链滴

PyQt5 快速开发与实战（二）

作者：Ricky2020

原文链接：https://ld246.com/article/1612327806309

来源网站：链滴

3.4 信号与槽关联

1. 信号signal与slot 是Qt的核心机制。
2. 如何为控件发射的信号指定对应的处理槽函数?
    1. 在窗口的UI设计中添加信号和槽
    2. 通过代码连接信号和槽
    3. 通过Eric的"生成对话框代码"的功能产生信号和槽

3.4.1 简单入门

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName("Form")
        Form.resize(400, 300)
        self.closeWinBtn = QtWidgets.QPushButton(Form)
        self.closeWinBtn.setGeometry(QtCore.QRect(80, 120, 93, 28))
        self.closeWinBtn.setObjectName("closeWinBtn")

        self.retranslateUi(Form)
        self.closeWinBtn.clicked.connect(Form.close)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
        _translate = QtCore.QCoreApplication.translate
        Form.setWindowTitle(_translate("Form", "Form"))
        self.closeWinBtn.setText(_translate("Form", "关闭窗口"))
```

```
import sys
from PyQt5 import QtWidgets,QtCore
from untitled import *

class myFirstSignalSlot(QtWidgets.QWidget, Ui_Form):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

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

3.4.2 快速进阶

1. 新手的两大困惑

    1. PyQt默认有哪些信号与槽

操作使用Qt Designer右下角的signal/slot编辑

    2. 如何使用这些信号与槽

操作将有信号与槽的ui文件编译为py文件后，自行查看文件

2. 使用Eric6生成信号和槽

    1. 切换到Eric 的窗体选项卡
    2. 单击右键选择编译窗体，生成py文件
    3. 再去右键，选择"生成对话框代码"，弹出窗体代码生成器
    4. 在窗体代码生成器中，点击新建
    5. 然后选择相应的信号发射
    6. 进行代码的改写

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName("Form")
        Form.resize(400, 300)
        self.closeWinBtn = QtWidgets.QPushButton(Form)
        self.closeWinBtn.setGeometry(QtCore.QRect(120, 120, 93, 28))
        self.closeWinBtn.setObjectName("closeWinBtn")

        self.retranslateUi(Form)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
        _translate = QtCore.QCoreApplication.translate
        Form.setWindowTitle(_translate("Form", "Form"))
        self.closeWinBtn.setText(_translate("Form", "关闭窗口"))
```

```
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QWidget, QApplication

from Ui_Eric002 import Ui_Form


class Form(QWidget, Ui_Form):
    """
    Class documentation goes here.
    """
    def __init__(self, parent=None):
        """
        Constructor

        @param parent reference to the parent widget
        @type QWidget
        """
        super(Form, self).__init__(parent)
        self.setupUi(self)

    @pyqtSlot()
    def on_closeWinBtn_clicked(self):
        """
        Slot documentation goes here.
        """
        # TODO: not implemented yet
        self.close()

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

3.5 菜单栏与工具栏

3.5.1 界面设计

MainWindow即为主窗口，主要包含菜单栏。工具栏、任务栏

1. 对于一级菜单 可以使用&F来添加菜单的快捷键
2. 子菜单可以使用Qt Designer右下角的动作编辑器或者属性编辑器中的Shortcut来添加快捷键
3. 右键窗体，可以选择添加工具栏
4. 可以在属性编辑器中新建addWinAction来添加动作
5. 可以在动作编辑器中建立工具栏的图标等，然后拖入进工具栏
6. 将ui文件转换为py文件
7. 测试用例

3.5.2 测试效果

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(800, 600)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 26))
        self.menubar.setObjectName("menubar")
        self.menu_F = QtWidgets.QMenu(self.menubar)
        self.menu_F.setObjectName("menu_F")
        self.menu_E = QtWidgets.QMenu(self.menubar)
        self.menu_E.setObjectName("menu_E")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)
        self.toolBar = QtWidgets.QToolBar(MainWindow)
        self.toolBar.setObjectName("toolBar")
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.toolBar)
        MainWindow.insertToolBarBreak(self.toolBar)
        self.actionOpen = QtWidgets.QAction(MainWindow)
        self.actionOpen.setObjectName("actionOpen")
        self.actionNew = QtWidgets.QAction(MainWindow)
        self.actionNew.setObjectName("actionNew")
        self.actionClose = QtWidgets.QAction(MainWindow)
        self.actionClose.setObjectName("actionClose")
        self.addWinAction = QtWidgets.QAction(MainWindow)
        self.addWinAction.setObjectName("addWinAction")
        self.menu_F.addAction(self.actionOpen)
        self.menu_F.addAction(self.actionNew)
        self.menu_F.addAction(self.actionClose)
        self.menubar.addAction(self.menu_F.menuAction())
        self.menubar.addAction(self.menu_E.menuAction())
        self.toolBar.addAction(self.addWinAction)

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.menu_F.setTitle(_translate("MainWindow", "文件(&F)"))
        self.menu_E.setTitle(_translate("MainWindow", "编辑(&E)"))
        self.toolBar.setWindowTitle(_translate("MainWindow", "toolBar"))
        self.actionOpen.setText(_translate("MainWindow", "Open"))
        self.actionOpen.setShortcut(_translate("MainWindow", "Ctrl+O"))
        self.actionNew.setText(_translate("MainWindow", "New"))
        self.actionClose.setText(_translate("MainWindow", "Close"))
```

```
        self.addWinAction.setText(_translate("MainWindow", "添加窗体"))
        self.addWinAction.setToolTip(_translate("MainWindow", "添加窗体"))


from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog

from Ui_main import Ui_MainWindow


class MainWindow(QMainWindow, Ui_MainWindow):
    """
    Class documentation goes here.
    """
    def __init__(self, parent=None):
        """
        Constructor

        @param parent reference to the parent widget
        @type QWidget
        """
        super(MainWindow, self).__init__(parent)
        self.setupUi(self)

    @pyqtSlot()
    def on_actionOpen_triggered(self):
        """
        Slot documentation goes here.
        """
        # TODO: not implemented yet
        file, ok = QFileDialog.getOpenFileName(self, "打开", "D:/", "All Files(*);;Text Files(*.txt)")
        self.statusbar.showMessage(file)


if __name__ == "__main__":
    import sys
    app = QApplication(sys.argv)
    myWin = MainWindow()
    myWin.show()
    sys.exit(app.exec_())
```

3.5.3 应用：加载其他窗口

概念在当前窗口中嵌入另一个窗口

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName("Form")
        Form.resize(400, 300)
        self.textEdit = QtWidgets.QTextEdit(Form)
        self.textEdit.setGeometry(QtCore.QRect(30, 30, 231, 161))
```

```
        self.textEdit.setObjectName("textEdit")

        self.retranslateUi(Form)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
        _translate = QtCore.QCoreApplication.translate
        Form.setWindowTitle(_translate("Form", "Form"))
        self.textEdit.setHtml(_translate("Form", "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0//EN\" \"http://www.w3.org/TR/REC-html40/strict.dtd\">\n"
"<html><head><meta name=\"qrichtext\" content=\"1\" /><style type=\"text/css\">\n"
"p, li { white-space: pre-wrap; }\n"
"</style></head><body style=\" font-family:\'SimSun\'; font-size:9pt; font-weight:400; font-style:normal;\">\n"
"<p style=\" margin-top:0px; margin-bottom:0px; margin-left:0px; margin-right:0px; -qt-block-indent:0; text-indent:0px;\">我是子窗体的内容</p>\n"
"<p style=\"-qt-paragraph-type:empty; margin-top:0px; margin-bottom:0px; margin-left:0px; margin-right:0px; -qt-block-indent:0; text-indent:0px;\"><br /></p></body></html>"))


from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(800, 600)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.gridLayoutWidget = QtWidgets.QWidget(self.centralwidget)
        self.gridLayoutWidget.setGeometry(QtCore.QRect(109, 49, 351, 311))
        self.gridLayoutWidget.setObjectName("gridLayoutWidget")
        self.gridLayout = QtWidgets.QGridLayout(self.gridLayoutWidget)
        self.gridLayout.setContentsMargins(0, 0, 0, 0)
        self.gridLayout.setObjectName("gridLayout")
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 26))
        self.menubar.setObjectName("menubar")
        self.menu = QtWidgets.QMenu(self.menubar)
        self.menu.setObjectName("menu")
        self.menu_2 = QtWidgets.QMenu(self.menubar)
        self.menu_2.setObjectName("menu_2")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)
        self.toolBar = QtWidgets.QToolBar(MainWindow)
        self.toolBar.setObjectName("toolBar")
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.toolBar)
        self.AddSubAction = QtWidgets.QAction(MainWindow)
        self.AddSubAction.setObjectName("AddSubAction")
        self.menubar.addAction(self.menu.menuAction())
        self.menubar.addAction(self.menu_2.menuAction())
        self.toolBar.addAction(self.AddSubAction)
```

```
        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.menu.setTitle(_translate("MainWindow", "文件"))
        self.menu_2.setTitle(_translate("MainWindow", "编辑"))
        self.toolBar.setWindowTitle(_translate("MainWindow", "toolBar"))
        self.AddSubAction.setText(_translate("MainWindow", "添加子窗体"))
        self.AddSubAction.setToolTip(_translate("MainWindow", "添加子窗体"))


from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QMainWindow, QApplication, QWidget

from Ui_MainWidget import Ui_MainWindow
from Ui_subWidget import Ui_Form
class ChildForm(QWidget, Ui_Form):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

class MainWindow(QMainWindow, Ui_MainWindow):
    """
    Class documentation goes here.
    """
    def __init__(self, parent=None):
        """
        Constructor

        @param parent reference to the parent widget
        @type QWidget
        """
        super(MainWindow, self).__init__(parent)
        self.setupUi(self)
        self.child = ChildForm()

    @pyqtSlot()
    def on_AddSubAction_triggered(self):
        """
        Slot documentation goes here.
        """
        # TODO: not implemented yet
        self.gridLayout.addWidget(self.child)
        self.child.show()



if __name__ == "__main__":
    import sys
    app = QApplication(sys.argv)
    Form = MainWindow()
```

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

3.6 打包资源文件

使用PyQt5生成的应用程序引用图片资源的两大主要方法

1. 将资源文件转换为python文件，然后引用python文件

注意Qt Designer中不能直接加入图片图标资源，而是需要编写.qrc文件

1. 手动新建apprc.qrc文件

```
<rcc version="1.0">
<qresource>

</qresource>

</rcc>
```

2. 在Eric中创建资源文件
   - 切换到资源选项卡
   - 新建资源
3. 在程序中通过相对路径引用外部的图片资源

3.6.1 使用Qt Designer 加载资源文件

1. 新建一个资源文件.qrc
2. 打开Qt Designer 创建一个普通的窗体
3. 进入资源浏览器，打开资源编辑界面
4. 创建前缀是pic的图片文件，加载文件路径
5. 使用文本打开qrc文件，发现是XML文件包含了加载进去的图片资源的路径

3.6.2 在窗体中使用资源文件

1. 使用Qt Designer在窗体中放置控件
2. 更改控件的Pixmap值为资源文件
3. 将ui文件转换为py文件

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName("Form")
        Form.resize(400, 300)
        self.label = QtWidgets.QLabel(Form)
```

```
        self.label.setGeometry(QtCore.QRect(50, 90, 291, 141))
        self.label.setText("")
        self.label.setPixmap(QtGui.QPixmap(":/pic/PyQt5/Chapter03/images/cartoon1.ico"))
        self.label.setObjectName("label")

        self.retranslateUi(Form)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
        _translate = QtCore.QCoreApplication.translate
        Form.setWindowTitle(_translate("Form", "Form"))

import apprcc_rc
```

3.6.3 转换资源文件为python文件

1. 使用命令pyrcc5 -o pyfile.py rcc.qrc

2. 转换后的py文件中可以看出该文件已经使用了QtCore.qRegisterResourceData 进行了资源的初化注册，可以直接引用该文件了

注意资源文件的路径问题

- Qt Designer使用图片资源时，会在资源的引用路径之前加上':'(冒号)
- 此时的资源路径 = qresource这个标签的前缀pic + 源资源文件的路径

第四章 PyQt5 基本窗口控件

目标学会如何部署和调整控件

4.1 QMainWindow

QMainWindow主窗口为用户提供一个应用程序框架，有自己的布局，可以在布局中添加控件，比如具栏、菜单栏、状态栏

4.1.1 窗口类型介绍

1. QMainWindow、Qwidget、QDialog三个类都是用来创建窗口的，可以直接使用，也可以继承之再去使用

2. QMainWindow包含菜单栏、工具栏、状态栏、标题栏。是最常见的窗口形式

3. QDialog 是对话框窗口的基类。对话框主要用来执行短期任务，或者与用户进行交互，包括模态和模态对话框两种，没有工具栏状态栏等

4. 如果不确定是哪一种窗口类型，则可以选择普通窗口QWidget

4.1.2 创建主窗口

1. QMainWindow就是一个顶层窗口，它可以包含很多界面元素，如菜单栏、工具栏、子窗口等

2. QMainWindow中会有一个控件QWidget占位符来占据着整个中心窗口，可以使用setCentralWid

et来设置中心窗口

3. QMainWindow继承自QWidget类，拥有基类的所有方法和属性

4. QMainWindow类中比较重要的方法如下

- addToolBar　　添加工具栏
- centralWidget　　返回窗口中心的一个控件，未设置时返回NULL
- menuBar　　返回主窗口的菜单栏
- setCentralWidget　设置窗口中心的控件
- setStatusBar　　设置状态栏
- statusBar　　获得状态栏对象后，调用状态栏对象的showMessage(message,int timeout=0方法，第一个参数是要显示的message，第二个参数默认是0，表示一直显示状态栏信息，单位是毫秒5000表示停留在状态栏5s

注意QMainWindow不能设置布局(即使用setLayout()方法)，因为它有自己的布局，不需要再去设置

案例4-1 创建主窗口

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

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.resize(400, 200)
        self.status = self.statusBar()
        self.status.showMessage("这是状态栏提示", 5000)
        self.setWindowTitle("PyQt MainWindow例子")
        self.setWindowIcon(QIcon("images/cartoon1.ico"))
if __name__ == '__main__':
    app = QApplication(sys.argv)
    myWin = MainWindow()
    myWin.show()
    sys.exit(app.exec_())
```

4.1.3 将主窗口放在屏幕中间

```
import sys
from PyQt5.QtWidgets import QDesktopWidget, QApplication, QMainWindow

class WinForm(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("主窗口放在屏幕中间的例子")
        # 设置当前窗口的大小
        self.resize(370, 250)
        self.center()
    def center(self):
        # 获取当前显示屏幕的大小
        screen = QDesktopWidget().screenGeometry()
```

```
        # 获取当前创建的窗口的大小
        size = self.geometry()
        # 将创建的窗口移动到显示屏幕的中间位置
        self.move((screen.width() - size.width())/2, (screen.height() - size.height())/2)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = WinForm()
    win.show()
    sys.exit(app.exec_())
```

4.1.4 关闭窗口

1. 将按钮的关闭信号与自定义的槽函数关联起来，实现对当前实例化的应用程序的关闭操作

```
import sys
from PyQt5.QtWidgets import QMainWindow, QHBoxLayout, QPushButton, QApplication, QW
dget

class WinForm(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("关闭主窗口的例子")
        self.closeBtn = QPushButton('关闭主窗口')
        self.closeBtn.clicked.connect(self.onBtnClick)

        self.layout = QHBoxLayout()
        self.layout.addWidget(self.closeBtn)
        self.main_frame = QWidget()
        self.main_frame.setLayout(self.layout)
        self.setCentralWidget(self.main_frame)
        # 以上的布局就是顶层窗口---中心控件窗口---布局管理器放置在中心控件上---按钮放置在布局
理器这个容器中

    def onBtnClick(self):
        # sender 是发送信号的对象
        sender = self.sender()
        print(sender.text()+'被按下了')
        qApp = QApplication.instance()
        qApp.quit()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = WinForm()
    win.show()
    sys.exit(app.exec_())
```

2. 以上的布局就是顶层窗口---中心控件窗口---布局管理器放置在中心控件上---按钮放置在布局管理这个容器中

3. 在onBtnClick中获取QApplication的实例化对象，调用它的quit()函数来关闭窗口，在槽函数中还以获取发送信号的对象，这里通过text()获取发送对象的显示名称

4.2 QWidget

1. 基础窗口控件QWidget是所有用户界面对象的基类，所有窗口和控件都要直接或者间接继承自QWiget类

2. 窗口控件(Widget)是在PyQt中建立界面的主要元素

3. 没有嵌入到其他控件的控件称为窗口，一般窗口都有边框、标题栏、工具栏等

4. 控件是指按钮、复选框。文本框等组成程序的基本元素

5. 一个程序可以有多个窗口，一个窗口也可以有多个控件

4.2.1 窗口的坐标系统

1. QWidget 直接提供的成员函数 x()/y()获取窗口左上角的坐标，width()/height()获取客户区的宽度高度

2. QWidget的geometry()提供的成员函数 x()/y()获取客户区的左上角的坐标，width()/heught()获客户区的宽高

3. QWidget的frameGeometry()提供的成员函数 x()/y()获取窗口左上角的坐标，width()/height()获包含客户区、标题栏和边框在内的整个窗口的宽高

总结QWidget.geomotry()是客户区的位置大小；QWidget.frameGeomotry()是整个QWidget实例窗口的位置大小

4.2.2 常用的几何结构

QWidget的两种常用的几何结构

1. 不包含外边各种边框的几何结构（一般来说就是我们的客户区）

    1. 改变客户区的面积--其实就是一个矩形类的实例(QRect)

```
QWidget.resize(width, height)
QWidget.resize(Qsize)
```

    2. 获取客户区的大小

```
QWidget.size()
```

    3. 获取客户区的宽高

```
QWidget.width()
QWidget.height()
```

    4. 设置客户区的宽高

```
# 设置固定宽度
QWidget.setFixedWidth(int width)
# 设置固定高度
QWidget.setFixedHeight(int height)
# 设置固定宽高
QWidget.setFixedSize(int width,int height)
QWidget.seFixedSize(QSize size)
# 设置宽高的同时还设置客户区的位置
```

```
QWidget.setGeometry(int x,int y, int width, int height)
QWidget.setGeometry(QRect rect)
```

2. 包含外边各种边框的几何结构

    1. 获取整个窗口的大小和位置

```
QWidget.frameGeometry()
```

    2. 设置窗口的位置

```
QWidget.move(int x, int y)
QWidget.move(QPoint point)
```

    3. 获取窗口的左上角的坐标---即窗口的位置坐标

```
QWidget.pos()----即左上角的x、y坐标
```

案例4-4 屏幕坐标系统的显示

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

app = QApplication(sys.argv)
widget = QWidget()
btn = QPushButton(widget)
btn.setText("我是按钮")
# 以widget左上角为(0,0)
btn.move(20, 40)
widget.resize(300, 200)
# 以屏幕的左上角为(0,0)
widget.setWindowTitle('PyQt 坐标系统的显示案例')
widget.show()
# 获取窗口的左上角的坐标以及客户区的宽高
print("QWidget:")
print("w.x()=%d" % widget.x())
print("w.y()=%d" % widget.y())
print("w.width()=%d" % widget.width())
print("w.height()=%d" % widget.height())
# 获取客户区的左上角的坐标和客户区的宽高
print("QWidget.geometry:")
print("w.geometry().x()=%d" % widget.geometry().x())
print("w.geometry().y()=%d" % widget.geometry().y())
print("w.geometry().width()=%d" % widget.geometry().width())
print("w.geometry().height()=%d" % widget.geometry().height())
sys.exit(app.exec_())
```

4.2.3 创建第一个PyQt5应用程序

案例4-5 建立一个主窗口

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

```
app = QApplication(sys.argv)
win = QWidget()
win.resize(300, 200)
win.move(250, 150)
win.setWindowTitle('第一个应用程序APP')
win.show()
sys.exit(app.exec_())
```

解析

1. 这两行代码用来载人必须的模块，GUI窗口控件基本都在PyQt5.QtWidgets中

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

2. 每一个PyQt5程序都需要有一个QApplication对象，sys.argv 是一个命令行参数列表，比如py脚在shell中执行等启动脚本的参数是不一样的

```
app = QApplication(sys.argv)
```

3. QWidget控件是PyQt5中所有用户界面类的父类，第一行代码使用没有参数的构造函数，没有父类我们称不嵌套在控件中的控件也就是_没有父类的控件为窗口_

注意只有是窗口，才能设置窗口标题和大小，即后面两行代码才会生效

```
win = QWidget()
win.setWindowIcon()
win.setWindowTitle('第一个应用程序APP')
```

4. 使用resize()方法可以改变窗口的大小

```
win.resize(300, 200)
```

5. 使用move()方法可以设置窗口初始化的位置(x,y)

```
win.move(250, 150)
```

6. 使用show()方法将窗口控件显示在屏幕上

```
win.show()
```

7. **消息循环**最后进入到该程序的主循环，事件处理从本行代码开始，主循环接收事件消息并将其分发程序的各个控件，只有调用exit()或者主控件被销毁，主循环才圆满结束。使用sys.exit()方法退出可确保程序完整地执行结束，即运行成功，app.exec_()的返回值是0，否则返回值为非0

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

4.2.4 为应用程序设置图标

图标下载链接

案例4-6 设置程序图标

```
import sys
```

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

# 创建一个名为Icon的窗口类 继承自QWidget类
class Icon(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle("程序图标")
        self.setWindowIcon(QIcon("./images/cartoon1.ico"))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ico = Icon()
    ico.show()
    sys.exit(app.exec_())
```

解析from PyQt5.QtGui import QIcon导入QIcon模块，同时使用self.setWindowIcon(QIcon("./images/cartoon1.ico"))方法设置程序的图标，参数是一个QIcon类型的对象

4.2.5 显示气泡提示信息

案例 在关键操作的控件上设置一个气泡提示

```
import sys
from PyQt5.QtGui import QFont
from PyQt5.QtWidgets import QApplication, QWidget, QToolTip, QPushButton

class winForm(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        QToolTip.setFont(QFont('宋体', 10))
        self.setToolTip('widget')
        self.setGeometry(200, 300, 400, 400)
        self.setWindowTitle("气泡提示demo")
        btn = QPushButton(self)
        btn.setText('我是按钮')
        btn.setToolTip("这是一个<b>按钮</b>")
        btn.resize(100, 100)
        btn.move(20, 20)



if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = winForm()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 通过此行代码设置气泡提示信息的字体与字号大小

```
QToolTip.setFont(QFont('宋体', 10))
```

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

```
btn.setToolTip("这是一个<b>按钮</b>")
```

4.3 QLabel

1. QLabel对象作为一个占位符可以显示不可编辑的文本或者图片
2. 可以放置GIF动画，标记其他控件
3. 纯文本。链接或者富文本可以显示在标签上

常用方法

4. setAlignment　按照固定值方式对齐文本 如Qt.AlignLeft 水平方向靠左对齐
5. setIndent　　设置文本缩进
6. setPixmap　　设置QLabel为一个Pixmap图片资源
7. text()　　获取标签的文本内容
8. setText()　　设置QLabel的文本内容
9. selectedText() 返回所选择的字符
10. setBuddy()　　设置QLabel的助记符以及Buddy伙伴，使用setBuddy(QWidget *)设置 buddy，QLabel必须是文本内容，并且使用"&"符号设置了助记符

常用信号

11. linkActivated　当标签中嵌入的超链接被单击时，在新窗口打开这个超链接，setOpenExternalLinks特性必须为True
12. linkHovered　　当鼠标悬停在这个标签的超链接上时，需要用槽函数与这个信号进行绑定

案例4-7 显示QLabel标签

```
import sys
from PyQt5.QtGui import QPixmap, QPalette
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QLabel, QWidget, QVBoxLayout

class QLabelWin(QWidget):
    def __init__(self):
        super().__init__()
        label1 = QLabel(self)
        label2 = QLabel(self)
        label3 = QLabel(self)
        label4 = QLabel(self)
        # 初始化标签控件
        label1.setText("这是一个文本标签")
        # 表示填充背景
```

```
        label1.setAutoFillBackground(True)
        # 调色板
        palette = QPalette()
        palette.setColor(QPalette.Window, Qt.blue)
        label1.setPalette(palette)
        label1.setAlignment(Qt.AlignCenter)

        label2.setText("<a href='#'>欢迎使用Python GUI 应用</a>")
        label3.setAlignment(Qt.AlignCenter)
        label3.setToolTip('这是一个图片标签')
        label3.setPixmap(QPixmap("./images/python.jpg"))
        label4.setText("<a href='http://www.luffycity.com/'>欢迎访问路飞学城</a>")
        label4.setAlignment(Qt.AlignRight)
        label4.setToolTip("这是一个超链接标签")
        # 在窗口布局中添加控件
        vbox = QVBoxLayout()
        vbox.addWidget(label1)
        vbox.addStretch()
        vbox.addWidget(label2)
        vbox.addStretch()
        vbox.addWidget(label3)
        vbox.addStretch()
        vbox.addWidget(label4)
        # 允许标签label控件访问超链接
        label2.setOpenExternalLinks(False)
        # 打开允许访问超链接，默认是不允许，需要使用setOpenExternalLinks(True)允许浏览器访问
链接
        label4.setOpenExternalLinks(True)
        # 点击文本框绑定槽函数
        label4.linkActivated.connect(link_clicked)
        # 悬停在文本上绑定槽函数
        label2.linkHovered.connect(link_hovered)
        # 设置文本的交互形式是鼠标选择文本，使选择的文本处于选中状态
        label1.setTextInteractionFlags(Qt.TextSelectableByMouse)

        self.setLayout(vbox)
        self.setWindowTitle("QLabel 例子")

def link_clicked():
    print('当用鼠标点击label-4标签时，触发事件')

def link_hovered():
    print('当鼠标悬停在label-2标签时，触发事件')



if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = QLabelWin()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 设置文本标签居中显示

```
label1.setAlignment(Qt.AlignCenter)
```

2. 打开外部链接

提示打开允许访问超链接，默认是不允许，需要使用setOpenExternalLinks(True)允许浏览器访问超接

```
 label2.setOpenExternalLinks(False)
# 打开允许访问超链接，默认是不允许，需要使用setOpenExternalLinks(True)允许浏览器访问超链接
label4.setOpenExternalLinks(True)  # 设置True 表示鼠标点击超链接后就会跳转到对应的URL
# 点击label绑定槽函数
label4.linkActivated.connect(link_clicked)
# 悬停label绑定槽函数
label2.linkHovered.connect(link_hovered)
```

案例4-8 QLabel标签的快捷键的使用

```
import sys
from PyQt5.QtWidgets import *

class QLabelShortCut(QDialog):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("QLabel 快捷键的使用")
        nameLb1 = QLabel('&Name')
        nameEd1 = QLineEdit()
        nameLb1.setBuddy(nameEd1)

        nameLb2 = QLabel('&Password')
        nameEd2 = QLineEdit()
        nameLb2.setBuddy(nameEd2)
        btnOk = QPushButton('&OK')
        btnCancel = QPushButton('&Cancel')
        mainLayout = QGridLayout(self)
        mainLayout.addWidget(nameLb1, 0, 0)
        # 表示nameLb1 在栅格的第一行第二列并且行占据一行，列占据两列
        mainLayout.addWidget(nameEd1, 0, 1, 1, 2)
        mainLayout.addWidget(nameLb2, 1, 0)
        mainLayout.addWidget(nameEd2, 1, 1, 1, 2)
        mainLayout.addWidget(btnOk, 2, 1)
        mainLayout.addWidget(btnCancel, 2, 2)
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = QLabelShortCut()
    win.show()
    sys.exit(app.exec_())
```

解析

在弹出的窗口中，按"ALT+N"快捷键可以切换到第一个文本框，因为设置了&Name首字母快捷键

```
nameLb1 = QLabel('&Name')
```

```
nameEd1 = QLineEdit()
nameLb1.setBuddy(nameEd1)
```

4.4 文本框类控件

4.4.1 QLineEdit

单行文本控件，可以输入单行字符串，如果要输入多行，则需要使用QTextEdit类

常用方法

1. setAlignment()　　设置文本对齐方式
2. clear()　　清除文本框内容
3. setEchoMode()　　设置文本框显示格式
    - QLineEdit.Normal　表示正常显示输入的字符，此为默认选项
    - QLineEdit.NoEcho　表示不显示任何输入的内容，常用于密码类型的输入且保密密码的长度
    - QLineEdit.Password　表示显示的是输入的掩码，用户保护密码
    - QLineEdit.PasswordEchoOnEdit 表示在编辑时显示输入的字符，负责显示密码的输入----输结束后显示为掩码
4. setPlaceholderText()　设置文本框的浮动文字
5. setMaxLength()　　设置文本框所允许输入的最大字符数
6. setReadOnly()　　设置文本框为只读的
7. setText()　　设置文本框的内容
8. Text()　　获取文本框的内容
9. setDragEnabled()　　设置文本框是否允许拖拽
10. selectAll()　　全选
11. setFocus()　　获得焦点
12. setInputMask()　　设置输入掩码
13. setValidator()　　设置文本框的验证器
    - QIntValidator　　限制输入整数
    - QDoubleValidator　限制输入浮点数
    - QRegexpValidator　检查输入是否符合正则表达式

重点设置验证器中的正则检测是最常用的最实用的

14. 掩码由掩码字符和分隔符字符串组成，后面可以跟一个分号和空白字符，空白字符在编辑后会从本中删除
15. 常见的几种掩码注意
    - 000.000.000.000;_　　IP地址，空白字符是"_"
    - HH:HH:HH:HH:HH;_　　MAC地址
    - 0000:00:00　　日期，空白字符是空格
    -

AAAA-AAAA-AAAA-AAAA;# 许可证号，空白字符是"-"，所有字母字符都要转换为大写

常用信号

16. selectionChanged 只要选择改变了，这个信号就会被发射
17. textChanged 只要文本内容被修改了，就会发射这个信号
18. editingFinished 当编辑文本结束时，这个信号就会被发射出去

案例4-9 EchoMode的显示效果

```
import sys
from PyQt5.QtWidgets import QApplication, QLineEdit, QWidget, QFormLayout

class lineEditEchoMode(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-9 EchoMode的显示效果")

        flo = QFormLayout()
        pNormalLineEdit = QLineEdit()
        pNoEchoLineEdit = QLineEdit()
        pPasswordLineEdit = QLineEdit()
        pPasswordEchoOnEditLineEdit = QLineEdit()

        flo.addRow("Normal", pNormalLineEdit)
        flo.addRow("NoEcho", pNoEchoLineEdit)
        flo.addRow("Password", pPasswordLineEdit)
        flo.addRow("PasswordOnEdit", pPasswordEchoOnEditLineEdit)

        pNormalLineEdit.setPlaceholderText("Normal")
        pNoEchoLineEdit.setPlaceholderText("NoEcho")
        pPasswordLineEdit.setPlaceholderText("Password")
        pPasswordEchoOnEditLineEdit.setPlaceholderText("PasswordEchoOnEdit")

        # 设置显示效果
        pNormalLineEdit.setEchoMode(QLineEdit.Normal)
        pNoEchoLineEdit.setEchoMode(QLineEdit.NoEcho)
        pPasswordLineEdit.setEchoMode(QLineEdit.Password)
        pPasswordEchoOnEditLineEdit.setEchoMode(QLineEdit.PasswordEchoOnEdit)

        self.setLayout(flo)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = lineEditEchoMode()
    win.show()
    sys.exit(app.exec_())
```

案例4-10 验证器

```
import sys
from PyQt5.QtGui import QIntValidator, QDoubleValidator, QRegExpValidator
```

```
from PyQt5.QtCore import QRegExp
from PyQt5.QtWidgets import QApplication, QWidget, QFormLayout, QLineEdit

class lineEditValidators(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("验证器demo")

        flo = QFormLayout()
        pIntLineEdit = QLineEdit()
        pdoubleLineEdit = QLineEdit()
        pRegexpLineEdit = QLineEdit()

        flo.addRow("整型", pIntLineEdit)
        flo.addRow("浮点型", pdoubleLineEdit)
        flo.addRow("字符和数字", pRegexpLineEdit)

        pIntLineEdit.setPlaceholderText("整型")
        pdoubleLineEdit.setPlaceholderText("浮点型")
        pRegexpLineEdit.setPlaceholderText("字母和数字")

        # 整型  范围 [1, 99]
        pIntValidator = QIntValidator(self)
        pIntValidator.setRange(1,99)

        # 浮点型 范围 【-360,360】 精度：小数点后两位
        pDoubleValidator = QDoubleValidator(self)
        pDoubleValidator.setRange(-360, 360)
        # 设置标准符号
        pDoubleValidator.setNotation(QDoubleValidator.StandardNotation)
        pDoubleValidator.setDecimals(2)

        # 子母和数字
        reg = QRegExp("[a-zA-Z0-9]+$")
        pRegexpValidator = QRegExpValidator(self)
        pRegexpValidator.setRegExp(reg)

        # 设置验证器
        pIntLineEdit.setValidator(pIntValidator)
        pdoubleLineEdit.setValidator(pDoubleValidator)
        pRegexpLineEdit.setValidator(pRegexpValidator)

        self.setLayout(flo)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = lineEditValidators()
    win.show()
    sys.exit(app.exec_())
```

案例4-11 输入掩码

_用途_要限制用户的输入，除了验证器，还可以使用掩码，常见的有IP地址日期等

```
import sys
```

```
from PyQt5.QtWidgets import QLineEdit, QApplication, QWidget, QFormLayout


class lineEditMask(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('QLineEdit 的输入掩码的例子')

        flo = QFormLayout()
        pIPLineEdit = QLineEdit()
        pMACLineEdit = QLineEdit()
        pDateLineEdit = QLineEdit()
        pLicenseLineEdit = QLineEdit()

        pIPLineEdit.setInputMask("000.000.000.000;_")
        pMACLineEdit.setInputMask("HH:HH:HH:HH:HH:HH;_")
        pDateLineEdit.setInputMask("0000.00.00")
        pLicenseLineEdit.setInputMask(">AAAAA-AAAAA-AAAAA-AAAAA-AAAAA;#")
        flo.addRow("IP掩码", pIPLineEdit)
        flo.addRow("MAC掩码", pMACLineEdit)
        flo.addRow("日期掩码", pDateLineEdit)
        flo.addRow("许可证掩码", pLicenseLineEdit)

        self.setLayout(flo)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = lineEditMask()
    win.show()
    sys.exit(app.exec_())
```

案例4-12 QLineEdit的综合示例

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QLineEdit, QFormLayout
from PyQt5.QtGui import QIntValidator, QDoubleValidator, QRegExpValidator, QFont
from PyQt5.QtCore import QRegExp, Qt

class lineEditComprehensive(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("QLineEdit的综合案例")
        el = QLineEdit()
        el.setValidator(QIntValidator())
        el.setMaxLength(4)
        el.setAlignment(Qt.AlignRight)
        el.setFont(QFont('Arial', 20))
        e2 = QLineEdit()
        e2.setValidator(QDoubleValidator(0.99, 99.99, 2))
        flo = QFormLayout()
        flo.addRow("integer validator", el)
        flo.addRow("Double validator", e2)
        e3 = QLineEdit()
        e3.setInputMask('+99_9999_999999')
```

```
        flo.addRow("Input Mask", e3)
        e4 = QLineEdit()
        e4.textChanged.connect(self.textchanged)
        flo.addRow("Text changed", e4)
        e5 = QLineEdit()
        e5.setEchoMode(QLineEdit.Password)
        # 编辑结束时也就是回车键被按下时，因为只有回车键按下，编辑的修改才会生效
        e5.editingFinished.connect(self.enterPress)
        flo.addRow("Password", e5)
        e6 = QLineEdit("Hello PyQt5")
        e6.setReadOnly(True)
        flo.addRow("Read Only", e6)
        self.setLayout(flo)
    def textchanged(self, text):
        print("输入内容为：" + text)

    def enterPress(self):
        print("密码输入完毕")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = lineEditComprehensive()
    win.show()
    sys.exit(app.exec_())
```

解析

1. e1 显示文本的自定义字体，右对齐，只允许输入整数
2. e2 限制输入小数点后两位
3. e3 需要一个输入掩码应用于电话号码
4. e4 需要发射信号textchanged 链接到槽函数 textchanged()
5. e5 设置显示模式EchoMode为密码输入Password
6. e6 显示一个默认的文本，不能编辑为只读模式

4.4.2 QTextEdit

QTextEdit类是一个多行文本框控件，可以显示多行文本的内容，当文本内容超过控件显示范围时，以显示滚动条，

记住QTextEdit不仅可以显示文本还可以显示HTML文档

常见方法

1. setPlainText()　　设置多行文本框的内容
2. toPlainText()　　获取多行文本框的内容
3. setHtml()　　设置文本框内容HTML文档
4. toHtml()　　获取多行文本框内的HTML文档
5. clear()　　清除多行文本框的内容

案例4-13 QTextEdit的使用

```
import sys
from PyQt5.QtWidgets import QApplication, QTextEdit, QWidget, QVBoxLayout, QPushButton

class textEditDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-13 QTextEdit的使用")
        self.resize(300, 270)
        self.textEdit = QTextEdit()
        self.btnText = QPushButton("显示文本")
        self.btnHTML = QPushButton("显示HTML")
        layout = QVBoxLayout()
        layout.addWidget(self.textEdit)
        layout.addWidget(self.btnText)
        layout.addWidget(self.btnHTML)
        self.setLayout(layout)
        self.btnText.clicked.connect(self.btnText_Clicked)
        self.btnHTML.clicked.connect(self.btnHTML_Clicked)

    def btnText_Clicked(self):
        self.textEdit.setPlainText("Hello PyQt5！\n单击按钮")

    def btnHTML_Clicked(self):
        self.textEdit.setHtml("<font color='red' size='6'><red>Hello PyQt5！\n单击按钮</font>"


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = textEditDemo()
    win.show()
    sys.exit(app.exec_())
```

4.5 按钮类控件

4.5.1 QAbstractButton

概念QAbstractButton类是抽象类，不能实例化，是所有其他按钮类的父类，自定义按钮类也必须要承该类并实现想要的功能

基类的状态

1. isDown()　　提示按钮是否被按下
2. isChecked()　　提示按钮是否已经被标记
3. isEnable()　　提示按钮是否可以被用户点击
4. ischeckAble()　　提示按钮是否可以被标记
5. setAutoRepeat()　设置按钮是否在用户长按时可以自动重复执行槽函数

基类的信号

6. Pressed()　　当鼠标指针在按钮上左键按下时发射信号
7. Released()　　当鼠标左键被释放时发射该信号
8. Clicked()　　当鼠标左键被按下然后释放时或者快捷键被释放时发射该信号

9. Toggled()　　当按钮的标记发生开关式切换时发射该信号

4.5.2 QPushButton

QPushButton 类继承QAbstractButton类，形状为矩形可以显示文本标题或者图标，同时也是一个令按钮，可以执行一些命令或者响应一些事件，常见的有确认、申请、取消、关闭等

1. QPushButton类中的常见方法

- setCheckable()　　设置按钮是否已经被选中，是为True，反之False
- toggle()　　切换按钮的状态
- setIcon()　　设置按钮上的图标
- setEnabled()　　设置按钮是否可以使用
- isChecked()　　返回按钮的状态
- setDefault()　　设置按钮的默认状态
- setText()　　设置按钮的显示文本
- text()　　返回按钮的显示文本

2. 为QPushButton设置快捷键

```
self.button = QPushButton("&Down")
self.button.setDefault(True)
```

小知识点要想单纯显示&符号就需要使用"&&"，可以类比转义字符

案例4-14 QPushButton按钮的使用

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

class pushButtonDemo(QDialog):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-14 QPushButton按钮的使用")
        layout = QVBoxLayout()
        self.btn = QPushButton("Button1")
        self.btn.setCheckable(True)
        self.btn.toggle()
        # 使用lambda匿名函数来表示槽函数名---此时我们看到就可以传递我们想要传递的参数了
        self.btn.clicked.connect(lambda:self.wicthBtn(self.btn))
        # 直接使用函数名
        self.btn.clicked.connect(self.btnState)
        layout.addWidget(self.btn)

        self.btnImage = QPushButton("image")
        # 将非ico图标利用像素映射转换为ico类型支持的图片资源
        self.btnImage.setIcon(QIcon(QPixmap("./images/python.png")))
        self.btnImage.clicked.connect(lambda: self.wicthBtn(self.btnImage))
        self.btnImage.clicked.connect(self.btnState)
```

```
        layout.addWidget(self.btnImage)

        self.btnDisabled = QPushButton("Disabled")
        self.btnDisabled.setEnabled(False)
        layout.addWidget(self.btnDisabled)

        self.btnShortCut = QPushButton("&Download")
        self.btnShortCut.setDefault(True)
        self.btnShortCut.clicked.connect(lambda: self.wicthBtn(self.btnShortCut))
        self.btnShortCut.clicked.connect(self.btnState)
        layout.addWidget(self.btnShortCut)
        self.setLayout(layout)

    def btnState(self):
        if self.btn.isChecked():
            print('button pressed')
        else:
            print('button released')
    def wicthBtn(self, btn):
        print("clicked button is: " + btn.text())
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = pushButtonDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 使用_lambda匿名函数_的方式可以传递额外的参数 btn

```
self.btn.clicked.connect(lambda:self.wicthBtn(self.btn))
```

2. setIcon()使用QPixmap对象传递一个图像文件作为参数并将结果转换成QIcon的ico文件的路径

```
self.btnImage.setIcon(QIcon(QPixmap("./images/python.png")))
```

4.5.3 QRadioButton

当将单选按钮切换到on或者off时，就会发送toggle信号，绑定这个信号，在按钮状态发生改变时就触发相应的行为

常用方法

1. setCheckedable()　　设置按钮的是否被选中的状态
2. isChecked()　　返回按钮的状态
3. setText()　　设置按钮的显示文本
4. text()　　获取显示文本

经验之谈

其实toggle信号是在切换单选状态时发射的，而chicked的信号是在每次点击时触发的，在实际开发，一般只有状态的改变才有必要去响应，因此toggle信号更适合用于QRadioButton的状态监控

案例4-15 QRadioButton按钮的使用

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

class Radiodemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-15 QRadioButton按钮的使用")
        layout = QHBoxLayout()
        self.btn1 = QRadioButton("Button1")
        self.btn1.setChecked(True)
        self.btn1.toggled.connect(lambda: self.btnState(self.btn1))
        layout.addWidget(self.btn1)

        self.btn2 = QRadioButton("Button2")
        self.btn2.toggled.connect(lambda: self.btnState(self.btn2))
        layout.addWidget(self.btn2)
        self.setLayout(layout)

    def btnState(self, btn):
        if btn.text() == "Button1":
            if btn.isChecked() == True:
                print(btn.text() + "is selected")
            else:
                print(btn.text() + "is deselected")
        if btn.text() == "Button2":
            if btn.isChecked() == True:
                print(btn.text() + "is selected")
            else:
                print(btn.text() + "is deselected")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    radioDemo = Radiodemo()
    radioDemo.show()
    sys.exit(app.exec_())
```

解析

1. 在这个例子中，两个互斥的单选按钮被放置在窗口中，当选择按钮时按钮状态发生改变，将触发toggle信号，从而调用槽函数

```
self.btn1.toggled.connect(lambda: self.btnState(self.btn1))
self.btn2.toggled.connect(lambda: self.btnState(self.btn2))
```

当触发toggle信号后，使用btnState函数来检查按钮的状态

4.5.4 QCheckBox

1. 用户可以选择多个选项，和QPusheButton一样，复选框可以显示文本或者图标，其中文本可以通构造函数或者setText()来设置，图标可以通过setIcon()来设置

2. 在视觉上，QButtonGroup/QGroupBox可以把许多复选框组织在一起

3. QCheckBox在选中和取消选中时，都会发射一个 stateChangeed信号，如果想在复选框状态发改变时触发相应的行为，就连接该信号

4. 可以使用isChecked() 来查询复选框是否被选中

5. 除了常用的选中和未选中两种状态，QCheckBox还有第三种状态(半选中)， 来表明"没有变化"

6. 如果要使用第三种状态，则可以通过setTriState()来使它生效。并使用checkState()来查询当前的换状态

常用方法

7. setChecked() 设置复选框的状态

8. setText() 设置复选框的显示文本

9. text() 获取复选框的显示文本

10. isChecked() 检查复选框是否被选中

11. setTriState() 设置复选框为三态复选框--其中一状态是半选中

12. checkState() 检查复选框的状态

三态复选框的状态

13. Qt.Checked 组件被选中

14. Qt.PartiallyChecked 组件被半选中

15. Qt.Unchecked 组件未被选中

案例4-16 QCheckBox 按钮的使用

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

class CheckBoxDemo(QWidget):
    def __init__(self):
        super().__init__()
        groupBox = QGroupBox("Checkboxes")
        # 除去边框，使其与背景色融为一体
        groupBox.setFlat(True)
        layout = QHBoxLayout()
        self.checkBox1 = QCheckBox("&CheckBox1")
        self.checkBox1.setChecked(True)
        self.checkBox1.stateChanged.connect(lambda:self.btnState(self.checkBox1))
        layout.addWidget(self.checkBox1)

        self.checkBox2 = QCheckBox("checkBox2")
        self.checkBox2.stateChanged.connect(lambda: self.btnState(self.checkBox2))
        layout.addWidget(self.checkBox2)

        self.checkBox3 = QCheckBox("checkBox3")
        # 使用三态复选框
        self.checkBox3.setTristate(True)
        # 设置选中状态为半选中
        self.checkBox3.setCheckState(Qt.PartiallyChecked)
        self.checkBox3.stateChanged.connect(lambda: self.btnState(self.checkBox3))
        layout.addWidget(self.checkBox3)
```

```
        groupBox.setLayout(layout)

        mainLayout = QVBoxLayout()
        mainLayout.addWidget(groupBox)
        self.setLayout(mainLayout)

        self.setWindowTitle("案例4-16 QCheckBox 按钮的使用")

    def btnState(self, btn):
        chk1State = btn.text()+", is checked="+\
                    str(btn.isChecked())+", checkState="+\
                    str(btn.checkState())+"\n"

        print(chk1State)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = CheckBoxDemo()
    win.show()
    sys.exit(app.exec_())
```

_注意_三态复选框中的isChecked()获取的选中状态中半选中状态也是True，归于选中的那一方(毕竟选中是向选中状态进行过渡)

解析

1. 在这个例子中，将三个复选框添加到一个水平布局管理器中，并添加到一个QGroupBox组中

```
groupBox = QGroupBox("Checkboxes")
# 除去边框，使其与背景色融为一体
groupBox.setFlat(True)
```

2. 将复选框的stateChanged信号都链接到槽函数stateChanged()使用lambda匿名函数的方式将对传递给槽函数

```
def btnState(self, btn):
    chk1State = btn.text()+", is checked="+\
    str(btn.isChecked())+", checkState="+\
    str(btn.checkState())+"\n"

    print(chk1State)
```

3. 实例化一个QCheckBox类对象checkBox3，然后使用setTristate()开启三态模式

```
        self.checkBox3 = QCheckBox("checkBox3")
        # 使用三态复选框
        self.checkBox3.setTristate(True)
        # 设置选中状态为半选中
        self.checkBox3.setCheckState(Qt.PartiallyChecked)
        self.checkBox3.stateChanged.connect(lambda: self.btnState(self.checkBox3))
        layout.addWidget(self.checkBox3)
```

4.6 QComboBox(下拉列表框)

1. QComboBox是一个集按钮和下拉选项于一体的控件，也被称为下拉列表框
2. QComboBox常用的方法

- addItem()　　添加一个下拉选项
- addItems()　　从列表中添加下拉选项
- Clear()　　删除下拉选项集合中的所有选项
- count()　　返回下拉选项中的所有数据的统计
- currentText()　　返回当前选中项的文本
- itemText(i)　　获取索引为i的选项的文本
- currentIndex()　　返回当前选中项的索引
- setItemText(int index, text)设置index对应的那个选项的文本内容

3. QComboBox常用的信号

- Activated　　当用户选中一个下拉选项时发射该信号
- currentIndexChanged　当下拉选项的索引发生变化时发射该信号
- highlighted　　当选中一个已经被选中的下拉选项时，发射该信号

案例4-17 QComboBox下拉列表框的使用

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

class ComboxDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-17 QComboBox下拉列表框的使用")
        self.resize(300, 90)
        layout = QVBoxLayout()
        self.lb1 = QLabel("")
        self.cb = QComboBox()
        self.cb.addItem("C")
        self.cb.addItem("C++")
        self.cb.addItems(["Java", "C#", "Python"])
        self.cb.currentIndexChanged.connect(self.selectionOnchanged)
        layout.addWidget(self.cb)
        layout.addWidget(self.lb1)

        self.setLayout(layout)

    def selectionOnchanged(self, i):
        self.lb1.setText(self.cb.currentText())
        print("Items in the list are: ")
        for count in range(0, self.cb.count()):
            print('item'+str(count)+'='+self.cb.itemText(count))
            print("Current index", i, "selection changed", self.cb.currentText())
```

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = ComboxDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 添加下拉选项时可以使用单个添加还可以使用列表一次添加多个

```
self.cb.addItem("C")
self.cb.addItem("C++")
self.cb.addItems(["Java", "C#", "Python"])
```

2. 当下拉框的选项发生改变时就会发射currentIndexChanged这个信号并连接到自定义的那个槽函中selectionOnchanged

```
self.cb.currentIndexChanged.connect(self.selectionOnchanged)
```

3. 当选中下拉列表框中的一个选项时，将该选项中的文本设置为标签的文本

```
# i是选中列表项的索引，可以不使用传参直接使用currentIndex获取
def selectionOnchanged(self, i):
    self.lb1.setText(self.cb.currentText())
```

4.7 QSpinBox(计数器)

1. QSpinBox是一个计数器控件，允许用户选择还可以调节，也能手动输入

2. 默认QSpinBox的取值是0-99，步长是1

3. QSpinBox类和QDoubleSpinBox类均派生自QAbstractSpinBox类，前者用于处理整型数据，后则是处理浮点型数据

4. QDoubleSpinBox的默认精度是小数点后两位，但是可以通过setDecimals()来改变

常用的方法

5. setMinimum()　　设置计数器的下限也就是最小值

6. setMaximum()　　设置计数器的上限也就是最大值

7. setRange()　　设置计数器的最大值、最小值和步长值

8. setValue()　　设置计数器的当前值

9. Value()　　获取计数器的当前值

10. singleStep()　　设置计数器的步长值

常用的信号

11. valueChanged()　　QSpinBox计数器在当前值被改变时就会发射该信号，可以在槽函数中通过alue来获取计数器的当前值

案例4-18 QSpinBox计数器的使用

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

```
from PyQt5.QtGui import *

class spinDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-18 QSpinBox计数器的使用")
        self.resize(300, 100)
        layout = QVBoxLayout()
        self.lb1 = QLabel("current value:")
        self.lb1.setAlignment(Qt.AlignCenter)
        layout.addWidget(self.lb1)

        self.sp = QSpinBox()
        layout.addWidget(self.sp)
        self.sp.valueChanged.connect(self.valueChanged)

        self.setLayout(layout)

    def valueChanged(self):
        self.lb1.setText("current value :"+ str(self.sp.value()))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = spinDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 将一个标签和计数器放在一个垂直布局管理器中并且居中显示，并且将信号 self.sp.valueChanged.onnect(self.valueChanged)与自定义槽函数进行关联

```
self.sp = QSpinBox()
layout.addWidget(self.sp)
self.sp.valueChanged.connect(self.valueChanged)
```

2. valueChanged函数将计数器的当前值设置在标签文本中

```
def valueChanged(self):
    self.lb1.setText("current value :"+ str(self.sp.value()))
```

4.8 QSlider(滑动条)

1. QSlider控件提供了允许用户水平或者垂直方向移动的滑块，并将滑块的位置转换为一个合法范围的整数值

_本质_在槽函数中对滑块所在位置的处理就相当于是从一个整数范围内取值

2. 可以在构造函数时控制滑块的显示方式

```
self.slider = QSlider(Qt.Horizontal) # 水平方向的滑块
self.slider = QSlider(Qt.Vertical)   # 垂直方向的滑块
```

常用的方法

1. setMinimum()　　设置滑块的下限也就是最小值
2. setMaximum()　　设置滑块的上限也就是最大值
3. setValue()　　设置滑块的当前值
4. Value()　　获取滑块的当前值
5. setTickInterval()　　设置刻度的间隔值
6. setSingleStep()　　设置滑块递增/递减的步长值
7. setTickPosition()　　设置刻度标记的位置，可以输入一个枚举值，这个枚举值指定刻度线相对于块和用户操作的位置

- QSlider.NoTicks　　不绘制任何刻度线
- QSlider.TicksBothSides　在滑块的两侧绘制刻度线
- QSlider.TicksAbove　　在水平滑块上方绘制刻度线
- QSlider.TicksBelow　　在水平滑块的下面绘制刻度线
- QSlider.TicksLeft　　在垂直滑块的左边绘制刻度线
- QSlider.TicksRight　　在垂直滑块的右边绘制刻度线

常用的信号

8. valueChanged　　当滑块的值发生变化时发射此信号，此信号是最常使用的
9. sliderPressed　　当用户按下滑块时触发该信号
10. sliderMoved　　当用户拖动滑块时发射此信号
11. sliderReleased　　当用户释放滑块时发射该信号

案例4-19 QSlider滑块的使用

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

class sliderDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-19 QSlider滑块的使用")
        self.resize(300, 100)

        layout = QVBoxLayout()
        self.lb1 = QLabel("Hello PyQt5")
        self.lb1.setAlignment(Qt.AlignCenter)
        layout.addWidget(self.lb1)

        # 水平方向
        self.s1 = QSlider(Qt.Horizontal)
        # 设置最小值
        self.s1.setMinimum(10)
        # 设置最大值
        self.s1.setMaximum(50)
        # 设置步长
        self.s1.setSingleStep(3)
```

```
        # 设置当前值
        self.s1.setValue(20)
        # 刻度放置的位置，刻度在下方
        self.s1.setTickPosition(QSlider.TicksBelow)

        # 设置刻度间隔
        self.s1.setTickInterval(5)

        layout.addWidget(self.s1)
        self.setLayout(layout)
        # 连接信号槽函数
        self.s1.valueChanged.connect(self.valueChanged)
    def valueChanged(self):
        print('current slider value=%s' % self.s1.value())
        size = self.s1.value()
        self.lb1.setFont(QFont("Arial", size))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = sliderDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 在这个例子中，将一个标签和滑块放置在垂直布局中，将滑块的值变化关联到对应的槽函数上

```
# 连接信号槽函数
self.s1.valueChanged.connect(self.valueChanged)
```

2. 设置间隔为5，最大是50，最小是10，一共是(最大值-最小值)/5+1个刻度点，本例中就是(50-10)/
+1 = 9

```
# 设置最小值
self.s1.setMinimum(10)
# 设置最大值
self.s1.setMaximum(50)
# 设置步长
self.s1.setSingleStep(3)
# 设置当前值
self.s1.setValue(20)
# 刻度放置的位置，刻度在下方
self.s1.setTickPosition(QSlider.TicksBelow)
# 设置刻度间隔
self.s1.setTickInterval(5)
```

4.9 对话框类控件

4.9.1 QDialog

1. QDialog类的子类主要有

- QMessageBox

- QFileDialog
  - QInputDialog
  - QColorDialog
  - QFontDialog

2. 常用的方法

  - setWindowTitle()　　设置对话框的标题
  - setWindowModality()　设置窗口模态
    - Qt.NonModal　　非模态，可以跟程序的其他窗口交互
    - Qt.WindowModal　窗口模态，程序在未处理完当前对话框时，将阻止和对话框的父窗口互(但是可以跟其他应用程序窗口交互)
    - Qt.ApplicationModal应用程序模态，在当前任务未处理完毕时，阻止和其他任何的窗口进交互

案例4-20 QDialog对话框的使用

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

class DialogDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-20 QDialog对话框的使用")
        self.resize(350, 300)

        self.btn = QPushButton(self)
        self.btn.setText("弹出对话框")
        self.btn.move(50, 50)
        self.btn.clicked.connect(self.showdialog)

    def showdialog(self):
        dialog = QDialog()
        btn = QPushButton("ok", dialog)
        btn.move(50, 50)
        dialog.setWindowTitle("Dialog")
        # 设置对话框的模态属性为应用程序级别的模态
        dialog.setWindowModality(Qt.ApplicationModal)
        dialog.exec_()

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

解析

1. 在Dialog窗口的windowModality属性决定是否为模态或者非模态。当用户_按下Esc键时_，对话

会默认调用QDialog.reject(),然后关闭对话框窗口----------reject意思是抛弃、摒弃

2. 由于对话框的windowModality属性设置为Qt.ApplicationModal模态，用户只有处理完毕当前弹来的所有对话框后，才能关闭主窗口

dialog.setWindowModality(Qt.ApplicationModal)

总结

1. Qt.ApplicationModal用来设置弹出的对话框为完全模态，不将弹出来的所有对话框关闭是不能关主窗口的

2. Qt.WindowModal用来设置弹出的对话框为不完全模态，这样在处理主窗口时就不要先处理完毕出来的对话框，也能直接关闭主窗口了

4.9.2 QMessageBox

1. QMessageBox是一种通用的弹出式对话框，用于显示消息，允许用户通过点击不同标准的按钮对息进行反馈，每个标准都有一个预定义的文本、角色和十六进制数

2. QMessageBox类提供了常用的对话框，如提示、警告、错误、询问、关于等对话框，这些对话框是图标不同，其他的功能是一样的

3. **常用的方法**

- information(QWidget parent,title,text,button,defaultButton)　　弹出消息对话框
  - parent　　指定的父窗口控件
  - title　　对话框的标题
  - text　　对话框的文本
  - button　　多个标准按钮，默认是OK按钮
  - defaultButton　　默认选中的标准按钮，一般默认选中的就是第一个标准按钮
- question(QWidget parent,title,text,button,defaultButton)　　弹出问答对话框
- warning(QWidget parent,title,text,button,defaultButton)　　弹出警告对话框
- critical(QWidget parent,title,text,button,defaultButton)　　弹出严重错误对话框
- about(QWidget parent,title,text,button,defaultButton)

弹出关于对话框

- setTitle()　　设置标题
- setText()　　设置消息正文
- setIcon()　　设置弹出对话框的图标

4. **常用的标准按钮**

- QMessage.Ok　　同意操作
- QMessage.Cancel　　取消操作
- QMessage.Yes　　同意操作
- QMessage.No　　取消操作
- QMessage.Abort　　终止操作

- QMessage.Retry　重试操作
- QMessage.Ignore　忽略操作

案例4-21 QMessageBox的使用

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

class MyMessageBox(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-21 QMessageBox的使用")
        self.resize(300, 100)
        self.myButton = QPushButton(self)
        self.myButton.setText("点击弹出消息框")
        self.myButton.clicked.connect(self.msg)
    def msg(self):
        # 使用QMessageBox中的information对话框演示
        popmsg = QMessageBox.information(self,"标题", "消息正文", QMessageBox.Yes|QMessag
Box.No, QMessageBox.Yes)
        print(popmsg)
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = MyMessageBox()
    win.show()
    sys.exit(app.exec_())
```

4.9.3 QInputDialog

1. QInputDialog控件是一个标准对话框，由一个文本框和两个按钮(_OK和Cancel_按钮)组成
2. 用户单击OK或者按Enter键后，在父窗口可以收集通过QInputDialog控件输入的信息
3. 在QInputDialog中可以输入数字、字符串或者列表中的选项，标签用于显示必要的信息

常用的方法

4. getInt()　从控件中获得标准整数的输入
5. getDouble()　从控件中获得标准浮点数的输入
6. getText()　从控件中获得标准字符串的输入
7. getItem()　从控件中获得列表里的选项输入

案例4-22 QInputDialog输入对话框的使用

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

class inputDialogDemo(QWidget):
    def __init__(self):
        super().__init__()
```

```
        self.setWindowTitle("案例4-22 QInputDialog输入对话框的使用")

        layout = QFormLayout()
        self.btn1 = QPushButton("获得列表中的选项")
        self.btn1.clicked.connect(self.getItem)
        self.le1 = QLineEdit()
        layout.addRow(self.btn1, self.le1)

        self.btn2 = QPushButton("获得字符串")
        self.btn2.clicked.connect(self.getText)
        self.le2 = QLineEdit()
        layout.addRow(self.btn2, self.le2)

        self.btn3 = QPushButton("获得整数")
        self.btn3.clicked.connect(self.getInt)
        self.le3 = QLineEdit()
        layout.addRow(self.btn3, self.le3)

        self.setLayout(layout)

    def getItem(self):
        items = ("C", "C++", "Java", "Python")
        # 参数说明是父级窗口、标题、正文、迭代列表、默认第一个选项被选中、editable是否可以被
辑  默认是True
        item, ok = QInputDialog.getItem(self, "Select input dialog", "语言列表", items, 0, False)
        if ok and item:
            self.le1.setText(item)

    def getText(self):
        text, ok = QInputDialog.getText(self, "Text input dialog", "输入文本：")
        if  ok:
            self.le2.setText(text)

    def getInt(self):
        num, ok = QInputDialog.getInt(self, "integer input dialog", "输入整数：")
        if ok:
            self.le3.setText(str(num))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = inputDialogDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. QInputDialog.getItem()函数被调用时，弹出的对话框中包含一个Qcombo下拉列表框控件和两按钮，用户从其中选择一个选项时，允许用户确认或者取消操作

```
def getItem(self):
    items = ("C", "C++", "Java", "Python")
    item, ok = QInputDialog.getItem(self, "Select input dialog", "语言列表", items, 0, False)
    if ok and item:
        self.le1.setText(item)
```

2. 同理QInputDialog.getInt()包含一个QSpinBox计数器控件和两个按钮，允许用户输入整数

4.9.4 QFontDialog

1. QFontDialog控件是一个常用的字体选择对话框，可以让用户选择显示文本的字号大小、样式和格式

常用方法

2. getFont()　　从控件中选取所想要的字体大小、样式和格式

案例4-23 QFontDialog字体对话框的使用

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

class FontDialogDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-23 QFontDialog字体对话框的使用")
        layout = QVBoxLayout()
        self.fontButton = QPushButton("选择字体")
        self.fontButton.clicked.connect(self.getFont)
        self.fontEdit = QLabel("hello 测试字体例子")
        layout.addWidget(self.fontButton)
        layout.addWidget(self.fontEdit)

        self.setLayout(layout)
    def getFont(self):
        font, ok = QFontDialog.getFont()
        if ok:
            self.fontEdit.setFont(font)
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = FontDialogDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 通过接收选择的字体对象以及ok状态来在槽函数中作出响应

```
def getFont(self):
    font, ok = QFontDialog.getFont()
    if ok:
        self.fontEdit.setFont(font)
```

4.9.5 QFileDialog

1. QFileDialog是用于打开文件时使用的文件过滤器，用于显示指定扩展名的文件，也可以设置QFileDialog打开文件时的起始目录和指定的扩展名文件

2. **常用的方法**

- getOpenFileName()　　返回用户选择文件的文件名，并打开该文件
  - getSaveFileName()　　返回用户选择的文件名并保存文件
  - setFileMode()　　设置选择的文件类型，常见的有
    - QFileDialog.AnyFile　　任何文件
    - QFileDialog.ExistingFile　已存在的文件
    - QFileDialog.Directory　　文件目录
    - QFileDialog.ExistingFiles　已存在的多个文件
  - setFilter()　　设置过滤器，只显示过滤器允许的文件类型，其他的文件不显示在文件对框中

案例4-24 QFileDialog文件对话框的使用

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

class FileDialogDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-24 QFileDialog文件对话框的使用")
        layout = QVBoxLayout()
        self.btn = QPushButton("加载图片")
        self.btn.clicked.connect(self.getPic)
        self.lb1 = QLabel("")
        self.btn1 = QPushButton("加载文本文件")
        self.btn1.clicked.connect(self.getTxt)
        self.contents = QTextEdit()
        layout.addWidget(self.btn1)
        layout.addWidget(self.btn)
        layout.addWidget(self.lb1)
        layout.addWidget(self.contents)
        self.setLayout(layout)

    def getPic(self):
        # 下划线是一定要加的，不然会出错
        fname, _ = QFileDialog.getOpenFileName(self, "Open File", "c:\\","Image Files (*.jpg *.png
")
        self.lb1.setPixmap(QPixmap(fname))

    def getTxt(self):
        dig = QFileDialog()
        dig.setFileMode(QFileDialog.AnyFile)
        dig.setFilter(QDir.Files)

        if dig.exec_():
            filename = dig.selectedFiles()
            f = open(filename[0], 'r')
            with f:
                data = f.read()
```

```
            self.contents.setText(data)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = FileDialogDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 第一个按钮使用QFileDialog.getOpenFileName()，调用文件对话框来显示图像，并显示在第一个签中，负责打开的是C盘

```
def getPic(self):
    # 下划线是一定要加的，不然会出错
    fname, _ = QFileDialog.getOpenFileName(self, "Open File", "c:\\","Image Files (*.jpg *.png)")
    # 这里就是在标签上加载图片资源
    self.lb1.setPixmap(QPixmap(fname))
```

_注意_fname, _ = QFileDialog.getOpenFileName()中的下划线是必不可少的，不然程序会报错

参数分析

1. 第一个参数是指定父级组件
2. 第二个参数是QFileDialog对话框的标题
3. 第三个参数是对话框显示时默认打开的目录

- .　　代表程序运行目录
- /　　代表当前盘下的根目录

4. 第四个参数是对话框中文件扩展名的过滤器(Filter)，比如使用"Image Files(*.jpg *.png)"表示只能示扩展名为.jpg或者.png的文件
5. 第二个按钮使用文件对话框QFileDialog对象的exec方法来选择文件，并把所选择的文件的内容显在多行文本编辑框中

```
def getTxt(self):
    dig = QFileDialog()
    dig.setFileMode(QFileDialog.AnyFile)
    dig.setFilter(QDir.Files)

    if dig.exec_():
        filename = dig.selectedFiles()
        f = open(filename[0], 'r', encoding="utf-8")
        with f:
            data = f.read()
            self.contents.setText(data)
```

4.10 窗口绘图类控件

1. 主要为三大类

- QPainter　　画布

- QPen　　画笔
  - QBrush　　画刷

2. QPixmap的作用是加载并呈现本地图像，而图像的呈现本质也是通过绘图的方式来实现的，所以QPxmap也可以视为绘图的一个类

4.10.1 QPainter---自我理解为画布

1. QPainter类在QWidget控件上执行绘图操作，可以绘制从简单的直线到复杂的饼图

2. 绘制操作在QWidget.paintEvent()中完成

3. 绘制方法必须在QtGui.QPainter对象的begin()和end()之间

常用的方法

4. begin()　　开始在目标设备上绘制

5. drawArc()　　在起始角度和最终角度之间画弧

6. drawEllipse()　　在一个矩形内画圆

7. drawLine(int x1,int y1,int x2,int y2)　　绘制一条指定的端点坐标的线(x1,y1)--->(x2,y2)，并此时的画笔位置在(x2,y2)

8. drawPixmap()　　从图像文件中提取Pixmap并将其显示在指定的位置上

9. drawPolygon()　　使用坐标数组绘制多边形

10. drawRect(int x,inty,int w,int h) 在坐标(x,y)处为矩形的左上角起点绘制宽高w、h的矩形

11. drawText()　　在给定坐标处绘制显示的文字

12. fillRect()　　使用QColor参数填充矩形的颜色

13. setBrush()　　设置画刷的样式

14. setPen()　　设置画笔的样式

-------设置画笔的风格(PenStyle)

- Qt.NoPen　　没有线
- Qt.SolidLine　　一条简单的实线
- Qt.DashLine　　像素分割的短线
- Qt.DotLine　　像素分割的点
- Qt.DashDotLine　轮流交替的点和短线
- Qt.DashDotDotLine一条短线，两个点，重复交替
- Qt.MPenStyle　　画笔风格的掩码

案例4-25 绘制文字

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

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

```
        super().__init__()
        self.setWindowTitle("案例4-25 绘制文字")
        self.resize(300, 200)
        self.text = "欢迎学习 PyQt5"

    def paintEvent(self, event):
        painter = QPainter(self)
        painter.begin(self)
        # 自定义绘制方法
        self.drawText(event, painter)
        painter.end()

    def drawText(self, event, painter):
        # 设置画笔的颜色
        painter.setPen(QColor(168, 34, 3))
        # 设置字体
        painter.setFont(QFont('SimSun', 20))
        # 绘制文字
        painter.drawText(event.rect(), Qt.AlignCenter, self.text)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = DrawingTextDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 定义一个绘制事件，所有的绘制操作都发生在此事件中

```
def paintEvent(self, event):
    painter = QPainter(self)
    painter.begin(self)
    # 自定义绘制方法
    self.drawText(event, painter)
    painter.end()
```

2. QtGui.QPainter类负责所有低级别的绘制，所有的绘制方法都要放在begin和end之间，此处放置绘制方法是drawText

```
def drawText(self, event, painter):
    # 设置画笔的颜色
    painter.setPen(QColor(168, 34, 3))
    # 设置字体
    painter.setFont(QFont('SimSun', 20))
    # 绘制文字
    painter.drawText(event.rect(), Qt.AlignCenter, self.text)
```

案例4-26 绘制点

```
import sys, math
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *
```

```
class DrawingPointDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-26 绘制点")

        self.resize(300, 200)
    def paintEvent(self, event):
        # 初始化绘制工具
        painter = QPainter(self)
        # 开始在窗口中进行绘制
        painter.begin(self)
        # 自定义画点方法

        self.drawPointers(painter)
        painter.end()
    def drawPointers(self, painter):
        painter.setPen(Qt.red)
        size = self.size()

        for i in range(1000):
            # 绘制正弦函数图形，周期是[-100, 100]
            x = 100*(-1+2.0*i/1000) + size.width()/2.0
            y = -50*math.sin((x-size.width()/2.0)*math.pi/50)+size.height()/2.0
            painter.drawPoint(x, y)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = DrawingPointDemo()
    win.show()
    sys.exit(app.exec_()
```

解析

1. 画笔设置为红色，使用预定义的Qt.red颜色

```
painter.setPen(Qt.red)
```

2. 使用drawPoint()方法绘制一个个的点

```
painter.drawPoint(x, y)
```

4.10.2 QPen

1. QPen钢笔是一个基本的图形对象，用于绘制直线、曲线或者轮廓线画出矩形，椭圆形、多边形以其他形状等

案例4-27 QPen钢笔的使用

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

```
class PenDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-27 QPen钢笔的使用")
        self.setGeometry(300, 300, 280, 270)

    def paintEvent(self, Event):
        painter = QPainter()
        painter.begin(self)
        self.drawLines(painter)
        painter.end()
    def drawLines(self, painter):
        # pen的样式是线条黑色  2px  实线
        pen = QPen(Qt.black, 2, Qt.SolidLine)
        painter.setPen(pen)
        painter.drawLine(20, 40, 250, 40)

        pen.setStyle(Qt.DashLine)
        painter.setPen(pen)
        painter.drawLine(20, 120, 250, 120)

        pen.setStyle(Qt.DotLine)
        painter.setPen(pen)
        painter.drawLine(20, 160, 250, 160)

        pen.setStyle(Qt.DashDotDotLine)
        painter.setPen(pen)
        painter.drawLine(20, 200, 250, 200)

        pen.setStyle(Qt.CustomDashLine)
        pen.setDashPattern([1, 4, 5, 4])
        painter.setPen(pen)
        painter.drawLine(20, 240, 250, 240)
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = PenDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 创建一个QPen对象，设置颜色和线条样式(黑色 2px 实线)

```
pen = QPen(Qt.black, 2, Qt.SolidLine)
```

2. 自定义一条线条的样式，使用的数字列表定义样式　数字列表的个数必须是偶数个，[1,4,5,4]代表是1px的横线宽度，4px的间隔，5px的线条宽度，4px的间隔-----也就是说这样自定义的样式就是1p和5px的短线以4px的间隔为空余交替出现

```
pen.setStyle(Qt.CustomDashLine)
pen.setDashPattern([1, 4, 5, 4])
painter.setPen(pen)
painter.drawLine(20, 240, 250, 240)
```

4.10.3 QBrush

1. QBrush画刷是一个基本的图形对象，用于填充如矩形，椭圆或者多边形等的形状
2. QBrush有三种基本类型

- 预定义
- 过渡
- 纹理图案

案例4-28 QBrush画刷的使用

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

class BrushDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-28 QBrush画刷的使用")
        self.setGeometry(300, 300, 365, 280)

    def paintEvent(self, event):
        painter = QPainter()
        painter.begin(self)
        self.drawLines(painter)
        painter.end()

    def drawLines(self, painter):

        brush = QBrush(Qt.SolidPattern)
        painter.setBrush(brush)
        painter.drawRect(10, 15, 90, 60)

        brush = QBrush(Qt.Dense1Pattern)
        painter.setBrush(brush)
        painter.drawRect(130, 15, 90, 60)

        brush = QBrush(Qt.Dense2Pattern)
        painter.setBrush(brush)
        painter.drawRect(250, 15, 90, 60)

        brush = QBrush(Qt.Dense3Pattern)
        painter.setBrush(brush)
        painter.drawRect(10, 105, 90, 60)

        brush = QBrush(Qt.DiagCrossPattern)
        painter.setBrush(brush)
        painter.drawRect(10, 105, 90, 60)

        brush = QBrush(Qt.Dense5Pattern)
        painter.setBrush(brush)
        painter.drawRect(130, 105, 90, 60)
```

```
        brush = QBrush(Qt.Dense6Pattern)
        painter.setBrush(brush)
        painter.drawRect(250, 105, 90, 60)

        brush = QBrush(Qt.HorPattern)
        painter.setBrush(brush)
        painter.drawRect(10, 195, 90, 60)

        brush = QBrush(Qt.VerPattern)
        painter.setBrush(brush)
        painter.drawRect(130, 195, 90, 60)

        brush = QBrush(Qt.BDiagPattern)
        painter.setBrush(brush)
        painter.drawRect(250, 195, 90, 60)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = BrushDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 使用了9种不同的背景填充矩形，定义QBrush对象，通过QPainter对象的画刷设置，最后调用draRect()方法绘制矩形

```
brush = QBrush(Qt.SolidPattern)
painter.setBrush(brush)
painter.drawRect(10, 15, 90, 60)
```

4.10.4 QPixmap 图片像素映射

1. QPixmap类常用于绘图设备的图像显示，它可以作为对象加载到控件上，通常是标签或者按钮，于在标签或者按钮上显示图像

2. QPixmap可以读取的图像文件的类型有BMP/JPG/GIF/JPEG/PNG/等

3. **常用方法**

- copy()　　从QRect对象复制到QPixmap对象
- fromImage()　将QImage对象转换为QPixmap对象
- toImage()　　将QPixmap对象转换为QImage对象
- grabWidget() 从给定的窗口小部件创建一个像素图
- grabWindow() 在窗口中创建数据的像素图
- load()　　加载图片文件为QPixmap对象
- save()　　将QPixmap对象保存为文件

案例4-29 QPixmap像素映射的使用

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

class PixmapDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-29 QPixmap像素映射的使用")
        self.lb1 = QLabel()
        self.lb1.setPixmap(QPixmap("./images/python.jpg"))
        layout = QVBoxLayout()
        layout.addWidget(self.lb1)
        self.setLayout(layout)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = PixmapDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 在这个例子中，使用setPixmap()将图像显示在QLabel上

```
self.lb1 = QLabel()
self.lb1.setPixmap(QPixmap("./images/python.jpg"))
```

4.11 拖拽与剪切板

4.11.1 Drag 与 Drop

1. 基于MIME类型的拖拽数据传输就是基于QDrag类的

2. QMimeData对象将关联的数据与其对应的MIME类型相互关联

3. MIME类型的数据可以简单理解为互联网上的各种资源，比如文本、音频、和视频等资源，互联网的每一种资源都属于一种MIME类型的数据

QMimeData类函数

- hasText()　text()　text/plain
- hasHtml()　html()　text/html
- hasUrls()　urls()　text/uri-list
- hasImage()　imageData()　image/*
- hasColor()　colorData()　application/x-color

4. 许多QWidget对象都支持拖拽动作，允许拖拽数据的控件必须设置QWidget.setDragEnabled()为True

常用的拖拽事件

5. DragEnterEvent()　当执行一个拖拽控件操作时，并且鼠标指针进入改控件时，这个事件将被触发

_注意点_在这个事件中可以获取被操作的窗口控件，还可以有条件拒绝或者接收该控件的拖拽操作

6. DragMoveEvent()　当拖拽操作进行时会触发该事件

7. DragLeaveEvent()　当执行一个拖拽控件操作时，并且鼠标指针离开时触发

8. DropEvent()　当拖拽操作在目标控件上被释放时，这个事件被触发

案例4-30 拖拽功能的实现

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

class Combo(QComboBox):
    def __init__(self, title, parent):
        super().__init__()
        self.setAcceptDrops(True)
    def dragEnterEvent(self, event):
        print(event)
        # 判断拖拽的数据是不是text类型的
        # 有条件拒绝或者接收该控件的拖拽操作
        if event.mimeData().hasText():
            event.accept()
        else:
            event.ignore()
    # 释放拖拽操作事件被触发的前提一定是被操作控件通过检测后允许当前的拖拽行为
    # event.accept()这句代码不执行也就不会有释放控件事件的触发
    def dropEvent(self, event):
        print(event)
        self.addItem(event.mimeData().text())


class DragDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-30 拖拽功能的实现")
        layout = QFormLayout()
        edt1 = QLineEdit()
        edt1.setDragEnabled(True)
        com = Combo("Button", self)
        layout.addRow(edt1, com)
        self.setLayout(layout)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    drag = DragDemo()
    drag.show()
    sys.exit(app.exec_())
```

解析

1. DragEnterEvent会验证事件的MIME数据的是否包含字符串文本，如果包含，就接收该事件

2. dropEvent会在接收该事件后响应释放拖拽的操作，并将拖拽的数据文本作为下拉列表框的item

```
def dragEnterEvent(self, event):
    print(event)
    if event.mimeData().hasText():
        event.accept()
    else:
        event.ignore()
def dropEvent(self, event):
    print(event)
    self.addItem(event.mimeData().text())
```

4.11.2 QClipboard

1. QClipboard类提供了对系统剪切板的访问，可以在应用程序之间复制粘贴，类似QDrag类
2. QApplication类有一个静态方法clipboard()，它返回剪切板对象
3. 任何类型的MimeData数据都可以从剪切板上复制以及粘贴

常用的方法

4. clear()　　清除剪切板的内容
5. setImage()　　将QImage对象复制到剪切板上
6. setMimeData()　　将Mime数据设置到剪切板
7. setPixmap()　　将QPixmap对象复制到剪切板上
8. setText()　　将text文本复制到剪切板上
9. text()　　获取剪切板上的文本数据text

.....

QClipboard常用的信号

10. dataChanged　当剪切板内容发生改变时，这个信号就会被发射

案例4-31 QClipboard剪切板的使用

```
import sys, os
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *

class clipboardDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-31 QClipboard剪切板的使用")
        layout = QGridLayout()
        textCopyButton = QPushButton("&Copy Text")
        textPasteButton = QPushButton("Paste &Text")
        htmlCopyButton = QPushButton("C&opy Html")
        htmlPasteButton = QPushButton("Paste &Html")
        imageCopyButton = QPushButton("Co&py Image")
        imagePasteButton = QPushButton("Paste &Image")

        self.textLabel = QLabel("Original text")
        self.imageLabel = QLabel()
        self.imageLabel.setPixmap(QPixmap(os.path.join(os.path.dirname(__file__), 'images/new.
```

```
ng')))
        layout.addWidget(textCopyButton, 0, 0)
        layout.addWidget(imageCopyButton, 0, 1)
        layout.addWidget(htmlCopyButton, 0, 2)
        layout.addWidget(textPasteButton, 1, 0)
        layout.addWidget(imagePasteButton, 1, 1)
        layout.addWidget(htmlPasteButton, 1, 2)

        layout.addWidget(self.textLabel, 2, 0, 1, 2)
        layout.addWidget(self.imageLabel, 2, 2)

        self.setLayout(layout)
        textCopyButton.clicked.connect(self.copyText)
        textPasteButton.clicked.connect(self.pasteText)
        imageCopyButton.clicked.connect(self.copyImage)
        imagePasteButton.clicked.connect(self.pasteImage)
        htmlCopyButton.clicked.connect(self.copyHtml)
        htmlPasteButton.clicked.connect(self.pasteHtml)

    def copyText(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        clipboard.setText("我的是被复制的文本text")

    def pasteText(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        self.textLabel.setText(clipboard.text())

    def copyImage(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        clipboard.setPixmap(QPixmap(os.path.join(os.path.dirname(__file__), 'images/python.png'
))


    def pasteImage(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        self.imageLabel.setPixmap(clipboard.pixmap())

    def copyHtml(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        mimaData = QMimeData()
        mimaData.setHtml("<b>wbj520<font color=red>mumu</font></b>")
        clipboard.setMimeData(mimaData)


    def pasteHtml(self):
        # 获取剪切板对象
        clipboard = QApplication.clipboard()
        mimeData = clipboard.mimeData()
        if mimeData.hasHtml():
```

```
            self.textLabel.setText(mimeData.html())

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = clipboardDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 首先需要获取到clipboard对象

```
clipboard = QApplication.clipboard()
clipboard.setText("我的是被复制的文本text")
```

2. 剪切板操作常用方法中没有操作html的，但是上一节我们了解到所有的资源数据都对应着MIME数类型，所以可以使用MimeData数据

```
def copyHtml(self):
    # 获取剪切板对象
    clipboard = QApplication.clipboard()
    mimaData = QMimeData()
    mimaData.setHtml("<b>wbj520<font color=red>mumu</font></b>")
    clipboard.setMimeData(mimaData)

def pasteHtml(self):
    # 获取剪切板对象
    clipboard = QApplication.clipboard()
    mimeData = clipboard.mimeData()
    if mimeData.hasHtml():
        self.textLabel.setText(mimeData.html())
```

4.12 日历与时间

4.12.1 QCalendar

1. QCalendar是一个日历控件，是一个基于月份的视图，允许用户通过鼠标或者键盘来选择日期，默是选中今天的日期，也可以规定日期的范围

2. **常用的方法**

- setDataRange()　　设置日期返回以供选择
- setFirstDayOfWeek() 设置一周的第一天，默认是周日
    - Qt.Monday　周一

.....以此类推

- setMinimumDate()　　设置最小日期
- setMaximumDate()　　设置最大日期
- setSelectedDate()　　设置选中的日期，默认是今天
- maximumDate　　获取日历控件的最大日期

- minimumDate　　获取日历控件的最小日期
  - selectedDate　　获取当前选中的日期
  - setGridvisible()　　设置日历控件是否显示网格(网格可视化)

案例4-32 QCalendar日历的使用

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

class calendarDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-32 QCalendar日历的使用")
        self.initUI()

    def initUI(self):
        self.cal = QCalendarWidget(self)
        self.cal.setMinimumDate(QDate(1970, 1, 1))
        self.cal.setMaximumDate(QDate(3000, 1, 1))
        self.cal.setGridVisible(True)
        self.move(20, 20)
        # 表示日历控件中的日期被点击 [取属性对象]
        self.cal.clicked[QDate].connect(self.showDate)
        self.lb1 = QLabel(self)
        date = self.cal.selectedDate()
        # 将日期数据转换为字符串并且进行格式化操作
        self.lb1.setText(date.toString("yyyy-MM-dd dddd"))
        self.lb1.move(20, 250)
        self.setGeometry(100, 100, 400, 300)

    def showDate(self, date):
        print(date)
        print(date.toString("yyyy-MM-dd dddd"))
        self.lb1.setText(date.toString("yyyy-MM-dd dddd"))
```

解析

1. 创建日历组件，并设置最大最小日期

```
self.cal = QCalendarWidget(self)
self.cal.setMinimumDate(QDate(1970, 1, 1))
self.cal.setMaximumDate(QDate(3000, 1, 1))
self.cal.setGridVisible(True)
```

2. 从窗口组件中选定一个日期，会发射一个QDate点击的信号，捕捉到后连接到自定义的槽函数上

```
# 表示日历控件中的日期被点击 [取属性对象]
self.cal.clicked[QDate].connect(self.showDate)
```

3. 通过调用selectedDate方法检索所选定的日期，然后将日期对象转换为指定格式的字符串并显示在签上

```
def showDate(self, date):
    print(date)
    print(date.toString("yyyy-MM-dd dddd"))
    self.lb1.setText(date.toString("yyyy-MM-dd dddd"))
```

4.12.2 QDateTimeEdit

1. QDateTimeEdit 是一个允许用户编辑日期时间的控件，可以使用键盘上下箭头按钮增加或者减少期时间值

2. QDateTimeEdit通过setDisplayFormat来设置显示的日期时间格式

3. **常用的方法**

- setDisplayFormat()　设置日期时间显示格式
  - yyyy 代表年份，用4位数表示
  - MM　代表月份，取值是01-12
  - dd　代表日，取值是01-31
  - HH　代表小时，取值是00-23
  - mm　代表分钟，取值是00-59
  - ss　代表秒，取值是00-59
- setMinimumDate　设置控件的最小日期
- setMaximumDate　设置控件的最大日期
- time()　　获取编辑的时间
- date()　　获取编辑的日期

4. **常用的信号**

- dateChanged　　当日期改变时发射该信号
- dateTimeCahnged 当日期时间改变时发射该信号
- timeCahnged　　当时间改变时发射该信号

5. QDateTimeEdit的子类

- QDateEdit和QTimeEdit类均继承自QDateTimeEdit类
- 继承关系是QWidget---QAbstractSpinBox---QDateTimeEdit---(QDateEdit，QTimeEdit)
- 日期操作是QDateEdit，时间操作是QTimeEdit，日期时间操作是QDateTimeEdit

```
dateEdit = QDateEdit(self)
timeEdie = QTimeEdit(self)
dateEdit.setDisplayFormat("yyyy-MM-dd")
timeEdie.setDisplayFormat("HH:mm:ss")
```

- 设置弹出日历时要注意，用来弹出日历的类只能是QDateTimeEdit和QDateEdit， QTimeEdit语法上是可以的，但是不起作用不会弹出来

```
dateTimeEdit = QDateTimeEdit(self)
dateEdit = QDateEdit(self)
```

```
dateTimeEdit.setCalendarPopup(True)
dateEdit.setCalendarPopup(True)
```

6. 初始化 QDateTimeEdit类

案例 QDateTimeEdit的简单使用

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

class DateTimeEditDemo1(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例 QDateTimeEdit的简单使用")
        layout = QVBoxLayout()
        dateTimeEdit = QDateTimeEdit()
        dateTimeEdit2 = QDateTimeEdit(QDateTime.currentDateTime())
        # 设置日期时间的显示格式
        dateTimeEdit2.setDisplayFormat("yyyy-MM-dd HH:mm:ss")
        # 设置最小日期
        dateTimeEdit2.setMinimumDate(QDate.currentDate().addDays(-365))
        # 设置最大日期
        dateTimeEdit2.setMaximumDate(QDate.currentDate().addDays(365))
        # 设置弹出日历控件
        dateTimeEdit2.setCalendarPopup(True)

        dateEdit  = QDateEdit(QDate.currentDate())
        timeEdit = QTimeEdit(QTime.currentTime())
        layout.addWidget(dateTimeEdit)
        layout.addWidget(dateTimeEdit2)
        layout.addWidget(dateEdit)
        layout.addWidget(timeEdit)
        self.setLayout(layout)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = DateTimeEditDemo1()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 设置日期时间的显示格式

```
dateTimeEdit2.setDisplayFormat("yyyy-MM-dd HH:mm:ss")
```

2. 设置日期时间的范围，限制可选日期范围为距离今天的前后365天

```
# 设置最小日期
dateTimeEdit2.setMinimumDate(QDate.currentDate().addDays(-365))
# 设置最大日期
dateTimeEdit2.setMaximumDate(QDate.currentDate().addDays(365))
```

3. 弹出日历控件，默认是通过计数器的上下箭头改变数据的，现在可以弹出日历控件，　只需要调用面的一行代码

```
# 设置弹出日历控件
dateTimeEdit2.setCalendarPopup(True)
```

总结

1. 可以通过date()/dateTime()等方法来获取日期时间对象

2. 可以调用QDate的year()/month()/day()等函数来获取详细的年月日

案例4-33 QDateTimeEdit的使用

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

class DateTimeDemo(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-33 QDateTimeEdit的使用")
        self.initUI()

    def initUI(self):
        self.resize(300, 90)
        layout = QVBoxLayout()
        self.dateEdit = QDateTimeEdit(QDateTime.currentDateTime(), self)
        self.dateEdit.setDisplayFormat("yyyy-MM-dd HH:mm:ss")

        # 设置最小日期
        self.dateEdit.setMinimumDate(QDate.currentDate().addDays(-365))
        # 设置最大日期
        self.dateEdit.setMaximumDate(QDate.currentDate().addDays(365))
        self.dateEdit.setCalendarPopup(True)

        self.dateEdit.dateChanged.connect(self.onDateChanged)
        self.dateEdit.dateTimeChanged.connect(self.ondateTimeChanged)
        self.dateEdit.timeChanged.connect(self.onTimeChanged)

        self.btn = QPushButton("获取日期和时间")
        self.btn.clicked.connect(self.onButtonClicked)

        layout.addWidget(self.dateEdit)
        layout.addWidget(self.btn)
        self.setLayout(layout)

    # 日期发生改变时
    def onDateChanged(self, date):
        print(date)
    # 日期时间发生改变时
    def ondateTimeChanged(self, dateTime):
        print(dateTime)
    # 时间发生改变时
```

```
    def onTimeChanged(self, time):
        print(time)
    # 按钮被点击时
    def onButtonClicked(self):
        dateTime = self.dateEdit.dateTime()
        maxDate = self.dateEdit.maximumDate()
        minDate = self.dateEdit.minimumDate()
        maxDateTime = self.dateEdit.maximumDateTime()
        minDateTime = self.dateEdit.minimumDateTime()
        maxTime = self.dateEdit.maximumTime()
        minTime = self.dateEdit.minimumTime()
        print('\n选择日期时间')
        print('dateTime=%s' % dateTime.toString())
        print('maxDate=%s' % maxDate.toString())
        print('minDate=%s' % minDate.toString())
        print('maxDateTime=%s' % maxDateTime.toString())
        print('minDateTime=%s' % minDateTime.toString())
        print('maxTime=%s' % maxTime.toString())
        print('minTime=%s' % minTime.toString())


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = DateTimeDemo()
    win.show()
    sys.exit(app.exec_())
```

4.13 菜单栏、工具栏与状态栏

4.13.1 菜单栏

1. 在QMainWindow对象的标题栏下方，水平的QMenuBar被保留显示QMenu对象
2. QMenu类提供了一个可以添加到菜单栏的小控件，也用于创建上下文菜单和弹出子菜单
3. 每一个QMenu对象都可以包含一个或者多个QAction对象或者级联的QMenu对象
4. 可以使用createPopupMenu函数来创建弹出子菜单
5. menuBar函数用于返回主窗口的QMenuBar对象
6. 通过addAction函数可以在菜单中进行添加操作，addMenu函数可以将菜单添加到菜单栏中

常用的方法

7. menuBar()　　返回主窗口的QMenuBar对象
8. addMenu()　　在菜单栏中添加一个新的菜单
9. addAction()　在菜单中添加一个新的操作，包含文本或者图标
10. setEnabled()　设置菜单中的操作是否启用
11. addSeperator()　在菜单中添加一个分割线
12. clear()　　删除菜单、菜单栏的内容
13. setShortcut()　设置操作按钮的快捷方式

14. setText()　　设置每个菜单项的文本

15. setTitle()　　设置QMenu小控件的标题

16. text()　　获取每一个菜单项的文本

17. title()　　获取QMenu小控件的标题

常用的信号

18. triggered　　单击任何QAction按钮时，相应的QMenu对象都会发射triggered信号

案例4-34 QMenuBar菜单栏的使用

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

class MenuBarDemo(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-34 QMenuBar菜单栏的使用")
        layout = QHBoxLayout()
        bar = self.menuBar()
        file = bar.addMenu("File")
        file.addAction("New")
        save = QAction("Save", self)
        # 添加快捷方式
        save.setShortcut("Ctrl+S")
        file.addAction(save)
        # 给file菜单添加级联菜单
        edit = file.addMenu("Edit")
        edit.addAction("Copy")
        edit.addAction("Paste")
        quit = QAction("Quit", self)
        file.addAction(quit)

        file.triggered[QAction].connect(self.processTrigger)
        self.setLayout(layout)

    def processTrigger(self, sel):
        print(sel.text()+"  is triggered")
if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = MenuBarDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 顶级窗口必须是QMainWindow对象，才可以引用QMenuBar对象

2. 通过addMenu方法将file菜单添加到菜单栏上

```
bar = self.menuBar()
file = bar.addMenu("File")
```

3. 菜单栏中的操作按钮可以是字符串或者QAction对象

```
# 字符串文本
file.addAction("New")
# QAction对象
save = QAction("Save", self)
# 添加快捷方式
save.setShortcut("Ctrl+S")
file.addAction(save)
```

4. 菜单发射triggered信号，将信号连接到槽函数，该信号是点击QAction对象触发的

```
file.triggered[QAction].connect(self.processTrigger)
```

4.13.2 QToolBar工具栏

1. QToolBar控件是由文本控件、图标或其他小控件按钮组成的可移动面板，通常位于菜单栏下面

2. **常用的方法**

- addAction()　添加具有文本或者图标的工具按钮
- addSeperator() 添加分割线分组显示工具按钮
- addWidget()　添加工具栏中按钮以外的控件
- addToolBar()　使用QMainWindow类的方法添加一个新的工具栏
- setMovable() 设置工具栏是否可移动
- setOrientation()设置工具栏的显示方向
  - Qt.Horizontal 水平显示
  - Qt.Vertical　垂直显示

3. **常用的信号**

1. actionTriggered　当点击工具栏上的按钮时，触发该信号，并且参数传递该QAction对象的用到槽函数

案例4-35 QToolBar工具栏的使用

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

class ToolBarDemo(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-35 QToolBar工具栏的使用")
        self.resize(300, 200)

        layout = QVBoxLayout()
        toolBar = self.addToolBar("File")
        new = QAction(QIcon("./images/new.png"), "new", self)
        toolBar.addAction(new)
```

```
        open = QAction(QIcon("./images/open.png"), "open", self)
        toolBar.addAction(open)

        save = QAction(QIcon("./images/save.png"), "save", self)
        toolBar.addAction(save)

        toolBar.actionTriggered[QAction].connect(self.toolbtnPressed)
        toolBar.setMovable(True)
        self.setLayout(layout)

    def toolbtnPressed(self, action):
        print("pressed tool button is ", action.text())


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = ToolBarDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 调用QMainWindow窗口创建一个新的工具栏

```
toolBar = self.addToolBar("File")
```

2. 将具有图形和文本名称的QAction对象添加到工具栏中

```
open = QAction(QIcon("./images/open.png"), "open", self)
toolBar.addAction(open)
```

3. 将actionTriggered的信号连接到槽函数toolbtnPressed上

```
toolBar.actionTriggered[QAction].connect(self.toolbtnPressed)
```

4.13.3 QStatusBar状态栏

1. QMainWindow对象在底部保留一个水平条，作为状态栏(QStatusBar)，用于永久或者临时显示状信息

2. **常用的方法**

- addWidget()　　在状态栏中添加给定的窗口部件对象
- addPermanentWidget() 在状态栏中添加永久的小部件对象
- showMessage()　　在状态栏显示一条临时信息指定时间间隔
- clearMessage()　　删除状态栏中显示的信息
- removeWidget()　　从状态栏中删除指定的小控件
- setStatusBar()　　设置一个状态栏

案例4-36 QStatusBar状态栏的使用

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

class StatusBarDemo(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-36 QStatusBar状态栏的使用")
        menuBar = self.menuBar()
        file = menuBar.addMenu("File")
        file.addAction("show")
        file.triggered[QAction].connect(self.actionPressed)
        self.setCentralWidget(QTextEdit())
        self.statusBar = QStatusBar()
        self.setStatusBar(self.statusBar)

    def actionPressed(self, action):
        if action.text() == "show":
            # 默认显示时间为0也就是永久显示  此处设置显示5s 之后消失
            self.statusBar.showMessage(action.text()+"菜单栏选项被点击了", 5000)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = StatusBarDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 在这个例子中顶层窗口 MainWindow有一个菜单栏和一个QTextEdit控件作为窗口的中心部件

2. 当单击MenuBar中的菜单项show时就会触发triggered信号，并且连接槽函数actionPressed，将击的菜单项信息显示在状态栏中，持续5s

```
file.triggered[QAction].connect(self.actionPressed)
def actionPressed(self, action):
    if action.text() == "show":
        # 默认显示时间为0也就是永久显示  此处设置显示5s 之后消失
        self.statusBar.showMessage(action.text()+"菜单栏选项被点击了", 5000)
```

4.14 QPrinter

1. 打印图像是图像处理软件中的一个常用的功能，打印图像实际上是在QPaintDevice中画图

2. 打印时使用的是QPrinter对象，本质跟之前的QPixmap差不多，也是一种PaintDevice(绘图设备)

案例4-37 QPrinter打印图像的使用

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

```
class printerDemo(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("案例4-37 QPrinter打印图像的使用")

        self.ImageLabel = QLabel()
        self.ImageLabel.setSizePolicy(QSizePolicy.Ignored, QSizePolicy.Ignored)
        self.setCentralWidget(self.ImageLabel)
        self.image = QImage()
        self.createActions()
        self.createMenus()
        self.createToolBars()

        if self.image.load("./images/screen.png"):
            # image对象转换为pixmap对象
            self.ImageLabel.setPixmap(QPixmap.fromImage(self.image))
            self.resize(self.image.width(), self.image.height())

    def createActions(self):
        # self.tr()表示的是多语言国际化
        self.PrintAction = QAction(QIcon("./images/printer.png"), self.tr("打印"), self)
        self.PrintAction.setShortcut("Ctrl+P")
        self.PrintAction.triggered.connect(self.slotPrint)

    def createMenus(self):
        PrintMemu = self.menuBar().addMenu(self.tr("打印"))
        PrintMemu.addAction(self.PrintAction)

    def createToolBars(self):
        fileToolBar = self.addToolBar("Print")
        fileToolBar.addAction(self.PrintAction)

    def slotPrint(self):
        printer = QPrinter()
        printerDialog = QPrintDialog(printer, self)
        if printerDialog.exec_():
            painter = QPainter(printer)
            # 得到绘图的视图大小
            rect = painter.viewport()
            size = self.image.size()
            # 设置缩放比例为视图大小，并且保持宽高比
            size.scale(rect.size(), Qt.KeepAspectRatio)
            painter.setViewport((rect.x(), rect.y(), size.width(), size.height()))
            painter.setWindow(self.image.rect())
            painter.drawImage(0, 0, self.image)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = printerDemo()
    win.show()
    sys.exit(app.exec_())
```

解析

1. 创建了QAction对象给菜单栏和工具栏共用

```
def createActions(self):
    # self.tr()表示的是多语言国际化
    self.PrintAction = QAction(QIcon("./images/printer.png"), self.tr("打印"), self)
    self.PrintAction.setShortcut("Ctrl+P")
    self.PrintAction.triggered.connect(self.slotPrint)

def createMenus(self):
    PrintMemu = self.menuBar().addMenu(self.tr("打印"))
    PrintMemu.addAction(self.PrintAction)

def createToolBars(self):
    fileToolBar = self.addToolBar("Print")
    fileToolBar.addAction(self.PrintAction)
```

2. 多语言国际化编写字符串

```
self.tr("打印")
```

3. 缩放时设置宽高大小时，还要设置宽高比的模式，此处是保持宽高比

```
size.scale(rect.size(), Qt.KeepAspectRatio)
```