# [pyqt4学习之一：搭建环境和入门](http://www.cnblogs.com/zouzf/p/4308912.html)

 还在继续写Python小工具，想起之前用Tkinter被坑得半死，决定换个框架写UI，又想顺便了解一下qt，就学习一下pyqt4

**搭建环境**

win：现在安装包 <http://www.riverbankcomputing.co.uk/software/pyqt/download> ，测试环境是win8 64位、Python2.7， 所以下载了 [PyQt4-4.11.3-gpl-Py2.7-Qt4.8.6-x64.exe](http://sourceforge.net/projects/pyqt/files/PyQt4/PyQt-4.11.3/PyQt4-4.11.3-gpl-Py2.7-Qt4.8.6-x64.exe) ，一路next就行了，很简单，需要注意的是，安装包会自动搜索Python的安装位置，然后安装到Python27\Lib\site-packages\PyQt4 目录下。 参考： <http://www.2cto.com/kf/201111/111625.html>

 mac：需要下载Qt、Sip、pyqt4，本地环境是 OS 10.9.2 64位， qt 5.3， qt自行下载自行安装即可

sip 下载地址：[http://www.riverbankcomputing.co.uk/software/sip/download](http://www.riverbankcomputing.co.uk/software/sip/download" \t "_blank)  我下载的是 Source Packages 里的 sip-4.16.6.tar.gz，安装：

[复制代码](javascript:void(0);)

1 cd sip-4.16.6

2

3 python configure.py -d /Library/Python/2.7/site-packages --arch=x86\_64

4

5 make

6

7 sudo make install

[复制代码](javascript:void(0);)

pyqt下载地址：[http://www.riverbankcomputing.com/software/pyqt/download](http://www.riverbankcomputing.com/software/pyqt/download" \t "_blank)    我下载的是 Development Snapshots 下的 PyQt-mac-gpl-4.11.4-snapshot-8313084030e1.tar.gz。  安装如下（pyqt的目录和qt的安装目录要改成你具体的目录）：

[复制代码](javascript:void(0);)

1 cd PyQt-mac-gpl-4.11.4-snapshot-8313084030e1

2

3 python configure-ng.py -q /Users/watsy/Qt5.3/5.3/clang\_64/bin/qmake -d /Library/Python/2.7/site-packages/ --sip /System/Library/Frameworks/Python.framework/Versions/2.7/bin/sip

4

5 make

6

7 sudo make install

[复制代码](javascript:void(0);)

mac环境搭建参考： <http://blog.csdn.net/watsy/article/details/8857252>

**Python使用Qt的UI文件**

搭建好环境后，就可以用 designer设计器 来便捷地创建UI界面了，windows下在 Python27\Lib\site-packages\PyQt4 目录下，mac下在qt的安装目录 Qt5.3.0/5.3/clang\_64/bin 下

 designer生成的UI文件的的后缀是 .ui  ，命令： pyuic4 -x aaaaaaa.ui -o bbbbbb.py  即可把.ui文件转成py文件了。（可以先cd到.ui文件所在的目录，也可以直接输入.ui 文件的绝对路径）

注意： windows下可以打开 cmd.exe 直接输入pyuic4命令，mac下要设置一下pyuic4环境变量，否则就只能带上路径了，如：

/Volumes/mac\_data/soft/PyQt-mac-gpl-4.11.4-snapshot-83d48b092f7c/pyuic4 -x aaaaaaa.ui -o bbbbbb.py

**Python pyqt connection 自定义参数**

在响应一些控件的信号时，如果想传一些自定义参数进去，需要对信号进行封装，如：

[复制代码](javascript:void(0);)

1 self.callback1 = partial(self.on\_click, u"走364351asdf")

2 #self.connect(ui.pushButton, SIGNAL("clicked()"), self.callback1)

3 ui.pushButton.clicked.connect(self.callback1)

4

5

6 def on\_click\_common1(self, str1):

7 print(str1)

[复制代码](javascript:void(0);)

详细可以参考：<http://askandstudy.blog.163.com/blog/static/1997520582012013195282/>

# [PyQt的学习（1） 入门](http://blog.csdn.net/vagrxie/article/details/4272009)

# 动机：

刚学习完Python,借鉴以前学习C++时的经验，当时用MFC巩固了C++的学习，并且可以做出实际的程序，给了我继续学习的动力，学习Qt也是出于类似的目的。

为什么我要写，为了学习。不仅仅为了将来有个可以用的，好用的跨平台GUI库，也为了用Python快速开发时也能有个GUI库，但是又不喜欢TK，那么，综合考虑，Qt也就是不错的考虑了。

# 综述

这些文章基本遵循《C++ GUI Qt 4编程》第2版（C++ GUI Programming with Qt4, Second Edition）的流程，文字可能会很少，除非自己想出的例子，不然例子就是书中C++例子的Python版本，**基本上，可以以此书为蓝本对照这些文章中的例子来看。**

虽然用途不算太大，但是考虑到PyQt的好教材是如此的稀少，这样也不算完全没有价值吧，毕竟还有只明白Python不懂C++的人。

这里有一本PyQt的书《[GUI Programming with Python: Qt Edition](http://www.commandprompt.com/community/pyqt/)》，但是老到基本上是属于Qt2时代的东西了。

至于PyQt 嘛，你不会不知道到google上搜索一下，取得其下载地址吧？安装方式如此简单，点击下一步以后，就可以在Python中通过import PyQt4来使用这个库了。

# HelloWorld

自从K&R后，好像所有的程序语言讲解都是以HelloWorld开始，我们也不免俗吧,我们从HelloQT开始。并且，这个例子也可以作为一个测试程序，测试一下看看安装PyQt是否成功。

HelloQt.pyw

1 import sys  
2 from PyQt4 import QtGui  
3   
4   
5 app = QtGui.QApplication(sys.argv)  
6 label = QtGui.QLabel("Hello Qt!")  
7 label.show()  
8   
9 sys.exit(app.exec\_())

这样，就是一个现实"Hello Qt"的窗口，利用了QLabel部件。--说明一下，Qt中将MFC中常常称作控件（control）的东西称作部件（Widget），其实一回事。

这里也可以看出Qt的足够简洁，我很欣赏，而其对象使用的风格和方式也是比较好的，不然，尝试用VS生成一个类似的MFC程序试试？^^

其实这里还不足以见证Qt的强大,看下面的例子，QLabel竟然至此HTML样式。。。。。-\_-!

MoreHelloQt.pyw

1 import sys  
2 from PyQt4 import QtGui  
3   
4   
5 app = QtGui.QApplication(sys.argv)  
6 label = QtGui.QLabel("

HelloQT!

") #这里我没有办法让CSDN不将其解释为HTML。。。所以，参考界面的源代码看看是什么吧  
7 label.show()  
8   
9 sys.exit(app.exec\_())

强大吧：）

# Qt(C++) VS PyQt

这里，顺面比较一下PyQt与普通Qt（C++）生成程序的区别。一般而言，两者速度没有可比性，但是，速度在这里不是主要问题，原因在于PyQt的核心也就是Qt库，那是用C++写的，这样，一般而言不会占用太多时间的逻辑代码速度慢点，不会成为瓶颈，使得PyQt的程序可以跑得足够的快。但是，使用方式上，却没有失去Python的优雅语法，快速开发的能力，也结合了Qt的强大，呵呵，广告用语。。。。。。。。。。。来点实在的。

左边的是用C++开发出来的Qt程序，右边是PyQt开发出来的程序，由于都是使用了同一个库，看不出两者的区别。基本上，多懂了一种语言，就多了一种选择。。。比如说这个时候的Python。

# 建立连接

quit.pyw

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4 app = QtGui.QApplication(sys.argv)  
 5   
 6 quit = QtGui.QPushButton("Quit")  
 7   
 8 QtCore.QObject.connect(quit, QtCore.SIGNAL("clicked()"),  
 9                        app, QtCore.SLOT("quit()"))  
10   
11 quit.show()  
12 sys.exit(app.exec\_())

这里展示了Qt的信槽模式，有点怪异的是，在Python中，信号，槽，都是用字符串来表示-\_-!这点似乎有点奇怪。

我还不懂Qt的原理，也没有看过Qt的源代码，但是总是感觉这里奇怪，于是我翻看了一下Qt的SIGNAL，SLOT宏，于是一切也就没有那么奇怪了。

# define SLOT(a) qFlagLocation("1"#a QLOCATION)

# define SIGNAL(a) qFlagLocation("2"#a QLOCATION)

本来，他们就是以字符串来表示的。。。。。。。。我原本还以为在Qt中这些都是通过回调函数的形式出现的呢。。。。。。。汗-\_-!真那样，与一般的WxWidget原理有啥不同啊。。。呵呵，看来，Qt的原理还得好好理解理解。其实就现在的信息看，无非也就是Observer模式的一种扩展，再难也难不到哪儿去。

在PyQt的安装包中，有个tutorial，展示了更复杂一点的button使用方法，可以参考参考

morequit.pyw

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4   
 5 app = QtGui.QApplication(sys.argv)  
 6   
 7 quit = QtGui.QPushButton("Quit")  
 8 quit.resize(75, 30)  
 9 quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))  
10   
11 QtCore.QObject.connect(quit, QtCore.SIGNAL("clicked()"),  
12                        app, QtCore.SLOT("quit()"))  
13   
14 quit.show()  
15 sys.exit(app.exec\_())

分别是设置Button的大小和按钮文字的字体和样式，这个阶段就不深抠细节了，了解概念和Qt大概的风格就好。

# 窗口部件的布局

用过MFC的人都知道在MFC中创建一个随窗口动态改变控件大小的程序的困难。。。。。。需要每次OnMove,OnSize的时候去重新计算控件的大小，为了保持布局合理，最好还得将所有控件的位置用百分比来计算，痛苦不言而喻，可以参考我写的正则表达式测试程序0.3版（要么就是更老的版本）。但是在Qt中好像就要简单的比较多。看看示例：

layout.pyw

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4 app = QtGui.QApplication(sys.argv)  
 5   
 6 window = QtGui.QWidget()  
 7   
 8 spinBox = QtGui.QSpinBox()  
 9 slider = QtGui.QSlider(QtCore.Qt.Horizontal)  
10 spinBox.setRange(0, 130)  
11 slider.setRange(0, 130)  
12   
13 QtCore.QObject.connect(spinBox, QtCore.SIGNAL("valueChanged(int)"),  
14                        slider, QtCore.SLOT("setValue(int)"))  
15 QtCore.QObject.connect(slider, QtCore.SIGNAL("valueChanged(int)"),  
16                        spinBox, QtCore.SLOT("setValue(int)"))  
17 spinBox.setValue(35)  
18   
19 layout = QtGui.QHBoxLayout()  
20 layout.addWidget(spinBox)  
21 layout.addWidget(slider)  
22 window.setLayout(layout)  
23   
24 window.show()  
25 sys.exit(app.exec\_())

行了，基本上PyQt的程序是怎么样子的，大概有个了解了：），下面就开始慢慢来了。我也需要睡觉去了。

# [PyQt(2) 对话框](http://blog.csdn.net/vagrxie/article/details/4281633)

# 综述

对话框是比较简单的一种GUI形式，说其简单，不仅仅是处于开发的角度，从用户使用的角度来说也是如此，对话框可以说是一般图形界面程序的基本元素，即便是复杂的应用程序，往往也需要对话框来完成一些与用户交互的工程，并且，往往来说，创建基于对话框的程序也是比较简单的。也因为这样，在公司开发的众多工具当中，除了数据校验工具使用了MFC的文档视图（属于密集信息输出型），其他工具清一色的都是对话框，只不过，加了比较多的tab页，同样的强大。

最最简单的Win32汇编程序是一个MessageBox，这是对话框的一种简单形式，Windows中很多好用的公用对话框，也是极大的简化了开发。RAD工具的出现也几乎是为了对话框而设置的，当拖放控件的潮流出现后，很多人就开始设想下一步不用敲代码完成程序的编写了：）

基本上，开发对话框程序的难易，在一定程度上决定着一个GUI库的使用难易程度。下面介绍手动创建一个简单的对话框PyQt程序。

# 最简单形式的对话框：

这里就像是PyQt(1)中的例子quit.pyw一样，那个例子是一个单纯的按钮。。。。。。。。实际上又没有办法突然显示个按钮，所以Qt自动为其加上了一个边框，真正的程序应该是这样的：

simplestdialog.pyw

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4   
 5 app = QtGui.QApplication(sys.argv)  
 6   
 7 dialog = QtGui.QDialog()  
 8 quit = QtGui.QPushButton("Quit",dialog)  
 9 QtCore.QObject.connect(quit, QtCore.SIGNAL("clicked()"),  
10                        app, QtCore.SLOT("quit()"))  
11   
12 dialog.show()  
13 sys.exit(app.exec\_())

这样两个程序其实差不多，但是仔细看还是有一定区别的。如图：



默认的Qt对话框没有最大化和最小化按钮，而自动为button生成的有，然后button那里button占据了整个用户区域，而对话框这里由其自动生成的大小并没有。

# 从面向过程到面向对象

到目前为止，PyQt(1)中所有的程序都还是用面向过程的方式。大概如PyQt（1）layout.pyw例子中的形式。

在面向过程到面向对象的比较上原来奇趣的一个教程有很有示范性的对比。（reference 3）

比如，面向过程是这样的形式：

 1 #!/usr/bin/env python  
 2   
 3 # PyQt tutorial 3  
 4   
 5   
 6 import sys  
 7 from PyQt4 import QtCore, QtGui  
 8   
 9   
10 app = QtGui.QApplication(sys.argv)  
11   
12 window = QtGui.QWidget()  
13 window.resize(200, 120)  
14   
15 quit = QtGui.QPushButton("Quit", window)  
16 quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))  
17 quit.setGeometry(10, 40, 180, 40)  
18 QtCore.QObject.connect(quit, QtCore.SIGNAL("clicked()"),  
19                        app, QtCore.SLOT("quit()"))  
20   
21 window.show()  
22 sys.exit(app.exec\_())

这是同样程序的面向对象版本：

 1 #!/usr/bin/env python  
 2   
 3 # PyQt tutorial 4  
 4   
 5   
 6 import sys  
 7 from PyQt4 import QtCore, QtGui  
 8   
 9   
10 **class** MyWidget(QtGui.QWidget):  
11     **def** \_\_init\_\_(self, parent=None):  
12         QtGui.QWidget.\_\_init\_\_(self, parent)  
13   
14         self.setFixedSize(200, 120)  
15   
16         self.quit = QtGui.QPushButton("Quit", self)  
17         self.quit.setGeometry(62, 40, 75, 30)  
18         self.quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))  
19   
20         self.connect(self.quit, QtCore.SIGNAL("clicked()"),  
21                      QtGui.qApp, QtCore.SLOT("quit()"))  
22   
23   
24 app = QtGui.QApplication(sys.argv)  
25 widget = MyWidget()  
26 widget.show()  
27 sys.exit(app.exec\_())

以上是两个几乎完全一样的程序，除了实现方式上的不同。

在实现上：

面向过程的方式大概的思路就像是生成一个默认的widget，然后通过函数调用，通过子widget按钮的创建去修饰这个widget，然后显示。

面向对象的方式大概思路就像是先描述我们想要的widget是有一个子widget按钮的，大小是多少。。。。。然后直接创建一个我们描述过的想要的widget，将其显示。

有人说，编程的方式取决于你看世界的方式-\_-!其实这点用于对于面向过程和面向对象是比较合适的。

至于面向过程和面向对象的好处就不在这里费太多口舌了，也超出了本文的讨论范围。但是，即使是在这样简单的一个GUI程序中，不去讨论什么封装，数据隐藏等东西，也可以得出面向对象版本会更加容易复用一点。虽然N多书籍强调了使用面向对象技术最主要的原因不在于提高代码复用率，但是这一点确实是很有用并且最容易通过对比讲解的。比如，假如我们需要两个一样的这样的对话框显示不同标题，同时显示在不同位置的情况。

面向对象版本可以这样实现：

 1 #!/usr/bin/env python  
 2 # PyQt tutorial 4 ext objWindow  
 3   
 4 import sys  
 5 from PyQt4 import QtCore, QtGui  
 6   
 7   
 8 **class** MyWidget(QtGui.QWidget):  
 9     **def** \_\_init\_\_(self, parent=None):  
10         QtGui.QWidget.\_\_init\_\_(self, parent)  
11   
12         self.setFixedSize(200, 120)  
13   
14         self.quit = QtGui.QPushButton("Quit", self)  
15         self.quit.setGeometry(62, 40, 75, 30)  
16         self.quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))  
17   
18         self.connect(self.quit, QtCore.SIGNAL("clicked()"),  
19                      QtGui.qApp, QtCore.SLOT("quit()"))  
20   
21   
22 app = QtGui.QApplication(sys.argv)  
23 widget = MyWidget()  
24 widget.move(350,50)  
25 widget.setWindowTitle("A Widget")  
26 widget.show()  
27 anotherWidget = MyWidget()  
28 anotherWidget.move(50,50)  
29 anotherWidget.setWindowTitle("Another Widget")  
30 anotherWidget.show()  
31 sys.exit(app.exec\_())

尝试想象，面向过程的版本会怎么样？只能将所有的代码全部再复制一次，这面机械的操作我甚至都不想给出例子了。程序员的天性应该讨厌复制！

其实，这个面向对象版本的程序还有点问题，因为原来的程序中将对话框与应用程序混在一起了，所以连接按钮到qApp的退出槽上了，这样当你点击其中一个对话框的时候，两个对话框都关闭了。更加面向对象的做法是将其与对话框连接在一起。将那行代码改为：

self.connect(self.quit, QtCore.SIGNAL("clicked()"),

self, QtCore.SLOT("close()"))

这样就让两个对话框可以分开控制了，呵呵，关于类的职责问题一直是面向对象书籍讨论的重点，在这个简单的演示例子里面奇趣的教程也就没有考虑这么多了，所以才会出现这样的情况。

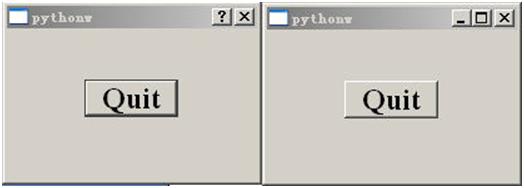
# 面向对象的对话框

前面的例子中，我们的widget是由QWidget继承而来，那么就是一个普通的widget,假如从dialog继承而来，那么就是一个dialog了，这样两者还是有一定区别的。

比如将上述的例子改成对话框。

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4   
 5 **class** MyDialog(QtGui.QDialog):  
 6     **def** \_\_init\_\_(self, parent=None):  
 7         QtGui.QDialog.\_\_init\_\_(self, parent)  
 8   
 9         self.setFixedSize(200, 120)  
10   
11         self.quit = QtGui.QPushButton("Quit", self)  
12         self.quit.setGeometry(62, 40, 75, 30)  
13         self.quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))  
14   
15         self.connect(self.quit, QtCore.SIGNAL("clicked()"),  
16                      self, QtCore.SLOT("close()"))  
17   
18   
19 app = QtGui.QApplication(sys.argv)  
20 dialog = MyDialog()  
21 dialog.show()  
22 sys.exit(app.exec\_())

显示的效果对比与前面的对比很相似：



这里我没有说明哪个是对话框，那个是widget,不过看过前面的对比也很明显了吧。这里有点要说明的是，在Qt中对话框与普通widget的区别与MFC中对话框与文档视图的区别要小多了，在上面的例子中我们也基本可以确定，当我们仅仅是想要显示一个button的时候，实际上Qt是为我们默认生成的是一个widget。

# 更复杂的对话框

《C++ GUI Qt4编程（第二版）》中第2章实现了一个比较复杂的对话框程序，我用Qt Creater尝试了其C++版本，好多代码要敲啊，所以反而丧失了再次敲代码去实现PyQt版本的意思。。。。。。。呵呵，不过，其实知道了大概流程和含义，看到C++版本的Qt程序，再实现一个PyQt版本的程序是非常省事的事情，都差不多。

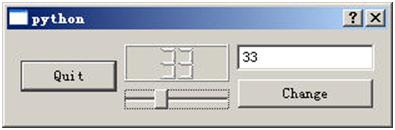
不过，作为学习，我还是实现一个简化版本的PyQt对话框程序吧，不然private slots，Q\_OBJECT，tr在PyQt中对应的东西我还真不知道。

complicateDialog.pyw

 1 import sys  
 2 from PyQt4 import QtCore, QtGui  
 3   
 4 **class** MyDialog(QtGui.QDialog):  
 5     **def** \_\_init\_\_(self, parent=None):  
 6         QtGui.QDialog.\_\_init\_\_(self, parent)  
 7       
 8         self.quit = QtGui.QPushButton("Quit")  
 9           
10         self.change = QtGui.QPushButton("Change")  
11         self.change.setEnabled(False)  
12   
13         # funny widget  
14         self.lcd = QtGui.QLCDNumber(2)  
15   
16         self.slider = QtGui.QSlider(QtCore.Qt.Horizontal)  
17         self.slider.setRange(0, 99)  
18         self.slider.setValue(0)  
19           
20         self.lineEdit = QtGui.QLineEdit()  
21           
22         self.connect(self.quit, QtCore.SIGNAL("clicked()"),  
23                      QtGui.qApp, QtCore.SLOT("quit()"))  
24         self.connect(self.lineEdit, QtCore.SIGNAL("textChanged(const QString&)"),  
25                      self.enableChangeButton)  
26         self.connect(self.slider, QtCore.SIGNAL("valueChanged(int)"),  
27                      self.SliderChange)  
28         self.connect(self.change, QtCore.SIGNAL("clicked()"),  
29                      self.Change)  
30           
31         self.rightLayout = QtGui.QVBoxLayout()  
32         self.rightLayout.addWidget(self.lineEdit)  
33         self.rightLayout.addWidget(self.change)  
34           
35         self.leftLayout = QtGui.QVBoxLayout()  
36         self.leftLayout.addWidget(self.lcd)  
37         self.leftLayout.addWidget(self.slider)  
38           
39         self.layout = QtGui.QHBoxLayout()  
40         self.layout.addWidget(self.quit)  
41         self.layout.addLayout(self.leftLayout)  
42         self.layout.addLayout(self.rightLayout)  
43           
44         self.setLayout(self.layout);  
45           
46     **def** enableChangeButton(self, text):  
47         self.change.setEnabled(text.isEmpty() == False)  
48   
49     **def** Change(self):  
50         value = int(self.lineEdit.text())  
51         self.lcd.display(value)  
52         self.slider.setValue(value)  
53   
54     **def** SliderChange(self):  
55         value = self.slider.value()  
56         self.lcd.display(value)  
57         self.lineEdit.setText(str(value))  
58   
59 app = QtGui.QApplication(sys.argv)  
60 dialog = MyDialog()  
61 dialog.show()  
62 sys.exit(app.exec\_())

我曾经怎么说来着？来个简化版？-\_-!最后写着写着，为了试验新特性，结果说是复杂版也不为过了。

效果如图：



这个程序演示了几个Qt的特性:

1. 很有意思的lcd效果的数字，这种效果竟然原生在Qt的Widget中，很有意思。
2. 其中演示了4个widget的交互，分别是lcd,slider,change,lineEdit其中三个是可以控制的，lcd被动显示，每改动一个，都能在其他三个（两个）中反映出来。控制的方式是通过connect某个singal到对话框类的某个方法上。其中我发现PyQt的connect消息方式很搞笑，为什么这样说呢？因为在Python中，竟然出现了"textChanged(const QString&)"这样的字符串-\_-!再强大的东西，毕竟是通过C++来实现的。。。呵呵，估计非C++背景的PyQt使用者会感到这样的语法比较困惑吧。。。。。。我也比较奇怪，PyQt为啥不干脆一次性将这样的Singal的字符串也改成Python的语法呢？呵呵，也许工作量会更大吧。当然，slot响应的方式倒是简化了一些，当时我还在想，以什么形式将slot的参数用字符串表示出来呢。。。。结果发现，完全不用表示。。。。如源代码所示。这方面有个文章讲的比较详细：[Connecting with signals and slots](http://www.commandprompt.com/community/pyqt/x1408)

1. layout的嵌套，实现了比较复杂的dialog,并且，在代码中我没有实际的通过任何坐标来控制它们，它们是自动放在了我希望它们在的地方，在这一点上，Qt对手工编写代码的友好性胜过MFC又何止百倍啊。

但是，其实上述代码的问题还是有的，比如，lineEdit widget可以输入任何形式的字符，而不仅仅是数字，这在我们这个例子中应该是不允许的，在Qt中也提供了一种远胜MFC的限制方式，不用继承并实现一个自己的lineEdit widget就能实现非常 复杂的限制功能，这和STL中泛型的算法思维有点类似。这就是Qt 中的Validator，功能强大到你甚至可以很简单的就使用正则表达式去限制lineEdit。。。。呵呵，强大。。。这里我根据需要，使用QIntValidator就足够了。

在上例中加入如下代码：

intValidator = QtGui.QIntValidator(0,99, self)

self.lineEdit = QtGui.QLineEdit()

self.lineEdit.setValidator(intValidator)

加在那个位置就不用我教了吧，就是定义self.lineEdit的上下位置。再试试效果，除了0,99的整数，什么都输不进入了。

# 题外话：

在进入使用designer之前，我决定先看看一些关于PyQt的资料,(以前就是看 reference1那本书。单纯这样从C++转换到Python，光靠自己的理解还是难免会有些问题，虽然摸索着上面那样的程序也能写出来：)。但是，就我的经验来说，在使用图形化的设计工具前不将手工代码的原理都弄清楚，以后可能会不再想回过头来彻底弄清楚了，（因为图形化设计会简单的多），但是，碰到稍微有点例外的情况，却会手足无措（使用MFC的经验）。

# reference

1. C++ GUI Qt4编程（第二版） C++ GUI Programming with Qt4, Second Edition。Jasmin Blanchette【加拿大】,Mark Summerfield【英】著，闫锋欣，曾泉人，张志强译，电子工业出版社。
2. [GUI Programming with Python: Qt Edition](http://www.commandprompt.com/community/pyqt/)
3. [Qt tutorial](http://doc.trolltech.com/4.0/tutorial.html)
4. PyQt GPL v4.4.4 for Python v2.6 examples from [riverbankcomputing](http://www.riverbankcomputing.co.uk/)