

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。



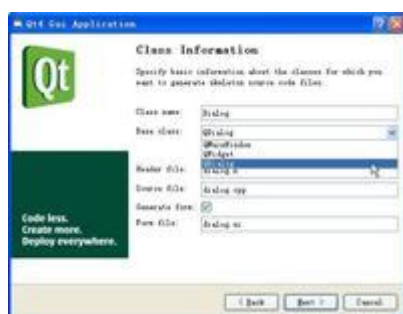
5. 下面输入工程名和要保存到的文件夹路径。我们这里的工程名为 helloworld。



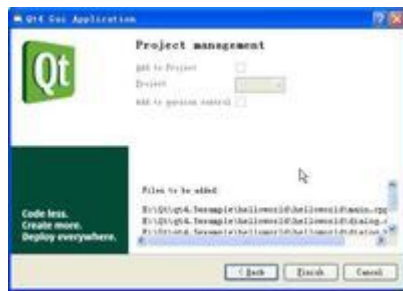
6. 这时软件自动添加基本的头文件，因为这个程序我们不需要其他的功能，所以直接点击 Next。



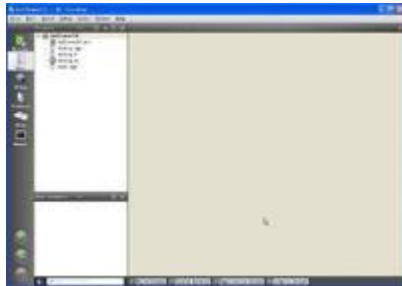
7. 我们将 base class 选为 QDialog 对话框类。然后点击 Next。



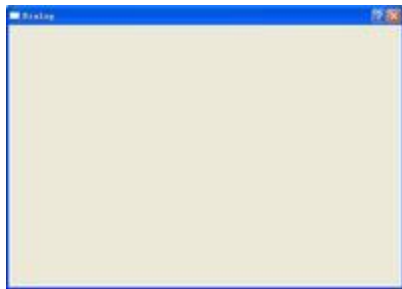
8. 点击 Finish，完成工程的建立。



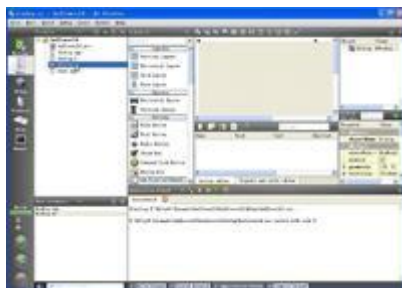
9. 我们可以看见工程中的所有文件都出现在列表中了。我们可以直接按下下面的绿色的 run 按钮或者按下 Ctrl+R 快捷键运行程序。



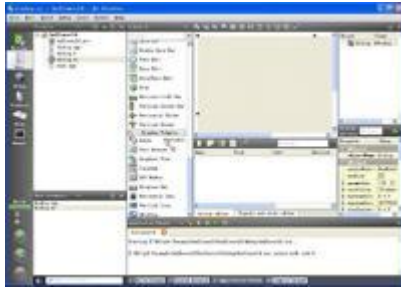
10. 程序运行会出现空白的对话框，如下图。



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。



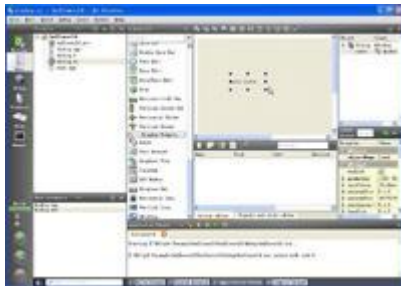
12. 我们在右边的器件栏里找到 Label 标签器件



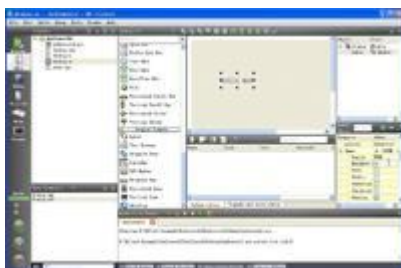
13. 按着鼠标左键将其拖到设计窗口上，如下图。



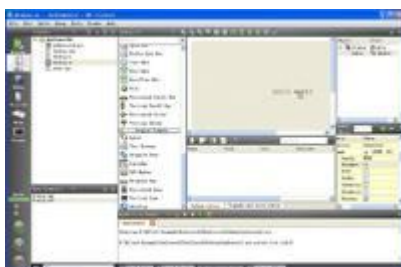
14. 我们双击它，并将其内容改为 helloworld.



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。

二、Qt Creator 编写多窗口程序（原创）

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：

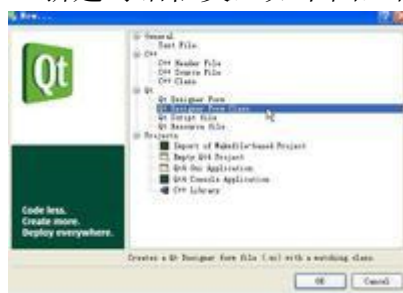
程序里我们先建立一个主工程，作为主界面，然后再建立一个对话框类，将其加入工程中，然后在程序中调用自己新建的对话框类来实现多窗口。

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。



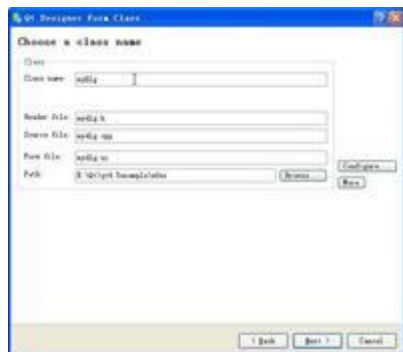
2. 新建对话框类，如下图，在新建中，选择 Qt Designer Form Class。



3. 选择 Dialog without Buttons。



4. 类名设为 myDlg。




```

{
    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;

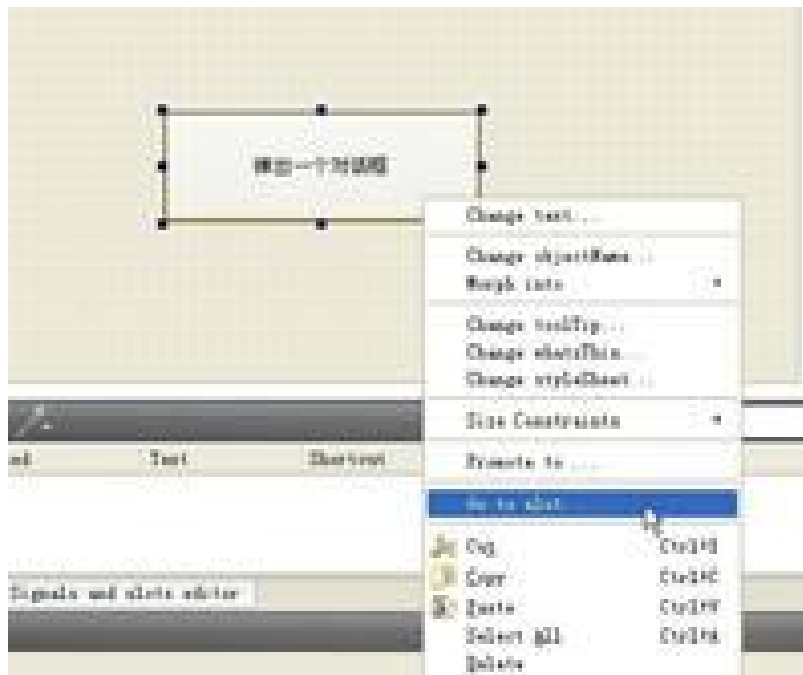
    if(my1.exec()==QDialog::Accepted)
    {
        w.show();
    }
    return a.exec();
}
```

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。

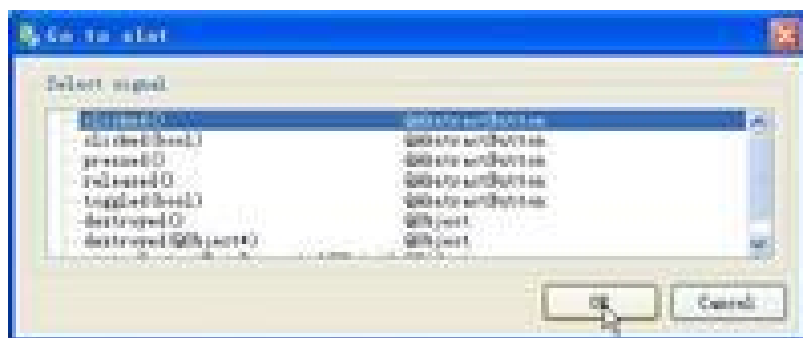
因为 `return a.exec();` 一句表示只要主窗口界面不退出，那么程序就会一直执行。所以只有用第一种方法，将该语句也放到 if 语句中，而在 else 语句中用 `else return 0;`，这样如果 enterBtn 没有被按下，那么程序就会结束执行了。

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）

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：

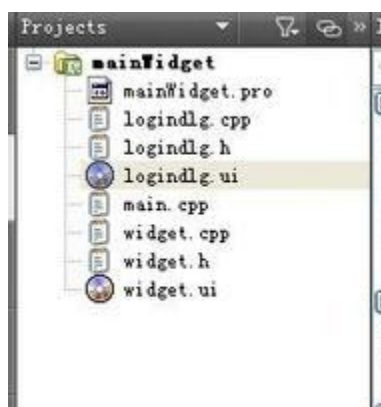
通过上节的多窗口原理实现由登录对话框进入主窗口，而用户名和密码可以用 if 语句进行判断。

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，我们可以使用用户名和密码框清空并且光标自动跳转到用户名输入框，最终的登录按钮的单击事件的槽函数如下：

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

四、Qt Creator 添加菜单图标（原创）

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。

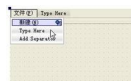
生成的窗口界面如下图。其中最上面的为菜单栏。



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。



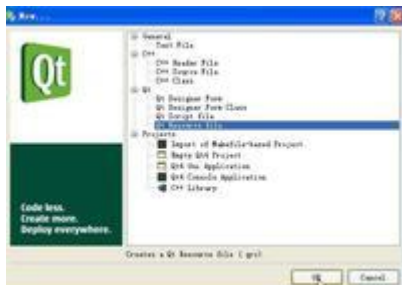
5. 双击这一条，可打开它的编辑对话框。我们看到 Icon 项，这里可以更改“新建”菜单的图标。



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。



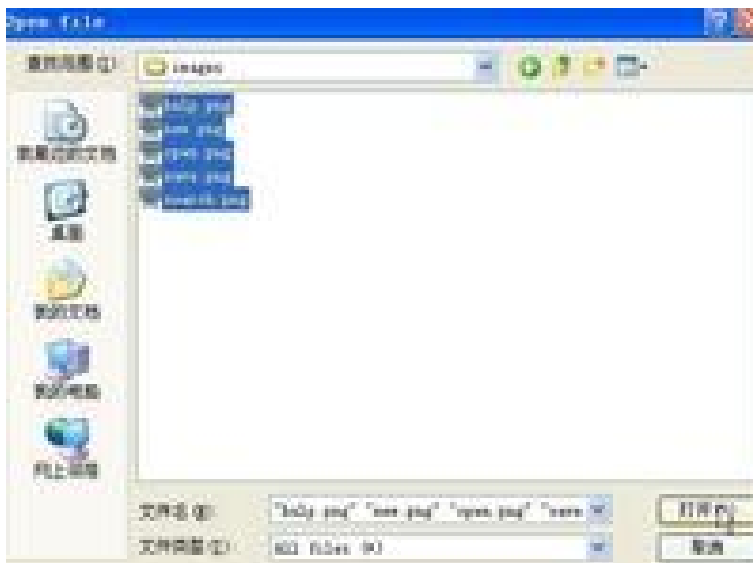
9. 我们最好先在工程文件夹里新建一个文件夹，如 images，然后将需要的图标文件放到其中。



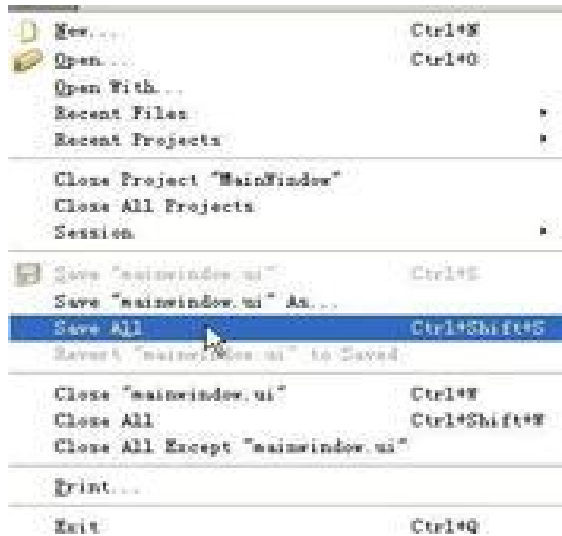
10. 在 Qt Creator 的 menu.qrc 文件中，我们点击 Add 下拉框，选择 Add Prefix。我们可以将生成的/new/prefix 前缀改为其他名字，如/File。



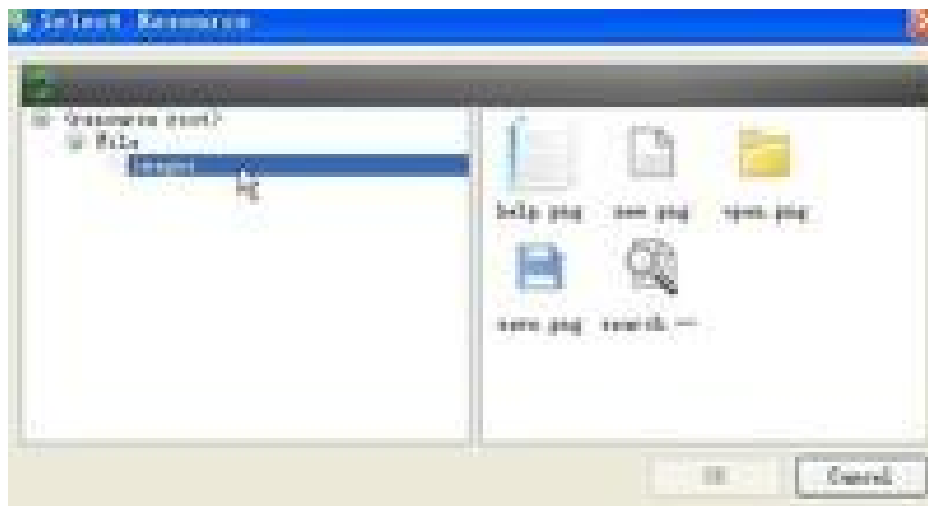
11. 然后再选择 Add 下拉框，选择 Add Files。再弹出的对话框中，我们到新建的 images 文件夹下，将里面的图标文件全部添加过来。



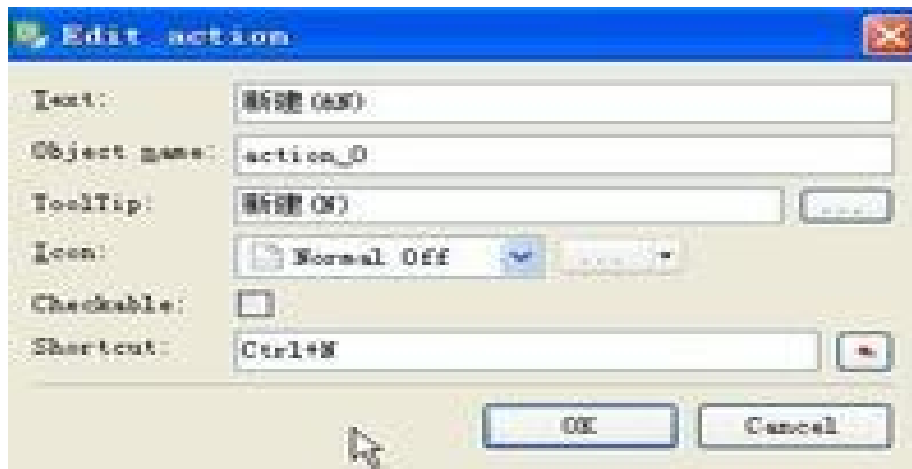
12. 添加完成后，我们在 Qt Creator 的 File 菜单里选择 Save All 选项，保存所做的更改。



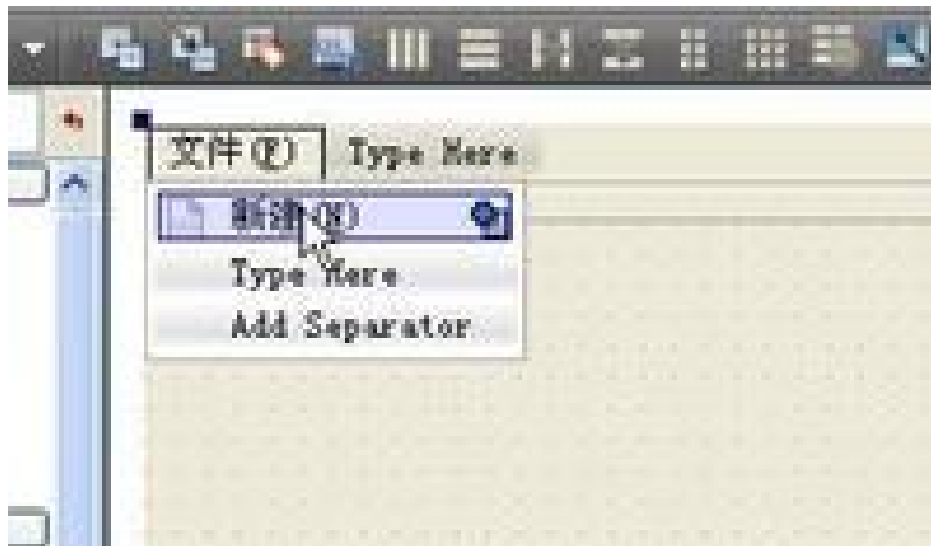
13. 这时再打开资源选择器，可以看到我们的图标都在这里了。(注意：如果不显示，可以按一下上面的 Reload 按钮)



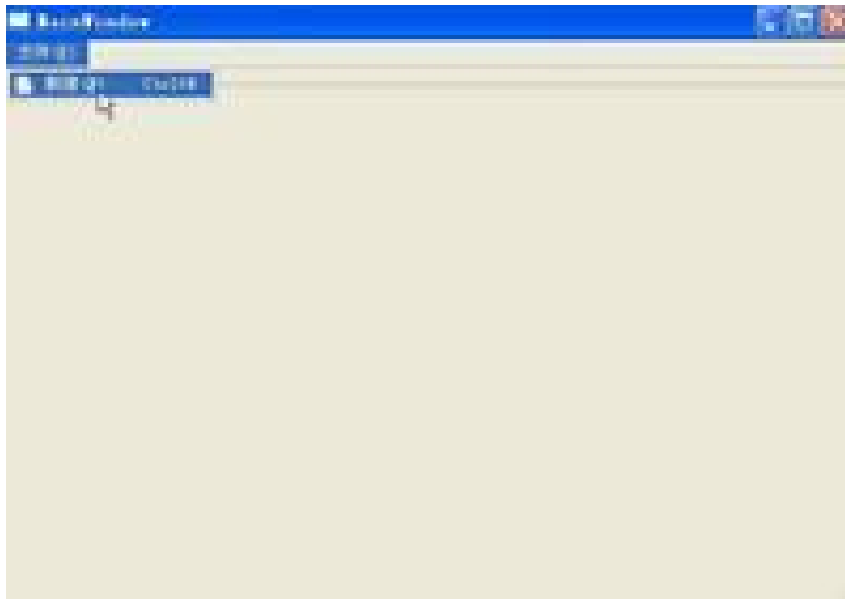
14. 我们将 new.png 作为“新建”菜单的图标，然后点击 Shortcut，并按下 Ctrl+N，便能将 Ctrl+N 作为“新建”菜单的快捷键。



15. 这时打开文件菜单，可以看到“新建”菜单已经有图标了。



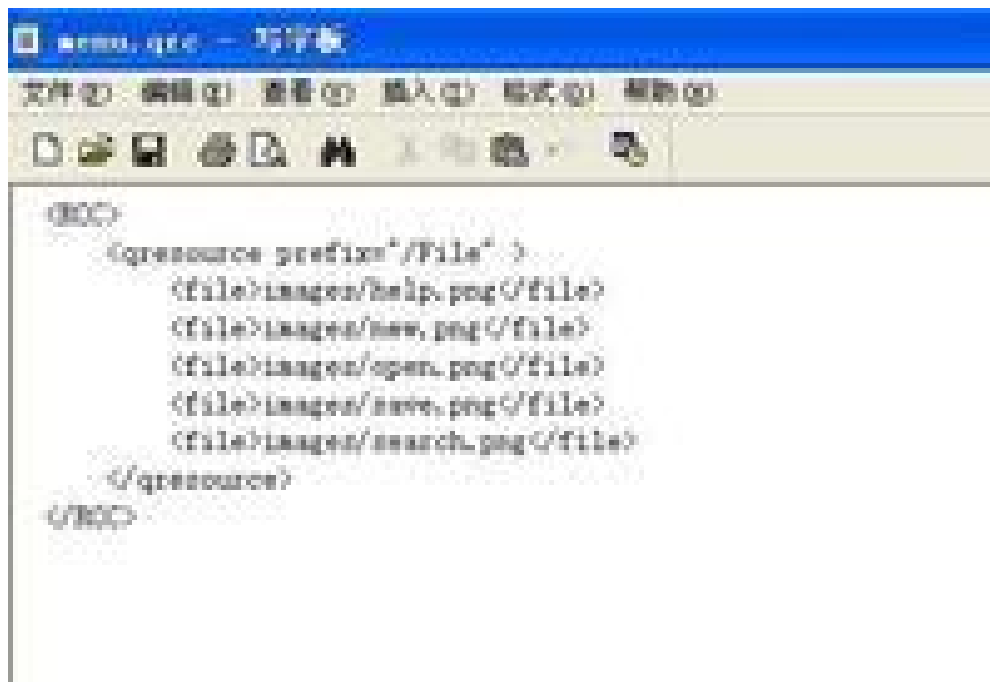
运行程序后效果如下。



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。

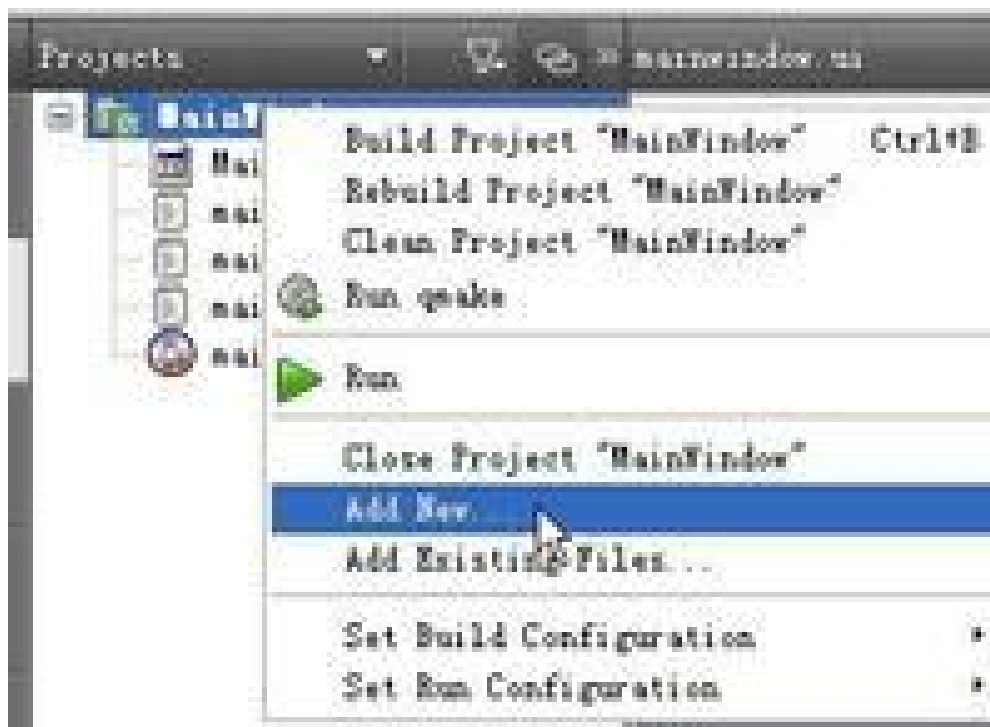


其具体内容如下。



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。



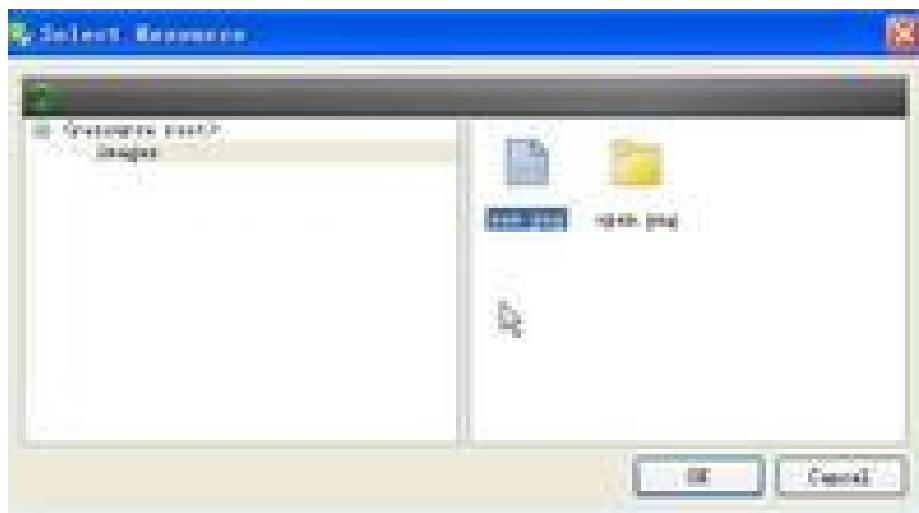
3. 将文件名设置为 menu.qrc。



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）

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。

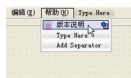
“文件”的子菜单如下图。中间的分割线可以点击 Add Separator 添加。



“编辑”子菜单的内容如下。

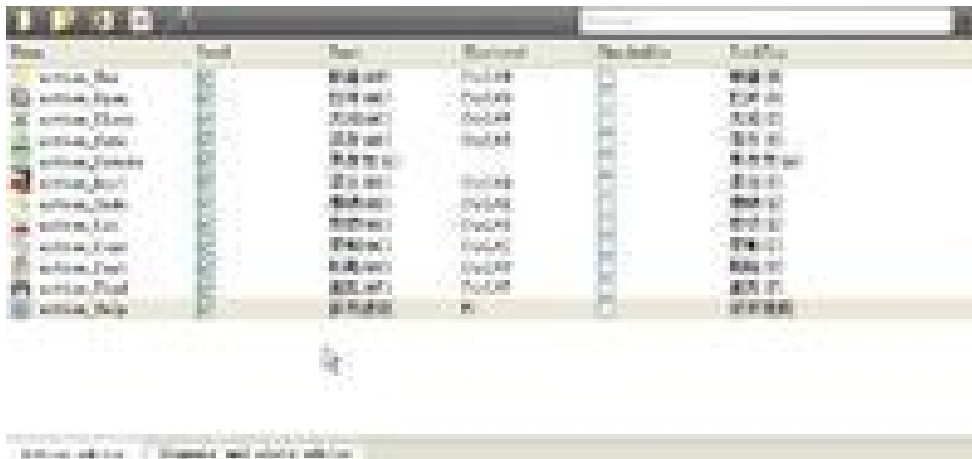


“帮助”子菜单的内容如下。



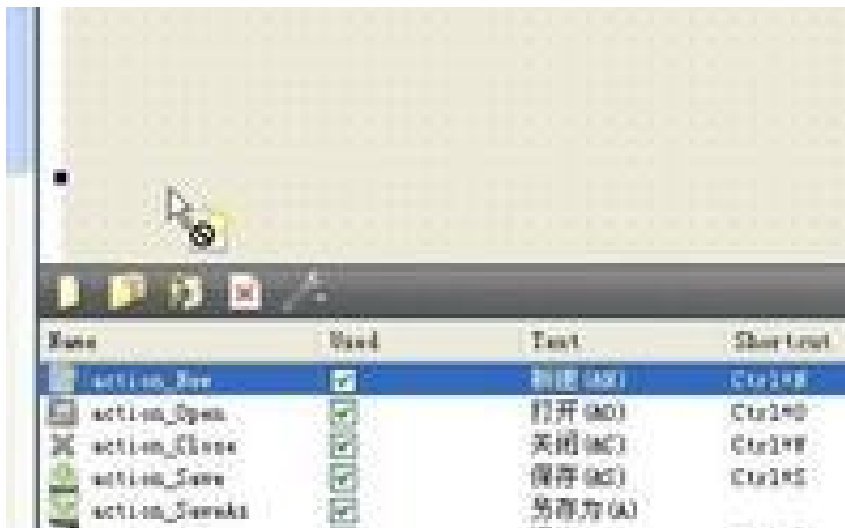
2. 我们在动作编辑器中对各个菜单的属性进行设置。

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。



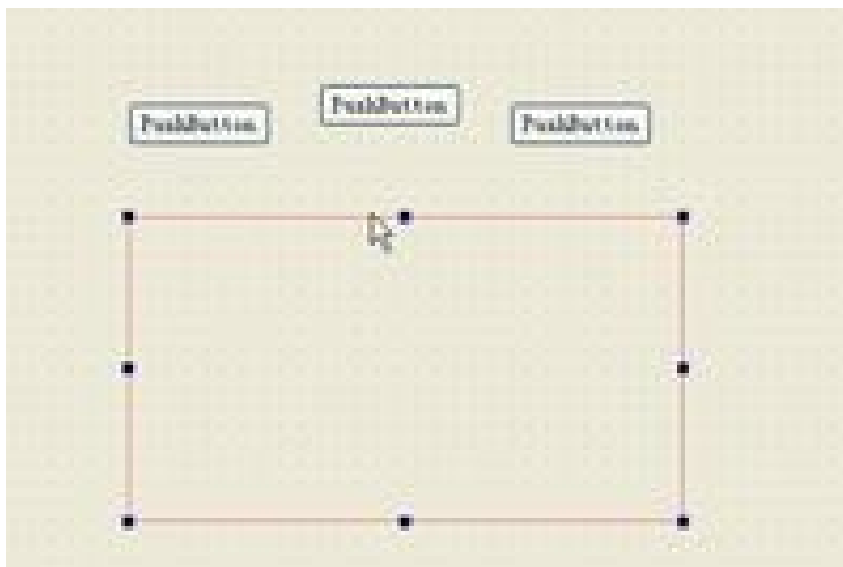
5. 最终效果如下。如果需要删除图标,可以在图标上点击右键选择 Remove action 即可。



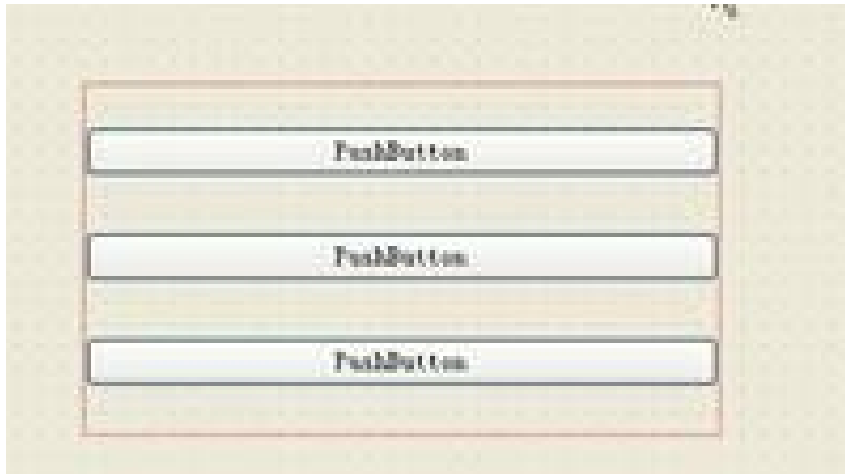
下面简述一下布局管理器。

(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。)

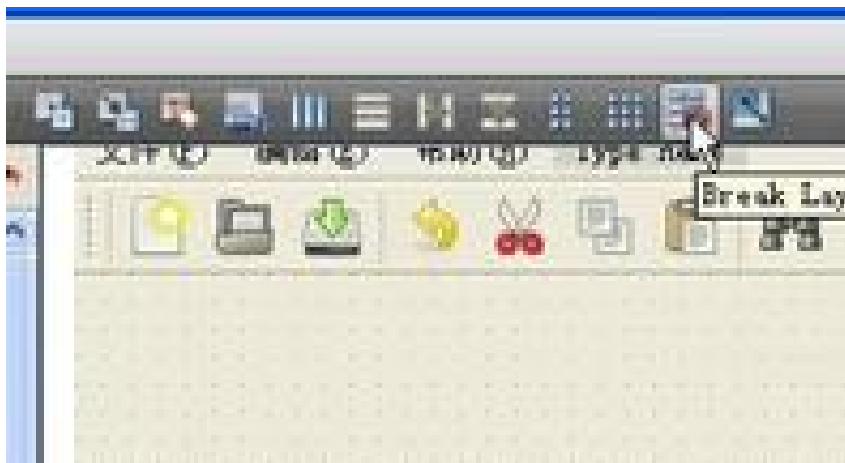
1. 在左边的器件栏里拖入三个 PushButton 和一个 Vertical Layout (垂直布局管理器) 到中心面板。如下图。



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。

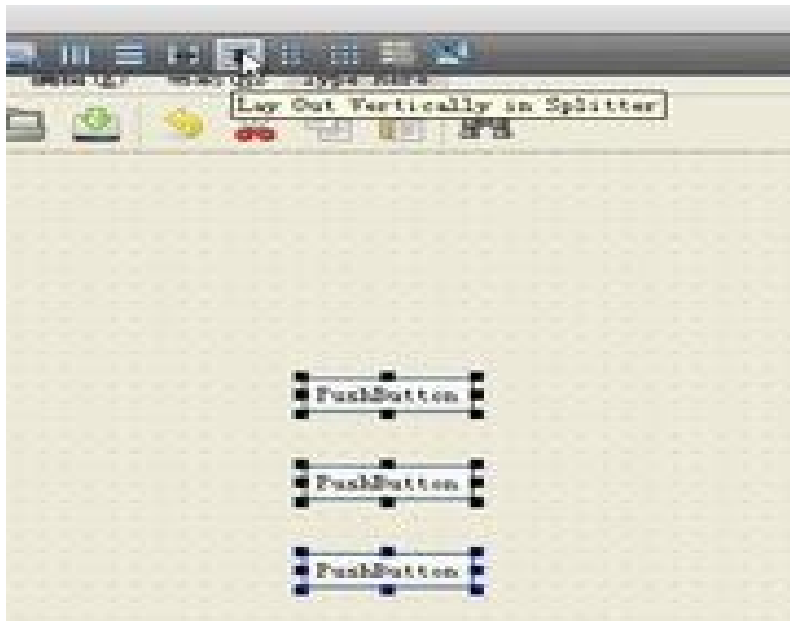


3. 我们将布局管理器整体选中，按下上面工具栏的 Break Layout 按钮，便可取消布局管理器。（我们当然也可以先将按钮移出，再按下 Delete 键将布局管理器删除。）

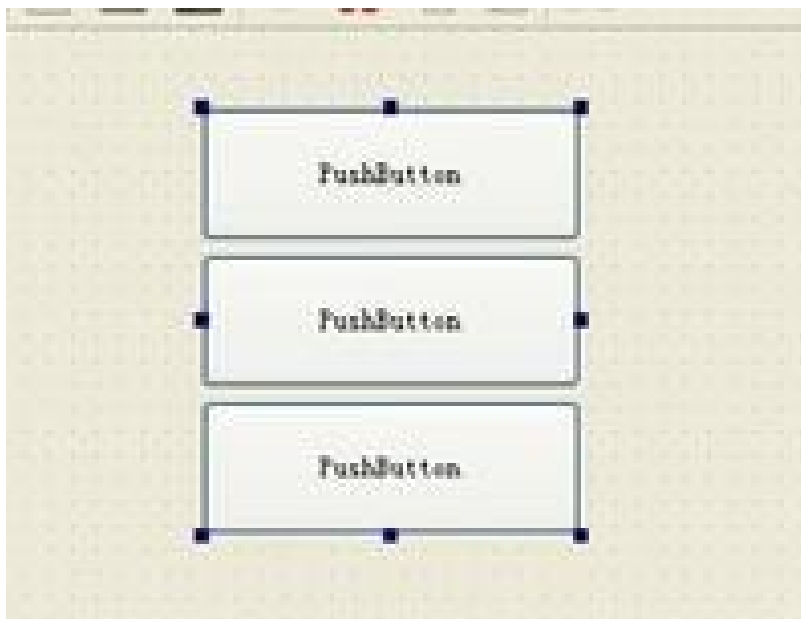


4. 下面我们改用分裂器部件（QSplitter）。

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）。

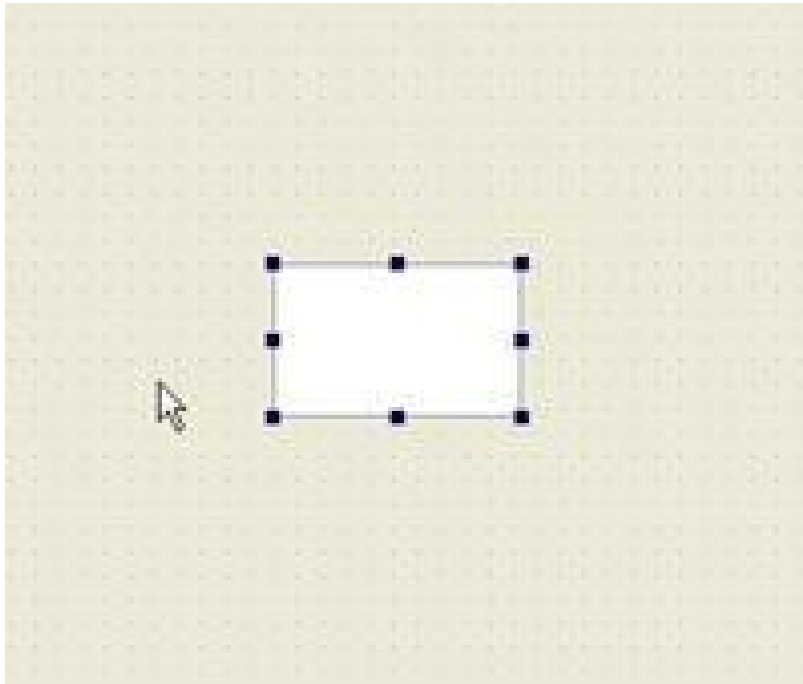


效果如下图。可以看到按钮的大小可以随之改动。这也就是分裂器和布局管理器的分别。

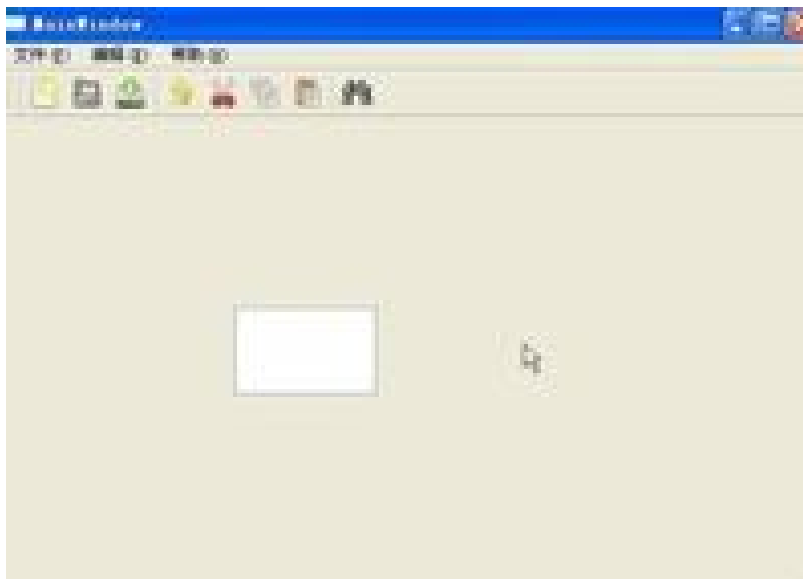


5. 其实布局管理器不但能控制器件的布局，还有个很重要的用途是，它能使器件的大小随着窗口大小的改变而改变。

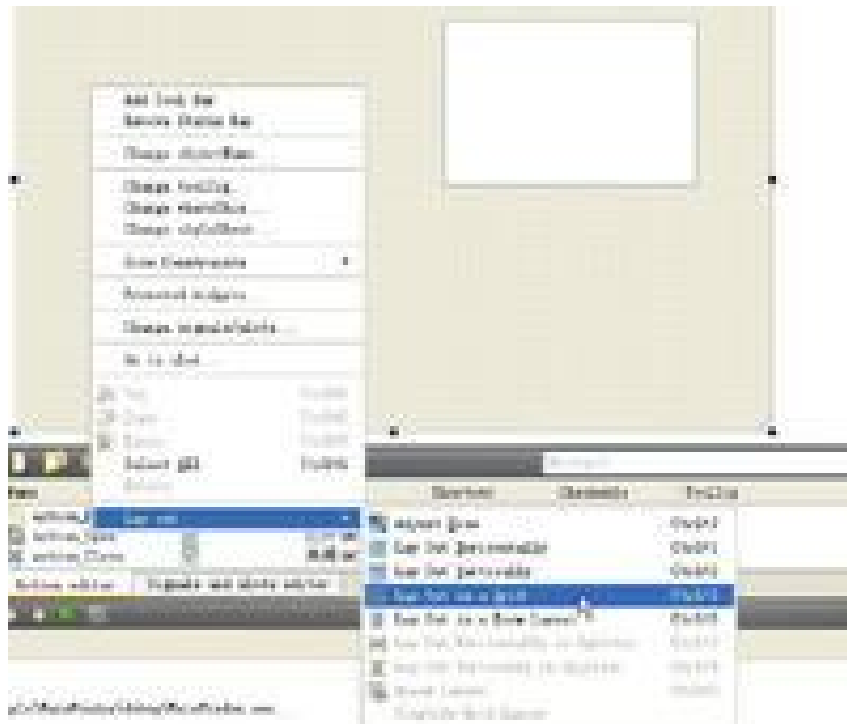
我们先在主窗口的中心拖入一个文本编辑器 Text Edit。



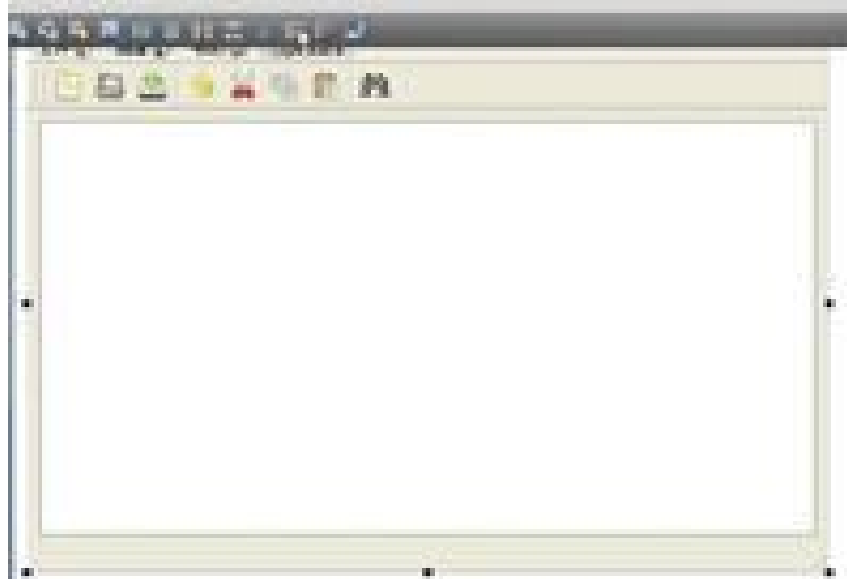
这时直接运行程序，效果如下。可以看到它的大小和位置不会随着窗口改变。



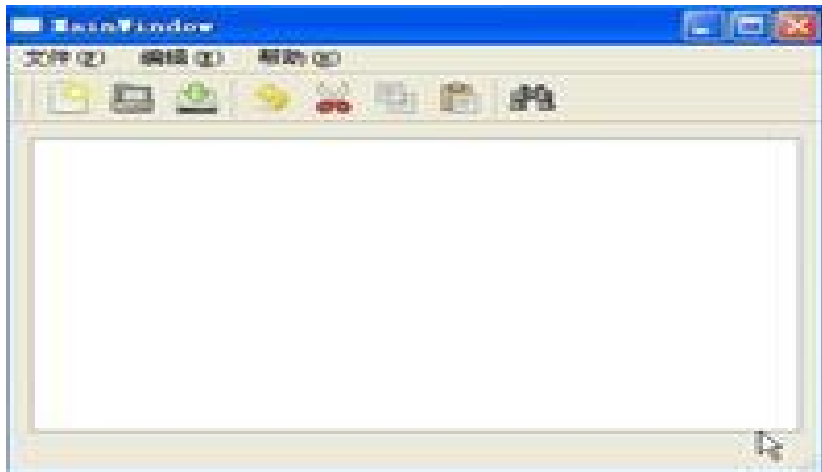
下面我们选中主窗口部件，然后在空白处点击鼠标右键，选择 Layout->Lay Out in a Grid，使整个主窗口的中心区处于网格布局管理器中。



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）

前面已经将界面做好了，这里我们为其添加代码，实现文本编辑的功能。

首先实现新建文件，文件保存，和文件另存为的功能。

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

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

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

新建文件，那么如果有正在编辑的文件，是否需要保存呢？

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

下面开始按这些关系写程序。

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

先在 `main.cpp` 文件中加入以下语句，让程序中可以使用中文。

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

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

这样在程序中使用中文，便能在运行时显示出来了。更改后文件如下图。

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
```

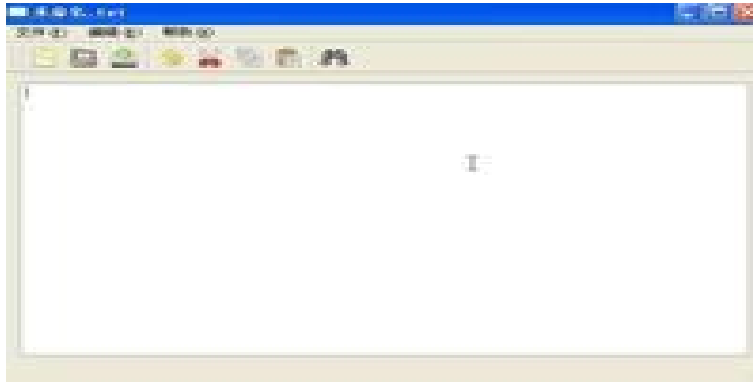
```
QString curFile; //保存当前文件的文件名  
void do_file_New(); //新建文件  
void do_file_SaveOrNot(); //修改过的文件是否保存  
void do_file_Save(); //保存文件  
void do_file_SaveAs(); //文件另存为  
bool saveFile(const QString& fileName); //存储文件
```

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
}
```

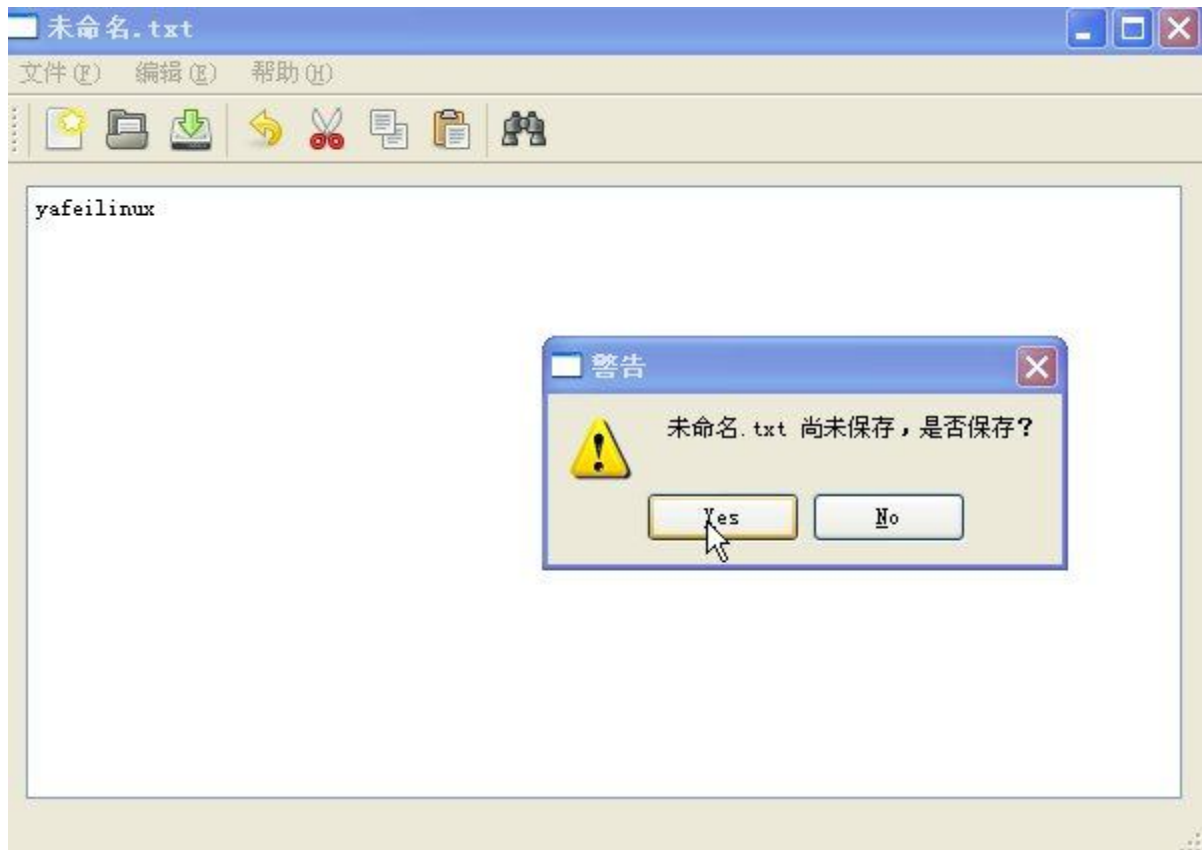
新建文件，先要判断正在编辑的文件是否需要保存。然后将新建的文件标志为未保存过状态。

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。

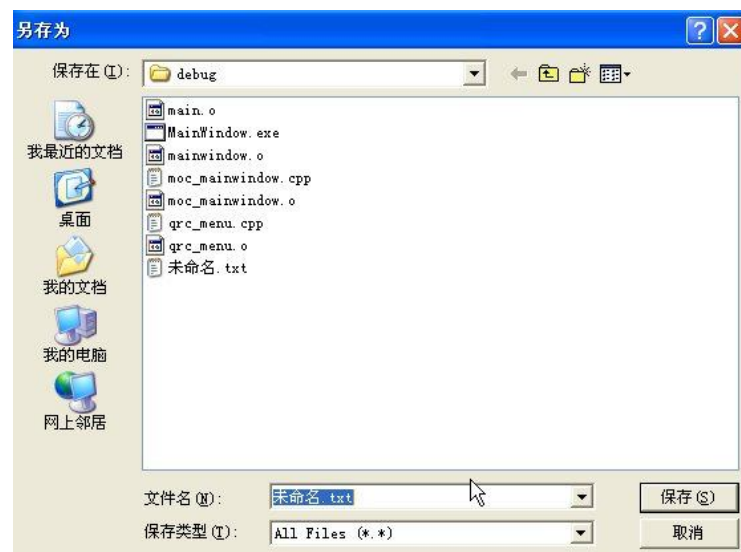
```
void MainWindow::do_file_Save() //保存文件
{
    if(isSaved){ //如果文件已经被保存过，直接保存文件
        saveFile(curFile);
    }
    else{
        do_file_SaveAs(); //如果文件是第一次保存，那么调用另存为
    }
}
```

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。



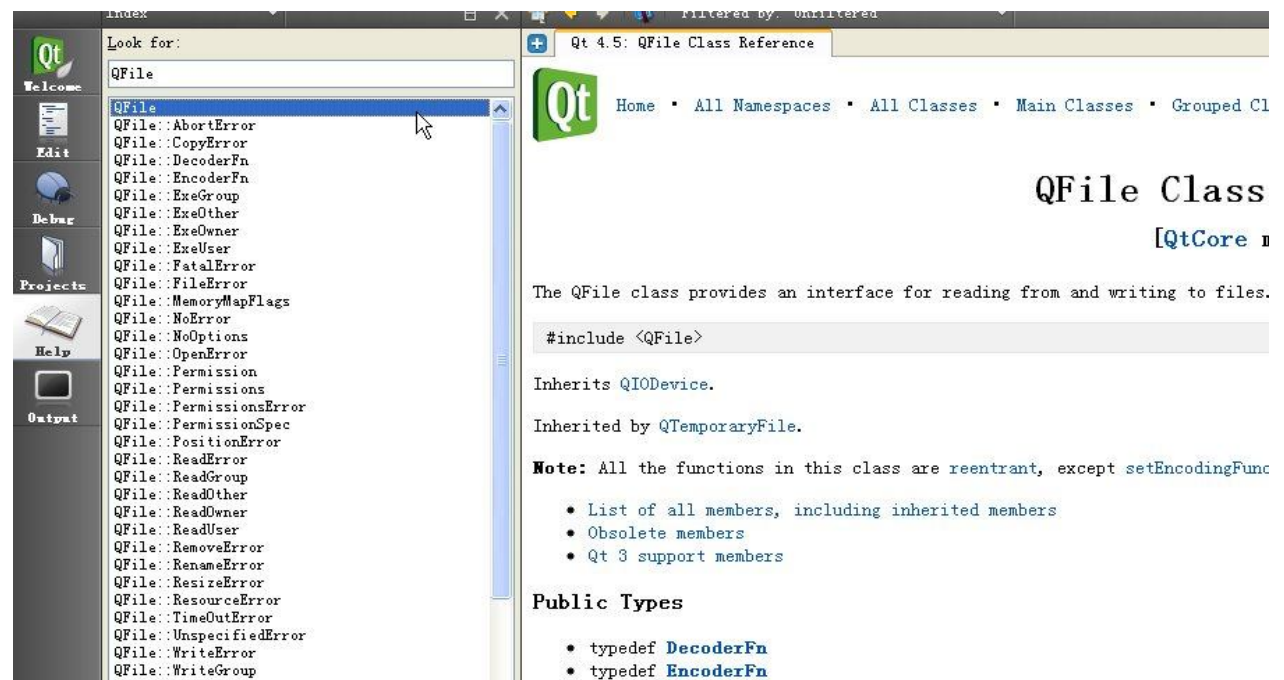
8. 下面是实际文件存储操作的函数定义。

```
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;
}
```

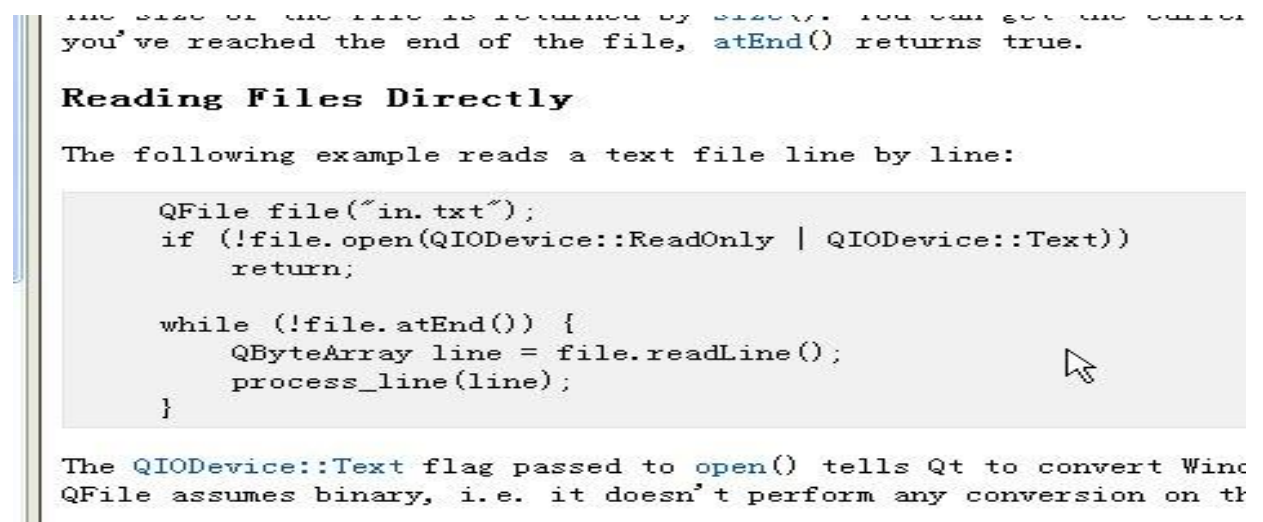
这个函数实现将文本文件进行存储。下面我们对其中的一些代码进行讲解。

`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。
//



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

//

right:

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

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

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

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

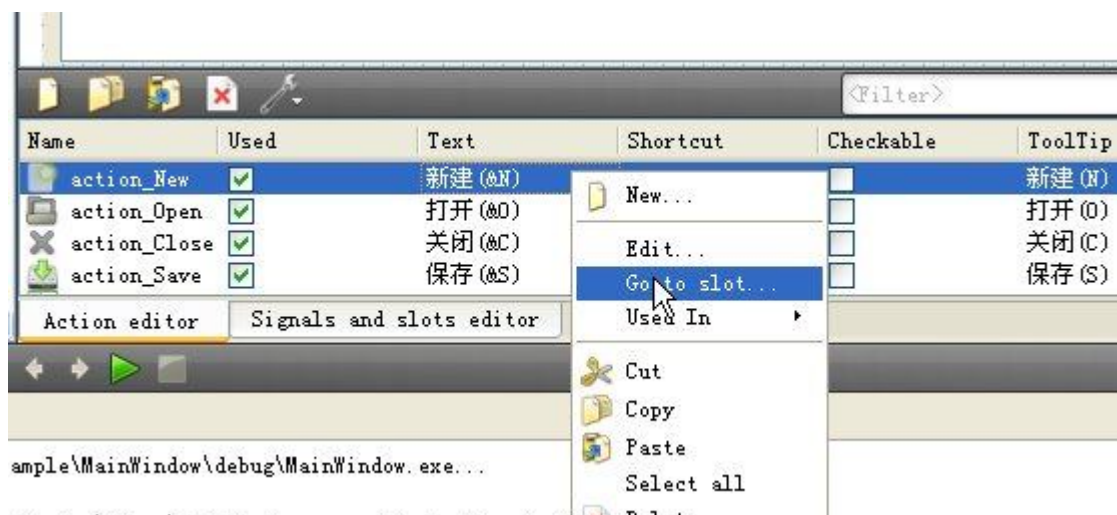
上面只是做了一个简单的说明。以后我们对自己不明白的类都可以去帮助里进行查找，这也许是我们以后要做的最多的一件事了。对于其中的英文解释，我们最好想办法弄明白它的大意，其实网上也有一些中文的翻译，但最好还是从一开始就尝试着看英文原版的帮助，这样以后才不会对中文翻译产生依赖。

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

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



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



同理，进入其他两个菜单的槽函数，将相应的操作的函数写入槽函数中。如下。
void MainWindow::on_action_New_triggered() //信号和槽的关联

```

{
do_file_New();
}
void MainWindow::on_action_Save_triggered()
{
do_file_Save();
}
void MainWindow::on_action_SaveAs_triggered()
{
do_file_SaveAs();
}

```

这时点击运行，就能够实现新建文件，保存文件，文件另存为的功能了。



然后实现打开，关闭，退出，撤销，复制，剪切，粘贴的功能。

先备份上次的工程文件，然后再将其打开。

1. 先在 `mainwindow.h` 文件中加入函数的声明。

```

void do_file_Open(); //打开文件
bool do_file_Load(const QString& fileName); //读取文件

```

2. 再在 `mainwindow.cpp` 文件中写函数的功能实现。

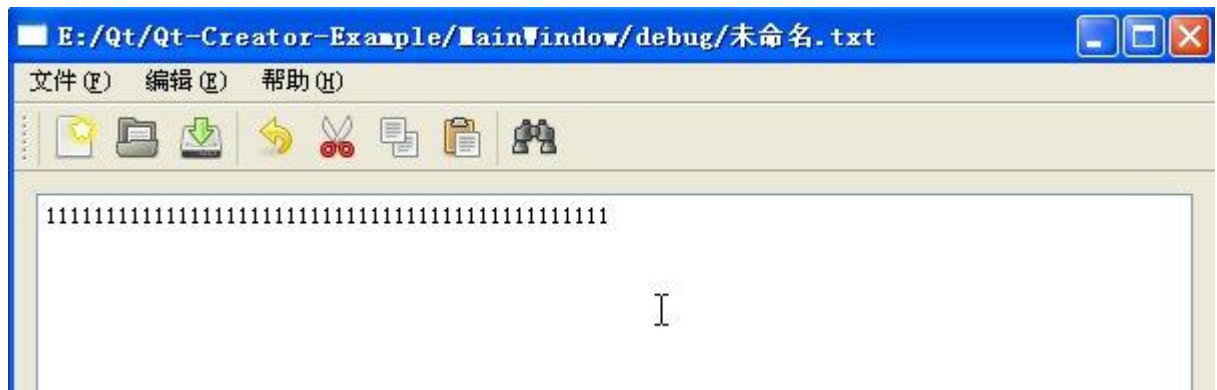
```

void MainWindow::do_file_Open() //打开文件
{
do_file_SaveOrNot(); //是否需要保存现有文件
QString fileName = QFileDialog::getOpenFileName(this);
//获得要打开的文件的名字
if(!fileName.isEmpty()) //如果文件名不为空
{
do_file_Load(fileName);
}
ui->textEdit->setVisible(true); //文本编辑器可见
}

```




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

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）

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

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

1. 首先还是保存以前的工程，然后再将其打开。

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



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

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

在 private slots 中添加

```
void show_findText();
```

在该函数中实现查找字符串的功能。

3. 我们进入查找菜单的触发事件槽函数，更改如下。

```
void MainWindow::on_action_Find_triggered()
{
    QDialog *findDlg = new QDialog(this);
```

```

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

```

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



5. 下面我们开始写实现查找功能的 show_findText() 函数。

```

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

```

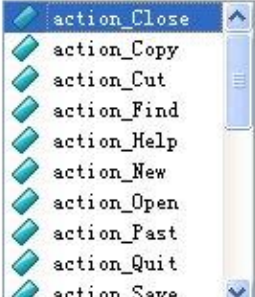
先用一个 QString 类的对象获得要查找的字符。然后我们一步一步写查找操作的语句。

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

```

190 void MainWindow::show_findText() //“查找下一个”按钮的槽函数
191 {
192     QString findText = find_textLineEdit->text();
193     //获取行编辑器中的内容
194     ui->[
195     }
196
197
198
199
200
201

```

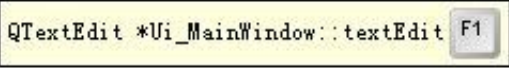


7. 我们用向下的方向键选中列表中的 textEdit。或者我们可以先输入 text，这时能缩减列表的内容。

```

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。



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

The FindFlags type is a typedef for QFlags<FindFlag>. It stores 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 QTextEdit 类的相关属性和方法。

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

find

findChild

findChildren

focusInEvent

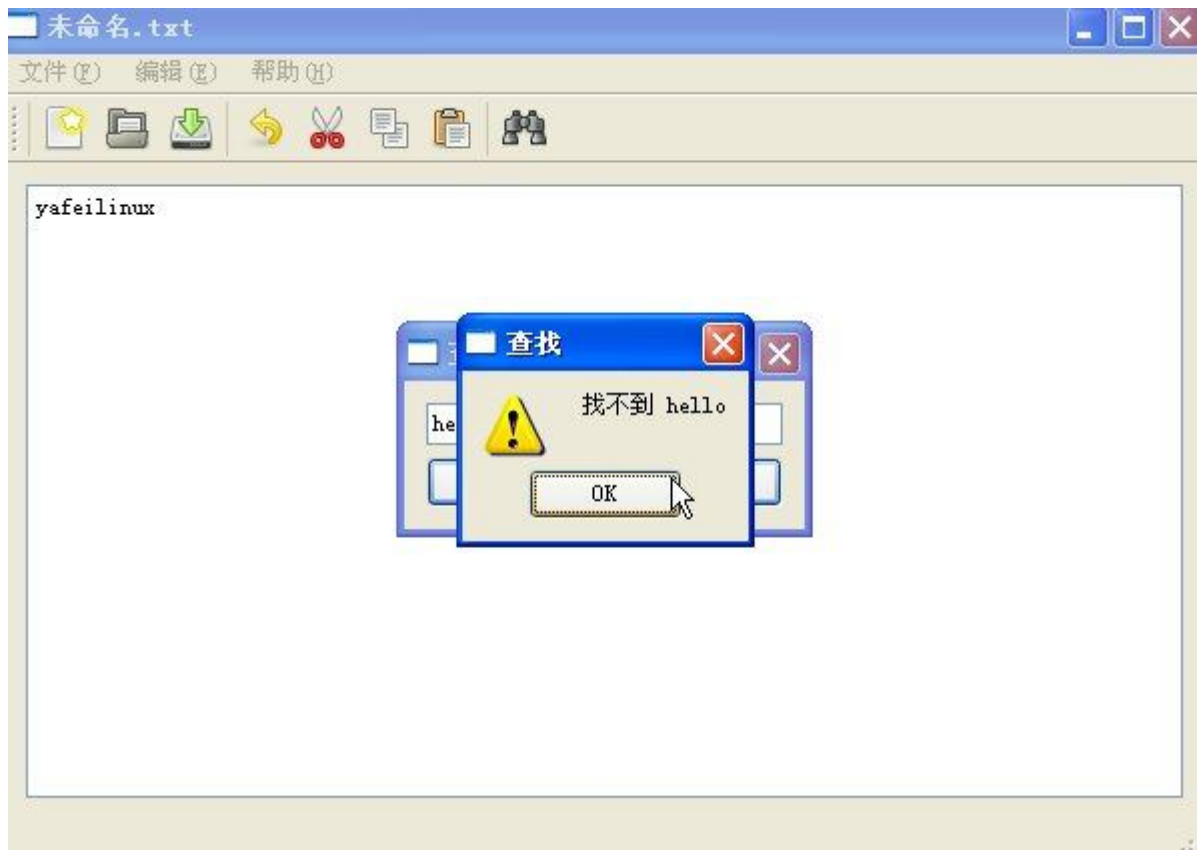
可以看到，当写完函数名和第一个“(”后，系统会自动显示出该函数的函数原型，这样可以使我们减少出错。

```
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
    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。

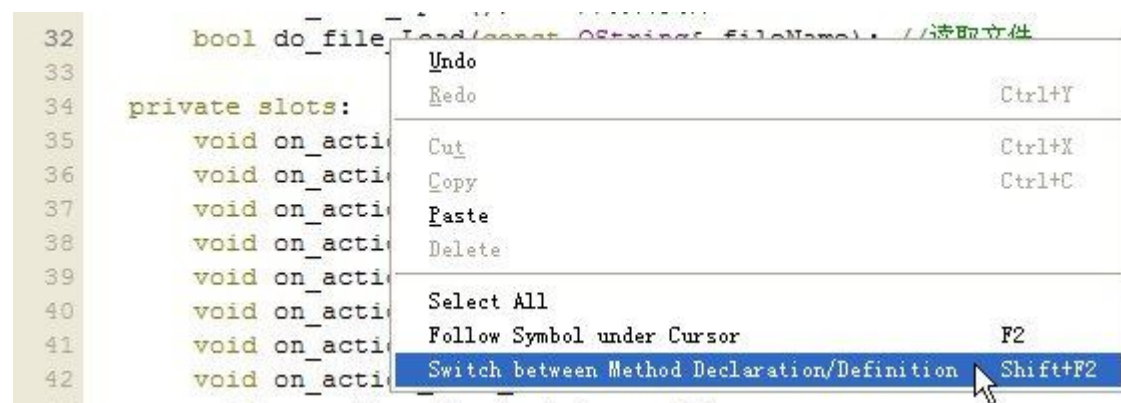
```
1 // 在 MainWindow 中 show_findText() // 查找下一个匹配的项
2
3 #include <QtGui/QTextDocument.h>
4 #include <QtGui/QTextEditor.h>
5 #include <QtGui/QTextFindDialog.h>
6 #include <QtGui/QMessageBox.h>
7 #include <QtGui/QTextCursor.h>
8 #include <QtGui/QTextBlock.h>
9 #include <QtGui/QTextLine.h>
10 #include <QtGui/QTextCharFormat.h>
11 #include <QtGui/QTextBlockFormat.h>
12 #include <QtGui/QTextBlockGroup.h>
13 #include <QtGui/QTextBlockGroupFormat.h>
14 #include <QtGui/QTextBlockGroupFormat.h>
15 #include <QtGui/QTextBlockGroupFormat.h>
16 #include <QtGui/QTextBlockGroupFormat.h>
17 #include <QtGui/QTextBlockGroupFormat.h>
18 #include <QtGui/QTextBlockGroupFormat.h>
19 #include <QtGui/QTextBlockGroupFormat.h>
20 #include <QtGui/QTextBlockGroupFormat.h>
21 #include <QtGui/QTextBlockGroupFormat.h>
22 #include <QtGui/QTextBlockGroupFormat.h>
23 #include <QtGui/QTextBlockGroupFormat.h>
24 #include <QtGui/QTextBlockGroupFormat.h>
25 #include <QtGui/QTextBlockGroupFormat.h>
26 #include <QtGui/QTextBlockGroupFormat.h>
27 #include <QtGui/QTextBlockGroupFormat.h>
28 #include <QtGui/QTextBlockGroupFormat.h>
29 #include <QtGui/QTextBlockGroupFormat.h>
30 #include <QtGui/QTextBlockGroupFormat.h>
31 #include <QtGui/QTextBlockGroupFormat.h>
32 #include <QtGui/QTextBlockGroupFormat.h>
33 #include <QtGui/QTextBlockGroupFormat.h>
34 #include <QtGui/QTextBlockGroupFormat.h>
35 #include <QtGui/QTextBlockGroupFormat.h>
36 #include <QtGui/QTextBlockGroupFormat.h>
37 #include <QtGui/QTextBlockGroupFormat.h>
38 #include <QtGui/QTextBlockGroupFormat.h>
39 #include <QtGui/QTextBlockGroupFormat.h>
40 #include <QtGui/QTextBlockGroupFormat.h>
41 #include <QtGui/QTextBlockGroupFormat.h>
42 #include <QtGui/QTextBlockGroupFormat.h>
43 #include <QtGui/QTextBlockGroupFormat.h>
44 #include <QtGui/QTextBlockGroupFormat.h>
45 #include <QtGui/QTextBlockGroupFormat.h>
46 #include <QtGui/QTextBlockGroupFormat.h>
47 #include <QtGui/QTextBlockGroupFormat.h>
48 #include <QtGui/QTextBlockGroupFormat.h>
49 #include <QtGui/QTextBlockGroupFormat.h>
50 #include <QtGui/QTextBlockGroupFormat.h>
51 #include <QtGui/QTextBlockGroupFormat.h>
52 #include <QtGui/QTextBlockGroupFormat.h>
53 #include <QtGui/QTextBlockGroupFormat.h>
54 #include <QtGui/QTextBlockGroupFormat.h>
55 #include <QtGui/QTextBlockGroupFormat.h>
56 #include <QtGui/QTextBlockGroupFormat.h>
57 #include <QtGui/QTextBlockGroupFormat.h>
58 #include <QtGui/QTextBlockGroupFormat.h>
59 #include <QtGui/QTextBlockGroupFormat.h>
60 #include <QtGui/QTextBlockGroupFormat.h>
61 #include <QtGui/QTextBlockGroupFormat.h>
62 #include <QtGui/QTextBlockGroupFormat.h>
63 #include <QtGui/QTextBlockGroupFormat.h>
64 #include <QtGui/QTextBlockGroupFormat.h>
65 #include <QtGui/QTextBlockGroupFormat.h>
66 #include <QtGui/QTextBlockGroupFormat.h>
67 #include <QtGui/QTextBlockGroupFormat.h>
68 #include <QtGui/QTextBlockGroupFormat.h>
69 #include <QtGui/QTextBlockGroupFormat.h>
70 #include <QtGui/QTextBlockGroupFormat.h>
71 #include <QtGui/QTextBlockGroupFormat.h>
72 #include <QtGui/QTextBlockGroupFormat.h>
73 #include <QtGui/QTextBlockGroupFormat.h>
74 #include <QtGui/QTextBlockGroupFormat.h>
75 #include <QtGui/QTextBlockGroupFormat.h>
76 #include <QtGui/QTextBlockGroupFormat.h>
77 #include <QtGui/QTextBlockGroupFormat.h>
78 #include <QtGui/QTextBlockGroupFormat.h>
79 #include <QtGui/QTextBlockGroupFormat.h>
80 #include <QtGui/QTextBlockGroupFormat.h>
81 #include <QtGui/QTextBlockGroupFormat.h>
82 #include <QtGui/QTextBlockGroupFormat.h>
83 #include <QtGui/QTextBlockGroupFormat.h>
84 #include <QtGui/QTextBlockGroupFormat.h>
85 #include <QtGui/QTextBlockGroupFormat.h>
86 #include <QtGui/QTextBlockGroupFormat.h>
87 #include <QtGui/QTextBlockGroupFormat.h>
88 #include <QtGui/QTextBlockGroupFormat.h>
89 #include <QtGui/QTextBlockGroupFormat.h>
90 #include <QtGui/QTextBlockGroupFormat.h>
91 #include <QtGui/QTextBlockGroupFormat.h>
92 #include <QtGui/QTextBlockGroupFormat.h>
93 #include <QtGui/QTextBlockGroupFormat.h>
94 #include <QtGui/QTextBlockGroupFormat.h>
95 #include <QtGui/QTextBlockGroupFormat.h>
96 #include <QtGui/QTextBlockGroupFormat.h>
97 #include <QtGui/QTextBlockGroupFormat.h>
98 #include <QtGui/QTextBlockGroupFormat.h>
99 #include <QtGui/QTextBlockGroupFormat.h>
100 #include <QtGui/QTextBlockGroupFormat.h>
```



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

第一，在函数声明的地方直接跳转到函数定义的地方。

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

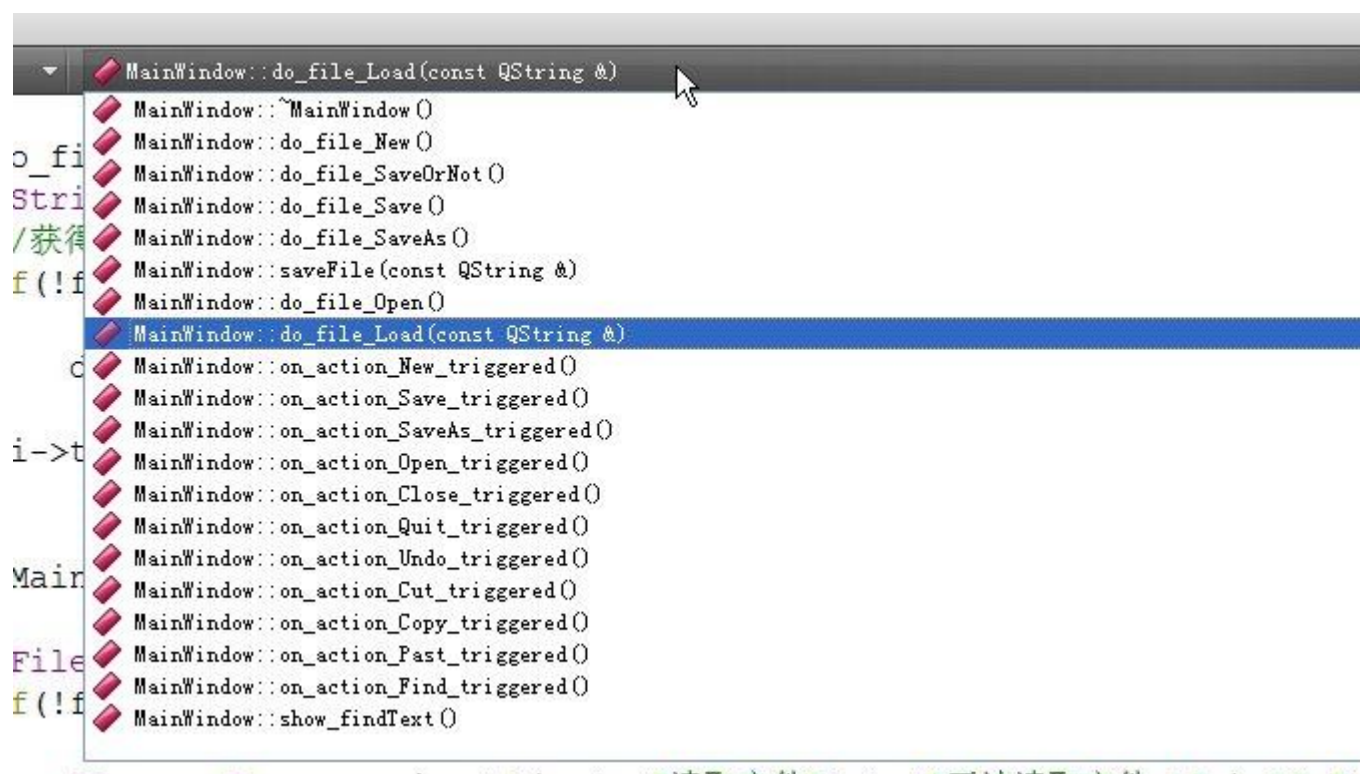


这时系统就会自动跳转到函数定义的位置。如下图。

```
101 bool MainWindow::do_file_Load(const QString& fileName) //读取文件
102 {
103     QFile file(fileName);
104     if(!file.open(QFile::ReadOnly | QFile::Text))
105     {
106         QMessageBox::warning(this, tr("读取文件"), tr("无法读取文件"),
107                               .arg(fileName).arg(file.errorStr()));
108     }
109     return false;
110 }
```

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。

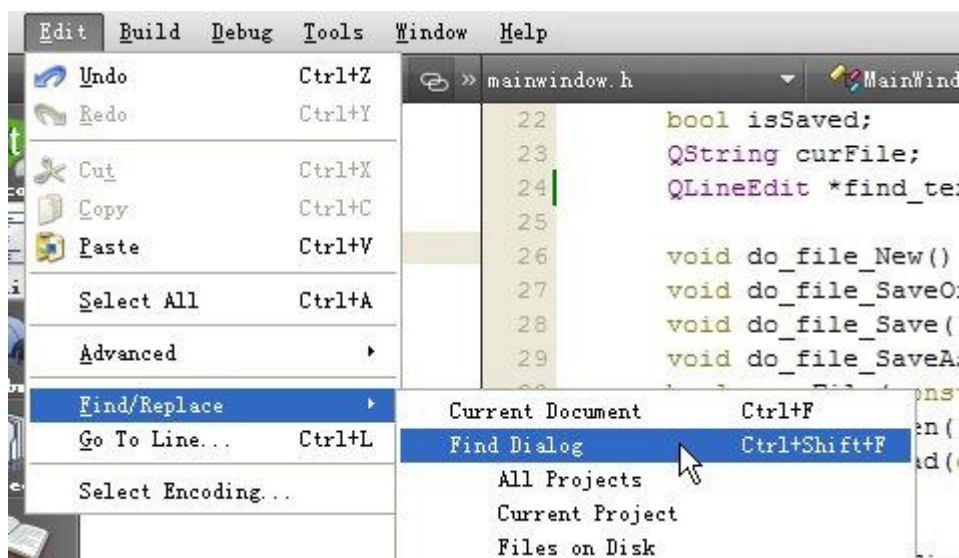


第三，利用查找功能。

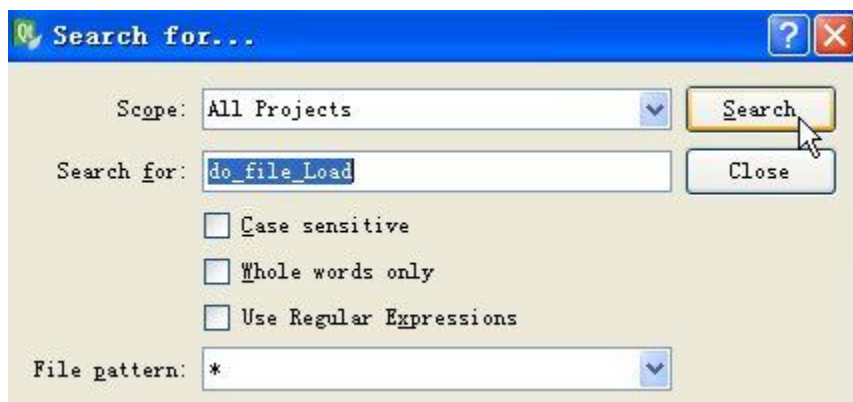
1. 我们先将鼠标定位到一个函数名上。

```
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();
```

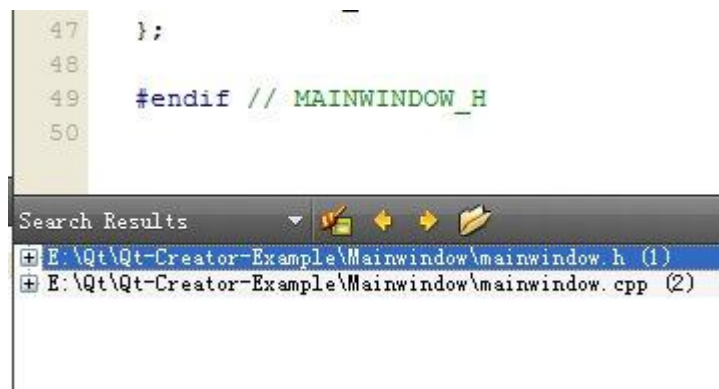
2. 然后选择 Edit->Find/Replace->Find Dialog。



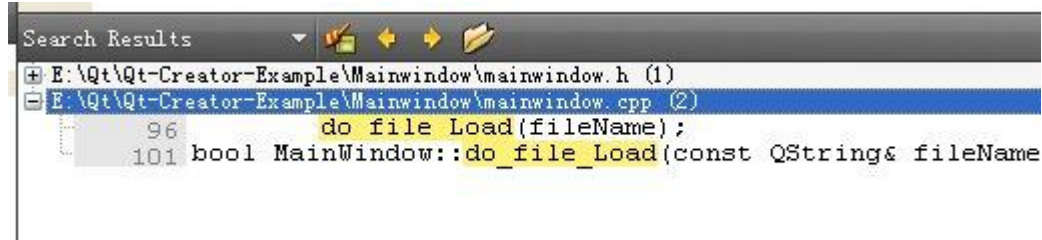
3. 这时会出现一个查找对话框，可以看到要查找的函数名已经写在里面了。



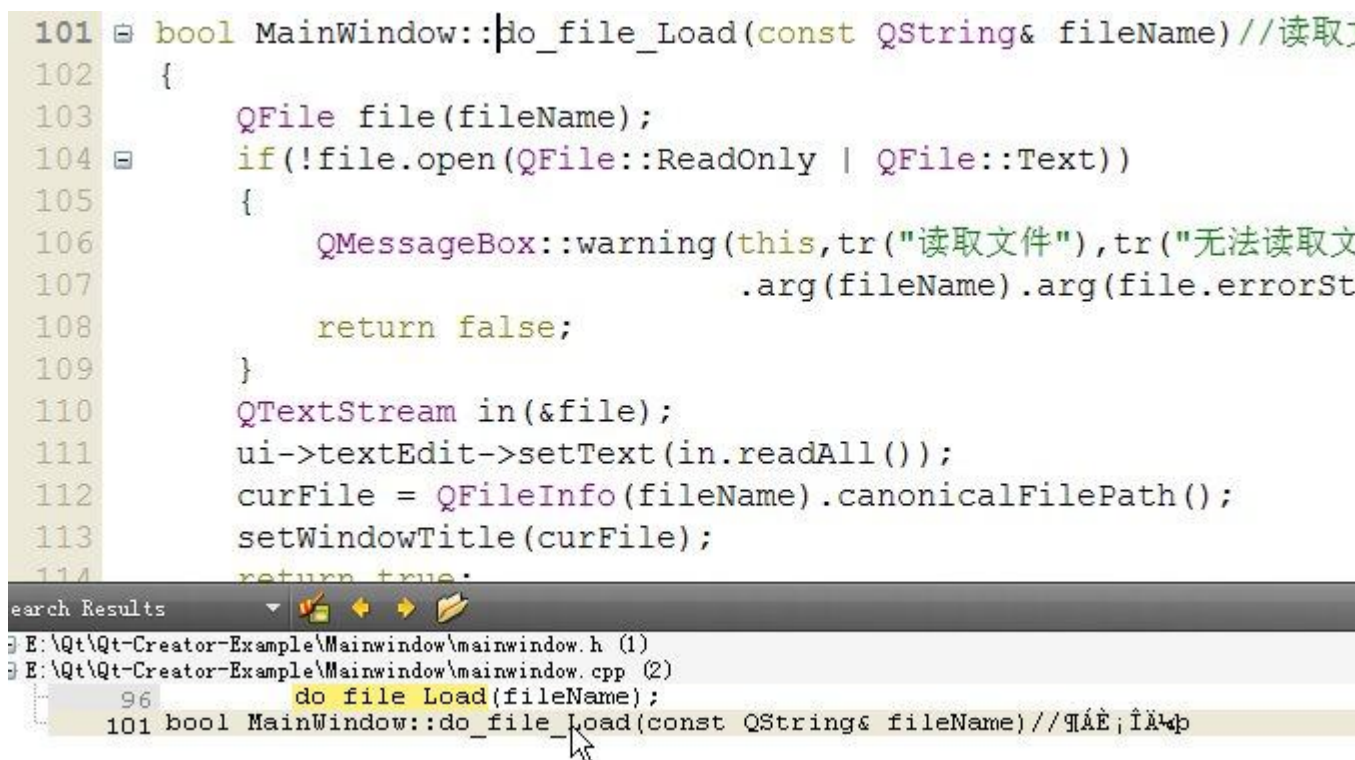
4. 当我们按下 Search 按钮后，会在查找结果窗口显示查找到的结果。



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）

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()
```

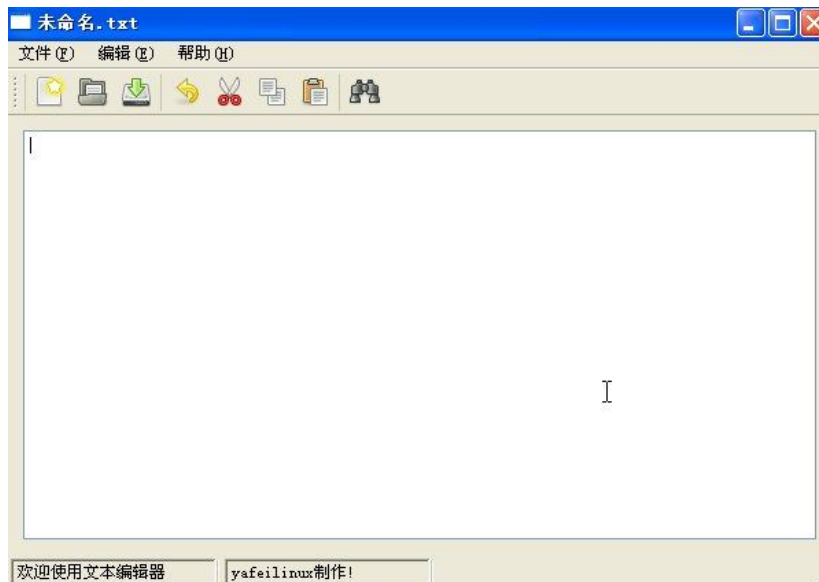
```
{
    QStatusBar* bar = ui->statusBar; //获取状态栏
    first_statusLabel = new QLabel; //新建标签
    first_statusLabel->setMinimumSize(150, 20); //设置标签最小尺寸
    first_statusLabel->setFrameShape(QFrame::WinPanel); //设置标签形状
    first_statusLabel->setFrameShadow(QFrame::Sunken); //设置标签阴影
    second_statusLabel = 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。

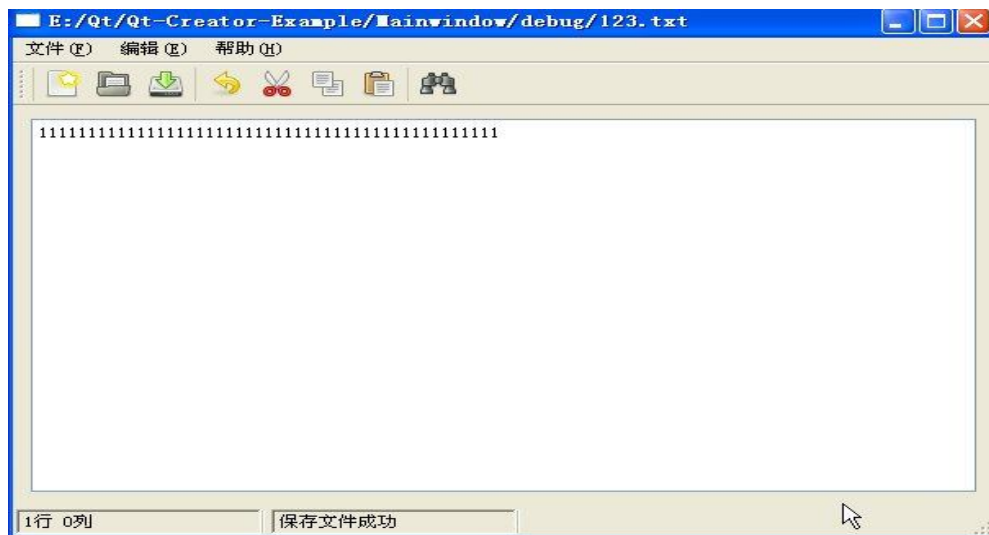
```
void MainWindow::do_cursorChanged()
{
    int rowNum = ui->textEdit->document()->blockCount();
    //获取光标所在行的行号
    const QTextCursor cursor = ui->textEdit->textCursor();
    int colNum = cursor.columnNumber();
    //获取光标所在列的列号
    first_statusLabel->setText(tr("%1 行 %2 列").arg(rowNum).arg(colNum));
    //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。



8. 在 `on_action_Find_triggered` 函数的后面添加如下语句。

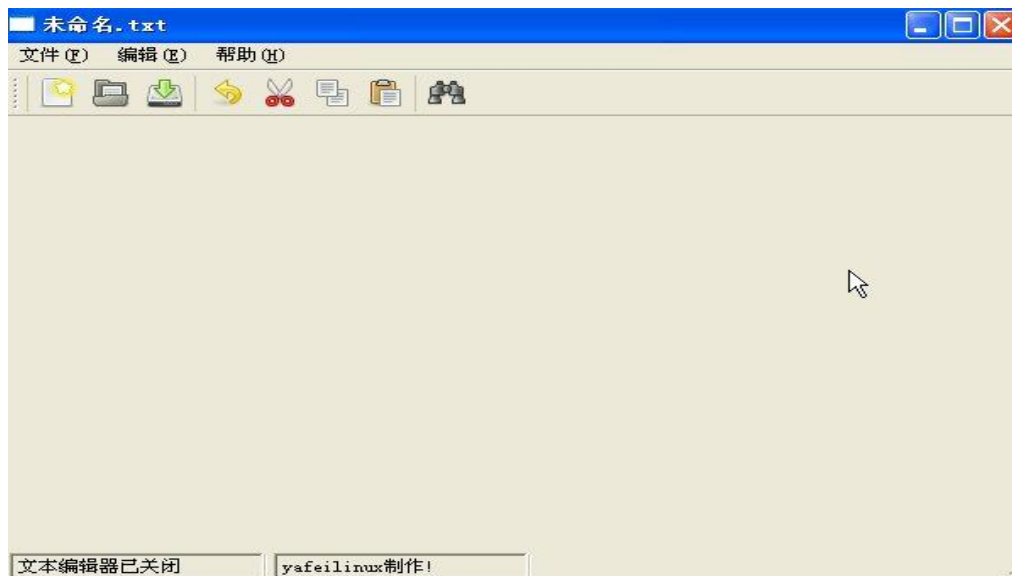
```
second_statusLabel->setText(tr("正在进行查找"));
```



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!"));
```



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）

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。

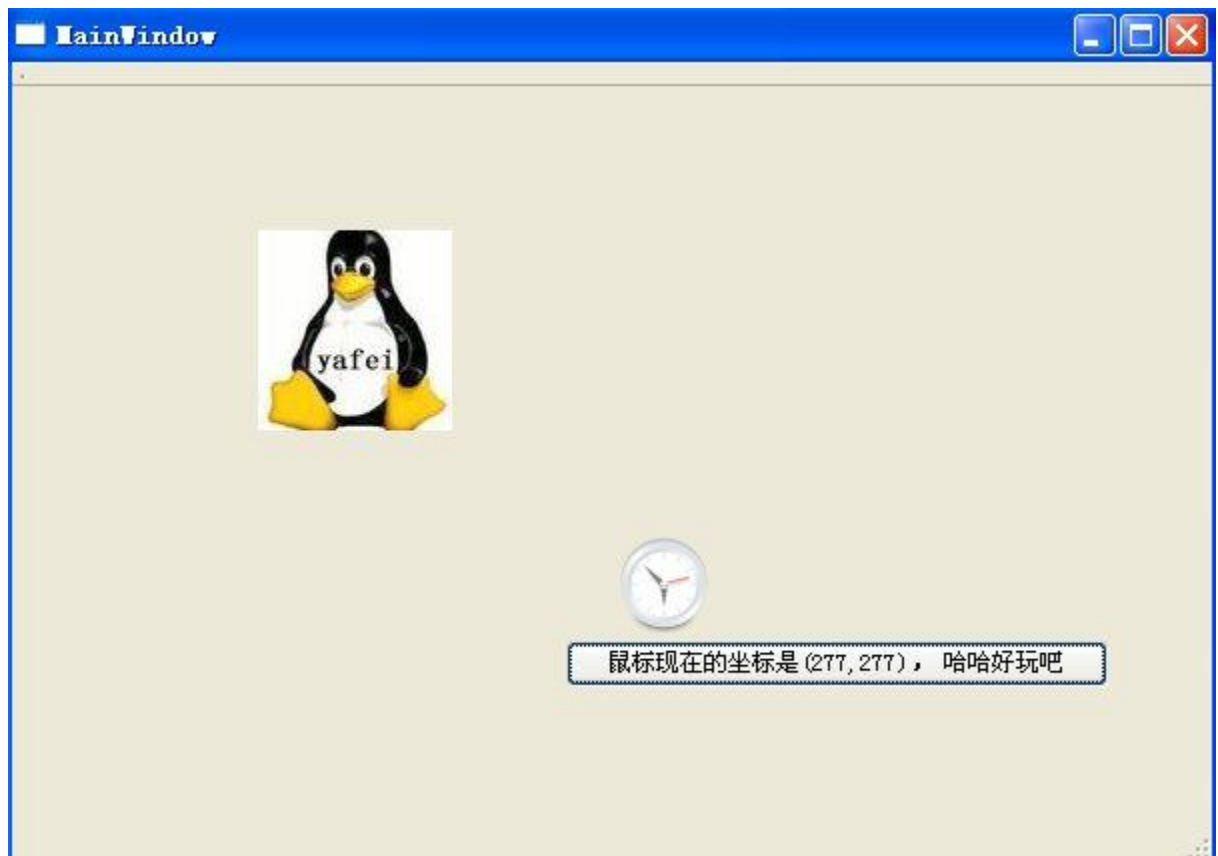
效果如下。



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）

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);

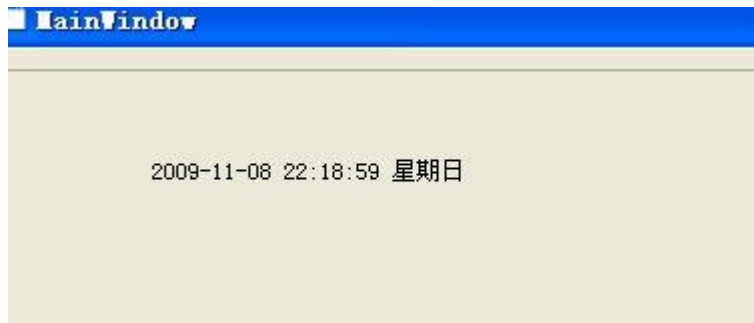
//新建定时器
connect(timer, SIGNAL(timeout()), this, SLOT(timerUpdate()));
//关联定时器计满信号和相应的槽函数
timer->start(1000);
//定时器开始计时，其中 1000 表示 1000ms 即 1 秒
```

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。

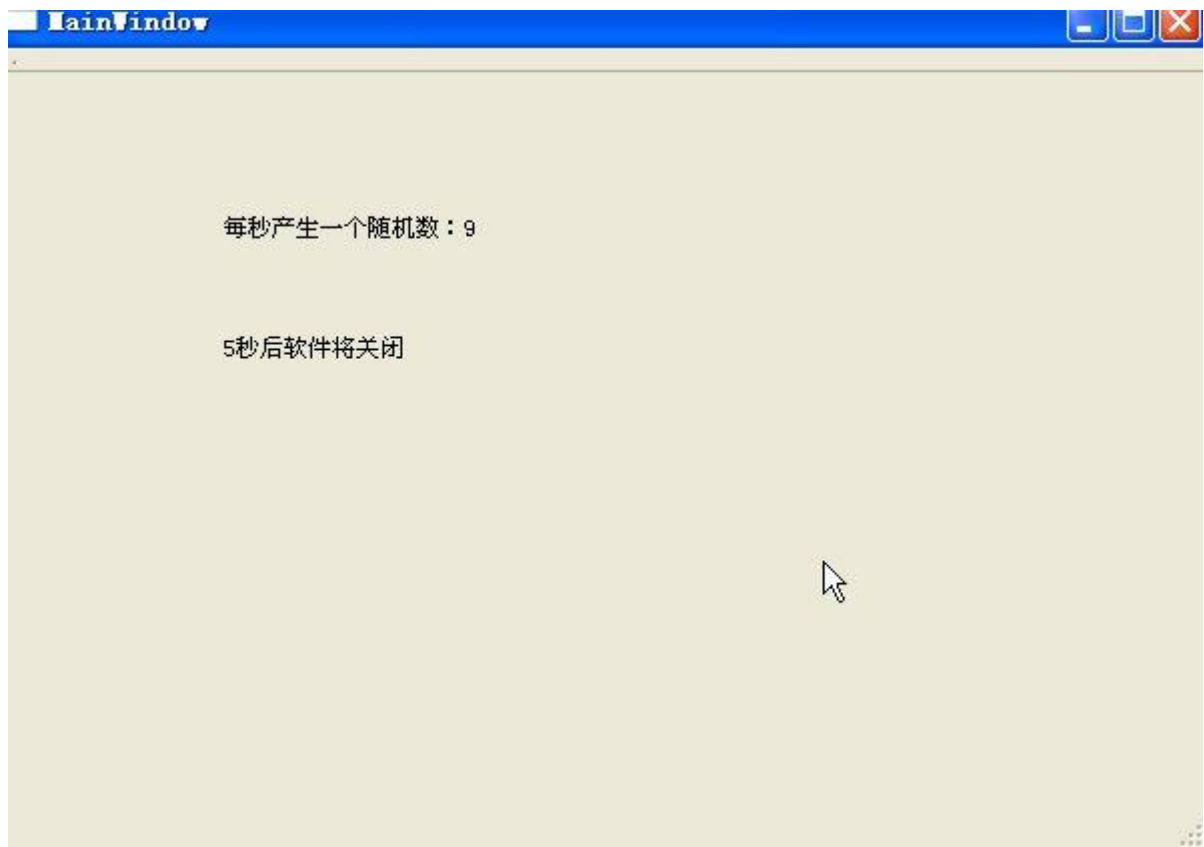
随机数的实现：

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）

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。

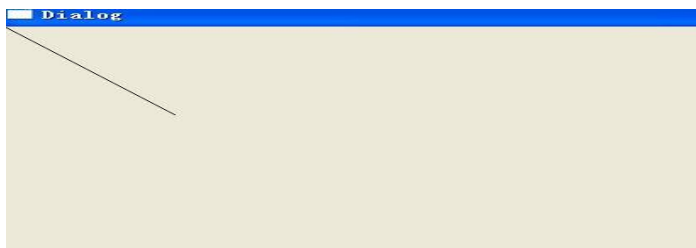
说明：以后使用的环境为基于 Qt 4.6 的 Qt Creator 1.3.0 windows 版本

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：



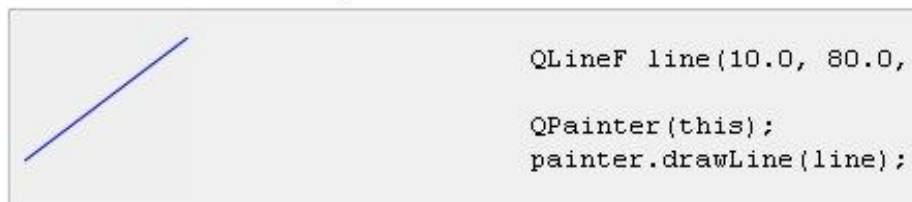
5. 在 qt 的帮助里可以查看所有的绘制函数，而且下面还给出了相关的例子。


```

void drawArc ( const QRectF & rectangle, int
void drawArc ( const QRect & rectangle, int
void drawArc ( int x, int y, int width, int height
void drawChord ( const QRectF & rectangle
void drawChord ( const QRect & rectangle,
void drawChord ( int x, int y, int width, int height
void drawConvexPolygon ( const QPointF
void drawConvexPolygon ( const QPoint *
void drawConvexPolygon ( const QPolygon
void drawConvexPolygon ( const QPolygon
void drawEllipse ( const QRectF & rectangle

```

Draws a line defined by *line*.



6. 我们下面将几个知识点说明一下，帮助大家更快入门。

将函数改为如下：

```

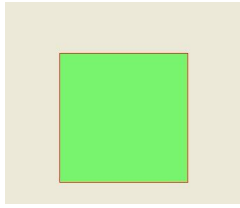
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);

    QPen pen; //画笔
    pen.setColor(QColor(255, 0, 0));
    QBrush brush(QColor(0, 255, 0, 125)); //画刷

    painter.setPen(pen); //添加画笔
    painter.setBrush(brush); //添加画刷
    painter.drawRect(100, 100, 200, 200); //绘制矩形
}

```

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



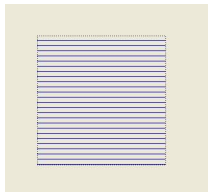
7. 其实画笔和画刷也有很多设置，大家可以查看帮助。

```
QPainter painter(this);

QPen pen(Qt::DotLine);
QBrush brush(Qt::blue);
brush.setStyle(Qt::HorPattern);

painter.setPen(pen);
painter.setBrush(brush);
painter.drawRect(100, 100, 200, 200);
```

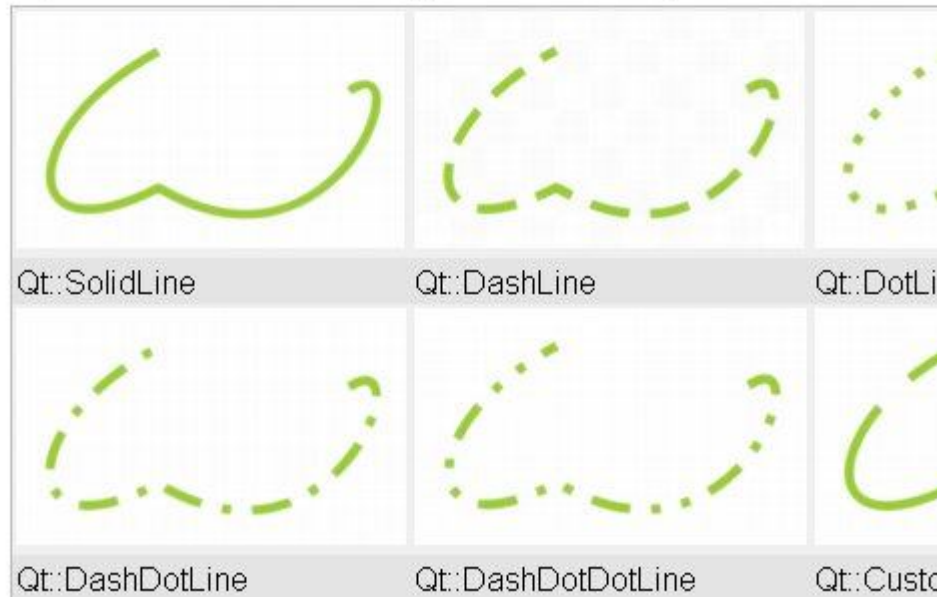
这里我们设置了画笔的风格为点线，画刷的风格为并行横线，效果如下：



在帮助里可以看到所有的风格。

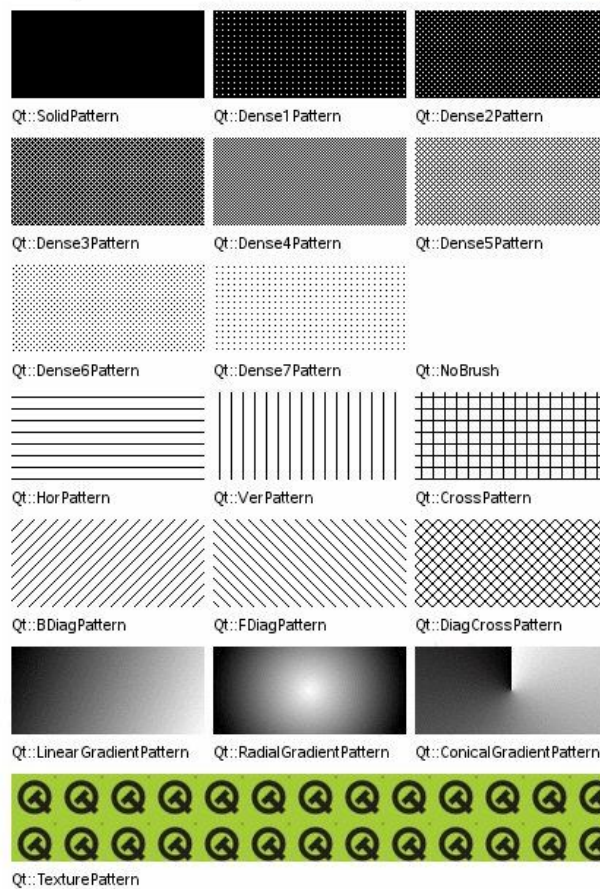
enum Qt::PenStyle

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



enum Qt::BrushStyle

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



我们这里用了 Qt::blue，Qt 自定义的几个颜色如下：

enum Qt::GlobalColor

Qt's predefined QColor objects:

Constant	Value	Description
Qt::white	3	White (#ffffff)
Qt::black	2	Black (#000000)
Qt::red	7	Red (#ff0000)
Qt::darkRed	13	Dark red (#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 (#a0a0a4)
Qt::darkGray	4	Dark gray (#808080)
Qt::lightGray	6	Light gray (#c0c0c0)
Qt::transparent	19	a transparent black value (i.e., QColor(0, 0, 0, 0))
Qt::color0	0	0 pixel value (for bitmaps)
Qt::color1	1	1 pixel value (for bitmaps)

See also QColor.

8. 画弧线，这是帮助里的一个例子。

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

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

这里要说明的是，画弧线时，角度被分成了十六分之一，就是说，要想为 30 度，就得是 30*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）

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。

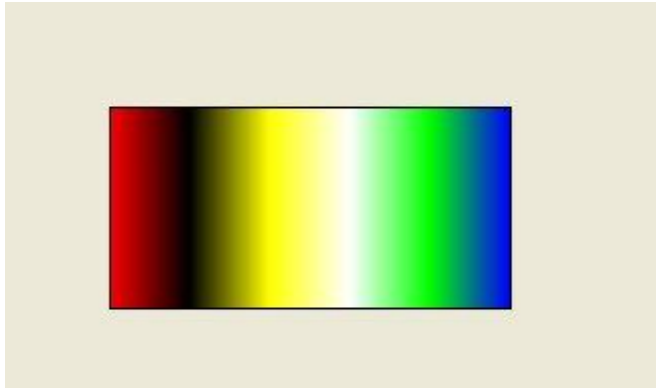
在 qt 中提供了三种渐变方式，分别是线性渐变，圆形渐变和圆锥渐变。如果能熟练应用它们，就能设计出炫目的填充效果。

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：

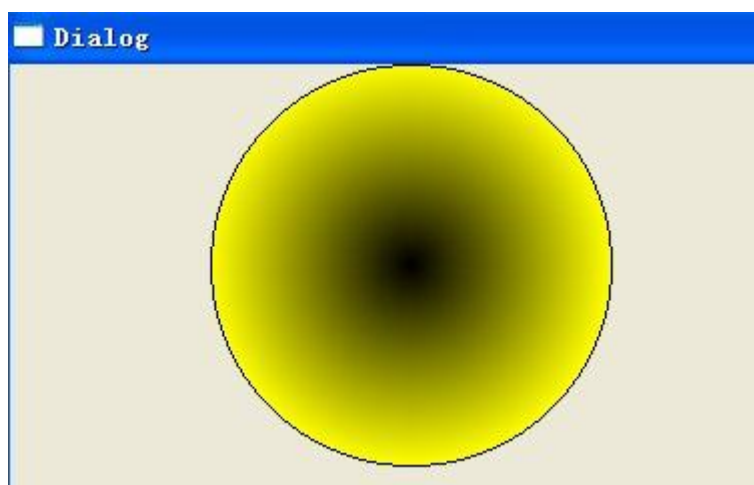


圆形渐变:

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
```

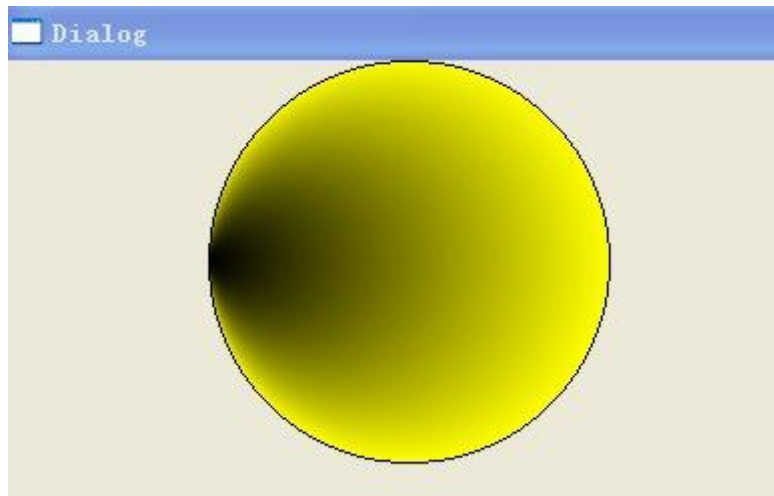
效果如下:



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：

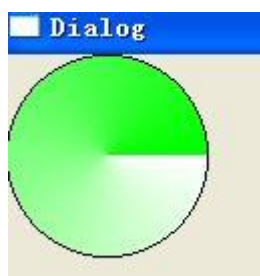


锥形渐变：

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);
```

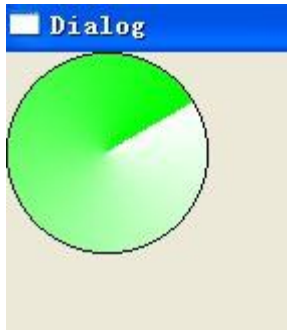
效果如下：



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）

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 QString
```

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\nNok
```

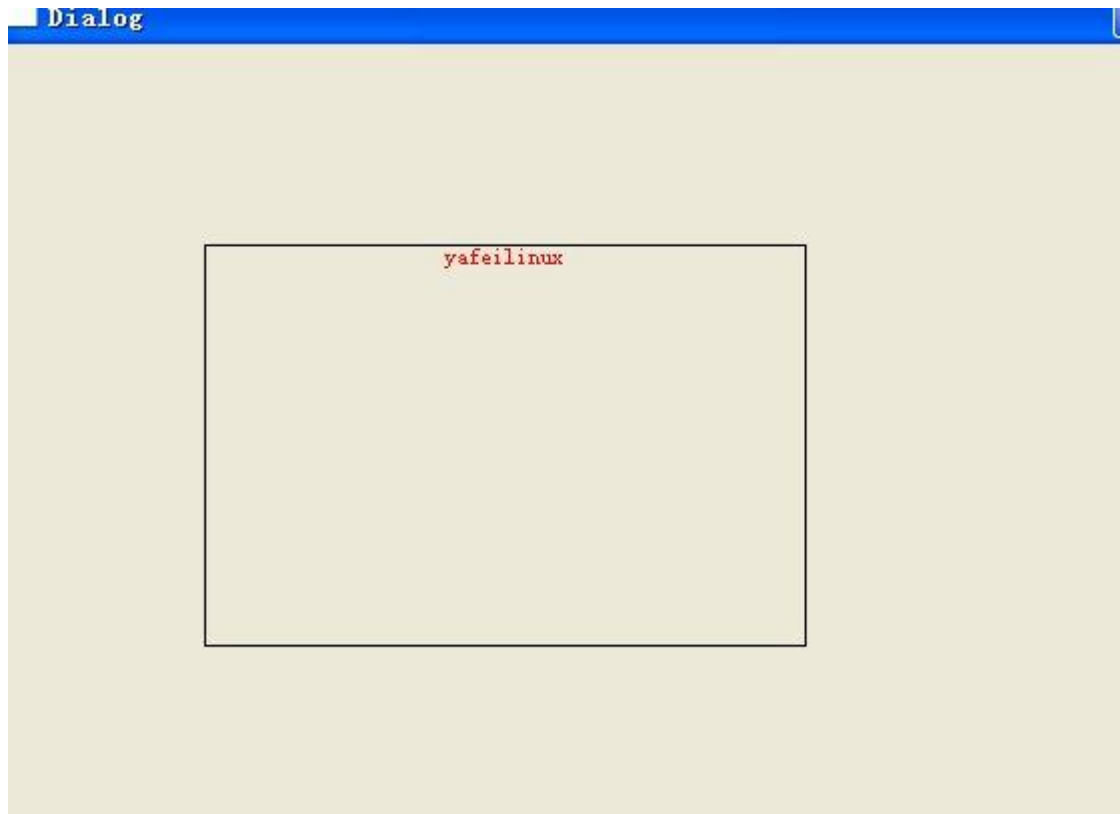
The *boundingRect* (if not null) is set to the what the bounding rectangle should be in OR of the following flags:

- 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。

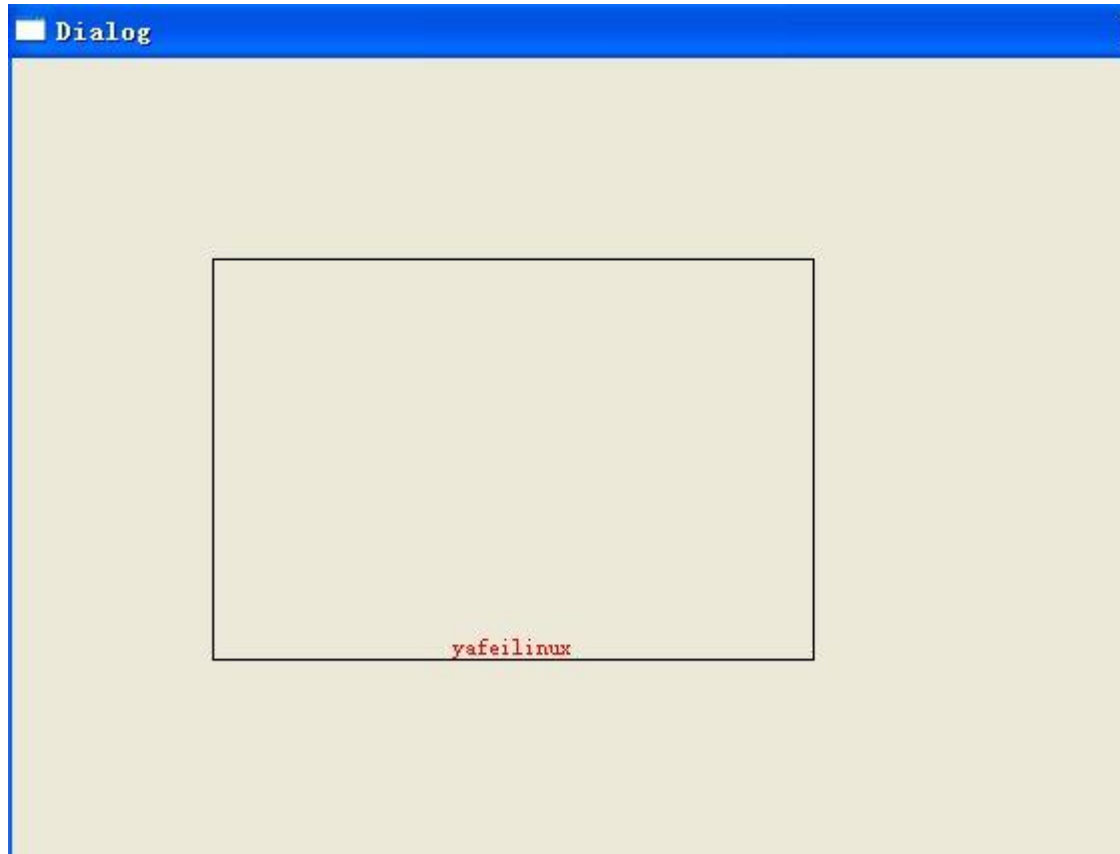
```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    QRectF ff(100, 100, 300, 200);  
    //设置一个矩形  
    painter.drawRect(ff);  
    //为了更直观地看到字体的位置，我们绘制出这个矩形  
    painter.setPen(QColor(Qt::red));  
    //设置画笔颜色为红色  
    painter.drawText(ff, Qt::AlignHCenter, "yafeilinux");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

效果如下。

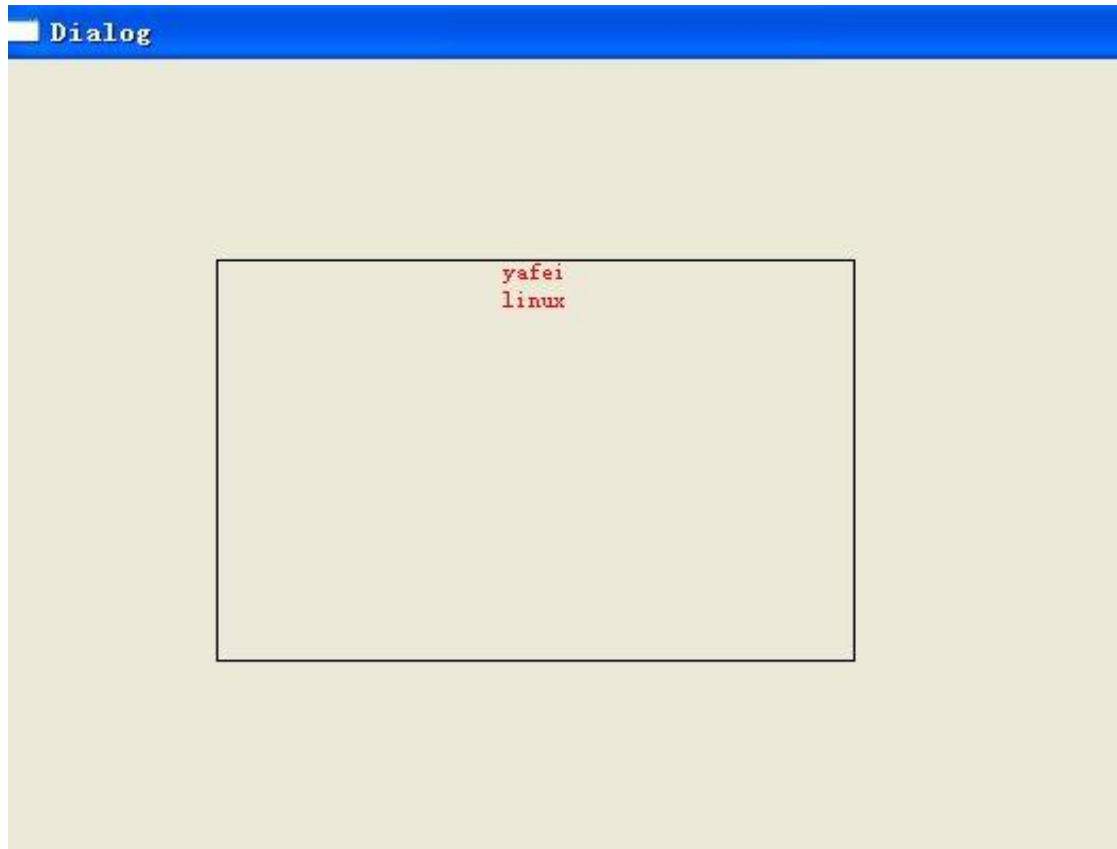


可以看到字符串是在最上面水平居中的。如果想让其在矩形正中间，我们可以使用 `Qt::AlignCenter`。

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。



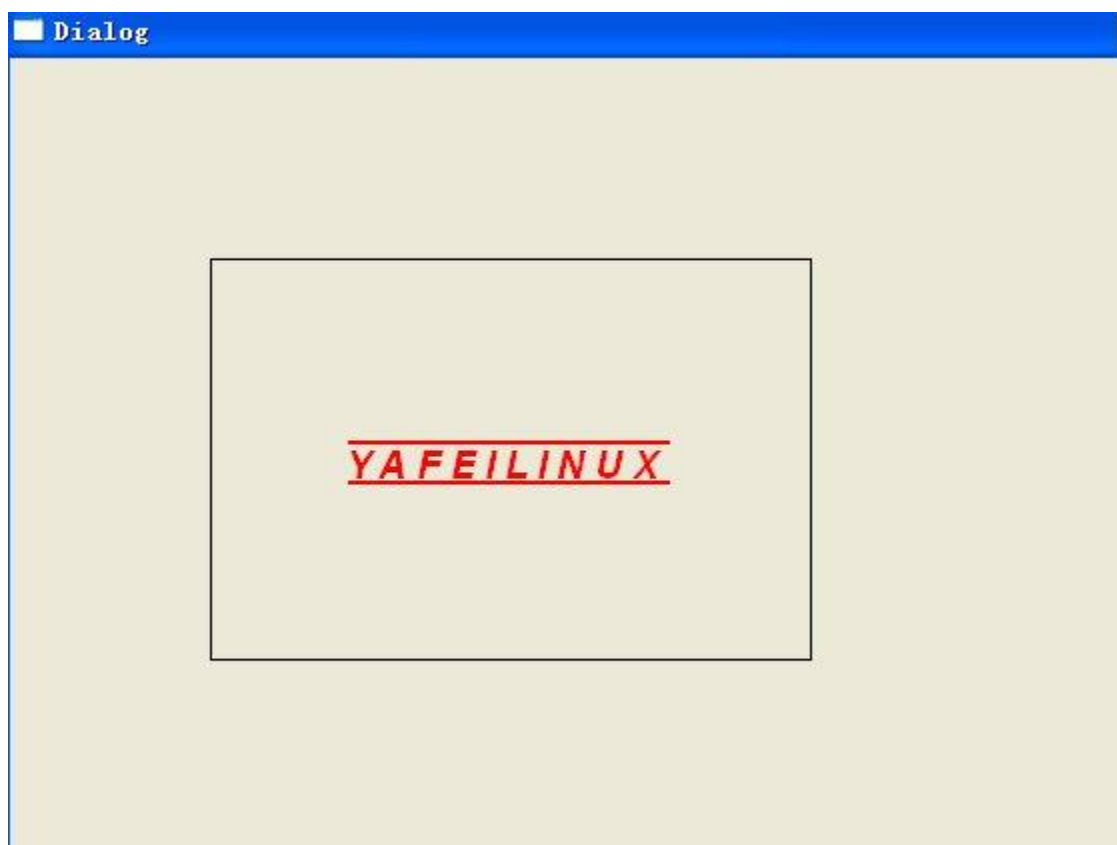
可以看到它有好几个枚举变量来设置字体。下面的例子我们对主要的几个选项进行演示。

更改函数如下。

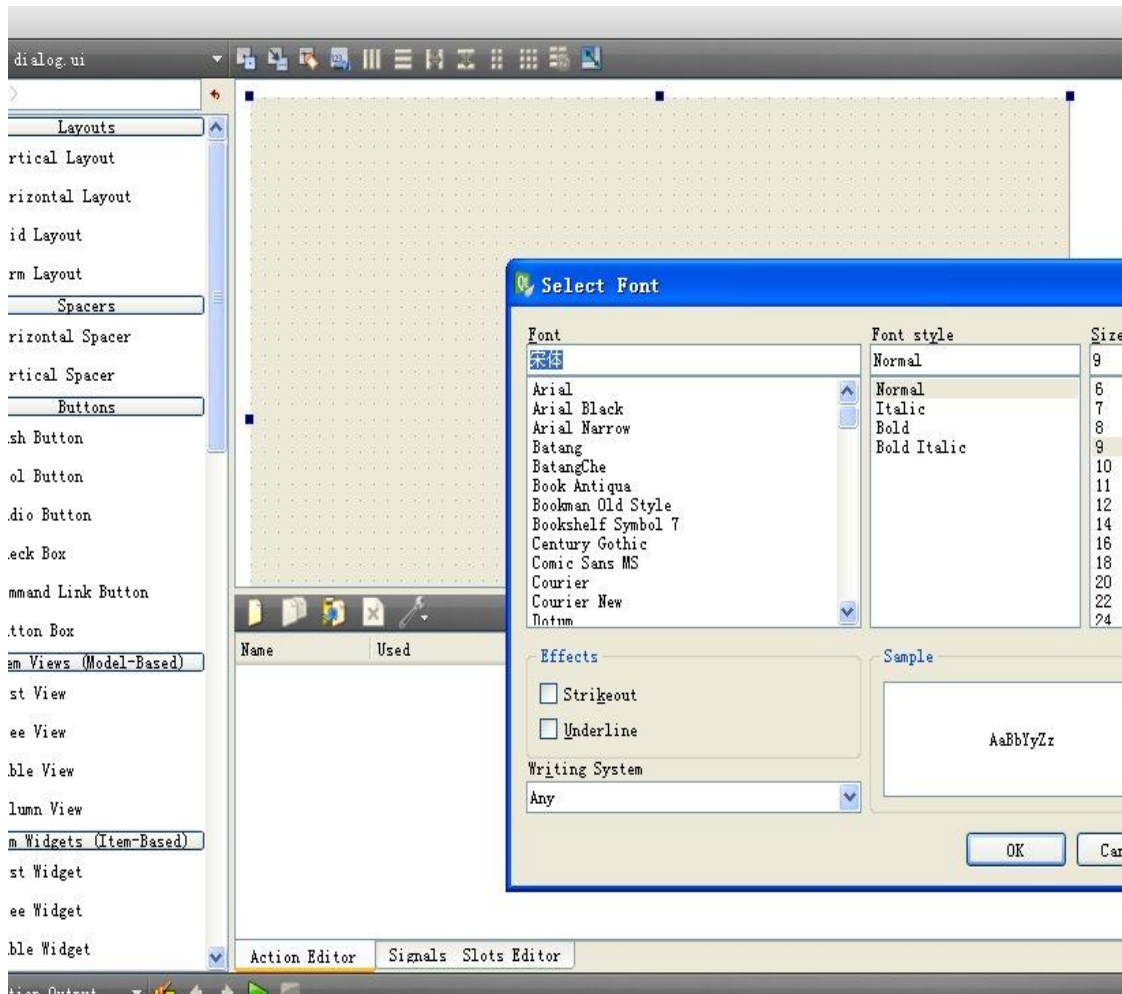
```
void Dialog::paintEvent(QPaintEvent *)
{
    QFont font("Arial", 20, QFont::Bold, true);
    //设置字体的类型，大小，加粗，斜体
```

```
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。

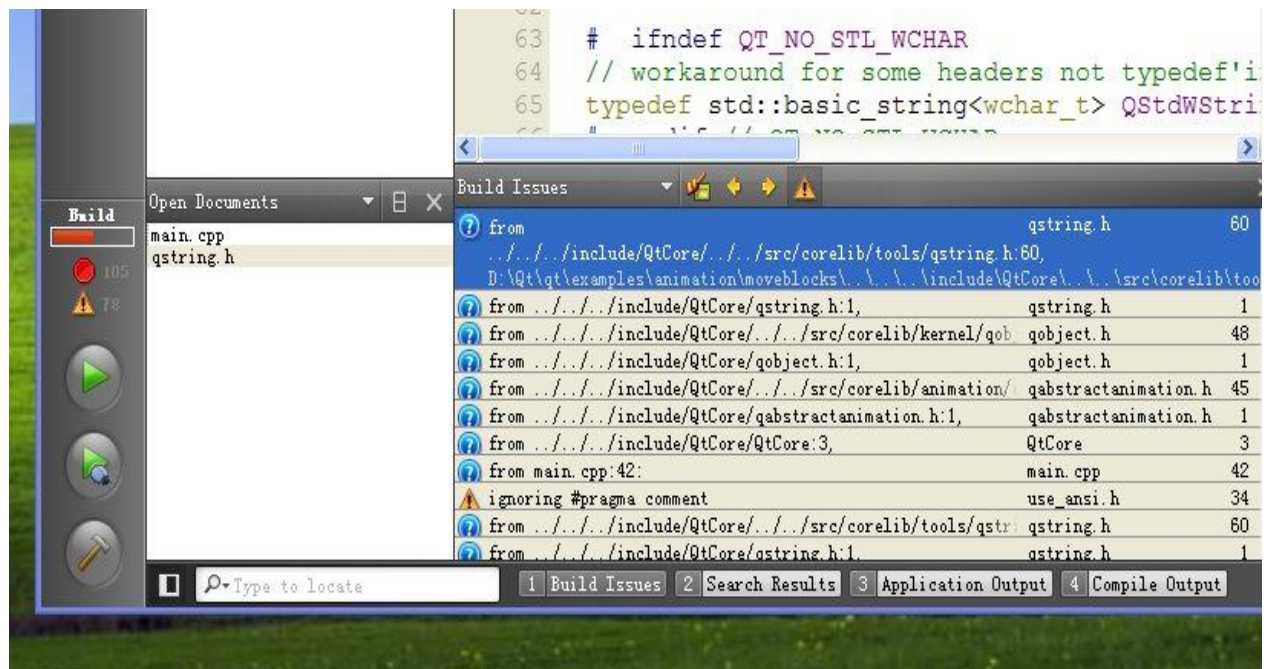


这里的所有字体我们可以在设计器中进行查看。如下。



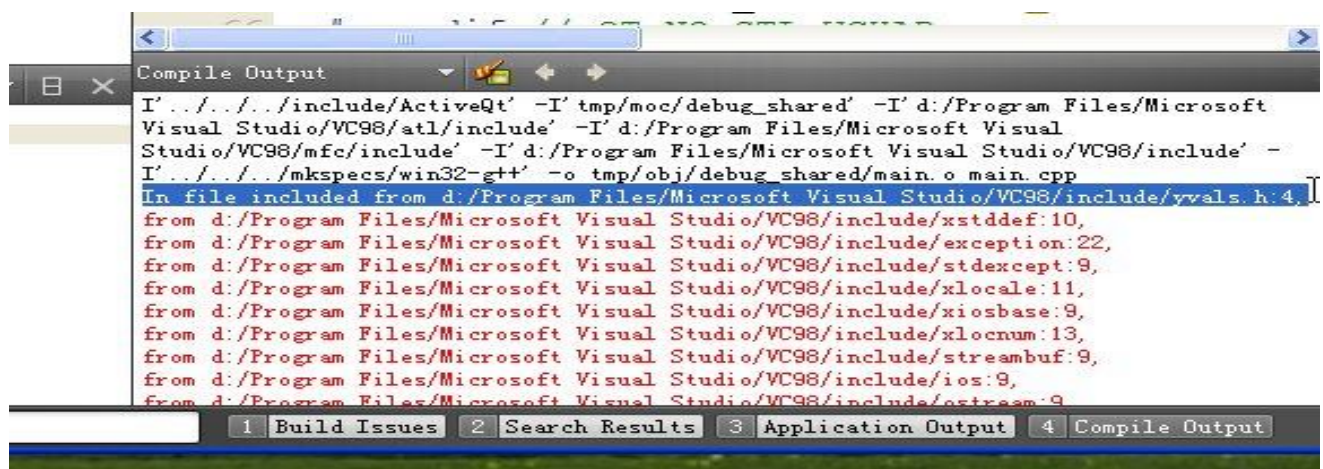
基于 Qt 4.6 的 Qt Creator 1.3.0 环境变量设置（原创）

如果你以前安装过 visual studio 2005 之类的软件，那么装上 Qt Creator 1.3.0 后，编译运行其自带的演示程序时就可能出现如下图的，105 个错误，几十个警告的问题。



我们查看输出窗口，如下图。会发现它居然显示 VC98 之类的东西，就是说它并没有去自己的 include 文件夹

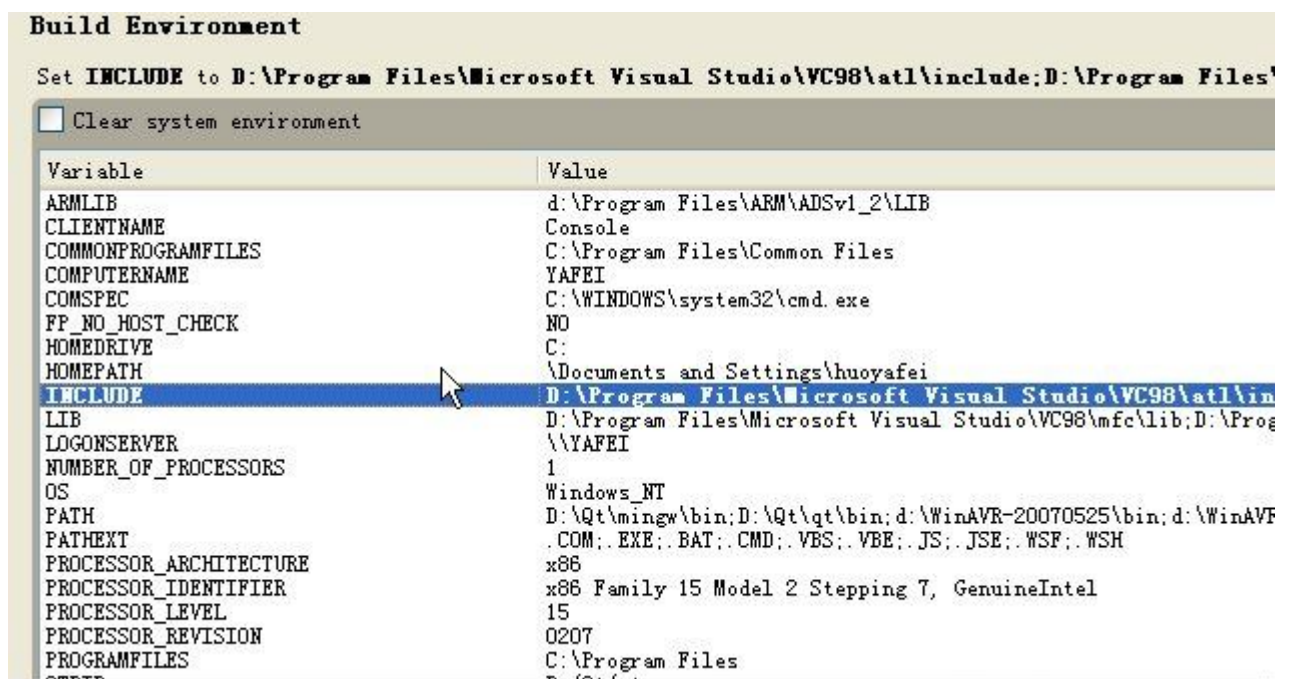
中查找文件。我们可以怀疑是系统环境变量的问题了。



点击 Qt Creator 界面左侧的 projects 图标，查看工程信息。这里我们主要查看编辑环境 Build Environment，点击其右侧的 show Details。



可以看到其中的 include 和 lib 均指向了 virtual studio 文件夹中，我们需要将其改正。



将他们都改为自己 Qt Creator 安装目录下的相关路径，如下图。（要换成你的安装路径）

COMPUTERNAME	YAFEI
COMSPEC	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK	NO
HOMEDRIVE	C:
HOMEPATH	\Documents and Settings\huoyafei
INCLUDE	D:\Qt\mingw\include
LIB	D:\Qt\mingw\lib
LOGONSERVER	\\YAFEI
NUMBER_OF_PROCESSORS	1
OS	Windows_NT
PATH	D:\Qt\mingw\bin;D:\Qt\bin;D:\Qt\...

改完后会发现新的设置已经显示出来了。

Clean Steps

Make: mingw32-make.exe clean -w in D:\Qt\qt\examples\animation

Build Environment

Set **INCLUDE** to D:\Qt\mingw\include

Set **LIB** to D:\Qt\mingw\lib

我们查看下面的 Run Environment，发现它已经自己改过来了。

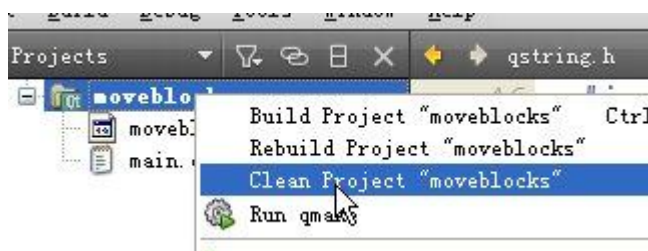
Run Environment

Summary: No changes to Environment

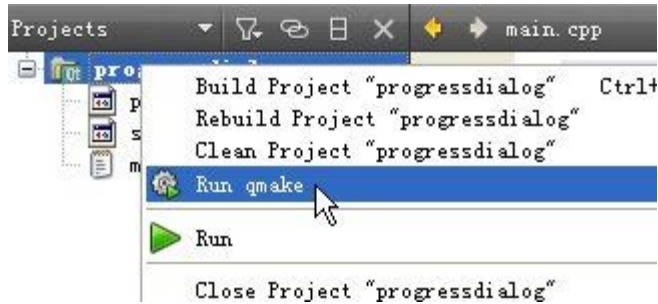
Base environment for this runconfiguration: Build Environment

Variable	Value
ARMLIB	d:\Program Files\ARM\ADSv1_2\LIB
CLIENTNAME	Console
COMMONPROGRAMFILES	C:\Program Files\Common Files
COMPUTERNAME	YAFEI
COMSPEC	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK	NO
HOMEDRIVE	C:
HOMEPATH	\Documents and Settings\huoyafei
INCLUDE	D:\Qt\mingw\include
LIB	D:\Qt\mingw\lib

回到编辑界面，右击工程文件，在弹出的菜单上选择 Clean project，清空以前的编译信息。



然后运行 Run qmake，生成 Makefile 文件。



最后，点击 run 或者 build 都可，这时程序已经能正常编译运行了。

基于 Qt 4.6 的 Qt Creator 1.3.0 写 helloworld 程序注意事项（原创）

注意：下面指的是在 windows 下，linux 下的情况可进行相应改变

昨天 Qt 4.6 和 Qt Creator 1.3.0 正式版发布了，但是如果以前用过旧版本，就可能出一些问题。

1. 用 debug 方式

如果你以前用了 Qt 4.5 的 Qt Creator，并且将 QtCored4.dll，QtGui4.dll，mingwm10.dll 等文件放到了 C 盘的 system 文件夹下。那么请先将它们删除，不然编译不会通过。

编译完 helloworld 程序后，如果要直接执行 exe 文件，需要将安装目录（新版 Qt）下的 qt/bin 目录下的 QtCored4.dll，QtGui4.dll，mingwm10.dll，和 libgcc_s_dw2-1.dll（这个是新增的）文件放在 exe 文件夹中。或者将它们放到系统的 system 文件夹下。

2. 选择 release 方式

编译程序后生成 exe 文件

1. 需要 Qt 安装目录下的 qt/bin 目录中的 QtGui4.dll，Qt Core4.dll，libgcc_s_dw2-1.dll 以及 mingwm10.dll 四个文件的支持，将它们拷贝到 exe 文件目录下。

2. 程序中默认只支持 png 图片，如果使用了 gif，jpg 等格式的文件是显示不出来的。需要将 Qt 安装目录下的 qt/plugins/ 目录中的 imageformats 文件夹拷贝到 exe 文件目录下（注意是整个文件夹）。而 imageformats 文件夹中只需要保留你需要的文件，例如你只需要支持 gif 文件，就只保留 qgif4.dll 即可。

‘Qt Creator 发布 release 软件相关注意事项（原创）

注意：环境是 windows

选择 release 编译程序后生成 exe 文件

1. 需要 Qt 安装目录下的 qt/bin 目录中的 QtGui4.dll 和 Qt Core4.dll 以及 mingwm10.dll 三个文件的支持，将它们拷贝到 exe 文件目录下。
2. 程序中默认只支持 png 图片，如果使用了 gif, jpg 等格式的文件是显示不出来的。需要将 Qt 安装目录下的 qt/plugins/目录中的 imageformats 文件夹拷贝到 exe 文件目录下（注意是整个文件夹）。而 imageformats 文件夹中只需要保留你需要的文件，例如你只需要支持 gif 文件，就只保留 qgif4.dll 即可。

Qt Creator 的 error: collect2: ld returned 1 exit status 问题

利用 Qt Creator 1.2.1 (Built on Sep 30 2009 at 05:21:42) 编译程序经常会出现 **error: collect2: ld returned 1 exit status** 的错误，但是自己的程序没有一点问题，怎么回事呢？

如果这时退出软件，再重新进入，打开刚才的工程，重新编译，就不会出现刚才的错误了。这应该是 Qt Creator 软件的问题吧！

后来发现是因为上次执行的程序还在运行，你打开 windows 的任务管理器中的进程可以看见你刚才运行的程序还在执行，我们看不见，是因为它在后台执行着。出现这个现象，是因为你写的代码的问题，比如在 main 函数里用了 `w.show();` 语句，就可能出现界面一闪而过，但它并没有关闭，而是在后台运行，所以再次运行时就会出错。我们可以在资源管理器中将该进程关闭，或者像上面那样直接关闭 Qt Creator。

示例：

```
#include <QtGui/QApplication>

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

执行后就会在后台运行。这时如果修改了代码再次运行程序，就会出现上面的错误。

在任务管理器中可以看见自己的程序：



将该进程结束，然后在重新运行，就不会出错了。

正确的代码应该这样写：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    loginDlg m;
    Widget w;
    if(m.exec()==QDialog::Accepted)
    {
        w.show();
        return a.exec();
    }
    else return 0; //关闭整个程序
}
```

这样新建的对象w就不是局部变量了，这样运行程序w表示的窗口不会一闪而过，会一直显示。程序也不会再出现上面的错误了。

QT 常用问题解答(转)

本文是我前几天一个网友告诉我的，当时看了感觉好，就保存下来。今天再次查看，感觉有必要把文章分享给各位学习 QT 的朋友，因为网上好用的 QT 资源真的好少。

1、如何在窗体关闭前自行判断是否可关闭

答：重新实现这个窗体的 `closeEvent()` 函数，加入判断操作

Quote:

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (maybeSave())
    {
        writeSettings();
        event->accept();
    }
    else
    {
        event->ignore();
    }
}
```

2、如何用打开和保存文件对话

答：使用 `QFileDialog`

Quote:

```
QString fileName = QFileDialog::getOpenFileName(this);
if (!fileName.isEmpty())
{
    loadFile(fileName);
}
```

Quote:

```
    QString fileName = QFileDialog::getSaveFileName(this);
    if (fileName.isEmpty())
    {
        return false;
    }
```

3、如何创建 Actions(可在菜单和工具栏里使用这些 Action)

答：

Quote:

```
newAct = new QAction(QIcon(":/images/new.png"), tr("&New"), this);
newAct->setShortcut(tr("Ctrl+N"));
newAct->setStatusTip(tr("Create a new file"));
connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

openAct = new QAction(QIcon(":/images/open.png"), tr("&Open..."), this);
openAct->setShortcut(tr("Ctrl+O"));
openAct->setStatusTip(tr("Open an existing file"));
connect(openAct, SIGNAL(triggered()), this, SLOT(open()));

saveAct = new QAction(QIcon(":/images/save.png"), tr("&Save"), this);
saveAct->setShortcut(tr("Ctrl+S"));
saveAct->setStatusTip(tr("Save the document to disk"));
connect(saveAct, SIGNAL(triggered()), this, SLOT(save()));

saveAsAct = new QAction(tr("Save &As..."), this);
saveAsAct->setStatusTip(tr("Save the document under a new name"));
connect(saveAsAct, SIGNAL(triggered()), this, SLOT(saveAs()));

exitAct = new QAction(tr("E&xit"), this);
exitAct->setShortcut(tr("Ctrl+Q"));
exitAct->setStatusTip(tr("Exit the application"));
connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));

cutAct = new QAction(QIcon(":/images/cut.png"), tr("Cu&t"), this);
cutAct->setShortcut(tr("Ctrl+X"));
cutAct->setStatusTip(tr("Cut the current selection's contents to the "
"clipboard"));
connect(cutAct, SIGNAL(triggered()), textEdit, SLOT(cut()));

copyAct = new QAction(QIcon(":/images/copy.png"), tr("&Copy"), this);
copyAct->setShortcut(tr("Ctrl+C"));
copyAct->setStatusTip(tr("Copy the current selection's contents to the "
"clipboard"));
connect(copyAct, SIGNAL(triggered()), textEdit, SLOT(copy()));

pasteAct = new QAction(QIcon(":/images/paste.png"), tr("&Paste"), this);
pasteAct->setShortcut(tr("Ctrl+V"));
pasteAct->setStatusTip(tr("Paste the clipboard's contents into the "
"current "
"selection"));
connect(pasteAct, SIGNAL(triggered()), textEdit, SLOT(paste()));
```



```
aboutAct = new QAction(tr("&About"), this);
aboutAct->setStatusTip(tr("Show the application's About box"));
connect(aboutAct, SIGNAL(triggered()), this, SLOT(about()));

aboutQtAct = new QAction(tr("About &Qt"), this);
aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));
connect(aboutQtAct, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
```

4、如何创建主菜单

答：采用上面的 QAction 的帮助，创建主菜单

Quote:

```
fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(newAct);
fileMenu->addAction(openAct);
fileMenu->addAction(saveAct);
fileMenu->addAction(saveAsAct);
fileMenu->addSeparator();
fileMenu->addAction(exitAct);

editMenu = menuBar()->addMenu(tr("&Edit"));
editMenu->addAction(cutAct);
editMenu->addAction(copyAct);
editMenu->addAction(pasteAct);

menuBar()->addSeparator();

helpMenu = menuBar()->addMenu(tr("&Help"));
helpMenu->addAction(aboutAct);
helpMenu->addAction(aboutQtAct);
```

5、如何创建工具栏

答：采用上面的 QAction 的帮助，创建工具栏

Quote:

```
fileToolBar = addToolBar(tr("File"));
fileToolBar->addAction(newAct);
fileToolBar->addAction(openAct);
fileToolBar->addAction(saveAct);

editToolBar = addToolBar(tr("Edit"));
editToolBar->addAction(cutAct);
```



```
editToolBar->addAction(copyAct);  
editToolBar->addAction(pasteAct);
```

6、如何使用配置文件保存配置

答：使用 QSettings 类

Quote:

```
QSettings settings("Trolltech", "Application Example");  
QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();  
QSize size = settings.value("size", QSize(400, 400)).toSize();
```

Quote:

```
QSettings settings("Trolltech", "Application Example");  
settings.setValue("pos", pos());  
settings.setValue("size", size());
```

7、如何使用警告、信息等对话框

答：使用 QMessageBox 类的静态方法

Quote:

```
int ret = QMessageBox::warning(this, tr("Application"),  
    tr("The document has been modified.\n"  
    "Do you want to save your changes?"),  
    QMessageBox::Yes | QMessageBox::Default,  
    QMessageBox::No,  
    QMessageBox::Cancel | QMessageBox::Escape);  
if (ret == QMessageBox::Yes)  
    return save();  
else if (ret == QMessageBox::Cancel)  
    return false;
```

8、如何使通用对话框中文化

答：对话框的中文化

比 如说，QColorDialog 的与文字相关的部分，主要在 qcolordialog.cpp 文件中，我们可以从 qcolordialog.cpp 用 lupdate 生成一个 ts 文件，然后用自定义这个 ts 文件的翻译，再用 lrelease 生成一个.qm 文件，当然了，主程序就要改变要支持多国语言了， 使用这个.qm 文件就可以了。

另外，还有一个更快的方法，在源代码解开后有一个目录 translations，下面有一些.ts, .qm 文件，我们拷贝一个：

Quote:

```
cp src/translations/qt_untranslated.ts ./qt_zh_CN.ts
```

然后，我们就用 Linguist 打开这个 qt_zh_CN.ts，进行翻译了，翻译完成后，保存后，再用 lrelease 命令生成 qt_zh_CN.qm，这样，我们把它加入到我们的 qt project 中，那些系统的对话框，菜单等等其它的默认是英文的东西就能显示成中文了。

9、在 Windows 下 Qt 里为什么没有终端输出？

答：把下面的配置项加入到.pro 文件中

Quote:

```
win32:CONFIG += console
```

10、Qt 4 for X11 OpenSource 版如何静态链接？

答：编译安装的时候加上-static 选项

Quote:

```
./configure -static      // 一定要加 static 选项  
gmake  
gmake install
```

然后，在 Makefile 文件中加 static 选项或者在.pro 文件中加上 QMAKE_LFLAGS += -static，就可以连接静态库了。

11、想在源代码中直接使用中文，而不使用 tr() 函数进行转换，怎么办？

答：在 main 函数中加入下面三条语句，但并不提倡

Quote:

```
QTextCodec::setCodecForLocale(QTextCodec::codecForName("UTF-8"));  
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("UTF-8"));  
QTextCodec::setCodecForTr(QTextCodec::codecForName("UTF-8"));
```

或者

Quote:

```
QTextCodec::setCodecForLocale(QTextCodec::codecForName("GBK"));  
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("GBK"));  
QTextCodec::setCodecForTr(QTextCodec::codecForName("GBK"));
```

使用 GBK 还是使用 UTF-8，依源文件中汉字使用的内码而定
这样，就可在源文件中直接使用中文，比如：

Quote:

```
QMessageBox::information(NULL, "信息", "关于本软件的演示信息",  
QMessageBox::Ok, QMessageBox::NoButtons);
```

12、为什么将开发的使用数据库的程序发布到其它机器就连接不上数据库？

答：这是由于程序找不到数据库插件而致，可照如下解决方法：

在 main 函数中加入下面语句：

Quote:

```
QApplication::addLibraryPath(strPluginsPath");
```

strPluginsPath 是插件所在目录，比如此目录为/myapplication/plugins
则将需要的 sql 驱动，比如 qsqlmysql.dll, qsqlodbc.dll 或对应的 .so 文件放到

/myapplication/plugins/sqldrivers/

目录下面就行了

这是一种解决方法，还有一种通用的解决方法，即在可执行文件目录下写
qt.conf 文件，把系统相关的一些目录配置写到 qt.conf 文件里，详细情况请参考 Qt Document Reference 里的 qt.conf 部分

13、如何创建 QT 使用的 DLL(.so) 以及如何使用此 DLL(.so)

答：创建 DLL 时其工程使用 lib 模板

Quote:

```
TEMPLATE=lib
```

而源文件则和使用普通的源文件一样，注意把头文件和源文件分开，因为在其它
程序使用此 DLL 时需要此头文件

在使用此 DLL 时，则在此工程源文件中引入 DLL 头文件，并在 .pro 文件中加入
下面配置项：

Quote:

```
LIBS += -Lyourdlllibpath -lyourdlllibname
```

Windows 下和 Linux 下同样(Windows 下生成的 DLL 文件名为 yourdlllibname.dll
而在 Linux 下生成的为 libyourdlllibname.so。注意，关于 DLL 程序的写法，
遵从各平台级编译器所定的规则。

14、如何启动一个外部程序

答：1、使用 `QProcess::startDetached()` 方法，启动外部程序后立即返回；
2、使用 `QProcess::execute()`，不过使用此方法时程序会阻塞直到此方法执行的程序结束后返回，这时候可使用 `QProcess` 和 `QThread` 这两个类结合使用的方法来处理，以防止在主线程中调用而导致阻塞的情况
先从 `QThread` 继承一个类，重新实现 `run()` 函数：

Quote:

```
class MyThread : public QThread
{
public:
    void run();
};

void MyThread::run()
{
    QProcess::execute("notepad.exe");
}
```

这样，在使用的时候则可定义一个 `MyThread` 类型的成员变量，使用时调用其 `start()` 方法：

Quote:

```
class .....
{.....
    MyThread thread;
    .....
};

.....
thread.start();
```

15、如何打印报表

答：Qt 目前对报表打印支持的库还很少，不过有种变通的方法，就是使用 XML+XSLT+XSL-FO 来进行报表设计，XML 输出数据，用 XSLT 将 XML 数据转换为 XSL-FO 格式的报表，由于现在的浏览器不直接支持 XSL-FO 格式的显示，所以暂时可用工具 (Apache FOP, Java 做的) 将 XSL-FO 转换为 PDF 文档来进行打印，转换和打印由 FOP 来做，生成 XSL-FO 格式的报表可以由 Qt 来生成，也可以由其它内容转换过来，比如有工具 (html2fo) 将 HTML 转换为 XSL-FO。

16、如何在系统托盘区显示图标

答：在 4.2 及其以上版本中使用 `QSystemTrayIcon` 类来实现

17、怎样将日志输出到文件中

答：（myer 提供）

Quote:

```
void myMessageOutput( QtMsgType type, const char *msg )
{
    switch ( type ) {
    case QtDebugMsg:
        //写入文件;
        break;
    case QtWarningMsg:
        break;
    case QtFatalMsg:
        abort();
    }
}

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

qDebug(), qWarning(), qFatal() 分别对应以上三种 type。

18、如何将图像编译到可执行程序中去

答：使用.qrc 文件

写.qrc 文件，例如：

res.qrc

Quote:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
    <file>images/cut.png</file>
    <file>images/new.png</file>
    <file>images/open.png</file>
    <file>images/paste.png</file>
    <file>images/save.png</file>
```

```
</qresource>
</RCC>
```

然后在.pro 中加入下面代码:

Quote:

```
RESOURCES          = res.qrc
```

在程序中使用:

Quote:

```
...
:images/copy.png
...
```

19、如何制作不规则形状的窗体或部件

答: 请参考下面的帖子

<http://www.qtcn.org/bbs/read.php?tid=8681>

20、删除数据库时出现“QSqlDatabasePrivate::removeDatabase: connection 'xxxx' is still in use, all queries will cease to work”该如何处理

答: 出现此种错误是因为使用了连接名字为 xxxx 的变量作用域没有结束, 解决方法是在所有使用了 xxxx 连接的数据库组件变量的作用域都结束后再使用 QSqlDatabase::removeDatabase("xxxx") 来删除连接。

21、如何显示一个图片并使其随窗体同步缩放

答: 下面给出一个从 QWidget 派生的类 ImageWidget, 来设置其背景为一个图片, 并可随着窗体改变而改变, 其实从下面的代码中可以引申出其它许多方法, 如果需要的话, 可以从这个类再派生出其它类来使用。

头文件: ImageWidget.hpp

Quote:

```
#ifndef IMAGEWIDGET_HPP
#define IMAGEWIDGET_HPP
```

```
#include <QtCore>
#include <QtGui>
```

```
class ImageWidget : public QWidget
{
    Q_OBJECT
public:
```

```

ImageWidget(QWidget *parent = 0, Qt::WindowFlags f = 0);
virtual ~ImageWidget();
protected:
void resizeEvent(QResizeEvent *event);
private:
QImage _image;
};

#endif

```

CPP 文件: ImageWidget.cpp

Quote:

```
#include "ImageWidget.hpp"
```

```

ImageWidget::ImageWidget(QWidget *parent, Qt::WindowFlags f)
: QWidget(parent, f)
{
_image.load("image/image_background");
setAutoFillBackground(true);    // 这个属性一定要设置
QPalette pal(palette());
pal.setBrush(QPalette::Window,
QBrush(_image.scaled(size(), Qt::IgnoreAspectRatio,
Qt::SmoothTransformation)));
setPalette(pal);
}

```

```

ImageWidget::~ImageWidget()
{
}

```

// 随着窗体变化而设置背景

```

void ImageWidget::resizeEvent(QResizeEvent *event)
{
QWidget::resizeEvent(event);
QPalette pal(palette());
pal.setBrush(QPalette::Window,
QBrush(_image.scaled(event->size(), Qt::IgnoreAspectRatio,
Qt::SmoothTransformation)));
setPalette(pal);
}

```

22、Windows 下如何读串口信息

答:可通过注册表来读 qt4.1.0 读取注册表得到 串口信息的方法!