Python Qt学习笔记

- Author：浅若清风cyf
- 参考教程：Python Qt 简介 | 白月黑羽 (byhy.net)
- 视频教程：Python Qt 图形界面编程 - PySide2 PyQt5 PyQt PySide*哔哩哔哩*bilibili

前言

- 为什么标题不是pyQt呢?
  - Qt库里面有非常强大的图形界面开发库，但是Qt库是C++语言开发的，PySide2、PyQt5可以让我们通过Python语言使用Qt。
  - PySide2 是Qt的 亲儿子， PyQt5 是Qt还没有亲儿子之前的收的 义子 （Riverbank Computing公司开发的）
  - PySide2于2018年7月发布。
  - pyQt5向pySide2迁移：通常修改导入包的名称即可
- python的GUI库：
  - Tkinter——Python官方采用的标准库
  - wxPython——基于wxWidgets的Python库
  - PySide2、PyQt5——基于Qt 的Python库（跨平台）【对应Qt5.15.2】（最新还有PySide6【对应Qt6】）
- 安装
  - 安装PySide2

```
pip install pyside2
# pip install pyside2 -i https://pypi.douban.com/simple/  # 国内镜像源
```

  - 安装pyQt5

```
pip install pyqt5-tools
```

入门

创建一个Hello World窗口（纯命令）

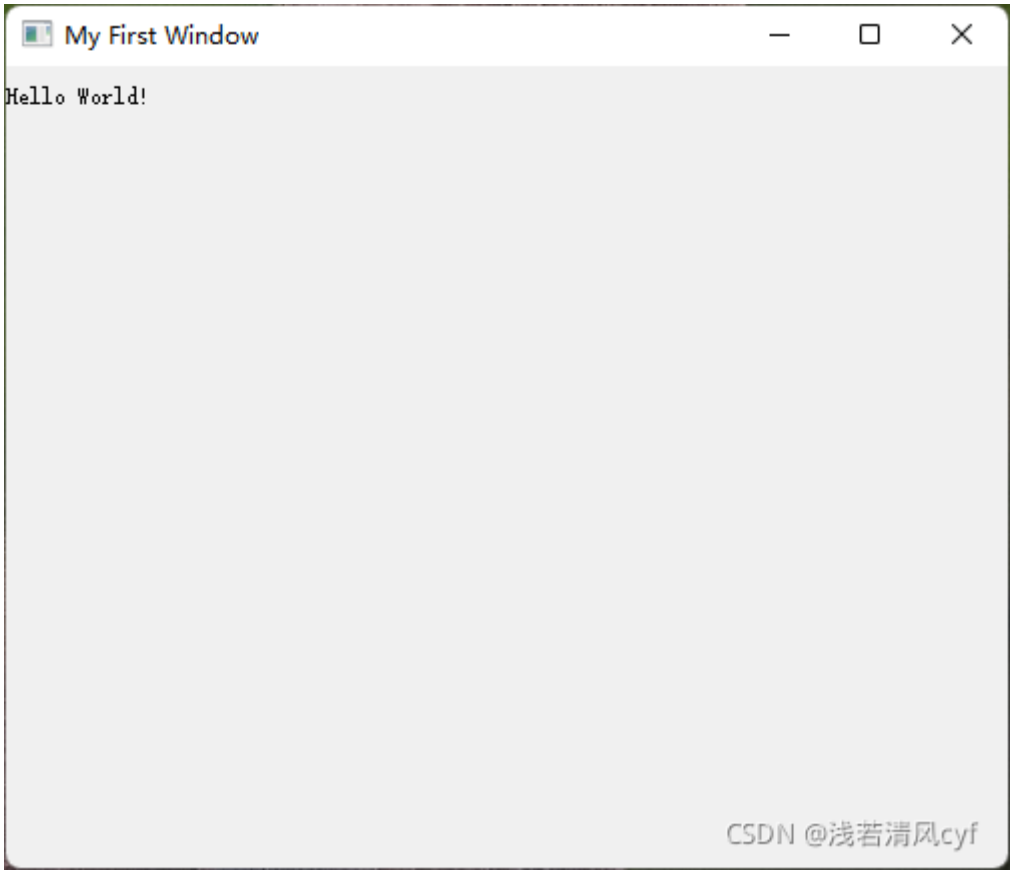
> 01_firstWindow.py

```
from PySide2.QtWidgets import QApplication, QMainWindow, QLabel

app = QApplication([])  # 创建Qt应用程序

window = QMainWindow()  # 初始化一个QMainWindow类型窗口
window.resize(500, 400)  # 窗口大小
```

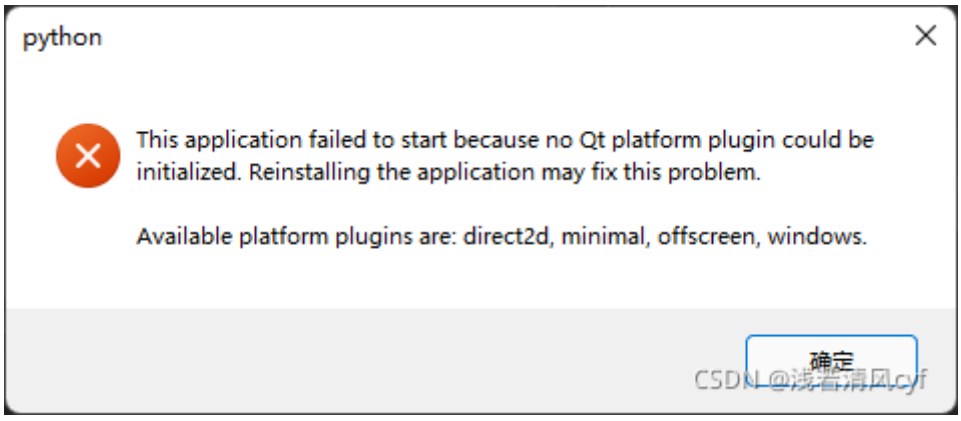
```
window.move(300, 310)  # 窗口移动到相对于屏幕的位置
window.setWindowTitle('My First Window')

label=QLabel(window)  # 创建一个标签控件，参数指定父窗口，即把该控件嵌入到window中
label.setText('Hello World!')

window.show()  # 设置主窗口显示

app.exec_()  # 启动Qt应用程序
```

> - QApplication 提供了整个图形界面程序的底层管理功能，比如：
>
> 初始化、程序入口参数的处理，用户事件（对界面的点击、输入、拖拽）分发给各个对应的控件...
>
> 必须在任何界面控件对象创建前，先创建它。

- 运行结果



- 出现问题：



- 解决：
  - 法1：添加以下代码，临时设置环境变量【推荐——在任何电脑都可用】

```
import os
import PySide2
dirname = os.path.dirname(PySide2.__file__)
plugin_path = os.path.join(dirname, 'plugins', 'platforms')
os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = plugin_path
```

- 法2：设置全局变量【不推荐——仅在自己的电脑生效，但无需在每段程序添加上述代码】

  在环境变量path中添加（根据自己的安装位置相应的修改路径）：`D:\DevEnv\Anaconda3\envs\py36_cv3\Lib\site-packages\PySide2\plugins\platforms`

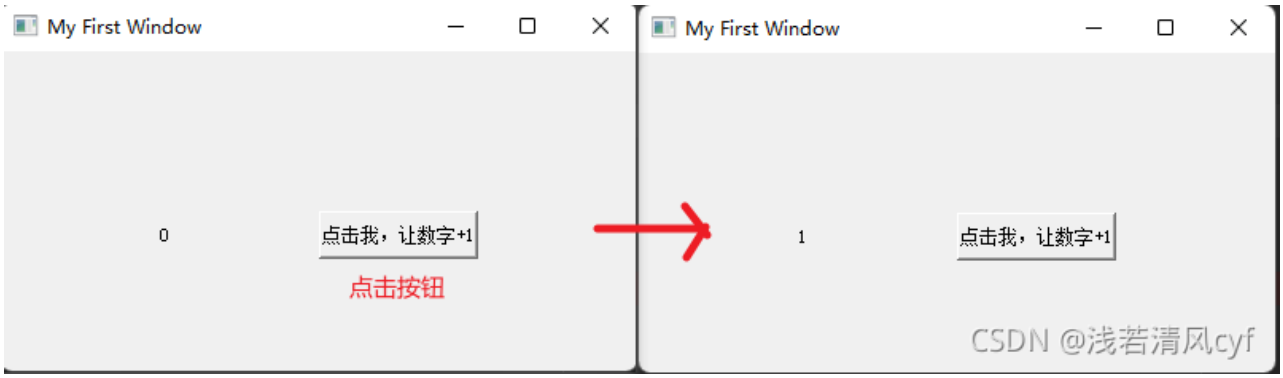
交互（信号Signal与槽Slot）

- 点击按钮，数字+1的简单交互程序

> 02_Interaction.py

```
from PySide2.QtWidgets import QApplication, QMainWindow, QLabel, QPushButton

import os
import PySide2

dirname = os.path.dirname(PySide2.__file__)
plugin_path = os.path.join(dirname, 'plugins', 'platforms')
os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = plugin_path

app = QApplication([])  # 创建Qt应用程序

window = QMainWindow()  # 初始化一个QMainWindow类型窗口
window.resize(400, 200)
window.move(300, 310)
window.setWindowTitle('My First Window')

count = 0

label = QLabel(window)  # 参数指定父窗口
label.setText(str(count))
label.move(100, 100)  # 位置相对于父组件移动(右移，下移)个像素

btn = QPushButton(window)
btn.setText('点击我，让数字+1')
btn.move(200, 100)


# 按钮点击事件的信号与槽函数绑定，实现交互
def countAdd():
    global count
    count += 1
    label.setText(str(count))


btn.clicked.connect(countAdd)  # 参数为函数对象

window.show()  # 设置主窗口显示

app.exec_()  # 启动Qt应用程序

```

- 运 行 结 果 ：



封装成一个类

> 03_packageToClass.py

```python
from PySide2.QtWidgets import QApplication, QMainWindow, QLabel, QPushButton

import os
import PySide2

dirname = os.path.dirname(PySide2.__file__)
plugin_path = os.path.join(dirname, 'plugins', 'platforms')
os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = plugin_path


# 将GUI（包括主窗口及控件）以及用到的变量封装成类，程序运行时启动一个QApplication对象，然后创建GUI对象，设置主窗口显示，最后启动QApplication对象
class MyFirstWin():
    # 初始化函数：GUI，成员变量，信号与槽的绑定等
    def __init__(self):
        # 成员变量
        self.count = 0

        # 主窗口
        self.window = QMainWindow()
        self.window.resize(400, 200)
        self.window.move(300, 310)
        self.window.setWindowTitle('My First Window')

        # 控件（嵌入主窗口）
        self.label = QLabel(self.window)  # 参数指定父窗口
        self.label.setText(str(self.count))
        self.label.move(100, 100)  # 位置相对于父组件移动(右移，下移)个像素

        self.btn = QPushButton(self.window)
        self.btn.setText('点击我，让数字+1')
        self.btn.move(200, 100)

        # 信号与槽的绑定
        self.btn.clicked.connect(self.countAdd)  # 按钮点击事件的信号与槽函数绑定，实现交互

    # 槽函数作为成员函数
    def countAdd(self):
        self.count += 1
```

```
        self.label.setText(str(self.count))


if __name__ == '__main__':
    app = QApplication([])  # 创建Qt应用程序

    myApp = MyFirstWin()  # 创建GUI
    myApp.window.show()  # 设置主窗口显示

    app.exec_()  # 启动Qt应用程序
```

使用Qt Designer 设计UI界面（推荐）

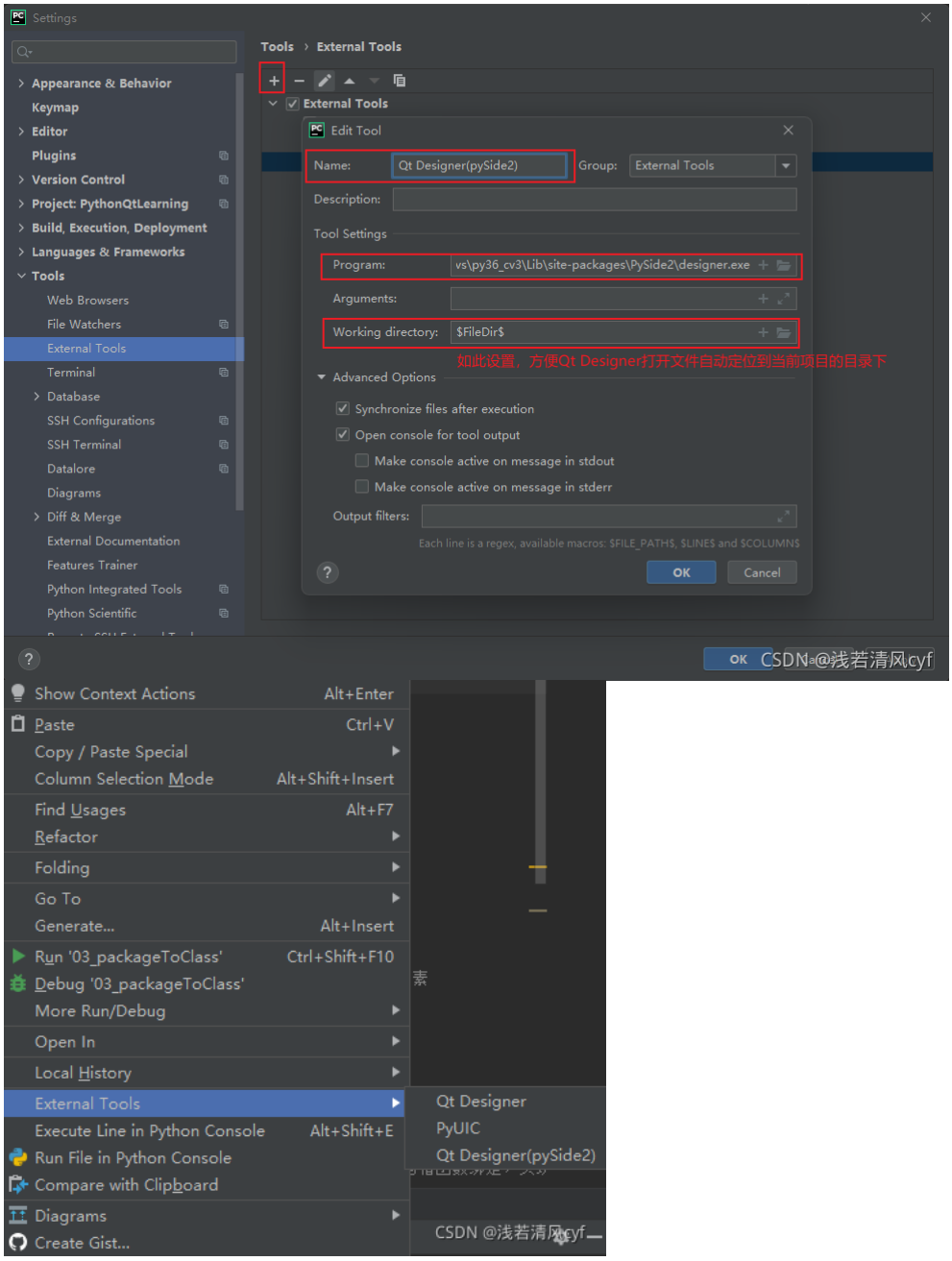
> 使用Qt Designer的优点：高效、所见所得、布局自适应

程序路径

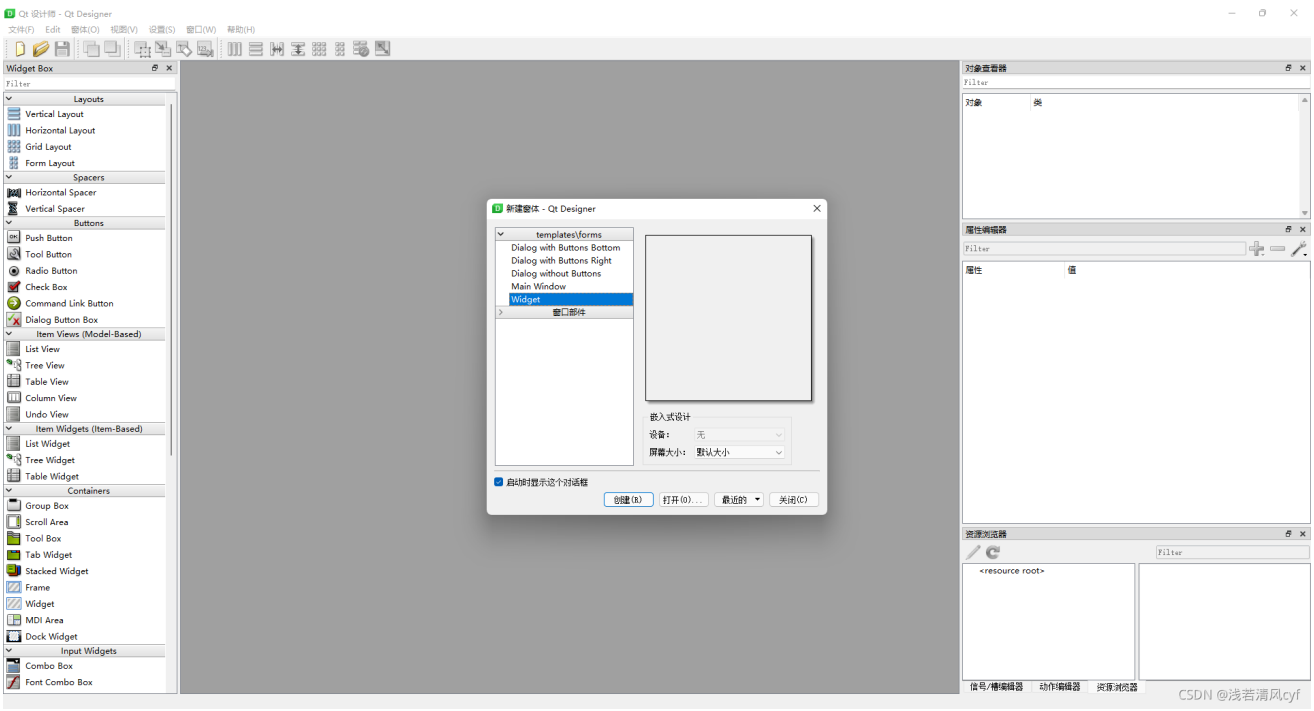
- `D:\DevEnv\Anaconda3\envs\py36_cv3\Lib\site-packages\PySide2\designer.exe`

添加到pycharm作为扩展工具（方便后续启动）

- 配置与使用：

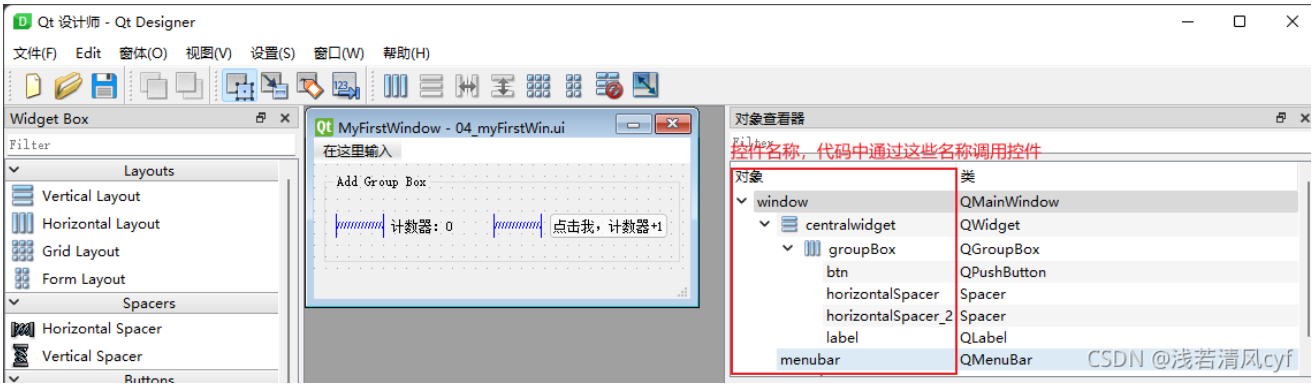

Settings
Tools › External Tools
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Project: PythonQtLearning
Build, Execution, Deployment
Languages & Frameworks
Tools
Web Browsers
File Watchers
External Tools
Terminal
Database
SSH Configurations
SSH Terminal
Datalore
Diagrams
Diff & Merge
External Documentation
Features Trainer
Python Integrated Tools
Python Scientific
External Tools
Edit Tool
Name: Qt Designer(pySide2)
Group: External Tools
Description:
Tool Settings
Program: vs\py36_cv3\Lib\site-packages\PySide2\designer.exe
Arguments:
Working directory: $FileDir$
如此设置，方便Qt Designer打开文件自动定位到当前项目的目录下
Advanced Options
Synchronize files after execution
Open console for tool output
Make console active on message in stdout
Make console active on message in stderr
Output filters:
Each line is a regex, available macros: $FILE_PATH$, $LINE$ and $COLUMN$
OK
Cancel
OK
CSDN@浅若清风cyf
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Go To
Generate... Alt+Insert
Run '03_packageToClass' Ctrl+Shift+F10
Debug '03_packageToClass'
More Run/Debug
Open In
Local History
External Tools
Execute Line in Python Console Alt+Shift+E
Run File in Python Console
Compare with Clipboard
Diagrams
Create Gist...
Qt Designer
PyUIC
Qt Designer(pySide2)
CSDN @浅若清风cyf

程序界面



绘制第一个UI界面

> 保存后生成一个ui文件（xml代码），代码中可加载ui直接创建窗口，无需通过代码设置UI样式



ui文件的使用

- UI文件有两种使用方法：动态加载、转换为py文件
- 那么我们该使用哪种方式比较好呢？动态加载还是转化为Python代码？
  - 白月黑羽建议：通常采用**动态加载比较方便**，因为改动界面后，不需要转化，直接运行，特别方便。
  - 但是，如果 你的**程序里面有非qt designer提供的控件**， 这时候，需要在代码里面加上一些额外的声明，而且 可能还会有奇怪的问题。往往就 要采用 转化Python代码的方法。

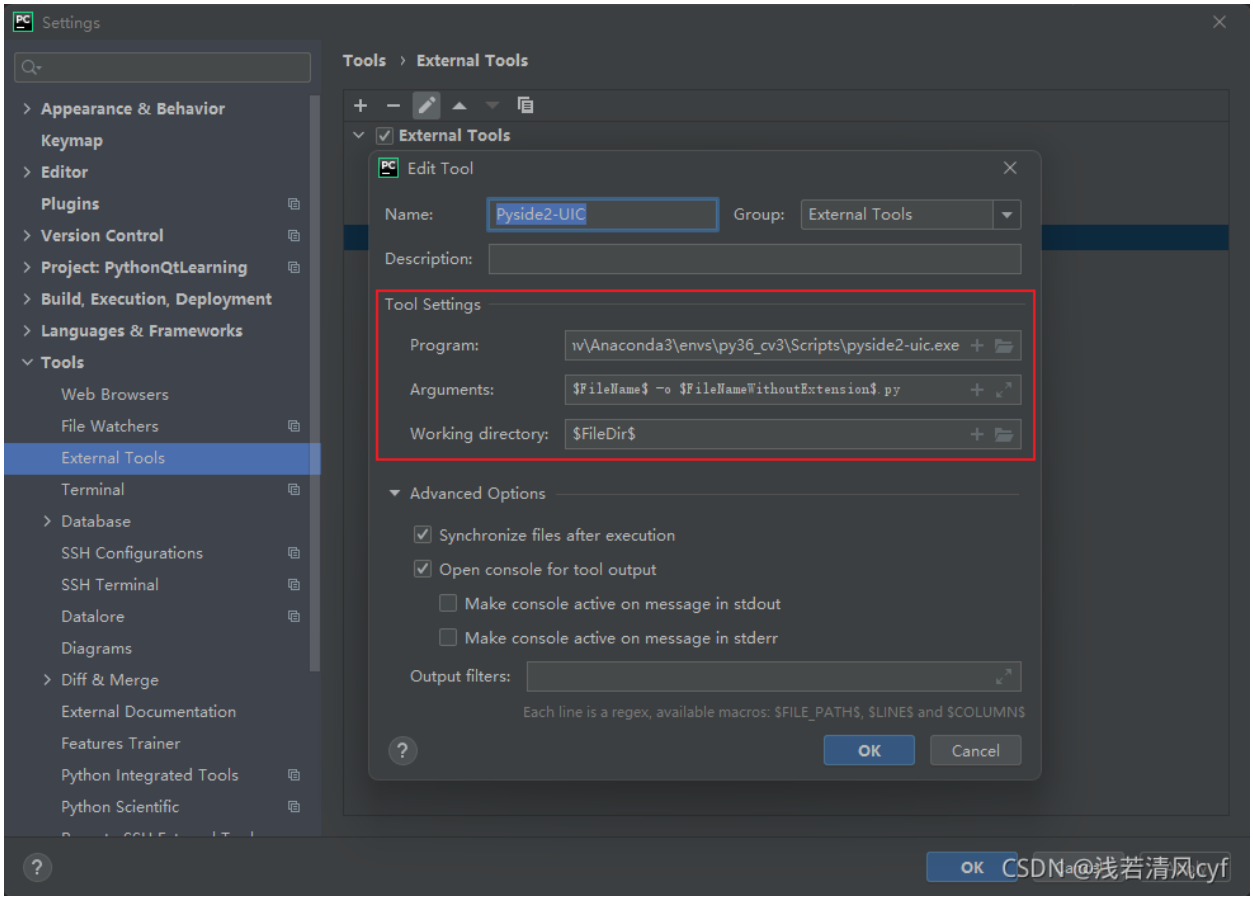
法1：动态加载

> 04_myFirstWinByQtDesigner_UI.py
>
> 对比未使用Qt Designer：将之前在初始化函数中定义窗口和控件简化为加载ui文件，代码：`self.ui = QUiLoader().load('04_myFirstWin.ui')`

```python
from PySide2.QtWidgets import QApplication
from PySide2.QtUiTools import QUiLoader  # --------注意：引入ui文件加载模块--------

import os
import PySide2

dirname = os.path.dirname(PySide2.__file__)
plugin_path = os.path.join(dirname, 'plugins', 'platforms')
os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = plugin_path


# 将GUI（包括主窗口及控件）以及用到的变量封装成类，程序运行时启动一个QApplication对象，然后创建GUI对象，设置主窗口显示，最后启动QApplication对象
class MyFirstWin():
    # 初始化函数：GUI，成员变量，信号与槽的绑定等
    def __init__(self):
        # 成员变量
        self.count = 0

        # 从文件中加载UI定义，创建一个相应的窗口对象
        # 注：里面的控件对象也成为窗口对象的属性
        self.ui = QUiLoader().load('04_myFirstWin.ui')  # UI中定义的控件可通过self.ui.XXX进行访问  # --------注意：加载ui文件--------

        # 信号与槽的绑定
        self.ui.btn.clicked.connect(self.countAdd)  # 按钮点击事件的信号与槽函数绑定，实现交互
# --------注意：通过self.ui访问控件--------

    # 槽函数作为成员函数
    def countAdd(self):
        self.count += 1
        self.ui.label.setText('计数器：'+str(self.count))


if __name__ == '__main__':
    app = QApplication([])  # 创建Qt应用程序

    myApp = MyFirstWin()  # 创建GUI
    myApp.ui.show()  # 设置主窗口显示（即：让ui定义的窗口显示）  # --------注意：设置ui显示----------

    app.exec_()  # 启动Qt应用程序

```

法2：UI文件转化为Python代码

- 命令：`pyside2-uic main.ui > ui_main.py`
- 将命令执行配置为pycharm扩展工具（后续点击扩展工具自动执行命令转换为py文件）
  - **Program** 填写：`D:\DevEnv\Anaconda3\envs\py36_cv3\Scripts\pyside2-uic.exe`
  - **Arguments** 填写：`$FileName$ -o $FileNameWithoutExtension$.py`
  - **Working directory** 填写：`$FileDir$`



- 使用

> 04_myFirstWinByQtDesigner_Py.py

```python
from PySide2.QtWidgets import QApplication,QMainWindow
from _04_myFirstWin import Ui_window

import os
import PySide2

dirname = os.path.dirname(PySide2.__file__)
plugin_path = os.path.join(dirname, 'plugins', 'platforms')
os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = plugin_path


# 将GUI（包括主窗口及控件）以及用到的变量封装成类，程序运行时启动一个QApplication对象，然后创建GUI对象，设置主窗口显示，最后启动QApplication对象
class MyFirstWin(QMainWindow):  # --------注意：继承父类--------
    # 初始化函数：GUI，成员变量，信号与槽的绑定等
    def __init__(self):
        # 初始化父类
        super().__init__()  # --------注意：先初始化父类--------

        # 成员变量
        self.count = 0

        # 从文件中加载UI定义，创建一个相应的窗口对象
        # 注：里面的控件对象也成为窗口对象的属性
        self.ui = Ui_window()  # --------注意：创建UI对象--------

        # 初始化界面
        self.ui.setupUi(self)  # --------注意：调用UI对象的初始化函数--------

        # 信号与槽的绑定
```

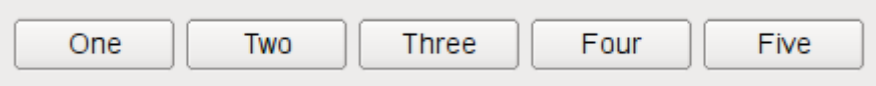
```
30          self.ui.btn.clicked.connect(self.countAdd)  # 按钮点击事件的信号与槽函数绑定，实现交互
31
32      # 槽函数作为成员函数
33      def countAdd(self):
34          self.count += 1
35          self.ui.label.setText('计数器：'+str(self.count))
36
37
38  if __name__ == '__main__':
39      app = QApplication([])  # 创建Qt应用程序
40
41      myApp = MyFirstWin()  # 创建GUI
42      myApp.show()  # 设置主窗口显示（即：让ui定义的窗口显示）  # --------注意--------
43
44      app.exec_()  # 启动Qt应用程序
```

布局

4种Layout布局

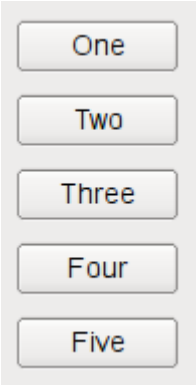
QHBoxLayout 水平布局

> 控件从左到右 水平横着摆放



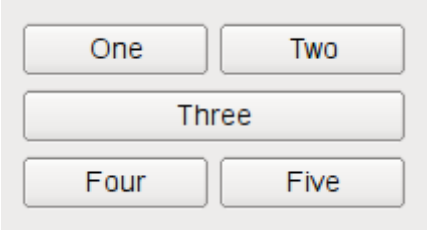
QVBoxLayout 垂直布局

> 控件从上到下竖着摆放



QGridLayout 表格布局

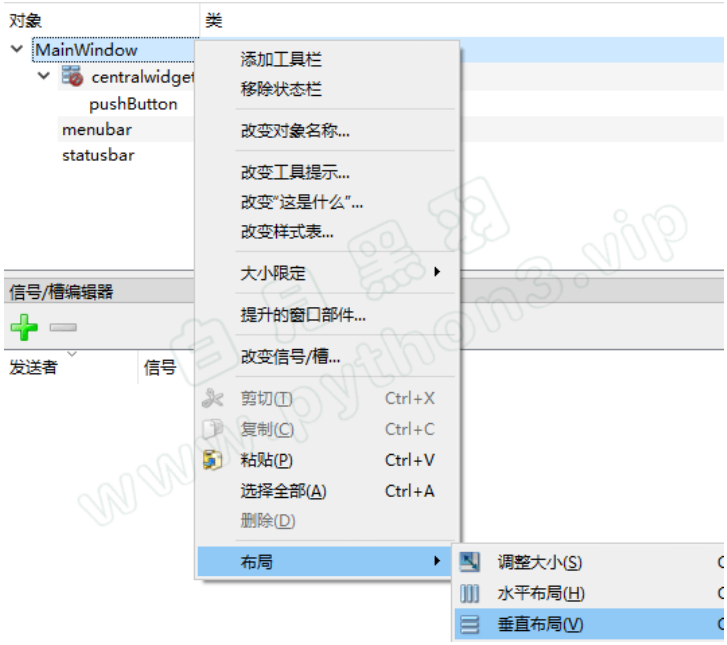
> 多个控件 格子状摆放，有的控件可以占据多个格子



QFormLayout 表单布局

> 就像一个只有两列的表格，非常适合填写注册表单这种类型的界面

布局设置



- 常用的属性
  - layoutStretch：设置已设置布局的控件的大小比例

    > Tip：控件间添加空白区域可配合弹簧使用

界面布局步骤建议（☆☆☆）

- 先不使用任何Layout，把所有控件 按位置 摆放在界面上
- 然后先从 **最内层开始** 进行控件的 Layout 设定
- **逐步拓展到外层** 进行控件的 Layout设定
- 最后调整 layout中控件的大小比例， 优先使用 Layout的 **layoutStrentch 属性**来控制

常用控件

按钮

- 信号：被点击
- 方法：改变文本
- 方法：禁用、启用

单行文本框

- 信号：文本被修改
- 信号：按下回车键
- 方法：获取文本
- 方法：设置提示
- 方法：设置文本
- 方法：清除所有文本
- 方法：拷贝文本到剪贴板
- 方法：粘贴剪贴板文本

多行纯文本框

- 信号：文本被修改
- 信号：光标位置改变
- 方法：获取文本
- 方法：获取选中文本
- 方法：设置提示
- 方法：设置文本
- 方法：在末尾添加文本
- 方法：在光标处插入文本
- 方法：清除所有文本
- 方法：拷贝文本到剪贴板
- 方法：粘贴剪贴板文本

文本浏览框

- 方法：在末尾添加文本
- 方法：在光标处插入文本

标签

- 方法：改变文本
- 显示图片

组合选择框

- 信号：选项改变
- 方法：添加一个选项
- 方法：添加多个选项
- 方法：清空选项
- 方法：获取当前选项文本

列表

- 方法：添加一个选项
- 方法：添加多个选项
- 方法：删除一个选项
- 方法：清空选项
- 方法：获取当前选项文本

表格

- 创建列 和 标题栏
- 方法：插入一行、删除一行
- 方法：设置单元格文本内容
- 方法：获取单元格文本的内容
- 方法：获取所有行数、列数
- 方法：获取当前选中是第几行
- 方法：设置表格行数、列数
- 方法：清除/删除所有内容
- 方法：设定列宽、宽度自动缩放
- 信号：单元格内容改动
- 实战练习

单选按钮 和 按钮组

- 说明
- 信号：选中状态改变

勾选按钮 和 按钮组

- 说明
- 信号：选中状态改变

tab页控件

- tab页中布局Layout

进度条

- 说明

数字输入框

- 获取数字
- 方法：设置数字

日期控件

- 获取日期

选择文件框

- 选择目录
- 选择单个文件
- 选择多个文件

树控件

提示框

输入对话框

菜单

工具栏

状态栏

剪贴板

MDI 多个子窗口

窗口跳转

- 从一个窗口跳转到另外一个窗口：实例化另外一个窗口，显示新窗口，关闭老窗口。

非模式对话框

```
from PySide2 import QtWidgets
import sys

class Window2(QtWidgets.QMainWindow):  # 窗口2

    def __init__(self):
        super().__init__()
        self.setWindowTitle('窗口2')

        centralWidget = QtWidgets.QWidget()
        self.setCentralWidget(centralWidget)

        button = QtWidgets.QPushButton('按钮2')

        grid = QtWidgets.QGridLayout(centralWidget)
        grid.addWidget(button)


class MainWindow(QtWidgets.QMainWindow):  # 窗口1（窗口1按钮点击事件绑定显示窗口2的槽函数）
    def __init__(self):
        super().__init__()
        self.setWindowTitle('窗口1')
```

```
23
24          centralWidget = QtWidgets.QWidget()
25          self.setCentralWidget(centralWidget)
26
27          button = QtWidgets.QPushButton('打开新窗口')
28          button.clicked.connect(self.open_new_window)  # --------------------!!!-------------
    -------
29
30          grid = QtWidgets.QGridLayout(centralWidget)
31          grid.addWidget(button)
32
33      def open_new_window(self):  # --------------------!!!--------------------
34          # 实例化另外一个窗口
35          self.window2 = Window2()  # 实例化新窗口
36          # 显示新窗口
37          self.window2.show()   # --------------------非模式对话框调用show()方法----------------
    ----
38          # 关闭自己
39          self.close()  # 关闭自己
40
41  if __name__ == '__main__':
42      app = QtWidgets.QApplication(sys.argv)
43      window = MainWindow()
44      window.show()
45      sys.exit(app.exec_())
```

模式对话框

> 原窗口调用模式对话框会阻止原窗口的进行执行，直到模式对话框关闭后才能继续执行

```
1   from PySide2 import QtWidgets
2   import sys
3
4   class MyDialog(QtWidgets.QDialog):
5       def __init__(self):
6           super().__init__()
7           self.setWindowTitle('模式对话框')
8
9           self.resize(500, 400)
10          self.textEdit = QtWidgets.QPlainTextEdit(self)
11          self.textEdit.setPlaceholderText("请输入薪资表")
12          self.textEdit.move(10, 25)
13          self.textEdit.resize(300, 350)
14
15          self.button = QtWidgets.QPushButton('统计', self)
16          self.button.move(380, 80)
17
18  class MainWindow(QtWidgets.QMainWindow):
19      def __init__(self):
20          super().__init__()
21          self.setWindowTitle('主窗口')
22
23          centralWidget = QtWidgets.QWidget()
24          self.setCentralWidget(centralWidget)
25
26          button = QtWidgets.QPushButton('打开模式对话框')
```

```
        button.clicked.connect(self.open_new_window)  # --------------------!!!-------------------

        grid = QtWidgets.QGridLayout(centralWidget)
        grid.addWidget(button)

    def open_new_window(self):  # --------------------!!!--------------------
        # 实例化一个对话框类
        self.dlg = MyDialog()
        # 显示对话框，代码阻塞在这里，
        # 等待对话框关闭后，才能继续往后执行
        self.dlg.exec_()  # --------------------模式对话框调用exec_()方法--------------------

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

创建子线程 与 自定义信号

> 对于一些响应时间长的操作（如HTTP请求等），可以采用创建python子线程的方法进行处理，避免阻塞主窗口

创建子线程

```
    def sendRequest(self):

        method = self.ui.boxMethod.currentText()
        url    = self.ui.editUrl.text()
        payload = self.ui.editBody.toPlainText()

        # 获取消息头
        headers = {}
        # 此处省略一些对消息头的处理

        req = requests.Request(method,
                               url,
                               headers=headers,
                               data=payload
                               )

        prepared = req.prepare()

        self.pretty_print_request(prepared)
        s = requests.Session()

        # 创建新的线程去执行发送方法，
        # 服务器慢，只会在新线程中阻塞
        # 不影响主线程  # --------------------!!!--------------------
        thread = Thread(target = self.threadSend,  # 调用的函数
                        args= (s, prepared)  # 调用函数的参数
                        )
        thread.start()  # 子线程执行
```

```
29
30      # 新线程入口函数
31      def threadSend(self,s,prepared):  # 定义子线程需要执行的函数
32
33          try:
34              r = s.send(prepared)
35              self.pretty_print_response(r)
36              except:
37                  self.ui.outputWindow.append(
38                      traceback.format_exc())
```

自定义信号与槽

> **背景：**在另外一个线程直接操作界面，可能会导致意想不到的问题，比如：输出显示不全，甚至程序崩溃。但是，我们确实经常需要在子线程中 更新界面。比如子线程是个爬虫，爬取到数据显示在界面上。怎么办呢?
>
> **解决方法**：让子线程发送自定义信号给主线程，让主线程调用自定义的槽函数完成GUI的更新

- 子线程：发送信号：emiit()方法
- 主线程：定义处理Signal信号的方法

```
1   from PySide2.QtWidgets import QApplication, QTextBrowser
2   from PySide2.QtUiTools import QUiLoader
3   from threading import Thread
4
5   from PySide2.QtCore import Signal,QObject  # ----------------------!!!----------------------
6
7   # 自定义信号源对象类型，一定要继承自 QObject
8   class MySignals(QObject):  # -------自定义信号的类（所有自定义信号都封装在这个类里面）---------
9
10      # 定义一种信号，两个参数 类型分别是： QTextBrowser 和 字符串
11      # 调用 emit方法 发信号时，传入参数 必须是这里指定的 参数类型（因为底层是基于C++开发的）
12      text_print = Signal(QTextBrowser,str)  # -------发射信号是传递一个QTextBrowser控件对象给槽函数，槽函数可对这个控件执行一些操作---------
13
14      # 还可以定义其他种类的信号
15      update_table = Signal(str)
16
17  # 实例化
18  global_ms = MySignals()    # --------实例化自定义信号，之后UI的类可以调用这个对象绑定自身定义的槽函数---------
19
20  class Stats:
21
22      def __init__(self):
23          self.ui = QUiLoader().load('main.ui')
24
25          # 自定义信号的处理函数
26          global_ms.text_print.connect(self.printToGui)  # ---自定义信号text_print绑定处理函数printToGui：global_ms为自定义信号类的实例化对象---
27
28
29      def printToGui(self,fb,text):  # -------自定义信号的处理方法（槽函数）-------
30          fb.append(str(text))
31          fb.ensureCursorVisible()
32
```

```
33      def task1(self):  # --------主窗口中定义的子线程方法，子线程会调用emit()方法发射自定义信号---
        ------
34          def threadFunc():
35              # 通过Signal 的 emit 触发执行 主线程里面的处理函数
36              # emit参数和定义Signal的数量、类型必须一致
37              # -----实际上就是：子线程指定需要更新哪个控件的内容，并把内容通过信号发送给主线程，让主
        线程完成GUI的更新--------
38              global_ms.text_print.emit(self.ui.infoBox1, '输出内容')  # --------第一个参数传递
        主窗口的QTextBrowser类型的控件给槽函数处理--------
39
40          thread = Threading.Thread(target = threadFunc )
41          thread.start()
42
43      def task2(self):
44          def threadFunc():
45              global_ms.text_print.emit(self.ui.infoBox2, '输出内容') # -----主窗口中定义的子线
        程方法，，子线程会调用emit()方法发射自定义信号-----
46
47          thread = Threading.Thread(target=threadFunc)
48          thread.start()
```

发布程序

- 使用PyInstaller制作独立可执行程序
- 安装：`pip install pyinstaller`
- 执行：`pyinstaller 程序入口.py  --noconsole --hidden-import PySide2.QtXml --icon="logo.ico"`

> `--noconsole` 指定不要命令行窗口，否则我们的程序运行的时候，还会多一个黑窗口。 但是我建议大家可以先去掉这个参数，等确定运行成功后，再加上参数重新制作exe。因为这个黑窗口可以显示出程序的报错，这样我们容易找到问题的线索。
>
> `--hidden-import PySide2.QtXml` 参数是因为这个 QtXml库是动态导入，PyInstaller没法分析出来，需要我们告诉它，

> 打包过程&注意事项：
>
> - PyInstaller是通过分析我们的代码里面的 `import` 语句，推断我们的程序需要哪些库的。
> - 但是有些代码，导入库的时候，是 动态导入 。
> - 所谓动态导入就是，写代码的时候并不确定要导入什么库，而是在运行的时候才知道。
> - 这种情况，不是用 import语句，而是用 `__import__` 或者 `exec` 、 `eval` 这样的方式，来导入库。
> - PyInstaller 没法分析出动态导入的库有哪些，我们可以通过命令行参数 `--hidden-import` 告诉它。