

本文出自“[豆子空间](http://devbean.blog.51cto.com/448512/194031)”博客，请务必保留此出处
<http://devbean.blog.51cto.com/448512/194031>

Qt 学习之路(1): 前言

Qt 是一个著名的 C++ 库——或许并不能说这只是一个 GUI 库，因为 Qt 十分庞大，并不仅仅是 GUI。使用 Qt，在一定程度上你获得的是一个“一站式”的服务：不再需要研究 STL，不再需要 C++ 的 `<string>`，因为 Qt 有它自己的 `QString` 等等。或许这样说很偏激，但 Qt 确实是一个“伟大的 C++ 库”。

我们所使用的 Qt，确切地说也就是它的 GUI 编程部分。C++ 的 GUI 编程同 Java 不同：GUI 并不是 C++ 标准的一部分。所以，如果使用 Java，那么你最好的选择就是 AWT/Swing，或者也可以使 SWT/JFace，但是，C++ 的 GUI 编程给了你更多的选择：wxWidget, gtk++ 以及 Qt。这几个库我都有接触，但是接触都不是很多，只能靠一些资料和自己的一点粗浅的认识说一下它们之间的区别(PS：更详尽的比较在[前面的文章](#)中有)。

首先说 wxWidget，这是一个标准的 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 wxWidget 上面来。wxWidget 有一个很大的优点，就是它的界面都是原生风格的。这是其他的库所不能做到的。wxWidget 的运行效率很高，据说在 Windows 平台上比起微软自家的 MFC 也不相上下。

gtk++ 其实是一个 C 库，不过由于 C++ 和 C 之间的关系，这点并没有很大的关系。但是，gtk++ 是一个使用 C 语言很优雅的实现了面向对象程序设计的范例。不过，这也同样带来了一个问题——它的里面带有大量的类型转换的宏来模拟多态，并且它的函数名“又臭又长(不过这点我倒觉得无所谓，因为它的函数名虽然很长，但是同样很清晰)”，使用下划线分割单词，看上去和 Linux 如出一辙。由于它是 C 语言实现，因此它的运行效率当然不在话下。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显得格格不入。

再来看 Qt，和 wxWidget 一样，它也是一个标准的 C++ 库。但是它的语法很类似于 Java 的 Swing，十分清晰，而且 SIGNAL/SLOT 机制使得程序看起来很明白——这也是我首先选择 Qt 的一个很重要的方面，因为我是学 Java 出身的 :)。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 `qmake` 对程序进行预处理，才能够再使用 `make` 或者 `nmake` 进行编译。并且它的界面也不是原生风格的，尽管 Qt 使用 `style` 机制十分巧妙的模拟了本地界面。另外值得一提的是，Qt 不仅仅运行在桌面环境中，Qt 已经被 Nokia 收购，它现在已经会成为 Symbian 系列的主要界面技术——Qt 是能够运行于嵌入式平台的。

以往人们对 Qt 的授权多有诟病。因为 Qt 的商业版本价格不菲，开源版本使用的是 GPL 协议。但是现在 Qt 的开源协议已经变成 LGPL。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，现在的 Qt 协议的争议已经不存在了——因为 wxWidgets 或者 gtk+ 同样使用的是类似的协议发布的。

在本系列文章中，我们将使用 Qt4 进行 C++ GUI 的开发。我是参照着《C++ GUI Programming with Qt4》一书进行学习的。其实，我也只是初学 Qt4，在这里将这个学习笔记记下来，希望能够方便更多的朋友学习 Qt4。我是一个 Java 程序员，感觉 Qt4 的一些命名

规范以及约束同 Java 有异曲同工之妙，因而从 Java 迁移到 Qt4 似乎困难不大。不过，这也主要是因为 Qt4 良好的设计等等。

闲话少说，还是尽快开始下面的学习吧！

Qt 学习之路(2): Hello, world!

何编程技术的学习第一课基本上都会是 Hello, world!，我也不想故意打破这个惯例——照理说，应该首先回顾一下 Qt 的历史，不过即使不说这些也并无大碍。

或许有人总想知道，Qt 这个单词是什么意思。其实，这并不是一个缩写词，仅仅是因为它的发明者，TrollTech 公司的 CEO, Haard Nord 和 Trolltech 公司的总裁 Eirik Chambe-Eng 在联合发明 Qt 的时候并没有一个很好的名字。在这里，字母 Q 是 Qt 库中所有类的前缀——这仅仅是因为在 Haard 的 emacs 的字体中，这个字母看起来特别的漂亮；而字母 t 则代表“toolkit”，这是在 Xt(X toolkit)中得到的灵感。

顺便说句，Qt 原始的公司就是上面提到的 Trolltech，貌似有一个中文名字是奇趣科技——不过现在已经被 Nokia 收购了。因此，一些比较旧的文章里面会提到 Trolltech 这个名字。

好了，闲话少说，先看看 Qt 的开发吧！事先说明一下，我是一个比较懒的人，不喜欢配置很多的东西，而 Qt 已经提供了一个轻量级的 IDE，并且它的网站上也有 for Eclipse 和 VS 的开发插件，不过在这里我并不想用这些大块头 😊

Qt 有两套协议——商业版本和开源的 LGPL 版本。不同的是前者要收费，而后者免费，当然，后者还要遵循 LGPL 协议的规定，这是题外话。

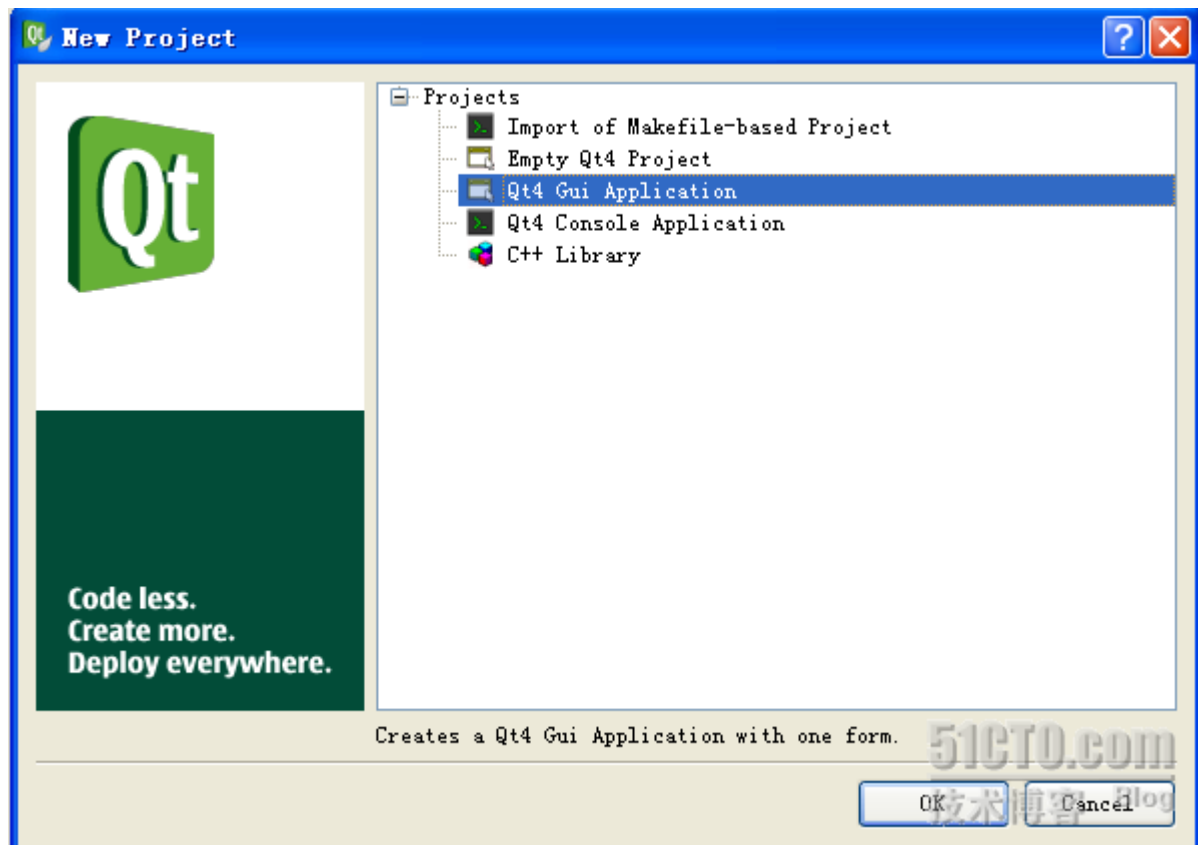
Qt 的网址是 <https://qt.nokia.com/downloads>，不过我打开这个站点总是很慢，不知道为什么。你可以找到大大的 LGPL/Free 和 Commercial，好了，我选的是 LGPL 版本的，下载包蛮大，但是下载并不会很慢。下载完成后安装就可以了，其它不用管了。这样，整个 Qt 的开发环境就装好了——如果你需要的话，也可以把 qmake 所在的目录添加进环境变量，不过我就不做了。

安装完成后会有个 Qt Creator 的东西，这就是官方提供的一个轻量级 IDE，不过它的功能还是蛮强大的。运行这个就会发现，其实 Qt 不仅仅是 Linux KDE 桌面的底层实现库。而且就是这个 IDE 的实现 😊 这个 IDE 就是用 Qt 完成的。

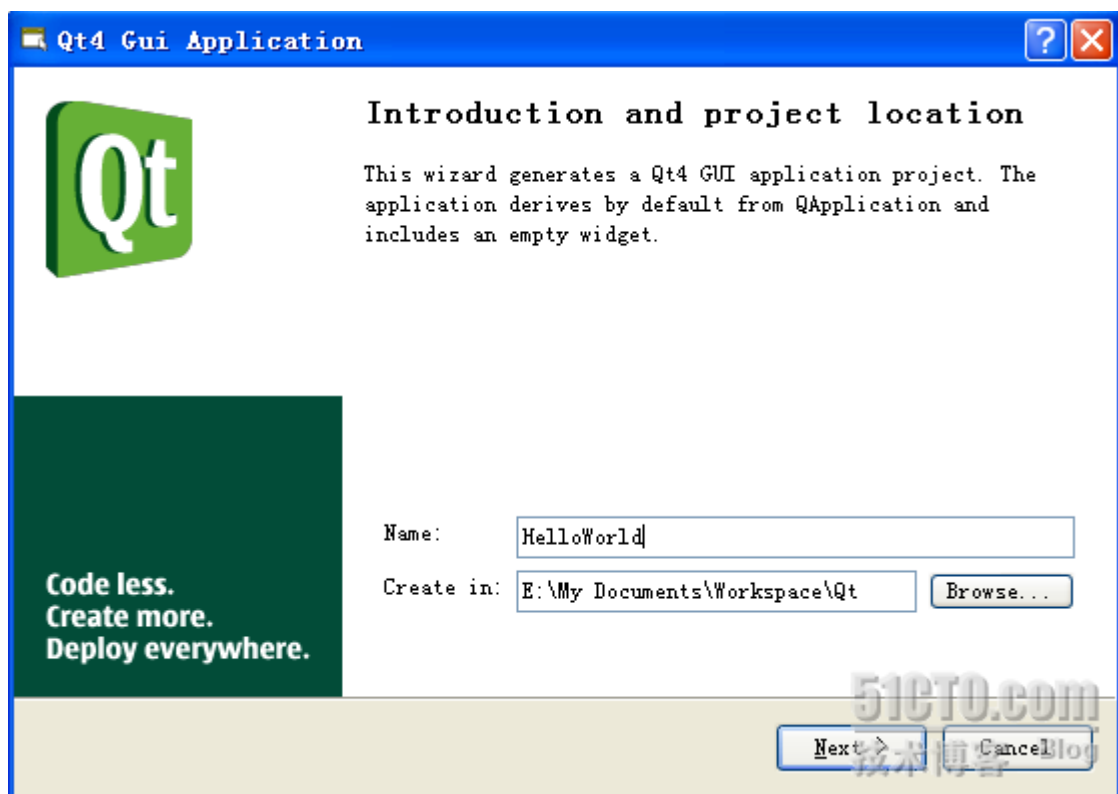
Qt Creator 左面从上到下依次是 Welcome(欢迎页面，就是一开始出现的那个)；Edit(我们的代码编辑窗口)；Debug(调试窗口)；Projects(工程窗口)；Help(帮助，这个帮助完全整合的 Qt 的官方文档，相当有用)；Output(输出窗口)。

下面我们来试试我们的 Hello, world! 吧！

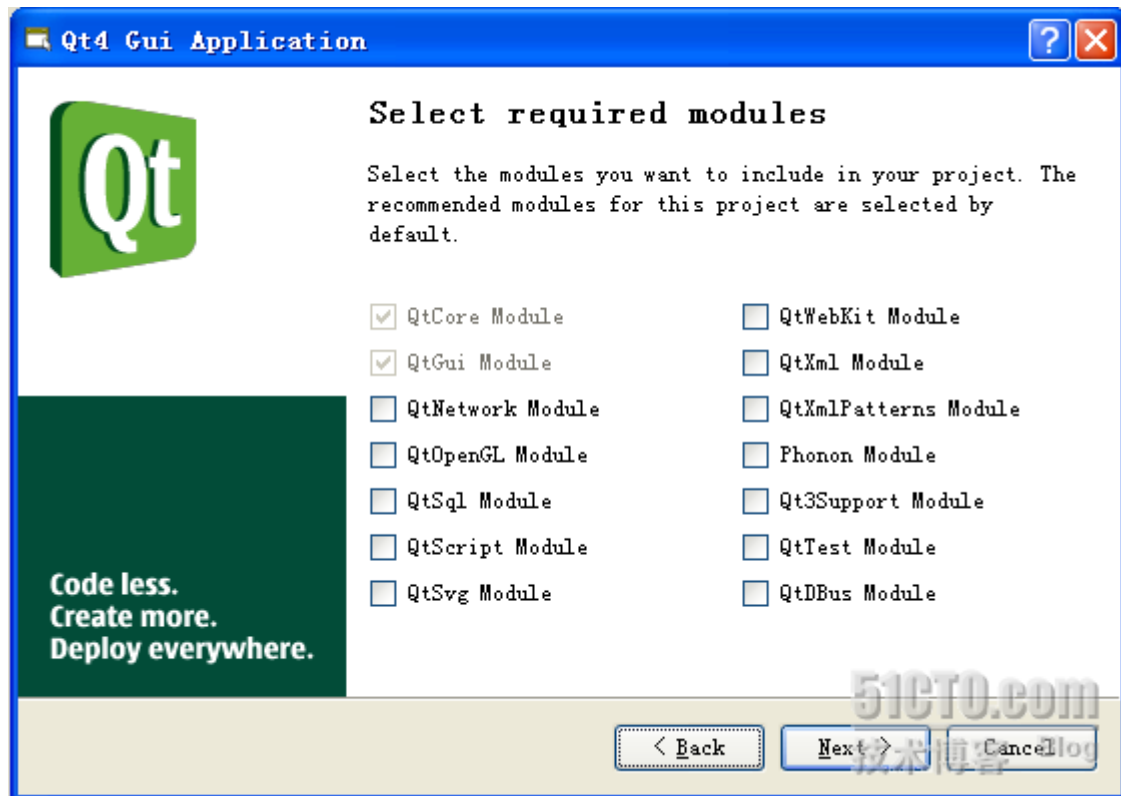
在 Edit 窗口空白处点右键，有 New project... 这里我们选第三项，Qt Gui Application。



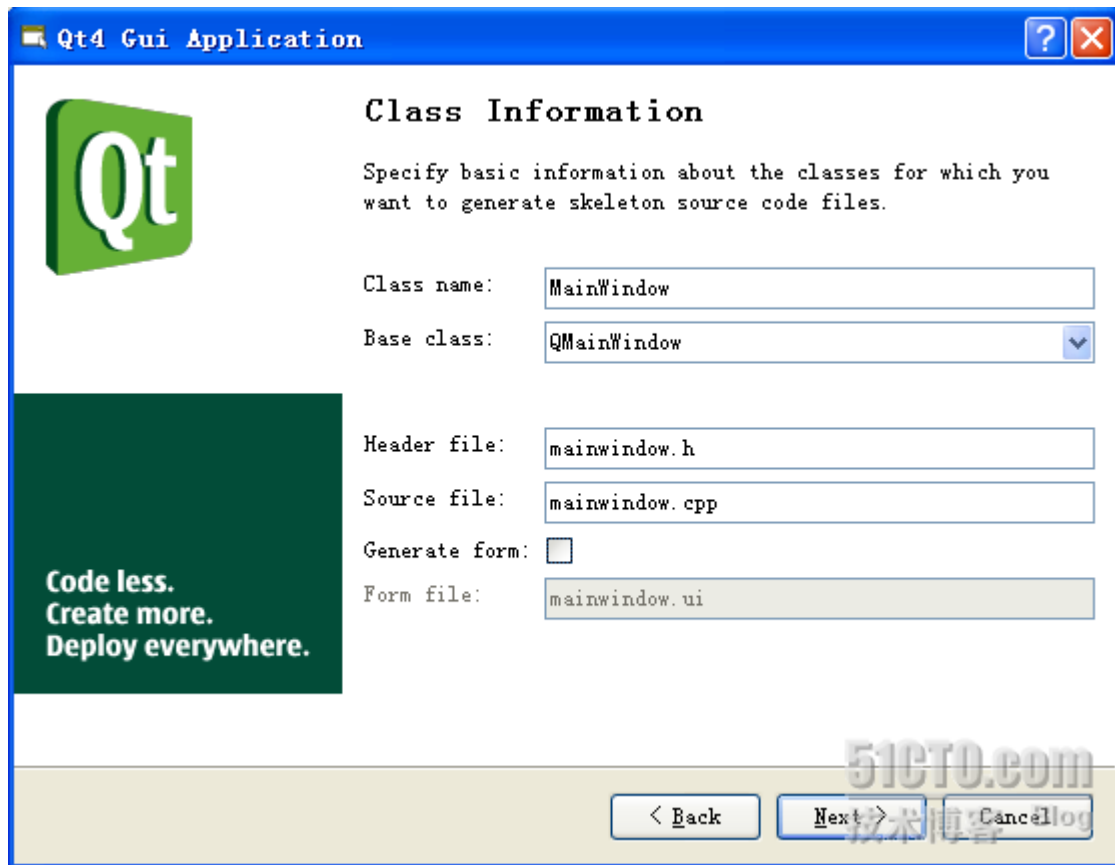
然后点击 OK，来到下一步，输入工程名字和保存的位置



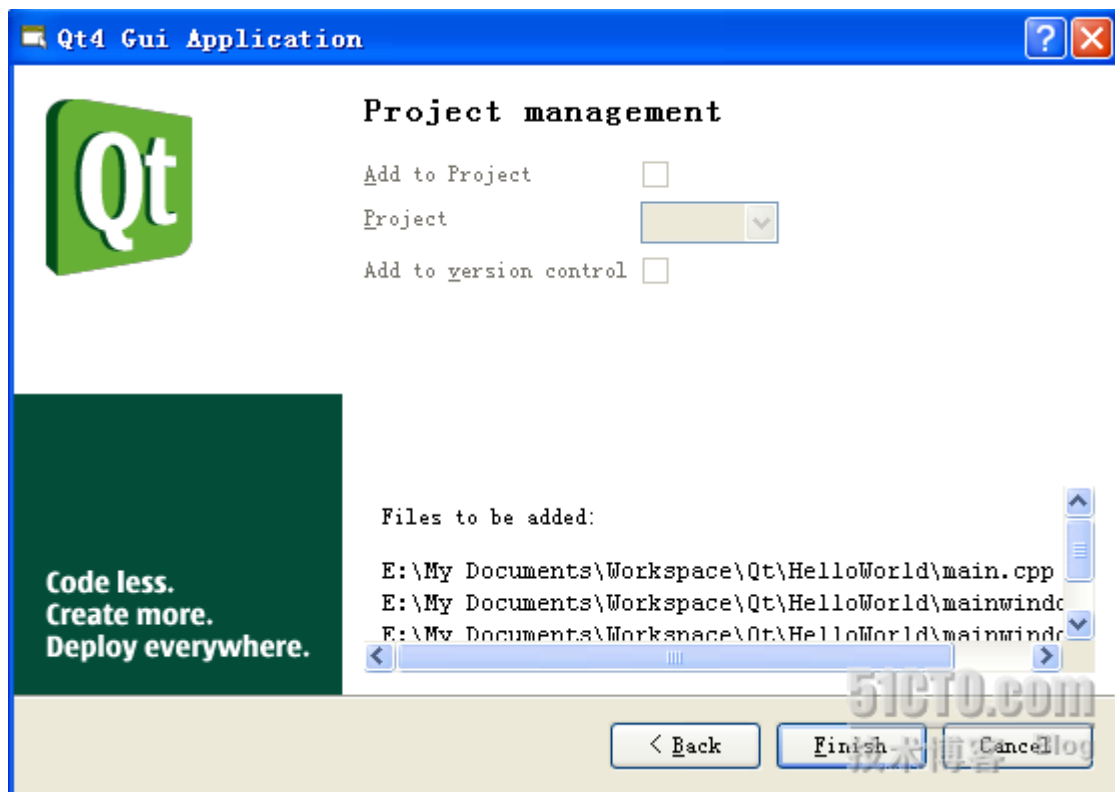
点击 Next，来到选择库的界面。这里我们系统默认为我们选择了 Qt core 和 GUI，还记得我们建的是 Gui Application 吗？嗯，就是这里啦，它会自动为我们加上 gui 这个库。现在应该就能看出，Qt 是多么庞大的一个库，它不仅仅有 Gui，而且有 Network，OpenGL，XML 之类。不过，现在在这里我们不作修改，直接 Next。



下一个界面需要我们定义文件名，我们不修改默认的名字，只是为了清除起见，把 generate form 的那个勾去掉即可。

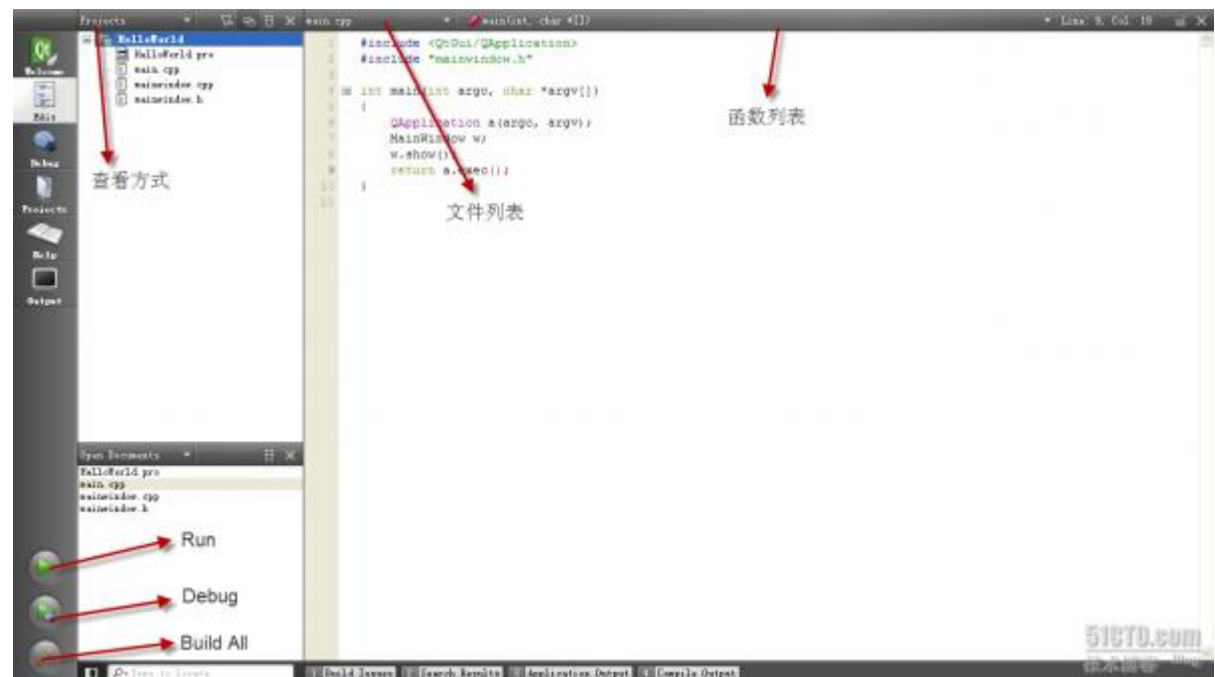


Next 之后终于到了 Finish 了——漫长的一系列啊！检查无误后 Finish 就好啦！



之后可以看到，IDE 自动生成了四个文件，一个.pro 文件，两个.cpp 和一个.h。这里说明一下，.pro 就是工程文件(project)，它是 qmake 自动生成的用于生产 makefile 的配置文件。

这里我们先不去管它。`main.cpp` 里面就是一个 `main` 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。



现在，我们把 `main.cpp` 中的代码修改一下：

```
#include "QtGui/QApplication"
#include "QLabel"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QLabel *label = new QLabel("Hello, world!");
    label->show();
    return a.exec();
}
```

好了！我们的第一个 Qt 程序已经完成了。

PS：截了很多图，说得详细些，以后可就没这么详细的步骤啦，嘿嘿...相信很多朋友应该一下子就能看明白这个 IDE 应该怎么使用的了，无需我多费口舌。呵呵。

下一篇中，将会对这个 `Hello, world!` 做一番逐行解释！

Qt 学习之路(3): Hello, world!(续)

面来逐行解释一下前面的那个 **Hello, world!** 程序，尽管很简单，但却可以对 Qt 程序的结构有一个清楚的认识。现在再把代码贴过来：

```
#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();
}
```

第 1 行和第 2 行就是需要引入的头文件。和普通的 C++ 程序没有什么两样，如果要使用某个组件，就必须引入相应的头文件，这类似于 Java 的 **import** 机制。值得说明的是，Qt 中头文件和类名是一致的。也就是说，如果你要使用某个类的话，它的类名就是它的头文件名。

第 3 行是空行 :)

第 4 行是 **main** 函数函数头。这与普通的 C++ 程序没有什么两样，学过 C++ 的都明白。因此你可以看到，实际上，Qt 完全通过普通的 **main** 函数进入，这不同于 **wxWidgets**，因为 **wxWidgets** 的 **Hello, world** 需要你继承它的一个 **wxApp** 类，并覆盖它的 **wxApp::OnInit** 方法，系统会自动将 **OnInit** 编译成入口函数。不过在 Qt 中，就不需要这些了。

第 5 行，噢噢，大括号...

第 6 行，创建一个 **QApplication** 对象。这个对象用于管理应用程序级别的资源。**QApplication** 的构造函数要求两个参数，分别来自 **main** 的那两个参数，因此，Qt 在一定程度上是支持命令行参数的。

第 7 行，创建一个 **QLabel** 对象，并且能够显示 **Hello, world!** 字符串。和其他库的 **Label** 控件一样，这是用来显示文本的。在 Qt 中，这被称为一个 **widget** (翻译出来是小东西，不过这个翻译并不好...)，它等同于 Windows 技术里面的控件 (**controls**) 和容器 (**containers**)。也就是说，**widget** 可以放置其他的 **widget**，就像 **Swing** 的组件。大多数 Qt 程序使用 **QMainWindow** 或者 **QDialog** 作为顶级组件，但 Qt 并不强制要求这点。在这个例子中，顶级组件就是一个 **QLabel**。

第 8 行，使这个 **label** 可见。组件创建出来之后通常是不可见的，要求我们手动的使它们可见。这样，在创建出组建之后我们就可以对它们进行各种定制，以避免出现之后在屏幕上面会有闪烁。

第 9 行，将应用程序的控制权移交给 Qt。这时，程序的事件循环就开始了，也就是说，这时可以相应你发出的各种事件了。这类似于 g++ 最后的一行 `gtk_main()`。

第 10 行，大括号.....程序结束了。

注意，我们并没有使用 `delete` 去删除创建的 `QLabel`，因为在程序结束后操作系统会回收这个空间——这只是因为这个 `QLabel` 占用的内存比较小，但有时候这么做会引起麻烦的，特别是在大程序中，因此必须小心。

好了，程序解释完了。按照正常的流程，下面应该编译。前面也提过，Qt 的编译不能使用普通的 `make`，而必须先使用 `qmake` 进行预编译。所以，第一步应该是在工程目录下使用

```
qmake -project
```

命令创建 `.pro` 文件(比如说是叫 `helloworld.pro`)。然后再在 `.pro` 文件目录下使用

```
qmake helloworld.pro (make)
```

或者

```
qmake -tp vc helloworld.pro (nmake)
```

生成 `makefile`，然后才能调用 `make` 或者是 `nmake` 进行编译。不过因为我们使用的是 IDE，所以这些步骤就不需要我们手动完成了。

值得说明一点的是，这个 `qmake` 能够生成标准的 `makefile` 文件，因此完全可以利用 `qmake` 自动生成 `makefile`——这是题外话。

好了，下面修改一下源代码，把 `QLabel` 的创建一句改成

```
QLabel *label = new QLabel("<h2><font color='red'>Hello</font>, world!<h2>");
```

运行一下：

51CTO.com

本图片来自51CTO博客 Blog.51cto.com

若未注明出处则为非法转载

同 Swing 的 JLabel 一样，Qt 也是支持 HTML 解析的。

好了，这个 Hello, world 就说到这里！明确一下 Qt 的程序结构，在一个 Qt 源代码中，一下两条语句是必不可少的：

```
QApplication app(argc, argv);  
//...  
return app.exec();
```

本文出自“[豆子空间](http://devbean.blog.51cto.com/448512/194137)”博客，请务必保留此出处

<http://devbean.blog.51cto.com/448512/194137>

Qt 学习之路(4)：初探信号槽

看过了简单的 Hello, world! 之后，下面来看看 Qt 最引以为豪的信号槽机制！

所谓信号槽，简单来说，就像是插销一样：一个插头和一个插座。怎么说呢？当某种事件发生之后，比如，点击了一下鼠标，或者按了某个按键，这时，这个组件就会发出一个信号。就像是广播一样，如果有了事件，它就漫天发声。这时，如果有一个槽，正好对应上这个信号，那么，这个槽的函数就会执行，也就是回调。就像广播发出了，如果你感兴趣，那么你就会对这个广播有反应。干巴巴的解释很无力，还是看代码：

```
#include <QtGui/QApplication>  
#include <QtGui/QPushButton>  
  
int main(int argc, char *argv[])  
{  
    QApplication a(argc, argv);  
    QPushButton *button = new QPushButton("Quit");  
    QObject::connect(button, SIGNAL(clicked()), &a, SLOT(quit()));  
    button->show();  
    return a.exec();  
}
```

这是在 Qt Creator 上面新建的文件，因为前面已经详细的说明怎么新建工程，所以这里就不再赘述了。这个程序很简单，只有一个按钮，点击之后程序退出。（顺便说一句，Qt 里面的 button 被叫做 QPushButton，真搞不明白为什么一个简单的 button 非得加上 push 呢？呵呵）

主要是看这一句：

```
QObject::connect(button, SIGNAL(clicked()), &a, SLOT(quit()));
```

QObject 是所有类的根。**Qt** 使用这个 **QObject** 实现了一个单根继承的 **C++**。它里面有一个 **connect** 静态函数，用于连接信号槽。

当一个按钮被点击时，它会发出一个 **clicked** 信号，意思是，向周围的组件们声明：我被点击啦！当然，其它很多组件都懒得理他。如果对它感兴趣，就告诉 **QObject** 说，你帮我盯着点，只要 **button** 发出 **clicked** 信号，你就告诉我——想了想之后，说，算了，你也别告诉我了，直接去执行我的某某函数吧！就这样，一个信号槽就形成了。具体来说呢，这个例子就是 **QApplication** 的实例 **a** 说，如果 **button** 发出了 **clicked** 信号，你就去执行我的 **quit** 函数。所以，当我们点击 **button** 的时候，**a** 的 **quit** 函数被调用，程序退出了。所以，在这里，**clicked()** 就是一个信号，而 **quit()** 就是槽，形象地说就是把这个信号插进这个槽里面去。

Qt 使用信号槽机制完成了事件监听操作。这类似与 **Swing** 里面的 **listener** 机制，只是要比这个 **listener** 简单得多。以后我们会看到，这种信号槽的定义也异常的简单。值得注意的是，这个信号槽机制仅仅是使用的 **QObject** 的 **connect** 函数，其他并没有什么耦合——也就是说，完全可以利用这种机制实现你自己的信号监听！不过，这就需要使用 **qmake** 预处理一下了！

细心的你或许发现，在 **Qt Creator** 里面，**SIGNAL** 和 **SLOT** 竟然变颜色了！没错，**Qt** 确实把它们当成了关键字！实际上，**Qt** 正是利用它们扩展了 **C++** 语言，因此才需要使用 **qmake** 进行预处理，比便使普通的 **C++** 编译器能够顺利编译。另外，这里的 **signal** 和 **Unix** 系统里面的 **signal** 没有任何的关系！哦哦，有一点关系，那就是名字是一样的！

信号槽机制是 **Qt** 关键部分之一，以后我们还会再仔细的探讨这个问题的。

Qt 学习之路(5)：组件布局

同 **Swing** 类似，**Qt** 也提供了几种组件定位的技术。其中就包括绝对定位和布局定位。

顾名思义，绝对定位就是使用最原始的定位方法，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，以及怎么设置组件的大小了。但是这样做的一个问题是，如果用户改变了窗口大小，比如点击了最大化或者拖动窗口边缘，这时，你就要自己编写相应的函数来响应这些变化，以避免那些组件还只是静静地呆在一个角落。或者，更简单的方法是直接禁止用户改变大小。

不过，**Qt** 提供了另外一种机制，就是布局，来解决这个问题。你只要把组件放入某一种布局之中，当需要调整大小或者位置的时候，**Qt** 就知道怎样进行调整。这类似于 **Swing** 的布局管理器，不过 **Qt** 的布局没有那么多，只有有限的几个。

来看一下下面的例子：

```
#include <QtGui/QApplication>
#include <QtGui/QWidget>
```

```

#include <QtGui/QSpinBox>
#include <QtGui/QSlider>
#include <QtGui/QHBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Enter your age");

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

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

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

    window->show();

    return app.exec();
}

```

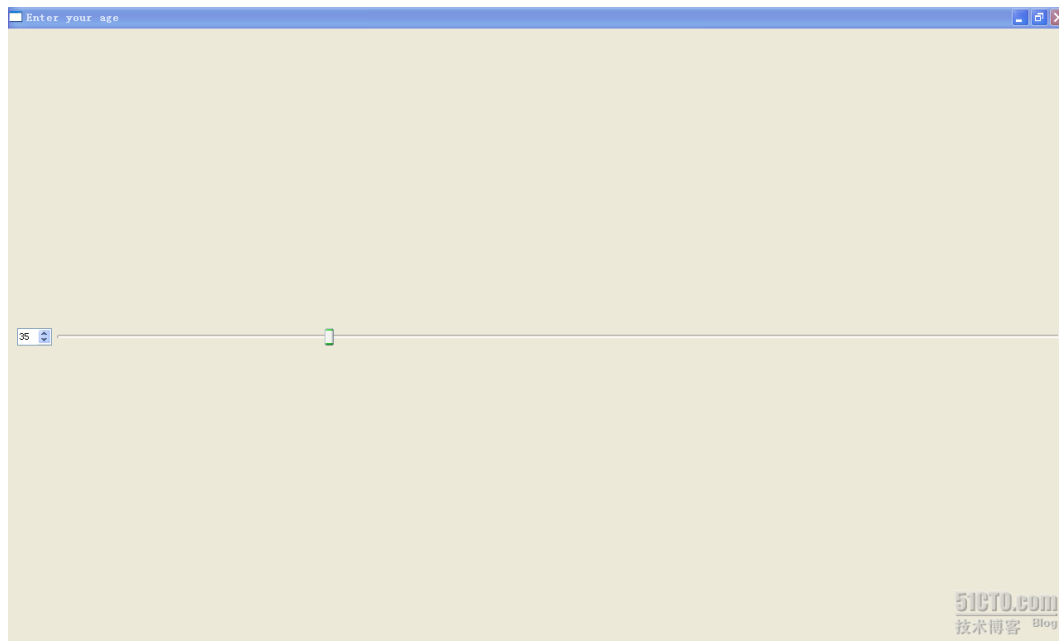
这里使用了两个新的组件：**QSpinBox** 和 **QSlider**，以及一个新的顶级窗口 **QWidget**。**QSpinBox** 是一个有上下箭头的微调器，**QSlider** 是一个滑动杆，只要运行一下就会明白到底是什么东西了。

代码并不是那么难懂，还是来简单的看一下。首先创建了一个 **QWidget** 的实例，调用 **setWindowTitle** 函数来设置窗口标题。然后创建了一个 **QSpinBox** 和 **QSlider**，分别设置了它们值的范围，使用的是 **setRange** 函数。然后进行信号槽的链接。这点后面再详细说明。然后是一个 **QHBoxLayout**，就是一个水平布局，按照从左到右的顺序进行添加，使用 **addWidget** 添加好组件后，调用 **QWidget** 的 **setLayout** 把 **QWidget** 的 **layout** 设置为我们定义的这个 **Layout**，这样，程序就完成了！

编译运行一下，可以看到效果：



如果最大化的话：



虽然我并没有添加任何代码，但是那个 **layout** 就已经明白该怎样进行布局。

或许你发现，那两个信号槽的链接操作会不会产生无限递归？因为 **setValue** 就会引发 **valueChanged** 信号！答案是不会。这两句语句实现了，当 **spinBox** 发出 **valueChanged** 信号的时候，会回调 **slider** 的 **setValue**，以更新 **slider** 的值；而 **slider** 发出 **valueChanged** 信号的时候，又会回调 **slider** 的 **setValue**。但是，如果新的 **value** 和旧的 **value** 是一样的话，是不会发出这个信号的，因此避免了无限递归。

迷糊了吧？举个例子看。比如下面的 **spinBox->setValue(35)** 执行的时候，首先，**spinBox** 会将自己的值设为 **35**，这样，它的值与原来的不一样了(在没有 **setValue** 之前的时候，默认值是 **0**)，于是它发出了 **valueChanged** 信号。**slider** 接收到这个信号，于是回调自己的 **setValue** 函数，将它的值也设置成 **35**，它也发出了 **valueChanged** 信号。当然，此时 **spinBox** 又收到了，不过它发现，这个 **35** 和它本身的价值是一样的，于是它就不发出信号，所以信号传递就停止了。

那么，你会问，它们是怎么知道值的呢？答案很简单，因为你的信号和槽都接受了一个 `int` 参数！新的值就是通过这个进行传递的。实际上，我们利用 Qt 的信号槽机制完成了一个数据绑定，使两个组件或者更多组件的状态能够同步变化。

Qt 一共有三种主要的 `layout`，分别是：

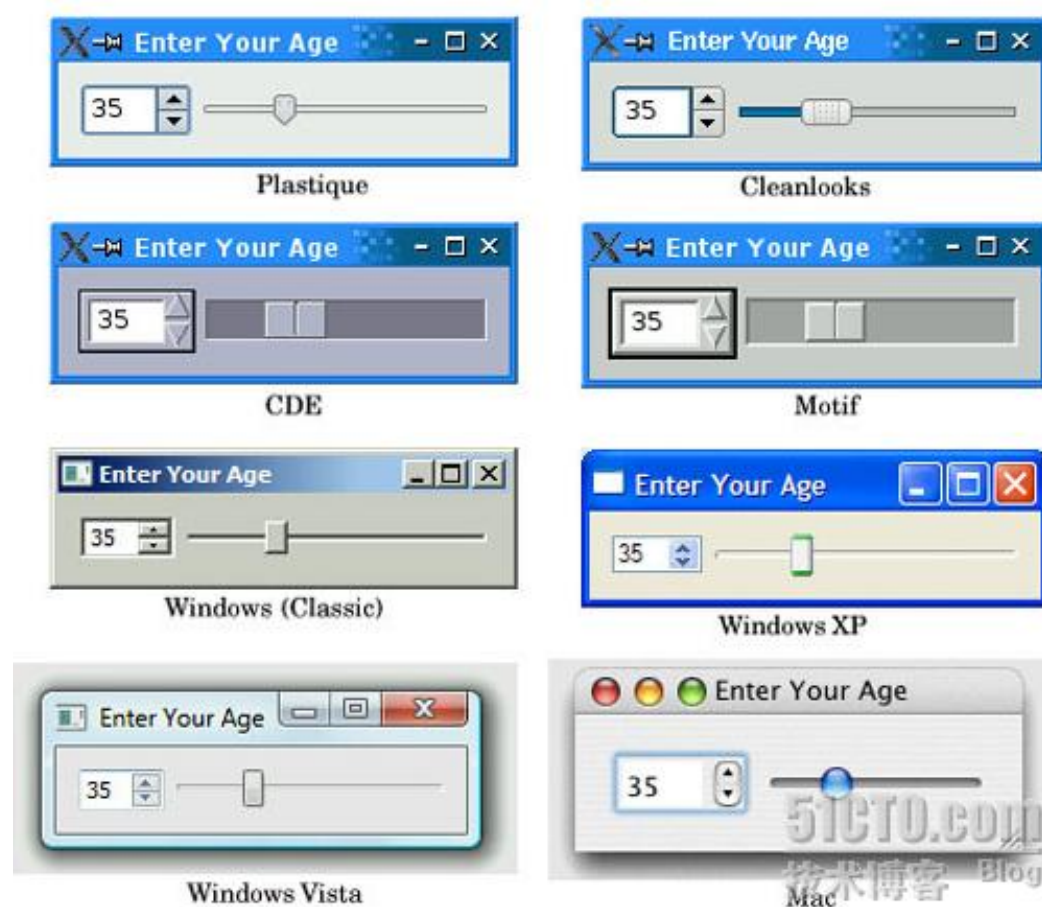
`QHBoxLayout`- 按照水平方向从左到右布局；

`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`。

`layout` 使用 `addWidget` 添加组件，使用 `addLayout` 可以添加子布局，因此，这就有了无穷无尽的组合方式。

我是在 Windows 上面进行编译的，如果你要是在其他平台上面，应用程序就会有不同的样子：



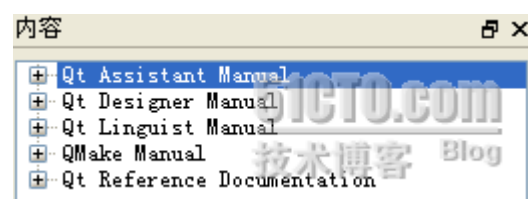
还记得前面曾经说过，Qt 不是使用的原生组件，而是自己绘制模拟的本地组件的样子，不过看看这个截图，它模拟的不能说百分百一致，也可说是惟妙惟肖了... :)

Qt 学习之路(6): API 文档的使用

今天来说一下有关 Qt API 文档的使用。因为 Qt 有一个商业版本，因此它的文档十分健全，而且编写良好。对于开发者来说，查看文档时开发必修课之一——没有人能够记住那么多 API 的使用！

在 Qt 中查看文档是一件很简单的事情。如果你使用 QtCreator，那么左侧的 Help 按钮就是文档查看入口。否则的话，你可以在 Qt 的安装目录下的 bin 里面的 assistant.exe 中看到 Qt 的文档。在早期版本中，Qt 的文档曾以 HTML 格式发布，不过在 2009.03 版中我没有找到 HTML 格式的文档，可能 Qt 已经把它全部换成二进制格式的吧？——当然，如果你全部安装了 Qt 的组件，是可以在开始菜单中找到 assistant 的！

assistant 里面的文档有很多项：



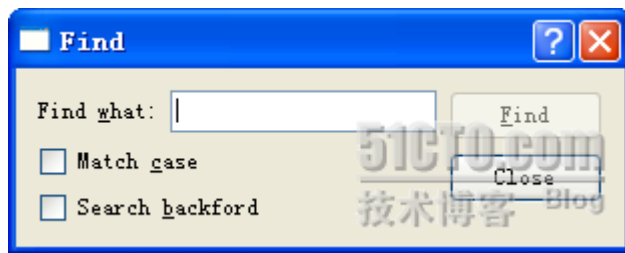
其中，第一个是帮助的帮助:-); 第二个是 Qt Designer 的帮助；第三个是 Qt Linguist 的帮助；第四个是 QMake 的帮助；最后一个是 Qt 的 API 文档，在 QtCreator 中默认打开的就是这部分。

不过，关于文档的内容这里实在不好阐述，因为整个文档太大了，我也并没有看过多少，很多时候都是随用随查，就好像是字典一样——谁也不会天天没事抱着本字典去看不是？还有就是这里的文档都是英文的，不过如果是做开发的话，了解一些英文还是很有帮助的，不是吗？

Qt 学习之路(7): 创建一个对话框(上)

首先说明一点，在 C++ GUI Programming with Qt4, 2nd 中，这一章连同以后的若干章一起，完成了一个比较完整的程序——一个模仿 Excel 的电子表格。不过这个程序挺大的，而且书中也没有给出完整的源代码，只是分段分段的——我不喜欢这个样子，我想要看到我写出来的是什么东西，这是最主要的，而不是慢慢的过上几章的内容才能看到自己的作品。所以，我打算换一种方式，每章只给出简单的知识，但是每章都能够运行出东西来。好了，扯完了，下面开始！

以前说的主要是一些基础知识，现在我们来真正做一个东西——一个查找对话框。什么？什么叫查找对话框？唉唉，先看看我们的最终作品吧！



好了，首先新建一个工程，就叫 FindDialog 吧！嗯，当然还是 Qt Gui Application，然后最后一步注意，Base Dialog 选择 QDialog，而不是默认的 QMainWindow，因为我们要学习建立对话框嘛！名字随便起，不过我就叫 finddialog 啦！Generate form 还是不要的。然后 Finish 就好了。

打开 finddialog.h，开始编写头文件。

```
#ifndef FINDDIALOG_H
#define FINDDIALOG_H

#include <QtGui/QDialog>

class QCheckBox;
class QLabel;
class QLineEdit;
class QPushButton;

class FindDialog : public QDialog
{
    Q_OBJECT

public:
    FindDialog(QWidget *parent = 0);
    ~FindDialog();

signals:
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

private slots:
    void findClicked();
    void enableFindButton(const QString &text);

private:
    QLabel *label;
    QLineEdit *lineEdit;
```



```
    QCheckBox *caseCheckBox;  
    QCheckBox *backwardCheckBox;  
    QPushButton *findButton;  
    QPushButton *closeButton;  
};  
  
#endif // FINDDIALOG_H
```

大家都是懂得 C++ 的啊，所以什么 `#ifndef`，`#define` 和 `#endif` 的含义和用途就不再赘述了。

首先，声明四个用到的类。这里做的是前向声明，否则的话是编译不过的，因为编译器不知道这些类是否存在。简单来说，所谓前向声明就是告诉编译器，我要用这几个类，而且这几个类存在，你就不要担心它们存不存在的问题啦！

然后是我们的 `FindDialog`，继承自 `QDialog`。

下面是一个重要的东西：`Q_OBJECT`。这是一个宏。凡是定义信号槽的类都必须声明这个宏。至于为什么，我们以后再说。

然后是 `public` 的构造函数和析构造函数声明。

然后是一个 `signal:`，这是 Qt 的关键字——还记得前面说过的嘛？Qt 扩展了 C++ 语言，因此它有自己的关键字——这是对信号的定义，也就是说，`FindDialog` 有两个 `public` 的信号，它可以在特定的时刻发出这两个信号，就这里来说，如果用户点击了 `Find` 按钮，并且选中了 `Search backward`，就会发出 `findPrevious()`，否则发出 `findNext()`。

紧接着是 `private slots:` 的定义，和前面的 `signal` 一样，这是私有的槽的定义。也就是说，`FindDialog` 具有两个槽，可以接收某些信号，不过这两个槽都是私有的。

为了 `slots` 的定义，我们需要访问 `FindDialog` 的组件，因此，我们把其中的组件定义为成员变量以便访问。正是因为需要定义这些组件，才需要对它们的类型进行前向声明。因为我们仅仅使用的是指针，并不涉及到这些类的函数，因此并不需要 `include` 它们的头文件——当然，你想直接引入头文件也可以，不过那样的话编译速度就会慢一些。

好了，头文件先说这些，下一篇再说源代码啦！休息，休息一下！

Qt 学习之路(8): 创建一个对话框(下)

接着上一篇，下面是源代码部分：


```

#include <QtGui>
#include "finddialog.h"

FindDialog::FindDialog(QWidget *parent)
    : QDialog(parent)
{
    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit;
    label->setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search &backford"));

    findButton = new QPushButton(tr("&Find"));
    findButton->setDefault(true);
    findButton->setEnabled(false);

    closeButton = new QPushButton(tr("Close"));

    connect(lineEdit, SIGNAL(textChanged(const QString&)), this, SLOT(enableFindButton(const QString&)));
    connect(findButton, SIGNAL(clicked()), this, SLOT(findClicked()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));

    QHBoxLayout *topLeftLayout = new QHBoxLayout;
    topLeftLayout->addWidget(label);
    topLeftLayout->addWidget(lineEdit);

    QVBoxLayout *leftLayout = new QVBoxLayout;
    leftLayout->addLayout(topLeftLayout);
    leftLayout->addWidget(caseCheckBox);
    leftLayout->addWidget(backwardCheckBox);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
    setLayout(mainLayout);

    setWindowTitle(tr("Find"));

```

```

        setFixedHeight(sizeHint().height());
    }

FindDialog::~FindDialog()
{
}

void FindDialog::findClicked()
{
    QString text = lineEdit->text();
    Qt::CaseSensitivity cs = caseCheckBox->isChecked() ? Qt::CaseInsensiti
ve : Qt::CaseSensitive;
    if(backwardCheckBox->isChecked()) {
        emit findPrevious(text, cs);
    } else {
        emit findNext(text, cs);
    }
}

void FindDialog::enableFindButton(const QString &text)
{
    findButton->setEnabled(!text.isEmpty());
}

```

C++ 文件要长一些哦——不过，它们的价钱也会更高，嘿嘿——嗯，来看代码，第一行 `include` 的是 `QtGui`。`Qt` 是分模块的，记得我们建工程的时候就会问你，使用哪些模块？`QtCore`？`QtGui`？`QtXml`？等等。这里，我们引入 `QtGui`，它包括了 `QtCore` 和 `QtGui` 模块。不过，这并不是最好的做法，因为 `QtGui` 文件很大，包括了 GUI 的所有组件，但是很多组件我们根本是用不到的——就像 `Swing` 的 `import`，你可以 `import` 到类，也可以使用 `*`，不过都不会建议使用 `*`，这里也是一样的。我们最好只引入需要的组件。不过，那样会把文件变长，现在就用 `QtGui` 啦，只要记得正式开发时不能这么用就好啦！

构造函数有参数初始化列表，用来调用父类的构造函数，相当于 `Java` 里面的 `super()` 函数。这是 `C++` 的相关知识，不是 `Qt` 发明的，这里不再赘述。

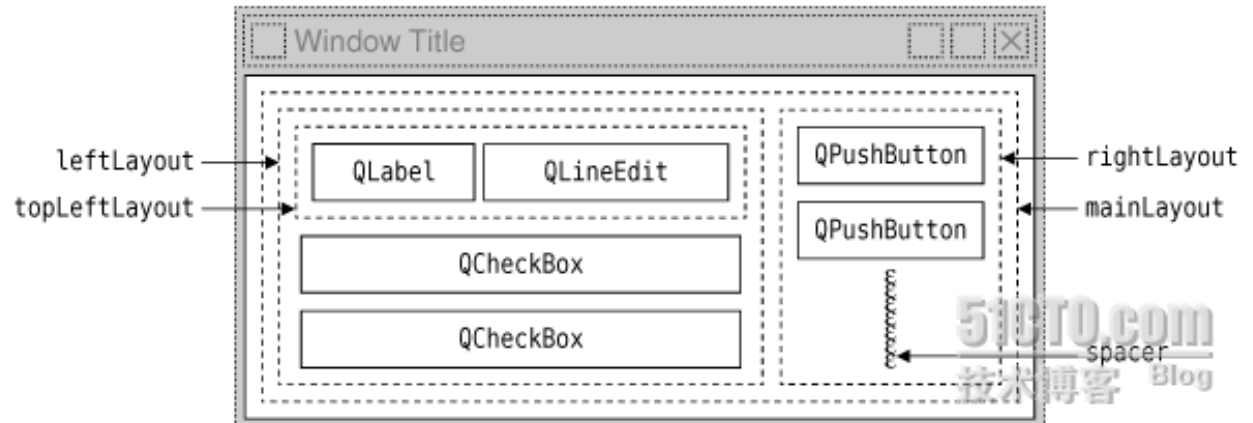
然后新建一个 `QLabel`。还记得前面的 `Hello, world!` 里面也使用过 `QLabel` 吗？那时候只是简单的传入一个字符串啊！这里怎么是一个函数 `tr()`？函数 `tr()` 全名是 `QObject::tr()`，被它处理的字符串可以使用工具提取出来翻译成其他语言，也就是做国际化使用。这以后还会仔细讲解，只要记住，**`Qt` 的最佳实践：如果你想让你的程序国际化的话，那么，所有用户可见的字符串都要使用 `QObject::tr()`！**但是，为什么我们没有写 `QObject::tr()`，而仅仅是 `tr()` 呢？原来，`tr()` 函数是定义在 `Object` 里面的，所有使用了 `Q_OBJECT` 宏的类都自动具有 `tr()` 函数。

字符串中的&代表快捷键。注意看下面的 `findButton` 的 `&Find`，它会生成 `F`ind 字符串，当你按下 `Alt+F` 的时候，这个按钮就相当于被点击——这么说很难受，相信大家都明白什么意思。同样，前面 `label` 里面也有一个&，因此它的快捷键就是 `Alt+W`。不过，这个 `label` 使用了 `setBuddy` 函数，它的意思是，当 `label` 获得焦点时，比如按下 `Alt+W`，它的焦点会自动传给它的 `buddy`，也就是 `lineEdit`。看，这就是伙伴的含义(`buddy` 英文就是伙伴的意思)。

后面几行就比较简单了：创建了两个 `QCheckBox`，把默认的按钮设为 `findButton`，把 `findButton` 设为不可用——也就是变成灰色的了。

再下面是三个 `connect` 语句，用来连接信号槽。可以看到，当 `lineEdit` 发出 `textChanged(const QString&)` 信号时，`FindDialog` 的 `enableFindButton(const QString&)` 函数会被调用——这就是回调，是有系统自动调用，而不是你去调用——当 `findButton` 发出 `clicked()` 信号时，`FindDialog` 的 `findClicked()` 函数会被调用；当 `closeButton` 发出 `clicked()` 信号时，`FindDialog` 的 `close()` 函数会被调用。注意，`connect()` 函数也是 `QObject` 的，因为我们继承了 `QObject`，所以能够直接使用。

后面的很多行语句都是 `layout` 的使用，虽然很复杂，但是很清晰——编写 `layout` 布局最重要一点就是思路清楚，想清楚哪个套哪个，就会很好编写。这里我们的对话框实际上是这个样子的：



注意那个 `spacer` 是由 `rightLayout` 的 `addStretch()` 添加的，就像弹簧一样，把上面的组件“顶起来”。

最后的 `setWindowTitle()` 就是设置对话框的标题，而 `setFixedHeight()` 是设置成固定的高度，其参数值 `sizeHint()` 返回“最理想”的大小，这里我们使用的是 `height()` 函数去到“最理想”的高度。

好了，下面该编写槽了——虽然说是 `slot`，但实际上它就是普通的函数，既可以和其他函数一样使用，又可以被系统回调。

先看 `findClicked()` 函数。首先取出 `lineEdit` 的输入值；然后判断 `caseCheckBox` 是不是选中，如果选中就返回 `Qt::CaseInsensitive`，否则返回 `Qt::CaseSensitive`，用于判断是不是大小写敏感的查找；最后，如果 `backwardCheckBox` 被选中，就 `emit`(发出)信号 `findPrevious()`，否则 `emit` 信号 `findNext`。

`enableFindButton()` 则根据 `lineEdit` 的内容是不是变化——这是我们的 `connect` 连接的——来设置 `findButton` 是不是可以使用，这个很简单，不再说了。

这样，`FindDialog.cpp` 也就完成了。下面编写 `main.cpp`——其实 `QtCreator` 已经替我们完成了——

```
#include <QApplication>

#include "finddialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    FindDialog *dialog = new FindDialog;
    dialog->show();
    return app.exec();
}
```

运行一下看看我们的成果吧！

虽然很简单，也没有什么实质性的功能，但是我们已经能够制作对话框了——Qt 的组件成百上千，不可能全部介绍完，只能用到什么学什么，更重要的是，我们已经了解了其编写思路，否则的话，即便是你拿着全世界所有的砖瓦，没有设计图纸，你也不知道怎么把它们组合成高楼大厦啊！

嘿嘿，下回见！

Qt 学习之路(9)：深入了解信号槽

信号槽机制是 Qt 编程的基础。通过信号槽，能够使 Qt 各组件在不知道对方的情形下能够相互通讯。这就将类之间的关系做了最大程度的解耦。

槽函数和普通的 C++ 成员函数没有很大的区别。它们也可以使 `virtual` 的；可以被重写；可以使 `public`、`protected` 或者 `private` 的；可以由其它的 C++ 函数调用；参数可以是任何类型的。如果说区别，那就是，槽函数可以和一个信号相连接，当这个信号发生时，它可以被自动调用。

`connect()`语句的原型类似于:

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

这里, `sender` 和 `receiver` 都是 `QObject` 类型的, `signal` 和 `slot` 都是没有参数名称的函数签名。`SIGNAL()`和 `SLOT()`宏用于把参数转换成字符串。

深入的说, 信号槽还有更多可能的用法, 如下所示。

一个信号可以和多个槽相连:

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

注意, 如果是这种情况, 这些槽会一个接一个的被调用, 但是它们的调用顺序是不确定的。

多个信号可以连接到一个槽:

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

这是说, 只要任意一个信号发出, 这个槽就会被调用。

一个信号可以连接到另外的一个信号:

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

这是说, 当第一个信号发出时, 第二个信号被发出。除此之外, 这种信号-信号的形式和信号-槽的形式没有什么区别。

槽可以被取消链接:

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

这种情况并不经常出现，因为当一个对象 `delete` 之后，Qt 自动取消所有连接到这个对象上面的槽。

为了正确的连接信号槽，信号和槽的参数个数、类型以及出现的顺序都必须相同，例如：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

这里有一种例外情况，如果信号的参数多于槽的参数，那么这个参数之后的那些参数都会被忽略掉，例如：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

这里，`const QString &`这个参数就会被槽忽略掉。

如果信号槽的参数不相容，或者是信号或槽有一个不存在，或者在信号槽的连接中出现了参数名字，在 `Debug` 模式下编译的时候，Qt 都会很智能的给出警告。

在这之前，我们仅仅在 `widgets` 中使用到了信号槽，但是，注意到 `connect()` 函数其实是在 `QObject` 中实现的，并不局限于 `GUI`，因此，只要我们继承 `QObject` 类，就可以使用信号槽机制了：

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }

public slots:
    void setSalary(int newSalary);

signals:
    void salaryChanged(int newSalary);

private:
    int mySalary;
};
```

在使用时，我们给出下面的代码：

```
void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

这样，当 `setSalary()` 调用的时候，就会发出 `salaryChanged()` 信号。注意这里的 `if` 判断，这是避免递归的方式！还记得前面提到的循环连接吗？如果没有 `if`，当出现了循环连接的时候就会产生无限递归。

Qt 学习之路(10): Meta-Object 系统

前面说过，Qt 使用的是自己的预编译器，它提供了对 C++ 的一种扩展。利用 Qt 的信号槽机制，就可以把彼此独立的模块相互连接起来，不需要实现知道模块的任何细节。

为了达到这个目的，Qt 提出了一个 Meta-Object 系统。它提供了两个关键的作用：信号槽和内省。

面向对象程序设计里面会讲到 Smalltalk 语言有一个元类系统。所谓元类，就是这里所说的 Meta-Class。如果写过 HTML，会知道 HTML 标签里面也有一个 `<meta>`，这是用于说明页面的某些属性的。同样，Qt 的 Meta-Object 系统也是类似的作用。内省又称为反射，允许程序在运行时获得类的相关信息，也就是 meta-information。什么是 meta-information 呢？举例来说，像这个类叫什么名字？它有什么属性？有什么方法？它的信号列表？它的槽列表？等等这些信息，就是这个类的 meta-information，也就是“元信息”。这个机制还提供了对国际化的支持，是 QSA(Qt Script for Application)的基础。

标准 C++ 并没有 Qt 的 meta-information 所需要的动态 meta-information。所以，Qt 提供了一个独立的工具，moc，通过定义 Q_OBJECT 宏实现到标准 C++ 函数的转变。moc 使用纯 C++ 实现的，因此可以在任何编译器中使用。

这种机制工作过程是：

首先，Q_OBJECT 宏声明了一些 QObject 子类必须实现的内省的函数，如 `metaObject()`，`tr()`，`qt_metacall()` 等；

第二，Qt 的 moc 工具实现 Q_OBJECT 宏声明的函数和所有信号；

第三，QObject 成员函数 `connect()` 和 `disconnect()` 使用这些内省函数实现信号槽的连接。

以上这些过程是 `qmake`, `moc` 和 `QObject` 自动处理的, 你不需要去考虑它们。如果实现好奇的话, 可以通过查看 `QMetaObject` 的文档和 `moc` 的源代码来一睹芳容。

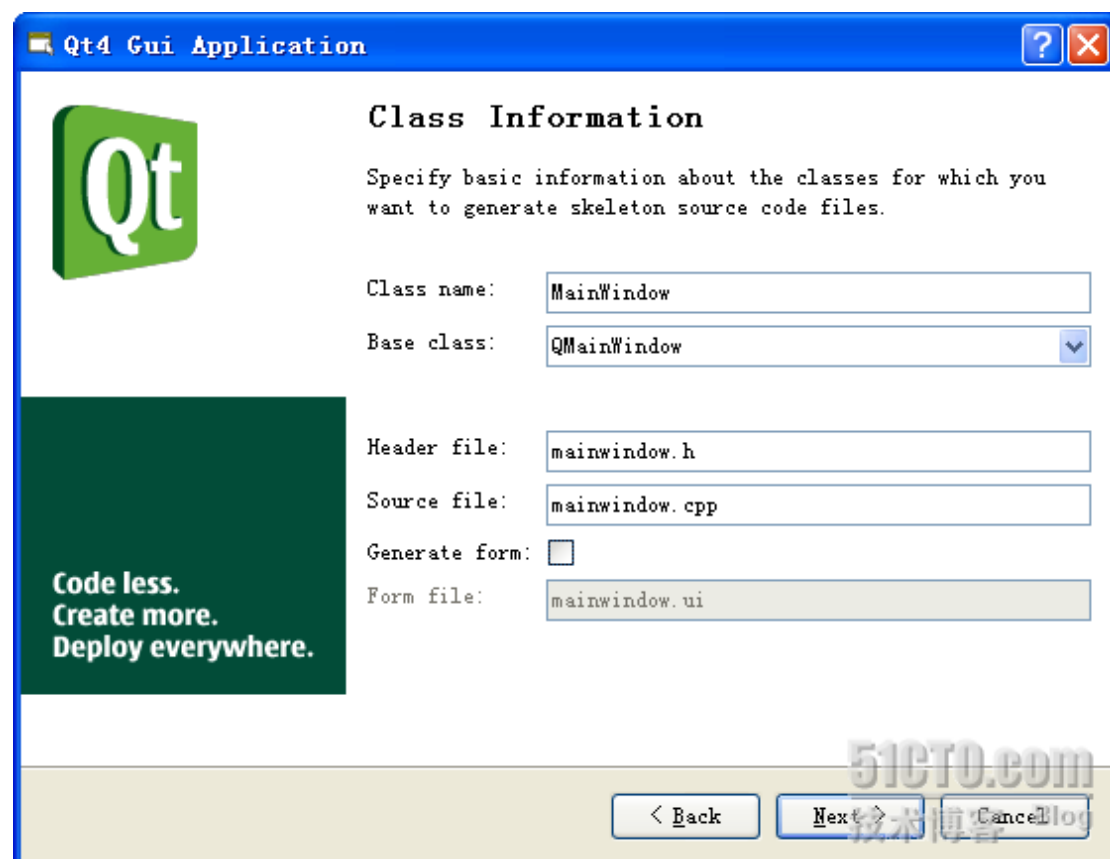
Qt 学习之路(11): MainWindow

尽管 Qt 提供了很方便的快速开发工具 `QtDesigner` 用来拖放界面元素, 但是现在我并不打算去介绍这个工具, 原因之一在于我们的学习大体上是依靠手工编写代码, 过早的接触设计工具并不能让我们对 Qt 的概念突飞猛进.....

前面说过, 本教程很大程度上依照的是《C++ GUI Programming with Qt4, 2nd Edition》这本书。但是, 这本书中接下来的部分用了很大的篇幅完成了一个简单的类似 Excel 的程序。虽然最终效果看起来很不错, 但我并不打算完全依照这个程序来, 因为这个程序太大, 以至于我们在开始之后会有很大的篇幅接触不到能够运行的东西, 这无疑会严重打击学习的积极性——至少我是如此, 看不到做的东西很难受——所以, 我打算重新组织一下这个程序, 请大家按照我的思路试试看吧!

闲话少说, 下面开始新的篇章!

就像 `Swing` 的顶层窗口一般都是 `JFrame` 一样, Qt 的 GUI 程序也有一个常用的顶层窗口, 叫做 `MainWindow`。好了, 现在我们新建一个 `Gui Application` 项目 `MyApp`, 注意在后面选择的时候选择 `Base Class` 是 `QMainWindow`。



然后确定即可。此时，QtCreator 已经为我们生成了必要的代码，我们只需点击一下 **Run**，看看运行出来的结果。



一个很简单的窗口，什么都没有，这就是我们的主窗口了。

MainWindow 继承自 **QMainWindow**。**QMainWindow** 窗口分成几个主要的区域：



最上面是 **Window Title**，用于显示标题和控制按钮，比如最大化、最小化和关闭等；下面一些是 **Menu Bar**，用于显示菜单；再下面一点是 **Toolbar areas**，用于显示工具条，注意，Qt 的主窗口支持多个工具条显示，因此这里是 **ares**，你可以把几个工具条并排显示在这里，就像 Word2003 一样；工具条下面是 **Dock window areas**，这是停靠窗口的显示区域，所谓停靠窗口就是像 Photoshop 的工具箱一样，可以在主窗口的四周显示；再向下是 **Status Bar**，就是状态栏；中间最大的 **Central widget** 就是主要的工作区了。

好了，今天的内容不多，我们以后的工作就是要对这个 **MainWindow** 进行修改，以满足我们的各种需要。

Qt 学习之路(12): 菜单和工具条

在前面的 `QMainWindow` 的基础之上, 我们开始着手建造我们的应用程序。虽然现在已经有一个框架, 但是, 确切地说我们还一行代码没有写呢! 下面的工作就不那么简单了! 在这一节里面, 我们要为我们的框架添加菜单和工具条。

就像 `Swing` 里面的 `Action` 一样, `Qt` 里面也有一个类似的类, 叫做 `QAction`。顾名思义, `QAction` 类保存有关于这个动作, 也就是 `action` 的信息, 比如它的文本描述、图标、快捷键、回调函数(也就是信号槽), 等等。神奇的是, `QAction` 能够根据添加的位置来改变自己的样子——如果添加到菜单中, 就会显示成一个菜单项; 如果添加到工具条, 就会显示成一个按钮。这也是为什么要把菜单和按钮放在一节里面。下面开始学习!

首先, 我想添加一个打开命令。那么, 就在头文件里面添加一个私有的 `QAction` 变量:

```
class QAction;  
//...  
private:  
    QAction *openAction;  
//...
```

注意, 不要忘记 `QAction` 类的前向声明哦! 要不就会报错的!

然后我们要在 `cpp` 文件中添加 `QAction` 的定义。为了简单起见, 我们直接把它定义在构造函数里面:

```
openAction = new QAction(tr("&Open"), this);  
openAction->setShortcut(QKeySequence::Open);  
openAction->setStatusTip(tr("Open a file."));
```

第一行代码创建一个 `QAction` 对象。`QAction` 有几个重载的构造函数, 我们使用的是

```
QAction(const QString &text, QObject* parent);
```

这一个。它有两个参数, 第一个 `text` 是这个动作的文本描述, 用来显示文本信息, 比如在菜单中的文本; 第二个是 `parent`, 一般而言, 我们通常传入 `this` 指针就可以了。我们不需要去关心这个 `parent` 参数具体是什么, 它的作用是指明这个 `QAction` 的父组件, 当这个父组件被销毁时, 比如 `delete` 或者由系统自动销毁, 与其相关联的这个 `QAction` 也会自动被销毁。

如果你还是不明白构造函数的参数是什么意思, 或者说想要更加详细的了解 `QAction` 这个类, 那么就需要自己翻阅一下它的 `API` 文档。前面说过有关 `API` 的使用方法, 这里不再赘述。这也

是学习 Qt 的一种方法，因为 Qt 是一个很大的库，我们不可能面面俱到，因此只为说道用到的东西，至于你自己想要实现的功能，就需要自己去查文档了。

第二句，我们使用了 `setShortcut` 函数。`shortcut` 是这个动作的快捷键。Qt 的 `QKeySequence` 已经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，或者是在 `QtCreator` 中输入 `::` 后会有系统的自动补全功能显示出来。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，所以，推荐使用 `QKeySequence` 类来添加快捷键，这样，它会根据平台的不同来定义不同的快捷键。

第三句是 `setStatusTip` 函数。这是添加状态栏的提示语句。状态栏就是主窗口最下面的一条。现在我们的程序还没有添加状态栏，因此你是看不到有什么作用的。

下面要做的是把这个 `QAction` 添加到菜单和工具条：

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

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

`QMainWindow` 有一个 `menuBar()` 函数，会返回菜单栏，也就是最上面的那一条。如果不存在会自动创建，如果已经存在就返回那个菜单栏的指针。直接使用返回值添加一个菜单，也就是 `addMenu`，参数是一个 `QString`，也就是显示的菜单名字。然后使用这个 `QMenu` 指针添加这个 `QAction`。类似的，使用 `addToolBar` 函数的返回值添加了一个工具条，并且把这个 `QAction` 添加到了上面。

好了，主要的代码已经写完了。不过，如果你只修改这些话，是编译不过的哦！因为像 `menuBar()` 函数返回一个 `QMenuBar` 指针，但是你并没有 `include` 它的头文件哦！虽然没有明着写出 `QMenuBar` 这个类，但是实际上你已经用到了它的 `addMenu` 函数了，所以还是要注意的！

下面给出来全部的代码：

1. mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui/QMainWindow>
```

```

class QAction;

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private:
    QAction *openAction;
};

#endif // MAINWINDOW_H

```

2. mainwindow.cpp

```

#include <QtGui/QAction>
#include <QtGui/QMenu>
#include <QtGui/QMenuBar>
#include <QtGui/QKeySequence>
#include <QtGui/QToolBar>
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    openAction = new QAction(tr("&Open"), this);
    openAction->setShortcut(QKeySequence::Open);
    openAction->setStatusTip(tr("Open a file.));

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

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

MainWindow::~MainWindow()
{
}

```

main.cpp 没有修改，这里就不给出了。下面是运行结果：



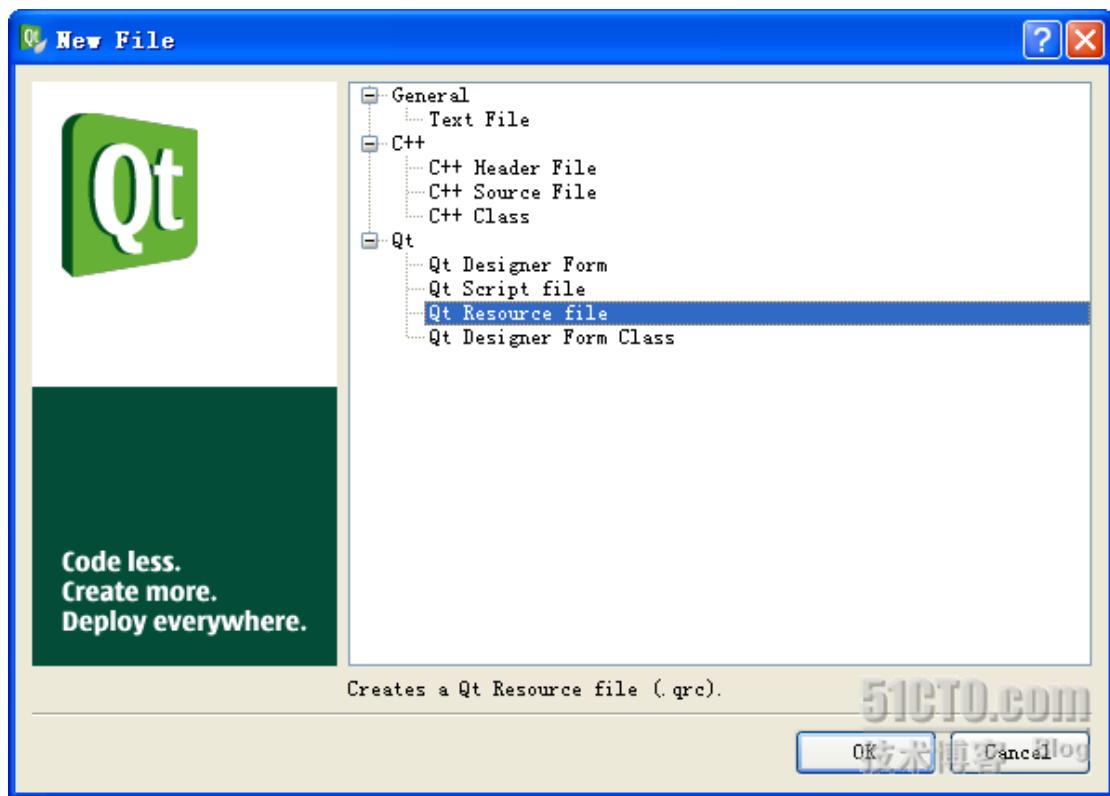
很丑，是吧？不过我们已经添加上了菜单和工具条了哦！按一下键盘上的 **Alt+F**，因为这是我们给它定义的快捷键。虽然目前挺难看，不过以后就会变得漂亮的！想想看，Linux 的 KDE 桌面可是 Qt 实现的呢！

Qt 学习之路(13)：菜单和工具条(续)

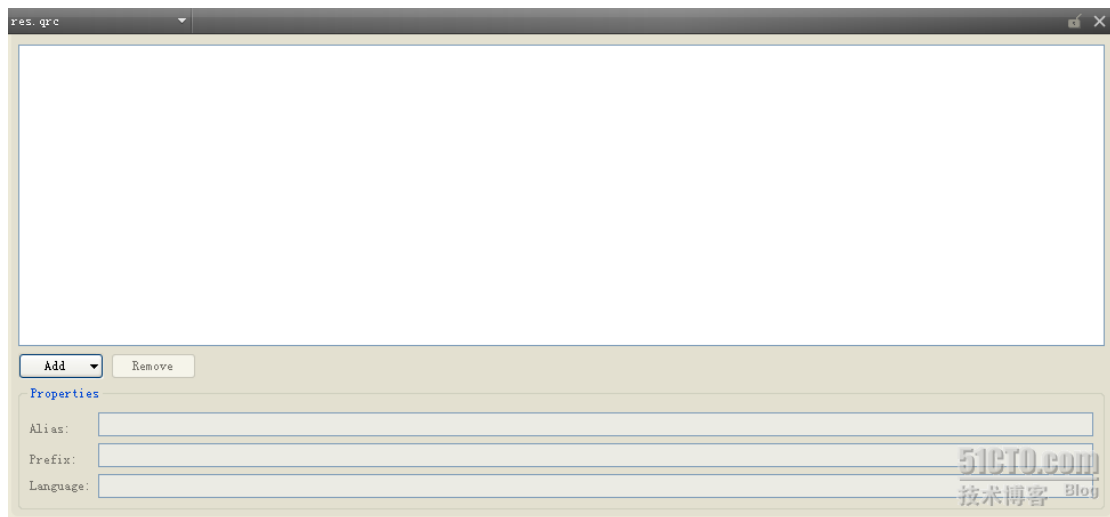
前面一节我们已经把 QAction 添加到菜单和工具条上面。现在我们要添加一些图片美化一下，然后把信号槽加上，这样，我们的 action 就可以相应啦！

首先来添加图标。QAction 的图标会显示在菜单项的前面以及工具条按钮上面显示。

为了添加图标，我们首先要使用 Qt 的资源文件。在 QtCreator 的项目上右击，选择 New File...，然后选择 resource file。

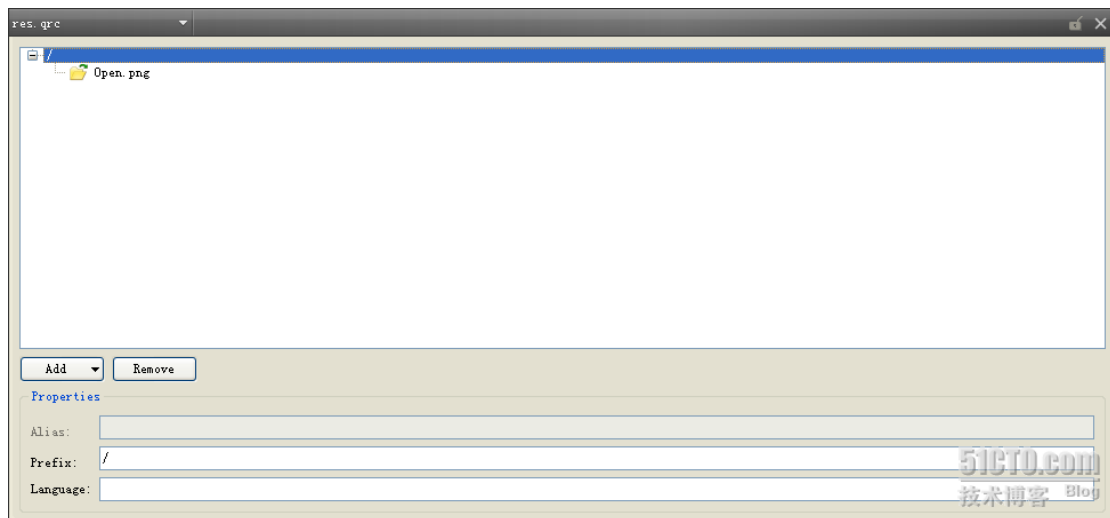


然后点击 **next**，选择好位置，**Finish** 即可。为了使用方便，我就把这个文件建在根目录下，建议应该在仔细规划好文件之后，建在专门的 **resources** 文件夹下。完成之后，生成的是一个 **.qrc** 文件，**qrc** 其实是 **Qt Resource Collection** 的缩写。它只是一个普通的 XML 文件，可以用记事本等打开。不过，这里我们不去深究它的结构，完全利用 **QtCreator** 操作这个文件，



点击 **Add** 按钮，首先选择 **Add prefix**，然后把生成的 **/new/prefix** 改成 **/**。这是 **prefix** 就是以后使用图标时需要提供的前缀，以 **/** 开头。添加过 **prefix** 之后，然后在工程文件中添加一个图标，再选择 **Add file**，选择那个图标。这样完成之后保存 **qrc** 文件即可。

说明一下，QToolBar 的图标大小默认是 32*32，菜单默认是 16*16。如果提供的图标小于要求的尺寸，则不做操作，Qt 不会为你放大图片；反之，如果提供的图标文件大于相应的尺寸要求，比如是 64*64，Qt 会自动缩小尺寸。



图片的路径怎么看呢？可以看出，Qt 的资源文件视图使用树状结构，根是/，叶子节点就是图片位置，连接在一起就是路径。比如这张图片的路径就是/Open.png。

注意，为了简单起见，我们没有把图标放在专门的文件夹中。正式的项目中应该单独有一个 **resources** 文件夹放资源文件的。

然后回到前面的 `mainwindow.cpp`，在构造函数中修改代码：

```
openAction = new QAction(tr("&Open"), this);
openAction->setShortcut(QKeySequence::Open);
openAction->setStatusTip(tr("Open a file.));
openAction->setIcon(QIcon(":/Open.png")); // Add code.
```

我们使用 `setIcon` 添加图标。添加的类是 `QIcon`，构造函数需要一个参数，是一个字符串。由于我们要使用 `qrc` 中定义的图片，所以字符串以 `:` 开始，后面跟着 `prefix`，因为我们先前定义的 `prefix` 是 `/`，所以就需要一个 `/`，然后后面是 `file` 的路径。这是在前面的 `qrc` 中定义的，打开 `qrc` 看看那张图片的路径即可。

好了，图片添加完成，然后点击运行，看看效果吧！



瞧！我们只需要修改 `QAction`，菜单和工具条就已经为我们做好了相应的处理，还是很方便的！

下一步，为 `QAction` 添加事件响应。还记得 Qt 的事件响应机制是基于信号槽吗？点击 `QAction` 会发出 `triggered()` 信号，所以，我们要做的是声明一个 `slot`，然后 `connect` 这个信号。

mainwindow.h

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private slots:
    void open();

private:
    QAction *openAction;
};
```

因为我们的 `open()` 目前只要在类的内部使用，因此定义成 `private slots` 即可。然后修改 `cpp` 文件：

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    openAction = new QAction(tr("&Open"), this);
    openAction->setShortcut(QKeySequence::Open);
    openAction->setStatusTip(tr("Open a file."));
    openAction->setIcon(QIcon(":/Open.png"));
    connect(openAction, SIGNAL(triggered()), this, SLOT(open()));

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



```

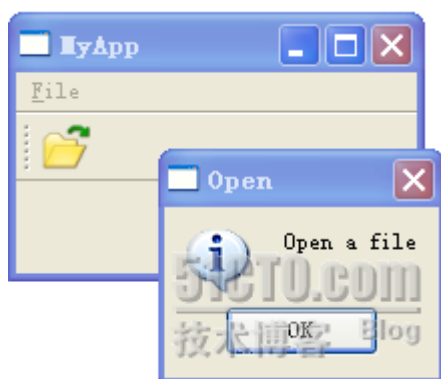
        file->addAction(openAction);

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

    void MainWindow::open()
    {
        QMessageBox::information(NULL, tr("Open"), tr("Open a file"));
    }

```

注意，我们在 `open()` 函数中简单的弹出一个标准对话框，并没有其他的操作。编译后运行，看看效果：



好了，关于 `QAction` 的动作也已经添加完毕了！

至此，`QAction` 有关的问题先告一段落。最后说一下，如果你还不知道怎么添加子菜单的话，看一下 `QMenu` 的 API，里面会有一个 `addMenu` 函数。也就是说，创建一个 `QMenu` 然后添加就可以的啦！

Qt 学习之路(14): 状态栏

有段时间没有写过博客了。假期去上海旅游，所以一直没有能够上网。现在又来到这里，开始新的篇章吧！

今天的内容主要还是继续完善前面的那个程序。我们要为我们的程序加上一个状态栏。

状态栏位于主窗口的最下方，提供一个显示工具提示等信息的地方。一般地，当窗口不是最大化的时候，状态栏的右下角会有一个可以调节大小的控制点；当窗口最大化的时候，这个控制点会自动消失。`Qt` 提供了一个 `QStatusBar` 类来实现状态栏。

Qt 具有一个相当成熟的 GUI 框架的实现——这一点感觉比 Swing 要强一些——Qt 似乎对 GUI 的开发做了很多设计，比如 QMainWindow 类里面就有一个 statusBar()函数，用于实现状态栏的调用。类似 menuBar()函数，如果不存在状态栏，该函数会自动创建一个，如果已经创建则会返回这个状态栏的指针。如果你要替换掉已经存在的状态栏，需要使用 QMainWindow 的 setStatusBar()函数。

在 Qt 里面，状态栏显示的信息有三种类型：临时信息、一般信息和永久信息。其中，临时信息指临时显示的信息，比如 QAction 的提示等，也可以设置自己的临时信息，比如程序启动之后显示 Ready，一段时间后自动消失——这个功能可以使用 QStatusBar 的 showMessage()函数来实现；一般信息可以用来显示页码之类的；永久信息是不会消失的信息，比如可以在状态栏提示用户 Caps Lock 键被按下之类。

QStatusBar 继承自 QWidget，因此它可以添加其他的 QWidget。下面我们在 QStatusBar 上添加一个 QLabel。

首先在 class 的声明中添加一个私有的 QLabel 属性：

```
private:
    QAction *openAction;
    QLabel *msgLabel;
```

然后在其构造函数中添加：

```
msgLabel = new QLabel;
msgLabel->setMinimumSize(msgLabel->sizeHint());
msgLabel->setAlignment(Qt::AlignHCenter);

statusBar()->addWidget(msgLabel);
```

这里，第一行创建一个 QLabel 的对象，然后设置最小大小为其本身的建议大小——注意，这样设置之后，这个最小大小可能是变化的——最后设置显示规则是水平居中(HCenter)。最后一行使用 statusBar()函数将这个 label 添加到状态栏。编译运行，将鼠标移动到工具条或者菜单的 QAction 上，状态栏就会有相应的提示：



看起来是不是很方便？只是，我们很快发现一个问题：当没有任何提示时，状态栏会有一个短短的竖线：



这是什么呢？其实，这是 `QLabel` 的边框。当没有内容显示时，`QLabel` 只显示出自己的一个边框。但是，很多情况下我们并不希望有这条竖线，于是，我们对 `statusBar()` 进行如下设置：

```
statusBar()->setStyleSheet(QString("QStatusBar::item{border: 0px}"));
```

这里先不去深究这句代码是什么意思，简单来说，就是把 `QStatusBar` 的子组件的 `border` 设置为 0，也就是没有边框。现在再编译试试吧！那个短线消失了！

`QStatusBar` 右下角的大小控制点可以通过 `setSizeGripEnabled()` 函数来设置是否存在，详情参见 API 文档。

好了，现在，我们的状态栏已经初步完成了。由于 `QStatusBar` 可以添加多个 `QWidget`，因此，我们可以构建出很复杂的状态栏。

Qt 学习之路(15): Qt 标准对话框之 `QFileDialog`

《Qt 学习之路》已经写到了第 15 篇，然而现在再写下去却有点困难，原因是当初并没有想到会连续的写下去，因此并没有很好的计划这些内容究竟该怎样去写。虽然前面说过，本教程主要线路参考《C++ Gui Programming with Qt 4, 2nd Edition》，然而最近的章节由于原文是一个比较完整的项目而有所改变，因此现在不知道该从何写起。

我本打算介绍很多组件的使用，因为 Qt 有很多组件，各种组件用法众多，根本不可能介绍完，只能把 API 放在手边，边用边查。所以，对于很多组件我只是简单的介绍一下，具体用法还请自行查找(确切地说，我知道的也并不多，很多时候还是要到 API 里面去找)。

下面还是按照我们的进度，从 Qt 的标准对话框开始说起。所谓标准对话框，其实就是 Qt 内置的一些对话框，比如文件选择、颜色选择等等。今天首先介绍一下 `QFileDialog`。

`QFileDialog` 是 `Qt` 中用于文件打开和保存的对话框，相当于 `Swing` 里面的 `JFileChooser`。下面我们打开我们前面使用的工程。我们已经很有先见之明的写好了一个打开的 `action`，还记得前面的代码吗？当时，我们只是弹出了一个消息对话框(这也是一种标准对话框哦~)用于告知这个信号槽已经联通，现在我们要写真正的打开代码了！

修改 `MainWindow` 的 `open` 函数：

```
void MainWindow::open()
{
    QString path = QFileDialog::getOpenFileName(this, tr("Open Image"), ".",
tr("Image Files (*.jpg *.png)"));
    if(path.length() == 0) {
        QMessageBox::information(NULL, tr("Path"), tr("You didn't select a
ny files.));
    } else {
        QMessageBox::information(NULL, tr("Path"), tr("You selected ") +
path);
    }
}
```

编译之前别忘记 `include QFileDialog` 哦！然后运行一下吧！点击打开按钮，就会弹出打开对话框，然后选择文件或者直接点击取消，会有相应的消息提示。

`QFileDialog` 提供了很多静态函数，用于获取用户选择的文件。这里我们使用的是 `getOpenFileName()`，也就是“获取打开文件名”，你也可以查看 `API` 找到更多的函数使用。不过，这个函数的参数蛮长的，而且类型都是 `QString`，并不好记。考虑到这种情况，`Qt` 提供了另外的写法：

```
QFileDialog *fileDialog = new QFileDialog(this);
fileDialog->setWindowTitle(tr("Open Image"));
fileDialog->setDirectory(".");
fileDialog->setFilter(tr("Image Files (*.jpg *.png)"));
if(fileDialog->exec() == QDialog::Accepted) {
    QString path = fileDialog->selectedFiles()[0];
    QMessageBox::information(NULL, tr("Path"), tr("You selected ") +
path);
} else {
    QMessageBox::information(NULL, tr("Path"), tr("You didn't select a
ny files.));
}
```

不过，这两种写法虽然功能差别不大，但是弹出的对话框却并不一样。`getOpenFileName()` 函数在 **Windows** 和 **MacOS X** 平台上提供的是本地的对话框，而 `QFileDialog` 提供的始终是 **Qt** 自己绘制的对话框(还记得前面说过，**Qt** 的组件和 **Swing** 类似，也是自己绘制的，而不都是调用系统资源 **API**)。

为了说明 `QFileDialog::getOpenFileName()` 函数的用法，还是先把函数签名放在这里：

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

第一个参数 `parent`，用于指定父组件。注意，很多 **Qt** 组件的构造函数都会有这么一个 `parent` 参数，并提供一个默认值 `0`；

第二个参数 `caption`，是对话框的标题；

第三个参数 `dir`，是对话框显示时默认打开的目录，`"."` 代表程序运行目录，`"/"` 代表当前盘符的根目录(**Windows**, **Linux** 下/就是根目录了)，也可以是平台相关的，比如 `"C:\\\\"` 等；

第四个参数 `filter`，是对话框的后缀名过滤器，比如我们使用 `"Image Files(*.jpg *.png)"` 就让它只能显示后缀名是 `jpg` 或者 `png` 的文件。如果需要使用多个过滤器，使用 `;;` 分割，比如 `"JPEG Files(*.jpg);;PNG Files(*.png)"`；

第五个参数 `selectedFilter`，是默认选择的过滤器；

第六个参数 `options`，是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。

如果我要想选择多个文件怎么办呢？**Qt** 提供了 `getOpenFileNames()` 函数，其返回值是一个 `QStringList`。你可以把它理解成一个只能存放 `QString` 的 `List`，也就是 **STL** 中的 `list<string>`。

好了，我们已经能够选择打开文件了。保存也是类似的，`QFileDialog` 类也提供了保存对话框的函数 `getSaveFileName`，具体使用还是请查阅 **API**。

Qt 学习之路(16): Qt 标准对话框之 `QColorDialog`

继续来说 **Qt** 的标准对话框，这次说说 `QColorDialog`。这是 **Qt** 提供的颜色选择对话框。

使用 `QColorDialog` 也很简单，Qt 提供了 `getColor()` 函数，类似于 `QFileDialog` 的 `getOpenFileName()`，可以直接获得选择的颜色。我们还是使用前面的 `QAction` 来测试下这个函数：

```
| QColor color = QColorDialog::getColor(Qt::white, this);  
| QString msg = QString("r: %1, g: %2, b: %3").arg(QString::number(c  
| color.red()), QString::number(color.green()), QString::number(color.blue()));  
| QMessageBox::information(NULL, "Selected color", msg);
```

不要忘记 `include QColorDialog` 哦！这段代码虽然很少，但是内容并不少。

第一行 `QColorDialog::getColor()` 调用了 `QColorDialog` 的 `static` 函数 `getColor()`。这个函数有两个参数，第一个是 `QColor` 类型，是对话框打开时默认选择的颜色，第二个是它的 `parent`。

第二行比较长，涉及到 `QString` 的用法。如果我没记错的话，这些用法还没有提到过，本着“有用就说”的原则，尽管这些和 `QColorDialog` 毫不相干，这里还是解释一下。`QString("r: %1, g: %2, b: %3")` 创建了一个 `QString` 对象。我们使用了参数化字符串，也就是那些 `%1` 之类。在 Java 的 `properties` 文件中，字符参数是用 `{0}`、`{1}` 之类实现的。其实这都是一些占位符，也就是，后面会用别的字符串替换掉这些值。占位符的替换需要使用 `QString` 的 `arg()` 函数。这个函数会返回它的调用者，因此可以使用链式调用写法。它会按照顺序替换掉占位符。然后是 `QString::number()` 函数，这也是 `QString` 的一个 `static` 函数，作用就是把 `int`、`double` 等值换成 `QString` 类型。这里是把 `QColor` 的 `R`、`G`、`B` 三个值输出了出来。关于 `QString` 类，我们会在以后详细说明。

第三行就比较简单了，使用一个消息对话框把刚刚拼接的字符串输出。

现在就可以运行这个测试程序了。看上去很简单，不是吗？

`QColorDialog` 还有一些其他的函数可以使用。

`QColorDialog::setCustomColor()` 可以设置用户自定义颜色。这个函数有两个值，第一个是自定义颜色的索引，第二个是自定义颜色的 `RGB` 值，类型是 `QRgb`，大家可以查阅 `API` 文档来看看这个类的使用，下面只给出一个简单的用法：

```
| QColorDialog::setCustomColor(0, QRgb(0x0000FF));
```

`getColor()` 还有一个重载的函数，签名如下：

```
| QColorDialog::getColor( const QColor & initial, QWidget * parent, const QStrin  
| g & title, ColorDialogOptions options = 0 )
```

第一个参数 **initial** 和前面一样，是对话框打开时的默认选中的颜色；

第二个参数 **parent**，设置对话框的父组件；

第三个参数 **title**，设置对话框的 **title**；

第四个参数 **options**，是 `QColorDialog::ColorDialogOptions` 类型的，可以设置对话框的一些属性，如是否显示 **Alpha** 值等，具体属性请查阅 **API** 文档。特别的，这些值是可以使用 **OR** 操作的。

`QColorDialog` 相对简单一些，**API** 文档也很详细，大家遇到问题可以查阅文档的哦！

Qt 学习之路(17): Qt 标准对话框之 `QMessageBox`

好久没有更新博客，主要是公司里面还在验收一些东西，所以没有及时更新。而且也在写一个基于 `Qt` 的画图程序，基本上类似于 `PS` 的东西，主要用到的是 `Qt Graphics View Framework`。好了，现在还是继续来说说 `Qt` 的标准对话框吧！

这次来说一下 `QMessageBox` 以及类似的几种对话框。其实，我们已经用过 `QMessageBox` 了，就在之前的几个程序中。不过，当时是大略的说了一下，现在专门来说说这几种对话框。

先来看一下最熟悉的 `QMessageBox::information`。我们在以前的代码中这样使用过：

```
QMessageBox::information(NULL, "Title", "Content", QMessageBox::Yes |  
QMessageBox::No, QMessageBox::Yes);
```

下面是一个简单的例子：



现在我们从 **API** 中看看它的函数签名：

```
static StandardButton QMessageBox::information ( QWidget *  
parent, const QString & title, const QString & text, StandardButtons buttons = Ok,  
StandardButton defaultButton = NoButton );
```

首先，它是 **static** 的，所以我们能够使用类名直接访问到(怎么看都像废话...)；然后看它那一堆参数，第一个参数 **parent**，说明它的父组件；第二个参数 **title**，也就是对话框的标题；第三个参数 **text**，是对话框显示的内容；第四个参数 **buttons**，声明对话框放置的按钮，默认是只放置一个 **OK** 按钮，这个参数可以使用或运算，例如我们希望有一个 **Yes** 和一个 **No** 的按钮，可以使用 `QMessageBox::Yes | QMessageBox::No`，所有的按钮类型可以在 `QMessageBox` 声明的 `StandardButton` 枚举中找到；第五个参数 `defaultButton` 就是默认选中的按钮，默认值是 `NoButton`，也就是哪个按钮都不选中。这么多参数，豆子也是记不住的啊！所以，我们在用 `QtCreator` 写的时候，可以在输入 `QMessageBox::information` 之后输入(，稍等一下，`QtCreator` 就会帮我们吧函数签名显示在右上方了，还是挺方便的一个功能！

`Qt` 提供了五个类似的接口，用于显示类似的窗口。具体代码这里就不做介绍，只是来看一下样子吧！

```
QMessageBox::critical(NULL, "critical", "Content", QMessageBox::Yes |  
QMessageBox::No, QMessageBox::Yes);
```



```
QMessageBox::warning(NULL, "warning", "Content", QMessageBox::Yes |  
QMessageBox::No, QMessageBox::Yes);
```



```
QMessageBox::question(NULL, "question", "Content", QMessageBox::Yes |  
QMessageBox::No, QMessageBox::Yes);
```




```
QMessageBox::about(NULL, "About", "About this application");
```

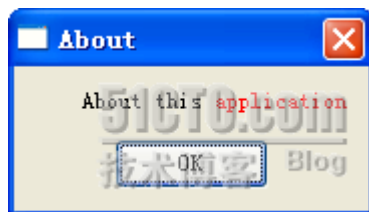


请注意，最后一个 `about()` 函数是没有后两个关于 `button` 设置的按钮的！

`QMessageBox` 对话框的文本信息时可以支持 HTML 标签的。例如：

```
QMessageBox::about(NULL, "About", "About this <font  
color='red'>application</font>");
```

运行效果如下：



如果我们想自定义图片的话，也是很简单的。这时候就不能使用这几个 `static` 的函数了，而是要我们自己定义一个 `QMessageBox` 来使用：

```
QMessageBox message(QMessageBox::NoIcon, "Title", "Content with icon.");  
message.setIconPixmap(QPixmap("icon.png"));  
message.exec();
```

这里我们使用的是 `exec()` 函数，而不是 `show()`，因为这是一个模态对话框，需要有它自己的事件循环，否则的话，我们的对话框会一闪而过哦(感谢 `laetitia` 提醒)。

需要注意的是，同其他的程序类似，我们在程序中定义的相对路径都是要相对于运行时的 `.exe` 文件的地址的。比如我们写 `"icon.png"`，意思是在 `.exe` 的当前目录下寻找一个 `"icon.png"` 的文件。这个程序的运行效果如下：



还有一点要注意，我们使用的是 **png** 格式的图片。因为 **Qt** 内置的处理图片格式是 **png**，所以这不会引起很大的麻烦，如果你要使用 **jpeg** 格式的图片的话，**Qt** 是以插件的形式支持的。在开发时没有什么问题，不过如果要部署的话，需要注意这一点。

最后再来说一下怎么处理对话框的交互。我们使用 **QMessageBox** 类的时候有两种方式，一是使用 **static** 函数，另外是使用构造函数。

首先来说一下 **static** 函数的方式。注意，**static** 函数都是要返回一个 **StandardButton**，我们就可以通过判断这个返回值来对用户的操作做出相应。

```
QMessageBox::StandardButton rb = QMessageBox::question(NULL, "Show Qt", "Do you want to show Qt dialog?", QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
if(rb == QMessageBox::Yes)
{
    QMessageBox::aboutQt(NULL, "About Qt");
}
```

如果要使用构造函数的方式，那么我们就自己运行判断一下啦：

```
QMessageBox message(QMessageBox::NoIcon, "Show Qt", "Do you want to show Qt dialog?", QMessageBox::Yes | QMessageBox::No, NULL);
if(message.exec() == QMessageBox::Yes)
{
    QMessageBox::aboutQt(NULL, "About Qt");
}
```

其实道理上也是差不多的。

Qt 学习之路(18): Qt 标准对话框之 **QInputDialog**

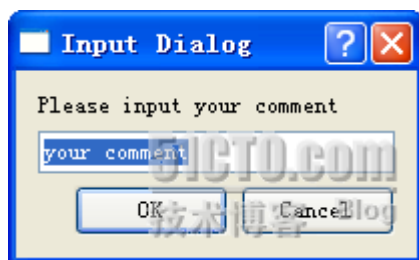
这是 Qt 标准对话框的最后一部分。正如同其名字显示的一样，QInputDialog 用于接收用户的输入。QInputDialog 提供了一些简单的 static 函数，用于快速的建立一个对话框，正像 QColorDialog 提供了 getColor 函数一样。

首先来看看 getText 函数：

```
bool isOK;
QString text = QInputDialog::getText(NULL, "Input Dialog",
                                     "Please input your comment",
                                     QLineEdit::Normal,
                                     "your comment",
                                     &isOK);

if(isOK) {
    QMessageBox::information(NULL, "Information",
                             "Your comment is: <b>" + text + "</b>",
                             QMessageBox::Yes | QMessageBox::No,
                             QMessageBox::Yes);
}
```

代码比较简单，使用 getText 函数就可以弹出一个可供用户输入的对话框：



下面来看一下这个函数的签名：

```
static QString QInputDialog::getText ( QWidget * parent,
                                       const QString & title,
                                       const QString & label,
                                       QLineEdit::EchoMode mode = QLineEdit::Normal,
                                       const QString & text = QString(),
                                       bool * ok = 0,
                                       Qt::WindowFlags flags = 0 )
```

第一个参数 parent，也就是那个熟悉的父组件的指针；第二个参数 title 就是对话框的标题；第三个参数 label 是在输入框上面的提示语句；第四个参数 mode 用于指明这个 QLineEdit 的输入模式，取值范围是 QLineEdit::EchoMode，默认是 Normal，也就是正常显示，你也可以声明为 password，这样就是密码的输入显示了，具体请查阅 API；第五个参数 text 是 QLineEdit 的默认字符串；第六个参数 ok 是可选的，如果非 NULL，则当用户按下对话框的 OK 按钮时，

这个 `bool` 变量会被置为 `true`，可以由这个去判断用户是按下的 `OK` 还是 `Cancel`，从而获知这个 `text` 是不是有意义；第七个参数 `flags` 用于指定对话框的样式。

虽然参数很多，但是每个参数的含义都比较明显，大家只要参照 `API` 就可以知道了。

函数的返回值是 `QString`，也就是用户在 `QLineEdit` 里面输入的内容。至于这个内容有没有意义，那就要看那个 `ok` 参数是不是 `true` 了。

`QInputDialog` 不仅提供了获取字符串的函数，还有 `getInteger`，`getDouble`，`getItem` 三个类似的函数，这里就不一一介绍。

Qt 学习之路(19): 事件(event)

面说了几个标准对话框，下面不打算继续说明一些组件的使用，因为这些使用很难讲完，很多东西都是与实际应用相关的。实际应用的复杂性决定了我们根本不可能把所有组件的所有使用方法都说明白。这次来说说 `Qt` 相对高级一点的特性：事件。

事件(event)是有系统或者 `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。一些事件是在对用户操作做出响应的时候发出，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。

一般来说，使用 `Qt` 编程时，我们并不会把主要精力放在事件上，因为在 `Qt` 中，需要我们关心的事件总会发出一个信号。比如，我们关心的是 `QPushButton` 的鼠标点击，但我们不需要关心这个鼠标点击事件，而是关心它的 `clicked()` 信号。这与其他的一些框架不同：在 `Swing` 中，你所要关心的是 `JButton` 的 `ActionListener` 这个点击事件。

`Qt` 的事件很容易和信号槽混淆。这里简单的说明一下，`signal` 由具体对象发出，然后会马上交给由 `connect` 函数连接的 `slot` 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` 的事件也是可以进入事件队列，而是直接处理的。并且，事件还可以使用“事件过滤器”进行过滤。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 `QPushButton`，那么我们就需要重写它的鼠标点击事件和键盘处理事件，并且在恰当的时候发出 `clicked()` 信号。

还记得我们在 `main` 函数里面创建了一个 `QApplication` 对象，然后调用了它的 `exec()` 函数吗？其实，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` 的所有事件都继承于 `QEvent` 类。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，我们应该在子类中重定义这些函数。下面来看一个例子。

Qt 学习之路(20): 事件接收与忽略

本章内容也是关于 Qt 事件。或许这一章不能有一个完整的例子，因为对于事件总是感觉很抽象，还是从底层上理解一下比较好的吧！

前面说到了事件的作用，下面来看看我们如何来接收事件。回忆一下前面的代码，我们在子类中重写了事件函数，以便让这些子类按照我们的需要完成某些功能，就像下面的代码：

```
void MyLabel::mousePressEvent(QMouseEvent * event)

{

    if(event->button() == Qt::LeftButton) {

        // do something

    } else {

        QLabel::mousePressEvent(event);

    }

}
```

上面的代码和前面类似，在鼠标按下的事件中检测，如果按下的是左键，做我们的处理工作，如果不是左键，则调用父类的函数。这在某种程度上说，是把事件向上传递给父类去响应，也就是说，我们在子类中“忽略”了这个事件。

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，就会继续向其他类传递。其实，Qt 的事件对象都有一个 `accept()` 函数和 `ignore()` 函数。正如它们的名字，前者用来告诉 Qt，事件处理函数“接收”了这个事件，不要再传递；后者则告诉 Qt，事件处理函数“忽略”了这个事件，需要继续传递，寻找另外的接受者。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。

事实上，我们很少使用 `accept()` 和 `ignore()` 函数，而是想上面的示例一样，如果希望忽略事件，只要调用父类的响应函数即可。记得我们曾经说过，Qt 中的事件大部分是 `protected` 的，因此，重写的函数必定存在着其父类中的响应函数，这个方法是可行的。为什么要这么做呢？因为我们无法确认父类中的这个处理函数没有操作，如果我们在子类中直接忽略事件，Qt 不会再去寻找其他的接受者，那么父类的操作也就不能进行，这可能会有潜在的危险。另外我们查看一下 `QWidget` 的 `mousePressEvent()` 函数的实现：

```
void QWidget::mousePressEvent(QMouseEvent *event)

{

    event->ignore();

    if ((windowType() == Qt::Popup)) {
```

```

        event->accept();

        QWidget* w;

        while ((w = qApp->activePopupWidget()) && w != this){

            w->close();

            if (qApp->activePopupWidget() == w) // widget does not
want to dissappear

                w->hide(); // hide at least

        }

        if (!rect().contains(event->pos())){

            close();

        }

    }

}

```

请注意第一条语句，如果所有子类都没有覆盖 `mousePressEvent` 函数，这个事件会在这里被忽略掉，也就是停止传播。另外也可以看到，如果你在子类直接 `ignore` 了这个事件，`QWidget` 事件处理函数就不会被调用，那么，后面的 `Popup` 操作或许就这么“莫名其妙”地消失了。

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，那就是在窗口关闭的时候。如果你在窗口关闭时需要有个询问对话框，那么就需要这么去写：

```

void MainWindow::closeEvent(QCloseEvent * event)

{

    if(continueToClose()) {

        event->accept();

    } else {

        event->ignore();
    }
}

```

```

    }

}

bool MainWindow::continueToClose()
{
    if(QMessageBox::question(this,

                               tr("Quit"),

                               tr("Are you sure to quit this application?"),

                               QMessageBox::Yes | QMessageBox::No,

                               QMessageBox::No)

        == QMessageBox::Yes) {

        return true;

    } else {

        return false;

    }

}

```

这样，我们经过询问之后才能正常退出程序。

```

#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QMouseEvent>

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 app(argc, argv);
    EventLabel *label = new EventLabel;
    label->setWindowTitle("MouseEvent Demo");
    label->resize(300, 200);
    label->show();
    return app.exec();
}

```

这里我们继承了 `QLabel` 类，重写了 `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`)时把坐标显示在这个 `Label` 上面。注意我们在 `mouseReleaseEvent` 函数里面有关 `QString` 的构造。我们没有使用 `arg` 参数的方式，而是使用 C 语言风格的 `sprintf` 来构造 `QString` 对象，如果你对 C 语法很熟悉(估计很多 C++ 程序员都会比较熟悉的吧)，那么就可以在 Qt 中试试熟悉的 C 格式化写法啦！

Qt 学习之路(21): event()

今天要说的是 `event()` 函数。记得之前曾经提到过这个函数，说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)。

`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，那么，就需要注意这个 `event()` 函数了。为了达到这种目的，我们可以重写 `event()` 函数。例如，如果你希望在窗口中的 `tab` 键按下时将焦点移动到下一组件，而不是让具有焦点的组件处理，那么你就可以继承 `QWidget`，并重写它的 `event()` 函数，已达到这个目的：

```
bool MyWidget::event(QEvent *event) {

    if (event->type() == QEvent::KeyPress) {

        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);

        if (keyEvent->key() == Qt::Key_Tab) {

            // 处理 Tab 键

            return true;

        }

    }

    return QWidget::event(event);

}
```

`event()` 函数接受一个 `QEvent` 对象，也就是需要这个函数进行转发的对象。为了进行转发，必定需要有一系列的类型判断，这就可以调用 `QEvent` 的 `type()` 函数，其返回值是 `QEvent::Type` 类型的枚举。我们处理过自己需要的事件后，可以直接 `return` 回去，对于其他我们不关心的事件，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。

`event()` 函数返回值是 `bool` 类型，如果传入的事件已被识别并且处理，返回 `true`，否则返回 `false`。如果返回值是 `true`，`QApplication` 会认为这个事件已经处理完毕，会继续处理事件队列中的下一事件；如果返回值是 `false`，`QApplication` 会尝试寻找这个事件的下一个处理函数。

`event()` 函数的返回值和事件的 `accept()` 和 `ignore()` 函数不同。`accept()` 和 `ignore()` 函数用于不同的事件处理器之间的沟通，例如判断这一事件是否处理；`event()` 函数的返回值主要是通知 `QApplication` 的 `notify()` 函数是否处理下一事件。为了更加明晰这一点，我们来看看 `QWidget` 的 `event()` 函数是如何定义的：

```

bool QWidget::event(QEvent *event) {

    switch (e->type()) {

        case QEvent::KeyPress:

            keyPressEvent((QKeyEvent *)event);

            if (!((QKeyEvent *)event)->isAccepted())

                return false;

            break;

        case QEvent::KeyRelease:

            keyReleaseEvent((QKeyEvent *)event);

            if (!((QKeyEvent *)event)->isAccepted())

                return false;

            break;

        // more...

    }

    return true;

}

```

QWidget 的 event() 函数使用一个巨大的 switch 来判断 QEvent 的 type，并且分发给不同的事件处理函数。在事件处理函数之后，使用这个事件的 isAccepted() 方法，获知这个事件是不是被接受，如果没有被接受则 event() 函数立即返回 false，否则返回 true。

另外一个必须重写 event() 函数的情形是有自定义事件的时候。如果你的程序中有自定义事件，则必须重写 event() 函数以便将自定义事件进行分发，否则你的自定义事件永远也不会被调用。关于自定义事件，我们会在以后的章节中介绍。

Qt 学习之路(22): 事件过滤器

Qt 创建了 QEvent 事件对象之后，会调用 QObject 的 event() 函数做事件的分发。有时候，你可能需要在调用 event() 函数之前做一些另外的操作，比如，对话框上某些组件可能并不需要响应回车按下的事件，此时，你就需要重新定义组件的 event() 函数。如果组件很多，就需要重写很多次 event() 函数，这显然没有效率。为此，你可以使用一个事件过滤器，来判断是否需要调用 event() 函数。

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 )
```

如果 `watched` 对象安装了事件过滤器，这个函数会被调用并进行事件过滤，然后才轮到组件进行事件处理。在重写这个函数时，如果你需要过滤掉某个事件，例如停止对这个事件的响应，需要返回 `true`。

```
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);

    }

}
```

上面的例子中为 `MainWindow` 建立了一个事件过滤器。为了过滤某个组件上的事件，首先需要判断这个对象是哪个组件，然后判断这个事件的类型。例如，我不想让 `textEdit` 组件处理键盘事件，于是就首先找到这个组件，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`。对于其他组件，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。

在创建了过滤器之后，下面要做的是安装这个过滤器。安装过滤器需要调用 `installEventFilter()` 函数。这个函数的签名如下：

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

这个函数是 `QObject` 的一个函数，因此可以安装到任何 `QObject` 的子类，并不仅仅是 `UI` 组件。这个函数接收一个 `QObject` 对象，调用了这个函数安装事件过滤器的组件会调用 `filterObj` 定义的 `eventFilter()` 函数。例如，`textField.installEventFilter(obj)`，则如果有事件发送到 `textField` 组件是，会先调用 `obj->eventFilter()` 函数，然后才会调用 `textField.event()`。

当然，你也可以把事件过滤器安装到 `QApplication` 上面，这样就可以过滤所有的事件，已获得更大的控制权。不过，这样做的后果就是会降低事件分发的效率。

如果一个组件安装了多个过滤器，则最后一个安装的会最先调用，类似于堆栈的行为。

注意，如果你在事件过滤器中 `delete` 了某个接收组件，务必将返回值设为 `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的组件必须在同一线程，否则，过滤器不起作用。另外，如果在 `install` 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。

事件的调用最终都会调用 `QCoreApplication` 的 `notify()` 函数，因此，最大的控制权实际上是重写 `QCoreApplication` 的 `notify()` 函数。由此可以看出，Qt 的事件处理实际上是分层五个层次：重定义事件处理函数，重定义 `event()` 函数，为单个组件安装事件过滤器，为 `QApplication` 安装事件过滤器，重定义 `QCoreApplication` 的 `notify()` 函数。这几个层次的控制权是逐层增大的。

Qt 学习之路(23)：自定义事件

这部分将作为 Qt 事件部分的结束。我们在前面已经从大概上了解了 Qt 的事件机制。下面要说的是如何自定义事件。

Qt 允许你创建自己的事件类型，这在多线程的程序中尤其有用，当然，也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，而不是使用信号槽呢？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 中的自定义事件很简单，同其他类似的库的使用很相似，都是要继承一个类进行扩展。在 Qt 中，你需要继承的类是 `QEvent`。注意，在 **Qt3** 中，你需要继承的类是 `QCustomEvent`，不过这个类在 **Qt4** 中已经被废除(这里的废除是不建议使用，并不是从类库中删除)。

继承 `QEvent` 类，你需要提供一个 `QEvent::Type` 类型的参数，作为自定义事件的类型值。这里的 `QEvent::Type` 类型是 `QEvent` 里面定义的一个 `enum`，因此，你是可以传递一个 `int` 的。重要的是，你的事件类型不能和已经存在的 `type` 值重复，否则会有不可预料的错误发生！因为系统会将你的事件当做系统事件进行派发和调用。在 Qt 中，系统将保留 0 - 999 的值，也就是说，你的事件 `type` 要大于 999。具体来说，你的自定义事件的 `type` 要在 `QEvent::User` 和 `QEvent::MaxUser` 的范围之间。其中，`QEvent::User` 值是 1000，`QEvent::MaxUser` 的值是 65535。从这里知道，你最多可以定义 64536 个事件，相信这个数字已经足够大了！但是，即便如此，也只能保证用户自定义事件不能

覆盖系统事件,并不能保证自定义事件之间不会被覆盖。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 );
```

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, 其默认值为 `-1`, 返回值是创建的这个 `Type` 类型的值。如果 `hint` 是合法的, 不会发生任何覆盖, 则会返回这个值; 如果 `hint` 不合法, 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, 因此不必另外添加同步。

你可以在 `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` 中提供了两种发送方式:

- `static bool QCoreApplication::sendEvent(QObject * receiver, QEvent * event):` 事件被 `QCoreApplication` 的 `notify()` 函数直接发送给 `receiver` 对象, 返回值是事件处理函数的返回值。使用这个函数**必须**要在栈上创建对象, 例如:

```
QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);

QApplication::sendEvent(mainWindow, &event);
```

- `static bool QCoreApplication::postEvent(QObject * receiver, QEvent * event):` 事件被 `QCoreApplication` 追加到事件列表的最后, 并等待处理, 该函数将事件追加后会立即返回, 并且注意, 该函数是线程安全的。另外一点是, 使用这个函数**必须**要在堆上创建对象, 例如:

```
QApplication::postEvent(object, new MyEvent(QEvent::registerEventType(2048))
);
```

这个对象不需要手动 `delete`, `Qt` 会自动 `delete` 掉! 因此, 如果在 `post` 事件之后调用 `delete`, 程序可能会崩溃。另外, `postEvent()` 函数还有一个重载的版本, 增加一个优先级参数, 具体请参见 `API`。通过调用 `sendPostedEvent()` 函数可以让已提交的事件立即得到处理。

如果要处理自定义事件, 可以重写 `QObject` 的 `customEvent()` 函数, 该函数接收一个 `QEvent` 对象作为参数。注意, 在 `Qt3` 中这个参数是 `QCustomEvent` 类型的。你可以像前面介绍的重写 `event()` 函数的方法去重写这个函数:

```
void CustomWidget::customEvent(QEvent *event) {

    CustomEvent *customEvent = static_cast<CustomEvent *>(event);

    // ....

}
```

另外, 你也可以通过重写 `event()` 函数来处理自定义事件:

```

bool CustomWidget::event(QEvent *event) {

    if (event->type() == MyCustomEventType) {

        CustomEvent *myEvent = static_cast<CustomEvent *>(event);

        // processing...

        return true;

    }

    return QWidget::event(event);

}

```

这两种办法都是可行的。

好了，至此，我们已经概略的介绍了 Qt 的事件机制，包括事件的派发、自定义等一系列的问题。下面的章节将继续我们的学习之路！

Qt 学习之路(24): QPainter

多些大家对我的支持啊！有朋友也提出，前面的几节有关 **event** 的教程缺少例子。因为 **event** 比较难做例子，也就没有去写，只是把大概写了一下。今天带来的是新的部分，有关 Qt 的 2D 绘图。这部分不像前面的内容，还是比较好理解的啦！所以，例子也会增加出来。

有人问豆子拿 Qt 做什么，其实，豆子就是在做一个 Qt 的画图程序，努力朝着 Photoshop 和 GIMP 的方向发展。但这终究要经过很长的时间、很困难的路程的，所以也放在网上开源，有兴趣的朋友可以来试试的呀...

好了，闲话少说，来继续我们的学习吧！

Qt 的绘图系统允许使用相同的 API 在屏幕和打印设备上绘制。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；QPainterDevice 是一个二维空间的抽象，这个二维空间可以由 QPainter 在上面进行绘制；QPaintEngine 提供了画笔 painter 在不同的设备上进行绘制的统一的接口。QPaintEngine 类用在 QPainter 和 QPainterDevice 之间，并且通常对开发人员是透明的，除非你需要自定义一个设备，这时候你就必须重新定义 QPaintEngine 了。

下图给出了这三个类之间的层次结构(出自 Qt API 文档)：



这种实现的主要好处是，所有的绘制都遵循着同一种绘制流程，这样，添加可以很方便的添加新的特性，也可以为不支持的功能添加一个默认的实现方式。另外需要说明一点，Qt 提供了一个独立的 QtOpenGL 模块，可以让你在 Qt 的应用程序中使用 OpenGL 功能。该模块提供了一个 OpenGL 的模块，可以像其他的 Qt 组件一样的使用。它的不同之处在于，它是使用 OpenGL 作为显示技术，使用 OpenGL 函数进行绘制。对于这个组件，我们以后会再介绍。

通过前面的介绍我们知道，Qt 的绘图系统实际上是说，使用 QPainter 在 QPaintDevice 上面进行绘制，它们之间使用 QPaintEngine 进行通讯。好了，下面我们来看看怎么使用 QPainter。

首先我们定义一个组件，同前面的定义类似：

```
class PaintedWidget : public QWidget
{
public:
    PaintedWidget();

protected:
    void paintEvent(QPaintEvent *event);
};
```

这里我们只定义了一个构造函数，并且重定义 `paintEvent()` 函数。从名字就可以看出，这实际上是一个事件的回调函数。请注意，一般而言，Qt 的事件函数都是 `protected` 的，所以，如果你要重写事件，就需要继承这个类了。至于事件相关的东西，我们在前面的内容已经比较详细的叙述了，这里不再赘述。

构造函数里面主要是一些大小之类的定义，这里不再详细说明：

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

我们关心的是 `paintEvent()` 函数的实现：

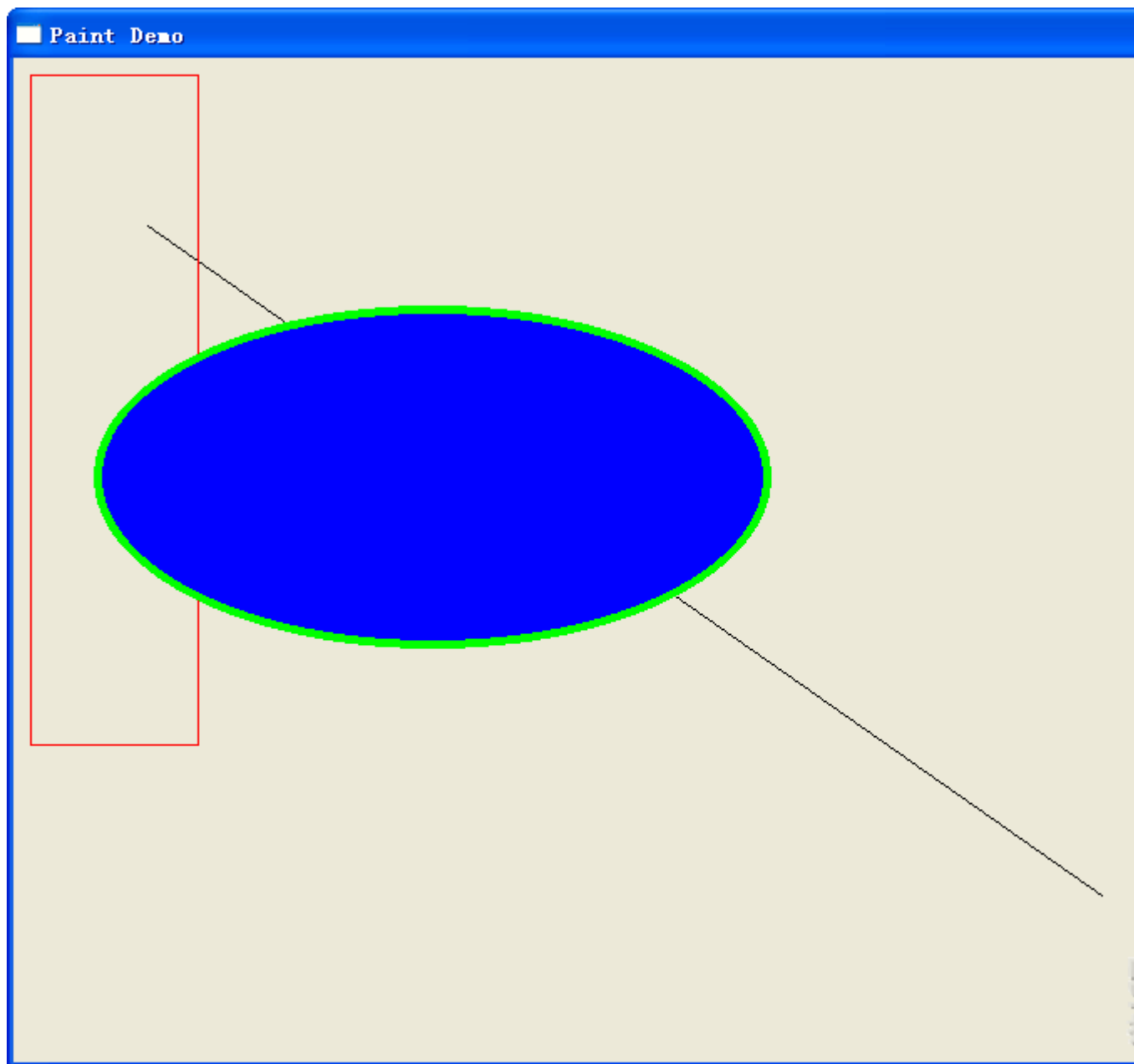
```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.drawLine(80, 100, 650, 500);
}
```

```
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);  
}
```

为了把我们的程序运行起来，下面是 `main()` 函数：

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

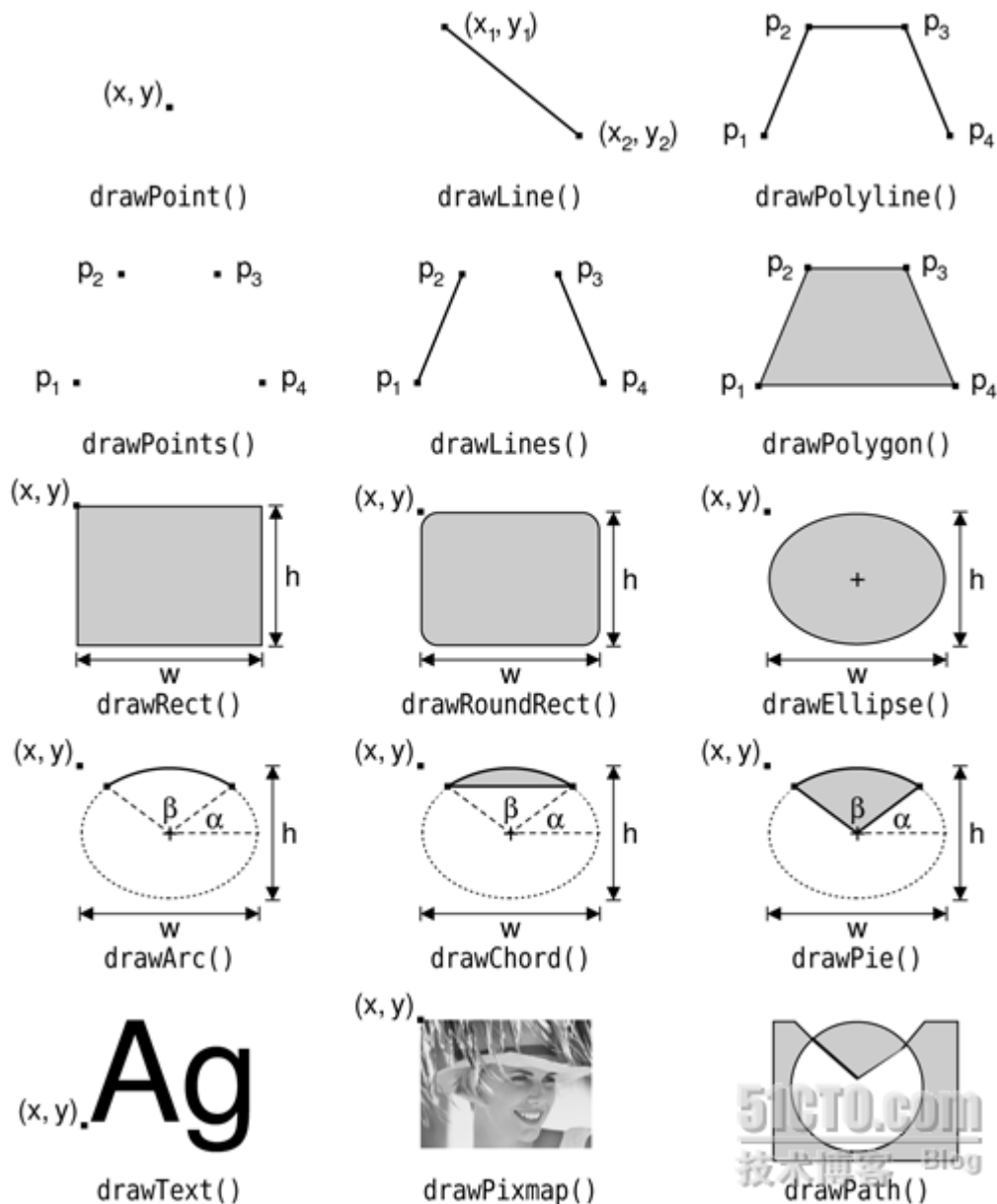
运行结果如下所示：



首先，我们声明了一个 `QPainter` 对象。注意，我们在这个函数的栈空间建立了对象，因此不需要 `delete`。

`QPainter` 接收一个 `QPaintDevice*` 类型的参数。`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画，而现在我们希望在这个 `widget` 上画，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` 和 `drawEllipse` 等。具体的参数请参阅 API 文档。下图给出了 `QPainter` 的 `draw` 函数的实例，本图来自 *C++ GUI Programming with Qt4, 2nd Edition*。



好了，今天先到这里，我们将在下一章中继续对这个 `paintEvent()` 函数进行说明。

Qt 学习之路(25): QPainter(续)

首先还是要先把上次的代码拿上来。

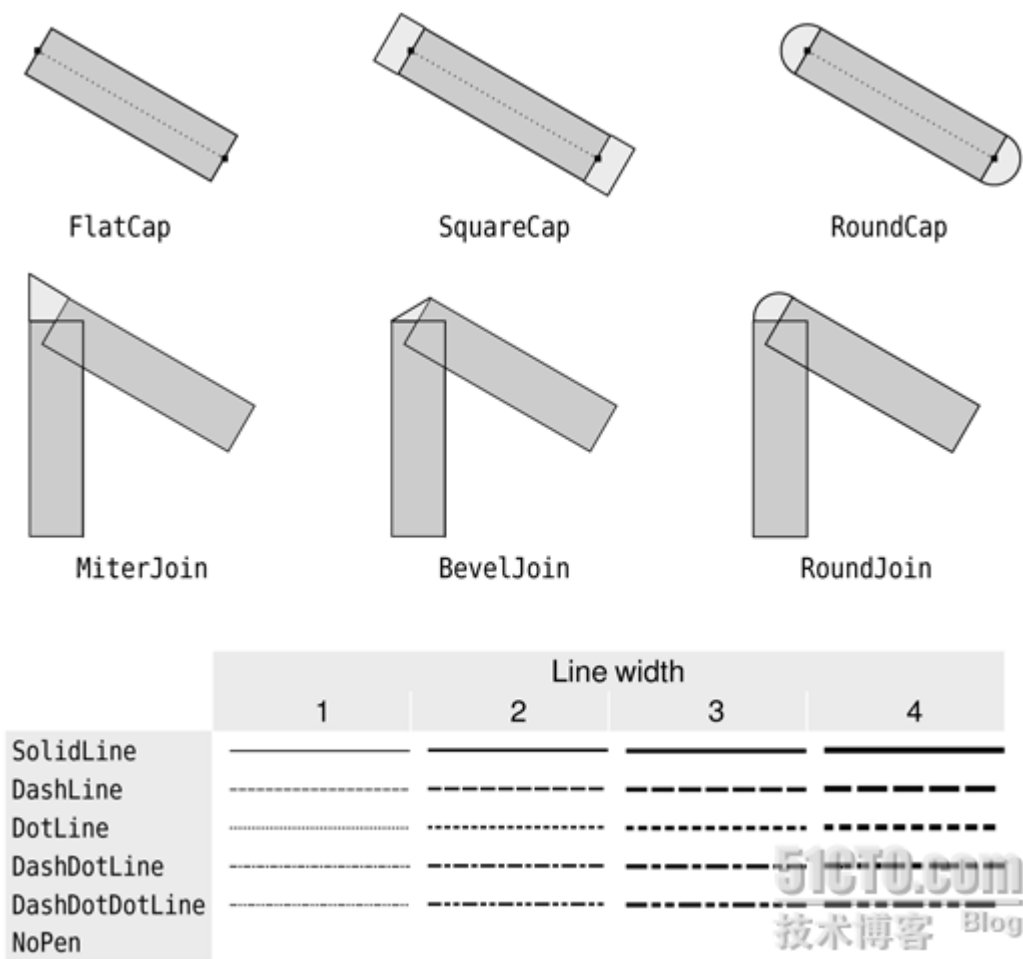
```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.drawLine(80, 100, 650, 500);
    painter.setPen(Qt::red);
    painter.drawRect(10, 10, 100, 400);
    painter.setPen(QPen(Qt::green, 5));
```

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

上次我们说的是 Qt 绘图相关的架构，以及 QPainter 的建立和 drawXXXX 函数。可以看到，基本上代码中已经设计到得函数还剩下两个：setPen()和 setBrush()。现在，我们就要把这两个函数讲解一下。

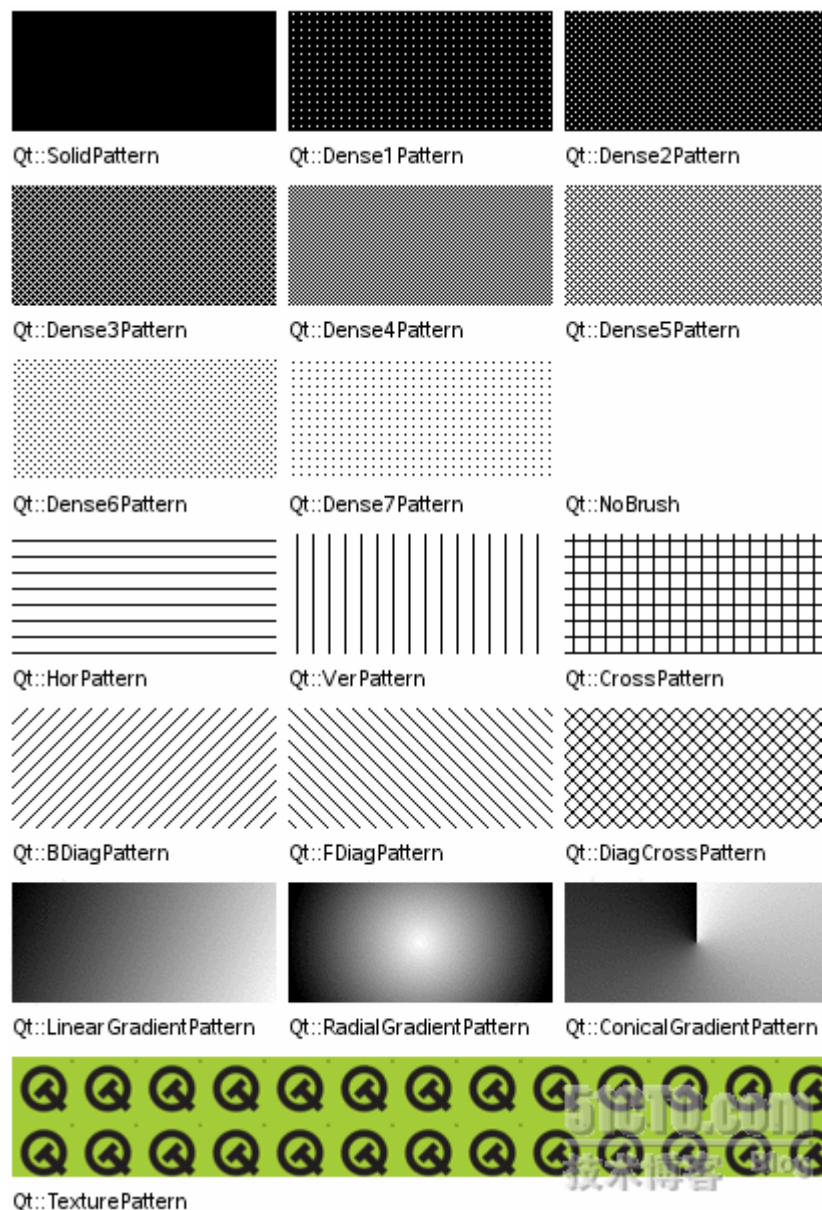
Qt 绘图系统提供了三个主要的参数设置，画笔(pen)、画刷(brush)和字体(font)。这里我们要说明的是画笔和画刷。

所谓画笔，是用于绘制线的，比如线段、轮廓线等，都需要使用画笔绘制。画笔类即 QPen，可以设置画笔的样式，例如虚线、实现之类，画笔的颜色，画笔的转折点样式等。画笔的样式可以在创建时指定，也可以由 setStyle()函数指定。画笔支持三种主要的样式：笔帽(cap)，结合点(join)和线形 (line)。这些样式具体显示如下(图片来自 C++ GUI Programming with Qt4, 2nd Edition):



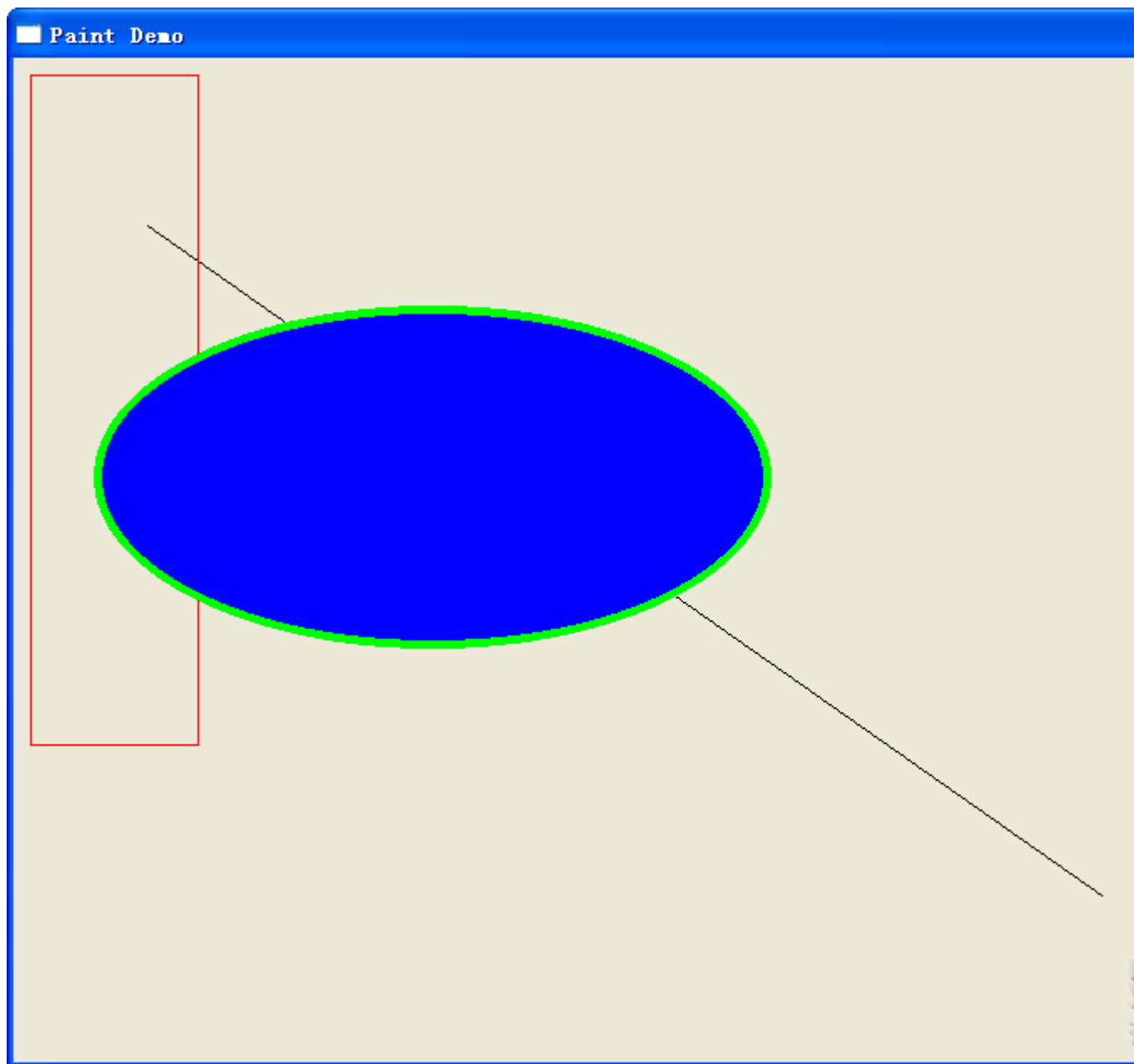
上图共分成三行：第一行是 Cap 样式，第二行是 Join 样式，第三行是 Line 样式。QPen 允许你使用 setCapStyle()、setJoinStyle()和 setStyle()分别进行设置。具体请参加 API 文档。

所谓画刷，主要用来填充封闭的几何图形。画刷主要有两个参数可供设置：颜色和样式。当然，你也可以使用纹理或者渐变色来填充图形。请看下面的图片(图片出自 [Qt API 文档](#)):



这里给出了不同 **style** 的画刷的表现。同画笔类似，这些样式也可用通过一个 **enum** 进行设置。

明白了这些之后我们再来看看我们的代码。首先，我们直接使用 **drawLine()**函数，由于没有设置任何样式，所以使用的是默认的 **1px**，黑色，**solid** 样式画了一条直线；然后使用 **setPen()** 函数，将画笔设置成 **Qt::red**，即红色，画了一个矩形；最后将画笔设置成绿色，**5px**，画刷设置成蓝色，画了一个椭圆。这样便显示出了我们最终的样式：



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。这样，因为椭圆是最后画的，因此在最上方。

在我们学习 OpenGL 的时候，肯定听过这么一句话：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。怎么理解呢？比如，你把颜色设置成红色，那么，直到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 `set` 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椭圆，它的样式还会是绿色 5px 的轮廓和蓝色的填充，除非你显式地调用了 `set` 进行更新。这可能是绘图系统较多的实现方式，因为无论是 OpenGL、QPainter 还是 Java2D，都是这样实现的(DirectX 不大清楚)。

Qt 学习之路(26): 反走样

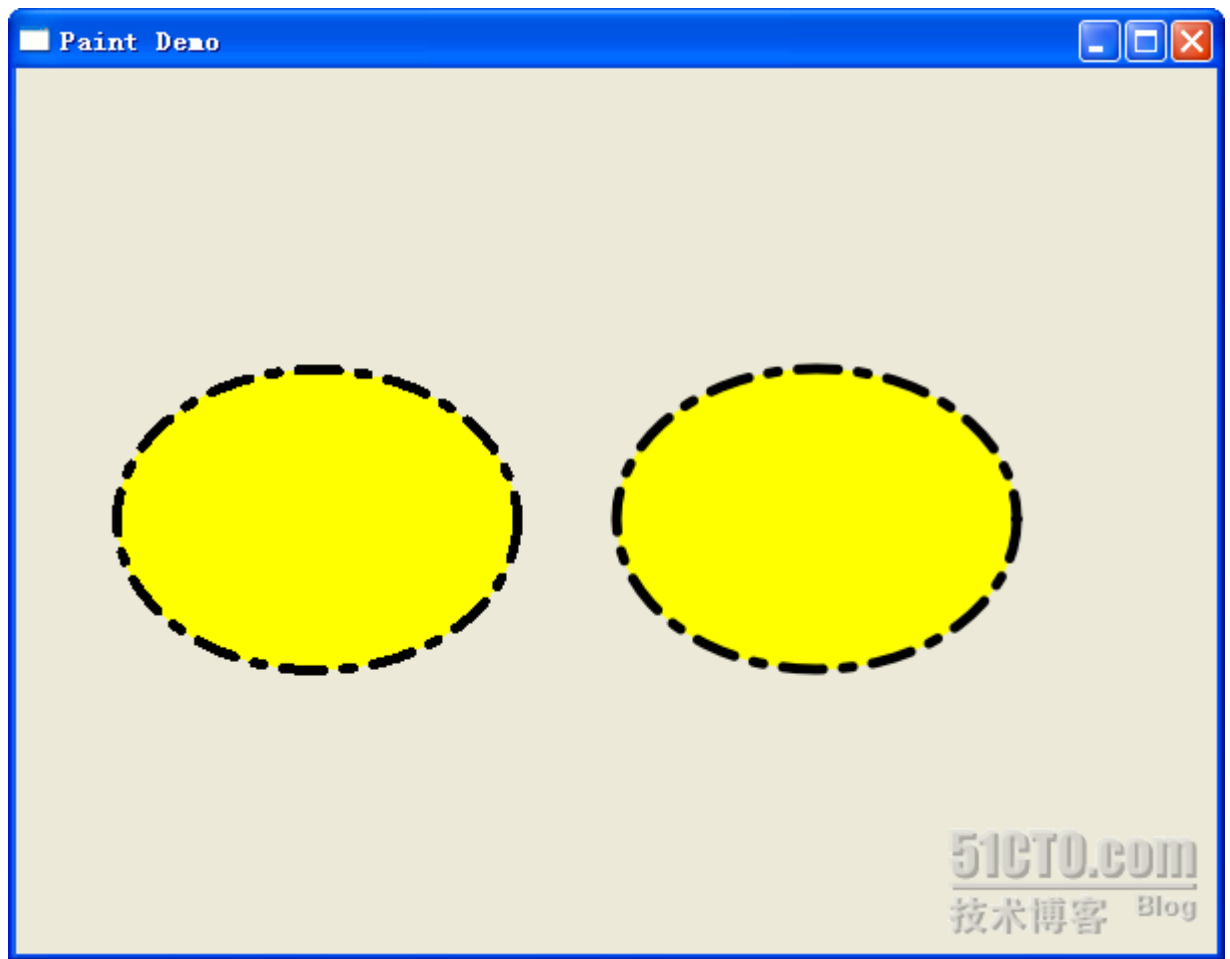
今天继续前面的内容。既然已经进入 2D 绘图部分，那么就先继续研究一下有关 QPainter 的东西吧！

反走样是图形学中的重要概念，用以防止“锯齿”现象的出现。很多系统的绘图 API 里面都会内置了反走样的算法，不过默认一般都是关闭的，Qt 也不例外。下面我们来看看代码。这段代码仅仅给出了 paintEvent 函数，相信你可以很轻松地替换掉前面章节中的相关代码。

```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    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);
}
```

看看运行后的效果：



左边的是没有使用反走样技术的,右边是使用了反走样技术的。二者的差别可以很容易的看出来。

下面来看看相关的代码。为了尝试画笔的样式,这里故意使用了一个新的画笔:

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

我们对照着 API 去看,第一个参数是画笔颜色,这里设置为黑色;第二个参数是画笔的粗细,这里是 5px;第三个是画笔样式,我们使用了 DashDotLine,正如同其名字所示,是一个短线和一个点相间的类型;第四个是 RoundCap,也就是圆形笔帽。然后我们使用一个黄色的画刷填充,画了一个椭圆。

后面的一个和前面的十分相似,唯一的区别是多了一句

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

,不过这句也很清楚,就是设置 Antialiasing 属性为 true。如果你学过图形学就会知道,这个长长的单词就是“反走样”。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。

看到这里你会发现，反走样的效果其实比不走样要好得多，那么，为什么不默认打开反走样呢？这是因为，反走样是一种比较复杂的算法，在一些对图像质量要求不高的应用中，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。

还有一个疑问，既然反走样比不反走样的图像质量高很多，不进行反走样的绘制还有什么作用呢？前面说的是一个方面，也就是，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，是不必须要进行反走样的。另外还有一点，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。请看下面的图片：



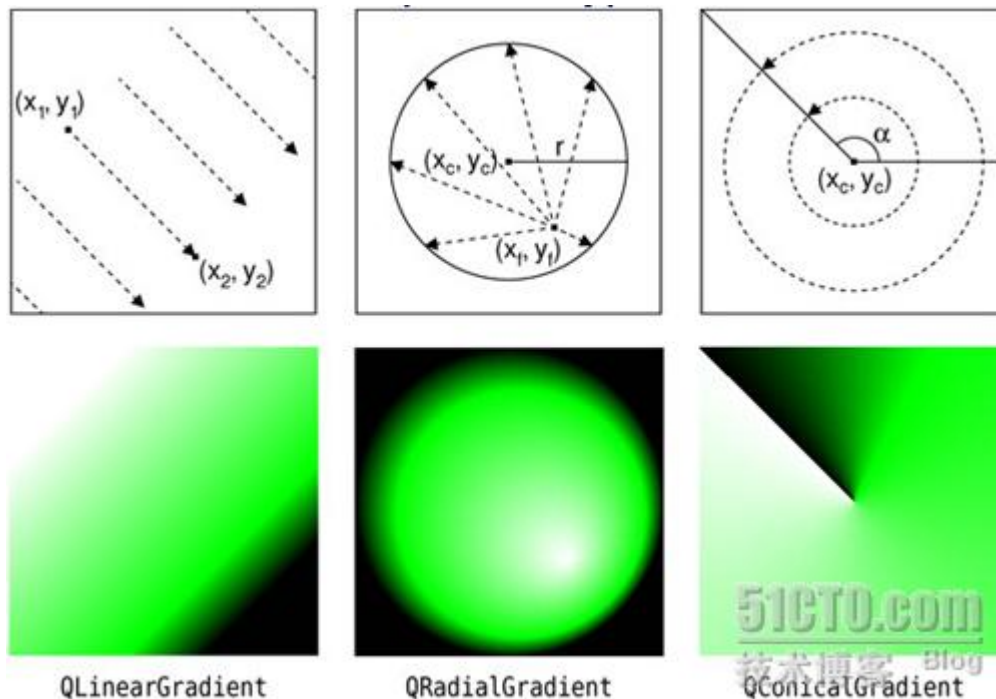
51CTO.com
技术博客 Blog

上图是使用 **Photoshop** 的铅笔和画笔工具画的 **1** 像素的点在放大到 **3200%** 视图下载下来的。**Photoshop** 里面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。如果你了解为什么这样，请查阅计算机图形学里面关于反走样的原理部分。

Qt 学习之路(27): 渐变填充

前面说了有关反走样的相关知识，下面来说一下渐变。渐变是绘图中很常见的一种功能，简单来说就是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，渐变的设置就是在 **QBrush** 里面。

Qt 提供了三种渐变画刷，分别是线性渐变(**QLinearGradient**)、辐射渐变(**QRadialGradient**)、角度渐变(**QConicalGradient**)。如下图所示(图片出自 **C++ GUI Programming with Qt4, 2nd Edition**)：



下面我们来看一下线性渐变 QLinearGradient 的用法。

```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    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);
}
```

同前面一样，这里也仅仅给出了 paintEvent() 函数里面的代码。

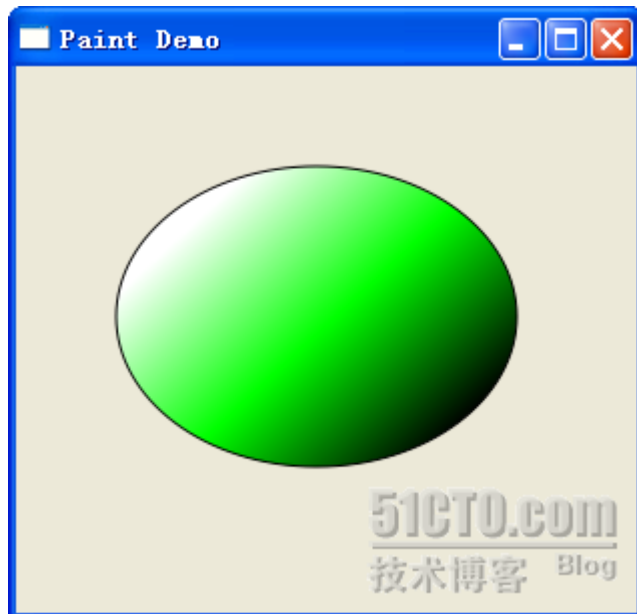
首先我们打开了反走样，然后创建一个 QLinearGradient 对象实例。QLinearGradient 构造函数有四个参数，分别是 x1, y1, x2, y2，即渐变的起始点和终止点。在这里，我们从(60, 50)开始渐变，到(200, 200)止。

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 )
```

它的意思是吧 **position** 位置的颜色设置成 **color**。其中，**position** 是一个 0 - 1 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。比如这个线性渐变，就是说，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。

在创建 **QBrush** 时，把这个渐变对象传递进去，就是我们的结果啦：

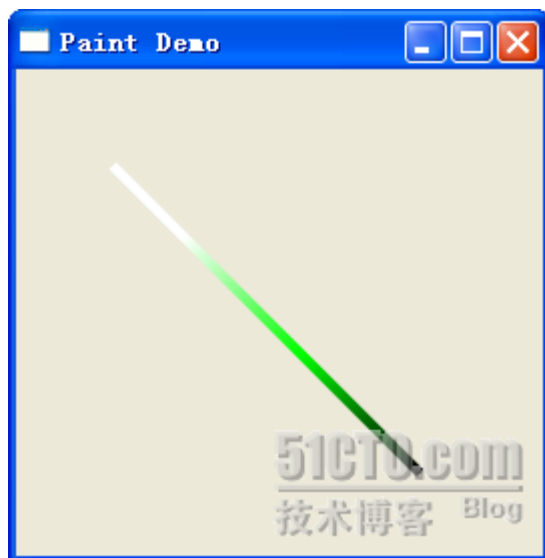


那么，我们怎么让线段也是渐变的呢？要知道，直线是用画笔绘制的啊！这里，如果你仔细查阅了 API 文档就会发现，**QPen** 是接受 **QBrush** 作为参数的。也就是说，你可以利用一个 **QBrush** 创建一个 **QPen**，这样，**QBrush** 所有的填充效果都可以用在画笔上了！

```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    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.setPen(QPen(QBrush(linearGradient), 5));
    painter.drawLine(50, 50, 200, 200);
}
```

看看我们的渐变线吧！



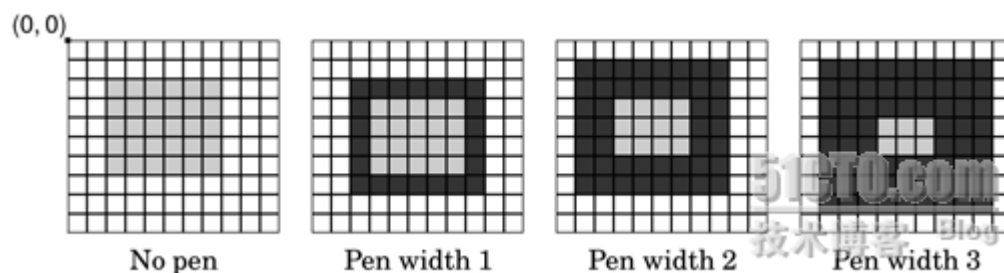
Qt 学习之路(28): 坐标变换

经过前面的章节，我们已经能够画出一些东西来，主要就是使用 **QPainter** 的相关函数。今天，我们要看的是 **QPainter** 的坐标系统。

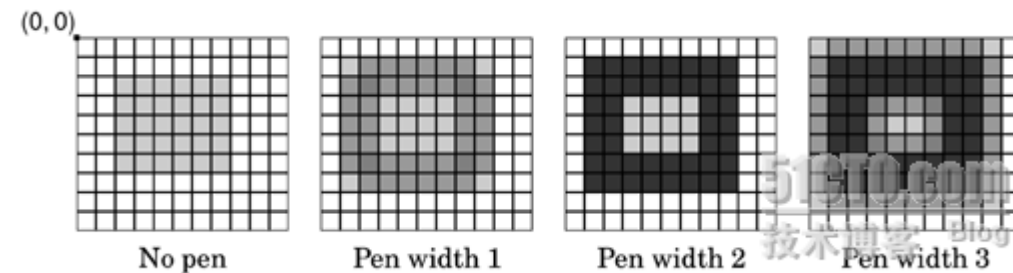
同很多坐标系统一样，**QPainter** 的默认坐标的原点(0, 0)位于屏幕的左上角，X 轴正方向是水平向右，Y 轴正方向是竖直向下。在这个坐标系统中，每个像素占据 1×1 的空间。你可以把它想象成是一张坐标值，其中的每个小格都是 1 个像素。这么说来，一个像素的中心实际上是一个“半像素坐标系”，也就是说，像素(x, y)的中心位置其实是在(x + 0.5, y + 0.5)的位置上。因此，如果我们使用 **QPainter** 在(100, 100)处绘制一个像素，那么，这个像素的中心坐标是(100.5, 100.5)。

这种细微的差别在实际应用中，特别是对坐标要求精确的系统中是很重要的。首先，只有在禁止反走样，也就是默认状态下，才会有这 0.5 像素的偏移；如果使用了反走样，那么，我们画(100, 100)位置的像素时，**QPainter** 会在(99.5, 99.5), (99.5, 100.5), (100.5, 99.5)和(100.5, 100.5)四个位置绘制一个亮色的像素，这么产生的效果就是在这四个像素的焦点处(100, 100)产生了一个像素。如果不需要这个特性，就需要将 **QPainter** 的坐标系平移(0.5, 0.5)。

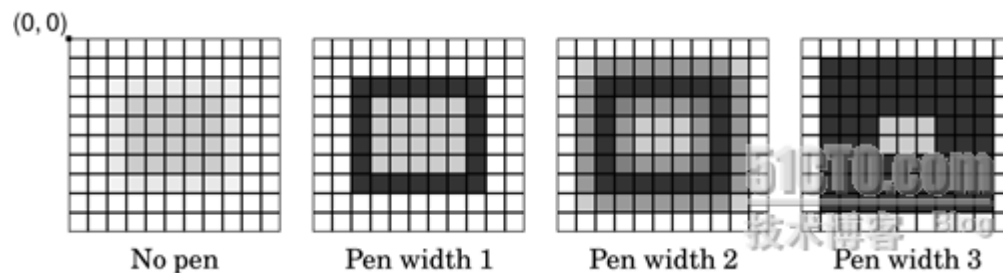
这一特性在绘制直线、矩形等图形的时候都会用到。下图给出了在没有反走样技术时，使用 **drawRect(2, 2, 6, 5)** 绘制一个矩形的示例。在 **No Pen** 的情况下，请注意矩形左上角的像素是在(2, 2)，其中心位置是在(2.5, 2.5)的位置。然后注意下有不同的 **Pen** 的值的绘制样式，在 **Pen** 宽为 1 时，实际画出的矩形的面积是 7×6 的(图出自 C++ GUI Programming with Qt4, 2nd Edition):



在具有反走样时，使用 `drawRect(2, 2, 6, 5)` 的效果如下(图出自 C++ GUI Programming with Qt4, 2nd Edition):



注意我们前面说过，通过平移 `QPainter` 的坐标系来消除着 0.5 像素的差异。下面给出了使用 `drawRect(2.5, 2.5, 6, 5)` 在反走样情况下绘制的矩形(图出自 C++ GUI Programming with Qt4, 2nd Edition):



请对比与上图的区别。

在上述的 `QPainter` 的默认坐标系下，`QPainter` 提供了视口(viewport)窗口(window)机制，用于绘制与绘制设备的大小和分辨率无关的图形。视口和窗口是紧密的联系在一起的，它们一般都是矩形。视口是由物理坐标确定其大小，而窗口则是由逻辑坐标决定。我们在使用 `QPainter` 进行绘制时，传给 `QPainter` 的是逻辑坐标，然后，Qt 的绘图机制会使用坐标变换将逻辑坐标转换成物理坐标后进行绘制。

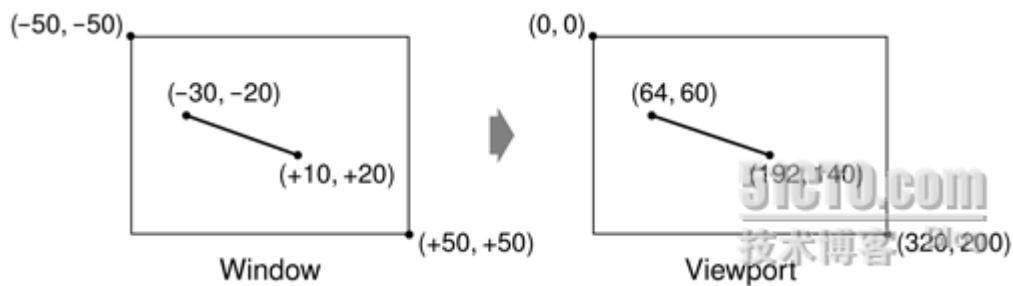
通常，视口和窗口的坐标是一致的。比如一个 600 x 800 的 widget(这是一个 widget，或许是一个对话框，或许是一个面板等等)，默认情况下，视口和窗口都是一个 320 x 200 的矩形，原点都在(0, 0)，此时，视口和窗口的坐标是相同的。

注意到 `QPainter` 提供了 `setWindow()` 和 `setViewport()` 函数，用来设置视口和窗口的矩形大小。比如，在上面所述的 320 x 200 的 widget 中，我们要设置一个从(-50, -50)到(+50, +50)，原点在中心的矩形窗口，就可以使用

```
painter.setWindow(-50, -50, 100, 100);
```

其中，(-50, -50)指明了原点，100, 100 指明了窗口的长和宽。这里的“指明原点”意思是，逻辑坐标的(-50, -50)对应着物理坐标的(0, 0)；“长和宽”说明，逻辑坐标系下的长 100，宽 100 实际上对应物理坐标系的长 320，宽 200。

或许你已经发现这么一个好处，我们可以随时改变 window 的范围，而不改变底层物理坐标系。这就是前面所说的，视口与窗口的作用：“绘制与绘制设备的大小和分辨率无关的图形”，如下图所示(图出自 C++ GUI Programming with Qt4, 2nd Edition):



除了视口与窗口的变化，**QPainter** 还提供了一个“世界坐标系”，同样也可以变换图形。所不同的是，视口与窗口实际上是统一图形在两个坐标系下的表达，而世界坐标系的变换是通过改变坐标系来平移、缩放、旋转、剪切图形。为了清楚起见，我们来看下面一个例子：

```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    QFont font("Courier", 24);
    painter.setFont(font);
    painter.drawText(50, 50, "Hello, world!");
    QTransform transform;
    transform.rotate(+45.0);
    painter.setWorldTransform(transform);
    painter.drawText(60, 60, "Hello, world!");
}
```

为了显示方便，我在这里使用了 **QFont** 改变了字体。**QPainter** 的 **drawText()** 函数提供了绘制文本的功能。它有几种重载形式，我们使用了其中的一种，即制定文本的坐标然后绘制。需要注意的是，这里的坐标是文字**左下角**的坐标(特别提醒这一点，因为很多绘图系统，比如 **Java2D** 都是把左上角作为坐标点的)！下面是运行结果：



我们使用 `QTransform` 做了一个 `rotate` 变换。这个变换就是旋转，而且是顺时针旋转 `45` 度。然后我们使用这个变换设置了 `QPainter` 的世界坐标系，注意到 `QPainter` 是一个状态机，所以这种变换并不会改变之前的状态，因此只有第二个 `Hello, world!` 被旋转了。确切的说，被旋转的是坐标系而不是这个文字！请注意体会这两种说法的不同。

Qt 学习之路(29): 绘图设备

绘图设备是指继承 `QPainterDevice` 的子类。`Qt` 一共提供了四个这样的类，分别是 `QPixmap`、`QBitmap`、`QImage` 和 `QPicture`。其中，`QPixmap` 专门为图像在屏幕上的显示做了优化，而 `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` 的 `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。`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也不需要传指针，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，这里只要知道传递 `QPixmap` 不必使用指针就好了。

`QBitmap` 继承自 `QPixmap`，因此具有 `QPixmap` 的所有特性。`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。

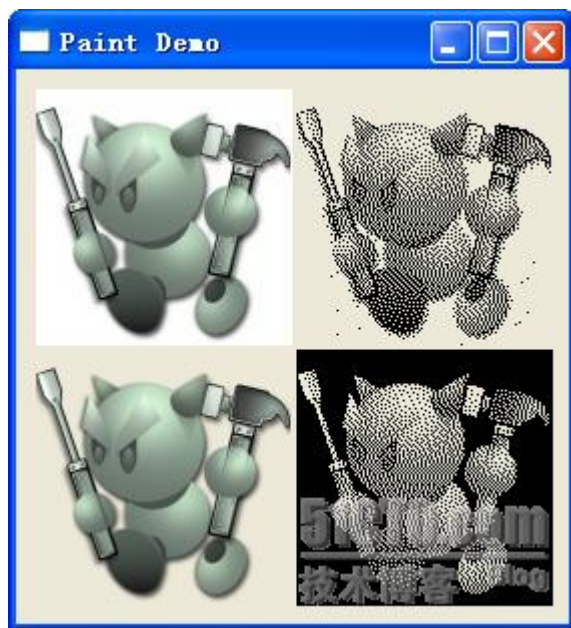
下面我们来看同一个图像文件在 `QPixmap` 和 `QBitmap` 下的不同表现：

```
void PaintedWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    QPixmap pixmap("Cat.png");
    QBitmap bitmap("Cat.png");
    painter.drawPixmap(10, 10, 128, 128, pixmap);
    painter.drawPixmap(140, 10, 128, 128, bitmap);
}
```



```
QPixmap pixmap2("Cat2.png");  
QBitmap bitmap2("Cat2.png");  
painter.drawPixmap(10, 140, 128, 128, pixmap2);  
painter.drawPixmap(140, 140, 128, 128, bitmap2);  
}
```

先来看一下运行结果：



这里我们给出了两张 png 图片。Cat.png 是没有透明色的纯白背景，而 Cat2.png 是具有透明色的背景。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；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。

32-bit

	0xff7aa327	
0xff7aa327	0xffbd9527	0xffedba31

```


 QImage image(3, 3, QImage::Format_RGB32);
 QRgb value;

 value = qRgb(189, 149, 39); // 0xffbd9527
 image.setPixel(1, 1, value);

 value = qRgb(122, 163, 39); // 0xff7aa327
 image.setPixel(0, 1, value);
 image.setPixel(1, 0, value);

 value = qRgb(237, 187, 51); // 0xffedba31
 image.setPixel(2, 1, value);

```



如上图所示(出自 Qt API 文档)，我们声明了一个 `QImage` 对象，大小是 `3 x 3`，颜色模式是 `RGB32`，即使用 `32` 位数值表示一个颜色的 `RGB` 值，也就是说每种颜色使用 `8` 位。然后我们对每个像素进行颜色赋值，从而构成了这个图像。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。

最后一个需要说明的是 `QPicture`。这是一个可以记录和重现 `QPainter` 命令的绘图设备。`QPicture` 将 `QPainter` 的命令序列化到一个 `IO` 设备，保存为一个平台独立的文件格式。这种格式有时候会是“元文件(meta-files)”。Qt 的这种格式是二进制的，不同于某些本地的元文件，Qt 的 `picture` 文件没有内容上的限制，只要是能够被 `QPainter` 绘制的元素，不论是字体还是 `pixmap`，或者是变换，都可以保存进一个 `picture` 中。

`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。回忆下我们这里所说的 `QPaintDevice`，实际上是说可以有 `QPainter` 绘制的对象。`QPicture` 使用系统的分辨率，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);           // paint in picture
painter.drawEllipse(10,20, 80,70); // draw an ellipse
painter.end();                     // painting done
picture.save("drawing.pic");        // save 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");           // load picture
QPainter painter;
painter.begin(&myImage);               // paint in myImage
painter.drawPicture(0, 0, picture);    // draw the picture at (0,0)
painter.end();
```

Qt 学习之路(30): Graphics View Framework

现在基本上也已经到了 2D 绘图部分的尾声，所谓重头戏都是在最后压轴的，现在我们就来看看在绘图部分功能最强大的 **Graphics View**。我们经常说 KDE 桌面，新版本的 KDE 桌面就是建立在 **Graphics View** 的基础之上，可见其强大之处。

Qt 的白皮书里面这样写道：“Qt Graphics View 提供了用于管理和交互大量定制的 2D 图形对象的平面以及可视化显示对象的视图 widget，并支持缩放和旋转功能。Graphics View 使用 BSP（二进制空间划分）树形可非常快速地找到对象，因此即使是包含百万个对象的大型场景，也能实时图形化显示。”

Graphics View 是一个基于 item 的 M-V 架构的框架。

基于 item 意思是，它的每一个组件都是一个 item。这是与 QPainter 的状态机不同。回忆一下，使用 QPainter 绘图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；而对于 Graphics View 来说，相同的过程可以是，首先创建一个场景 scene，然后创建一个 line 对象和一个 polygon 对象，再使用 scene 的 add()函数将 line 和 polygon 添加到 scene，最后通过视口 view 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 draw 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。

M-V 架构的意思是，Graphics View 提供一个 model 和一个 view。所谓 model 就是我们添加的种种对象，所谓 view 就是我们观察这些对象的视口。同一个 model 可以由很多 view 从不同的角度进行观察，这是很常见的需求。使用 QPainter 就很难实现这一点，这需要很复杂的计算，而 Qt 的 Graphics View 就可以很容易的实现。

Graphics View 提供了一个 QGraphicsScene 作为场景，即是我们添加图形的空间，相当于整个世界；一个 QGraphicsView 作为视口，也就是我们观察的窗口，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；一些 QGraphicsItem 作为图形元件，以便 scene 添加，Qt 内置了很多图形，比如 line、polygon 等，都是继承自 QGraphicsItem。

下面我们来看一下代码：

```
#include <QtGui>

class DrawApp : public QWidget {
```

```

public:
    DrawApp();
protected:
    void paintEvent(QPaintEvent *event);
};

DrawApp::DrawApp()
{
}

void DrawApp::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.drawLine(10, 10, 150, 300);
}

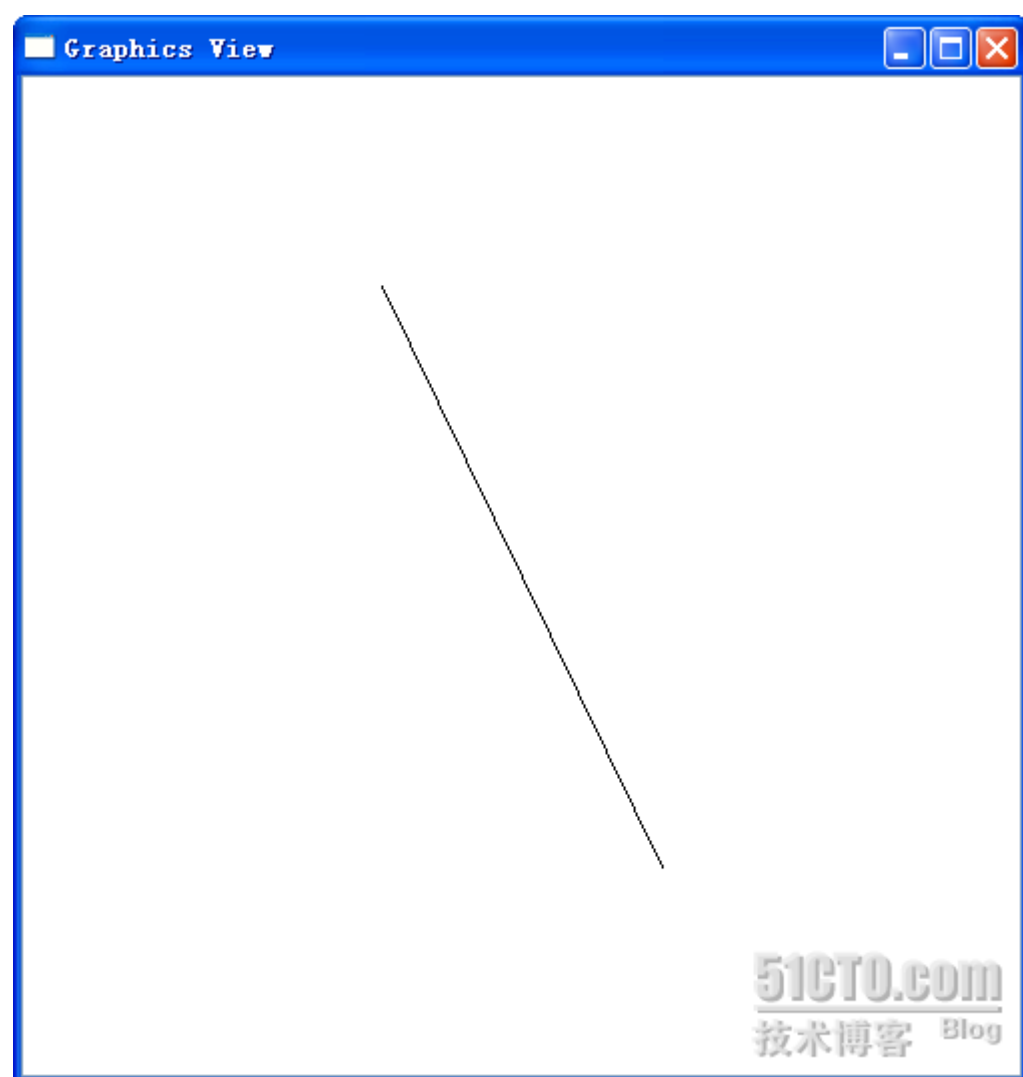
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QGraphicsScene *scene = new QGraphicsScene;
    scene->addLine(10, 10, 150, 300);
    QGraphicsView *view = new QGraphicsView(scene);
    view->resize(500, 500);
    view->setWindowTitle("Graphics View");
    view->show();

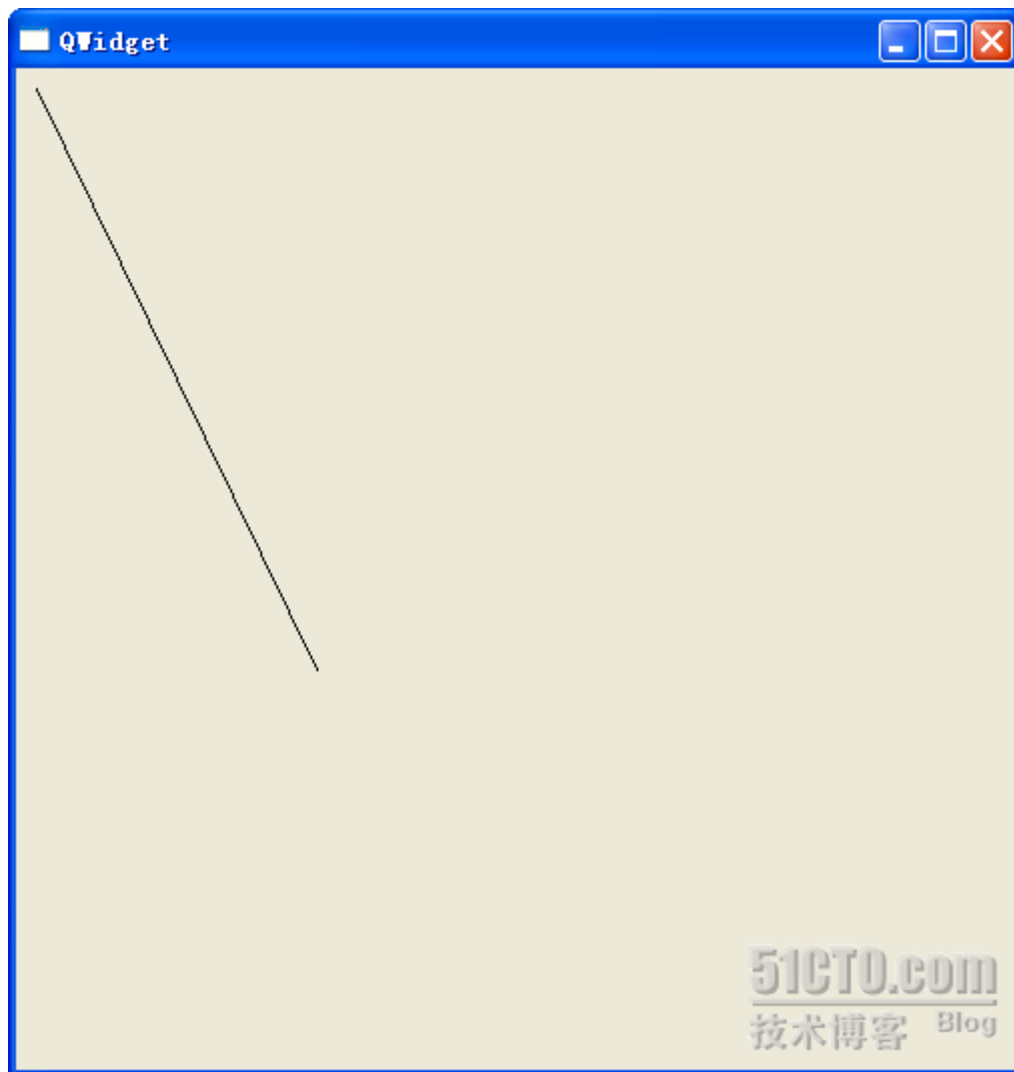
    DrawApp *da = new DrawApp;
    da->resize(500, 500);
    da->setWindowTitle("QWidget");
    da->show();
    return a.exec();
}

```

为了突出重点，我们就直接 `include` 了 `QtGui`，不过在实际应用中不建议这么做。这里提供了直线的两种实现：一个是 `DrawApp` 使用我们前面介绍的技术，重写 `paintEvent()` 函数，这里就不在赘述，重点来看 `main()` 函数里面的实现。

首先，我们创建了一个 `QGraphicsScene` 作为场景，然后在 `scene` 中添加了一个直线，这样就把我们需要的图形元件放到了 `scene` 中。然后创建一个 `QGraphicsView` 对象进行观察。就这样，我们就是用 `Graphics View` 搭建了一个最简单的应用。运行这个程序来看结果：





第一张图是 **Graphics View** 的，第二个是 **DrawApp** 的。虽然这两个直线是同样的坐标，但是，**DrawApp** 按照原始坐标绘制出了直线，而 **Graphics View** 则按照坐标绘制出直线之后，自动将直线居中显示在 **view** 视口。你可以通过拖动 **Graphics View** 来看直线是一直居中显示的。

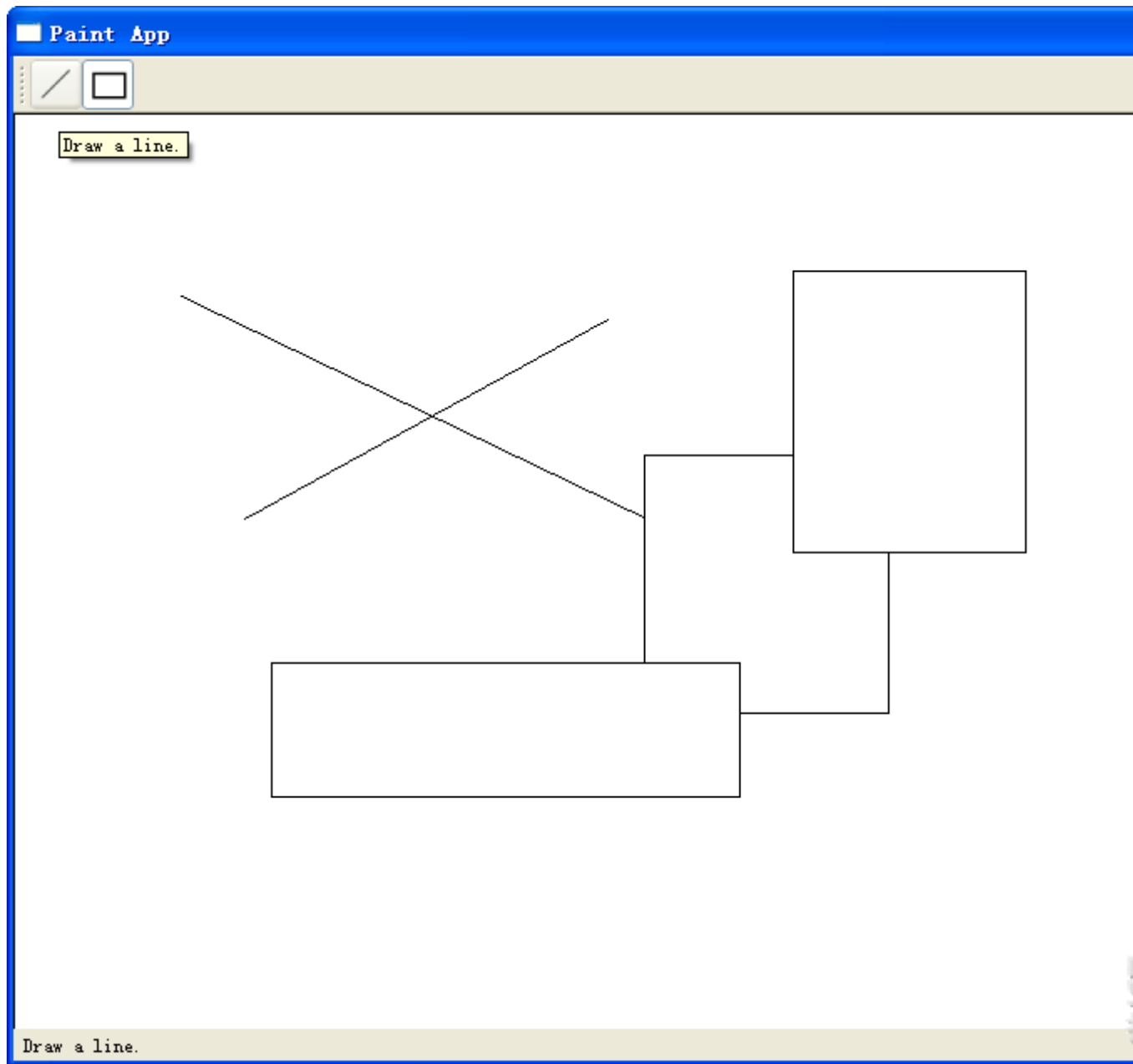
这里仅仅是一个很简单的对比，不过你已经可以看到 **Graphics View** 功能的强大。仅这一个居中的操作，如果你是用 **QPainter**，就需要很大的计算量了！当然，如果你不需要这种居中，**Graphics View** 也是可以像 **QPainter** 绘制的一样进行显示的。

Qt 学习之路(31): 一个简易画板的实现(QWidget)

说实话，本来我是没有打算放一个很大的例子的，一则比较复杂，二来或许需要很多次才能说得完。不过，现在已经说完了绘图部分，所以计划还是上一个这样的例子。这里我会只做出一个简单的画板程序，大体上就是能够画直线和矩形吧。这样，我计划分成两种实现，一是使用普通的 **QWidget** 作为画板，第二则是使用 **Graphics View Framework** 来实现。因为前面有朋友说不大明白 **Graphics View** 的相关内容，所以计划如此。

好了，现在先来看看我们的主体框架。我们的框架还是使用 **Qt Creator** 创建一个 **Gui Application** 工程。

简单的 `main()` 函数就不再赘述了，这里首先来看 `MainWindow`。顺便说一下，我一般不会使用 `ui` 文件，所以这些内容都是手写的。首先先来看看最终的运行结果：



或许很简单，但是至少我们能够把前面所说的各种知识串连起来，这也就达到目的了。

现在先来看看 `MainWindow` 的代码：

`mainwindow.h`

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui>
```

```

#include "shape.h"
#include "paintwidget.h"

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);

signals:
    void changeCurrentShape(Shape::Code newShape);

private slots:
    void drawLineActionTriggered();
    void drawRectActionTriggered();

};

#endif // MAINWINDOW_H

```

mainwindow.cpp

```

#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QToolBar *bar = this->addToolBar("Tools");
    QActionGroup *group = new QActionGroup(bar);

    QAction *drawLineAction = new QAction("Line", bar);
    drawLineAction->setIcon(QIcon(":/line.png"));
    drawLineAction->setToolTip(tr("Draw a line."));
    drawLineAction->setStatusTip(tr("Draw a line."));
    drawLineAction->setCheckable(true);
    drawLineAction->setChecked(true);
    group->addAction(drawLineAction);

    bar->addAction(drawLineAction);
    QAction *drawRectAction = new QAction("Rectangle", bar);
    drawRectAction->setIcon(QIcon(":/rect.png"));
    drawRectAction->setToolTip(tr("Draw a rectangle."));
    drawRectAction->setStatusTip(tr("Draw a rectangle."));
}

```

```

drawRectAction->setCheckable(true);
group->addAction(drawRectAction);
bar->addAction(drawRectAction);

QLabel *statusMsg = new QLabel;
statusBar()->addWidget(statusMsg);

PaintWidget *paintWidget = new PaintWidget(this);
setCentralWidget(paintWidget);

connect(drawLineAction, SIGNAL(triggered()),
        this, SLOT(drawLineActionTriggered()));
connect(drawRectAction, SIGNAL(triggered()),
        this, SLOT(drawRectActionTriggered()));
connect(this, SIGNAL(changeCurrentShape(Shape::Code)),
        paintWidget, SLOT(setCurrentShape(Shape::Code)));
}

void MainWindow::drawLineActionTriggered()
{
    emit changeCurrentShape(Shape::Line);
}

void MainWindow::drawRectActionTriggered()
{
    emit changeCurrentShape(Shape::Rect);
}

```

应该说，从以往的学习中可以看出，这里的代码没有什么奇怪的了。我们在 **MainWindow** 类里面声明了一个信号，**changeCurrentShape(Shape::Code)**，用于按钮按下后通知画图板。注意，**QAction** 的 **triggered()** 信号是没有参数的，因此，我们需要在 **QAction** 的槽函数中重新 **emit** 我们自己定义的信号。构造函数里面创建了两个 **QAction**，一个是 **drawLineAction**，一个是 **drawRectAction**，分别用于绘制直线和矩形。**MainWindow** 的中心组件是 **PaintWidget**，也就是我们的画图板。下面来看看 **PaintWidget** 类：

paintwidget.h

```

#ifndef PAINTWIDGET_H
#define PAINTWIDGET_H

#include <QtGui>
#include <QDebug>
#include "shape.h"
#include "line.h"
#include "rect.h"

```

```

class PaintWidget : public QWidget
{
    Q_OBJECT

public:
    PaintWidget(QWidget *parent = 0);

public slots:
    void setCurrentShape(Shape::Code s)
    {
        if(s != currShapeCode) {
            currShapeCode = s;
        }
    }

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);

private:
    Shape::Code currShapeCode;
    Shape *shape;
    bool perm;
    QList<Shape*> shapeList;
};

#endif // PAINTWIDGET_H

```

paintwidget.cpp

```

#include "paintwidget.h"

PaintWidget::PaintWidget(QWidget *parent)
    : QWidget(parent), currShapeCode(Shape::Line), shape(NULL), perm(false)
{
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
}

void PaintWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

```



```

        painter.setBrush(Qt::white);
        painter.drawRect(0, 0, size().width(), size().height());
        foreach(Shape * shape, shapeList) {
            shape->paint(painter);
        }
        if(shape) {
            shape->paint(painter);
        }
    }

    void PaintWidget::mousePressEvent(QMouseEvent *event)
    {
        switch(currShapeCode)
        {
            case Shape::Line:
            {
                shape = new Line;
                break;
            }
            case Shape::Rect:
            {
                shape = new Rect;
                break;
            }
        }
        if(shape != NULL) {
            perm = false;
            shapeList<<shape;
            shape->setStart(event->pos());
            shape->setEnd(event->pos());
        }
    }

    void PaintWidget::mouseMoveEvent(QMouseEvent *event)
    {
        if(shape && !perm) {
            shape->setEnd(event->pos());
            update();
        }
    }

    void PaintWidget::mouseReleaseEvent(QMouseEvent *event)
    {

```

```
| perm = true;  
| }
```

`PaintWidget` 类定义了一个 `slot`，用于接收改变后的新的 `ShapeCode`。最主要的是，`PaintWidget` 重定义了三个关于鼠标的事件：`mousePressEvent`，`mouseMoveEvent` 和 `mouseReleaseEvent`。

我们来想象一下如何绘制一个图形：图形的绘制与鼠标操作息息相关。以画直线为例，首先我们需要按下鼠标，确定直线的第一个点，所以在 `mousePressEvent` 里面，我们让 `shape` 保存下 `start` 点。然后在鼠标按下的状态下移动鼠标，此时，直线就会发生变化，实际上是直线的终止点在随着鼠标移动，所以在 `mouseMoveEvent` 中我们让 `shape` 保存下 `end` 点，然后调用 `update()` 函数，这个函数会自动调用 `paintEvent()` 函数，显示出我们绘制的内容。最后，当鼠标松开时，图形绘制完毕，我们将一个标志位置为 `true`，此时说明这个图形绘制完毕。

为了保存我们曾经画下的图形，我们使用了一个 `List`。每次按下鼠标时，都会把图形存入这个 `List`。可以看到，我们在 `paintEvent()` 函数中使用了 `foreach` 遍历了这个 `List`，绘制出历史图形。`foreach` 是 `Qt` 提供的一个宏，用于遍历集合中的元素。

最后我们来看看 `Shape` 类。

`shape.h`

```
| #ifndef SHAPE_H  
| #define SHAPE_H  
|  
| #include <QtGui>  
|  
| class Shape  
| {  
| public:  
|  
|     enum Code {  
|         Line,  
|         Rect  
|     };  
|  
|     Shape();  
|  
|     void setStart(QPoint s)  
|     {  
|         start = s;  
|     }  
|  
|     void setEnd(QPoint e)  
|     {
```

```

        end = e;
    }

    QPoint startPoint()
    {
        return start;
    }

    QPoint endPoint()
    {
        return end;
    }

    void virtual paint(QPainter & painter) = 0;

protected:
    QPoint start;
    QPoint end;
};

#endif // SHAPE_H

```

shape.cpp

```

#include "shape.h"

Shape::Shape()
{
}

```

Shape 类最重要的就是保存了 **start** 和 **end** 两个点。为什么只要这两个点呢？因为我们要绘制的是直线和矩形。对于直线来说，有了两个点就可以确定这条直线，对于矩形来说，有了两个点作为左上角的点和右下角的点也可以确定这个矩形，因此我们只要保存两个点，就足够保存这两种图形的位置和大小的信息。**paint()**函数是 **Shape** 类的一个纯虚函数，子类都必须实现这个函数。我们现在有两个子类：**Line** 和 **Rect**，分别定义如下：

line.h

```

#ifndef LINE_H
#define LINE_H

#include "shape.h"

class Line : public Shape
{
public:

```

```
    Line();

    void paint(QPainter &painter);
};

#endif // LINE_H
```

line.cpp

```
#include "line.h"

Line::Line()
{
}

void Line::paint(QPainter &painter)
{
    painter.drawLine(start, end);
}
```

rect.h

```
#ifndef RECT_H
#define RECT_H

#include "shape.h"

class Rect : public Shape
{
public:
    Rect();

    void paint(QPainter &painter);
};

#endif // RECT_H
```

rect.cpp

```
#include "rect.h"

Rect::Rect()
{
}

void Rect::paint(QPainter &painter)
{
}
```

```
painter.drawRect(start.x(), start.y(),
                  end.x() - start.x(), end.y() - start.y());
}
```

使用 `paint()` 函数，根据两个点的数据，`Line` 和 `Rect` 都可以绘制出它们自身来。此时就可以看出，我们之所以要建立一个 `Shape` 作为父类，因为这两个类有几乎完全相似的数据对象，并且从语义上来说，`Line`、`Rect` 与 `Shape` 也完全是一个 **is-a** 的关系。如果你想要添加颜色等的信息，完全可以在 `Shape` 类进行记录。这也就是类层次结构的好处。

代码很多也会比较乱，附件里面是整个工程的文件，有兴趣的朋友不妨看看哦！

Qt 学习之路(32): 一个简易画板的实现(Graphics View)

这一次将介绍如何使用 `Graphics View` 来实现前面所说的画板。前面说了很多有关 `Graphics View` 的好话，但是没有具体的实例很难说究竟好在哪里。现在我们就把前面的内容使用 `Graphics View` 重新实现一下，大家可以对比一下看有什么区别。

同前面相似的内容就不再叙述了，我们从上次代码的基础上进行修改，以便符合我们的需要。首先来看 `MainWindow` 的代码：

mainwindow.cpp

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QToolBar *bar = this->addToolBar("Tools");
    QActionGroup *group = new QActionGroup(bar);

    QAction *drawLineAction = new QAction("Line", bar);
    drawLineAction->setIcon(QIcon(":/line.png"));
    drawLineAction->setToolTip(tr("Draw a line.));
    drawLineAction->setStatusTip(tr("Draw a line.));
    drawLineAction->setCheckable(true);
    drawLineAction->setChecked(true);
    group->addAction(drawLineAction);

    bar->addAction(drawLineAction);
    QAction *drawRectAction = new QAction("Rectangle", bar);
    drawRectAction->setIcon(QIcon(":/rect.png"));
    drawRectAction->setToolTip(tr("Draw a rectangle.));
    drawRectAction->setStatusTip(tr("Draw a rectangle.));
    drawRectAction->setCheckable(true);
    group->addAction(drawRectAction);
    bar->addAction(drawRectAction);
```

```

    QLabel *statusMsg = new QLabel;
    statusBar()->addWidget(statusMsg);

    PaintWidget *paintWidget = new PaintWidget(this);
    QGraphicsView *view = new QGraphicsView(paintWidget, this);
    setCentralWidget(view);

    connect(drawLineAction, SIGNAL(triggered()),
           this, SLOT(drawLineActionTriggered()));
    connect(drawRectAction, SIGNAL(triggered()),
           this, SLOT(drawRectActionTriggered()));
    connect(this, SIGNAL(changeCurrentShape(Shape::Code)),
           paintWidget, SLOT(setCurrentShape(Shape::Code)));
}

void MainWindow::drawLineActionTriggered()
{
    emit changeCurrentShape(Shape::Line);
}

void MainWindow::drawRectActionTriggered()
{
    emit changeCurrentShape(Shape::Rect);
}

```

由于 `mainwindow.h` 的代码与前文相同，这里就不再贴出。而 `cpp` 文件里面只有少数几行与前文不同。由于我们使用 **Graphics View**，所以，我们必须把 `item` 添加到 `QGprahicsScene` 里面。这里，我们创建了 `scene` 的对象，而 `scene` 对象需要通过 `view` 进行观察，因此，我们需要再使用一个 `QGraphcisView` 对象，并且把这个 `view` 添加到 `MainWindow` 里面。

我们把 `PaintWidget` 当做一个 `scene`，因此 `PaintWidget` 现在是继承 `QGraphicsScene`，而不是前面的 `QWidget`。

`paintwidget.h`

```

#ifndef PAINTWIDGET_H
#define PAINTWIDGET_H

#include <QtGui>
#include <QDebug>

#include "shape.h"
#include "line.h"
#include "rect.h"

```

```

class PaintWidget : public QGraphicsScene
{
    Q_OBJECT

public:
    PaintWidget(QWidget *parent = 0);

public slots:
    void setCurrentShape(Shape::Code s)
    {
        if(s != currShapeCode) {
            currShapeCode = s;
        }
    }

protected:
    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);

private:
    Shape::Code currShapeCode;
    Shape *currItem;
    bool perm;
};

#endif // PAINTWIDGET_H

```

paintwidget.cpp

```

#include "paintwidget.h"

PaintWidget::PaintWidget(QWidget *parent)
    : QGraphicsScene(parent), currShapeCode(Shape::Line), currItem(NULL),
    perm(false)
{
}

void PaintWidget::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    switch(currShapeCode)
    {
        case Shape::Line:
            {

```

```

        Line *line = new Line;
        currItem = line;
        addItem(line);
        break;
    }
    case Shape::Rect:
    {
        Rect *rect = new Rect;
        currItem = rect;
        addItem(rect);
        break;
    }
}
if(currItem) {
    currItem->startDraw(event);
    perm = false;
}
QGraphicsScene::mousePressEvent(event);
}

void PaintWidget::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    if(currItem && !perm) {
        currItem->drawing(event);
    }
    QGraphicsScene::mouseMoveEvent(event);
}

void PaintWidget::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
    perm = true;
    QGraphicsScene::mouseReleaseEvent(event);
}
}

```

我们把继承自 `QWidget` 改成继承自 `QGraphicsScene`，同样也会有鼠标事件，只不过在这里我们把鼠标事件全部转发给具体的 `item` 进行处理。这个我们会在下面的代码中看到。另外一点是，每一个鼠标处理函数都包含了调用其父类函数的语句。

shape.h

```

#ifndef SHAPE_H
#define SHAPE_H

#include <QtGui>

```



```

class Shape
{
public:

    enum Code {
        Line,
        Rect
    };

    Shape();

    virtual void startDraw(QGraphicsSceneMouseEvent * event) = 0;
    virtual void drawing(QGraphicsSceneMouseEvent * event) = 0;
};

#endif // SHAPE_H

```

shape.cpp

```

#include "shape.h"

Shape::Shape()
{
}

```

Shape 类也有了变化：还记得我们曾经说过，Qt 内置了很多 **item**，因此我们不必全部重写这个 **item**。所以，我们要使用 Qt 提供的类，就不需要在我们的类里面添加新的数据成员了。这样，我们就有了不带有额外的数据成员的 **Shape**。那么，为什么还要提供 **Shape** 呢？因为我们在 **scene** 的鼠标事件中需要修改这些数据成员，如果没有这个父类，我们就需要按照 **Code** 写一个长长的 **switch** 来判断是那一个图形，这样是很麻烦的。所以我们依然创建了一个公共的父类，只要调用这个父类的 **draw** 函数即可。

line.h

```

#ifndef LINE_H
#define LINE_H

#include <QGraphicsLineItem>
#include "shape.h"

class Line : public Shape, public QGraphicsLineItem
{
public:
    Line();

    void startDraw(QGraphicsSceneMouseEvent * event);

```

```

    void drawing(QGraphicsSceneMouseEvent * event);
};

#endif // LINE_H

```

line.cpp

```

#include "line.h"

Line::Line()
{
}

void Line::startDraw(QGraphicsSceneMouseEvent * event)
{
    setLine(QLineF(event->scenePos(), event->scenePos()));
}

void Line::drawing(QGraphicsSceneMouseEvent * event)
{
    QLineF newLine(line().p1(), event->scenePos());
    setLine(newLine);
}

```

Line 类已经和前面有了变化,我们不仅仅继承了 **Shape**, 而且继承了 **QGraphicsLineItem** 类。这里我们使用了 C++ 的多继承机制。这个机制是很危险的, 很容易发生错误, 但是这里我们的 **Shape** 并没有继承其他的类, 只要函数没有重名, 一般而言是没有问题的。如果不希望出现不推荐的多继承(不管怎么说, 多继承虽然危险, 但它是符合面向对象理论的), 那就想办法使用组合机制。我们之所以使用多继承, 目的是让 **Line** 类同时具有 **Shape** 和 **QGraphicsLineItem** 的性质, 从而既可以直接添加到 **QGraphicsScene** 中, 又可以调用 **startDraw()** 等函数。

同样的还有 **Rect** 这个类:

rect.h

```

#ifndef RECT_H
#define RECT_H

#include <QGraphicsRectItem>
#include "shape.h"

class Rect : public Shape, public QGraphicsRectItem
{
public:
    Rect();
}

```

```

    void startDraw(QGraphicsSceneMouseEvent * event);
    void drawing(QGraphicsSceneMouseEvent * event);
};

#endif // RECT_H

```

rect.cpp

```

#include "rect.h"

Rect::Rect()
{
}

void Rect::startDraw(QGraphicsSceneMouseEvent * event)
{
    setRect(QRectF(event->scenePos(), QSizeF(0, 0)));
}

void Rect::drawing(QGraphicsSceneMouseEvent * event)
{
    QRectF r(rect().topLeft(),
              QSizeF(event->scenePos().x() - rect().topLeft().x(), event->scenePos().y() - rect().topLeft().y()));
    setRect(r);
}

```

Line 和 **Rect** 类的逻辑都比较清楚，和前面的基本类似。所不同的是，Qt 并没有使用我们前面定义的两个 **Qpoint** 对象记录数据，而是在 **QGraphicsLineItem** 中使用 **QLineF**，在 **QGraphicsRectItem** 中使用 **QRectF** 记录数据。这显然比我们的两个点的数据记录高级得多。其实，我们也完全可以使用这样的数据结构去重定义前面那些 **Line** 之类。

这样，我们的程序就修改完毕了。运行一下你会发现，几乎和前面的实现没有区别。这里说“几乎”，是在第一个点画下的时候，**scene** 会移动一段距离。这是因为 **scene** 是自动居中的，由于我们把 **Line** 的第一个点设置为(0, 0)，因此当我们把鼠标移动后会有一个偏移。

看到这里或许并没有显示出 **Graphics View** 的优势。不过，建议在 **Line** 或者 **Rect** 的构造函数里面加上下面的语句，

```

setFlag(QGraphicsItem::ItemIsMovable, true);
setFlag(QGraphicsItem::ItemIsSelectable, true);

```

此时，你的 **Line** 和 **Rect** 就已经支持选中和拖放了！值得试一试哦！不过，需要注意的是，我们重写了 **scene** 的鼠标控制函数，所以这里的拖动会很粗糙，甚至说是不正确，你需要动动脑筋重新设计我们的类啦！

Qt 学习之路(33): 国际化(上)

2D 绘图部分基本告一段落, 还在想下面的部分要写什么, 本来计划先说下 **view-model** 的相关问题, 但是前面看到有朋友问关于国际化的问题, 所以现在先来说说 **Qt** 的国际化吧!

Qt 中的国际化的方法有很多, 常用的有使用 **QTextCodec** 类和使用 **tr()** 函数。前者将编码名称写到代码里面, 除非你使用 **Unicode** 编码, 否则国际化依然是一个问题; 后者就不会有这个问题, 并且这也是 **Qt** 推荐的做法。因此, 我们主要来说使用 **tr()** 函数的方法进行应用程序的国际化。

我们先来看一个很简单的 **MainWindow**。为了清楚起见, 这里只给出了 **cpp** 文件的内容:

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QMenuBar *menuBar = new QMenuBar(this);
    QMenu *fileMenu = new QMenu(tr("&File"), menuBar);
    QAction *newFile = new QAction(tr("&New..."), fileMenu);
    fileMenu->addAction(newFile);
    QAction *openFile = new QAction(tr("&Open..."), fileMenu);
    fileMenu->addAction(openFile);
    menuBar->addMenu(fileMenu);
    setMenuBar(menuBar);

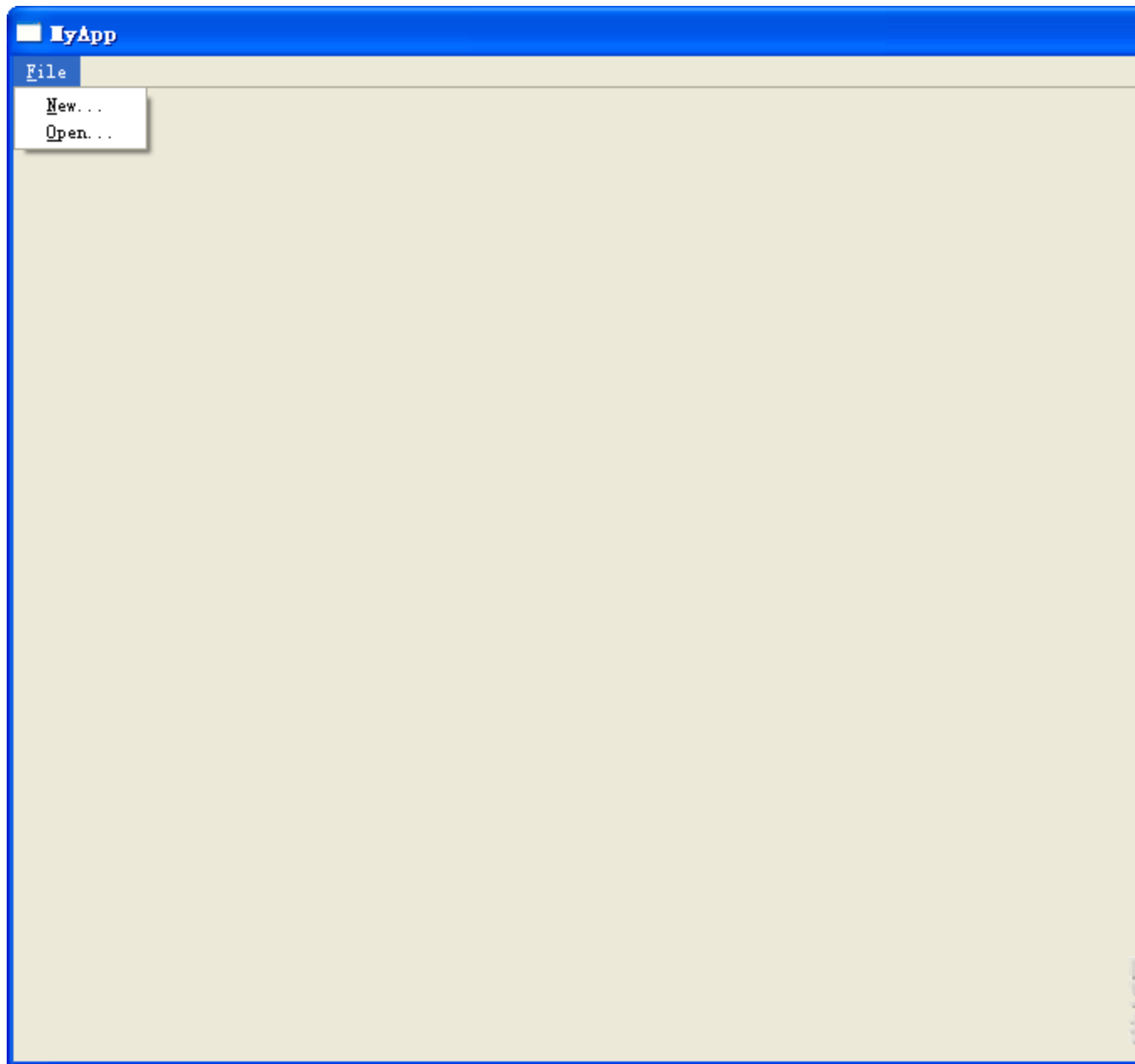
    connect(openFile, SIGNAL(triggered()), this, SLOT(fileOpen()));
}

MainWindow::~MainWindow()
{
}

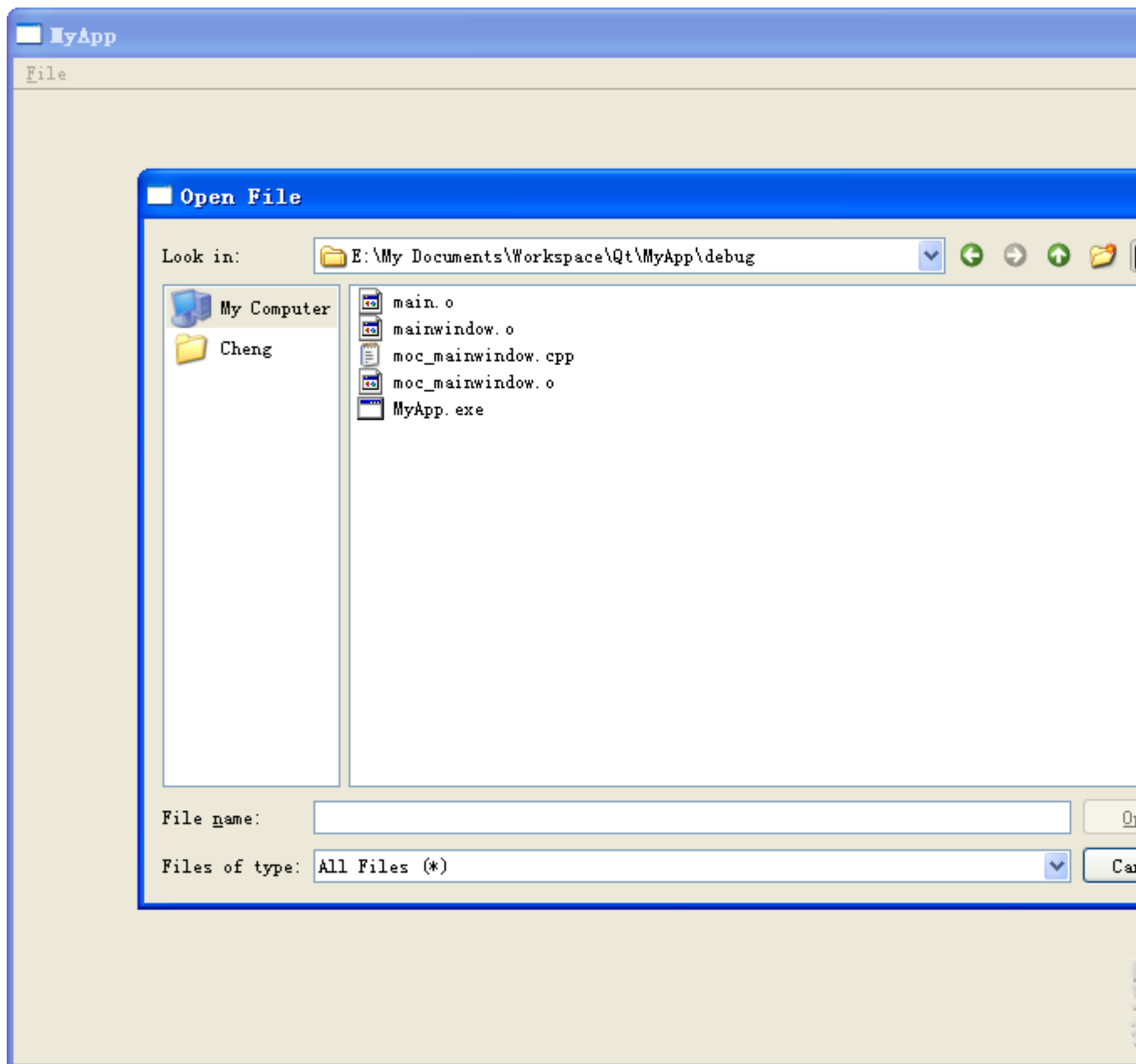
void MainWindow::fileOpen()
{
    QFileDialog *fileDialog = new QFileDialog(this);
    fileDialog->setWindowTitle(tr("Open File"));
    fileDialog->setDirectory(".");
    if(fileDialog->exec() == QDialog::Accepted) {
        QString path = fileDialog->selectedFiles()[0];
        QMessageBox::information(NULL, tr("Path"), tr("You selected\n%1").arg(path));
    } else {
        QMessageBox::information(NULL, tr("Path"), tr("You didn't select a
```

```
ny files."));  
|    }  
|}
```

这是一个很简单的类，运行结果想必大家也都非常清楚：就是一个主窗口，上面有一个菜单栏，一个 **File** 菜单，里面有两个菜单项：



之所以把运行图贴出来，是为了大家能够看清，在代码中的**&**符号实际在界面中显示成为一条下划线，标记出这个菜单或者菜单项的快捷键。按照代码，当我们点击了 **Open** 时，会弹出一个打开文件的对话框：



这里的 **slot** 里面的代码在前文中已经详细介绍过。也许你会问，为什么要用这种麻烦的写法呢？因为我们曾经说过，使用 **static** 函数实际上是直接调用系统的对话框，而这种构造函数法是 Qt 自己绘制的。这对我们后面的国际化是有一定的影响的。

好了，都已经准备好了，下面开始进行国际化。所谓国际化，实际上不仅仅是把界面中的各种文字翻译成另外的语言，还有一些工作是要进行书写方式、货币等的转换。比如，阿拉伯书写时从右向左的，这些在国际化工作中必须完成。但是在这里，我们只进行最简单的工作，就是把界面的文字翻译成中文。

首先，我们需要在 **pro** 文件中增加一行：

```
TRANSLATIONS += myapp.ts
```

myapp.ts 是我们需要创建的翻译文件。这个文件的名字是任意的，不过后缀名需要是 **ts**。然后我们打开命令提示符，进入到工程所在目录，比如我的是 **E:\My Documents\Workspace\Qt\MyApp**，也就是 **pro** 文件所在的文件夹，然后输入命令

```
lupdate MyApp.pro
```

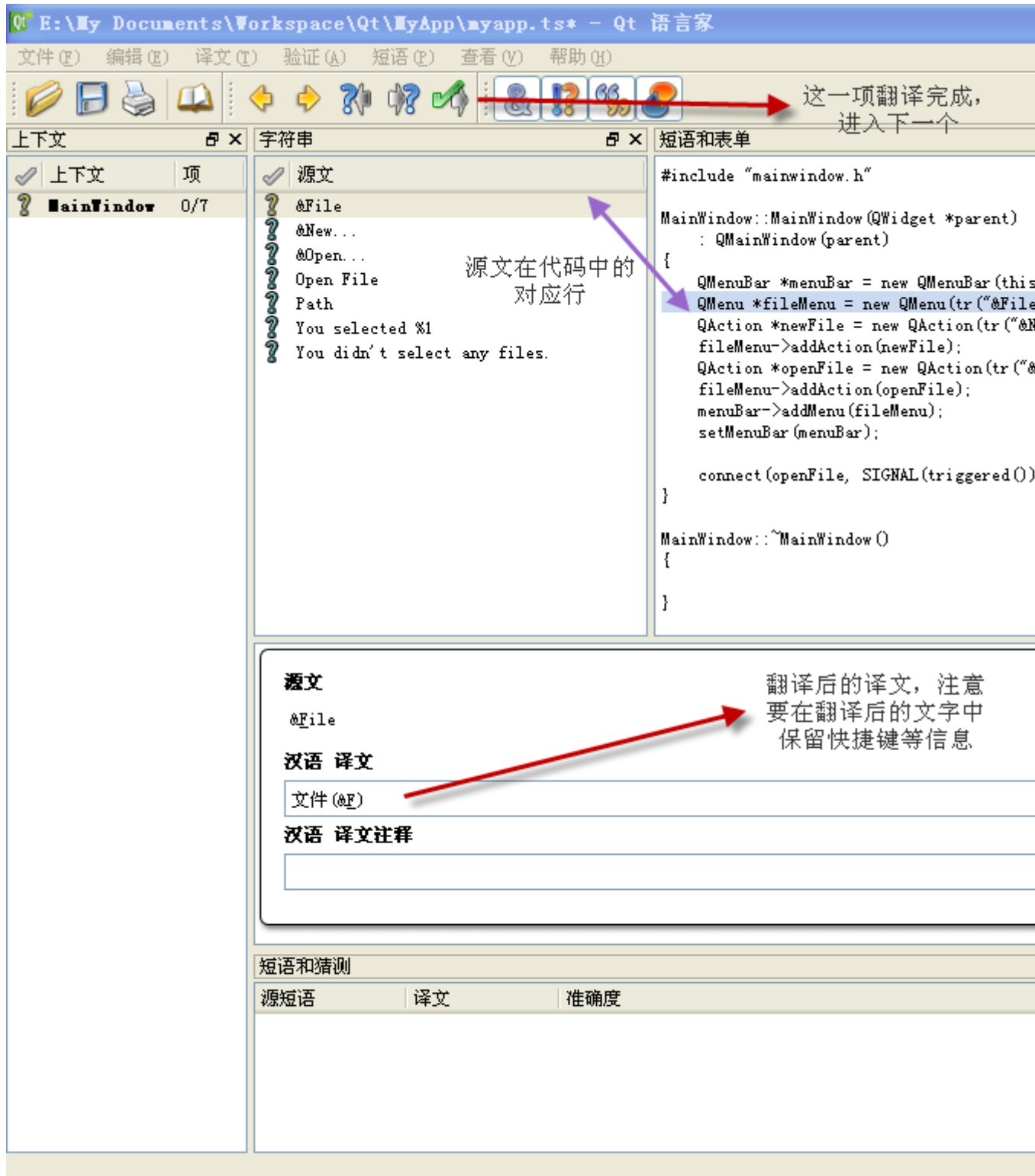
，如果你出现的是命令不存在，请注意将 **Qt** 的 **bin** 目录添加到环境变量中。此时，如果更新的数目，说明 **ts** 文件创建成功：



```
Updating 'myapp.ts' ...  
Found 7 source text(s) (7 new and 0 already existing)
```

最后一行是说，找到 **7** 个需要翻译的原文字，**0** 个已经存在。也就是说，这个文件是新建的。这时你会在工程目录下找到这个 **myapp.ts** 文件。也许你会奇怪，为什么这里还会说已存在的数目呢？因为 **Qt** 这个工具很智能的能够识别出已经存在的文字和修改或新增的文字，这样在以后的工作中就不需要一遍遍重复翻译以前的文字了。这也就是为什么这个工具的名字是“**lupdate**”的原因，因为它是“**update**”，而不仅仅是生成。

如果你有兴趣的话，可以用记事本打开这个 **ts** 文件，这个文件实际上是一个 **XML** 文件，结构很清晰。不过，我们要使用专业的翻译工具进行翻译。**Qt** 提供了一个工具，**Qt Linguist**，你可以在开始菜单的 **Qt** 项下面的 **Tools** 中找到。用它可以打开我们的 **ts** 文件，然后进行我们的翻译工作：



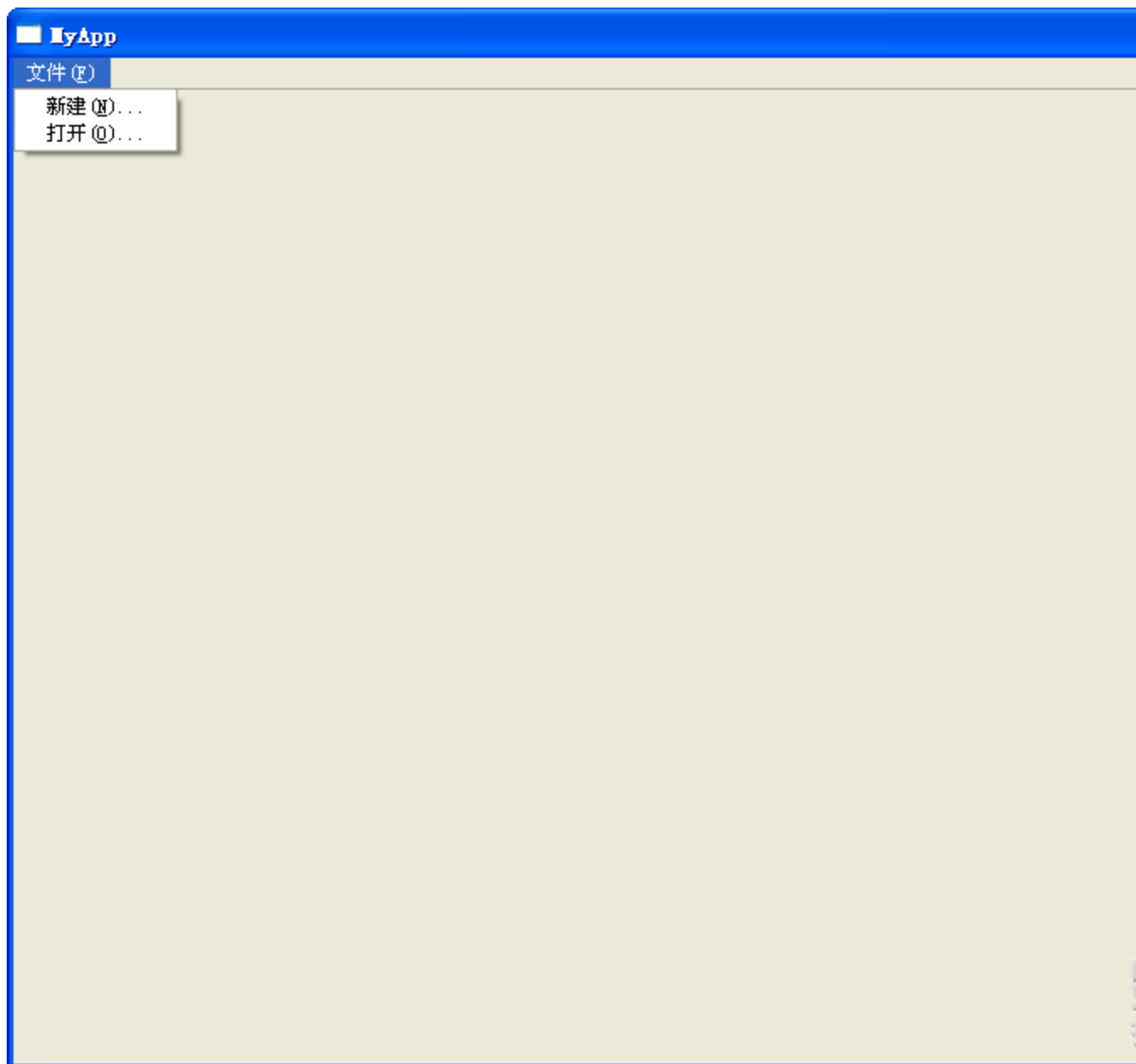
完全翻译完成后保存文件，然后在文件菜单下有个“发布”。点击这个按钮，工程目录下会有一个 myapp.qm 文件，这就是我们翻译得到的文件。Qt 的 qm 文件实际上是二进制格式的，因此它经过了高度的优化，体积很小。

下面我们要修改 main() 函数，使之加载这个 qm 文件：


```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTranslator qtTranslator;
    qtTranslator.load("myapp.qm");
    a.installTranslator(&qtTranslator);
    MainWindow w;
    w.resize(800, 600);
    w.show();
    return a.exec();
}
```

注意，`QTranslator` 类实际是在 `QtCore` 下面的。代码还是很清晰：创建一个 `QTranslator` 对象，然后加载 `qm` 文件，然后将这个对象安装到 `QApplication` 类。好了，现在大功告成，重新编译后运行下程序吧！

咦？怎么还是英文的？哪里有错误了呢？这里往往令人疑惑，其实，这是由于我们使用 `load()` 函数加载 `qm` 文件时使用的是相对路径，这样直接 `load("myapp.qm")`，其实会在当前编译后的 `exe` 所在目录下寻找这个 `qm` 文件，所以，只要我们把 `qm` 文件同 `exe` 放在同一目录下，再次运行：



现在，这个界面已经是中文了吧！其实，这一小细节已经说明，qm 文件其实是动态加载到 exe 文件中的，而不是直接编译进去的。这一点为我们进行动态切换语言提供了基础。

Qt 学习之路(34): 国际化(下)

上次说了国际化的过程，现在来看一下具体的国际化的相关代码。

在代码中，我们使用 `tr()` 将需要翻译的字符串标记出来。`lupdate` 工具就是提取出 `tr()` 函数中的相关字符串。`tr()` 函数是 `QObject` 类的一个 `static` 函数，其签名如下：

```
static QString tr(const char *sourceText, const char *comment = 0, int n = -1);
```

虽然我们只传了一个参数，但是实际上 `tr()` 函数是接受 3 个参数的。第一个参数是我们需要翻译的文字，如果使用 `qm` 文件有对应的字符串，则使用对应的字符串进行替换，否则将显示 `sourceText` 参数指定的字符串。第二个参数是一个注释，用于解释前面的 `sourceText` 的含义，比如 `table` 一词既可以当做桌子翻译，又可以当成表格翻译，这时你就需要提供这个注释。或许你会问，使用翻译工具的时候不是有源代码吗？问题是，有可能人家不使用这个翻译工具，而使用别的工具，这样就不能保证会有这个源代码的预览；并且，你的程序不一定必须要发布源代码的；翻译人员往往只得到我们导出的 `ts` 文件，如果你加上注释，就可以方便翻译人员进行翻译。最后一个参数 `n` 用于指定字符串是否为复数。我们知道，很多语言，比如英语，很多名词的单复数形式是不相同的，为了解决这个问题，Qt 在 `tr()` 函数中提供了一个参数 `n`。请看如下代码：

```
int n = messages.count();
showMessage(tr("%n message(s) saved", "", n));
```

对于 `n` 的值的不同，Qt 会翻译成不同的文字，例如：

n	翻译结果
0	0 message saved
1	1 message saved
2	2 messages saved
5	5 messages saved

`tr()` 函数是 `QObject` 的函数，如果你的类不是继承自 `QObject`，就不能直接使用 `tr()` 函数。比如我们在 `main()` 函数中希望增加一句设置 `MainWindow` 的 `title` 的代码：

```
w.setWindowTitle(tr("MyApp"));
```

直接这样写是无法通过编译的，因为 `main()` 函数是全局函数，所以这个 `tr()` 是找不到的。解决办法一是显式地调用 `QObject` 的函数：

```
w.setWindowTitle(QObject::tr("MyApp"));
```

或者，你可以使用 `QCoreApplication` 的 `translate()` 函数。你一定还记得，我们的 `main()` 函数的第一句总是 `QApplication app;`，其实，`QApplication` 就是 `QCoreApplication` 的子类。所以，我们也能这样去写：

```
w.setWindowTitle(app.translate("MyApp"));
```

由于在 Qt 程序中，`QCoreApplication` 是一个单例类，因此，Qt 提供了一个宏 `qApp`，用于很方便的访问 `QCoreApplication` 的这个单例。所以，在其他文件中，我们也可以直接调用 `qApp.translate()` 来替换 `tr()`，不过这并没有必要。

如果你的翻译文本中包含了需要动态显示的数据，比如我们上次代码中的

```
QMessageBox::information(NULL, tr("Path"), tr("You selected\n%1").arg(path));
```

这句你当然可以写成

```
QMessageBox::information(NULL, tr("Path"), "You selected\n" + path);
```

但这种连接字符串的方式就**不能够**使用 `tr()` 函数了！因此，如果你需要像 C 语言的 `printf()` 函数这种能够格式化输出并且需要翻译时，你必须使用我们例子中的 `%1` 加 `arg()` 函数！

如果你想要翻译函数外部的字符串，你需要使用两个宏 `QT_TR_NOOP()` 和 `QT_TRANSLATE_NOOP()`。前者是用来翻译一个字符串，后者可以翻译多个字符串。它们的使用方法如下：

```
QString FriendlyConversation::greeting(int type)
{
    static const char *greeting_strings[] = {
        QT_TR_NOOP("Hello"),
        QT_TR_NOOP("Goodbye")
    };
    return tr(greeting_strings[type]);
}
```

```
static const char *greeting_strings[] = {
    QT_TRANSLATE_NOOP("FriendlyConversation", "Hello"),
    QT_TRANSLATE_NOOP("FriendlyConversation", "Goodbye")
};

QString FriendlyConversation::greeting(int type)
{
    return tr(greeting_strings[type]);
}

QString global_greeting(int type)
{
    return qApp->translate("FriendlyConversation",
                           greeting_strings[type]);
}
```

好了，以上就是我们用到的大部分函数和宏。除此之外，如果我们运行前面的例子就会发现，实际上我们只是翻译了菜单等内容，打开文件对话框并没有被翻译。原因是我们没有给出国际化的信息。那么，怎么才能让 Qt 翻译这些内建的文字呢？我们要在 `main()` 函数中添加几句：

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

```

    QApplication a(argc, argv);
    QTranslator qtTranslator;
    qtTranslator.load("myapp.qm");
    a.installTranslator(&qtTranslator);
    QTranslator qtTranslator2;
    qtTranslator2.load("qt_zh_CN.qm");
    a.installTranslator(&qtTranslator2);
    MainWindow w;
    w.resize(800, 600);
    w.show();
    return a.exec();
}

```

我们又增加了一个 `QTranslator` 对象。`Qt` 实际上是提供了内置字符串的翻译 `qm` 文件的。我们需要在 `Qt` 安装目录下的 `translations` 文件夹下找到 `qt_zh_CN.qm`，然后同前面一样，将它复制到 `exe` 所在目录。现在再运行一下程序：哈哈已经完全变成中文了吧！

至此，我们的 `Qt` 程序的国际化翻译部分就结束啦！

Qt 学习之路(35): Qt 容器类之顺序存储容器

本来计划先来说下 `model/view` 的，结果发现 `model/view` 涉及到一些关于容器的内容，于是就把容器部分提前了。

容器 `Containers`，有时候也被称为集合 `collections`，指的是能够在内存中存储其他特定类型的对象的对象，这种对象一般是通用的模板类。`C++` 提供了一套完整的解决方案，成为标准模板库 `Standard Template Library`，也就是我们常说的 `STL`。

`Qt` 提供了它自己的一套容器类，这就是说，在 `Qt` 的应用程序中，我们可以使用标准 `C++` 的 `STL`，也可以使用 `Qt` 的容器类。`Qt` 容器类的好处在于，它提供了平台无关的行为，以及**隐式数据共享**技术。所谓平台无关，即 `Qt` 容器类不因编译器的不同而具有不同的实现；所谓“隐式数据共享”，也可以称作“写时复制 `copy on write`”，这种技术允许在容器类中使用传值参数，而不会发生额外的性能损失。`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。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。

今天我们要说的是“顺序储存容器”。所谓顺序存储，就是它存储数据的方式是一个接一个的，线性的。

第一个顺序存储容器是 `QVector<T>`，即向量。`QVector<T>` 是一个类似数组的容器，它将数据存储存储在连续内存区域。同 `C++` 数组不同之处在于，`QVector<T>` 知道它自己的长度，并且可以改变大小。对于获取随机位置的数据，或者是在末尾处添加数据，`QVector<T>` 的效率都是很高的，但是，在中间位置插入数据或者删除数据，它的效率并不是很高。在内存中 `QVector<T>` 的存储类似下图(出自 `C++ GUI Programming with Qt4, 2nd Edition`):

0	1	2	3	4
937.81	25.984	308.74	310.92	40.9

同 STL 的 `vector<T>` 类类似, `QVector<T>` 也提供了 `[]` 的重载, 我们可以使用 `[]` 赋值:

```
QVector<double> v(2);  
v[0] = 1.1;  
v[1] = 1.2;
```

如果实现不知道 `vector` 的长度, 可以创建一个空参数的 `vector`, 然后使用 `append()` 函数添加数据:

```
QVector<double> v;  
v.append(1.1);  
v.append(1.2);
```

在 `QVector<T>` 类中, `<<` 也被重载, 因此, 我们也可以直接使用 `<<` 操作符:

```
QVector<double> v;  
v << 1.1 << 1.2;
```

注意, 如果 `QVector<T>` 中的数据没有被显式地赋值, 那么, 数据项将使用加入类的默认构造函数进行初始化, 如果是基本数据类型和指针, 则初始化为 0.

`QLinkedList<T>` 是另外一种顺序存储容器。在数据结构中, 这是一个链表, 使用指针连接起所有数据。它的内存分布如下(出自 C++ GUI Programming with Qt4, 2nd Edition):



正如数据结构中所描述的那样, `QLinkedList<T>` 的优点是数据的插入和删除很快, 但是随机位置值的访问会很慢。与 `QVector<T>` 不同, `QLinkedList<T>` 并没有提供重载的 `[]` 操作符, 你只能使用 `append()` 函数, 或者 `<<` 操作符进行数据的添加, 或者你也可以使用遍历器, 这个我们将在后面内容中详细描述。

`QList<T>` 是一个同时拥有 `QVector<T>` 和 `QLinkedList<T>` 的大多数有点的顺序存储容器类。它像 `QVector<T>` 一样支持快速的随机访问, 重载了 `[]` 操作符, 提供了索引访问的方式; 它像 `QLinkedList<T>` 一样, 支持快速的添加、删除操作。除非我们需要进行在很大的集合的中间位置的添加、删除操作, 或者是需要所有元素在内存中必须连续存储, 否则我们应该一直使用 `QList<T>`。

`QList<T>` 有几个特殊的情况。一个是 `QStringList`, 这是 `QList<QString>` 的子类, 提供针对 `QString` 的很多特殊操作。`QStack<T>` 和 `QQueue<T>` 分别实现了数据结构中的堆栈和队列, 前者具有 `push()`, `pop()`, `top()` 函数, 后者具有 `enqueue()`, `dequeue()`, `head()` 函数。具体情况请查阅 API 文档。

另外需要指出的一点是，我们所说的模板类中的占位符 **T**，可以使基本数据类型，比如 **int**，**double** 等，也可以指针类型，可以是类类型。如果是类类型的话，必须提供默认构造函数，拷贝构造函数和赋值操作符。**Qt** 的内置类中的 **QByteArray**，**QDateTime**，**QRegExp**，**QString** 和 **QVariant** 是满足这些条件的。但是，**QObject** 的子类并不符合这些条件，因为它们通常缺少拷贝构造函数和赋值操作符。不过这并不是一个问题，因为我们可以存储 **QObject** 的指针，而不是直接存储值。**T** 也可以是一个容器，例如：

```
QList<QVector<int> > list;
```

注意，在最后两个 **>** 之间有一个空格，这是为了防止编译器把它解析成 **>>** 操作符。这个空格是必不可少的，切记切记！

下面我们来看一个类(出自 **C++ GUI Programming with Qt4, 2nd Edition**):

```
class Movie
{
public:
    Movie(const QString &title = "", int duration = 0);

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    void setDuration(int duration) { myDuration = duration; }
    QString duration() const { return myDuration; }

private:
    QString myTitle;
    int myDuration;
};
```

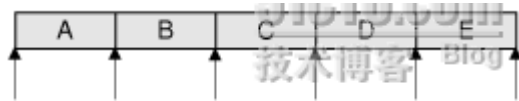
我们能不能把这个类放进 **Qt** 容器类呢？答案是肯定的。下面我们来对照着前面所说的要求：第一，虽然这个类的构造函数有两个参数，但是这两个参数都有默认值，因此，像 **Movie()** 这种写法是允许的，所以，它有默认构造函数；第二，这个类表面上看上去没有拷贝构造函数和赋值操作符，但是 **C++** 编译器会为我们提供一个默认的实现，因此这个条件也是满足的。对于这个类而言，默认拷贝构造函数已经足够，无需我们自己定义。所以，我们可以放心的把这个类放进 **Qt** 的容器类。

Qt 学习之路(36): Qt 容器类之遍历器和隐式数据共享

前面说过，**Qt** 容器类提供了两种遍历器：**Java** 风格的和 **STL** 风格的。前者比较容易使用，后者则可以用在一些通过算法中，功能比较强大。

对于每一个容器类，都有与之相对应的遍历器：只读遍历器和读写遍历器。只读遍历器有 **QVectorIterator<T>**，**QLinkedListIterator<T>** 和 **QListIterator<T>** 三种；读写遍历器同样也有三种，只不过名字中具有一个 **Mutable**，即 **QMutableVectorIterator<T>**，**QMutableLinkedListIterator<T>** 和 **QMutableListIterator<T>**。这里我们只讨论 **QList** 的遍历器，其余遍历器具有几乎相同的 **API**。

Java 风格的遍历器的位置如下图所示(出自 C++ GUI Programming with Qt4, 2nd Edition):



可以看出, **Java** 风格的遍历器, 遍历器不指向任何元素, 而是指向第一个元素之前、两个元素之间或者是最后一个元素之后的位置。使用 **Java** 风格的遍历器进行遍历的典型代码是:

```
QList<double> list;
// ...
QListIterator<double> i(list);
while (i.hasNext()) {
    doSomethingWith(i.next());
}
```

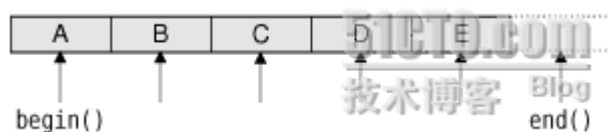
这个遍历器默认指向第一个元素, 使用 `hasNext()` 和 `next()` 函数从前向后遍历。你也可以使用 `toBack()` 函数让遍历器指向最后一个元素的后面的位置, 然后使用 `hasPrevious()` 和 `previous()` 函数进行遍历。

这是只读遍历器, 而读写遍历器则可以在遍历的时候进行增删改的操作, 例如:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    if (i.next() < 0.0)
        i.remove();
}
```

当然, 读写遍历器也是可以从后向前遍历的, 具体 **API** 和前面的几乎相同, 这里就不再赘述。

对应于 **Java** 风格的遍历器, 每一个顺序容器类 `C<T>` 都有两个 **STL** 风格的遍历器: `C<T>::iterator` 和 `C<T>::const_iterator`。正如名字所暗示的那样, `const_iterator` 不允许我们对遍历的数据进行修改。`begin()` 函数返回指向第一个元素的 **STL** 风格的遍历器, 例如 `list[0]`, 而 `end()` 函数则会返回指向最后一个之后的元素的 **STL** 风格的遍历器, 例如如果一个 `list` 长度为 5, 则这个遍历器指向 `list[5]`。下图所示 **STL** 风格遍历器的合法位置:



如果容器是空的, `begin()` 和 `end()` 是相同的。这也是用于检测容器是否为空的方法之一, 不过调用 `isEmpty()` 函数会更加方便。

STL 风格遍历器的语法类似于使用指针对数组的操作。我们可以使用++和--运算符使遍历器移动到下一位置，遍历器的返回值是指向这个元素的指针。例如 QVector<T>的 iterator 返回值是 T * 类型，而 const_iterator 返回值是 const T * 类型。

一个典型的使用 STL 风格遍历器的代码是：

```
QList<double>::iterator i = list.begin();
while (i != list.end()) {
    *i = qAbs(*i);
    ++i;
}
```

对于某些返回容器的函数而言，如果需要使用 STL 风格的遍历器，我们需要建立一个返回值的拷贝，然后再使用遍历器进行遍历。如下面的代码所示：

```
QList<int> list = splitter->sizes();
QList<int>::const_iterator i = list.begin();
while (i != list.end()) {
    doSomething(*i);
    ++i;
}
```

而如果你直接使用返回值，就像下面的代码：

```
// WRONG
QList<int>::const_iterator i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
    doSomething(*i);
    ++i;
}
```

这种写法一般不是你所期望的。因为 sizes()函数会返回一个临时对象，当函数返回时，这个临时对象就要被销毁，因此调用临时对象的 begin()函数是相当不明智的做法。并且这种写法也会有性能问题，因为 Qt 每次循环都要重建临时对象。因此请注意，**如果要使用 STL 风格的遍历器，并且要遍历作为返回值的容器，就要先创建返回值的拷贝，然后进行遍历。**

在使用 Java 风格的只读遍历器时，我们不需要这么做，因此系统会自动为我们创建这个拷贝，所以，我们只需很简单的按下面的代码书写：

```
QListIterator<int> i(splitter->sizes());
while (i.hasNext()) {
    doSomething(i.next());
}
```

这里我们提出要建立容器的拷贝，似乎是一项很昂贵的操作。其实并不然。还记得我们上节说过一个隐式数据共享吗？Qt 就是使用这个技术，让拷贝一个 Qt 容器类和拷贝一个指针那么快速。如果我们只进行读操作，数据是不会被复制的，只有当这些需要复制的数据需要进行写操作，这些数据才会被真正的复制，而这一切都是自动进行的，也正因为这个原因，隐式数据共享有时也被称为“写时复制”。隐式数据共享不需要我们做任何额外的操作，它是自动进行的。隐式数据共享让我们有一种可以很方便的进行值返回的编程风格：

```
 QVector<double> sineTable()
 {
     QVector<double> vect(360);
     for (int i = 0; i < 360; ++i)
         vect[i] = std::sin(i / (2 * M_PI));
     return vect;
 }
// call
 QVector<double> v = sineTable();
```

Java 中我们经常这么写，这样子也很自然：在函数中创建一个对象，操作完毕后将其返回。但是在 C++ 中，很多人都会说，要避免这么写，因为最后一个 return 语句会进行临时对象的拷贝工作。如果这个对象很大，这个操作会很昂贵。所以，资深的 C++ 高手们都会有一个 STL 风格的写法：

```
 void sineTable(std::vector<double> &vect)
 {
     vect.resize(360);
     for (int i = 0; i < 360; ++i)
         vect[i] = std::sin(i / (2 * M_PI));
 }
// call
 QVector<double> v;
 sineTable(v);
```

这种写法通过传入一个引用避免了拷贝工作。但是这种写法就不那么自然了。而隐式数据共享的使用让我们能够放心的按照第一种写法书写，而不必担心性能问题。

Qt 所有容器类以及其他一些类都使用了隐式数据共享技术，这些类包括 QByteArray, QBrush, QFont, QImage, QPixmap 和 QString。这使得这些类在参数和返回值中使用传值方式相当高效。

不过，为了正确使用隐式数据共享，我们需要建立一个良好的编程习惯。这其中之一就是，对 **list** 或者 **vector** 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，而 **at()** 函数是不能作为左值的，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** 容器，在

数据改变时 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()**。

最后，Qt 提供了一种不使用遍历器进行遍历的方法：**foreach** 循环。这实际上是一个宏，使用代码如下所示：

```
QLinkedList<Movie> list;
Movie movie;
...
foreach (movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

很多语言，特别是动态语言，以及 Java 1.5 之后，都有 **foreach** 的支持。Qt 中使用宏实现了 **foreach** 循环，有两个参数，第一个是单个的对象，成为遍历对象，相当于指向容器元素类型的一个指针，第二个是一个容器类。它的意思很明确：每次取出容器中的一个元素，赋值给前面的遍历元素进行操作。需要注意的是，在循环外面定义遍历元素，对于定义中具有逗号的类而言，如 **QPair<int, double>**，是唯一的选择。

Qt 学习之路(37): Qt 容器类之关联存储容器

今天我们来说 Qt 容器类中的关联存储容器。所谓关联存储容器，就是容器中存储的一般是二元组，而不是单个的对象。二元组一般表述为 **<Key-Value>**，也就是“键-值对”。

首先，我们看看数组的概念。数组可以看成是一种 **<int-Object>** 形式的键-值对，它的 **Key** 只能是 **int**，而值的类型是 **Object**，也就是任意类型(注意，这里我们只是说数组可以是任意类型，这个 **Object** 并不必须是一个对象)。现在我们扩展数组的概念，把 **Key** 也做成任意类型的，而不仅仅是 **int**，这样就是一个关联容器了。如果学过数据结构，典型的关联容器就是散列(**Hash Map**，哈希表)。Qt 提供两种关联容器类型：**QMap<K, T>**和 **QHash<K, T>**。

QMap<K, T>是一种键-值对的数据结构，它实际上使用跳表 **skip-list** 实现，按照 **K** 进行升序的方式进行存储。使用 **QMap<K, T>**的 **insert()**函数可以向 **QMap<K, T>**中插入数据，典型的代码如下：

```
QMap<QString, int> map;
map.insert("eins", 1);
map.insert("sieben", 7);
map.insert("dreiundzwanzig", 23);
```

同样，**QMap<K, T>**也重载了 **[]**运算符，你可以按照数组的复制方式进行使用：

```
map["eins"] = 1;
map["sieben"] = 7;
map["dreiuundzwanzig"] = 23;
```

[]操作符同样也可以像数组一样取值。但是请注意，如果在一个非 `const` 的 `map` 中，使用[]操作符取一个不存在的 `Key` 的值，则这个 `Key` 会被自动创建，并将其关联的 `value` 赋予一个空值。如果要避免这种情况，请使用 `QMap<K, T>` 的 `value()` 函数：

```
int val = map.value("dreiuundzwanzig");
```

如果 `key` 不存在，基本类型和指针会返回 `0`，对象类型则会调用默认构造函数，返回一个对象，与[]操作符不同的是，`value()` 函数不会创建一个新的键-值对。如果你希望让不存在的键返回一个默认值，可以传给 `value()` 函数第二个参数：

```
int seconds = map.value("delay", 30);
```

这行代码等价于：

```
int seconds = 30;
if (map.contains("delay"))
    seconds = map.value("delay");
```

`QMap<K, T>` 中的 `K` 和 `T` 可以是基本数据类型，如 `int`，`double`，可以是指针，或者是拥有默认构造函数、拷贝构造函数和赋值运算符的类。并且 `K` 必须要重载 `<` 运算符，因为 `QMap<K, T>` 需要按 `K` 升序进行排序。

`QMap<K, T>` 提供了 `keys()` 和 `values()` 函数，可以获得键的集合和值的集合。这两个集合都是使用 `QList` 作为返回值的。

`Map` 是单值类型的，也就是说，如果一个新的值分配给一个已存在的键，则旧值会被覆盖。如果你需要让一个 `key` 可以索引多个值，可以使用 `QMultiMap<K, T>`。这个类允许一个 `key` 索引多个 `value`，如：

```
QMultiMap<int, QString> multiMap;
multiMap.insert(1, "one");
multiMap.insert(1, "eins");
multiMap.insert(1, "uno");

QList<QString> vals = multiMap.values(1);
```

`QHash<K, T>` 是使用散列存储的键-值对。它的接口同 `QMap<K, T>` 几乎一样，但是它们两个的实现需求不同。`QHash<K, T>` 的查找速度比 `QMap<K, T>` 快很多，并且它的存储是不排序的。对于 `QHash<K, T>` 而言，`K` 的类型必须重载了 `==` 操作符，并且必须被全局函数 `qHas`

h())所支持，这个函数用于返回 **key** 的散列值。Qt 已经为 **int**、指针、**QChar**、**QString** 和 **QByteArray** 实现了 **qHash()**函数。

QHash<K, T>会自动地为散列分配一个初始大小，并且在插入数据或者删除数据的时候改变散列的大小。我们可以使用 **reserve()**函数扩大散列，使用 **squeeze()**函数将散列缩小到最小大小(这个最小大小实际上是能够存储这些数据的最小空间)。在使用时，我们可以使用 **reserve()**函数将数据项扩大到我们所期望的最大值，然后插入数据，完成之后使用 **squeeze()**函数收缩空间。

QHash<K, T>同样也是单值类型的，但是你可以使用 **insertMulti()**函数，或者是使用 **QMultiHash<K, T>**类来为一个键插入多个值。另外，除了 **QHash<K, T>**，Qt 也提供了 **QCache<K, T>**来提供缓存，**QSet<K>**用于仅存储 **key** 的情况。这两个类同 **QHash<K, T>**一样具有 **K** 的类型限制。

遍历关联存储容器的最简单的办法是使用 **Java** 风格的遍历器。因为 **Java** 风格的遍历器的 **next()**和 **previous()**函数可以返回一个键-值对，而不仅仅是值，例如：

```
QMap<QString, int> map;
...
int sum = 0;
QMapIterator<QString, int> i(map);
while (i.hasNext())
    sum += i.next().value();
```

如果我们并不需要访问键-值对，可以直接忽略 **next()**和 **previous()**函数的返回值，而是调用 **key()**和 **value()**函数即可，如：

```
QMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() > largestValue) {
        largestKey = i.key();
        largestValue = i.value();
    }
}
```

Mutable 遍历器则可以修改 **key** 对应的值：

```
QMutableMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() < 0.0)
        i.setValue(-i.value());
}
```

如果是 STL 风格的遍历器，则可以使用它的 `key()` 和 `value()` 函数。而对于 `foreach` 循环，我们就需要分别对 `key` 和 `value` 进行循环了：

```
QMultiMap<QString, int> map;
...
foreach (QString key, map.keys()) {
    foreach (int value, map.values(key)) {
        doSomething(key, value);
    }
}
```

Qt 学习之路(38): model-view 架构

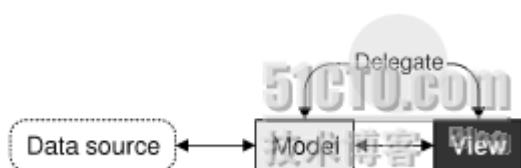
从这一节开始，我们进入 `model-view` 阶段。这一阶段主要还是依据 `C++ GUI Programming with Qt4, 2nd Edition`。

我们的系统有很多数据显示的需求，比如从数据库中把数据取出，然后以自己的方式显示在我们自己的应用程序的界面中。进行这一操作的典型方式是使用 Qt 的 `Item View` 类。

在早期的 Qt 版本中，要实现这个功能，我们需要定义一个 `widget`，然后在这个 `widget` 中保存一个数据对象，比如是个 `list`，然后我们对这个 `list` 进行查找、插入等的操作，或者把修改的地方写回这个 `list`，然后刷新 `widget` 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，如果全部存入一个 `widget` 的数据对象中，效率会很低，并且这样的设计也很难在 `widgets` 之间共享变量，也就是说，如果你要几个组件共享一个数据对象，要么你就要用 `getter` 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，简单来说，这是一个 `model-view-controller` 模型，即模型-视图-控制器。在 `MVC` 中，模型负责获取需要显示的数据，并且能够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。

对于 Qt 而言，它使用的是一个类似于 `MVC` 模型的 `model-view` 架构。其中，`model` 就相当于 `MVC` 架构中的 `model`，而对于控制器部分，Qt 使用的是另外一种抽象，代理 `delegate`。代理被用来提供对 `item` 渲染和编辑的控制。对于每种视图，Qt 都提供了一个默认的代理，对于大多数应用来说，我们只需要使用这个默认的代理即可。这其中的类关系如下图所示(出自 `C++ GUI Programming with Qt 4, 2nd Edition`)



使用 Qt 的 **model-view** 架构，我们可以让 **model** 是取回 **view** 所要展示的数据，这样就可以在不降低性能的情形下处理大量数据。并且你可以把一个 **model** 注册给多个 **view**，让这些 **view** 能够显示同样的数据，也就是为同一个数据提供不同的显示方式。Qt 会自动地对这些 **view** 保持同步，自动刷新所有的 **view** 以显示最新的数据。这样，我们就可以只对 **model** 进行修改，**view** 会自动更新。

在少量数据的情形下，我们不需要动用 **model** 这样重量级的组件。Qt 为了方便起见也提供了 **item view** 类，分别是 **QListWidget**、**QTableWidget** 和 **QTreeView**，使用这些类可以直接对 **item** 进行操作。这种实现很像 Qt 早期版本，组件中包含了相应的 **item**，例如 **QTableWidget** 中包含有 **QTableWidgetItem** 等。但是对于很大的数据，我们则需要使用 Qt 的 **view** 类，比如 **QListView**、**QTableView** 和 **QTreeView**，同时需要提供一个 **model**，可以是自定义 **model**，也可以是 Qt 预置的 **model**。例如，如果数据来自数据库，那么你可以使用 **QTableView** 和 **QSqlTableModel** 这两个类。

今天就说这些，下次我们将开始进入对 **model-view** 架构的具体介绍。

Qt 学习之路(39): QListWidget

前面一节简单概述著名的 **MVC** 模式在 Qt 中的实现，现在我们从 **QListWidget** 开始了解 Qt 提供的一系列方便的 **item view** 类。

第一个要说的是 **QListWidget**。这个类为我们展示一个 **List** 列表的视图。下面还是先看代码：

listwidget.h

```
| #ifndef LISTWIDGET_H
| #define LISTWIDGET_H
|
| #include <QtGui>
|
| class QListWidget : public QWidget
| {
| public:
|     QListWidget();
|
| private:
|     QLabel *label;
|     QListWidget *list;
| };
|
| #endif // LISTWIDGET_H
```

listwidget.cpp

```
| #include "listwidget.h"
|
| QListWidget::QListWidget()
| {
```



```

|     label = new QLabel;
|     label->setFixedWidth(70);
|     list = new QListWidget;
|     list->addItem(new QListWidgetItem(QIcon(":/images/line.PNG"), tr("Line
|     ")));
|     list->addItem(new QListWidgetItem(QIcon(":/images/rect.PNG"), tr("Rect
|     angle")));
|     list->addItem(new QListWidgetItem(QIcon(":/images/oval.PNG"), tr("Ova
|     l")));
|     list->addItem(new QListWidgetItem(QIcon(":/images/tri.PNG"), tr("Trian
|     gle")));
|     QHBoxLayout *layout = new QHBoxLayout;
|     layout->addWidget(label);
|     layout->addWidget(list);
|
|     setLayout(layout);
|
|     connect(list, SIGNAL(currentTextChanged(QString)), label, SLOT(setText
|     (QString)));
| }

```

main.cpp

```

| #include <QtGui>
| #include "listwidget.h"
|
| int main(int argc, char *argv[])
| {
|     QApplication a(argc, argv);
|     ListWidget lw;
|     lw.resize(400, 200);
|     lw.show();
|     return a.exec();
| }

```

一共三个文件，但是都比较清晰。我们先建立了一个 `ListWidget` 类，然后在 `main` 函数中将其显示出来。

`ListWidget` 类中包含一个 `QLabel` 对象和一个 `QListWidget` 对象。创建这个 `QListWidget` 对象很简单，只需要使用 `new` 运算符创建出来，然后调用 `addItem()` 函数即可将 `item` 添加到这个对象中。我们添加的对象是 `QListWidgetItem` 的指针，它有四个重载的函数，我们使用的是其中的一个，它接受两个参数，第一个是 `QIcon` 引用类型，作为 `item` 的图标，第二个是 `QString` 类型，作为这个 `item` 后面的文字说明。当然，我们也可以使用 `insertItem()` 函数在特定的位置动态的增加 `item`，具体使用请查阅 `API` 文档。最后，我们将这个 `QListWidget` 的 `curr`

entTextChanged()信号同 QLabel 的 setText()连接起来，这样，在我们点击 item 的时候，label 上面的文字就可以改变了。



我们还可以设置 `viewModel` 这个参数，来确定使用不同的视图进行显示。比如，我们使用下面的语句：

```
list->setViewMode(QListView::IconMode);
```

再来看看程序界面吧！



Qt 学习之路(40): QTreeWidgetItem

接着前面的内容，今天要说的是另外一个 item view class, `QTreeWidgetItem`。顾名思义，这个类用来展示树型结构。同前面说的 `QListWidget` 类似，这个类需要同另外一个辅助类 `QTreeWidgetItem` 一同使用。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` 的特性的时候，我们可以直接使用 `QTreeWidgetItem` 代替。

下面来看代码。

treewidget.h

```
#ifndef TREEWIDGET_H
#define TREEWIDGET_H

#include <QtGui>

class TreeWidget : public QWidget
{
public:
    TreeWidget();

private:
    QTreeWidget *tree;
};

#endif // TREEWIDGET_H
```

treewidget.cpp

```
#include "treewidget.h"

TreeWidget::TreeWidget()
{
    tree = new QTreeWidget(this);
    tree->setColumnCount(1);
    QTreeWidgetItem *root = new QTreeWidgetItem(tree, QStringList(QString("Root")));
    QTreeWidgetItem *leaf = new QTreeWidgetItem(root, QStringList(QString("Leaf 1")));
    root->addChild(leaf);
    QTreeWidgetItem *leaf2 = new QTreeWidgetItem(root, QStringList(QString("Leaf 1")));
    leaf2->setCheckState(0, Qt::Checked);
    root->addChild(leaf2);
    QList<QTreeWidgetItem*> rootList;
    rootList << root;
    tree->insertTopLevelItems(0, rootList);
}
```

首先，我们在构造函数里面创建了一个 `QTreeWidget` 的实例。然后我们调用 `setColumnCount()` 函数设定栏数。这个函数的效果我们以后再看。然后我们要向 `QTreeWidget` 添加 `QTreeWidgetItem`。`QTreeWidgetItem` 有九个重载的构造函数。我们在这里只是来看看其中的一个，其余的请自行查阅 [API 文档](#)。这个构造函数的签名如下：

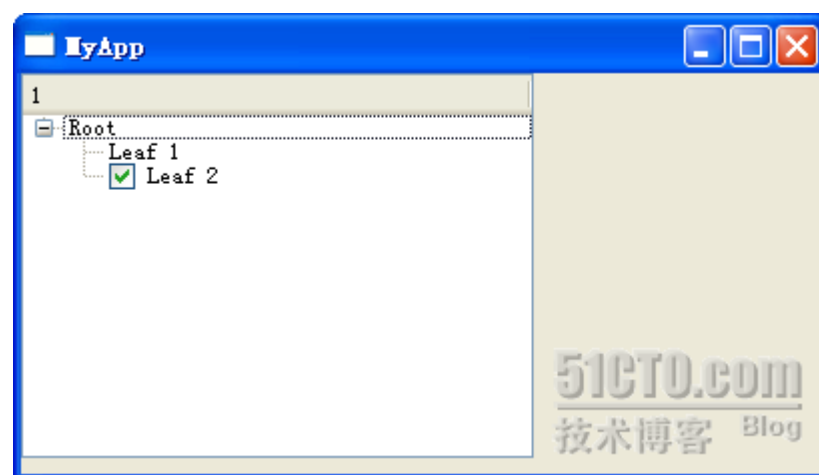
```
QTreeWidgetItem::QTreeWidgetItem ( QTreeWidgetItem * parent, const QStringList & strings, int type = Type );
```

这里有 3 个参数，第一个参数用于指定这个 item 属于哪一个树；第二个参数是指定这个 item 显示的文字；第三个参数用于指定这个 item 的类型。Type 有两个可行的取值：QTreeWidgetItem::Type 和 QTreeWidgetItem::UserType，由于我们并没有定义用户类型，所以只使用其默认值即可。这里你会奇怪，第二个参数为什么是一个 QStringList 类型的，而不是 QString 类型的？我们先不去管它，继续下面的代码。

后面我们又创建了一个 QTreeWidgetItem，注意它的第一个参数不是 QTreeWidgetItem 而是 QTreeWidgetItem 类型的，这就把它的父节点设置为前面我们定义的 root 了。然后我们使用了 setCheckState() 函数，让它变得可以选择，最后使用 addChild() 函数把它添加进来。

最后一步，我们创建了一个 QList 类型，前面的 root 添加进去，然后 insert 到 top items。这里可以想象到，由于这个树组件可以由多个根组成（严格来说这已经不是树了，不过姑且还是叫树吧），所以我们传进来的是一个 list。

好了，编译运行一下我们的代码吧！



样子同我们想象的基本一致，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：

```
#include "listwidget.h"

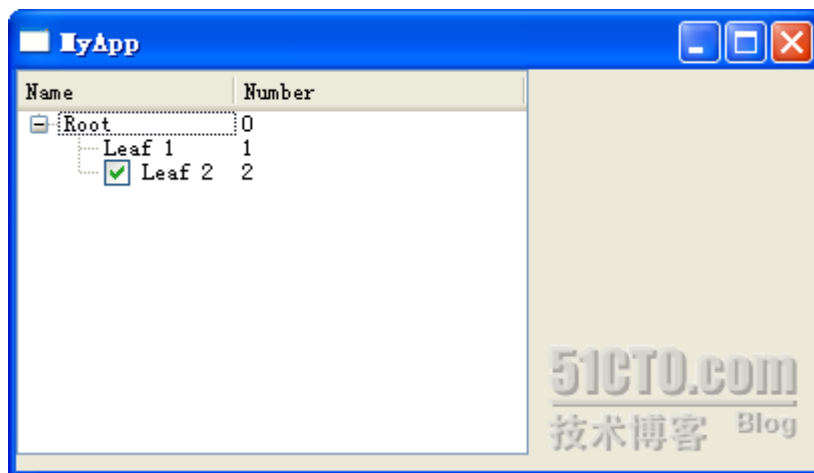
TreeWidget::TreeWidget()
{
    tree = new QTreeWidgetItem(this);
    tree->setColumnCount(2);
    QStringList headers;
    headers << "Name" << "Number";
    tree->setHeaderLabels(headers);
```

```

QStringList rootTextList;
rootTextList << "Root" << "0";
QTreeWidgetItem *root = new QTreeWidgetItem(tree, rootTextList);
QStringList leafTextList;
leafTextList << "Leaf 1" << "1";
QTreeWidgetItem *leaf = new QTreeWidgetItem(root, leafTextList);
root->addChild(leaf);
QStringList leaf2TextList;
leaf2TextList << "Leaf 2" << "2";
QTreeWidgetItem *leaf2 = new QTreeWidgetItem(root, leaf2TextList);
leaf2->setCheckState(0, Qt::Checked);
root->addChild(leaf2);
QList<QTreeWidgetItem *> rootList;
rootList << root;
tree->insertTopLevelItems(0, rootList);
}

```

我们把 `columnCount` 设为 2，然后传入的 `QStringList` 对应的有 2 个元素。这样再来运行一下：

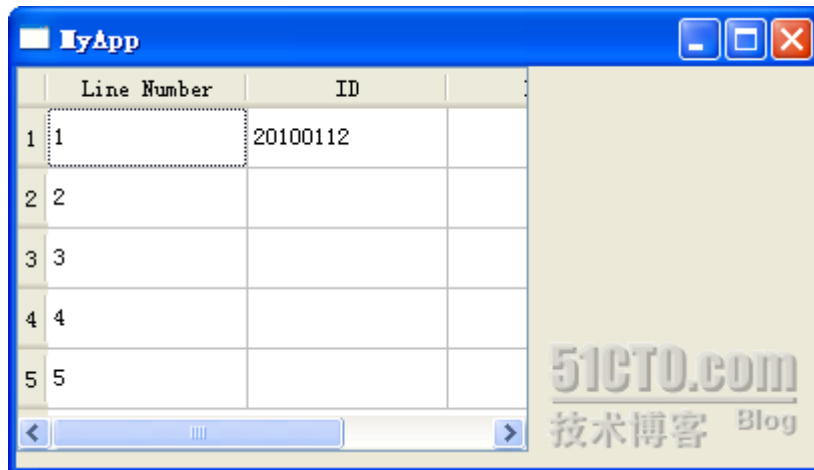


原来这个 `columnCount` 就是用于在列表中显示树的！这样，你就可以很容易的将树和列表结合在一起，从而实现类似 Windows 资源管理器的界面。当然，如果你不需要显示这个 `header`，可以调用 `setHeaderHidden()` 函数将这个功能隐藏掉。

Qt 学习之路(41): QTableWidget

是接触到了最简单的 `model-view` 的封装类，更复杂和强大的 `model-view` 类型的应用还没有见识到呢！

`QTableWidget` 用起来也很方便，并不比前面的两个复杂到哪里去。我们运行的结果是这样的：



下面是代码:

tablewidget.h

```
#ifndef TABLEWIDGET_H
#define TABLEWIDGET_H

#include <QtGui>

class TableWidget : public QWidget
{
public:
    TableWidget();

private:
    QTableWidgetItem *table;
};

#endif // TABLEWIDGET_H
```

tablewidget.cpp

```
#include "tablewidget.h"

TableWidget::TableWidget()
{
    table = new QTableWidgetItem(this);
    table->setColumnCount(3);
    table->setRowCount(5);
    QStringList headers;
    headers << "Line Number" << "ID" << "Name" << "Age" << "Sex";
    table->setHorizontalHeaderLabels(headers);
    table->setItem(0, 0, new QTableWidgetItem(QString("1")));
}
```

```

    table->setItem(1, 0, new QTableWidgetItem(QString("2")));
    table->setItem(2, 0, new QTableWidgetItem(QString("3")));
    table->setItem(3, 0, new QTableWidgetItem(QString("4")));
    table->setItem(4, 0, new QTableWidgetItem(QString("5")));
    table->setItem(0, 1, new QTableWidgetItem(tr("20100112")));
}

```

代码看起来很清楚。首先创建了 `QTableWidget` 对象，然后设置列数和行数。接下来使用一个 `QStringList`，把每一列的标题设置了一下。然后调用 `addItem()` 函数。这个函数前两个参数分别是行 `row` 和列 `col`，然后第三个参数构建一个 `QTableWidgetItem` 对象，这样，Qt 就会把这个对象放在第 `row` 行第 `col` 列的单元格里面。注意，这里的行和列都是从 0 开始的。

Qt 学习之路(42): QStringListModel

今天开始我们要看看 Qt 的 `model-view` 类了。正如前面说的那样，之前三节的 `item class` 类只是 Qt 为了方便我们使用而封装了一些操作。比起真正的 `model-view` 类来，那些类更易于使用，但是功能也会更简单，并且缺少实时性的支持，比如我们并不方便实现插入、删除等一些常见操作。而现在我们要说的 `model-view` 类使用起来可能会复杂一些，但是功能强大，并且在 `model` 更新时会自动更新 `view`，而 `model` 多是一些数据集合，因此比较便于操作。

`model-view` 类中，`view` 大致有三种：`list`、`tree` 和 `table`，但是 `model` 千奇百怪，不同的业务，甚至同样的业务不同的建模都会有不同的 `model`。为了方便使用，Qt 提供了一些预定义好的 `model` 供我们使用。`QStringListModel` 是最简单的一种。

顾名思义，`QStringListModel` 就是封装了 `QStringList` 的 `model`。`QStringList` 是一种很常用的数据类型，它实际上是一个字符串列表。我们可以想象，对于一个 `list` 来说，如果提供一个字符串列表形式的数据，就应该能够把这个数据展示出来。因为二者是一致的：`QStringList` 是线性的，而 `list` 也是线性的。所以，`QStringListModel` 很多时候都会作为 `QListView` 的 `model`。

下面我们来看怎么使用它们。比起前面的 `QListWidget`，这里要使用两个类：`QStringListModel` 和 `QListView`，并且还有一些辅助类。不过你可以看到，即便这样复杂的工作，我们的代码也不会很多的：

mylistview.h

```

#ifndef MYLISTVIEW_H
#define MYLISTVIEW_H

#include <QtGui>

class MyListView : public QWidget
{
    Q_OBJECT
public:
    MyListView();

private:

```

```

    QStringListModel *model;
    QListView *listView;

private slots:
    void insertData();
    void deleteData();
    void showData();
};

#endif // MYLISTVIEW_H

```

mylistview.cpp

```

#include "mylistview.h"

MyListView::MyListView()
{
    model = new QStringListModel(this);
    QStringList data;
    data << "Letter A" << "Letter B" << "Letter C";
    model->setStringList(data);
    listView = new QListView(this);
    listView->setModel(model);
    QHBoxLayout *btnLayout = new QHBoxLayout;
    QPushButton *insertBtn = new QPushButton(tr("insert"), this);
    QPushButton *delBtn = new QPushButton(tr("Delete"), this);
    QPushButton *showBtn = new QPushButton(tr("Show"), this);
    btnLayout->addWidget(insertBtn);
    btnLayout->addWidget(delBtn);
    btnLayout->addWidget(showBtn);
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->addWidget(listView);
    mainLayout->addLayout(btnLayout);
    this->setLayout(mainLayout);

    connect(insertBtn, SIGNAL(clicked()), this, SLOT(insertData()));
    connect(delBtn, SIGNAL(clicked()), this, SLOT(deleteData()));
    connect(showBtn, SIGNAL(clicked()), this, SLOT(showData()));
}

void MyListView::insertData()
{
    bool isOK;
    QString text = QInputDialog::getText(NULL, "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);
    }
}

void MyListView::deleteData()
{
    if(model->rowCount() > 1) {
        model->removeRows(listView->currentIndex().row(), 1);
    }
}

void MyListView::showData()
{
    QStringList data = model->stringList();
    QString str;
    foreach(QString s, data) {
        str += s + "\n";
    }

    QMessageBox::information(this, "Data", str);
}

```

来看看我们的代码吧。

首先我们创建一个 `QStringListModel` 的对象。然后创建一个 `QStringList` 对象，并且把这个对象设置为 `model` 的数据。此时，这个 `model` 已经拥有数据了。然后，我们创建一个 `QListView` 的对象，并把 `model` 设置为它的 `model`。后面是三个按钮的创建以及信号槽的连接，这里就不再赘述。

先来运行一下看看结果吧！



我们只是把 `QStringListModel` 设置为 `QListView` 的 `model`，`QListView` 就已经可以把 `model` 里面的数据展示出来了。下面我们看看增、删、改的操作。

先来看增加数据的操作。这部分是在代码中的 `insertData()` 函数实现的。先把那个函数拿出来看看：

```
void MyListView::insertData()
{
    bool isOK;
    QString text = QInputDialog::getText(NULL, "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);
    }
}
```

我们使用 `QInputDialog::getText()` 函数要求用户输入数据。这部分在前面讲过，这里也不再赘述。如果用户点击了 `OK` 按钮，首先，我们使用 `listView->currentIndex()` 函数，获取 `QListView` 当前行。注意，这个函数的返回值是一个 `QModelIndex` 类型。这个类我们以后再说，

只要知道这个类保存了三个重要的数据：行、列以及属于哪一个 **model**。我们调用其 **row()** 函数获得行，这个返回值是一个 **int**，也就是第几行。然后 **model** 插入一行。**insertRows()** 函数签名如下：

```
bool insertRows(int row, int count, const QModelIndex &parent = QModelIndex());
```

这个函数原本是 **QAbstractListModel** 类的函数，而 **QStringListModel** 把它覆盖了。所以我们会发现它还需要有一个列的参数。我们调用 **insertRows(row, 1);**，所谓 **1** 就是指 **1** 列（因为我们使用 **list** 是一维的，列数永远是 **1**），而前面又把 **row** 保存成当前行，因此，这行语句实际上是在当前行插入一行。然后我们使用 **model** 的 **index()** 函数获取当前行的 **QModelIndex** 对象，使用 **setData()** 函数把我们用 **QInputDialog** 接受的数据插入。这里其实是一个冗余的操作，因为用 **currentIndex()** 函数已经获取当前行了。这么写仅仅是为了展示如何使用这个函数。不过，你知道了 **insertRow()** 函数，就可以很容易的做出插入空白行的效果了。然后我们把当前行设为新插入的一行，并调用 **edit()** 函数，这个函数使得这一行可以被编辑。就这样，我们向 **model** 插入了数据。

然后来看删除数据的操作：

```
void MyListView::deleteData()
{
    if(model->rowCount() > 1) {
        model->removeRows(listView->currentIndex().row(), 1);
    }
}
```

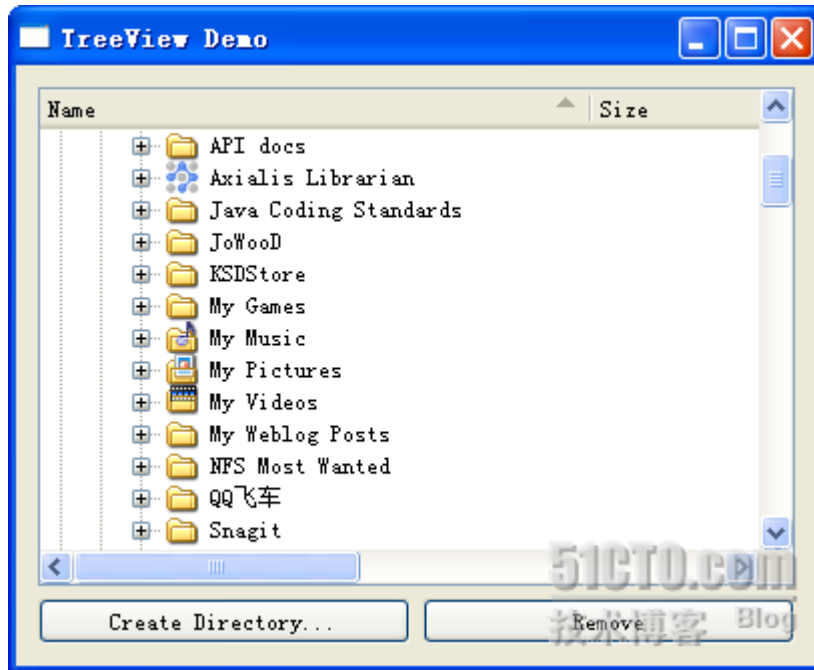
使用 **model** 的 **removeRows()** 函数可以轻松的完成这个功能。这个函数同前面所说的 **insertRows()** 很类似，就不再多说了。需要注意的是，我们用 **rowCount()** 函数判断了一下，要求最终始终保留 **1** 行。这是因为如果你把数据全部删除，你就不能再插入数据了，因为那时候按照我们所写的插入逻辑就不对了。所以，前面所说的插入操作实际上还需要再详细考虑。

最后那个 **showData()** 仅仅为了查看 **model** 的数据，没有什么要说的东西。你可以在 **insert** 或者 **remove** 完成后查看一下 **model** 里面的数据是不是真的被修改了。

关于 **QStringListModel** 就说这么多。你可以看到，我们的几乎所有操作都是针对 **model** 的，也就是说，我们直接针对的是数据，而 **model** 侦测到数据发生了变化，会立刻通知 **view** 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 **view** 的同步等操作。这也是 **model-view** 模型的一个便捷之处。

Qt 学习之路(43): QDirModel

今天我们来看一个很有用的 **model**： **QDirModel**。这个 **model** 允许我们在 **view** 中显示操作系统的目录结构。这次让我们先来看看运行结果：



这个界面很熟悉吧？不过这可不是由 `QFileDialog` 打开的哦，这是我们自己实现的。而提供这种实现支持的，就是 `QDirModel` 和 `QTreeView`。我们来看一下代码。

mytreeview.h

```
#ifndef MYLISTVIEW_H
#define MYLISTVIEW_H

#include <QtGui>

class MyTreeView : public QWidget
{
    Q_OBJECT
public:
    MyTreeView();

private:
    QDirModel *model;
    QTreeView *treeView;

private slots:
    void mkdir();
    void rm();
};

#endif // MYLISTVIEW_H
```

mytreeview.cpp

```
#include "mylistview.h"

MyTreeView::MyTreeView()
{
    model = new QDirModel;
    model->setReadOnly(false);
    model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    treeView = new QTreeView;
    treeView->setModel(model);
    treeView->header()->setStretchLastSection(true);
    treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
    treeView->header()->setSortIndicatorShown(true);
    treeView->header()->setClickable(true);

    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);

    QHBoxLayout *btnLayout = new QHBoxLayout;
    QPushButton *createBtn = new QPushButton(tr("Create Directory..."));
    QPushButton *delBtn = new QPushButton(tr("Remove"));
    btnLayout->addWidget(createBtn);
    btnLayout->addWidget(delBtn);
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->addWidget(treeView);
    mainLayout->addLayout(btnLayout);
    this->setLayout(mainLayout);

    connect(createBtn, SIGNAL(clicked()), this, SLOT(mkdir()));
    connect(delBtn, SIGNAL(clicked()), this, SLOT(rm()));
}

void MyTreeView::mkdir()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid()) {
        return;
    }
    QString dirName = QInputDialog::getText(this,
tr("Create Director
y"),
```

```

tr("Directory name
"));
    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid()) {
            QMessageBox::information(this,
                                    tr("Create Directory"),
                                    tr("Failed to create the direc
tory"));
        }
    }
}

void MyTreeView::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)));
    }
}
}

```

构造函数中，首先我们创建了 `QDirModel` 的一个对象，并且设置 `ReadOnly` 为 `false`，也就是说我们可以对其进行修改。而下一个 `setSorting()` 函数是对其进行排序，排序的依据也很清楚：文件夹优先(`QDir::DirsFirst`)，忽略大小写(`QDir::IgnoreCase`)，而且是根据名字排序(`QDir::Name`)。更多的规则组合可以参见 [API 文档](#)了。

然后我们创建一个 `QTreeView` 实例，并且把 `model` 设置为刚刚的 `QDirModel` 实例。然后我们开始设置 `QTreeView` 的相关属性。首先把 `stretchLastSection` 设置为 `true`。如果把这个属性设置为 `true`，就是说，当 `QTreeView` 的宽度大于所有列宽之和时，最后一列的宽度自动扩展以充满最后的边界；否则就让最后一列的宽度保持原始大小。第二个 `setSortIndicator()` 函数是设置哪一列进行排序。由于我们前面设置了 `model` 是按照名字排序，所以我们这个传递的第一个参数是 `0`，也就是第 `1` 列。`setSortIndicatorShown()` 函数设置显示列头上面的排序小箭头。`setClickable(true)` 则允许鼠标点击列头。这样，我们的 `QTreeView` 就设置完毕了。最

后，我们通过 `QDir::currentPath()` 获取当前 `exe` 文件运行时路径，并把这个路径当成程序启动时显示的路径。`expand()` 函数即展开这一路径；`scrollTo()` 函数是把视图的视口滚动到这个路径的位置；`resizeColumnToContents()` 是要求把列头适应内容的宽度，也就是不产生...符号。这样，我们就通过一系列的参数设置好了 `QTreeView`，让它能够为我们展示目录结构。

至于后面的两个 `slot`，其实并不能理解。第一个 `mkdir()` 函数就是创建一个文件夹。

```
void MyTreeView::mkdir()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid()) {
        return;
    }
    QString dirName = QInputDialog::getText(this,
                                              tr("Create Directory"),
                                              tr("Directory name"),
                                              QLineEdit::Normal,
                                              QString(),
                                              nullptr);
    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid()) {
            QMessageBox::information(this,
                                     tr("Create Directory"),
                                     tr("Failed to create the directory"));
        }
    }
}
```

正如它的代码所示，首先获取选择的目录。后面这个 `isValid()` 的判断很重要，因为默认情况下是没有目录被选择的，此时这个路径是非法的，为了避免程序出现异常，必须要有这一步判断。然后会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。这时候你就可以到硬盘上的实际位置看看啦！

删除目录的代码也很类似：

```
void MyTreeView::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)));
        }
    }
}

```

同样需要实现检测路径是否合法。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。

注意，`QDirModel` 在最新版 Qt 中已经不建议使用了。文档中说使用 `QFileSystemModel` 代替。由于这两者的函数几乎一样，所以就没有对代码进行修改。与 `QDirModel` 不同的是，`QFileSystemModel` 会启动自己的线程进行文件夹的扫描，因此不会发生因扫描文件夹而导致的主线程阻塞的现象。另外，无论 `QDirModel` 还是 `QFileSystemModel` 都会对 `model` 结果进行缓存，如果你要立即刷新结果，前者提供了 `refresh()` 函数，而后者会通知 `QFileSystemWatcher` 类。

Qt 学习之路(44): QSortFilterProxyModel

Qt 为我们预定义了很多 `model`，前面已经说过了 `QStringListModel`、`QDirModel` (也算是 Qt 推荐使用的 `QFileSystemModel` 吧，这个在上一章最后重新加上了一段话，没有注意的朋友去看看哦)。今天我们要说的这个 `QSortFilterProxyModel` 并不能单独使用，看它的名字就会知道，它只是一个“代理”，真正的数据需要另外的一个 `model` 提供，并且它是用来排序和过滤的。所谓过滤，也就是说按照你输入的内容进行数据的筛选，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，因而功能强大。

我们从代码开始看起：

sortview.h

```

#ifndef SORTVIEW_H
#define SORTVIEW_H

#include <QtGui>

class SortView : public QWidget
{
    Q_OBJECT
public:
    SortView();

private:
    QListView *view;

```

```

    QStringListModel *model;
    QSortFilterProxyModel *modelProxy;
    QComboBox *syntaxBox;

private slots:
    void filterChanged(QString text);
};

#endif // SORTVIEW_H

```

sortview.cpp

```

#include "sortview.h"

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)));
    }

    void SortView::filterChanged(QString text)
    {
        QRegExp::PatternSyntax syntax = QRegExp::PatternSyntax(
            syntaxBox->itemData(syntaxBox->currentIndex()).toInt());
        QRegExp regExp(text, Qt::CaseInsensitive, syntax);
        modelProxy->setFilterRegExp(regExp);
    }

```

至于 `main()` 函数的内容，由于和前面的代码几乎是一样的，这里就不再贴出来了。我们使用的是 `QColor::colorNames()` 函数提供的数据。这个函数返回值是一个 `QStringList` 类型的变量，可以给出预定义的颜色名字。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。那么我们需要过滤哪一列呢？虽然 `QStringListModel` 只有一列，但是我們也需要使用 `setFilterKeyColumn()` 函数设置一下，以便让这个 `proxy` 知道要过滤的是第 0 列。最后重要的一点是，`QListView` 的 `model` 必须设置为 `QSortFilterProxyModel`，否则是看不到效果的。

下面的 `QLineEdit` 提供过滤数据的输入，这个没什么好说的。后面的 `QComboBox` 列出了三项：

```

syntaxBox->addItem(tr("Regular expression"), QRegExp::RegExp);
syntaxBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
syntaxBox->addItem(tr("Fixed string"), QRegExp::FixedString);

```

这是正则表达式的类型。正则表达式有一套通用的语法，但是对于不同的环境，正则表达式的规则可能是不一样的。第一个 `QRegExp::RegExp` 提供了最一般的正则表达式语法，不过这个语法不支持贪婪限定符。这也是 `Qt` 默认的规则。如果你需要使用贪婪限定符，需要使用 `QRegExp::RegExp2`，根据文档描述，这个将会是 `Qt5` 的默认规则。第二个是 `Unix` 下 `shell` 很常见的一种规则。第三个即固定表达式，也就是说基本上不使用正则表达式的。

我们使用 `connect()` 函数，将 `QLineEdit` 的 `textChanged()` 信号同 `slot` 连接起来。其中我们的 `slot` 函数如下所示：

```

    void SortView::filterChanged(QString text)
    {
        QRegExp::PatternSyntax syntax = QRegExp::PatternSyntax(
            syntaxBox->itemData(syntaxBox->currentIndex()).toInt());
        QRegExp regExp(text, Qt::CaseInsensitive, syntax);
    }

```

```
modelProxy->setFilterRegExp(regExp);  
}
```

第一步，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 `proxy` 的过滤器的表达式。好了，就这样运行一下看看效果吧！



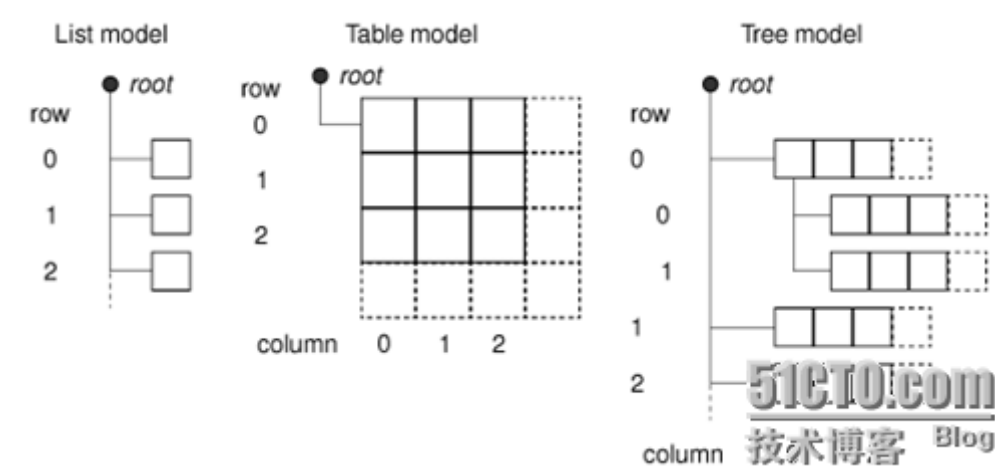
Qt 学习之路(45): 自定义 model 之一

前面我们说了 Qt 提供的几个预定义 model。但是，面对变化万千的需求，那几个 model 是远远不能满足我们的需要的。另外，对于 Qt 这种框架来说，model 的选择首先要能满足绝大多数功能的需要，这就是说，可能这个 model 中的某些功能你永远也不会用到，但是还要带着它，这样做的后果就是效率不会很高。所以，我们还必须要能够自定义 model。

在我们真正的完成自定义 model 之前，先来看看在 Qt 的 model-view 架构中的几个关键的概念。一个 model 中的每个数据元素都有一个 model 索引。这个索引指明这个数据位于 model 的位置，比如行、列等。这就是前面我们曾经说到过的 `QModelIndex`。每个数据元素还要有一组属性值，称为角色(roles)。这个属性值并不是数据的内容，而是它的属性，比如说，这个数据是用来展示数据的，还是用于显示列头的？因此，这组属性值实际上是 Qt 的一个 enum 定义的，比较常见的有 `Qt::DisplayRole` 和 `Qt::EditRole`，另外还有 `Qt::ToolTipRole`，`Qt::StatusTipRole`，和 `Qt::WhatsThisRole` 等。并且，还有一些属性是用来描述基本的展现属性的，比如 `Qt::FontRole`，`Qt::TextAlignmentRole`，`Qt::TextColorRole`，`Qt::BackgroundColorRole` 等。

对于 list model 而言，要定位其中的一个数据只需要有一个行号就可以了，这个行号可以通过 `QModelIndex::row()` 函数进行访问；对于 table model 而言，这种定位需要有两个值：行号和列号，这两个值可以通过 `QModelIndex::row()` 和 `QModelIndex::column()` 这两个函数访问到。另外，对于 tree model 而言，用于定位的可以是这个元素的父节点。实际上，不仅仅

是 **tree model**，并且 **list model** 和 **table model** 的元素也都有自己的父节点，只不过对于 **list model** 和 **table model**，它们元素的父节点都是相同的，并且指向一个非法的 **QModelIndex**。对于所有的 **model**，这个父节点都可以通过 **QModelIndex::parent()** 函数访问到。这就是说，每个 **model** 的项都有自己的角色数据，0 个、1 个或多个子节点。既然每个元素都有自己的子元素，那么它们就可以通过递归的算法进行遍历，就像数据结构中树的遍历一样。关于父节点的描述，请看下面这张图(出自 **C++ GUI Programming with Qt4, 2nd Edition**):



下面我们通过一个简单的例子来看看如何实现自定义 **model**。这个例子来自 **C++ GUI Programming with Qt4, 2nd Edition**。首先描述一下需求。这里我们要实现的是一个类似于货币汇率表的 **table**。或许你会想，这是一个很简单的实现，直接用 **QTableWidget** 不就可以了么？的确，如果直接使用 **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(在这里，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)。那么，这张表要有 $100 \times 100 = 10000$ 个数据项。现在要求我们减少存储空间。于是我们想，如果我们的数据不是显示的数据，而是这种货币相对于美元的汇率，那么，其他货币的汇率都可以根据这个汇率计算出来了。比如说，我存储的是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。我没有必要存储 10000 个数据项，只要存储 100 个就够了。于是，我们要自己实现一个 **model**。

CurrencyModel 就是这样一个 **model**。它底层的数据使用一个 **QMap<QString, double>** 类型的数据，作为 **key** 的 **QString** 是货币名字，作为 **value** 的 **double** 是这种货币对美元的汇率。然后我们来看代码：

```
.h
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) co
```

```

nst;
private:
    QString currencyAt(int offset) const;
    QMap<QString, double> currencyMap;
};

```

.cpp

```

CurrencyModel::CurrencyModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}

int CurrencyModel::rowCount(const QModelIndex & parent) const
{
    return currencyMap.count();
}

int CurrencyModel::columnCount(const QModelIndex & parent) const
{
    return currencyMap.count();
}

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) / currencyM
ap.value(rowCurrency);
        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}

QVariant CurrencyModel::headerData(int section, Qt::Orientation orientation, i
nt role) const
{
}

```

```

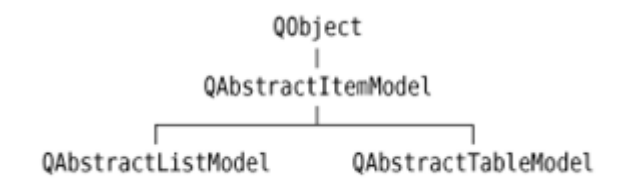
        if (role != Qt::DisplayRole)
            return QVariant();
        return currencyAt(section);
    }

    void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
    {
        currencyMap = map;
        reset();
    }

    QString CurrencyModel::currencyAt(int offset) const
    {
        return (currencyMap.begin() + offset).key();
    }

```

我们选择了继承 `QAbstractTableModel`。虽然是自定义 `model`，但各种 `model` 之间也会有很多共性。`Qt` 提供了一系列的抽象类供我们继承，以便让我们只需要覆盖掉几个函数就可以轻松地定义出我们自己的 `model`。`Qt` 提供了 `QAbstractListModel` 和 `QAbstractTableModel` 两类，前者是一维数据 `model`，后者是二维数据 `model`。如果你的数据很复杂，那么可以直接继承 `QAbstractItemModel`。这三个类之间的关系可以表述如下：（出自 `C++ GUI Programming with Qt4, 2nd Edition`）：



构造函数中没有添加任何代码，只要调用父类的构造函数就可以了。然后我们重写了 `rowCount()` 和 `columnCount()` 这两个函数，用于返回 `model` 的行数和列数。由于我们使用一维的 `map` 记录数据，因此这里的行和列都是 `map` 的大小。然后我们看最复杂的 `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) / currencyM
ap.value(rowCurrency);
|         return QString("%1").arg(amount, 0, 'f', 4);
|     }
|     return QVariant();
| }

```

`data()`函数返回单元格的数据。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`。至此，我们还是第一次见到这个类型。这个类型相当于是 `Java` 里面的 `Object`，它把绝大多数 `Qt` 提供的数据类型都封装起来，起到一个数据类型“擦除”的作用。比如我们的 `table` 单元格可以是 `string`，也可以是 `int`，也可以是一个颜色值，那么这么多类型怎么返回呢？于是，`Qt` 提供了这个 `QVariant` 类型，你可以把这很多类型都存放进去，到需要使用的时候使用一系列的 `to` 函数取出来即可。比如你把 `int` 包装成一个 `QVariant`，使用的时候要用 `QVariant::toInt()` 重新取出来。这里需要注意的是，`QVariant` 类型的放入和取出必须是相对应的，你放入一个 `int` 就必须按 `int` 取出，不能用 `toString()`，`Qt` 不会帮你自动转换。或许你会问，`Qt` 不是提供了一个 `QObject` 类型吗？为什么不像 `Java` 一样都用 `Object` 呢？关于这一点豆子也没有官方文档，不过可以猜测一下。和 `Java` 不同，`C++` 的面向对象体系不是单根的，`C++` 对象并不是都继承于某一个类，因此，如果你要实现一个这种功能的类，做到“类型擦除”，就必须用一个类包含所有的数据类型。就相当于设计一个能放进所有形状盒子，你才能把各种各样的形状放进去。这样的话这个类就会变得异常庞大。对于 `Qt`，`QObject` 类是大多数类继承的类，理应越小越好，因此就把这个功能抽取出来，形成了一个新类。这也只是豆子的猜测，大家不必往心里去:-)

好了，下面看这个类的内容。首先判断传入的 `index` 是不是合法，如果不合法直接 `return` 一个空白的 `QVariant`。然后如果 `role` 是 `Qt::TextAlignmentRole`，也就是文本的对象方式，那么就返回 `int(Qt::AlignRight | Qt::AlignVCenter)`；否则，`role` 如果是 `Qt::DisplayRole`，就按照我们前面所说的逻辑进行计算，然后按照字符串返回。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` 里面是 `int`，而 `else` 里面是 `QString`，这就是 `QVariant` 的作用了，也正是“类型擦除”的意思。

剩下的三个函数就很简单了：`headerData()`返回列名或者行名；`setCurrencyMap()`用于设置底层的数据源；`currencyAt()`返回偏移量为 `offset` 的键值。

至于调用就很简单了：

```

| CurrencyTable::CurrencyTable()
| {
|     QMap<QString, double> data;
|     data["NOK"] = 1.0000;
|     data["NZD"] = 0.2254;
|     data["SEK"] = 1.1991;
|     data["SGD"] = 0.2592;
|     data["USD"] = 0.1534;
|
|     CurrencyModel *model = new CurrencyModel;

```

```

model->setCurrencyMap(data);

QTableView *view = new QTableView(this);
view->setModel(model);
view->resize(400, 300);
}

```

好了，最后让我们来看一下最终结果吧！



	NOK	NZD	SEK	SGD
NOK	1.0000	0.2254	1.1991	0.
NZD	4.4366	1.0000	5.3199	1.
SEK	0.8340	0.1880	1.0000	0.
SGD	3.8580	0.8696	4.6262	1.
USD	6.5189	1.4694	7.8168	1.

注意，这一章中的代码不是完整的代码，缺少 **view** 的头文件，不过这只是一个空白的文件。你也可以直接把 **view** 的代码放到 **main()** 函数里面运行。

Qt 学习之路(46): 自定义 model 之二

前面的例子已经比较清楚的给出了自定义 **model** 的方法，就是要覆盖我们所需要的那几个函数就可以了。但是，前面的例子仅仅是简单的展示数据，也就是说数据时只读的。那么，如何能做到读写数据呢？那就要来看进来的例子了。这个例子也是来自 **C++GUI Programming with Qt 4, 2nd Edition** 这本书的。

还是先来看代码吧：

citymodel.h

```

class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:

```

```

CityModel(QObject *parent = 0);

void setCities(const QStringList &cityNames);
int rowCount(const QModelIndex &parent) const;
int columnCount(const QModelIndex &parent) const;
QVariant data(const QModelIndex &index, int role) const;
bool setData(const QModelIndex &index, const QVariant &value, int rol
e);
QVariant headerData(int section, Qt::Orientation orientation, int role) co
nst;
Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;

    QStringList cities;
    QVector<int> distances;
};

```

citymodel.cpp

```

CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}

int CityModel::rowCount(const QModelIndex & parent) const
{
    return cities.count();
}

int CityModel::columnCount(const QModelIndex & parent) const
{
    return cities.count();
}

QVariant CityModel::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) {
        if (index.row() == index.column()) {
            return 0;
        }
        int offset = offsetOf(index.row(), index.column());
        return distances[offset];
    }
    return QVariant();
}

QVariant CityModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if (role == Qt::DisplayRole) {
        return cities[section];
    }
    return QVariant();
}

bool CityModel::setData(const QModelIndex &index, const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column() && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(), index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}

Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column()) {
        flags |= Qt::ItemIsEditable;
    }
    return flags;
}

```

```

void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}

int CityModel::offsetOf(int row, int column) const
{
    if (row < column) {
        qSwap(row, column);
    }
    return (row * (row - 1) / 2) + column;
}

```

代码很长，但实际上和前面我们的那个例子非常相似。这个 **model** 也是用于 **table** 的，因此还是继承了 **QAbstractTableModel**。**CityModel** 内部有两个数据源：一个 **QStringList** 类型的对象，一个 **QVector<int>** 类型的对象。前者用于保存城市的名字，需要用户显示的给出；后者是 **model** 内部维护的一个存放 **int** 的向量。这个 **CityModel** 就是要在 **table** 中显示两个城市之间的距离。同前面的例子一样，如果我们要把所有数据都保存下来，显然会造成数据的冗余：城市 **A** 到城市 **B** 的距离同城市 **B** 到城市 **A** 的距离是一样的！因此我们还是自定义一个 **model**。同样这个 **CityModel** 有个简单的空构造函数，**rowCount()** 和 **columnCount()** 函数也是返回 **list** 的长度。**data()** 函数根据 **role** 的不同返回不同的值。由于在 **table** 中坐标是由 **row** 和 **column** 给出的，因此需要有一个二维坐标到一维坐标的转换，这就是 **offsetOf()** 函数的作用。我们把主要精力放在 **setData()** 函数上面。

```

bool CityModel::setData(const QModelIndex &index, const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column() && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(), index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}

```

这个函数在用户编辑数据时会自动调用。也就是说，这时我们的数据已经不是只读的了。函数开始是一个长长的判断：`index` 要是合法的；`index` 的 `row` 和 `column` 不相等，也就是说两个城市是不同的；数据想的 `role` 是 `Qt::EditRole`。如果满足了这三个条件，才会执行下面的操作。首先，由 `row` 和 `column` 坐标定位到表中的数据项在 `vector` 中的位置。然后用户新修改的数据被作为参数 `value` 传入，所以我们要把这个参数赋值给 `distances`。`createIndex()` 函数根据 `column` 和 `row` 值生成一个 `QModelIndex` 对象。请注意这里的顺序：`row` 和 `column` 是颠倒的！这就把坐标为(`row`, `column`)的点关于主对角线对称的那个点(`column`, `row`)的 `index` 找到了。还记得我们的需求吗？当我们修改了一个数据时，对应的数据也要被修改，就是这个功能的实现。我们需要 `emit dataChanged()` 信号，这个信号接收两个参数：一个是被修改的数据的左上角的坐标，一个是被修改的数据的右下角的坐标。为什么会有两个坐标呢？因此我们修改的数据不一定只是一个。像这里，我们只修改了一个数据，因此这两个值是相同的。数据更新了，我们用这个信号通知 `view` 刷新，这样就可以显示新的数据了。最后，如果函数数据修改成功就返回 `true`，否则返回 `false`。

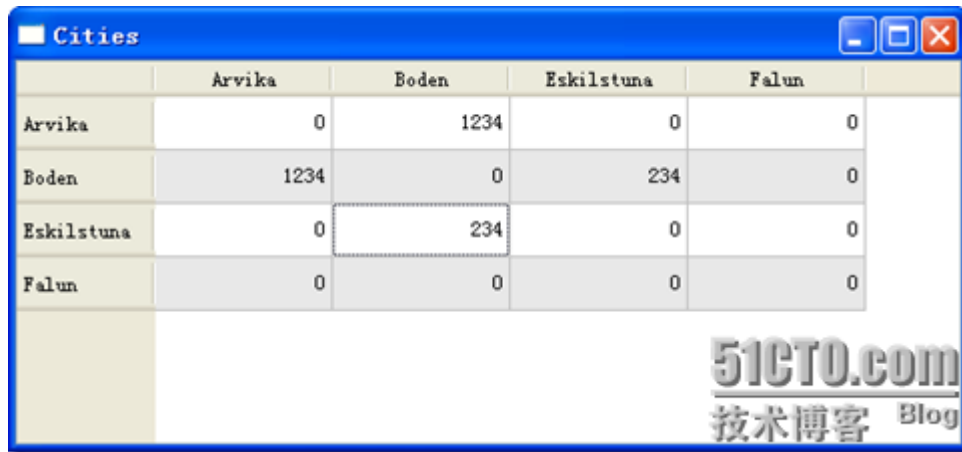
最后，我们在 `main()` 函数中显示出来这个 `model`：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QStringList cities;
    cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun";

    CityModel cityModel;
    cityModel.setCities(cities);

    QTableView tableView;
    tableView.setModel(&cityModel);
    tableView.setAlternatingRowColors(true);
    tableView.setWindowTitle(QObject::tr("Cities"));
    tableView.show();
    return a.exec();
}
```

这样，我们就把这个 `model` 做完了。最后来看看效果吧！



	Arvika	Boden	Eskilstuna	Falun
Arvika	0	1234	0	0
Boden	1234	0	234	0
Eskilstuna	0	234	0	0
Falun	0	0	0	0

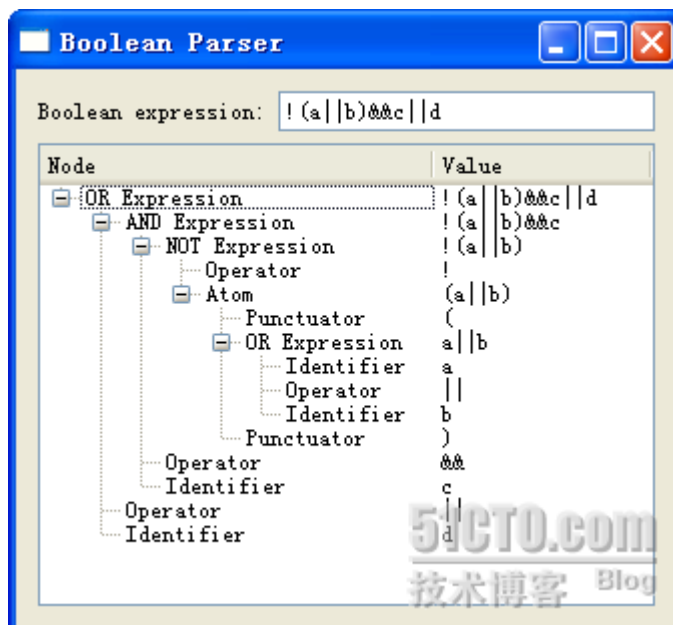
Qt 学习之路(47): 自定义 Model 之三

今天来说的是自定义 model 中最复杂的例子。这个例子同样也是出自 C++ GUI Programming with Qt 4, 2nd Edition 这本书。

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，类似的分析可以比较方便的进行表达式化简、求值等一系列的计算。同样，这个技术也可以很方便的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；
- **BooleaModel**: 布尔表达式的 model，实际上是一个 tree model，用于将布尔表达式表示成一棵树；
- **BooleanParser**: 将布尔表达式生成分析树的分析器；
- **BooleanWindow**: 输入布尔表达式并进行分析，展现成一棵树。

这个例子可能是目前为止最复杂的一个了，所以先来看看最终的结果，以便让我们心中有数：



先来看这张图片，我们输入的布尔表达式是 `!(a||b)&&c||d`，在下面的 **Node** 栏中，用树的形式将这个表达式分析了出来。如果你熟悉编译原理，这个过程很像词法分析的过程：将一个语句分析称一个一个独立的词素。

我们从最底层的 **Node** 类开始看起，一步步构造这个程序。

Node.h

```
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.cpp

```
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** 里面还有一个 **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** 是一个 **QList<Node *>** 类型的列表，保存这个 **node** 的子节点。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**。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，将所有子元素的指针清为 **0**。

在构造完子节点之后，我们开始构造 **model**：

booleanmodel.h

```
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.cpp

```
BooleanModel::BooleanModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}

BooleanModel::~~BooleanModel()
{
    delete rootNode;
```

```

}

void BooleanModel::setRootNode(Node *node)
{
    delete rootNode;
    rootNode = node;
    reset();
}

QModelIndex BooleanModel::index(int row, int column,
                                const QModelIndex &parent) const
{
    if (!rootNode || row < 0 || column < 0)
        return QModelIndex();
    Node *parentNode = nodeFromIndex(parent);
    Node *childNode = parentNode->children.value(row);
    if (!childNode)
        return QModelIndex();
    return createIndex(row, column, childNode);
}

Node *BooleanModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return rootNode;
    }
}

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;
}

```

```

}

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:

```



```

| {
|     if (!rootNode || row < 0 || column < 0)
|         return QModelIndex();
|     Node *parentNode = nodeFromIndex(parent);
|     Node *childNode = parentNode->children.value(row);
|     if (!childNode)
|         return QModelIndex();
|     return createIndex(row, column, childNode);
| }

```

如果 `rootNode` 或者 `row` 或者 `column` 非法，返回一个非法的 `QModelIndex`。然后使用 `nodeFromIndex()` 函数取得索引为 `parent` 的节点，然后我们使用 `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。最后，当是一个有效值时，由 `createIndex()` 函数返回有效地 `QModelIndex` 对象。

对于具有层次结构的 `model` 来说，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。因此，`createIndex()` 中第三个参数就是这个内部的指针。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 `node`。

后面的 `rowCount()` 和 `columnCount()` 这两个函数比较简单，就是要获得 `model` 的行和列的值。由于我们的 `model` 定义成 2 列，所以在 `columnCount()` 函数中始终返回 2。

`parent()` 函数要返回子节点的父节点的索引，我们要从子节点开始寻找，直到找到父节点的父节点，这样就能定位到父节点，从而得到子节点的位置。而 `data()` 函数要返回每个单元格的返回值，经过前面两个例子，我想这个函数已经不会有很大的困难了的。`headerData()` 函数返回列头的名字，同前面一样，这里就不再赘述了。

前面的代码很长，`BooleanWindow` 部分就很简单了。就是把整个 `view` 和 `model` 组合起来。另外的一个 `BooleanParser` 类没有什么 GUI 方面的代码，是纯粹的算法问题。如果我看得没错的话，这里应该使用的是编译原理里面的递归下降词法分析，有兴趣的朋友可以到网上查一下相关的资料。我想在以后的《自己动手写编译器》中再详细介绍这个算法。

Qt 学习之路(48): 自定义委托

好久没有来写文章了，由于家里面宽带断了，所以一直没能更新，今天现在写上一篇。

还是继续前面的内容。前面我们分三次把自定义 `model` 说完了，其实主要还是那三个实例。在 `model/view` 架构中，与 `model` 同等重要的就是 `view`。

我们知道，在经典的 MVC 模型中，`view` 用于向用户展示 `model` 的数据。但是，Qt 提供的不是 MVC 三层架构，而是一个 `model/view` 设计。这种设计并没有包含一个完整而独立的

组件用于管理用户的交互。一般来说，**view** 仅仅是用作对 **model** 数据的展示和对用户输入的处理，而不应该去做其他的工作。在这种结构中，为了获得对用户输入控制的灵活性，这种交互工作交给了 **delegate**，也就是“委托”，去完成。简单来说，就像它们的名字一样，**view** 将用户输入委托给 **delegate** 处理，而自己去处理这种输入。这些组件提供一种输入能力，并且能够在某些 **view** 中提供这种交互情形下的渲染，比如在 **table** 中通过双击单元格即可编辑内容等。对这种控制委托的标准接口被定义在 **QAbstractItemDelegate** 类中。

delegate 可以用于渲染内容，这是通过 **paint()** 和 **sizeHint()** 函数来完成的。但是，对于一些简单的基于组件的 **delegate**，可以通过继承 **QItemDelegate** 或者 **QStyledItemDelegate** 来实现。这样就可以避免要完全重写 **QAbstractItemDelegate** 中所需要的所有函数。对于一些相对比较通用的函数，在这两个类中已经有了一个默认的实现。

Qt 提供的标准组件使用 **QItemDelegate** 提供编辑功能的支持。这种默认的实现被用在 **QListView**, **QTableView** 和 **QTreeView** 之中。**view** 实用的 **delegate** 可以通过 **itemDelegate()** 函数获得。**setItemDelegate()** 函数则可以为一个标准组件设置自定义的 **delegate**。

Qt 4.4 版本之后提供了两个可以被继承的 **delegate** 类：**QItemDelegate** 和 **QStyledItemDelegate**。默认的 **delegate** 是 **QStyledItemDelegate**。这两个类可以被相互替代，用于给 **view** 组件提供绘制和编辑的功能。它们之间的主要区别在于，**QStyledItemDelegate** 使用当前的风格(**style**)去绘制组件。所以，在自定义 **delegate** 或者需要使用 Qt style sheets 时，建议使用 **QStyledItemDelegate** 作为父类。使用这两个类的代码通常是一样的，除了需要使用 **style** 进行绘制的部份。如果你希望为 **view item** 自定义绘制函数，最好实现一个自定义的 **style**。这个你可以通过 **QStyle** 类来实现。

如果 **delegate** 没有支持为你的数据类型进行绘制，或者你希望自己绘制 **item**，那么就可以继承 **QStyledItemDelegate** 类，并且重写 **paint()** 或者还需要重写 **sizeHint()** 函数。**paint()** 函数会被每一个 **item** 独立调用，而 **sizeHint()** 函数则可以定义每一个 **item** 的大小。在重写 **paint()** 函数的时候，通常需要用 **if** 语句找到你需要进行渲染的数据类型并进行绘制，其他的数据类型需要调用父类的实现进行绘制。

一个自定义的 **delegate** 也可以直接提供一个编辑器，而不是使用内置的编辑器工厂(**editor item factory**)。如果你需要这种功能，那么需要实现一下几个函数：

- **createEditor()**: 返回修改数据的组件；
- **setEditorData()**: 为 **editor** 提供编辑的原始数据；
- **updateEditorGeometry()**: 保证 **editor** 显示在 **item view** 的合适位置以及大小；
- **setModelData()**: 根据 **editor** 的数据更新 **model** 的数据。

好了，这就是一个自定义 **delegate** 的实现了。下面来看一个例子。

这是一个歌曲及其时间的例子。使用的是 **QTableWidget**，一共有两列，第一列是歌曲名字，第二列是歌曲持续的时间。为了表示这个数据，我们建立一个 **Track** 类：

track.h

```
#ifndef TRACK_H
#define TRACK_H
```

```

#include <QtCore>

class Track
{
public:
    Track(const QString &title = "", int duration = 0);

    QString title;
    int duration;
};

#endif // TRACK_H

```

track.cpp

```

#include "track.h"

Track::Track(const QString &title, int duration)
    : title(title), duration(duration)
{
}

```

这个类的构造函数没有做任何操作，只是把 **title** 和 **duration** 这两个参数通过构造函数初始化列表赋值给内部的成员变量。注意，现在这两个成员变量都是 **public** 的，在正式的程序中应该声明为 **private** 的才对。然后来看 **TrackDelegate** 类：

trackdelegate.h

```

#ifndef TRACKDELEGATE_H
#define TRACKDELEGATE_H

#include <QtGui>

class TrackDelegate : public QStyledItemDelegate
{
    Q_OBJECT

public:
    TrackDelegate(int durationColumn, QObject *parent = 0);

    void paint(QPainter *painter, const QStyleOptionViewItem &option, const QModelIndex &index) const;

    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option, const QModelIndex &index) const;

    void setEditorData(QWidget *editor, const QModelIndex &index) const;

```

```

        void setModelData(QWidget *editor, QAbstractItemModel *model, const
QModelIndex &index) const;

private slots:
    void commitAndCloseEditor();

private:
    int durationColumn;
};

#endif // TRACKDELEGATE_H

```

trackdelegate.cpp

```

#include "trackdelegate.h"

TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
    : QStyledItemDelegate(parent)
{
    this->durationColumn = durationColumn;
}

void TrackDelegate::paint(QPainter *painter, const QStyleOptionViewItem &option, const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QString text = QString("%1:%2").arg(secs / 60, 2, 10, QChar('0')).arg(secs % 60, 2, 10, QChar('0'));
        QTextOption o(Qt::AlignRight | Qt::AlignVCenter);
        painter->drawText(option.rect, text, o);
    } else {
        QStyledItemDelegate::paint(painter, option, index);
    }
}

QWidget *TrackDelegate::createEditor(QWidget *parent, const QStyleOptionViewItem &option, const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()), this, SLOT(commitAndCloseEditor()));
    }
}

```

```

        return timeEdit;
    } else {
        return QStyledItemDelegate::createEditor(parent, option, index);
    }
}

void TrackDelegate::commitAndCloseEditor()
{
    QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}

void TrackDelegate::setEditorData(QWidget *editor, const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
        QStyledItemDelegate::setEditorData(editor, index);
    }
}

void TrackDelegate::setModelData(QWidget *editor, QAbstractItemModel *model, const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QStyledItemDelegate::setModelData(editor, model, index);
    }
}

```

正如前面所说的，这个类继承了 `QStyledItemDelegate`，覆盖了其中的四个函数。通过前面的讲解，我们已经了解到这些函数的作用。至于实现，我们前面也说过，需要通过 `QModelIndex` 选择我们需要进行渲染的列，然后剩下的数据类型仍然需要显式地调用父类的相应函数。由于我们在 `Track` 里面存储的是歌曲的秒数，所以在 `paint()` 里面需要用除法计算出分钟数，用 `% 60` 计算秒数。其他的函数都比较清楚，请注意代码。

最后写一个使用的类:

trackeditor.h

```
#ifndef TRACKEDITOR_H
#define TRACKEDITOR_H

#include <QtGui>
#include "track.h"

class TrackEditor : public QDialog
{
    Q_OBJECT

public:
    TrackEditor(QList<Track> *tracks, QWidget *parent);

private:
    QList<Track> *tracks;
    QTableWidgetItem *tableWidget;
};

#endif // TRACKEDITOR_H
```

trackeditor.cpp

```
#include "trackeditor.h"
#include "trackdelegate.h"

TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
    : QDialog(parent)
{
    this->tracks = tracks;

    tableWidget = new QTableWidgetItem(tracks->count(), 2);
    tableWidget->setItemDelegate(new TrackDelegate(1));
    tableWidget->setHorizontalHeaderLabels(QStringList() << tr("Track") <<
tr("Duration"));

    for (int row = 0; row < tracks->count(); ++row) {
        Track track = tracks->at(row);

        QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
        tableWidget->setItem(row, 0, item0);

        QTableWidgetItem *item1 = new QTableWidgetItem(QString::num
```

```

ber(track.duration));
        item1->setTextAlignment(Qt::AlignRight);
        tableWidget->setItem(row, 1, item1);
    }

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(tableWidget);
    this->setLayout(mainLayout);
}

```

其实也并没有很大的不同，只是我们使用 `setItemDelegate()` 函数设置了一下 `delegate`。然后写 `main()` 函数：

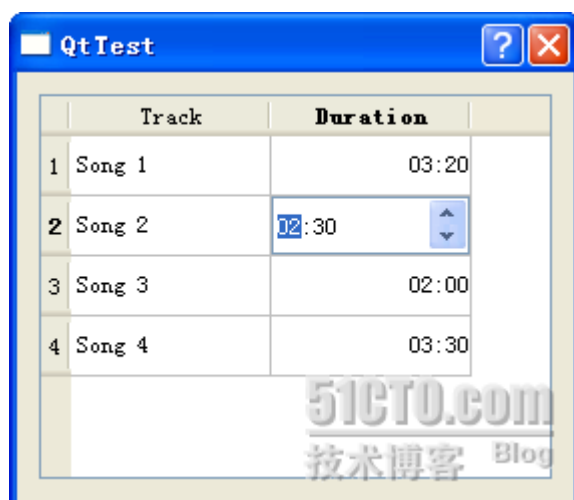
```

#include <QtGui>
#include "trackeditor.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QList<Track> tracks;
    Track t1("Song 1", 200);
    Track t2("Song 2", 150);
    Track t3("Song 3", 120);
    Track t4("Song 4", 210);
    tracks << t1 << t2 << t3 << t4;
    TrackEditor te(&tracks, NULL);
    te.show();
    return a.exec();
}

```

好了，运行一下看看效果吧！



Qt 学习之路(49): 通用算法

关于 Qt 的 **model-view** 部分就告一段落，今天我们开始新的部分。或许有些朋友觉得前面的部分说得很简单。对此我也没有办法，毕竟，Qt 是一个很庞大的库，一时半会根本不可能穷尽所有内容，并且我也很多东西不知道，有时候也必须去查找资料才能明白。

今天开始的部分是关于 Qt 提供的一些通用算法。这部分内容来自 **C++ GUI Programming with Qt 4, 2nd Edition**。

<QtAlgorithms> 提供了一系列通用的模板函数，用于实现容器上面的基本算法。这部分算法很多依赖于 STL 风格的遍历器(还记得前面曾经说过的 Java 风格的遍历器和 STL 风格的遍历器吗？)。实际上，C++ STL 也提供了很多通用算法，包含在 **<algorithm>** 头文件内。这部分算法对于 Qt 容器同样也是适用的。因此，如果你想使用的算法在 Qt 的 **<QtAlgorithms>** 头文件中没有包含，那么就可以使用 STL 的算法代替，这并不会产生什么冲突。这里我们来说几个 Qt 中的通用算法。虽然这些算法都是很简单，但是，库函数往往会比自己编写的更有效率，因此还是推荐使用系统提供的函数的。

首先是 **qFind()** 函数。**qFind()** 函数会在容器中查找一个特定的值。它的参数中有一个起始位置和终止位置，如果被查找的元素存在，函数返回第一个匹配项的位置，否则则返回终止位置。注意，我们这里说的“位置”，实际上是 STL 风格的遍历器。我们知道，使用 STL 风格遍历器是可以反映一个位置的。例如下面的例子，**i** 的值将是 **list.begin() + 1**，而 **j** 会是 **list.end()**：

```
QStringList list;
list << "Emma" << "Karl" << "James" << "Marianne";

QStringList::iterator i = qFind(list.begin(), list.end(), "Karl");
QStringList::iterator j = qFind(list.begin(), list.end(), "Petra");
```

qBinaryFind() 的行为很像 **qFind()**，所不同的是，**qBinaryFind()** 是二分查找算法，它只适用于查找排序之后的集合，而 **qFind()** 则是标准的线性查找。通常，二分查找法使用条件更为苛刻，但是效率也会更高。

qFill() 会使用给定值对容器进行填充。例如：

```
QLinkedList<int> list(10);
qFill(list.begin(), list.end(), 1009);
```

正如其他基于遍历器的算法一样，**qFill()** 也可以针对容器的一部分进行操作，例如下面的代码将会把 **vector** 的前 5 位设置成 1009，而最后 5 位设置为 2013：

```
QVector<int> vect(10);
qFill(vect.begin(), vect.begin() + 5, 1009);
qFill(vect.end() - 5, vect.end(), 2013);
```

qCopy() 算法可以实现将一个容器中的元素复制到另一个容器，例如：

```

| QVector<int> vect(list.count());
| qCopy(list.begin(), list.end(), vect.begin());

```

qCopy()也可以用于同一容器中的元素的复制。qCopy()操作成功的关键是源容器和目的容器的范围不会发生溢出。例如如下代码，我们将把一个列表的最后两个元素复制给前两个元素：

```

| qCopy(list.begin(), list.begin() + 2, list.end() - 2);

```

qSort()实现了容器元素的递增排序，使用起来也很简单：

```

| qSort(list.begin(), list.end());

```

默认情况下，qSort()将使用 < 运算符进行元素的比较。这暗示如果需要的话，你必须定义 < 运算符。如果需要按照递减排序，需要将 qGreater<T>() 当作第三个参数传给 qSort() 函数。例如：

```

| qSort(list.begin(), list.end(), qGreater<int>());

```

注意，这里的 T 实际上是容器的泛型类型。实际上，我们可以利用第三个参数对排序进行定义。例如，我们自定义的数据类型中有一个大小写不敏感的 QString 的小于比较函数：

```

| bool insensitiveLessThan(const QString &str1, const QString &str2)
| {
|     return str1.toLower() < str2.toLower();
| }

```

那么，我们可以这样使用 qSort() 从而可以利用这个函数：

```

| QStringList list;
| // ...
| qSort(list.begin(), list.end(), insensitiveLessThan);

```

qStableSort() 函数类似与 qSort()，所不同之处在于它是稳定排序。稳定排序是算法设计上的一个名词，意思是，在排序过程中，如果有两个元素相等，那么在排序结果中这两个元素的先后顺序同排序前的原始顺序是一致的。举个例子，对于一个序列：a1, a5, a32, a31, a4，它们的大小顺序是 a1 < a31 = a32 < a4 < a5，那么稳定排序之后的结果应该是 a1, a32, a31, a4, a5，也就是相等的元素在排序结果中出现的顺序和原始顺序是一致的。稳定排序在某些场合是很有用的，比如，现在有一份按照学号排序的学生成绩单。你想按照成绩高低重新进行排序，对于成绩一样的学生，还是遵循原来的学号顺序。这时候就要稳定排序了。

qDeleteAll() 函数将对容器中存储的所有指针进行 delete 操作。这个函数仅在容器元素是指针的情形下才适用。执行过这个函数之后，容器中的指针均被执行了 delete 运算，但是这些指针依然被存储在容器中，成为野指针，你需要调用容器的 clear() 函数来避免这些指针的误用：

```
qDeleteAll(list);  
list.clear();
```

qSwap() 函数可以交换两个元素的位置。例如：

```
int x1 = line.x1();  
int x2 = line.x2();  
if (x1 > x2)  
    qSwap(x1, x2);
```

最后，在<QtGlobal>头文件中，也定义了几个有用的函数。这个头文件被其他所有的头文件 include 了，因此你不需要显式的 include 这个头文件了。

在这个头文件中有这么几个函数：qAbs() 返回参数的绝对值，qMin() 和 qMax() 则返回两个值的最大值和最小值。

Qt 学习之路(50): QString

这段时间回家，一直没有来得及写，今天才发现博客的编辑器有了新版。还是先来试试新版编辑器的功能吧！

今天要说的是 **QString**。之所以把 **QString** 单独拿出来，是因为 **string** 是很常用的一个数据结构，甚至在很多语言中，比如 **JavaScript**，都是把 **string** 作为一种同 **int** 等一样的基本数据结构来实现的。

每一个 **GUI** 程序都需要 **string**，这些 **string** 可以用在界面上的提示语，也可以用作一般的数据结构。**C++** 语言提供了两种字符串的实现：**C** 风格的字符串，以 '\0' 结尾；**std::string**，即标准模版库中的类。**Qt** 则提供了自己的字符串实现：**QString**。**QString** 以 16 位 **Unicode** 进行编码。我们平常用的 **ASCII** 等一些编码集都作为 **Unicode** 编码的子集提供。关于编码的问题，我们会到以后的时候再详细说明。

在使用 **QString** 的时候，我们不需要担心内存分配以及关于 '\0' 结尾的这些注意事项。**QString** 会把这些问题解决。通常，你可以把 **QString** 看作是一个 **QChar** 的向量。另外，与 **C** 风格的字符串不同，**QString** 中间是可以包含 '\0' 符号的，而 **length()** 函数则会返回整个字符串的长度，而不仅仅是从开始到 '\0' 的长度。

同 **Java** 的 **String** 类类似，**QString** 也重载的 **+** 和 **+=** 运算符。这两个运算符可以把两个字符串连接到一起，正像 **Java** 里面的操作一样。**QString** 可以自动的对占用内存空间进行扩充，这种连接操作是很迅速的。下面是这两个操作符的使用：

```
1.    QString str = "User: ";  
2.    str += userName + "\n";
```

QString 的 **append()** 函数则提供了类似的操作，例如：

```
1.    str = "User: ";
```

```
2.     str.append(userName);
3.     str.append("\n");
```

C 语言中有 `printf()` 函数作为格式化输出，`QString` 则提供了一个 `sprintf()` 函数实现了相同的功能：

```
1.     str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

这句代码将输出：**perfect competition 100.0%**，同 C 语言的 `printf()` 一样。不过前面我们也见到了 Qt 提供的另一种格式化字符串输出的函数 `arg()`：

```
1.     str = QString("%1 %2 (%3s-%4s)")
2.     .arg("permissive").arg("society").arg(1950).arg(1970);
```

这段代码中，`%1`，`%2`，`%3`，`%4` 作为占位符，将被后面的 `arg()` 函数中的内容依次替换，比如 `%1` 将被替换成 `permissive`，`%2` 将被替换成 `society`，`%3` 将被替换成 `1950`，`%4` 将被替换成 `1970`，最后，这句代码输出为：**permissive society (1950s-1970s)**。`arg()` 函数比起 `sprintf()` 来是类型安全的，同时它也接受多种的数据类型作为参数，因此建议使用 `arg()` 函数而不是传统的 `sprintf()`。

使用 `static` 的函数 `number()` 可以把数字转换成字符串。例如：

```
1.     QString str = QString::number(54.3);
```

你也可以使用非 `static` 函数 `setNum()` 来实现相同的目的：

```
1.     QString str;
2.     str.setNum(54.3);
```

而一系列的 `to` 函数则可以将字符串转换成其他基本类型，例如 `toInt()`，`toDouble()`，`toLong()` 等。这些函数都接受一个 `bool` 指针作为参数，函数结束之后将根据是否转换成功设置为 `true` 或者 `false`：

```
1.     bool ok;
2.     double d = str.toDouble(&ok);
3.     if(ok)
4.     {
5.         // do something...
6.     } else {
7.         // do something...
8.     }
```

对于 `QString`，Qt 提供了很多操作函数，例如，使用 `mid()` 函数截取子串：

```
1.     QString x = "Nine pineapples";
2.     QString y = x.mid(5, 4);           // y == "pine"
```

```
3.    QString z = x.mid(5);           // z == "pineapples"
```

`mid()`函数接受两个参数，第一个是起始位置，第二个是取串的长度。如果省略第二个参数，则会从起始位置截取到末尾。正如上面的例子显示的那样。

函数 `left()`和 `right()`类似，都接受一个 `int` 类型的参数 `n`，都是对字符串进行截取。不同之处在于，`left()`函数从左侧截取 `n` 个字符，而 `right()`从右侧开始截取。下面是 `left()`的例子：

```
1.    QString x = "Pineapple";
2.    QString y = x.left(4);         // y == "Pine"
```

函数 `indexOf()`返回字符串的位置，如：

```
1.    QString x = "sticky question";
2.    QString y = "sti";
3.    x.indexOf(y);                  // returns 0
4.    x.indexOf(y, 1);               // returns 10
5.    x.indexOf(y, 10);              // returns 10
6.    x.indexOf(y, 11);              // returns -1
```

函数 `startsWith()`和 `endsWith()`可以检测字符串是不是以某个特定的串开始或结尾，例如：

```
1.    if (url.startsWith("http:") && url.endsWith(".png"))
2.    {
3.    }
```

这段代码等价于

```
1.    if (url.left(5) == "http:" && url.right(4) == ".png")
2.    {
3.    }
```

不过，前者要比后者更加清楚简洁，并且性能也更快一些。

`QString` 还提供了 `replace()`函数供实现字符串的替换功能；`trimmed()`函数去除字符串两侧的空白字符(注意，空白字符包括空格、`Tab` 以及换行符，而不仅仅是空格)；`toLowerCase()`和 `toUpperCase()`函数会将字符串转换成小写大写字符串；`remove()`和 `insert()`函数提供了删除和插入字符串的能力；`simplified()`函数可以将串中的所有连续的空白字符替换成一个，并且把两端的空白字符去除，例如 " \t "会返回一个空格" "。

将 `const char *`类型的 C 风格字符串转换成 `QString` 也是很常见的需求，简单来说，`QString` 的 `+=`即可完成这个功能：

```
1.    str += " (1870)";
```

这里，我们将 `const char *` 类型的字符串 `"(1870)"` 转换为 `QString` 类型。如果需要显式的转换，可以使用 `QString` 的强制转换操作，或者是使用函数 `fromAscii()` 等。为了将 `QString` 类型转成 `const char *` 字符串，需要进行两步操作，一是使用 `toAscii()` 获得一个 `QByteArray` 类型对象，然后调用它的 `data()` 或者 `constData()` 函数，例如：

```
1.    printf("User: %s\n", str.toAscii().data());
```

为了方便使用，Qt 提供了一个宏 `qPrintable()`，这个宏等价于 `toAscii().constData()`，例如：

```
1.    printf("User: %s\n", qPrintable(str));
```

我们调用 `QByteArray` 类上面的 `data()` 或者 `constData()` 函数，将获得 `QByteArray` 内部的一个 `const char*` 类型的字符串，因此，我们不需要担心内存泄漏等问题，Qt 会替我们管理好内存。不过这也暗示我们，注意不要使用这个指针太长时间，因为如果 `QByteArray` 被 `delete`，那么这个指针也就成为野指针了。如果这个 `QByteArray` 对象没有被放在一个变量中，那么当语句结束后，`QByteArray` 对象就会被 `delete`，这个指针也就被 `delete` 了。

Qt 学习之路(51): QByteArray 和 QVariant

前面我们在介绍 `QString` 的最后部分曾经提到了 `QByteArray` 这个类。现在我们就首先对这个类进行介绍。

`QByteArray` 具有类似与 `QString` 的 API。它也有相应的函数，比如 `left()`，`right()`，`mid()` 等。这些函数不仅名字和 `QString` 一样，而且也具有几乎相同的功能。`QByteArray` 可以存储原生的二进制数据和 8 位编码的文本数据。这句话怎么理解呢？我们知道，计算机内部所有的数据都是以 0 和 1 的形式存储的。这种形式就是二进制。比如一串 0、1 代码：1000，计算机并不知道它代表的是什么，这需要由上下文决定：它可以是整数 8，也可以是一个 ARGB 的颜色(准确的说，整数 8 的编码并不是这么简单，但我们姑且这个理解吧)。对于文件，即便是一个文本文件，读出时也可以按照二进制的形式读出，这就是二进制格式。如果把这些二进制的 0、1 串按照编码解释成一个个字符，就是文本形式了。因此，`QByteArray` 实际上是原生的二进制，但是也可以当作是文本，因此拥有文本的一些操作。但是，我们还是建议使用 `QString` 表示文本，重要的原因是，`QString` 支持 Unicode。

为了方便期间，`QByteArray` 自动的保证“最后一个字节之后的那个位”是 `'\0'`。这就使得 `QByteArray` 可以很容易的转换成 `const char *`，也就是上一章节中我们提到的那两个函数。同样，作为原生二进制存储，`QByteArray` 中间也可以存储 `'\0'`，而不必须是 `'\0'` 在最后一位。

在有些情况下，我们希望把数据存储在一个变量中。例如，我有一个数组，既希望存整数，又希望存浮点数，还希望存 `string`。对于 Java 来说，很简单，只要把这个数组声明成 `Object[]` 类型的。这是什么意思呢？实际上，这里用到的是继承。在 Java 中，`int` 和 `float` 虽然是原生数据类型，但是它们都有分别对应一个包装类 `Integer` 和 `Float`。所有这些 `Integer`、`Float` 和 `String` 都是继承于 `Object`，也就是说，`Integer`、`Float` 和 `String` 都是一个(也就是 is-a 的关系) `Object`，这样，`Object` 的数组就可以存储不同的类型。但是，C++ 中没有这样一个 `Object` 类，原因在于，Java 是单根的，而 C++ 不是。在 Java 中，所有类都可以上溯到 `Object` 类，但是 C++ 中没有这么一个根。那么，怎么实现这么的操作呢？一种办法是，我们都存成 `string`

类，比如 `int i=10`，我就存"10"字符串。简单的数据类型固然可以，可复杂一些的呢？比如一个颜色？难道要把 **ARGB** 所有的值都转化成 `string`？这种做法很复杂，而且失去了 **C++** 的类型检查等好处。于是我们想另外的办法：创建一个 **Object** 类，这是一个“很大很大的”类，里面存储了几乎所有的数据类型，比如下面的代码：

```
1.     class Object
2.     {
3.     public:
4.         int intValue;
5.         float floatValue;
6.         string stringValue;
7.     };
```

这个类怎么样？它足以存储 `int`、`float` 和 `string` 了。嗯，这就是我们的思路，也是 **Qt** 的思路。在 **Qt** 中，这样的类就是 `QVariant`。

`QVariant` 可以保存很多 **Qt** 的数据类型，包括 `QBrush`、`QColor`、`QCursor`、`QDateTime`、`QFont`、`QKeySequence`、`QPalette`、`QPen`、`QPixmap`、`QPoint`、`QRect`、`QRegion`、`QSize` 和 `QString`，并且还有 **C++** 基本类型，如 `int`、`float` 等。`QVariant` 还能保存很多集合类型，如 `QMap<QString, QVariant>`、`QStringList` 和 `QList<QVariant>`。item view classes，数据库模块和 `QSettings` 都大量使用了 `QVariant` 类，，以方便我们读写数据。

`QVariant` 也可以进行嵌套存储，例如

```
1.     QMap<QString, QVariant> pearMap;
2.     pearMap["Standard"] = 1.95;
3.     pearMap["Organic"] = 2.25;
4.
5.     QMap<QString, QVariant> fruitMap;
6.     fruitMap["Orange"] = 2.10;
7.     fruitMap["Pineapple"] = 3.85;
8.     fruitMap["Pear"] = pearMap;
```

`QVariant` 被用于构建 **Qt Meta-Object**，因此是 **QtCore** 的一部分。当然，我们也可以在 **GUI** 模块中使用，例如

```
1.     QIcon icon("open.png");
2.     QVariant variant = icon;
3.     // other function
4.     QIcon icon = variant.value<QIcon>();
```

我们使用了 `value<T>()` 模版函数，获取存储在 `QVariant` 中的数据。这种函数在非 **GUI** 数据中同样适用，但是，在非 **GUI** 模块中，我们通常使用 `toInt()` 这样的一系列 `to...()` 函数，如 `toString()` 等。

如果你觉得 **QVariant** 提供的存储数据类型太少，也可以自定义 **QVariant** 的存储类型。被 **QVariant** 存储的数据类型需要有一个默认的构造函数和一个拷贝构造函数。为了实现这个功能，首先必须使用 **Q_DECLARE_METATYPE()**宏。通常会将这个宏放在类的声明所在头文件的下面：

```
1.      Q_DECLARE_METATYPE(BusinessCard)
```

然后我们就可以使用：

```
1.      BusinessCard businessCard;
2.      QVariant variant = QVariant::fromValue(businessCard);
3.      // ...
4.      if (variant.canConvert<BusinessCard>()) {
5.          BusinessCard card = variant.value<BusinessCard>();
6.          // ...
7.      }
```

由于 VC 6 的编译器限制，这些模板函数不能使用，如果你使用这个编译器，需要使用 **qVariantFromValue()**, **qVariantValue<T>()** 和 **qVariantCanConvert<T>()** 这三个宏。

如果自定义数据类型重写了 **<<**和**>>**运算符，那么就可以直接在 **QDataStream** 中使用。不过首先需要使用 **qRegisterMetaTypeStreamOperators<T>()**宏进行注册。这就能够让 **QSettings** 使用操作符对数据进行操作，例如

```
1.      qRegisterMetaTypeStreamOperators<BusinessCard>("BusinessCard");
```

Qt 学习之路(tip): Qt 容器和算法拾遗

Qt 提供了另外的容器，比如 **QPair<T1, T2>**，可以存储两个值，类似于 **std::pair<T1, T2>**。还有 **QVarLengthArray<T, Prealloc>**，这是一个 **QVector<T>**的低级实现。因为它需要预分配内存，并且没有隐式的内存共享机制。但是它的开销低于 **QVector<T>**，更适合资源紧张的情况。

关于 **Qt** 的通用算法，还有 **qCopyBackward()**和 **qEqual()**两个。具体可以查阅 **Qt** 文档中 **Algorithms** 一章。

Qt 学习之路(52): 拖放技术之一

拖放 **Drag and Drop**，有时又被称为 **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，并且，操作系统与应用程序之间进行剪贴板的内容交换，也可以被认为是 **DnD** 的一部分。

DnD 其实是由两部分组成的：**Drag** 和 **Drop**。**Drag** 是将被拖放对象“拖动”，**Drop** 是将被拖放对象“放下”，前者一般是一个按下鼠标的过程，而后者则是一个松开鼠标的过程，这两者之间鼠标一直是被按下的。当然，这只是一种通常的情况，其他情况还是要看应用程序的具体实现。

对于 **Qt** 而言，**widget** 既可以作为 **drag** 对象，也可以作为 **drop** 对象，或者二者都是。下面的一个例子来自 **C++ GUI Programming with Qt 4, 2nd Edition**。在这个例子中，我们创建一个程序，可以将系统中的文本文件拖放过来，然后在窗口中读取内容。

mainwindow.h

```
1.  #ifndef MAINWINDOW_H
2.  #define MAINWINDOW_H
3.
4.  #include <QtGui>
5.
6.  class MainWindow : public QMainWindow
7.  {
8.      Q_OBJECT
9.
10.     public:
11.         MainWindow(QWidget *parent = 0);
12.         ~MainWindow();
13.
14.     protected:
15.         void dragEnterEvent(QDragEnterEvent *event);
16.         void dropEvent(QDropEvent *event);
17.
18.     private:
19.         bool readFile(const QString &fileName);
20.         QTextEdit *textEdit;
21.     };
22.
23. #endif // MAINWINDOW_H
```

mainwindow.cpp

```
1.  #include "mainwindow.h"
2.
3.  MainWindow::MainWindow(QWidget *parent)
4.      : QMainWindow(parent)
5.  {
6.      textEdit = new QTextEdit;
7.      setCentralWidget(textEdit);
8.
9.      textEdit->setAcceptDrops(false);
10.     setAcceptDrops(true);
11.
12.     setWindowTitle(tr("Text Editor"));
13. }
14.
15.  MainWindow::~MainWindow()
16.  {
17.  }
18.
19.  void MainWindow::dragEnterEvent(QDragEnterEvent *event)
```

```

20.     {
21.         if (event->mimeType() != "text/uri-list") {
22.             event->acceptProposedAction();
23.         }
24.     }
25.
26. void MainWindow::dropEvent(QDropEvent *event)
27. {
28.     QList<QUrl> urls = event->mimeType()->urls();
29.     if (urls.isEmpty()) {
30.         return;
31.     }
32.
33.     QString fileName = urls.first().toLocalFile();
34.     if (fileName.isEmpty()) {
35.         return;
36.     }
37.
38.     if (readFile(fileName)) {
39.         setWindowTitle(tr("%1 - %2").arg(fileName, tr("Drag File")));
40.     }
41. }
42.
43. bool MainWindow::readFile(const QString &fileName)
44. {
45.     bool r = false;
46.     QFile file(fileName);
47.     QTextStream in(&file);
48.     QString content;
49.     if(file.open(QIODevice::ReadOnly)) {
50.         in >> content;
51.         r = true;
52.     }
53.     textEdit->setText(content);
54.     return r;
55. }

```

main.cpp

```

1.  #include <QtGui/QApplication>
2.  #include "mainwindow.h"
3.
4.  int main(int argc, char *argv[])
5.  {
6.      QApplication a(argc, argv);

```

```

7.         MainWindow w;
8.         w.show();
9.         return a.exec();
10.    }

```

这里的代码并不是很复杂。在 `MainWindow` 中，一个 `QTextEdit` 作为窗口中间的 widget。这个类中有两个 `protected` 的函数: `dragEnterEvent()` 和 `dropEvent()`，这两个函数都是继承自 `QWidget`，看它们的名字就知道这是两个事件，而不仅仅是 `signal`。

在构造函数中，我们创建了 `QTextEdit` 的对象。默认情况下，`QTextEdit` 可以接受从其他的应用程序拖放过来的文本类型的信息。如果用户把一个文件拖到这里面，那么就会把文件名插入到文本的当前位置。但是我们希望让 `MainWindow` 读取文件内容，而不仅仅是插入文件名，所以我们在 `MainWindow` 中对 `drop` 事件进行了处理，因此要把 `QTextEdit` 的 `setAcceptDrops()` 函数置为 `false`，并且把 `MainWindow` 的 `setAcceptDrops()` 置为 `true`，以便让 `MainWindow` 对 `drop` 事件进行处理。

当用户将对象拖动到组件上面时，`dragEnterEvent()` 函数会被回调。如果我们在事件处理代码中调用 `acceptProposeAction()` 函数，我们就可以向用户暗示，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。如果我们调用了这样的函数，那么 `Qt` 会自动地以光标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。因此，我们首先检查拖放的 `MIME` 类型。`MIME` 类型为 `text/uri-list` 通常用来描述一个 `URI` 的列表。这些 `URI` 可以是文件名，可以是 `URL` 或者其他的资源描述符。`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/>。

当用户将对象释放到组件上面时，`dropEvent()` 函数会被回调。我们使用 `QMimeData::urls()` 来或者 `QUrl` 的一个 `list`。通常，这种拖动应该只用一个文件，但是也不排除多个文件一起拖动。因此我们需要检查这个 `list` 是否为空，如果不为空，则取出第一个。如果不成立，则立即返回。最后我们调用 `readFile()` 函数读取文件内容。关于读取操作我们会在以后的章节中详细说明，这里不再赘述。

好了，至此我们的小程序就解释完毕了，运行一下看看效果吧！

对于拖动和脱离，`Qt` 也提供了类似的函数: `dragMoveEvent()` 和 `dragLeaveEvent()`，不过对于大部分应用而言，这两个函数的使用率要小得多。

Qt 学习之路(53): 拖放技术之二

很长时间没有来写博客了，前段时间一直在帮同学弄一个 `spring-mvc` 的项目，今天终于做完了，不过公司里面又要开始做 `flex 4`，估计还会忙一段时间吧！

接着上次的说，上次说到了拖放技术，今天依然是一个例子，同样是来自《`C++ GUI Programming with Qt 4, 2nd Edition`》的。

这次的 `demo` 还算是比较实用：实现的是两个 `list` 之间的数据互拖。在很多项目中，这一需求还是比较常见的吧！下面也就算是抛砖引玉了啊！

projectlistwidget.h

```

1.     #ifndef PROJECTLISTWIDGET_H
2.     #define PROJECTLISTWIDGET_H
3.
4.     #include <QtGui>
5.
6.     class ProjectListWidget : public QListWidget

```

```

7.     {
8.         Q_OBJECT
9.
10.    public:
11.        ProjectListWidget(QWidget *parent = 0);
12.
13.    protected:
14.        void mousePressEvent(QMouseEvent *event);
15.        void mouseMoveEvent(QMouseEvent *event);
16.        void dragEnterEvent(QDragEnterEvent *event);
17.        void dragMoveEvent(QDragMoveEvent *event);
18.        void dropEvent(QDropEvent *event);
19.
20.    private:
21.        void performDrag();
22.
23.        QPoint startPos;
24.    };
25.
26. #endif // PROJECTLISTWIDGET_H

```

projectlistwidget.cpp

```

1.    #include "projectlistwidget.h"
2.
3.    ProjectListWidget::ProjectListWidget(QWidget *parent)
4.        : QListWidget(parent)
5.    {
6.        setAcceptDrops(true);
7.    }
8.
9.    void ProjectListWidget::mousePressEvent(QMouseEvent *event)
10.    {
11.        if (event->button() == Qt::LeftButton)
12.            startPos = event->pos();
13.        QListWidget::mousePressEvent(event);
14.    }
15.
16.    void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
17.    {
18.        if (event->buttons() & Qt::LeftButton) {
19.            int distance = (event->pos() - startPos).manhattanLength();
20.            if (distance >= QApplication::startDragDistance())
21.                performDrag();
22.        }
23.        QListWidget::mouseMoveEvent(event);

```

```

24.     }
25.
26.     void ProjectListWidget::performDrag()
27.     {
28.         QListWidgetItem *item = currentItem();
29.         if (item) {
30.             QMimeData *mimeData = new QMimeData;
31.             mimeData->setText(item->text());
32.
33.             QDrag *drag = new QDrag(this);
34.             drag->setMimeData(mimeData);
35.             drag->setPixmap(QPixmap(":/images/person.png"));
36.             if (drag->exec(Qt::MoveAction) == Qt::MoveAction)
37.                 delete item;
38.         }
39.     }
40.
41.     void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
42.     {
43.         ProjectListWidget *source =
44.             qobject_cast<ProjectListWidget *>(event->source());
45.         if (source && source != this) {
46.             event->setDropAction(Qt::MoveAction);
47.             event->accept();
48.         }
49.     }
50.
51.     void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
52.     {
53.         ProjectListWidget *source =
54.             qobject_cast<ProjectListWidget *>(event->source());
55.         if (source && source != this) {
56.             event->setDropAction(Qt::MoveAction);
57.             event->accept();
58.         }
59.     }
60.
61.     void ProjectListWidget::dropEvent(QDropEvent *event)
62.     {
63.         ProjectListWidget *source =
64.             qobject_cast<ProjectListWidget *>(event->source());
65.         if (source && source != this) {
66.             addItem(event->mimeData()->text());
67.             event->setDropAction(Qt::MoveAction);

```

```

68.         event->accept();
69.     }
70. }

```

我们从构造函数开始看起。Qt 中很多组件是可以接受拖放的，但是默认动作都是不允许的，因此在此构造函数中，我们调用 `setAcceptDrops(true);` 函数，让组件能够接受拖放事件。

在 `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！

然后我们重写了 `mouseMoveEvent()` 事件。下面还是先来看看代码：

```

1.  void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
2.  {
3.      if (event->buttons() & Qt::LeftButton) {
4.          int distance = (event->pos() - startPos).manhattanLength();
5.          if (distance >= QApplication::startDragDistance())
6.              performDrag();
7.      }
8.      QListWidget::mouseMoveEvent(event);
9.  }

```

在这里判断了如果鼠标拖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”。这是什么意思呢？我们看一下 `event.pos() - startPos` 是什么。还记得在 `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，没错，这就是向量！其实，所谓曼哈顿距离就是两点之间的距离(至于为什么叫这个奇怪的名字，大家查查百科就知道啦！)，也就是这个向量的长度。下面又是一个判断，如果大于 `QApplication::startDragDistance()`，我们才进行 `drag` 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()` 开始处理拖放过程。我们创建了一个 `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。这些动作的类型为 `Qt::CopyAction`，`Qt::MoveAction` 和 `Qt::LinkAction`。返回值会有这三种动作，同时增加一个 `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 `delete` 掉它。

`ProjectListWidget` 不仅能够发出拖动事件，而且能够接受同一应用程序中的不同 `ProjectListWidget` 对象的数据。在 `dragEnterEvent()` 中，我们使用 `event->source()` 获取这样的对象：如果拖放数据来自同一类型的对象，并且来自同一应用程序则返回其指针，否则返回 `NULL`。我们使用 `qobject_cast` 宏将指针转换成 `ProjectListWidget*` 类型，然后设置接受 `Qt::MoveAction` 类型的拖动。`dragMoveEvent()` 则和这个函数具有相同的代码，因为我们需要重写拖动移动的代码。

最后在 `dropEvent()` 函数中，我们取出 `QDrag` 中的 `mimeData` 数据，调用 `addItem()` 添加到当前的列表中。这样，一个相对完整的拖放的代码就完成了。

拖放技术是 Qt 中功能强大的一个技术，但是对于不涉及数据的同一组件中拖动，或许仅仅简单的实现 `mouse event` 就足够了，具体还是要自己斟酌啦！

Qt 学习之路(54): 自定义拖放数据对象

前面的例子都是使用的系统提供的拖放对象 `QMimeData` 进行拖放数据的存储，比如使用 `QMimeData::setText()` 创建文本，使用 `QMimeData::urls()` 创建 URL 对象。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 `QMimeData` 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 `QMimeData::setData()` 函数作为二进制数据存储到 `QMimeData` 中，然后使用 `QMimeData::Data()` 读取；
2. 继承 `QMimeData`，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 `QMimeData`，然后使用任意合适的数据结构进行存储。

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；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 `QMimeData` 类，这可能会导致类爆炸；如果数据很大的话，这种方式可能会降低系统的可维护性。然而，后两种实现方式就不会有这些问题，或者说是能够减小这种问题，并且能够让我们有完全控制权。

我们先来看一个应用，使用 `QTableWidget` 来进行拖放操作，拖放的类型包括 `plain/text`、`plain/html` 和 `plain/csv`。如果使用第一种实现方法，我们的代码将会如下所示：

```
1.  void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
2.  {
3.      if (event->buttons() & Qt::LeftButton) {
4.          int distance = (event->pos() - startPos).manhattanLength();
5.          if (distance >= QApplication::startDragDistance())
6.              performDrag();
7.      }
8.      QTableWidget::mouseMoveEvent(event);
9.  }
10.
11. void MyTableWidget::performDrag()
12. {
13.     QString plainText = selectionAsPlainText();
14.     if (plainText.isEmpty())
15.         return;
16.
17.     QMimeData *mimeData = new QMimeData;
18.     mimeData->setText(plainText);
19.     mimeData->setHtml(toHtml(plainText));
20.     mimeData->setData("text/csv", toCsv(plainText).toUtf8());
21.
22.     QDrag *drag = new QDrag(this);
23.     drag->setMimeData(mimeData);
```

```

24.         if (drag->exec(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
25.             deleteSelection();
26.     }

```

对于这段代码，我们应该已经很容易的理解：在 `performDrag()` 函数中，我们调用 `QMimeData` 的 `setText()` 和 `setHTML()` 函数存储 `plain/text` 和 `plain/html` 数据，使用 `setData()` 将 `text/csv` 类型的数据作为二进制 `QByteArray` 类型存储。

```

1.     QString MyTableWidget::toCsv(const QString &plainText)
2.     {
3.         QString result = plainText;
4.         result.replace("\\", "\\");
5.         result.replace("\"", "\\");
6.         result.replace("\t", "\t");
7.         result.replace("\n", "\n");
8.         result.prepend("\n");
9.         result.append("\n");
10.        return result;
11.    }
12.
13.    QString MyTableWidget::toHtml(const QString &plainText)
14.    {
15.        QString result = Qt::escape(plainText);
16.        result.replace("\t", "<td>");
17.        result.replace("\n", "<n<tr><td>");
18.        result.prepend("<table><n<tr><td>");
19.        result.append("<n</table>");
20.        return result;
21.    }

```

`toCsv()` 和 `toHtml()` 函数将数据取出并转换成我们需要的 `csv` 和 `html` 类型的数据。例如，下面的数据

```

Red  Green  Blue

Cyan  Yellow  Magenta

```

转换成 `csv` 格式为：

```

"Red", "Green", "Blue"

"Cyan", "Yellow", "Magenta"

```

转换成 `html` 格式为：

```

<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>

```

在放置的函数中我们像以前一样使用：

```

1.     void MyTableWidget::dropEvent(QDropEvent *event)

```



```

2.     {
3.         if (event->mimeType()->hasFormat("text/csv")) {
4.             QByteArray csvData = event->mimeType()->data("text/csv");
5.             QString csvText = QString::fromUtf8(csvData);
6.             // ...
7.             event->acceptProposedAction();
8.         } else if (event->mimeType()->hasFormat("text/plain")) {
9.             QString plainText = event->mimeType()->text();
10.            // ...
11.            event->acceptProposedAction();
12.        }
13.    }

```

虽然我们接受三种数据类型，但是在这个函数中我们只接受两种类型。至于 `html` 类型，我们希望如果用户将 `QTableWidget` 的数据拖到一个 `HTML` 编辑器，那么它就会自动转换成 `html` 代码，但是我们不计划支持将外部的 `html` 代码拖放到 `QTableWidget` 上。为了让这段代码能够工作，我们需要在构造函数中设置 `setAcceptDrops(true)` 和 `setSelectionMode(ContiguousSelection)`。

好了，上面就是我们所说的第一种方式的实现。这里并没有给出完整的实现代码，大家可以根据需要自己实现一下试试。下面我们将按照第二种方法重新实现这个需求。

```

1.     class TableMimeData : public QMimeData
2.     {
3.         Q_OBJECT
4.
5.     public:
6.         TableMimeData(const QTableWidget *tableWidget,
7.             const QTableWidgetSelectionRange &range);
8.
9.         const QTableWidget *tableWidget() const { return myTableWidget; }
10.        QTableWidgetSelectionRange range() const { return myRange; }
11.        QStringList formats() const;
12.
13.    protected:
14.        QVariant retrieveData(const QString &format,
15.            QVariant::Type preferredType) const;
16.
17.    private:
18.        static QString toHtml(const QString &plainText);
19.        static QString toCsv(const QString &plainText);
20.
21.        QString text(int row, int column) const;
22.        QString rangeAsPlainText() const;
23.
24.        const QTableWidget *myTableWidget;
25.        QTableWidgetSelectionRange myRange;

```

```

26.         QStringList myFormats;
27.     };

```

为了避免存储具体的数据，我们存储 **table** 和选择区域的坐标的指针。

```

1.     TableMimeData::TableMimeData(const QTableWidget *tableWidget,
2.                                   const QTableWidgetSelectionRange &range)
3.     {
4.         myTableWidget = tableWidget;
5.         myRange = range;
6.         myFormats << "text/csv" << "text/html" << "text/plain";
7.     }
8.
9.     QStringList TableMimeData::formats() const
10.    {
11.        return myFormats;
12.    }

```

构造函数中，我们对私有变量进行初始化。**formats()** 函数返回的是被 **MIME** 数据对象支持的数据类型列表。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。

```

1.     QVariant TableMimeData::retrieveData(const QString &format,
2.                                           QVariant::Type preferredType) const
3.     {
4.         if (format == "text/plain") {
5.             return rangeAsPlainText();
6.         } else if (format == "text/csv") {
7.             return toCsv(rangeAsPlainText());
8.         } else if (format == "text/html") {
9.             return toHtml(rangeAsPlainText());
10.        } else {
11.            return QMimeData::retrieveData(format, preferredType);
12.        }
13.    }

```

函数 **retrieveData()** 将给出的 **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。

```

1.     void MyTableWidget::dropEvent(QDropEvent *event)
2.     {
3.         const TableMimeData *tableData =
4.             qobject_cast<const TableMimeData *>(event->mimeType());
5.

```

```

6.         if (tableData) {
7.             const QWidget *otherTable = tableData->tableWidget();
8.             QTableWidgetItemSelectionRange otherRange = tableData->range();
9.             // ...
10.            event->acceptProposedAction();
11.        } else if (event->mimeType()->hasFormat("text/csv")) {
12.            QByteArray csvData = event->mimeType()->data("text/csv");
13.            QString csvText = QString::fromUtf8(csvData);
14.            // ...
15.            event->acceptProposedAction();
16.        } else if (event->mimeType()->hasFormat("text/plain")) {
17.            QString plainText = event->mimeType()->text();
18.            // ...
19.            event->acceptProposedAction();
20.        }
21.        QWidget::mousePressEvent(event);
22.    }

```

在放置的函数中，我们需要按照我们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 `QWidget` 相关数据，如果转换失败，我们则使用一般的处理方式。

Qt 学习之路(55): 剪贴板操作

大家对剪贴板都很熟悉。我们可以简单的把它理解成一个数据的存储池，可以把外面的数据放置进去，也可以把里面的数据取出来。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，因此我们可以在自己的应用中很容易的实现。下面还是从代码开始：

clipboarddemo.h

```

1.  #ifndef CLIPBOARDDEMO_H
2.  #define CLIPBOARDDEMO_H
3.
4.  #include <QtGui/QWidget>
5.
6.  class ClipboardDemo : public QWidget
7.  {
8.      Q_OBJECT
9.
10.     public:
11.         ClipboardDemo(QWidget *parent = 0);
12.
13.     private slots:
14.         void setClipboard();
15.         void getClipboard();
16.     };

```

```
17.  
18. #endif // CLIPBOARDDEMO_H
```

clipboarddemo.cpp

```
1. #include <QtGui>  
2. #include "clipboarddemo.h"  
3.  
4. ClipboardDemo::ClipboardDemo(QWidget *parent)  
5.     : QWidget(parent)  
6.     {  
7.     QVBoxLayout *mainLayout = new QVBoxLayout(this);  
8.     QHBoxLayout *northLayout = new QHBoxLayout;  
9.     QHBoxLayout *southLayout = new QHBoxLayout;  
10.  
11.     QTextEdit *editor = new QTextEdit;  
12.     QLabel *label = new QLabel;  
13.     label->setText("Text Input: ");  
14.     label->setBuddy(editor);  
15.     QPushButton *copyButton = new QPushButton;  
16.     copyButton->setText("Set Clipboard");  
17.     QPushButton *pasteButton = new QPushButton;  
18.     pasteButton->setText("Get Clipboard");  
19.  
20.     northLayout->addWidget(label);  
21.     northLayout->addWidget(editor);  
22.     southLayout->addWidget(copyButton);  
23.     southLayout->addWidget(pasteButton);  
24.     mainLayout->addLayout(northLayout);  
25.     mainLayout->addLayout(southLayout);  
26.  
27.     connect(copyButton, SIGNAL(clicked()), this, SLOT(setClipboard()));  
28.     connect(pasteButton, SIGNAL(clicked()), this, SLOT(getClipboard()));  
29. }  
30.  
31. void ClipboardDemo::setClipboard()  
32. {  
33.     QClipboard *board = QApplication::clipboard();  
34.     board->setText("Text from Qt Application");  
35. }  
36.  
37. void ClipboardDemo::getClipboard()  
38. {  
39.     QClipboard *board = QApplication::clipboard();
```

```

40.         QString str = board->text();
41.         QMessageBox::information(NULL, "From clipboard", str);
42.     }

```

main.cpp

```

1.     #include "clipboarddemo.h"
2.
3.     #include <QtGui>
4.     #include <QApplication>
5.
6.     int main(int argc, char *argv[])
7.     {
8.         QApplication a(argc, argv);
9.         ClipboardDemo w;
10.        w.show();
11.        return a.exec();
12.    }

```

`main()` 函数很简单，就是把我们的 `ClipboardDemo` 类显示了出来。我们重点来看 `ClipboardDemo` 中的代码。

构造函数同样没什么复杂的内容，我们把一个 `label`。一个 `textedit` 和两个 `button` 摆放到窗口中。这些代码已经能够很轻易的写出来了；然后进行了信号槽的连接。

```

1.     void ClipboardDemo::setClipboard()
2.     {
3.         QClipboard *board = QApplication::clipboard();
4.         board->setText("Text from Qt Application");
5.     }
6.
7.     void ClipboardDemo::getClipboard()
8.     {
9.         QClipboard *board = QApplication::clipboard();
10.        QString str = board->text();
11.        QMessageBox::information(NULL, "From clipboard", str);
12.    }

```

在 `slot` 函数中，我们使用 `QApplication::clipboard()` 函数访问到系统剪贴板。这个函数的返回值是 `QClipboard` 的指针。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** 默认就是支持 **Ctrl+C**, **Ctrl+V** 等快捷键操作的。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。

在 **X11** 系统中，鼠标中键(一般就是滚轮)可以支持剪贴操作的。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：

```
1. void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
2. {
3.     QClipboard *clipboard = QApplication::clipboard();
4.     if (event->button() == Qt::MidButton
5.         && clipboard->supportsSelection()) {
6.         QString text = clipboard->text(QClipboard::Selection);
7.         pasteText(text);
8.     }
9. }
```

这里的 **supportsSelection()** 在 **X11** 平台返回 **true**，其余平台都是返回 **false** 的。

另外，**QClipboard** 提供了 **dataChanged()** 信号，以便监听剪贴板数据变化。

Qt 学习之路(56): 二进制文件读写

今天开始进入 **Qt** 的另一个部分：文件读写，也就是 **IO**。文件读写在很多应用程序中都是需要的。**Qt** 通过 **QIODevice** 提供了 **IO** 的抽象，这种设备(device)具有读写字节块的能力。常用的 **IO** 读写的类包括以下几个：

QFile	访问本地文件系统或者嵌入资源
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
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

QProcess、**QTcpSocket**、**QUdpSocket** 和 **QSslSocket** 是顺序访问设备，它们的数据只能访问一遍，也就是说，你只能从第一个字节开始访问，直到最后一个字节。**QFile**、**QTemporaryFile** 和 **QBuffer** 是随机访问设备，你可以从任何位置访问任意次数，还可以使用 **QIODevice::seek()** 函数来重新定位文件指针。

在访问方式上，**Qt** 提供了两个更高级别的抽象：使用 **QDataStream** 进行二进制方式的访问和使用 **QTextStream** 进行文本方式的访问。这些类可以帮助我们控制字节顺序和文本编码，使程序员从这种问题中解脱出来。

QFile 对于访问独立的文件是非常方便的，无论是在文件系统中还是在应用程序的资源文件中。**Qt** 同样也提供了 **QDir** 和 **QFileInfo** 两个类，用于处理文件夹相关事务以及查看文件信息等。

这次我们先从二进制文件的读写说起。

以二进制格式访问数据的最简单的方式是实例化一个 **QFile** 对象，打开文件，然后使用 **QDataStream** 进行访问。**QDataStream** 提供了平台独立的访问数据格式的方法，这些数据格式包括标准的 C++ 类型，如 **int**、**double** 等；多种 **Qt** 类型，如 **QByteArray**、**QFont**、**QImage**、**QPixmap**、**QString** 和 **QVariant**，以及 **Qt** 的容器类，如 **QList<T>** 和 **QMap<K, T>**。先看如下的代码：

```
1.    QImage image("philip.png");
2.
3.    QMap<QString, QColor> map;
4.    map.insert("red", Qt::red);
5.    map.insert("green", Qt::green);
6.    map.insert("blue", Qt::blue);
7.
8.    QFile file("facts.dat");
9.    if (!file.open(QIODevice::WriteOnly)) {
10.        std::cerr << "Cannot open file for writing: "
11.                    << printable(file.errorString()) << std::endl;
12.        return;
13.    }
14.
15.    QDataStream out(&file);
16.    out.setVersion(QDataStream::Qt_4_3);
17.
18.    out << quint32(0x12345678) << image << map;
```

这里，我们首先创建了一个 **QImage** 对象，一个 **QMap<QString, QColor>**，然后使用 **QFile** 创建了一个名为 "facts.dat" 的文件，然后以只写方式打开。如果打开失败，直接 **return**；否则我们使用 **QFile** 的指针创建一个 **QDataStream** 对象，然后设置 **version**，这个我们以后再详细说明，最后就像 **std** 的 **cout** 一样，使用 **<<** 运算符输出结果。

0x12345678 成为“魔术数字”，这是二进制文件输出中经常使用的一种技术。我们定义的二进制格式通常具有一个这样的“魔术数字”，用于标志文件格式。例如，我们在文件最开始写入 **0x12345678**，在读取的时候首先检查这个数字是不是 **0x12345678**，如果不是的话，这就不是可识别格式，因此根本不需要去读取。一般二进制格式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 **0xCAFE BABE**(很 **Java** 的名字)，使用二进制查看器就可以查看。魔术数字是一个 32 位的无符号整数，因此我们使用 **quint32** 宏来得到一个平台无关的 32 位无符号整数。

在这段代码中我们使用了一个 **printable()** 宏，这个宏实际上是把 **QString** 对象转换成 **const char ***。注意到我们使用的是 C++ 标准错误输出 **cerr**，因此必须使用这个转换。当然，**QString::toString()** 函数也能够完成同样的操作。

读取的过程就很简单了，需要注意的是读取必须同写入的过程一一对应，即第一个写入 **q**

uint32 型的魔术数字，那么第一个读出的也必须是一个 quint32 格式的数据，如

```
1.     quint32 n;
2.     QImage image;
3.     QMap<QString, QColor> map;
4.
5.     QFile file("facts.dat");
6.     if (!file.open(QIODevice::ReadOnly)) {
7.         std::cerr << "Cannot open file for reading: "
8.                     << QString(file.errorString()) << std::endl;
9.         return;
10.    }
11.
12.    QDataStream in(&file);
13.    in.setVersion(QDataStream::Qt_4_3);
14.
15.    in >> n >> image >> map;
```

好了，数据读出了，拿着到处去用吧！

这个 **version** 是干什么用的呢？对于二进制的读写，随着 Qt 的版本升级，可能相同的内容有了不同的读写方式，比如可能由大端写入变成了小端写入等，这样的话旧版本 Qt 写入的内容就不能正确的读出，因此需要设定一个版本号。比如这里我们使用 **QDataStream::Qt_4_3**，意思是，我们使用 Qt 4.3 的方式写入数据。实际上，现在的最高版本号已经是 **QDataStream::Qt_4_6**。如果这么写，就是说，4.3 版本之前的 Qt 是不能保证正确读写文件内容的。那么，问题就来了：我们以硬编码的方式写入这个 **version**，岂不是不能使用最新版的 Qt 的读写了？

解决方法之一是，我们不仅仅写入一个魔术数字，同时写入这个文件的版本。例如：

```
1.     QFile file("file.xxx");
2.     file.open(QIODevice::WriteOnly);
3.     QDataStream out(&file);
4.
5.     // Write a header with a "magic number" and a version
6.     out << (quint32)0xA0B0C0D0;
7.     out << (quint32)123;
8.
9.     out.setVersion(QDataStream::Qt_4_0);
10.
11.    // Write the data
12.    out << lots_of_interesting_data;
```

这个 **file.xxx** 文件的版本号是 **123**。我们认为，如果版本号是 **123** 的话，则可以使用 **Qt_4_0** 版本读取。所以我们的读取代码就需要判断一下：

```
1.     QFile file("file.xxx");
2.     file.open(QIODevice::ReadOnly);
3.     QDataStream in(&file);
4.
```



```
5.     // Read and check the header
6.     quint32 magic;
7.     in >> magic;
8.     if (magic != 0xA0B0C0D0)
9.         return XXX_BAD_FILE_FORMAT;
10.
11.    // Read the version
12.    quint32 version;
13.    in >> version;
14.    if (version < 100)
15.        return XXX_BAD_FILE_TOO_OLD;
16.    if (version > 123)
17.        return XXX_BAD_FILE_TOO_NEW;
18.
19.    if (version <= 110)
20.        in.setVersion(QDataStream::Qt_3_2);
21.    else
22.        in.setVersion(QDataStream::Qt_4_0);
23.
24.    // Read the data
25.    in >> lots_of_interesting_data;
26.    if (version >= 120)
27.        in >> data_new_in_XXX_version_1_2;
28.    in >> other_interesting_data;
```

这样，我们就可以比较完美的处理二进制格式的数据读写了。