

# Qt

## 学习之路 2

# 前言

本资料整理自豆子的博客(<http://www.devbean.net/>)

# 目录

## 目录

1 序.....	5
2 Qt 简介.....	6
3 Hello, world!.....	7
4 信号槽.....	14
5 自定义信号槽.....	18
6 Qt 模块简介.....	24
7 MainWindow 简介.....	28
8 添加动作.....	31
9 资源文件.....	36
10 对象模型.....	40
11 布局管理器.....	43
12 菜单栏、工具栏和状态栏.....	47
13 对话框简介.....	50
14 对话框数据传递.....	53
15 标准对话框 QMessageBox.....	56
16 深入 Qt5 信号槽新语法.....	60
17 文件对话框.....	64
18 事件.....	68
19 事件的接受与忽略.....	72
20 event().....	78
21 事件过滤器.....	81
22 事件总结.....	84
23 自定义事件.....	88
24 Qt 绘制系统简介.....	90
25 画刷和画笔.....	93
26 反走样.....	98
27 渐变.....	99
28 坐标系统.....	104
29 绘制设备.....	111
30 Graphics View Framework.....	115
31 贪吃蛇游戏（1）.....	118
32 贪吃蛇游戏（2）.....	122
33 贪吃蛇游戏（3）.....	128
34 贪吃蛇游戏（4）.....	132
35 文件.....	135
36 二进制文件读写.....	139
37 文本文件读写.....	143
38 存储容器.....	147
39 遍历容器.....	153

40 隐式数据共享 .....	160
41 model/view 架构 .....	162
42 QListWidget、QTreeWidget 和 QTableWidget .....	164
43 QStringListModel .....	170
44 QFileSystemModel .....	174
45 模型 .....	178
46 视图和委托 .....	183
47 视图选择 .....	187
48 QSortFilterProxyModel .....	193
49 自定义只读模型 .....	196
50 自定义可编辑模型 .....	201
51 布尔表达式树模型 .....	203
52 使用拖放 .....	215
53 自定义拖放数据 .....	221
54 剪贴板 .....	229
55 数据库操作 .....	231
56 使用模型操作数据库 .....	237
57 可视化显示数据库数据 .....	239
58 编辑数据库外键 .....	242
59 使用流处理 XML .....	245
60 使用 DOM 处理 XML .....	253
61 使用 SAX 处理 XML .....	258
62 保存 XML .....	263
63 使用 QJson 处理 JSON .....	266
64 使用 QJsonDocument 处理 JSON .....	270
65 访问网络 (1) .....	273
66 访问网络 (2) .....	277
67 访问网络 (3) .....	284
68 访问网络 (4) .....	289
69 进程 .....	290
70 进程间通信 .....	293
71 线程简介 .....	297
72 线程和事件循环 .....	299
73 Qt 线程相关类 .....	304
74 线程和 QObject .....	307
75 线程总结 .....	312
76 QML 和 QtQuick 2 .....	319
77 QML 语法 .....	323
78 QML 基本元素 .....	328
79 QML 组件 .....	333

# 1 序

[51CTO](#) 上面曾经有过这么一个系列，具体是 Qt 的入门教程。当时强调过，那些文章大致是根据《C++ GUI Programming with Qt 4, 2nd Edition》编写的。时过境迁，现在回头看看，已经过去了整整三年。如果你仔细看下那篇系列文章就会发现，发表时间竟然是 2009 年 8 月 20 日；而今天是 2012 年 8 月 20 日。或者是冥冥之中的感觉，竟然选择了同一个时间。

现在，按照年前做过的计划，我会来履行我的承诺，**重新修订《Qt 学习之路》**。不过，豆子计划将其取名为《Qt 学习之路 2》，或者就当作是 2.0 版本吧！

从网上的反应来看，这个系列的文章获得了很多读者的认可。时间已经过去三年，Qt 的发展也有了翻天覆地的变化。如果不受出售事件的影响，Qt 5 即将在 2012 年 9 月发布。而现在，最新代码库里面已经有了 beta。这意味着，Qt 5 的特性已经确定，不会再有大的改变。所以，我觉得，我已经可以着手进行一次修订。

本次修订的原则是，结构上大致保持前一版本的顺序不变，包括基本知识的介绍、常用 GUI 组件的介绍、常用技术的介绍等；内容上将结合 Qt 4 与 Qt 5 两个部分。在可以预见的未来，Qt 4 的程序，无论从旧代码的维护，还是新的程序的出现，都不会立刻退出历史舞台。Qt 5 也并不像 Qt 4 与 Qt 3 的升级那样的激烈，因此，我觉得有必要同时介绍这两个版本。当然，我并不确定这种“同时”会不会一直持续到系列的最末，因为也有可能 Qt 5 以一种摧枯拉朽之势，将 Qt 4 扫出历史舞台。这一切尚未可知。鉴于此，豆子才不将本系列命名为《Qt 5 学习之路》，而是以第二版称呼。

另外，对于上一版本，豆子还是很内疚的。因为并不是一个完整的介绍，Qt 的很多优秀特性，比如 XML，比如数据库，比如网络，都没有进行介绍。这主要是因为当时接触 Qt 也并不是很多，很多特性没有使用过，即便抄书写出来，也会觉得心里没底。现在豆子对 Qt 了解更多，所以，在这次修订中，豆子将竭尽全力将一些用到的特性介绍一下。

至于本系列的定位，豆子主张将其定位于入门教程。不过，如果可能的话，豆子希望能够在其中穿插一些有关 Qt 实现的相关内容。这部分内容肯定不会是基础的，比如信号槽的实现等。不过，对于这一点豆子也不敢肯定，毕竟要接触到实现层面上的东西，总要花费一定时间和精力。

这次修订，没有了《C++ GUI Programming with Qt 4》这本书作为提纲，一切都将按照自己的思路来。豆子将尽量跟随这本书的顺序，同时希望能够按照 Qt 5 的思路，按照模块来介绍 Qt。当然，作为修订版，本次修订的着重点在于 Qt 5，Qt 4 的内容将追随 Qt 5 进行介绍。同前文一样，本系列也会参考《C++ GUI Programming with Qt 4》一书，不过鉴于本书的某些自认为不合适的组织(比如以一个过大的项目作为示例)，本版更多会直接参考 Qt 文

档。很多原理性内容，可能会直接来源于文档，所以，感兴趣的朋友建议直接翻阅文档，以文档原文为准。

说了这么多，总之就是，尽量完成一篇相对高质量的教程。如果有任何建议或者意见，欢迎给豆子留言。

以此，是为序。

## 2 Qt 简介

Qt 是一个著名的 C++ 应用程序框架。你并不能说它只是一个 GUI 库，因为 Qt 十分庞大，并不仅仅是 GUI 组件。使用 Qt，在一定程度上你获得的是一个“一站式”的解决方案：不再需要研究 STL，不再需要 C++ 的<string>，不再需要到处去找解析 XML、连接数据库、访问网络的各种第三方库，因为 Qt 自己内置了这些技术。

Qt 是一个跨平台的框架。跨平台 GUI 通常有三种实现策略：

1. **API 映射：**API 映射是说，界面库使用同一套 API，将其映射到不同的底层平台上面。大体相当于将不同平台的 API 提取公共部分。比如说，将 Windows 平台上的按钮控件和 Mac OS 上的按钮组件都取名为 Button。当你使用 Button 时，如果在 Windows 平台上，则编译成按钮控件；如果在 Mac OS 上，则编译成按钮组件。这么做的好处是，所有组件都是原始平台自有的，外观和原生平台一致；缺点是，编写库代码的时候需要大量工作用于适配不同平台，并且，只能提取相同部分的 API。比如 Mac OS 的文本框自带拼写检测，但是 Windows 上面没有，则不能提供该功能。这种策略的典型代表是 wxWidgets。这也是一个标准的 C++ 库，和 Qt 一样庞大。它的语法看上去和 MFC 类似，有大量的宏。据说，一个 MFC 程序员可以很容易的转换到 wxWidgets 上面来。
2. **API 模拟：**前面提到，API 映射会“缺失”不同平台的特定功能，而 API 模拟则是解决这一问题。不同平台的有差异 API，将使用工具库自己的代码用于模拟出来。按照前面的例子，Mac OS 上的文本框有拼写检测，但是 Windows 的没有。那么，工具库自己提供一个拼写检测算法，让 Windows 的文本框也有相同的功能。API 模拟的典型代表是 wine —— 一个 Linux 上面的 Windows 模拟器。它将大部分 Win32 API 在 Linux 上面模拟了出来，让 Linux 可以通过 wine 运行 Windows 程序。由此可以看出，API 模拟最大优点是，应用程序无需重新编译，即可运行到特定平台上。另外一个例子是微软提供的 DirectX，这个开发库将屏蔽掉不同显卡硬件所提供的具体功能。使用这个库，你无需担心硬件之间的差异，如果有的显卡没有提供该种功能，SDK 会使用软件的方式加以实现。（关于举例，可以参考文末一段精彩的讨论。）

3. **GUI 模拟：**任何平台都提供了图形绘制函数，例如画点、画线、画面等。有些工具库利用这些基本函数，在不同绘制出自己的组件，这就是 GUI 模拟。GUI 模拟的工作量无疑是很大的，因为需要使用最基本的绘图函数将所有组件画出来；并且这种绘制很难保证和原生组件一模一样。但是，这一代价带来的优势是，可以很方便的修改组件的外观——只要修改组件绘制函数即可。很多跨平台的 GUI 库都是使用的这种策略，例如 gtk+（这是一个 C 语言的图形界面库。使用 C 语言很优雅地实现了面向对象程序设计。不过，这也同样带来了一个问题——使用大量的类型转换的宏来模拟多态，并且它的函数名一般都比较长，使用下划线分割单词，看上去和 Linux 如出一辙。gtk+ 并不是模拟的原生界面，而有它自己的风格，所以有时候就会和操作系统的界面格格不入。），Swing 以及我们的 Qt。

Qt 和 wxWidgets 一样，也是一个标准的 C++ 库。但是它的语法类似于 Java 的 Swing，十分清晰，而且使用信号槽（signal/slot）机制，让程序看起来很明白——这也是很多人优先选择 Qt 的一个很重要的原因。不过，所谓“成也萧何，败也萧何”。这种机制虽然很清楚，但是它所带来的后果是你需要使用 Qt 的 moc 对程序进行预处理，才能够再使用标准的 make 或者 nmake 进行正常的编译，并且信号槽的调用要比普通的函数调用慢大约一个数量级（Qt 4 文档中说明该数据，但 Qt 5 尚未有官方说明）。Qt 的界面也不是原生风格的，尽管 Qt 使用 style 机制十分巧妙地模拟了原生界面。另外值得一提的是，Qt 不仅仅能够运行在桌面环境中，还可以运行在嵌入式平台以及手机平台。

Qt 第一版于 1991 年由 Trolltech（奇趣科技）发布。后来在 2008 年，Nokia 斥资 1.5 亿美元收购 TrollTech，将 Qt 应用于 Symbian 程序开发。2012 年 8 月 9 日，Nokia 将 Qt 以 400 万欧元的价格出售给 Digia。

伴随着 Qt，一直有两种授权协议：商业授权以及开源授权。在 Qt 的早期版本，商业授权包含一些开源授权不提供的组件，但是在近期版本则不存在这个问题。以往人们对 Qt 的开源授权多有诟病。早期版本的 Qt 使用与 GPL 不兼容的协议授权，这直接导致了 KDE 与 GNOME 的战争（由于 Linux 使用 GPL 协议发布，GPL 协议具有传染性，作为 Linux 桌面环境的 KDE 却是基于与 GPL 不兼容的 Qt 开发，这就不遵守 GPL 协议）。不过，现在 Qt 的开源版本使用的是 GPLv3 以及 LGPL 协议。这意味着，你可以将 Qt 作为一个库连接到一个闭源软件里面。可以说，Qt 协议的争议已经不存在了。

## 3 Hello, world!

想要学习 Qt 开发，首先要搭建 Qt 开发环境。好在现在搭建 Qt 开发环境还是比较简单的。我们可以到 [Qt 官方网站](#) 找到最新版本的 Qt。在 [Downloads](#) 页面，可以看到有几个版本的 Qt：Qt SDK、Qt Library、Qt Creator 等等。它们分别是：

- **Qt SDK:** 包含了 Qt 库以及 Qt 的开发工具 (IDE、i18n 等工具)，是一套完整的开发环境。当然，这个的体积也是最大的 (Windows 平台大约 1.7G, 其它平台大约 780M)。如果仅仅为开发 Qt，建议选择这一项下载安装。安装方法很简单，同普通程序没有什么区别。所需注意的是，安装过程中可能能够选择是否安装源代码，是否安装 mingw 编译器 (Windows)，这个就按照需要进行选择即可。另外值得说明的是，Qt SDK 通常比单独的 Qt 库版本要旧一些。比如现在 Qt 正式版是 4.8.2，但是 Qt SDK 的最新版 1.2.1 中包含的 Qt 是 4.8.1。
- **Qt Library:** 仅包含 Qt 库。如果您已经安装了 Qt 开发环境，为了升级一下 SDK 中提供的 Qt 库版本，就可以安装这一个。安装过之后，应该需要在 IDE 中配置安装路径，以便找到最新版本的 Qt (如果不是覆盖安装的话)。
- **Qt Creator:** 基于 Qt 构建的一个轻量级 IDE，现在最新版是 2.5.2，还是比较好用的，建议使用 Qt Creator 进行开发。当然，如果你已经习惯了 VS2010 这样的工具，可以在页面最下方找到相应的 Addin。很多朋友希望阅读 Qt 代码以提高自己的开发水平。当然，Qt 的经典代码是 KDE，不过这个项目不大适合初学者阅读。此时，我们就可以选择阅读 Qt Creator 的代码，它的代码还是比较清晰的。

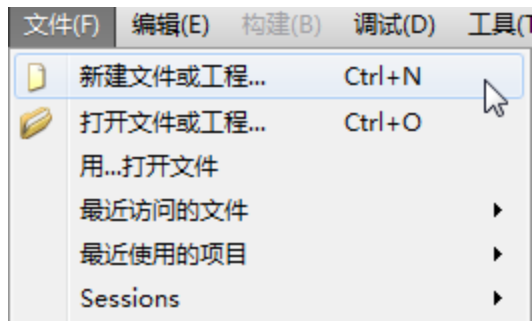
当我们安装完成 Qt 开发环境之后，就可以使用 Qt Creator 进行开发。在本系列中，豆子会一直使用这个 IDE 进行讲解。至于编译器，豆子一般会使用 mingw 或者 gcc。为了编译 Qt 5 的程序，你应该使用 gcc 4.5 以上的版本，这意味着，如果你是使用 Qt SDK 自带的 mingw，是不能编译 Qt 5 的程序的 (因为这个自带的版本是 4.4)，你应该升级 mingw 为 4.5 以上版本。

至此，我们已经有了 Qt 4 的完整开发环境。如果你想要开发 Qt 5，由于现在 (2012 年 8 月) Qt 5 还处于测试阶段，并没有提供二进制库，所以我们需要使用 git 自己获取 Qt 5 的源代码自己编译 (一般需要几个小时时间)。豆子非常不建议在 Windows 上编译 Qt 5，因为可能会出很多问题。如果你想尝试，可以参考[这里](#)。豆子提一句，在 Windows 上编译 Qt 5，需要安装 perl (并且要安装 GetOpt::Long 模块)、python 和 git，并且需要找到彼此路径。相比而言，Linux 上面就会简单很多。豆子建议，如果你想在 Windows 上尝试 Qt 5，可以考虑安装一个虚拟机，使用 Linux 平台；或者自己试着直接在 Windows 本地编译。豆子的环境是使用 openSUSE。openSUSE 的 [Qt 5.0 Development Snapshots](#) 已经提供了 Qt 5 二进制版本，免去了编译的过程。基于此，本文的 Qt 4 版本将在 Windows 平台上使用 mingw 进行测试；Qt 5 版本将在 openSUSE 上使用 gcc 4.6 进行测试。在未来官方推出 Qt 5 Windows 平台的二进制版本，也不排除在 Windows 上面测试 Qt 5 代码。

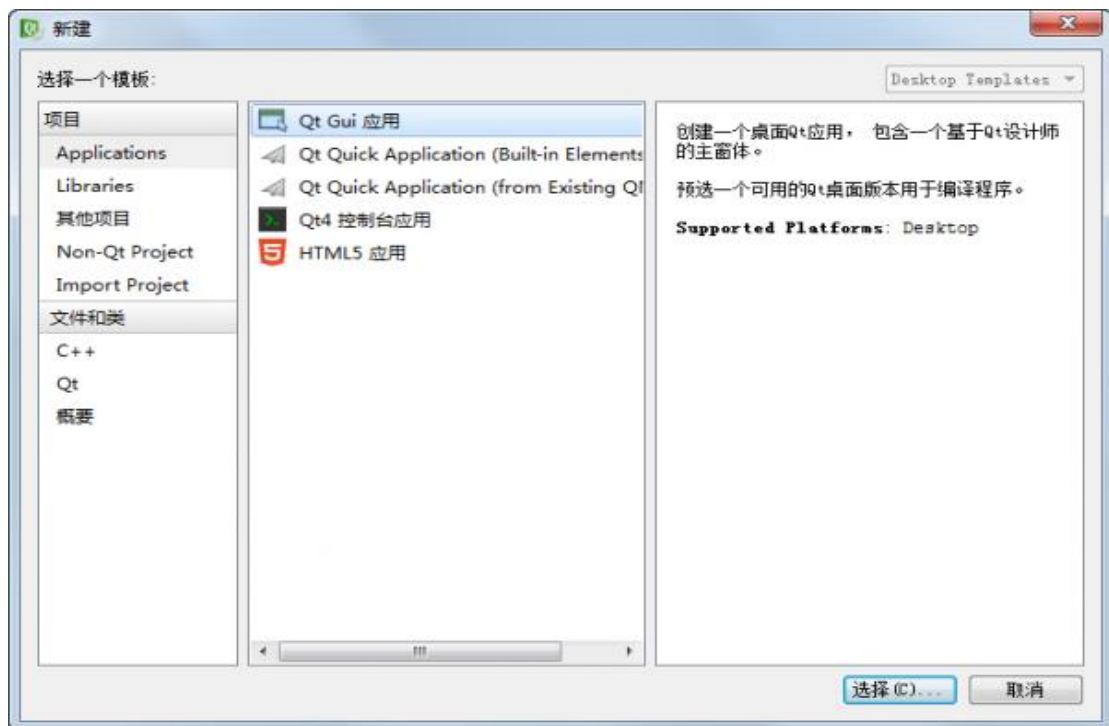
在 Qt Creator 中，我们可以在菜单栏的工具-选项-构建和运行的“Qt 版本”和“工具链”这两个选项卡中配置 Qt Creator 所使用的 Qt 版本和编译器。这或许是最重要的步骤，包括添加新的 Qt 版本以及以后的切换编译器或者 Qt 升级等。



下面尝试开发第一个 Qt 项目：HelloWorld。在 Qt Creator 中新建一个工程：



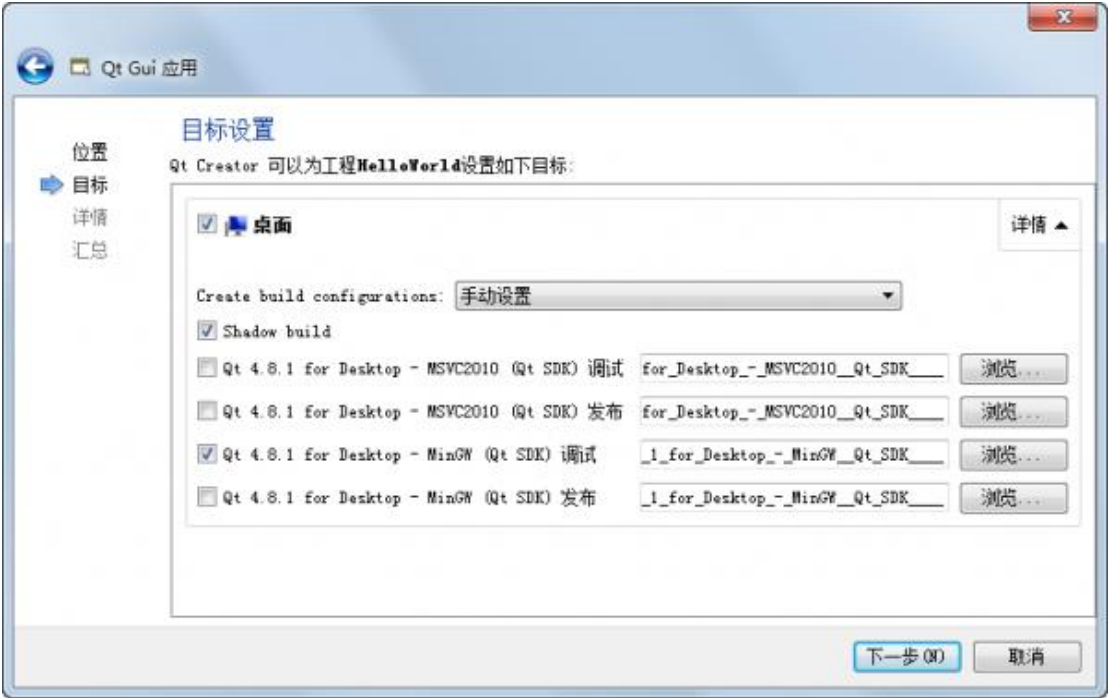
点击这个“新建文件或工程”，在左侧选择项目-Applications，中间选择 Qt Gui 应用，然后点击“选择...”：



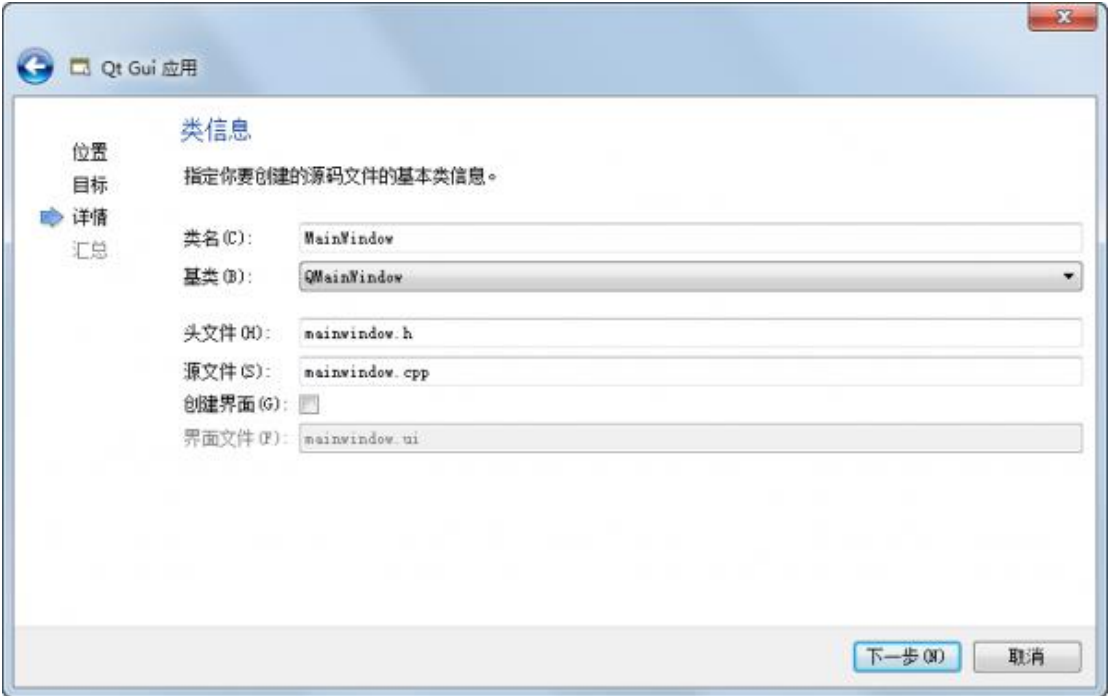
在弹出的对话框中填写名称、创建路径等信息：



点击“下一步”，选择该工程的编译器。这里我们只选择 mingw 调试即可（在以后的项目中，根据自己的需要选择。）Shadow Build 的含义是“影子构建”，即将构建生成的文件不放在源代码文件夹下。这样可以最大地保持源代码文件夹的整洁。



点击“下一步”，可以选择生成的主窗口文件。不过在我们的简单示例中是不需要这么复杂的窗口的，因此我们尽可能简单地选择，将“创建界面”的选择去除：



终于到了最后一步。这里是在询问我们是否添加版本控制。对于我们的小项目当然是不需要的，所以选择“无”，然后点击“完成”即可：



可以看到，Qt Creator 帮助我们在 HelloWorld 项目文件夹下生成了四个文件：main.cpp，mainwindow.cpp,mainwindow.h 和 HelloWorld.pro。pro 文件就是 Qt 工程文件(project file)，由 qmake 处理，生成 make 程序所需要的 makefile；main.cpp 里面就是一个 main 函数，作为应用程序的入口函数；其他两个文件就是先前我们曾经指定的文件名的文件。

我们将 main.cpp 修改如下：

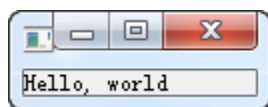
```
#include <QApplication>
#include <QLabel>

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

    QLabel label("Hello, world");
    label.show();

    return app.exec();
}
```

点击 Qt Creator 左侧下面的绿色三角按钮即可运行（这里一共有三个按钮，从上到下分别是“运行”、“调试”和“构建”）。如果没有错误的话，就会看到运行结果：



这个程序有这么几行。我们解释一下。

前两行是 C++ 的 `include` 语句,这里我们引入的是 `QApplication` 以及 `QLabel` 这两个类。`main()` 函数中第一句是创建一个 `QApplication` 类的实例。对于 Qt 程序来说, `main()` 函数一般以创建 `application` 对象 (GUI 程序是 `QApplication`, 非 GUI 程序是 `QCoreApplication`。 `QApplication` 实际上是 `QCoreApplication` 的子类。) 开始, 后面才是实际业务的代码。这个对象用于管理 Qt 程序的生命周期, 开启事件循环, 这一切都是必不可少的。在我们创建了 `QApplication` 对象之后, 直接创建一个 `QLabel` 对象, 构造函数赋值“Hello, world”, 当然就是能够在 `QLabel` 上面显示这行文本。最后调用 `QLabel` 的 `show()` 函数将其显示出来。`main()` 函数最后, 调用 `app.exec()`, 开启事件循环。我们现在可以简单地将事件循环理解成一段无限循环。正因为如此, 我们在栈上构建了 `QLabel` 对象, 却能够一直显示在那里 (试想, 如果不是无限循环, `main()` 函数立刻会退出, `QLabel` 对象当然也就直接析构了)。

示例程序我们已经讲解完毕。下面再说一点。我们可以将上面的程序改写成下面的代码吗?

```
#include <QApplication>
#include <QLabel>

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

    QLabel *label = new QLabel("Hello, world");
    label->show();

    return app.exec();
}
```

答案是, 不可以建议这样做!

首先, 按照标准 C++ 来看这段程序。这里存在着内存泄露。当 `exec()` 退出时 (也就是事件循环结束的时候。窗口关闭, 事件循环就会结束), `label` 是没办法 `delete` 的。这就造成了内存泄露。当然, 由于程序结束, 操作系统会负责回收内存, 所以这个问题不会很严重。即便你这样修改了代码再运行, 也不会有任何错误。

早期版本的 Qt 可能会有问题（详见本文最后带有删除线的部分，不过豆子也没有测试，只是看到有文章这样介绍），不过在新版本的 Qt 基本不存在问题。在新版本的 Qt 中，`app.exec()`的实现机制确定，当最后一个可视组件关闭之后，主事件循环（也就是 `app.exec()`）才会退出，`main()`函数结束（此时会销毁 `app`）。这意味着，所有可视元素已经都关闭了，也就不存在后文提到的，`QPaintDevice` 没有 `QApplication` 实例这种情况。另外，如果你是显式关闭了 `QApplication` 实例，例如调用了 `qApp->quit()`之类的函数，`QApplication` 的最后一个动作将会是关闭所有窗口。所以，即便在这种情况下，也不会出现类这种问题。由于是在 `main()`函数中，当 `main()`函数结束时，操作系统会回收进程所占用的资源，相当于没有内存泄露。不过，这里有一个潜在的问题：操作系统只会粗暴地释放掉所占内存，并不会调用对象的析构函数（这与调用 `delete` 运算符是不同的），所以，很有可能有些资源占用不会被“正确”释放。事实上，在最新版的 Sailfish OS 上面就有这样的代码：

```
#include <QApplication>

int main(int argc, char *argv[])
{
    QScopedPointer<QApplication> app(new QApplication(argc, argv));
    QScopedPointer<QQuickView> view(new QQuickView);
    view->setSource("/path/to/main.qml");
    ...
    return app->exec();
}
```

这段代码不仅在堆上创建组件实例，更是把 `QApplication` 本身创建在了堆上。不过，注意，它使用了智能指针，因此我们不需要考虑操作系统直接释放内存导致的资源占用的问题。

当然，允许使用并不一定意味着我们建议这样使用。毕竟，这是种不好的用法（就像我们不推荐利用异常控制业务逻辑一样），因为存在内存泄露。而且对程序维护者也是不好的。所以，我们还是推荐在栈上创建组件。因为要靠人工管理 `new` 和 `delete` 的出错概率要远大于在栈上的自动控制。除此之外，在堆上和在栈上创建已经没有任何区别。

如果你必须在堆上创建对象，不妨增加一句：

```
label->setAttribute(Qt::WA_DeleteOnClose);
```

这点提示足够告诉程序维护者，你已经考虑到内存问题了。

~~严重的是，`label` 是建立在堆上的，`app` 是建立在栈上的。这意味着，`label` 会在 `app` 之后析构。也就是说，`label` 的生命周期长于 `app` 的生命周期。这可是 Qt 编程的大忌。因为在 Qt 中，所有的 `QPaintDevice` 必须要在有 `QApplication` 实例的情况下创建和使用。大家好奇的话，可以提一句，`QLabel` 继承自 `QWidget`，`QWidget` 则是 `QPaintDevice` 的子类。~~

之所以上面的代码不会有问题，是因为 `app` 退出时，`label` 已经关闭，这样的话，`label` 的所有 `QPaintDevice` 一般都不会被访问到了。但是，如果我们的程序，在 `app` 退出时，组件却没有关闭，这就会造成程序崩溃。（如果你想知道，怎样做才能让 `app` 退出时，组件却不退出，那么豆子可以告诉你，当你的程序在打开了一个网页的下拉框时关闭窗口，你的程序就会崩溃了！）

## 4 信号槽

信号槽是 Qt 框架引以为豪的机制之一。熟练使用和理解信号槽，能够设计出解耦的非常漂亮的程序，有利于增强我们的技术设计能力。

所谓信号槽，实际就是观察者模式。当某个事件发生之后，比如，按钮检测到自己被点击了一下，它就会发出一个信号（`signal`）。这种发出是没有目的的，类似广播。如果有对象对这个信号感兴趣，它就会使用连接（`connect`）函数，意思是，用自己的一个函数（成为槽（`slot`））来处理这个信号。也就是说，当信号发出时，被连接的槽函数会自动被回调。这就类似观察者模式：当发生了感兴趣的事件，某一个操作就会被自动触发。（这里提一句，Qt 的信号槽使用了额外的处理来实现，并不是 GoF 经典的观察者模式的实现方式。）

为了体验一下信号槽的使用，我们以一段简单的代码说明：

```
// !!! Qt 5
#include <QApplication>
#include <QPushButton>

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

    QPushButton button("Quit");
    QObject::connect(&button, &QPushButton::clicked, &QApplication::quit);
    button.show();

    return app.exec();
}
```

这里再次强调，我们的代码是以 Qt 5 为主线，这意味着，有的代码放在 Qt 4 上是不能编译的。因此，豆子会在每一段代码的第一行添加注释，用以表明该段代码是使用 Qt 5 还是 Qt 4 进行编译。读者在测试代码的时候，需要自行选择相应的 Qt 版本。

我们按照前面文章中介绍的在 Qt Creator 中创建工程的方法创建好工程，然后将 `main()` 函数修改为上面的代码。点击运行，我们会看到一个按钮，上面有“Quit”字样。点击按钮，程序退出。

按钮在 Qt 中被称为 `QPushButton`。对它的创建和显示，同前文类似，这里不做过多的讲解。我们这里要仔细分析 `QObject::connect()` 这个函数。

在 Qt 5 中，`QObject::connect()` 有五个重载：

```
QMetaObject::Connection connect(const QObject *, const char *,
                                const QObject *, const char *,
                                Qt::ConnectionType);

QMetaObject::Connection connect(const QObject *, const QMetaMethod &,
                                const QObject *, const QMetaMethod &,
                                Qt::ConnectionType);

QMetaObject::Connection connect(const QObject *, const char *,
                                const char *,
                                Qt::ConnectionType) const;

QMetaObject::Connection connect(const QObject *, PointerToMemberFunction,
                                const QObject *, PointerToMemberFunction,
                                Qt::ConnectionType);

QMetaObject::Connection connect(const QObject *, PointerToMemberFunction,
                                Functor);
```

这五个重载的返回值都是 `QMetaObject::Connection`，现在不去关心这个返回值。下面我们先来看看 `connect()` 函数最常用的一般形式：

```
// !!! Qt 5
connect(sender, signal, receiver, slot);
```

这是我们最常用的形式。`connect()` 一般会使用前面四个参数，第一个是发出信号的对象，第二个是发送对象发出的信号，第三个是接收信号的对象，第四个是接收对象在接收到信号之后所需要调用的函数。也就是说，当 `sender` 发出了 `signal` 信号之后，会自动调用 `receiver` 的 `slot` 函数。

这是最常用的形式，我们可以套用这个形式去分析上面给出的五个重载。第一个，`sender` 类型是 `const QObject *`，`signal` 的类型是 `const char *`，`receiver` 类型是 `const QObject *`，

slot 类型是 `const char *`。这个函数将 `signal` 和 `slot` 作为字符串处理。第二个，`sender` 和 `receiver` 同样是 `const QObject *`，但是 `signal` 和 `slot` 都是 `const QMetaMethod &`。我们可以将每个函数看做是 `QMetaMethod` 的子类。因此，这种写法可以使用 `QMetaMethod` 进行类型比对。第三个，`sender` 同样是 `const QObject *`，`signal` 和 `slot` 同样是 `const char *`，但是却缺少了 `receiver`。这个函数其实是将 `this` 指针作为 `receiver`。第四个，`sender` 和 `receiver` 也都存在，都是 `const QObject *`，但是 `signal` 和 `slot` 类型则是 `PointerToMemberFunction`。看这个名字就应该知道，这是指向成员函数的指针。第五个，前面两个参数没有什么不同，最后一个参数是 `Functor` 类型。这个类型可以接受 `static` 函数、全局函数以及 `Lambda` 表达式。

由此我们可以看出，`connect()` 函数，`sender` 和 `receiver` 没有什么区别，都是 `QObject` 指针；主要是 `signal` 和 `slot` 形式的区别。具体到我们的示例，我们的 `connect()` 函数显然是使用的第五个重载，最后一个参数是 `QApplication` 的 `static` 函数 `quit()`。也就是说，当我们的 `button` 发出了 `clicked()` 信号时，会调用 `QApplication` 的 `quit()` 函数，使程序退出。

信号槽要求信号和槽的参数一致，所谓一致，是参数类型一致。如果不一致，允许的情况是，槽函数的参数可以比信号的少，即便如此，槽函数存在的那些参数的顺序也必须和信号的前面几个一致起来。这是因为，你可以在槽函数中选择忽略信号传来的数据（也就是槽函数的参数比信号的少），但是不能说信号根本没有这个数据，你就要在槽函数中使用（就是槽函数的参数比信号的多，这是不允许的）。

如果信号槽不符合，或者根本找不到这个信号或者槽函数的话，比如我们改成：

```
QObject::connect(&button, &QPushButton::clicked, &QApplication::quit2);
```

由于 `QApplication` 没有 `quit2` 这样的函数的，因此在编译时，会有编译错误：

```
'quit2' is not a member of QApplication
```

这样，使用成员函数指针，我们就不会担心在编写信号槽的时候会出现函数错误。

借助 Qt 5 的信号槽语法，我们可以将一个对象的信号连接到 `Lambda` 表达式，例如：

```
// !!! Qt 5
#include <QApplication>
#include <QPushButton>
#include <QDebug>

int main(int argc, char *argv[])
{
```



```

    QApplication app(argc, argv);

    QPushButton button("Quit");
    QObject::connect(&button, &QPushButton::clicked, [](bool) {
        qDebug() << "You clicked me!";
    });
    button.show();

    return app.exec();
}

```

注意这里的 Lambda 表达式接收一个 bool 参数，这是因为 QPushButton 的 clicked() 信号实际上是有一个参数的。Lambda 表达式中的 qDebug() 类似于 cout，将后面的字符串打印到标准输出。如果要编译上面的代码，你需要在 pro 文件中添加这么一句：

```
QMAKE_CXXFLAGS += -std=c++0x
```

然后正常编译即可。

Qt 4 的信号槽同 Qt 5 类似。在 Qt 4 的 QObject 中，有三个不同的 connect() 重载：

```

bool connect(const QObject *, const char *,
             const QObject *, const char *,
             Qt::ConnectionType);

bool connect(const QObject *, const QMetaMethod &,
             const QObject *, const QMetaMethod &,
             Qt::ConnectionType);

bool connect(const QObject *, const char *,
             const char *,
             Qt::ConnectionType) const

```

除了返回值，Qt 4 的 connect() 函数与 Qt 5 最大的区别在于，Qt 4 的 signal 和 slot 只有 const char \* 这么一种形式。如果我们将上面的代码修改为 Qt 4 的，则应该是这样的：

```

// !!! Qt 4
#include <QApplication>
#include <QPushButton>

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

```

```
QApplication app(argc, argv);

QPushButton button("Quit");
QObject::connect(&button, SIGNAL(clicked()),
                &app,    SLOT(quit()));

button.show();

return app.exec();
}
```

我们使用了 `SIGNAL` 和 `SLOT` 这两个宏，将两个函数名转换成了字符串。注意，即使 `quit()` 是 `QApplication` 的 `static` 函数，也必须传入一个对象指针。这也是 Qt 4 的信号槽语法的局限之处。另外，注意到 `connect()` 函数的 `signal` 和 `slot` 都是接受字符串，因此，不能将全局函数或者 `Lambda` 表达式传入 `connect()`。一旦出现连接不成功的情况，Qt 4 是没有编译错误的（因为一切都是字符串，编译期是不检查字符串是否匹配），而是在运行时给出错误。这无疑会增加程序的不稳定性。

信号槽机制是 Qt 的最大特性之一。这次我们只是初略了解了信号槽，知道了如何使用 `connect()` 函数进行信号槽的连接。在后面的内容中，我们将进一步介绍信号槽，了解如何设计自己的信号槽等等。

## 5 自定义信号槽

上一节我们详细分析了 `connect()` 函数。使用 `connect()` 可以让我们连接系统提供的信号和槽。但是，Qt 的信号槽机制并不仅仅是使用系统提供的那部分，还会允许我们自己设计自己的信号和槽。这也是 Qt 框架的设计思路之一，用于我们设计解耦的程序。本节将讲解如何在自己的程序中自定义信号槽。

信号槽不是 GUI 模块提供的，而是 Qt 核心特性之一。因此，我们可以在普通的控制台程序使用信号槽。

经典的观察者模式在讲解举例的时候通常会举报纸和订阅者的例子。有一个报纸类 `Newspaper`，有一个订阅者类 `Subscriber`。`Subscriber` 可以订阅 `Newspaper`。这样，当 `Newspaper` 有了新的内容的时候，`Subscriber` 可以立即得到通知。在这个例子中，观察者是 `Subscriber`，被观察者是 `Newspaper`。在经典的实现代码中，观察者会将自身注册到被观察者的一个容器中（比如 `subscriber.registerTo(newspaper)`）。被观察者发生了任何变化的时候，会主动遍历这个容器，依次通知各个观察者（`newspaper.notifyAllSubscribers()`）。

下面我们看看使用 Qt 的信号槽，如何实现上述观察者模式。注意，这里我们仅仅是使用这个案例，我们的代码并不是去实现一个经典的观察者模式。也就是说，我们使用 Qt 的信号槽机制来获得同样的效果。

```
//!!! Qt5
#include <QObject>

////////// newspaper.h
class Newspaper : public QObject
{
    Q_OBJECT
public:
    Newspaper(const QString & name) :
        m_name(name)
    {
    }

    void send()
    {
        emit newPaper(m_name);
    }

signals:
    void newPaper(const QString &name);

private:
    QString m_name;
};

////////// reader.h
#include <QObject>
#include <QDebug>

class Reader : public QObject
{
    Q_OBJECT
public:
    Reader() {}

    void receiveNewspaper(const QString & name)
    {
        qDebug() << "Receives Newspaper: " << name;
    }
};
```

```

////////// main.cpp
#include <QCoreApplication>

#include "newspaper.h"
#include "reader.h"

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    Newspaper newspaper("Newspaper A");
    Reader reader;
    QObject::connect(&newspaper, &Newspaper::newPaper,
                    &reader,    &Reader::receiveNewspaper);
    newspaper.send();

    return app.exec();
}

```

当我们运行上面的程序时，会看到终端输出 `Receives Newspaper: Newspaper A` 这样的字样。

下面我们来分析下自定义信号槽的代码。

这段代码放在了三个文件，分别是 `newspaper.h`, `reader.h` 和 `main.cpp`。为了减少文件数量，可以把 `newspaper.h` 和 `reader.h` 都放在 `main.cpp` 的 `main()` 函数之前吗？答案是，可以，但是需要有额外的操作。具体问题，我们在下面会详细说明。

首先看 `Newspaper` 这个类。这个类继承了 `QObject` 类。只有继承了 `QObject` 类的类，才具有信号槽的能力。所以，为了使用信号槽，必须继承 `QObject`。凡是 `QObject` 类（不管是直接子类还是间接子类），都应该在第一行代码写上 `Q_OBJECT`。不管是不是使用信号槽，都应该添加这个宏。这个宏的展开将为我们提供信号槽机制、国际化机制以及 Qt 提供的非基于 C++ RTTI 的反射能力。因此，如果你觉得你的类不需要使用信号槽，就不添加这个宏，就是错误的。其它很多操作都会依赖于这个宏。注意，这个宏将由 `moc`（我们会在后面章节中介绍 `moc`。这里你可以将其理解为一种预处理器，是比 C++ 预处理器更早执行的预处理器。）做特殊处理，不仅仅是宏展开这么简单。`moc` 会读取标记了 `Q_OBJECT` 的头文件，生成以 `moc_` 为前缀的文件，比如 `newspaper.h` 将生成 `moc_newspaper.cpp`。你可以到构建目录查看这个文件，看看到底增加了什么内容。注意，由于 `moc` 只处理头文件中的标记了 `Q_OBJECT` 的类声明，不会处理 `cpp` 文件中的类似声明。因此，如果我们的 `Newspaper` 和 `Reader` 类位于 `main.cpp` 中，是无法得到 `moc` 的处理的。解决方法是，我们手动调用 `moc` 工具处理 `main.cpp`，并且将 `main.cpp` 中的 `#include "newspaper.h"` 改为 `#include "moc_newspaper.h"` 就可以了。不过，这是相当繁琐的步骤，为了避免这样修改，

我们还是将其放在头文件中。许多初学者会遇到莫名其妙的错误，一加上 `Q_OBJECT` 就出错，很大一部分是因为没有注意到这个宏应该放在头文件中。

`Newspaper` 类的 `public` 和 `private` 代码块都比较简单，只不过它新加了一个 `signals`。`signals` 块所列出的，就是该类的信号。信号就是一个一个的函数名，返回值是 `void`（因为无法获得信号的返回值，所以也就无需返回任何值），参数是该类需要让外界知道的数据。信号作为函数名，不需要在 `cpp` 函数中添加任何实现（我们曾经说过，Qt 程序能够使用普通的 `make` 进行编译。没有实现的函数名怎么会通过编译？原因还是在 `moc`，`moc` 会帮我们实现信号函数所需要的函数体，所以说，`moc` 并不是单纯的将 `Q_OBJECT` 展开，而是做了很多额外的操作）。

`Newspaper` 类的 `send()` 函数比较简单，只有一个语句 `emit newPaper(m_name);`。`emit` 是 Qt 对 C++ 的扩展，是一个关键字（其实也是一个宏）。`emit` 的含义是发出，也就是发出 `newPaper()` 信号。感兴趣的接收者会关注这个信号，可能还需要知道是哪份报纸发出的信号？所以，我们将实际的报纸名字 `m_name` 当做参数传给这个信号。当接收者连接这个信号时，就可以通过槽函数获得实际值。这样就完成了数据从发出者到接收者的一个转移。

`Reader` 类更简单。因为这个类需要接受信号，所以我们将其继承了 `QObject`，并且添加了 `Q_OBJECT` 宏。后面则是默认构造函数和一个普通的成员函数。Qt 5 中，任何成员函数、`static` 函数、全局函数和 `Lambda` 表达式都可以作为槽函数。与信号函数不同，槽函数必须自己完成实现代码。槽函数就是普通的成员函数，因此作为成员函数，也会受到 `public`、`private` 等访问控制符的影响。（我们没有说信号也会受此影响，事实上，如果信号是 `private` 的，这个信号就不能在类的外面连接，也就没有任何意义。）

`main()` 函数中，我们首先创建了 `Newspaper` 和 `Reader` 两个对象，然后使用 `QObject::connect()` 函数。这个函数我们上一节已经详细介绍过，这里应该能够看出这个连接的含义。然后我们调用 `Newspaper` 的 `send()` 函数。这个函数只有一个语句：发出信号。由于我们的连接，当这个信号发出时，自动调用 `reader` 的槽函数，打印出语句。

这样我们的示例程序讲解完毕。我们基于 Qt 的信号槽机制，不需要观察者的容器，不需要注册对象，就实现了观察者模式。

下面总结一下自定义信号槽需要注意的事项：

- 发送者和接收者都需要是 `QObject` 的子类（当然，槽函数是全局函数、`Lambda` 表达式等无需接收者的时候除外）；
- 使用 `signals` 标记信号函数，信号是一个函数声明，返回 `void`，不需要实现函数代码；
- 槽函数是普通的成员函数，作为成员函数，会受到 `public`、`private`、`protected` 的影响；

- 使用 emit 在恰当的位置发送信号;
- 使用 QObject::connect()函数连接信号和槽。

## Qt4

下面给出 Qt 4 中相应的代码:

```
//!!! Qt4
#include <QObject>

////////// newspaper.h
class Newspaper : public QObject
{
    Q_OBJECT
public:
    Newspaper(const QString & name) :
        m_name(name)
    {
    }

    void send() const
    {
        emit newPaper(m_name);
    }

signals:
    void newPaper(const QString &name) const;

private:
    QString m_name;
};

////////// reader.h
#include <QObject>
#include <QDebug>

class Reader : public QObject
{
    Q_OBJECT
public:
    Reader() {}

public slots:
```

```

void receiveNewspaper(const QString & name) const
{
    qDebug() << "Receives Newspaper: " << name;
}
};

////////// main.cpp
#include <QCoreApplication>

#include "newspaper.h"
#include "reader.h"

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    Newspaper newspaper("Newspaper A");
    Reader reader;
    QObject::connect(&newspaper, SIGNAL(newPaper(QString)),
                    &reader,    SLOT(receiveNewspaper(QString)));
    newspaper.send();

    return app.exec();
}

```

注意下 Qt 4 与 Qt 5 的区别。

Newspaper 类没有什么区别。

Reader 类，receiveNewspaper()函数放在了 public slots 块中。在 Qt 4 中，槽函数必须放在由 slots 修饰的代码块中，并且要使用访问控制符进行访问控制。其原则同其它函数一样：默认是 private 的，如果要在外部访问，就应该是 public slots；如果只需要在子类访问，就应该是 protected slots。

main()函数中，QObject::connect()函数，第二、第四个参数需要使用 SIGNAL 和 SLOT 这两个宏转换成字符串（具体事宜我们在上一节介绍过）。注意 SIGNAL 和 SLOT 的宏参数并不是取函数指针，而是除去返回值的函数声明，并且 const 这种参数修饰符是忽略不计的。

下面说明另外一点，我们提到了“槽函数是普通的成员函数，作为成员函数，会受到 public、private、protected 的影响”，public、private 这些修饰符是供编译器在编译期检查的，因此其影响在于编译期。对于 Qt4 的信号槽连接语法，其连接是在运行时完成的，因此即便是

`private` 的槽函数也是可以作为槽进行连接的。但是，如果你使用了 Qt5 的新语法，新语法提供了编译期检查（取函数指针），因此取 `private` 函数的指针是不能通过编译的。

## 6 Qt 模块简介

Qt 5 与 Qt 4 最大的一个区别之一是底层架构有了修改。Qt 5 引入了模块化的概念，将众多功能细分到几个模块之中。Qt 4 也有模块的概念，但是是一种很粗的划分，而 Qt 5 则更加细化。本节主要对 Qt 5 的模块进行一个简单的介绍，以便以后大家需要哪些功能的时候知道到哪个模块去寻找。

Qt 5 模块分为 Essentials Modules 和 Add-on Modules 两部分。前者是基础模块，在所有平台上都可用；后者是扩展模块，建立在基础模块的基础之上，在能够运行 Qt 的平台之上可以酌情引入。

Qt 基础模块分为以下几个：

- **Qt Core**，提供核心的非 GUI 功能，所有模块都需要这个模块。这个模块的类包括了动画框架、定时器、各个容器类、时间日期类、事件、IO、JSON、插件机制、智能指针、图形（矩形、路径等）、线程、XML 等。所有这些类都可以通过 `<QtCore>` 头文件引入。
- **Qt Gui**，提供 GUI 程序的基本功能，包括与窗口系统的集成、事件处理、OpenGL 和 OpenGL ES 集成、2D 图像、字体、拖放等。这些类一般由 Qt 用户界面类内部使用，当然也可以用于访问底层的 OpenGL ES 图像 API。Qt Gui 模块提供的是所有图形用户界面程序都需要的通用功能。
- **Qt Multimedia**，提供视频、音频、收音机以及摄像头等功能。这些类可以通过 `<QtMultimedia>` 引入，而且需要在 `pro` 文件中添加 `QT += multimedia`。
- **Qt Network**，提供跨平台的网络功能。这些类可以通过 `<QtNetwork>` 引入，而且需要在 `pro` 文件中添加 `QT += network`。
- **Qt Qml**，提供供 QML（一种脚本语言，也提供 JavaScript 的交互机制）使用的 C++ API。这些类可以通过 `<QtQml>` 引入，而且需要在 `pro` 文件中添加 `QT += qml`。
- **Qt Quick**，允许在 Qt/C++ 程序中嵌入 Qt Quick（一种基于 Qt 的高度动画的用户界面，适合于移动平台开发）。这些类可以通过 `<QtQuick>` 引入，而且需要在 `pro` 文件中添加 `QT += quick`。



- **Qt SQL**, 允许使用 SQL 访问数据库。这些类可以通过 `<QtSql>` 引入, 而且需要在 `pro` 文件中添加 `QT += sql`。
- **Qt Test**, 提供 Qt 程序的单元测试功能。这些类可以通过 `<QtTest>` 引入, 而且需要在 `pro` 文件中添加 `QT += testlib`。
- **Qt Webkit**, 基于 WebKit2 的实现以及一套全新的 QML API (顺便说一下, Qt 4.8 附带的是 QtWebkit 2.2)。

Qt 扩展模块则有更多的选择:

- **Qt 3D**, 提供声明式语法, 在 Qt 程序中可以简单地嵌入 3D 图像。Qt 3D 为 Qt Quick 添加了 3D 内容渲染。Qt 3D 提供了 QML 和 C++ 两套 API, 用于开发 3D 程序。
- **Qt Bluetooth**, 提供用于访问蓝牙无线设备的 C++ 和 QML API。
- **Qt Contacts**, 用于访问地址簿或者联系人数据库的 C++ 和 QML API。
- **Qt Concurrent**, 封装了底层线程技术的类库, 方便开发多线程程序。
- **Qt D-Bus**, 这是一个仅供 Unix 平台使用的类库, 用于利用 D-Bus 协议进行进程间交互。
- **Qt Graphical Effects**, 提供一系列用于实现图像特效的类, 比如模糊、锐化等。
- **Qt Image Formats**, 支持图片格式的一系列插件, 包括 TIFF、MNG、TGA 和 WBMP。
- **Qt JS Backend**, 该模块没有公开的 API, 是 V8 JavaScript 引擎的一个移植。这个模块仅供 QtQml 模块内部使用。
- **Qt Location**, 提供定位机制、地图和导航技术、位置搜索等功能的 QML 和 C++ API。
- **Qt OpenGL**, 方便在 Qt 应用程序中使用 OpenGL。该模块仅仅为了程序从 Qt 4 移植到 Qt 5 的方便才保留下来, 如果你需要在新 Qt 5 程序中使用 OpenGL 相关技术, 需要使用的是 QtGui 模块中的 QOpenGL。
- **Qt Organizer**, 使用 QML 和 C++ API 访问组织事件 (organizer event)。organizer API 是 Personal Information Management API 的一部分, 用于访问 Calendar 信息。通过 Organizer API 可以实现: 从日历数据库访问日历时间、导入 iCalendar 事件或者将自己的事件导出到 iCalendar。

- **Qt Print Support**, 提供对打印功能的支持。
- **Qt Publish and Subscribe**, 为应用程序提供对项目值的读取、导航、订阅等的功能。
- **Qt Quick 1**, 从 Qt 4 移植过来的 QtDeclarative 模块, 用于提供与 Qt 4 的兼容。如果你需要开发新的程序, 需要使用 QtQuick 模块。
- **Qt Script**, 提供脚本化机制。这也是为提供与 Qt 4 的兼容性, 如果要使用脚本化支持, 请使用 QtQml 模块的 QJS\* 类。
- **Qt Script Tools**, 为使用了 Qt Script 模块的应用程序提供的额外的组件。
- **Qt Sensors**, 提供访问各类传感器的 QML 和 C++ 接口。
- **Qt Service Framework**, 提供客户端发现其他设备的服务。Qt Service Framework 为在不同平台上发现、实现和访问服务定义了一套统一的机制。
- **Qt SVG**, 提供渲染和创建 SVG 文件的功能。
- **Qt System Info**, 提供一套 API, 用于发现系统相关的信息, 比如电池使用量、锁屏、硬件特性等。
- **Qt Tools**, 提供了 Qt 开发的方便工具, 包括 Qt CLucene、Qt Designer、Qt Help 以及 Qt UI Tools 。
- **Qt Versit**, 提供了对 Versit API 的支持。Versit API 是 Personal Information Management API 的一部分, 用于 QContacts 和 vCard 以及 QOrganizerItems 和 iCalendar 之间的相互转换。
- **Qt Wayland**, 仅用于 Linux 平台, 用于替代 QWS, 包括 Qt Compositor API (server) 和 Wayland 平台插件 (clients)。
- **Qt WebKit**, 从 Qt 4 移植来的基于 WebKit1 和 QWidget 的 API。
- **Qt Widgets**, 使用 C++ 扩展的 Qt Gui 模块, 提供了一些界面组件, 比如按钮、单选框等。
- **Qt XML**, SAX 和 DOM 的 C++ 实现。该模块已经废除, 请使用 QXmlStreamReader/Writer。
- **Qt XML Patterns**, 提供对 XPath、XQuery、XSLT 和 XML Schema 验证的支持。

这里需要强调一点，由于 Qt 的扩展模块并不是 Qt 必须安装的部分，因此 Qt 在未来版本中可能会提供更多的扩展模块，这里给出的也仅仅是一些现在确定会包含在 Qt 5 中的一部分，另外还有一些，比如 Qt Active、Qt QA 等，则可能会在 beta 及以后版本中出现。

Qt 4 也分成若干模块，但是这些模块与 Qt 5 有些许多不同。下面是 Qt 4 的模块：

- **QtCore**，Qt 提供的非 GUI 核心类库，这一部分与 Qt 5 大致相同，只不过 Qt 4 的 core 类库中并不包含 JSON、XML 处理等。
- **QtGui**，图形用户界面组件，这个模块相当于 Qt 5 的 QtGui 与 QtWidgets 两个模块的总和。
- **QtMultimedia**，多媒体支持，类似 Qt 5 的相关部分。
- **QtNetwork**，网络支持，类似 Qt 5。
- **QtOpenGL**，提供对 OpenGL 的支持。在 Qt 5 中，这部分被移植到 QtGui 模块。
- **QtOpenVG**，提供对 OpenVG 的支持。
- **QtScript**，提供对 Qt Scripts 的支持。Qt Script 是一种类似于 JavaScript 的脚本语言。在 Qt 5 中，推荐使用 QtQml 的 JavaScript 部分。
- **QtScriptTools**，为 Qt Script 提供的额外组件。
- **QtSql**，提供对 SQL 数据库的支持。
- **QtSvg**，提供对 SVG 文件的支持。
- **QtWebKit**，提供显示和编辑 Web 内容。
- **QtXml**，XML 处理，这部分在 Qt 5 中被添加到了 QtCore。
- **QtXmlPatterns**，提供对 XQuery、XPath 等的支持。
- **QtDeclarative**，用于编写动画形式的图形用户界面的引擎。
- **Phonon**，多媒体框架。
- **Qt3Support**，Qt 3 兼容类库。

下面是 Qt 4 的一些工具模块：

- **QtDesigner**，用于扩展 Qt Designer。
- **QtUiTools**，用于在自己的引用程序中处理 Qt Designer 生成的 form 文件。
- **QtHelp**，联机帮助。
- **QtTest**，单元测试。

下面是专门供 Windows 平台的模块：

- **QAxContainer**，用于访问 ActiveX 控件。
- **QAxServer**，用于编写 ActiveX 服务器。

下面是专门供 Unix 平台的模块：

- **QtDBus**，使用 D-Bus 提供进程间交互。

## 7 MainWindow 简介

[前面一篇](#)大致介绍了 Qt 各个模块的相关内容，目的是对 Qt 框架有一个高屋建瓴般的了解。从现在开始，我们将开始尝试使用 Qt 开始新的历程。由于我们已经比较详细地介绍过信号槽的相关内容，因此我们可以用一个新的程序开始进一步的学习，同时对信号槽有一个比较深入的理解。

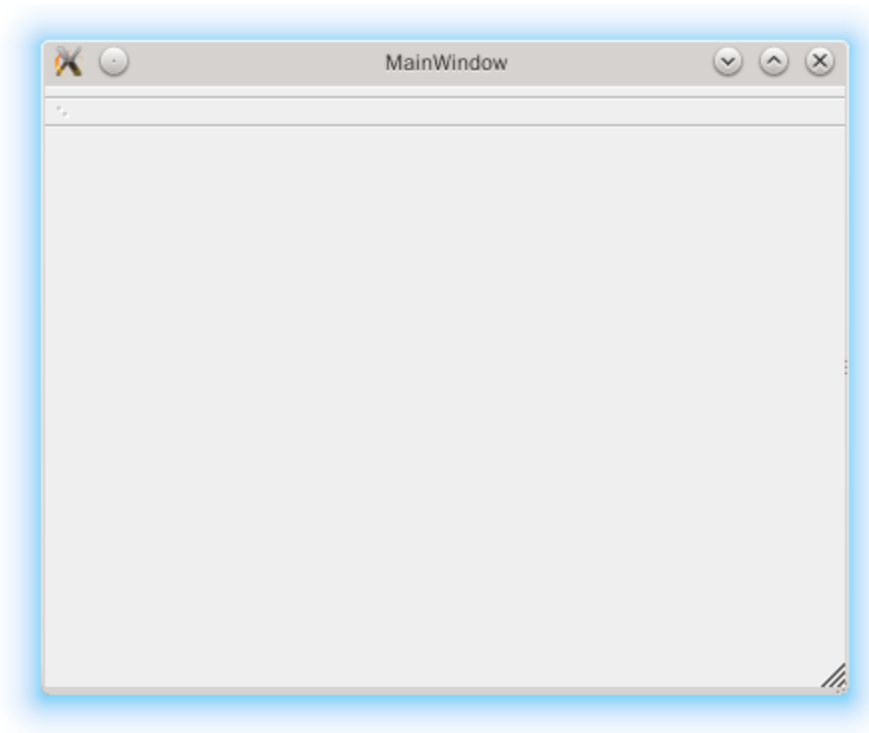
**QMainWindow** 是 Qt 框架带来的一个预定义好的主窗口类。所谓主窗口，就是一个普通意义上的应用程序(不是指游戏之类的那种)最顶层的窗口。比如你现在正在使用的浏览器，那么主窗口就是这个浏览器窗口。试着回想一下经典的主窗口，通常是由一个标题栏，一个菜单栏，若干工具栏和一个任务栏。在这些子组件之间则是我们的工作区。事实上，**QMainWindow** 正是这样的一种布局。

下面我们新建一个工程。还记得在新建工程的时候，Qt Creator 通常会帮助我们创建一个 **MainWindow** 吗？我们曾经为了介绍信号槽，将 **main()**函数做了修改。现在我们直接使用 Qt Creator 生成的代码来编译运行一下：

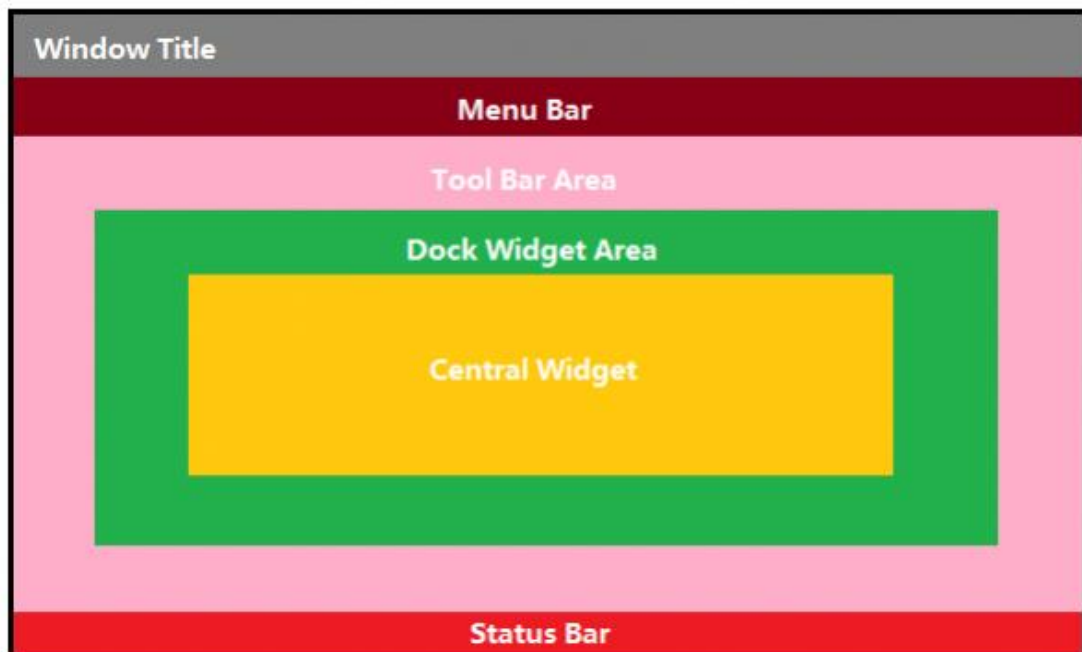
```
#include <QApplication>
#include "mainwindow.h"
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    MainWindow win;
```

```
win.show();  
return app.exec();  
}
```

在 openSUSE 上运行结果如下：



我们仔细看看这个窗口。虽然不太明显，但它实际上分成了几个部分：



主窗口的最上面是 **Window Title**，也就是标题栏，通常用于显示标题和控制按钮，比如最大化、最小化和关闭等。通常，各个图形界面框架都会使用操作系统本地代码来生成一个窗口。所以，你会看到在 **KDE** 上面，主窗口的标题栏是 **KDE** 样式的；在 **Windows** 平台上，标题栏是 **Windows** 风格的。如果你不喜欢本地样式，比如 **QQ** 这种，它其实是自己将标题栏绘制出来，这种技术称为 **DirectUI**，也就是无句柄绘制，这不在本文的讨论范畴。**Window Title** 下面是 **Menu Bar**，也就是菜单栏，用于显示菜单。窗口最底部是 **Status Bar**，称为状态栏。当我们鼠标滑过某些组件时，可以在状态栏显示某些信息，比如浏览器中，鼠标滑过带有链接的文字，你会在底部看到链接的实际 **URL**。

除去上面说的三个横向的栏，中间是以矩形区域表示。我们可以看出，最外层称为 **Tool Bar Area**，用于显示工具条区域。之所以是矩形表示，是因为，**Qt** 的主窗口支持多个工具条。你可以将工具条拖放到不同的位置，因此这里说是 **Area**。我们可以把几个工具条并排显示在这里，就像 **Word2003** 一样，也可以将其分别放置，类似 **Photoshop**。在工具条区域内部是 **Dock Widget Area**，这是停靠窗口的显示区域。所谓停靠窗口，就像 **Photoshop** 的工具箱一样，可以停靠在主窗口的四周，也可以浮动显示。主窗口最中间称为 **Central Widget**，就是我们程序的工作区。通常我们会将程序最主要的工作区域放置在这里，类似 **Word** 的稿纸或者 **Photoshop** 的画布等等。

对于一般的 **Qt** 应用程序，我们所需做的，就是编写我们的主窗口代码，主要是向其中添加各种组件，比如菜单、工具栏等，当然，最重要的就是当中的工作区。当我们将这些都处理完之后，基本上程序的工具也可以很好地实现。

通常我们的程序主窗口会继承自 **QMainWindow**，以便获得 **QMainWindow** 提供的各种便利的函数。这也是 **Qt Creator** 生成的代码所做的。

由于 **QMainWindow** 这个类在 **Qt 5** 中并没有什么改变，因此上面的代码可以直接拿到 **Qt 4** 中进行编译。事实上，我们使用 **Qt Creator** 生成的代码也是可以直接在 **Qt 4** 中编译。只不过需要注意一点：**Qt 4** 中没有 **widgets** 模块，因此在 **pro** 文件中，我们通常需要这么来写：

```
QT += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET    = qtdemo
TEMPLATE  = app

SOURCES   += main.cpp \
            mainwindow.cpp

HEADERS    += mainwindow.h
```

简单解释一下 pro 文件。首先，我们定义了 QT，用于告诉编译器，需要使用哪些模块。这些模块都在前面章节中有过介绍。我们通常需要添加 core 和 gui。第二行，如果 Qt 的主版本号（QT\_MAJOR\_VERSION）大于 4，也就是 Qt 5，则需要另外添加 widgets（因为在 Qt 5 中，所有组件都是在 widgets 模块定义的）。TARGET 是生成的程序的名字。TEMPLATE 是生成 makefile 所使用的模板，比如 app 就是编译成一个可执行程序，而 lib 则是编译成一个链接库（默认是动态链接库）。SOURCES 和 HEADERS 顾名思义，就是项目所需要的源代码文件和头文件。现在，我们只需使用默认的 pro 文件即可。以后随着项目的不断增大，pro 文件通常会非常复杂。

## 8 添加动作

本节，我们将在前面主窗口基础之上，添加菜单和工具栏等的动作。虽然 Qt Creator 已经帮我们实现了主窗口的框架代码，但是具体的功能，还是需要我们一行行添加。

Qt 使用 QAction 类作为动作。顾名思义，这个类就是代表了窗口的一个“动作”，这个动作可能显示在菜单，作为一个菜单项，当用户点击该菜单项，对用户的点击做出响应；也可能在工具栏，作为一个工具栏按钮，用户点击这个按钮就可以执行相应的操作。有一点值得注意：无论是出现在菜单栏还是工具栏，用户选择之后，所执行的动作应该都是一样的。因此，Qt 并没有专门的菜单项类，只是使用一个 QAction 类，抽象出公共的动作。当我们把 QAction 对象添加到菜单，就显示成一个菜单项，添加到工具栏，就显示成一个工具按钮。用户可以通过点击菜单项、点击工具栏按钮、点击快捷键来激活这个动作。

QAction 包含了图标、菜单文字、快捷键、状态栏文字、浮动帮助等信息。当把一个 QAction 对象添加到程序中时，Qt 自己选择使用哪个属性来显示，无需我们关心。同时，Qt 能够保证把 QAction 对象添加到不同的菜单、工具栏时，显示内容是同步的。也就是说，如果我们在菜单中修改了 QAction 的图标，那么在工具栏上面这个 QAction 所对应的按钮的图标也会同步修改。

下面我们来看看如何在 QMainWindow 中使用 QAction：

```
// !!! Qt 5
// ===== mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public QMainWindow
{
```

```

    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    void open();

    QAction *openAction;
};

#endif // MAINWINDOW_H

// ===== mainwindow.cpp
#include <QAction>
#include <QMenuBar>
#include <QMessageBox>
#include <QStatusBar>
#include <QToolBar>

#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    setWindowTitle(tr("Main Window"));

    openAction = new QAction(QIcon(":/images/doc-open"), tr("&Open..."), this);
    openAction->setShortcuts(QKeySequence::Open);
    openAction->setStatusTip(tr("Open an existing file"));
    connect(openAction, &QAction::triggered, this, &MainWindow::open);

    QMenu *file = menuBar()->addMenu(tr("&File"));
    file->addAction(openAction);

    QToolBar *toolBar = addToolBar(tr("&File"));
    toolBar->addAction(openAction);

    statusBar() ;
}

MainWindow::~MainWindow()
{
}

```



```
void MainWindow::open()
{
    QMessageBox::information(this, tr("Information"), tr("Open"));
}
```

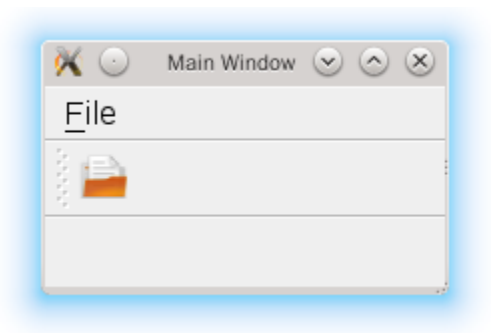
上面的代码分别属于两个文件：`mainwindow.h` 和 `mainwindow.cpp`。为了让 `MainWindow` 运行起来，我们还需要修改 `main()` 函数如下：

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

    MainWindow win;
    win.show();

    return app.exec();
}
```

当我们编辑好文件，点击运行，可以看到 `MainWindow` 的运行结果：



这是一个相对完整的程序。首先，我们在 `MainWindow` 类中添加了一个私有函数 `open()` 和一个私有变量 `openAction`。在 `MainWindow` 的构造函数中，第一行我们调用了 `setWindowTitle()`，设置主窗口的标题。注意我们的文本使用 `tr()` 函数，这是一个用于 Qt 国际化的函数。在后续章节中我们可以看到，我们可以使用 Qt 提供的国际化工具，将 `tr()` 函数的字符串提取出来，进行国际化。由于所需进行国际化的文本应该被大多数人认识，所以，`tr()` 函数里面一般会英文文本。

然后，我们在堆上创建了 `openAction` 对象。在 `QAction` 构造函数，我们传入了一个图标、一个文本和 `this` 指针。我们将在后面的文章中解释 `this` 指针的含义。

图标我们使用了 `QIcon`，传入值是一个字符串，这个字符串对应于 `Qt` 资源文件中的一段路径。`Qt` 资源文件的后缀名是 `qrc`。如果我们使用 `Qt Creator`，我们可以在新建文件中看到 `Qt` 资源文件。`Qt` 资源文件其实是一个 `XML` 描述的文件，表示 `Qt` 应用程序所需要的各个资源。我们可以使用普通文本编辑器打开这个文件：

```
<RCC>
  <qresource prefix="/images">
    <file alias="doc-open">document-open.png</file>
  </qresource>
</RCC>
```

我们会在后面的章节中详细介绍 `Qt` 资源文件（注意，资源文件需要在 `pro` 文件中使用 `RESOURCES` 引入。）。这里只需要了解，`QIcon` 的参数，以 `:` 开始，意味着从资源文件中查找资源。`:/images/doc-open` 就是找到了这里的 `document-open.png` 这个文件。（我们使用的是 `png` 格式的图片，这是 `Qt` 内置支持的图片格式。其他格式的图片，比如 `jpg`、`gif` 则需要插件支持。这些插件实际已经随着 `Qt` 一同发布。）

`QAction` 第二个参数中，文本值前面有一个 `&`，意味着这将成为一个快捷键。注意看截图中 `File` 的 `F` 有一个下划线。

下面一句，我们使用了 `setShortcut()` 函数，用于说明这个 `QAction` 的快捷键。`Qt` 的 `QKeySequence` 为我们定义了很多内置的快捷键，比如我们使用的 `Open`。你可以通过查阅 `API` 文档获得所有的快捷键列表。这个与我们自己定义的有什么区别呢？简单来说，我们完全可以自己定义一个 `tr("Ctrl+O")` 来实现快捷键。原因在于，这是 `Qt` 跨平台性的体现。比如 `PC` 键盘和 `Mac` 键盘是不一样的，一些键在 `PC` 键盘上有，而 `Mac` 键盘上可能并不存在，或者反之。使用 `QKeySequence` 类来添加快捷键，会根据平台的不同来定义相应的快捷键。

`setStatusTip()` 则实现了当用户鼠标滑过这个 `action` 时，会在主窗口下方的状态栏显示相应的提示。

后面的 `connect()` 函数，将这个 `QAction` 的 `triggered()` 信号与 `MainWindow` 类的 `open()` 函数连接起来。当用户点击了这个 `QAction` 时，会自动触发 `MainWindow` 的 `open()` 函数。这是我们之前已经了解过的。

下面的 `menuBar()`、`toolBar()` 和 `statusBar()` 三个是 `QMainWindow` 的函数，用于创建并返回菜单栏、工具栏和状态栏。我们可以从代码清楚地看出，我们向菜单栏添加了一个 `File` 菜单，并且把这个 `QAction` 对象添加到这个菜单；同时新增加了一个 `File` 工具栏，也把 `QAction` 对象添加到了这个工具栏。我们可以看到，在菜单中，这个对象被显示成一个菜单项，在工具栏变成了一个按钮。至于状态栏，则是出现在窗口最下方，用于显示动作对象的提示信息的。

至于 open()函数中的内容，我们会在后文介绍。这里可以运行一下，你会看到，触发这个动作，程序会弹出一个提示框。

下面是 Qt 4 版本的程序，具体非常类似，这里不再赘述。

```
// !!! Qt 4
// ===== mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void open();

private:
    QAction *openAction;
};

#endif // MAINWINDOW_H

// ===== mainwindow.cpp
#include <QAction>
#include <QMenuBar>
#include <QMessageBox>
#include <QToolBar>

#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    setWindowTitle(tr("Main Window"));

    openAction = new QAction(QIcon(":/images/doc-open"), tr("&Open..."), this);
    openAction->setShortcuts(QKeySequence::Open);
    openAction->setStatusTip(tr("Open an existing file"));
```

```

        connect(openAction, SIGNAL(triggered()), this, SLOT(open()));

        QMenu *file = menuBar()->addMenu(tr("&File"));
        file->addAction(openAction);

        QToolBar *toolBar = addToolBar(tr("&File"));
        toolBar->addAction(openAction);

        statusBar();
    }

MainWindow::~MainWindow()
{
}

void MainWindow::open()
{
    QMessageBox::information(this, tr("Information"), tr("Open"));
}

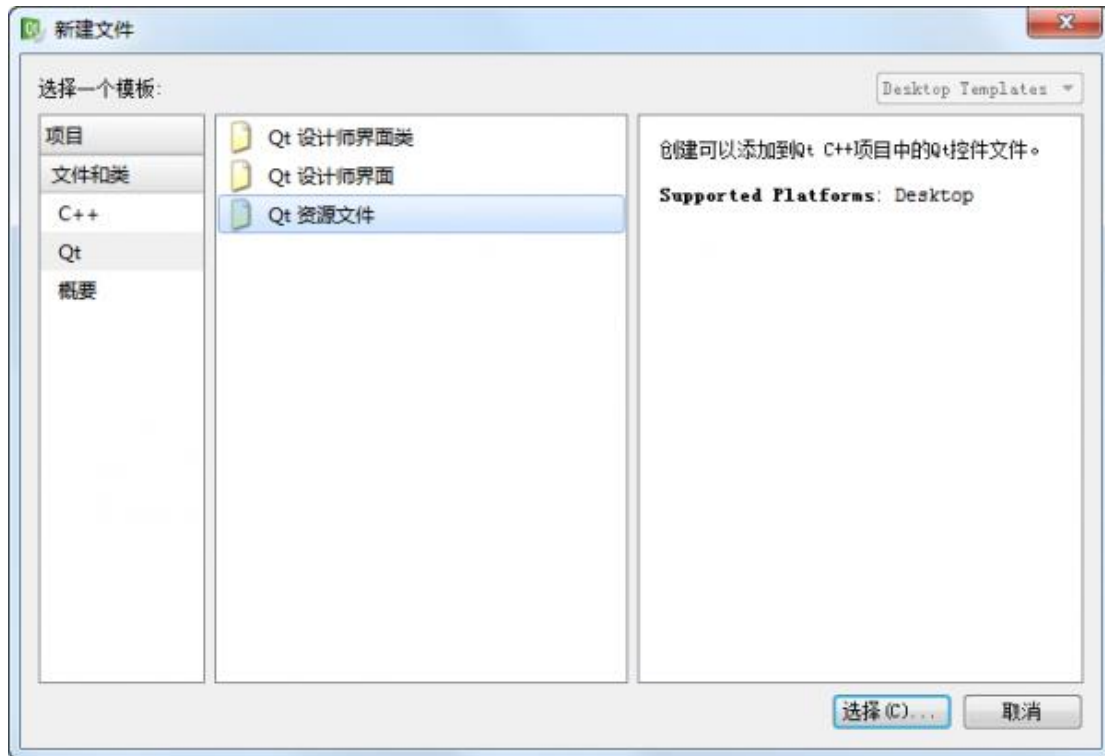
```

## 9 资源文件

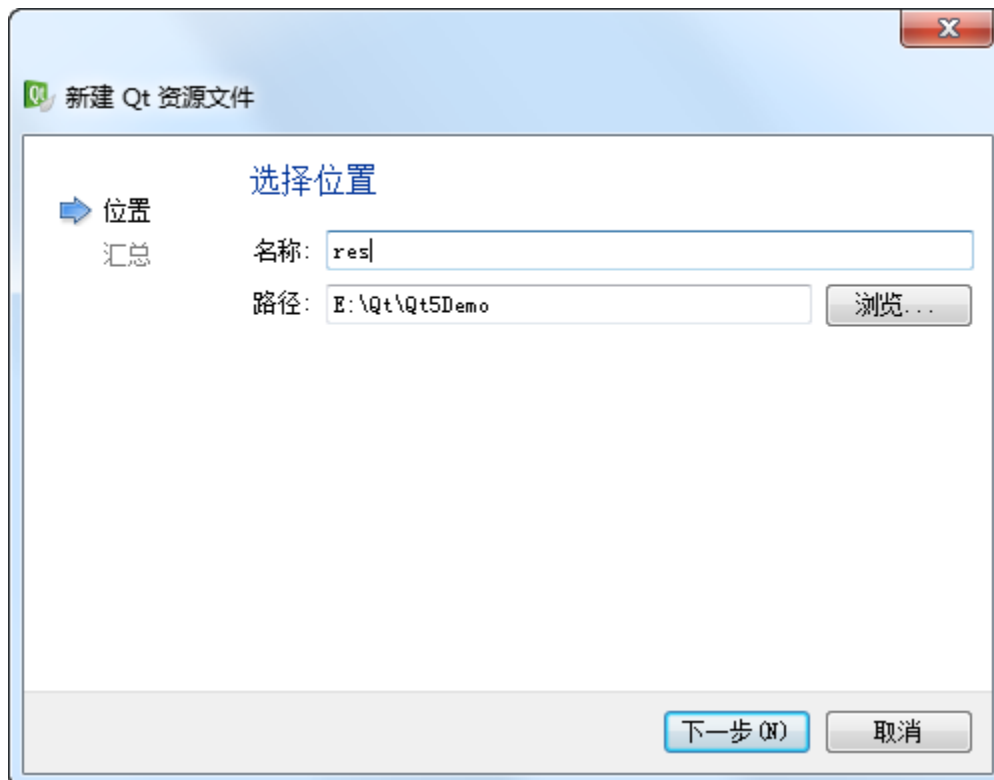
上一章节中我们介绍了如何使用 QAction 添加动作。其中，我们使用 QIcon 加载了一张 png 图片。那时候我们使用的是 Qt 资源文件。现在我们来仔细了解下 Qt 的资源系统。

Qt 资源系统是一个跨平台的资源机制，用于将程序运行时所需要的资源以二进制的形式存储于可执行文件内部。如果你的程序需要加载特定的资源（图标、文本翻译等），那么，将其放置在资源文件中，就再也不需要担心这些文件的丢失。也就是说，如果你将资源以资源文件形式存储，它是会编译到可执行文件内部。

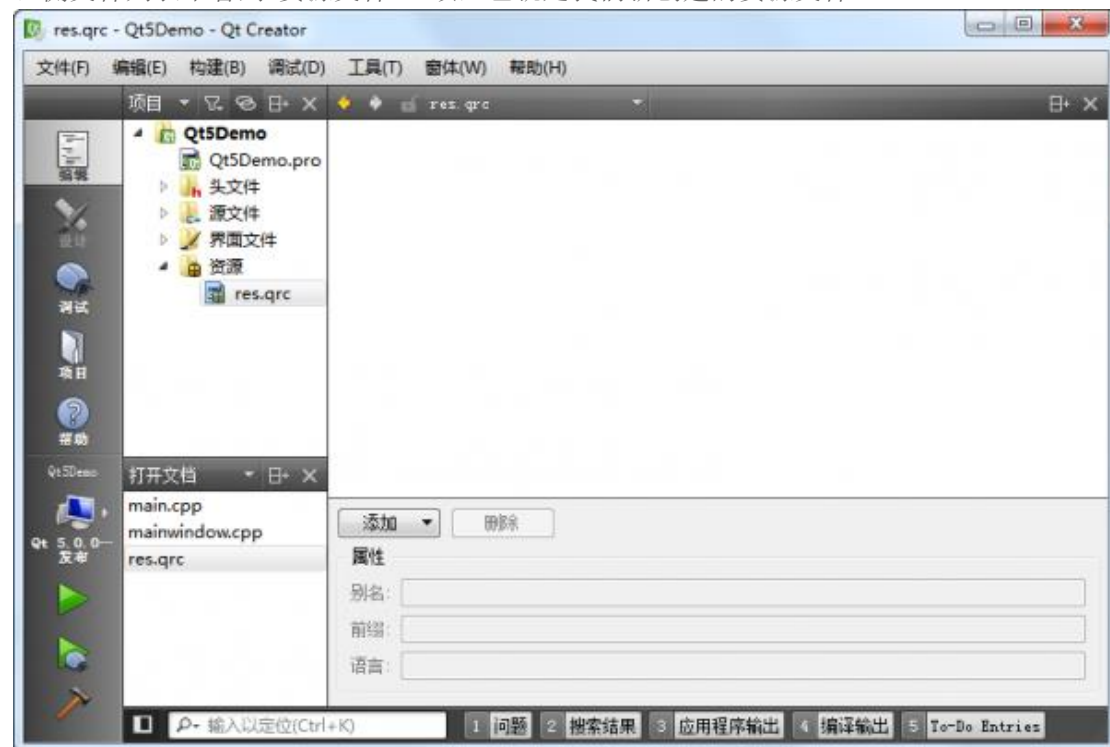
使用 Qt Creator 可以很方便地创建资源文件。我们可以在工程上点右键，选择“添加新文件...”，可以在 Qt 分类下找到“Qt 资源文件”：



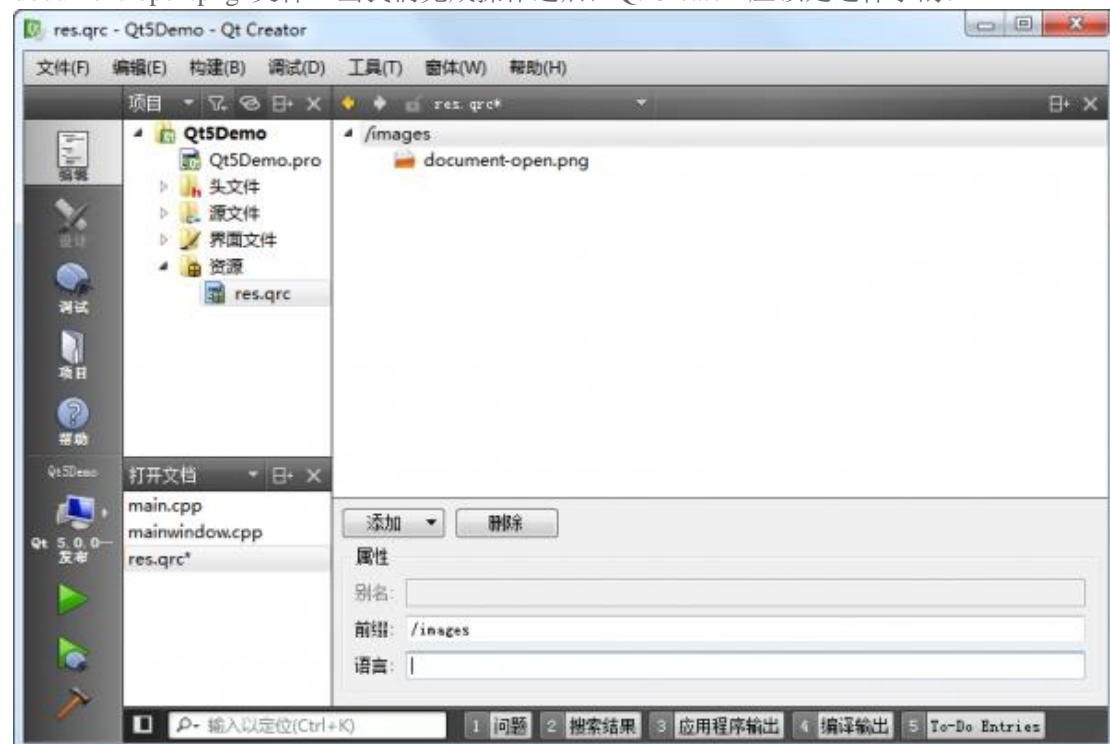
点击“选择...”按钮，打开“新建 Qt 资源文件”对话框。在这里我们输入资源文件的名字和路径：



点击下一步，选择所需要的版本控制系统，然后直接选择完成。我们可以在 Qt Creator 的左侧文件列表中看到“资源文件”一项，也就是我们新创建的资源文件：

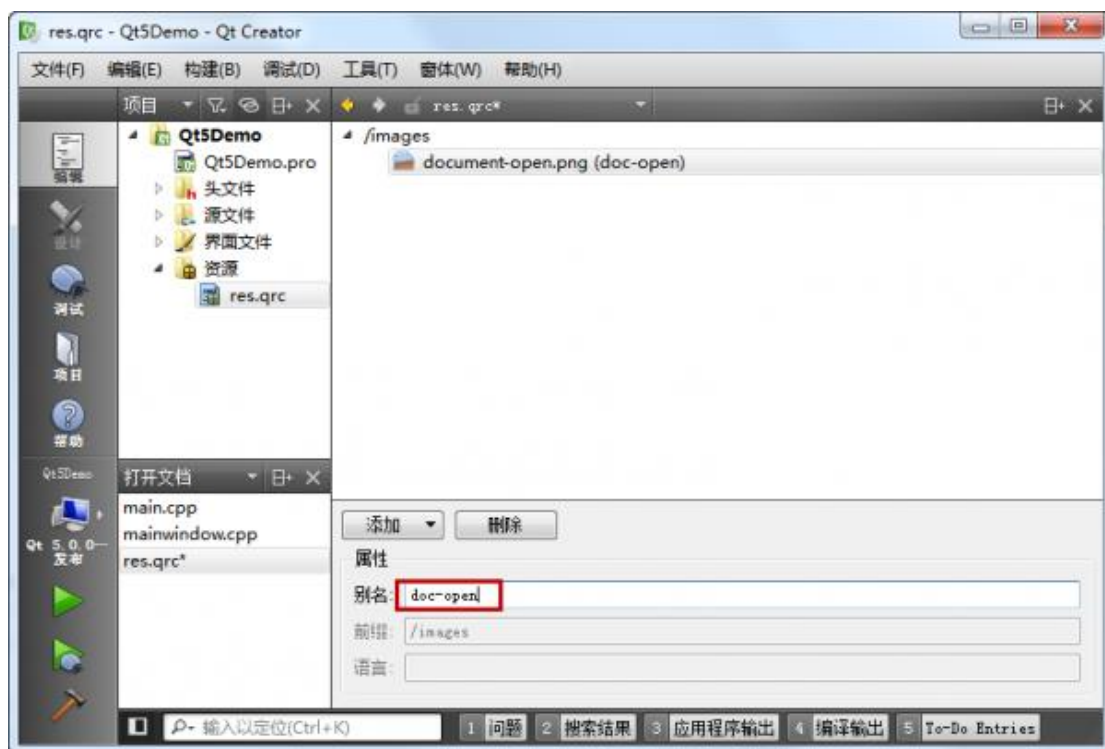


右侧的编辑区有个“添加”，我们首先需要添加前缀，比如我们将前缀取名为 images。然后选中这个前缀，继续点击添加文件，可以找到我们所需添加的文件。这里，我们选择 document-open.png 文件。当我们完成操作之后，Qt Creator 应该是这样子的：



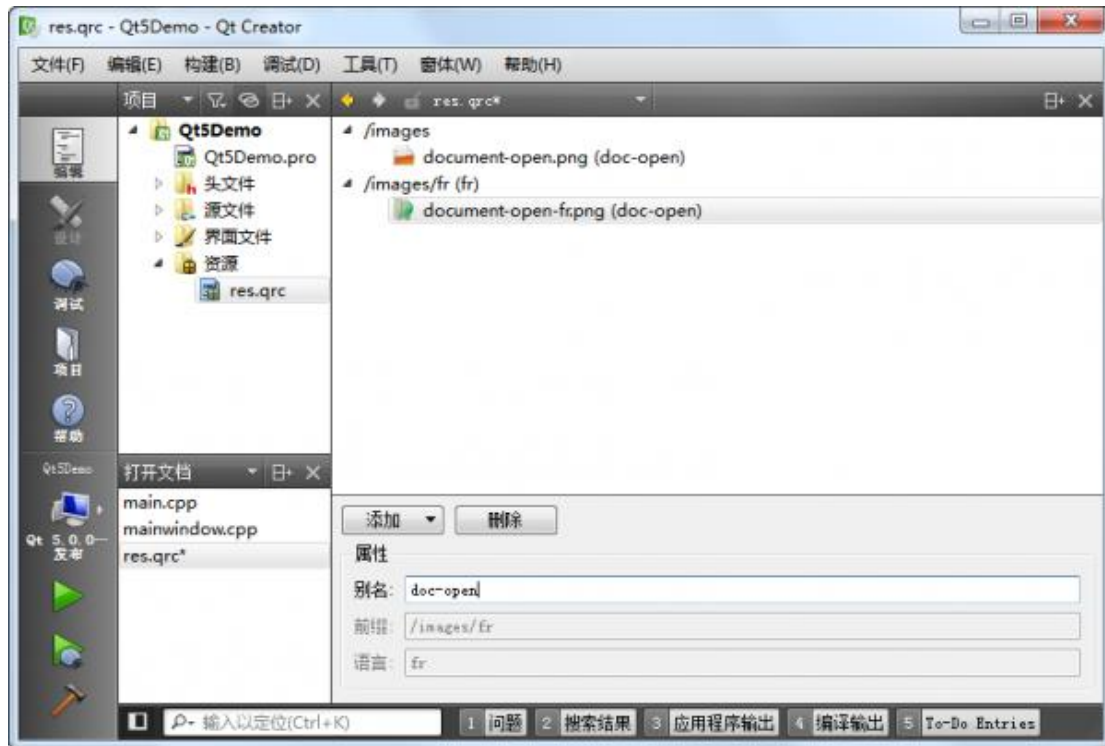
接下来，我们还可以添加另外的前缀或者另外的文件。这取决于你的需要。当我们添加完成之后，我们可以像前面一章讲解的那样，通过使用 `:` 开头的路径来找到这个文件。比如，我们的前缀是 `/images`，文件是 `document-open.png`，那么就可以使用 `/images/document-open.png` 找到这个文件。

这么做带来的一个问题是，如果以后我们要更改文件名，比如将 `document-open.png` 改成 `docopen.png`，那么，所有使用了这个名字的路径都需要修改。所以，更好的办法是，我们给这个文件去一个“别名”，以后就以这个别名来引用这个文件。具体做法是，选中这个文件，添加别名信息：



这样，我们可以直接使用 `/images/doc-open` 引用到这个资源，无需关心图片的真实文件名。

如果仔细观察，还会看到资源编辑窗口最下方有一个“语言”。这个可以对资源进行国际化。比如我们新建一个前缀，将语言设置为 `fr`，再添加一个文件 `document-open-fr.png`：



我们可以使用:/images/fr/doc-open 引用到 document-open-fr.png 这个文件。这个“语言”的作用是，如果 Qt 发现，本机的本地化信息是 fr 的话（QLocale::system().name()返回 fr\_FR），则使用:/images/fr/doc-open 这个图片；如果不是，则默认使用:/images/doc-open 这个。

如果我们使用文本编辑器打开 res.qrc 文件，就会看到一下内容：

```
<RCC>
  <qresource prefix="/images">
    <file alias="doc-open">document-open.png</file>
  </qresource>
  <qresource prefix="/images/fr" lang="fr">
    <file alias="doc-open">document-open-fr.png</file>
  </qresource>
</RCC>
```

我们可以对比一下，看看 Qt Creator 帮我们生成的是怎样的 qrc 文件。当我们编译工程之后，我们可以在构建目录中找到 qrc\_res.cpp 文件，这就是 Qt 将我们的资源编译成了 C++ 代码。

## 10 对象模型



标准 C++ 对象模型在运行时效率方面卓有成效，但是在某些特定问题域下的静态特性就显得捉襟见肘。GUI 界面需要同时具有运行时的效率以及更高级别的灵活性。为了解决这一问题，Qt “扩展”了标准 C++。所谓“扩展”，实际是在使用标准 C++ 编译器编译 Qt 源程序之前，Qt 先使用一个叫做 moc (Meta Object Compiler, 元对象编译器) 的工具，先对 Qt 源代码进行一次预处理 (注意，这个预处理与标准 C++ 的预处理有所不同。Qt 的 moc 预处理发生在标准 C++ 预处理器工作之前，并且 Qt 的 moc 预处理不是递归的。)，生成标准 C++ 源代码，然后再使用标准 C++ 编译器进行编译。如果你曾经为信号函数这样的语法感到奇怪 (现在我们已经编译过一些 Qt 程序，你应当注意到了，信号函数是不需要编写实现代码的，那怎么可以通过标准 C++ 的编译呢？)，这其实就是 moc 进行了处理之后的效果。

Qt 使用 moc，为标准 C++ 增加了一些特性：

- 信号槽机制，用于解决对象之间的通讯，这个我们已经了解过了，可以认为是 Qt 最明显的特性之一；
- 可查询，并且可设计的对象属性；
- 强大的事件机制以及事件过滤器；
- 基于上下文的字符串翻译机制 (国际化)，也就是 tr() 函数，我们简单地介绍过；
- 复杂的定时器实现，用于在事件驱动的 GUI 中嵌入能够精确控制的任务集成；
- 层次化的可查询的对象树，提供一种自然的方式管理对象关系。
- 智能指针 (QPointer)，在对象析构之后自动设为 0，防止野指针；
- 能够跨越库边界的动态转换机制。

通过继承 QObject 类，我们可以很方便地获得这些特性。当然，这些特性都是由 moc 帮助我们实现的。moc 其实实现的是一个叫做元对象系统 (meta-object system) 的机制。正如上面所说，这是一个标准 C++ 的扩展，使得标准 C++ 更适合于进行 GUI 编程。虽然利用模板可以达到类似的效果，但是 Qt 没有选择使用模板。按照 Qt 官方的说法，模板虽然是内置语言特性，但是其语法实在是复杂，并且由于 GUI 是动态的，利用静态的模板机制有时候很难处理。而自己使用 moc 生成代码更为灵活，虽然效率有些降低 (一个信号槽的调用大约相当于四个模板函数调用)，不过在现代计算机上，这点性能损耗实在是可以忽略。

在本节中，我们将主要介绍 Qt 的对象树。还记得我们前面在 MainWindow 的例子中看到了 parent 指针吗？现在我们就来解释这个 parent 到底是干什么的。

QObject 是以对象树的形式组织起来的。当你创建一个 QObject 对象时，会看到 QObject 的构造函数接收一个 QObject 指针作为参数，这个参数就是 parent，也就是父对象指针。这相当于，在创建 QObject 对象时，可以提供一个其父对象，我们创建的这个 QObject 对象会自动添加到其父对象的 children() 列表。当父对象析构的时候，这个列表中的所有对象也会被析构。（注意，这里的父对象并不是继承意义上的父类！）这种机制在 GUI 程序设计中相当有用。例如，一个按钮有一个 QShortcut（快捷键）对象作为其子对象。当我们删除按钮的时候，这个快捷键理应被删除。这是合理的。

QWidget 是能够在屏幕上显示的一切组件的父类。QWidget 继承自 QObject，因此也继承了这种对象树关系。一个孩子自动地成为父组件的一个子组件。因此，它会显示在父组件的坐标系统中，被父组件的边界剪裁。例如，当用户关闭一个对话框的时候，应用程序将其删除，那么，我们希望属于这个对话框的按钮、图标等应该一起被删除。事实就是如此，因为这些都是对话框的子组件。

当然，我们也可以自己删除子对象，它们会自动从其父对象列表中删除。比如，当我们删除了一个工具栏时，其所在的主窗口会自动将该工具栏从其子对象列表中删除，并且自动调整屏幕显示。

我们可以使用 QObject::dumpObjectTree() 和 QObject::dumpObjectInfo() 这两个函数进行这方面的调试。

Qt 引入对象树的概念，在一定程度上解决了内存问题。

当一个 QObject 对象在堆上创建的时候，Qt 会同时为其创建一个对象树。不过，对象树中对象的顺序是没有定义的。这意味着，销毁这些对象的顺序也是未定义的。Qt 保证的是，任何对象树中的 QObject 对象 delete 的时候，如果这个对象有 parent，则自动将其从 parent 的 children() 列表中删除；如果有孩子，则自动 delete 每一个孩子。Qt 保证没有 QObject 会被 delete 两次，这是由析构顺序决定的。

如果 QObject 在栈上创建，Qt 保持同样的行为。正常情况下，这也不会发生什么问题。来看下下面的代码片段：

```
{
    QWidget window;
    QPushButton quit("Quit", &window);
}
```

作为父组件的 window 和作为子组件的 quit 都是 QObject 的子类（事实上，它们都是 QWidget 的子类，而 QWidget 是 QObject 的子类）。这段代码是正确的，quit 的析构函数不会被调用两次，因为标准 C++（ISO/IEC 14882:2003）要求，局部对象的析构顺序应

该按照其创建顺序的相反过程。因此，这段代码在超出作用域时，会先调用 `quit` 的析构函数，将其从父对象 `window` 的子对象列表中删除，然后才会再调用 `window` 的析构函数。

但是，如果我们使用下面的代码：

```
{
    QPushButton quit("Quit");
    QWidget window;

    quit.setParent(&window);
}
```

情况又有所不同，析构顺序就有了问题。我们看到，在上面的代码中，作为父对象的 `window` 会首先被析构，因为它是最后一个创建的对象。在析构过程中，它会调用子对象列表中每一个对象的析构函数，也就是说，`quit` 此时就被析构了。然后，代码继续执行，在 `window` 析构之后，`quit` 也会被析构，因为 `quit` 也是一个局部变量，在超出作用域的时候当然也需要析构。但是，这时候已经是第二次调用 `quit` 的析构函数了，C++ 不允许调用两次析构函数，因此，程序崩溃了。

由此我们看到，Qt 的对象树机制虽然帮助我们在一定程度上解决了内存问题，但是也引入了一些值得注意的事情。这些细节在今后的开发过程中很可能时不时跳出来烦扰一下，所以，我们最好从开始就养成良好习惯，在 Qt 中，尽量在构造的时候就指定 `parent` 对象，并且大胆在堆上创建。

## 11 布局管理器

所谓 GUI 界面，归根结底，就是一堆组件的叠加。我们创建一个窗口，把按钮放上面，把图标放上面，这样就成了一个界面。在放置时，组件的位置尤其重要。我们必须指定组件放在哪里，以便窗口能够按照我们需要的方式进行渲染。这就涉及到组件定位的机制。Qt 提供了两种组件定位机制：绝对定位和布局定位。

顾名思义，绝对定位就是一种最原始的定位方法：给出这个组件的坐标和长宽值。这样，Qt 就知道该把组件放在哪里以及如何设置组件的大小。但是这样做带来的一个问题是，如果用户改变了窗口大小，比如点击最大化按钮或者使用鼠标拖动窗口边缘，采用绝对定位的组件是不会有响应的。这也很自然，因为你并没有告诉 Qt，在窗口变化时，组件是否需要更新自己以及如何更新。如果你需要让组件自动更新——这是很常见的需求，比如在最大化时，Word 总会把稿纸区放大，把工具栏拉长——就要自己编写相应的函数来响应这些变化。或者，还有更简单的方法：禁止用户改变窗口大小。但这总不是长远之计。

针对这种变化的需求，Qt 提供了另外一种机制——布局——来解决这个问题。你只要把组件放入某一种布局，布局由专门的布局管理器进行管理。当需要调整大小或者位置的时候，Qt 使用对应的布局管理器进行调整。下面来看一个例子：

```
// !!! Qt 5

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

    QWidget window;
    window.setWindowTitle("Enter your age");

    QSpinBox *spinBox = new QSpinBox(&window);
    QSlider *slider = new QSlider(Qt::Horizontal, &window);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);

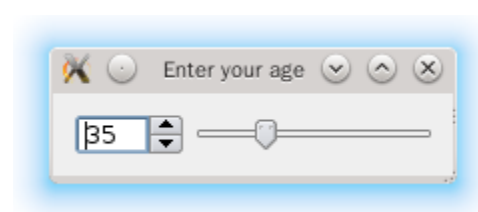
    QObject::connect(slider, &QSlider::valueChanged, spinBox,
&QSpinBox::setValue);
    void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
    QObject::connect(spinBox, spinBoxSignal, slider, &QSlider::setValue);
    spinBox->setValue(35);

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    window.setLayout(layout);

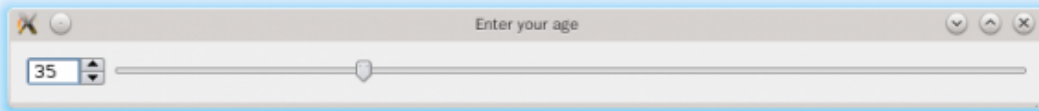
    window.show();

    return app.exec();
}
```

这段例子还是有些东西值得解释的。我们可以先来看看运行结果：



当我们拖动窗口时，可以看到组件自动有了变化：



我们在这段代码中引入了两个新的组件：**QSpinBox** 和 **QSlider**。**QSpinBox** 就是只能输入数字的输入框，并且带有上下箭头的步进按钮。**QSlider** 则是带有滑块的滑竿。我们可以从上面的截图中清楚地辨别出这两个组件。当我们创建了这两个组件的实例之后，我们使用 **setRange()** 函数设置其范围。既然我们的窗口标题是“Enter your age（输入你的年龄）”，那么把 **range**（范围）设置为 0 到 130 应该足够了。

有趣的部分在下面的 **connect()** 函数。我们已经清楚 **connect()** 函数的使用，因此我们写出

```
QObject::connect(slider, &QSlider::valueChanged, spinBox,  
&QSpinBox::setValue);
```

将 **slider** 的 **valueChanged()** 信号同 **spinBox** 的 **setValue()** 函数相连。这是我们熟悉的。但是，当我们直接写

```
QObject::connect(spinBox, &QSpinBox::valueChanged, slider,  
&QSlider::setValue);
```

的时候，编译器却会报错：

```
no matching function for call to 'QObject::connect(QSpinBox*&, <unresolved  
overloaded function type>, QSlider*&, void (QAbstractSlider::*)(int))'
```

这是怎么回事呢？从出错信息可以看出，编译器认为 **QSpinBox::valueChanged** 是一个 **overloaded** 的函数。我们看一下 **QSpinBox** 的文档发现，**QSpinBox** 的确有两个信号：

- **void valueChanged(int)**
- **void valueChanged(const QString &)**

当我们使用 **&QSpinBox::valueChanged** 取函数指针时，编译器不知道应该取哪一个函数（记住前面我们介绍过的，经过 **moc** 预处理后，**signal** 也是一个普通的函数。）的地址，因此报错。解决的方法很简单，编译器不是不能确定哪一个函数吗？那么我们就显式指定一个函数。方法就是，我们创建一个函数指针，这个函数指针参数指定为 **int**：

```
void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
```

然后我们将这个函数指针作为 **signal**，与 **QSlider** 的函数连接：

```
void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
```

这样便避免了编译错误。

仔细观察这两个 `connect()` 的作用，它们实际完成了一个双向的数据绑定。当然，对于 Qt 自己的信号函数，我们可以比较放心地使用。但是，如果是我们自己的信号，应当注意避免发生无限循环！

下面的代码，我们创建了一个 `QHBoxLayout` 对象。显然，这就是一个布局管理器。然后将这两个组件都添加到这个布局管理器，并且把该布局管理器设置为窗口的布局管理器。这些代码看起来都是顺理成章的，应该很容易明白。并且，布局管理器很聪明地做出了正确的行为：保持 `QSpinBox` 宽度不变，自动拉伸 `QSlider` 的宽度。

Qt 提供了几种布局管理器供我们选择：

- `QHBoxLayout`：按照水平方向从左到右布局；
- `QVBoxLayout`：按照竖直方向从上到下布局；
- `QGridLayout`：在一个网格中进行布局，类似于 HTML 的 table；
- `QFormLayout`：按照表格布局，每一行前面是一段文本，文本后面跟随一个组件（通常是输入框），类似 HTML 的 form；
- `QStackedLayout`：层叠的布局，允许我们将几个组件按照 Z 轴方向堆叠，可以形成向导那种一页一页的效果。

当然，我们也可以使用 Qt 4 来编译上面的代码，不过，正如大家应该想到的一样，我们必须把 `connect()` 函数修改一下：

```
// !!! Qt 4

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

    QWidget window;
    window.setWindowTitle("Enter your age");

    QSpinBox *spinBox = new QSpinBox(&window);
    QSlider *slider = new QSlider(Qt::Horizontal, &window);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);
```

```

QObject::connect(slider, SIGNAL(valueChanged(int)),
                 spinBox, SLOT(setValue(int)));
QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                 slider, SLOT(setValue(int)));
spinBox->setValue(35);

QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(spinBox);
layout->addWidget(slider);
window.setLayout(layout);

window.show();

return app.exec();
}

```

这里我们强调一下，上面的代码在 Qt 5 中同样可以编译通过。不过，我们减少了使用函数指针指定信号的步骤。也就是说，在 Qt 5 中，如果你想使用 overloaded 的 signal，有两种方式可供选择：

1. 使用 Qt 4 的 SIGNAL 和 SLOT 宏，因为这两个宏已经指定了参数信息，所以不存在这个问题；
2. 使用函数指针显式指定使用哪一个信号。

有时候，使用 Qt 4 的语法更简洁。但是需要注意的是，Qt 4 的语法是没有编译期错误检查的。这也是同 Qt 5 的信号槽新语法不同之处之一。

## 12 菜单栏、工具栏和状态栏

在之前的《[添加动作](#)》一文中，我们已经了解了，Qt 将用户与界面进行交互的元素抽象为一种“动作”，使用 QAction 类表示。QAction 可以添加到菜单上、工具栏上。期间，我们还详细介绍了一些细节问题，比如资源文件的使用、对象模型以及布局管理器。这一节则是详细介绍关于菜单栏、工具栏以及状态栏的相关内容。

我们假设窗口还是建立在 QMainWindow 类之上，这会让我们开发简单许多。当然，在实际开发过程中，QMainWindow 通常只作为“主窗口”，对话框窗口则更多地使用 QDialog 类。我们会在后面看到，QDialog 类会缺少一些 QMainWindow 类提供方便的函数，比如 menuBar() 以及 toolBar()。

下面还是回到《[添加动作](#)》一文中的代码片段：

```
openAction = new QAction(QIcon(":/images/doc-open"), tr("&Open..."), this);
openAction->setShortcuts(QKeySequence::Open);
openAction->setStatusTip(tr("Open an existing file"));
connect(openAction, &QAction::triggered, this, MainWindow::open);

QMenu *file = menuBar()->addMenu(tr("&File"));
file->addAction(openAction);

QToolBar *toolBar = addToolBar(tr("&File"));
toolBar->addAction(openAction);
```

我们看到，使用 `menuBar()` 函数，Qt 为我们创建了一个菜单栏。`menuBar()` 是 `QMainWindow` 提供的函数，因此你是不会在 `QWidget` 或者 `QDialog` 中找到它的。这个函数会返回窗口的菜单栏，如果没有菜单栏则会新创建一个。这也就解释了，为什么我们可以直接使用 `menuBar()` 函数的返回值，毕竟我们并没有创建一个菜单栏对象啊！原来，这就是 `menuBar()` 为我们创建好并且返回了的。

Qt 中，表示菜单的类是 `QMenuBar`（你应该已经想到这个名字了）。`QMenuBar` 代表的是窗口最上方的一条菜单栏。我们使用其 `addMenu()` 函数为其添加菜单。尽管我们只是提供了一个字符串作为参数，但是 Qt 为将其作为新创建的菜单的文本显示出来。至于 `&` 符号，我们已经解释过，这可以为菜单创建一个快捷键。当我们创建出来了菜单对象时，就可以把 `QAction` 添加到这个菜单上面，也就是 `addAction()` 函数的作用。

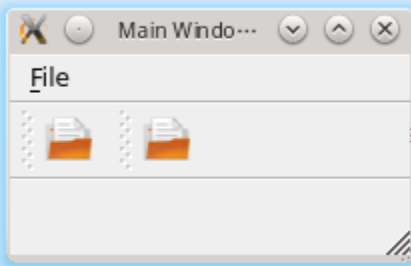
下面的 `QToolBar` 部分非常类似。顾名思义，`QToolBar` 就是工具栏。我们使用的是 `addToolBar()` 函数添加新的工具栏。为什么前面一个是 `menuBar()` 而现在的是 `addToolBar()` 呢？因为一个窗口只有一个菜单栏，但是却可能有多个工具栏。如果我们将代码修改一下：

```
QToolBar *toolBar = addToolBar(tr("&File"));
toolBar->addAction(openAction);

QToolBar *toolBar2 = addToolBar(tr("Tool Bar 2"));
toolBar2->addAction(openAction);
```

我们看到，现在有两个工具栏了：





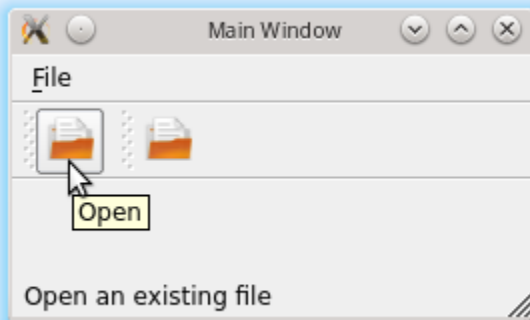
工具栏可以设置成固定的、浮动的等等，具体设置可以参考 Qt 文档。

前面我们说过，使用 `QAction::setStatusTip()` 可以设置该动作在状态栏上的提示文本。但我们现在把鼠标放在按钮上，是看不到这个提示文本的。原因很简单，我们没有添加一个状态栏。怎么添加呢？类似前面的 `QMainWindow::menuBar()`，`QMainWindow` 有一个 `statusBar()` 函数。让我们把这个函数添加上去：

```
QToolBar *toolBar2 = addToolBar(tr("Tool Bar 2"));
toolBar2->addAction(openAction);

statusBar();
```

然后编译运行一下：



我们添加了一个孤零零的 `statusBar()` 显得不伦不类，但是，同前面的 `menuBar()` 的实现类似，这个函数会返回一个 `QStatusBar` 对象，如果没有则先创建再返回。

`QStatusBar` 继承了 `QWidget`，因此，我们可以将其它任意 `QWidget` 子类添加到状态栏，从而实现类似 Photoshop 窗口底部那种有比例显示、有网格开关的复杂状态栏。有关 `QStatusBar` 的更多信息，请参考 Qt 文档。

对于没有这些函数的 `QDialog` 或者 `QWidget` 怎么办呢？要记得，`QToolBar` 以及 `QStatusBar` 都是 `QWidget` 的子类，因此我们就可以将其结合布局管理器添加到另外的 `QWidget` 上面。`QLayout` 布局提供了 `setMenuBar()` 函数，可以方便的添加菜单栏。具体细节还是详见文档。

至此，我们已经将组成窗口元素介绍过一遍。结合这些元素以及布局管理，我们就应该可以实现一个简单的通用的窗口。当我们完成窗口布局之后，我们就可以考虑向其中添加功能。这就是我们后面章节的内容。

## 13 对话框简介

对话框是 GUI 程序中不可或缺的组成部分。很多不能或者不适合放入主窗口的功能组件都必须放在对话框中设置。对话框通常会是一个顶层窗口，出现在程序最上层，用于实现短期任务或者简洁的用户交互。尽管 `Ribbon` 界面的出现在一定程度上减少了对话框的使用几率，但是，我们依然可以在最新版本的 `Office` 中发现不少对话框。因此，在可预见的未来，对话框会一直存在于我们的程序之中。

Qt 中使用 `QDialog` 类实现对话框。就像主窗口一样，我们通常会设计一个类继承 `QDialog`。`QDialog`（及其子类，以及所有 `Qt::Dialog` 类型的类）的对于其 `parent` 指针都有额外的解释：如果 `parent` 为 `NULL`，则该对话框会作为一个顶层窗口，否则则作为其父组件的子对话框（此时，其默认出现的位置是 `parent` 的中心）。顶层窗口与非顶层窗口的区别在于，顶层窗口在任务栏会有自己的位置，而非顶层窗口则会共享其父组件的位置。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("Main Window"));

    openAction = new QAction(QIcon(":/images/doc-open"), tr("&Open..."), this);
    openAction->setShortcuts(QKeySequence::Open);
    openAction->setStatusTip(tr("Open an existing file"));
    connect(openAction, &QAction::triggered, this, &MainWindow::open);

    QMenu *file = menuBar()->addMenu(tr("&File"));
    file->addAction(openAction);

    QToolBar *toolBar = addToolBar(tr("&File"));
    toolBar->addAction(openAction);
}
```

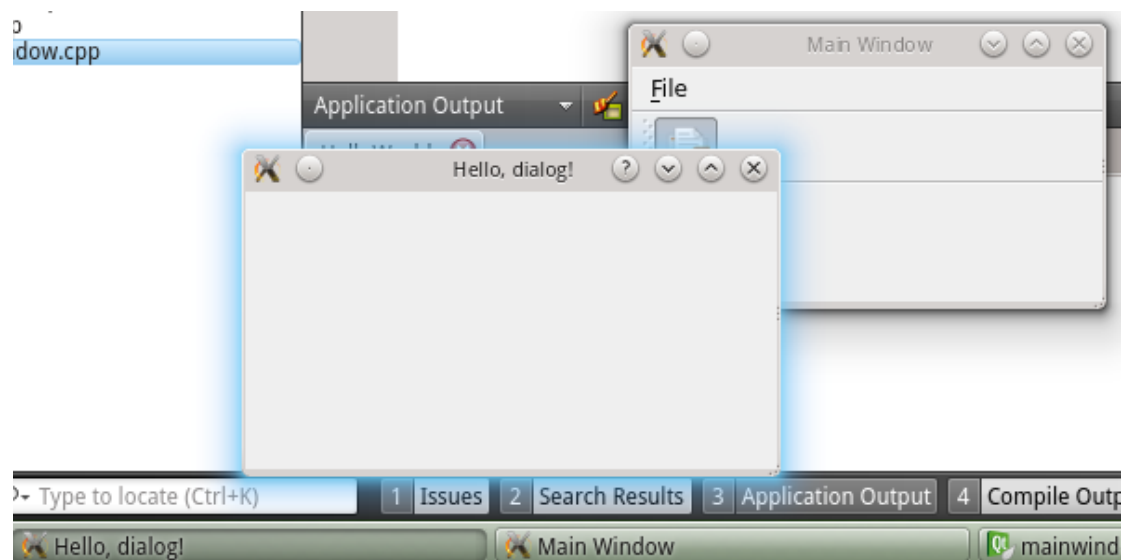
```

MainWindow::~MainWindow()
{
}

void MainWindow::open()
{
    QDialog dialog;
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.exec();
}

```

上面我们使用了前面的示例代码。注意看的是 `open()` 函数里面的内容。我们使用 `QDialog` 创建了一个对话框，设置其标题为“Hello, dialog!”，然后调用 `exec()` 将其显示出来。注意看的是任务栏的图标，由于我们没有设置对话框的 `parent` 指针，我们会看到在任务栏出现了对话框的位置：



我们修改一下 `open()` 函数的内容：

```

void MainWindow::open()
{
    QDialog dialog(this);
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.exec();
}

```

重新运行一下，对比一下就会看到 `parent` 指针的有无对 `QDialog` 实例的影响。

对话框分为模态对话框和非模态对话框。所谓模态对话框，就是会阻塞同一应用程序中其它窗口的输入。模态对话框很常见，比如“打开文件”功能。你可以尝试一下记事本的打开文件，当打开文件对话框出现时，我们是不能对除此对话框之外的窗口部分进行操作的。与此相反的是非模态对话框，例如查找对话框，我们可以在显示着查找对话框的同时，继续对记事本的内容进行编辑。

Qt 支持模态对话框和非模态对话框。其中，Qt 有两种级别的模态对话框：应用程序级别的模态和窗口级别的模态，默认是应用程序级别的模态。应用程序级别的模态是指，当该种模态的对话框出现时，用户必须首先对对话框进行交互，直到关闭对话框，然后才能访问程序中其他的窗口。窗口级别的模态是指，该模态仅仅阻塞与对话框关联的窗口，但是依然允许用户与程序中其它窗口交互。窗口级别的模态尤其适用于多窗口模式，更详细的讨论可以看[以前发表过的文章](#)。

Qt 使用 `QDialog::exec()`实现应用程序级别的模态对话框，使用 `QDialog::open()`实现窗口级别的模态对话框，使用 `QDialog::show()`实现非模态对话框。回顾一下我们的代码，在上面的示例中，我们调用了 `exec()`将对话框显示出来，因此这就是一个模态对话框。当对话框出现时，我们不能与主窗口进行任何交互，直到我们关闭了该对话框。

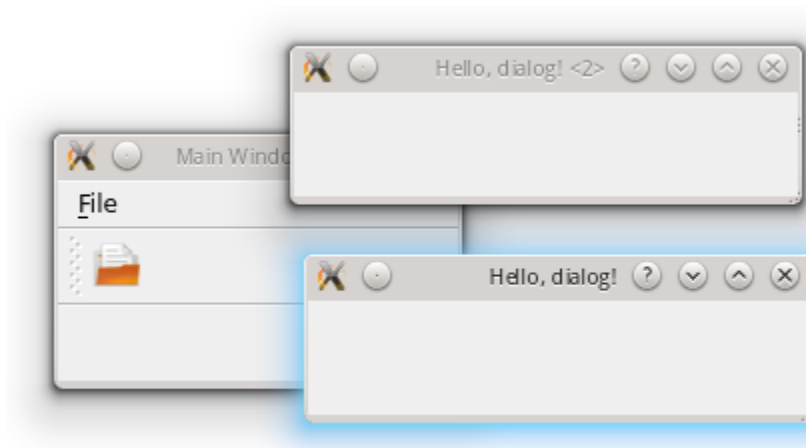
下面我们试着将 `exec()`修改为 `show()`，看看非模态对话框：

```
void MainWindow::open()
{
    QDialog dialog(this);
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.show();
}
```

是不是事与愿违？对话框竟然一闪而过！这是因为，`show()`函数不会阻塞当前线程，对话框会显示出来，然后函数立即返回，代码继续执行。注意，`dialog` 是建立在栈上的，`show()`函数返回，`MainWindow::open()`函数结束，`dialog` 超出作用域被析构，因此对话框消失了。知道了原因就好改了，我们将 `dialog` 改成堆上建立，当然就没有这个问题了：

```
void MainWindow::open()
{
    QDialog *dialog = new QDialog;
    dialog->setWindowTitle(tr("Hello, dialog!"));
    dialog->show();
}
```

对于一下这个非模态对话框和之前的模态对话框。我们在对话框出现的时候可以与主窗口交互，因此我们可以建立多个相同的对话框：



如果你足够细心，应该发现上面的代码是有问题的：dialog 存在内存泄露！dialog 使用 new 在堆上分配空间，却一直没有 delete。解决方案也很简单：将 MainWindow 的指针赋给 dialog 即可。还记得我们前面说过的 Qt 的对象系统吗？

不过，这样做有一个问题：如果我们的对话框不是在一个界面类中出现呢？由于 QWidget 的 parent 必须是 QWidget 指针，那就限制了我们不能将一个普通的 C++ 类指针传给 Qt 对话框。另外，如果对内存占用有严格限制的话，当我们将主窗口作为 parent 时，主窗口不关闭，对话框就不会被销毁，所以会一直占用内存。在这种情景下，我们可以设置 dialog 的 WindowAttribute：

```
void MainWindow::open()
{
    QDialog *dialog = new QDialog;
    dialog->setAttribute(Qt::WA_DeleteOnClose);
    dialog->setWindowTitle(tr("Hello, dialog!"));
    dialog->show();
}
```

setAttribute()函数设置对话框关闭时，自动销毁对话框。另外，QObject 还有一个 deleteLater()函数，该函数会在当前事件循环结束时销毁该对话框（具体到这里，需要使用 exec()开始一个新的事件循环）。关于事件循环，我们会在后面的文章中详细说明。

## 14 对话框数据传递

对话框的出现用于完成一个简单的或者是短期的任务。对话框与主窗口之间的数据交互相当重要。本节将讲解如何在对话框和主窗口之间进行数据交互。按照前文的讲解，对话框分为模态和非模态两种。我们也将以这两种为例，分别进行阐述。

模态对话框使用了 `exec()` 函数将其显示出来。`exec()` 函数的真正含义是开启一个新的事件循环（我们会在后面的章节中详细介绍有关事件的概念）。所谓事件循环，可以理解成一个无限循环。Qt 在开启了事件循环之后，系统发出的各种事件才能够被程序监听到。这个事件循环相当于一种轮询的作用。既然是无限循环，当然在开启了事件循环的地方，代码就会被阻塞，后面的语句也就不会被执行到。因此，对于使用了 `exec()` 显示的模态对话框，我们可以在 `exec()` 函数之后直接从对话框的对象获取到数据值。

看一下下面的代码：

```
void MainWindow::open()
{
    QDialog dialog(this);
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.exec();
    qDebug() << dialog.result();
}
```

上面的代码中，我们使用 `exec()` 显示一个模态对话框。最后一行代码，`qDebug()` 类似于 `std::cout` 或者 Java 的 `System.out.println()` 语句，将后面的信息输出到标准输出，一般就是控制台。使用 `qDebug()` 需要引入头文件。在 `exec()` 函数之后，我们直接可以获取到 `dialog` 的数据值。注意，`exec()` 开始了一个事件循环，代码被阻塞到这里。由于 `exec()` 函数没有返回，因此下面的 `result()` 函数也就不会被执行。直到对话框关闭，`exec()` 函数返回，此时，我们就可以取得对话框的数据。

需要注意的一点是，如果我们设置 `dialog` 的属性为 `WA_DeleteOnClose`，那么当对话框关闭时，对象被销毁，我们就不能使用这种办法获取数据了。在这种情况下，我们可以考虑使用 `parent` 指针的方式构建对话框，避免设置 `WA_DeleteOnClose` 属性；或者是利用另外的方式。

实际上，`QDialog::exec()` 是有返回值的，其返回值是 `QDialog::Accepted` 或者 `QDialog::Rejected`。一般我们会使用类似下面的代码：

```
void MainWindow::open()
{
    QDialog dialog(this);
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.exec();
    qDebug() << dialog.result();
}
```

来判断对话框的返回值，也就是用户是点击了“确定”还是“取消”。更多细节请参考 `QDialog` 文档。

模态对话框相对简单，如果是非模态对话框，`QDialog::show()`函数会立即返回，如果我们也这么写，就不可能取得用户输入的数据。因为 `show()` 函数不会阻塞主线程，`show()` 立即返回，用户还没有来得及输入，就要执行后面的代码，当然是不会有正确结果的。那么我们就应该换一种思路获取数据，那就是使用信号槽机制。

由于非模态对话框在关闭时可以调用 `QDialog::accept()` 或者 `QDialog::reject()` 或者更通用的 `QDialog::done()` 函数，所以我们可以在这里发出信号。另外，如果找不到合适的信号发出点，我们可以重写 `QDialog::closeEvent()` 函数，在这里发出信号。在需要接收数据的窗口（这里是主窗口）连接到这个信号即可。类似的代码片段如下所示：

```
//!!! Qt 5
// in dialog:
void UserAgeDialog::accept()
{
    emit userAgeChanged(newAge); // newAge is an int
    QDialog::accept();
}

// in main window:
void MainWindow::showUserAgeDialog()
{
    UserAgeDialog *dialog = new UserAgeDialog(this);
    connect(dialog, &UserAgeDialog::userAgeChanged, this,
    &MainWindow::setUserAge);
    dialog->show();
}

// ...

void MainWindow::setUserAge(int age)
{
    userAge = age;
}
```

上面的代码很简单，这里不再赘述。另外，上述代码的 Qt 4 版本也应该可以很容易地实现。

不要担心如果对话框关闭，是不是还能获取到数据。因为 Qt 信号槽的机制保证，在槽函数在调用的时候，我们始终可以使用 `sender()` 函数获取到 signal 的发出者。关于 `sender()` 函数，可以在文档中找到更多的介绍。顺便说一句，`sender()` 函数的存在使我们可以利用这个函数，来实现一个只能打开一个的非模态对话框（方法就是在对话框打开时在一个对话框映射表中记录下标记，在对话框关闭时利用 `sender()` 函数判断是不是该对话框，然后从映射表中将其删除）。

## 15 标准对话框 QMessageBox

所谓标准对话框，是 Qt 内置的一系列对话框，用于简化开发。事实上，有很多对话框都是通用的，比如打开文件、设置颜色、打印设置等。这些对话框在所有程序中几乎相同，因此没有必要在每一个程序中都自己实现这么一个对话框。

Qt 的内置对话框大致分为以下几类：

- QColorDialog: 选择颜色；
- QFileDialog: 选择文件或者目录；
- QFontDialog: 选择字体；
- QInputDialog: 允许用户输入一个值，并将其值返回；
- QMessageBox: 模态对话框，用于显示信息、询问问题等；
- QPageSetupDialog: 为打印机提供纸张相关的选项；
- QPrintDialog: 打印机配置；
- QPrintPreviewDialog: 打印预览；
- QProgressDialog: 显示操作过程。

这里我们简单地介绍一下标准对话框 QMessageBox 的使用。在前面有了关于对话框的基础之上，应该可以结合文档很轻松地学习如何使用 Qt 的标准对话框。其它种类的标准对话框，我们将在后面的章节中再一一介绍。

QMessageBox 用于显示消息提示。我们一般会使用其提供的几个 static 函数：

- void about(QWidget \* parent, const QString & title, const QString & text): 显示关于对话框。这是一个最简单的对话框，其标题是 title，内容是 text，父窗口是 parent。对话框只有一个 OK 按钮。
- void aboutQt(QWidget \* parent, const QString & title = QString()): 显示关于 Qt 对话框。该对话框用于显示有关 Qt 的信息。



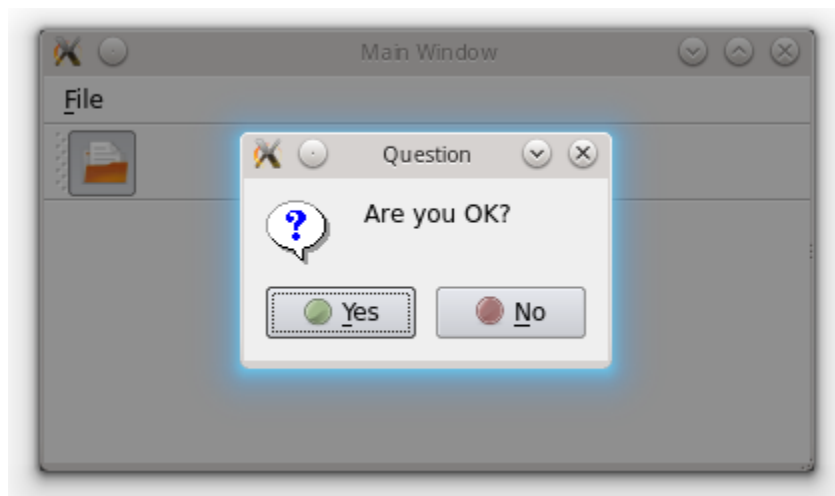
- `StandardButton critical(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)`: 显示严重错误对话框。这个对话框将显示一个红色的错误符号。我们可以通过 `buttons` 参数指明其显示的按钮。默认情况下只有一个 `Ok` 按钮，我们可以使用 `StandardButtons` 类型指定多种按钮。
- `StandardButton information(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)`: `QMessageBox::information()`函数与 `QMessageBox::critical()`类似，不同之处在于这个对话框提供一个普通信息图标。
- `StandardButton question(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = StandardButtons( Yes | No ), StandardButton defaultButton = NoButton)`: `QMessageBox::question()`函数与 `QMessageBox::critical()`类似，不同之处在于这个对话框提供一个问号图标，并且其显示的按钮是“是”和“否”两个。
- `StandardButton warning(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)`: `QMessageBox::warning()`函数与 `QMessageBox::critical()`类似，不同之处在于这个对话框提供一个黄色叹号图标。

我们可以通过下面的代码来演示下如何使用 `QMessageBox`。

```
if (QMessageBox::Yes == QMessageBox::question(this,
                                             tr("Question"),
                                             tr("Are you OK?"),
                                             QMessageBox::Yes | QMessageBox::No,
                                             QMessageBox::Yes)) {
    QMessageBox::information(this, tr("Hmmm..."), tr("I'm glad to hear that!"));
} else {
    QMessageBox::information(this, tr("Hmmm..."), tr("I'm sorry!"));
}
```

我们使用 `QMessageBox::question()`来询问一个问题。这个对话框的父窗口是 `this`，也就是我们的 `MainWindow`（或者其他 `QWidget` 指针）。`QMessageBox` 是 `QDialog` 的子类，这意味着它的初始显示位置将会是在 `parent` 窗口的中央（我们在前面的章节中提到过这一点）。第二个参数是对话框的标题。第三个参数是我们想要显示的内容。这里就是我们需要询问的文字。下面，我们使用或运算符（`|`）指定对话框应该出现的按钮。这里我们希望是一个 `Yes` 和一个 `No`。最后一个参数指定默认选择的按钮。这个函数有一个返回值，用于确定用户点击的是哪一个按钮。按照我们的写法，应该很容易的看出，这是一个模态对话框，

因此我们可以直接获取其返回值。如果返回值是 Yes，也就是说用户点击了 Yes 按钮，我们显示一个普通消息对话框，显示“I’m glad to hear that!”，否则则显示“I’m sorry!”。运行一下我们的程序片段，就可以看到其中的不同：

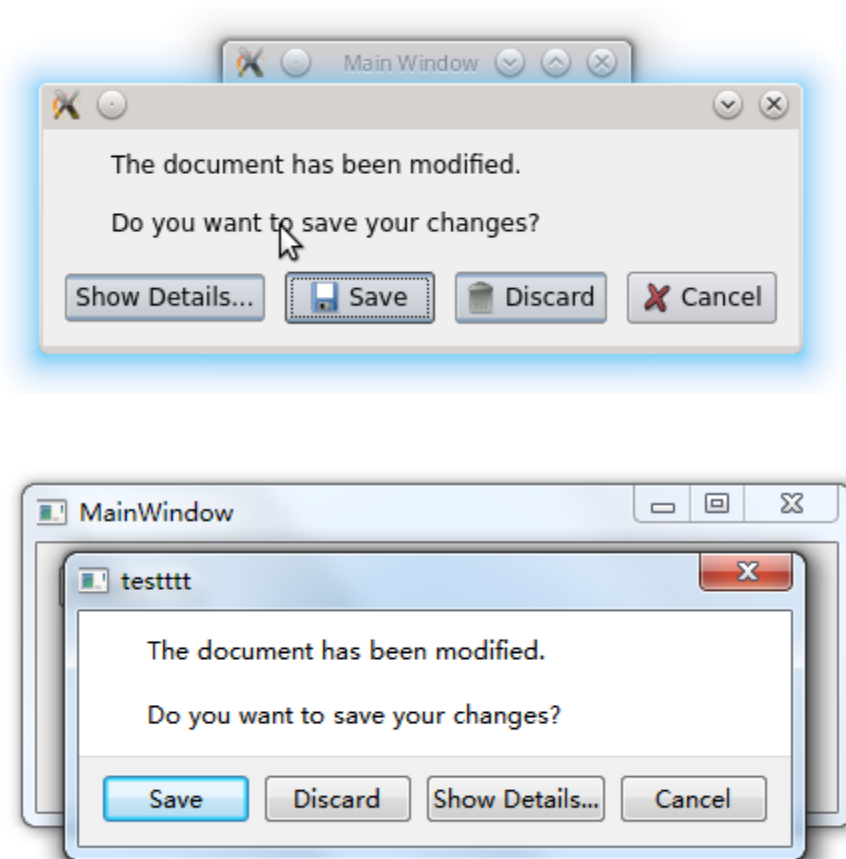


QMessageBox 类的 static 函数优点是方便使用，缺点也很明显：非常不灵活。我们只能使用简单的几种形式。为了能够定制 QMessageBox 细节，我们必须使用 QMessageBox 的属性设置 API。如果我们希望制作一个询问是否保存的对话框，我们可以使用如下的代码：

```
QMessageBox msgBox;
msgBox.setText(tr("The document has been modified.));
msgBox.setInformativeText(tr("Do you want to save your changes?));
msgBox.setDetailedText(tr("Differences here...));
msgBox.setStandardButtons(QMessageBox::Save
                          | QMessageBox::Discard
                          | QMessageBox::Cancel);
msgBox.setDefaultButton(QMessageBox::Save);
int ret = msgBox.exec();
switch (ret) {
case QMessageBox::Save:
    qDebug() << "Save document!";
    break;
case QMessageBox::Discard:
    qDebug() << "Discard changes!";
    break;
case QMessageBox::Cancel:
    qDebug() << "Close document!";
    break;
}
```

`msgBox` 是一个建立在栈上的 `QMessageBox` 实例。我们设置其主要文本信息为“The document has been modified.”，`informativeText` 则是会在对话框中显示的简单说明文字。下面我们使用了一个 `detailedText`，也就是详细信息，当我们点击了详细信息按钮时，对话框可以自动显示更多信息。我们自己定义的对话框的按钮有三个：保存、丢弃和取消。然后我们使用了 `exec()` 使其成为一个模态对话框，根据其返回值进行相应的操作。

同时在 KDE 和 Windows 7 上编译运行一下上面的代码，我们可以看到一些区别：



除去对话框样式，我们值得注意的是 `QMessageBox` 下方按钮的排列顺序。KDE 上是 Show Details...、Save、Discard 和 Cancel；而 Windows 7 上则是 Save、Discard、Show Details... 和 Cancel。我们并没有指定按钮的顺序，Qt 已经帮我们按照不同平台的使用习惯对其进行了调整。这一点在 Mac OS 上也会有相应的体现。对于一个普通的 `QDialog` 而言，Qt 使用的是 `QDialogButtonBox` 这个类来实现不同平台的对话框按钮顺序的显示的。更多细节请参考这个类的文档。

## 16 深入 Qt5 信号槽新语法

在前面的章节（[信号槽](#)和[自定义信号槽](#)）中，我们详细介绍了有关 Qt 5 的信号槽新语法。由于这次改动很大，许多以前看起来不是问题的问题接踵而来，因此，我们用单独的一章重新介绍一些 Qt 5 的信号槽新语法。

### 基本用法

Qt 5 引入了信号槽的新语法：使用函数指针能够获得编译期的类型检查。使用我们在[自定义信号槽](#)中设计的 Newspaper 类，我们来看看其基本语法：

```
//!!! Qt5
#include <QObject>

////////// newspaper.h
class Newspaper : public QObject
{
    Q_OBJECT
public:
    Newspaper(const QString & name) :
        m_name(name)
    {
    }

    void send() const
    {
        emit newPaper(m_name);
    }

signals:
    void newPaper(const QString &name) const;

private:
    QString m_name;
};

////////// reader.h
#include <QObject>
#include <QDebug>

class Reader : public QObject
{
```

```

    Q_OBJECT
public:
    Reader() {}

    void receiveNewspaper(const QString & name) const
    {
        qDebug() << "Receives Newspaper: " << name;
    }
};

////////// main.cpp
#include <QCoreApplication>

#include "newspaper.h"
#include "reader.h"

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    Newspaper newspaper("Newspaper A");
    Reader reader;
    QObject::connect(&newspaper, &Newspaper::newPaper,
                    &reader,    &Reader::receiveNewspaper);
    newspaper.send();

    return app.exec();
}

```

在 `main()` 函数中，我们使用 `connect()` 函数将 `newspaper` 对象的 `newPaper()` 信号与 `reader` 对象的 `receiveNewspaper()` 槽函数联系起来。当 `newspaper` 发出这个信号时，`reader` 相应的槽函数就会自动被调用。这里我们使用了取址操作符，取到 `Newspaper::newPaper()` 信号的地址，同样类似的取到了 `Reader::receiveNewspaper()` 函数地址。编译器能够利用这两个地址，在编译期对这个连接操作进行检查，如果有个任何错误（包括对象没有这个信号，或者信号参数不匹配等），编译时就会发现。

## 有重载的信号

如果信号有重载，比如我们向 `Newspaper` 类增加一个新的信号：

```
void newPaper(const QString &name, const QDate &date);
```

此时如果还是按照前面的写法，编译器会报出一个错误：由于这个函数（注意，信号实际也是一个普通的函数）有重载，因此不能用一个取址操作符获取其地址。回想一下 Qt 4 中的处理。在 Qt 4 中，我们使用 SIGNAL 和 SLOT 两个宏来连接信号槽。如果有一个带有两个参数的信号，像上面那种，那么，我们就可以使用下面的代码：

```
QObject::connect(&newspaper, SIGNAL(newPaper(QString, QDate)),
                &reader,     SLOT(receiveNewspaper(QString, QDate)));
```

注意，我们临时增加了一个 receiveNewspaper() 函数的重载，以便支持两个参数的信号。在 Qt 4 中不存在我们所说的错误，因为 Qt 4 的信号槽连接是带有参数的。因此，Qt 能够自己判断究竟是哪一个信号对应了哪一个槽。

对此，我们也给出了一个解决方案，使用一个函数指针来指明到底是哪一个信号：

```
void (Newspaper:: *newPaperNameDate)(const QString &, const QDate &) =
&Newspaper::newPaper;
QObject::connect(&newspaper, newPaperNameDate,
                &reader,     &Reader::receiveNewspaper);
```

这样，我们使用了函数指针 newPaperNameDate 声明一个带有 QString 和 QDate 两个参数，返回值是 void 的函数，将该函数作为信号，与 Reader::receiveNewspaper() 槽连接起来。这样，我们就回避了之前编译器的错误。归根结底，这个错误是因为函数重载，编译器不知道要取哪一个函数的地址，而我们显式指明一个函数就可以了。

如果你觉得这种写法很难看，想像前面一样写成一行，当然也是由解决方法的：

```
QObject::connect(&newspaper,
                (void (Newspaper:: *) (const QString &, const QDate
&))&Newspaper::newPaper,
                &reader,
                &Reader::receiveNewspaper);
```

这是一种换汤不换药的做法：我们只是声明了一个匿名的函数指针，而之前我们的函数指针是有名字的。不过，我们并不推荐这样写，而是希望以下的写法：

```
QObject::connect(&newspaper,
                static_cast<void (Newspaper:: *) (const QString &, const QDate
&)>(&Newspaper::newPaper),
                &reader,
                &Reader::receiveNewspaper);
```

对比上面两种写法。第一个使用的是 C 风格的强制类型转换。此时，如果你改变了信号的类型，那么你就会有一个潜在的运行时错误。例如，如果我们把(const QString &, const QDate &)两个参数修改成(const QDate &, const QString &), C 风格的强制类型转换就会失败，并且这个错误只能在运行时发现。而第二种则是 C++ 推荐的风格，当参数类型改变时，编译器会检测到这个错误。

注意，这里我们只是强调了函数参数的问题。如果前面的对象都错了呢？比如，我们写的 newspaper 对象并不是一个 Newspaper，而是 Newspaper2？此时，编译器会直接失败，因为 connect()函数会去寻找 sender->\*signal，如果这两个参数不满足，则会直接报错。

## 带有默认参数的槽函数

Qt 允许信号和槽的参数数目不一致：槽函数的参数数目要比信号参数少。这是因为，我们信号参数实际是作为一种返回值。正如普通的函数调用一样，我们可以选择忽略函数返回值，但是不能使用一个并不存在的返回值。如果槽函数的参数数目比信号的多，在槽函数中就使用到这些参数的时候，实际这些参数并不存在（因为信号参数比槽的少，因此并没有传过来），函数就会报错。这种情况往往有两个原因：一是槽参数就是比信号少，此时我们可以像前面那种写法直接连接。另外一个原因是，信号参数带有默认值。比如

```
void QPushButton::clicked(bool checked = false)
```

就是这种情况。

然而，有一种情况，槽函数的参数可以比信号的多，那就是槽函数的参数带有默认值。比如，我们的 Newspaper 和 Reader 有下面的代码：

```
// Newspaper
signals:
    void newPaper(const QString &name);
// Reader
void receiveNewspaper(const QString &name, const QDate &date =
QDate::currentDate());
```

虽然 Reader::receiveNewspaper()的参数数目比 Newspaper::newPaper()多，但是由于 Reader::receiveNewspaper()后面一个参数带有默认值，所以该参数不是必须提供的。但是，如果你按照前面的写法，比如如下的代码：

```
QObject::connect(&newspaper,
                static_cast<void (Newspaper::*)(const QString
&)>(&Newspaper::newPaper),
                &reader,
```

```
static_cast<void (Reader::*)(const QString &, const QDate &
=QDate::currentDate())>(&Reader::receiveNewspaper));
```

你会得到一个断言错误：

The slot requires more arguments than the signal provides.

我们不能在函数指针中使用函数参数的默认值。这是 C++ 语言的限制：参数默认值只能使用在直接地函数调用中。当使用函数指针取其地址的时候，默认参数是不可见的！

当然，此时你可以选择 Qt 4 的连接语法。如果你还是想使用 Qt 5 的新语法，目前的办法只有一个：Lambda 表达式。不要担心你的编译器不支持 Lambda 表达式，因为在你使用 Qt 5 的时候，能够支持 Qt 5 的编译器都是支持 Lambda 表达式的。于是，我们的代码就变成了：

```
QObject::connect(&newspaper,
                 static_cast<void (Newspaper::*)(const QString
&)>(&Newspaper::newPaper),
                 [=](const QString &name) { /* Your code here. */ });
```

## 17 文件对话框

在前面的章节中，我们讨论了 Qt 标准对话框 QMessageBox 的使用。所谓标准对话框，其实也就是一个普通的对话框。因此，我们同样可以将 QDialog 所提供的其它特性应用到这种标准对话框上面。今天，我们继续讨论另外一个标准对话框：QFileDialog，也就是文件对话框。在本节中，我们将尝试编写一个简单的文本文件编辑器，我们将使用 QFileDialog 来打开一个文本文件，并将修改过的文件保存到硬盘。这或许是我们在本系列中所提供的第一个带有实际功能的实例。

首先，我们需要创建一个带有文本编辑功能的窗口。借用我们前面的程序代码，应该可以很方便地完成：

```
openAction = new QAction(QIcon(":/images/file-open"), tr("&Open..."), this);
openAction->setShortcuts(QKeySequence::Open);
openAction->setStatusTip(tr("Open an existing file"));

saveAction = new QAction(QIcon(":/images/file-save"), tr("&Save..."), this);
saveAction->setShortcuts(QKeySequence::Save);
saveAction->setStatusTip(tr("Save a new file"));
```



```

QMenu *file = menuBar()->addMenu(tr("&File"));
file->addAction(openAction);
file->addAction(saveAction);

QToolBar *toolBar = addToolBar(tr("&File"));
toolBar->addAction(openAction);
toolBar->addAction(saveAction);

textEdit = new QTextEdit(this);
setCentralWidget(textEdit);

```

我们在菜单和工具栏添加了两个动作：打开和保存。接下来是一个 `QTextEdit` 类，这个类用于显示富文本文件。也就是说，它不仅仅用于显示文本，还可以显示图片、表格等等。不过，我们现在只用它显示纯文本文件。`QMainWindow` 有一个 `setCentralWidget()` 函数，可以将一个组件作为窗口的中心组件，放在窗口中央显示区。显然，在一个文本编辑器中，文本编辑区就是这个中心组件，因此我们将 `QTextEdit` 作为这种组件。

我们使用 `connect()` 函数，为这两个 `QAction` 对象添加响应的动作：

```

/// !!!Qt5
connect(openAction, &QAction::triggered, this, &MainWindow::openFile);
connect(saveAction, &QAction::triggered, this, &MainWindow::saveFile);

/// !!!Qt4
connect(openAction, SIGNAL(triggered()), this, SLOT(openFile()));
connect(saveAction, SIGNAL(triggered()), this, SLOT(saveFile()));

```

这些应该都不是问题。我们应该能够很清楚这些代码的含义。下面是最主要的 `openFile()` 和 `saveFile()` 这两个函数的代码：

```

void MainWindow::openFile()
{
    QString path = QFileDialog::getOpenFileName(this,
                                                tr("Open File"),
                                                ".",
                                                tr("Text Files (*.txt)"));

    if(!path.isEmpty()) {
        QFile file(path);
        if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
            QMessageBox::warning(this, tr("Read File"),
                                tr("Cannot open file:\n%1").arg(path));
            return;
        }
    }
}

```

```

        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    } else {
        QMessageBox::warning(this, tr("Path"),
                               tr("You did not select any file."));
    }
}

void MainWindow::saveFile()
{
    QString path = QFileDialog::getSaveFileName(this,
                                                  tr("Open File"),
                                                  ".",
                                                  tr("Text Files (*.txt)"));

    if(!path.isEmpty()) {
        QFile file(path);
        if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
            QMessageBox::warning(this, tr("Write File"),
                                   tr("Cannot open file:\n%1").arg(path));

            return;
        }
        QTextStream out(&file);
        out << textEdit->toPlainText();
        file.close();
    } else {
        QMessageBox::warning(this, tr("Path"),
                               tr("You did not select any file."));
    }
}

```

在 `openFile()` 函数中，我们使用 `QFileDialog::getOpenFileName()` 来获取需要打开的文件的路径。这个函数具有一个长长的签名：

```

QString getOpenFileName(QWidget * parent = 0,
                        const QString & caption = QString(),
                        const QString & dir = QString(),
                        const QString & filter = QString(),
                        QString * selectedFilter = 0,
                        Options options = 0)

```

不过注意，它的所有参数都是可选的，因此在一定程度上说，这个函数也是简单的。这六个参数分别是：

- **parent**: 父窗口。我们前面介绍过, Qt 的标准对话框提供静态函数, 用于返回一个模态对话框 (在一定程度上这就是外观模式的一种体现);
- **caption**: 对话框标题;
- **dir**: 对话框打开时的默认目录, “.” 代表程序运行目录, “/” 代表当前盘符的根目录 (特指 Windows 平台; Linux 平台当然就是根目录), 这个参数也可以是平台相关的, 比如 “C:\\” 等;
- **filter**: 过滤器。我们使用文件对话框可以浏览很多类型的文件, 但是, 很多时候我们仅希望打开特定类型的文件。比如, 文本编辑器希望打开文本文件, 图片浏览器希望打开图片文件。过滤器就是用于过滤特定的后缀名。如果我们使用 “Image Files (\*.jpg \*.png)”, 则只能显示后缀名是 jpg 或者 png 的文件。如果需要多个过滤器, 使用 “;” 分割, 比如 “JPEG Files (\*.jpg);;PNG Files (\*.png)”;
- **selectedFilter**: 默认选择的过滤器;
- **options**: 对话框的一些参数设定, 比如只显示文件夹等等, 它的取值是 `enum QFileDialog::Option`, 每个选项可以使用 `|` 运算组合起来。

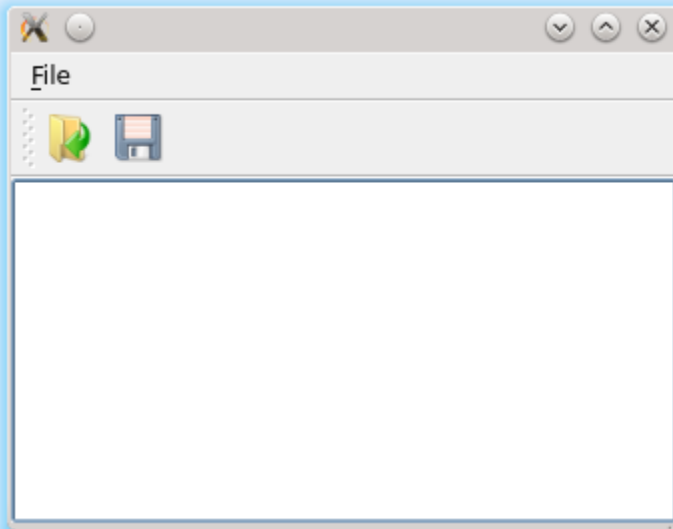
`QFileDialog::getOpenFileName()` 返回值是选择的文件路径。我们将其赋值给 `path`。通过判断 `path` 是否为空, 可以确定用户是否选择了某一文件。只有当用户选择了一个文件时, 我们才执行下面的操作。在 `saveFile()` 中使用的 `QFileDialog::getSaveFileName()` 也是类似的。使用这种静态函数, 在 Windows、Mac OS 上面都是直接调用本地对话框, 但是 Linux 上则是 `QFileDialog` 自己的模拟。这暗示了, 如果你不使用这些静态函数, 而是直接使用 `QFileDialog` 进行设置, 就像我们前面介绍的 `QMessageBox` 的设置一样, 那么得到的对话框很可能与系统对话框的外观不一致。这一点是需要注意的。

首先, 我们创建一个 `QFile` 对象, 将用户选择的文件路径传递给这个对象。然后我们需要打开这个文件, 使用的是 `QFile::open()`, 其参数是指定的打开方式, 这里我们使用只读方式和文本方式打开这个文件 (因为我们选择的是后缀名 `txt` 的文件, 可以认为是文本文件。当然, 在实际应用中, 可能需要进行进一步的判断)。 `QFile::open()` 打开成功则返回 `true`, 由此继续进行下面的操作: 使用 `QTextStream::readAll()` 读取文件所有内容, 然后将其赋值给 `QTextEdit` 显示出来。最后不要忘记关闭文件。另外, `saveFile()` 函数也是类似的, 只不过最后一步, 我们使用 `<<` 重定向, 将 `QTextEdit` 的内容输出到一个文件中。关于文件操作, 我们会在后面的章节中进一步介绍。

这里需要注意一点: 我们的代码仅仅是用于演示, 很多必须的操作并没有进行。比如, 我们没有检查这个文件的实际类型是不是一个文本文件。并且, 我们使用了 `QTextStream::readAll()` 直接读取文件所有内容, 如果这个文件有 100M, 程序会立刻死掉,

这些都是实际程序必须考虑的问题。不过这些内容已经超出我们本章的介绍，也就不再详细说明。

至此，我们的代码已经介绍完毕，马上可以编译运行一下了：



本章的代码可以在这里下载：

- [ch17-qt4.zip](#)
- [ch17-qt5.zip](#)

## 18 事件

事件（event）是由系统或者 Qt 本身在不同的时刻发出的。当用户按下鼠标、敲下键盘，或者是窗口需要重新绘制的时候，都会发出一个相应的事件。一些事件在对用户操作做出响应时发出，如键盘事件等；另一些事件则是由系统自动发出，如计时器事件。

事件也就是我们通常说的“事件驱动（event drive）”程序设计的基础概念。事件的出现，使得程序代码不会按照原始的线性顺序执行。想想看，从最初的 C 语言开始，我们的程序就是以一种线性的顺序执行代码：这一条语句执行之后，开始执行下一条语句；这一个函数执行过后，开始执行下一个函数。这种类似“批处理”的程序设计风格显然不适合于处理复杂的用户交互。我们来想象一下用户交互的情景：我们设计了一堆功能放在界面上，用户点击了“打开文件”，于是开始执行打开文件的操作；用户点击了“保存文件”，于是开始执行保存文件的操作。我们不知道用户究竟想进行什么操作，因此也就不能预测接下来将会调用哪一个

函数。如果我们设计了一个“文件另存为”的操作，如果用户不点击，这个操作将永远不会被调用。这就是所谓的“事件驱动”，我们的程序的执行顺序不再是线性的，而是由一个个事件驱动着程序继续执行。没有事件，程序将阻塞在那里，不执行任何代码。

在 Qt 中，事件的概念似乎同信号槽类似。的确如此，一般来说，使用 Qt 组件时，我们并不会把主要精力放在事件上。因为在 Qt 中，我们关心的更多的是事件关联的一个信号。比如，对于 QPushButton 的鼠标点击，我们不需要关心这个鼠标点击事件，而是关心它的 clicked() 信号的发出。这与其他的一些 GUI 框架不同：在 Swing 中，你所要关心的是 JButton 的 ActionListener 这个点击事件。由此看出，相比于其他 GUI 框架，Qt 给了我们额外的选择：信号槽。

但是，Qt 中的事件和信号槽却并不是可以相互替代的。信号由具体的对象发出，然后会马上交给由 connect() 函数连接的槽进行处理；而对于事件，Qt 使用一个事件队列对所有发出的事件进行维护，当新的事件产生时，会被追加到事件队列的尾部。前一个事件完成后，取出后面的事件进行处理。但是，必要的时候，Qt 的事件也可以不进入事件队列，而是直接处理。信号一旦发出，对应的槽函数一定会被执行。但是，事件则可以使用“事件过滤器”进行过滤，对于有些事件进行额外的处理，另外的事件则不关心。总的来说，如果我们使用组件，我们关心的是信号槽；如果我们自定义组件，我们关心的是事件。因为我们可以通过事件来改变组件的默认操作。比如，如果我们要自定义一个能够响应鼠标事件的 EventLabel，我们就需要重写 QLabel 的鼠标事件，做出我们希望的操作，有可能还得在恰当的时候发出一个类似按钮的 clicked() 信号（如果我们希望让这个 EventLabel 能够被其它组件使用）或者其它信号。

在前面我们也曾经简单提到，Qt 程序需要在 main() 函数创建一个 QApplication 对象，然后调用它的 exec() 函数。这个函数就是开始 Qt 的事件循环。在执行 exec() 函数之后，程序将进入事件循环来监听应用程序的事件。当事件发生时，Qt 将创建一个事件对象。Qt 中所有事件类都继承于 QEvent。在事件对象创建完毕后，Qt 将这个事件对象传递给 QObject 的 event() 函数。event() 函数并不直接处理事件，而是按照事件对象的类型分派给特定的事件处理函数（event handler）。关于这一点，我们会在以后的章节中详细说明。

在所有组件的父类 QWidget 中，定义了很多事件处理的回调函数，如 keyPressEvent()、keyReleaseEvent()、mouseDoubleClickEvent()、mouseMoveEvent()、mousePressEvent()、mouseReleaseEvent() 等。这些函数都是 protected virtual 的，也就是说，我们可以在子类中重新实现这些函数。下面来看一个例子：

```
class EventLabel : public QLabel
{
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
```

```

};

void EventLabel::mouseMoveEvent(QMouseEvent *event)
{
    this->setText(QString("<center><h1>Move: (%1, %2)</h1></center>")
        .arg(QString::number(event->x()),
        QString::number(event->y())));
}

void EventLabel::mousePressEvent(QMouseEvent *event)
{
    this->setText(QString("<center><h1>Press: (%1, %2)</h1></center>")
        .arg(QString::number(event->x()),
        QString::number(event->y())));
}

void EventLabel::mouseReleaseEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Release: (%d, %d)</h1></center>",
        event->x(), event->y());
    this->setText(msg);
}

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

    EventLabel *label = new EventLabel;
    label->setWindowTitle("MouseEvent Demo");
    label->resize(300, 200);
    label->show();

    return a.exec();
}

```

我们编译运行上面的代码，就可以理解到有关事件的使用方法。

EventLabel 继承了 QLabel，覆盖了 mousePressEvent()、mouseMoveEvent() 和 mouseReleaseEvent() 三个函数。我们并没有添加什么功能，只是在鼠标按下（press）、鼠标移动（move）和鼠标释放（release）的时候，把当前鼠标的坐标值显示在这个 Label 上面。由于 QLabel 是支持 HTML 代码的，因此我们直接使用了 HTML 代码来格式化文字。

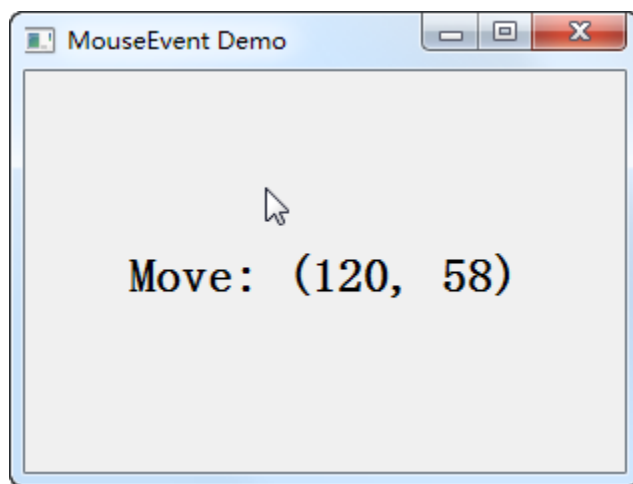
QString 的 `arg()` 函数可以自动替换掉 QString 中出现的占位符。其占位符以 `%` 开始，后面是占位符的位置，例如 `%1`, `%2` 这种。

```
QString("[%1, %2"]').arg(x, y);
```

语句将会使用 `x` 替换 `%1`, `y` 替换 `%2`, 因此, 这个语句生成的 QString 为 `[x, y]`。

在 `mousePressEvent()` 函数中, 我们使用了另外一种 QString 的构造方法。我们使用类似 C 风格的格式化函数 `sprintf()` 来构造 QString。

运行上面的代码, 当我们点击了一下鼠标之后, `label` 上将显示鼠标当前坐标值。



为什么要点击鼠标之后才能在 `mouseMoveEvent()` 函数中显示鼠标坐标值? 这是因为 QWidget 中有一个 `mouseTracking` 属性, 该属性用于设置是否追踪鼠标。只有鼠标被追踪时, `mouseMoveEvent()` 才会发出。如果 `mouseTracking` 是 `false` (默认即是), 组件在至少一次鼠标点击之后, 才能够被追踪, 也就是能够发出 `mouseMoveEvent()` 事件。如果 `mouseTracking` 为 `true`, 则 `mouseMoveEvent()` 直接可以被发出。知道了这一点, 我们就可以在 `main()` 函数中直接设置下:

```
EventLabel *label = new EventLabel;  
label->setWindowTitle("MouseEvent Demo");  
label->resize(300, 200);  
label->setMouseTracking(true);  
label->show();
```

这样子就没有这个问题了。

## 19 事件的接受与忽略

版本:

1. 2012-09-29
2. 2013-04-23 更新有关 `accept()`和 `ignore()`函数的相关内容。
3. 2013-12-02 增加有关 `accept()`和 `ignore()`函数的示例。

上一章我们介绍了有关事件的相关内容。我们曾经提到,事件可以依情况接受和忽略。现在,我们就来了解下有关事件的更多的知识。

首先来看一段代码:

```
//!!! Qt5
// ----- custombutton.h ----- //
class CustomButton : public QPushButton
{
    Q_OBJECT
public:
    CustomButton(QWidget *parent = 0);
private:
    void onButtonCliecked();
};

// ----- custombutton.cpp ----- //
CustomButton::CustomButton(QWidget *parent) :
    QPushButton(parent)
{
    connect(this, &CustomButton::clicked,
            this, &CustomButton::onButtonCliecked);
}

void CustomButton::onButtonCliecked()
{
    qDebug() << "You clicked this!";
}

// ----- main.cpp ----- //
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
```



```

CustomButton btn;
btn.setText("This is a Button!");
btn.show();

return a.exec();
}

```

这是一段简单的代码，经过我们前面一段时间的学习，我们已经能够知道这段代码的运行结果：点击按钮，会在控制台打印出“You clicked this!”字符串。这是我们前面介绍过的内容。下面，我们向 CustomButton 类添加一个事件函数：

```

// CustomButton
...
protected:
    void mousePressEvent(QMouseEvent *event);
...

// ----- custombutton.cpp ----- //
...
void CustomButton::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        qDebug() << "left";
    } else {
        QPushButton::mousePressEvent(event);
    }
}
...

```

我们重写了 CustomButton 的 mousePressEvent() 函数，也就是鼠标按下。在这个函数中，我们判断如果鼠标按下的是左键，则打印出来“left”字符串，否则，调用父类的同名函数。编译运行这段代码，当我们点击按钮时，“You clicked this!”字符串不再出现，只有一个“left”。也就是说，我们把父类的实现覆盖掉了。由此可以看出，父类 QPushButton 的 mousePressEvent() 函数中肯定发出了 clicked() 信号，否则的话，我们的槽函数怎么会不执行了呢？这暗示我们一个非常重要的细节：**当重写事件回调函数时，时刻注意是否需要通过调用父类的同名函数来确保原有实现仍能进行！**比如我们的 CustomButton 了，如果像我们这么覆盖函数，clicked() 信号永远不会发生，你连接到这个信号的槽函数也就永远不会被执行。这个错误非常隐蔽，很可能会浪费你很多时间才能找到。因为这个错误不会有任何提示。这一定程度上说，我们的组件“忽略”了父类的事件，但这更多的是一种违心之举，一种错误。

通过调用父类的同名函数，我们可以把 Qt 的事件传递看成链状：如果子类没有处理这个事件，就会继续向其父类传递。Qt 的事件对象有两个函数：accept()和 ignore()。正如它们的名字一样，前者用来告诉 Qt，这个类的事件处理函数想要处理这个事件；后者则告诉 Qt，这个类的事件处理函数不想要处理这个事件。在事件处理函数中，可以使用 isAccepted()来查询这个事件是不是已经被接收了。具体来说：如果一个事件处理函数调用了事件对象的 accept()函数，这个事件就不会被继续传播给其父组件；如果它调用了事件的 ignore()函数，Qt 会从其父组件中寻找另外的接受者。

事实上，我们很少会使用 accept()和 ignore()函数，而是像上面的示例一样，如果希望忽略事件（所谓忽略，是指自己不要这个事件），只要调用父类的响应函数即可。记得我们曾经说过，Qt 中的事件都是 protected 的，因此，重写的函数必定存在着其父类中的响应函数，所以，这个方法是可行的。为什么要这么做，而不是自己去手动调用这两个函数呢？因为我们无法确认父类中的这个处理函数有没有额外的操作。如果我们在子类中直接忽略事件，Qt 会去寻找其他的接收者，该子类的父类的操作会被忽略（因为没有调用父类的同名函数），这可能会有潜在的危险。为了避免自己去调用 accept()和 ignore()函数，而是尽量调用父类实现，Qt 做了特殊的设计：事件对象默认是 accept 的，而作为所有组件的父类 QWidget 的默认实现则是调用 ignore()。这么一来，如果你自己实现事件处理函数，不调用 QWidget 的默认实现，你就等于是接受了事件；如果你要忽略事件，只需调用 QWidget 的默认实现。这一点我们前面已经说明。下面可以从代码级别来理解这一点，我们可以查看一下 QWidget 的 mousePressEvent()函数的实现：

```
//!!! Qt5
void QWidget::mousePressEvent(QMouseEvent *event)
{
    event->ignore();
    if ((windowType() == Qt::Popup)) {
        event->accept();
        QWidget* w;
        while ((w = QApplication::activePopupWidget()) && w != this){
            w->close();
            if (QApplication::activePopupWidget() == w)
                w->hide(); // hide at least
        }
        if (!rect().contains(event->pos())){
            close();
        }
    }
}
```

这段代码在 Qt4 和 Qt5 中基本一致（区别在于 activePopupWidget()一行，Qt4 的版本是 qApp->activePopupWidget()）。注意函数的第一个语句：event->ignore()，如果子类都没有重写这个函数，Qt 会默认忽略这个事件，继续寻找下一个事件接收者。如果我们在子

类的 `mousePressEvent()` 函数中直接调用了 `accept()` 或者 `ignore()`，而没有调用父类的同名函数，`QWidget::mousePressEvent()` 函数中关于 `Popup` 判断的那段代码就不会被执行，因此可能会出现默认其妙的怪异现象。

针对 `accept()` 和 `ignore()`，我们再来看一个例子：

```
class CustomButton : public QPushButton
{
    Q_OBJECT
public:
    CustomButton::CustomButton(QWidget *parent)
        : QPushButton(parent)
    {
    }
protected:
    void mousePressEvent(QMouseEvent *event)
    {
        qDebug() << "CustomButton";
    }
};

class CustomButtonEx : public CustomButton
{
    Q_OBJECT
public:
    CustomButtonEx::CustomButtonEx(QWidget *parent)
        : CustomButton(parent)
    {
    }
protected:
    void mousePressEvent(QMouseEvent *event)
    {
        qDebug() << "CustomButtonEx";
    }
};

class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    CustomWidget::CustomWidget(QWidget *parent)
        : QWidget(parent)
    {
    }
}
```

```
protected:
    void mousePressEvent(QMouseEvent *event)
    {
        qDebug() << "CustomWidget";
    }
};

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow::MainWindow(QWidget *parent = 0)
        : QMainWindow(parent)
    {
        CustomWidget *widget = new CustomWidget(this);
        CustomButton *cbex = new CustomButton(widget);
        cbex->setText(tr("CustomButton"));
        CustomButtonEx *cb = new CustomButtonEx(widget);
        cb->setText(tr("CustomButtonEx"));
        QVBoxLayout *widgetLayout = new QVBoxLayout(widget);
        widgetLayout->addWidget(cbex);
        widgetLayout->addWidget(cb);
        this->setCentralWidget(widget);
    }
protected:
    void mousePressEvent(QMouseEvent *event)
    {
        qDebug() << "MainWindow";
    }
};
```

这段代码在一个 `MainWindow` 中添加了一个 `CustomWidget`，里面有两个按钮对象：`CustomButton` 和 `CustomButtonEx`。每一个类都重写了 `mousePressEvent()` 函数。运行程序点击 `CustomButtonEx`，结果是

#### CustomButtonEx

这是因为我们重写了鼠标按下的事件，但是并没有调用父类函数或者显式设置 `accept()` 或 `ignore()`。下面我们在 `CustomButtonEx` 的 `mousePressEvent()` 第一行增加一句 `event->accept()`，重新运行，发现结果不变。正如我们前面所说，`QEvent` 默认是 `accept` 的，调用这个函数并没有什么区别。然后我们将 `CustomButtonEx` 的 `event->accept()` 改成 `event->ignore()`。这次运行结果是

CustomButtonEx  
CustomWidget

ignore()说明我们想让事件继续传播，于是 CustomButtonEx 的父组件 CustomWidget 也收到了这个事件，所以输出了自己的结果。同理，CustomWidget 又没有调用父类函数或者显式设置 accept()或 ignore()，所以事件传播就此打住。这里值得注意的是，CustomButtonEx 的事件传播给了父组件 CustomWidget，而不是它的父类 CustomButton。事件的传播是在组件层次上面的，而不是依靠类继承机制。

接下来我们继续测试，在 CustomWidget 的 mousePressEvent() 中增加 QWidget::mousePressEvent(event)。这次的输出是

CustomButtonEx  
CustomWidget  
MainWindow

如果你把 `QWidget::mousePressEvent(event)` 改成 `event->ignore()`，结果也是一样的。这正如我们前面说的，`QWidget` 的默认是调用 `event->ignore()`。

不过，事情也不是绝对的。在一个情形下，我们必须使用 `accept()` 和 `ignore()` 函数，那就是窗口关闭的事件。**注意，不要试图用前面了解到的有关 `accept()` 和 `ignore()` 的知识来理解这个事件！**对于 `QCloseEvent` 事件，调用 `accept()` 意味着 Qt 会停止事件的传播，调用 `ignore()` 则意味着事件继续传播。这与前面所说的正好相反。回到我们前面写的简单的文本编辑器。我们在构造函数中添加如下代码：

[illegible]

```

                                QMessageBox::No) == QMessageBox::Yes;

    if (exit) {
        event->accept();
    } else {
        event->ignore();
    }
} else {
    event->accept();
}
}
}

```

setWindowTitle()函数可以使用 [\*] 这种语法来表明，在窗口内容发生改变时（通过 setWindowModified(true)函数通知），Qt 会自动在标题上面的 [\*] 位置替换成 \* 号。我们使用 Lambda 表达式连接 QTextEdit::textChanged() 信号，将 windowModified 设置为 true。然后我们需要重写 closeEvent() 函数。在这个函数中，我们首先判断是不是有过修改，如果有，则弹出询问框，问一下是否要退出。如果用户点击了“Yes”，则接受关闭事件，这个事件所在的操作就是关闭窗口。因此，一旦接受事件，窗口就会被关闭；否则窗口继续保留。当然，如果窗口内容没有被修改，则直接接受事件，关闭窗口。

## 20 event()

前面的章节中我们曾经提到 event() 函数。事件对象创建完毕后，Qt 将这个事件对象传递给 QObject 的 event() 函数。event() 函数并不直接处理事件，而是将这些事件对象按照它们不同的类型，分发给不同的事件处理器（event handler）。

如上所述，event() 函数主要用于事件的分发。所以，如果你希望在事件分发之前做一些操作，就可以重写这个 event() 函数了。例如，我们希望在一个 QWidget 组件中监听 tab 键的按下，那么就可以继承 QWidget，并重写它的 event() 函数，来达到这个目的：

```

bool CustomWidget::event(QEvent *e)
{
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return QWidget::event(e);
}

```

`CustomWidget` 是一个普通的 `QWidget` 子类。我们重写了它的 `event()` 函数，这个函数有一个 `QEvent` 对象作为参数，也就是需要转发的事件对象。函数返回值是 `bool` 类型。如果传入的事件已被识别并且处理，则需要返回 `true`，否则返回 `false`。如果返回值是 `true`，并且，该事件对象设置了 `accept()`，那么 Qt 会认为这个事件已经处理完毕，不会再将这个事件发送给其它对象，而是会继续处理事件队列中的下一事件。注意，在 `event()` 函数中，调用事件对象的 `accept()` 和 `ignore()` 函数是没有作用的，不会影响到事件的传播。

我们可以通过使用 `QEvent::type()` 函数可以检查事件的实际类型，其返回值是 `QEvent::Type` 类型的枚举。我们处理过自己感兴趣的事件之后，可以直接返回 `true`，表示我们已经对此事件进行了处理；对于其它我们不关心的事件，则需要调用父类的 `event()` 函数继续转发，否则这个组件就只能处理我们定义的事件了。为了测试这一种情况，我们可以尝试下面的代码：

```
bool CustomTextEdit::event(QEvent *e)
{
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return false;
}
```

`CustomTextEdit` 是 `QTextEdit` 的一个子类。我们重写了其 `event()` 函数，却没有调用父类的同名函数。这样，我们的组件就只能处理 Tab 键，再也无法输入任何文本，也不能响应其它事件，比如鼠标点击之后也不会有光标出现。这是因为我们只处理了 `KeyPress` 类型的事件，并且如果不是 `KeyPress` 事件，则直接返回 `false`，鼠标事件根本不会被转发，也就没有了鼠标事件。

通过查看 `QObject::event()` 的实现，我们可以理解，`event()` 函数同前面的章节中我们所说的事件处理器有什么联系：

```
//!!! Qt5
bool QObject::event(QEvent *e)
{
    switch (e->type()) {
        case QEvent::Timer:
            timerEvent((QTimerEvent*)e);
            break;
```

```

    case QEvent::ChildAdded:
    case QEvent::ChildPolished:
    case QEvent::ChildRemoved:
        childEvent((QChildEvent*)e);
        break;
    // ...
    default:
        if (e->type() >= QEvent::User) {
            customEvent(e);
            break;
        }
        return false;
    }
    return true;
}

```

这是 Qt 5 中 `QObject::event()` 函数的源代码（Qt 4 的版本也是类似的）。我们可以看到，同前面我们所说的一样，Qt 也是使用 `QEvent::type()` 判断事件类型，然后调用了特定的事件处理器。比如，如果 `event->type()` 返回值是 `QEvent::Timer`，则调用 `timerEvent()` 函数。可以想象，`QWidget::event()` 中一定会有如下的代码：

```

switch (event->type()) {
    case QEvent::MouseMove:
        mouseMoveEvent((QMouseEvent*)event);
        break;
    // ...
}

```

事实也的确如此。`timerEvent()` 和 `mouseMoveEvent()` 这样的函数，就是我们前面章节所说的事件处理器 `event handler`。也就是说，`event()` 函数中实际是通过事件处理器来响应一个具体的事件。这相当于 `event()` 函数将具体事件的处理“委托”给具体的事件处理器。而这些事件处理器是 `protected virtual` 的，因此，我们重写了某一个事件处理器，即可让 Qt 调用我们自己实现的版本。

由此可以见，`event()` 是一个集中处理不同类型的事件的地方。如果你不想重写一大堆事件处理器，就可以重写这个 `event()` 函数，通过 `QEvent::type()` 判断不同的事件。鉴于重写 `event()` 函数需要十分小心注意父类的同名函数的调用，一不留神就可能出现错误，所以一般还是建议只重写事件处理器（当然，也必须记得是不是应该调用父类的同名处理器）。这其实暗示了 `event()` 函数的另外一个作用：屏蔽掉某些不需要的事件处理器。正如我们前面的 `CustomTextEdit` 例子看到的那样，我们创建了一个只能响应 `tab` 键的组件。这种作用是重写事件处理器所不能实现的。



## 21 事件过滤器

有时候，对象需要查看、甚至要拦截发送到另外对象的事件。例如，对话框可能想要拦截按键事件，不让别的组件接收到；或者要修改回车键的默认处理。

通过前面的章节，我们已经知道，Qt 创建了 `QEvent` 事件对象之后，会调用 `QObject` 的 `event()` 函数处理事件的分发。显然，我们可以在 `event()` 函数中实现拦截的操作。由于 `event()` 函数是 `protected` 的，因此，需要继承已有类。如果组件很多，就需要重写很多个 `event()` 函数。这当然相当麻烦，更不用说重写 `event()` 函数还得小心一堆问题。好在 Qt 提供了另外一种机制来达到这一目的：事件过滤器。

`QObject` 有一个 `eventFilter()` 函数，用于建立事件过滤器。这个函数的签名如下：

```
virtual bool QObject::eventFilter ( QObject * watched, QEvent * event );
```

这个函数正如其名字显示的那样，是一个“事件过滤器”。所谓事件过滤器，可以理解成一种过滤代码。想想做化学实验时用到的过滤器，可以将杂质留到滤纸上，让过滤后的液体溜走。事件过滤器也是如此：它会检查接收到的事件。如果这个事件是我们感兴趣的类型，就进行我们自己的处理；如果不是，就继续转发。这个函数返回一个 `bool` 类型，如果你想将参数 `event` 过滤出来，比如，不想让它继续转发，就返回 `true`，否则返回 `false`。事件过滤器的调用时间是目标对象（也就是参数里面的 `watched` 对象）接收到事件对象之前。也就是说，如果你在事件过滤器中停止了某个事件，那么，`watched` 对象以及以后所有的事件过滤器根本不会知道这么一个事件。

我们来看一段简单的代码：

```
class MainWindow : public QMainWindow
{
public:
    MainWindow();
protected:
    bool eventFilter(QObject *obj, QEvent *event);
private:
    QTextEdit *textEdit;
};

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
```

```

        setCentralWidget(textEdit);

        textEdit->installEventFilter(this);
    }

    bool MainWindow::eventFilter(QObject *obj, QEvent *event)
    {
        if (obj == textEdit) {
            if (event->type() == QEvent::KeyPress) {
                QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
                qDebug() << "Ate key press" << keyEvent->key();
                return true;
            } else {
                return false;
            }
        } else {
            // pass the event on to the parent class
            return QMainWindow::eventFilter(obj, event);
        }
    }
}

```

MainWindow 是我们定义的一个类。我们重写了它的 `eventFilter()` 函数。为了过滤特定组件上的事件，首先需要判断这个对象是不是我们感兴趣的组件，然后判断这个事件的类型。在上面的代码中，我们不想让 `textEdit` 组件处理键盘按下事件。所以，首先我们找到这个组件，如果这个事件是键盘事件，则直接返回 `true`，也就是过滤掉了这个事件，其他事件还是要继续处理，所以返回 `false`。对于其它的组件，我们并不保证是不是还有过滤器，于是最保险的办法是调用父类的函数。

`eventFilter()` 函数相当于创建了过滤器，然后我们需要安装这个过滤器。安装过滤器需要调用 `QObject::installEventFilter()` 函数。这个函数的签名如下：

```
void QObject::installEventFilter ( QObject * filterObj )
```

这个函数接受一个 `QObject *` 类型的参数。记得刚刚我们说的，`eventFilter()` 函数是 `QObject` 的一个成员函数，因此，任意 `QObject` 都可以作为事件过滤器（问题在于，如果你没有重写 `eventFilter()` 函数，这个事件过滤器是没有任何作用的，因为默认什么都不会过滤）。已经存在的过滤器则可以通过 `QObject::removeEventFilter()` 函数移除。

我们可以向一个对象上面安装多个事件处理器，只要调用多次 `installEventFilter()` 函数。如果一个对象存在多个事件过滤器，那么，最后一个安装的会第一个执行，也就是后进先执行的顺序。

还记得我们前面的那个例子吗？我们使用 `event()` 函数处理了 Tab 键：

```
bool CustomWidget::event(QEvent *e)
{
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return QWidget::event(e);
}
```

现在，我们可以给出一个使用事件过滤器的版本：

```
bool FilterObject::eventFilter(QObject *object, QEvent *event)
{
    if (object == target && event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        } else {
            return false;
        }
    }
    return false;
}
```

事件过滤器的强大之处在于，我们可以为整个应用程序添加一个事件过滤器。记得，`installEventFilter()` 函数是 `QObject` 的函数，`QApplication` 或者 `QCoreApplication` 对象都是 `QObject` 的子类，因此，我们可以向 `QApplication` 或者 `QCoreApplication` 添加事件过滤器。这种全局的事件过滤器将会在所有其它特性对象的事件过滤器之前调用。尽管很强大，但这种行为会严重降低整个应用程序的事件分发效率。因此，除非是不得不使用的情况，否则的话我们不应该这么做。

**注意**，如果你在事件过滤器中 `delete` 了某个接收组件，务必将函数返回值设为 `true`。否则，Qt 还是会将事件分发给这个接收组件，从而导致程序崩溃。

事件过滤器和被安装过滤器的组件必须在同一线程，否则，过滤器将不起作用。另外，如果在安装过滤器之后，这两个组件到了不同的线程，那么，只有等到二者重新回到同一线程的时候过滤器才会有效。

## 22 事件总结

Qt 的事件是整个 Qt 框架的核心机制之一，也比较复杂。说它复杂，更多是因为它涉及到的函数众多，而处理方法也很多，有时候让人难以选择。现在我们简单总结一下 Qt 中的事件机制。

Qt 中有很多种事件：鼠标事件、键盘事件、大小改变的事件、位置移动的事件等等。如何处理这些事件，实际有两种选择：

1. 所有事件对应一个事件处理函数，在这个事件处理函数中用一个很大的分支语句进行选择，其代表作就是 win32 API 的 `WndProc()` 函数：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
```

在这个函数中，我们需要使用 `switch` 语句，选择 `message` 参数的类型进行处理，典型代码是：

```
switch(message)
{
    case WM_PAINT:
        // ...
        break;
    case WM_DESTROY:
        // ...
        break;
    ...
}
```

2. 每一种事件对应一个事件处理函数。Qt 就是使用的这么一种机制：

- `mouseEvent()`
- `keyPressEvent()`
- ...

Qt 具有这么多种事件处理函数，肯定有一个地方对其进行分发，否则，Qt 怎么知道哪一种事件调用哪一个事件处理函数呢？这个分发的函数，就是 `event()`。显然，当 `QMouseEvent` 产生之后，`event()` 函数将其分发给 `mouseEvent()` 事件处理器进行处理。

`event()` 函数会有两个问题：

1. `event()` 函数是一个 `protected` 的函数，这意味着我们要想重写 `event()`，必须继承一个已有的类。试想，我的程序根本不要鼠标事件，程序中所有组件都不允许处理鼠标事件，是不是我得继承所有组件，一一重写其 `event()` 函数？`protected` 函数带来的另外一个问题是，如果我基于第三方库进行开发，而对方没有提供源代码，只有一个链接库，其它都是封装好的。我怎么去继承这种库中的组件呢？
2. `event()` 函数的确有一定的控制，不过有时候我的需求更严格一些：我希望那些组件根本看不到这种事件。`event()` 函数虽然可以拦截，但其实也是接收到了 `QMouseEvent` 对象。我连让它收都收不到。这样做的好处是，模拟一种系统根本没有那个事件的效果，所以其它组件根本不会收到这个事件，也就无需修改自己的事件处理函数。这种需求怎么办呢？

这两个问题是 `event()` 函数无法处理的。于是，Qt 提供了另外一种解决方案：事件过滤器。事件过滤器给我们一种能力，让我们能够完全移除某种事件。事件过滤器可以安装到任意 `QObject` 类型上面，并且可以安装多个。如果要实现全局的事件过滤器，则可以安装到 `QApplication` 或者 `QCoreApplication` 上面。这里需要注意的是，如果使用 `installEventFilter()` 函数给一个对象安装事件过滤器，那么该事件过滤器只对该对象有效，只有这个对象的事件需要先传递给事件过滤器的 `eventFilter()` 函数进行过滤，其它对象不受影响。如果给 `QApplication` 对象安装事件过滤器，那么该过滤器对程序中的每一个对象都有效，任何对象的事件都是先传给 `eventFilter()` 函数。

事件过滤器可以解决刚刚我们提出的 `event()` 函数的两点不足：首先，事件过滤器不是 `protected` 的，因此我们可以向任何 `QObject` 子类安装事件过滤器；其次，事件过滤器在目标对象接收到事件之前进行处理，如果我们将事件过滤掉，目标对象根本不会见到这个事件。

事实上，还有一种方法，我们没有介绍。Qt 事件的调用最终都会追溯到 `QCoreApplication::notify()` 函数，因此，最大的控制权实际上是重写 `QCoreApplication::notify()`。这个函数的声明是：

```
virtual bool QCoreApplication::notify ( QObject * receiver, QEvent * event );
```

该函数会将 `event` 发送给 `receiver`，也就是调用 `receiver->event(event)`，其返回值就是来自 `receiver` 的事件处理器。注意，这个函数为任意线程的任意对象的任意事件调用，因此，它不存在事件过滤器的线程的问题。不过我们并不推荐这么做，因为 `notify()` 函数只有一个，而事件过滤器要灵活得多。

现在我们可以总结一下 Qt 的事件处理，实际上是有五个层次：

1. 重写 `paintEvent()`、`mousePressEvent()` 等事件处理函数。这是最普通、最简单的形式，同时功能也最简单。
2. 重写 `event()` 函数。`event()` 函数是所有对象的事件入口，`QObject` 和 `QWidget` 中的实现，默认是把事件传递给特定的事件处理函数。
3. 在特定对象上面安装事件过滤器。该过滤器仅过滤该对象接收到的事件。
4. 在 `QCoreApplication::instance()` 上面安装事件过滤器。该过滤器将过滤所有对象的所有事件，因此和 `notify()` 函数一样强大，但是它更灵活，因为可以安装多个过滤器。全局的事件过滤器可以看到 `disabled` 组件上面发出的鼠标事件。全局过滤器有一个问题：只能用在主线程。
5. 重写 `QCoreApplication::notify()` 函数。这是最强大的，和全局事件过滤器一样提供完全控制，并且不受线程的限制。但是全局范围内只能有一个被使用（因为 `QCoreApplication` 是单例的）。

为了进一步了解这几个层次的事件处理方式的调用顺序，我们可以编写一个测试代码：

```
class Label : public QWidget
{
public:
    Label()
    {
        installEventFilter(this);
    }

    bool eventFilter(QObject *watched, QEvent *event)
    {
        if (watched == this) {
            if (event->type() == QEvent::MouseButtonPress) {
                qDebug() << "eventFilter";
            }
        }
        return false;
    }

protected:
    void mousePressEvent(QMouseEvent *)
    {
        qDebug() << "mousePressEvent";
    }
}
```

```

    }

    bool event(QEvent *e)
    {
        if (e->type() == QEvent::MouseButtonPress) {
            qDebug() << "event";
        }
        return QWidget::event(e);
    }
};

class EventFilter : public QObject
{
public:
    EventFilter(QObject *watched, QObject *parent = 0) :
        QObject(parent),
        m_watched(watched)
    {
    }

    bool eventFilter(QObject *watched, QEvent *event)
    {
        if (watched == m_watched) {
            if (event->type() == QEvent::MouseButtonPress) {
                qDebug() << "QApplication::eventFilter";
            }
        }
        return false;
    }

private:
    QObject *m_watched;
};

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Label label;
    app.installEventFilter(new EventFilter(&label, &label));
    label.show();
    return app.exec();
}

```

我们可以看到，鼠标点击之后的输出结果是：

```
QApplication::eventFilter  
eventFilter  
event  
mousePressEvent
```

因此可以知道，全局事件过滤器被第一个调用，之后是该对象上面的事件过滤器，其次是 `event()` 函数，最后是特定的事件处理函数。

## 23 自定义事件

尽管 Qt 已经提供了很多事件，但对于更加千变万化的需求来说，有限的事件都是不够的。例如，我要支持一种新的设备，这个设备提供一种崭新的交互方式，那么，这种事件如何处理呢？所以，允许创建自己的事件类型也就势在必行。即便是不说那种非常极端的例子，在多线程的程序中，自定义事件也是尤其有用。当然，事件也并不是局限在多线程中，它可以用在单线程的程序中，作为一种对象间通讯的机制。那么，为什么我需要使用事件，而不是信号槽呢？主要原因是，事件的分发既可以是同步的，又可以是异步的，而函数的调用或者说是槽的回调总是同步的。事件的另外一个好处是，它可以使用过滤器。

Qt 自定义事件很简单，同其它类库的使用很相似，都是要继承一个类进行扩展。在 Qt 中，你需要继承的类是 `QEvent`。

继承 `QEvent` 类，最重要的是提供一个 `QEvent::Type` 类型的参数，作为自定义事件的类型值。回忆一下，这个 `type` 是我们在处理事件时用于识别事件类型的代号。比如在 `event()` 函数中，我们使用 `QEvent::type()` 获得这个事件类型，然后与我们定义的实际类型对比。

`QEvent::Type` 是 `QEvent` 定义的一个枚举。因此，我们可以传递一个 `int` 值。但是需要注意的是，我们的自定义事件类型不能和已经存在的 `type` 值重复，否则会有不可预料错误发生。因为系统会将你新增加的事件当做系统事件进行派发和调用。在 Qt 中，系统保留 0 - 999 的值，也就是说，你的事件 `type` 要大于 999。这种数值当然非常难记，所以 Qt 定义了两个边界值：`QEvent::User` 和 `QEvent::MaxUser`。我们的自定义事件的 `type` 应该在这两个值的范围之间。其中，`QEvent::User` 的值是 1000，`QEvent::MaxUser` 的值是 65535。从这里知道，我们最多可以定义 64536 个事件。通过这两个枚举值，我们可以保证我们自己的事件类型不会覆盖系统定义的事件类型。但是，这样并不能保证自定义事件相互之间不会被覆盖。为了解决这个问题，Qt 提供了一个函数：`registerEventType()`，用于自定义事件的注册。该函数签名如下：

```
static int QEvent::registerEventType ( int hint = -1 );
```



这个函数是 `static` 的，因此可以使用 `QEvent` 类直接调用。函数接受一个 `int` 值，其默认值是 `-1`；函数返回值是向系统注册的新的 `Type` 类型的值。如果 `hint` 是合法的，也就是说这个 `hint` 不会发生任何覆盖（系统的以及其它自定义事件的），则会直接返回这个值；否则，系统会自动分配一个合法值并返回。因此，使用这个函数即可完成 `type` 值的指定。这个函数是线程安全的，不必另外添加同步。

我们可以在 `QEvent` 子类中添加自己的事件所需要的数据，然后进行事件的发送。`Qt` 中提供了两种事件发送方式：

```
static bool QApplication::sendEvent(QObject *receiver,  
                                   QEvent *event);
```

1. 直接将 `event` 事件发送给 `receiver` 接受者，使用的是 `QCoreApplication::notify()` 函数。函数返回值就是事件处理函数的返回值。在事件被发送的时候，`event` 对象并不会被销毁。通常我们会在栈上创建 `event` 对象，例如：

```
QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);
QApplication::sendEvent(mainWindow, &event);

static void QCoreApplication::postEvent(QObject *receiver,
                                         QEvent *event);
```

2.将 event 事件及其接受者 receiver 一同追加到事件队列中，函数立即返回。

因为 `post` 事件队列会持有事件对象，并且在其 `post` 的时候将其 `delete` 掉，因此，我们必须在堆上创建 `event` 对象。当对象被发送之后，再试图访问 `event` 对象就会出现 问题（因为 `post` 之后，`event` 对象就会被 `delete`）。

当控制权返回到主线程循环是，保存在事件队列中的所有事件都通过 `notify()` 函数发送出去。

事件会根据 `post` 的顺序进行处理。如果你想要改变事件的处理顺序，可以考虑为其指定一个优先级。默认的优先级是 `Qt::NormalEventPriority`。

这个函数是线程安全的。

Ot 还提供了一个函数:

[illegible]

这个函数的作用是，将事件队列中的接受者为 `receiver`，事件类似为 `event_type` 的所有事件立即发送给 `receiver` 进行处理。需要注意的是，来自窗口系统的事件并不由这个函数进行处理，而是 `processEvent()`。详细信息请参考 `Qt API 手册`。

现在，我们已经能够自定义事件对象，已经能够将事件发送出去，还剩下最后一步：处理自定义事件。处理自定义事件，同前面我们讲解的那些处理方法没有什么区别。我们可以重写 `QObject::customEvent()` 函数，该函数接收一个 `QEvent` 对象作为参数：

```
void QObject::customEvent(QEvent *event);
```

我们可以通过转换 `event` 对象类型来判断不同的事件：

```
void CustomWidget::customEvent(QEvent *event) {
    CustomEvent *customEvent = static_cast<CustomEvent *>(event);
    // ...
}
```

当然，我们也可以在 `event()` 函数中直接处理：

```
bool CustomWidget::event(QEvent *event) {
    if (event->type() == MyCustomEventType) {
        CustomEvent *myEvent = static_cast<CustomEvent *>(event);
        // processing...
        return true;
    }
    return QWidget::event(event);
}
```

## 24 Qt 绘制系统简介

Qt 的绘图系统允许使用相同的 API 在屏幕和其它打印设备上绘制。整个绘图系统基于 `QPainter`，`QPainterDevice` 和 `QPaintEngine` 三个类。

`QPainter` 用来执行绘制的操作；`QPaintDevice` 是一个二维空间的抽象，这个二维空间允许 `QPainter` 在其上面进行绘制，也就是 `QPainter` 工作的空间；`QPaintEngine` 提供了画笔（`QPainter`）在不同的设备上进行绘制的统一的接口。`QPaintEngine` 类应用于 `QPainter` 和 `QPaintDevice` 之间，通常对开发人员是透明的。除非你需要自定义一个设备，否则你是不需要关心 `QPaintEngine` 这个类的。我们可以把 `QPainter` 理解成画笔；把 `QPaintDevice` 理解成使用画笔的地方，比如纸张、屏幕等；而对于纸张、屏幕而言，肯定要使用不同的画

笔绘制，为了统一使用一种画笔，我们设计了 `QPaintEngine` 类，这个类让不同的纸张、屏幕都能使用一种画笔。

下图给出了这三个类之间的层次结构（出自 Qt API 文档）：



上面的示意图告诉我们，Qt 的绘图系统实际上是，使用 `QPainter` 在 `QPainterDevice` 上进行绘制，它们之间使用 `QPaintEngine` 进行通讯（也就是翻译 `QPainter` 的指令）。

下面我们通过一个实例来介绍 `QPainter` 的使用：

```
//!!! Qt4/Qt5

class PaintedWidget : public QWidget
{
    Q_OBJECT
public:
    PaintedWidget(QWidget *parent = 0);
protected:
    void paintEvent(QPaintEvent *);
};
```

注意我们重写了 `QWidget` 的 `paintEvent()` 函数。这或许是在理解了 Qt 事件系统之后首次实际应用。接下来就是 `PaintedWidget` 的源代码：

```
//!!! Qt4/Qt5

PaintedWidget::PaintedWidget(QWidget *parent) :
    QWidget(parent)
{
    resize(800, 600);
    setWindowTitle(tr("Paint Demo"));
}

void PaintedWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(80, 100, 650, 500);
    painter.setPen(Qt::red);
    painter.drawRect(10, 10, 100, 400);
}
```

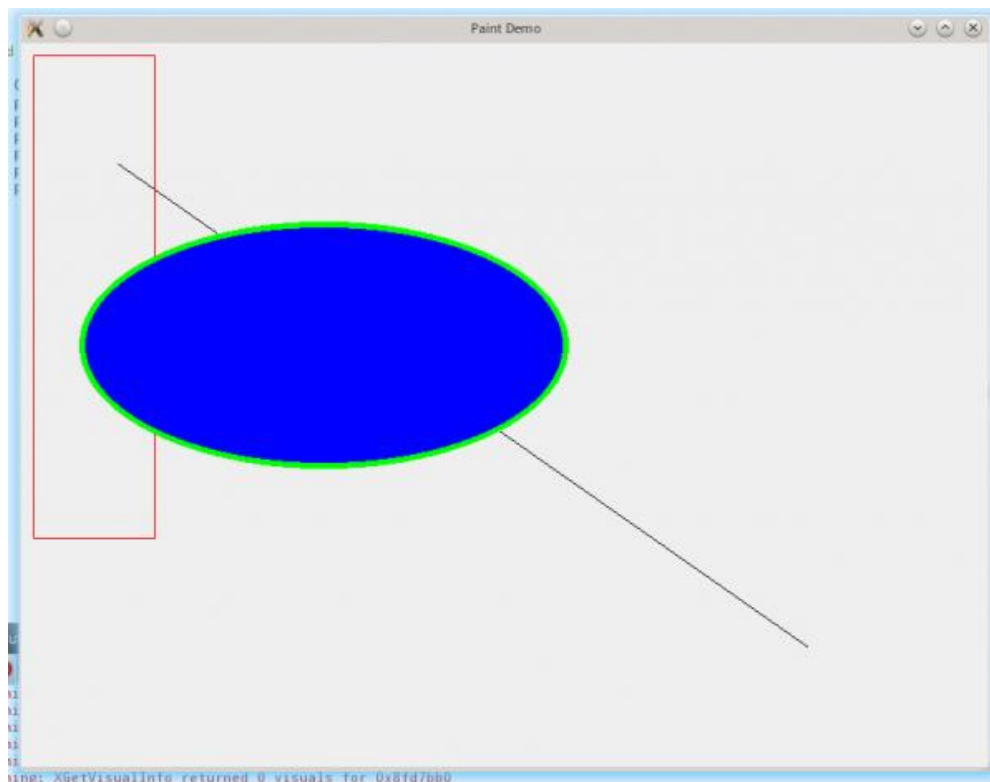
```
painter.setPen(QPen(Qt::green, 5));  
painter.setBrush(Qt::blue);  
painter.drawEllipse(50, 150, 400, 200);  
}
```

在构造函数中，我们仅仅设置了窗口的大小和标题。而 `paintEvent()` 函数则是绘制的代码。首先，我们在栈上创建了一个 `QPainter` 对象，也就是说，每次运行 `paintEvent()` 函数的时候，都会重建这个 `QPainter` 对象。注意，这一点可能会引发某些细节问题：由于我们每次重建 `QPainter`，因此第一次运行时所设置的画笔颜色、状态等，第二次再进入这个函数时就会全部丢失。有时候我们希望保存画笔状态，就必须自己保存数据，否则的话则需要将 `QPainter` 作为类的成员变量。

`QPainter` 接收一个 `QPaintDevice` 指针作为参数。`QPaintDevice` 有很多子类，比如 `QImage`，以及 `QWidget`。注意回忆一下，`QPaintDevice` 可以理解成要在哪里去绘制，而现在我们希望画在这个组件，因此传入的是 `this` 指针。

`QPainter` 有很多以 `draw` 开头的函数，用于各种图形的绘制，比如这里的 `drawLine()`，`drawRect()` 以及 `drawEllipse()` 等。当绘制轮廓线时，使用 `QPainter` 的 `pen()` 属性。比如，我们调用了 `painter.setPen(Qt::red)` 将 `pen` 设置为红色，则下面绘制的矩形具有红色的轮廓线。接下来，我们将 `pen` 修改为绿色，5 像素宽 (`painter.setPen(QPen(Qt::green, 5))`)，又设置了画刷为蓝色。这时候再调用 `draw` 函数，则是具有绿色 5 像素宽轮廓线、蓝色填充的椭圆。

运行一下我们的程序，可以看到最终效果：



我们会在后面的章节详细介绍画笔 `QPen` 和画刷 `QBrush` 的属性。

另外要说明一点，请注意我们的绘制顺序，首先是直线，然后是矩形，最后是椭圆。按照这样的绘制顺序，可以看到直线是第一个绘制，位于最下一层；矩形是第二个绘制，在中间一层；椭圆是最后绘制，在最上层。

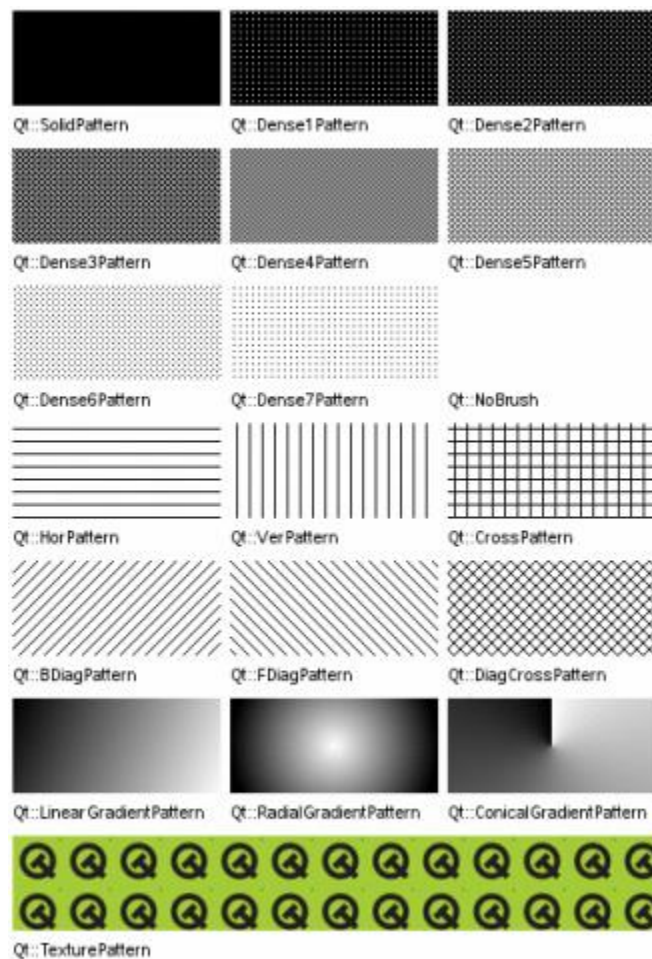
如果了解 `OpenGL`，肯定听说过这么一句话：`OpenGL` 是一个状态机。所谓状态机，就是说，`OpenGL` 保存的只是各种状态。比如，将画笔颜色设置成红色，那么，除非你重新设置另外的颜色，它的颜色会一直是红色。`QPainter` 也是这样，它的状态不会自己恢复，除非你使用了各种设置函数。因此，如果在上面的代码中，我们在椭圆绘制之后再画一个矩形，它的样式还会是绿色 5 像素的轮廓线以及蓝色的填充，除非你显式地调用了设置函数进行状态的更新。这是大多数绘图系统的实现方式，包括 `OpenGL`、`QPainter` 以及 `Java2D`。正因为 `QPainter` 是一个状态机，才会引出我们前面曾经介绍过的一个细节问题：由于 `paintEvent()` 是需要重复进入的，因此，需要注意第二次进入时，`QPainter` 的状态是不是和第一次一致，否则的话可能会造成闪烁的现象。这个闪烁并不是由于双缓冲的问题，而是由于绘制状态的快速切换。

## 25 画刷和画笔

前面一章我们提到，Qt 绘图系统定义了两个绘制时使用的关键属性：画刷和画笔。前者使用 `QBrush` 描述，大多用于填充；后者使用 `QPen` 描述，大多用于绘制轮廓线。

`QBrush` 定义了 `QPainter` 的填充模式，具有样式、颜色、渐变以及纹理等属性。

画刷的 `style()` 定义了填充的样式，使用 `Qt::BrushStyle` 枚举，默认值是 `Qt::NoBrush`，也就是不进行任何填充。我们可以从下面的图示中看到各种填充样式的区别：



画刷的 `color()` 定义了填充模式的颜色。这个颜色可以是 `Qt` 预定义的颜色常量，也就是 `Qt::GlobalColor`，也可以是任意 `QColor` 对象。

画刷的 `gradient()` 定义了渐变填充。这个属性只有在样式是 `Qt::LinearGradientPattern`、`Qt::RadialGradientPattern` 或者 `Qt::ConicalGradientPattern` 之一时才有效。渐变可以由 `QGradient` 对象表示。`Qt` 提供了三种渐变：`QLinearGradient`、`QConicalGradient` 和 `QRadialGradient`，它们都是 `QGradient` 的子类。我们可以使用如下代码片段来定义一个渐变的画刷：

```
QRadialGradient gradient(50, 50, 50, 50, 50);
gradient.setColorAt(0, QColor::fromRgbF(0, 1, 0, 1));
gradient.setColorAt(1, QColor::fromRgbF(0, 0, 0, 0));

QBrush brush(gradient);
```

当画刷样式是 `Qt::TexturePattern` 时，`texture()` 定义了用于填充的纹理。注意，即使你没有设置样式为 `Qt::TexturePattern`，当你调用 `setTexture()` 函数时，`QBrush` 会自动将 `style()` 设置为 `Qt::TexturePattern`。

`QPen` 定义了用于 `QPainter` 应该怎样画线或者轮廓线。画笔具有样式、宽度、画刷、笔帽样式和连接样式等属性。画笔的样式 `style()` 定义了线的样式。画刷 `brush()` 用于填充画笔所绘制的线条。笔帽样式 `capStyle()` 定义了使用 `QPainter` 绘制的线的末端；连接样式 `joinStyle()` 则定义了两条线如何连接起来。画笔宽度 `width()` 或 `widthF()` 定义了画笔的宽。注意，不存在宽度为 0 的线。假设你设置 `width` 为 0，`QPainter` 依然会绘制出一条线，而这个线的宽度为 1 像素。也就是说，画笔宽度通常至少是 1 像素。

这么多参数既可以在构造时指定，也可以使用 `set` 函数指定，完全取决于你的习惯，例如：

```
QPainter painter(this);
QPen pen(Qt::green, 3, Qt::DashDotLine, Qt::RoundCap, Qt::RoundJoin);
painter.setPen(pen);
等价于
QPainter painter(this);
QPen pen; // creates a default pen

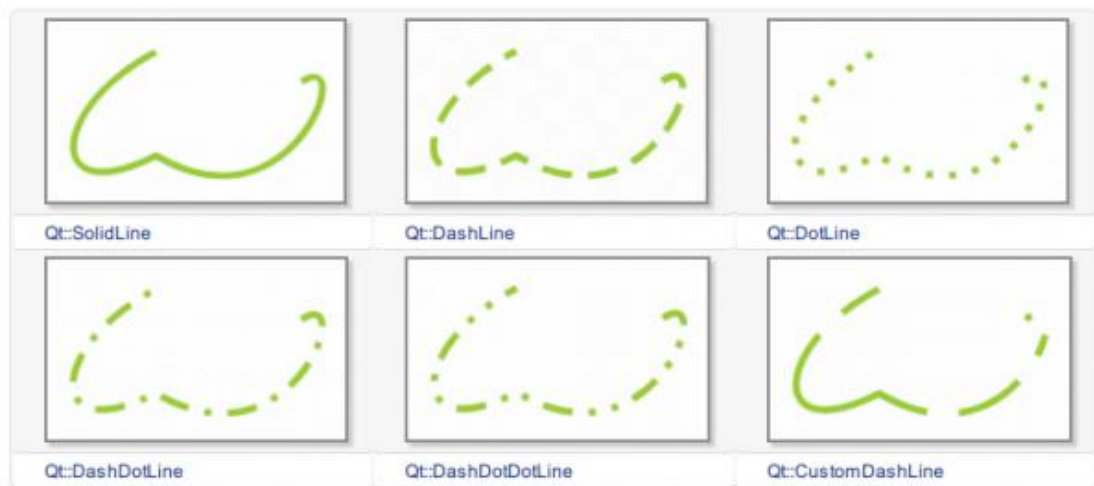
pen.setStyle(Qt::DashDotLine);
pen.setWidth(3);
pen.setBrush(Qt::green);
pen.setCapStyle(Qt::RoundCap);
pen.setJoinStyle(Qt::RoundJoin);

painter.setPen(pen);
```

使用构造函数的优点是代码较短，但是参数含义不明确；使用 `set` 函数则正好反过来。

默认的画笔属性是纯黑色，0 像素，方形笔帽（`Qt::SquareCap`），斜面型连接（`Qt::BevelJoin`）。

下面是画笔样式的示例：



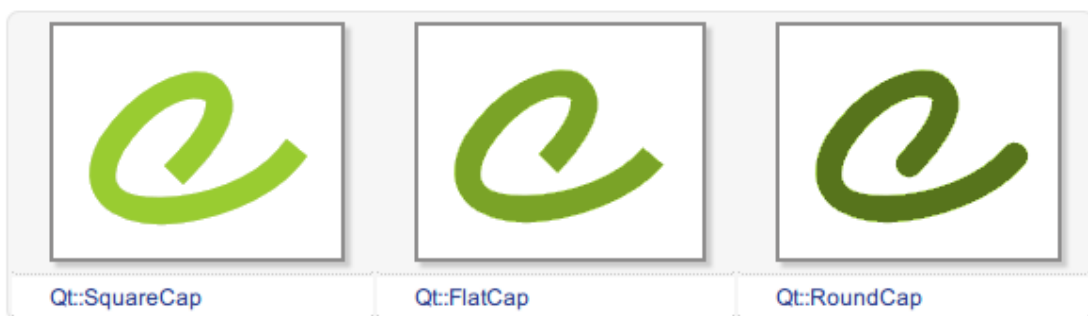
你也可以使用 `setDashPattern()` 函数自定义样式，例如如下代码片段：

```
QPen pen;
QVector<qreal> dashes;
qreal space = 4;

dashes << 1 << space << 3 << space << 9 << space
      << 27 << space << 9 << space;

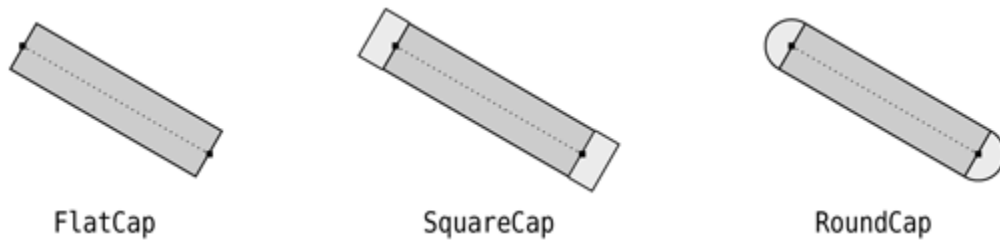
pen.setDashPattern(dashes);
```

笔帽定义了画笔末端的样式，例如：

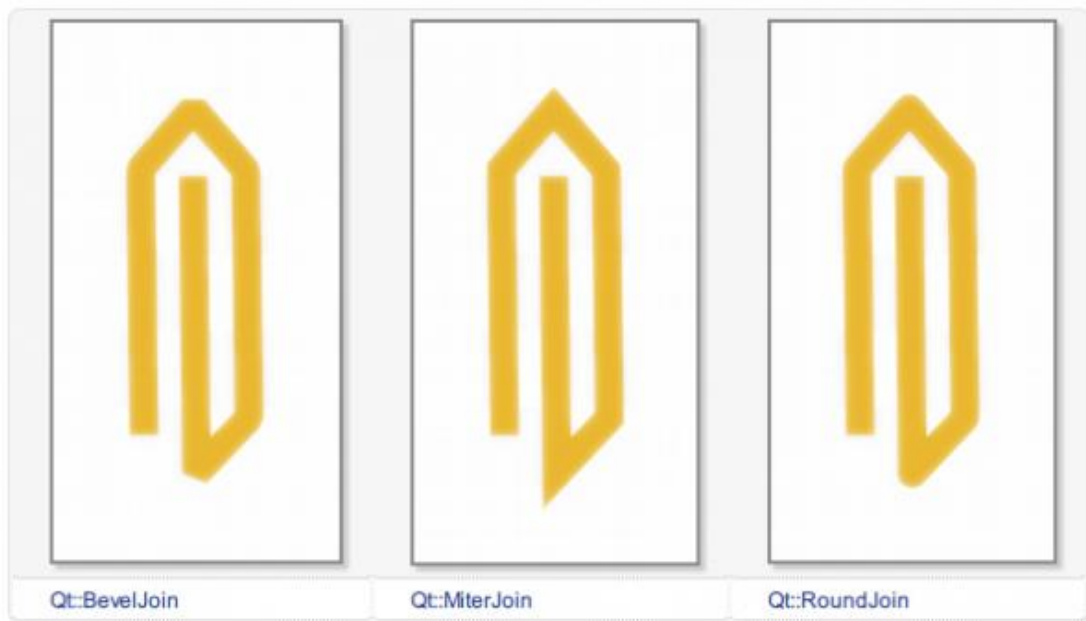


他们之间的区别是，`Qt::SquareCap` 是一种包含了最后一个点的方形端点，使用半个线宽覆盖；`Qt::FlatCap` 不包含最后一个点；`Qt::RoundCap` 是包含最后一个点的圆形端点。具体可以参考下面的示例（出自《C++ GUI Programming with Qt 4, 2nd Edition》）：





连接样式定义了两条线连接时的样式，例如：



同样，可以参考下面图示来理解这几种连接样式的细节（出自《C++ GUI Programming with Qt 4, 2nd Edition》）：



注意，我们前面说了，`QPainter` 也是一个状态机，这里我们所说的这些属性都是处于这个状态机之中的，因此，我们应该记得是否要将其保存下来或者是重新构建。

## 26 反走样

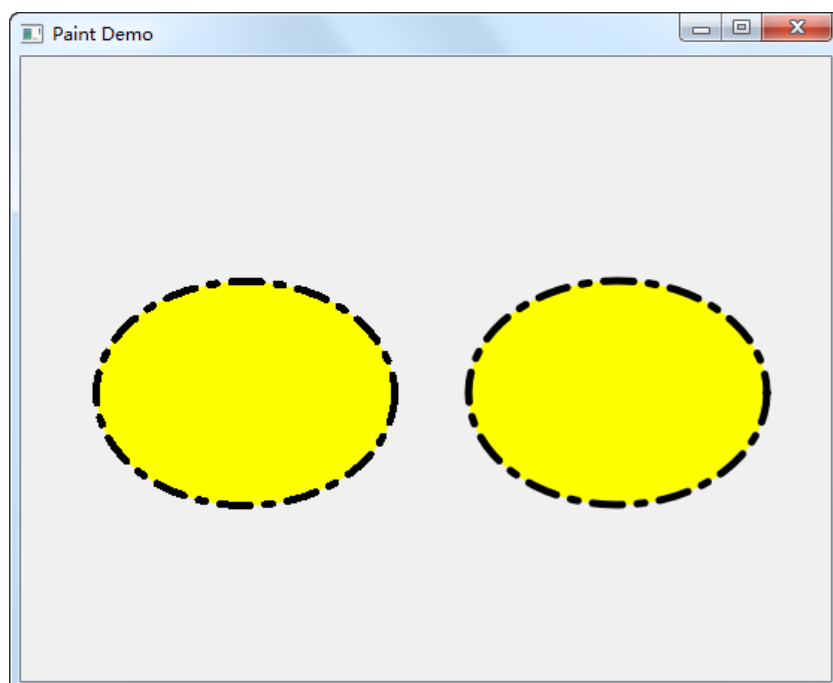
我们在光栅图形显示器上绘制非水平、非垂直的直线或多边形边界时，或多或少会呈现锯齿状外观。这是因为直线和多边形的边界是连续的，而光栅则是由离散的点组成。在光栅显示设备上表现直线、多边形等，必须在离散位置采样。由于采样不充分重建后造成的信息失真，就叫走样；用于减少或消除这种效果的技术，就称为反走样。

反走样是图形学中的重要概念，用以防止通常所说的“锯齿”现象的出现。很多系统的绘图 API 里面都内置了有关反走样的算法，不过由于性能问题，默认一般是关闭的，Qt 也不例外。下面我们来看看代码：

```
void paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(QPen(Qt::black, 5, Qt::DashDotLine, Qt::RoundCap));
    painter.setBrush(Qt::yellow);
    painter.drawEllipse(50, 150, 200, 150);

    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setPen(QPen(Qt::black, 5, Qt::DashDotLine, Qt::RoundCap));
    painter.setBrush(Qt::yellow);
    painter.drawEllipse(300, 150, 200, 150);
}
```

看看运行后的结果：



注意看左侧椭圆与右侧椭圆在边界处的区别。左侧没有使用反锯齿，明显显示出锯齿的样子；右侧则增加了反锯齿代码。

在这段代码中，我们创建了一个黑色 5 像素宽的画笔，使用了点线的样式，圆形笔帽：

```
painter.setPen(QPen(Qt::black, 5, Qt::DashDotLine, Qt::RoundCap));
```

然后我们使用一个黄色的画刷填充，绘制一个椭圆。

第二个椭圆的绘制与第一个十分相似，唯一的区别是多了一句

```
painter.setRenderHint(QPainter::Antialiasing, true);
```

显然，我们通过这条语句，将 **Antialiasing** 属性（也就是反走样）设置为 **true**。经过这句设置，我们就打开了 **QPainter** 的反走样功能。还记得我们曾经说过，**QPainter** 是一个状态机，因此，只要这里我们打开了它，之后所有的代码都会是反走样绘制的了。由于反走样需要比较复杂的算法，在一些对图像质量要求不是很高的应用中，是不需要进行反走样的。为了提高效率，一般的图形绘制系统，如 **Java2D**、**OpenGL** 之类都是默认不进行反走样的。

虽然反走样比不反走样的图像质量高很多，但是，没有反走样的图形绘制还是有很大用处的。首先，就像前面说的一样，在一些对图像质量要求不高的环境下，或者说性能受限的环境下，比如嵌入式和手机环境，一般是不进行反走样的。另外，在一些必须精确操作像素的应用中，也是不能进行反走样的。这是由于反走样技术本身的限制的。请看下面的图片：



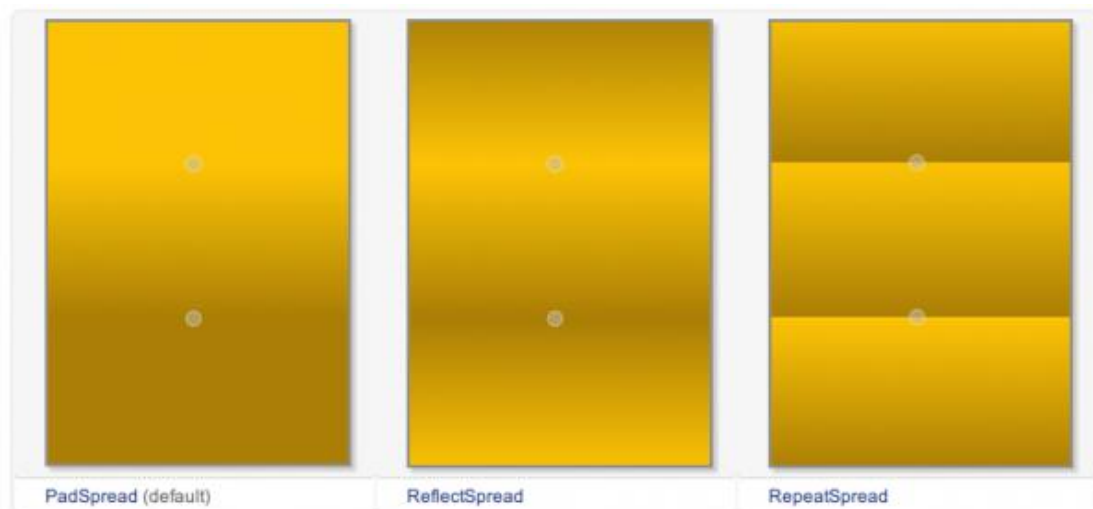
这是使用 **Photoshop** 的铅笔和画笔工具绘制的 1 像素的点，放大 3200% 的视图。在一定程度上，我们可以认为，**Photoshop** 的铅笔工具是不进行反走样，而画笔是要进行反走样的。在放大的情况下就会知道，有反走样的情况下是不能进行精确到 1 像素的操作的。因为反走样很难让你控制到 1 个像素。这不是 **Photoshop** 画笔工具的缺陷，而是反走样算法的问题。反走样之所以看起来比较模糊，就是因为它需要以一种近似色来替换原始的像素色，这样一来就会显得模糊而圆滑。

## 27 渐变

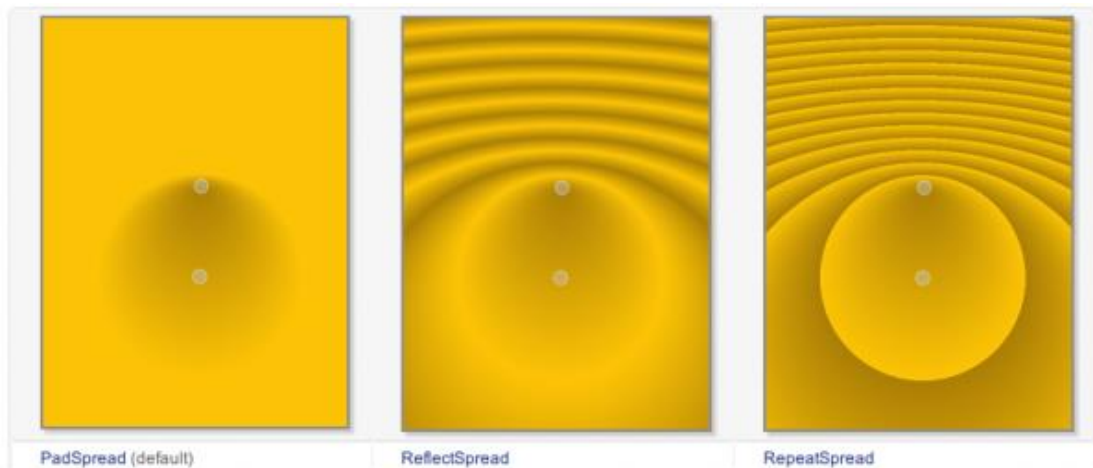
渐变是绘图中很常见的一种功能，简单来说就是可以把几种颜色混合在一起，让它们能够自然地过渡，而不是一下子变成另一种颜色。渐变的算法比较复杂，写得不好效率会很低，好在很多绘图系统都内置了渐变的功能，Qt 也不例外。渐变一般是用在填充里面的，所以，设置渐变是在 `QBrush` 里面。

Qt 提供了三种渐变：线性渐变（`QLinearGradient`）、辐射渐变（`QRadialGradient`）和角度渐变（`QConicalGradient`）。我们可以在 Qt API 手册中看到这几种渐变的区别：

线性渐变：



辐射渐变：



角度渐变：



具体细节可以参考文档。下面我们通过一个示例看看如何使用渐变进行填充：

```
void paintEvent(QPaintEvent *)
{
    QPainter painter(this);

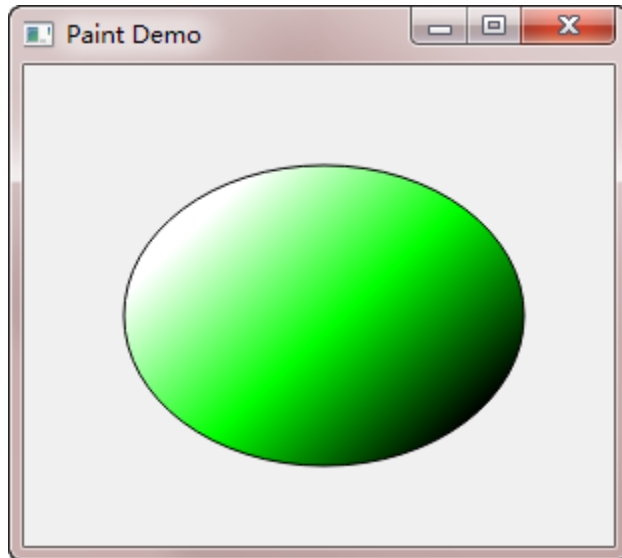
    painter.setRenderHint(QPainter::Antialiasing, true);
    QLinearGradient linearGradient(60, 50, 200, 200);
    linearGradient.setColorAt(0.2, Qt::white);
    linearGradient.setColorAt(0.6, Qt::green);
    linearGradient.setColorAt(1.0, Qt::black);
    painter.setBrush(QBrush(linearGradient));
    painter.drawEllipse(50, 50, 200, 150);
}
```

像以前一样，我们也只给出了 `paintEvent()` 的代码。这段代码看起来也相当清晰：首先我们打开了反走样，然后创建一个 `QLinearGradient` 对象实例。`QLinearGradient` 也就是线性渐变，其构造函数有四个参数，分别是 `x1`, `y1`, `x2`, `y2`，即渐变的起始点和终止点。在这里，我们从 `(60, 50)` 点开始渐变，到 `(200, 200)` 点止。关于坐标的具体细节，我们会在后面的章节中详细介绍。渐变的颜色是在 `setColorAt()` 函数中指定的。下面是这个函数的签名：

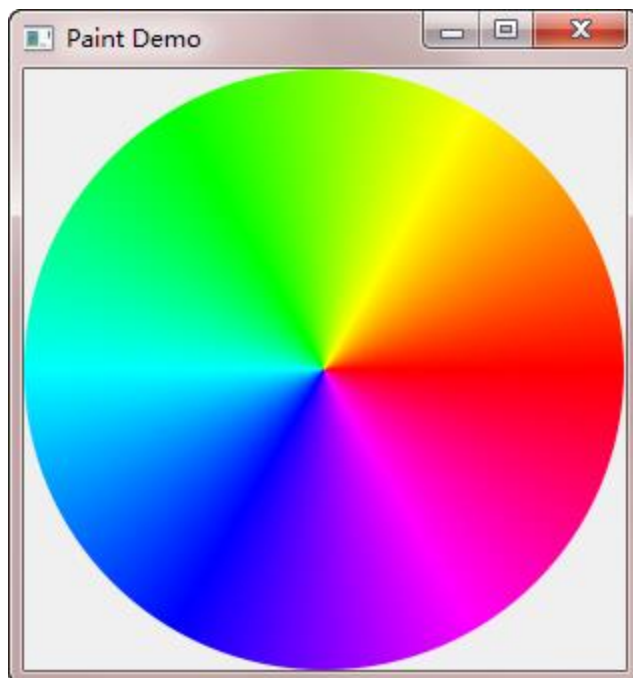
```
void QGradient::setColorAt ( qreal position, const QColor & color )
```

这个函数的作用是，把 `position` 位置的颜色设置成 `color`。其中，`position` 是一个 `[0, 1]` 闭区间的数字。也就是说，`position` 是相对于我们建立渐变对象时做的那个起始点和终止点区

间的一个比例。以这个线性渐变为例，在从 (60, 50) 到 (200, 200) 的线段上，在 0.2，也就五分之一处设置成白色，在 0.6 也就是五分之三处设置成绿色，在 1.0 也就是终点处设置成黑色。创建 `QBrush` 对象时，把这个渐变对象传递进去，然后就可以运行了：



下面我们开始一个更复杂，也更实用一些的例子：绘制一个色轮（color wheel）。所谓色轮，其实就是一个带有颜色的圆盘（或许你没听说过这个名字，但是你肯定见过这个东西），下面是色轮的运行结果：



我们来看看它的代码：

```
void ColorWheel::paintEvent(QPaintEvent *)
```

```

{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);

    const int r = 150;
    QConicalGradient conicalGradient(0, 0, 0);

    conicalGradient.setColorAt(0.0, Qt::red);
    conicalGradient.setColorAt(60.0/360.0, Qt::yellow);
    conicalGradient.setColorAt(120.0/360.0, Qt::green);
    conicalGradient.setColorAt(180.0/360.0, Qt::cyan);
    conicalGradient.setColorAt(240.0/360.0, Qt::blue);
    conicalGradient.setColorAt(300.0/360.0, Qt::magenta);
    conicalGradient.setColorAt(1.0, Qt::red);

    painter.translate(r, r);

    QBrush brush(conicalGradient);
    painter.setPen(Qt::NoPen);
    painter.setBrush(brush);
    painter.drawEllipse(QPoint(0, 0), r, r);
}

```

首先还是新建 `QPainter` 对象，开启反走样。然后将圆盘半径定义为 150。下面创建一个角度渐变实例，其构造函数同样接受三个参数：

```

QConicalGradient::QConicalGradient ( qreal cx, qreal cy, qreal angle )

```

前两个参数 `cx` 和 `cy` 组成角度渐变的中心点，第三个参数是渐变的起始角度。在我们的例子中，我们将渐变中心点设置为 `(0, 0)`，起始角度为 0。类似线性渐变，角度渐变的 `setColorAt()` 函数同样接受两个参数，第一个是角度比例，第二个是颜色。例如，

```

conicalGradient.setColorAt(0.0, Qt::red);

```

将 0 度角设置为红色；

```

conicalGradient.setColorAt(60.0/360.0, Qt::yellow);

```

将 60 度角设置为黄色。由于一个圆周是 360 度，所以 `60.0/360.0` 即是这个角度的比例。其余代码以此类推。最后一句，我们将 1.0 处设置为红色，也就是重新回到起始处。至于颜色的分布，这是由颜色空间定义的，有兴趣的朋友可以查阅有关颜色模型的理论。

```
painter.translate(r, r);
```

这是我们唯一不熟悉的函数。QPainter::translate(x, y)函数意思是，将坐标系的原点设置到(x, y)点。原本坐标系原点位于左上角，我们使用 translate(r, r)，将坐标原点设置为 (r, r)。这么一来，左上角的点的坐标就应该是 (-r, -r)。

最后，我们使用 drawEllipse()函数绘制圆盘。注意，由于我们已经把坐标原点设置为 (r, r)，因此，在绘制时，圆心应该是新的坐标 (0, 0)，而不是原来的 (r, r)。

PS：为了理解 translate()函数的作用，可以思考下，如果去掉 translate()函数的调用，我们的程序应该如何修改。答案是：

```
void ColorWheel::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);

    const int r = 150;
    QConicalGradient conicalGradient(r, r, 0);

    conicalGradient.setColorAt(0.0, Qt::red);
    conicalGradient.setColorAt(60.0/360.0, Qt::yellow);
    conicalGradient.setColorAt(120.0/360.0, Qt::green);
    conicalGradient.setColorAt(180.0/360.0, Qt::cyan);
    conicalGradient.setColorAt(240.0/360.0, Qt::blue);
    conicalGradient.setColorAt(300.0/360.0, Qt::magenta);
    conicalGradient.setColorAt(1.0, Qt::red);

    QBrush brush(conicalGradient);
    painter.setPen(Qt::NoPen);
    painter.setBrush(brush);
    painter.drawEllipse(QPoint(r, r), r, r);
}
```

不仅我们需要修改最后的绘制语句，还需要注意修改 QConicalGradient 定义时传入的中心点的坐标。

## 28 坐标系统



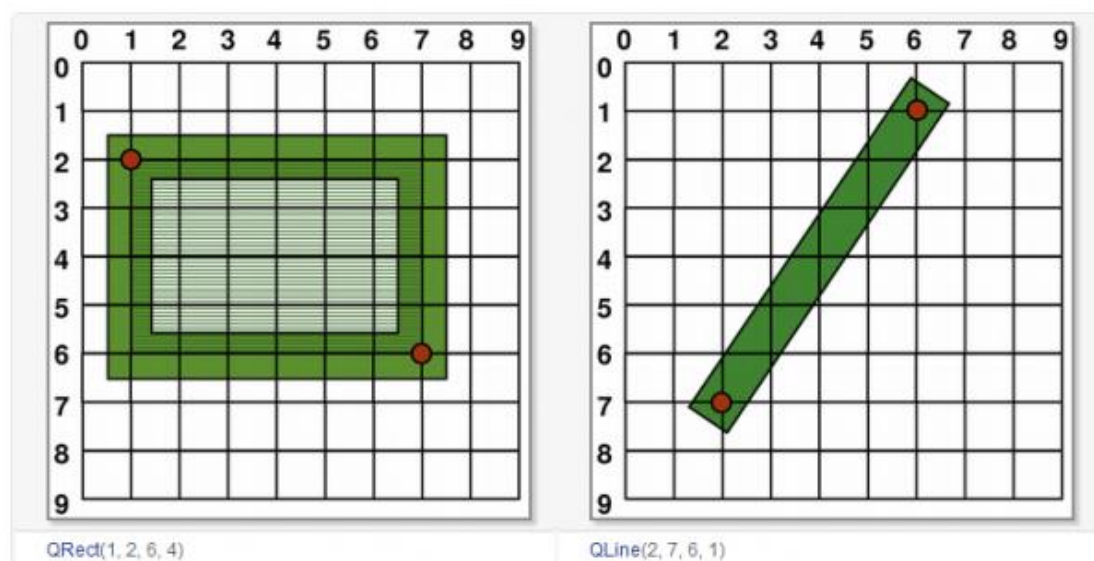
在经历过实际操作，以及前面一节中我们见到的那个 `translate()` 函数之后，我们可以详细了解 Qt 的坐标系统了。泛泛而谈坐标系统，有时候会觉得枯燥无味，难以理解，好在现在我们已经有了基础。

坐标系统是由 `QPainter` 控制的。我们前面说过，`QPaintDevice`、`QPaintEngine` 和 `QPainter` 是 Qt 绘制系统的三个核心类。`QPainter` 用于进行绘制的实际操作；`QPaintDevice` 是那些能够让 `QPainter` 进行绘制的“东西”（准确的术语叫做，二维空间）的抽象层（其子类有 `QWidget`、`QPixmap`、`QPicture`、`QImage` 和 `QPrinter` 等）；`QPaintEngine` 提供供 `QPainter` 使用的用于在不同设备上绘制的统一的接口。

由于 `QPaintDevice` 是进行绘制的对象，因此，所谓坐标系统，也就是 `QPaintDevice` 上面的坐标。默认坐标系统位于设备的左上角，也就是坐标原点  $(0, 0)$ 。x 轴方向向右；y 轴方向向下。在基于像素的设备上（比如显示器），坐标的默认单位是像素，在打印机上则是点（ $1/72$  英寸）。

将 `QPainter` 的逻辑坐标与 `QPaintDevice` 的物理坐标进行映射的工作，是由 `QPainter` 的变换矩阵（transformation matrix）、视口（viewport）和窗口（window）完成的。如果你不理解这些术语，可以简单了解下有关图形学的内容。实际上，对图形的操作，底层的数学都是进行的矩阵变换、相乘等运算。

在 Qt 的坐标系统中，每个像素占据  $1 \times 1$  的空间。你可以把它想象成一张方格纸，每个小格都是 1 个像素。方格的焦点定义了坐标，也就是说，像素  $(x, y)$  的中心位置其实是在  $(x + 0.5, y + 0.5)$  的位置上。这个坐标系统实际上是一个“半像素坐标系”。我们可以通过下面的示意图来理解这种坐标系：

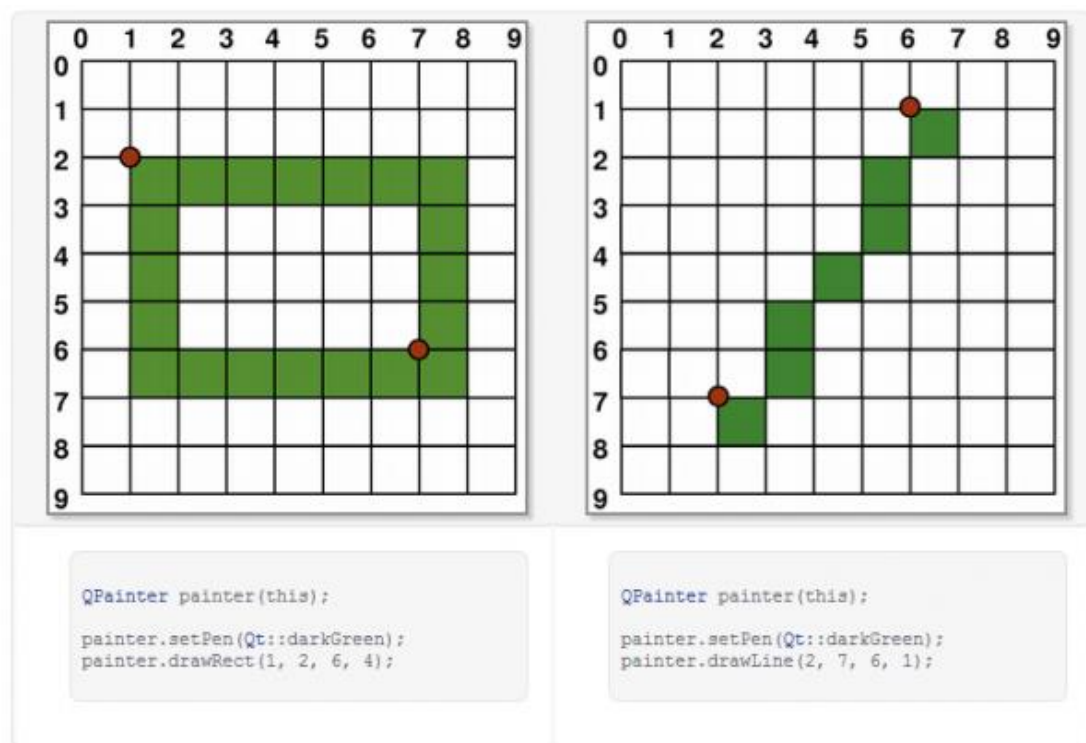


我们使用一个像素的画笔进行绘制，可以看到，每一个绘制像素都是以坐标点为中心的矩形。注意，这是坐标的逻辑表示，实际绘制则与此不同。因为在实际设备上，像素是最小单位，

我们不能像上面一样，在两个像素之间进行绘制。所以在实际绘制时，Qt 的定义是，绘制点所在像素是逻辑定义点的右下方的像素。

我们前面已经介绍过，Qt 的绘制分为走样和反走样两种。对此，我们必须分别对待。

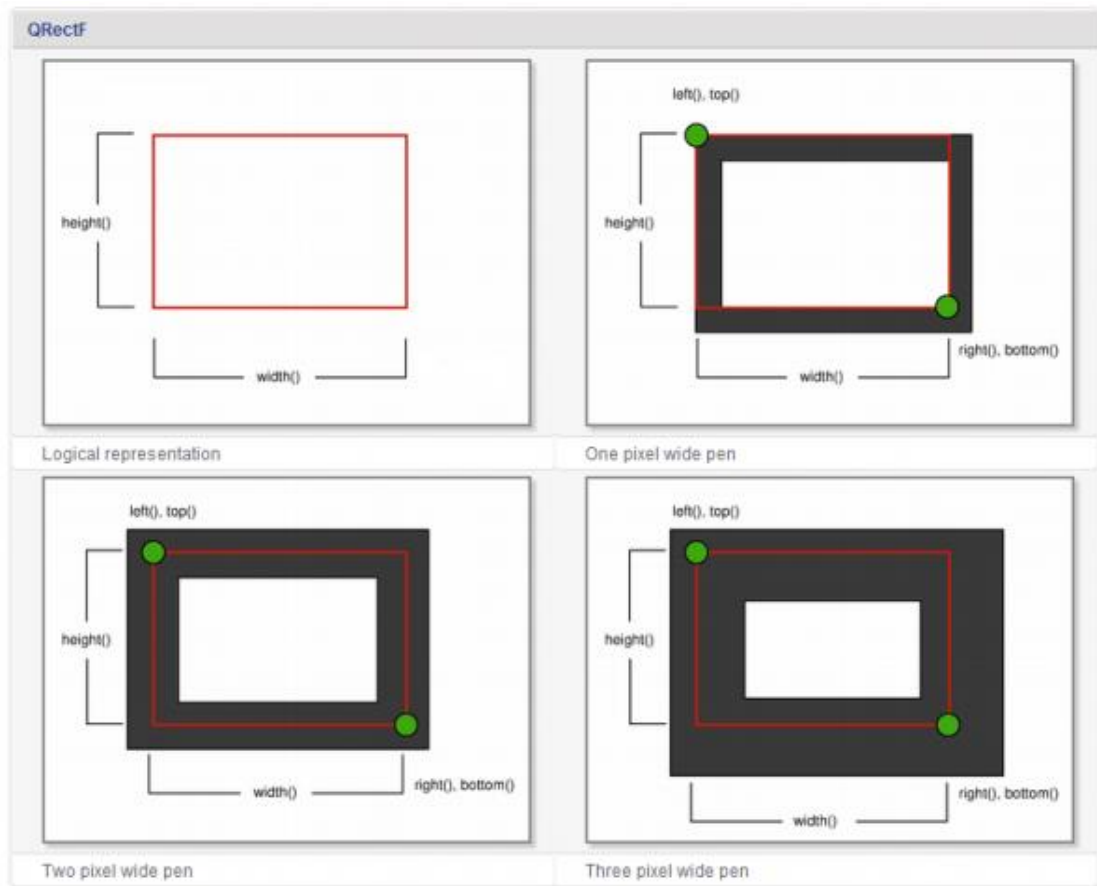
一个像素的绘制最简单，我们从这里开始：



从上图可以看出，当我们绘制矩形左上角 (1, 2) 时，实际绘制的像素是在右下方。

当绘制大于 1 个像素时，情况比较复杂：如果绘制像素是偶数，则实际绘制会包裹住逻辑坐标值；如果是奇数，则是包裹住逻辑坐标值，再加上右下角一个像素的偏移。具体请看下面

的图示：



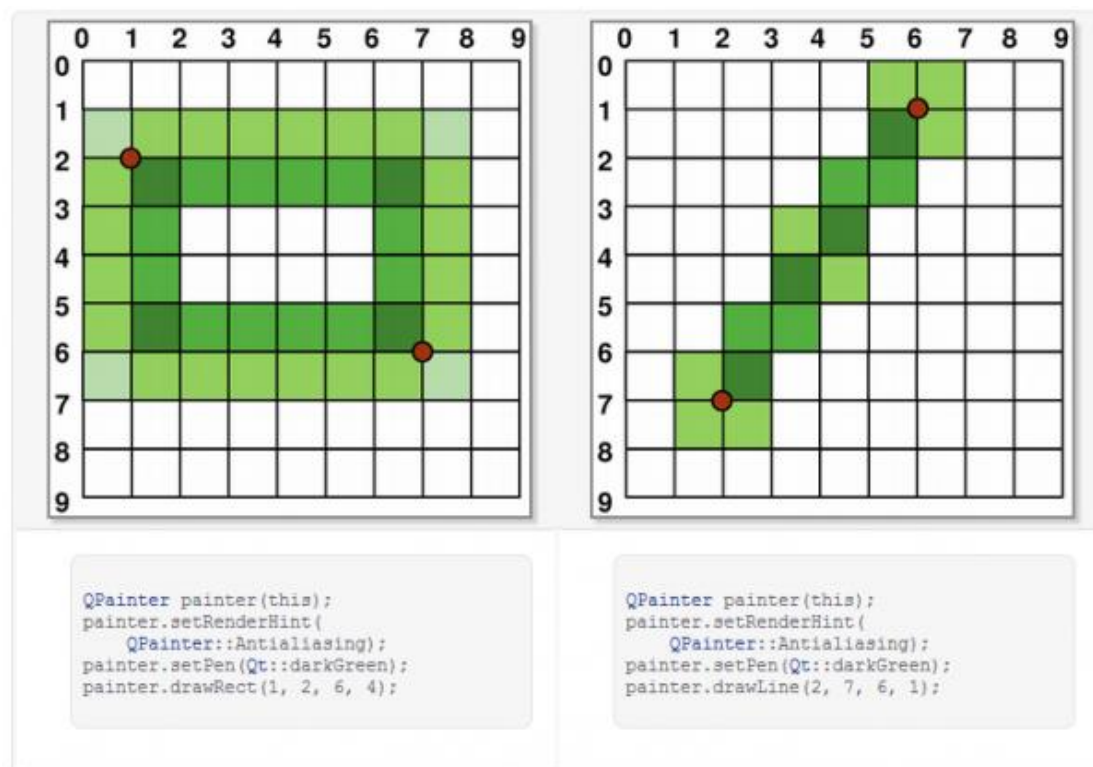
从上图可以看出，如果实际绘制是偶数像素，则会将逻辑坐标值夹在相等的两部分像素之间；如果是奇数，则会在右下方多出一个像素。

Qt 的这种处理，带来的一个问题是，我们可能获取不到真实的坐标值。由于历史原因，`QRect::right()`和 `QRect::bottom()`的返回值并不是矩形右下角点的真实坐标值：

`QRect::right()`返回的是  $\text{left()} + \text{width()} - 1$ ；`QRect::bottom()`则返回  $\text{top()} + \text{height()} - 1$ ，上图的绿色点指出了这两个函数的返回点的坐标。

为避免这个问题，我们建议是使用 `QRectF`。`QRectF` 使用浮点值，而不是整数值，来描述坐标。这个类的两个函数 `QRectF::right()`和 `QRectF::bottom()`是正确的。如果你不得不使用 `QRect`，那么可以利用 `x() + width()` 和 `y() + height()` 来替代 `right()` 和 `bottom()` 函数。

对于反走样，实际绘制会包裹住逻辑坐标值：



这里我们不去解释为什么在反走样是，像素颜色不是一致的，这是由于反走样算法导致，已经超出本节的内容。

Qt 同样提供了坐标变换。前面说，图形学大部分算法依赖于矩阵计算，坐标变换便是其中的代表：每一种变换都对应着一个矩阵乘法（如果你想知道学的线性代数有什么用处，这就是应用之一了 ;-P）。我们会以一个实际的例子来了解坐标变换。在此之前，我们需要了解两个函数：QPainter::save()和 QPainter::restore()。

前面说过，QPainter 是一个状态机。那么，有时我想保存下当前的状态：当我临时绘制某些图像时，就可能想这么做。当然，我们有最原始的办法：将可能改变的状态，比如画笔颜色、粗细等，在临时绘制结束之后全部恢复。对此，QPainter 提供了内置的函数：save()和 restore()。save()就是保存下当前状态；restore()则恢复上一次保存的结果。这两个函数必须成对出现：QPainter 使用栈来保存数据，每一次 save()，将当前状态压入栈顶，restore()则弹出栈顶进行恢复。

在了解了这两个函数之后，我们就可以进行示例代码了：

```
void PaintDemo::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
```

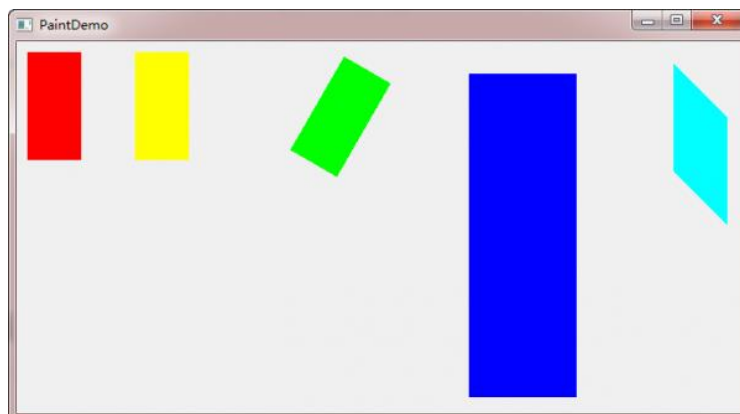
```

painter.fillRect(10, 10, 50, 100, Qt::red);
painter.save();
painter.translate(100, 0); // 向右平移 100px
painter.fillRect(10, 10, 50, 100, Qt::yellow);
painter.restore();
painter.save();
painter.translate(300, 0); // 向右平移 300px
painter.rotate(30); // 顺时针旋转 30 度
painter.fillRect(10, 10, 50, 100, Qt::green);
painter.restore();
painter.save();
painter.translate(400, 0); // 向右平移 400px
painter.scale(2, 3); // 横坐标单位放大 2 倍, 纵坐标放大 3 倍
painter.fillRect(10, 10, 50, 100, Qt::blue);
painter.restore();
painter.save();
painter.translate(600, 0); // 向右平移 600px
painter.shear(0, 1); // 横向不变, 纵向扭曲 1 倍
painter.fillRect(10, 10, 50, 100, Qt::cyan);
painter.restore();
}

```

Qt 提供了四种坐标变换：平移 `translate`，旋转 `rotate`，缩放 `scale` 和扭曲 `shear`。在这段代码中，我们首先在 (10, 10) 点绘制一个红色的 50×100 矩形。保存当前状态，将坐标系平移到 (100, 0)，绘制一个黄色的矩形。注意，`translate()` 操作平移的是坐标系，不是矩形。因此，我们还是在 (10, 10) 点绘制一个 50×100 矩形，现在，它跑到了右侧的位置。然后恢复先前状态，也就是把坐标系重新设为默认坐标系（相当于进行 `translate(-100, 0)`），再进行下面的操作。之后也是类似的。由于我们只是保存了默认坐标系的状态，因此我们之后的 `translate()` 横坐标值必须增加，否则就会覆盖掉前面的图形。所有这些操作都是针对坐标系的，因此在绘制时，我们提供的矩形的坐标参数都是不变的。

运行结果如下：



Qt 的坐标分为逻辑坐标和物理坐标。在我们绘制时，提供给 `QPainter` 的都是逻辑坐标。之前我们看到的坐标变换，也是针对逻辑坐标的。所谓物理坐标，就是绘制底层 `QPaintDevice` 的坐标。单单只有逻辑坐标，我们是不能在设备上绘制的。要想在设备上绘制，必须提供设备认识的物理坐标。Qt 使用 `viewport-window` 机制将我们提供的逻辑坐标转换成绘制设备使用的物理坐标，方法是，在逻辑坐标和物理坐标之间提供一层“窗口”坐标。视口是由任意矩形指定的物理坐标；窗口则是该矩形的逻辑坐标表示。默认情况下，物理坐标和逻辑坐标是一致的，都等于设备矩形。

视口坐标（也就是物理坐标）和窗口坐标是一个简单的线性变换。比如一个 `400×400` 的窗口，我们添加如下代码：

```
void PaintDemo::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setWindow(0, 0, 200, 200);
    painter.fillRect(0, 0, 200, 200, Qt::red);
}
```

我们将窗口矩形设置为左上角坐标为 `(0, 0)`，长和宽都是 `200px`。此时，坐标原点不变，还是左上角，但是，对于原来的 `(400, 400)` 点，新的窗口坐标是 `(200, 200)`。我们可以理解成，逻辑坐标被“重新分配”。这有点类似于 `translate()`，但是，`translate()` 函数只是简单地将坐标原点重新设置，而 `setWindow()` 则是将整个坐标系进行了修改。这段代码的运行结果是将整个窗口进行了填充。

试比较下面两行代码的区别（还是 `400×400` 的窗口）：

```
painter.translate(200, 200);
painter.setWindow(-160, -320, 320, 640);
```

第一行代码，我们将坐标原点设置到 `(200, 200)` 处，横坐标范围是 `[-200, 200]`，纵坐标范围是 `[-200, 200]`。第二行代码，坐标原点也是在窗口正中心，但是，我们将物理宽 `400px` 映射成窗口宽 `320px`，物理高 `400px` 映射成窗口高 `640px`，此时，横坐标范围是 `[-160, 160]`，纵坐标范围是 `[-320, 320]`。这种变换是简单的线性变换。假设原来有个点坐标是 `(64, 60)`，那么新的窗口坐标下对应的坐标应该是 `((-160 + 64 * 320 / 400), (-320 + 60 * 640 / 400)) = (-108.8, -224)`。

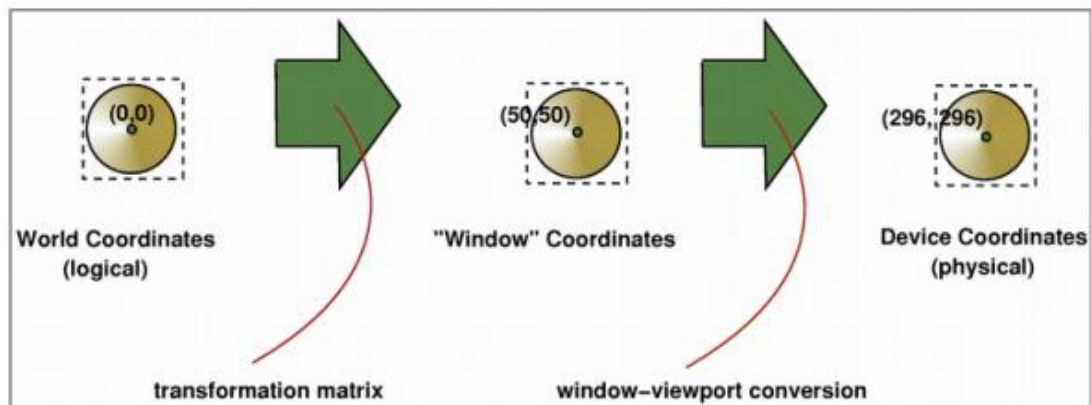
下面我们再来理解下视口的含义。还是以一段代码为例：

```
void PaintDemo::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setViewport(0, 0, 200, 200);
```

```
painter.fillRect(0, 0, 200, 200, Qt::red);  
}
```

这段代码和前面一样，只是把 `setWindow()` 换成了 `setViewport()`。前面我们说过，`window` 代表窗口坐标，`viewport` 代表物理坐标。也就是说，我们将物理坐标区域定义为左上角位于  $(0, 0)$ ，长高都是 `200px` 的矩形。然后还是绘制和上面一样的矩形。如果你认为运行结果是  $1/4$  窗口被填充，那就错了。实际是只有  $1/16$  的窗口被填充。这是由于，我们修改了物理坐标，但是没有修改相应的窗口坐标。默认的逻辑坐标范围是左上角坐标为  $(0, 0)$ ，长宽都是 `400px` 的矩形。当我们将物理坐标修改为左上角位于  $(0, 0)$ ，长高都是 `200px` 的矩形时，窗口坐标范围不变，也就是说，我们将物理宽 `200px` 映射成窗口宽 `400px`，物理高 `200px` 映射成窗口高 `400px`，所以，原始点  $(200, 200)$  的坐标变成了  $((0 + 200 * 200 / 400), (0 + 200 * 200 / 400)) = (100, 100)$ 。

现在我们可以用一张图示总结一下逻辑坐标、窗口坐标和物理坐标之间的关系：



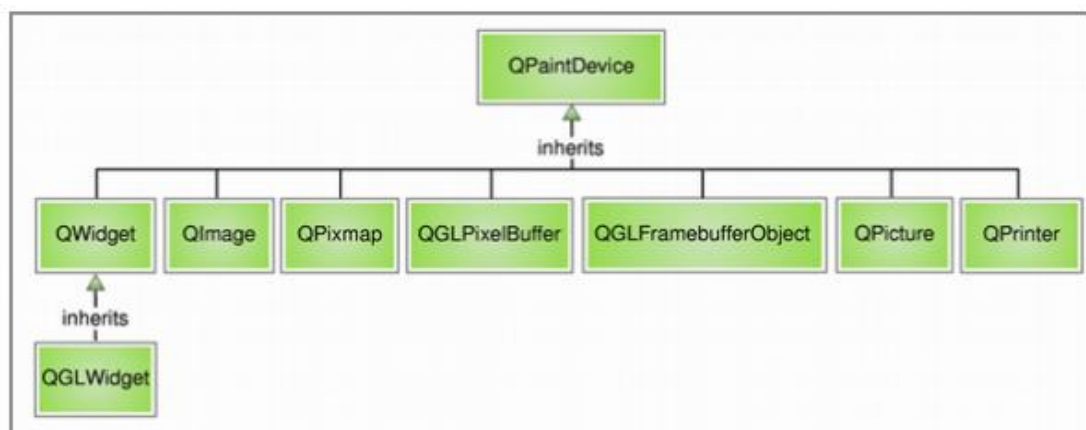
我们传给 `QPainter` 的是逻辑坐标（也称为世界坐标），逻辑坐标可以通过变换矩阵转换成窗口坐标，窗口坐标通过 `window-viewport` 转换成物理坐标（也就是设备坐标）。

## 29 绘制设备

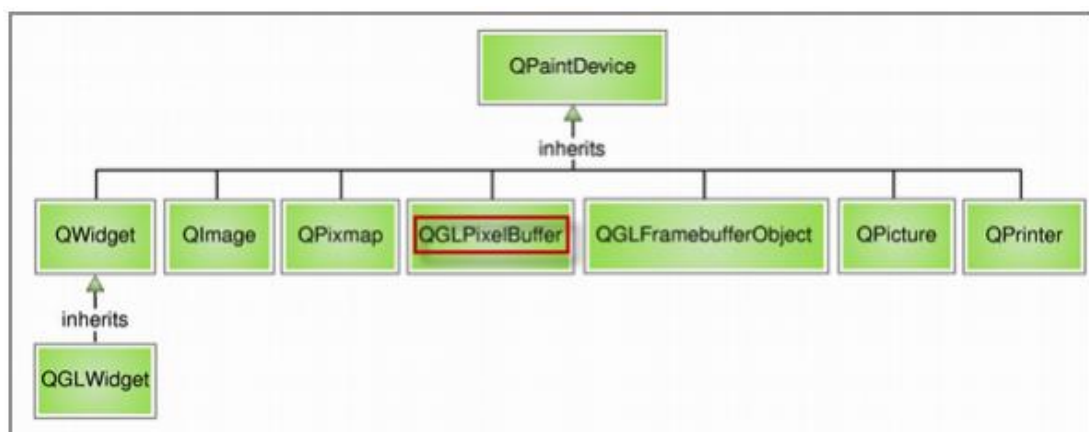
绘图设备是继承 `QPainterDevice` 的类。`QPaintDevice` 就是能够进行绘制的类，也就是说，`QPainter` 可以在任何 `QPaintDevice` 的子类上进行绘制。现在，Qt 提供了若干这样的类：

Qt4:





Qt5:



上面的是 Qt4 的相关类图，下面是 Qt5。这两部分大致相同，只是在 Qt5 中，QGLPixelBuffer 已经被废弃。本章我们关注的是 QPixmap、QBitmap、QImage 和 QPicture 这几个类。另外的部分，QWidget 就是所有组件的父类，我们已经在前面的章节中使用过，这里不再赘述。QGLWidget 和 QGLFramebufferObject，顾名思义，就是关于 OpenGL 的相关类。在 Qt 中，我们可以方便地结合 OpenGL 进行绘制。由于这部分需要牵扯到 OpenGL 的相关内容，现在也不再深入。在我们选择的几个类中，大多与图像密切相关。

QPixmap 专门为图像在屏幕上的显示做了优化；QBitmap 是 QPixmap 的一个子类，它的色深限定为 1，你可以使用 QPixmap 的 isQBitmap() 函数来确定这个 QPixmap 是不是一个 QBitmap。QImage 专门为图像的像素级访问做了优化。QPicture 则可以记录和重现 QPainter 的各条命令。下面我们将分两部分介绍这四种绘图设备。

QPixmap 继承了 QPaintDevice，因此，你可以使用 QPainter 直接在上面绘制图形。QPixmap 也可以接受一个字符串作为一个文件的路径来显示这个文件，比如你想在程序之中打开 png、jpeg 之类的文件，就可以使用 QPixmap。使用 QPainter::drawPixmap() 函数可以把这个文件绘制到一个 QLabel、QPushButton 或者其他设备上面。正如前面所



说的那样，QPixmap 是针对屏幕进行特殊优化的，因此，它与实际的底层显示设备息息相关。注意，这里说的显示设备并不是硬件，而是操作系统提供的原生的绘图引擎。所以，在不同的操作系统平台下，QPixmap 的显示可能会有所差别。

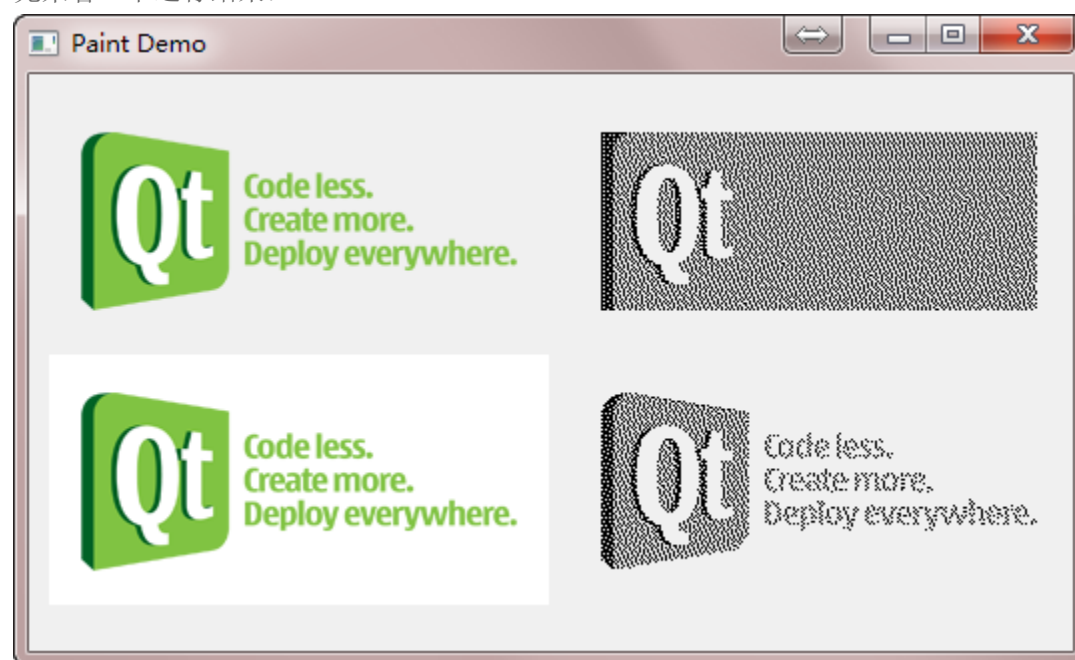
QPixmap 提供了静态的 grabWidget() 和 grabWindow() 函数，用于将自身图像绘制到目标上。同时，在使用 QPixmap 时，你可以直接使用传值的形式，不需要传指针，因为 QPixmap 提供了“隐式数据共享”。关于这一点，我们会在以后的章节中详细描述。简单来说，就是一般对于大型数据（图像无疑就是这种“大型数据”），为性能起见，通常会采用传指针的方式，但是由于 QPixmap 内置了隐式数据共享，所以只要知道传递 QPixmap。

前面说过，QBitmap 继承自 QPixmap，因此具有 QPixmap 的所有特性。不同之处在于，QBitmap 的色深始终为 1。色深这个概念来自计算机图形学，是指用于表现颜色的二进制的位数。我们知道，计算机里面的数据都是使用二进制表示的。为了表示一种颜色，我们也会使用二进制。比如我们要表示 8 种颜色，需要用 3 个二进制位，这时我们就说色深是 3。因此，所谓色深为 1，也就是使用 1 个二进制位表示颜色。1 个位只有两种状态：0 和 1，因此它所表示的颜色就有两种，黑和白。所以说，QBitmap 实际上是只有黑白两色的图像数据。由于 QBitmap 色深小，因此只占用很少的存储空间，所以适合做光标文件和笔刷。

下面我们来看同一个图像文件在 QPixmap 和 QBitmap 下的不同表现：

```
void paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pixmap("qt-logo.png");
    QBitmap bitmap("qt-logo.png");
    painter.drawPixmap(10, 10, 250, 125, pixmap);
    painter.drawPixmap(270, 10, 250, 125, bitmap);
    QPixmap whitePixmap("qt-logo-white.png");
    QBitmap whiteBitmap("qt-logo-white.png");
    painter.drawPixmap(10, 140, 250, 125, whitePixmap);
    painter.drawPixmap(270, 140, 250, 125, whiteBitmap);
}
```

先来看一下运行结果：



这里我们给出了两张 png 图片。qt-logo.png 具有透明背景，qt-logo-white.png 具有白色背景。我们分别使用 QPixmap 和 QBitmap 来加载它们。注意看它们的区别：白色的背景在 QBitmap 中消失了，而透明色在 QBitmap 中转换成了黑色（“黑色”，记住，QBitmap 只有两种颜色：黑色和白色）；其他颜色则是使用点的疏密程度来体现的。

QPixmap 使用底层平台的绘制系统进行绘制，无法提供像素级别的操作，而 QImage 则是使用独立于硬件的绘制系统，实际上是自己绘制自己，因此提供了像素级别的操作，并且能够在不同系统之上提供一个一致的显示形式。



QImage 与 QPixmap 相比，最大的优势在于能够进行像素级别的操作。我们通过上面的示意图可以看到，我们声明一个 3 x 3 像素的 QImage 对象，然后利用 setPixel() 函数进行颜

色的设置。你可以把 `QImage` 想象成一个 RGB 颜色的二维数组，记录了每一像素的颜色。值得注意的是，在 `QImage` 上进行绘制时，不能使用 `QImage::Format_Indexed8` 这种格式。

最后一种 `QPicture` 是平台无关的，因此它可以使用在多种设备之上，比如 `svg`、`pdf`、`ps`、打印机或者屏幕。回忆下我们曾经说的 `QPaintDevice`，实际上是说可以由 `QPainter` 进行绘制的对象。`QPicture` 使用系统分辨率，并且可以调整 `QPainter` 来消除不同设备之间的显示差异。如果我们要记录下 `QPainter` 的命令，首先要使用 `QPainter::begin()` 函数，将 `QPicture` 实例作为参数传递进去，以便告诉系统开始记录，记录完毕后使用 `QPainter::end()` 命令终止。代码示例如下：

```
QPicture picture;
QPainter painter;
painter.begin(&picture);           // 在 picture 进行绘制
painter.drawEllipse(10, 20, 80, 70); // 绘制一个椭圆
painter.end();                     // 绘制完成
picture.save("drawing.pic");       // 保存 picture
如果我们要重现命令，首先要使用 QPicture::load() 函数进行装载：
QPicture picture;
picture.load("drawing.pic");       // 加载 picture
QPainter painter;
painter.begin(&myImage);           // 在 myImage 上开始绘制
painter.drawPicture(0, 0, picture); // 在 (0, 0) 点开始绘制 picture
painter.end();                     // 绘制完成
```

我们也可以直接使用 `QPicture::play()` 进行绘制。这个函数接受一个 `QPainter` 对象，也就是进行绘制的画笔。

## 30 Graphics View Framework

`Graphics View` 提供了一种接口，用于管理大量自定义的 2D 图形元素，并与之进行交互；还提供了用于将这些元素进行可视化显示的观察组件，并支持缩放和旋转。我们通常所说的 Linux 的 KDE 桌面环境，就是建立在 `Graphics View` 基础之上的（尽管新版本的 KDE 有向 `QML` 迁移的趋势）。

`Graphics View` 框架包含了一套完整的事件体系，可以用于与场景中的元素进行双精度的交互。这些元素同样支持键盘事件、鼠标事件等。`Graphics View` 使用了 `BSP` 树（`Binary Space Partitioning tree`，这是一种被广泛应用于图形学方面的数据结构）来提供非常快速的元素发现，也正因为如此，才能够实现一种上百万数量级元素的实时显示机制。

Graphics View 最初在 Qt 4.2 引入，来取代 Qt 3 中的 QCanvas。当然，在最新的 Qt5 中，Qt3 的代码已经不能继续使用了（尽管在一定程度上，Qt4 还是可以使用这些遗留代码）。

Graphics View 是一个基于元素(item)的 MV 架构的框架。它可以分成三个部分：元素 item、场景 scene 和视图 view。

基于元素的意思是，它的每一个组件都是一个独立的元素。这是与我们之前讲到过的 QPainter 状态机机制不同。回忆一下，使用 QPainter 绘图，大多是采用一种面向过程的描述方式：首先使用 drawLine()画一条直线，然后使用 drawPolygon()画一个多边形。对于 Graphics View，相同的过程可以是，首先创建一个场景（scene），然后创建一个直线对象和一个多边形对象，再使用场景的 add()函数，将直线和多边形添加到场景中，最后通过视图进行观察，就可以看到了。乍看起来，后者似乎更加复杂，但是，如果你的图像中包含了成千上万的直线、多边形之类，管理这些对象要比管理 QPainter 的绘制语句容易得多。并且，这些图形对象也更加符合面向对象的设计要求：一个很复杂的图形可以很方便的复用。

MV 架构的意思是，Graphics View 提供一个 model 和一个 view（正如 MVC 架构，只不过 MV 架构少了 C 这么一个组件）。所谓模型（model）就是我们添加的种种对象；所谓视图（view）就是我们观察这些对象的视口。同一个模型可以由很多视图从不同的角度进行观察，这是很常见的需求。使用 QPainter 很难实现这一点，这需要很复杂的计算，而 Graphics View 可以很容易的实现。

Graphics View 提供了 QGraphicsScene 作为场景，即是允许我们添加图形的空间，相当于整个世界；QGraphicsView 作为视口，也就是我们的观察窗口，相当于照相机的取景框，这个取景框可以覆盖整个场景，也可以是场景的一部分；QGraphicsItem 作为图形元件，以便添加到场景中去，Qt 内置了很多图形，比如直线、多边形等，它们都是继承自 QGraphicsItem。

下面我们通过一段代码看看 Graphics View 的使用。

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

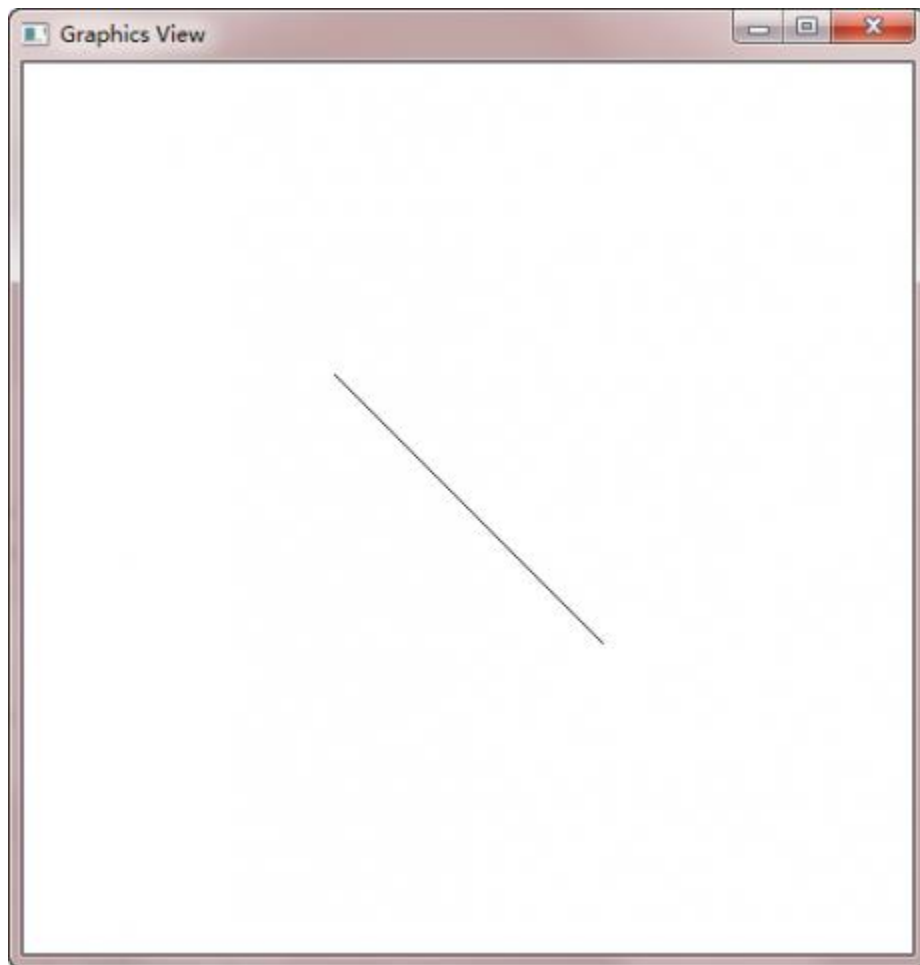
    QGraphicsScene scene;
    scene.addLine(0, 0, 150, 150);

    QGraphicsView view(&scene);
    view.setWindowTitle("Graphics View");
    view.resize(500, 500);
    view.show();

    return app.exec();
}
```

```
}
```

这段代码很简单：首先创建一个场景，也就是 `QGraphicsScene` 对象。然后我们使用 `addLine()` 函数向场景中添加了一个直线，起始点和终点坐标分别是 `(0, 0)` 和 `(150, 150)`。可以想象，这是一个边长 `150px` 的正方形的对角线。通过这两步，我们已经有了场景和元素。之后，我们创建一个 `GraphicsView` 对象，绑定到一个场景上（也就是我们前面创建的 `scene` 对象）。注意，`QGraphicsScene` 不是 `QWidget` 的子类，因此该构造函数并不是调用的 `QGraphicsView(QWidget *parent)`。接下来，我们可以运行一下代码：



我们看到，这个直线自动在视图居中显示。这并不需要我们进行任何额外的代码。如果不想这么做，我们可以给 `scene` 设置一下 `sceneRect()` 属性：

```
QGraphicsScene scene;  
scene.setSceneRect(0, 0, 300, 300);  
scene.addLine(0, 0, 150, 150);  
  
QGraphicsView view(&scene);  
view.setWindowTitle("Graphics View");
```

```
// view.resize(500, 500);  
view.show();
```

不仅如此，我们还去掉了 `view.resize()` 一行。`QGraphicsScene` 的 `sceneRect` 属性供 `QGraphicsView` 确定视图默认的滚动条区域，并且协助 `QGraphicsScene` 管理元素索引。之所以去掉 `view.resize()` 一行，是因为我们让系统去决定视图的最小尺寸（否则的话，我们需要手动将窗口标题栏等的大小同时考虑设置）。

## 31 贪吃蛇游戏（1）

经过前面一段时间的学习，我们已经了解到有关 Qt 相当多的知识。现在，我们将把前面所讲过的知识综合起来，开发一个贪吃蛇游戏。游戏很简单，相信大家都有见过，多多少少也都玩过。我们在实现这个贪吃蛇游戏时，会利用到事件系统、Graphics View Framework、QPainter 等相关内容，也会了解到一个游戏所具有的一些特性，比如游戏循环等，在 Qt 中如何体现出来。当然，最重要的是，通过一个相对较大的程序，学习到如何将之前的点点滴滴结合在一起。

本部分的代码出自：<http://qtcollege.co.il/developing-a-qt-snake-game/>，但是有一些基于软件工程方面考虑的修改，例如常量放置的位置等。

前面说过，Qt 提供了自己的绘制系统，还提供了 Graphics View Framework。很明显，绘制图形和移动图形，是一个游戏的核心。对于游戏而言，将其中的每一个部分看做对象是非常合理的，也是相当有成效的。因此，我们选择 Graphics View Framework 作为核心框架。回忆一下，这个框架具有一系列面向对象的特性，能够让我们将一个个图形作为对象进行处理。同时，Graphics View Framework 的性能很好，即便是数千上万的图形也没有压力。这一点非常适合于游戏。

正如我们前面所说，Graphics View Framework 有三个主要部分：

- `QGraphicsScene`：能够管理元素的非 GUI 容器；
- `QGraphicsItem`：能够被添加到场景的元素；
- `QGraphicsView`：能够观察场景的可视化组件视图。

对于游戏而言，我们需要一个 `QGraphicsScene`，作为游戏发生的舞台；一个 `QGraphicsView`，作为观察游戏舞台的组件；以及若干元素，用于表示游戏对象，比如蛇、食物以及障碍物等。

大致分析过游戏组成以及各部分的实现方式后，我们可以开始编码了。这当然是一个 GUI 工程，主窗口应该是一个 `QGraphicsView`。为了以后的实现方便（比如，我们希望向工具栏添加按钮等），我们不会直接以 `QGraphicsView` 作为顶层窗口，而是将其添加到一个主窗口上。这里，我们不会使用 `QtDesigner` 进行界面设计，而是直接编码完成（**注意，我们这里的代码并不一定能够通过编译，因为会牵扯到其后几章的内容，因此，如果需要编译代码，请在全部代码讲解完毕之后进行**）：

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QGraphicsScene;
class QGraphicsView;

class GameController;

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void adjustViewSize();

private:
    void initScene();
    void initSceneBackground();

    QGraphicsScene *scene;
    QGraphicsView *view;

    GameController *game;
};

#endif // MAINWINDOW_H
```

在头文件中声明了 `MainWindow`。构造函数除了初始化成员变量，还设置了窗口的大小，并且需要对场景进行初始化：

```
MainWindow::MainWindow(QWidget *parent) :
```

```

MainWindow(parent),
scene(new QGraphicsScene(this)),
view(new QGraphicsView(scene, this)),
game(new GameController(*scene, this))
{
    setCentralWidget(view);
    resize(600, 600);

    initScene();
    initSceneBackground();

    QTimer::singleShot(0, this, SLOT(adjustViewSize()));
}

```

值得说明的是最后一行代码。singleShot()函数原型如下：

```
static void QTimer::singleShot(int msec, QObject * receiver, const char * member);
```

该函数接受三个参数，简单来说，它的作用是，在 msec 毫秒之后，调用 receiver 的 member 槽函数。在我们的代码中，第一个参数传递的是 0，也就是 0ms 之后，调用 this->adjustViewSize()。这与直接调用 this->adjustViewSize();有什么区别呢？如果你看文档，这一段的解释很隐晦。文档中写到：“It is very convenient to use this function because you do not need to bother with a timerEvent or create a local QTimer object”，也就是说，它的作用是方便使用，无需重写 timerEvent()函数或者是创建一个局部的 QTimer 对象。当我们使用 QTimer::singleShot(0, ...)的时候，实际上也是对 QTimer 的简化，而不是简单地函数调用。QTimer 的处理是将其放到事件列表中，等到下一次事件循环开始时去调用这个函数。那么，QTimer::singleShot(0, ...)意思是，在下一次事件循环开始时，立刻调用指定的槽函数。在我们的例子中，我们需要在视图绘制完毕后才去改变大小（视图绘制当然是在 paintEvent()事件中），因此我们需要在下一次事件循环中调用 adjustViewSize()函数。这就是为什么我们需要用 QTimer 而不是直接调用 adjustViewSize()。如果熟悉 flash，这相当于 flash 里面的 callLater()函数。接下来看看 initScene()和 initSceneBackground()的代码：

```

void MainWindow::initScene()
{
    scene->setSceneRect(-100, -100, 200, 200);
}

void MainWindow::initSceneBackground()
{
    QPixmap bg(TILE_SIZE, TILE_SIZE);
    QPainter p(&bg);
    p.setBrush(QBrush(Qt::gray));
}

```



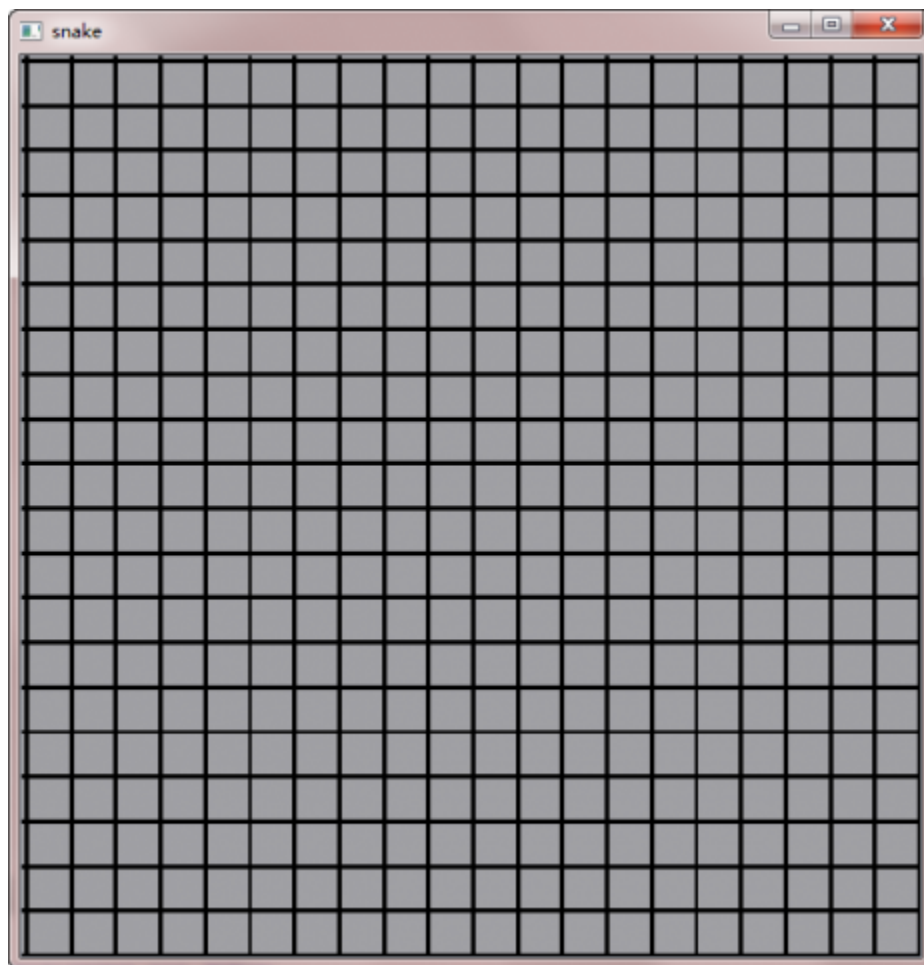
```
p.drawRect(0, 0, TILE_SIZE, TILE_SIZE);

view->setBackgroundBrush(QBrush(bg));
}
```

initScene()函数设置场景的范围，是左上角在 (-100,-100)，长和宽都是 200px 的矩形。默认情况下，场景是无限大的，我们代码的作用是设置了一个有限的范围。Graphics View Framework 为每一个元素维护三个不同的坐标系：场景坐标，元素自己的坐标以及其相对于父组件的坐标。除了元素在场景中的位置，其它几乎所有位置都是相对于元素坐标系的。所以，我们选择的矩形 (-100, -100, 200, 200)，实际是设置了场景的坐标系。此时，如果一个元素坐标是 (-100, -100)，那么它将出现在场景左上角，(100, 100) 的坐标则是在右下角。

initSceneBackground()函数看似很长，实际却很简单。首先我们创建一个边长 TILE\_SIZE 的 QPixmap，将其使用灰色填充矩形。我们没有设置边框颜色，默认就是黑色。然后将这个 QPixmap 作为背景画刷，铺满整个视图。

现在我们的程序看起来是这样的：



在后面的章节中，我们将继续我们的游戏之旅。下一章，我们开始创建游戏对象。

## 32 贪吃蛇游戏（2）

下面我们继续上一章的内容。在上一章中，我们已经完成了地图的设计，当然是相当简单的。在我们的游戏中，另外的主角便是蛇和食物。下面我们便开始这部分的发展。

我们的地图是建立在 `QGraphicsScene` 的基础之上的，所以，里面的对象应该是 `QGraphicsItem` 实例。通常，我们会把所有的图形元素（这里便是游戏中需要的对象，例如蛇、食物等）设计为 `QGraphicsItem` 的子类，在这个类中添加绘制自身的代码以及动画逻辑。这也是面向对象的开发方式：封装自己的属性和操作。在我们的游戏中，应该有三个对象：蛇 `Snake`、食物 `Food` 以及墙 `Wall`。

我们从食物开始。因为它是最简单的。我们将其作为一个红色的小圆饼，大小要比地图中的一个方格要小，因此我们可以将其放置在一个方格中。正如上面分析的那样，我们的 `Food` 类需要继承 `QGraphicsItem`。按照接口约束，`QGraphicsItem` 的子类需要重写至少两个函数：`boundingRect()`和 `paint()`。

`boundingRect()`返回一个用于包裹住图形元素的矩形，也就是这个图形元素的范围。需要注意的是，这个矩形必须能够**完全包含**图形元素。所谓“完全包含”，意思是，在图形元素有动画的时候，这个矩形也必须将整个图形元素包含进去。如果范围矩形过小。图形会被剪切；如果范围矩形过大，就会影响性能。

`paint()`的作用是使用 `QPainter` 将图形元素绘制出来。

下面是 `food.h` 和 `food.cpp` 的内容：

```
////////// food.h //////////
#ifndef FOOD_H
#define FOOD_H

#include <QGraphicsItem>

class Food : public QGraphicsItem
{
public:
    Food(qreal x, qreal y);

    QRectF boundingRect() const;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *);

    QPainterPath shape() const;
```

```

};

#endif // FOOD_H

////////// food.cpp //////////
#include <QPainter>

#include "constants.h"
#include "food.h"

static const qreal FOOD_RADIUS = 3;

Food::Food(qreal x, qreal y)
{
    setPos(x, y);
    setData(GD_Type, GO_Food);
}

QRectF Food::boundingRect() const
{
    return QRectF(-TILE_SIZE, -TILE_SIZE,
                  TILE_SIZE * 2, TILE_SIZE * 2 );
}

void Food::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
    painter->save();

    painter->setRenderHint(QPainter::Antialiasing);
    painter->fillPath(shape(), Qt::red);

    painter->restore();
}

QPainterPath Food::shape() const
{
    QPainterPath p;
    p.addEllipse(QPointF(TILE_SIZE / 2, TILE_SIZE / 2), FOOD_RADIUS,
FOOD_RADIUS);
    return p;
}

```

虽然这段代码很简单，我们还是有必要解释一下。构造函数接受两个参数： $x$  和  $y$ ，用于指定该元素的坐标。`setData()`函数是我们之后要用到的，这里简单提一句，它的作用为该图形元素添加额外的数据信息，类似于散列一样的键值对的形式。`boundingRect()`简单地返回一个 `QRect` 对象。由于我们的元素就是一个圆形，所以我们返回的是一个简单的矩形。注意，这个矩形的范围实际是四倍于实际区域的：以元素坐标  $(x, y)$  为中心，边长为 `TILE_SIZE * 2` 的正方形。我们还重写了 `shape()`函数。这也是一个虚函数，但是并不是必须覆盖的。这个函数返回的是元素实际的路径。所谓路径，可以理解成元素的矢量轮廓线，就是 `QPainterPath` 所表示的。我们使用 `addEllipse()`函数，添加了一个圆心为  $(\text{TILE\_SIZE} / 2, \text{TILE\_SIZE} / 2)$ ，半径 `FOOD\_RADIUS` 的圆，其范围是左上角为  $(x, y)$  的矩形。由于设置了 `shape()`函数，`paint()`反而更简单。我们所要做的，就是把 `shape()`函数定义的路径绘制出来。注意，我们使用了 `QPainter::save()`和 `QPainter::restore()`两个函数，用于保存画笔状态。

现在我们有第一个图形元素，那么，就让我们把它添加到场景中吧！对于一个游戏，通常需要一个中心控制的类，用于控制所有游戏相关的行为。我们将其取名为 `GameController`。

`GameController` 的工作是，初始化场景中的游戏对象，开始游戏循环。每一个游戏都需要有一个游戏循环，类型于事件循环。想象一个每秒滴答 30 次的表。每次响起滴答声，游戏对象才有机会执行相应的动作：移动、检查碰撞、攻击或者其它一些游戏相关的活动。为了方便起见，我们将这一次滴答成为一帧，那么，每秒 30 次滴答，就是每秒 30 帧。游戏循环通常使用定时器实现，因为应用程序不仅仅是一个游戏循环，还需要响应其它事件，比如游戏者的鼠标键盘操作。正因为如此，我们不能简单地使用无限的 `for` 循环作为游戏循环。

在 `Graphics View Framework` 中，每一帧都应该调用一个称为 `advance()`的函数。

`QGraphicsScene::advance()`会调用场景中每一个元素自己的 `advance()`函数。所以，如果图形元素需要做什么事，必须重写 `QGraphicsItem` 的 `advance()`，然后在游戏循环中调用这个函数。

`GameController` 创建并开始游戏循环。当然，我们也可以加入 `pause()`和 `resume()`函数。现在，我们来看看它的实现：

```
GameController::GameController(QGraphicsScene *scene, QObject *parent) :
    QObject(parent),
    scene(scene),
    snake(new Snake(this))
{
    timer.start(1000/33);

    Food *a1 = new Food(0, -50);
    scene->addItem(a1);

    scene->addItem(snake);
```

```

        scene->installEventFilter(this);

        resume();
    }

```

GameController 的构造函数。首先开启充当游戏循环的定时器，定时间隔是 1000 / 33 毫秒，也就是每秒 30 (1000 / 33 = 30) 帧。GameController 有两个成员变量：scene 和 snake，我们将第一个食物和蛇都加入到场景中。同时，我们为 GameController 添加了事件过滤器，以便监听键盘事件。这里我们先不管这个事件过滤器，直接看看后面的代码：

```

void GameController::pause()
{
    disconnect(&timer, SIGNAL(timeout()),
               scene, SLOT(advance()));
}

void GameController::resume()
{
    connect(&timer, SIGNAL(timeout()),
            scene, SLOT(advance()));
}

```

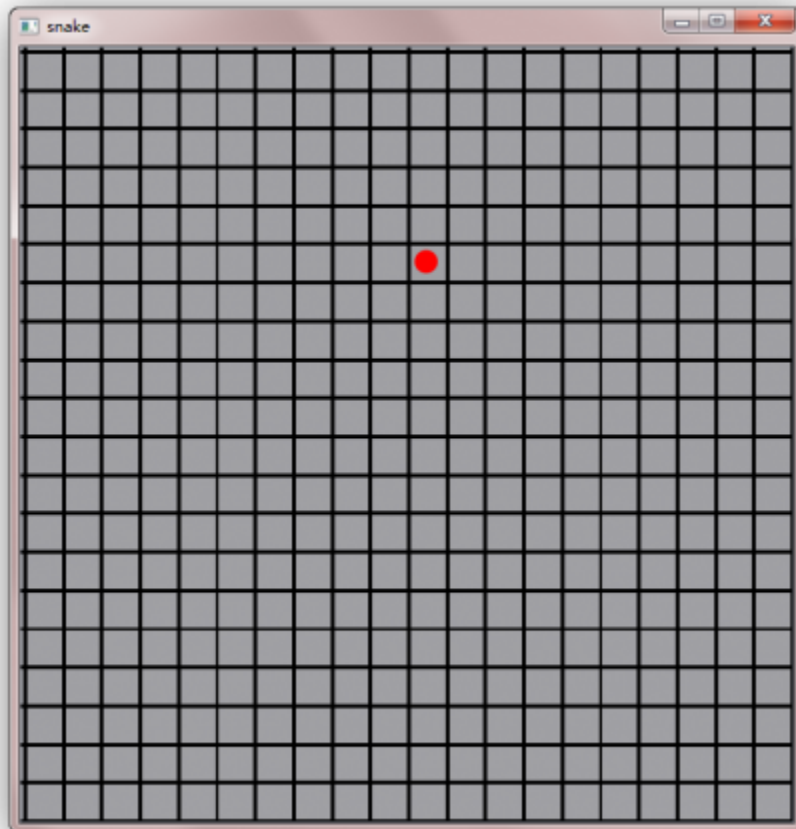
pause() 和 resume() 函数很简答：我们只是连接或者断开定时器的信号。当我们把这一切都准备好之后，我们把 GameController 添加到 MainWindow 中：

```

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent),
      game(new GameController(scene, this))
{
    ...
}

```

由于 GameController 在构造时已经开始游戏循环，因此我们不需要另外调用一个所谓的“start”函数。这样，我们就把第一个食物添加到了游戏场景：



接下来是有关蛇的处理。

蛇要更复杂一些。在我们的游戏中，蛇是由黄色的小方块组成，这是最简单的实现方式了。第一个是蛇的头部，紧接着是它的身体。对此，我们有两个必须面对的困难：

1. 蛇具有复杂得多的形状。因为蛇的形状随着游戏者的控制而不同，因此，我们必须找出一个能够恰好包含蛇头和所有身体块的矩形。这也是 `boundingRect()` 函数所要解决的问题。
2. 蛇会长大（比如吃了食物之后）。因此，我们需要在蛇对象中增加一个用于代表蛇身体长度的 `growing` 变量：当 `growing` 为正数时，蛇的身体增加一格；当 `growing` 为负数时，蛇的身体减少一格。
3. `advance()` 函数用于编码移动部分，这个函数会在一秒内调用 30 次（这是我们在 `GameController` 的定时器中决定的）。

我们首先从 `boundingRect()` 开始看起：

```

QRectF Snake::boundingRect() const
{
    qreal minX = head.x();
    qreal minY = head.y();
    qreal maxX = head.x();
    qreal maxY = head.y();

    foreach (QPointF p, tail) {
        maxX = p.x() > maxX ? p.x() : maxX;
        maxY = p.y() > maxY ? p.y() : maxY;
        minX = p.x() < minX ? p.x() : minX;
        minY = p.y() < minY ? p.y() : minY;
    }

    QPointF tl = mapFromScene(QPointF(minX, minY));
    QPointF br = mapFromScene(QPointF(maxX, maxY));

    QRectF bound = QRectF(tl.x(), // x
                           tl.y(), // y
                           br.x() - tl.x() + SNAKE_SIZE, // width
                           br.y() - tl.y() + SNAKE_SIZE //height
                           );

    return bound;
}

```

这个函数的算法是：遍历蛇身体的每一个方块，找出所有部分的最大的 x 坐标和 y 坐标，以及最小的 x 坐标和 y 坐标。这样，夹在其中的便是蛇身体的外围区域。

shape()函数决定了蛇身体的形状，我们遍历蛇身体的每一个方块向路径中添加：

```

QPainterPath Snake::shape() const
{
    QPainterPath path;
    path.setFillRule(Qt::WindingFill);

    path.addRect(QRectF(0, 0, SNAKE_SIZE, SNAKE_SIZE));

    foreach (QPointF p, tail) {
        QPointF itemp = mapFromScene(p);
        path.addRect(QRectF(itemp.x(), itemp.y(), SNAKE_SIZE, SNAKE_SIZE));
    }

    return path;
}

```

在我们实现了 `shape()` 函数的基础之上，`paint()` 函数就很简单了：

```
void Snake::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
    painter->save();
    painter->fillPath(shape(), Qt::yellow);
    painter->restore();
}
```

现在我们已经把蛇“画”出来。下一章中，我们将让它“动”起来，从而完成我们的贪吃蛇游戏。

## 33 贪吃蛇游戏（3）

继续前面一章的内容。上次我们讲完了有关蛇的静态部分，也就是绘制部分。现在，我们开始添加游戏控制的代码。首先我们从最简单的四个方向键开始：

```
void Snake::moveLeft()
{
    head.rx() -= SNAKE_SIZE;
    if (head.rx() < -100) {
        head.rx() = 100;
    }
}

void Snake::moveRight()
{
    head.rx() += SNAKE_SIZE;
    if (head.rx() > 100) {
        head.rx() = -100;
    }
}

void Snake::moveUp()
{
    head.ry() -= SNAKE_SIZE;
    if (head.ry() < -100) {
        head.ry() = 100;
    }
}
```



```

void Snake::moveDown()
{
    head.ry() += SNAKE_SIZE;
    if (head.ry() > 100) {
        head.ry() = -100;
    }
}

```

我们有四个以 `move` 开头的函数，内容都很类似：分别以 `SNAKE_SIZE` 为基准改变头部坐标，然后与场景边界比较，大于边界值时，设置为边界值。这么做的结果是，当蛇运动到场景最右侧时，会从最左侧出来；当运行到场景最上侧时，会从最下侧出来。

然后我们添加一个比较复杂的函数，借此，我们可以看出 `Graphics View Framework` 的强大之处：

```

void Snake::handleCollisions()
{
    QList collisions = collidingItems();

    // Check collisions with other objects on screen
    foreach (QGraphicsItem *collidingItem, collisions) {
        if (collidingItem->data(GD_Type) == GO_Food) {
            // Let GameController handle the event by putting another apple
            controller.snakeAteFood(this, (Food *)collidingItem);
            growing += 1;
        }
    }

    // Check snake eating itself
    if (tail.contains(head)) {
        controller.snakeAteItself(this);
    }
}

```

顾名思义，`handleCollisions()` 的意思是处理碰撞，也就是所谓的“碰撞检测”。首先，我们使用 `collidingItems()` 取得所有碰撞的元素。这个函数的签名是：

```

QList<QGraphicsItem *> QGraphicsItem::collidingItems(
    Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const

```

该函数返回与这个元素碰撞的所有元素。`Graphics View Framework` 提供了四种碰撞检测的方式：

- **Qt::ContainsItemShape**: 如果被检测物的形状 (**shape()**) 完全包含在检测物内, 算做碰撞;
- **Qt::IntersectsItemShape**: 如果被检测物的形状 (**shape()**) 与检测物有交集, 算做碰撞;
- **Qt::ContainsItemBoundingRect**: 如果被检测物的包含矩形 (**boundingRect()**) 完全包含在检测物内, 算做碰撞;
- **Qt::IntersectsItemBoundingRect**: 如果被检测物的包含矩形 (**boundingRect()**) 与检测物有交集, 算做碰撞。

注意, 该函数默认是 **Qt::IntersectsItemShape**。回忆一下, 我们之前编写的代码, **Food** 的 **boundingRect()** 要大于其实际值, 却不影响我们的游戏逻辑判断, 这就是原因: 因为我们使用的是 **Qt::IntersectsItemShape** 判断检测, 这与 **boundingRect()** 无关。

后面的代码就很简单了。我们遍历所有被碰撞的元素, 如果是食物, 则进行吃食物的算法, 同时将蛇的长度加 1。最后, 如果身体包含了头, 那就是蛇吃了自己的身体。

还记得我们在 **Food** 类中有这么一句:

```
setData(GD_Type, GO_Food);
```

**QGraphicsItem::setData()** 以键值对的形式设置元素的自定义数据。所谓自定义数据, 就是对应用程序有所帮助的用户数据。**Qt** 不会使用这种机制来存储数据, 因此你可以放心地将所需要的数据存储到元素对象。例如, 我们在 **Food** 的构造函数中, 将 **GD\_Type** 的值设置为 **GO\_Food**。那么, 这里我们取出 **GD\_Type**, 如果其值是 **GO\_Food**, 意味着这个 **QGraphicsItem** 就是一个 **Food**, 因此我们可以将其安全地进行后面的类型转换, 从而完成下面的代码。

下面是 **advance()** 函数的代码:

```
void Snake::advance(int step)
{
    if (!step) {
        return;
    }
    if (tickCounter++ % speed != 0) {
        return;
    }
    if (moveDirection == NoMove) {
        return;
    }
}
```

```

    }

    if (growing > 0) {
        QPointF tailPoint = head;
        tail << tailPoint;
        growing -= 1;
    } else {
        tail.takeFirst();
        tail << head;
    }

    switch (moveDirection) {
        case MoveLeft:
            moveLeft();
            break;
        case MoveRight:
            moveRight();
            break;
        case MoveUp:
            moveUp();
            break;
        case MoveDown:
            moveDown();
            break;
    }

    setPos(head);
    handleCollisions();
}

```

QGraphicsItem::advance()函数接受一个 int 作为参数。这个 int 代表该函数被调用的时间。QGraphicsItem::advance()函数会被 QGraphicsScene::advance()函数调用两次：第一次时这个 int 为 0，代表即将开始调用；第二次这个 int 为 1，代表已经开始调用。在我们的代码中，我们只使用不为 0 的阶段，因此当 !step 时，函数直接返回。

tickCounter 实际是我们内部的一个计时器。我们使用 speed 作为蛇的两次动作的间隔时间，直接影响到游戏的难度。speed 值越大，两次运动的间隔时间越大，游戏越简单。这是因为随着 speed 的增大，tickCounter % speed != 0 的次数响应越多，刷新的次数就会越少，蛇运动得越慢。

moveDirection 显然就是运动方向，当是 NoMove 时，函数直接返回。

growing 是正在增长的方格数。当其大于 0 时，我们将头部追加到尾部的位罝，同时减少一个方格；当其小于 0 时，我们删除第一个，然后把头部添加进去。我们可以把 growing 看做即将发生的变化。比如，我们将 growing 初始化为 7。第一次运行 advance()时，由于 7 > 1，因此将头部追加，然后 growing 减少 1。直到 growing 为 0，此时，蛇的长度不再发生变化，直到我们吃了一个食物。

下面是相应的方向时需要调用对应的函数。最后，我们设置元素的坐标，同时检测碰撞。

## 34 贪吃蛇游戏（4）

这将是我们这个稍大一些的示例程序的最后一部分。在本章中，我们将完成 GameController 中有关用户控制的相关代码。

首先，我们来给 GameController 添加一个事件过滤器：

```
bool GameController::eventFilter(QObject *object, QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        handleKeyPressed((QKeyEvent *)event);
        return true;
    } else {
        return QObject::eventFilter(object, event);
    }
}
```

回忆一下，我们使用 QGraphicsScene 作为游戏场景。为什么不直接继承 QGraphicsScene，重写其 keyPressEvent()函数呢？这里的考虑是：第一，我们不想只为重写一个键盘事件而继承 QGraphicsScene。这不符合面向对象设计的要求。继承首先应该有“是一个（is-a）”的关系。我们将游戏场景继承 QGraphicsScene 当然满足这个关系，无可厚非。但是，继承还有一个“特化”的含义，我们只想控制键盘事件，并没有添加其它额外的代码，因此感觉并不应该作此继承。第二，我们希望将表示层与控制层分离：明明已经有了 GameController，显然，这是一个用于控制游戏的类，那么，为什么键盘控制还要放在场景中呢？这岂不将控制与表现层耦合起来了吗？基于以上两点考虑，我们选择不继承 QGraphicsScene，而是在 GameController 中为场景添加事件过滤器，从而完成键盘事件的处理。下面我们看看这个 handleKeyPressed()函数是怎样的：

```
void GameController::handleKeyPressed(QKeyEvent *event)
{
    switch (event->key()) {
```

```

        case Qt::Key_Left:
            snake->setMoveDirection(Snake::MoveLeft);
            break;
        case Qt::Key_Right:
            snake->setMoveDirection(Snake::MoveRight);
            break;
        case Qt::Key_Up:
            snake->setMoveDirection(Snake::MoveUp);
            break;
        case Qt::Key_Down:
            snake->setMoveDirection(Snake::MoveDown);
            break;
    }
}

```

这段代码并不复杂：只是设置蛇的运动方向。记得我们在前面的代码中，已经为蛇添加了运动方向的控制，因此，我们只需要修改这个状态，即可完成对蛇的控制。由于前面我们已经在蛇的对象中完成了相应控制的代码，因此这里的游戏控制就是这么简单。接下来，我们要完成游戏逻辑：吃食物、生成新的食物以及咬到自己这三个逻辑：

```

void GameController::snakeAteFood(Snake *snake, Food *food)
{
    scene.removeItem(food);
    delete food;

    addNewFood();
}

```

首先是蛇吃到食物。如果蛇吃到了食物，那么，我们将食物从场景中移除，然后添加新的食物。为了避免内存泄露，我们需要在这里 `delete` 食物，以释放占用的空间。当然，你应该想到，我们肯定会在 `addNewFood()`函数中使用 `new` 运算符重新生成新的食物。

```

void GameController::addNewFood()
{
    int x, y;

    do {
        x = (int) (qrand() % 100) / 10;
        y = (int) (qrand() % 100) / 10;

        x *= 10;
        y *= 10;
    } while (x == snake->getX() && y == snake->getY());
}

```

```

    } while (snake->shape().contains(snake->mapFromScene(QPointF(x + 5, y +
5))));

    Food *food = new Food(x , y);
    scene.addItem(food);
}

```

在 `addNewFood()` 代码中，我们首先计算新的食物的坐标：使用一个循环，直到找到一个不在蛇身体中的坐标。为了判断一个坐标是不是位于蛇的身体上，我们利用蛇的 `shape()` 函数。需要注意的是，`shape()` 返回元素坐标系中的坐标，而我们计算而得的 `x`, `y` 坐标位于场景坐标系，因此我们必须利用 `QGraphicsItem::mapFromScene()` 将场景坐标系映射为元素坐标系。当我们计算出食物坐标后，我们在堆上重新创建这个食物，并将其添加到游戏场景。

```

void GameController::snakeAteItself(Snake *snake)
{
    QTimer::singleShot(0, this, SLOT(gameOver()));
}

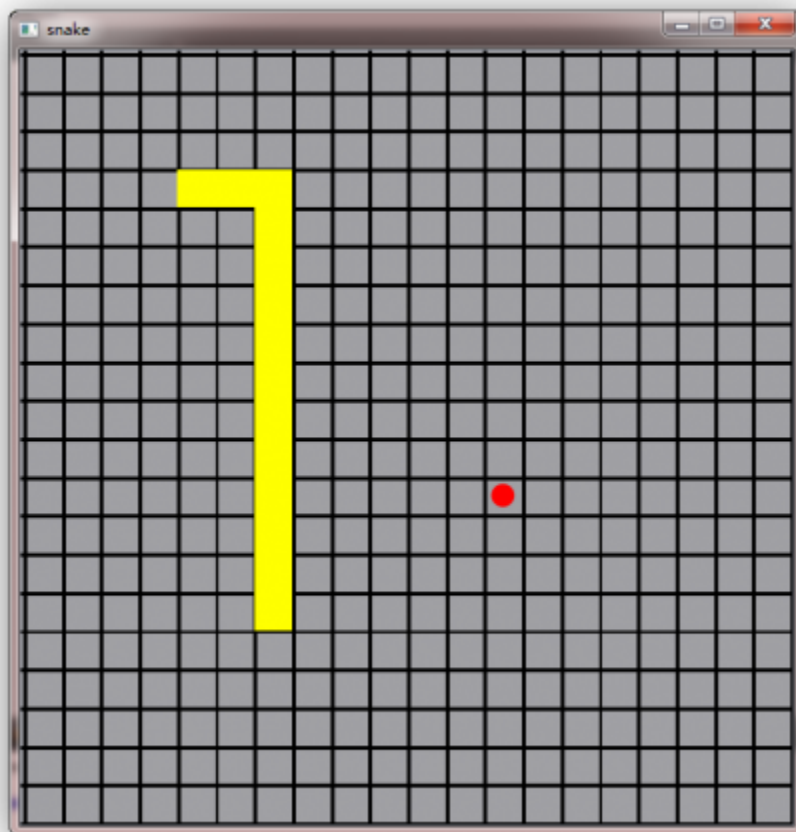
void GameController::gameOver()
{
    scene.clear();

    snake = new Snake(*this);
    scene.addItem(snake);
    addNewFood();
}

```

如果蛇咬到了它自己，游戏即宣告结束。因此，我们直接调用 `gameOver()` 函数。这个函数将场景清空，然后重新创建蛇并增加第一个食物。为什么我们不直接调用 `gameOver()` 函数，而是利用 `QTimer` 调用呢（希望你没有忘记 `QTimer::singleShot(0, ...)` 的用法）？这是因为，我们不应该在一个 `update` 操作中去清空整个场景。因此我们使用 `QTimer`，在 `update` 事件之后完成这个操作。

至此，我们已经把这个简单的贪吃蛇游戏全部完成。最后我们来看一下运行结果：



文末的附件中是我们当前的全部代码。如果你检查下这部分代码，会发现我们其实还没有完成整个游戏：**Wall** 对象完全没有实现，难度控制也没有完成。当然，通过我们的讲解，希望你已经理解了我们设计的原则以及各部分代码之间的关系。如果感兴趣，可以继续完成这部分代码。豆子在 [github](#) 上面创建了一个代码库，如果你感觉自己的改进比较成功，或者希望与大家分享，欢迎 clone 仓库提交代码！

附件：[snake](#)

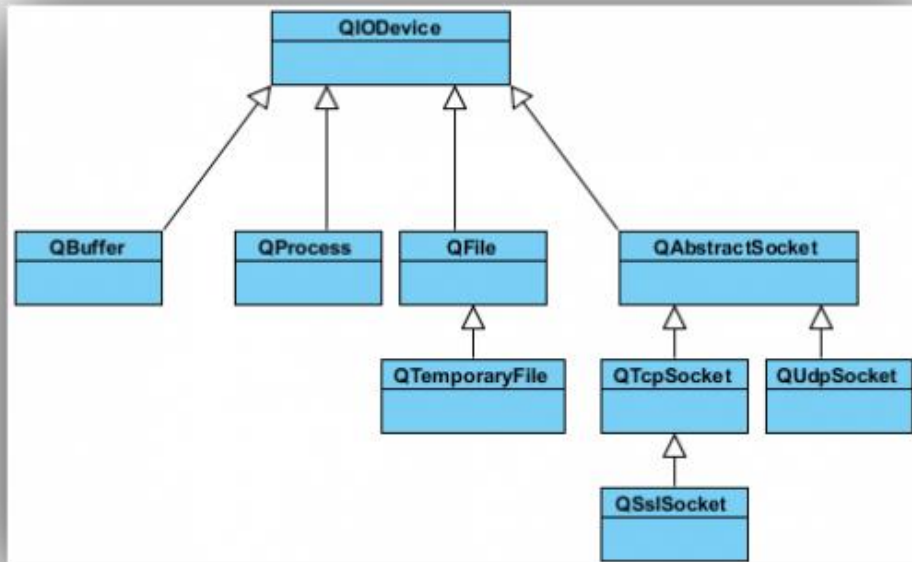
git: `git@github.com:devbean/snake-game.git`

## 35 文件

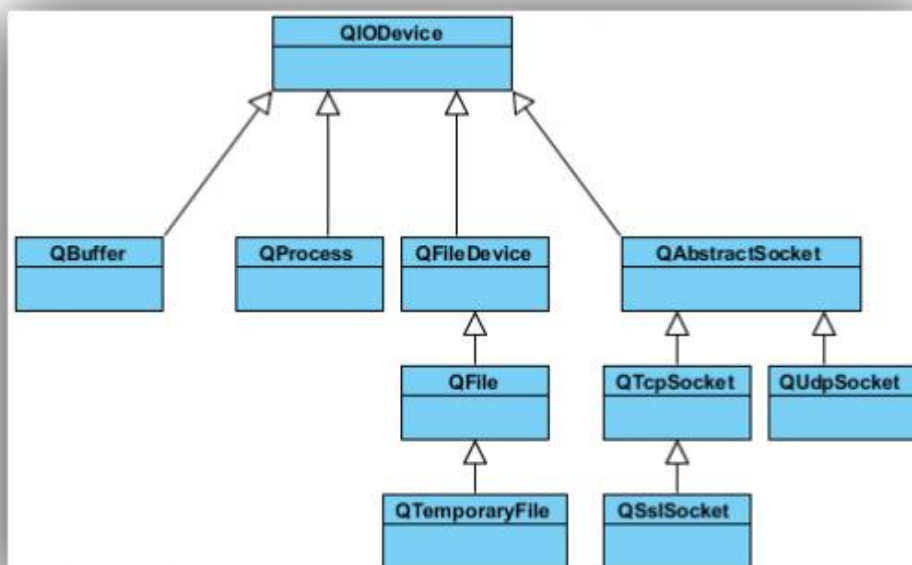
文件操作是应用程序必不可少的部分。Qt 作为一个通用开发库，提供了跨平台的文件操作能力。从本章开始，我们来了解下 Qt 的文件以及输入输出的功能，也就是 I/O 系统。

Qt 通过 QIODevice 提供了对 I/O 设备的抽象，这些设备具有读写字节块的能力。下面是 I/O 设备的类图：

Qt4



Qt5





从上面的类图可以看出，Qt4 和 Qt5 在 I/O 设备部分大同小异。只不过 Qt5 新增加了一个 `QIODevice` 类。途中所涉及的类及其用途简要说明如下：

- `QIODevice`：所有 I/O 设备类的父类，提供了字节块读写的通用操作以及基本接口；
- `QFile`：访问本地文件或者嵌入资源；
- `QTemporaryFile`：创建和访问本地文件系统的临时文件；
- `QBuffer`：读写 `QByteArray`；
- `QProcess`：运行外部程序，处理进程间通讯；
- `QAbstractSocket`：所有套接字类的父类；
- `QTcpSocket`：TCP 协议网络数据传输；
- `QUdpSocket`：传输 UDP 报文；
- `QSslSocket`：使用 SSL/TLS 传输数据；
- `QIODevice`：Qt5 新增加的类，提供了有关文件操作的通用实现。

这其中，`QProcess`、`QTcpSocket`、`QUdpSocket` 和 `QSslSocket` 是顺序访问设备。所谓“顺序访问”，是指它们的数据只能访问一遍：从头走到尾，从第一个字节开始访问，直到最后一个字节，中途不能返回去读取上一个字节；`QFile`、`QTemporaryFile` 和 `QBuffer` 是随机访问设备，可以访问任意位置任意次数，还可以使用 `QIODevice::seek()` 函数来重新定位文件访问位置指针。

本章将主要介绍 `QFile` 及其相关类，后面的章节则开始介绍有关输入输出的流。

在所有的 I/O 设备中，文件 I/O 是最重要的部分之一。因为我们大多数的程序依旧需要首先访问本地文件（当然，在云计算大行其道的将来，这一观点可能改变）。`QFile` 提供了从文件中读取和写入数据的能力。Qt5 新加入的 `QIODevice` 类，则将这部分公共操作放到了这个单独的类中。显然，这部分代码在 Qt4 中位于 `QFile` 类。这样看来，Qt5 的代码结构更为清晰，层次更好。

我们通常会将文件路径作为参数传给 `QFile` 的构造函数。不过也可以在创建好对象最后，使用 `setFileName()` 来修改。`QFile` 需要使用 `/` 作为文件分隔符，不过，它会自动将其转换成操作系统所需要的形式。例如 `C:/windows` 这样的路径在 Windows 平台下同样是可以的。

**QFile** 主要提供了有关文件的各种操作，比如打开文件、关闭文件、刷新文件等。我们可以使用 **QDataStream** 或 **QTextStream** 类来读写文件，也可以使用 **QIODevice** 类提供的 **read()**、**readLine()**、**readAll()**以及 **write()**这样的函数。值得注意的是，有关文件本身的信息，比如文件名、文件所在目录的名字等，则是通过 **QFileInfo** 获取，而不是自己分析文件路径字符串。

下面我们使用一段代码来看看 **QFile** 的有关操作：

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

    QFile file("in.txt");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qDebug() << "Open file failed.";
        return -1;
    }else {
        while (!file.atEnd()) {
            qDebug() << file.readLine();
        }
    }

    QFileInfo info(file);
    qDebug() << info.isDir();
    qDebug() << info.isExecutable();
    qDebug() << info.baseName();
    qDebug() << info.completeBaseName();
    qDebug() << info.suffix();
    qDebug() << info.completeSuffix();

    return app.exec();
}
```

在这段代码中，我们首先使用 **QFile** 创建了一个文件对象。这个文件名字是 **in.txt**。如果你不知道应该把它放在哪里，可以使用 ~~**app.applicationFilePath();**~~或者 **app.applicationDirPath();**来获得应用程序的执行路径。只要将这个文件放在同执行路径一致的目录下即可。可以使用 **QDir::currentPath()**来获得应用程序执行时的当前路径。只要将这个文件放在与当前路径一致的目录下即可。然后，我们使用 **open()**函数打开这个文件，打开形式是只读方式，文本格式。这个类似于 **fopen()**的 **r** 这样的参数。**open()**函数返回一个 **bool** 类型，如果打开失败，我们在控制台输出一段提示然后程序退出。否则，我们利用 **while** 循环，将每一行读到的内容输出。

程序的第二部分，我们使用 `QFileInfo` 获取有关该文件的信息。`QFileInfo` 有很多类型的函数，我们只举出一些例子。比如这里，`isDir()`检查该文件是否是目录；`isExecutable()`检查该文件是否是可执行文件等。`baseName()`可以直接获得文件名；`suffix()`则直接获取文件后缀名。我们可以由下面的示例看到，`baseName()`和 `completeBaseName()`，以及 `suffix()`和 `completeSuffix()`的区别：

```
QFileInfo fi("/tmp/archive.tar.gz");
QString base = fi.baseName(); // base = "archive"
QString cbase = fi.completeBaseName(); // base = "archive.tar"
QString ext = fi.suffix(); // ext = "gz"
QString ext = fi.completeSuffix(); // ext = "tar.gz"
```

## 36 二进制文件读写

在上一章中，我们介绍了有关 `QFile` 和 `QFileInfo` 两个类的使用。我们提到，`QIODevice` 提供了 `read()`、`readLine()`等基本操作。同时，Qt 还提供了更高一级的操作：用于二进制的流 `QDataStream` 和用于文本流的 `QTextStream`。本节，我们将讲解有关 `QDataStream` 的使用以及一些技巧。下一章则是 `QTextStream` 的相关内容。

`QDataStream` 提供了基于 `QIODevice` 的二进制数据的序列化。数据流是一种二进制流，这种流完全不依赖于底层操作系统、CPU 或者字节顺序（大端或小端）。例如，在安装了 Windows 平台的 PC 上面写入的一个数据流，可以不经任何处理，直接拿到运行了 Solaris 的 SPARC 机器上读取。由于数据流就是二进制流，因此我们也可以直接读写没有编码的二进制数据，例如图像、视频、音频等。

`QDataStream` 既能够存取 C++ 基本类型，如 `int`、`char`、`short` 等，也可以存取复杂的数据类型，例如自定义的类。实际上，`QDataStream` 对于类的存储，是将复杂的类分割为很多基本单元实现的。

结合 `QIODevice`，`QDataStream` 可以很方便地对文件、网络套接字等进行读写操作。我们从代码开始看起：

```
QFile file("file.dat");
file.open(QIODevice::WriteOnly);
QDataStream out(&file);
out << QString("the answer is");
out << (qint32)42;
```

在这段代码中，我们首先打开一个名为 `file.dat` 的文件（注意，我们为简单起见，并没有检查文件打开是否成功，这在正式程序中是不允许的）。然后，我们将刚刚创建的 `file` 对象的指针传递给一个 `QDataStream` 实例 `out`。类似于 `std::cout` 标准输出流，`QDataStream` 也重载了输出重定向 `<<` 运算符。后面的代码就很简单了：将“the answer is”和数字 42 输出到数据流（如果你不明白这句话的意思，这可是宇宙终极问题的答案 ;-P 请自行搜索《银河系漫游指南》）。由于我们的 `out` 对象建立在 `file` 之上，因此相当于将宇宙终极问题的答案写入 `file`。

需要指出一点：最好使用 `Qt` 整型来进行读写，比如程序中的 `qint32`。这保证了在任意平台和任意编译器都能够有相同的行为。

我们通过一个例子来看看 `Qt` 是如何存储数据的。例如 `char *`字符串，在存储时，会首先存储该字符串包括 `\0` 结束符的长度（32 位整型），然后是字符串的内容以及结束符 `\0`。在读取时，先以 32 位整型读出整个的长度，然后按照这个长度取出整个字符串的内容。

但是，如果你直接运行这段代码，你会得到一个空白的 `file.dat`，并没有写入任何数据。这是因为我们的 `file` 没有正常关闭。为性能起见，数据只有在文件关闭时才会真正写入。因此，我们必须在最后添加一行代码：

```
file.close(); // 如果不想关闭文件，可以使用 file.flush();
```

重新运行一下程序，你就得到宇宙终极问题的答案了。

我们已经获得宇宙终极问题的答案了，下面，我们要将这个答案读取出来：

```
QFile file("file.dat");  
file.open(QIODevice::ReadOnly);  
QDataStream in(&file);  
QString str;  
qint32 a;  
in >> str >> a;
```

这段代码没什么好说的。唯一需要注意的是，你必须按照写入的顺序，将数据读取出来。也就是说，程序数据写入的顺序必须预先定义好。在这个例子中，我们首先写入字符串，然后写入数字，那么就首先读出来的就是字符串，然后才是数字。顺序颠倒的话，程序行为是不确定的，严重时会造成程序崩溃。

由于二进制流是纯粹的字节数据，带来的问题是，如果程序不同版本之间按照不同的方式读取（前面说过，`Qt` 保证读写内容的一致，但是并不能保证不同 `Qt` 版本之间的一致），数据就会出现错误。因此，我们必须提供一种机制来确保不同版本之间的一致性。通常，我们会使用如下的代码写入：

```

QFile file("file.dat");
file.open(QIODevice::WriteOnly);
QDataStream out(&file);

// 写入魔术数字和版本
out << (quint32)0xA0B0C0D0;
out << (qint32)123;

out.setVersion(QDataStream::Qt_4_0);

// 写入数据
out << lots_of_interesting_data;

```

这里，我们增加了两行代码：

```
out << (quint32)0xA0B0C0D0;
```

用于写入魔术数字。所谓魔术数字，是二进制输出中经常使用的一种技术。二进制格式是人不可读的，并且通常具有相同的后缀名（比如 `dat` 之类），因此我们没有办法区分两个二进制文件哪个是合法的。所以，我们定义的二进制格式通常具有一个魔术数字，用于标识文件的合法性。在本例中，我们在文件最开始写入 `0xA0B0C0D0`，在读取的时候首先检查这个数字是不是 `0xA0B0C0D0`。如果不是的话，说明这个文件不是可识别格式，因此根本不需要去继续读取。一般二进制文件都会有这么一个魔术数字，例如 Java 的 `class` 文件的魔术数字就是 `0xCAFEBAFE`，使用二进制查看器就可以查看。魔术数字是一个 32 位的无符号整型，因此我们使用 `quint32` 来得到一个平台无关的 32 位无符号整型。

接下来一行，

```
out << (qint32)123;
```

是标识文件的版本。我们用魔术数字标识文件的类型，从而判断文件是不是合法的。但是，文件的不同版本之间也可能存在差异：我们可能在第一版保存整型，第二版可能保存字符串。为了标识不同的版本，我们只能将版本写入文件。比如，现在我们的版本是 123。

下面一行还是有关版本的：

```
out.setVersion(QDataStream::Qt_4_0);
```

上面一句是文件的版本号，但是，Qt 不同版本之间的读取方式可能也不一样。这样，我们就得指定 Qt 按照哪个版本去读。这里，我们指定以 Qt 4.0 格式去读取内容。

当我们这样写入文件之后，我们在读取的时候就需要增加一系列的判断：

```

QFile file("file.dat");
file.open(QIODevice::ReadOnly);
QDataStream in(&file);

// 检查魔术数字
quint32 magic;
in >> magic;
if (magic != 0xA0B0C0D0) {
    return BAD_FILE_FORMAT;
}

// 检查版本
quint32 version;
in >> version;
if (version < 100) {
    return BAD_FILE_TOO_OLD;
}
if (version > 123) {
    return 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_version_1_2;
}
in >> other_interesting_data;

```

这段代码就是按照前面的解释进行的。首先读取魔术数字，检查文件是否合法。如果合法，读取文件版本：小于 100 或者大于 123 都是不支持的。如果在支持的版本范围内（ $100 \leq \text{version} \leq 123$ ），则当是小于等于 110 的时候，按照 Qt\_3\_2 的格式读取，否则按照 Qt\_4\_0 的格式读取。当设置完这些参数之后，开始读取数据。

至此，我们介绍了有关 QDataStream 的相关内容。那么，既然 QIODevice 提供了 read()、readLine()之类的函数，为什么还要有 QDataStream 呢？QDataStream 同 QIODevice 有什么区别？区别在于，有些 QIODevice 支持随机读写，而 QDataStream 提供的是流的形式，不允许随机读写。我们通过下面一段代码看看什么是流的形式：

```

QFile file("file.dat");
file.open(QIODevice::ReadWrite);

QDataStream stream(&file);
QString str = "the answer is 42";
QString strout;

stream << str;
file.flush();
stream >> strout;

```

在这段代码中，我们首先向文件中写入数据，紧接着把数据读出来。有什么问题吗？运行之后你会发现，`strout` 实际是空的。为什么没有读取出来？我们不是已经添加了 `file.flush()`；语句吗？原因并不在于文件有没有写入，而是在于我们使用的是“流”。所谓流，就像水流一样，它的游标会随着输出向后移动。当使用 `<<` 操作符输出之后，流的游标已经到了最后，此时你再去读，当然什么也读不到了。所以你需要在输出之后重新把游标设置为 0 的位置才能够继续读取。具体代码片段如下：

```

stream << str;
stream.device()->seek(0);
stream >> strout;

```

## 37 文本文件读写

上一章我们介绍了有关二进制文件的读写。二进制文件比较小巧，却不是人可读的格式。而文本文件是一种人可读的文件。为了操作这种文件，我们需要使用 `QTextStream` 类。

`QTextStream` 和 `QDataStream` 的使用类似，只不过它是操作纯文本文件的。另外，像 XML、HTML 这种，虽然也是文本文件，可以由 `QTextStream` 生成，但 Qt 提供了更方便的 XML 操作类，这里就不包括这部分内容了。

`QTextStream` 会自动将 Unicode 编码同操作系统的编码进行转换，这一操作对开发人员是透明的。它也会将换行符进行转换，同样不需要自己处理。`QTextStream` 使用 16 位的 `QChar` 作为基础的数据存储单位，同样，它也支持 C++ 标准类型，如 `int` 等。实际上，这是将这种标准类型与字符串进行了相互转换。

`QTextStream` 同 `QDataStream` 的使用基本一致，例如下面的代码将把“The answer is 42”写入到 `file.txt` 文件中：

```

QFile data("file.txt");
if (data.open(QFile::WriteOnly | QIODevice::Truncate)) {

```

```

QTextStream out(&data);
out << "The answer is " << 42;
}

```

这里，我们在 `open()` 函数中增加了 `QIODevice::Truncate` 打开方式。我们可以从下表中看到这些打开方式的区别：

枚举值	描述
<code>QIODevice::NotOpen</code>	未打开
<code>QIODevice::ReadOnly</code>	以只读方式打开
<code>QIODevice::WriteOnly</code>	以只写方式打开
<code>QIODevice::ReadWrite</code>	以读写方式打开
<code>QIODevice::Append</code>	以追加的方式打开，新增加的内容将被追加到文件末尾
<code>QIODevice::Truncate</code>	以重写的方式打开，在写入新的数据时会将原有数据全部清除，游标设置在文件开头。
<code>QIODevice::Text</code>	在读取时，将行结束符转换成 <code>\n</code> ；在写入时，将行结束符转换成本地格式，例如 Win32 平台上是 <code>\r\n</code>
<code>QIODevice::Unbuffered</code>	忽略缓存

我们在这里使用了 `QFile::WriteOnly | QIODevice::Truncate`，也就是以只写并且覆盖已有内容的形式操作文件。注意，`QIODevice::Truncate` 会直接将文件内容清空。

虽然 `QTextStream` 的写入内容与 `QDataStream` 一致，但是读取时却会有些困难：

```

QFile data("file.txt");
if (data.open(QFile::ReadOnly)) {
    QTextStream in(&data);
    QString str;
    int ans = 0;
}

```



```
in >> str >> ans;
}
```

在使用 `QDataStream` 的时候，这样的代码很方便，但是使用了 `QTextStream` 时却有所不同：读出的时候，`str` 里面将是 `The answer is 42`，`ans` 是 `0`。这是因为以文本形式写入数据，是没有数据之间的分隔的。还记得我们前面曾经说过，使用 `QDataStream` 写入的时候，实际上会在要写入的内容前面，额外添加一个这段内容的长度值。而文本文件则没有类似的操作。因此，使用文本文件时，很少会将其分割开来读取，而是使用诸如

`QTextStream::readLine()` 读取一行，使用 `QTextStream::readAll()` 读取所有文本这种函数，之后再对获得的 `QString` 对象进行处理。

默认情况下，`QTextStream` 的编码格式是 `Unicode`，如果我们需要使用另外的编码，可以使用

```
stream.setCodec("UTF-8");
```

这样的函数进行设置。

另外，为方便起见，`QTextStream` 同 `std::cout` 一样提供了很多描述符，被称为 `stream manipulators`。因为文本文件是供人去读的，自然需要良好的格式（相比而言，二进制文件就没有这些问题，只要数据准确就可以了）。这些描述符是一些函数的简写，我们可以从文档中找到：

描述符	等价于
<code>bin</code>	<code>setIntegerBase(2)</code>
<code>oct</code>	<code>setIntegerBase(8)</code>
<code>dec</code>	<code>setIntegerBase(10)</code>
<code>hex</code>	<code>setIntegerBase(16)</code>
<code>showbase</code>	<code>setNumberFlags(numberFlags()   ShowBase)</code>
<code>forcesign</code>	<code>setNumberFlags(numberFlags()   ForceSign)</code>

forcepoint	setNumberFlags(numberFlags()   ForcePoint)
noshowbase	setNumberFlags(numberFlags() & ~ShowBase)
noforcesign	setNumberFlags(numberFlags() & ~ForceSign)
noforcepoint	setNumberFlags(numberFlags() & ~ForcePoint)
uppercasebase	setNumberFlags(numberFlags()   UppercaseBase)
uppercasedigits	setNumberFlags(numberFlags()   UppercaseDigits)
lowercasebase	setNumberFlags(numberFlags() & ~UppercaseBase)
lowercasedigits	setNumberFlags(numberFlags() & ~UppercaseDigits)
fixed	setRealNumberNotation(FixedNotation)
scientific	setRealNumberNotation(ScientificNotation)
left	setFieldAlignment(AlignLeft)
right	setFieldAlignment(AlignRight)
center	setFieldAlignment(AlignCenter)
endl	operator<<("\n")和 flush()
flush	flush()
reset	reset()

ws                                    skipWhiteSpace()

bom                                    setGenerateByteOrderMark(true)

这些描述符只是一些函数的简写。例如，我们想要输出 12345678 的二进制形式，那么可以直接使用

```
out << bin << 12345678;
```

就可以了。这等价于

```
out.setIntegerBase(2);  
out << 12345678;
```

更复杂的，如果我们想要舒服 1234567890 的带有前缀、全部字母大写的十六进制格式（0xBC614E），那么只要使用

```
out << showbase << uppercasedigits << hex << 12345678;
```

即可。

不仅是 QIODevice，QTextStream 也可以直接把内容输出到 QString。例如

```
QString str;  
QTextStream(&str) << oct << 31 << " " << dec << 25 << endl;
```

这提供了一种简单的处理字符串内容的方法。

## 38 存储容器

存储容器（containers）有时候也被称为集合（collections），是能够在内存中存储其它特定类型的对象，通常是一些常用的数据结构，一般是通用模板类的形式。C++ 提供了一套完整的解决方案，作为标准模板库（Standard Template Library）的组成部分，也就是常说的 STL。

Qt 提供了另外一套基于模板的容器类。相比 STL，这些容器类通常更轻量、更安全、更容易使用。如果你对 STL 不大熟悉，或者更喜欢 Qt 风格的 API，那么你就应该选择使用这些类。当然，你也可以在 Qt 中使用 STL 容器，没有任何问题。

本章的目的，是让你能够选择使用哪个容器，而不是告诉你这个类都有哪些函数。这个问题可以在文档中找到更清晰的回答。

Qt 的容器类提供了隐式数据共享、不可变的特性，并且为速度做了优化，具有较低的内存占用量等。另外一点比较重要的，它们是线程安全的。这些容器类是平台无关的，即不因编译器的不同而具有不同的实现；隐式数据共享，有时也被称作“写时复制（copy on write）”，这种技术允许在容器类中使用传值参数，但却不会出现额外的性能损失。遍历是容器类的重要操作。Qt 容器类提供了类似 Java 的遍历器语法，同样也提供了类似 STL 的遍历器语法，以方便用户选择自己习惯的编码方式。相比而言，Java 风格的遍历器更易用，是一种高层次的函数；而 STL 风格的遍历器更高效，同时能够支持 Qt 和 STL 的通用算法。最后一点，在一些嵌入式平台，STL 往往是不可用的，这时你就只能使用 Qt 提供的容器类，除非你想自己创建。顺便提一句，除了遍历器，Qt 还提供了自己的 foreach 语法（C++ 11 也提供了类似的语法，但有所区别，详见[这里的](#) foreach 循环一节）。

Qt 提供了顺序存储容器：QList，QLinkedList，QVector，QStack 和 QQueue。对于绝大多数应用程序，QList 是最好的选择。虽然它是基于数组实现的列表，但它提供了快速的向前添加和向后追加的操作。如果你需要链表，可以使用 QLinkedList。如果你希望所有元素占用连续地址空间，可以选择 QVector。QStack 和 QQueue 则是 LIFO 和 FIFO 的。

Qt 还提供了关联容器：QMap，QMultiMap，QHash，QMultiHash 和 QSet。带有“Multi”字样的容器支持在一个键上面关联多个值。“Hash”容器提供了基于散列函数的更快的查找，而非 Hash 容器则是基于二分搜索的有序集合。

另外两个特例：QCache 和 QContiguousCache 提供了在有限缓存空间中的高效 hash 查找。

我们将 Qt 提供的各个容器类总结如下：

- **QList<T>**：这是至今为止提供的最通用的容器类。它将给定的类型 T 的对象以列表的形式进行存储，与一个整型的索引关联。QList 在内部使用数组实现，同时提供基于索引的快速访问。我们可以使用 QList::append() 和 QList::prepend() 在列表尾部或头部添加元素，也可以使用 QList::insert() 在中间插入。相比其它容器类，QList 专门为这种修改操作作了优化。QStringList 继承自 QList<QString>。
- **QLinkedList<T>**：类似于 QList，除了它是使用遍历器进行遍历，而不是基于整数索引的随机访问。对于在中部插入大量数据，它的性能要优于 QList。同时具有更好的遍历器语义（只要数据元素存在，QLinkedList 的遍历器就会指向一个合法元素，相比之下，当插入或删除数据时，QList 的遍历器就会指向一个非法值）。
- **QVector<T>**：用于在内存的连续区存储一系列给定类型的值。在头部或中间插入数据可能会非常慢，因为这会引起大量数据在内存中的移动。

- **QStack<T>**: 这是 **QVector** 的子类, 提供了后进先出 (LIFO) 语义。相比 **QVector**, 它提供了额外的函数: **push()**, **pop()**和 **top()**。
- **QQueue<T>**: 这是 **QList** 的子类, 提供了先进先出 (FIFO) 语义。相比 **QList**, 它提供了额外的函数: **enqueue()**, **dequeue()**和 **head()**。
- **QSet<T>**: 提供单值的数学上面的集合, 具有快速的查找性能。
- **QMap<Key, T>**: 提供了字典数据结构 (关联数组), 将类型 **T** 的值同类型 **Key** 的键关联起来。通常, 每个键与一个值关联。**QMap** 以键的顺序存储数据; 如果顺序无关, **QHash** 提供了更好的性能。
- **QMultiMap<Key, T>**: 这是 **QMap** 的子类, 提供了多值映射: 一个键可以与多个值关联。
- **QHash<Key, T>**: 该类同 **QMap** 的接口几乎相同, 但是提供了更快的查找。**QHash** 以字母顺序存储数据。
- **QMultiHash<Key, T>**: 这是 **QHash** 的子类, 提供了多值散列。

所有的容器都可以嵌套。例如, **QMap<QString, QList<int>>** 是一个映射, 其键是 **QString** 类型, 值是 **QList<int>** 类型, 也就是说, 每个值都可以存储多个 **int**。这里需要注意的是, **C++** 编译器会将连续的两个 **>** 当做输入重定向运算符, 因此, 这里的两个 **>** 中间必须有一个空格。

能够存储在容器中的数据必须是**可赋值数据类型**。所谓可赋值数据类型, 是指具有默认构造函数、拷贝构造函数和赋值运算符的类型。绝大多数数据类型, 包括基本类型, 比如 **int** 和 **double**, 指针, **Qt** 数据类型, 例如 **QString**、**QDate** 和 **QTime**, 都是可赋值数据类型。但是, **QObject** 及其子类 (**QWidget**、**QTimer** 等) 都不是。也就是说, 你不能使用 **QList<QWidget>** 这种容器, 因为 **QWidget** 的拷贝构造函数和赋值运算符不可用。如果你需要这种类型的容器, 只能存储其指针, 也就是 **QList<QWidget \***>。

如果要使用 **QMap** 或者 **QHash**, 作为键的类型必须提供额外的辅助函数。**QMap** 的键必须提供 **operator<()**重载, **QHash** 的键必须提供 **operator==()**重载和一个名字是 **qHash()** 的全局函数。

作为例子, 我们考虑如下的代码:

```
struct Movie
{
    int id;
    QString title;
```

```
    QDate releaseDate;
};
```

作为 `struct`，我们当做纯数据类使用。这个类没有额外的构造函数，因此编译器会为我们生成一个默认构造函数。同时，编译器还会生成默认的拷贝构造函数和赋值运算符。这就满足了将其放入容器类存储的条件：

```
QList<Movie> movs;
```

Qt 容器类可以直接使用 `QDataStream` 进行存取。此时，容器中所存储的类型必须也能够使用 `QDataStream` 进行存储。这意味着，我们需要重载 `operator<<()`和 `operator>>()`运算符：

```
QDataStream &operator<<(QDataStream &out, const Movie &movie)
{
    out << (quint32)movie.id << movie.title
        << movie.releaseDate;
    return out;
}

QDataStream &operator>>(QDataStream &in, Movie &movie)
{
    quint32 id;
    QDate date;

    in >> id >> movie.title >> date;
    movie.id = (int)id;
    movie.releaseDate = date;
    return in;
}
```

根据数据结构的相关内容，我们有必要对这些容器类的算法复杂性进行定量分析。算法复杂度关心的是在数据量增长时，容器的每一个函数究竟有多快（或者多慢）。例如，向 `QLinkedList` 中部插入数据是一个相当快的操作，并且与 `QLinkedList` 中已经存储的数据量无关。另一方面，如果 `QVector` 中已经保存了大量数据，向 `QVector` 中部插入数据会非常慢，因为在内存中，有一半的数据必须移动位置。为了描述算法复杂度，我们引入  $O$  记号（大写字母  $O$ ，读作“大  $O$ ”）：

- 常量时间： $O(1)$ 。如果一个函数的运行时间与容器中数据量无关，我们说这个函数是常量时间的。`QLinkedList::insert()`就是常量时间的。

- 对数时间： $O(\log n)$ 。如果一个函数的运行时间是容器数据量的对数关系，我们说这个函数是对数时间的。`qBinaryFind()`就是对数时间的。
- 线性时间： $O(n)$ 。如果一个函数的运行时间是容器数据量的线性关系，也就是说直接与数量相关，我们说这个函数是线性时间的。`QVector::insert()`就是线性时间的。
- 线性对数时间： $O(n \log n)$ 。线性对数时间要比线性时间慢，但是要比平方时间快。
- 平方时间： $O(n^2)$ 。平方时间与容器数据量的平方关系。

基于上面的表示，我们来看看 Qt 顺序容器的算法复杂度：

	查找	插入	前方添加	后方追加
<code>QLinkedList&lt;T&gt;</code>	$O(n)$	$O(1)$	$O(1)$	$O(1)$
<code>QList&lt;T&gt;</code>	$O(1)$	$O(n)$	统计 $O(1)$	统计 $O(1)$
<code>QVector&lt;T&gt;</code>	$O(1)$	$O(n)$	$O(n)$	统计 $O(1)$

上表中，所谓“统计”，意思是统计意义上的数据。例如“统计  $O(1)$ ”是说，如果只调用一次，其运行时间是  $O(n)$ ，但是如果调用多次（例如  $n$  次），则平均时间是  $O(1)$ 。

下表则是关联容器的算法复杂度：

	查找键		插入	
	平均	最坏	平均	最坏
<code>QMap&lt;Key, T&gt;</code>	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
<code>QMultiMap&lt;Key, T&gt;</code>	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
<code>QHash&lt;Key, T&gt;</code>	统计 $O(1)$	$O(n)$	$O(1)$	统计 $O(n)$

QSet<Key, T>                      统计 O(1)              O(n)              O(1)              统计 O(n)

QVector、QHash 和 QSet 的头部添加是统计意义上的  $O(\log n)$ 。然而，通过给定插入之前的元素个数来调用 `QVector::reserve()`、`QHash::reserve()` 和 `QSet::reserve()`，我们可以把复杂度降到  $O(1)$ 。我们会在下面详细讨论这个问题。

`QVector<T>`、`QString` 和 `QByteArray` 在连续内存空间中存储数据。`QList<T>` 维护指向其数据的指针数组，提供基于索引的快速访问（如果 `T` 就是指针类型，或者是与指针大小相同的其它类型，那么 `QList` 的内部数组中存的就是其实际值，而不是其指针）。

`QHash<Key, T>` 维护一张散列表，其大小与散列中数据量相同。为避免每次插入数据都要重新分配数据空间，这些类都提供了多余实际值的数据位。

我们通过下面的代码来了解这一算法：

```
QString onlyLetters(const QString &in)
{
    QString out;
    for (int j = 0; j < in.size(); ++j) {
        if (in[j].isLetter())
            out += in[j];
    }
    return out;
}
```

我们创建了一个字符串，每次动态追加一个字符。假设我们需要追加 15000 个字符。在算法运行过程中，当达到以下空间时，会进行重新分配内存空间，一共会有 18 次：4, 8, 12, 16, 20, 52, 116, 244, 500, 1012, 2036, 4084, 6132, 8180, 10228, 12276, 14324, 16372。最后，这个 `out` 对象一共有 16372 个 Unicode 字符，其中 15000 个是有实际数据的。

上面的分配数据有些奇怪，其实是有章可循的：

- `QString` 每次分配 4 个字符，直到达到 20。
- 在 20 到 4084 期间，每次分配大约一倍。准确地说，每次会分配下一个 2 的幂减 12。（某些内存分配器在分配 2 的幂数时会有非常差的性能，因为他们会占用某些字节做预订）
- 自 4084 起，每次多分配 2048 个字符（4096 字节）。这是有特定意义的，因为现代操作系统在重新分配一个缓存时，不会复制整个数据；物理内存页只是简单地被重新排序，只有第一页和最后一页的数据会被复制。



`QByteArray` 和 `QList<T>` 实际算法与 `QString` 非常类似。

对于那些能够使用 `memcpy()`（包括基本的 C++ 类型，指针类型和 Qt 的共享类）函数在内存中移动的数据类型，`QVector<T>` 也使用了类似的算法；对于那些只能使用拷贝构造函数和析构函数才能移动的数据类型，使用的是另外一套算法。由于后者的消耗更高，所以 `QVector<T>` 减少了超出空间时每次所要分配的额外内存数。

`QHash<Key, T>` 则是完全不同的形式。`QHash` 的内部散列表每次会增加 2 的幂数；每次增加时，所有数据都要重新分配到新的桶中，其计算公式是 `qHash(key) %`

`QHash::capacity()`（`QHash::capacity()` 就是桶的数量）。这种算法同样适用于 `QSet<T>` 和 `QCache<Key, T>`。如果你不明白“桶”的概念，请查阅数据结构的相关内容。

对于大多数应用程序，Qt 默认的增长算法已经足够。如果你需要额外的控制，`QVector<T>`、`QHash<Key, T>`、`QSet<T>`、`QString` 和 `QByteArray` 提供了一系列函数，用于检测和指定究竟要分配多少内存：

- `capacity()`：返回实际已经分配内存的元素数目（对于 `QHash` 和 `QSet`，则是散列表中桶的个数）
- `reserve(size)`：为指定数目的元素显式地预分配内存。
- `squeeze()`：释放那些不需要真实存储数据的内存空间。

如果你知道容器大约有多少数据，那么你可以通过调用 `reserve()` 函数来减少内存占用。如果已经将所有数据全部存入容器，则可以调用 `squeeze()` 函数，释放所有未使用的预分配空间。

## 39 遍历容器

上一节我们大致了解了有关存储容器的相关内容。对于所有的容器，最常用的操作就是遍历。本章我们将详细了解有关遍历器的内容。

尽管这个问题不是本章需要考虑的，但是我们还是需要来解释下，为什么要有遍历器。没有遍历器时，如果我们需要向外界提供一个列表，我们通常会将其返回：

```
QList<int> intlist() const
{
    return list;
}
```

这么做的问题是：向用户暴露了集合的内部实现。用户知道，原来你用的就是一个 `QList` 啊~那我就可以向里面增加东西了，或者修改其中的内容。有时这不是我们所期望的。很多时候，我们只是想提供用户一个集合，只允许用户知道这个集合中有什么，而不是对它进行修改。为此，我们希望有这么一种对象：通过它就能够提供一种通用的访问集合元素的方法，不管底层的集合是链表还是散列，都可以通过这种对象实现。这就是遍历器。

Qt 的容器类提供了两种风格的遍历器：Java 风格和 STL 风格。这两种风格的遍历器在通过非 `const` 函数对集合进行修改时都是不可用的。

## Java 风格的遍历器

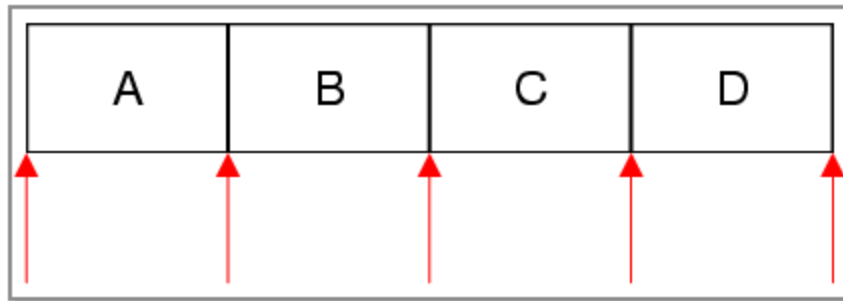
Java 风格的遍历器是在 Qt4 首先引入的，是 Qt 应用程序首先推荐使用的形式。这种风格比起 STL 风格的遍历器更方便。方便的代价就是不如后者高效。它们的 API 非常类似于 Java 的遍历器类，故名。

每一种容器都有两种 Java 风格的遍历器：一种提供只读访问，一种提供读写访问：

容器	只读遍历器	读写遍历器
<code>QList&lt;T&gt;, QQueue&lt;T&gt;</code>	<code>QListIterator&lt;T&gt;</code>	<code>QMutableListIterator&lt;T&gt;</code>
<code>QLinkedList&lt;T&gt;</code>	<code>QLinkedListIterator&lt;T&gt;</code>	<code>QMutableLinkedListIterator&lt;T&gt;</code>
<code>QVector&lt;T&gt;, QStack&lt;T&gt;</code>	<code>QVectorIterator&lt;T&gt;</code>	<code>QMutableVectorIterator&lt;T&gt;</code>
<code>QSet&lt;T&gt;</code>	<code>QSetIterator&lt;T&gt;</code>	<code>QMutableSetIterator&lt;T&gt;</code>
<code>QMap&lt;Key, T&gt;, QMultiMap&lt;Key, T&gt;</code>	<code>QMapIterator&lt;T&gt;</code>	<code>QMutableMapIterator&lt;T&gt;</code>
<code>QHash&lt;Key, T&gt;, QMultiHash&lt;Key, T&gt;</code>	<code>QHashIterator&lt;T&gt;</code>	<code>QMutableHashIterator&lt;T&gt;</code>

这里我们只讨论 `QList` 和 `QMap` 的遍历器。`QLinkedList`、`QVector` 和 `QSet` 的遍历器接口与 `QList` 的是一样的；`QHash` 遍历器的接口则同 `QMap` 是一样的。

不同于下面我们将要介绍的 STL 风格的遍历器，Java 风格的遍历器指向的是两个元素之间的位置，而不是指向元素本身。因此，它们可能会指向集合第一个元素之前的位置，也可能指向集合的最后一个元素之后的位置，如下图所示：



我们通过下面的代码看看如何使用这种遍历器：

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

QListIterator<QString> i(list);
while (i.hasNext()) {
    qDebug() << i.next();
}
```

首先，我们使用 `list` 对象创建一个遍历器。刚刚创建完成时，该遍历器位于第一个元素之前（也就是 A 之前）。我们通过调用 `hasNext()` 函数判断遍历器之后的位置上是否有元素。如果有，调用 `next()` 函数将遍历器跳过其后的元素。`next()` 函数返回刚刚跳过的元素。当然，我们也可以使用 `hasPrevious()` 和 `previous()` 函数来从尾部开始遍历，详细内容可以参考 API 文档。

`QListIterator` 是只读遍历器，不能插入或者删除数据。如果需要这些操作，我们可以使用 `QMutableListIterator`。来看下面的代码：

```
QMutableListIterator<int> i(list);
while (i.hasNext()) {
    if (i.next() % 2 != 0) {
        i.remove();
    }
}
```

这段代码使用 `QMutableListIterator` 遍历集合，如果其值是奇数则将其删除。在每次循环中都要调用 `next()` 函数。正如前面所说，它会跳过其后的一个元素。`remove()` 函数会删除

我们刚刚跳过的元素。调用 `remove()` 函数并不会将遍历器置位不可用，因此我们可以连续调用这个函数。向前遍历也是类似的，这里不再赘述。

如果我们需要修改已经存在的元素，使用 `setValue()` 函数。例如：

```
QMutableListIterator<int> i(list);
while (i.hasNext()) {
    if (i.next() > 128) {
        i.setValue(128);
    }
}
```

如同 `remove()` 函数，`setValue()` 也是对刚刚跳过的元素进行操作。实际上，`next()` 函数返回的是集合元素的非 `const` 引用，因此我们根本不需要调用 `setValue()` 函数：

```
QMutableListIterator<int> i(list);
while (i.hasNext()) {
    i.next() *= 2;
}
```

`QMapItrator` 也是类似的。例如，使用 `QMapItrator` 我们可以将数据从 `QMap` 复制到 `QHash`：

```
QMap<int, QWidget *> map;
QHash<int, QWidget *> hash;

QMapIterator<int, QWidget *> i(map);
while (i.hasNext()) {
    i.next();
    hash.insert(i.key(), i.value());
}
```

## STL 风格的遍历器

STL 风格的遍历器从 Qt 2.0 就开始提供。这种遍历器能够兼容 Qt 和 STL 的通用算法，并且为速度进行了优化。同 Java 风格遍历器类似，Qt 也提供了两种 STL 风格的遍历器：一种是只读访问，一种是读写访问。我们推荐尽可能使用只读访问，因为它们要比读写访问的遍历器快一些。

容器

只读遍历器

读写遍历器

<code>QList&lt;T&gt;,QQueue&lt;T&gt;</code>	<code>QList&lt;T&gt;::const_iterator</code>	<code>QList&lt;T&gt;::iterator</code>
<code>QLinkedList&lt;T&gt;</code>	<code>QLinkedList&lt;T&gt;::const_iterator</code>	<code>QLinkedList&lt;T&gt;::iterator</code>
<code>QVector&lt;T&gt;,QStack&lt;T&gt;</code>	<code>QVector&lt;T&gt;::const_iterator</code>	<code>QVector&lt;T&gt;::iterator</code>
<code>QSet&lt;T&gt;</code>	<code>QSet&lt;T&gt;::const_iterator</code>	<code>QSet&lt;T&gt;::iterator</code>
<code>QMap&lt;Key, T&gt;,QMultiMap&lt;Key, T&gt;</code>	<code>QMap&lt;Key, T&gt;::const_iterator</code>	<code>QMap&lt;Key, T&gt;::iterator</code>
<code>QHash&lt;Key, T&gt;,QMultiHash&lt;Key, T&gt;</code>	<code>QHash&lt;Key, T&gt;::const_iterator</code>	<code>QHash&lt;Key, T&gt;::iterator</code>

STL 风格的遍历器具有类似数组指针的行为。例如，我们可以使用 `++` 运算符让遍历器移动到下一个元素，使用 `*` 运算符获取遍历器所指的元素。对于 `QVector` 和 `QStack`，虽然它们是在连续内存区存储元素，遍历器类型是 `typedef T*,const_iterator` 类型则是 `typedef const T*`。

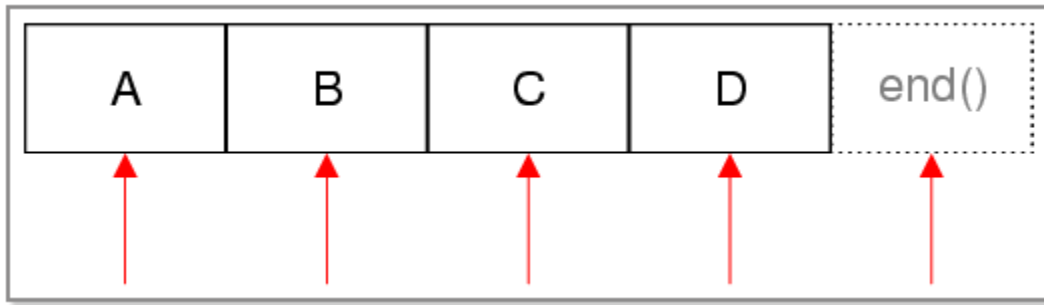
我们还是以 `QList` 和 `QMap` 为例，理由如上。下面是有关 `QList` 的相关代码：

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

QList<QString>::iterator i;
for (i = list.begin(); i != list.end(); ++i) {
    *i = (*i).toLower();
}
```

不同于 Java 风格遍历器，STL 风格遍历器直接指向元素本身。容器的 `begin()` 函数返回指向该容器第一个元素的遍历器；`end()` 函数返回指向该容器最后一个元素之后的元素的遍历器。`end()` 实际是一个非法位置，永远不可达。这是为跳出循环做的一个虚元素。如果集合是空的，`begin()` 等于 `end()`，我们就不能执行循环。

下图是 STL 风格遍历器的示意图：



我们使用 `const_iterator` 进行只读访问，例如：

```
QList<QString>::const_iterator i;
for (i = list.constBegin(); i != list.constEnd(); ++i) {
    qDebug() << *i;
}
```

`QMap` 和 `QHash` 的遍历器，`*` 运算符返回集合键值对。下面的代码，我们打印出 `QMap` 的所有元素：

```
QMap<int, int> map;

QMap<int, int>::const_iterator i;
for (i = map.constBegin(); i != map.constEnd(); ++i) {
    qDebug() << i.key() << ":" << i.value();
}
```

由于有隐式数据共享（我们会在后面的章节介绍该部分内容），即使一个函数返回集合中元素的值也不会有很大的代价。Qt API 包含了很多以值的形式返回 `QList` 或 `QStringList` 的函数（例如 `QSplitter::sizes()`）。如果你希望使用 STL 风格的遍历器遍历这样的元素，应该使用容器的拷贝，例如：

```
// 正确的方式
const QList<QString> sizes = splitter->sizes();
QList<QString>::const_iterator i;
for (i = sizes.begin(); i != sizes.end(); ++i)
    ...

// 错误的方式
QList<QString>::const_iterator i;
for (i = splitter->sizes().begin();
     i != splitter->sizes().end(); ++i)
    ...
```

这个问题不存在于那些返回集合的 `const` 或非 `const` 引用的函数。隐式数据共享对 STL 风格遍历器造成的另外影响是，在容器上运行着非 `const` 遍历器的时候，不能对容器进行拷贝。Java 风格的遍历器没有这个问题。

## foreach 关键字

如果我们仅仅想要遍历集合所有元素，我们可以使用 Qt 的 `foreach` 关键字。这个关键字是 Qt 特有的，通过预处理器进行处理。C++ 11 也提供了自己的 `foreach` 关键字，不过与此还是有[区别](#)的。

`foreach` 的语法是 `foreach (variable, container)`。例如，我们使用 `foreach` 对 `QLinkedList` 进行遍历：

```
QLinkedList<QString> list;
...
QString str;
foreach (str, list) {
    qDebug() << str;
}
```

这段代码与下面是等价的：

```
QLinkedList<QString> list;
...
QLinkedListIterator<QString> i(list);
while (i.hasNext()) {
    qDebug() << i.next();
}
```

如果类型名中带有逗号，比如 `QPair<int, int>`，我们只能像上面一样，先创建一个对象，然后使用 `foreach` 关键字。如果没有逗号，则可以直接在 `foreach` 关键字中使用新的对象，例如：

```
QLinkedList<QString> list;
...
foreach (const QString &str, list) {
    qDebug() << str;
}
```

Qt 会在 `foreach` 循环时自动拷贝容器。这意味着，如果在遍历时修改集合，对于正在进行的遍历是没有影响的。即使不修改容器，拷贝也是会发生的。但是由于存在隐式数据共享，这种拷贝还是非常迅速的。

因为 `foreach` 创建了集合的拷贝，使用集合的非 `const` 引用也不能实际修改原始集合，所修改的只是这个拷贝。

## 40 隐式数据共享

Qt 中许多 C++ 类使用了隐式数据共享技术，来最大化资源利用率和最小化拷贝时的资源消耗。当作为参数传递时，具有隐式数据共享的类即安全又高效。在数据传递时，实际上只是传递了数据的指针（这一切都是隐含帮你完成的），而只有在函数发生需要写入的情况时，数据才会被拷贝（也就是通常所说的写时复制）。本章我们将介绍有关隐式数据共享的相关内容，以便为恰当地使用前面所介绍的容器夯实基础。

具有数据共享能力的类包含了一个指向共享数据块的指针。这个数据块包含了数据本身以及数据的引用计数。当共享对象创建出来时，引用计数被设置为 1。当新的对象引用到共享数据时，引用计数增加；当对象引用不再引用数据时，引用计数减少。当引用计数变为 0 时，共享数据被删除。

在我们操作共享数据时，实际有两种拷贝对象的方法：我们通常称其为深拷贝和浅拷贝。深拷贝意味着要重新构造一个全新的对象；浅拷贝则仅仅复制引用，也就是上面所说的那个指向共享数据块的指针。深拷贝对内存和 CPU 资源都是很昂贵的；浅拷贝则非常快速，因为它仅仅是设置一个新的指针，然后将引用计数加 1。具有隐式数据共享的对象，其赋值运算符使用的是浅拷贝来实现的。

这种隐式数据共享的好处是，程序不需要拥有不必要的重复数据，减少数据拷贝的需求。重复数据的代价是降低内存使用率（因为内存存储了更多重复的数据）。通过数据共享，对象可以更简单地作为值来传递以及从函数中返回。

隐式数据共享是在底层自动完成的，程序人员无需关心。这也是“隐式”一词的含义。从 Qt4 开始，即使在多线程程序中，隐式数据共享也是起作用的。在很多人看来，隐式数据共享和多线程是不兼容的，这是由引用计数的实现方式决定的。但是，Qt 使用了原子性的引用计数来避免多线程环境下可能出现的执行顺序打断的行为。需要注意的是，原子引用计数并不能保证线程安全，还是需要恰当的锁机制。这种观点对所有类似的场合都是适用的。原子引用计数能够保证的是，线程肯定操作自己的数据，线程自己的数据是安全的。总的来说，从 Qt4 开始，你可以放心使用隐式数据共享的类，即使在多线程环境下。



我们可以使用 `QSharedData` 和 `QSharedDataPointer` 类实现自己的隐式数据共享类。

当对象即将被修改,并且其引用计数大于 1 时,隐式数据共享自动将数据从共享块中拿出。隐式共享类必须控制其内部数据,在任何修改其数据的函数中,将数据自动取出。

`QPen` 使用了隐式数据共享技术,我们以 `QPen` 为例,看看隐式数据共享是如何起作用的:

```
void QPen::setStyle(Qt::PenStyle style)
{
    detach(); // 从共享区取出数据
    d->style = style; // 设置数据 (更新)
}

void QPen::detach()
{
    if (d->ref != 1) {
        ... // 执行深拷贝
    }
}
```

凡是支持隐式数据共享的 Qt 类都支持类似的操作。用户甚至不需要知道对象其实已经共享。因此,你应该把这样的类当作普通类一样,而不应该依赖于其共享的特色作一些“小动作”。事实上,这些类的行为同普通类一样,只不过添加了可能的共享数据的优点。因此,你大可以使用按值传参,而无须担心数据拷贝带来的性能问题。例如:

```
QPixmap p1, p2;
p1.load("image.bmp");
p2 = p1; // p1 和 p2 共享数据

QPainter paint;
paint.begin(&p2); // 从此, p2 与 p1 分道扬镳
paint.drawText(0,50, "Hi");
paint.end();
```

上例中, `p1` 和 `p2` 在 `QPainter::begin()` 一行之前都是共享数据的,直到这一语句。因为该语句开始, `p2` 就要被修改了。

注意,前面已经提到过,不要在使用了隐式数据共享的容器上,在有非 `const` STL 风格的遍历器正在遍历时复制容器。另外还有一点,对于 `QList` 或者 `QVector`,我们应该使用 `at()` 函数而不是 `[]` 操作符进行只读访问。原因是 `[]` 操作符既可以是左值又可以是右值,这让 Qt 容器很难判断到底是左值还是右值,这意味着无法进行隐式数据共享;而 `at()` 函数不能作左值,因此可以进行隐式数据共享。另外一点是,对于 `begin()`, `end()` 以及其他一些非 `const`

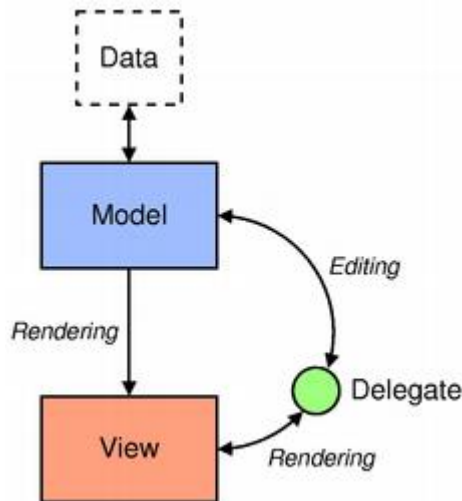
遍历器，由于数据可能改变，因此 Qt 会进行深复制。为了避免这一点，要尽可能使用 `const_iterator`、`constBegin()`和 `constEnd()`。

## 41 model/view 架构

有时，我们的系统需要显示大量数据，比如从数据库中读取数据，以自己的方式显示在自己的应用程序的界面中。早期的 Qt 要实现这个功能，需要定义一个组件，在这个组件中保存一个数据对象，比如一个列表。我们对这个列表进行查找、插入等的操作，或者把修改的地方写回，然后刷新组件进行显示。这个思路很简单，也很清晰，但是对于大型程序，这种设计就显得苍白无力。比如，在一个大型系统中，你的数据可能很大，全部存入一个组件的数据对象中，效率会很低，并且这样的设计也很难在不同组件之间共享数据。如果你要几个组件共享一个数据对象，要么你就要用存取函数公开这个数据对象，要么你就必须把这个数据对象放进不同的组件分别进行维护。

Smalltalk 语言发明了一种崭新的实现，用来解决这个问题，这就是著名的 MVC 模型。对这个模型无需多言。MVC 是 Model-View-Controller 的简写，即模型-视图-控制器。在 MVC 中，模型负责获取需要显示的数据，并且存储这些数据的修改。每种数据类型都有它自己对应的模型，但是这些模型提供一个相同的 API，用于隐藏内部实现。视图用于将模型数据显示给用户。对于数量很大的数据，或许只显示一小部分，这样就能很好的提高性能。控制器是模型和视图之间的媒介，将用户的动作解析成对数据的操作，比如查找数据或者修改数据，然后转发给模型执行，最后再将模型中需要被显示的数据直接转发给视图进行显示。MVC 的核心思想是分层，不同的层应用不同的功能。

Qt 4 开始，引入了类似的 model/view 架构来处理数据和面向最终用户的显示之间的关系。当 MVC 的 V 和 C 结合在一起，我们就得到了 model/view 架构。这种架构依然将数据和界面分离，但是框架更为简单。同样，这种架构也允许使用不同界面显示同一数据，也能够在不改变数据的情况下添加新的显示界面。为了处理用户输入，我们还引入了委托（delegate）。引入委托的好处是，我们能够自定义数据项的渲染和编辑。



Model View 概览

如上图所示，模型与数据源进行交互，为框架中其它组件提供接口。这种交互的本质在于数据源的类型以及模型的实现方式。视图从模型获取模型索引，这种索引就是数据项的引用。通过将这个模型索引反向传给模型，视图又可以从数据源获取数据。在标准视图中，委托渲染数据项；在需要编辑数据时，委托使用直接模型索引直接与模型进行交互。

总的来说，model/view 架构将传统的 MV 模型分为三部分：模型、视图和委托。每一个组件都由一个抽象类定义，这个抽象类提供了基本的公共接口以及一些默认实现。模型、视图和委托则使用信号槽进行交互：

- 来自模型的信号通知视图，其底层维护的数据发生了改变；
- 来自视图的信号提供了有关用户与界面进行交互的信息；
- 来自委托的信号在用户编辑数据项时使用，用于告知模型和视图编辑器的状态。

所有的模型都是 `QAbstractItemModel` 的子类。这个类定义了供视图和委托访问数据的接口。模型并不存储数据本身。这意味着，你可以将数据存储在一个数据结构中、另外的类中、文件中、数据库中，或者其他你能想到的东西中。我们将在后面再详细讨论这些内容。

`QAbstractItemModel` 提供的接口足够灵活，足以应付以表格、列表和树的形式显示的数据。但是，如果你需要为列表或者表格设计另外的模型，直接继承 `QAbstractListModel` 和 `QAbstractTableModel` 类可能更好一些，因为这两个类已经实现了很多通用函数。关于这部分内容，我们也会在后文中详述。

Qt 内置了许多标准模型：

- **QStringListModel**: 存储简单的字符串列表。
- **QStandardItemModel**: 可以用于树结构的存储，提供了层次数据。
- **QFileSystemModel**: 本地系统的文件和目录信息。
- **QSqlQueryModel**、**QSqlTableModel** 和 **QSqlRelationalTableModel**: 存取数据库数据。

正如上面所说，如果这些标准模型不能满足你的需要，就必须继承 **QAbstractItemModel**、**QAbstractListModel** 或者 **QAbstractTableModel**，创建自己的模型类。

Qt 还提供了一系列预定义好的视图：**QListView** 用于显示列表，**QTableView** 用于显示表格，**QTreeView** 用于显示层次数据。这些类都是 **QAbstractItemView** 的子类。这意味着，如果你要创建新的视图类，则可以继承 **QAbstractItemView**。

**QAbstractItemDelegate** 则是所有委托的抽象基类。自 Qt 4.4 依赖，默认的委托实现是 **QStyledItemDelegate**。但是，**QStyledItemDelegate** 和 **QItemDelegate** 都可以作为视图的编辑器，二者的区别在于，**QStyledItemDelegate** 使用当前样式进行绘制。在实现自定义委托时，推荐使用 **QStyledItemDelegate** 作为基类，或者结合 Qt style sheets。

如果你觉得 model/view 模型过于复杂，或者有很多功能是用不到的，Qt 还有一系列方便使用的类。这些类都是继承自标准的视图类，并且继承了标准模型。这些类并不是为其他类继承而准备的，只是为了使用方便。它们包括 **QListWidget**、**QTreeWidget** 和 **QTableWidget**。这些类远不如视图类灵活，不能使用另外的模型，因此只适用于简单的情形。

## 42 QListWidget、QTreeWidget 和 QTableWidget

上一章我们了解了 model/view 架构的基本概念。现在我们从最简单的 **QListWidget**、**QTreeWidget** 和 **QTableWidget** 三个类开始了解最简单的 model/view 的使用。这部分内容的确很难组织。首先，从最标准的 model/view 开始，往往会纠结于复杂的代码；但是，如果从简单的 **QListWidget**、**QTreeWidget** 和 **QTableWidget** 开始，由于这三个类都是继承自各自的 view 类，很难避免 model/view 的相关内容。于是，我们这部分的组织是，首先进行简单的数据显示，更复杂的设置则放在后面的章节。

## QListWidget

我们要介绍的第一个是 QListWidget。先来看下面的代码示例：

```
label = new QLabel(this);
label->setFixedWidth(70);

listWidget = new QListWidget(this);

new QListWidgetItem(QIcon(":/Chrome.png"), tr("Chrome"), listWidget);
new QListWidgetItem(QIcon(":/Firefox.png"), tr("Firefox"), listWidget);

listWidget->addItem(new QListWidgetItem(QIcon(":/IE.png"), tr("IE")));
listWidget->addItem(new QListWidgetItem(QIcon(":/Netscape.png"),
tr("Netscape")));
listWidget->addItem(new QListWidgetItem(QIcon(":/Opera.png"), tr("Opera")));
listWidget->addItem(new QListWidgetItem(QIcon(":/Safari.png"), tr("Safari")));
listWidget->addItem(new QListWidgetItem(QIcon(":/TheWorld.png"),
tr("TheWorld")));
listWidget->addItem(new QListWidgetItem(QIcon(":/Traveler.png"),
tr("Traveler")));

QListWidgetItem *newItem = new QListWidgetItem;
newItem->setIcon(QIcon(":/Maxthon.png"));
newItem->setText(tr("Maxthon"));
listWidget->insertItem(3, newItem);

QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(label);
layout->addWidget(listWidget);

setLayout(layout);

connect(listWidget, SIGNAL(currentTextChanged(QString)),
        label, SLOT(setText(QString)));
```

QListWidget 是简单的列表组件。当我们不需要复杂的列表时，可以选择 QListWidget。QListWidget 中可以添加 QListWidgetItem 类型作为列表项，QListWidgetItem 即可以有文本，也可以有图标。上面的代码显示了三种向列表中添加列表项的方法（实际是两种，后两种其实是一样的），我们的列表组件是 listWidget，那么，向 listWidget 添加列表项可以：第一，使用下面的语句

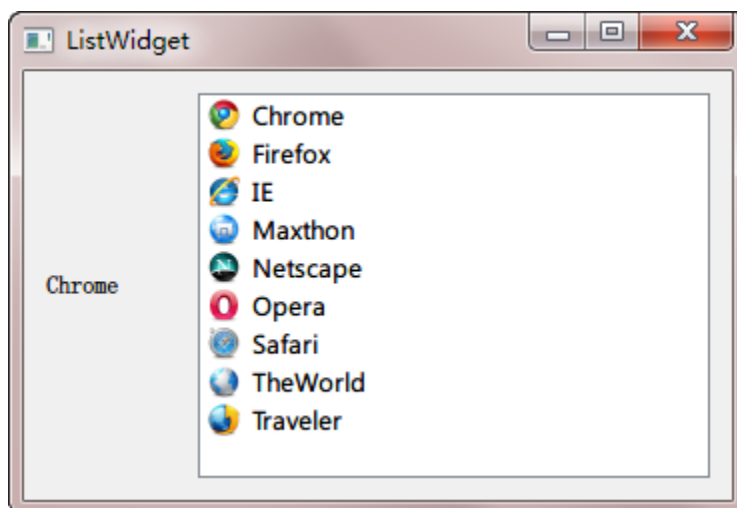
```
new QListWidgetItem(QIcon(":/Chrome.png"), tr("Chrome"), listWidget);
```

第二，使用

```
listWidget->addItem(new QListWidgetItem(QIcon(":/IE.png"), tr("IE")));  
// 或者  
QListWidgetItem *newItem = new QListWidgetItem;  
newItem->setIcon(QIcon(":/Maxthon.png"));  
newItem->setText(tr("Maxthon"));  
listWidget->insertItem(3, newItem);
```

注意这两种添加方式的区别：第一种需要在构造时设置所要添加到的 `QListWidget` 对象；第二种方法不需要这样设置，而是要调用 `addItem()` 或者 `insertItem()` 自行添加。如果你仔细查阅 `QListWidgetItem` 的构造函数，会发现有一个默认的 `type` 参数。该参数有两个合法值：`QListWidgetItem::Type`（默认）和 `QListWidgetItem::UserType`。如果我们继承 `QListWidgetItem`，可以设置该参数，作为我们子类的一种区别，以便能够在 `QListWidget` 区别处理不同子类。

我们的程序的运行结果如下：



我们可以利用 `QListWidget` 发出的各种信号来判断是哪个列表项被选择，具体细节可以参考文档。另外，我们也可以改变列表的显示方式。前面的列表是小图标显示，我们也可以更改为图标显示，只要添加一行语句：

```
listWidget->setViewMode(QListView::IconMode);
```

结果如下：



## QTreeWidget

我们要介绍的第二个组件是 `QTreeWidget`。顾名思义，这是用来展示树型结构（也就是层次结构）的。同前面说的 `QListWidget` 类似，这个类需要同另外一个辅助类 `QTreeWidgetItem` 一起使用。不过，既然是提供方面的封装类，即便是看上去很复杂的树，在使用这个类的时候也是显得比较简单的。当不需要使用复杂的 `QTreeView` 特性的时候，我们可以直接使用 `QTreeWidget` 代替。

下面我们使用代码构造一棵树：

```
QTreeWidget treeWidget;  
treeWidget.setColumnCount(1);  
  
QTreeWidgetItem *root = new QTreeWidgetItem(&treeWidget,  
                                              QStringList(QString("Root")));  
new QTreeWidgetItem(root, QStringList(QString("Leaf 1")));  
QTreeWidgetItem *leaf2 = new QTreeWidgetItem(root, QStringList(QString("Leaf  
2")));  
leaf2->setCheckState(0, Qt::Checked);  
  
QList<QTreeWidgetItem *> rootList;  
rootList << root;  
treeWidget.insertTopLevelItems(0, rootList);  
  
treeWidget.show();
```

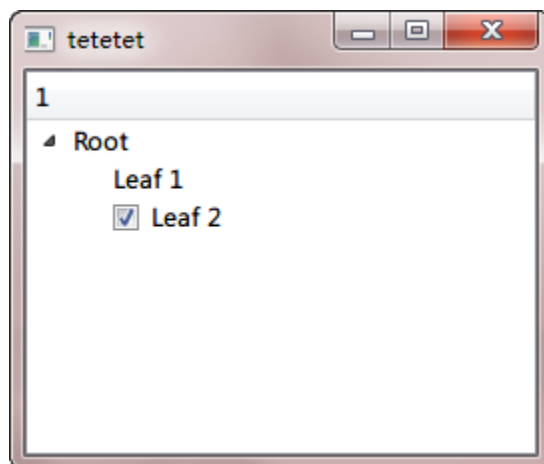
首先，我们创建了一个 `QTreeWidget` 实例。然后我们调用 `setColumnCount()` 函数设定栏数。这个函数的效果我们会在下文了解到。最后，我们向 `QTreeWidget` 添加

QTreeWidgetItem。QTreeWidgetItem 有很多重载的构造函数。我们在这里看看其中的一个，其余的请自行查阅文档。这个构造函数的签名如下：

```
QTreeWidgetItem(QTreeWidgetItem *parent, const QStringList &strings, int type = Type);
```

这里有 3 个参数，第一个参数用于指定这个项属于哪一个树，类似前面的 QListWidgetItem，如果指定了这个值，则意味着该项被直接添加到树中；第二个参数指定显示的文字；第三个参数指定其类型，同 QListWidgetItem 的 type 参数十分类似。值得注意的是，第二个参数是 QStringList 类型的，而不是 QString 类型。我们会在下文了解其含义。

在这段代码中，我们创建了作为根的 QTreeWidgetItemroot。然后添加了第一个叶节点，之后又添加一个，而这个则设置了可选标记。最后，我们将这个 root 添加到一个 QTreeWidgetItem 的列表，作为 QTreeWidgetItem 的数据项。此时你应该想到，既然 QTreeWidgetItem 接受 QList 作为项的数据，它就能够支持多棵树的一起显示，而不仅仅是单根树。下面我们来看看运行结果：



从代码来看，我们能够想象到这个样子，只是这个树的头上怎么会有一个 1？还记得我们跳过去的那个函数吗？下面我们修改一下代码看看：

```
QTreeWidgetItem treeWidget;  
  
QStringList headers;  
headers << "Name" << "Number";  
treeWidget.setHeaderLabels(headers);  
  
QStringList rootTextList;  
rootTextList << "Root" << "0";  
QTreeWidgetItem *root = new QTreeWidgetItem(&treeWidget, rootTextList);
```



```

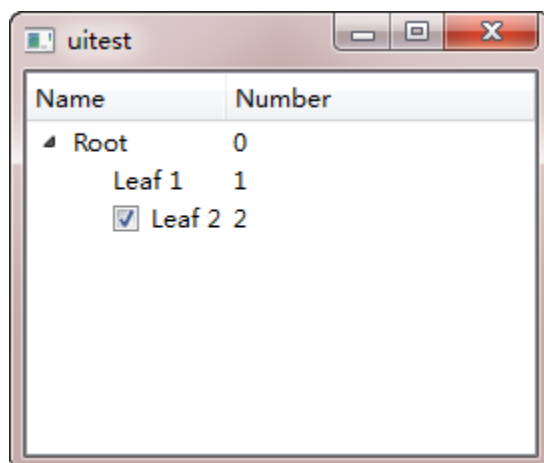
new QTreeWidgetItem(root, QStringList() << QString("Leaf 1") << "1");
QTreeWidgetItem *leaf2 = new QTreeWidgetItem(root,
        QStringList() << QString("Leaf 2") << "2");
leaf2->setCheckState(0, Qt::Checked);

QList<QTreeWidgetItem *> rootList;
rootList << root;
treeWidget.insertTopLevelItems(0, rootList);

treeWidget.show();

```

这次我们没有使用 `setColumnCount()`，而是直接使用 `QStringList` 设置了 headers，也就是树的表头。接下来我们使用的还是 `QStringList` 设置数据。这样，我们实现的是带有层次结构的树状表格。利用这一属性，我们可以比较简单地实现类似 Windows 资源管理器的界面。



如果你不需要显示这个表头，可以调用 `setHeaderHidden()` 函数将其隐藏。

## QTableWidget

我们要介绍的最后一个是 `QTableWidget`。`QTableWidget` 并不比前面的两个复杂到哪里去，这点我们可以从代码看出来：

```

QTableWidget tableWidget;
tableWidget.setColumnCount(3);
tableWidget.setRowCount(5);

QStringList headers;
headers << "ID" << "Name" << "Age" << "Sex";

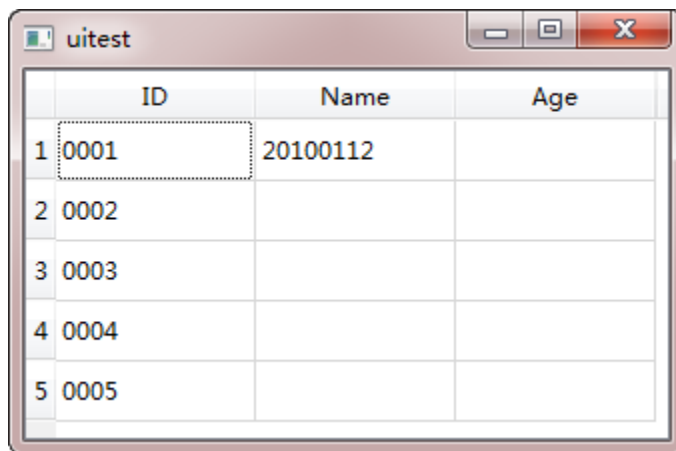
```

```
tableWidget.setHorizontalHeaderLabels(headers);

tableWidget.setItem(0, 0, new QTableWidgetItem(QString("0001")));
tableWidget.setItem(1, 0, new QTableWidgetItem(QString("0002")));
tableWidget.setItem(2, 0, new QTableWidgetItem(QString("0003")));
tableWidget.setItem(3, 0, new QTableWidgetItem(QString("0004")));
tableWidget.setItem(4, 0, new QTableWidgetItem(QString("0005")));
tableWidget.setItem(0, 1, new QTableWidgetItem(QString("20100112")));

tableWidget.show();
```

这段代码运行起来是这样子的：



	ID	Name	Age
1	0001	20100112	
2	0002		
3	0003		
4	0004		
5	0005		

首先我们创建了 `QTableWidget` 对象，然后设置列数和行数。接下来使用一个 `QStringList`，设置每一列的标题。我们可以通过调用 `setItem()` 函数来设置表格的单元格的数据。这个函数前两个参数分别是行索引和列索引，这两个值都是从 0 开始的，第三个参数则是一个 `QTableWidgetItem` 对象。Qt 会将这个对象放在第 `row` 行第 `col` 列的单元格中。有关 `QTableWidgetItem` 的介绍完全可以参见上面的 `QListWidgetItem` 和 `QTreeWidgetItem`。

## 43 QStringListModel

上一章我们已经了解到有关 `list`、`table` 和 `tree` 三个最常用的视图类的便捷类的使用。前面也提到过，由于这些类仅仅是提供方便，功能、实现自然不如真正的 `model/view` 强大。从本章起，我们将了解最基本的 `model/view` 模型的使用。

既然是 `model/view`，我们也会分为两部分：`model` 和 `view`。本章我们将介绍 Qt 内置的最简单的一个模型：`QStringListModel`。接下来，我们再介绍另外的一些内置模型，在此基

础上，我们将了解到 Qt 模型的基本架构，以便为最高级的应用——自定义模型——打下坚实的基础。

QStringListModel 是最简单的模型类，具备向视图提供字符串数据的能力。

QStringListModel 是一个可编辑的模型，可以为组件提供一系列字符串作为数据。我们可以将其看作是封装了 QStringList 的模型。QStringList 是一种很常用的数据类型，实际上是一个字符串列表（也就是 QList<QString>）。既然是列表，它也就是线性的数据结构，因此，QStringListModel 很多时候都会作为 QListView 或者 QComboBox 这种只有一列的视图组件的数据模型。

下面我们通过一个例子来看看 QStringListModel 的使用。首先是我们的构造函数：

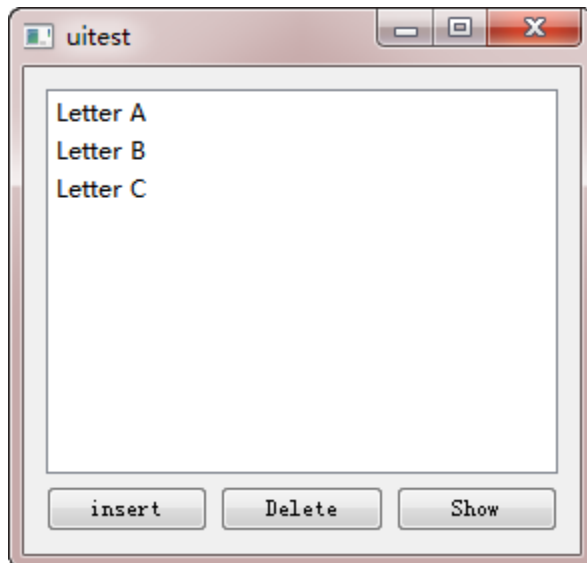
```
MyListView::MyListView()
{
    QStringList data;
    data << "Letter A" << "Letter B" << "Letter C";
    model = new QStringListModel(this);
    model->setStringList(data);

    listView = new QListView(this);
    listView->setModel(model);

    QHBoxLayout *btnLayout = new QHBoxLayout;
    QPushButton *insertBtn = new QPushButton(tr("insert"), this);
    connect(insertBtn, SIGNAL(clicked()), this, SLOT(insertData()));
    QPushButton *delBtn = new QPushButton(tr("Delete"), this);
    connect(delBtn, SIGNAL(clicked()), this, SLOT(deleteData()));
    QPushButton *showBtn = new QPushButton(tr("Show"), this);
    connect(showBtn, SIGNAL(clicked()), this, SLOT(showData()));
    btnLayout->addWidget(insertBtn);
    btnLayout->addWidget(delBtn);
    btnLayout->addWidget(showBtn);

    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->addWidget(listView);
    mainLayout->addLayout(btnLayout);
    setLayout(mainLayout);
}
```

我们不贴出完整的头文件了，只看源代码文件。首先，我们创建了一个 QStringList 对象，向其中插入了几个数据；然后将其作为 QStringListModel 的底层数据。这样，我们可以理解为，QStringListModel 将 QStringList 包装了起来。剩下来的只是简单的界面代码，这里不再赘述。试运行一下，程序应该是这样的：



接下来我们来看几个按钮的响应槽函数。

```
void MyListView::insertData()
{
    bool isOK;
    QString text = QDialog::getText(this, "Insert",
                                     "Please input new data:",
                                     QLineEdit::Normal,
                                     "You are inserting new data.",
                                     &isOK);

    if (isOK) {
        int row = listView->currentIndex().row();
        model->insertRows(row, 1);
        QModelIndex index = model->index(row);
        model->setData(index, text);
        listView->setCurrentIndex(index);
        listView->edit(index);
    }
}
```

首先是 `insertData()` 函数。我们使用 `QInputDialog::getText()` 函数要求用户输入数据。这是 Qt 的标准对话框，用于获取用户输入的字符串。这部分在前面的章节中已经讲解过。当用户点击了 OK 按钮，我们使用 `listView->currentIndex()` 函数，获取 `QListView` 当前行。这个函数的返回值是一个 `QModelIndex` 类型。我们会在后面的章节详细讲解这个类，现在只要知道这个类保存了三个重要的数据：行索引、列索引以及该数据属于哪一个模型。我们调用其 `row()` 函数获得行索引，该返回值是一个 `int`，也就是当前是第几行。然后我们向模型插入新的一行。`insertRows()` 函数签名如下：

```
bool insertRows(int row, int count, const QModelIndex &parent = QModelIndex());
```

该函数会将 `count` 行插入到模型给定的 `row` 的位置，新行的数据将会作为 `parent` 的子元素。如果 `row` 为 0，新行将被插入到 `parent` 的所有数据之前，否则将在指定位置的数据之前。如果 `parent` 没有子元素，则会新插入一个单列数据。函数插入成功返回 `true`，否则返回 `false`。我们在这段代码中调用的是 `insertRows(row, 1)`。这是 `QStringListModel` 的一个重载。参数 1 说明要插入 1 条数据。记得之前我们已经把 `row` 设置为当前行，因此，这行语句实际上是在当前的 `row` 位置插入 `count` 行，这里的 `count` 为 1。由于我们没有添加任何数据，实际效果是，我们在 `row` 位置插入了 1 个空行。然后我们使用 `model` 的 `index()` 函数获取当前行的 `QModelIndex` 对象，利用 `setData()` 函数把我们用 `QInputDialog` 接受的数据设置为当前行数据。接下来，我们使用 `setCurrentIndex()` 函数，把当前行设为新插入的一行，并调用 `edit()` 函数，使这一行可以被编辑。

以上是我们提供的一种插入数据的方法：首先插入空行，然后选中新插入的空行，设置新的数据。这其实是一种冗余操作，因为 `currentIndex()` 已经获取到当前行。在此，我们仅仅是为了介绍这些函数的使用。因此，除去这些冗余，我们可以使用一种更简洁的写法：

```
void MyListView::insertData()
{
    bool isOK;
    QString text = QInputDialog::getText(this, "Insert",
                                         "Please input new data:",
                                         QLineEdit::Normal,
                                         "You are inserting new data.",
                                         &isOK);

    if (isOK) {
        QModelIndex currIndex = listView->currentIndex();
        model->insertRows(currIndex.row(), 1);
        model->setData(currIndex, text);
        listView->edit(currIndex);
    }
}
```

接下来是删除数据：

```
void MyListView::deleteData()
{
    if (model->rowCount() > 1) {
        model->removeRows(listView->currentIndex().row(), 1);
    }
}
```

使用模型的 `removeRows()` 函数可以轻松完成这个操作。这个函数同前面所说的 `insertRows()` 很类似，这里不再赘述。需要注意的是，我们用 `rowCount()` 函数判断了一下，要求最终始终保留 1 行。这是因为我们写的简单地插入操作所限制，如果把数据全部删除，就不能再插入数据了。所以，前面所说的插入操作实际上还需要再详细考虑才可以解决这个问题。

最后是简单地将所有数据都显示出来：

```
void MyListView::showData()
{
    QStringList data = model->stringList();
    QString str;
    foreach(QString s, data) {
        str += s + "\n";
    }
    QMessageBox::information(this, "Data", str);
}
```

这段代码没什么好说的。

关于 `QStringListModel` 我们简单介绍这些。从这些示例中可以看到，几乎所有操作都是针对模型的，也就是说，我们直接对数据进行操作，当模型检测到数据发生了变化，会立刻通知视图进行刷新。这样，我们就可以把精力集中到对数据的操作上，而不用担心视图的同步显示问题。这正是 `model/view` 模型所带来的一个便捷之处。

## 44 QFileSystemModel

上一章我们详细了解了 `QStringListModel`。本章我们将再来介绍另外一个内置模型：`QFileSystemModel`。看起来，`QFileSystemModel` 比 `QStringListModel` 要复杂得多；事实也是如此。但是，虽然功能强大，`QFileSystemModel` 的使用还是简单的。

让我们从 Qt 内置的模型说起。实际上，Qt 内置了两种模型：`QStandardItemModel` 和 `QFileSystemModel`。`QStandardItemModel` 是一种多用途的模型，能够让列表、表格、树等视图显示不同的数据结构。这种模型会将数据保存起来。试想一下，列表和表格所要求的数据结构肯定是不一样的：前者是一维的，后者是二维的。因此，模型需要保存有实际数据，当视图是列表时，以一维的形式提供数据；当视图是表格时，以二维的形式提供数据。`QFileSystemModel` 则是另外一种方式。它的作用是维护一个目录的信息。因此，它不需要保存数据本身，而是保存这些在本地文件系统中的实际数据的一个索引。我们可以利用 `QFileSystemModel` 显示文件系统的信息、甚至通过模型来修改文件系统。

QTreeView 是最适合应用 QFileSystemModel 的视图。下面我们看一段代码：

```
FileSystemWidget::FileSystemWidget(QWidget *parent) :
    QWidget(parent)
{
    model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());

    treeView = new QTreeView(this);
    treeView->setModel(model);
    treeView->setRootIndex(model->index(QDir::currentPath()));

    QPushButton *mkdirButton = new QPushButton(tr("Make Directory..."), this);
    QPushButton *rmButton = new QPushButton(tr("Remove"), this);
    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addWidget(mkdirButton);
    buttonLayout->addWidget(rmButton);

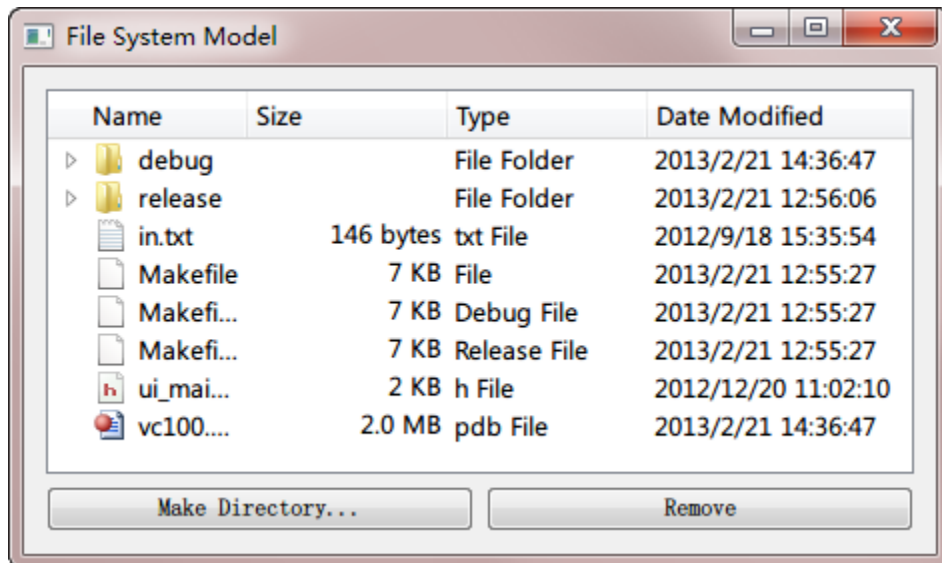
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(treeView);
    layout->addLayout(buttonLayout);

    setLayout(layout);
    setWindowTitle("File System Model");

    connect(mkdirButton, SIGNAL(clicked()),
            this, SLOT(mkdir()));
    connect(rmButton, SIGNAL(clicked()),
            this, SLOT(rm()));
}
```

构造函数很简单，我们首先创建了 QFileSystemModel 实例，然后将其作为一个 QTreeView 的模型。注意我们将 QFileSystemModel 的根目录路径设置为当前目录。剩下的都很简单，我们添加了按钮之类，这些都不再赘述。对于 treeView 视图，我们使用了 setRootIndex() 对模型进行过滤。我们可以尝试一下，去掉这一句的话，我们的程序会显示整个文件系统的目录；而这一句的作用是，从模型中找到 QDir::currentPath() 所对应的索引，然后显示这一位置。也就是说，这一语句的作用实际是设置显示哪个目录。我们会在后面的章节中详细讨论 index() 函数。

现在，我们可以运行以下程序看看界面：



虽然我们基本一行代码都没写(有关文件系统的代码都没有写),但是从运行截图可以看出, `QFileSystemModel` 完全将所能想到的东西——名称、大小、类型、修改时间等全部显示出来,可见其强大之处。

下面是 `mkdir()`槽函数:

```
void FileSystemWidget::mkdir()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid()) {
        return;
    }
    QString dirName = QInputDialog::getText(this,
                                             tr("Create Directory"),
                                             tr("Directory name"));

    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid()) {
            QMessageBox::information(this,
                                     tr("Create Directory"),
                                     tr("Failed to create the directory"));
        }
    }
}
```

正如代码所示, 首先我们获取选择的目录。后面这个 `isValid()`判断很重要, 因为默认情况下是没有目录被选择的,此时路径是非法的,为了避免程序出现异常,必须要有这一步判断。然后弹出对话框询问新的文件夹名字, 如果创建失败会有提示, 否则就是创建成功。这时候你会发现, 硬盘的实际位置的确创建了新的文件夹。



下面则是 `rm()`槽函数:

```
void FileSystemWidget::rm()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid()) {
        return;
    }
    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok) {
        QMessageBox::information(this,
                                   tr("Remove"),
                                   tr("Failed to remove %1").arg(model->fileName(index)));
    }
}
```

这里同样需要先检测路径是否合法。另外需要注意的是，目录和文件的删除不是一个函数，需要调用 `isDir()`函数检测。这一步在代码中有很清楚的描述，这里就不再赘述了。

实际上，我们这里不需要十分担心 `QFileSystemModel` 的性能问题，因为它会启动自己的线程进行文件夹扫描，不会发生因扫描文件夹而导致的主线程阻塞的现象。另外需要注意的是，`QFileSystemModel` 会对模型的结果进行缓存，如果你要立即刷新结果，需要通知 `QFileSystemWatcher` 类。

如果仔细查看就会发现，我们的视图不能排序不能点击列头。为此，我们可以使用下面代码：

```
treeView->header()->setStretchLastSection(true);
treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
treeView->header()->setSortIndicatorShown(true);
#if QT_VERSION >= 0x050000
    treeView->header()->setSectionsClickable(true);
#else
    treeView->header()->setClickable(true);
#endif
```

这是 Qt 中视图类常用的一种技术：如果我们要修改有关列头、行头之类的位置，我们需要从视图类获取到列头对象，然后对其进行设置。正如代码中所显示的那样。注意上面代码片段的最后一部分，我们使用一个条件判断来确定 Qt4 与 Qt5 的不同。

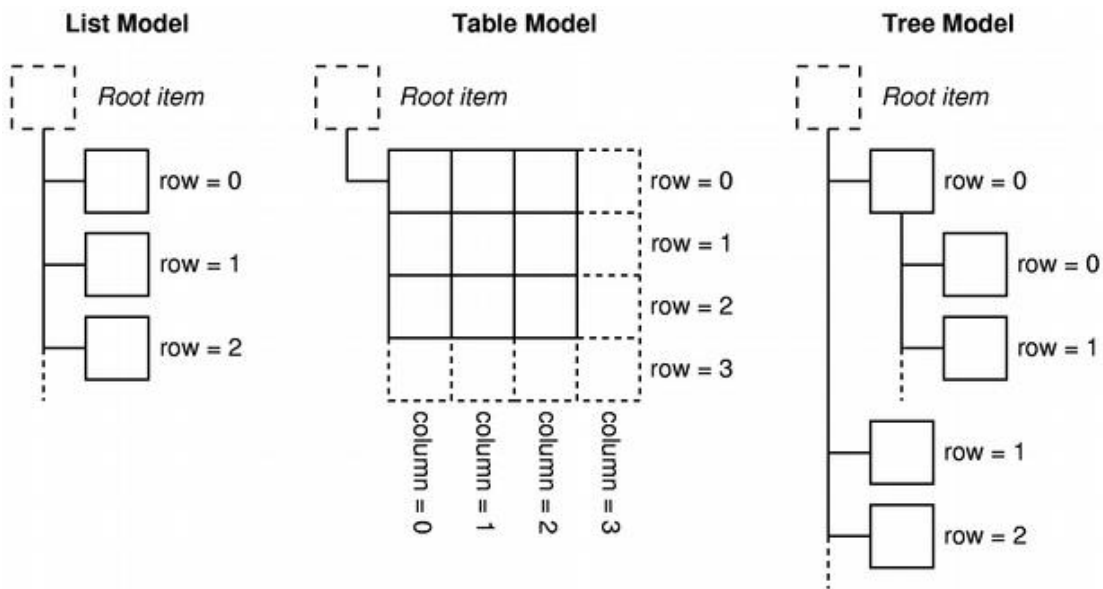
现在我们简单介绍了有关两个常用的模型类：QStringListModel 和 QFileSystemModel。下一章，我们将在此基础上详细介绍模型的相关细节。

## 45 模型

在前面两章的基础之上，我们将开始介绍 model 的通用概念。

在 model/view 架构中，model 提供一种标准接口，供视图和委托访问数据。在 Qt 中，这个接口由 QAbstractItemModel 类进行定义。不管底层数据是如何存储的，只要是 QAbstractItemModel 的子类，都提供一种表格形式的层次结构。视图利用统一的转换来访问模型中的数据。但是，需要提供的是，尽管模型内部是这样组织数据的，但是并不要求也得这样子向用户展示数据。

下面是各种 model 的组织示意图。我们利用此图来理解什么叫“一种表格形式的层次结构”。



如上图所示，List Model 虽然是线性的列表，也有一个 Root Item（根节点），之下才是呈线性的一个个数据，而这些数据实际可以看作是一个只有一列的表格，但是它是有层次的，因为有一个根节点。Table Model 就比较容易理解，只是也存在一个根节点。Tree Model 主要面向层次数据，而每一层次都可以都很多列，因此也是一个带有层次的表格。

为了能够使得数据的显示同存储分离，我们引入模型索引（model index）的概念。通过索引，我们可以访问模型的特定元素的特定部分。视图和委托使用索引来请求所需要的数据。由此

可以看出，只有模型自己需要知道如何获得数据，模型所管理的数据类型可以使用通用的方式进行定义。索引保存有创建的它的那个模型的指针，这使得同时操作多个模型成为可能。

```
QAbstractItemModel *model = index.model();
```

模型索引提供了所需要的信息的**临时索引**，可以用于通过模型取回或者修改数据。由于模型随时可能重新组织其内部的结构，因此模型索引很可能变成不可用的，此时，就不应该保存这些数据。如果你需要长期有效的数据片段，必须创建**持久索引**。持久索引保证其引用的数据及时更新。临时索引（也就是通常使用的索引）由 `QModelIndex` 类提供，持久索引则是 `QPersistentModelIndex` 类。

为了定位模型中的数据，我们需要三个属性：行号、列号以及父索引。下面我们对其一一进行解释。

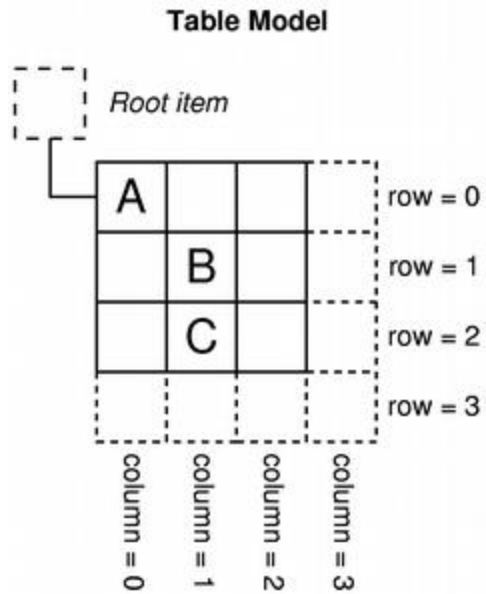
我们前面介绍过模型的基本形式：数据以二维表的形式进行存储。此时，一个数据可以由行号和列号进行定位。注意，我们仅仅是使用“二维表”这个名词，并不意味着模型内部真的是以二维数组的形式进行存储；所谓“行号”“列号”，也仅仅是为方便描述这种对应关系，并不真的是有行列之分。通过指定行号和列号，我们可以定位一个元素项，取出其信息。此时，我们获得的是一个索引对象（回忆一下，通过索引我们可以获取具体信息）：

```
QModelIndex index = model->index(row, column, ...);
```

模型提供了一个简单的接口，用于列表以及表格这种非层次视图的数据获取。不过，正如上面的代码暗示的那样，实际接口并不是那么简单。我们可以通过文档查看这个函数的原型：

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

这里，我们仅仅使用了前两个参数。通过下图来理解一下：



在一个简单的表格中，每一个项都可以由行号和列号确定。因此，我们只需提供两个参数即可获取到表格中的某一个数据项：

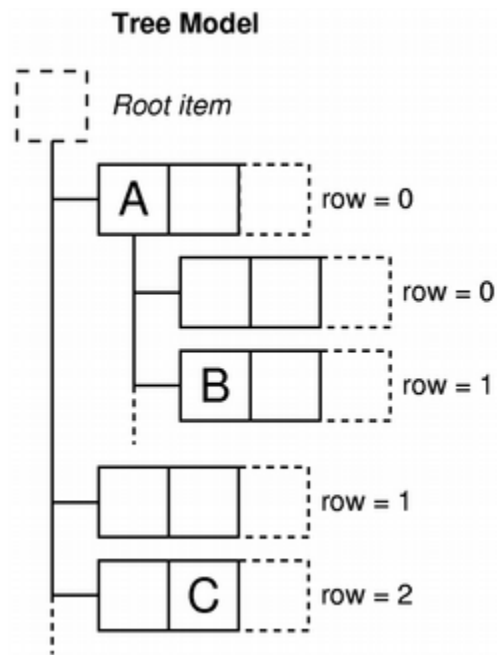
```
QModelIndex indexA = model->index(0, 0, QModelIndex());  
QModelIndex indexB = model->index(1, 1, QModelIndex());  
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

函数的最后一个参数始终是 `QModelIndex()`，接下来我们就要讨论这个参数的含义。

在类似表格的视图中，比如列表和表格，行号和列号足以定位一个数据项。但是，对于树型结构，仅有两个参数就不足够了。这是因为树型结构是一个层次结构，而层次结构中每一个节点都有可能是另外一个表格。所以，每一个项需要指明其父节点。前面说过，在模型外部只能用索引访问内部数据，因此，`index()`函数还需要一个 `parent` 参数：

```
QModelIndex index = model->index(row, column, parent);
```

类似的，我们来看看下面的示意图：



图中，A 和 C 都是模型中的顶级项：

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

A 还有自己的子项。那么，我们就应该使用下面的代码获取 B 的索引：

```
QModelIndex indexB = model->index(1, 0, indexA);
```

由此我们看到，如果只有行号和列号两个参数，B 的行号是 1，列号是 0，这同与 A 同级的行号是 1，列号是 0 的项相同，所以我们通过 `parent` 属性区别开来。

以上我们讨论了有关索引的定位。现在来看看模型的另外一个部分：数据角色。模型可以针对不同的组件（或者组件的不同部分，比如按钮的提示以及显示的文本等）提供不同的数据。例如，`Qt::DisplayRole` 用于视图的文本显示。通常来说，数据项包含一系列不同的数据角色，这些角色定义在 `Qt::ItemDataRole` 枚举中。

我们可以通过指定索引以及角色来获得模型所提供的的数据：

```
QVariant value = model->data(index, role);
```

通过为每一个角色提供恰当的数据，模型可以告诉视图和委托如何向用户显示内容。不同类型的视图可以选择忽略自己不需要的数据。当然，我们也可以添加我们所需要的额外数据。

总结一下：

- 模型使用索引来提供给视图和委托有关数据项的位置的信息，这样做的好处是，模型之外的对象无需知道底层的数据存储方式；
- 数据项通过行号、列号以及父项三个坐标进行定位；
- 模型索引由模型在其它组件（视图和委托）请求时才会被创建；
- 如果使用 `index()` 函数请求获得一个父项的可用索引，该索引会指向模型中这个父项下面的数据项。这个索引指向该项的一个子项；如果使用 `index()` 函数请求获得一个父项的不可用索引，该索引指向模型的最顶级项；
- 角色用于区分数据项的不同类型的数据。

下面回到前面我们曾经见过的模型 `QFileSystemModel`，看看如何从模型获取数据。

```
QFileSystemModel *model = new QFileSystemModel;
QModelIndex parentIndex = model->index(QDir::currentPath());
int numRows = model->rowCount(parentIndex);
```

在这个例子中，我们创建了 `QFileSystemModel` 的实例，使用 `QFileSystemModel` 重载的 `index()` 获取索引，然后使用 `rowCount()` 函数计算当前目录下有多少数据项（也就是行数）。前面一章中迷迷糊糊的代码，现在已经相当清楚了。

为简单起见，下面我们只关心模型第一列。我们遍历所有数据，取得第一列索引：

```
for (int row = 0; row < numRows; ++row) {
    QModelIndex index = model->index(row, 0, parentIndex);
```

我们使用 `index()` 函数，第一个参数是每一行行号，第二个参数是 `0`，也就是第一列，第三个参数是 `parentIndex`，也就是当前目录作为父项。我们可以使用模型的 `data()` 函数获取每一项的数据。注意，该函数返回值是 `QVariant`，实际是一个字符串，因此我们直接转换成 `QString`：

```
QString text = model->data(index, Qt::DisplayRole).toString();
// 使用 text 数据
}
```

上面的代码片段显示了从模型获取数据的一些有用的函数：

- 模型的数目信息可以通过 `rowCount()` 和 `columnCount()` 获得。这些函数需要制定父项；

- 索引用于访问模型中的数据。我们需要利用行号、列号以及父项三个参数来获得该索引；
- 当我们使用 `QModelIndex()` 创建一个空索引使用时，我们获得的就是模型中最顶级项；
- 数据项包含了不同角色的数据。为获取特定角色的数据，必须指定这个角色。

## 46 视图和委托

前面我们介绍了模型的概念。下面则是另外一个基本元素：视图。在 `model/view` 架构中，视图是数据从模型到最终用户的途径。数据通过视图向用户进行显示。此时，这种显示方式不必须同模型的存储结构相一致。实际上，很多情况下，数据的显示同底层数据的存储是完全不同的。

我们使用 `QAbstractItemModel` 提供标准的模型接口，使用 `QAbstractItemView` 提供标准的视图接口，而结合这两者，就可以将数据同表现层分离，在视图中利用前面所说的模型索引。视图管理来自模型的数据的布局：既可以直接渲染数据本身，也可以通过委托渲染和编辑数据。

视图不仅仅用于展示数据，还用于在数据项之间的导航以及数据项的选择。另外，视图也需要支持很多基本的用户界面的特性，例如右键菜单以及拖放。视图可以提供数据编辑功能，也可以将这种编辑功能交由某个委托完成。视图可以脱离模型创建，但是在其进行显示之前，必须存在一个模型。也就是说，视图的显示是完全基于模型的，这是不能脱离模型存在的。对于用户的选择，多个视图可以相互独立，也可以进行共享。

某些视图，例如 `QTableView` 和 `QTreeView`，不仅显示数据，还会显示列头或者表头。这些是由 `QHeaderView` 视图类提供的。在《[QFileSystemModel](#)》一章的最后，我们曾经提到过这个问题。表头通常访问视图所包含的同一模型。它们使用 `QAbstractItemModel::headerData()` 函数从模型中获取数据，然后将其以标签 `label` 的形式显示出来。我们可以通过继承 `QHeaderView` 类，实现某些更特殊的功能。

正如前面的章节介绍的，我们通常会为视图提供一个模型。拿前面我们曾经见过的一个例子来看：

```
QStringList data;
data << "0" << "1" << "2";
model = new QStringListModel(this);
model->setStringList(data);
```





```

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;
};

```

正如前面所说，委托就是供视图实现某种高级的编辑功能。不同于经典的 Model-View-Controller (MVC) 模式，model/view 没有将用户交互部分完全分离。一般地，视图将数据向用户进行展示并且处理通用的输入。但是，对于某些特殊要求（比如这里的要求必须输入数字），则交予委托完成。这些组件提供输入功能，同时也能渲染某些特殊数据项。委托的接口由 `QAbstractItemDelegate` 定义。在这个类中，委托通过 `paint()` 和 `sizeHint()` 两个函数渲染用户内容（也就是说，你必须自己将渲染器绘制出来）。为使用方便，从 4.4 开始，Qt 提供了另外的基于组件的子类：`QItemDelegate` 和 `QStyledItemDelegate`。默认的委托是 `QStyledItemDelegate`。二者的区别在于绘制和向视图提供编辑器的方式。`QStyledItemDelegate` 使用当前样式绘制，并且能够使用 Qt Style Sheet（我们会在后面的章节对 QSS 进行介绍），因此我们推荐在自定义委托时，使用 `QStyledItemDelegate` 作为基类。不过，除非自定义委托需要自己进行绘制，否则，二者的代码其实是一样的。

继承 `QStyledItemDelegate` 需要实现以下几个函数：

- `createEditor()`：返回一个组件。该组件会被作为用户编辑数据时所使用的编辑器，从模型中接受数据，返回用户修改的数据。
- `setEditorData()`：提供上述组件在显示时所需要的默认值。
- `updateEditorGeometry()`：确保上述组件作为编辑器时能够完整地显示出来。
- `setModelData()`：返回给模型用户修改过的数据。

下面依次看看各函数的实现：

```

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;
}
```

在 `createEditor()` 函数中，`parent` 参数会作为新的编辑器的父组件。

```
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);
}
```

`setEditorData()` 函数从模型中获取需要编辑的数据（具有 `Qt::EditRole` 角色）。由于我们知道它就是一个整型，因此可以放心地调用 `toInt()` 函数。`editor` 就是所生成的编辑器实例，我们将其强制转换成 `QSpinBox` 实例，设置其数据作为默认值。

```
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);
}
```

在用户编辑完数据后，委托会调用 `setModelData()` 函数将新的数据保存到模型中。因此，在这里我们首先获取 `QSpinBox` 实例，得到用户输入值，然后设置到模型相应的位置。标准的 `QStyledItemDelegate` 类会在完成编辑时发出 `closeEditor()` 信号，视图会保证编辑器已经关闭并且销毁，因此无需对内存进行管理。由于我们的处理很简单，无需在发出 `closeEditor()` 信号，但是在复杂的实现中，记得可以在这里发出这个信号。针对数据的任何操作都必须提交给 `QAbstractItemModel`，这使得委托独立于特定的视图。当然，在真实应用中，我们需要检测用户的输入是否合法，是否能够存入模型。

```
void SpinBoxDelegate::updateEditorGeometry(QWidget *editor,
                                           const QStyleOptionViewItem &option,
                                           const QModelIndex &index) const
{
    editor->setGeometry(option.rect);
}
```

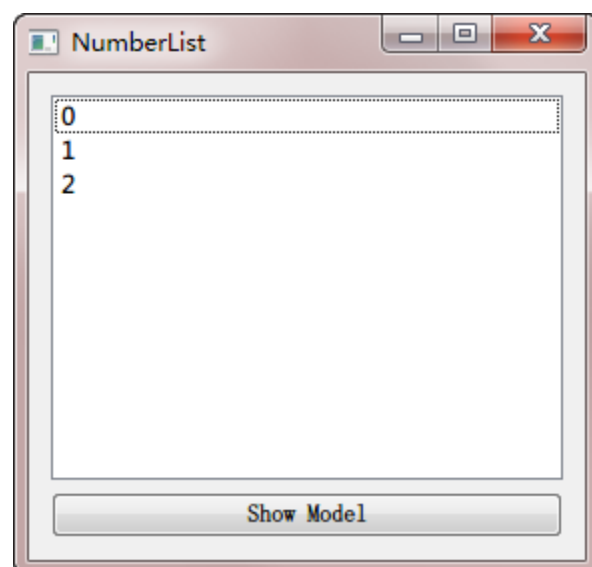
最后，由于我们的编辑器只有一个数字输入框，所以只是简单将这个输入框的大小设置为单元格的大小（由 `option.rect` 提供）。如果是复杂的编辑器，我们需要根据单元格参数（由 `option` 提供）、数据（由 `index` 提供）结合编辑器（由 `editor` 提供）计算编辑器的显示位置和大小。

现在，我们的委托已经编写完毕。接下来需要将这个委托设置为 `QListView` 所使用的委托：

```
listView->setItemDelegate(new SpinBoxDelegate(listView));
```

值得注意的是，`new` 操作符并不会真的创建编辑器实例。相反，只有在真正需要时，Qt 才会生成一个编辑器实例。这保证了程序运行时的性能。

然后我们运行下程序：



## 47 视图选择

选择是视图中常用的一个操作。在列表、树或者表格中，通过鼠标点击可以选中某一项，被选中项会变成高亮或者反色。在 Qt 中，选择也是使用了一种模型。在 `model/view` 架构中，这种选择模型提供了一种更通用的对选择操作的描述。对于一般应用而言，Qt 内置的选择模型已经足够，但是，Qt 还是允许你创建自己的选择模型，来实现一些特殊的操作。

Qt 使用 `QItemSelectionModel` 类获取视图中项目的选择情况。这个模型保持有项目的索引，并且独立于任何视图。这意味着，我们可以让不同的视图共享同一个选择模型，从而达

到一种同步操作的目的。选择由选择区域组成。模型只将选区的开始和结束的索引位置记录下来，以保证对于很大的选区也有很好的性能。非连续选区则由多个连续选择组成。

选择会直接应用于选择模型所维护的那些被选中的索引上面。最新的选择就是当前选择。这意味着，即便界面上没有显示有任何项目被选择，如果通过某些命令对选区进行操作，同样会有作用。

在视图中，始终存在一个当前项和被选择项（即便从界面上看不到有任何选择）。与通常所想的不同，当前项和选择项是相互独立的两个状态。一个项目可以即是当前项又是选择项。下表是当前项和选择项的区别：

当前项	选择项
只能有一个当前项。	可以有多个选择项。
使用键盘或者鼠标点击可以改变当前项。	选择项使用两种状态：选择和未选择，这取决于项目之前的状态和其它一些设置，例如，单选或多选。只有在用户进行交互的时候，这种状态才会发生改变。
当前项可以使用 F2 或者鼠标双击进行编辑（前提是程序允许）。	当前项可以结合另外一个锚点指定被选择或者去除选择的一块选区（或二者的结合）。
当前项通常会有一个焦点框进行标识。	选择项使用选区颜色进行标识。

在处理选择的时候，我们可以将 `QItemSelectionModel` 当成数据模型中所有数据项的选择状态的一个记录。一旦选择模型创建好，这些数据项就可以在不知道哪些项被选择的情况下进行选择、取消选择或者改变选择状态的操作。所有被选择项的索引都在可随时更改，其它组件也可以通过信号槽机制修改这些选择的信息。

标准视图类（`QListView`、`QTreeView` 以及 `QTableView`）已经提供了默认的选择模型，足以满足大多数应用程序的需求。某一个视图的选择模型可以通过 `selectionModel()` 函数获取，然后使用 `setSelectionModel()` 提供给其它视图共享，因此，一般没有必要新建选择模型。

如果需要创建一个选区，我们需要指定一个模型以及一对索引，使用这些数据创建一个 `QItemSelection` 对象。这两个索引应该指向给定的模型中的数据，并且作为一个块状选区的左上角和右下角的索引。为了将选区应用到模型上，需要将选区提交到选择模型。这种操作有多种实现，对于现有选择模型有着不同的影响。

下面我们来看一些代码片段。首先构建一个总数 32 个数据项的表格模型，然后将其设置为一个表格视图的数据：

```
QTableWidget tableWidget(8, 4);

QItemSelectionModel *selectionModel = tableWidget.selectionModel();
```

在代码的最后，我们获得 `QTableView` 的选择模型，以备以后使用。现在，我们没有修改模型中的数据，而是选择表格左上角的一些单元格。下面我们来看看代码如何实现：

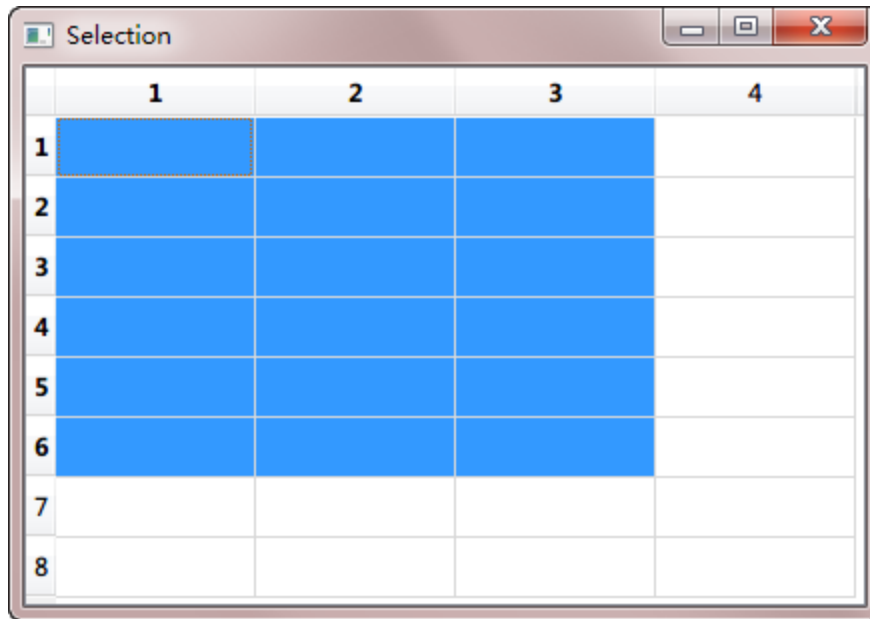
```
QModelIndex topLeft = tableWidget.model()->index(0, 0, QModelIndex());
QModelIndex bottomRight = tableWidget.model()->index(5, 2, QModelIndex());
```

接下来，我们将获得的两个索引定义为选区。为达这一目的，我们首先构造一个 `QItemSelection` 对象，然后将其赋值给我们获取的选择模型：

```
QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

正如前面我们说的，首先利用左上角和右下角的坐标构建一个 `QItemSelection` 对象，然后将这个对象设置为选择模型的选择区。`select()`函数的第一个参数就是需要选择的选区，第二个参数是选区的标志位。Qt 提供了很多不同的操作，可以参考下 `QItemSelectionModel::SelectionFlags` 的文档。在本例中，我们使用了 `QItemSelectionModel::Select`，这意味着选区中所包含的所有单元格都会被选择。

下面就是我们的运行结果：



现在我们知道如何设置选区。下面来看看如何获取选区。获取选区需要使用 `selectedIndexes()` 函数。该函数返回一个无序列表。我们可以通过遍历这个列表获得哪些被选择：

```
QModelIndexList indexes = selectionModel->selectedIndexes();
QModelIndex index;

foreach(index, indexes) {
    QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
    model->setData(index, text);
}
```

在选择发生更改时，选择模型会发出信号。我们可以连接 `selectionChanged()` 信号，在选区改变时检查哪个项目发生了变化。这个信号有两个参数：第一个是新选择的项目，第二个是刚刚被取消选择的项目。在下面的示例中，我们通过 `selectionChanged()` 信号，将所有新选择的项目填充字符串，将所有被取消选择的部分清空：

```
void MainWindow::updateSelection(const QItemSelection &selected,
                                const QItemSelection &deselected)
{
    QModelIndex index;
    QModelIndexList items = selected.indexes();

    foreach (index, items) {
        QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
        model->setData(index, text);
    }
}
```

```

        items = deselected.indexes();

        foreach (index, items) {
            model->setData(index, "");
        }
    }
}

```

通过 `currentChanged()`，我们可以追踪当前有焦点的项。同 `selectionChanged()` 信号类似，这个信号也有两个参数：第一个是新的当前项，第二个是上一个当前项。下面的代码则是该信号的使用：

```

void MainWindow::changeCurrent(const QModelIndex &current,
                               const QModelIndex &previous)
{
    statusBar()->showMessage(
        tr("Moved from (%1,%2) to (%3,%4)")
        .arg(previous.row()).arg(previous.column())
        .arg(current.row()).arg(current.column()));
}

```

这些信号可以用来监控选区的改变。如果你还要直接更新选区，我们还有另外的方法。

同样是利用前面所说的 `QItemSelectionModel::SelectionFlag`，我们可以对选区进行组合操作。还记得我们在前面的 `select()` 函数中使用过的第二个参数吗？当我们替换这个参数，就可以获得不同的组合方式。最常用的就是 `QItemSelectionModel::Select`，它的作用是将所有指定的选区都选择上。`QItemSelectionModel::Toggle` 则是一种取反的操作：如果指定的部分原来已经被选择，则取消选择，否则则选择上。`QItemSelectionModel::Deselect` 则是取消指定的已选择的部分。在下面的例子中，我们使用 `QItemSelectionModel::Toggle` 对前面的示例作进一步的操作：

```

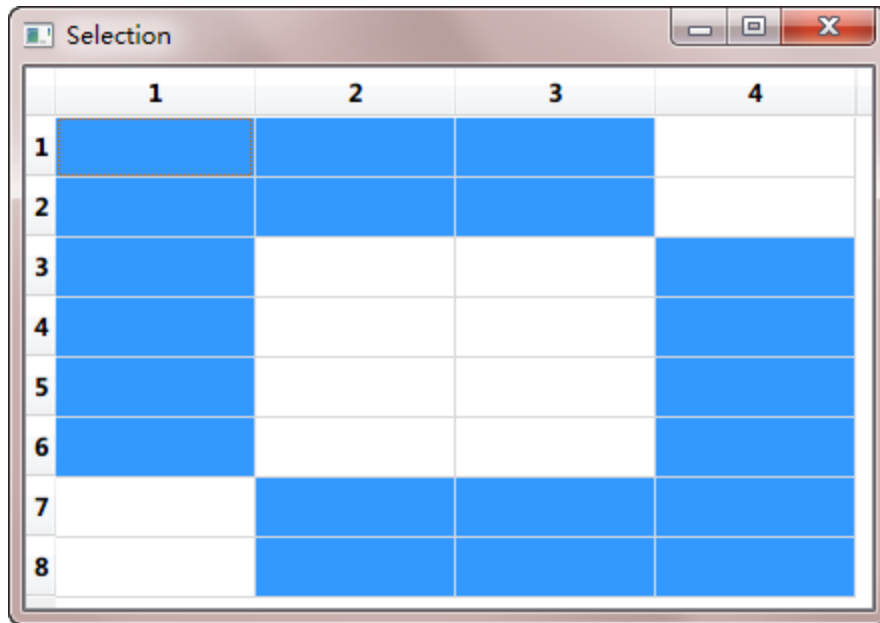
QItemSelection toggleSelection;

topLeft = tableWidget.model()->index(2, 1, QModelIndex());
bottomRight = tableWidget.model()->index(7, 3, QModelIndex());
toggleSelection.select(topLeft, bottomRight);

selectionModel->select(toggleSelection, QItemSelectionModel::Toggle);

```

运行结果将如下所示：



默认情况下，选择操作只会影响到指定的模型索引。但是，我们也可以改变这一设置。例如，只选择整行或者整列：

```
QItemSelection columnSelection;

topLeft = model->index(0, 1, QModelIndex());
bottomRight = model->index(0, 2, QModelIndex());

columnSelection.select(topLeft, bottomRight);

selectionModel->select(columnSelection,
                      QItemSelectionModel::Select |
                      QItemSelectionModel::Columns);

QItemSelection rowSelection;

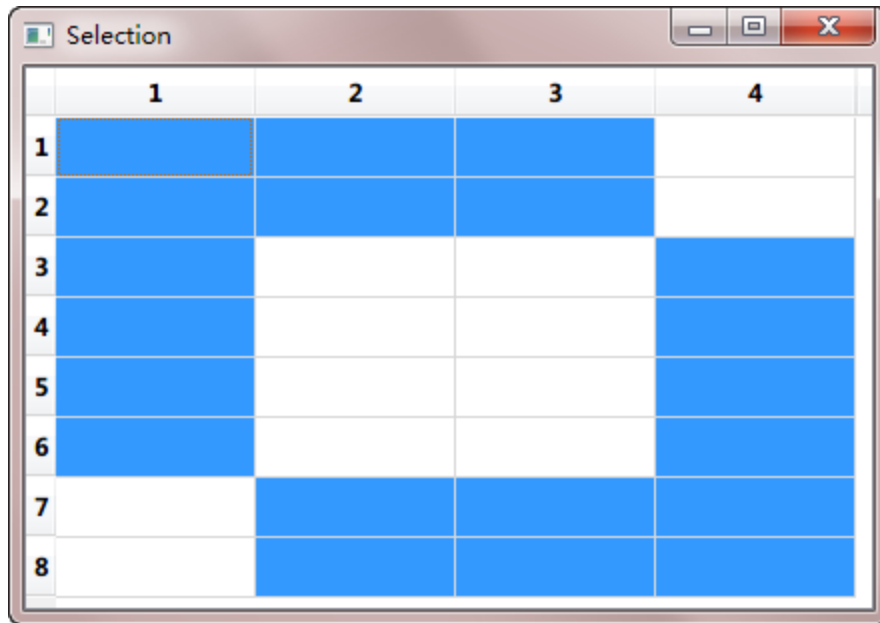
topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(1, 0, QModelIndex());

rowSelection.select(topLeft, bottomRight);

selectionModel->select(rowSelection,
                      QItemSelectionModel::Select | QItemSelectionModel::Rows);
```

上面的代码，我们依然使用两个索引设置了一个区域，但是，在选择的使用我们使用了 `QItemSelectionModel::Rows` 和 `QItemSelectionModel::Columns` 这两个参数，因此只会选择这两个区域中指定的行或者列：





使用 `QItemSelectionModel::Current` 参数可以将当前选区替换为新的选区；使用 `QItemSelectionModel::Clear` 则会将原来已有的选区全部取消。为了进行全选，我们可以设置选区为左上角和右下角两个索引：

```
QModelIndex topLeft = model->index(0, 0, parent);
QModelIndex bottomRight = model->index(model->rowCount(parent)-1,
model->columnCount(parent)-1, parent);

QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

## 48 QSortFilterProxyModel

从本章开始，我们将逐步了解有关自定义模型的相关内容。尽管前面我们曾经介绍过 Qt 提供的几个内置模型：`QStringListModel` 和 `QFileSystemModel`，但对于千变万化的需求而言，这些显然是远远不够的。于是，Qt 也允许我们对模型进行自定义。

在正式开始介绍自定义模型之前，我们先来了解一个新的类：`QSortFilterProxyModel`。之所以将这个类放在这里，是因为在一定程序上，我们可以使用 `QSortFilterProxyModel` 获得一些可能必须自定义才能达到的效果。`QSortFilterProxyModel` 并不能单独使用。顾名思义，它是一个“代理”，其真正的数据需要另外的一个模型提供。它的作用是对数据进行排序和过滤。排序很好理解，而过滤，则是按照输入的内容对数据进行筛选，很像 Excel 里面的过滤器。不过 Qt 提供的过滤功能是基于正则表达式的，功能很强大。

下面我们根据代码来了解下 `QSortFilterProxyModel` 的使用：

```

class SortView : public QWidget
{
    Q_OBJECT
public:
    SortView();

private:
    QListView *view;
    QStringListModel *model;
    QSortFilterProxyModel *modelProxy;
    QComboBox *syntaxBox;

private slots:
    void filterChanged(const QString &text);
};

```

头文件中，我们声明了一个类 `SortView`，继承自 `QWidget`。它有四个成员变量以及一个私有槽函数。

```

SortView::SortView()
{
    model = new QStringListModel(QColor::colorNames(), this);

    modelProxy = new QSortFilterProxyModel(this);
    modelProxy->setSourceModel(model);
    modelProxy->setFilterKeyColumn(0);

    view = new QListView(this);
    view->setModel(modelProxy);

    QLineEdit *filterInput = new QLineEdit;
    QLabel *filterLabel = new QLabel(tr("Filter"));
    QHBoxLayout *filterLayout = new QHBoxLayout;
    filterLayout->addWidget(filterLabel);
    filterLayout->addWidget(filterInput);

    syntaxBox = new QComboBox;
    syntaxBox->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Preferred);
    syntaxBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    QLabel *syntaxLabel = new QLabel(tr("Syntax"));
    QHBoxLayout *syntaxLayout = new QHBoxLayout;
    syntaxLayout->addWidget(syntaxLabel);
}

```

```

syntaxLayout->addWidget(syntaxBox);

QVBoxLayout *layout = new QVBoxLayout(this);
layout->addWidget(view);
layout->addLayout(filterLayout);
layout->addLayout(syntaxLayout);

connect(filterInput, SIGNAL(textChanged(QString)),
        this, SLOT(filterChanged(QString)));
}

```

在构造函数中，我们首先创建一个 `QStringListModel` 对象，其内容是 Qt 预定义的所有颜色的名字（利用 `QColor::colorNames()` 获取）。然后是 `QSortFilterProxyModel` 对象，我们将其原模型设置为刚刚创建的 `model`，也就是要为此 `model` 进行代理；然后将 `FilterKeyColumn` 设置为 0，也就是仅仅对第一列进行过滤。我们使用一个 `QStringListModel` 包装这个数据，这和前面的内容没有什么区别。然后创建一个 `QSortFilterProxyModel` 对象，使用它的 `setSourceModel()` 函数将前面定义的 `QStringListModel` 传进去，也就是我们需要对这个 `model` 进行代理。最后重要的一点是，`QListView` 的数据源必须设置为 `QSortFilterProxyModel`，而不是最开始的 `model` 对象。

作为过滤选项，`syntaxBox` 添加了三个数据项：

```

syntaxBox->addItem(tr("Regular expression"), QRegExp::RegExp);
syntaxBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
syntaxBox->addItem(tr("Fixed string"), QRegExp::FixedString);

```

这正是正则表达式的几种类型。正则表达式自己有一套相对通用的语法，但是对于不同的语言环境（例如 Java、C# 和 Python），其具体定义可能会略有差别。这里我们使用的是 Qt 自己的正则表达式处理工具（C++ 本身并没有解析正则表达式的机制，虽然 boost 提供了一套）。第一个 `QRegExp::RegExp` 提供了最一般的正则表达式语法，但这个语法不支持贪婪限定符。这也是 Qt 默认的规则；如果需要使用贪婪限定符，需要使用 `QRegExp::RegExp2`。尽管在 Qt4 的文档中声明，`QRegExp::RegExp2` 将会作为 Qt5 的默认规则，但其实并不是这样。第二个我们提供的是 Unix shell 常见的一种规则，使用通配符处理。第三个即固定表达式，也就是说基本上不使用正则表达式。

接下来我们看看 `filterChanged()` 函数的实现：

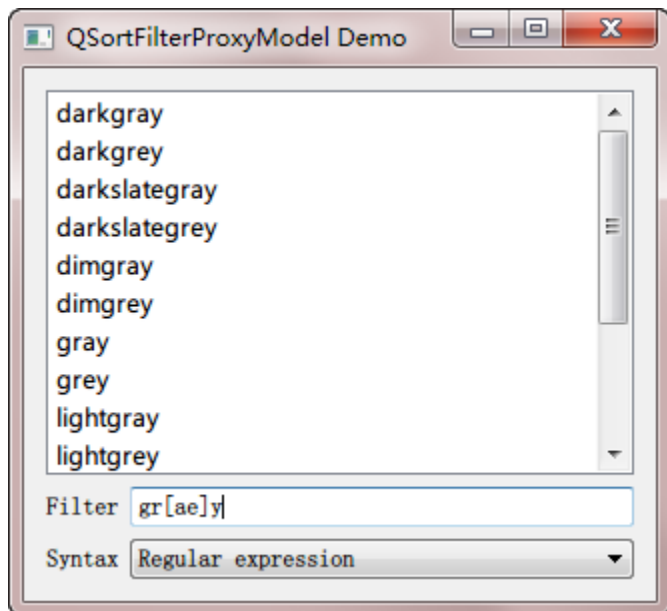
```

void SortView::filterChanged(const QString &text)
{
    QRegExp::PatternSyntax syntax = QRegExp::PatternSyntax(
        syntaxBox->itemData(syntaxBox->currentIndex()).toInt());
    QRegExp regExp(text, Qt::CaseInsensitive, syntax);
}

```

```
modelProxy->setFilterRegExp(regExp);  
}
```

在这段代码中,首先使用 `QComboBox` 的选择值创建一个 `QRegExp::PatternSyntax` 对象;然后利用这个语法规则构造一个正则表达式,注意我们在 `QLineEdit` 里面输入的内容是通过参数传递进来的,然后设置数据模型代理的过滤表达式。下面可以运行一下看看结果:



上图中,我们输入的是 `gr[ae]y` 作为正则表达式。这是说,我们希望获取这样一个颜色的名字:它的名字中有这样的四个字母,第一个字母是 `g`,第二个字母是 `r`,第三个字母要么是 `a`,要么是 `e`,第四个字母是 `y`。如果找到符合条件的名字,就要把它过滤出来,显示到列表中,不符合条件的全部不显示。我们的程序正是这样的结果。如果你对这个正则表达式不熟悉,请自行查阅有关正则表达式的内容。

## 49 自定义只读模型

`model/view` 模型将数据与视图分割开来,也就是说,我们可以为不同的视图, `QListView`、`QTableView` 和 `QTreeView` 提供一个数据模型,这样我们可以从不同角度来展示数据的方方面面。但是,面对变化万千的需求,Qt 预定义的几个模型是远远不能满足需要的。因此,我们还必须自定义模型。

类似 `QAbstractView` 类之于自定义视图, `QAbstractItemModel` 为自定义模型提供了一个足够灵活的接口。它能够支持数据源的层次结构,能够对数据进行增删改操作,还能够支持拖放。不过,有时候一个灵活的类往往显得过于复杂,所以,Qt 又提供了

`QAbstractListModel` 和 `QAbstractTableModel` 两个类来简化非层次数据模型的开发。顾名思义，这两个类更适合于结合列表和表格使用。

本节，我们正式开始对自定义模型进行介绍。

在开始自定义模型之前，我们首先需要思考这样一个问题：我们的数据结构适合于哪种视图的显示方式？是列表，还是表格，还是树？如果我们的数据仅仅用于列表或表格的显示，那么 `QAbstractListModel` 或者 `QAbstractTableModel` 已经足够，它们为我们实现了很多默认函数。但是，如果我们的数据具有层次结构，并且必须向用户显示这种层次，我们只能选择 `QAbstractItemModel`。不管底层数据结构是怎样的格式，最好都要直接考虑适应于标准的 `QAbstractItemModel` 的接口，这样就可以让更多视图能够轻松访问到这个模型。

现在，我们开始自定义一个模型。这个例子修改自《C++ GUI Programming with Qt4, 2nd Edition》。首先描述一下需求。我们想要实现的是一个货币汇率表，就像银行营业厅墙上挂着的那种电子公告牌。当然，你可以选择 `QTableWidget`。的确，直接使用 `QTableWidget` 确实很方便。但是，试想一个包含了 100 种货币的汇率表。显然，这是一个二维表，并且对于每一种货币，都需要给出相对于其他 100 种货币的汇率（我们把自己对自己的汇率也包含在内，只不过这个汇率永远是 1.0000）。现在，按照我们的设计，这张表要有  $100 \times 100 = 10000$  个数据项。我们希望减少存储空间，有没有更好的方式？于是我们想，如果我们的数据不是直接向用户显示的数据，而是这种货币相对于美元的汇率，那么其它货币的汇率都可以根据这个汇率计算出来了。比如，我存储人民币相对美元的汇率，日元相对美元的汇率，那么人民币相对日元的汇率只要作一下比就可以得到了。这种数据结构就没有必要存储 10000 个数据项，只要存储 100 个就够了（实际情况中这可能是不现实的，因为两次运算会带来更大的误差，但这不在我们现在的考虑范畴中）。

于是我们设计了 `CurrencyModel` 类。它底层使用 `QMap<QString, double>` 数据结构进行存储，`QString` 类型的键是货币名字，`double` 类型的值是这种货币相对美元的汇率。（这里提一点，实际应用中，永远不要使用 `double` 处理金额敏感的数据！因为 `double` 是不精确的，不过这一点显然不在我们的考虑中。）

首先从头文件开始看起：

```
class CurrencyModel : public QAbstractTableModel
{
public:
    CurrencyModel(QObject *parent = 0);
    void setCurrencyMap(const QMap<QString, double> &map);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role)
const;
```

```
private:
    QString currencyAt(int offset) const;
    QMap<QString, double> currencyMap;
};
```

这段代码平淡无奇，我们继承了 `QAbstractTableModel` 类，然后重写了所要求的几个函数。构造函数同样如此：

```
CurrencyModel::CurrencyModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

`rowCount()` 和 `columnCount()` 用于返回行和列的数目。记得我们保存的是每种货币相对美元的汇率，而需要显示的是它们两两之间的汇率，因此这两个函数都应该返回这个 `map` 的项数：

```
int CurrencyModel::rowCount(const QModelIndex & parent) const
{
    return currencyMap.count();
}

int CurrencyModel::columnCount(const QModelIndex & parent) const
{
    return currencyMap.count();
}
```

`headerData()` 用于返回列名：

```
QVariant CurrencyModel::headerData(int section, Qt::Orientation, int role) const
{
    if (role != Qt::DisplayRole) {
        return QVariant();
    }
    return currencyAt(section);
}
```

我们在前面的章节中介绍过有关角色的概念。这里我们首先判断这个角色是不是用于显示的，如果是，则调用 `currencyAt()` 函数返回第 `section` 列的名字；如果不是则返回一个空白的 `QVariant` 对象。`currencyAt()` 函数定义如下：

```
QString CurrencyModel::currencyAt(int offset) const
{
```

```
return (currencyMap.begin() + offset).key();
}
```

如果不了解 `QVariant` 类，可以简单认为这个类型相当于 Java 里面的 `Object`，它把 Qt 提供的大部分数据类型封装起来，起到一个类型擦除的作用。比如我们的单元格的数据可以是 `string`，可以是 `int`，也可以是一个颜色值，这么多类型怎么使用一个函数返回呢？回忆一下，返回值并不用于区分一个函数。于是，Qt 提供了 `QVariant` 类型。你可以把很多类型存放进去，到需要使用的时候使用一系列的 `to` 函数取出来即可。比如把 `int` 包装成一个 `QVariant`，使用的时候要用 `QVariant::toInt()` 重新取出来。这非常类似于 `union`，但是 `union` 的问题是，无法保持没有默认构造函数的类型，于是 Qt 提供了 `QVariant` 作为 `union` 的一种模拟。

`setCurrencyMap()` 函数则是用于设置底层的实际数据。由于我们不可能将这种数据硬编码，所以我们必须为模型提供一个用于设置的函数：

```
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    beginResetModel();
    currencyMap = map;
    endResetModel();
}
```

我们当然可以直接设置 `currencyMap`，但是我们依然添加了 `beginResetModel()` 和 `endResetModel()` 两个函数调用。这将告诉关心这个模型的其它类，现在要重置内部数据，大家要做好准备。这是一种契约式的编程方式。

接下来便是最复杂的 `data()` 函数：

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid()) {
        return QVariant();
    }

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());
        if (currencyMap.value(rowCurrency) == 0.0) {
            return "####";
        }
        double amount = currencyMap.value(columnCurrency)
```

```

        / currencyMap.value(rowCurrency);
        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}

```

`data()`函数返回一个单元格的数据。它有两个参数：第一个是 `QModelIndex`，也就是单元格的位置；第二个是 `role`，也就是这个数据的角色。这个函数的返回值是 `QVariant` 类型。我们首先判断传入的 `index` 是不是合法，如果不合法直接返回一个空白的 `QVariant`。然后如果 `role` 是 `Qt::TextAlignmentRole`，也就是文本的对齐方式，返回 `int(Qt::AlignRight | Qt::AlignVCenter)`；如果是 `Qt::DisplayRole`，就按照我们前面所说的逻辑进行计算，然后以字符串的格式返回。这时候你就会发现，其实我们在 `if...else...` 里面返回的不是一种数据类型：`if` 里面返回的是 `int`，而 `else` 里面是 `QString`，这就是 `QVariant` 的作用了。

为了看看实际效果，我们可以使用这样的 `main()`函数代码：

```

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

    QMap<QString, double> data;
    data["USD"] = 1.0000;
    data["CNY"] = 0.1628;
    data["GBP"] = 1.5361;
    data["EUR"] = 1.2992;
    data["HKD"] = 0.1289;

    QTableView view;
    CurrencyModel *model = new CurrencyModel(&view);
    model->setCurrencyMap(data);
    view.setModel(model);
    view.resize(400, 300);
    view.show();

    return a.exec();
}

```

这是我们的实际运行效果：



	CNY	EUR	GBP	HKD	USD
CNY	1.0000	7.9803	9.4355	0.7918	6.1425
EUR	0.1253	1.0000	1.1823	0.0992	0.7697
GBP	0.1060	0.8458	1.0000	0.0839	0.6510
HKD	1.2630	10.0791	11.9170	1.0000	7.7580
USD	0.1628	1.2992	1.5361	0.1289	1.0000

自定义只读模型

## 50 自定义可编辑模型

上一章我们了解了如何自定义只读模型。顾名思义，只读模型只能够用于展示只读数据，用户不能对其进行修改。如果允许用户修改数据，则应该提供可编译的模型。可编辑模型与只读模型非常相似，至少在展示数据方面几乎是完全一样的，所不同的是可编译模型需要提供用户编辑数据后，应当如何将数据保存到实际存储值中。

我们还是利用上一章的 `CurrencyModel`，在此基础上进行修改。相同的代码这里不再赘述，我们只列出增加以及修改的代码。相比只读模型，可编辑模型需要增加以下两个函数的实现：

```
Qt::ItemFlags flags(const QModelIndex &index) const;
bool setData(const QModelIndex &index, const QVariant &value,
             int role = Qt::EditRole);
```

还记得之前我们曾经介绍过，在 Qt 的 model/view 模型中，我们使用委托 delegate 来实现数据的编辑。在实际创建编辑器之前，委托需要检测这个数据项是不是允许编辑。模型必须让委托知道这一点，这是通过返回模型中每个数据项的标记 flag 来实现的，也就是这个 `flags()` 函数。这本例中，只有行和列的索引不一致的时候，我们才允许修改（因为对角线上的值恒为 1.0000，不应该对其进行修改）：

```

Qt::ItemFlags CurrencyModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column()) {
        flags |= Qt::ItemIsEditable;
    }
    return flags;
}

```

注意，我们并不是在判断了 `index.row() != index.column()` 之后直接返回 `Qt::ItemIsEditable`，而是返回 `QAbstractItemModel::flags(index) | Qt::ItemIsEditable`。这是因为我们不希望丢失原来已经存在的那些标记。

我们不需要知道在实际编辑的过程中，委托究竟做了什么，只需要提供一种方式，告诉 Qt 如何将委托获得的用户输入的新的数据保存到模型中。这一步骤是通过 `setData()` 函数实现的：

```

bool CurrencyModel::setData(const QModelIndex &index,
                           const QVariant &value, int role)
{
    if (index.isValid()
        && index.row() != index.column()
        && role == Qt::EditRole) {
        QString columnCurrency = headerData(index.column(),
                                             Qt::Horizontal, Qt::DisplayRole)
                                   .toString();
        QString rowCurrency = headerData(index.row(),
                                           Qt::Vertical, Qt::DisplayRole)
                               .toString();
        currencyMap.insert(columnCurrency,
                           value.toDouble() * currencyMap.value(rowCurrency));
        emit dataChanged(index, index);
        return true;
    }
    return false;
}

```

回忆一下我们的业务逻辑：我们的底层数据结构中保存的是各个币种相对美元的汇率，显示的时候，我们使用列与行的比值获取两两之间的汇率。例如，当我们修改 `currencyMap["CNY"]`/`currencyMap["HKD"]` 的值时，我们认为人民币相对美元的汇率发生了变化，而港币相对美元的汇率保持不变，因此新的数值应当是用户新输入的值与原来 `currencyMap["HKD"]` 的乘积。这正是上面的代码所实现的逻辑。另外注意，在实际修改之前，我们需要检查 `index` 是否有效，以及从业务来说，行列是否不等，最后还要检查角色是不是 `Qt::EditRole`。这里为方便起见，我们使用了 `Qt::EditRole`，也就是编辑时的角

色。但是，对于布尔类型，我们也可以选择使用设置 `Qt::ItemIsUserCheckable` 标记的 `Qt::CheckStateRole`，此时，Qt 会显示一个选择框而不是输入框。注意这里的底层数据都是一样的，只不过显示方式的区别。当数据重新设置是，模型必须通知视图，数据发生了变化。这要求我们必须发出 `dataChanged()` 信号。由于我们只有一个数据发生了改变，因此这个信号的两个参数是一致的（`dataChanged()` 的两个参数是发生改变的数据区域的左上角和右下角的索引值，由于我们只改变了一个单元格，所以二者是相同的）。最后，如果数据修改成功就返回 `true`，否则返回 `false`。

当我们完成以上工作时，还需要修改一下 `data()` 函数：

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid()) {
        return QVariant();
    }

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole || role == Qt::EditRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());
        if (currencyMap.value(rowCurrency) == 0.0) {
            return "####";
        }
        double amount = currencyMap.value(columnCurrency)
            / currencyMap.value(rowCurrency);
        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

我们的修改很简单：仅仅是增加了 `role == Qt::EditRole` 这么一行判断。这意味着，当是 `EditRole` 时，Qt 会提供一个默认值。我们可以试着删除这个判断来看看其产生的效果。

最后运行一下程序，修改一下数据就会发现，当我们修改过一个单元格后，Qt 会自动刷新所有受影响的数据的值。这也正是 `model / view` 模型的强大之处：对数据模型的修改会直接反映到视图上。

## 51 布尔表达式树模型

本章将会是自定义模型的最后一部分。原本打算结束这部分内容，不过实在不忍心放弃这个示例。来自于 *C++ GUI Programming with Qt 4, 2nd Edition* 这本书的布尔表达式树模型的示例相当精彩，复杂而又不失实用性，所以我们还是以这个例子结束这部分内容。

这个例子是将布尔表达式分析成一棵树。这个分析过程在离散数学中经常遇到，特别是复杂的布尔表达式。类似的分析方法可以套用于表达式化简、求值等一系列的运算。同时，这个技术也可以很方便地分析一个表达式是不是一个正确的布尔表达式。在这个例子中，一共有四个类：

- **Node**: 组成树的节点；
- **BooleanModel**: 布尔表达式的模型，实际上这是一个树状模型，用于将布尔表达式形象地呈现为一棵树；
- **BooleanParser**: 分析布尔表达式的分析器；
- **BooleanWindow**: 图形用户界面，用户在此输入布尔表达式并进行分析，最后将结果展现成一棵树。

首先，我们来看看最基础的 **Node** 类。这是分析树的节点，也是构成整棵树的基础。

```
class Node
{
public:
    enum Type { Root, OrExpression, AndExpression, NotExpression, Atom,
                Identifier, Operator, Punctuator };

    Node(Type type, const QString &str = "");
    ~Node();

    Type type;
    QString str;
    Node *parent;
    QList<Node *> children;
};
```

**Node** 的 `cpp` 文件也非常简单：

```
Node::Node(Type type, const QString &str)
{
    this->type = type;
    this->str = str;
    parent = 0;
```

```

}

Node::~~Node()
{
    qDeleteAll(children);
}

```

Node 很像一个典型的树的节点：一个 Node 指针类型的 parent 属性，保存父节点；一个 QString 类型的 str，保存数据。另外，Node 还有一个 Type 属性，指明这个 Node 的类型，是一个词素，还是操作符，或者其他什么东西；children 是 QList<Node \*>类型，保存这个 node 的所有子节点。注意，在 Node 类的析构函数中，使用了 qDeleteAll() 这个全局函数。这个函数是将 [start, end) 范围内的所有元素进行 delete 运算。因此，它的参数的元素必须是指针类型的。并且，这个函数使用 delete 之后并不会将指针赋值为 0，所以，如果要在析构函数之外调用这个函数，建议在调用之后显示的调用 clear() 函数，将列表所有元素的指针重置为 0。

虽然我们将这个例子放在自定义模型这部分，但实际上这个例子的核心类是 BooleanParser。我们来看一下 BooleanParser 的代码：

```

// booleanparser.h
class BooleanParser
{
public:
    Node *parse(const QString &expr);

private:
    Node *parseOrExpression();
    Node *parseAndExpression();
    Node *parseNotExpression();
    Node *parseAtom();
    Node *parseIdentifier();
    void addChild(Node *parent, Node *child);
    void addToken(Node *parent, const QString &str, Node::Type type);
    bool matchToken(const QString &str) const;

    QString in;
    int pos;
};

// booleanparser.cpp
Node *BooleanParser::parse(const QString &expr)
{
    in = expr;
}

```

```

        in.replace(" ", "");
        pos = 0;

        Node *node = new Node(Node::Root);
        addChild(node, parseOrExpression());
        return node;
    }

Node *BooleanParser::parseOrExpression()
{
    Node *childNode = parseAndExpression();
    if (matchToken("||")) {
        Node *node = new Node(Node::OrExpression);
        addChild(node, childNode);
        while (matchToken("||")) {
            addToken(node, "||", Node::Operator);
            addChild(node, parseAndExpression());
        }
        return node;
    } else {
        return childNode;
    }
}

Node *BooleanParser::parseAndExpression()
{
    Node *childNode = parseNotExpression();
    if (matchToken("&&")) {
        Node *node = new Node(Node::AndExpression);
        addChild(node, childNode);
        while (matchToken("&&")) {
            addToken(node, "&&", Node::Operator);
            addChild(node, parseNotExpression());
        }
        return node;
    } else {
        return childNode;
    }
}

Node *BooleanParser::parseNotExpression()
{
    if (matchToken("!")) {
        Node *node = new Node(Node::NotExpression);

```

```

        while (matchToken("!"))
            addToken(node, "!", Node::Operator);
        addChild(node, parseAtom());
        return node;
    } else {
        return parseAtom();
    }
}

Node *BooleanParser::parseAtom()
{
    if (matchToken("(")) {
        Node *node = new Node(Node::Atom);
        addToken(node, "(", Node::Punctuator);
        addChild(node, parseOrExpression());
        addToken(node, ")", Node::Punctuator);
        return node;
    } else {
        return parseIdentifier();
    }
}

Node *BooleanParser::parseIdentifier()
{
    int startPos = pos;
    while (pos < in.length() && in[pos].isLetterOrNumber())
        ++pos;
    if (pos > startPos) {
        return new Node(Node::Identifier,
                        in.mid(startPos, pos - startPos));
    } else {
        return 0;
    }
}

void BooleanParser::addChild(Node *parent, Node *child)
{
    if (child) {
        parent->children += child;
        parent->str += child->str;
        child->parent = parent;
    }
}

```

```

void BooleanParser::addToken(Node *parent, const QString &str,
                             Node::Type type)
{
    if (in.mid(pos, str.length()) == str) {
        addChild(parent, new Node(type, str));
        pos += str.length();
    }
}

bool BooleanParser::matchToken(const QString &str) const
{
    return in.mid(pos, str.length()) == str;
}

```

这里我们一次把 **BooleanParser** 的所有代码全部列了出来。我们首先从轮廓上面来看一下，**BooleanParser** 作为核心类，并没有掺杂有关界面的任何代码。这是我们提出这个例子的另外一个重要原因：分层。对于初学者而言，如何设计好一个项目至关重要。分层就是其中一个重要的设计手法。或许你已经明白了 MVC 架构的基本概念，在这里也不再赘述。简单提一句，所谓分层，就是将程序的不同部分完全分离。比如这里的 **BooleanParser** 类，仅仅是处理 **Node** 的节点，然后返回处理结果，至于处理结果如何显示，**BooleanParser** 不去关心。通过前面我们了解到的 model/view 的相关知识也可以看出，这样做的好处是，今天我们可以使用 **QAbstractItemModel** 来显示这个结果，明天我发现图形界面不大合适，我想换用字符界面显示——没问题，只需要替换掉用于显示的部分就可以了。

大致了解了 **BooleanParser** 的总体设计思路（也就是从显示逻辑完全剥离开来）后，我们详细看看这个类的业务逻辑，也就是算法。虽然算法不是我们这里的重点，但是针对一个示例而言，这个算法是最核心的部分，并且体现了一类典型的算法，豆子觉得还是有必要了解下。

注意到 **BooleanParser** 类只有一个公共函数，显然我们必须从这里着手来理解这个算法。在 **Node \* parse(const QString &)** 函数中，首先将传入的布尔表达式的字符串保存下来，避免直接修改参数（这也是库的接口设计中常见的一个原则：不修改参数）；然后我们将其中的空格全部去掉，并将 **pos** 设为 0。**pos** 就是我们在分析布尔表达式字符串时的当前字符位置，起始为 0。之后我们创建了 **Root** 节点——布尔表达式的树状表达，显然需要有一个根节点，所以我们在这里直接创建根节点，这个根节点就是一个完整的布尔表达式。

首先我们先来看看布尔表达式的文法：

```

BE → BE OR BE
    | BE AND BE
    | NOT BE
    | (BE)

```



```
| RE | true | false
RE → RE RELOP RE | (RE) | E
E → E op E | -E | (E) | Identifier | Number
```

这是一个相对比较完整的布尔表达式文法。这里我们只使用其中一部分：

```
BE → BE OR BE
    | BE AND BE
    | NOT BE
    | (BE)
    | Identifier
```

从我们简化的文法可以看出，布尔表达式 `BE` 可以由 `BE | BE`、`BE AND BE`、`NOT BE`、`(BE)` 和 `Identifier` 五部分组成，而每一部分都可以再递归地由 `BE` 进行定义。

接下来看算法的真正核心：我们按照上述文法来展开算法。要处理一个布尔表达式，或运算的优先级是最低，应该最后被处理。一旦或运算处理完毕，意味着整个布尔表达式已经处理完毕，所以我们在调用了 `addChild(node, parseOrExpression())` 之后，返回整个 `node`。下面来看 `parseOrExpression()` 函数。要想处理 `OR` 运算，首先要处理 `AND` 运算，于是 `parseOrExpression()` 函数的第一句，我们调用了 `parseAndExpression()` 函数。要想处理 `AND` 运算，首先要处理 `NOT` 运算，于是 `parseAndExpression()` 的第一句，我们调用了 `parseNotExpression()` 函数。在 `parseNotExpression()` 函数中，检查第一个字符是不是 `!`，如果是，意味着这个表达式是一个 `NOT` 运算，生成 `NOT` 节点。`NOT` 节点可能会有两种不同的情况：

1. 子表达式（也就是用括号包围起来的部分，由于这部分的优先级最高，所以看做是一个完整的子表达式），子表达式是原子性的，需要一个独立的处理，也要生成一个节点，其分隔符是 `(` 和 `)`。`(` 和 `)` 之间又是一个完整的布尔表达式，回忆一下，一个完整的布尔表达式最后要处理的部分是 `OR` 运算，因此调用 `parseOrExpression()` 函数进行递归。
2. 标识符（如果 `!` 符号后面不是 `(` 和 `)`，则只能是一个标识符，这是布尔表达式文法决定的），我们使用 `parseIdentifier()` 函数来获得这个标识符。这个函数很简单：从 `pos` 位置开始一个个检查当前字符是不是字母，如果是，说明这个字符是标识符的一部分，如果不是，说明标识符已经在上一个字符的位置结束（注意，是上一个字符的位置，而不是当前字符，当检测到当前字符不是字母时，说明标识符已经在上一个字母那里结束了，当前字母不属于标识符的一部分），我们截取 `startPos` 开始，`pos - startPos` 长度的字符串作为标识符名称，而不是 `pos - startPos + 1` 长度。

`NOT` 节点处理完毕，函数返回到 `parseAndExpression()`。如果 `NOT` 节点后面是 `&&`，说明是 `AND` 节点。我们生成一个 `AND` 节点，把刚刚处理过的 `NOT` 节点添加为其子节点，

如果一直找到了 && 符号，就要一直作为 AND 节点处理，直到找到的不是 &&，AND 节点处理完毕，返回这个 node。另一方面，如果 NOT 节点后面不是 &&，说明根本不是 AND 节点，则直接把刚刚处理过的 NOT 节点返回。函数重新回到 parseOrExpression() 这里。此时需要检查是不是 ||，其过程同 && 类型，这里不再赘述。

这个过程看起来非常复杂，实际非常清晰：一层一层按照文法递归执行，从最顶层一直到最底层。如果把有限自动机图示画出来，这个过程非常简洁明了。这就是编译原理的词法分析中最重要的算法之一：**递归下降算法**。由于这个算法简洁明了，很多编译器的词法分析都是使用的这个算法（当然，其性能有待商榷，所以成熟的编译器很可能选择了其它性能更好的算法）。最后，如果你觉得对这部分理解困难，不妨跳过，原本有关编译原理的内容都比较复杂。

最复杂的算法已经完成，接下来是 BooleanModel 类：

```
class BooleanModel : public QAbstractItemModel
{
public:
    BooleanModel(QObject *parent = 0);
    ~BooleanModel();

    void setRootNode(Node *node);

    QModelIndex index(int row, int column,
                      const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;

private:
    Node *nodeFromIndex(const QModelIndex &index) const;

    Node *rootNode;
};
```

BooleanModel 类继承了 QAbstractItemModel。之所以不继承 QAbstractListModel 或者 QAbstractTableModel，是因为我们要构造一个带有层次结构的模型。在构造函数中，我们把根节点的指针赋值为 0，因此我们提供了另外的一个函数 setRootNode()，将根节点进行有效地赋值。而在析构中，我们直接使用 delete 操作符将这个根节点释放掉。在

setRootNode()函数中，首先我们释放原有的根节点，再将根节点赋值。此时我们需要通知所有视图对界面进行重绘，以表现最新的数据：

```
BooleanModel::BooleanModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}

BooleanModel::~~BooleanModel()
{
    delete rootNode;
}

void BooleanModel::setRootNode(Node *node)
{
    beginResetModel();
    delete rootNode;
    rootNode = node;
    endResetModel();
}
```

直接继承 QAbstractItemModel 类，我们必须实现它的五个纯虚函数。首先是 index() 函数。这个函数在 QAbstractTableModel 或者 QAbstractListModel 中不需要实现，因此那两个类已经实现过了。但是，因为我们现在继承的 QAbstractItemModel，必须提供一个合适的实现：

```
QModelIndex BooleanModel::index(int row, int column,
                                const QModelIndex &parent) const
{
    if (!rootNode || row < 0 || column < 0)
        return QModelIndex();
    Node *parentNode = nodeFromIndex(parent);
    Node *childNodes = parentNode->children.value(row);
    if (!childNodes)
        return QModelIndex();
    return createIndex(row, column, childNodes);
}
```

index() 函数用于返回第 row 行，第 column 列，父节点为 parent 的那个元素的 QModelIndex 对象。对于树状模型，我们关注的是其 parent 参数。在我们实现中，如果 rootNode 或者 row 或者 column 非法，直接返回一个非法的 QModelIndex。否则的话，使用 nodeFromIndex() 函数取得索引为 parent 的节点，然后使用 children 属性（这是我

们前面定义的 Node 里面的属性)获得子节点。如果子节点不存在,返回一个非法值;否则,返回由 createIndex()函数创建的一个有效的 QModelIndex 对象。对于具有层次结构的模型,只有 row 和 column 值是不能确定这个元素的位置的,因此, QModelIndex 中除了 row 和 column 之外,还有一个 void \*或者 int 的空白属性,可以存储一个值。在这里我们就把父节点的指针存入,这样,就可以由这三个属性定位这个元素。这里的 createIndex()第三个参数就是这个内部使用的指针。所以我们自己定义一个 nodeFromIndex()函数的时候要注意使用 QModelIndex 的 internalPointer()函数获得这个内部指针,从而定位我们的节点。

rowCount()和 columnCount()两个函数相对简单:

```
int BooleanModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
        return 0;
    return parentNode->children.count();
}

int BooleanModel::columnCount(const QModelIndex & /* parent */) const
{
    return 2;
}
```

对于 rowCount(), 显然返回的是 parentNode 的子节点的数目;对于 columnCount(), 由于我们界面分为两列, 所以始终返回 2。

parent()函数返回子节点所属的父节点的索引。我们需要从子节点开始寻找, 直到找到其父节点的父节点, 这样才能定位到这个父节点, 从而得到子节点的位置。而 data()函数则要返回每个单元格的显示值。在前面两章的基础之上, 我们应该可以很容易地理解这两个函数的内容。headerData()函数返回列头的名字, 同前面一样, 这里就不再赘述了:

```
QModelIndex BooleanModel::parent(const QModelIndex &child) const
{
    Node *node = nodeFromIndex(child);
    if (!node)
        return QModelIndex();
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();
    Node *grandparentNode = parentNode->parent;
```

```

    if (!grandparentNode)
        return QModelIndex();

    int row = grandparentNode->children.indexOf(parentNode);
    return createIndex(row, 0, parentNode);
}

QVariant BooleanModel::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();

    if (index.column() == 0) {
        switch (node->type) {
            case Node::Root:
                return tr("Root");
            case Node::OrExpression:
                return tr("OR Expression");
            case Node::AndExpression:
                return tr("AND Expression");
            case Node::NotExpression:
                return tr("NOT Expression");
            case Node::Atom:
                return tr("Atom");
            case Node::Identifier:
                return tr("Identifier");
            case Node::Operator:
                return tr("Operator");
            case Node::Punctuator:
                return tr("Punctuator");
            default:
                return tr("Unknown");
        }
    } else if (index.column() == 1) {
        return node->str;
    }

    return QVariant();
}

QVariant BooleanModel::headerData(int section,

```

```

Qt::Orientation orientation,
int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}

```

最后是我们定义的一个辅助函数：

```

Node *BooleanModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return rootNode;
    }
}

```

正如我们上面所说的那样，我们利用 `index` 内部存储的一个指针来获取 `index` 对应的节点。

最后，`BooleanWindow` 类非常简单，我们不再详细解释它的代码：

```

BooleanWindow::BooleanWindow()
{
    label = new QLabel(tr("Boolean expression:"));
    lineEdit = new QLineEdit;

    booleanModel = new BooleanModel(this);

    treeView = new QTreeView;
    treeView->setModel(booleanModel);

    connect(lineEdit, SIGNAL(textChanged(const QString &)),
            this, SLOT(booleanExpressionChanged(const QString &)));

    QGridLayout *layout = new QGridLayout;
    layout->addWidget(label, 0, 0);
    layout->addWidget(lineEdit, 0, 1);
}

```

```

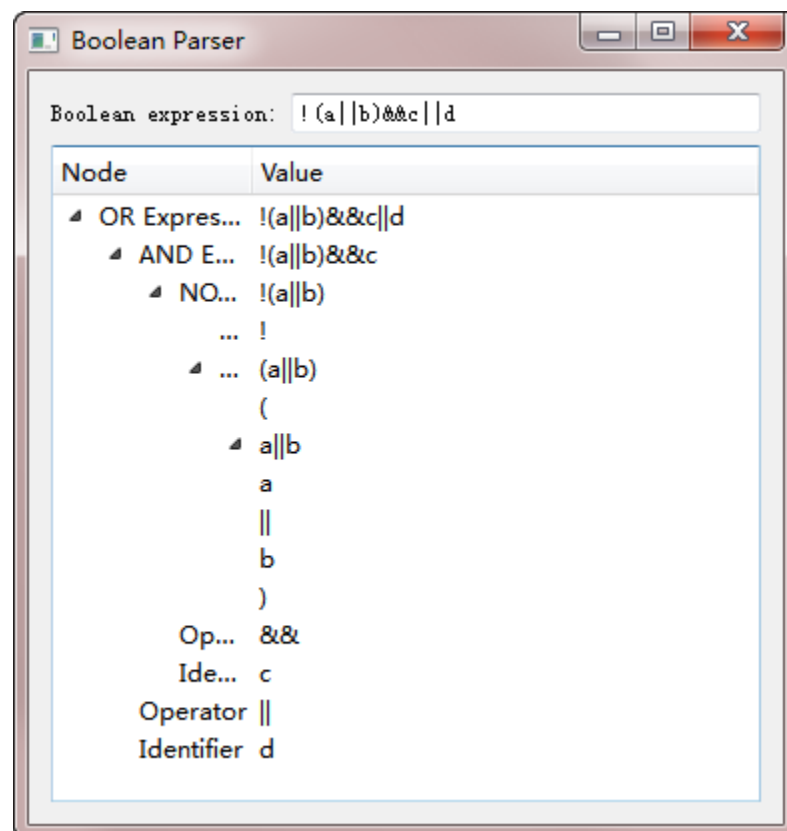
layout->addWidget(treeView, 1, 0, 1, 2);
setLayout(layout);

setWindowTitle(tr("Boolean Parser"));
}

void BooleanWindow::booleanExpressionChanged(const QString &expr)
{
    BooleanParser parser;
    Node *rootNode = parser.parse(expr);
    booleanModel->setRootNode(rootNode);
}

```

这样，我们的布尔表达式树模型已经创建完毕。下面来运行一下看看效果：



最后，我们附上整个项目的代码：[下载](#)

## 52 使用拖放

拖放（Drag and Drop），通常会简称为 DnD，是现代软件开发中必不可少的一项技术。它提供了一种能够在应用程序内部甚至是应用程序之间进行信息交换的机制。操作系统与应用程序之间进行的剪贴板内容的交换，也可以被认为是拖放的一部分。

拖放其实是由两部分组成的：拖动和释放。拖动是将被拖放对象进行移动，释放是将被拖放对象放下。前者是一个按下鼠标按键并移动的过程，后者是一个松开鼠标按键的过程；通常这两个操作之间的鼠标按键是被一直按下的。当然，这只是一种普遍的情况，其它情况还是要看应用程序的具体实现。对于 Qt 而言，一个组件既可以作为被拖动对象进行拖动，也可以作为释放掉的目的地对象，或者二者都是。

在下面的例子中（来自 C++ GUI Programming with Qt4, 2nd Edition），我们将创建一个程序，将操作系统中的文本文件拖进来，然后在窗口中读取内容。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);
private:
    bool readFile(const QString &fileName);
    QTextEdit *textEdit;
};
```

注意到我们需要重写 `dragEnterEvent()` 和 `dropEvent()` 两个函数。顾名思义，前者是拖放进入的事件，后者是释放鼠标的事件。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    textEdit->setAcceptDrops(false);
    setAcceptDrops(true);

    setWindowTitle(tr("Text Editor"));
}

MainWindow::~~MainWindow()
```



```
{  
}
```

在构造函数中，我们创建了 `QTextEdit` 的对象。默认情况下，`QTextEdit` 可以接受从其它应用程序拖放过来的文本类型的数据。如果用户把一个文件拖到这面，默认会把文件名插入到光标位置。但是我们希望让 `MainWindow` 读取文件内容，而不是仅仅插入文件名，所以我们在 `MainWindow` 中加入了拖放操作。首先要把 `QTextEdit` 的 `setAcceptDrops()` 函数置为 `false`，并且把 `MainWindow` 的 `setAcceptDrops()` 置为 `true`，这样我们就能够让 `MainWindow` 截获拖放事件，而不是交给 `QTextEdit` 处理。

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)  
{  
    if (event->mimeTypeData()->hasFormat("text/uri-list")) {  
        event->acceptProposedAction();  
    }  
}
```

当用户将对象拖动到组件上面时，系统会回调 `dragEnterEvent()` 函数。如果我们在事件处理代码中调用 `acceptProposedAction()` 函数，就可以向用户暗示，你可以将拖动的对象放在这个组件上。默认情况下，组件是不会接受拖放的。如果我们调用了这个函数，那么 Qt 会自动以光标样式的变化来提示用户是否可以将对象放在组件上。在这里，我们希望告诉用户，窗口可以接受拖放，但是我们仅接受某一种类型的文件，而不是全部文件。我们首先检查拖放文件的 MIME 类型信息。MIME 类型由 Internet Assigned Numbers Authority (IANA) 定义，Qt 的拖放事件使用 MIME 类型来判断拖放对象的类型。关于 MIME 类型的详细信息，请参考 <http://www.iana.org/assignments/media-types/>。MIME 类型为 `text/uri-list` 通常用来描述一个 URI 列表。这些 URI 可以是文件名，可以是 URL 或者其它的资源描述符。如果发现用户拖放的是一个 `text/uri-list` 数据（即文件名），我们便接受这个动作。

```
void MainWindow::dropEvent(QDropEvent *event)  
{  
    QList<QUrl> urls = event->mimeTypeData()->urls();  
    if (urls.isEmpty()) {  
        return;  
    }  
  
    QString fileName = urls.first().toLocalFile();  
    if (fileName.isEmpty()) {  
        return;  
    }  
  
    if (readFile(fileName)) {  
        setWindowTitle(tr("%1 - %2").arg(fileName, tr("Drag File")));  
    }  
}
```

```

    }
}

bool MainWindow::readFile(const QString &fileName)
{
    bool r = false;
    QFile file(fileName);
    QString content;
    if(file.open(QIODevice::ReadOnly)) {
        content = file.readAll();
        r = true;
    }
    textEdit->setText(content);
    return r;
}

```

当用户将对象释放到组件上面时，系统回调 `dropEvent()` 函数。我们使用 `QMimeData::urls()` 来获得 `QUrl` 的一个列表。通常，这种拖动应该只有一个文件，但是也不排除多个文件一起拖动。因此我们需要检查这个列表是否为空，如果不为空，则取出第一个，否则立即返回。最后我们调用 `readFile()` 函数读取文件内容。这个函数的内容很简单，我们前面也讲解过有关文件的操作，这里不再赘述。现在可以运行下看看效果了。

接下来的例子也是来自 `C++ GUI Programming with Qt4, 2nd Edition`。在这个例子中，我们将创建左右两个并列的列表，可以实现二者之间数据的相互拖动。

```

class ProjectListWidget : public QListWidget
{
    Q_OBJECT
public:
    ProjectListWidget(QWidget *parent = 0);
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);
private:
    void performDrag();
    QPoint startPos;
};

```

`ProjectListWidget` 是我们的列表的实现。这个类继承自 `QListWidget`。在最终的程序中，将会是两个 `ProjectListWidget` 的并列。

```
ProjectListWidget::ProjectListWidget(QWidget *parent)
    : QListWidget(parent)
{
    setAcceptDrops(true);
}
```

构造函数我们设置了 `setAcceptDrops()`，使 `ProjectListWidget` 能够支持拖动操作。

```
void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        startPos = event->pos();
    QListWidget::mousePressEvent(event);
}

void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QListWidget::mouseMoveEvent(event);
}

void ProjectListWidget::performDrag()
{
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeData = new QMimeData;
        mimeData->setText(item->text());

        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeData);
        drag->setPixmap(QPixmap(":/images/person.png"));
        if (drag->exec(Qt::MoveAction) == Qt::MoveAction)
            delete item;
    }
}
```

`mousePressEvent()`函数中，我们检测鼠标左键点击，如果是的话就记录下当前位置。需要注意的是，这个函数最后需要调用系统自带的处理函数，以便实现通常的那种操作。这在一些重写事件的函数中都是需要注意的，前面我们已经反复强调过这一点。

`mouseMoveEvent()`函数判断了，如果鼠标在移动的时候一直按住左键（也就是 `if` 里面的内容），那么就计算一个 `manhattanLength()`值。从字面上翻译，这是个“曼哈顿长度”。首先来看看 `event.pos() - startPos` 是什么。在 `mousePressEvent()`函数中，我们将鼠标按下的坐标记录为 `startPos`，而 `event.pos()`则是鼠标当前的坐标：一个点减去另外一个点，这就是一个位移向量。所谓曼哈顿距离就是两点之间的距离（按照勾股定理进行计算而来），也就是这个向量的长度。然后继续判断，如果大于 `QApplication::startDragDistance()`，我们才进行释放的操作。当然，最后还是要调用系统默认的鼠标拖动函数。这一判断的意义在于，防止用户因为手的抖动等因素造成的鼠标拖动。用户必须将鼠标拖动一段距离之后，我们才认为他是希望进行拖动操作，而这一距离就是 `QApplication::startDragDistance()`提供的，这个值通常是 4px。

`performDrag()`开始处理拖放的过程。这里，我们要创建一个 `QDrag` 对象，将 `this` 作为 `parent`。`QDrag` 使用 `QMimeData` 存储数据。例如我们使用 `QMimeData::setText()`函数将一个字符串存储为 `text/plain` 类型的数据。`QMimeData` 提供了很多函数，用于存储诸如 URL、颜色等类型的数据。使用 `QDrag::setPixmap()`则可以设置拖动发生时鼠标的样式。`QDrag::exec()`会阻塞拖动的操作，直到用户完成操作或者取消操作。它接受不同类型的动作作为参数，返回值是真正执行的动作。这些动作的类型一般为 `Qt::CopyAction`，`Qt::MoveAction` 和 `Qt::LinkAction`。返回值会有这几种动作，同时还会有一个 `Qt::IgnoreAction` 用于表示用户取消了拖放。这些动作取决于拖放源对象允许的类型，目的对象接受的类型以及拖放时按下的键盘按键。在 `exec()`调用之后，Qt 会在拖放对象不需要的时候释放掉。

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
    ProjectListWidget *source =
        qobject_cast(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}

void ProjectListWidget::dropEvent(QDropEvent *event)
{
    ProjectListWidget *source =
        qobject_cast(event->source());
    if (source && source != this) {
        addItem(event->mimeTypeData()->text());
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

`dragMoveEvent()`和 `dropEvent()`相似。首先判断事件的来源（`source`），由于我们是两个 `ProjectListWidget` 之间相互拖动，所以来源应该是 `ProjectListWidget` 类型的（当然，这个 `source` 不能是自己，所以我们还得判断 `source != this`）。`dragMoveEvent()`中我们检查的是被拖动的对象；`dropEvent()`中我们检查的是释放的对象：这二者是不同的。

附件：[ProjectListWidget](#)

## 53 自定义拖放数据

上一章中，我们的例子使用系统提供的拖放对象 `QMimeType` 进行拖放数据的存储。比如使用 `QMimeType::setText()` 创建文本，使用 `QMimeType::urls()` 创建 URL 对象等。但是，如果你希望使用一些自定义的对象作为拖放数据，比如自定义类等等，单纯使用 `QMimeType` 可能就没有那么容易了。为了实现这种操作，我们可以从下面三种实现方式中选择一个：

1. 将自定义数据作为 `QByteArray` 对象，使用 `QMimeType::setData()` 函数作为二进制数据存储到 `QMimeType` 中，然后使用 `QMimeType::data()` 读取
2. 继承 `QMimeType`，重写其中的 `formats()` 和 `retrieveData()` 函数操作自定义数据
3. 如果拖放操作仅仅发生在同一个应用程序，可以直接继承 `QMimeType`，然后使用任意合适的数据结构进行存储

这三种选择各有千秋：第一种方法不需要继承任何类，但是有一些局限：即是拖放不会发生，我们也必须将自定义的数据对象转换成 `QByteArray` 对象，在一定程度上，这会降低程序性能；另外，如果你希望支持很多种拖放的数据，那么每种类型的数据都必须使用一个 `QMimeType` 类，这可能会导致类爆炸。后两种实现方式则不会有这些问题，或者说是能够减小这种问题，并且能够让我们有完全的控制权。

下面我们使用第一种方法来实现一个表格。这个表格允许我们选择一部分数据，然后拖放到另外的一个空白表格中。在数据拖动过程中，我们使用 CSV 格式对数据进行存储。

首先来看头文件：

```
class DataTableWidget : public QTableWidgetItem
{
    Q_OBJECT
public:
    DataTableWidget(QWidget *parent = 0);
```

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);
private:
    void performDrag();
    QString selectionText() const;

    QString toHtml(const QString &plainText) const;
    QString toCsv(const QString &plainText) const;
    void fromCsv(const QString &csvText);

    QPoint startPos;
};
```

这里，我们的表格继承自 `QTableWidget`。虽然这是一个简化的 `QTableView`，但对于我们的演示程序已经绰绰有余。

```
DataTableWidget::DataTableWidget(QWidget *parent)
    : QTableWidget(parent)
{
    setAcceptDrops(true);
    setSelectionMode(ContiguousSelection);

    setColumnCount(3);
    setRowCount(5);
}

void DataTableWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        startPos = event->pos();
    }
    QTableWidget::mousePressEvent(event);
}

void DataTableWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance()) {
            performDrag();
        }
    }
}
```

```

    }
}

void DataTableWidget::dragEnterEvent(QDragEnterEvent *event)
{
    DataTableWidget *source =
        qobject_cast<DataTableWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}

void DataTableWidget::dragMoveEvent(QDragMoveEvent *event)
{
    DataTableWidget *source =
        qobject_cast<DataTableWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}

```

构造函数中，由于我们要针对两个表格进行相互拖拽，所以我们设置了 `setAcceptDrops()` 函数。选择模式设置为连续，这是为了方便后面我们的算法简单。`mousePressEvent()`，`mouseMoveEvent()`，`dragEnterEvent()`以及 `dragMoveEvent()`四个事件响应函数与前面几乎一摸一样，这里不再赘述。注意，这几个函数中有一些并没有调用父类的同名函数。关于这一点我们在前面的章节中曾反复强调，但这里我们不希望父类的实现被执行，因此完全屏蔽了父类实现。下面我们来看 `performDrag()`函数：

```

void DataTableWidget::performDrag()
{
    QString selectedString = selectionText();
    if (selectedString.isEmpty()) {
        return;
    }

    QMimeData *mimeData = new QMimeData;
    mimeData->setHtml(toHtml(selectedString));
    mimeData->setData("text/csv", toCsv(selectedString).toUtf8());

    QDrag *drag = new QDrag(this);
    drag->setMimeData(mimeData);
}

```

```

        if (drag->exec(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction) {
            selectionModel()->clearSelection();
        }
    }
}

```

首先我们获取选择的文本（`selectionText()`函数），如果为空则直接返回。然后创建一个 `QMimeData` 对象，设置了两个数据：HTML 格式和 CSV 格式。我们的 CSV 格式是以 `QByteArray` 形式存储的。之后我们创建了 `QDrag` 对象，将这个 `QMimeData` 作为拖动时所需要的数据，执行其 `exec()`函数。`exec()`函数指明，这里的拖动操作接受两种类型：复制和移动。当执行的是移动时，我们将已选区域清除。

需要注意一点，`QMimeData` 在创建时并没有提供 `parent` 属性，这意味着我们必须手动调用 `delete` 将其释放。但是，`setMimeData()`函数会将其所有权转移到 `QDrag` 名下，也就是会将其 `parent` 属性设置为这个 `QDrag`。这意味着，当 `QDrag` 被释放时，其名下的所有 `QMimeData` 对象都会被释放，所以结论是，我们实际是无需，也不能手动 `delete` 这个 `QMimeData` 对象。

```

void DataTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeTypeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        fromCsv(csvText);
        event->acceptProposedAction();
    }
}

```

`dropEvent()`函数也很简单：如果是 CSV 类型，我们取出数据，转换成字符串形式，调用了 `fromCsv()`函数生成新的数据项。

几个辅助函数的实现比较简单：

```

QString DataTableWidget::selectionText() const
{
    QString selectionString;
    QString headerString;
    QAbstractItemModel *itemModel = model();
    QTableWidgetItemSelectionRange selection = selectedRanges().at(0);
    for (int row = selection.topRow(), firstRow = row;
         row <= selection.bottomRow(); row++) {
        for (int col = selection.leftColumn();
             col <= selection.rightColumn(); col++) {
            if (row == firstRow) {

```



```

headerString.append(horizontalHeaderItem(col)->text()).append("\t");
    }
    QModelIndex index = itemModel->index(row, col);
    selectionString.append(index.data().toString()).append("\t");
    }
    selectionString = selectionString.trimmed();
    selectionString.append("\n");
    }
    return headerString.trimmed() + "\n" + selectionString.trimmed();
}

```

```

QString DataTableWidget::toHtml(const QString &plainText) const
{
    #if QT_VERSION >= 0x050000
        QString result = plainText.toHtmlEscaped();
    #else
        QString result = Qt::escape(plainText);
    #endif
    result.replace("\t", "<td>");
    result.replace("\n", "\n<tr><td>");
    result.prepend("<table>\n<tr><td>");
    result.append("\n</table>");
    return result;
}

```

```

QString DataTableWidget::toCsv(const QString &plainText) const
{
    QString result = plainText;
    result.replace("\\", "\\");
    result.replace("\"", "\\");
    result.replace("\t", "\", \"");
    result.replace("\n", "\"\n\"");
    result.prepend("\"");
    result.append("\"");
    return result;
}

```

```

void DataTableWidget::fromCsv(const QString &csvText)
{
    QStringList rows = csvText.split("\n");
    QStringList headers = rows.at(0).split(", ");
    for (int h = 0; h < headers.size(); ++h) {
        QString header = headers.at(h);
    }
}

```

```

        headers.replace(h, header.replace("'", ""));
    }
    setHorizontalHeaderLabels(headers);
    for (int r = 1; r < rows.size(); ++r) {
        QStringList row = rows.at(r).split(", ");
        setItem(r - 1, 0, new QTableWidgetItem(row.at(0).trimmed().replace("'", "")));
        setItem(r - 1, 1, new QTableWidgetItem(row.at(1).trimmed().replace("'", "")));
    }
}

```

虽然看起来很长，但是这几个函数都是纯粹算法，而且算法都比较简单。注意 `toHtml()` 中我们使用条件编译语句区分了一个 Qt4 与 Qt5 的不同函数。这也是让同一代码能够同时应用于 Qt4 和 Qt5 的技巧。`fromCsv()` 函数中，我们直接将下面表格的前面几列设置为拖动过来的数据，注意这里有一些格式上面的变化，主要用于更友好地显示。

最后是 `MainWindow` 的一个简单实现：

```

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    topTable = new DataTableWidget(this);
    QStringList headers;
    headers << "ID" << "Name" << "Age";
    topTable->setHorizontalHeaderLabels(headers);
    topTable->setItem(0, 0, new QTableWidgetItem(QString("0001")));
    topTable->setItem(0, 1, new QTableWidgetItem(QString("Anna")));
    topTable->setItem(0, 2, new QTableWidgetItem(QString("20")));
    topTable->setItem(1, 0, new QTableWidgetItem(QString("0002")));
    topTable->setItem(1, 1, new QTableWidgetItem(QString("Tommy")));
    topTable->setItem(1, 2, new QTableWidgetItem(QString("21")));
    topTable->setItem(2, 0, new QTableWidgetItem(QString("0003")));
    topTable->setItem(2, 1, new QTableWidgetItem(QString("Jim")));
    topTable->setItem(2, 2, new QTableWidgetItem(QString("21")));
    topTable->setItem(3, 0, new QTableWidgetItem(QString("0004")));
    topTable->setItem(3, 1, new QTableWidgetItem(QString("Dick")));
    topTable->setItem(3, 2, new QTableWidgetItem(QString("24")));
    topTable->setItem(4, 0, new QTableWidgetItem(QString("0005")));
    topTable->setItem(4, 1, new QTableWidgetItem(QString("Tim")));
    topTable->setItem(4, 2, new QTableWidgetItem(QString("22")));

    bottomTable = new DataTableWidget(this);
}

```

```

QWidget *content = new QWidget(this);
QVBoxLayout *layout = new QVBoxLayout(content);
layout->addWidget(topTable);
layout->addWidget(bottomTable);

setCentralWidget(content);

setWindowTitle("Data Table");
}

```

这段代码没有什么新鲜内容，我们直接将其跳过。最后编译运行下程序，按下 `shift` 并点击表格两个单元格即可选中，然后拖放到另外的空白表格中来查看效果。

下面我们换用继承 `QMimeData` 的方法来尝试重新实现上面的功能。

```

class TableMimeData : public QMimeData
{
    Q_OBJECT
public:
    TableMimeData(const QTableWidgetItem *tableWidget,
                  const QTableWidgetItemSelectionRange &range);
    const QTableWidgetItem *tableWidget() const
    {
        return dataTableWidget;
    }
    QTableWidgetItemSelectionRange range() const
    {
        return selectionRange;
    }
    QStringList formats() const
    {
        return dataFormats;
    }
protected:
    QVariant retrieveData(const QString &format,
                          QVariant::Type preferredType) const;
private:
    static QString toHtml(const QString &plainText);
    static QString toCsv(const QString &plainText);
    QString text(int row, int column) const;
    QString selectionText() const;

    const QTableWidgetItem *dataTableWidget;
    QTableWidgetItemSelectionRange selectionRange;
}

```

```

    QStringList dataFormats;
};

```

为了避免存储具体的数据，我们存储表格的指针和选择区域的坐标的指针；`dataFormats` 指明这个数据对象所支持的数据格式。这个格式列表由 `formats()` 函数返回，意味着所有被 MIME 数据对象支持的数据类型。这个列表是没有先后顺序的，但是最佳实践是将“最适合”的类型放在第一位。对于支持多种类型的应用程序而言，有时候会直接选用第一个符合的类型存储。

```

TableMimeData::TableMimeData(const QTableWidgetItem *tableWidget,
                             const QTableWidgetItemSelectionRange &range)
{
    dataTableWidget = tableWidget;
    selectionRange = range;
    dataFormats << "text/csv" << "text/html";
}

```

函数 `retrieveData()` 将给定的 MIME 类型作为 `QVariant` 返回。参数 `format` 的值通常是 `formats()` 函数返回值之一，但是我们并不能假定一定是这个值之一，因为并不是所有的应用程序都会通过 `formats()` 函数检查 MIME 类型。一些返回函数，比如 `text()`, `html()`, `urls()`, `imageData()`, `colorData()` 和 `data()` 实际上都是在 `QMimeData` 的 `retrieveData()` 函数中实现的。第二个参数 `preferredType` 给出我们应该在 `QVariant` 中存储哪种类型的数据。在这里，我们简单的将其忽略了，并且在 `else` 语句中，我们假定 `QMimeData` 会自动将其转换成所需要的类型：

```

QVariant TableMimeData::retrieveData(const QString &format,
                                     QVariant::Type preferredType) const
{
    if (format == "text/csv") {
        return toCsv(selectionText());
    } else if (format == "text/html") {
        return toHtml(selectionText());
    } else {
        return QMimeData::retrieveData(format, preferredType);
    }
}

```

在组件的 `dragEvent()` 函数中，需要按照自己定义的数据类型进行选择。我们使用 `qobject_cast` 宏进行类型转换。如果成功，说明数据来自同一应用程序，因此我们直接设置 `QTableWidgetItem` 相关数据，如果转换失败，我们则使用一般的处理方式。这也是这类程序通常的处理方式：

```

void DataTableWidget::dropEvent(QDropEvent *event)
{
    const TableMimeData *tableData =
        qobject_cast<const TableMimeData *>(event->mimeType());

    if (tableData) {
        const QTableWidgetItem *otherTable = tableData->tableWidget();
        QTableWidgetItemSelectionRange otherRange = tableData->range();
        // ...
        event->acceptProposedAction();
    } else if (event->mimeType()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeType()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        // ...
        event->acceptProposedAction();
    }
    QTableWidgetItem::mousePressEvent(event);
}

```

由于这部分代码与前面的相似，感兴趣的童鞋可以根据前面的代码补全这部分，所以这里不再给出完整代码。

## 54 剪贴板

剪贴板的操作经常和前面所说的拖放技术在一起使用。大家对剪贴板都很熟悉。我们可以简单地把它理解成一个数据存储池，外面的数据可以存进去，里面数据也可以取出来。剪贴板是由操作系统维护的，所以这提供了跨应用程序的数据交互的一种方式。Qt 已经为我们封装好很多关于剪贴板的操作，我们可以在自己的应用中很容易实现对剪贴板的支持，代码实现起来也是很简单的：

```

class ClipboardDemo : public QWidget
{
    Q_OBJECT
public:
    ClipboardDemo(QWidget *parent = 0);
private slots:
    void setClipboardContent();
    void getClipboardContent();
};

```

我们定义了一个 ClipboardDemo 类。这个类只有两个槽函数，一个是从剪贴板获取内容，一个是给剪贴板设置内容。

```
ClipboardDemo::ClipboardDemo(QWidget *parent)
    : QWidget(parent)
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    QHBoxLayout *northLayout = new QHBoxLayout;
    QHBoxLayout *southLayout = new QHBoxLayout;

    QTextEdit *editor = new QTextEdit;
    QLabel *label = new QLabel;
    label->setText("Text Input: ");
    label->setBuddy(editor);
    QPushButton *copyButton = new QPushButton;
    copyButton->setText("Set Clipboard");
    QPushButton *pasteButton = new QPushButton;
    pasteButton->setText("Get Clipboard");

    northLayout->addWidget(label);
    northLayout->addWidget(editor);
    southLayout->addWidget(copyButton);
    southLayout->addWidget(pasteButton);
    mainLayout->addLayout(northLayout);
    mainLayout->addLayout(southLayout);

    connect(copyButton, SIGNAL(clicked()), this, SLOT(setClipboardContent()));
    connect(pasteButton, SIGNAL(clicked()), this, SLOT(getClipboardContent()));
}
```

主界面也很简单：程序分为上下两行，上一行显示一个文本框，下一行是两个按钮，分别为设置剪贴板和读取剪贴板。最主要的代码还是在两个槽函数中：

```
void ClipboardDemo::setClipboardContent()
{
    QClipboard *board = QApplication::clipboard();
    board->setText("Text from Qt Application");
}

void ClipboardDemo::getClipboardContent()
{
    QClipboard *board = QApplication::clipboard();
    QString str = board->text();
    QMessageBox::information(NULL, "From clipboard", str);
}
```

```
}
```

槽函数也很简单。我们使用 `QApplication::clipboard()` 函数获得系统剪贴板对象。这个函数的返回值是 `QClipboard` 指针。我们可以从这个类的 API 中看到，通过 `setText()`，`setImage()` 或者 `setPixmap()` 函数可以将数据放置到剪贴板内，也就是通常所说的剪贴或者复制的操作；使用 `text()`，`image()` 或者 `pixmap()` 函数则可以从剪贴板获得数据，也就是粘贴。

另外值得说的是，通过上面的例子可以看出，`QTextEdit` 默认就支持 `Ctrl+C`，`Ctrl+V` 等快捷操作的。不仅如此，很多 Qt 的组件都提供了很方便的操作，因此我们需要从文档中获取具体的信息，从而避免自己重新去发明轮子。

`QClipboard` 提供的数据类型很少，如果需要，我们可以继承 `QMimeData` 类，通过调用 `setMimeData()` 函数让剪贴板能够支持我们自己的数据类型。具体实现我们已经在前面的章节中有过介绍，这里不再赘述。

在 X11 系统中，鼠标中键（一般是滚轮）可以支持剪贴操作。为了实现这一功能，我们需要向 `QClipboard::text()` 函数传递 `QClipboard::Selection` 参数。例如，我们在鼠标按键释放的事件中进行如下处理：

```
void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
{
    QClipboard *clipboard = QApplication::clipboard();
    if (event->button() == Qt::MidButton
        && clipboard->supportsSelection()) {
        QString text = clipboard->text(QClipboard::Selection);
        pasteText(text);
    }
}
```

这里的 `supportsSelection()` 函数在 X11 平台返回 `true`，其余平台都是返回 `false`。这样，我们便可以为 X11 平台提供额外的操作。

另外，`QClipboard` 提供了 `dataChanged()` 信号，以便监听剪贴板数据变化。

## 55 数据库操作

Qt 提供了 `QtSql` 模块来提供平台独立的基于 SQL 的数据库操作。这里我们所说的“平台独立”，既包括操作系统平台，有包括各个数据库平台。另外，我们强调了“基于 SQL”，因

为 NoSQL 数据库至今没有一个通用查询方法，所以不可能提供一种通用的 NoSQL 数据库的操作。Qt 的数据库操作还可以很方便的与 model/view 架构进行整合。通常来说，我们对数据库的操作更多地在于对数据库表的操作，而这正是 model/view 架构的长项。

Qt 使用 QSqlDatabase 表示一个数据库连接。更底层上，Qt 使用驱动（drivers）来与不同的数据库 API 进行交互。Qt 桌面版本提供了如下几种驱动：

驱动	数据库
QDB2	IBM DB2 (7.1 或更新版本)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity (ODBC) – Microsoft SQL Server 及其它兼容 ODBC 的数据库
QPSQL	PostgreSQL (7.3 或更新版本)
QSQLITE2	SQLite 2
QSQLITE	SQLite 3
QSYMSQL	针对 Symbian 平台的 SQLite 3
QTDS	Sybase Adaptive Server (自 Qt 4.7 起废除)

不过，由于受到协议的限制，Qt 开源版本并没有提供上面所有驱动的二进制版本，而仅仅以源代码的形式提供。通常，Qt 只默认搭载 QSqlite 驱动（这个驱动实际还包括 Sqlite 数据库，也就是说，如果需要使用 Sqlite 的话，只需要该驱动即可）。我们可以选择把这些驱动作为 Qt 的一部分进行编译，也可以当作插件编译。



如果习惯于使用 SQL 语句，我们可以选择 QSqlQuery 类；如果只需要使用高层次的数据库接口（不关心 SQL 语法），我们可以选择 QSqlTableModel 和 QSqlRelationalTableModel。本章我们介绍 QSqlQuery 类，在后面的章节则介绍 QSqlTableModel 和 QSqlRelationalTableModel。

在使用时，我们可以通过

```
QSqlDatabase::drivers();
```

找到系统中所有可用的数据库驱动的名字列表。我们只能使用出现在列表中的驱动。由于默认情况下，QtSql 是作为 Qt 的一个模块提供的。为了使用有关数据库的类，我们必须早 .pro 文件中添加这么一句：

```
QT += sql
```

这表示，我们的程序需要使用 Qt 的 core、gui 以及 sql 三个模块。注意，如果需要同时使用 Qt4 和 Qt5 编译程序，通常我们的 .pro 文件是这样的：

```
QT += core gui sql
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

这两句也很明确：Qt 需要加载 core、gui 和 sql 三个模块，如果主版本大于 4，则再添加 widgets 模块。

下面来看一个简单的函数：

```
bool connect(const QString &dbName)
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    // db.setHostName("host");
    // db.setDatabaseName("dbname");
    // db.setUserName("username");
    // db.setPassword("password");
    db.setDatabaseName(dbName);
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               db.lastError().text());
        return false;
    }
    return true;
}
```

我们使用 `connect()` 函数创建一个数据库连接。我们使用 `QSqlDatabase::addDatabase()` 静态函数完成这一请求，也就是创建了一个 `QSqlDatabase` 实例。注意，数据库连接使用自己的名字进行区分，而不是数据库的名字。例如，我们可以使用下面的语句：

```
QSqlDatabase db=QSqlDatabase::addDatabase("SQLITE",
QString("conn%1").arg(dbName));
```

此时，我们是使用 `addDatabase()` 函数的第二个参数来给这个数据库连接一个名字。在这个例子中，用于区分这个数据库连接的名字是 `QString("conn%1").arg(dbName)`，而不是“SQLITE”。这个参数是可选的，如果不指定，系统会给出一个默认的名字

`QSqlDatabase::defaultConnection`，此时，Qt 会创建一个默认的连接。如果你给出的名字与已存在的名字相同，新的连接会替换掉已有的连接。通过这种设计，我们可以为一个数据库建立多个连接。

我们这里使用的是 `sqlite` 数据库，只需要指定数据库名字即可。如果是数据库服务器，比如 `MySQL`，我们还需要指定主机名、端口号、用户名和密码，这些语句使用注释进行了简单的说明。

接下来我们调用了 `QSqlDatabase::open()` 函数，打开这个数据库连接。通过检查 `open()` 函数的返回值，我们可以判断数据库是不是正确打开。

QtSql 模块中的类大多具有 `lastError()` 函数，用于检查最新出现的错误。如果你发现数据库操作有任何问题，应该使用这个函数进行错误的检查。这一点我们也在上面的代码中进行了体现。当然，这只是最简单的实现，一般来说，更好的设计是，不要在数据库操作中混杂界面代码（并且将这个 `connect()` 函数放在一个专门的数据库操作类中）。

接下来我们可以在 `main()` 函数中使用这个 `connect()` 函数：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if (connect("demo.db")) {
        QSqlQuery query;
        if (!query.exec("CREATE TABLE student ("
                        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                        "name VARCHAR,"
                        "age INT))) {
            QMessageBox::critical(0, QObject::tr("Database Error"),
                                query.lastError().text());
            return 1;
        }
    } else {
```

```

        return 1;
    }
    return a.exec();
}

```

main()函数中，我们调用这个 connect()函数打开数据库。如果打开成功，我们通过一个 QSqlQuery 实例执行了 SQL 语句。同样，我们使用其 lastError()函数检查了执行结果是否正确。

注意这里的 QSqlQuery 实例的创建。我们并没有指定是为哪一个数据库连接创建查询对象，此时，系统会使用默认的连接，也就是使用没有第二个参数的 addDatabase()函数创建的那个连接（其实就是名字为 QSqlDatabase::defaultConnection 的默认连接）。如果没有这么一个连接，系统就会报错。也就是说，如果没有默认连接，我们在创建 QSqlQuery 对象时必须指明是哪一个 QSqlDatabase 对象，也就是 addDatabase()的返回值。

我们还可以通过使用 QSqlQuery::isActive()函数检查语句执行正确与否。如果 QSqlQuery 对象是活动的，该函数返回 true。所谓“活动”，就是指该对象成功执行了 exec()函数，但是还没有完成。如果需要设置为不活动的，可以使用 finish()或者 clear()函数，或者直接释放掉这个 QSqlQuery 对象。这里需要注意的是，如果存在一个活动的 SELECT 语句，某些数据库系统不能成功完成 connect()或者 rollback()函数的调用。此时，我们必须首先将活动的 SELECT 语句设置成不活动的。

创建过数据库表 student 之后，我们开始插入数据，然后将其独取出来：

```

if (connect("demo.db")) {
    QSqlQuery query;
    query.prepare("INSERT INTO student (name, age) VALUES (?, ?)");
    QVariantList names;
    names << "Tom" << "Jack" << "Jane" << "Jerry";
    query.addBindValue(names);
    QVariantList ages;
    ages << 20 << 23 << 22 << 25;
    query.addBindValue(ages);
    if (!query.execBatch()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               query.lastError().text());
    }
    query.finish();
    query.exec("SELECT name, age FROM student");
    while (query.next()) {
        QString name = query.value(0).toString();
        int age = query.value(1).toInt();
        qDebug() << name << ": " << age;
    }
}

```

```
    }  
} else {  
    return 1;  
}
```

依旧连接到我们创建的 `demo.db` 数据库。我们需要插入多条数据，此时可以使用 `QSqlQuery::exec()` 函数一条一条插入数据，但是这里我们选择了另外一种方法：批量执行。首先，我们使用 `QSqlQuery::prepare()` 函数对这条 SQL 语句进行预处理，问号 `?` 相当于占位符，预示着以后我们可以使用实际数据替换这些位置。简单说明一下，预处理是数据库提供的一种特性，它会将 SQL 语句进行编译，性能和安全性都要优于普通的 SQL 处理。在上面的代码中，我们使用一个字符串列表 `names` 替换掉第一个问号的位置，一个整型列表 `ages` 替换掉第二个问号的位置，利用 `QSqlQuery::addBindValue()` 我们将实际数据绑定到这个预处理的 SQL 语句上。需要注意的是，`names` 和 `ages` 这两个列表里面的数据需要一一对应。然后我们调用 `QSqlQuery::execBatch()` 批量执行 SQL，之后结束该对象。这样，插入操作便完成了。

另外说明一点，我们这里使用了 ODBC 风格的 `?` 占位符，同样，我们也可以使用 Oracle 风格的占位符：

```
query.prepare("INSERT INTO student (name, age) VALUES (:name, :age)");
```

此时，我们就需要使用

```
query.bindValue(":name", names);  
query.bindValue(":age", ages);
```

进行绑定。Oracle 风格的绑定最大的好处是，绑定的名字和值很清晰，与顺序无关。但是这里需要注意，`bindValue()` 函数只能绑定一个位置。比如

```
query.prepare("INSERT INTO test (name1, name2) VALUES (:name, :name)");  
// ...  
query.bindValue(":name", name);
```

只能绑定第一个 `:name` 占位符，不能绑定到第二个。

接下来我们依旧使用同一个查询对象执行一个 `SELECT` 语句。如果存在查询结果，`QSqlQuery::next()` 会返回 `true`，直到到达结果最末，返回 `false`，说明遍历结束。我们利用这一点，使用 `while` 循环即可遍历查询结果。使用 `QSqlQuery::value()` 函数即可按照 `SELECT` 语句的字段顺序获取到对应的数据库存储的数据。

对于数据库事务的操作，我们可以使用 `QSqlDatabase::transaction()` 开启事务，`QSqlDatabase::commit()` 或者 `QSqlDatabase::rollback()` 结束事务。使用 `QSqlDatabase::database()` 函数则可以根据名字获取所需要的数据库连接。

## 56 使用模型操作数据库

前一章我们使用 SQL 语句完成了对数据库的常规操作，包括简单的 `CREATE`、`SELECT` 等语句的使用。我们也提到过，Qt 不仅提供了这种使用 SQL 语句的方式，还提供了一种基于模型的更高级的处理方式。这种基于 `QSqlTableModel` 的模型处理更为高级，如果对 SQL 语句不熟悉，并且不需要很多复杂的查询，这种 `QSqlTableModel` 模型基本可以满足一般的需求。本章我们将介绍 `QSqlTableModel` 的一般使用，对比 SQL 语句完成对数据库的增删改查等的操作。值得注意的是，`QSqlTableModel` 并不一定非得结合 `QListView` 或 `QTableView` 使用，我们完全可以用其作一般性处理。

首先我们来看看如何使用 `QSqlTableModel` 进行 `SELECT` 操作：

```
if (connect("demo.db")) {
    QSqlTableModel model;
    model.setTable("student");
    model.setFilter("age > 20 and age < 25");
    if (model.select()) {
        for (int i = 0; i < model.rowCount(); ++i) {
            QSqlRecord record = model.record(i);
            QString name = record.value("name").toString();
            int age = record.value("age").toInt();
            qDebug() << name << ": " << age;
        }
    }
} else {
    return 1;
}
```

我们依旧使用了前一章的 `connect()` 函数。接下来我们创建了 `QSqlTableModel` 实例，使用 `setTable()` 函数设置所需要操作的表格；`setFilter()` 函数则是添加过滤器，也就是 `WHERE` 语句所需要的部分。例如上面代码中的操作实际相当于 SQL 语句

```
SELECT * FROM student WHERE age > 20 and age < 25
```

使用 `QSqlTableModel::select()` 函数进行操作，也就是执行了查询操作。如果查询成功，函数返回 `true`，由此判断是否发生了错误。如果没有错误，我们使用 `record()` 函数取出一行记录，该记录是以 `QSqlRecord` 的形式给出的，而 `QSqlRecord::value()` 则取出一个列的实际数据值。注意，由于 `QSqlTableModel` 没有提供 `const_iterator` 遍历器，因此不能使用 `foreach` 宏进行遍历。

另外需要注意，由于 `QSqlTableModel` 只是一种高级操作，肯定没有实际 `SQL` 语句方便。具体来说，我们使用 `QSqlTableModel` 只能进行 `SELECT *` 的查询，不能只查询其中某些列的数据。

下面一段代码则显示了如何使用 `QSqlTableModel` 进行插入操作：

```
QSqlTableModel model;
model.setTable("student");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 1), "Cheng");
model.setData(model.index(row, 2), 24);
model.submitAll();
```

插入也很简单：`model.insertRows(row, 1)`；说明我们想在索引 0 的位置插入 1 行新的数据。使用 `setData()` 函数则开始准备实际需要插入的数据。注意这里我们向 `row` 的第一个位置写入 `Cheng`（通过 `model.index(row, 1)`，回忆一下，我们把 `model` 当作一个二维表，这个坐标相当于第 `row` 行第 1 列），其余以此类推。最后，调用 `submitAll()` 函数提交所有修改。这里执行的操作可以用如下 `SQL` 表示：

```
INSERT INTO student (name, age) VALUES ('Cheng', 24)
```

当我们取出了已经存在的数据后，对其进行修改，然后重新写入数据库，即完成了一次更新操作：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        QSqlRecord record = model.record(0);
        record.setValue("age", 26);
        model.setRecord(0, record);
        model.submitAll();
    }
}
```

这段代码中，我们首先找到 `age = 25` 的记录，然后将 `age` 重新设置为 26，存入相同的位置（在这里都是索引 0 的位置），提交之后完成一次更新。当然，我们也可以类似其它模型一样的设置方式：`setData()`函数。具体代码片段如下：

```
if (model.select()) {
    if (model.rowCount() == 1) {
        model.setData(model.index(0, 2), 26);
        model.submitAll();
    }
}
```

注意我们的 `age` 列是第 3 列，索引值为 2，因为前面还有 `id` 和 `name` 两列。这里的更新操作则可以用如下 SQL 表示：

```
UPDATE student SET age = 26 WHERE age = 25
```

删除操作同更新类似：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        model.removeRows(0, 1);
        model.submitAll();
    }
}
```

如果使用 SQL 则是：

```
DELETE FROM student WHERE age = 25
```

当我们看到 `removeRows()`函数就应该想到：我们可以一次删除多行。事实也正是如此，这里不再赘述。

## 57 可视化显示数据库数据

前面我们用了两个章节介绍了 Qt 提供的两种操作数据库的方法。显然，使用 `QSqlQuery` 的方式更灵活，功能更强大，而使用 `QSqlTableModel` 则更简单，更方便与 `model/view` 结

合使用（数据库应用很大一部分就是以表格形式显示出来，这正是 model/view 的强项）。本章我们简单介绍使用 QSqlTableModel 显示数据的方法。当然，我们也可以选择使用 QSqlQuery 获取数据，然后交给 view 显示，而这需要自己给 model 提供数据。鉴于我们前面已经详细介绍过如何使用自定义 model 以及如何使用 QTableWidgetItem，所以我们这里不再详细说明这一方法。

我们还是使用前面一直在用的 student 表，直接来看代码：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if (connect("demo.db")) {
        QSqlTableModel *model = new QSqlTableModel;
        model->setTable("student");
        model->setSort(1, Qt::AscendingOrder);
        model->setHeaderData(1, Qt::Horizontal, "Name");
        model->setHeaderData(2, Qt::Horizontal, "Age");
        model->select();

        QTableView *view = new QTableView;
        view->setModel(model);
        view->setSelectionMode(QAbstractItemView::SingleSelection);
        view->setSelectionBehavior(QAbstractItemView::SelectRows);
        // view->setColumnHidden(0, true);
        view->resizeColumnsToContents();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);

        QHeaderView *header = view->horizontalHeader();
        header->setStretchLastSection(true);

        view->show();
    } else {
        return 1;
    }
    return a.exec();
}
```

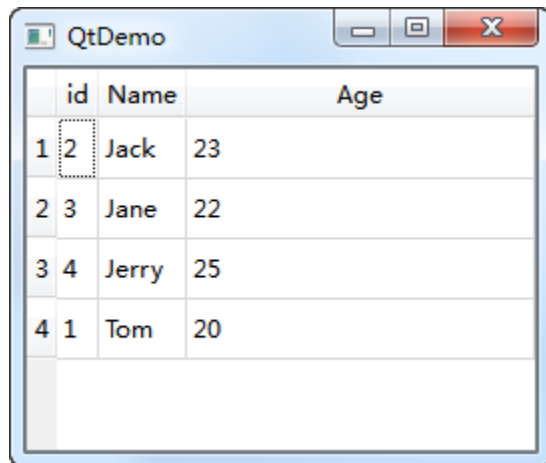
这里的 connect()函数还是我们前面使用过的，我们主要关注剩下的代码。

正如[前一章](#)的代码所示，我们在 main()函数中创建了 QSqlTableModel 对象，使用 student 表。student 表有三列：id, name 和 age，我们选择按照 name 排序，使用 setSort()函数达到这一目的。然后我们设置每一列的列头。这里我们只使用了后两列，第一列没有设置，所以依旧显示为列名 id。



在设置好 model 之后，我们又创建了 QTableView 对象作为视图。注意这里的设置：单行选择，按行选择。resizeColumnsToContents()说明每列宽度适配其内容；setEditTriggers()则禁用编辑功能。最后，我们设置最后一列要充满整个窗口。我们的代码中有一行注释，设置第一列不显示。由于我们使用了 QSqlTableModel 方式，不能按列查看，所以我们在视图级别上面做文章：将不想显示的列隐藏掉。

接下来运行代码即可看到效果：



如果看到代码中很多“魔术数字”，更好的方法是，使用一个枚举将这些魔术数字隐藏掉，这也是一种推荐的方式：

```
enum ColumnIndex
{
    Column_ID = 0,
    Column_Name = 1,
    Column_Age = 2
};

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

    if (connect("demo.db")) {
        QSqlTableModel *model = new QSqlTableModel;
        model->setTable("student");
        model->setSort(Column_Name, Qt::AscendingOrder);
        model->setHeaderData(Column_Name, Qt::Horizontal, "Name");
        model->setHeaderData(Column_Age, Qt::Horizontal, "Age");
        model->select();

        QTableView *view = new QTableView;
        view->setModel(model);
    }
}
```

```

        view->setSelectionMode(QAbstractItemView::SingleSelection);
        view->setSelectionBehavior(QAbstractItemView::SelectRows);
        view->setColumnHidden(Column_ID, true);
        view->resizeColumnsToContents();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);

        QHeaderView *header = view->horizontalHeader();
        header->setStretchLastSection(true);

        view->show();
    } else {
        return 1;
    }
    return a.exec();
}

```

## 58 编辑数据库外键

前面几章我们介绍了如何对数据库进行操作以及如何使用图形界面展示数据库数据。本章我们将介绍如何对数据库的数据进行编辑。当然，我们可以选择直接使用 SQL 语句进行更新，这一点同前面所说的 model/view 的编辑没有什么区别。除此之外，Qt 还为图形界面提供了更方便的展示并编辑的功能。

普通数据的编辑很简单，这里不再赘述。不过，我们通常会遇到多个表之间存在关联的情况。首先我们要提供一个 city 表：

```

CREATE TABLE city (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR);

INSERT INTO city (name) VALUES ('Beijing');
INSERT INTO city (name) VALUES ('Shanghai');
INSERT INTO city (name) VALUES ('Nanjing');
INSERT INTO city (name) VALUES ('Tianjin');
INSERT INTO city (name) VALUES ('Wuhan');
INSERT INTO city (name) VALUES ('Hangzhou');
INSERT INTO city (name) VALUES ('Suzhou');
INSERT INTO city (name) VALUES ('Guangzhou');

```

由于 city 表是一个参数表，所以我们直接将所需要的城市名称直接插入到表中。接下来我们创建 student 表，并且使用外键连接 city 表：

```
CREATE TABLE student (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    name VARCHAR,  
    age INTEGER,  
    address INTEGER,  
    FOREIGN KEY(address) REFERENCES city(id));
```

我们重新创建 student 表（如果你使用的 RDBMS 支持 ALTER TABLE 语句直接修改表结构，就不需要重新创建了；否则的话只能先删除旧的表，再创建新的表，例如 sqlite）。

这里需要注意一点，如果此时我们在 Qt 中直接使用

```
INSERT INTO student (name, age, address) VALUES ('Tom', 24, 100);
```

语句，尽管我们的 city 中没有 ID 为 100 的记录，但还是可以成功插入的。这是因为虽然 Qt 中的 sqlite 使用的是支持外键的 sqlite3，但 Qt 将外键屏蔽掉了。为了启用外键，我们需要首先使用 QSqlQuery 执行：

```
PRAGMA foreign_keys = ON;
```

然后就会发现这条语句不能成功插入了。接下来我们插入一些正常的数据：

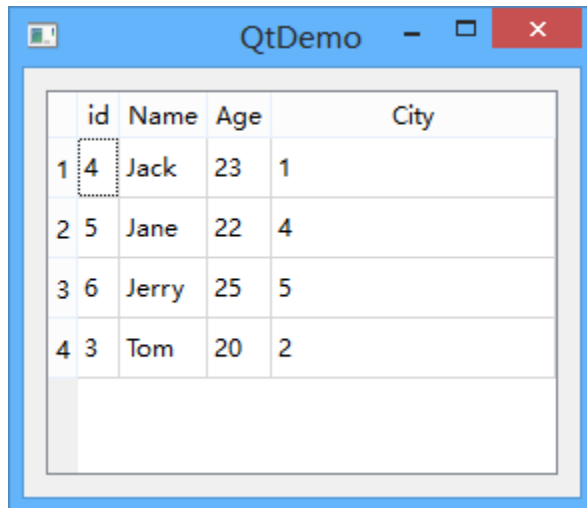
```
INSERT INTO student (name, age, address) VALUES ('Tom', 20, 2);  
INSERT INTO student (name, age, address) VALUES ('Jack', 23, 1);  
INSERT INTO student (name, age, address) VALUES ('Jane', 22, 4);  
INSERT INTO student (name, age, address) VALUES ('Jerry', 25, 5);
```

下面，我们使用 model/view 方式来显示数据：

```
QSqlTableModel *model = new QSqlTableModel(this);  
model->setTable("student");  
model->setSort(ColumnID_Name, Qt::AscendingOrder);  
model->setHeaderData(ColumnID_Name, Qt::Horizontal, "Name");  
model->setHeaderData(ColumnID_Age, Qt::Horizontal, "Age");  
model->setHeaderData(ColumnID_City, Qt::Horizontal, "City");  
model->select();  
  
QTableView *view = new QTableView(this);  
view->setModel(model);  
view->setSelectionMode(QAbstractItemView::SingleSelection);  
view->setSelectionBehavior(QAbstractItemView::SelectRows);  
view->resizeColumnsToContents();
```

```
QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

这段代码和我们前面见到的没有什么区别。我们可以将其补充完整后运行一下看看：



	id	Name	Age	City
1	4	Jack	23	1
2	5	Jane	22	4
3	6	Jerry	25	5
4	3	Tom	20	2

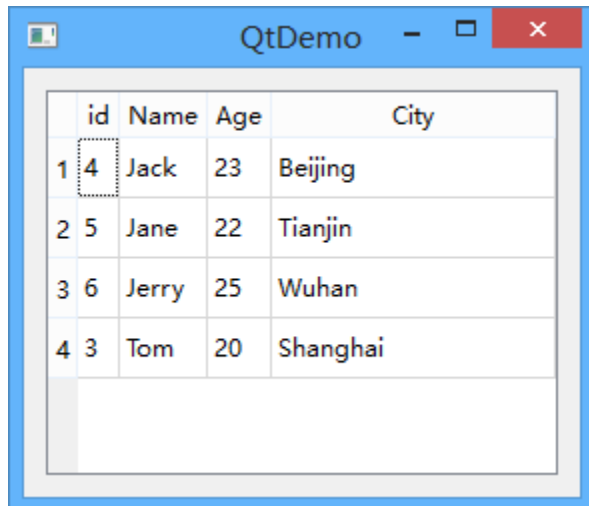
注意外键部分：City 一列仅显示出了我们保存的外键。如果我们使用 `QSqlQuery`，这些都不是问题，我们可以将外键信息放在一个 SQL 语句中 `SELECT` 出来。但是，我们不想使用 `QSqlQuery`，那么现在可以使用另外的一个模型：`QSqlRelationalTableModel`。`QSqlRelationalTableModel` 与 `QSqlTableModel` 十分类似，可以为一个数据库表提供可编辑的数据模型，同时带有外键的支持。下面我们修改一下我们的代码：

```
QSqlRelationalTableModel *model = new QSqlRelationalTableModel(this);
model->setTable("student");
model->setSort(ColumnID_Name, Qt::AscendingOrder);
model->setHeaderData(ColumnID_Name, Qt::Horizontal, "Name");
model->setHeaderData(ColumnID_Age, Qt::Horizontal, "Age");
model->setHeaderData(ColumnID_City, Qt::Horizontal, "City");
model->setRelation(ColumnID_City, QSqlRelation("city", "id", "name"));
model->select();

QTableView *view = new QTableView(this);
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
view->resizeColumnsToContents();
view->setItemDelegate(new QSqlRelationalDelegate(view));

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

这段代码同前面的几乎一样。我们首先创建一个 `QSqlRelationalTableModel` 对象。注意，这里我们有一个 `setRelation()` 函数的调用。该语句说明，我们将第 `ColumnID_City` 列作为外键，参照于 `city` 表的 `id` 字段，使用 `name` 进行显示。另外的 `setItemDelegate()` 语句则提供了一种用于编辑外键的方式。运行一下程序看看效果：



此时，我们的外键列已经显示为 `city` 表的 `name` 字段的实际值。同时在编辑时，系统会自动成为一个 `QComboBox` 供我们选择。当然，我们需要自己将选择的外键值保存到实际记录中，这部分我们前面已经有所了解。

## 59 使用流处理 XML

本章开始我们将了解到如何使用 Qt 处理 XML 格式的文档。

XML (eXtensible Markup Language) 是一种通用的文本格式，被广泛运用于数据交换和数据存储(虽然近年来 JSON 盛行，大有取代 XML 的趋势，但是对于一些已有系统和架构，比如 WebService，由于历史原因，仍旧会继续使用 XML)。XML 由 World Wide Web Consortium (W3C) 发布，作为 SHML (Standard Generalized Markup Language) 的一种轻量级方言。XML 语法类似于 HTML，与后者的主要区别在于 XML 的标签不是固定的，而是可扩展的；其语法也比 HTML 更为严格。遵循 XML 规范的 HTML 则被称为 XHTML (不过这一点有待商榷，感兴趣的话可以详见[这里](#))。

我们说过，XML 类似一种元语言，基于 XML 可以定义出很多新语言，比如 SVG (Scalable Vector Graphics) 和 MathML (Mathematical Markup Language)。SVG 是一种用于矢量绘图的描述性语言，Qt 专门提供了 QtSVG 对其进行解释；MathML 则是用于描述数学公式的语言，Qt Solutions 里面有一个 QtMmlWidget 模块专门对其进行解释。

另外一面，针对 XML 的通用处理，Qt4 提供了 QtXml 模块；针对 XML 文档的 Schema 验证以及 XPath、XQuery 和 XSLT，Qt4 和 Qt5 则提供了 QtXmlPatterns 模块。Qt 提供了三种读取 XML 文档的方法：

- **QXmlStreamReader**：一种快速的基于流的方式访问良格式 XML 文档，特别适合于实现一次解析器（所谓“一次解析器”，可以理解成我们只需读取文档一次，然后像一个遍历器从头到尾一次性处理 XML 文档，期间不会有反复的情况，也就是不会读完第一个标签，然后读第二个，读完第二个又返回去读第一个，这是不允许的）；
- **DOM (Document Object Model)**：将整个 XML 文档读入内存，构建成一个树结构，允许程序在树结构上向前向后移动导航，这是与另外两种方式最大的区别，也就是允许实现多次解析器（对应于前面所说的一次解析器）。DOM 方式带来的问题是需要一次性将整个 XML 文档读入内存，因此会占用很大内存；
- **SAX (Simple API for XML)**：提供大量虚函数，以事件的形式处理 XML 文档。这种解析办法主要是由于历史原因提出的，为了解决 DOM 的内存占用提出的（在现代计算机上，这个一般已经不是问题了）。

在 Qt4 中，这三种方式都位于 QtXml 模块中。Qt5 则将 QXmlStreamReader/QXmlStreamWriter 移动到 QtCore 中，QtXml 则标记为“不再维护”，这已经充分表明了 Qt 的官方意向。

至于生成 XML 文档，Qt 同样提供了三种方式：

- **QXmlStreamWriter**，与 QXmlStreamReader 相对应；
- **DOM 方式**，首先在内存中生成 DOM 树，然后将 DOM 树写入文件。不过，除非我们程序的数据结构中本来就维护着一个 DOM 树，否则，临时生成树再写入肯定比较麻烦；
- **纯手工生成 XML 文档**，显然，这是最复杂的一种方式。

使用 QXmlStreamReader 是 Qt 中最快最方便的读取 XML 的方法。因为 QXmlStreamReader 使用了递增式的解析器，适合于在整个 XML 文档中查找给定的标签、读入无法放入内存的大文件以及处理 XML 的自定义数据。

每次 QXmlStreamReader 的 readNext() 函数调用，解析器都会读取下一个元素，按照下表中展示的类型进行处理。我们通过表中所列的有关函数即可获得相应的数据值：

类型	示例	有关函数
StartDocument	-	documentVersion(),documentEncoding(),isStandalone
EndDocument	-	
StartElement	<item>	namespaceUri(),name(),attributes(),namespaceDeclar
EndElement	</item>	namespaceUri(),name()
Characters	AT&T	text(),isWhitespace(),isCDATA()
Comment	<!-- fix -->	text()
DTD	<!DOCTYPE ...>	text(),notationDeclarations(),entityDeclarations(),dtdN
EntityReference	&trade;	name(),text()
ProcessingInstruction	<?alert?>	processingInstructionTarget(),processingInstructionDa
Invalid	>&<!	error(), errorString()

考虑如下 XML 片段：

```
<doc>
  <quote>Einmal ist keinmal</quote>
</doc>
```

一次解析过后，我们通过 readNext()的遍历可以获得如下信息：

```
StartDocument
StartElement (name() == "doc")
StartElement (name() == "quote")
Characters (text() == "Einmal ist keinmal")
EndElement (name() == "quote")
```

```
EndElement (name() == "doc")
EndDocument
```

通过 `readNext()` 函数的循环调用，我们可以使用 `isStartElement()`、`isCharacters()` 这样的函数检查当前读取的类型，当然也可以直接使用 `state()` 函数。

下面我们看一个完整的例子。在这个例子中，我们读取一个 XML 文档，然后使用一个 `QTreeWidget` 显示出来。我们的 XML 文档如下：

```
<bookindex>
  <entry term="sidebearings">
    <page>10</page>
    <page>34-35</page>
    <page>307-308</page>
  </entry>
  <entry term="subtraction">
    <entry term="of pictures">
      <page>115</page>
      <page>244</page>
    </entry>
    <entry term="of vectors">
      <page>9</page>
    </entry>
  </entry>
</bookindex>
```

首先来看头文件：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

    bool readFile(const QString &fileName);
private:
    void readBookindexElement();
    void readEntryElement(QTreeWidgetItem *parent);
    void readPageElement(QTreeWidgetItem *parent);
    void skipUnknownElement();

    QTreeWidgetItem *treeWidget;
    QXmlStreamReader reader;
```



```
};
```

MainWindow 显然就是我们的主窗口，其构造函数也没有什么好说的：

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    setWindowTitle(tr("XML Reader"));

    treeWidget = new QTreeWidget(this);
    QStringList headers;
    headers << "Items" << "Pages";
    treeWidget->setHeaderLabels(headers);
    setCentralWidget(treeWidget);
}

MainWindow::~MainWindow()
{
}
```

接下来看几个处理 XML 文档的函数，这正是我们关注的要点：

```
bool MainWindow::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        QMessageBox::critical(this, tr("Error"),
                               tr("Cannot read file %1").arg(fileName));
        return false;
    }
    reader.setDevice(&file);
    while (!reader.atEnd()) {
        if (reader.isStartElement()) {
            if (reader.name() == "bookindex") {
                readBookindexElement();
            } else {
                reader.raiseError(tr("Not a valid book file"));
            }
        } else {
            reader.readNext();
        }
    }
    file.close();
    if (reader.hasError()) {
```

```

        QMessageBox::critical(this, tr("Error"),
                               tr("Failed to parse file %1").arg(fileName));

        return false;
    } else if (file.error() != QFile::NoError) {
        QMessageBox::critical(this, tr("Error"),
                               tr("Cannot read file %1").arg(fileName));

        return false;
    }
    return true;
}

```

readFile()函数用于打开给定文件。我们使用 QFile 打开文件，将其设置为 QXmlStreamReader 的设备。也就是说，此时 QXmlStreamReader 就可以从这个设备（QFile）中读取内容进行分析了。接下来便是一个 while 循环，只要没读到文件末尾，就要一直循环处理。首先判断是不是 StartElement，如果是的话，再去处理 bookindex 标签。注意，因为我们的根标签就是 bookindex，如果读到的不是 bookindex，说明标签不对，就要发起一个错误（raiseError()）。如果不是 StartElement（第一次进入循环的时候，由于没有事先调用 readNext()，所以会进入这个分支），则调用 readNext()。为什么这里要用 while 循环，XML 文档不是只有一个根标签吗？直接调用一次 readNext()函数不就好了？这是因为，XML 文档在根标签之前还有别的内容，比如声明，比如 DTD，我们不能确定第一个 readNext()之后就是根标签。正如我们提供的这个 XML 文档，首先是声明，其次才是根标签。如果你说，第二个不就是根标签吗？但是 XML 文档还允许嵌入 DTD，还可以写注释，这就不确定数目了，所以为了通用起见，我们必须用 while 循环判断。处理完之后就可以关闭文件，如果有错误则显示错误。

接下来看 readBookindexElement()函数：

```

void MainWindow::readBookindexElement()
{
    Q_ASSERT(reader.isStartElement() && reader.name() == "bookindex");
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(treeWidget->invisibleRootItem());
            } else {
                skipUnknownElement();
            }
        }
    }
}

```

```

        } else {
            reader.readNext();
        }
    }
}

```

注意第一行我们加了一个断言。意思是，如果在进入函数的时候，`reader` 不是 `StartElement` 状态，或者说标签不是 `bookindex`，就认为出错。然后继续调用 `readNext()`，获取下面的数据。后面还是 `while` 循环。如果是 `EndElement`，退出，如果又是 `StartElement`，说明是 `entry` 标签（注意我们的 XML 结构，`bookindex` 的子元素就是 `entry`），那么开始处理 `entry`，否则跳过。

那么下面来看 `readEntryElement()` 函数：

```

void MainWindow::readEntryElement(QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(parent);
    item->setText(0, reader.attributes().value("term").toString());

    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(item);
            } else if (reader.name() == "page") {
                readPageElement(item);
            } else {
                skipUnknownElement();
            }
        } else {
            reader.readNext();
        }
    }
}

```

这个函数接受一个 `QTreeWidgetItem` 指针，作为根节点。这个节点被当做这个 `entry` 标签在 `QTreeWidgetItem` 中的根节点。我们设置其名字是 `entry` 的 `term` 属性的值。然后继续读取下一个数据。同样使用 `while` 循环，如果是 `EndElement` 就继续读取；如果是 `StartElement`，

则按需调用 `readEntryElement()` 或者 `readPageElement()`。由于 `entry` 标签是可以嵌套的，所以这里有一个递归调用。如果既不是 `entry` 也不是 `page`，则跳过位置标签。

然后是 `readPageElement()` 函数：

```
void MainWindow::readPageElement(QTreeWidgetItem *parent)
{
    QString page = reader.readElementText();
    if (reader.isEndElement()) {
        reader.readNext();
    }

    QString allPages = parent->text(1);
    if (!allPages.isEmpty()) {
        allPages += ", ";
    }
    allPages += page;
    parent->setText(1, allPages);
}
```

由于 `page` 是叶子节点，没有子节点，所以不需要使用 `while` 循环读取。我们只是遍历了 `entry` 下所有的 `page` 标签，将其拼接成合适的字符串。

最后 `skipUnknownElement()` 函数：

```
void MainWindow::skipUnknownElement()
{
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

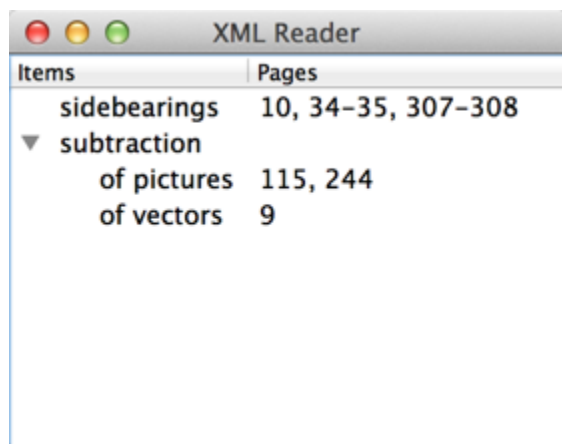
        if (reader.isStartElement()) {
            skipUnknownElement();
        } else {
            reader.readNext();
        }
    }
}
```

我们没办法确定到底要跳过多少位置标签，所以还是得用 `while` 循环读取，注意位置标签中所有子标签都是未知的，因此只要是 `StartElement`，都直接跳过。

好了，这是我们的全部程序。只要在 `main()` 函数中调用一下即可：

```
MainWindow w;  
w.readFile("books.xml");  
w.show();
```

然后就能看到运行结果：



The screenshot shows a window titled "XML Reader" with a table-like structure. The table has two columns: "Items" and "Pages". The data is as follows:

Items	Pages
sidebearings	10, 34-35, 307-308
▼ subtraction	
of pictures	115, 244
of vectors	9

值得一提的是，虽然我们的代码比较复杂，但是思路很清晰，一层一层地处理，这正是递归下降算法的有一个示例。我们曾在前面讲解[布尔表达式的树模型](#)章节使用过这个思想。

## 60 使用 DOM 处理 XML

DOM 是由 W3C 提出的一种处理 XML 文档的标准接口。Qt 实现了 DOM Level 2 级别的不验证读写 XML 文档的方法。

与[上一章](#)所说的流的方式不同，DOM 一次性读入整个 XML 文档，在内存中构造为一棵树（被称为 DOM 树）。我们能够在这棵树上进行导航，比如移动到下一节点或者返回上一节点，也可以对这棵树进行修改，或者是直接将这颗树保存为硬盘上的一个 XML 文件。考虑下面一个 XML 片段：

```
<doc>  
  <quote>Scio me nihil scire</quote>  
  <translation>I know that I know nothing</translation>  
</doc>
```

我们可以认为是如下一棵 DOM 树：

```
Document
|--Element(doc)
|   |--Element(quote)
|   |   |--Text("Scio me nihil scire")
|   |--Element(translation)
|       |--Text("I know that I know nothing")
```

上面所示的 DOM 树包含了不同类型的节点。例如，Element 类型的节点有一个开始标签和对应的一个结束标签。在开始标签和结束标签之间的内容作为这个 Element 节点的子节点。在 Qt 中，所有 DOM 节点的类型名字都以 QDom 开头，因此，QDomElement 就是 Element 节点，QDomText 就是 Text 节点。不同类型的节点则有不同类型的子节点。例如，Element 节点允许包含其它 Element 节点，也可以是其它类型，比如 EntityReference, Text, CDATASection, ProcessingInstruction 和 Comment。按照 W3C 的规定，我们有如下的包含规则：

```
[Document]
  <- [Element]
  <- DocumentType
  <- ProcessingInstrument
  <- Comment
[Attr]
  <- [EntityReference]
  <- Text
[DocumentFragment] | [Element] | [EntityReference] | [Entity]
  <- [Element]
  <- [EntityReference]
  <- Text
  <- CDATASection
  <- ProcessingInstrument
  <- Comment
```

上面表格中，带有 [] 的可以带有子节点，反之则不能。

下面我们还是以上一章所列出的 books.xml 这个文件来作示例。程序的目的还是一样的：用 QTreeWidget 来显示这个文件的结构。需要注意的是，由于我们选用 DOM 方式处理 XML，无论是 Qt4 还是 Qt5 都需要在 .pro 文件中添加这么一句：

```
QT += xml
```

头文件也是类似的：

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

    bool readFile(const QString &fileName);
private:
    void parseBookindexElement(const QDomElement &element);
    void parseEntryElement(const QDomElement &element, QTreeWidgetItem *parent);
    void parsePageElement(const QDomElement &element, QTreeWidgetItem *parent);
    QTreeWidgetItem *treeWidget;
};

```

MainWindow 的构造函数和析构函数和上一章是一样的，没有任何区别：

```

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("XML DOM Reader"));

    treeWidget = new QTreeWidgetItem(this);
    QStringList headers;
    headers << "Items" << "Pages";    treeWidget->setHeaderLabels(headers);
    setCentralWidget(treeWidget);
}

MainWindow::~~MainWindow()
{
}

```

readFile()函数则有了变化：

```

bool MainWindow::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        QMessageBox::critical(this, tr("Error"),
                                tr("Cannot read file %1").arg(fileName));
        return false;
    }

    QString errorStr;

```





两个函数的参数基本类似。第二个函数有五个参数，第一个是 `QByteArray`，也就是所读取的真实数据，由 `QIODevice` 即可获得这个数据，而 `QFile` 就是 `QIODevice` 的子类；第二个参数确定是否处理命名空间，如果设置为 `true`，处理器会自动设置标签的前缀之类，因为我们的 XML 文档没有命名空间，所以直接设置为 `false`；剩下的三个参数都是关于错误处理。后三个参数都是输出参数，我们传入一个指针，函数会设置指针的实际值，以便我们在外面获取并进行进一步处理。

当 `QDomDocument::setContent()` 函数调用完毕并且没有错误后，我们调用 `QDomDocument::documentElement()` 函数获得一个 `Document` 元素。如果这个 `Document` 元素标签是 `bookindex`，则继续向下处理，否则则报错。

```
void MainWindow::parseBookindexElement(const QDomElement &element)
{
    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry") {
            parseEntryElement(child.toElement(),
                              treeWidget->invisibleRootItem());
        }
        child = child.nextSibling();
    }
}
```

如果根标签正确，我们取第一个子标签，判断子标签不为空，也就是存在子标签，然后再判断其名字是不是 `entry`。如果是，说明我们正在处理 `entry` 标签，则调用其自己的处理函数；否则则取下一个标签（也就是 `nextSibling()` 的返回值）继续判断。注意我们使用这个 `if` 只选择 `entry` 标签进行处理，其它标签直接忽略掉。另外，`firstChild()` 和 `nextSibling()` 两个函数的返回值都是 `QDomNode`。这是所有节点类的基类。当我们需要对节点进行操作时，我们必须将其转换成正确的子类。这个例子中我们使用 `toElement()` 函数将 `QDomNode` 转换成 `QDomElement`。如果转换失败，返回值将是空的 `QDomElement` 类型，其 `tagName()` 返回空字符串，`if` 判断失败，其实也是符合我们的要求的。

```
void MainWindow::parseEntryElement(const QDomElement &element,
                                   QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(parent);
    item->setText(0, element.attribute("term"));

    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry") {
            parseEntryElement(child.toElement(), item);
        }
    }
}
```

```

        } else if (child.toElement().tagName() == "page") {
            parsePageElement(child.toElement(), item);
        }
        child = child.nextSibling();
    }
}

```

在 `parseEntryElement()` 函数中，我们创建了一个树组件的节点，其父节点是根节点或另外一个 `entry` 节点。接着我们又开始遍历这个 `entry` 标签的子标签。如果是 `entry` 标签，则递归调用自身，并且把当前节点作为父节点；否则则调用 `parsePageElement()` 函数。

```

void MainWindow::parsePageElement(const QDomElement &element,
                                   QTreeWidgetItem *parent)
{
    QString page = element.text();
    QString allPages = parent->text(1);
    if (!allPages.isEmpty()) {
        allPages += ", ";
    }
    allPages += page;
    parent->setText(1, allPages);
}

```

`parsePageElement()` 则比较简单，我们还是通过字符串拼接设置叶子节点的文本。这与上一章的步骤大致相同。

程序运行结果同上一章一模一样，这里不再贴出截图。

通过这个例子我们可以看到，使用 DOM 当时处理 XML 文档，除了一开始的 `setContent()` 函数，其余部分已经与原始文档没有关系了，也就是说，`setContent()` 函数的调用之后，已经在内存中构建好了一个完整的 DOM 树，我们可以在这棵树上面进行移动，比如取相邻节点（`nextSibling()`）。对比上一章流的方式，虽然我们早早关闭文件，但是我们始终使用的是 `readNext()` 向下移动，同时也不存在 `readPrevious()` 这样的函数。

## 61 使用 SAX 处理 XML

前面两章我们介绍了使用流和 DOM 的方式处理 XML 的相关内容，本章将介绍处理 XML 的最后一种方式：SAX。SAX 是一种读取 XML 文档的标准 API，同 DOM 类似，并不以语言为区别。Qt 的 SAX 类基于 SAX2 的 Java 实现，不过具有一些必要的名称上

的转换。相比 DOM，SAX 的实现更底层因而处理起来通常更快。但是，我们前面介绍的 `QXmlStreamReader` 类更偏向 Qt 风格的 API，并且比 SAX 处理器更快，所以，现在我们之所以使用 SAX API，更主要的是为了把 SAX API 引入 Qt。在我们通常的项目中，并不需要真的使用 SAX。

Qt 提供了 `QXmlSimpleReader` 类，提供基于 SAX 的 XML 处理。同前面所说的 DOM 方式类似，这个类也不会对 XML 文档进行有效性验证。`QXmlSimpleReader` 可以识别良格式的 XML 文档，支持 XML 命名空间。当这个处理器读取 XML 文档时，每当到达一个特定位置，都会调用一个用于处理解析事件的处理类。注意，这里所说的“事件”，不同于 Qt 提供的鼠标键盘事件，这仅是处理器在到达预定位置时发出的一种通知。例如，当处理器遇到一个标签的开始时，会发出“新开始一个标签”这个通知，也就是一个事件。我们可以从下面的例子中来理解这一点：

```
<doc>
  <quote>Gnothi seauton</quote>
</doc>
```

当读取这个 XML 文档时，处理器会依次发出下面的事件：

```
startDocument()
startElement("doc")
startElement("quote")
characters("Gnothi seauton")
endElement("quote")
endElement("doc")
endDocument()
```

每出现一个事件，都会有一个回调，这个回调函数就是在称为 `Handler` 的处理类中定义的。上面给出的事件都是在 `QXmlContentHandler` 接口中定义的。为简单起见，我们省略了一些函数。`QXmlContentHandler` 仅仅是众多处理接口中的一个，我们还有 `QXmlEntityResolver`，`QXmlDTDHandler`，`QXmlErrorHandler`，`QXmlDeclHandler` 以及 `QXmlLexicalHandler` 等。这些接口都是纯虚类，分别定义了不同类型的处理事件。对于大多数应用程序，`QXmlContentHandler` 和 `QXmlErrorHandler` 是最常用的两个。

为简化处理，Qt 提供了一个 `QXmlDefaultHandler`。这个类实现了以上所有的接口，每个函数都提供了一个空白实现。也就是说，当我们需要实现一个处理器时，只需要继承这个类，覆盖我们所关心的几个函数即可，无需将所有接口定义的函数都实现一遍。这种设计在 Qt 中并不常见，但是如果你熟悉 Java，就会感觉非常亲切。Java 中很多接口都是如此设计的。

使用 SAX API 与 `QXmlStreamReader` 或者 DOM API 之间最大的区别是，使用 SAX API 要求我们必须自己记录当前解析的状态。在另外两种实现中，这并不是必须的，我们可以使

用递归轻松地处理，但是 SAX API 则不允许（回忆下，SAX 仅允许一遍读取文档，递归意味着你可以先深入到底部再回来）。

下面我们使用 SAX 的方式重新解析前面两章所出现的示例程序。

```
class MainWindow : public QMainWindow, public QXmlDefaultHandler
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

    bool readFile(const QString &fileName);

protected:
    bool startElement(const QString &namespaceURI,
                     const QString &localName,
                     const QString &qName,
                     const QXmlAttributes &attributes);
    bool endElement(const QString &namespaceURI,
                   const QString &localName,
                   const QString &qName);
    bool characters(const QString &str);
    bool fatalError(const QXmlParseException &exception);
private:
    QTreeWidgetItem *treeWidget;
    QTreeWidgetItem *currentItem;
    QString currentText;
};
```

注意，我们的MainWindow不仅继承了QMainWindow，还继承了QXmlDefaultHandler。也就是说，主窗口自己就是XML的解析器。我们重写了startElement()，endElement()，characters()，fatalError()几个函数，其余函数不关心，所以使用了父类的默认实现。成员变量相比前面的例子也多出两个，为了记录当前解析的状态。

MainWindow的构造函数和析构函数同前面没有变化：

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("XML Reader"));

    treeWidget = new QTreeWidgetItem(this);
```

```

    QStringList headers;
    headers << "Items" << "Pages";
    treeWidget->setHeaderLabels(headers);
    setCentralWidget(treeWidget);
}

MainWindow::~MainWindow()
{
}

```

下面来看 readFile() 函数：

```

bool MainWindow::readFile(const QString &fileName)
{
    currentItem = 0;

    QFile file(fileName);
    QDomInputSource inputSource(&file);
    QDomSimpleReader reader;
    reader.setContentHandler(this);
    reader.setErrorHandler(this);
    return reader.parse(inputSource);
}

```

这个函数中，首先将成员变量清空，然后读取 XML 文档。注意我们使用了 QDomSimpleReader，将 ContentHandler 和 ErrorHandler 设置为自身。因为我们仅重写了 ContentHandler 和 ErrorHandler 的函数。如果我们还需要另外的处理，还需要继续设置其它的 handler。parse() 函数是 QDomSimpleReader 提供的函数，开始进行 XML 解析。

```

bool MainWindow::startElement(const QString & /*namespaceURI*/,
                              const QString & /*localName*/,
                              const QString &qName,
                              const QDomAttributes &attributes)
{
    if (qName == "entry") {
        currentItem = new QTreeWidgetItem(currentItem ?
            currentItem : treeWidget->invisibleRootItem());
        currentItem->setText(0, attributes.value("term"));
    } else if (qName == "page") {
        currentText.clear();
    }
    return true;
}

```

`startElement()`在读取到一个新的开始标签时被调用。这个函数有四个参数，我们这里主要关心第三和第四个参数：第三个参数是标签的名字（正式的名字是“限定名”，qualified name，因此形参是 `qName`）；第四个参数是属性列表。前两个参数主要用于带有命名空间的 XML 文档的处理，现在我们不关心命名空间。函数开始，如果是 `<entry>` 标签，我们创建一个新的 `QTreeWidgetItem`。如果这个标签是嵌套在另外的 `<entry>` 标签中的，`currentItem` 被定义为当前标签的子标签，否则则是根标签。我们使用 `setText()`函数设置第一列的值，同前面的章节类似。如果是 `<page>` 标签，我们将 `currentText` 清空，准备接下来的处理。最后，我们返回 `true`，告诉 SAX 继续处理文件。如果有任何错误，则可以返回 `false` 告诉 SAX 停止处理。此时，我们需要覆盖 `QXmlDefaultHandler` 的 `errorString()`函数来返回一个恰当的错误信息。

```
bool MainWindow::characters(const QString &str)
{
    currentText += str;
    return true;
}
```

注意下我们的 XML 文档。`characters()`仅在 `<page>` 标签中出现。因此我们在 `characters()` 中直接追加 `currentText`。

```
bool MainWindow::endElement(const QString & /*namespaceURI*/,
                           const QString & /*localName*/,
                           const QString &qName)
{
    if (qName == "entry") {
        currentItem = currentItem->parent();
    } else if (qName == "page") {
        if (currentItem) {
            QString allPages = currentItem->text(1);
            if (!allPages.isEmpty())
                allPages += ", ";
            allPages += currentText;
            currentItem->setText(1, allPages);
        }
    }
    return true;
}
```

`endElement()`在遇到结束标签时调用。和 `startElement()`类似，这个函数的第三个参数也是标签的名字。我们检查如果是 `</entry>`，则将 `currentItem` 指向其父节点。这保证了

currentItem 恢复到处理 <entry> 标签之前所指向的节点。如果是 </page>, 我们需要把新读到的 currentText 追加到第二列。

```
bool MainWindow::fatalError(const QDomParseException &exception)
{
    QMessageBox::critical(this,
                          tr("SAX Error"),
                          tr("Parse error at line %1, column %2:\n %3")
                          .arg(exception.lineNumber())
                          .arg(exception.columnNumber())
                          .arg(exception.message()));

    return false;
}
```

当遇到处理失败的时候, SAX 会回调 fatalError()函数。我们这里仅仅向用户显示出来哪里遇到了错误。如果你想看这个函数的运行, 可以将 XML 文档修改为不合法的形式。

我们程序的运行结果同前面还是一样的, 这里也不再赘述了。

## 62 保存 XML

前面几章我们讨论了读取 XML 文档的三种方法。虽然各有千秋, 但是 Qt 推荐的是使用 QDomStreamReader。与此同时, 许多应用程序不仅需要读取 XML, 还需要写入 XML。为此, Qt 同样提供了三种方法:

1. 使用 QDomStreamWriter;
2. 构造一个 DOM 树, 然后调用其 save()函数;
3. 使用 QString 手动生成 XML。

可以看出, 无论我们使用哪种读取方式, 这几种写入的方法都与此无关。这是因为 W3C 仅仅定义了如何处理 XML 文档, 并没有给出如何生成 XML 文档的标准方法 (尽管当我们使用 DOM 方式读取的时候, 依旧可以使用同样的 DOM 树写入)。

如同 Qt 所推荐的, 我们也推荐使用 QDomStreamWriter 生成 XML 文档。这个类帮助我们做了很多工作, 比如特殊字符的转义。接下来我们使用 QDomStreamWriter 将前面几章使用的 XML 文档生成出来:

```

QFile file("bookindex.xml");
if (!file.open(QFile::WriteOnly | QFile::Text)) {
    qDebug() << "Error: Cannot write file: "
        << qPrintable(file.errorString());
    return false;
}

QXmlStreamWriter xmlWriter(&file);
xmlWriter.setAutoFormatting(true);
xmlWriter.writeStartDocument();
xmlWriter.writeStartElement("bookindex");
xmlWriter.writeStartElement("entry");
xmlWriter.writeAttribute("term", "sidebearings");
xmlWriter.writeTextElement("page", "10");
xmlWriter.writeTextElement("page", "34-35");
xmlWriter.writeTextElement("page", "307-308");
xmlWriter.writeEndElement();
xmlWriter.writeStartElement("entry");
xmlWriter.writeAttribute("term", "subtraction");
xmlWriter.writeStartElement("entry");
xmlWriter.writeAttribute("term", "of pictures");
xmlWriter.writeTextElement("page", "115");
xmlWriter.writeTextElement("page", "224");
xmlWriter.writeEndElement();
xmlWriter.writeStartElement("entry");
xmlWriter.writeAttribute("term", "of vectors");
xmlWriter.writeTextElement("page", "9");
xmlWriter.writeEndElement();
xmlWriter.writeEndElement();
xmlWriter.writeEndDocument();
file.close();
if (file.error()) {
    qDebug() << "Error: Cannot write file: "
        << qPrintable(file.errorString());
    return false;
}
// ...

```

首先，我们以只写方式创建一个文件。然后基于该文件我们创建了 `QXmlStreamWriter` 对象。`setAutoFormatting()` 函数告诉 `QXmlStreamWriter` 要有格式输出，也就是会有标签的缩进。我们也可以使用 `QXmlStreamWriter::setAutoFormattingIndent()` 设置每个缩进所需要的空格数。接下来是一系列以 `write` 开始的函数。这些函数就是真正输出时需要用到的。注意这些函数以 `write` 开始，有 `Start` 和 `End` 两个对应的名字。正如其名字暗示那样，



一个用于写入开始标签，一个用于写入结束标签。`writeStartDocument()`开始进行 XML 文档的输出。这个函数会写下

```
<?xml version="1.0" encoding="UTF-8"?>
```

一行。与 `writeStartDocument()`相对应的是最后的 `writeEndDocument()`，告诉 `QXmlStreamWriter`，这个 XML 文档已经写完。下面我们拿出一段典型的代码：

```
xmlWriter.writeStartElement("entry");  
xmlWriter.writeAttribute("term", "of vectors");  
xmlWriter.writeTextElement("page", "9");  
xmlWriter.writeEndElement();
```

显然，这里我们首先写下一个 `entry` 的开始标签，然后给这个标签一个属性 `term`，属性值是 `of vectors`。`writeTextElement()`函数则会输出一个仅包含文本内容的标签。最后写入这个标签的关闭标签。这段代码的输出结果就是：

```
<entry term="of vectors">  
  <page>9</page>  
</entry>
```

其余部分与此类似，这里不再赘述。这样，我们就输出了一个与前面章节所使用的相同的 XML 文档：

```
<?xml version="1.0" encoding="UTF-8"?>  
<bookindex>  
  <entry term="sidebearings">  
    <page>10</page>  
    <page>34-35</page>  
    <page>307-308</page>  
  </entry>  
  <entry term="subtraction">  
    <entry term="of pictures">  
      <page>115</page>  
      <page>224</page>  
    </entry>  
    <entry term="of vectors">  
      <page>9</page>  
    </entry>  
  </entry>  
</bookindex>
```

尽管我们推荐使用 `QXmlStreamWriter` 生成 XML 文档,但是如果现在已经有了 DOM 树,显然直接调用 `QDomDocument::save()` 函数更为方便。在某些情况下,我们需要手动生成 XML 文档,比如通过 `QTextStream`:

```
//!!! Qt4
QTextStream out(&file);
out.setCodec("UTF-8");
out << "<doc>\n"
    << "    <quote>" << Qt::escape(quoteText) << "</quote>\n"
    << "    <translation>" << Qt::escape(translationText)
    << "</translation>\n"
    << "</doc>\n";

//!!! Qt5
QTextStream out(&file);
out.setCodec("UTF-8");
out << "<doc>\n"
    << "    <quote>" << quoteText.toHtmlEscaped() << "</quote>\n"
    << "    <translation>" << translationText.toHtmlEscaped()
    << "</translation>\n"
<< "</doc>\n";
```

这种办法是最原始的办法:我们直接除了字符串,把字符串拼接成 XML 文档。需要注意的是, `quoteText` 和 `translationText` 都需要转义,这是 XML 规范里面要求的,需要将文本中的 `<`, `>` 以及 `&` 进行转义。不过,转义函数在 Qt4 中是 `Qt::escape()`,而 Qt5 中则是 `QString::toHtmlEscaped()`,需要按需使用。

## 63 使用 QJson 处理 JSON

XML 曾经是各种应用的配置和传输的首选方式。但是现在 XML 遇到了一个强劲的对手:JSON。我们可以在[这里](#)看到有关 JSON 的语法。总体来说,JSON 的数据比 XML 更紧凑,在传输效率上也要优于 XML。不过 JSON 数据的层次化表达不及 XML,至少不如 XML 那样突出。不过这并不会阻止 JSON 的广泛应用。

一个典型的 JSON 文档可以像下面的例子:

```
{
  "encoding" : "UTF-8",
  "plug-ins" : [
    "python",
```

```

        "c++",
        "ruby"
    ],
    "indent" : { "length" : 3, "use_space" : true }
}

```

JSON 的全称是 JavaScript Object Notation，与 JavaScript 密不可分。熟悉 JavaScript 的童鞋马上就会发现，JSON 的语法就是 JavaScript 对象声明的语法。JSON 文档其实就是一个 JavaScript 对象，因而也称为 JSON 对象，以大括号作为起止符，其实质是不排序的键值对，其中键要求是 string 类型，值可以是任意类型。比如上面的示例，键 encoding 的值是字符串 UTF-8；键 plug-ins 的值是一个数组类型，在 JSON 中，数组以中括号表示，这个数组是一个字符串列表，分别有 python、c++ 和 ruby 三个对象；键 indent 的值是一个对象，这个对象有两个属性，length = 3，use\_space = true。

对于 JSON 的解析，我们可以使用 [QJson](#) 这个第三方库。QJson 可以将 JSON 数据转换为 QVariant 对象，将 JSON 数组转换成 QVariantList 对象，将 JSON 对象转换成 QVariantMap 对象。我们在[这里](#)使用 git clone 出 QJson 的整个代码。注意 QJson 没有提供链接库的 pro 文件，因此我们只需要将所有源代码文件添加到我们的项目即可（如同这些文件是我们自己写的一样）。接下来就可以使用 QJson 读取 JSON 内容：

```

#include "parser.h"
//////////
QJson::Parser parser;
bool ok;

QString json("{\"encoding\" : \"UTF-8\",",
    "\"plug-ins\" : [",
    "\"python\",",
    "\"c++\",",
    "\"ruby\"",
    "],",
    "\"indent\" : { \"length\" : 3, \"use_space\" : true }",
    "}");

QVariantMap result = parser.parse(json.toUtf8(), &ok).toMap();
if (!ok) {
    qFatal("An error occurred during parsing");
    exit (1);
}

qDebug() << "encoding:" << result["encoding"].toString();
qDebug() << "plugins:";

```

```
foreach (QVariant plugin, result["plug-ins"].toList()) {
    qDebug() << "\t-" << plugin.toString();
}

QVariantMap nestedMap = result["indent"].toMap();
qDebug() << "length:" << nestedMap["length"].toInt();
qDebug() << "use_space:" << nestedMap["use_space"].toBool();
```

将 JSON 对象转换成 QVariant 对象很简单，基本只需要下面几行：

```
// 1. 创建 QJson::Parser 对象
QJson::Parser parser;

bool ok;
// 2. 将 JSON 对象保存在一个对象 json 中，进行数据转换
QVariant result = parser.parse(json, &ok);
```

`QJson::Parser::parse()` 函数接受两个参数，第一个参数是 JSON 对象，可以是 `QIODevice` \*或者是 `QByteArray`；第二个参数是转换成功与否，如果成功则被设置为 `true`。函数返回转换后的 `QVariant` 对象。注意我们转换后的对象其实是一个 `QVariantMap` 类型，可以像 `QMap` 一样使用重载的 `[]` 获取键所对应的值。另外，由于 `result["plug-ins"]` 是一个 `QVariantList` 对象（因为是由 JSON 数组返回的），因而可以调用其 `toList()` 函数，通过遍历输出每一个值。

如果需要将 `QVariant` 生成 JSON 对象，我们则使用 `QJson::Serializer` 对象。例如：

```
QVariantList people;

QVariantMap bob;
bob.insert("Name", "Bob");
bob.insert("Phonenumber", 123);

QVariantMap alice;
alice.insert("Name", "Alice");
alice.insert("Phonenumber", 321);

people << bob << alice;

QJson::Serializer serializer;
bool ok;
QByteArray json = serializer.serialize(people, &ok);

if (ok) {
```

```

        qDebug() << json;
    } else {
        qCritical() << "Something went wrong:" << serializer.errorMessage();
    }
}

```

QJson::Serializer 和前面的 QJson::Parser 的用法相似，只需要调用 QJson::Serializer::serialize()即可将 QVariant 类型的数据转换为 JSON 格式。其返回值是 QByteArray 类型，可以用于很多其它场合。

上面是 QJson 的主要使用方法。其实 QJson 还提供了另外一个类 QObjectHelper，用于 QVariant 和 QObject 之间的转换。注意我们上面所说的 QJson 的转换需要的是 QVariant 类型的数据，无论是转换到 JSON 还是从 JSON 转换而来。但是通常我们在应用程序中使用的是 QObject 及其子类。QObjectHelper 提供了一个工具函数，完成 QVariant 和 QObject 之间的转换。例如我们有下面的类：

```

class Person : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString name READ name WRITE setName)
    Q_PROPERTY(int phoneNumber READ phoneNumber WRITE setPhoneNumber)
    Q_PROPERTY(Gender gender READ gender WRITE setGender)
    Q_PROPERTY(QDate brithday READ brithday WRITE setBrithday)
    Q_ENUMS(Gender)

public:
    Person(QObject* parent = 0);
    ~Person();

    QString name() const;
    void setName(const QString& name);

    int phoneNumber() const;
    void setPhoneNumber(const int phoneNumber);

    enum Gender {Male, Female};
    void setGender(Gender gender);
    Gender gender() const;

    QDate brithday() const;
    void setBrithday(const QDate& dob);

private:

```

```

QString m_name;
int m_phoneNumber;
Gender m_gender;
QDate m_dob;
};

```

那么，我们可以使用下面的代码将 `Person` 类进行 JSON 序列化：

```

Person person;
person.setName("Flavio");
person.setPhoneNumber(123456);
person.setGender(Person::Male);
person.setDob(QDate(1982, 7, 12));

QVariantMap variant = QObjectHelper::qobject2qvariant(&person);
QJson::Serializer serializer;
QDebug() << serializer.serialize( variant);

```

以及：

```

QJson::Parser parser;
QVariant variant = parser.parse(json);
Person person;
QObjectHelper::qvariant2qobject(variant.toMap(), &person);

```

进行反序列化。

## 64 使用 QJsonDocument 处理 JSON

上一章我们了解了如何使用 `QJson` 处理 JSON 文档。`QJson` 是一个基于 Qt 的第三方库，适用于 Qt4 和 Qt5 两个版本。不过，如果你的应用仅仅需要考虑兼容 Qt5，其实已经有了内置的处理函数。Qt5 新增加了处理 JSON 的类，与 XML 类库类似，均以 `QJson` 开头，在 `QtCore` 模块中，不需要额外引入其它模块。Qt5 新增加六个相关类：

`QJsonArray`                      封装 JSON 数组

`QJsonDocument`                  读写 JSON 文档

QJsonObject	封装 JSON 对象
QJsonObject::iterator	用于遍历 QJsonObject 的 STL 风格的非 const 遍历器
QJsonParseError	报告 JSON 处理过程中出现的错误
QJsonValue	封装 JSON 值

我们还是使用前一章的 JSON 文档，这次换用 QJsonDocument 来解析。注意，QJsonDocument 要求使用 Qt5，本章中所有代码都必须在 Qt5 环境下进行编译运行。

```
QString json("{\"
    \"encoding\" : \"UTF-8\",
    \"plug-ins\" : [
        \"python\",
        \"c++\",
        \"ruby\"
    ],
    \"indent\" : { \"length\" : 3, \"use_space\" : true }
}");
QJsonParseError error;
QJsonDocument jsonDocument = QJsonDocument::fromJson(json.toUtf8(), &error);
if (error.error == QJsonParseError::NoError) {
    if (jsonDocument.isObject()) {
        QVariantMap result = jsonDocument.toVariant().toMap();
        qDebug() << "encoding:" << result["encoding"].toString();
        qDebug() << "plugins:";

        foreach (QVariant plugin, result["plug-ins"].toList()) {
            qDebug() << "\t-" << plugin.toString();
        }

        QVariantMap nestedMap = result["indent"].toMap();
        qDebug() << "length:" << nestedMap["length"].toInt();
        qDebug() << "use_space:" << nestedMap["use_space"].toBool();
    }
} else {
    qFatal(error.errorString().toUtf8().constData());
    exit(1);
}
```

这段代码与前面的几乎相同。QJsonDocument::fromJson()可以由 QByteArray 对象构造一个 QJsonDocument 对象，用于我们的读写操作。这里我们传入一个 QJsonParseError 对象的指针作为第二个参数，用于获取解析的结果。如果 QJsonParseError::error()的返回值为 QJsonParseError::NoError，说明一切正常，则继续以 QVariant 的格式进行解析（由于我们知道这是一个 JSON 对象，因此只判断了 isObject()。当处理未知的 JSON 时，或许应当将所有情况都考虑一边，包括 isObject()、isArray()以及 isEmpty()）。

也就是说，如果需要使用 QJsonDocument 处理 JSON 文档，我们只需要使用下面的代码模板：

```
// 1. 创建 QJsonParseError 对象，用来获取解析结果
QJsonParseError error;
// 2. 使用静态函数获取 QJsonDocument 对象
QJsonDocument jsonDocument = QJsonDocument::fromJson(json.toUtf8(), &error);
// 3. 根据解析结果进行处理
if (error.error == QJsonParseError::NoError) {
    if (!(jsonDocument.isNull() || jsonDocument.isEmpty())) {
        if (jsonDocument.isObject()) {
            // ...
        } else if (jsonDocument.isArray()) {
            // ...
        }
    }
} else {
    // 检查错误类型
}
```

将 QVariant 对象生成 JSON 文档也很简单：

```
QVariantList people;

QVariantMap bob;
bob.insert("Name", "Bob");
bob.insert("Phonenumber", 123);

QVariantMap alice;
alice.insert("Name", "Alice");
alice.insert("Phonenumber", 321);

people << bob << alice;

QJsonDocument jsonDocument = QJsonDocument::fromVariant(people);
if (!jsonDocument.isNull()) {
```



```
qDebug() << jsonDocument.toJson();  
}
```

这里我们仍然使用的是 `QJsonDocument`，只不过这次我们需要使用 `QJsonDocument::fromVariant()` 函数获取 `QJsonDocument` 对象。`QJsonDocument` 也可以以二进制格式读取对象，比如 `QJsonDocument::fromBinaryData()` 和 `QJsonDocument::fromRawData()` 函数。当我们成功获取到 `QJsonDocument` 对象之后，可以使用 `toJson()` 生成 JSON 文档。

以上介绍了当我们有一个 JSON 文档时，如何使用 `QJsonDocument` 进行处理。如果我们没有 JSON 文档，那么我们可以使用 `QJsonDocument` 的 `setArray()` 和 `setObject()` 函数动态设置该对象，然后再生成对应的 JSON 格式文档。不过这部分需求比较罕见，因为我们直接可以从 `QVariant` 值类型获取。

Qt5 提供的 JSON 类库直接支持[隐式数据共享](#)，因此我们不需要为复制的效率担心。

## 65 访问网络（1）

现在的应用程序很少有纯粹单机的。大部分为了各种目的都需要联网操作。为此，Qt 提供了自己的网络访问库，方便我们对网络资源进行访问。本章我们将介绍如何使用 Qt 进行最基本的网络访问。

Qt 进行网络访问的类是 `QNetworkAccessManager`，这是一个名字相当长的类，不过使用起来并不像它的名字一样复杂。为了使用网络相关的类，你需要在 `pro` 文件中添加 `QT += network`。

`QNetworkAccessManager` 类允许应用程序发送网络请求以及接受服务器的响应。事实上，Qt 的整个访问网络 API 都是围绕着这个类进行的。`QNetworkAccessManager` 保存发送的请求的最基本的配置信息，包含了代理和缓存的设置。最好的是，这个 API 本身就是异步设计，这意味着我们不需要自己为其开启线程，以防止界面被锁死（这里我们可以简单了解下，Qt 的界面活动是在一个主线程中进行。网络访问是一个相当耗时的操作，如果整个网络访问的过程以同步的形式在主线程进行，则当网络访问没有返回时，主线程会被阻塞，界面就会被锁死，不能执行任何响应，甚至包括一个代表响应进度的滚动条都会被卡死在那里。这种设计显然是不友好的。）。异步的设计避免了这一系列的问题，但是却要求我们使用更多的代码来监听返回。这类似于我们前面提到的 `QDialog::exec()` 和 `QDialog::show()` 之间的区别。`QNetworkAccessManager` 是使用信号槽来达到这一目的的。

一个应用程序仅需要一个 `QNetworkAccessManager` 类的实例。所以，虽然 `QNetworkAccessManager` 本身没有被设计为单例，但是我们应该把它当做单例使用。一旦一个 `QNetworkAccessManager` 实例创建完毕，我们就可以使用它发送网络请求。这些请求都返回 `QNetworkReply` 对象作为响应。这个对象一般会包含有服务器响应的数据。

下面我们用一个例子来看如何使用 `QNetworkAccessManager` 进行网络访问。这个例子不仅会介绍 `QNetworkAccessManager` 的使用，还将设计到一些关于程序设计的细节。

我们的程序是一个简单的天气预报的程序，使用 `OpenWeatherMap` 的 API 获取数据。我们可以在[这里](#)找到其 API 的具体介绍。

我们前面说过，一般一个应用使用一个 `QNetworkAccessManager` 就可以满足需要，因此我们自己封装一个 `NetWorker` 类，并把这个类作为单例。注意，我们的代码使用了 `Qt5` 进行编译，因此如果你需要将代码使用 `Qt4` 编译，请自行修改相关部分。

```
// !!! Qt5
#ifndef NETWORKER_H
#define NETWORKER_H

#include <QObject>

class QNetworkReply;

class NetWorker : public QObject
{
    Q_OBJECT
public:
    static NetWorker * instance();
    ~NetWorker();

    void get(const QString &url);
signals:
    void finished(QNetworkReply *reply);
private:
    class Private;
    friend class Private;
    Private *d;

    explicit NetWorker(QObject *parent = 0);
    NetWorker(const NetWorker &) Q_DECL_EQ_DELETE;
    NetWorker& operator=(NetWorker rhs) Q_DECL_EQ_DELETE;
};
```

```
#endif // NETWORKER_H
```

NetWorker 是一个单例类，因此它有一个 `instance()` 函数用来获得这唯一的实例。作为单例模式，要求构造函数、拷贝构造函数和赋值运算符都是私有的，因此我们将这三个函数都放在 `private` 块中。注意我们增加了一个 `Q_DECL_EQ_DELETE` 宏。这个宏是 Qt5 新增加的，意思是将它所修饰的函数声明为 `deleted`（这是 C++11 的新特性）。如果编译器支持 `= delete` 语法，则这个宏将会展开为 `= delete`，否则则展开为空。我们的 NetWorker 只有一个 `get` 函数，顾名思义，这个函数会执行 HTTP GET 操作；一个信号 `finished()`，会在获取到服务器响应后发出。`private` 块中还有三行关于 `Private` 的代码：

```
class Private;
friend class Private;
Private *d;
```

这里声明了一个 NetWorker 的内部类，然后声明了这个内部类的 `d` 指针。`d` 指针是 C++ 程序常用的一种设计模式。它的存在与 C++ 程序的编译有关。在 C++ 中，保持二进制兼容性非常重要。如果你能够保持二进制兼容，则当以后升级库代码时，用户不需要重新编译自己的程序即可直接运行（如果你使用 Qt5.0 编译了一个程序，这个程序不需要重新编译就可以运行在 Qt5.1 下，这就是二进制兼容；如果不需要修改源代码，但是必须重新编译才能运行，则是源代码兼容；如果必须修改源代码并且再经过编译，例如从 Qt4 升级到 Qt5，则称二者是不兼容的）。保持二进制兼容的很重要的一个原则是不要随意增加、删除成员变量。因为这会导致类成员的寻址偏移量错误，从而破坏二进制兼容。为了避免这个问题，我们将一个类的所有私有变量全部放进一个单独的辅助类中，而在需要使用这些数据的类值提供一个这个辅助类的指针。注意，由于我们的辅助类是私有的，用户不能使用它，所以针对这个辅助类的修改不会影响到外部类，从而保证了二进制兼容。关于二进制兼容的问题，我们会在以后的文章中更详细的说明，这里仅作此简单介绍。

下面来看 NetWorker 的实现。

```
class NetWorker::Private
{
public:
    Private(NetWorker *q) :
        manager(new QNetworkAccessManager(q))
    {}

    QNetworkAccessManager *manager;
};
```

`Private` 是 NetWorker 的内部类，扮演者前面我们所说的那个辅助类的角色。NetWorker::Private 类主要有一个成员变量 `QNetworkAccessManager *`，把

QNetworkAccessManager 封装起来。NetWorker::Private 需要其被辅助的类 NetWorker 的指针，目的是作为 QNetworkAccessManager 的 parent，以便 NetWorker 析构时能够自动将 QNetworkAccessManager 析构。当然，我们也可以通过将 NetWorker::Private 声明为 QObject 的子类来达到这一目的。

```
NetWorker *NetWorker::instance()
{
    static NetWorker netWorker;
    return &netWorker;
}
```

instance()函数很简单，我们声明了一个 static 变量，将其指针返回。这是 C++ 单例模式的最简单写法，由于 C++ 标准要求类的构造函数不能被打断，因此这样做也是线程安全的。

```
NetWorker::NetWorker(QObject *parent) :
    QObject(parent),
    d(new NetWorker::Private(this))
{
    connect(d->manager, &QNetworkAccessManager::finished,
            this, &NetWorker::finished);
}

NetWorker::~NetWorker()
{
    delete d;
    d = 0;
}
```

构造函数参数列表我们将 d 指针进行赋值。构造函数内容很简单，我们将 QNetworkAccessManager 的 finished()信号进行转发。也就是说，当 QNetworkAccessManager 发出 finished()信号时，NetWorker 同样会发出自己的 finished()信号。析构函数将 d 指针删除。由于 NetWorker::Private 是在堆上创建的，并且没有继承 QObject，所以我们必须手动调用 delete 运算符。

```
void NetWorker::get(const QString &url)
{
    d->manager->get(QNetworkRequest(QUrl(url)));
}
```

get()函数也很简单，直接将用户提供的 URL 字符串提供给底层的 QNetworkAccessManager，实际上是将操作委托给底层 QNetworkAccessManager 进行。

现在我们将 `QNetworkAccessManager` 进行了简单的封装。下一章我们开始针对 `OpenWeatherMap` 的 API 进行编码。

## 66 访问网络（2）

上一章我们了解了 `NetWorker` 类的简单实现。不仅如此，我们还提到了几个 C++ 开发时常用的设计模式。这些在接下来得代码中依然会用到。

现在我们先来研究下 `OpenWeatherMap` 的相关 API。之所以选择 `OpenWeatherMap`，主要是因为这个网站提供了简洁的 API 接口，非常适合示例程序，并且其开发也不需要额外申请 App ID。`OpenWeatherMap` 的 API 可以选择返回 JSON 或者 XML，这里我们选择使用 JSON 格式。在进行查询时，`OpenWeatherMap` 支持使用城市名、地理经纬度以及城市 ID，为简单起见，我们选择使用城市名。我们先来看一个例子：

[http://api.openweathermap.org/data/2.5/weather?q=Beijing,cn&mode=json&units=metric&lang=zh\\_cn](http://api.openweathermap.org/data/2.5/weather?q=Beijing,cn&mode=json&units=metric&lang=zh_cn)。下面是这个链接的参数分析：

参数名字	传入值	说明
q	Beijing,cn	查询中国北京的天气
mode	json	返回格式为 JSON
units	metric	返回单位为公制
lang	zh_cn	返回语言为中文

点击链接，服务器返回一个 JSON 字符串（此时你应该能够使用浏览器看到这个字符串）：

```
{"coord":{"lon":116.397232,"lat":39.907501},"sys":{"country":"CN","sunrise":1381530122,"sunset":1381570774},"weather":[{"id":800,"main":"Clear","description":"晴","icon":"01d"}],"base":"gdpstations","main":{"temp":20,"pressure":1016,"humidity":34,"temp_min":20,"temp_max":20},"wind":{"speed":2,"deg":50},"clouds":{"all":0},"dt":1381566600,"id":1816670,"name":"Beijing","cod":200}
```

我们从[这里](#)找到 JSON 各个字段的含义。现在我们关心的是：时间（dt）；气温（temp）；气压（pressure）；湿度（humidity）和天气状况（weather）。基于此，我们设计了 WeatherInfo 类，用于封装服务器返回的信息：

```
class WeatherDetail
{
public:
    WeatherDetail();
    ~WeatherDetail();

    QString desc() const;
    void setDesc(const QString &desc);

    QString icon() const;
    void setIcon(const QString &icon);

private:
    class Private;
    friend class Private;
    Private *d;
};

class WeatherInfo
{
public:
    WeatherInfo();
    ~WeatherInfo();

    QString cityName() const;
    void setCityName(const QString &cityName);

    quint32 id() const;
    void setId(quint32 id);

    QDateTime dateTime() const;
    void setDateTime(const QDateTime &dateTime);

    float temperature() const;
    void setTemperature(float temperature);

    float humidity() const;
    void setHumidity(float humidity);

    float pressure() const;
```

```

void setPressure(float pressure);

QList<WeatherDetail *> details() const;
void setDetails(const QList<WeatherDetail *> details);

private:
    class Private;
    friend class Private;
    Private *d;
};

QDebug operator <<(QDebug dbg, const WeatherDetail &w);
QDebug operator <<(QDebug dbg, const WeatherInfo &w);

```

WeatherInfo 和 WeatherDetail 两个类相互合作存储我们所需要的数据。由于是数据类，所以只有单纯的 setter 和 getter 函数，这里不再把源代码写出来。值得说明的是最后两个全局函数：

```

QDebug operator <<(QDebug dbg, const WeatherDetail &w);
QDebug operator <<(QDebug dbg, const WeatherInfo &w);

```

我们重写了<<运算符，以便能够使用类似 qDebug() << weatherInfo;这样的语句进行调试。它的实现是这样的：

```

QDebug operator <<(QDebug dbg, const WeatherDetail &w)
{
    dbg.nospace() << "("
        << "Description: " << w.desc() << "; "
        << "Icon: " << w.icon()
        << ")";
    return dbg.space();
}

QDebug operator <<(QDebug dbg, const WeatherInfo &w)
{
    dbg.nospace() << "("
        << "id: " << w.id() << "; "
        << "City name: " << w.cityName() << "; "
        << "Date time: " <<
w.dateTime().toString(Qt::DefaultLocaleLongDate) << ": " << endl
        << "Temperature: " << w.temperature() << ", "
        << "Pressure: " << w.pressure() << ", "
        << "Humidity: " << w.humidity() << endl
        << "Details: [";
}

```

```

        foreach (WeatherDetail *detail, w.details()) {
            dbg.nospace() << "( Description: " << detail->desc() << ", "
                << "Icon: " << detail->icon() << ")", ";
        }
        dbg.nospace() << "] )";
        return dbg.space();
    }
}

```

这两个函数虽然比较长，但是很简单，这里不再赘述。

下面我们来看主窗口：

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    class Private;
    friend class Private;
    Private *d;
};

```

正如前面所说的，这里依然使用了 `d` 指针模式。头文件没有什么可说的。  
`MainWindow::Private` 的实现依旧简单：

```

class MainWindow::Private
{
public:
    Private()
    {
        netWorker = NetWorker::instance();
    }

    void fetchWeather(const QString &cityName) const
    {
        netWorker->get(QString("http://api.openweathermap.org/data/2.5/weather?q=%1&mode=json&units=metric&lang=zh_cn").arg(cityName));
    }

    NetWorker *netWorker;
};

```



```
};
```

我们将 `MainWindow` 所需要的 `NetWorker` 作为 `MainWindow::Private` 的一个成员变量。`MainWindow::Private` 提供了一个 `fetchWeather()` 函数。由于 `NetWorker` 提供的函数都是相当底层的, 为了提供业务级别的处理, 我们将这样的函数封装在 `MainWindow::Private` 中。当然, 你也可以在 `NetWorker` 中直接提供类似的函数, 这取决于你的系统分层设计。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent),
      d(new MainWindow::Private)
{
    QComboBox *cityList = new QComboBox(this);
    cityList->addItem(tr("Beijing"), QLatin1String("Beijing,cn"));
    cityList->addItem(tr("Shanghai"), QLatin1String("Shanghai,cn"));
    cityList->addItem(tr("Nanjing"), QLatin1String("Nanjing,cn"));
    QLabel *cityLabel = new QLabel(tr("City: "), this);
    QPushButton *refreshButton = new QPushButton(tr("Refresh"), this);
    QHBoxLayout *cityListLayout = new QHBoxLayout;
    cityListLayout->setDirection(QBoxLayout::LeftToRight);
    cityListLayout->addWidget(cityLabel);
    cityListLayout->addWidget(cityList);
    cityListLayout->addWidget(refreshButton);

    QVBoxLayout *weatherLayout = new QVBoxLayout;
    weatherLayout->setDirection(QBoxLayout::TopToBottom);
    QLabel *cityNameLabel = new QLabel(this);
    weatherLayout->addWidget(cityNameLabel);
    QLabel *dateTimeLabel = new QLabel(this);
    weatherLayout->addWidget(dateTimeLabel);

    QWidget *mainWidget = new QWidget(this);
    QVBoxLayout *mainLayout = new QVBoxLayout(mainWidget);
    mainLayout->addLayout(cityListLayout);
    mainLayout->addLayout(weatherLayout);
    setCentralWidget(mainWidget);
    resize(320, 120);
    setWindowTitle(tr("Weather"));

    connect(d->netWorker, &NetWorker::finished, [=] (QNetworkReply *reply) {
        qDebug() << reply;
        QJsonParseError error;
        QJsonDocument jsonDocument = QJsonDocument::fromJson(reply->readAll(),
&error);
        if (error.error == QJsonParseError::NoError) {
```

```

        if (!(jsonDocument.isNull() || jsonDocument.isEmpty()) &&
jsonDocument.isObject()) {
            QVariantMap data = jsonDocument.toVariant().toMap();
            WeatherInfo weather;
            weather.setCityName(data[QLatin1String("name")].toString());
            QDateTime dateTime;
            dateTime.setTime_t(data[QLatin1String("dt")].toLongLong());
            weather.setDateTime(dateTime);
            QVariantMap main = data[QLatin1String("main")].toMap();
            weather.setTemperature(main[QLatin1String("temp")].toFloat());
            weather.setPressure(main[QLatin1String("pressure")].toFloat());
            weather.setHumidity(main[QLatin1String("humidity")].toFloat());
            QVariantList detailList =
data[QLatin1String("weather")].toList();
            QList<WeatherDetail *> details;
            foreach (QVariant w, detailList) {
                QVariantMap wm = w.toMap();
                WeatherDetail *detail = new WeatherDetail;

detail->setDesc(wm[QLatin1String("description")].toString());
                detail->setIcon(wm[QLatin1String("icon")].toString());
                details.append(detail);
            }
            weather.setDetails(details);

            cityNameLabel->setText(weather.cityName());

dateTimeLabel->setText(weather.dateTime().toString(Qt::DefaultLocaleLongDate)
);
        }
        } else {
            QMessageBox::critical(this, tr("Error"), error.errorString());
        }
        reply->deleteLater();
    });
    connect(refreshButton, &QPushButton::clicked, [=] () {

d->fetchWeather(cityList->itemData(cityList->currentIndex()).toString());
    });
}

MainWindow::~MainWindow()
{
    delete d;
}

```

```
d = 0;
}
```

接下来我们来看 MainWindow 的构造函数和析构函数。构造函数虽然很长但是并不复杂，主要是对界面的构建。我们这里略过这些界面的代码，直接看两个信号槽的连接。

```
connect(d->netWorker, &NetWorker::finished, [=] (QNetworkReply *reply) {
    QJsonParseError error;
    QJsonDocument jsonDocument = QJsonDocument::fromJson(reply->readAll(),
&error);
    if (error.error == QJsonParseError::NoError) {
        if (!(jsonDocument.isNull() || jsonDocument.isEmpty()) &&
jsonDocument.isObject()) {
            QVariantMap data = jsonDocument.toVariant().toMap();
            WeatherInfo weather;
            weather.setCityName(data[QLatin1String("name")].toString());
            QDateTime dateTime;
            dateTime.setTime_t(data[QLatin1String("dt")].toLongLong());
            weather.setDateTime(dateTime);
            QVariantMap main = data[QLatin1String("main")].toMap();
            weather.setTemperature(main[QLatin1String("temp")].toFloat());
            weather.setPressure(main[QLatin1String("pressure")].toFloat());
            weather.setHumidity(main[QLatin1String("humidity")].toFloat());
            QVariantList detailList =
data[QLatin1String("weather")].toList();
            QList<WeatherDetail *> details;
            foreach (QVariant w, detailList) {
                QVariantMap wm = w.toMap();
                WeatherDetail *detail = new WeatherDetail;

detail->setDesc(wm[QLatin1String("description")].toString());
                detail->setIcon(wm[QLatin1String("icon")].toString());
                details.append(detail);
            }
            weather.setDetails(details);

            cityNameLabel->setText(weather.cityName());

dateTimeLabel->setText(weather.dateTime().toString(Qt::DefaultLocaleLongDate)
);
        }
    } else {
        QMessageBox::critical(this, tr("Error"), error.errorString());
    }
}
```

```

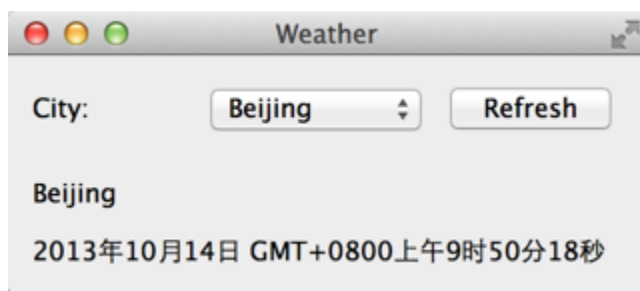
        reply->deleteLater();
    });
    connect(refreshButton, &QPushButton::clicked, [=] () {

d->fetchWeather(cityList->itemData(cityList->currentIndex()).toString());
    });

```

由于使用了 Qt5，我们选择新的连接语法。第一个 `connect()` 函数中，我们按照 API 文档中描述的那样对服务器返回的 JSON 字符串进行解析，然后将数据填充到一个 `WeatherInfo` 的对象。然后操作界面的两个控件显示数据。值得注意的是函数的最后一行，`reply->deleteLater()`。当网络请求结束时，`delete` 服务器返回的 `QNetworkReply` 对象是用户的责任。用户需要选择一个恰当的时机进行 `delete` 操作。但是，我们不能直接在 `finished()` 信号对应的槽函数中调用 `delete` 运算符。相反，我们需要使用 `deleteLater()` 函数，正如前面代码中显示的那样。第二个槽函数则相对简单，仅仅是重新获取新的数据。

选择我们可以运行下程序了：



## 67 访问网络（3）

[上一章](#)我们了解了如何使用我们设计的 `NetWorker` 类实现我们所需要的网络操作。本章我们将继续完善前面介绍的天气程序。

注意到我们在 `WeatherDetail` 类中有一个 `icon` 属性。到现在为止我们还没有用到这个属性。下面我们考虑如何修改我们的程序。

通过查看 [OpenWeatherMap](#) 的相关 API 我们可以发现，当我们查询天气时会附带这么一个 `icon` 属性。这个属性其实是网站上的一个天气的图片。还是以上一章我们见到的 JSON 返回值为例：

```

{"coord":{"lon":116.397232,"lat":39.907501},"sys":{"country":"CN","sunrise":1381530122,"sunset":1381570774},"weather":[{"id":800,"main":"Clear","description":"晴","icon":"01d"}],"base":"gdps

```

```
stations", "main": {"temp": 20, "pressure": 1016, "humidity": 34, "temp_min": 20, "temp_max": 20}, "wind": {"speed": 2, "deg": 50}, "clouds": {"all": 0}, "dt": 1381566600, "id": 1816670, "name": "Beijing", "cod": 200}
```

注意到其中的 `icon:01d` 这个键值对。通过文档我们知道，`01d` 实际对应于网站上的一张图片：<http://openweathermap.org/img/w/01d.png>。这就是我们的思路：当我们获取到实际天气数据时，我们根据这个返回值从网站获取到图片，然后显示到我们的程序中。

回忆下我们的 `NetWorker` 类的实现。我们将其 `finished()` 信号与我们自己实现的槽函数连接起来，其代码大致相当于：

```
connect(d->netWorker, &NetWorker::finished, [=] (QNetworkReply *reply) {  
    ...  
});
```

我们将 `finished()` 信号与一个 `Lambda` 表达式连接起来，其参数就是服务器的响应值。这样一来就会有一个问题：我们实际是有两次网络请求，第一次是向服务器请求当前的天气情况，第二次是根据第一次响应值去请求一张图片。每次网络请求完成时都会发出 `finished()` 信号，这就要求我们在槽函数中区分当前到底是哪一个请求的返回。所以，我们需要修改下有关网络请求的代码：

```
class NetWorker : public QObject  
{  
    ...  
    QNetworkReply *get(const QString &url);  
    ...  
};  
  
...  
  
QNetworkReply * NetWorker::get(const QString &url)  
{  
    return d->manager->get(QNetworkRequest(QUrl(url)));  
}
```

首先要修改的是 `NetWorker` 类的 `get()` 函数。我们要让这个函数返回一个 `QNetworkReply` \*变量。这个对象其实是 `QNetworkAccessManager::get()` 函数的返回值，我们简单地将其返回出来。接下来要修改的是 `MainWindow::Private` 的代码：

```
class MainWindow::Private  
{  
public:  
    Private(MainWindow *q) :
```

```

        mainWindow(q)
    {
        netWorker = NetWorker::instance();
    }

    void fetchWeather(const QString &cityName)
    {
        QNetworkReply *reply =
netWorker->get(QString("http://api.openweathermap.org/data/2.5/weather?q=%1&m
ode=json&units=metric&lang=zh_cn").arg(cityName));
        replyMap.insert(reply, FetchWeatherInfo);
    }

    void fetchIcon(const QString &iconName)
    {
        QNetworkReply *reply =
netWorker->get(QString("http://openweathermap.org/img/w/%1.png").arg(iconName
));
        replyMap.insert(reply, FetchWeatherIcon);
    }

    NetWorker *netWorker;
    MainWindow *mainWindow;
    QMap<QNetworkReply *, RemoteRequest> replyMap;
};

```

我们的请求是在 `MainWindow::Private` 私有类中完成的，为此添加了一个 `QMap` 属性。注意我们在原有的 `fetchWeather()` 和新增的 `fetchIcon()` 函数中都将 `NetWorker::get()` 函数的返回值保存下来。`RemoteRequest` 只是一个枚举，定义如下：

```

enum RemoteRequest {
    FetchWeatherInfo,
    FetchWeatherIcon
};

```

显然，我们的代码能够清晰地描述出我们的网络请求的返回结果对应于哪一种操作：`fetchWeather()` 中 `NetWorker::get()` 函数的返回值对应于 `FetchWeatherInfo` 操作，而 `fetchIcon()` 中 `NetWorker::get()` 函数的返回值则对应于 `FetchWeatherIcon` 操作。我们不需要区分每种操作的具体 URL 地址，因为我们的响应依照操作的不同而不同，与 URL 无关。

下面我们只看槽函数的改变：

```

connect(d->netWorker, &NetWorker::finished, [=] (QNetworkReply *reply) {
    RemoteRequest request = d->replyMap.value(reply);
    switch (request) {
    case FetchWeatherInfo:
    {
        QJsonParseError error;
        QJsonDocument jsonDocument = QJsonDocument::fromJson(reply->readAll(),
&error);
        if (error.error == QJsonParseError::NoError) {
            if (!(jsonDocument.isNull() || jsonDocument.isEmpty()) &&
jsonDocument.isObject()) {
                QVariantMap data = jsonDocument.toVariant().toMap();
                WeatherInfo weather;
                weather.setCityName(data[QLatin1String("name")].toString());
                QDateTime dateTime;
                dateTime.setTime_t(data[QLatin1String("dt")].toLongLong());
                weather.setDateTime(dateTime);
                QVariantMap main = data[QLatin1String("main")].toMap();
                weather.setTemperature(main[QLatin1String("temp")].toFloat());
                weather.setPressure(main[QLatin1String("pressure")].toFloat());
                weather.setHumidity(main[QLatin1String("humidity")].toFloat());
                QVariantList detailList =
data[QLatin1String("weather")].toList();
                QList details;
                foreach (QVariant w, detailList) {
                    QVariantMap wm = w.toMap();
                    WeatherDetail *detail = new WeatherDetail;

detail->setDesc(wm[QLatin1String("description")].toString());
                    detail->setIcon(wm[QLatin1String("icon")].toString());
                    details.append(detail);

                    QHBoxLayout *weatherDetailLayout = new QHBoxLayout;
                    weatherDetailLayout->setDirection(QBoxLayout::LeftToRight);
                    weatherDetailLayout->addWidget(new QLabel(detail->desc(),
this));

                    weatherDetailLayout->addWidget(new QLabel(this));
                    weatherLayout->addLayout(weatherDetailLayout);

                    d->fetchIcon(detail->icon());
                }
                weather.setDetails(details);

                cityNameLabel->setText(weather.cityName());
            }
        }
    }
}

```

```

dateTimeLabel->setText(weather.dateTime().toString(Qt::DefaultLocaleLongDate)
);
    }
    } else {
        QMessageBox::critical(this, tr("Error"), error.errorString());
    }
    break;
}
case FetchWeatherIcon:
{
    QHBoxLayout *weatherDetailLayout = (HBoxLayout
*)weatherLayout->itemAt(2)->layout();
    QLabel *iconLabel = (QLabel *)weatherDetailLayout->itemAt(1)->widget();
    QPixmap pixmap;
    pixmap.loadFromData(reply->readAll());
    iconLabel->setPixmap(pixmap);
    break;
}
}

reply->deleteLater();
});

```

槽函数最大的变化是，我们依照 `MainWindow::Private` 中保存的对应值，找到这个 `reply` 对应的操作类型，然后使用一个 `switch` 语句进行区分。注意我们在 `FetchWeatherInfo` 操作的 `foreach` 循环中增加了对 `WeatherDetail` 数据的显示。在末尾使用一个 `d->fetchIcon(detail->icon())` 语句从网络获取对应的图片。在 `FetchWeatherIcon` 操作中，我们根据 `HBoxLayout` 的 `itemAt()` 函数找到先前添加的用于显示图片的 `label`，然后读取 `reply` 的数据值，以二进制的形式加载图片。虽然代码很长，有些函数我们也是第一次见到，但是整体思路很简单。下面来看最终的运行结果：





我们今天介绍了这种技术，用于区分一个程序中的多次网络请求（这在一个应用中是经常遇到的）。当然这只是其中一种解决方案，如果你有更好的解决方案，也请留言告诉豆子~

## 68 访问网络（4）

前面几章我们了解了如何使用 `QNetworkAccessManager` 访问网络。在此基础上，我们已经实现了一个简单的查看天气的程序。在这个程序中，我们使用 `QNetworkAccessManager` 进行网络的访问，从一个网络 API 获取某个城市的当前天气状况。

如果你仔细观察就会发现，即便我们没有添加任何相关代码，`QNetworkAccessManager` 的网络访问并不会阻塞 GUI 界面。也就是说，即便是在进行网络访问的时候，我们的界面还是可以响应的。相比之下，如果你对 Java 熟悉，就会了解到，在 Java 中，进行 Socket 通讯时，界面默认是阻塞的，当程序进行网络访问操作时，界面不能对我们的操作做出任何响应。由此可以看出，`QNetworkAccessManager` 的网络访问默认就是异步的、非阻塞的。这样的实现固然很好，也符合大多数程序的应用情形：我们当然希望程序界面能够始终对用户操作做出响应。不过，在某些情况下，我们还是希望会有一些同步的网络操作。典型的是登录操作。在登录时，我们必须等待网络返回结果，才能让界面做出响应：是验证成功进入系统，还是验证失败做出提示？这就是本章的主要内容：如何使用 `QNetworkAccessManager` 进行同步网络访问。

当我们重新运行先前编译好的程序，可以看看这样一个操作：由于我们的界面是不阻塞的，那么当我们第一次点击了 `Refresh` 按钮之后，马上切换城市再点击一次 `Refresh` 按钮，就会看到第一次的返回结果一闪而过。这是因为第一次网络请求尚未完成时，用户又发送了一次请求，Qt 会将两次请求的返回结果顺序显示。这样处理结果可能会出现与预期不一致的情况（比如第一次请求响应由于某种原因异常缓慢，第二次却很快，此时第二次结果会比第一次先到，那么很明显，当第一次结果返回时，第二次的结果就会被覆盖掉。我们假设认为用户需要第二次的返回，那么就会出现异常）。

要解决这种情况，我们可以在有网络请求时将界面锁死，不允许用户进行更多的操作（更好的方法是仅仅锁住某些按钮，而不是整个界面。不过这里我们以锁住整个界面为例）。我们的解决方案很简单：当 `QNetworkAccessManager` 发出请求之后，我们进入一个新的事件循环，将操作进行阻塞。我们的代码示例如下：

```
void fetchWeather(const QString &cityName)
{
    QEventLoop eventLoop;
    connect(netWorker, &NetWorker::finished,
           &eventLoop, &QEventLoop::quit);
```

```

    QNetworkReply *reply =
netWorker->get(QString("http://api.openweathermap.org/data/2.5/weather?q=%1&m
ode=json&units=metric&lang=zh_cn").arg(cityName));
    replyMap.insert(reply, FetchWeatherInfo);
    eventLoop.exec();
}

```

注意，我们在函数中创建了一个 `QEventLoop` 实例，将其 `quit()` 与 `NetWorker::finished()` 信号连接起来。当 `NetWorker::finished()` 信号发出时，`QEventLoop::quit()` 就会被调用。在 `NetWorker::get()` 执行之后，调用 `QEventLoop::exec()` 函数开始事件循环。此时界面就是被阻塞。

现在我们只是提供了一种很简单的思路。当然这并不是最好的思路：程序界面直接被阻塞，用户获得不了任何提示，会误以为程序死掉。更好的做法是做一个恰当的提示，不过这已经超出我们本章的范畴。更重要的是，这种思路并不完美。**如果你的程序是控制台程序（没有 GUI 界面），或者是某些特殊的情况下，会造出死锁！** 控制台程序中发送死锁的原因在于在非 GUI 程序中另外启动事件循环会将主线程阻塞，`QNetworkAccessManager` 的所有信号都不会收到。“某些特殊的情况”，我们会在后面有关线程的章节详细解释。不过，要完美解决这个问题，我们必须使用另外的线程。[这里](#)有一个通用的解决方案，感兴趣的童鞋可以详细了解下。

## 69 进程

进程是操作系统的基础之一。一个进程可以认为是一个正在执行的程序。我们可以把进程当做计算机运行时的一个基础单位。关于进程的讨论已经超出了本章的范畴，现在我们假定你是了解这个概念的。

在 Qt 中，我们使用 `QProcess` 来表示一个进程。这个类可以允许我们的应用程序开启一个新的外部程序，并且与这个程序进行通讯。下面我们用一个非常简单的例子开始我们本章有关进程的阐述。

```

//!!! Qt5
QString program = "C:/Windows/System32/cmd.exe";
QStringList arguments;
arguments << "/c" << "dir" << "C:\\";
QProcess *cmdProcess = new QProcess; cmdProcess->start(program, arguments);
QObject::connect(cmdProcess, &QProcess::readyRead, [=] () {
    QTextCodec *codec = QTextCodec::codecForName("GBK");
    QString dir = codec->toUnicode(cmdProcess->readAll());
}

```

```
qDebug() << dir;
});
```

这是一段 Qt5 的程序，并且仅能运行于 Windows 平台。简单来说，这段程序通过 Qt 开启了一个新的进程，这个进程相当于执行了下面的命令：

```
C:\Windows\System32\cmd.exe /c dir C:\
```

注意，我们可以在上面的程序中找到这个命令的每一个字符。事实上，我们可以把一个进程看做执行了一段命令（在 Windows 平台就是控制台命令；在 Linux 平台（包括 Unix）则是执行一个普通的命令，比如 ls）。我们的程序相当于执行了 dir 命令，其参数是 C:\，这是由 arguments 数组决定的（至于为什么我们需要将 dir 命令作为参数传递给 cmd.exe，这是由于 Windows 平台的规定。在 Windows 中，dir 命令并不是一个独立的可执行程序，而是通过 cmd.exe 进行解释；这与 ls 在 Linux 中的地位不同，在 Linux 中，ls 就是一个可执行程序。因此如果你需要在 Linux 中执行 ls，那么 program 的值应该就是 ls）。

上面程序的运行结果类似于：

驱动器 C 中的卷是 SYSTEM

卷的序列号是 EA62-24AB

C:\ 的目录

2013/05/05	20:41		1,024 .rnd
2013/08/22	23:22	<DIR>	PerfLogs
2013/10/18	07:32	<DIR>	Program Files
2013/10/30	12:36	<DIR>	Program Files (x86)
2013/10/31	20:30		12,906 shared.log
2013/10/18	07:33	<DIR>	Users
2013/11/06	21:41	<DIR>	Windows
		2 个文件	13,930 字节
		5 个目录	22,723,440,640 可用字节

上面的输出会根据不同机器有所不同。豆子是在 Windows 8.1 64 位机器上测试的。

为了开启进程，我们将外部程序名字（program）和程序启动参数（arguments）作为参数传给 QProcess::start() 函数。当然，你也可以使用 setProgram() 和 setArguments() 进行设置。此时，QProcess 进入 Starting 状态；当程序开始执行之后，QProcess 进入 Running 状态，并且发出 started() 信号。当进程退出时，QProcess 进入 NotRunning 状态（也是初始

状态），并且发出 `finished()` 信号。`finished()` 信号以参数的形式提供进程的退出代码和退出状态。如果发送错误，`QProcess` 会发出 `error()` 信号

`QProcess` 允许你将一个进程当做一个顺序访问的 I/O 设备。我们可以使用 `write()` 函数将数据提供给进程的标准输入；使用 `read()`、`readLine()` 或者 `getChar()` 函数获取其标准输出。由于 `QProcess` 继承自 `QIODevice`，因此 `QProcess` 也可以作为 `QXmlReader` 的输入或者直接使用 `QNetworkAccessManager` 将其生成的数据上传到网络。

进程通常有两个预定义的通道：标准输出通道（`stdout`）和标准错误通道（`stderr`）。前者就是常规控制台的输出，后者则是由进程输出的错误信息。这两个通道都是独立的数据流，我们可以通过使用 `setReadChannel()` 函数来切换这两个通道。当进程的当前通道可用时，`QProcess` 会发出 `readReady()` 信号。当有了新的标准输出数据时，`QProcess` 会发出 `readyReadStandardOutput()` 信号；当有了新的标准错误数据时，则会发出 `readyReadStandardError()` 信号。我们前面的示例程序就是使用了 `readReady()` 信号。注意，由于我们是运行在 Windows 平台，Windows 控制台的默认编码是 GBK，为了避免出现乱码，我们必须设置文本的编码方式。

通道的术语可能会引起误会。注意，进程的输出通道对应着 `QProcess` 的 **读** 通道，进程的输入通道对应着 `QProcess` 的 **写** 通道。这是因为我们使用 `QProcess` “读取” 进程的输出，而我们针对 `QProcess` 的“写入”则成为进程的输入。`QProcess` 还可以合并标准输出和标准错误通道，使用 `setProcessChannelMode()` 函数设置 `MergedChannels` 即可实现。

另外，`QProcess` 还允许我们使用 `setEnvironment()` 为进程设置环境变量，或者使用 `setWorkingDirectory()` 为进程设置工作目录。

前面我们所说的信号槽机制，类似于前面我们介绍的 `QNetworkAccessManager`，都是异步的。与 `QNetworkAccessManager` 不同在于，`QProcess` 提供了同步函数：

- `waitForStarted()`：阻塞到进程开始；
- `waitForReadyRead()`：阻塞到可以从进程的当前读通道读取新的数据；
- `waitForBytesWritten()`：阻塞到数据写入进程；
- `waitForFinished()`：阻塞到进程结束；

注意，在主线程（调用了 `QApplication::exec()` 的线程）调用上面几个函数会让界面失去响应。

## 70 进程间通信

上一章我们了解了有关进程的基本知识。我们将进程理解为相互独立的正在运行的程序。由于二者是相互独立的，就存在交互的可能性，也就是我们所说的进程间通信（Inter-Process Communication, IPC）。不过也正因此，我们的一些简单的交互方式，比如普通的信号槽机制等，并不适用于进程间的相互通信。我们说过，进程是操作系统的基本调度单元，因此，进程间交互不可避免与操作系统的实现息息相关。

Qt 提供了四种进程间通信的方式：

1. 使用共享内存（shared memory）交互：这是 Qt 提供了一种各个平台均有支持的进程间交互的方式。
2. TCP/IP：其基本思想就是将同一机器上面的两个进程一个当做服务器，一个当做客户端，二者通过网络协议进行交互。除了两个进程是在同一台机器上，这种交互方式与普通的 C/S 程序没有本质区别。Qt 提供了 `QNetworkAccessManager` 对此进行支持。
3. D-Bus：freedesktop 组织开发的一种低开销、低延迟的 IPC 实现。Qt 提供了 `QtDBus` 模块，把信号槽机制扩展到进程级别（因此我们前面强调是“普通的”信号槽机制无法实现 IPC），使得开发者可以在一个进程中发出信号，由其它进程的槽函数响应信号。
4. QCOP（Qt COmmunication Protocol）：QCOP 是 Qt 内部的一种通信协议，用于不同的客户端之间在同一地址空间内部或者不同的进程之间的通信。目前，这种机制只用于 Qt for Embedded Linux 版本。

从上面的介绍中可以看到，通用的 IPC 实现大致只有共享内存和 TCP/IP 两种。后者我们前面已经大致介绍过（应用程序级别的 `QNetworkAccessManager` 或者更底层的 `QTcpSocket` 等）；本章我们主要介绍前者。

Qt 使用 `QSharedMemory` 类操作共享内存段。我们可以把 `QSharedMemory` 看做一种指针，这种指针指向分配出来的一个共享内存段。而这个共享内存段是由底层的操作系统提供，可以供多个线程或进程使用。因此，`QSharedMemory` 可以看做是专供 Qt 程序访问这个共享内存段的指针。同时，`QSharedMemory` 还提供了单一线程或进程互斥访问某一内存区域的能力。当我们创建了 `QSharedMemory` 实例后，可以使用其 `create()` 函数请求操作系统分配一个共享内存段。如果创建成功（函数返回 `true`），Qt 会自动将系统分配的共享内存段连接（attach）到本进程。

前面我们说过，IPC 离不开平台特性。作为 IPC 的实现之一的共享内存也遵循这一原则。有关共享内存段，各个平台的实现也有所不同：

- **Windows:** `QSharedMemory` 不“拥有”共享内存段。当使用了共享内存段的所有线程或进程中的某一个销毁了 `QSharedMemory` 实例，或者所有的都退出，Windows 内核会自动释放共享内存段。
- **Unix:** `QSharedMemory` “拥有”共享内存段。当最后一个线程或进程同共享内存分离，并且调用了 `QSharedMemory` 的析构函数之后，Unix 内核会将共享内存段释放。注意，这里与 Windows 不同之处在于，如果使用了共享内存段的线程或进程没有调用 `QSharedMemory` 的析构函数，程序将会崩溃。
- **HP-UX:** 每个进程只允许连接到一个共享内存段。这意味着在 HP-UX 平台，`QSharedMemory` 不应被多个线程使用。

下面我们通过一段经典的代码来演示共享内存的使用。这段代码修改自 Qt 自带示例程序（注意这里直接使用了 Qt5，Qt4 与此类似，这里不再赘述）。程序有两个按钮，一个按钮用于加载一张图片，然后将该图片放在共享内存段；第二个按钮用于从共享内存段读取该图片并显示出来。

```
//!!! Qt5

class QSharedMemory;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    QSharedMemory *sharedMemory;
};
```

头文件中，我们将 `MainWindow` 添加一个 `sharedMemory` 属性。这就是我们的共享内存段。接下来得实现文件中：

```
const char *KEY_SHARED_MEMORY = "Shared";

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent),
      sharedMemory(new QSharedMemory(KEY_SHARED_MEMORY, this))
{
    QWidget *mainWidget = new QWidget(this);
```

```

QVBoxLayout *mainLayout = new QVBoxLayout(mainWidget);
setCentralWidget(mainWidget);

QPushButton *saveButton = new QPushButton(tr("Save"), this);
mainLayout->addWidget(saveButton);
QLabel *picLabel = new QLabel(this);
mainLayout->addWidget(picLabel);
QPushButton *loadButton = new QPushButton(tr("Load"), this);
mainLayout->addWidget(loadButton);

```

构造函数初始化列表中我们将 `sharedMemory` 成员变量进行初始化。注意我们给出一个键（Key），前面说过，我们可以把 `QSharedMemory` 看做是指向系统共享内存段的指针，而这个键就可以看做指针的名字。多个线程或进程使用同一个共享内存段时，该键值必须相同。接下来是两个按钮和一个标签用于界面显示，这里不再赘述。

下面来看加载图片按钮的实现：

```

connect(saveButton, &QPushButton::clicked, [=]() {
    if (sharedMemory->isAttached()) {
        sharedMemory->detach();
    }
    QString filename = QFileDialog::getOpenFileName(this);
    QPixmap pixmap(filename);
    picLabel->setPixmap(pixmap);

    QBuffer buffer;
    QDataStream out(&buffer);
    buffer.open(QBuffer::ReadWrite);
    out << pixmap;

    int size = buffer.size();
    if (!sharedMemory->create(size)) {
        qDebug() << tr("Create Error: ") << sharedMemory->errorString();
    } else {
        sharedMemory->lock();
        char *to = static_cast<char*>(sharedMemory->data());
        const char *from = buffer.data().constData();
        memcpy(to, from, qMin(size, sharedMemory->size()));
        sharedMemory->unlock();
    }
});

```

点击加载按钮之后，如果 `sharedMemory` 已经与某个线程或进程连接，则将其断开（因为我们要向共享内存段写入内容了）。然后使用 `QFileDialog` 选择一张图片，利用 `QBuffer` 将图片数据作为 `char *` 格式。在即将写入共享内存之前，我们需要请求系统创建一个共享内存段（`QSharedMemory::create()` 函数），创建成功则开始写入共享内存段。需要注意的是，在读取或写入共享内存时，都需要使用 `QSharedMemory::lock()` 函数对共享内存段加锁。共享内存段就是一段普通内存，所以我们使用 C 语言标准函数 `memcpy()` 复制内存段。不要忘记之前我们对共享内存段加锁，在最后需要将其解锁。

接下来是加载按钮的代码：

```
connect(loadButton, &QPushButton::clicked, [=]() {
    if (!sharedMemory->attach()) {
        qDebug() << tr("Attach Error: ") << sharedMemory->errorString();
    } else {
        QBuffer buffer;
        QDataStream in(&buffer);
        QPixmap pixmap;
        sharedMemory->lock();
        buffer.setData(static_cast<sharedMemory->constData>(),
sharedMemory->size());
        buffer.open(QBuffer::ReadWrite);
        in >> pixmap;
        sharedMemory->unlock();
        sharedMemory->detach();
        picLabel->setPixmap(pixmap);
    }
});
```

如果共享内存段已经连接，还是用 `QBuffer` 读取二进制数据，然后生成图片。注意我们在操作共享内存段时还是要先加锁再解锁。最后在读取完毕后，将共享内存段断开连接。

注意，如果某个共享内存段不是由 Qt 创建的，我们也是可以在 Qt 应用程序中使用。不过这种情况下我们必须使用 `QSharedMemory::setNativeKey()` 来设置共享内存段。使用原始键（native key）时，`QSharedMemory::lock()` 函数就会失效，我们必须自己保护共享内存段不会在多线程或进程访问时出现问题。

IPC 使用共享内存通信是一个很常用的开发方法。多个进程间得通信要比多线程间得通信少一些，不过在某一族的应用情形下，比如 QQ 与 QQ 音乐、QQ 影音等共享用户头像，还是非常有用的。



## 71 线程简介

前面我们讨论了有关进程以及进程间通讯的相关问题，现在我们开始讨论线程。事实上，现代的程序中，使用线程的概率应该大于进程。特别是在多核时代，CPU 的主频已经进入瓶颈，另辟蹊径地提高程序运行效率就是使用线程，充分利用多核的优势。有关线程和进程的区别已经超出了本章的范畴，我们简单提一句，一个进程可以有一个或更多线程同时运行。线程可以看做是“轻量级进程”，进程完全由操作系统管理，线程即可以由操作系统管理，也可以由应用程序管理。

Qt 使用 QThread 来管理线程。下面来看一个简单的例子：

```
///Qt5
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *widget = new QWidget(this);
    QVBoxLayout *layout = new QVBoxLayout;
    widget->setLayout(layout);
    QLCDNumber *lcdNumber = new QLCDNumber(this);
    layout->addWidget(lcdNumber);
    QPushButton *button = new QPushButton(tr("Start"), this);
    layout->addWidget(button);
    setCentralWidget(widget);

    QTimer *timer = new QTimer(this);
    connect(timer, &QTimer::timeout, [=]() {
        static int sec = 0;
        lcdNumber->display(QString::number(sec++));
    });

    WorkerThread *thread = new WorkerThread(this);
    connect(button, &QPushButton::clicked, [=]() {
        timer->start(1);
        for (int i = 0; i < 2000000000; i++);
        timer->stop();
    });
}
```

我们的主界面有一个用于显示时间的 LCD 数字面板还有一个用于启动任务的按钮。程序的目的是用户点击按钮，开始一个非常耗时的运算（程序中我们以一个 2000000000 次的循环来替代这个非常耗时的工作，在真实的程序中，这可能是一个网络访问，可能是需要复制一个很大的文件或者其它任务），同时 LCD 开始显示逝去的毫秒数。毫秒数通过一个计时器

QTimer 进行更新。计算完成后，计时器停止。这是一个很简单的应用，也看不出有任何问题。但是当我们开始运行程序时，问题就来了：点击按钮之后，程序界面直接停止响应，直到循环结束才开始重新更新。

有经验的开发者立即指出，这里需要使用线程。这是因为 Qt 中所有界面都是在 UI 线程中（也被称为主线程，就是执行了 QApplication::exec()的线程），在这个线程中执行耗时的操作（比如那个循环），就会阻塞 UI 线程，从而让界面停止响应。界面停止响应，用户体验自然不好，不过更严重的是，有些窗口管理程序会检测到你的程序已经失去响应，可能会建议用户强制停止程序，这样一来你的程序可能就此终止，任务再也无法完成。所以，为了避免这一问题，我们要使用 QThread 开启一个新的线程：

```
///Qt5
class WorkerThread : public QThread
{
    Q_OBJECT
public:
    WorkerThread(QObject *parent = 0)
        : QThread(parent)
    {
    }
protected:
    void run()
    {
        for (int i = 0; i < 1000000000; i++);
        emit done();
    }
signals:
    void done();
};

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *widget = new QWidget(this);
    QVBoxLayout *layout = new QVBoxLayout;
    widget->setLayout(layout);
    lcdNumber = new QLCDNumber(this);
    layout->addWidget(lcdNumber);
    QPushButton *button = new QPushButton(tr("Start"), this);
    layout->addWidget(button);
    setCentralWidget(widget);

    QTimer *timer = new QTimer(this);
```

```

connect(timer, &QTimer::timeout, [=]() {
    static int sec = 0;
    lcdNumber->display(QString::number(sec++));
});

WorkerThread *thread = new WorkerThread(this);
connect(thread, &WorkerThread::done, timer, &QTimer::stop);
connect(thread, &WorkerThread::finished, thread,
&WorkerThread::deleteLater);
connect(button, &QPushButton::clicked, [=]() {
    timer->start(1);
    thread->start();
});
}

```

注意，我们增加了一个 `WorkerThread` 类。`WorkerThread` 继承自 `QThread` 类，重写了其 `run()` 函数。我们可以认为，`run()` 函数就是新的线程需要执行的代码。在这里就是要执行这个循环，然后发出计算完成的信号。而在按钮点击的槽函数中，使用 `QThread::start()` 函数启动一个线程（注意，这里不是 `run()` 函数）。再次运行程序，你会发现现在界面已经不会被阻塞了。另外，我们将 `WorkerThread::deleteLater()` 函数与 `WorkerThread::finished()` 信号连接起来，当线程完成时，系统可以帮我们清除线程实例。这里的 `finished()` 信号是系统发出的，与我们自定义的 `done()` 信号无关。

这是 Qt 线程的最基本的使用方式之一（确切的说，这种使用已经不大推荐使用，不过因为看起来很清晰，而且简单使用起来也没有什么问题，所以还是有必要介绍）。代码看起来很简单，不过，如果你认为 Qt 的多线程编程也很简单，那就大错特错了。Qt 多线程的优势设计使得它使用起来变得容易，但是坑很多，稍不留神就会被绊住，尤其是涉及到与 `QObject` 交互的情况。稍懂多线程开发的童鞋都会知道，调试多线程开发简直就是煎熬。下面几章，我们会更详细介绍有关多线程编程的相关内容。

## 72 线程和事件循环

前面一章我们简单介绍了如何使用 `QThread` 实现线程。现在我们开始详细介绍如何“正确”编写多线程程序。我们这里的大部分内容来自于 [Qt 的一篇 Wiki 文档](#)，有兴趣的童鞋可以去看原文。

在介绍在以前，我们要认识两个术语：

- **可重入的 (Reentrant)**：如果多个线程可以在同一时刻调用一个类的所有函数，并且保证每一次函数调用都引用一个唯一的数据，就称这个类是可重入的 (Reentrant means that all the functions in the referenced class can be called simultaneously by multiple threads, provided that each invocation of the functions reference unique data.)。大多数 C++ 类都是可重入的。类似的，一个函数被称为可重入的，如果该函数允许多个线程在同一时刻调用，而每一次的调用都只能使用其独有的数据。全局变量就不是函数独有的数据，而是共享的。换句话说，这意味着类或者函数的使用者必须使用某种额外的机制（比如锁）来控制对对象的实例或共享数据的序列化访问。
- **线程安全 (Thread-safe)**：如果多个线程可以在同一时刻调用一个类的所有函数，即使每一次函数调用都引用一个共享的数据，就说这个类是线程安全的 (Threadsafe means that all the functions in the referenced class can be called simultaneously by multiple threads even when each invocation references shared data.)。如果多个线程可以在同一时刻访问函数的共享数据，就称这个函数是线程安全的。

进一步说，对于一个类，如果不同的实例可以被不同线程同时使用而不受影响，就说这个类是可重入的；如果这个类的所有成员函数都可以被不同线程同时调用而不受影响，即使这些调用针对同一个对象，那么我们就说这个类是线程安全的。由此可以看出，线程安全的语义要强于可重入。接下来，我们从事件开始讨论。之前我们说过，Qt 是事件驱动的。在 Qt 中，事件由一个普通对象表示 (QEvent 或其子类)。这是事件与信号的一个很大区别：事件总是由某一种类型的对象表示，针对某一个特殊的对象，而信号则没有这种目标对象。所有 QObject 的子类都可以通过覆盖 QObject::event() 函数来控制事件的对象。

事件可以由程序生成，也可以在程序外部生成。例如：

- QKeyEvent 和 QMouseEvent 对象表示键盘或鼠标的交互，通常由系统的窗口管理器产生；
- QTimerEvent 事件在定时器超时时发送给一个 QObject，定时器事件通常由操作系统发出；
- QChildEvent 在增加或删除子对象时发送给一个 QObject，这是由 Qt 应用程序自己发出的。

需要注意的是，与信号不同，事件并不是一产生就被分发。事件产生之后被加入到一个队列中（这里的队列含义同数据结构中的概念，先进先出），该队列即被称为事件队列。事件分发器遍历事件队列，如果发现事件队列中有事件，那么就把这个事件发送给它的目标对象。这个循环被称作事件循环。事件循环的伪代码描述大致如下所示：

```
while (is_active)
{
```

```
while (!event_queue_is_empty) {  
    dispatch_next_event();  
}  
wait_for_more_events();  
}
```

正如前面所说的，调用 `QCoreApplication::exec()` 函数意味着进入了主循环。我们把事件循环理解为一个无限循环，直到 `QCoreApplication::exit()` 或者 `QCoreApplication::quit()` 被调用，事件循环才真正退出。

伪代码里面的 `while` 会遍历整个事件队列，发送从队列中找到的事件；  
`wait_for_more_events()` 函数则会阻塞事件循环，直到又有新的事件产生。我们仔细考虑这段代码，在 `wait_for_more_events()` 函数所得到的新的事件都应该是由程序外部产生的。因为所有内部事件都应该在事件队列中处理完毕了。因此，我们说事件循环在 `wait_for_more_events()` 函数进入休眠，并且可以被下面几种情况唤醒：

- 窗口管理器的动作（键盘、鼠标按键按下、与窗口交互等）；
- 套接字动作（网络传来可读的数据，或者是套接字非阻塞写等）；
- 定时器；
- 由其它线程发出的事件（我们会在后文详细解释这种情况）。

在类 UNIX 系统中，窗口管理器（比如 X11）会通过套接字（Unix Domain 或 TCP/IP）向应用程序发出窗口活动的通知，因为客户端就是通过这种机制与 X 服务器交互的。如果我们决定要实现基于内部的 `socketpair(2)` 函数的跨线程事件的派发，那么窗口的管理活动需要唤醒的是：

- 套接字 `socket`
- 定时器 `timer`

这也正是 `select(2)` 系统调用所做的：它监视窗口活动的一组描述符，如果在一定时间内没有活动，它会发出超时消息（这种超时是可配置的）。Qt 所要做的，就是把 `select()` 的返回值转换成一个合适的 `QEvent` 子类的对象，然后将其放入事件队列。好了，现在你已经知道事件循环的内部机制了。

至于为什么需要事件循环，我们可以简单列出一个清单：

- **组件的绘制与交互：** `QWidget::paintEvent()` 会在发出 `QPaintEvent` 事件时被调用。该事件可以通过内部 `QWidget::update()` 调用或者窗口管理器（例如显示一个隐藏的窗口）发出。所有交互事件（键盘、鼠标）也是类似的：这些事件都要求有一个事件循环才能发出。
- **定时器：** 长话短说，它们会在 `select(2)` 或其他类似的调用超时时被发出，因此你需要允许 Qt 通过返回事件循环来实现这些调用。
- **网络：** 所有低级网络类（`QTcpSocket`、`QUdpSocket` 以及 `QTcpServer` 等）都是异步的。当你调用 `read()` 函数时，它们仅仅返回已可用的数据；当你调用 `write()` 函数时，它们仅仅将写入列入计划列表稍后执行。只有返回事件循环的时候，真正的读写才会执行。注意，这些类也有同步函数（以 `waitFor` 开头的函数），但是它们并不推荐使用，就是因为它们会阻塞事件循环。高级的类，例如 `QNetworkAccessManager` 则根本不提供同步 API，因此必须要求事件循环。

有了事件循环，你就会想怎样阻塞它。阻塞它的理由可能有很多，例如我就想让 `QNetworkAccessManager` 同步执行。在解释为什么**永远不要阻塞事件循环**之前，我们要了解究竟什么是“阻塞”。假设我们有一个按钮 `Button`，这个按钮在点击时会发出一个信号。这个信号会与一个 `Worker` 对象连接，这个 `Worker` 对象会执行很耗时的操作。当点击了按钮之后，我们观察从上到下的函数调用堆栈：

```
main(int, char **)
QApplication::exec()
[...]
QWidget::event(QEvent *)
Button::mousePressEvent(QMouseEvent *)
Button::clicked()
[...]
Worker::doWork()
```

我们在 `main()` 函数开始事件循环，也就是常见的 `QApplication::exec()` 函数。窗口管理器侦测到鼠标点击后，Qt 会发现并将其转换成 `QMouseEvent` 事件，发送给组件的 `event()` 函数。这一过程是通过 `QApplication::notify()` 函数实现的。注意我们的按钮并没有覆盖 `event()` 函数，因此其父类的实现将被执行，也就是 `QWidget::event()` 函数。这个函数发现这个事件是一个鼠标点击事件，于是调用了对应的事件处理函数，就是 `Button::mousePressEvent()` 函数。我们重写了这个函数，发出 `Button::clicked()` 信号，而正是这个信号会调用 `Worker::doWork()` 槽函数。有关这一机制我们在前面的事件部分曾有阐述，如果不明白这部分机制，请参考[前面的章节](#)。

在 `worker` 努力工作的时候，事件循环在干什么？或许你已经猜到了答案：什么都没做！事件循环发出了鼠标按下的事件，然后等着事件处理函数返回。此时，它一直是阻塞的，直到

Worker::doWork()函数结束。注意，我们使用了“阻塞”一词，也就是说，所谓**阻塞事件循环**，意思是没有事件被派发处理。

在事件就此卡住时，**组件也不会更新自身**（因为 QPaintEvent 对象还在队列中），**也不会有其它什么交互发生**（还是同样的原因），**定时器也不会超时并且网络交互会越来越慢直到停止**。也就是说，前面我们大费周折分析的各种依赖事件循环的活动都会停止。这时候，需要窗口管理器会检测到你的应用程序不再处理任何事件，于是**告诉用户你的程序失去响应**。这就是为什么我们需要快速地处理事件，并且尽可能快地返回事件循环。

现在，重点来了：我们不可能避免业务逻辑中的耗时操作，那么怎样做才能既可以执行那些耗时的操作，又不会阻塞事件循环呢？一般会有三种解决方案：第一，我们将任务移到另外的线程（正如我们[上一章](#)看到的那样，不过现在我们暂时略过这部分内容）；第二，我们手动强制运行事件循环。想要强制运行事件循环，我们需要在耗时的任务中一遍遍地调用 QCoreApplication::processEvents()函数。QCoreApplication::processEvents()函数会发出事件队列中的所有事件，并且立即返回到调用者。仔细想一下，我们在这里所做的，就是模拟了一个事件循环。

另外一种解决方案我们在[前面的章节](#)提到过：使用 QEventLoop 类重新进入新的事件循环。通过调用 QEventLoop::exec()函数，我们重新进入新的事件循环，给 QEventLoop::quit()槽函数发送信号则退出这个事件循环。拿前面的例子来说：

```
QEventLoop eventLoop;
connect(netWorker, &NetWorker::finished,
        &eventLoop, &QEventLoop::quit);
QNetworkReply *reply = netWorker->get(url);
replyMap.insert(reply, FetchWeatherInfo);
eventLoop.exec();
```

QNetworkReply 没有提供阻塞式 API，并且要求有一个事件循环。我们通过一个局部的 QEventLoop 来达到这一目的：当网络响应完成时，这个局部的事件循环也会退出。

前面我们也强调过：通过“其它的入口”进入事件循环要特别小心：因为它会导致递归调用！现在我们可以看看为什么会递归调用了。回过头来看看按钮的例子。当我们在 Worker::doWork()槽函数中调用了 QCoreApplication::processEvents()函数时，用户再次点击按钮，槽函数 Worker::doWork()又一次被调用：

```
main(int, char **)
QApplication::exec()
[...]
QWidget::event(QEvent *)
Button::mousePressEvent(QMouseEvent *)
Button::clicked()
```

```
[...]
Worker::doWork() // <strong>第一次调用</strong>
QCoreApplication::processEvents() // <strong>手动发出所有事件</strong>
[...]
QWidget::event(QEvent * ) // <strong>用户又点击了一下按钮…</strong>
Button::mousePressEvent(QMouseEvent *)
Button::clicked() // <strong>又发出了信号…</strong>
[...]
Worker::doWork() // <strong>递归进入了槽函数! </strong>
```

当然，这种情况也有解决的办法：我们可以在调用 `QCoreApplication::processEvents()` 函数时传入 `QEventLoop::ExcludeUserInputEvents` 参数，意思是不要再次派发用户输入事件（这些事件仍旧会保留在事件队列中）。

幸运的是，在**删除事件**（也就是由 `QObject::deleteLater()` 函数加入到事件队列中的事件）中，**没有**这个问题。这是因为删除事件是由另外的机制处理的。删除事件只有在事件循环有比较小的“嵌套”的情况下才会被处理，而不是调用了 `deleteLater()` 函数的那个循环。例如：

```
QObject *object = new QObject;
object->deleteLater();
QDialog dialog;
dialog.exec();
```

这段代码**并不会**造成野指针（注意，`QDialog::exec()` 的调用是嵌套在 `deleteLater()` 调用所在的事件循环之内的）。通过 `QEventLoop` 进入局部事件循环也是类似的。在 Qt 4.7.3 中，唯一的例外是，在没有事件循环的情况下直接调用 `deleteLater()` 函数，那么，之后第一个进入的事件循环会获取这个事件，然后直接将这个对象删除。不过这也是合理的，因为 Qt 本来不知道会执行删除操作的那个“外部的”事件循环，所以第一个事件循环就会直接删除对象。

## 73 Qt 线程相关类

希望上一章有关事件循环的内容还没有把你绕晕。本章将重新回到有关线程的相关内容上来。在前面的章节我们了解了有关 `QThread` 类的简单使用。不过，Qt 提供的有关线程的类可不那么简单，否则的话我们也没必要再三强调使用线程一定要万分小心，一不留神就会陷入陷阱。

事实上，Qt 对线程的支持可以追溯到 2000 年 9 月 22 日发布的 Qt 2.2。在这个版本中，Qt 引入了 `QThread`。不过，当时对线程的支持并不是默认开启的。Qt 4.0 开始，线程成为所有平台的默认开启选项（这意味着如果不需要线程，你可以通过编译选项关闭它，不过这不是



我们现在的重点)。现在版本的 Qt 引入了很多类来支持线程，下面我们将开始逐一了解它们。

QThread 是我们将要详细介绍的第一个类。它也是 Qt 线程类中最核心的底层类。由于 Qt 的跨平台特性，QThread 要隐藏掉所有平台相关的代码。

正如前面所说，要使用 QThread 开始一个线程，我们可以创建它的一个子类，然后覆盖其 QThread::run()函数：

```
class Thread : public QThread
{
protected:
    void run()
    {
        /* 线程的相关代码 */
    }
};
```

然后我们这样使用新建的类来开始一个新的线程：

```
Thread *thread = new Thread;
thread->start(); // 使用 start() 开始新的线程
```

注意，从 Qt 4.4 开始，QThread 就已经不是抽象类了。QThread::run()不再是纯虚函数，而是有了一个默认的实现。这个默认实现其实是简单地调用了 QThread::exec()函数，而这个函数，按照我们前面所说的，其实是开始了一个事件循环（有关这种实现的进一步阐述，我们将在后面的章节详细介绍）。

QRunnable 是我们要介绍的第二个类。这是一个轻量级的抽象类，用于开始一个另外线程的任务。这种任务是运行过后就丢弃的。由于这个类是抽象类，我们需要继承 QRunnable，然后重写其纯虚函数 QRunnable::run()：

```
class Task : public QRunnable
{
public:
    void run()
    {
        /* 线程的相关代码 */
    }
};
```

要真正执行一个 `QRunnable` 对象，我们需要使用 `QThreadPool` 类。顾名思义，这个类用于管理一个线程池。通过调用 `QThreadPool::start(runnable)` 函数，我们将一个 `QRunnable` 对象放入 `QThreadPool` 的执行队列。一旦有线程可用，线程池将会选择一个 `QRunnable` 对象，然后在那个线程开始执行。所有 Qt 应用程序都有一个全局线程池，我们可以使用 `QThreadPool::globalInstance()` 获得这个全局线程池；与此同时，我们也可以自己创建私有的线程池，并进行手动管理。

需要注意的是，`QRunnable` 不是一个 `QObject`，因此也就没有内建的与其它组件交互的机制。为了与其它组件进行交互，你必须自己编写低级线程原语，例如使用 `mutex` 守护来获取结果等。

`QtConcurrent` 是我们要介绍的最后一个对象。这是一个高级 API，构建于 `QThreadPool` 之上，用于处理大多数通用的并行计算模式：`map`、`reduce` 以及 `filter`。它还提供了 `QtConcurrent::run()` 函数，用于在另外的线程运行一个函数。注意，`QtConcurrent` 是一个命名空间而不是一个类，因此其中的所有函数都是命名空间内的全局函数。

不同于 `QThread` 和 `QRunnable`，`QtConcurrent` 不要求我们使用低级同步原语：所有的 `QtConcurrent` 都返回一个 `QFuture` 对象。这个对象可以用来查询当前的运算状态（也就是任务的进度），可以用来暂停/回复/取消任务，当然也可以用来获得运算结果。注意，并不是所有的 `QFuture` 对象都支持暂停或取消的操作。比如，由 `QtConcurrent::run()` 返回的 `QFuture` 对象不能取消，但是由 `QtConcurrent::mappedReduced()` 返回的是可以的。`QFutureWatcher` 类则用来监视 `QFuture` 的进度，我们可以用信号槽与 `QFutureWatcher` 进行交互（注意，`QFuture` 也没有继承 `QObject`）。

下面我们可以对比一下上面介绍过的三种类：

特性	QThread	QRunnable	QtConcurrent
高级 API	✗	✗	✓
面向任务	✗	✓	✓
内建对暂停/恢复/取消的支持	✗	✗	✓
具有优先级	✓	✗	✗

可运行事件循环

✓

✗

✗

## 74 线程和 QObject

前面两个章节我们从事件循环和线程类库两个角度阐述有关线程的问题。本章我们将深入线程间得交互，探讨线程和 QObject 之间的关系。在某种程度上，这才是多线程编程真正需要注意的问题。

现在我们已经讨论过事件循环。我们说，每一个 Qt 应用程序至少有一个事件循环，就是调用了 `QCoreApplication::exec()` 的那个事件循环。不过，`QThread` 也可以开启事件循环。只不过这是一个受限于线程内部的事件循环。因此我们将处于调用 `main()` 函数的那个线程，并且由 `QCoreApplication::exec()` 创建开启的那个事件循环成为主事件循环，或者直接叫主循环。注意，`QCoreApplication::exec()` 只能在调用 `main()` 函数的线程调用。主循环所在的线程就是主线程，也被成为 GUI 线程，因为所有有关 GUI 的操作都必须在这个线程进行。`QThread` 的局部事件循环则可以通过在 `QThread::run()` 中调用 `QThread::exec()` 开启：

```
class Thread : public QThread
{
protected:
    void run() {
        /* ... 初始化 ... */
        exec();
    }
};
```

记得我们前面介绍过，Qt 4.4 版本以后，`QThread::run()` 不再是纯虚函数，它会调用 `QThread::exec()` 函数。与 `QCoreApplication` 一样，`QThread` 也有 `QThread::quit()` 和 `QThread::exit()` 函数来终止事件循环。

线程的事件循环用于为线程中的所有 `QObject`s 对象分发事件；默认情况下，这些对象包括线程中创建的所有对象，或者是在别处创建完成后被移动到该线程的对象（我们会在后面详细介绍“移动”这个问题）。我们说，一个 `QObject` 的线程依附性（thread affinity）是指它所在的那个线程。它同样适用于在 `QThread` 的构造函数中构建的对象：

```
class MyThread : public QThread
{
public:
```

```

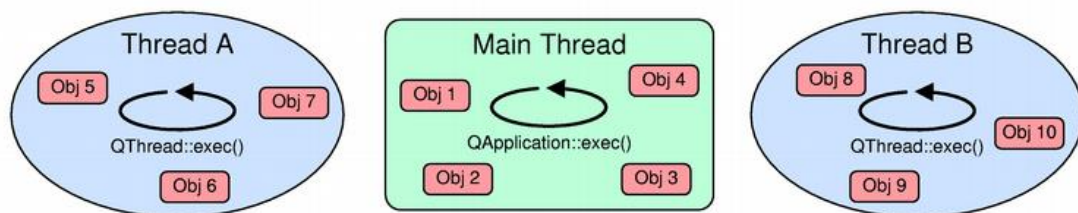
MyThread()
{
    otherObj = new QObject;
}

private:
    QObject obj;
    QObject *otherObj;
    QScopedPointer yetAnotherObj;
};

```

在我们创建了 `MyThread` 对象之后，`obj`、`otherObj` 和 `yetAnotherObj` 的线程依附性是怎样的？是不是就是 `MyThread` 所表示的那个线程？要回答这个问题，我们必须看看究竟是哪个线程创建了它们：实际上，是调用了 `MyThread` 构造函数的线程创建了它们。因此，这些对象不在 `MyThread` 所表示的线程，而是在创建了 `MyThread` 的那个线程中。

我们可以通过调用 `QObject::thread()` 可以查询一个 `QObject` 的线程依附性。注意，在 `QCoreApplication` 对象之前创建的 `QObject` 没有所谓线程依附性，因此也就没有对象为其派发事件。也就是说，实际是 `QCoreApplication` 创建了代表主线程的 `QThread` 对象。



我们可以使用线程安全的 `QCoreApplication::postEvent()` 函数向一个对象发送事件。它将事件加入到对象所在的线程的事件队列中，因此，如果这个线程没有运行事件循环，这个事件也不会被派发。

值得注意的是，`QObject` 及其所有子类都不是线程安全的（但都是可重入的）。因此，你不能有两个线程同时访问一个 `QObject` 对象，除非这个对象的内部数据都已经很好地序列化（例如为每个数据访问加锁）。记住，在你从另外的线程访问一个对象时，它可能正在处理所在线程的事件循环派发的事件！基于同样的原因，你也不能在另外的线程直接 `delete` 一个 `QObject` 对象，相反，你需要调用 `QObject::deleteLater()` 函数，这个函数会给对象所在线程发送一个删除的事件。

此外，`QWidget` 及其子类，以及所有其它 GUI 相关类（即便不是 `QObject` 的子类，例如 `QPixmap`），甚至不是可重入的：它们只能在 GUI 线程访问。

QObject 的线程依附性是可以改变的，方法是调用 `QObject::moveToThread()` 函数。该函数会改变一个对象及其所有子对象的线程依附性。由于 QObject 不是线程安全的，所以我们只能在该对象所在线程上调用这个函数。也就是说，我们只能在对象所在线程将这个对象移动到另外的线程，不能在另外的线程改变对象的线程依附性。还有一点是，Qt 要求 QObject 的所有子对象都必须和其父对象在同一线程。这意味着：

- 不能对有父对象（parent 属性）的对象使用 `QObject::moveToThread()` 函数
- 不能在 `QThread` 中以这个 `QThread` 本身作为父对象创建对象，例如：

```
class Thread : public QThread {
    void run() {
        QObject *obj = new QObject(this); // 错误!
    }
};
```

这是因为 `QThread` 对象所依附的线程是创建它的那个线程，而不是它所代表的线程。

Qt 还要求，在代表一个线程的 `QThread` 对象销毁之前，所有在这个线程中的对象都必须先 `delete`。要达到这一点并不困难：我们只需在 `QThread::run()` 的栈上创建对象即可。

现在的问题是，既然线程创建的对象都只能在函数栈上，怎么能让这些对象与其它线程的对象通信呢？Qt 提供了一个优雅清晰的解决方案：我们在线程的事件队列中加入一个事件，然后在事件处理函数中调用我们所关心的函数。显然这需要线程有一个事件循环。这种机制依赖于 moc 提供的反射：因此，只有信号、槽和使用 `Q_INVOKABLE` 宏标记的函数可以在另外的线程中调用。

`QMetaObject::invokeMethod()` 静态函数会这样调用：

```
QMetaObject::invokeMethod(object, "methodName",
                           Qt::QueuedConnection,
                           Q_ARG(type1, arg1),
                           Q_ARG(type2, arg2));
```

注意，上面函数调用中出现的参数类型都必须提供一个公有构造函数，一个公有的析构函数和一个公有的复制构造函数，并且要使用 `qRegisterMetaType()` 函数向 Qt 类型系统注册。

跨线程的信号槽也是类似的。当我们将信号与槽连接起来时，`QObject::connect()` 的最后一个参数将指定连接类型：

- `Qt::DirectConnection`：直接连接意味着槽函数将在信号发出的线程直接调用

- **Qt::QueuedConnection:** 队列连接意味着向接受者所在线程发送一个事件，该线程的事件循环将获得这个事件，然后之后的某个时刻调用槽函数
- **Qt::BlockingQueuedConnection:** 阻塞的队列连接就像队列连接，但是发送者线程将会阻塞，直到接受者所在线程的事件循环获得这个事件，槽函数被调用之后，函数才会返回
- **Qt::AutoConnection:** 自动连接（默认）意味着如果接受者所在线程就是当前线程，则使用直接连接；否则将使用队列连接

注意在上面每种情况中，发送者所在线程都是无关紧要的！在自动连接情况下，Qt 需要查看信号发出的线程是不是与**接受者所在线程**一致，来决定连接类型。注意，Qt 检查的是**信号发出的线程**，而不是信号发出的对象所在的线程！我们可以看看下面的代码：

```
class Thread : public QThread
{
    Q_OBJECT
signals:
    void aSignal();
protected:
    void run() {
        emit aSignal();
    }
};

/* ... */
Thread thread;
Object obj;
QObject::connect(&thread, SIGNAL(aSignal()), &obj, SLOT(aSlot()));
thread.start();
```

aSignal()信号在一个新的线程被发出（也就是 Thread 所代表的线程）。注意，因为这个线程并不是 Object 所在的线程（Object 所在的线程和 Thread 所在的是同一个线程，回忆下，信号槽的连接方式与发送者所在线程无关），所以这里将会使用队列连接。

另外一个常见的错误是：

```
class Thread : public QThread
{
    Q_OBJECT
slots:
    void aSlot() {
        /* ... */
    }
};
```

```

    }
protected:
    void run() {
        /* ... */
    }
};

/* ... */
Thread thread;
Object obj;
QObject::connect(&obj, SIGNAL(aSignal()), &thread, SLOT(aSlot()));
thread.start();
obj.emitSignal();

```

这里的 obj 发出 aSignal() 信号时，使用哪种连接方式？答案是：直接连接。因为 Thread 对象所在线程发出了信号，也就是信号发出的线程与接受者是同一个。在 aSlot() 槽函数中，我们可以直接访问 Thread 的某些成员变量，但是注意，在我们访问这些成员变量时，Thread::run() 函数可能也在访问！这意味着二者并发进行：这是一个完美的导致崩溃的隐藏 bug。

另外一个例子可能更为重要：

```

class Thread : public QThread
{
    Q_OBJECT
slots:
    void aSlot() {
        /* ... */
    }
protected:
    void run() {
        QObject *obj = new Object;
        connect(obj, SIGNAL(aSignal()), this, SLOT(aSlot()));
        /* ... */
    }
};

```

这个例子也会使用队列连接。然而，这个例子比上面的例子更具隐蔽性：在这个例子中，你可能会觉得，Object 所在 Thread 所代表的线程中被创建，又是访问的 Thread 自己的成员数据。稍有不慎便会写出这种代码。

为了解决这个问题，我们可以这么做：Thread 构造函数中增加一个函数调用：  
moveToThread(this):

```

class Thread : public QThread {
Q_OBJECT
public:
    Thread() {
        moveToThread(this); // 错误!
    }

    /* ... */
};

```

实际上，这的确可行（因为 `Thread` 的线程依附性被改变了：它所在的线程成了自己），但是这并不是一个好主意。这种代码意味着我们其实误解了线程对象（`QThread` 子类）的设计意图：`QThread` 对象不是线程本身，它们其实是用于管理它所代表的线程的对象。因此，它们应该在另外的线程被使用（通常就是它自己所在的线程），而不是在自己所代表的线程中。

上面问题的最好的解决方案是，将处理任务的部分与管理线程的部分分离。简单来说，我们可以利用一个 `QObject` 的子类，使用 `QObject::moveToThread()` 改变其线程依附性：

```

class Worker : public QObject
{
Q_OBJECT
public slots:
    void doWork() {
        /* ... */
    }
};

/* ... */
QThread *thread = new QThread;
Worker *worker = new Worker;
connect(obj, SIGNAL(workReady()), worker, SLOT(doWork()));
worker->moveToThread(thread);
thread->start();

```

## 75 线程总结

前面我们已经详细介绍过有关线程的一些值得注意的事项。现在我们开始对线程做一些总结。

有关线程，你可以做的是：



- 在 `QThread` 子类添加信号。这是绝对安全的，并且也是正确的（前面我们已经详细介绍过，发送者的线程依附性没有关系）

不应该做的是：

- 调用 `moveToThread(this)` 函数
- 指定连接类型：这通常意味着你正在做错误的事情，比如将 `QThread` 控制接口与业务逻辑混杂在了一起（而这应该放在该线程的一个独立对象中）
- 在 `QThread` 子类添加槽函数：这意味着它们将在错误的线程被调用，也就是 `QThread` 对象所在线程，而不是 `QThread` 对象管理的线程。这又需要你指定连接类型或者调用 `moveToThread(this)` 函数
- 使用 `QThread::terminate()` 函数

不能做的是：

- 在线程还在运行时退出程序。使用 `QThread::wait()` 函数等待线程结束
- 在 `QThread` 对象所管理的线程仍在运行时就销毁该对象。如果你需要某种“自行销毁”的操作，你可以把 `finished()` 信号同 `deleteLater()` 槽连接起来

那么，下面一个问题是：我什么时候应该使用线程？

**首先，当你不得不使用同步 API 的时候。**

如果你需要使用一个没有非阻塞 API 的库或代码（所谓非阻塞 API，很大程度上就是指信号槽、事件、回调等），那么，避免事件循环被阻塞的解决方案就是使用进程或者线程。不过，由于开启一个新的工作进程，让这个进程去完成任务，然后再与当前进程进行通信，这一系列操作的代价都要比开启线程要昂贵得多，所以，线程通常是最好的选择。

一个很好的例子是地址解析服务。注意我们这里并不讨论任何第三方 API，仅仅假设一个有这样功能的库。这个库的工作是将一个主机名转换成地址。这个过程需要去到一个系统（也就是域名系统，Domain Name System, DNS）执行查询，这个系统通常是一个远程系统。一般这种响应应该瞬间完成，但是并不排除远程服务器失败、某些包可能会丢失、网络可能失去链接等等。简单来说，我们的查询可能会等几十秒钟。

UNIX 系统上的标准 API 是阻塞的（不仅是旧的 `gethostbyname(3)`，就连新的 `getservbyname(3)` 和 `getaddrinfo(3)` 也是一样）。Qt 提供的 `QHostInfo` 类同样用于地址解

析，默认情况下，内部使用一个 `QThreadPool` 提供后台运行方式的查询（如果关闭了 Qt 的线程支持，则提供阻塞式 API）。

另外一个例子是图像加载和缩放。`QImageReader` 和 `QImage` 只提供了阻塞式 API，允许我们从设备读取图片，或者是缩放到不同的分辨率。如果你需要处理很大的图像，这种任务会花费几十秒钟。

**其次，当你希望扩展到多核应用的时候。**

线程允许你的程序利用多核系统的优势。每一个线程都可以被操作系统独立调度，如果你的程序运行在多核机器上，调度器很可能会将每一个线程分配到各自的处理器上面运行。

举个例子，一个程序需要为很多图像生成缩略图。一个具有固定 `n` 个线程的线程池，每一个线程交给系统中的一个可用的 CPU 进行处理（我们可以使用 `QThread::idealThreadCount()` 获取可用的 CPU 数）。这样的调度将会把图像缩放工作交给所有线程执行，从而有效地提升效率，几乎达到与 CPU 数的线性提升（实际情况不会这么简单，因为有时候 CPU 并不是瓶颈所在）。

**第三，当你不想被别人阻塞的时候。**

这是一个相当高级的话题，所以你现在可以暂时不看这段。这个问题的一个很好的例子是在 WebKit 中使用 `QNetworkAccessManager`。WebKit 是一个现代的浏览器引擎。它帮助我们展示网页。Qt 中的 `QWebView` 就是使用的 WebKit。

`QNetworkAccessManager` 则是 Qt 处理 HTTP 请求和响应的通用类。我们可以将它看做浏览器的网络引擎。在 Qt 4.8 之前，这个类没有使用任何协助工作线程，所有的网络处理都是在 `QNetworkAccessManager` 及其 `QNetworkReply` 所在线程完成。

虽然在网络处理中不使用线程是一个好主意，但它也有一个很大的缺点：如果你不能及时从 socket 读取数据，内核缓冲区将会被填满，于是开始丢包，传输速度将会直线下降。

socket 活动（也就是从一个 socket 读取一些可用的数据）是由 Qt 的事件循环管理的。因此，阻塞事件循环将会导致传输性能的损失，因为没有人会获得有数据可读的通知，因此也就没有人能够读取这些数据。

但是什么会阻塞事件循环？最坏的答案是：WebKit 自己！只要收到数据，WebKit 就开始生成网页布局。不幸的是，这个布局的过程非常复杂和耗时，因此它会阻塞事件循环。尽管阻塞时间很短，但是足以影响到正常的数据传输（宽带连接在这里发挥了作用，在很短时间内就可以塞满内核缓冲区）。

总结一下上面所说的内容：

- WebKit 发起一次请求
- 从服务器响应获取一些数据
- WebKit 利用到达的数据开始进行网页布局，阻塞事件循环
- 由于事件循环被阻塞，也就没有了可用的事件循环，于是操作系统接收了到达的数据，但是却不能从 `QNetworkAccessManager` 的 `socket` 读取
- 内核缓冲区被填满，传输速度变慢

网页的整体加载时间被自身的传输速度的降低而变得越来越坏。

注意，由于 `QNetworkAccessManager` 和 `QNetworkReply` 都是 `QObject`，所以它们都不是线程安全的，因此你不能将它们移动到另外的线程继续使用。因为它们可能同时有两个线程访问：你自己的和它们所在的线程，这是因为派发给它们的事件会由后面一个线程的事件循环发出，但你不能确定哪一线程是“后面一个”。

Qt 4.8 之后，`QNetworkAccessManager` 默认会在一个独立的线程处理 HTTP 请求，所以导致 GUI 失去响应以及操作系统缓冲区过快填满的问题应该已经被解决了。

那么，什么情况下不应该使用线程呢？

## 定时器

这可能是最容易误用线程的情况了。如果我们需要每隔一段时间调用一个函数，很多人可能会这么写代码：

```
// 最错误的代码
while (condition) {
    doWork();
    sleep(1); // C 库里面的 sleep(3) 函数
}
```

当读过我们前面的文章之后，可能又会引入线程，改成这样的代码：

```
// 错误的代码
class Thread : public QThread {
protected:
    void run() {
        while (condition) {
            // 注意，如果我们要在别的线程修改 condition，那么它也需要加锁
        }
    }
}
```

```

        doWork();
        sleep(1); // 这次是 QThread::sleep()
    }
}
};

```

最好最简单的实现是使用定时器，比如 `QTimer`，设置 1s 超时，然后将 `doWork()` 作为槽：

```

class Worker : public QObject
{
    Q_OBJECT
public:
    Worker()
    {
        connect(&timer, SIGNAL(timeout()), this, SLOT(doWork()));
        timer.start(1000);
    }
private slots:
    void doWork()
    {
        /* ... */
    }
private:
    QTimer timer;
};

```

我们所需要的就是开始事件循环，然后每隔一秒 `doWork()` 就会被自动调用。

## 网络/状态机

下面是一个很常见的处理网络操作的设计模式：

```

socket->connect(host);
socket->waitForConnected();

data = getData();
socket->write(data);
socket->waitForBytesWritten();

socket->waitForReadyRead();
socket->read(response);

reply = process(response);

```

```
socket->write(reply);
socket->waitForBytesWritten();
/* ... */
```

在经过前面几章的介绍之后，不用多说，我们就会发现这里的问题：大量的 `waitFor*()` 函数会阻塞事件循环，冻结 UI 界面等等。注意，上面的代码还没有加入异常处理，否则的话肯定会更复杂。这段代码的错误在于，我们的网络实际是异步的，如果我们非得按照同步方式处理，就像拿起枪打自己的脚。为了解决这个问题，很多人会简单地将这段代码移动到一个新的线程。

一个更抽象的例子是：

```
result = process_one_thing();

if (result->something()) {
    process_this();
} else {
    process_that();
}

wait_for_user_input();
input = read_user_input();
process_user_input(input);
/* ... */
```

这段抽象的代码与前面网络的例子有“异曲同工之妙”。

让我们回过头来看看这段代码究竟是做了什么：我们实际是想创建一个状态机，这个状态机要根据用户的输入作出合理的响应。例如我们网络的例子，我们实际是想要构建这样的东西：

```
空闲 → 正在连接（调用 connectToHost()）
正在连接 → 成功连接（发出 connected() 信号）
成功连接 → 发送登录数据（将登录数据发送到服务器）
发送登录数据 → 登录成功（服务器返回 ACK）
发送登录数据 → 登录失败（服务器返回 NACK）
```

以此类推。

既然知道我们的实际目的，我们就可以修改代码来创建一个真正的状态机（Qt 甚至提供了一个状态机类：`QStateMachine`）。创建状态机最简单的方法是使用一个枚举来记住当前状态。我们可以编写如下代码：

```
class Object : public QObject
```

```

{
    Q_OBJECT
    enum State {
        State1, State2, State3 /* ... */
    };
    State state;
public:
    Object() : state(State1)
    {
        connect(source, SIGNAL(ready()), this, SLOT(doWork()));
    }
private slots:
    void doWork() {
        switch (state) {
            case State1:
                /* ... */
                state = State2;
                break;
            case State2:
                /* ... */
                state = State3;
                break;
            /* ... */
        }
    }
};

```

source 对象是哪来的？这个对象其实就是我们关心的对象：例如，在网络例子中，我们可能希望把 socket 的 `QAbstractSocket::connected()` 或者 `QIODevice::readyRead()` 信号与我们的槽函数连接起来。当然，我们很容易添加更多更合适的代码（比如错误处理，使用 `QAbstractSocket::error()` 信号就可以了）。这种代码是真正异步、信号驱动的设计。

### 将任务分割成若干部分

假设我们有一个很耗时的计算，我们不能简单地将它移动到另外的线程（或者是我们根本无法移动它，比如这个任务必须在 GUI 线程完成）。如果我们将这个计算任务分割成小块，那么我们就可以及时返回事件循环，从而让事件循环继续派发事件，调用处理下一个小块的函数。回一下如何实现队列连接，我们就可以轻松完成这个任务：将事件提交到接收对象所在线程的事件循环；当事件发出时，响应函数就会被调用。

我们可以使用 `QMetaObject::invokeMethod()` 函数，通过指定 `Qt::QueuedConnection` 作为调用类型来达到相同的效果。不过这要求函数必须是内省的，也就是说这个函数要么是一

个槽函数，要么标记有 `Q_INVOKABLE` 宏。如果我们还需要传递参数，我们需要使用 `qRegisterMetaType()` 函数将参数注册到 Qt 元类型系统。下面是代码示例：

```
class Worker : public QObject
{
    Q_OBJECT
public slots:
    void startProcessing()
    {
        processItem(0);
    }

    void processItem(int index)
    {
        /* 处理 items[index] ... */
        if (index < numberOfItems) {
            QMetaObject::invokeMethod(this,
                                      "processItem",
                                      Qt::QueuedConnection,
                                      Q_ARG(int, index + 1));
        }
    }
};
```

由于没有任何线程调用，所以我们可以轻易对这种计算任务执行暂停/恢复/取消，以及获取结果。

至此，我们利用五个章节将有关线程的问题简单介绍了下。线程应该说是全部设计里面最复杂的部分之一，所以这部分内容也会比较困难。在实际运用中肯定会更多的问题，这就只能让我们具体分析了。

## 76 QML 和 QtQuick 2

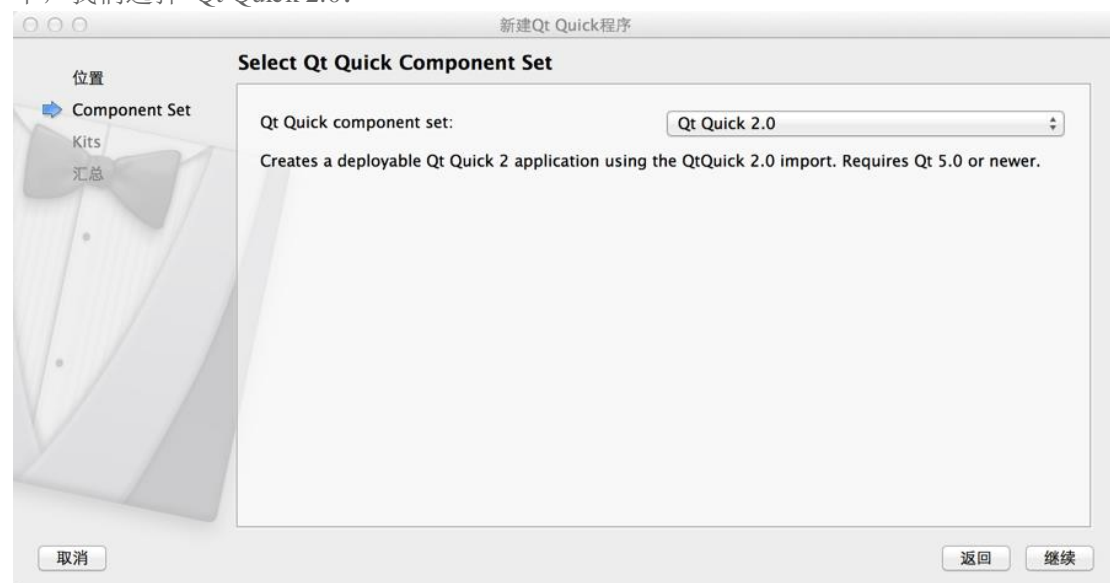
前面我们已经了解了 Qt 的一部分内容。这部分内容全部集中在 C++ 方面。也就是说，至今为止我们的程序都是使用 C++ 语言完成的。这在 Qt 5 之前的版本中是唯一的途径。不过，自从 Qt 5 开始，情况有了变化。事实上，从 Qt 4.7 开始，Qt 引入了一种声明式脚本语言，称为 QML（Qt Meta Language 或者 Qt Modeling Language），作为 C++ 语言的一种替代。而 Qt Quick 就是使用 QML 构建的一套类库。

QML 是一种基于 JavaScript 的声明式语言。在 Qt 5 中，QML 有了长足进步，并且同 C++ 并列成为 Qt 的首选编程语言。也就是说，使用 Qt 5，我们不仅可以使用 C++ 开发 Qt 程序，而且可以使用 QML。虽然 QML 是解释型语言，性能要比 C++ 低一些，但是新版 QML 使用 V8，Qt 5.2 又引入了专为 QML 优化的 V4 引擎，使得其性能不再有明显降低。在 Nokia 发布 Qt 4.7 的时候，QML 被用于开发手机应用程序，全面支持触摸操作、流畅的动画效果等。但是在 Qt 5 中，QML 已经不仅限于开发手机应用，也可以用户开发传统的桌面程序。

QML 文档描述了一个对象树。QML 元素包含了其构造块、图形元素（矩形、图片等）和行为（例如动画、切换等）。这些 QML 元素按照一定的嵌套关系构成复杂的组件，供用户交互。

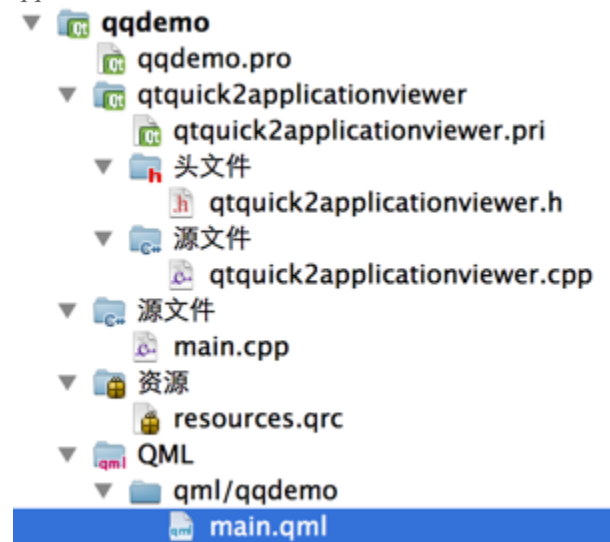
本章我们先来编写一个简单的 QML 程序，了解 QML 的基本概念。需要注意的是，这里的 Qt Quick 使用的是 Qt Quick 2 版本，与 Qt 4.x 提供的 Qt Quick 1 不兼容。

首先，使用 Qt Creator 创建一个 Qt Quick Application。在之后的 Qt Quick Component 选项中，我们选择 Qt Quick 2.0：





完成创建之后，我们可以看到一个 QML 项目所需要的基本文件。这里的项目名字为 qqdemo，而且有一个自己添加的资源文件，所以这个示意图可能与你的不同。



qtquick2applicationviewer 里面的内容是 Qt Creator 帮助我们生成的，这里面的文件一般不会被修改。我们可以认为这个文件夹下的文件就是运行 QML 的一个加载器。Application Viewer 用于加载 QML 文件并将其解释显示，类似于浏览器解释运行网页。

我们真正关心的是 main.qml 里面的内容：

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

这段代码看起来很简单，事实也的确如此。一个 QML 文档分为 `import` 和 `declaration` 两部分。前者用于引入文档中所需要的组件（有可能是类库，也可以是一个 JavaScript 文件或者另外的 QML 文件）；后者用于声明本文档中的 QML 元素。

第一行，我们使用 `import` 语句引入 `QtQuick 2.0`。由于这只是一个示例程序，所以我们没有引入更多的内容。

每一个 QML 有且只有一个根元素，类似于 XML 文档。这个根元素就是这个 QML 文档中定义的 QML 元素，在这个例子中就是一个 `Rectangle` 对象。注意一下这个 QML 文档的具体语法，非常类似于 JSON 的定义，使用键值对的形式区分元素属性。所以我们能够很清楚看到，我们定义了一个矩形，宽度为 360 像素，高度为 360 像素。记得我们说过，QML 文档定义了一个对象树，所以 QML 文档中元素是可以嵌套的。在这个矩形中，我们又增加了一个 `Text` 元素，顾名思义，就是一个文本。`Text` 显示的是 `Hello World` 字符串，而这个字符串是由 `qsTr()` 函数返回的。`qsTr()` 函数就是 `QObject::tr()` 函数的 QML 版本，用于返回可翻译的字符串。`Text` 的位置则是由锚点（`anchor`）定义。示例中的 `Text` 位置定义为 `parent` 中心，其中 `parent` 属性就是这个元素所在的外部的元素。同理，我们可以看到 `MouseArea` 是充满父元素的。`MouseArea` 还有一个 `onClicked` 属性。这是一个回调，也就是鼠标点击事件。`MouseArea` 可以看作是可以相应鼠标事件的区域。当点击事件发出时，就会执行 `onClicked` 中的代码。这段代码其实是让整个程序退出。注意我们的 `MouseArea` 充满整个矩形，所以整个区域都可以接受鼠标事件。

当我们运行这个项目时，我们就可以看到一个宽和高都是 360 像素的矩形，中央有一行文本，鼠标点击矩形任意区域就会使其退出：



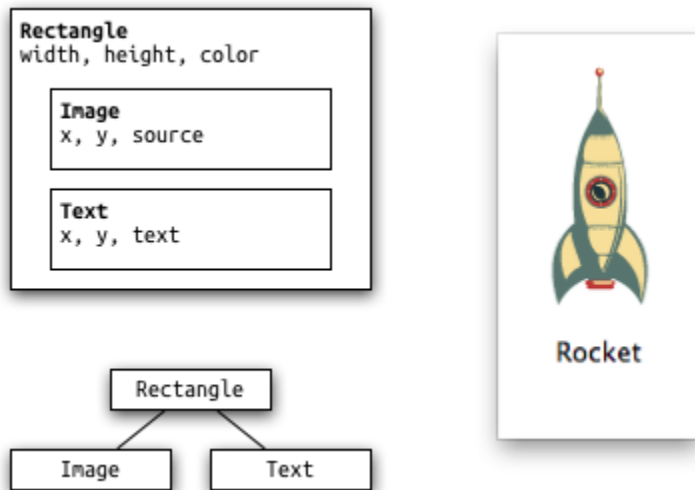
Hello World

接下来我们可以改变 `main.qml` 文件中的“Hello World”字符串，不重新编译直接运行，就会看到运行结果也会相应的变化。这说明 QML 文档是运行时解释的，不需要经过编译。所以，利用 QML 的解释执行的特性，QML 尤其适合于快速开发和原型建模。另外，由于 QML 比 C++ 简单很多，所以 QML 也适用于提供插件等机制。

## 77 QML 语法

前面我们已经见识过 QML 文档。一个 QML 文档分为 `import` 和对象声明两部分。如果你要使用 Qt Quick，就需要 `import QtQuick 2`。QML 是一种声明语言，用于描述程序界面。QML 将用户界面分解成一块块小的元素，每一元素都由很多组件构成。QML 定义了用户界面元素的外观和行为；更复杂的逻辑则可以结合 JavaScript 脚本实现。这有点类似于 HTML 和 JavaScript 的关系，前者用来显示界面，后者用来定义行为。我们这部分文章有些来自于 [QmlBook](#)，在此表示感谢！

QML 在最简单的元素关系是层次关系。子元素处于相对于父元素的坐标系统中。也就是说，子元素的 `x` 和 `y` 的坐标值始终相对于父元素。这一点比起 Graphics View Framework 要简单得多。



下面我们使用一个简单的示例文档来了解 QML 的语法：

```
// rectangle.qml
import QtQuick 2.0
// 根元素: Rectangle
Rectangle {
    // 命名根元素
    id: root // 声明属性: <name>: <value>
    width: 120; height: 240
    color: "#D8D8D8" // 颜色属性
    // 声明一个嵌套元素（根元素的子元素）
    Image {
        id: rocket
        x: (parent.width - width)/2; y: 40 // 使用 parent 引用父元素
        source: 'assets/rocket.png'
    }
    // 根元素的另一个子元素
    Text {
        // 该元素未命名
        y: rocket.y + rocket.height + 20 // 使用 id 引用元素
        width: root.width // 使用 id 引用元素
        horizontalAlignment: Text.AlignHCenter
        text: 'Rocket'
    }
}
```

第一个需要注意的是 import 语句。前面我们简单介绍过，QML 文档总要有 import 部分，用于指定该文档所需要引入的模块。通常这是一个模块名和版本号，比如这里的 QtQuick 2.0。当然，我们也可以引入自己的模块或者其他文件，具体细节会在后面的章节中详细介绍。

QML 文档的第二个部分是 QML 元素。一个 QML 文档有且只有一个根元素，类似 XML 文档的规定。在我们的例子中，这个根元素就是 **Rectangle** 元素。QML 元素使用 {} 包围起来。{} 之中是该元素的属性；属性以键值对 **name : value** 的形式给出。这十分类似与 JSON 语法。QML 元素可以有一个 **id** 属性，作为该元素的名字。以后我们可以直接用这个名字指代该元素，相当于该元素的指针。需要注意的是，**id** 属性在整个 QML 文档中必须是唯一的。QML 元素允许嵌套，一个 QML 元素可以没有、可以有一个或多个子元素。子元素可以使用 **parent** 关键字访问其父元素。正如上面的例子中显示的那样，我们可以用 **id**，也可以用 **parent** 关键字访问其他元素。一个最佳实践是，将根元素的 **id** 命名为 **root**。这样我们就可以很方便地访问到根元素。

QML 文档的注释使用 // 或者 /\* \*/。这同 C/C++ 或者 JavaScript 是一致的。

QML 元素的属性就是键值对，这同 JSON 是一致的。属性是一些预定义的类型，也可以有自己的初始值。比如下面的代码：

```
Text {
    // (1) 标识符
    id: thisLabel
    // (2) x、y 坐标
    x: 24; y: 16
    // (3) 绑定
    height: 2 * width
    // (4) 自定义属性
    property int times: 24
    // (5) 属性别名
    property alias anotherTimes: times
    // (6) 文本和值
    text: "Greetings " + times
    // (7) 字体属性组
    font.family: "Ubuntu"
    font.pixelSize: 24
    // (8) 附加属性 KeyNavigation
    KeyNavigation.tab: otherLabel
    // (9) 属性值改变的信号处理回调
    onHeightChanged: console.log('height:', height)
    // 接收键盘事件需要设置 focus
    focus: true
    // 根据 focus 值改变颜色
    color: focus?"red":"black"
}
```

标识符 **id** 用于在 QML 文档中引用这个元素。**id** 并不是一个字符串，而是一个特殊的标识符类型，这是 QML 语法的一部分。如前文所述，**id** 在文档中必须是唯一的，并且一旦

指定，不允许重新设置为另外的元素。因此，`id` 很像 C++ 的指针。和指针类似，`id` 也不能以数字开头，具体规则同 C++ 指针的命名一致。`id` 看起来同其它属性没有什么区别，但是，我们不能使用 `id` 反查出具体的值。例如，`aElement.id` 是不允许的。

元素 `id` 应该在 QML 文档中是唯一的。实际上，QML 提供了一种动态作用域（dynamic-scoping）的机制，后加载的文档会覆盖掉前面加载的文档的相同 `id`。这看起来能够“更改”`id` 的指向，其意义是构成一个 `id` 的查询链。如果当前文档没有找到这个 `id`，那么可以在之前加载的文档中找到。这很像全局变量。不过，这种代码很难维护，因为这种机制意味着你的代码依赖于文档的加载顺序。不幸的是，我们没有办法关闭这种机制。因此，在选用 `id` 时，我们一定要注意唯一性这个要求，否则很有可能出现一些很难调试的问题。

属性的值由其类型决定。如果一个属性没有给值，则会使用属性的默认值。我们可以通过查看文档找到属性默认值究竟是什么。

属性可以依赖于其它属性，这种行为叫作绑定。绑定类似信号槽机制。当所依赖的属性发生变化时，绑定到这个属性的属性会得到通知，并且自动更新自己的值。例如上面的 `height: 2 * width`。`height` 依赖于 `width` 属性。当 `width` 改变时，`height` 会自动发生变化，将自身的值更新为 `width` 新值的两倍。`text` 属性也是一个绑定的例子。注意，`int` 类型的属性会自动转换成字符串；并且在值变化时，绑定依然成立。

系统提供的属性肯定是不够的。所以 QML 允许我们自定义属性。我们可以使用 `property` 关键字声明一个自定义属性，后面是属性类型和属性名，最后是属性值。声明自定义属性的语法是 `property <type> <name> : <value>`。如果没有默认值，那么将给出系统类型的默认值。

我们也可以声明一个默认属性，例如：

```
// MyLabel.qml
import QtQuick 2.0
Text {
    default property var defaultText
    text: "Hello, " + defaultText.text
}
```

在 `MyLabel` 中，我们声明了一个默认属性 `defaultText`。注意这个属性的类型是 `var`。这是一种通用类型，可以保存任何类型的属性值。

默认属性的含义在于，如果一个子元素在父元素中，但是没有赋值给父元素的任何属性，那么它就成了这个默认属性。利用上面的 `MyLabel`，我们可以有如下的代码：

```
MyLabel {
    Text { text: "world" }
```

```
}
```

MyLabel.qml 实际可以直接引入到另外的 QML 文档，当做一个独立的元素使用。所以，我们可以把 MyLabel 作为根元素。注意 MyLabel 的子元素 Text 没有赋值给 MyLabel 的任何属性，所以，它将自动成为默认属性 defaultText 的值。因此，上面的代码其实等价于：

```
MyLabel {  
    defaultText: Text { text: "world" }  
}
```

如果仔细查看代码你会发现，这种默认属性的写法很像嵌套元素。其实嵌套元素正是利用这种默认属性实现的。所有可以嵌套元素的元素都有一个名为 data 的默认属性。所以这些嵌套的子元素都是添加到了 data 属性中。

属性也可以有别名。我们使用 alias 关键字声明属性的别名：property alias <name> : <reference>。别名和引用类似，只是给一个属性另外一个名字。C++ 教程里面经常说，“引用即别名”，这里就是“别名即引用”。这种技术对于导出属性非常有用。例如，我们希望让一个子元素的属性外部可用，那么就可以给这个属性一个别名，让外部文档通过这个别名访问这个属性。别名不需要特别声明属性类型，它使用被引用属性的类型或者 id。需要注意的是，属性别名在组件完全初始化之后才可用。因此，下面的代码是非法的：

```
property alias myLabel: label  
myLabel.text: "error" // 错误！此时组件还没有初始化  
property alias myLabelText: myLabel.text // 错误！不能为属性别名的属性创建别名
```

属性也可以分组。分组可以让属性更具结构化。上面示例中的 font 属性另外一种写法是：

```
font { family: "Ubuntu"; pixelSize: 24 }
```

有些属性可以附加到元素本身，其语法是<Element>.<property>: <value>。

每一个属性都可以发出信号，因而都可以关联信号处理函数。这个处理函数将在属性值变化时调用。这种值变化的信号槽命名为 on + 属性名 + Changed，其中属性名要首字母大写。例如上面的例子中，height 属性变化时对应的槽函数名字就是 onHeightChanged。

QML 和 JavaScript 关系密切。我们将在后面的文章中详细解释，不过现在可以先看个简单的例子：

```
Text {  
    id: label  
    x: 24; y: 24  
    // 自定义属性，表示空格按下的次数
```

```

property int spacePresses: 0
text: "Space pressed: " + spacePresses + " times"
// (1) 文本变化的响应函数
onTextChanged: console.log("text changed to:", text)
// 接收键盘事件，需要设置 focus 属性
focus: true
// (2) 调用 JavaScript 函数
Keys.onSpacePressed: {
    increment()
}
// 按下 Esc 键清空文本
Keys.onEscapePressed: {
    label.text = ''
}
// (3) 一个 JavaScript 函数
function increment() {
    spacePresses = spacePresses + 1
}
}

```

Text 元素会发出 `textChanged` 信号。我们使用 `on + 信号名`，信号名首字母大写的属性表示一个槽函数。也就是说，当 Text 元素发出 `textChanged` 信号时，`onTextChanged` 就会被调用。类似的，`onSpacePressed` 属性会在空格键按下时被调用。此时，我们调用了 JavaScript 函数。

QML 文档中可以定义 JavaScript 函数，语法同普通 JavaScript 函数一样。

QML 的绑定机制同 JavaScript 的赋值运算符有一定类似。它们都可以将右面的值赋值给前面。不同之处在于，绑定会在后面的值发生改变时，重新计算前面的值；但是赋值只是一次性的。

## 78 QML 基本元素

QML 基本元素可以分为可视元素和不可视元素两类。可视元素（例如前面提到过的 `Rectangle`）具有几何坐标，会在屏幕上占据一块显示区域。不可视元素（例如 `Timer`）通常提供一种功能，这些功能可以作用于可视元素。

本章我们将会集中介绍集中最基本的可视元素：`Item`、`Rectangle`、`Text`、`Image` 和 `MouseArea`。



**Item** 是所有可视元素中最基本的一个。它是所有其它可视元素的父元素，可以说是所有其它可视元素都继承 **Item**。**Item** 本身没有任何绘制，它的作用是定义所有可视元素的通用属性：

分组	属性
----	----

几何	<b>x</b> 和 <b>y</b> 用于定义元素左上角的坐标， <b>width</b> 和 <b>height</b> 则定义了元素的范围。 <b>z</b> 定义了元素上下的层叠关系。
----	--

布局	<b>anchors</b> （具有 <b>left</b> 、 <b>right</b> 、 <b>top</b> 、 <b>bottom</b> 、 <b>vertical</b> 和 <b>horizontal center</b> 等属性）用于定位元素相对于其它元素的 <b>margins</b> 的位置。
----	--

键盘处理	<b>Key</b> 和 <b>KeyNavigation</b> 属性用于控制键盘； <b>focus</b> 属性则用于启用键盘处理，也就是获取焦点。
------	---

变形	提供 <b>scale</b> 和 <b>rotate</b> 变形以及更一般的针对 <b>x</b> 、 <b>y</b> 、 <b>z</b> 坐标值变换以及 <b>transformOrigin</b> 点的 <b>transform</b> 属性列表。
----	--

可视化	<b>opacity</b> 属性用于控制透明度； <b>visible</b> 属性用于控制显示/隐藏元素； <b>clip</b> 属性用于剪切元素； <b>smooth</b> 属性用于增强渲染质量。
-----	---

状态定义	提供一个由状态组成的列表 <b>states</b> 和当前状态 <b>state</b> 属性；同时还有一个 <b>transitions</b> 列表，用于设置状态切换时的动画效果。
------	---

前面我们说过，**Item** 定义了所有可视元素都具有的属性。所以在下面的内容中，我们会再次详细介绍这些属性。

除了定义通用属性，**Item** 另外一个重要作用是作为其它可视元素的容器。从这一点来说，**Item** 非常类似于 **HTML** 中 **div** 标签的作用。

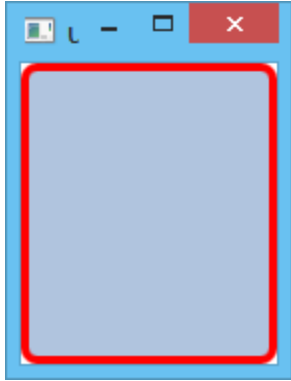
**Rectangle** 继承了 **Item**，并在 **Item** 的基础之上增加了填充色属性、边框相关的属性。为了定义圆角矩形，**Rectangle** 还有一个 **radius** 属性。下面的代码定义了一个宽 100 像素、高 150 像素，浅金属蓝填充，红色 4 像素的边框的矩形：

```
Rectangle {  
  id: rect  
  width: 100
```

```

height: 150
color: "lightsteelblue"
border {
    color: "#FF0000"
    width: 4
}
radius: 8
}

```



QML 中的颜色值可以使用颜色名字，也可以使用 # 十六进制的形式。这里的颜色名字同 SVG 颜色定义一致，具体可以参见[这个网页](#)。

Rectangle 除了 color 属性之外，还有一个 gradient 属性，用于定义使用渐变色填充。例如：

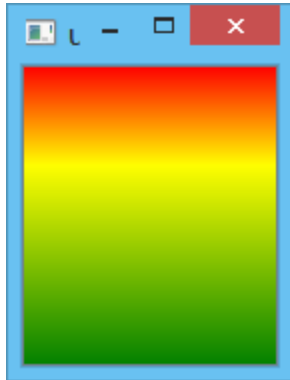
```

Rectangle {
    width: 100
    height: 150
    gradient: Gradient {
        GradientStop { position: 0.0; color: "red" }
        GradientStop { position: 0.33; color: "yellow" }
        GradientStop { position: 1.0; color: "green" }
    }
    border.color: "slategray"
}

```

gradient 要求一个 Gradient 对象。该对象需要一个 GradientStop 的列表。我们可以这样理解渐变：所谓渐变，就是我们指定在某个位置必须是某种颜色，这期间的过渡色则由计算而得。GradientStop 对象就是用于这种指定，它需要两个属性：position 和 color。前者是一个 0.0 到 1.0 的浮点数，说明 y 轴方向的位置，例如元素的最顶部是 0.0，最底部是 1.0，介于最顶和最底之间的位置可以用这么一个浮点数表示，也就是一个比例；后者是这个位置的颜色值。例如上面的 GradientStop { position: 0.33; color: "yellow" } 说明在从上往下三分之一处是黄色。当前最新版本的 QML (Qt 5.2) 只支持 y 轴方向的渐变，如果需要 x

轴方向的渐变，则需要执行旋转操作，我们会在后文说明。另外，当前版本 QML 也不支持角度渐变。如果你需要角度渐变，那么最好选择一张事先制作的图片。这段代码的执行结果如下：

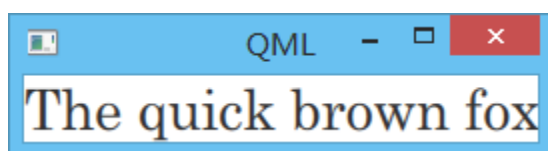


需要注意的是，**Rectangle** 必须同时指定（显式地或隐式地）宽和高，否则的话是不能在屏幕上显示出来的。这通常是一个常见的错误。

如果需要显示文本，你需要使用 **Text** 元素。**Text** 元素最重要的属性当然就是 **text** 属性。这个属性类型是 **string**。**Text** 元素会根据文本和字体计算自己的初始宽度和高度。字体则可以通过字体属性组设置（例如 **font.family**、**font.pixelSize** 等）。如果要设置文本颜色，只需要设置 **color** 属性。**Text** 最简单的使用如下：

```
Text {
    text: "The quick brown fox"
    color: "#303030"
    font.family: "Century"
    font.pixelSize: 28
}
```

运行结果：



**Text** 元素中的文本可以使用 **horizontalAlignment** 和 **verticalAlignment** 属性指定对齐方式。为了进一步增强文本渲染，我们还可以使用 **style** 和 **styleColor** 两个属性。这两个属性允许我们指定文本的显示样式和这些样式的颜色。对于很长的文本，通常我们会选择在文本末尾使用 **...**，此时我们需要使用 **elide** 属性。**elide** 属性还允许你指定 **...** 的显示位置。

如果不希望使用这种显示方式，我们还可以选择通过 `wrapMode` 属性指定换行模式。例如下面的代码：

```
Text {
    width: 160
    height: 120
    text: "A very very long text"
    elide: Text.ElideMiddle
    style: Text.Sunken
    styleColor: '#FF4444'
    verticalAlignment: Text.AlignTop
    font {
        pixelSize: 24
    }
}
```

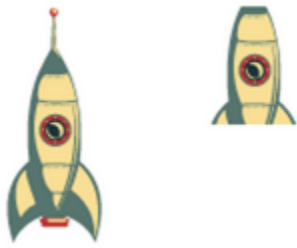
这里的 `Text` 元素的文本省略号位置这一行文本的中部；具有一个 `#FF4444` 颜色的样式 `Sunken`。

`Text` 元素的作用是显示文本。它不会显示文本的任何背景，这是另外的元素需要完成的事情。

`Image` 元素则用于显示图像。目前 `QML` 支持的图像格式有 `PNG`、`JPG`、`GIF` 和 `BMP` 等。除此之外，我们也可以直接给 `source` 属性一个 `URL` 来自动从网络加载图片，也可以通过 `fillMode` 属性设置改变大小的行为。例如下面代码片段：

```
Image {
    x: 12;
    y: 12
    // width: 48
    // height: 118
    source: "assets/rocket.png"
}

Image {
    x: 112;
    y: 12
    width: 48
    height: 118/2
    source: "assets/rocket.png"
    fillMode: Image.PreserveAspectCrop
    clip: true
}
```



注意这里我们说的 URL，可以是本地路径（./images/home.png），也可以使网络路径（http://example.org/home.png）。这也是 QML 的一大特色：网络透明。如果还记得先前我们尝试做的那个天气预报程序，那时候为了从网络加载图片，我们费了很大的精力。但是在 QML 中，这都不是问题。如果一个 URL 是网络的，QML 会自动从这个地址加载对应的资源。

上面的代码中，我们使用了 `Image.PreserveAspectCrop`，意思是等比例切割。此时，我们需要同时设置 `clip` 属性，避免所要渲染的对象超出元素范围。

最后一个我们要介绍的基本元素是 `MouseArea`。顾名思义，这个元素用于用户交互。这是一个不可见的矩形区域，用于捕获鼠标事件。我们在前面的例子中已经见过这个元素。通常，我们会将这个元素与一个可视元素结合起来使用，以便这个可视元素能够与用户交互。例如：

```
Rectangle {
    id: rect1
    x: 12;
    y: 12
    width: 76;
    height: 96
    color: "lightsteelblue"
    MouseArea {
        /* ~~ */
    }
}
```

`MouseArea` 是 QtQuick 的重要组成部分，它将可视化展示与用户输入控制解耦。通过这种技术，你可以显示一个较小的元素，但是它有一个很大的可交互区域，以便在界面显示与用户交互之间找到一个平衡（如果在移动设备上，较小的区域非常不容易被用户成功点击。苹果公司要求界面的交互部分最少要有 40 像素以上，才能够很容易被手指点中）。

## 79 QML 组件

前面我们简单介绍了几种 QML 的基本元素。QML 可以由这些基本元素组合成一个复杂的元素，方便以后我们的重用。这种组合元素就被称为组件。组件就是一种可重用的元素。QML 提供了很多方法来创建组件。不过，本章我们只介绍一种方式：基于文件的组件。基于文件的组件将 QML 元素放置在一个单独的文件中，然后给这个文件一个名字。以后我们就可以通过这个名字来使用这个组件。例如，如果有一个文件名为 `Button.qml`，那么，我们就可以在 QML 中使用 `Button { ... }` 这种形式。

下面我们通过一个例子来演示这种方法。我们要创建一个带有文本说明的 `Rectangle`，这个矩形将成为一个按钮。用户可以点击矩形来响应事件。

```
import QtQuick 2.0

Rectangle {
    id: root

    property alias text: label.text
    signal clicked

    width: 116; height: 26
    color: "lightsteelblue"
    border.color: "slategrey"

    Text {
        id: label
        anchors.centerIn: parent
        text: "Start"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            root.clicked()
        }
    }
}
```

我们将这个文件命名为 `Button.qml`，放在 `main.qml` 同一目录下。这里的 `main.qml` 就是 IDE 帮我们生成的 QML 文件。此时，我们已经创建了一个 QML 组件。这个组件其实就是一个预定义好的 `Rectangle`。这是一个按钮，有一个 `Text` 用于显示按钮的文本；有一个 `MouseArea` 用于接收鼠标事件。用户可以定义按钮的文本，这是用过设置 `Text` 的 `text` 属性实现的。为了不对外暴露 `Text` 元素，我们给了它的 `text` 属性一个别名。signal `clicked` 给这个 `Button` 一个信号。由于这个信号是无参数的，我们也可以写成 `signal clicked()`，二

者是等价的。注意，这个信号会在 `MouseArea` 的 `clicked` 信号被发出，具体就是在 `MouseArea` 的 `onClicked` 属性中调用个这个信号。

下面我们需要修改 `main.qml` 来使用这个组件：

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    Button {
        id: button
        x: 12; y: 12
        text: "Start"
        onClicked: {
            status.text = "Button clicked!"
        }
    }

    Text {
        id: status
        x: 12; y: 76
        width: 116; height: 26
        text: "waiting ..."
        horizontalAlignment: Text.AlignHCenter
    }
}
```

在 `main.qml` 中，我们直接使用了 `Button` 这个组件，就像 `QML` 其它元素一样。由于 `Button.qml` 与 `main.qml` 位于同一目录下，所以不需要额外的操作。但是，如果我们将 `Button.qml` 放在不同目录，比如构成如下的目录结果：

```
app
|- QML
|   |- main.qml
|- components
|   |- Button.qml
```

那么，我们就需要在 `main.qml` 的 `import` 部分增加一行 `import ../components` 才能够找到 `Button` 组件。

有时候，选择一个组件的根元素很重要。比如我们的 `Button` 组件。我们使用 `Rectangle` 作为其根元素。`Rectangle` 元素可以设置背景色等。但是，有时候我们并不允许用户设置背景

色。所以，我们可以选择使用 **Item** 元素作为根。事实上，**Item** 元素作为根元素会更常见一些。