PyQt5 入门（第二版）

万泽

2016 年 3 月 19 日

目 录

前言

第二版声明

随着作者编程水平的上升，对很多内容都有了不同的理解。本文进行了进一步的重新修改，推出第二版。

本文的例子都是基于 PyQt5。如果读者需要使用 PyQt4，请参看附录后面的 PyQt4 和 PyQt5 的区别整理一小节。

安装和配置

安装 pyqt5

就 ubuntu 下安装 pyqt5 是很简单的，如下所示：

```
sudo apt-get install python3-pyqt5
```

如果你使用的是默认的 python3 版本的话，否则你可能需要手工编译 pyqt5。

其中安装 Qt5 不需要我们多费心，请确保下面几个软件包安装上去了（参考了 这个网页 和 这个网页 ）：

```
sudo apt-get install qt5-default
sudo apt-get install qtbase5-dev
sudo apt-get install qtdeclarative5-dev
```

上面的第三个可能并不需要安装，其中第一个 `qt5-default` 和 qmake 的 v5 版本有关，然后 `qtbase5-dev` 肯定是需要安装的。不管怎么说，确保这些都装上吧。

然后再在 PyQt 的官网 下载 `SIP` 的源码，运行 `python3 configure.py` 输出 makefile，后面就是大家熟悉的 `make`，`sudo make install` 。

继续再下载 `PyQt5` 的源码，安装步骤同上，这里就不赘述了。

在后面我们会提到 `pyuic5` 和 `pyrcc5` 命令，其在下面这个软件包里面：

```
sudo apt-get install pyqt5-dev-tools
```

检查 pyqt5 安装情况执行以下脚本即可，显示的是当前安装的 pyqt5 的版本号:

```
>>> from PyQt5.QtCore import QT_VERSION_STR
>>> print(QT_VERSION_STR)
5.2.1
```

本文的代码都是 PyQt 版本号都是上面的，没有特别的理由，会一直维持在这个版本号里面了。

安装 pyqt4

安装 pyqt4 大致过程类似上面，只是一些细节上的改动了，比如:

ubuntu 下安装 pyqt4 即:

```
sudo apt-get install python3-pyqt4
```

如果你需要使用 qt designer 来辅助设计 GUI，你还需要额外安装 qt designer 软件和 pyuic4 和 pyrcc4 命令。（顺便再次提醒下 pyrcc4 对中文目录目前支持有问题（201410））

```
sudo apt-get install pyqt4-dev-tools qt4-designer
```

检查 pyqt4 安装情况执行以下脚本即可，显示的是当前安装的 pyqt4 的版本号:

```
from PyQt4.QtCore import QT_VERSION_STR
print(QT_VERSION_STR)
```

beginning

窗口

请看到下面的代码:

```
import sys
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class MyWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setGeometry(0, 0, 800, 600)
        # 坐标 0 0 大小 800 600
        self.setWindowTitle('myapp')

myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

首先导入 sys 宏包，这是为了后面接受 sys.argv 参数。关于引入这里值得一提的是: 很多 pyqt4 原放在 QtGui 里面的一些 QWidget 在 pyqt5 里面都放入 QtWidgets 里面去了。

接下来我们定义了 MyWidget 类，它继承自 QWidget 类。然后通过 QWidget 类的 `setGeometry` 方法来调整窗口的左顶点的坐标位置和窗口的大小。

然后通过 `setWindowTitle` 方法来设置这个窗口程序的标题，这里就简单设置为”myapp” 了。

任何窗口程序都需要创建一个 QApplication 类的实例，这里是 myapp。然后接下来创建 QWidget 类的实例 mywidget，然后通过调用 mywidget 的方法 `show` 来显示窗体。

最后我们看到系统要退出是调用的 myapp 实例的 exec_ 方法。

加上图标

```
import sys
from PyQt5.QtGui import QIcon
from PyQt5.QtWidgets import QWidget, QApplication

class MyWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))

myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

这个程序相对上面的程序就增加了一个 setWindowIcon 方法，这个方法调用了 QtGui.QIcon 方法，然后后面跟的就是图标的存放路径，使用相对路径。在运行这个例子的时候，请随便弄个图标文件过来。

为了简单起见这个程序就使用了 QWidget 类的 resize 方法来设置窗体的大小。

弹出提示信息

```
import sys
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
```

```
class MyWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
        self.setToolTip(' 看什么看 ^_^')
        QToolTip.setFont(QFont\
        (' 微软雅黑', 12))

myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

上面这段代码和前面的代码的不同就在于 MyWidget 类的初始函数新加入了两条命令。其中 `setToolTip` 方法设置具体显示的弹出的提示文本内容，然后后面调用 QToolTip 类的 `setFont` 方法来设置字体和字号，我不太清楚这里随便设置系统的字体微软雅黑是不是有效。

这样你的鼠标停放在窗口上一会儿会弹出一小段提示文字。

关闭窗体时询问

```
import sys
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class MyWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.setWindowTitle('myapp')
```

```
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
        self.setToolTip(' 看什么看 ^_^')
        QToolTip.setFont(QFont\
        (' 微软雅黑', 12))

    def closeEvent(self, event):
        # 重新定义 colseEvent
        reply = QMessageBox.question\
        (self, ' 信息',
            " 你确定要退出吗? ",
             QMessageBox.Yes,
             QMessageBox.No)
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()

myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

这段代码和前面代码的不同就是重新定义了 `colseEvent` 事件。这段代码的核心就是 QtGui 类的 QMessageBox 类的 question 方法，这个方法将会弹出一个询问窗体。这个方法接受四个参数：第一个参数是这个窗体所属的母体，这里就是 self 也就是实例 mywidget；第二个参数是弹出窗体的标题；第三个参数是一个标准 button；第四个参数也是一个标准 button，是默认（也就是按 enter 直接选定的）的 button。然后这个方法返回的是那个被点击了的标准 button 的标识符，所以后面和标准 `QMessageBox.Yes` 比较了，然后执行 event 的 accept 方法。

屏幕居中显示窗体

```
import sys
from PyQt5.QtGui  import *
from PyQt5.QtWidgets import *

class MyWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.center()
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
        self.setToolTip(' 看什么看 ^_^')
        QToolTip.setFont(QFont\
        (' 微软雅黑', 12))

    def closeEvent(self, event):
        # 重新定义 colseEvent
        reply = QMessageBox.question\
        (self, ' 信息',
            " 你确定要退出吗? ",
             QMessageBox.Yes,
             QMessageBox.No)
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
     #center method
    def center(self):
        screen = QDesktopWidget().screenGeometry()
        size =  self.geometry()
        self.move((screen.width()-size.width())/2,\
         (screen.height()-size.height())/2)
```

```
myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

这个例子和前面相比改动是新建了一个 `center` 方法，接受一个实例，这里是 mywidget。然后对这个实例也就是窗口的具体位置做一些调整。

QDesktopWidget 类的 `screenGeometry` 方法返回一个量，这个量的 width 属性就是屏幕的宽度（按照 pt 像素计，比如 1366×768，宽度就是 1366），这个量的 height 属性就是屏幕的高度。

然后 QWidget 类的 `geometry` 方法同样返回一个量，这个量的 width 是这个窗体的宽度，这个量的 height 属性是这个窗体的高度。

然后调用 QWidget 类的 move 方法，这里是对 mywidget 这个实例作用。我们可以看到 move 方法的 X，Y 是从屏幕的坐标原点 (0,0) 开始计算的。第一个参数 X 表示向右移动了多少宽度，Y 表示向下移动了多少高度。

整个函数的作用效果就是将这个窗体居中显示。

QMainWindow 类

QtGui.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。（之前使用的是 QWidget 类，现在换成 QMainWindow 类。）

前面第一个例子都是用的 QtGui.QWidget 类创建的一个窗体。关于 QWidget 和 QMainWindow 这两个类的区别 根据这个网站 得出的结论是：QWdget 类在 Qt 中是所有可画类的基础（这里的意思可能是窗体的基础吧。）任何基于 QWidget 的类都可以作为独立窗体而显示出来而不需要母体（parent）。

QMainWindow 类是针对主窗体一般需求而设计的，它预定义了菜单栏状态栏和其他 widget（窗口小部件）。因为它继承自 QWidget，所以前面谈及的一些属性修改都适用于它。那么首先我们将之前的代码中的 QWidget 类换成 QMainWindow 类。

```
import sys
from PyQt5.QtGui  import *
from PyQt5.QtWidgets import *

class MyWidget(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.center()
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
        self.setToolTip(' 看什么看 ^_^')
        QToolTip.setFont(QFont\
        (' 微软雅黑', 12))

    def closeEvent(self, event):
        # 重新定义 colseEvent
        reply = QMessageBox.question\
        (self, ' 信息',
            " 你确定要退出吗? ",
             QMessageBox.Yes,
             QMessageBox.No)
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
     #center method
    def center(self):
        screen = QDesktopWidget().screenGeometry()
        size =  self.geometry()
        self.move((screen.width()-size.width())/2,\
         (screen.height()-size.height())/2)
```

```
myapp = QApplication(sys.argv)
mywidget = MyWidget()
mywidget.show()
sys.exit(myapp.exec_())
```

现在程序运行情况良好，我们继续加点东西进去。

加上状态栏

```
import sys
from PyQt5.QtGui  import *
from PyQt5.QtWidgets import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(800,600)
        self.center()
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
        self.setToolTip(' 看什么看 ^_^')
        QToolTip.setFont(QFont\
        (' 微软雅黑', 12))

    def closeEvent(self, event):
        # 重新定义 colseEvent
        reply = QMessageBox.question\
        (self, ' 信息',
            " 你确定要退出吗? ",
             QMessageBox.Yes,
             QMessageBox.No)
```

```
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
     #center method
    def center(self):
        screen = QDesktopWidget().screenGeometry()
        size =  self.geometry()
        self.move((screen.width()-size.width())/2,\
         (screen.height()-size.height())/2)

myapp = QApplication(sys.argv)
mainwindow = MainWindow()
mainwindow.show()
mainwindow.statusBar().showMessage(' 程序已就绪...')
sys.exit(myapp.exec_())
```

这个程序和前面的区别在于最后倒数第二行，调用 mainwindow 这个 QMainWindow 类生成的实例的 `statusBar` 方法生成一个 QStatusBar 对象，然后调用 QStatusBar 类的 `showMessage` 方法来显示一段文字。

如果你希望这段代码在 `__init__` 方法里面，那么具体实现过程也与上面描述的类似。

加上菜单栏

```
import sys

from PyQt5.QtGui  import *
from PyQt5.QtWidgets import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.initUI()

    def initUI(self):
        self.resize(800,600)
        self.center()
        self.setWindowTitle('myapp')
        self.setWindowIcon(QIcon\
        ('icons/myapp.ico'))
    # 菜单栏
        menu_control = self.menuBar().addMenu('Contorl')
        act_quit = menu_control.addAction('quit')
        act_quit.triggered.connect(self.close)

        menu_help = self.menuBar().addMenu('Help')
        act_about = menu_help.addAction('about...')
        act_about.triggered.connect(self.about)
        act_aboutqt = menu_help.addAction('aboutqt')
        act_aboutqt.triggered.connect(self.aboutqt)


    # 状态栏
        self.statusBar().showMessage(' 程序已就绪...')
        self.show()

    def about(self):
        QMessageBox.about(self,"about this software","wise system")
    def aboutqt(self):
        QMessageBox.aboutQt(self)

    def closeEvent(self, event):
        # 重新定义 colseEvent
        reply = QMessageBox.question\
        (self, ' 信息',
```

```
            " 你确定要退出吗？",
             QMessageBox.Yes,
             QMessageBox.No)
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
     #center method
    def center(self):
        screen = QDesktopWidget().screenGeometry()
        size =  self.geometry()
        self.move((screen.width()-size.width())/2,\
         (screen.height()-size.height())/2)

myapp = QApplication(sys.argv)
mainwindow = MainWindow()
sys.exit(myapp.exec_())
```

和上面讨论加上状态栏类似，这里用 QMainWindow 类的 `menuBar` 方法来获得一个菜单栏对象。然后用这个菜单栏对象的 `addMenu` 方法来创建一个新的菜单对象（QMenu 类），addMenu 方法里面的内容是新建菜单要显示的文本。

然后继续给之前的菜单对象加上动作，调用菜单对象的 `addAction` 方法，我们看到 menuBar 创建了一个菜单栏对象，然后使用 addMenu 方法创建了一个菜单，同时返回的是一个菜单对象，然后对这个菜单对象使用 addAction 方法，这个方法给菜单添加了一个动作，或者说一个 item 一个内容，然后 addAction 返回的是一个动作对象，然后对这个动作对象进行信号－槽机制连接，将其和一个函数连接起来了。

在这里这个动作对象，就是菜单的下拉选项，如果我们用鼠标点击一下的话，将会触发 `triggered` 信号，如果我们 connect 方法连接到某个槽上（或者某个你定义的函数），那么将会触发这个函数的执行。下面就信号－槽机制详细说明之。

信号－槽机制

GUI 程序一般都引入一种事件和信号机制，well，简单来说就是一个循环程序，这个循环程序等到某个时刻程序会自动做某些事情比如刷新程序界面啊，或者扫描键盘鼠标之类的，等用户点击鼠标或者按了键盘之后，它会接受这个信号然后做出相应的反应。

所以你一定猜到了，`close` 函数可能就是要退出这个循环程序。我们调用主程序的 `exec_` 方法，就是开启这个循环程序。

pyqt4 的旧的信号－槽连接语句我在这里忽略了，在这里值得提醒的是 pyqt5 已经不支持旧的信号－槽连接语句了。下面就新的语句说明之。

```
act_exit.triggered.connect(self.close)
```

我们看到新的信号－槽机制语句变得更精简更易懂了。整个过程就是如我前面所述，某个对象发出了某个信号，然后用 connect 将这个信号和某个槽（或者你定义的某个函数）连接起来即形成了一个反射弧了。

这里的槽就是 self 主窗口实例的 close 方法，这个是主窗口自带的函数。

然后我们看到 aboutqt 和 about 函数。具体读者如果不懂请翻阅 QMessageBox 类的静态方法 `about` 和 `aboutqt` 。

如何查阅资料

查看 pydoc

如果要看 python3 的所有模块用 help 函数之后生成的信息，使用如下命令打开网页查看。

```
pydoc3  -b
```

如果要看 python2 的信息：

```
pydoc  -p 1234
```

这里 -p 后面跟的是你的 localhost 的端口号，只要不被使用就行。

相关网络资源

请参看本文最下面的参考资料部分。

信号－槽详解

考虑到 pyqt5 只支持新式信号－槽机制了，这里将新式信号－槽机制详细说明，由于新式信号－槽机制在 pyqt4 上也能正常运行，所以新来的程序员推荐都用新式信号－槽机制。下面只介绍新式的信号－槽机制。

信号 (singal) 可以连接无数多个槽 (slot)，或者没有连接槽也没有问题，信号也可以连接其他的信号。正如前面所述，连接的基本语句形式如下：`who.singal.connect(slot)`。比如说按钮最常见的内置信号 `triggered` ，而槽实际上就是某个函数，比如主窗体的 `self.close` 方法。

信号就是 `QObject` 的一个属性，pyqt 的窗体有很多内置信号，你也可以定义自己的信号，这个后面再提及。信号还没和槽连接起来就只是一个属性，只有通过 `connect` 方法连接起来，信号－槽机制就建立起来了。类似的信号还有 `disconnect` 方法和 `emit` 方法。disconnect 就是断开信号－槽机制，而 emit 就是激活那个信号。

pyqt 很多内置信号和内置槽将 GUI 的事件驱动细节给隐藏了，如果你自己定义自己的信号或者槽可能对 who.singal.connect(slot) 这样简洁的形式如何完成工作的感到困惑。这里先简要地介绍一下。

信号都是类的一个属性，新的信号必须继承自 QObject，然后由 `PyQt5.QtCore.pyqtSingal`（在 pyqt4 下是 PyQt4.QtCore.pyqtSingal。）方法创建，这个方法接受的参数中最重要的是 types 类型，比如 int，bool 之类的，你可以认为这是信号传递的参数类型，但实际传递这些参数值的是 emit 方法。然后槽实际上就是经过特殊封装的函数，这些函数当然需要接受一些参数或者不接受参数，而这些参数具体的值传进来的是由 emit 方法执行的，然后我们通过 who.singal.connect(slot) 这样的形式将某个信号和某个槽连接起来，who 的信号，然后信号类自带的连接方法，然后连接到 slot 某个函数上，在这里隐藏的一个重要细节就是 emit 方法，比如说你定义一个新的信号，需要将点击屏幕的具体 x,y 坐标发送出去，内置的信号－槽将这一机制都完成了，如果你自己定义的信号和槽的话，比如 `pyqtSingal(int,int)` ，发送给 func(x,y)，具体 x 和 y 的值你需要

通过 emit(x,y) 来发送。至于什么时候发送，已经发送的 x,y 值的获取，这应该又是另外一个信号—槽机制的细节。

请看下面这个例子：

```
import sys
from PyQt5.QtWidgets import QHBoxLayout,QSlider,QSpinBox,QApplication,QWidget
from PyQt5.QtCore import Qt

app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("enter your age")
spinBox = QSpinBox()
slider = QSlider(Qt.Horizontal)
spinBox.setRange(0,130)
slider.setRange(0,130)

spinBox.valueChanged.connect(slider.setValue)
slider.valueChanged.connect(spinBox.setValue)

spinBox.setValue(35)

layout = QHBoxLayout()
layout.addWidget(spinBox)
layout.addWidget(slider)



window.setLayout(layout)
window.show()

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

第 16 行将 spinBox 的 `valueChanged` 信号和 slider 的 `setValue` 槽连接起来了，其中 QSpinBox 内置的 `valueChanged` 信号发射自带的一个参数就是改变后的

值，这个值传递给了 QSlider 的内置槽 `setValue` ，从而将 slider 的值设置为新值。第 17 行如果 slider 的值发生了改变，那么会发送 valueChanged 信号，然后又传递给了 spinBox，并执行了内置槽 setValue，由于此时的值即为原值，这样 spinBox 内的值就没有发生改变了，如此程序不会陷入死循环。

自定义信号

正如前所述及自定义信号由 PyQt5.QtCore.pyqtSingal（在 pyqt4 下是 PyQt4.QtCore.pyqtSinga 方法创建，具体格式如下：

```
from PyQt5.QtCore import QObject, pyqtSignal

class Foo(QObject):
    closed = pyqtSignal()
    range_changed = pyqtSignal(int, int, name='rangeChanged')
```

上面 Foo 类里面自定义了一个新的信号，它必须是 GObject 的子类。然后定义了一个 closed 信号，没有接受任何参数。下面是 range_changed 信号，接受了一个 int 和一个 int 类型，然后这个信号的名字是 rangeChanged，name 选项是一个可选项，如果不填那么信号的名字就是 range_changed。

信号还可以 overload，不过似乎不太适合 python。

注意信号必须定义为类的属性，同时必须是 GObject 的子类。

自定义槽

按照 python 格式自己定义的函数就是所谓的自定义槽了。不过推荐用 pyqt 的槽装饰器来定义槽。

```
from PyQt4.QtCore import  pyqtSlot
    #1
    @pyqtSlot()
    def foo(self):
```

```
        pass
    #2
    @pyqtSlot(int, str)
    def foo(self, arg1, arg2):
        pass
    #3
    @pyqtSlot(int, name='bar')
    def foo(self, arg1):
        pass
    #4
    @pyqtSlot(int, result=int)
    def foo(self, arg1):
        pass
    #5
    @pyqtSlot(int, QObject)
    def foo(self, arg1):
        pass
```

上面的第一个例子定义了名叫 foo 的一个槽，然后不接受任何参数。第二个槽接受一个 int 类型的值和 str 类型的值。第三个槽名字叫做 bar，接受一个 int 类型的值，第四个槽接受一个 int 类型的值，然后返回的是一个 int 类型的值，第五个操作接受一个 int 类型的值和一个 GObject 类型的值，此处应该暗指其他 pyqt 窗体类型都可以作为参数进行传递。

```
@pyqtSlot(int)
@pyqtSlot('QString')
def valueChanged(self, value):
    pass
```

这里定义了两个槽，名字都叫做 valueChanged，一个接受 int 类型，一个接受 QString 类型，同前面信号的 overload 一样，在 python 中不推荐这么使用，还是明晰一点比较好。

发射信号

信号对象有 emit 方法用来发射信号，然后信号对象还有 disconnect 方法断开某个信号和槽的连接。

一个信号可以连接多个槽，多个信号可以连接同一个槽，一个信号可以与另外一个信号相连接。

下面通过一个例子详解自建信号还有自建槽并建立发射机制的情况。

```
from PyQt5.QtWidgets import QDialog,QLabel,QLineEdit,QCheckBox,QPushButton,QHBoxLayout,QVBoxLay
from PyQt5.QtCore import  Qt ,pyqtSignal,QObject,pyqtSlot


class FindDialog(QDialog):
    findNext = pyqtSignal(str,Qt.CaseSensitivity)
    findPrevious = pyqtSignal(str,Qt.CaseSensitivity)

    def __init__(self,parent=None):
        super().__init__(parent)
        label = QLabel(self.tr("Find &what:"))
        self.lineEdit = QLineEdit()
        label.setBuddy(self.lineEdit)

        self.caseCheckBox=QCheckBox(self.tr("Match &case"))
        self.backwardCheckBox=QCheckBox(self.tr("Search &backward"))
        self.findButton = QPushButton(self.tr("&Find"))
        self.findButton.setDefault(True)
        self.findButton.setEnabled(False)
        closeButton=QPushButton(self.tr("Close"))

        self.lineEdit.textChanged.connect(self.enableFindButton)
        self.findButton.clicked.connect(self.findClicked)
        closeButton.clicked.connect(self.close)
```

```
        topLeftLayout=QHBoxLayout()
        topLeftLayout.addWidget(label)
        topLeftLayout.addWidget(self.lineEdit)
        leftLayout=QVBoxLayout()
        leftLayout.addLayout(topLeftLayout)
        leftLayout.addWidget(self.caseCheckBox)
        leftLayout.addWidget(self.backwardCheckBox)
        rightLayout = QVBoxLayout()
        rightLayout.addWidget(self.findButton)
        rightLayout.addWidget(closeButton)
        rightLayout.addStretch()
        mainLayout=QHBoxLayout()
        mainLayout.addLayout(leftLayout)
        mainLayout.addLayout(rightLayout)
        self.setLayout(mainLayout)

        self.setWindowTitle(self.tr("Find"))
        self.setFixedHeight(self.sizeHint().height())


    def enableFindButton(self,text):
        self.findButton.setEnabled(bool(text))
    @pyqtSlot()
    def findClicked(self):
        text = self.lineEdit.text()
        if self.caseCheckBox.isChecked():
            cs=Qt.CaseSensitive
        else:
            cs=Qt.CaseInsensitive

        if self.backwardCheckBox.isChecked():
            self.findPrevious.emit(text,cs)
        else:
```

```
            self.findNext.emit(text,cs)


if __name__ == '__main__':
    import sys
    app=QApplication(sys.argv)
    findDialog = FindDialog()
    def find(text,cs):
        print('find:',text,'cs',cs)
    def findp(text,cs):
        print('findp:',text,'cs',cs)

    findDialog.findNext.connect(find)
    findDialog.findPrevious.connect(findp)
    findDialog.show()
    sys.exit(app.exec_())
```

首先自建的信号必须是类的属性，然后这个类必须是 QObject 的子类，这里 QDialog 是继承自 QObject 的。请看到第 9 行和第 10 行，通过 pyqtSignal 函数来自建信号，此信号有两个参数，一个是 str 字符变量，一个是 Qt.CaseSensitivity 的枚举值。假设我们输入一些文字了，然后点击 Find 按钮，请看到第 26 行，点击之后将执行 findClicked 槽，按钮的 clicked 信号是不带参数的。所以后面定义的 findClicked 槽（简单的函数也可以）也没有任何参数。

findClicked 槽的 53-57 行确定了当前的 QLineEdit 的 text 值和 cs 也就是大小写是否检查的状态。然后根据向前或者向后是否勾选来确定接下来要发送的信号。比如 findNext 信号调用 emit 方法，对应两个参数也传递过去了。而这个 findNext 正是我们前面自定义的信号，正是对应的两个参数类型。

我们再看到这里简单做了一个测试程序，70-73 行定义了两个简单的函数，然后 75，76 行将 findDialog 的这两个信号和上面两个函数连接起来。于是当我们点击 Find 按钮，首先执行 findClicked 槽，然后假设这里发送了 findNext 信号（附带两个参数），然后信号又和 find 函数相连（参数传递给了 find 函数），然后执行 find 函数。整个过程就是这样的。

信号－槽机制的反思

在接下来 Qt designer 这一章也会详细讨论这个问题，我们使用 Qt designer 来设计和修改 ui 文件——对应程序中大部分的静态视图元素，主要的目的倒不是为了快速 GUI 程序编写，其实写代码也挺快的，主要的目的就是为了代码复用。当我们养成习惯，强迫自己程序中的静态视图元素都进入 ui 文件，这不仅增强了 ui 文件的复用性，而且也增强了剩下来的 python 代码的复用性。这其中很大一部分就是这里讨论的信号－槽机制的功劳。

当我们自定义的类加载好 ui 文件之后，该类里面的代码实际上就剩下两个工作：

1. 把本窗体的信号和槽都编写好
2. 把母窗体和子窗体和信号－槽接口写好。

一般程序的用户互动接口大多在最顶层，也就是用户一般喜欢在菜单栏找到所有可能对程序的控制，这些控制的实现函数如果放在都放在母窗体，那么整个程序的代码复用性会降到最低，而如果我们将这些实现函数分别移到和其视图窗体最紧密的窗体类中，那么不仅代码复用性会大大提高，而且这些槽或函数的编写也会简单很多。那么我们该如何组织这些信号和槽（实现函数）呢？我在这里提出组织学上的一些抽象原则：

1. 最小组织原则，凡是小组织能够自我实现的功能绝不上传到更大一级的组织中去。
2. 大组织对小组织元素的某些实现的引用，采用明文引用原则。比如说母窗体中有一个小窗体有一个编辑器，母窗体想要操控这个编辑器执行剪切操作，那么采用明文引用，也就是 self.textEdit.cut。
3. 小组织对大组织属性的引用采用信号激活原则，比如说某个编辑器发生了内容修改，你可以自定义一个信号，该信号为标题修改信号，然后信号触发母窗体的某个方法，这样达到修改母窗体的标题的目的。而在母窗体中，只需要在声明是将小组织的信号和大组织的某个方法连接起来即可。

使用 Qt designer

其实我们不一定要使用 Qt designer，Qt desinger 的目的主要不是为了快速绘制 GUI，而是一种模块化编程思路。利用 Qt designer 在代码复用上 ui 文件只是很小的一部分，关键是将 ui 抽离之后，剩下的 py 文件里面定义的大多是信号和槽，其中槽就

是函数，这些函数复用性是很高的。而对于不同的程序最大差异化的不分就是不同的信号和信号与槽之间的连接了。下面将通过一个 timer 计时器小程序简单演示下如何利用 Qt designer 快速 Qt 编程。

资源文件管理

资源管理

pyqt 都用 qrc 文件来管理软件内部的资源文件（如图标文件，翻译文件等）。qrc 文件的编写格式如下：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
</qresource>
</RCC>
```

qrc 的编写还是很简单的，完全可以手工编写之。上面代码第三行的 images/copy.png 的意思就是 qrc 文件所在目录下的 images 文件夹，里面的 copy.png 文件。

qrc 文件编写好了你需要运行如下命令

```
pyrcc5  wise.qrc  -o  wise_rc.py
```

这样将会输出一个 wise_rc.py 文件，你如果要使用里面的资源，首先

```
import  wise_rc
```

然后引用路径如下’:/images/copy.png’，这样就可以使用该图标文件了。

上面是 pyqt5 的情况，对于 pyqt4 类似的有：

```
pyrcc4  wise.qrc  -o  wise_rc.py
```

值得一提的是 pyrcc4 还有一个额外的选项 `-py3`，用于生成 python3 的代码。

推荐一个项目里面所有的资源文件都用一个 qrc 文件来管理。

配置文件管理

pyqt4 和 pyqt5 里的 QtCore 子模块里提供了 **QSettings** 类来方便管理软件的配置文件。

QSettings 构造函数

一般先推荐把 OrganizationName 和 ApplicationName 设置好。

```
app.setOrganizationName("Wise")
app.setApplicationName("wise")
```

然后接下来是构建一个 QSettings 对象。

```
QSettings(parent)
```

在设置好组织名和软件名之后，如果如上简单 `QSettings()` 来创建一个配置文件对象，不带任何参数，parent 取默认值，那么所谓的 format 取的默认值是 `QSettings.NativeFormat`，然后所谓的 scope 取的默认值是 `QSettings.UserScope`。这里的 scope 还有 QSettings.SystemScope，这个和软件的配置文件权限有关，这里先略过了，一般就使用默认的 UserScope 吧。

fromat 如果取默认的 NativeFormat 那么具体软件配置文件的安装目录如下：

- 如果是 linux 系统，比如上面的例子具体配置文件就是：

```
/home/wanze/.config/Wise/wise.conf
```

- 如果是 windows 系统，那么上面的例子具体就是：

```
HKEY_CURRENT_USER\Software\Wise\wise
```

windows 下配置是放在注册表里面的。

- 苹果系统还需要一个 OrganizationDomain 变量去 set，然后苹果系统我非常不熟悉，这里略过了。

IniFormat

如果你希望配置文件都以 ini 形式存储，那么你需要采取如下格式初始化配置文件对象：

```
self.settings = QSettings(QSettings.IniFormat,QSettings.UserScope,”Wise”,”wise”)
```

这样配置文件就在这里：`/home/wanze/.config/Wise/wise.ini` 。这里是 linux 系统的情况，苹果系统略过，windows 系统官方文档给出的是：`%APPDATA%\Wise\wise.ini` ，这个 `%APPDATA%` 我不清楚具体在哪里。

你可以通过调用 `self.settings.fileName()` 来查看该配置文件对象具体的路径所在。

推荐配置文件作为 mainwindow 实例的属性如上 self.settings 来确定，然后所有的子窗体都可以通过调用 self 来获得同一的配置文件对象。

ini 文件存放 DIY

如果你希望 ini 文件放在你喜欢的地方，下面是配置文件构造函数的第三种形式：

```
QSettings(”wise.ini”,QSettings.IniFormat)
```

第一个参数是你的配置文件名，第二个参数是 format。如上相对路径的话则是从你目前软件运行时的文件夹算起。

你可以通过调用 `settings.fileName()` 来看看该配置文件的具体所在。

ini 文件注意事项

ini 文件是大小写不敏感的，所以尽量避免两个变量名相近只是大小写不同。

不要使用“\”和“/”。windows 里 \ 会转换成/，而“/”使用来表示配置文件中分组关系的。

存值和读值

配置文件对象建立之后你就可以很方便地存放一些值和读取值了。存值用 `setValue` 方法，取值用 `value` 方法。如下所示：

```
settings.setValue(”editor/wrapMargin”, 68)
margin = self.settings.value(”editor/wrapMargin”)
```

如果 setValue 的键在配置文件对象中已经存在，那么将更新值，如果要修改立即生效，可以使用 `sync` 方法，sync 方法不接受参数，就是立即同步配置文件中的更新。

`value` 方法第一个参数是“键”，第二个参数是可选值，也就是如果没找到这个键，那么将会返回的值。一般最好还是写上，否则可能配置文件不在了，你就会发生读取错误。

其他方法还有：

contains 接受一个“键”，字符串对象，返回 bool 值，看看这个键是不是存在。

remove 接受一个“键“，移除该键。

allkeys 不接受参数，返回所有的“键“。

clear 不接受参数，清除所有的“键”。

群组管理

```
settings.setValue(”editor/wrapMargin”, 68)
```

如上例子所示“/”表示数据结构中的分组，如果有很多值都有相同的前缀，也就是同属一组，那么可以使用 beginGroup 方法和 endGroup 方法来管理。如下所示：

```
settings.beginGroup(”editor”)
settings.setValue(”wrapMargin”, 68)
settings.engGroup()
```

布局管理

布局管理是 GUI 设计中不可回避的一个话题，这里详细讨论下 pyqt 的布局管理。正如前所述及，pyqt5 用于布局管理的类都移到了 QtWidgets 子模块那里了，首先是最基本的 `QHBoxLayout` 和 `QVBoxLayout` 。

QBoxLayout

`QHBoxLayout` 和 `QVBoxLayout` 一个是横向排布，一个是竖向排布。它们的使用方法如下所示：

```
mainLayout=QHBoxLayout()
mainLayout.addWidget(button1)
mainLayout.addWidget(button2)
self.setLayout(mainLayout)
```

Layout 对象就好像一个封装器，Layout 里面还可以有 Layout，当然还有其他一些窗体子单元，都通过 `addWidget` 方法来确立封装关系。最后主母窗口主要接受一个 Layout 对象，使用的是 `setLayout` 方法。

```
from PyQt5.QtWidgets import QApplication,QWidget,QLabel,QVBoxLayout,QPushButton,QLineEdit, QMes

class Form(QWidget):
    def __init__(self):
```

```
        super().__init__()
        nameLabel = QLabel("Name:")
        self.nameLine = QLineEdit()
        self.submitButton = QPushButton("Submit")
        bodyLayout = QVBoxLayout()
        bodyLayout.addWidget(nameLabel)
        bodyLayout.addWidget(self.nameLine)
        bodyLayout.addWidget(self.submitButton)

        self.submitButton.clicked.connect(self.submit)

        self.setLayout(bodyLayout)
        self.setWindowTitle("Hello Qt")
        self.show()

    def submit(self):
        name = self.nameLine.text()

        if name == "":
            QMessageBox.information(self, "Empty Field",
                                    "Please enter a name.")
            return
        else:
            QMessageBox.information(self, "Success!",
                                    "Hello %s!" % name)

if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    screen = Form()
    sys.exit(app.exec_())
```

pyqt4 版本就是把头引入语句改成

```
from PyQt4.QtGui import ...
```

addStretch 方法

插入一个分隔符，也就是设计器里面的弹簧。

QGridLayout

在 tkinter 中有个 grid 方法，也就是网格布局，同样 pyqt 中也有个网格布局对象 QGridLayout。QGridLayout 的用法和上面 QBoxLayout 类似，除了 **addWidget** 方法后面还可以接受两个额外的参数表示几行几列。

请看到下面的例子。这个例子很好地演示了 QGridLayout 的使用。其中 `(i-1)//3` 即该数对 3 取商，本来的 1 2 3 4 5 6...将变成 0 0 0 1 1 1 2 2 2...正好对应网格中的几行，而 `(i-1)%3` 即该数对 3 取余，本来的 1 2 3 4 5 6...将变成 0 1 2 0 1 2 0 1 2...正好对应网格中的几列的概念。

```
from PyQt5.QtWidgets import QApplication,QWidget,QPushButton,QGridLayout

class Form(QWidget):
    def __init__(self):
        super().__init__()
        bodyLayout = QGridLayout()
        for i in range(1,10):
            button = QPushButton(str(i))
            bodyLayout.addWidget(button,(i-1)//3,(i-1)%3)
            print(i,(i-1)//3,(i-1)%3)
        self.setLayout(bodyLayout)
        self.setWindowTitle("the grid layout")
        self.show()


if __name__ == '__main__':
```

```
    import sys
    app = QApplication(sys.argv)
    screen = Form()
    sys.exit(app.exec_())
```

QFormLayout

QFormLayout，表单布局，常用于提交某个配置信息的表单。

请看到下面的例子。这个例子来自 pyqt5 源码 examples 文件夹 layouts 文件夹里面的 basiclayouts.py 文件，做了简化主要用于演示表单布局。

```
from PyQt5.QtWidgets import (QApplication, QDialog,QDialogButtonBox, QFormLayout, QGroupBox,QLa


class Dialog(QDialog):
    def __init__(self):
        super().__init__()
        self.createFormGroupBox()
        buttonBox = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)
        buttonBox.accepted.connect(self.accept)
        buttonBox.rejected.connect(self.reject)
        mainLayout = QVBoxLayout()
        mainLayout.addWidget(self.formGroupBox)
        mainLayout.addWidget(buttonBox)
        self.setLayout(mainLayout)
        self.setWindowTitle("user info")

    def createFormGroupBox(self):
        self.formGroupBox = QGroupBox("your infomation")
        layout = QFormLayout()
        layout.addRow(QLabel("name:"), QLineEdit())
        layout.addRow("age:", QSpinBox())
```

```
        layout.addRow(QLabel("other infomation:"), QTextEdit())
        self.formGroupBox.setLayout(layout)


if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    dialog = Dialog()
    sys.exit(dialog.exec_())
```

这里 QDialog 类和 QDialogButtonBox 类我们且不去管他，QDialog 类和下面的 accept 和 reject 方法有关，而 QDialogButtonBox 和最下面的两个按钮和绑定的喜好 accepted 和 rejected 有关。

然后我们看到下面创建表单的那个函数，其中 QGroupBox 也是一个窗体类型，带有标题。接下来就是 QFormLayout 表单布局的核心代码：

```
layout = QFormLayout()
layout.addRow(QLabel("name:"), QLineEdit())
layout.addRow(QLabel("age:"), QSpinBox())
layout.addRow(QLabel("other infomation:"), QTextEdit())
self.formGroupBox.setLayout(layout)
```

我们看到前面的 layout 的创建和后面母窗体使用本 layout 的 setLayout 方法和前面两个布局都是类似的，除了表单布局是一行行的，它的方法不是 addWidget，而是 addRow ，然后 addRow 方法严格意义上可以接受两个窗体类型（包括 layout 类型），另外第一个参数还可以是字符串，即显示的文字 。

快捷键和 Tab 键管理

什么是伙伴关系

一般是通过 QLabel 的 setBuddy 方法来关联某个输入窗体。然后 QLabel 有一个快捷键，当你按下这个快捷键，输入焦点就会转到这个 QLabel 对应的伙伴输入窗体

上。

快捷键

QShortcut 类

文本前用 & 会引入对应的 Alt+w 之类的快捷键。

然后 QAction 在初始化的时候有

然后 QAction 有方法

QKeySequence

`QKeySequence` 类在 pyqt4 和 pyqt5 中来自 QtGui 子模块，是快捷键的解决方案。比如可以直接引用 `QKeySequence.Open` 来表示快捷键 Ctrl+O。可用的构造函数如下所示：

```
QKeySequence(QKeySequence.Print)
QKeySequence(tr("Ctrl+P"))
QKeySequence(tr("Ctrl+p"))
QKeySequence(Qt.CTRL + Qt.Key_P)
```

我不太喜欢第一种表达方式，不是任何软件都有打印操作，况且打印和某个快捷键之间并没有逻辑联系，只有程序员的个人使用经验，这是不小的记忆负担。我比较喜欢第四种写法，看上去意义更加清晰，Qt 来自 QtCore 子模块。

字母按键就是类似 `Qt.Key_W` 这样的形式，Shift 按键是 Qt.SHIFT，Meta 按键是 Qt.META，CTRL 按键是 `Qt.CTRL` ，ALT 按键是 `Qt.ALT` 。

国际化支持

本小节参考资料除了官方文档之外还有这个网站 。

这里指的 pyqt 的软件国际化支持主要是指 i18n，也就是两种语言，英语和本土语言。其中软件的字符串都是英语，然后用 `self.tr()` 封装。

然后在你的项目里新建一个 translations 文件夹，新建如下一个小文件 `wise.pro`，这里的 wise 是你的模块具体的名字，随意修改之。这个文件的内容简要如下：

```
SOURCES += ../main.py  ../__init__.py \
                ../Widgets/__init__.py

TRANSLATIONS += wise_zh_CN.ts
```

SOURCES 是你希望扫描的 py 文件，如果该文件有前面所说的 self.tr() 封装，那么里面的字符串 pylupdate5 工具就可以扫描出来。这里支持路径的相对表达。但是不支持 glob 语法。

第二个变量就是 TRANSLATIONS 就是你希望生成的目标翻译 ts 文件的文件名，一般是如下格式：

```
{PROJECT_NAME}_{QLocale.system().name()}.ts
```

其中 PROJECT_NAME 是你项目的名字，而 QLocale.system().name() 是你当前机器所用的目标语言简写，你可以在 python3 的 eval 模式下查看一下：

```
>>> from PyQt5.QtCore import QLocale
>>> QLocale.system().name()
'zh_CN'
```

然后你需要用 pylupdate5 小工具处理该 pro 文件：

```
pylupdate5 wise.pro
```

这样你就可以看到生成的 `wise_zh_CH.ts` 文件了，然后请确保安装了 qt4-dev-tools，

```
sudo apt-get install qt4-dev-tools
```

这样你就可以双击打开 ts 文件，操作很简单，看见对应的英文单词，然后填上相应的中文解释。操作玩了点击发布，即看到生成的 qm 文件，或者使用命令行工具 **lrelease** 。

使用翻译文件

样例如下：

```
from PyQt5.QtCore import QTranslator,QLocale
    myapp = QApplication(sys.argv)
    translator = QTranslator()
    if translator.load('wise_'+ QLocale.system().name()+'.qm',
        ":/translations/"):
        myapp.installTranslator(translator)
```

首先你需要构建一个 QTranslator 对象，然后调用该对象的方法 load，这里第一个参数是要 load 的 qm 文件名，第二个参数是 qm 文件的路径，可以使用前面谈及的 qrc 引用路径。

最后你的主母窗口 myapp 使用 installTranslator 方法把这个 QTranslator 对象加进去即可。

使用 qt 官方翻译文件

有些 qt 窗体内部文字可能不好 DIY，这时需要如上一样加载 qt 的官方翻译文件。代码如下所示：

```
    translator_qt = QTranslator()
    if translator_qt.load('qt_'+ QLocale.system().name()+'.qm',":/translations/"):
#        print('i found qt')
        myapp.installTranslator(translator_qt)
```

这样你的主母窗口 myapp 现在需要加载两个翻译文件了。

官方 qt 翻译文件在 qt 源码的 translations 文件夹里面，可以如下通过 git clone 获取。

```
git clone https://gitorious.org/qt/qttranslations.git
```

ts 文件如前所述用 **lrelease** 命令处理以下，或者直接用语言工具打开然后发布即可。

附录

PyQt4 和 PyQt5 的区别整理

- 很多 pyqt4 原放在 QtGui 里面的一些 QWidget 在 pyqt5 里面都放入 QtWidgets 里面去了。一个简单的解决兼容性问题的方案就是在 pyqt4 里面引入的时候写上:

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

在 pyqt5 里面引入的时候写上:

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
```

引用信号发射对象

`sender` 方法来自 GObject，所以一般 Qt 里的窗体对象都可以用。其用法主要在槽里面，调用 `self.sender()` ，即返回一个发射该信号从而调用该槽的对象。

菜单栏看不见?

不过可能你会遇到麻烦，我就折腾了好久，因为菜单栏总是显示不出来，然后才发现是系统环境的问题，我在 GNOME 下看不到 pyqt5 做的软件的菜单栏了，但是到 Ubuntu 默认的 Unity 环境下最上面的面板就是菜单栏了，这个值得说一下。

如果你在 Unity 环境（Ubuntu14.04）下，那么不需要做什么，如果你在 gnome 或者 KDE 上，那么 qt5 的菜单栏可能会显示不出来，你需要删除下面这个小东西。

```
sudo apt-get remove appmenu-qt5
```

把这个小软件删除，pyqt5 上的菜单栏就能正常显示了，不过在 unity 环境下菜单栏不会显示在最上面的面板上了，而是常规的在图形 GUI 标题栏下面了。

参考资料

1. pyqt4 教程，PyQt4_Tutorial
2. Rapid GUI Programming with Python and Qt , 书籍的源码
3. C++-GUI-Programming-with-Qt-4-Second Edition 书籍的源码
4. PyQt4 各个类参考