# PyQt4工具包简介

## 关于PyQt

PyQt作为一组Python模块的实现。有超过300个类和超过6000个方法。它是个跨平台的工具包，它可以运行在所有的主流操作系统上。包括Unix、Windows、Mac。它具有双重授权，开发者可以选择GPL或者商业授权。以前GPL版本只能在Unix上获得。从PyQt4开始，可以在所有支持的平台上获得GPL协议。

由于有大量可用的类，它们被分成多个模块。



图：PyQt4模块

* **QtCore** 模块包括了核心的非GUI功能，该模块用来对时间、文件、目录、各种数据类型、流、网址、媒体类型、线程或进程进行处理。
* **QtGui** 模块包括图形化窗口组件和及相关类。包括如按钮、窗体、状态栏、滑块、位图、颜色、字体等等。
* **QtHelp** 模块包含了用于创建和查看可查找的文档的类。
* **QtNetwork** 模块包括网络编程的类。这些类可以用来编写TCP/IP和UDP的客户端和服务器。它们使得网络编程更容易和便捷。
* **QtOpenGL** 模块使用OpenGL库来渲染3D和2D图形。该模块使得Qt GUI库和OpenGL库无缝集成。
* **QtScript** 模块包含了使PyQt应用程序使用JavaScript解释器编写脚本的类。
* **QtSql** 模块提供操作数据库的类。
* **QtSvg** 模块提供了显示SVG文件内容的类。可缩放矢量图形(SVG)是一种用XML描述二维图形和图形应用的语言。
* **QtTest** 模块包含了对PyQt应用程序进行单元测试的功能。（PyQt没有实现完全的Qt单元测试框架，相反，它假设使用标准的Python单元测试框架来实现模拟用户和GUI进行交互。）
* **QtWebKit** 模块实现了基于开源浏览器引擎WebKit的浏览器引擎。
* **QtXml** 包括处理XML文件的类，该模块提供了SAX和DOM API的接口。
* **QtXmlPatterns** 模块包含的类实现了对XML和自定义数据模型的XQuery和XPath的支持。
* **phonon** 模块包含的类实现了跨平台的多媒体框架，可以在PyQt应用程序中使用音频和视频内容。
* **QtMultimedia** 模块提供了低级的多媒体功能，开发人员通常使用 **phonon** 模块。
* **QtAssistant** 模块包含的类允许集成 **Qt Assistant** 到PyQt应用程序中，提供在线帮助。
* **QtDesigner** 模块包含的类允许使用PyQt扩展 **Qt Designer** 。
* **Qt** 模块综合了上面描述的模块中的类到一个单一的模块中。这样做的好处是你不用担心哪个模块包含哪个特定的类，坏处是加载进了整个Qt框架，从而增加了应用程序的内存占用。
* **uic** 模块包含的类用来处理.ui文件，该文件由Qt Designer创建，用于描述整个或者部分用户界面。它包含的加载.ui文件和直接渲染以及从.ui文件生成Python代码为以后执行的类。

# 第一个程序

## 简单的例子

这个例子十分简单，它仅仅现实一个小窗体。但是我们可以在这个窗体上进行很多操作，我们可以调整大小、最大化、最小化。这些需要很多编码，有人已经把这些功能写好了。因为它在所有的应用程序中重复，因此没有必要再写一次，所以它被隐藏了起来。PyQt是一个高级的工具包。如果我们在一个低级的工具包下写代码，那么接下来的例子将很容易的达到几十行。

*# -\*- coding: utf-8 -\*-*\_\_author\_\_ = **'lhq'  
  
import** sys  
**from** PyQt4 **import** QtGui  
  
app = QtGui.QApplication(sys.argv)  
  
widget = QtGui.QWidget()  
widget.resize(250, 150)  
widget.setWindowTitle(**'simple'**)  
widget.show()  
  
sys.exit(app.exec\_())

上面的代码显示在屏幕上显示一个小窗体。

**import** sys  
**from** PyQt4 **import** QtGui

我们在这里进行一些必要的import操作。基本的GUI组件在 QtGui 模块中。

app = QtGui.QApplication(sys.argv)

每个PyQt4程序必须创建一个application对象，application在 QtGui 模块中， sys.argv 参数是命令行中的一组参数。Python脚本可以在shell中运行，这样，我们可以控制脚本的启动。

widget = QtGui.QWidget()

QWidget 窗口组件是PyQt4中所有用户界面对象的基类，我们使用 QWidget 默认的构造，没有父亲。没有父亲的窗口组件称为窗体。

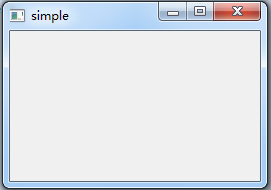
widget.show()

show() 方法将窗口呈现在屏幕上。

sys.exit(app.exec\_())

最后，我们输入应用程序的主事件循环，事件处理从这里开始。主事件循环从窗口系统接收事件并分发到应用程序的窗口组件上。当主事件循环结束，如果我们调用 exit() 方法或者主窗口组件被销毁。 sys.exit() 方法确保干净的退出。将通知环境应用程序是如何结束的。

你是否想知道为什么 exec\_() 方法会有下划线，一切皆有含义，这显然是因为exec是Python的关键字，因此，用 exec\_() 来取代它。



## 应用程序图标

应用程序图标是一个小图像，一般显示在标题栏的左上角，在接下来的例子中，我们将会展示如何在PyQt中实现。我们也会介绍一些新方法。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui  
  
  
**class** Icon(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)  
  
 self.setGeometry(300, 300, 250, 150)  
 self.setWindowTitle(**'Icon'**)  
 self.setWindowIcon(QtGui.QIcon(**'icons/panda.png'**))  
  
  
app = QtGui.QApplication(sys.argv)  
icon = Icon()  
icon.show()  
sys.exit(app.exec\_())

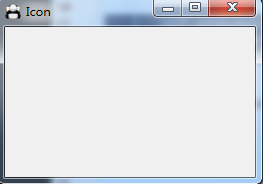
上一个例子用面向过程的方式编写。Python语言支持面向过程和面向对象编程。在PyQt4中编程意味着OOP编程。

**class** Icon(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)

在面向对象编程中三个最重要东西是类、数据和方法。这里我们创建一个名为 Icon 的新类，继承自 QtGui.QWidget 类，意味着我们必须调用两个构造方法，第一个为 Icon 类，第二个为继承的父类。

self.setGeometry(300, 300, 250, 150)  
self.setWindowTitle(**'Icon'**)  
self.setWindowIcon(QtGui.QIcon(**'icons/panda.png'**))

所有这三个类，均继承自 QtGui.QWidget ， setGeometry() 方法干了两件事，它定义了窗体在屏幕上的位置，并设置窗体的大小。开始的两个参数是窗体的x和y的位置，第三个是宽度，第四个是高度。最后一个方法设置了应用程序图标。通过这个，我们创建了一个 QIcon 对象， QIcon 接受我们想要显示的图标的路径。



## 关闭窗体

关闭一个窗体很明显的方式是点击标题栏上的x标记，在接下来的例子中，我们将展示如何用编程的方式关闭窗体，我们将简略的接触到信号和槽。

接下来是构造一个 QPushButton ，我们将用在例子中。

QPushButton(string text, QWidget parent = None)

参数 text 是显示在按钮上的文本， parent 是放置按钮的父亲，在这里是 QWidget 。

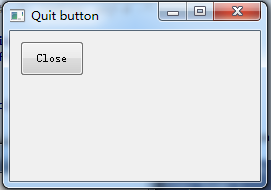
*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
**class** QuitButton(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)  
  
 self.setGeometry(300, 300, 250, 150)  
 self.setWindowTitle(**'Quit button'**)  
  
 quit = QtGui.QPushButton(**'Close'**, self)  
 quit.setGeometry(10, 10, 64, 35)  
  
 self.connect(quit, QtCore.SIGNAL(**'clicked()'**),  
 QtGui.qApp, QtCore.SLOT(**'quit()'**))  
  
  
app = QtGui.QApplication(sys.argv)  
qb = QuitButton()  
qb.show()  
sys.exit(app.exec\_())

quit = QtGui.QPushButton(**'Close'**, self)  
quit.setGeometry(10, 10, 64, 35)

创建一个按钮并放置在 QWidget 上，就像把 QWidget 放置在屏幕上一样。

self.connect(quit, QtCore.SIGNAL(**'clicked()'**),  
 QtGui.qApp, QtCore.SLOT(**'quit()'**))

PyQt4中的事件处理系统是建立信号和槽机制。如果点击按钮，信号 clicked() 将会发射，槽可以是PyQt的槽或者任何的Python调用。 QtCore.QObject.connect() 把信号和槽连接起来。在这个例子中槽是PyQt预定义的 quit() 槽，发射方和接收方两个对象间进行通讯，发射方为按钮，接收方为application对象。



## 消息框

默认情况下，当我们点击标题栏上的关闭按钮， QWdiget 将关闭。有时候我们需要改变默认行为，比如：当我们在编辑器中打开一个文件并且做了一些修改，显示一个消息框来确认这个动作。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui  
  
**class** MessageBox(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)  
  
 self.setGeometry(300, 300, 250, 150)  
 self.setWindowTitle(**'message box'**)  
  
  
 **def** closeEvent(self, event):  
 reply = QtGui.QMessageBox.question(self, **'Message'**,  
 **"Are you sure to quit?"**, QtGui.QMessageBox.Yes, QtGui.QMessageBox.No)  
  
 **if** reply == QtGui.QMessageBox.Yes:  
 event.accept()  
 **else**:  
 event.ignore()  
  
app = QtGui.QApplication(sys.argv)  
qb = MessageBox()  
qb.show()  
sys.exit(app.exec\_())

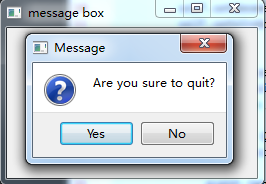
如果我们关闭 QWidget ，将会产生一个 QCloseEvent 事件，我们需要重新实现 closeEvent() 事件来改变组件的行为。

reply = QtGui.QMessageBox.question(self, **'Message'**,  
 **"Are you sure to quit?"**, QtGui.QMessageBox.Yes, QtGui.QMessageBox.No)

我们显示一个两个按钮的消息框，Yes和No。第一个字符串显示在标题栏上，第二个字符串是信息文本，显示在对话框中，返回值存储在 relay 变量中。

**if** reply == QtGui.QMessageBox.Yes:  
 event.accept()  
**else**:  
 event.ignore()

这里我们测试返回值，如果点击Yes按钮，表明接受这个事件，将会关闭窗口组件并进回到应用程序的结尾，反之则忽略该关闭事件。



## 屏幕居中显示窗体

下面的脚本我们将展示如何在屏幕中央显示窗体。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui  
  
**class** Center(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)  
  
 self.setWindowTitle(**'center'**)  
 self.resize(250, 150)  
 self.center()  
  
 **def** center(self):  
 screen = QtGui.QDesktopWidget().screenGeometry()  
 size = self.geometry()  
 self.move((screen.width()-size.width())/2, (screen.height()-size.height())/2)  
  
  
app = QtGui.QApplication(sys.argv)  
qb = Center()  
qb.show()  
sys.exit(app.exec\_())

screen = QtGui.QDesktopWidget().screenGeometry()

计算屏幕分辨率。

size = self.geometry()

获得 QWidget 的尺寸。

self.move((screen.width()-size.width())/2, (screen.height()-size.height())/2)

# 菜单和工具栏

## 主窗体

QMainWindow 类提供应用程序主窗口，可以创建一个经典的拥有状态栏、工具栏和菜单栏的应用程序骨架。

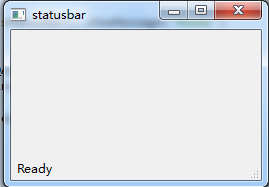
## 状态栏

状态栏是用来显示状态信息的窗口组件。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui  
  
**class** MainWindow(QtGui.QMainWindow):  
 **def** \_\_init\_\_(self):  
 QtGui.QMainWindow.\_\_init\_\_(self)  
  
 self.resize(250, 150)  
 self.setWindowTitle(**'statusbar'**)  
  
 self.statusBar().showMessage(**'Ready'**)  
  
  
app = QtGui.QApplication(sys.argv)  
main = MainWindow()  
main.show()  
sys.exit(app.exec\_())

self.statusBar().showMessage(**'Ready'**)

通过掉用 QMainWindow 的 statusBar() 方法来得到状态栏， showMessage() 将消息显示在状态栏上。



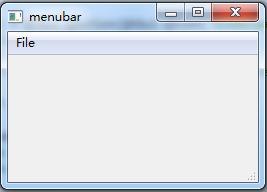
## 菜单栏

菜单栏是GUI应用程序最明显的部分之一，这是分布在各个菜单的命令组合，在控制台程序中你需要记住所有那些晦涩难懂的命令，有了这些公认的标准，将进一步缩短学习新应用程序的时间。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
**class** MainWindow(QtGui.QMainWindow):  
 **def** \_\_init\_\_(self):  
 QtGui.QMainWindow.\_\_init\_\_(self)  
  
 self.resize(250, 150)  
 self.setWindowTitle(**'menubar'**)  
  
 exit = QtGui.QAction(QtGui.QIcon(**'icons/exit.png'**), **'Exit'**, self)  
 exit.setShortcut(**'Ctrl+Q'**)  
 exit.setStatusTip(**'Exit application'**)  
 self.connect(exit, QtCore.SIGNAL(**'triggered()'**), QtCore.SLOT(**'close()'**))  
  
 self.statusBar()  
  
 menubar = self.menuBar()  
 file = menubar.addMenu(**'&File'**)  
 file.addAction(exit)  
  
app = QtGui.QApplication(sys.argv)  
main = MainWindow()  
main.show()  
sys.exit(app.exec\_())

menubar = self.menuBar()  
file = menubar.addMenu(**'&File'**)  
file.addAction(exit)

首先我们使用``QMainWindow`` 类的 menuBar() 方法创建一个菜单栏，然后使用 addMenu() 方法增加一个菜单项，最后我们插入 Action 对象到文件菜单。



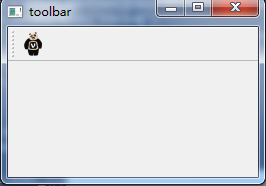
## 工具栏

菜单集合了应用程序中所有可用的命令，工具栏提供了快速访问最常用的命令功能。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
**class** MainWindow(QtGui.QMainWindow):  
 **def** \_\_init\_\_(self):  
 QtGui.QMainWindow.\_\_init\_\_(self)  
  
 self.resize(250, 150)  
 self.setWindowTitle(**'toolbar'**)  
  
 exit = QtGui.QAction(QtGui.QIcon(**'icons/panda2.png'**), **'Exit'**, self)  
 exit.setShortcut(**'Ctrl+Q'**)  
 self.connect(exit, QtCore.SIGNAL(**'triggered()'**), QtCore.SLOT(**'close()'**))  
  
 self.toolbar = self.addToolBar(**'Exit'**)  
 self.toolbar.addAction(exit)  
  
  
app = QtGui.QApplication(sys.argv)  
main = MainWindow()  
main.show()  
sys.exit(app.exec\_())

exit = QtGui.QAction(QtGui.QIcon(**'icons/panda2.png'**), **'Exit'**, self)  
exit.setShortcut(**'Ctrl+Q'**)

GUI程序通过命令来控制，这些命令可以通过菜单、上下文菜单、工具栏或者快捷键来启动。PyQt引入了行为(Action)来简化开发。行为对象可以拥有菜单标题、图标、快捷键、状态栏内容、aasda“这是什么？”内容以及提示。在我们的例子中，我们定义了一个行为对象，包括图标、提示和快捷键。



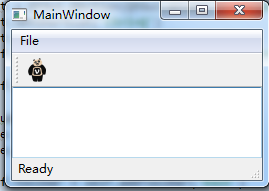
## 放在一起

在这部分的最后一个例子中，我们将会创建一个菜单栏、工具栏和状态栏。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
**class** MainWindow(QtGui.QMainWindow):  
 **def** \_\_init\_\_(self):  
 QtGui.QMainWindow.\_\_init\_\_(self)  
  
 self.resize(250, 150)  
 self.setWindowTitle(**'MainWindow'**)  
  
 textEdit = QtGui.QTextEdit()  
 self.setCentralWidget(textEdit)  
  
 exit = QtGui.QAction(QtGui.QIcon(**'icons/panda2.png'**), **'Exit'**, self)  
 exit.setShortcut(**'Ctrl+Q'**)  
 exit.setStatusTip(**'Exit application'**)  
 self.connect(exit, QtCore.SIGNAL(**'triggered()'**), QtCore.SLOT(**'close()'**))  
  
 self.statusBar().showMessage(**'Ready'**)  
  
 menubar = self.menuBar()  
 file = menubar.addMenu(**'&File'**)  
 file.addAction(exit)  
  
 self.toolbar = self.addToolBar(**'Exit'**)  
 self.toolbar.addAction(exit)  
  
  
app = QtGui.QApplication(sys.argv)  
main = MainWindow()  
main.show()  
sys.exit(app.exec\_())

textEdit = QtGui.QTextEdit()  
self.setCentralWidget(textEdit)

这里我们创建了一个文本编辑控件，把它设置成 QMainWinow 的中心组件。中心组件将会占据所有留下的空间。



# 布局管理

编程中的一个重要事情是布局管理，布局管理是如何在窗体上摆放窗口组件。可以有两种方式进行管理：绝对定位或使用布局类。

## 绝对定位

程序员用像素指定每个控件的位置和尺寸。使用绝对定位时，你必须理解几件事情。

* 如果你调整窗体的大小，组件的尺寸和位置并不会改变
* 在不同的平台上，程序可能看起来不一样
* 改变程序的字体可能破坏布局
* 如果你决定改变你的布局，你必须完全重做你的布局，这将是乏味并且浪费时间的
* *# -\*- coding: utf-8 -\*-  
    
  # absolute.py***import** sys  
  **from** PyQt4 **import** QtGui  
    
    
  **class** Example(QtGui.QWidget):  
    
   **def** \_\_init\_\_(self):  
   super(Example, self).\_\_init\_\_()  
    
   self.initUI()  
    
   **def** initUI(self):  
   label1 = QtGui.QLabel(**'Zetcode'**, self)  
   label1.move(15, 10)  
    
   label2 = QtGui.QLabel(**'tutorials for programmers'**, self)  
   label2.move(35, 40)  
    
   self.setWindowTitle(**'Absolute'**)  
   self.resize(250, 150)  
    
  app = QtGui.QApplication(sys.argv)  
  ex = Example()  
  ex.show()  
  sys.exit(app.exec\_())
* 我们简单的调用 move() 方法来定位组件。在我们的 QLabel 例子中，我们用x和y坐标来定位。坐标系统从左上角开始，x值从左到右增长，y值从上到下增长。

## 框布局

使用布局类管理布局更灵活、更实用。这是在窗体上摆放组件的首选方式。基本的布局类是 QHBoxLayout 和 QVBoxLayout ，它们可以横向和纵向排列窗口组件。

假设我们想要摆放两个按钮到右下角。为了创建这样一个布局，我们需要一个水平框和一个垂直框。我们通过增加 **延展因素** 来创建必要的间隔。

*# -\*- coding: utf-8 -\*-  
  
# boxlayout.py***import** sys  
**from** PyQt4 **import** QtGui  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
 okButton = QtGui.QPushButton(**"OK"**)  
 cancelButton = QtGui.QPushButton(**"Cancel"**)  
  
 hbox = QtGui.QHBoxLayout()  
 hbox.addStretch(1)  
 hbox.addWidget(okButton)  
 hbox.addWidget(cancelButton)  
  
 vbox = QtGui.QVBoxLayout()  
 vbox.addStretch(1)  
 vbox.addLayout(hbox)  
  
 self.setLayout(vbox)  
  
 self.setWindowTitle(**'box layout'**)  
 self.resize(300, 150)  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

hbox = QtGui.QHBoxLayout()  
hbox.addStretch(1)  
hbox.addWidget(okButton)  
hbox.addWidget(cancelButton)

我们创建一个水平框布局，增加一个延展因素和两个按钮。

vbox = QtGui.QVBoxLayout()  
vbox.addStretch(1)  
vbox.addLayout(hbox)

为了创建所需的布局，我们把水平布局放到垂直布局中。

self.setLayout(vbox)

最后，我们设置窗体的主布局。

## 网格布局

最常用的布局类是网格布局，网格布局把空间划分成行和列。我们使用 QGridLayout 类来创建网格布局。

*# -\*- coding: utf-8 -\*-  
  
# gridlayout1.py***import** sys  
**from** PyQt4 **import** QtGui  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
 self.setWindowTitle(**'grid layout'**)  
  
 names = [**'Cls'**, **'Bck'**, **''**, **'Close'**, **'7'**, **'8'**, **'9'**, **'/'**,  
 **'4'**, **'5'**, **'6'**, **'\*'**, **'1'**, **'2'**, **'3'**, **'-'**,  
 **'0'**, **'.'**, **'='**, **'+'**]  
  
 grid = QtGui.QGridLayout()  
  
 j = 0  
 pos = [(0, 0), (0, 1), (0, 2), (0, 3),  
 (1, 0), (1, 1), (1, 2), (1, 3),  
 (2, 0), (2, 1), (2, 2), (2, 3),  
 (3, 0), (3, 1), (3, 2), (3, 3 ),  
 (4, 0), (4, 1), (4, 2), (4, 3)]  
  
 **for** i **in** names:  
 button = QtGui.QPushButton(i)  
 **if** j == 2:  
 grid.addWidget(QtGui.QLabel(**''**), 0, 2)  
 **else**: grid.addWidget(button, pos[j][0], pos[j][1])  
 j = j + 1  
  
 self.setLayout(grid)  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

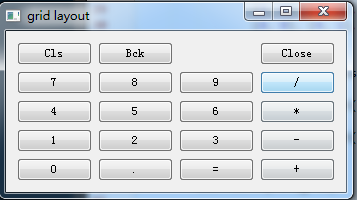
在该例子中，我们创建了一个按钮格，增加一个 QLabel 窗口组件来填补一个空白。

grid = QtGui.QGridLayout()

Here we create a grid layout.

**if** j == 2:  
 grid.addWidget(QtGui.QLabel(**''**), 0, 2)  
**else**:   
 grid.addWidget(button, pos[j][0], pos[j][1])

调用 addWidget() 方法来把窗口组件加到网格中，参数是部件（ widget ），行（ row ）和列（ column ）数字。



组件可以在表格中跨越多列或多行，在下一个例子中我们将演示这个。

*# -\*- coding: utf-8 -\*-***import** sys  
**from** PyQt4 **import** QtGui  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
 title = QtGui.QLabel(**'Title'**)  
 author = QtGui.QLabel(**'Author'**)  
 review = QtGui.QLabel(**'Review'**)  
  
 titleEdit = QtGui.QLineEdit()  
 authorEdit = QtGui.QLineEdit()  
 reviewEdit = QtGui.QTextEdit()  
  
 grid = QtGui.QGridLayout()  
 grid.setSpacing(10)  
  
 grid.addWidget(title, 1, 0)  
 grid.addWidget(titleEdit, 1, 1)  
  
 grid.addWidget(author, 2, 0)  
 grid.addWidget(authorEdit, 2, 1)  
  
 grid.addWidget(review, 3, 0)  
 grid.addWidget(reviewEdit, 3, 1, 5, 1)  
  
 self.setLayout(grid)  
  
 self.setWindowTitle(**'grid layout'**)  
 self.resize(350, 300)  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

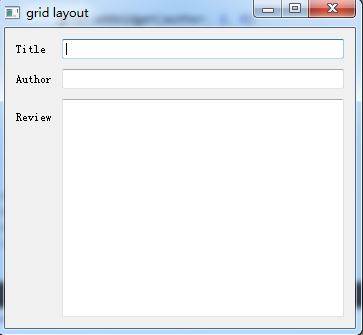
我们创建了一个窗体，上面有三个标签，两个单行编辑框和一个文本编辑框组件，使用 QGridLayout 来布局。

grid = QtGui.QGridLayout()  
grid.setSpacing(10)

我们创建了一个表格布局，并且设置组件间的间隔。

grid.addWidget(reviewEdit, 3, 1, 5, 1)

如果我们增加一个窗口组件到网格中，我们可以提供窗口组件的行跨度和列跨度。在这个例子中，我们设置 reviewEdit 占用5行。



# 事件和信号

在PyQt4教程的这部分中，我们将探讨应用中事件和信号的发生。

事件是GUI程序的重要部分，由用户或者系统产生。当我们调用应用的 exec\_() 方法，应用进入主循环。主循环获取事件并把它们发往对象。Trolltech引入了独一无二的信号和槽机制。

事件是GUI程序的主要部分，所有GUI应用程序都是事件驱动的。应用在它的生命周期中产生的不同事件交互。事件主要由用户产生，但是它们也可以由其他方式产生，如：互联网，窗口管理器，定时器。在事件模型中，由三个参与者：

::

* event source
* event object
* event target
* 事件来源
* 事件对象
* 事件目标

**事件对象** 是指状态改变的对象，它产生了事件。 **事件对象** （Event）封装了事件元的状态改变。 **事件目标** 是事件对象想要想要通知的对象。事件来源对象代理了事件的目标要处理的任务。

当我们调用了程序 exec\_() 方法，程序进入了主循环。主循环捕获事件并把它们发往对象。信号和槽用于对象之间的通讯。当一个特殊的事件发生时，将发射 **信号** ， **槽** 可以是任何Python调用，当链接到槽的信号发射，该槽将被调用。

## 信号和槽

*# -\*- coding: utf-8 -\*-  
  
# sigslot.py***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 lcd = QtGui.QLCDNumber(self)  
 slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)  
  
 vbox = QtGui.QVBoxLayout()  
 vbox.addWidget(lcd)  
 vbox.addWidget(slider)  
  
 self.setLayout(vbox)  
 self.connect(slider, QtCore.SIGNAL(**'valueChanged(int)'**), lcd,  
 QtCore.SLOT(**'display(int)'**))  
  
 self.setWindowTitle(**'Signal & slot'**)  
 self.resize(250, 150)  
  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

在我们的例子中，我们显示了一个LCD数字和一个滑块，通过拖动滑块来改变LCD的值。

self.connect(slider, QtCore.SIGNAL(**'valueChanged(int)'**), lcd,  
 QtCore.SLOT(**'display(int)'**))

这里我们连接滑块的 valueChanged() 信号到LCD数字的 display() 槽。

connect 方法有4个参数， sender 是发送信号的对象， signal 是发射的信号， receiver 是接收信号的对象， 最后， slog 是对信号反应的方法。



## 重新实现事件处理程序

*# -\*- coding: utf-8 -\*-  
  
# escape.py***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.setWindowTitle(**'Escape'**)  
 self.resize(250, 150)  
  
 **def** keyPressEvent(self, event):  
 **if** event.key() == QtCore.Qt.Key\_Escape:  
 self.close()  
  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

在我们的例子中，我们重新实现了 keyPressEvent() 处理。

**def** keyPressEvent(self, event):  
 **if** event.key() == QtCore.Qt.Key\_Escape:  
 self.close()

如果我们按下了escape按钮，程序将退出。

## 事件发送者

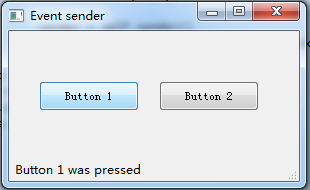
有时需要方便的知道哪个组件发出的信号，PyQt4有 sender() 方法。

*# -\*- coding: utf-8 -\*-  
  
# sender.py***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
  
**class** Example(QtGui.QMainWindow):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 button1 = QtGui.QPushButton(**"Button 1"**, self)  
 button1.move(30, 50)  
  
 button2 = QtGui.QPushButton(**"Button 2"**, self)  
 button2.move(150, 50)  
  
 self.connect(button1, QtCore.SIGNAL(**'clicked()'**),  
 self.buttonClicked)  
  
 self.connect(button2, QtCore.SIGNAL(**'clicked()'**),  
 self.buttonClicked)  
  
 self.statusBar().showMessage(**'Ready'**)  
 self.setWindowTitle(**'Event sender'**)  
 self.resize(290, 150)  
  
  
 **def** buttonClicked(self):  
  
 sender = self.sender()  
 self.statusBar().showMessage(sender.text() + **' was pressed'**)  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

两个按钮都连接了同一个信号。

**def** buttonClicked(self):  
  
 sender = self.sender()  
 self.statusBar().showMessage(sender.text() + **' was pressed'**)

通过调用 sender() 方法我们确定信号来源。在程序的状态栏，我们显示按下的按钮的标签。



## 发射信号

从 QtCore.QObject 继承的对象可以发射信号。如果点击按钮，将产生一个 clicked() 信号。在接下来的例子中可以看到如何发射信号。

*# -\*- coding: utf-8 -\*-  
  
# emit.py***import** sys  
**from** PyQt4 **import** QtGui, QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.connect(self, QtCore.SIGNAL(**'closeEmitApp()'**),  
 QtCore.SLOT(**'close()'**))  
  
 self.setWindowTitle(**'emit'**)  
 self.resize(250, 150)  
  
 **def** mousePressEvent(self, event):  
 self.emit(QtCore.SIGNAL(**'closeEmitApp()'**))  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
sys.exit(app.exec\_())

我们创建一个名为 closeEmitApp() 的新信号，在鼠标的按下实践中发射该信号。

**def** mousePressEvent(self, event):  
 self.emit(QtCore.SIGNAL(**'closeEmitApp()'**))

这里我们把手工创建的 closeEmitApp() 信号和 close() 槽连接。

self.connect(self, QtCore.SIGNAL(**'closeEmitApp()'**),  
 QtCore.SLOT(**'close()'**))

# 对话框

对话框窗体或对话框是现代GUI应用不可或缺的一部分。dialog定义为两个或多个人之间的交谈。在计算机程序中dialog是一个窗体，用来和程序“交谈”。对话框用来输入数据、修改数据、改变程序设置等等。对话框是用户和计算机程序沟通的重要手段。

## 输入对话框QInputDialog

QInputDialog 提供一个简单的对话框，以便从用户获取单个值。输入值可以是一个字符串，一个数字或者列表的一项。

*# -\*- coding: utf-8 -\*-  
  
# inputdialog.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.button = QtGui.QPushButton(**'Dialog'**, self)  
 self.button.setFocusPolicy(QtCore.Qt.NoFocus)  
  
 self.button.move(20, 20)  
 self.connect(self.button, QtCore.SIGNAL(**'clicked()'**),  
 self.showDialog)  
 self.setFocus()  
  
 self.label = QtGui.QLineEdit(self)  
 self.label.move(130, 22)  
  
 self.setWindowTitle(**'InputDialog'**)  
 self.setGeometry(300, 300, 350, 80)  
  
  
 **def** showDialog(self):  
 text, ok = QtGui.QInputDialog.getText(self, **'Input Dialog'**,  
 **'Enter your name:'**)  
  
 **if** ok:  
 self.label.setText(str(text))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()

这个例子有一个按钮和一个单行编辑器，按钮显示输入对话框来获取值，输入的值将会显示在单行编辑器中。

text, ok = QtGui.QInputDialog.getText(self, **'Input Dialog'**,  
 **'Enter your name:'**)

这行代码显示输入对话框，第一个字符串是对话框标题，第二个是对话框里面的消息。对话框返回输入的文本那一个布尔值。如果我们点击 ok 按钮，布尔值是 True ，否则为 False 。

## 颜色对话框QcolorDialog

颜色对话框为定制颜色提供一个对话框组件。

*# -\*- coding: utf-8 -\*-  
  
# colordialog.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 color = QtGui.QColor(0, 0, 0)  
  
 self.button = QtGui.QPushButton(**'Dialog'**, self)  
 self.button.setFocusPolicy(QtCore.Qt.NoFocus)  
 self.button.move(20, 20)  
  
 self.connect(self.button, QtCore.SIGNAL(**'clicked()'**),  
 self.showDialog)  
 self.setFocus()  
  
 self.widget = QtGui.QWidget(self)  
 self.widget.setStyleSheet(**"QWidget { background-color: %s }"** % color.name())  
 self.widget.setGeometry(130, 22, 100, 100)  
  
 self.setWindowTitle(**'ColorDialog'**)  
 self.setGeometry(300, 300, 250, 180)  
  
  
 **def** showDialog(self):  
  
 col = QtGui.QColorDialog.getColor()  
  
 **if** col.isValid():  
 self.widget.setStyleSheet(**"QWidget { background-color: %s }"** % col.name())  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()

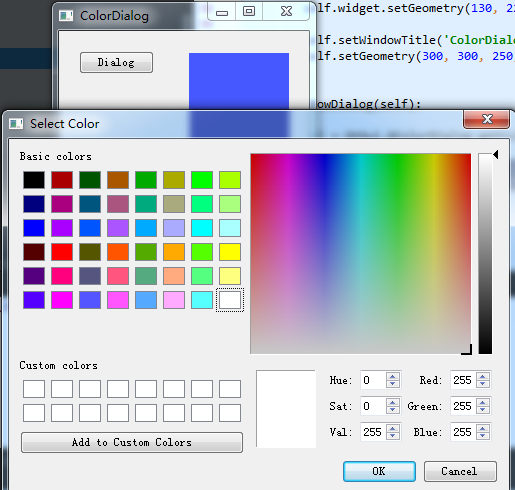
该例子显示一个按钮和一个 QWidge 对象。组件的背景设为黑色，我们可以用 QColorDialog 来改变它的背景。

col = QtGui.QColorDialog.getColor()

这行代码将会弹出一个 QColorDialog 。

**if** col.isValid():  
 self.widget.setStyleSheet(**"QWidget { background-color: %s }"** % col.name())

检查颜色是否有效，如果点击了取消按钮，将返回无效的颜色。如果颜色有些，我们使用样式修改背景颜色。



## 文件对话框QFileDialog

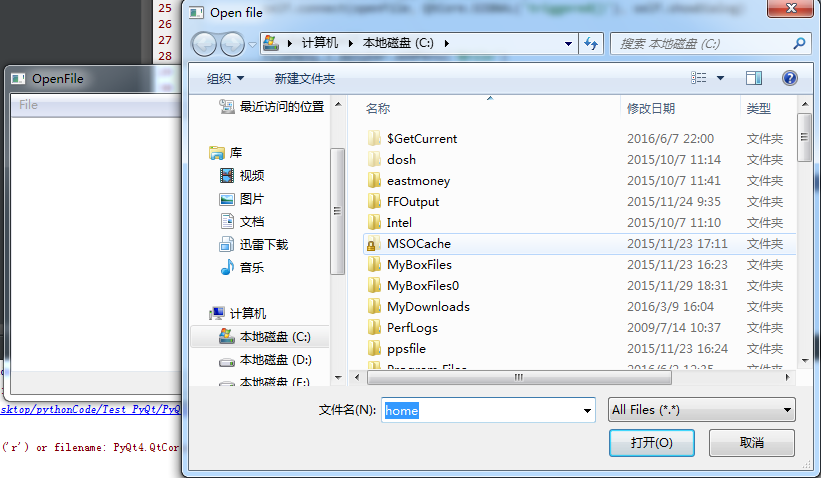
QFileDialog 允许用户选择文件或文件夹，可选择文件来打开和保存。

*# openfiledialog.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QMainWindow):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.textEdit = QtGui.QTextEdit()  
 self.setCentralWidget(self.textEdit)  
 self.statusBar()  
 self.setFocus()  
  
 openFile = QtGui.QAction(QtGui.QIcon(**'open.png'**), **'Open'**, self)  
 openFile.setShortcut(**'Ctrl+O'**)  
 openFile.setStatusTip(**'Open new File'**)  
 self.connect(openFile, QtCore.SIGNAL(**'triggered()'**), self.showDialog)  
  
 menubar = self.menuBar()  
 fileMenu = menubar.addMenu(**'&File'**)  
 fileMenu.addAction(openFile)  
  
 self.setGeometry(300, 300, 350, 300)  
 self.setWindowTitle(**'OpenFile'**)  
  
 **def** showDialog(self):  
  
 filename = QtGui.QFileDialog.getOpenFileName(self, **'Open file'**,  
 **'/home'**)  
 fname = open(filename)  
 data = fname.read()  
 self.textEdit.setText(data)  
  
app = QtGui.QApplication(sys.argv)  
ex = Example()  
ex.show()  
app.exec\_()

这个例子显示一个菜单，中间放置一个文本编辑框，还有一个状态栏。状态机仅为了设计目的显示。菜单项显示 QFileDialog 来选择文件，文件的内容加载进文本编辑器。

filename = QtGui.QFileDialog.getOpenFileName(self, **'Open file'**,  
 **'/home'**)

我们弹出 QFileDialog ， getOpenFileName 方法的第一个字符串是标题，第二个字符串指定对话框的工作目录，文件过滤默认设置 All files(\*) 。



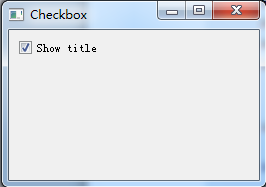
# 窗口组件

窗口组件是应用程序的基本构建块。PyQt4编程工具包拥有范围广泛的各种窗口组件。按钮、选择框、滑块、列表框等等，程序员工作所需要的一切。在教程的这部分中，我们将介绍一些有用的窗口组件。

## 复选框QCheckBox

QCheckBox （复选框） 是一个由两种状态的窗口组件。 On 和 Off 。他是一个带标签的框。每段一个复选框被选中和或者清楚时，都将发射信号 stateChanged() 。

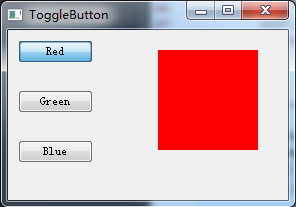
*# -\*- coding: utf-8 -\*-  
  
# checkbox.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.setGeometry(300, 300, 250, 150)  
 self.setWindowTitle(**'Checkbox'**)  
  
 self.cb = QtGui.QCheckBox(**'Show title'**, self)  
 self.cb.setFocusPolicy(QtCore.Qt.NoFocus)  
 self.cb.move(10, 10)  
 self.cb.toggle()  
 self.connect(self.cb, QtCore.SIGNAL(**'stateChanged(int)'**),  
 self.changeTitle)a  
  
 **def** changeTitle(self, value):  
  
 **if** self.cb.isChecked():  
 self.setWindowTitle(**'Checkbox'**)  
 **else**:  
 self.setWindowTitle(**''**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()



## 切换按钮QPushButton

PyQt4没有切换按钮的窗口组件，为了创建切换按钮，我们使用特殊模式的 QPushButton 。切换按钮是指一个两种状态的按钮，按下和非按下。通过点击切换两种状态。在某种状态下来这种方式很合适。

*# -\*- coding: utf-8 -\*-  
  
# togglebutton.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.color = QtGui.QColor(0, 0, 0)  
  
 self.red = QtGui.QPushButton(**'Red'**, self)  
 self.red.setCheckable(True)  
 self.red.move(10, 10)  
  
 self.connect(self.red, QtCore.SIGNAL(**'clicked()'**), self.setColor)  
  
 self.green = QtGui.QPushButton(**'Green'**, self)  
 self.green.setCheckable(True)  
 self.green.move(10, 60)  
  
 self.connect(self.green, QtCore.SIGNAL(**'clicked()'**), self.setColor)  
  
 self.blue = QtGui.QPushButton(**'Blue'**, self)  
 self.blue.setCheckable(True)  
 self.blue.move(10, 110)  
  
 self.connect(self.blue, QtCore.SIGNAL(**'clicked()'**), self.setColor)  
  
 self.square = QtGui.QWidget(self)  
 self.square.setGeometry(150, 20, 100, 100)  
 self.square.setStyleSheet(**"QWidget { background-color: %s }"** %  
 self.color.name())  
  
 self.setWindowTitle(**'ToggleButton'**)  
 self.setGeometry(300, 300, 280, 170)  
  
  
 **def** setColor(self):  
  
 source = self.sender()  
  
 **if** source.text() == **"Red"**:  
 **if** self.red.isChecked():  
 self.color.setRed(255)  
 **else**: self.color.setRed(0)  
  
 **elif** source.text() == **"Green"**:  
 **if** self.green.isChecked():  
 self.color.setGreen(255)  
 **else**: self.color.setGreen(0)  
  
 **else**:  
 **if** self.blue.isChecked():  
 self.color.setBlue(255)  
 **else**: self.color.setBlue(0)  
  
 self.square.setStyleSheet(**"QWidget { background-color: %s }"** %  
 self.color.name())  
  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()



## 滑块QSlider

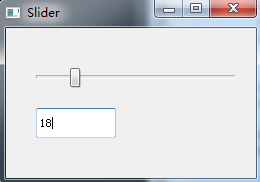
滑块是由一个简单的滑柄的窗口组件。该滑柄可以前后拖动，通过这种方式我们可以为特定任务选择值。有时候使用滑块比简单提供数值或使用微调框(spin box)更自然。 QLabel 显示文字或图像。

该例子中我们将显示一个滑块和一个标签。这次，标签将显示一个图像，滑块用来控制标签。

*# -\*- coding: utf-8 -\*-  
  
# slider.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)  
 self.slider.setFocusPolicy(QtCore.Qt.NoFocus)  
 self.slider.setGeometry(30, 40, 200, 20)  
 self.connect(self.slider, QtCore.SIGNAL(**'valueChanged(int)'**),  
 self.changeValue)  
  
 self.textEdit = QtGui.QLineEdit(self)  
 self.textEdit.setGeometry(30, 80, 80, 30)  
  
 self.setWindowTitle(**'Slider'**)  
 self.setGeometry(300, 300, 250, 150)  
  
  
 **def** changeValue(self):  
 self.textEdit.setText(str(self.slider.value()))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()

这个例子中我们模拟音量控制，通过拖动滑块的滑柄来改变标签上的图片。

self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)



## 进度条QProgressBar

进度条使用来处理长时间任务的窗口组件，当看到它的动画时，用户就知道我们的任务正在进行中。在PyQt4工具包中， QProgressBar 窗口组件提供水平或者垂直的进度条。任务被分成一些阶段。程序员可以为进度条设置最小值和最大值。默认是0，99.

*# -\*- coding: utf-8 -\*-  
  
# progressbar.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.pbar = QtGui.QProgressBar(self)  
 self.pbar.setGeometry(30, 40, 200, 25)  
  
 self.button = QtGui.QPushButton(**'Start'**, self)  
 self.button.setFocusPolicy(QtCore.Qt.NoFocus)  
 self.button.move(40, 80)  
  
 self.button1 = QtGui.QPushButton(**'ReStart'**, self)  
 self.button1.setFocusPolicy(QtCore.Qt.NoFocus)  
 self.button1.move(150, 80)  
  
 self.connect(self.button, QtCore.SIGNAL(**'clicked()'**),  
 self.doAction)  
 self.connect(self.button1, QtCore.SIGNAL(**'clicked()'**),  
 self.reStart)  
  
 self.timer = QtCore.QBasicTimer()  
 self.step = 0  
  
 self.setWindowTitle(**'ProgressBar'**)  
 self.setGeometry(300, 300, 250, 150)  
  
  
 **def** timerEvent(self, event):  
 **if** self.step >= 100:  
 self.timer.stop()  
 **return** self.step = self.step + 1  
 self.pbar.setValue(self.step)  
  
 **def** doAction(self):  
 **if** self.timer.isActive():  
 self.timer.stop()  
 self.button.setText(**'Start'**)  
 **else**:  
 self.timer.start(100, self)  
 self.button.setText(**'Stop'**)  
  
 **def** reStart(self):  
 self.step = 0  
 self.pbar.setValue(self.step)  
 self.button.setText(**'Start'**)  
 self.timer.stop()  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()

这个例子中有一个水平进度表和一个按钮，按钮开始或结束进度条。

self.pbar = QtGui.QProgressBar(self)

构建 QProgressBar 。

self.timer = QtCore.QBasicTimer()

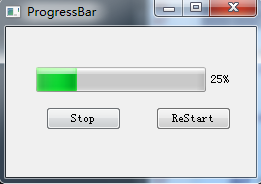
我们使用定时器对象激活进度条。

self.timer.start(100, self)

通过调用 start() 方法加载定时器事件，该方法有两个参数，超时时间（ timeout ）和接受事件的对象（ object ）。

**def** timerEvent(self, event):  
 **if** self.step >= 100:  
 self.timer.stop()  
 **return** self.step = self.step + 1  
 self.pbar.setValue(self.step)

每个 QObject 和它的继承者都有个 QObject.timerEvent 事件处理程序。为了应对定时器事件，我们重新实现了该事件处理程序。



## 日历窗口QCalendarWidget

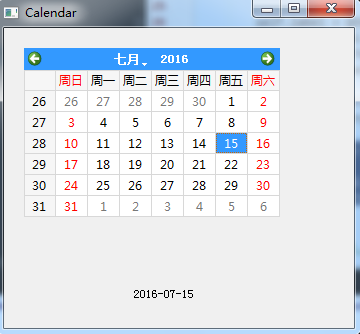
QCalendarWidget 提供基于月份的日历窗口组件，它允许用户简单并且直观的选择日期。

*# -\*- coding: utf-8 -\*-  
  
# calendar.py***import** sys  
**from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
 **def** initUI(self):  
  
 self.cal = QtGui.QCalendarWidget(self)  
 self.cal.setGridVisible(True)  
 self.cal.move(20, 20)  
 self.connect(self.cal, QtCore.SIGNAL(**'selectionChanged()'**),  
 self.showDate)  
  
  
 self.label = QtGui.QLabel(self)  
 date = self.cal.selectedDate()  
 self.label.setText(str(date.toPyDate()))  
 self.label.move(130, 260)  
  
 self.setWindowTitle(**'Calendar'**)  
 self.setGeometry(300, 300, 350, 300)  
  
 **def** showDate(self):  
  
 date = self.cal.selectedDate()  
 self.label.setText(str(date.toPyDate()))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 app = QtGui.QApplication(sys.argv)  
 ex = Example()  
 ex.show()  
 app.exec\_()

如果从日历上选择一个日期， selectionChanged() 信号将会发射。我们连接该方法到自定义的 showDate() 方法上。

**def** showDate(self):  
 date = self.cal.selectedDate()  
 self.label.setText(str(date.toPyDate()))

通过调用 selectedDate() 方法获得日期，然后转换日期对象到字符串并设置到标签上。



## QLineEdit

QLineEdit 窗口组件用来输入或者编辑单行纯文本，有撤销/重做，剪切/粘贴和拖放功能。

*#!/usr/bin/python  
# -\*- coding: utf-8 -\*-  
  
# ZetCode PyQt4 tutorial  
#  
# This example shows text which  
# is entered in a QLineEdit  
# in a QLabel widget.  
#  
# author: Jan Bodnar  
# website: zetcode.com  
# last edited: December 2010***from** PyQt4 **import** QtGui  
**from** PyQt4 **import** QtCore  
  
  
**class** Example(QtGui.QWidget):  
 **def** \_\_init\_\_(self, parent=None):  
 QtGui.QWidget.\_\_init\_\_(self, parent)  
  
 self.initUI()  
  
  
 **def** initUI(self):  
  
 self.label = QtGui.QLabel(self)  
 edit = QtGui.QLineEdit(self)  
  
 edit.move(60, 100)  
 self.label.move(60, 40)  
  
 self.connect(edit, QtCore.SIGNAL(**'textChanged(QString)'**),  
 self.onChanged)  
  
 self.setWindowTitle(**'QLineEdit'**)  
 self.setGeometry(250, 200, 350, 250)  
  
  
 **def** onChanged(self, text):  
 self.label.setText(text)  
 self.label.adjustSize()  
  
  
**def** main():  
  
 app = QtGui.QApplication([])  
 exm = Example()  
 exm.show()  
 app.exec\_()  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

## QSplitter

QSplitter 使得用户可以通过拖动子窗口组件的边界来控制子窗口组件的尺寸。在我们的例子中，我们显示由两个分离器组织的三个 QFrame 窗口组件。

*# -\*- coding: utf-8 -\*-  
  
# ZetCode PyQt4 tutorial  
#  
# This example shows  
# how to use QSplitter widget.  
#  
# author: Jan Bodnar  
# website: zetcode.com  
# last edited: December 2010***from** PyQt4 **import** QtGui, QtCore  
  
  
**class** Example(QtGui.QWidget):  
  
 **def** \_\_init\_\_(self):  
 super(Example, self).\_\_init\_\_()  
  
 self.initUI()  
  
  
 **def** initUI(self):  
  
 hbox = QtGui.QHBoxLayout(self)  
  
 topleft = QtGui.QFrame(self)  
 topleft.setFrameShape(QtGui.QFrame.StyledPanel)  
  
 topright = QtGui.QFrame(self)  
 topright.setFrameShape(QtGui.QFrame.StyledPanel)  
  
 bottom = QtGui.QFrame(self)  
 bottom.setFrameShape(QtGui.QFrame.StyledPanel)  
  
 splitter1 = QtGui.QSplitter(QtCore.Qt.Horizontal)  
 splitter1.addWidget(topleft)  
 splitter1.addWidget(topright)  
  
 splitter2 = QtGui.QSplitter(QtCore.Qt.Vertical)  
 splitter2.addWidget(splitter1)  
 splitter2.addWidget(bottom)  
  
 hbox.addWidget(splitter2)  
 self.setLayout(hbox)  
  
 self.setWindowTitle(**'QSplitter'**)  
 *# QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('Cleanlooks'))* self.setGeometry(250, 200, 350, 250)  
  
  
**def** main():  
  
 app = QtGui.QApplication([])  
 ex = Example()  
 ex.show()  
 app.exec\_()  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

例子中有三个框架窗口组件和两个分离器。

topleft = QtGui.QFrame(self)  
topleft.setFrameShape(QtGui.QFrame.StyledPanel)

为了看到 QFrame 之间的边界，我们使用带样式的框架。

splitter1 = QtGui.QSplitter(QtCore.Qt.Horizontal)  
splitter1.addWidget(topleft)  
splitter1.addWidget(topright)

创建一个 QSplitter 并加入两个 框架。

splitter2 = QtGui.QSplitter(QtCore.Qt.Vertical)  
splitter2.addWidget(splitter1)  
splitter2.addWidget(bottom)

也可以把一个分离器加到另一个分离器窗口组件中。

