

一、Qt Creator 的安装和 hello world 程序的编写（原创）

2009-10-17 21:45

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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)

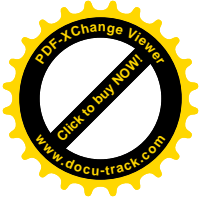
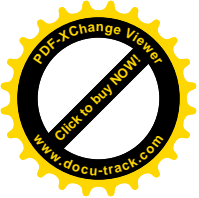
下载完成后，直接安装即可，安装过程中按默认设置即可。



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 菜单来新建工程。



4. 这里我们选择 Qt4 Gui Application。



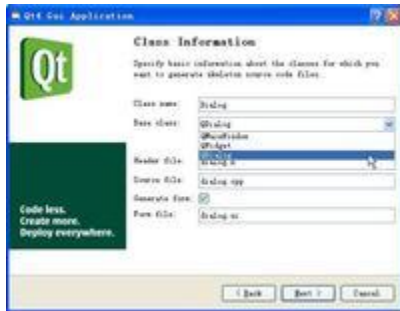
5. 下面输入工程名和要保存到的文件夹路径。我们这里的工程名为 helloworld。



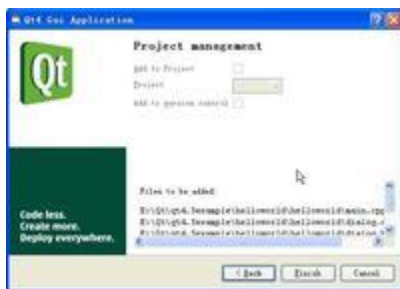
6. 这时软件自动添加基本的头文件，因为这个程序我们不需要其他的功能，所以直接点击 Next。



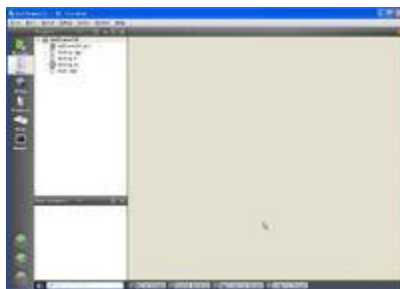
7. 我们将 base class 选为 QDialog 对话框类。然后点击 Next。



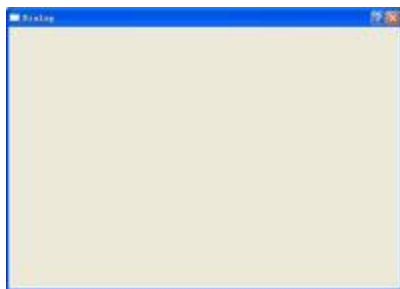
8. 点击 Finish，完成工程的建立。



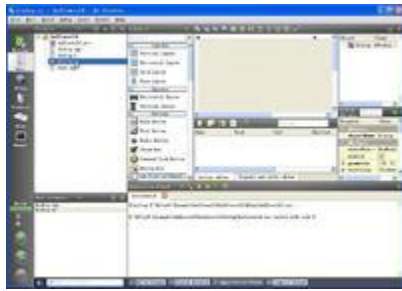
9. 我们可以看见工程中的所有文件都出现在列表中了。我们可以直接按下下面的绿色的 run 按钮或者按下 Ctrl+R 快捷键运行程序。



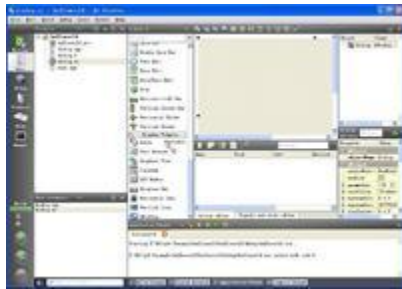
10. 程序运行会出现空白的对话框，如下图。



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。



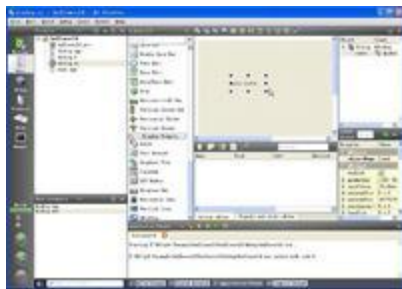
12. 我们在右边的器件栏里找到 Label 标签器件



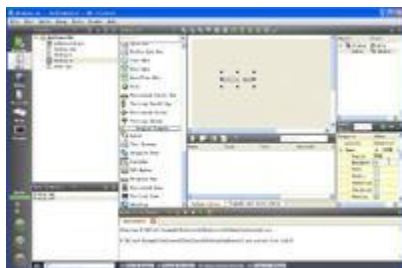
13. 按着鼠标左键将其拖到设计窗口上，如下图。

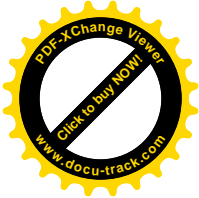
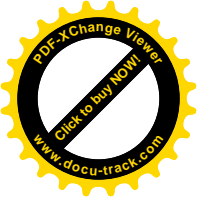


14. 我们双击它，并将其内容改为 helloworld。

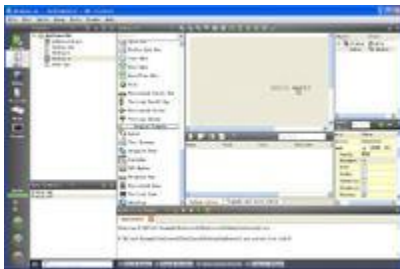


15. 我们在右下角的属性栏里将字体大小由 9 改为 15。





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 helloworld。



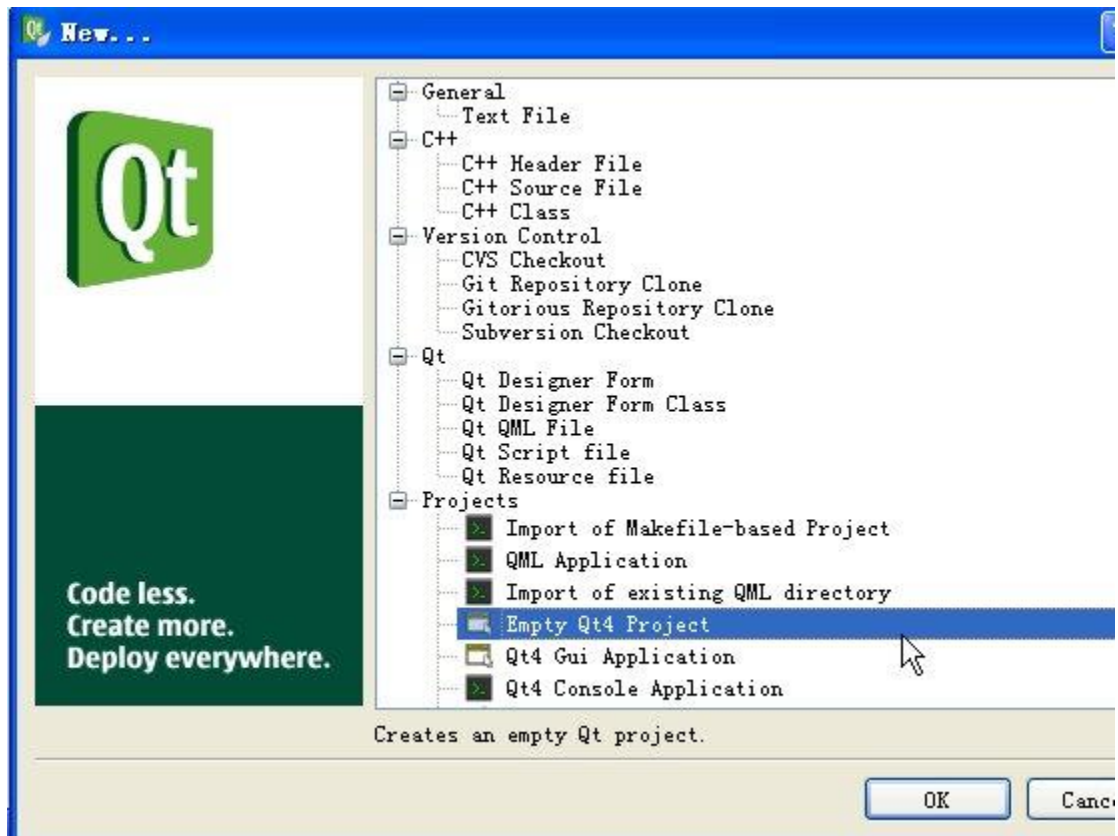
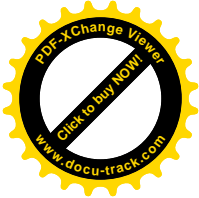
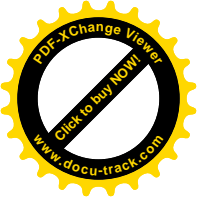
到这里 helloworld 程序便完成了。

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。

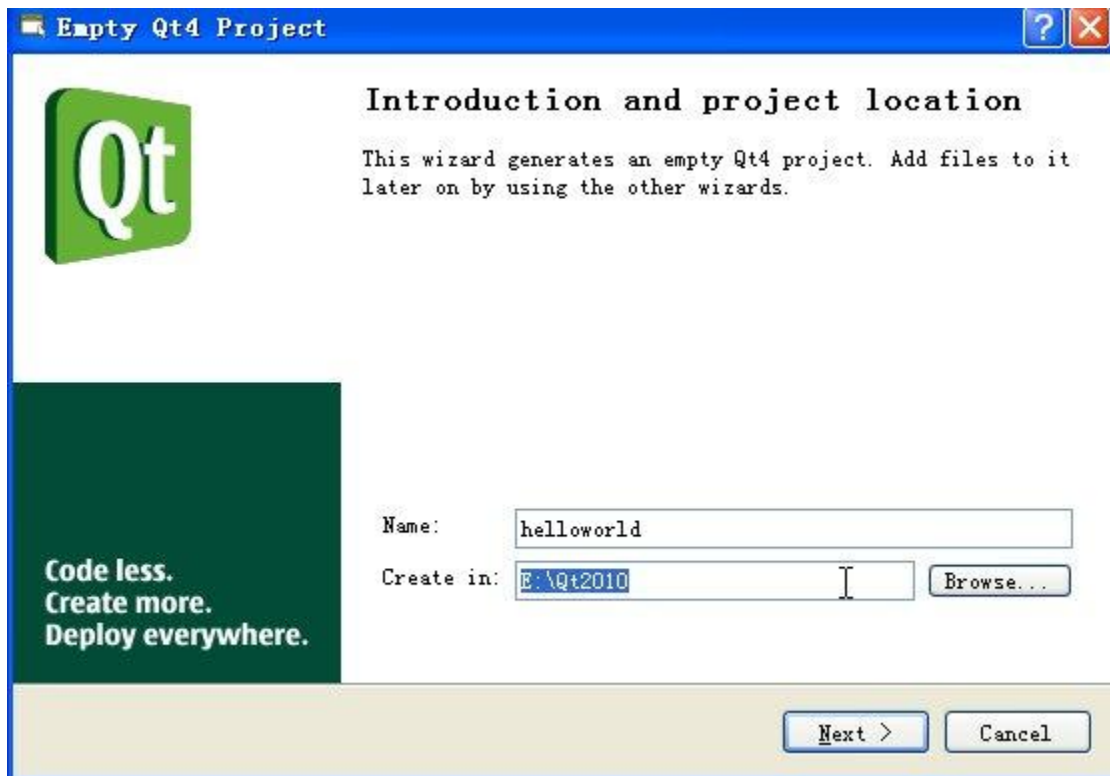
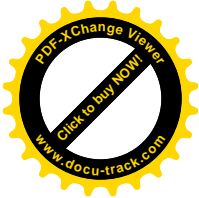
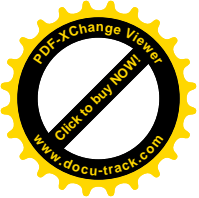
解决这个问题我们可以将相应的.dll 文件放到系统中。在 Qt Creator 的安装目录的 qt 文件下的 bin 文件夹下（我安装在了 D 盘，所以路径是 D:\Qt\2009.04\qt\bin），可以找到所有的相关.dll 文件。在这里找到 mingwm10.dll 文件，将其复制到 C:\WINDOWS\system 文件夹下，即可。下面再提示缺少什么 dll 文件，都像这样解决就可以了。

附 1：用纯源码编写。

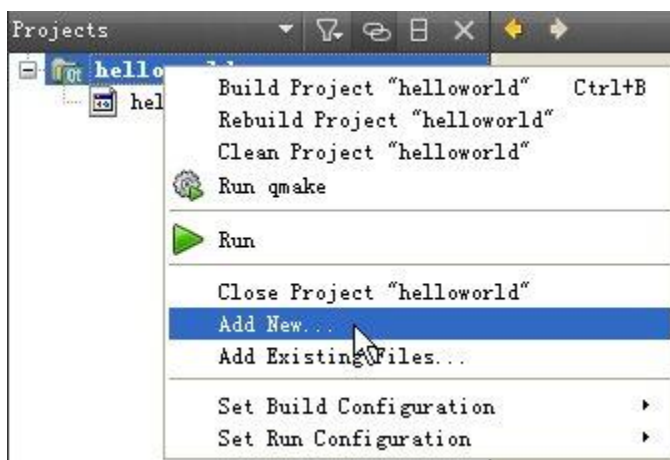
1. 新建空的 Qt4 工程。



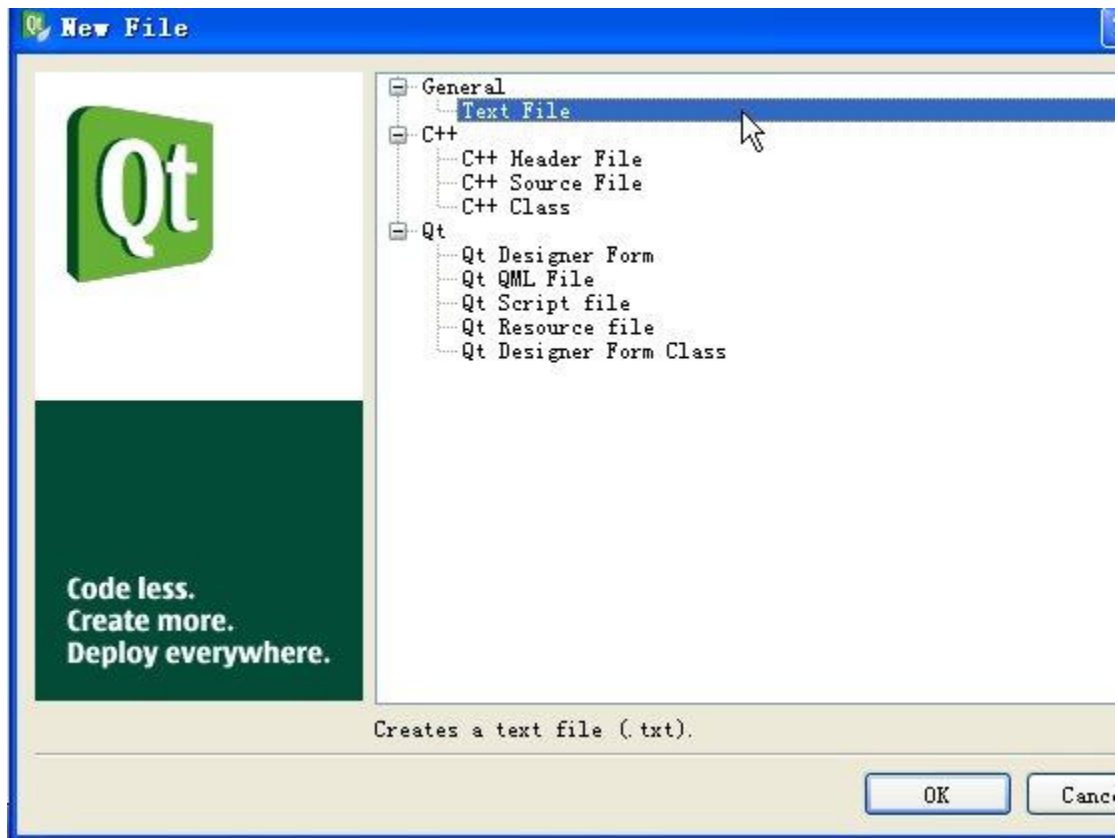
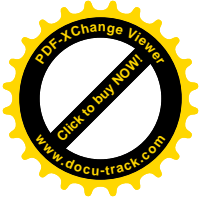
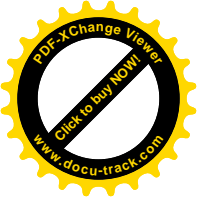
2. 工程名为 helloworld, 并选择工程保存路径 (提示: 路径中不能有中文)。



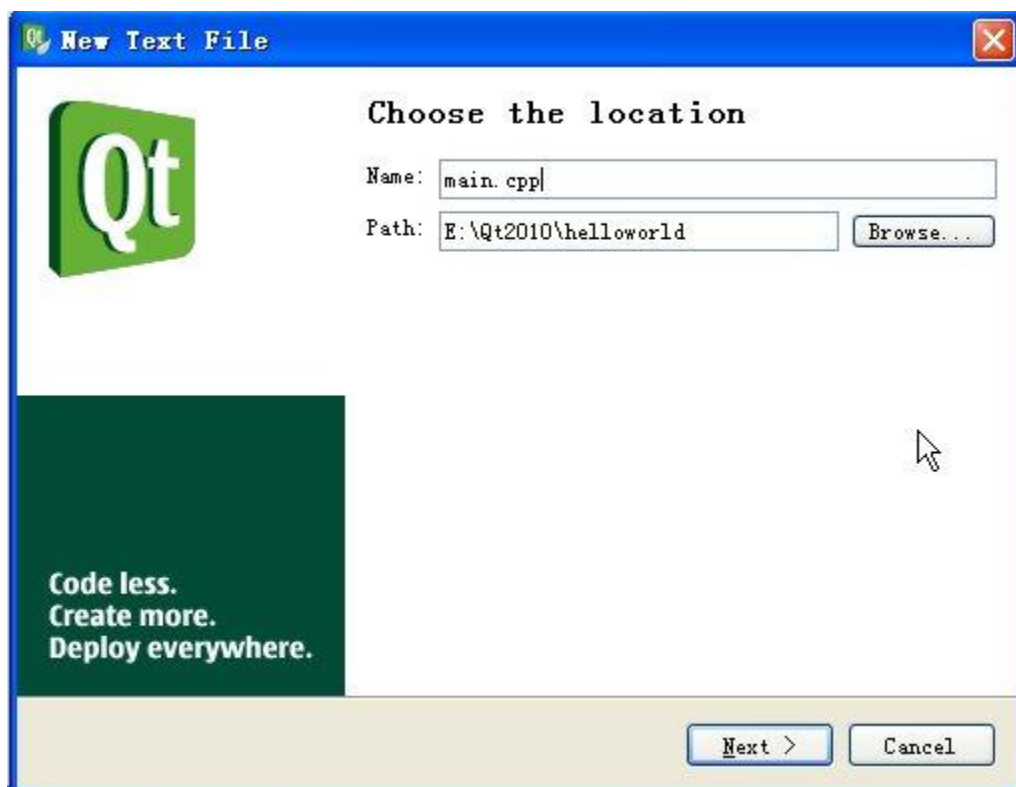
3. 在新建好的工程中添加文件。右击工程文件夹，弹出的菜单中选择 Add New。

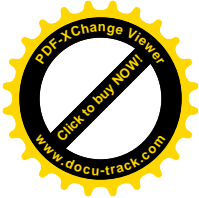
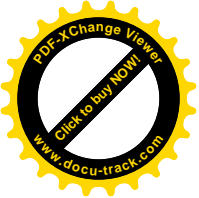


4. 选择普通文件。点击 Ok。

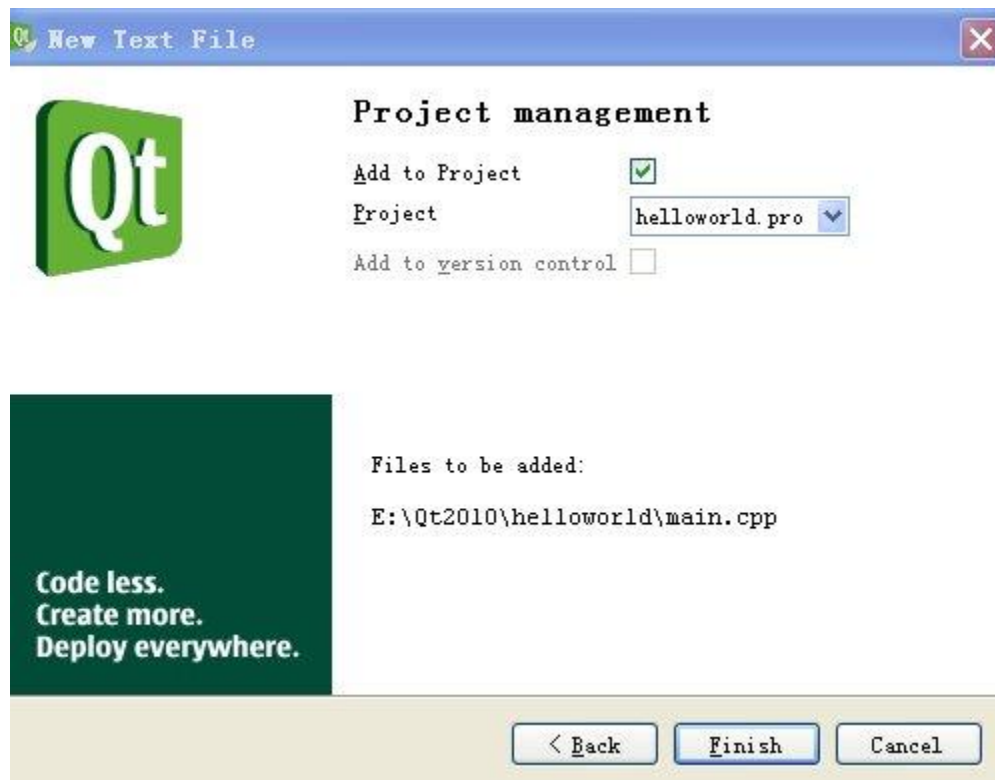


5. 文件名为 main.cpp，点击 Next 进入下一步。





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。

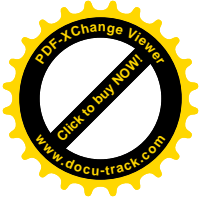
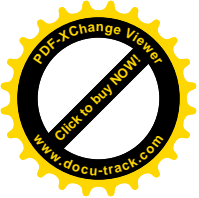


7. 在 main.cpp 文件中添加代码。

```
#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。

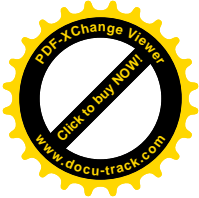
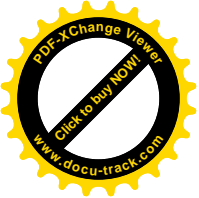
```
#include <QtGui>

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。



```
int main(int argc, char* argv[])
{
    QApplication
    QDialog* dd
    dd -> show()

    return app.e
}
```



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);

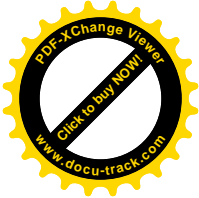
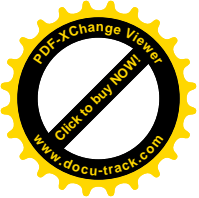
    QDialog* dd = new QDialog();

    QLabel* label = new QLabel(dd);
    label->setText("hello world");

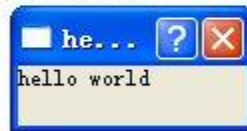
    dd -> show();

    return app.exec();
}
```

12. 运行效果如下。

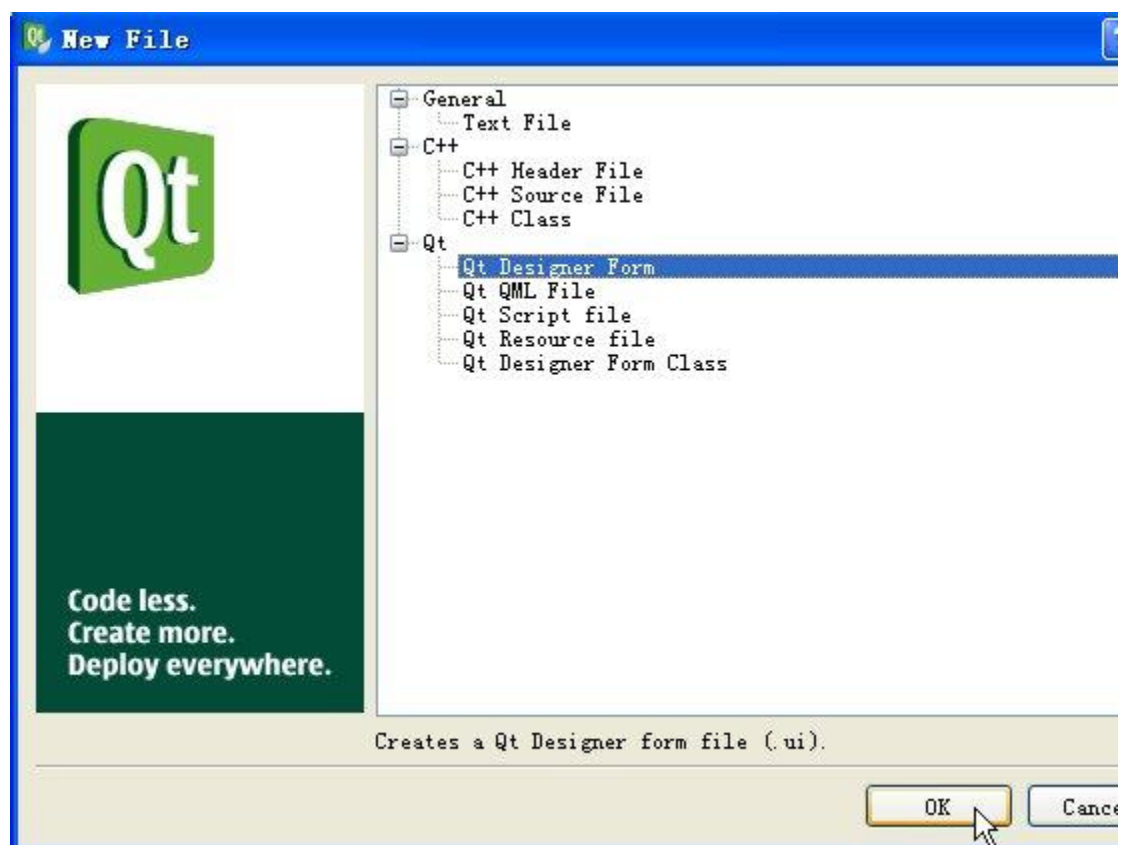


```
action app(argc,argv);  
  
* dd = new QDialog();  
  
    label = new QLabel(dd);  
    setText("hello world");  
  
    show();  
  
app.exec();
```



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。

Choose a form template

templates\forms

Dialog with Buttons Bottom

Dialog with Buttons Right

Dialog without Buttons

Main Window

Widget

+

Widgets

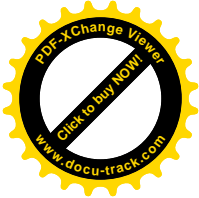
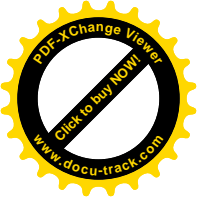
Embedded Design

Device: None

Screen Size: Default size

Next >

3. 你可以更改文件名，我们这里使用默认设置。



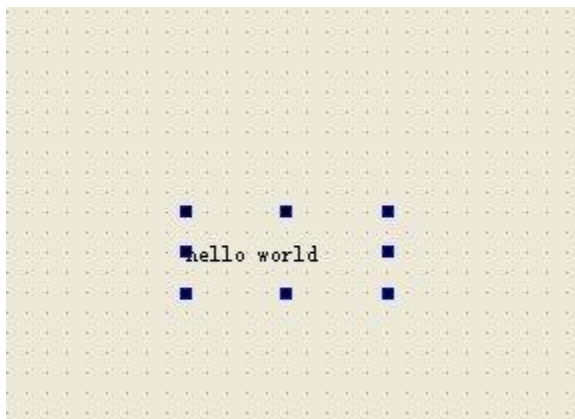
Choose the location

Name:

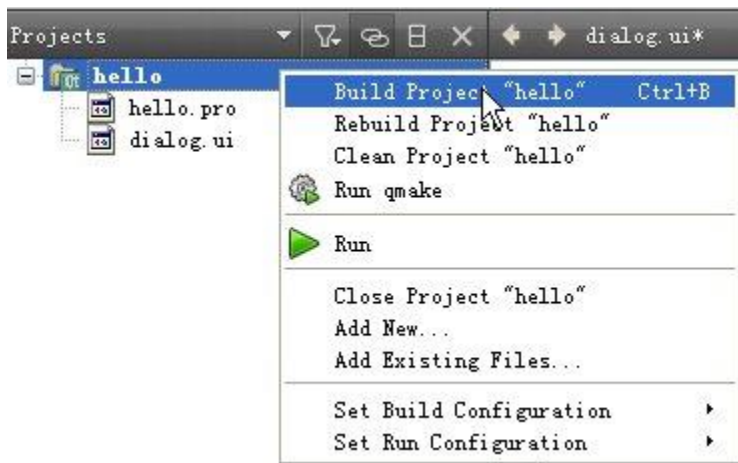
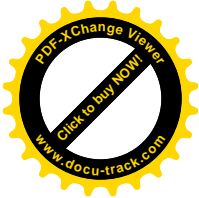
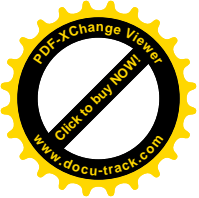
Path:



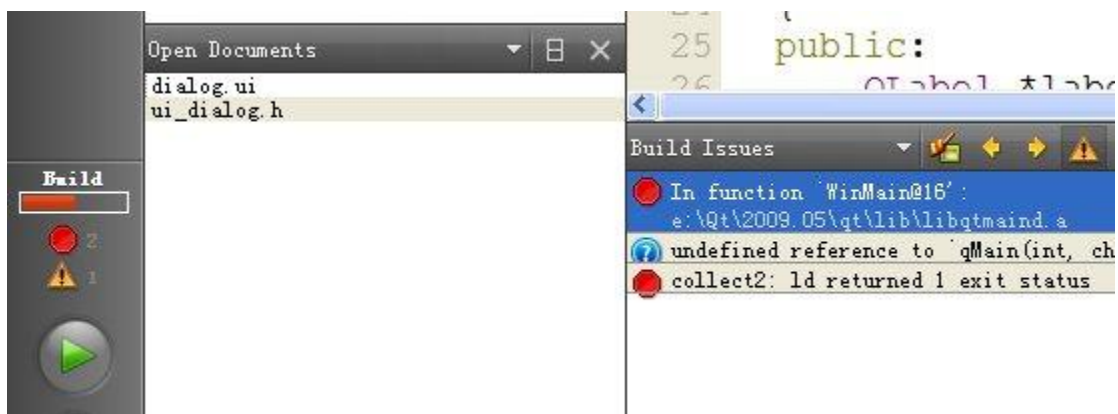
4. 在新建好的框口上添加一个标签，并更改文本为 hello world。




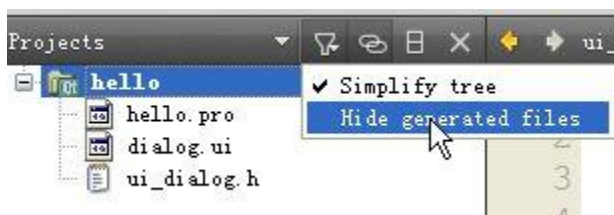
5. 在工程文件夹上点击右键，弹出的菜单中选择第一项编译工程。



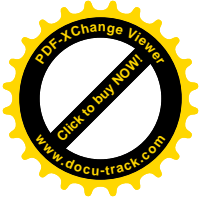
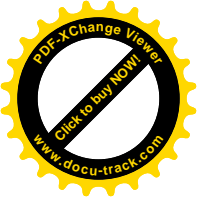
6. 因为还没有写主函数，所以现在编译文件会出现错误，不过没关系，因为我们只是想编译一下 ui 文件。



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

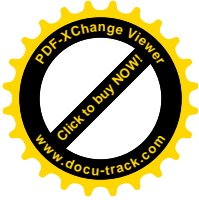
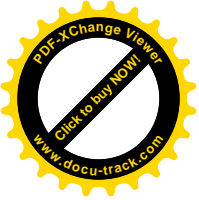


8. ui 文件对应的.h 文件默认为 ui_dialog.h（例如 form.ui 对应 ui_form.h）。

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。

```
projects | ui_dialog.h | <Select Symbol>
hello
  hello.pro
  dialog.ui
  ui_dialog.h
1  /*****
2  ** Form generated from
3  **
4  ** Created: Sun Jan 17
5  **      by: Qt User In
6  **
7  ** WARNING! All changes
8  *****/
9
10 #ifndef UI_DIALOG_H
11 #define UI_DIALOG_H
12
13 #include <QtCore/QVariant>
14 #include <QtGui/QAction>
15 #include <QtGui/QApplication>
16 #include <QtGui/QButtonGroup>
17 #include <QtGui/QDialog>
18 #include <QtGui/QHeaderView>
19 #include <QtGui/QLabel>
20
21 QT_BEGIN_NAMESPACE
22
23 class Ui_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(400, 300);
        label = new QLabel(Dialog);
        label->setObjectName(QString::fromUtf8("label"));
        label->setGeometry(QRect(130, 140, 101, 41));

        retranslateUi(Dialog);

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

    void retranslateUi(QDialog *Dialog)
    {
        Dialog->setWindowTitle(QApplication::translate("Dialog", "Dia
        label->setText(QApplication::translate("Dialog", "hello world
    } // retranslateUi

};

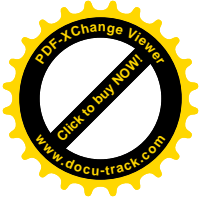
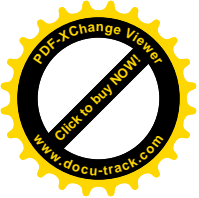
namespace Ui {
    class Dialog: public Ui_Dialog {};
} // namespace Ui
```

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。

```
main.cpp*
1 #include <QtGui>
2 #include "ui_dialog.h"
3
4 int main(int argc, char* argv[])
5 {
6     QApplication app(argc, argv);
7     QDialog* dd = new QDialog();
8     Ui_Dialog *ui = new Ui_Dialog();
9     ui->setupUi(dd);
10    dd->show();
11    return app.exec();
12 }
```

11. 这时运行程序，效果如下。



```
Ui_Dialog *ui = new Ui_Dialog();  
ui->setupUi(dd);  
ui->show();  
return app.exec();
```



在这篇文章中我们一共讲述了一种方法写 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。

二、Qt Creator 编写多窗口程序（原创）

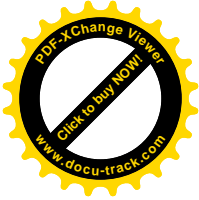
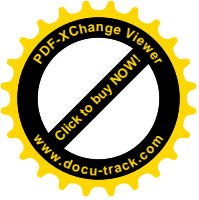
2009-10-18 15:42

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

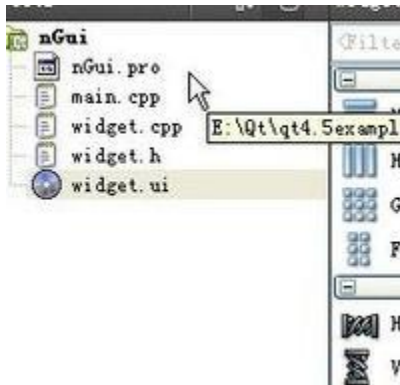
实现原理：



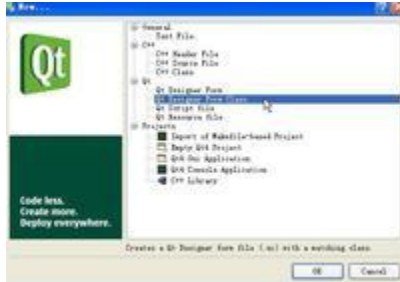
程序里我们先建立一个主工程，作为主界面，然后再建立一个对话框类，将其加入工程中，然后在程序中调用自己新建的对话框类来实现多窗口。

实现过程：

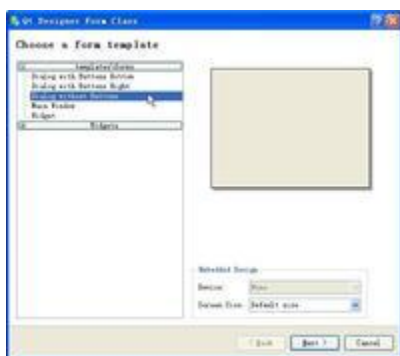
1. 首先新建 Qt4 Gui Application 工程，工程名为 nGui，Base class 选为 QWidget。建立好后工程文件列表如下图。



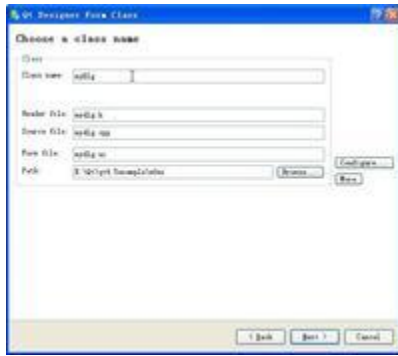
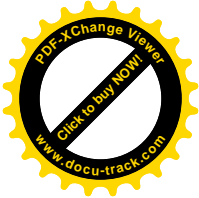
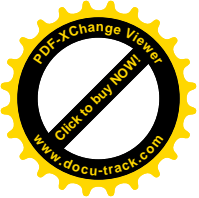
2. 新建对话框类，如下图，在新建中，选择 Qt Designer Form Class。



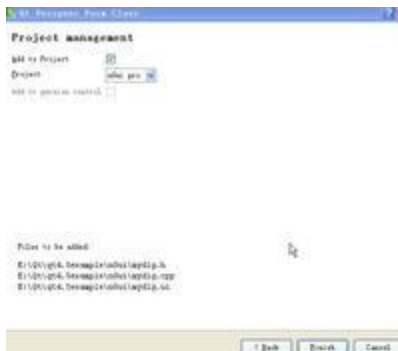
3. 选择 Dialog without Buttons。



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。

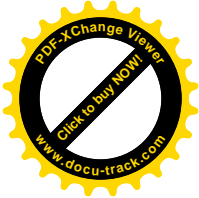
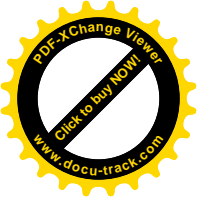


6. 如下图，在 myDlg.ui 中拖入一个 Push Button，将其上的文本改为“进入主窗口”，在其属性窗口中将其 objectName 改为 enterBtn，在下面的 Signals and slots editor 中进行信号和槽的关联，其中，Sender 设为 enterBtn，Signal 设为 clicked()，Receive 设为 myDlg，Slot 设为 accept()。这样就实现了单击这个按钮使这个对话框关闭并发出 Accepted 信号的功能。下面我们将利用这个信号。



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)
    {

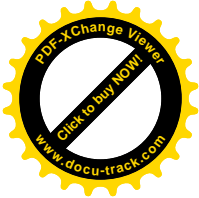
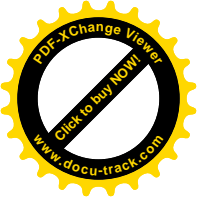
        Widget w;

        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: ld 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;

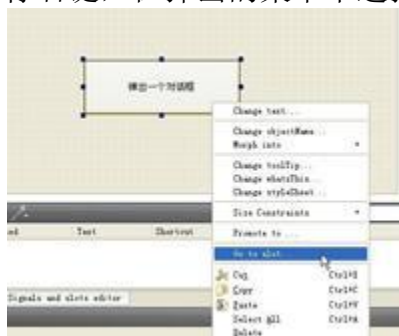
    if(my1.exec()==QDialog::Accepted)
    {
        w.show();
    }
    return a.exec();
}
```

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: ld 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。

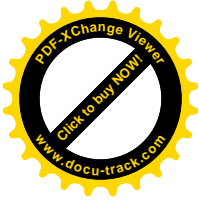
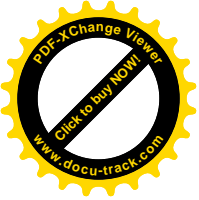
因为 `return a.exec();` 一句表示只要主窗口界面不退出，那么程序就会一直执行。所以只有用第一种方法，将该语句也放到 if 语句中，而在 else 语句中用 `else return 0;`，这样如果 enterBtn 没有被按下，那么程序就会结束执行了。

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();
```

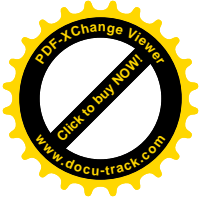
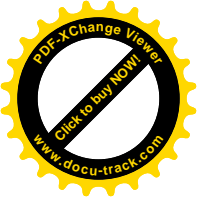
my2 为我们新建对话框类的另一个对象，但是 my2 我们还没有定义，所以在 widget.h 文件中添加相应代码，如下，先加入头文件，再加入 my2 的定义语句，这里我们将其放到 private 里，因为一般的函数都放在 public 里，而变量都放在 private 里。

```
#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。



三、Qt Creator 登录对话框（原创）

2009-10-18 18:38

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

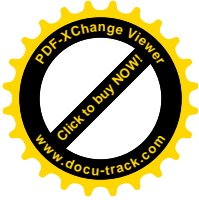
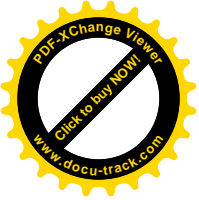
实现过程：

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：

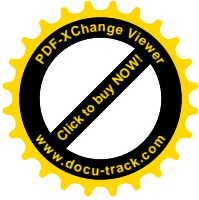
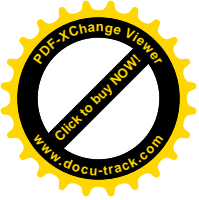


5. 右击登录按钮选择 go to slot，再选择 clicked()，然后进入其单击事件的槽函数，写入一句

```
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。

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。

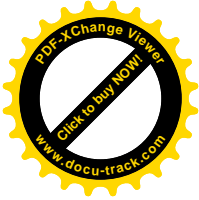
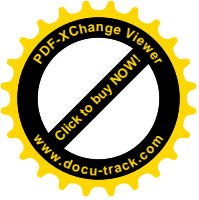


如果输入错误，便会弹出警告提示框：



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。



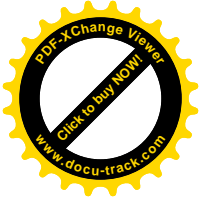
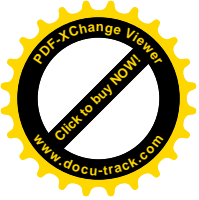
我们可以更改 if 判断语句，使这样的输入也算正确。

```
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，我们希望能使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：



四、Qt Creator 添加菜单图标（原创）

2009-10-27 21:22

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux>
转载请注明出处。

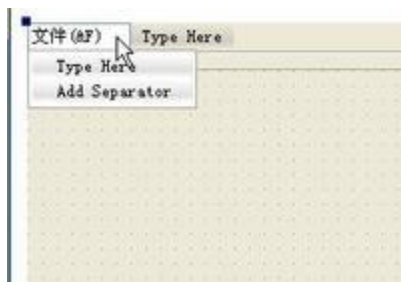
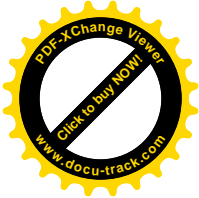
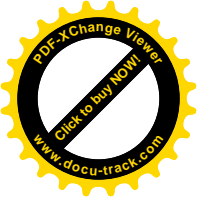
在下面的几节，我们讲述 Qt 的 MainWindow 主窗口部件。这一节只讲述怎样在其上的菜单栏里添加菜单和图标。

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）



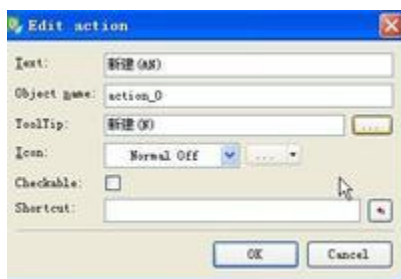
3. 输入完按下 Enter 键确认即可，然后在子菜单中加入“新建(&N)”，确定后，效果如下图。



4. 我们在下面的动作编辑窗口可以看到新加的“新建”菜单。



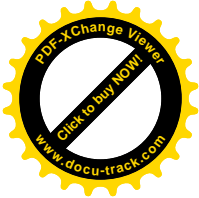
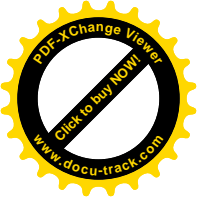
5. 双击这一条，可打开它的编辑对话框。我们看到 Icon 项，这里可以更改“新建”菜单的图标。



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。



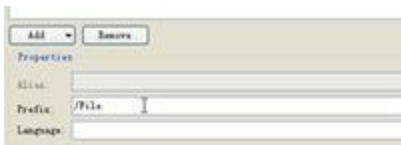
8. 添加完后如下图。可以看到添加的文件为 menu.qrc。



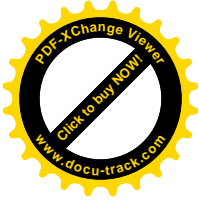
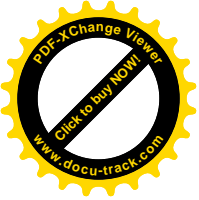
9. 我们最好先在工程文件夹里新建一个文件夹，如 images，然后将需要的图标文件放到其中。



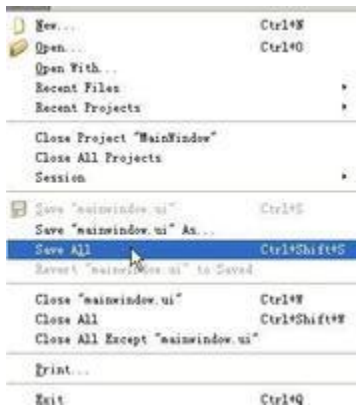
10. 在 Qt Creator 的 menu.qrc 文件中，我们点击 Add 下拉框，选择 Add Prefix。我们可以将生成的/new/prefix 前缀改为其他名字，如/File。



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。



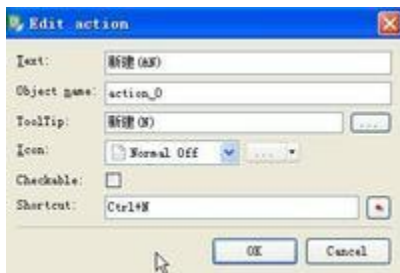
12. 添加完成后，我们在 Qt Creator 的 File 菜单里选择 Save All 选项，保存所做的更改。（注意：一定要先保存刚才的 qrc 文件，不然在资源管理器中可能看不见自己添加的资源！）



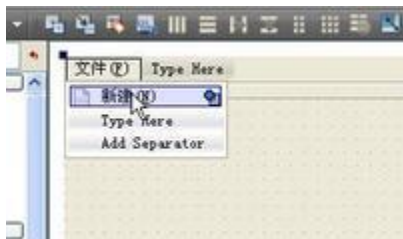
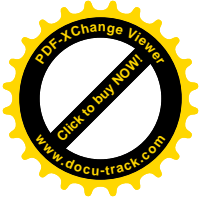
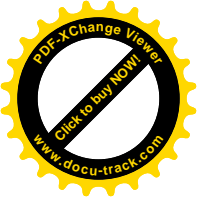
13. 这时再打开资源选择器，可以看到我们的图标都在这里了。（注意：如果不显示，可以按一下上面的 Reload 按钮）



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。



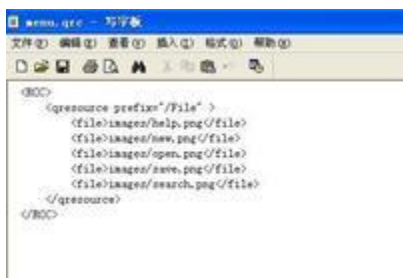
运行程序后效果如下。



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。

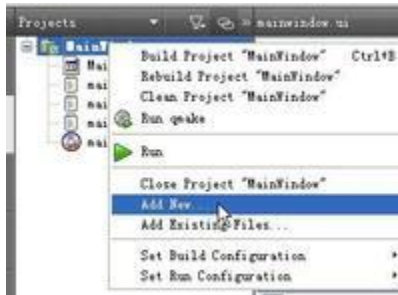
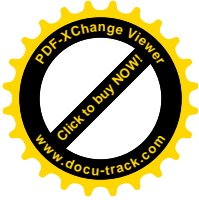
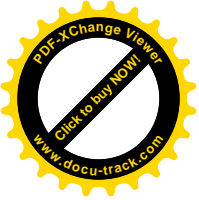


其具体内容如下。

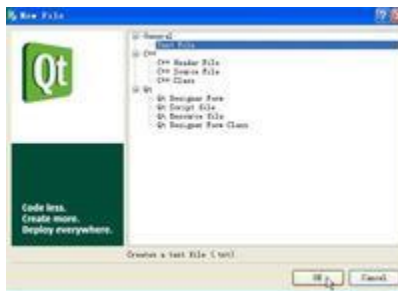


附：第二种添加资源文件的方法。

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。

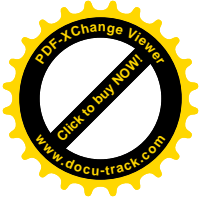
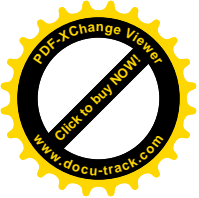


4. 添加好文件后将其内容修改如下。可以看到就是用第一种方法生成的 menu.qrc 文件的内容。



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（原创）

2009-10-29 12:22

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

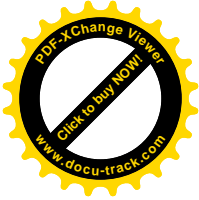
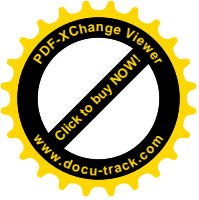
“文件”的子菜单如下图。中间的分割线可以点击 Add Separator 添加。



“编辑”子菜单的内容如下。



“帮助”子菜单的内容如下。



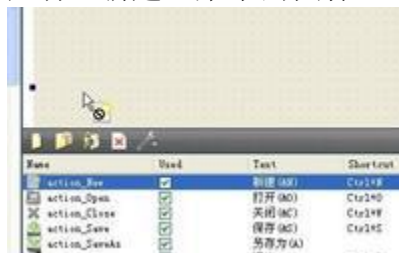
2. 我们在动作编辑器中对各个菜单的属性进行设置。

如下图。

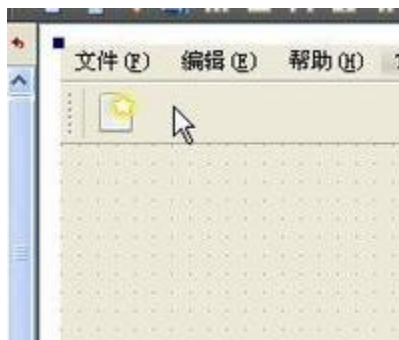


3. 我们拖动“新建”菜单的图标，将其放到工具栏里。

拖动“新建”菜单的图标。

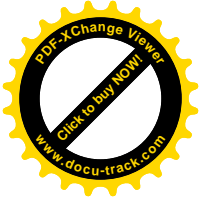
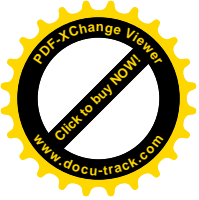


将其放到菜单栏下面的工具栏里。



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





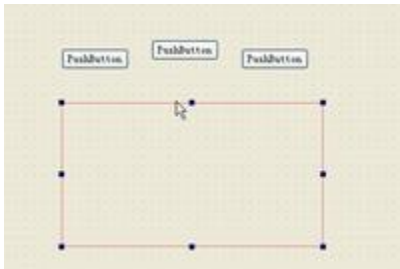
5. 最终效果如下。如果需要删除图标，可以在图标上点击右键选择 Remove action 即可。



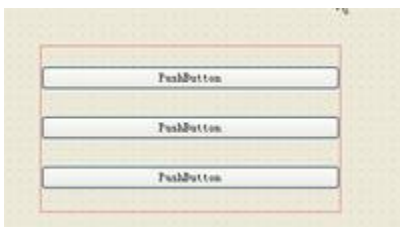
下面简述一下布局管理器。

（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。）

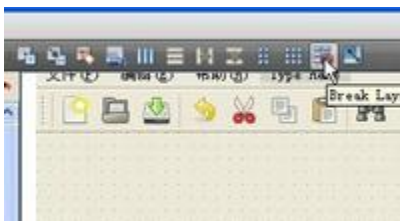
1. 在左边的器件栏里拖入三个 PushButton 和一个 Vertical Layout（垂直布局管理器）到中心面板。如下图。



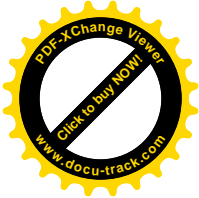
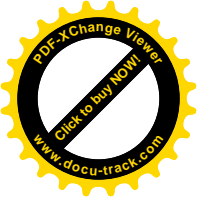
2. 将这三个按钮放入垂直布局管理器，效果如下。可以看到按钮垂直方向排列，并且宽度可以改变，但高度没有改变。



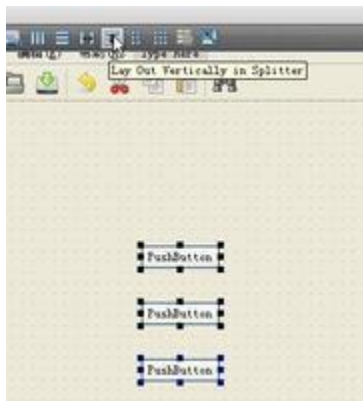
3. 我们将布局管理器整体选中，按下上面工具栏的 Break Layout 按钮，便可取消布局管理器。（我们当然也可以先将按钮移出，再按下 Delete 键将布局管理器删除。）



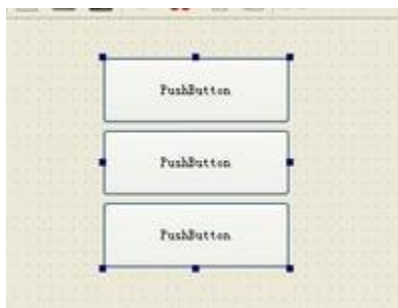
4. 下面我们改用分裂器部件（QSplitter）。



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）。

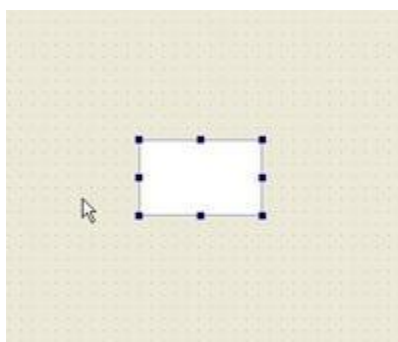


效果如下图。可以看到按钮的大小可以随之改动。这也就是分裂器和布局管理器的分别。

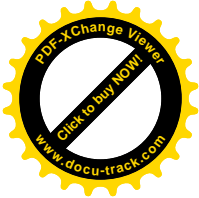
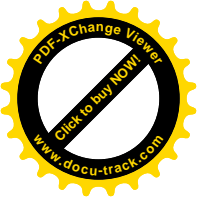


5. 其实布局管理器不但能控制器件的布局，还有个很重要的用途是，它能使器件的大小随着窗口大小的改变而改变。

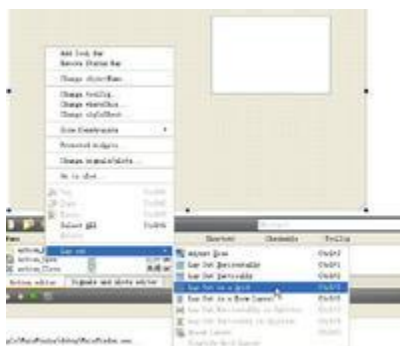
我们先在主窗口的中心拖入一个文本编辑器 Text Edit。



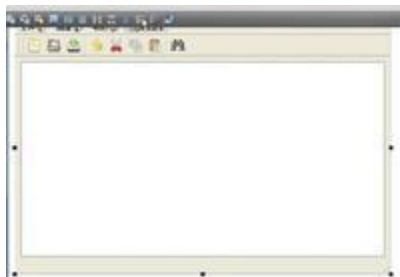
这时直接运行程序，效果如下。可以看到它的大小和位置不会随着窗口改变。



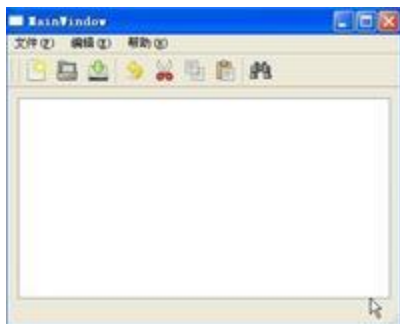
下面我们选中主窗口部件，然后在空白处点击鼠标右键，选择 Layout->Lay Out in a Grid，使整个主窗口的中心区处于网格布局管理器中。



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。



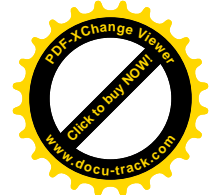
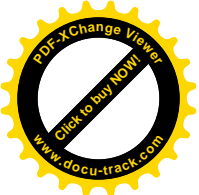
运行一下程序，可以看到无论怎样拉伸窗口，文本编辑框的大小都会随之改变。



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

<http://www.qtcn.org/bbs/read.php?tid=23252&page=1&toread=1> 下载到。



六、Qt Creator 实现文本编辑（原创）

2009-10-31 22:35

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

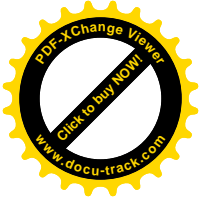
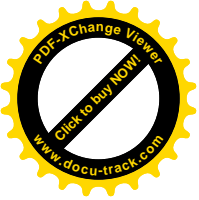
1. 打开 Qt Creator，在 File 菜单中选择 Open，然后在工程文件夹中打开 MainWindow.pro 工程文件。

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。



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。

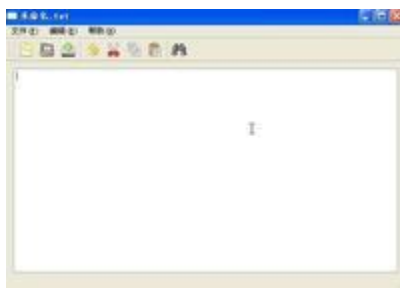
3. 在 `mainwindow.cpp` 中先加入头文件 `#include <QtGui>`，然后在构造函数里添加以下几行代码。

```
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
```

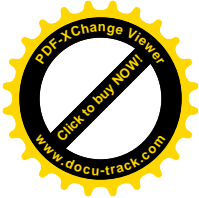
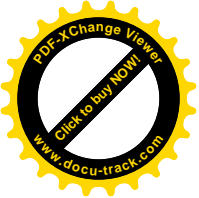
这是对主窗口进行初始化。效果如下。



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;
```

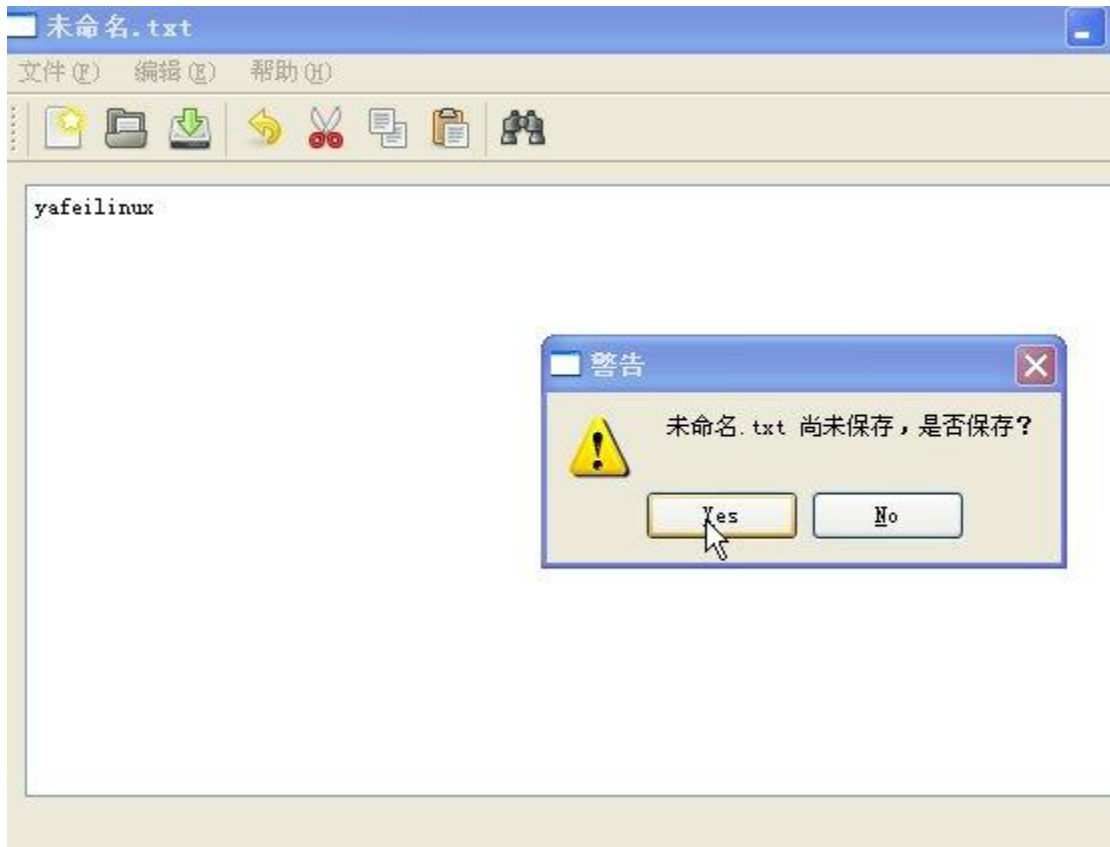
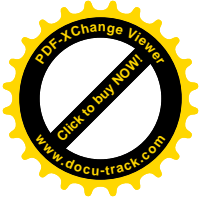
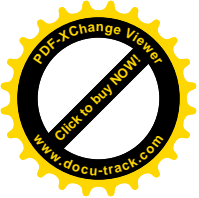
```
curFile = tr("未命名.txt");  
setWindowTitle(curFile);  
ui->textEdit->clear(); //清空文本编辑器  
ui->textEdit->setVisible(true); //文本编辑器可见  
}
```

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。



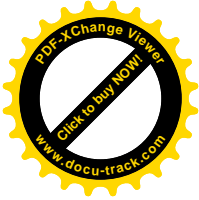
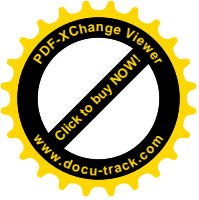
6. 再添加“保存”操作的函数定义。

```
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
    }
}
```

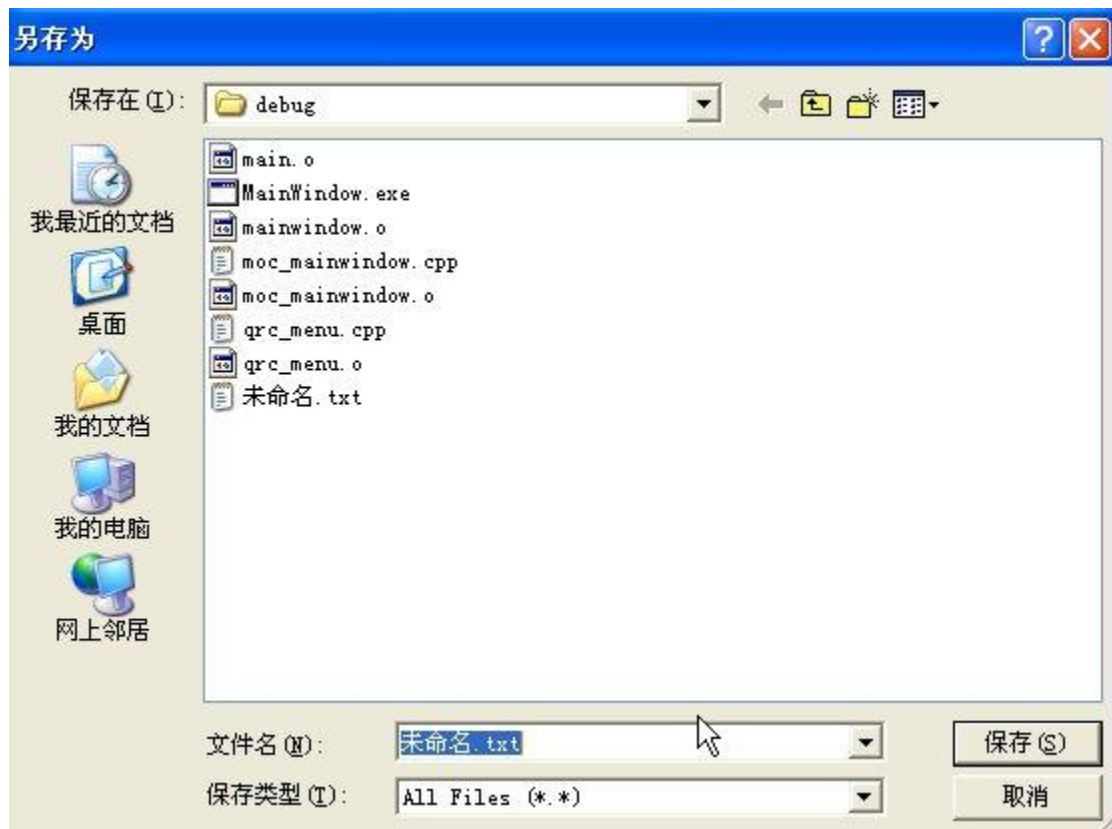
对文件进行保存时，先判断其是否已经被保存过，如果没有被保存过，就要先对其进行另存为操作。

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);
    }
}
```

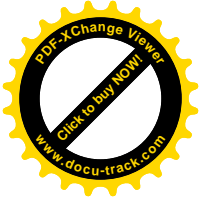
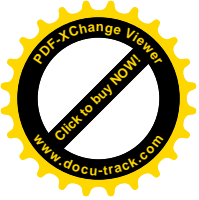


这里弹出一个文件对话框，显示文件另存为的路径。



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::canonicalFilePath(fileName); //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;
}
```

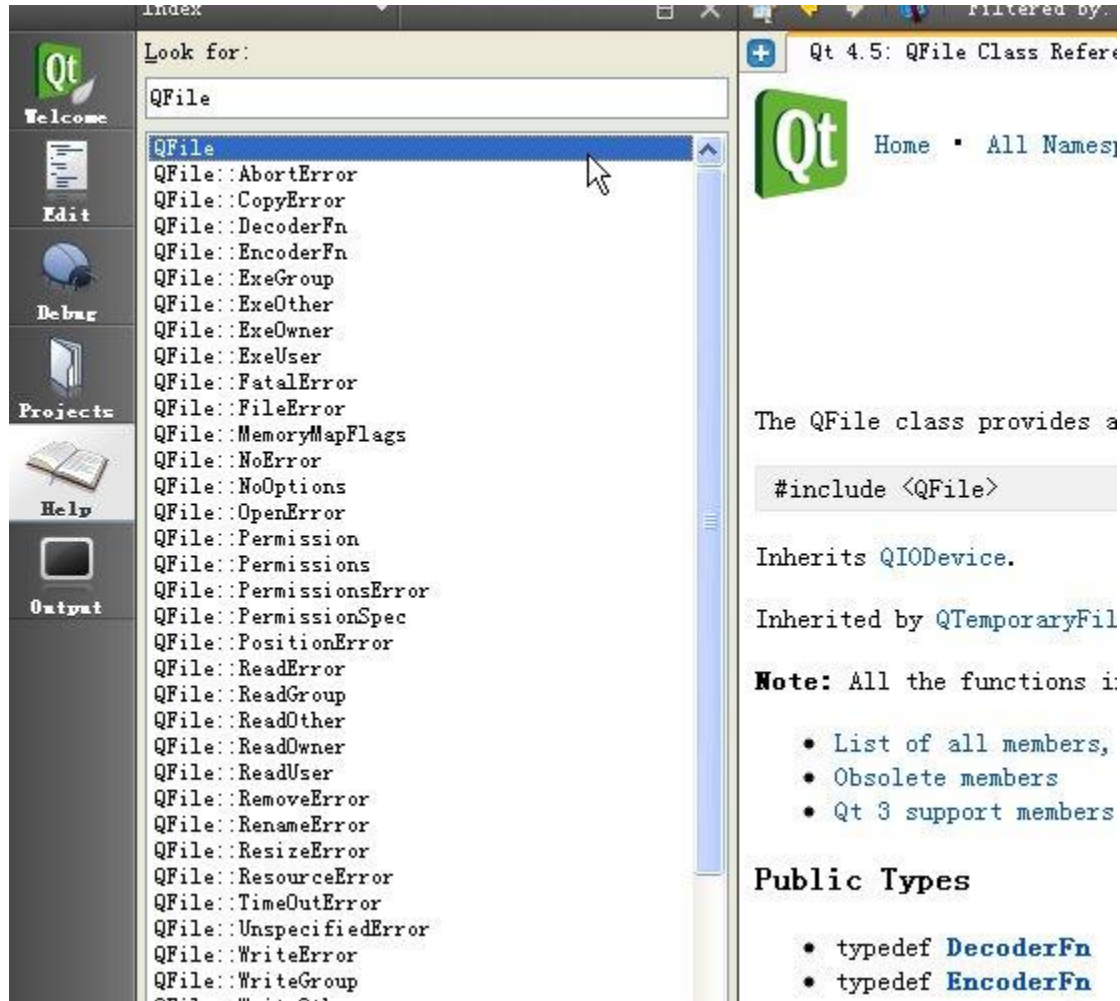


}

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。

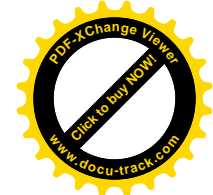
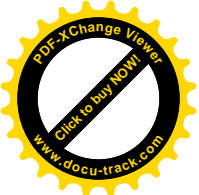
点击 Qt Creator 最左侧的 Help，在其中输入 `QFile`，在搜索到的列表中选择 `QFile` 即可。这时在右侧会显示出 `QFile` 类中所有相关信息以及他们的用法和说明。



//

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 current position in the file by `pos()`. When 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 line endings to Unix. If you don't pass this flag, `QFile` assumes binary, i.e. it doesn't perform any conversion on the data.

//

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";
```

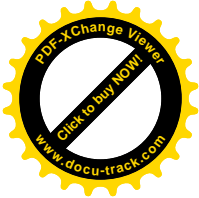
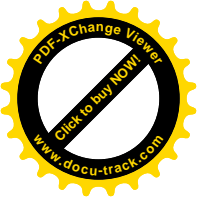
`QDataStream` is similar, in that you can use operator `<<()` to write data and

When you use `QFile`, `QFileInfo`, and `QDir` to access the file system with Qt, you can specify the encoding to use. By default, Qt uses UTF-8 encoding. If you want to use standard C++ APIs (`<cstdio>` or `<iostream>`), you can use the `encodeName()` and `decodeName()` functions to convert between Uni-

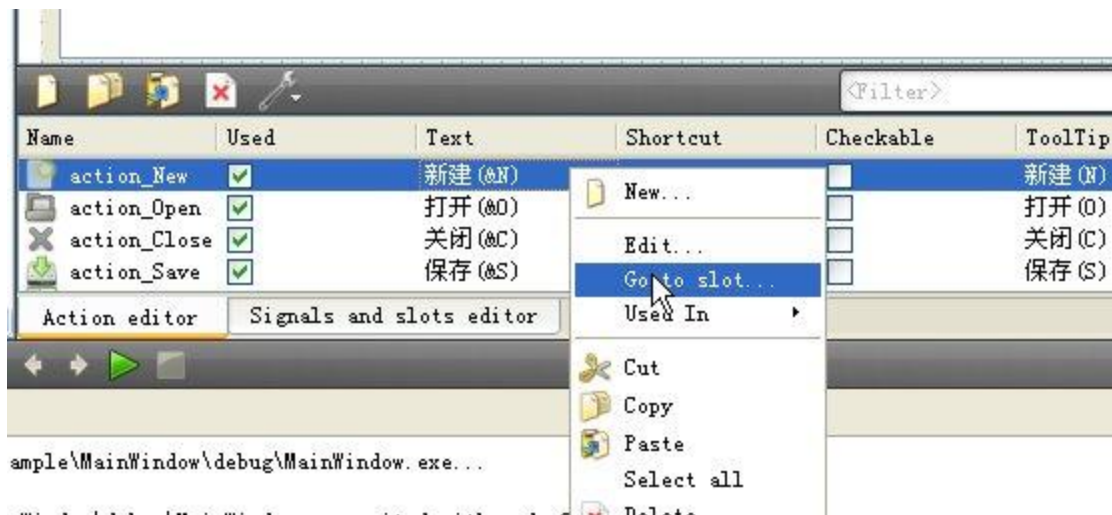
上面只是做了一个简单的说明。以后我们对自己不明白的类都可以去帮助里进行查找，这也许是我们以后要做的最多的一件事了。对于其中的英文解释，我们最好想办法弄明白它的大意，其实网上也有一些中文的翻译，但最好还是从一开始就尝试着看英文原版的帮助，这样以后才不会对中文翻译产生依赖。

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()

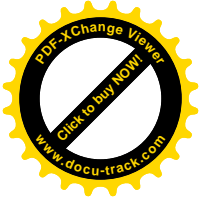
```
{
do_file_Save();
}
```

void MainWindow::on_action_SaveAs_triggered()

```
{
do_file_SaveAs();
}
```

最终的 mainwindow.cpp 文件如下。





```

56 @ void MailWindow::on_action_new_triggered() // 发送邮件按钮
57 {
58     do_file_new();
59 }
60
61 @ void MailWindow::on_action_save_triggered()
62 {
63     do_file_save();
64 }
65
66 @ void MailWindow::on_action_refresh_triggered()
67 {
68     do_file_refresh();
69 }

```

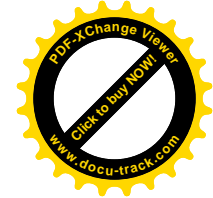
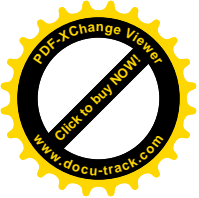
[illegible]

先备份上次的工程文件，然后再将其打开。

```
void do file Open(); //打开文件
```

2. 再在 `mainwindow.cpp` 文件中写函数的功能实现。

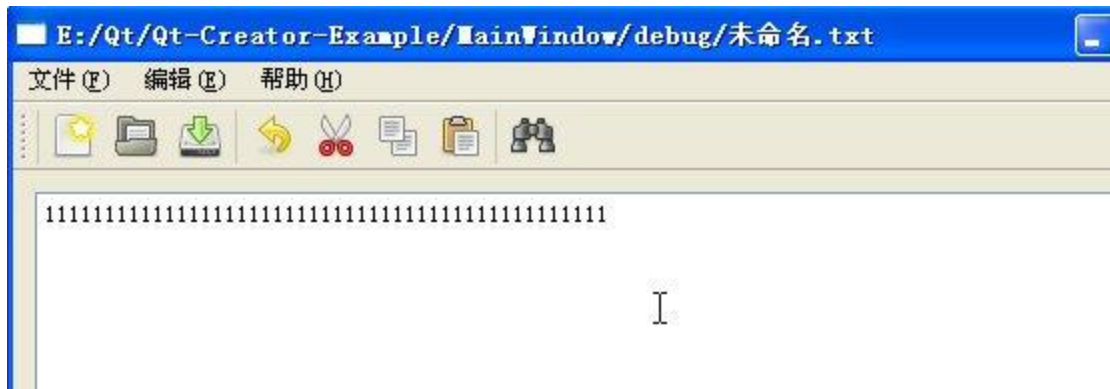
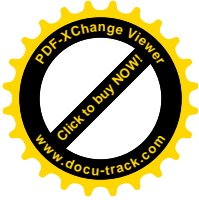
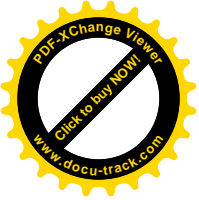
```
{
do_file_SaveOrNot();//是否需要保存现有文件
QString fileName = QFileDialog::getOpenFileName(this);
//获得要打开的文件的名字
if(!fileName.isEmpty())//如果文件名不为空
{
do_file_Load(fileName);
}
```



```
}  
ui->textEdit->setVisible(true); //文本编辑器可见  
}
```



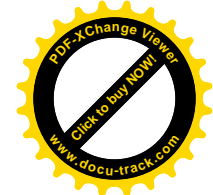
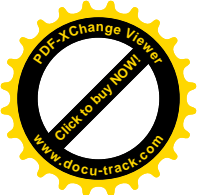
```
bool MainWindow::do_file_Load(const QString& fileName) //读取文件  
{  
    QFile file(fileName);  
    if(!file.open(QFile::ReadOnly | QFile::Text))  
    {  
        QMessageBox::warning(this, tr("读取文件"), tr("无法读取文件 %1:\n%2.").arg(fileName).arg(file.errorString()));  
        return false; //如果打开文件失败，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();  
}
```

```
116 void MainWindow::on_action_Open_triggered() //打开文件  
117 {  
118     //打开文件对话框  
119     QString fileName = QFileDialog::getOpenFileName(this,  
120     //选择要打开的文件的名字  
121     //设置文件名字  
122     ui->textEdit->fileName(),  
123     //设置要打开的文件类型  
124     QString::fromLocal8Bit("*.txt"));  
125     if (fileName.isEmpty())  
126         return;  
127     QFile file(fileName);  
128     if (!file.exists())  
129         return;  
130     if (!file.open(QIODevice::ReadOnly | QIODevice::Text))  
131         return;  
132     QTextStream in(&file);  
133     QString content = in.readAll();  
134     ui->textEdit->setText(content);  
135     in.close();  
136 }  
137  
138 void MainWindow::on_action_Close_triggered()  
139 {  
140     do_file_SaveDialog();  
141     ui->textEdit->clear();  
142 }  
143  
144 void MainWindow::on_action_Quit_triggered() //退出操作  
145 {  
146     on_action_Close_triggered(); //先执行关闭操作  
147     QApplication::quit(); //再退出系统，qApp是指向应用程序的主指针  
148 }  
149  
150 void MainWindow::on_action_undo_triggered()  
151 {  
152     ui->textEdit->undo();  
153 }  
154  
155 void MainWindow::on_action_redo_triggered()  
156 {  
157     ui->textEdit->redo();  
158 }  
159  
160 void MainWindow::on_action_Copy_triggered()  
161 {  
162     ui->textEdit->copy();  
163 }  
164  
165 void MainWindow::on_action_Past_triggered()  
166 {  
167     ui->textEdit->paste();  
168 }
```

因为复制，撤销，全选，粘贴，剪切等功能，是 QTextEdit 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。

七、Qt Creator 实现文本查找（原创）

2009-11-06 21:46

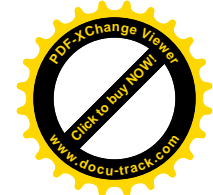
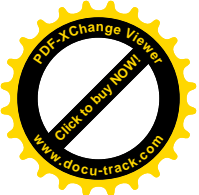
声明：本文原创于 yafeilinux 的 百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

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。

```
1 // MainWindow::on_action_Find_triggered()
2
3 QDialog *findDlg = new QDialog(this);
4 //新建一个对话框，用于查找操作，this 表明它的父窗口是 MainWindow。
5 findDlg->setWindowTitle(tr("查找"));
6 //设置对话框的标题
7 QLineEdit *find_textLineEdit = new QLineEdit(findDlg);
8 //将行编辑器加入到新建的查找对话框中
9 QPushButton *find_Btn = new QPushButton(tr("查找下一个"), findDlg);
10 //加入一个“查找下一个”的按钮
11 QVBoxLayout* layout = new QVBoxLayout(findDlg);
12 layout->addWidget(find_textLineEdit);
13 layout->addWidget(find_Btn);
14 //新建一个垂直布局管理器，并将行编辑器和按钮加入其中
15 findDlg->show();
16 //显示对话框
17 connect(find_Btn, SIGNAL(clicked()), this, SLOT(show_findText()));
18 //设置“查找下一个”按钮的单击事件和其槽函数的关联
```

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();`

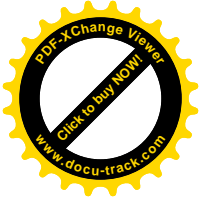
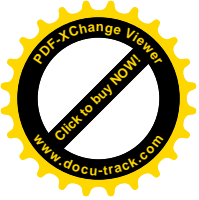
在该函数中实现查找字符串的功能。

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

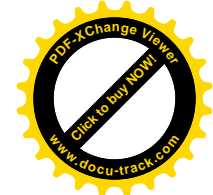
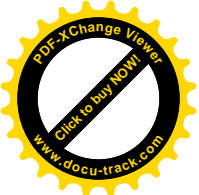
```
{  
    QString findText = find_textLineEdit->text();  
    //获取行编辑器中的内容  
}
```

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。

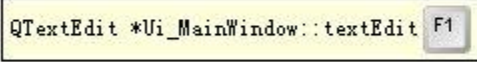
6. 在下一行写下 `ui`，然后直接按下键盘上的“<.”键，这时系统会根据是否是指针对象而自动生成“->”或“.”，因为 `ui` 是指针对象，所以自动生成“->”号，而且弹出了 `ui` 中的所有部件名称的列表。如下图。



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 }
196
197
198
```

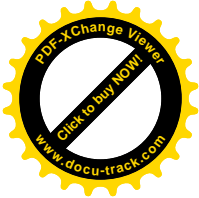
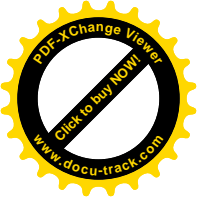


8. 如上图我们将鼠标放到 textEdit 上，这时便出现了 textEdit 的类名信息，且后面出现一个 F1 按键。我们按下键盘上的 F1，便能出现 textEdit 的帮助。



```
mainwindow.cpp*  MainWindow::show_findText()  Line: 194, Col: 17
178     QPushButton *find_Btn = new QPushButton(tr("查找下一个"));
179     //加入一个“查找下一个”的按钮
180     QVBoxLayout* layout = new QVBoxLayout(findDlg);
181     layout->addWidget(find_textLineEdit);
182     layout->addWidget(find_Btn);
183     //新建一个垂直布局管理器，并将行编辑器和按钮加入其中
184     findDlg ->show();
185     //显示对话框
186     connect(find_Btn,SIGNAL(clicked()),this,SLOT(show_findText());
187     //设置“查找下一个”按钮的单击事件和其槽函数的关联
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
```

9. 我们在帮助中向下拉，会发现这里有一个 find 函数。



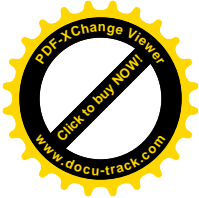
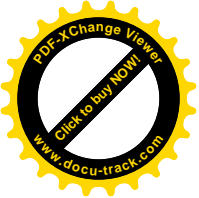
- QString **documentTitle** () const
- void **ensureCursorVisible** ()
- QList<ExtraSelection> **extraSelections** () const
- bool **find** (const QString & *exp*, QTextDocument::FindFlags *options* = 0)
- QString **fontFamily** () const
- bool **fontItalic** () const
- qreal **fontPointSize** () const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

10. 我们点击 find，查看其详细说明。

```
Go to Help Mode  ◆ ◆
bool QTextEdit::find ( const QString & exp,
    QTextDocument::FindFlags options = 0 )

Finds the next occurrence of the string, exp, using the given
options. Returns true if exp was found and changes the cursor to
select the match; otherwise returns false.
```

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。



Go to Help Mode

enum QTextDocument::FindFlag
flags QTextDocument::FindFlags

This enum describes the options available to `QTextDocument`'s find function. The options can be OR-ed together from the following list:

Constant	Value	Description
<code>QTextDocument::FindBackward</code>	<code>0x00001</code>	Search backwards instead of forwards.
<code>QTextDocument::FindCaseSensitively</code>	<code>0x00002</code>	By default find works case insensitive. Specifying this option changes the behaviour to a case sensitive find operation.
<code>QTextDocument::FindWholeWords</code>	<code>0x00004</code>	Makes find match only complete words.

The `FindFlags` type is a typedef for `QFlags<FindFlag>`. It stores an OR combination of `FindFlag` values.

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->te
93     //获取行编辑器中的内容
94     ui->textEdit->f
95
96
97
```

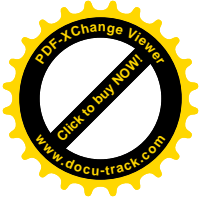
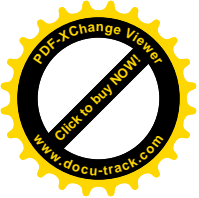
find

findChild

findChildren

focusInEvent

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。



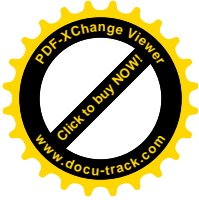
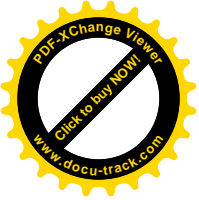
```
void MainWindow::show_findText()//“查找下一个”按钮的槽函数
{
    QString findText = find_textLineEdit->text();
    //获取行编辑器中的内容
    bool find(const QString &exp, QTextDocument::Find
    ui->textEdit->find(
}
```

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。

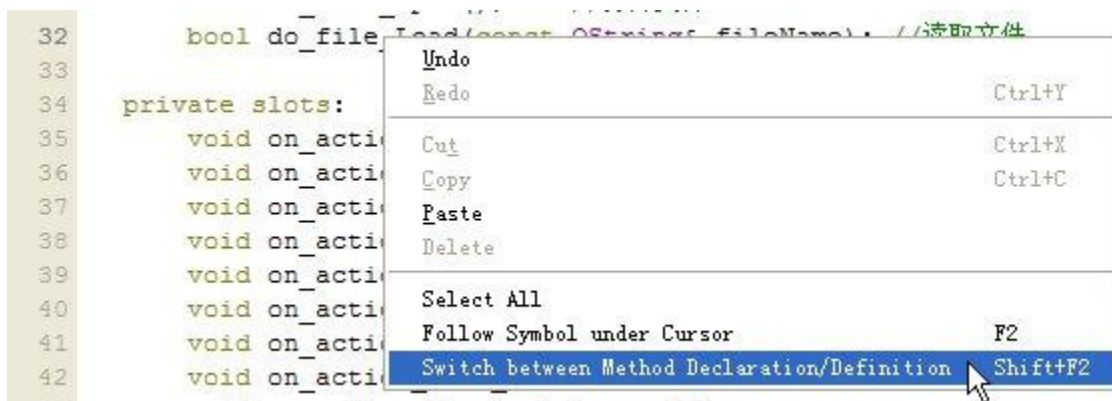
```
void MainWindow::show_findtext() //显示下一个匹配的字符串
{
    QString findtext = find_text_lineedit->text();
    // 获取当前编辑器中的文本
    QTextDocument *pDoc = find_text_lineedit->document();
    QTextCursor cursor(pDoc);
    cursor.movePosition(QTextCursor::Next, Qt::FindText, 1);
    // 将找到的文本高亮显示
    QTextCursor selectCursor(cursor);
    selectCursor.movePosition(QTextCursor::Previous, Qt::FindText, 1);
    pDoc->setCursor(selectCursor);
}
```



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。

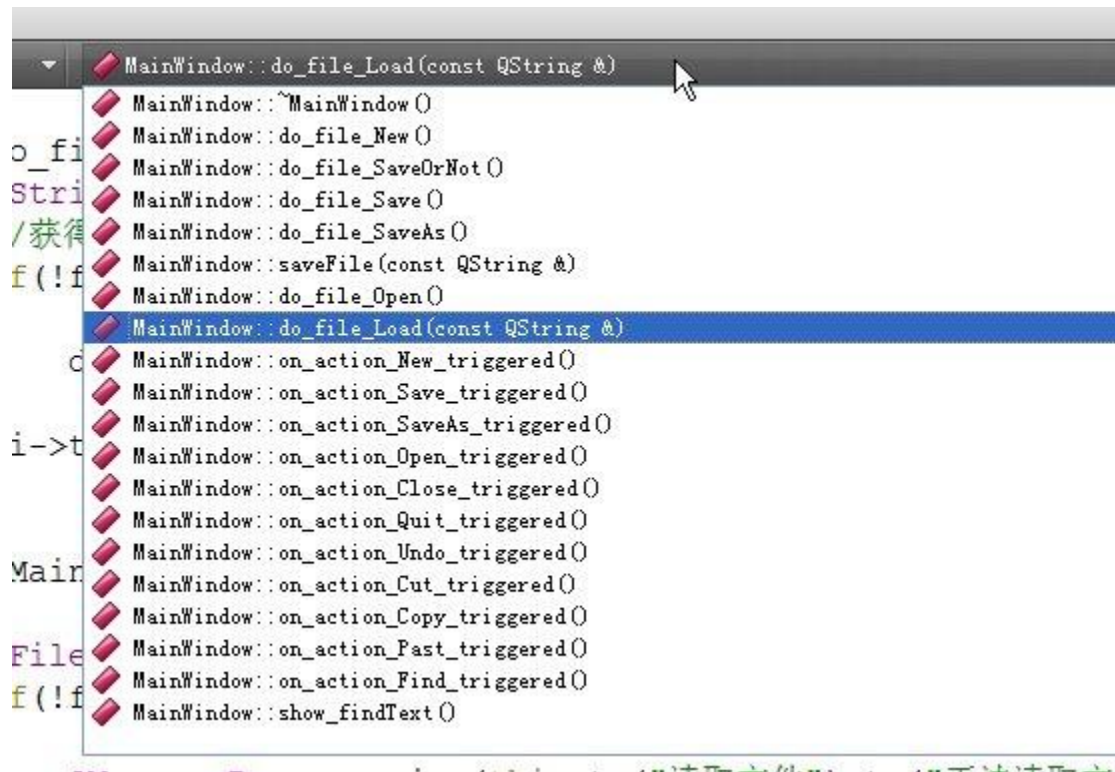
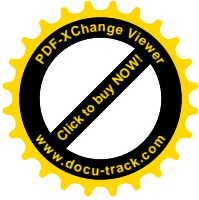
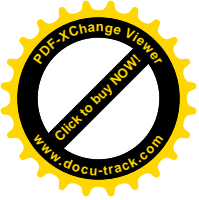


这时系统就会自动跳转到函数定义的位置。如下图。

```
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(
107             .arg(fileName).arg(f
108         return false;
```

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。

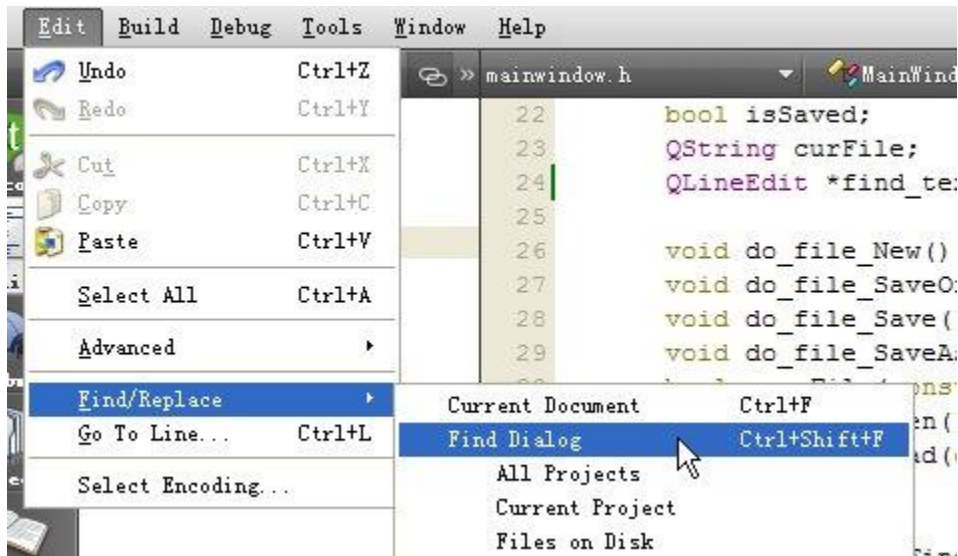
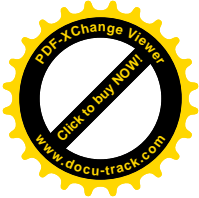
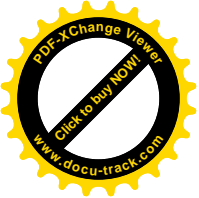


第三，利用查找功能。

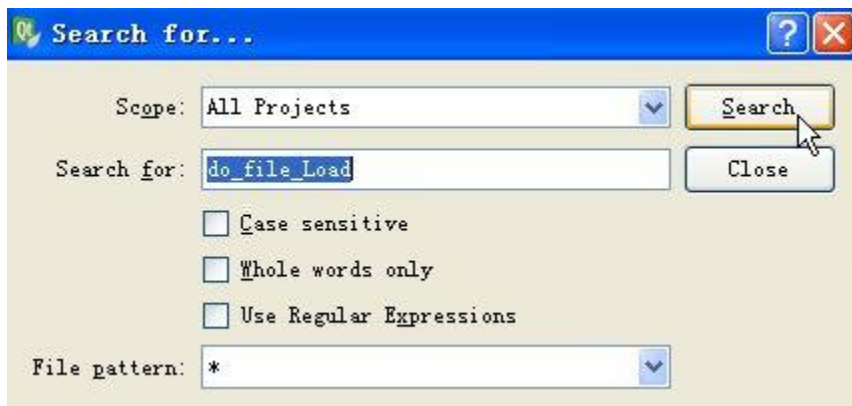
1. 我们先将鼠标定位到一个函数名上。

```
void do_file_SaveAs(); //文件另存为
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();
```

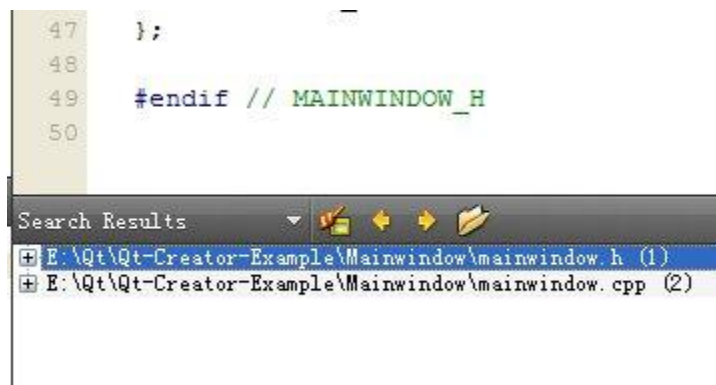
2. 然后选择 Edit->Find/Replace->Find Dialog。



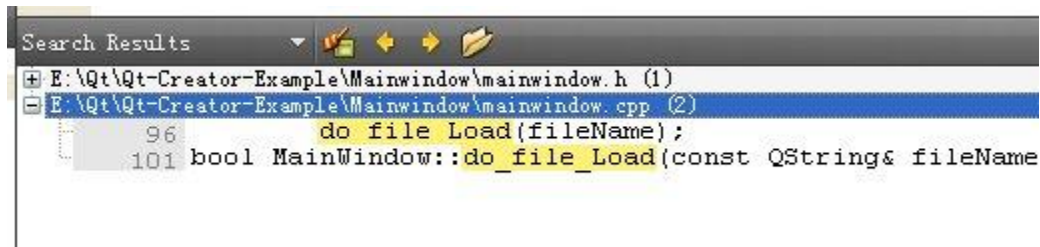
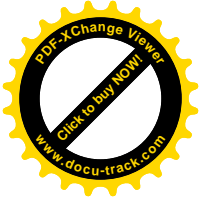
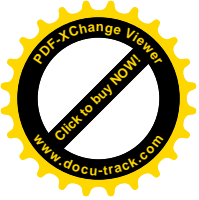
3. 这时会出现一个查找对话框，可以看到要查找的函数名已经写在里面了。



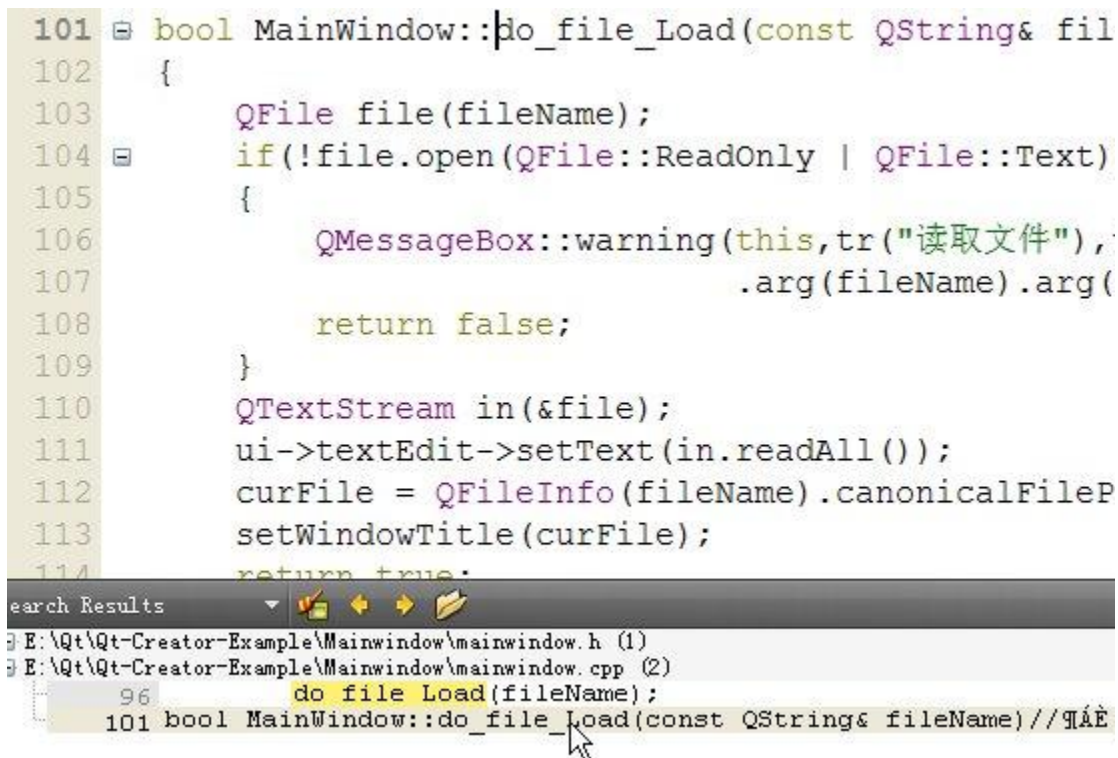
4. 当我们按下 Search 按钮后，会在查找结果窗口显示查找到的结果。



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。

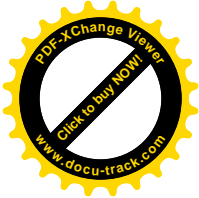
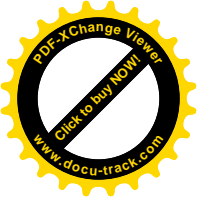
在以后的文章中，我们不会再很详细地去用帮助来说明一个函数是怎么来的，该怎么用，这些应该自己试着去查找。

八、Qt Creator 实现状态栏显示（原创）

2009-11-07 12:01

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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* first_statusLabel; //声明两个标签对象，用于显示状态信息
```

```
QLabel* second_statusLabel;
```

```
void init_statusBar(); //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::init_statusBar()

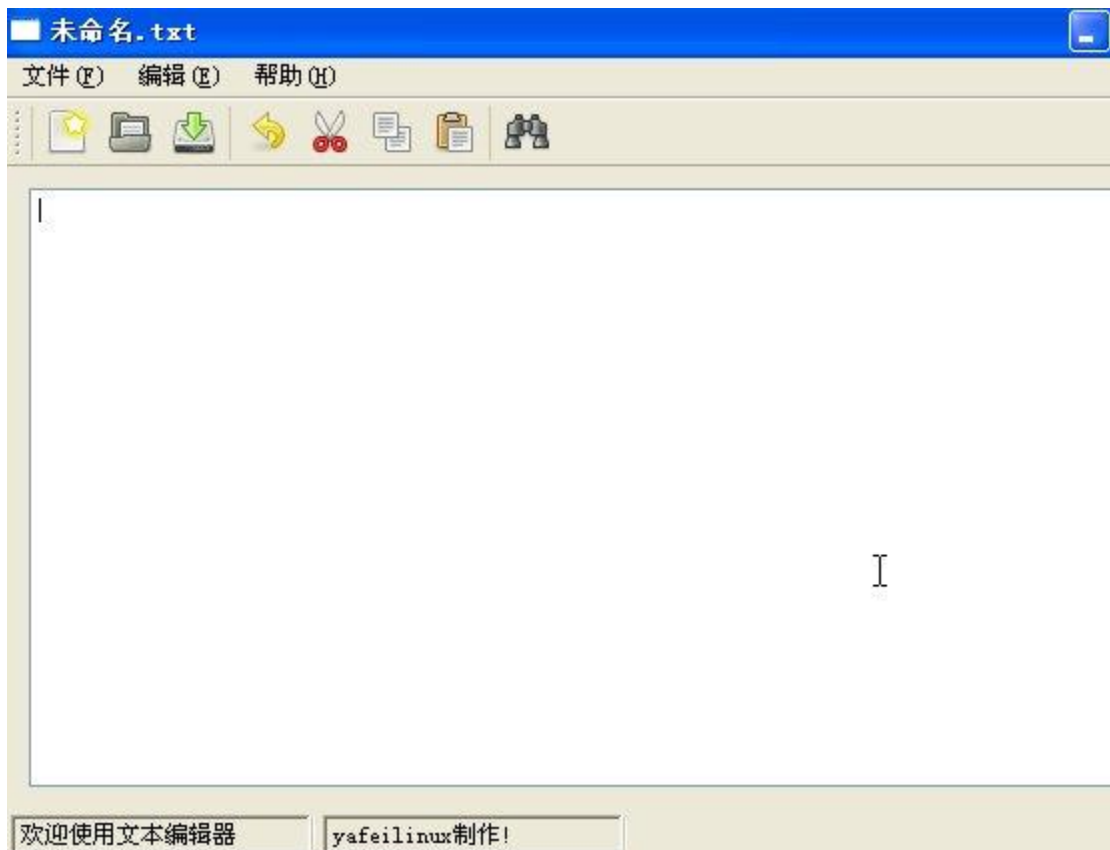
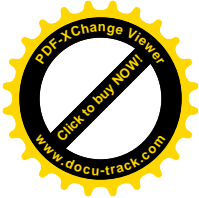
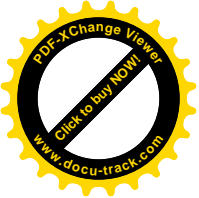
{
    QStatusBar* bar = ui->statusBar; //获取状态栏
    first_statusLabel = new QLabel; //新建标签
    first_statusLabel->setMinimumSize(150, 20); //设置标签最小尺寸
    first_statusLabel->setFrameShape(QFrame::WinPanel); //设置标签形状
    first_statusLabel->setFrameShadow(QFrame::Sunken); //设置标签阴影
    second_statusLabel = new QLabel;
    second_statusLabel->setMinimumSize(150, 20);
    second_statusLabel->setFrameShape(QFrame::WinPanel);
    second_statusLabel->setFrameShadow(QFrame::Sunken);
    bar->addWidget(first_statusLabel);
    bar->addWidget(second_statusLabel);
    first_statusLabel->setText(tr("欢迎使用文本编辑器")); //初始化内容
    second_statusLabel->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。

```
init_statusBar();
```

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()

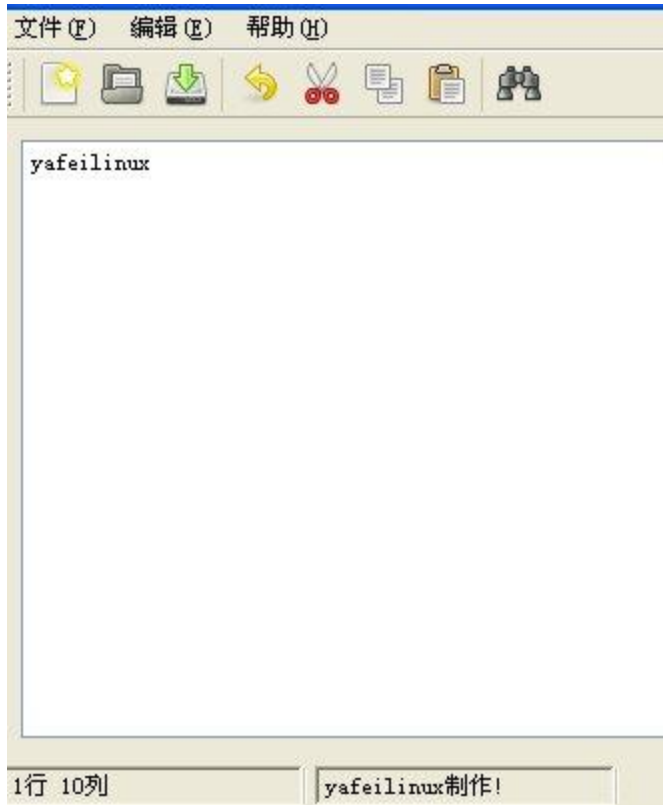
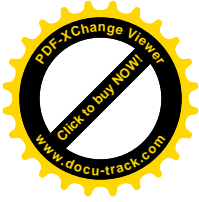
{
    int rowNum = ui->textEdit->document()->blockCount();
    //获取光标所在行的行号
    const QTextCursor cursor = ui->textEdit->textCursor();
    int colNum = cursor.columnNumber();
    //获取光标所在列的列号
    first_statusLabel->setText(tr("%1 行 %2 列").arg(rowNum).arg(colNum));
    //在状态栏显示光标位置
}
```

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。

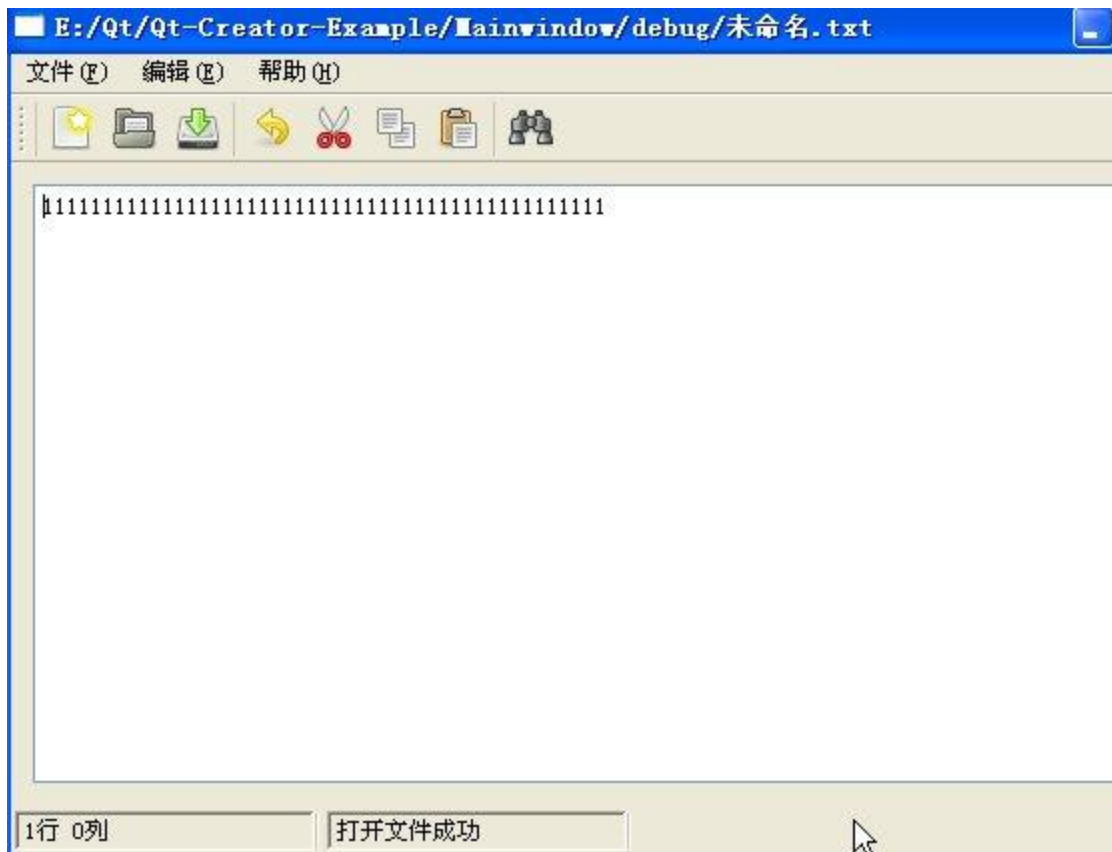
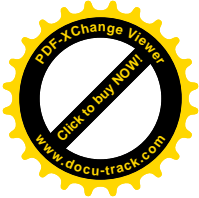
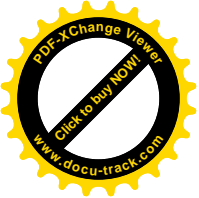
```
connect(ui->textEdit, SIGNAL(cursorPositionChanged()), this, SLOT(do_cursorChanged()));
```

这时运行程序。效果如下。



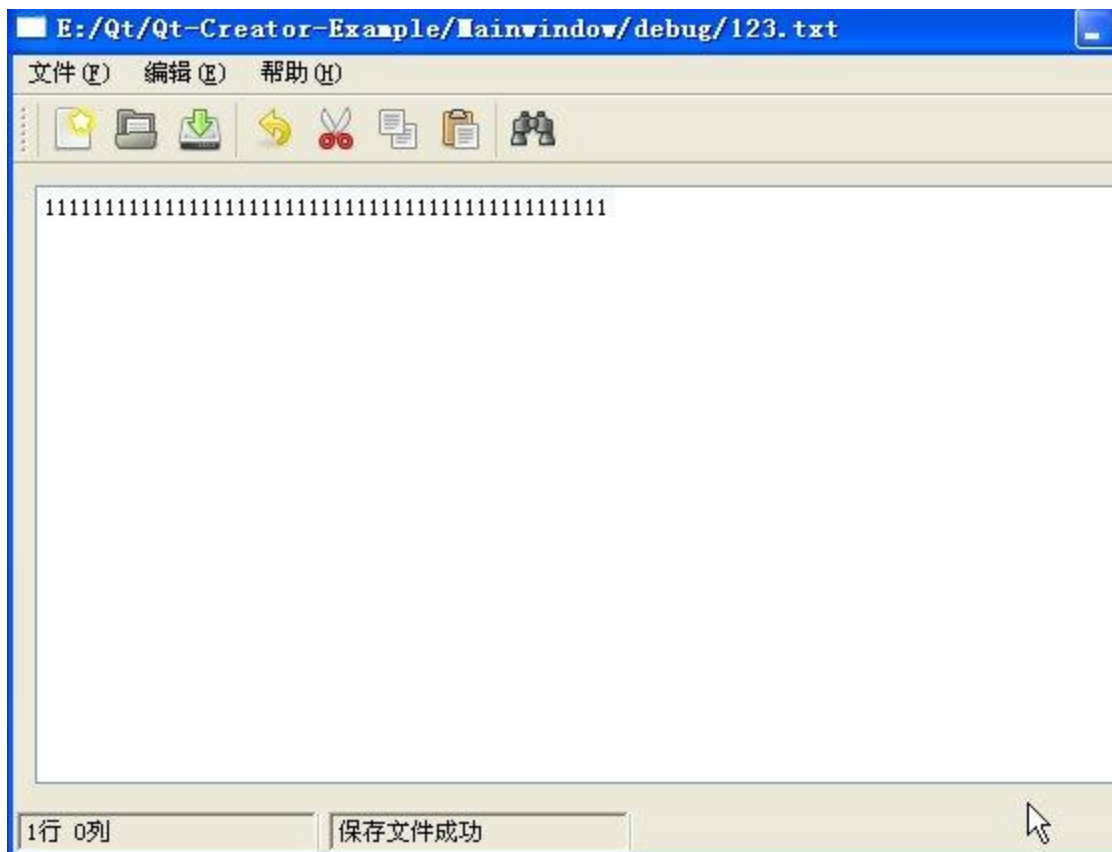
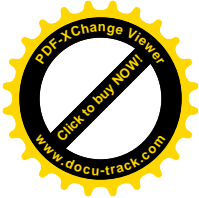
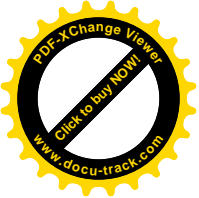
6. 在 do_file_Load 函数的最后添加下面语句。

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



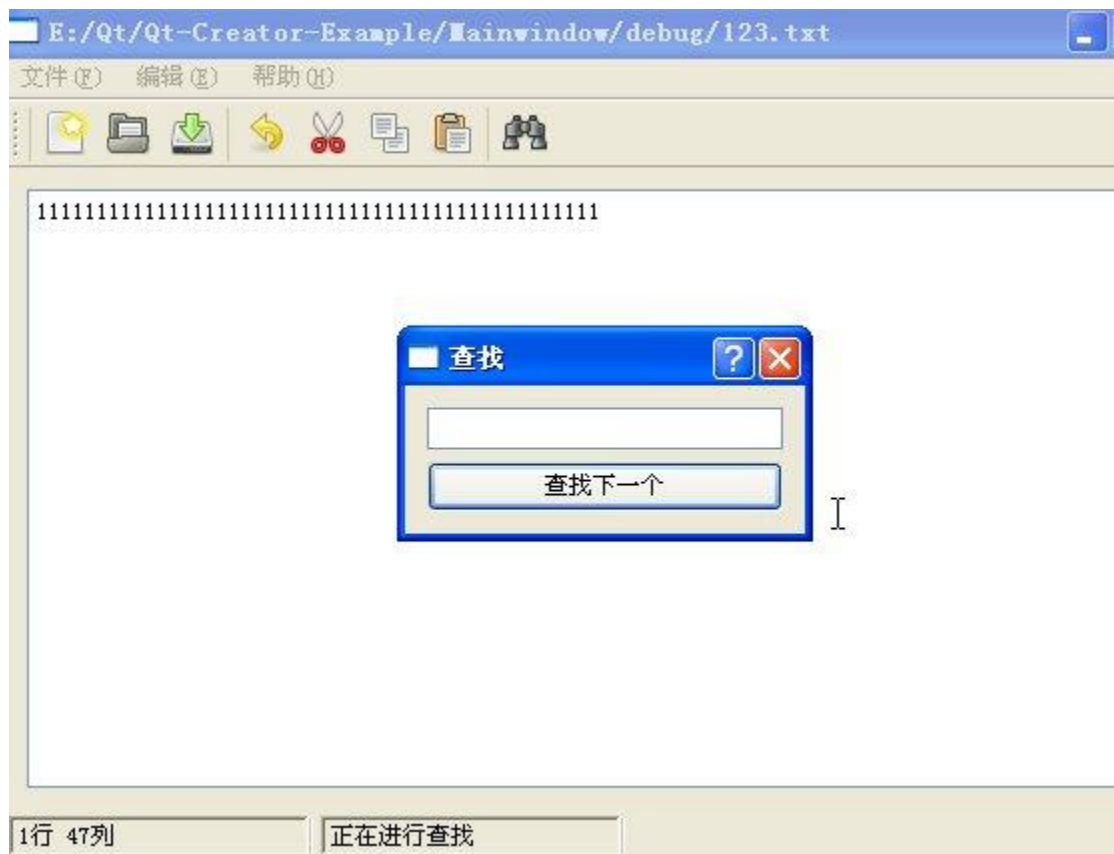
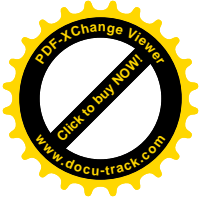
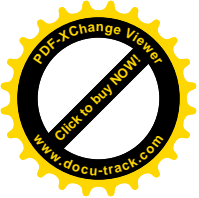
7. 在 saveFile 函数的最后添加以下语句。

```
second_statusLabel->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。

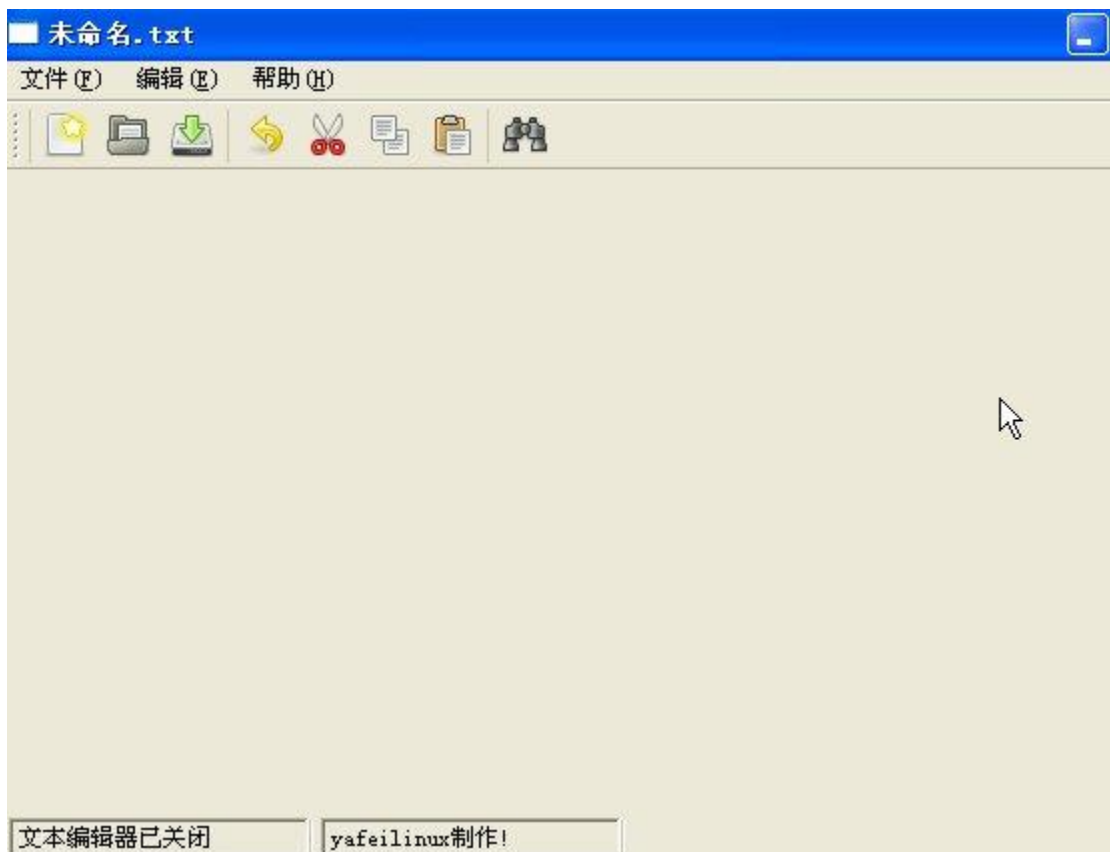
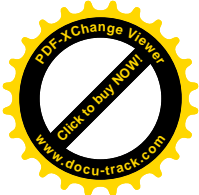
```
second_statusLabel->setText(tr("正在进行查找"));
```



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

```
first_statusLabel->setText(tr("文本编辑器已关闭"));
```

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

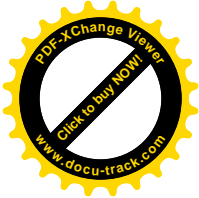
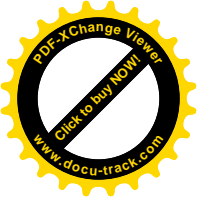


最终的 mainwindow.cpp 文件内容如下。

[illegible]



最终的 mainwindow.h 文件如下。



```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui/QMainWindow>
#include <QListWidget>
#include <QLabel>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0)
        : QMainWindow(parent) {}

private:
    Ui::MainWindow ui;

    void loadSaveData() //加载/保存数据文件内容, 为QtGui的窗口内容提供数据
    void on_action_Open_triggered() //保存当前窗口内容
    void on_action_Save_triggered() //保存+T行编辑内容, 用于输入更复杂的内部
    void on_action_Print_triggered() //打印内容+预览内容, 用于显示预览内容
    void on_action_PrintPreview_triggered() //打印预览内容
    void on_action_SaveAs_triggered() //另存为文件
    void on_action_Open_triggered() //打开文件
    void on_action_Load_triggered() //加载文件
    void on_action_Save_triggered() //保存文件

private slots:
    void on_action_Find_triggered();
    void on_action_Past_triggered();
    void on_action_Copy_triggered();
    void on_action_Cut_triggered();
    void on_action_Undo_triggered();
    void on_action_Quit_triggered();
    void on_action_Close_triggered();
    void on_action_Open_triggered();
    void on_action_SaveAs_triggered();
    void on_action_Save_triggered();
    void on_action_New_triggered();
    void show_findText();
    void do_cursorChanged(); //获取光标位置信息
};

#endif // MAINWINDOW_H
```



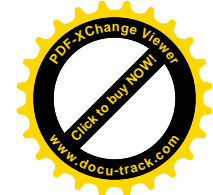
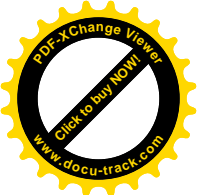
到这里整个文本编辑器的程序就算写完了。我们这里没有写帮助菜单的功能实现，大家可以自己添加。而且程序中也有很多漏洞和不完善的地方，如果有兴趣，大家也可以自己修改。因为时间和篇幅的原因，我们这里就不再过多的讲述。

九、Qt Creator 中鼠标键盘事件的处理实现 自定义鼠标指针（原创）

2009-11-08 21:24

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

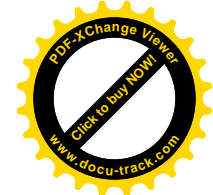
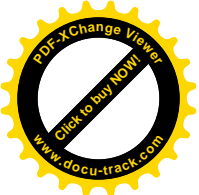
```
ui->label->setText(tr("按下键盘上的 A 键试试！"));
```

```
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，注意这里要用绝对路径，且要用“/”，而不能用“\”
    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
}
```

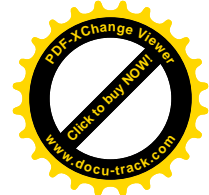
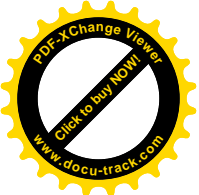


```
ui->pushButton->move(m->pos());  
//让按钮跟随鼠标移动  
}  
  
/*以下是键盘按下事件*/  
  
void MainWindow::keyPressEvent(QKeyEvent *k)  
{  
    if(k->key() == Qt::Key_A) //判断是否是 A 键按下  
    {  
        ui->label->setPixmap(QPixmap("E:/Qt/Qt-Creator-Example/event/linux.jpg"));  
        ui->label->resize(100,100);  
        //更改标签图片和大小  
    }  
}
```

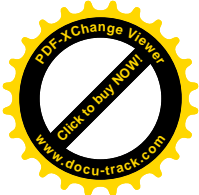
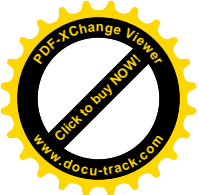
注意：这两个函数不是自己新建的，而是对已有函数的重定义，所有函数名和参数都不能改。第一个函数对鼠标移动事件进行了重写。其中实现了鼠标指针的更改，和按钮跟随鼠标移动的功能。

第二个函数对键盘的 A 键按下实现了新的功能。

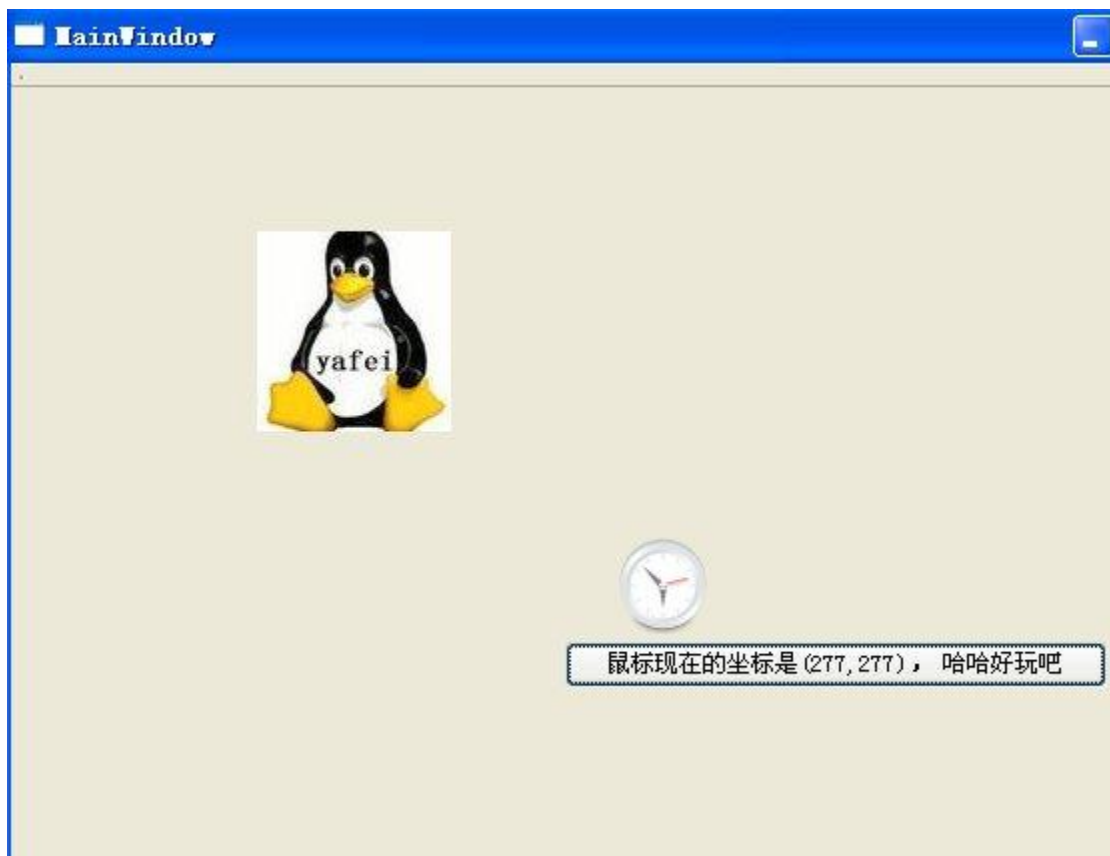
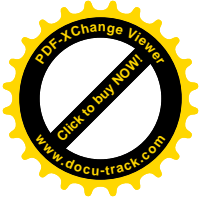
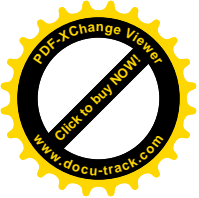
效果如下。



按下鼠标的一个键，并移动鼠标。



按下键盘上的 A 键。



十、Qt Creator 中实现定时器和产生随机数 (原创)

2009-11-08 22:20

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux>
转载请注明出处。

有两种方法实现定时器。

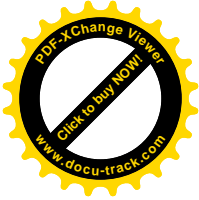
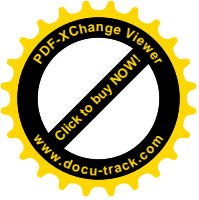
第一种。自己建立关联。

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
}
```

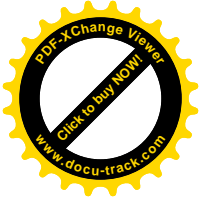
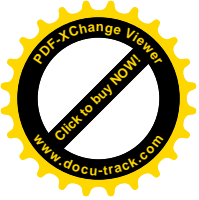
5. 运行程序，效果如下。



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
```

```
}
```

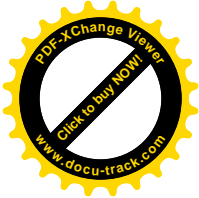
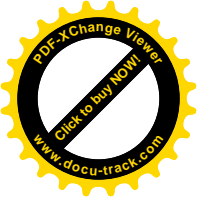
```
}
```

这里添加了三个定时器，并都在定时器事件中判断它们，然后执行相应的功能。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

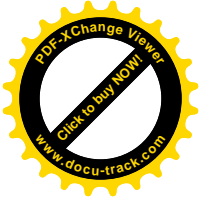
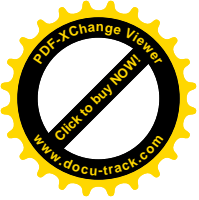


程序每次启动时 `time(0)` 返回的值都不同，这样就实现了产生不同初始值的功能。

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。





十一、Qt 2D 绘图（一）绘制简单图形（原创）

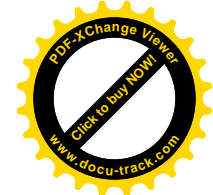
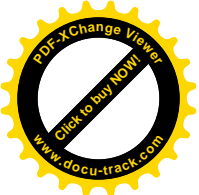
2009-12-10 12:05

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

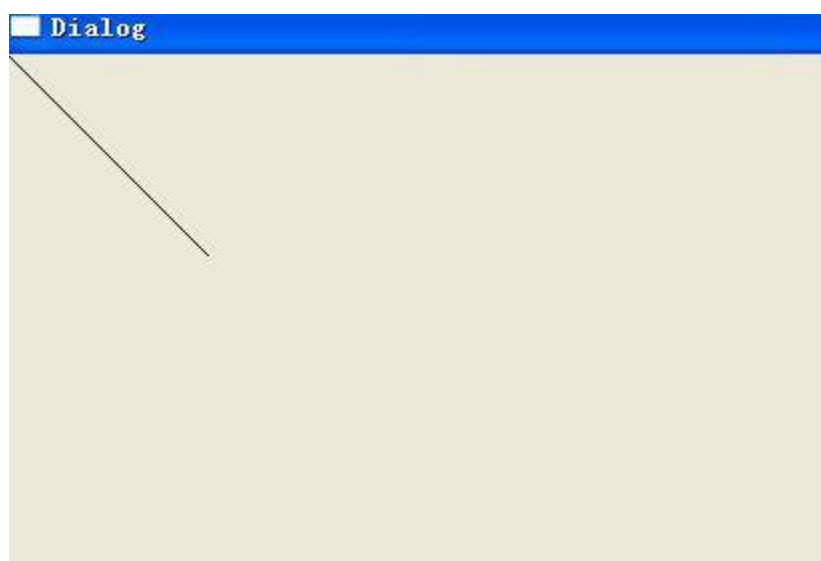
1. 新建 Qt4 Gui Application 工程，我这里使用的工程名为 painter01，选用 QDialog 作为 Base class
2. 在 dialog.h 文件中声明重绘事件函数 `void paintEvent(QPaintEvent *)`;
3. 在 dialog.cpp 中添加绘图类 QPainter 的头文件包含 `#include <QPainter>`



4. 在下面进行该函数的重定义。

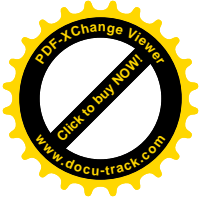
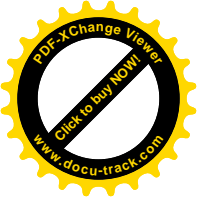
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 100);
}
```

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：

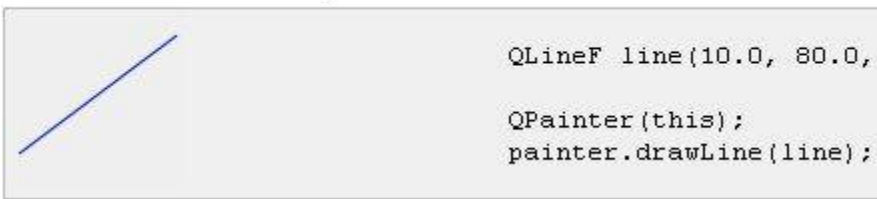


5. 在 qt 的帮助里可以查看所有的绘制函数，而且下面还给出了相关的例子。

```
void drawArc ( const QRectF & rectangle, if
void drawArc ( const QRect & rectangle, int
void drawArc ( int x, int y, int width, int heigh
void drawChord ( const QRectF & rectangle
void drawChord ( const QRect & rectangle,
void drawChord ( int x, int y, int width, int he
void drawConvexPolygon ( const QPointF
void drawConvexPolygon ( const QPoint *
void drawConvexPolygon ( const QPolygo
void drawConvexPolygon ( const QPolygo
void drawEllipse ( const QRectF & rectanal
```



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。

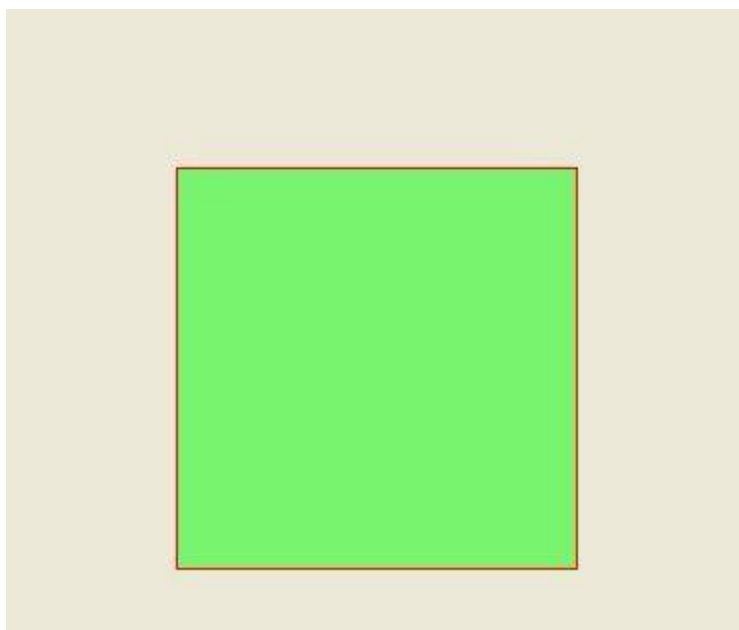
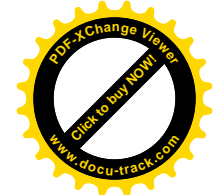
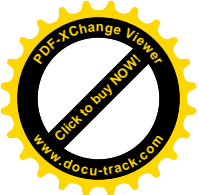
将函数改为如下：

```
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：



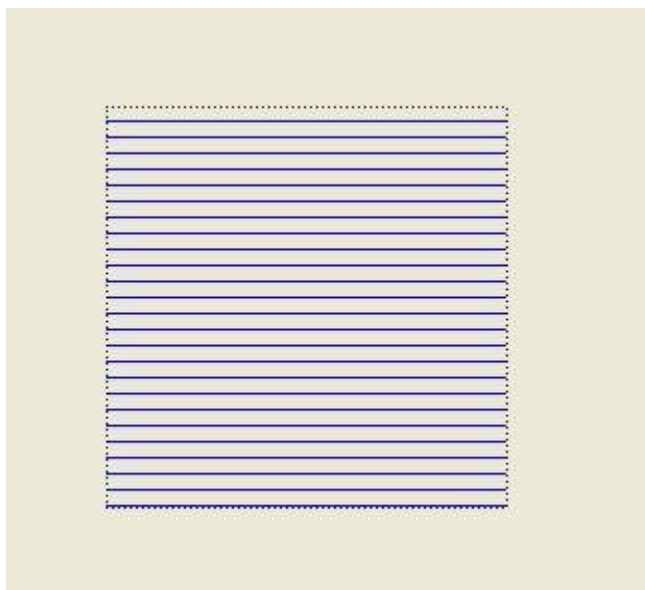
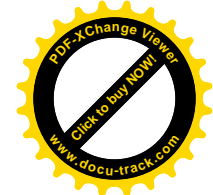
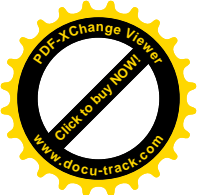
7. 其实画笔和画刷也有很多设置，大家可以查看帮助。

```
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);
```

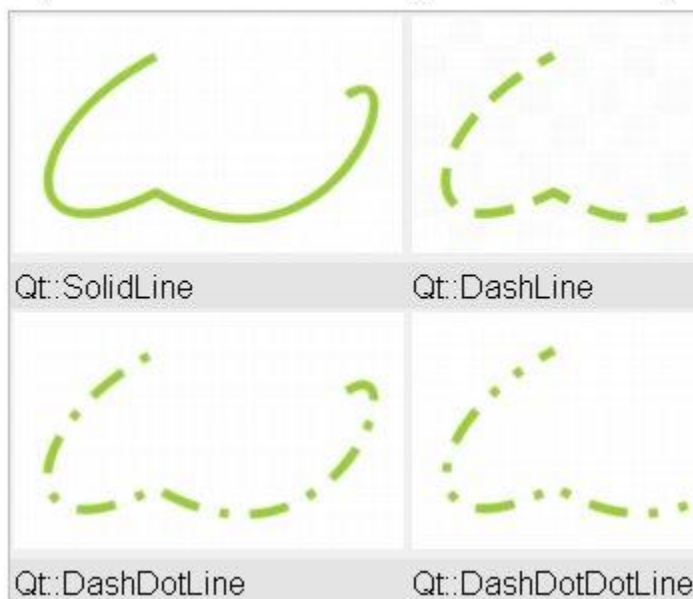
这里我们设置了画笔的风格为点线，画刷的风格为并行横线，效果如下：

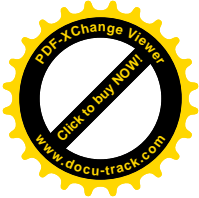
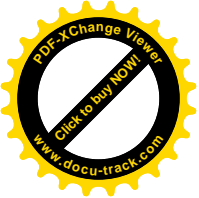


在帮助里可以看到所有的风格。

enum Qt::PenStyle

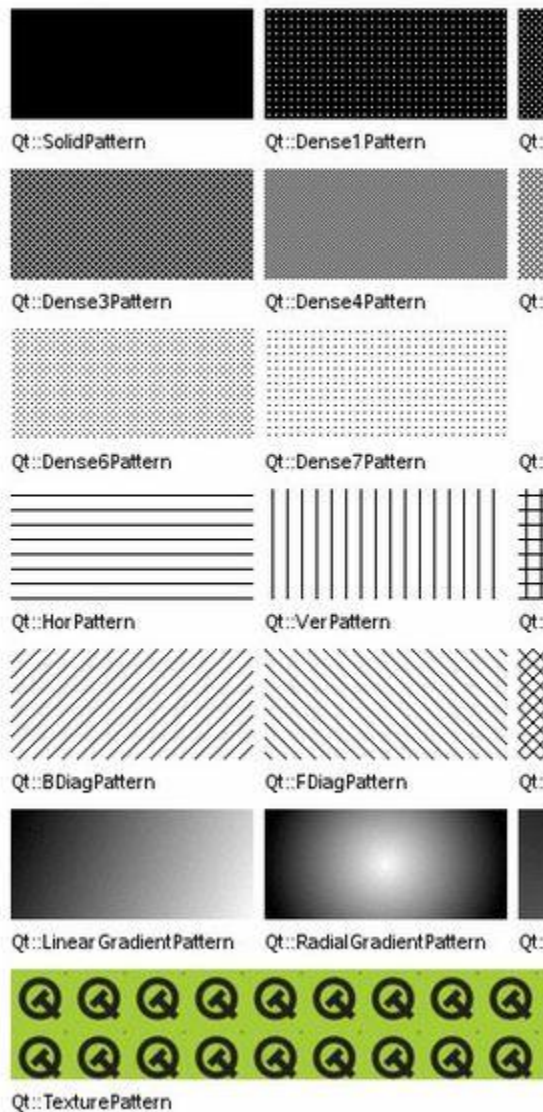
This enum type defines the pen styles that can be drawn using QPainter. The style



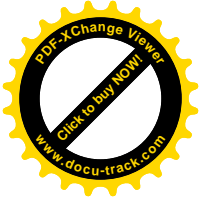
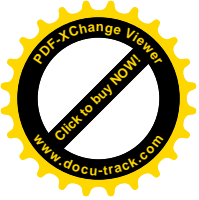


enum Qt::BrushStyle

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



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	
Qt::white	3	White (#ffffff)
Qt::black	2	Black (#000000)
Qt::red	7	Red (#ff0000)
Qt::darkRed	13	Dark red (#800000)
Qt::green	8	Green (#00ff00)
Qt::darkGreen	14	Dark green (#008000)
Qt::blue	9	Blue (#0000ff)
Qt::darkBlue	15	Dark blue (#000080)
Qt::cyan	10	Cyan (#00ffff)
Qt::darkCyan	16	Dark cyan (#008080)
Qt::magenta	11	Magenta (#ff00ff)
Qt::darkMagenta	17	Dark magenta (#800080)
Qt::yellow	12	Yellow (#ffff00)
Qt::darkYellow	18	Dark yellow (#808000)
Qt::gray	5	Gray (#a0a0a0)
Qt::darkGray	4	Dark gray (#808080)
Qt::lightGray	6	Light gray (#c0c0c0)
Qt::transparent	19	a transparent black \
Qt::color0	0	0 pixel value (for bitn
Qt::color1	1	1 pixel value (for bitn

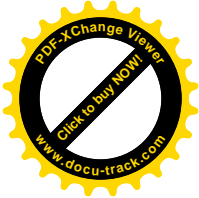
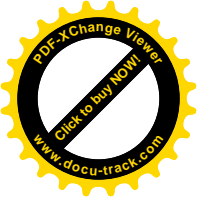
See also QColor.

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*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。这里就不再详述。



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;

QPainter painter(this);
painter.drawArc(rectangle, startAngle, spanAngle);
```

十二、Qt 2D 绘图（二）渐变填充（原创）

2009-12-11 11:47

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

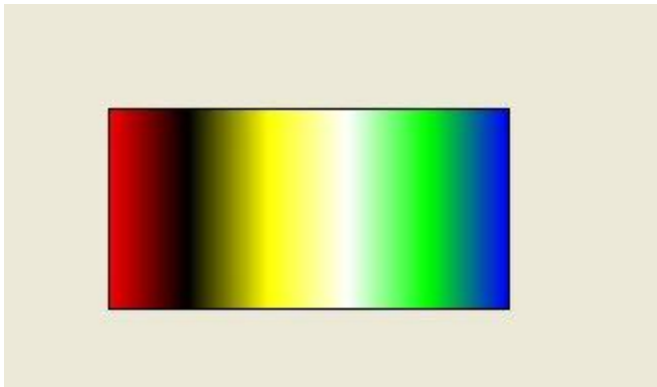
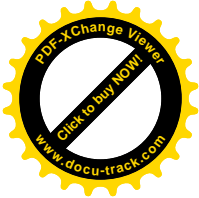
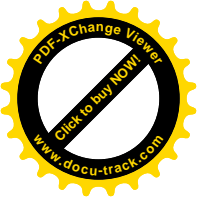
在 qt 中提供了三种渐变方式，分别是线性渐变，圆形渐变和圆锥渐变。如果能熟练应用它们，就能设计出炫目的填充效果。

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：

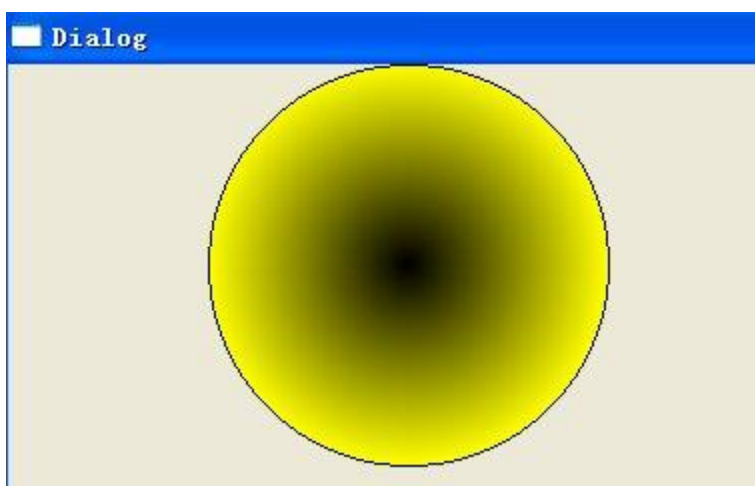


圆形渐变:

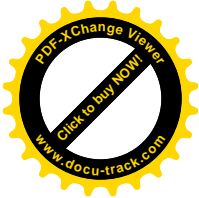
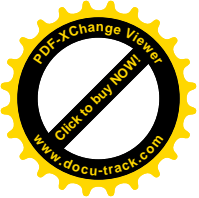
1. 更改函数内容如下:

```
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:

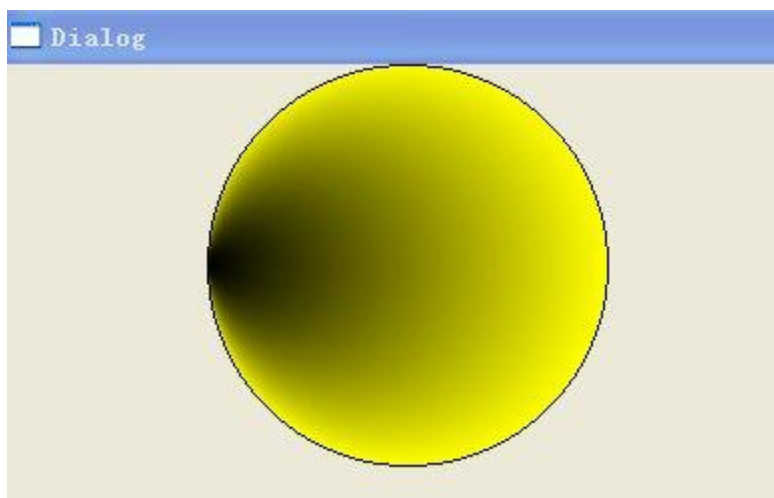


2. 要想改变填充的效果, 只需要改变焦点的位置和渐变的颜色位置即可。



改变焦点位置: `QRadialGradient radialGradient(200, 100, 100, 100, 100);`

效果如下:

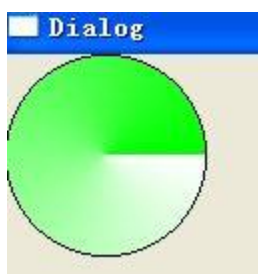


锥形渐变:

1. 更改函数内容如下:

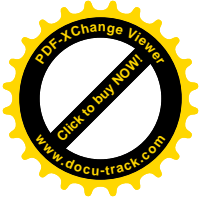
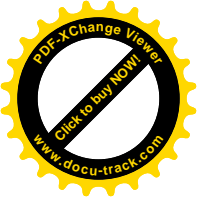
```
//圆锥渐变
    QConicalGradient conicalGradient(50, 50, 0);
    //圆心为 (50, 50) , 开始角度为 0
    conicalGradient.setColorAt(0, Qt::green);
    conicalGradient.setColorAt(1, Qt::white);
    //从圆心的 0 度角开始逆时针填充
    QPainter painter(this);
    painter.setBrush(conicalGradient);
    painter.drawEllipse(0, 0,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



其实三种渐变的设置都在于焦点和渐变颜色的位置，如果想设计出漂亮的渐变效果，还要有美术功底啊！

十三、Qt 2D 绘图（三）绘制文字（原创）

2009-12-22 11:42

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

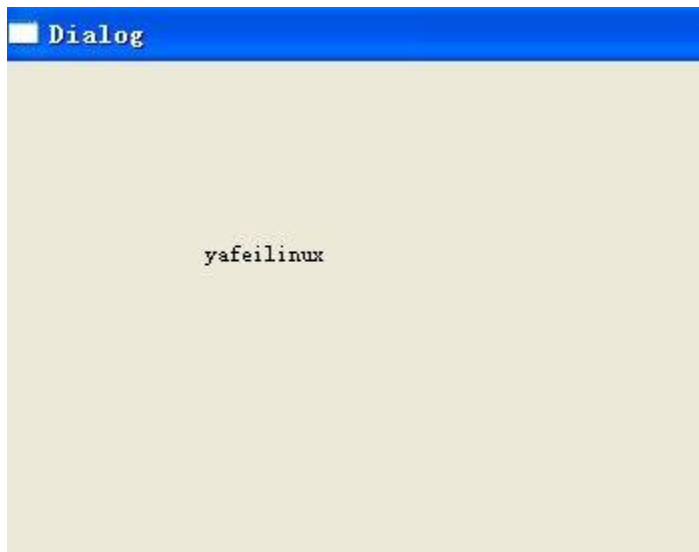
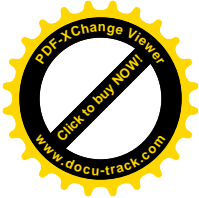
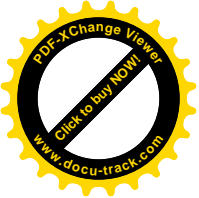
接着上一次的教程，这次我们学习在窗体上绘制文字。

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 QStr
```

This is an overloaded function.

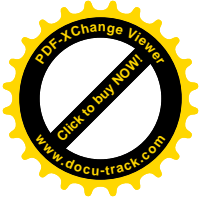
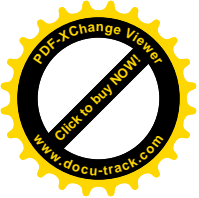
Draws the given *text* within the provided *rectangle*.

Qt by
Trolltech

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

The *boundingRect* (if not null) is set to the what the bounding rectangle should be OR of the following flags:

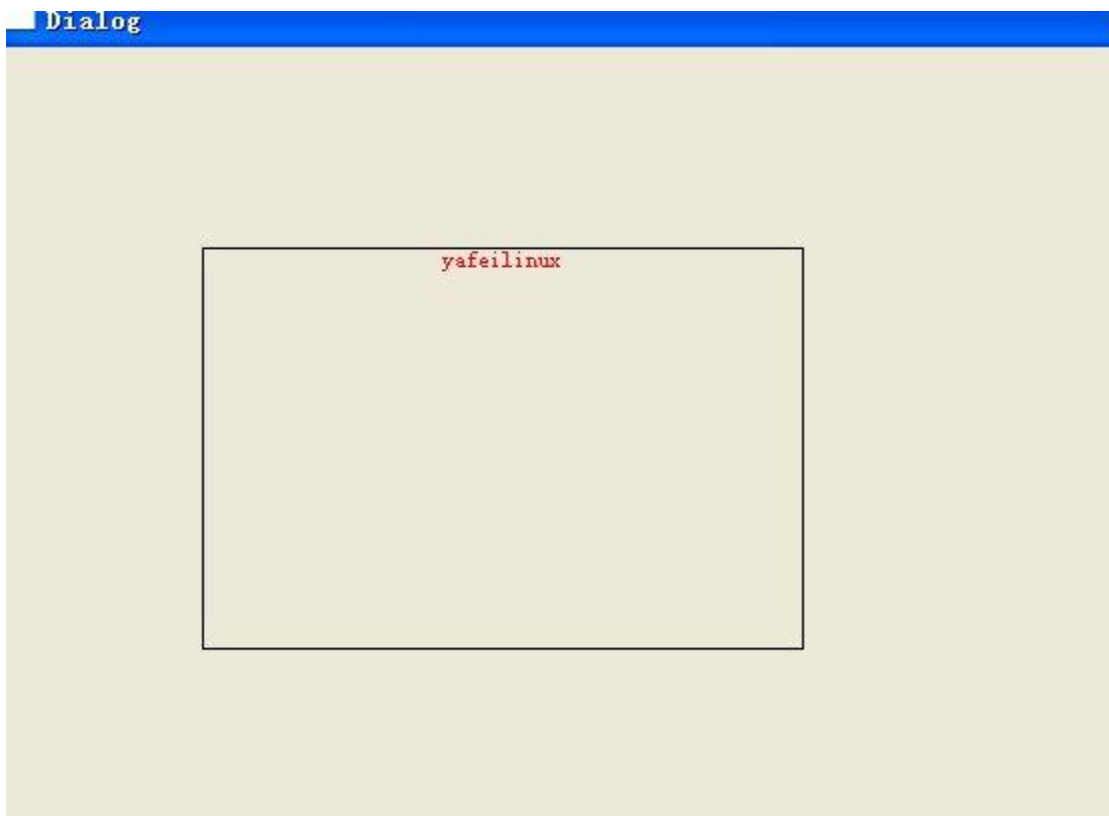
- Qt::AlignLeft
- Qt::AlignRight
- Qt::AlignHCenter
- Qt::AlignJustify
- Qt::AlignTop
- Qt::AlignBottom
- Qt::AlignVCenter
- Qt::AlignCenter
- Qt::TextDontClip
- Qt::TextSingleLine
- Qt::TextExpandTabs
- Qt::TextShowMnemonic
- Qt::TextWordWrap
- Qt::TextIncludeTrailingSpaces



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。

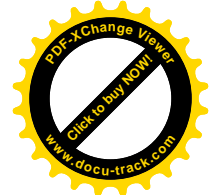
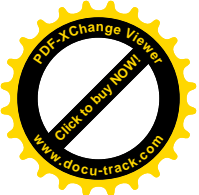
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QRectF ff(100, 100, 300, 200);
    //设置一个矩形
    painter.drawRect(ff);
    //为了更直观地看到字体的位置，我们绘制出这个矩形
    painter.setPen(QColor(Qt::red));
    //设置画笔颜色为红色
    painter.drawText(ff, Qt::AlignHCenter, "yafeilinux");
    //我们这里先让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

效果如下。

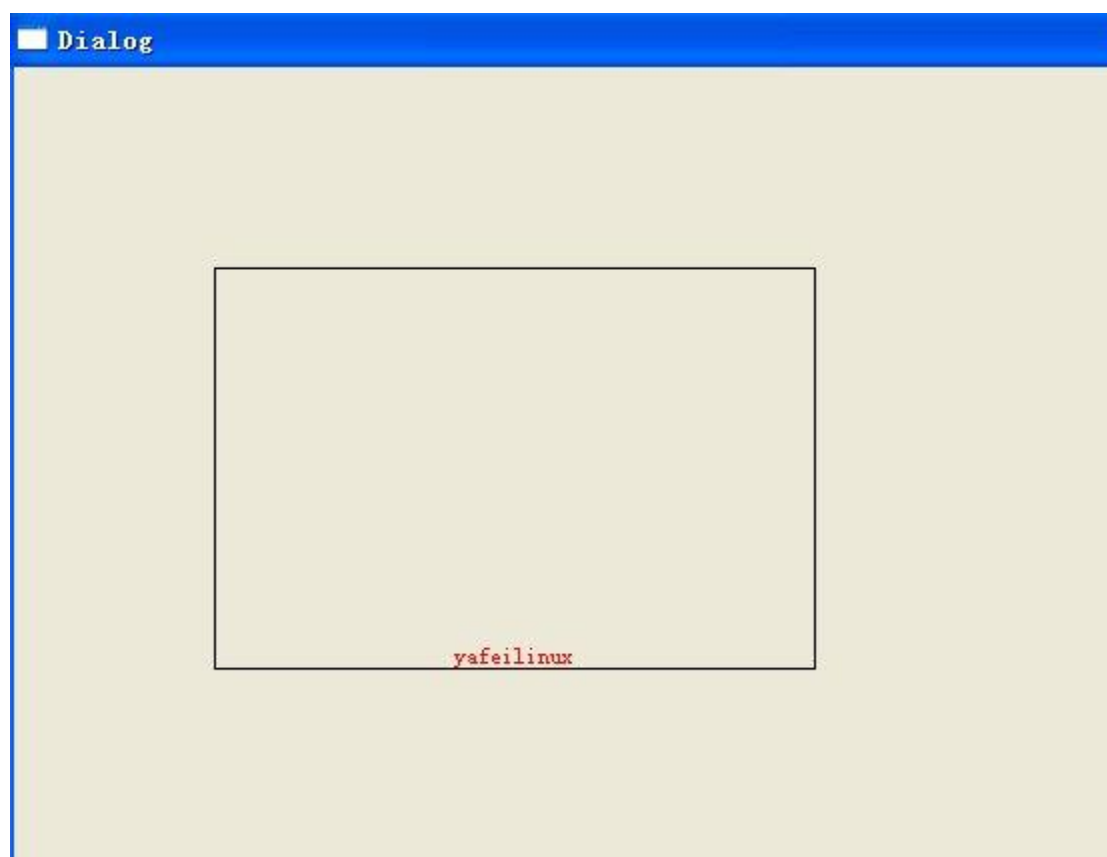


可以看到字符串是在最上面水平居中的。如果想让其在矩形正中间，我们可以使用 `Qt::AlignCenter`。

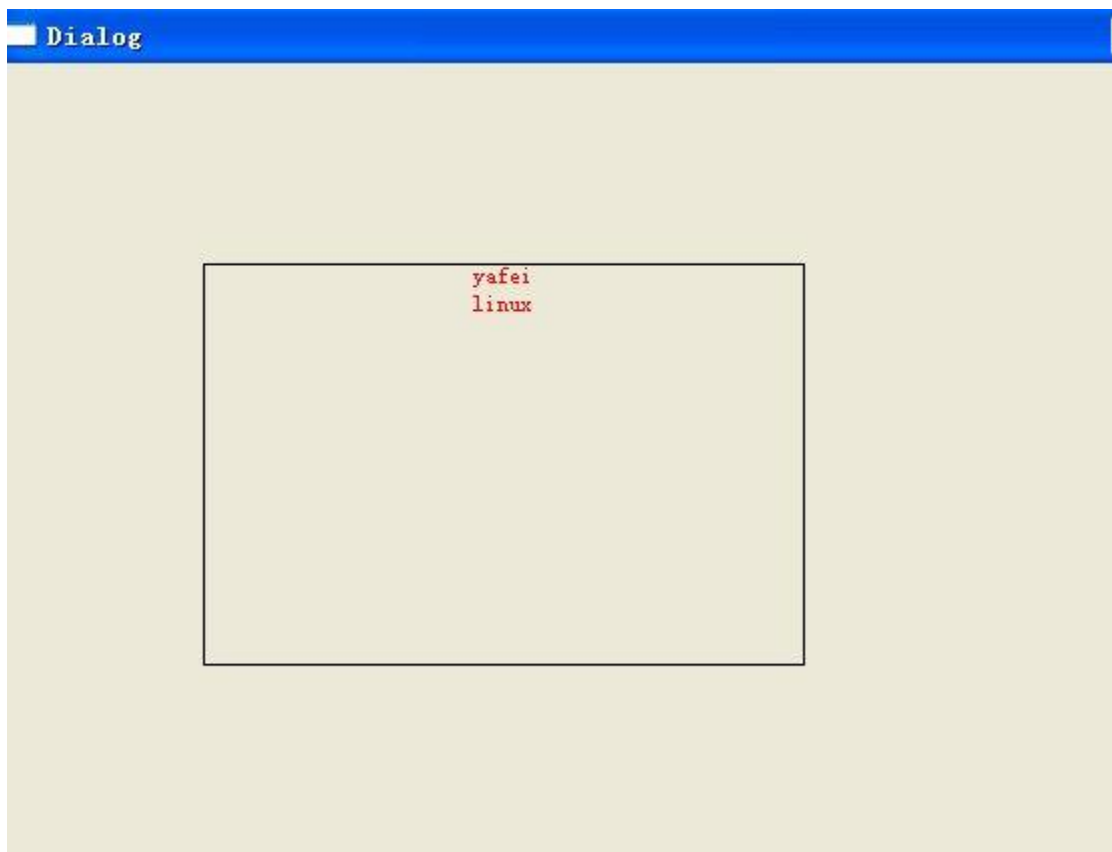
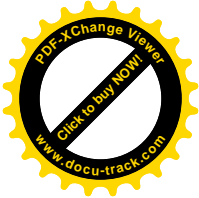
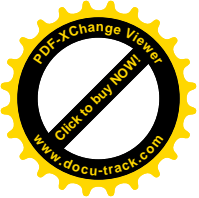
这里我们也可以使用两个枚举变量进行按位与操作，例如可以使用 `Qt::AlignBottom|Qt::AlignHCenter` 实现让文字显示在矩形下面的正中间。效



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。



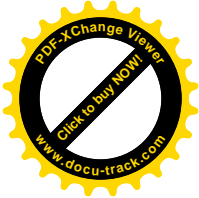
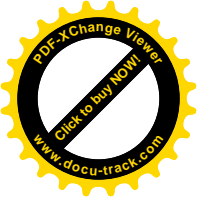
3. 如果要使文字更美观，我们就需要使用 QFont 类来改变字体。先在帮助中查看一下这个类。



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。

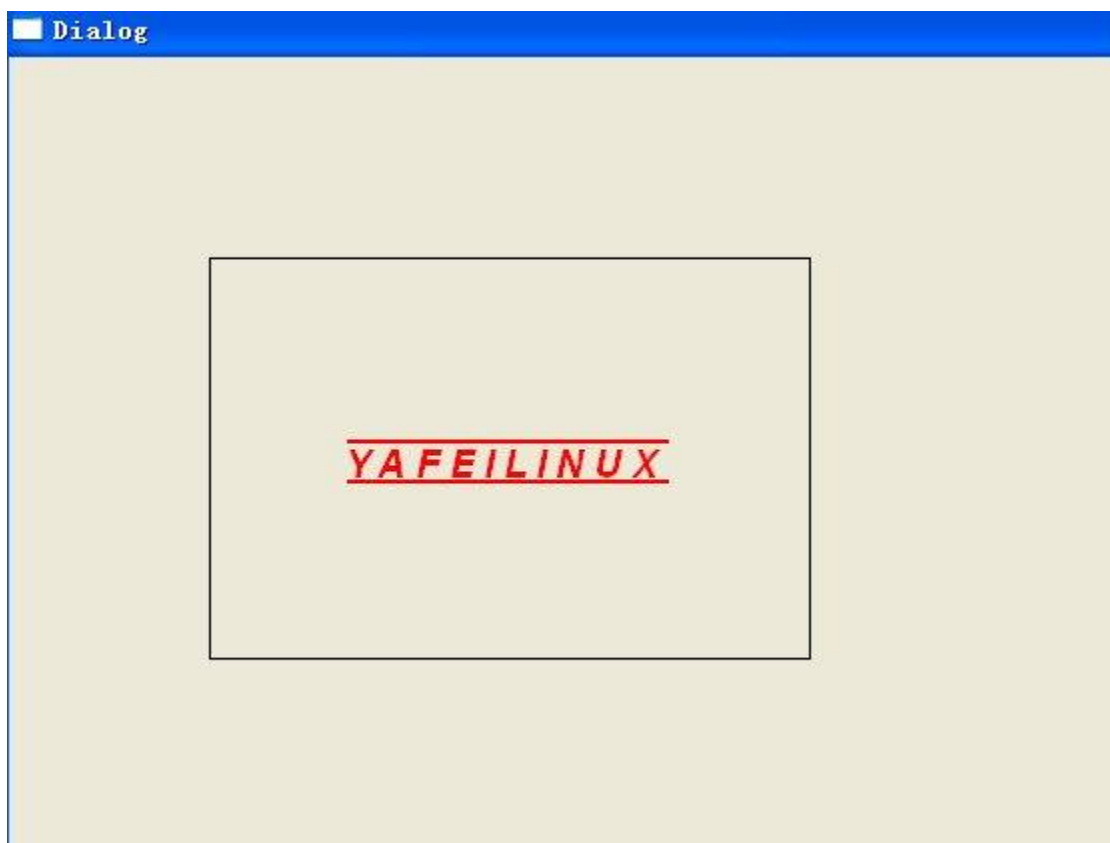
更改函数如下。

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

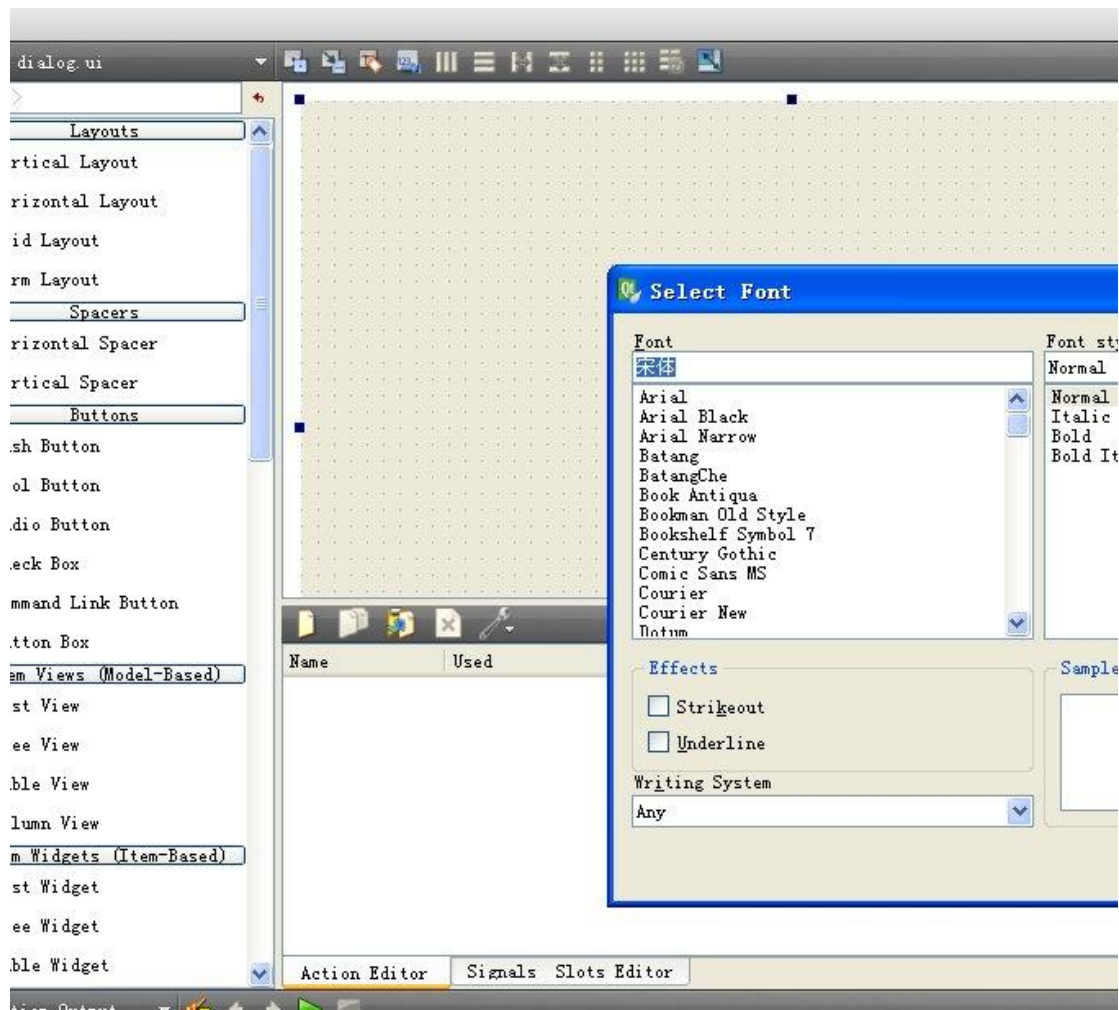
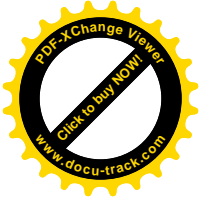
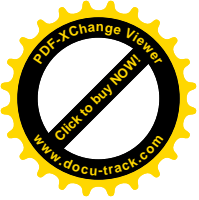


```
font.setUnderline(true);  
//设置下划线  
font.setOverline(true);  
//设置上划线  
font.setCapitalization(QFont::SmallCaps);  
//设置大小写  
font.setLetterSpacing(QFont::AbsoluteSpacing, 5);  
//设置间距  
QPainter painter(this);  
painter.setFont(font);  
//添加字体  
QRectF ff(100, 100, 300, 200);  
painter.drawRect(ff);  
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。



十四、Qt 2D 绘图（四）绘制路径（原创）

2010-01-17 17:32

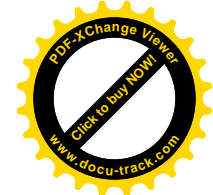
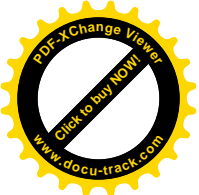
声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

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);
```



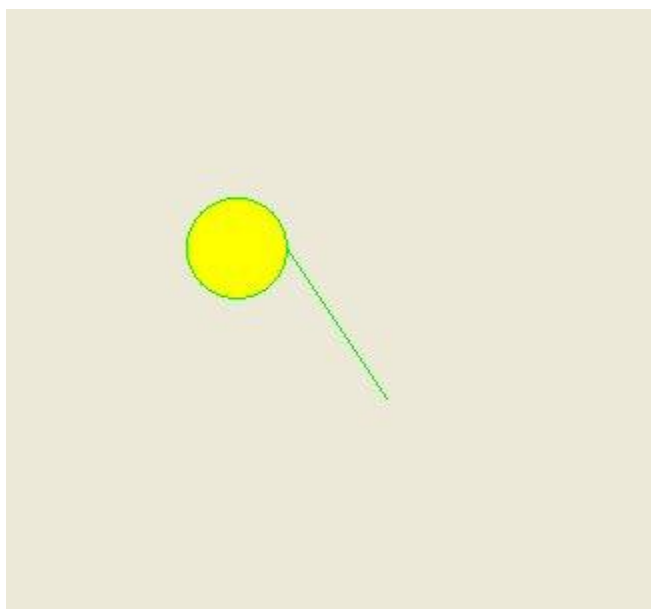
```
path.lineTo(200, 200);

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。



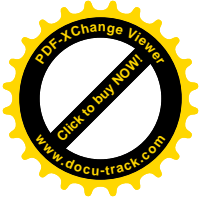
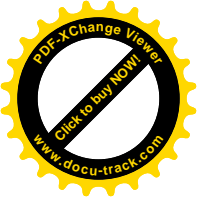
2. 上面绘制圆和直线都有对应的函数啊，为什么还要加入一个 painterPath 呢？

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);
```

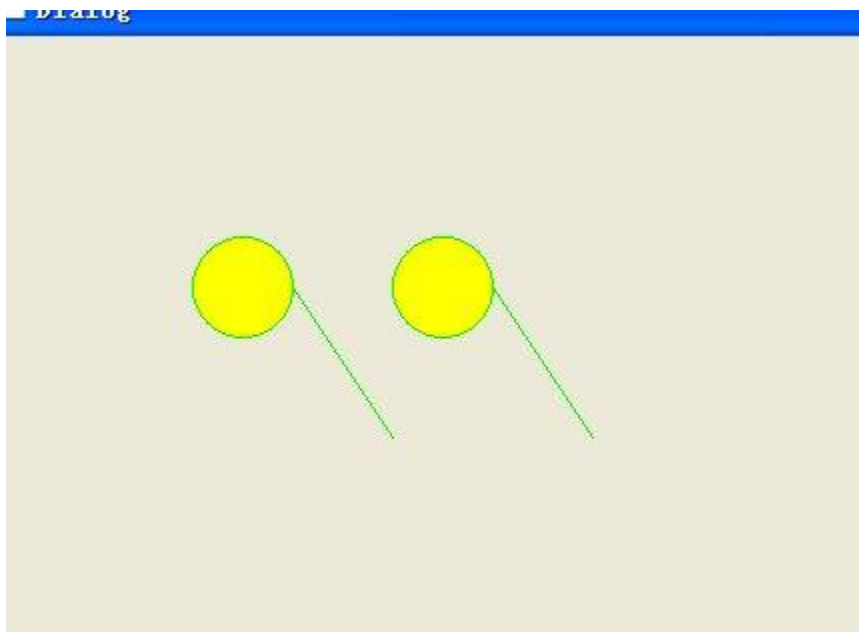


```
painter.drawPath(path);

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。

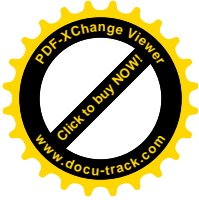
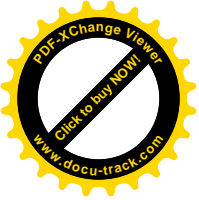
3. 这里我们应该注意的是绘制完一个图形后，当前的位置在哪里。

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：

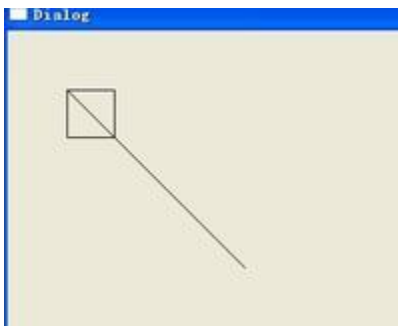
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainterPath path;

    path.addRect(50, 50, 40, 40);
    path.lineTo(200, 200);

    QPainter painter(this);

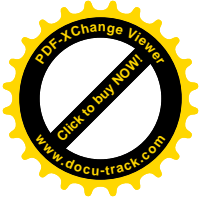
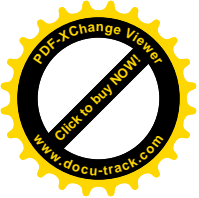
    painter.drawPath(path);
}
```

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 *)
{
```

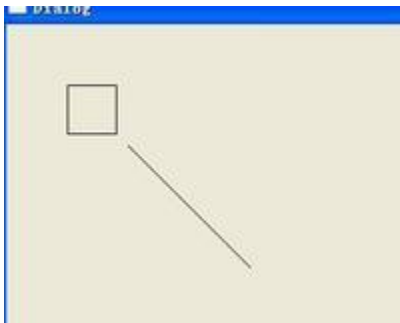


```
QPainterPath path;

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。

十五、Qt 2D 绘图（五）显示图片（原创）

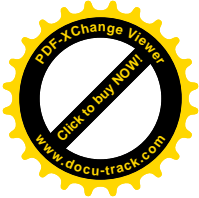
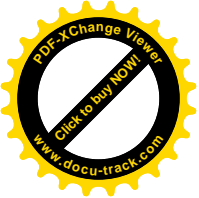
2010-01-24 12:05

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

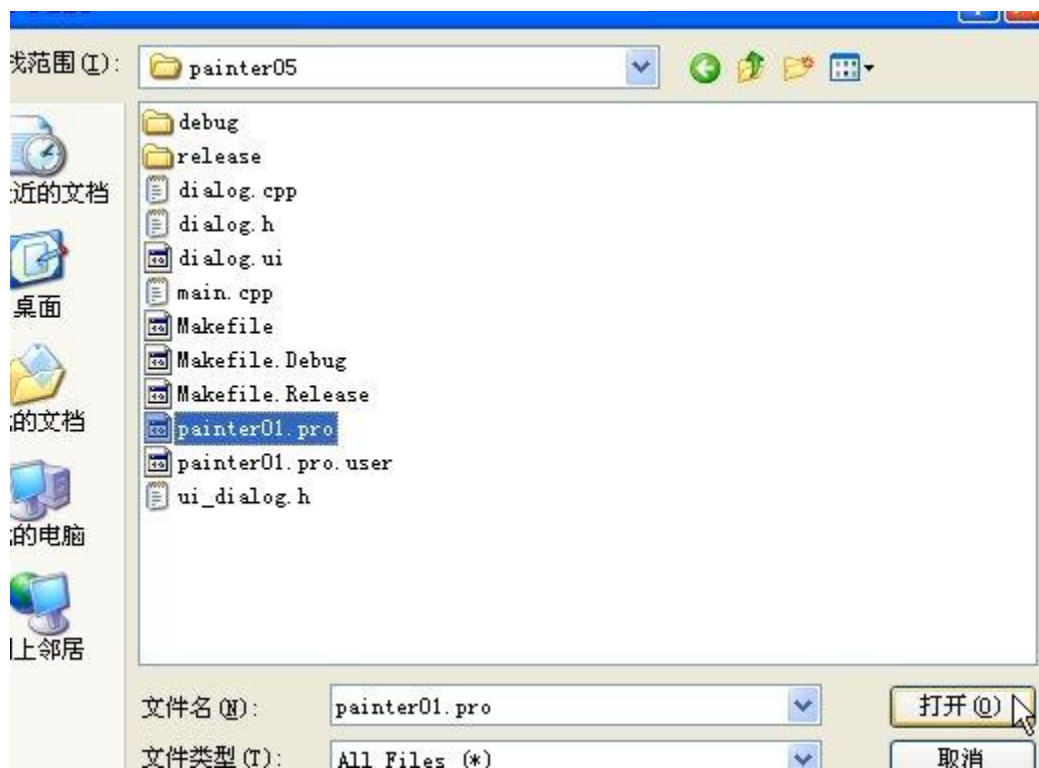
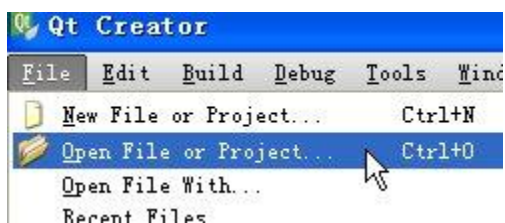
现在我们来实现在窗口上显示图片，并学习怎样将图片进行平移，缩放，旋转和扭曲。这里我们是利用 QPixmap 类来实现图片显示的。

一、利用 QPixmap 显示图片。

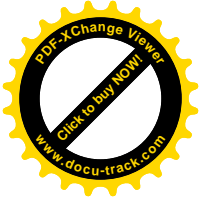
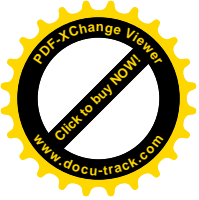
1. 将以前的工程文件夹进行复制备份，我们这里将工程文件夹改名为 painter05。（以前已经说过，经常备份工程目录，是个很好的习惯）
2. 在工程文件夹的 debug 文件夹中新建文件夹，我这里命名为 images，用来存放要用的图片。我这里放了一张 linux.jpg 的图片。如下图所示。



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。



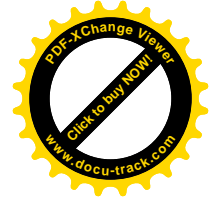
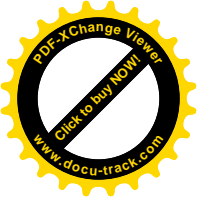
（说明：下面的操作都会和坐标有关，这里请先进行操作，我们在下一节将会讲解坐标系。）

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。

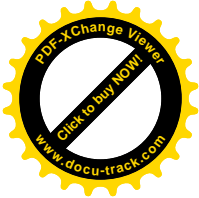
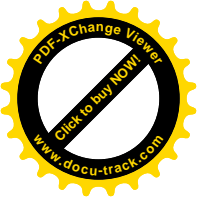
我们可以使用 QPixmap 类中的 scaled() 函数来实现图片的放大和缩小。

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。




)

x);

`Qt::KeepAspectRatio`; F1

enum Qt::AspectRatioMode

This enum type defines what happens to the scaling an rectangle.

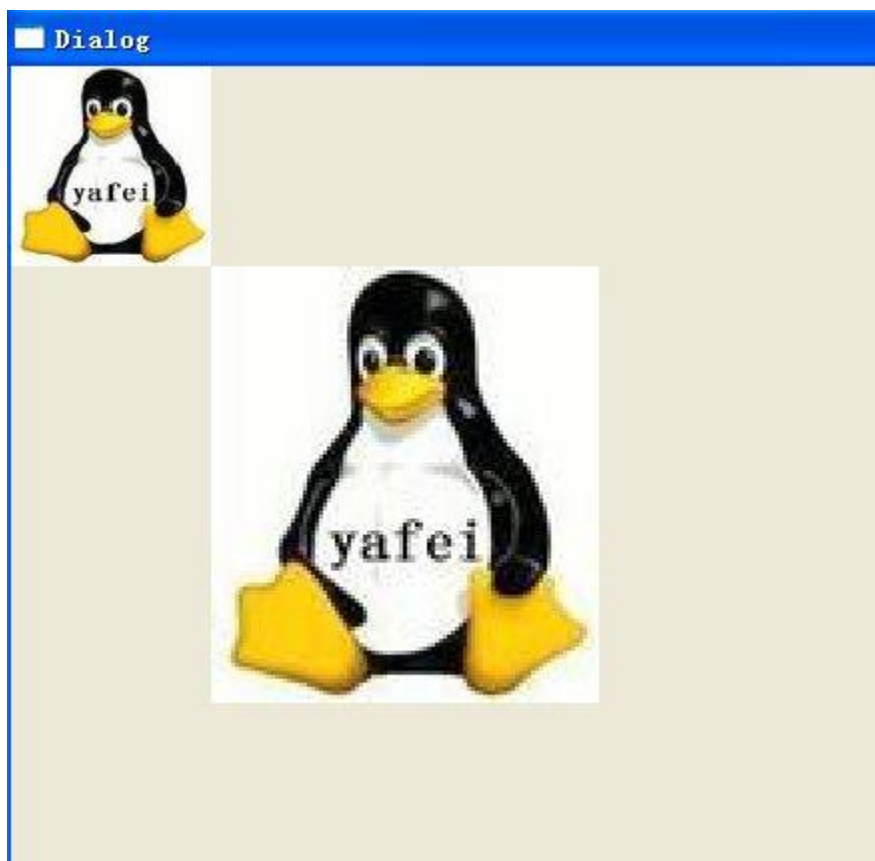
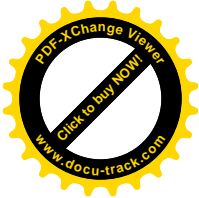
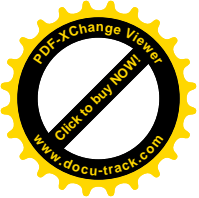


IgnoreAspectRatio KeepAsp

Constant	Value
<code>Qt::IgnoreAspectRatio</code>	0
<code>Qt::KeepAspectRatio</code>	1

这是个枚举变量，这里三个值，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。

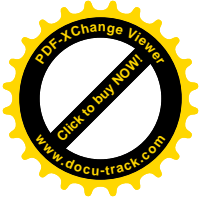
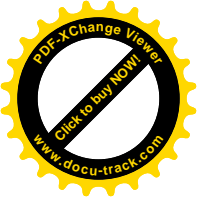
旋转使用的是 QPainter 类的 rotate() 函数,它默认是以原点为中心进行旋转的。我们要改变旋转的中心, 可以使用前面讲到的 translate() 函数完成。

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，前面的参数实现横行变形，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：

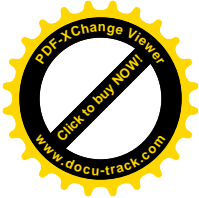
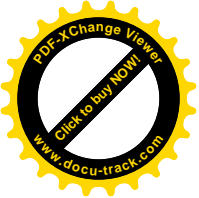
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load("images/linux.jpg");
    painter.drawPixmap(0,0,100,100,pix);
    painter.shear(0.5,0); //横向扭曲
    painter.drawPixmap(100,0,100,100,pix);
}
```

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。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。

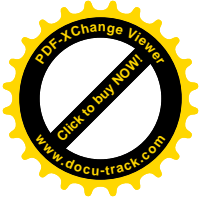
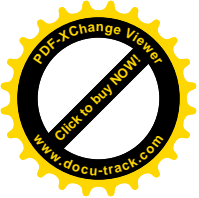
十六、Qt 2D 绘图（六）坐标系（原创）

2010-01-24 14:06

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux>
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，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。

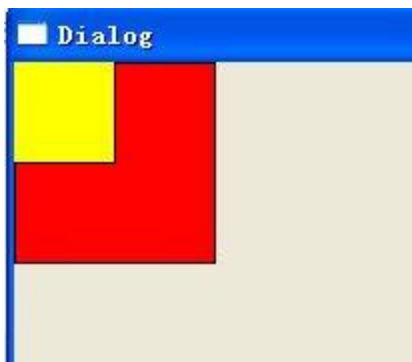
一、坐标系简介。

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);
}
```

我们先在坐标原点 (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。

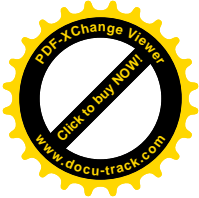
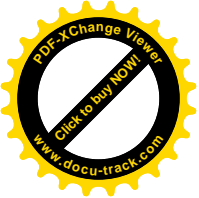


二、坐标系变换。

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);

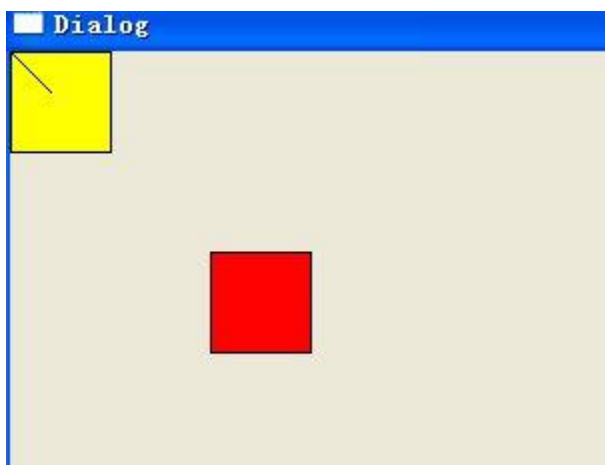
painter.translate(100, 100); //将点 (100, 100) 设为原点

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

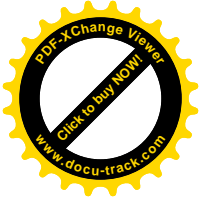
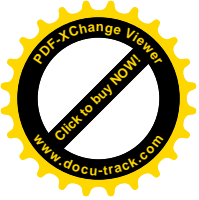
(-100, -100) 点。要想使原来的 (0, 0) 点重新成为原点，就是将 (-100, -100) 设为原点。

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);

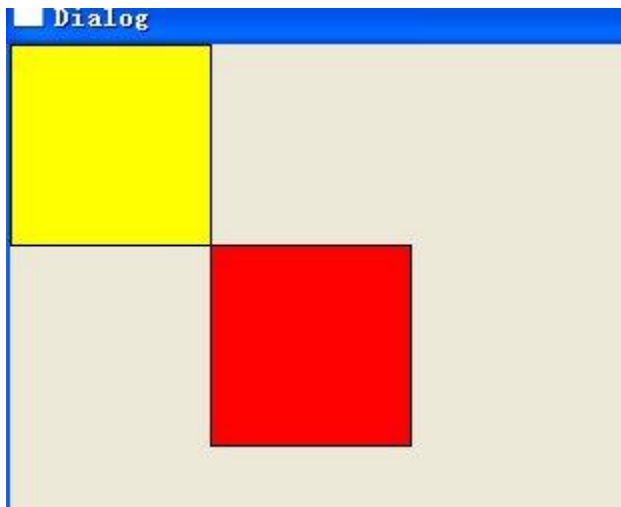
    painter.scale(2, 2); //放大两倍

    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

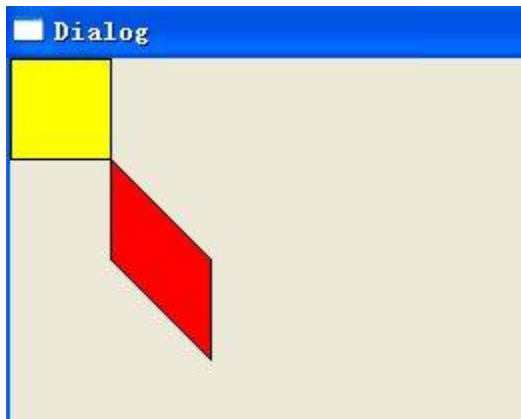
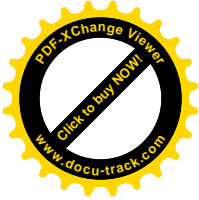
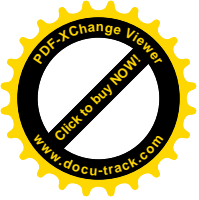
（100，100）点。

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 更改时就会对横行进行扭曲，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。

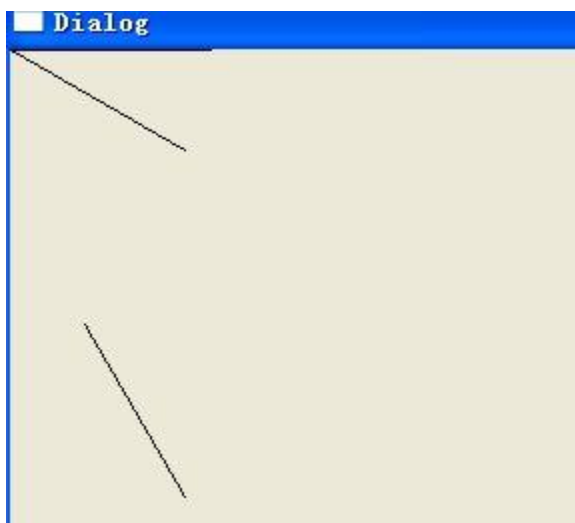
4. 利用 `rotate()` 函数进行比例变换，实现缩放效果。

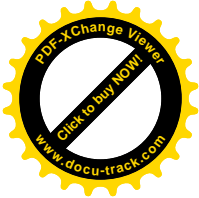
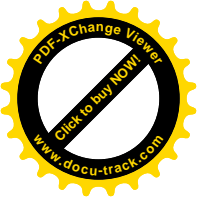
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);

    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：

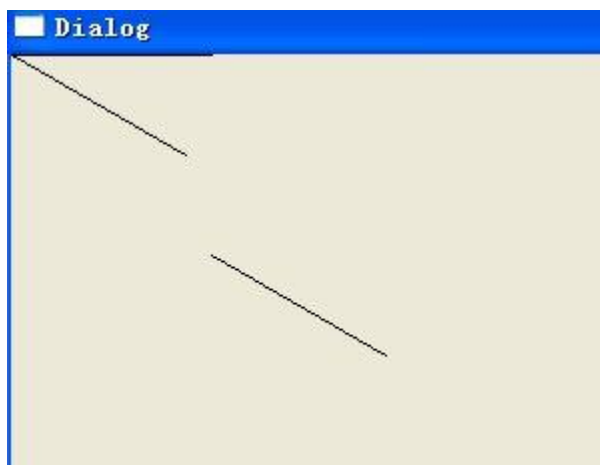
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);

    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);
}
```

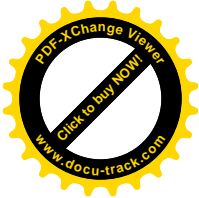
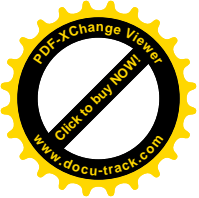
效果如下。



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。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



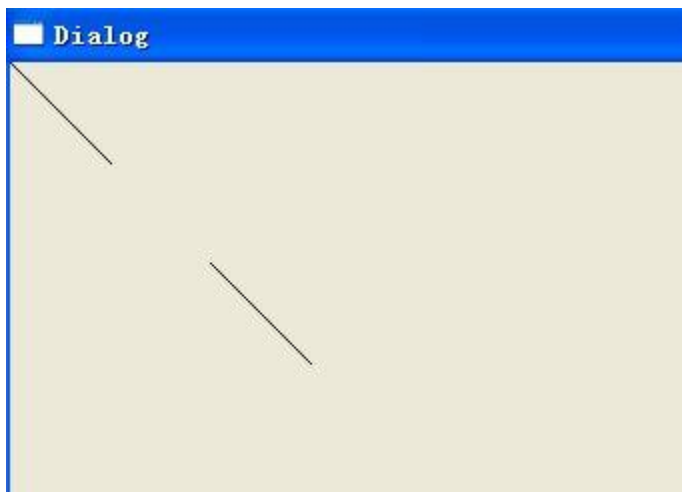
出栈的操作。

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。



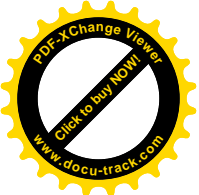
利用好这两个函数，可以实现快速的坐标系切换，绘制出不同的图形。

十七、Qt 2D 绘图（七）Qt 坐标系统深入（原创）

2010-02-11 17:31

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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();
}
```

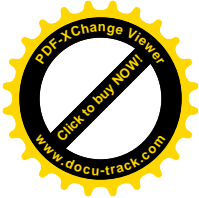
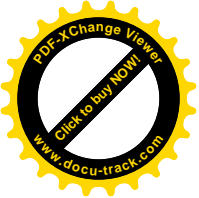
这里应用了 `qDebug()` 函数，利用该函数可以在程序运行时将程序中的一些信息输出，在 Qt Creator 中会将信息输出到其下面的 Application Output 窗口。这个函数很有用，在进行简单的程序调试时，都是利用该函数进行的。我们这里利用它将鼠标指针的坐标值输出出来。

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：



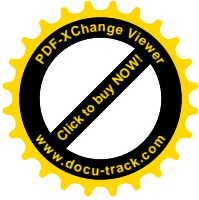
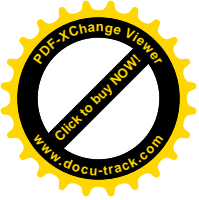
我们点击矩形右下顶点，是（100，100），比以前的（50，50）扩大了 2 倍。

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 QPainter 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。

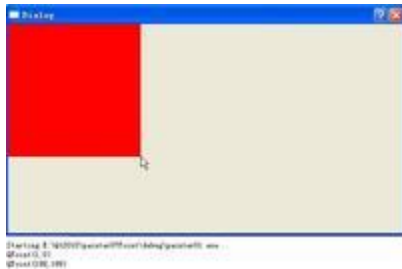
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    pix.fill(Qt::red);      //背景填充为红色
```



```
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：

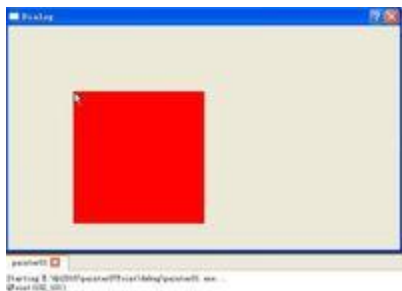


可以看到其左上角为 (0, 0) 点，右下角为 (200, 200) 点，是没有问题的。

2. 我们再将函数更改如下。

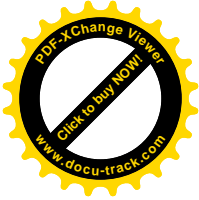
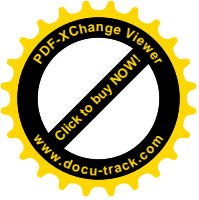
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QPixmap pix(200, 200);  
    pix.fill(Qt::red);    //背景填充为红色  
    painter.drawPixmap(100, 100, pix);  
}
```

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。



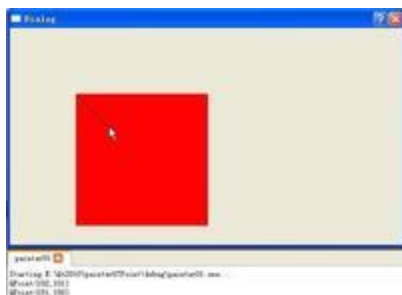
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    pix.fill(Qt::red);      //背景填充为红色
    QPainter pp(&pix);      //新建 QPainter 类对象，在 pix 上进行绘图
    pp.drawLine(0,0,50,50); //在 pix 上的 (0, 0) 点和 (50,
50) 点之间绘制直线
    painter.drawPixmap(100,100,pix);
}
```

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。



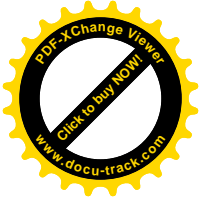
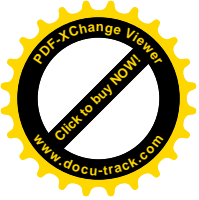
结果是直线的两端的坐标分别是 (100, 100)，(150, 150)。我们从中可以得出这样的结论：

第一，QWidget 和 QPixmap 各有一套坐标系，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：

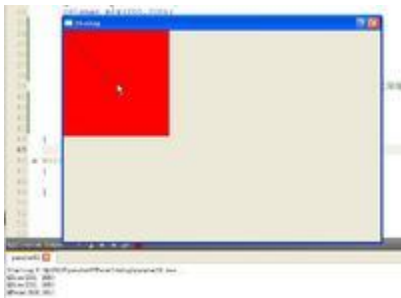
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);

    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);
}
```



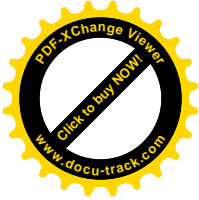
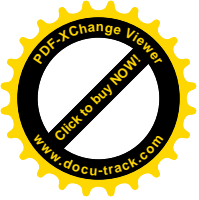
例子二：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);

    qDebug() << pix.size();    //放大前输出 pix 的大小

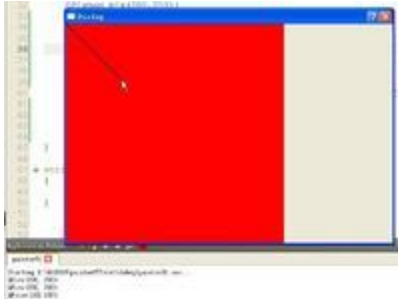
    painter.scale(2, 2);        //窗口坐标扩大 2 倍

    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，到底是怎么样呢？

其实只需记住一句话：

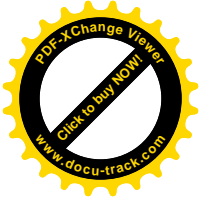
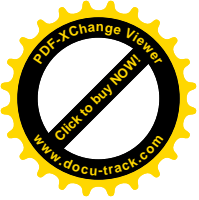
所有的绘图设备都有自己的坐标系统，它们互不影响。

十八、Qt 2D 绘图（八）涂鸦板（原创）

2010-02-12 12:23

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

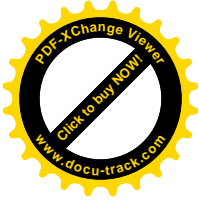
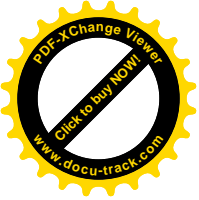
2. 在 dialog.cpp 中进行修改。

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();
    }
}

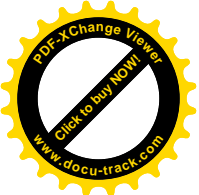
void Dialog::mouseReleaseEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton) //鼠标左键释放
    {
        endPoint = event->pos();
        update();
    }
}
```

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。下面我们将进行放大后的涂鸦。



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。

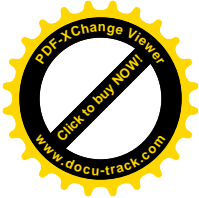
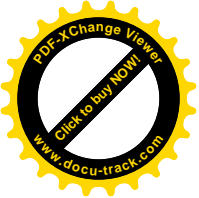
再在构造函数以外进行 zoomIn() 函数的定义:

```
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);
```

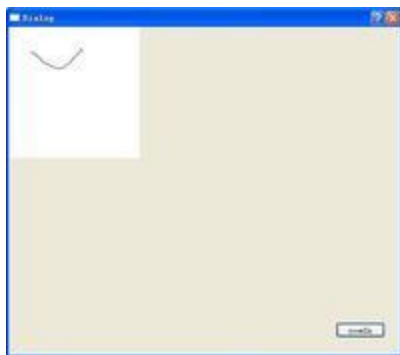
```
pp.drawLine(lastPoint, endPoint);    //根据鼠标指针前后两个位置就行绘制直线
```

```
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
```

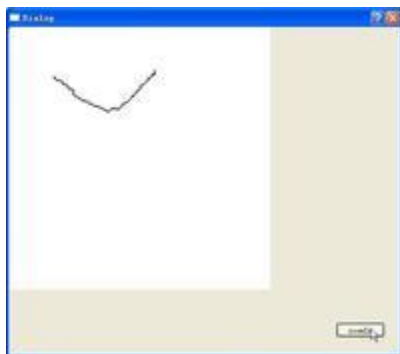
```
QPainter painter(this);  
painter.scale(scale, scale); //进行放大操作  
painter.drawPixmap(0, 0, pix);  
}
```

这时运行程序。

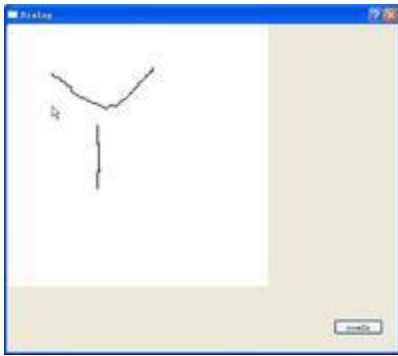
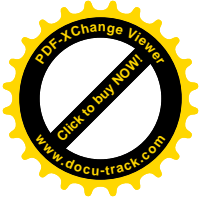
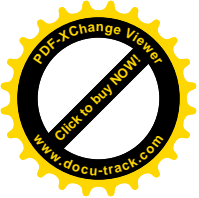
先随意画一个图形，如下图。



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。



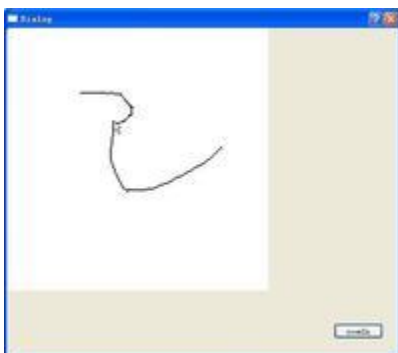
有了前面一节的知识，我们就不难理解出现这个问题的原因了。窗口的坐标扩大了，但是画布的坐标并没有扩大，而我们画图用的坐标值是鼠标指针的，鼠标指针又是获取的窗口的坐标值。现在窗口和画布的同一点的坐标并不相等，所以就出现了这样的问题。

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;

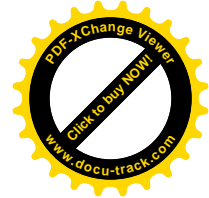
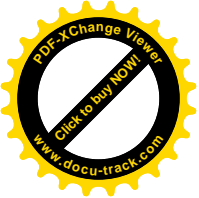
    QPainter painter(this);
    painter.scale(scale, scale); //进行放大操作
    painter.drawPixmap(0, 0, pix);
}
```

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。

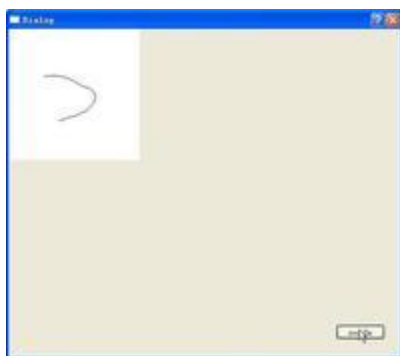
下面我们简单说说另一种方法。

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;

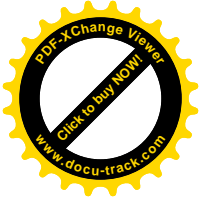
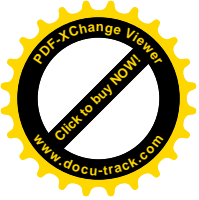
    QPainter painter(this);
    painter.drawPixmap(0, 0, pix);
}
```

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
        QPainter pter(&copyPix);
        pter.scale(scale, scale);
        pter.drawPixmap(0, 0, pix); //将以前画布上的内容复制
        pix = copyPix; //将放大后的内容再复制回原来的画
        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，你可以查看绘图软件的教程（[链接过去](#)）。

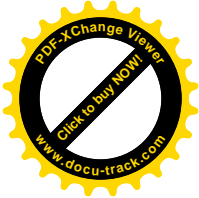
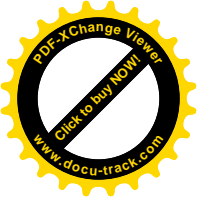
十九、Qt 2D 绘图（九）双缓冲绘图简介（原创）

2010-02-12 18:32

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux>
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。

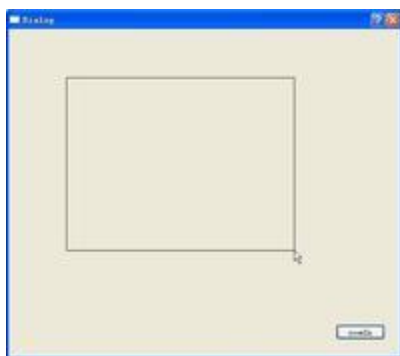


第一步:

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,y,w,h;
    x = lastPoint.x();
    y = lastPoint.y();
    w = endPoint.x() - x;
    h = endPoint.y() - y;
    painter.drawRect(x,y,w,h);
}
```

然后运行，效果如下。



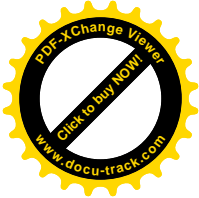
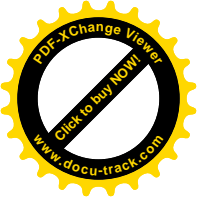
这时我们已经可以拖出一个矩形了，但是这样直接在窗口上绘图，以前画的矩形是不能保存住的。所以我们下面加入画布，在画布上进行绘图。

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:

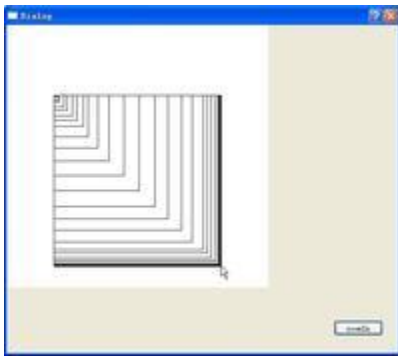
```
void Dialog::paintEvent(QPaintEvent *)
{
    int x,y,w,h;
    x = lastPoint.x();
    y = lastPoint.y();
    w = endPoint.x() - x;
    h = endPoint.y() - y;
    QPainter pp(&pix);
```



```
pp.drawRect(x, y, w, h);

QPainter painter(this);
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
```

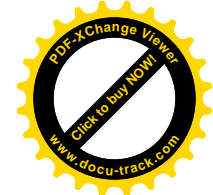
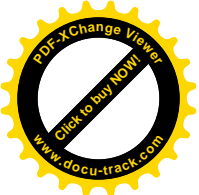
然后在 dialog.cpp 中的构造函数里进行变量初始化：

```
isDrawing = false;
```

最后更改函数如下：

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

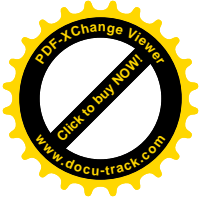
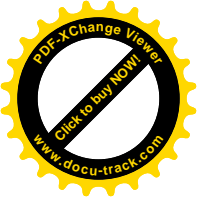
```
    QPainter pp(&tempPix);
    pp.drawRect(x, y, w, h);
    painter.drawPixmap(0, 0, tempPix);
}
else
{
    QPainter pp(&pix);
    pp.drawRect(x, y, w, h);
    painter.drawPixmap(0, 0, pix);
}
}

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，一个是那个绘图软件（[链接到那里](#)），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，就是俄罗斯方块程序（[链接到那里](#)），那个是对这些知识的应用。如果你有兴趣，可以看一下。

二十、Qt 2D 绘图（十）图形视图框架简介（原创）

2010-02-21 13:35

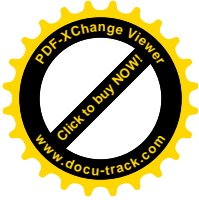
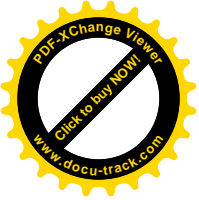
声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

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：

<http://hi.baidu.com/yafeilinux/blog/item/f7040630723a0612eac4af20.html>
1

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



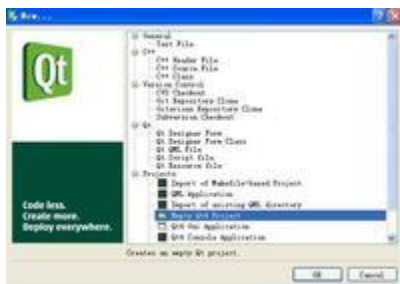
以这里我们只是提及一下，让一些刚入门的朋友知道有这样一个可用的框架。

最简单的使用：

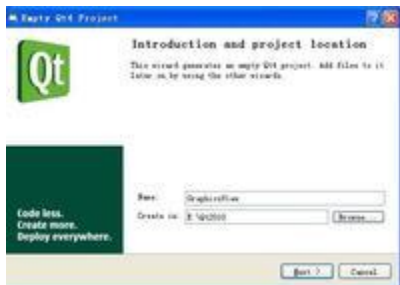
The QGraphics View Framework 包含三个大类：QGraphicsItem 项类（或者叫做图元类），QGraphicsScene 场景类，和 QGraphicsView 视图类。

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：



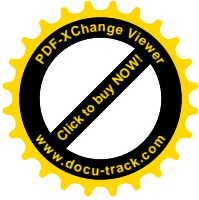
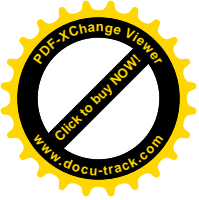
2. 更改工程名和存放路径。



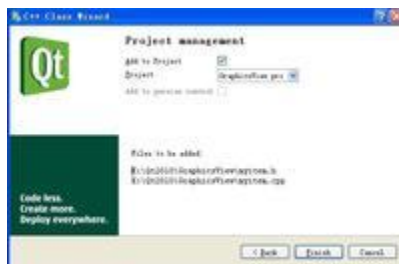
3. 然后新建 C++ 类。



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:

```
#ifndef MYITEM_H
#define MYITEM_H

#include <QWidget>

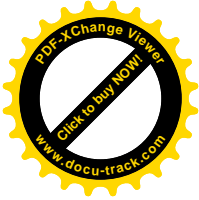
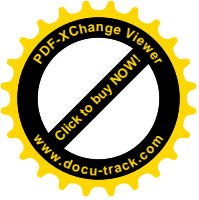
class MyItem : public QWidget
{
    Q_OBJECT

public:
    MyItem()
    {
        // 这里可以添加一些初始化代码, 比如设置一些属性等
        QPushButton *button = new QPushButton("button", this);
        connect(button, SIGNAL(clicked()), this, SLOT(slotClicked()));
    }

private:
    void slotClicked()
    {
        // 这里可以添加一些槽函数代码
    }
};

#endif // MYITEM_H
```

myitem.cpp 文件的内容如下:



```
int main() {
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

main.cpp 的内容如下:

```
int main() {
    QApplication app(argc, argv);
    QWidget widget;
    widget.setWindowTitle("Qt 4.6.2");
    widget.resize(200, 200);
    widget.show();
    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。

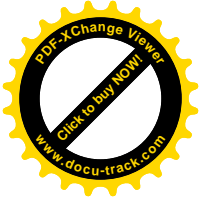
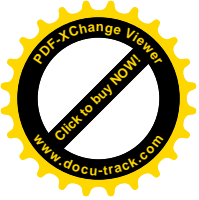
二十一、Qt 数据库 (一) 简介 (原创)

2010-03-02 12:03

声明: 本文原创于 yafeilinux 的百度博客, <http://hi.baidu.com/yafeilinux> 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 QSql 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 QSql 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：

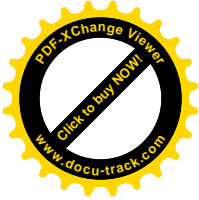
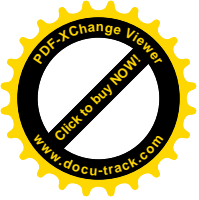
Driver Type	Description
QDB2	IBM DB2
QIBASE	Borland InterBase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTD5	Sybase Adaptive Server

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。

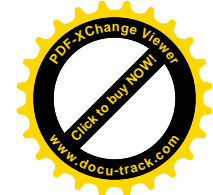
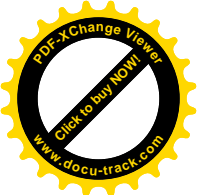
```
#include <QtCore/QCoreApplication>
#include <QtSql>

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");
```

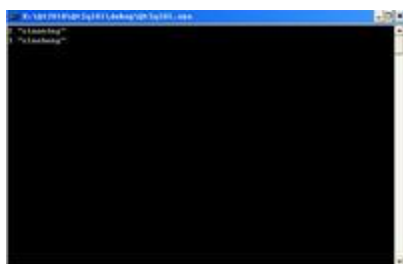


```
//查找表中 id >=2 的记录 id 项和 name 项的值
while(query.next()) //query.next() 指向查找到的第一条记录，然后每次后移一条记录
{
    int ele0 =
query.value(0).toInt(); //query.value(0) 是 id 的值，将其转换为 int 型
    QString ele1=query.value(1).toString();
    qDebug() << ele0 <<ele1 ; //输出两个值
}

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。



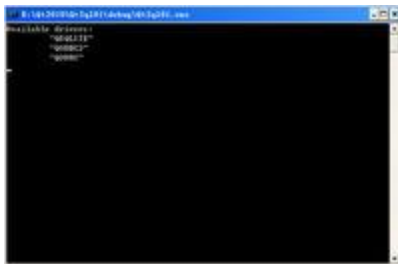
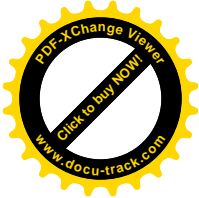
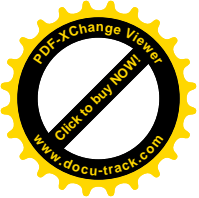
5. 我们可以将主函数更改如下。

```
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。

二十二、Qt 数据库（二）添加 MySQL 数据库驱动插件（原创）

2010-03-08 18:06

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

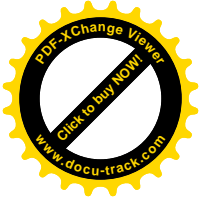
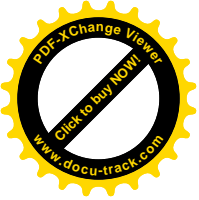
我们进入 Qt Creator 的安装目录，然后进入相对应的文件夹下，比方我这里是

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：



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。

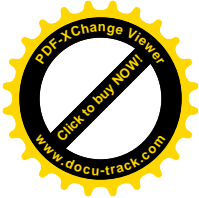
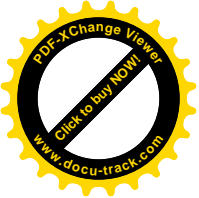
我们可以到 MySQL 的官方主页 <http://www.mysql.com> 进行下载最新的 MySQL 的 windows 版本，现在具体的下载页面地址为：

<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。

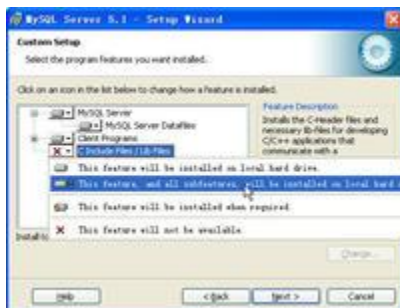
当然你也可以到中文网站上进行下载：<http://www.mysql.cn/>，随便下一个 windows 的版本就行。

2. 安装软件。

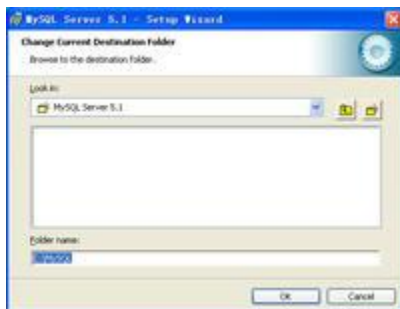
我们选择定制安装 Custom。



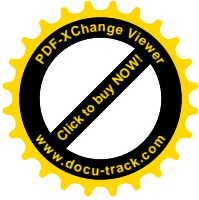
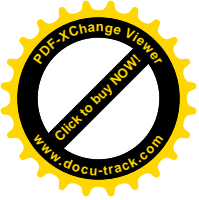
然后选中安装 Include 文件和 Lib 文件。



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。

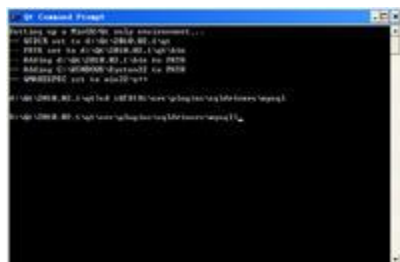


3. 进行编译。

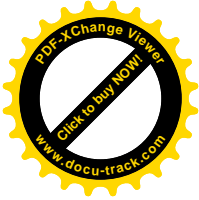
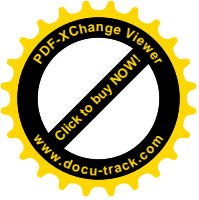
我们在桌面上开始菜单中找到 Qt Creator 的菜单，然后打开 Qt Command Prompt。



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。

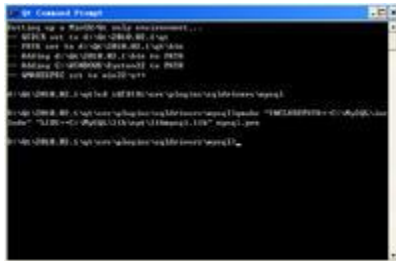


然后输入第二条命令：

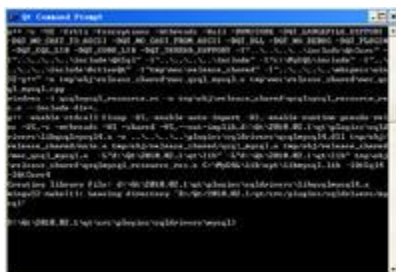


```
qmake "INCLUDEPATH+=C:\MySQL\include"  
"LIBS+=C:\MySQL\lib\opt\libmysql.lib" mysql.pro
```

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

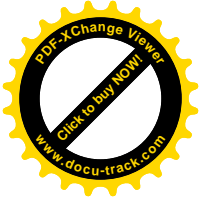
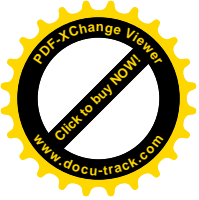
其内容如下：



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。

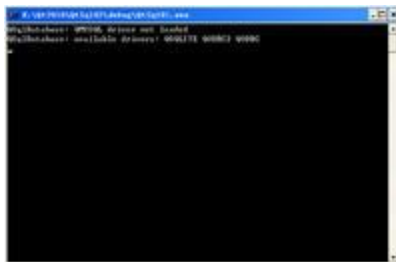


```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL"); //添加数据库驱动

    return a.exec();
}
```

运行程序，效果如下。



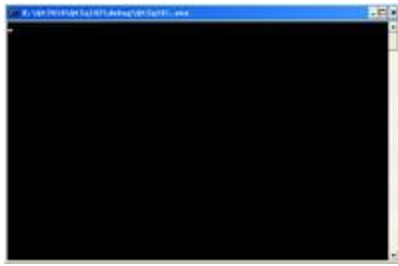
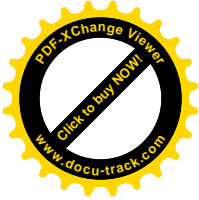
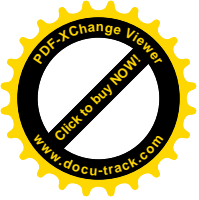
这里提示：**QSqlDatabase: QMYSQL driver not loaded**。

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。

如下图：



3. 这时再运行程序，就没有提示了。



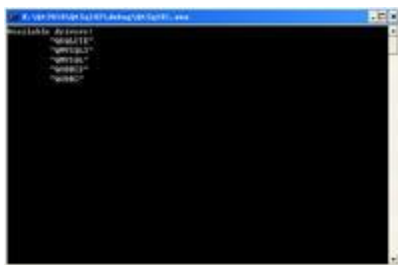
4. 我们再将主函数更改一下，测试这时可用的数据库驱动。

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

    qDebug() << "Available drivers:";
    QStringList drivers = QSqlDatabase::drivers();
    foreach(QString driver, drivers)
        qDebug() << "\t" << driver;

    return a.exec();
}
```

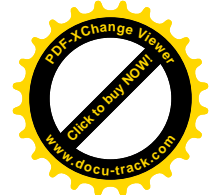
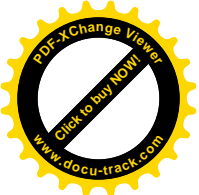
运行效果如下：



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。



```
        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：

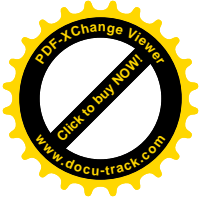
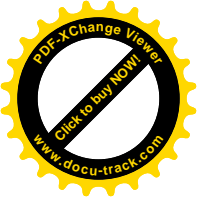
```
#include <QtGui/QApplication>
#include "widget.h"
#include "connection.h"

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。



id	name
0	first
1	second
2	third
3	fourth
4	fifth

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。

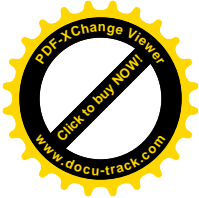
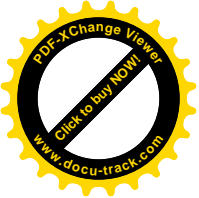
```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
        qDebug() << query.value(0).toInt() <<
query.value(1).toString();
    }
}
```

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。

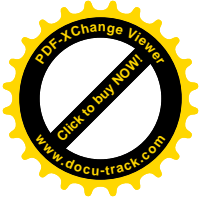
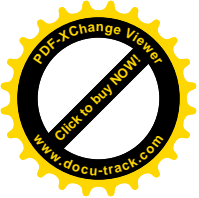
`record()` : 获得现在指向的记录。

`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。

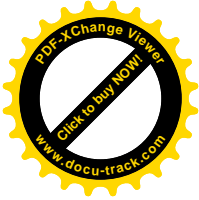
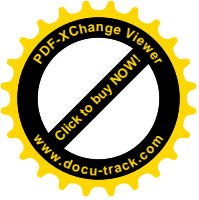
需要说明，当刚执行完 `query.exec("select * from student");` 这行代码时，`query` 是指向结果集以外的，我们可以利用 `query.next()`，当第一次执行这句代码时，`query` 便指向了结果集的第一条记录。当然我们也可以利用 `seek(0)` 函数或者 `first()` 函数使 `query` 指向结果集的第一条记录。但是为了节省内存开销，推荐的方法是，在 `query.exec("select * from student");` 这行代码前加上 `query.setForwardOnly(true);` 这条代码，此后只能使用 `next()`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

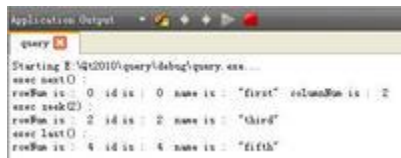


```
}
```

然后在 widget.cpp 文件中添加头文件。

```
#include <QSqlRecord>
```

运行程序，结果如下：



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

2010-03-12 21:51

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 转载请注明出处。

接着上一篇教程。

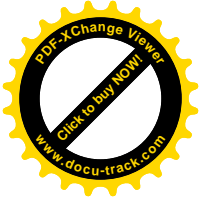
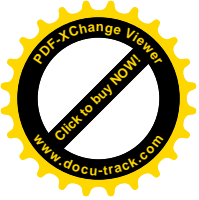
二，在 SQL 语句中使用变量。

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)");

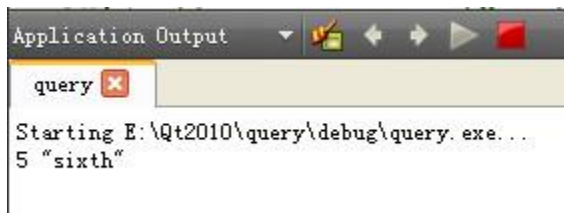
    query.bindValue(0, 5);
    query.bindValue(1, "sixth");
    query.exec();

    //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，而后再利用 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();
```

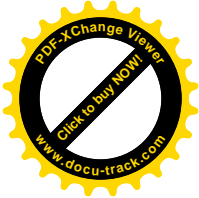
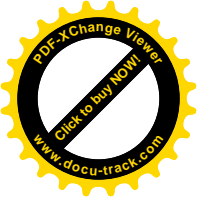
我们也可以利用 addBindValue() 函数，这样就可以省去编号，它是按顺序给属性赋值的，如下：

```
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。**



下面我们就可以利用绑定操作在 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。

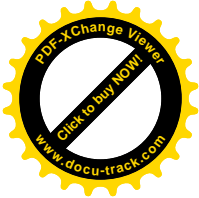
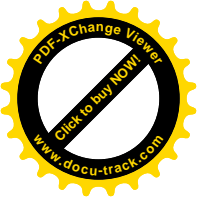
三、批处理操作。

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>` 。

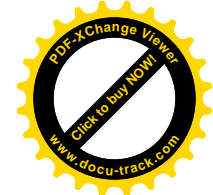
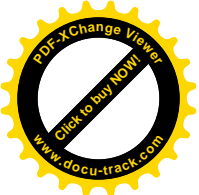
我们在程序中利用列表存储了同一属性的多个值，然后进行了值绑定。最后执行 `execBatch()` 函数进行批处理。注意程序中利用 `QVariant(QVariant::String)` 来输入空值 `NULL`，因为前面都是 `QString` 类型的，所以这里要使用 `QVariant::String` 使格式一致化。

运行效果如下：

```
Application Output
query
Starting E:\Qt2010\query\debug\query.exe...
0 "first"
1 "second"
2 "third"
3 "fourth"
4 "fifth"
10 "xiaoming"
11 "xiaoliang"
12 "xiaogang"
13 ""
```

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：

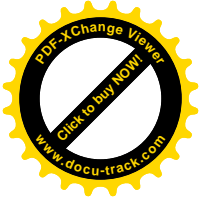
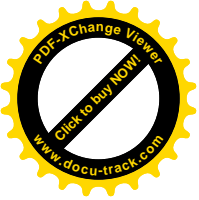
```
void Widget::on_pushButton_clicked()
{
    if(QSqlDatabase::database().driver()->hasFeature(QSqlDriver::T
ransactions))
    {
        //先判断该数据库是否支持事务操作
        QSqlQuery query;
        if(QSqlDatabase::database().transaction()) //启动事务
操作
    {
        //
        //下面执行各种数据库操作
        query.exec("insert 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：



```
Application Output
query
Starting E:\Qt2010\query\debug\query.exe...
0 "first"
2 "third"
3 "fourth"
4 "fifth"
14 "hello"
```

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。

二十五、Qt 数据库（五） QSqlQueryModel (原创)

2010-03-22 21:11

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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。

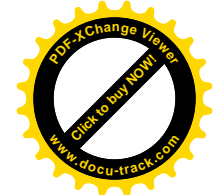
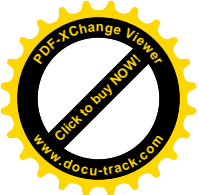
QSqlQueryModel 类为 SQL 的结果集提供了一个只读的数据模型，下面我们先利用这个类进行一个最简单的操作。

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
vchar)");
        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。

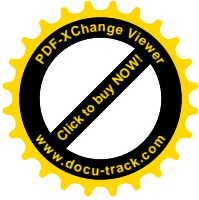
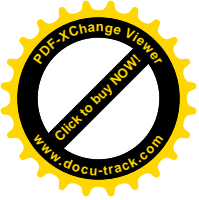
然后进入 `main.cpp`，将其内容更改如下：

```
#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”);” 用来查询整个 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”按钮，效果如下：



我们在工程文件夹下查看数据库文件：

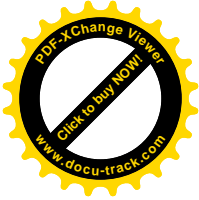
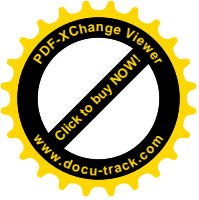


下面我们利用这个模型来操作数据库。

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();
```

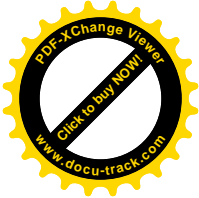
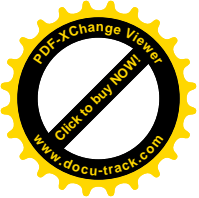
这样就可以输出表中的值了，你可以运行程序测试一下。

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：

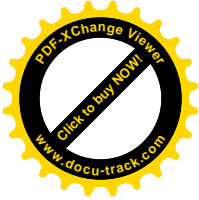
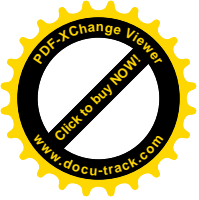


发现这次新的记录已经添加了。在上面我们执行了添加记录的 SQL 语句，但是在添加记录之前，就已经进行显示了，所以我们的更新没能动态的显示出来。为了能让其动态地显示我们的更新，我们可以将函数更改如下：

```
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。

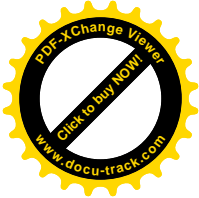
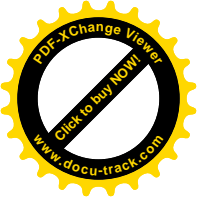
1. 我们在工程中添加 C++ Class，然后 Class name 设为 QSqlQueryModel，Base Class 设为 QSqlQueryModel，如下：



2. 我们将 mysqlquerymodel.h 中的内容更改如下：

```
class QSqlQueryModel : public QSqlQueryModel
{
public:
    QSqlQueryModel();
    //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);

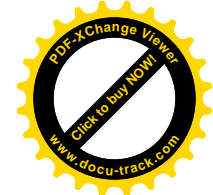
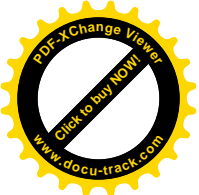
    QVariant data(const QModelIndex &item, int role=Qt::DisplayRole)
const;
    //
private:
```



```
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);

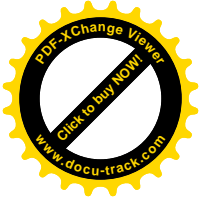
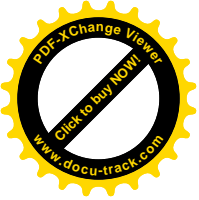
    if (role == Qt::TextColorRole && index.column() == 0)
        return QVariantFromValue(QColor(Qt::red)); //第一个属性
    的字体颜色为红色
    return value;
}
```

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。

二十六、Qt 数据库（六） QSqlTableModel （原创）

2010-03-26 15:52

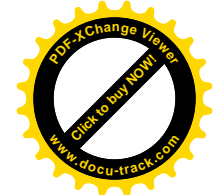
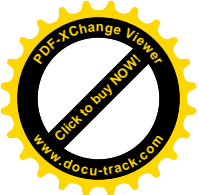
声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux>
转载请注明出处。

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 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>
#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 varchar)"));
    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
```

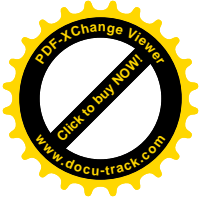
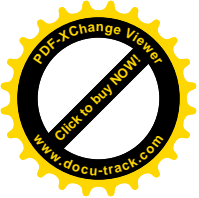
为了在数据库中能使用中文，我们使用了 `QObject` 类的 `tr()` 函数。而在下面的 `main()` 函数中我们也需要添加相应的代码来支持中文。

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，所以我们在这里声明它。

我们到 widget.cpp 文件中的构造函数里添加如下代码：

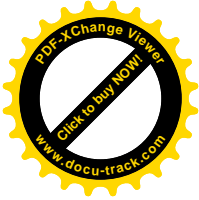
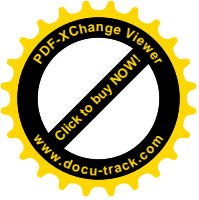
```
model = new QSqlTableModel(this);
model->setTable("student");
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
model->select(); //选取整个表的所有行
// model->removeColumn(1); //不显示 name 属性列, 如果这时添加记录, 则该
属性的值添加不上
ui->tableView->setModel(model);
//
ui->tableView->setEditTriggers(QAbstractItemView::NoEditTriggers);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

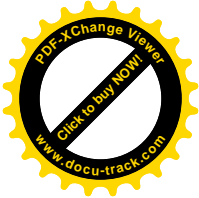
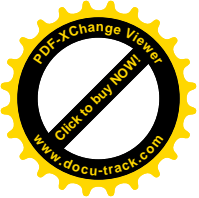
我们需要在 `widget.cpp` 文件中添加头文件：

```
#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。当然这也是因为我们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。

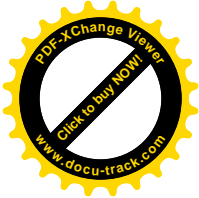
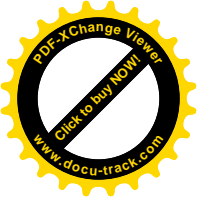
2. 我们进入“返回全表”按钮的单击事件槽函数，更改如下：

```
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
    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：

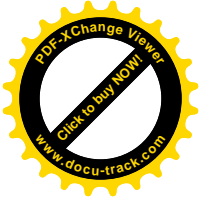
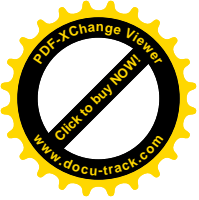


这是按下“按 id 降序排列”按钮后的效果。

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  
要删除当前行吗?"),  
QMessageBox::Yes, QMe  
ssageBox::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：

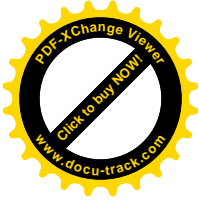
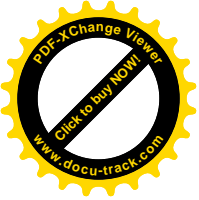


我们点击第二行，然后单击“删除选中行”按钮，出现了警告框。这时你会发现，表中的第二行前面出现了一个小感叹号，表明该行已经被修改了，但是还没有真正的在数据库中修改，这时的数据有个学名叫脏数据 (Dirty Data)。当我们按钮“ Yes ”按钮后数据库中的数据就会被删除，如果按下“ No ”，那么更改就会取消。

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。

二十七、Qt 数据库（七）

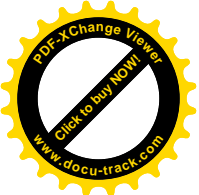
QSqlRelationalTableModel（原创）

2010-03-26 17:34

声明：本文原创于 yafeilinux 的百度博客，<http://hi.baidu.com/yafeilinux> 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之上添加了外键（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 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
vchar, 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)");

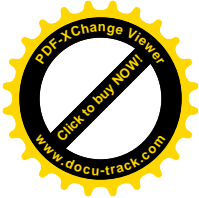
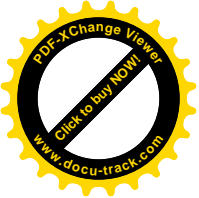
    query.exec("create table course (id int primary key, name vchar,
teacher vchar)");
    query.exec("insert into course values
(1,'Math','yafeilinux1')");
    query.exec("insert into course values
(2,'English','yafeilinux2')");
    query.exec("insert into course values
(3,'Computer','yafeilinux3')");
    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:

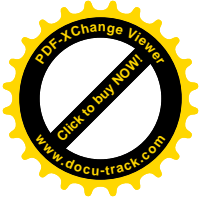
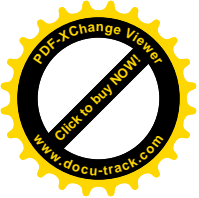
```
    model = new QSqlRelationalTableModel(this);  
    model->setEditStrategy(QSqlTableModel::OnFieldChange); //属性  
变化时写入数据库  
    model->setTable("student");  
    model->setRelation(2, QSqlRelation("course", "id", "name"));  
    //将 student 表的第三个属性设为 course 表的 id 属性的外键, 并将其  
显示为 course 表的 name 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。