

# 第5篇 (基础) Qt布局管理器

一、完善菜单

二、向工具栏添加菜单图标

三、布局管理器

四、实现新建文件、文件保存和另存为功能

五、实现打开、关闭、退出、撤销、复制、剪切、粘贴等功能

六、添加查找对话框

七、实现查找功能

八、添加动作状态提示

九、显示其他临时信息

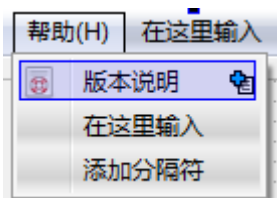
十、显示永久信息

代码地址：<https://github.com/Lornatang/QtStartQuicklyTutorial/tree/main/Example05>












## 一、完善菜单

1. 新建Qt Gui应用，项目名称为 `myMainWindow`，基类选择 `QMainWindow`，类名为 `MainWindow`。

2. 完成后，在设计模式添加菜单项，并添加资源文件，向其中添加菜单图标。最终各个菜单如下图所示。

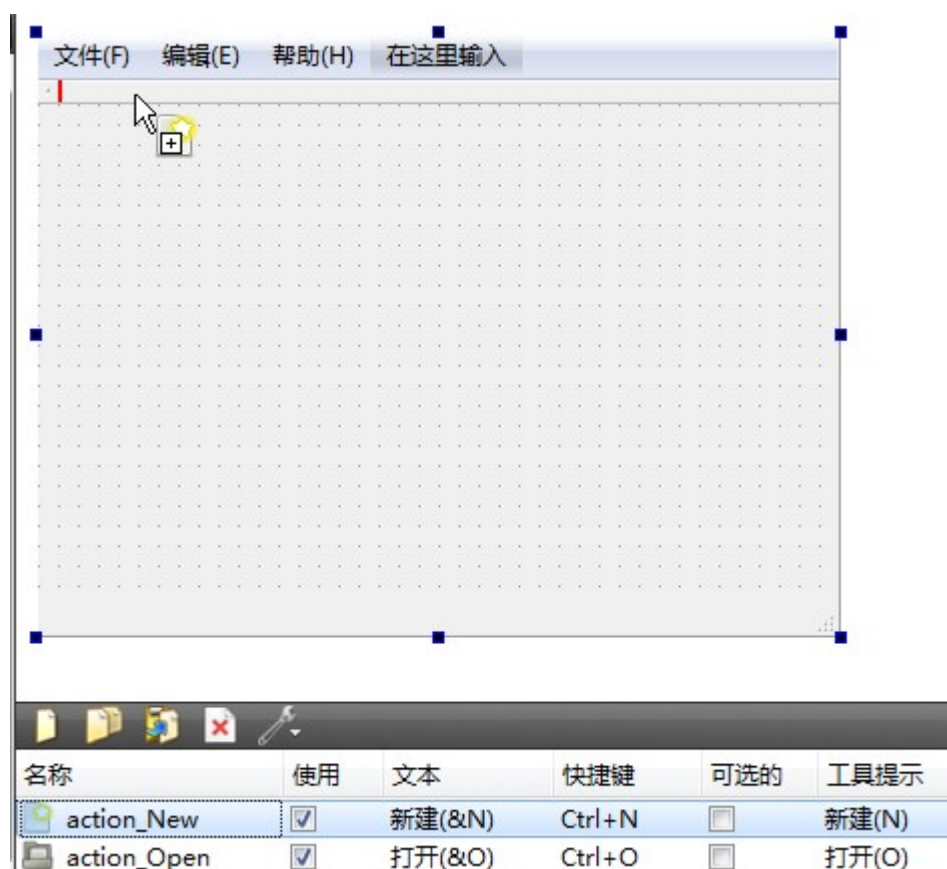


在 **Action** 编辑器中修改动作的对象名称、图标和快捷键，最终如下图所示。

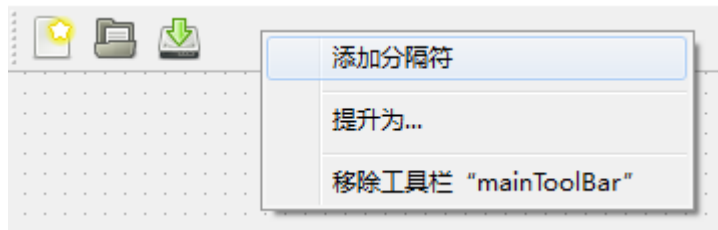
名称	使用	文本	快捷键	可选的	工具提示
 action_New	<input checked="" type="checkbox"/>	新建(&N)	Ctrl+N	<input type="checkbox"/>	新建(N)
 action_Open	<input checked="" type="checkbox"/>	打开(&O)	Ctrl+O	<input type="checkbox"/>	打开(O)
 action_Close	<input checked="" type="checkbox"/>	关闭(&C)	Ctrl+W	<input type="checkbox"/>	关闭(C)
 action_Save	<input checked="" type="checkbox"/>	保存(&S)	Ctrl+S	<input type="checkbox"/>	保存(S)
 action_SaveAs	<input checked="" type="checkbox"/>	另存为(&A)		<input type="checkbox"/>	另存为(A)
 action_Exit	<input checked="" type="checkbox"/>	退出(&X)	Ctrl+Q	<input type="checkbox"/>	退出(X)
 action_Undo	<input checked="" type="checkbox"/>	撤销(&Z)	Ctrl+Z	<input type="checkbox"/>	撤销(Z)
 action_Cut	<input checked="" type="checkbox"/>	剪切(&X)	Ctrl+X	<input type="checkbox"/>	剪切(X)
 action_Copy	<input checked="" type="checkbox"/>	复制(&C)	Ctrl+C	<input type="checkbox"/>	复制(C)
 action_Paste	<input checked="" type="checkbox"/>	粘贴(&V)	Ctrl+V	<input type="checkbox"/>	粘贴(V)
 action_Find	<input checked="" type="checkbox"/>	查找(&F)	Ctrl+F	<input type="checkbox"/>	查找(F)
 action_Help	<input checked="" type="checkbox"/>	版本说明	Ctrl+H	<input type="checkbox"/>	版本说明

## 二、向工具栏添加菜单图标

可以将动作编辑器中的动作拖动到工具栏中作为快捷图标使用，如下图所示。



可以在工具栏上点击鼠标右键来添加分隔符，如下图所示。

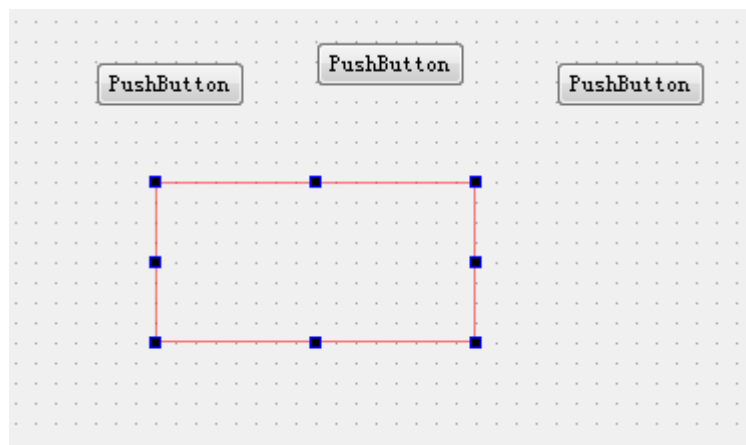


最终工具栏如下图所示。

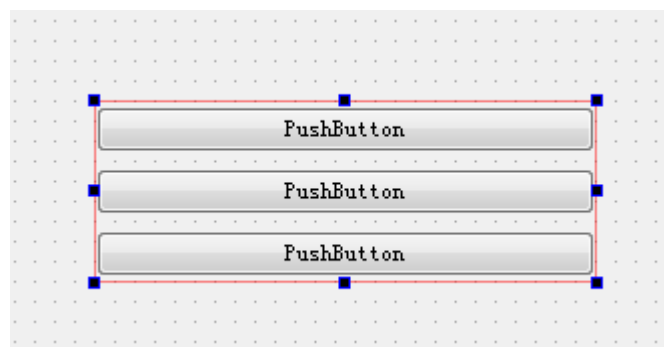


## 三、布局管理器

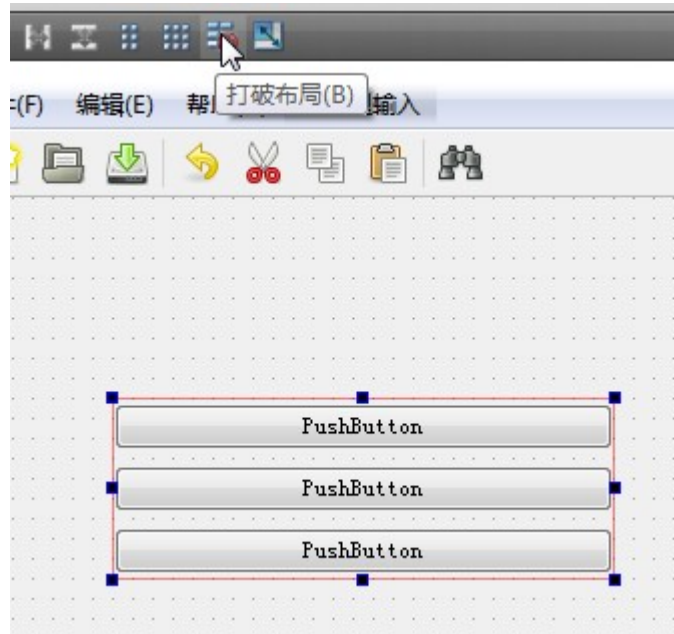
1. 从左边控件栏中拖入三个按钮 `PushButton` 和一个 `Vertical Layout`（垂直布局管理器）到界面上，如下图所示。



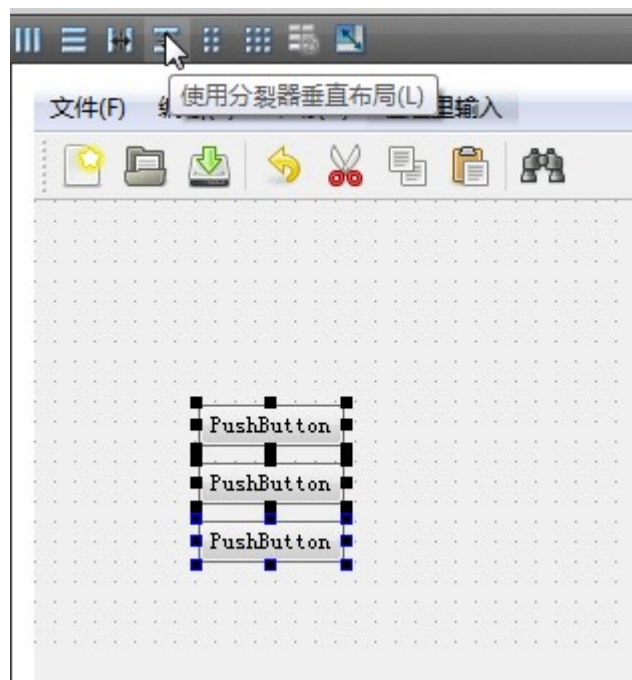
然后将三个按钮拖入到布局管理器中，这时三个按钮就会自动垂直排列，并且进行水平拉伸，无论如何改变布局管理器的大小，按钮总是水平方向变化。如下图所示。



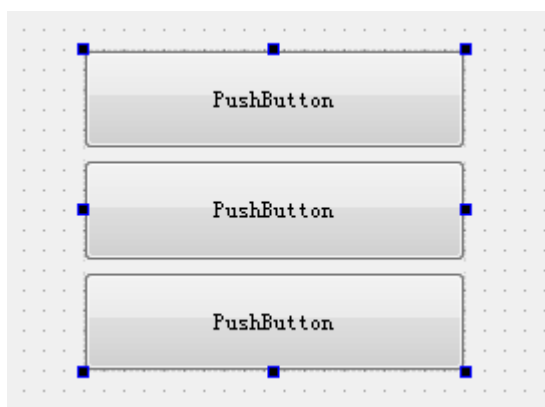
2. 我们可以选中布局管理器，然后按下上方工具栏中的“打破布局”按钮来删除布局管理器。（当然也可以先将三个按钮移出，然后按下 `Delete` 键来删除布局管理器，如果不移出按钮，那么会将它们同时删除。）如下图所示。



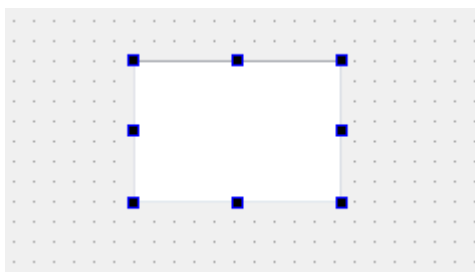
3. 下面我们使用分裂器（`QSplitter`）来进行布局，先同时选中三个按钮，然后按下上方工具栏中的“使用分裂器垂直布局”按钮，如下图所示。



然后我们进行放大，可以发现，使用分裂器按钮纵向是可以变大的，这就是分裂器和布局管理器的重要区别。如下图所示。



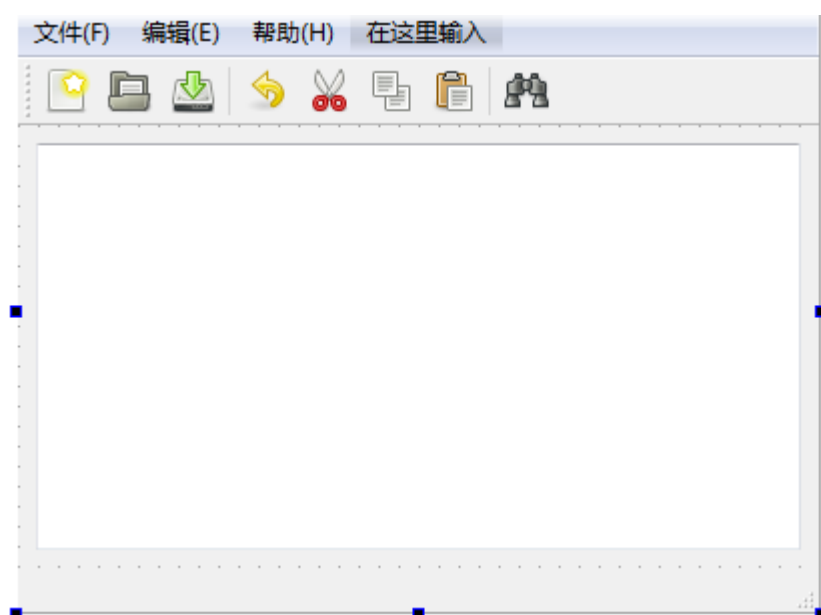
4. 布局管理器除了可以对部件进行布局以外，还有个重要用途，就是使部件随着窗口的大小变化而变化。我们删除界面上的部件，然后拖入一个文本编辑器 `Text Edit` 部件。如下图所示。



然后我们在界面上点击鼠标右键，选择布局 → 栅格布局（或者使用快捷键 `Ctrl+G`）。



这时整个文本编辑器部件就会填充中央区域。如下图所示。现在运行程序，可以发现，无论怎样拉伸窗口，文本编辑器总是填充整个中央区域。





这一篇中我们主要介绍了布局管理器的应用，而且都是在设计模式对布局管理器的使用。这里使用三种方式进行了演示：从控件栏中拖入布局管理器；在工具栏中使用图标；还有使用右键菜单（当然还有快捷键的方式）。

## 四、实现新建文件、文件保存和另存为功能

1. 首先来分析下整个流程，当新建文件时，要考虑是否保存正在编辑的文件，如果需要保存，还要根据该文件以前是否保存过来进行保存或者另存为操作。下面我们根据这里的分析来添加需要的函数和对象。

2. 打开上一篇完成的项目，然后先在 `main.cpp` 文件中添加代码来保证代码中可以使用中文字符。首先添加 `#include <QTextCodec>` 头文件包含，然后在主函数中添加如下代码：

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

3. 在 `mainwindow.h` 文件中添加 `public` 函数声明：

```
void newFile();    // 新建操作
bool maybeSave(); // 判断是否需要保存
bool save();       // 保存操作
bool saveAs();     // 另存为操作
bool saveFile(const QString &fileName); // 保存文件
```

这里的几个函数就是用来完成功能逻辑的，下面我们会添加它们的定义来实现相应的功能。因为这几个功能联系紧密，所以这几个函数会相互调用。

4. 然后添加 `private` 变量定义：

```
// 为真表示文件没有保存过，为假表示文件已经被保存过了
bool isUntitled;
// 保存当前文件的路径
QString curFile;
```

这里的 `isUntitled` 是一个标志，用来判断文档是否被保存过。而 `curFile` 用来保存当前打开的文件的路径。

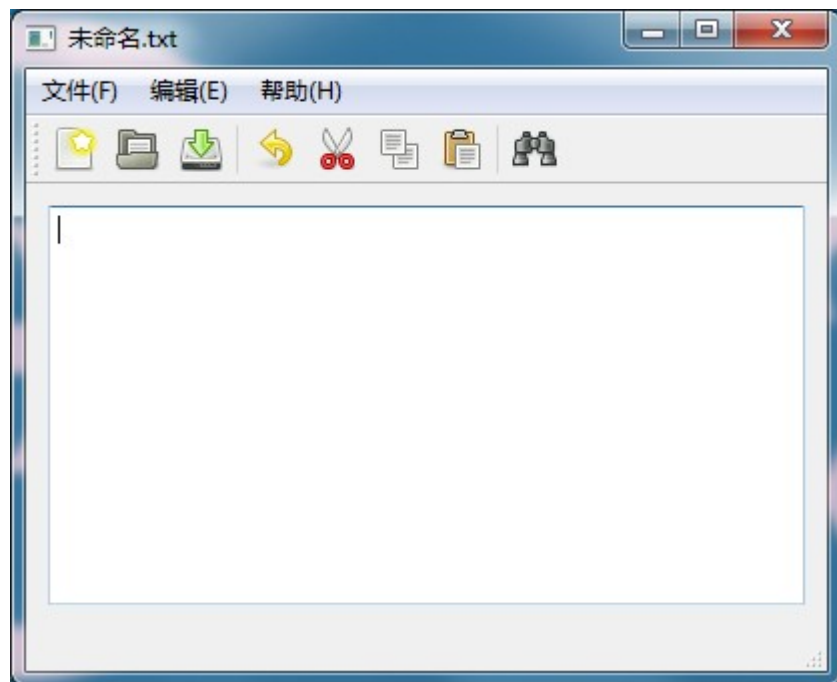
5. 下面到 `mainwindow.cpp` 文件，先添加头文件：

```
#include <QMessageBox>
#include <QPushButton>
#include <QFileDialog>
```



```
#include <QTextStream>
然后在构造函数中添加如下代码来进行一些初始化操作：
// 初始化文件为未保存状态
isUntitled = true;
// 初始化文件名为"未命名.txt"
curFile = tr("未命名.txt");
// 初始化窗口标题为文件名
setWindowTitle(curFile);
```

这里设置了在启动程序时窗口标题显示文件的名称，效果如下图所示。



6. 下面添加那几个函数的定义。

首先是新建文件操作的函数：

```
void MainWindow::newFile()
{
    if (maybeSave()) {
        isUntitled = true;
        curFile = tr("未命名.txt");
        setWindowTitle(curFile);
        ui->textEdit->clear();
        ui->textEdit->setVisible(true);
    }
}
```

这里先使用 `maybeSave()` 来判断文档是否需要保存，如果已经保存完了，则新建文档，并进行初始化。下面是 `maybeSave()` 函数的定义：

```

bool MainWindow::maybeSave()
{
    // 如果文档被更改了
    if (ui->textEdit->document()->isModified()) {
        // 自定义一个警告对话框
        QMessageBox box;
        box.setWindowTitle(tr("警告"));
        box.setIcon(QMessageBox::Warning);
        box.setText(curFile + tr(" 尚未保存, 是否保存?"));
        QPushButton *yesBtn = box.addButton(tr("是(&Y)"),
            QMessageBox::YesRole);
        box.addButton(tr("否(&N)"), QMessageBox::NoRole);
        QPushButton *cancelBut = box.addButton(tr("取消"),
            QMessageBox::RejectRole);

        box.exec();
        if (box.clickedButton() == yesBtn)
            return save();
        else if (box.clickedButton() == cancelBut)
            return false;
    }
    // 如果文档没有被更改, 则直接返回true
    return true;
}

```

这里先使用了 `isModified()` 来判断文档是否被更改了, 如果被更改了, 则弹出对话框让用户选择是否进行保存, 或者取消操作。如果取消操作, 那么就返回 `false`, 什么都不执行。下面是 `save()` 函数的定义:

```

bool MainWindow::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

```

这里如果文档以前没有保存过, 那么执行另存为操作 `saveAs()`, 如果已经保存过, 那么调用 `saveFile()` 执行文件保存操作。下面是 `saveAs()` 函数的定义:

```

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
        tr("另存为"), curFile);
    if (fileName.isEmpty()) return false;
    return saveFile(fileName);
}

```

这里使用 `QFileDialog` 来实现了一个另存为对话框，并且获取了文件的路径，然后使用文件路径来保存文件。下面是 `saveFile()` 函数的定义：

```
bool MainWindow::saveFile(const QString &fileName)
{
    QFile file(fileName);

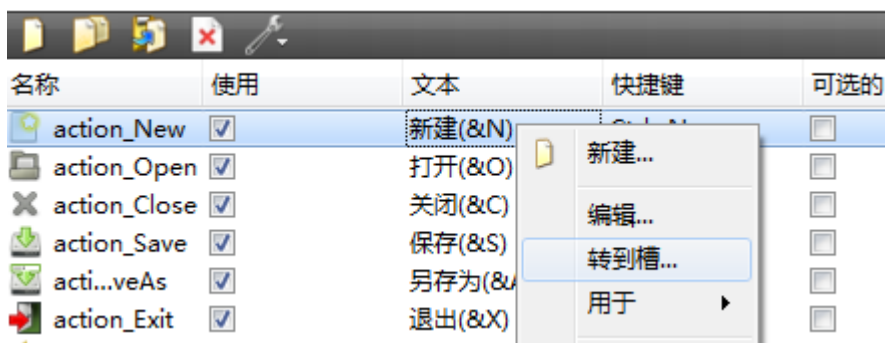
    if (!file.open(QFile::WriteOnly | QFile::Text)) {

        // %1和%2分别对应后面arg两个参数，/n起换行的作用
        QMessageBox::warning(this, tr("多文档编辑器"),
                               tr("无法写入文件 %1:/n %2")
                               .arg(fileName).arg(file.errorString()));
        return false;
    }
    QTextStream out(&file);
    // 鼠标指针变为等待状态
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << ui->textEdit->toPlainText();
    // 鼠标指针恢复原来的状态
    QApplication::restoreOverrideCursor();
    isUntitled = false;
    // 获得文件的标准路径
    curFile = QFile::info(fileName).canonicalFilePath();
    setWindowTitle(curFile);
    return true;
}
```

该函数执行真正的文件保存操作。先是使用一个 `QFile` 类对象来指向要保存的文件，然后将其使用写入方式打开。打开后再使用 `QTextStream` 文本流将编辑器中的内容写入到文件中。

这里使用了很多新的类，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。

7. 设置菜单功能。双击 `mainwindow.ui` 文件，在图形界面窗口下面的 `Action` 编辑器里，我们右击“新建”菜单一条，选择“转到槽”，然后选择 `triggered()`，进入其触发事件槽。如下图所示。



同理，进入其他两个菜单的槽，将相应的操作的函数写入槽中。最终代码如下：

```
void MainWindow::on_action_New_triggered()
{
    newFile();
}
void MainWindow::on_action_Save_triggered()
{
    save();
}
void MainWindow::on_action_SaveAs_triggered()
{
    saveAs();
}
```

现在运行程序，已经能够实现新建文件，保存文件，文件另存为的功能了。

## 五、实现打开、关闭、退出、撤销、复制、剪切、粘贴等功能

先到 `mainwindow.h` 文件中添加 `public` 函数声明：

```
bool loadFile(const QString &fileName); // 加载文件
```

然后到 `mainwindow.cpp` 文件中添加该函数的定义：

```
bool MainWindow::loadFile(const QString &fileName)
{
    QFile file(fileName); // 新建QFile对象
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("多文档编辑器"),
            tr("无法读取文件 %1:\n%2.")
                .arg(fileName).arg(file.errorString()));
        return false; // 只读方式打开文件，出错则提示，并返回false
    }
}
```

```

    QTextStream in(&file); // 新建文本流对象
    QApplication::setOverrideCursor(Qt::WaitCursor);
    // 读取文件的全部文本内容，并添加到编辑器中
    ui->textEdit->setPlainText(in.readAll());      QApplication::restoreOverrideCursor
();

    // 设置当前文件
    curFile = QFile::fileName().canonicalFilePath();
    setWindowTitle(curFile);
    return true;
}

```

这里的操作和 `saveFile()` 函数是相似的。下面到设计模式，分别进入其他几个动作的触发信号的槽，更改如下：

```

// 打开动作
void MainWindow::on_action_Open_triggered()
{
    if (maybeSave()) {

        QString fileName = QFileDialog::getOpenFileName(this);

        // 如果文件名不为空，则加载文件
        if (!fileName.isEmpty()) {
            loadFile(fileName);
            ui->textEdit->setVisible(true);
        }
    }
}
// 关闭动作
void MainWindow::on_action_Close_triggered()
{
    if (maybeSave()) {
        ui->textEdit->setVisible(false);
    }
}
// 退出动作
void MainWindow::on_action_Exit_triggered()
{
    // 先执行关闭操作，再退出程序
    // qApp是指向应用程序的全局指针
    on_action_Close_triggered();
    qApp->quit();
}
// 撤销动作
void MainWindow::on_action_Undo_triggered()
{
    ui->textEdit->undo();
}
// 剪切动作
void MainWindow::on_action_Cut_triggered()
{
    ui->textEdit->cut();
}

```

```
// 复制动作
void MainWindow::on_action_Copy_triggered()
{
    ui->textEdit->copy();
}
// 粘贴动作
void MainWindow::on_action_Paste_triggered()
{
    ui->textEdit->paste();
}
```

这里可以看到，复制、粘贴等常用功能是QTextEdit已经实现的，我们只需要调用相应的函数。虽然实现了退出功能，但是，有时候会使用窗口标题栏的关闭按钮来关闭程序，这里我们需要使用关闭事件处理函数来实现相应的功能。

下面到mainwindow.h文件中，先添加头文件包含 `#include <QCloseEvent>`，然后添加函数声明：

```
protected:
    void closeEvent(QCloseEvent *event); // 关闭事件

然后到mainwindow.cpp文件中添加该函数的定义：
void MainWindow::closeEvent(QCloseEvent *event)
{
    // 如果maybeSave()函数返回true，则关闭程序
    if (maybeSave()) {
        event->accept();
    } else { // 否则忽略该事件
        event->ignore();
    }
}
```

关于事件的概念，会在后面的教程中讲解。

## 六、添加查找对话框

1. 我们继续在前一篇程序的基础之上进行更改。首先到 `mainwindow.h` 文件中添加类的前置声明（对于什么是前置声明，以及这样使用的好处，可以在网上百度）：

```
class QLineEdit;
class QDialog;
```

前置声明所在的位置跟头文件包含的位置相同。

然后在 `private` 中添加对象定义：

```
QLineEdit *findLineEdit;  
QDialog *findDlg;
```

下面再添加一个私有槽声明：

```
private slots:  
    void showFindText();
```

槽可以看做是一个函数，只不过可以和信号进行关联。

2. 下面到 `mainwindow.cpp` 文件中，因为前面在头文件中使用了类的前置声明，所以这里需要先添加头文件包含：

```
#include <QLineEdit>#include <QDialog>#include <QPushButton>
```

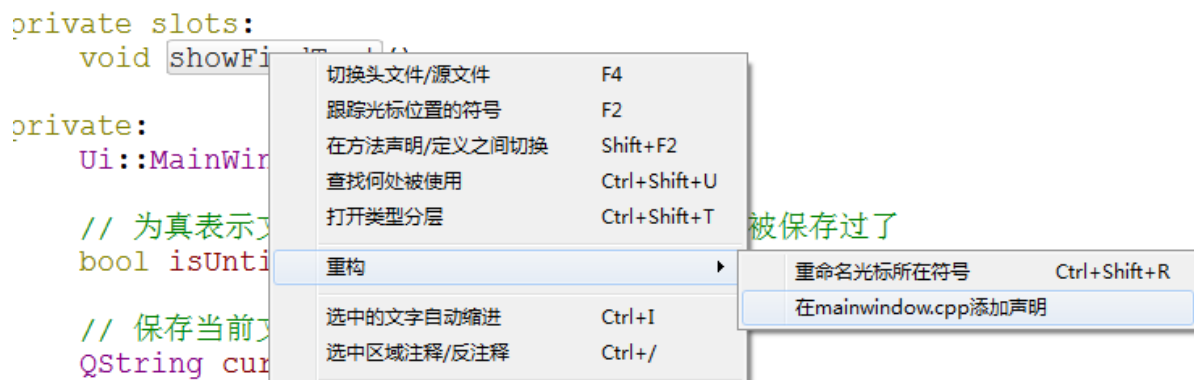
然后在构造函数中进行初始化操作，即添加如下代码：

```
findDlg = new QDialog(this);  
findDlg->setWindowTitle(tr("查找"));  
findLineEdit = new QLineEdit(findDlg);  
QPushButton *btn= new QPushButton(tr("查找下一个"), findDlg);  
QVBoxLayout *layout= new QVBoxLayout(findDlg);  
layout->addWidget(findLineEdit);  
layout->addWidget(btn);  
connect(btn, SIGNAL(clicked()), this, SLOT(showFindText()));
```

这里创建了一个对话框，然后将一个行编辑器和一个按钮放到了上面，并使用布局管理器进行布局。最后将按钮的单击信号关联到了自定义的显示查找到的文本槽上。下面来添加该槽的定义。

3. 这里先说一个可以快速从头文件声明处创建函数定义的方法。到 `mainwindow.h` 文件中，将鼠标定位到 `showFindText()` 函数上，然后点击右键，在弹出的菜单中选择“重构”→“在 `mainwindow.cpp` 添加声明”，或者直接使用 `Alt+Enter` 快捷键，这样就会直接在 `mainwindow.cpp` 文件中添加函数定义，并跳转到该函数处。如下图所示。





## 七、实现查找功能

下面我们来分步骤完成 `showFindText()` 函数。在讲解过程中会涉及一些很实用的功能的介绍。

1. 先在函数中添加一行代码来获取行编辑器中要查找的字符串。

```
void MainWindow::showFindText()
{
    QString str = findLineEdit->text();
}
```

2. 在下一行，我们先输入 `ui`，然后按下键盘上的 `>` 键，这时就会自动输入 `.` 或者 `->`，并且列出 `ui` 上所有可用部件的对象名。如下图所示。

```
void MainWindow::showFindText()
{
    QString str = findLineEdit->text();
    ui->
}
```

3. 我们要输入 `textEdit`，先输入 `t`，这时会自动弹出 `textEdit`，只需要按下回车键即可。如下图所示。

```
void MainWindow::showFindText()
{
    QString str = findLineEdit->text();
    ui->textEdit
}
```

4. 下面我们将光标放到 `textEdit` 上，这时就会出现 `QTextEdit` 类的简单介绍，如下图所示。



5. 按照提示，我们按下键盘上的 `F1` 键，就会在编辑器的右侧打开 `QTextEdit` 类的帮助文档。如下图所示。这时还可以按下上面的“切换至帮助模式”来进入到帮助模式中打开该文档。



6. 我们在该类的 `Public Functions` 公共函数列表中发现有一个 `find()` 函数。如下图所示。

```
QList<ExtraSelection> extraSelections () const
bool find ( const QString & exp, QTextDocument::FindFlags options = 0 )
QString fontFamily () const
```

7. 从字面意思上可以知道该函数应该是用于查找功能的，我们点击该函数进入到它的详细介绍处。如下图所示。

```
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.

8. 根据介绍可以知道该函数用于查询指定的 `exp` 字符串，如果找到了就将光标跳转到查找到的位置，如果没有找到就返回 `false`。这个函数还有一个

`QTextDocument::FindFlags` 参数，为了了解该参数的意思，我们点击该参数进入其详细介绍处。如下图所示。

```
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.

可以看到该参数是一个枚举变量，用来指定查找的方式，分别是向后查找、区分大小写、全词匹配等。如果不指定该参数，默认的是向前查找、不区分大小写、包含该字符串的词也可以查找到。这几个变量还可以使用 `|` 符号来一起使用。

9. 根据帮助，我们补充完该行代码：

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

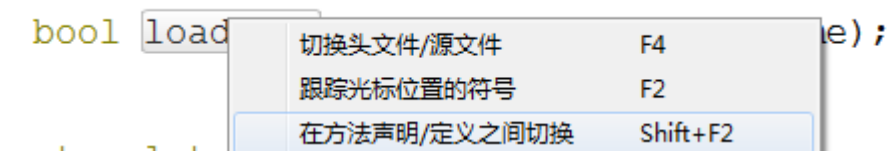
10. 这时已经能实现查找的功能了。但是我们刚才看到 `find` 的返回值类型是 `bool` 型，而且，我们也应该为查找不到字符串作出提示。将这行代码更改为：

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

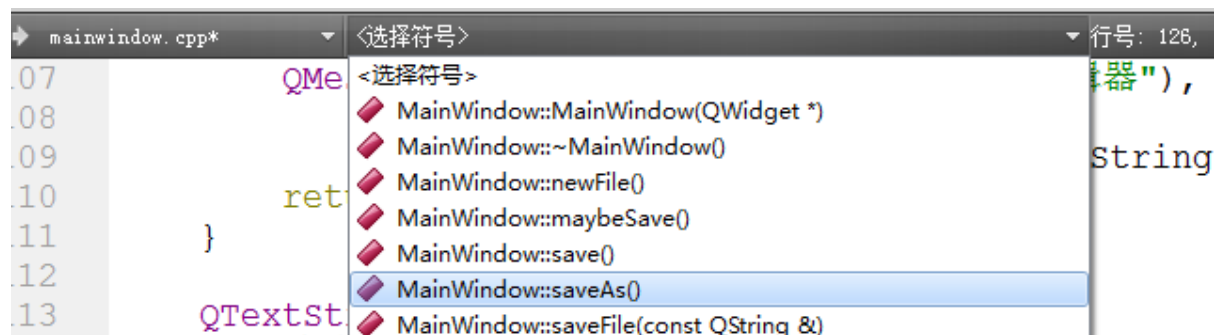
到这里查找函数就基本讲完了。

11. 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QtCreator中有几种快速定位函数的方法。

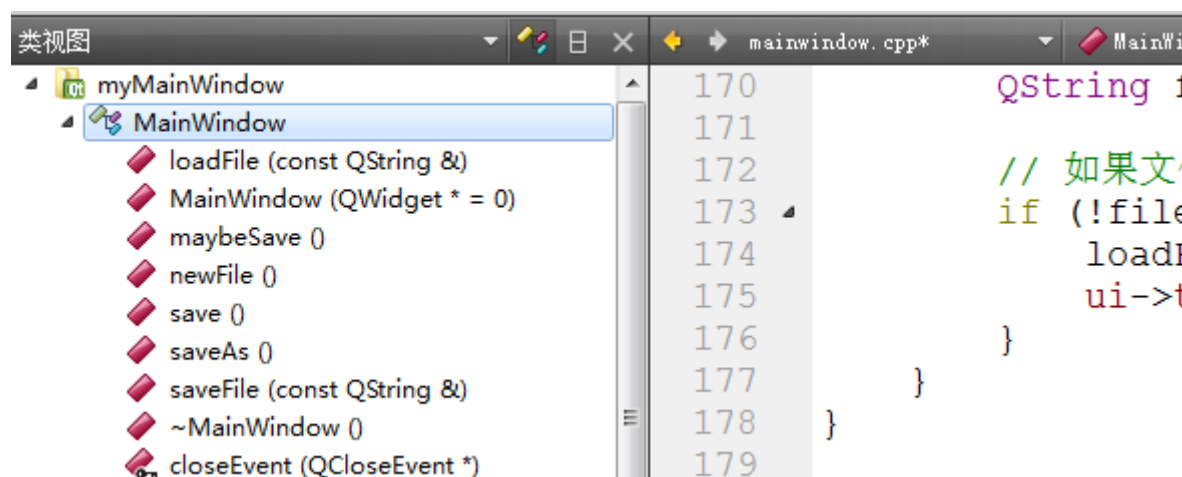
第一种，在函数声明的地方直接跳转到函数定义的地方。例如我们在 `mainwindow.h` 文件的 `loadFile()` 函数上点击鼠标右键，在弹出的菜单上选择“在方法声明/定义之间切换”，这时就会自动跳转到 `mainwindow.cpp` 文件中该函数的定义处。如下图所示。当然还可以反向使用。



第二种，快速查看一个文件里的所有函数。可以在编辑器正上方的下拉框里查看正在编辑的文件中所有的函数的列表，点击一个函数就会跳转到指定位置。如下图所示。



第三种，使用类视图或者大纲视图。在项目列表上面的下拉框中可以更改查看的内容，如果选择为类视图或者大纲，则会显示文件中所有的函数的列表。如下图所示。



第四种，使用查找功能查看函数的所有调用处。在一个函数名上点击鼠标右键，然后选择“查找何处被使用”菜单，这时就会在下面的搜索结果栏中显示该函数所有的使用位置。我们可以通过点击一个位置来跳转到该位置。如下图所示。

```
21
22     void newFile();    // 新建操作
23     bool maybeSave(); // 判断是否需要保存
24     bool save();       // 保存操作
25     bool saveAs();     // 另存为操作
26     bool saveFile(const QString &fileName);
27
28     bool loadFile(const QString &fileName);
29
30
31 private slots:
```

搜索结果    C++ 使用: MainWindow::newFile    找到3 个匹配

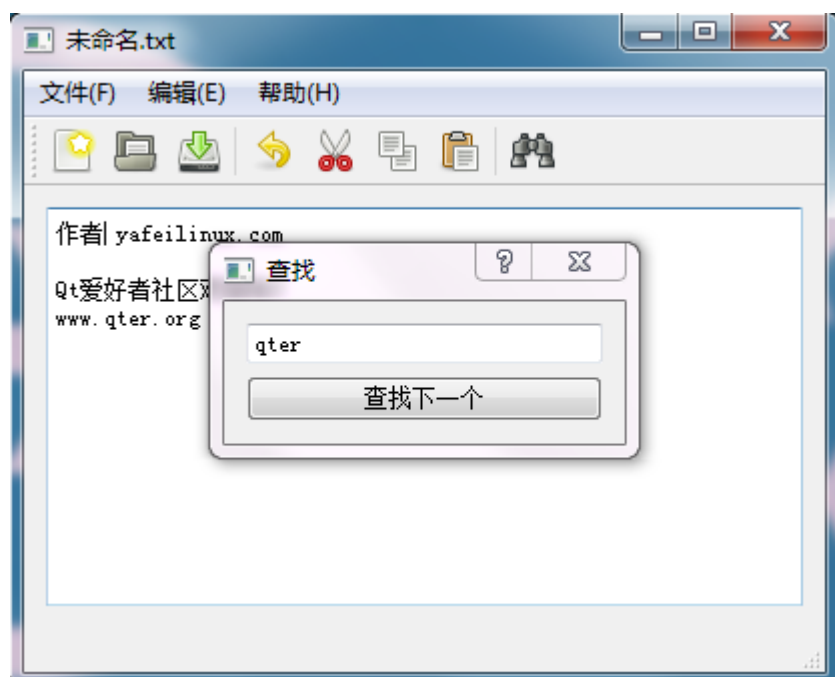
C++ 使用:    MainWindow::newFile

- ▾ F:\myMainWindow\mainwindow.cpp (2)
  - 45 void MainWindow::newFile()
  - 150 newFile();
- ▾ F:\myMainWindow\mainwindow.h (1)
  - 22 void newFile(); // 新建操作

12. 最后，我们来实现界面上的查找功能。从设计模式进入查找动作的触发信号的槽，更改如下：

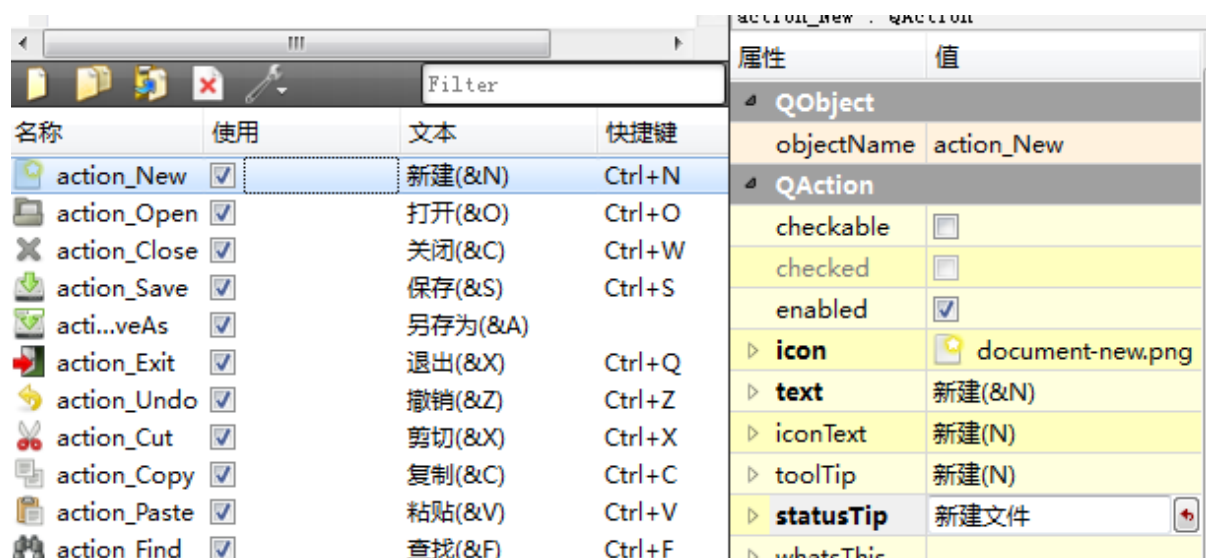
```
void MainWindow::on_action_Find_triggered()
{
    findDlg->show();
}
```

这时运行程序，效果如下图所示。

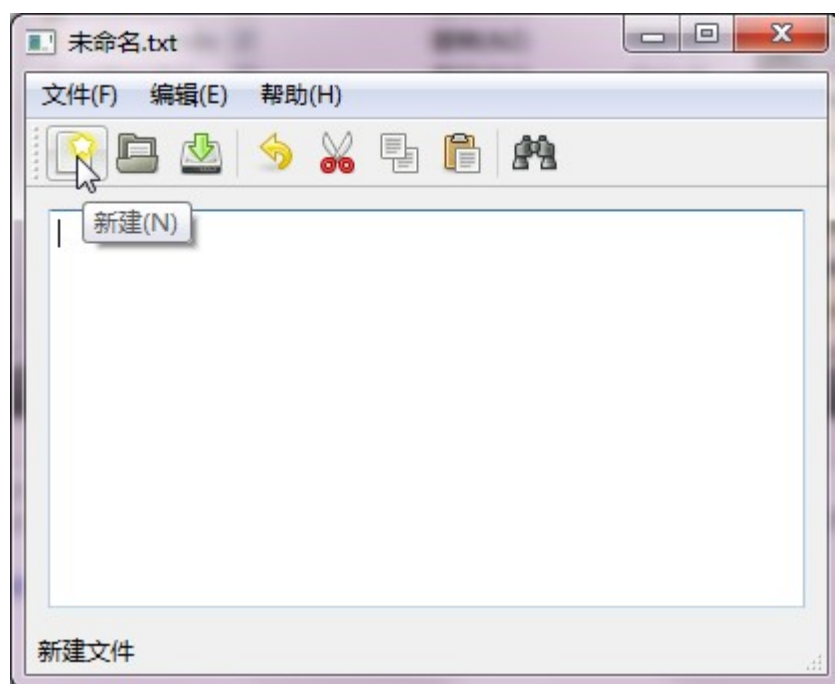


## 八、添加动作状态提示

1. 首先还是打开上一篇完成的程序。对于菜单动作添加状态提示，可以很容易的在设计器中来完成。
2. 下面进入设计模式，在Action编辑器中选中新建动作，然后在右面的属性编辑器中将其 `statusTip` 更改为“新建文件”。如下图所示。



3. 这时运行程序，当光标移动到新建动作上时，在下面的状态栏将会出现设置的提示。如下图所示。



我们可以按照这种方式来设置其他动作的状态栏提示信息。

## 九、显示其他临时信息

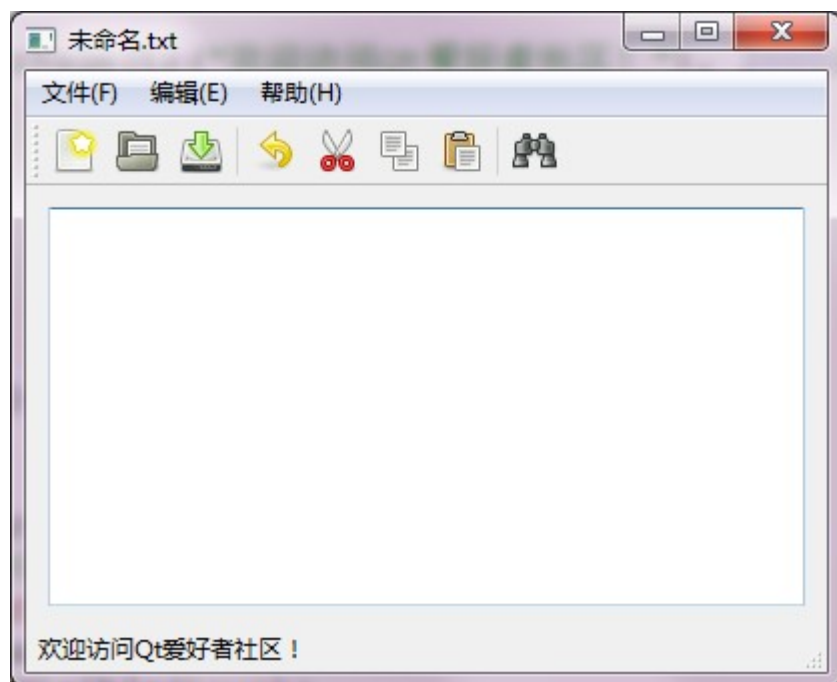
状态信息可以被分为三类：临时信息，如一般的提示信息，上面讲到的动作提示就是临时信息；正常信息，如显示页数和行号；永久信息，如显示版本号或者日期。可以使用 `showMessage()` 函数来显示一个临时消息，它会出现在状态栏的最左边。一般用 `addWidget()` 函数添加一个 `QLabel` 到状态栏上用于显示正常信息，它会生成到状态栏的最左边，可能会被临时消息所掩盖。

1. 我们到 `mainwindow.cpp` 文件的构造函数最后面添加如下一行代码：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区！"));
```

这样就可以在运行程序时显示指定的状态提示了。效果如下图所示。





这个提示还可以设置显示的时间。如：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区!"), 2000);
```

这样提示显示2000毫秒即2秒后会自动消失。

2. 下面我们在状态栏添加一个标签部件用来显示一般的提示信息。因为无法在设计模式向状态栏添加部件，所以只能使用代码来实现。先在 `mainwindow.h` 文件中添加类的前置声明：

```
class QLabel;
```

然后添加一个私有对象定义：

```
QLabel *statusLabel;
```

下面到 `mainwindow.cpp` 文件中，先添加头文件声明：

```
#include <QLabel>
```

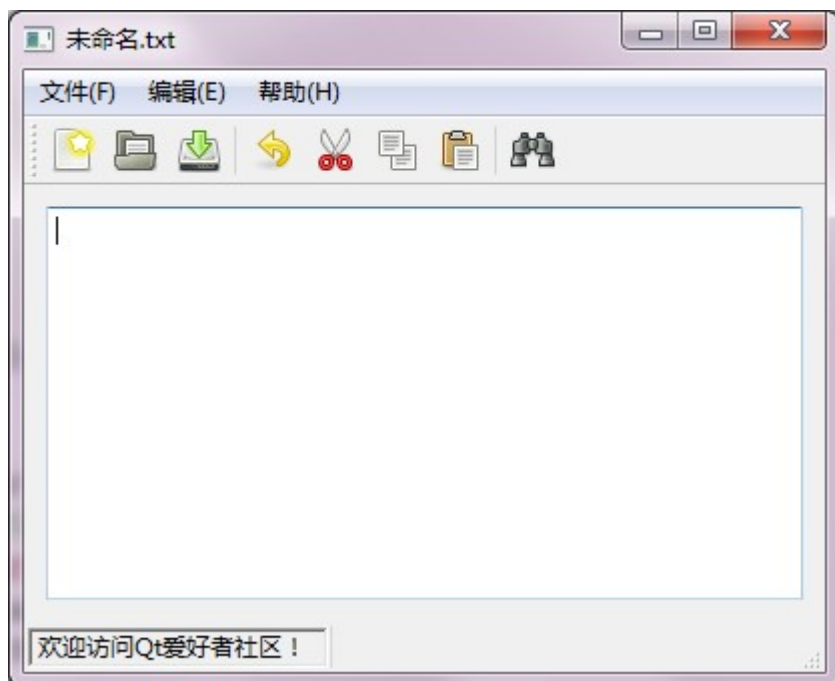
然后到构造函数中将前面添加的：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区!"), 2000);
```

一行代码注释掉，再添加如下代码：

```
statusLabel = new QLabel;  
statusLabel->setMinimumSize(150, 20); // 设置标签最小大小  
statusLabel->setFrameShape(QFrame::WinPanel); // 设置标签形状  
statusLabel->setFrameShadow(QFrame::Sunken); // 设置标签阴影  
ui->statusBar->addWidget(statusLabel);  
statusLabel->setText(tr("欢迎访问Qt爱好者社区!"));
```

这时运行程序，效果如下图所示。



下面就可以在需要显示状态的时候，调用 `statusLabel` 来设置文本了。

## 十、显示永久信息

如果要显示永久信息，要使用 `addPermanentWidget()` 函数来添加一个如 `QLabel` 一样的可以显示信息的部件，它会生成在状态栏的最右端，不会被临时消息所掩盖。

我们在构造函数中添加如下代码：

```
QLabel *permanent = new QLabel(this);  
permanent->setFrameStyle(QFrame::Box | QFrame::Sunken);  
permanent->setText(
```

```
tr("<a href=\"http://www.yafeilinux.com\">yafeilinux.com</a>"));
permanent->setTextFormat(Qt::RichText);
permanent->setOpenExternalLinks(true);
ui->statusBar->addPermanentWidget(permanent);
```

这样就在状态栏的右侧添加了一个网站的超链接，点击该链接就会自动在浏览器中打开网站。运行程序，效果如下图所示。

