

PyQt4 精彩实例分析 实例 1 Hello Kitty!

本实例实现一个"Hello Kitty!"例子，简单介绍 Qt 编程的基本流程，以及 Qt 程序的编译运行方式，实例效果图如图所示。



这是一个简单的例子，整个对话框只有一个按钮，单击该按钮，对话框关闭，退出程序。

实现代码如下：

```
[python]
01. from PyQt4.QtGui import *
02. from PyQt4.QtCore import *
03. import sys
04.
05. app=QApplication(sys.argv)
06. b=QPushButton("Hello Kitty!")
07. b.show()
08. app.connect(b,SIGNAL("clicked()"),app,SLOT("quit()"))
09. app.exec_()
```

第 1 行导入 `PyQt4.QtGui` 的所有类及模块，包括 `QApplication`，所有 Qt 图形化应用程序都必须包含此文件，它包含了 Qt 图形化应用程序的各种资源，基本设置，控制流以及事件处理等。

第 5 行新创建了一个 `QApplication` 对象，每个 Qt 应用程序都必须有且只有一个 `QApplication` 对象，采用 `sys.argv` 作为参数，便于程序处理命令行参数。

第 6 行创建了一个 `QPushButton` 对象，并设置它的显示文本为“Hello Kitty! ”，由于此处并没有指定按钮的父窗体，因此以自己作为主窗口。

第 7 行调用 `show()` 方法，显示此按钮。控件被创建时，默认是不显示的，必须调用 `show()` 函数来显示它。

第 8 行的 `connect` 方法是 Qt 最重要的特征，即信号与槽的机制。当按钮被按下则触发 `clicked` 信号，与之相连的 `QApplication` 对象的槽 `quit()` 响应按钮单击信号，执行退出应用程序的操作。关于信号与槽机制在本实例最后将进行详细的分析。

最后调用 `QApplication` 的 `exec_()` 方法，程序进入消息循环，等待可能输入进行响应。Qt 完成事件处理及显示的工作，并在应用程序退出时返回 `exec_()` 的值。

最后执行程序即可出现上图所示的对话框，一个简单的 Hello Kitty! 例子完成。

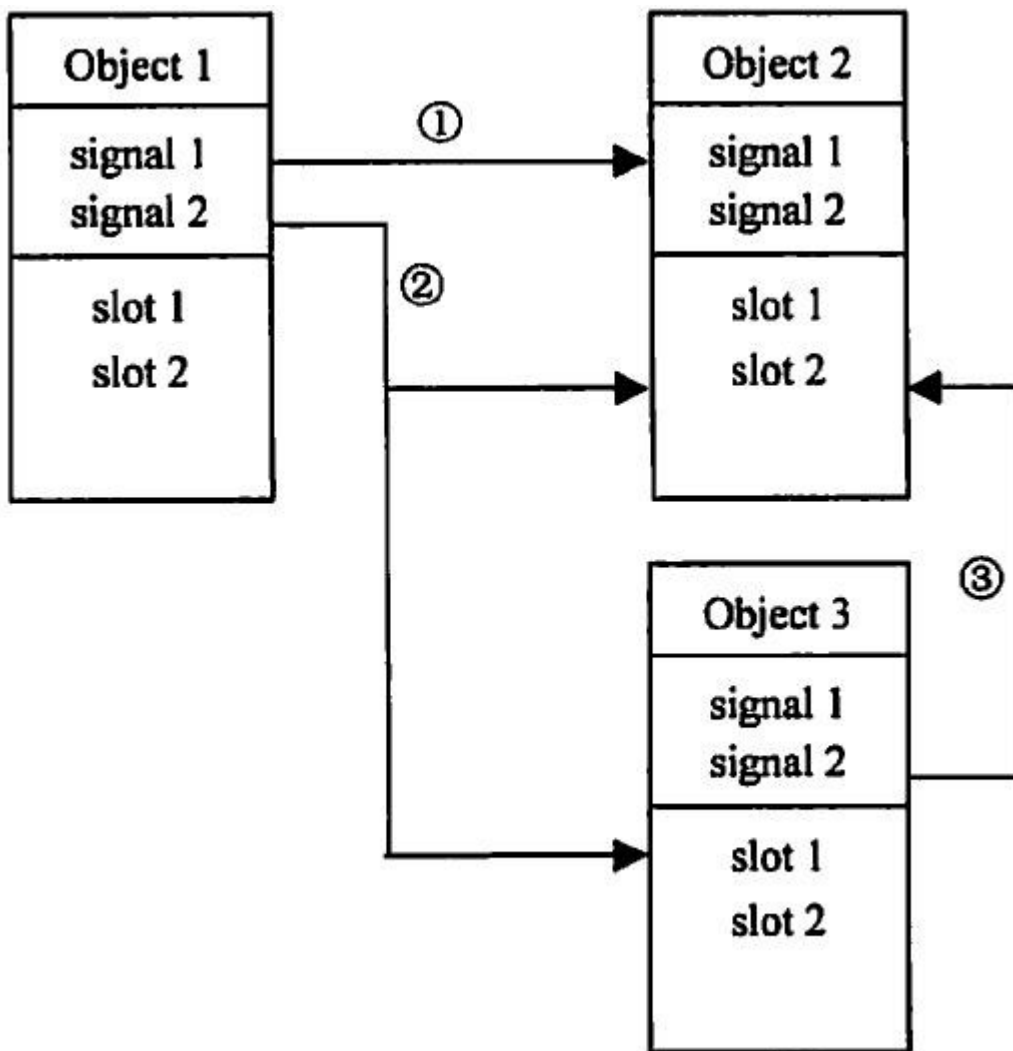
信号与槽机制作为 Qt 最重要的特性，提供了任意两个 Qt 对象之间的通信机制。其中，信号会在某个特定情况或动作下被触发，槽是用于接收并处理信号的函数。例如，要将一个窗口中的变化情况通知给另一个窗口，则一个窗口发送信号，另一个窗口的槽接收此信号并进行相应的操作，即可实现两个窗口之间的通信。这比传统的图形化程序采用回调函数的方式实现对象间通信要简单灵活得多。每个 Qt 对象都包含预定的信号和槽，当某一特定事件发生时，一个信号被发射，与信号相关联的槽则会响应信号完成相应的处理。

信号与槽机制常用的连接方式为：

```
connect(Object1,SIGNAL(signal),Object2,SLOT(slot))
```

`signal` 为对象 `Object1` 的信号，`slot` 为对象 `Object2` 的槽，Qt 的窗口部件都包含若干个预定义的信号和若干个预定义的槽。当一个类被继承时，该类的信号和槽也同时被继承。开始人员也可以根据需求定义自己的信号和槽。

信号与槽机制可以有多种连接方式，下图描述了信号与槽的多种可能连接方式。



1. 一个信号可以与另一个信号相连

```
connect(Object1,SIGNAL(signal1),Object2,SIGNAL(signal1))
```

即表示 Object1 的信号 1 发射可以触发 Object2 的信号 1 发射。

2. 表示一个信号可以与多个槽相连

```
connect(Object1,SIGNAL(signal2),Object2,SLOT(slot2))
```

```
connect(Object1,SIGNAL(signal2),Object3,SLOT(slot1))
```

3. 表示同一个槽可以响应多个信号

```
connect(Object1,SIGNAL(signal2),Object2,SLOT(slot2))
```

```
connect(Object3,SIGNAL(signal2),Object2,SLOT(slot2))
```

PyQt4 精彩实例分析 实例 2 标准对话框的使用

和大多数操作系统一样，Windows 及 Linux 都提供了一系列的标准对话框，如文件选择，字体选择，颜色选择等，这些标准对话框为应用程序提供了一致的观感。Qt 对这些标准对话框都定义了相关的类，这些类让使用者能够很方便地使用标准对话框进行文件，颜色以及字体的选择。标准对话框在软件设计过程中是经常需要使用的。

Qt 提供的标准对话框除了本实例提到的，还有

`QErrorMessage`, `QInputDialog`, `QMessageBox`, `QPrintDialog`, `QProcessDialog` 等，这些标准对话框的使用在本书的后续部分将会陆续介绍。

本实例主要演示上面几种标准对话框的使用，如下图所示



在上图中，单击“文件对话框”按钮，会弹出文件选择对话框，选中的文件名将显示在右边，单击“颜色对话框”按钮，会弹出颜色选择对话框，选中的颜色将显示在右边，单击“字体对话框”按钮，会弹出字体选择对话框，选中的字体将更新右边显示的字符串。

实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class StandardDialog(QDialog):

    def __init__(self, parent=None):
        super(StandardDialog, self).__init__(parent)

        self.setWindowTitle("Standard Dialog")

        filePushButton=QPushButton(self.tr("文件对话框"))
        colorPushButton=QPushButton(self.tr("颜色对话框"))
        fontPushButton=QPushButton(self.tr("字体对话框"))

        self.fileLineEdit=QLineEdit()
        self.colorFrame=QFrame()
        self.colorFrame.setFrameShape(QFrame.Box)
        self.colorFrame.setAutoFillBackground(True)
        self.fontLineEdit=QLineEdit("Hello World!")

        layout=QGridLayout()
        layout.addWidget(filePushButton,0,0)
        layout.addWidget(self.fileLineEdit,0,1)
        layout.addWidget(colorPushButton,1,0)
        layout.addWidget(self.colorFrame,1,1)
        layout.addWidget(fontPushButton,2,0)
        layout.addWidget(self.fontLineEdit,2,1)
```

```

        self.setLayout(layout)

        self.connect(filePushButton, SIGNAL("clicked()"), self.openFile)
        self.connect(colorPushButton, SIGNAL("clicked()"), self.openColor)
        self.connect(fontPushButton, SIGNAL("clicked()"), self.openFont)

    def openFile(self):

        s=QFileDialog.getOpenFileName(self, "Open file dialog", "/", "Python
files(*.py)")
        self.fileLineEdit.setText(str(s))

    def openColor(self):

        c=QColorDialog.getColor(Qt.blue)
        if c.isValid():
            self.colorFrame.setPalette(QPalette(c))

    def openFont(self):

        f,ok=QFontDialog.getFont()
        if ok:
            self.fontLineEdit.setFont(f)

    app=QApplication(sys.argv)
    form=StandardDialog()
    form.show()
    app.exec_()

```

第 6 行设定 `tr` 方法使用 `utf8` 编码来解析文字。

第 13 行设置程序的标题。

第 15 到 17 行创建各个按钮控件。

第 19 行创建一个 `QLineEdit` 类实例 `fileLineEdit`，用来显示选择的文件名。

第 20 行创建一个 `QFrame` 类实例 `colorFrame`，当用户选择不同的颜色时，`colorFrame` 会根据用户选择的颜色更新其背景。

第 23 行创建一个 `QLineEdit` 类实例 `fontLineEdit`，当用户选择不同的字体时，`fontLineEdit` 会根据用户选择的字体更新其内容。

第 25 到 33 行将各个控件进行布局。

第 35 到 37 行将各个按钮的 `clicked` 信号相应的槽进行连接。

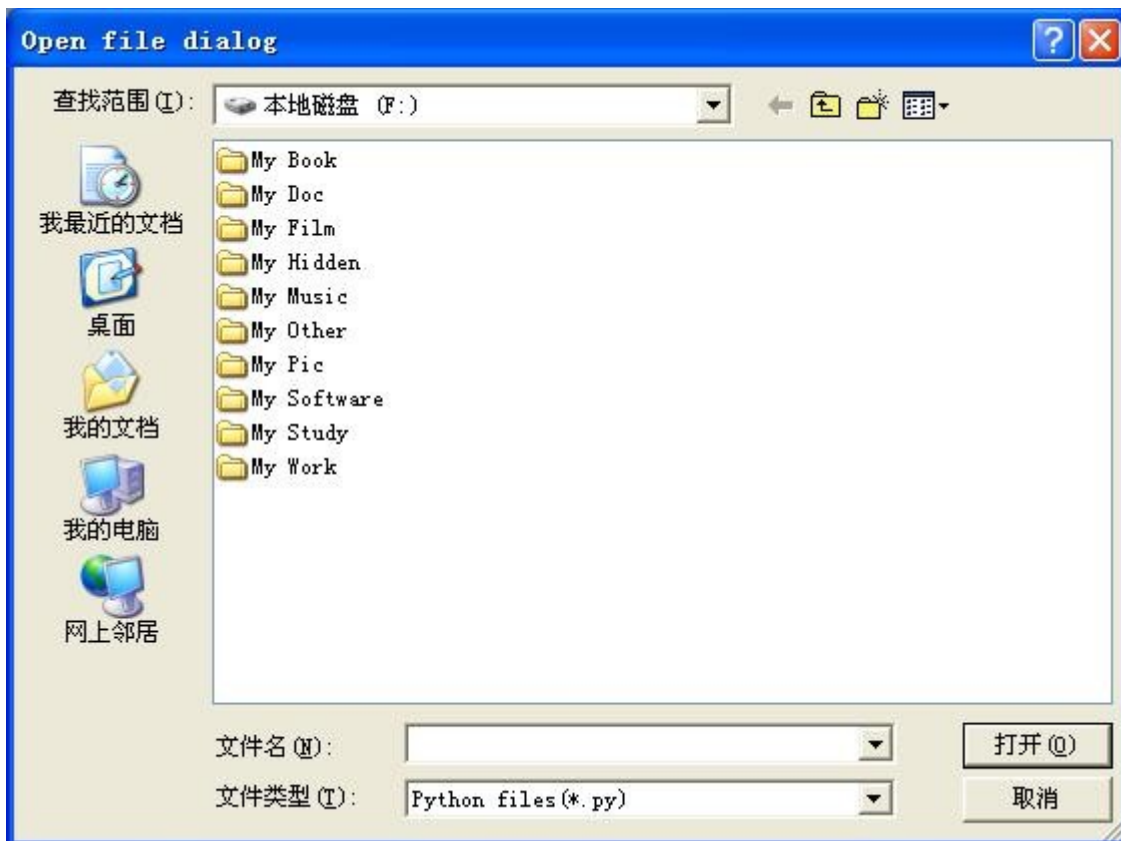
`slotFile()` 方法是文件对话框按钮的 `clicked` 信号的槽，其中 `getOpenFileName()` 是 `QFileDialog` 类的一个静态方法，返回用户选择的文件名，如果用户选择取消，则返回一个空串。函数形式如下：

```

QString getOpenFileName (QWidget parent = None, QString caption = QString(),
QString directory = QString(), QString filter = QString(), Options options = 0)
QString getOpenFileName (QWidget parent = None, QString caption = QString(),
QString directory = QString(), QString filter = QString(), QString
selectedFilter = None, Options options = 0)

```

调用 `getOpenFileName()` 函数将创建一个模态的文件对话框，如下图所示。`directory` 参数指定了默认的目录，如果 `directory` 参数带有文件名，则该文件将是默认选中的文件，`filter` 参数对文件类型进行过滤，只有与过滤器匹配的文件类型才显示，`filter` 可以同时指定多种过滤方式供用户选择，多种过滤器之间用 `;;` 隔开，用户选择的过滤器通过参数 `selectedFilter` 返回。



`QFileDialog` 类还提供了类似的其他静态函数，如下表，通过这些函数，用户可以方便地定制自己的文件对话框。

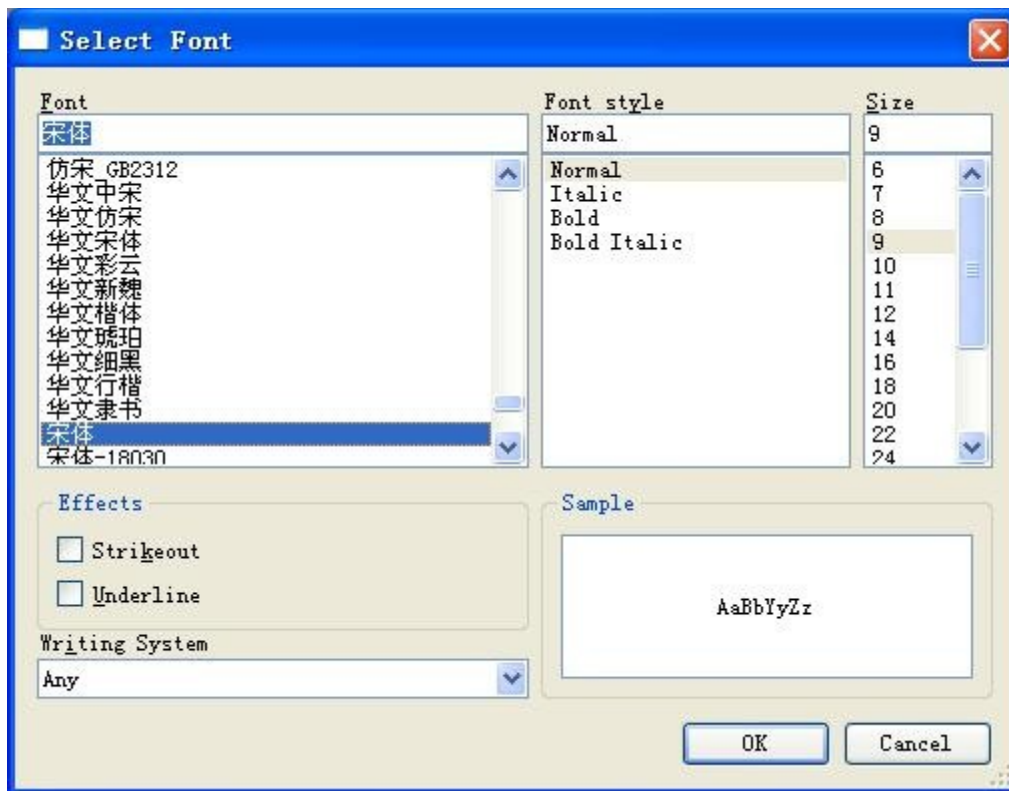
静态函数	说明
<code>getOpenFileName</code>	获得用户选择的文件名
<code>getSaveFileName</code>	获得用户保存的文件名
<code>getExistingDirectory</code>	获得用户选择的已存在的目录名
<code>getOpenFileNames</code>	获得用户选择的文件名列表

`slotColor()` 函数是颜色对话框按钮 `clicked` 信号的槽。其中 `getColor()` 是 `QColorDialog` 的一个静态函数，返回用户选择的颜色值，函数形式如下：

```
QColor QColorDialog.getColor (QColor initial = Qt.white, QWidget parent = None)
```

```
QColor QColorDialog.getColor (QColor, QWidget, QString, QColorDialogOptions  
options = 0)
```

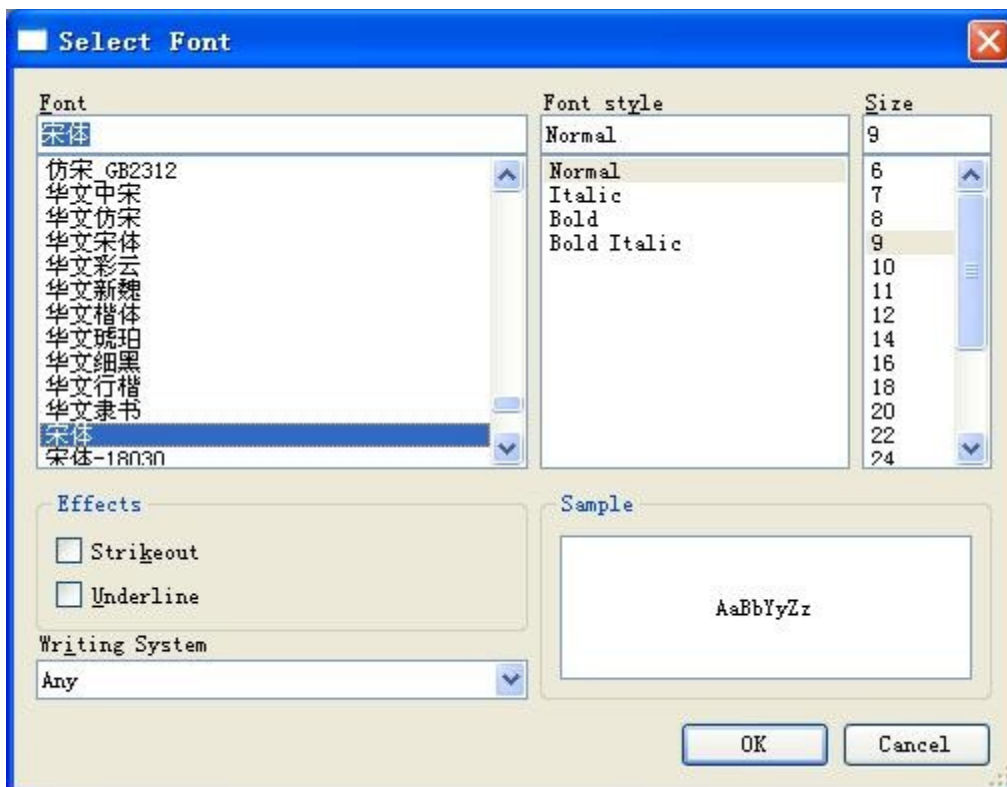
调用 `getColor()` 函数将创建一个模态的颜色对话框，如下图所示。`initial` 参数指定了默认的颜色，默认为白色，通过 `isValid()` 可以判断用户选择的颜色是否有效，若用户选择取消，`isValid()` 将返回 `false`。



slotFont()函数是字体对话框按钮 clicked 信号的槽。其中 getFont()是 QFontDialog 的一个静态函数，返回用户选择的字体，函数形式如下：

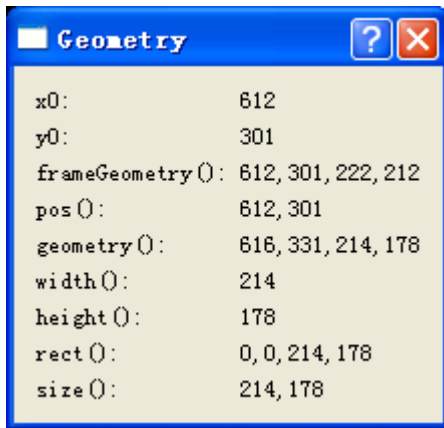
```
(QFont, bool) getFont (QFont, QWidget, QString, FontDialogOptions)
(QFont, bool) getFont (QFont, QWidget, QString)
(QFont, bool) getFont (QFont, QWidget parent = None)
(QFont, bool) getFont (QWidget parent = None)
```

调用 getFont()函数将创建一个模态的字体对话框，如下图所示。用户选择 OK，函数返回(用户选择的字体, True)，否则返回(默认字体, False)



PyQt4 精彩实例分析 实例 3 各类位置信息

Qt 提供了很多关于获取窗体位置及显示区域大小的函数，本实例利用一个简单的对话框显示窗体的各种位置信息，包括窗体的所在点位置，长，宽信息等。本实例的目的是分析各个有关位置信息的函数之间的区别，如 `x()`, `y()`, `pos()`, `rect()`, `size()`, `geometry()` 等，以及在不同的情况下应使用哪个函数来获取位置信息。实现的效果如下图：



在实例中，分别调用了 `x()`, `y()`, `frameGeometry()`, `pos()`, `geometry()`, `width()`, `height()`, `rect()`, `size()` 几个函数，这几个函数均是 `QWidget` 提供的。当改变对话框的大小，或移动对话框时，调用各个函数所获得的信息显示也相应地发生变化，从变化中可得知各函数之间的区别。

由于本实例的目的是为了分析各函数之间的区别并获得结论，而程序本身的实现较为简单，因此只简单介绍实现过程。实现代码如下：

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import sys

class Geometry(QDialog):

    def __init__(self, parent=None):
        super(Geometry, self).__init__(parent)

        self.setWindowTitle("Geometry")

        Label1=QLabel("x0:")
        Label2=QLabel("y0:")
        Label3=QLabel("frameGeometry():")
        Label4=QLabel("pos():")
        Label5=QLabel("geometry():")
        Label6=QLabel("width():")
        Label7=QLabel("height():")
        Label8=QLabel("rect():")
        Label9=QLabel("size():")

        self.xLabel=QLabel()
        self.yLabel=QLabel()
        self.frameGeoLabel=QLabel()
        self.posLabel=QLabel()
        self.geoLabel=QLabel()
        self.widthLabel=QLabel()
        self.heightLabel=QLabel()
        self.rectLabel=QLabel()
        self.sizeLabel=QLabel()
```



```

layout=QGridLayout()
layout.addWidget(Label1,0,0)
layout.addWidget(self.xLabel,0,1)
layout.addWidget(Label2,1,0)
layout.addWidget(self.yLabel,1,1)
layout.addWidget(Label3,2,0)
layout.addWidget(self.frameGeoLabel,2,1)
layout.addWidget(Label4,3,0)
layout.addWidget(self.posLabel,3,1)
layout.addWidget(Label5,4,0)
layout.addWidget(self.geoLabel,4,1)
layout.addWidget(Label6,5,0)
layout.addWidget(self.widthLabel,5,1)
layout.addWidget(Label7,6,0)
layout.addWidget(self.heightLabel,6,1)
layout.addWidget(Label8,7,0)
layout.addWidget(self.rectLabel,7,1)
layout.addWidget(Label9,8,0)
layout.addWidget(self.sizeLabel,8,1)

self.setLayout(layout)

self.updateLabel()

def moveEvent(self,event):
    self.updateLabel()

def resizeEvent(self,event):
    self.updateLabel()

def updateLabel(self):
    temp=QString()

    self.xLabel.setText(temp.setNum(self.x()))
    self.yLabel.setText(temp.setNum(self.y()))
    self.frameGeoLabel.setText(temp.setNum(self.frameGeometry().x())+", "+
                                temp.setNum(self.frameGeometry().y())+", "+
                                temp.setNum(self.frameGeometry().width())+", "+
temp.setNum(self.frameGeometry().height()))

    self.posLabel.setText(temp.setNum(self.pos().x())+", "+
                           temp.setNum(self.pos().y()))
    self.geoLabel.setText(temp.setNum(self.geometry().x())+", "+
                           temp.setNum(self.geometry().y())+", "+
                           temp.setNum(self.geometry().width())+", "+
                           temp.setNum(self.geometry().height()))
    self.widthLabel.setText(temp.setNum(self.width()))
    self.heightLabel.setText(temp.setNum(self.height()))
    self.rectLabel.setText(temp.setNum(self.rect().x())+", "+
                            temp.setNum(self.rect().y())+", "+
                            temp.setNum(self.rect().width())+", "+
                            temp.setNum(self.rect().height()))
    self.sizeLabel.setText(temp.setNum(self.size().width())+", "+
                           temp.setNum(self.size().height()))

app=QApplication(sys.argv)
form=Geometry()
form.show()
app.exec_()

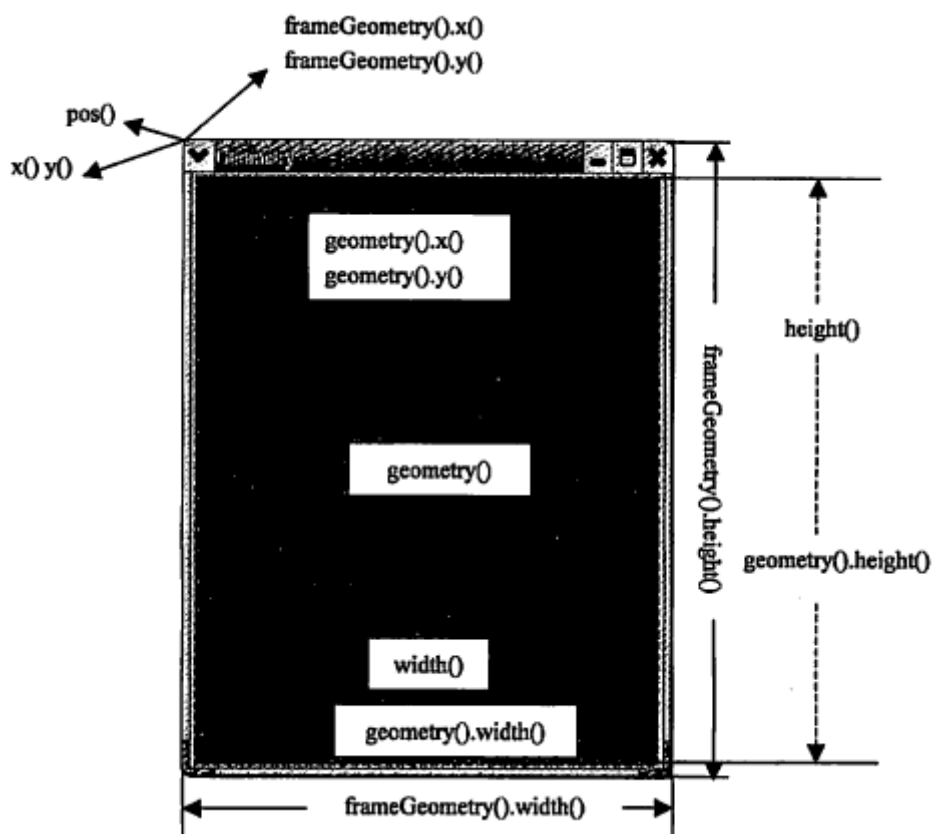
```


程序初始化时调用 `updateLabel()` 函数，以获得各位置函数的信息并显示。

`updateLabel()` 函数负责调用各个位置函数获得结果并显示。

重定义 `QWidget` 的 `moveEvent()` 和 `resizeEvent()` 函数，分别响应对话框的移动事件和调整大小事件，使得窗体在被移动或窗体大小发生改变时，能同步更新各函数结果的显示。

通过这个例子可以获得如下图所示的结论。



`x()`, `y()` 和 `pos()` 函数都是获得整个窗体左上角的坐标位置。而 `frameGeometry()` 与 `geometry()` 相对应，`frameGeometry()` 是获得整个窗体的左上顶点和长，宽值，而 `geometry()` 函数获得的是窗体内中央区域的左上顶点坐标以及长，宽值。直接调用 `width()` 和 `height()` 函数获得的是中央区域的长和宽的值。

还有两个函数 `rect()` 和 `size()`，调用它们获得的结果也都是对于窗体的中央区域而言的，`size()` 获得的是窗体中央区域的长，宽值，`rect()` 与 `geometry()` 一样返回一个 `QRect` 对象。其中，两个函数获得的长，宽值是一样的，都是窗体中央区域的长，宽值，只是左上顶点坐标值不一样，`geometry()` 获得的左上顶点坐标是相对于父窗体而言的坐标，而 `rect()` 获得的左上顶点坐标始终为 `(0, 0)`。

因此，在实际应用中需根据情况使用正确的位置信息函数以获得准确的位置尺寸信息，尤其是在编写对位置精度要求较高的程序时，如地图浏览程序，更应注意函数的选择，避免产生不必要的误差。

在编写程序时，初始化窗体时最好不要使用 `setGeometry()` 函数，而用 `resize()` 和 `move()` 函数代替，因为使用 `setGeometry()` 会导致窗体 `show()` 之后在错误的位置上停留很短暂的一段时间，带来闪烁现象。

PyQt4 精彩实例分析 实例 4 使用标准输入框

本实例演示如何使用标准输入框，Qt 提供了一个 `QInputDialog` 类，`QInputDialog` 类提供了一种简单方面的对话框来获得用户的单个输入信息，目前提供了 4 种数据类型的输入，可以是一个字符串，一个 `Int` 类型数据，一个 `double` 类型数据或是一个下拉列表框的条目。其中包括一个提示标签，一个输入控件。若是调用字符串输入框，则为一个 `QLineEdit`，若是调用 `Int` 类型或 `double` 类型，则为一个 `QSpinBox`，若是调用列表条目输入框，则为一个 `QComboBox`，还包括一个确定输入(Ok)按钮和一个取消输入(Cancel)按钮。

本实例的实现效果如下图。



实例中列举了以上 4 种输入类型，右侧的按钮用于弹出标准输入对话框修改各条信息的值。具体实现代码如下所示：

```
# -*- coding: utf-8 -*-
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class InputDlg(QDialog):
    def __init__(self, parent=None):
        super(InputDlg, self).__init__(parent)

        label1=QLabel(self.tr("姓名"))
        label2=QLabel(self.tr("性别"))
        label3=QLabel(self.tr("年龄"))
        label4=QLabel(self.tr("身高"))

        self.nameLabel=QLabel("TengWei")
        self.nameLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
        self.sexLabel=QLabel(self.tr("男"))
        self.sexLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
        self.ageLabel=QLabel("25")
        self.ageLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
        self.statureLabel=QLabel("168")
        self.statureLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)

        nameButton=QPushButton("...")
        sexButton=QPushButton("...")
        ageButton=QPushButton("...")
        statureButton=QPushButton("...")

        self.connect(nameButton, SIGNAL("clicked()"), self.slotName)
        self.connect(sexButton, SIGNAL("clicked()"), self.slotSex)
        self.connect(ageButton, SIGNAL("clicked()"), self.slotAge)
        self.connect(statureButton, SIGNAL("clicked()"), self.slotStature)

        layout=QGridLayout()
```

```

        layout.addWidget(label1,0,0)
        layout.addWidget(self.nameLabel,0,1)
        layout.addWidget(nameButton,0,2)
        layout.addWidget(label2,1,0)
        layout.addWidget(self.sexLabel,1,1)
        layout.addWidget(sexButton,1,2)
        layout.addWidget(label3,2,0)
        layout.addWidget(self.ageLabel,2,1)
        layout.addWidget(ageButton,2,2)
        layout.addWidget(label4,3,0)
        layout.addWidget(self.statureLabel,3,1)
        layout.addWidget(statureButton,3,2)

    self.setLayout(layout)

    self.setWindowTitle(self.tr("资料收集"))

def slotName(self):
    name,ok=QInputDialog.getText(self,self.tr("用户名"),
                                self.tr("请输入新的名字:"),
                                QLineEdit.Normal,self.nameLabel.text())
    if ok and (not name.isEmpty()):
        self.nameLabel.setText(name)

def slotSex(self):
    list=QStringList()
    list.append(self.tr("男"))
    list.append(self.tr("女"))
    sex,ok=QInputDialog.getItem(self,self.tr("性别"),self.tr("请选择性别"),list)

    if ok:
        self.sexLabel.setText(sex)

def slotAge(self):
    age,ok=QInputDialog.getInteger(self,self.tr("年龄"),
                                self.tr("请输入年龄:"),
                                int(self.ageLabel.text()),0,150)

    if ok:
        self.ageLabel.setText(str(age))

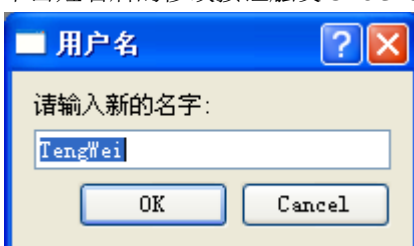
def slotStature(self):
    stature,ok=QInputDialog.getDouble(self,self.tr("身高"),
                                    self.tr("请输入身高:"),
                                    float(self.statureLabel.text()),0,2300.00)
    if ok:
        self.statureLabel.setText(str(stature))

app=QApplication(sys.argv)
form=InputDlg()
form.show()
app.exec_()

```

构造函数中声明了对话框中用到的控件以及各按钮触发的槽函数。

单击姓名后的修改按钮触发 `slotName()` 函数，弹出标准字符串输入对话框，如下图。



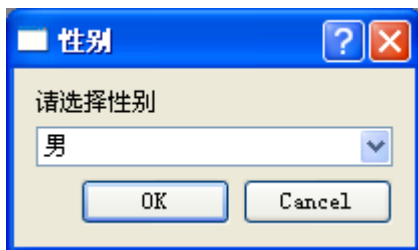
调用 `QInputDialog` 的 `getText()` 函数弹出标准字符串输入对话框，`getText()` 函数原型如下：

```
(QString, bool ok) QInputDialog.getText (QWidget, QString, QString,  
QLineEdit.EchoMode mode = QLineEdit.Normal, QString text = QString(),  
Qt.WindowFlags flags = 0)
```

此函数的第一个参数为标准输入对话框的父窗口，第二个参数为标准输入对话框的标题名，第三个参数为标准输入对话框的标签提示，第四个参数 `mode` 指定标准输入对话框中 `QLineEdit` 控件的输入模式，第五个参数 `text` 为标准字符串输入对话框弹出时 `QLineEdit` 控件默认出现的文字，最后一个参数指明标准输入对话框的窗体标识。

`slotName()` 函数中的第 3 行判断 `ok` 的值，若用户单击了“OK”按钮，则把新输入的姓名更新至显示标签。

单击性别后的修改按钮触发 `slotSex()` 函数，弹出标准条目选择对话框，如下图。



第 1, 2, 3 行创建一个 `QStringList` 对象，包括两个 `QString` 项，用于标准输入对话框中下拉列表框的条目显示。

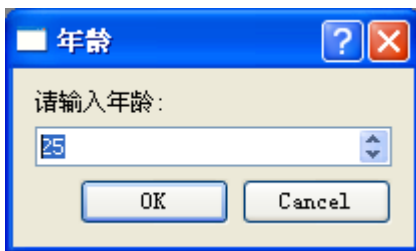
第 4 行调用 `QInputDialog` 的 `getItem()` 函数弹出标准条目选择对话框，`getItem()` 函数原型如下：

```
(QString, bool ok) getItem (QWidget, QString, QString, QStringList, int current  
= 0, bool editable = True, Qt.WindowFlags flags = 0)
```

此函数的第一个参数为标准输入对话框的父窗口，第二个参数为标准输入对话框的标题名，第三个参数为标准输入对话框的标签提示，第四个参数指定标准输入对话框中 `QComboBox` 控件显示的可选条目，为一个 `QStringList` 对象，第五个参数 `current` 为标准条目选择对话框弹出时 `QComboBox` 控件中默认显示的条目序号，第六个参数 `editable` 指定 `QComboBox` 控件中显示的文字是否可编辑，最后一个参数指明标准输入对话框的窗体标识。

第 6 行判断 `ok` 的值，若用户单击了“OK”按钮，则把新输入的性别更新至显示标签。

单击年龄后的修改按钮触发 `slotAge()` 函数，弹出标准 `int` 类型输入对话框，如下图。



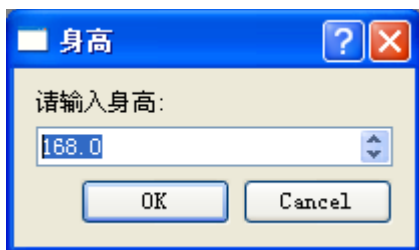
调用 `QInputDialog` 的 `getInteger()` 函数弹出标准 `int` 类型输入对话框，`getInteger()` 函数原型如下：

```
(int, bool ok) getInteger (QWidget, QString, QString, int value = 0, int min =  
-2147483647, int max = 2147483647, int step = 1, Qt.WindowFlags flags = 0)
```

此函数的第一个参数为标准输入对话框的父窗口，第二个参数为标准输入对话框的标题名，第三个参数为标准输入对话框的标签提示，第四个参数 `value` 指定标准输入对话框中 `QSpinBox` 控件默认显示值，第五六个参数指定 `QSpinBox` 控件的数值范围，第七个参数 `step` 指定 `QSpinBox` 控件的步进值。

第 4 行判断 `ok` 的值，若用户单击了“OK”按钮，则把新输入的年龄值更新至显示标签。

单击身高后的修改按钮触发 `slotStature()` 函数，弹出标准 `double` 类型输入对话框，如下图。



调用 `QInputDialog` 的 `getDouble` 函数弹出标准 `double` 类型输入对话框，`getDouble()` 函数原型如下：

```
(float, bool ok) getDouble (QWidget, QString, QString, float value = 0, float  
min = -2147483647, float max = 2147483647, int decimals = 1, Qt.WindowFlags  
flags = 0)
```

此函数的第一个参数为标准输入对话框的父窗窗口，第二个参数为标准输入对话框的标题名，第三个参数为标准输入对话框的标签提示，第四个参数 **value** 指定标准输入对话框中 `QSpinBox` 控件默认显示值，第五六个参数指定 `QSpinBox` 控件的数值范围，第七个参数 **decimals** 指定 `QSpinBox` 控件的步进值。

第 4 行判断 `ok` 的值，若用户单击了“OK”按钮，则把新输入的身高值更新至显示标签。

PyQt4 精彩实例分析 实例 5 各种消息框的使用

在实际的程序开发中，经常会用到各种各样的消息框来给用户一些提示或提醒，Qt 提供了 `QMessageBox` 类来实现此项功能。在本实例中，分析了各种消息框的使用方式及之间的区别。各种消息框的使用如图所示：



实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MessageBoxDlg(QDialog):
    def __init__(self, parent=None):
        super(MessageBoxDlg, self).__init__(parent)
        self.setWindowTitle("MessageBox")
        self.label=QLabel("About Qt MessageBox")
        questionButton=QPushButton("Question")
        informationButton=QPushButton("Information")
        warningButton=QPushButton("Warning")
        criticalButton=QPushButton("Critical")
        aboutButton=QPushButton("About")
        aboutqtButton=QPushButton("About Qt")
        customButton=QPushButton("Custom")

        gridLayout=QGridLayout(self)
        gridLayout.addWidget(self.label, 0, 0, 1, 2)
        gridLayout.addWidget(questionButton, 1, 0)
        gridLayout.addWidget(informationButton, 1, 1)
        gridLayout.addWidget(warningButton, 2, 0)
        gridLayout.addWidget(criticalButton, 2, 1)
        gridLayout.addWidget(aboutButton, 3, 0)
        gridLayout.addWidget(aboutqtButton, 3, 1)
        gridLayout.addWidget(customButton, 4, 0)

        self.connect(questionButton, SIGNAL("clicked()"), self.slotQuestion)
self.connect(informationButton, SIGNAL("clicked()"), self.slotInformation)
        self.connect(warningButton, SIGNAL("clicked()"), self.slotWarning)
        self.connect(criticalButton, SIGNAL("clicked()"), self.slotCritical)
        self.connect(aboutButton, SIGNAL("clicked()"), self.slotAbout)
        self.connect(aboutqtButton, SIGNAL("clicked()"), self.slotAboutQt)
        self.connect(customButton, SIGNAL("clicked()"), self.slotCustom)

    def slotQuestion(self):
```

```

button=QMessageBox.question(self, "Question",
                             self.tr("已到达文档结尾, 是否从头查找?"),
                             QMessageBox.Ok|QMessageBox.Cancel,
                             QMessageBox.Ok)
if button==QMessageBox.Ok:
    self.label.setText("Question button/Ok")
elif button==QMessageBox.Cancel:
    self.label.setText("Question button/Cancel")
else:
    return

def slotInformation(self):
    QMessageBox.information(self, "Information",
                             self.tr("填写任意想告诉于用户的信息!"))
    self.label.setText("Information MessageBox")

def slotWarning(self):
    button=QMessageBox.warning(self, "Warning",
                               self.tr("是否保存对文档的修改?"),
                               QMessageBox.Save|QMessageBox.Discard|
QMessageBox.Cancel,
                               QMessageBox.Save)
    if button==QMessageBox.Save:
        self.label.setText("Warning button/Save")
    elif button==QMessageBox.Discard:
        self.label.setText("Warning button/Discard")
    elif button==QMessageBox.Cancel:
        self.label.setText("Warning button/Cancel")
    else:
        return

def slotCritical(self):
    QMessageBox.critical(self, "Critical",
                         self.tr("提醒用户一个致命的错误!"))
    self.label.setText("Critical MessageBox")

def slotAbout(self):
    QMessageBox.about(self, "About", self.tr("About 事例"))
    self.label.setText("About MessageBox")

def slotAboutQt(self):
    QMessageBox.aboutQt(self, "About Qt")
    self.label.setText("About Qt MessageBox")

def slotCustom(self):
    customMsgBox=QMessageBox(self)
    customMsgBox.setWindowTitle("Custom message box")
    lockButton=customMsgBox.addButton(self.tr("锁定"),
                                       QMessageBox.ActionRole)
    unlockButton=customMsgBox.addButton(self.tr("解锁"),
                                       QMessageBox.ActionRole)
    cancelButton=customMsgBox.addButton("cancel", QMessageBox.ActionRole)

    customMsgBox.setText(self.tr("这是一个自定义消息框!"))
    customMsgBox.exec_()

    button=customMsgBox.clickedButton()
    if button==lockButton:
        self.label.setText("Custom MessageBox/Lock")
    elif button==unlockButton:
        self.label.setText("Custom MessageBox/Unlock")
    elif button==cancelButton:
        self.label.setText("Custom MessageBox/Cancel")

```



```
app=QApplication(sys.argv)
MessageBox=MessageBoxDlg()
MessageBox.show()
app.exec_()
```

本实例主要分析 7 种类型的消息框，包括 **Question** 消息框，**Information** 消息框，**Warning** 消息框，**Critical** 消息框，**About** 消息框，**About Qt** 消息框以及 **Custom** 自定义消息框。

Question 消息框，**Information** 消息框，**Warning** 消息框和 **Critical** 消息框的用法大同小异，这些消息框一般都包含一条提示信息，一个图标以及若干个按钮，它们的作用都是给用户提供一些提醒或一些简单的询问。按图标的不同可区分为以下 4 个级别

Question: 为正常的操作提供一个简单的询问。

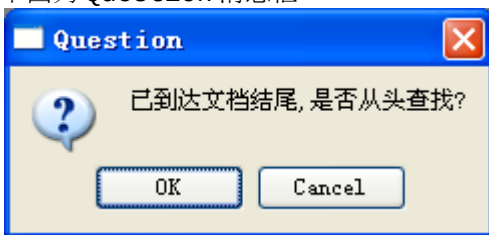
Information: 为正常的操作提供一个提示。

Warning: 提醒用户发生了一个错误。

Critical: 警告用户发生了一个严重错误。

下面分别对各种消息框的使用方法进行分析。

下图为 **Question** 消息框。



关于 **Question** 消息框，调用时直接使用 `QMessageBox.question()` 即可。

第一个参数为消息框的父窗口指针。

第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息，前 3 个参数对于其他几种消息框基本是一样的。

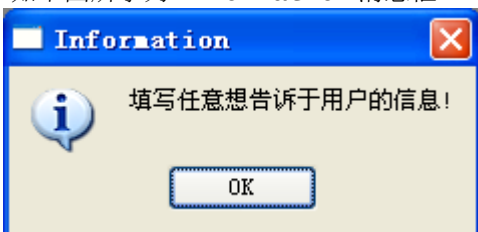
后面两个参数都是对消息框按钮的设定，`QMessageBox` 类提供了许多标准按钮，如 `QMessageBox.Ok`, `QMessageBox.Close`, `QMessageBox.Discard` 等，具体可查询 Qt 帮助。

第四个参数即填写希望在消息框中出现的按钮，可根据需要在标准按钮中选择，用“|”连写，默认为 `QMessageBox.Ok`。

第五个参数为默认按钮，即消息框出现时，焦点默认处于哪个按钮上。

函数的返回值为按下的按钮，当用户按下 **Escape** 键时，相当于返回 `QMessageBox.Cancel`。

如下图所示为 **Information** 消息框。



Information 消息框使用频率最高也最简单，直接调用 `QMessageBox.information()` 即可。

第一个参数为消息框的父窗口指针。

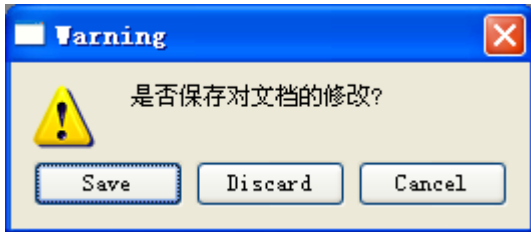
第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息。

后面的两个参数与 **Question** 消息框的用法一样，但在使用的过程中，经常会省略后两个参数，直接使用默认的 `QMessageBox.Ok` 按钮。

Information 消息框和 **Question** 消息框可以通用，使用 **Question** 消息框的地方都可以使用 **Information** 消息框替换。

如下图所示为 **Warning** 消息框。



Warning 消息框的最常用法为当用户进行了一个非正常操作时，提醒用户并询问是否进行某项操作，如关闭文档，提醒并询问用户是否保存对文档的修改。实例中实现的即是此操作。

函数调用的方式与前面 **Question** 消息框的调用方式大致相同。

第一个参数为消息框的父窗口指针。

第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息，

第四个参数为希望在消息框中出现的按钮，可根据需要在标准按钮中选择，用“|”连写，默认为 `QMessageBox.Ok`。

第五个参数为默认按钮，即消息框出现时，焦点默认处于哪个按钮上。

如下图所示为 **Critical** 消息框。



Critical 消息框是在系统出现严重错误时对用户进行提醒的。它的用法也相对简单，通常情况和 **Information** 消息框一样，在调用时只填写前 3 个参数即可。

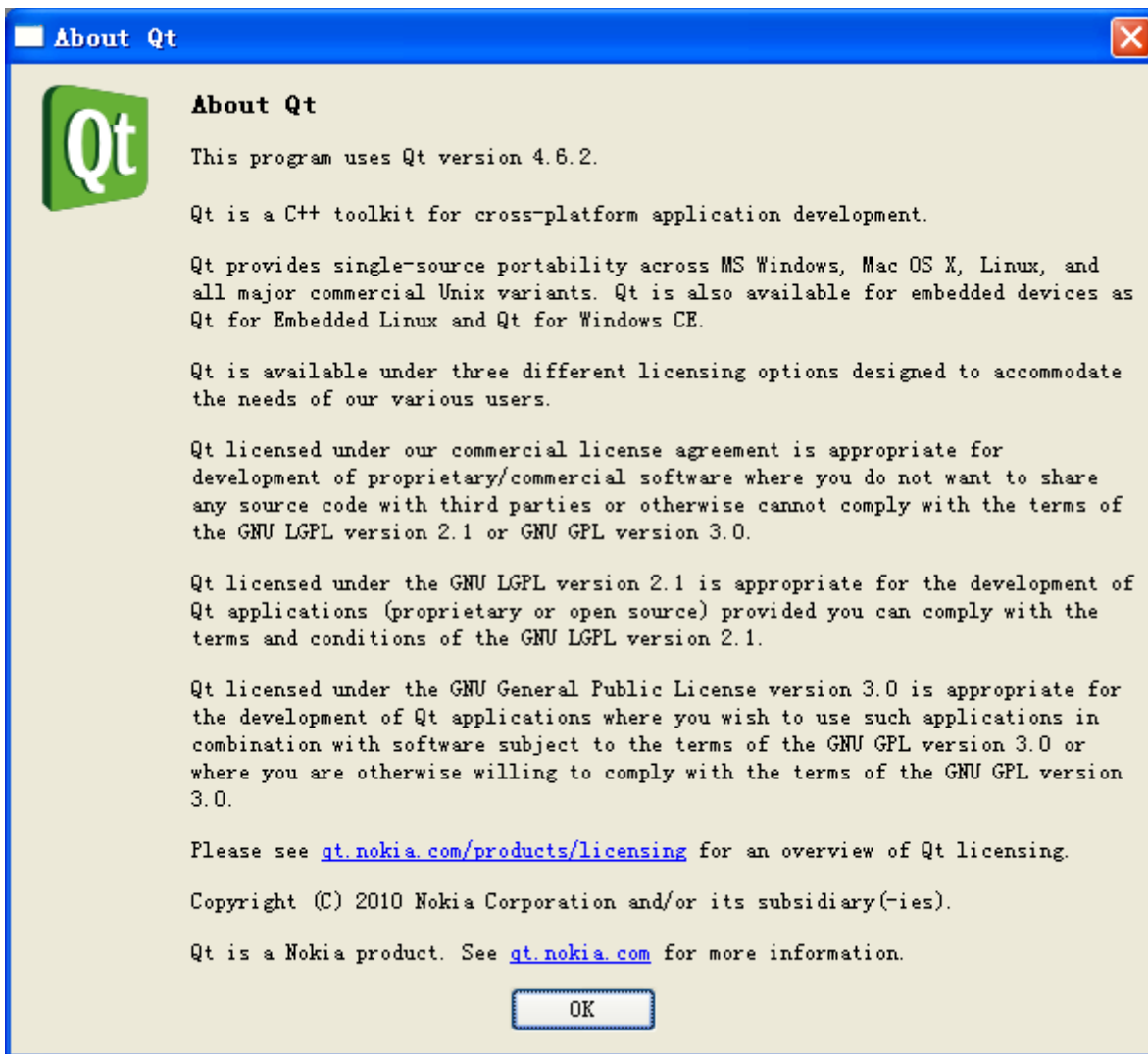
如下图所示为 **About** 消息框。



About 消息框一般用于提供系统的版本等信息。只需提供信息而并不需要用户反馈信息，因此它的用法相对简单，直接调用 `QMessageBox.about()`，并只用指定消息框父窗口，标题栏以及信息的内容即可。

在介绍完以上几种基本消息框的用法后，还有两种特殊的消息框类型，分别是“**About Qt** 消息框”以及自定义消息框。

如下图所示为 **About Qt** 消息框。



“About Qt 消息框”是 Qt 预定好的一种消息框，用于提供 Qt 的相关信息，只需直接调用 `QMessageBox.aboutQt()`，并提定父窗口和标题栏即可，其中显示的内容是 Qt 预定义好的。

最后，当以上所有的消息框都不能满足开发的需求时，Qt 还允许 Custom 自定义消息框。包括消息框的图标，按钮，内容等都可根据需要进行设定。本实例中即实现了一个如下图所示的自定义消息框。



在 `slotCustom()` 函数中，第 84 行首先创建一个 `QMessageBox` 对象 `customMsgBox`。第 85 行设置此消息框的标题栏为 `Custom message box`。

第 86-90 行定义消息框所需的按钮，因此 `QMessageBox` 类提供了一个 `addButton()` 函数来为消息框增加自定义按钮，`addButton()` 函数的第一个参数为按钮显示的文字，第二个参数为按钮类型的描述，具体可查阅 `QMessageBox.ButtonRole`，当然也可使用 `addButton()` 函数来加入一个标准按钮。如第 90 行在消息框中加入了一个 `QMessageBox.Cancel` 按钮。消息框将会按调用 `addButton()` 的先后次序在消息框中由左至右依次插入按钮。

第 92 行调用 `setText` 设置自定义消息框中显示的提示信息内容。

第 93 行调用 `exec()` 显示此自定义消息框。

后面几行代码完成的都是实例中一些显示的功能，此处不再讨论。

通过本实例的分析可见，Qt 提供的消息框类型基本涵盖了开发应用中使用的各种情况，并且提供了自定义消息框的方式，满足各种特殊的需求，在实际应用中关键是分析实际的应用需求，根据不同的应用环境选择最合适的消息框，以使程序简洁而合理。

PyQt4 精彩实例分析 实例 6 实现 QQ 抽屉效果

抽屉效果是软件界面设计中的一种常用形式，目前很多流行软件都采用了抽屉效果，如腾讯公司的 QQ 软件，抽屉效果可以以一种动态直观的方式在有限大小的界面上扩展出更多的功能。本实例在 Qt 下实现抽屉效果，如下图所示。



具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MyQQ(QToolBox):
    def __init__(self, parent=None):
        super(MyQQ, self).__init__(parent)

        toolButton1_1=QToolButton()
        toolButton1_1.setText(self.tr("朽木"))
        toolButton1_1.setIcon(QIcon("image/9.gif"))
        toolButton1_1.setIconSize(QSize(60,60))
        toolButton1_1.setAutoRaise(True)
        toolButton1_1.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

        toolButton1_2=QToolButton()
        toolButton1_2.setText(self.tr("Cindy"))
        toolButton1_2.setIcon(QIcon("image/8.gif"))
        toolButton1_2.setIconSize(QSize(60,60))
        toolButton1_2.setAutoRaise(True)
```

```

toolButton1_2.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton1_3=QToolButton()
toolButton1_3.setText(self.tr("了了"))
toolButton1_3.setIcon(QIcon("image/1.gif"))
toolButton1_3.setIconSize(QSize(60,60))
toolButton1_3.setAutoRaise(True)
toolButton1_3.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton1_4=QToolButton()
toolButton1_4.setText(self.tr("张三虎"))
toolButton1_4.setIcon(QIcon("image/3.gif"))
toolButton1_4.setIconSize(QSize(60,60))
toolButton1_4.setAutoRaise(True)
toolButton1_4.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton1_5=QToolButton()
toolButton1_5.setText(self.tr("CSDN"))
toolButton1_5.setIcon(QIcon("image/4.gif"))
toolButton1_5.setIconSize(QSize(60,60))
toolButton1_5.setAutoRaise(True)
toolButton1_5.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton2_1=QToolButton()
toolButton2_1.setText(self.tr("天的另一边"))
toolButton2_1.setIcon(QIcon("image/5.gif"))
toolButton2_1.setIconSize(QSize(60,60))
toolButton2_1.setAutoRaise(True)
toolButton2_1.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton2_2=QToolButton()
toolButton2_2.setText(self.tr("蓝绿不分"))
toolButton2_2.setIcon(QIcon("image/6.gif"))
toolButton2_2.setIconSize(QSize(60,60))
toolButton2_2.setAutoRaise(True)
toolButton2_2.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton3_1=QToolButton()
toolButton3_1.setText(self.tr("老牛"))
toolButton3_1.setIcon(QIcon("image/7.gif"))
toolButton3_1.setIconSize(QSize(60,60))
toolButton3_1.setAutoRaise(True)
toolButton3_1.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

toolButton3_2=QToolButton()
toolButton3_2.setText(self.tr("张三疯"))
toolButton3_2.setIcon(QIcon("image/8.gif"))
toolButton3_2.setIconSize(QSize(60,60))
toolButton3_2.setAutoRaise(True)
toolButton3_2.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

groupbox1=QGroupBox()
vlayout1=QVBoxLayout(groupbox1)
vlayout1.setMargin(10)
vlayout1.setAlignment(Qt.AlignCenter)
vlayout1.addWidget(toolButton1_1)
vlayout1.addWidget(toolButton1_2)
vlayout1.addWidget(toolButton1_3)
vlayout1.addWidget(toolButton1_4)
vlayout1.addWidget(toolButton1_5)
vlayout1.addStretch()

groupbox2=QGroupBox()

```

```

        vlayout2=QVBoxLayout(groupbox2)
        vlayout2.setMargin(10)
        vlayout2.setAlignment(Qt.AlignCenter)
        vlayout2.addWidget(toolButton2_1)
        vlayout2.addWidget(toolButton2_2)

        groupbox3=QGroupBox()
        vlayout3=QVBoxLayout(groupbox3)
        vlayout3.setMargin(10)
        vlayout3.setAlignment(Qt.AlignCenter)
        vlayout3.addWidget(toolButton3_1)
        vlayout3.addWidget(toolButton3_2)

        self.addItem(groupbox1,self.tr("我的好友"))
        self.addItem(groupbox2,self.tr("同事"))
        self.addItem(groupbox3,self.tr("黑名单"))

    app=QApplication(sys.argv)
    myqq=MyQQ()
    myqq.setWindowTitle("My QQ")
    myqq.show()
    app.exec_()

```

MyQQ 类继承自 QToolBox，QToolBox 提供了一种列状的层叠窗体，本实例通过 QToolBox 来实现一种抽屉效果，QToolButton 提供了一种快速访问命令或选择项的按钮，通常在工具条中使用。

第 75 行创建了一个 QGroupBox 类实例，在本例中对应每一个抽屉。

第 12 行创建了一个 QToolButton 类实例，在这里 QToolButton 分别对应于抽屉中的每一个按钮。

第 13-15 行对按钮的文字，图标以及大小等进行设置。

第 16 行设置按钮的 AutoRaise 属性为 True，即当鼠标离开时，按钮自动恢复成弹起状态。

第 17 行设置按钮的 ToolButtonStyle 属性，ToolButtonStyle 属性主要用来描述按钮的文字和图标的显示方式。Qt 定义了 4 种 QToolButtonStyle 类型，分别介绍如下。

Qt.ToolButtonIconOnly: 只显示图标。

Qt.ToolButtonTextOnly: 只显示文字。

Qt.ToolButtonTextBesideIcon: 文字显示在图标旁边。

Qt.ToolButtonTextUnderIcon: 文字显示在图标下面。

程序员可以根据显示需要调整显示方式。

第 76 行创建一个 QVBoxLayout 类实例，用来设置抽屉内各按钮的布局。

第 77，78 行设置布局中各按钮的显示间距和显示位置。

第 79-83 行将抽屉内的各个按钮加入。

第 84 行调用 addStretch() 方法在按钮之后插入一个占位符，使得所有按钮能靠上对齐。并且在整个抽屉大小发生改变时，保证按钮的大小不发生变化。

其它行都是实现类似的功能。

第 100-102 行把准备好的抽屉插入至 QToolBox 中。

PyQt4 精彩实例分析 实例 7 表格的使用

制作统计软件时经常会使用表格将资料列出，或是通过表格进行资料的设置，在 Qt 中可以使用 `QTableWidget` 实现一个表格。本实例演示如何使用表格，并在表格中嵌入控件。如下图所示为“表格的使用”对话框。



`QTableWidget` 类提供了一个灵活的和可编辑的表格控件，包含很多 API，可以处理标题，行列，单元格和选中区域，`QTableWidget` 可以嵌入编辑框或显示控件，并可通过拖动控制柄调节各单元格的大小。表格的每一项可以定义成不同的属性，可以显示文本，也可以插入控件，这样给表格的使用带来了很好的扩展性。

本实例的实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MyTable(QTableWidget):
    def __init__(self, parent=None):
        super(MyTable, self).__init__(parent)
        self.setColumnCount(5)
        self.setRowCount(2)
        self.setItem(0, 0, QTableWidgetItem(self.tr("性别")))
        self.setItem(0, 1, QTableWidgetItem(self.tr("姓名")))
        self.setItem(0, 2, QTableWidgetItem(self.tr("出生日期")))
        self.setItem(0, 3, QTableWidgetItem(self.tr("职业")))
        self.setItem(0, 4, QTableWidgetItem(self.tr("收入")))
        lbp1=QLabel()
        lbp1.setPixmap(QPixmap("image/4.gif"))
        self.setCellWidget(1, 0, lbp1)
        twi1=QTableWidgetItem("Tom")
        self.setItem(1, 1, twi1)
        dte1=QDateTimeEdit()
        dte1.setDateTime(QDateTime.currentDateTime())
        dte1.setDisplayFormat("yyyy/mm/dd")
        dte1.setCalendarPopup(True)
        self.setCellWidget(1, 2, dte1)
        cbw=QComboBox()
        cbw.addItem("Worker")
        cbw.addItem("Famer")
        cbw.addItem("Doctor")
        cbw.addItem("Lawyer")
```

```
        cbw.addItem("Soldier")
        self.setCellWidget(1,3,cbw)
        sb1=QSpinBox()
        sb1.setRange(1000,10000)
        self.setCellWidget(1,4,sb1)
```

```
app=QApplication(sys.argv)
myqq=MyTable()
myqq.setWindowTitle("My Table")
myqq.show()
app.exec_()
```

第 18-20 行在表格中插入一个 QLabel 控件，并设置 QLabel 的图形属性。

第 21-22 行设置表格单元的属性为文本显示。

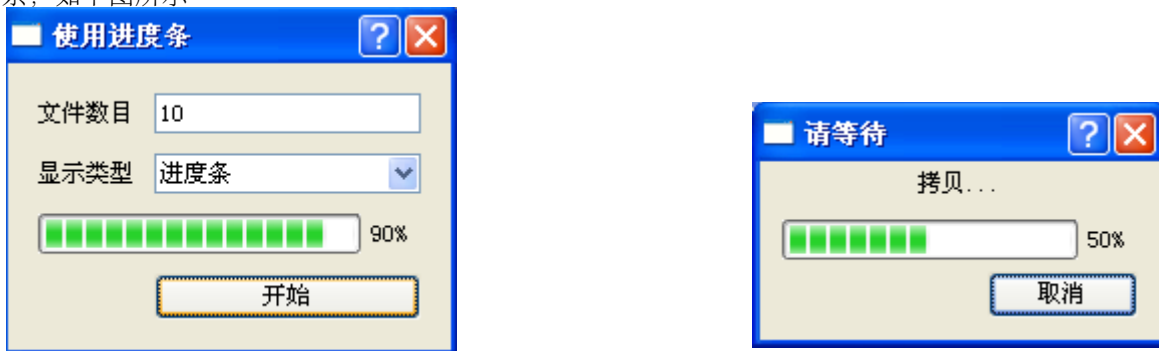
第 23-27 行在表格中插入一个 QDateTimeEdit 控件，该控件可以编辑日期时间，setCalendarPopup() 方法设置是否弹出日历编辑器。

第 28-34 行在表格中插入一个 QComboBox 控件，调用 QTableWidgetItem 的 setCellWidget() 函数可在某个指定的表格单元格中插入一个控件，函数的前两个参数用于指定单元格的行，列号。

第 35-37 行在表格中插入一个 QSpinBox 控件。

PyQt4 精彩实例分析 实例 8 使用进度条

通常在处理长时间任务时需要提供进度条的显示，告诉用户当前任务的进展情况。本实例演示如何使用进度条，如下图所示。



Qt 提供了两种显示进度条的方式，一种是 `QProgressBar`，另一种是

`QProgressDialog`。`QProgressBar` 类提供了种横向或纵向显示进度条的控件表示方式，用来描述任务的完成情况。`QProgressDialog` 类提供了一种针对慢速过程的进度对话框表示方式，用于描述任务完成的进度情况。标准的进度条对话框包括一个进度显示条，一个取消按钮以及一个标签。

`QProgressBar` 有几个重要的属性值，`minimum`、`maximum` 决定进度条提示的最小值和最大值，`format` 决定进度条显示文字的格式，可以有 3 种显示格式：`%p%`、`%v`、`%m`。`%p%` 显示完成的百分比，这是默认显示方式；`%v` 显示当前的进度值；`%m` 显示总的步进值。`invertedAppearance` 属性可以让进度条以反方向显示进度。

`QProgressDialog` 也有几个重要的属性值，决定了进度条对话框何时出现，出现多长时间，分别是 `minimum`、`maximum` 和 `minimumDuration`。`minimum` 和 `maximum` 分别表示进度条的最小值和最大值，决定了进度条的变化范围，`minimumDuration` 为进度条对话框出现前的等待时间。系统根据所需完成的工作量估算一个预计花费的时间，若大于设定的等待时间 `minimumDuration`，则出现进度条对话框；若小于设定的等待时间，则不出现进度条对话框。

进度条使用了一个步进值的概念，即一时设置好进度条的最大值和最小值，进度条将会显示完成的步进值占总的步进值的百分比，百分比的计算公式为：

百分比 = $(\text{value}() - \text{minimum}()) / (\text{maximum}() - \text{minimum}())$

本例具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class Progress(QDialog):
    def __init__(self, parent=None):
        super(Progress, self).__init__(parent)
        self.setWindowTitle(self.tr("使用进度条"))
        numLabel=QLabel(self.tr("文件数目"))
        self.numLineEdit=QLineEdit("10")
        typeLabel=QLabel(self.tr("显示类型"))
        self.typeComboBox=QComboBox()
        self.typeComboBox.addItem(self.tr("进度条"))
        self.typeComboBox.addItem(self.tr("进度对话框"))

        self.progressBar=QProgressBar()
```

```

startPushButton=QPushButton(self.tr("开始"))

layout=QGridLayout()
layout.addWidget(numLabel,0,0)
layout.addWidget(self.numLineEdit,0,1)
layout.addWidget(typeLabel,1,0)
layout.addWidget(self.typeComboBox,1,1)
layout.addWidget(self.progressBar,2,0,1,2)
layout.addWidget(startPushButton,3,1)
layout.setMargin(15)
layout.setSpacing(10)

self.setLayout(layout)

self.connect(startPushButton,SIGNAL("clicked()"),self.slotStart)

def slotStart(self):
    num=int(self.numLineEdit.text())

    if self.typeComboBox.currentIndex()==0:
        self.progressBar.setMinimum(0)
        self.progressBar.setMaximum(num)

        for i in range(num):
            self.progressBar.setValue(i)
            QThread.sleep(100)

    elif self.typeComboBox.currentIndex()==1:
        progressDialog=QProgressDialog(self)
        progressDialog.setWindowModality(Qt.WindowModal)
        progressDialog.setMinimumDuration(5)
        progressDialog.setWindowTitle(self.tr("请等待"))
        progressDialog.setLabelText(self.tr("拷贝..."))
        progressDialog.setCancelButtonText(self.tr("取消"))
        progressDialog.setRange(0,num)

        for i in range(num):
            progressDialog.setValue(i)
            QThread.sleep(100)
            if progressDialog.wasCanceled():
                return

    app=QApplication(sys.argv)
    progress=Progress()
    progress.show()
    app.exec_()

```

第 38 行获得当前需要复制的文件数目，这里对应进度条的总的步进值。

第 40-46 行采用进度条的方式显示进度。

第 41，42 行设置进度条的步进范围从 0 到需要复制的文件数目。

第 45，46 行模拟每一个文件的复制过程，这里通过 `QThread.sleep(100)` 来模拟，在实际中使用文件复制过程来替换，进度条的总的步进值为需要复制的文件数目，当复制完成一个文件后，步进值增加 1。

第 48-61 行采用进度对话框的方式显示进度。

第 49 行创建一个进度对话框。

第 50 行设置进度对话框采用模态方式进行显示，即显示进度的同时，其他窗口将不响应输入信号。

第 51 行设置进度对话框出现等待时间，此处设定为 5 秒，默认为 4 秒。

第 52-54 行设置进度对话框的窗体标题，显示文字信息以及取消按钮的显示文字。

第 55 行设置进度对话框的步进范围。

第 57-61 行模拟每一个文件复制过程，这里通过 `QThread::msleep(100)` 进行模拟，在实际中使用文件复制过程来替换，进度条的总的步进值为需要复制的文件数目，当复制完一个文件后，步进值增加 1，这里需要使用 `processEvents()` 来正常响应事件循环，以确保应用程序不会出现阻塞。

第 60，61 行检测“取消”按钮是否被触发，若触发则退出循环并关闭进度对话框，在实际应用中，此处还需进行相关的清理工作。

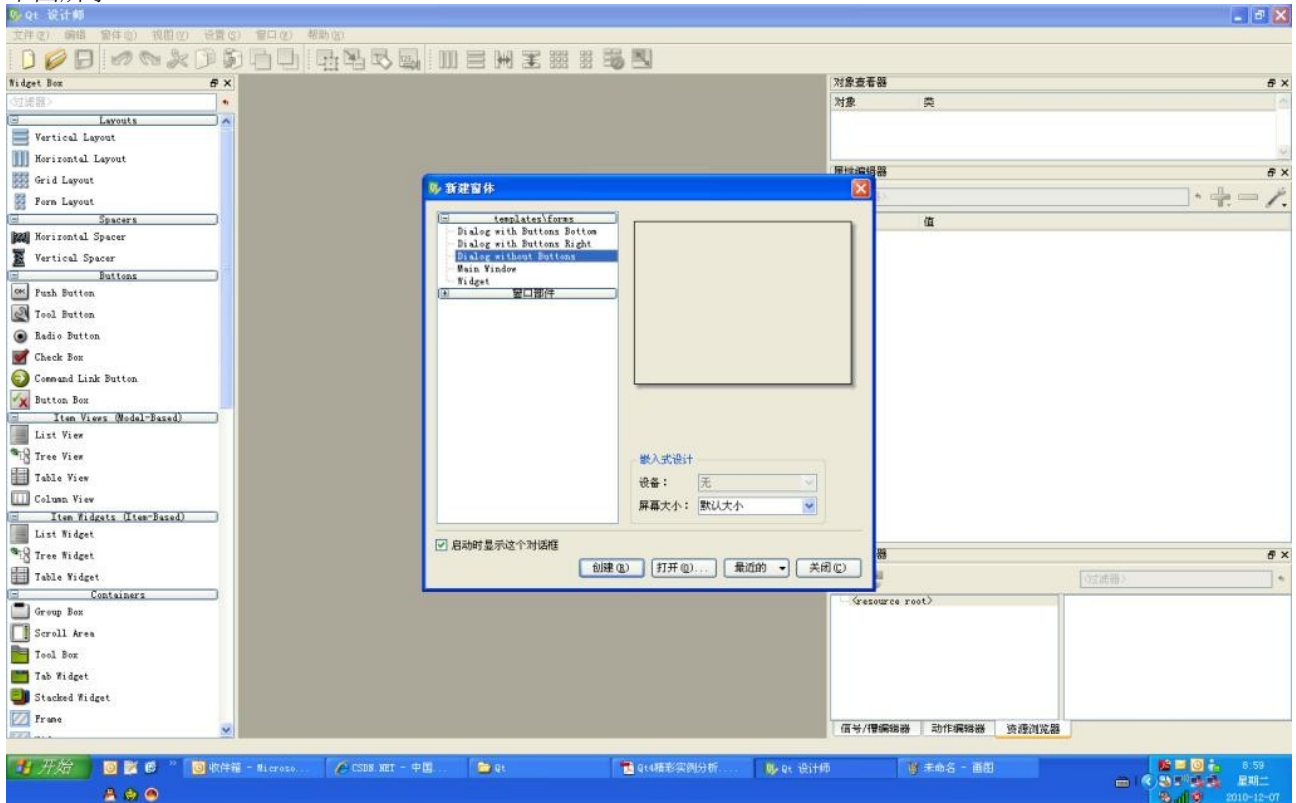
进度对话框的使用有两种方法，即模态方式与非模态方式。本实例中使用的是模态方式，模态方式的使用比较简单方便，但必须使用 `processEvents` 来使事件循环保持正常进行状态，从而确保应用不会阻塞。若使用非模态方式，则需要通过 `QTime` 来实现定时设置进度条的值。

PyQt4 精彩实例分析 实例 9 利用 Qt Designer 设计一个对话框

在 Qt 编程中，程序员通常都是使用手动编写 Python 源代码来进行 Qt 程序开发，但有些程序员也喜欢使用可视化的方法进行对话框设计，因此，Qt 为习惯利用可视化方式进行窗口程序设计的程序员提供了 **Designer**，它可以给一个应用程序提供全部或者部分对话框。用 Qt Designer 设计的对话框和用 Python 写代码写成的对话框是一样的，可以用作一个常用的工具，并不对编辑产生影响。使用 Qt Designer 可以方便快速地对对话框进行修改，在对话框经常需要变化的情况下，这是一种很好的方式。使用 Qt Designer 设计对话框一般都有如下几个步骤：

- 1) 创建窗体并在窗体中放置各种控件。
- 2) 对窗体进行布局设计。
- 3) 设置各控件的标签顺序。
- 4) 创建信号和槽。
- 5) 连接信号和槽。

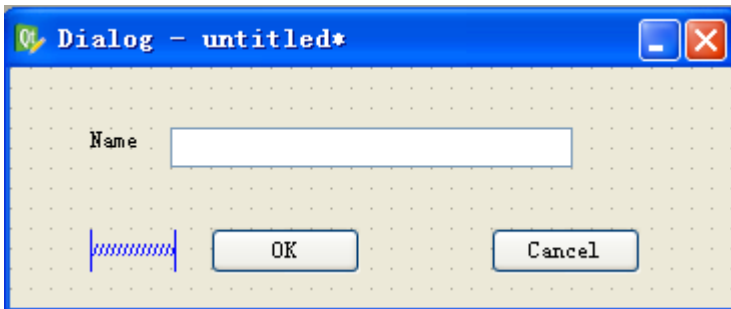
Qt Designer 的启动可以通过命令行运行 **designer** 完成，或 Windows 下的开始菜单完成，启动后界面如下图所示。



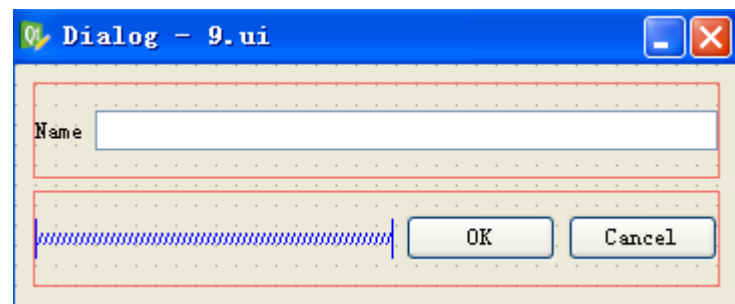
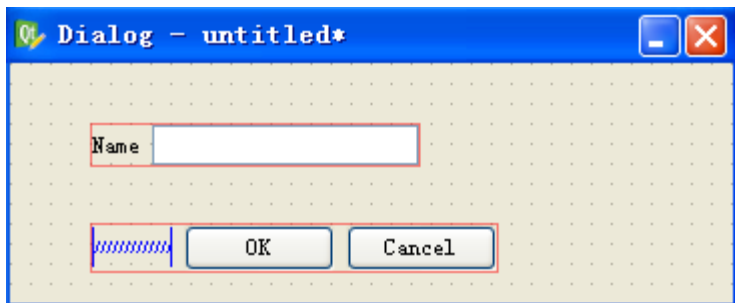
Qt Designer 提供如下 5 种表单模板可供选择：

- 1) 底部带“确定”，“取消”按钮的对话框窗体。
- 2) 右侧带“确定”，“取消”按钮的对话框窗体。
- 3) 不带按钮的对话框窗体。
- 4) Main Window 类型窗体。
- 5) 通用窗体。

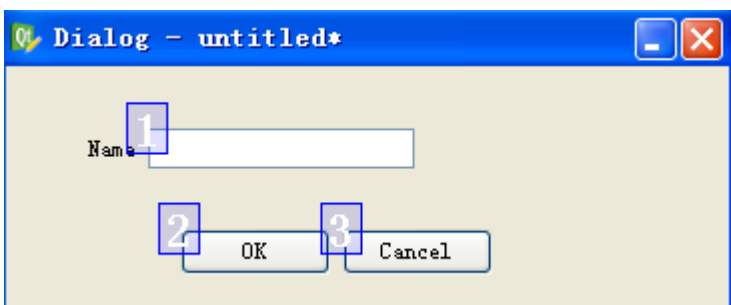
这里选择创建一种不带按钮的对话框窗体，接下来需要做的就是窗体中放置各种需要的控件，Qt Designer 的设计空间列出了所有控件以及各控件的属性设置窗体。在窗体中放置一个 Label 和 LineEdit，两个 PushButton 和一个 Horizontal Spacer 控件，并设置种控件的 text 属性，如下图所示。在开始向窗体中放置控件时，不用太在意控件对齐与否，只用放置大概位置即可。



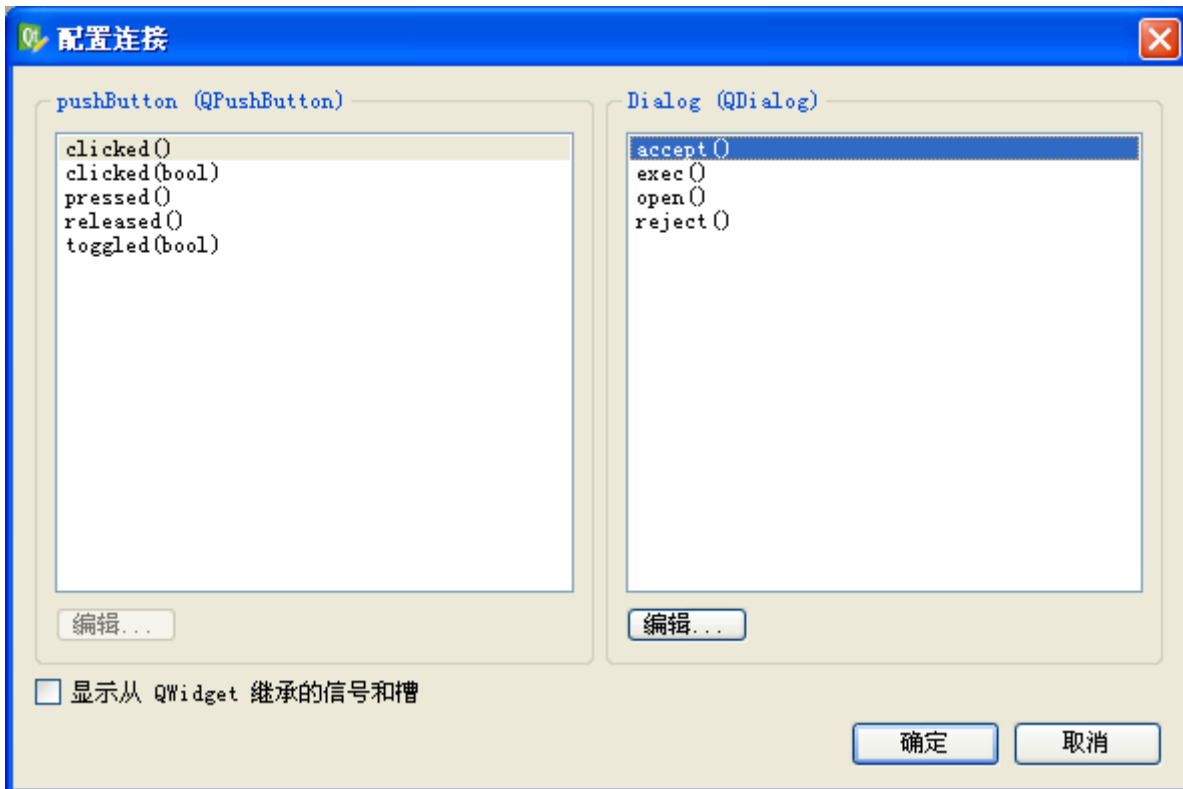
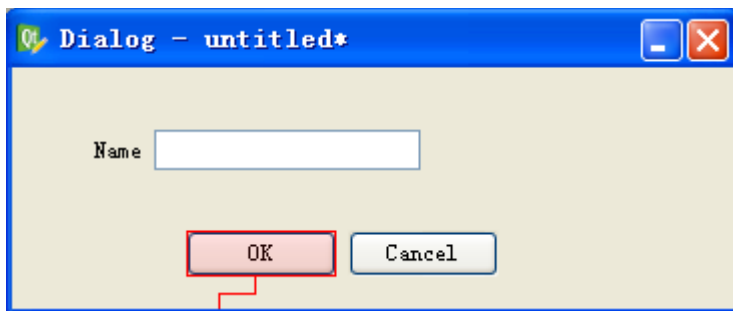
接下来对窗体的各个控件进行布局设计，选择位于同一行的所有控件，选择 Qt Designer 菜单中的“窗体”-->“水平布局”或右击选择“布局”-->“水平布局”，然后选择两个水平布局，右击选择“布局”-->“栅格布局”，完成所选控件的水平布局。完成布局设计后适当调整整个窗体的大小，以适合控件的大小，如下图所示。



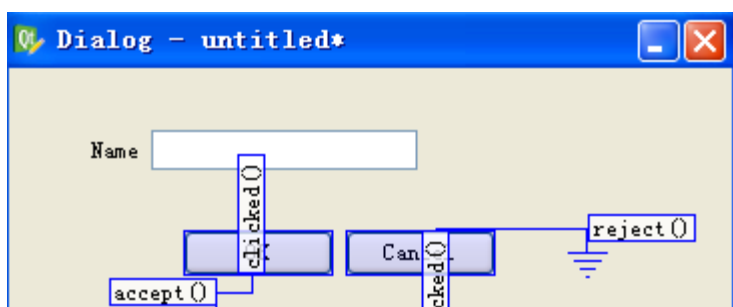
然后对各控件的标签顺序进行设置，选择 Qt Designer 菜单中的“编辑”-->“编辑 Tab 顺序”，进入标签设置模式，窗体中各个控件上出现一个蓝色的小框，框内的数字表示该控件的标签顺序，即焦点顺序，如下图所示，可以单击蓝色小框修改标签顺序。完成标签顺序设置后选择 Qt Designer 菜单中的“编辑”-->“编辑窗口部件”离开标签设置模式。



接下来进行信号和槽的连接，选择 Qt Designer 菜单中的“编辑”-->“编辑信号/槽”，进入信号/槽连接模式，如下图所示。此时单击 OK 按钮，然后拖动鼠标，可以发现有一根红色的类似接地线的标志线被拖出，松开鼠标，弹出信号/槽的连接配置窗口，如下图所示。



连接配置窗口左侧列出了按钮 OK 的所有信号，右侧列出了对话框的所有槽，选择 OK 按钮的 `clicked()` 信号和右侧的 `accept()`，单击“确定”按钮，此时完成按钮 OK 的信号/槽的连接，用同样的方式配置按钮 Cancel 的信号/槽，如下图所示。



至此，关于 Qt Designer 的操作就结束了，生成一个 .ui 文件，保存为 9.ui。

然后使用 `pyuic4` 命令生成一个 py 文件，如下图所示。

```
C:\WINDOWS\system32\cmd.exe
驱动器 F 中的卷没有标签。
卷的序列号是 2045-FCDB

F:\My Doc\Python\Qt 的目录

2010-12-07 09:22 <DIR> .
2010-12-07 09:22 <DIR> ..
2010-12-03 11:04      206 1.py
2010-12-03 09:59    1,878 2.py
2010-12-02 20:06    3,321 3.py
2010-12-02 20:44    3,249 4.py
2010-12-04 18:45    4,516 5.py
2010-12-04 22:58    4,160 6.py
2010-12-05 16:21    1,454 7.py
2010-12-06 22:38    2,331 8.py
2010-12-07 09:22    2,553 9.ui
2010-12-04 21:59 <DIR> image
2010-12-02 13:07 43,306,240 Qt4精彩实例分析.pdf
2010-12-02 12:51 30,468,944 精通Qt4编程.pdf
      11 个文件      73,798,852 字节
       3 个目录 17,280,000,000 可用字节

F:\My Doc\Python\Qt>pyuic4 -o ui_9.py 9.ui

F:\My Doc\Python\Qt>
```

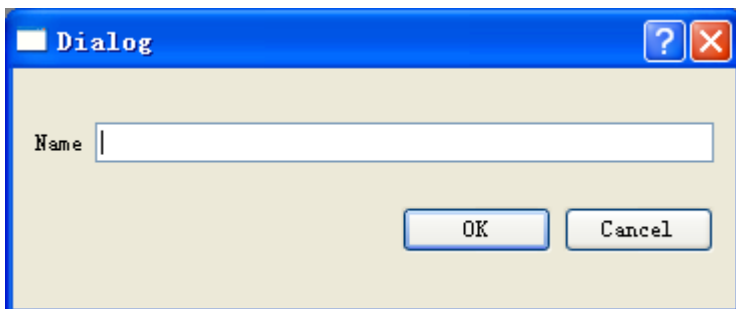
接下来编写一个 9.py 主文件来运行这个程序。具体实现代码如下：

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
import ui_9

class TestDialog(QDialog, ui_9.Ui_Dialog):
    def __init__(self, parent=None):
        super(TestDialog, self).__init__(parent)
        self.setupUi(self)

app=QApplication(sys.argv)
dialog=TestDialog()
dialog.show()
app.exec_()
```

运行后分别单击 OK 和 Cancel 按钮，执行对话框的 accept() 和 reject() 函数，如下图所示。



使用 **Qt Designer** 设计对话框是一种简单有效的方法，可以节省设计对话框的时间，而且修改方便，直观，对于初学者来说，这是一种入门的好方法。但随着程序越来越复杂，**Qt Designer** 也有不利的地方：首先，使用 **Qt Designer** 生成的代码比较庞大，很多代码是自动生成的，不利于开发者阅读，其次对于初学者而言，使用 **Qt Designer** 不利于掌握 **Qt** 编程的本质（类似于 **.NET**）。因此，笔者还是建议昼使用手动的方式编写源代码，这样能更好地理解 **Qt** 编程的本质，更多地体验 **Qt** 编程的乐趣。本书的绝大部分实例都是采用手动编写代码的方式进行实现的。

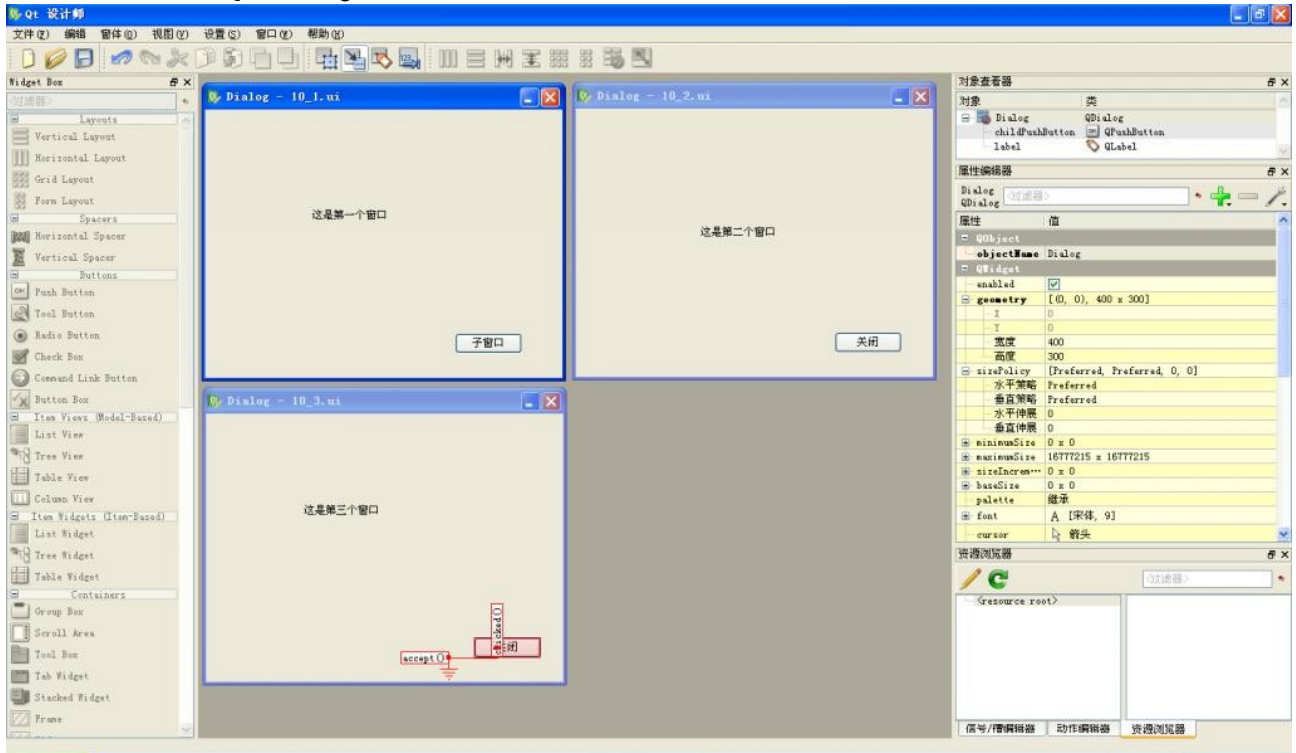
PyQt4 精彩实例分析 实例 10 在程序中使用 Ui

本实例使用一个简单的例子说明如何在程序开发中使用 Designer 生成 .ui 文件。本实例利用 Qt Designer 生成了 3 个简单的 ui，在使用时，两个 ui 插入到主程序的 `QTabWidget` 中，另一个 ui 由按钮触发弹出，如下图所示。主程序窗口 `TestDialog` 采用的是手动编写代码的实现方式。

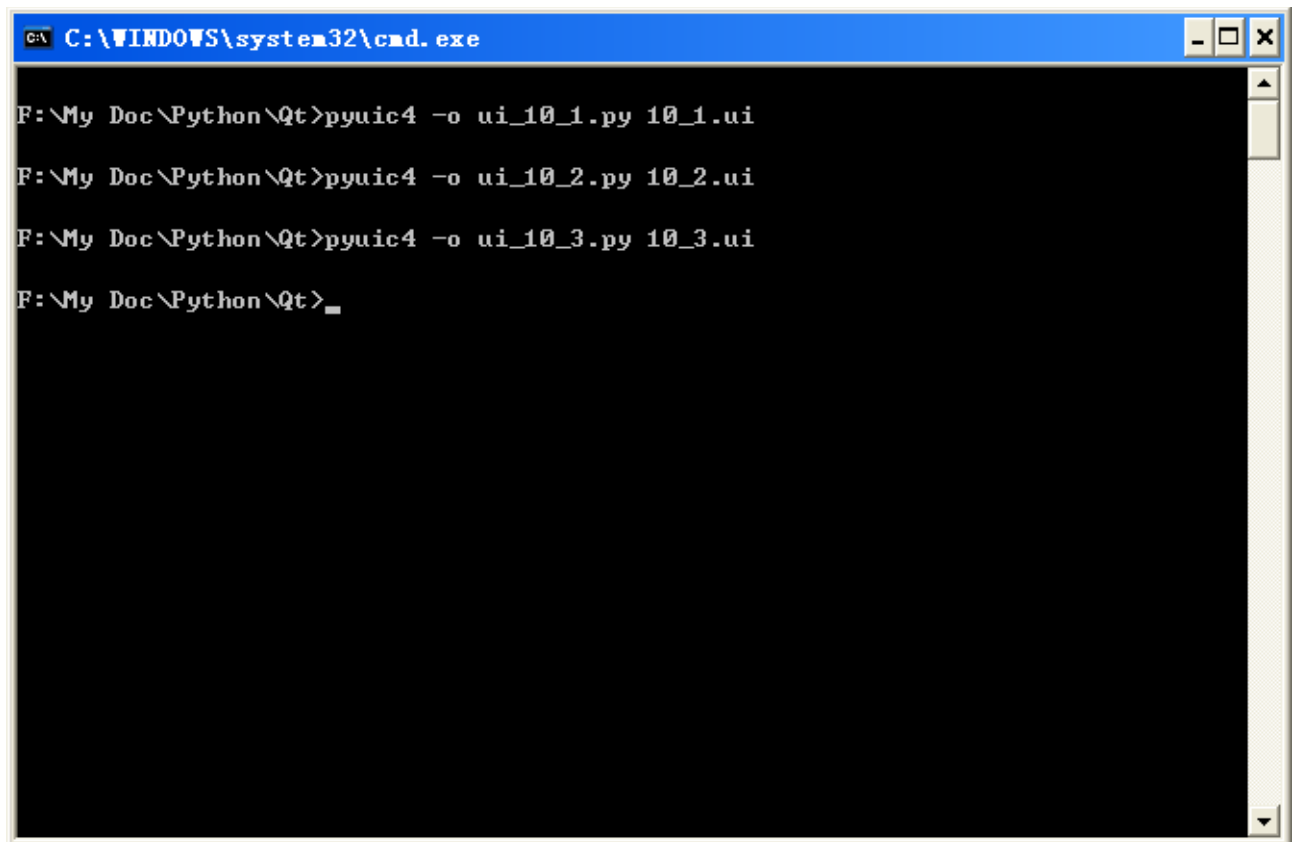


利用 Qt Designer 设计生成的 .ui 文件，在使用时可利用 Qt 自带的工具 `pyuic4` 生成 `ui_xxx.py` 文件进行使用。

下图是本实例使用 Qt Designer 设计的 3 个 ui 文件。



然后使用 pyuic4 对这三个 ui 文件进行转换，如下图所示。



下面是本实例的代码实现。

```
from PyQt4.QtGui import *  
from PyQt4.QtCore import *
```

```

import ui_10_1, ui_10_2, ui_10_3
import sys

class TestDialog(QDialog):
    def __init__(self, parent=None):
        super(TestDialog, self).__init__(parent)

        firstUi=ui_10_1.Ui_Dialog()
        secondUi=ui_10_2.Ui_Dialog()
        self.thirdUi=ui_10_3.Ui_Dialog()

        tabWidget=QTabWidget(self)
        w1=QWidget()
        firstUi.setupUi(w1)
        w2=QWidget()
        secondUi.setupUi(w2)

        tabWidget.addTab(w1, "First")
        tabWidget.addTab(w2, "Second")
        tabWidget.resize(380, 380)

self.connect(firstUi.childPushButton, SIGNAL("clicked()"), self.slotChild)

self.connect(secondUi.closePushButton, SIGNAL("clicked()"), self, SLOT("reject()"))

    def slotChild(self):
        dlg=QDialog()
        self.thirdUi.setupUi(dlg)
        dlg.exec_()

app=QApplication(sys.argv)
dialog=TestDialog()
dialog.show()
app.exec_()

```

import ui_10_1, ui_10_2, ui_10_3 这三个文件是通过 pyuic4 工具根据相应的 ui 文件生成的。

slotChild()槽函数用于响应弹出子窗口的按钮事件。

主程序中声明了 3 个变量，firstUi, secondUi, thirdUi 分别对应 3 个 ui。

第 14 行首先创建一个 QTabWidget 对象。

第 15, 16 行创建第一个 ui，首先新建一个 QWidget 对象，以此 QWidget 对象为参数调用第一个 ui 的 setupUi()函数，生成第一个 ui 页面。

第 17, 18 行以同样的方式创建第二个 ui 画面。

第 20-22 行在 QTabWidget 对象中插入两个准备好的 ui 页面。

第 24 行连接第一个 ui 页面上的 childPushButton 的 clicked()信号与 slotChild()槽函数。

第 25 行连接第二个 ui 页面上 closePushButton 的 clicked()信号与 reject()槽函数，关闭主窗口程序。

实现弹出对话框的槽函数 slotChild()中，首先新建一个 QDialog 对象，以此 QDialog 对象为参数调用第三个 ui 对象的 setupUi()函数，最后调用 exec()显示此对话框。

PyQt4 精彩实例分析 实例 11 动态加载 Ui

Qt 提供了一个 `uic` 模块，包括了与 `ui` 相关的函数，如 `loadUi()`，可使程序在运行中动态加载 Designer 设计生成的 `.ui` 文件，本实例即利用 `uic.loadUi()` 实现实例 10 中的弹出窗口部分。

实现动态加载 `ui`，首先需要在程序中导入 `uic`。

```
from PyQt4 import uic
```

然后修改 `slotChild()` 槽函数的实现代码如下：

```
def slotChild(self):  
    dlg=uic.loadUi("10_3.ui")  
    dlg.exec_()
```

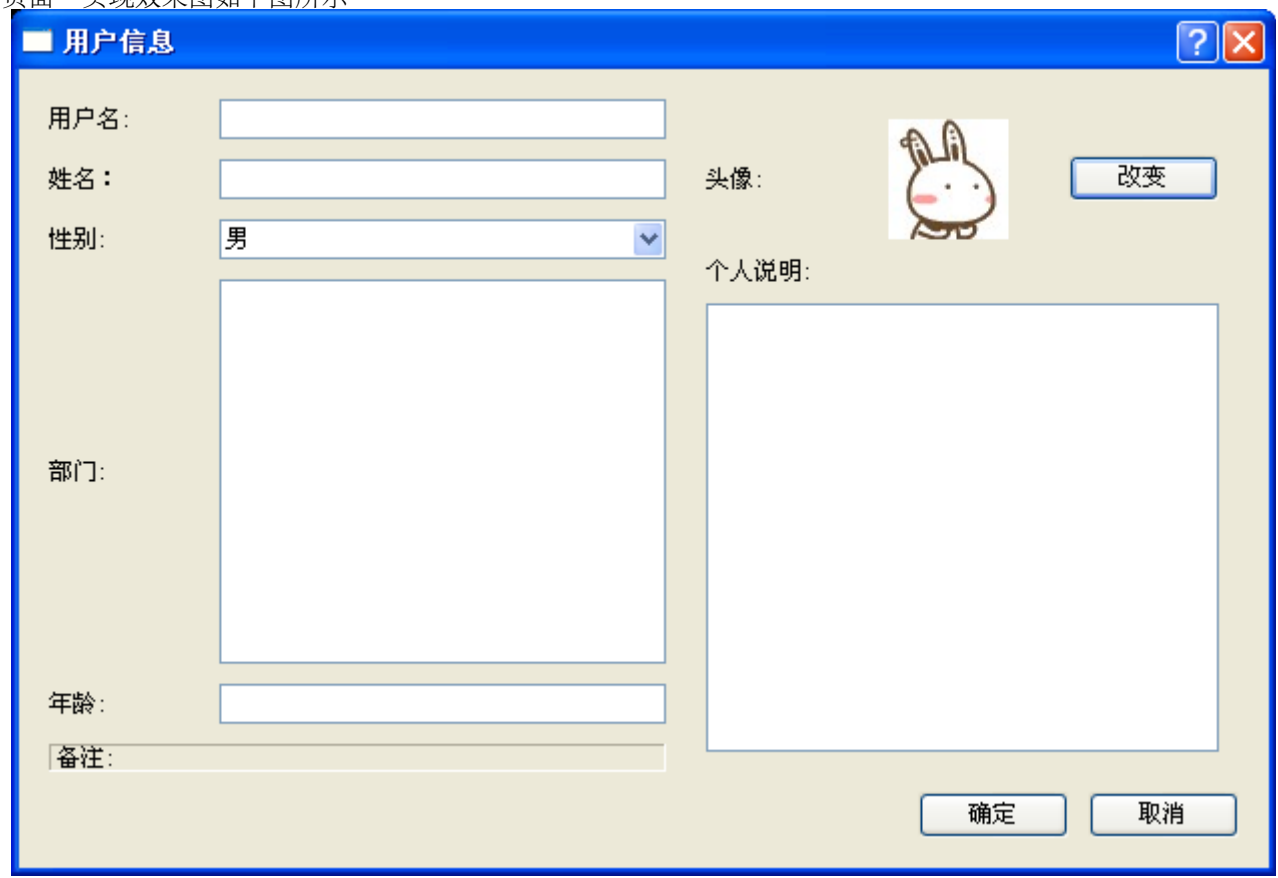
第一行调用 `uic` 的 `loadUi()` 函数根据 `ui` 文件生成一个相应的 `QDialog` 对象，并将此对象返回。

调用 `dlg.exec_()` 显示此子窗口。

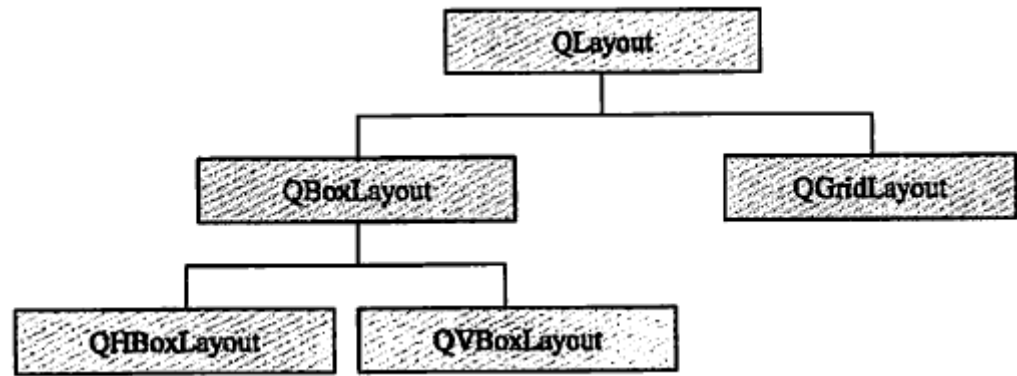
这种动态加载的方式不用生成 `ui_10_3.py` 文件，在程序运行时才会被加载。采用这种方式最大的好处是可以在不重新生成 `ui_xxx.py` 文件的情况下，改变窗口的布局。但也存在不方便的地方，即在主程序中对子窗口的控件进行操作比较复杂。

PyQt4 精彩实例分析 实例 12 基本布局管理

本实例利用基本布局管理(QHBoxLayout, QVBoxLayout, QGridLayout)实现一个类似 QQ 的用户资料修改页面。实现效果图如下图所示。

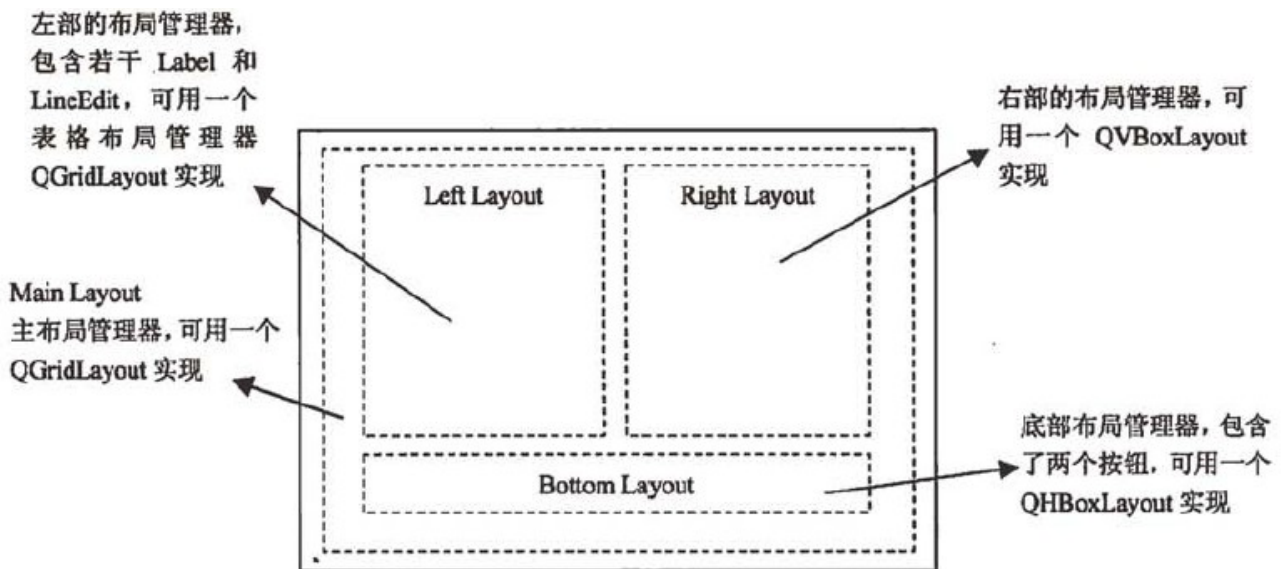


Qt 提供的布局类以及它们之间的继承关系如下图所示。



常用到的布局类有 QHBoxLayout, QVBoxLayout, QGridLayout 3 种，分别水平排列布局，垂直排列布局和表格排列布局。Qt3 中的 QHBox 和 QVBox 到 Qt4 以后被废弃。布局中最常用的方法有 addWidget() 和 addLayout(), addWidget() 方法用于在布局中插入控件，addLayout() 用于在布局中插入子布局。

下面通过实例的实现过程了解布局管理的使用方法。首先通过一个示意图了解此对话框的布局结构，如下图所示。



从上图中可知，本实例共用到 4 个布局管理器，分别是 LeftLayout, RightLayout, BottomLayout 和 MainLayout。

下面是具体的实现。

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class LayoutDialog(QDialog):
    def __init__(self, parent=None):
        super(LayoutDialog, self).__init__(parent)
        self.setWindowTitle(self.tr("用户信息"))

        label1=QLabel(self.tr("用户名:"))
        label2=QLabel(self.tr("姓名: "))
        label3=QLabel(self.tr("性别:"))
        label4=QLabel(self.tr("部门:"))
        label5=QLabel(self.tr("年龄:"))
        otherLabel=QLabel(self.tr("备注:"))
        otherLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
        userLineEdit=QLineEdit()
        nameLineEdit=QLineEdit()
        sexComboBox=QComboBox()
        sexComboBox.insertItem(0, self.tr("男"))
        sexComboBox.insertItem(1, self.tr("女"))
        departmentTextEdit=QTextEdit()
        ageLineEdit=QLineEdit()

        labelCol=0
        contentCol=1

        leftLayout=QGridLayout()
        leftLayout.addWidget(label1, 0, labelCol)
        leftLayout.addWidget(userLineEdit, 0, contentCol)
        leftLayout.addWidget(label2, 1, labelCol)
        leftLayout.addWidget(nameLineEdit, 1, contentCol)
        leftLayout.addWidget(label3, 2, labelCol)
        leftLayout.addWidget(sexComboBox, 2, contentCol)
```

```

leftLayout.addWidget(label4, 3, labelCol)
leftLayout.addWidget(departmentTextEdit, 3, contentCol)
leftLayout.addWidget(label5, 4, labelCol)
leftLayout.addWidget(ageLineEdit, 4, contentCol)
leftLayout.addWidget(otherLabel, 5, labelCol, 1, 2)
leftLayout.setColumnStretch(0, 1)
leftLayout.setColumnStretch(1, 3)

label6=QLabel(self.tr("头像:"))
iconLabel=QLabel()
icon=QPixmap("image/2.jpg")
iconLabel.setPixmap(icon)
iconLabel.resize(icon.width(), icon.height())
iconPushButton=QPushButton(self.tr("改变"))
hLayout=QHBoxLayout()
hLayout.setSpacing(20)
hLayout.addWidget(label6)
hLayout.addWidget(iconLabel)
hLayout.addWidget(iconPushButton)

label7=QLabel(self.tr("个人说明:"))
descTextEdit=QTextEdit()

rightLayout=QVBoxLayout()
rightLayout.setMargin(10)
rightLayout.addLayout(hLayout)
rightLayout.addWidget(label7)
rightLayout.addWidget(descTextEdit)

OKPushButton=QPushButton(self.tr("确定"))
cancelPushButton=QPushButton(self.tr("取消"))
bottomLayout=QHBoxLayout()
bottomLayout.addStretch()
bottomLayout.addWidget(OKPushButton)
bottomLayout.addWidget(cancelPushButton)

mainLayout=QGridLayout(self)
mainLayout.setMargin(15)
mainLayout.setSpacing(10)
mainLayout.addLayout(leftLayout, 0, 0)
mainLayout.addLayout(rightLayout, 0, 1)
mainLayout.addLayout(bottomLayout, 1, 0, 1, 2)
mainLayout.setSizeConstraint(QLayout.SetFixedSize)

app=QApplication(sys.argv)
dialog=LayoutDialog()
dialog.show()
app.exec_()

```

第 13-26 行定义对话框左侧的控件。其中第 19 行设置控件的风格，`setFrameStyle()` 是 `QFrame` 的方法，参数以或的方式设定控件的面板风格，由形式 (`QFrame.Shape`) 和阴影 (`QFrame.Shadow`) 两项配合设定。其中，形状有 `NoFrame`, `Panel`, `Box`, `HLine`, `VLine` 以及 `WinPanel` 6 种，阴影有 `Plain`, `Raised` 和 `Sunken` 3 种，具体的效果读者可自行搭配试验。

在定义代码中，可以不必为各个控件指定父窗口，使用布局管理会自动指定布局管理下的所有控件的父窗口。

第 28-44 行定义控件布局，实现左部布局。

第 28 行定义一个 `QGridLayout` 对象 `leftLayout`，由于此布局管理器并不是主布局管理器，因此不用指定父窗口，最后由主布局管理器统一指定。

`QGridLayout` 类的 `addWidget()` 方法用来向布局中加入需布局的控件，第 32-42 行调用此方法插入需布局的控件。`addWidget()` 的函数原型如下：

```
addWidget (self, QWidget)  
addWidget (self, QWidget, int, int, Qt.Alignment alignment = 0)  
addWidget (self, QWidget, int, int, int, int, Qt.Alignment alignment = 0)
```

QWidget 参数为需插入的控件对象，后面的两个 **int** 参数为插入的行和列，再后面两上 **int** 参数为跨度的行数和跨度的列数，**alignment** 参数描述各控件的对齐方式。

第 41 行和 42 行设定两列分别占用的空间的比例，此处设定两列的空间比为 1:3。即使对话框框架大小改变了，两列之间的宽度比依然保持不变。

第 46-56 行实现对话框右上侧的头像选择区的布局，此处采用一个 **QHBoxLayout** 类进行布局管理。**QHBoxLayout** 默认采取自左向右的方式顺序排列插入的控件，也可通过调用 **setDirection()** 方法设定排列的顺序，例如：

```
hLayout.setDirection(QBoxLayout.RightToLeft)
```

第 53 行调用 **QLayout** 的 **setSpacing()** 方法设定各个控件之间的间距为 20。

第 58-65 行代码实现对话框右侧的布局。由一个 **QVBoxLayout** 实现布局，**QVBoxLayout** 默认自上而下顺序排列插入的控件或子布局，也可通过 **setDirection()** 方法改变排列的顺序。由于右侧上部的头像选择区已使用布局，因此第 63 行调用 **addLayout()** 方法在布局中插入子布局。

第 67-72 行代码实现对话框下方两个按钮的布局，采用 **QHBoxLayout** 实现。

第 70 行调用 **addStretch()** 方法在按钮之前插入一个占位符，使两个按钮能靠右对齐。并且在整个对话框的大小发生改变时，保证按钮的大小不发生变化。合理使用 **addStretch()** 能让界面的布局效果增色不少。

最后实现主布局，用一个 **QGridLayout** 实现，并在定义主布局时指定父窗口 **self**，也可调用 **self.setLayout(mainLayout)** 实现。

第 75 行设定对话框的边距为 15。

第 79 行插入的子布局占用了两列，使用的函数方法和前面的是一样的，也调用 **addLayout()** 方法，只是参数不同，原型如下：

```
addLayout (self, QLayout, int, int, int, int, Qt.Alignment alignment = 0)
```

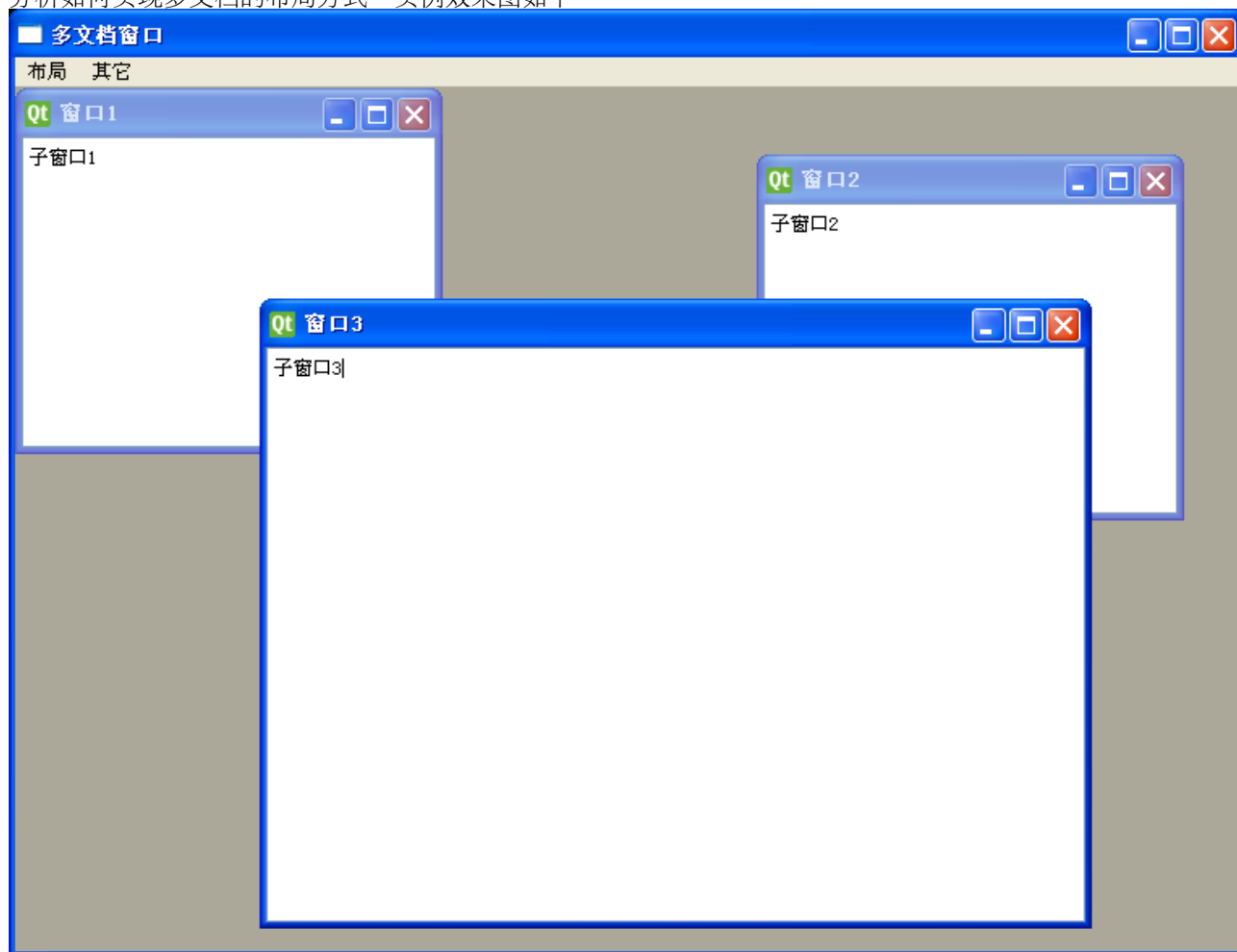
QLayout 参数为需插入的控件对象，后面的两个 **int** 参数为插入的行和列，再后面两上 **int** 参数为跨度的行数和跨度的列数，**alignment** 参数描述各控件的对齐方式。

最后，第 80 行设定对话框的控件总是最优化显示，并且用户无法改变对话框的大小，所谓最优化显示，即控件都按其 **sizeHint()** 的大小显示。

在 Qt3 的布局中，插入占用多行或多列的方法为 **addMultiCellWidget()** 和 **addMultiCellLayout()**，在 Qt4 中废弃了这两种方法，而是统一成 **addWidget()** 和 **addLayout()** 两种方法。

PyQt4 精彩实例分析 实例 13 多文档

在使用 QMainWindow 作为主窗口时，经常会用到多文档的方式对文件进行显示，本实例通过一个简单的例子分析如何实现多文档的布局方式。实例效果图如下。



Qt 提供了一个 QWorkspace 类，利用 QWorkspace 类可以很方便地实现多文档的应用。QWorkspace 类继承自 QWidget 类，因此只需在 QMainWindow 主窗口中把 QWorkspace 对象设置为中央窗体即可。QWorkspace 类提供了许多对子窗口进行排序的函数接口，如 cascade(), arrangeIcon(), title() 等。

实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWidget(QMainWindow):
    def __init__(self, parent=None):
        super(MainWidget, self).__init__(parent)
        self.setWindowTitle(self.tr("多文档窗口"))
        self.workSpace=QWorkspace()
        self.setCentralWidget(self.workSpace)

        window1=QMainWindow()
        window1.setWindowTitle(self.tr("窗口 1"))
```

```

edit1=QTextEdit(self.tr("子窗口 1"))
window1.setCentralWidget(edit1)
window2=QMainWindow()
window2.setWindowTitle(self.tr("窗口 2"))
edit2=QTextEdit(self.tr("子窗口 2"))
window2.setCentralWidget(edit2)
window3=QMainWindow()
window3.setWindowTitle(self.tr("窗口 3"))
edit3=QTextEdit(self.tr("子窗口 3"))
window3.setCentralWidget(edit3)

self.workSpace.addWindow(window1)
self.workSpace.addWindow(window2)
self.workSpace.addWindow(window3)

self.createMenu()
self.slotScroll()

def createMenu(self):
    layoutMenu=self.menuBar().addMenu(self.tr("布局"))
    arrange=QAction(self.tr("排列图标"),self)

self.connect(arrange,SIGNAL("triggered()"),self.workSpace,SLOT("arrangeIcons()"))
)
    layoutMenu.addAction(arrange)

    tile=QAction(self.tr("平铺"),self)

self.connect(tile,SIGNAL("triggered()"),self.workSpace,SLOT("tile()"))
    layoutMenu.addAction(tile)

    cascade=QAction(self.tr("层叠"),self)

self.connect(cascade,SIGNAL("triggered()"),self.workSpace,SLOT("cascade()"))
    layoutMenu.addAction(cascade)

    otherMenu=self.menuBar().addMenu(self.tr("其它"))
    scrollAct=QAction(self.tr("滚动"),self)
    self.connect(scrollAct,SIGNAL("triggered()"),self.slotScroll)
    otherMenu.addAction(scrollAct)
    otherMenu.addSeparator()

    nextAct=QAction(self.tr("下一个"),self)

self.connect(nextAct,SIGNAL("triggered()"),self.workSpace,SLOT("activateNextWindow()"))
    otherMenu.addAction(nextAct)

    previousAct=QAction(self.tr("上一个"),self)

self.connect(previousAct,SIGNAL("triggered()"),self.workSpace,SLOT("activatePrevious
Window()"))
    otherMenu.addAction(previousAct)

def slotScroll(self):
    self.workSpace.setScrollBarsEnabled(not
self.workSpace.scrollBarsEnabled())

    app=QApplication(sys.argv)
    main=MainWidget()
    main.show()
    app.exec_()

```

slotScroll()槽函数完成对多文档空间 QWorkspace 的滑动条进行设置。

第 12 行创建了一个 `QWorkspace` 对象。

第 13 行设置主窗口的中央窗体为 `QWorkspace` 对象，以使主窗口能实现多文档的布局方式。

`createMenu()` 函数创建主窗口的菜单栏

第 15-26 行新建三个子窗口用于在主窗口中显示。

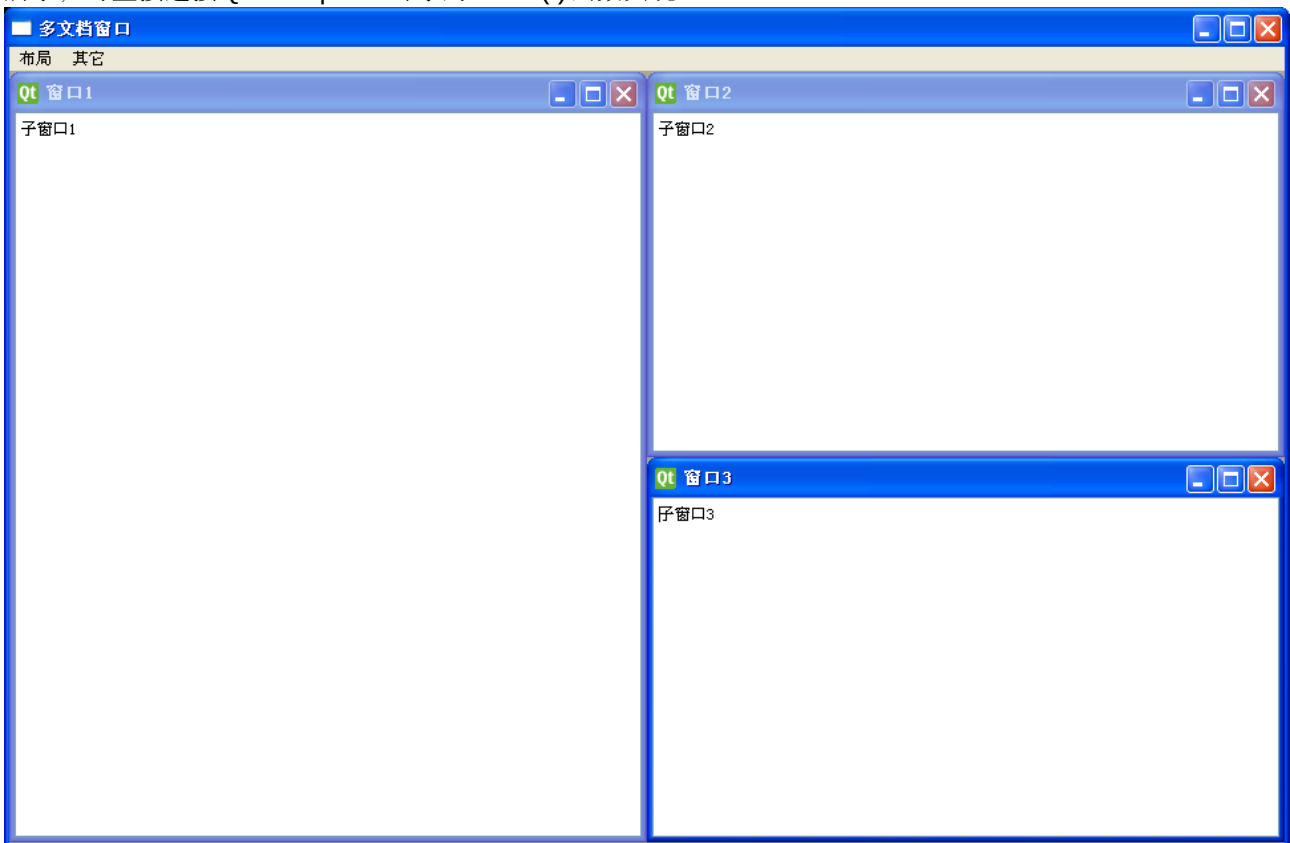
第 28-30 行在 `workSpace` 中插入这些子窗口，即实现了多文档的显示。

第 37-39 行实现对子窗口的 `arrangeIcons` 布局，它的布局方式是将所有子窗口以标题栏的方式在主窗口的底部进行排列，如下图所示。可直接把菜单的 `triggered()` 信号与 `QWorkspace` 对象的 `arrangeIcons()` 方法相连。

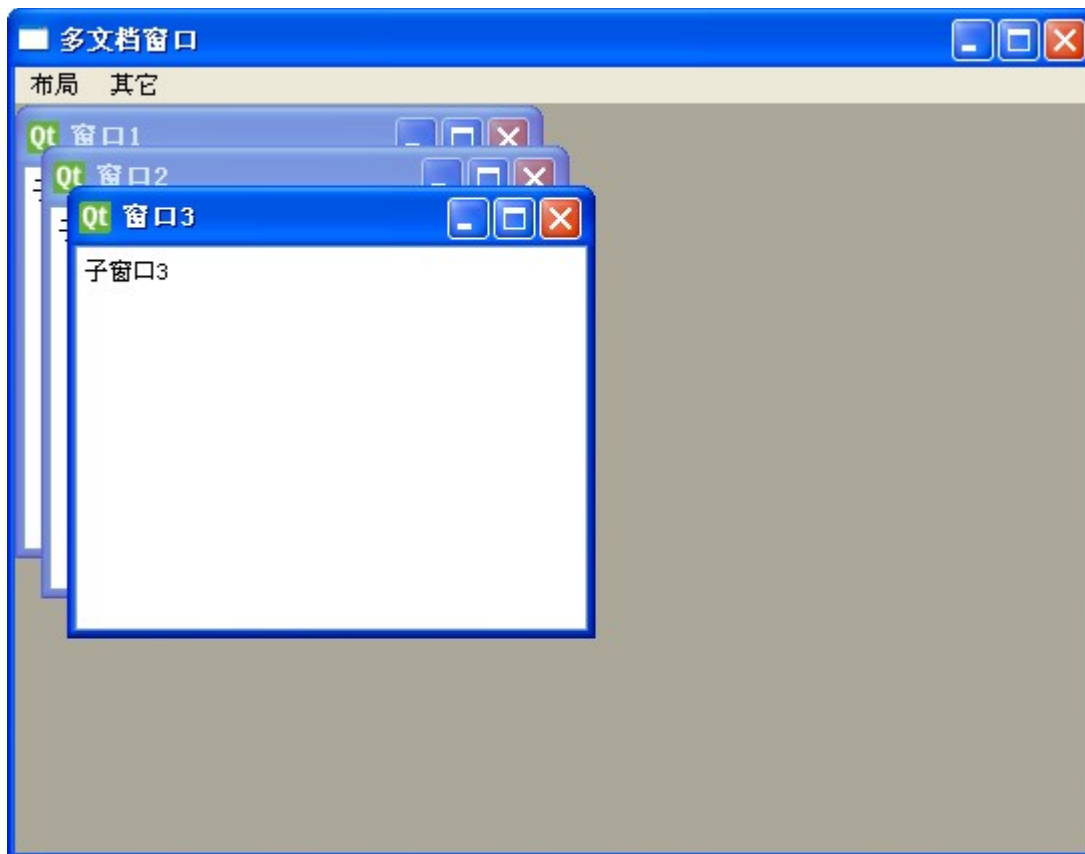


注意此排列方式，仅对已经最小化的子窗口起作用。

第 41-43 行实现子窗口的 `tile` 布局，`tile` 的意思是用子窗口把主窗口像铺瓦片或贴瓷砖一样排满，如下图所示，可直接连接 `QWorkspace` 对象的 `tile()` 函数实现。



第 45-47 行实现对子窗口的 `cascade` 布局，即子窗口的层叠显示，如下图所示，可直接连接 `QWorkspace` 对象的 `cascade()` 函数实现。



第 55-57 行使下一个子窗口获得焦点，成为当前应用子窗口，直接把菜单项与 `QWorkspace` 对象的 `activateNextWindow()` 函数连接实现。

第 59-61 行使前一个子窗口获得焦点，成为当前应用子窗口，直接把菜单项与 `QWorkspace` 对象的 `activatePreviousWindow()` 函数连接实现。

子窗口的顺序由 `QWorkspace` 的 `WindowOrder` 属性决定，有以下两种可能的顺序。

`QWorkspace.CreationOrder`: 子窗口创建的先后顺序。

`QWorkspace.StackingOrder`: 子窗口堆栈的顺序，即处于最上方的子窗口是最后一个子窗口。

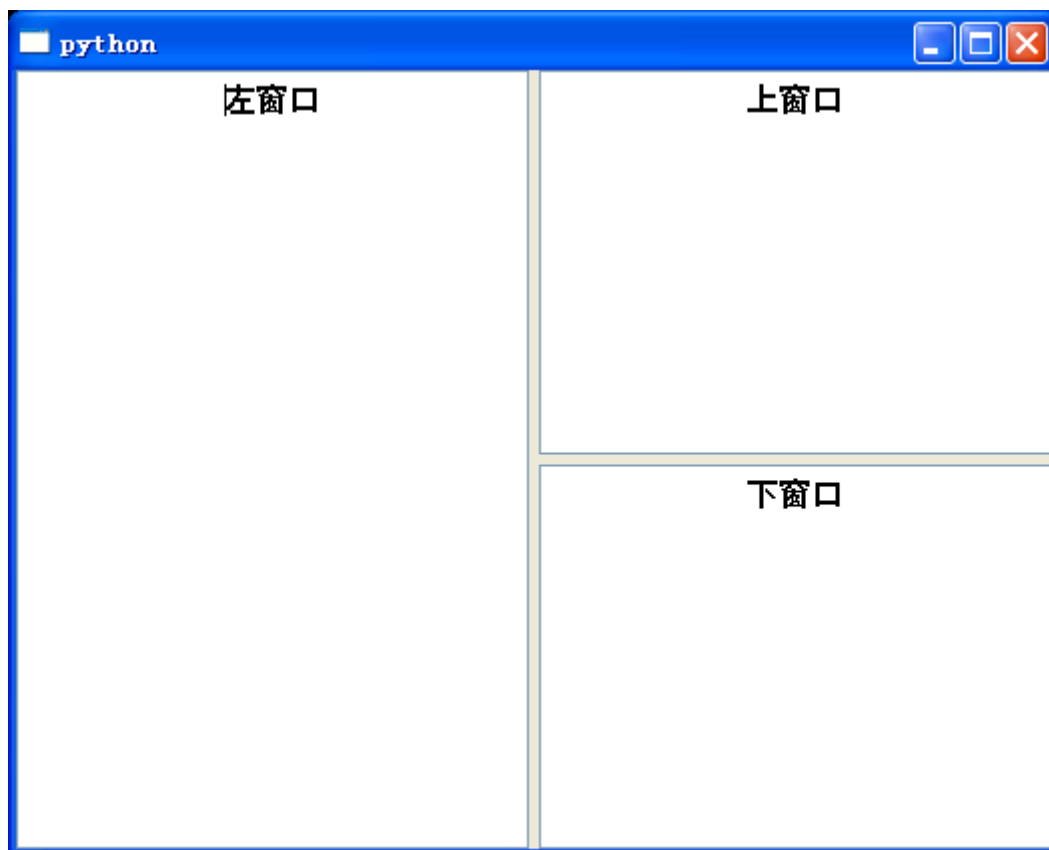
其中，`CreationOrder` 是默认的子窗口顺序。

当调用 `windowList()` 函数获得主窗口的所有子窗口列表时，以 `WindowOrder` 作为参数，返回的子窗口列表即以指明的顺序进行排列。

第 64 行通过 `QWorkspace` 的 `setScrollBarsEnabled()` 函数，可对 `workSpace` 滑动条的可用性进行设置。

PyQt4 精彩实例分析 实例 14 分割窗口

分割窗口是应用程序中经常用到的，它可以灵活分布窗口的布局，经常用于类似文件资源管理器的窗口设计中。本实例实现一个分割窗口使用的例子，实现的效果图如下。



整个对话框由 3 个窗口组成，各个窗口之间的大小可随意拖动改变。此实例使用 `QSplitter` 类来实现，实现代码如下所示：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWidget(QMainWindow):
    def __init__(self, parent=None):
        super(MainWidget, self).__init__(parent)
        font=QFont(self.tr("黑体"), 12)
        QApplication.setFont(font)

        mainSplitter=QSplitter(Qt.Horizontal, self)
        leftText=QTextEdit(self.tr("左窗口"), mainSplitter)
        leftText.setAlignment(Qt.AlignCenter)
        rightSplitter=QSplitter(Qt.Vertical, mainSplitter)
        rightSplitter.setOpaqueResize(False)
        upText=QTextEdit(self.tr("上窗口"), rightSplitter)
        upText.setAlignment(Qt.AlignCenter)
        bottomText=QTextEdit(self.tr("下窗口"), rightSplitter)
        bottomText.setAlignment(Qt.AlignCenter)
        mainSplitter.setStretchFactor(1, 1)
```

```
        mainSplitter.setWindowTitle(self.tr("分割窗口"))

        self.setCentralWidget(mainSplitter)

    app=QApplication(sys.argv)
    main=MainWidget()
    main.show()
    app.exec_()
```

第 11-12 行指定显示的字体。

第 14 行定义一个 `QSplitter` 类对象，为主分割窗口，设定此分割窗为水平分割窗。

第 15 行定义一个 `QTextEdit` 类对象，并插入主分割窗口中。

第 16 行调用 `setAlignment()` 方法，设定 `TextEdit` 中文字的对齐方式，常用的有以下几种。

`Qt.AlignLeft`: 左对齐。

`Qt.AlignRight`: 右对齐。

`Qt.AlignCenter`: 文字居中(`Qt.AlignHCenter` 为水平居中，`Qt.AlignVCenter` 为垂直居中)。

`Qt.AlignUp`: 文字与顶端对齐。

`Qt.AlignBottom`: 文字与底部对齐。

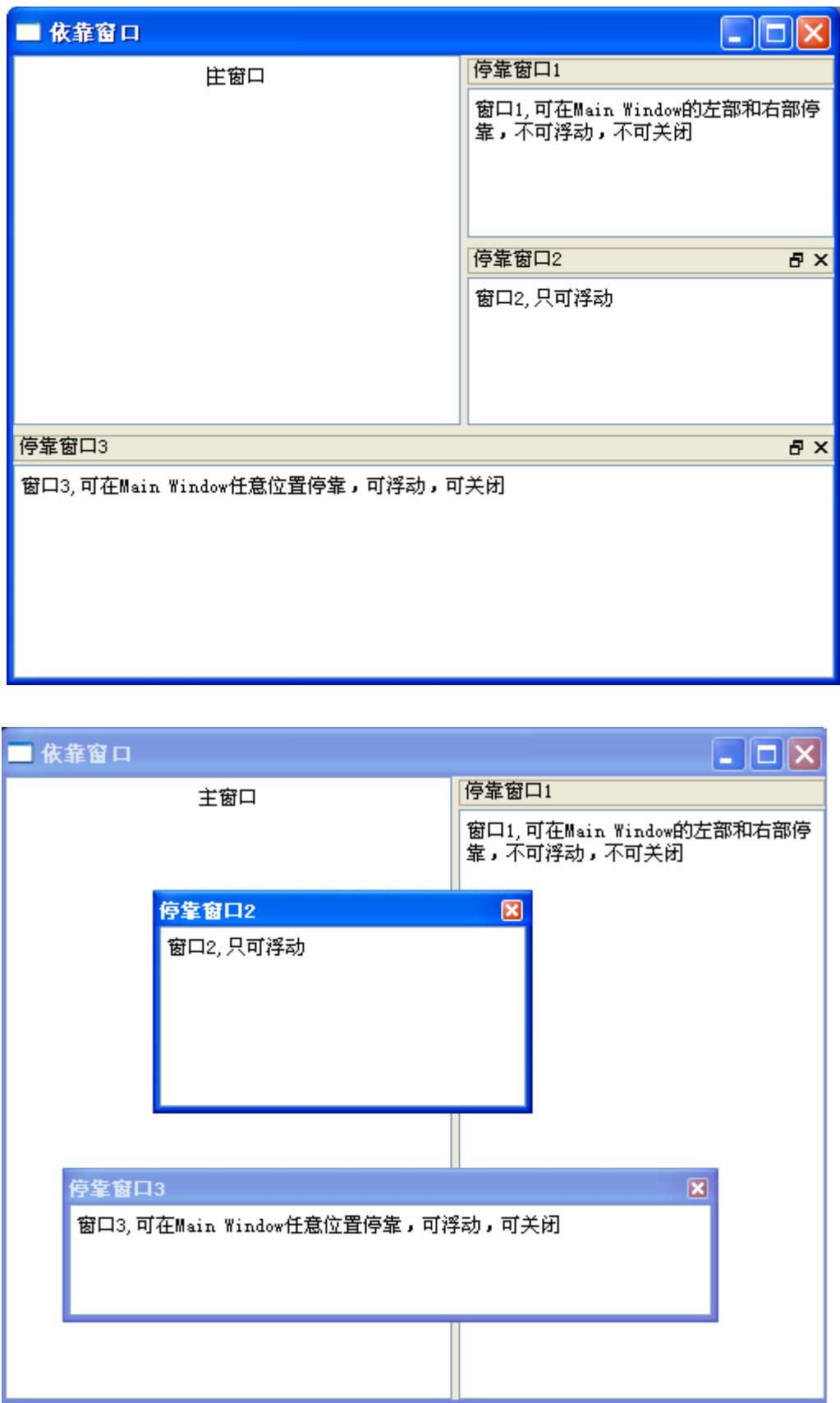
第 17 行定义一个右部的分割窗口，定义为垂直分割窗，并以主分割窗口为父窗口。

第 18 行调用的方法 `setOpaqueResize(boolean)` 用于设定分割窗的分割条在拖动时是否为实时更新显示，若设为 `True` 则实时更新显示，若设为 `False` 则在拖动时只显示一条灰色的精线条，在拖动到位并弹起鼠标后再显示分割条。默认设为 `True`，这和 Qt3 正好相反，Qt3 中默认为 `False`。

第 23 行 `setStretchFactor()` 方法用于设定可伸缩控件，它的第一个参数指定设置的控件序号，控件序号按插入的先后次序从 0 起依次编号，第二个参数大于 0 的值表示此控件为可伸缩控件。此实例中设定右部分割窗为可伸缩控件，当整个对话框的宽度发生改变时，左部的文件编辑框宽度保持不变，右部的分割窗宽度随整个对话框大小的改变进行调整。

PyQt4 精彩实例分析 实例 15 停靠窗口

本实例实现停靠窗口的基本使用方法，实现的效果图如下所示。



本实例实现的停靠窗口的可实现状态已在各窗口中进行了描述，停靠窗口 1 只可在主窗口的左边和右边停靠，停靠窗口 2 只可在浮动和在右停靠两种状态间切换，并且不可移动，停靠窗口 3 可实现停靠窗口的各个状态。具体状态此处不再一一用图示的方式列出，读者可自行运行代码进行试验。

具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle(self.tr("停靠窗口"))

        te=QTextEdit(self.tr("主窗口"))
        te.setAlignment(Qt.AlignCenter)
        self.setCentralWidget(te)

        #停靠窗口 1
        dock1=QDockWidget(self.tr("停靠窗口 1"), self)
        dock1.setFeatures(QDockWidget.DockWidgetMovable)
        dock1.setAllowedAreas(Qt.LeftDockWidgetArea|Qt.RightDockWidgetArea)
        te1=QTextEdit(self.tr("窗口 1, 可在 Main Window 的左部和右部停靠，不可浮动，不可关闭"))
        dock1.setWidget(te1)
        self.addDockWidget(Qt.RightDockWidgetArea, dock1)

        #停靠窗口 2
        dock2=QDockWidget(self.tr("停靠窗口 2"), self)
        dock2.setFeatures(QDockWidget.DockWidgetFloatable|
QDockWidget.DockWidgetClosable)
        te2=QTextEdit(self.tr("窗口 2, 只可浮动"))
        dock2.setWidget(te2)
        self.addDockWidget(Qt.RightDockWidgetArea, dock2)

        #停靠窗口 3
        dock3=QDockWidget(self.tr("停靠窗口 3"), self)
        dock3.setFeatures(QDockWidget.AllDockWidgetFeatures)
        te3=QTextEdit(self.tr("窗口 3, 可在 Main Window 任意位置停靠，可浮动，可关闭"))
        dock3.setWidget(te3)
        self.addDockWidget(Qt.BottomDockWidgetArea, dock3)

        app=QApplication(sys.argv)
        main=MainWindow()
        main.show()
        app.exec_()
```

第 11 行设置主窗口的标题栏文字。

第 13-15 行定义一个 QTextEdit 对象作为主窗口，并把此编辑框设为 MainWindow 的中央窗体。

第 18-23 行设置停靠窗口 1。

第 26-30 行设置停靠窗口 2。

第 33-37 行设置停靠窗口 3。

设置停靠窗口的一般流程为：

- 1) 创建一个 `QDockWidget` 对象的停靠窗体。
- 2) 设置此停靠窗体的属性，通常调用 `setFeatures()` 及 `setAllowedAreas()` 两个方法。
- 3) 新建一个要插入停靠窗体的控件，本实例中为 `QTextEdit`，也可为其他控件，常用的一般为 `QListWidget` 和 `QTextEdit`。
- 4) 把控件插入停靠窗体，调用 `QDockWidget` 的 `setWidget()` 方法。
- 5) 使用 `addDockWidget()` 方法在 `MainWindow` 中加入此停靠窗体。

本实例的 3 个停靠窗体都是按此流程实现的，此处需要重点介绍的是设置停靠窗体状态的方法 `setAllowedAreas()` 和 `setFeatures()`。

其中 `setAllowedAreas()` 方法设置停靠窗体可停靠的区域，原型如下：

```
setAllowedAreas (self, Qt.DockWidgetAreas)
```

参数 `Qt.DockWidgetAreas` 指定了停靠窗体可停靠的区域，包括以下几种。

`Qt.LeftDockWidgetArea`: 可在主窗口的左侧停靠。

`Qt.RightDockWidgetArea`: 可在主窗口的右侧停靠。

`Qt.TopDockWidgetArea`: 可在主窗口的顶端停靠。

`Qt.BottomDockWidgetArea`: 可在主窗口的底部停靠。

`Qt.AllDockWidgetArea`: 可在主窗口任意（以上四个）部位停靠。

`Qt.NoDockWidgetArea`: 可停靠在插入处。

各区域设定可采用或(`|`)的方式进行设定，如本实例中的第 8 行。

`setFeatures()` 方法设置停靠窗体的特性，原型如下：

```
setFeatures (self, DockWidgetFeatures)
```

参数 `QDockWidgetFeatures` 指定停靠窗体的特性，包括以下几种。

`QDockWidget.DockWidgetClosable`: 停靠窗可关闭，右上角的关闭按钮。

`QDockWidget.DockWidgetMovable`: 停靠窗可移动。

`QDockWidget.DockWidgetFloatable`: 停靠窗可浮动。

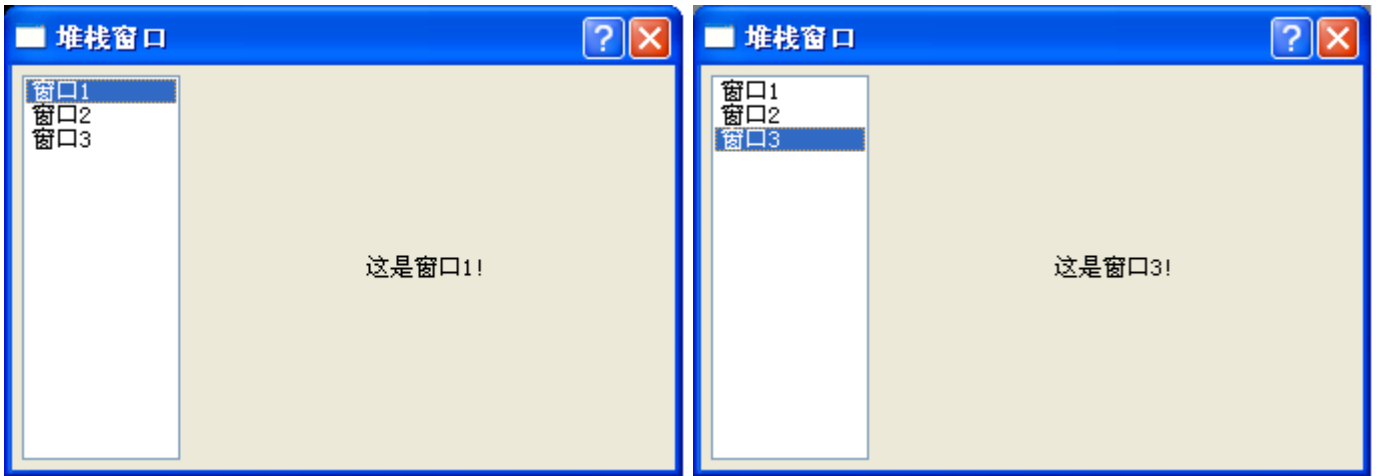
`QDockWidget.AllDockWidgetFeature`: 此参数表示拥有停靠窗的所有特性。

`QDockWidget.NoDockWidgetFeature`: 不可移动，不可关闭，不可浮动。

此参数也可采用或(`|`)的方式对停靠窗进行特性的设定，如实例中的第 27 行所示。

PyQt4 精彩实例分析 实例 16 堆栈窗口

本实例实现一个堆栈窗体的使用，实现效果图如下所示。



选择左侧列表框不同的选项，右侧则显示所选的窗体。

实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class StockDialog(QDialog):
    def __init__(self, parent=None):
        super(StockDialog, self).__init__(parent)
        self.setWindowTitle(self.tr("堆栈窗口"))

        listWidget=QListWidget()
        listWidget.insertItem(0, self.tr("窗口 1"))
        listWidget.insertItem(1, self.tr("窗口 2"))
        listWidget.insertItem(2, self.tr("窗口 3"))
        label1=QLabel(self.tr("这是窗口 1!"))
        label2=QLabel(self.tr("这是窗口 2!"))
        label3=QLabel(self.tr("这是窗口 3!"))

        stack=QStackedWidget()
        stack.addWidget(label1)
        stack.addWidget(label2)
        stack.addWidget(label3)

        mainLayout=QHBoxLayout(self)
        mainLayout.setMargin(5)
        mainLayout.setSpacing(5)
        mainLayout.addWidget(listWidget)
        mainLayout.addWidget(stack, 0, Qt.AlignHCenter)
        mainLayout.setStretchFactor(listWidget, 1)
        mainLayout.setStretchFactor(stack, 3)

        self.connect(listWidget, SIGNAL("currentRowChanged(int)"), stack, SLOT("setCurrentIndex(int)"))
```

```
app=QApplication(sys.argv)
main=StockDialog()
main.show()
app.exec_()
```

第 13-16 行创建一个 `QListWidget` 控件，并在控件中插入 3 个条目，作为选择项。

第 17-19 行创建 3 个 `QLabel` 标签控件，作为堆栈窗口显示的三层窗体。

第 21 行创建一个 `QStackedWidget` 堆栈窗。

第 22-24 行调用 `addWidget()` 方法把前面创建的 3 个标签控件依次插入堆栈窗中。

第 26-32 行使用 `QHBoxLayout` 对整个对话框进行布局。

第 33 行连接 `QListWidget` 的 `currentRowChanged()` 信号与堆栈窗的 `setCurrentIndex()` 槽，实现按选择显示窗体。此处的堆栈窗体 `index` 按插入的顺序从 0 起依次排序，与 `QListWidget` 的条目排序相一致。

本实例分析了堆栈窗的基本使用方法。在实际应用中，堆栈窗口多与列表框 `QListWidget` 及下拉列表框 `QComboBox` 配合使用。

PyQt4 精彩实例分析 实例 17 综合布局实例

本实例综合应用前面介绍的布局方法实现一个复杂的窗口布局，实现效果图如下所示。其中包括了基本布局，分割窗以及堆栈窗。


综合布局实例

个人基本资料
联系方式
详细信息

用户名:
姓名:
性别:
部门:
年龄:
备注:

男

头像:
个人说明:



改变

修改

关闭

综合布局实例

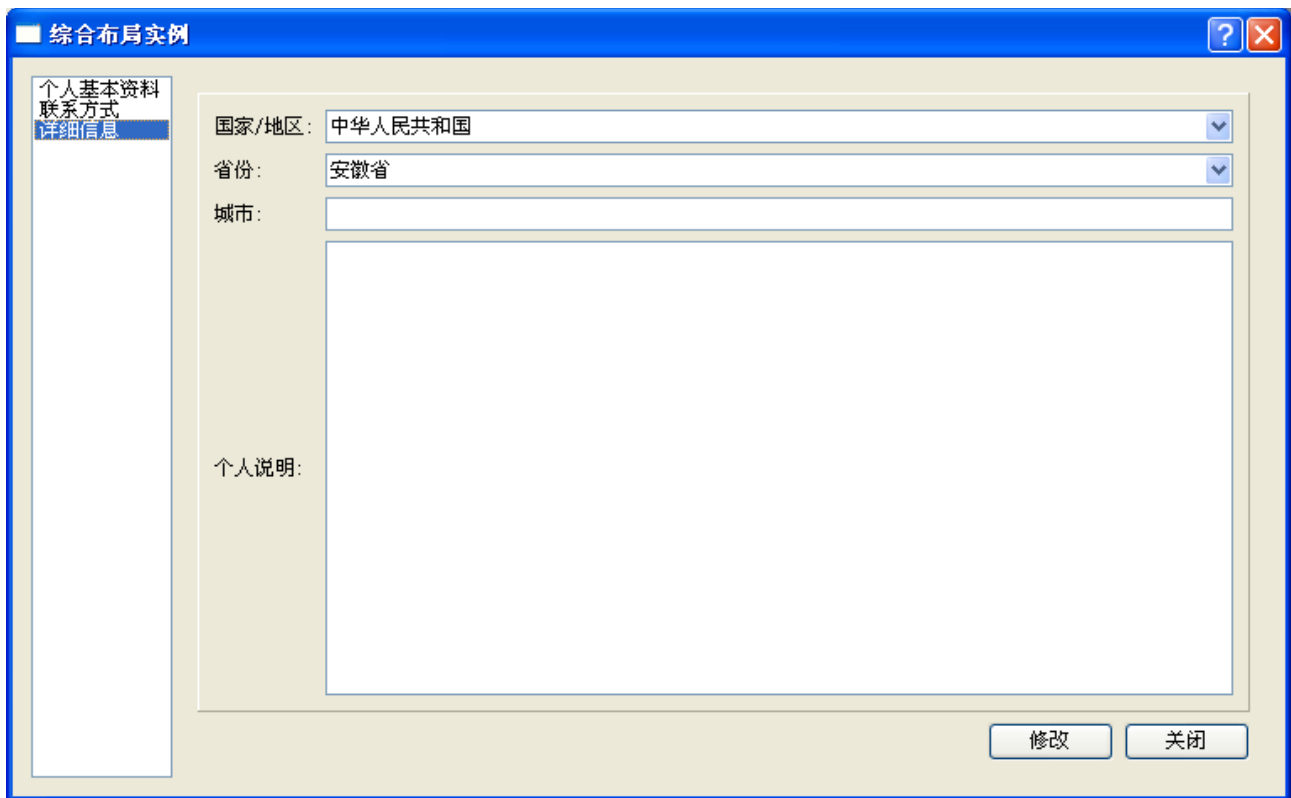
个人基本资料
联系方式
详细信息

电子邮件:
联系地址:
邮政编码:
移动电话:
办公电话:

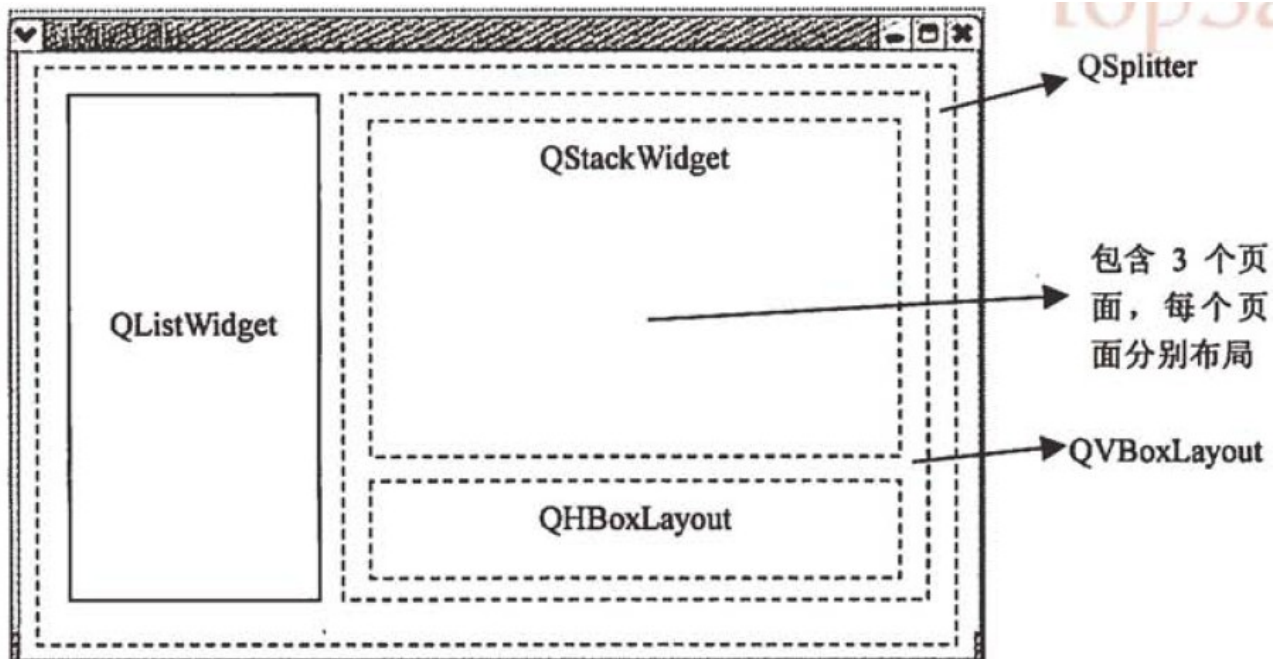
接收留言

修改

关闭



首先对整个窗体的构成进行一个整体的分析。最外层的是一个分割窗体 `QSplitter`，分割窗的左侧为一个 `QListWidget`，右侧为一个 `QVBoxLayout` 布局，包括一个堆栈窗 `QStackWidget` 和一个按钮布局，在堆栈窗中包含 3 个窗体，每个窗体采用基本布局方式进行布局管理。窗体的布局可用如下的示意图表示。



堆栈窗中的 3 个页面分别定义了 3 个 `QWidget` 子类，包括“个人基本资料”页，由 `BaseInfo` 类实现，“联系方式”页，由 `Contact` 类实现，“详细信息”页，由 `Detail` 类实现。

“个人基本资料”页与实例 12 中的内容一样，可直接引用。

“联系方式”页和“详细信息”页都是采用表格布局的方式进行布局管理，用法与前面所介绍的类似，此处不再重复进行分析。

具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
```

```
QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))
```

```
class StockDialog(QDialog):
    def __init__(self, parent=None):
        super(StockDialog, self).__init__(parent)
        self.setWindowTitle(self.tr("综合布局实例"))

        mainSplitter=QSplitter(Qt.Horizontal)
        mainSplitter.setOpaqueResize(True)

        listWidget=QListWidget(mainSplitter)
        listWidget.insertItem(0, self.tr("个人资料"))
        listWidget.insertItem(1, self.tr("联系方式"))
        listWidget.insertItem(2, self.tr("详细信息"))

        frame=QFrame(mainSplitter)
        stack=QStackedWidget()
        stack.setFrameStyle(QFrame.Panel|QFrame.Raised)

        baseInfo=BaseInfo()
        contact=Contact()
        detail=Detail()
        stack.addWidget(baseInfo)
        stack.addWidget(contact)
        stack.addWidget(detail)

        amendPushButton=QPushButton(self.tr("修改"))
        closePushButton=QPushButton(self.tr("关闭"))

        buttonLayout=QHBoxLayout()
        buttonLayout.addStretch(1)
        buttonLayout.addWidget(amendPushButton)
        buttonLayout.addWidget(closePushButton)

        mainLayout=QVBoxLayout(frame)
        mainLayout.setMargin(10)
        mainLayout.setSpacing(6)
        mainLayout.addWidget(stack)
        mainLayout.addLayout(buttonLayout)
```

```
self.connect(listWidget, SIGNAL("currentRowChanged(int)"), stack, SLOT("setCurrentIndex(int)"))
```

```
self.connect(closePushButton, SIGNAL("clicked()"), self, SLOT("close()"))
```

```
        layout=QHBoxLayout(self)
        layout.addWidget(mainSplitter)
        self.setLayout(layout)
```

```
class BaseInfo(QWidget):
    def __init__(self, parent=None):
        super(BaseInfo, self).__init__(parent)

        label1=QLabel(self.tr("用户名:"))
        label2=QLabel(self.tr("姓名: "))
        label3=QLabel(self.tr("性别:"))
```

```

label4=QLabel(self.tr("部门:"))
label5=QLabel(self.tr("年龄:"))
otherLabel=QLabel(self.tr("备注:"))
otherLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
userLineEdit=QLineEdit()
nameLineEdit=QLineEdit()
sexComboBox=QComboBox()
sexComboBox.insertItem(0,self.tr("男"))
sexComboBox.insertItem(1,self.tr("女"))
departmentTextEdit=QTextEdit()
ageLineEdit=QLineEdit()

labelCol=0
contentCol=1

leftLayout=QGridLayout()
leftLayout.addWidget(label1,0,labelCol)
leftLayout.addWidget(userLineEdit,0,contentCol)
leftLayout.addWidget(label2,1,labelCol)
leftLayout.addWidget(nameLineEdit,1,contentCol)
leftLayout.addWidget(label3,2,labelCol)
leftLayout.addWidget(sexComboBox,2,contentCol)
leftLayout.addWidget(label4,3,labelCol)
leftLayout.addWidget(departmentTextEdit,3,contentCol)
leftLayout.addWidget(label5,4,labelCol)
leftLayout.addWidget(ageLineEdit,4,contentCol)
leftLayout.addWidget(otherLabel,5,labelCol,1,2)
leftLayout.setColumnStretch(0,1)
leftLayout.setColumnStretch(1,3)

label6=QLabel(self.tr("头像:"))
iconLabel=QLabel()
icon=QPixmap("image/2.jpg")
iconLabel.setPixmap(icon)
iconLabel.resize(icon.width(),icon.height())
iconPushButton=QPushButton(self.tr("改变"))
hLayout=QHBoxLayout()
hLayout.setSpacing(20)
hLayout.addWidget(label6)
hLayout.addWidget(iconLabel)
hLayout.addWidget(iconPushButton)

label7=QLabel(self.tr("个人说明:"))
descTextEdit=QTextEdit()

rightLayout=QVBoxLayout()
rightLayout.setMargin(10)
rightLayout.addLayout(hLayout)
rightLayout.addWidget(label7)
rightLayout.addWidget(descTextEdit)
mainLayout=QGridLayout(self)
mainLayout.setMargin(15)
mainLayout.setSpacing(10)
mainLayout.addLayout(leftLayout,0,0)
mainLayout.addLayout(rightLayout,0,1)
mainLayout.setSizeConstraint(QLayout.SetFixedSize)

```

```

class Contact(QWidget):
    def __init__(self,parent=None):
        super(Contact,self).__init__(parent)
        label1=QLabel(self.tr("电子邮件:"))
        label2=QLabel(self.tr("联系地址:"))
        label3=QLabel(self.tr("邮政编码:"))

```

```

label4=QLabel(self.tr("移动电话:"))
label5=QLabel(self.tr("办公电话:"))

mailLineEdit=QLineEdit()
addressLineEdit=QLineEdit()
codeLineEdit=QLineEdit()
mpLineEdit=QLineEdit()
phoneLineEdit=QLineEdit()
receiveCheckBox=QCheckBox(self.tr("接收留言"))

layout=QGridLayout(self)
layout.addWidget(label1,0,0)
layout.addWidget(mailLineEdit,0,1)
layout.addWidget(label2,1,0)
layout.addWidget(addressLineEdit,1,1)
layout.addWidget(label3,2,0)
layout.addWidget(codeLineEdit,2,1)
layout.addWidget(label4,3,0)
layout.addWidget(mpLineEdit,3,1)
layout.addWidget(receiveCheckBox,3,2)
layout.addWidget(label5,4,0)
layout.addWidget(phoneLineEdit,4,1)

class Detail(QWidget):
    def __init__(self,parent=None):
        super(Detail,self).__init__(parent)
        label1=QLabel(self.tr("国家/地区:"))
        label2=QLabel(self.tr("省份:"))
        label3=QLabel(self.tr("城市:"))
        label4=QLabel(self.tr("个人说明:"))

        countryComboBox=QComboBox()
        countryComboBox.addItem(self.tr("中华人民共和国"))
        countryComboBox.addItem(self.tr("香港"))
        countryComboBox.addItem(self.tr("台北"))
        countryComboBox.addItem(self.tr("澳门"))
        provinceComboBox=QComboBox()
        provinceComboBox.addItem(self.tr("安徽省"))
        provinceComboBox.addItem(self.tr("北京市"))
        provinceComboBox.addItem(self.tr("江苏省"))
        cityLineEdit=QLineEdit()
        remarkTextEdit=QTextEdit()

        layout=QGridLayout(self)
        layout.addWidget(label1,0,0)
        layout.addWidget(countryComboBox,0,1)
        layout.addWidget(label2,1,0)
        layout.addWidget(provinceComboBox,1,1)
        layout.addWidget(label3,2,0)
        layout.addWidget(cityLineEdit,2,1)
        layout.addWidget(label4,3,0)
        layout.addWidget(remarkTextEdit,3,1)

app=QApplication(sys.argv)
main=StockDialog()
main.show()
app.exec_()

```

第 22-30 行创建一个 `QStackWidget` 对象，第 23 行调用 `setFrameStyle()` 方法对堆栈窗的显示风格进行设置。

第 28, 29, 30 行在堆栈窗中顺序插入“个人基本资料”，“联系方式”，“详细信息”3 个页面。

第 32-38 行创建两个按钮，并且 `QHBoxLayout` 对其进行布局。

第 40-44 行采用 `QVBoxLayout` 生成主布局。

第 13 行创建一个水平分割窗，作为主布局框。

第 16-19 行在水平分割窗的左侧窗体中插入一个 `QListWidget` 作为条目选择框，并在列表框中依次插入相应的条目。

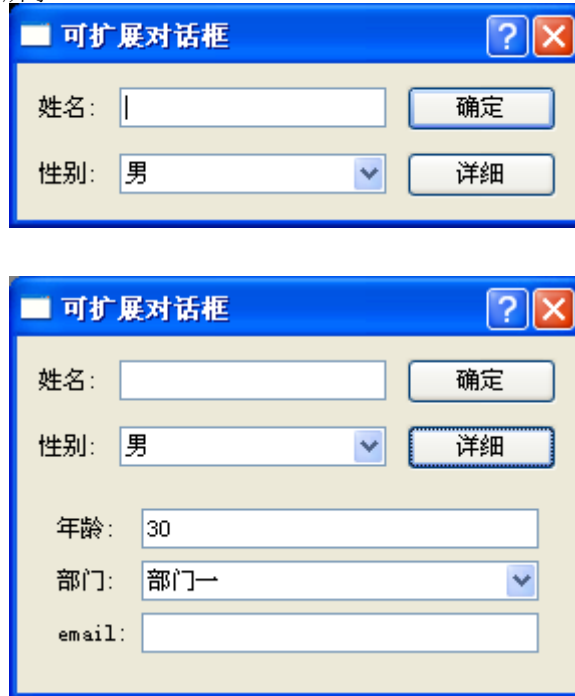
第 46 行把列表框的 `currentRowChanged()` 信号与堆栈窗的 `setCurrentIndex()` 槽相连接，达到按用户选择的条目显示页面的要求。

本实例综合应用了各种布局方式，完成一个较为复杂的界面显示。包括了各种基本布局类的应用，堆栈窗的应用和分割窗的应用。

要达到同样的显示效果，会有多种可能的布局方案，在实际应用中，应根据具体情况进行选择，使用最方便合理的布局方式。一般来说，`QGridLayout` 功能较为强大，能完成 `QHBoxLayout` 与 `QVBoxLayout` 的功能，但视具体情况，若只是简单的控件水平或竖直排列，使用 `QHBoxLayout` 和 `QVBoxLayout` 更加方便，`QGridLayout` 适合较为整齐方正的界面布局。

PyQt4 精彩实例分析 实例 18 可扩展对话框

可扩展对话框一般用于使用用户有区分的场合。通常情况下，只出现基本的对话框体，当有高级用户使用，或需要更多信息时，通过某种方式的切换显示完整的对话框体，切换的工作通常由一个按钮来实现。本实例即实现了一个简单的填写资料的例子，通常情况下，只需填写姓名和性别，在有特殊需要时，还需填写更多信息则切换至完整对话框体。如下图所示。



当单击“详细”按钮时，对话框扩展，显示其他更详细的信息，再次单击“详细”按钮，扩展窗口又重新隐藏。

具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class Extension(QDialog):
    def __init__(self, parent=None):
        super(Extension, self).__init__(parent)
        self.setWindowTitle(self.tr("可扩展对话框"))

        nameLabel=QLabel(self.tr("姓名:"))
        nameLineEdit=QLineEdit()
        sexLabel=QLabel(self.tr("性别:"))
        sexComboBox=QComboBox()
        sexComboBox.addItem(self.tr("男"))
        sexComboBox.addItem(self.tr("女"))

        okButton=QPushButton(self.tr("确定"))
        detailButton=QPushButton(self.tr("详细"))

        self.connect(detailButton, SIGNAL("clicked()"), self.slotExtension)
```

```

btnBox=QDialogButtonBox(Qt.Vertical)
btnBox.addButton(okButton,QDialogButtonBox.ActionRole)
btnBox.addButton(detailButton,QDialogButtonBox.ActionRole)

baseLayout=QGridLayout()
baseLayout.addWidget(nameLabel,0,0)
baseLayout.addWidget(nameLineEdit,0,1)
baseLayout.addWidget(okButton,0,2)
baseLayout.addWidget(sexLabel,1,0)
baseLayout.addWidget(sexComboBox,1,1)
baseLayout.addWidget(detailButton,1,2)

ageLabel=QLabel(self.tr("年龄:"))
ageLineEdit=QLineEdit("30")
departmentLabel=QLabel(self.tr("部门:"))
departmentComboBox=QComboBox()
departmentComboBox.addItem(self.tr("部门一"))
departmentComboBox.addItem(self.tr("部门二"))
departmentComboBox.addItem(self.tr("部门三"))
emailLabel=QLabel("email:")
emailLineEdit=QLineEdit()

self.detailWidget=QWidget()
detailLayout=QGridLayout(self.detailWidget)
detailLayout.addWidget(ageLabel,0,0)
detailLayout.addWidget(ageLineEdit,0,1)
detailLayout.addWidget(departmentLabel,1,0)
detailLayout.addWidget(departmentComboBox,1,1)
detailLayout.addWidget(emailLabel,2,0)
detailLayout.addWidget(emailLineEdit,2,1)
self.detailWidget.hide()

mainLayout=QVBoxLayout()
mainLayout.addLayout(baseLayout)
mainLayout.addWidget(self.detailWidget)
mainLayout.setSizeConstraint(QLayout.SetFixedSize)
mainLayout.setSpacing(10)

self.setLayout(mainLayout)

def slotExtension(self):
    if self.detailWidget.isHidden():
        self.detailWidget.show()
    else:
        self.detailWidget.hide()

app=QApplication(sys.argv)
main=Extension()
main.show()
app.exec_()

```

第 13-54 行分别构建两部分窗体的内容。

第 57-63 行对整个对话框进行布局，其中，调用 `setSizeConstraint(QLayout.SetFixedSize)` 设置窗体的大小固定，不能经过拖动改变大小，否则当再次单击“详细”按钮时，对话框不能恢复到初始状态。

`slotExtension()` 函数完成窗体扩展切换的工作，在用户单击 `detailButton` 时调用此函数，首先检测 `detailWidget` 窗体处于何种状态。若此时是隐藏状态，则应用 `show()` 函数显示 `detailWidget` 窗体，否则调用 `hide()` 隐藏 `detailWidget` 窗体。

通过本实例的分析，可了解扩展对话框的基本实现方法，其中最关键的部分有以下两点：

1) 在整个对话框的构造函数中调用。

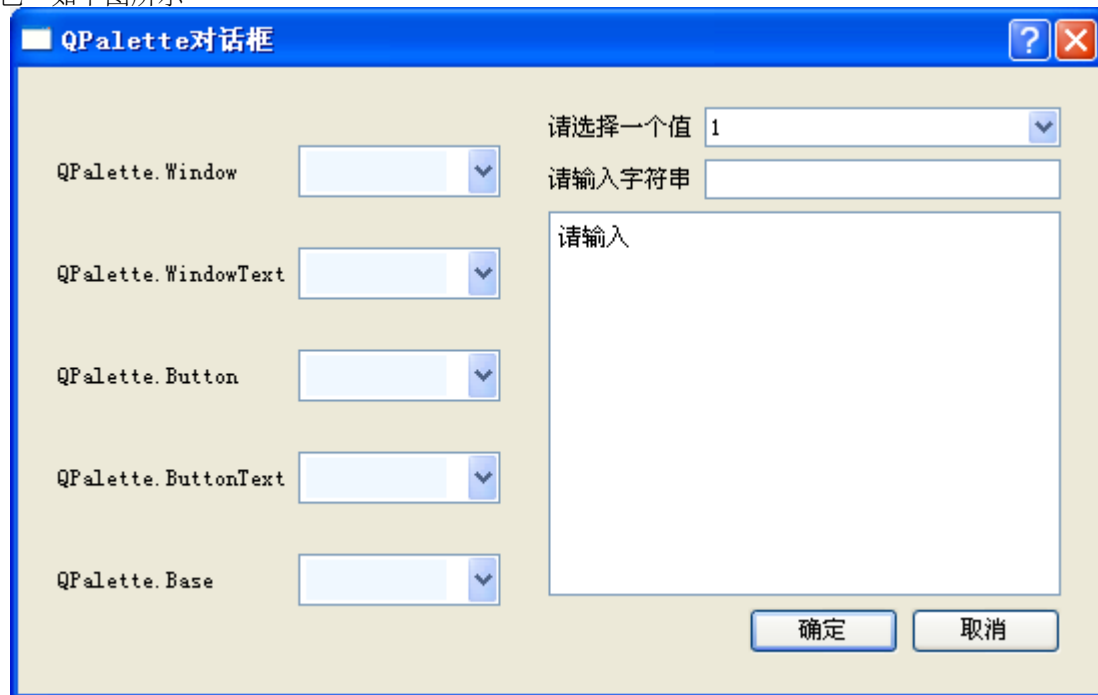
```
mainLayout.setSizeConstraint(QLayout.SetFixedSize)
```

这个设置保证了对话框的尺寸保持相对固定，始终是各控件组合的默认尺寸，在扩展部分显示时，对话框尺寸根据需显示的控件进行扩展调整，而在扩展部分隐藏时，对话框尺寸又恢复至初始状态。

2) 切换按钮的实现。整个窗体可扩展的工作都是在此按钮所连接的槽函数中完成。

PyQt4 精彩实例分析 实例 19 利用 QPalette 改变控件颜色

在实际应用中，常常需要改变某个控件的颜色外观，如背景，文字颜色等，Qt 提供的调色板类 `QPalette` 专门用于管理对话框的外观显示。本实例即通过一个具体的例子，分析如何利用 `QPalette` 来改变窗体中控件的颜色。如下图所示。



`QPalette` 类相当于对话框或是控件的调色板，它管理着控件或窗体的所有颜色信息，每个窗体或控件都包含一个 `QPalette` 对象，在显示时按照它的 `QPalette` 对象中对各部分各状态下的颜色的描述来进行绘制。就像油漆匠的油漆计划，当要刷墙时，到计划中去查一下墙需要刷成什么颜色；当要刷门时，到计划中去查一下门需要刷成什么颜色。采用这种方式可以很方便地对窗体的各种颜色信息进行管理。

`QPalette` 类有两个基本的概念，一个是 `ColorGroup`，另一个是 `ColorRole`，其中 `ColorGroup` 指的是 3 种不同的状态，包括以下几种。

`QPalette.Active`: 获得焦点的状态。

`QPalette.Inactive`: 未获得焦点的状态。

`QPalette.Disable`: 不可用状态。

通常情况下，`Active` 状态与 `Inactive` 状态下颜色显示是一致的，当然也可根据需要设置成不一样的颜色。

`ColorRole` 指的是颜色主题，即对窗体中不同的部位颜色的分类，如 `QPalette.Window` 是指背景色，`QPalette.WindowText` 指的是前景色等，由于数量较多，此处不一一列举，具体使用时可查阅 Qt 的在线帮助。

`QPalette` 类使用最多，最重要的函数是 `setColor()` 函数，其原型如下：

```
setColor (self, ColorRole, QColor)
```

```
setColor (self, ColorGroup, ColorRole, QColor)
```

第一个 `setColor()` 函数对某个主题的颜色进行设置，并不区分状态；第二个 `setColor()` 函数对主题颜色进行设置的同时，还区分了状态，即对某个主题在某个状态下的颜色进行了设置。

`QPalette` 类同时还提供了 `setBrush()` 函数，通过画刷的设置来显示进行更改，这样就有可能使用图片而不仅是单一的颜色来对主题进行填充了。

Qt 之前版本中有关背景色设置的函数如 `setBackgroundColor()` 或是前景色设置的函数如

`setForegroundColor()`在Qt4中都被废止，统一由`QPalette`类进行管理。

如`setBackgroundColor()`函数可由以下语句代替：

```
xxx.setAutoFillBackground(True)

p=xxx.palette()

p.setColor(QPalette.Window,color)

xxx.setPalette(p)
```

如果并不是用单一的颜色填充背景，也可将`setColor()`函数换成`setBrush()`函数对背景主题进行设置。

注意要先调用`setAutoFillBackground(True)`设置窗体自动填充背景。

本实例的窗体分为两个部分，一部分用于对不同主题颜色的选择，另一部分用于显示选择的颜色对于窗体外观的改变。

具体实际代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class Palette(QDialog):
    def __init__(self, parent=None):
        super(Extension, self).__init__(parent)
        self.setWindowTitle(self.tr("QPalette 对话框"))

        mainLayout=QHBoxLayout(self)
        self.ctrlFrame=QFrame()
        self.contentFrame=QFrame()
        self.contentFrame.setAutoFillBackground(True)
        self.createCtrlFrame()
        self.createContentFrame()
        mainLayout.addWidget(self.ctrlFrame)
        mainLayout.addWidget(self.contentFrame)

    def createCtrlFrame(self):
        label1=QLabel("QPalette.Window")
        self.windowComboBox=QComboBox()
        label2=QLabel("QPalette.WindowText")
        self.windowTextComboBox=QComboBox()
        label3=QLabel("QPalette.Button")
        self.buttonComboBox=QComboBox()
        label4=QLabel("QPalette.ButtonText")
        self.buttonTextComboBox=QComboBox()
        label5=QLabel("QPalette.Base")
        self.baseComboBox=QComboBox()

        self.fillColorList(self.windowComboBox)
        self.fillColorList(self.windowTextComboBox)
        self.fillColorList(self.buttonComboBox)
        self.fillColorList(self.buttonTextComboBox)
        self.fillColorList(self.baseComboBox)

    self.connect(self.windowComboBox, SIGNAL("currentIndexChanged(int)"), self.slotWindow)
    self.connect(self.windowTextComboBox, SIGNAL("currentIndexChanged(int)"), self.slo
```

```

tWindowText)

self.connect(self.buttonComboBox, SIGNAL("currentIndexChanged(int)"), self.slotButton)

self.connect(self.buttonTextComboBox, SIGNAL("currentIndexChanged(int)"), self.slotButtonText)

self.connect(self.baseComboBox, SIGNAL("currentIndexChanged(int)"), self.slotBase)

        gridLayout=QGridLayout()
        gridLayout.addWidget(label1, 0, 0)
        gridLayout.addWidget(self.windowComboBox, 0, 1)
        gridLayout.addWidget(label2, 1, 0)
        gridLayout.addWidget(self.windowTextComboBox, 1, 1)
        gridLayout.addWidget(label3, 2, 0)
        gridLayout.addWidget(self.buttonComboBox, 2, 1)
        gridLayout.addWidget(label4, 3, 0)
        gridLayout.addWidget(self.buttonTextComboBox, 3, 1)
        gridLayout.addWidget(label5, 4, 0)
        gridLayout.addWidget(self.baseComboBox)

        self.ctrlFrame.setLayout(gridLayout)

def fillColorList(self, comboBox):
    colorList=QColor.colorNames()

    for color in colorList:
        pix=QPixmap(QSize(70,20))
        pix.fill(QColor(color))
        comboBox.addItem(QIcon(pix), color)
        comboBox.setIconSize(QSize(70,20))
        comboBox.setSizeAdjustPolicy(QComboBox.AdjustToContents)

def createContentFrame(self):
    label1=QLabel(self.tr("请选择一个值"))
    valueComboBox=QComboBox()
    valueComboBox.addItem("1")
    valueComboBox.addItem("2")
    label2=QLabel(self.tr("请输入字符串"))
    stringLineEdit=QLineEdit()
    textEditText=QTextEdit(self.tr("请输入"))
    hLayout=QHBoxLayout()
    okButton=QPushButton(self.tr("确定"))
    cancelButton=QPushButton(self.tr("取消"))
    hLayout.addStretch()
    hLayout.addWidget(okButton)
    hLayout.addWidget(cancelButton)
    gridLayout=QGridLayout()
    gridLayout.addWidget(label1, 0, 0)
    gridLayout.addWidget(valueComboBox, 0, 1)
    gridLayout.addWidget(label2, 1, 0)
    gridLayout.addWidget(stringLineEdit, 1, 1)
    gridLayout.addWidget(textEditText, 2, 0, 1, 2)
    gridLayout.addLayout(hLayout, 3, 0, 1, 2)
    self.contentFrame.setLayout(gridLayout)

def slotWindow(self):
    colorList=QColor.colorNames()
    color=QColor(colorList[self.windowComboBox.currentIndex()])
    p=self.contentFrame.palette()
    p.setColor(QPalette.Window, color)
    self.contentFrame.setPalette(p)

```

```

def slotWindowText(self):
    colorList=QColor.colorNames()
    color=QColor(colorList[self.windowComboBox.currentIndex()])
    p=self.contentFrame.palette()
    p.setColor(QPalette.WindowText,color)
    self.contentFrame.setPalette(p)

def slotButton(self):
    colorList=QColor.colorNames()
    color=QColor(colorList[self.windowComboBox.currentIndex()])
    p=self.contentFrame.palette()
    p.setColor(QPalette.Button,color)
    self.contentFrame.setPalette(p)

def slotButtonText(self):
    colorList=QColor.colorNames()
    color=QColor(colorList[self.windowComboBox.currentIndex()])
    p=self.contentFrame.palette()
    p.setColor(QPalette.ButtonText,color)
    self.contentFrame.setPalette(p)

def slotBase(self):
    colorList=QColor.colorNames()
    color=QColor(colorList[self.windowComboBox.currentIndex()])
    p=self.contentFrame.palette()
    p.setColor(QPalette.Base,color)
    self.contentFrame.setPalette(p)

app=QApplication(sys.argv)
main=Palette()
main.show()
app.exec_()

```

其中，`createCtrlFrame()`函数完成窗体左半部分颜色选择区的创建，`createContentFrame()`函数完成右半部分的创建，`fillColorList()`函数完成向颜色下拉列表框中插入颜色的工作。

第 24 行创建一对 `QPalette.Window` 背景色的选择下拉列表框。

第 34 行调用 `fillColorList()`函数向下拉列表框中插入各种不同的颜色选项。

第 39 行选择下拉列表框的 `currentIndexChanged` 信号与改变背景色的槽函数 `slotWindow()`。

其他颜色选择控件的创建与此类似，这里不再重复。

第 93, 94 行获得当前选择的颜色值。

第 95 行调用 `QWidget` 类的 `palette()`函数，获得右部窗体 `contentFrame` 的调色板信息。

第 96 行调用 `QPalette` 类的 `setColor()`函数，设置 `contentFrame` 窗体的 `Window` 类颜色，即背景色，`setColor()`的第一个参数为设置的顏色主题，第二个参数为具体的颜色值。

第 97 行调用 `QWidget` 的 `setPalette()`把修改好的调色板信息应用回 `contentFrame` 窗体中，更新显示。

其他颜色选择响应槽函数与此函数类似。

`slotWindowText()`函数响应对文字颜色的选择，也即对前景色进行设置。

`slotButton()`函数响应对按钮背景色的选择。

`slotButtonText()`函数响应对按钮上文字颜色的选择。

`slotBase()`函数响应对可输入文本框背景色的选择。

`fillColorList()`函数用于插入颜色。

第 60 行调用 `QColor` 类的 `colorNames()` 函数获得 Qt 所有知道名称的颜色名称列表，返回的第一个字符串列表 `colorList`。

第 62-67 行对颜色名称列表进行遍历。

第 63 行新建一个 `QPixmap` 对象 `pix` 作为显示颜色的图标。

第 64 行为 `pix` 填充当前遍历的颜色。

第 65 行调用 `QComboBox` 的 `addItem()` 函数为下拉列表框插入一个条目，并以准备好的 `pix` 作为插入条目的图标，名称设为颜色的名称。

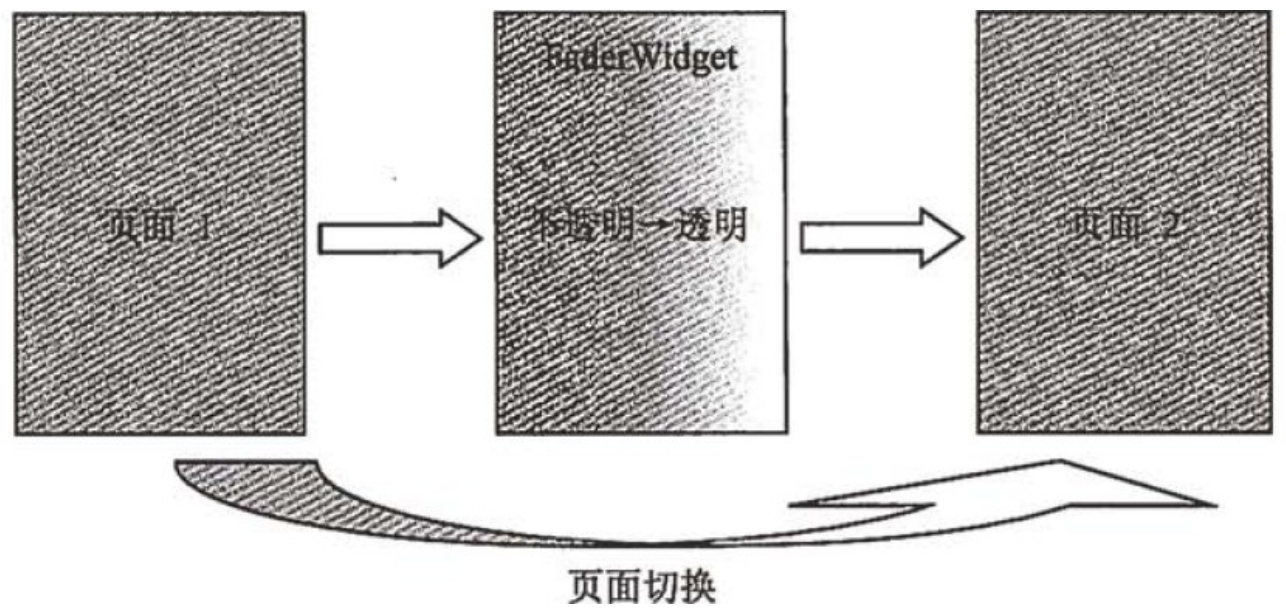
第 66 行设置图标的尺寸，图标默认尺寸是一个方形，把它设置为与 `pix` 相同尺寸的长方形。

第 67 行调用 `QComboBox` 的 `setSizeAdjustPolicy()` 设置下拉列表框的尺寸调整策略为 `AdjustToContents`，符合内容的大小。

PyQt4 精彩实例分析 实例 20 窗体的淡入淡出效果

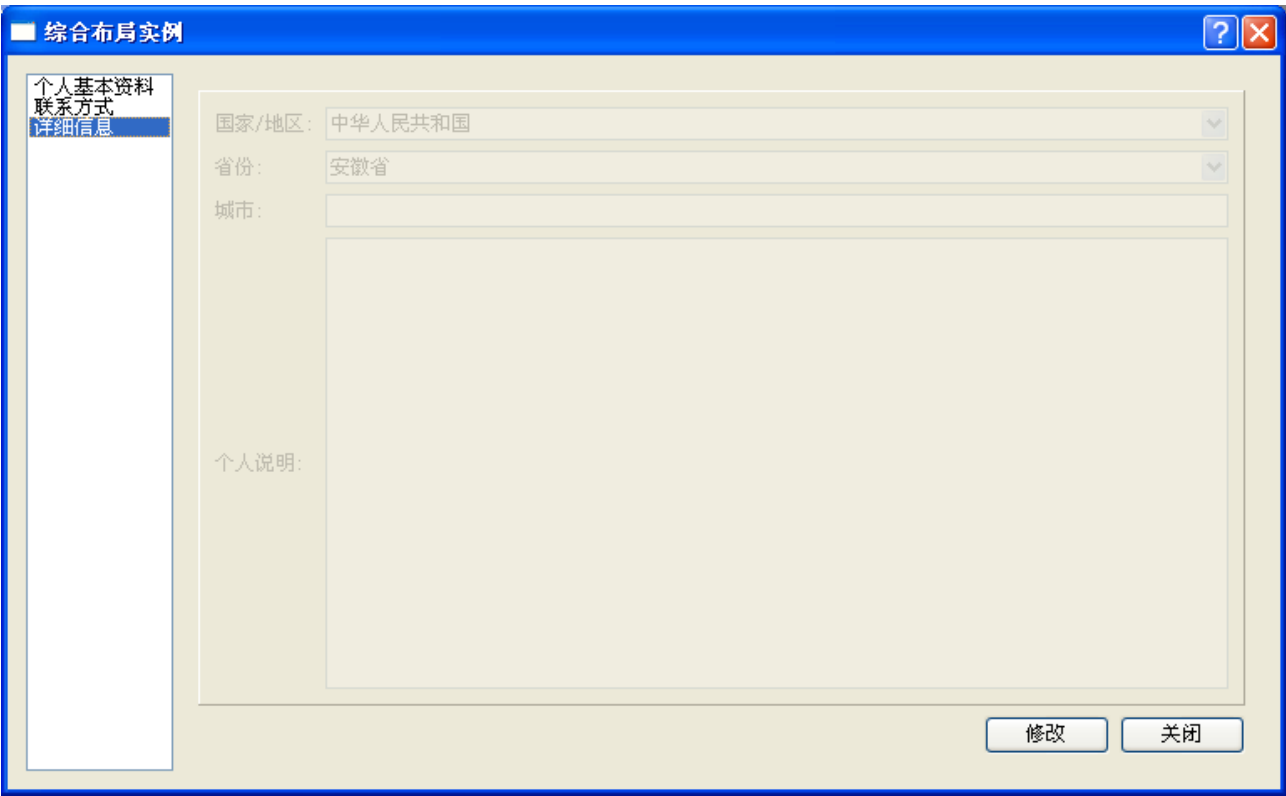
本实例实现一个窗体淡入淡出效果的例子，当窗体进行页面切换时，原页面的消失和新页面的显现并不是瞬间切换的，而是逐渐消隐和逐渐显现的过程。

本实例实现淡入淡出效果的基本原理可由下图描述。



当对话框由页面 1 切换至页面 2 时，在响应页面切换命令的同时，新建一个 **FaderWidget** 窗体，此窗体是一个与对话框等尺寸的空白窗体，此窗体由不透明逐渐变为完全透明，即实现页面的淡入淡出效果。

本实例将实例 17 的实现改造成淡入淡出效果，主对话框由一个列表框 **QListWidget** 和一个堆栈窗体 **QStackedWidget** 组成，列表框中列出了可显示的 3 个页面名称：“基本资料”，“联系方式”和“详细资料”，如下图所示。堆栈窗体中包含了对应的 3 个页面。



具体实现代码如下:

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class StockDialog(QDialog):
    def __init__(self, parent=None):
        super(StockDialog, self).__init__(parent)
        self.setWindowTitle(self.tr("综合布局实例"))

        mainSplitter=QSplitter(Qt.Horizontal)
        mainSplitter.setOpaqueResize(True)

        self.listWidget=QListWidget(mainSplitter)
        self.listWidget.insertItem(0, self.tr("个人基本资料"))
        self.listWidget.insertItem(1, self.tr("联系方式"))
        self.listWidget.insertItem(2, self.tr("详细信息"))

        frame=QFrame(mainSplitter)
        self.stack=QStackedWidget()
        self.stack.setFrameStyle(QFrame.Panel|QFrame.Raised)

        baseInfo=BaseInfo()
        contact=Contact()
        detail=Detail()
        self.stack.addWidget(baseInfo)
        self.stack.addWidget(contact)
        self.stack.addWidget(detail)

        amendPushButton=QPushButton(self.tr("修改"))
        closePushButton=QPushButton(self.tr("关闭"))

        buttonLayout=QHBoxLayout()
        buttonLayout.addStretch(1)
        buttonLayout.addWidget(amendPushButton)
        buttonLayout.addWidget(closePushButton)

        mainLayout=QVBoxLayout(frame)
        mainLayout.setMargin(10)
        mainLayout.setSpacing(6)
        mainLayout.addWidget(self.stack)
        mainLayout.addLayout(buttonLayout)

self.connect(self.listWidget, SIGNAL("currentRowChanged(int)"), self.stack, SLOT("setCurrentIndex(int)"))

self.connect(closePushButton, SIGNAL("clicked()"), self, SLOT("close()"))

        layout=QHBoxLayout(self)
        layout.addWidget(mainSplitter)
        self.setLayout(layout)
        #例 20 代码开始-----
        self.faderWidget=None

self.connect(self.listWidget, SIGNAL("currentItemChanged(QListWidgetItem, QListWidgetItem")
```

```

),
        self.changePage)

self.connect(self.stack, SIGNAL("currentChanged(int)"), self.fadeInWidget)

def changePage(self, current, previous):
    if not current:
        current=previous
    self.stack.setCurrentWidget(current)

def fadeInWidget(self, index):
    self.faderWidget=FaderWidget(self.stack.widget(index))
    self.faderWidget.start()

class FaderWidget(QWidget):
    def __init__(self, parent=None):
        super(FaderWidget, self).__init__(parent)

        if parent:
            self.startColor=parent.palette().window().color()
        else:
            self.startColor=Qt.White

        self.currentAlpha=0
        self.duration=1000

        self.timer=QTimer(self)
        self.connect(self.timer, SIGNAL("timeout()"), self.update)
        self.setAttribute(Qt.WA_DeleteOnClose)
        self.resize(parent.size())

    def start(self):
        self.currentAlpha=255
        self.timer.start(100)
        self.show()

    def paintEvent(self, event):
        semiTransparentColor=self.startColor
        semiTransparentColor.setAlpha(self.currentAlpha)
        painter=QPainter(self)
        painter.fillRect(self.rect(), semiTransparentColor)
        self.currentAlpha-=(255*self.timer.interval()/self.duration)

        if self.currentAlpha<=0:
            self.timer.stop()
            self.close()

#例 20 代码结束-----
class BaseInfo(QWidget):
    def __init__(self, parent=None):
        super(BaseInfo, self).__init__(parent)

        label1=QLabel(self.tr("用户名:"))
        label2=QLabel(self.tr("姓名: "))
        label3=QLabel(self.tr("性别:"))
        label4=QLabel(self.tr("部门:"))
        label5=QLabel(self.tr("年龄:"))
        otherLabel=QLabel(self.tr("备注:"))
        otherLabel.setFrameStyle(QFrame.Panel|QFrame.Sunken)
        userLineEdit=QLineEdit()
        nameLineEdit=QLineEdit()
        sexComboBox=QComboBox()
        sexComboBox.insertItem(0, self.tr("男"))

```



```

sexComboBox.insertItem(1, self.tr("女"))
departmentTextEdit=QTextEdit()
ageLineEdit=QLineEdit()

labelCol=0
contentCol=1

leftLayout=QGridLayout()
leftLayout.addWidget(label1, 0, labelCol)
leftLayout.addWidget(userLineEdit, 0, contentCol)
leftLayout.addWidget(label2, 1, labelCol)
leftLayout.addWidget(nameLineEdit, 1, contentCol)
leftLayout.addWidget(label3, 2, labelCol)
leftLayout.addWidget(sexComboBox, 2, contentCol)
leftLayout.addWidget(label4, 3, labelCol)
leftLayout.addWidget(departmentTextEdit, 3, contentCol)
leftLayout.addWidget(label5, 4, labelCol)
leftLayout.addWidget(ageLineEdit, 4, contentCol)
leftLayout.addWidget(otherLabel, 5, labelCol, 1, 2)
leftLayout.setColumnStretch(0, 1)
leftLayout.setColumnStretch(1, 3)

label6=QLabel(self.tr("头像:"))
iconLabel=QLabel()
icon=QPixmap("image/2.jpg")
iconLabel.setPixmap(icon)
iconLabel.resize(icon.width(), icon.height())
iconPushButton=QPushButton(self.tr("改变"))
hLayout=QHBoxLayout()
hLayout.setSpacing(20)
hLayout.addWidget(label6)
hLayout.addWidget(iconLabel)
hLayout.addWidget(iconPushButton)

label7=QLabel(self.tr("个人说明:"))
descTextEdit=QTextEdit()

rightLayout=QVBoxLayout()
rightLayout.setMargin(10)
rightLayout.addLayout(hLayout)
rightLayout.addWidget(label7)
rightLayout.addWidget(descTextEdit)
mainLayout=QGridLayout(self)
mainLayout.setMargin(15)
mainLayout.setSpacing(10)
mainLayout.addLayout(leftLayout, 0, 0)
mainLayout.addLayout(rightLayout, 0, 1)
mainLayout.setSizeConstraint(QLayout.SetFixedSize)

```

```

class Contact(QWidget):
    def __init__(self, parent=None):
        super(Contact, self).__init__(parent)
        label1=QLabel(self.tr("电子邮件:"))
        label2=QLabel(self.tr("联系地址:"))
        label3=QLabel(self.tr("邮政编码:"))
        label4=QLabel(self.tr("移动电话:"))
        label5=QLabel(self.tr("办公电话:"))

        mailLineEdit=QLineEdit()
        addressLineEdit=QLineEdit()
        codeLineEdit=QLineEdit()
        mpLineEdit=QLineEdit()
        phoneLineEdit=QLineEdit()

```

```

receiveCheckBox=QCheckBox(self.tr("接收留言"))

layout=QGridLayout(self)
layout.addWidget(label1,0,0)
layout.addWidget(mailLineEdit,0,1)
layout.addWidget(label2,1,0)
layout.addWidget(addressLineEdit,1,1)
layout.addWidget(label3,2,0)
layout.addWidget(codeLineEdit,2,1)
layout.addWidget(label4,3,0)
layout.addWidget(mpLineEdit,3,1)
layout.addWidget(receiveCheckBox,3,2)
layout.addWidget(label5,4,0)
layout.addWidget(phoneLineEdit,4,1)

class Detail(QWidget):
    def __init__(self,parent=None):
        super(Detail,self).__init__(parent)
        label1=QLabel(self.tr("国家/地区:"))
        label2=QLabel(self.tr("省份:"))
        label3=QLabel(self.tr("城市:"))
        label4=QLabel(self.tr("个人说明:"))

        countryComboBox=QComboBox()
        countryComboBox.addItem(self.tr("中华人民共和国"))
        countryComboBox.addItem(self.tr("香港"))
        countryComboBox.addItem(self.tr("台北"))
        countryComboBox.addItem(self.tr("澳门"))
        provinceComboBox=QComboBox()
        provinceComboBox.addItem(self.tr("安徽省"))
        provinceComboBox.addItem(self.tr("北京市"))
        provinceComboBox.addItem(self.tr("江苏省"))
        cityLineEdit=QLineEdit()
        remarkTextEdit=QTextEdit()

        layout=QGridLayout(self)
        layout.addWidget(label1,0,0)
        layout.addWidget(countryComboBox,0,1)
        layout.addWidget(label2,1,0)
        layout.addWidget(provinceComboBox,1,1)
        layout.addWidget(label3,2,0)
        layout.addWidget(cityLineEdit,2,1)
        layout.addWidget(label4,3,0)
        layout.addWidget(remarkTextEdit,3,1)

app=QApplication(sys.argv)
main=StockDialog()
main.show()
app.exec_()

```

FaderWidget 继承自 QWidget，包含一个 start() 函数，调用 start() 函数即开始 FaderWidget 窗体的渐变过程，重新实现了 paintEvent() 函数，并声明了一个定时器变量定时调用 paintEvent() 函数。

第 71-74 行首先判断是否有父窗体存在，若有父窗体，则设置起始窗体颜色为父窗体的背景色，若没有父窗体，则设置起始窗体颜色为白色。

第 76 行设置透明度的初始值。

第 77 行设置渐变时长为 1000 毫秒。

第 79 行创建一个定时对象。

第 80 行连接定时器响应 timeout() 信号，每一间隔时间结束后调用一次 update() 函数重画窗体。

第 81 行设置 `FaderWidget` 的窗体属性为 `Qt.WA_DeleteOnClose`，当整个对话框关闭时，此渐变窗体也同时关闭。

第 82 行设置 `FaderWidget` 窗体的尺寸为父窗体的大小。

`FaderWidget` 的 `start()` 函数显示此窗体并开始渐变。首先设置窗体显示的最初透明度为 255，即不透明，启动定时器，以 100 毫秒为周期进行重画，最后调用 `show()` 函数显示此窗体。

第 90 行用一个 `QColor` 对象保存重绘窗体的颜色，设置为 `startColor`，即父窗体的颜色或白色。

第 91 行设置此颜色的透明度。

第 92 行以 `FaderWidget` 窗体创建一个 `QPainter` 对象。

第 93 行利用此颜色填充整个窗体。

第 94 行修改 `currentAlpha` 值，第重绘一次透明度降低一定的值，直到透明度为 0，即完全透明。

第 96-98 行对透明度的值进行判断，若透明度已小于或等于零，则停止定时器，也即不再调用重画命令，并关闭此 `FaderWidget` 窗体。

完成 `FaderWidget` 渐变窗体后，需在主对话框实现的合适位置和时机调用此渐变窗体以实现淡入淡出效果。

在 `StackDialog` 类中声明了两个槽函数，一个 `changePage()` 完成页面切换的工作，一个 `fadeInWidget()` 完成页面切换时的淡入淡出效果。

第 54 行的 `connect` 函数，为列表框 `currentItemChanged()` 信号连接响应槽函数 `changePage()`，即当用户选择下拉列表框中的条目时，调用 `changePage()` 更改右侧堆栈窗中的显示页面。

第 56 行的 `connect` 函数，为堆栈窗的 `currentChanged()` 信号连接响应槽函数 `fadeInWidget()`，即当堆栈窗的页面发生改变时调用 `fadeInWidget()` 来实现页面的淡入淡出效果。

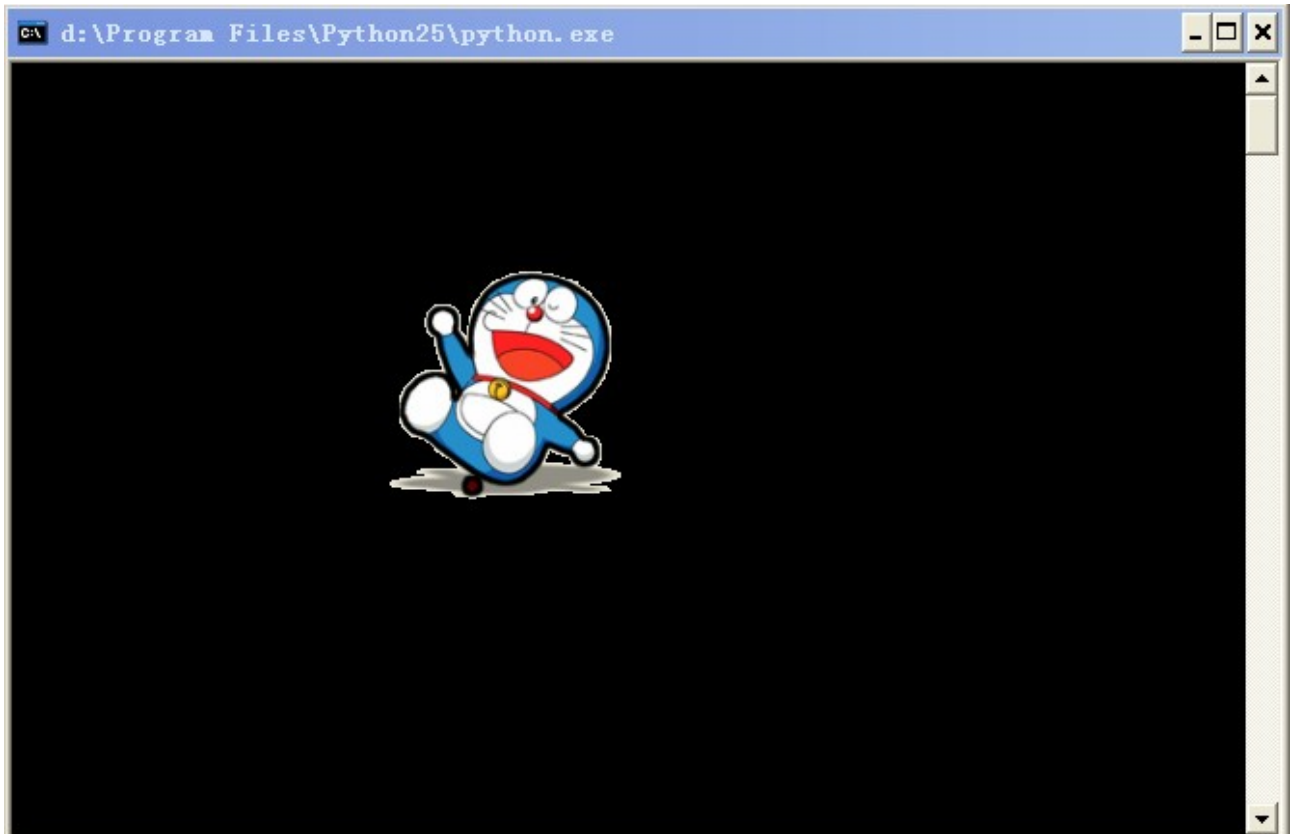
`changePage()` 槽函数主要完成页面的切换工作。

`fadeInWidget()` 槽函数在堆栈窗的页面发生变化时被调用，是实现页淡入淡出的关键函数，函数首先判断是否已有渐变窗体对象存在，若已有则关闭已存在的渐变窗体，再以堆栈窗的当前窗体为父窗口创建渐变窗体对象，并调用 `start()` 函数开始窗体渐变，实现淡入淡出效果。

PyQt4 精彩实例分析 实例 21 不规则窗体

常见的窗体通常是各种方形的对话框，如前面实例中实现的所有对话框都是这样的。但有时也会需要用到非方形的窗体，如圆形，椭圆形甚至是不规则形状的对话框。

本实例即实现了一个对 PNG 图形外沿为形状的不规则形状对话框，如下图所示。



在图中所示的哆啦 A 梦即为一个不规则窗体，实例在不规则窗体中绘制了作为窗体形状的 PNG 图片，也可在不规则窗体上放置按钮等控件，可以通过鼠标左键拖动窗体，鼠标右键关闭窗口体。

具体实现代码如下：

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

class ShapeWidget(QWidget):
    def __init__(self, parent=None):
        super(ShapeWidget, self).__init__(parent)

        pix=QPixmap("image/21.png", "0", Qt.AvoidDither|Qt.ThresholdDither|
Qt.ThresholdAlphaDither)
        self.resize(pix.size())
        self.setMask(pix.mask())
        self.dragPosition=None

    def mousePressEvent(self, event):
        if event.button()==Qt.LeftButton:
            self.dragPosition=event.globalPos()-
self.frameGeometry().topLeft()
            event.accept()
        if event.button()==Qt.RightButton:
            self.close()
```

```

def mousePressEvent(self, event):
    if event.buttons() & Qt.LeftButton:
        self.move(event.globalPos()-self.dragPosition)
        event.accept()

def paintEvent(self, event):
    painter=QPainter(self)
    painter.drawPixmap(0,0,QPixmap("image/21.png"))

app=QApplication(sys.argv)
form=ShapeWidget()
form.show()
app.exec_()

```

ShapeWidget 即为此不规则窗体类，继承自 QWidget 类。在类中重定义的鼠标事件 mousePressEvent(), mouseMoveEvent() 以及绘制函数 paintEvent(), 使不规则窗体能用鼠标随意拖动。

第 9 行新建一个 QPixmap 对象。

第 10 行重设主窗体的尺寸为所读取的图片的大小。

第 11 行的 setMask() 命令是实现不规则窗体的关键，setMask() 的作用是为调用它的控件增加一个遮罩，遮住所选区域以外的部分使之看起来是透明的，它的参数可为一个 QPixmap 对象或一个 QRegion 对象，此处调用 QPixmap 的 mask() 函数获得图片自身的遮罩，为一个 QPixmap 对象。实例中使用的是 png 格式的图片，它的透明部分实际上即是一个遮罩。

重定义鼠标按下响应函数 mousePressEvent(QMouseEvent) 和鼠标移动响应函数 mouseMoveEvent(QMouseEvent)，使不规则窗体能响应鼠标事件，随意拖动。

在 mousePressEvent 函数中，首先判断按下的是否为鼠标左键，若是则保存当前鼠标点所在的位置相对于窗体左上角的偏移值 dragPosition，如果按下鼠标右键，则关闭窗口体。

此处的 frameGeometry().topLeft() 仍然表示整个窗体的左上角，而并不是所见的不规则窗体的左上角。

在 mouseMoveEvent 函数中，首先判断当前鼠标状态，调用 event.buttons() 返回鼠标的状态，若为左侧按钮则调用 QWidget 的 move() 函数把窗体移动至鼠标当前点，由于 move() 函数的参数指的是窗体的左上角的位置，因此要用鼠标当前点的位置减去相对窗体左上角的偏移值 dragPosition。

ShapeWidget 的重画函数主要完成在窗体上绘制图片的工作，此处为方便显示在窗体上绘制的即是用来确定窗体外形的 PNG 图片。

PyQt4 精彩实例分析 实例 22 电子钟

本实例实现一个数字电子钟程序，效果图如下图显示于桌面上，并可随意拖动至桌面任意位置。



具体实现代码如下：

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

class DigiClock(QLCDNumber):
    def __init__(self, parent=None):
        super(DigiClock, self).__init__(parent)

        p=self.palette()
        p.setColor(QPalette.Window, Qt.blue)
        self.setPalette(p)

        self.dragPosition=None

        self.setWindowFlags(Qt.FramelessWindowHint)
        self.setWindowOpacity(0.5)

        timer=QTimer(self)
        self.connect(timer, SIGNAL("timeout()"), self.showTime)
        timer.start(1000)

        self.showColon=True
        self.showTime()
        self.resize(150, 60)

    def mousePressEvent(self, event):
        if event.button()==Qt.LeftButton:
```

```

        self.dragPosition=event.globalPos()-
self.frameGeometry().topLeft()
        event.accept()
    if event.button()==Qt.RightButton:
        self.close()

    def mouseMoveEvent(self,event):
        if event.buttons() & Qt.LeftButton:
            self.move(event.globalPos()-self.dragPosition)
            event.accept()

    def showTime(self):
        time=QTime.currentTime()
        text=time.toString("hh:mm")
        if self.showColon:
            text.replace(2,1,":")
            self.showColon=False
        else:
            text.replace(2,1," ")
            self.showColon=True
        self.display(text)

app=QApplication(sys.argv)
form=DigiClock()
form.show()
app.exec_()

```

数字电子钟类 **DigiClock** 继承自 **QLCDNumber** 类，类中重定义了鼠标按下事件和鼠标移动事件以使电子钟可随意拖动；**showTime()** 函数用于显示当前的时间；**dragPosition** 用于保存鼠标点相对电子钟窗体左上角的偏移值；**showColon** 表示显示时间时是否显示“:”。

第 9-11 行完成电子钟窗体背景色的设置，在前面的实例中已经分析过利用调色板类 **QPalette** 进行控件颜色控制的方法，此处设置背景色为蓝色。

第 15 行调用 **QWidget** 类的 **setWindowFlags()** 函数设置窗体的标识，此处设置为 **Qt.FramelessWindowHint**，表示此窗体为一个没有面板边框和标题栏的窗体。

第 16 行调用 **setWindowOpacity()** 函数设置窗体的透明度为 **0.5**，即半透明，但此函数在 **X11** 系统中不起作用，当程序在 **windows** 系统下运行时，此函数起作用，即电子钟半透明显示。

第 18-20 行完成电子钟中用于时间显示的定时器部分。

第 18 行创建一个定时器对象。

第 19 行连接定时器的 **timeout()** 信号与显示时间的槽函数 **showTime()**。

第 20 行调用 **start()** 函数以 **1000** 毫秒为周期启动定时器。

第 23 行调用一次 **showTime()** 函数，初始时间显示。

第 24 行初始化 **showColon** 为 **True**。

DigiClock 的 **showTime()** 函数完成电子钟的显示时间的功能。

第 39 行调用 **QTime** 的 **currentTime()** 函数获取当前的系统时间，保存在一个 **QTime** 对象中。

第 40 行调用 **QTime** 的 **toString()** 函数把获取的当前时间转换成字符串类型。为便于显示，**toString()** 函数的参数需指定转换后时间的显示格式。

H/h: 小时（若使用 **H** 表示小时，则无论何时都以 **24** 小时制显示小时；若使用 **h** 表示小时，则当同时指定 **AM/PM** 时，采用 **12** 小时制显示小时，其他情况下仍采用 **24** 小时制进行显示）。

m: 分钟。

s: 秒钟。

AP/A: 显示 AM 或 PM。

Ap/a: 显示 am 或 pm。

可根据实际显示需要进行格式设置，如：

hh:mm:ss A 22:30:08 PM

H:mm:s a 10:30:8 pm

QTime 的 toString() 函数也可直接利用 Qt.DateFormat 作为参数指定时间显示的格式，如：
Qt.TextDate, Qt.ISODate, Qt.LocaleDate 等。

第 41-46 行完成电子钟“时”与“分”之前的表示秒钟的两个点的闪现功能。

第 47 行调用 QLCDNumber 类的 display() 函数显示转换好的字符串时间。

鼠标按下事件响应函数 mousePressEvent(QMouseEvent) 和鼠标移动事件响应函数
mouseMoveEvent(QMouseEvent) 的重定义使电子钟能用鼠标在桌面上随意拖动。

鼠标按下响应函数中，首先判断按下的是否为鼠标左键，若是则保存当前鼠标点所在的位置相对于窗体左上角的偏移值 dragPosition；如果按下鼠标右键，则退出窗体。

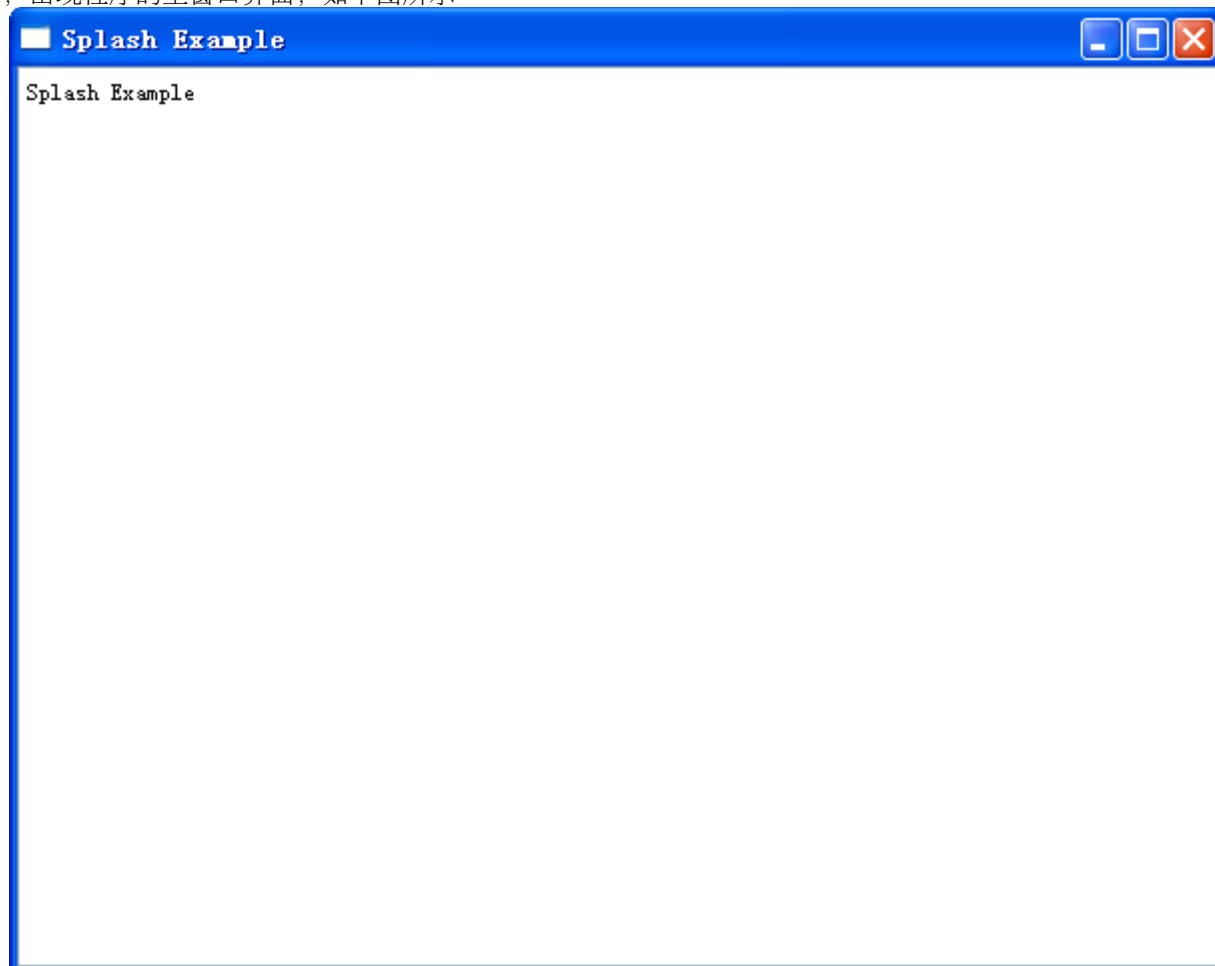
在鼠标移动响应函数中，首先判断当前鼠标状态，调用 e.buttons() 返回鼠标的状态，若为左侧按钮则调用 QWidget 的 move() 函数把窗体移动至鼠标当前点。由于 move() 函数的参数指的是窗体的左上角的位置，因此要用鼠标当前点的位置减去相对窗体左上角的偏移值 dragPosition。

PyQt4 精彩实例分析 实例 23 程序启动画面

大多数应用程序启动时都会在程序完全启动时显示一个启动画面，在程序完全启动后消失。程序启动画面可以显示一些有关产品的信息，让用户在等待程序启动的同时了解有关产品的功能，也是一个宣传的方式。

QSplashScreen 类提供了在程序启动过程中显示的启动画面的功能。本实例实现一个出现程序启动画面的例子。

当运行程序时，在显示屏的中央出现一个启动画面，经过一段时间，应用程序完成初始化工作后，启动画面隐去，出现程序的主窗口界面，如下图所示。



具体实现代码如下：

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle("Splash Example")
        edit=QTextEdit()
        edit.setText("Splash Example")
        self.setCentralWidget(edit)

        self.resize(600, 450)

        QThread.sleep(3)
```

```
app=QApplication(sys.argv)

splash=QSplashScreen(QPixmap("image/23.png"))
splash.show()
app.processEvents()
window=MainWindow()
window.show()
splash.finish(window)

app.exec_()
```

第 19 行利用 `QPixmap` 对象创建一个 `QSplashScreen` 对象。

第 20 行调用 `show()` 函数显示此启动图片。

第 21 行调用 `processEvents()` 使程序在显示启动画面的同时仍能响应鼠标等其他事件。

第 22, 23 行正常创建主窗体对象，并调用 `show()` 函数显示。

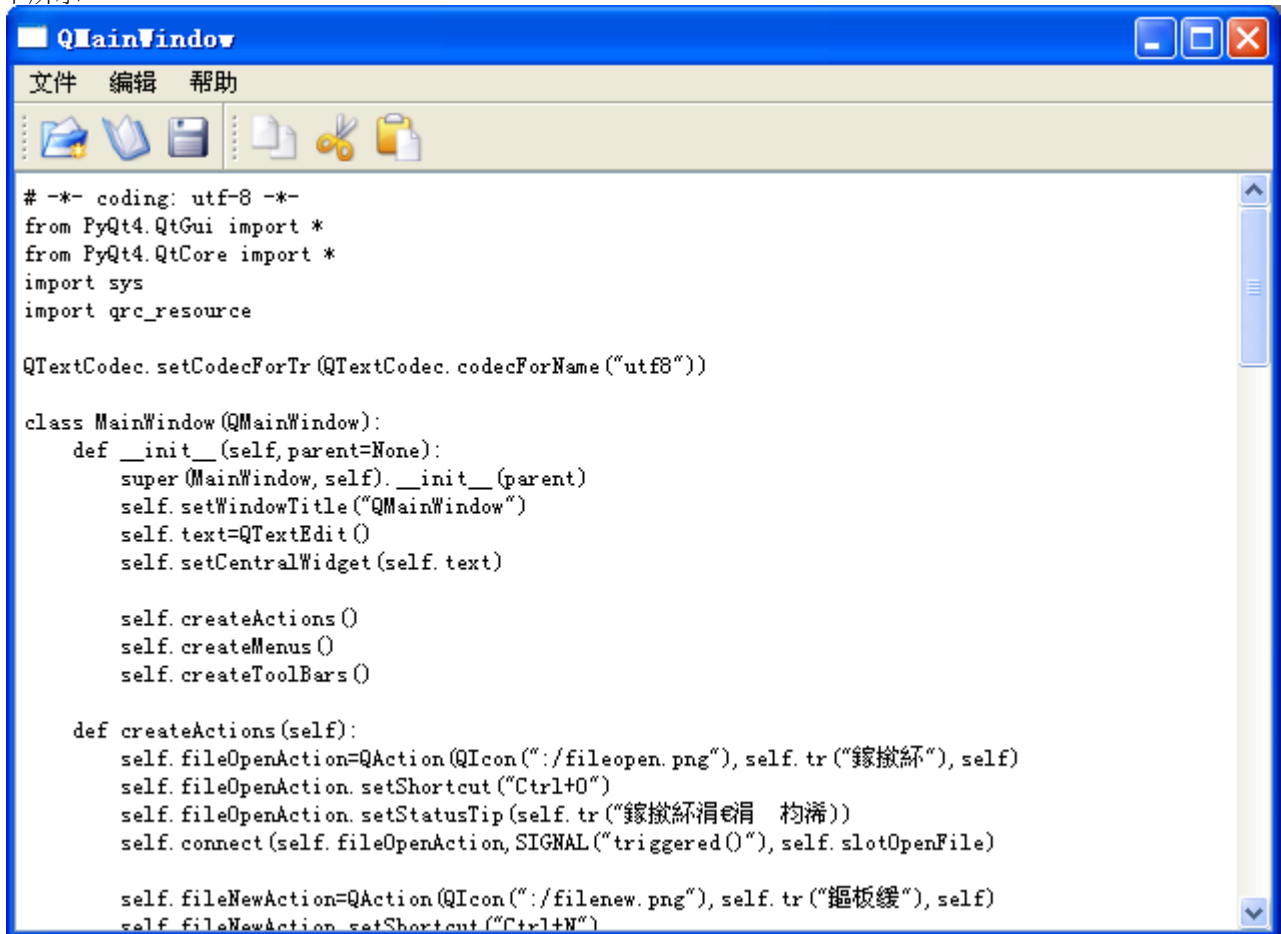
第 24 行调用 `QSplashScreen` 类的 `finish()` 函数，表示在主窗体对象初始化完成后，结束启动画面。

第 26 行正常地调用 `exec_()` 函数。

由于启动画面一般在当程序初始化时间较长的情况下出现，本实例中为了使程序初始化时间加长以显示启动画面，在主窗体的构造函数中调用 `QThread.sleep()` 函数，使主窗口程序在初始化时休眠几秒钟。

PyQt4 精彩实例分析 实例 24 基本 QMainWindow 主窗口程序

本实例实现一个基本主窗口程序，包含一个菜单条，一个工具栏，中央可编辑窗体及状态栏。实现的效果图如下所示。



具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
import qrc_resource

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle("QMainWindow")
        self.text=QTextEdit()
        self.setCentralWidget(self.text)

        self.createActions()
        self.createMenus()
        self.createToolBars()

    def createActions(self):
        self.fileOpenAction=QAction(QIcon(":/fileopen.png"), self.tr("打开"), self)
        self.fileOpenAction.setShortcut("Ctrl+O")
        self.fileOpenAction.setStatusTip(self.tr("打开一个文件"))
        self.connect(self.fileOpenAction, SIGNAL("triggered()"), self.slotOpenFile)

        self.fileNewAction=QAction(QIcon(":/filenew.png"), self.tr("新建"), self)
        self.fileNewAction.setShortcut("Ctrl+N")
```

```

self.connect(self.fileOpenAction, SIGNAL("triggered()"), self.slotOpenFile)

        self.fileNewAction=QAction(QIcon(":/filenew.png"), self.tr("新建"), self)
        self.fileNewAction.setShortcut("Ctrl+N")
        self.fileNewAction.setStatusTip(self.tr("新建一个文件"))

self.connect(self.fileNewAction, SIGNAL("triggered()"), self.slotNewFile)

        self.fileSaveAction=QAction(QIcon(":/filesave.png"), self.tr("保存"), self)
        self.fileSaveAction.setShortcut("Ctrl+S")
        self.fileSaveAction.setStatusTip(self.tr("保存文件"))

self.connect(self.fileSaveAction, SIGNAL("triggered()"), self.slotSaveFile)

        self.exitAction=QAction(QIcon(":/filequit.png"), self.tr("退出"), self)
        self.exitAction.setShortcut("Ctrl+Q")
        self.setStatusTip(self.tr("退出"))
        self.connect(self.exitAction, SIGNAL("triggered()"), self.close)

        self.cutAction=QAction(QIcon(":/editcut.png"), self.tr("剪切"), self)
        self.cutAction.setShortcut("Ctrl+X")
        self.cutAction.setStatusTip(self.tr("剪切到粘贴板"))
        self.connect(self.cutAction, SIGNAL("triggered()"), self.text.cut)

        self.copyAction=QAction(QIcon(":/editcopy.png"), self.tr("复制"), self)
        self.copyAction.setShortcut("Ctrl+C")
        self.copyAction.setStatusTip(self.tr("复制到粘贴板"))
        self.connect(self.copyAction, SIGNAL("triggered()"), self.text.copy)

        self.pasteAction=QAction(QIcon(":/editpaste.png"), self.tr("粘贴"), self)
        self.pasteAction.setShortcut("Ctrl+V")
        self.pasteAction.setStatusTip(self.tr("粘贴内容到当前处"))
        self.connect(self.pasteAction, SIGNAL("triggered()"), self.text.paste)

        self.aboutAction=QAction(self.tr("关于"), self)
        self.connect(self.aboutAction, SIGNAL("triggered()"), self.slotAbout)

def createMenus(self):
    fileMenu=self.menuBar().addMenu(self.tr("文件"))
    fileMenu.addAction(self.fileNewAction)
    fileMenu.addAction(self.fileOpenAction)
    fileMenu.addAction(self.fileSaveAction)
    fileMenu.addAction(self.exitAction)

    editMenu=self.menuBar().addMenu(self.tr("编辑"))
    editMenu.addAction(self.copyAction)
    editMenu.addAction(self.cutAction)
    editMenu.addAction(self.pasteAction)

    aboutMenu=self.menuBar().addMenu(self.tr("帮助"))
    aboutMenu.addAction(self.aboutAction)

def createToolBars(self):
    fileToolBar=self.addToolBar("File")
    fileToolBar.addAction(self.fileNewAction)
    fileToolBar.addAction(self.fileOpenAction)
    fileToolBar.addAction(self.fileSaveAction)

    editTool=self.addToolBar("Edit")
    editTool.addAction(self.copyAction)

```

```

        editTool.addAction(self.cutAction)
        editTool.addAction(self.pasteAction)

    def slotNewFile(self):
        newWin=MainWindow()
        newWin.show()

    def slotOpenFile(self):
        fileName=QFileDialog.getOpenFileName(self)
        if fileName.isEmpty()==False:
            if self.text.document().isEmpty():
                self.loadFile(fileName)
            else:
                newWin=MainWindow()
                newWin.show()
                newWin.loadFile(fileName)

    def loadFile(self, fileName):
        file=QFile(fileName)
        if file.open(QIODevice.ReadOnly|QIODevice.Text):
            textStream=QTextStream(file)
            while textStream.atEnd()==False:
                self.text.append(textStream.readLine())

    def slotSaveFile(self):
        pass

    def slotAbout(self):
        QMessageBox.about("about me", self.tr("这是我们的第一个例子"))

app=QApplication(sys.argv)
main=MainWindow()
main.show()
app.exec_()

```

`createActions()`函数用于创建所有的动作，`createMenus()`函数用于创建菜单，`createToolBars()`函数用于创建工具栏。

第 13 行新建一个 `QTextEdit` 对象。

第 14 行把 `QTextEdit` 作为主窗口的中央窗体。

第 16 行调用创建动作的函数。

第 17 行调用创建菜单的函数。

第 18 行调用创建工具栏的函数。

菜单与工具栏都与 `QAction` 类密切相关，工具栏上的功能按钮与菜单中的选项条目相对应，完成相同的功能，使用相同的快捷键与图标。`QAction` 类为用户提供了一个统一的命令接口，无论是从菜单触发还是从工具栏触发，或快捷键触发都调用同样的操作接口，达到同样的目的。

第 21-24 行实现的是“打开文件”动作，第 21 行在创建这个动作时，指定了此动作使用的图标，名称以及父窗口。

第 22 行设置了此动作的快捷键为 `Ctrl+O`。

第 23 行设定了状态条显示，当鼠标光标移至动作对应的菜单条目或工具栏按钮上时，在状态条上显示“打开文件”的提示。

第 24 行连接此动作触发时所调用的槽函数 `slotOpenFile()`。

第 21 行中指定动作所使用的图标时，使用的是 `QIcon(":/fileopen.png")`，需要在此程序之外新建一个

QRC 的资源文件: `resources.qrc`

文件内容如下:

```
<!DOCTYPE RCC>
<RCC version="1.0">
<qresource>
<file alias="icon.png">images/icon.png</file>
<file alias="filenew.png">images/filenew.png</file>
<file alias="fileopen.png">images/fileopen.png</file>
<file alias="filesave.png">images/filesave.png</file>
<file alias="filesaveas.png">images/filesaveas.png</file>
<file alias="fileclose.png">images/filequit.png</file>
<file alias="filequit.png">images/filequit.png</file>
<file alias="editcut.png">images/editcut.png</file>
<file alias="editcopy.png">images/editcopy.png</file>
<file alias="editpaste.png">images/editpaste.png</file>
<file alias="editadd.png">images/editadd.png</file>
<file alias="editedit.png">images/editedit.png</file>
<file alias="editdelete.png">images/editdelete.png</file>
</qresource>
</RCC>
```

同时 `images` 文件夹要与同一目录下。

然后使用如下命令, 生成一个 `qrc_resources.py` 文件, 然后在程序中导入。

```
pyrcc4 -o qrc_resources.py resources.qrc
```

这些内容, 在以后我会拿出来专门解释。

“剪切”, “复制”和“粘贴”动作连接的触发响应槽函数, 分别直接使用 `QTextEdit` 对象的 `cut()`, `copy()` 和 `paste()` 函数即可。

第 57 行“关于”动作的触发响应槽函数使用的是 `QApplication` 的 `slotabout()`。

在创建动作时, 也可不指定图标, 这类动作一般只在菜单中出现, 而不在工具栏上使用。

第 60-64 行实现文件菜单, `Qt4` 的菜单实现与 `Qt3` 有所不同, 简化了菜单的实现过程。

第 60 行直接调用 `QMainWindow` 的 `menuBar()` 函数即可得到主窗口的菜单条指针, 再调用菜单条 `QMenuBar` 的 `addMenu()` 函数, 即完成菜单条中插入一个新菜单 `fileMenu`, `fileMenu` 为一个 `QMenu` 类对象。

第 61-64 行调用 `QMenu` 的 `addAction()` 函数在菜单中加入菜单条目“打开”, “新建”, “保存”和“退出”。

第 66-69 行实现编辑菜单。

第 70-71 行实现帮助菜单。

第 75-78 行实现文件工具栏, 第 80-83 实现编辑工具栏。

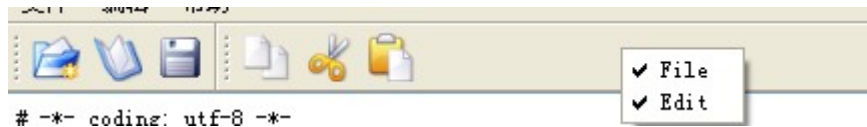
主窗口的工具栏上可以有多个工具条, 一般采用一个菜单对应一个工具条的方式, 也可根据需要进行工具条的划分。

第 75 行直接调用 `QMainWindow` 的 `addToolBar()` 函数即可获得主窗口的工具条对象, 每新建一个工具条调用一次 `addToolBar()` 函数, 赋予不同的名称, 即可在主窗口中新增一个工具条。

第 76-78 行调用 `QToolBar` 的 `addAction()` 函数在工具条中插入属于本工具条的动作。

第 80-83 行编辑工具条的实现与文件工具条类似。

两个工具条的显示可以由用户进行选择，在工具栏上单击鼠标右键将弹出工具条显示的选择菜单，如下图所示。



用户对需要显示的工具条进行选择。

工具条是一个可移动的窗口，它可停靠的区域由 `QToolBar` 的 `allowAreas` 决定，包括 `Qt.LeftToolBarArea`, `Qt.RightToolBarArea`, `Qt.TopToolBarArea`, `Qt.BottomToolBarArea` 和 `Qt.AllToolBarAreas`。

默认为 `Qt.AllToolBarAreas`，启动默认出现于主窗口的顶部。

可通过调用 `setAllowAreas()` 函数来指定工具条可依靠的区域，如：

```
fileToolBar.setAllowAreas(Qt.TopToolBarArea|Qt.LeftToolBarArea)
```

此函数限定文件工具条只可出现在主窗口的顶部或左侧。

工具条也可通过调用 `setMovable()` 函数设定工具条的可移动性，如：

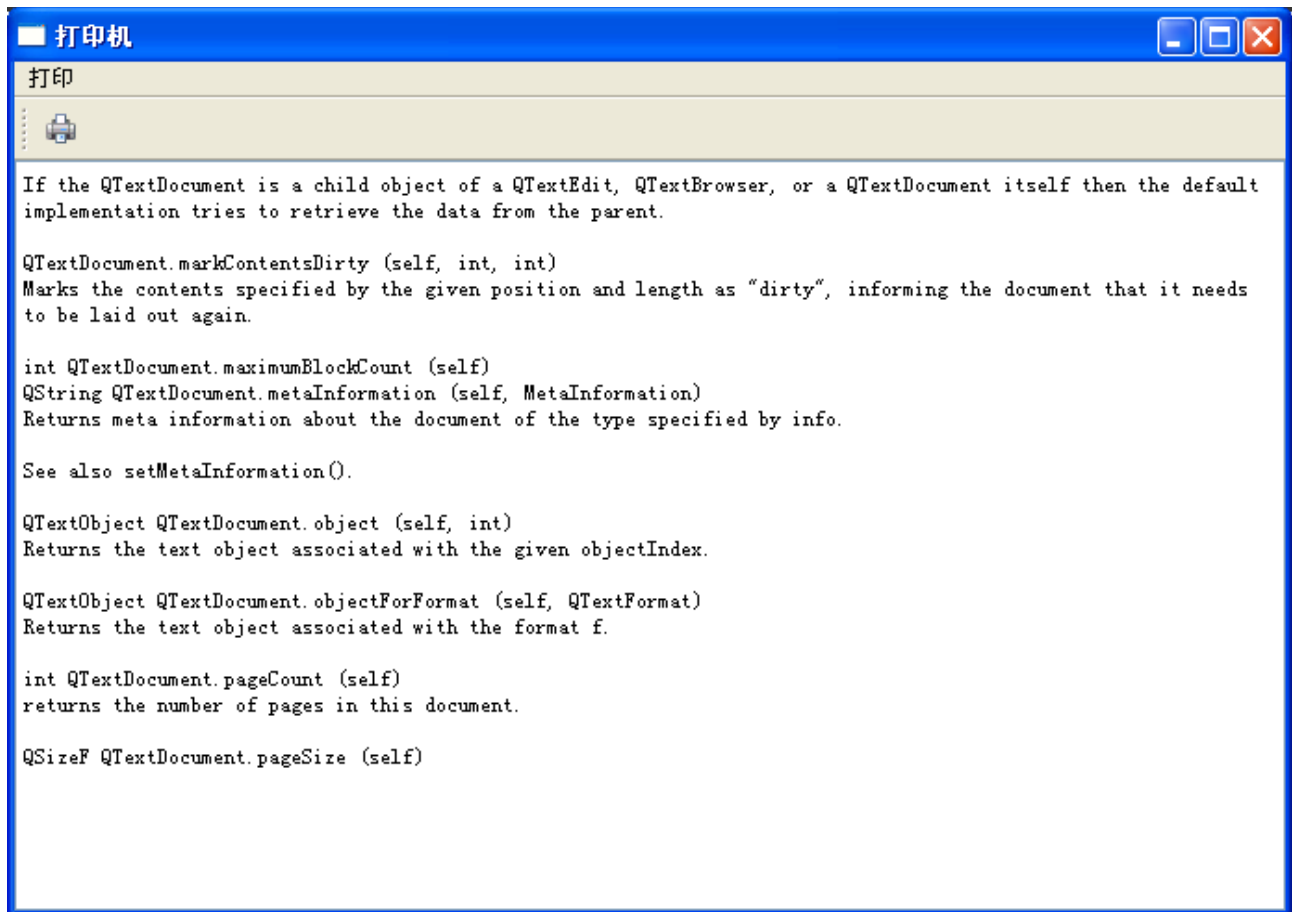
```
fileToolBar.setMovable(False)
```

指定文件工具条不可移动，只出现于主窗口的顶部。

本例中很多槽函数都实现的不全，此例只是展示基本的 `QMainWindow` 用法，以后还会再详细讲解。

PyQt4 精彩实例分析 实例 25 打印文本

打印文本在文本编辑工作中经常使用，本实例实现使用打印机打印文本的功能，如下图所示。



具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle(self.tr("打印机"))
        self.text=QTextEdit()
        self.setCentralWidget(self.text)

        self.createActions()
        self.createMenus()
        self.createToolBars()

        file=QFile("QPrinter.txt")
        if file.open(QIODevice.ReadOnly|QIODevice.Text):
            textStream=QTextStream(file)
            while not textStream.atEnd():
```



```

        self.text.append(textStream.readLine())
    file.close()

def createActions(self):
    self.PrintAction=QAction(QIcon("images/print.png"),self.tr("打印"),self)
    self.PrintAction.setShortcut("Ctrl+P")
    self.PrintAction.setStatusTip(self.tr("打印"))
    self.connect(self.PrintAction,SIGNAL("triggered()"),self.slotPrint)

def createMenus(self):
    PrintMenu=self.menuBar().addMenu(self.tr("打印"))
    PrintMenu.addAction(self.PrintAction)

def createToolBars(self):
    fileToolBar=self.addToolBar("Print")
    fileToolBar.addAction(self.PrintAction)

def slotPrint(self):
    printer=QPrinter()
    printDialog=QPrintDialog(printer,self)
    if printDialog.exec_():
        doc=self.text.document()
        doc.print_(printer)

app=QApplication(sys.argv)
main=MainWindow()
main.show()
app.exec_()

```

第 11 行设置窗体使用的字体和窗体标题。

第 12, 13 行创建一个 QTextEdit 控件 text，并设置为中心窗体。

第 15-17 行创建菜单，工具条等部件。

第 19-24 行从文本文件 QPrinter.txt 中逐行读取数据并显示在 text 中。

第 41 行新建一个 QPrinter 对象。

第 42 行创建一个 QPrintDialog 对象，参数为 QPrinter 对象。

QPrintDialog 是 Qt 提供的标准对话框，为打印机的使用提供了一种方便，规范的方法。如下图所示，QPrintDialog 标准对话框提供了打印机的选择，配置功能，并允许使用者改变文档有关的设置，如页面大小，方向，打印类型(彩色/灰度)，页边距以及打印份数等。



第 43 行判断打印对话框显示后用户是否单击“打印”按钮，或单击“打印”按钮，则相关打印属性将就可以通过创建 `QPrintDialog` 对象时使用的 `QPrinter` 对象获得，或用户单击“取消”按钮，则不执行后续的打印操作。

第 44 行获得 `QTextEdit` 对象的文档。

第 45 行打印。

PyQt4 精彩实例分析 实例 26 打印图像

打印图像是图像处理软件中的一个常用功能，本实例实现使用打印机打印图像的功能，如下图所示。



(这张图片的寓意你懂得)

打印图像实际上是在一个 `QPaintDevice` 中画图，与平常在 `QWidget`、`QPixmap` 和 `QImage` 中画图一样，都是创建一个 `QPainter` 对象进行画图，只是打印使用的是 `QPrinter`，`QPrinter` 本质上也是一个绘图设备 `QPaintDevice`。

具体实现代码如下：

```
# -*- coding: utf-8 -*-
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

QTextCodec.setCodecForTr(QTextCodec.codecForName("utf8"))

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle(self.tr("打印图片"))
        self.imageLabel=QLabel()

self.imageLabel.setSizePolicy(QSizePolicy.Ignored, QSizePolicy.Ignored)
self.setCentralWidget(self.imageLabel)

self.image=QImage()
```

```

self.createActions()
self.createMenus()
self.createToolBars()

if self.image.load("postgresql.jpg"):
    self.imageLabel.setPixmap(QPixmap.fromImage(self.image))
    self.resize(self.image.width(), self.image.height())

def createActions(self):
    self.PrintAction=QAction(QIcon("images/print.png"), self.tr("打印"), self)
    self.PrintAction.setShortcut("Ctrl+P")
    self.PrintAction.setStatusTip(self.tr("打印"))
    self.connect(self.PrintAction, SIGNAL("triggered()"), self.slotPrint)

def createMenus(self):
    PrintMenu=self.menuBar().addMenu(self.tr("打印"))
    PrintMenu.addAction(self.PrintAction)

def createToolBars(self):
    fileToolBar=self.addToolBar("Print")
    fileToolBar.addAction(self.PrintAction)

def slotPrint(self):
    printer=QPrinter()
    printDialog=QPrintDialog(printer, self)
    if printDialog.exec_():
        painter=QPainter(printer)
        rect=painter.viewport()
        size=self.image.size()
        size.scale(rect.size(), Qt.KeepAspectRatio)

painter.setViewport(rect.x(), rect.y(), size.width(), size.height())
painter.setWindow(self.image.rect())
painter.drawImage(0, 0, self.image)

app=QApplication(sys.argv)
main=MainWindow()
main.show()
app.exec_()

```

第 11 行设置窗体的标题。

第 12-14 行创建一个放置图像的 QLabel 对象 imageLabel，并将该 QLabel 对象设置为中心窗体。

第 18-20 行创建菜单，工具条等部件。

第 22-24 行在 imageLabel 对象中放置图像。

第 41 行新建一个 QPrinter 对象。

第 42 行创建一个 QPrintDialog 对象，参数为 QPrinter 对象。

第 43 行判断打印对话框显示后用户是否单击“打印”按钮，若单击“打印”按钮，则相关打印属性可以通过创建 QPrintDialog 对象时使用的 QPrinter 对象获得，若用户单击“取消”按钮，则不执行后续的打印操作。

第 44 行创建一个 QPainter 对象，并指定绘图设备为一个 QPrinter 对象。

第 45 行获得 QPainter 对象的视口矩形。

第 46 行获得图像的大小。

第 47, 48 行按照图形的比例大小重新设置视口矩形。

第 49 行设置 QPainter 窗口大小为图像的大小。

第 50 行打印。