Qt程序设计方法

主框架与事件系统

School of Computer Science
and Engineering

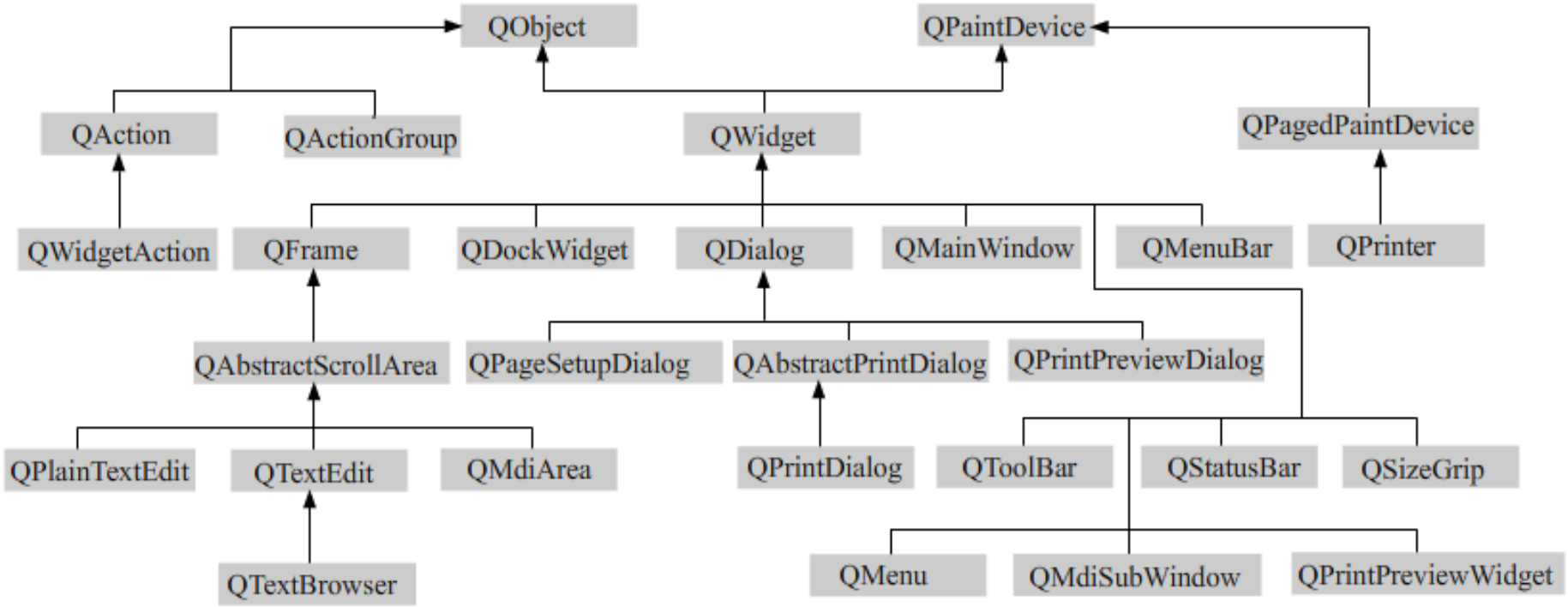
2024-2025-暑期课程

应用程序主窗口

应用程序主窗口

这一章开始接触应用程序主窗口的相关内容。对于日常见到的应用程序而言，它们中的许多都是基于主窗口的，主窗口中包含了菜单栏、工具栏、状态栏和中心区域等。本章会详细介绍主窗口的每一个部分，还会涉及到资源管理、富文本处理、拖放操作和文档打印等相关内容。

在Qt中提供了以QMainWindow类为核心的主窗口框架，它包含了众多相关的类：



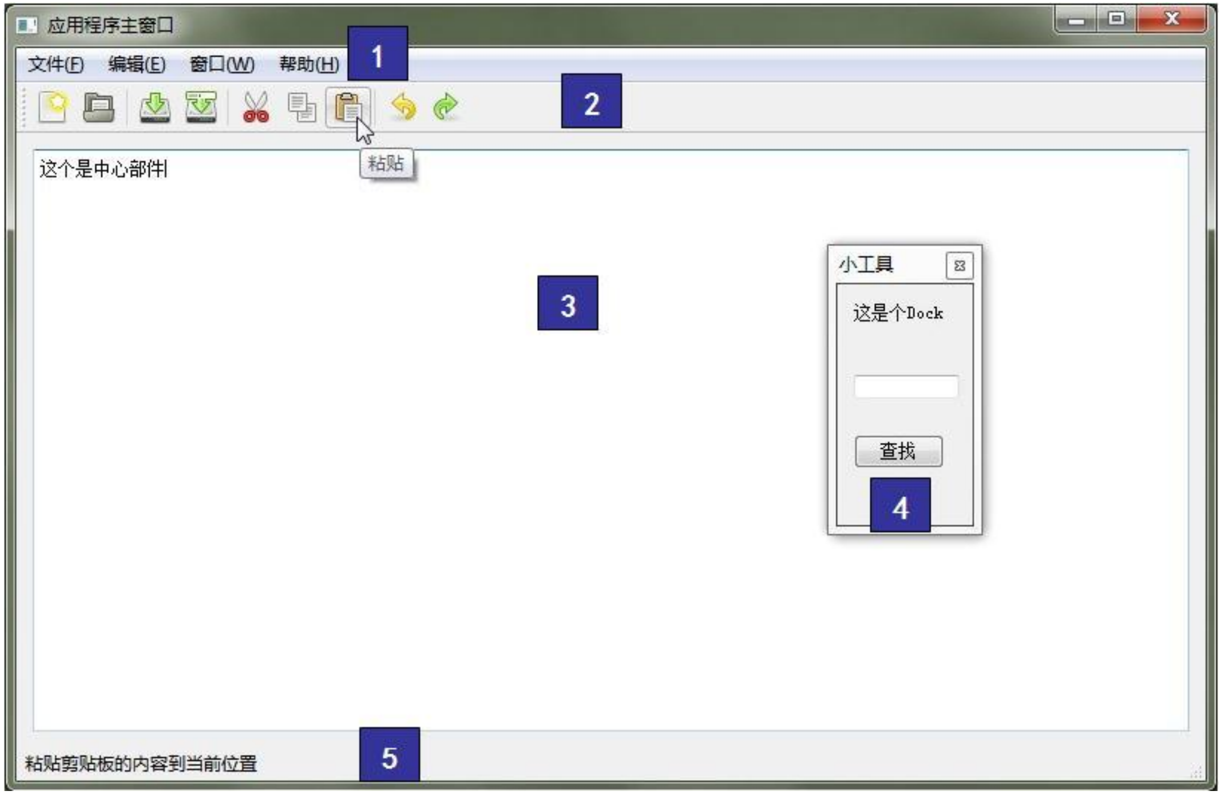
主 要 内 容

- 主窗口框架
- 富文本处理
- 拖放操作
- 打印文档
- 小结

主窗口框架

- 菜单栏和工具栏
- 使用资源系统
- 中心部件
- Dock部件
- 状态栏
- 自定义菜单

主窗口为建立应用程序用户界面提供了一个框架，Qt提供了QMainWindow和与其相关的一些类来进行主窗口的管理。QMainWindow类拥有自己的布局：

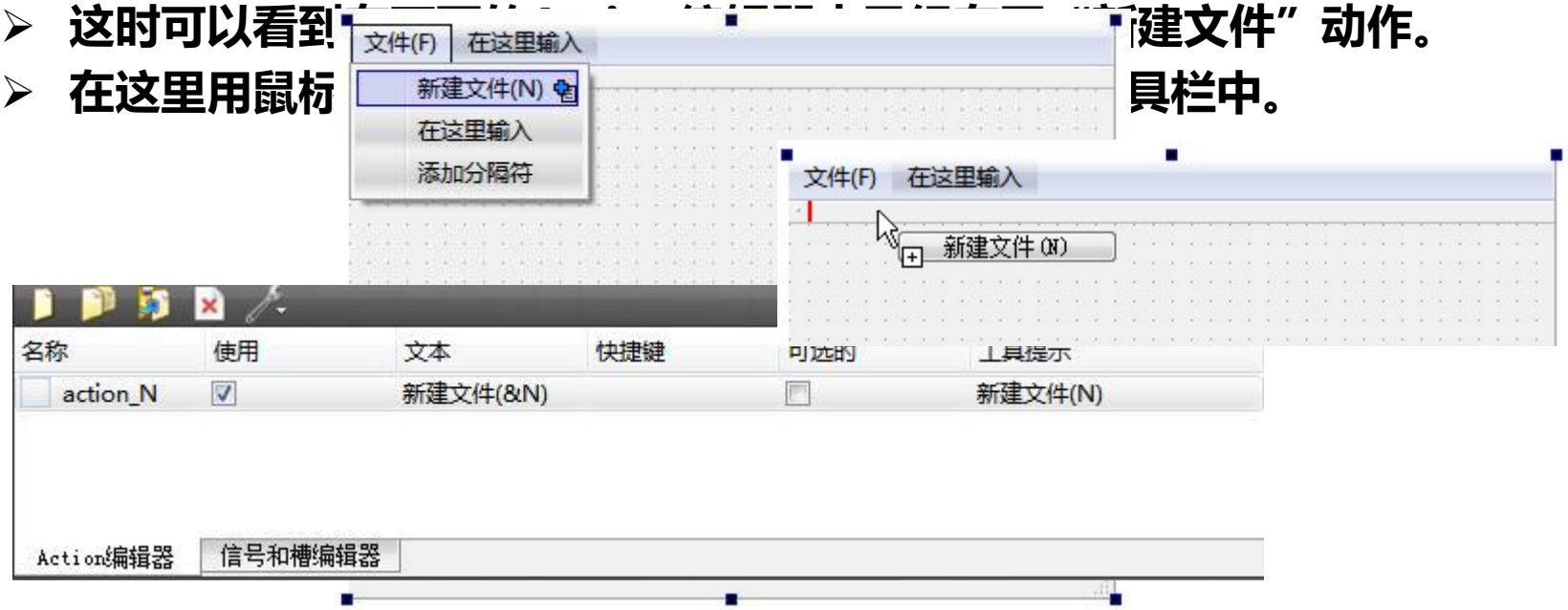


④Dock部件 [illegible]

在设计器中给菜单栏、工具栏添加动作

在Qt Creator中新建Qt Widgets应用，类名默认为MainWindow，基类默认为QMainWindow不做改动。建立好项目后，在文件列表中双击mainwindow.ui文件进入设计模式。

- 添加菜单，双击左上角的“在这里输入”，修改为“文件(&F)”，这里要使用英文半角的括号，“&F”被称为加速键，表明程序运行时，可以按下Alt+F键来激活该菜单。修改完成后，按下回车键，并在弹出的下拉菜单中，将第一项改为“新建文件(&N)”并按下回车键。
- 这时可以看到[illegible]建文件”动作。
- 在这里用鼠标[illegible]具栏中。



菜单栏

- QMenuBar类提供了一个水平的菜单栏，在QMainWindow中可以直接获取它的默认存在的菜单栏，向其中添加QMenu类型的菜单对象，然后向弹出菜单中添加QAction类型的动作对象。
- 在QMenu中还提供了间隔器，可以在设计器中向添加菜单那样直接添加间隔器，或者在代码中使用addSeparator()函数来添加，它是一条水平线，可以将菜单分成几组，使得布局很整齐。
- 在应用程序中很多普通的命令都是通过菜单来实现的，而我们也希望能将这些菜单命令放到工具栏中，以方便使用。QAction就是这样一种命令动作，它可以同时放在菜单和工具栏中。一个QAction动作包含了一个图标，一个菜单显示文本，一个快捷键，一个状态栏显示文本，一个“What’s This?”显示文本和一个工具提示文本。这些都可以在构建QAction类对象时在构造函数中指定。
- 另外还可以设置QAction的checkable属性，如果指定这个动作的checkable为true，那么当选中这个菜单时就会在它的前面显示“√”之类的表示选中状态的符号，如果该菜单有图标，那么就会用线框将图标围住，用来表示该动作被选中了。

代码方式添加菜单

```
// 添加编辑菜单
QMenu *editMenu = ui->menuBar->addMenu(tr("编辑(&E)"));

// 添加打开菜单
 QAction *action_Open = editMenu->addAction(
QIcon( “../images/open.png"),tr("打开文件(&O)"));

// 设置快捷键
action_Open->setShortcut(QKeySequence("Ctrl+O"));

// 在工具栏中添加动作
ui->mainToolBar->addAction(action_Open);
```

工具栏

- 工具栏QToolBar类提供了一个包含了一组控件的可以移动的面板。在上面已经看到可以将QAction对象添加到工具栏中，它默认只是显示一个动作的图标，这个可以在QToolBar的属性栏中进行更改。

- 在设计器中可以查看QToolBar的属性栏，其中toolButtonStyle属性设置图标和相应文本的显示及其相对位置等；movabel属性设置状态栏是否可以移动；allowedArea设置允许停靠的位置；iconsize属性设置图标的大小；floatable属性设置是否可以悬浮。

在工具栏中添加部件

```
QToolButton *toolBtn = new QToolButton(this);       // 创建QToolButton
toolBtn->setText(tr("颜色"));
QMenu *colorMenu = new QMenu(this);                 // 创建一个菜单
colorMenu->addAction(tr("红色"));
colorMenu->addAction(tr("绿色"));
toolBtn->setMenu(colorMenu);                        // 添加菜单
toolBtn->setPopupMode(QToolButton::MenuButtonPopup); // 设置弹出
    模式
ui->mainToolBar->addWidget(toolBtn);                // 向工具栏添加
    QToolButton按钮
QSpinBox *spinBox = new QSpinBox(this);             // 创建QSpinBox
ui->mainToolBar->addWidget(spinBox);                // 向工具栏添加QSpinBox
    部件
```

资源系统

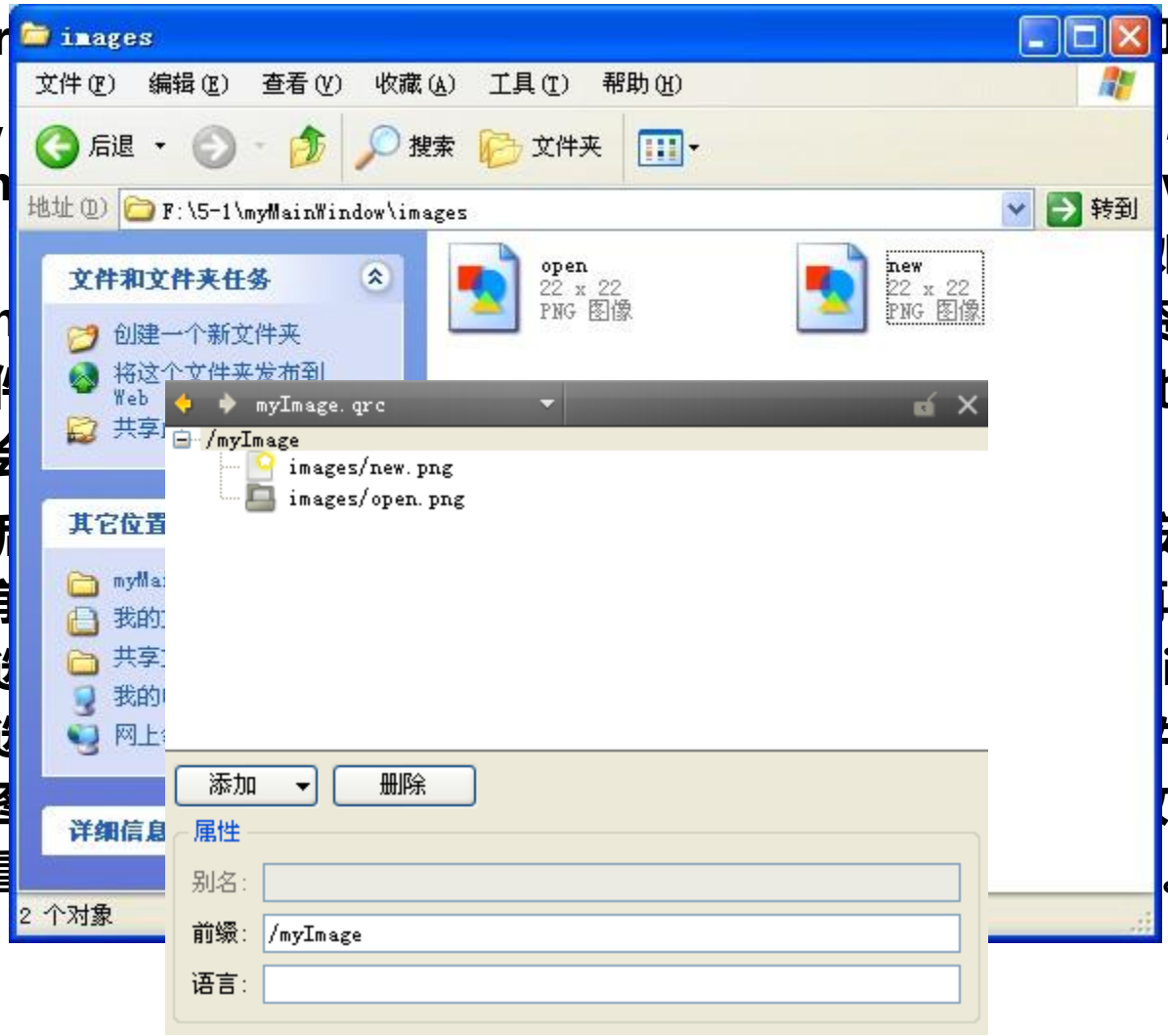
Qt资源系统是一个独立于平台的（跟使用的操作系统无关）用于在可执行文件中存储二进制文件的机制。

- 可以让需要的文件（图片、文本等）包含到程序编程生成的可执行文件（例如exe文件）中。

- 保证了程序中使用的文件不会丢失、不会因为存放路径而导致程序运行错误。

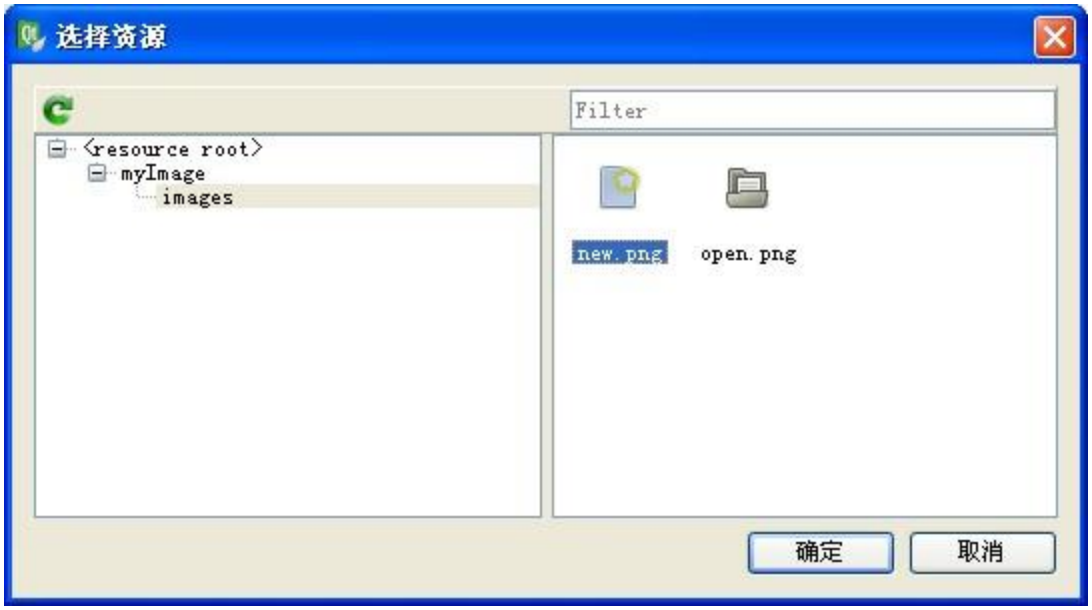
在Qt Creator中添加资源

- 第一步，添加Qt资源文件。往项目中添加新文件，选择Qt分类中的Qt Resour…可。
- 第二步，…，就是新建的myIm…w中新建一个名为…如这里放入了一个“r…资源系统要求资源文件…地方，当添加资源时会…
- 然后…按钮，选择“添加前…再点击“添加”按钮，选…images文件夹中，选…件中就出现了添加的图…文（注意：这一点很重…。



- 第三步，使用图片。在设计模式Action编辑器中双击“新建文件”动作，这时会弹出编辑动作对话框。在其中将对象名称改为“action_New”，工具提示改为“新建文件”，然后按下图标后面的按钮，进入选择资源界面。

- 第一次进入该界面还没有显示可用的资源，需要按下左上角的重新加载绿色箭头图标，这时图片资源就显示出来了。这里选择new.png图片，然后按下确定按钮。



- 如果在编写代码时使用new.png图片，那么就可以将其路径指定为“:/myImage/images/new.png”，前缀“/myImage”是添加资源时手动设置的。

资源文件介绍

- **在使用资源时添加的qrc资源文件其实是一个XML格式的文本文件，进入编辑模式，在myImage.qrc文件上点击鼠标右键，选择“用…打开”→“Plain Text Editor”，这时就会看到myImage.qrc的内容如下：**

```
<RCC>
    <qresource prefix="/myImage">
        <file>images/new.png</file>
        <file>images/open.png</file>
    </qresource>
</RCC>
```

在这里指明了文件类型为RCC，表明是Qt资源文件。然后是资源前缀，在下面罗列了添加的图片的路径。

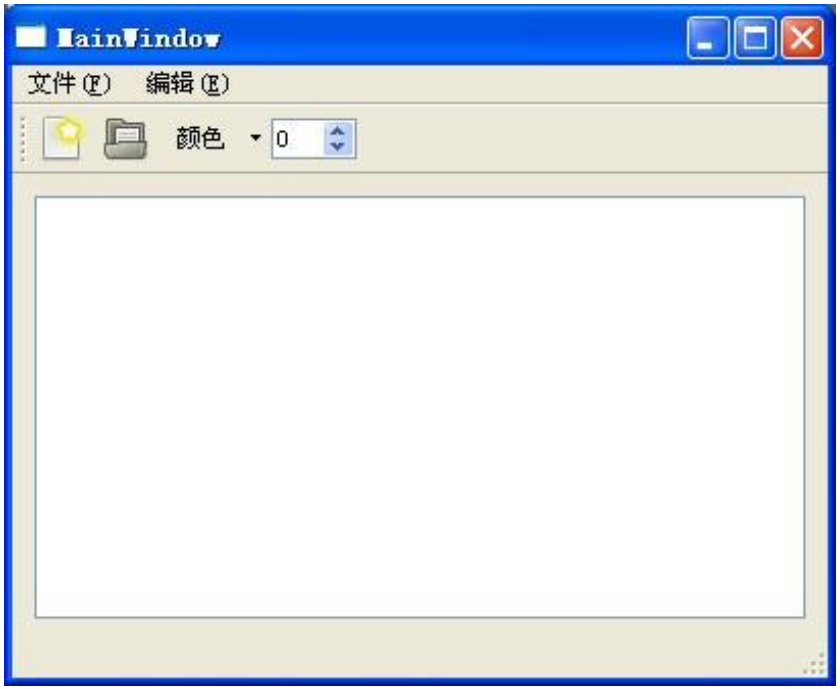
- **当往项目中添加了一个资源文件时，会自动往工程文件myMainWindow.pro中添加代码：**

```
RESOURCES += \
             myImage.qrc
```

这表明项目中使用了资源文件myImage.qrc。

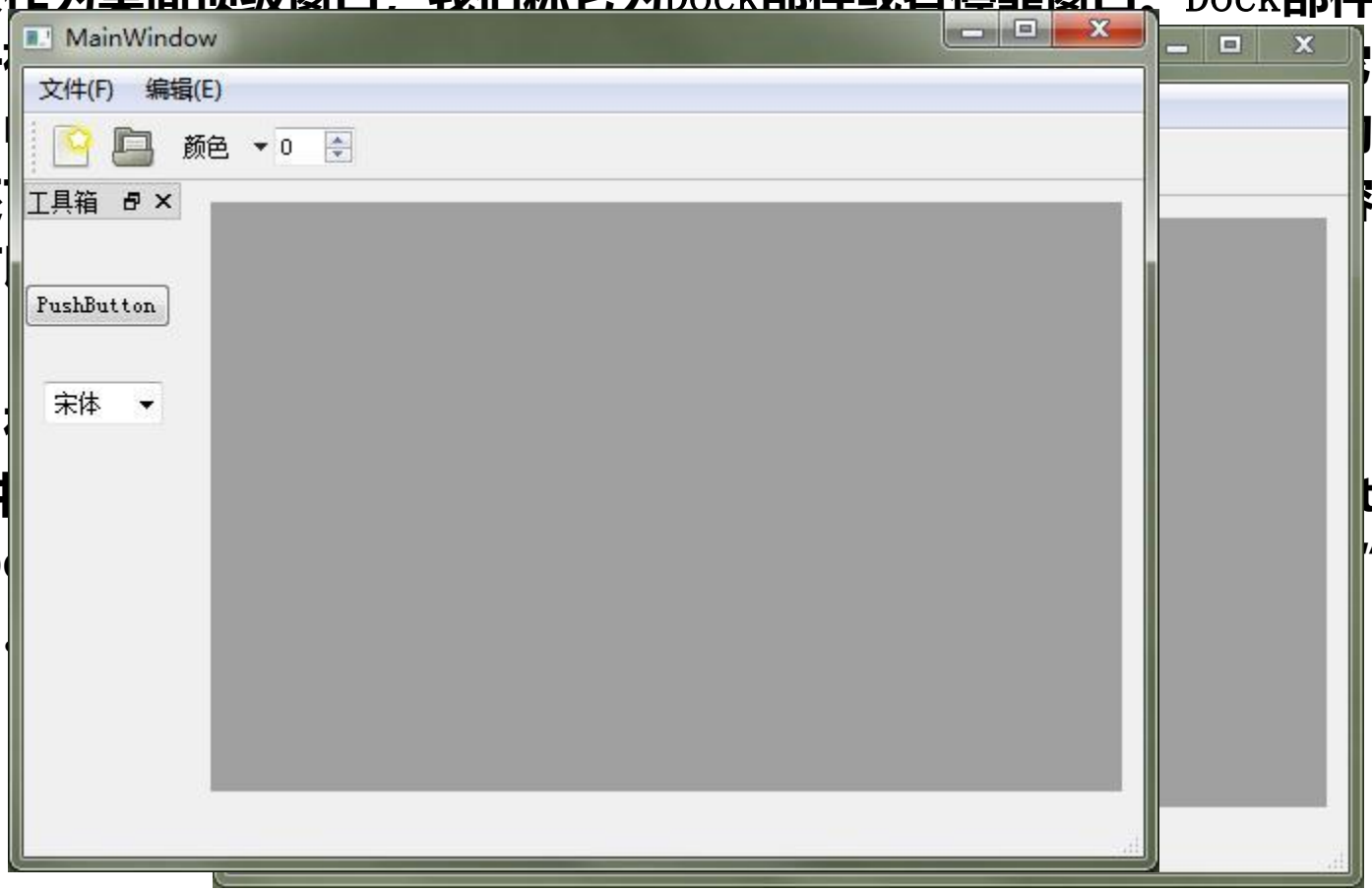
中心部件

- 在主窗口的中心区域可以放置一个中心部件，它一般是一个编辑器或者浏览器。这里支持单文档部件，也支持多文档部件。一般的，会在这里放置一个部件，然后使用布局管理器使其充满整个中心区域，并可以随着窗口的大小变化而改变大小。例如在设计模式中，往中心区域拖入一个Text Edit，然后点击界面，按下Ctrl+G使其处于一个栅格布局中。 效果如下。



Dock部件

- QDockWidget类提供了这样一个部件，它可以停靠在QMainWindow中也可以悬浮起来作为桌面顶级窗口，我们称它为Dock部件或者停靠窗口。Dock部件一般用于在主窗口的地方，还区域，可



- 例如：Dock中 Combo "工具箱"

状态栏

- QStatusBar类提供了一个水平条，用来显示状态信息。QMainWindow中默认提供了一个状态栏。

- 状态信息可以被分为三类：临时信息，如一般的提示信息；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。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。

- 在状态栏的最右端，还有一个QSizeGrip部件，用来调整窗口的大小，可以使用setSizeGripEnabled()函数来禁用它。

目前的设计器中还不支持直接向状态栏中拖放部件，所以需要使用代码来生成。
例如：

```
// 显示临时消息，显示2000毫秒即2秒钟
ui->statusBar->showMessage(tr("欢迎使用多文档编辑器"), 2000);
// 创建标签，设置标签样式并显示信息，将其以永久部件的形式添加到状态栏
QLabel *permanent = new QLabel(this);
permanent->setFrameStyle(QFrame::Box | QFrame::Sunken);
permanent->set
ui->statusBar->
```

此时运行程序，字符串在显示一会儿后就自动消失了，示在状态栏最右端。

自定义菜单

Qt中的QWidgetAction类可以实现自定义菜单的功能。为了实现自定义菜单，需要新建一个类，它继承自QWidgetAction类，并且在其中重新实现createWidget()函数。

自学内容：写一个程序，实现了这样一个菜单：它包含一个标签和一个行编辑器，可以在行编辑器中输入字符串，然后按下回车键，就可以自动将字符串输入到中心部件文本编辑器中。

5.2 富文本处理

富文本（Rich Text）或者叫做富文本格式，简单来说就是在文档中可以使用多种格式，比如字体颜色、图片和表格等等。它是与纯文本（Plain Text）相对而言的，比如Windows上的记事本就是纯文本编辑器，而Word就是富文本编辑器。

富文本文档结构

文本块

表格、列表与图片

查找功能

语法高亮与HTML

富文本文档结构

在Qt中提供了对富文本处理的支持。Qt中对富文本的处理分为了编辑操作和只读操作两种方式。

- 编辑操作是使用基于光标的一些接口函数，这样更好的模拟了用户的编辑操作，更加容易理解，而且不会丢失底层的文档框架；
- 而对于文档结构的概览，使用了只读的分层次的接口函数，它们有利于文档的检索和输出。

对于文档的读取和编辑要使用不同方面的两组接口。

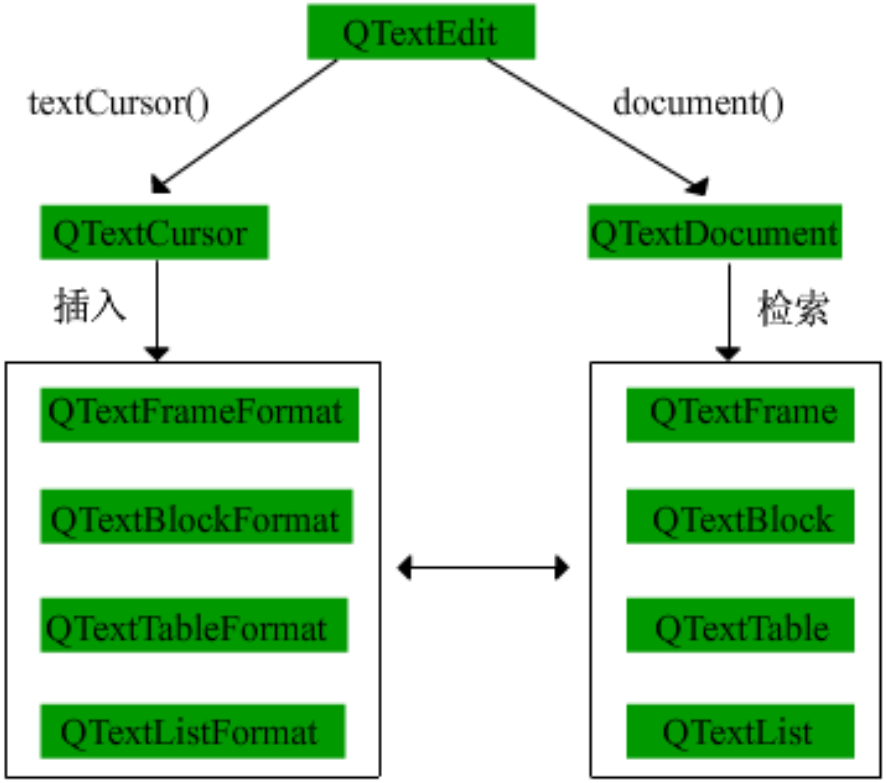
- 文档的光标主要基于QTextCursor类
- 文档的框架主要基于QTextDocument类。

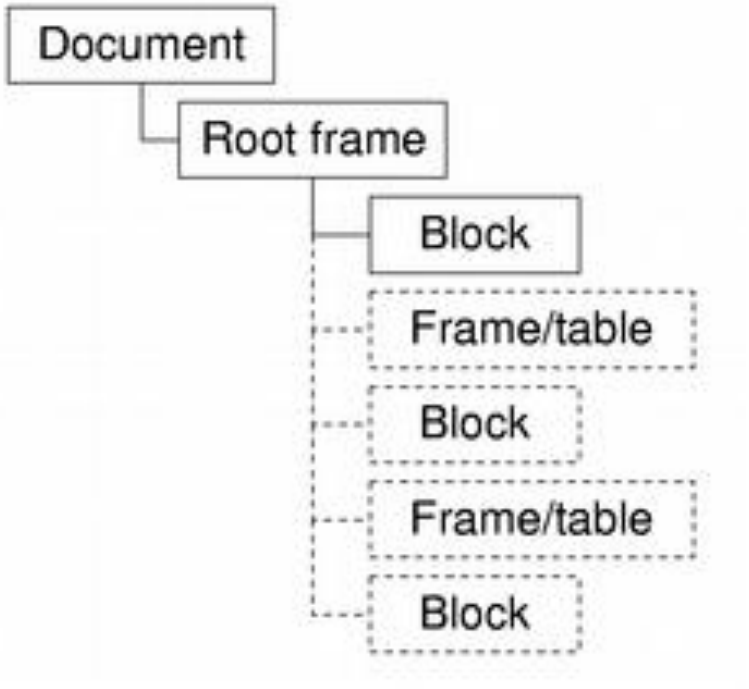
一个富文本文档的结构被分为了几种元素来表示，分别是框架（QTextFrame）、文本块（QTextBlock）、表格（QTextTable）和列表（QTextList）。

每种元素的格式又使用相应的format类来表示，它们分别是框架格式（QTextFrameFormat）、文本块格式（QTextBlockFormat）、表格格式（QTextTableFormat）和列表格式（QTextListFormat），这些格式一般在编辑文档时使用，所以它们常和QTextCursor类配合使用。

因为QTextEdit类就是一个富文本编辑器，所以在构建QTextEdit类的对象时就已经构建了一个QTextDocument类对象和一个QTextCursor类对象，只需调用它们进行相应的操作即可。

一个空的文档包含了一个根框架（Root frame），这个根框架又包含了一个空的文本块（Block）。框架将一个文档分为多个部分，在根框架里可以再添加文本块、子框架和表格等。





设置根框架

```
QTextDocument *document = ui->textEdit->document(); //获取文档对象

QTextFrame *rootFrame = document->rootFrame();      // 获取根框架

QTextFrameFormat format;                            // 创建框架格式

format.setBorderBrush(Qt::red);                     // 边界颜色

format.setBorder(3);                                // 边界宽度

rootFrame->setFrameFormat(format);                  // 框架使用格式
```

添加子框架

```
QTextFrameFormat frameFormat;

frameFormat.setBackground(Qt::lightGray);              // 设置背景颜色

frameFormat.setMargin(10);                             // 设置边距

frameFormat.setPadding(5);                             // 设置填衬

frameFormat.setBorder(2);

//设置边框样式
frameFormat.setBorderStyle(QTextFrameFormat::BorderStyle_Dotted);

QTextCursor cursor = ui->textEdit->textCursor();       // 获取光标

cursor.insertFrame(frameFormat);                       // 在光标处插入框架
```

文本块

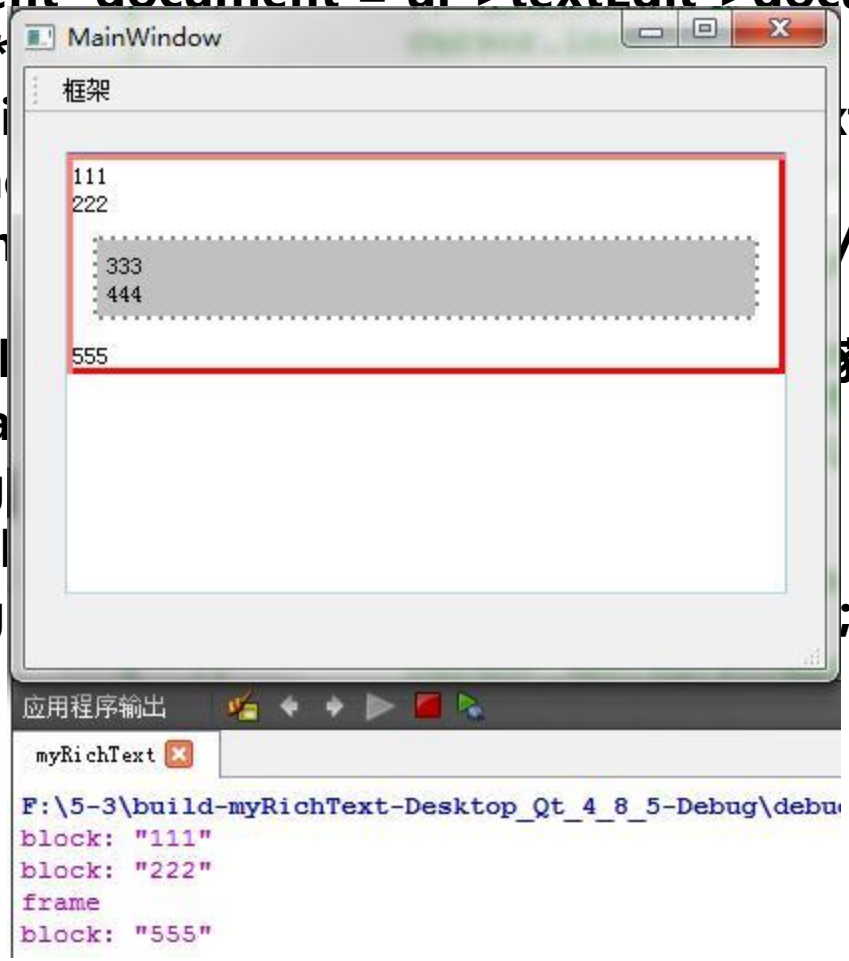
文本块QTextBlock类为文本文档QTextDocument提供了一个文本片段（QTextFragment）的容器。

一个文本块可以看做是一个段落，但是它不能使用回车换行，因为一个回车换行就表示创建一个新的文本块。QTextBlock提供了只读接口，它是前面提到的文档分层次的接口的一部分，如果QTextFrame看做是一层，那么其中的QTextBlock就是另一层。

文本块的格式由QTextBlockFormat类来处理，它主要涉及对齐方式，文本块四周的边白，缩进等内容。而文本块中的文本内容的格式，比如字体大小、加粗、下划线等内容，则由QTextCharFormat类来设置。

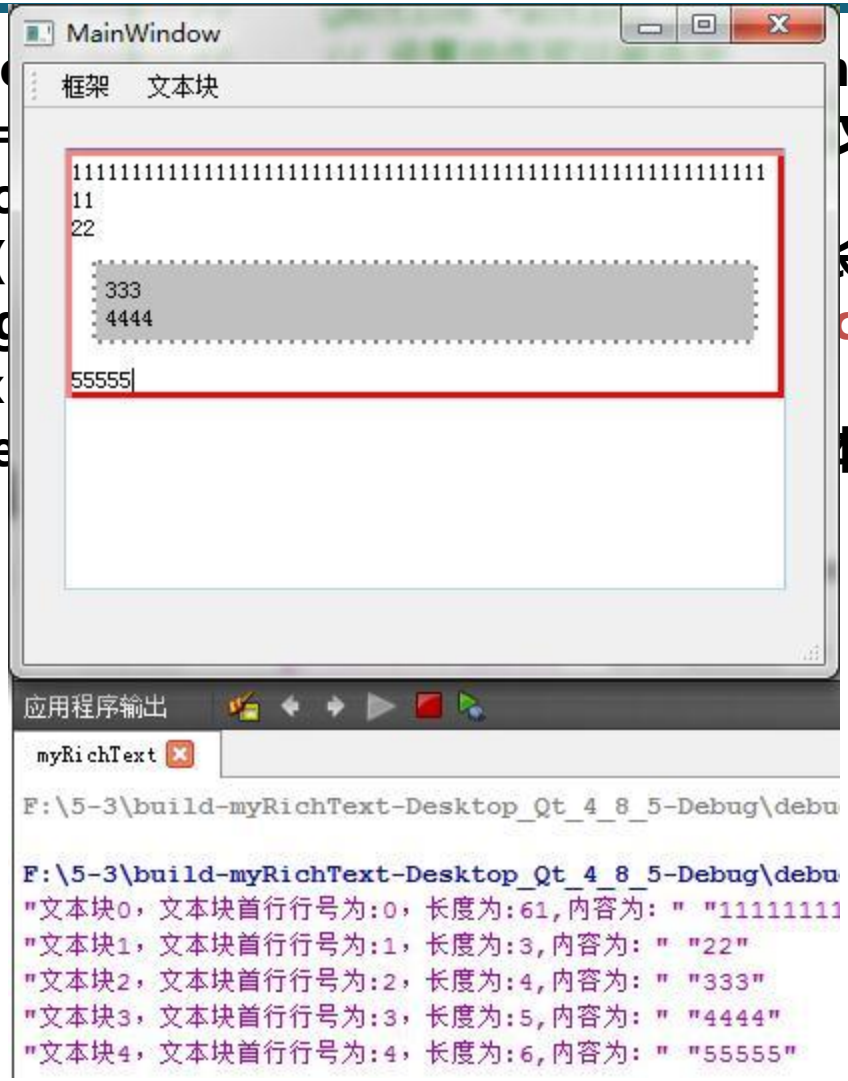
遍历框架

```
QTextDocument *document = ui->textEdit->document();
 QTextFrame *
 QTextFrame::i                                          xtFrame类的迭代器
 for (it = fram
     QTextFram                                          // 获取当前框架的指
 针
     QTextBlocl                                         获取当前文本块
     if (childFra
         qDebug
     else if (chil
         qDebug                                         ;
}
```



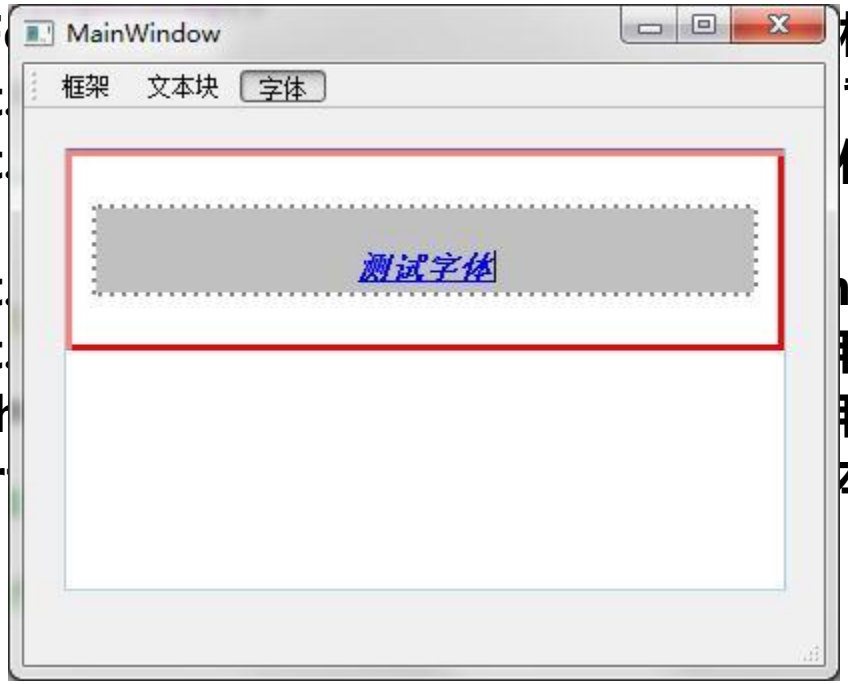
遍历子框架

```
QTextDocument *dc                                    nt();
 QTextBlock block =                                  文档的第一个文本块
 for (int i = 0; i < dc
    qDebug() << tr(                                  长度为:%3,内容为：")
             .arg(i).arg                             ck.length())
             << block
    block = block.ne                                 本块
 }
```



编辑文本块及其内容的格式

```
QTextCursor cursor = ui->textEdit->textCursor();
 QTextBlockFormat blockFormat;              // 文本块格式
 blockFormat.setAlignment(Qt::AlignCenter); // 水平居中
 cursor.insertBlock(blockFormat);           // 使用文本块格式
 QTextCharF                                                  格式
 charFormat.                                                 背景色
 charFormat.                                                 体颜色
 // 使用宋体，
 charFormat.                                                 t::Bold, true));
 charFormat.                                                 用下划线
 cursor.setCh                                                用字符格式
 cursor.inser                                                本
```



表格、列表和图片

- 插入表格

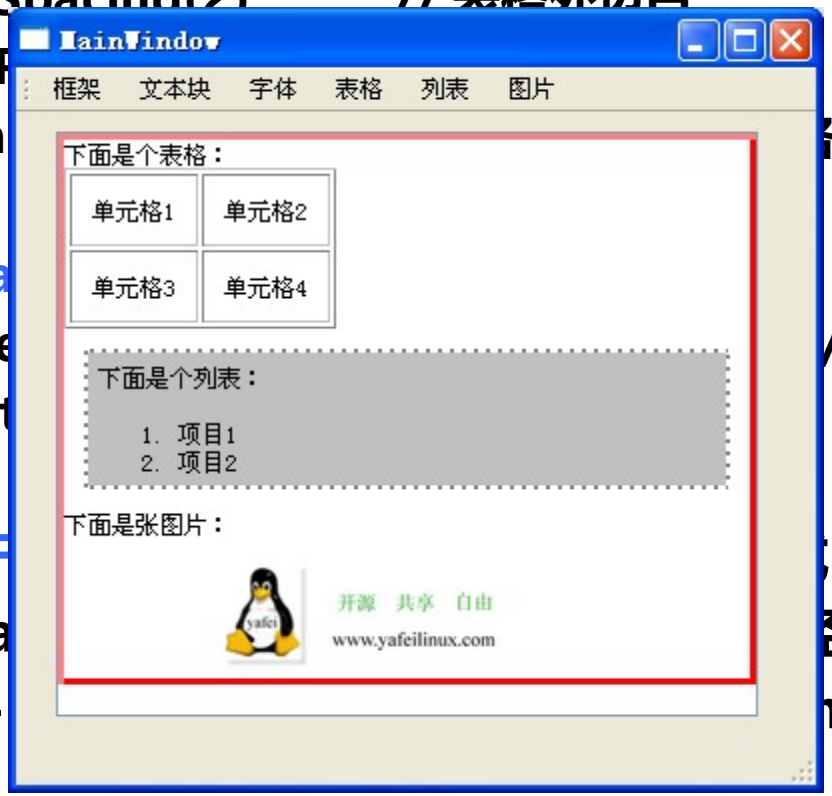
```
QTextCursor cursor = ui->textEdit->textCursor();
QTextTableFormat format;          // 表格格式
format.setCellSpacing(2);         // 表格外边白
format.setCellP
cursor.insertTa
```

- 插入列表

```
QTextListForma
format.setStyle                                  // 数字编号
ui->textEdit->t
```

- 插入图片

```
QTextImageF                                      ;
format.setNa                                     径
ui->textEdit-                                    mat);
```



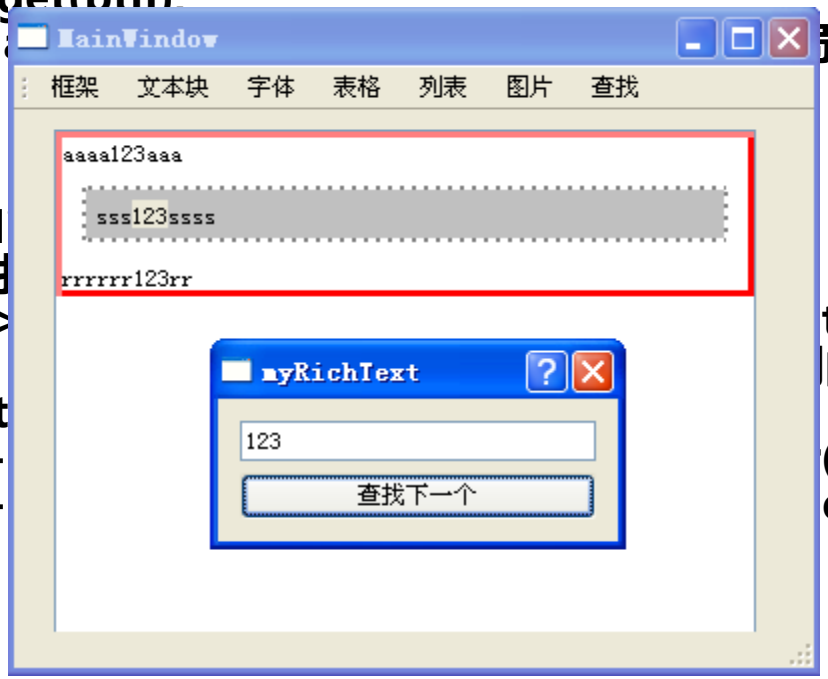
查找功能

- 查找文本

```
QDialog *dlg = new QDialog(this);          // 创建对话框
lineEdit = new QLineEdit(dlg);             // 创建行编辑器
QPushButton *btn = new QPushButton(dlg);  // 创建按钮
btn->setText(tr("查找下一个"));
connect(btn,SIGNAL(clicked()),this,SLOT(findNext())); // 关联信号和槽
QVBoxLayout *layout = new QVBoxLayout;    // 创建垂直布局管理器
layout->addWidget(lineEdit);              // 添加部件
layout->addWidget(btn);
dlg->setLayout(l                                          管理器
dlg->show();
```

- 查找下一个

```
QString string = l
// 使用查找函数查找
bool isfind = ui->                                        t::FindBackward);
if(isfind){                                               的编号
    qDebug() << t
             .arg(ui-                                     ())
             .arg(ui-                                     er());
}
```



语法高亮

在使用Qt Creator编辑代码时可以发现，输入关键字时会显示不同的颜色，这就是所谓的语法高亮。

在Qt的富文本处理中提供了QSyntaxHighlighter类来实现语法高亮。为了实现这个功能，需要创建QSyntaxHighlighter类的子类，然后重新实现highlightBlock()函数，使用时直接将QTextDocument类对象指针作为其父部件指针，这样就可以自动调用highlightBlock()函数了。

例如，自定义的类为MySyntaxHighlighter，像这样来使用：

```
highlighter = new MySyntaxHighlighter(ui->textEdit->document());
```

这里创建了MySyntaxHighlighter类的对象，并且使用编辑器的文档对象指针作为其参数，这样，每当编辑器中的文本改变时都会调用highlightBlock()函数来设置语法高亮。

重新实现highlightBlock()函数：

```
QTextCharFormat myFormat;            // 字符格式
 myFormat.setFontWeight(QFont::Bold);
 myFormat.setForeground(Qt::green);
 QString pattern = "\\bchar\\b";      // 要匹配的字符，这里是“char”单词
 QRegExp expression(pattern);         // 创建正则表达式
 int index = text.indexOf(expression); // 从位置0开始匹配字符串
 // 如果匹配成功，那么返回值为字符串的起始位置，它大于或等于0
 while (index >= 0) {
    int length = expression.matchedLength(); // 要匹配字符串的长度
    setFormat(index, length, myFormat);     // 对要匹配的字符串设置格式
    index = text.indexOf(expression, index + length); // 继续匹配
 }
```

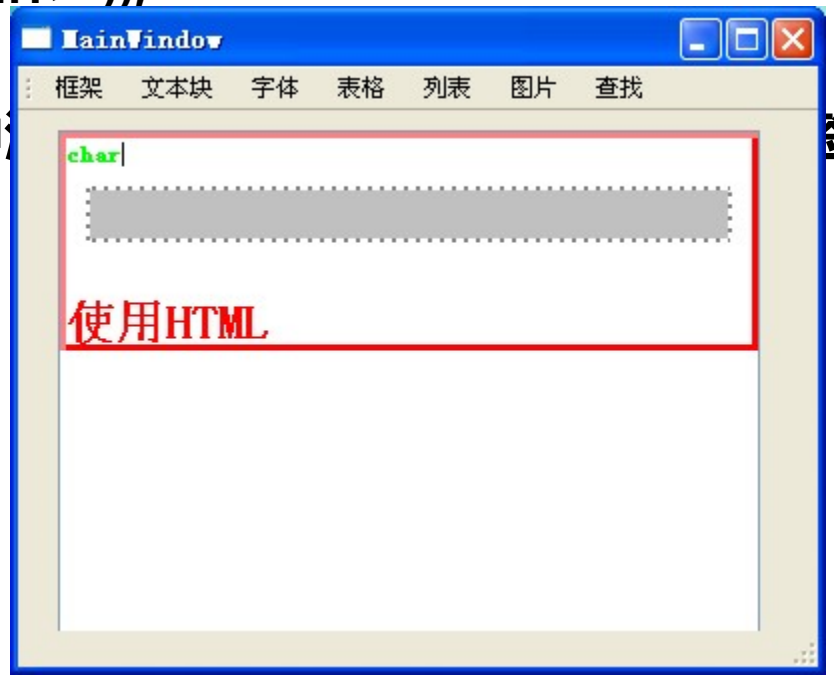
在这里主要是使用了正则表达式和QString类的indexOf()函数来进行字符串的匹配，如果匹配成功，则使用QSyntaxHighlighter类的setFormat()函数来设置字符格式。

HTML

在富文本处理中还提供了对HTML子集的支持，可以在QLabel或者QTextEdit添加文本时使用HTML标签或者CSS属性，例如：

```
ui->textEdit->append(tr("<h1><font color=red>使用
HTML</font></h1>"));
```

这里往编辑器中



签，

5.3 拖放操作

对于一个实用的应用程序，我们不仅希望能从文件菜单中打开一个文件，更希望可以通过拖动，直接将桌面上的文件拖入程序界面上来打开，就像可以将.pro文件拖入Qt Creator中来打开整个项目一样。Qt中提供了强大的拖放机制，拖放操作分为拖动（Drag）和放下（Drop）两种操作。当数据被拖动时会被存储为MIME (Multipurpose Internet Mail Extensions)类型，在Qt中使用QMimeData类来表示MIME类型的数据，并使用QDrag类来完成数据的转移，而整个拖放操作都是在几个鼠标事件和拖放事件中完成的。

使用拖放打开文件

自定义拖放操作

使用拖放打开文件

例如：将桌面上的txt文本文件拖入自己编写的程序中来打开。使用拖放需要声明两个函数：

```
protected:
    void dragEnterEvent(QDragEnterEvent *event); // 拖动进入事件
    void dropEvent(QDropEvent *event);           // 放下事件
```

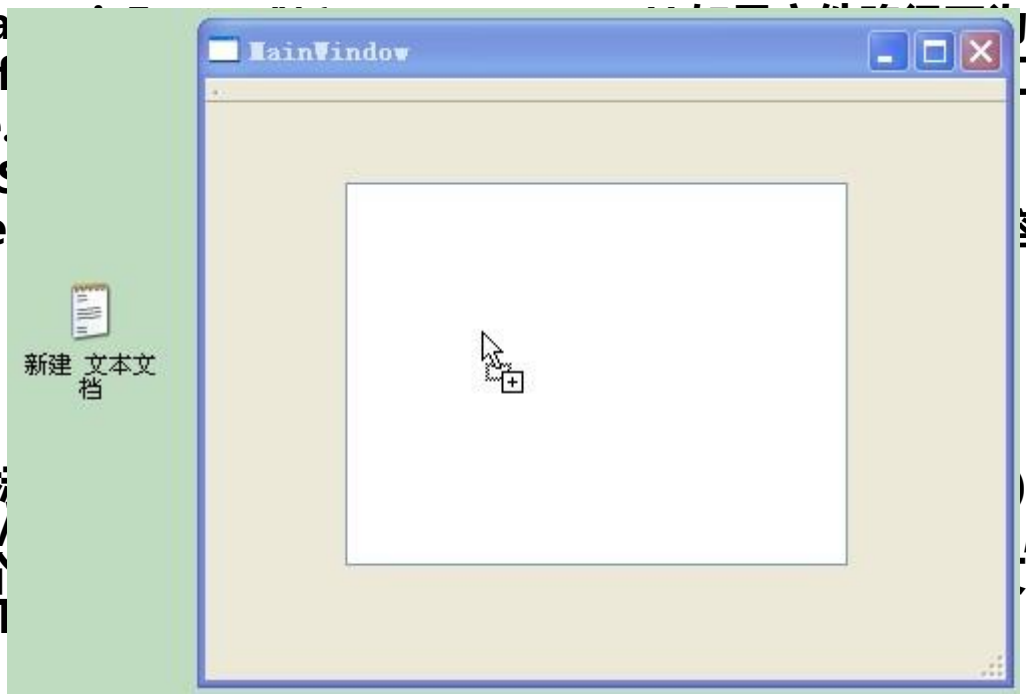
拖入操作

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event) // 进入事
    件
{
   if(event->mimeData()->hasUrls())                    // 数据中是否包含URL
      event->acceptProposedAction();                   // 如果是则接收动作
   else event->ignore();                               // 否则忽略该事件
}
```

当鼠标拖拽一个数据进入主窗口时，就会触发dragEventEvent()事件处理函数，获取其中的MIME数据，然后查看它是否包含URL路径，因为拖入的文本文件实际上就是拖入了它的路径，这就是event->mimeData()->hasUrls()实现的功能。如果有这样的数据，就接收它，否则就忽略该事件。

放下操作

```
void MainWindow::dropEvent(QDropEvent *event)          // 放下事件
{
    const QMimeData *mimeData = event->mimeData();     // 获取MIME数据
    if(mimeData->hasUrls()){                           // 如果数据中包含URL
        QList<QUrl> urlList = mimeData->urls();        // 获取URL列表
        // 将其中第一个URL表示为本地文件路径
        QString fileName = urlList.at(0).toLocalFile();
        if(!fileNa                                                  )空
            QFile f                                                 ]开该文件
            if(!file.
            QTextS
            ui->te                                                  容读入编辑器
        }
    }
}
```



当松开鼠标   事件处理函数，这里获取了M   ，所以获取了列表中的第一个   件路径。然后使用QFile和QT

MIME类型数据处理函数

在QMimeData类中提供了几个函数来方便的处理常见的MIME数据：

测试函数	获取函数	设置函数	MIME类型
hasText()	text()	setText()	text/plain
hasHtml()	html()	setHtml()	text/html
hasUrls()	urls()	setUrls()	text/uri-list
hasImage()	imageData()	setImageData()	image/*
hasColor()	colorData()	setColorData()	application/x-color

自定义拖放操作

下面以在窗口中拖动图片为例，需要声明以下几个函数：

```
protected:
    void mousePressEvent(QMouseEvent *event);      // 鼠标按下事件
    void dragEnterEvent(QDragEnterEvent *event);   // 拖动进入事件
    void dragMoveEvent(QDragMoveEvent *event);     // 拖动事件
    void dropEvent(QDropEvent *event);             // 放下事件
```

- mousePressEvent： 为拖动图片做准备工作，将图片数据放到自定义的MIME类型中。
- dragEnterEvent： 开始拖动，先判断是否包含需要移动的类型。
- dragMoveEvent：拖动过程。
- dropEvent：放下图片，创建新的图片放到光标处。

当鼠标按下时会触发鼠标按下事件，进而执行其处理函数，在这里进行了一系列操作，大体上可以分为六步。

- 第一步，先获取鼠标指针所在处的部件的指针，将它强制转换为QLabel类型的指针，然后使用inherits()函数判断它是否是QLabel类型，如果不是则直接返回，不再进行下面的操作。
- 第二步，因为不仅要在拖动的数据中包含图片数据，还要包含它的位置信息，所以需要使用自定义的MIME类型。这里使用了QByteArray字节数组来存放图片数据和位置数据。然后使用QDataStream类将数据写入数组中。其中位置信息是当前鼠标指针的坐标减去图片左上角的坐标而得到的差值。
- 第三步，创建了QMimeData类对象指针，使用了自定义的MIME类型“myimage/png”，将字节数组放入QMimeData中。
- 第四步，为了移动数据，必须创建QDrag类对象，然后为其添加QMimeData数据。这里为了在移动过程中一直显示图片，需要使用setPixmap()函数为其设置图片。然后使用setHotSpot()函数指定了鼠标在图片上点击的位置，这里是相对于图片左上角的位置，如果不设定这个，那么在拖动图片过程中，指针会位于图片的左上角。
- 第五步，在移动图片过程中希望原来的图片有所改变来表明它正在被操作，所以为其添加了一层阴影。
- 第六步，执行拖动操作，这需要使用QDrag类的exec()函数。这个函数可以设定所支持的放下动作和默认的放下动作，比如这里设置了支持复制动作Qt::CopyAction和移动动作Qt::MoveAction，并设置默认的动作是复制。这就是说拖动图片，可以是移动它，也可以是进行复制，而默认的是复制操作，比如使用acceptProposedAction()函数时就是使用默认的操作。当图片被放下后exec()函数就会返回操作类型，这个返回值由下面要讲到的dropEvent()函数中的设置决定。这里判断到底进行了什么操作，如果是移动操作，那么就删除原来的图片，如果是复制操作，就恢复原来的图片。

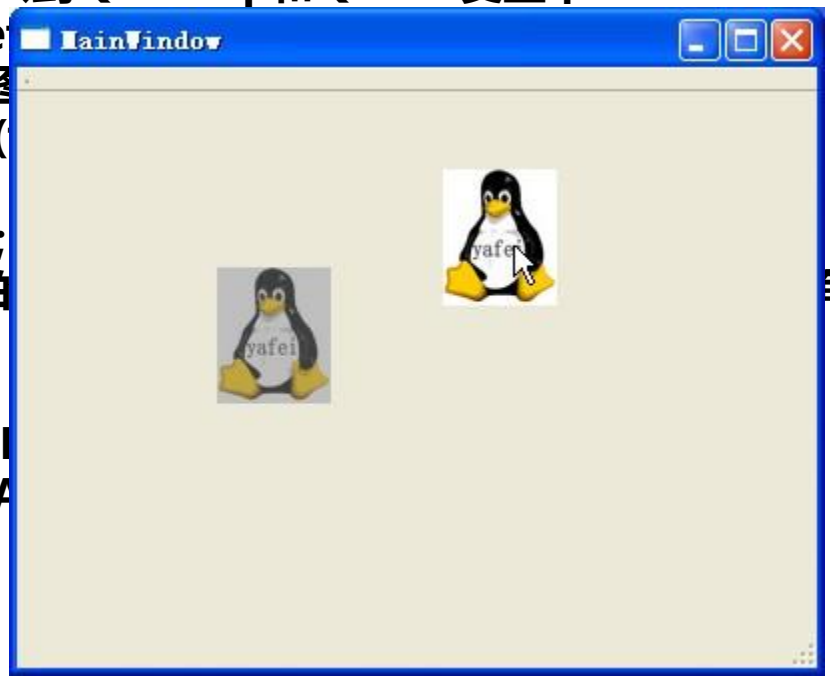
```
void MainWindow::mousePressEvent(QMouseEvent *event)   //鼠标按下事件
{
    // 第一步：获取图片
    // 将鼠标指针所在位置的部件强制转换为QLabel类型
    QLabel *child = static_cast<QLabel*>(childAt(event->pos()));
    if(!child->inherits("QLabel")) return; // 如果部件不是QLabel则直接返回
    QPixmap pixmap = *child->pixmap();     // 获取QLabel中的图片

    // 第二步：自定义MIME类型
    QByteArray itemData;                                  // 创建字节数组
    QDataStream dataStream(&itemData, QIODevice::WriteOnly); // 创建数据流
    // 将图片信息，位置信息输入到字节数组中
    dataStream << pixmap << QPoint(event->pos() - child->pos());

    // 第三步：将数据放入QMimeData中
QMimeData *mimeData = new QMimeData;  // 创建QMimeData用来存放要移动的数
      据
   // 将字节数组放入QMimeData中，这里的MIME类型是我们自己定义的
    mimeData->setData("myimage/png", itemData);
```

// 第四步：将QMimeData数据放入QDrag中

```
QDrag *drag = new QDrag(this);     // 创建QDrag，它用来移动数据
   drag->setMimeData(mimeData);
   drag->setPixmap(pixmap);//在移动过程中显示图片，若不设置则默认显示一个小
    矩形
   drag->setHotSpot(event->pos() - child->pos()); // 拖动时鼠标指针的位置不变

   // 第五步：给原图片添加阴影
   QPixmap tempPixmap = pixmap; // 使原图片添加阴影
   QPainter painter;          // 创建QPainter，用来绘制QPixmap
   painter.begin(&tempPixmap);
   // 在图片的外接矩形中添加一层透明的淡黑色形成阴影效果
   painter.fillRect(pixmap.rect(), QColor(127, 127, 127, 127));
   painter.end();
   child->setPixmap(tempPixmap); // 在移动图片过程中，让原图片添加一层黑色阴
    影
   // 第六步：执行拖放操作
   if (drag->exec(Qt::CopyAction | Qt::MoveAction, Qt::CopyAction)
         == Qt::MoveAction)// 设置拖放可以是移动和复制操作，默认是复制操作
      child->close();       // 如果是移动操作，那么拖放完成后关闭原标签
   else {
      child->show();          // 如果是复制操作，那么拖放完成后显示标签
      child->setPixmap(pixmap); // 显示原图片，不再使用阴影
   }
}
```

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event) // 拖动进入事
    件
{
    // 如果有我们定义的MIME类型数据，则进行移动操作
    if (event->mimeData()->hasFormat("myimage/png")) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::dragMoveEvent(QDragMoveEvent *event)   // 拖动事件
{
    if (event->mimeData()->hasFormat("myimage/png")) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    } else {
        event->ignore();
    }
}
```

```
void MainWindow::dropEvent(QDropEvent *event) // 放下事件
{
    if (event->mimeData()->hasFormat("myimage/png")) {
        QByteArray itemData = event->mimeData()->data("myimage/png");
        QDataStream  dataStream(&itemData, QIODevice::ReadOnly);
        QPixmap pixmap;
        QPoint offset;
        // 使用数据流将字节数组中的数据读入到QPixmap和QPoint变量中
         dataStream >> pixmap >> offse
        // 新建标签，为其添加图片，并根据图
        QLabel *newLabel = new QLabel(
        newLabel->setPixmap(pixmap);
        newLabel->resize(pixmap.size());
        // 让图片移动到放下的位置，不设置的                                    角
        newLabel->move(event->pos() -
        newLabel->show();
        newLabel->setAttribute(Qt::WA_
        event->setDropAction(Qt::MoveA
        event->accept();
    } else {
        event->ignore();
    }
}
```



打印文档

Qt 5中的Qt Print Support模块提供了对打印的支持。
只需要使用一个QPrinter类和一个打印对话框
QPrintDialog类就可以完成文档的打印操作。

打印文档

打印预览

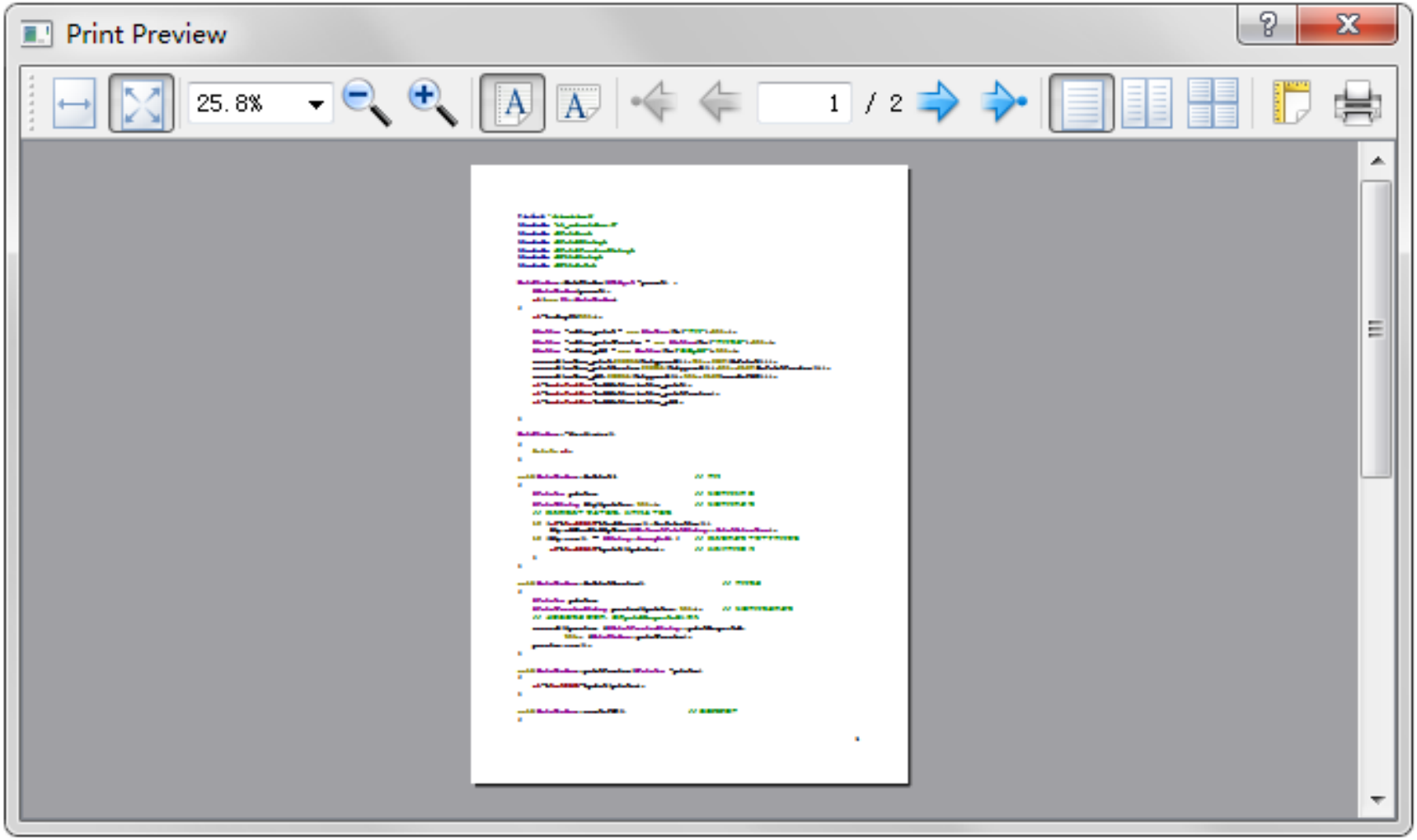
生成PDF文档

打印文档

```
QPrinter printer;                          // 创建打印机对象

QPrintDialog dlg(&printer, this);          // 创建打印对话框

// 如果编辑器中有选中区域，则打印选中区域
if (ui->textEdit->textCursor().hasSelection())
    dlg.addEnabledOption(QAbstractPrintDialog::PrintSelection);

if (dlg.exec() == QDialog::Accepted) {    // 如果在对话框中按下了打印
 按钮
    ui->textEdit->print(&printer);         // 则执行打印操作
```

在这里先建立了QPrinter类对象，它代表了一个打印设备。然后创建了一个打印对话框，如果编辑器中有选中区域则打印该区域，否则打印整个页面。

打印预览



生成PDF文件

```
QString fileName = QFileDialog::getSaveFileName(this, tr("导出PDF文件"),
                                                QString(), "*.pdf");
    if (!fileName.isEmpty()) {
        if (QFileInfo(fileName).suffix().isEmpty())
            fileName.append(".pdf");       // 如果文件后缀为空，则默认使用.pdf
        QPrinter printer;
        printer.setOutputFormat(QPrinter::PdfFormat);    // 指定输出格式为pdf
        printer.setOutputFileName(fileName);
        ui->textEdit->print(&printer);
    }
```

在生成PDF文档的槽中，使用文件对话框来获取要保存文件的路径，如果文件名没有指定后缀则为其添加“.pdf”后缀。然后为QPrinter对象指定输出格式和文件路径，这样就可以将文档打印成PDF格式了。

5.5 小结

通过学习这一章，要掌握主窗口各个部件的使用，能够自行开发出一个简单的基于QMainWindow的程序。对于资源文件的使用、富文本处理、拖放操作和打印文档等内容，需要了解其主要内容，在需要时可以直接拿来使用。

主窗口由哪几部分组成？

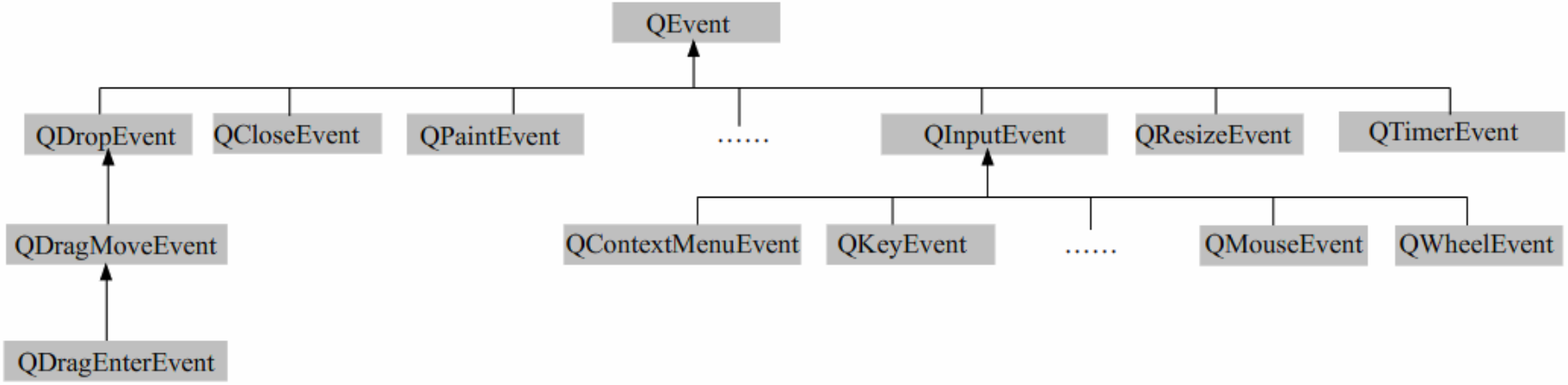
Qt资源文件是什么，有什么作用？

Qt中的富文本文档结构？

Qt中自定义拖放操作的实现步骤？

事件系统

在Qt中，事件作为一个对象，继承自QEvent类，常见的有键盘事件QKeyEvent、鼠标事件QMouseEvent和定时器事件QTimerEvent等，它们与QEvent类的继承关系如图所示。



主 要 内 容

- Qt中的事件
- 鼠标事件和滚轮事件
- 键盘事件
- 定时器事件与随机数
- 事件过滤器与事件的发送
- 小结

6.1 Qt中的事件

事件是对各种应用程序需要知道的由应用程序内部或者外部产生的事情或者动作的通称。在Qt中使用一个对象来表示一个事件，它继承自QEvent类。

事件与信号并不相同，比如我们使用鼠标点击了一下界面上的按钮，那么就会产生鼠标事件QMouseEvent（不是按钮产生的），而因为按钮被按下了，所以它会发出clicked()单击信号（是按钮产生的）。这里一般只关心按钮的单击信号，而不用考虑鼠标事件，但是如果要设计一个按钮，或者当鼠标点击按钮时让它产生别的效果，那么就要关心鼠标事件了。可以看到，事件与信号是两个不同层面的东西，它们的发出者不同，作用也不同。在Qt中，任何QObject的子类的实例都可以接收和处理事件。

常见事件：鼠标事件、键盘事件、定时事件、上下文菜单事件、关闭事件、拖放事件、绘制事件等。

- 事件的处理

- 事件的传递

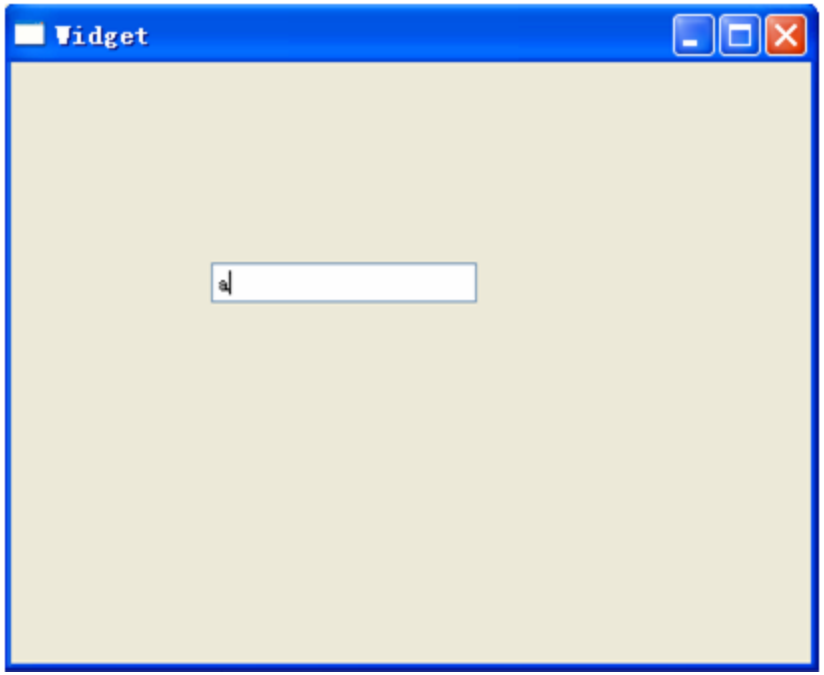
事件的处理

- 方法一：重新实现部件的paintEvent()，mousePressEvent()等事件处理函数。这是最常用也的一种方法，不过它只能用来处理特定部件的特定事件。例如前一章实现拖放操作，就是用的这种方法。
- 方法二：重新实现notify()函数。这个函数功能强大，提供了完全的控制，可以在事件过滤器得到事件之前就获得它们。但是，它一次只能处理一个事件。
- 方法三：向QApplication对象上安装事件过滤器。因为一个程序只有一个QApplication对象，所以这样实现的功能与使用notify()函数是相同的，优点是可以同时处理多个事件。
- 方法四：重新实现event()函数。QObject类的event()函数可以在事件到达默认的事件处理函数之前获得该事件。
- 方法五：在对象上安装事件过滤器。使用事件过滤器可以在一个界面类中同时处理不同子部件的不同事件。

在实际编程中，最常用的是方法一，其次是方法五。

重新实现事件处理函数

例如：使用自定义的Widget作为主窗口（继承自QWidget），然后在上　面放置一个自定义的MyLineEdit（继承自QLineEdit）。



- 在MyLineEdit中添加键盘按下事件处理函数声明：

```
protected:
    void keyPressEvent(QKeyEvent *event);
```

- 事件处理函数的定义：

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)    // 键盘按下事件
{
    qDebug() << tr("MyLineEdit键盘按下事件");
    QLineEdit::keyPressEvent(event);   // 执行QLineEdit类的默认事件处理
    event->ignore();                   // 忽略该事件
}
```

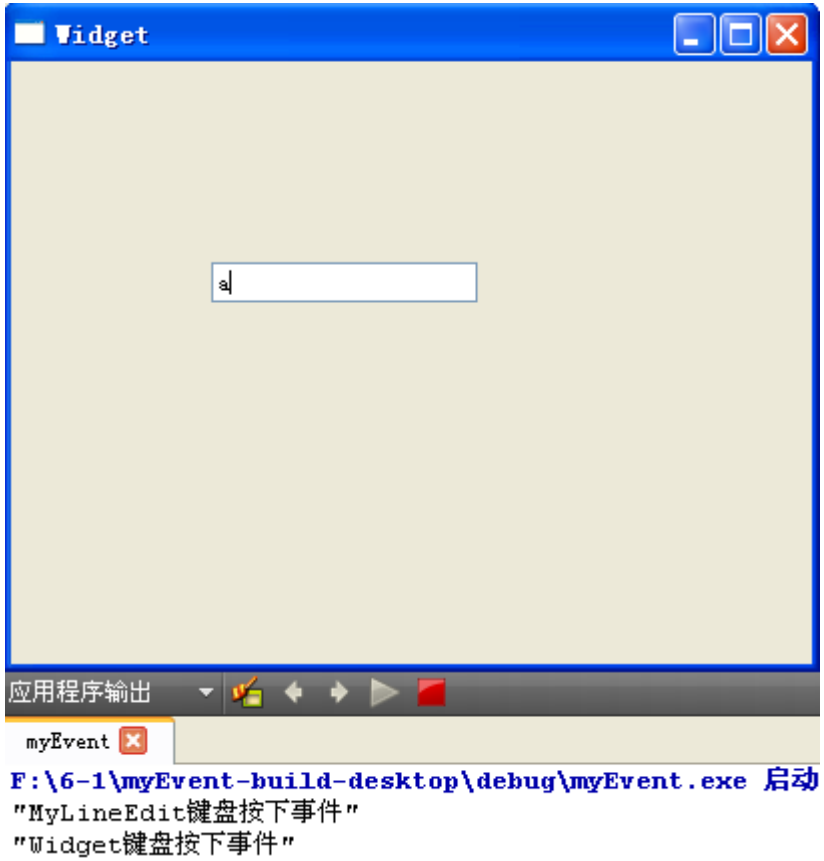
- 在Widget中添加键盘按下事件处理函数声明：

```
protected:
    void keyPressEvent(QKeyEvent *event);
```

- 事件处理函数的定义：

```
void Widget::keyPressEvent(QKeyEvent *event)
{
    qDebug() << tr("Widget键盘按下事件");
}
```

从这个例子中可以看到，事件是先传递给指定窗口部件的，这里确切的说应该是先传递给获得焦点的窗口部件的。但是如果该部件忽略掉该事件，那么这个事件就会传递给这个部件的父部件。在重新实现事件处理函数时，一般要调用父类的相应的事件处理函数来实现默认的操作。



安装事件过滤器

在MyLineEdit中添加函数声明：

```
bool event(QEvent *event);
```

该函数定义：

```
bool MyLineEdit::event(QEvent *event)  // 事件
{
    if(event->type() == QEvent::KeyPress)
        qDebug() << tr("MyLineEdit的event()函数");
    return QLineEdit::event(event);  //执行QLineEdit类event()函数的默认操作
}
```

在MyLineEdit的event()函数中使用了QEvent的type()函数来获取事件的类型，如果是键盘按下事件QEvent::KeyPress，则输出信息。因为event()函数具有bool型的返回值，所以在该函数的最后要使用return语句，这里一般是返回父类的event()函数的操作结果。

在Widget中进行事件过滤器函数的声明：

bool *eventFilter*(QObject *obj, QEvent *event);

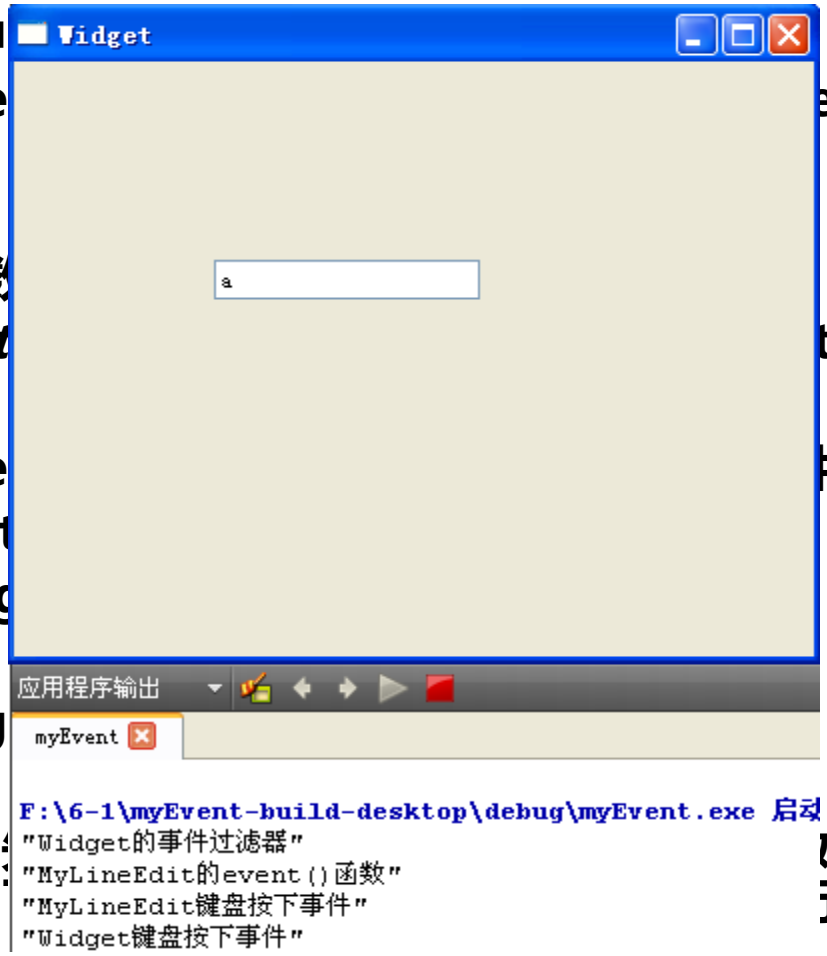
在widget.cpp文件中

lineEdit->installEve eEdit安装事件过滤器

下面是事件过滤器函数

```
bool Widget::event t) // 事件过滤器
{
        if(obj == line 上的事件
                if(event
                qDebug
         }
        return QWidg
}
```

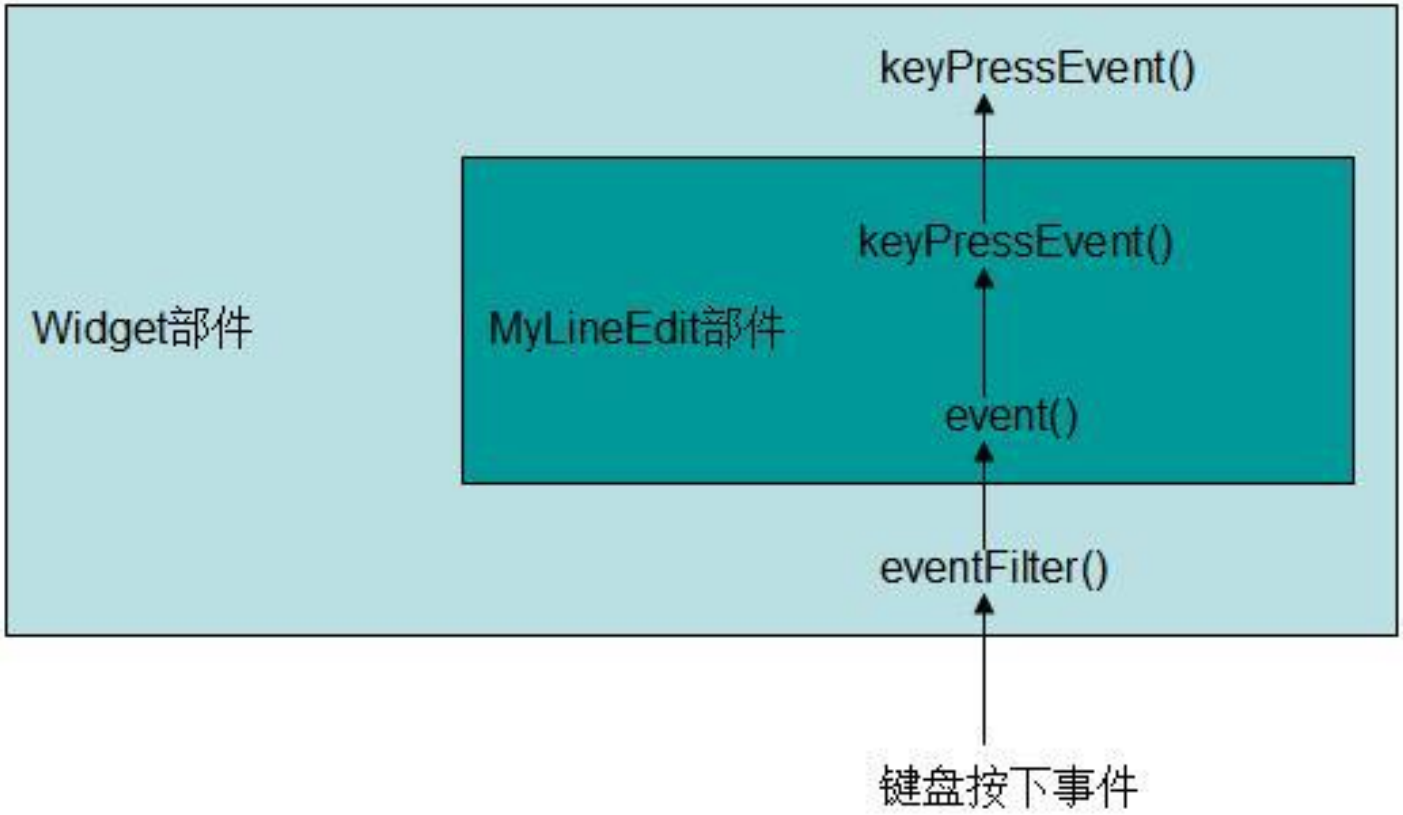
在事件过滤器中， 如果是，再判断事件类型。最后返回 行结果。



事件的传递

在每
函数
时接
QEv
其子

从前
然后
要注
而事



ec()
运行
或
器，
还
的，

6.2 鼠标事件和滚轮事件

QMouseEvent类用来表示一个鼠标事件，当在窗口部件中按下鼠标或者移动鼠标指针时，都会产生鼠标事件。利用**QMouseEvent**类可以获知鼠标是哪个键按下了，还有鼠标指针的当前位置等信息。通常是重定义部件的鼠标事件处理函数来进行一些自定义的操作。

QWheelEvent类用来表示鼠标滚轮事件，在这个类中主要是获取滚轮移动的方向和距离。

下面来看一个实际的例子，这个例子要实现的效果是：可以在界面上按着鼠标左键来拖动窗口，双击鼠标左键来使其全屏，按着鼠标右键则使指针变为一个自定义的图片，而使用滚轮则可以放大或者缩小编辑器中的内容。

```
void Widget::mousePressEvent(QMouseEvent *event) // 鼠标按下事件
{
    if(event->button() == Qt::LeftButton){       // 如果是鼠标左键按下
        QCursor cursor;
        cursor.setShape(Qt::ClosedHandCursor);
        QApplication::setOverrideCursor(cursor); // 使鼠标指针暂时改变形状
        offset = event->globalPos() - pos();    // 获取指针位置和窗口位置的差值
    }
    else if(event->button() == Qt::RightButton){ // 如果是鼠标右键按下
        QCursor cursor(QPixmap("../yafeilinux.png"));
        QApplication::setOverrideCursor(cursor);// 使用自定义的图片作为鼠标指针
    }
}
```

在鼠标按下事件处理函数中，先判断是哪个按键按下，如果是鼠标左键，那么就更改指针的形状，并且存储当前指针位置与窗口位置的差值。这里使用了globalPos()函数来获取鼠标指针的位置，这个位置是指针在桌面上的位置，因为窗口的位置就是指的它在桌面上的位置。另外，还可以使用QMouseEvent类的pos()函数获取鼠标指针在窗口中的位置。如果是鼠标右键按下，那么就将指针显示为我们自己的图片。

```
void Widget::mouseMoveEvent(QMouseEvent *event) // 鼠标移动事件
{
    if(event->buttons() & Qt::LeftButton){      // 这里必须使用buttons()
    QPoint temp;
    temp = event->globalPos() - offset;
    move(temp);// 使用鼠标指针当前的位置减去差值，就得到了窗口应该移动的位
     置
    }
}
```

在鼠标移动事件处理函数中，先判断是否是鼠标左键按下，如果是，那么就使用前面获取的差值来重新设置窗口的位置。因为在鼠标移动时，会检测所有按下的键，而这时使用QMouseEvent的button()函数无法获取哪个按键被按下，只能使用buttons()函数，所以这里使用buttons()和Qt::LeftButton进行按位与的方法来判断是否是鼠标左键按下。

```
void Widget::mouseReleaseEvent(QMouseEvent *event) // 鼠标释放事件
{
    QApplication::restoreOverrideCursor();        // 恢复鼠标指针形状
}
```

在鼠标释放函数中进行了恢复鼠标形状的操作，这里使用的restoreOverrideCursor()函数要和前面的setOverrideCursor()函数配合使用。

```
void Widget::mouseDoubleClickEvent(QMouseEvent *event) // 鼠标双击事
    件
{
   if(event->button() == Qt::LeftButton){            // 如果是鼠标左键按下
      if(windowState() != Qt::WindowFullScreen)      // 如果现在不是全屏
         setWindowState(Qt::WindowFullScreen);       // 将窗口设置为全屏
      else setWindowState(Qt::WindowNoState);        // 否则恢复以前的大小
   }
}
```

在鼠标双击事件处理函数中使用setWidowState()函数来使窗口处于全屏状态或者恢复以前的大小。

```
void Widget::wheelEvent(QWheelEvent *event)    // 滚轮事件
{
    if(event->delta() > 0){                    // 当滚轮远离使用者时
        ui->textEdit->zoomIn();                // 进行放大
    }else{                                     // 当滚轮向使用者方向旋转时
        ui->textEdit->zoomOut();               // 进行缩小
    }
}
```

在滚轮事件处理函数中，使用QWheelEvent类的delta()函数获取了滚轮移动的距离，每当滚轮旋转一下，默认的是15度，这时delta()函数就会返回15*8即整数120。当滚轮向远离使用者的方向旋转时，返回正值；当向着靠近使用者的方向旋转时，返回负值。这样便可以利用这个函数的返回值来判断滚轮的移动方向，从而进行编辑器中内容的放大或者缩小操作。

6.3 键盘事件

QKeyEvent类用来描述一个键盘事件。当键盘按键被按下或者被释放时，键盘事件便会被发送给拥有键盘输入焦点的部件。

QKeyEvent的key()函数可以获取具体的按键，需要特别说明的是，回车键在这里是Qt::Key_Return；键盘上的一些修饰键，比如Ctrl和Shift等，这里需要使用QKeyEvent的modifiers()函数来获取它们。例如：

```
void Widget::keyPressEvent(QKeyEvent *event)      // 键盘按下事件
{
   if(event->modifiers() == Qt::ControlModifier){ // 是否按下Ctrl键
      if(event->key() == Qt::Key_M)            // 是否按下M键
         setWindowState(Qt::WindowMaximized);  // 窗口最大化
   }
   else QWidget::keyPressEvent(event);
}

void Widget::keyReleaseEvent(QKeyEvent *event)    // 按键释放事件
{
   // 其他操作
}
```

6.4 定时器事件与随机数

QTimerEvent类用来描述一个定时器事件。对于一个QObject的子类，只需要使用int QObject::startTimer (int interval)函数来开启一个定时器，这个函数需要输入一个以毫秒为单位的整数作为参数来表明设定的时间，它返回一个整型编号来代表这个定时器。当定时器溢出时就可以在timerEvent()函数中获取该定时器的编号来进行相关操作。

编程中更多的是使用QTimer类来实现一个定时器，它提供了更高层次的编程接口，比如可以使用信号和槽，还可以设置只运行一次的定时器。所以在以后的章节中，如果使用定时器，那么一般都是使用的QTimer类。

关于随机数，在Qt中是使用qrand()和qsrand()两个函数实现的。

通过ID使用定时器

使用QTimerEvent的timerId()函数来获取定时器的编号，然后判断是哪一个定时器并分别进行不同的操作。

在构造函数中：

```
id1 = startTimer(1000);              // 开启一个1秒定时器，返回其ID
id2 = startTimer(2000);
id3 = startTimer(3000);
```

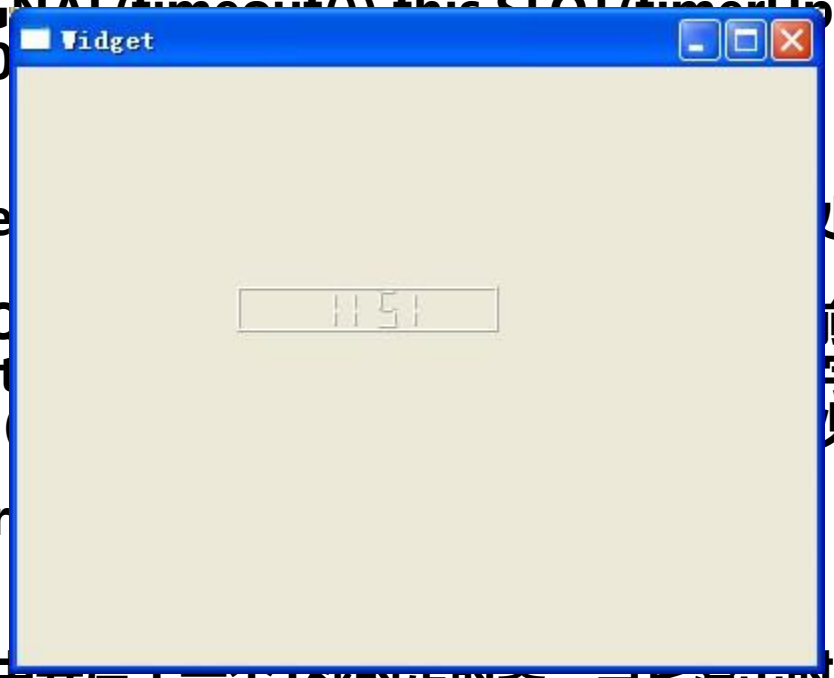
下面是定时器事件函数的定义：

```
void Widget::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == id1) {       // 判断是哪个定时器
        qDebug() << "timer1";
    }
    else if (event->timerId() == id2) {
        qDebug() << "timer2";
    }
    else {
        qDebug() << "timer3";
    }
}
```

通过信号和槽实现定时器

在构造函数中：

```
QTimer *timer = new QTimer(this);          // 创建一个新的定时器
// 关联定时器的溢出信号到槽上
connect(timer,SIG                                          date()));
timer->start(1000
```



溢出处理：

```
void Widget::time                                          处理
{
    QTime time = C                                         前时间
    QString text = t                                       字符串
    if((time.second(                                       少就将“：”显示为
     空格
    ui->lcdNumber
}
```

这里在构造函数中开启了一个1秒的定时器，当它溢出时就会发射timeout()信号，这时就会执行我们的定时器溢出处理函数。在槽里我们获取了当前的时间，并且将它转换为可以显示的字符串。

随机数

构造函数里添加一行代码：

```
qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
```

然后在timerUpdate()函数里面添加如下代码：

```
int rand = qrand() % 300;          // 产生300以内的正整数
ui->lcdNumber->move(rand, rand);
```

在使用qrand()函数产生随机数之前，一般要使用qsrand()函数为其设置初值，如果不设置初值，那么每次运行程序，qrand()都会产生相同的一组随机数。为了每次运行程序时，都可以产生不同的随机数，我们要使用qsrand()设置一个不同的初值。这里使用了QTime类的secsTo()函数，它表示两个时间点之间所包含的秒数，比如代码中就是指从零点整到当前时间所经过的秒数。当使用qrand()要获取一个范围内的数值时，一般是让它与一个整数取余，比如这里与300取余，就会使所有生成的数值在0-299之间（包含0和299）。

事件过滤器与事件的发送

自学：

- 事件过滤器前面已经讲过了，需要课下对事件过滤器进行更加深入的学习。
- Qt的事件系统可以获取事件，那么可以手动发送事件吗？

小结

学习完本章，要掌握基本的事件的处理方法，包括重新实现事件处理函数和使用事件过滤器。对于定时器和随机数的知识，它们在实现一些特殊效果以及动画和游戏中会经常使用到，要学会使用。

- Qt中处理事件的5种方法？
- Qt中事件传递顺序？
- Qt中鼠标滚轮事件中如何计算滚轮移动距离？
- Qt中定时器的两种使用方法？