` 即数据块总大小减去数据块开头两个字节的大小,就可以获得实际数据的大小了。这时使用 `seek(0)` 跳转到数据块的开头,然后将获取的大小信息填到前面空留的两个字节处。然后使用了 `nextPendingConnection()` 函数获取了连接的套接字,然后使用 `write()` 函数将 `block` 数据发送出去。这里还关联了套接字的 `disconnected()` 信号和 `deleteLater()` 槽,表明当连接断开时删除该套接字。而调用的 `disconnectFromHost()` 函数会一直等待套接字将所有数据发送完毕,然后关闭该套接字,并发射 `disconnected()` 信号。可以先运行程序。

可以看到,编写 TCP 服务器端程序,可以创建 `QTcpServer` 类,然后调用它的 `listen()` 函数来进行监听,要对连接过来的客户端进行操作,可以通过关联 `newConnection()` 信号,在槽中进行。可以使用 `nextPendingConnection()` 函数来获取连接的套接字。

下面编写客户端程序。(项目源码路径: `src\18\18-7\tcpClient`)新建 Qt Gui 应用,项目名称为 `tcpClient`,类名为 `Client`,基类选择 `QDialog`,完成后向 `tcpClient.pro` 文件中添加“`QT += network`”一行代码,并保存该文件。然后进入 `client.ui` 文件中,往界面上拖入 3 个 `Label`,两个 `Line Edit` 和一个 `Push Button`,如图 18-7 所示。将“主机”标签后的 `Line Edit` 的 `objectName` 更改为 `hostLineEdit`;“端口”标签后的 `Line Edit` 的 `objectName` 更改为 `portLineEdit`;将“接收到的信息”标签的 `objectName` 更改为 `messageLabel`;将“连接”按钮的 `objectName` 更改为 `connectButton`。

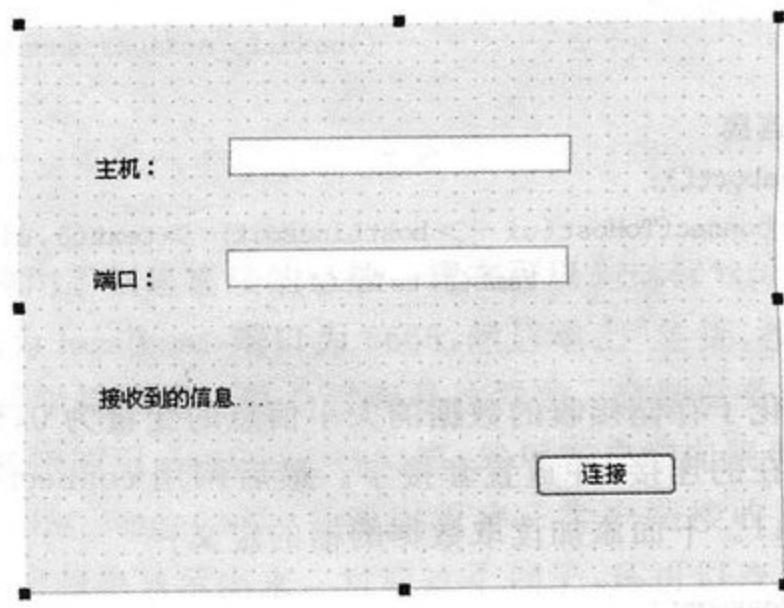


图 18-7 TCP 客户端界面设计效果

进入 `client.h` 文件中,先添加头文件和类的前置声明:

```
#include <QAbstractSocket>
class QTcpSocket;
```

然后添加私有对象和变量的定义：

```
QTcpSocket *tcpSocket;
QString message;
//用来存放数据的大小信息
quint16 blockSize;
```

再添加几个私有槽的声明：

```
private slots:
void newConnect();
void readMessage();
void displayError(QAbstractSocket::SocketError);
```

下面转到 client.cpp 文件中，先添加头文件 `#include <QtNetwork>`，然后在构造函数中添加如下代码：

```
tcpSocket = new QTcpSocket(this);
connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(readMessage()));
connect (tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(displayError(QAbstractSocket::SocketError)));
```

这里关联了两个信号到自定义的槽上，当有可读的数据时会发射 `readyRead()` 信号，当发生错误时会发射 `error()` 信号。下面添加 `newConnect()` 槽的定义：

```
void Client::newConnect()
{
    //初始化数据大小信息为 0
    blockSize = 0;

    //取消已有的连接
    tcpSocket ->abort();
    tcpSocket ->connectToHost(ui ->hostLineEdit ->text(), ui ->portLineEdit ->text().toInt());
}
```

这个槽中先初始化了存储接收的数据的大小信息的变量为 0，然后使用 `abort()` 函数取消了当前已经存在的连接，并重置套接字。最后调用 `connectToHost()` 函数连接到指定主机的指定端口。下面添加读取数据的槽的定义：

```
void Client::readMessage()
{
    QDataStream in(tcpSocket);
    //设置数据流版本，这里要和服务器端相同
    in.setVersion(QDataStream::Qt_4_6);

    //如果是刚开始接收数据
```

```

if (blockSize == 0) {
    //判断接收的数据是否大于两字节,也就是文件的大小信息所占的空间
    //如果是则保存到 blockSize 变量中,否则直接返回,继续接收数据
    if(tcpSocket->bytesAvailable() < (int)sizeof(quint16)) return;
    in >> blockSize;
}
//如果没有得到全部的数据,则返回,继续接收数据
if(tcpSocket->bytesAvailable() < blockSize) return;
//将接收到的数据存放到变量中
in >> message;
//显示接收到的数据
qDebug() << message;
ui->messageLabel->setText(message);
}

```

在读取数据时,先读取了数据的大小信息,然后使用该大小信息来判断是否已经读取到了所有的数据。下面添加显示错误信息的槽的定义:

```

void Client::displayError(QAbstractSocket::SocketError)
{
    qDebug() << tcpSocket->errorString();
}

```

这里只是简单输出了错误信息。最后从设计模式进入“连接”按钮的单击信号对应的槽,更改如下:

```

void Client::on_connectButton_clicked()
{
    newConnect();
}

```

这里只是简单调用了创建连接的函数。现在可以先运行 `tcpServer` 程序,然后再运行该程序,输入主机为 `localhost`,端口为 6666,然后单击“连接”按钮,这时就可以显示从服务器发送回来的字符串了。整个过程是这样的:启动服务器端,开始监听端口。客户端单击“连接”按钮调用 `newConnect()` 槽,然后客户端向服务器发送连接请求,这时服务器便调用 `sendMessage()` 槽,发送字符串。客户端接收到字符串时,调用 `readMessage()` 槽来将字符串显示出来。对应这个例子,还可以查看 Qt 提供的 `Fortune Client` 和 `Fortune Server` 示例程序,它们演示了怎样使用 `QTcpSocket` 和 `QTcpServer` 来编写 TCP 客户端—服务器应用程序。还有 `Blocking Fortune Client` 示例程序演示了怎样在一个独立的线程中(没有使用事件循环)使用同步 `QTcpSocket`; `Threaded Fortune Server` 示例程序演示了多线程 TCP 服务器,其中每一个客户端都使用一个单独的线程。

下面再来介绍一个稍微复杂点的例子,它实现了大型文件的传输,而且还可以显示

传输进度。这个例子也是由客户端和服务器端组成的,这次在客户端进行数据的发送,在服务器端进行数据的接收。通过这个例子应该进一步掌握对发送和接收数据的处理,可以灵活地在客户端或者服务器端进行数据的发送或者接收。对应这个例子,也可以参考一下 Qt 自带的 loopback Example 示例程序。

先编写客户端程序。(项目源码路径: src\18\18-8\tcpClient)新建 Qt Gui 应用,项目名称为 tcpClient,类名为 Client,基类选择 QDialog,完成后向 tcpClient.pro 文件中添加“QT += network”一行代码,并保存该文件。然后进入 client.ui 文件中,往界面上拖入 3 个 Label、两个 Line Edit、两个 Push Button 和一个 Progress Bar。然后将设置这些部件的属性,将“主机”后的 Line Edit 的 objectName 改为 hostLineEdit;“端口”后的 Line Edit 的 objectName 改为 portLineEdit;Progress Bar 的 objectName 改为 clientProgressBar,其 value 属性设为 0;“状态”Label 的 objectName 为 clientStatusLabel;“打开”按钮的 objectName 为 openButton;“发送”按钮的 objectName 为 sendButton,最终效果如图 18-8 所示。

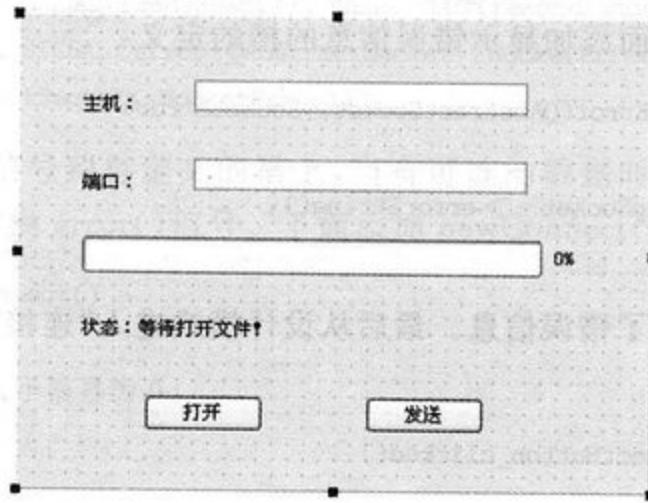


图 18-8 TCP 客户端界面设计效果

下面进入 client.h 文件中,添加头文件和类的前置声明:

```
#include <QAbstractSocket>
class QTcpSocket;
class QFile;
```

再添加几个私有对象和变量的定义:

```
QTcpSocket *tcpClient;
QFile *localFile;           //要发送的文件
qint64 totalBytes;          //发送数据的总大小
qint64 bytesWritten;        //已经发送数据大小
qint64 bytesToWrite;        //剩余数据大小
qint64 payloadSize;         //每次发送数据的大小
QString fileName;           //保存文件路径
QByteArray outBlock;         //数据缓冲区,即存放每次要发送的数据块
```

然后添加几个私有槽声明：

```
private slots:
    void openFile();
    void send();
    void startTransfer();
    void updateClientProgress(qint64);
    void displayError(QAbstractSocket::SocketError);
```

再转到 client.cpp 文件中添加头文件包含：

```
#include <QtNetwork>
#include <QFileDialog>
```

然后到构造函数中添加如下代码：

```
payloadSize = 64 * 1024; //64KB
totalBytes = 0;
bytesWritten = 0;
bytesToWrite = 0;
tcpClient = new QTcpSocket(this);

//当连接服务器成功时,发出 connected()信号,开始传送文件
connect(tcpClient, SIGNAL(connected()), this, SLOT(startTransfer()));
connect(tcpClient, SIGNAL(bytesWritten(qint64)),
        this, SLOT(updateClientProgress(qint64)));
connect(tcpClient, SIGNAL(error(QAbstractSocket::SocketError)),
        this, SLOT(displayError(QAbstractSocket::SocketError)));
ui ->sendButton ->setEnabled(false);
```

这里只是对变量进行了初始化,然后关联了几个信号和槽。下面添加打开文件槽的定义：

```
void Client::openFile()
{
    fileName = QFileDialog::getOpenFileName(this);
    if (!fileName.isEmpty()) {
        ui ->sendButton ->setEnabled(true);
        ui ->clientStatusLabel ->setText(tr("打开文件 %1 成功!").arg(fileName));
    }
}
```

这里使用 QFileDialog 来打开一个本地的文件。下面添加连接到服务器槽的定义：

```
void Client::send()
{
```

```
ui ->sendButton ->setEnabled(false);

//初始化已发送字节为 0
bytesWritten = 0;
ui ->clientStatusLabel ->setText(tr("连接中..."));
tcpClient ->connectToHost(ui ->hostLineEdit ->text(),
                           ui ->portLineEdit ->text().toInt());
}
```

这里调用了 `connectToHost()` 来连接到服务器。下面来实现文件的传输，在实际的文件传输以前，需要将整个传输的数据的大小、文件名的大小还有文件名等信息放在数据的开头进行传输，这里可以把它们统称为文件头结构。发送文件头结构的方法与前面一个例子中发送数据大小信息是相似的，只不过这里使用了 `qint64` 类型，它可以表示更大的数据。在 `startTransfer()` 槽中来发送文件头结构，下面来添加它的定义：

```
void Client::startTransfer()
{
    localFile = new QFile(fileName);
    if (! localFile ->open(QFile::ReadOnly)) {
        qDebug() << "client: open file error!";
        return;
    }
    //获取文件大小
    totalBytes = localFile ->size();
    QDataStream sendOut(&outBlock, QIODevice::WriteOnly);
    sendOut.setVersion(QDataStream::Qt_4_0);
    QString currentFileName = fileName.right(fileName.size() -
                                              fileName.lastIndexOf("/") - 1);
    //保留总大小信息空间、文件名大小信息空间，然后输入文件名
    sendOut << qint64(0) << qint64(0) << currentFileName;

    //这里的总大小是总大小信息、文件名大小信息、文件名和实际文件大小的总和
    totalBytes += outBlock.size();
    sendOut.device() ->seek(0);

    //返回 outBlock 的开始，用实际的大小信息代替两个 qint64(0) 空间
    sendOut << totalBytes << qint64((outBlock.size() - sizeof(qint64) * 2));

    //发送完文件头结构后剩余数据的大小
    bytesToWrite = totalBytes - tcpClient ->write(outBlock);

    ui ->clientStatusLabel ->setText(tr("已连接"));
}
```

```

    outBlock.resize(0);
}
}

```

这里总大小信息 totalBytes 是要发送的整个数据的大小,它包括了文件头结构和实际文件的大小。这个 totalBytes 要放在数据流的最开始,占用第一个 qint64(0)的空间,然后是文件名的大小,它放在 totalBytes 之后,占用第二个 qint64(0)的空间,再往后是文件名。而 outBlock 是暂存数据的,最后要将其清零,即 resize(0)。

发送完文件头结构以后,就应该发送实际的文件了,这是在更新进度条槽中实现的,下面添加它的定义:

```

void Client::updateClientProgress(qint64 numBytes)
{
    //已经发送数据的大小
    bytesWritten += (int)numBytes;

    //如果已经发送了数据
    if (bytesToWrite > 0) {
        //每次发送 payloadSize 大小的数据,这里设置为 64 KB,如果剩余的数据不足 64 KB
        //就发送剩余数据的大小
        outBlock = localFile ->read(qMin(bytesToWrite, payloadSize));

        //发送完一次数据后还剩余数据的大小
        bytesToWrite -= (int)tcpClient ->write(outBlock);

        //清空发送缓冲区
        outBlock.resize(0);
    } else { //如果没有发送任何数据,则关闭文件
        localFile ->close();
    }
    //更新进度条
    ui ->clientProgressBar ->setMaximum(totalBytes);
    ui ->clientProgressBar ->setValue(bytesWritten);
    //如果发送完毕
    if (bytesWritten == totalBytes) {
        ui ->clientStatusLabel ->setText(tr("传送文件 %1 成功").arg(fileName));
        localFile ->close();
        tcpClient ->close();
    }
}

```

数据是分为数据块进行发送的,每次发送的数据块的大小为 payloadSize 指定的大小,这里为 64 KB。如果剩余的数据不足 64 KB,则发送剩余的数据,这就是 qMin() 函数的作用。每当有数据发送时就更新进度条,如果数据发送完毕了,便关闭本地文件和

客户端套接字。下面添加显示错误槽的定义：

```
void Client::displayError(QAbstractSocket::SocketError)
{
    qDebug() << tcpClient->errorString();
    tcpClient->close();
    ui->clientProgressBar->reset();
    ui->clientStatusLabel->setText(tr("客户端就绪"));
    ui->sendButton->setEnabled(true);
}
```

这里输出了错误信息，然后进行了一些重置工作。下面进入设计模式，然后分别进入“打开”按钮和“发送”按钮的单击信号对应的槽，更改如下：

```
//打开按钮
void Client::on_openButton_clicked()
{
    ui->clientProgressBar->reset();
    ui->clientStatusLabel->setText(tr("状态：等待打开文件！"));
    openFile();
}

//发送按钮
void Client::on_sendButton_clicked()
{
    send();
}
```

这里只是调用了相应的槽，每次打开新的文件时都清空进度条。因为程序中使用了中文，所以还要在 main.cpp 文件中添加相应的代码。现在可以运行一下程序了。

下面再来添加服务器端程序。（项目源码路径：src\18\18-8\tcpServer）新建 Qt Gui 应用，项目名称为 tcpServer，类名为 Server，基类选择 QDialog，完成后向 tcpServer.pro 文件中添加“QT+=network”一行代码，并保存该文件。然后进入 server.ui 文件中，往界面上拖入一个 Label、一个 Push Button 和一个 Progress Bar。将“服务器端”Label 的 objectName 改为 serverStatusLabel；进度条 Progress Bar 的 objectName 改为 serverProgressBar，设置其 value 属性为 0；“开始监听”按钮的 objectName 改为 startButton。最终效果如图 18-9 所示。下面进入 server.h 文件中，先添加头文件和类的前置声明：

```
#include <QAbstractSocket>
#include <QTcpServer>
class QTcpSocket;
class QFile;
```

然后添加几个私有对象和变量的定义：

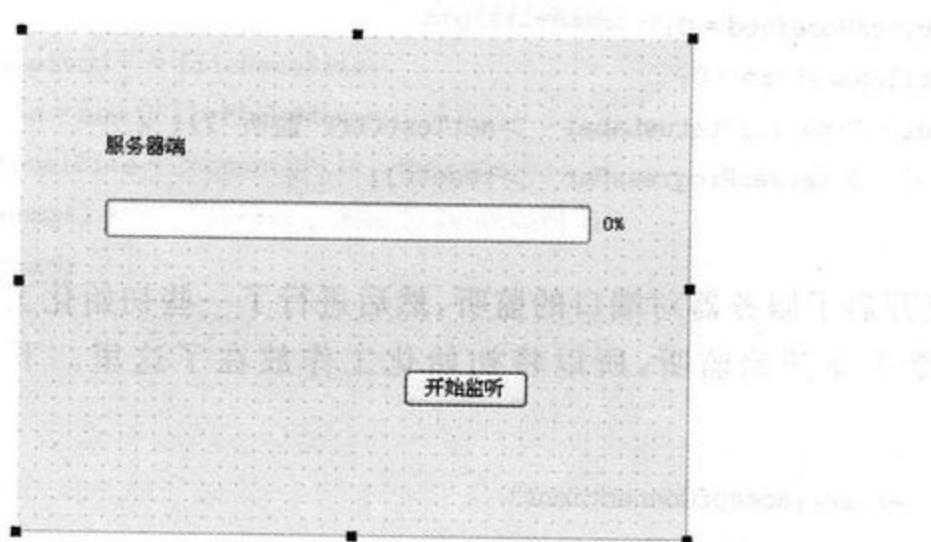


图 18-9 TCP 服务器端界面设计效果

```
QTcpServer tcpServer;
QTcpSocket *tcpServerConnection;
qint64 totalBytes;           //存放总大小信息
qint64 bytesReceived;       //已收到数据的大小
qint64 fileNameSize;        //文件名的大小信息
QString fileName;           //存放文件名
QFile *localFile;           //本地文件
QByteArray inBlock;          //数据缓冲区
```

再添加几个私有槽声明：

```
private slots:
    void start();
    void acceptConnection();
    void updateServerProgress();
    void displayError(QAbstractSocket::SocketError socketError);
```

再转到 server.cpp 文件中，先添加头文件包含 `#include <QtNetwork>`，然后在构造函数中添加如下代码：

```
connect(&tcpServer, SIGNAL(newConnection()), this, SLOT(acceptConnection()));
```

这里只是进行了信号和槽的关联。下面先来添加开启监听槽的实现：

```
void Server::start()
{
    if (!tcpServer.listen(QHostAddress::LocalHost, 6666)) {
        qDebug() << tcpServer.errorString();
        close();
        return;
    }
    ui->startButton->setEnabled(false);
    totalBytes = 0;
```

```
bytesReceived = 0;
fileNameSize = 0;
ui ->serverStatusLabel ->setText(tr("监听"));
ui ->serverProgressBar ->reset();
}
```

这里开启了服务器对端口的监听，然后进行了一些初始化工作，因为每次接收完文件后，都要重新开始监听，所以将初始化工作放在了这里。下面添加接收连接槽的实现：

```
void Server::acceptConnection()
{
    tcpServerConnection = tcpServer.nextPendingConnection();
    connect(tcpServerConnection, SIGNAL(readyRead()),
            this, SLOT(updateServerProgress()));
    connect(tcpServerConnection, SIGNAL(error(QAbstractSocket::SocketError)), this,
            SLOT(displayError(QAbstractSocket::SocketError)));
    ui ->serverStatusLabel ->setText(tr("接受连接"));
    //关闭服务器，不再进行监听
    tcpServer.close();
}
```

这里接收到了来的连接请求，并获取其套接字，然后进行了信号和槽的关联。注意在最后使用 close() 函数关闭了服务器，不再监听端口。下面添加更新进度条槽的实现，文件的实际接收工作也在这个槽中进行：

```
void Server::updateServerProgress()
{
    QDataStream in(tcpServerConnection);
    in.setVersion(QDataStream::Qt_4_0);

    //如果接收到的数据小于 16 个字节，保存到来的文件头结构
    if (bytesReceived <= sizeof(qint64) * 2) {
        if((tcpServerConnection->bytesAvailable() >= sizeof(qint64) * 2)
           && (fileNameSize == 0)) {
            //接收数据总大小信息和文件名大小信息
            in >> totalBytes >> fileNameSize;
            bytesReceived += sizeof(qint64) * 2;
        }
        if((tcpServerConnection->bytesAvailable() >= fileNameSize)
           && (fileNameSize != 0)) {
            //接收文件名，并建立文件
            in >> fileName;
            ui ->serverStatusLabel ->setText(tr("接收文件 %1 ..."))
        }
    }
}
```

```

        .arg(fileName));
bytesReceived += fileNameSize;
localFile = new QFile(fileName);
if (! localFile ->open(QIODevice::WriteOnly)) {
    qDebug() << "server: open file error!";
    return;
}
} else {
    return;
}
}

//如果接收的数据小于总数据,那么写入文件
if (bytesReceived < totalBytes) {
    bytesReceived += tcpServerConnection ->bytesAvailable();
    inBlock = tcpServerConnection ->readAll();
    localFile ->write(inBlock);
    inBlock.resize(0);
}
ui ->serverProgressBar ->setMaximum(totalBytes);
ui ->serverProgressBar ->setValue(bytesReceived);

//接收数据完成时
if (bytesReceived == totalBytes) {
    tcpServerConnection ->close();
    localFile ->close();
    ui ->startButton ->setEnabled(true);
    ui ->serverStatusLabel ->setText(tr("接收文件 %1 成功!").arg(fileName));
}
}
}

```

这里先分别接收了数据总大小、文件名大小以及文件名等文件头结构的信息，然后才接收实际的文件。后面更新了进度条。如果接收完成，那么关闭套接字和本地文件，并进行提示。下面添加显示错误槽的实现：

```

void Server::displayError(QAbstractSocket::SocketError socketError)
{
    qDebug() << tcpServerConnection ->errorString();
    tcpServerConnection ->close();
    ui ->serverProgressBar ->reset();
    ui ->serverStatusLabel ->setText(tr("服务端就绪"));
    ui ->startButton ->setEnabled(true);
}

```

这里显示了错误信息，然后进行了一些设置工作。下面进入“开始监听”按钮的单

击信号槽中,更改如下:

```
void Server::on_startButton_clicked()
{
    start();
}
```

这里只是简单调用了 start()槽。因为程序中使用了中文,所以还要在 main.cpp 文件中添加相应的代码。现在可以先运行 tcpServer 程序,单击“开始监听”按钮,然后再运行 tcpClient 程序,输入主机 localhost,端口 6666,并打开一个文件,最后单击“发送”按钮来发送文件。整个过程是这样的:服务器端单击“开始监听”按钮,调用 start()开始监听端口。然后客户端填写完主机、端口,选择好文件,单击“发送”按钮则调用 send()连接到服务器。服务器一旦发现有连接请求,就会调用 acceptConnection()获得连接的套接字,等待获取数据。而连接一旦建立,客户端套接字便发射 connected()信号,从而调用 startTransfer()来发送文件头结构。这时服务器端发现有数据到来便更新进度条,也就是调用 updateServerProgress(),在其中获取了发送过来的数据。而在客户端,当有数据发送时,也会调用 updateClientProgress()来更新进度条,在其中将后面的数据发送出去。

18.6 小结

本章讲述了利用 Qt 进行网络相关编程的基本内容,涉及了常用的 HTTP、FTP、UDP 和 TCP 等通信协议,并对每个协议都通过例子进行了简单的编程演示。Qt 的网络模块是一个很庞大的体系,还有很多内容在本章中并没用涉及,例如,网络代理 QNetworkProxy 类、承载管理 QNetworkConfigurationManager 类、QNetworkSession 类以及涉及通信安全的 QSsl 相关类等,有需要的可以参考相关帮助文档。

在《Qt 及 Qt Quick 开发实战精解》中的局域网聊天工具实例程序综合应用了 UDP 和 TCP 等相关内容,学习完本章可以接着学习该实例程序。

第 19 章

进程和线程

Qt 提供了对进程和线程的支持。本章讲述了怎样在 Qt 应用程序中启动一个进程,以及几种常用的进程间通信方法。线程部分讲到了怎样编写一个多线程 Qt 应用程序,还涉及了线程同步等内容。本章最后详细讲解了 Qt 文档中“可重入”和“线程安全”两个术语的相关内容。

19.1 进 程

设计应用程序时,有时不希望将一个不太相关的功能集成到程序中,或者是因为该功能与当前设计的应用程序联系不大,或者是因为该功能已经可以使用现成的程序很好地实现了,这时就可以在当前的应用程序中调用外部的程序来实现该功能,这就会使用到进程。Qt 应用程序可以很容易地启动一个外部应用程序,而且 Qt 也提供了多种进程间通信的方法。

19.1.1 运行一个进程

Qt 的 QProcess 类用来启动一个外部程序并与其进行通信。要启动一个进程,可以使用 start() 函数,然后将程序名称和运行这个程序所要使用的命令行参数作为该函数的参数。执行完 start() 函数后,QProcess 进入 Starting 状态,程序运行后 QProcess 就会进入 Running 状态并发射 started() 信号。进程退出后,QProcess 重新进入 NotRunning 状态(初始状态)并发射 finished() 信号。发射的 finished() 信号提供了进程的退出代码和退出状态,也可以调用 exitCode() 来获取上一个结束进程的退出代码,使用 exitStatus() 来获取它的退出状态。任何时间发生了错误,QProcess 都会发射 error() 信号,也可以调用 error() 来查看错误的类型和上次发生的错误。使用 state() 可以查看当前进程的状态。

QProcess 允许将一个进程视为一个顺序 I/O 设备。可以像使用 QTcpSocket 访问一个网络连接一样来读/写一个进程。可以调用 write() 向进程的标准输入进行写

入,调用 read()、readLine() 和 getChar() 等从标准输出进行读取。因为 QProcess 继承自 QIODevice,它也可以作为 QXmlReader 的数据源,或者为 QFtp 产生用于上传的数据。

下面来看一个启动进程的例子。(项目源码路径: src\19\19-1\myProcess)新建 Qt Gui 应用,名称为 myProcess,类名为 MainWindow,基类保持 QMainWindow 不变。完成后进入设计模式,往界面上拖入一个 Push Button,并将其显示文本更改为“启动一个进程”。然后进入 mainwindow.h 文件,先添加头文件包含:

```
# include <QProcess>
```

然后添加一个私有对象定义:

```
QProcess myProcess;
```

下面从设计模式进入“启动一个进程”按钮的单击信号槽中,更改如下:

```
void MainWindow::on_pushButton_clicked()
{
    myProcess.start("notepad.exe");
}
```

这里启动了 Windows 系统的记事本程序(即 notepad.exe,因为它在 Windows 的系统目录下,该目录已经加在系统 PATH 环境变量中,所以不需要写具体路径)。现在可以运行程序,然后单击按钮,这时会启动一个记事本。可以看到,使用 QProcess 运行一个程序是很简单的。下面使用参数来运行 Windows 的 cmd.exe 命令行提示符程序,并且使用 dir 命令作为其参数,然后读取执行完的结果。这里还对 QProcess 的一些信号进行关联,显示相关信息。

到 mainwindow.h 文件中添加私有槽声明:

```
void showResult();
void showState(QProcess::ProcessState);
void showError();
void showFinished(int, QProcess::ExitStatus);
```

然后到 mainwindow.cpp 文件中添加头文件包含 #include <QDebug>,再在构造函数中添加信号和槽的关联:

```
connect(&myProcess, SIGNAL(readyRead()), this, SLOT(showResult()));
connect(&myProcess, SIGNAL(stateChanged(QProcess::ProcessState)), this, SLOT(showState
    (QProcess::ProcessState)));
connect(&myProcess, SIGNAL(error(QProcess::ProcessError)), this, SLOT(showError()));
connect(&myProcess, SIGNAL(finished(int,QProcess::ExitStatus)), this, SLOT(showFinished
    (int, QProcess::ExitStatus)));
```

下面先更改“启动一个进程”按钮的单击信号槽中的代码:

```
void MainWindow::on_pushButton_clicked()
{
    QString program = "cmd.exe";
    QStringList arguments;
    arguments << "/c dir&pause";
    myProcess.start(program, arguments);
}
```

这里为 start() 函数指定了程序名称和命令行参数。命令行参数使用了“/c dir&pause”，这里的“/c”指定命令 dir 在 cmd 中执行，“&pause”指定运行完命令后暂停。下面添加显示运行结果的槽的定义：

```
void MainWindow::showResult()
{
    qDebug() << "showResult: " << endl << QString(myProcess.readAll());
```

这里使用了 readAll() 函数来读取所有的运行结果，使用 QString() 是为了进行编码转换，这样才可以正常显示结果中的中文字符，在后面会看到，我们在 main.cpp 文件中添加了相关的代码来支持编码转换。下面添加显示状态变化槽的定义：

```
void MainWindow::showState(QProcess::ProcessState state)
{
    qDebug() << "showState: ";
    if (state == QProcess::NotRunning) {
        qDebug() << "Not Running";
    } else if (state == QProcess::Starting) {
        qDebug() << "Starting";
    } else {
        qDebug() << "Running";
    }
}
```

这里只是根据不同的状态进行了不同的提示输出。下面添加显示错误槽的定义：

```
void MainWindow::showError()
{
    qDebug() << "showError: " << endl << myProcess.errorString();
```

这里使用了 errorString() 来获取错误信息，并将其输出。下面添加显示结束信息槽的定义：

```
void MainWindow::showFinished(int exitCode, QProcess::ExitStatus exitStatus)
{
    qDebug() << "showFinished: " << endl << exitCode << exitStatus;
```

}

这里输出了退出代码和退出状态。下面到 main.cpp 文件中，添加头文件：

```
#include <QTextCodec>
```

然后在主函数中添加如下一行代码：

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
```

这样就可以在程序中使用 QString() 来进行字符串的编码转换了。现在可以运行程序，然后查看一下输出结果信息。

QProcess 也提供了一组函数，可以脱离事件循环来使用，它们会挂起调用的线程直到确定的信号被发射：

- waitForStarted() 阻塞直到进程已经启动；
- waitForReadyRead() 阻塞直到在当前读通道上有可读的数据；
- waitForBytesWritten() 阻塞直到一个有效负载数据已经被写入到进程；
- waitForFinished() 阻塞直到进程已经结束。

在主线程(调用 QApplication::exec() 的线程)中调用这些函数可能引起用户界面的冻结。下面的代码片段演示了在没有事件循环的情况下运行 gzip 来压缩字符串“Qt rocks!”：

```
QProcess gzip;
gzip.start("gzip", QStringList() << "-c");
if (! gzip.waitForStarted())
    return false;
gzip.write("Qt rocks!");
gzip.closeWriteChannel();
if (! gzip.waitForFinished())
    return false;
QByteArray result = gzip.readAll();
```

19.1.2 进程间通信

Qt 提供了多种方法在 Qt 应用程序中实现进程间通信 IPC(Inter-Process Communication)。简单介绍如下：

(1) TCP/IP

跨平台的 QtNetwork 模块提供了众多的类来实现网络编程。它提供了高层的类(例如 QNetworkAccessManager、QFtp 等)来使用指定的应用程序级协议，也提供了较低层的类(例如 QTcpSocket、QTcpServer 和 QSslSocket)来实现相关协议。

(2) 共享内存

QSharedMemory 是跨平台的共享内存类，提供了访问操作系统共享内存的实现。它允许多个线程和进程安全的访问共享内存段。此外，QSystemSemaphore 可用于控

制系统的共享资源的访问以及进程间通信。

(3) D-Bus

QtDBus 模块是一个 Unix 库, 可以使用 D-Bus 协议来实现进程间通信。它将 Qt 的信号和槽机制扩展到了 IPC 层面, 允许从一个进程发射的信号关联到另一个进程的槽上。可以在帮助中查看 D-Bus 关键字, 在对应的文档中有其详细的介绍。

(4) Qt 通信协议 (QCOP)

QCopChannel 类实现了在客户端程序间使用有名管道来进行消息传输的协议。QCopChannel 仅在嵌入式 Linux(可以在帮助中查看 Qt for Embedded Linux 关键字)编程中可用。与 QtDBus 模块相似, QCOP 将 Qt 的信号和槽机制扩展到了 IPC 层面, 允许从一个进程发射的信号关联到另一个进程的槽上。而与 QtDBus 不同的是, QCOP 并不依赖于第三方库。

Qt 中还有一个 QLocalSocket 类提供了一个本地套接字, 在 Windows 上它是一个有名管道, 在 Unix 上它是一个本地域套接字。与其对应的是 QLocalServer 类, 它提供了一个基于本地套接字的服务器。这两个类的使用与前一章的 QTcpSocket/QTcpServer 很相似, 这里就不再过多介绍。可以参考一下这两个类的帮助文档, 也可以参考一下 Qt 提供的 Local Fortune Client Example 和 Local Fortune Server Example 示例程序。关于 Qt 进程间通信的内容, 可以参考 Qt 帮助中 Inter-Process Communication in Qt 关键字对应的文档。下面来看一个使用共享内存的例子, 它实现的功能是: 先将一张图片写入到共享内存段中, 然后再从共享内存段读出该图片。

(项目源码路径: src\19\19-2\myIPC) 新建 Qt Gui 应用, 名称为 myIPC, 类名为 Dialog, 基类选择 QDialog。完成后进入设计模式, 向界面中放入两个 Push Button 部件和一个 Label 部件。将一个按钮的显示文本更改为“从文件中加载图片”, 将其 objectName 属性更改为 loadFromFileButton, 将另一个按钮的显示文本更改为“从共享内存显示图片”, 将其 objectName 属性更改为 loadFromSharedMemoryButton。然后进入 dialog.h 文件, 先添加头文件包含:

```
#include <QSharedMemory>
```

然后添加两个公有槽声明:

```
public slots:
    void loadFromFile();
    void loadFromMemory();
```

再添加一个私有函数声明:

```
private:
    void detach();
```

最后添加一个私有对象定义:

```
QSharedMemory sharedMemory;
```

下面到 dialog.cpp 文件中,先添加头文件:

```
# include <QFileDialog>
# include <QBuffer>
# include <QDebug>
```

然后在构造函数中添加如下一行代码:

```
sharedMemory.setKey("QSharedMemoryExample");
```

在使用共享内存以前,需要先为其指定一个 key,系统用它来作为底层共享内存段的标识。这个 key 可以指定为任意的字符串。下面添加从文件加载图片槽的定义:

```
void Dialog::loadFromFile()
{
    if (sharedMemory.isAttached())
        detach();
    ui->label->setText(tr("选择一个图片文件!"));
    QString fileName = QFileDialog::getOpenFileName(0, QString(), QString(), tr("Images
(*.png *.jpg)"));
    QImage image;
    if (!image.load(fileName)) {
        ui->label->setText(tr("选择的文件不是图片,请选择图片文件!"));
        return;
    }
    ui->label->setPixmap(QPixmap::fromImage(image));
    //将图片加载到共享内存
    QBuffer buffer;
    buffer.open(QBuffer::ReadWrite);
    QDataStream out(&buffer);
    out << image;
    int size = buffer.size();
    if (!sharedMemory.create(size)) {
        ui->label->setText(tr("无法创建共享内存段!"));
        return;
    }
    sharedMemory.lock();
    char * to = (char *)sharedMemory.data();
    const char * from = buffer.data().data();
    memcpy(to, from, qMin(sharedMemory.size(), size));
    sharedMemory.unlock();
}
```

这里先使用 isAttached() 函数判断该进程是否已经连接到共享内存段,如果是,那么就调用 detach() 先将该进程与共享内存段进行分离。然后使用 QFileDialog 类来打

开一个图片文件，并将其显示到标签上。为了将图片加载到共享内存，这里使用了 QBuffer 来暂存图片，这样便可以获得图片的大小，还可以获得图片数据的指针。后面使用了 create() 函数来创建指定大小的共享内存段，其大小的单位是字节，该函数还会自动将共享内存段连接到本进程上。创建完共享内存段以后，使用了 memcpy() 函数将 buffer 对应的数据段复制到共享内存段。在进行共享内存段的操作时，需要先进行加锁，即调用 lock() 函数。等操作完成后，调用 unlock() 来进行解锁。这样在同一时间，就只能有一个进程允许操作共享内存段了。需要说明，如果将最后一个连接在共享内存段上的进程进行分离，那么系统便会释放该共享内存段。因为现在只有一个进程连接在共享内存段上，所以不能将它们进行分离。下面添加从内存中显示图片槽的实现代码：

```
void Dialog::loadFromMemory()
{
    if (! sharedMemory.attach()) {
        ui->label->setText(tr("无法连接到共享内存段,\n"
                               "请先加载一张图片!"));
        return;
    }

    QBuffer buffer;
    QDataStream in(&buffer);
    QImage image;

    sharedMemory.lock();
    buffer.setData((char *)sharedMemory.constData(), sharedMemory.size());
    buffer.open(QBuffer::ReadOnly);
    in >> image;
    sharedMemory.unlock();

    sharedMemory.detach();
    ui->label->setPixmap(QPixmap::fromImage(image));
}
```

这里先使用 attach() 函数将进程连接到共享内存段，然后使用 QBuffer 来读取内存段中的数据，完成后将其传输到 QImage 对象中供下面进行显示。在操作共享内存段时要使用 lock() 进行加锁，操作完成后使用 unlock() 进行解锁。因为现在已经不需要使用共享内存了，所以调用 detach() 函数将进程与共享内存段进行分离。最后将图片显示到标签中。下面添加分离函数的实现代码：

```
void Dialog::detach()
{
    if (! sharedMemory.detach())
        ui->label->setText(tr("无法从共享内存中分离!"));
```

```
}
```

这里就是调用了 QSharedMemory 的 detach() 函数,如果失败则进行提示。下面进入设计模式,分别进入两个按钮的单击信号对应的槽中,修改如下:

```
void Dialog::on_loadFromFileButton_clicked()
{
    loadFromFile();
}

void Dialog::on_loadFromSharedMemoryButton_clicked()
{
    loadFromMemory();
}
```

这里就是调用了相应的槽。因为在代码中使用了中文,所以还需要在 main.cpp 文件中添加相应的代码。现在运行两次程序,在一个运行的实例上单击“从文件中加载图片”按钮,然后选择一张图片。在第二个运行的实例上单击“从共享内存显示图片”按钮,这时便会显示第一个实例中加载的图片,效果如图 19-1 所示。

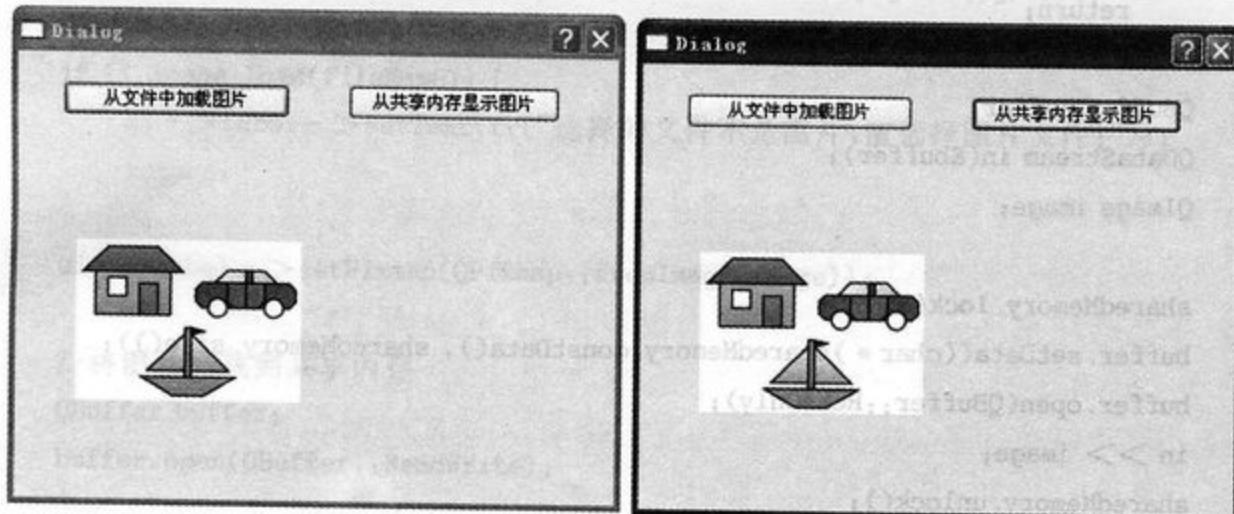


图 19-1 共享内存运行效果

19.2 线程

Qt 提供了对线程的支持,这包括一组与平台无关的线程类、一个线程安全的发送事件的方式,以及跨线程的信号-槽的关联。这些使得可以很容易地开发可移植的多线程 Qt 应用程序,可以充分利用多处理器的机器。多线程编程也可以有效解决在不冻结一个应用程序的用户界面情况下执行一个耗时的操作问题。对应本节的内容,可以在帮助中查看 Thread Support in Qt 关键字。

19.2.1 使用 QThread 启动线程

Qt 中的 QThread 类提供了平台无关的线程。一个 QThread 代表了一个在应用程

序中可以独立控制的线程,它与进程中的其他线程分享数据,但是是独立执行的。相对于一般的程序都是从 main() 函数开始执行,QThread 从 run() 函数开始执行。默认的,run() 通过调用 exec() 来开启事件循环。要创建一个线程,需要子类化 QThread 并且重新实现 run() 函数。例如:

```
class MyThread : public QThread
{
protected:
    void run();
};

void MyThread::run()
{
    QTcpSocket socket;
    ...
    socket.connectToHost(hostName, portNumber);
    exec();
}
```

这样会在一个线程中创建一个 QTcpSocket,然后执行这个线程的事件循环。可以在外部创建该线程的实例,然后调用 start() 函数来开始执行该线程,start() 默认会调用 run() 函数。当从 run() 函数返回后,线程便执行结束,就像应用程序离开 main() 函数一样。QThread 会在开始、结束和终止时发射 started()、finished() 和 terminated() 等信号。也可以使用 isFinished() 和 isRunning() 来查询线程的状态。可以使用 wait() 来阻塞直到线程结束执行。每个线程都会从操作系统获得自己的堆栈,操作系统会决定堆栈的默认大小,也可以使用 setStackSize() 来设置一个自定义的堆栈大小。

每一个线程可以有自己的事件循环,可以通过调用 exec() 函数来启动事件循环,可以通过调用 exit() 或者 quit() 来停止事件循环。在一个线程中拥有一个事件循环,可以使它能够关联其他线程中的信号到本线程的槽上,这使用了队列关联机制,就是在使用 connect() 函数进行信号和槽的关联时,将 Qt::ConnectionType 类型的参数指定为 Qt::QueuedConnection。拥有事件循环还可以使该线程能够使用需要事件循环的类,比如 QTimer 和 QTcpSocket 类等。注意,在线程中是无法使用任何的部件类的。

在极端情况下,可能想要强制终止一个正在执行的线程,这时可以使用 terminate() 函数。但是,线程是否会被立即终止依赖于操作系统的调度策略。可以在调用完 terminate() 后调用 QThread::wait() 来同步终止。使用 terminate() 函数,线程可能在任何时候被终止而无法进行一些清理工作,因此该函数是很危险的,一般不建议使用,只有在绝对必要的时候使用。

静态函数 currentThreadId() 和 currentThread() 可以返回当前执行的线程的标识符,前者返回一个该线程的系统特定的 ID;后者返回一个 QThread 指针。QThread 也提供了多个平台无关的睡眠函数,其中 sleep() 精度为秒,msleep() 精度为毫秒,usleep

(1) 精度为微秒。

下面来看一个在图形界面程序中启动一个线程的例子，在界面上有两个按钮，一个用于开启一个线程，一个用于关闭该线程。（项目源码路径：src\19\19-3\myThread）新建 Qt Gui 应用，名称为 myThread，类名为 Dialog，基类选择 QDialog。完成后进入设计模式，向界面中放入两个 Push Button 按钮，将第一个按钮的显示文本更改为“启动线程”，将其 objectName 属性更改为 startButton；将第二个按钮的显示文本更改为“终止线程”，将其 objectName 属性更改为 stopButton，将其 enabled 属性取消选中。然后向项目中添加新的 C++ 类，类名设置为“MyThread”，基类设置为“QThread”，类型信息选择“继承自 QObject”。完成后进入 mythread.h 文件，先添加一个公有函数声明：

```
void stop();
```

然后再添加一个函数声明和一个变量的定义：

```
protected:  
    void run();  
private:  
    volatile bool stopped;
```

这里 stopped 变量使用了 volatile 关键字，这样可以使它在任何时候都保持最新的值，从而可以避免在多个线程中访问它时出错。然后进入 mythread.cpp 文件中，先添加头文件 #include <QDebug>，然后在构造函数中添加如下代码：

```
stopped = false;
```

这里将 stopped 变量初始化为 false。下面添加 run() 函数的定义：

```
void MyThread::run()  
{  
    qreal i = 0;  
    while (! stopped)  
        qDebug() << QString("in MyThread: %1").arg(i++);  
    stopped = false;  
}
```

这里一直判断 stopped 变量的值，只要它为 false，那么就一直打印字符串。下面添加 stop() 函数的定义：

```
void MyThread::stop()  
{  
    stopped = true;  
}
```

stop() 函数中将 stopped 变量设置为了 true，这样便可以结束 run() 函数中的循环，从而从 run() 函数中退出，这样整个线程也就结束了。这里使用了 stopped 变量来

实现了进程的终止，并没有使用危险的 terminate() 函数。

下面在 Dialog 类中使用自定义的线程。先到 dialog.h 文件中，添加头文件包含：

```
#include "mythread.h"
```

然后添加私有对象的定义：

```
MyThread thread;
```

下面到设计模式，分别进入两个按钮的单击信号对应的槽，更改如下：

```
//启动线程按钮
```

```
void Dialog::on_startButton_clicked()
```

```
{
```

```
    thread.start();
```

```
    ui->startButton->setEnabled(false);
```

```
    ui->stopButton->setEnabled(true);
```

```
}
```

```
//终止线程按钮
```

```
void Dialog::on_stopButton_clicked()
```

```
{
```

```
    if (thread.isRunning()) {
```

```
        thread.stop();
```

```
        ui->startButton->setEnabled(true);
```

```
        ui->stopButton->setEnabled(false);
```

```
}
```

```
}
```

在启动线程时调用了 start() 函数，然后设置了两个按钮的状态。在终止线程时，先使用 isRunning() 来判断线程是否在运行，如果是，则调用 stop() 函数来终止线程，并且更改两个按钮的状态。现在运行程序，单击“启动线程”按钮，查看应用程序输出栏的输出，然后再按下“终止线程”按钮，可以看到已经停止输出了。

19.2.2 同步线程

Qt 中的 QMutex、QReadWriteLock、QSemaphore 和 QWaitCondition 类提供了同步线程的方法。虽然使用线程的思想是多个线程可以尽可能地并发执行，但是总有一些时刻、一些线程必须停止来等待其他线程。例如，如果两个线程尝试同时访问相同的全局变量，结果通常是不确定的。

QMutex 提供了一个互斥锁(mutex)，在任何时间至多有一个线程可以获得 mutex。如果一个线程尝试获得 mutex，而此时 mutex 已经被锁住了，这个线程将会睡眠，直到现在获得 mutex 的线程对 mutex 进行解锁为止。互斥锁经常用于对共享数据(例如，可以同时被多个线程访问的数据)的访问进行保护。

QReadWriteLock 即读一写锁,与 QMutex 很相似,只不过它将对共享数据的访问区分为“读”访问和“写”访问,允许多个线程同时对数据进行“读”访问。在可能的情况下使用 QReadWriteLock 代替 QMutex,可以提高多线程程序的并发度。

QSemaphore 即信号量,是 QMutex 的一般化,用来保护一定数量的相同的资源,而互斥锁 mutex 只能保护一个资源。在本节后面的部分将讲述一个使用信号量实现的经典的生产者—消费者的例子。

QWaitCondition 即条件变量,允许一个线程在一些条件满足时唤醒其他的线程。一个或者多个线程可以被阻塞来等待一个 QWaitCondition 来设置一个用于 wakeOne() 或者 wakeAll() 的条件。使用 wakeOne() 可以唤醒一个随机选取的等待线程,而使用 wakeAll() 可以唤醒所有正在等待的线程。关于这个类的使用,可以参考一下帮助文档,也可以参考 Qt 提供的 Wait Conditions 示例程序,它演示了如何使用 QWaitCondition 来解决生产者—消费者问题。

下面将讲述一个使用信号量来解决生产者—消费者问题的例子,这个例子演示了怎样使用 QS_semaphore 信号量来保护对生产者线程和消费者线程共享的环形缓冲区的访问。生产者向缓冲区中写入数据,直到它到达缓冲区的终点,这时它会从起点重新开始,覆盖已经存在的数据。消费者线程读取产生的数据,并将其输出。这个例子中包括两个类: Producer 和 Consumer,它们都继承自 QThread。环形缓冲区用来在这两个类之间进行通信,保护缓冲区的信号量被设置为了全局变量。

(项目源码路径: src\19\19-4\mySemaphores) 新建空的 Qt 项目,项目名称为 mySemaphores,完成后向项目中添加新的 main.cpp 文件。下面进入 main.cpp 文件,先添加头文件包含和变量的定义:

```
#include <QtCore>
#include <stdio.h>
#include <stdlib.h>
#include <QDebug>

const int DataSize = 10;
const int BufferSize = 5;
char buffer[BufferSize];
QS_semaphore freeBytes(BufferSize);
QS_semaphore usedBytes;
```

这里 DataSize 变量的值是生产者将要产生的数据数量,为了使这个例子尽可能简单,这里将它设置为常量。BufferSize 是环形缓冲区的大小,比 DataSize 小,这意味着在某一时刻生产者将到达缓冲区的终点,然后从起点重新开始。

为了同步生产者和消费者,需要使用两个信号量。其中 freeBytes 信号量控制缓冲区的空闲区域(就是生产者还没有添加数据或者消费者已经进行了读取的区域);usedBytes 信号量控制已经使用的缓冲区区域(就是生产者已经添加数据但是消费者还没有进行读取的区域)。这两个信号量一起保证了生产者永远不会在消费者前多于

BufferSize个字节,而消费者永远不会读取生产者还没有生产的数据。freeBytes信号量初始化为BufferSize,因为要保证最开始整个缓冲区是空的;usedBytes信号量初始化为0。

下面添加生产者类,继续添加如下代码:

```
class Producer : public QThread
{
public:
    void run();
};

void Producer::run()
{
    qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    for (int i = 0; i < DataSize; ++i) {
        freeBytes.acquire();
        buffer[i % BufferSize] = "ACGT"[qrand() % 4];
        qDebug() << QString("producer: %1").arg(buffer[i % BufferSize]);
        usedBytes.release();
    }
}
```

生产者产生 DataSize 字节的数据,在它将一个字节写入到环形缓冲区之前,它必须先使用 freeBytes 信号量获得一个空闲的字节。QSemaphore::acquire() 函数用于获得一定数量的资源,默认获得一个资源。如果要获得的资源数目大于可用的资源数目(可以使用 available() 函数获得),那么这个调用将会被阻塞,直到有足够的可用资源。然后使用随机数来随机获取“A”、“C”、“G”、“T”这 4 个字母中的一个,并进行了提示输出。最后生产者使用 release() 函数释放了 usedBytes 信号量的一个字节。这样,空闲字节成功转换成了已经被使用的字节,等待被消费者进行读取。

下面添加消费者类,继续添加如下代码:

```
class Consumer : public QThread
{
public:
    void run();
};

void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedBytes.acquire();
        qDebug() << QString("consumer: %1").arg(buffer[i % BufferSize]);
        freeBytes.release();
    }
}
```

```
    }
```

这里的代码与生产者是很相似的,只不过这次是获取一个已经使用的字节,然后释放一个空闲的字节。

下面来添加主函数,继续添加如下代码:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return app.exec();
}
```

这里创建了两个线程并且调用了 `QThread::wait()` 来确保在程序退出以前这两个线程有时间执行完毕。程序的执行过程: 最初, 只有生产者线程可以执行, 消费者线程被阻塞来等待 `usedBytes` 信号量被释放(它初始可用数量为 0)。一旦生产者将一个字节放入缓冲区, `freeBytes` 的可用大小即 `freeBytes.available()` 返回 `BufferSize - 1`, 而 `usedBytes.available()` 返回 1。这时, 可能发生两件事情: 或者消费者线程读取这个字节, 或者生产者线程产生第二个字节。两个线程的执行顺序是无法确定的。

最后需要在 `mySemaphores.pro` 文件中添加一行代码“`CONFIG += console`”, 这样便可以在控制台中显示结果了。现在可以运行程序查看输出结果。对应这个例子, 可以查看一下 Qt 提供的 `Semaphores Example` 示例程序。

19.2.3 可重入与线程安全

在查看 Qt 的帮助文档时, 很多类的开始都写着“All functions in this class are re-entrant”, 或者“All functions in this class are thread-safe”。在 Qt 文档中, 术语“可重入(reentrant)”和“线程安全(thread-safe)”用来标记类和函数, 来表明怎样在多线程应用程序中使用它们:

- 一个线程安全的函数可以同时被多个线程调用, 即便是这些调用使用了共享数据。因为该共享数据的所有实例都被序列化了。
- 一个可重入的函数也可以同时被多个线程调用, 但是只能是在每个调用使用自己的数据时。

因此, 一个线程安全的函数总是可重入的, 但是一个可重入的函数不总是线程安全的。推而广之, 如果只要每个线程使用一个类的不同实例, 该类的成员函数就可以被多个线程安全地调用, 那么这个类被称为可重入的; 如果即使所有的线程使用一个类的相

同实例,该类的成员函数也可以被多个线程安全调用,那么这个类被称为线程安全的。注意:只有在文档中标记为线程安全(thread-safe)的 Qt 类,它们的目的才是被用于多线程。所以,如果一个函数没有被标记为线程安全的或者可重入的,它不应该被多个线程使用;如果一个类没有标记为线程安全的或者可重入的,该类的一个特定的实例,不应该被多个线程访问。

1. 可重入

C++类一般是可重入的,因为它们只访问自己的数据成员。任何线程都可以调用一个可重入类的一个实例的一个成员函数,只要没有其他线程在同一时间调用该类的相同实例的成员函数即可。例如,下面的 Counter 类是可重入的:

```
class Counter
{
public:
    Counter() { n = 0; }
    void increment() { ++n; }
    void decrement() { --n; }
    int value() const { return n; }
private:
    int n;
};
```

这个类并不是线程安全的,因为如果多个线程尝试修改数据成员 n,结果便是不可预测的。这是因为++和--操作并不总是原子的(原子操作即一个操作不会被其他线程中断)。事实上,它们会被分为 3 个机器指令:第一条指令向寄存器中加载变量的值;第二条指令递增或者递减寄存器的值;第三条指令将寄存器的值存储到内存中。如果线程 A 和线程 B 同时加载了变量的旧值,然后递增它们的寄存器并存储回去,它们相互覆盖,结果变量只递增了一次。

2. 线程安全

对于前面讲到的线程 A 和线程 B 的情况,很明显,访问应该序列化进行:在线程 B 执行相同的操作前,线程 A 必须执行完 3 条机器指令而不能被中断,反之亦然。一个简单的方法来使类成为线程安全的,就是使用 QMutex 来保护对数据成员的所有访问:

```
class Counter
{
public:
    Counter() { n = 0; }
    void increment() { QMutexLocker locker(&mutex); ++n; }
    void decrement() { QMutexLocker locker(&mutex); --n; }
    int value() const { QMutexLocker locker(&mutex); return n; }
```

```
private:  
    mutable QMutex mutex;  
    int n;  
};
```

这里的 `QMutexLocker` 类在其构造函数中自动锁住 `mutex`, 然后在析构函数调用时对其进行解锁, 在函数结束时会调用析构函数。锁住 `mutex` 确保了不同线程的访问可以序列化进行。数据成员 `mutex` 使用了 `mutable` 限定符, 是因为需要在 `value()` 函数中对 `mutex` 进行加锁和解锁, 而它是一个 `const` 函数。

19.2.4 线程和 `QObject`

`QThread` 继承自 `QObject`, 发射信号来告知线程的开始和结束执行等状态, 也提供了一些槽。这里比较感兴趣的是, `QObject` 可以在多线程中使用发射信号来调用其他线程中的槽, 而且向其他线程中的对象发送事件。这些之所以成为可能, 是因为每一个线程都允许有自己的事件循环。

1. `QObject` 的可重入性

`QObject` 是可重入的。它的大多数非 GUI 子类, 例如 `QTimer`、`QTcpSocket`、`QUdpSocket`、`QFtp` 和 `QProcess`, 也都是可重入的, 可以在多个线程中同时使用这些类。需要注意, 这些类是被设计为在单一的线程中创建和使用的, 在一个线程中创建一个对象, 然后在另外一个线程中调用这个对象的一个函数, 是不能保证一定可以工作的。需要注意 3 个约束条件:

- `QObject` 的子对象必须在创建它的父对象的线程中创建。这意味着, 永远不要将 `QThread` 对象(`this`)作为在该线程中创建的对象的父对象(因为 `QThread` 对象本身是在其他线程中创建的)。
- 事件驱动对象只能在单一的线程中使用。具体来说, 这条规则应用在定时器机制和网络模块中。例如, 不可以在对象所在的线程以外的其他线程中启动一个定时器或者连接一个套接字。
- 必须确保在删除 `QThread` 对象以前删除在该线程中所创建的所有对象。可以通过在 `run()` 函数中在栈上创建对象来保证这一点。

虽然 `QObject` 是可重入的, 但是对于 GUI 类, 尤其是 `QWidget` 及其所有子类, 是不可重入的, 它们只能在主线程中使用。如前所述, `QCoreApplication::exec()` 也必须在主线程中调用。在实际应用中, 无法在主线程以外的线程中使用 GUI 类的问题, 可以简单地通过这样的方式来解决: 将一些非常耗时的操作放在一个单独的工作线程中来进行, 等该工作线程完成后将结果返回给主线程, 最后由主线程将结果显示到屏幕上。Qt 中的 `Mandelbrot` 和 `Blocking Fortune Client` 示例程序都使用了这种方法, 可以参考一下。

2. 每个线程的事件循环

每一个线程都可以有它自己的事件循环。初始化线程使用 `QCoreApplication::`

`exec()`来开启它的事件循环;其他的线程可以使用`QThread::exec()`来开启一个事件循环,如图19-2所示。与`QCoreApplication`相似,`QThread`提供了一个`exit()`函数和一个`quit()`槽。在一个线程中使用事件循环,使得该线程可以使用那些需要事件循环的非GUI类(例如,`QTimer`、`QTcpSocket`和`QProcess`)。也使得该线程可以关联任意一个线程的信号到一个指定线程的槽。

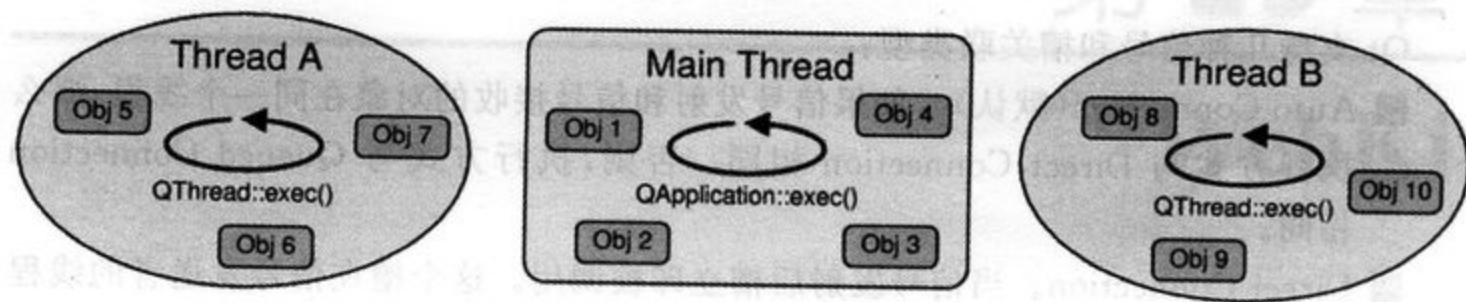


图 19-2 线程的事件循环

如果在线程中创建了一个`QObject`对象,那么这个`QObject`对象被称为居住在该线程(live in the thread)。发往这个对象的事件由该线程的事件循环进行分派。可以使用`QObject::thread()`获得该对象所在的线程。对于在`QApplication`之前创建的`QObject`对象,`QObject::thread()`返回0,这意味着主线程将只处理对于这些没有线程的对象的发送事件(posted events),而不会进行其他任何的事件处理。可以使用`QObject::moveToThread()`来改变对象及其孩子所在的线程(如果该对象有父对象,那么它无法被移动)。

在其他线程(不是拥有该`QObject`对象的线程)中`delete`删除(获取通过其他方式访问)该`QObject`对象是不安全的,除非可以确保当前该对象没有在处理事件。可以使用`QObject::deleteLater()`来代替,这样会发送一个DeferredDelete事件,最终该对象所在线程的事件循环将会获取该事件。默认的,拥有该`QObject`对象的线程,就是创建该`QObject`对象的线程,只要没有调用过`QObject::moveToThread()`函数。

如果没有运行事件循环,事件将不会传送到对象。例如,如果在一个线程中创建了一个`QTimer`对象,但是从来没有调用`exec()`函数,那么`QTimer`将永远不会发射它的`timeout()`信号。调用`deleteLater()`也不会工作。可以使用线程安全函数`QCoreApplication::postEvent()`在任何时间向任何线程的任何对象发送事件。该事件将会被创建该对象的线程的事件循环分派。所有的线程都支持事件过滤器,但是被监视的对象必须与监控对象在同一个线程中。相似的,`QCoreApplication::sendEvent`(与`postEvent()`不同)只能向调用该函数的线程中的对象分派事件。

3. 从其他线程中访问`QObject`子类

`QObject`和它所有的子类都不是线程安全的,这包括了整个事件传递系统。需要时刻记着,当正在从其他线程中访问一个对象时,可能事件循环正在向这个对象传递事件。如果调用一个没有在当前线程中的对象的函数,而这个对象可能会获得事件,那么就必须使用`mutex`来保护对这个`QObject`子类内部数据的所有访问,否则,可能出现

崩溃或者其他意外行为。

与其他对象相似, QThread 对象居住于(live in)创建该对象的线程。在 QThread 子类中提供槽一般是不安全的, 除非使用 mutex 来保护该成员变量。但是, 可以安全地在 QThread::run() 函数中发射信号, 因为信号发射是线程安全的。

4. 跨线程的信号和槽

Qt 支持几种信号和槽关联类型:

- Auto Connection(默认)。如果信号发射和信号接收的对象在同一个线程, 那么执行方式与 Direct Connection 相同。否则, 执行方式与 Queued Connection 相同。
- Direct Connection。当信号发射后槽立即被调用。这个槽在信号发送者的线程中执行, 而接收者并非必须在同一个线程。
- Queued Connection。当控制权返回到接收者线程的事件循环后才调用槽。槽在接收者的线程中被执行。
- Blocking Queued Connection。槽的调用与 Queued Connection 相同。不同的是当前线程会阻塞直到槽返回。注意: 使用这种方式关联在相同线程中的对象, 会引起死锁。
- Unique Connection。执行方式与 Auto Connection 相同。只不过关联必须是唯一的。例如, 如果在相同的两个对象之间, 相同的信号已经关联到了相同的槽上, 那么这个关联就不是唯一的, 这时 connect() 返回 false。

可以通过向 connect() 函数传递附加的参数来指定关联类型。要注意, 如果在接收者线程中有一个事件循环, 那么当发送者与接收者在不同的线程中时, 使用 Direct Connection 是不安全的。类似的, 调用其他线程中对象的任何函数也是不安全的。不过, 需要明确 QObject::connect() 函数本身是线程安全的。

19.3 小结

本章主要讲解了如何创建一个多进程或者多线程的应用程序, 还涉及了简单的进程间通信和线程间同步等内容。

Qt 中还提供了并发编程的支持, 可以在不使用互斥锁、读一写锁、等待条件或者信号量等低级线程原语的情况下编写多线程程序, 而且程序还可以发布到将来的多核心系统上。关于并发编程, 可以在帮助中查看一下 Concurrent Programming 关键字。

第 20 章

WebKit

WebKit 是一个开源的浏览器引擎。Qt 中提供了基于 WebKit 的 QtWebKit 模块, 它包含了一组相关的类。要使用该模块, 需要在.pro 项目文件中添加“Qt += webkit”一行代码, 可以通过添加 #include <QtWebKit> 头文件包含来使用该模块中的所有类, 当然也可以只包含具体使用到的类的头文件。

20.1 QtWebKit 模块

QtWebKit 提供了一个 Web 浏览器引擎, 使用它可以很容易地将万维网 (World Wide Web) 中的内容嵌入到 Qt 应用程序中。与此同时, 本地也可以对 Web 内容进行控制。QtWebKit 可以呈现 HTML (HyperText Markup Language, 超文本标记语言) 文档、XHTML (Extensible HyperText Markup Language, 可扩展超文本标记语言) 文档和 SVG (Scalable Vector Graphics, 可缩放矢量图形) 文档, 风格使用 CSS (Cascading Style Sheets, 层叠样式表), 脚本使用 JavaScript。在 JavaScript 执行环境和 Qt 对象模型间搭建的桥梁, 实现了使用 WebKit 的 JavaScript 环境访问本地对象。这一点可以在帮助中参考 The QtWebKit Bridge 关键字对应的文档。通过整合 Qt 的网络模块, 实现了从 Web 服务器、本地文件系统甚至 Qt 资源系统中透明的加载 Web 页面。

QWebView 是 QtWebKit 模块主要的窗体部件, 可以在各种应用程序中显示 Internet 上的网页内容。QWebView 作为一个窗口部件, 可以嵌入到窗体或者图形视图部件中, 而且该类还提供了便捷的函数来下载和显示站点内容, 例如:

```
QWebView * view = new QWebView(parent);
view -> load(QUrl("http://qt.nokia.com/"));
view -> show();
```

QWebView 用来显示 Web 页面, 每个 QWebView 实例都包含一个 QWebPage 对象。QWebPage 提供了对一个页面文档结构的访问, 描述了如框架 (frame)、访问历史记录和可编辑内容的撤销/重做栈等特色。每个 QWebPage 都包含一个 QWebFrame

对象作为它的主框架。HTML 中的每一个单独框架都可以使用 QWebFrame 来表示，这个类包含了到 JavaScript 窗口对象的桥梁，而且可以进行绘制。在 QWebPage 的主框架中可以包含很多的子框架。

综上所述，QWebView 中包含了一个 QWebPage，而在 QWebPage 中包含了一个或者多个 QWebFrame。它们的结构如图 20-1 所示。

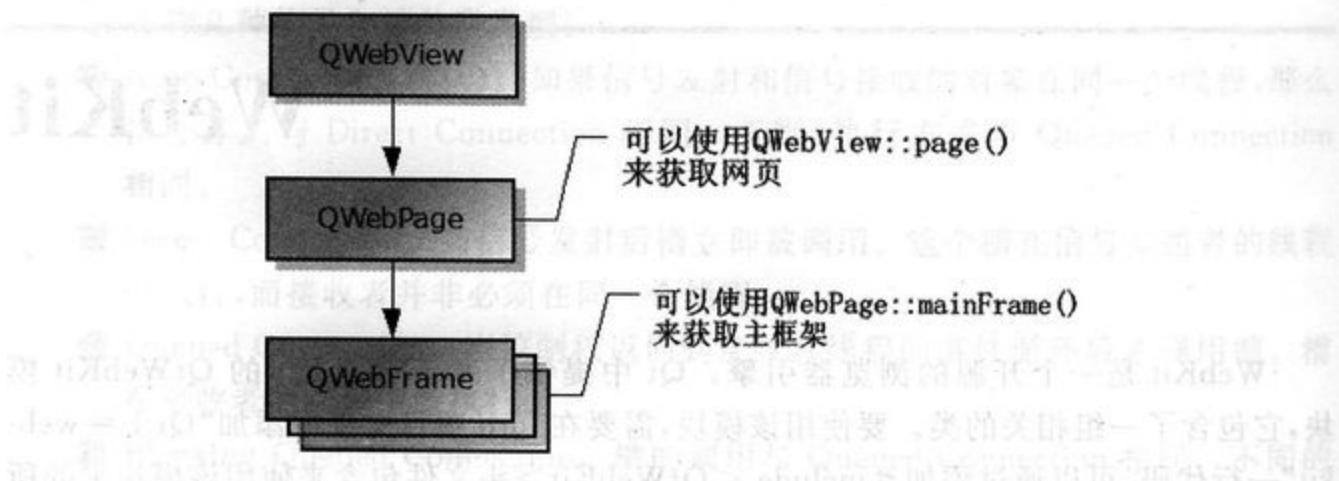


图 20-1 QtWebKit 架构图

HTML 文档中单独的元素可以通过 DOM JavaScript 接口访问，在 QtWebKit 中与这个接口等价的接口由 QWebElement 来表示。QWebElement 对象可以使用 QWebFrame 的 findAllElement() 和 findFirstElement() 函数来获取。一般网页浏览器的特色设置都可以通过 QWebSettings 类来配置，可以通过默认设置为所有的 QWebPage 实例提供默认值。单独的属性可以使用页面指定的设置对象进行重写。

20.2 基于 QtWebKit 的网页浏览器

这一节将介绍一个简单的网页浏览器的例子，在整个过程中还会对 QtWebKit 模块的各个组件做进一步的讲解。这个网页浏览器包含了最基本的网页显示、导航菜单、显示网站图标、提供历史记录以及字符串查找等功能。

20.2.1 显示一个网页

现在先来实现最简单的显示一个网页的功能，并添加常用的导航键。（项目源码路径：src\20\20-1\myWebKit）新建 Qt Gui 应用，项目名称为 myWebKit，类名为 MainWindow，基类选择 QMainWindow。完成后在 myWebKit.pro 文件中添加“QT += webkit”一行代码，并保存该文件。下面到 mainwindow.h 文件中，先添加类的前置声明：

```
class QWebView;
class QLineEdit;
```

然后添加槽声明：

```
protected slots:
    void changeLocation(); //改变路径
    void setProgress(int); //更新进度
    void adjustTitle(); //更新标题显示
    void finishLoading(bool); //加载完成后进行处理
```

然后添加私有对象和变量的定义：

```
QWebView * view;
QLineEdit * locationEdit;
int progress;
```

下面进入 mainwindow.cpp 文件中，先添加头文件包含：

```
#include <QWebView>
#include <QLineEdit>
```

然后在构造函数中添加如下代码：

```
progress = 0;
view = new QWebView(this);
setCentralWidget(view);
resize(800, 600);

//关联信号和槽
connect(view, SIGNAL(loadProgress(int)), this, SLOT(setProgress(int)));
connect(view, SIGNAL(titleChanged(QString)), this, SLOT(adjustTitle()));
connect(view, SIGNAL(loadFinished(bool)), this, SLOT(finishLoading(bool)));

locationEdit = new QLineEdit(this);
connect(locationEdit, SIGNAL(returnPressed()), this, SLOT(changeLocation()));

//向工具栏添加动作和部件
ui->mainToolBar->addAction(view->pageAction(QWebPage::Back));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Forward));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Reload));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Stop));
ui->mainToolBar->addWidget(locationEdit);

//设置并加载初始网页地址
locationEdit->setText("http://www.baidu.com");
view->load(QUrl("http://www.baidu.com"));
```

QWebView 类提供了一个窗口部件用来显示和编辑网页文档。因为它是一个窗口部件,所以可以嵌入到其他窗口中,这里将它作为主窗口的中心部件。可以使用 load() 函数来加载一个站点,比如这里加载了百度网站的主页网址。另外,也可以使用 setUrl() 来加载一个站点,如果已经拥有了可用的 HTML 内容,那么还可以使用 setHtml() 来代替。当 QWebView 开始加载时,会发射 loadStarted() 信号;而每当一个网页元素(例如一张图片或一个脚本等)加载完成时,都会发射 loadProgress() 信号;最后,当加载全部完成后,会发射 loadFinished() 信号,如果加载成功,该函数的参数为 true,否则为 false。可以使用 title() 来获取 HTML 文档的标题,如果标题发生了改变,将会发射 titleChanged() 信号。另外,使用 setTextSizeMultiplier() 函数可以设置 QWebView 中显示的文本的字体大小。如果需要自定义上下文菜单,可以重新实现 contextMenuEvent(),然后使用从 pageAction() 获得的动作来填充菜单,在 QWebPage 中通过枚举变量 QWebPage::WebAction 定义了几十个常用的功能动作,比如这里代码中添加的前进、后退等导航功能等。可以到 QWebPage 类的帮助文档中查看。另外,这些动作还可以自定义文本和图标,可以参考一下 QAction 类的帮助文档。

下面来添加几个槽的定义。首先是在行编辑器中改变站点地址后按下回车键执行的槽:

```
void MainWindow::changeLocation()
{
    QUrl url = QUrl(locationEdit ->text());
    view ->load(url);
    view ->setFocus();
}
```

这里从行编辑器中获得了站点网址,然后加载。下面添加更新加载进度槽的实现:

```
void MainWindow::setProgress(int p)
{
    progress = p;
    adjustTitle();
}
```

这里获取了当前的进度,然后调用了调整标题槽,下面添加它的定义:

```
void MainWindow::adjustTitle()
{
    if (progress <= 0 || progress >= 100) {
        setWindowTitle(view ->title());
    } else {
        setWindowTitle(QString(" %1 (%2%)").arg(view ->title()).arg(progress));
    }
}
```

在这里通过判断加载进度来进行标题的显示,如果正在加载,则显示加载进度,否则直接显示标题。下面添加完成加载后的处理槽的实现:

```
void MainWindow::finishLoading(bool finished)
{
    if (finished) {
        progress = 100;
        setWindowTitle(view->title());
    } else {
        setWindowTitle("web page loading error!");
    }
}
```

这里根据是否加载成功,分别显示不同的标题。现在可以运行程序,如果能正常显示百度主页,这时可以把地址更换为“<http://www.163.com>”,然后按下回车键来打开网易网站的首页,效果如图 20-2 所示。然后可以再试一下前进、后退等按钮的功能。



图 20-2 浏览网页运行效果

20.2.2 显示网站图标

每一个网站都有一个 Logo 图标,即所谓的 FavIcons,一般显示在网站标题的前面。可以使用 QWebView 的 icon() 函数来获取该图标。(项目源码路径: src\20\20-2\myWebKit)继续在前面的程序中添加代码。先在 mainwindow.h 文件中添加一个槽的声明:

```
void changeIcon();
```

然后在 mainwindow.cpp 文件中添加该槽的实现代码:

```
void MainWindow::changeIcon()
{
    setWindowIcon(view->icon());
}
```

这里将使用 icon() 获取的网站图片设置为了主窗口的图标。另外，还要在构造函数中添加如下代码：

```
view->settings()->setIconDatabasePath("./");
connect(view, SIGNAL(iconChanged()), this, SLOT(changeIcon()));
```

在加载网站图标以前，必须先创建用于存储网站图标的数据库，这个可以使用 QWebSetting 类的 setIconDatabasePath() 来实现，这个函数中只需要设置一下创建数据库的保存路径即可，它自动创建该数据库文件。然后关联了 QWebView 的 iconChanged() 信号，每当网页的图标被加载或者被改变时，都会发射该信号，这里利用该信号来实现显示不同网页时显示对应的图标。

每一个 QWebPage 对象都有它自己的 QWebSettings 对象，用来配置该页面的设置。QWebSettings 允许配置浏览器的属性，例如字体大小和类型、自定义样式表的位置、还有 JavaScript 和插件这样的通用属性。独立的属性可以使用 setAttribute() 函数来设置。枚举变量 QWebSettings::WebAttribute 定义了 QWebSettings 可以配置的所有属性，可以到该类的帮助文档中进行查看。QWebSettings 也用来配置全局的属性，例如网页的内存缓冲、图标数据库、本地数据库存储和离线应用程序存储等。

下面运行程序，效果如图 20-3 所示。

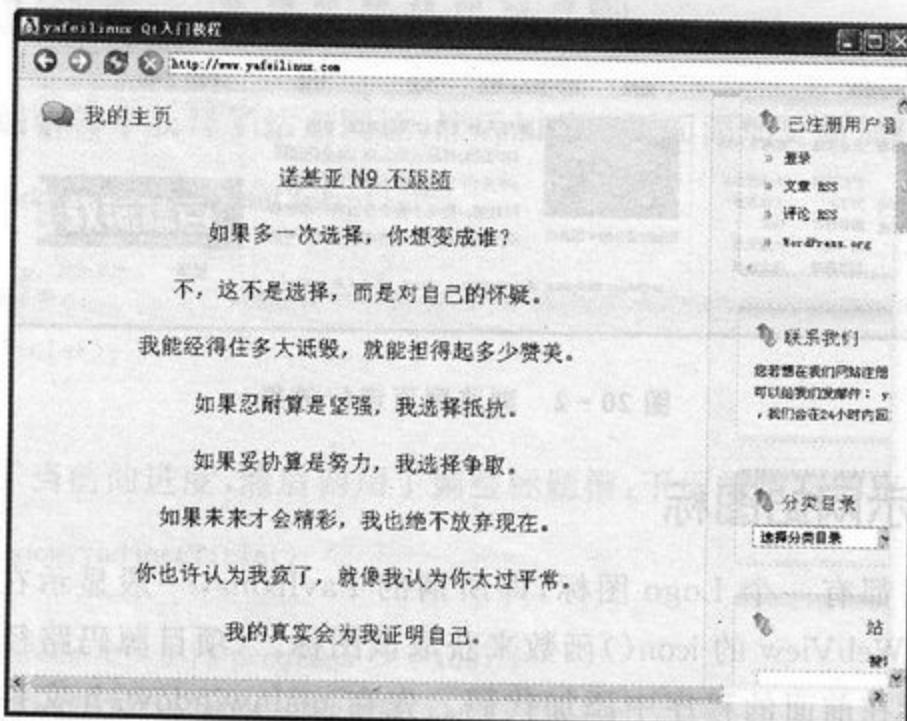


图 20-3 显示网站图标运行效果

20.2.3 显示历史记录

一般浏览器都支持显示浏览过的网页的历史记录，在 QtWebKit 中，QWebHistory 类表示了一个 QWebPage 的历史，下面继续添加代码来显示网页的浏览历史。（项目源码路径：src\20\20-3\myWebKit）先在 mainwindow.h 文件中添加头文件包含和类的前置声明：

```
#include <QModelIndex>
class QListWidget;
```

然后添加两个槽声明：

```
void showHistory(); //显示历史记录
void gotoHistory(QModelIndex); //转到历史记录
```

再添加一个私有对象定义：

```
QListWidget * historyList;
```

该列表部件用来显示浏览历史的条目。下面转到 mainwindow.cpp 文件中，先添加头文件包含：

```
#include <QListWidget>
#include <QWebHistory>
```

然后在构造函数中添加代码，先在 addWidget(locationEdit)一行代码前添加一行代码：

```
ui ->mainToolBar ->addAction(tr("历史"), this, SLOT(showHistory()));
```

这里向工具栏中添加了一个“历史”动作按钮，单击它可以显示历史记录窗口。然后在构造函数的最后添加如下代码：

```
historyList = new QListWidget;
historyList ->setWindowTitle(tr("历史记录"));
historyList ->setMinimumWidth(300);
connect(historyList, SIGNAL(clicked(QModelIndex)), this, SLOT(gotoHistory(QModelIndex)));
```

这里创建了列表部件，然后对其进行了初始化设置。现在添加显示历史记录窗口的槽的定义：

```
void MainWindow::showHistory()
{
    QList<QWebHistoryItem> list;
    list = view ->history() ->items();
    historyList ->clear();
    foreach (QWebHistoryItem item, list) {
```

```
QListWidgetItem * history = new QListWidgetItem(item.icon(), item.title());
historyList ->addItem(history);
}
historyList ->show();
}
```

使用 QWebView 的 history() 函数可以返回一个 QWebHistory 对象, 它表示了一个 QWebPage 的历史。QWebHistory 使用了当前项的概念, 将访问过的页面分为了当前项、当前项以前的项目和当前项以后的项目 3 部分。可以分别使用 back() 和 forward() 函数向后或者向前进行跳转。可以使用 currentItem() 来获取当前项。历史中任意的项目都可以使用 goToItem() 函数指定为当前项。使用 backItems() 可以获取所有可以向后浏览的项目的列表, forwardItems() 可以获取所有可以向前浏览的项目的列表, 而所有项目的列表可以使用 items() 函数获取。所有项目的总数可以使用 count() 获取, 还可以使用 clear() 来清空历史。历史中的一条项目由 QWebHistoryItem 来表示, 其中包含了页面信息、页面的地址以及最后一次访问的时间等属性, 该类提供了多个函数来访问这些属性, 如表 20-1 所列。

表 20-1 QWebHistoryItem 类中访问属性的函数

函数	描述
title()	页面标题
url()	页面地址
originalUrl()	用于访问页面的 URL
lastVisited()	用户上次访问该页面的日期和事件
icon()	与网页相关的图标
userData()	与历史条目一起存储的用户特定的数据

下面添加单击历史记录条目的槽的实现:

```
void MainWindow::gotoHistory(QModelIndex index)
{
    QUrl url = view ->history() ->itemAt(index.row()).url();
    view ->load(url);
}
```

这里获取了单击的条目的网页路径, 然后加载该路径。因为代码中使用了中文, 所以还要在 main.cpp 文件中添加相应的代码。现在运行程序, 效果如图 20-4 所示。



图 20-4 显示历史记录运行效果

20.2.4 链接跳转和查找功能

单击百度主页上的一些超链接时,它却没能自动完成页面的跳转,这可以通过手动添加代码来实现。另外,QWebPage 中还提供了函数来实现网页中字符串的查找和高亮显示。下面我们就来看一下这些功能的应用。

(项目源码路径: src\20\20-4\myWebKit)先在 mainwindow.h 文件中添加头文本包含 #include <QUrl>,然后添加两个槽声明:

```
void urlChanged(QUrl);      //单击超链接时进行跳转
void findText();            //查找字符串
```

下面进入 mainwindow.cpp 文件中,先在构造函数中添加代码:

```
view ->page() ->setLinkDelegationPolicy(QWebPage::DelegateAllLinks);
connect(view ->page(), SIGNAL(linkClicked(QUrl)), this, SLOT(urlChanged(QUrl)));
```

然后再添加改变地址槽的定义:

```
void MainWindow::urlChanged(QUrl url)
{
    view ->load(url);
}
```

这样就实现了单击页面中的超链接进行自动跳转的功能。每当用户单击一个链接时,QWebPage 都会发射 linkClicked()信号,不过,前提是要使用 setLinkDelegationPolicy()函数将链接代理策略设置为 DelegateAllLinks。

下面再到构造函数中,在 addWidget(locationEdit)一行代码前,添加如下一行

代码：

```
ui ->mainToolBar ->addAction(tr("查找"), this, SLOT(findText()));
```

然后再添加查找字符串槽的定义：

```
void MainWindow::findText()
{
    view ->page() ->findText("yafeilinux", QWebPage::HighlightAllOccurrences);
}
```

这样就实现了查找字符串“yafeilinux”，并将其高亮显示的功能。QWebPage 的 findText() 函数可以在页面中查找指定的字符串。这里还将该函数的第二个参数即查找标志设置为了 HighlightAllOccurrences，这样就可以高亮显示所有匹配的字符串了。效果如图 20-5 所示。

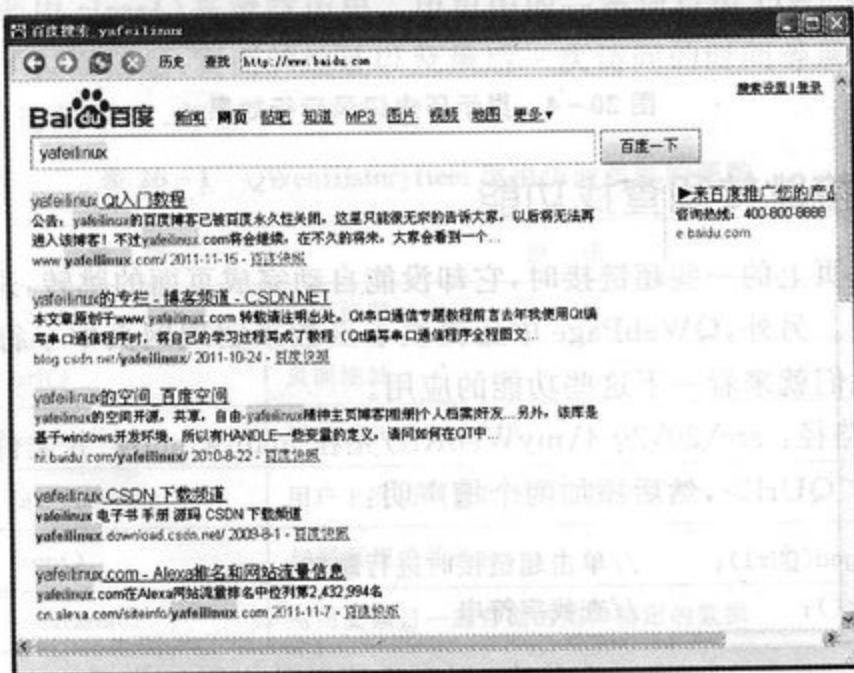


图 20-5 查找并高亮显示字符串效果

20.3 小结

本章讲述了 WebKit 在 Qt 中的简单应用。因为 WebKit 本身是一个非常庞大的项目，涉及了 HTML、CSS、JavaScript 等多方面的知识。因为这里只是作为入门的介绍，所以只讲解了 QtWebKit 中最基本的一些内容。Qt 对 WebKit 的支持是在逐渐加强的，对于现在一些支持不是太好的地方，在将来的版本中会进一步改善的。

参考文献

- [1] 布兰切特,萨墨菲尔德. C++ GUI Qt 4 编程(第 2 版)[M]. 同峰欣,等,译. 北京:电子工业出版社,2008.
- [2] 蔡志明,卢传富,李立夏,等. 精通 Qt 4 编程[M]. 北京:电子工业出版社,2008.
- [3] 成洁,卢紫毅. Linux 窗口程序设计——Qt4 精彩实例分析[M]. 北京:清华大学出版社,2008.
- [4] 李普曼,拉茹瓦. C++ Primer 中文版(第 4 版)[M]. 李师贤,等,译. 北京:人民邮电出版社,2006.
- [5] 王珊,萨师煊. 数据库系统概论(第 4 版)[M]. 北京:高等教育出版社,2006.
- [6] 谢希仁. 计算机网络(第 5 版)[M]. 北京:电子工业出版社,2008.

策划编辑：董立娟

封面设计：**runsign** 蓝正设计·赫健

Qt 应用编程系列丛书

《Qt及Qt Quick开发实战精解》

✓《Qt Creator快速入门》

内容简介

本书是基于Qt Creator集成开发环境的Qt入门书籍，详细介绍了Qt Creator开发环境的使用和Qt基本知识点的应用。本书内容主要包括Qt的基本应用，以及Qt在图形动画、影音媒体、数据处理和网络通信方面的应用内容。

读者对象

本书适合没有Qt编程基础、有Qt编程基础但是没有形成知识框架以及想学习Qt中某一方面应用的读者。对于想进一步学习Qt开发实例或者Qt Quick的读者，可以学习一下与本书配套的《Qt及Qt Quick开发实战精解》一书。

共享资料

本书配套源码可以从www.yafeilinux.com下载，还可以到Qt爱好者社区（www.qter.org）的本系列丛书专版讨论交流。

作者简介

霍亚飞，网名yafeilinux，嵌入式软件工程师，热爱编程，热爱开源！在博客中发表了大量Qt、Linux教程和开源软件，被众多网友奉为经典！参与创建了www.yafeilinux.com和Qt爱好者社区（www.qter.org），进行Qt及开源项目的推广和普及！

上架建议：程序设计/嵌入式系统

ISBN 978-7-5124-0783-1



9 787512 407831 >

定价：59.00元