Qt designer 设计流程：

创建一个 PyQt4 的步骤，通常的方法是用 QtDesigner 工具创建 GUI 界面。可以在窗口上添加部件，并可以对部件进行一些属性配置。一般的过程如下：

1、 使用 QtDesinger 创建 GUI 界面；
2、 在属性编辑器中修改部件 widget 的名字；
3、 使用 pyuic4 工具生成一个 python 类
4、 通过 GUI 对应的类来运行程序；
5、 通过设置自己的 slots 来扩展功能；
6、 当使用窗口部件的时候，可以从 PyQt's Classes 查询。Qt 易于理解的方式来命名函数，例如：setText。

生成的 UI 文件如 test.ui 需要转化为.py 文件，命名格式不固定，eg：Ui_test.py 或者 test_Ui.py。此文件中会有一个 UI 对象类（用户接口类）叫法不严谨，知道是 ui 文件转化过来的一个对象就行了，Ui_Form。Ui_Form 中包含了窗口部件的属性操作，用户可以进一步对部件的属性进行修改和更新。

```
from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName(_fromUtf8("Form"))
        Form.resize(400, 300)
        Form.setWindowTitle(QtGui.QApplication.translate("Form", "Form",
None, QtGui.QApplication.UnicodeUTF8))
        self.PushButton = QtGui.QPushButton(Form)
        self.PushButton.setGeometry(QtCore.QRect(70, 50, 271, 201))
        self.PushButton.setText(QtGui.QApplication.translate("Form",
"PushButton_ZTE", None, QtGui.QApplication.UnicodeUTF8))
        self.PushButton.setObjectName(_fromUtf8("PushButton"))

        self.retranslateUi(Form)
        QtCore.QObject.connect(self.PushButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), Form.close)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
            pass
```

如果要对这个窗口进行运行显示，还需要把这个 UI 对象类导入到一个新的.py 文件，

窗口类中需要调用这个 UI 对象类，该窗口类从 QtGui.QMainWindow 继承而来。

下面的代码是我们创建的窗口类的.py 文件，test.py，创建过程如下：

1、导入要用到的模块

2、创建从 QtGui.QMainWindow 继承而来的类，并给该类起一个类名；在函数__init__()中完成对窗口类的一些初始化工作。首先是父类的一个初始化 QtGui.QWidget.__init__(self, parent) ，创建 UI 类对象并进行初始化的工作。这个格式基本上是固定的。

3、__name__ == “__main__”

4、创建一个应用程序对象 app = QtGui.QApplication(sys.argv)所有的 GUI 程序必须有这条语句。创建刚才定义的窗口类对象，并进行显示；然后是程序的一个执行和退出操作。

```
import sys
from PyQt4 import QtCore, QtGui
from test_ui import Ui_Form

class MyForm(QtGui.QMainWindow):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.ui = Ui_Form()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    myapp = MyForm()
    myapp.show()
    sys.exit(app.exec_())
```

信号/槽

信号/槽的概念理解：所谓信号就是当用户操作窗口中的部件时（点击、鼠标划过等），会触发一个事件消息，这个事件消息就称之为信号；槽是信号对应的函数，称为槽函数。这里需要注意的是事件信号与槽 slot 函数的一个连接操作：

```
QtCore.QObject.connect(self.ui.button_open,QtCore.SIGNAL("clicked()"), self.file_dialog)
```

self.ui.button_open 发送信号的部件对象

QtCore.SIGNAL("clicked()")事件的类型

self.file_dialog 关联的事件响应函数

```
def file_dialog(self):#响应函数的一个定义，和类中普通的函数一样。
```

有参数的 Singals/Slots，有些部件需要一些参数信息来表明部件的一个属性值。例如，QtabWidget 的第几个 tab 页，每个 tab 页从 0 开始顺序编号。在连接 Singals/Slots 的时候，给 Slots 函数增加一个参数：

```
QtCore.QObject.connect(self.ui.pkgTabs,QtCore.SIGNAL("currentChanged(int)"),self.tab_change)
```

self.ui.pkgTabs 发送信号的部件对象

QtCore.SIGNAL("currentChanged(int)")事件的类型，可以定义自己的事件类型，注意参数的传递

self.tab_change 关联的事件响应函数

def tab_change(self, tab_id): 响应函数的定义，需要传入的参数 tab_id

print tab_id

PyQt4 工具包简介

PyQt 是用来创建 GUI 应用程序的工具包。PyQt 是一个高度抽象的工具包。

PyQt 是 Python 编程语言与已获成功的 Qt 库的混合体。

PyQt 的实现被视作 Python 的一个模块。由 300 多个类和 6000 多个函数与方法构成。

PyQt 是跨平台的工具包，可以在所有主流的操作系统上运行（unix、Windows、Mac）

PyQt 有大量的类，被划分为如下几个模块：

QtCore 模块：包含了核心的非 GUI 功能函数，用于以下方面：日期、文件和目录、数据结构、数据流、URL、MIME、线程和进程。

QtGui 模块：包含了绘图组件以及与绘图相关的类，比如按钮、窗口、状态栏、工具栏、滑块、位图、颜色、字体等。

QtNetWork 模块：包含用于网络编程的类，用户可以用这些类实现 TCP/IP 和 UDP 的客户端或服务器。并且使用这些类会使网络编程更加容易、轻便。

QtXml 模块：包含用于处理 XML 文件的类，该模块提供了 SAX 和 DOM API 两种 XML 文件处理方式的实现。

QtSvg 模块：包含了用于显示 SVG 文件内容的类。（可缩放矢量图形，参考 http://zh.wikipedia.org/wiki/SVG）

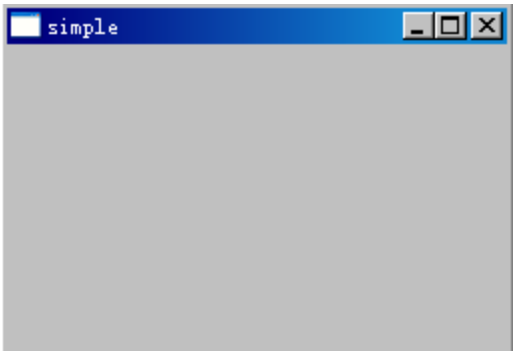
QtOpenGL 模块：用于渲染使用 OpenGL 库创建的 3D 或 2D 图形。并且它支持 Qt GUI 库和 OpenGL 库的无缝结合。

QtSql 则库提供了用于操作数据库的类。

1、一个简单的窗口界面，代码如下：

```
# simple.py
import sys
from PyQt4 import QtGui          基本的窗口部件在 QtGui 模块中
app = QtGui.QApplication(sys.argv)
widget = QtGui.QWidget()
#窗口属性的一个操作
widget.resize(250, 150)
widget.setWindowTitle('simple')

widget.show()
sys.exit(app.exec_())
```

程序截图：



app = QtGui.QApplication(sys.argv)

每一个 PyQt4 程序都需要有一个 application 对象，application 类包含在 QtGui 模块中。sys.argv 参数是一个命令行参数列表。Python 脚本可以从 shell 中执行，参数可以让我们选择启动脚本的方式。

widget = QtGui.QWidget()

QWidget 部件是 PyQt4 中所有用户界面类的父类。这里我们使用没有参数的默认构造函数，它没有继承其它类。我们称没有父类的 widget 为一个 window。

sys.exit(app.exec_())

最后我们进入该程序的主循环。事件处理从本行语句开始。主循环接受事件消息并将其分发给程序的各个部件。如果调用 exit()或主部件被销毁，主循环就会结束。使用 sys.exit()方法退出可以确保程序可以完整的结束，这种情况下系统的环境变量会记录程序是如何退出的。

也许你会疑惑，为什么 exec_()方法会有一个下划线。这是因为 exec 是 Python 的关键字，为避免冲突，PyQt 使用 exec_()替代。

2、程序图标

```
# icon.py
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/web.png'))

app = QtGui.QApplication(sys.argv)
icon = Icon()
icon.show()
sys.exit(app.exec_())
```

Python 语言同时支持面向过程和面向对象两种编程方法。

PyQt 编程是面向对象的。

```
class Icon(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)
```

面向对象编程中最重要的是类、属性和方法。创建了一个名为 Icon 的新类，该类继承 QtGui.QWidget 类。因此必须调用两个构造函数——Icon 的构造函数和继承类 QtGui.QWidget 类的构造函数。

self.setGeometry(300, 300, 250, 150)

self.setWindowTitle('Icon')

self.setWindowIcon(QtGui.QIcon('icons/web.png'))

setGeometry()方法完成两个功能——设置窗口在屏幕上的位置和设置窗口本身的大小。前两个参数是窗口在屏幕上的 x 和 y 坐标。后两个参数是窗口本身的宽和高。

setWindowIcon()方法用来设置程序图标，它需要一个 QIcon 类型的对象作为参数。QIcon 构造函数时，需要提供要显示的图标的路径（相对或绝对路径）。

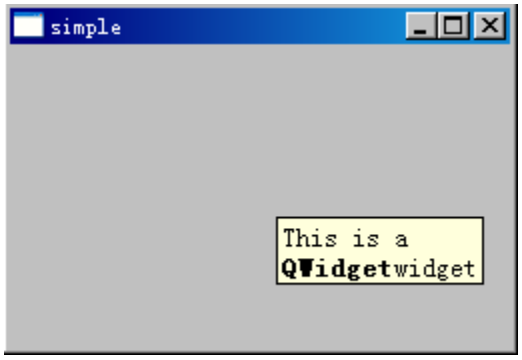
3、显示提示信息

在初始化函数__init__(self, parent = None)中增加两行初始化代码：

self.setToolTip('This is a QWidget widget')

QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))

程序截图：



self.setToolTip('This is a QWidget widget')

要创建工具提示，则需要调用 setToolTip()方法。该方法接受富文本格式的参数。

QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))

因为默认的 QToolTip 字体看起来比较糟糕，我们可以通过上面的语句设置想要的字体和字体大小。

4、关闭窗口

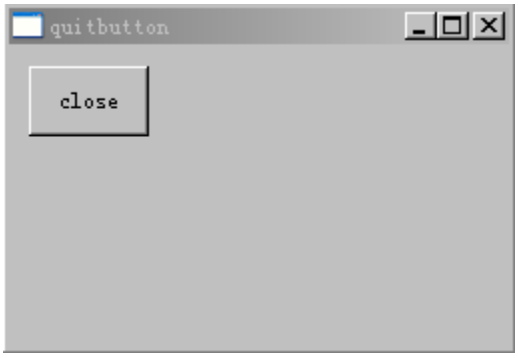
QPushButton 的构造函数：QPushButton(string text, QWidget parent = None)

text 表示将显示在按钮上的文本。parent 是其对象，用于指定按钮显示在哪个部件中。在我们的示例中，parent 是一个 QWidget 对象。

```
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('quitbutton')

        quit = QtGui.QPushButton('close',self)
```

```
        quit.setGeometry(10,10,60,35)

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

app = QtGui.QApplication(sys.argv)
qt = QuitButton()
qt.show()
    sys.exit(app.exec_())
```

程序截图：




```
  quit = QtGui.QPushButton('close',self)
  quit.setGeometry(10,10,60,35)
```

这两句用来创建一个按钮，并把它放在 QWidget 部件上。就像把 QWidget 部件放在屏幕上一样。

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

PyQt4 的事件处理系统建立在信号-槽机制之上。如果我们单击 close 按钮，那么信号 clicked()就会被触发，槽函数可以是 PyQt 自带的槽函数，也可以是任何 Python 可以调用的函数等。QtCore.QObject.connect()方法可以将信号和槽函数连接起来。在我们的示例中槽函数是 PyQt 中已定义的 quit()函数。通过 connect 方法就可以建立发送者（quit 按钮）和接受者（应用程序对象）之间的通信。

5、消息提示框

```
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()
```

如果我们关闭 QWidget 窗口，QCloseEvent 事件就会被触发。要改变原有的 widget 窗口的关闭行为，我们需要截获消息处理函数 closeEvent（），重新对这个方法进行实现。

6、重置窗口到屏幕中间

```
    def center(self):
        screen = QtGui.QDesktopWidget().screenGeometry()
        size = self.geometry()
        self.move((screen.width() - size.width())/2,
```

```
                (screen.height() - size.height())/2)
```

screen = QtGui.QDesktopWidget().screenGeometry()

该语句用来计算出显示器的分辨率（screen.width , screen.height）。

size = self.geometry()

该语句用来获取 QWidget 窗口的大小（size.width, size.height）。

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

该语句将窗口移动到屏幕的中间位置。

PyQt4 中的菜单和工具栏

1、主窗口

QMainWindow 类用来创建应用程序的主窗口。可以创建一个包含菜单来、工具栏、状态栏的经典应用程序框架。

2、状态栏 statusBar

用来显示状态信息的串口部件。

```
import sys
from PyQt4 import QtGui

class MainWindow(QtGui.QMainWindow):
    def __init__(self, parent = None):
        QtGui.QMainWindow.__init__(self,parent)

        self.resize(250,150)
        self.setWindowTitle("MainWindow")
        self.statusBar().showMessage("Ready")

app = QtGui.QApplication(sys.argv)
mw = MainWindow()
mw.show()
sys.exit(app.exec_())
```

运行截图：

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

使用 QApplication 类的 statusBar()方法创建状态栏。使用 showMessage()方法将信息显示在状态栏中。

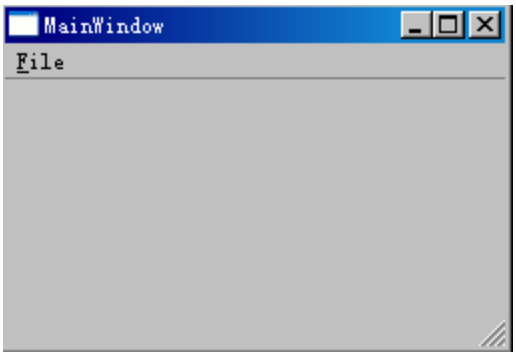
3、菜单栏 addMenu

菜单栏是 GUI 程序最明显的组成部分。它由一组位于不同菜单中的命令组成。在控制

台程序中，我们必须记住那些晦涩难懂的命令。但在 GUI 程序中，通过菜单栏我们将命令合理的放置在不同的菜单中来降低学习新应用程序的时间开销。

```
    menubar = self.menuBar()
    file = menubar.addMenu('&File')

    exit = QtGui.QAction(QtGui.QIcon(),'Exit',self)
    exit.setShortcut('Ctrl+Q')
    exit.setStatusTip('Exit application')
    exit.connect(exit,QtCore.SIGNAL('triggered()'),QtGui.qApp,QtCore.
SLOT('quit()'))

    file.addAction(exit)
```

程序截图：



首先我们使用 QMainWindow 类的 menuBar()方法创建一个菜单栏。然后使用 addMenu()方法添加一个菜单。最后我们把动作对象（这里是 exit）添加到 file 菜单中。

创建菜单栏的步骤（与上面的创建方式不大一样 PyQt 里的方法）：

1、创建一个菜单栏；　　`self.menubar = QtGui.QMenuBar(MainWindow)`

2、在菜单栏上创建菜单项，并对菜单项起一个名称，菜单项可以有多个；

```
    self.menu_F = QtGui.QMenu(self.menubar)
    self.menu_F.setTitle(QtGui.QApplication.translate("MainWindow",
"&File", None, QtGui.QApplication.UnicodeUTF8))
        self.menu_F.setObjectName(_fromUtf8("menu_F"))
```

3、将菜单项设置在主窗口中

```
    MainWindow.setMenuBar(self.menubar)
```

4、工具栏 addToolBar

菜单对程序中的所有命令进行分组放置，而工具栏是提供快速执行常用命令的方法。

```
    exit = QtGui.QAction(QtGui.QIcon(),'Exit',self)
    exit.setShortcut('Ctrl+Q')
    exit.setStatusTip('Exit application')

    exit.connect(exit,QtCore.SIGNAL('triggered()'),QtGui.qApp,QtCo
    re.SLOT('quit()'))

    self.toolbar = self.addToolBar('Exit')
    self.toolbar.addAction(exit)
```

GUI 应用程序的行为是由命令来控制的，这些命令可以来自菜单、上下文菜单、工具栏或它们的快捷方式。PyQt 通过引入 actions 来简化编程难度，一个 action 对象可以拥有菜单、文本、图标、快捷方式、状态信息、“这是什么？”文本或工具提示等。在我们的示例程序中，我们定义了一个拥有图标、工具提示和快捷方式的 action 对象。

self.connect(self.exit, QtCore.SIGNAL('triggered()'), QtGui.qApp, QtCore.SLOT('quit()'))

该语句将 action 对象的 triggered()信号连接到预定义的 quit()槽函数。

self.toolbar = self.addToolBar('Exit')

该语句创建一个工具栏，然后使用语句 self.toolbar.addAction(exit)将 action 对象添加到该工具栏（这里是 exit）。

PyQt4 中的布局管理器

布局管理是指在窗口中安排部件位置的方法。

布局管理有两种工作方式：绝对定位方式（absolute positioning）和布局类别方式（layout classes）。

1、绝对定位方式

该方式是指程序员指定每个部件的位置和尺寸像素。当使用绝对定位方式编程时需要注意一下几点：

- 改变窗口大小时，窗口中部件的大小和位置不会随之改变。
- 在不同的平台上，应用程序可能会看起来不尽相同。
- 在应用程序中改变字体可能会导致布局混乱。
- 如果你打算改变窗口布局，你就必须得重新书写所有部件的布局，这一工作会非常乏味且耗时较多。

label = QtGui.QLabel('care', self)

label.move(35, 40)

简单使用 move()方法来设置部件的位置。通过 x 和 y 坐标来指定 QLabel 部件的位置，坐标起点为左上角的顶点。x 坐标从左向右增长，y 坐标从上向下增长。

实现步骤：	1、创建部件 2、将部件移到窗口的坐标点，用 move()方法来实现。

2、Box 布局

使用布局类别方式的布局管理器比绝对布局管理器更加灵活实用。它是窗口部件首选的布局管理方式。最基本的布局类别是 QHBoxlayout 和 QVBoxlayout 布局管理方式，分别将窗口部件水平和垂直排列。

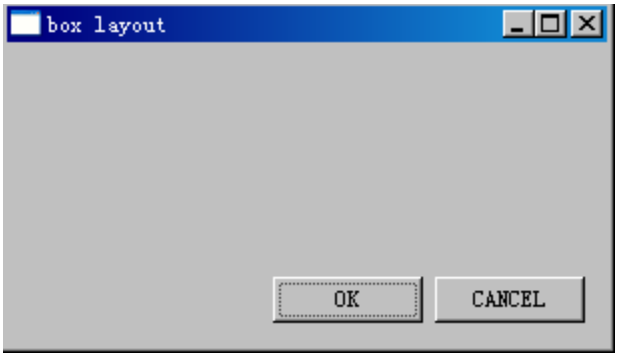
Eg：将两个按钮放在窗口的右下角。

为创建该布局，我们需要使用一个水平 Box 和一个垂直 Box，另外为了创建必须的空白空间，我们还需要添加一个伸缩间隔元素（stretch factor）。

```
import sys
from PyQt4 import QtGui

class BoxlayOut(QtGui.QWidget):
    def __init__(self,parent = None):
        QtGui.QWidget.__init__(self)

        self.setWindowTitle("box layout")
```

```
        ok = QtGui.QPushButton('OK')
        cancel = QtGui.QPushButton('CANCEL')

        hBox = QtGui.QHsaaaaBoxLayout()
        hBox.addStretch(1)
        hBox.addWidget(ok)
        hBox.addWidget(cancel)

        vBox = QtGui.QVBoxLayout()
        vBox.addStretch(1)
        vBox.addLayout(hBox)

        self.setLayout(vBox)
        self.resize(300,150)

app = QtGui.QApplication(sys.argv)
BL = BoxlayOut()
BL.show()
sys.exit(app.exec_())
```

程序截图：




```
ok = QtGui.QPushButton('OK')
cancel = QtGui.QPushButton('CANCEL')
```

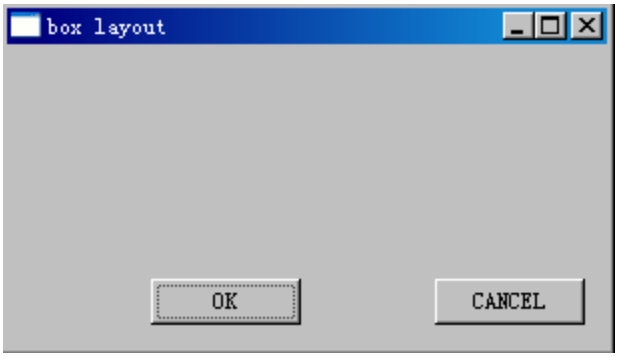
用来创建两个按钮（OK 和 CANCEL）

```
        hBox = QtGui.QHBoxLayout()
        hBox.addStretch(1)
        hBox.addWidget(ok)
        hBox.addWidget(cancel)
```

创建一个水平 box 布局，增加一个伸缩间隔元素和两个按钮。（按钮左边增加一个可伸缩的空单元），窗口布局格式如上图所示，如果更改代码如下：

```
        hBox = QtGui.QHBoxLayout()
        hBox.addStretch(1)
        hBox.addWidget(ok)
        hBox.addStretch(1)
        hBox.addWidget(cancel)
```

程序运行截图：



可以看到在窗口底部的左边和两个按钮中间各增加了一个可伸缩间隔元素。

```
vBox = QtGui.QVBoxLayout()
vBox.addStretch(1)
vBox.addLayout(hBox)
```

创建一个垂直 Box 布局管理器，在上面增加一个可伸缩间隔元素，并将水平 Box 布局放入垂直 Box 布局中。

```
self.setLayout(vBox)
```

最后，设置窗口的主布局。

Box 布局实现步骤：　1、创建一个水平或垂直 Box 布局
2、使用 addWidget()方法将部件添加到布局中
3、使用 addStretch()方法在部件之间增加可伸缩间隔
4、将创建的 Box 布局添加到我们的部件中，addLayout()

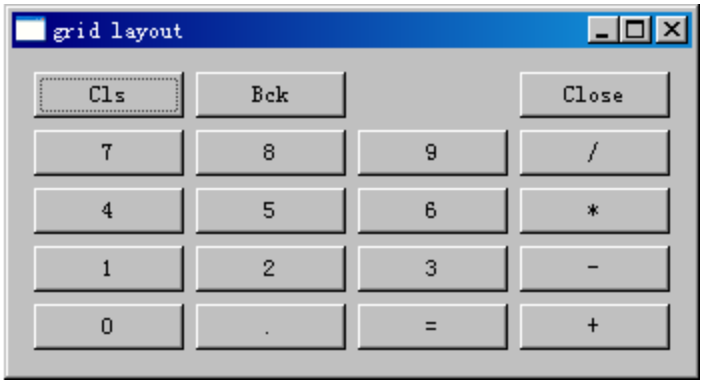
3、网格布局

最通用的布局类别是网格布局（QGridLayout）。该布局方式将窗口空间划分为许多行和列。要创建该布局方式，我们要用到 QGridLayout 类。

```
import sys
from PyQt4 import QtGui

class GridLayout(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(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 2 == j:
                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)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

程序截图：



为了填补 Bck 和 Close 按钮之间的空白，我们使用 QLabel 部件。

```
grid = QtGui.QGridLayout()
```

该语句创建一个网格布局。

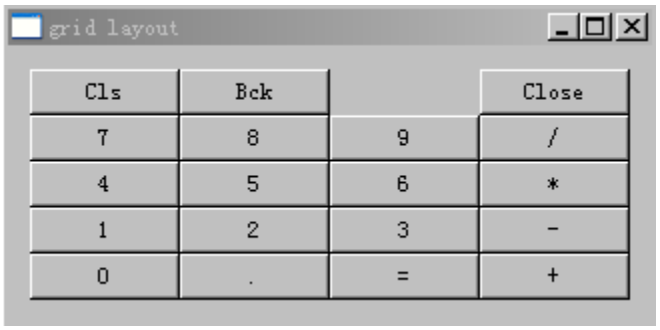
```
            if 2 == j:
                grid.addWidget(QtGui.QLabel(' '),0,2)
            else:
                grid.addWidget(button,pos[j][0],pos[j][1])
```

使用 addWidget()方法，将部件加入到网格布局中，addWidget()方法参数依次为要加入部件的对象名，行号和列好。

网格布局实现步骤：　1、创建网格布局对象 QtGui.QGridLayout()
2、将创建好的部件添加到网格布局对象中，调用 addWidget()方法
3、将网格布局对象设置到该部件中，调用 setLayout()

```
grid.setSpacing(0)
```

设置网格布局中部件之间的间隔（同行的横向间隔），这里设置为 0，其运行结果如下：



PyQt4 的事件与信号

1、事件

事件（Events）是 GUI 程序中很重要的一部分。它由用户或系统产生。当我们调用程序的 exec_()方法时，程序就会进入主循环中。主循环捕获事件并将它们发送给相应的对象进行处理。为此，奇趣公司（Trolltech）引入了信号与槽机制。

2、信号和槽

当用户单击一个按钮，拖动一个滑块或进行其它动作时，相应的信号就会被发射。除此之外，信号还可以因为环境的变化而被发射。比如一个运行的时钟将会发射时间信号等。而所谓的槽则是一个方法，该方法将会响应它所连接的信号。在 Python 中，槽可以是任何可以被调用的对象。

```
import sys
from PyQt4 import QtGui,QtCore

class SigSlot(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self,parent)

        self.setWindowTitle('single & slot')
        lcd = QtGui.QLCDNumber(self)
        slider = QtGui.QSlider(QtCore.Qt.Horizontal,self)

        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.lcd)
        vbox.addWidget(self.slider)

        self.setLayout(vbox)

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

        self.resize(250,150)

app = QtGui.QApplication(sys.argv)
ss = SigSlot()
ss.show()
```

```
sys.exit(app.exec_())
```

在这个示例中，我们创建了一个 LCD 显示器和一个滑块。通过拖动滑块我们可改变 LCD 显示器的数字。

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

这里我们将滑块的 valueChanged()信号连接到 LCD 显示器的 display()槽函数上。连接方法 connect 有 4 个参数：信号发送者对象（这里是 slider 对象），要发射的信号（这里是 valueChanged 信号），信号的接收者对象（这里是 lcd 对象），对信号做出响应的槽函数（这里是 display 方法）。

3、重写事件处理方法

```
import sys
from PyQt4 import QtGui,QtCore

class Escape(QtGui.QWidget):
    def __init__(self,parent = None):
        QtGui.QWidget.__init__(self)

        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)
qb = Escape()
qb.show()
sys.exit(app.exec_())
```

在上面的示例中，我们重新实现了 keyPressEvent()事件处理方法。

```
def keyPressEvent(self, event):
        if event.key() == QtCore.Qt.Key_Escape:
            self.close()
```

通过上面的方法，当我们按下 ESC 键时程序就会结束。

4、发射信号

继承自 QtCore.QObject 的对象可以均可以发射信号。如果我们单击一个按钮，那么一个 clicked()信号就会被触发。在接下来的示例中，我们将学习如何手动发射一个信号。

```
self.connect(self,QtCore.SIGNAL('closeEmitApp()'),QtCore.SLOT('close()'))
    def mousePressEvent(self,event):
        self.emit(QtCore.SIGNAL('closeEmitApp()'))
```

创建了一个新的信号 closeEmitApp()，该信号在按下鼠标事件发生时被发射。

```
self.emit(QtCore.SIGNAL('closeEmitApp()'))
```

使用 PyQt 内建的 emit 函数发射信号 closeEmitApp()。

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

使用 connect 函数将手动创建的 closeEmitApp()信号和程序的 close()槽函数连接起来。这样在用户按下鼠标的任意键时，程序就会结束。

PyQt4 中的对话框

对话窗口和对话框是现代 GUI 应用程序必不可少的一部分。对话的形式有在输入框内键如内容，修改已有的数据，改变应用程序的设置等。对话框在人机交互中扮演着非常重要的角色。

从本质上说，只存在两种形式的对话框：预定义对话框和定制对话框。

1、 预定义对话框

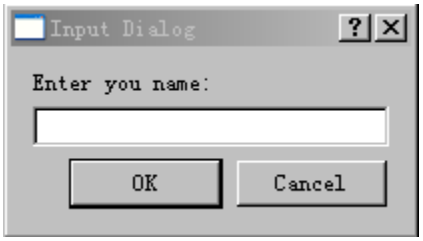
QInputDialog 输入对话框

QInputDialog 提供了一种获取用户单值数据的简洁形式。它接受的数据有字符串，数字和列表中的一项数据等。

```
        text,ok = QtGui.QInputDialog.getText(self, 'Input Dialog', 'Enter
you name:')
        if ok:
            self.label.setText(unicode(text))
```

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

该语句用来显示一个输入对话框。第一个参数'Input Dialog'是对话框的标题。第二个参数'Enter your name'将作为提示信息显示在对话框内。该对话框将返回用户输入的内容和一个布尔值，如果用户单击 OK 按钮确认输入，则返回的布尔值为 true，否则返回的布尔值为 false。



2、 定制颜色对话框

QColorDialog 颜色对话框

QcolorDialog 提供了用于选择颜色的对话框。

使用颜色对话框 QcolorDialog，我们可以改变 widget 部件的背景色。

```
        colordialog = QtGui.QColorDialog
        col = colordialog.getColor()
```

这两行语句用来弹出颜色对话框，并回去颜色对话框的颜色返回值。

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

以上的语句首先检测颜色是否可用。如果用户单击了颜色对话框的取消按钮，则对话框将不返回任何可用的颜色。如果颜色可用，我们就使用 stylesheets 设置 widget 部件的背景色。

3、 定制的字体对话框

```
        font, ok = QtGui.QFontDialog.getFont()
         if ok:
```

```
            self.label.setFont(font)
```

font, ok = QtGui.QFontDialog.getFont()

该语句将弹出字体对话框。

if ok:

    self.label.setFont(font)

用户在选择了字体后单机 OK 按钮，使用标签对象的 setFont 方法设置标签内容的字体。

4、 QFileDialog 文件对话框

文件对话框允许用户选择文件或文件夹，被选择的文件可进行读或写操作。

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

本示例程序是基于 QMainWindow 窗口部件的，因为我们需要将文本编辑器设置为中心部件（QWidget 部件类没有提供 setCentralWidget 方法）。无须依赖布局管理器。

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

该语句将弹出文件对话框。getOpenFileName()方法的第一个字符串参数'Open File'将显示在弹出对话框的标题栏。第二个字符串参数用来指定对话框的工作目录。默认情况下文件过滤器被设置为不过滤任何文件（所有工作目录中的文件/文件夹都会被显示）。

```
        file = open(filename)
        data = file.read()
        self.textEdit.setText(data)
```

以上三行语句将读取被选择的文件并将其内容显示在文本编辑器中