

## Qt 教程及软件

# Qt Creator 系列教程

为了使更多的 **Qt** 初学者能尽快入门 **Qt**，也为了 **Qt** 及 **Qt Creator** 的快速普及，我们花费大量精力写出了这一系列教程。虽然教程的知识可能很浅显，虽然教程的语言可能不规范，但是它却被数十万网友所认可。我们会将这一系列教程一直写下去，它将涉及 **Qt** 的方方面面。

本系列教程一直在更新... ...

版权声明：该系列教程全部原创，版权归 [www.yafeilinux.com](http://www.yafeilinux.com) 所有，您可以自由转载，但必须保留该声明，且不能用于商业用途。

进入 **qt** 帮助教程目录

进入开源项目目录

进入读者资料整理目录

蛋蛋整理（1466853244），欢迎学习交流，也希望有心人继续整理相关资料附在本文后边，并通知本人。

文档修订记录：

2011-01-09 整理了 [www.yafeilinux.com](http://www.yafeilinux.com) 的 qt creater 教程，编制了部分目录

整理人---蛋蛋 qq: 1466853244，进一步修订后请联系我，我更新相关记录

文档修订记录：

网站专题类：

专题类：

[方块游戏系列](#) [Qt 串口通信专题](#) [Qt 涂鸦板程序图文详细教程](#)

连载类：

- 一、Qt Creator 的安装和 hello world 程序的编写
- 二、Qt Creator 编写多窗口程序
- 三、Qt Creator 登录对话框
- 四、Qt Creator 添加菜单图标
- 五、Qt Creator 布局管理器的使用
- 六、Qt Creator 实现文本编辑
- 七、Qt Creator 实现文本查找
- 八、Qt Creator 实现状态栏显示
- 九、Qt Creator 中鼠标键盘事件的处理实现自定义鼠标指针
- 十、Qt Creator 中实现定时器和产生随机数
- 十一、Qt 2D 绘图（一）绘制简单图形
- 十二、Qt 2D 绘图（二）渐变填充
- 十三、Qt 2D 绘图（三）绘制文字
- 十四、Qt 2D 绘图（四）绘制路径
- 十五、Qt 2D 绘图（五）显示图片
- 十六、Qt 2D 绘图（六）坐标系统
- 十七、Qt 2D 绘图（七）Qt 坐标系统深入
- 十八、Qt 2D 绘图（八）涂鸦板

## [十九、Qt 2D 绘图（九）双缓冲绘图简介](#)

- 二十、Qt 2D 绘图（十）图形视图框架简介
- 二十一、Qt 数据库（一）简介
- 二十二、Qt 数据库（二）添加 MySQL 数据库驱动插件
- 二十三、Qt 数据库（三）利用 QSqlQuery 类执行 SQL 语句（一）
- 二十四、Qt 数据库（四）利用 QSqlQuery 类执行 SQL 语句（二）
- 二十五、Qt 数据库（五）QSqlQueryModel
- 二十六、Qt 数据库（六）QSqlTableModel
- 二十七、Qt 数据库（七）QSqlRelationalTableModel
- 二十八、Qt 数据库（八）XML（一）
- 二十九、Qt 数据库（九）XML（二）
- 三十、Qt 数据库（十）XML（三）
- 三十一、Qt 4.7.0 及 Qt Creator 2.0 beta 版安装全程图解
- 三十二、第一个 Qt Quick 程序（QML 程序）
- 三十三、体验 QML 演示程序
- 三十四、Qt Quick Designer 介绍
- 三十五、QML 组件
- 三十六、QML 项目之 Image 和 BorderImage
- 三十七、Flipable、Flickable 和状态与动画
- 三十八、QML 视图
- 三十九、QtDeclarative 模块
- 四十、使用 Nokia Qt SDK 开发 Symbian 和 Maemo 终端软件
- 四十一、Qt 网络（一）简介

- 四十二、Qt 网络(二) HTTP 编程
- 四十三、Qt 网络(三) FTP(一)
- 四十四、Qt 网络(四) FTP(二)
- 四十五、Qt 网络(五) 获取本机网络信息
- 四十六、Qt 网络(六) UDP
- 四十七、Qt 网络(七) TCP(一)
- 四十八、Qt 网络(八) TCP(二)

[目录预留](#)

系列教程会在 Qt Creator 2.1 正式推出后大幅度更新。。。。。 (预计 2011 年 1 月)

---

## Qt 系列开源软件

为加快初学者的入门和 **Qt** 的快速普及，我们在以后会陆续推出一系列开源软件。

**使用 Qt Creator，享受编程的快乐，让我们一起行动起来！**

第一款软件：

[多文档编辑器](#)

第二款软件：

[音乐播放器](#)

第三款软件：

[局域网聊天工具](#) (局域网聊天工具 07 月 17 日更新，添加文本智能编辑功能)

第四款软件：

[Wincom 串口调试软件 \(附有 Lincom 的源码\)](#)

期待... ...

这些文章建议有心人整理一下，附在本文后边，我留下了尽可能大的目录空间

---

## 网友的 Qt 相关文章

[Hello Qt \( 在 Linux 下编写运行 Qt 程序 \)](#)

---

### Qt 文章转载

免责声明：以下所有文章均转载自网友的博客或各相关网站，其版权归原作者所有。我们在所有的文章中都明确加入了原作者的版权声明和原文出处，如果有任何问题请与原作者联系，本站不对文章内容付任何责任。如果您不想让您的文章在本站转载，请联系我们  
[www.yafeilinux.com](http://www.yafeilinux.com)。

- 1.[在线教程-Qt 参考文档](#)
- 2.[Qt 学习之路文章列表](#)
- 3.[Qt 的 graphics View 框架](#)
- 4.[AT2440EVB II+WINCE5.0 板上跑 QT 程序](#)
- 5.[架设移动开发环境：Qt-wince 平台](#)
- 6.[OpenGL 贴图问题（转帖）](#)
- 7.[OpenGL 实现场景漫游\(转帖\)](#)
- 8.[上网本版 MeeGo 1.0 正式发布，核心软件平台支持 N900\(转载\)](#)
- 9.[MeeGo 为 Linux 带来魔法（转载）](#)
- 10.[Maemo ，青涩的机会，辽远的未来（转载）](#)
- 11.[Qt 编写 Mplayer 前端程序\(转载\)](#)

## 目录预留

## 目录预留

## 目录预留

## 目录预留











## 二、Qt Creator的安装和hello world程序的编写

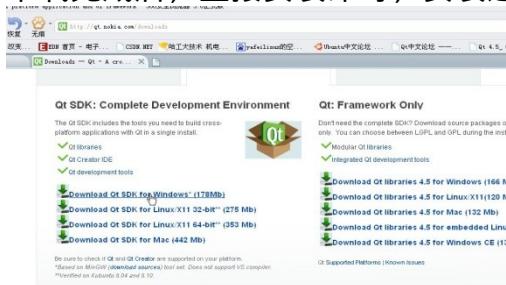
本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

我们这里讲述 windows 下的 Qt Creator，在 Linux 下基本相同。本文先讲述基本的下载、安装和最简单程序的编写，然后在附录里又讲解了两种其他的编写程序的方法。

1.首先到 Qt 的官方网站上下载 Qt Creator，这里我们下载 windows 版的。

下载地址：<http://qt.nokia.com/downloads> 如下图我们下载：Download Qt SDK for Windows\* (178Mb)

下载完成后，直接安装即可，安装过程中按默认设置即可。



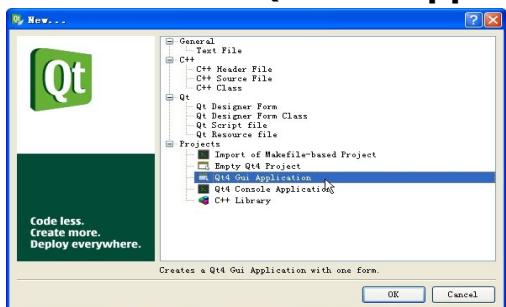
2.运行 Qt Creator，首先弹出的是欢迎界面，这里可以打开其自带的各种演示程序。



3.我们用 File->New 来新建工程。



4.这里我们选择 Qt4 Gui Application。



5.下面输入工程名和要保存到的文件夹路径。我们这里的工程名为 hello world。



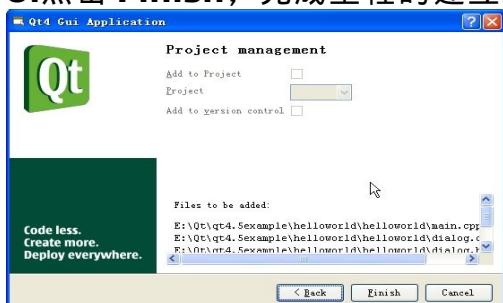
6.这时软件自动添加基本的头文件，因为这个程序我们不需要其他的功能，所以直接点击 **Next**。



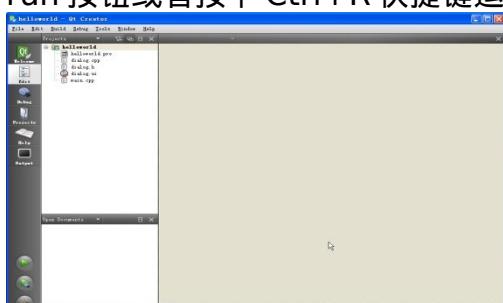
7.我们将 **base class** 选为 **QDialog** 对话框类。然后点击 **Next**。



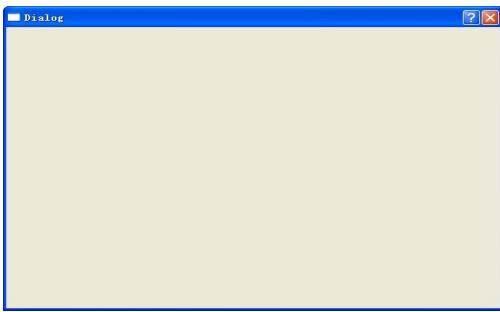
8.点击 **Finish**，完成工程的建立。



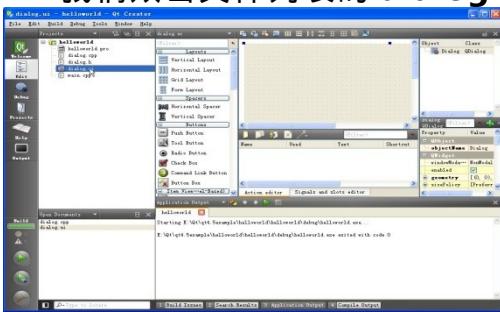
9.我们可以看见工程中的所有文件都出现在列表中了。我们可以直接按下下面的绿色的 run 按钮或者按下 Ctrl+R 快捷键运行程序。



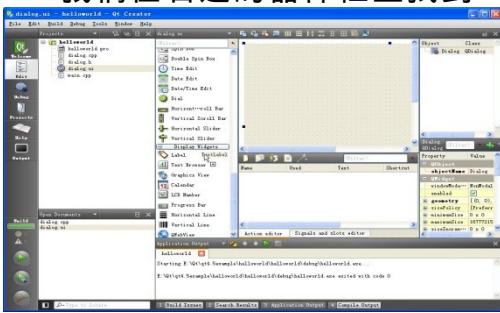
10.程序运行会出现空白的对话框，如下图。



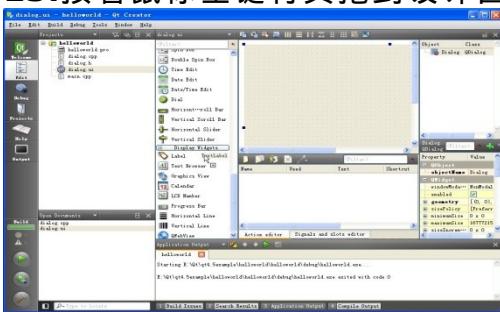
11. 我们双击文件列表的 **dialog.ui** 文件，便出现了下面所示的图形界面编辑界面。



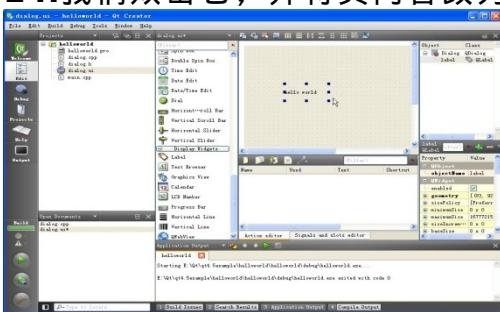
12. 我们在右边的器件栏里找到 **Label** 标签器件



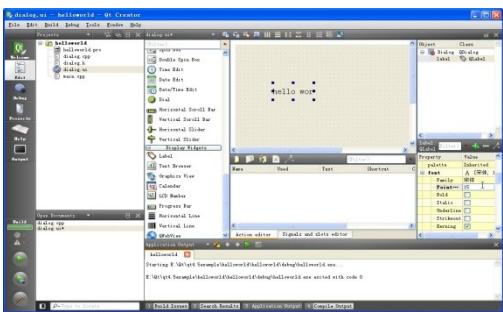
13. 按着鼠标左键将其拖到设计窗口上，如下图。



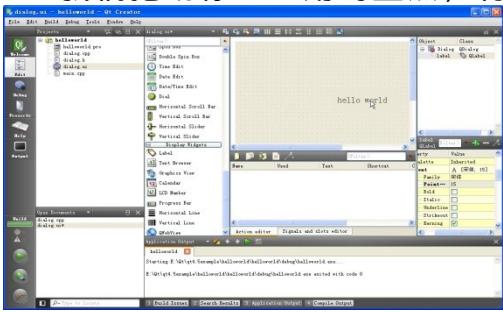
14. 我们双击它，并将其内容改为 **hello world**。



15. 我们在右下角的属性栏里将字体大小由 **9** 改为 **15**。



16. 我们拖动标签一角的蓝点，将全部文字显示出来。



17. 再次按下运行按钮，便会出现 **hello world**。



到这里 hello world 程序便完成了。

Qt Creator 编译的程序，在其工程文件夹下会有一个 debug 文件夹，其中有程序的.exe 可执行文件。但 Qt Creator 默认是用动态链接的，就是可执行程序在运行时需要相应的.dll 文件。我们点击生成的.exe 文件，首先可能显示“**没有找到 mingwm10.dll，因此这个应用程序未能启动。重新安装应用程序可能会修复此问题。**”表示缺少 mingwm10.dll 文件。

解决这个问题我们可以将相应的.dll 文件放到系统中。在 Qt Creator 的安装目录的 qt 文件下的 bin 文件夹下（我安装在了 D 盘，所以路径是 D:\Qt\2009.04\qt\bin），可以找到所有的相关.dll 文件。

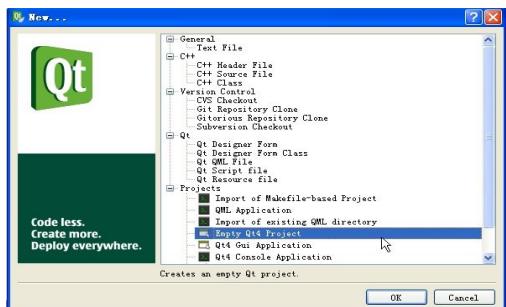
方法一：在这里找到 **mingwm10.dll** 文件，将其复制到 C:\WINDOWS\system 文件夹下即可。下面再提示缺少什么 dll 文件，都像这样解决就可以了。

方法二：将这些 dll 文件都与.exe 文件放到同一个文件夹下。不过这样每个.exe 文件都要放一次。

方法三：将 D:\Qt\2009.04\qt\bin 加入系统 Path 环境变量。右击我的电脑->属性->高级->环境变量->在系统变量列表中找到 Path, 将路径加入其中即可。

## 附 1：用纯源码编写。

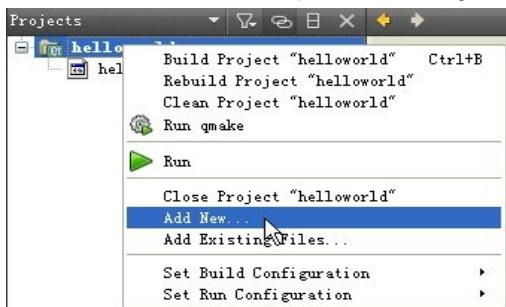
### 1. 新建空的 Qt4 工程。



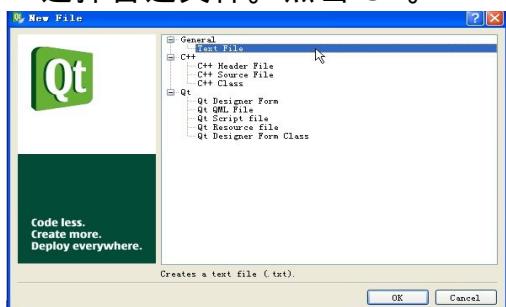
2. 工程名为 **hello world**, 并选择工程保存路径 ( 提示: 路径中不能有中文 )。



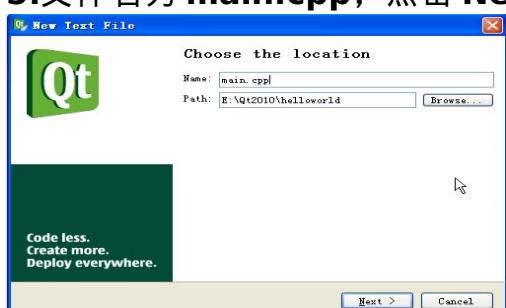
3. 在新建好的工程中添加文件。右击工程文件夹，弹出的菜单中选择 **Add New...**。



4. 选择普通文件。点击 **Ok**。



5. 文件名为 **main.cpp**, 点击 **Next >** 进入下一步。



6. 这里自动将这个文件添加到了新建的工程中。保持默认设置，点击完成。



## 7. 在 main.cpp 文件中添加代码。

```
#include <QtGui>

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

8. 这时点击运行，程序执行了，但看不到效果，因为程序里什么也没做。我们点击信息框右上角的红色方块，停止程序运行。



9. 我们再更改代码。添加一个对话框对象。

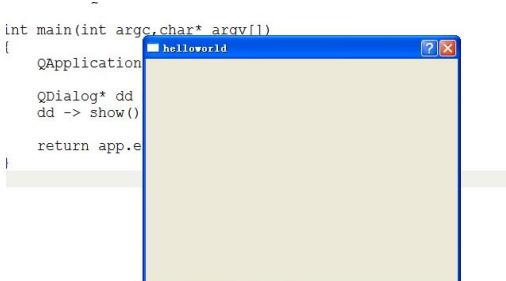
```
#include <QtGui>

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

    QDialog* dd = new QDialog();
    dd->show();

    return app.exec();
}
```

10. 运行效果如下。



11. 我们更改代码如下，在对话框上添加一个标签对象，并显示 **hello world**。

```

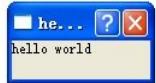
#include <QtGui>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QDialog* dd = new QDialog();
    QLabel* label = new QLabel(dd);
    label->setText("hello world");
    dd->show();
    return app.exec();
}

ACTION app(argc, argv);

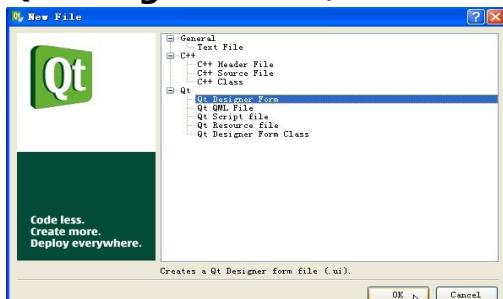
* dd = new QDialog();
label = new QLabel(dd);
setText("hello world");
how();
app.exec();

```

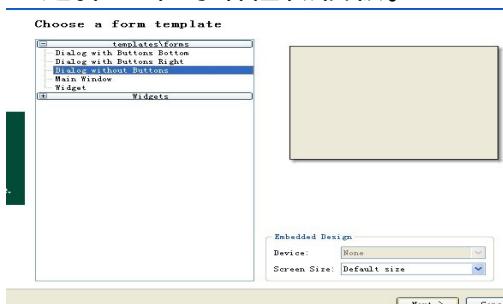


## 附 2：利用 ui 文件。

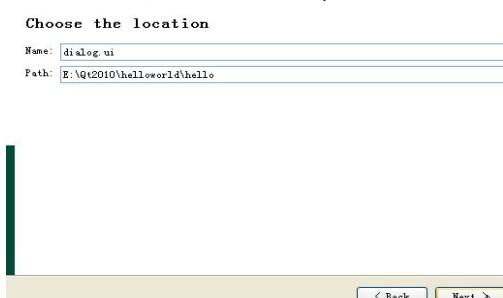
1. 建立新的空工程，这里的工程名为 **hello**，建立好工程后，添加新文件。这里添加 **Qt Designer Form**。



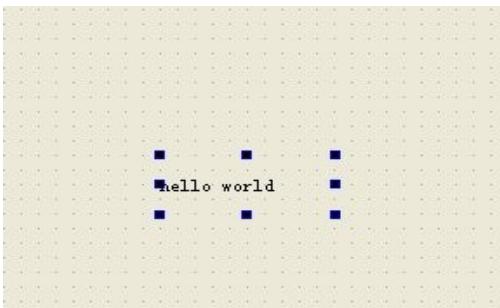
2. 选择一个对话框做模板。



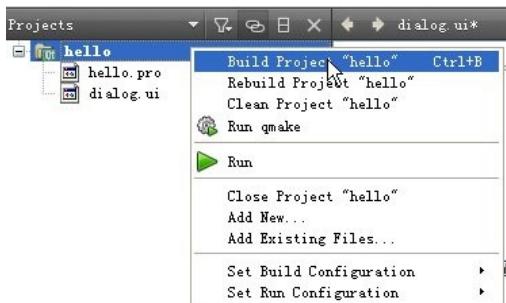
3. 你可以更改文件名，我们这里使用默认设置。



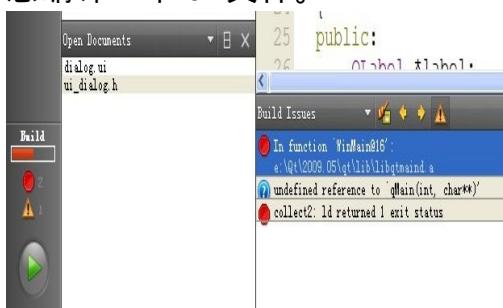
4. 在新建好的框口上添加一个标签，并更改文本为 **hello world**。



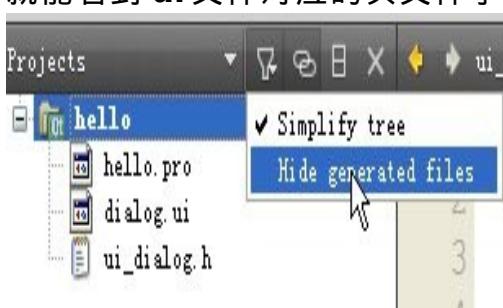
5.在工程文件夹上点击右键，弹出的菜单中选择第一项编译工程。



6.因为还没有写主函数，所以现在编译文件会出现错误，不过没关系，因为我们只是想编译一下 ui 文件。



7.点击 这个图标，去掉弹出的菜单中第二项前的对勾，显示隐藏的文件。这时你就能看到 ui 文件对应的头文件了。

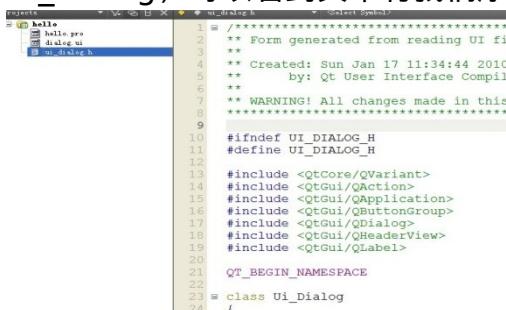


而如果去掉菜单中的第一项前的对勾，列表中的文件就会分类显示，如图



8.ui 文件对应的.h 文件默认为 ui\_dialog.h (例如 form.ui 对应 ui\_form.h)。

其中是设计器设计的窗口的对应代码。我们这里的.h文件是最简单的，其类名为Ui\_Dialog，可以看到其中有我们添加的标签对象。



```
1 // ****
2 ** Form generated from reading UI fi
3 **
4 ** Created: Sun Jan 17 11:34:44 2010
5 ** by: Qt User Interface Compil
6 **
7 ** WARNING! All changes made in this
8 *****
```

```
10 #ifndef UI_DIALOG_H
11 #define UI_DIALOG_H
```

```
13 #include <QtCore/QVariant>
14 #include <QtGui/QAction>
15 #include <QtGui/QApplication>
16 #include <QtGui/QButtonGroup>
17 #include <QtGui/QDialog>
18 #include <QtGui/QHeaderView>
19 #include <QtGui.QLabel>
```

```
21 QT_BEGIN_NAMESPACE
22
23 class Ui_Dialog
24 {
```

9.在这个类里有一个 **setupUi** 函数，我们就是利用这个函数来使用设计好的窗口的。

```
class Ui_Dialog
{
public:
    QLabel *label;

void setupUi(QDialog *Dialog)
{
    if (Dialog->objectName().isEmpty())
        Dialog->setObjectName(QString::fromUtf8("Dialog"));
    Dialog->resize(411, 141);
    label = new QLabel(Dialog);
    label->setObjectName(QString::fromUtf8("label"));
    label->setGeometry(QRect(130, 140, 101, 41));

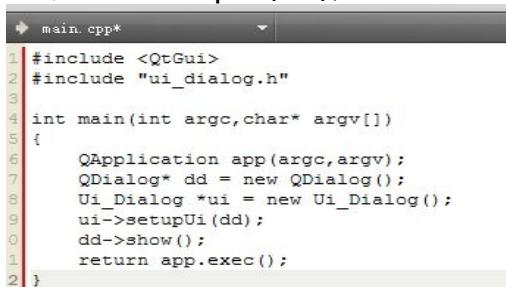
    retranslateUi(Dialog);

    QMetaObject::connectSlotsByName(Dialog);
} // setupUi

void retranslateUi(QDialog *Dialog)
{
    Dialog->setWindowTitle(QApplication::translate("Dialog", "Dialog", 0, QApplication::UnicodeUTF8));
    label->setText(QApplication::translate("Dialog", "Hello world", 0, QApplication::UnicodeUTF8));
} // retranslateUi
};
```

10.我们添加 **main.cpp** 文件，并更改内容如下。

其中 **ui->setupUi(dd);**一句就是将设计的窗口应用到新建的窗口对象上。



```
1 #include <QtGui>
2 #include "ui_dialog.h"

3 int main(int argc, char* argv[])
4 {
    QApplication app(argc, argv);
    QDialog* dd = new QDialog();
    Ui_Dialog *ui = new Ui_Dialog();
    ui->setupUi(dd);
    dd->show();
    return app.exec();
}
```

11.这时运行程序，效果如下。



```
1 // -----
2 L_Dialog *ui = new L_Dialog();
3 i->setupUi(dd);
4 i->show();
5 return app.exec();
```

在这篇文章中我们一共讲述了三种方法写 **Hello World** 程序，其实也就是两种，一种用设计器，一种全部用代码生成，其实他们是等效的。因为我们已经看到，就算是设计器生成，其实也是写了一个对应的 **ui.h** 文件，只不过这个文件是自动生成的，不用我们自己写而已。

分类：[Qt系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 14,156 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 二、Qt Creator 编写多窗口程序

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

实现功能：

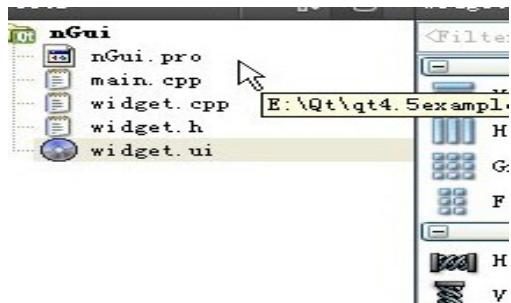
程序开始出现一个对话框，按下按钮后便能进入主窗口，如果直接关闭这个对话框，便不能进入主窗口，整个程序也将退出。当进入主窗口后，我们按下按钮，会弹出一个对话框，无论如何关闭这个对话框，都会回到主窗口。

实现原理：

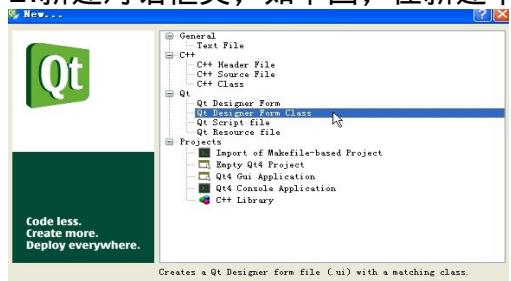
程序里我们先建立一个主工程，作为主界面，然后再建立一个对话框类，将其加入工程中，然后在程序中调用自己新建的对话框类来实现多窗口。

实现过程：

1.首先新建Qt4 Gui Application 工程，工程名为 nGui，Base class 选为 QWidget。建立好后工程文件列表如下图。



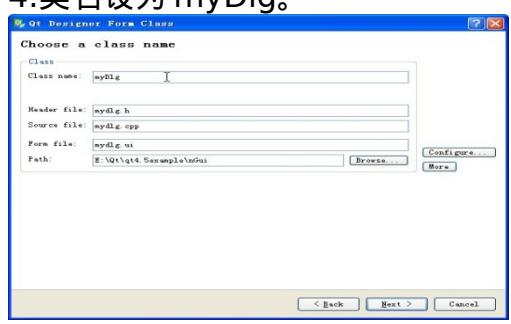
2.新建对话框类，如下图，在新建中，选择 Qt Designer Form Class。



3.选择 Dialog without Buttons。



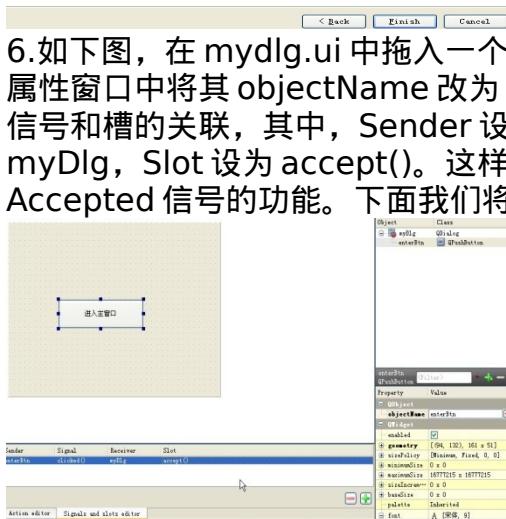
4.类名设为 myDlg。



5.点击 Finish 完成。注意这里已经默认将其加入到了我们刚建的工程中了。



E:\Qt\Qt4.5\example\gui\mydlg.h  
E:\Qt\Qt4.5\example\gui\mydlg.cpp  
E:\Qt\Qt4.5\example\gui\mydlg.ui



6.如下图，在 mydlg.ui 中拖入一个 Push Button，将其上的文本改为“进入主窗口”，在其属性窗口中将其 objectName 改为 enterBtn，在下面的 Signals and slots editor 中进行信号和槽的关联，其中，Sender 设为 enterBtn，Signal 设为 clicked()，Receive 设为 myDlg，Slot 设为 accept()。这样就实现了单击这个按钮使这个对话框关闭并发出 Accepted 信号的功能。下面我们将利用这个信号。

7.修改主函数 main.cpp，如下：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "mydlg.h"      //加入头文件
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    myDlg my1;      //建立自己新建的类的对象 my1
    if(my1.exec()==QDialog::Accepted)  //利用 Accepted 信号判断 enterBtn 是否被按下
    {
        w.show();      //如果被按下，显示主窗口
        return a.exec();  //程序一直执行，直到主窗口关闭
    }
    else return 0;  //如果没被按下，则不会进入主窗口，整个程序结束运行
}
```

主函数必须这么写，才能完成所要的功能。  
如果主函数写成下面这样：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "mydlg.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    myDlg my1;
```

```

if(my1.exec()==QDialog::Accepted)
{
Widget w;
    w.show();
}
return a.exec();
}

```

这样，因为 w 是在 if 语句里定义的，所以当 if 语句执行完后它就无效了。这样导致的后果就是，按下 enterBtn 后，主界面窗口一闪就没了。如果此时对程序改动了，再次点击运行时，就会出现 **error: collect2: Id returned 1 exit status** 的错误。这是因为虽然主窗口没有显示，但它只是隐藏了，程序并没有结束，而是在后台运行。所以这时改动程序，再运行时便会出错。你可以按下调试栏上面的红色 Stop 停止按钮来停止程序运行。你也可以在 windows 任务管理器的进程中将该进程结束，而后再次运行就没问题了，当然先关闭 Qt Creator，而后再重新打开，这样也能解决问题。

如果把程序改为这样：

```

#include <QtGui/QApplication>
#include "widget.h"
#include "mydlg.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    myDlg my1;
    Widget w;
    if(my1.exec()==QDialog::Accepted)
    {
        w.show();
    }
    return a.exec();
}

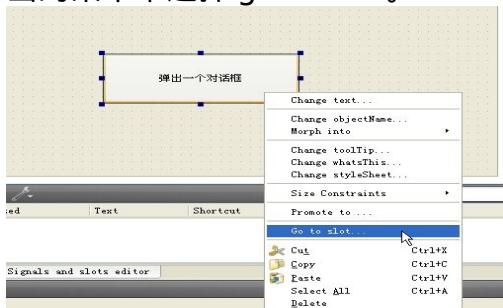
```

这样虽然解决了上面主窗口一闪而过的问题，但是，如果在 my1 对话框出现的时候不点 enterBtn，而是直接关闭对话框，那么此时整个程序应该结束执行，但是事实是这样的吗？如果你此时对程序进行了改动，再次按下 run 按钮，你会发现又出现了 **error: collect2: Id returned 1 exit status** 的错误，这说明程序并没有结束，我们可以打开 windows 任务管理器，可以看到我们的程序仍在执行。

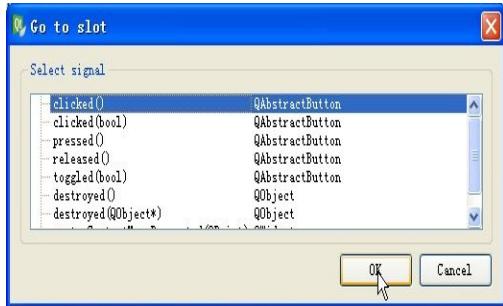
因为 return a.exec(); 一句表示只要主窗口界面不退出，那么程序就会一直执行。所以只有用第一种方法，将该语句也放到 if 语句中，而在 else 语句中用 **else return 0;**，这样如果 **enterBtn** 没有被按下，那么程序就会结束了。

到这里，我们就实现了一个界面结束执行，然后弹出另一个界面的程序。下面我们在主窗口上加一个按钮，按下该按钮，弹出一个对话框，但这个对话框关闭，不会使主窗口关闭。

8.如下图，在主窗口加入按钮，显示文本为“弹出一个对话框”，在其上点击鼠标右键，在弹出的菜单中选择 go to slot。



9.我们选择单击事件 clicked()。



10.我们在弹出的槽函数中添加一句：

```
my2.show();
```

my2 为我们新建对话框类的另一个对象，但是 my2 我们还没有定义，所以在 widget.h 文件中添加相应代码，如下，先加入头文件，再加入 my2 的定义语句，这里我们将其放到 private 里，因为一般的函数都放在 public 里，而变量都放在 private 里。

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QtGui/QWidget>
#include "mydlg.h" //包含头文件
namespace Ui
{
class Widget;
}
class Widget : public QWidget
{
Q_OBJECT
public:
Widget(QWidget *parent = 0);
~Widget();
private:
Ui::Widget *ui;
myDlg my2; //对 my2 进行定义
private slots:
void on_pushButton_clicked();
};
#endif // WIDGET_H
```

到这里，再运行程序，便能完成我们实验要求的功能了。整个程序里，我们用两种方法实现了信号和槽函数的关联，第一个按钮我们直接在设计器中实现其关联；第二个按钮我们自己写了槽函数语句，其实图形的设计与直接写代码效果是一样的。

这个程序里我们实现了两类窗口打开的方式，一个是自身消失而后打开另一个窗口，一个是打开另一个窗口而自身不消失。可以看到他们实现的方法是不同的。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 9,872 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 三、Qt Creator 登录对话框

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

实现功能：

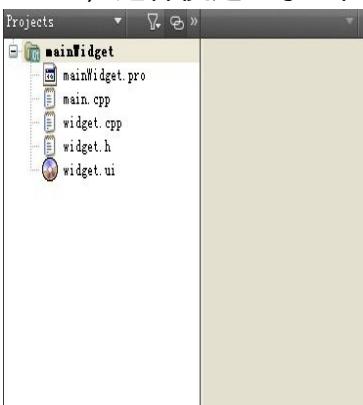
在弹出对话框中填写用户名和密码，按下登录按钮，如果用户名和密码均正确则进入主窗口，如果有错则弹出警告对话框。

实现原理：

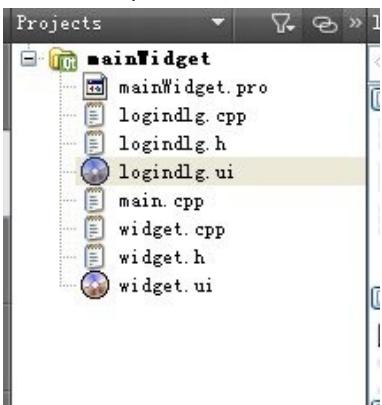
通过上节的多窗口原理实现由登录对话框进入主窗口，而用户名和密码可以用 if 语句进行判断。

实现过程：

1. 先新建 Qt4 Gui Application 工程，工程名为 mainWidget，选用 QWidget 作为 Base class，这样便建立了主窗口。文件列表如下：



2. 然后新建一个 Qt Designer Form Class 类，类名为 loginDlg，选用 Dialog without Buttons，将其加入上面的工程中。文件列表如下：



3. 在 logindlg.ui 中设计下面的界面：行输入框为 Line Edit。其中用户名后面的输入框在属性中设置其 object Name 为 usrLineEdit，密码后面的输入框为 pwdLineEdit，登录按钮为 loginBtn，退出按钮为 exitBtn。



4. 将 exitBtn 的单击后效果设为退出程序，关联如下：

| Sender  | Signal    | Receiver | Slot    |
|---------|-----------|----------|---------|
| exitBtn | clicked() | loginDlg | close() |

5.右击登录按钮选择 go to slot，再选择 clicked(),然后进入其单击事件的槽函数，写入一句

```
void loginDlg::on_loginBtn_clicked()
{
    accept();
}
```

6.改写 main.cpp:

```
#include <QtGui/QApplication>
#include "widget.h"
#include "logindlg.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    loginDlg login;
    if(login.exec()==QDialog::Accepted)
    {
        w.show();
        return a.exec();
    }
    else return 0;
}
```

7.这时执行程序，可实现按下登录按钮进入主窗口，按下退出按钮退出程序。

8.添加用户名密码判断功能。将登陆按钮的槽函数改为：

```
void loginDlg::on_loginBtn_clicked()
{
if(m_ui->usrLineEdit->text()==tr("qt")&&m_ui->pwdLineEdit-
>text()==tr("123456"))
//判断用户名和密码是否正确
accept();
else{
QMessageBox::warning(this,tr("Warning"),tr("user name or password
error!"),QMessageBox::Yes);
//如果不正确，弹出警告对话框
}
}
```

并在 logindlg.cpp 中加入 `#include <QtGui>` 的头文件。如果不加这个头文件， QMessageBox 类不可用。

(说明：由于版本原因，现在的程序默认生成的 **ui** 类对象可能是 **ui**，而不是 **m\_ui**，请到 **logindlg.h** 中查看。)

9.这时再执行程序，输入用户名为 **qt**，密码为 **123456**，按登录按钮便能进入主窗口了，如果输入错了，就会弹出警告对话框。



如果输入错误，便会弹出警告提示框：



10.在 logindlg.cpp 的 loginDlg 类构造函数里，添上初始化语句，使密码显示为小黑点。

```
loginDlg::loginDlg(QWidget *parent) :  
    QDialog(parent),  
    m_ui(new Ui::loginDlg)  
{  
    m_ui->setupUi(this);  
    m_ui->pwdLineEdit->setEchoMode(QLineEdit::Password);  
}
```

效果如下：



11.如果输入如下图中的用户名，在用户名前不小心加上了一些空格，结果程序按错误的用户名对待了。



我们可以更改 if 判断语句，使这样的输入也算正确。

```
void loginDlg::on_loginBtn_clicked()
{
if(m_ui->usrLineEdit->text().trimmed()==tr("qt")&&m_ui->pwdLineEdit-
>text()==tr("123456"))
accept();
else{
QMessageBox::warning(this,tr("Warning"),tr("user name or password
error!"),QMessageBox::Yes);
}
}
```

加入的这个函数的作用就是移除字符串开头和结尾的空白字符。

12.最后，如果输入错误了，重新回到登录对话框时，我们希望可以使用用户名和密码框清空并且光标自动跳转到用户名输入框，最终的登录按钮的单击事件的槽函数如下：

```
void loginDlg::on_loginBtn_clicked()
{
if(m_ui->usrLineEdit->text().trimmed()==tr("qt")&&m_ui->pwdLineEdit-
>text()==tr("123456"))
//判断用户名和密码是否正确
accept();
else{
QMessageBox::warning(this,tr("Warning"),tr("user name or password
error!"),QMessageBox::Yes);
//如果不正确，弹出警告对话框
m_ui->usrLineEdit->clear();//清空用户名输入框
m_ui->pwdLineEdit->clear();//清空密码输入框
m_ui->usrLineEdit->setFocus();//将光标转到用户名输入框
}
}
```

最终的 loginDlg.cpp 文件如下图：

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 7,049 views  
Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 四、Qt Creator 添加菜单图标

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

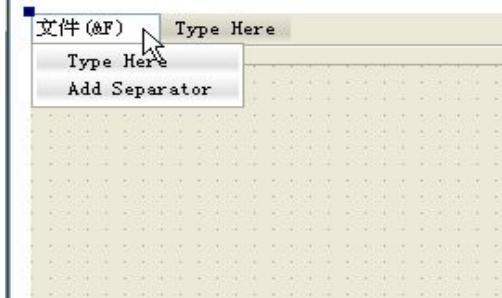
在下面的几节，我们讲述 Qt 的 MainWindow 主窗口部件。这一节只讲述怎样在其上的菜单栏里添加菜单和图标。

1. 新建 Qt4 Gui Application 工程，将工程命名为 MainWindow，其他选项默认即可。

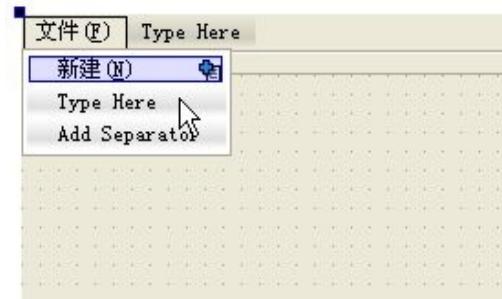
生成的窗口界面如下图。其中最上面的为菜单栏。



2. 我们在 Type Here 那里双击，并输入“文件(&F)”，这样便可将其文件菜单的快捷键设为 Alt+F。（注意括号最好用英文半角输入，这样看着美观）



3. 输入完按下 Enter 键确认即可，然后在子菜单中加入“新建(&N)”，确定后，效果如下图。



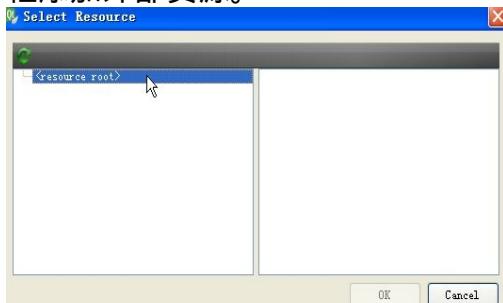
4. 我们在下面的动作编辑窗口可以看到新加的“新建”菜单。



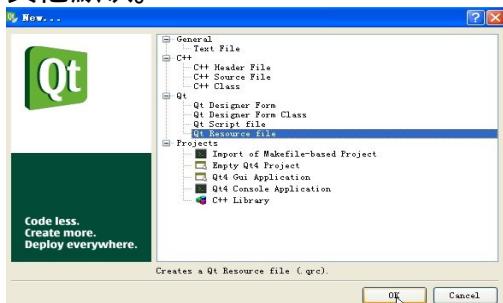
5. 双击这一条，可打开它的编辑对话框。我们看到 Icon 项，这里可以更改“新建”菜单的图标。



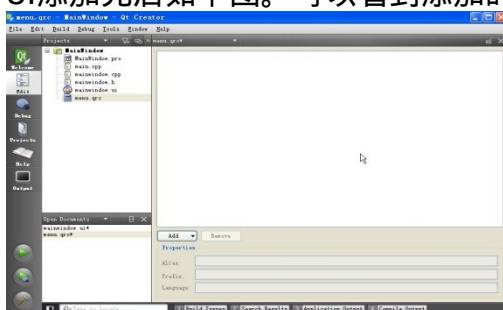
6. 我们点击后面的...号，进入资源选择器，但现在这里面是空的。所以下面我们需要给该工程添加外部资源。



7. 添加资源有两种方法。一种是直接添加系统提供的资源文件，然后选择所需图标。另一种是自己写资源文件。我们主要介绍第一种。新建 Qt Resources file，将它命名为 menu。其他默认。



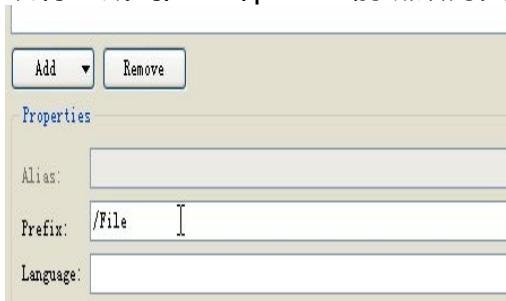
8. 添加完后如下图。可以看到添加的文件为 menu.qrc。



9. 我们最好先在工程文件夹里新建一个文件夹，如 images，然后将需要的图标文件放到其中。



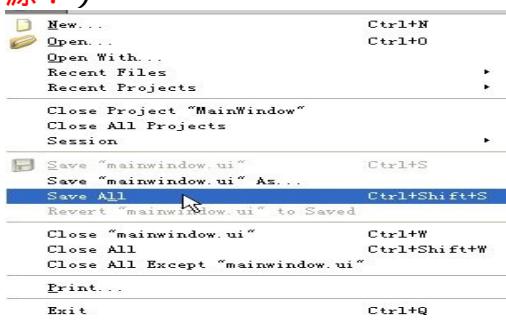
10. 在 Qt Creator 的 menu.qrc 文件中，我们点击 Add 下拉框，选择 Add Prefix。我们可以将生成的/new/prefix 前缀改为其他名字，如/File。



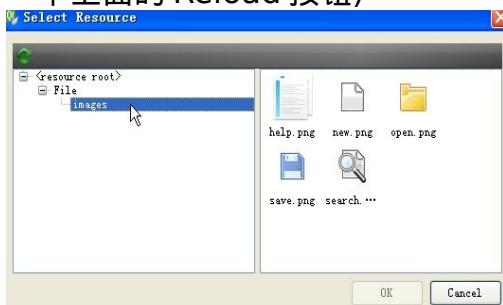
11. 然后再选择 Add 下拉框，选择 Add Files。再弹出的对话框中，我们到新建的 images 文件夹下，将里面的图标文件全部添加过来。



12. 添加完成后，我们在 Qt Creator 的 File 菜单里选择 Save All 选项，保存所做的更改。  
**(注意：一定要先保存刚才的 qrc 文件，不然在资源管理器中可能看不见自己添加的资源！)**



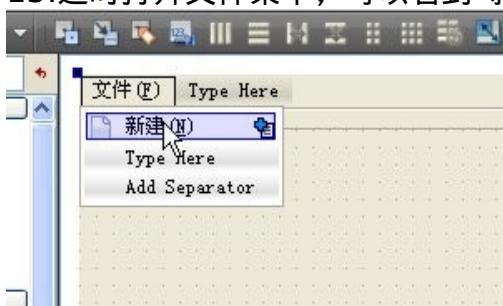
13.这时再打开资源选择器，可以看到我们的图标都在这里了。(注意：如果不显示，可以按一下上面的 Reload 按钮)



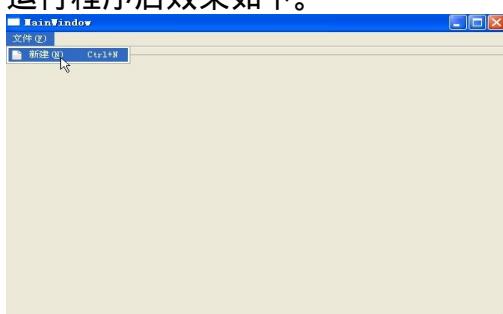
14.我们将 new.png 作为“新建”菜单的图标，然后点击 Shortcut，并按下 Ctrl+N，便能将 Ctrl+N 作为“新建”菜单的快捷键。



15.这时打开文件菜单，可以看到“新建”菜单已经有图标了。



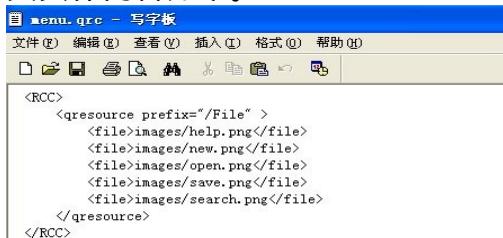
运行程序后效果如下。



16.我们在工程文件夹下查看建立的 menu.qrc 文件，可以用写字板将它打开。



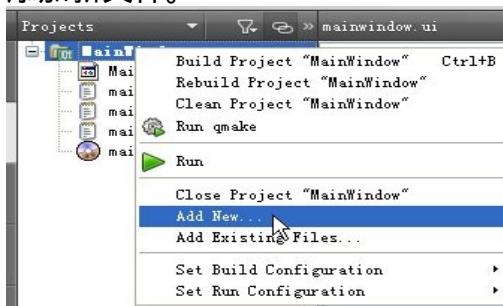
其具体内容如下。



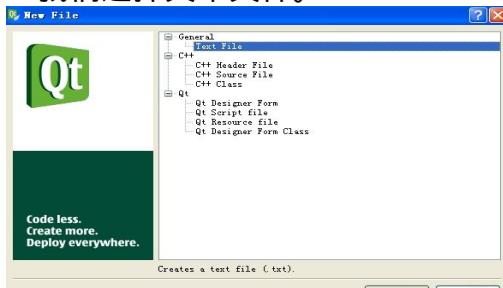
```
<RCC>
<qresource prefix="/File" >
<file>images/help.png</file>
<file>images/new.png</file>
<file>images/open.png</file>
<file>images/save.png</file>
<file>images/search.png</file>
</qresource>
</RCC>
```

附：第二种添加资源文件的方法。

1.首先右击工程文件夹，在弹出的菜单中选择 Add New，添加新文件。也可以用 File 中的添加新文件。



2.我们选择文本文件。



3.将文件名设置为 menu.qrc。



4.添加好文件后将其内容修改如下。可以看到就是用第一种方法生成的 menu.qrc 文件的内容。



```
1 <RCC>
2   <qresource>
3     <file>images/new.png</file>
4     <file>images/open.png</file>
5   </qresource>
6 </RCC>
7 |
```

5.保存文件后，在资源管理器中可以看到添加的图标文件。

## 五、Qt Creator 布局管理器的使用

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

上篇讲解了如何在 Qt Creator 中添加资源文件，并且为菜单添加了图标。这次我们先对那个界面进行一些完善，然后讲解一些布局管理器的知识。

首先对菜单进行完善。

1.我们在上一次的基础上再加入一些常用菜单。

“文件”的子菜单如下图。中间的分割线可以点击 Add Separator 添加。



“编辑”子菜单的内容如下。



“帮助”子菜单的内容如下。



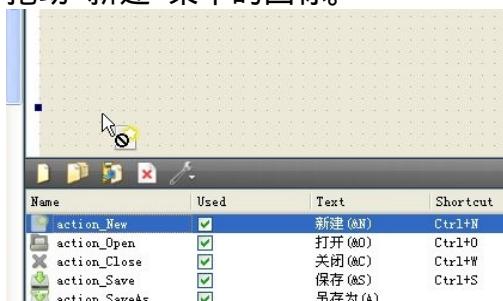
2.我们在动作编辑器中对各个菜单的属性进行设置。

如下图。

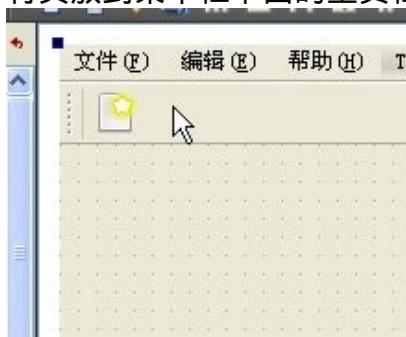
| Name          | Used | Text    | Shortcut | Checkable | ToolTip |
|---------------|------|---------|----------|-----------|---------|
| action_New    | ✓    | 新建 (N)  | Ctrl+N   | □         | 新建 (N)  |
| action_Open   | ✓    | 打开 (O)  | Ctrl+O   | □         | 打开 (O)  |
| action_Close  | ✓    | 关闭 (C)  | Ctrl+W   | □         | 关闭 (C)  |
| action_Save   | ✓    | 保存 (S)  | Ctrl+S   | □         | 保存 (S)  |
| action_SaveAs | ✓    | 另存为 (A) |          | □         | 另存为 (A) |
| action_Quit   | ✓    | 退出 (X)  | Ctrl+Q   | □         | 退出 (X)  |
| action_Undo   | ✓    | 撤销 (Z)  | Ctrl+Z   | □         | 撤销 (Z)  |
| action_Cut    | ✓    | 剪切 (X)  | Ctrl+X   | □         | 剪切 (X)  |
| action_Copy   | ✓    | 复制 (C)  | Ctrl+C   | □         | 复制 (C)  |
| action_Paste  | ✓    | 粘贴 (V)  | Ctrl+V   | □         | 粘贴 (V)  |
| action_Find   | ✓    | 查找 (F)  | Ctrl+F   | □         | 查找 (F)  |
| action_Help   | ✓    | 版本说明    | F1       | □         | 版本说明    |

3. 我们拖动“新建”菜单的图标，将其放到工具栏里。

拖动“新建”菜单的图标。



将其放到菜单栏下面的工具栏里。



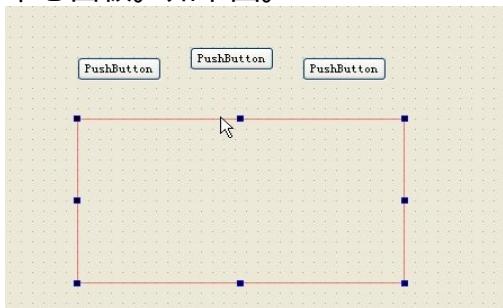
4. 我们再添加其他几个图标。使用 Append Separator 可以添加分割线。



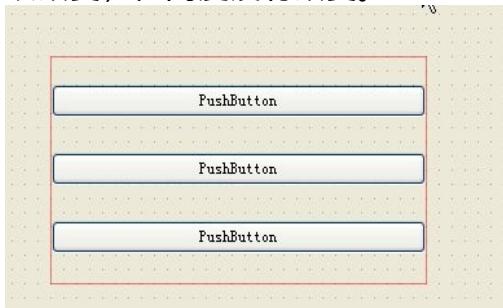
5. 最终效果如下。如果需要删除图标，可以在图标上点击右键选择 Remove action 即可。  
下面简述一下布局管理器。

( 这里主要以垂直布局管理器进行讲解，其他类型管理器用法与之相同，其效果可自己验证。 )

1. 在左边的器件栏里拖入三个PushButton 和一个Vertical Layout ( 垂直布局管理器 ) 到中心面板。如下图。



2. 将这三个按钮放入垂直布局管理器，效果如下。可以看到按钮垂直方向排列，并且宽度可以改变，但高度没有改变。

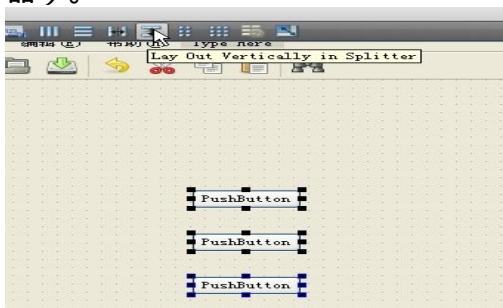


3. 我们将布局管理器整体选中，按下上面工具栏的 Break Layout 按钮，便可取消布局管理器。（我们当然也可以先将按钮移出，再按下 Delete 键将布局管理器删除。）

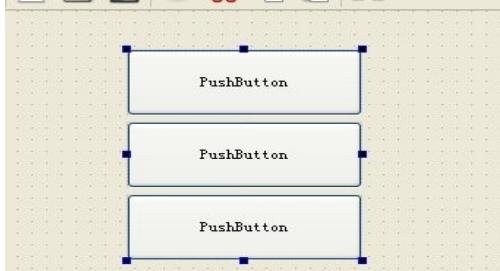


4. 下面我们改用分裂器部件 ( QSplitter ) 。

先将三个按钮同时选中，再按下上面工具栏的 Lay Out Vertically in Splitter ( 垂直分裂器 ) 。

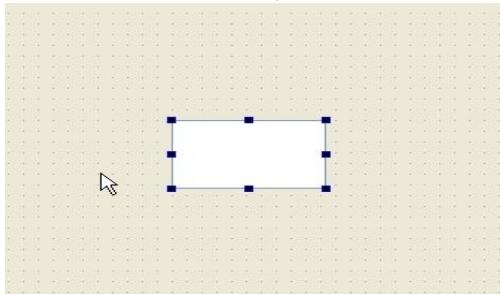


效果如下图。可以看到按钮的大小可以随之改动。这也就是分裂器和布局管理器的分别。

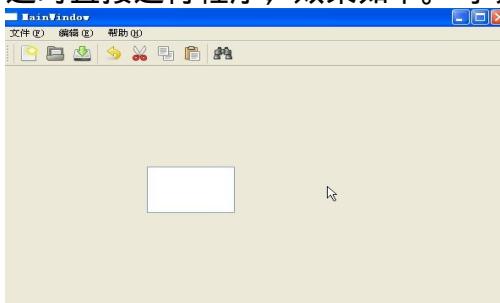


5.其实布局管理器不但能控制器件的布局，还有个很重要的用途是，它能使器件的大小随着窗口大小的改变而改变。

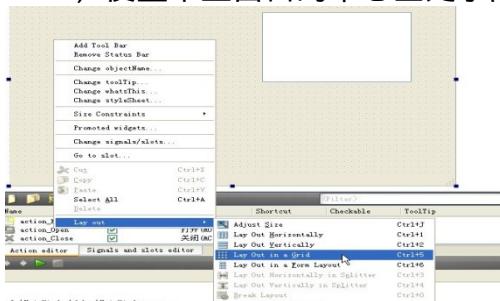
我们先在主窗口的中心拖入一个文本编辑器 Text Edit。



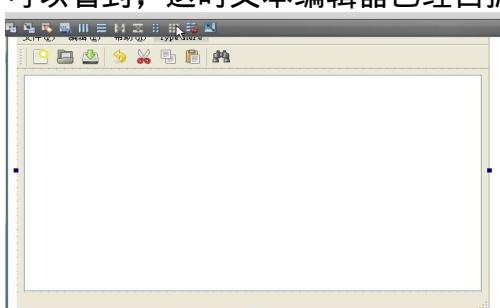
这时直接运行程序，效果如下。可以看到它的大小和位置不会随着窗口改变。



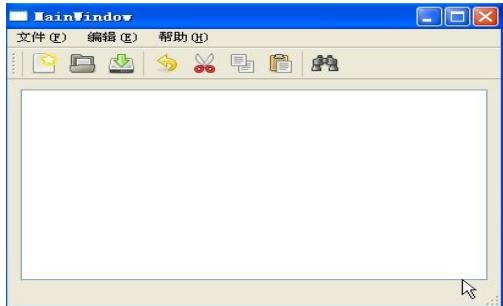
下面我们选中主窗口部件，然后在空白处点击鼠标右键，选择 Layout->Lay Out in a Grid，使整个主窗口的中心区处于网格布局管理器中。



可以看到，这时文本编辑器已经占据了整个主窗口的中心区。



运行一下程序，可以看到无论怎样拉伸窗口，文本编辑框的大小都会随之改变。



我们在这里一共讲述了三种使用布局管理器的方法，一种是去器件栏添加，一种是用工具栏的快捷图标，还有一种是使用鼠标右键的选项。

程序中用到的图标是我从 Ubuntu 中复制的，可以到[资源下载](#)页面下载到。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 5,487 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 六、Qt Creator 实现文本编辑

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

前面已经将界面做好了，这里我们为其添加代码，实现文本编辑的功能。

首先实现新建文件，文件保存，和文件另存为的功能。

（我们先将上次的工程文件夹进行备份，然后再对其进行修改。在写较大的程序时，经常对源文件进行备份，是个很好的习惯。）

在开始正式写程序之前，我们先要考虑一下整个流程。因为我们要写记事本一样的软件，所以最好先打开 windows 中的记事本，进行一些简单的操作，然后考虑怎样去实现这些功能。再者，再强大的软件，它的功能也是一个一个加上去的，不要设想一下子写出所有的功能。我们这里先实现新建文件，保存文件，和文件另存为三个功能，是因为它们联系很紧，而且这三个功能总的代码量也不是很大。

因为三个功能之间的关系并不复杂，所以我们这里便不再画流程图，而只是简单描述一下。

新建文件，那么如果有正在编辑的文件，是否需要保存呢？

如果需要进行保存，那这个文件以前保存过吗？如果没有保存过，就应该先将其另存为。

下面开始按这些关系写程序。

**1. 打开 Qt Creator，在 File 菜单中选择 Open，然后在工程文件夹中打开 MainWindow.pro 工程文件。**

先在 main.cpp 文件中加入以下语句，让程序中可以使用中文。

在其中加入 #include <QTextCodec> 头文件包含，再在主函数中加入下面一行：

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

这样在程序中使用中文，便能在运行时显示出来了。更改后文件如下图。

```
main.cpp*  main(int, char **[])
1 #include <QtGui/QApplication>
2 #include <QTextCodec> //加入头文件
3 #include "mainwindow.h"
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8     QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
9     //实现中文显示
10    MainWindow w;
11    w.show();
12    return a.exec();
13 }
```

**2. 在 mainwindow.h 文件中的 private 下加入以下语句。**

bool isSaved; //为 true 时标志文件已经保存，为 false 时标志文件尚未保存

QString curFile; //保存当前文件的文件名

void do\_file\_New(); //新建文件

void do\_file\_SaveOrNot(); //修改过的文件是否保存

void do\_file\_Save(); //保存文件

void do\_file\_SaveAs(); //文件另存为

bool saveFile(const QString& fileName); //存储文件

这些是变量和函数的声明。其中 isSaved 变量起到标志的作用，用它来标志文件是否被保存过。然后我们再在相应的源文件里进行这些函数的定义。

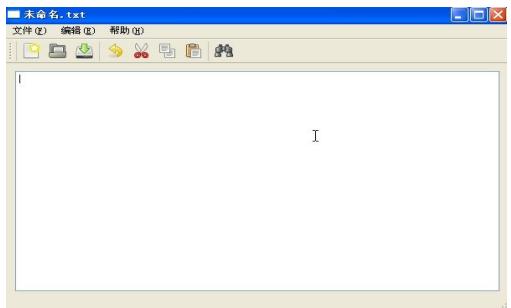
**3. 在 mainwindow.cpp 中先加入头文件 #include <QtGui>，然后在构造函数里添加以下几行代码。**

isSaved = false; //初始化文件为未保存过状态

curFile = tr("未命名.txt"); //初始化文件名为“未命名.txt”

setWindowTitle(curFile); //初始化主窗口的标题

这是对主窗口进行初始化。效果如下。



#### 4.然后添加“新建”操作的函数定义。

```
void MainWindow::do_file_New() //实现新建文件的功能
{
    do_file_SaveOrNot();
    isSaved = false;
    curFile = tr("未命名.txt");
    setWindowTitle(curFile);
    ui->textEdit->clear(); //清空文本编辑器
    ui->textEdit->setVisible(true); //文本编辑器可见
}
```

新建文件，先要判断正在编辑的文件是否需要保存。然后将新建的文件标志为未保存过状态。

#### 5.再添加 do\_file\_SaveOrNot 函数的定义。

```
void MainWindow::do_file_SaveOrNot() //弹出是否保存文件对话框
{
    if(ui->textEdit->document()->isModified()) //如果文件被更改过，弹出保存对话框
    {
        QMessageBox box;
        box.setWindowTitle(tr("警告"));
        box.setIcon(QMessageBox::Warning);
        box.setText(curFile + tr(" 尚未保存，是否保存？"));
        box.setStandardButtons(QMessageBox::Yes | QMessageBox::No);
        if(box.exec() == QMessageBox::Yes) //如果选择保存文件，则执行保存操作
            do_file_Save();
    }
}
```

这个函数实现弹出一个对话框，询问是否保存正在编辑的文件。



#### 6.再添加“保存”操作的函数定义。

```
void MainWindow::do_file_Save() //保存文件
```

```

{
if(isSaved){ //如果文件已经被保存过，直接保存文件
saveFile(curFile);
}
else{
do_file_SaveAs(); //如果文件是第一次保存，那么调用另存为
}
}

```

对文件进行保存时，先判断其是否已经被保存过，如果没有被保存过，就要先对其进行另存为操作。

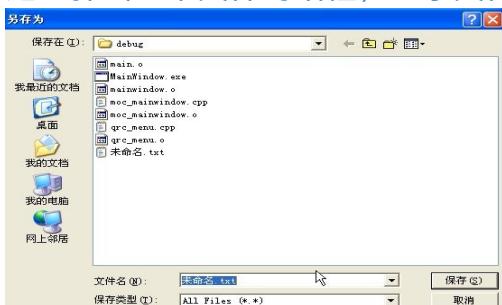
### 7.下面是“另存为”操作的函数定义。

```

void MainWindow::do_file_SaveAs() //文件另存为
{
QString fileName = QFileDialog::getSaveFileName(this,tr("另存为"),curFile);
//获得文件名
if(!fileName.isEmpty()) //如果文件名不为空，则保存文件内容
{
saveFile(fileName);
}
}

```

这里弹出一个文件对话框，显示文件另存为的路径。



### 8.下面是实际文件存储操作的函数定义。

```

bool MainWindow::saveFile(const QString& fileName)
//保存文件内容，因为可能保存失败，所以具有返回值，来表明是否保存成功
{
QFile file(fileName);
if(!file.open(QFile::WriteOnly | QFile::Text))
//以只写方式打开文件，如果打开失败则弹出提示框并返回
{
QMessageBox::warning(this,tr("保存文件"),
tr("无法保存文件 %1:\n %2").arg(fileName)
.arg(file.errorString()));
return false;
}
//%1,%2 表示后面的两个 arg 参数的值
QTextStream out(&file); //新建流对象，指向选定的文件

```

```

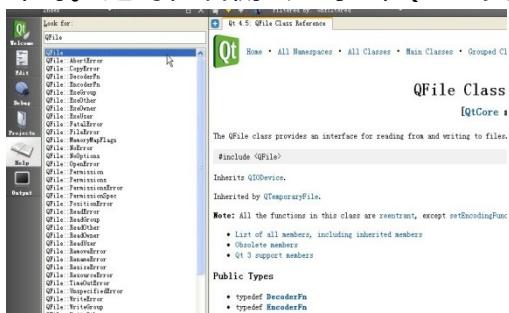
out << ui->textEdit->toPlainText(); //将文本编辑器里的内容以纯文本的形式输出到流
对象中
isSaved = true;
curFile = QFile::fileInfo(fileName).canonicalFilePath(); //获得文件的标准路径
setWindowTitle(curFile); //将窗口名称改为现在窗口的路径
return true;
}

```

这个函数实现将文本文件进行存储。下面我们对其中的一些代码进行讲解。

`QFile file(fileName);`一句，定义了一个 `QFile` 类的对象 `file`，其中 `filename` 表明这个文件就是我们保存的的文件。然后我们就可以用 `file` 代替这个文件，来进行一些操作。Qt 中文件的操作和 C, C++ 很相似。对于 `QFile` 类对象怎么使用，我们可以查看帮助。

点击 Qt Creator 最左侧的 Help，在其中输入 `QFile`，在搜索到的列表中选择 `QFile` 即可。这时在右侧会显示出 `QFile` 类中所有相关信息以及他们的用法和说明。



//  
我们往下拉，会发现下面有关于怎么读取文件的示例代码。  
//

```

The size of the file is returned by size(), you can get the same
you've reached the end of the file, atEnd() returns true.

Reading Files Directly

The following example reads a text file line by line:

QFile file("in.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;

while (!file.atEnd()) {
    QByteArray line = file.readLine();
    process_line(line);
}

The QIODevice::Text flag passed to open() tells Qt to convert Windows
QFile assumes binary, i.e. it doesn't perform any conversion on the

```

//  
再往下便能看到用 `QTextStream` 类对象，进行字符串输入的例子。下面也提到了 `QFileInfo` 和 `QDir` 等相关的类，我们可以点击它们去看一下具体的使用说明。  
//

```

right:

QFile file("out.txt");
if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
    return;

QTextStream out(&file);
out << "The magic number is: " << 49 << "\n";

QDataStream is similar, in that you can use operator<<() to write data and operator>>() to read it

When you use QFile, QFileInfo, and QDir to access the file system with Qt, you can use Unicode file
names to an 8-bit encoding. If you want to use standard C++ APIs (<iostream> or <iostream>) or platform-specific
code can use the encodeName() and decodeName() functions to convert between Unicode file names and 8-bit

```

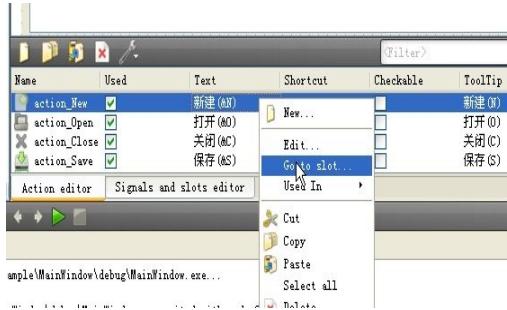
上面只是做了一个简单的说明。以后我们对自己不明白的类都可以去帮助里进行查找，这也许是我们以后要做的最多的一件事了。对于其中的英文解释，我们最好想办法弄明白它的大

意，其实网上也有一些中文的翻译，但最好还是从一开始就尝试着看英文原版的帮助，这样以后才不会对中文翻译产生依赖。

我们这次只是很简单的说明了一下怎样使用帮助文件，这不表明它不重要，而是因为这里不可能将每个类的帮助都解释一遍，没有那么多时间，也没有那么大的篇幅。而更重要的是因为，我们这个教程只是引你入门，所以很多东西需要自己去尝试。

在以后的教程里，如果不是特殊情况，就不会再对其中的类进行详细解释，文章中的重点是对整个程序的描述，其中不明白的类，自己查看帮助。

**9.双击mainwindow.ui文件，在图形界面窗口下面的Action Editor动作编辑器里，我们右击“新建”菜单一条，选择Go to slot，然后选择triggered（），进入其触发事件槽函数。**



同理，进入其他两个菜单的槽函数，将相应的操作的函数写入槽函数中。如下。

```
void MainWindow::on_action_New_triggered() //信号和槽的关联
{
    do_file_New();
}
void MainWindow::on_action_Save_triggered()
{
    do_file_Save();
}
void MainWindow::on_action_SaveAs_triggered()
{
    do_file_SaveAs();
}
```

最终的mainwindow.cpp文件如下。

```

mainwindow.cpp -> MainWindow -> Mainwindow

1 //加入了新建、保存、另存为的功能
2
3 #include "mainwindow.h"
4 #include <QMainWindow.h>
5 #include <QTextEdit.h>
6
7 Mainwindow::Mainwindow(QWidget *parent)
8 {
9     QMainWindow(QMainWindow(parent), ui->MainWindow);
10
11     ui->label->setAutoFillBackground(true);
12     ui->label->setText("欢迎来到我的第一个工程");
13     ui->label->show();
14
15     setWindowTitle("我的第一个工程");
16 }
17
18 Mainwindow::~Mainwindow()
19 {
20     delete ui;
21 }
22
23 void Mainwindow::do_file_New() //实现新建文件的功能
24 {
25     do_file_SaveOrNot();
26
27     QString curFile = tr("未命名 (%d)");
28     ui->textEdit->clear();
29     ui->textEdit->clear();
30     ui->textEdit->selectAll();
31 }
32
33 void Mainwindow::do_file_SaveOrNot() //弹出是否保存文件对话框
34 {
35     if(ui->menuEdit->document()->isModified())
36     {
37         QMessageBox::StandardButtons buttons = QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel;
38         QMessageBox::Question(this, "警告", "是否保存?", buttons);
39         if(buttons == QMessageBox::Yes)
40             do_file_Save();
41         else if(buttons == QMessageBox::Cancel)
42             do_file_SaveAs();
43     }
44 }
45
46 void Mainwindow::do_file_Save() //保存文件
47 {
48     if(ui->textEdit->document()->isModified())
49     {
50         ui->textEdit->saveFile(curFile);
51     }
52     else
53     {
54         do_file_SaveAs(); //如果文件第一次保存，那么调用另存为
55     }
56 }
57
58 void Mainwindow::do_file_SaveAs() //文件另存为
59 {
60     QString fileName = QFileDialog::getSaveFileName(this, tr("另存为"), curFile);
61
62     if(fileName.isEmpty())
63     {
64         saveFile(fileName);
65     }
66 }
67
68 bool Mainwindow::saveFile(const QString &fileName)
69 {
70     //保存文件内容，因为只有保存失败，所以具有返回值，来表明是否保存成功
71     QFile file(fileName);
72     if(file.open(QIODevice::WriteOnly | QIODevice::Text))
73     {
74         //以只写方式打开文件，如果打开失败则弹出错误并返回
75         QMessageBox::warning(this, tr("保存文件"),
76                             tr("无法打开文件 %1, 错误信息: %2").arg(fileName),
77                             QMessageBox::Ok);
78         return false;
79     }
80     QTextStream out(&file);
81     out << ui->textEdit->toPlainText();
82
83     file.close();
84     curFile = fileName;
85     MainWindowTitle(curFile);
86     return true;
87 }
88
89 void Mainwindow::on_action_New_triggered() //信号和槽的关联
90 {
91     do_file_New();
92 }
93
94 void Mainwindow::on_action_Save_triggered()
95 {
96     do_file_Save();
97 }
98
99 void Mainwindow::on_action_SaveAs_triggered()
100 {
101     do_file_SaveAs();
102 }

```

## 最终的 mainwindow.h 文件如下。

```

mainwindow.h -> Mainwindow
1 #ifndef MAINWINDOW_H
2 #define MAINWINDOW_H
3
4 #include <QMainWindow>
5
6 namespace UI
7 {
8     class Mainwindow;
9 }
10
11 class Mainwindow : public QMainWindow
12 {
13     Q_OBJECT
14
15     public:
16         Mainwindow(QWidget *parent = 0);
17         ~Mainwindow();
18
19     private:
20         UI::Mainwindow *ui;
21         bool isSaved; //为true时标志文件已保存过，为false时标志文件尚未保存过
22         QString curFile; //保存当前文件的文件名
23
24         void do_file_New(); //新建文件
25         void do_file_SaveOrNot(); //是否需要保存现有文件
26         void do_file_Save(); //保存文件
27         void do_file_SaveAs(); //文件另存为
28         bool saveFile(const QString &fileName); //存储文件
29
30     signals:
31         void on_action_SaveAs_triggered();
32         void on_action_Save_triggered();
33         void on_action_New_triggered();
34 }
35
36 #endif // MAINWINDOW

```

这时点击运行，就能够实现新建文件，保存文件，文件另存为的功能了。

然后实现打开，关闭，退出，撤销，复制，剪切，粘贴的功能。

**先备份上次的工程文件，然后再将其打开。**

**1.先在 `mainwindow.h` 文件中加入函数的声明。**

`void do_file_Open(); //打开文件`

`bool do_file_Load(const QString& fileName); //读取文件`

**2.再在 `mainwindow.cpp` 文件中写函数的功能实现。**

`void MainWindow::do_file_Open()//打开文件`

{

`do_file_SaveOrNot();//是否需要保存现有文件`

`QString fileName = QFileDialog::getOpenFileName(this);`

```

//获得要打开的文件的名字
if(!fileName.isEmpty())//如果文件名不为空
{
    do_file_Load(fileName);
}
ui->textEdit->setVisible(true); //文本编辑器可见
}

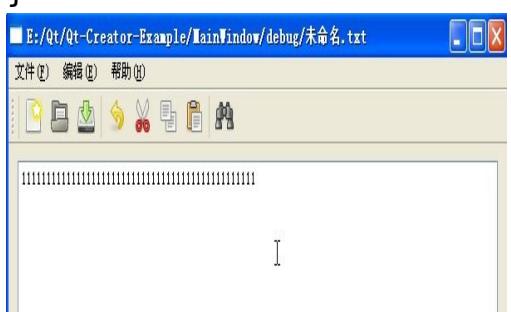
```



```

bool MainWindow::do_file_Load(const QString& fileName) //读取文件
{
    QFile file(fileName);
    if(!file.open(QFile::ReadOnly | QFile::Text))
    {
        QMessageBox::warning(this,tr("读取文件"),tr("无法读取文件 %1:\n%2.").arg(fileName).arg(file.errorString()));
        return false; //如果打开文件失败，弹出对话框，并返回
    }
    QTextStream in(&file);
    ui->textEdit->setText(in.readAll()); //将文件中的所有内容都写到文本编辑器中
    curFile = QFileInfo(fileName).canonicalFilePath();
    setWindowTitle(curFile);
    return true;
}

```



上面的打开文件函数与文件另存为函数相似，读取文件的函数与文件存储函数相似。

**3.**然后按顺序加入更菜单的关联函数，如下。

```

void MainWindow::on_action_Open_triggered() //打开操作
{
    do_file_Open();
}
//
void MainWindow::on_action_Close_triggered() //关闭操作

```

```

{
do_file_SaveOrNot();
ui->textEdit->setVisible(false);
}
//
void MainWindow::on_action_Quit_triggered() //退出操作
{
on_action_Close_triggered(); //先执行关闭操作
qApp->quit(); //再退出系统, qApp 是指向应用程序的全局指针
}
//
void MainWindow::on_action_Undo_triggered() //撤销操作
{
ui->textEdit->undo();
}
//
void MainWindow::on_action_Cut_triggered() //剪切操作
{
ui->textEdit->cut();
}
//
void MainWindow::on_action_Copy_triggered() //复制操作
{
ui->textEdit->copy();
}
//
void MainWindow::on_action_Past_triggered() //粘贴操作
{
ui->textEdit->paste();
}

57 void MainWindow::do_file_Open() //打开文件
58 {
59     do_file_SaveOrNot(); //是否需要保存现有文件
60     QFileDialog::getOpenFileName(this);
61     //获得要打开的文件的名字
62     if(!fileName.isEmpty()) //如果文件名不为空
63     {
64         do_file_Load(fileName);
65     }
66     ui->textEdit->setVisible(true); //文本编辑器可见
67 }
68 bool MainWindow::do_file_Load(const QString& fileName) //读取文件
69 {
70     QFile file(fileName);
71     if(!file.open(QFile::ReadOnly | QFile::Text))
72     {
73         QMessageBox::warning(this,tr("读取文件"),tr("无法读取文件 %1:\n%2.")
74                             .arg(fileName).arg(file.errorString()));
75         return false;
76     }
77     QTextStream in(&file);
78     ui->textEdit->setText(in.readAll());
79     curFile = QFileInfo(fileName).canonicalFilePath();
80     setWindowTitle(curFile);
81     return true;
82 }
83 void MainWindow::on_action_Close_triggered()
84 {
85     do_file_SaveOrNot();
86     ui->textEdit->setVisible(false);
87 }
88 void MainWindow::on_action_Quit_triggered() //退出操作
89 {
90     on_action_Close_triggered(); //先执行关闭操作
91     qApp->quit(); //再退出系统, qApp 是指向应用程序的全局指针
92 }
93 void MainWindow::on_action_Undo_triggered()
94 {
95     ui->textEdit->undo();
96 }
97 void MainWindow::on_action_Cut_triggered()
98 {
99     ui->textEdit->cut();
100 }
101 void MainWindow::on_action_Copy_triggered()
102 {
103     ui->textEdit->copy();
104 }
105 void MainWindow::on_action_Past_triggered()
106 {
107     ui->textEdit->paste();
108 }

```

因为复制，撤销，全选，粘贴，剪切等功能，是TextEdit默认就有的，所以我们只需调用一下相应函数就行。

到这里，除了查找和帮助两个菜单的功能没有加上以外，其他功能都已经实现了。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 6,269 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 七、Qt Creator 实现文本查找

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

现在加上查找菜单的功能。因为这里要涉及关于 Qt Creator 的很多实用功能，所以单独用一篇文章来介绍。

以前都用设计器设计界面，而这次我们用代码实现一个简单的查找对话框。对于怎么实现查找功能的，我们详细地分步说明了怎么进行类中方法的查找和使用。其中也将 Qt Creator 智能化的代码补全功能和程序中函数的声明位置和定义位置间的快速切换进行了介绍。

**1.**首先还是保存以前的工程，然后再将其打开。

我们发现 Qt Creator 默认的字体有点小，可以按下 Ctrl 键的同时按两下+键，来放大字体。也可以选择 Edit->Advanced->Increase Font Size。

```
[70] void MainWindow::on_action_Find_triggered()
[71] {
[72]     QDialog *findDlg = new QDialog(this);
[73]     //新建一个对话框，用于查找操作
[74]     findDlg->setWindowTitle(tr("查找"));
[75]     //设置对话框的标题
[76]     find_textLineEdit = new QLineEdit(findDlg);
[77]     //将行编辑器加入到新建的查找对话框中
[78]     QpushButton *find_Btn = new QPushButton(tr("查找下一个"),findDlg);
[79]     //加入一个“查找下一个”的按钮
[80]     QVBoxLayout* layout = new QVBoxLayout(findDlg);
[81]     layout->addWidget(find_textLineEdit);
[82]     layout->addWidget(find_Btn);
[83]     //新建一个垂直布局管理器，并将行编辑器和按钮加入其中
[84]     findDlg ->show();
[85]     //显示对话框
[86]     connect(find_Btn,SIGNAL(clicked()),this,SLOT(show_findText()));
[87] }
[88]
```

**2.**在 **mainwindow.h** 中加入 **#include <QLineEdit>** 的头文件包含，在 **private** 中添加

**QLineEdit \*find\_textLineEdit;** //声明一个行编辑器，用于输入要查找的内容

在 **private slots** 中添加

**void show\_findText();**

在该函数中实现查找字符串的功能。

**3.**我们进入查找菜单的触发事件槽函数，更改如下。

```
void MainWindow::on_action_Find_triggered()
{
    QDialog *findDlg = new QDialog(this);
    //新建一个对话框，用于查找操作，this 表明它的父窗口是 MainWindow。
    findDlg->setWindowTitle(tr("查找"));
    //设置对话框的标题
    find_textLineEdit = new QLineEdit(findDlg);
    //将行编辑器加入到新建的查找对话框中
    QPushbutton *find_Btn = new QPushbutton(tr("查找下一个"),findDlg);
    //加入一个“查找下一个”的按钮
    QVBoxLayout* layout = new QVBoxLayout(findDlg);
    layout->addWidget(find_textLineEdit);
    layout->addWidget(find_Btn);
    //新建一个垂直布局管理器，并将行编辑器和按钮加入其中
    findDlg ->show();
    //显示对话框
    connect(find_Btn,SIGNAL(clicked()),this,SLOT(show_findText()));
    //设置“查找下一个”按钮的单击事件和其槽函数的关联
}
```

这里我们直接用代码生成了一个对话框，其中一个行编辑器可以输入要查找的字符，一个按钮可以进行查找操作。我们将这两个部件放到了一个垂直布局管理器中。然后显示这个对话框。并设置了那个按钮单击事件与 show\_findText()函数的关联。



## 5.下面我们开始写实现查找功能的 **show\_findText()** 函数。

```
void MainWindow::show_findText()//“查找下一个”按钮的槽函数
{
    QString findText = find_textLineEdit->text();
    //获取行编辑器中的内容
}
```

先用一个 **QString** 类的对象获得要查找的字符。然后我们一步一步写查找操作的语句。

6.在下一行写下 **ui**，然后直接按下键盘上的“<.”键，这时系统会根据是否是指针对象而自动生成“->”或“.”，因为 **ui** 是指针对象，所以自动生成“->”号，而且弹出了 **ui** 中的所有部件名称的列表。如下图。

```
190 void MainWindow::show_findText()//“查找下一个”按钮的槽函数
191 {
192     QString findText = find_textLineEdit->text();
193     //获取行编辑器中的内容
194     ui->
195 }
```

7.我们用向下的方向键选中列表中的 **textEdit**。或者我们可以先输入 **text**，这时能缩减列表的内容。

```
190 void MainWindow::show_findText()//“查找下一个”按钮的槽函数
191 {
192     QString findText = find_textLineEdit->text();
193     //获取行编辑器中的内容
194     ui->textEdit
195 }
```

8.如上图我们将鼠标放到 **textEdit** 上，这时便出现了 **textEdit** 的类名信息，且后面出现一个 **F1** 按键。我们按下键盘上的 **F1**，便能出现 **textEdit** 的帮助。

The screenshot shows the Qt Creator IDE with the code editor open. A tooltip for the 'find' function of the QTextEdit class is displayed, providing detailed documentation and examples. The code in the editor is related to a search feature in a text editor application.

```

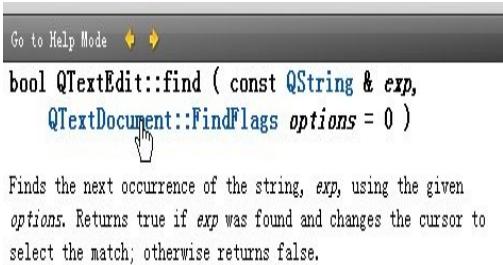
178     QPushButton* find_Btn = new QPushButton(tr("查找下一个"));
179     //...
180     QBoxLayout* layout = new QVBoxLayout(findDlg);
181     layout->addWidget(findLineEdit);
182     layout->addWidget(find_Btn);
183     //创建一个垂直堆栈容器，并将行编辑器和按钮加入其中
184     findDlg->setLayout(layout);
185     //...
186     connect(find_Btn, SIGNAL(clicked()), this, SLOT(show_findDlg));
187     //设置“查找下一个”按钮的单击事件为调用槽的关联
188 }
189
190 void MainWindow::show_findText() //“查找下一个”按钮的槽函数
191 {
192     QString findText = find_textLineEdit->text();
193     //获取行编辑器中的内容
194     ui->textEdit;
195 }
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220

```

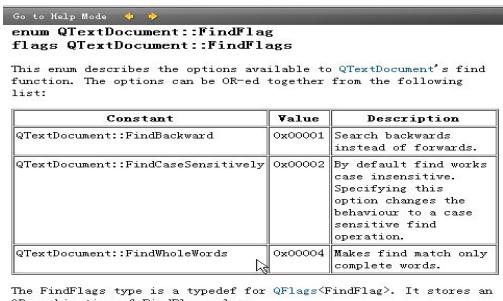
## 9.我们在帮助中向下拉，会发现这里有一个 **find** 函数。

- `QString documentTitle () const`
- `void ensureCursorVisible ()`
- `QList<ExtraSelection> extraSelections () const`
- `bool find ( const QString & exp, QTextDocument::FindFlags options = 0 )`
- `QString fontFamily () const`
- `bool fontItalic () const`
- `qreal fontPointSize () const`
- `bool fontUnderline () const`

## 10.我们点击 **find**，查看其详细说明。



## 11.可以看到 **find** 函数可以实现文本编辑器中字符串的查找。其中有一个 **FindFlags** 的参数，我们点击它查看其说明。



## 12.可以看到它是一个枚举变量（enum），有三个选项，第一项是向后查找（即查找光标以前的内容，这里的前后是相对的说法，比如第一行已经用完了，光标在第二行时，把第一行叫做向后。），第二项是区分大小写查找，第三项是查找全部。

### 13.我们选用第一项，然后写出下面的语句。

```
ui->textEdit->find(findText,QTextDocument::FindBackward);
```

//将行编辑器中的内容在文本编辑器中进行查找

当我们刚打出“f”时，就能自动弹出 `textEdit` 类的相关属性和方法。

```

90 void MainWindow::show_findText() //“查找下一个”
91 {
92     QString findText = find_textLineEdit->text();
93     //获取行编辑器中的内容
94     ui->textEdit->f
95 }
96
97

```

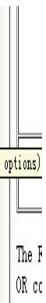


可以看到，当写完函数名和第一个“(”后，系统会自动显示出该函数的函数原型，这样可以使我们减少出错。

```

void MainWindow::show_findText() //“查找下一个”按钮的槽函数
{
    QString findText = find_textLineEdit->text();
    //获取行编辑器中的内容
    bool find(const QString &exp, QTextDocument::FindFlags options)
    ui->textEdit->find(
}

```



**14.**这时已经能实现查找的功能了。但是我们刚才看到 **find** 的返回值类型是 **bool** 型，而且，我们也应该为查找不到字符串作出提示。

```

if(!ui->textEdit->find(findText,QTextDocument::FindBackward))
{
    QMessageBox::warning(this,tr("查找"),tr("找不到 %1")
    .arg(findText));
}

```

因为查找失败返回值是 **false**，所以 if 条件加了“！”号。在找不到时弹出警告对话框。



**15.**到这里，查找功能就基本上写完了。**show\_findText()**函数的内容如下。

```

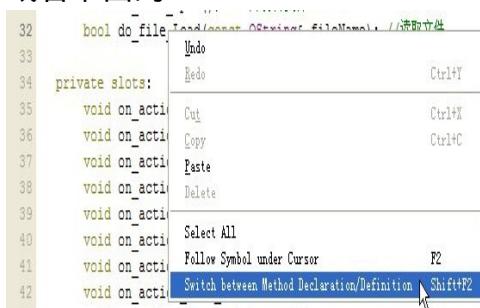
190 void MainWindow::show_findText() //“查找下一个”按钮的槽函数
191 {
192
193     QString findText = find_textLineEdit->text();
194     //获取行编辑器中的内容
195     if(!ui->textEdit->find(findText,QTextDocument::FindBackward))
196     {
197         QMessageBox::warning(this,tr("查找"),tr("找不到 %1")
198         .arg(findText));
199     }
200     //将行编辑器中的内容在文本编辑器中进行查找,FindBackward表明查找光标以前的内容
201 }

```

我们会发现随着程序功能的增强，其中的函数也会越来越多，我们都会为查找某个函数的定义位置感到头疼。而在 **Qt Creator** 中有几种快速定位函数的方法，我们这里讲解三种。

第一，在函数声明的地方直接跳转到函数定义的地方。

如在 do\_file\_Load 上点击鼠标右键，在弹出的菜单中选择 Follow Symbol under Cursor 或者下面的 Switch between Method Declaration/Definition。

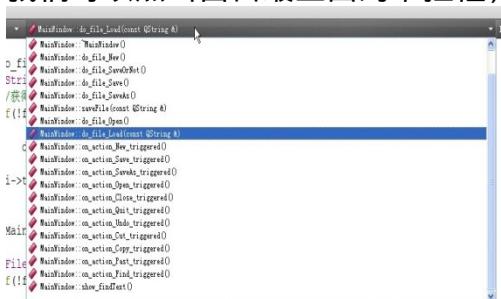


这时系统就会自动跳转到函数定义的位置。如下图。

```
101 bool MainWindow::do_file_Load(const QString& fileName)//读取文件
102 {
103     QFile file(fileName);
104     if(!file.open(QFile::ReadOnly | QFile::Text))
105     {
106         QMessageBox::warning(this,tr("读取文件"),tr("无法读取文件 %1:\n%2",
107                             .arg(fileName).arg(file.errorString())));
108     }
109 }
```

第二，快速查找一个文件里的所有函数。

我们可以点击窗口最上面的下拉框，这里会显示本文件中所有函数的列表。



第三，利用查找功能。

1. 我们先将鼠标定位到一个函数名上。

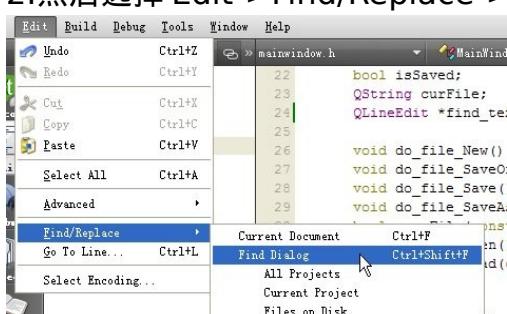
```
void do_file_SaveAs(); //文件为存
bool saveFile(const QString& fileName); //存储文件
void do_file_Open(); //打开文件
bool do_file_Load(const QString& fileName); //读取文件
```

】

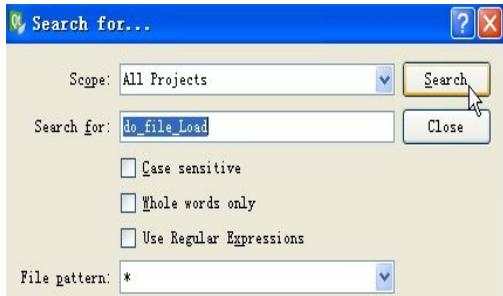
private slots:

```
void on_action_Find_triggered();
```

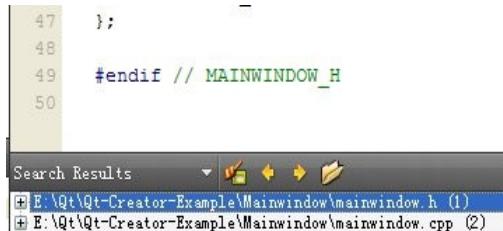
2. 然后选择 Edit->Find/Replace->Find Dialog。



3.这时会出现一个查找对话框，可以看到要查找的函数名已经写在里面了。



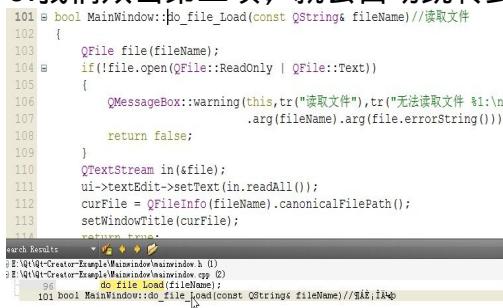
4.当我们按下 Search 按钮后，会在查找结果窗口显示查找到的结果。



5.我们点击第二个文件。会发现在这个文件中有两处关键字是高亮显示。



6.我们双击第二项，就会自动跳转到函数的定义处。



文章讲到这里，我们已经很详细地说明了怎样去使用一个类里面没有用过的方法函数；也说明了Qt Creator中的一些便捷操作。可以看到，Qt Creator开发环境，有很多很人性化的设计，我们应该熟练应用它们。

在以后的文章中，我们不会再很详细地去用帮助来说明一个函数是怎么来的，该怎么用，这些应该自己试着去查找。

分类：[Qt系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,720 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 八、Qt Creator 实现状态栏显示

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在程序主窗口 **Mainwindow** 中，有菜单栏，工具栏，中心部件和状态栏。前面几个已经讲过了，这次讲解状态栏的使用。

程序中有哪些不明白的类或函数，请自己查看帮助。

**1.**我们在 **mainwindow.h** 中做一下更改。

加入头文件包含： #include <QLabel>

加入私有变量和函数：

```
QLabel* firstStatusLabel; //声明两个标签对象，用于显示状态信息
```

```
QLabel* secondStatusLabel;
```

```
void initStatusBar(); //初始化状态栏
```

加入一个槽函数声明： void do\_cursorChanged(); //获取光标位置信息

**2.**在 **mainwindow.cpp** 中加入状态栏初始化函数的定义。

```
void MainWindow::initStatusBar()
```

```
{
```

```
QStatusBar* bar = ui->statusBar; //获取状态栏
```

```
firstStatusLabel = new QLabel; //新建标签
```

```
firstStatusLabel->setMinimumSize(150,20); //设置标签最小尺寸
```

```
firstStatusLabel->setFrameShape(QFrame::WinPanel); //设置标签形状
```

```
firstStatusLabel->setFrameShadow(QFrame::Sunken); //设置标签阴影
```

```
secondStatusLabel = new QLabel;
```

```
secondStatusLabel->setMinimumSize(150,20);
```

```
secondStatusLabel->setFrameShape(QFrame::WinPanel);
```

```
secondStatusLabel->setFrameShadow(QFrame::Sunken);
```

```
bar->addWidget(firstStatusLabel);
```

```
bar->addWidget(secondStatusLabel);
```

```
firstStatusLabel->setText(tr("欢迎使用文本编辑器")); //初始化内容
```

```
secondStatusLabel->setText(tr("yafeilinux 制作!"));
```

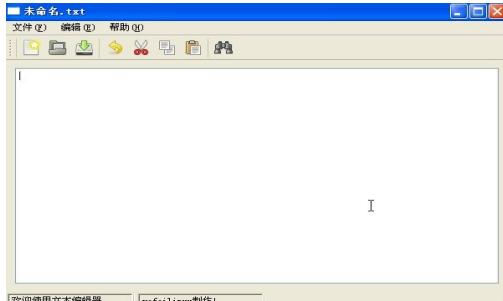
```
}
```

这里将两个标签对象加入到了主窗口的状态栏里，并设置了他们的外观和初值。

**3.**在构造函数里调用状态栏初始化函数。

```
initStatusBar();
```

这时运行程序，效果如下。



**4.**在 **mainwindow.cpp** 中加入获取光标位置的函数的定义。

```
void MainWindow::do_cursorChanged()
```

```
{
```

```
int rowNum = ui->textEdit->document()->blockCount();
```

```
//获取光标所在行的行号  
const QTextCursor cursor = ui->textEdit->textCursor();  
int colNum = cursor.columnNumber();  
//获取光标所在列的列号  
firstStatusLabel->setText(tr("%1 行 %2 列").arg(rowNum).arg(colNum));  
//在状态栏显示光标位置  
}
```

这个函数可获取文本编辑框中光标的位置，并显示在状态栏中。

## 5. 在构造函数添加光标位置改变信号的关联。

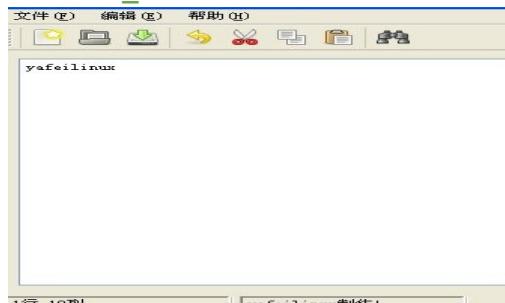
```
connect(ui->textEdit,SIGNAL(cursorPositionChanged()),this,SLOT(do_cursorChanged()));
```

这时运行程序。效果如下。



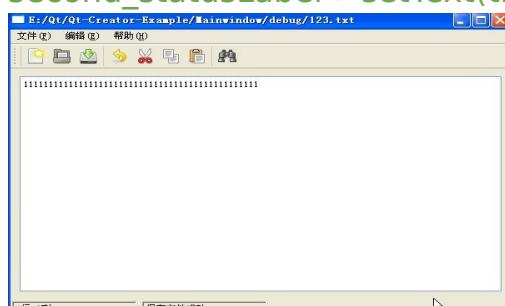
## 6. 在 do\_file\_Load 函数的最后添加下面语句。

```
secondStatusLabel->setText(tr("打开文件成功"));
```



## 7. 在 saveFile 函数的最后添加以下语句。

```
secondStatusLabel->setText(tr("保存文件成功"));
```



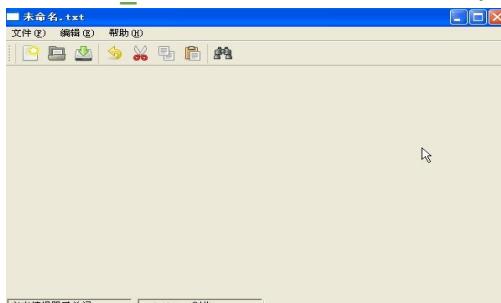
## 8. 在 on\_action\_Find\_triggered 函数的后面添加如下语句。

```
secondStatusLabel->setText(tr("正在进行查找"));
```



9. 在 `on_action_Close_triggered` 函数最后添加如下语句。

```
firstStatusLabel->setText(tr("文本编辑器已关闭"));  
secondStatusLabel->setText(tr("yafeilinux 制作!"));
```



最终的mainwindow.cpp文件内容如下。

```
/*设置到了剪贴板，保存为文件的图标，显示的是图标+文件名+“已粘贴”字样，类似：剪切、粘贴的功能*/
//设置到了剪贴板的图标，显示的是图标+文件名+“已粘贴”字样，类似：剪切、粘贴的功能

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    isSaved = false; // 初始化文件为未保存状态
    setWindowTitle("Qt文本编辑器"); // 设置窗口标题
    setWindowIcon(QIcon(":/image/icon.png")); // 初始化主窗口的图标
    connect(ui->textEdit, SIGNAL(cursorPositionChanged()), this, SLOT(do_cursorChanged()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::init_StatusBar()
{
    first_StatusLabel = new QLabel("行数: 0 / 列数: 0");
    second_StatusLabel = new QLabel("字体: 黑体 / 字号: 16");
    first_StatusLabel->setStyleSheet("font-size: 10px; color: black; border: 1px solid black; padding: 2px; margin-right: 10px");
    first_StatusLabel->setFrameShadow(QFrame::Sunken); // 设置图标阴影效果
    second_StatusLabel = new QLabel("字体: 黑体 / 字号: 16");
    second_StatusLabel->setStyleSheet("font-size: 10px; color: black; border: 1px solid black; padding: 2px; margin-right: 10px");
    second_StatusLabel->setFrameShadow(QFrame::Sunken);
    bar_top->addWidget(first_StatusLabel);
    bar_top->addWidget(second_StatusLabel);
    first_StatusLabel->setText(tr("行数 %列数"), arg(rowNum), arg(colNum));
    second_StatusLabel->setText(tr("字体 %字号"));
}

void MainWindow::cursorChanged()
{
    int rowNum = ui->textEdit->document()->blockCount();
    if(鼠标光标在行的号) //如果光标在行的号
        const QTextCursor cursor = ui->textEdit->textCursor();
        ui->textEdit->selectAll(); //选中所有文字
    if(鼠标光标在列的号) //如果光标在列的号
        first_StatusLabel->setText(tr("%行 %列列"));
    //在状态栏显示光标的位
}

void MainWindow::do_file_Open() //实现新建文件的功能
{
    do_file_saveOnNot();
    isSaved = false;
    QFileDialog dialog(this, tr("打开文件..."));
    dialog.setFileMode(QFileDialog::ExistingFiles);
    dialog.setNameFilter("文本文件 (*.txt)");
    ui->textEdit->clear(); //清空文本编辑器
    ui->textEdit->setTabVisible(true); //文件编辑器可见。
}

void MainWindow::do_file_Save() //退出是否保存文件对话框
{
    if(ui->textEdit->document())->isModified() //如果文件被更改过
    {
        QMessageBox box;
        box.setWindowIcon(QIcon(":/icon/icon.png"));
        box.setCaption(QMessageBox::Information);
        box.setText("你对文件做了修改, 是否保存? ");
        box.setStandardButtons(QMessageBox::Yes | QMessageBox::No);
        if(box.exec() == QMessageBox::Yes) //如果选择保存文件, 则执行保存操作
            do_file_SaveAs();
    }
}

void MainWindow::do_file_SaveAs() //保存文件
{
    if(ui->textEdit->document()) //如果文件已经保存过, 直接保存文件
        saveFile(outFile);
    else
        do_file_SaveAs(); //如果文件是第一次保存, 那么调用另存为
}

void MainWindow::do_file_SaveAs() //文件另存为
{
    QFileDialog dialog(this, tr("另存为..."));
    dialog.setFileMode(QFileDialog::SaveFile);
    dialog.setNameFilter("文本文件 (*.txt)");
    if(!fileName.isEmpty()) //如果文件名不为空, 则保持文件内容
        saveFile(fileName);
    else
        dialog.selectFile(fileName);
}

void MainWindow::saveFile(const QString &fileName)
{
    QFile file(fileName);
    if(!file.open(QFile::WriteOnly | QFile::Text))
        QMessageBox::warning(this, tr("无法打开文件 %1").arg(fileName),
                             tr("无法打开文件 %1").arg(file.errorString()));
    return false;
}

QTextStream out(&file);
out << ui->textEdit->toPlainText();

isSaved = true;
outFile = QFileInfo(fileName).canonicalFilePath();
ui->statusBar->showText(tr("保存文件成功"));
ui->statusBar->showText(tr("保存文件成功"));

    return true;
}
```

```

30 void MainWindow::do_file_Save() //保存文件
31 {
32     if(isSaved) //如果文件已经被保存过，直接保存文件
33     {
34         saveFile(cufile);
35     }
36     else
37     {
38         do_file_SaveAs(); //如果文件是第一次保存，那么调用另存为
39     }
40 }
41 void MainWindow::do_file_SaveAs() //文件另存为
42 {
43     QString fileName = QFileDialog::getSaveFileName(this, tr("另存为"), cufile);
44     if(fileName.isEmpty()) //如果文件名不为空，则保存文件内容
45     {
46         saveFile(fileName);
47     }
48 }
49 bool MainWindow::saveFile(const QString &fileName)
50 {
51     QFile file(fileName);
52     if(file.open(QIODevice::WriteOnly | QIODevice::Text))
53     {
54         QTextStream out(&file);
55         out.setCodec("UTF-8");
56         out.setText(ui->textEdit->toPlainText());
57         file.close();
58     }
59     QMessageBox::warning(this, tr("保存文件"),
60                         tr("文件保存成功 %1").arg(fileName));
61     return true;
62 }
63 QTextStream out(&file);
64 out.setCodec("UTF-8");
65 out.setText(ui->textEdit->toPlainText());
66 outFile = QFileInfo(fileName).canonicalFilePath();
67 second_statusLabel->setText(tr("保存文件成功"));
68 second_statusBar->setText(tr("保存文件成功"));
69 second_statusBar->show();
70 }
71 void MainWindow::on_action_SaveAs_triggered()
72 {
73     do_file_SaveAs();
74 }
75 void MainWindow::on_action_Open_triggered()
76 {
77     do_file_Open();
78 }
79 void MainWindow::on_action_Close_triggered()
80 {
81     do_file_SaveOrNot();
82     if(first_statusLabel->text() == tr("文本编辑器已关闭"))
83     {
84         first_statusLabel->setText(tr("文本编辑器已打开"));
85     }
86     second_statusLabel->setText(tr("文本编辑器已打开"));
87 }
88 void MainWindow::on_action_Quit_triggered() //退出操作
89 {
90     on_action_Close_triggered(); //先执行关闭操作
91     qApp->quit(); //再退出系统，qApp是启动应用程序的全局指针
92 }
93 void MainWindow::on_action_Undo_triggered()
94 {
95     ui->textEdit->undo();
96 }
97 void MainWindow::on_action_Cut_triggered()
98 {
99     ui->textEdit->cut();
100 }
101 void MainWindow::on_action_Copy_triggered()
102 {
103     ui->textEdit->copy();
104 }
105 void MainWindow::on_action_Past_triggered()
106 {
107     ui->textEdit->paste();
108 }
109 void MainWindow::on_action_Find_triggered()
110 {
111     QFileDialog findDlg = new QFileDialog(this);
112     findDlg->setWindowTitle(tr("查找"));
113     //设置对话框的按钮
114     QLineEdit *findText = new QLineEdit(findDlg);
115     //将行编辑框加入到新建的查找对话框中
116     QPushButton *find = new QPushButton(tr("查找下一个"), findDlg);
117     //将行编辑框和按钮添加到布局中
118     QVBoxLayout *layout = new QVBoxLayout(findDlg);
119     layout->addWidget(findText);
120     layout->addWidget(find);
121     //将布局设置为对话框的子部件，并将编辑框和按钮加入其中
122     findDlg->setLayout(layout);
123     //显示对话框
124     secondStatusLabel->setText(tr("正在运行查找"));
125     connect(find, SIGNAL(clicked()), this, SLOT(show_findText()));
126     //使对话框在按下“查找”按钮时单击事件和其槽函数的关系
127 }
128 void MainWindow::show_findText() //查找下一个按钮的槽函数
129 {
130     QString findText = findTextLineEdit->text(); //获取行编辑框中的内容
131     if(ui->textEdit->find(findText, QtTextDocument::FindBackward))
132     {
133         QMessageBox::warning(this, tr("查找"),
134                             tr("找不到 %1").arg(findText));
135     }
136 }
137 //两个编辑器中的内容在文本编辑器中进行查找，FindBackward表明查找光标以前的内容
138 }
139 
```

最终的mainwindow.h文件如下。

```

1 #ifndef MAINWINDOW_H
2 #define MAINWINDOW_H
3
4 #include <QGuiApplication>
5 #include <QMainWindow>
6 #include <QLineEdit>
7 #include <QLabel>
8
9 namespace Ui
10 {
11     class MainWindow;
12 }
13
14 class MainWindow : public QMainWindow
15 {
16     Q_OBJECT
17
18 public:
19     MainWindow(QWidget *parent = 0);
20     ~MainWindow();
21
22 private:
23     Ui::MainWindow *ui;
24     bool isSaved; //为true时标志文件已经保存过，为false时标志文件尚未保存
25     QString cufile; //保存当前文件的文件名
26     QLineEdit *find_textLineEdit; //声明一行编辑框，用于输入要查找的内容
27     QLabel *firstStatusLabel; //声明一个标签，用于显示状态信息
28     QLabel *secondStatusLabel;
29
30     void do_file_New(); //新建文件
31     void do_file_Save(); //保存文件
32     void do_file_SaveAs(); //另存文件
33     bool do_file_Open(); //打开文件
34     void do_file_Load(const QString &fileName); //读取文件
35     void init_StatusBar(); //初始化状态栏
36
37 private slots:
38     void on_action_Find_triggered();
39     void on_action_Past_triggered();
40     void on_action_Copy_triggered();
41     void on_action_Cut_triggered();
42     void on_action_Undo_triggered();
43     void on_action_Quit_triggered();
44     void on_action_Close_triggered();
45     void on_action_Open_triggered();
46     void on_action_SaveAs_triggered();
47     void on_action_Save_triggered();
48     void on_action_New_triggered();
49     void show_findText();
50     void do_cursorChanged(); //获取光标位置信息
51 };
52
53 #endif // MAINWINDOW_H

```

到这里整个文本编辑器的程序就算写完了。我们这里没有写帮助菜单的功能实现，大家可以自己添加。而且程序中也有很多漏洞和不完善的地方，如果有兴趣，大家也可以自己修改。因为时间和篇幅的原因，我们这里就不再过多的讲述。

分类: [Qt 系列教程](#) 作者: [yafeilinux](#) 日期: 四月 30th, 2010. 3,366 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 九、Qt Creator 中鼠标键盘事件的处理实现自定义鼠标指针

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

我们前面一直在说信号，比方说用鼠标按了一下按钮，这样就会产生一个按钮的单击信号，然后我们可以在相应的槽函数里进行相应功能的设置。其实在按下鼠标后，程序要先接收到鼠标按下的事件，然后将这个事件按默认的设置传给按钮。可以看出，事件和信号并不是一回事，事件比信号更底层。而我们以前把单击按钮也叫做事件，这是不确切的，不过大家都知道是什么意思，所以当时也没有细分。

Qt 中的事件可以在 **QEvent** 中查看。下面我们只是找两个例子来进行简单的演示。

**1.还是先建立一个 Qt4 Gui Application 工程，我这里起名为 event。**

**2.添加代码，让程序中可以使用中文。**

即在 main.cpp 文件中加入 #include <QTextCodec> 的头文件包含。

再在下面的主函数里添加

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

**3.在mainwindow.h 文件中做一下更改。**

添加 #include <QtGui> 头文件。因为这样就包含了 QtGui 中所有的子文件。

在 public 中添加两个函数的声明

```
void mouseMoveEvent(QMouseEvent *);  
void keyPressEvent(QKeyEvent *);
```

**4.我们在 mainwindow.ui 中添加一个 Label 和一个 PushButton，将他们拉长点，因为一会要在上面显示标语。**

**5.在 mainwindow.cpp 中的构造函数里添加两个部件的显示文本。**

```
ui->label->setText(tr("按下键盘上的 A 键试试！"));  
ui->pushButton->setText(tr("按下鼠标的一个键，然后移动鼠标试试"));
```

**6.然后在下面进行两个函数的定义。**

*/\*以下是鼠标移动事件\*/*

```
void MainWindow::mouseMoveEvent(QMouseEvent *m)  
{//这里的函数名和参数不能更改  
QCursor my(QPixmap("E:/Qt/Qt-Creator-Example/event/time.png"));  
//为鼠标指针选择图片，注意这里如果用绝对路径，要用“/”，而不能用“\”  
//也可以将图片放到工程文件夹得 debug 文件夹下，这样用相对路径“time.png”就可以了  
QApplication::setOverrideCursor(my);  
//将鼠标指针更改为自己设置的图片  
int x = m->pos().x();  
int y = m->pos().y();  
//获取鼠标现在的位置坐标  
ui->pushButton->setText(tr("鼠标现在的坐标是(%1,%2)， 哈哈好玩  
吧").arg(x).arg(y));  
//将鼠标的位置坐标显示在按钮上  
ui->pushButton->move(m->pos());  
//让按钮跟随鼠标移动  
}  
  
/*以下是键盘按下事件*/  
void MainWindow::keyPressEvent(QKeyEvent *k)
```

```

{
if(k->key() == Qt::Key_A) //判断是否是 A 键按下
{
ui->label->setPixmap(QPixmap("E:/Qt/Qt-Creator-Example/event/linux.jpg"));
ui->label->resize(100,100);
//更改标签图片和大小
}
}

```

注意：这两个函数不是自己新建的，而是对已有函数的重定义，所有函数名和参数都不能改。第一个函数对鼠标移动事件进行了重写。其中实现了鼠标指针的更改，和按钮跟随鼠标移动的功能。

第二个函数对键盘的 A 键按下实现了新的功能。

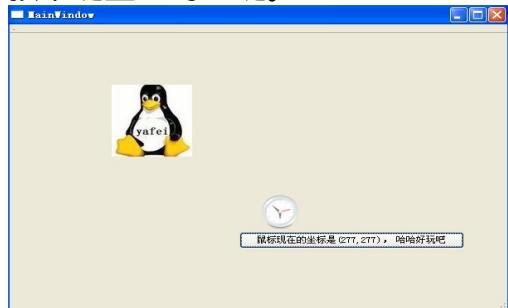
效果如下。



按下鼠标的一个键，并移动鼠标。



按下键盘上的 A 键。



分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 4,107 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十、Qt Creator 中实现定时器和产生随机数

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

有两种方法实现定时器。

第一种。自己建立关联。

1.新建 Gui 工程，工程名可以设置为 **timer**。并在主界面上添加一个标签 **label**，并设置其显示内容为“**0000-00-00 00:00:00 星期日**”。

2.在 **mainwindow.h** 中添加槽函数声明。

```
private slots:
```

```
void timerUpDate();
```

3.在 **mainwindow.cpp** 中添加代码。

添加 #include <QtCore> 的头文件包含，这样就包含了 QtCore 下的所有文件。

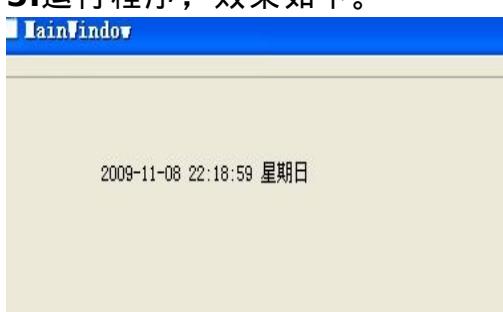
构造函数里添加代码：

```
QTimer *timer = new QTimer(this);  
//新建定时器  
connect(timer,SIGNAL(timeout()),this,SLOT(timerUpDate()));  
//关联定时器计满信号和相应的槽函数  
timer->start(1000);  
//定时器开始计时，其中 1000 表示 1000ms 即 1 秒
```

4.然后实现更新函数。

```
void MainWindow::timerUpDate()  
{  
    QDateTime time = QDateTime::currentDateTime();  
    //获取系统现在的时间  
    QString str = time.toString("yyyy-MM-dd hh:mm:ss dddd");  
    //设置系统时间显示格式  
    ui->label->setText(str);  
    //在标签上显示时间  
}
```

5.运行程序，效果如下。



第二种。使用事件。（有点像单片机中的定时器啊）

1.新建工程。在窗口上添加两个标签。

2.在 **main.cpp** 中添加代码，实现中文显示。

```
#include <QTextCodec>  
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

3.在 **mainwindow.h** 中添加代码。

```
void timerEvent(QTimerEvent *);
```

4.在 **mainwindow.cpp** 中添加代码。

添加头文件 #include <QtCore>

在构造函数里添加以下代码。

```
startTimer(1000); //其返回值为 1，即其 timerId 为 1  
startTimer(5000); //其返回值为 2，即其 timerId 为 2  
startTimer(10000); //其返回值为 3，即其 timerId 为 3
```

添加了三个定时器，它们的 timerId 分别为 1, 2, 3。注意，第几个定时器的返回值就为几。所以要注意定时器顺序。

在下面添加函数实现。

```
void MainWindow::timerEvent(QTimerEvent *t) //定时器事件  
{  
    switch(t->timerId()) //判断定时器的句柄  
    {  
        case 1 : ui->label->setText(tr("每秒产生一个随机数: %1").arg(qrand()%10));break;  
        case 2 : ui->label_2->setText(tr("5 秒后软件将关闭"));break;  
        case 3 : qApp->quit();break; //退出系统  
    }  
}
```

这里添加了三个定时器，并都在定时器事件中判断它们，然后执行相应功能。这样就不用每个定时器都写一个关联函数和槽函数了。

随机数的实现：

上面程序中的 qrand()，可以产生随机数，qrand()%10 可以产生 0-9 之间的随机数。要想产生 100 以内的随机数就%100。以此类推。

但这样每次启动程序后，都按同一种顺序产生随机数。为了实现每次启动程序产生不同的初始值。我们可以使用 qsrand(time(0)); 实现设置随机数的初值，而程序每次启动时 time(0) 返回的值都不同，这样就实现了产生不同初始值的功能。

我们将 qsrand(time(0)); 一句加入构造函数里。

程序最终运行效果如下。



分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 3,561 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十一、Qt 2D 绘图（一）绘制简单图形

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

说明：以后使用的环境为基于 Qt 4.6 的 Qt Creator 1.3.0 windows 版本

本文介绍在窗口上绘制最简单的图形的方法。

1. 新建 Qt4 Gui Application 工程，我这里使用的工程名为 **painter01**，选用

**QDialog** 作为 **Base class**

2. 在 **dialog.h** 文件中声明重绘事件函数 **void paintEvent(QPaintEvent \*);**

3. 在 **dialog.cpp** 中添加绘图类 **QPainter** 的头文件包含 **#include <QPainter>**

4. 在下面进行该函数的重定义。

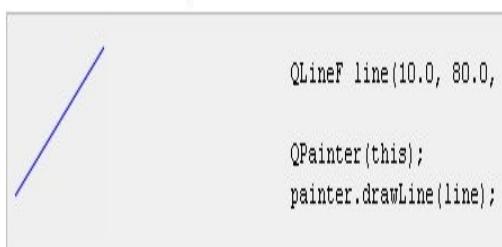
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.drawLine(0,0,100,100);  
}
```

其中创建了 **QPainter** 类对象，它是用来进行绘制图形的，我们这里画了一条线 Line，其中的参数为线的起点 (0, 0)，和终点 (100, 100)。这里的数值指的是像素，详细的坐标设置我们以后再讲，这里知道 (0, 0) 点指的是窗口的左上角即可。运行效果如下：



5. 在 **qt** 的帮助里可以查看所有的绘制函数，而且下面还给出了相关的例子。

```
void drawArc ( const QRect & rectangle, int  
void drawArc ( const QRect & rectangle, int  
void drawArc ( int x, int y, int width, int height  
void drawChord ( const QRectF & rectangle  
void drawChord ( const QRect & rectangle,  
void drawChord ( int x, int y, int width, int height  
void drawConvexPolygon ( const QPointF  
void drawConvexPolygon ( const QPoint *  
void drawConvexPolygon ( const QPolygonF  
void drawConvexPolygon ( const QPolygon &  
void drawEllipse ( const QRectF & rectangle  
Draws a line defined by line.
```



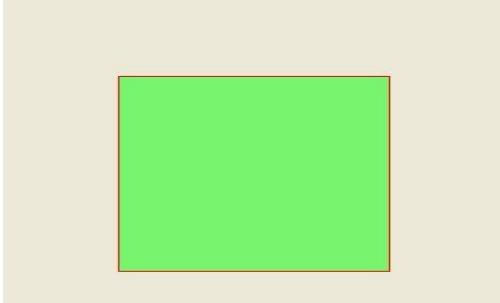
6. 我们下面将几个知识点说明一下，帮助大家更快入门。

将函数改为如下：

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

```
QPen pen; //画笔  
pen.setColor(QColor(255,0,0));  
QBrush brush(QColor(0,255,0,125)); //画刷  
painter.setPen(pen); //添加画笔  
painter.setBrush(brush); //添加画刷  
painter.drawRect(100,100,200,200); //绘制矩形  
}
```

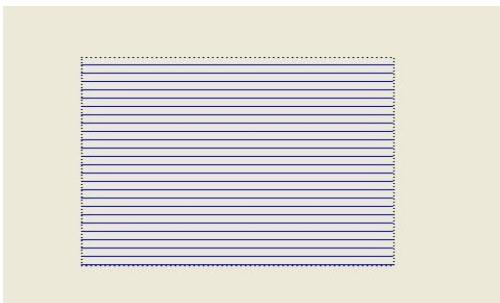
这里的 pen 用来绘制边框，brush 用来进行封闭区域的填充， QColor 类用来提供颜色，我们这里使用了 rgb 方法来生成颜色，即 ( red, green, blue )，它们取值分别是 0-255，例如 ( 255, 0, 0 ) 表示红色，而全 0 表示黑色，全 255 表示白色。后面的 ( 0, 255, 0, 125 )，其中的 125 是透明度 ( alpha ) 设置，其值也是从 0 到 255，0 表示全透明。最后将画笔和画刷添加到 painter 绘制设备中，画出图形。这里的 Rect 是长方形，其中的参数为 ( 100, 100 ) 表示起始坐标，200, 200 表示长和宽。效果如下：



## 7. 其实画笔和画刷也有很多设置，大家可以查看帮助。

```
QPainter painter(this);  
QPen pen(Qt::DotLine);  
QBrush brush(Qt::blue);  
brush.setStyle(Qt::HorPattern);  
painter.setPen(pen);  
painter.setBrush(brush);  
painter.drawRect(100,100,200,200);
```

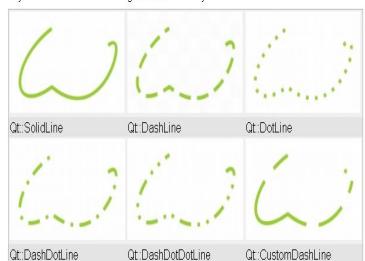
这里我们设置了画笔的风格为点线，画刷的风格为并行横线，效果如下：



在帮助里可以看到所有的风格。

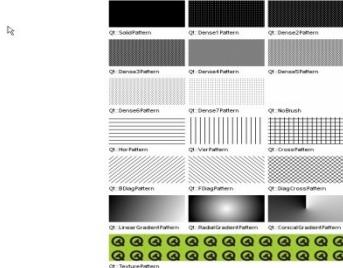
#### enum Qt::PenStyle

This enum type defines the pen styles that can be drawn using QPainter. The styles are:



#### enum Qt::BrushStyle

This enum type defines the brush styles supported by Qt, i.e. the fill pattern of shapes drawn using QPainter.



我们这里用了 Qt::blue, Qt 自定义的几个颜色如下:

#### enum Qt::GlobalColor

Qt's predefined QColor objects:

| Constant        | Value | Description  |
|-----------------|-------|--|
| Qt::white       | 3     | White (#FFFFFF)                                      |
| Qt::black       | 2     | Black (#000000)                                      |
| Qt::red         | 7     | Red (#FF0000)  |
| Qt::darkRed     | 13    | Dark red (#8B0000)                                   |
| Qt::green       | 8     | Green (#008000)                                      |
| Qt::darkGreen   | 14    | Dark green (#008B00)                                 |
| Qt::blue        | 9     | Blue (#0000FF)                                       |
| Qt::darkBlue    | 15    | Dark blue (#00008B)                                  |
| Qt::cyan        | 10    | Cyan (#00FFFF)                                       |
| Qt::darkCyan    | 16    | Dark cyan (#008B8B)                                  |
| Qt::magenta     | 11    | Magenta (#FF00FF)                                    |
| Qt::darkMagenta | 17    | Dark magenta (#8B008B)                               |
| Qt::yellow      | 12    | Yellow (#FFFF00)                                     |
| Qt::darkYellow  | 18    | Dark yellow (#8B8B00)                                |
| Qt::gray        | 5     | Gray (#CCCCCC)                                       |
| Qt::darkGray    | 4     | Dark gray (#8B8B8B)                                  |
| Qt::lightGray   | 6     | Light gray (#EDEDED)                                 |
| Qt::transparent | 19    | a transparent black value (i.e., QColor(0, 0, 0, 0)) |
| Qt::color0      | 0     | 0 pixel value (for bitmaps)                          |
| Qt::color1      | 1     | 1 pixel value (for bitmaps)                          |

See also QColor.

## 8.画弧线，这是帮助里的一个例子。

```
QRectF rectangle(10.0, 20.0, 80.0, 60.0); //矩形
int startAngle = 30 * 16; //起始角度
int spanAngle = 120 * 16; //跨越度数
QPainter painter(this);
painter.drawArc(rectangle, startAngle, spanAngle);
```

这里要说明的是，画弧线时，角度被分成了十六分之一，就是说，要想为 30 度，就得是  $30 \times 16$ 。它有起始角度和跨度，还有位置矩形，要想画出自己想要的弧线，就要有一定的几何知识了。这里就不再祥述。

#### enum Qt::GlobalColor

Qt's predefined QColor objects:

| Constant        | Value | Description  |
|-----------------|-------|--|
| Qt::white       | 3     | White (#FFFFFF)                                      |
| Qt::black       | 2     | Black (#000000)                                      |
| Qt::red         | 7     | Red (#FF0000)  |
| Qt::darkRed     | 13    | Dark red (#8B0000)                                   |
| Qt::green       | 8     | Green (#008000)                                      |
| Qt::darkGreen   | 14    | Dark green (#008B00)                                 |
| Qt::blue        | 9     | Blue (#0000FF)                                       |
| Qt::darkBlue    | 15    | Dark blue (#00008B)                                  |
| Qt::cyan        | 10    | Cyan (#00FFFF)                                       |
| Qt::darkCyan    | 16    | Dark cyan (#008B8B)                                  |
| Qt::magenta     | 11    | Magenta (#FF00FF)                                    |
| Qt::darkMagenta | 17    | Dark magenta (#8B008B)                               |
| Qt::yellow      | 12    | Yellow (#FFFF00)                                     |
| Qt::darkYellow  | 18    | Dark yellow (#8B8B00)                                |
| Qt::gray        | 5     | Gray (#CCCCCC)                                       |
| Qt::darkGray    | 4     | Dark gray (#8B8B8B)                                  |
| Qt::lightGray   | 6     | Light gray (#EDEDED)                                 |
| Qt::transparent | 19    | a transparent black value (i.e., QColor(0, 0, 0, 0)) |
| Qt::color0      | 0     | 0 pixel value (for bitmaps)                          |
| Qt::color1      | 1     | 1 pixel value (for bitmaps)                          |

See also QColor.

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 4,917 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十二、Qt 2D 绘图（二）渐变填充

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

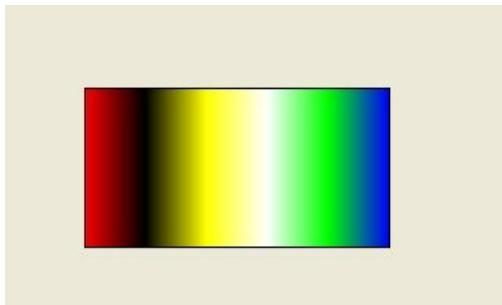
在 **qt** 中提供了三种渐变方式，分别是线性渐变，圆形渐变和圆锥渐变。如果能熟练应用它们，就能设计出炫目的填充效果。

线性渐变：

1. 更改函数如下：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QLinearGradient linearGradient(100,150,300,150);  
    //从点 ( 100, 150 ) 开始到点 ( 300, 150 ) 结束，确定一条直线  
    linearGradient.setColorAt(0,Qt::red);  
    linearGradient.setColorAt(0.2,Qt::black);  
    linearGradient.setColorAt(0.4,Qt::yellow);  
    linearGradient.setColorAt(0.6,Qt::white);  
    linearGradient.setColorAt(0.8,Qt::green);  
    linearGradient.setColorAt(1,Qt::blue);  
    //将直线开始点设为 0，终点设为 1，然后分段设置颜色  
    painter.setBrush(linearGradient);  
    painter.drawRect(100,100,200,100);  
    //绘制矩形，线性渐变线正好在矩形的水平中心线上  
}
```

效果如下：

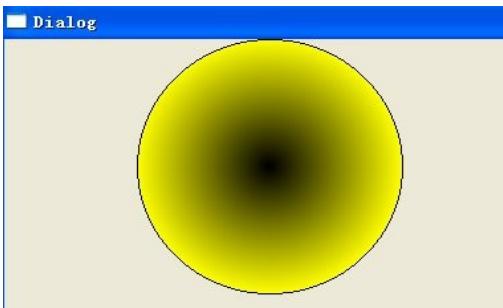


圆形渐变：

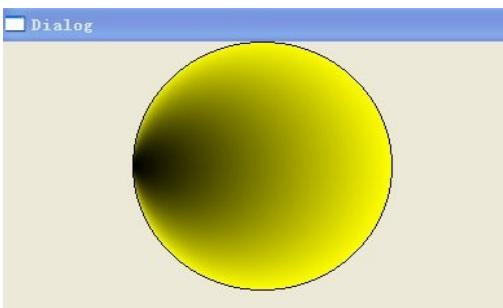
1. 更改函数内容如下：

```
QRadialGradient radialGradient(200,100,100,200,100);  
//其中参数分别为圆形渐变的圆心 ( 200, 100 )，半径 100，和焦点 ( 200, 100 )  
//这里让焦点和圆心重合，从而形成从圆心向外渐变的效果  
radialGradient.setColorAt(0,Qt::black);  
radialGradient.setColorAt(1,Qt::yellow);  
//渐变从焦点向整个圆进行，焦点为起始点 0，圆的边界为 1  
QPainter painter(this);  
painter.setBrush(radialGradient);  
painter.drawEllipse(100,0,200,200);  
//绘制圆，让它正好和上面的圆形渐变的圆重合
```

效果如下：



**2.**要想改变填充的效果，只需要改变焦点的位置和渐变的颜色位置即可。  
改变焦点位置：`QRadialGradient radialGradient(200,100,100,100,100);`  
效果如下：



**锥形渐变：**

**1.**更改函数内容如下：

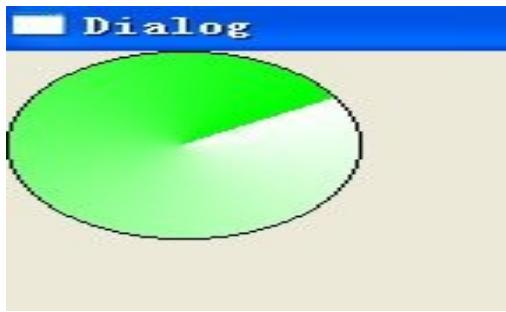
//圆锥渐变

```
QConicalGradient conicalGradient(50,50,0);
//圆心为 ( 50, 50 ) , 开始角度为 0
conicalGradient.setColorAt(0,Qt::green);
conicalGradient.setColorAt(1,Qt::white);
//从圆心的 0 度角开始逆时针填充
QPainter painter(this);
painter.setBrush(conicalGradient);
painter.drawEllipse(0,0,100,100);
```

效果如下：



**2.**可以更改开始角度，来改变填充效果  
`QConicalGradient conicalGradient(50,50,30);`  
开始角度设置为 30 度，效果如下：



分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 2,902 views  
Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十三、Qt 2D 绘图（三）绘制文字

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

接着上一次的教程，这次我们学习在窗体上绘制文字。

## 1. 绘制最简单的文字。

我们更改重绘函数如下：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.drawText(100,100,"yafeilinux");  
}
```

我们在(100, 100)的位置显示了一行文字，效果如下。



**2.**为了更好的控制字体的位置。我们使用另一个构造函数。在帮助里查看**drawText**, 如下。

```
void QPainter::drawText( const QRectF & rectangle, int flags, const QString & text, QRectF * boundingRect = 0 )
```

This is an overloaded function.

Draws the given text within the provided rectangle.

```
Qt by  
Ruebeck
```

```
QPainter painter(this);  
painter.drawText(rect, Qt::AlignCenter, tr("Qt by Nokia"));
```

The `boundingRect` (if not null) is set to the what the bounding rectangle should be in order to enclose the whole text. The `flags` argument consists of one or more of the following flags:

- `Qt::AlignLeft`
- `Qt::AlignRight`
- `Qt::AlignHCenter`
- `Qt::AlignJustify`
- `Qt::AlignTop`
- `Qt::AlignBottom`
- `Qt::TextWordBreak`

这里我们看到了构造函数的原型和例子。其中的 flags 参数可以控制字体在矩形中的位置。我们更改函数内容如下。

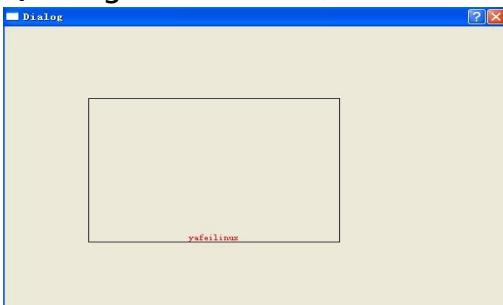
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QRectF ff(100,100,300,200);  
    //设置一个矩形  
    painter.drawRect(ff);  
    //为了更直观地看到字体的位置，我们绘制  
    painter.setPen(QColor(Qt::red));  
    //设置画笔颜色为红色  
    painter.drawText(ff,Qt::AlignHCenter)  
    //我们这里先让字体水平居中  
}
```

效果如下。

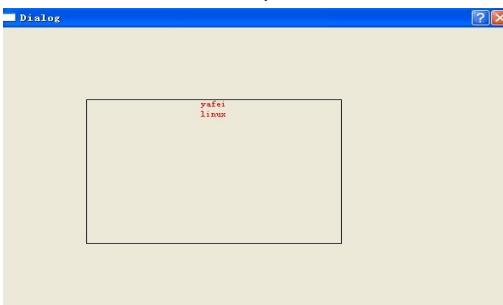


可以看到字符串是在最上面水平居中的。如果想让其在矩形正中间，我们可以使用 Qt::AlignCenter。

这里我们也可以使用两个枚举变量进行按位与操作，例如可以使用 Qt::AlignBottom | Qt::AlignHCenter 实现让文字显示在矩形下面的正中间。效果如下。



对于较长的字符串，我们也可以利用“\n”进行换行，例如“yafei\nlinux”。效果如下。



**3.如果要使文字更美观，我们就需要使用 QFont 类来改变字体。先在帮助中查看一下这个类。**

**QFont Class Reference**  
[QtGui module]

The QFont class specifies a font used for drawing text. More...

Include <QtGui>

**Note:** All functions in this class are reentrant.

- List of all members, including inherited members
- Qt 5 support members

**Public Types**

```
enum Capitalization { NoneCase, AllUpperCase, AllLowerCase, SmallCaps, Capitalize }
enum SpacingType { PercentageSpacing, AbsoluteSpacing }
enum Stretch { UltraCondensed, ExtraCondensed, Condensed, SemiCondensed, ... , UltraExpanded }
enum Style { StyleNormal, StyleItalic, StyleOblique }
enum StyleStrategy { PreferDefault, PreferSystem, PreferDevice, PreferOutline, ... , PreferQuality }
enum Weight { Light, Normal, DemiBold, Bold, Black }
```

**Public Functions**

```
QFont();
QFont(const QFont & family, int pointSize = -1, int weight = -1, bool italic = false)
QFont(const QFont & font, QPaintDevice * pd)
QFont(const QFont & font)
~QFont()
void reset();
```

可以看到它有好几个枚举变量来设置字体。下面的例子我们对主要的几个选项进行演示。  
更改函数如下。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QFont font("Arial", 20, QFont::Bold, true);
    //设置字体的类型，大小，加粗，斜体
    font.setUnderline(true);
    //设置下划线
    font.setOverline(true);
```

```

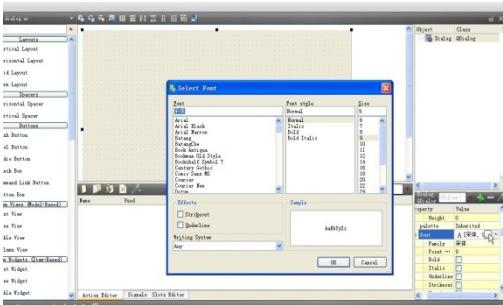
//设置上划线
font.setCapitalization(QFont::SmallCaps);
//设置大小写
font.setLetterSpacing(QFont::AbsoluteSpacing,5);
//设置间距
QPainter painter(this);
painter.setFont(font);
//添加字体
QRectF ff(100,100,300,200);
painter.drawRect(ff);
painter.setPen(QColor(Qt::red));
painter.drawText(ff.Qt::AlignCenter,"yafeilinux");
}

```

效果如下。



这里的所有字体我们可以在设计器中进行查看。如下。



分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 3,062 views  
 Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十四、Qt 2D 绘图（四）绘制路径

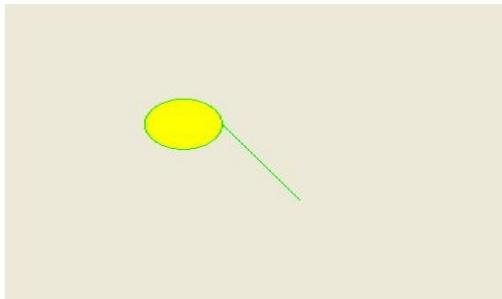
本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

接着上一次的教程，这次我们学习在窗体上绘制路径。**QPainterPath** 这个类很有用，这里我们只是说明它最常使用的功能，更深入的以后再讲。

1. 我们更改 **paintEvent** 函数如下。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainterPath path;  
    path.addEllipse(100,100,50,50);  
    path.lineTo(200,200);  
    QPainter painter(this);  
    painter.setPen(Qt::green);  
    painter.setBrush(Qt::yellow);  
    painter.drawPath(path);  
}
```

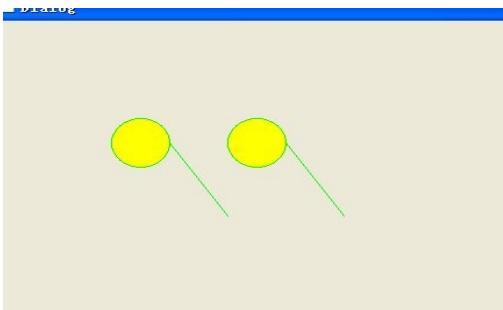
这里我们新建了一个 **painterPath** 对象，并加入了一个圆和一条线。然后绘制这个路径。效果如下。



2. 上面绘制圆和直线都有对应的函数啊，为什么还要加入一个 **painterPath** 呢？我们再添加几行代码，你就会发现它的用途了。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainterPath path;  
    path.addEllipse(100,100,50,50);  
    path.lineTo(200,200);  
    QPainter painter(this);  
    painter.setPen(Qt::green);  
    painter.setBrush(Qt::yellow);  
    painter.drawPath(path);  
    QPainterPath path2;  
    path2.addPath(path);  
    path2.translate(100,0);  
    painter.drawPath(path2);  
}
```

效果如下。



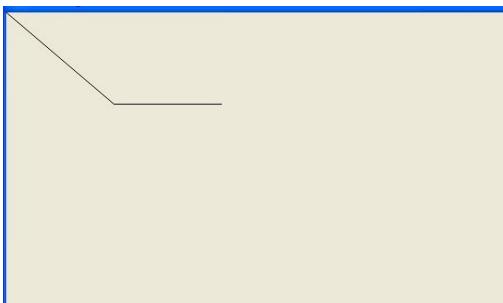
这里我们又新建了一个 painterPath 对象 path2，并将以前的 path 添加到它上面，然后我们更改了原点坐标为 (100, 0)，这时你发现我们复制了以前的图形。[这也就是 painterPath 类最主要的用途，它能保存你已经绘制好的图形。](#)

**3.**这里我们应该注意的是绘制完一个图形后，当前的位置在哪里。

例如：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.lineTo(100,100);
    path.lineTo(200,100);
    QPainter painter(this);
    painter.drawPath(path);
}
```

效果如下。

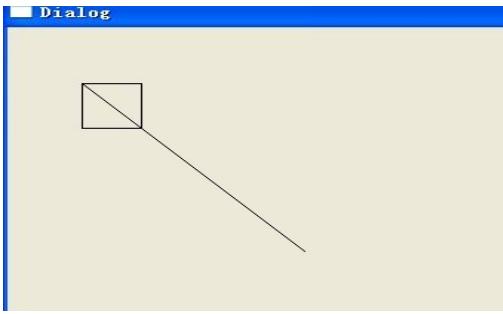


可以看到默认是从原点 (0, 0) 开始绘图的，当画完第一条直线后，当前点应该在 (100, 100) 处，然后画第二条直线。

再如：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.addRect(50,50,40,40);
    path.lineTo(200,200);
    QPainter painter(this);
    painter.drawPath(path);
}
```

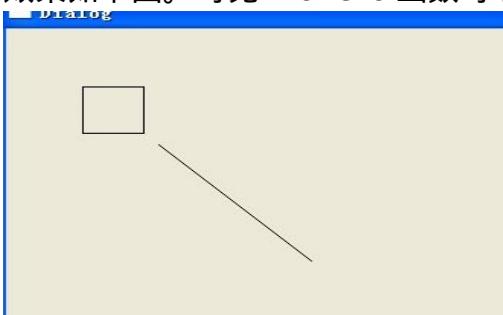
效果如下。可见画完矩形后，当前点在矩形的左上角顶点，然后从这里开始画直线。



我们可以自己改变当前点的位置。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.addRect(50,50,40,40);
    path.moveTo(100,100);
    path.lineTo(200,200);
    QPainter painter(this);
    painter.drawPath(path);
}
```

效果如下图。可见 `moveTo` 函数可以改变当前点的位置。



这里我们只讲解了绘制路径类最简单的应用，其实这个类很有用，利用它可以设计出很多特效。有兴趣的朋友可以查看一下它的帮助。因为我们这里只是简介，所以不再深入研究。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 2,714 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十五、Qt 2D 绘图（五）显示图片

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

现在我们来实现在窗口上显示图片，并学习怎样将图片进行平移，缩放，旋转和扭曲。这里我们是利用 **QPixmap** 类来实现图片显示的。

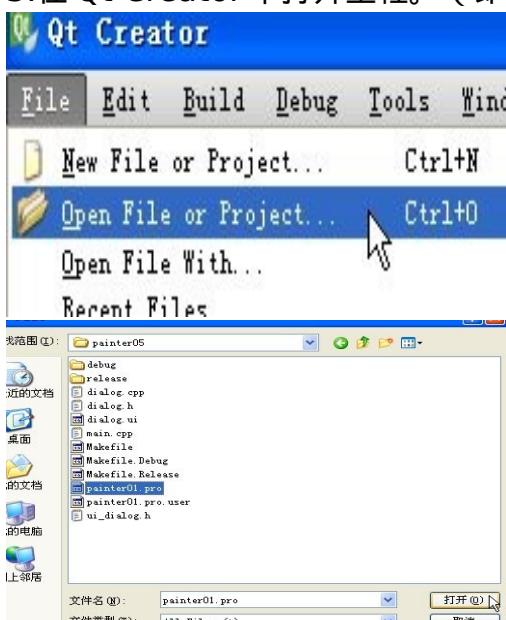
### 一、利用 **QPixmap** 显示图片。

1. 将以前的工程文件夹进行复制备份，我们这里将工程文件夹改名为 painter05。（以前已经说过，经常备份工程目录，是个很好的习惯）

2. 在工程文件夹的 debug 文件夹中新建文件夹，我这里命名为 images，用来存放要用的图片。我这里放了一张 linux.jpg 的图片。如下图所示。



3. 在 Qt Creator 中打开工程。（即打开工程文件夹中的.pro 文件），如图。



4. 将 dialog.cpp 文件中的 paintEvent() 函数更改如下。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load("images/linux.jpg");
    painter.drawPixmap(0,0,100,100,pix);
}
```

这里新建 **QPixmap** 类对象，并为其添加图片，然后在以 (0, 0) 点开始的宽和高都为 100 的矩形中显示该图片。你可以改变矩形的大小，看一下效果啊。最终程序运行效果如下。



(说明：下面的操作都会和坐标有关，这里请先进行操作，我们在下一节将会讲解坐标系统。)

## 二、利用更改坐标原点实现平移。

Qpainter 类中的 translate() 函数实现坐标原点的改变，改变原点后，此点将会成为新的原点 (0, 0)；

例如：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix;  
    pix.load("images/linux.jpg");  
    painter.drawPixmap(0,0,100,100,pix);  
    painter.translate(100,100); //将 (100, 100) 设为坐标原点  
    painter.drawPixmap(0,0,100,100,pix);  
}
```

这里将 (100, 100) 设置为了新的坐标原点，所以下面在 (0, 0) 点贴图，就相当于在以前的 (100, 100) 点贴图。效果如下。



## 三、实现图片的缩放。

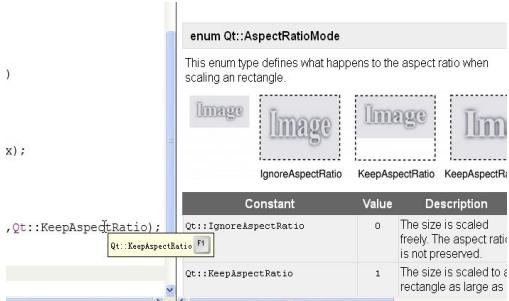
我们可以使用 QPixmap 类中的 scaled() 函数来实现图片的放大和缩小。

例如：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix;  
    pix.load("images/linux.jpg");  
    painter.drawPixmap(0,0,100,100,pix);  
    qreal width = pix.width(); //获得以前图片的宽和高  
    qreal height = pix.height();  
    pix = pix.scaled(width*2,height*2,Qt::KeepAspectRatio);  
    //将图片的宽和高都扩大两倍，并且在给定的矩形内保持宽高的比值
```

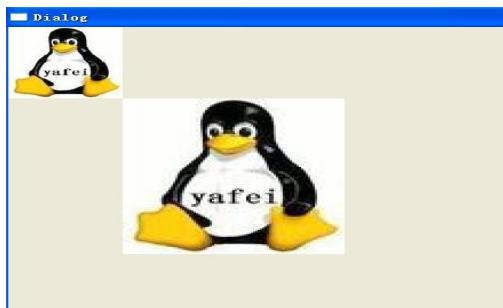
```
    painter.drawPixmap(100,100,pix);
}
```

其中参数 Qt::KeepAspectRatio，是图片缩放的方式。我们可以查看其帮助。将鼠标指针放到该代码上，当出现 F1 提示时，按下 F1 键，这时就可以查看其帮助了。当然我们也可以直接在帮助里查找该代码。



这是个枚举变量，这里有三个值，只看其图片就可大致明白，Qt::IgnoreAspectRatio 是不保持图片的长宽比，Qt::KeepAspectRatio 是在给定的矩形中保持长宽比，最后一个也是保持长宽比，但可能超出给定的矩形。这里给定的矩形是由我们显示图片时给定的参数决定的，例如 painter.drawPixmap(0,0,100,100,pix);就是在以 (0, 0) 点为起始点的宽和高都是 100 的矩形中。

程序运行效果如下。



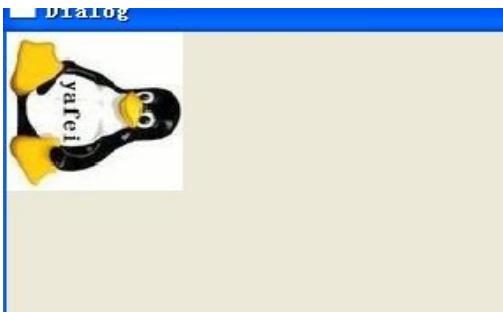
#### 四、实现图片的旋转。

旋转使用的是 QPainter 类的 rotate() 函数，它默认是以原点为中心进行旋转的。我们要改变旋转的中心，可以使用前面讲到的 translate() 函数完成。

例如：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load("images/linux.jpg");
    painter.translate(50,50); //让图片的中心作为旋转的中心
    painter.rotate(90); //顺时针旋转 90 度
    painter.translate(-50,-50); //使原点复原
    painter.drawPixmap(0,0,100,100,pix);
}
```

这里必须先改变旋转中心，然后再旋转，然后再将原点复原，才能达到想要的效果。  
运行程序，效果如下。



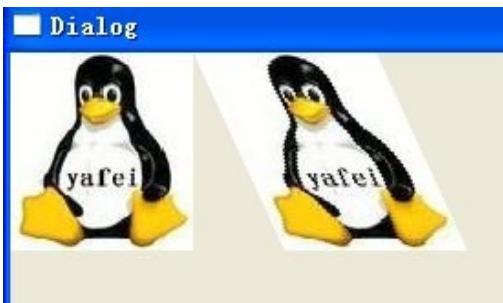
## 五、实现图片的扭曲。

实现图片的扭曲，是使用的 QPainter 类的 shear(qreal sh, qreal sv) 函数完成的。它有两个参数，前面的参数实现横向变形，后面的参数实现纵向变形。当它们的值为 0 时，表示不扭曲。

例如：

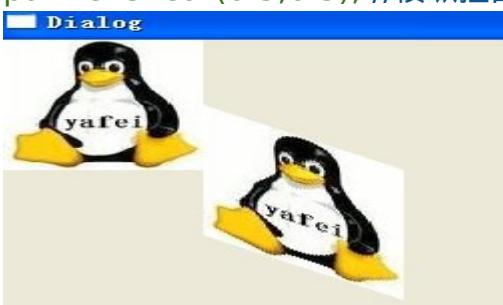
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix;  
    pix.load("images/linux.jpg");  
    painter.drawPixmap(0,0,100,100,pix);  
    painter.shear(0.5,0); //横向扭曲  
    painter.drawPixmap(100,0,100,100,pix);  
}
```

效果如下：



其他扭曲效果：

```
painter.shear(0,0.5); //纵向扭曲  
painter.shear(0.5,0.5); //横纵扭曲
```





图片形状的变化，其实就是利用坐标系的变化来实现的。我们在下一节中将会讲解坐标系统。这一节中的几个函数，我们可以在其帮助文件中查看其详细解释。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 4,777 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十六、Qt 2D 绘图（六）坐标系统

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

前面一节我们讲解了图片的显示，其中很多都用到了坐标的变化，这一节我们简单讲一下 Qt 的坐标系统，其实也还是主要讲上一节的那几个函数。这里我们先讲解一下 Qt 的坐标系，然后讲解那几个函数，它们分别是：

**translate()** 函数，进行平移变换；**scale()** 函数，进行比例变换；**rotate()** 函数，进行旋转变换；**shear()** 函数，进行扭曲变换。

最后介绍两个有用的函数 **save()** 和 **restore()**，利用它们来保存和弹出坐标系的状态，从而实现快速利用几个变换来绘图。

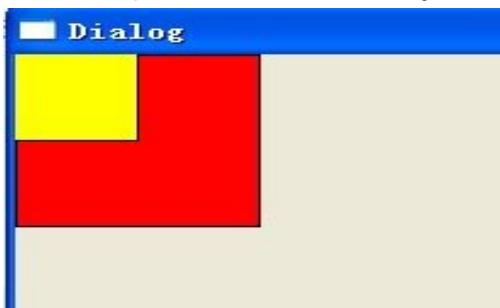
### 一、坐标系简介。

Qt 中每一个窗口都有一个坐标系，默认的，窗口左上角为坐标原点，然后水平向右依次增大，水平向左依次减小，垂直向下依次增大，垂直向上依次减小。原点即为 (0, 0) 点，然后以像素为单位增减。

例如：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.setBrush(Qt::red);  
    painter.drawRect(0,0,100,100);  
    painter.setBrush(Qt::yellow);  
    painter.drawRect(-50,-50,100,100);  
}
```

我们先在原点 (0, 0) 绘制了一个长宽都是 100 像素的红色矩形，又在 (-50, -50) 点绘制了一个同样大小的黄色矩形。可以看到，我们只能看到黄色矩形的一部分。效果如下图。



### 二、坐标系变换。

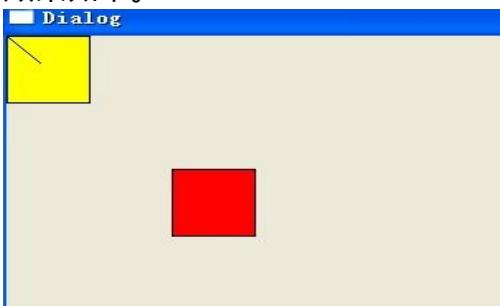
坐标系变换是利用变换矩阵来进行的，我们可以利用 **QTransform** 类来设置变换矩阵，因为一般我们不需要进行更改，所以这里不在涉及。下面我们只是对坐标系的平移，缩放，旋转，扭曲等应用进行介绍。

#### 1. 利用 **translate()** 函数进行平移变换。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.setBrush(Qt::yellow);  
    painter.drawRect(0,0,50,50);  
    painter.translate(100,100); //将点 (100, 100) 设为原点  
    painter.setBrush(Qt::red);  
    painter.drawRect(0,0,50,50);  
    painter.translate(-100,-100);
```

```
    painter.drawLine(0,0,20,20);  
}
```

效果如下。



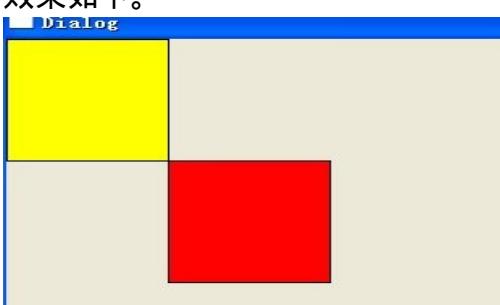
这里将 (100, 100) 点作为了原点，所以此时 (100, 100) 就是 (0, 0) 点，以前的 (0, 0) 点就是 (-100, -100) 点。要想使原来的 (0, 0) 点重新成为原点，就是将 (-100, -100) 设为原点。

## 2. 利用 **scale()** 函数进行比例变换，实现缩放效果。

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

```
{  
    QPainter painter(this);  
    painter.setBrush(Qt::yellow);  
    painter.drawRect(0,0,100,100);  
    painter.scale(2,2); //放大两倍  
    painter.setBrush(Qt::red);  
    painter.drawRect(50,50,50,50);  
}
```

效果如下。



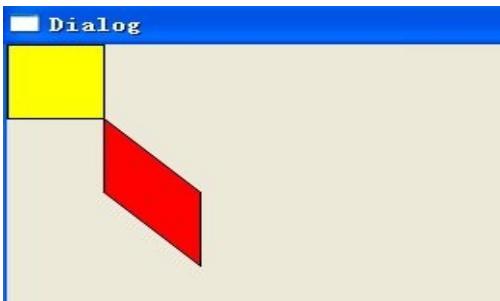
可以看到，`painter.scale(2,2)`，是将横纵坐标都扩大了两倍，现在的 (50, 50) 点就相当于以前的 (100, 100) 点。

## 3. 利用 **shear()** 函数就行扭曲变换。

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

```
{  
    QPainter painter(this);  
    painter.setBrush(Qt::yellow);  
    painter.drawRect(0,0,50,50);  
    painter.shear(0,1); //纵向扭曲变形  
    painter.setBrush(Qt::red);  
    painter.drawRect(50,0,50,50);  
}
```

效果如下。

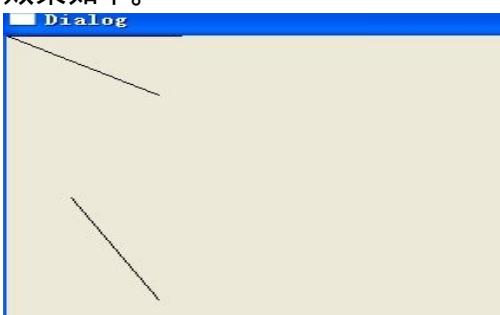


这里，`painter.shear(0,1)`，是对纵向进行扭曲，0 表示不扭曲，当将第一个 0 更改时就会对横向进行扭曲，关于扭曲变换到底是什么效果，你观察一下是很容易发现的。

#### 4. 利用 `rotate()` 函数进行比例变换，实现缩放效果。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.drawLine(0,0,100,0);  
    painter.rotate(30); //以原点为中心，顺时针旋转 30 度  
    painter.drawLine(0,0,100,0);  
    painter.translate(100,100);  
    painter.rotate(30);  
    painter.drawLine(0,0,100,0);  
}
```

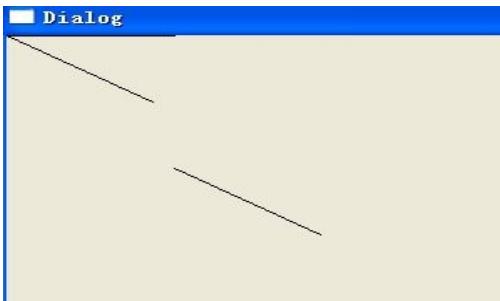
效果如下。



因为默认的 `rotate()` 函数是以原点为中心进行顺时针旋转的，所以我们要想使其以其他点为中心进行旋转，就要先进行原点的变换。这里的 `painter.translate(100,100)` 将 (100, 100) 设置为新的原点，想让直线以其为中心进行旋转，可是你已经发现效果并非如此。是什么原因呢？我们添加一条语句，如下：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.drawLine(0,0,100,0);  
    painter.rotate(30); //以原点为中心，顺时针旋转 30 度  
    painter.drawLine(0,0,100,0);  
    painter.rotate(-30);  
    painter.translate(100,100);  
    painter.rotate(30);  
    painter.drawLine(0,0,100,0);  
}
```

效果如下。



这时就是我们想要的效果了。我们加的一句代码为 `painter.rotate(-30)`，这是因为前面已经将坐标旋转了 30 度，我们需要将其再旋转回去，才能是以前正常的坐标系统。不光这个函数如此，这里介绍的这几个函数均如此，所以很容易出错。下面我们将利用两个函数来很好的解决这个问题。

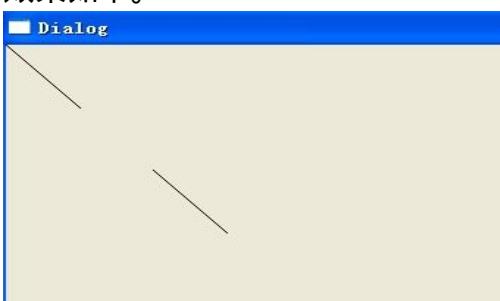
### 三、坐标系状态的保护。

我们可以先利用 `save()` 函数来保存坐标系现在的状态，然后进行变换操作，操作完之后，再用 `restore()` 函数将以前的坐标系状态恢复，其实就是一个入栈和出栈的操作。

例如：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.save(); //保存坐标系状态  
    painter.translate(100,100);  
    painter.drawLine(0,0,50,50);  
    painter.restore(); //恢复以前的坐标系状态  
    painter.drawLine(0,0,50,50);  
}
```

效果如下。



利用好这两个函数，可以实现快速的坐标系切换，绘制出不同的图形。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,196 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十七、Qt 2D 绘图（七）Qt 坐标系统深入

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

接着上面一节，前面只是很简单的讲解了一下 **Qt** 坐标系统的概念，通过对几个函数的应用，我们应该已经对 **Qt** 的坐标系统有了一个模糊的认识。那么现在就来让我们更深入地研究一下 **Qt** 窗口的坐标。希望大家把这一节的例子亲手做一下，不要被我所说的东西搞晕了！

我们还是在以前的工程中进行操作。

获得坐标信息：

为了更清楚地获得坐标信息，我们这里利用鼠标事件，让鼠标点击左键时输出该点的坐标信息。

**1.**在工程中的 **dialog.h** 文件中添加代码。

添加头文件： `#include <QMouseEvent>`

在 `public` 中添加函数声明： `void mousePressEvent(QMouseEvent *);`

然后到 **dialog.cpp** 文件中：

添加头文件： `#include <QDebug>`

定义函数：

```
void Dialog::mousePressEvent(QMouseEvent *event)
{
    qDebug() << event->pos();
}
```

这里应用了 `qDebug()` 函数，利用该函数可以在程序运行时将程序中的一些信息输出，在 Qt Creator 中会将信息输出到其下面的 Application Output 窗口。这个函数很有用，在进行简单的程序调试时，都是利用该函数进行的。我们这里利用它将鼠标指针的坐标值输出出来。

**2.**然后更改重绘事件函数。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawRect(0,0,50,50);
}
```

我们绘制了一个左上顶点为  $(0, 0)$ ，宽和高都是 50 的矩形。

**3.**这时运行程序。并在绘制的矩形左上顶点点击一下鼠标左键。效果如下。（点击可看大图）



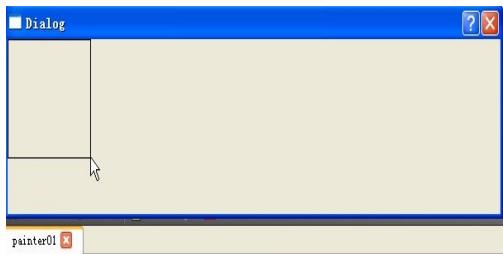
因为鼠标点的不够准确，所以输出的是  $(1, 0)$ ，我们可以认为左上角就是原点  $(0, 0)$  点。你可以再点击一下矩形的右下角，它的坐标应该是  $(50, 50)$ 。这个方法掌握了以后，我们就开始研究这些坐标了。

研究放大后的坐标

**1.**我们现在进行放大操作，然后查看其坐标的变化。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.scale(2,2); //横纵坐标都扩大 2 倍
    painter.drawRect(0,0,50,50);
}
```

我们将横纵坐标都扩大 2 倍，然后运行程序，查看效果：



我们点击矩形右下顶点，是 (100, 100)，比以前的 (50, 50) 扩大了 2 倍。

### 研究 **QPixmap** 或 **QImage** 的坐标

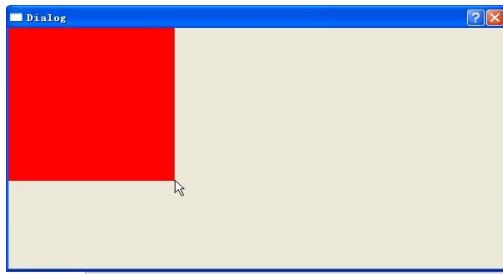
对于 **QWidget**, **QPixmap** 或 **QImage** 等都是绘图设备，我们都可以在其上利用 **Painter** 进行绘图。现在我们研究一下 **QPixmap** 的坐标 (**QImage** 与其效果相同)。

#### 1. 我们更改重绘事件函数如下。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix(200,200);  
    pix.fill(Qt::red); //背景填充为红色  
    painter.drawPixmap(0,0,pix);  
}
```

这里新建了一个宽、高都是 200 像素的 **QPixmap** 类对象，并将其背景颜色设置为红色，然后从窗口的原点 (0, 0) 点添加该 **QPixmap** 类对象。为了表述方便，在下面我们将这个 **QPixmap** 类对象 **pix** 称为画布。

我们运行程序，并在画布的左上角和右下角分别点击一下，效果如下：

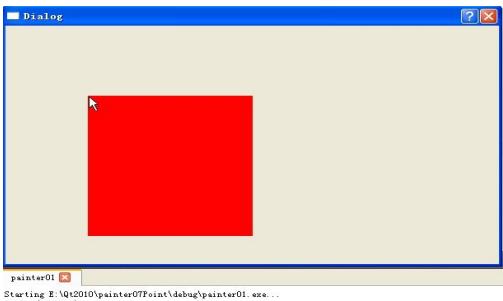


可以看到其左上角为 (0, 0) 点，右下角为 (200, 200) 点，是没有问题的。

#### 2. 我们再将函数更改如下。

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix(200,200);  
    pix.fill(Qt::red); //背景填充为红色  
    painter.drawPixmap(100,100,pix);  
}
```

这时我们从窗口的 (100, 100) 点添加该画布，那么此时我们再点击画布的右上角，其坐标会是多少呢？



可以看到，它是(100, 100)，没错，这是窗口上的坐标，那么这是不是画布上的坐标呢？

### 3. 我们接着更改函数。

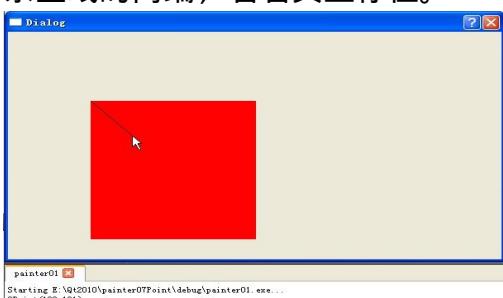
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix(200,200);  
    pix.fill(Qt::red); //背景填充为红色  
    QPainter pp(&pix); //新建 QPainter 类对象，在 pix 上进行绘图  
    pp.drawLine(0,0,50,50); //在 pix 上的(0, 0)点和(50, 50)点之间绘制直线  
    painter.drawPixmap(100,100,pix);  
}
```

这里我们又新建了一个 QPainter 类对象 pp，其中 pp(&pix) 表明，pp 所进行的绘图都是在画布 pix 上进行的。

现在先说明一下：

QPainter painter(this)，this 就表明了是在窗口上进行绘图，所以利用 painter 进行的绘图都是在窗口上的，painter 进行的坐标变化，是变化的窗口的坐标系；而利用 pp 进行的绘图都是在画布上进行的，如果它进行坐标变化，就是变化的画布的坐标系。

我们在画布上的(0, 0)点和(50, 50)点之间绘制了一条直线。这时运行程序，点击这条直线的两端，看看其坐标值。



结果是直线的两端的坐标分别是(100, 100)，(150, 150)。我们从中可以得出这样的结论：

第一，QWidget 和 QPixmap 各有一套坐标系统，它们互不影响。可以看到，无论画布在窗口的什么位置，它的坐标原点依然在左上角，为(0, 0)点，没有变。

第二，我们所得到的鼠标指针的坐标值是窗口提供的，不是画布的坐标。

下面我们继续研究：

### 4. 比较下面两个例子。

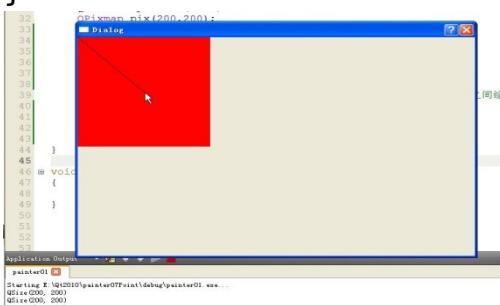
例子一：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix(200,200);  
    pix.fill(Qt::red);  
    painter.drawPixmap(100,100,pix);  
}
```

```

qDebug() << pix.size(); //放大前输出 pix 的大小
pix.fill(Qt::red);
QPainter pp(&pix);
pp.scale(2,2); //pix 的坐标扩大 2 倍
pp.drawLine(0,0,50,50); //在 pix 上的 (0, 0) 点和 (50, 50) 点之间绘制直线
qDebug() << pix.size(); //放大后输出 pix 的大小
painter.drawPixmap(0,0,pix);
}

```

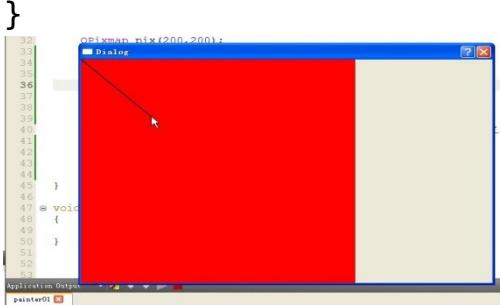


## 例子二：

```

void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    qDebug() << pix.size(); //放大前输出 pix 的大小
    painter.scale(2,2); //窗口坐标扩大 2 倍
    pix.fill(Qt::red);
    QPainter pp(&pix);
    pp.drawLine(0,0,50,50); //在 pix 上的 (0, 0) 点和 (50, 50) 点之间绘制直线
    qDebug() << pix.size(); //放大后输出 pix 的大小
    painter.drawPixmap(0,0,pix);
}

```



两个例子中都使直线的长度扩大了两倍，但是第一个例子是扩大的画布的坐标系，第二个例子是扩大的窗口的坐标系，你可以看一下它们的效果。

**你仔细看一下输出，两个例子中画布的大小都没有变。**

如果你看过了我写的那个绘图软件的教程（[链接过去](#)），现在你就能明白我在其中讲“问题一”时说的意思了：虽然画布看起来是大了，但是其大小并没有变，其中坐标也没有变。变的是像素的大小或者说像素间的距离。

但是，有一点你一定要搞明白，这只是在 QPixmap 与 QWidget 结合时才出现的，是相对的说法。其实利用 scale() 函数是会让坐标变化的，我们在开始的例子已经证明了。

结论：

现在是不是已经很乱了，一会儿是窗口，一会儿是画布，一会儿坐标变化，一会儿又不变了，到底怎么样呢？

其实只需记住一句话：

**所有的绘图设备都有自己的坐标系统，它们互不影响。**

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 2,880 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十八、Qt 2D 绘图（八）涂鸦板

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

上面一节我们深入分析了一下 Qt 的坐标系统，这一节我们在前面程序的基础上稍加改动，设计一个涂鸦板程序。

简单的涂鸦板：

**1.** 我们再在程序中添加函数。

我们在 dialog.h 里的 public 中再添加鼠标移动事件和鼠标释放事件的函数声明：

```
void mouseMoveEvent(QMouseEvent *);  
void mouseReleaseEvent(QMouseEvent *);
```

在 private 中添加变量声明：

```
QPixmap pix;  
QPoint lastPoint;  
QPoint endPoint;
```

因为在函数里声明的 QPixmap 类对象是临时变量，不能存储以前的值，所以为了实现保留上次的绘画结果，我们需要将其设为全局变量。

后两个 QPoint 变量存储鼠标指针的两个坐标值，我们需要用这两个坐标值完成绘图。

**2.** 在 **dialog.cpp** 中进行修改。

在构造函数里进行变量初始化。

```
resize(600,500); //窗口大小设置为 600*500  
pix = QPixmap(200,200);  
pix.fill(Qt::white);
```

然后进行其他几个函数的定义：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter pp(&pix);  
    pp.drawLine(lastPoint,endPoint); //根据鼠标指针前后两个位置就行绘制直线  
    lastPoint = endPoint; //让前一个坐标值等于后一个坐标值，这样就能实现画出连续的  
线  
    QPainter painter(this);  
    painter.drawPixmap(0,0,pix);  
}  
void Dialog::mousePressEvent(QMouseEvent *event)  
{  
    if(event->button() == Qt::LeftButton) //鼠标左键按下  
        lastPoint = event->pos();  
}  
void Dialog::mouseMoveEvent(QMouseEvent *event)  
{  
    if(event->buttons() & Qt::LeftButton) //鼠标左键按下的同时移动鼠标  
    {  
        endPoint = event->pos();  
        update();  
    }  
}  
void Dialog::mouseReleaseEvent(QMouseEvent *event)  
{  
    if(event->button() == Qt::LeftButton) //鼠标左键释放
```

```
{  
    endPoint = event->pos();  
    update();  
}  
}
```

这里的 update() 函数，是进行界面重绘，执行该函数时就会执行那个重绘事件函数。

3. 这时运行程序，效果如下。（[点击图片可将其放大](#)）



这样简单的涂鸦板程序就完成了。下面我们进行放大后的涂鸦。

放大后再进行涂鸦：

1. 添加放大按钮。

在 dialog.h 中添加头文件声明： #include <QPushButton>

在 private 中添加变量声明：

```
int scale;
```

```
QPushButton *pushBtn;
```

然后再在下面写上按钮的槽函数声明：

```
private slots:
```

```
    void zoomIn();
```

2. 在 dialog.cpp 中进行更改。

在构造函数里添加如下代码：

```
scale = 1; //设置初始放大倍数为 1，即不放大
```

```
pushBtn = new QPushButton(this); //新建按钮对象
```

```
pushBtn->setText(tr("zoomIn")); //设置按钮显示文本
```

```
pushBtn->move(500,450); //设置按钮放置位置
```

```
connect(pushBtn,SIGNAL(clicked()),this,SLOT(zoomIn())); //对按钮的单击事件和其槽  
函数进行关联
```

这里我们利用代码添加了一个按钮对象，用它来实现放大操作。

再在构造函数以外进行 zoomIn() 函数的定义：

```
void Dialog::zoomIn() //按钮单击事件的槽函数
```

```
{  
    scale *=2;  
    update();  
}
```

3. 通过上一节的学习，我们应该已经知道想让画布的内容放大有两个办法，一个是直接放大画布的坐标，一个是放大窗口的坐标。

我们主要讲解放大窗口坐标。

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

```
{
```

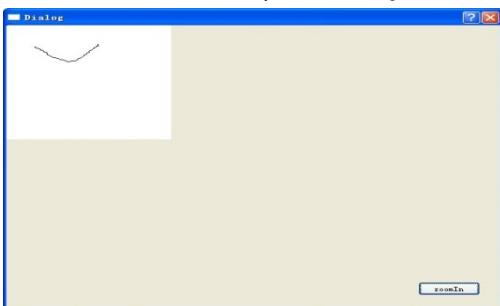
```

    QPainter pp(&pix);
    pp.drawLine(lastPoint,endPoint); //根据鼠标指针前后两个位置就行绘制直线
    lastPoint = endPoint; //让前一个坐标值等于后一个坐标值，这样就能实现画出连续的
线
    QPainter painter(this);
    painter.scale(scale,scale); //进行放大操作
    painter.drawPixmap(0,0,pix);
}

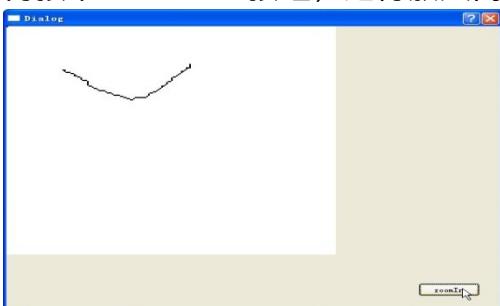
```

这时运行程序。

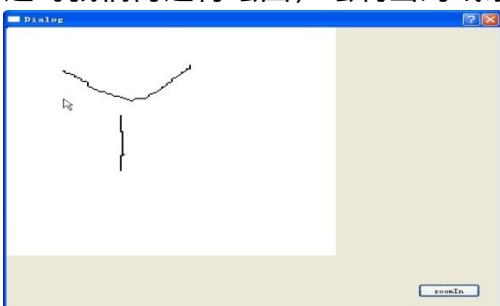
先随意画一个图形，如下图。



再按下“zoomIn”按钮，进行放大两倍。可以看到图片放大了，效果如下。



这时我们再进行绘图，绘制出的线条已经不能和鼠标指针的轨迹重合了。效果如下图。



有了前面一节的知识，我们就不难理解出现这个问题的原因了。窗口的坐标扩大了，但是画布的坐标并没有扩大，而我们画图用的坐标值是鼠标指针的，鼠标指针又是获取的窗口的坐标值。现在窗口和画布的同一点的坐标并不相等，所以就出现了这样的问题。

其实解决办法很简单，窗口放大了多少倍，就将获得的鼠标指针的坐标值缩小多少倍就行了。

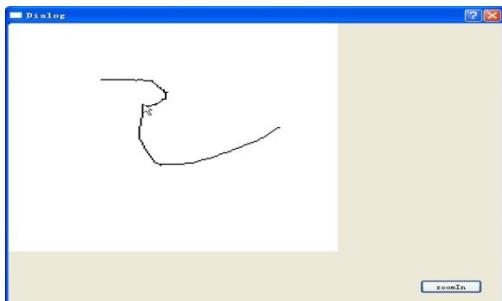
```

void Dialog::paintEvent(QPaintEvent *)
{
    QPainter pp(&pix);
    pp.drawLine(lastPoint/scale,endPoint/scale);
    lastPoint = endPoint;
    QPainter painter(this);
    painter.scale(scale,scale); //进行放大操作
}

```

```
    painter.drawPixmap(0,0,pix);
}
```

运行程序，效果如下：



此时已经能进行正常绘图了。

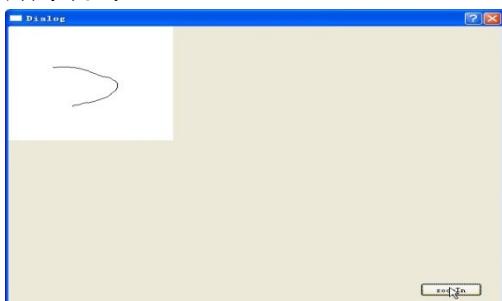
这种用改变窗口坐标大小来改变画布面积的方法，实际上是有损图片质量的。就像将一张位图放大一样，越放大越不清晰。原因就是，它的像素的个数没有变，如果将可视面积放大，那么单位面积里的像素个数就变少了，所以画质就差了。

下面我们简单说说另一种方法。

放大画布坐标。

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter pp(&pix);
    pp.scale(scale,scale);
    pp.drawLine(lastPoint/scale,endPoint/scale);
    lastPoint = endPoint;
    QPainter painter(this);
    painter.drawPixmap(0,0,pix);
}
```

效果如下：

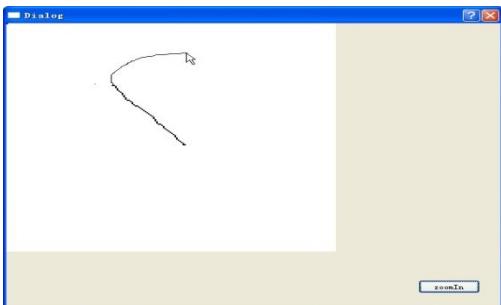


此时，画布中的内容并没有放大，而且画布也没有变大，不是我们想要的，所以我们再更改一下函数。

```
void Dialog::paintEvent(QPaintEvent *)
{
    if(scale!=1) //如果进行放大操作
    {
        QPixmap copyPix(pix.size()*scale); //临时画布，大小变化了 scale 倍
        QPainter pter(&copyPix);
        pter.scale(scale,scale);
        pter.drawPixmap(0,0,pix); //将以前画布上的内容复制到现在的画布上
        pix = copyPix; //将放大后的内容再复制回原来的画布上，这样只传递内容，不传递坐标系
    }
}
```

```
    scale =1; //让 scale 重新置 1
}
QPainter pp(&pix);
pp.scale(scale,scale);
pp.drawLine(lastPoint/scale,endPoint/scale);
lastPoint = endPoint;
QPainter painter(this);
painter.drawPixmap(0,0,pix);
}
```

此时运行效果如下：



这样就好了。可以看到，这样放大后再进行绘制，出来的效果是不同的。

我们就讲到这里，如果你有兴趣，可以接着研究！

怎么应用上面讲到的内容，你可以查看 [Qt 涂鸦板程序图文详细教程](#)。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 2,987 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 十九、Qt 2D 绘图（九）双缓冲绘图简介

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

上面一节我们实现了涂鸦板的功能，但是如果我们想在涂鸦板上绘制矩形，并且可以动态地绘制这个矩形，也就是说我们可以用鼠标画出随意大小的矩形，那该怎么办呢？

我们先进行下面的三步，最后引出所谓的双缓冲绘图的概念。

第一步：

我们更改上一节的那个程序的重绘函数。

```
void Dialog::paintEvent(QPaintEvent) {*}
{
    QPainter painter(this);
    int x = lastPoint.x();
    int y = lastPoint.y();
    int w = endPoint.x() - x;
    int h = endPoint.y() - y;
    painter.drawRect(x,y,w,h);
}
```

然后运行，效果如下。



这时我们已经可以拖出一个矩形了，但是这样直接在窗口上绘图，以前画的矩形是不能保存住的。所以我们下面加入画布，在画布上进行绘图。

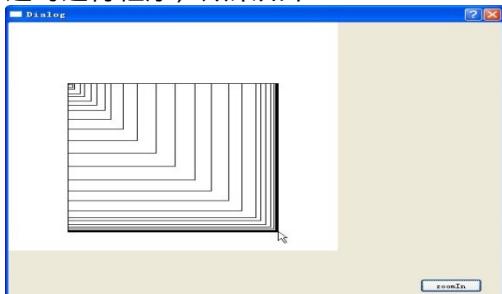
第二步：

我们先在构造函数里将画布设置大点：pix = QPixmap(400,400);

然后更改函数，如下：

```
void Dialog::paintEvent(QPaintEvent) {*}
{
    QPainter painter(this);
    int x = lastPoint.x();
    int y = lastPoint.y();
    int w = endPoint.x() - x;
    int h = endPoint.y() - y;
    QPainter pp(&pix);
    pp.drawRect(x,y,w,h);
    painter.drawPixmap(0,0,pix);
}
```

这时运行程序，效果如下：



现在虽然能画出矩形，但是却出现了无数个矩形，这不是我们想要的结果，我们希望能像第一步那样绘制矩形，所以我们再加入一个临时画布。

第三步：

首先，我们在 dialog.h 中的 private 里添加变量声明：

```
QPixmap tempPix; // 临时画布  
bool isDrawing; // 标志是否正在绘图
```

然后在 dialog.cpp 中的构造函数里进行变量初始化：

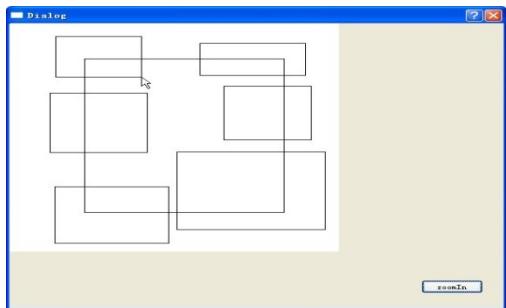
```
isDrawing = false;
```

最后更改函数如下：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    int x = lastPoint.x();  
    int y = lastPoint.y();  
    int w = endPoint.x() - x;  
    int h = endPoint.y() - y;  
    QPainter painter(this);  
    if(isDrawing) // 如 果 正 在 绘 图  
    {  
        tempPix = pix; // 将以前 pix 中的内容复制到 tempPix 中，这样实现了交互绘图  
        QPainter pp(&tempPix);  
        pp.drawRect(x,y,w,h);  
        painter.drawPixmap(0,0,tempPix);  
    }  
    else  
    {  
        QPainter pp(&pix);  
        pp.drawRect(x,y,w,h);  
        painter.drawPixmap(0,0,pix);  
    }  
}  
void Dialog::mousePressEvent(QMouseEvent *event)  
{  
    if(event->button()==Qt::LeftButton) // 鼠标左键按下  
    {  
        lastPoint = event->pos();  
        isDrawing = true; // 正在绘图  
    }  
}  
void Dialog::mouseMoveEvent(QMouseEvent *event)  
{  
    if(event->buttons()&Qt::LeftButton) // 鼠标左键按下的同时移动鼠标  
    {  
        endPoint = event->pos();  
        update();  
    }  
}  
void Dialog::mouseReleaseEvent(QMouseEvent *event)  
{  
    if(event->button() == Qt::LeftButton) // 鼠标左键释放  
    {  
        endPoint = event->pos();  
        isDrawing = false; // 结束绘图  
        update();  
    }  
}
```

我们使用两个画布，就解决了绘制矩形等图形的问题。

其中 `tempPix = pix;`一句代码很重要，就是它，才实现了消除那些多余的矩形。



### 双缓冲绘图简介：

根据我的理解，如果将第一步中不用画布，直接在窗口上进行绘图叫做无缓冲绘图，那么第二步中用了一个画布，将所有内容都先画到画布上，在整体绘制到窗口上，就该叫做单缓冲绘图，那个画布就是一个缓冲区。这样，第三步，用了两个画布，一个进行临时的绘图，一个进行最终的绘图，这样就叫做双缓冲绘图。

我们已经看到，利用双缓冲绘图可以实现动态交互绘制。其实，Qt 中所有部件进行绘制时，都是使用的双缓冲绘图。就算是第一步中我们没有用画布，Qt 在进行自身绘制时也是使用的双缓冲绘图，所以我们刚才那么说，只是为了更好地理解双缓冲的概念。

---

到这里，我们已经可以进行一些 Qt 2D 绘图方面的设计了。我这里有两个例子，一个是那个 [Qt 涂鸦板程序](#)，它是实践了我所讲的这几节的内容，可以说是对这几节内容的一个综合。还有一个例子，就是 [方块游戏系列](#) 那个是对这些知识的应用。如果你有兴趣，可以看一下。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,098 views  
Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

## 二十、Qt 2D 绘图（十）图形视图框架简介

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

我们前面用基本的绘图类实现了一个绘图软件，但是，我们无法做出像 Word 或者 Flash 中那样，绘制出来的图形可以作为一个元件进行任意变形。我们要想很容易地做出那样的效果，就要使用 Qt 中的图形视图框架。

The QGraphics View Framework (图形视图框架)，在 Qt Creator 中的帮助里可以查看它的介绍，当然那是英文的，这里有一篇中文的翻译，大家可以看一下：[Qt 的 graphics View 框架](#)

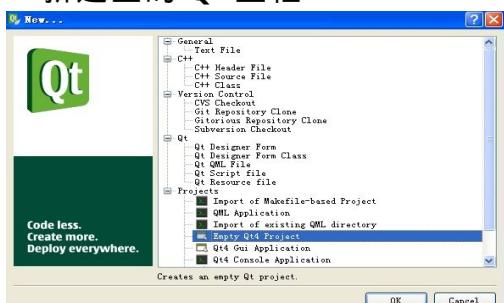
如果你的程序中要使用大量的 2D 图元，并且想要这些图元都能进行单独或群组的控制，你就要使用这个框架了。比方说像 Flash 一样的矢量绘图软件，各种游戏软件。但是因为这里涉及的东西太多了，不可能用一两篇文章就介绍清楚，所以这里我们只是提及一下，让一些刚入门的朋友知道有这样一个可用的框架。

最简单的使用：

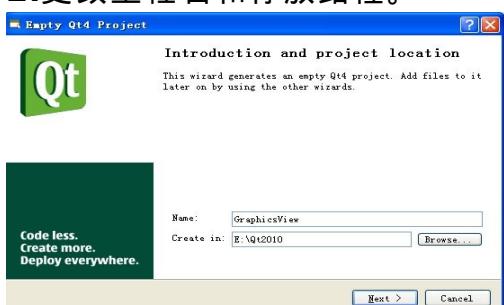
The QGraphics View Framework 包含三个大类：QGraphicsItem 项类（或者叫做图元类），QGraphicsScene 场景类，和 QGraphicsView 视图类。

QGraphicsItem 用来绘制你所要用到的图形，QGraphicsScene 用来包含并管理所有的图元，QGraphicsView 用来显示所有场景。而他们三个都拥有自己各自的坐标系统。我们下面就要来建立一个工程，完成一个最简单的例子。

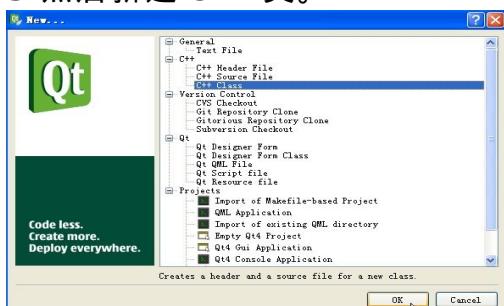
### 1. 新建空的 Qt 工程：



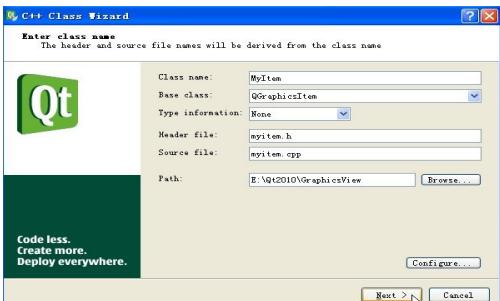
### 2. 更改工程名和存放路径。



### 3. 然后新建 C++ 类。



### 4. 更改类名为 MyItem，基类填写为 QGraphicsItem，如下图：



5.可以看到新建的类默认已经添加到了工程里。



6.新建 C++ Source File, 更改名字为 main.cpp, 如下图:



7.然后更改各文件的内容。

更改完成后, myitem.h 文件内容如下:

```
myitem.h
#ifndef MYITEM_H
#define MYITEM_H

#include <QGraphicsItem>

class MyItem : public QGraphicsItem
{
public:
    MyItem();
    //因为QGraphicsItem是抽象基类, 所以必须最少实现下面两个纯虚函数
    QRectF boundingRect() const;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
               QWidget *widget);
};

#endif // MYITEM_H
```

myitem.cpp 文件的内容如下:

```
myitem.cpp
#include "myitem.h"
#include <QPainter>
MyItem::MyItem()
{
}

QRectF MyItem::boundingRect() const
{
    //返回项的外框, 场景类利用这个外框来确定项的位置
    qreal adjust = 0.5;
    return QRectF(-18 - adjust, -22 - adjust,
                 36 + adjust, 60 + adjust);
}

void MyItem::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                   QWidget *widget)
{
    //绘制项
    painter->drawRect(0,0,20,20);
}
```

main.cpp 的内容如下:

```
main.cpp // main.cpp (Qt 4.7.1)
#include <QtGui.h>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QGraphicsScene scene;
    //新建场景
    scene.setSceneRect(-300, -300, 600, 600);
    scene.setItemIndexMethod(QGraphicsScene::NoIndex);
    //设置项索引方式
    MyItem *item = new MyItem;
    scene.addItem(item);
    //新项并添加到场景中
    QGraphicsView view(&scene);
    //为场景新建视图
    view.setRenderHint(QPainter::Antialiasing);
    view.setCacheMode(QGraphicsView::CacheBackground);
    view.setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate);
    view.setDragMode(QGraphicsView::ScrollHandDrag);
    view.resize(400, 300);
    view.show();
    //设置视图的几个属性，并显示出来
    return app.exec();
}
```

运行程序，最终效果如下：



这里我们只是演示了一下使用这个框架完成最简单的程序的过程，只起到抛砖引玉的作用。这个框架很复杂，但是功能也很强大，Qt Creator 中自带了几个相关的例子（在帮助中查找 Graphics View Examples 即可），你可以参考一下。因为篇幅问题，我们就只讲这么多，如果以后有机会，我会推出一个相关的专题来讲述这个框架。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,006 views

Tags: [2D 绘图](#), [creator](#), [qt](#), [yafeilinux](#), [教程](#)

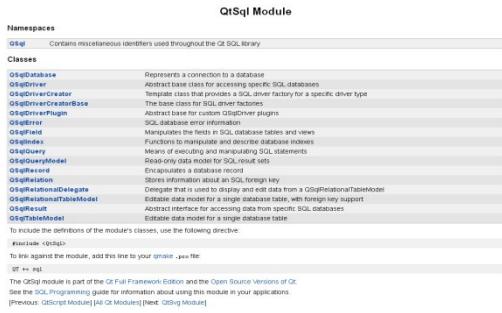
## 二十一、Qt 数据库（一）简介

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

从今天开始我们学习 Qt 数据库编程的内容。

先说明：我们以后使用现在最新的基于 Qt 4.6.2 的 Qt Creator 1.3.1 Windows 版本，该版本是 2010 年 2 月 17 日发布的。

数据库几乎是每个较大的软件所必须应用的，而在 Qt 中也使用 **QtSql** 模块实现了对数据库的完美支持。我们在 Qt Creator 的帮助中查找 **QtSql Module**，其内容如下图：



可以看到这个模块是一组类的集合，使用这个模块我们需要加入头文件 **#include <QtSql>**，而在工程文件中需要加入一行代码：**QT += sql**

这里每个类的作用在后面都有简单的介绍，你也可以进入其中查看其详细内容。下面我们先简单的说一下 **QSqlDatabase** 类和 **QSqlQuery** 类。

**QSqlDatabase** 类实现了数据库连接的操作，现在 Qt 支持的数据库类型有如下几种：

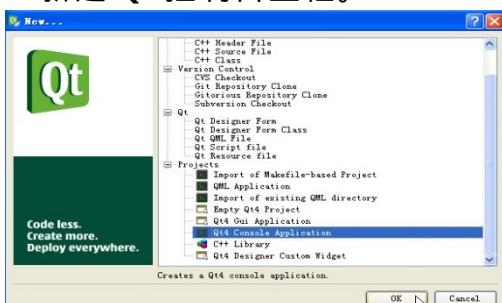
| Driver Type | Description                                 |
|-------------|---|
| QDB2        | IBM DB2                                     |
| QIBASE      | Borland InterBase Driver                    |
| QMYSQL      | MySQL Driver                                |
| QOCI        | Oracle Call Interface Driver                |
| QODBC       | ODBC Driver (includes Microsoft SQL Server) |
| QPSQL       | PostgreSQL Driver                           |
| QSQLITE     | SQLite version 3 or above                   |
| QSQLITE2    | SQLite version 2                            |
| QTDS        | Sybase Adaptive Server                      |

而现在我们使用的免费的 Qt 只提供了 SQLite 和 ODBC 数据库的驱动(我们可以在 Qt Creator 安装目录下的 qt\plugins\sqldrivers 文件夹下查看)，而其他数据库的驱动需要我们自己添加。SQLite 是一个小巧的嵌入式数据库，关于它的介绍你可以自己在网上查找。

**QSqlQuery** 类用来执行 SQL 语句。（关于 SQL 语句：在我的教程中只会出现很简单的 SQL 语句，你没有相关知识也可以看懂，但是如果想进行深入学习，就需要自己学习相关知识了。）

下面我们就先利用这两个类来实现最简单的数据库程序，其他的类我们会在以后的教程中逐个学习到。

### 1. 新建 Qt 控制台工程。



2. 选择上 **QtSql** 模块，这样就会自动往工程文件中添加 **QT += sql** 这行代码了。



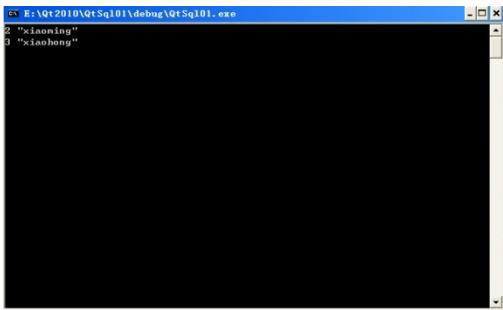
### 3.修改 main.cpp 中的内容如下。

```
#include <QtCore/QCoreApplication>
#include <QtSql>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE"); //添加数据库驱动
    db.setDatabaseName(":memory:"); //数据库连接命名
    if(!db.open()) //打开数据库
    {
        return false;
    }
    QSqlQuery query; //以下执行相关 QSL 语句
    query.exec("create table student(id int primary key,name varchar)");
    //新建 student 表, id 设置为主键, 还有一个 name 项
    query.exec("insert into student values(1,'xiaogang')");
    query.exec("insert into student values(2,'xiaoming')");
    query.exec("insert into student values(3,'xiaohong')");
    //向表中插入 3 条记录
    query.exec("select id,name from student where id >= 2");
    //查找表中 id >=2 的记录的 id 项和 name 项的值
    while(query.next()) //query.next()指向查找到的第一条记录, 然后每次后移一条
记录
    {
        int ele0 = query.value(0).toInt(); //query.value(0)是 id 的值, 将其转换为
int 型
        QString ele1 =query.value(1).toString();
        qDebug() << ele0 << ele1 ; //输出两个值
    }

    return a.exec();
}
```

我们使用了 SQLite 数据库, 连接名为“:memory:”表示这是建立在内存中的数据库, 也就是说该数据库只在程序运行期间有效。如果需要保存该数据库文件, 我们可以将它更改为实际的文件路径。

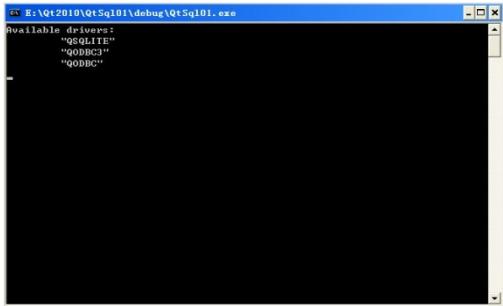
### 4.最终效果如下。



5.我们可以将主函数更改如下。

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    qDebug() << "Available drivers:";
    QStringList drivers = QSqlDatabase::drivers();
    foreach(QString driver, drivers)
        qDebug() << "\t" << driver;
    return a.exec();
}
```

这样运行程序就可以显示现在所有能用的数据库驱动了。



可以看到现在可用的数据库驱动只有三个。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 4,062 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十二、Qt 数据库（二）添加 MySQL 数据库驱动插件

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一节的末尾我们已经看到，现在可用的数据库驱动只有**3**种，那么怎样使用其他的数据库呢？在**Qt**中，我们需要自己编译其他数据库驱动的代码，让它们以插件的形式来使用。下面我们就以现在比较流行的**MySQL**数据库为例，说明一下怎样在**Qt Creator**中添加数据库驱动插件。

在讲述之前，我们先看一下**Qt Creator**中数据库的插件到底放在哪里。

我们进入**Qt Creator**的安装目录，然后进入相对应的文件夹下，比方我这里是 D:\Qt\2010.02.1\qt\plugins\sqldrivers

在这里我们可以看见几个文件，如下图：



根据名字中的关键字，我们可以判断出这就是ODBC数据库和SQLite数据库的驱动插件。下面我们编译好MySQL数据库驱动后，也会在这里出现相对应的文件。

首先：我们查看怎样安装数据库插件。

我们打开**Qt Creator**，在帮助中搜索**SQL Database Drivers**关键字。这里列出了编译**Qt**支持的所有数据库的驱动的方法。

我们下拉到在**windows**上编译**QMYSQL**数据库插件的部分，其内容如下：

### How to Build the QMYSQL Plugin on Windows

You need to get the MySQL installation files. Run `setup.exe` and choose "Custom Install". Install the "Libs & Include Files" Module.  
Build the plugin as follows (here it is assumed that MySQL is installed in `c:\mysql`):

```
cd %QTDIR%\src\plugins\sqldrivers\mysql  
qmake "INCLUDEPATH+=c:\MySQL\include" "LIBS+=c:\MySQL\MySQL Server (version)\lib\opt\libmysql.lib" mysql.pro  
make
```

If you are not using a Microsoft compiler, replace `make` with `mingw32-make` in the line above.

Note: This database plugin is not supported for Windows CE.

Note: Including `“-o Makefile”` as an argument to `qmake` to tell it where to build the makefile can cause the plugin to be built in release mode only. If you are expecting a debug version to be built as well, don't use the `“-o Makefile”` option.

这里详细介绍了整个编译的过程，其可以分为以下几步：

第一，下载MySQL的安装程序，在安装时选择定制安装，这时选中安装**Libs**和**Include**文件。安装位置可以是C:\MySQL。

**注意：安装位置不建议改动，因为下面进行编译的命令中使用了安装路径，如果改动，那么下面也要进行相应改动。**

第二，进行编译。我们按照实际情况输入的命令如下。

```
cd %QTDIR%\src\plugins\sqldrivers\mysql  
qmake "INCLUDEPATH+=C:\MySQL\include"  
"LIBS+=C:\MySQL\lib\opt\libmysql.lib" mysql.pro  
mingw32-make
```

**注意：在上面的命令中 `qmake` 之后如果加上“`-o Makefile`”选项，那么这个插件只能在以 `release` 模式编译程序时才能使用，所以我们上面没有加这个选项。**

然后：我们按照上面的过程进行相应操作。

**1. 我们先下载 MySQL 的安装文件。**

我们可以到 MySQL 的官方主页 <http://www.mysql.com> 进行下载最新的 MySQL 的 windows 版本，现在具体的下载页面地址为：

<http://www.mysql.com/downloads/mirror.php?id=383405#mirrors>

我们不进行注册，直接点击其下面的

No thanks, just take me to the downloads!

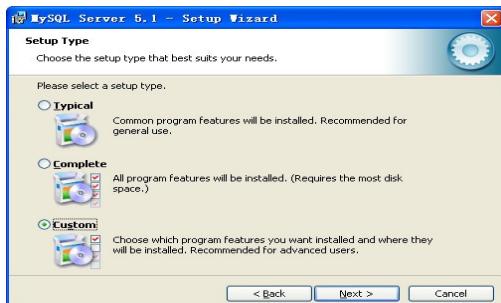
可以在其中选择一个镜像网点进行下载，我使用的是 Asia 下的最后一个，就是台湾的镜像网点下载的。

下载到的文件名为：mysql-essential-5.1.44-win32，其中的 win32 表明是 32 位的 windows 系统，这一点一定要注意。文件大小为 40M 左右。

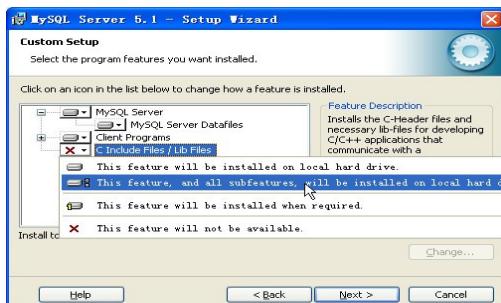
当然你也可以到中文网站上进行下载：<http://www.mysql.cn/>，随便下一个 windows 的版本就行。

## 2. 安装软件。

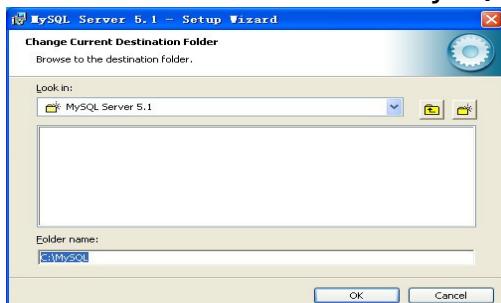
我们选择定制安装 Custom。



然后选中安装 Include 文件和 Lib 文件。



我们将安装路径更改为：C:\MySQL。



最终的界面如下。



安装完成后，我们不进行任何操作，所以将两个选项都取消。



### 3. 进行编译。

我们在桌面上开始菜单中找到 Qt Creator 的菜单，然后打开 Qt Command Prompt。



然后输入第一条命令 `cd %QTDIR%\src\plugins\sqldrivers\mysql` 后按回车，运行效果如下。

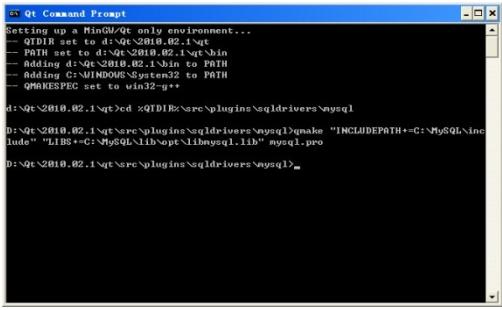
```
Setting up a MinGW/Qt only environment...
-- QTDIR set to d:\Qt\2010.02.1\Qt
-- PATH set to d:\Qt\2010.02.1\Qt\bin
-- Adding d:\Qt\2010.02.1\bin to PATH
-- Adding c:\mingw\sysw32 to PATH
-- QMKMSPKG set to win32-qwt

d:\Qt\2010.02.1\Qt>cd %QTDIR%\src\plugins\sqldrivers\mysql
D:\Qt\2010.02.1\Qt\src\plugins\sqldrivers\mysql>
```

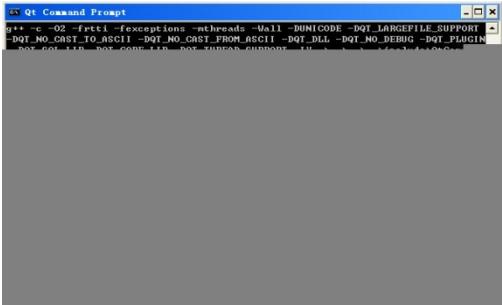
然后输入第二条命令：

`qmake "INCLUDEPATH+=C:\MySQL\include"`  
`"LIBS+=C:\MySQL\lib\opt\libmysql.lib"` mysql.pro

按回车后运行效果如下：



最后输入：mingw32-make，按下回车后经过几秒的编译，最终效果如下：



整个编译过程中都没有出现错误提示，可以肯定插件已经编译完成了。

4. 我们再次进入 **Qt Creator** 安装目录下存放数据库驱动插件的文件夹。

我这里是 D:\Qt\2010.02.1\qt\plugins\sqldrivers

其内容如下：

可以看到已经有了和 MySQL 相关的文件了。

最后：我们编写程序测试插件。

**1. 我们将上一次的主函数更改如下。**

```
int main(int argc, char *argv[1])
```

```
int main(int argc, char *argv[])
{
QCoreApplication a(argc, argv);
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL"); //添加数据库驱动
return a.exec();
}
```

运行程序，效果如下。

这里提示：OSqlDatabase: OMySQL driver not loaded。

2.这时我们需要将 C:\MySQL\bin 目录下的 libmySQL.dll 文件复制到我们 Qt Creator 安装目录下的 qt\bin 目录中。

如下圖：

3.这时再运行程序，就没有提示了。

4. 我们再将主函数更改一下，测试这时可用的数据库驱动。

int main(int argc, char \*argv[1])

11

```
QCoreApplication a(argc, argv);
```

```
qDebug() << "Available drivers:";
```

```
QStringList drivers = QSqlDatabase::drivers();
```

```
foreach(QString driver, drivers)
qDebug() << "\t" << driver;
return a.exec();
}
```

运行效果如下：

可以看到，现在已经有了 MySQL 的数据库驱动了。

我们这里只介绍了 MySQL 驱动插件在 windows 下的编译方法，其他数据库和其他平台的编译方法可以按照帮助中的说明进行，我们不再介绍。其实 Qt 不仅可以编译现成的数据库驱动插件，我们也可以编写自己的数据库驱动插件，当然这是一件相当复杂的事情，我们这里也就不再进行介绍。

关于 MySQL 的使用，我们的教程里现在不再涉及，在

<http://dev.mysql.com/doc/refman/5.1/zh/index.html> 这里有 MySQL 的中文参考手册，你可以进行参考。

可以到[资源下载](#)页面下载生成的文件。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 4,543 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十三、Qt 数据库（三）利用 QSqlQuery 类执行 SQL 语句（一）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

**SQL** 即结构化查询语言，是关系数据库的标准语言。前面已经提到，在 **Qt** 中利用 **QSqlQuery** 类实现了执行 **SQL** 语句。需要说明，我们这里只是 **Qt** 教程，而非专业的数据库教程，所以我们不会对数据库中的一些知识进行深入讲解，下面只是对最常用的几个知识点进行讲解。

我们下面先建立一个工程，然后讲解四个知识点，分别是：

一，操作 **SQL** 语句返回的结果集。

二，在 **SQL** 语句中使用变量。

三，批处理操作。

四，事务操作。

我们新建 Qt4 Gui Application 工程，我这里工程名为 query，然后选中 QSql 模块，Base class 选 QWidget。工程建好后，添加 C++ Header File，命名为 connection.h，更改其内容如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(":memory:");
    if (!db.open()) {
        QMessageBox::critical(0, qApp->tr("Cannot open database"),
                             qApp->tr("Unable to establish a database connection."
                                      ), QMessageBox::Cancel);
        return false;
    }
    QSqlQuery query;
    query.exec("create table student (id int primary key, "
              "name varchar(20));");
    query.exec("insert into student values(0, 'first')");
    query.exec("insert into student values(1, 'second')");
    query.exec("insert into student values(2, 'third')");
    query.exec("insert into student values(3, 'fourth')");
    query.exec("insert into student values(4, 'fifth')");
    return true;
}
```

#endif // CONNECTION\_H

然后更改 main.cpp 的内容如下：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "connection.h"
int main(int argc, char *argv[])
{
```

```
QApplication a(argc, argv);

if (!createConnection())
    return 1;

Widget w;
w.show();
return a.exec();
}
```

可以看到，我们是在主函数中打开数据库的，而数据库连接用一个函数完成，并单独放在一个文件中，这样的做法使得主函数很简洁。我们今后使用数据库时均使用这种方法。我们打开数据库连接后，新建了一个学生表，并在其中插入了几条记录。

表中的一行就叫做一条记录，一列是一个属性。这个表共有 5 条记录，id 和 name 两个属性。程序中的“`id int primary key`”表明 id 属性是主键，也就是说以后添加记录时，必须有 id 项。下面我们打开 `widget.ui` 文件，在设计器中向界面上添加一个 Push Button，和一个 Spin Box。将按钮的文本改为“查询”，然后进入其单击事件槽函数，更改如下。

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
        qDebug() << query.value(0).toInt() << query.value(1).toString();
    }
}
```

我们在 `widget.cpp` 中添加头文件：

```
#include <QSqlQuery>
#include <QtDebug>
```

然后运行程序，单击“查询”按钮，效果如下：

可以看到在输出窗口，表中的所有内容都输出出来了。这表明我们的数据库连接已经成功建立了。

一，操作 **SQL** 语句返回的结果集。

在上面的程序中，我们使用 `query.exec("select * from student");` 来查询出表中所有的内容。其中的 SQL 语句“`select * from student`”中“\*”号表明查询表中记录的所有属性。而当 `query.exec("select * from student");` 这条语句执行完后，我们便获得了相应的执行结果，因为获得的结果可能不止一条记录，所以我们称之为结果集。

结果集其实就是查询到的所有记录的集合，而在 `QSqlQuery` 类中提供了多个函数来操作这个集合，需要注意这个集合中的记录是从 0 开始编号的。最常用的有：

**seek(int n)** : `query` 指向结果集的第 `n` 条记录。

**first()** : `query` 指向结果集的第一条记录。

**last()** : `query` 指向结果集的最后一条记录。

**next()** : `query` 指向下一条记录，每执行一次该函数，便指向相邻的下一条记录。

**previous()** : `query` 指向上一条记录，每执行一次该函数，便指向相邻的上一条记录。

**record()** : 获得现在指向的记录。

**value(int n)**：获得属性的值。其中 n 表示你查询的第 n 个属性，比方上面我们使用“select \* from student”就相当于“select id, name from student”，那么 value(0) 返回 id 属性的值，value(1) 返回 name 属性的值。该函数返回 QVariant 类型的数据，关于该类型与其他类型的对应关系，可以在帮助中查看 QVariant。

**at()**：获得现在 query 指向的记录在结果集中的编号。

需要说明，当刚执行完 query.exec("select \* from student"); 这行代码时，query 是指向结果集以外的，我们可以利用 query.next()，当第一次执行这句代码时，query 便指向了结果集的第一条记录。当然我们也可以利用 seek(0) 函数或者 first() 函数使 query 指向结果集的第一条记录。但是为了节省内存开销，推荐的方法是，在 query.exec("select \* from student"); 这行代码前加上 query.setForwardOnly(true); 这条代码，此后只能使用 next() 和 seek() 函数。

下面将“查询”按钮的槽函数更改如下：

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    query.exec("select * from student");
    qDebug() << "exec next() :";
    if(query.next())
        //开始就先执行一次 next()函数，那么 query 指向结果集的第一条记录
    {
        int rowNum = query.at();
        //获取 query 所指向的记录在结果集中的编号
        int columnNum = query.record().count();
        //获取每条记录中属性（即列）的个数
        int fieldNo = query.record().indexOf("name");
        //获取“name”属性所在列的编号，列从左向右编号，最左边的编号为 0
        int id = query.value(0).toInt();
        //获取 id 属性的值，并转换为 int 型
        QString name = query.value(fieldNo).toString();
        //获取 name 属性的值
        qDebug() << "rowNum is :" << rowNum //将结果输出
            << " id is :" << id
            << " name is :" << name
            << " columnNum is :" << columnNum;
    }
    qDebug() << "exec seek(2) :";
    if(query.seek(2))
        //定位到结果集中编号为 2 的记录，即第三条记录，因为第一条记录的编号为 0
    {
        qDebug() << "rowNum is :" << query.at()
            << " id is :" << query.value(0).toInt()
            << " name is :" << query.value(1).toString();
    }
    qDebug() << "exec last() :";
    if(query.last())
        //定位到结果集中最后一条记录
    {
        qDebug() << "rowNum is :" << query.at()
            << " id is :" << query.value(0).toInt()
```

```
        << " name is :" << query.value(1).toString();
    }
}
```

然后在 widget.cpp 文件中添加头文件。

```
#include <QSqlRecord>
```

运行程序，结果如下：

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 四月 30th, 2010. 3,845 views  
Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十四、Qt 数据库（四）利用 QSqlQuery 类执行 SQL 语句（二）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

接着上一篇教程。

二，在 **SQL** 语句中使用变量。

我们先看下面的一个例子，将“查询”按钮的槽函数更改如下：

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    query.prepare("insert into student (id, name) "
                 "values (:id, :name)");
    query.bindValue(0, 5);
    query.bindValue(1, "sixth");
    query.exec();
    //下面输出最后一条记录
    query.exec("select * from student");
    query.last();
    int id = query.value(0).toInt();
    QString name = query.value(1).toString();
    qDebug() << id << name;
}
```

运行效果如下：

可以看到，在 student 表的最后又添加了一条记录。在上面的程序中，我们先使用了 `prepare()` 函数，在其中利用了“`:id`”和“`:name`”来代替具体的数据，而后又利用 `bindValue()` 函数给 `id` 和 `name` 两个属性赋值，这称为绑定操作。其中编号 0 和 1 分别代表“`:id`”和“`:name`”，就是说按照 `prepare()` 函数中出现的属性从左到右编号，最左边是 0。这里的“`:id`”和“`:name`”，叫做占位符，这是 ODBC 数据库的表示方法，还有一种 Oracle 的表示方法就是全部用“`?`”号。如下：

```
query.prepare("insert into student (id, name) "
             "values (?, ?)");
query.bindValue(0, 5);
query.bindValue(1, "sixth");
query.exec();
```

我们也可以利用 `addBindValue()` 函数，这样就可以省去编号，它是按顺序给属性赋值的，如下：

```
query.prepare("insert into student (id, name) "
             "values (?, ?)");
query.addBindValue(5);
query.addBindValue("sixth");
query.exec();
```

当用 ODBC 的表示方法时，我们也可以将编号用实际的占位符代替，如下：

```
query.prepare("insert into student (id, name) "
             "values (:id, :name)");
query.bindValue(":id", 5);
query.bindValue(":name", "sixth");
query.exec();
```

以上各种形式的表示方式效果是一样的。特别注意，在最后一定要执行 exec() 函数，所做的操作才能被真正执行。

下面我们就利用绑定操作在 SQL 语句中使用变量了。

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    query.prepare("select name from student where id = ?");
    int id = ui->spinBox->value(); //从界面获取 id 的值
    query.addBindValue(id); //将 id 值进行绑定
    query.exec();
    query.next(); //指向第一条记录
    qDebug() << query.value(0).toString();
}
```

运行程序，效果如下：

我们改变 spinBox 的数值大小，然后按下“查询”按钮，可以看到对应的结果就出来了。

三，批处理操作。

当要进行多条记录的操作时，我们就可以利用绑定进行批处理。看下面的例子。

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery q;
    q.prepare("insert into student values (?, ?)");
    QVariantList ints;
    ints << 10 << 11 << 12 << 13;
    q.addBindValue(ints);
    QVariantList names;
    names << "xiaoming" << "xiaoliang" << "xiaogang" <<
    QVariant(QVariant::String);
    //最后一个为空字符串，应与前面的格式相同
    q.addBindValue(names);
    if (!q.execBatch()) //进行批处理，如果出错就输出错误
        qDebug() << q.lastError();
    //下面输出整张表
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
        int id = query.value(0).toInt();
        QString name = query.value(1).toString();
        qDebug() << id << name;
    }
}
```

然后在 widget.cpp 文件中添加头文件 `#include <QSqlError>`。

我们在程序中利用列表存储了同一属性的多个值，然后进行了值绑定。最后执行 `execBatch()` 函数进行批处理。注意程序中利用 `QVariant(QVariant::String)` 来输入空值 `NULL`，因为前面都是 `QString` 类型的，所以这里要使用 `QVariant::String` 使格式一致化。

运行效果如下：

#### 四， 事务操作。

事务是数据库的一个重要功能，所谓事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位。在 Qt 中用 transaction() 开始一个事务操作，用 commit() 函数或 rollback() 函数进行结束。commit() 表示提交，即提交事务的所有操作。具体地说就是将事务中所有对数据库的更新写回到数据库，事务正常结束。rollback() 表示回滚，即在事务运行的过程中发生了某种故障，事务不能继续进行，系统将事务中对数据库的所有已完成的操作全部撤销，回滚到事务开始时的状态。

如下面的例子：

```
void Widget::on_pushButton_clicked()
{
    if(QSqlDatabase::database().driver()->hasFeature(QSqlDriver::Transactions))
    {
        //先判断该数据库是否支持事务操作
        QSqlQuery query;
        if(QSqlDatabase::database().transaction()) //启动事务操作
        {
            //
            //下面执行各种数据库操作
            query.exec("insert into student values (14, 'hello')");
            query.exec("delete from student where id = 1");
            //
            if(!QSqlDatabase::database().commit())
            {
                qDebug() << QSqlDatabase::database().lastError(); //提交
                if(!QSqlDatabase::database().rollback())
                    qDebug() << QSqlDatabase::database().lastError(); //回滚
            }
        }
        //输出整张表
        query.exec("select * from student");
        while(query.next())
            qDebug() << query.value(0).toInt() << query.value(1).toString();
    }
}
```

然后在 widget.cpp 文件中添加头文件 `#include <QSqlDriver>`。  
QSqlDatabase::database() 返回程序前面所生成的连接的 QSqlDatabase 对象。  
hasFeature() 函数可以查看一个数据库是否支持事务。

运行结果如下：

可以看到结果是正确的。

对 SQL 语句我们就介绍这么多，其实 Qt 中提供了更为简单的不需要 SQL 语句就可以操作数据库的方法，我们在下一节讲述这些内容。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 2,783 views  
Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十五、Qt 数据库（五） QSqlQueryModel

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一篇的最后我们说到，Qt 中使用了自己的机制来避免使用 SQL 语句，它为我们提供了更简单的数据库操作和数据显示模型。它们分别是只读的 **QSqlQueryModel**，操作单表的 **QSqlTableModel** 和以及可以支持外键的 **QSqlRelationalTableModel**。这次我们先讲解 **QSqlQueryModel**。

**QSqlQueryModel** 类为 SQL 的结果集提供了一个只读的数据模型，下面我们先利用这个类进行一个最简单的操作。

我们新建 Qt4 Gui Application 工程，我这里工程名为 queryModel，然后选中 QSql 模块，Base class 选 QWidget。工程建好后，添加 C++ Header File，命名为 database.h，更改其内容如下：

```
#ifndef DATABASE_H
#define DATABASE_H
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open()) return false;
    QSqlQuery query;
    query.exec("create table student (id int primary key, name varchar)");
    query.exec("insert into student values (0,'yafei0')");
    query.exec("insert into student values (1,'yafei1')");
    query.exec("insert into student values (2,'yafei2')");
    return true;
}
#endif // DATABASE_H
```

这里我们使用了 `db.setDatabaseName("database.db");`，我们没有再使用以前的内存数据库，而是使用了真实的文件，这样后面对数据库进行的操作就能保存下来了。

然后进入 main.cpp，将其内容更改如下：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "database.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if(!createConnection())
        return 1;
    Widget w;
    w.show();
    return a.exec();
}
```

下面我们在 widget.ui 中添加一个显示为“查询”的 Push Button，并进入其单击事件槽函数，更改如下：

```
void Widget::on_pushButton_clicked()
{
```

```
QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("select * from student");
model->setHeaderData(0, Qt::Horizontal, tr("id"));
model->setHeaderData(1, Qt::Horizontal, tr("name"));
QTableView *view = new QTableView;
view->setModel(model);
view->show();
}
```

我们新建了 QSqlQueryModel 类对象 model，并用 setQuery() 函数执行了 SQL 语句 “(“select \* from student”);” 来查询整个 student 表的内容，可以看到，该类并没有完全避免 SQL 语句。然后我们设置了表中属性显示时的名字。最后我们建立了一个视图 view，并将这个 model 模型关联到视图中，这样数据库中的数据就能在窗口上的表中显示出来了。

我们在 widget.cpp 中添加头文件：

```
#include <QSqlQueryModel>
#include <QTableView>
```

我们运行程序，并按下“查询”按键，效果如下：

我们在工程文件夹下查看数据库文件：

下面我们利用这个模型来操作数据库。

**1.** 我们在 **void Widget::on\_pushButton\_clicked()** 函数中添加如下代码：

```
int column = model->columnCount(); //获得列数
int row = model->rowCount(); // 获得行数
QSqlRecord record = model->record(1); //获得一条记录
QModelIndex index = model->index(1,1); //获得一条记录的一个属性的值
qDebug() << "column num is:" << column << endl
    << "row num is:" << row << endl
    << "the second record is:" << record << endl
    << "the data of index(1,1) is:" << index.data();
```

我们在 widget.cpp 中添加头文件：

```
#include <QSqlRecord>
#include <QModelIndex>
#include <QDebug>
```

此时运行程序，效果如下：

**2.** 当然我们在这里也可以使用前面介绍过的 **query** 执行 **SQL** 语句。

例如我们在 void Widget::on\_pushButton\_clicked() 函数中添加如下代码：

```
QSqlQuery query = model->query();
query.exec("select name from student where id = 2 ");
query.next();
qDebug() << query.value(0).toString();
```

这样就可以输出表中的值了，你可以运行程序测试一下。

**3.** 当我们将函数改为如下。

```
void Widget::on_pushButton_clicked()
{
```

```

QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("select * from student");
model->setHeaderData(0, Qt::Horizontal, tr("id"));
model->setHeaderData(1, Qt::Horizontal, tr("name"));
QTableView *view = new QTableView;
view->setModel(model);
view->show();
QSqlQuery query = model->query();
query.exec("insert into student values (10,'yafei10')");
//插入一条记录
}

```

这时我们运行程序，效果如下：

我们发现表格中并没有增加记录，怎么回事呢？

我们关闭程序，再次运行，效果如下：

发现这次新的记录已经添加了。在上面我们执行了添加记录的 SQL 语句，但是在添加记录之前，就已经进行显示了，所以我们的更新没能动态的显示出来。为了能让其动态地显示我们的更新，我们可以将函数更改如下：

```

void Widget::on_pushButton_clicked()
{
    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("select * from student");
    model->setHeaderData(0, Qt::Horizontal, tr("id"));
    model->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view = new QTableView;
    view->setModel(model);
    view->show();
    QSqlQuery query = model->query();
    query.exec("insert into student values (20,'yafei20')");
    //插入一条记录
    model->setQuery("select * from student"); //再次查询整张表
    view->show(); //再次进行显示
}

```

这时运行程序，效果如下：

可以看到，这时已经将新添的记录显示出来了。

刚开始我们就讲到，这个模型默认是只读的，所以我们在窗口上并不能对表格中的内容进行修改。但是我们可以创建自己的模型，然后按照我们自己的意愿来显示数据和修改数据。要想使其可读写，需要自己的类继承自 QSqlQueryModel，并且重写 setData() 和 flags() 两个函数。如果我们要改变数据的显示，就要重写 data() 函数。

下面的例子中我们让 student 表的 id 属性列显示红色，name 属性列可编辑。

**1.我们在工程中添加 C++ Class，然后 Class name 设为 MySqlQueryModel，Base Class 设为 QSqlQueryModel，如下：**

**2.我们将 mysqlquerymodel.h 中的内容更改如下：**

```

class MySqlQueryModel : public QSqlQueryModel
{

```

```

public:
    MySqlQueryModel();
    //下面三个函数都是虚函数,我们对其进行重载
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value, int role);
    QVariant data(const QModelIndex &item, int role=Qt::DisplayRole) const;
    //
private:
    bool setName(int studentId, const QString &name);
    void refresh();
};

然后将 mysqlquerymodel.cpp 文件更改如下：
#include "mysqlquerymodel.h"
#include <QSqlQuery>
#include <QColor>
MySqlQueryModel::MySqlQueryModel()
{
}
Qt::ItemFlags MySqlQueryModel::flags(
    const QModelIndex &index) const //返回表格是否可更改的标志
{
    Qt::ItemFlags flags = QSqlQueryModel::flags(index);
    if (index.column() == 1) //第二个属性可更改
        flags |= Qt::ItemIsEditable;
    return flags;
}
bool MySqlQueryModel::setData(const QModelIndex &index, const QVariant
&value, int /* role */)
    //添加数据
{
    if (index.column() < 1 || index.column() > 2)
        return false;
    QModelIndex primaryKeyIndex = QSqlQueryModel::index(index.row(), 0);
    int id = data(primaryKeyIndex).toInt(); //获取 id 号
    clear();
    bool ok;
    if (index.column() == 1) //第二个属性可更改
        ok = setName(id, value.toString());
    refresh();
    return ok;
}
void MySqlQueryModel::refresh() //更新显示
{
    setQuery("select * from student");
    setHeaderData(0, Qt::Horizontal, QObject::tr("id"));
    setHeaderData(1, Qt::Horizontal, QObject::tr("name"));
}

```

```
bool MySqlQueryModel::setName(int studentId, const QString &name) //添加
name 属性的值
{
    QSqlQuery query;
    query.prepare("update student set name = ? where id = ?");
    query.addBindValue(name);
    query.addBindValue(studentId);
    return query.exec();
}
QVariant MySqlQueryModel::data(const QModelIndex &index, int role) const
//更改数据显示样式
{
    QVariant value = QSqlQueryModel::data(index, role);
    if (role == Qt::TextColorRole && index.column() == 0)
        return QVariantFromValue(QColor(Qt::red)); //第一个属性的字体颜色为红色
    return value;
}
```

在 widget.cpp 文件中添加头文件： #include “mysqlquerymodel.h”

然后更改函数如下：

```
void Widget::on_pushButton_clicked()
{
    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("select * from student");
    model->setHeaderData(0, Qt::Horizontal, tr("id"));
    model->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view = new QTableView;
    view->setModel(model);
    view->show();
    MySqlQueryModel *myModel = new MySqlQueryModel; //创建自己模型的对象
    myModel->setQuery("select * from student");
    myModel->setHeaderData(0, Qt::Horizontal, tr("id"));
    myModel->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view1 = new QTableView;
    view1->setWindowTitle("mySqlQueryModel"); //修改窗口标题
    view1->setModel(myModel);
    view1->show();
}
```

运行效果如下：

可以看到我们要的效果已经出来了。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,085 views  
Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十六、Qt 数据库（六） QSqlTableModel

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一篇我们讲到只读的 **QSqlQueryModel** 也可以使其可编辑，但是很麻烦。Qt 提供了操作单表的 **QSqlTableModel**，如果我们需要对表的内容进行修改，那么我们就可以直接使用这个类。

**QSqlTableModel**，该类提供了一个可读写单张 SQL 表的可编辑数据模型。我们下面就对其的几个常用功能进行介绍，分别是修改，插入，删除，查询，和排序。

在开始讲之前，我们还是新建 Qt4 Gui Application 工程，我这里工程名为 `tableModel`，然后选中 `QtSql` 模块，`Base class` 选 `QWidget`。工程建好后，添加 C++ Header File，命名为 `database.h`，更改其内容如下：

```
#ifndef DATABASE_H
#define DATABASE_H
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QObject>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open()) return false;
    QSqlQuery query;
    query.exec(QObject::tr("create table student (id int primary key, name
vchar)"));
    query.exec(QObject::tr("insert into student values (0,'刘明')"));
    query.exec(QObject::tr("insert into student values (1,'陈刚')"));
    query.exec(QObject::tr("insert into student values (2,'王红')"));
    return true;
}
#endif // DATABASE_H
```

为了在数据库中能使用中文，我们使用了 **QObject** 类的 `tr()` 函数。而在下面的 `main()` 函数中我们也需要添加相应的代码来支持中文。

然后将 `main.cpp` 的文件更改如下：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "database.h"
#include <QTextCodec>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    if(!createConnection())
        return 1;
    Widget w;
    w.show();
    return a.exec();
}
```

下面我们打开 `widget.ui`，设计界面如下：

其中的部件有 Table View 和 Line Edit，其余是几个按钮部件。

然后在 widget.h 中加入头文件： #include <QSqlTableModel>

在 private 中声明对象： QSqlTableModel \*model;

因为我们要在不同的函数中使用 model 对象，所以我们在里声明它。

我们到 widget.cpp 文件中的构造函数里添加如下代码：

```
model = new QSqlTableModel(this);
model->setTable("student");
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
model->select(); //选取整个表的所有行
// model->removeColumn(1); //不显示 name 属性列,如果这时添加记录，则该属性的值
添加不上
ui->tableView->setModel(model);
// ui->tableView->setEditTriggers(QAbstractItemView::NoEditTriggers); //使其不可编辑
```

此时运行程序，效果如下：

可以看到，这个模型已经完全脱离了 SQL 语句，我们只需要执行 select() 函数就能查询整张表。上面有两行代码被注释掉了，你可以取消注释，测试一下它们的作用。

第一，修改操作。

1. 我们进入“提交修改”按钮的单击事件槽函数，修改如下：

```
void Widget::on_pushButton_clicked() //提交修改
{
    model->database().transaction(); //开始事务操作
    if (model->submitAll()) {
        model->database().commit(); //提交
    } else {
        model->database().rollback(); //回滚
        QMessageBox::warning(this, tr("tableModel"),
                             tr("数据库错误: %1")
                             .arg(model->lastError().text()));
    }
}
```

这里用到了事务操作，真正起提交操作的是 model->submitAll() 一句，它提交所有更改。

2. 进入“撤销修改”按钮单击事件槽函数，并更改如下：

```
void Widget::on_pushButton_2_clicked() //撤销修改
{
    model->revertAll();
}
```

它只有简单的一行代码。

我们需要在 widget.cpp 文件中添加头文件：

```
#include <QMessageBox>
#include <QSqlError>
```

此时运行程序，效果如下：

我们将“陈刚”改为“李强”，如果我们点击“撤销修改”，那么它就会重新改为“陈刚”，而当我们点击“提交修改”后它就会保存到数据库，此时再点击“撤销修改”就修改不回来了。

可以看到，这个模型可以将所有修改先保存到 model 中，只有当我们执行提交修改后，才会真正写入数据库。当然这也是因为在最开始设置了它的保存策略：

```
model->setEditStrategy(QSqlTableModel::OnManualSubmit);  
OnManualSubmit 表明我们要提交修改才能使其生效。
```

第二，查询操作。

1. 我们进入“查询”按钮的单击事件槽函数，更改如下：

```
void Widget::on_pushButton_7_clicked() //查询  
{  
    QString name = ui->lineEdit->text();  
    model->setFilter(QObject::tr("name = '%1'").arg(name)); //根据姓名进行筛选  
    model->select(); //显示结果  
}
```

我们使用 setFilter() 函数进行关键字筛选，这个函数是对整个结果集进行查询。为了使用变量，我们使用了 QObject 类的 tr() 函数。

2. 我们进入“返回全表”按钮的单击事件槽函数，更改如下：

```
void Widget::on_pushButton_8_clicked() //返回全表  
{  
    model->setTable("student"); //重新关联表  
    model->select(); //这样才能再次显示整个表的内容  
}
```

为了再次显示整个表的内容，我们需要再次关联这个表。

运行效果如下：

我们输入一个姓名，点击“查询”按钮后，就可以显示该记录了。再点击“返回全表”按钮则返回。

第三，排序操作。

我们分别进入“按 id 升序排列”和“按 id 降序排列”按钮的单击事件槽函数，更改如下：

```
void Widget::on_pushButton_5_clicked() //升序  
{  
    model->setSort(0,Qt::AscendingOrder); //id 属性，即第 0 列，升序排列  
    model->select();  
}  
void Widget::on_pushButton_6_clicked() //降序  
{  
    model->setSort(0,Qt::DescendingOrder);  
    model->select();  
}
```

我们这里使用了 setSort() 函数进行排序，它有两个参数，第一个参数表示按第几个属性排序，表头从左向右，最左边是第 0 个属性，这里就是 id 属性。第二个参数是排序方法，有升序和降序两种。

运行程序，效果如下：

这是按下“按 id 降序排列”按钮后的效果。

第四，删除操作。

我们进入“删除选中行”按钮单击事件槽函数，进行如下更改：

```
void Widget::on_pushButton_4_clicked() //删除当前行  
{
```

```

int curRow = ui->tableView->currentIndex().row();
//获取选中的行
model->removeRow(curRow);
//删除该行
int ok = QMessageBox::warning(this,tr("删除当前行!"),tr("你确定"
                                              "删除当前行吗?"),
                               QMessageBox::Yes,QMessageBox::No);
if(ok == QMessageBox::No)
{
    model->revertAll(); //如果不删除，则撤销
}
else model->submitAll(); //否则提交，在数据库中删除该行
}

```

这里删除行的操作会先保存在 model 中，当我们执行了 submitAll() 函数后才会真正的在数据库中删除该行。这里我们使用了一个警告框来让用户选择是否真得要删除该行。

运行效果如下：

我们点击第二行，然后单击“删除选中行”按钮，出现了警告框。这时你会发现，表中的第二行前面出现了一个小感叹号，表明该行已经被修改了，但是还没有真正的在数据库中修改，这时的数据有个学名叫脏数据(Dirty Data)。当我们按钮“Yes”按钮后数据库中的数据就会被删除，如果按下“No”，那么更改就会取消。

第五，插入操作。

我们进入“添加记录”按钮的单击事件槽函数，更改如下：

```

void Widget::on_pushButton_3_clicked() //插入记录
{
    int rowNum = model->rowCount(); //获得表的行数
    int id = 10;
    model->insertRow(rowNum); //添加一行
    model->setData(model->index(rowNum,0),id);
    //model->submitAll(); //可以直接提交
}

```

我们在表的最后添加一行，因为在 student 表中我们设置了 id 号是主键，所以这里必须使用 setData 函数给新加的行添加 id 属性的值，不然添加行就不会成功。这里可以直接调用 submitAll() 函数进行提交，也可以利用“提交修改”按钮进行提交。

运行程序，效果如下：

按下“添加记录”按钮后，就添加了一行，不过在该行的前面有个星号，如果我们按下“提交修改”按钮，这个星号就会消失。当然，如果我们将上面代码里的提交函数的注释去掉，也就不会有这个星号了。

可以看到这个模型很强大，而且完全脱离了 **SQL** 语句，就算你不怎么懂数据库，也可以利用它进行大部分常用的操作。我们也看到了，这个模型提供了缓冲区，可以先将修改保存起来，当我们执行提交函数时，再去真正地修改数据库。当然，这个模型比前面的模型更高级，前面讲的所有操作，在这里都能执行。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 3,406 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十七、Qt 数据库

### (七) QSqlRelationalTableModel

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

讲完 **QSqlTableModel** 了，我们这次讲这个类的扩展类

**QSqlRelationalTableModel**，它们没有太大的不同，唯一的就是后者在前者的基础上添加了外键（或者叫外码）的支持。

**QSqlRelationalTableModel**，该类为单张的数据库表提供了一个可编辑的数据模型，它支持外键。

我们还是新建 Qt4 Gui Application 工程，我这里工程名为 relationalTableModel，然后选中 QtSql 模块，Base class 选 QWidget。工程建好后，添加 C++ Header File，命名为 database.h，更改其内容如下：

```
#ifndef DATABASE_H
#define DATABASE_H
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open()) return false;
    QSqlQuery query;
    query.exec("create table student (id int primary key, name varchar, course
int)");
    query.exec("insert into student values (1,'yafei0',1)");
    query.exec("insert into student values (2,'yafei1',1)");
    query.exec("insert into student values (3,'yafei2',2)");
    query.exec("create table course (id int primary key, name varchar, teacher
varchar)");
    query.exec("insert into course values (1,'Math','yafeilinux1')");
    query.exec("insert into course values (2,'English','yafeilinux2')");
    query.exec("insert into course values (3,'Computer','yafeilinux3')");
    return true;
}
#endif // DATABASE_H
```

我们在这里建立了两个表，student 表中有一项是 course，它是 int 型的，而 course 表的主键也是 int 型的。如果要将 course 项和 course 表进行关联，它们的类型就必须相同，一定要注意这一点。

然后将 main.cpp 中的内容更改如下：

```
#include <QtGui/QApplication>
#include "widget.h"
#include "database.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if(!createConnection()) return 1;
    Widget w;
```

```
w.show();
return a.exec();
}
```

我们在 widget.h 中添加头文件： #include <QSqlRelationalTableModel>

然后在 private 中声明对象： QSqlRelationalTableModel \*model;

我们在 widget.ui 中添加一个 Table View 部件到窗体上，然后到 widget.cpp 中的构造函数里添加如下代码：

```
model = new QSqlRelationalTableModel(this);
model->setEditStrategy(QSqlTableModel::OnFieldChange); //属性变化时写入数据库
model->setTable("student");
model->setRelation(2,QSqlRelation("course","id","name"));
//将 student 表的第三个属性设为 course 表的 id 属性的外键，并将其显示为 course 表的 name 属性的值
model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
model->setHeaderData(1, Qt::Horizontal, QObject::tr("Name"));
model->setHeaderData(2, Qt::Horizontal, QObject::tr("Course"));
model->select();
ui->tableView->setModel(model);
```

我们修改了 model 的提交策略，OnFieldChange 表示只要属性被改动就马上写入数据库，这样就不需要我们再执行提交函数了。setRelation() 函数实现了创建外键，注意它的格式就行了。

运行效果如下：

可以看到 Course 属性已经不再是编号，而是具体的课程了。关于外键，你也应该有一定的认识了吧，说简单点就是将两个相关的表建立一个桥梁，让它们关联起来。

那么我们也希望，如果用户更改课程属性，那么他只能在课程表中有的课程中进行选择，而不能随意填写课程。在 Qt 中的 QSqlRelationalDelegate 委托类就能实现这个功能。我们只需在上面的构造函数的最后添加一行代码：

```
ui->tableView->setItemDelegate(new QSqlRelationalDelegate(ui->tableView));
添加代理（委托），在我这里不知为什么会出现 QSqlRelationalDelegate is not a type name 的提示，不过可以编译通过。
```

我们需要在 widget.cpp 中添加头文件： #include <QSqlRelationalDelegate>

运行效果如下：

可以看到这时修改 Course 属性时，就会出现一个下拉框，只能选择 course 表中的几个值。而利用这个类来操作数据库，与前面讲到的 **QSqlTableModel** 没有区别，这里就不再重复。这几篇文章一共讲了好几种操作数据库的方法，到底应该使用哪个呢？那就看你的需求了，根据这几种方法的特点进行选择吧。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：四月 30th, 2010. 2,422 views

Tags: [creator](#), [qt](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十八、Qt 数据库（八）XML（一）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

我们把 XML 放到数据库部分来讲，因为 XML 与数据库有着千丝万缕的联系，这里我们不再对 XML 进行过多的介绍，如果你还没有接触过它，可以在网上搜索一下关键字，其实对于我们下面讲述的内容，即便你不知道什么是 XML，你也会很快掌握的。

在 Qt 中提供了 QtXml 模块实现了对 XML 数据的处理，我们在 Qt 帮助中输入关键字 QtXml Module，可以看到该模块的类表。

在这里我们可以看到所有相关的类，它们主要是服务于两种操作 XML 文档的方法：DOM 和 SAX。Dom ( Document Object Model，即文档对象模型 ) 把 XML 文档转换成应用程序可以遍历的树形结构，这样便可以随机访问其中的节点。它的缺点是需要将整个 XML 文档读入内存，消耗内存较多。对于 SAX 我们放到后面再讲。

除了上面的两种方法外，Qt 还提供了简单的 QXmlStreamReader 和 QXmlStreamWriter 对 XML 文档进行读写。

下面我们先介绍使用 DOM 的方式来操作 XML 文档。

下面是一个规范的 **XML 文档**：

```
<?xml version="1.0" encoding="UTF-8"?> //XML 说明
<library> //根元素
<book id="01"> //library 元素的第一个子元素，“id”是其属性
<title>Qt</title> //book 元素的子元素，“Qt”是元素的文本
<author>shiming</author> //book 元素的子元素，title 元素的兄弟元素
</book> //结束标记名
<book id="02">
<title>Linux</title>
<author>yafei</author>
</book>
</library>
```

( 我们为了讲述方便使用了//注释，其实 XML 文档中是没有这些注释的 )

可以看到，一个规范的 XML 文档，是用 XML 说明开始的，主要由各元素组成。XML 文档第一个元素就是根元素，XML 文档必须有且只有一个根元素。元素是可以嵌套的。

下面我们就使用程序读出该文档中所有信息。

在 Qt Creator 中新建控制台工程 Qt4 Console Application，工程名为 xml01，在选择模块页选中 QtXml ( 如果在这里没有添加，就需要在工程文件中手动添加 QT += xml )。

下面我们更改 main.cpp 的内容如下：

```
#include <QtCore/QCoreApplication>
#include <QtXml>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QDomDocument doc; //新建 QDomDocument 类对象，它代表一个 XML 文档
    QFile file("my.xml"); //建立指向“my.xml”文件的 QFile 对象
    if (!file.open(QIODevice::ReadOnly)) return 0; //以只读方式打开
    if (!doc.setContent(&file)) { file.close(); return 0; }
    //将文件内容读到 doc 中
    file.close();
```

```
//关闭文件
QDomNode firstNode = doc.firstChild(); //获得 doc 的第一个节点，即 XML 说明
qDebug() << firstNode.nodeName() //输出 XML 说明
<< firstNode.nodeValue();
return a.exec();
}
```

我们先运行一下程序。然后在工程文件夹的 debug 文件夹下点击鼠标右键，新建文本文档，改名为“my.xml”，这里一定要注意把原来的“.txt”后缀改为“.xml”。然后我们利用记事本打开该文件，并将上面的 XML 文档的信息写入其中，不要写注释信息，然后保存。

再次运行程序，效果如下：

我们不愿意让输出信息出现双引号，可以更改程序代码：

```
qDebug() << qPrintable(firstNode.nodeName()) //输出 XML 说明
<< qPrintable(firstNode.nodeValue());
这里利用了 qPrintable() 函数。
```

效果如下：

下面我们在 return a.exec(); 代码前继续添加代码：

```
QDomElement docElem = doc.documentElement(); //返回根元素
QDomNode n = docElem.firstChild(); //返回根节点的第一个子节点
while(!n.isNull())
{
    //如果节点不为空
    if (n.isElement()) //如果节点是元素
    {
        QDomElement e = n.toElement(); //将其转换为元素
        qDebug() << qPrintable(e.tagName()) //返回元素标记
        << qPrintable(e.attribute("id")); //返回元素 id 属性的值
    }
    n = n.nextSibling(); //下一个兄弟节点
}
```

这样便能输出根元素及其子元素了。我们这里使用了 firstChild() 函数和 nextSibling() 函数，然后利用 while() 循环来实现对所有子元素的遍历。运行结果如下：

下面我们更改 if() 语句中的代码，用另一种方法遍历 book 元素的所有子元素。

```
if (n.isElement()) //如果节点是元素
{
    QDomElement e = n.toElement(); //将其转换为元素
    qDebug() << qPrintable(e.tagName()) //返回元素标记
    << qPrintable(e.attribute("id")); //返回元素 id 属性的值
    QDomNodeList list = e.childNodes(); //获得元素 e 的所有子节点的列表
    for(int i=0; i<list.count(); i++) //遍历该列表
    {
        QDomNode node = list.at(i);
        if(node.isElement())
```

```
qDebug() << " " << qPrintable(node.toElement().tagName())
<< qPrintable(node.toElement().text());
}
}
```

这里使用了 `childNodes()` 函数获得了元素所有子节点的列表，然后通过遍历这个列表实现了遍历其所有子元素。运行程序，效果如下：

**小结：**通过上面的例子，我们实现了对一个 XML 文档的读取。可以看到，在 QDom 中，是将整个 XML 文件读到内存中的 doc 对象中的。然后使用节点（QDomNode）操作 doc 对象，像 XML 说明，元素，属性，文本等等都被看做是节点，这样就使得操作 XML 文档变得很简单，我们只需通过转换函数将节点转换成相应的类型，如

`QDomElement e = n.toElement();`

在下一节我们将讲述 **XML** 文件的创建和写入。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 五月 4th, 2010. 3,886 views

Tags: [creator](#), [dom](#), [qt](#), [xml](#), [yafeilinux](#), [教程](#), [数据库](#)

## 二十九、Qt 数据库（九）XML（二）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一节中我们用手写的方法建立了一个 XML 文档，并且用 DOM 的方法对其进行了读取。现在我们使用代码来创建那个 XML 文档，并且对它实现查找，更新，插入等操作。

首先，我们新建 Qt4 Gui Application 工程，工程名为 xml02，然后添加 QDom 模块，我们选择 QWidget 为 Base class。

**1.**为了在程序中可以使用中文，我们先在 **main.cpp** 文件中添加头文件：

```
#include <QTextCodec>
```

然后在 main() 函数中添加一行代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

**2.**然后到 **widget.ui** 文件中，将界面设计如下：

其中用到的部件有 Push Button，ListWidget，Label 和 Line Edit。

**3.**我们再到 **widget.cpp** 文件中，添加头文件：#include <QDomDocument>

然后在构造函数中添加代码如下：

```
QFile file("my.xml");
if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
//只写方式打开，并清空以前的信息
QDomDocument doc;
QDomProcessingInstruction instruction; //添加处理指令
instruction = doc.createProcessingInstruction("xml",
"version=\\"1.0\\" encoding=\\"UTF-8\\\"");
doc.appendChild(instruction);
QDomElement root = doc.createElement(tr("书库"));
doc.appendChild(root); //添加根元素
//添加第一个 book 元素及其子元素
QDomElement book = doc.createElement(tr("图书"));
QDomAttr id = doc.createAttribute(tr("编号"));
QDomElement title = doc.createElement(tr("书名"));
QDomElement author = doc.createElement(tr("作者"));
QDomText text;
id.setValue(tr("1"));
book.setAttributeNode(id);
text = doc.createTextNode(tr("Qt"));
title.appendChild(text);
text = doc.createTextNode(tr("shiming"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);
//添加第二个 book 元素及其子元素
book = doc.createElement(tr("图书"));
id = doc.createAttribute(tr("编号"));
title = doc.createElement(tr("书名"));
author = doc.createElement(tr("作者"));
```

```

id.setValue(tr("2"));
book.setAttributeNode(id);
text = doc.createTextNode(tr("Linux"));
title.appendChild(text);
text = doc.createTextNode(tr("yafei"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);
QTextStream out(&file);
doc.save(out,4); //将文档保存到文件，4为子元素缩进字符数
file.close();

```

这样我们就建立起了一个 XML 文档，过程并不复杂，只要注意一下元素的父子关系就可以了。最后我们用 save() 函数将数据从 doc 中保存到文件中。

**4.**下面我们读取整个 XML 文档，我们从 **widget.ui** 中单击“查看全部信息”按钮，进入它的单击事件槽函数，并更改如下：

```

void Widget::on_pushButton_clicked() //显示全部
{
ui->listWidget->clear(); //先清空显示
QFile file("my.xml");
if (!file.open(QIODevice::ReadOnly)) return ;
QDomDocument doc;
if (!doc.setContent(&file))
{
file.close();
return ;
}
file.close();
//返回根节点及其子节点的元素标记名
QDomElement docElem = doc.documentElement(); //返回根元素
QDomNode n = docElem.firstChild(); //返回根节点的第一个子节点
while(!n.isNull()) //如果节点不为空
{
if (n.isElement()) //如果节点是元素
{
QDomElement e = n.toElement(); //将其转换为元素
ui->listWidget->addItem(e.tagName()+e.attribute(tr("编号")));
QDomNodeList list = e.childNodes();
for(int i=0; i<list.count(); i++)
{
QDomNode node = list.at(i);
if(node.isElement())
ui->listWidget->addItem(" "+node.toElement().tagName()
+" : "+node.toElement().text());
}
}
}

```

```
}

n = n.nextSibling(); //下一个兄弟节点
}
}
```

这里的代码就是上一节我们讲的读取 XML 文档所用的代码，只是将以前的 qDebug() 输出换成了在 listWidget 上进行输出。运行程序，单击按键，效果如下：

5.下面我们加入添加功能，进入“添加”按键的单击事件槽函数，更改如下：

```
void Widget::on_pushButton_5_clicked() //添加
{
    ui->listWidget->clear(); //我们先清空显示，然后显示“无法添加！”
    ui->listWidget->addItem(tr("无法添加！"));

    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
        file.close();
        return;
    }
    file.close();

    QDomElement root = doc.documentElement();
    QDomElement book = doc.createElement(tr("图书"));
    QDomAttr id = doc.createAttribute(tr("编号"));
    QDomElement title = doc.createElement(tr("书名"));
    QDomElement author = doc.createElement(tr("作者"));
    QDomText text;

    QString num = root.lastChild().toElement().attribute(tr("编号"));
    int count = num.toInt() + 1;
    id.setValue(QString::number(count));
    //我们获得了最后一个孩子结点的编号，然后加 1，便是新的编号
    book.setAttributeNode(id);
    text = doc.createTextNode(ui->lineEdit_2->text());
    //注意：你那可能不是 lineEdit_2。
    title.appendChild(text);
    text = doc.createTextNode(ui->lineEdit_3->text());
    author.appendChild(text);
    book.appendChild(title);
    book.appendChild(author);
    root.appendChild(book);
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
    QTextStream out(&file);
    doc.save(out,4); //将文档保存到文件，4 为子元素缩进字符数
    file.close();
    ui->listWidget->clear(); //最后更改显示为“添加成功！”
```

```
ui->listWidget->addItem(tr("添加成功！"));
}
```

这里先用只读方式打开 XML 文件，将其读入 doc 中，然后关闭。我们将新的节点加入到最前面，并使其“编号”为以前的最后一个节点的编号加 1。最后我们再用只写的方式打开 XML 文件，将修改完的 doc 写入其中。运行程序，效果如下：

我们先添加一个节点，然后再查看全部信息，发现确实已经添加成功了。

## 6. 查找，删除，更新操作。

因为这三个功能都要先利用“编号”进行查找，所以我们放在一起实现。

我们先在 widget.h 文件中添加一个函数声明：

```
void doXml(const QString operate);
```

它有一个参数，用来传递所要进行的操作。

然后到 widget.cpp 文件中对它进行定义：

```
void Widget::doXml(const QString operate)
{
    ui->listWidget->clear();
    ui->listWidget->addItem(tr("没有找到相关内容！"));
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
        file.close();
        return ;
    }
    file.close();
    QDomNodeList list = doc.elementsByTagName(tr("图书"));
    //以标签名进行查找
    for(int i=0; i<list.count(); i++)
    {
        QDomElement e = list.at(i).toElement();
        if(e.attribute(tr("编号")) == ui->lineEdit->text())
        {
            //如果元素的“编号”属性值与我们所查的相同
            if(operate == "delete") //如果是删除操作
            {
                QDomElement root = doc.documentElement(); //取出根节点
                root.removeChild(list.at(i)); //从根节点上删除该节点
                QFile file("my.xml"); //保存更改
                if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
                QTextStream out(&file);
                doc.save(out,4);
                file.close();
                ui->listWidget->clear();
                ui->listWidget->addItem(tr("删除成功！"));
            }
        }
    }
}
```

```
}

else if(operate == "update") //如果是更新操作
{
QDomNodeList child = list.at(i).childNodes();
//找到它的所有子节点，就是“书名”和“作者”
child.at(0).toElement().firstChild().setNodeValue(ui->lineEdit_2->text());
//将它子节点的首个子节点（就是文本节点）的内容更新
child.at(1).toElement().firstChild().setNodeValue(ui->lineEdit_3->text());
QFile file("my.xml"); //保存更改
if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
QTextStream out(&file);
doc.save(out,4); //将文档保存到文件，4为子元素缩进字符数
file.close();
ui->listWidget->clear();
ui->listWidget->addItem(tr("更新成功！"));
}

else if(operate == "find") //如果是查找操作
{
ui->listWidget->clear();
ui->listWidget->addItem(e.tagName()+e.attribute(tr("编号")));
QDomNodeList list = e.childNodes();
for(int i=0; i<list.count(); i++)
{
QDomNode node = list.at(i);
if(node.isElement())
ui->listWidget->addItem(" "+node.toElement().tagName()
+" : "+node.toElement().text());
}
}
}
}
}
}
```

```
然后我们分别进入“查找”，“删除”，“更新”三个按键的单击事件槽函数，更改如下：  
void Widget::on_pushButton_2_clicked() //查找  
{  
doXml("find");  
}  
void Widget::on_pushButton_3_clicked() //删除  
{  
doXml("delete");  
}  
void Widget::on_pushButton_4_clicked() //更新  
{  
doXml("update");  
}
```

这时，我们运行程序。

在“图书编号”中输入 1，然后点击“查找”按键，效果如下：

这时我们将下面的“书名”和“作者”进行更改，然后单击“更新”按键，效果如下：

再次点击“查找”按键：

然后点击“删除”按键：

再点击“查看全部信息”按键：

我们发现，所有的操作的结果都是正确的。

我们的 **Dom** 部分就讲到这里，下一节我们讲述用 **SAX** 的方法读取 **XML** 文档。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 5th, 2010. 3,154 views

Tags: [creator](#), [dom](#), [qt](#), [xml](#), [yafeilinux](#), [教程](#), [数据库](#)

## 三十、Qt 数据库（十）XML（三）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

我们前面讲述了用 DOM 的方法对 XML 文档进行操作，DOM 实现起来很灵活，但是这样也就使得编程变得复杂了些，而且我们前面也提到过，DOM 需要预先把整个 XML 文档都读入内存，这样就使得它不适合处理较大的文件。

下面我们讲述另一种读取 XML 文档的方法，即 SAX。是的，如果你只想读取并显示整个 XML 文档，那么 SAX 是很好的选择，因为它提供了比 DOM 更简单的接口，并且它不需要将整个 XML 文档一次性读入内存，这样便可以用来读取较大的文件。我们对 SAX 不再进行过多的介绍，因为现在你就可以掌握我们下面要讲的内容了。如果你对 SAX 有兴趣，可以到网上查找相关资料。

在 Qt 的 QtXml 模块中提供了一个 QXmlSimpleReader 的类，它便是基于 SAX 的 XML 解析器。这个解析器是基于事件的，但这些事件由它自身进行关联，我们并不需要进行设置。我们只需知道，当解析器解析一个 XML 的元素时，就会执行相应的事件，我们只需要重写这些事件处理函数，就能让它按照我们想法进行解析。

比如要解析下面的元素：

```
<title>Qt</title>
```

解析器会依次调用如下事件处理函数：startElement()，characters()，endElement()。我们可以在 startElement() 中获得元素名（如“title”）和属性，在 characters() 中获得元素中的文本（如“Qt”），在 endElement() 中进行一些结束读取该元素时想要进行的操作。而所有的这些事件处理函数我们都可以通过继承 QXmlDefaultHandler 类来重写。

下面我们先看一个简单的例子：

1. 新建空工程 Empty Qt4 Project，工程名为 xml03。
2. 我们在工程中添加 C++ Class，Class name 为 MySAX，Base class 为 QXmlDefaultHandler
3. 我们再添加一个 main.cpp 文件。
4. 然后我们在 xml03.pro 文件中加入一行代码：QT += xml

下面是几个文件的内容：

main.cpp 文件：

mysax.h 文件：

mysax.cpp 文件：

（注：上面程序中的注释//开始读写元素，应为//开始读取元素）

我们先运行一下程序，然后将第一节我们建立的“**my.xml**”文件复制到我们现在的工程文件夹的 **debug** 文件夹下。

然后再运行程序，效果如下：

可以看到文件的解析过程如下：

```
QFile file(fileName);
QXmlInputSource inputSource(&file); //读取文件内容
QXmlSimpleReader reader; //建立 QXmlSimpleReader 对象
reader.setContentHandler(this); //设置内容处理程序
reader.setErrorHandler(this); //设置错误处理程序
```

```
reader.parse(inputSource); //解析文件
```

这里，`setContentHandler()`就是设置了`startElement()`, `characters()`, `endElement()`等事件的处理程序。而且我们一般都要设置错误处理程序`setErrorHandler()`，最后我们使用`parse()`来对文件进行解析，在解析过程中会不停地调用事件处理函数，当然整个调用过程是在内部进行的，不用我们去进行设置。利用 SAX 读取 XML 文档是十分方便快速的。

我们 XML 的内容就讲到这里，当然，Qt 还提供了基于流的 XML 处理类，Qt 也提供了一个`QtXmlPatterns` 模块来进行 XML 文档的处理，这些内容我们就不再进行讲述。

分类: [Qt 系列教程](#) 作者: yafeilinux 日期: 五月 6th, 2010. 2,969 views

Tags: [creator](#), [qt](#), [SAX](#), [xml](#), [yafeilinux](#), [教程](#), [数据库](#)

## 三十一、Qt 4.7.0 及 Qt Creator 2.0 beta 版安装全程图解

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

现在诺基亚 Qt 官网上已经提供了最新的 Qt 4.7.0 和 Qt Creator 2.0 beta 版的下载，我们第一时间对其进行了测试使用，并将其作为我们系列教程的一部分，来和大家一起尝鲜全新的 Qt。

说明：下面我们只是下载了 Qt 的桌面版开发环境，其中并没有包含移动平台的开发库。我们会在这个专题的最后讲解 Nokia Qt SDK 完整版的安装和使用。

**1. 我们到官方网站下载相关软件。**

<http://qt.nokia.com/developer/qt-qtcreator-prerelease>

**2. 我们需要分别下载 Qt 及 Qt Creator。**

**3. 我们先安装 qt-win-opensource-4.7.0-beta1-mingw，安装开始时的设置全部使用默认设置即可。在最后会弹出如下警告框，我们选择“是”即可。**

**4. 然后我们安装 qt-creator-win-opensource-2.0.0-beta1，全部保持默认设置即可。**

**5. 下面我们打开安装好的 Qt Creator，其界面如下。**

可以看到，新的 Qt Creator 的界面有所变化。

**6. 我们先新建常用的 Gui 工程 Qt Gui Application，工程名设为“hello”，然后设置下面的保存路径，注意路径中不能有中文。后面的选项全部默认即可。**

**7. 建立好工程后，界面如下。**

**8. 我们双击 mainwindow.ui 文件，这时便进入了界面设置模式，我们在窗口上添加一个 Label 部件，然后更改其内容为“hello world！”效果如下。**

可以看到，新的 Qt Creator 中添加了一个界面设计模式的选项，我们可以更方便的从代码编辑界面转到可视的窗口设计界面。

**9. 这时我们再回到 Edit 界面中，可以看到 ui 文件的内容，它是一个 XML 文件。**

**10. 这时我们点击 Run 按钮，运行程序，如果这时出现下面的提示框，我们选中其中的选择框，然后点击“Save All”按钮即可。**

说明：下面的几步只是在以前没有装过 Qt 的系统上才会出现，如果你以前已经安装过 Qt 的其他版本，这里的情况有所不同。不过，你也可以按照下面的方法将你新安装的 Qt Creator 设置为最新的 Qt4.7.0。

**11. 此时程序并没有成功运行，而是输出了下面的错误。**

Qt version **Qt in PATH** is invalid. Set valid Qt Version in Tools/Options  
Could not determine the path to the binaries of the Qt installation, maybe the  
qmake path is wrong?

Error while building project hello (target: Desktop)  
When executing build step 'qmake'

如下图：

错误提示的意思是：在 PATH 系统环境变量中的 Qt 版本信息不可用，我们需要在 Tools/Options 菜单中进行设置。

**12.**我们进入 **Qt Creator** 的 **Tools** 菜单，再进入其 **Options** 子菜单。然后在 **Qt4** 页  
面可以进行 **Qt** 的版本设置。

可以看到，这时还没有可用的 Qt 版本。我们有两种方法进行设置，一种是去设置环境变量，  
一种是直接在这里添加 Qt 的安装路径。

**方法一：**

我们在桌面上右击“我的电脑”，然后选择“属性”，然后选择“高级”页面，再选择下面的“环  
境变量”，如下图。

我们选择下面的列表中的“Path”项，点击“编辑按钮”。如下图。

这时我们将我们的 Qt4.7.0 的 bin 文件夹的路径加入环境变量中，我这里是“C:\Qt\4.7.0-  
beta1\bin”，如果你开始按照默认路径安装，也应该是这个路径，如果不是，请改为自己的  
路径。我们需要将“;C:\Qt\4.7.0-beta1\bin”加入变量值中，注意这里前面有一个英文半角  
的“;”号，它用来将我们的变量值与已有的变量值隔开。如下图：

然后我们点击确定就可以了。

**方法二，在下面附录中给出。**

**13.**我们设置好环境变量后，需要关闭 **Qt Creator**，然后重新打开。

这时我们再进入 Qt 版本设置界面，发现它已经发现 Qt in PATH 了，我们点击 Qt in  
PATH，在下面的 MinGW Directory 中设置一下 MinGW 文件夹的路径，也就是我们安装的  
Qt Creator 的目录下的 mingw 文件夹的路径，我这里是“C:\Qt\qtcreator-  
1.3.83\mingw”。设置好后按“OK”按钮即可。如下图。

**14.**我们在 **Qt Creator** 的 **File** 菜单中选择 **Recent Projects**，然后打开我们上次关  
闭的工程。

如下图。

这时我们再次运行程序，已经成功了。效果如下。

可以看到，新的 **Qt Creator** 的界面有所不同，但是它在编写普通程序时，与以前的  
版本差别不大。而它支持的全新的 **Qt Quick** 编程界面，我们会在下一节进行讲述。

附录：手动设置 **Qt** 的版本信息。

我们在上面讲述了通过设置环境变量来设置 Qt 的版本信息，其实，我们也可以直接通过手动  
添加 Qt 的路径来设置 Qt 的版本。方法如下。

**1.**我们在 **Qt** 版本设置界面单击右上角的“+”号按钮，这时效果如下。

**2.**然后将下面的信息添加完整，如下。

**3.**我们重新打开我们的工程文件，这时如果你已经用过方法一进行环境变量设置，并且已经运行过该程序了，那么可能会弹出如下窗口。

我们选择第二项，在新的文件夹下编译该工程即可。这时运行程序，便没有错误了。

说明：第二种方法看似很简单，但是它存在一些缺点。比如，我们想直接双击运行工程文件夹下已经编译好的.exe 文件时，就会出现缺少.dll 文件的提示。而第一种方法就不会再出现这样的提示。所以我们推荐使用方法一。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 11th, 2010. 3,951 views

Tags: [beta](#), [creator](#), [qt](#), [Qt 4.7](#), [Qt 4.7.0](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt4.7](#), [Qt4.7.0](#), [Qt 版本](#), [yafeilinux](#), [安装](#), [教程](#), [环境变量](#)

## 三十二、第一个 Qt Quick 程序 ( QML 程序 )

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

上一节我们详细讲述了 Qt 4.7 和 Qt Creator 2.0 的安装。这一节，我们讲述 Qt Quick 的应用。

Qt Quick 是 Qt 4.7 主推的技术，下面是 Qt 官网对其进行的介绍：

Qt Quick 是一种高级用户界面技术，使用它可轻松地创建供移动和嵌入式设备使用的动态触摸式界面和轻量级应用程序。三种全新的技术共同构成了 Qt Quick 用户界面创建工具包：一个改进的 Qt Creator IDE、一种新增的简便易学的语言 (QML) 和一个新加入 Qt 库中名为 QtDeclarative 的模块，这些使得 Qt 更加便于不熟悉 C++ 的开发人员和设计人员使用。

下面我们先到 Qt Creator 中查看相关帮助文件，让我们对它有个大体的了解。

我们查看 Qt Creator 中的帮助 Help，首先进入我们眼帘的便是 Qt Creator 的介绍，在这里你可以查看 Qt Creator 的相关信息和使用方法。

我们查看下面的目录，可以看到这里有简单的工程的建立教程。我们进入 Creating an Animated Application 的链接，这个便是一个最简单的 QML 工程的教程，你可以参考一下。

下面我们建立自己的 QML 工程。

1. 新建 Qt QML Application，工程名设置为 helloWorld。

2. 我们点击 helloWorld.qmlproject 文件。

在这里可以看到它就是包含了几个文件夹的路径信息，默认的都是本工程文件夹。

在最上面，有一句提示，Do you want to enable the experimental Qt Quick Designer? 你是否要启用实验中的 Qt Quick Designer？当然，所以我们点击后面的按钮来启用 Qt Quick Designer。

3. 这时弹出一个提示框。

它的大体内容是，如果启用 Qt Quick Designer，将影响 Qt Creator 的整体稳定性。还告诉了我们怎么停用 Qt Quick Designer。我们选择“Enable Qt Quick Designer”。

4. 我们关闭 Qt Creator，然后重新打开它。我们再次打开刚才建立的工程。

双击 helloWorld.qml 文件，这时我们期盼已久的 Qt Quick Designer 界面终于出现了。对于这个界面，我们以后再详细讲解。

5. 我们再次回到 Edit 模式下，查看 helloWorld.qml 文件的内容。

```
import Qt 4.6
Rectangle {
    width: 200
    height: 200
    Text {
        x: 66
        y: 93
        text: "Hello World"
    }
}
```

这就是传说中的 QML 语言了，看上去有点像 CSS，就像官网所说的，它是 JavaScript 的扩展。我们这里先不对这些代码做什么解释，到后面会专门来讲这个语言的。

6. 我们这时运行程序，效果如下。

7. 我们更换一下程序的皮肤。

在 skin 菜单中选择一个皮肤。

运行效果如下：

我们可以在其上右击鼠标，选择 Quit 菜单，退出程序。

8. 关于停用 Qt Quick。

我们打开 Help 菜单，进入 About Plugins 子菜单。然后将 Qt Quick 项的对勾去掉即可。

到这里，一个最简单的 Qt Quick 程序就完成了。我们可以看到，这是一个全新的体验，它与以前的 Qt 应用是完全不同的。在以后的教程里我们会对 Qt Quick 及其包含的 QML 语言进行全面的讲解。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 11th, 2010. 3,629 views

Tags: [creator](#), [QML](#), [qt](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt4.7](#), [yafeilinux](#), [教程](#)

## 三十三、体验 QML 演示程序

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

前面我们讲述了简单的 QML 工程的建立。对于一个新生的事物，我们最感兴趣的是它到底能做什么，而对于 QML，Qt 已经自带了大量的演示程序，我们可以体验一下 QML 的强大。

1.所有的小例子都在 Qt4.7 的安装目录下的 examples\declarative 目录下。

我这里的路径是 C:\Qt\4.7.0-beta1\examples\declarative 。

2.运行 qml 文件的方法：

**方法一：使用 qml.exe**

所有的 qml 文件都需要使用一个叫 qml.exe 的程序来运行。它在 C:\Qt\4.7.0-beta1\bin 目录下。我们双击 qml.exe，然后用它打开一个 qml 文件，效果如下：

为了以后方便使用，我们可以让 qml.exe 成为 qml 文件的默认运行程序。我们在任意一个 qml 文件上右击鼠标，在“打开方式”菜单中选择“选择程序...”然后找到 qml.exe，并选中下面的“始终使用选择的程序打开这种文件”选择框，然后按“确定”，以后就可以直接双击打开 qml 文件了。

**方法二：在 Qt Creator 中打开**

为了安全起见，我们先将要使用的文件夹复制出来，比如我们将 C:\Qt\4.7.0-beta1\examples\declarative 下的 animations 文件夹复制到 F 盘中。然后我们打开 Qt Creator，新建工程，这里选择 Import Existing Qt QML Directory 一项。如下图。

然后填写工程名，路径指定到我们复制的 animations 文件夹。

完成后我们点击 Qt Creator 工程列表中的任意一个 qml 文件，然后点击运行按钮，就可以执行了。

注意：演示程序中可能存在一些错误或警告，比如上面的 color-animation.qml 使用第一种方法，直接用 qml.exe 打开是无法执行的，我们只能使用第二种方法运行它。

3.除了在 examples\declarative 中的这些小例子外，Qt 还提供了几个较大的例子。

它们在 C:\Qt\4.7.0-beta1\demos\declarative 目录中，不过现在这里的大部分 qml 文件都不能直接执行，我们都需要使用上面的方法二，在 Qt Creator 中执行它们。下面是其中几个的简单介绍：

Samegame 的例子：

webbrowser 的例子：

注意这个例子需要能上网才能正常运行，我们可以使用键盘上的左右方向键来缩放网页。

photoviewer 的例子：

这个例子也是自动从网上下载图片的，速度有点慢。

4.查看源代码。

看完了 qml 文件运行时的绚丽效果，我们下面来查看一下它的源代码。我们当然可以在 Qt Creator 中直接查看其代码，其实更简单的方法是，使用记事本或写字板等文本编辑器直接打开 qml 文件。下面是一个 qml 文件的内容。

现在你可以发现了，利用 QML，编写几句简单的代码就能实现让人心动的界面。那么我们就在以后的教程中好好学习一下这个让人期待的 QML 语言吧。

**体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !**

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 18th, 2010. 3,539 views

Tags: [creator](#), [declarative](#), [demo](#), [QML](#), [qml.exe](#), [qt](#), [Qt 4.7](#), [Qt Creator 2.0](#), [Qt Quick](#), [yafeilinux](#)

## 三十四、Qt Quick Designer 介绍

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在对 Qt Quick 和 QML 有了一些了解后，现在我们就来熟悉一下这个全新的 Qt Quick Designer 界面。在这一篇中我们会用一个例子来介绍一下 Qt Quick Designer 界面，但是不会对代码进行过多的讲解。到下一篇我们讲解 QML 组件时会对这个程序生成的代码进行逐行讲解，再往后的几篇，我们就会对 Qt Quick 中的几个特色功能进行举例讲解，而在讲解例子的同时，我们也会将 QML 语言的讲述加入其中。我们会在完成几个实例的同时掌握 QML 语言。

我们新建一个 Qt QML Application，我这里的工程名为“helloWorld”。

### 一，熟悉 Qt Quick Designer 界面

这是整个 Qt Quick Designer 界面，它由几个面板组成，下面分别进行介绍。

主设计面板，也就是我们下面所说的场景。这是我们的主设计区，所有的项目都要放到这里，当程序执行时，就是显示的这个面板上的内容。

Navigator 导航器面板。场景中所有的项目都在这里列出。在这里，我们可以选中一个特定的项目，那么场景中对应的项目也会被选中，我们也可以在这里拖拽项目来更改它们的从属关系，或者点击项目后面的“眼睛”图标来设置它是否可以显示。

Properties 属性面板。在这里我们可以设置项目的属性。比如项目名称，填充颜色，边框颜色，项目大小和位置，以及项目的缩放，旋转和不透明度等。

Library 库面板。在这里提供了一些常用的部件，我们可以将它们拖放到场景中来使用。在 Resources 子面板中，我们可以看到我们工程文件夹下的图片等资源，我们也可以将它们拖拽到场景中直接使用。当我们以后新建了组件后，它也会出现在 Library 中。

状态面板。这里可以为场景新建或删除一个状态。

设计模式和代码编辑模式的转换，我们可以点击“Edit”图标进入代码编辑界面。

### 二，简单的使用。

1. 我们从库面板中拖入一个 Rectangle 到场景中，调整它的大小。然后在属性面板中更改其 ID 为“myButton”，并更改其颜色。将其 Radius 属性更改为 10，这时它就会变为圆角了。这时的属性面板如下。

2. 然后我们从库面板中拖入一个 Mouse Area 部件到“myButton”上，注意，要使得 Mouse Area 部件成为“myButton”的子项目，在导航器面板中可以看到它们的关系。

这时我们选中了这个 Mouse Area 部件，在属性面板中将其 id 改为“myMouseArea”，然后在 Geometry 子面板中点击图标，鼠标区域填充整个“myButton”。

3. 这时我们在状态面板中点击一下后面的带有加号的方块，新建一个状态。如下图。

我们在 State1 状态下，更改场景中的“Hello World”，改变它的字体大小，并更改颜色。如下图。

然后我们进入 Advanced 子面板，更改一下 Opacity（不透明度）和 Rotation（旋转）的值。如下图。

4. 我们下面点击“Edit”图标，进入代码编辑界面。

在这里我们找到 state 代码段，添加一行代码 when: myMouseArea.pressed，如下：

```
states: [
State {
name: "State1"
when: myMouseArea.pressed
这里我们省略了其他代码。
```

5. 此时我们运行程序，效果如下。

然后我们点击按钮，效果如下：

这样这个简单的例子就讲完了。我们可以注意到，当我们在设计器中更改界面时，编辑器中的代码就自动改变了。其实，如果我们在编辑器中更改了代码，对应的设计器中的界面也会相应改变的。这样就实现了真正的所见即所得。在后面的章节中你会进一步体会到 Qt Quick 实现一个炫酷界面是那么简单。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com)！

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 26th, 2010. 2,367 views

Tags: [creator](#), [declarative](#), [QML](#), [qt](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt Quick Designer](#), [Qt4.7](#), [yafeilinux](#), 教程

```

State {
    name: "state1"
    PropertyChanges {
        target: myText
        font.pointSize:20; color:"red"
        rotation:30; opacity:0.5
    }
    when:myMouseArea.pressed
}
]

```

这里我们使用了“[ ]”括号，它表示列表属性，也就是说，在方括号中间可以写多个项目作为其属性值，它们需要用英文“,”隔开。例如：

```
[
State{},
State{}
]
```

因为我们这里只有一个项目，所以也可以省去方括号。在 State{} 中，我们又使用了 PropertyChanges{} 项目，它用来改变特定项目的属性值。这里我们先指定了要改变的项目是 “myText”，然后再更改其属性值。这里的 opacity 属性，是不透明度，当为 0 时表示全透明，为 1 时表示完全不透明，所有项目的该属性默认值都是 1。

下面一句 when:myMouseArea.pressed，表示当鼠标区域被点击时，进入该状态。这里需要说明的是，因为默认的开始状态，就是不做改变时的状态，所以我们要指定什么时候进入我们新建的状态。

（其实我们也可以将“state1”作为默认的状态，我们可以在“mainWindow”中添加一行代码：state: “state1” 这样就表明了使“state1”作为初始状态。但这时我们要将 when:myMouseArea.pressed 一行代码删去。）

此时我们再进入 Design 界面，效果如下：

5.这时整个程序的代码如下：

```

import Qt 4.6
Rectangle{
id: mainWindow; width:200; height:200
Text {
id: myText
text: "Hello World!"
anchors.centerIn:parent
}
Rectangle{
id:myButton; width:50; height:30
x:75; y:136; radius:10; color:"blue"
MouseArea{id:myMouseArea; anchors.fill:parent}
}
states: [
State {
name: "state1"
PropertyChanges {
target: myText
font.pointSize:20; color:"red"
rotation:30; opacity:0.5
}
when:myMouseArea.pressed
}
]
}

```

运行程序，效果如下：

上面我们一步一步重写了整个程序，并在每一步完成时都查看了设计器中的界面。就像前一节所说的那样，Qt Quick 实现了真正的可视化编程，代码与界面完全同步，让用户可以直观的看到执行结果。这个程序很小可以这样实现，但是如果稍大点的程序，所有的项目的定义都写在一个文件中，就会显得很繁琐。而且，像按钮一样的部件，我们可能需要多次使用，我们也不愿意重复编写代码。所以，我们需要将子项目单独来写，这样就形成了组件。

#### QML 组件

1.我们在工程中新建 Qt QML File，命名为“Button”。

注意：组件的名称的首字母一定要大写。

2.然后我们将 Button.qml 文件中的内容更改如下：

```
import Qt 4.7
Rectangle{
    id:myButton; width:50; height:30
    radius:10; color:"blue"
    signal clicked()
    MouseArea{id:myMouseArea; anchors.fill:parent
    onClicked: myButton.clicked()
    }
}
```

这里我们新建了一个 signal 信号函数 clicked()。当执行该函数时，Button 组件就会发出 clicked() 信号。我们在 MouseArea 中添加了一行代码：

```
onClicked: myButton.clicked()
```

其中 onClicked 关联到 MouseArea 的 clicked() 信号，它相当于槽函数，这行代码的效果是，当 MouseArea 被按下时，就执行“myButton”的 clicked() 函数。

3.我们在 helloWorld.qml 中删除以下代码

```
Rectangle{
    id:myButton; width:50; height:30
    x:75; y:136; radius:10; color:"blue"
    MouseArea{id:myMouseArea; anchors.fill:parent}
}
```

和 when:myMouseArea.pressed

4.我们进入 helloWorld.qml 的设计器界面。

可以看到，在库面板，已经有 Button 元件了，我们将一个 Button 拖入场景中。如下图。

5.我们回到 helloWorld.qml 文件中，在新增的代码中添加一行代码。

```
Button {
    id: button1
    x: 75
    y: 135
    onClicked: mainWindow.state = "state1"
}
```

这个 onClicked 就是对应按钮的 clicked 信号。当按钮被按下时，我们让主窗口进入新建的状态。

6.此时执行程序，效果如下。

我们可以看到，QML 组件是可以直接在其他文件中使用的，我们不需要进行声明，或者包含头文件。而工程中的一个组件，会自动添加到 Library 库中供我们使用。我们也看到了 QML 中的信号和槽也十分简单。在这一节中我们讲述了 QML 语言的一些基本知识，而且引出了组件的概念，最后还讲述了信号的关联。这部分内容很多，也有些乱，我们最好多进行一些练习，对于各个项目的属性可以查看帮助文件。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !

分类：[Qt 系列教程](#) 作者：[yafeilinux](#) 日期：五月 26th, 2010. 2,488 views  
Tags: [creator](#), [QML](#), [QML组件](#), [qt](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt Quick Designer](#), [Qt4.7](#)

## 三十五、QML 组件

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一节中，我们简单讲述了 Qt Quick Designer 的使用。现在我们开始讲述程序中生成的代码，然后引出 QML 组件的概念。这一节也包含了对 QML 语言的简单讲解。

我们先来看一下上一次生成的代码。

首先看一下整个文件的框架：

```
import Qt 4.6 //QML 文件都要以 import 开头
Rectangle { //这里定义了一个矩形，以及它包含的所有子项目
    id: mainWindow //矩形的 id 属性，相当于 C++ 中的对象名，需要用小写字母开头
    width: 200 //矩形的宽和高
    height: 200
    //下面是矩形的三个子项目
    Text {
        //这里定义了文本的内容
        ...
    }
    Rectangle {
        //这里定义了一个矩形，以及该矩形中的所有子项目
        ...
    }
    states: [
        //这里是一系列状态
        ...
    ]
}
```

下面我们查看每一个子项目的内容：

文本项目：

```
Text {
    id: text1 //id 属性
    x: 66 //在父项目 (mainWindow) 中的位置
    y: 93
    text: "Hello World" //文本内容
}
```

矩形项目：

```
Rectangle {
    id: myButton //属性
    x: 50
    y: 134
    width: 100
    height: 27
    color: "#8d4848"
    radius: 10
    MouseArea { //子项目
        id: myMouseArea
        x: 12
        y: 6
        width: 100
        height: 100
        anchors.fill: parent //填充整个父项目 (myButton)
        anchors.bottomMargin: 0
        anchors.topMargin: 0
        anchors.leftMargin: 0
        anchors.rightMargin: 0
    }
}
```

```

状态项目：
states: [
State {
name: "State1" //新建的状态
when: myMouseArea.pressed //当鼠标按下时启用
PropertyChanges { //属性改变
target: text1 //目标是 text1 项目
x: 25 //下面是对 text1 各属性的改变
y: 48
width: 154
height: 57
color: "#38883e"
rotation: 30
scale: 1
opacity: 0.5
font.bold: false
verticalAlignment: "AlignVCenter"
horizontalAlignment: "AlignHCenter"
font.pointSize: 20
}
}
]

```

以上除了我们添加的那行代码外，其他的代码全部是自动生成的，而且格式很规范。其实这些代码我们完全可以自己来写。下面我们就重写上面的代码，再一点一点得实现整个程序的功能。

### 1.添加主窗口。

```

import Qt 4.6
Rectangle{
id: mainWindow; width:200; height:200
}

```

在 QML 中，必须有一个根元素，第一个元素即是根元素，它将做为所有项目的父项目。对于项目的属性，可以一个属性写一行，也可以像上面这样，所有的属性都写在一行，不过中间要用英文的“;”分号隔开。对于 id 属性，它和我们接触过的变量名一样，必须用小写字母或下划线开头，且不能包含字母，数字和下划线以外的字符。

这时我们进入 Design 界面，其内容如下：

### 2.添加文本项目。

因为其是“mainWindow”的子项目，所以 Text{}要写在“mainWindow”的两个{}号里面。  
我们在下面输入“Tex”，这时会自动弹出待选列表，我们发现列表中第一个就是“Text”，所有我们直接按下回车键，让系统自动补全输入。

可以看到系统自动生成了“id”和“text”两个属性值。我们只需更改其内容即可。然后我们再添加文本的位置属性。其内容如下：

```

Text {
id: myText
text: "Hello World!"
anchors.centerIn:parent
}

```

这里的 anchors 是一个布局管理器，它可以固定项目的位置，我们这里的 anchors.centerIn:parent 就是表示“myText”要在其父项目的中心。我们这时再查看 Design 界面，效果如下：

### 3.添加矩形项目。

我们添加一个矩形，作为按钮。它也是“mainWindow”的子项目。而在按钮中还有一个 MouseArea 子项目，它填充整个按钮，因为只有添加了这个 MouseArea，按钮才能接受鼠标点击事件。又因为，

MouseArea 只是填充了按钮，所以只有在按钮上点击，才会触发鼠标点击事件，而点击其他地方，是没有效果的。代码如下：

```
Rectangle{
id:myButton; width:50; height:30
x:75; y:136; radius:10; color:"blue"
MouseArea{id:myMouseArea; anchors.fill:parent}
}
```

这里有个 radius 属性，它是矩形的半径值，简单的说就是它可以使矩形的四个角变为弧线。现在我们在看一下 Design 界面，效果如下：

#### 4. 状态项目。

说是状态项目，更确切的说是“mainWindow”的状态属性，然后在其状态属性中又包含状态项目。

states: [

```
State {
    name: "state1"
    PropertyChanges {
        target: myText
        font.pointSize:20; color:"red"
        rotation:30; opacity:0.5
    }
    when:myMouseArea.pressed
}
```

这里我们使用了“[ ]”括号，它表示列表属性，也就是说，在方括号中间可以写多个项目作为其属性值，它们需要用英文“,”隔开。例如：

```
[  
State{},  
State{}  
]
```

因为我们这里只有一个项目，所以也可以省去方括号。在 State{} 中，我们又使用了 PropertyChanges{} 项目，它用来改变特定项目的属性值。这里我们先指定了要改变的项目是“myText”，然后再更改其属性值。这里的 opacity 属性，是不透明度，当为 0 时表示全透明，为 1 时表示完全不透明，所有项目的该属性默认值都是 1。

下面一句 when:myMouseArea.pressed，表示当鼠标区域被点击时，进入该状态。这里需要说明的是，因为默认的开始状态，就是不做改变时的状态，所以我们要指定什么时候进入我们新建的状态。

（其实我们也可以将“state1”作为默认的状态，我们可以在“mainWindow”中添加一行代码：state：“state1”这样就表明了使“state1”作为初始状态。但这时我们要将 when:myMouseArea.pressed 一行代码删去。）

此时我们再进入 Design 界面，效果如下：

#### 5. 这时整个程序的代码如下：

```
import Qt 4.6
Rectangle{
id:mainWindow; width:200; height:200
Text {
id: myText
text: "Hello World!"
anchors.centerIn:parent
}
Rectangle{
id:myButton; width:50; height:30
x:75; y:136; radius:10; color:"blue"
MouseArea{id:myMouseArea; anchors.fill:parent}
}
states: [
State {
```

```
name: "state1"
PropertyChanges {
target: myText
font.pointSize:20; color:"red"
rotation:30; opacity:0.5
}
when:myMouseArea.pressed
}
]
}
```

运行程序，效果如下：

上面我们一步一步重写了整个程序，并在每一步完成时都查看了设计器中的界面。就像前一节所说的那样，Qt Quick 实现了真正的可视化编程，代码与界面完全同步，让用户可以直观的看到执行结果。这个程序很小可以这样实现，但是如果稍大点的程序，所有的项目的定义都写在一个文件中，就会显得很繁琐。而且，像按钮一样的部件，我们可能需要多次使用，我们也不愿意重复编写代码。所以，我们需要将子项目单独来写，这样就形成了组件。

QML 组件

1.我们在工程中新建 Qt QML File，命名为“Button”。

注意：组件的名称的首字母一定要大写。

2.然后我们将 Button.qml 文件中的内容更改如下：

```
import Qt 4.7
Rectangle{
id:myButton; width:50; height:30
radius:10; color:"blue"
signal clicked()
MouseArea{id:myMouseArea; anchors.fill:parent
onClicked: myButton.clicked()
}
}
```

这里我们新建了一个 signal 信号函数 clicked()。当执行该函数时，Button 组件就会发出 clicked() 信号。我们在 MouseArea 中添加了一行代码：

```
onClicked: myButton.clicked()
```

其中 onClicked 关联到 MouseArea 的 clicked() 信号，它相当于槽函数，这行代码的效果是，当 MouseArea 被按下时，就执行“myButton”的 clicked() 函数。

3.我们在 helloWorld.qml 中删除以下代码

```
Rectangle{
id:myButton; width:50; height:30
x:75; y:136; radius:10; color:"blue"
MouseArea{id:myMouseArea; anchors.fill:parent}
}
```

和 when:myMouseArea.pressed

4.我们进入 helloWorld.qml 的设计器界面。

可以看到，在库面板，已经有 Button 元件了，我们将一个 Button 拖入场景中。如下图。

5.我们回到 helloWorld.qml 文件中，在新增的代码中添加一行代码。

```
Button {
id: button1
x: 75
y: 135
onClicked: mainWindow.state = "state1"
}
```

这个 onClicked 就是对应按钮的 clicked 信号。当按钮被按下时，我们让主窗口进入新建的状态。

6.此时执行程序，效果如下。

我们可以看到，QML 组件是可以直接在其他文件中使用的，我们不需要进行声明，或者包含头文件。而工程中的一个组件，会自动添加到 Library 库中供我们使用。我们也看到了 QML 中的信号和槽也十分简单。在这一节中我们讲述了 QML 语言的一些基本知识，而且引出了组件的概念，最后还讲述了信号的关联。这部分内容很多，也有些乱，我们最好多进行一些练习，对于各个项目的属性可以查看帮助文件。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com)！

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 26th, 2010. 2,490 views

Tags: [creator](#), [QML](#), [QML 组件](#), [qt](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt Quick Designer](#), [Qt4.7](#)

## 三十六、QML项目之Image和BorderImage

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

从这一节开始我们讲述 QML 的一些基本组成元素和项目。正是这些部件才使得 QML 变得使用简单但功能强大。

我们先打开 Qt Creator，然后进入帮助模式，在左上角选择 Contents 方式显示。我们点击 Qt Reference Documentation 一项，这时就显示出了 Qt4.7 的内容列表。如下图。

我们选择 Qt API Overviews 的 UI design & Qt Quick 一项。如下图。

这里有 Declarative UI 和 QML 的概述，下面列有一些教程和 QML 相关内容的文章列表。我们可以查看自己所需要的内容。我们点击 Reference 下面的 QML Elements 一项。

这里列出了所有的标准 Qt Declarative 元素和 QML 项目。如下图。

我们以后可以在这里查看自己需要用到的元素和项目的帮助文件。在这一节里，我们主要介绍 QML Items 中的 Image 和 BorderImage 两个项目。

### 一、Image 图片

1. 我们新建 Qt QML Application，工程名为“myImage”。

2. 我们在库面板中拖入一个 Image 到场景中，在属性面板中我们可以选择一张图片的路径。

其实，更好的方法是，我们将图片放到工程文件夹中，这样在库面板的资源栏中我们就可以直接看到该图片了，而且该图片也会显示在工程文件列表中。

3. 我们将两张图片放到工程文件夹下。如下图。

然后在库面板的资源页面查看添加的图片。我们可以将图片直接拖入场景。如下图。

在工程文件列表中也会显示这两个图片。

4. 图片平铺方式。

在属性面板中我们可以设置图片的平铺方式 Fill Mode。如下图。

Stretch：默认选择的是 Stretch 一项，表示拉伸图片。就是说当将图片缩放时会拉伸图片。

其效果如下：

当图片缩放或者旋转时我们也可以选中属性面板中 Smooth 一项，使图片变得平滑。效果如下：

PreserveAspectFit：拉伸时缩放图片，总是显示完整图片。

PreserveAspectCrop：拉伸时缩放图片，但是可能对图片进行裁剪。

Tile：平铺图片。

TileVertically：竖直平铺图片。

TileHorizontally：水平平铺图片。

5. 使用网络上的图片。

我们也可以使用网络上的图片，直接在属性面板上更改图片路径为图片的地址即可。

这时，如果你的电脑连接着网络，那么图片会自动下载并显示出来，效果如下。

但是有时候从网上下载图片是很慢的，所有我们希望在没有下载完图片时，图片区域可以有些提示，所以我们利用 Image 的 status 属性，在 Edit 代码编辑界面，更改 Image 段代码如下：

```
Image {
```

```

id: image1
Text{id:text1} //用于显示信息
width: 200
height: 200
fillMode: "Tile"
source: "http://j.imagehost.org/0317/linux.jpg"
states: [
    State { //没有下载完图片时的状态
        name: "loading"
        when: image1.status != Image.Ready
        PropertyChanges {
            target: text1
            text:"loading..."
        }
    }
]
}

```

这时运行程序效果如下：

## 二、BorderImage 边界图片

边界图片，顾名思义，就是将一张图片作为窗口的边界。它主要的特点就是，在这里用四条线将一张图片分成了 9 部分。如下图。

这四条线分别用它们到图片各边界的像素值来表示，上下左右依次是 top, bottom, left 和 right。比如 top = 50 就是说离图片上边界 50 像素的地方就是上边界线。这样，将图片分为 9 个区域后，它们各自就有了不同的平铺规定。

下面我们先看例子，再进行总结。

1. 我们在库面板中选中 Border Image，拖入场景。

然后在属性面板中输入图片的路径，并设置其各边界线的值均为 30。如下图。

这时我们拉伸图片，查看效果。

可以看到，图片四个角没有变化，其他区域都被拉伸了。这时我们再将各边界线的值改为 60，然后查看效果。

可以看到，被边界线分到四个角的图片是不被拉伸的。

2. 平铺方式。

在这里我们可以分别指定水平方向的平铺方式（horizontalTileMode 属性）和竖直方向的平铺方式（verticalTileMode 属性）。平铺方式有以下三种：

BorderImage.Stretch：缩放图像以适合拉伸。（默认值）

BorderImage.Repeat：平铺图像，当空间不够时，最后一个图像可能会被裁剪。

BorderImage.Round：平铺图像，但是缩放所有平铺的图像，确保最后一个图像不会被裁剪。

我们上面就是使用的 Stretch 方式。下面我们改用其他方式，查看效果。

我们先更改代码如下：

```

BorderImage {
    id: borderimage1
    horizontalTileMode:BorderImage.Repeat
    verticalTileMode:BorderImage.Repeat
    border.bottom:30;border.top:30;border.right:30;border.left:30
    anchors.fill:parent
    source: "colors.png"
}

```

这时运行程序，效果如下：

我们更改平铺方式：

```
horizontalTileMode:BorderImage.Round  
verticalTileMode:BorderImage.Round
```

再次运行程序，效果如下：

我们再次更改平铺方式：

```
horizontalTileMode:BorderImage.Round  
verticalTileMode:BorderImage.Repeat
```

效果如下：

### 3. 结论

使用边界图片，我们需要先指定四条边界线，然后指定水平和竖直方向的平铺方式。

图片被四条边界线分为 9 个区域：

区域 5 会通过 horizontalTileMode 和 verticalTileMode 进行平铺。

区域 2,8 会通过 horizontalTileMode 进行平铺。

区域 4,6 会通过 verticalTileMode 进行平铺。

区域 1,3,7,9 的图像不会变化。

这一节中我们讲述了图片的使用。可以看到，在 QML 中图片是很容易操作的，我们可以按照我们自己的想法来显示图片。

体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 27th, 2010. 1,811 views

Tags: [BorderImage](#), [Image](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt4.7](#)

# 三十七、Flipable、Flickable 和状态与动画

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在前面的例子中我们已经多次提到过状态 State 了，在这一节中我们再次讲解一下 QML 中状态和动画的知识，然后讲解两个特效：Flipable 翻转效果和 Flickable 弹动效果。

我们先新建一个 Qt QML Application 工程，命名为“myAnimation”。

## 一、状态与动画

在 QML 中提供了多个实用的动画元素。其列表如下。

下面我们进行简单的讲解。

### 1. PropertyAnimation 属性动画。

列表中的 NumberAnimation 数值动画，ColorAnimation 颜色动画和 RotationAnimation 旋转动画都继承自 PropertyAnimation。

例如将程序代码更改如下：

```
import Qt 4.6
Rectangle {
    width: 300; height: 200
    Rectangle{
        id:page; width:50; height:50
        x:0; y:100; color:"red"
        PropertyAnimation on x{ to:100; duration:1000 }
    }
}
```

其中的属性动画的代码可以用数值动画来代替：

```
NumberAnimation on x{ to:100; duration:1000}
```

顾名思义，数值动画，就是只能对类型为 real 的属性进行动画设置。例如上面对 x 属性，使其在 1000ms 即一秒的时间里由以前的 0 变为 100。效果如下：

我们再将属性动画改为：

```
PropertyAnimation on color{ to:"blue"; duration:1000}
```

它可以用颜色动画来代替，相当于：

```
ColorAnimation on color{ to:"blue"; duration:1000}
```

颜色动画只能用于类型是 color 的属性。效果如下：

### 2. 缓冲曲线

我们很多时候不想让动画只是线性的变化，例如实现一些皮球落地，刹车等特殊动画效果，我们就可以在动画中使用缓冲曲线。

例如：

```
NumberAnimation on x{ to:100; duration:1000
    easing.type: "InOutElastic"}
```

这里的曲线类型有很多种，我们可以查看 QML PropertyAnimation Element Reference 关键字，在这个帮助文件中列出了所有的曲线类型。

### 3. 状态过渡动画

我们将程序代码更改如下：

```
Rectangle {
    width: 300; height: 200
    Rectangle{
        id:page; width:50; height:50
        x:0; y:100; color:"red"
        MouseArea{id:mouseArea; anchors.fill:parent}
        states: State {name: "state1"
            when:mouseArea.pressed
            PropertyChanges {target: page
                x:100;color:"blue"
            }
        }
    }
}
```

```

}
transitions: Transition {
from: ""; to: "state1"
NumberAnimation{property:"x";duration:500}
ColorAnimation{duration:500}
}
}
}

```

这里我们设置了一个新的状态“state1”，当鼠标在小矩形上按下时进入该状态。这种状态之间的改变我们前面已经讲过。但是我们想让两个状态之间进行变化时成为连续的，具有动画效果，那么就要使用上面的 transitions 状态过渡。

可以看到，在 transitions 中我们使用了 Transition{} 元素，然后从 “” 到 “state1” 即从默认状态进入 “state1” 状态。下面我们分别使用了数值动画和颜色动画，这样当从默认状态过渡到新建状态时，就会变为连续的动画。注意：这里颜色动画元素中省去了指定颜色属性。

运行程序，效果如下：

#### 4.并行动画

在上面的例子里我们看到，数值动画和颜色动画是并行执行的，其实我们也可以明确指出，让它们并行执行。那就是 ParallelAnimation 并行动画。上面的代码可以更改为：

```

ParallelAnimation{
NumberAnimation{property:"x";duration:500}
ColorAnimation{duration:500}
}

```

#### 5.序列动画

与上面的并行动画相对应的是序列动画 SequentialAnimation，使用它我们可以使两个动画按顺序执行，也就是一个执行完了，另一个才执行。

```

SequentialAnimation{
NumberAnimation{property:"x";duration:500}
ColorAnimation{duration:500}
}

```

这样当红色方块完成移动后再变为蓝色。

#### 6.属性默认动画。

有时我们不希望设置固定的状态，而是想当一个属性改变时，它就能执行默认的动画。那么我们就可以使用 Behavior{} 元素。我们将程序代码更改如下：

```

Rectangle {
width: 300; height: 200
Rectangle{
id:page; width:50; height:50; y:100; color:"red"
x:mouseArea.pressed?100:0;
MouseArea{id:mouseArea; anchors.fill:parent}
Behavior on x { NumberAnimation{ duration:500 } }
}
}

```

这里使鼠标按下时属性 x 为 100，并设置了 x 的默认动画，这样只要 x 发生了变化，它就会执行默认的动画。

#### 7.其他动画元素。

在开始的动画元素列表中还有其他一些动画元素没有讲到，我们会在后面的应用中使用到它们。你也可以先查看一下它们的帮助文档。

#### 二、Flipable 翻转效果

在 QML 中提供了一种可以将图片翻转的特效 Flipable，它具有强烈的 3D 视觉效果。

我们更改代码如下：

```

Rectangle{
width:300; height:250
Flipable{
    id:flipable; property int angle : 0 // 翻转角度
    width:back.width; height:back.height
}
}

```

```

property bool flipped : false // 用 来 标 志 是 否 翻 转
front: Image {source:"front.png"} // 指 定 前 面 的 图 片
back: Image {source:"back.png"} // 指 定 后 面 的 图 片
    transform:Rotation{           // 指 定 原 点
        origin.x:flipable.width/2; origin.y:flipable.height/2
        axis.x:0;      axis.y:1;    axis.z:0 // 指 定 按 y 轴 旋 转
                                            angle:flipable.angle
}
states:State{
    name:"back"          // 背 面 的 状 态
    PropertyChanges       {target:flipable;
                           angle:180}
                           when:flipable.flipped
}
transitions: Transition {
    NumberAnimation{property:"angle";duration:1000}
}
MouseArea{
    anchors.fill:parent
    onClicked:flipable.flipped
}
// 当 鼠 标 按 下 时 翻 转
}

```

运行效果如下：

我们可以看到，使用 Flipable 时，我们需要设置其前面和后面的图片，并设置背面的状态，然后设置旋转，并为状态改变设置动画就可以了。

通过改变转轴和角度，我们可以使用 Flipable 设计出很多其他特效。

### 三、Flickable 弹动效果

所谓 Flickable 效果就是你可以拖动它，它会根据你鼠标拖动的速度不同而移动不同的距离，并且这个移动好像有惯性一样，就像你推一下平面上的玩具汽车一样。我们看一个例子。

将程序代码更改如下：

```

Rectangle{
width:200; height:200
Flickable{
width:200;height:200
Image{id: picture; source:"01.jpg"}
contentWidth:picture.width
contentHeight:picture.height
}
}

```

这时运行程序，我们拖动整个图片，更改拖动的速度，这种感觉很爽！

我们拖动图片的角落，它会自动弹回去。

对于这样一个较大的图片，我们可以使用 Flickable 效果来查看整张图片。其实到底是否可以移动整个图片，取决于 contentWidth 和 contentHeight 的大小。

我们如果将代码改为：contentWidth:100;contentHeight:100

那么图片就无法通过拖动显示全部内容了。

对于 Flickable 效果我们在下一节中还会继续接触到，到时候我们可以看一下它更强的功能。

这一节中我们讲述了动画效果和两个特效，其实这一节的内容就是整个 QML 的核心内容。因为 QML 设计的 Declarative 界面主要内容就是其动画效果。我们也看到了，其实像翻转效果和弹动效果等都是为手机的触屏而设计的，所以要感受到它的真实效果，最好能在一台触屏手机上测试程序。

**体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !**

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 28th, 2010. 1,877 views

Tags: [creator](#), [Flickable](#), [Flipable](#), [QML](#), [qt](#), [Qt 4.7.0](#), [Qt Creator 2.0](#), [yafeilinux](#), [状态与动画](#)



## 三十八、QML 视图

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在 QML 中提供了三种视图方式：ListView 列表视图、GridView 网格视图和 PathView 路径视图。这三种视图都是继承自 Flickable，所以它们都有 Flickable 效果。下面我们简单介绍一下 ListView 和 PathView。

### 一、ListView 列表视图。

如果你了解 Qt 的模型视图结构，那么这一节的内容就很好理解了，如果你没接触过，也没关系，因为它其实很简单。在 Qt 中我们要想利用视图显示一些数据，并不是将这些数据直接放到视图中的，因为视图只管显示，它不存储数据。我们的数据要放在数据模型中。但是数据模型中只是存放数据，它并不涉及数据的显示方式。所以，我们还要用一个叫做代理的东东来设置数据模型中的数据怎样在视图中显示。那么就构成了下面的关系。

我们先看下面的例子：

1. 新建一个 Qt QML Application 工程，命名为“myView”。

2. 我们更改代码如下：

```
import Qt 4.6
Rectangle {
    width:200; height:200
    ListModel{ //数据模型
        id:listModel
        ListElement{name:"Tom";number:"001"}
        ListElement{name:"John";number:"002"}
        ListElement{name:"Sum";number:"003"}
    }
    Component{ //代理
        id:delegate
        Item{ id:wrapper; width:200; height:40
            Column{
                x:5; y:5
                Text{text:<b>Name:</b>" + name}
                Text{text:<b>Number:</b>" + number}
            }
        }
    }
    Component{ //高亮条
        id:highlight
        Rectangle{color:"lightsteelblue";radius:5}
    }
    ListView{ //视图
        width:parent.width; height:parent.height
        model:listModel //关联数据模型
        delegate:delegate //关联代理
        highlight:highlight //关联高亮条
        focus:true //可以获得焦点，这样就可以响应键盘了
    }
}
```

运行效果如下：

我们可以拖动整个列表，而且可以使用键盘的方向键来选择列表中的项目。

在这个程序中，我们先设置了数据模型，在其中加入了一些数据。然后设置了代理，在代理中我们设置了要怎样显示我们的数据。最后，我们在视图中关联了数据模型和代理，将数据显示出来了。这里为了达到更好的显示效果，我们使用了一个高亮条。其中的代理和高亮条都可以使用 Component{} 组件来实现。

3. 我们可以对视图做一些设置。

我们可以设置 keyNavigationWraps:true 使到达最后一个项目后重新返回第一个项目。

我们可以设置 orientation:ListView.Horizontal 使列表水平显示。这时你拖动列表，发现了吧，它可以自动移动到下一条，这就是 Flickable 的作用。默认的是 ListView.Vertical 竖直显示。

## 二、PathView 路径视图

1.什么是路径视图，我们先来看一个例子。

```
Rectangle {  
    width:300; height:300;  
    ListModel{ //数据模型  
        id:listModel  
        ListElement{icon:"01.gif"}  
        ListElement{icon:"02.gif"}  
        ListElement{icon:"03.gif"}  
        ListElement{icon:"04.gif"}  
    }  
    Component{ //代理  
        id:delegate  
        Item{ id:wrapper; width:50; height:50  
            Column{  
                Image {source:icon; width:50; height:50}  
            }  
        }  
    }  
}  
PathView{ //路径视图  
    anchors.fill:parent; model:listModel; delegate:delegate  
    path:Path{startX:120; startY:200  
        PathQuad{x:120; y:25; controlX:260; controlY:125}  
        PathQuad{x:120; y:200; controlX:-20; controlY:125}  
    }  
}
```

效果如下：

你可以拖动一个图标查看效果，是的，所有图标的转起来了。这就是路径视图。我们在程序中，设置了一个路径，如上面的：

```
path:Path{startX:120; startY:200  
    PathQuad{x:120; y:25; controlX:260; controlY:125}  
    PathQuad{x:120; y:200; controlX:-20; controlY:125}  
}
```

它们确定了一个椭圆形，所有的项目都在这个路径上，当拖动一个项目，所有的项目都会在路径上移动。

### 2.关于 Path

在 QML 中提供了三种 Path。PathLine 直线，PathQuad 二次贝塞尔曲线，PathCubic 三次贝塞尔曲线。你可以在帮助中查看它们的使用，这里不再进行过多介绍。

### 3.路径属性。

我们可以通过路径属性 PathAttribute，来设置不同路径上不同位置的项目。

例如我们更改上面的程序：

```
Component{ //代理  
    id:delegate  
    Item{ id:wrapper; width:50; height:50  
        scale:PathView.scale; opacity:PathView.opacity  
        Column{  
            Image {source:icon; width:50; height:50}  
        }  
    }  
}  
PathView{  
    anchors.fill:parent; model:listModel; delegate:delegate  
    path:Path{startX:120; startY:200  
        PathAttribute{name:"scale"; value:1.0}  
        PathAttribute{name:"opacity"; value:1.0}  
    }  
}
```

```
PathQuad{x:120;y:25;controlX:260;controlY:125}
PathAttribute{name:"scale";value:0.5}
PathAttribute{name:"opacity";value:0.5}
PathQuad{x:120;y:200;controlX:-20;controlY:125}
}
}
}
```

效果如下：

我们在 Path 中设置了路径属性，使得在不同点的图片具有不同的效果，这里设置了缩放和不透明度两个属性。我们只需设置开始点和结束点两个点的属性，这样就会在整个路径上进行线性插值。

这一节介绍了两个视图，还有一个 GridView 网格视图，它的操作是相似的，在这里就不再进行介绍了。

**体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !**

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 28th, 2010. 2,055 views

Tags: [declarative](#), [GridView](#), [ListView](#), [PathView](#), [QML](#), [Qt Creator 2.0](#), [Qt4.7.0](#), [yafeilinux](#)

## 三十九、QtDeclarative 模块

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在一开始我们就提到过 Qt Quick 由三部分组成，前面我们已经讲了 Qt Quick Designer 和 QML，这一节我们讲述 QtDeclarative 模块。

我们在帮助中查找 QtDeclarative Module 关键字。这里列出了该模块的所有相关类。应该说明这个模块中的类是 Qt 新加的标准 C++ 类，不是 QML 元素。

这个模块的作用就是将 QML 元素与以前的标准 C++ 类相结合。而且它提供了一个很简单的方法使新建的 QML 文件作为一个项目加入到以前的图形视图的应用中。下面我们来看一个简单的例子。

1. 我们新建 Empty Qt Project，工程名为“myDeclarative”。然后添加一个 C++ Source File，命名为 main.cpp。

2. 我们在工程文件 myDeclarative.pro 中添加一行代码： QT += declarative 表明使用了 QtDeclarative 模块。

3. 我们更改 main.cpp 的内容如下。

```
#include <QtDeclarative/QDeclarativeView>
#include <QtDeclarative/QDeclarativeItem>
#include <QtDeclarative/QDeclarativeEngine>
#include <QtDeclarative/QDeclarativeComponent>
#include <QtGui>
int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QDeclarativeEngine engine;
    QDeclarativeComponent component(&engine,QUrl("main.qml"));
    QDeclarativeItem *item = qobject_cast<QDeclarativeItem *>(component.create());
    QDeclarativeView *view = new QDeclarativeView;
    QGraphicsScene *scene = new QGraphicsScene;
    view->setScene(scene);
    scene->addItem(item);
    view->show();
    return app.exec();
}
```

4. 我们新建 Qt QML File，文件名为“main.qml”。其内容如下。

```
import Qt 4.6
Rectangle {
    width: 200
    height: 200
    color:"green"
    Rectangle {
        id: rectangle1
        x: 50
        y: 50
        width: 100
        height: 100
        color: "blue"
        radius: 30
    }
}
```

5. 然后我们运行程序，这时可能发现窗口中是空白的，而输出窗口中会出现下面的提示：

```
QDeclarativeComponent: Component is not ready
QGraphicsScene::addItem: cannot add null item
```

这是因为，在 Qt Creator 2.0 中，将目标文件与源代码分开了。也就是说，我们的源代码放在了一个文件夹中（如这里的 myDeclarative 文件夹），编译生成的文件放在了另一个文件夹中（这里是 myDeclarative-build-desktop 文件夹），你可以到磁盘中查看一下。而程序中 QUrl("main.qml") 表

明在本文件夹中的 main.qml 文件，就是说 main.qml 要在目标文件夹下，而现在它却在源代码文件夹下，所以程序运行时没有找到该文件。

6. 我们可以将 main.qml 文件复制到 myDeclarative-build-desktop 文件夹中，但是这样每次在 Qt Creator 中改动该文件都要复制一次，很麻烦。所以我们也可以直接改程序中的路径，将 QUrl("main.qml") 改为 QUrl("../myDeclarative/main.qml") 就可以了。当然，最好的方法是，新建一个资源文件，将 main.qml 文件加入资源文件中。

7. 运行程序，效果如下：

8. 下面我们来讲解一下 main.cpp 文件中的内容。

首先是头文件包含，这里一定要明确指定二级目录，如

```
#include <QtDeclarative/QDeclarativeView>  
而不能像其他 Qt 头文件一样只写 #include <QtDeclarative>
```

因为现在的测试版中，这样会出错误。

然后是后面的：

```
QDeclarativeEngine engine;
```

```
QDeclarativeComponent component(&engine,QUrl("main.qml"));
```

```
QDeclarativeItem *item = qobject_cast<QDeclarativeItem *>(component.create());
```

这里的 QDeclarativeEngine 用来对下面的 QML 组件进行环境配置，而 QML 组件 QDeclarativeComponent 用来代表一个 QML 文件。再往下，QDeclarativeItem 就是相当于一个 QGraphicsItem，我们使用类型转换将 QML 组件转换为一个项目。

再往下：

```
QDeclarativeView *view = new QDeclarativeView;
```

```
QGraphicsScene *scene = new QGraphicsScene;
```

```
view->setScene(scene);
```

```
scene->addItem(item);
```

```
view->show();
```

我们新建了一个视图，然后新建一个场景，将项目加入场景中，然后在视图中显示出来，这与以前的图形视图编程是一样的。需要说明的是，这里使用了 QDeclarativeView 类，它是 QtDeclarative 模块的组成部分，拥有 QGraphicsView 的所有功能。当然如果这里改为 QGraphicsView 也是可以的。

9. 可以看到，要将一个 QML 文件作为一个项目用于图形视图编程是很简单的。我们只需要四步：

第一，新建 QDeclarativeEngine

第二，新建一个 QDeclarativeComponent

第三，生成 QDeclarativeItem

第四，将 QDeclarativeItem 作为一个项目用于图形视图中。

其实 QtDeclarative 模块还有很多功能，我们在这里就不再过多介绍了。有兴趣的朋友可以先自己查看帮助文件。

到这里 Qt Quick 部分的教程也就告一段落了。其实我们只是讲解了 Qt Quick 很小的一部分知识，要想感受 Qt Quick 的强大，还是需要靠大量的编程去体会的。

**体验全新的 Qt 4.7.0，更多精彩内容，尽在 [www.yafeilinux.com](http://www.yafeilinux.com) !**

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 29th, 2010. 5,351 views

Tags: [declarative](#), [QML](#), [Qt 4.7.0](#), [Qt Creator 2.0](#), [Qt Quick](#), [Qt4.7](#), [Qt4.7.0](#), [QtDeclarative](#)

## 四十、使用 Nokia Qt SDK 开发 Symbian 和 Maemo 终端软件

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

Nokia Qt SDK Beta 已经发布了，利用它可以很容易地开发 Symbian 和 Maemo 终端软件，将来也会支持 MeeGo 平台。它与我们以前用的 Qt 开发环境的区别是，它的目标平台是移动平台，如 Symbian，也就是说利用它编译生成的可执行文件，只能在模拟器或 Symbian 手机上运行，不能直接在电脑上运行。而以前我们的开发目标是桌面系统，如 Windows。当然这个 SDK 中的 Qt Creator 就是我们以前应用的 Qt Creator，我们也可以很容易地让它用于桌面软件的开发。

Nokia Qt SDK 开发 Symbian 程序教程：[教程](#)

几天前诺基亚 Qt 举行了全球的诺基亚 Qt SDK Beta 概览网上直播。下面是相关内容。

诺基亚 Qt SDK Beta 概览 (中文网上直播)：

诺基亚 Qt SDK 现已推出 Beta 版，供开发伙伴使用并搜集反馈意见。该 SDK 支持最新版的 Qt4.6 库，包括面向移动开发的 Qt APIs。SDK 提供了同时面向 Symbian 和 Maemo 终端的统一开发伙伴工具，其中包括 Qt Creator IDE、一个模拟器，也包括真机调试，和一站式安装包。

使用诺基亚 Qt SDK，开发伙伴们开展工作将更快更好。能在不同平台间使用同样代码实现更佳过渡，诺基亚 Qt SDK 向大家提供了面向多种终端的最有效方法。

这个网络直播讲解了 SDK 的功能及其安装设置，也演示了应用的编码和测试。

查看直播的视频：[诺基亚 Qt SDK Beta 概览视频](#)

查看相关 PDF 文件：[诺基亚 Qt SDK Beta 概览文档](#)

因为上面的视频和教程中对 Nokia Qt SDK 的使用已经讲得很详细了，所以我们这里就不再讲述这个过程了。下面简单说明一下，怎样在 Qt Creator 中使用以前的 Qt 库开发桌面程序。

我们在 Options 菜单中查看 Qt Versions，如果我们以前已经安装了 Qt 的其他版本，并且已经加入了系统 PATH 环境变量中，那么这里会显示出来。如果没有在 PATH 中添加，那么我们需要添加一下，或者直接在这里的 Manual 中添加，详细设置请参考[三十一、Qt 4.7.0 及 Qt Creator 2.0 beta 版安装全程图解](#)一文。

这里设置好以后，我们只需要在建立工程时选择 Desktop ( 表明是桌面程序 ) 中的 Qt in PATH 即可。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：五月 29th, 2010. 2,720 views  
Tags: [Maemo](#), [Nokia Qt SDK](#), [Qt SDK Beta 概览网上直播](#), [Symbian](#)

## 四十一、Qt 网络（一）简介

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

提示：以后我们使用的编程环境为 Windows 下基于 Qt 4.6.3 的 Qt Creator 1.3.1。

从这一节开始我们讲述 Qt 网络应用方面的编程知识。在开始这部分知识的学习之前，你最好已经拥有了一定的网络知识和 Qt 的编程基础。在下面的教程中我们不会对一个常用的网络名词去进行详细解释，对于不太了解的地方，你可以参考相关书籍。不过，你也没有必要非得先去学习网络教材，而后再学习本部分内容，因为 Qt 提供了简单明了的接口函数，使得我们这里并没有涉及太多专业的知识。看完教程后，你也许会发现，自己虽然不懂网络，但却可以编写网络应用程序了。

下面我们打开 Qt Creator，在 Help 页面中我们搜索 QtNetwork Module 关键字，其内容如下图。

在 Qt 中提供了网络模块 ( QtNetwork Module ) 来用于网络程序的开发，可以看到，在这里提供了多个相关类。有用于 FTP 编程的 QFtp 类，用于 HTTP 编程的 QNetworkAccessManager 类和 QNetworkReply 类，用于获得本机信息的 QHostInfo 类，用于 Tcp 编程的 QTcpServer 类和 QTcpSocket 类，用于 UDP 编程的 QUdpSocket 类，用于网络加密的 QSslSocket 类，用于网络代理的 QNetworkProxy 类等等。

如果你以前就使用过 Qt 进行网络部分编程，或者看过其他教材上相关内容，你可能会问，这里怎么没有了 QHttp 类。我们现在搜索 QHttp 关键字，其内容如下。

可以看到这里有一个警告：

This class is obsolete. It is provided to keep old source code working. We strongly advise against using it in new code.

大概意思是：这个类是过时的。它的提供只是为了保证旧的源代码。我们强烈建议在新代码中不要使用它。所以在我们的教程中不会再讲解这个类，对于 HTTP 部分的编程，我们使用 QNetworkAccessManager 类和 QNetworkReply 类。

最后需要说明的是：使用这个模块我们需要在工程文件中添加 QT += network，然后使用时添加 #include <QtNetwork> 头文件。

对于网络部分相关的例子，我们可以查看其演示程序。在 Windows 的开始菜单中选择 Qt Creator 的安装目录，然后选择 Qt Demo 菜单。我们可以在 Networking 菜单中找到网络部分的例子。如下图。

我们可以运行这些例子查看效果，也可以查看它们的帮助文件，如下图，点击 Documentation 即可。

当我们对 Qt 中的网络编程有了一定了解之后，我们就可以开始下一步的学习了。

分类：[Qt 系列教程](#) 作者：[yafeilinux](#) 日期：六月 16th, 2010. 2,882 views

Tags: [QtNetwork Module](#), [Qt 网络](#), [Qt 网络教程](#), [yafeilinux](#), [网络教程](#)

## 四十二、Qt 网络（二）HTTP 编程

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

HTTP 即超文本传输协议，它是一种文件传输协议。这一节中我们将讲解如何利用 HTTP 从网站上下载文件。

上一节中我们已经提到过了，现在 Qt 中使用 QNetworkAccessManager 类和 QNetworkReply 类来进行 HTTP 的编程。下面我们先看一个简单的例子，然后再进行扩展。

### （一）最简单的实现。

1. 我们新建 Qt4 Gui QApplication。

工程名为“http”，然后选中 QtNetwork 模块，最后 Base class 选择 QWidget。注意：如果新建工程时没有添加 QtNetwork 模块，那么就要手动在工程文件.pro 中添加代码 QT += network，表明我们使用了网络模块。

2. 我们在 widget.ui 文件中添加一个 Text Browser，如下图。

3.. 在 widget.h 中我们添加代码。

添加头文件：#include <QtNetwork>

私有变量 private 中：QNetworkAccessManager \*manager;

私有槽函数 private slots 中：void replyFinished(QNetworkReply \*);

4. 在 widget.cpp 文件中添加代码。

在构造函数中添加如下代码：

```
manager = new QNetworkAccessManager(this); //新建 QNetworkAccessManager 对象
connect(manager,SIGNAL(finished(QNetworkReply*)), //关联信号和槽
        this,SLOT(replyFinished(QNetworkReply*)));
manager->get(QNetworkRequest(QUrl("http://www.yafeilinux.com"))); //发送请求
```

然后定义函数：

```
void Widget::replyFinished(QNetworkReply *reply) //当回复结束后
{
```

```
    QTextCodec *codec = QTextCodec::codecForName("utf8");
    //使用 utf8 编码，这样才可以显示中文
    QString all = codec->toUnicode(reply->readAll());
    ui->textBrowser->setText(all);
    reply->deleteLater(); //最后要释放 reply 对象
}
```

5. 运行效果如下。

6. 代码分析。

上面实现了最简单的应用 HTTP 协议下载网页的程序。QNetworkAccessManager 类用于发送网络请求和接受回复，具体的，它是用 QNetworkRequest 类来管理请求，QNetworkReply 类进行接收入复，并对数据进行处理。

在上面的代码中，我们使用了下面的代码来发送请求：

```
manager->get(QNetworkRequest(QUrl("http://www.yafeilinux.com")));
```

它返回一个 QNetworkReply 对象，这个下面再讲。我们只需知道只要发送请求成功，它就会下载数据。

而当数据下载完成后，manager 会发出 finished() 信号，我们对它进行了关联：

```
connect(manager,SIGNAL(finished(QNetworkReply*)),
        this,SLOT(replyFinished(QNetworkReply*)));
```

也就是说，当下载数据结束时，就会执行 replyFinished() 函数。在这个函数中我们对接收的数据进行处理：

```
QTextCodec *codec = QTextCodec::codecForName("utf8");
QString all = codec->toUnicode(reply->readAll());
ui->textBrowser->setText(all);
```

这里，为了能显示下载的网页中的中文，我们使用了 QTextCodec 类对象，应用 utf8 编码。

使用 reply->readAll() 函数就可以将下载的所有数据读出。然后，我们在 textBrowser 中将数据显示出来。当 reply 对象已经完成了它的功能时，我们需要将它释放，就是最后一条代码：

```
reply->deleteLater();
```

### （二）功能扩展

通过上面的例子可以看到，Qt 中编写基于 HTTP 协议的程序是十分简单的，只有十几行代码。不过，一般我们下载文件都想要看到下载进度。下面我们就更改上面的程序，让它可以下载任意的文件，并且显示下载进度。

1. 我们更改 widget.ui 文件如下图。

这里我们添加了一个 Line Edit，一个 Label，一个 Progress Bar 和一个 Push Button，它们的熟悉保持默认即可。我们在 Push Button 上点击鼠标右键，选择 Go to slot，然后选择 clicked()，进入其单击事件槽函数，现在我们先不写代码。

在写代码之前，我们先介绍一下整个程序执行的流程：

开始我们先让进度条隐藏。当我们在 Line Edit 中输入下载地址，点击下载按钮后，我们应用输入的下载地址，获得文件名，在磁盘上新建一个文件，用于保存下载的数据，然后进行链接，并显示进度条。在下载过程中，我们将每次获得的数据都写入文件中，并更新进度条，在接收完文件后，我们重新隐藏进度条，并做一些清理工作。

根据这个思路，我们开始代码的编写。

2. 我们在 widget.h 文件中添加代码，完成后其部分内容如下。

```
class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();
    void startRequest(QUrl url); //请求链接
protected:
    void changeEvent(QEvent *e);
private:
    Ui::Widget *ui;
    QNetworkAccessManager *manager;
    QNetworkReply *reply;
    QUrl url; //存储网络地址
    QFile *file; //文件指针
private slots:
    void on_pushButton_clicked(); //下载按钮的单击事件槽函数
    void httpFinished(); //完成下载后的处理
    void httpReadyRead(); //接收到数据时的处理
    void updateDataReadProgress(qint64,qint64); //更新进度条
};
```

3. widget.cpp 文件中的相关内容如下。

(1) 构造函数中：

```
manager = new QNetworkAccessManager(this);
ui->progressBar->hide();
```

我们在构造函数中先隐藏进度条。等开始下载时再显示它。

(2) 下载按钮的单击事件槽函数。

```
void Widget::on_pushButton_clicked() //下载按钮
{
    url = ui->lineEdit->text();
    //获取在界面中输入的 url 地址，如 http://zz.onlinedown.net/down/laolafangkuaijin.rar
    QFileInfo info(url.path());
    QString fileName(info.fileName());
    //获取文件名
    if (fileName.isEmpty()) fileName = "index.html";
    //如果文件名为空，则使用“index.html”，
    //例如使用“http://www.yafeilinux.com”时，文件名就为空
    file = new QFile(fileName);
    if(!file->open(QIODevice::WriteOnly))
    { //如果打开文件失败，则删除 file，并使 file 指针为 0，然后返回
```

```

    qDebug() << "file open error";
    delete file;
    file = 0;
    return;
}
startRequest(url); //进行链接请求
ui->progressBar->setValue(0); //进度条的值设为 0
ui->progressBar->show(); //显示进度条
}

```

这里我们先从界面中获取输入的地址，然后分解出文件名。因为地址中可能没有文件名，这时我们就使用一个默认的文件名。然后我们用这个文件名新建一个文件，这个文件会保存到工程文件夹的 debug 文件夹下。下面我们打开文件，然后进行链接，并显示进度条。

#### ( 3 ) 链接请求函数。

```

void Widget::startRequest(QUrl url) //链接请求
{
    reply = manager->get(QNetworkRequest(url));
    //下面关联信号和槽
    connect(reply,SIGNAL(finished()),this,SLOT(httpFinished()));
    //下载完成后
    connect(reply,SIGNAL(readyRead()),this,SLOT(httpReadyRead()));
    //有可用数据时
    connect(reply,SIGNAL(downloadProgress(qint64,qint64)),
            this,SLOT(updateDataReadProgress(qint64,qint64)));
    //更新进度条
}

```

在上一个例子中我们就提到了 manager->get(QNetworkRequest(url))，返回的是一个 QNetworkReply 对象，这里我们获得这个对象，使用它完成显示数据下载进度的功能。这里主要是关联了几个信号和槽。当有可用数据时，reply 就会发出 readyRead() 信号，我们这时就可以将可用的数据保存下来。就是在这里，实现了数据分段下载保存，这样比下载完所有数据再保存，要节省很多内存。而利用 reply 的 downloadProgress() 信号，很容易就实现了进度条的显示。

#### ( 4 ) 保存数据函数。

```

void Widget::httpReadyRead() //有可用数据
{
    if (file) file->write(reply->readAll()); //如果文件存在，则写入文件
}

```

这里当 file 可用时，将下载的数据写入文件。

#### ( 5 ) 更新进度条函数。

```

void Widget::updateDataReadProgress(qint64 bytesRead, qint64 totalBytes)
{
    ui->progressBar->setMaximum(totalBytes); //最大值
    ui->progressBar->setValue(bytesRead); //当前值
}

```

每当有数据到来时，都更新进度条。

#### ( 6 ) 完成下载。

```

void Widget::httpFinished() //完成下载
{
    ui->progressBar->hide();
    file->flush();
    file->close();
    reply->deleteLater();
    reply = 0;
    delete file;
}

```

```
    file = 0;  
}
```

这里只是当下载完成后，进行一些处理。

4. 我们运行程序，效果如下。

下载网页文件：

下载华军软件园上的劳拉方块游戏：

下载完成后可以看到工程文件夹中 debug 文件夹中的下载的文件。

我们 HTTP 应用的内容就讲到这里，可以看到它是很容易的，也不需要你了解太多的 HTTP 的原理知识。关于相关的类的其他使用，你可以查看其帮助。在上面的例子中，我们只是为了讲解知识，所以程序还很不完善，对于一个真正的工程，我们还需要注意更多的细节，你可以查看 Qt 演示程序 HTTP Client 的源代码。

下一节我们将讲述 FTP 的内容。

分 类：[Qt 系列教程](#) 作 者：yafeilinux 日 期：六月 17th, 2010. 4,414 views  
Tags: [creator](#), [http](#), [qt](#), [Qt http](#), [Qt 网络](#), [yafeilinux](#), [教程](#)

## 四十三、Qt 网络（三）FTP（一）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

上一节我们讲述了 HTTP 的编程，这一节讲述与其及其相似的 FTP 的编程。FTP 即 File Transfer Protocol，也就是文件传输协议。FTP 的主要作用，就是让用户连接上一个远程计算机，查看远程计算机有哪些文件，然后把文件从远程计算机上拷贝到本地计算机，或者把本地计算机的文件送到远程计算机上。

在 Qt 中，我们可以使用上一节讲述的 QNetworkAccessManager 和 QNetworkReply 类来进行 FTP 程序的编写，因为它们用起来很简单。但是，对于较复杂的 FTP 操作，Qt 还提供了 QFtp 类，利用这个类，我们很容易写出一个 FTP 客户端程序。下面我们先在帮助中查看这个类。

在 QFtp 中，所有的操作都对应一个特定的函数，我们可以称它们为命令。如 connectToHost()连接到服务器命令，login()登录命令，get()下载命令，mkdir()新建目录命令等。因为 QFtp 类以异步方式工作，所以所有的这些函数都不是阻塞函数。也就是说，如果一个操作不能立即执行，那么这个函数就会直接返回，直到程序控制权返回 Qt 事件循环后才真正执行，它们不会影响界面的显示。

所有的命令都返回一个 int 型的编号，使用这个编号让我们可以跟踪这个命令，查看其执行状态。当每条命令开始执行时，都会发出 commandStarted() 信号，当该命令执行结束时，会发出 commandFinished() 信号。我们可以利用这两个信号和命令的编号来获取命令的执行状态。当然，我们不想执行每条命令都要记下它的编号，所以我们也可以使用 currentCommand() 来获取现在执行的命令，其返回值与命令的对应关系如下图。

下面我们先看一个简单的 FTP 客户端的例子，然后对它进行扩展。

在这个例子中我们从 FTP 服务器上下载一个文件并显示出来。

1. 我们新建 Qt4 Gui QApplication。

工程名为“myFtp”，然后选中 QtNetwork 模块，最后 Base class 选择 QWidget。

2. 修改 widget.ui 文件。

在其中添加一个 Text Browser 和一个 Label，效果如下。

3. 在 main.cpp 中进行修改。

为了在程序中可以使用中文，我们在 main.cpp 中添加头文件 #include <QTextCodec>  
并在 main() 函数中添加代码： QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

4. 在 widget.h 中进行修改。

先添加头文件：#include <QFtp>

再在 private 中定义对象：QFtp \*ftp;

添加私有槽函数：

private slots:

    void ftpCommandStarted(int);

    void ftpCommandFinished(int,bool);

5. 在 widget.cpp 中进行更改。

(1) 在构造函数中添加代码：

ftp = new QFtp(this);

    ftp->connectToHost("ftp.qt.nokia.com"); //连接到服务器

    ftp->login(); //登录

    ftp->cd("qt"); //跳转到“qt”目录下

    ftp->get("INSTALL"); //下载“INSTALL”文件

    ftp->close(); //关闭连接

    connect(ftp,SIGNAL(commandStarted(int)),

                this,SLOT(ftpCommandStarted(int)));

    //当每条命令开始执行时发出相应的信号

    connect(ftp,SIGNAL(commandFinished(int,bool))),

                this,SLOT(ftpCommandFinished(int,bool))));

//当每条命令执行结束时发出相应的信号

我们在构造函数里执行了几个 FTP 的操作，登录站点，并下载了一个文件。然后我们又关联了两个信号和槽，用来跟踪命令的执行情况。

(2) 实现槽函数：

void Widget::ftpCommandStarted(int)

{

    if(ftp->currentCommand() == QFtp::ConnectToHost){

```

        ui->label->setText(tr("正在连接到服务器..."));
    }
    if (ftp->currentCommand() == QFtp::Login){
        ui->label->setText(tr("正在登录..."));
    }
    if (ftp->currentCommand() == QFtp::Get){
        ui->label->setText(tr("正在下载..."));
    }
    else if (ftp->currentCommand() == QFtp::Close){
        ui->label->setText(tr("正在关闭连接..."));
    }
}

```

每当命令执行时，都会执行 `ftpCommandStarted()` 函数，它有一个参数 `int id`，这个 `id` 就是调用命令时返回的 `id`，如 `int loginID = ftp->login();` 这时，我们就可以用 `if(id == loginID)` 来判断执行的是不是 `login()` 函数。但是，我们不想为每个命令都设置一个变量来存储其返回值，所以，我们这里使用了 `ftp->currentCommand()`，它也能获取当前执行的命令的类型。在这个函数里我们让开始不同的命令时显示不同的状态信息。

```

void Widget::ftpCommandFinished(int,bool error)
{
    if(ftp->currentCommand() == QFtp::ConnectToHost){
        if(error) ui->label->setText(tr("连接服务器出现错误: %1").arg(ftp->errorString()));
        else ui->label->setText(tr("连接到服务器成功"));
    }
    if (ftp->currentCommand() == QFtp::Login){
        if(error) ui->label->setText(tr("登录出现错误: %1").arg(ftp->errorString()));
        else ui->label->setText(tr("登录成功"));
    }
    if (ftp->currentCommand() == QFtp::Get){
        if(error) ui->label->setText(tr("下载出现错误: %1").arg(ftp->errorString()));
        else {
            ui->label->setText(tr("已经完成下载"));
            ui->textBrowser->setText(ftp->readAll());
        }
    }
    else if (ftp->currentCommand() == QFtp::Close){
        ui->label->setText(tr("已经关闭连接"));
    }
}

```

这个函数与 `ftpCommandStarted()` 函数相似，但是，它是在一个命令执行结束时执行的。它有两个参数，第一个 `int id`，就是调用命令时返回的编号，我们在上面已经讲过了。第二个是 `bool error`，它标志现在执行的命令是否出现了错误。如果出现了错误，那么 `error` 为 `true`，否则为 `false`。我们可以利用它来输出错误信息。在这个函数中，我们在完成一条命令时显示不同的状态信息，并显示可能的出错信息。在 `if (ftp->currentCommand() == QFtp::Get)` 中，也就是已经完成下载时，我们让 `textBrowser` 显示下载的信息。

6.运行程序，效果如下。

登录状态。

下载完成后。

7.出错演示。

下面我们演示一下出错时的情况。

将构造函数中的代码 `ftp->login();` 改为 `ftp->login("tom","123456");`  
这时我们再运行程序：

可以看到，它输出了错误信息，指明了错误的指令和出错的内容。其实我们设置的这个错误，也是想告诉大家，在FTP中如果没有设置用户名和密码，那么默认的用户名应该是anonymous，这时密码可以任意填写，而使用其他用户名是会出错的。

[在下一节中，我们将会对这个程序进行扩展，让它可以浏览服务器上的所有文件，并进行下载。](#)

分 类 : [Qt 系列教程](#) 作 者 : yafeilinux 日 期 : 六 月 22nd, 2010. 2,149 views  
Tags: [ftp](#), [QFtp](#), [Qt 网络](#)

## 四十四、Qt 网络（四）FTP（二）

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

前面讲述了一个最简单的 FTP 客户端程序的编写，这一节我们将这个程序进行扩展，使其可以浏览并能下载服务器上的所有文件。

1.更改 widget.ui 文件如下。

我们删除了 Text Browser，加入了几个 Label，Line Edit，Push Button 部件，一个 Tree Widget 及一个 Progress Bar 部件。然后我们对其中几个部件做如下更改。

(1) 将“FTP 服务器”标签后的 Line Edit 的 objectName 属性改为“ftpServerLineEdit”，其 text 属性改为“ftp.qt.nokia.com”。

(2) 将“用户名”标签后的 Line Edit 的 objectName 属性改为“userNameLineEdit”，其 text 属性改为“anonymous”，将其 toolTip 属性改为“默认用户名请使用： anonymous，此时密码任意。”

(3) 将“密码”标签后的 Line Edit 的 objectName 属性改为“passWordLineEdit”，其 text 属性改为“123456”，将其 echoMode 属性改为“Password”。

(4) 将“连接”按钮的 objectName 属性改为“connectButton”。

(5) 将“返回上一级目录”按钮的 objectName 属性改为“cdToParentButton”。

(6) 将“下载”按钮的 objectName 属性改为“downloadButton”。

(7) 将 Tree Widget 的 objectName 属性改为“fileList”，然后在 Tree Widget 部件上单击鼠标右键，选择 Edit Items 菜单，添加列属性如下。

最终的界面如下。

下面我们的程序中，就是实现在用户填写完相关信息后，按下“连接”按钮，就可以连接到 FTP 服务器，并在 Tree Widget 中显示服务器上的所有文件，我们可以按下“下载”按钮来下载选中的文件，并使用进度条显示下载进度。

2.更改 widget.h 文件。

(1) 添加头文件 #include <QtGui>

(2) 在 private 中添加变量：

QHash<QString, bool> isDirectory;

//用来存储一个路径是否为目录的信息

QString currentPath;

//用来存储现在的路径

(3) 添加槽函数：

private slots:

void on\_downloadButton\_clicked();

void on\_cdToParentButton\_clicked();

void on\_connectButton\_clicked();

void ftpCommandFinished(int,bool);

void ftpCommandStarted(int);

void updateDataTransferProgress(qint64,qint64 );

//更新进度条

void addToList(const QUrlInfo &urlInfo);

//将服务器上的文件添加到 Tree Widget 中

void processItem(QTreeWidgetItem\*,int);

//双击一个目录时显示其内容

3.更改 widget.cpp 的内容。

(1) 实现“连接”按钮的单击事件槽函数。

```
void Widget::on_connectButton_clicked() //连接按钮
{
```

```
    ui->fileList->clear();
```

```
    currentPath.clear();
```

```
    isDirectory.clear();
```

```
    ftp = new QFtp(this);
```

```
    connect(ftp,SIGNAL(commandStarted(int)),this,SLOT(ftpCommandStarted(int)));
```

```
    connect(ftp,SIGNAL(commandFinished(int,bool)),
```

```
    this,SLOT(ftpCommandFinished(int,bool))));
```

```

connect(ftp,SIGNAL(listInfo(QUrlInfo)),this,SLOT(addToList(QUrlInfo)));
connect(ftp,SIGNAL(dataTransferProgress(qint64,qint64)),
        this,SLOT(updateDataTransferProgress(qint64,qint64)));
QString ftpServer = ui->ftpServerLineEdit->text();
QString userName = ui->userNameLineEdit->text();
QString passWord = ui->passWordLineEdit->text();
ftp->connectToHost(ftpServer,21); //连接到服务器,默认端口号是 21
ftp->login(userName,passWord); //登录
}

```

我们在“连接”按钮的单击事件槽函数中新建了 ftp 对象，然后关联了相关的信号和槽。这里的 listInfo() 信号由 ftp->list() 函数发射，它将在登录命令完成时调用，下面我们提到。而 dataTransferProgress() 信号在数据传输时自动发射。最后我们从界面上获得服务器地址，用户名和密码等信息，并以它们为参数执行连接和登录命令。

### (2) 更改 ftpCommandFinished() 函数。

我们在相应位置做更改。

首先，在登录命令完成时，我们调用 list() 函数：

```

ui->label->setText(tr("登录成功"));
ftp->list(); //发射 listInfo() 信号，显示文件列表
然后，在下载命令完成时，我们使下载按钮可用：
ui->label->setText(tr("已经完成下载"));
ui->downloadButton->setEnabled(true);
最后再添加一个 if 语句，处理 list 命令完成时的情况：
if (ftp->currentCommand() == QFtp::List){
    if (isDirectory.isEmpty())
        { //如果目录为空,显示“empty”
            ui->fileList->addTopLevelItem(
                new QTreeWidgetItem(QStringList() << tr("<empty>")));
            ui->fileList->setEnabled(false);
            ui->label->setText(tr("该目录为空"));
        }
}

```

我们在 list 命令完成时，判断文件列表是否为空，如果为空，就让 Tree Widget 不可用，并显示“empty”条目。

### (3) 添加文件列表函数的内容如下。

```

void Widget::addToList(const QUrlInfo &urlInfo) //添加文件列表
{
    QTreeWidgetItem *item = new QTreeWidgetItem;
    item->setText(0, urlInfo.name());
    item->setText(1, QString::number(urlInfo.size()));
    item->setText(2, urlInfo.owner());
    item->setText(3, urlInfo.group());
    item->setText(4, urlInfo.lastModified().toString("MMM dd yyyy"));
    QPixmap pixmap(urlInfo.isDir() ? "../dir.png" : "../file.png");
    item->setIcon(0, pixmap);
    isDirectory[urlInfo.name()] = urlInfo.isDir();
    //存储该路径是否为目录的信息
    ui->fileList->addTopLevelItem(item);
    if (!ui->fileList->currentItem()) {
        ui->fileList->setCurrentItem(ui->fileList->topLevelItem(0));
        ui->fileList->setEnabled(true);
    }
}

```

当 ftp->list() 函数执行时会发射 listInfo() 信号，此时就会执行 addToList() 函数，在这里我们将文件信息显示在 Tree Widget 上，并在 isDirectory 中存储该文件的路径及其是否为目录的信息。为了使文件与目录进行区分，我们使用了不同的图标 file.png 和 dir.png 来表示它们，这两个图标放在了工程文件夹中。

### (4) 将构造函数的内容更改如下。

```

{
    ui->setupUi(this);
    ui->progressBar->setValue(0);
    connect(ui->fileList,SIGNAL(itemActivated(QTreeWidgetItem*,int)),
            this,SLOT(processItem(QTreeWidgetItem*,int)));
    //鼠标双击列表中的目录时，我们进入该目录
}

```

这里我们只是让进度条的值为 0，然后关联了 Tree Widget 的一个信号 itemActivated()。当鼠标双击一个条目时，发射该信号，我们在槽函数中判断该条目是否为目录，如果是则进入该目录。

(5) processItem()函数的实现如下。

```
void Widget::processItem(QTreeWidgetItem* item,int) //打开一个目录
```

```
{
    QString name = item->text(0);
    if (isDirectory.value(name)) { //如果这个文件是个目录，则打开
        ui->fileList->clear();
        isDirectory.clear();
        currentPath += '/';
        currentPath += name;
        ftp->cd(name);
        ftp->list();
        ui->cdToParentButton->setEnabled(true);
    }
}
```

(6) “返回上一级目录”按钮的单击事件槽函数如下。

```
void Widget::on_cdToParentButton_clicked() //返回上级目录按钮
{
    ui->fileList->clear();
    isDirectory.clear();
    currentPath = currentPath.left(currentPath.lastIndexOf('/'));
    if (currentPath.isEmpty()) {
        ui->cdToParentButton->setEnabled(false);
        ftp->cd("/");
    } else {
        ftp->cd(currentPath);
    }
    ftp->list();
}
```

在返回上一级目录时，我们取当前路径的最后一个“/”之前的部分，如果此时路径为空了，我们就让“返回上一级目录”按钮不可用。

(7) “下载”按钮单击事件槽函数如下。

```
void Widget::on_downloadButton_clicked() //下载按钮
{
    QString fileName = ui->fileList->currentItem()->text(0);
    QFile *file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly))
    {
        delete file;
        return;
    }
    ui->downloadButton->setEnabled(false); //下载按钮不可用，等下载完成后才可用
    ftp->get(ui->fileList->currentItem()->text(0), file);
}
```

在这里我们获取了当前项目的文件名，然后新建文件，使用 get()命令下载服务器上的文件到我们新建的文件中。

(8) 更新进度条函数内容如下。

```
void Widget::updateDataTransferProgress( //进度条
```

```
    qint64 readBytes,qint64 totalBytes)
{
    ui->progressBar->setMaximum(totalBytes);
    ui->progressBar->setValue(readBytes);
}
```

4.运行程序，效果如下。

开始界面如下。

登录成功时界面如下。

下载文件时界面如下。

当一个目录为空时界面如下。

4.流程说明。

整个程序的流程就和我们实现函数的顺序一样。用户在界面上输入服务器的相关信息，然后我们利用这些信息进行连接并登录服务器，等登录服务器成功时，我们列出服务器上所有的文件。对于一个目录，我们可以进入其中，并返回上一级目录，我们可以下载文件，并显示下载的进度。

对于 ftp 的操作，全部由那些命令和信号来完成，我们只需要调用相应的命令，并在其发出信号时，进行对应的处理就可以了。而对于文件的显示，则是视图部分的知识了。

5.其他说明。

最后需要说明的是，因为为了更好的讲解知识，使得程序简单化，所以我们省去了很多细节上的处理，如果需要，你可以自己添加。比如断开连接和取消下载，你都可以使用

ftp->abort() 函数。你也可以参考 Qt 自带的 Ftp Example 例子。对于其他操作，比如上传等，你可以根据需要添加。

FTP 的相关编程就讲到到这里。

( 错误更改提示：——2010 年 08 月 19 日

在程序中的 QFile 应该使用全局变量，然后 file = new QFile(fileName);

当下载完成后，我们要使用 file->close(); 关闭文件，不然下载的小文件可能为空。

更改如下：

1. 在 widget.h 中的声明 private 对象

QFile \*file;

2. 在 widget.cpp 文件中：

( 1 ) 在 void Widget::on\_downloadButton\_clicked() 函数里将 file 的定义改为：

file = new QFile(fileName);

( 2 ) 然后在 void Widget::ftpCommandFinished(int,bool error) // 结束命令

函数中更改：

```
if (ftp->currentCommand() == QFtp::Get){
    if(error) ui->label->setText(tr("下载出现错误: %1").arg(ftp->errorString()));
    else {
        ui->label->setText(tr("已经完成下载"));
        ui->downloadButton->setEnabled(true);
        file->close(); // 添加这行代码
    }
}
```

分类：[Qt 系列教程](#) 作者：[yafeilinux](#) 日期：六月 22nd, 2010. 1,861 views  
Tags: [ftp](#), [QFtp](#), [Qt 网络](#)

## 四十五、Qt 网络（五）获取本机网络信息

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

前面讲完了 HTTP 和 FTP，下面本来该讲解 UDP 和 TCP 了。不过，在讲解它们之前，我们先在这一节里讲解一个以后要经常用到的名词，那就是 IP 地址。

对于 IP 地址，其实，会上网的人都应该听说过它。如果你实在很不了解它，那么我们简单的说：IP 即 Internet Protocol（网络之间互联的协议），协议就是规则，地球人都用一样的规则，所以我们可以访问全球任何的网站；而 IP 地址就是你联网时分配给你机子的一个地址。如果把网络比喻成地图，那 IP 地址就像地图上的经纬度一样，它确定了你的主机在网络中的位置。其实知道我们以后要用 IP 地址来代表网络中的一台计算机就够了。（^\_~不一定科学但是很直白的表述）

下面我们就讲解如何获取自己电脑的 IP 地址以及其他网络信息。这一节中，我们会涉及到网络模块（QtNetwork Module）中的 QHostInfo，QHostAddress，QNetworkInterface 和 QNetworkAddressEntry 等几个类。下面是详细内容。

我们新建 Qt4 Gui Application 工程，工程名为 myIP，选中 QtNetwork 模块，Base class 选择 QWidget。

我们在 widget.h 文件中包含头文件：#include <QtNetwork>

1. 使用 QHostInfo 获取主机名和 IP 地址。

（1）获取主机名。

我们在 widget.cpp 文件中的构造函数中添加代码：

```
QString localHostName = QHostInfo::localHostName();
```

```
qDebug() << "localHostName: " << localHostName;
```

这里我们使用了 QHostInfo 类的 localHostName 类来获取本机的计算机名称。

运行程序，在下面的输出栏里的信息如下：

可以看到，这里获取了计算机名。我们可以在桌面上“我的电脑”图标上点击鼠标右键，然后选择“属性”菜单，查看“计算机名”一项，和我们的输出结果是一样的，如下图。

（2）获取本机的 IP 地址。

我们继续在构造函数中添加代码：

```
QHostInfo info = QHostInfo::fromName(localHostName);
```

```
qDebug() << "IP Address: " << info.addresses();
```

我们应用 QHostInfo 类的 fromName() 函数，使用上面获得的主机名为参数，来获取本机的信息。然后再利用 QHostInfo 类的 addresses() 函数，获取本机的所有 IP 地址信息。运行程序，输出信息如下：

在我这里只有一条 IP 地址。但是，在其他系统上，可能出现多条 IP 地址，其中可能包含了 IPv4 和 IPv6 的地址，一般我们需要使用 IPv4 的地址，所以我们可以只输出 IPv4 的地址。

我们继续添加代码：

```
foreach(QHostAddress address,info.addresses())
{
    if(address.protocol() == QAbstractSocket::IPv4Protocol)
        qDebug() << address.toString();
}
```

因为 IP 地址由 QHostAddress 类来管理，所以我们可以使用该类来获取一条 IP 地址，然后使用该类的 protocol() 函数来判断其是否为 IPv4 地址。如果是 IPv6 地址，可以使用 QAbstractSocket::IPv6Protocol 来判断。最后我们将 IP 地址以 QString 类型输出。

我们以后要使用的 IP 地址都是用这个方法获得的，所以这个一定要掌握。运行效果如下：

（3）以主机名获取 IP 地址。

我们在上面讲述了用本机的计算机名获取本机的 IP 地址。其实 QHostInfo 类也可以用来获取任意主机名的 IP 地址，如一个网站的 IP 地址。在这里我们可以使用 lookupHost() 函数。它是基于信号和槽的，一旦查找到了 IP 地址，就会触发槽函数。具体用法如下。

我们在 widget.h 文件中添加一个私有槽函数：

```
private slots:
void lookedUp(const QHostInfo &host);
```

然后在 widget.cpp 中的构造函数中先将上面添加的代码全部删除，然后添加以下代码：

```
QHostInfo::lookupHost("www.baidu.com",
```

```
    this,SLOT(lookedUp(QHostInfo)));
```

这里我们查询百度网站的 IP 地址，如果查找到，就会执行我们的 lookedUp() 函数。

在 widget.cpp 中添加 lookedUp() 函数的实现代码：

```
void Widget::lookedUp(const QHostInfo &host)
{
    qDebug() << host.addresses().first().toString();
}
```

这里我们只是简单地输出第一个 IP 地址。输出信息如下：

其实，我们也可以使用 lookupHost() 函数，通过输入 IP 地址反向查找主机名，只需要将上面代码中的 “www.baidu.com”换成一个 IP 地址就可以了，如果你有兴趣可以研究一下，不过返回的结果可能不是你想象中的那样。

**小结：可以看到 QHostInfo 类的作用：通过主机名来查找 IP 地址，或者通过 IP 地址来反向查找主机名。**

2. 通过 QNetworkInterface 类来获取本机的 IP 地址和网络接口信息。

QNetworkInterface 类提供了程序所运行时的主机的 IP 地址和网络接口信息的列表。在每一个网络接口信息中都包含了 0 个或多个 IP 地址，而每一个 IP 地址又包含了和它相关的子网掩码和广播地址，它们三者被封装在一个 QNetworkAddressEntry 对象中。网络接口信息中也提供了硬件地址信息。我们将 widge.cpp 构造函数中以前添加的代码删除，然后添加以下代码。

```
QList<QNetworkInterface> list = QNetworkInterface::allInterfaces();
// 获取所有网络接口的列表
foreach(QNetworkInterface interface,list)
{
    // 遍历每一个网络接口
    qDebug() << "Device: "<<interface.name();
    // 设备名
    qDebug() << "HardwareAddress: "<<interface.hardwareAddress();
    // 硬件地址
    QList<QNetworkAddressEntry> entryList = interface.addressEntries();
    // 获取 IP 地址条目列表，每个条目中包含一个 IP 地址，一个子网掩码和一个广播地址
    foreach(QNetworkAddressEntry entry,entryList)
    {
        // 遍历每一个 IP 地址条目
        qDebug()<<"IP Address: "<<entry.ip().toString();
        // IP 地址
        qDebug()<<"Netmask: "<<entry.netmask().toString();
        // 子网掩码
        qDebug()<<"Broadcast: "<<entry.broadcast().toString();
        // 广播地址
    }
}
```

这里我们获取了本机的网络设备的相关信息。运行程序，输出如下：

其实，如果我们只想利用 QNetworkInterface 类来获取 IP 地址，那么就没必要像上面那样复杂，这个类提供了一个便捷的函数 allAddresses() 来获取 IP 地址，例如：

```
QString address = QNetworkInterface::allAddresses().first().toString();
```

3. 总结。

在这一节中我们学习了如何来查找本机网络设备的相关信息。其实，以后最常用的还是其中获取 IP 地址的方法。我们以后可以利用一个函数来获取 IP 地址：

```
QString      Widget::getIP()      //      获      取      ip      地      址
{
    QList<QHostAddress>      list      =      QNetworkInterface::allAddresses();
    foreach      (QHostAddress      address,      list)
    {
        if(address.protocol() == QAbstractSocket::IPv4Protocol)
        //      我      们      使      用      IPv4      地      址
        return      address.toString();
    }
}
```

```
        return 0;  
    }
```

这一节就讲到这里，在下面的几节中我们将利用 IP 地址进行 UDP 和 TCP 的编程。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：六月 25th, 2010. 2,232 views

Tags: [IPv4](#), [IPv6](#), [IP 地址](#), [QHostAddress](#), [QHostInfo](#), [QNetworkAddressEntry](#), [QNetworkInterface](#), [Qt 网络](#)

## 四十六、Qt 网络（六）UDP

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

这一节讲述 UDP 编程的知识。UDP ( User Datagram Protocol 即用户数据报协议 ) 是一个轻量级的，不可靠的，面向数据报的无连接协议。对于 UDP 我们不再进行过多介绍，如果你对 UDP 不是很了解，而且不知道它有什么用，那么我们这里就举个简单的例子：我们现在几乎每个人都使用的腾讯 QQ，其聊天时就是使用 UDP 协议进行消息发送的。就像 QQ 那样，当有很多用户，发送的大部分都是短消息，要求能及时响应，并且对安全性要求不是很高的情况下使用 UDP 协议。

在 Qt 中提供了 QUdpSocket 类来进行 UDP 数据报 ( datagrams ) 的发送和接收。这里我们还要了解一个名词 Socket，也就是常说的“套接字”。Socket 简单地说，就是一个 IP 地址加一个 port 端口。因为我们要传输数据，就要知道往哪个机子上传送，而 IP 地址确定了一台主机，但是这台机子上可能运行着各种各样的网络程序，我们要往哪个程序中发送呢？这时就要使用一个端口来指定 UDP 程序。所以说，Socket 指明了数据报传输的路径。

下面我们将编写两个程序，一个用来发送数据报，可以叫做客户端；另一个用来接收数据报，可以叫做服务器端，它们均应用 UDP 协议。这样也就构成了所谓的 C/S ( 客户端/服务器 ) 编程模型。我们会在编写程序的过程中讲解一些相关的网络知识。

### (一) 发送端 (客户端)

1. 我们新建 Qt4 Gui Application，工程名为“udpSender”，选中 QtNetwork 模块，Base class 选择 QWidget。

2. 我们在 widget.ui 文件中，往界面上添加一个 Push Button，更改其显示文本为“开始广播”，然后进入其单击事件槽函数。

3. 我们在 widget.h 文件中更改。

添加头文件：#include <QtNetwork>

添加 private 私有对象：QUdpSocket \*sender;

4. 我们在 widget.cpp 中进行更改。

在构造函数中添加：sender = new QUdpSocket(this);

更改“开始广播”按钮的单击事件槽函数：

```
void Widget::on_pushButton_clicked() //发送广播
{
```

```
    QByteArray datagram = "hello world!";
    sender->writeDatagram(datagram.data(),datagram.size(),
                           QHostAddress::Broadcast,45454);
```

}

这里我们定义了一个 QByteArray 类型的数据报 datagram，其内容为“hello world!”。然后我们使用 QUdpSocket 类的 writeDatagram() 函数来发送数据报，这个函数有四个参数，分别是数据报的内容，数据报的大小，主机地址和端口号。对于数据报的大小，它根据平台的不同而不同，但是这里建议不要超过 512 字节。这里我们使用了广播地址 QHostAddress::Broadcast，这样就可以同时给网络中所有的主机发送数据报了。对于端口号，它是可以随意指定的，但是一般 1024 以下的端口号通常属于保留端口号，所以我们最好使用大于 1024 的端口，最大为 65535。我们这里使用了 45454 这个端口号，一定要注意，在下面要讲的服务器程序中，也要使用相同的端口号。

5. 发送端就这么简单，我们运行程序，效果如下。

### (二) 接收端 (服务器端)

1. 我们新建 Qt4 Gui Application，工程名为“udpReceiver”，选中 QtNetwork 模块，Base class 选择 QWidget。此时工程文件列表中应包含两个工程，如下图。

2. 我们在 udpReceiver 工程中的 widget.ui 文件中，向界面上添加一个 Label 部件，更改其显示文本为“等待接收数据！”，效果如下。

3. 我们在 udpReceiver 工程中的 widget.h 文件中更改。

添加头文件：#include <QtNetwork>

添加 private 私有对象：QUdpSocket \*receiver;

添加私有槽函数：

private slots:

```
void processPendingDatagram();
```

4. 我们在 udpReceiver 工程中的 widget.cpp 文件中更改。

在构造函数中：

```
receiver = new QUdpSocket(this);
receiver->bind(45454,QUdpSocket::ShareAddress);
connect(receiver,SIGNAL(readyRead()),this,SLOT(processPendingDatagram()));
我们在构造函数中将 receiver 绑定到 45454 端口，这个端口就是上面发送端设置的端口，二者必须一样才能保证接收到数据报。我们这里使用了绑定模式 QUdpSocket::ShareAddress，它表明其他服务也可以绑定到这个端口上。因为当 receiver 发现有数据报到达时就会发出 readyRead()信号，所以我们将其和我们的数据报处理函数相关联。
```

数据报处理槽函数实现如下：

```
void Widget::processPendingDatagram() //处理等待的数据报
{
    while(receiver->hasPendingDatagrams()) //拥有等待的数据报
    {
        QByteArray datagram; //用于存放接收的数据报
        datagram.resize(receiver->pendingDatagramSize());
        //让 datagram 的大小为等待处理的数据报的大小，这样才能接收到完整的数据
        receiver->readDatagram(datagram.data(),datagram.size());
        //接收数据报，将其存放到 datagram 中
        ui->label->setText(datagram);
        //将数据报内容显示出来
    }
}
```

5.我们在工程列表中 udpReceiver 工程上点击鼠标右键，在弹出的菜单上选择 run 菜单来运行该工程。

6.第一次运行该程序时，系统可能会提示警告，我们选择“解除阻止”。

如果是在 linux 下，你可能还需要关闭防火墙。

7.我们同时再运行 udpSender 程序。然后点击其上的“发送广播”按钮，这时会在 udpReceiver 上显示数据报的内容。效果如下。

可以看到，UDP 的应用是很简单的。我们只需要在发送端执行 writeDatagram() 函数进行数据报的发送，然后在接收端绑定端口，并关联 readyRead() 信号和数据报处理函数即可。

[下一节我们讲述 TCP 的应用。](#)

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：六月 28th, 2010. 2,247 views  
Tags: [QtNetwork](#), [Qt 网络](#), [QUdpSocket](#), [UDP](#), [教程](#)

## 四十七、Qt 网络 (七) TCP(一)

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

TCP 即 Transmission Control Protocol，传输控制协议。与 UDP 不同，它是面向连接和数据流的可靠传输协议。也就是说，它能使一台计算机上的数据无差错的发往网络上的其他计算机，所以当要传输大量数据时，我们选用 TCP 协议。

TCP 协议的程序使用的是客户端/服务器模式，在 Qt 中提供了 QTcpSocket 类来编写客户端程序，使用 QTcpServer 类编写服务器端程序。我们在服务器端进行端口的监听，一旦发现客户端的连接请求，就会发出 newConnection() 信号，我们可以关联这个信号到我们自己的槽函数，进行数据的发送。而在客户端，一旦有数据到来就会发出 readyRead() 信号，我们可以关联此信号，进行数据的接收。其实，在程序中最难理解的地方就是程序的发送和接收了，为了让大家更好的理解，我们在这一节只是讲述一个传输简单的字符串的例子，在下一节再进行扩展，实现任意文件的传输。

### 一、服务器端。

在服务器端的程序中，我们监听本地主机的一个端口，这里使用 6666，然后我们关联 newConnection() 信号与自己写的 sendMessage() 槽函数。就是说一旦有客户端的连接请求，就会执行 sendMessage() 函数，在这个函数里我们发送一个简单的字符串。

1. 我们新建 Qt4 Gui Application，工程名为“tcpServer”，选中 QtNetwork 模块，Base class 选择 QWidget。（说明：如果一些 Qt Creator 版本没有添加模块一项，我们就需要在工程文件 tcpServer.pro 中添加一行代码：QT += network）

2. 我们在 widget.ui 的设计区添加一个 Label，更改其 objectName 为 statusLabel，用于显示一些状态信息。如下：

3. 在 widget.h 文件中做以下更改。

添加头文件：#include <QtNetWork>  
添加 private 对象：QTcpServer \*tcpServer;

添加私有槽函数：

private slots:  
void sendMessage();

4. 在 widget.cpp 文件中进行更改。

在其构造函数中添加代码：

```
tcpServer = new QTcpServer(this);
if(!tcpServer->listen(QHostAddress::LocalHost,6666))
{
    //监听本地主机的 6666 端口，如果出错就输出错误信息，并关闭
    qDebug() << tcpServer->errorString();
    close();
}
connect(tcpServer,SIGNAL(newConnection()),this,SLOT(sendMessage()));
```

//连接信号和相应槽函数

我们在构造函数中使用 tcpServer 的 listen() 函数进行监听，然后关联了 newConnection() 和我们自己的 sendMessage() 函数。

下面我们实现 sendMessage() 函数。

```
void Widget::sendMessage()
{
    QByteArray block; //用于暂存我们要发送的数据
    QDataStream out(&block,QIODevice::WriteOnly);
    //使用数据流写入数据
    out.setVersion(QDataStream::Qt_4_6);
    //设置数据流的版本，客户端和服务器端使用的版本要相同
    out<<(quint16) 0;
    out<<tr("hello Tcp!!!");
    out.device()->seek(0);
    out<<(quint16) (block.size() - sizeof(quint16));
    QTcpSocket *clientConnection = tcpServer->nextPendingConnection();
    //我们获取已经建立的连接的子套接字
    connect(clientConnection,SIGNAL(disconnected()),clientConnection,
            SLOT(deleteLater()));
    clientConnection->write(block);
```

```
clientConnection->disconnectFromHost();
ui->statusLabel->setText("send message successful!!!");
//发送数据成功后，显示提示
}
```

这个是数据发送函数，我们主要介绍两点：

(1) 为了保证在客户端能接收到完整的文件，我们都在数据流的最开始写入完整文件的大小信息，这样客户端就可以根据大小信息来判断是否接受到了完整的文件。而在服务器端，我们在发送数据时就要首先发送实际文件的大小信息，但是，文件的大小一开始是无法预知的，所以我们先使用了 out<<(quint16) 0; 在 block 的开始添加了一个 quint16 大小的空间，也就是两字节的空间，它用于后面放置文件的大小信息。然后 out<<tr("hello Tcp!!!"); 输入实际的文件，这里是字符串。当文件输入完成后我们在使用 out.device()->seek(0); 返回到 block 的开始，加入实际的文件大小信息，也就是后面的代码，它是实际文件的大小：out<<(quint16) (block.size() - sizeof(quint16));

(2) 在服务器端我们可以使用 tcpServer 的 nextPendingConnection() 函数来获取已经建立的连接的 Tcp 套接字，使用它来完成数据的发送和其它操作。比如这里，我们关联了 disconnected() 信号和 deleteLater() 槽函数，然后我们发送数据

```
clientConnection->write(block);
```

然后是 clientConnection->disconnectFromHost(); 它表示当发送完成时就会断开连接，这时就会发出 disconnected() 信号，而最后调用 deleteLater() 函数保证在关闭连接后删除该套接字

```
clientConnection.
```

5. 这样服务器的程序就完成了，我们先运行一下程序。

## 二、客户端。

我们在客户端程序中向服务器发送连接请求，当连接成功时接收服务器发送的数据。

1. 我们新建 Qt4 Gui Application，工程名为“tcpClient”，选中 QtNetwork 模块，Base class 选择 QWidget。

2. 我们在 widget.ui 中添加几个标签 Label 和两个 Line Edit 以及一个按钮 Push Button。

其中“主机”后的 Line Edit 的 objectName 为 hostLineEdit，“端口号”后的为 portLineEdit。“收到的信息”标签的 objectName 为 messageLabel。

3. 在 widget.h 文件中做更改。

添加头文件：#include <QtNetwork>

添加 private 变量：

```
QTcpSocket *tcpSocket;
QString message; //存放从服务器接收到的字符串
quint16 blockSize; //存放文件的大小信息
```

添加私有槽函数：

```
private slots:
    void newConnect(); //连接服务器
    void readMessage(); //接收数据
    void displayError(QAbstractSocket::SocketError); //显示错误
```

4. 在 widget.cpp 文件中做更改。

(1) 在构造函数中添加代码：

```
tcpSocket = new QTcpSocket(this);
connect(tcpSocket,SIGNAL(readyRead()),this,SLOT(readMessage()));
connect(tcpSocket,SIGNAL(error(QAbstractSocket::SocketError)),
        this,SLOT(displayError(QAbstractSocket::SocketError)));
```

这里关联了 tcpSocket 的两个信号，当有数据到来时发出 readyRead() 信号，我们执行读取数据的 readMessage() 函数。当出现错误时发出 error() 信号，我们执行 displayError() 槽函数。

(2) 实现 newConnect() 函数。

```
void Widget::newConnect()
```

```
{
    blockSize = 0; //初始化其为 0
    tcpSocket->abort(); //取消已有的连接
    tcpSocket->connectToHost(ui->hostLineEdit->text(),
                            ui->portLineEdit->text().toInt());
    //连接到主机，这里从界面获取主机地址和端口号
```

```
}
```

这个函数实现了连接到服务器，下面会在“连接”按钮的单击事件槽函数中调用这个函数。

(3) 实现 readMessage() 函数。

```
void Widget::readMessage()
```

```
{
```

```
    QDataStream in(tcpSocket);
```

```
    in.setVersion(QDataStream::Qt_4_6);
```

```
//设置数据流版本，这里要和服务器端相同
```

```
    if(blockSize==0) //如果是刚开始接收数据
```

```
{
```

```
    //判断接收的数据是否有两字节，也就是文件的大小信息
```

```
    //如果有则保存到 blockSize 变量中，没有则返回，继续接收数据
```

```
    if(tcpSocket->bytesAvailable() < (int)sizeof(quint16)) return;
```

```
    in >> blockSize;
```

```
}
```

```
    if(tcpSocket->bytesAvailable() < blockSize) return;
```

```
//如果没有得到全部的数据，则返回，继续接收数据
```

```
    in >> message;
```

```
//将接收到的数据存放到变量中
```

```
    ui->messageLabel->setText(message);
```

```
//显示接收到的数据
```

```
}
```

这个函数实现了数据的接收，它与服务器端的发送函数相对应。首先我们要获取文件的大小信息，然后根据文件的大小来判断是否接收到了完整的文件。

(4) 实现 displayError() 函数。

```
void Widget::displayError(QAbstractSocket::SocketError)
```

```
{
```

```
    qDebug() << tcpSocket->errorString(); //输出错误信息
```

```
}
```

这里简单的实现了错误信息的输出。

(5) 我们在 widget.ui 中进入“连接”按钮的单击事件槽函数，然后更改如下。

```
void Widget::on_pushButton_clicked() //连接按钮
```

```
{
```

```
    newConnect(); //请求连接
```

```
}
```

这里直接调用了 newConnect() 函数。

5. 我们运行程序，同时运行服务器程序，然后在“主机”后填入“localhost”，在“端口号”后填入“6666”，点击“连接”按钮，效果如下。

可以看到我们正确地接收到了数据。因为服务器端和客户端是在同一台机子上运行的，所以我这里填写了“主机”为“localhost”，如果你在不同的机子上运行，需要在“主机”后填写其正确的 IP 地址。

到这里我们最简单的 TCP 应用程序就完成了，在下一节我们将会对它进行扩展，实现任意文件的传输。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：七月 12th, 2010. 3,298 views  
Tags: [Qt Creator 2.0](#), [Qt 网络](#), [TCP](#), [yafeilinux](#), [教程](#)

## 四十八、Qt 网络（八）TCP(二)

本文章原创于 [www.yafeilinux.com](http://www.yafeilinux.com) 转载请注明出处。

在上一节里我们使用 TCP 服务器发送一个字符串，然后在 TCP 客户端进行接收。在这一节我们重新写一个客户端程序和一个服务器程序，这次我们让客户端进行文件的发送，服务器进行文件的接收。有了上一节的基础，这一节的内容就很好理解了，注意一下几个信号和槽的关联即可。当然，我们这次要更深入了解一下数据的发送和接收的处理方法。

### 一、客户端

这次我们先讲解客户端，在客户端里我们与服务器进行连接，一旦连接成功，就会发出 connected() 信号。这时我们就进行文件的发送。

在上一节我们已经看到，发送数据时我们先发送了数据的大小信息。这一次，我们要先发送文件的总大小，然后文件名长度，然后是文件名，这三部分我们合称为文件头结构，最后再发送文件数据。所以在发送函数里我们就要进行相应的处理，当然，在服务器的接收函数里我们也要进行相应的处理。对于文件大小，这次我们使用了 qint64，它是 64 位的，可以表示一个很大的文件了。

1. 同前一节，我们新建工程，将工程命名为“tcpSender”。注意添加 network 模块。

2. 我们在 widget.ui 文件中将界面设计如下。

这里“主机”后的 Line Edit 的 objectName 为 hostLineEdit；“端口”后的 Line Edit 的 objectName 为 portLineEdit；下面的 Progress Bar 的 objectName 为 clientProgressBar，其 value 属性设为 0；“状态”Label 的 objectName 为 clientStatusLabel；“打开”按钮的 objectName 为 openButton；“发送”按钮的 objectName 为 sendButton；

3. 在 widget.h 文件中进行更改。

(1) 添加头文件 #include <QtNetwork>

(2) 添加 private 变量：

QTcpSocket \*tcpClient;

QFile \*localFile; //要发送的文件

qint64 totalBytes; //数据总大小

qint64 bytesWritten; //已经发送数据大小

qint64 bytesToWrite; //剩余数据大小

qint64 loadSize; //每次发送数据的大小

QString fileName; //保存文件路径

QByteArray outBlock; //数据缓冲区，即存放每次要发送的数据

(3) 添加私有槽函数：

private slots:

void send(); //连接服务器

void startTransfer(); //发送文件大小等信息

void updateClientProgress(qint64); //发送数据，更新进度条

void displayError(QAbstractSocket::SocketError); //显示错误

void openFile(); //打开文件

4. 在 widget.cpp 文件中进行更改。

添加头文件：#include <QFileDialog>

(1) 在构造函数中添加代码：

loadSize = 4\*1024;

totalBytes = 0;

bytesWritten = 0;

bytesToWrite = 0;

tcpClient = new QTcpSocket(this);

connect(tcpClient,SIGNAL(connected()),this,SLOT(startTransfer()));

//当连接服务器成功时，发出 connected() 信号，我们开始传送文件

connect(tcpClient,SIGNAL(bytesWritten(qint64)),this,

SLOT(updateClientProgress(qint64)));

//当有数据发送成功时，我们更新进度条

connect(tcpClient,SIGNAL(error(QAbstractSocket::SocketError)),this,

SLOT(displayError(QAbstractSocket::SocketError)));

ui->sendButton->setEnabled(false);

//开始使“发送”按钮不可用

我们主要是进行了变量的初始化和几个信号和槽函数的关联。

(2) 实现打开文件函数。

```
void Widget::openFile() //打开文件
{
    fileName = QFileDialog::getOpenFileName(this);
    if(!fileName.isEmpty())
    {
        ui->sendButton->setEnabled(true);
        ui->clientStatusLabel->setText(tr("打开文件 %1 成功！")
                                         .arg(fileName));
    }
}
```

该函数将在下面的“打开”按钮单击事件槽函数中调用。

(3) 实现连接函数。

```
void Widget::send() //连接到服务器，执行发送
{
    ui->sendButton->setEnabled(false);
    bytesWritten = 0;
    //初始化已发送字节为 0
    ui->clientStatusLabel->setText(tr("连接中..."));
    tcpClient->connectToHost(ui->hostLineEdit->text(),
                             ui->portLineEdit->text().toInt());//连接
}
```

该函数将在“发送”按钮的单击事件槽函数中调用。

(4) 实现文件头结构的发送。

```
void Widget::startTransfer() //实现文件大小等信息的发送
{
    localFile = new QFile(fileName);
    if(!localFile->open(QFile::ReadOnly))
    {
        qDebug() << "open file error!";
        return;
    }
    totalBytes = localFile->size();
    //文件总大小
    QDataStream sendOut(&outBlock,QIODevice::WriteOnly);
    sendOut.setVersion(QDataStream::Qt_4_6);
    QString currentFileName = fileName.right(fileName.size() - fileName.lastIndexOf('/')-1);
    sendOut << qint64(0) << qint64(0) << currentFileName;
    //依次写入总大小信息空间，文件名大小信息空间，文件名
    totalBytes += outBlock.size();
    //这里的总大小是文件名大小等信息和实际文件大小的总和
    sendOut.device()->seek(0);
    sendOut << totalBytes << qint64((outBlock.size() - sizeof(qint64)*2));
    //返回 outBolock 的开始，用实际的大小信息代替两个 qint64(0)空间
    bytesToWrite = totalBytes - tcpClient->write(outBlock);
    //发送完头数据后剩余数据的大小
    ui->clientStatusLabel->setText(tr("已连接"));
    outBlock.resize(0);
}
```

(5) 下面是更新进度条，也就是发送文件数据。

```
void Widget::updateClientProgress(qint64 numBytes) //更新进度条，实现文件的传送
{
    bytesWritten += (int)numBytes;
    //已经发送数据的大小
    if(bytesToWrite > 0) //如果已经发送了数据
    {
```

```

        outBlock = localFile->read(qMin(bytesToWrite,loadSize));
        //每次发送 loadSize 大小的数据，这里设置为 4KB，如果剩余的数据不足 4KB,
        //就发送剩余数据的大小
        bytesToWrite -= (int)tcpClient->write(outBlock);
        //发送完一次数据后还剩余数据的大小
        outBlock.resize(0);
        //清空发送缓冲区
    }
else
{
    localFile->close(); //如果没有发送任何数据，则关闭文件
}
ui->clientProgressBar->setMaximum(totalBytes);
ui->clientProgressBar->setValue(bytesWritten);
//更新进度条
if(bytesWritten == totalBytes) //发送完毕
{
    ui->clientStatusLabel->setText(tr("传送文件 %1 成功").arg(fileName));
    localFile->close();
    tcpClient->close();
}
}

(6) 实现错误处理函数。
void Widget::displayError(QAbstractSocket::SocketError) //显示错误
{
    qDebug() << tcpClient->errorString();
    tcpClient->close();
    ui->clientProgressBar->reset();
    ui->clientStatusLabel->setText(tr("客户端就绪"));
    ui->sendButton->setEnabled(true);
}

```

(7) 我们从 widget.ui 中分别进行“打开”按钮和“发送”按钮的单击事件槽函数，然后更改如下。

```

void Widget::on_openButton_clicked() //打开按钮
{
    openFile();
}
void Widget::on_sendButton_clicked() //发送按钮
{
    send();
}

```

5. 我们为了使程序中的中文不显示乱码，在 main.cpp 文件中更改。

添加头文件：#include <QTextCodec>

在 main 函数中添加代码：QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

6. 运行程序，效果如下。

## 7. 程序整体思路分析。

我们设计好界面，然后按下“打开”按钮，选择我们要发送的文件，这时调用了 openFile() 函数。然后我们点击“发送”按钮，调用 send() 函数，与服务器进行连接。当连接成功时就会发出 connected() 信号，这时就会执行 startTransfer() 函数，进行文件头结构的发送，当发送成功时就会发出 bytesWritten(qint64) 信号，这时我们执行 updateClientProgress(qint64 numBytes) 进行文件数据的传输和进度条的更新。这里使用了一个 loadSize 变量，我们在构造函数中将其初始化为 4\*1024 即 4 字节，它的作用是，我们将整个大的文件分成很多小的部分进行发送，每部分为 4 字节。而当连接出现问题时就会发出 error(QAbstractSocket::SocketError) 信号，这时就会执行 displayError() 函数。对于程序中其他细节我们就不再分析，希望大家能自己编程研究一下。

二、服务器端。

我们在服务器端进行数据的接收。服务器端程序是很简单的，我们开始进行监听，一旦发现有连接请求就发出 newConnection() 信号，然后我们便接受连接，开始接收数据。

1. 新建工程，名字为“tcpReceiver”。

2. 我们更改 widget.ui 文件，设计界面如下。

其中“服务器端”Label 的 objectName 为 serverStatusLabel；进度条 Progress Bar 的 objectName 为 serverProgressBar，设置其 value 属性为 0；“开始监听”按钮的 objectName 为 startButton。效果如下。

3. 更改 widget.h 文件的内容。

(1) 添加头文件：#include <QtNetwork>

(2) 添加私有变量：

```
QTcpServer tcpServer;
QTcpSocket *tcpServerConnection;
qint64 totalBytes; //存放总大小信息
qint64 bytesReceived; //已收到数据的大小
qint64 fileNameSize; //文件名的大小信息
QString fileName; //存放文件名
QFile *localFile; //本地文件
```

QByteArray inBlock; //数据缓冲区

(3) 添加私有槽函数：

private slots:

```
void on_startButton_clicked();
void start(); //开始监听
void acceptConnection(); //建立连接
void updateServerProgress(); //更新进度条，接收数据
void displayError(QAbstractSocket::SocketError socketError);
//显示错误
```

4. 更改 widget.cpp 文件。

(1) 在构造函数中添加代码：

```
totalBytes = 0;
bytesReceived = 0;
fileNameSize = 0;
connect(&tcpServer, SIGNAL(newConnection()), this,
SLOT(acceptConnection()));
//当发现新连接时发出 newConnection() 信号
```

(2) 实现 start() 函数。

void Widget::start() //开始监听

```
{
    ui->startButton->setEnabled(false);
    bytesReceived = 0;
    if(!tcpServer.listen(QHostAddress::LocalHost, 6666))
    {
        qDebug() << tcpServer.errorString();
        close();
        return;
    }
    ui->serverStatusLabel->setText(tr("监听"));
}
```

(3) 实现接受连接函数。

void Widget::acceptConnection() //接受连接

```
{
    tcpServerConnection = tcpServer.nextPendingConnection();
    connect(tcpServerConnection, SIGNAL(readyRead()), this,
SLOT(updateServerProgress()));
    connect(tcpServerConnection,
SIGNAL(error(QAbstractSocket::SocketError)), this,
```

```

        SLOT(displayError(QAbstractSocket::SocketError)));
    ui->serverStatusLabel->setText(tr("接受连接"));
    tcpServer.close();
}
(4) 实现更新进度条函数。
void Widget::updateServerProgress() //更新进度条，接收数据
{
    QDataStream in(tcpServerConnection);
    in.setVersion(QDataStream::Qt_4_6);
    if(bytesReceived <= sizeof(qint64)*2)
    { //如果接收到的数据小于 16 个字节，那么是刚开始接收数据，我们保存到//来的头文件信息
        if((tcpServerConnection->bytesAvailable() >= sizeof(qint64)*2)
            && (fileNameSize == 0))
        { //接收数据总大小信息和文件名大小信息
            in >> totalBytes >> fileNameSize;
            bytesReceived += sizeof(qint64) * 2;
        }
        if((tcpServerConnection->bytesAvailable() >= fileNameSize)
            && (fileNameSize != 0))
        { //接收文件名，并建立文件
            in >> fileName;
            ui->serverStatusLabel->setText(tr("接收文件 %1 ...")
                .arg(fileName));
            bytesReceived += fileNameSize;
            localFile = new QFile(fileName);
            if(!localFile->open(QFile::WriteOnly))
            {
                qDebug() << "open file error!";
                return;
            }
        }
        else return;
    }
    if(bytesReceived < totalBytes)
    { //如果接收的数据小于总数据，那么写入文件
        bytesReceived += tcpServerConnection->bytesAvailable();
        inBlock = tcpServerConnection->readAll();
        localFile->write(inBlock);
        inBlock.resize(0);
    }
    ui->serverProgressBar->setMaximum(totalBytes);
    ui->serverProgressBar->setValue(bytesReceived);
    //更新进度条
    if(bytesReceived == totalBytes)
    { //接收数据完成时
        tcpServerConnection->close();
        localFile->close();
        ui->startButton->setEnabled(true);
        ui->serverStatusLabel->setText(tr("接收文件 %1 成功！")
            .arg(fileName));
    }
}
(5) 错误处理函数。
void Widget::displayError(QAbstractSocket::SocketError) //错误处理
{
    qDebug() << tcpServerConnection->errorString();
}

```

```
tcpServerConnection->close();
ui->serverProgressBar->reset();
ui->serverStatusLabel->setText(tr("服务端就绪"));
ui->startButton->setEnabled(true);
}
```

(6) 我们在 widget.ui 中进入“开始监听”按钮的单击事件槽函数，更改如下。

```
void Widget::on_startButton_clicked() //开始监听按钮
{
    start();
}
```

5. 我们为了使程序中的中文不显示乱码，在 main.cpp 文件中更改。

添加头文件：#include <QTextCodec>

在 main 函数中添加代码：QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

6. 运行程序，并同时运行 tcpSender 程序，效果如下。

我们先在服务器端按下“开始监听”按钮，然后在客户端输入主机地址和端口号，然后打开要发送的文件，点击“发送”按钮进行发送。

在这两节里我们介绍了 TCP 的应用，可以看到服务器端和客户度端都可以当做发送端或者接收端，而且数据的发送与接收只要使用相对应的协议即可，它是可以根据用户的需要来进行编程的，没有固定的形式。

分类：[Qt 系列教程](#) 作者：yafeilinux 日期：七月 12th, 2010. 2,622 views

Tags: [qt](#), [Qt 网络](#), [TCP](#), [yafeilinux](#), [教程](#)