PyQt 学习笔记(1)——Hello world

PyQt 是 python 和 Qt 的绑定。我们知道，在 python 中有很多 GUI 的库，比如自带的 Tkinter，还有些更加强大的外部库，比如 wxpython，PyGTK，PyQt 等等，其中 PyQt 是基于 Qt 的一个 python GUI 库，可以用于快速开发。

Qt 是很高级的，它的库提供了很多已经设计好了的屏幕对象，控件，和很多的类，由于 Qt 是用 C++写的，而且是跨平台和面向对象的，PyQt 不仅继承了这些优点，还因为 python 的语言的简单而具有了更多的优势。

在 Qt 中，由最原始的基类 Qobject，Qwidget 继承于 Qobject，代表了所有窗口控件，是所有窗口控件的父类，比如 Qlabel，Qdialog 等等。

关于 PyQt 的版本我们现在一般用的是 PyQt4，这个版本是基于 Qt 的版本而变化和开发的。

按照惯例，我们先来个 PyQt 版的 Hello World 程序：

```
#! /usr/bin/env python
#coding=utf-8

import sys
from PyQt4.QtGui import *

app = QApplication(sys.argv)
button = QPushButton("Hello World!")
button.show()
app.exec_()
```

运行结果如下：

和 Tk 以及 Wxpython 差不多，也是先创建一个应用，然后再是消息循环，直到终止，其中的创建控件（按钮）和显示和其它 GUI 类似，最后的 app.exec_() 是消息循环。

PyQt 学习笔记(2)——Dumb Dialogs

对话框在 GUI 编程中是比较重要的控件，这里按照对话框的“智能”水平，把对话框分为”Dumb Dialogs,Standard Dialogs,Smart Dialogs”三种。

首先是 Dumb Dialogs。也就是比较傻瓜式的对话框，看一个例子：

我们要设计一个简单的字体设置对话框，要求可选择字体，和设置字体大小。

程序如下：font.py

```
#!/usr/bin/env python
#coding=utf-8

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

class FontPropertiesDlg(QDialog):
    #初始化
    def __init__(self, parent=None):
       #super 函数完成对话框的初始化
        super(FontPropertiesDlg, self).__init__(parent)
        FontStyleLabel = QLabel(u"中文字体:")
        FontstyleComboBox = QComboBox()
        FontstyleComboBox.addItems([u"宋体", u"黑体", u"仿宋",
                                    u"隶书", u"楷体"])
        FontSizeLabel = QLabel(u"字体大小")
        FontSizeSpinBox = QSpinBox()
        FontSizeSpinBox.setRange(0, 90)
        FontEffectCheckBox =QCheckBox(u"使用特效")
        okButton = QPushButton(u"确定")
        cancelButton = QPushButton(u"取消")
       #确定控件的布局
        buttonLayout = QHBoxLayout()
        buttonLayout.addStretch()
        buttonLayout.addWidget(okButton)
        buttonLayout.addWidget(cancelButton)
        layout = QGridLayout()
        layout.addWidget(FontStyleLabel, 0, 0)
        layout.addWidget(FontstyleComboBox, 0, 1)
        layout.addWidget(FontSizeLabel, 1, 0)
        layout.addWidget(FontSizeSpinBox, 1, 1)
        layout.addWidget(FontEffectCheckBox,1,2)
        layout.addLayout(buttonLayout, 2, 0,1,3)
```

```
            self.setLayout(layout)
        #窗口的标题
            self.setWindowTitle(u"字体")

app = QApplication(sys.argv)
font= FontPropertiesDlg()
font.show()
app.exec_()
```

结果如下：

程序的内容就是创建一个对话框的窗体，其中创建了一些控件，然后通过 layout 方法来在窗体上布局这些控件，最后是创建应用和消息循环。下面看看 layout 是如何布局的。

这里用到了两个 layout 一个是 buttonlayout，一个 layout，其中的 buttonlayout 是放到了 layout 上。

```
buttonLayout = QHBoxLayout()
buttonLayout.addStretch()
buttonLayout.addWidget(okButton)
buttonLayout.addWidget(cancelButton)
```

button 是水平放置的，QHBoxLayout，而且用到了 Stretch，也就是说这里的确定和取消按钮虽然是水平放置的，但是由于 stretch 的作用，两个按钮会靠右放置，而且随着窗口大小的改变，也是靠右边的，如上图。(如果把上面第二句去掉的话，两个按钮的显示就变了，左右平局分布)。

PyQt 学习笔记(3)——Standard Dialogs

一般的标准对话框是什么样子的呢？我们还是以 word 里的字体设置为例。一般都是通过菜单，工具栏，按钮等的响应而创建的对话框，而且对话框分为模态（modal）和非模态（modalless），对于标准对话框，当用户按下确定按钮，对话框消失，并且主窗口得到了用户确认的信息（设置的字体），按取消按钮，对话框消失，没有别的改变。

首先看一个 modal 对话框的例子：用户点击按钮，弹出字体设置对话框，用户点击确定，显示用户的选择，关闭窗口。

```
#!/usr/bin/env python
#coding=utf-8

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

class FontPropertiesDlg(QDialog):

    def __init__(self,parent=None):
        super(FontPropertiesDlg, self).__init__(parent)
        FontStyleLabel = QLabel(u"中文字体:")
        self.FontstyleComboBox = QComboBox()
        self.FontstyleComboBox.addItems([u"宋体",u"黑体", u"仿宋",
                                         u"隶书", u"楷体"])
        self.FontEffectCheckBox =QCheckBox(u"使用特效")
        FontSizeLabel = QLabel(u"字体大小")
        self.FontSizeSpinBox = QSpinBox()
        self.FontSizeSpinBox.setRange(1, 90)
        okButton = QPushButton(u"确定")
        cancelButton = QPushButton(u"取消")

        buttonLayout = QHBoxLayout()
        buttonLayout.addStretch()
        buttonLayout.addWidget(okButton)
        buttonLayout.addWidget(cancelButton)
        layout = QGridLayout()
        layout.addWidget(FontStyleLabel, 0, 0)
        layout.addWidget(self.FontstyleComboBox, 0, 1)
        layout.addWidget(FontSizeLabel, 1, 0)
        layout.addWidget(self.FontSizeSpinBox, 1, 1)
        layout.addWidget(self.FontEffectCheckBox,1,2)
        layout.addLayout(buttonLayout, 2, 0)
        self.setLayout(layout)
```

```
            self.connect(okButton,SIGNAL("clicked()"),self,SLOT("ac
            cept()"))

            self.connect(cancelButton,SIGNAL("clicked()"),self,SLOT
            ("reject()"))
            self.setWindowTitle(u"字体")


    class MainDialog(QDialog):
        def __init__(self,parent=None):
            super(MainDialog,self).__init__(parent)
            self.FontPropertiesDlg=None
            self.format=dict(fontstyle=u”宋
体”,fontsize=1,fonteffect=False)
            FontButton1 = QPushButton(u"设置字体(模态)")
            FontButton2 = QPushButton(u"设置字体(非模态)")
            self.label = QLabel(u"默认选择")
            layout = QGridLayout()
            layout.addWidget(FontButton1,0,0)
            layout.addWidget(FontButton2,0,1)
            layout.addWidget(self.label)
            self.setLayout(layout)

            self.connect(FontButton1,SIGNAL("clicked()"),self.FontM
            odalDialog)
            self.connect(FontButton2,SIGNAL("clicked()"),self.FontM
            odalessDialog)

            self.setWindowTitle(u"模态和非模态对话框")
            self.updataData()

        def updataData(self):
            self.label.setText(u"选择的字体：%s<br>字体大小：%d<br>是否特
效：%s"
%(self.format["fontstyle"],self.format["fontsize"],self.format["f
onteffect"]))
        def FontModalDialog(self):
            dialog = FontPropertiesDlg(self)
            if dialog.exec_():
```

```
            self.format["fontstyle"] =
unicode(dialog.FontstyleComboBox.currentText())
            self.format["fontsize"] =
dialog.FontSizeSpinBox.value()
            self.format["fonteffect"] =
dialog.FontEffectCheckBox.isChecked()
            self.updataData()
        def FontModalessDialog(self):
            pass


    app = QApplication(sys.argv)
    font= MainDialog()
    font.show()
    app.exec_()
```

结果如下：

其中右边的对话框是弹出的模态对话框，通过设置，确定以后可以在主对话框上显示出更新的内容。

其中“设置字体（非模态）“函数没有完成功能。

在模态对话框里面，我们是调用了 dialog 的 exec_()函数，它能产生并显示对话框，并进入消息循环，此时就不能和主窗口进行交互了，只能点击确定或者取消，关闭此模态对话框才能继续。

PyQt 学习笔记(4)——Smart Dialogs

这里的 smart dialogs 一般都是指的非模态对话框，就是用户想要实时的看到自己的操作对主窗口的影响，当然在某些模态对话框中，添加预览功能可以实现上面的需求，但是这些在非模态对话框中是很容易做到的。

我们在前面模态对话框的字体选择程序基础上加以修改。

先写一个类，ModelessDialog.py,如下：

```
#!/usr/bin/env python
#coding=utf-8
```

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

class FontPropertiesDlg(QDialog):

    def __init__(self,format,parent=None):
        super(FontPropertiesDlg, self).__init__(parent)
        self.format=format
        FontStyleLabel = QLabel(u"中文字体:")
        self.FontstyleComboBox = QComboBox()
        self.FontstyleComboBox.addItems([u"宋体", u"黑体", u"仿宋",
                                u"隶书", u"楷体"])
        self.FontEffectCheckBox =QCheckBox(u"使用特效")
        FontSizeLabel = QLabel(u"字体大小")
        self.FontSizeSpinBox = QSpinBox()
        self.FontSizeSpinBox.setRange(1, 90)
        applyButton = QPushButton(u"应用")
        cancelButton = QPushButton(u"取消")

        buttonLayout = QHBoxLayout()
        buttonLayout.addStretch()
        buttonLayout.addWidget(applyButton)
        buttonLayout.addWidget(cancelButton)
        layout = QGridLayout()
        layout.addWidget(FontStyleLabel, 0, 0)
        layout.addWidget(self.FontstyleComboBox, 0, 1)
        layout.addWidget(FontSizeLabel, 1, 0)
        layout.addWidget(self.FontSizeSpinBox, 1, 1)
        layout.addWidget(self.FontEffectCheckBox,1,2)
        layout.addLayout(buttonLayout, 2, 0)
        self.setLayout(layout)

        self.connect(applyButton,SIGNAL("clicked()"),self.apply)
        self.connect(cancelButton,SIGNAL("clicked()"),self,SLOT("re
        ject()"))
```

```
        self.setWindowTitle(u"字体")


    def apply(self):
        self.format["fontstyle"] =
unicode(self.FontstyleComboBox.currentText())
        self.format["fontsize"] = self.FontSizeSpinBox.value()
        self.format["fonteffect"] =
self.FontEffectCheckBox.isChecked()
        self.emit(SIGNAL("changed"))
```

然后我们在上一个模态对话框程序中添加：

```
Import ModelessDialog
```

然后添加函数

```
    def FontModalessDialog(self):
        dialog = ModelessDialog.FontPropertiesDlg(self.format,self)
        self.connect(dialog,SIGNAL("changed"),self.updataData)
        dialog.show()
```

即可。

结果如下：

只要我们点击应用，左边主对话框立刻更新用户选择。我们也可以看到程序中两个最重要的地方：

```
1: self.emit(SIGNAL("changed"))
2: self.connect(dialog,SIGNAL("changed"),self.updataData)
```

可以看到只要用户点击“应用”，后程序发现填出对话框内容发生改变，则立刻产生消息”changed”，主程序消息循环得到这个消息，立刻运行 updataData，所以主窗口更新了内容。

下面我们再看一种比上面非模态对话框更方便快速的方法，就是没有按钮，只要我们改变了对话框的内容，结果就立马显示。我们称这种对话框为”live dialog”。

首先还是先写一个类，单独一个 py 文件：LiveDialog.py

```
#!/usr/bin/env python
#coding=utf-8

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

class FontPropertiesDlg(QDialog):

    def __init__(self,format,callback,parent=None):
        super(FontPropertiesDlg, self).__init__(parent)\

        self.format = format
        self.callback = callback

        FontStyleLabel = QLabel(u"中文字体:")
        self.FontstyleComboBox = QComboBox()
        self.FontstyleComboBox.addItems([u"宋体", u"黑体", u"仿宋",
                                u"隶书", u"楷体"])
        self.FontEffectCheckBox =QCheckBox(u"使用特效")
        FontSizeLabel = QLabel(u"字体大小")
        self.FontSizeSpinBox = QSpinBox()
        self.FontSizeSpinBox.setRange(1, 90)

        layout = QGridLayout()
        layout.addWidget(FontStyleLabel, 0, 0)
        layout.addWidget(self.FontstyleComboBox, 0, 1)
        layout.addWidget(FontSizeLabel, 1, 0)
        layout.addWidget(self.FontSizeSpinBox, 1, 1)
        layout.addWidget(self.FontEffectCheckBox,1,2)
        self.setLayout(layout)
```

```
        self.connect(self.FontstyleComboBox,SIGNAL("itemSelected"),
        self.apply)
```

```
        self.connect(self.FontEffectCheckBox,SIGNAL("toggled(bool)")
        , self.apply)

        self.connect(self.FontSizeSpinBox,SIGNAL("valueChanged(int)
        "), self.apply)
        self.setWindowTitle(u"字体")

    def apply(self):
        self.format["fontstyle"] =
unicode(self.FontstyleComboBox.currentText())
        self.format["fontsize"] = self.FontSizeSpinBox.value()
        self.format["fonteffect"] =
self.FontEffectCheckBox.isChecked()
        self.callback()
```

然后在 font.py 中添加：

```
import LiveDialog
```

在类的 init 中添加：

```
self.FontPropertiesDlg=None
```

然后添加方法：

```
    def FontLiveDialog(self):
        if self.FontPropertiesDlg is None:
            self.FontPropertiesDlg =
LiveDialog.FontPropertiesDlg(self.format,self.updataData,self)
        self.FontPropertiesDlg.show()
        self.FontPropertiesDlg.raise_()
        self.FontPropertiesDlg.activateWindow()
```

运行就可以实时改变用户的选择了。

PyQt 学习笔记(5)——Mian Window

主窗口（MianWindow）一般是应用程序的框架，在主窗口上我们可以添加我们需要的 widget，添加菜单，工具栏，状态栏等等。下面看看在 PyQt 中如何建立主窗口。

```
    # MianWindow.py
    #!/usr/bin/env python
    #coding=utf-8
```

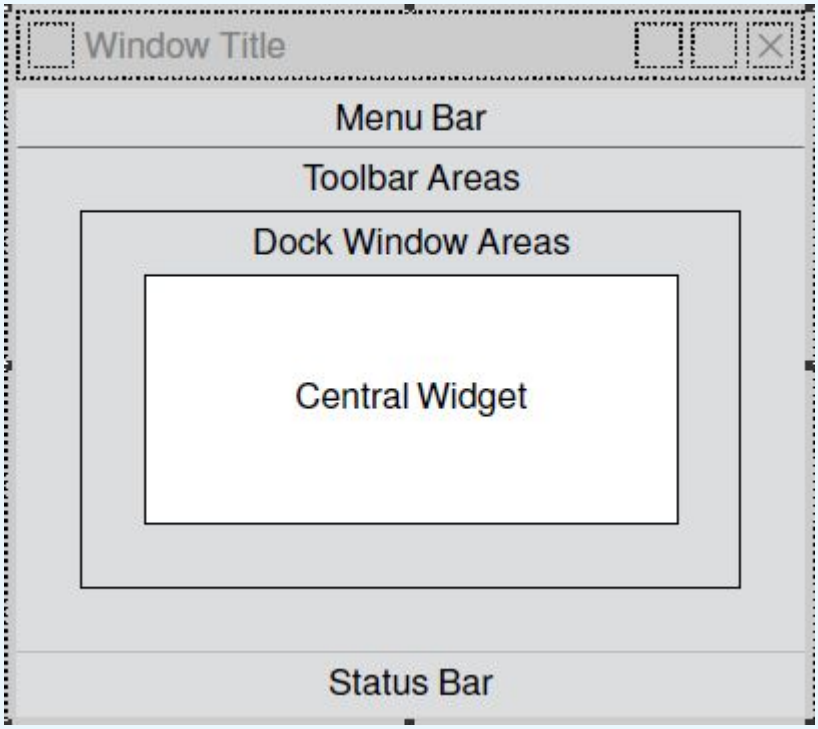
```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self,parent=None):
        super(MainWindow,self).__init__(parent)
        self.status = self.statusBar()
        self.status.showMessage("This is StatusBar",5000)
        self.setWindowTitle("PyQt MianWindow")

def main():
    app = QApplication(sys.argv)
    app.setWindowIcon(QIcon("./images/icon.png"))
    form = MainWindow()
    form.show()
    app.exec_()

main()
```

上面是一个简单的产生 MainWindow 的例子：

在窗口类中，初始化同样是由 super 完成的，然后我们命名了窗口的标题。最后消息循环显示窗口，非常简单。

状态栏可以直接由 statusBar（）产生，showMessage（）来显示消息。

下面看看一个典型的应用程序框架窗口的布局和组成：



可以看到，布局一个框架窗口是一个系统工程，需要多方面的内容。后面会慢慢学习。

PyQt 学习笔记(6)——Actions and Key Sequences

在 Qt 中，目前我认为做的最好的两种机制就是：SIGNAL and SLOTS 和 Actions and Key Sequences。

Actions and Key Sequence 我对他的理解就是各种动作带来的核心一致反应。举个例子：

比如我们在应用程序中常见的“新建文件”这一功能，他的实现可以通过下面几种方式：

1 点击菜单：File->New 菜单项

2 点击工具栏：new 的图标

3 键盘快捷方式：如 Ctrl+N

上面的这三种 actions，其实带来的结果（Sequence）是一样的，就是新建文件，那么在程序中如何做到三者的统一和同步，Qt 给了很好的解决方法。在 PyQt 中，它把类似上面的 actions 封装（encapsulates）到一个 Qaction 的 class 中，下面举个例子：（windows 环境下）

```
#!/usr/bin/env python
#coding=utf-8
```

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

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

            fileNewAction=QAction(QIcon("./images/filenew.png"),"&New",self)
            fileNewAction.setShortcut(QKeySequence.New)
            helpText = "Create a new file"
            fileNewAction.setToolTip(helpText)
            fileNewAction.setStatusTip(helpText)
            self.connect(fileNewAction,SIGNAL("triggered()"),self.fileNew)

            self.fileMenu = self.menuBar().addMenu("&File")
            self.fileMenu.addAction(fileNewAction)

            filetoolbar = self.addToolBar("File")
            filetoolbar.addAction(fileNewAction)

            self.status = self.statusBar()
            self.status.showMessage("This is StatusBar",5000)
            self.setWindowTitle("PyQt MianWindow")

        def fileNew(self):
            self.status.showMessage("You have created a new file!",9000)
    def main():
        app = QApplication(sys.argv)
        app.setApplicationName("PyQt MianWindow")
        app.setWindowIcon(QIcon("./images/icon.png"))
```

```
    form = MainWindow()
    form.show()
    app.exec_()


main()
```

上面程序的目的就是：要让点击菜单 new，Ctrl+N，点击工具栏 new 按钮三种 action 都执行一个命令 fileNew（）。

其中红色部分就是 Qaction 部分，其中的 QKeySequence.New 就是基本多平台都统一使用的新建的响应快捷键 Ctrl+N，如果我们需要的快捷键没有，那么我们可以自己设置，就是填写快捷键的名称比如：fileNewAction.setShortcut（“Ctrl+N”）。把这个 action 都给了菜单 new 和工具栏，通过 connect 绑定，他们都执行同一响应。

从上面可以看到，每次创建一个 QAction 都需要五六行，如果在一个应用程序中都这么创建会很费时间的，所以我们可以写一个函数来封装这一功能：

```
def createAction(self,text,slot=None,shortcut=None, icon=None,
                 tip=None,checkable=False,signal="triggered()"):
    action = QAction(text, self)
    if icon is not None:
        action.setIcon(QIcon("./images/%s.png" % icon))
    if shortcut is not None:
        action.setShortcut(shortcut)
    if tip is not None:
        action.setToolTip(tip)
        action.setStatusTip(tip)
    if slot is not None:
        self.connect(action, SIGNAL(signal), slot)
    if checkable:
        action.setCheckable(True)
    return action
```

有了这个函数以后，我们可以定义上面的 fileNewAction 了：

```
fileNewAction = self.createAction("&New...", self.fileNew,
                    QKeySequence.New, "filenew", "Create an image file")
```

一句话搞定。QKeySequence.New 也可以用”Ctrl+n”代替

PyQt 学习笔记(7)——Qt Designer(1)

我们用 Qt Designer 设计一个对话框。假如我们想设计一个下面的对话框：

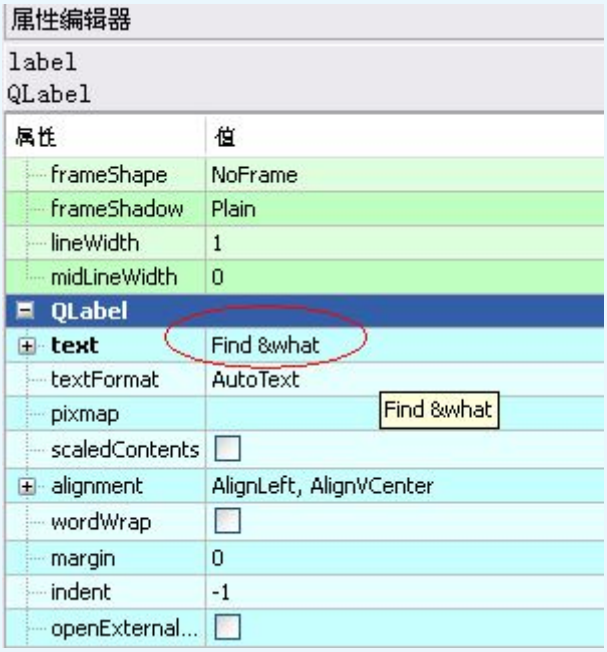
一，创建工程

打开 Designer，在弹出的新建窗体对话框中，选择 templates\forms 中的 Dialog with Button Bottom，点击创建，就会创建一个如下图所示的对话框。

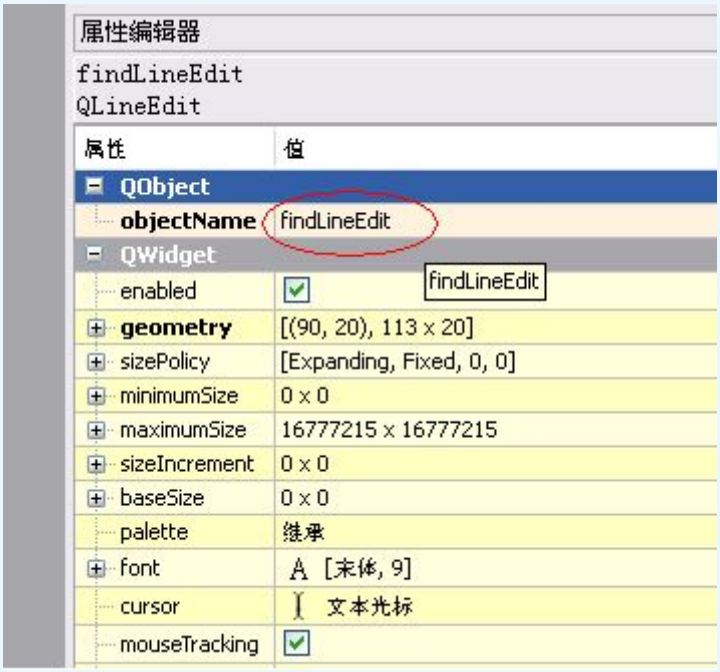
如图我们得到了一个带有按钮的对话框，我们选定两个按钮，然后点击右键，删除这两个按钮，然后我们这个 form 就变空了，以便接下来我们自己设置。

二，放置 widgets 和 buttons 等

在 Qt Designer 窗口左边有一个 widget box，里面有很多图标代表了一些 widget，我们在 Display Widget 里面找到 Label 控件，然后把它拖到我们的窗体 form 中，放到左上角，选中它，然后我们在 Qt Designer 窗口右边的属性编辑器中找到“Text”属性，然后把它的值改为“Find &what”，如下图：

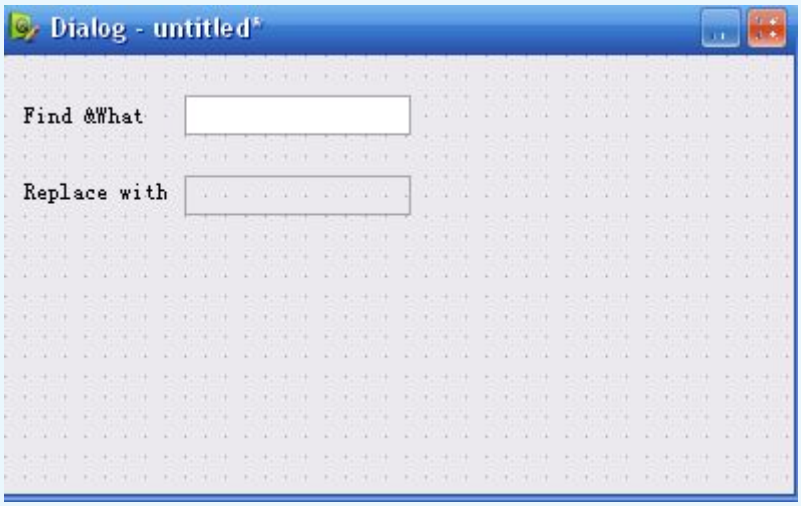


然后我们再从 widget box 中的 input widgets 中找到 Line Edit ，把它拖到对话框中 Label 的右边，对齐。然后在属性编辑器找到 ObjectName 属性，把值改为 findLineEdit。为什么上面的 LableName 我们不改呢？因为我们不会在程序中用到它，这里改了 LineEdit 的 Name 是因为在程序中会用到这个值。



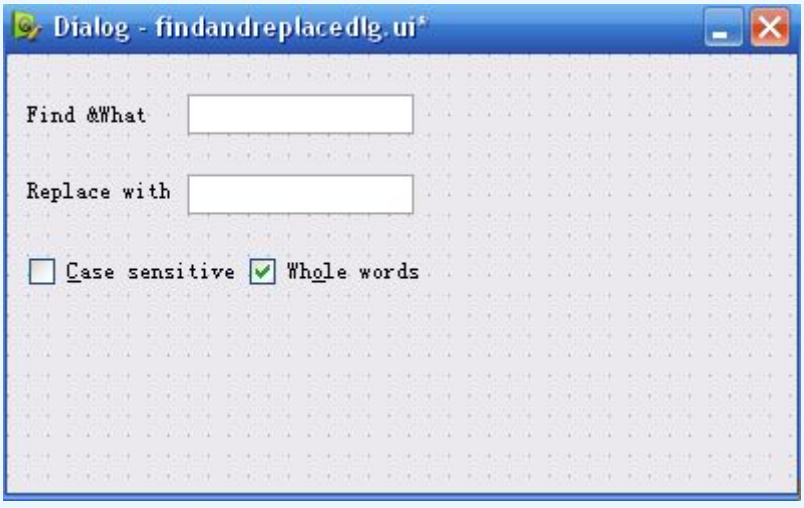
然后，在拖一个 Label 和一个 Line Edit 分别放到上面的两个 widget 下面，其中 Label 的 text 属性改为“Replace with”，而 Line Edit 的 ObjectName 改为

replaceLineEdit。结果应该如下图所示：

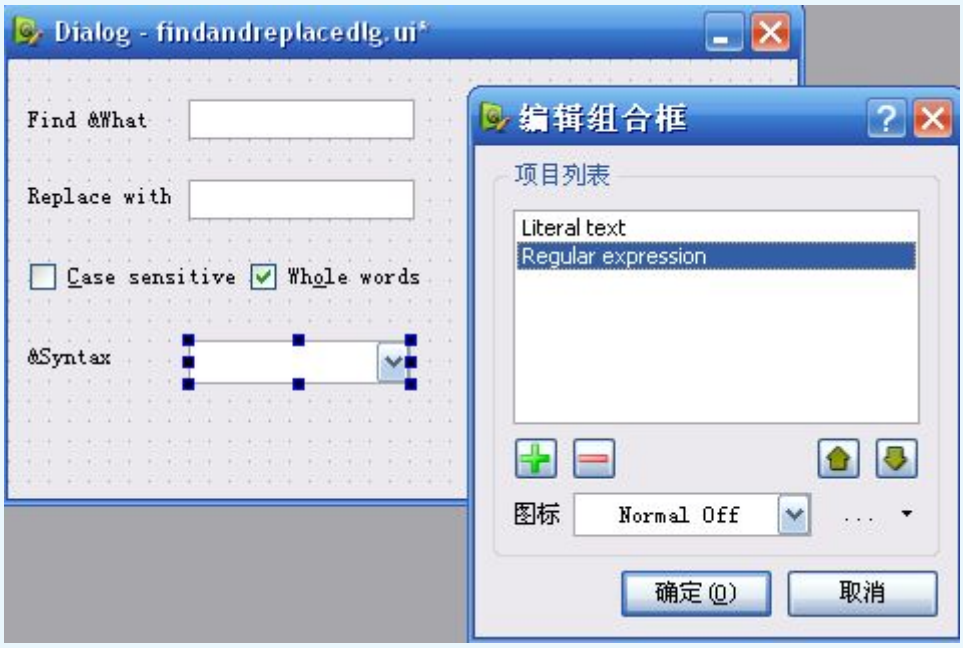


先把上面的保存为：findandreplacedlg.ui

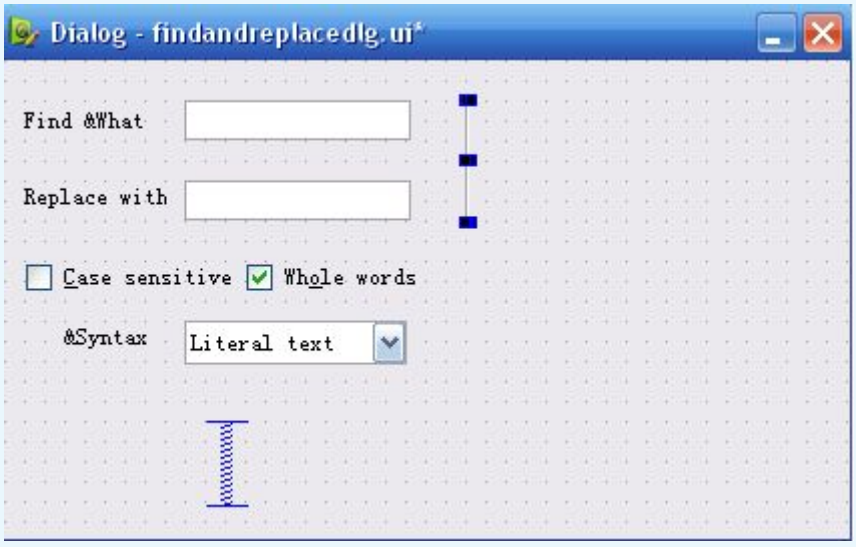
下面，我们在上面的基础上增加两个 checkboxs，从 widget box 中的 Buttons 中找到 Check Box，然后拖两个到对话框中，放到 label 和 line edit 的下面。把第一个 check box 的 ObjectName 改为 caseCheckBox，把它的 Text 属性值改为&Case sensitive;把第二个 check box 的 ObjectName 改为 wholCheckBox，Text 属性改为 Wh&ole words，并且把“checked”状态改为“true”。此时对话框如下所示：



下面，在上面基础上添加一个 Label 和一个 ComboBox。其中的 Label 放到第一个 check box 的下面,它的 Text 属性改为&Syntax。然后拖一个 ComboBox 放到这个 Label 的右边，它的 Objectname 设为“syntax ComboBox”。然后我们要给这个 ComboBox 添加两个 Items。方法是：选中这个 ComboBox，点击右键，在弹出的菜单中选第一项 Edit Items,然后在弹出的对话框中点击图标“+”，添加一项 Item，并把内容改为“Literal text”，然后再同样的方法添加一项“Regular expression_r”。如下图所示：



下面我们就要在 dialog 的右边创建 button 了，在创建之前。我们还需要向窗口添加两个东西。首先添加一个 Vertical Spacer(在 widget box 中的 Spacers group 中)，它的作用是当 dialog 的大小改变时，窗口里的 widgets 不会布局发生散化或者变乱。第二个需要添加的是 Vertical Line(在 widget box 中的 Display widgets 中)，把它放到两个 LineEdit 的右边，用于分割这些 widgets 和我们即将添加的 buttons。如下图所示：



下面我们开始创建 button。从 Button group 中拖一个 Push Button 到窗口中，放到右上位置，把它的 ObjectName 改为“findButton”，它的 text 改为“&Find”。

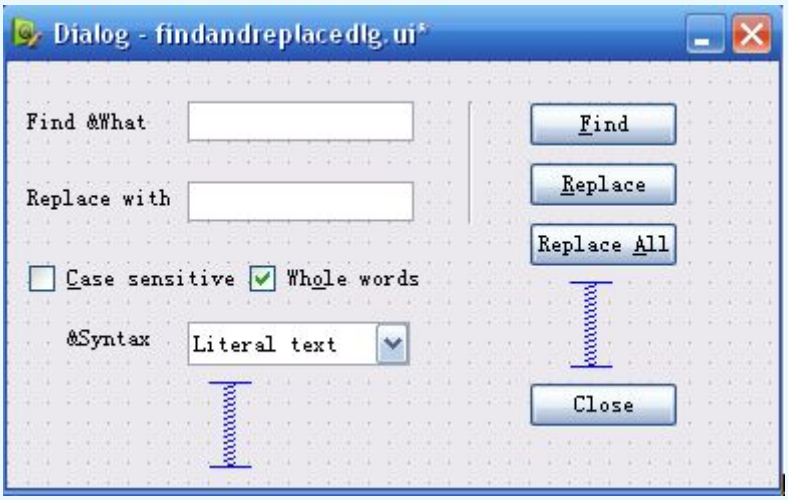
再拖个一个 push button 放到这个 find 的下面，ObjectName 改为

“replaceButton”，text 改为“&Replace”。

创建第三个 button 在 replace button 的下面，ObjectName 改为“replaceAllButton”，text 改为“Replace &All”。然后拖一个 Vertical spacer 放到下面。

最后我们创建第四个 button，ObjectName 改为“closeButton”，text 改为“Close”。

现在所有的 widgets 和 button 我们都创建完了，如下图所示：



参考资料《Rapid GUI Programing with PyQt》chapter 7

PyQt 学习笔记(8)——Qt Designer(2)

三，完成窗口的布局 Layout

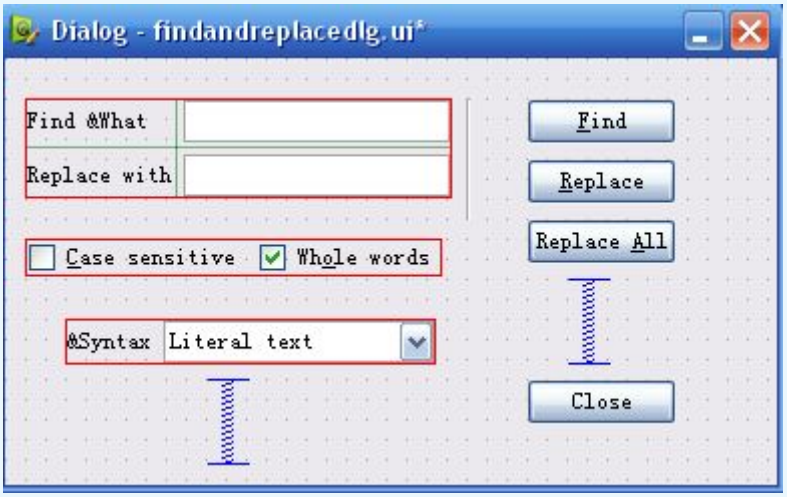
关于窗口的布局，不同人可能有不同的喜好，下面简单的演示一下。

首先，我们把 Find what 和 replace With 的两个 Label 和两个 Line Edit 布局到一起，我们按住 shift 键，鼠标选中这四个 widgets。然后点击菜单 Form->Lay Out in a Grid 项（或者点击 toolbar button），四个 widgets 会用红线圈起来，当然这些红线在程序运行时是不会显示的。

然后选择两个 CheckBox，点击菜单 Form->Lay Out Horizontally，水平布局。

同样我们也把下面的一个 Label 和 ComboBox 设为水平布局。

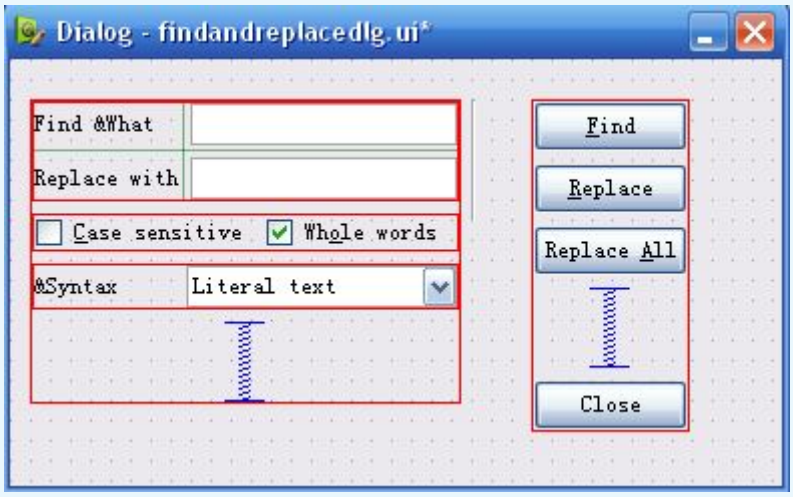
如上所示，现在已经有三个 layout 了，一个格型布局，两个水平布局。



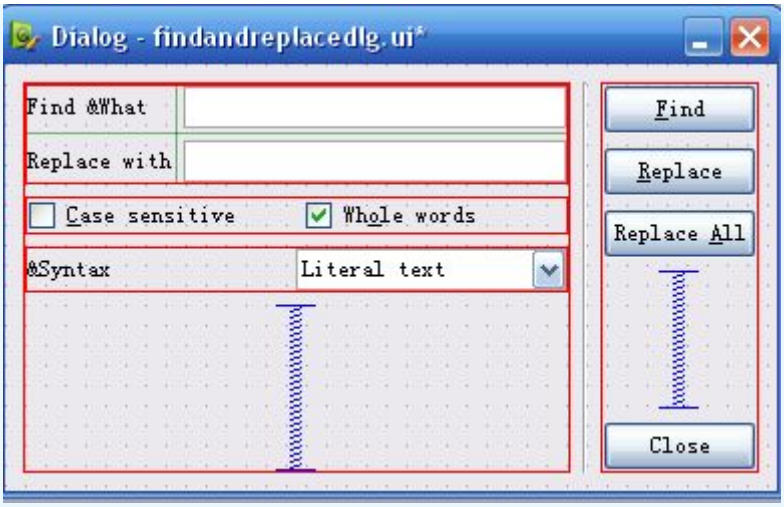
下面我们把这三个布局放到一个布局里面，我们通过鼠标拖拽来选定三个 layout，只要覆盖到 layout 就行，注意不要接触来 Vertical Line，但是要接触到 Vertical spacer，选中以后，点击菜单 Form->Lay Out Vertically。

然后用同样的方法，拖拽选中右边的四个 button 和一个 Vertical spacer，选中以后，点击菜单 Form->Lay Out Vertically。

最终我们有了两个垂直的 layouts，和中间一个 Vertical Line，如下图所示：



最后，我们不要选中任何东西，点击菜单 Form->Lay Out Horizontally 完成最终 Form 的布局，最终结果如下图所示：

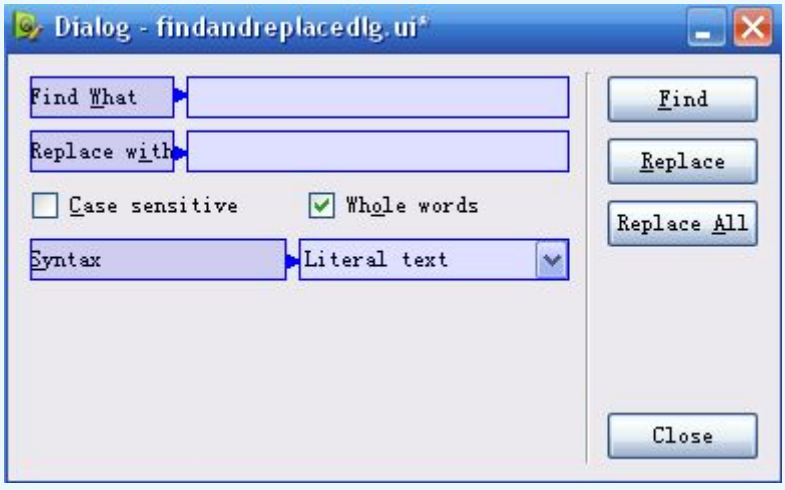


可以看到，比上面有了些变化，窗口布局的非常合理和漂亮了。

可以通过预览来看一看。

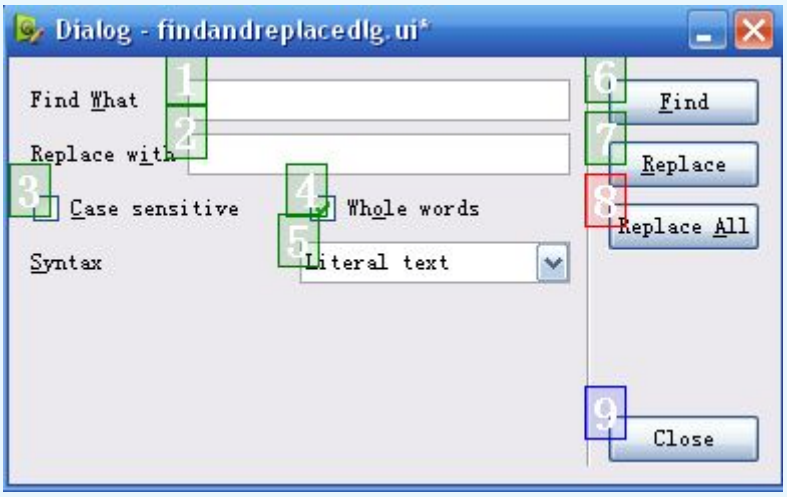
四，设计工作的一些收尾

我们发现 Label 的 buddy 我们还没有弄好,所以在 label 中会显示我们输入的”&”符号，下面我们编辑一下。点击菜单 Edit->Edit Buddies 转到 buddy 模式。然后我们为 label 选择伙伴 buddy，选定一个 label，然后把它拖到想绑定的 buddy 上，比如选定 Find what Label，然后把它拖到 Line Edit 上。类似的我们可以绑定三对 buddy，如下图所示：

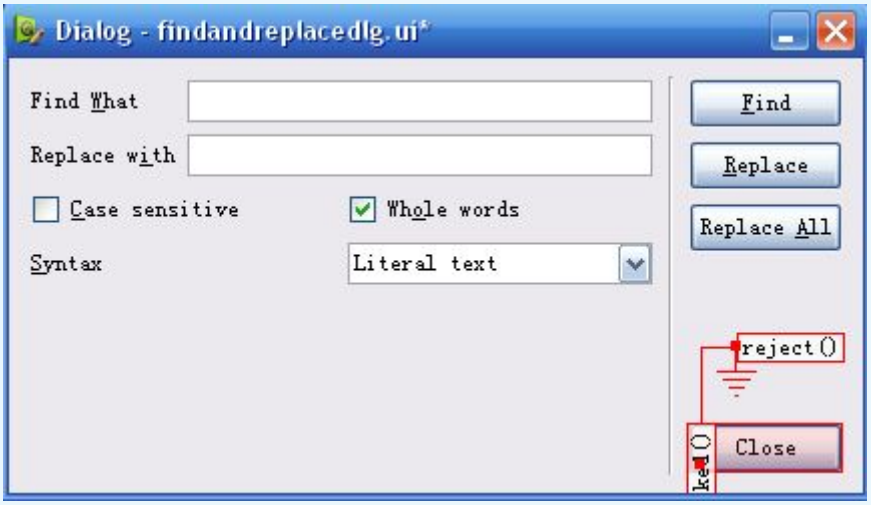


按 F3可以离开 buddy 模式，进入到窗口编辑模式。

同样，我们可以设置 Tab 的顺寻，通过点击 Edit->Edit Tab Order，进入到编辑 tab order 的模式，如下图。然后通过点击来改变顺序，同样按 F3可以退出编辑模式.



关于 button 方法的绑定，其中 Find，Replace，Replace All 三个，需要绑定用户自己写的方法，而 colse 按钮可以绑定到对话框的 reject()，下面看看如何绑定：点击菜单 Edit->Edit Signal/Slots，然后点击 close button 拖一下到 form 上，放开鼠标左键，会弹出一个 Configure connection 的对话框，从左边列表框中点击 clicked()，从右边点击 reject ()，然后点击 OK，结束，如下图。



按 F3离开模式。

最后可以给这个对话框命名，ObjectName 设为：FindAndReplaceDlg，把 windowTitle 设为“Find and Replace”。其中的 ObjectName 是我们在程序中要用到这个对话框时，他的类名，或者 import 的东西。

保存为 findandreplacedlg.ui 文件。

五，转化 ui 文件为 py 文件

打开 cmd 命令行，在 findandreplacedlg.ui 文件所在的目录运行下面的命令：

```
pyuic4 -o ui_findandreplacedlg.py findandreplacedlg.ui
```

就会在同样的目录生成一个名为 ui_findandreplacedlg.py 的文件，然后我们就可在别的工程中 import 这个文件来产生我们想要的对话框了。

参考资料《Rapid GUI Programing with PyQt》chapter 7

PyQt 学习笔记(9)——Qt Designer(3)

六，如何在工程中使用

如何使用上面我们产生的 py 文件呢？首先我们建立一个 findandreplacedlg.py。我们将在这个文件中使用。

首先是 import

```
import re
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import ui_findandreplacedlg
MAC = "qt_mac_set_native_menubar" in dir()
```

这里的 MAC 是个布尔型的，判断我们的操作系统是否是苹果系统，如果是，MAC=TRUE。

然后我们产生一个对话框类，注意它的两个父类：

```
class FindAndReplaceDlg(QDialog,
        ui_findandreplacedlg.Ui_FindAndReplaceDlg):
    def __init__(self, text, parent=None):
        super(FindAndReplaceDlg, self).__init__(parent)
        self.__text = unicode(text)
        self.__index = 0
        self.setupUi(self)
        if not MAC:
            self.findButton.setFocusPolicy(Qt.NoFocus)
            self.replaceButton.setFocusPolicy(Qt.NoFocus)
            self.replaceAllButton.setFocusPolicy(Qt.NoFocus)
            self.closeButton.setFocusPolicy(Qt.NoFocus)
        self.updateUi()
```

我们从 Qdialod 和 ui_findandreplacedlg.Ui_FindAndReplaceDlg 继承生成一个子类。在初始化函数__init__中，text 参数是我们需要给这个窗口的参数，即我们要 findandreplace 的内容，他可以是一个文件，一段字符串等等。

Super 和以前的一样，对话框初始化，注意是继承于 Qdialog，不是我们在 Designer 中设计的对话框。

self.setupUi(self) 这一句的作用才是把我们在 Designer 中设计的界面搬到我们这个对话框中，包括它的 widgets，tab order 等等。这个 setupUi 是在 ui_findandreplacedlg.py 中由 pyuic4生成的。

更重要的是，这个 setupUi()方法会调用 QtCore.QmetaObject.connectSlotsByName()方法，这个方法的作用是它会自动创建 signal-slots connection ，这种 connection 是基于我们子类里的方法，和窗口的 widgets 之间的，只要我们定义的方法，它的名字是 on_widgetName_signalName 的这种格式，就会自动把这个 widgets 和这个 signaName connected。

举个例子，在我们的 form 中，我们曾把 Find what 这个 Label 右边的 Line Edit 这个 widget 的 ObjectName 命名为 findLineEdit ，跟这个 widget 相关的信号，就是这个 Line Edit 被编辑了，就会 emit 一个信号 textEdited（Qstring），所以如果我们想绑定这个 widget 和这个 signal，就不用调用 connected（）方法了，只要我们把方法的名字命名为 on_findLineEdit_textEdited()就可以了，connect 的工作就交给 setupUi 去完成了。

后面的四句 setFocusPolicy 是为了方便键盘用户（windows 和 Linux）的，就是使 Tab 键只能在可编辑 widgets 之间切换，对于 MAC 系统不做执行。

最后的 setupUi（）方法，是为了在 findLineEdit 没有输入内容的时候，让几个功能按钮不可用，当当输入了以后变为可用。

下面是我们定义一些方法：

```
    @pyqtSignature("QString")
    #确保正确的 connect，括号里的参数为 signal 的参数
    def on_findLineEdit_textEdited(self, text):
        self.__index = 0
        self.updateUi()
#此函数就是发现 LineEdit 有内容输入，buttons 变为可用
    def makeRegex(self):
        #利用正则表达式来查找内容
        findText = unicode(self.findLineEdit.text())
        if unicode(self.syntaxComboBox.currentText()) ==
"Literal":
            findText = re.escape(findText)
            flags = re.MULTILINE|re.DOTALL|re.UNICODE
        if not self.caseCheckBox.isChecked():
            flags |= re.IGNORECASE
        if self.wholeCheckBox.isChecked():
            findText = r"\b%s\b" % findText
```

```
            return re.compile(findText, flags)


        @pyqtSignature("")
        def on_findButton_clicked(self):
            regex = self.makeRegex()
            match = regex.search(self.__text, self.__index)
            if match is not None:
                self.__index = match.end()
                self.emit(SIGNAL("found"), match.start())
            else:
                self.emit(SIGNAL("notfound")
        @pyqtSignature("")
        def on_replaceButton_clicked(self):
            regex = self.makeRegex()
            self.__text =
regex.sub(unicode(self.replaceLineEdit.text()),
                                  self.__text, 1)


        @pyqtSignature("")
        def on_replaceAllButton_clicked(self):
            regex = self.makeRegex()
            self.__text =
regex.sub(unicode(self.replaceLineEdit.text()),
                                  self.__text)
        def updateUi(self):
            #判断 LineEdit 是否为空，从而决定 buttons 是否可用
            enable = not self.findLineEdit.text().isEmpty()
            self.findButton.setEnabled(enable)
            self.replaceButton.setEnabled(enable)
            self.replaceAllButton.setEnabled(enable)

        def text(self):
           #返回修改后的结果
            return self.__text
```

可以用下面的程序测试一下结果：

```
if __name__ == "__main__":
    import sys

    text = """US experience shows that, unlike traditional patents,
software patents do not encourage innovation and R&D, quite the
contrary. In particular they hurt small and medium-sized
enterprises
and generally newcomers in the market. They will just weaken the
market
and increase spending on patents and litigation, at the expense
of
technological innovation and research. Especially dangerous are
attempts to abuse the patent system by preventing interoperability
as a
means of avoiding competition with technological ability.
--- Extract quoted from Linus Torvalds and Alan Cox's letter
to the President of the European Parliament
http://www.effi.org/patentit/patents_torvalds_cox.html"""

    def found(where):
        print "Found at %d" % where

    def nomore():
        print "No more found"

    app = QApplication(sys.argv)
    form = FindAndReplaceDlg(text)
    form.connect(form, SIGNAL("found"), found)
    form.connect(form, SIGNAL("notfound"), nomore)
    form.show()
    app.exec_()
    print form.text()
```

参考资料《Rapid GUI Programing with PyQt》chapter 7

PyQt 学习笔记(10)——Events and Signals

事件和信号在应用程序中是非常重要的，在 PyQt4中，与 wxpython 有许多不同，下面一起看看。

一，Events

事件是任何 GUI 程序中最重要的部分。事件由用户或者系统生成，当我们调用应用程序的 exec_()方法时，应用程序进入了它的循环，主循环获取事件并将它们送给特定的对象处理。PyQt 引入了一种独一无二的信号和插槽来处理事件。

二，信号与插槽(signal and slots)

当用户点击按钮，拖动滑块等操作时会产生信号，同时环境也可以产生信号，比如时钟的信号。插槽是一种针对信号进行处理的方法。Python 中的插槽可以是任何的 python 可调用部分。

```
# !/usr/bin/python

import sys

from PyQt4 import QtGui, QtCore

class SigSlot(QtGui.QWidget):

    def __init__(self, parent=None):

        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('signal & slot')

        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.resize(250, 150)

app = QtGui.QApplication(sys.argv)

qd = SigSlot()

qd.show()

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

上面我们定义了一个 LCD number 和一个 slider，通过拖拽 slider 来改变 LCD 的数值。

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

这一句就是将 valueChanged()信号与 lcd 数字的 display()插槽相关联。连接方法有四个参数，sender 是发送信号的对象，signal 是它产生的信号，receiver 是信号接收的对象，最后 slot 是相应信号的方法。

三，事件处理重载

PyQt 中的事件处理主要是通过对事件处理者的重载来进行的。看下面一个例子：

```
# !/usr/bin/python
import sys
from PyQt4 import QtGui, QtCore
class Escape(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle('escape')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()'))
    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_())
```

重载了 KeyPressEvent()的方法，按下 ESC 键，程序关闭。

四，发送信号

通过 QtCore.Qobject 创建对象可以发送信号。如果我们点击按钮，就会产生一个 clicked()的信号，看例子：

```
# !/usr/bin/python
import sys
from PyQt4 import QtGui, QtCore
class Escape(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle('escape')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()'))
```

```
    def keyPressEvent(self, event):
        self.emit(QtCore.SIGNAL('closeEmitApp()'))
app = QtGui.QApplication(sys.argv)
qb = Escape()
qb.show()
sys.exit(app.exec_())
```

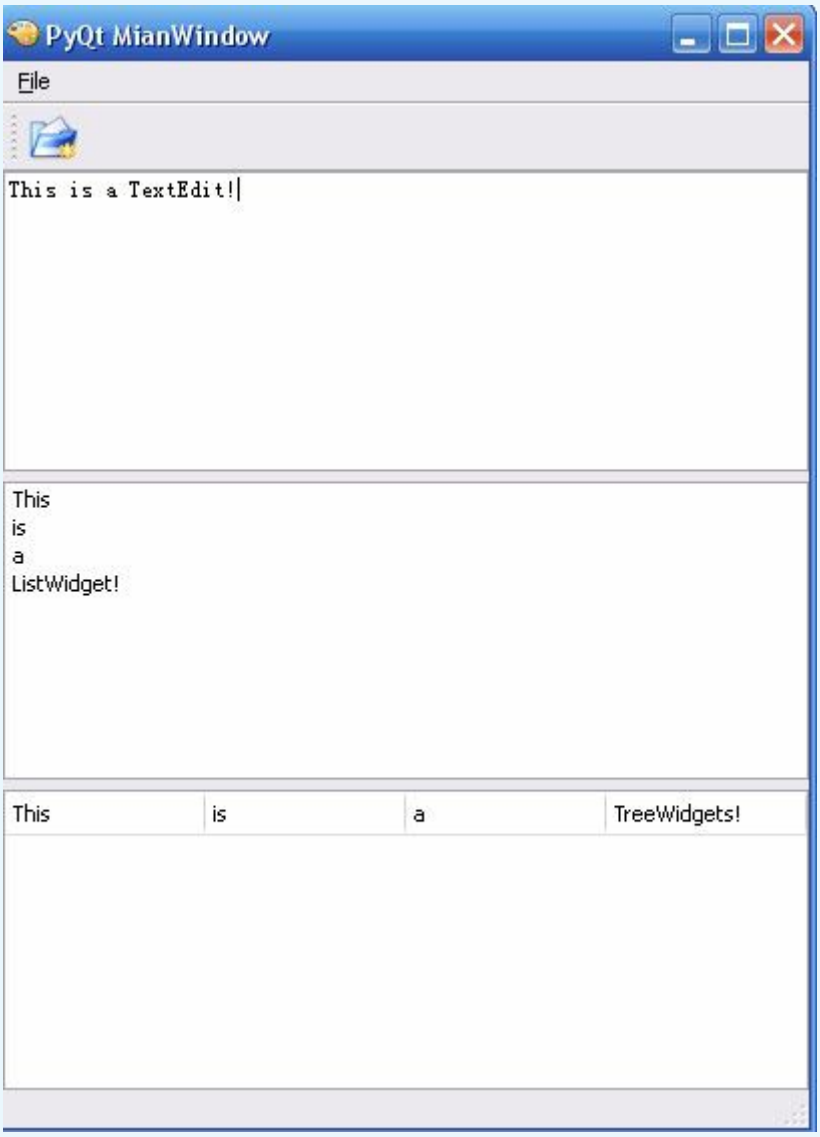
创建了一个叫 closeEmitApp() 的新信号。这个信号是在鼠标按下时产生的。

通过 emit() 来产生一个信号。

然后将我们手工产生的 closeEmitApp() 信号与 close() 插槽连接。

PyQt 学习笔记(11)——QSplitter 分割窗口

QSplitter 是用来分割窗口的，比如下面的窗口：



QmainWindow 上面有三个 widget，一个 TextEdit，一个 ListWidget，一个 TreeWidget，要让他们分占窗口，并且鼠标放到两个 widget 的边界，还可以拖拽来改变 widget 的大小。

下面我们看看如何实现：

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

        self.textedit = QTextEdit()
        self.textedit.setText("This is a TextEdit!")

        self.listwidget = QListWidget()
        self.listwidget.addItem("This\nis\na\nListWidget!")

        self.treewidget = QTreeWidget()

        self.treewidget.setHeaderLabels(['This','is','a','TreeWidge
        ts!'])

        splitter = QSplitter(self)
        splitter.addWidget(self.textedit)
        splitter.addWidget(self.listwidget)
        splitter.addWidget(self.treewidget)
        splitter.setOrientation(Qt.Vertical)
        self.setCentralWidget(splitter)
```

我们创建了三个 widget 以后，我们通过创建一个 Qsplitter(parent)，注意它的参数此处是 self，也就是我们的 MainWindow。

然后把三个 widget 通过 addwidget 方法加到 Qsplitter 上，如果我们不调用 setOrientation(Qt.Vertical)的话，三个控件默认是水平摆放的，这里我们调用了，变为了垂直摆放，最后把 splitter 放到了 MainWindow 上。

另外 Qsplitter 还有别的很多 methods。

Qsplitter.insertWidget(self,int index,Qwidget widget)在 index 序号处插入一个 widget。

Qsplitter.indexOf(self,Qwidget widget)可以得到一个 widget 的 index。

当然如果要隐藏某一个 widget，可以调用 hide（）方法，要显示可以调用 show（）方法。

Qsplitter.count()返回一个 splitter 里面的 widget 的数目。

QSplitter.restoreState()和 QSplitter.saveState()用来保存和恢复状态，一般和 Qsetting 一起使用。

PyQt 学习笔记(12)——QSetting

参考资料：Qt documentation

用户对应用程序经常有这样的要求：要求它能记住它的 settings，比如窗口大小,位置，一些别的设置，还有一个经常用的，就是 recent files，等等这些都可以通过 Qsettings 来实现。

我们知道，这些 settings 一般都是存在系统里的，比如 windows 一般都写在系统注册表或者写 INI 文件，mac 系统一般都在 XML 文件里，那么按照一般的标准来说，许多应用程序是用 INI 文件来实现的。而 Qsettings 就是提供了一种方便的方法来存储和恢复应用程序的 settings。

QSettings 的 API 是基于 Qvariant，Qvariant 是一种数据类型的集合，它包含了大部分通常的 Qt 数据类型，比如 QString，QRec，QImage，等等。

当我们创建一个 Qsettings 的对象时，我们需要传递给它两个参数，第一个是你公司或者组织的名称，第二个事你的应用程序的名称。比如：

Settings = Qsettings(“MySoft”,”QtPad”)

公司名称：MySoft，程序名称：QtPad

假如我们在应用程序中多次要用到 Qsettings，为了简单其间，我们可以在主程序中先如下声明。

```
QtCore.QCoreApplication.setOrganizationName("MySoft")
QtCore.QCoreApplication.setOrganizationDomain("mysoft.com")
QtCore.QCoreApplication.setApplicationName("QtPad")
```

当然前提是已经 from PyQt4 import QtCore

然后在应用程序的任何地方想要声明一个 Qsettings 类型的变量，便不需要书写两个参数了，直接用 settings = Qsettings 即可。

那么如何用它来保持应用程序的 settings 信息呢？我们以字典数据类型与之类比，它也有 key，以及对应的 value。比如下面例子：

```
settings = Qsettings(“MySoft”,”QtPad”)
Mainwindow = QmainWindow()
settings.setValue(“pos”,QVariant(Mainwindow.pos())
settings.setValue(“size”,QVariant(Mainwindow.size())
```

上面两句就是把当前窗口的位置，和大小两个信息记录到了 settings 中，其中的 key 就是”pos”和”size”两个 Qstring 类型，而它所对应的值就是 QVariant 类型的。当然如果我们要写的 key 已在 settings 中存在的话，则会覆盖原来的值，写入新值。

如何读取 Qsettings 里的内容呢？如下：

```
Pos = settngs.value(“pos”).toPoint()
Size = settings.value(“size”).toSize()
```

当然如果 key 所对应的 value 是 int 型的，也可 toInt()，如果没有我们要找的 key，则会返回一个 null QVariant 如果用 toInt 的话会得到0。

那么实际应用中我们一般会如下：

```
pos= settings.value("pos", QVariant(QPoint(200, 200))).toPoint()
size= settings.value("size", QVariant(QSize(400, 400))).toSize()
self.resize(size)
self.move(pos)
```

意思是，如果 settings 里有以前存下的(用 setValue 设置的)pos 和 size 的值，则读取，如果没有，不会返回 null，而会使用我们给它的起始值——default value——即应用程序第一次运行时的情况。

注意：因为 QVariant 是不会提供所有数据类型的转化的，比如有 toInt(),toPoint(),toSize(),但是却没有对 Qcolor，Qimage 和 Qpixmap 等数据类型的转化，此时我们可以用 QVariant.value（），具体参看 QVariant 模块说明。

下面看看如何在应用程序中使用：

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        ...
        self.readSettings()
        ...
    def readSettings(self):
        settings = Qsettings(“MySoft”,” QtPad”)
        pos=settings.value("pos",QVariant(QPoint(200,200))).toPoint()
        size=settings.value("size",QVariant(QSize(400,400))).toSize()
        self.resize(size)
        self.move(pos)
    def writeSettings(self):
```

```
        settings = Qsettings(“MySoft”,” QtPad”)
        settings.setValue("pos", QVariant(self.pos()))
        settings.setValue("size", QVariant(self.size()))
    def closeEvent(self,event):
        if self.maybeSave():
            self.writeSettings()
            event.accept()
        else:
            event.ignore()
```

上面是一般应用程序的应用方法。

下面再看一些 Qsettings 里常用的 metho：

Qsettings.annKeys(self) 返回所有的 key，以 list 的形式

Qsettings.applicationName(self) 返回应用程序名称

Qsettings.clear(self) 清楚此 settings 里的内容

Bool Qsettings.contains(self,key) 返回真，如果存在名为 key 的 key

Qsettings.remove(self, keyname) 清楚 key 及其所对应的 value

Qsetting.fileName() 返回写入注册表地址，或者 INI 文件路径

等等，请参看帮助文档。

我们可以探索一下，这些 settings 在应用程序关闭以后到底存到了什么地方呢？

我们可以在上面的程序中的 writeSettings 中，后面加一句话：

Print Settings.fileName()

这个在 windows 下，默认 Qsettings 会打印出这个程序的系统注册表所在地：

这个结果是：\HKEY_CURRENT_USER\Software\MySoft\QtPad

如下图：

由此我们可以看出，这个 writesettings 其实就是个写注册表的过程。

当然，我们也可以不写注册表，我们写 ini 文件：

```
settings = QSettings("./QtPad.ini", QSettings.IniFormat)
settings.setValue("pos", QVariant(self.pos()))
```

```
settings.setValue("size", QVariant(self.size()))
```

就会在当前文件夹下产生一个 QtPad.ini 文件，打开后文件内容为：

```
[General]
pos=@Point(200 200)
size=@Size(400 400)
```

更多关于 Qsettings 内容请参考帮助文档。

PyQt 学习笔记(13)——QFile

QFile 提供了一个读写文件的接口，QFile 经常和 QTextStream 以及 QDataStream 一起使用。

使用 QFile 时候，一般要传递一个 file name 参数，但也可以通过 setFileName() 方法在 QFile 实例化以后在设置。注意文件路径要用”/”而不要要”\”来间隔。

exists（）方法用来 check 文件是否存在，remove()用来 remove a file。

用 open（）打开文件，用 close（）关闭文件，读写文件一般使用 QDataStream() 和 QTextStream()。当然我们也可以使用 python 标准内置方法 read(),readLine(),readAll(),write()等。

文件的大小可以通过 size（）方法得到，文件的位置可以用 pos()方法，seek()移动到文件的特定位置，atEnd（）返回真即到了文件结尾。

下面看一个例子：

```
def loadFile(self, fileName):
    file = QFile(fileName)
    if not file.open(QFile.ReadOnly | QFile.Text):
        QMessageBox.warning(self,
        self.tr("Application"),self.tr("Cannot read file
        %1:\n%2.").arg(fileName).arg(file.errorString()))
        return
    inf = QTextStream(file)
```

if not 一句当打开文件出现问题(比如文件不存在)时会弹出 waring 对话框并返回。其中 open 里面的两个参数，ReadOnly 和 Text 分别代表只读，和转化换行符为”\n”。

QTextStream 是读取文件全部内容于 inf，如果想打印出里面的内容，我们可以通过：Inf.readAll(),用 cont = inf.readAll()把 text 的内容以 Qstring 的类型读到 cont 中，用 print cont 即可打印文件的内容了。

关于 QFile 的 methods 情参看 documentation

下面主要看看如何用 QDataStream 读取文件。

什么是 data stream 呢？它是一个二进制流信息，并且这样的文件具有平台无关性，不管是 linux，windows，而且与 CPU 的字节序无关，在 windows 下写的 data stream 它可以在别的其他系统中读取。

```
from PyQt4.QtCore import *
myFile = QFile('./file.dat')
if not myFile.open(QFile.WriteOnly | QFile.Text):
    print "something is wrong!"
out = QDataStream(myFile)
out<<QString("test 1 2 3")
myFile.close()
```

我们就在当前目录下建立了一个二进制文件 file.dat ，用 UltraEdit 打开我们会发现，它是8bit 的 data 转化成16bit 的 Unicode Qstring 的形式存到硬盘上，当然我们是假设我们的系统默认的是8bit 的编码。所以这里的一个”t”是用了十六进制 0x7400两个字节来表示的。

下面我们看一下如何读取上面写的 file.dat 文件

```
readfile = QFile("./file.dat")
myFile.open(QFile.ReadOnly)
cont = QDataStream(myFile)
qst = QString()
cont>>qst
print qst
```

结果为打印出”test 1 2 3”

读取二进制文件：看下面的例子

```
>>> from PyQt4.QtCore import *
>>> pcapfile = QFile('C:/Python25/code/pcap/test.pcap')
>>> pcapfile.open(QFile.ReadOnly)
True
>>> out = QDataStream(pcapfile)
>>> QDataStream.readRawData(out,4)
'\xd4\xc3\xb2\xa1'
>>> QDataStream.readRawData(out,4)
'\x02\x00\x04\x00'
>>> type(QDataStream.readRawData(out,4))
<type 'str'>
>>>
```

读取 pcap 文件，以二进制的方式，可以读取任意字节出来，注意读取的是 str 格式的。

PyQt 学习笔记(14)——布局管理

转自：

http://www.blogjava.net/glorywine/archive/2008/07/30/217842.html

布局是 GUI 程序开发中非常重要的一个环节，而布局管理就是要规划如何在窗口放置需要的部件，PyQt4中有两种方法来完成布局任务，一个是绝对位置(absolute positioning)，另一个就是使用布局类(layout class)。

一，绝对位置(absolute positioning)

这种方法要求程序员在程序中指定每一个部件的坐标位置和大小，注意事项：

1指定了坐标和大小的部件不能够随着窗口大小的变化而变化；

2程序在不同的操作系统平台上也许会有变化；

3改变字体可能会引起布局的混乱；

4如果需要改变当前的布局，就需要重新编码，这意味着非常大的工作量。

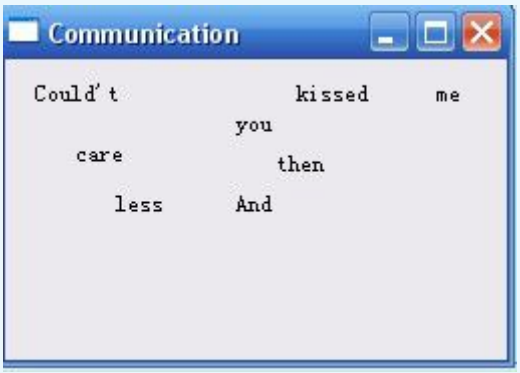
下面看一个例子：

```
# !/usr/bin/python
import sys
from PyQt4 import QtGui
class Absolute(QtGui.QWidget):
    def __init__(self,parent=None):
        QtGui.QWidget.__init__(self,parent)
        self.setWindowTitle('Communication')
        label = QtGui.QLabel('Could\'t',self)
        label.move(15,10)
        label = QtGui.QLabel('care',self)
        label.move(35,40)
        label = QtGui.QLabel('less',self)
        label.move(55,65)
        label = QtGui.QLabel('And', self)
        label.move(115, 65)
        label = QtGui.QLabel('then', self)
        label.move(135, 45)
        label = QtGui.QLabel('you', self)
        label.move(115, 25)
```

```
        label = QtGui.QLabel('kissed', self)
        label.move(145, 10)
        label = QtGui.QLabel('me', self)
        label.move(215, 10)
        self.resize(250, 150)

app = QtGui.QApplication(sys.argv)
qb = Absolute()
qb.show()
sys.exit(app.exec_())
```

在这里就是简单的调用 move() 方法来指定部件的放置坐标，坐标的顶点就是窗口的左上角，x 由左向右不断增大，y 由上到下不断增大。

结果如下图：



二，使用布局类

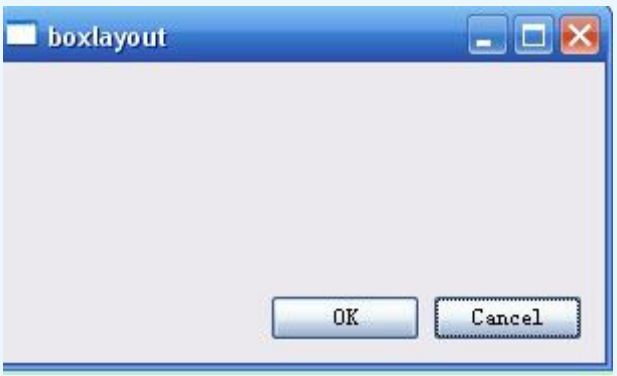
首先看 Box Layout

最基本的布局类是 QHBoxLayout 和 VHBoxLayout，它们将部件线性水平或垂直排列。这里假设我们要讲两个按钮放在窗口的右下角，要实现这样的布局，使用一个 QHBoxLayout 和一个 QVBoxLayout，而其他的空间，通过添加 stretch factor 来实现。

```
# !/usr/bin/python

import sys
from PyQt4 import QtGui

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

```
        self.setWindowTitle('boxlayout')

        ok = QtGui.QPushButton('OK')
        cancel = QtGui.QPushButton('Cancel')

        hbox = QtGui.QHBoxLayout()
        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)
qb = BoxLayout()
qb.show()
sys.exit(app.exec_())
```

结果如下：



QGridLayout

最常用的布局类是 QGridLayout，他将窗口分为不同的行和列

```
# !/usr/bin/python
```

```
import sys
from PyQt4 import QtGui

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

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

结果如下所示：

当然一个部件也可以占多行多列，如下面的例子：

```
# !/usr/bin/python

import sys
from PyQt4 import QtGui

class GridLayout2(QtGui.QWidget):
    def __init__(self,parent=None):
        QtGui.QWidget.__init__(self,parent)

        self.setWindowTitle('grid layout2')

        title = QtGui.QLabel('Tltle')
        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.resize(350,300)

app = QtGui.QApplication(sys.argv)
qb = GridLayout2()
qb.show()
sys.exit(app.exec_())
```

结果如下图所示：

其中 `grid.setSpacing(10)` 是设定部件之间的距离是10个像素。

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

这个部件放到了第三行，第一列， 并且后面的5和1，分别代表所占的行数和列数，就是表示放到第三行，大小占5行。

有了上面的讲解，布局管理基本上就能实现合理。

PyQt 学习笔记(15)——QFileDialog

参考资料：Qt documentation

QFileDialog 类提供了一个供用户选择文件或者目录的对话框。

创建一个 QfileDialog 最简单的方法是调用静态函数，就是直接应用平台提供的文件对话框(windows,linux 等)。

一，调用静态函数法：

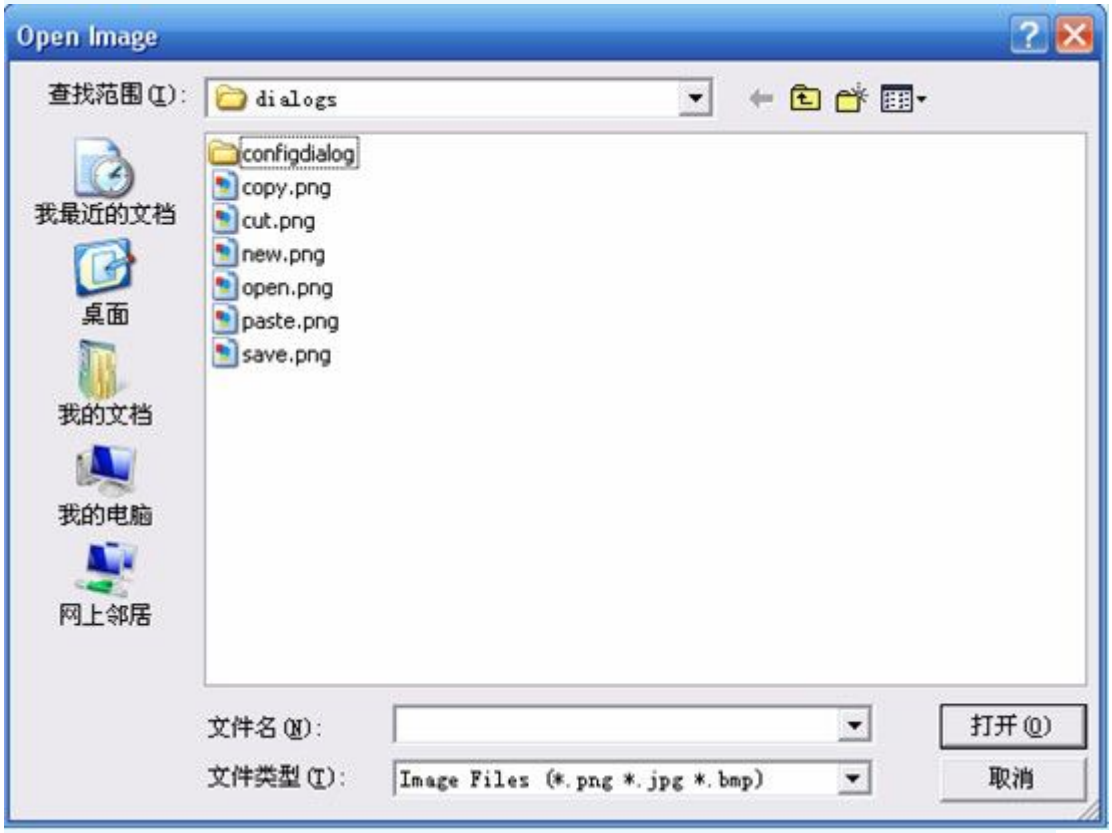
先看一下它的初始化函数

```
__init__(self,Qwidget parent=None,Qstring caption=Qstring(),Qstring directory =Qstring(),Qstring filter=Qstring())
```

比如说下面的方法 openFile 是主窗口类里的 method：

```
def openFile(self):
    fileName = QFileDialog.getOpenFileName(self,self.tr(“Open Image”),Qstring(),self.tr(“Image Files(*.png *.jpg *.bmp)”))
```

调用后的结果如下：



以上是在 windows xp 下的样子。

可以看到参数里有三个 QStrng 类型的参数，第一个代表弹出对话框的标题；第二是默认的路径，如果空，则默认为当前路径；第三个是代表了过滤器，如果要有多个过滤器的话，可以用”::”分割，如："Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"

得到的 fileName 就是我们选择的文件和它的路径。

当然 QFile 的静态函数有很多，如下面所列，可以根据需要选择：

Static Methods

QString **getExistingDirectory** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), Options *options* = QFileDialog.ShowDirsOnly)
QString **getOpenFileName** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)
QString **getOpenFileName** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), QString *selectedFilter* = None, Options *options* = 0)
tuple **getOpenFileNameAndFilter** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)
QStringList **getOpenFileNames** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)
QStringList **getOpenFileNames** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), QString *selectedFilter* = None, Options *options* = 0)
tuple **getOpenFileNamesAndFilter** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)
QString **getSaveFileName** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)
QString **getSaveFileName** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), QString *selectedFilter* = None, Options *options* = 0)
tuple **getSaveFileNameAndFilter** (QWidget *parent* = None, QString *caption* = QString(), QString *dir* = QString(), QString *filter* = QString(), Options *options* = 0)

通过实验我们发现，上面的对话框基本为模态对话框。

另存为对话框也可以通过静态方法来实现：

```
def saveAsFile(self):
```

```
fileName = QFileDialog.getSaveFileName(self,self.tr("Save Image"),Qstring(),self.tr("Image Files(*.png *.jpg *.bmp)"))
```

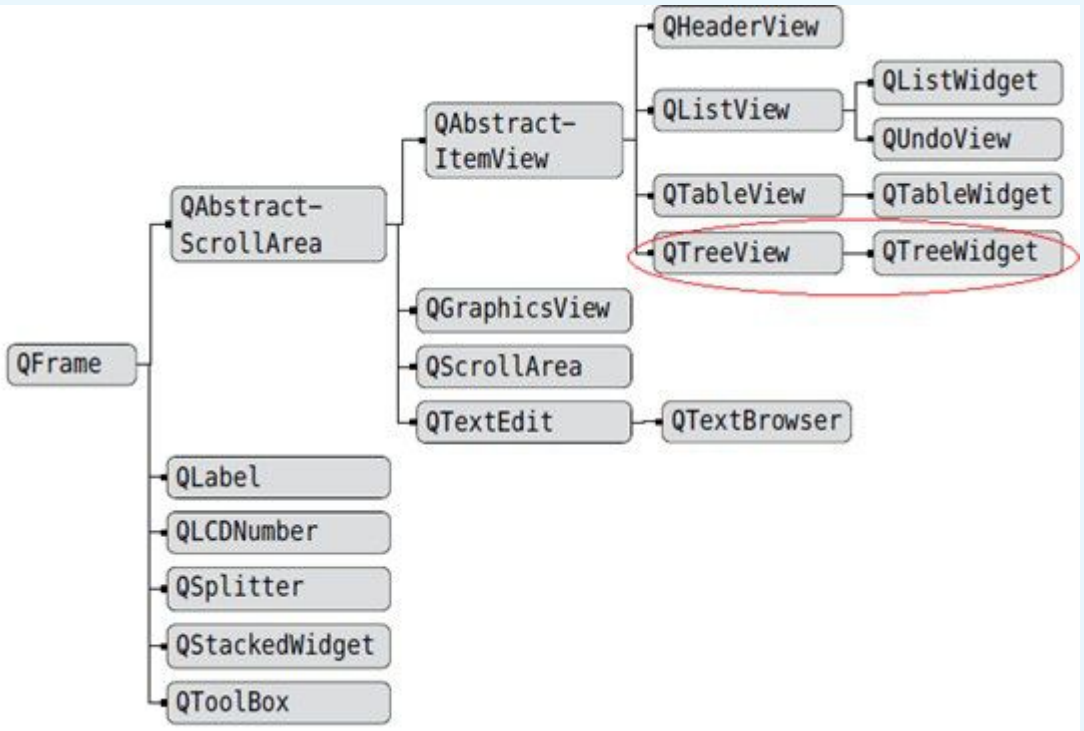
此模式下，会自动判断是否文件名已存在等等。

QFile 类还为我们提供了自己定制文件对话框的功能。具体请参看帮助文档。

PyQt 学习笔记(16)——QTreeWidget

参考资料：Qt documentation online（因为这个帮助文档是基于 C++做的，里面的语句是 C++写的，不过因为 PyQt 做了很好的移植，方法的名称，参数等等基本都可以在 python 中套用）

QTreeWidget 的继承关系如下图：



The QTreeWidget class provides a tree view that uses a predefined tree model.

因为继承关系是 QAbstractItemView->QTreeView->QTreeWidget ，所以和 QTableWidget 很多地方是类似的。

如果需要特殊的模式，比如显示硬盘信息及内部文件的 dir 模式等，都需要用 QTreeView，而不是用 QTreeWidget。

和 QTableWidget 类似，一般步骤是先创建一个 QTreeWidget 实例，然后设置列数，然后再添加头。

```
# !/usr/bin/python
import sys
```

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
class TreeWidget(QMainWindow):
    def __init__(self,parent=None):
        QWidget.__init__(self,parent)
        self.setWindowTitle('TreeWidget')
        self.tree = QTreeWidget()
        self.tree.setColumnCount(2)
        self.tree.setHeaderLabels(['Key','Value'])
        root= QTreeWidgetItem(self.tree)
        root.setText(0,'root')
        child1 = QTreeWidgetItem(root)
        child1.setText(0,'child1')
        child1.setText(1,'name1')
        child2 = QTreeWidgetItem(root)
        child2.setText(0,'child2')
        child2.setText(1,'name2')
        child3 = QTreeWidgetItem(root)
        child3.setText(0,'child3')
        child4 = QTreeWidgetItem(child3)
        child4.setText(0,'child4')
        child4.setText(1,'name4')
        self.tree.addTopLevelItem(root)
        self.setCentralWidget(self.tree)
app = QApplication(sys.argv)
tp = TreeWidget()
tp.show()
app.exec_()
```

结果如下

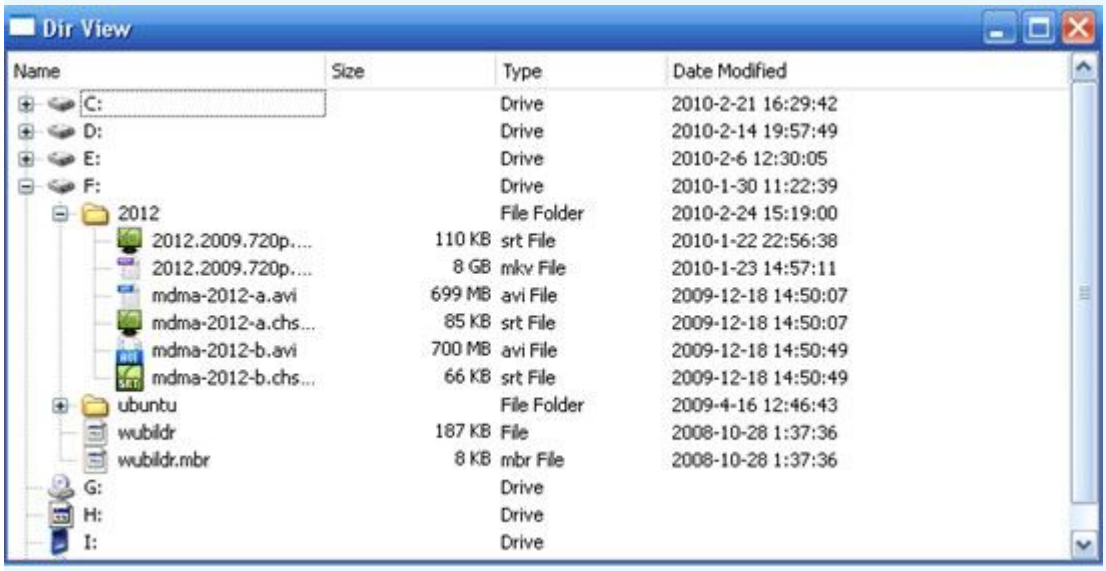
其中的 QtreeWidgetItem 就是一层层的添加的，其实还是不太方便的。

在应用程序中一般不是这样来创建 QTreeView 的，特别是比较复杂的 Tree，一般都是通过 QTreeView 来实现而不是 QTreeWidget 来实现的。

这种与 QTreeWidget 最大的区别就是，我们自己来定制模式，当然也有些系统提供给我们的模式，比如我们的文件系统盘的树列表，比如下面的：

```
import sys
from PyQt4 import QtCore, QtGui
if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    model = QtGui.QDirModel() #系统给我们提供的
    tree = QtGui.QTreeView()
    tree.setModel(model)
    tree.setWindowTitle(tree.tr("Dir View"))
    tree.resize(640, 480)
    tree.show()
    sys.exit(app.exec_())
```

结果如下所示：



所以一般的我们自己定制模式，步骤如下：

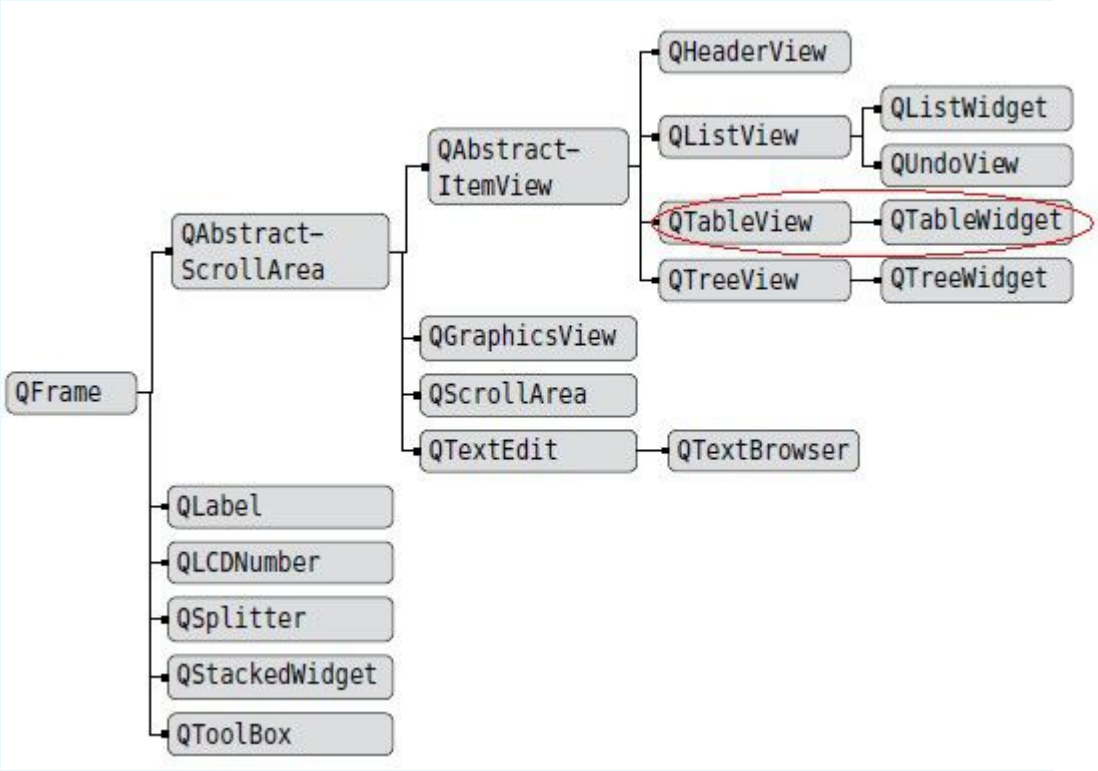
```
import sys
from PyQt4 import QtCore, QtGui
if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    model = TreeModel(需要处理的数据)
    view = QtGui.QTreeView()
    view.setModel(model)
    view.setWindowTitle("Simple Tree Model")
    view.show()
    sys.exit(app.exec_())
```

其中的 TreeModel 类是我们自己写的，如何显示我们的数据，可以查看相关资料。

PyQt 学习笔记(17)——QTableWidget

参考资料：Qt documentation online（因为这个帮助文档是基于 C++做的，里面的语句是 C++写的，不过因为 PyQt 做了很好的移植，方法的名称，参数等等基本都可以在 python 中套用）

先看一下类的继承图：



如上所示，QtableWidget 是继承于 QtableView 的。所以 QtableView 的方法也在 QtableWidget 中继承了。

QTableWidget 类提供了一个默认模式的表格，它是基于 Item 的，这个 Item 是由 QTableWidgetItem 提供的。如果你要构建自己的数据模式，请使用 QTableView 而不是 QTableWidget。

一，如何构建一个 QtableWidget。

```
# !/usr/bin/python
import sys
from PyQt4.QtGui import *
class TableWidget(QMainWindow):
    def __init__(self,parent=None):
        QWidget.__init__(self,parent)
        self.setWindowTitle('TableWidget')
        self.table = QTableWidget(10,6)
        self.setCentralWidget(self.table)
app = QApplication(sys.argv)
tb = TableWidget()
tb.show()
app.exec_()
```

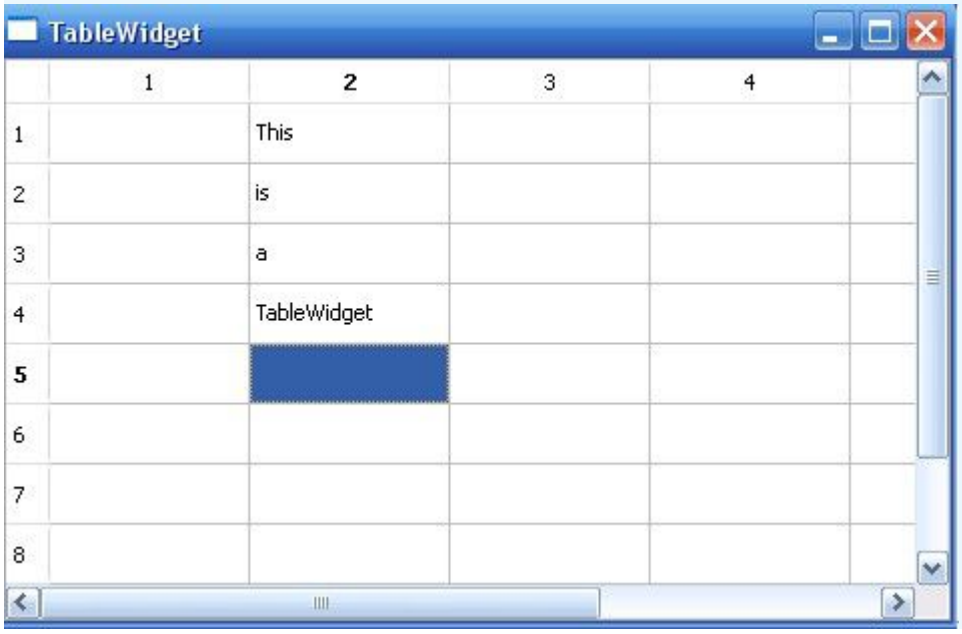
结果如下图所示：创建了一个10行6列的表格，可编辑可输入。

初始化的时候也可以不设置行数和列数。而等到创建完了以后再设。

比如：

```
self.table = QTableWidget()
self.table.setRowCount(10)
self.table.setColumnCount(6)
```

这样结果是跟上面一样的。



二，添加表头。

可以添加水平和垂直表头，QtableWidget 提供了两个方法来添加表头，非常方便。

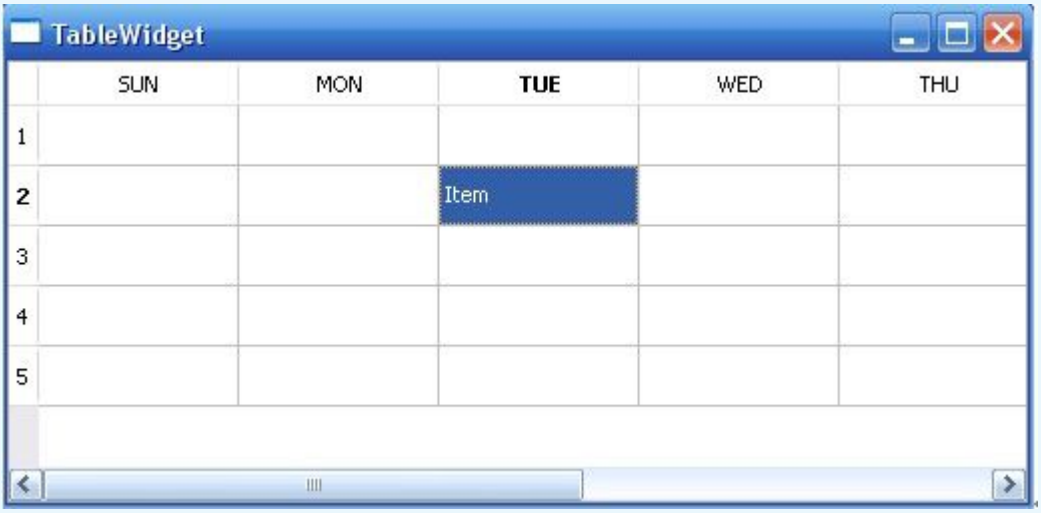
```
self.table = QTableWidget(5,7)
self.table.setHorizontalHeaderLabels(['SUN','MON','TUE','WED',
                                      'THU','FIR','SAT'])
```

上面两句就是添加水平表头。假如我们不添加表头，那么表头默认的数字就是代表所在行或者所在列。

三，添加表项。

```
self.newItem = QTableWidgetItem('Item')
self.table.setItem(1,2,self.newItem)
```

如下图：可以看出，行列数是指不算标题行，都是从第0行，或者第0列开始计数的。



下面我们通过循环来添加表项的所有内容：

```
self.table = QTableWidget(5,7)
self.table.setHorizontalHeaderLabels(['SUN','MON','TUE','WED',
                                      'THU','FIR','SAT'])
       for i in range(self.table.rowCount()):
           for j in range(self.table.columnCount()):
               cnt = '(%d,%d)'% (i,j)
               newItem = QTableWidgetItem(cnt)
               self.table.setItem(i,j,newItem)
```

QTableWidget.rowCount()是得到行数，int 型。

QTableWidget.columnCount()是得到列数，int 型

结果如下：

四，修改表项内容

QTableWidget.clear(self) 清楚所有表项及表头

QTableWidget.clearContents(self) 只清楚表项，不清楚表头。

QTableWidget.insertColumn(self, int column) 在某一列插入新的一列。

QTableWidget.insertRow(self, int row)在某一行插入新的一行。

QTableWidget.removeColumn(self, int column) 移除 column 列及其内容。

QTableWidget.removeRow(self, int row)移除第 row 行及其内容。

五，关于显示的一些问题，外观

QTableView.setShowGrid (self, bool show) 从 TableView 继承而来的，是否显示表格的横竖线，默认情况下是显示的，如上面的例子，如果设为 setShowGrid(False) ，则不显示分割线，横竖都没有。

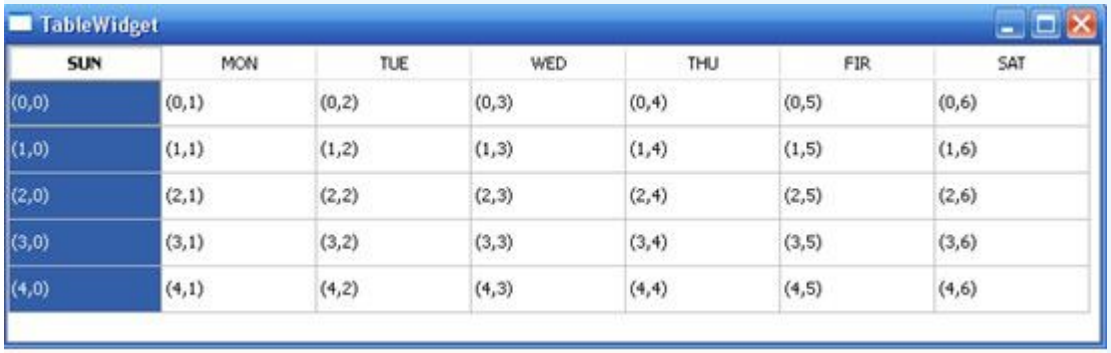
另外，还可以通过 hideRow(),hideColumn(),showRow(),showColumn()等来隐藏或显示特定行和列。

还有一个是否显示表头的问题，比如很多情况下我们只需要水平表头，不需要垂直表头，怎么办呢？我们在上面的例子中加上这么一句：

self.table.verticalHeader().setVisible(False)

setVisible 是所有 Qwidget 都有的方法，而 self.table.verticalHeader() 是得到了一个表头，表头也是 QheaderView 继承来的，也是 Qwidget 的子类，所以也可以调用 setVisible()方法来显示或者隐藏表头。

结果如下图：



因为继承关系，父类的很多方法都可以调用，所以 QTableWidget 的方法非常之多，应该有几百个，一一学习是不可能的，只能用到的时候去查。下面介绍几个继承于上面父类的方法。

QabstractItemView 是 QTableWidget 的父类的父类，他有下面几个方法，我们 QTableWidget 中经常调用，就是是否项目可编辑，点击选择是选择行，还是可以选择列，是可以选择多行（多列），还是只可以选择单行（单列），等等非常好用，如下的列子：

self.table.setEditTriggers(QTableWidget.NoEditTriggers)

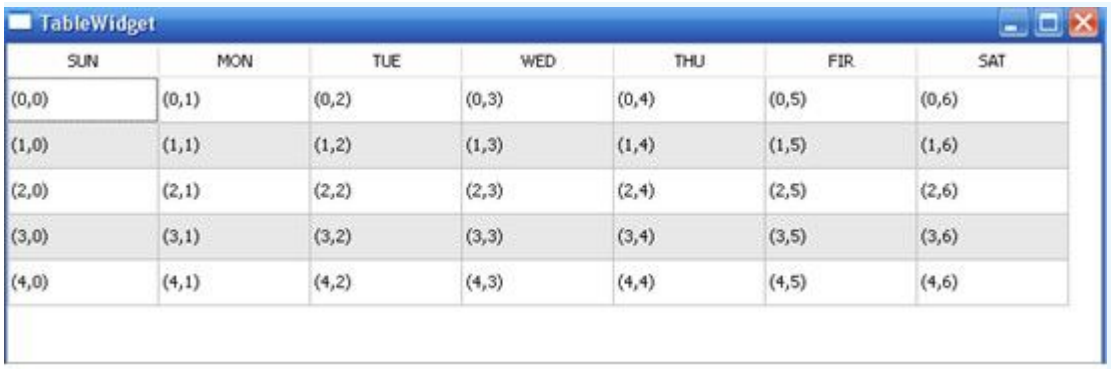
self.table.setSelectionBehavior(QTableWidget.SelectRows)

self.table.setSelectionMode(QTableWidget.SingleSelection)

self.table.setAlternatingRowColors(True)

第一句，设为不可编辑状态，第二是选择行，第三是选择单个行，第四是隔行改变颜色。

结果如下：



不能编辑，不能选择一列或者单个项目，只能选择单一行。

另外可以修改行宽高等大小信息，还有各行，列，等的颜色问题，带图标的标题等等美化措施，请参看 documentation。

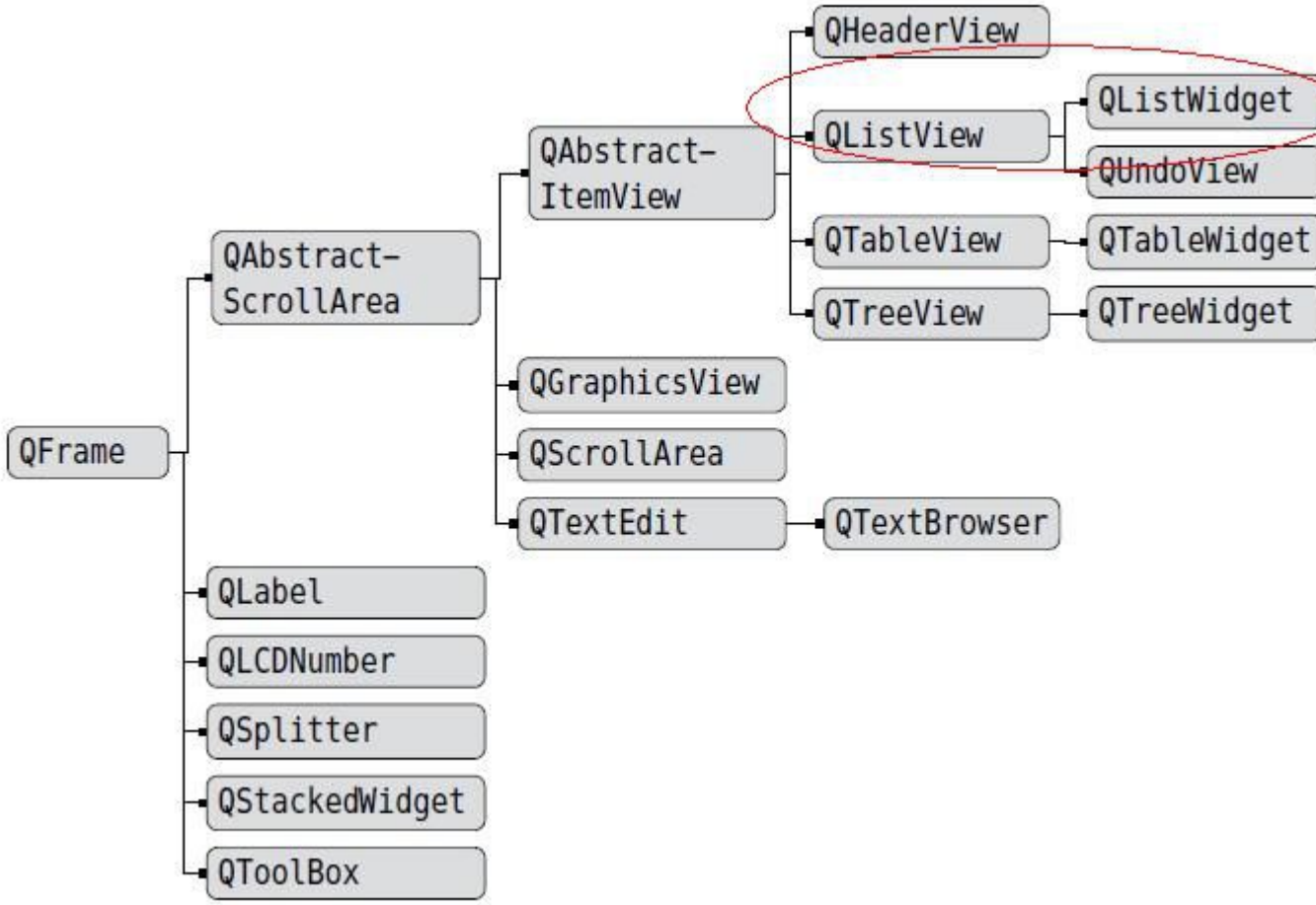
六，Qt Singal

参看帮助文档

PyQt 学习笔记(18)——QListWidget

参考资料：QT documentation

QListWidget 是一个列表框，使用非常简单：它的继承关系：



例子:

```
# !/usr/bin/python
import sys
from PyQt4.QtGui import *
from PyQt4 import QtCore

class ListWidget(QMainWindow):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent)
        self.setWindowTitle('ListWidget')
        self.List = QListWidget(self)
        self.List.setSortingEnabled(1)
        item = ['OaK','Banana','Apple',' Orange','Grapes','Jayesh']
        listItem = []
        for lst in item:
            listItem.append(QListWidgetItem(lst))
```

```
        for i in range(len(listItem)):
            self.List.insertItem(i+1,listItem[i])
        self.setCentralWidget(self.List)


app = QApplication(sys.argv)
tb = ListWidget()
tb.show()
app.exec_()
```

其中的 `self.List.setSortingEnabled(1)` 是排序，按字母来的，如果没有这句，默认是不排序的。

结果：