

# PyQT5 使用手册

## 作者:向仔州

PyQT5 环境搭建,pycharIDE 版本.....	11
Sublime text3 运行 PyQT5.....	13
VScode 搭建pyQT5.....	14
QObject()类使用.....	17
给对象添加属性, 使用到该对象的时候可以获取自定义属性.....	17
Qobject.setProperty(“属性名”, “值”)//设置对象属性名函数.....	17
Qobject.Property(“属性名”)//返回对象属性对应的值.....	17
给本对象设置父对象.....	18
Qobject.setParent(传入对象) //设置父对象函数使用.....	18
QObject 设置对象的父子关系实际应用.....	18
QObject.children 函数使用, 获取本对象下面的子对象.....	19
QObject.findChildren(对象参数) //查看本对象下面有多少个子对象.....	20
这种对象父子关系查询搜索主要用于界面的内存管理.....	20
创建两个窗口来演示父子对象, 也就是父子控件的关系.....	21
QObject 信号与槽.....	22
QPushButton.clicked.connect(槽函数) //按钮信号与槽的使用.....	22
对对象类型判定的应用.....	23
对象.isWidgetType() //判断一个对象是不是控件.....	23
对象.inherits(类名) 判断对象是否继承某个类, 继承返回 True, 否则返回 False.....	23
super().__init__() 语法使用.....	24
QWidget 窗口使用.....	25
QWidget.resize(长,宽) //设置 Widget 窗口大小.....	25
QWidget.move(x,y) //移动 Widget 窗口位置.....	25
QWidget.setStyleSheet(“写入 QSS 代码”) //QSS 代码设置窗口样式, 颜色, 造型.....	25
QWidget.x() #获取窗口 x 位置,但是有个问题要注意.....	25
QWidget.y() #获取窗口 y 位置, 但是有个问题要注意.....	25
QWidget.width() #获取窗口宽度.....	25
QWidget.height() #获取窗口高度.....	25
QWidget.geometry() # geometry 读取窗口的坐标是从窗口内部计算.....	26
QWidget 鼠标操作.....	26
QWidget.setCursor(Qt.BusyCursor) //设置窗口的鼠标为繁忙的样式.....	26
QWidget.setCursor(Qt.ForbiddenCursor) //鼠标在窗口显示 xx 样式.....	27
QWidget.setCursor(Qt.CrossCursor) //鼠标在窗口显示+样式.....	27
如果系统默认鼠标样式形状不够用, 可以自己加入图片来设置鼠标样式.....	27
QCursor 鼠标类使用.....	27
cursor.pos() #获取鼠标在桌面的 xy 位置.....	29
鼠标移动过程中实时获取鼠标坐标.....	29
mouseMoveEvent 获取鼠标在窗口的相对位置.....	30
mouseMoveEvent(对象参数, 接受鼠标数据变量).localPos() #返回鼠标现在在窗口的位置.....	30
self.findChild(查找什么类)#父类里面使用的函数, 用来提取子类对象.....	30
窗口当前状态事件消息.....	31
showEvent()回调函数, closeEvent()回调函数.....	31
moveEvent 窗口移动回调函数, resizeEvent 窗口大小变化回调函数.....	31
enterEvent #鼠标进入窗口回调函数, leaveEvent #鼠标离开窗口回调函数.....	32
mousePressEvent #鼠标按下回调函数, mouseReleaseEvent #鼠标释放回调函数.....	32
mouseDoubleClickEvent #鼠标双击回调函数.....	32
keyPressEvent #键盘按下回调函数, keyReleaseEvent #键盘松开回调函数.....	33
父子窗口里面多个空间消息转发问题, 也就是消息产生后不知道转发给哪个窗口? .....	33
多窗口层级管理, 就是底层窗口显示到前面来.....	35
QLabel 使用.....	35

QLabel.lower() //将第 1 层对象向下移动一层.....	36
QLabel.raise_() //将第 2 层向上移动.....	36
QLabel.stackUnder( 放入顶层的对象 QLabel ).....	37
<b>    如果我想鼠标点击某个窗口，某个窗口就出现在顶层，如何实现？ .....</b>	<b>37</b>
<b>QIcon 使用，添加图标.....</b>	<b>38</b>
对象 = QIcon( “ 图片路径 ” ) //用 QIcon 导入图像，然后返回给对象.....	38
对象.setWindowIcon( “QIcon” ) //主窗口左上边图标，用 QIcon 对象导入.....	38
<b>QWidget 窗口透明度测试.....</b>	<b>38</b>
QWidget.setWindowOpacity( 透明度参数 )//透明度在 1~0 之间，0 全透明，1 不透明.....	38
<b>QWidget 窗口状态获取</b>	
QWidget.windowState() //获取窗口状态，窗口无状态参数 Qt.WindowNoState.....	39
<b>结合 Qt.WindowFullScreen, Qt.WindowMinimized 制作一个无边框窗口.....</b>	<b>41</b>
<b>    在无边框窗口上加控件.....</b>	<b>42</b>
QWidget.width() #获取窗口控件宽度.....	43
QPushButton.width() #获取按钮宽度.....	43
QPushButton.pressed.connect( 函数参数 )#按钮按下发送信号.....	43
QWidget.close() #关闭窗口.....	43
<b>    点击按钮，改变按钮显示状态.....</b>	<b>44</b>
QWidget.isMaximized() #获取当前窗口状态.....	44
<b>    让你的窗口或者按钮，其它控件暂时无法使用。 .....</b>	<b>44</b>
QPushButton.setEnabled(参数) #参数 false 不可点击，参数 True 可以点击默认也是可以点击.....	44
QPushButton.isEnabled() #返回按钮状态是可点击(True)还是不可点击(False).....	44
<b>    窗口编辑时状态显示，编辑后状态显示.....</b>	<b>45</b>
QWidget.setWindowModified(True) #显示 setWindowTitle 里面的[*]星星符号.....	45
QWidget.isActiveWindow() #判断窗口当前是否被激活，返回 True 或者 False.....	45
<b>    窗口界面交互设计.....</b>	<b>46</b>
<b>        监听文本框状态，看看文本框是否有字符输入？ .....</b>	<b>46</b>
QLineEdit.textChanged.connect(函数参数) #编辑框字符变化信号发送函数.....	46
<b>        如何让按钮在编辑框输入字符后才能被点击.....</b>	<b>47</b>
QPushButton.setEnabled(参数) #参数 True 按钮可以点击，False 按钮不可以点击.....	47
变量 = QLineEdit.text() #获取编辑框输入的字符，返回给变量.....	47
<b>    鼠标移动到窗口的时候，状态栏显示提示信息.....</b>	<b>48</b>
<b>        用 QWidget 做窗口状态栏会编译报错，因为窗口状态栏是 QMainWindow 类实现的.....</b>	<b>48</b>
QMainWindow().statusBar() //给窗口设置状态栏.....	48
QMainWindow().setStatusTip( “参数” )//设置鼠标移动到窗口，状态栏显示得字符串.....	48
<b>        鼠标移动到窗口里的控件，状态栏显示控件提示.....</b>	<b>48</b>
QLabel.setStatusTip( “参数” )//设置鼠标移动到标签，状态栏显示字符.....	48
QLabel.setToolTip( “参数” ) //设置鼠标移动到标签，在标签旁边显示提示符.....	49
QLabel.setToolTipDuration(延时时间) //鼠标移动到标签，提示显示持续多久.....	49
<b>    QPushButton 控件实现的底层原理，QAbstractButton 按钮抽象基类.....</b>	<b>50</b>
画家 = QPainter(参数) #创建画家，返回画家对象。参数:表示在什么对象上画画.....	51
笔对象 = QPen(颜色，画笔粗细) #创建画笔，返回画笔对象.....	51
QPainter().setPen(笔对象) #将画笔拿给画家.....	51
QPainter().drawEllipse(起始坐标 x, 起始坐标 y, 方形长, 方形宽) #系统会自动在这个方形里面做内切椭圆.....	51
<b>    QAbstractButton 也是鼠标点击对象之后，自带信号功能的.....</b>	<b>52</b>
<b>    QPushButton 点击后自己修改自己.....</b>	<b>53</b>
int 变量 = int(参数) #将字符串转换成十进制。参数:填入字符串.....	53
str 字符串变量 = str(参数) #将十进制转换成字符串，参数: 填入十进制.....	53
<b>    QPushButton 显示图标.....</b>	<b>53</b>
返回图标对象 = QIcon( “路径字符” ) #路径字符就是你的图标位置.....	53
QPushButton.setIcon(图标对象) #图标对象就是上面返回的对象.....	53
QPushButton.setIconSize(QSize(长, 宽)) #设置图标大小.....	54
QPushButton.setAutoRepeat(True) #按住按钮不放，就可以一直重复发按钮信号.....	54
<b>    让控件处于自动按下状态，而不是鼠标去点击后才按下.....</b>	<b>55</b>

控件.setDown(参数) #参数为 True, 控件自动处于鼠标按住状态, 记住是按住, 还没有放开	55
其实鼠标按下功能也是 QAbstractButton 具备的, 只是下面三个控件继承过来了。	55
变量 = 对象.isCheckable() #返回对象是否可以被选中能被选中返回 True 不能选中返回 false	55
对象.setCheckable(True) #使控件具备可以被选中功能	56
对象.toggle() #控件状态切换, 比如已经勾选状态变为非勾选, 非勾选变为勾选, 按下变为未按下, 未按下变为按下, 实现类似全选, 反选功能	56
<b>如果控件变为不可选了, toggle 也能让其控件反选</b>	57
QAbstractButton 排他性, 只要继承这个类的都有排他性, 比如按钮	57
<b>多个按钮选择实现</b>	57
用排他性功能让多个按钮同时只能选择一个	58
对象.setAutoExclusive(True) #启动控件排他性	58
按钮被按住按下, 按钮被释放, 按钮被点击, 按钮选择状态改变实现	59
对象.pressed.connect(槽函数) #按钮被按下执行的槽函数	59
对象.released.connect(槽函数) #按钮被释放执行的槽函数	59
对象.clicked.connect (槽函数) #按钮被点击执行的槽函数, 其实按下释放就会触发该函数	59
对象.toggled.connect (槽函数) #对象状态切换执行槽函数, 一般在单选和复选框用	59
<b>设置按钮某部分点击才有效, 其余部分点击无效</b>	59
菜单控件 QMenu	61
对象 = QMenu() #创建菜单, 只是创建菜单对象, 不能显示, 需要嵌入其它对象显示	61
QPushButton::setMenu(菜单对象) #将菜单对象嵌入进按钮	61
返回子菜单项对象 = QAction(“填写子菜单名”, 填入嵌入的主菜单对象) #创建子菜单项	61
QAction::setIcon(填入 QIcon 函数) #在子菜单项加入图标	61
QIcon(“填入 png 图片路径”) #将图片转换成 setIcon 可以传入的参数	61
QMenu::addAction(子菜单项对象) #向主菜单加入子菜单对象	61
<b>如何监听某一个菜单项被点击?</b>	62
QAction::triggered.connect(槽函数) #子菜单项 QAction 被点击后触发槽函数	62
QAction(QIcon(png 图片路径), “菜单名”, 要嵌入的主菜单对象) #第 2 种菜单图标简化方法	62
QAction::addSeparator() #在多个 QAction 子菜单项之间加入分割线	62
<b>在主菜单下除了创建下拉子菜单, 还可以创建子子菜单</b>	63
扁平化设置(按钮为例)	64
QPushButton::setFlat(True) #将按钮进行扁平化	64
右键菜单实现(就是鼠标在窗口右键显示菜单)	64
contextMenuEvent(self, QContextMenuEvent) #鼠标右键空白窗口, 触发菜单事件, 这个函数是固定模式, 不要随便写函数名	64
对象.globalPos() #获取鼠标在窗口的当前坐标	65
列表按钮(自带扁平功能)	66
QCommandLinkButton(“传入大标题字符”, “传入小标题字符”, 嵌入的对象) #创建列表按钮	66
QCommandLinkButton::setText(“传入字符”) #修改大标题	66
QCommandLinkButton::setDescription(“传入字符”) #修改小标题	66
QCommandLinkButton::setIcon(“传入图标”) #列表按钮传入图标	66
工具按钮, 只显示图标的按钮	66
对象 = QToolButton(要嵌入的对象)	66
QToolButton::setText(“字符”) #给工具按钮增加名字	67
QToolButton::setIcon(QIcon(“加入图标路径”)) #给工具按钮加图标	67
QToolButton::setIcon(QSize(x, y)) #设置工具按钮大	67
QToolButton::setToolTip(“输入要提示的字符”) #鼠标移动到按钮功能提示	67
<b>设置按钮样式和风格</b>	68
QToolButton::setToolButtonStyle(参数)	68
<b>按钮方向图标</b>	68
QToolButton::setArrowType(参数)	68
命令连接按钮, 扁平化的箭头按钮	69
对象 = QCommandLinkButton(输入标题字符串, 描述内容字符串, 父对象) #创建命令链接按钮	69
QCommandLinkButton::setText(标题字符串) #修改初始化的标题	69
QCommandLinkButton::setDescription(描述字符串) #修改初始化的描述	69
单选按钮使用	70

对象 = QRadioButton( “按钮名称”, 嵌入的父对象) #单选按钮创建.....	70
QRadioButton::setIcon(QIcon(路径)) #给单选按钮加图标.....	70
<b>程序启动后单选按钮默认选择某一项.....</b>	<b>70</b>
QRadioButton::setChecked(True) #设置某个对象启动后默认选择.....	70
<b>单选按钮被选中后触发信号, 知道哪一个单选被选了.....</b>	<b>70</b>
QRadioButton::isChecked() #检查对象是否被选上, 被选上返回 True, 未被选上返回 False.....	71
对象 = QButtonGroup(传入可选按钮处于的界面对象).....	72
QButtonGroup:: addButton(传入可选按钮对象) #向按钮组增加可选按钮.....	72
<b>设置可选按钮的 ID 值, 在可选按钮很多的时候可以用 for 循环来创建按钮, 同时也可以用 for 循环检查某个按钮按下.....</b>	<b>73</b>
QButtonGroup::setId(传入可选按钮对象, 设置 ID 参数) #设置可选按钮的 ID 值.....	73
返回 ID 号 = QButtonGroup::id(传入可选按钮对象) #获取可选按钮 ID 值.....	73
<b>各组可选按钮被按下对应的事件组被触发.....</b>	<b>73</b>
QButtonGroup::buttonClicked.connect(传入槽函数) #对应的可选按钮组事件触发函数.....	73
QButtonGroup::buttonClicked[int].connect(传入槽函数).....	74
<b>多选按钮实现, 就是多个打钩.....</b>	<b>75</b>
对象 = QCheckBox( “传入显示字符”, 嵌入的父对象) #创建可选按钮.....	75
QCheckBox::setIcon(QIcon(图片路径)) #设置可选按钮图标.....	76
QCheckBox::stateChanged.connect(槽函数) #勾选按钮选中后触发事件槽, 取消勾选也会触发.....	76
<b>如何知道哪一个勾选按钮被选中呢? .....</b>	<b>76</b>
返回值=QCheckBox::isChecked()##检查指定对象是否被勾选, 被勾选返回 True, 未被勾选返回 False.....	76
<b>输入文本框.....</b>	<b>77</b>
对象 = QLineEdit(嵌入的对象) #创建文本框.....	77
对象 = QLineEdit(“传入字符串”, 嵌入的对象) #创建文本框时加入字符串, 那么文本框默认显示字符串.....	77
QLineEdit::setText(“传入字符串”) #设置文本内容, (可以将获取的数据转成字符串显示到文本框 ).....	77
<b>获取文本框内容</b>	
变量 = QLineEdit::text() #获取文本框内容返回给变量.....	77
将 A 文本内容复制给 B 文本, 使用 setText 或者 insert 插入(注意 insert 有区别).....	78
QLineEdit::setText(传入文本内容) #指定文本框对象, 写内容, 也会覆盖以前的文本内容.....	78
QLineEdit::insert(传入文本内容) #插入文本, 不会覆盖文本以前的内容, 只会在以前的文本后追加.....	78
QLineEdit::setEchoMode(参数) #设置文本框输出的内容类型.....	78
QLineEdit.Password 密码不显示.....	79
QLineEdit::setPlaceholderText( “字符串 ”) #传入字符串, 文本框背景显示提示字符.....	79
QLineEdit::setAlignment(参数) #设置文本框里面的字符居中, 左对齐, 右对齐.....	80
<b>QFrame 控制矩形空间, 凸起, 凹下, 阴影, 线宽, 一些矩形控件外观显示样式.....</b>	<b>80</b>
生成矩形框对象 = QFrame(父对象).....	80
QFrame :: setFrameShape(类型) #设置窗体类型.....	81
QFrame :: setLineWidth(宽度) #边框外线宽度.....	82
QFrame :: setMidLineWidth(宽度) #边框中线宽度.....	82
QFrame.Panel #设置矩形框为平面.....	82
QFrame :: setFrameRect(参数) #设置 frame 突出部分的矩形框架尺寸.....	82
<b>QAbstractScrollArea 滚动空间类, 继承 QAbstractScrollArea 类的子类都具备滚动功能, 如滚动条.....</b>	<b>82</b>
<b>QTextEdit 多行文本使用.....</b>	<b>82</b>
多行文本对象 = QTextEdit(父对象) #该类继承至 QAbstractScrollArea.....	82
多行文本对象 = QTextEdit( “默认字符串”,父对象).....	82
QTextEdit::append( “填入追加字符串 ”) #多行文本, 追加下一行文本.....	83
QTextEdit::setText( “ ” ) #空冒号可以清空多行文本内容.....	83
QTextEdit::clear() #clear 也可以清空多行文本内容.....	83
QTextEdit::setPlainText(填入内容) #覆盖以前的内容, 填入新内容.....	83

QTextEdit::setHtml(传入 html 格式内容) #传入文本的字符，按照 html 语法设置的特性显示.....	84
<b>步长编辑器 QSpinBox.....</b>	<b>85</b>
对象 = QSpinBox(父对象) #创建步长编辑器.....	85
QSpinBox::setMaximum(参数) #步长编辑器最大长度，参数就是设置的长度值.....	85
QSpinBox::setMinimum(参数) #步长编辑器最小长度.....	85
QSpinBox::setRange(最小值,最大值) #一次性设置步长最小值最大值.....	85
QSpinBox::setSingleStep(参数) #设置步长最小增减值.....	85
QSpinBox::setSuffix("周") #设置数字单位.....	85
对象 = QDoubleSpinBox(父对象) #带浮点显示步长编辑器.....	86
QDoubleSpinBox:: setMaximum(参数) #设置浮点数最大长度，最小长度同理用 setMinimum.....	86
QDoubleSpinBox::setSingleStep(参数) #设置小数递增也是用步长最小增减值.....	86
<b>槽函数中直接用 lambda 实现.....</b>	<b>86</b>
QDoubleSpinBox::value() #获取当前步长编辑器的值.....	86
<b>取消按钮，步长编辑器数据改变一次，槽函数执行一次.....</b>	<b>86</b>
QDoubleSpinBox::valueChanged.connect(槽函数).....	86
<b>QTime 系统时间获取.....</b>	<b>87</b>
对象 = QTime.currentTime() #获取系统当前运行时间，从 0 开始计算.....	87
对象.start() #启动系统时间获取.....	87
对象.elapsed() #获取当前系统运行时间 ms 毫秒.....	87
日期时间变量 = QDateTime.currentDateTime() #获取当前日期时间.....	87
对象.date() # 获取日期，但是重复执行(对象.date)日期是不会更新的，必须执行 QDateTime.currentDateTime() 之后，(对象.date) 才会更新.....	87
对象.time() #只获取当前时间，时间更新和.date 同理.....	87
时间更新需要执行 QDateTime.currentDateTime().....	87
<b>QComboBox 下拉选择框.....</b>	<b>88</b>
对象 = QComboBox(父对象) #创建下拉框.....	88
QComboBox::addItem(传入下拉框显示得字符) #创建下拉行，执行一次创建一个下拉行.....	88
QComboBox::addItems(传入元组或者列表) #一次性创建多个下拉框.....	88
对象 = QStandardItemModel() #创建一个空模型.....	89
返回值 = QComboBox::count() #获取下拉框数量.....	89
返回值 = QComboBox::currentIndex() #获取当前选中的哪一行下拉框.....	90
返回值 = QComboBox::currentText() #获取当前选中下拉框的内容.....	90
QComboBox::currentData() #获取当前选中下拉框的数据.....	90
QComboBox::currentIndexChanged.connect(槽函数(值)).....	91
QComboBox::currentIndexChanged[str].connect(槽函数) #[str]指定传递给槽函数形参的内容为 addItem 定义的字符，而不是下拉框编号.....	91
<b>两个下拉框联动显示案例.....</b>	<b>91</b>
<b>滑块使用 QSlider.....</b>	<b>93</b>
对象 = QSlider(父对象) #创建滑块对象.....	93
QSlider::valueChanged.connect(槽函数) #滑块滑动，不停的执行槽函数.....	93
QSlider::setMaximum(参数) #设置滑块最大值.....	93
QSlider::setMinimum(参数) #设置滑块最小值.....	93
QSlider::setSingleStep(步长值) #设置滑块步长，滑块按照 5 个步长移动，但是鼠标无效，必须键 盘上下键操作.....	94
QSlider::setValue(50) #自动设置滑块数值.....	94
QSlider::setOrientation(Qt.Horizontal) #设置滑块摆放方向.....	94
QSlider::setTickInterval(刻度间距值) #设置刻度线间距.....	95
<b>在滑块按钮上显示滑动数值.....</b>	<b>95</b>
<b>带上下箭头的滑块 QScrollBar.....</b>	<b>97</b>
对象 = QScrollBar(父对象) #创建上下箭头滑块.....	97
Qt.Horizontal #水平滚动条.....	98
对象 = QDial(父对象) #创建圆形滚动盘.....	98
QDial::valueChanged.connect(槽函数) #滚动盘滑动触发槽函数.....	98
QDial::SetNotchesVisible(True) #给滚动盘加刻度.....	99

QDial::SetNotchTarget(刻度间隔) #设置刻度之间间隔距离.....	99
<b>区域选择的控件(橡皮筋选择控件) QRubberBand.....</b>	<b>99</b>
对象 = QRubberBand(选择框形状, 父对象) #选择框创建.....	99
QRubberBand::setGeometry(起始位置 x, 起始位置 y, 显示宽度, 显示高度) #对几何对象的位置, 长宽进行修改实时修改.....	100
返回 x,y,长,宽值 = QRect(矩形起始 x 坐标,矩形起始 y 坐标,矩形长,矩形宽) #创建一个指定长宽的矩形类.....	100
返回 x,y,长,宽值 = QRect(矩形起始 x 坐标,矩形起始 y 坐标,矩形长,矩形宽).normalized() xy 支持负值反转.....	101
<b>弹出新对话框实现 QDialog.....</b>	<b>102</b>
对象 = QDialog() #创建模态对话框.....	102
QDialog::exec() 阻塞程序, 等待鼠标点击取消模态对话框, 程序才向下执行.....	102
QDialog::open() 非阻塞, 模态对话框和主窗口同时运行.....	102
QDialog::show() #就算模态对话框嵌入进父窗口, 两个窗口都能分开操作.....	103
QDialog::accept() #返回接收, 传递数字 1 给 exec().....	104
QDialog::reject() #返回拒绝, 传递数字 0 给 exec().....	104
QDialog::done(参数) #返回参数, 传递参数给 exec().....	104
对象 = QFontDialog(父窗口) #字体选择模态对话框.....	104
QFontDialog::show() #显示字体选择模态对话框.....	104
QFontDialog::exec() #阻塞字体模态对话框.....	105
对象 = QFontDialog::selectedFont() #获取选择完的字体.....	105
对象.family() #获取选择的是哪一类字体.....	105
<b>模态对话框获取要打开的文件路径 QFileDialog.....</b>	<b>105</b>
返回文件路径字符串 = QFileDialog.getOpenFileName(父对象,对话框标题,指定打开目录,(过滤字符串也),选择文件类型汇总, 默认选择文件) #获取选择的文件路径.....	105
QFileDialog.getOpenFileName 适合在按钮的槽函数中执行.....	106
<b>输入模态对话框 QInputDialog.....</b>	<b>107</b>
对象 = QInputDialog(父对象,参数) #创建模态对话框.....	107
QInputDialog::setComboBoxItems(列表) #输入模态对话框增加条目, 条目多少根据列表来算.....	107
返回元组 = QInputDialog.getInt(父对象,外标题,内标题,默认值).....	107
返回元组 = QInputDialog.getItem(父对象,外标题,内标题,列表).....	108
<b> QLabel 做展示数据使用.....</b>	<b>108</b>
对象 = QLabel(默认显示字符串,父控件) #创建标签控件.....	108
QLabel::setAlignment(对齐方式).....	108
Qt.AlignVCenter #垂直方向居中.....	108
QLabel::setIndent(参数) #水平对齐方向保持相对间距设置, 如果没有提起设置右对齐, 默认就是左对齐保持相对间距.....	109
QLabel::setPixmap(父对象(可选),QPixmap 参数) #标签显示图片.....	109
QPixmap(传入图片路径字符串) #主要用于绘制图片.....	109
QLabel::adjustSize() #根据图片大小, 自动调整标签控件长宽.....	110
QLabel::setScaledContents(True) #图片服从标签尺寸.....	110
QLabel::setNum(整形数/变量) #显示整形数据/浮点型数据.....	110
得到动画对象 = QMovie(gif 动画路径字符串) #获取 gif 动画.....	111
QMovie::start() #启动动画, 不执行这句, gif 动画不会启动.....	111
QLabel::setMovie(传入 QMovie 动画对象) #在标签上显示 gif 动画.....	111
<b>数码管显示 QLCDNumber.....</b>	<b>112</b>
对象 = QLCDNumber(显示多少位数字(可选), 父对象) #创建数码管显示.....	112
QLCDNumber::display(数值) #显示整数.....	112
QLCDNumber::display(数值) #显示小数, 前提是显示数量位数够长.....	113
<b>进度条控件 QProgressBar.....</b>	<b>113</b>
对象 = QProgressBar(父对象) #进度条创建.....	113
QProgressBar::setMinimum(最小值) #设置进度条最小值.....	113
QProgressBar::setMaximum(最大值) #设置进度条最大值.....	113
QProgressBar::setValue(50) #当前进度条显示的位置.....	113

QProgressBar::reset() #重置进度条，也就是进度条清 0 .....	113
QProgressBar::setFormat(字符串传入) #给进度条设置中文提示,%v 显示进度条值.....	114
QProgressBar::SetOrientation(参数) #设置进度条显示方向，默认水平.....	114
<b>进度条实时显示.....</b>	<b>115</b>
对象 = QTimer(父对象) #创建定时器.....	115
QTimer::timeout.connect(传入槽函数) #定时时间到执行槽函数.....	115
QTimer::start(定时时间) #启动定时器，设置定时时间，定时时间到执行 timeout.connect 指定的槽函数.....	115
返回值 = QProgressBar::value() #获取进度条当前值.....	115
<b>对话框控件 QErrorMessage，弹出一些警告，错误提示.....</b>	<b>116</b>
对象 = QErrorMessage(父对象) #创建对话框，因为该对话框继承至 QDialog,所以支持 exec, open, show 三种模态状态.....	116
QErrorMessage::setWindowTitle(传入显示的字符串) #对话框标题显示.....	116
QerrorMessage::ShowMessage(传入显示的字符串) #对话框里面显示内容.....	116
<b>模态对话框进度条控件.....</b>	<b>117</b>
对象 = QProgressDialog(父对象) #创建进度条对话框，进度条对话框要 4 秒之后才显示，奇怪.....	117
QProgressDialog::setValue(进度值) #设置进度条对话框值.....	117
QProgressDialog::setMinimumDuration(参数) #设置等待时间，参数写 0 表示不等的.....	117
QProgressDialog(中间标题,取消按钮标题,最小值,最大值,父对象) #创建进度条对话框,设置内容.....	117
<b>消息弹出 QMessageBox.....</b>	<b>118</b>
对象 = QMessageBox(显示图标,显示标题,显示内容,显示按钮,父对象) #消息对话框.....	118
<b>布局管理器.....</b>	<b>118</b>
<b>垂直布局管理器.....</b>	<b>118</b>
对象 = QVBoxLayout() #创建垂直布局管理器.....	118
窗口对象.setLayout(传入布局管理器对象) #将布局管理器嵌入进窗口.....	118
<b>水平布局管理器.....</b>	<b>119</b>
对象 = QHBoxLayout() #创建水平布局管理器.....	119
布局嵌套布局.....	119
对象 = QVBoxLayout(布局方式) #盒子布局函数，他有两个子类 QHBoxLayout 和 QVBoxLayout, 它唯一的区别就是需要使用布局方式指定是垂直布局还是水平布局.....	119
QVBoxLayout::setLayout() #在垂直布局管理器窗口里面加入其它布局方式.....	120
<b>布局管理器控件两边间距隔离.....</b>	<b>120</b>
<b>布局管理器指定某个控件比其它控件尺寸大，拉伸的时候大控件也按比例变大.....</b>	<b>121</b>
VBoxLayout::addWidget(控件对象,拉伸窗口比例) #拉伸窗口比例是按百分比来算的，如有 3 行控件,addWidget(控件对象,1) #表示该控件尺寸占窗口 1 倍.....	121
addWidget(控件对象,2) #表示该控件尺寸为两倍控件大小.....	121
addWidget(控件对象) #没有参数，表示该控件尺寸最小.....	121
QVBoxLayout::SetStretchFactor(指定控件,拉伸系数) #指定某个控件拉伸改变尺寸，其余控件在拉伸的时候尺寸不变。.....	122
<b>表单布局管理器 QFormLayout.....</b>	<b>122</b>
对象 = QFormLayout() #创建表单布局管理器.....	122
QFormLayout::addRow(控件对象 1,控件对象 2) #第 1 种排列控件方式，控件并列布局.....	122
表单布局内部加入水平布局.....	123
QFormLayout::insertRow(插入某行,控件/布局对象) #在表单布局管理器某行插入新控件/布局对象.....	123
QFormLayout::rowCount() #获取表单布局管理器行数.....	123
QFormLayout::removeRow(传入布局对象/控件对象/行号) #删除布局管理器的某行布局.....	124
QLabel::destroyed.connect() #实时监听标签控件是否被删除.....	125
QFormLayout::setVerticalSpacing(间距值) #表单布局里面的控件垂直间距设置.....	125
QFormLayout::setHorizontalSpacing(间距值) #表单布局内部控件之间水平方向间距.....	125
<b>网格布局管理器 QGridLayout.....</b>	<b>126</b>
对象 = QGridLayout() #创建网格布局管理器.....	126

QGridLayout::addWidget(控件对象) #加载布局控件默认是垂直布局.....	126
QGridLayout::addWidget(父对象(可以不填),控件对象, 布局某行, 布局某列).....	126
QGridLayout::addWidget(父对象(可以不填),控件对象, 布局某行, 布局某列,多占几行, 多占几列).....	127
QGridLayout::setColumnStretch(某列,拉伸系数) #网格布局管理器控件列弹簧.....	127
QGridLayout::setRowStretch(某行,拉伸系数) #设置网格布局某行控件高度伸缩系数.....	128
对象 = QStackedLayout(父对象<可以不填后面用 setLayout 载入>) #创建栈式布局管理器.....	128
QStackedLayout::setCurrentIndex(传入层号) #选择指定层控件.....	128
QStackedLayout::currentChanged.connect(槽函数) #栈式布局管理器, 界面切换, 槽函数启动。....	129
界面不管是手动还是自动切换, 都会执行槽函数.....	129
QStackedLayout::currentChanged.connect(lambda val:print(val)) #界面切换的时候, 切换到第几层了, 会将当前切换层传递给槽函数 val.....	129
QStackedLayout::removeWidget(传入控件) #栈式布局得到的控件, 指定某层/某控件移除.....	129
控件.setFixedSize(长度,宽度) #布局管理器不管怎么伸缩, 设置最小的显示尺寸, 该尺寸无法因为伸缩再缩小.....	129
<b>QSS 使用.....</b>	130
<b>如何加载调用 QSS 文件.....</b>	130
插入内容: 如果 QSS 加载出现 gbk 错误	
<b>UnicodeDecodeError: 'gbk' codec can't decode byte 0xaf in position 49: ....</b>	130
<b>ID 选择器使用.....</b>	131
控件大名 #创建的对象 ID{.....}.....	131
控件.setObjectName(设置 ID 字符串) #设置控件 ID, 方便 QSS 单独设置样式.....	131
<b>伪状态使用.....</b>	132
控件#选择器:hover{...} #鼠标移动到控件执行.....	132
<b>通配符选择器.....</b>	132
*{...} #让所有控件的样式都执行*号里面的内容.....	132
<b>类选择器.....</b>	132
点 . 控件 {...} #类选择器, 比如控件创建的子类, 子类控件就不会被样式修改.....	132
image(传入图片路径) #加载图片.....	132
设置图片长宽.....	133
<b>子控件选择器.....</b>	133
复选框, 切换选择状态, 图片显示 QCheckBox.....	133
控件::indicator{...} #指示器设置, 一般设置些默认参数, 只支持 QCheckBox,QRadioButton...133	133
<b>子选择器.....</b>	134
控件#选择器>子控件{.....}.....	135
#控件,#控件,#控件{...} #多控件选择, 对多控件进行样式修改, 修改多少个控件, 就用多少个‘,’逗号隔离开.....	136
<b>用伪状态给控件加边框.....</b>	136
border #创建矩形边框.....	136
solid 实线, dotted 虚线.....	136
控件:pressed{...} #鼠标按下伪状态.....	136
<b>盒子模型.....</b>	136
border-width: 填入边框线宽.....	136
border-style: 上边框样式 右边框样式 下边框样式 左边框样式 #填入边框线的类型.....	136
设置上右下左, 每条边框线.....	136
border-top-style: 设置上边框样式 #border-style 一般用于同一设置, 而 border-top-style 只设置上边框.....	136
border-top-style: none; #none 为无边框设置.....	137
border-width: 上边宽度 右边宽度 下边宽度 左边宽度 #设置边框各边宽度.....	137
border-bottom-width: 下边宽度 #单独设置边框下边宽度.....	137
border-color: 颜色 1 颜色 2 #设置边框对边颜色.....	138
border-color: 上边颜色 右边颜色 下边颜色 左边颜色 #设置边框 4 边颜色.....	138
border-left-color: 颜色 #单独设置边框左边颜色.....	138
<b>颜色渐变.....</b>	138

lineargradient(扫描的 x 起始位置, 扫描的 y 起始位置,扫描结束的 x 位置,扫描结束的 y 位置,stop: 扫描起始位置 颜色, stop:扫描中间位置 颜色, stop:扫描结束位置 颜色) #颜色渐变设置.....	138
<b>辐射渐变.....</b>	<b>140</b>
qradialgradient(圆心 x 位置,圆心 y 位置,radius:圆心向外辐射的半径,光源辐射方向 x 位置,光源辐 射方向 y 位置,起始颜色,结束颜色); #圆形辐射渐变.....	140
<b>角度渐变.....</b>	<b>141</b>
qconicalgradient(cx:0.5,cy:0.5,angle:10,stop:0 red,stop:1 orange) #角度渐变.....	141
border-image:url(传入图片路径) #加载图片.....	141
background-image:url(传入图片路径) #设置图片到背景显示.....	142
margin:矩形缩放多少个像素 #外边距就是让控件向内缩放.....	142
padding: 前景控件内缩距离 #内边距设置.....	142
padding-top: 前景控件内缩相差顶部距离 #内边距顶部距离设置.....	143
<b>设置字体.....</b>	<b>143</b>
font-size: 字体大小 px #设置字体大小.....	143
font-style: italic; #字体为斜体.....	143
font-weight: 100~900; #设置字体粗细.....	143
<b>针对布局管理器的控件, 在 QSS 设置控件尺寸最大和最小。 .....</b>	<b>144</b>
<b>QSpinBox 控件 QSS 设置.....</b>	<b>144</b>
QSpinBox::up-button #设置向上按钮尺寸大小.....	145
subcontrol-position #QSS 设置控件位置.....	145
<b>向 QSpinBox 上下按键加入图标.....</b>	<b>146</b>
QSpinBox::up-button:hover{...} #加入向上按钮伪状态, 鼠标移动到向上按钮控件执行 .....	146
QSpinBox::down-button:hover{...} #加入向下按钮伪状态, 鼠标移动到向下按钮控件执行...147	147
<b>QCheckBox 勾选按钮 QSS 设置.....</b>	<b>147</b>
<b>PyQT5 自定义信号.....</b>	<b>147</b>
对象 = pyqtSignal(数据类型) #信号发送数据类型的值给槽函数, 数据类型就接收数据的定义(可 以是字符串, 整形, 浮点型).....	148
pyqtSignal().emit(数据/变量) #将变量和数据传递给信号, 信号将发出去.....	148
对象 = pyqtSignal(数据类型) #信号发送数据类型的值给槽函数, 数据类型就接收数据的定义(可 以是字符串, 整形, 浮点型).....	148
pyqtSignal().emit(数据/变量) #将变量和数据传递给信号, 信号将发出去.....	148
<b>信号发送多个参数实现.....</b>	<b>149</b>
对象 = pyqtSignal([数据类型],[数据类型]) #信号用[]括号设置多个参数类型, 但是只能发射其中 一个类型.....	149
<b>控件动画实现.....</b>	<b>150</b>
对象 = QPropertyAnimation(父对象) #属性动画类, 创建动画.....	150
QPropertyAnimation::setTargetObject(传入控件) #将要用于做动画的控件对象传入.....	150
QPropertyAnimation::setPropertyNames(参数) #做动画要用什么属性。 .....	150
QPropertyAnimation::setStartValue(坐标参数) #动画在窗口开始位置.....	150
QPropertyAnimation::setEndValue(坐标参数) #动画在窗口结束位置.....	150
QPropertyAnimation::setDuration(ms 毫秒) #设置动画从开始到结束运行时间.....	150
QPropertyAnimation::start() #启动动画.....	150
PropertyAnimation::setPropertyNames(b"size") #使用 size 就是控件尺寸大小动画变化.....	151
QPropertyAnimation::setPropertyNames(b"geometry") #使用 geometry 设置控件坐标变化, 也可以同 时尺寸变化.....	152
QRect(控件 x 坐标, 控件 y 坐标,控件宽度尺寸, 控件高度尺寸) #QRect 即设置控件位置, 也设 置控件尺寸大小.....	151
QPropertyAnimation(父对象(一定是窗口类 QWidget),b"windowOpacity") #设置窗口为显示~透明 模式 0 为完全显示, 1 为完全透明.....	152
<b>设置窗口透明与不透明来回闪烁.....</b>	<b>152</b>
QPropertyAnimation::setKeyValueAt(填入动画时间区域,透明值) #在动画运行过程中间插入控 制.....	152
<b>动画移动节奏控制.....</b>	<b>152</b>
QPropertyAnimation::setEasingCurve(QEasingCurve.OutQuad) #设置动画由快到慢.....	153

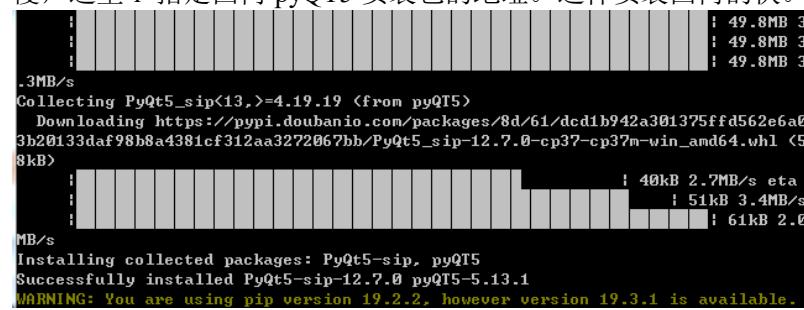
QPropertyAnimation::setEasingCurve(QEasingCurve.InBounce) #设置动画弹簧效果，开始移动是弹簧，移动结尾是匀速.....	153
<b>PyQT5 多线程使用.....</b>	<b>154</b>
<b>多线程创建.....</b>	<b>154</b>
对象 = 类(QThread) #类继承至 QThread,实现 run 函数，就形成多线程了 .....	154
线程自定义信号，用于界面实时更新.....	155
QMutex 互斥锁(线程锁)与全局变量在多线程读写的安全.....	158
注意:在 pyQT5 多线程使用全局变量 global 只能在 run 函数里面声明.....	158
两个线程相互通信或者多个线程之间相互通信传递数据，使用信号连接信号.....	158
发送端对象 = pyqtSignal(int) #比如创建个信号，发送 int 数据.....	158
接收端对象 = pyqtSignal(int) #接收线程创建个信号，接收 int 数据.....	158
发送端对象.connect(接收线程.接收端对象[int]) #发送端信号用 connect 连接接收端信号....	158
<b>综合应用案例，程序打包 exe.....</b>	<b>160</b>
<b>    应用案例实现.....</b>	<b>160</b>
border:none; #取消控件四周边框.....	161
对象 = QSequentialAnimationGroup(传入要显示序列动画的窗口对象).....	162
QSequentialAnimationGroup::addAnimation(添加 QPropertyAnimation 创建的动画对象) #按照添加顺序，显示创建的属性动画.....	162
QPropertyAnimation::setLoopCount(传入重复次数) #属性动画重复播放次数，如果填入-1 表示无限重复播放.....	162
QSequentialAnimationGroup::addPause(传入停止时间 ms 毫秒) #序列动画暂停.....	162
QSequentialAnimationGroup::start() #启动动画.....	162
<b>    界面滑动切换.....</b>	<b>165</b>
<b>    将程序打包成 exe 文件.....</b>	<b>168</b>

## PyQT5 环境搭建, pycharIDE 版本

确保学过 C++, 和 C++QT4.7 知识, 确保计算机安装了 python3.x 解释器, IDE 使用 pychar 开发, 确保 pip 软件安装成功, 在 windows 平台上使用 pyQT5.

### PyQT5 安装包安装

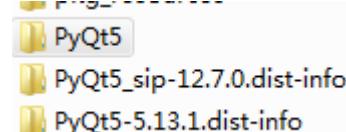
```
D:\pycharm>pip install pyQT5 -i https://pypi.douban.com/simple
pip install pyQT5 -i https://pypi.douban.com/simple //因为安装默认官方路径的 pyQT5 太慢, 这里-i 指定国内 pyQT5 安装包的地址。这样安装国内的快。
```



```
.3MB/s
Collecting PyQt5-sip<13,>-4.19.19 <from pyQT5>
  Downloading https://pypi.douban.com/packages/8d/61/dcd1b942a301375ffd562e6a03b20133daf98b8a4381cf312aa3272067bb/PyQt5_sip-12.7.0-cp37-cp37m-win_amd64.whl (58kB)
    : 49.8MB 3
    : 49.8MB 3
    : 49.8MB 3
  MB/s
Installing collected packages: PyQt5-sip, pyQT5
Successfully installed PyQt5-sip-12.7.0 pyQT5-5.13.1
WARNING: You are using pip version 19.2.2, however version 19.3.1 is available.
```

安装时候会显示一大堆进度条, 安装成功就是图上这个样子

PyQT5 包安装之后会在 D:\pycharm\Lib\site-packages 这个目录下



再安装 pyQT5-Tools 工具

```
D:\pycharm>pip install PyQt5-tools
```

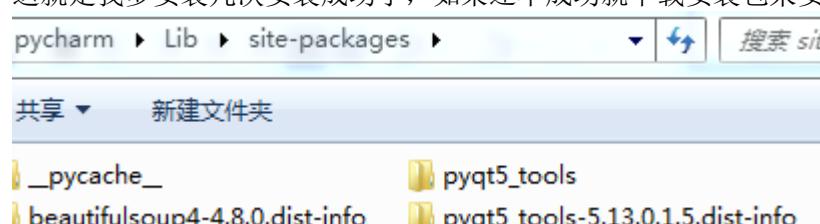
pip install PyQt5-tools

有可能出现安装出错, 如果出错显示很多红色字符, 你就多安装几次。

```
Successfully installed PyQt5-tools-5.13.0.1.5 click-7.0 pyqt5-5.13.0 python-dotenv-0.10.3
```

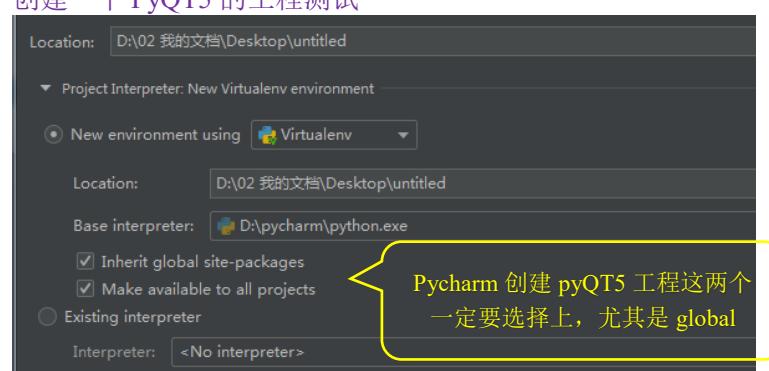
```
WARNING: You are using pip version 19.2.2, however version 19.3.1 is available.
```

这就是我多安装几次安装成功了, 如果还不成功就下载安装包来安装。

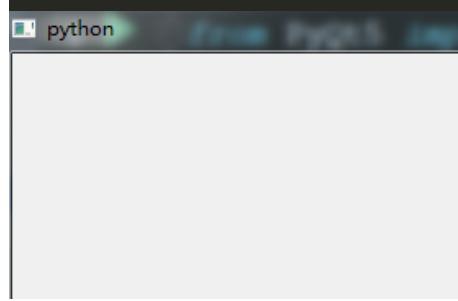


在 site-packages 目录下出现了 pyqt5\_tools 目录。

创建一个 PyQT5 的工程测试



```
from PyQt5 import QtWidgets, QtGui  
import sys  
  
app = QtWidgets.QApplication(sys.argv)  
window = QtWidgets.QWidget()  
window.show()  
sys.exit(app.exec_())
```



工程创建运行成功

换一个方式使用 pyQT5 的库，这种方式更简单

```
from PyQt5.Qt import * #这样包含方便些，Pyqt5库全部包含  
import sys  
  
app = QApplication(sys.argv)  
  
window = QWidget()  
window.setWindowTitle("真全栈窗口")  
window.resize(500, 500) #设置主窗口大小  
window.move(100, 100) #设置窗口移动位置  
  
label = QLabel(window) #将标签嵌入进父窗口  
label.setText("标签窗口")  
label.move(200, 200)  
window.show() #显示窗口  
sys.exit(app.exec_())
```

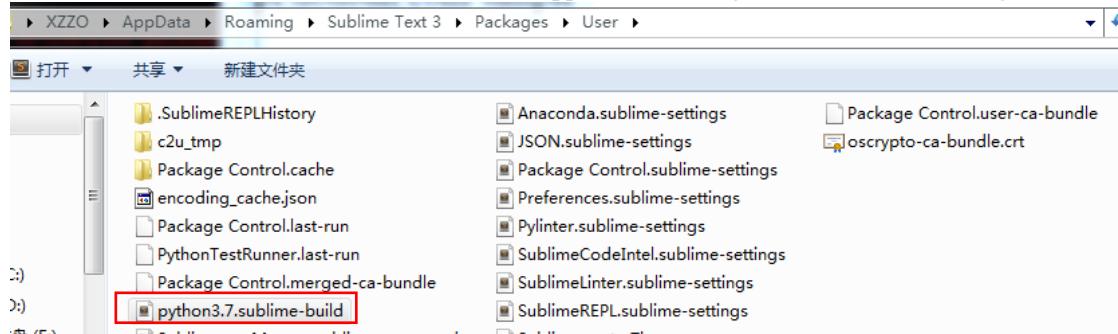


再次证明 pycharm 的 pyQT5 环境搭建好了。

## Sublime text3 运行 PyQt5

pip install pyQT5 -i https://pypi.douban.com/simple  
确保 pyQT5 图形库安装

进入你的 sublime 安装位置 C:\Users\XZZO\AppData\Roaming\Sublime Text 3\Packages\User



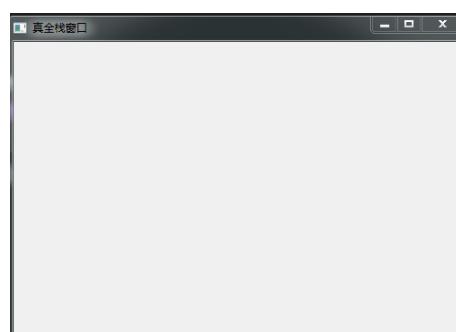
然后在 sublime text3 配置目录下修改 python3.7.sublime-build

```
{  
    "cmd": ["C:\\\\Users\\\\XZZO\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\python.",  
    "file_regex": "^[ ]*File \"(.*)\"", "line ([0-9]*)",  
    "selector": "source.python",  
    "shell":true, }  
增加这句话，记住逗号不要取消
```

然后再 sublime\_Guide 手册里面确认 sublime 的 python 运行环境和寻找 python 库的路径已经设置完成

```
1 from PyQt5.Qt import *  
2 import sys  
3  
4 app = QApplication(sys.argv)  
5  
6 window = QWidget()  
7 window.setWindowTitle("真全栈窗口")  
8 window.resize(500, 500)  
9 window.show()  
10 sys.exit(app.exec_())  
11
```

测试代码



运行成功

# VScode 搭建 pyQT5



除了安装 VscodePyQT5 插件，还有按照前面的内容安装 PyQt5 库和环境  
再按照《 VscodeConfig20200714.pdf 》文档配置 Vscode 下的 PyQt5 环境  
`python.exe -m pip install --upgrade pip`  
让 python2.7 支持 pip 21.0 版本

VScode 运行 PyOT5 出现无法导入问题

```
Traceback (most recent call last):
  File "d:\pyQT5test\test.py", line 2, in <module>
    from PyQt5.Qt import *
ImportError: No module named PyQt5.Qt
```

这时候只有先确认自身的 pyOT5 是否安装成功

打开 IDLE，输入 `import PyQt5` 如果什么都没有打印，证明 PyQt5 安装成功，用 `help` 检查下

```
>>> import PyQt5
>>> help(PyQt5)
Help on package PyQt5:

NAME
    PyQt5

DESCRIPTION
    # Copyright (c) 2020 Riverbank Computing Limited <ir
m>
    #
    # This file is part of PyQt5.
    #
    # This file may be used under the terms of the GNU G
    # version 3.0 as published by the Free Software Four
    # the file LICENSE included in the packaging of thi
e
    #
    # following information to ensure the GNU General Pu
    # requirements will be met: http://www.gnu.org/copyl
    #
    # If you do not wish to use this file under the term
    # then you may purchase a commercial license. For m
    # info@riverbankcomputing.com.
    #
    # This file is provided AS IS with NO WARRANTY OF AN
    # WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR

PACKAGE CONTENTS
    QAxContainer
    Qt
    QtBluetooth
    QtCore
```

打印这些内容，证明 PyQt5 安装成功

```
Traceback (most recent call last):
  File "d:\pyQT5test\test.py", line 2, in <module>
    from PyQt5.Qt import *
ImportError: No module named PyQt5.Qt
```



这种情况可能是点击 Run Code 造成的

```
[Running] python -u "d:\pyQT5test\test.py"
  File "d:\pyQT5test\test.py", line 10
SyntaxError: Non-ASCII character '\xe7' in file d:\pyQT5test\test.py on line 10,
```

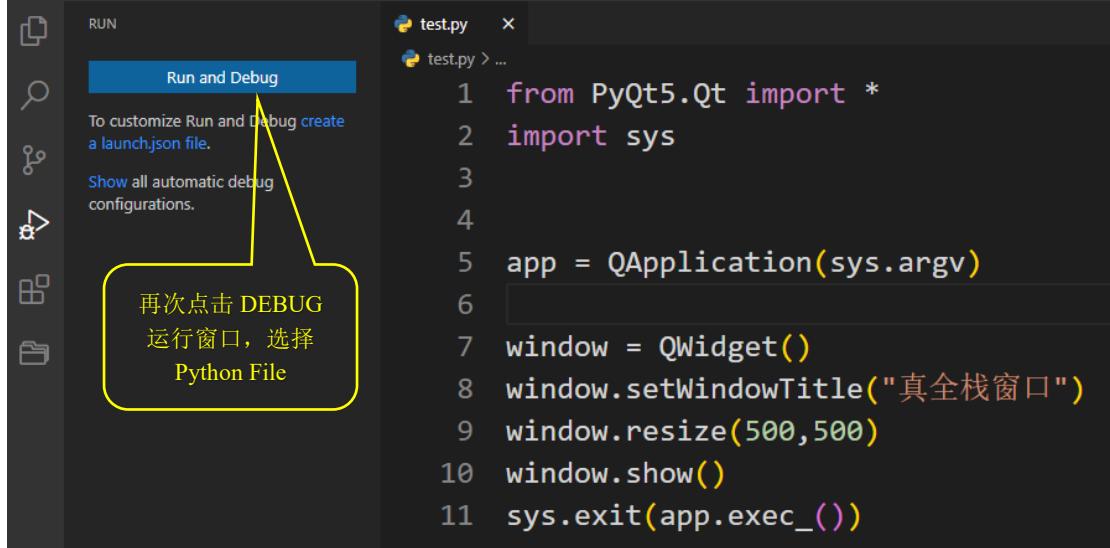


ASCII 编码错误，也是点击 Run Code 造成的

我们用 DEBUG 运行模式来试试

如果我们在 setting.json 设置了 debug 模式运行其它终端，那么要将其关闭掉

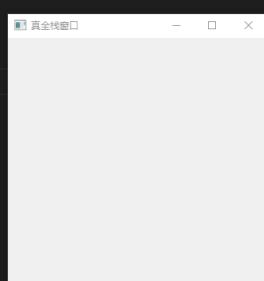
```
{  
  /*"terminal.integrated.shell.windows": "D:\\ESP32system\\msys32\\msys2_shell.cmd",*/  
  "terminal.integrated.shellArgs.windows": ["-defterm", "-mingw32", "-no-start", "-here"],  
  我关闭了 DEBUG 运行模式下首先运行 ESP32 环境的脚本。
```



```
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)

window = QWidget()
window.setWindowTitle("真全栈窗口")
window.resize(500,500)
window.show()
sys.exit(app.exec_())
```



窗口运行成功

这证明了主要问题还是在 Run Code 这个运行功能上，我的 PyQt5 安装库和环境配置没问题

可以在 settings.json 指定 python 运行程序路径，这样保险点。

```
"python.pythonPath": "C:\\\\Users\\\\XZ666\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\python.exe",
```

Run code 运行 PyQt5 设置

```
SyntaxError: Non-ASCII character '\\xe7'
```

运行过程中出现'\\xe7'，是编码问题

在 py 文件顶部加入 `#-- coding:UTF-8 --` 解决问题

```
#-- coding:UTF-8 --
from PyQt5.Qt import *
import sys
```

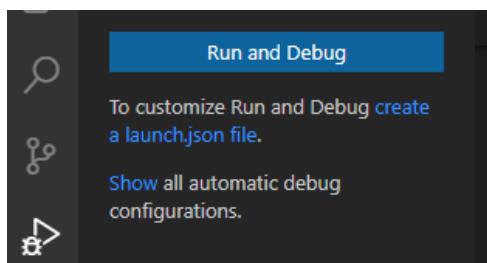
ImportError: No module named PyQt5.Qt 问题

```
Traceback (most recent call last):
File "d:\\pyQT5test\\test.py", line 2, in <module>
    from PyQt5.Qt import *
```

```
ImportError: No module named PyQt5.Qt
```

我们再次回到 Run code 无法导入 pyQT5 库的问题

看来只能用 debug 模式来运行



可以在 debug 模式下 点击 `create a launch.json file` 来创建 launch.json 配置文件，然后选择 python file。这样以后点击运行就不会让你再选择文件了。

## QObject类使用

```
Qobject.setObjectName(参数) //给 Qobjcet 类创建的对象取唯一名称  
Qobject.objectName() //获取对象的唯一名称
```

```
from PyQt5.Qt import * #这样包含方便些，Pyqt5库全部包含  
import sys  
  
obj = QObject()  
obj.setObjectName("给obj对象取唯一名称")  
print(obj.objectName())  
  
给obj对象取唯一名称
```

给对象添加属性，使用到该对象的时候可以获取自定义属性

Qobject.setProperty("属性名", "值") //设置对象属性名函数

Qobject.Property("属性名") //返回对象属性对应的值

```
obj = QObject()  
obj.setObjectName("给obj对象取唯一名称")  
  
obj.setProperty("name1", "12345") #给对象添加1个属性  
obj.setProperty("name2", "109876") #给对象添加2属性  
  
print(obj.objectName())  
print(obj.property("name2")) #指定属性名就可以获取对应值  
print(obj.property("name1"))
```

给obj对象取唯一名称  
109876  
12345

这就是属性名获取的值

这个属性设置主要应用在 QSS 样式表和用于装饰器的信号与槽。

给本对象设置父对象

Qobject.setParent(传入对象) //设置父对象函数使用

```
from PyQt5.Qt import * #这样包含方便些, PyQt5库全部包含
import sys
```

```
obj1 = QObject()
obj2 = QObject()
print("obj1地址", obj1) #获取obj1对象地址
print("obj2地址", obj2) #获取obj2对象地址
```

```
obj1.setParent(obj2) #将obj2设置成obj1的父对象
print(obj1.parent())
```

```
obj1地址 <PyQt5.QtCore.QObject object at 0x0000000002EDA318>
obj2地址 <PyQt5.QtCore.QObject object at 0x0000000002EDA3A8>
<PyQt5.QtCore.QObject object at 0x0000000002EDA3A8>
```

obj2 就是 obj1 的  
父对象

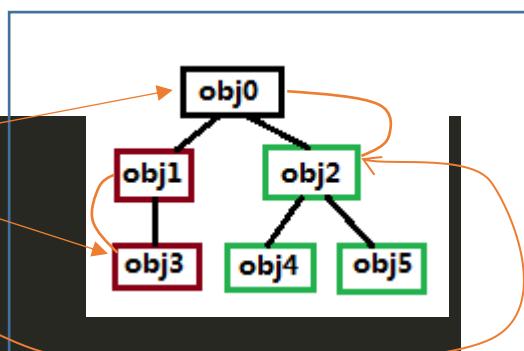
QObject 设置对象的父子关系实际应用

```
obj0 = QObject()
obj1 = QObject()
obj2 = QObject()
obj3 = QObject()
obj4 = QObject()
obj5 = QObject()

print("obj0", obj0)
print("obj1", obj1)
print("obj2", obj2)
print("obj3", obj3)
print("obj4", obj4)
print("obj5", obj5)

obj1.setParent(obj0)
obj3.setParent(obj1)

obj2.setParent(obj0)
obj4.setParent(obj2)
obj5.setParent(obj2)
```



```
print("==obj4===", obj4.parent())#我们测试下Obj4父节点是谁
```

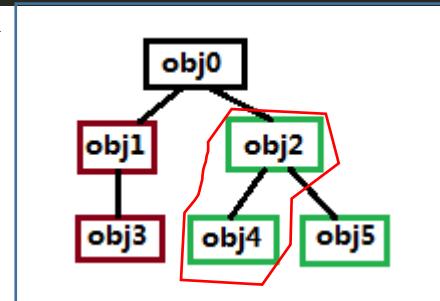
根据 setParent 设置，父子 Qobject 节点关系符号上图要求

```

obj0 <PyQt5.QtCore.QObject object at 0x00000000002F5B318>
obj1 <PyQt5.QtCore.QObject object at 0x00000000002F5B3A8>
obj2 <PyQt5.QtCore.QObject object at 0x00000000002F5B438>◀
obj3 <PyQt5.QtCore.QObject object at 0x00000000002F5B4C8>
obj4 <PyQt5.QtCore.QObject object at 0x00000000002F5B558>
obj5 <PyQt5.QtCore.QObject object at 0x00000000002F5B5E8>
====obj4==== <PyQt5.QtCore.QObject object at 0x00000000002F5B438>

```

Obj4 地址的父对象正好是 obj2，符合父子对象图要求  
记住子对象只能有一个父对象，但是父对象可以有多个子对象。



QObject.children 函数使用，获取本对象下面的子对象

```

obj1.setParent (obj0)
obj3.setParent (obj1)

obj2.setParent (obj0)
obj4.setParent (obj2)
obj5.setParent (obj2)

print ("obj0 下面的子对象", obj0.children ())#获取Obj0下面的子对象

```

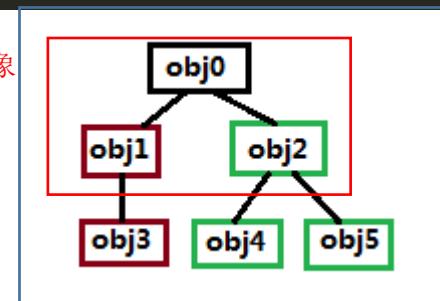
```

obj0 <PyQt5.QtCore.QObject object at 0x00000000002EFB318>
obj1 <PyQt5.QtCore.QObject object at 0x00000000002EFB3A8>◀
obj2 <PyQt5.QtCore.QObject object at 0x00000000002EFB438>◀
obj3 <PyQt5.QtCore.QObject object at 0x00000000002EFB4C8>
obj4 <PyQt5.QtCore.QObject object at 0x00000000002EFB558>
obj5 <PyQt5.QtCore.QObject object at 0x00000000002EFB5E8>
obj0 下面的子对象 [<PyQt5.QtCore.QObject object at 0x00000000002EFB3A8>, <PyQt5.QtCore.QObject object at 0x00000000002EFB438>]

```

获取的是 obj1 和 obj2 对象

证明了 children 函数只能获取本对象下面的直接子对象  
像 obj3 这些间接的，多级子对象是获取不到的



```
QObject.findChildren(对象参数) //查看本对象下面有多少个子对象
```

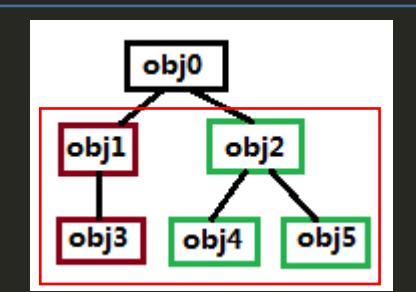
```
obj1.setParent(obj0)
```

```
obj3.setParent(obj1)
```

```
obj2.setParent(obj0)
```

```
obj4.setParent(obj2)
```

```
obj5.setParent(obj2)
```



```
print(obj0.findChildren(QObject)) #查看obj0下面有多少个子节点
```

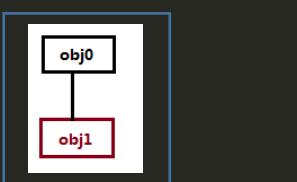
```
[<PyQt5.QtCore.QObject object at 0x00000000002EDB3a8>, <PyQt5.QtCore.QObject object at 0x00000000002EDB4c8>, <PyQt5.QtCore.QObject object at 0x00000000002EDB438>, <PyQt5.QtCore.QObject object at 0x00000000002EDB558>, <PyQt5.QtCore.QObject object at 0x00000000002EDB5E8>]
```

Obj0 下面有 5 个子对象

这种对象父子关系查询搜索主要用于界面的内存管理

```
obj0 = QObject()
```

```
obj1 = QObject()
```



```
obj1.setParent(obj0) #设置obj0位父对象
```



```
def func():
    obj0 = QObject()
    obj1 = QObject()

    obj1.setParent(obj0) #设置obj0位父对象

    obj1.destroyed.connect(lambda : print("obj1对象被释放了"))#监听obj1是否被释放

func()
```

我们监听发现，当函数执行完，父对象释放后，子对象 obj1 也释放了

创建两个窗口来演示父子对象，也就是父子控件的关系

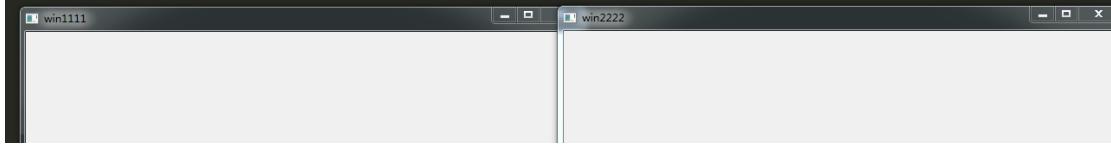
```
pyqt5.py
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)

win1 = QWidget()
win1.setWindowTitle("win1111")
win1.show()

win2 = QWidget()
win2.setWindowTitle("win2222")
win2.show()

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



成功创建两个窗口

下面我想把 win2 做成 win1 的子空间，也就是子对象。

```
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)

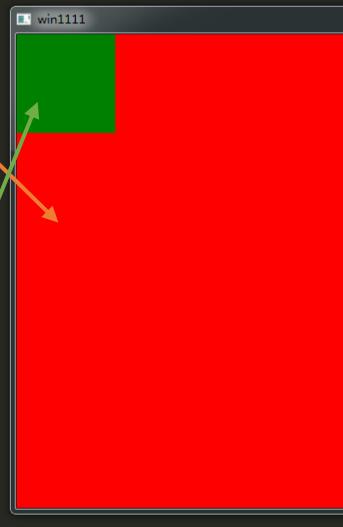
win1 = QWidget()
win1.setWindowTitle("win1111")
win1.setStyleSheet("background-color: red;")
win1.show()

win2 = QWidget()
win2.setWindowTitle("win2222")
win2.setStyleSheet("background-color: green;")

win2.setParent(win1)
win2.resize(100, 100)

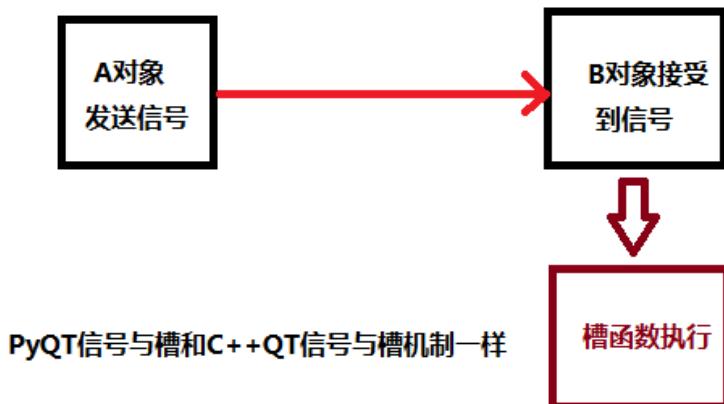
win2.show()

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



现在 win2 就是 win1 窗口的子控件，嵌入进 win1 窗口了。在 QT4.7 userguide 文档我也讲过类似的窗口嵌套关系。所以这也就符合上面讲的父对象里面嵌套子对象节点。

## QObject 信号与槽



QPushButton.clicked.connect(槽函数) //按钮信号与槽的使用

clicked 就是信号, pyQt 把信号做进了按钮类, 这样用起来方便, 你只需要写槽函数就是了

```
from PyQt5.Qt import *
import sys
```

这就是自己随便写的槽函数

```
def caofunc():#定义个槽函数
    print("槽函数执行")
```

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

```
win = QWidget()
button = QPushButton(win)
button.setText("按钮")
```

```
button.clicked.connect(caofunc) #链接槽函数
```

```
win.show()
```

将槽函数的函数名放入按钮的 connect  
信号槽链接函数即可

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

槽函数执行  
槽函数执行



点击按钮就会执行槽函数

## 对象类型判定的应用

对象.isWidgetType() //判断一个对象是不是控件

```
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)

obj = QObject()
w = QWidget()
btn = QPushButton()
label = QLabel()

print(obj.isWidgetType()) #不是控件返回False
print(w.isWidgetType()) #是控件返回True
print(btn.isWidgetType()) #是控件返回True
print(label.isWidgetType()) #是控件返回True

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

```
False
True
True
True
```

对象.inherits(类名) 判断对象是否继承某个类，继承返回 True，否则返回 False

```
obj = QObject()
w = QWidget()
btn = QPushButton()
label = QLabel()

print(obj.inherits("QWidget")) # 是不是继承至QWidget
print(w.inherits("QWidget")) # 是不是继承至QWidget
print(btn.inherits("QWidget")) # 是不是继承至QWidget
print(label.inherits("QWidget")) # 是不是继承至QWidget
```

```
False
True
True
True
```

```
super().__init__() 语法使用
```

```
from PyQt5.Qt import *
import sys
```

```
class testcls(QWidget):
```

```
    def __init__(self):
```

```
        super().__init__() # 调用父类,也就是QWidget类的__init__()函数初始化
```

```
        btn = QPushButton(self) # self就类似C++ this, 将按钮嵌入这个testcls类创建的窗口
```

```
        btn.move(0, 0)
```

```
        btn.setText("按钮")
```

```
        label1 = QLabel(self)
```

```
        label1.move(10, 30)
```

```
        label1.setText("标签111")
```

```
        label2 = QLabel(self)
```

```
        label2.move(10, 50)
```

```
        label2.setText("标签222")
```

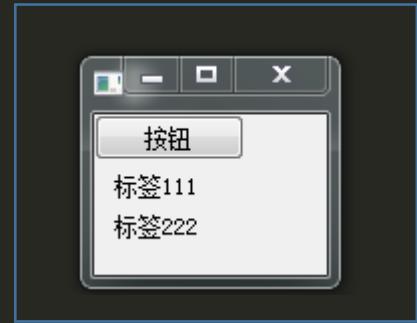
```
app = QApplication(sys.argv)
```

```
xzzobj = testcls()
```

```
xzzobj.show()
```

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

在 pyQT5 创建的类里面初始化控件,一定要加入  
super().\_\_init\_\_(), 否则会编译报错



我现在想获取 xzzobj 窗口里面所有 QLabel 对象的属性。

```
class testcls(QWidget):
```

```
    def __init__(self):
```

```
        super().__init__() # 调用父类,也就是QWidget类的__init__()函数初始化
```

```
        btn = QPushButton(self) # self就类似C++ this, 将按钮嵌入这个testcls类创建的窗口
```

```
        btn.move(0, 0)
```

```
        btn.setText("按钮")
```

```
        label1 = QLabel(self)
```

```
        label1.move(10, 30)
```

```
        label1.setText("标签111")
```

```
        label2 = QLabel(self)
```

```
        label2.move(10, 50)
```

```
        label2.setText("标签222")
```

我用 children 获取该类的所有控件

```
for labe in self.children():
```

```
    if labe.inherits("QLabel"):  
        print("True")
```

然后判断每次获取的控件是不是 QLabel,  
如果是就打印出来,或者你提取出来直接  
执行 QSS 的 setStyleSheet 批量修改样式

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

```
xzzobj = testcls()
```

```
xzzobj.show()
```

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

这就是 inherits 的实际用途, 可以批量修改一些东西。

```
label2.move(10, 50)
```

```
label2.setText("标签222")
```

你看, 直接批量修改标签样式

```
for labe in self.children():
```

```
    if labe.inherits("QLabel"):  
        labe.setStyleSheet("background-color:red;")
```



## QWidget 窗口使用

QWidget. resize(长,宽) //设置 Widget 窗口大小  
QWidget. move(x,y) //移动 Widget 窗口位置  
QWidget. setStyleSheet("写入 QSS 代码") //QSS 代码设置窗口样式, 颜色, 造型

```
app = QApplication(sys.argv)

w1 = QWidget()
w1.resize(500, 500) # 设置w1窗口长宽

w2 = QWidget(w1)
w2.resize(200, 200) # 设置w2窗口长宽
w2.setStyleSheet("background-color:blue;") # 设置w2背景颜色
w2.move(100, 100) # w2窗口在w1窗口上, 向右下移动了100个像素px

w1.show()

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

QWidget.x() #获取窗口 x 位置,但是有个问题要注意  
QWidget.y() #获取窗口 y 位置, 但是有个问题要注意  
QWidget. width() #获取窗口宽度  
QWidget. height() #获取窗口高度

```
w1 = QWidget()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

print("w1窗口x坐标 = ", w1.x())
print("w1窗口y坐标 = ", w1.y())
print("w1窗口宽度 = ", w1.width())
print("w1窗口高度 = ", w1.height())

w1.show()
```

```
w1窗口x坐标 = 100
w1窗口y坐标 = 150
w1窗口宽度 = 500
w1窗口高度 = 500
```

这个 x 和 y 坐标看起没有问题  
但是 x 和 y 坐标的计算方法我们得详细说一下



```
QWidget.geometry() # geometry 读取窗口的坐标是从窗口内部计算
app = QApplication(sys.argv)
```

```
w1 = QWidget()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)
```

```
print("geometry x坐标 y坐标", w1.geometry())
w1.show()
```

geometry x坐标 y坐标 PyQt5.QtCore.QRect(100, 150, 500, 500)



### QWidget 鼠标操作

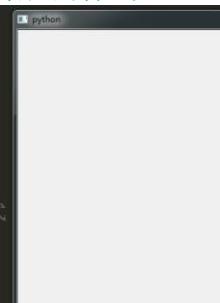
```
QWidget().setCursor(Qt.BusyCursor) //设置窗口的鼠标为繁忙的样式
```

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

```
w1 = QWidget()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)
```

```
w1.setCursor(Qt.BusyCursor) #设置鼠标移到到该窗口是繁忙的样式
w1.show()
```

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



```
QWidget().setCursor(Qt.ForbiddenCursor) //鼠标在窗口显示 xx 样式
```

```
QWidget().setCursor(Qt.CrossCursor) //鼠标在窗口显示+样式
```

以上程序自己运行就是，很简单

Shape	Qt::CursorShape	Value	Cursor Name	Shape	Qt::CursorShape	Value	Cursor Name
↑	Qt::ArrowCursor	left_ptr	↑	Qt::SizeVerCursor	size_ver		
↑	Qt::UpArrowCursor	up_arrow	↔	Qt::SizeHorCursor	size_hor		
+	Qt::CrossCursor	cross	↗	Qt::SizeBDiagCursor	size_bdiag		
〔〕	Qt::IBeamCursor	ibeam	↖	Qt::SizeFDiagCursor	size_fdiag		
〔〕	Qt::WaitCursor	wait	⊕	Qt::SizeAllCursor	size_all		
↙	Qt::BusyCursor	left_ptr_watch	⤻	Qt::SplitVCursor	split_v		
🚫	Qt::ForbiddenCursor	forbidden	⤻	Qt::SplitHCursor	split_h		
👉	Qt::PointingHandCursor	pointing_hand	👉	Qt::OpenHandCursor	openhand		
❓	Qt::WhatsThisCursor	whats_this	?	Qt::ClosedHandCursor	closedhand		

这就是鼠标样式的枚举值列表，自己实验

如果系统默认鼠标样式形状不够用，可以自己加入图片来设置鼠标样式

QCursor 鼠标类使用

```
• 2 import sys
  3
  4
• 5 app = QApplication(sys.argv)
  6
  7
• 8 w1 = QWidget()
  9 w1.resize(500, 500) # 设置w1窗口长宽
10 w1.move(100, 150)
11
• 12 pix = QPixmap("D:\\02 我的文档\\Desktop\\timg.jpg")
13
14
• 15 cursor = QCursor(pix) #创建一个鼠标对象
16
17 w1.setCursor(cursor)
18 w1.show()
19
20 sys.exit(app.exec_())
21
```

1. 创建 QPixmap 图片对象把图片导入进来

2. 把图片对象赋值给鼠标类，这时候就不用鼠标默认枚举了，直接传入图片对象就是

3. 打开窗口鼠标样式功能

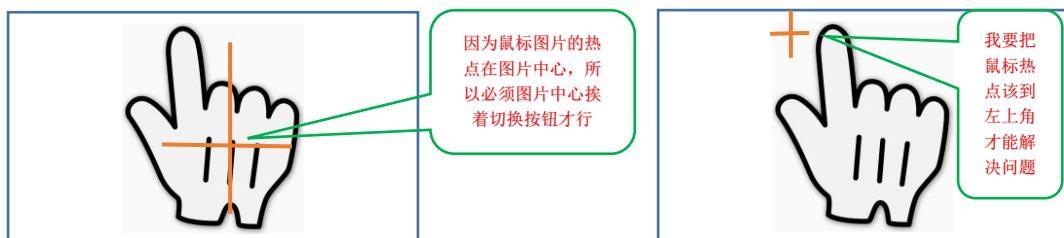
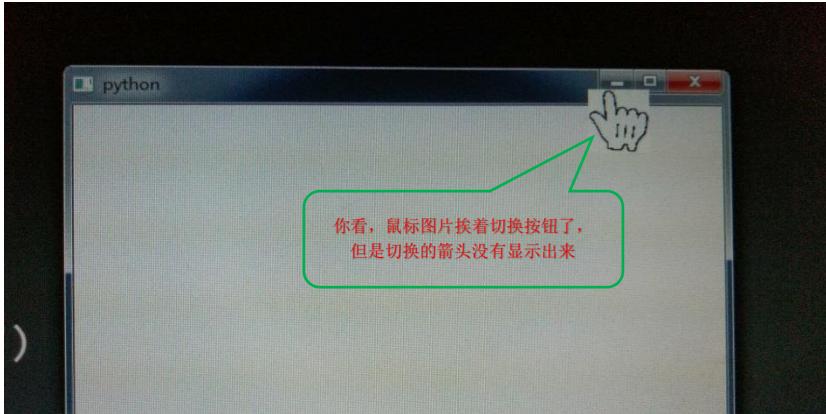
如果我不想鼠标这么大，怎么办？

```
3
4
• 5 app = QApplication(sys.argv)
6
7
• 8 w1 = QWidget()
9 w1.resize(500, 500) # 设置w1窗口长宽
10 w1.move(100, 150)
11
• 12 pix = QPixmap("D:\\02 我的文档\\Desktop\\timg.jpg")
13
14 zoom = pix.scaled(50, 50) # 将图片进行缩放，然后把缩放的图片返回给新的对象
15
• 16 cursor = QCursor(zoom) # 这次传入缩放后的图片对象
17
18 w1.setCursor(cursor)
19 w1.show()
20
21 sys.exit(app.exec_())
```

1. 把图片缩小

2. 记住不要放入 pix，因为 pix 还是原始图像的大小，要放入缩放后的 zoom

现在鼠标图片还有一个问题



```
pyqt5.py
1 from PyQt5.Qt import *
2 import sys
3
4
5 app = QApplication(sys.argv)
6
7
8 w1 = QWidget()
9 w1.resize(500, 500) # 设置w1窗口长宽
10 w1.move(100, 150)
11
12 pix = QPixmap("D:\\02 我的文档\\Desktop\\timg.jpg")
13
14 zoom = pix.scaled(50, 50) # 将图片进行缩放，然后把缩放的图片返回给新的对象
15
16 cursor = QCursor(zoom,0,0) # 这次传入缩放后的图片对象，鼠标热点在坐标0,0，就是左上角位置
17
18 w1.setCursor(cursor)
19 w1.show()
20
21 sys.exit(app.exec_())
```

```
pyqt5.py
1 from PyQt5.Qt import *
2 import sys
3
4
5 app = QApplication(sys.argv)
6
7
8 w1 = QWidget()
9 w1.resize(500, 500) # 设置w1窗口长宽
10 w1.move(100, 150)
11
12 pix = QPixmap("D:\\02 我的文档\\Desktop\\timg.jpg")
13
14 zoom = pix.scaled(50, 50) # 将图片进行缩放，然后把缩放的图片返回给新的对象
15
16 cursor = QCursor(zoom,0,0) # 这次传入缩放后的图片对象，鼠标热点在坐标0,0，就是左上角位置
17
18 w1.setCursor(cursor)
19 w1.show()
20
```

```
cursor.pos() #获取鼠标在桌面的 xy 位置
```

The screenshot shows a Sublime Text editor with a file named 'pyqt5.py'. The code is as follows:

```
4
5 app = QApplication(sys.argv)
6
7
8 w1 = QWidget()
9 w1.resize(500, 500) # 设置w1窗口长宽
10 w1.move(100, 150)
11
12 basepox = w1.cursor() #获取鼠标在整个桌面的相对位置
13 print(basepox.pos()) #pos()里面就是存放的鼠标在桌面的x, y位置
14
15 w1.show()
16
17 sys.exit(app.exec_())
18
```

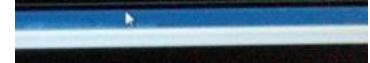
Two callout boxes point to specific lines of code:

- A yellow box points to line 12: "获取窗口的鼠标对象" (Get the cursor object of the window).
- A yellow box points to line 13: "打印出鼠标在桌面的位置" (Print the mouse position on the desktop).

To the right of the code editor is a terminal window titled 'python' showing the output of the script.

PyQt5.QtCore.QPoint(918, 0)

打印结果正确鼠标就在 x=918, y=0 的位置



鼠标移动过程中实时获取鼠标坐标

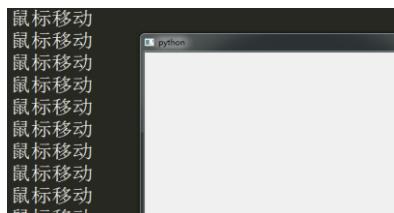
```
class Mywindow(QWidget):
    def mouseMoveEvent(self, event): # 这是鼠标事件触发回调函数, 固定模式
        print("鼠标移动")

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)
w1.setMouseTracking(True) # 启动鼠标在w1窗口移动自动执行mouseMoveEvent回调函数

w1.show()

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



只要鼠标在窗口移动，就会触发 mouseMoveEvent 回调函数

## mouseMoveEvent 获取鼠标在窗口的相对位置

mouseMoveEvent(对象参数, 接受鼠标数据变量).localPos() #返回鼠标现在在窗口的位置

```
from PyQt5.Qt import *
import sys
import time

class Mywindow(QWidget):

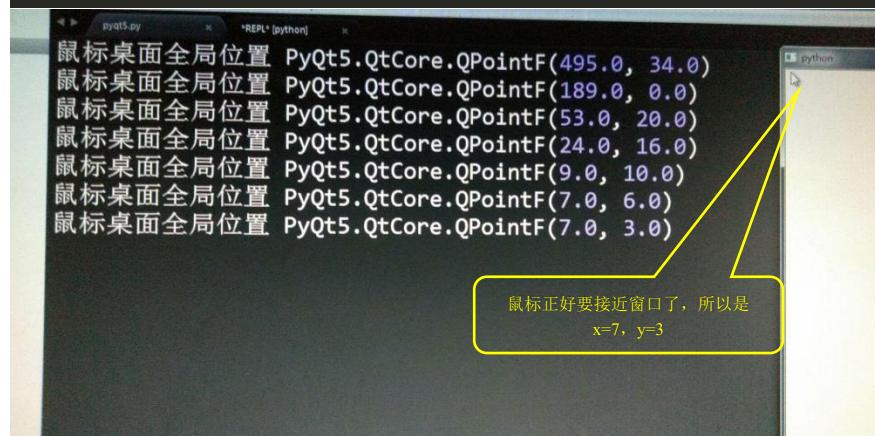
    def mouseMoveEvent(self, event): # 这是鼠标事件触发回调函数, 固定模式
        print("鼠标桌面全局位置", event.localPos())
        time.sleep(0.1)
        #这里加延时是因为sublime print输出太快会导致软件卡死, 这是sublime软件原因

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)
w1.setMouseTracking(True) # 启动鼠标在w1窗口移动自动执行mouseMoveEvent回调函数

w1.show()

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



鼠标实时坐标值获取出来有什么用? 下面来做个案例, 标签跟随鼠标移动  
self.findChaild(查找什么类) #父类里面使用的函数, 用来提取子类对象

```
class Mywindow(QWidget):

    def mouseMoveEvent(self, event): # 这是鼠标事件触发回调函数, 固定模式
        #print("鼠标桌面全局位置", event.localPos())
        #time.sleep(0.1) # 因为不使用print卡死输出, 所以不用延时
        labe = self.findChild(QLabel)#父类查找子类
        labe.move(event.localPos().x(), event.localPos().y()) #提取鼠标坐标的具体x和y值

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)
w1.setMouseTracking(True) # 启动鼠标在w1窗口移动自动执行mouseMoveEvent回调函数

label = QLabel(w1)
label.setText("鼠标移动实时获取")

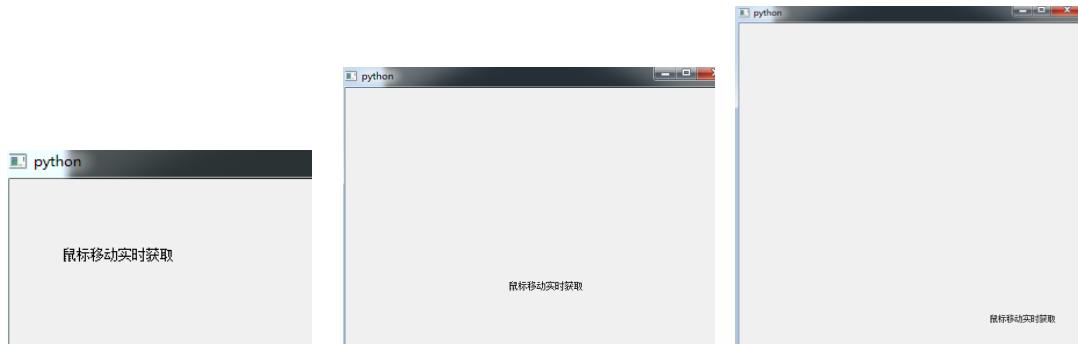
w1.show()

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

3.因为有子类继承我 Mywindow 父类创建的 w1 对象, 那么 w1 就是父对象, 所以我在父类用 self.findChild 去找到子类, 将其对象提取出来, 然后用 move 操作这个子类对象

2.因为 label 标签对象不是全局的, 但是我又想在其它类中操作 label 对象, 那么我必须让 label 对象继承至父类, 在这个指定继承的父类里面去操作它, 其它类还是不行哦

1.创造 label 标签对象



## 窗口当前状态事件消息

showEvent()回调函数, closeEvent()回调函数

```
class Mywindow(QWidget):
    def showEvent(self, QShowEvent): #事件消息固定函数, 这是窗口运行调用函数
        print("窗口打开")

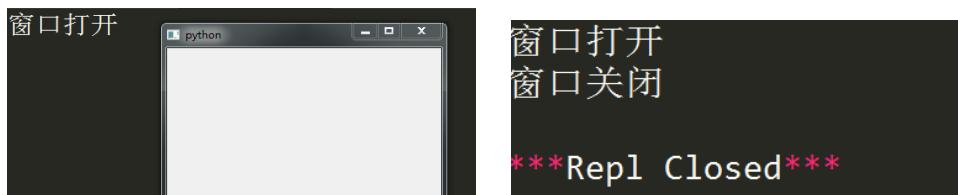
    def closeEvent(self, QCloseEvent):#事件消息固定函数, 这是窗口关闭调用函数
        print("窗口关闭")

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

w1.show()

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



moveEvent #窗口移动回调函数, resizeEvent #窗口大小变化回调函数

```
class Mywindow(QWidget):
    def moveEvent(self, QMoveEvent):
        print("窗口被移动了")

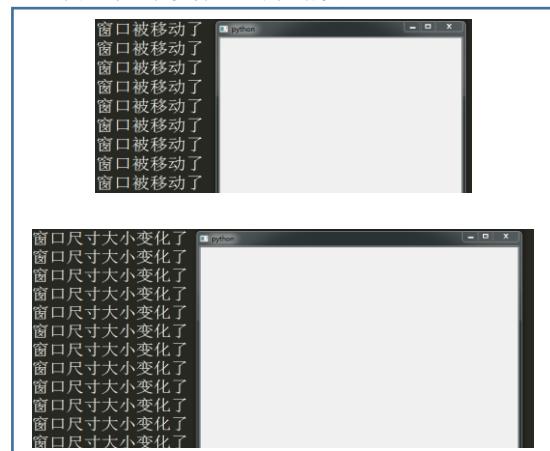
    def resizeEvent(self, QResizeEvent):
        print("窗口尺寸大小变化了")

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

w1.show()

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



```

enterEvent #鼠标进入窗口回调函数, leaveEvent #鼠标离开窗口回调函数
class Mywindow(QWidget):

    def enterEvent(self, QEvent):
        print("鼠标进来了")

    def leaveEvent(self, QEvent):
        print("鼠标离开窗口")

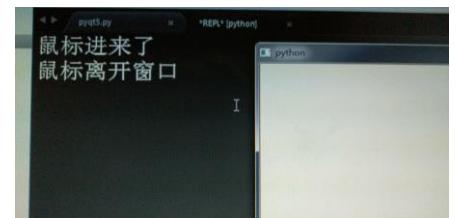
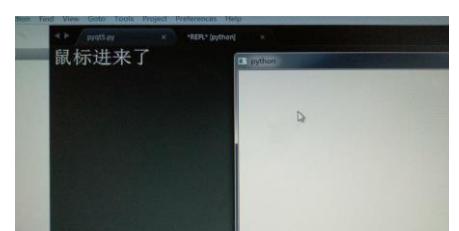
app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

w1.show()

sys.exit(app.exec_())

```



mousePressEvent #鼠标按下回调函数, mouseReleaseEvent #鼠标释放回调函数,  
mouseDoubleClickEvent #鼠标双击回调函数

```

class Mywindow(QWidget):

    def mousePressEvent(self, QMouseEvent):
        print("鼠标被按下")

    def mouseReleaseEvent(self, QMouseEvent):
        print("鼠标释放了")

    def mouseDoubleClickEvent(self, QMouseEvent):
        print("鼠标双击了")

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

w1.show()

sys.exit(app.exec_())

```

鼠标被按下  
鼠标释放了  
鼠标被按下  
鼠标释放了  
鼠标双击了  
鼠标释放了

测试成功

```

keyPressEvent #键盘按下回调函数, keyReleaseEvent #键盘松开回调函数
class Mywindow(QWidget):
    def keyPressEvent(self, QKeyEvent):
        print("键盘某个键按下")

    def keyReleaseEvent(self, QKeyEvent):
        print("键盘某个键松开")

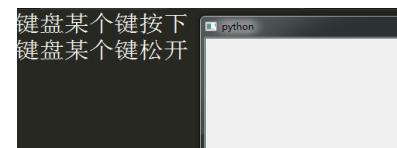
app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500) # 设置w1窗口长宽
w1.move(100, 150)

w1.show()

sys.exit(app.exec_())

```



父子窗口里面多个空间消息转发问题，也就是消息产生后不知道转发给哪个窗口？

```

class Mywindow(QWidget):
    def mousePressEvent(self, QMouseEvent):
        print("顶层父窗口鼠标按下")

class widwindow(QWidget):
    def mousePressEvent(self, QMouseEvent):
        print("子窗口按下")

class label(QLabel):
    def mousePressEvent(self, QMouseEvent):
        print("标签被按下")

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500)
w1.move(100, 150)

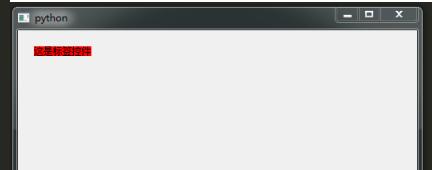
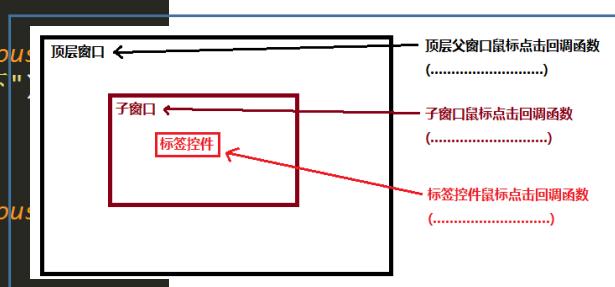
subwin = widwindow(w1) # subwin子窗口放在w1父窗口里面
subwin.resize(300, 300)
subwin.move(20, 20)
subwin.setStyleSheet("background-color:yellow;")

lab = label(subwin)#label标签放入子窗口里面
lab.setText("这是标签控件")
lab.setStyleSheet("background-color:red;")

w1.show()

sys.exit(app.exec_())

```



运行发现子窗口黄色背景没有显示出来

```

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500)
w1.move(100, 150)

subwin = widwindow(w1) # subwin子窗口放在w1父窗口里面
subwin.resize(300, 300)
subwin.move(20, 20)
subwin.setAttribute(Qt.WA_StyledBackground, True)
#setAttribute 解决QSS在子窗口的问题
subwin.setStyleSheet("background-color:yellow;")

lab = label(subwin) # label标签放入子窗口里面
lab.setText("这是标签控件")
lab.setStyleSheet("background-color:red;")

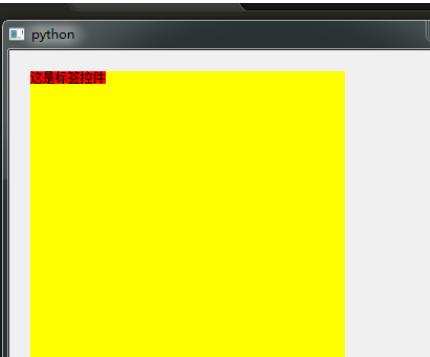
w1.show()

sys.exit(app.exec_())

```



子窗口按下  
顶层父窗口鼠标按下  
标签被按下



你点击父窗口，子窗口，标签控件区域就会打印对应区域的回调函数。

```

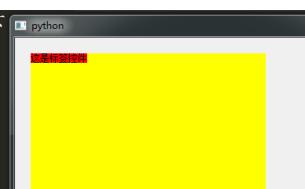
class Mywindow(QWidget):
    def mousePressEvent(self, QMouseEvent):
        print("顶层父窗口鼠标按下")

class widwindow(QWidget):
    def mousePressEvent(self, QMouseEvent):
        print("子窗口按下")

class label(QLabel):
    pass
    # def mousePressEvent(self, QMouseEvent):
    #     print("标签被按下")

```

如果 Label 鼠标点击回调函  
数取消



那么你点击红色标签的时候就会去执行 label 的父类，也就是子窗口的回调函数。所以这就符合了前面图中所讲的消息层层转发的机制，如果底层没实现，就会调用底层父类实现的相同回调函数执行，如果子窗口也没有实现回调，就会调用父类的相同回调函数，这就是层层上传。除非你底层 label 不继承子窗口 subwin 对象。

```

app = QApplication(sys.argv)

w1 = Mywindow()
w1.resize(500, 500)
w1.move(100, 150)

subwin = widwindow(w1) # subwin子窗口放在w1父窗口里面
subwin.resize(300, 300)
subwin.move(20, 20)
subwin.setAttribute(Qt.WA_StyledBackground, True)
# setAttribute 解决QSS在子窗口的问题
subwin.setStyleSheet("background-color:yellow;")

#lab = label(subwin) # label标签放入子窗口里面
lab = QLabel(subwin) # 把标签对象换成QLabel生成
lab.setText("这是标签控件")
lab.setStyleSheet("background-color:red;")

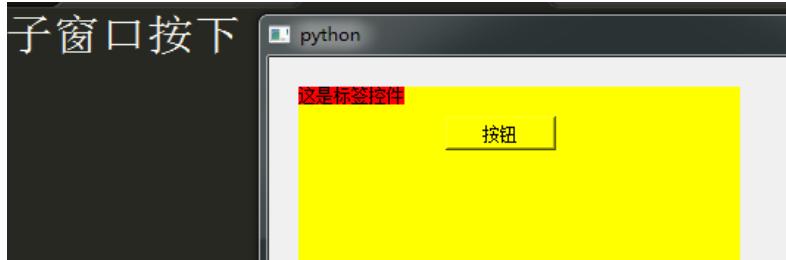
btn = QPushButton(subwin)
btn.setText("按钮")
btn.move(100, 20)

w1.show()

```

我将 QLabel 也继承到子窗口

我将按钮继承至子窗口，你将发现按钮按下类似鼠标按子窗口回调函数没有打印



为什么红色的标签会打印子窗口回调函数，反而按钮不会打印子窗口回调函数呢？

因为标签就不是需要鼠标点击的控件，所以 pyQT 内部没有实现标签事件处理回调函数。但是按钮是需要鼠标点击的空间，所以 pyQT 类别执行了事件处理回调函数，那么子窗口回调函数就不用处理了。

## 多窗口层级管理，就是底层窗口显示到前面来

### QLabel 使用

```

from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("层级关系")
window.resize(500,500)

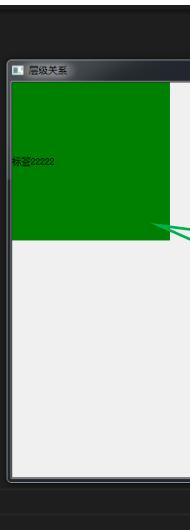
label1 = QLabel(window) #红色标签
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")

label2 = QLabel(window) #绿色标签
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")

window.show()
sys.exit(app.exec_())

```

你看，我后面创建的  
label2 标签挡住了前面  
label1 红色的标签



```

app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("层级关系")
window.resize(500,500)

label1 = QLabel(window) #红色标签
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")

label2 = QLabel(window) #绿色标签
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")
label2.move(100,100) #移动下label2

window.show()

```



方法 1，将顶层对象，移动到底层。

`QLabel.lower() //将第 1 层对象向下移动一层`

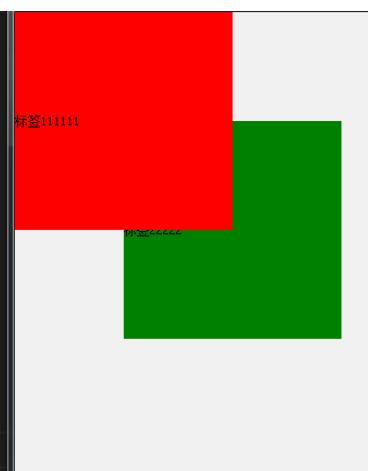
```

window.setWindowTitle("层级关系")
window.resize(500,500)

label1 = QLabel(window) #红色标签
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")

label2 = QLabel(window) #绿色标签
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")
label2.move(100,100) #移动下label2
label2.lower() #将标签2移动到后面

```



第 2 种方法，将底层对象移动到顶层

`QLabel.raise_() //将第 2 层向上移动`

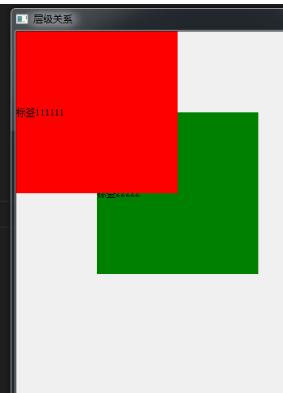
```

window.setWindowTitle("层级关系")
window.resize(500,500)

label1 = QLabel(window) #红色标签
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")
label1.raise_() #底层向上层移动

label2 = QLabel(window) #绿色标签
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")
label2.move(100,100) #移动下label2

```



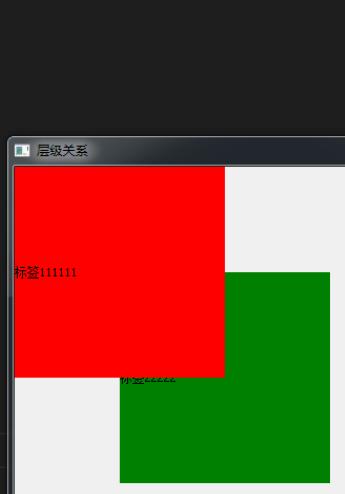
### 第3中方法

QLabel.stackUnder( 放入顶层的对象 QLabel )

```
label1 = QLabel(window) #红色标签
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")

label2 = QLabel(window) #绿色标签
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")
label2.move(100,100) #移动下label2

label2.stackUnder(label1) #将label1放入label2的顶层
```



如果我想鼠标点击某个窗口，某个窗口就出现在顶层，如何实现？

```
class Label(QLabel):
    def mousePressEvent(self,event): #鼠标点击回调函数
        self.raise_() #将对象放入顶层

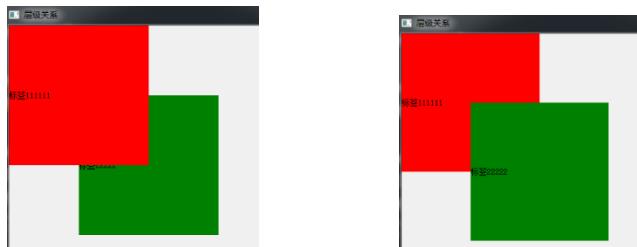
app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("层级关系")
window.resize(500,500)

label1 = Label(window) #将QLabel换掉
label1.setText("标签111111")
label1.resize(200,200)
label1.setStyleSheet("background-color:red;")

label2 = Label(window) #将QLabel换掉
label2.setText("标签222222")
label2.resize(200,200)
label2.setStyleSheet("background-color:green;")
label2.move(100,100) #移动下label2
```

主要就是创建点击后的  
回调函数，用这个回调  
类去创建 QLabel 对象

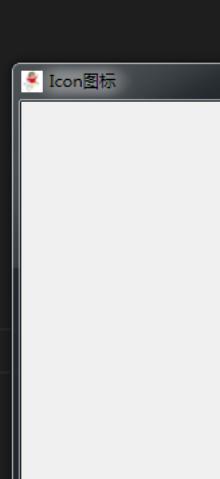
部分代码



## QIcon 使用，添加图标

```
对象 = QIcon(" 图片路径 ") //用 QIcon 导入图像，然后返回给对象  
对象.setWindowIcon(" QIcon") //主窗口左上边图标，用 QIcon 对象导入
```

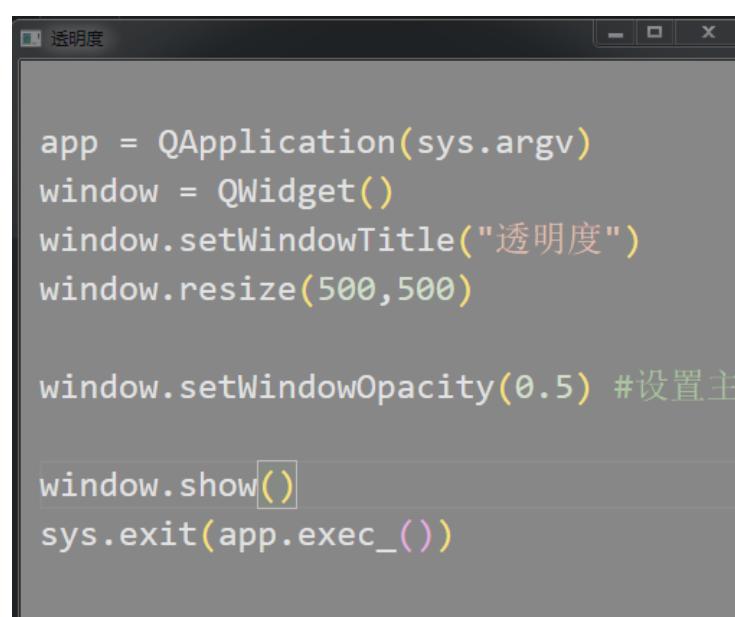
```
app = QApplication(sys.argv)  
window = QWidget()  
window.setWindowTitle("Icon图标")  
window.resize(500,500)  
  
icon = QIcon("timg.jpg") #用 QIcon 导入图标  
window.setWindowIcon(icon) #显示在主窗口左上角  
  
window.show()  
sys.exit(app.exec_())
```



## QWidget 窗口透明度测试

```
QWidget.setWindowOpacity( 透明度参数 ) //透明度在 1~0 之间，0 全透明，1 不透明
```

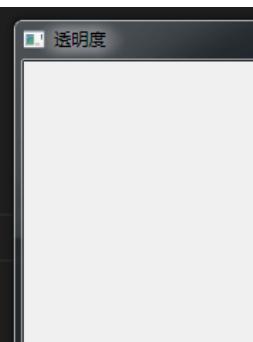
```
app = QApplication(sys.argv)  
window = QWidget()  
window.setWindowTitle("透明度")  
window.resize(500,500)  
  
window.setWindowOpacity(0.5) #设置主窗口透明度
```



设置 1，完全不透明

```
app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("透明度")
window.resize(500,500)

window.setWindowOpacity(1) #设置主窗口透明度
```



设置 0.1，基本完全透明

```
app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("透明度")
window.resize(500,500)

window.setWindowOpacity(0.1) #设置主窗口透明度

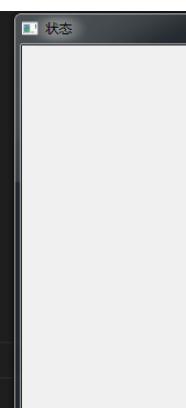
window.show()
sys.exit(app.exec_())
```

## QWidget 窗口状态获取

QWidget.windowState() //获取窗口状态，窗口无状态参数 Qt.WindowNoState

```
app = QApplication(sys.argv)
window = QWidget()
window.setWindowTitle("状态")
window.resize(500,500)

window.show()
print(window.windowState() == Qt.WindowNoState)#获取窗口状态
sys.exit(app.exec_())
```



窗口无状态，得到 **True**

```
QWidget.setWindowState( 参数 )//设置窗口最小化,  
参数, Qt.WindowMinimized 最小化
```

```
app = QApplication(sys.argv)  
window = QWidget()  
window.setWindowTitle("状态")  
window.resize(500,500)  
  
window.setWindowState(Qt.WindowMinimized) #窗口最小化  
window.show()  
  
sys.exit(app.exec_())
```



你看窗口打开了，只是看不到，因为窗口最小化了

```
QWidget.setWindowState(Qt.WindowFullScreen) #窗口最大化，全屏
```

```
app = QApplication(sys.argv)  
w1 = QWidget()  
w1.setWindowTitle("窗口1")  
w1.resize(500,500)  
  
w2 = QWidget()  
w2.setWindowTitle("窗口2")  
  
w1.show()  
w2.show()  
  
#window.setWindowState() #全屏  
  
sys.exit(app.exec_())
```

本来窗口是 w1 在后， w2 在前，后创建的在前很正常。

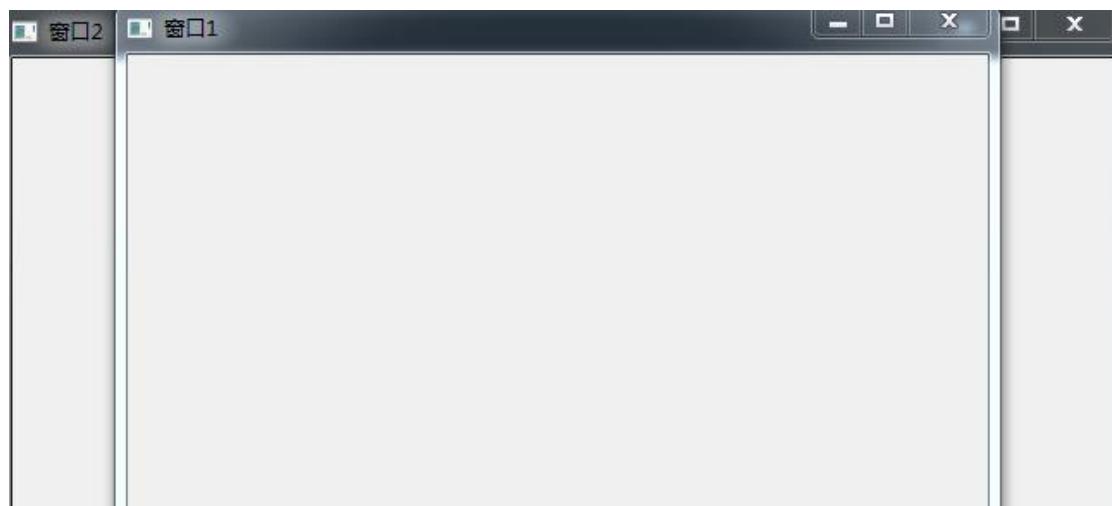
Qt.WindowActive 窗口焦点获取激活

```
app = QApplication(sys.argv)
w1 = QWidget()
w1.setWindowTitle("窗口1")
w1.resize(500,500)

w2 = QWidget()
w2.setWindowTitle("窗口2")

w1.show()
w2.show()
w1.setWindowState(Qt.WindowActive)#一定要设置在窗口显示后

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



结合 Qt.WindowFullScreen, Qt.WindowMinimized 制作一个无边框窗口

你看 QQ 就是无边框窗口

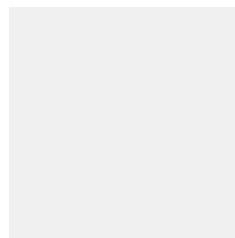


```
QWidget(flags = Qt.FramelessWindowHint) #取出窗口边框
```

```
app = QApplication(sys.argv)
w1 = QWidget(flags = Qt.FramelessWindowHint) #取出窗口边框
w1.setWindowTitle("窗口1")
w1.resize(500,500)

w1.show()

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



窗口无最大化，最小化边框了。

在无边框窗口上加控件

```
app = QApplication(sys.argv)
w1 = QWidget(flags = Qt.FramelessWindowHint)
w1.setWindowTitle("窗口1")
w1.resize(500,500)

button = QPushButton(w1) #在无边框窗口里创建关闭按钮
button.setText("关闭")

w1.show()

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





```
QWidget.width() #获取窗口控件宽度
QPushButton.width() #获取按钮宽度
app = QApplication(sys.argv)
w1 = QWidget(flags = Qt.FramelessWindowHint)
w1.setWindowTitle("窗口1")
w1.resize(500,500)

button = QPushButton(w1) #在无边框窗口里创建关闭按钮
button.setText("关闭")

button_W = button.width() #获取按钮大小
window_W = w1.width() #获取窗口大小
button.move((window_W-button_W),10) #窗口大小 - 按钮大小 , 就是按钮右移位置

w1.show()

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



```
QPushButton.pressed.connect( 函数参数 ) #按钮按下发送信号
QWidget.close() #关闭窗口
app = QApplication(sys.argv)
w1 = QWidget(flags = Qt.FramelessWindowHint)
w1.setWindowTitle("窗口1")
w1.resize(500,500)

button = QPushButton(w1) #在无边框窗口里创建关闭按钮
button.setText("关闭")

button_W = button.width() #获取按钮大小
window_W = w1.width() #获取窗口大小
button.move((window_W-button_W),10) #窗口大小 - 按钮大小 , 就是按钮右移位置

def close():
    w1.close()

button.pressed.connect(close) #按钮信号发送
w1.show()
sys.exit(app.exec_())
```

这样就可以点击关闭按钮，关闭窗口了

点击按钮，改变按钮显示状态

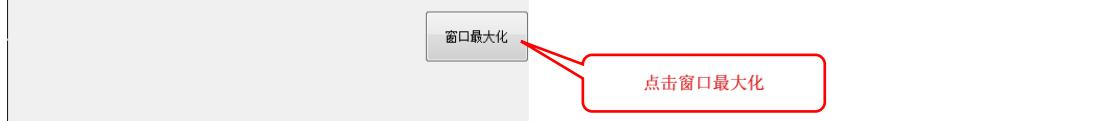
QWidget.isMaximized() #获取当前窗口状态

```
button = QPushButton(w1) #在无边框窗口里创建关闭按钮
button.setText("窗口最大化")
button.resize(100,50)

button_W = button.width() #获取按钮大小
window_W = w1.width() #获取窗口大小
button.move((window_W-button_W),10) #窗口大小 - 按钮大小 , 就是按钮右移位置

def window_max():
    if w1.isMaximized(): #判断当前窗口状态
        button.setText("窗口最大化")
        w1.showNormal() #窗口变小
    else:
        w1.showMaximized() #窗口最大化
        button.setText("窗口已经最大化")

button.pressed.connect(window_max) #按钮信号发送
w1.show()
sys.exit(app.exec_())
```



## 让你的窗口或者按钮，其它控件暂时无法使用。

QPushButton.setEnabled(参数) #参数 false 不可点击, 参数 True 可以点击默认也是可以点击。

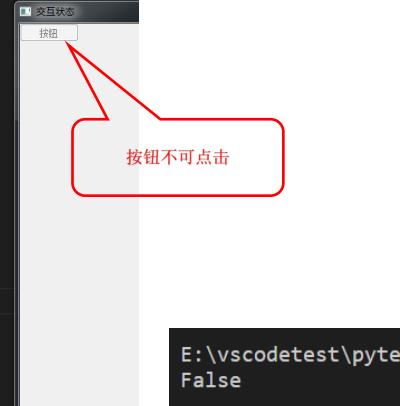
QPushButton.isEnabled() #返回按钮状态是可点击(True)还是不可点击(False)

```
app = QApplication(sys.argv)
w1 = QWidget()
w1.setWindowTitle("交互状态")
w1.resize(500,500)

button = QPushButton(w1) #创建按钮
button.setText("按钮")
button.pressed.connect(lambda : print("按钮被点击了"))
button.setEnabled(False) #让按钮处于不可点击状态

print(button.isEnabled()) #获取按钮现在的状态是可操作还是不可操作

w1.show()
sys.exit(app.exec_())
```



## 窗口编辑时状态显示，编辑后状态显示

```
QWidget.setWindowModified(True) #显示 setWindowTitle 里面的[*]星星符号
```

```
app = QApplication(sys.argv)

w1 = QWidget()
w1.setWindowTitle("交互状态[*]") #[*]显示**
w1.resize(500,500)
w1.setWindowModified(True) #让setWindowTitle里面[]中括号星星*显示出来

w1.show()
sys.exit(app.exec_())
```



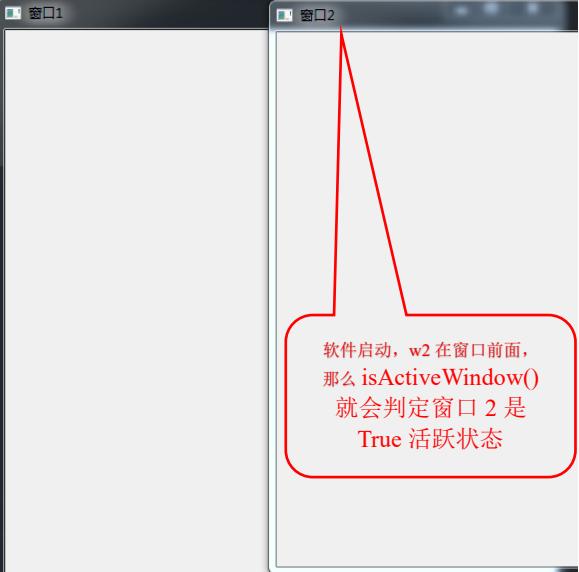
QWidget().isActiveWindow() #判断窗口当前是否被激活，返回 True 或者 False

```
app = QApplication(sys.argv)

w1 = QWidget()
w1.setWindowTitle("窗口1")
w1.resize(500,500)

w2 = QWidget()
w2.setWindowTitle("窗口2")
w2.resize(500,500)

w1.show()
w2.show()
print(w1.isActiveWindow())#查看窗口是否处于被点击或者操作状态
print(w2.isActiveWindow())
sys.exit(app.exec_())
```



False  
True

可以使用 isActiveWindow() 做很多窗口判断功能。

## 窗口界面交互设计

```
class window(QWidget): #window类继承QWidget功能
    def __init__(self): #self就是获取继承的QWidget功能
        super().__init__()
        self.setWindowTitle("主界面交互")
        self.resize(500,500)
        self.setup_ui()

    def setup_ui(self):
        label = QLabel(self) #self新建的标签放入继承的QWidget窗口
        label.setText("标签")
        label.move(100,50)

        ledit = QLineEdit(self) #self新建的编辑框放入继承的QWidget窗口
        ledit.setText("文本框")
        ledit.move(100,100)

        btn = QPushButton(self) #self新建的按钮框放入继承的QWidget窗口
        btn.setText("按钮")
        btn.move(100,150)

app = QApplication(sys.argv)
w = window() #创建window对象，就会把继承的QWidget创建出来
w.show()
sys.exit(app.exec_())
```



界面已经创建出来

监听文本框状态，看看文本框是否有字符输入？

`QLineEdit.textChanged.connect(函数参数)` #编辑框字符变化信号发送函数

函数参数：就是编辑框字符变化后，会自动执行槽函数，所以需要人为指定槽函数。

```
ledit = QLineEdit(self) #self新建的编辑框放入
ledit.setText("文本框")
ledit.move(100,100)

btn = QPushButton(self) #self新建的按钮框放入
btn.setText("按钮")
btn.move(100,150)

def text_cao(): #槽函数
    print("文本内容发生改变")

ledit.textChanged.connect(text_cao)
#编辑框信号必须在槽函数后面，才能传入槽函数地址
```

我给编辑框写了3个字符，编辑框槽函数执行3遍

文本内容发生改变  
文本内容发生改变  
文本内容发生改变

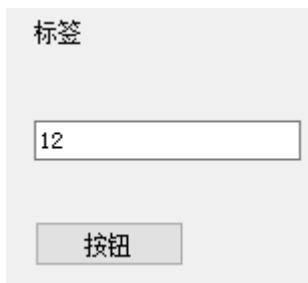
如何让按钮在编辑框输入字符后才能被点击

text 变量是编辑框系统默认变量，存放编辑框写入的字符，可以读出来使用  
QPushButton.setEnabled(参数) #参数 True 按钮可以点击，False 按钮不可以点击

```
btn.move(100,150)
btn.setEnabled(False) #默认按钮不可点击

def text_cao(text): #槽函数 text是编辑框系统默认变量
    if len(text) > 0:
        btn.setEnabled(True) #有字符输入编辑框，按钮可以点击
    else:
        btn.setEnabled(False)

redit.textChanged.connect(text_cao)
#编辑框信号必须在槽函数后面，才能传入槽函数地址
```



变量 = QLineEdit.text() #获取编辑框输入的字符，返回给变量

```
btn = QPushButton(self) #self新建的按钮框放入继承的QW
btn.setText("按钮")
btn.move(100,150)
btn.setEnabled(False) #默认按钮不可点击

def text_cao(text): #槽函数 text是编辑框系统默认变量
    ret = ledit.text() #获取输入的字符
    if ret == "XZ":
        btn.setEnabled(True)
    else:
        btn.setEnabled(False)

redit.textChanged.connect(text_cao)
#编辑框信号必须在槽函数后面，才能传入槽函数地址
```



随便输入字符按钮是不会有反应的

必须输入判断的”XZ”字符按钮才开点击

## 鼠标移动到窗口的时候，状态栏显示提示信息

用 QWidget 做窗口状态栏会编译报错，因为窗口状态栏是 QMainWindow 类实现的

```
app = QApplication(sys.argv)

w = QWidget()
w.setWindowTitle("信息提示案例")
w.resize(500,500)
w.setStatusBar() #这里会报错
w.setStatusTip("这是窗口") #setStatusTip状态栏提示信息函数
w.show()

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

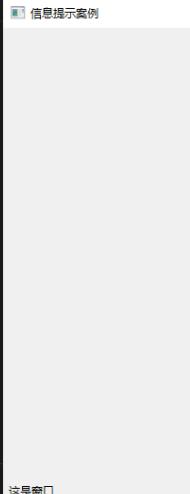
QMainWindow().statusBar() //给窗口设置状态栏

QMainWindow().setStatusTip("参数") //设置鼠标移动到窗口，状态栏显示得字符串

```
app = QApplication(sys.argv)

w = QMainWindow() #改成QMainWindow就可以了
w.setWindowTitle("信息提示案例")
w.resize(500,500)
w.setStatusBar() #设置状态栏
w.setStatusTip("这是窗口") #setStatusTip在状态栏写入提示信息
w.show()

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



鼠标移动到窗口里的控件，状态栏显示控件提示

QLabel.setStatusTip("参数") //设置鼠标移动到标签，状态栏显示字符

```
app = QApplication(sys.argv)

w = QMainWindow() #改成QMainWindow就可以了
w.setWindowTitle("信息提示案例")
w.resize(500,500)
w.setStatusBar() #设置状态栏
w.setStatusTip("这是窗口") #setStatusTip在状态栏写入提示信息

label = QLabel(w)
label.setText("标签控件")
label.setStatusTip("鼠标移动到标签") #设置鼠标移动到标签上，状态栏显示字符
w.show()
```



`QLabel.setToolTip("参数") //设置鼠标移动到标签，在标签旁边显示提示符`

```
label = QLabel(w)
label.setText("标签控件")
label.setStatusTip("鼠标移动到标签") #设置鼠标移动到标签上，状态栏显示字符
label.setToolTip("这是一个标签") #设置鼠标移动到标签，就在标签旁边显示提示符
w.show()
```



如果鼠标一直放在标签旁边，那么一直显示标签字符，感觉很挡事

`QLabel.setToolTipDuration(延时时间) //鼠标移动到标签，提示显示持续多久`

```
label = QLabel(w)
label.setText("标签控件")
label.setStatusTip("鼠标移动到标签") #设置鼠标移动到标签上，状态栏显示字符
label.setToolTip("这是一个标签") #设置鼠标移动到标签，就在标签旁边显示提示符
label.setToolTipDuration(500) #标签提示显示500ms
w.show()
```



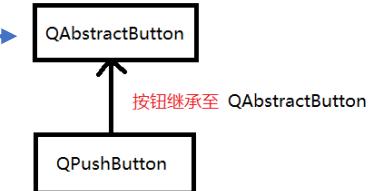
## QPushButton 控件实现的底层原理，QAbstractButton 按钮抽象基类

```
app = QApplication(sys.argv)

w = QMainWindow() #改成QMainWindow就可以了
w.setWindowTitle("QAbstractButton底层实现原理")
w.resize(500,500)

btn = QAbstractButton(w) #能否创建按钮?
btn.setText("AbstractButton类实验")

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



这种继承关系有什么用，为什么要研究 QAbstractButton？

我们先看看运行结果

```
D 10 btn = QAbstractButton(w) #能否创建按钮?

Exception has occurred: TypeError
PyQt5.QtWidgets.QAbstractButton represents a C++ abstract class and cannot be instantiated
```

你看报错，类型错误。说的是这种 C++ 实例类没有办法被抽象化。

那么我们就需要实例化 QAbstractButton 类，才能使用。怎么操作？

```
class BtnAbstract(QAbstractButton): #必须写个类继承QAbstractButton
    pass

btn = BtnAbstract(w) #能否创建按钮?
btn.setText("AbstractButton类实验")

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



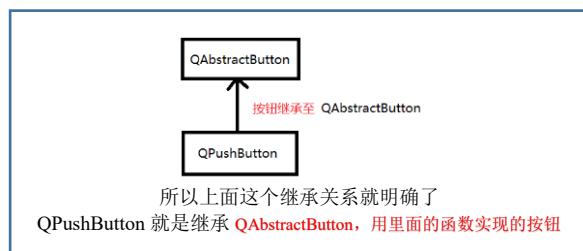
界面运行起来了，但是没有显示。

```
NotImplementedError: QAbstractButton.paintEvent() is abstract and must be overridden
```

提示没有实现 paintEvent 函数

```
QAbstractButton(QWidget *parent = Q_NULLPTR)
~QAbstractButton()
bool autoExclusive() const
bool autoRepeat() const
int autoRepeatDelay() const
int autoRepeatInterval() const
QButtonGroup * group() const
QIcon icon() const
 QSize iconSize() const
bool isCheckedable() const
bool isChecked() const
bool isDown() const
void setAutoExclusive(bool)
```

QAbstractButton 里面有这些函数，  
QPushButton 按钮就是继承 QAbstractButton  
类，然后使用里面的函数来实现按钮的

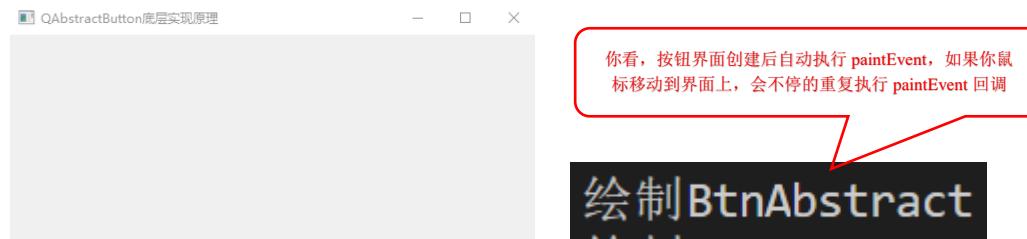


PaintEvent 就是继承 QAbstractButton 后，必须实现的函数，这个函数会被 QAbstractButton 回调。

```
class BtnAbstract(QAbstractButton): #必须写个类继承QAbstractButton
    def paintEvent(self, evr): #在类这里使用QAbstractButton里的paintEvent函数重新绘制
        print("绘制BtnAbstract")

btn = BtnAbstract(w) #能否创建按钮?
btn.setText("AbstractButton类实验")

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



完善 paintEvent 回调函数

```
画家 = QPainter(参数) #创建画家，返回画家对象。参数:表示在什么对象上画画
笔对象 = QPen(颜色, 画笔粗细) #创建画笔，返回画笔对象
QPainter().setPen(笔对象) #将画笔拿给画家
QPainter().drawEllipse(起始坐标 x, 起始坐标 y, 方形长, 方形宽) #系统会自动在这个方形里面做内切椭圆
```

```
class BtnAbstract(QAbstractButton):
    def paintEvent(self, evr):
        print("绘制BtnAbstract")
        painter = QPainter(self) #创建画家和纸
        pen = QPen(QColor(111,100,50),6) #QPen 给画家创建笔
        painter.setPen(pen) #把笔拿给画家

        painter.drawEllipse(0,0,100,100) #在纸上画椭圆

btn = BtnAbstract(w) #能否创建按钮?
btn.setText("AbstractButton类实验")

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



显示出来半圆了，看来是自定义按钮尺寸太小

```

btn = BtnAbstract(w) #能否创建按钮?
btn.setText("AbstractButton类实验")
btn.resize(200,200) #把自己创建的按钮对象尺寸变大, 可以看到完整的圆
w.show()
sys.exit(app.exec_())

```



你看, 按钮尺寸变大, 圆完整了。

QAbstractButton 也是鼠标点击对象之后, 自带信号功能的  
**Signals**

```

void clicked(bool checked = false)
void pressed()
void released()
void toggled(bool checked)

```

```

class BtnAbstract(QAbstractButton):
    def paintEvent(self, evr):
        #print("绘制BtnAbstract") #为了不影响点击输出, 屏蔽自动回调函数
        painter = QPainter(self) #创建画家和纸
        pen = QPen(QColor(111,100,50),6) #QPen 给画家创建笔
        painter.setPen(pen) #把笔拿给画家

        painter.drawEllipse(0,0,100,100) #在纸上画椭圆

btn = BtnAbstract(w) #能否创建按钮?
btn.setText("AbstractButton类实验")
btn.resize(200,200) #把自己创建的按钮对象尺寸变大, 可以看到完整的圆
btn.pressed.connect(lambda : print("btn被点击了"))
w.show()
sys.exit(app.exec_())

```



点击圆圈范围

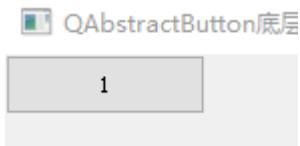
btn被点击了  
btn被点击了

你看, 显示我点击了。

这就是按钮设计的全过程, 也可以用 QAbstractButton 类设计其它控件。

## QPushButton 点击后自己修改自己

```
int 变量 = int(参数) #将字符串转换成十进制。参数:填入字符串  
str 字符串变量 = str(参数) #将十进制转换成字符串, 参数: 填入十进制  
  
w = QMainWindow() #改成QMainWindow就可以了  
w.setWindowTitle("QAbstractButton底层实现原理")  
w.resize(500,500)  
  
btn = QPushButton(w)  
btn.setText("1")  
  
def plus_one(): #槽函数放在信号前面  
    num = int(btn.text())+1 #QPushButton.text获取当前文本内容  
    btn.setText(str(num)) #将十进制转换成字符串  
  
btn.pressed.connect(plus_one) #信号与槽链接  
w.show()  
sys.exit(app.exec_())
```



默认是 1



计算点击次数, 我点击按钮 4 次

## QPushButton 显示图标

```
返回图标对象 = QIcon("路径字符") #路径字符就是你的图标位置  
QPushButton.setIcon(图标对象) #图标对象就是上面返回的对象  
  
btn = QPushButton(w)  
btn.setText("1")  
  
def plus_one(): #槽函数放在信号前面  
    num = int(btn.text())+1 #QPushButton.text获取当前文本内容  
    btn.setText(str(num)) #将十进制转换成字符串  
  
btn.pressed.connect(plus_one) #信号与槽链接  
  
icon = QIcon("tubiao.jpg") #在当前目录下找图标  
btn.setIcon(icon) #将图标放进按钮  
  
w.show()  
sys.exit(app.exec_())
```



图标添加成功

```
QPushButton.setIconSize(QSize(长, 宽)) #设置图标大小
icon = QIcon("tubiao.jpg") #在当前目录下找图标
btn.setIcon(icon) #将图标放进按钮
btn.resize(100,100) #一定要设置按钮大小
btn.setIconSize(QSize(100,100)) #再设置图标大小100x100
```



最好用背景透明图片

```
QPushButton.setAutoRepeat(True) #按住按钮不放，就可以一直重复发按钮信号
btn = QPushButton(w)
btn.setText("1")

def plus_one(): #槽函数放在信号前面
    num = int(btn.text())+1 #QPushButton.text获取当前文本内容
    btn.setText(str(num)) #将十进制转换成字符串
    print("按住按钮不放")

btn.pressed.connect(plus_one) #信号与槽链接

btn.resize(100,100) #一定要设置按钮大小
btn.setAutoRepeat(True) #让按钮拥有按住不放，不停发送信号功能
w.show()
sys.exit(app.exec_())
```

按住按钮不放  
按住按钮不放  
按住按钮不放  
按住按钮不放  
按住按钮不放  
按住按钮不放



这就是按住按钮发送了 7 次信号

## 让控件处于自动按下状态，而不是鼠标去点击后才按下

控件.setDown(参数) #参数为 True，控件自动处于鼠标按住状态，记住是按住，还没有放开  
其实鼠标按下功能也是 QAbstractButton 具备的，只是下面三个控件继承过来了。

```
btn = QPushButton(w)
btn.setText("这是按钮")
btn.move(100,100)

checkbox = QCheckBox(w)
checkbox.setText("这是勾选按钮")
checkbox.move(100,150)

radio = QRadioButton(w)
radio.setText("这是单选按钮")
radio.move(100,200)

btn.setDown(True) #按钮继承至QAbstractButton类，所以拥有setDown功能
checkbox.setDown(True) #勾选按钮也是继承至QAbstractButton类，所以拥有setDown功能
radio.setDown(True) #单选按钮也是继承至QAbstractButton类，所以拥有setDown功能
```



鼠标按住控件，就是这个样子。

变量 = 对象.isCheckable() #返回对象是否可以被选中能被选中返回 True 不能选中返回 False

```
btn = QPushButton(w)
btn.setText("这是按钮")
btn.move(100,100)

checkbox = QCheckBox(w)
checkbox.setText("这是勾选按钮")
checkbox.move(100,150)

radio = QRadioButton(w)
radio.setText("这是单选按钮")
radio.move(100,200)

print(btn.isCheckable()) #查询控件是否可以被选中
print(checkbox.isCheckable())
print(radio.isCheckable())
```

你看，只有按钮没有选中功能，单选和勾选都有选中功能

False  
True  
True

对象.setCheckable(True) #使控件具备可以被选中功能

```
btn = QPushButton(w)
btn.setText("这是按钮")
btn.move(100,100)
btn.setCheckable(True) #让按钮具备可以被选中功能

checkbox = QCheckBox(w)
checkbox.setText("这是勾选按钮")
checkbox.move(100,150)

radio = QRadioButton(w)
radio.setText("这是单选按钮")
radio.move(100,200)

print(btn.isCheckable()) #查询控件是否可以被选中
print(checkbox.isCheckable())
print(radio.isCheckable())
```

你看，按钮也具备选中功能了

True  
True  
True

对象.toggle() #控件状态切换，比如已经勾选状态变为非勾选，非勾选变为勾选，按下变为未按下，未按下变为按下，实现类似全选，反选功能

```
w = QWidget()

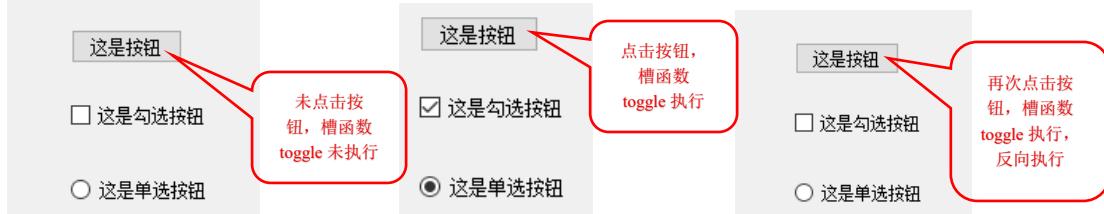
btn = QPushButton(w)
btn.setText("这是按钮")
btn.move(100,100)

checkbox = QCheckBox(w)
checkbox.setText("这是勾选按钮")
checkbox.move(100,150)

radio = QRadioButton(w)
radio.setText("这是单选按钮")
radio.move(100,200)

def cao():
    checkbox.toggle() #切换勾选翻转
    radio.toggle() #切换选择变反选

btn.pressed.connect(cao) # 按钮按下后去槽函数切换空间状态
```



如果控件变为不可选了， toggle 也能让其控件反选

```
checkbox.setEnabled(False) #将勾选按钮变成不可选  
radio.setEnabled(False) #将单选按钮变成不可选
```

```
def cao():  
    checkbox.toggle() #切换勾选翻转  
    radio.toggle() #切换单选翻转
```

```
btn.pressed.connect(cao) # 按钮按下后去槽函数切换空间状态
```



所以 `toggle` 并不受控件不可选限制

**QAbstractButton 排他性，只要继承这个类的都有排他性，比如按钮**  
多个按钮选择实现

```
btn1 = QPushButton(w)  
btn1.setText("按钮1111")  
btn1.move(100,100)  
  
btn2 = QPushButton(w)  
btn2.setText("按钮2222")  
btn2.move(200,100)  
  
btn3 = QPushButton(w)  
btn3.setText("按钮3333")  
btn3.move(300,100)  
  
print(btn1.isCheckable())#检测按钮是否具有可选中功能  
print(btn2.isCheckable())  
print(btn3.isCheckable())
```

False  
False  
False 按钮处于不可选择状态

```
btn1 = QPushButton(w)  
btn1.setText("按钮1111")  
btn1.move(100,100)  
btn1.setCheckable(True)#设置按钮具备可选性  
  
btn2 = QPushButton(w)  
btn2.setText("按钮2222")  
btn2.move(200,100)  
btn2.setCheckable(True)  
  
btn3 = QPushButton(w)  
btn3.setText("按钮3333")  
btn3.move(300,100)  
btn3.setCheckable(True)  
  
print(btn1.isCheckable())#返回True按钮具备可选性  
print(btn2.isCheckable())  
print(btn3.isCheckable())
```

True  
True  
True

按钮具备可选性后，就可以选择多个按钮，类似前端的偏好设置功能

用排他性功能让多个按钮同时只能选择一个

对象.setAutoExclusive(True) #启动控件排他性

```
btn1 = QPushButton(w)
btn1.setText("按钮1111")
btn1.move(100,100)
btn1.setCheckable(True)#设置按钮具备可选性
btn1.setAutoExclusive(True) #设置按钮具备排他性

btn2 = QPushButton(w)
btn2.setText("按钮2222")
btn2.move(200,100)
btn2.setCheckable(True)#设置按钮具备可选性
btn2.setAutoExclusive(True) #设置按钮具备排他性

btn3 = QPushButton(w)
btn3.setText("按钮3333")
btn3.move(300,100)
btn3.setCheckable(True)#设置按钮具备可选性
btn3.setAutoExclusive(True) #设置按钮具备排他性
```



排他性就是偏好设置按钮只能选择一个



排他性就是偏好设置按钮只能选择一个



排他性就是偏好设置按钮只能选择一个

## 按钮被按住按下，按钮被释放，按钮被点击，按钮选择状态改变实现

```
对象 . pressed.connect(槽函数) #按钮被按下执行的槽函数  
对象 . released.connect(槽函数) #按钮被释放执行的槽函数  
对象 . clicked.connect (槽函数) #按钮被点击执行的槽函数，其实按下释放就会触发该函数  
对象 . toggled.connect (槽函数) #对对象状态切换执行槽函数，一般在单选和复选框用
```

```
btn1 = QPushButton(w)  
btn1.setText("按钮1111")  
btn1.move(100,100)  
  
def pressdcao():  
    print("按钮被按下")  
def releasedcao():  
    print("按钮被释放")  
def clickedcao():  
    print("按钮被点击")  
def toggledcao():  
    print("按钮选中状态改变")  
  
btn1.pressed.connect(pressdcao) #按钮被按下槽函数  
btn1.released.connect(releasedcao) #按钮被释放  
btn1.clicked.connect(clickedcao) #按钮被点击  
btn1.toggled.connect(toggledcao) #按钮选中状态发生改变
```

按钮被按下  
按钮被释放  
按钮被点击

设置按钮某部分点击才有效，其余部分点击无效

```
def hitButton(self , point) #函数名是固定的，用自定义类创建按钮对象，按钮按下会回调该函数  
w = QWidget()  
  
class Btn(QPushButton):  
    def hitButton(self,point): #hitButton函数名是固定的，不能随便取  
        print(point) #获取鼠标点击按钮的坐标，坐标0,0是从按钮本身左上角开始，而不是窗口左上角  
        return True #返回True 才会执行该按钮连接的槽函数  
  
btn1 = Btn(w)  
btn1.setText("按钮")  
btn1.move(100,100)  
btn1.pressed.connect(lambda : print("按钮被按下"))
```

PyQt5.QtCore.QPoint(26, 15)  
按钮被按下

```
w = QWidget()  
  
class Btn(QPushButton):  
    def hitButton(self,point): #hitButton函数名是固定的，不能随便取  
        print(point) #获取鼠标点击按钮的坐标，坐标0,0是从按钮本身左上角开始，而不是窗口左上角  
        return False #返回False 只能获取坐标，无法触发槽函数  
  
btn1 = Btn(w)  
btn1.setText("按钮")  
btn1.move(100,100)  
btn1.pressed.connect(lambda : print("按钮被按下"))
```

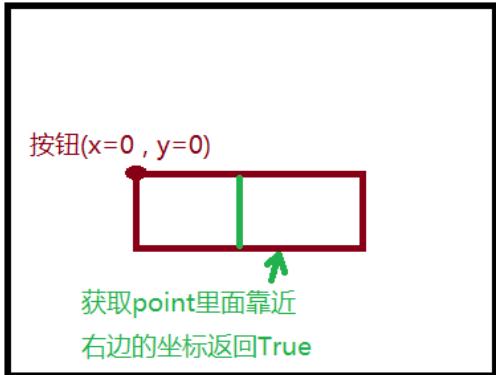
返回 False 不会执行槽函数



PyQt5.QtCore.QPoint(23, 11)  
PyQt5.QtCore.QPoint(27, 14)

我们使用返回值 True/False 来设计按钮指定部分触发槽函数功能

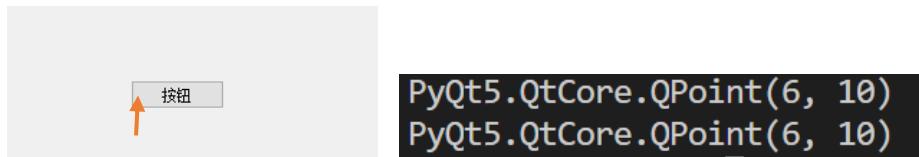
窗口(x, y)



```
w = QWidget()

class Btn(QPushButton):
    def hitButton(self, point): #hitButton函数名是固定的，不能随便取
        print(point) #获取鼠标点击按钮的坐标，坐标0,0是从按钮本身左上角开始，而不是窗口左上角
        if point.x() > (self.width()/2): #如果点击按钮位置 > 整个按钮宽度的一半
            return True #点击按钮右侧返回真触发槽函数
        return False #返回False 只能获取坐标，无法触发槽函数

btn1 = Btn(w)
btn1.setText("按钮")
btn1.move(100,100)
btn1.pressed.connect(lambda : print("按钮被按下"))
```



点击按钮左半部分，无法触发槽函数



点击按钮右半部分触发槽函数

## 菜单控件 QMenu

对象 = QMenu() #创建菜单，只是创建菜单对象，不能显示，需要嵌入其它对象显示  
QPushButton::setMenu(菜单对象) #将菜单对象嵌入进按钮

```
w = QWidget()
btn = QPushButton(w) #创建按钮
btn.setText("按钮")

menu = QMenu()#创建主菜单

btn.setMenu(menu) #因为这是QWidget类，所以顶层没有嵌套菜单的对象，只有用按钮对象来嵌入菜单

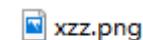
w.show()
sys.exit(app.exec_())
```



按钮 ▾

按钮嵌入了菜单功能，但是无法下拉，因为没有子菜单

返回子菜单项对象 = QAction("填写子菜单名", 填入嵌入的主菜单对象) #创建子菜单项  
QAction::setIcon(填入 QIcon 函数) #在子菜单项加入图标  
QIcon("填入 png 图片路径") #将图片转换成 QIcon 可以传入的参数  
QMenu::addAction(子菜单项对象) #向主菜单加入子菜单对象



test.py 在当前目录下有一个 xzz.png 图片做为图标。

```
app = QApplication(sys.argv)

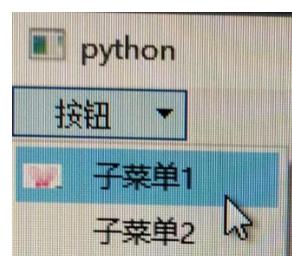
w = QWidget()

btn = QPushButton(w) #创建按钮
btn.setText("按钮")

menu = QMenu()#创建主菜单
action1 = QAction("子菜单1",menu) #子菜单创建
action1.setIcon(QIcon("xzz.png")) #加入png图标，没写路径证明图片在当前目录下也就是和我test.py同级目录
menu.addAction(action1)#向主菜单加入子菜单

action2 = QAction("子菜单2",menu) #子菜单创建
menu.addAction(action2)#向主菜单加入子菜单

btn.setMenu(menu) #因为这是QWidget类，所以顶层没有嵌套菜单的对象，只有用按钮对象来嵌入菜单
```



两个子菜单项(QAction)创建成功。

问题 qt.gui.icc: Failed ICC signature test 这是加入 QIcon 导致的，可以取消菜单图标来解决

如何监听某一个菜单项被点击？

QAction::triggered.connect(槽函数) #子菜单项 QAction 被点击后触发槽函数

```
w = QWidget()

btn = QPushButton(w) #创建按钮
btn.setText("按钮")

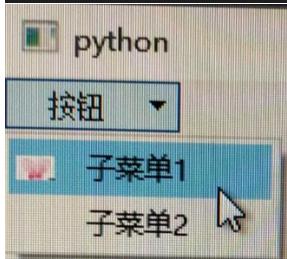
def cao():
    print("子菜单1被按下") #加入槽函数

menu = QMenu() #创建主菜单
action1 = QAction("子菜单1",menu) #子菜单创建
action1.setIcon(QIcon("xzz.png")) #加入png图标, 没写路径证明图片在当前目录下也就是和我test
action1.triggered.connect(cao) #加入信号槽, 子菜单项被点击, 触发信号槽链接
menu.addAction(action1) #向主菜单加入子菜单

action2 = QAction("子菜单2",menu) #子菜单创建
menu.addAction(action2) #向主菜单加入子菜单

btn.setMenu(menu) #因为这是QWidget类, 所以顶层没有嵌套菜单的对象, 只有用按钮对象来嵌入菜单

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



子菜单1被按下  
子菜单1被按下

点击子菜单1就会打印槽函数

QAction(QIcon(png 图片路径), "菜单名", 要嵌入的主菜单对象) #第 2 种菜单图标简化方法

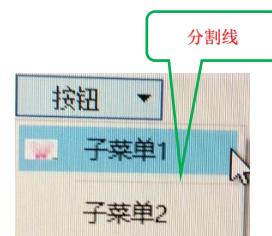
```
menu = QMenu() #创建主菜单
action1 = QAction(QIcon("xzz.png"), "子菜单1",menu) #子菜单创建
action1.triggered.connect(cao)
menu.addAction(action1) #向主菜单加入子菜单
```



效果一样的

QAction::addSeparator() #在多个 QAction 子菜单项之间加入分割线

```
menu = QMenu() #创建主菜单
action1 = QAction(QIcon("xzz.png"), "子菜单1",menu) #子菜单创建
menu.addAction(action1) #向主菜单加入子菜单
menu.addSeparator() #在action1子项和action2子项之间加入分割线
action2 = QAction("子菜单2",menu) #子菜单创建
menu.addAction(action2) #向主菜单加入子菜单
```



在主菜单下除了创建下拉子菜单，还可以创建子子菜单

```
menu = QMenu() # 创建主菜单
menu2 = QMenu(menu) # 创建子菜单的子菜单(子子菜单)
menu2.setTitle("子子菜单") # 给子子菜单取名

action1 = QAction(QIcon("xzz.png"), "子菜单1", menu) # 子菜单创建
menu.addAction(action1) # 向主菜单加入子菜单
menu.addSeparator() # 在 action1 子项和 action2 子项之间加入分割线
action2 = QAction("子菜单2", menu) # 子菜单创建
menu.addAction(action2) # 向主菜单加入子菜单
menu.addSeparator()
menu.addMenu(menu2) # 将子子菜单加入主菜单
```

btn.setMenu(menu) # 因为这是 QWidget 类，所以顶层没有嵌套菜单的对象，

python



这就是子子菜单，菜单下一级还可以创建 QAction 子菜单项或者菜单

```
menu = QMenu() # 创建主菜单
menu2 = QMenu(menu) # 创建子菜单的子菜单(子子菜单)
menu2.setTitle("子子菜单") # 给子子菜单取名

action1 = QAction(QIcon("xzz.png"), "子菜单1", menu) # 子菜单创建
menu.addAction(action1) # 向主菜单加入子菜单
menu.addSeparator() # 在 action1 子项和 action2 子项之间加入分割线
action2 = QAction("子菜单2", menu) # 子菜单创建
menu.addAction(action2) # 向主菜单加入子菜单
menu.addSeparator()

action3 = QAction("子菜单3", menu) # 创建子菜单项
menu2.addAction(action3) # 在子子菜单中加入子菜单项
```

menu.addMenu(menu2) # 将子子菜单加入主菜单

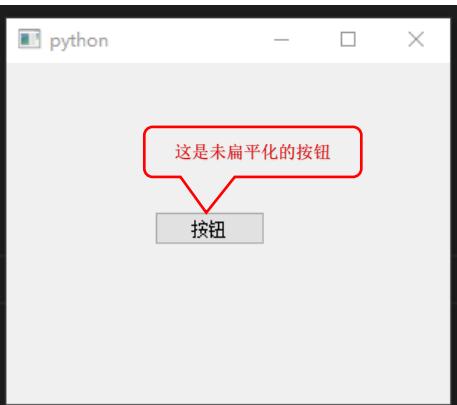


这就是 3 级菜单

在子菜单下面加入子菜单项

## 扁平化设置(按钮为例)

```
w = QWidget()  
  
btn = QPushButton(w) #创建按钮  
btn.setText("按钮")  
btn.move(100,100)  
  
w.show()  
sys.exit(app.exec_())
```



QPushButton::setFlat(True) #将按钮进行扁平化

```
w = QWidget()  
  
btn = QPushButton(w) #创建按钮  
btn.setText("按钮")  
btn.move(100,100)  
btn.setFlat(True) #将按钮进行扁平化  
w.show()  
sys.exit(app.exec_())
```



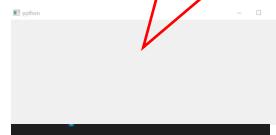
感觉这种扁平好单调哦，其实扁平是需要用 QSS 来设置样式才好看。

## 右键菜单实现(就是鼠标在窗口右键显示菜单)

contextMenuEvent(self, QContextMenuEvent) #鼠标右键空白窗口，触发菜单事件，这个函数是固定模式，不要随便写函数名

```
from PyQt5.Qt import *  
import sys  
  
app = QApplication(sys.argv)  
  
class window(QWidget): #定义一个带右击功能的窗口类  
    def contextMenuEvent(self,QContextMenuEvent):  
        print("展示菜单")  
  
w = window() #创建带右击功能的窗口  
  
w.show()  
sys.exit(app.exec_())
```

鼠标对空白窗口点击右键  
contextMenuEvent 回调函数执行



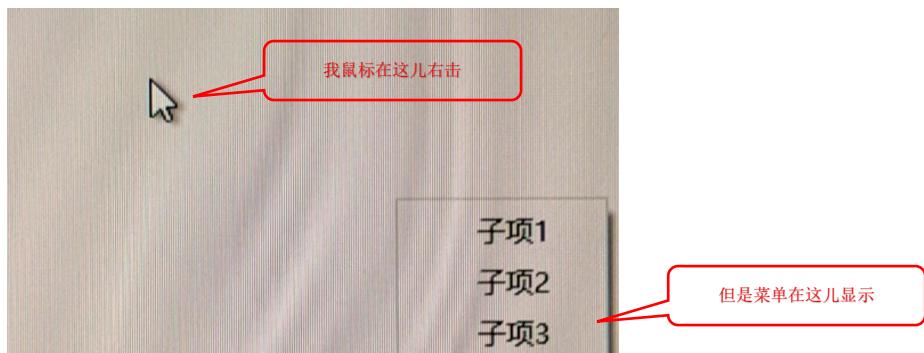
我们就在右键触发的回调函数中创建菜单，显示菜单

```
app = QApplication(sys.argv)

class window(QWidget): #定义一个带右击功能的窗口类
    def contextMenuEvent(self,QContextMenuEvent):
        print("展示菜单")
        menu = QMenu()
        menu.setTitle("主菜单")
        action1 = QAction("子项1",menu)
        menu.addAction(action1)
        menu.addSeparator()
        action2 = QAction("子项2",menu)
        menu.addAction(action2)
        menu.addSeparator()
        action3 = QAction("子项3",menu)
        menu.addAction(action3)
        menu.addAction(action1)#向主菜单加入子项
        menu.addAction(action2)#向主菜单加入子项
        menu.addAction(action3)#向主菜单加入子项

        menu.exec_(QPoint(1500,500)) #传入菜单显示得开始坐标x=1500, y=500

w = window() #创建带右击功能的窗口
w.show()
sys.exit(app.exec_())
```



菜单没有在鼠标指定位置显示？这是因为 menu.exec\_(...)设置的是固定坐标，下面进行修改  
对象.globalPos() #获取鼠标在窗口的当前坐标

```
class window(QWidget): #定义一个带右击功能的窗口类
    def contextMenuEvent(self,QContextMenuEvent):
        print("展示菜单")
        menu = QMenu()
        menu.setTitle("主菜单")
        action1 = QAction("子项1",menu)
        menu.addAction(action1)
        menu.addSeparator()
        action2 = QAction("子项2",menu)
        menu.addAction(action2)
        menu.addSeparator()
        action3 = QAction("子项3",menu)
        menu.addAction(action3)
        menu.addAction(action1)#向主菜单加入子项
        menu.addAction(action2)#向主菜单加入子项
        menu.addAction(action3)#向主菜单加入子项

        menu.exec_(QContextMenuEvent.globalPos()) #获取当前鼠标右击坐标显示菜单
```



鼠标右击的位置弹出菜单

## 列表按钮(自带扁平功能)

```
QCommandLinkButton("传入大标题字符", "传入小标题字符", 嵌入的对象) #创建列表按钮
app = QApplication(sys.argv)

w = QWidget()
btn = QCommandLinkButton("标题", "描述", w) #创建列表按钮, 传入的参数为默认显示

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



默认扁平显示, 带个箭头, 鼠标点击才有效



这就是鼠标点击的效果

大标题和小标题都可以修改

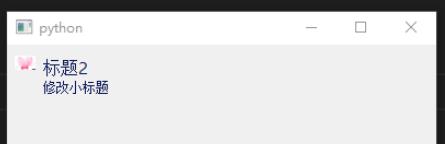
```
QCommandLinkButton::setText("传入字符") #修改大标题
w = QWidget()
btn = QCommandLinkButton("标题", "描述", w) #创建列表按钮, 传入的参数为默认显示
btn.setText("标题2") #修改大标题
w.show()
sys.exit(app.exec_())
```



大标题被修改

```
QCommandLinkButton::setDescription("传入字符") #修改小标题
QCommandLinkButton::setIcon("传入图标") #列表按钮传入图标
```

```
w = QWidget()
btn = QCommandLinkButton("标题", "描述", w) #创建列表按钮, 传入的参数为默认显示
btn.setText("标题2") #修改大标题
btn.setDescription("修改小标题") #修改小标题
btn.setIcon(QIcon("xzz.png")) #可以加入图标
w.show()
```



## 工具按钮, 只显示图标的按钮

对象 = QToolButton(要嵌入的对象)

```
app = QApplication(sys.argv)

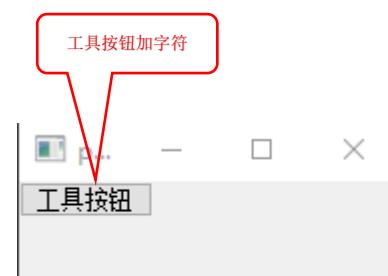
w = QWidget()
Tb = QToolButton(w) #创建工具按钮

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



```
QToolButton::setText("字符") #给工具按钮增加名字
```

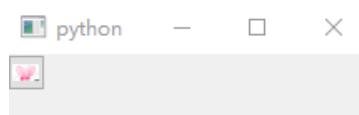
```
app = QApplication(sys.argv)
w = QWidget()
Tb = QToolButton(w) #创建工具按钮
Tb.setText("工具按钮") #按钮显示名字
w.show()
sys.exit(app.exec_())
```



感觉和普通按钮没什么区别啊？

```
QToolButton::setIcon(QIcon("加入图标路径")) #给工具按钮加图标
```

```
Tb = QToolButton(w) #创建工具按钮  
Tb.setText("工具按钮") #按钮显示名字  
Tb.setIcon(QIcon("xzz.png")) #图标按钮加入图片
```

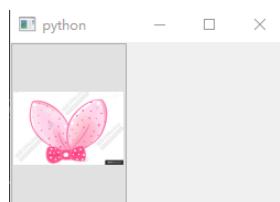


按钮图标有了，把文字覆盖了。

所以工具按钮最适合用图片显示，不要使用文字。

```
QToolButton::setIcon(QSize(x, y)) #设置工具按钮大小
```

```
Tb = QToolButton(w) #创建工具按钮  
Tb.setText("工具按钮") #按钮显示名字  
Tb.setIcon(QIcon("xzz.png")) #图标按钮加入图片  
Tb.setIconSize(QSize(100,150)) #设置图标大小
```



所以工具按钮一般用在窗口上方显示图标。

```
QToolButton::setToolTip("输入要提示的字符") #鼠标移动到按钮功能提示
```

```
Tb = QToolButton(w) #创建工具按钮  
Tb.setText("工具按钮") #按钮显示名字  
Tb.setIcon(QIcon("xzz.png")) #图标按钮加入图片  
Tb.setIconSize(QSize(100,150)) #设置图标大小  
Tb.setToolTip("按钮功能提示") #鼠标移动到按钮出现提示  
w.show()
```



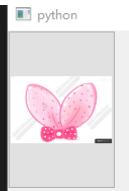
提示字符

## 设置按钮样式和风格

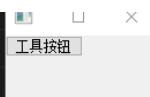
QToolButton::setToolButtonStyle(参数)

参数:	Qt.ToolButtonIconOnly	按钮仅显示图标
	Qt.ToolButtonTextOnly	按钮仅显示文字
	Qt.ToolButtonTextBesideIcon	按钮图标文字都显示
	Qt.ToolButtonTextUnderIcon	按钮文本显示在图标下方

```
Tb.setToolButtonStyle(Qt.ToolButtonIconOnly) #按钮仅显示图标
```



```
Tb.setToolButtonStyle(Qt.ToolButtonTextOnly) #按钮仅显示文字
```



```
Tb.setToolButtonStyle(Qt.ToolButtonTextBesideIcon) #图标文字都显示
```



```
Tb.setToolButtonStyle(Qt.ToolButtonTextUnderIcon) #文本显示在图标下方
```



## 按钮方向图标

QToolButton::setArrowType(参数)

参数:

Qt.UpArrow 按钮向上图标

```
Tb.setArrowType(Qt.UpArrow) #按钮向上图标
```



Qt.DownArrow 按钮向下图标

```
Tb.setArrowType(Qt.DownArrow) #按钮向下图标
```



Qt.LeftArrow 按钮向左图标

```
Tb.setArrowType(Qt.LeftArrow) #按钮向左图标
```



Qt.RightArrow 按钮向右图标

```
Tb.setArrowType(Qt.RightArrow) #按钮向右图标
```



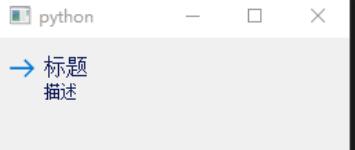
## 命令连接按钮，扁平化的箭头按钮

```
对象 = QCommandLinkButton(输入标题字符串,描述内容字符串,父对象) #创建命令链接按钮
```

```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

btn = QCommandLinkButton("标题","描述",w) #创建命令链接按钮
w.show()
sys.exit(app.exec_())
```



按钮按下有阴影

```
QCommandLinkButton::setText(标题字符串)#修改初始化的标题  
QCommandLinkButton::setDescription(描述字符串)#修改初始化的描述
```

```
btn = QCommandLinkButton("标题","描述",w) #创建命令链接按钮
btn.setText("标题2222") #修改标题
btn.setDescription("描述222")#修改描述
```



你看标题和描述被修改

命令连接按钮就这么个东西，没什么特别的

## 单选按钮使用

对象 = QRadioButton("按钮名称", 嵌入的父对象) #单选按钮创建

```
w = QWidget()
```

```
rb1 = QRadioButton("男",w)
```

```
rb1.move(100,100) #移动位置, 防止多个选项被重叠
```

```
rb2 = QRadioButton("女",w)
```

```
rb2.move(150,100)
```

```
rb3 = QRadioButton("老人",w)
```

```
rb3.move(200,100)
```

```
w.show()
```



这就是单选按钮。

QRadioButton::setIcon(QIcon(路径)) #给单选按钮加图标

```
rb1 = QRadioButton("男",w)
```

```
rb1.move(100,100) #移动位置, 防止多个选项被重叠
```

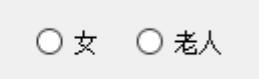
```
rb1.setIcon(QIcon("xzz.png")) #给单选按钮加入图标
```

```
rb2 = QRadioButton("女",w)
```

```
rb2.move(200,100)
```

```
rb3 = QRadioButton("老人",w)
```

```
rb3.move(250,100)
```



程序启动后单选按钮默认选择某一项

QRadioButton::setChecked(True) #设置某个对象启动后默认选择

```
rb1 = QRadioButton("男",w)
```

```
rb1.move(100,100) #移动位置, 防止多个选项被重叠
```

```
rb1.setIcon(QIcon("xzz.png")) #给单选按钮加入图标
```

```
rb2 = QRadioButton("女",w)
```

```
rb2.setChecked(True) #对象rb2设置Checked为True, 程序启动默认选rb2
```

```
rb2.move(200,100)
```

```
rb3 = QRadioButton("老人",w)
```

```
rb3.move(250,100)
```



单选按钮被选中后触发信号，知道哪一个单选被选了

QRadioButton::isChecked() #检查对象是否被选上，被选上返回 True，未被选上返回 False

```
rb1 = QRadioButton("男",w)
rb1.move(100,100) #移动位置，防止多个选项被重叠
rb1.setIcon(QIcon("xzz.png")) #给单选按钮加入图标
rb2 = QRadioButton("女",w)
rb2.setChecked(True) #对对象rb2设置Checked为True，程序启动默认选rb2
rb2.move(200,100)
rb3 = QRadioButton("老人",w)
rb3.move(250,100)

def cao():
    if rb2.isChecked() == True: #在槽函数里面判断对象是否被选上
        print("True")
    else:
        print("False")

rb2.toggled.connect(cao) #rb2对象就是‘女’被选上执行槽函数，取消选择也会执行槽函数
w.show()
```



qt.gui.icc: Failed ICC signature test  
False

系统开机不执行槽函数



False 取消选择 ‘女’ 执行槽函数



True 再次选上 ‘女’ 执行槽函数

如果在已有的单选按钮界面加另外一组单选按钮，而且另外一组的单选按钮不影响已有的单选按钮，那么需要定义两个窗口对象，也就是必须两组的单选按钮对象不同。

```
w = QWidget()

w1 = QWidget(w) #单选按钮第1组嵌入的对象
w1.move(100,100)
w1.resize(500,500)

w2 = QWidget(w) #单选按钮第2组嵌入的对象
w2.move(600,100)
w2.resize(500,500)
```

```

rb1 = QRadioButton("男",w1) #第1组单选按钮嵌入w1对象
rb1.move(100,100)
rb2 = QRadioButton("女",w1)
rb2.move(200,100)
rb3 = QRadioButton("老人",w1)
rb3.move(250,100)

rb4 = QRadioButton("男",w2)#第2组单选按钮嵌入w1对象
rb4.move(100,100)
rb5 = QRadioButton("女",w2)
rb5.move(200,100)
rb6 = QRadioButton("老人",w2)
rb6.move(250,100)

```

男  女  老人

男  女  老人

这样两组单选按钮相互不影响

这种做两组可选按钮需要建立两个对象太麻烦了，下面用可选按钮组来解决  
对象 = QPushButton(传入可选按钮处于的界面对象)

#用可选按钮组来解决同一个对象不同可选按钮组相互影响的问题

QPushButton:: addButton(传入可选按钮对象) #向按钮组增加可选按钮

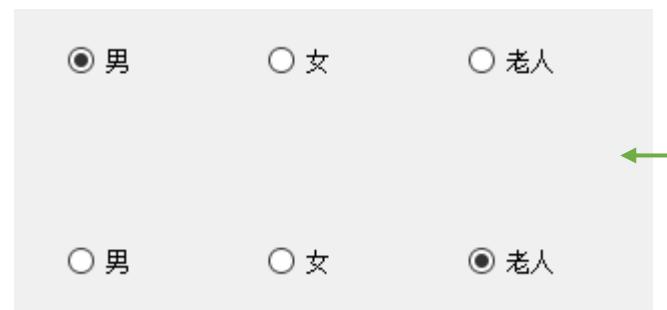
```

w = QWidget()

rb1 = QRadioButton("男",w)
rb1.move(100,100)
rb2 = QRadioButton("女",w)
rb2.move(200,100)
rb3 = QRadioButton("老人",w)
rb3.move(300,100)

rb4 = QRadioButton("男",w)
rb4.move(100,200)
rb5 = QRadioButton("女",w)
rb5.move(200,200)
rb6 = QRadioButton("老人",w)
rb6.move(300,200)

```



```

Group1 = QPushButton(w) #在同一个窗口(对象)划分出第1组可选按钮
Group1.addButton(rb1) #把rb1 rb2 rb3可选按钮分给第1组
Group1.addButton(rb2)
Group1.addButton(rb3)

```

```

Group2 = QPushButton(w) #在同一个窗口(对象)划分出第2组可选按钮
Group2.addButton(rb4) #把rb4 rb5 rb6可选按钮分给第2组
Group2.addButton(rb5)
Group2.addButton(rb6)

```

可选按钮实验成功，在同一个对象可以达到两组可选按钮相互不影响

设置可选按钮的 ID 值，在可选按钮很多的时候可以用 for 循环来创建按钮，同时也可以用 for 循环检查某个按钮按下

```
QButtonGroup::setId(传入可选按钮对象, 设置 ID 参数) #设置可选按钮的 ID 值  
返回 ID 号 = QButtonGroup::id(传入可选按钮对象) #获取可选按钮 ID 值  
  
Group1 = QButtonGroup(w) #在同一个窗口(对象)划分出第1组可选按钮  
Group1.addButton(rb1) #把rb1 rb2 rb3可选按钮分给第1组  
Group1.addButton(rb2)  
Group1.addButton(rb3)  
  
Group2 = QButtonGroup(w) #在同一个窗口(对象)划分出第2组可选按钮  
Group2.addButton(rb4) #把rb4 rb5 rb6可选按钮分给第2组  
Group2.addButton(rb5)  
Group2.addButton(rb6)  
  
Group1.setId(rb1,1) #给可选按钮设置ID  
Group1.setId(rb2,2)  
Group1.setId(rb3,3)  
print(Group1.id(rb1)) #获取rb1可选按钮的ID值  
print(Group1.id(rb2))
```

1  
2

可选按钮 ID 值打印出来

各组可选按钮被按下对应的事件组被触发

```
QButtonGroup::buttonClicked.connect(传入槽函数) #对应的可选按钮组事件触发函数  
  
Group1 = QButtonGroup(w) #可选按钮组1  
Group1.addButton(rb1)  
Group1.addButton(rb2)  
Group1.addButton(rb3)  
  
Group2 = QButtonGroup(w) #可选按钮组2  
Group2.addButton(rb4)  
Group2.addButton(rb5)  
Group2.addButton(rb6)  
  
def cao1():  
    print("可选按钮组1")  
  
Group1.buttonClicked.connect(cao1) #组1任意可选按钮被点击事件触发  
  
def cao2():  
    print("可选按钮组2")  
  
Group2.buttonClicked.connect(cao2) #组2任意可选按钮被点击事件触发
```

男       女       老人

男       女       老人

可选按钮组1



点击组 1 第 1 个可选按钮，组 1 事件被触发，组 2 事件不会触发，所以组 2 不受影响

男       女       老人

男       女       老人

可选按钮组1

可选按钮组1

点击组 1 第 2 个可选按钮，组 1 事件被触发，组 2 事件不会触发，所以组 2 不受影响

男       女       老人

男       女       老人

同上

可选按钮组1

可选按钮组1

可选按钮组1

男       女       老人

男       女       老人

可选按钮组2

男       女       老人

男       女       老人

可选按钮组2

可选按钮组2

点击组 2 可选按钮不会影响组 1

下面根据可选按钮信号的传参来决定是哪一个可选按钮被选上，这时候上面介绍的可选按钮 ID 设置就发挥作用了。

QButtonGroup::buttonClicked[int].connect(传入槽函数)  
# [int]就是向可选按钮槽函数传入 ID，证明某个可选按钮选择了  
槽函数也需要做如下更改

```
def 函数名(变量)

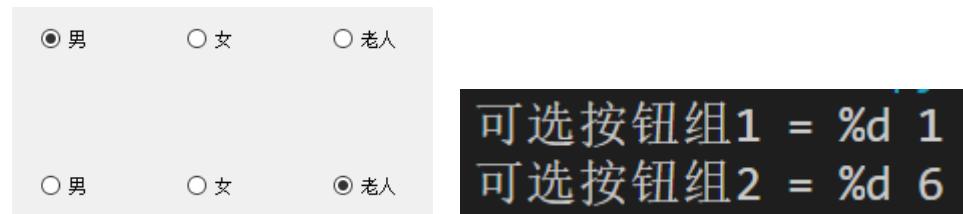
Group1.setId(rb1,1) #给可选按钮设置ID
Group1.setId(rb2,2)
Group1.setId(rb3,3)

Group2.setId(rb4,4) #给可选按钮设置ID
Group2.setId(rb5,5)
Group2.setId(rb6,6)
def cao1(val):
    print("可选按钮组1 = %d",val) #val获取被选择的可选按钮ID值

Group1.buttonClicked[int].connect(cao1) #组1任意可选按钮被点击事件触发传入可选按钮ID值

def cao2(val):
    print("可选按钮组2 = %d",val) #val获取被选择的可选按钮ID值

Group2.buttonClicked[int].connect(cao2) #组2任意可选按钮被点击事件触发传入可选按钮ID值
```



根据槽函数的 ID 值确定哪一个可选按钮被选中。

## 多选按钮实现，就是多个打钩

对象 = QCheckBox(“传入显示字符”，嵌入的父对象) #创建可选按钮

```
app = QApplication(sys.argv)

w = QWidget()

check = QCheckBox("标题",w) #创建勾选按钮

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



勾选按钮创建完成

```
QCheckBox::setIcon(QIcon(图片路径)) #设置可选按钮图标
```

```
w = QWidget()
```

```
check = QCheckBox("标题",w) #创建勾选按钮  
check.setIcon(QIcon("xzz.png"))  
w.show()
```

pyt... —

标题

图标设置完成。

```
QCheckBox::stateChanged.connect(槽函数) #勾选按钮选中后触发事件槽，取消勾选也会触发  
w = QWidget()
```

```
check = QCheckBox("标题",w) #创建勾选按钮  
check.setIcon(QIcon("xzz.png"))
```

```
def cao():  
    print("勾选按钮被触发")
```

```
check.stateChanged.connect(cao) #勾选按钮事件触发槽函数
```

```
w.show()
```

勾选按钮被触发  标题

勾选会触发槽函数

勾选按钮被触发

python

勾选按钮被触发  标题

取消勾选也会触发槽函数

如何知道哪一个勾选按钮被选中呢？

返回值 = QCheckBox::isChecked()

#检查指定对象是否被勾选，被勾选返回 True，未被勾选返回 False

```
w = QWidget()
```

```
check = QCheckBox("标题",w) #创建勾选按钮
```

```
def cao():  
    if check.isChecked() == True :  
        print("勾选按钮被勾选")  
    else:
```

```
        print("勾选按钮被释放")
```

```
check.stateChanged.connect(cao) #勾选按钮事件触发槽函数
```

勾选按钮被勾选  标题

勾选按钮被勾选  python

勾选按钮被释放  标题

## 输入文本框

```
对象 = QLineEdit(嵌入的对象) #创建文本框
```

```
w = QWidget()
```

```
edit = QLineEdit(w) #创建编辑框
```

```
w.show()
```

这就是编辑框，但是无法换行。



```
对象 = QLineEdit("传入字符串", 嵌入的对象)
```

```
#创建文本框时加入字符串, 那么文本框默认显示字符串
```

```
w = QWidget()
```

```
edit = QLineEdit("标题框",w) #创建编辑框, 默认显示"标题框"
```

```
w.show()
```



```
QLineEdit::setText("传入字符串") #设置文本内容, (可以将获取的数据转成字符串显示到文本框)
```

```
edit = QLineEdit("标题框",w) #创建编辑框, 默认显示"标题框"
```

```
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"
```

## 获取文本框内容

```
变量 = QLineEdit::text() #获取文本框内容返回给变量
```

```
w = QWidget()
```

```
edit = QLineEdit("标题框",w) #创建编辑框, 默认显示"标题框"  
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"
```

```
Button = QPushButton(w) #创建按钮, 触发文本框获取事件
```

```
Button.move(0,30)
```

```
def cao():
```

```
    txt = edit.text() #获取文本内容  
    print(txt)
```

```
Button.clicked.connect(cao) #点击按钮系统获取文本内容
```

```
w.show()
```



将 A 文本内容复制给 B 文本，使用 setText 或者 insert 插入(注意 insert 有区别)

QLineEdit::setText(传入文本内容) #指定文本框对象，写内容，也会覆盖以前的文本内容

```
edit = QLineEdit("A文本框", w) #创建编辑框，默认显示"标题框"  
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"  
edit.move(0,0)  
edit2 = QLineEdit("B文本框", w) #创建编辑框，默认显示"标题框"  
edit2.setText("得到内容") #默认显示得"标题框"被改成了"修改内容"  
edit2.move(200,0)
```

Button = QPushButton(w) #创建按钮，触发文本框获取事件

Button.move(0,30)

```
def cao():  
    txt = edit.text() #获取文本内容  
    edit2.setText(txt) #将获取的内容写入文本框2  
    print(txt)
```

txt 得到 A 文本框内容，这句就是  
将 A 文本内容赋值给 B 文本框

Button.clicked.connect(cao) #点击按钮系统获取文本内容



QLineEdit::insert(传入文本内容) #插入文本，不会覆盖文本以前的内容，只会在以前的文本后追加

```
def cao():  
    txt = edit.text() #获取文本内容  
    edit2.insert(txt) #将获取的内容插入到文本框2  
    print(txt)
```



```
QLineEdit::setEchoMode(参数) #设置文本框输出的内容类型
参数: QLineEdit.NoEcho 文本框不输出内容，但是实际是获取到内容的，只是不显示
      QLineEdit.Password 密码不显示
```

```
edit = QLineEdit("A文本框",w) #创建编辑框， 默认显示"标题框"
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"
edit.move(0,0)
edit2 = QLineEdit("B文本框",w) #创建编辑框， 默认显示"标题框"
edit2.setText("得到内容") #默认显示得"标题框"被改成了"修改内容"
edit2.move(200,0)
Button = QPushButton(w) #创建按钮， 触发文本框获取事件
Button.move(0,30)

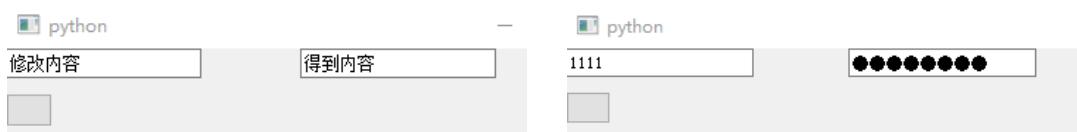
def cao():
    txt = edit.text() #获取文本内容
    edit2.insert(txt) #将获取的内容插入到文本框2
    edit2.setEchoMode(QLineEdit.NoEcho) #编辑框不输出内容
    print(txt)
```



文本框默认值                           文本框 A 输入内容点击按钮，文本框 B 不显示，  
但是文本框 B 是得到内容了的，只能在代码里面打印变量体现。

QLineEdit.Password 密码不显示

```
def cao():
    txt = edit.text() #获取文本内容
    edit2.insert(txt) #将获取的内容插入到文本框2
    edit2.setEchoMode(QLineEdit.Password) #编辑框用*号遮住，用在不显示密码场景
    print(txt)
```



文本框默认值                           B 文本框内容遮住。

QLineEdit::setPlaceholderText("字符串") #传入字符串，文本框背景显示提示字符

```
w = QWidget()

edit = QLineEdit("A文本框",w) #创建编辑框， 默认显示"标题框"
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"
edit.move(0,0)
edit.setPlaceholderText("输入字符串")
```



```
QLineEdit::setAlignment(参数) #设置文本框里面的字符居中，左对齐，右对齐  
参数: Qt.AlignRight 字符右对齐  
Qt.AlignLeft 左对齐  
Qt.AlignHCenter 居中
```

```
edit = QLineEdit("A文本框",w) #创建编辑框，默认显示"标题框"  
edit.setText("修改内容") #默认显示得"标题框"被改成了"修改内容"  
edit.move(0,0)  
edit.setAlignment(Qt.AlignRight)
```

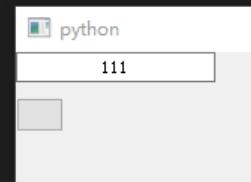


Qt.AlignLeft 左对齐

```
edit = QLineEdit("A文本框",w)  
edit.setText("修改内容")  
edit.move(0,0)  
edit.setAlignment(Qt.AlignLeft) #左对齐
```

Qt.AlignHCenter 居中

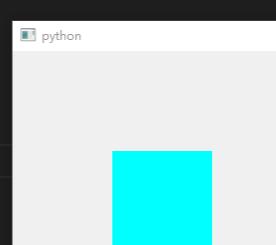
```
edit = QLineEdit("A文本框",w)  
edit.setText("修改内容")  
edit.move(0,0)  
edit.setAlignment(Qt.AlignHCenter) #居中
```



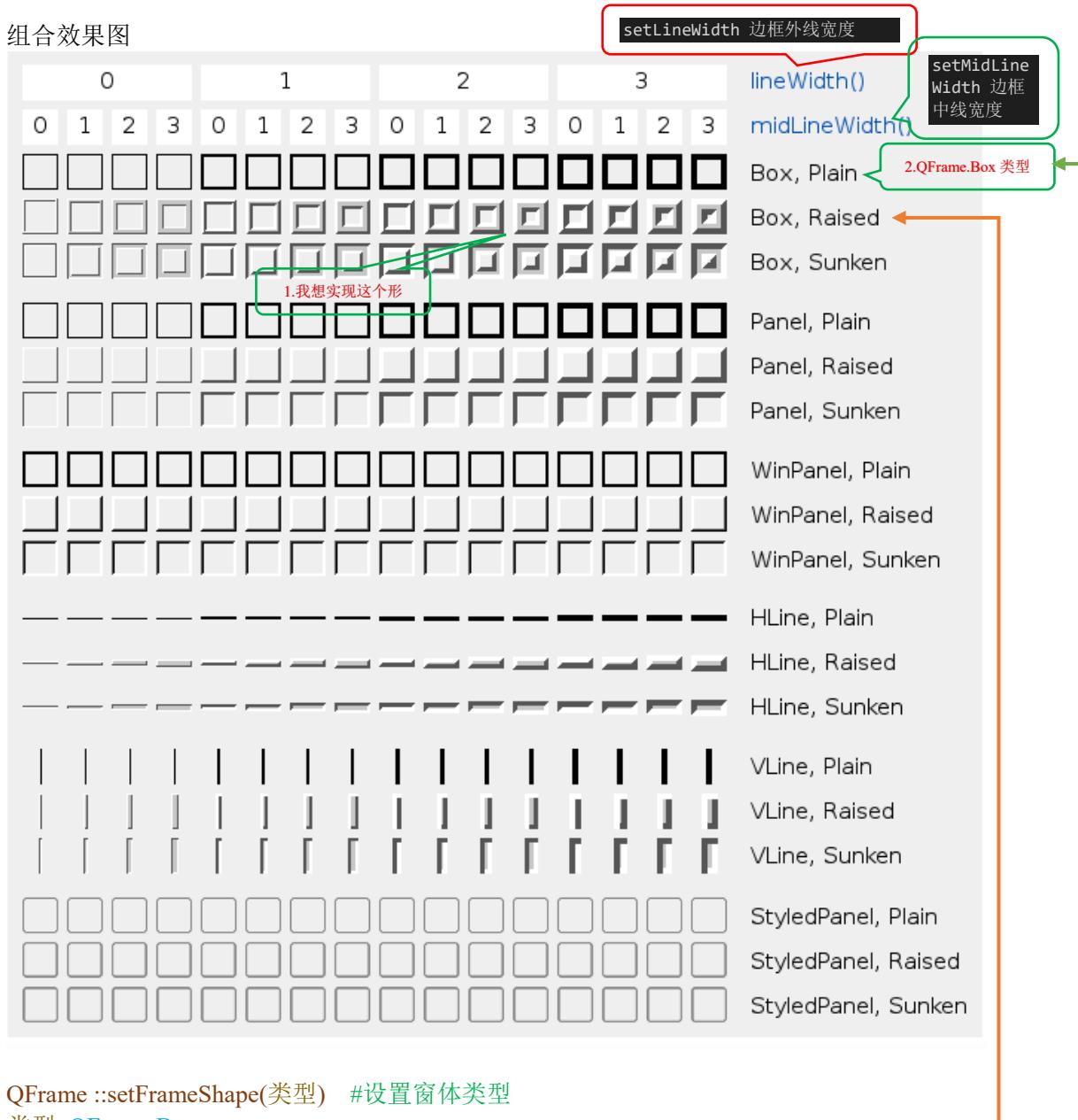
## QFrame 控制矩形空间，凸起，凹下，阴影，线宽，一些矩形控件外观显示样式

生成矩形框对象 = QFrame(父对象)

```
w = QWidget()  
w.resize(500,500)  
  
frame = QFrame(w) #创建矩形框  
frame.resize(100,100) #矩形框大小  
frame.move(100,100)  
frame.setStyleSheet("background-color:cyan") #矩形框背景颜色
```



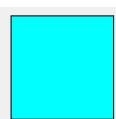
组合效果图



`QFrame :: setFrameShape(类型) #设置窗体类型`

类型: `QFrame.Box`

```
frame = QFrame(w) #创建矩形框
frame.resize(100,100) #矩形框大小
frame.move(100,100)
frame.setStyleSheet("background-color:cyan") #矩形框背景颜色
frame.setFrameShape(QFrame.Box) #设置边框类型
```



`QFrame :: setFrameShadow(参数) #框架阴影设计, 比如边框凸起`

```
frame = QFrame(w) #创建矩形框
frame.resize(100,100) #矩形框大小
frame.move(100,100)
frame.setStyleSheet("background-color:cyan") #矩形框背景颜色
frame.setFrameShape(QFrame.Box) #设置边框类型
frame.setFrameShadow(QFrame.Raised) #将边框凸起
```



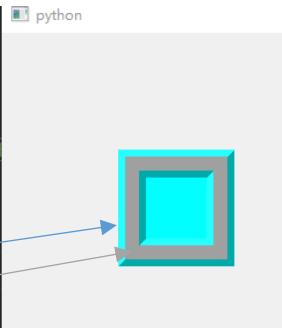
没凸起样式



QFrame.Raised 凸起样式

```
QFrame :: setLineWidth(宽度) #边框外线宽度  
QFrame :: setMidLineWidth(宽度) #边框中线宽度
```

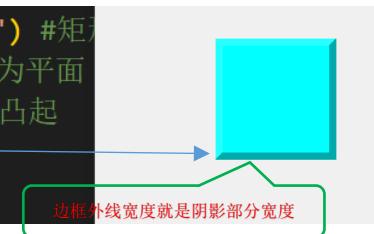
```
frame = QFrame(w) #创建矩形框  
frame.resize(100,100) #矩形框大小  
frame.move(100,100)  
frame.setStyleSheet("background-color:cyan") #矩形相  
frame.setFrameShape(QFrame.Box) #设置边框类型  
frame.setFrameShadow(QFrame.Raised) #将边框凸起  
frame.setLineWidth(6) #边框外线宽度  
frame.setMidLineWidth(12) #边框中线宽度
```



边框外线，就是最外边的淡绿色线，边框中线，就是中间灰色线。

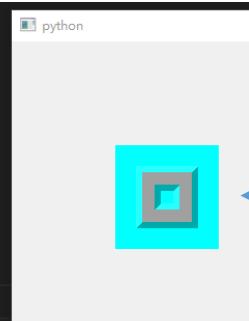
```
QFrame.Panel #设置矩形框为平面
```

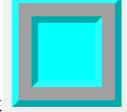
```
frame.setStyleSheet("background-color:cyan") #矩  
frame.setFrameShape(QFrame.Panel) #设置边框为平面  
frame.setFrameShadow(QFrame.Raised) #将边框凸起  
frame.setLineWidth(6) #边框外线宽度  
frame.setMidLineWidth(12) #边框中线宽度
```



```
QFrame :: setFrameRect(参数) #设置 frame 突出部分的矩形框架尺寸
```

```
frame = QFrame(w) #创建矩形框  
frame.resize(100,100) #矩形框大小  
frame.move(100,100)  
frame.setStyleSheet("background-color:cyan") #矩形框背景颜色  
frame.setFrameShape(QFrame.Box) #设置边框为平面  
frame.setFrameShadow(QFrame.Raised) #将边框凸起  
frame.setLineWidth(6) #边框外线宽度  
frame.setMidLineWidth(12) #边框中线宽度  
frame.setFrameRect(QRect(20,20,60,60)) #设置凸起部分矩形长宽
```



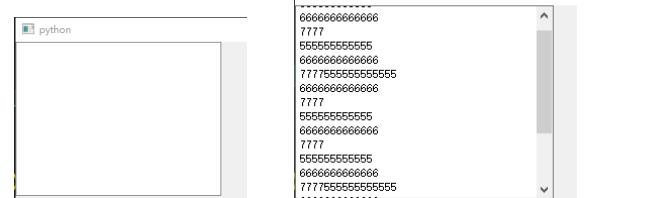
本来是这样 ，但是 setFrameRect，设置框架长宽之后，变成如上这样 

## QAbstractScrollArea 滚动空间类，继承 QAbstractScrollArea 类的子类都具备滚动功能，如滚动条

QTextEdit 多行文本使用

```
多行文本对象 = QTextEdit(父对象) #该类继承至 QAbstractScrollArea  
多行文本对象 = QTextEdit(“默认字符串”,父对象)
```

```
from PyQt5.Qt import *  
import sys  
  
app = QApplication(sys.argv)  
  
w = QWidget()  
w.resize(500,500)  
te = QTextEdit(w) #创建多行文本  
  
w.show()  
sys.exit(app.exec_())
```



```
QTextEdit::append("填入追加字符串") #多行文本，追加下一行文本
```

```
w = QWidget()
w.resize(500,500)
te = QTextEdit(w) #创建多行文本
te.append("11111") #在输入字符的下一排插入文本
te.append("22222") #在输入字符的下一排插入文本
te.append("33333") #在输入字符的下一排插入文本
```



是按照行追加

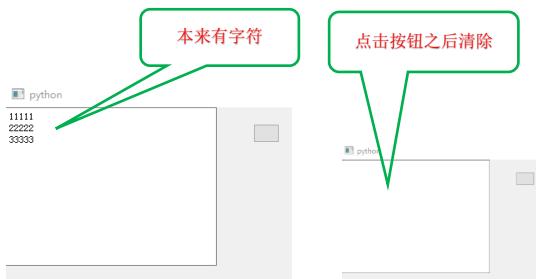
```
QTextEdit::setText("") #空冒号可以清空多行文本内容
```

```
w = QWidget()
w.resize(500,500)
te = QTextEdit(w) #创建多行文本
te.append("11111") #在输入字符的下一排插入文本
te.append("22222") #在输入字符的下一排插入文本
te.append("33333") #在输入字符的下一排插入文本

Btn = QPushButton(w)
Btn.move(300,20)

def cao():
    te.setText("") #清空多行文本

Btn.clicked.connect(cao) #按钮按下清空对行文本
```



```
QTextEdit::clear() #clear 也可以清空多行文本内容
```

```
te = QTextEdit(w) #创建多行文本
te.append("11111") #在输入字符的下一排插入文本
te.append("22222") #在输入字符的下一排插入文本
te.append("33333") #在输入字符的下一排插入文本

Btn = QPushButton(w)
Btn.move(300,20)

def cao():
    te.clear() #清空多行文本

Btn.clicked.connect(cao) #按钮按下清空对行文本
```

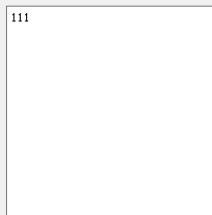


```
QTextEdit::setPlainText(填入内容) #覆盖以前的内容，填入新内容
```

```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

te = QTextEdit("111",w)
te.move(50,50)
te.resize(200,200)
```



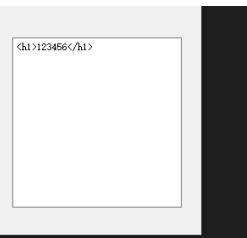
默认 Text 文本框显示 111

```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

te = QTextEdit("111",w)
te.move(50,50)
te.resize(200,200)

te.setPlainText("<h1>123456</h1>") #新增文本内容，覆盖老内容
```

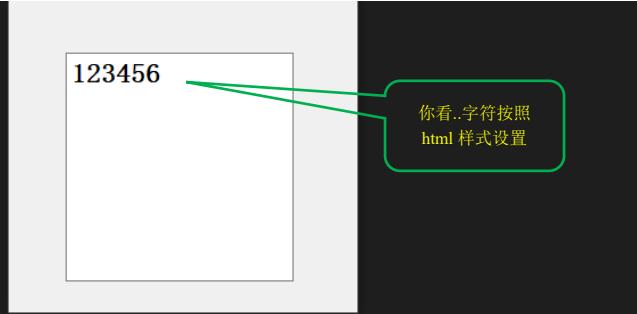


新增内容 html 字符

```
QTextEdit::setHtml(传入 html 格式内容) #传入文本的字符，按照 html 语法设置的特性显示
```

```
app = QApplication(sys.argv)
w = QWidget()
w.resize(500,500)

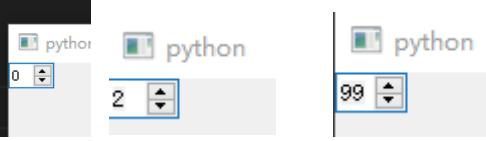
te = QTextEdit("111",w)
te.move(50,50)
te.resize(200,200)
te.setHtml("<h1>123456</h1>") #传入文本的字符，按照html语法设置的特性显示
```



## 步长编辑器 QSpinBox

对象 = QSpinBox(父对象) #创建步长编辑器

```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
```



默认步长编辑器最多增加到 99

QSpinBox::setMaximum(参数) #步长编辑器最大长度，参数就是设置的长度值

```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
sb.setMaximum(200) #设置步长编辑器增加的最大长度
```



QSpinBox::setMinimum(参数) #步长编辑器最小长度

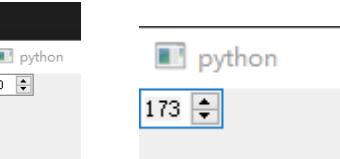
```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
sb.setMaximum(200) #设置步长编辑器增加的最大长度
sb.setMinimum(20) #设置步长编辑器减小的最小长度
```



最小只能到 20，最大可以到 200

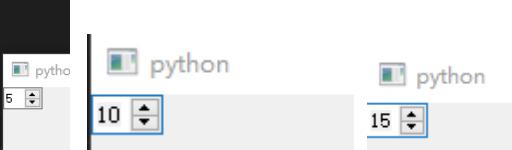
QSpinBox::setRange(最小值,最大值) #一次性设置步长最小值最大值

```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
sb.setRange(20,200) #直接设置步长最小值最大值范围
```



QSpinBox::setSingleStep(参数) #设置步长最小增减值

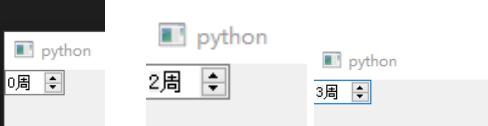
```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
sb.setSingleStep(5) #设置步长最小增加值
```



你看，点击一次增加，步长按照 5 个 5 个的加。

QSpinBox::setSuffix("周") #设置数字单位

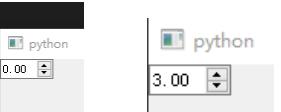
```
w = QWidget()
w.resize(500,500)
sb = QSpinBox(w) #创建步长编辑器
sb.setSuffix("周") #设置数字单位
```



如果想设置某月某日，或者百分比什么的，就可以用这种方式

对象 = QDoubleSpinBox(父对象) #带浮点显示步长编辑器

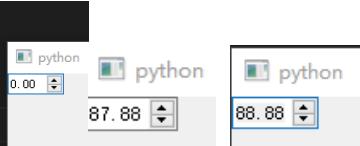
```
w = QWidget()
w.resize(500,500)
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setValue(3.00)
```



你看，步长显示器默认两个小数，但是递增是按照小数点前的数据递增的，比如 1.00,2.00,3.00

```
QDoubleSpinBox:: setMaximum(参数) #设置浮点数最大长度，最小长度同理用
setMinimum
```

```
w = QWidget()
w.resize(500,500)
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setMaximum(88.88) #设置浮点数最大长度
```



```
QDoubleSpinBox::setSingleStep(参数) #设置小数递增也是用步长最小增减值
```

```
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setSingleStep(0.01) #设置步长长度，这样可以小数递增
```



槽函数中直接用 lambda 实现

```
w = QWidget()
w.resize(500,500)
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setSingleStep(0.01) #设置步长长度，这样可以小数递增

Btn = QPushButton(w)
Btn.move(100,20)
Btn.clicked.connect(lambda: sb.setValue(66.66)) #如果按钮触发代码简短，可以直接在槽函数中用lambda
```



```
QDoubleSpinBox::value() #获取当前步长编辑器的值
```

```
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setSingleStep(0.01) #设置步长长度，这样可以小数递增

Btn = QPushButton(w)
Btn.move(100,20)
Btn.clicked.connect(lambda: print(sb.value())) #直接获取数据
```



取消按钮，步长编辑器数据改变一次，槽函数执行一次

```
QDoubleSpinBox::valueChanged.connect(槽函数)
```

```
w = QWidget()
w.resize(500,500)
sb = QDoubleSpinBox(w) #创建带浮点数显示得步长编辑器
sb.setSingleStep(0.01) #设置步长长度，这样可以小数递增

sb.valueChanged.connect(lambda val: print(val)) #直接获取步长编辑器数据
```



```
def cao(val):
    print(val)
sb.valueChanged.connect(cao) #直接获取步长编辑器数据
```

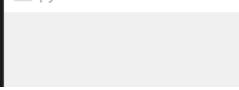
槽函数执行，直接用 lambda 接收第 1 个形参，这个形参就是得到的数据

这样也可以

## QTime 系统时间获取

```
对象 = QTime.currentTime() #获取系统当前运行时间，从 0 开始计算  
对象.start() #启动系统时间获取  
对象.elapsed() #获取当前系统运行时间 ms 毫秒
```

```
w = QWidget()  
w.resize(500,500)  
xtime = QTime.currentTime() #获取当前时间  
xtime.start()  
  
print(xtime.elapsed())
```

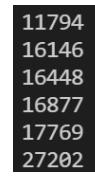
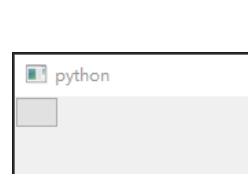


为什么是 0?

因为第 1 次获取系统运行时间之后，程序运行结束了。

不能将 elapsed() 放入 while 大循环，不然会一直获取系统时间

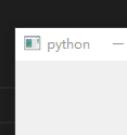
```
w = QWidget()  
w.resize(500,500)  
xtime = QTime.currentTime() #获取当前时间  
xtime.start()  
  
Btn = QPushButton(w)  
Btn.clicked.connect(lambda :print(xtime.elapsed())) #用按钮事件触发方式查看系统运行时间
```



点击按钮 获取程序运行时间，单位为 ms

```
日期时间变量 = QDateTime.currentDateTime() #获取当前日期时间
```

```
w = QWidget()  
w.resize(500,500)  
xtime = QDateTime.currentDateTime() #获取当前日期时间  
print(xtime)
```



```
PyQt5.QtCore.QDateTime(2021, 8, 17, 20, 54, 47, 218)
```

```
对象.date() #获取日期，但是重复执行(对象.date)日期是不会更新的，必须执行  
QDateTime.currentDateTime() 之后，(对象.date) 才会更新
```

```
w = QWidget()  
w.resize(500,500)  
xtime = QDateTime.currentDateTime() #获取当前日期时间  
  
print(xtime.date()) #只获取日期
```

```
PyQt5.QtCore.QDate(2021, 8, 17) 打印结果
```

```
对象.time() #只获取当前时间，时间更新和.date 同理
```

```
xtime = QDateTime.currentDateTime() #获取当前日期时间  
print(xtime.date()) #只获取日期  
print(xtime.time()) #只获取时间
```

```
PyQt5.QtCore.QDate(2021, 8, 17)  
PyQt5.QtCore.QTime(21, 5, 56, 465) 打印结果 465 是 ms 毫秒单位
```

时间更新需要执行 QDateTime.currentDateTime()

```
w = QWidget()  
w.resize(500,500)  
xtime = QDateTime.currentDateTime() #获取当前日期时间  
  
def cao():  
    xtime = QDateTime.currentDateTime() #更新实时日期和时间  
    print(xtime.date()) #只获取日期  
    print(xtime.time()) #只获取时间  
  
Btn = QPushButton(w)  
Btn.clicked.connect(cao) #用按钮事件触发方式查看系统运行时间  
PyQt5.QtCore.QDate(2021, 8, 17)  
PyQt5.QtCore.QTime(21, 11, 24, 17)  
PyQt5.QtCore.QDate(2021, 8, 17)  
PyQt5.QtCore.QTime(21, 11, 24, 466)  
PyQt5.QtCore.QDate(2021, 8, 17)  
PyQt5.QtCore.QTime(21, 11, 24, 954)  
PyQt5.QtCore.QDate(2021, 8, 17)  
PyQt5.QtCore.QTime(21, 11, 36, 313)
```



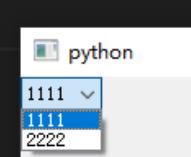
点击按钮，查看更新时间

## QComboBox 下拉选择框

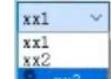
```
对象 = QComboBox(父对象) #创建下拉框
```

```
QComboBox::addItem(传入下拉框显示得字符) #创建下拉行，执行一次创建一个下拉行
```

```
cb = QComboBox(w) #创建下拉选择框  
cb.addItem("1111") #添加一行下拉框  
cb.addItem("2222") #添加2行下拉框
```



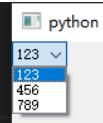
```
cb.addItem(QIcon("xxx.png"), "xx3")
```



加入 QIcon 下拉框可以显示图标

```
QComboBox::addItems(传入元组或者列表) #一次性创建多个下拉框
```

```
cb = QComboBox(w) #创建下拉选择框  
cb.addItems(["123", "456", "789"]) #Items一次创建多行下拉框
```



```
cb = QComboBox(w) #创建下拉选择框
```

```
cb.addItems(("123", "456", "789")) #Items元组也可以一次创建多个下拉框
```



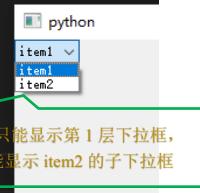
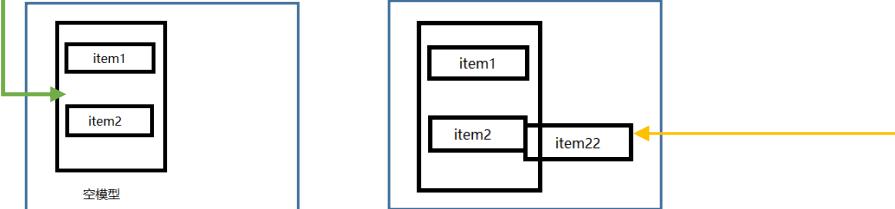
```
对象 = QStandardItemModel() #创建一个空模型
```

```
cb = QComboBox(w) #创建下拉选择框  
  
model = QStandardItemModel() #创建一个空模型  
item1 = QStandardItem("item1") #在模型第1层定义下拉框1  
item2 = QStandardItem("item2") #在模型第1层定义下拉框2  
item22 = QStandardItem("item22") #该item22 将加载到下拉框2的子下拉框中
```



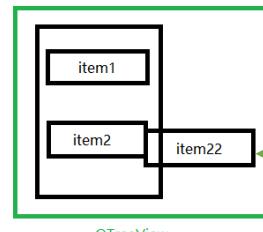
```
model.appendRow(item1) #在模型第1层加下拉框1  
model.appendRow(item2) #在模型第1层加下拉框2  
item2.appendRow(item22) #下拉框22加载到第1层下拉框2下面  
cb.setModel(model) #将设置好的下拉菜单模型放入选择框
```

这是因为 QComboBox 创建的下拉列表，不支持树形结构，所以 item2 无法展示分支



```
cb = QComboBox(w) #创建下拉选择框  
cb.resize(400,50) #加大下拉框尺寸才能看到树形结构  
model = QStandardItemModel() #创建一个空模型  
item1 = QStandardItem("item1") #在模型第1层定义下拉框1  
item2 = QStandardItem("item2") #在模型第1层定义下拉框2  
item22 = QStandardItem("item22") #该item22 将加载到下拉框2的子下拉框中
```

```
model.appendRow(item1) #在模型第1层加下拉框1  
model.appendRow(item2) #在模型第1层加下拉框2  
item2.appendRow(item22) #下拉框22加载到第1层下拉框2下面  
cb.setModel(model) #将设置好的下拉菜单模型放入选择框  
cb.setView(QTreeView(cb)) #用QTreeView将模型放入树形结构，然后将树形贴在QComboBox下拉框中显示
```



模型放入树形结构

```
返回值 = QComboBox::count() #获取下拉框数量
```

```
w = QWidget()  
w.resize(500,500)  
  
cb = QComboBox(w) #创建下拉选择框  
cb.resize(400,50) #加大下拉框尺寸才能看到树形结构  
cb.addItems(["abc","123","456"]) #一次创建3个下拉框  
  
Btn = QPushButton(w)  
Btn.move(100,80)  
Btn.clicked.connect(lambda : print(cb.count())) #打印获取下拉框数量
```



3

得到 3 行下拉框

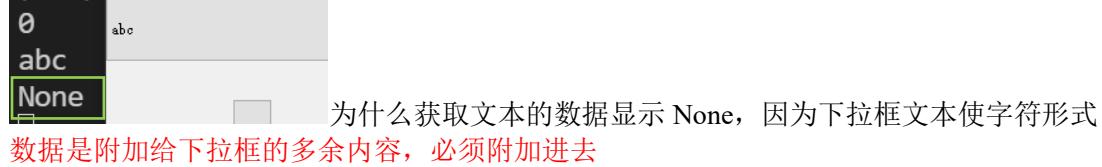
```
返回值 = QComboBox::currentIndex() #获取当前选中的哪一行下拉框  
返回值 = QComboBox::currentText() #获取当前选中下拉框的内容
```

```
cb = QComboBox(w) #创建下拉选择框  
cb.resize(400,50) #加大下拉框尺寸才能看到树形结构  
cb.addItems(["abc", "123", "456"]) #一次创建3个下拉框  
  
Btn = QPushButton(w)  
Btn.move(100,80)  
Btn.clicked.connect(lambda : print(cb.currentIndex())) #打印当前选中的哪一行下拉框  
Btn.clicked.connect(lambda : print(cb.currentText())) #打印当前选中的下拉框内容(文本内容用Text获取)
```

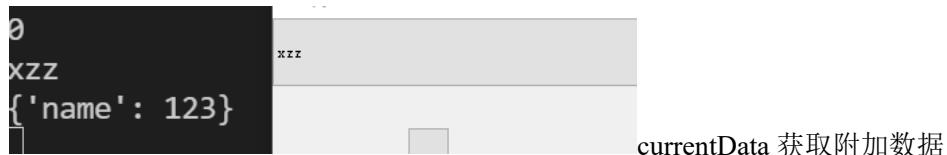


```
QComboBox::currentData() #获取当前选中下拉框的数据
```

```
cb = QComboBox(w) #创建下拉选择框  
cb.resize(400,50) #加大下拉框尺寸才能看到树形结构  
cb.addItems(["abc", "123", "456"]) #一次创建3个下拉框  
  
Btn = QPushButton(w)  
Btn.move(100,80)  
Btn.clicked.connect(lambda : print(cb.currentIndex())) #打印当前选中的哪一行下拉框  
Btn.clicked.connect(lambda : print(cb.currentText())) #打印当前选中的下拉框内容(文本内容用Text获取)  
Btn.clicked.connect(lambda : print(cb.currentData())) #打印当前选中的下拉框内容(数据内容Data获取)
```



```
cb = QComboBox(w) #创建下拉选择框  
cb.resize(400,50) #加大下拉框尺寸才能看到树形结构  
cb.addItem("xzz", {"name":123}) #只能用Item创建的下拉框可以加额外数据  
  
Btn = QPushButton(w)      用字典形式加入数据  
Btn.move(100,80)  
Btn.clicked.connect(lambda : print(cb.currentIndex())) #打印当前选中的哪一行下拉框  
Btn.clicked.connect(lambda : print(cb.currentText())) #打印当前选中的下拉框内容(文本内容用Text获取)  
Btn.clicked.connect(lambda : print(cb.currentData())) #打印当前选中的下拉框内容(数据内容Data获取)
```



```
cb.addItem("xzz", 123) #只能用Item创建的下拉框可以加额外数据  
直接写数据也可以
```

```
QComboBox::currentIndexChanged.connect(槽函数(值))
```

```
cb = QComboBox(w) #创建下拉选择框
cb.addItem("xzz")
cb.addItem("123456")
cb.addItem("123456789101112")
cb.currentIndexChanged.connect(lambda val:print(val)) #选中下拉框的时候马上得到下拉框行号
val 变量就是下拉框变化返回的行号
```



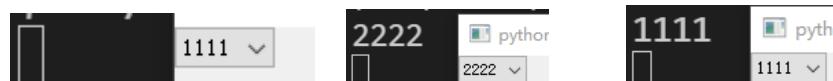
虽然 currentIndexChanged 获取的是当前选中的行号，你可以在槽函数中执行 currentText 获得内容。

```
QComboBox::currentIndexChanged[str].connect(槽函数) #[str]指定传递给槽函数形参的内容为 addItem 定义的字符，而不是下拉框编号
```

```
cb = QComboBox(w) #创建下拉选择框
cb.addItem("1111")
cb.addItem("2222")

def cao(val): #槽函数得到形参显示出来
    print(val) #得到addItem字符串形式

cb.currentIndexChanged[str].connect(cao) #信号前面加入Str就是将下拉框行字符串传递给形参,
#这样选中某个下拉框不会传编号，而是addItem定义的字符串
```



## 两个下拉框联动显示案例

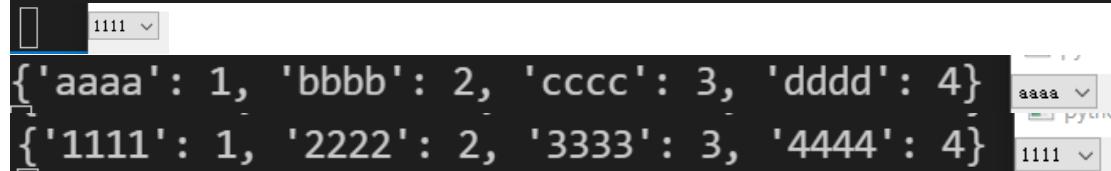
```
w = QWidget()
w.resize(500,500)

numdict={"1111": {"1111":1,"2222":2,"3333":3,"4444":4}, #字典里面嵌套字典
         "aaaa": {"aaaa":1,"bbbb":2,"cccc":3,"dddd":4} }

cb = QComboBox(w) #创建下拉选择框
cb.addItem("1111")
cb.addItem("aaaa")

def cao(val): #槽函数得到形参显示出来
    dictnum = numdict[val]
    print(dictnum)

cb.currentIndexChanged[str].connect(cao) #信号前面加入Str就是将下拉框行字符串传递给形参,
#这样选中某个下拉框不会传编号，而是addItem定义的字符串
```



```

numdict={"1111":{"1111":1,"2222":2,"3333":3,"4444":4}, #字典里面嵌套字典
        "aaaa":{"aaaa":1,"bbbb":2,"cccc":3,"dddd":4} }

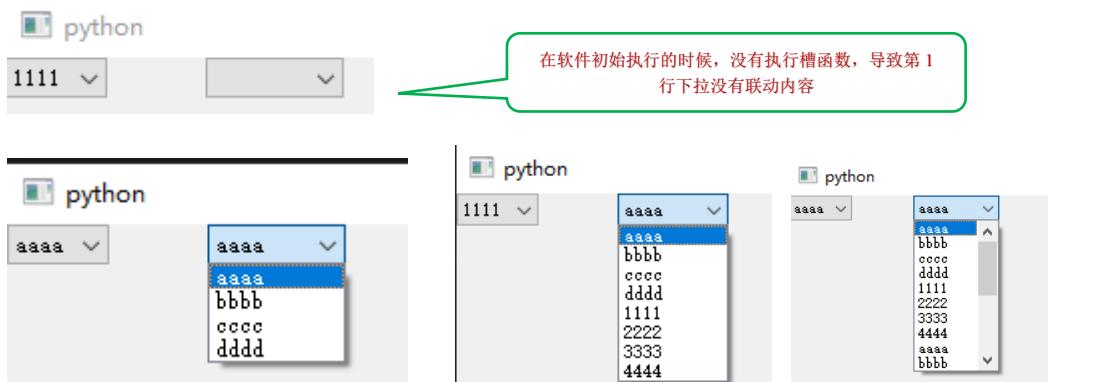
cb = QComboBox(w) #创建下拉选择框
cb.addItem("1111")
cb.addItem("aaaa")

ld = QComboBox(w) #创建下拉选择框2
ld.move(100,0)

def cao(val): #槽函数得到形参显示出来
    dictnum = numdict[val]
    ld.addItems(dictnum.keys()) #将字典嵌套的内容赋值给另外一个下拉框，要用addItems赋值字典多个内容才对

cb.currentIndexChanged[str].connect(cao) #信号前面加入Str就是将下拉框行字符串传递给形参,
                                         #这样选中某个下拉框不会传编号，而是addItem定义的字符

```



下拉之后产生了联动内容  
但是下拉另外一行，联动内容会叠加之前下拉的联动内容，来回下拉选择，导致内容不断重复累加。

```

def cao(val): #槽函数得到形参显示出来
    dictnum = numdict[val]
    ld.clear() #清除上一次下拉框内容，避免出现追加内容
    ld.addItems(dictnum.keys()) #将字典嵌套的内容赋值给另外一个下拉框，要用addItems赋值字典多个内容才对

cb.currentIndexChanged[str].connect(cao) #信号前面加入Str就是将下拉框行字符串传递给形参,
                                         #这样选中某个下拉框不会传编号，而是addItem定义的字符

```

来回切换没有出现追加问题

```

cb = QComboBox(w) #创建下拉选择框
cb.addItem("1111")
cb.addItem("aaaa")

ld = QComboBox(w) #创建下拉选择框2
ld.move(100,0)

ld.currentIndexChanged[str].connect(lambda val:print(val)) #监听联动下拉框的变化

def cao(val): #槽函数得到形参显示出来
    dictnum = numdict[val]
    ld.clear() #清除上一次下拉框内容，避免出现追加内容
    ld.addItems(dictnum.keys()) #将字典嵌套的内容赋值给另外一个下拉框，要用addItems赋值字典多个内容才对

cb.currentIndexChanged[str].connect(cao) #信号前面加入Str就是将下拉框行字符串传递给形参,
                                         #这样选中某个下拉框不会传编号，而是addItem定义的字符

```



## 滑块使用 QSlider

```
对象 = QSlider(父对象) #创建滑块对象
```

```
w = QWidget()  
w.resize(500,500)
```

```
sl = QSlider(w) #创建滑块
```



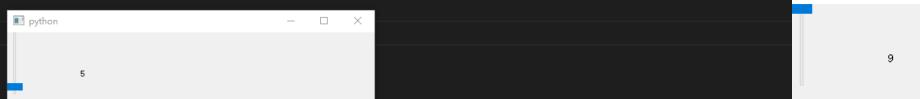
滑块滑动执行槽函数

```
QSlider::valueChanged.connect(槽函数) #滑块滑动, 不停的执行槽函数
```

```
sl = QSlider(w) #创建滑块
```

```
label = QLabel(w) #创建标签  
label.setText("0")  
label.move(100,50)
```

```
sl.valueChanged.connect(lambda val: label.setText(str(val))) #滑块滑动执行槽函数,  
#获取滑块值用str转成字符给label显示
```



滑块默认值最多到 9，而且滑块开始滑动位置数据有问题

```
label = QLabel(w) #创建标签  
label.setText("0")  
label.move(100,50)  
label.resize(100,30) #滑块默认是0~99, 标签太小所以没看见以为都是个位在重复显示
```

```
sl = QSlider(w) #创建滑块  
sl.move(100,100)
```



```
QSlider::setMaximum(参数) #设置滑块最大值  
QSlider::setMinimum(参数) #设置滑块最小值
```

```
label = QLabel(w) #创建标签  
label.setText("0")  
label.move(100,50)  
label.resize(100,30) #滑块默认是0~99, 标签太小所以没看见以为都是个位在重复显示  
  
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100) #设置滑块最大值  
sl.setMinimum(60) #设置滑块最小值
```



稍微动下滑块就 60 以上了

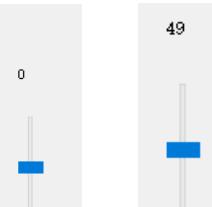
```
QSlider::setSingleStep(步长值) #设置滑块步长，滑块按照 5 个步长移动，但是鼠标无效，必须键盘上下键操作
```

```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(60) #设置滑块最小值  
sl.setSingleStep(5) #设置滑块步长，滑块按照5个步长移动，但是鼠标无效，必须键盘上下键操作
```



```
QSlider::setValue(50) #自动设置滑块数值
```

```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值
```



滑动点点，滑块到

50 了，初始程序文本显示为 0 是因为槽函数没有给文本赋值。但是滑块值其实是 50。

```
QSlider::setOrientation(Qt.Horizontal) #设置滑块摆放方向
```

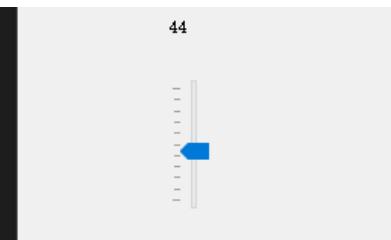
```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setOrientation(Qt.Horizontal) #设置滑块摆放方向
```



```
QSlider::setTickPosition(参数) #滑块显示刻度线
```

参数：  
QSlider.TicksLeft 滑块左边显示刻度线  
QSlider.TicksRight 滑块右边显示刻度线  
QSlider.TicksBothSides 滑块两边显示刻度线

```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksLeft) #滑块显示刻度线
```



```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线
```



滑块两边显示刻度线

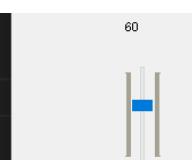
```
QSlider::setTickInterval(刻度间距值) #设置刻度线间距
```

```
sl = QSlider(w) #创建滑块  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线  
sl.setTickInterval(10) #设置刻度线间距, 每隔10个值绘制1个刻度线
```

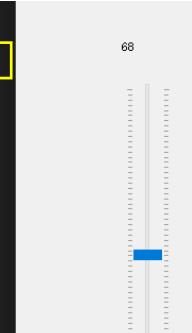


间距 1 个值绘制 1 个刻度线, 那么看起刻度线就很密

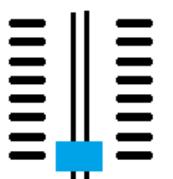
```
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线  
sl.setTickInterval(1) #设置刻度线间距, 每隔1个值绘制1个刻度线
```



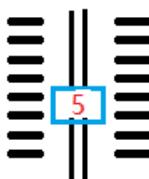
```
sl = QSlider(w) #创建滑块  
sl.resize(50,500) #加长滑块长度, 间隔1个单位的刻度线看起就不是很密  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线  
sl.setTickInterval(1) #设置刻度线间距, 每隔1个值绘制1个刻度线
```



在滑块按钮上显示滑动数值



滑块停止移动, 显  
示滑块本身的样子



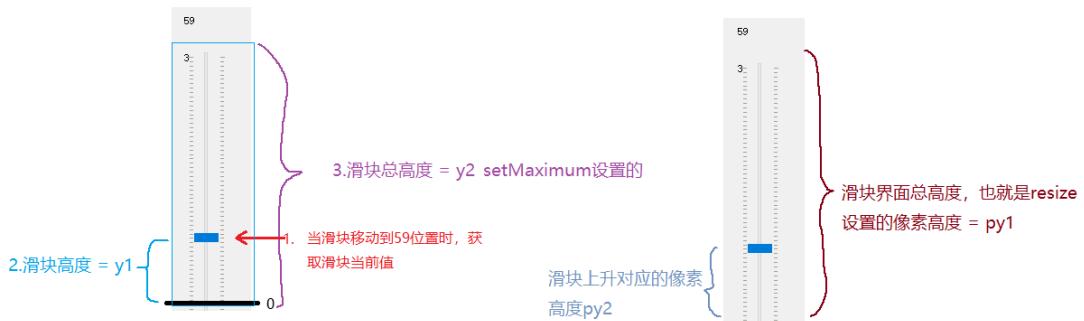
滑块手动移动显  
示当前数值

```
label = QLabel(w) #创建标签  
label.setText("0")  
label.move(100,50)  
label.resize(100,30) #滑块默认是0~99, 标签太小所以没看见以为都是个位在重复显示
```



```
sl = QSlider(w) #创建滑块  
sl.resize(50,500) #加长滑块长度, 间隔1个单位的刻度线看起就不是很密  
sl.move(100,100)  
sl.setMaximum(100)#设置滑块最大值  
sl.setMinimum(0) #设置滑块最小值  
sl.setValue(50) #自动设置滑块数值  
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线
```

```
lab = QLabel(sl) #将标签嵌入进滑块  
lab.setText("3")
```



y1 y2 和 py1 py2 是比例关系  
也就是滑块值与像素高度的关系

如滑块当前数值  $y1 = 59$   
 $y2 = 100$  (因为 `setMaximum(100)`)

$$\frac{y}{\text{整个控件高度 } py1} = \frac{y2 - \text{当前滑块值}}{y2} \quad \rightarrow \quad \frac{y}{\text{整个控件高度 } py1} = 1 - \text{当前滑块值}/y2$$

公式优化后

$$y = (1 - \text{当前滑块值} / y2) \times \text{整个控件高度 } py1$$

```

sl = QSlider(w) #创建滑块
sl.resize(50,500) #加长滑块长度, 间隔1个单位的刻度线看起就不是很密
sl.move(100,100)
sl.setMaximum(100)#设置滑块最大值
sl.setMinimum(0) #设置滑块最小值
sl.setValue(50) #自动设置滑块数值
sl.setTickPosition(QSlider.TicksBothSides) #滑块显示刻度线

lab = QLabel(sl) #将标签嵌入进滑块
lab.setText("3")

def cao(val):
    print(val)
    x = (sl.width() - lab.width())/2 #(整个滑块控件像素宽度 - 嵌入滑块的标签宽度)/2 = 标签居中
    y = (1 - sl.value() / sl.maximum() - sl.minimum()) * sl.height()
    lab.move(x,y)

sl.valueChanged.connect(cao) #滑块滑动执行槽函数,

```

因为滑块最小值为 0, 所以 minimum 可以不要



程序初始化的时候滑块标签在左上角看着很不习惯

程序初始化运行没有执行槽函数



滑块点动, 执行槽函数

解决初始程序, 标签在滑块左上角问题。

```

lab = QLabel(sl) #将标签嵌入进滑块
lab.setText("3")
lab.hide() #初始程序隐藏滑块上的标签

```

```

def cao(val):
    print(val)
    x = (sl.width() - lab.width()) / 2 #(整个滑块控件像素宽度 - 嵌入滑块的标签宽度)/2 = 标签居中
    y = (1 - sl.value() / sl.maximum() - sl.minimum()) * sl.height()
    lab.show() #执行滑块的时候显示滑块上标签
    lab.move(x,y)

```

滑块标签修改成功。

下面让滑块滑动，数值也跟着变化

```

lab = QLabel(sl) #将标签嵌入进滑块
lab.resize(20,20) #设置标签大小，显示0~99
lab.setText("3")
lab.hide() #初始程序隐藏滑块上的标签

def cao(val):
    print(val)
    x = (sl.width() - lab.width()) / 2 #(整个滑块控件像素宽度 - 嵌入滑块的标签宽度)/2 = 标签居中
    y = (1 - sl.value() / sl.maximum() - sl.minimum()) * sl.height()
    lab.show() #执行滑块的时候显示滑块上标签
    lab.move(x,y)
    lab.setText(str(val)) #显示当前滑动数字

```

```

lab = QLabel(sl) #将标签嵌入进滑块
lab.resize(20,20) #设置标签大小，显示0~99
lab.setText("3")
lab.hide() #初始程序隐藏滑块上的标签

def cao(val):
    print(val)
    x = (sl.width() - lab.width()) / 2 #(整个滑块控件像素宽度 - 嵌入滑块的标签宽度)/2 = 标签居中
    y = (1 - sl.value() / sl.maximum() - sl.minimum()) * (sl.height()-lab.height())
    lab.show() #执行滑块的时候显示滑块上标签
    lab.move(x,y)
    lab.setText(str(val)) #显示当前滑动数字

sl.valueChanged.connect(cao) #滑块滑动执行槽函数,

```

但是滑块移动得越来越大，或者移动得越来越小，标签还是会移除滑块中心。自行解决 BUG

## 带上下箭头的滑块 QScrollBar

对象 = QScrollBar(父对象) #创建上下箭头滑块

```

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

sb = QScrollBar(w) #创建上下箭头滑块

w.show()
sys.exit(app.exec_())

```

```
w = QWidget()
w.resize(500,500)

sb = QScrollBar(w) #创建上下箭头滑块
sb.resize(30,200) #设置宽度高度, 就可以滑动滚动条了
```

Qt.Horizontal #水平滚动条

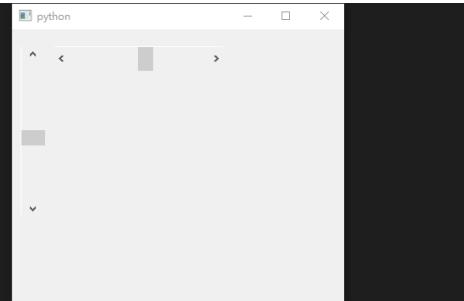
```
sb = QScrollBar(Qt.Horizontal,w) #创建水平滚动条
sb.resize(200,30) #设置宽度高度, 水平滚动条宽度要宽些
```

```
QScrollBar::valueChanged.connect(槽函数) #滚动条滑动槽函数执行
QScrollBar::setValue(设置滚动值) #设置滚动条滚动值
```

```
w = QWidget()
w.resize(500,500)

sbv = QScrollBar(w) #创建垂直滚动条
sbv.resize(30,200)
sbv.move(10,20)
sbh = QScrollBar(Qt.Horizontal,w) #创建水平滚动条
sbh.resize(200,30) #设置宽度高度, 水平滚动条宽度要宽些
sbh.move(50,20)

sbv.valueChanged.connect(lambda val: sbh.setValue(val)) #垂直滚动条滑动值间接改变水平滚动条滑动值
```



对象 = QDial(父对象) #创建圆形滚动盘

```
w = QWidget()
w.resize(500,500)

dia = QDial(w)#创建圆形滚动盘

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



```
QDial::valueChanged.connect(槽函数) #滚动盘滑动触发槽函数
```

```
w = QWidget()
w.resize(500,500)

dia = QDial(w)#创建圆形滚动盘

dia.valueChanged.connect(lambda val:print(val)) #获取滚动盘滚动数据

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

滚动盘滚动数据默认是 0~100

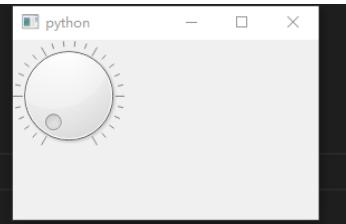
22
23
24
25
26

```
QDial::SetNotchesVisible(True) #给滚动盘加刻度
```

```
w = QWidget()
w.resize(500,500)

dia = QDial(w)#创建圆形滚动盘
dia.setNotchesVisible(True) #给滚动盘加刻度

dia.valueChanged.connect(lambda val:print(val)) #获取滚动盘滚动数据
```

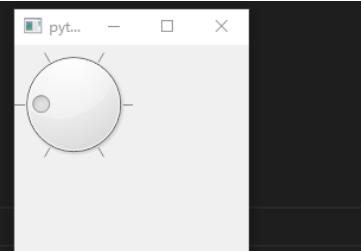


```
QDial::SetNotchTarget(刻度间隔) #设置刻度之间间隔距离
```

```
w = QWidget()
w.resize(500,500)

dia = QDial(w)#创建圆形滚动盘
dia.setNotchesVisible(True) #给滚动盘加刻度
dia.setNotchTarget(20) #刻度之间间隔20

dia.valueChanged.connect(lambda val:print(val)) #获取滚动盘滚动数据
```



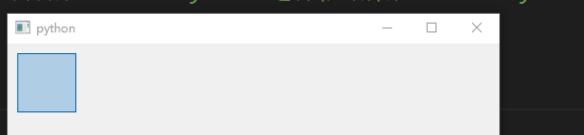
## 区域选择的控件(橡皮筋选择控件) QRubberBand

对象 = QRubberBand(选择框形状, 父对象) #选择框创建

```
w = QWidget()
w.resize(500,500)

rb = QRubberBand(QRubberBand.Rectangle,w) #创建橡皮筋选择框, Rectangle选择框为矩形
rb.setGeometry(10,10,60,60) #设置选择框开始位置x=10, y=10, 选择框结束位置x=60, y=60
#注意选择框默认是false, 不显示
rb.show() #所以我们用show让选择框显示

w.show()
```



这个矩形选择框, 实际应用是希望鼠标点击后产生



这样操作才合理

```
QRubberBand::setGeometry(起始位置 x, 起始位置 y, 显示宽度, 显示高度) #对几何对象的位置, 长宽进行修改实时修改
```

注意: 对于存在 Layout (布局) 方式的 Widget (窗口), 如布局管理器。setGeometry()函数只能设置 (x,y), 在设置宽高上不起作用, 需改用 setFixedSize()函数;

```
w = QWidget()
w.resize(100,200)
w.show() #我这儿本来已经显示了100x200的矩形截面
time.sleep(3)#等待3秒
w.setGeometry(300,300,1000,1000)#矩形起始位置被实时改变了, 从x=300.y=300开始, 长1000, 宽1000
sys.exit(app.exec_())
```



开始运行是小界面。连 show()都运行过了 变成大  
界面。所以 setGeometry 是可以实时修改对象矩形大小的。

返回 x,y,长,宽值 = QRect(矩形起始 x 坐标,矩形起始 y 坐标,矩形长,矩形宽) #创建一个指定长宽的矩形类

```
w = QWidget()
w.resize(100,200)
w.show() #我这儿本来已经显示了100x200的矩形截面
qr = QRect(100,100,500,500) #给矩形一个坐标和尺寸
time.sleep(3)#等待3秒
w.setGeometry(qr)#窗口使用QRect设置的坐标和尺寸
sys.exit(app.exec_())
```



3 秒前的界面

3 秒后的界面

```

class XWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("xxxxxxxx")
        self.resize(500, 500)
        #self.setupUi()
    def disp(self):
        pass

    def mousePressEvent(self, evt): #鼠标按下事件回调
        QMouseEvent
        self.rb = QRubberBand(QRubberBand.Rectangle, self) #创建橡皮选择框, Rectangle选择框为矩形
        self.originPos = evt.pos() #获取当前点下的坐标
        self.rb.setGeometry(QRect(self.originPos, QSize())) #可以传(self, int, int, int, int) 或者QRect
        self.rb.show()

    def mouseMoveEvent(self, evt): #鼠标移动事件回调
        self.rb.setGeometry(QRect(self.originPos, evt.pos()))

    def mouseReleaseEvent(self, evt):
        pass

```

Qsize() 表示长宽为 0, 所以鼠标点击下, 什么都没有反应

evt.pos()就是返回鼠标当前的 x,y 值, 将坐标赋值给 QRect

鼠标滑动的时候, 用 setGeometry 实时修改矩形尺寸, 传入鼠标变化的值 evt.pos 与鼠标按下的值 self.originPos 形成矩形

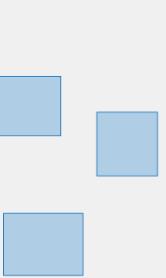
```

app = QApplication(sys.argv)

#w = QWidget()
#w.resize(500,500)
xw = XWidget()
xw.show()

#w.show()
sys.exit(app.exec_())

```



但是我发现选择框鼠标不能反向选择

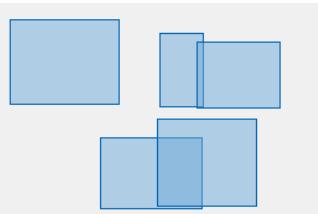
返回 x,y, 长, 宽值 = QRect(矩形起始 x 坐标, 矩形起始 y 坐标, 矩形长, 矩形宽).normalized() xy 支持负值反转

```

def mouseMoveEvent(self, evt): #鼠标移动事件回调
    self.rb.setGeometry(QRect(self.originPos, evt.pos()).normalized()) #支持负数反向拖动,
                                                                           #产生负数两点交换

```

支持反向选择



鼠标没得选择, 就会残留上一次的选择框, 如何保证鼠标选择只产生一次选择框?

```

def mousePressEvent(self, evt): #鼠标按下事件回调
    QMouseEvent
    self.rb = QRubberBand(QRubberBand.Rectangle, self) #创建橡皮选择框, Rectangle选择框为矩形
    self.originPos = evt.pos() #获取当前点下的坐标
    self.rb.setGeometry(QRect(self.originPos, QSize())) #可以传(self, int, int, int, int) 或者QRect
    self.rb.show() #鼠标点击的时候显示

def mouseMoveEvent(self, evt): #鼠标移动事件回调
    self.rb.setGeometry(QRect(self.originPos, evt.pos()).normalized()) #支持负数反向拖动,
                                                                           #产生负数两点交换

def mouseReleaseEvent(self, evt): #鼠标松开事件
    self.rb.hide() #鼠标松开的时候隐藏

```



鼠标滑动的时候显示，

鼠标松开的时候隐藏

这样虽然解决了只产生一次选择框的问题，但是每次鼠标松开的时候，选择框只是被隐藏，但是没有被消除，如果鼠标点击多了就会造成很多选择框隐藏在界面中，吃内存。

```
def mousePressEvent(self, evt): #鼠标按下事件回调
    QMouseEvent
    self.rb = QRubberBand(QRubberBand.Rectangle, self) #创建橡皮选择框, Rectangle选择框为矩形
    self.originPos = evt.pos() #获取当前点下的坐标
    self.rb.setGeometry(QRect(self.originPos, QSize())) #可以传(self, int, int, int, int) 或者 QRect
    self.rb.show() #鼠标点击的时候显示
```

因为鼠标每次点下，都会再创建个新的选择框

修改方式如下：

```
class XWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("xxxxxxxx")
        self.resize(500, 500)
        #self.setupUi()
        self.rb = QRubberBand(QRubberBand.Rectangle, self) #将创建橡皮选择框放入初始化函数执行
    def disp(self):
        pass

    def mousePressEvent(self, evt): #鼠标按下事件回调
        QMouseEvent
        self.originPos = evt.pos() #获取当前点下的坐标
        self.rb.setGeometry(QRect(self.originPos, QSize())) #可以传(self, int, int, int, int) 或者 QRect
        self.rb.show() #鼠标点击的时候显示

    def mouseMoveEvent(self, evt): #鼠标移动事件回调
        self.rb.setGeometry(QRect(self.originPos, evt.pos()).normalized()) #支持负数反向拖动,
        #产生负数两点交换

    def mouseReleaseEvent(self, evt): #鼠标松开事件
        self.rb.hide() #鼠标松开的时候隐藏
```

选择框在界面初始化只创建一次

鼠标点击和释放都只是显示和隐藏选择框，而不是创建新的选择框

选择框，选择控件的方式逻辑如下：

```
def mouseReleaseEvent(self, evt): #鼠标松开事件
    self.rb.hide() #鼠标松开的时候隐藏
    rect = self.rb.geometry() #松开的时候获取矩形框勾画的范围
    #循环选择框下面的控件，看看控件坐标是不是在选择框范围内
    #如果控件坐标在选择框范围内，显示选中，如果不在选择框范围内，显示不选中
    #代码自行实现.....
```

## 弹出新对话框实现 QDialog

对象 = QDialog() #创建模态对话框

QDialog::exec() 阻塞程序，等待鼠标点击取消模态对话框，程序才向下执行

```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500, 500)

d = QDialog() #创建模态对话框
d.resize(100, 100)
d.exec() #程序阻塞在这里，只有点x取消Qdialog窗口才继续向下执行

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

1. 模态对话框



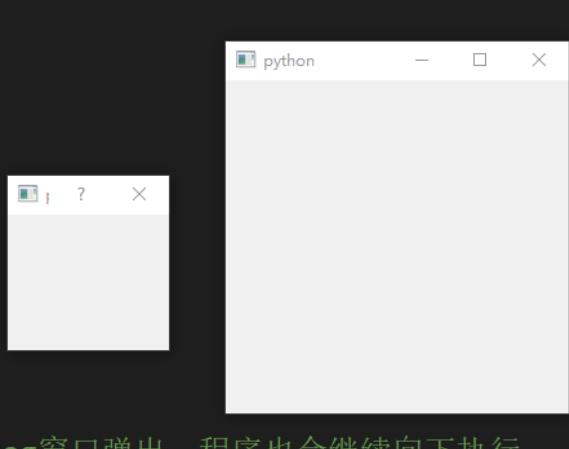
QDialog::open() 非阻塞，模态对话框和主窗口同时运行

```
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)
w = QWidget()
w.resize(500,500)

d = QDialog() #创建模态对话框
d.resize(100,100)
d.open() #程序非阻塞，就算Qdialog窗口弹出，程序也会继续向下执行

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

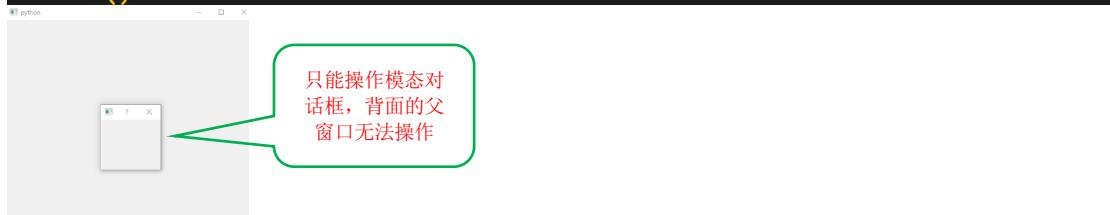


模态对话框嵌入父窗口

```
w = QWidget()
w.resize(500,500)

d = QDialog(w) #将模态对话框嵌入进父窗口，模态对话框没有点击取消之前，父窗口无法操作
d.resize(100,100)
d.open() #程序非阻塞

w.show()
```



QDialog::show() #就算模态对话框嵌入进父窗口，两个窗口都能分开操作

```
from PyQt5.Qt import *
import sys

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

d = QDialog(w) #将模态对话框嵌入进父窗口
d.resize(100,100)
d.show() #模态窗口和父窗口相互可以独立操作，模态窗口嵌入父窗口失效

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

```
QDialog::accept() #返回接收, 传递数字 1 给 exec()  
QDialog::reject() #返回拒绝, 传递数字 0 给 exec()  
QDialog::done(参数) #返回参数, 传递参数给 exec()
```

```
d = QDialog(w) #将模态对话框嵌入进父窗口  
d.resize(300,300)  
  
btn1 = QPushButton(d)  
btn1.setText("btn1")  
btn1.move(60,20)  
btn1.clicked.connect(lambda :d.accept())  
  
btn2 = QPushButton(d)  
btn2.setText("btn2")  
btn2.move(60,60)  
btn2.clicked.connect(lambda :d.reject())  
  
btn3 = QPushButton(d)  
btn3.setText("btn3")  
btn3.move(60,160)  
btn3.clicked.connect(lambda :d.done(8))  
  
result = d.exec() #阻塞方式可以返回点击accept或reject, done这三个其中一个的值  
print(result)
```

如果 btn1 按钮按下, 执行槽函数 accept, 模态对话框消失, 下面阻塞的 d.exec 返回 1

如果 btn2 按钮按下, 执行槽函数 reject, 模态对话框消失, 下面阻塞的 d.exec 返回 0

如果 btn3 按钮按下, 执行槽函数 done(8), 模态对话框消失, 下面阻塞的 d.exec 返回 8

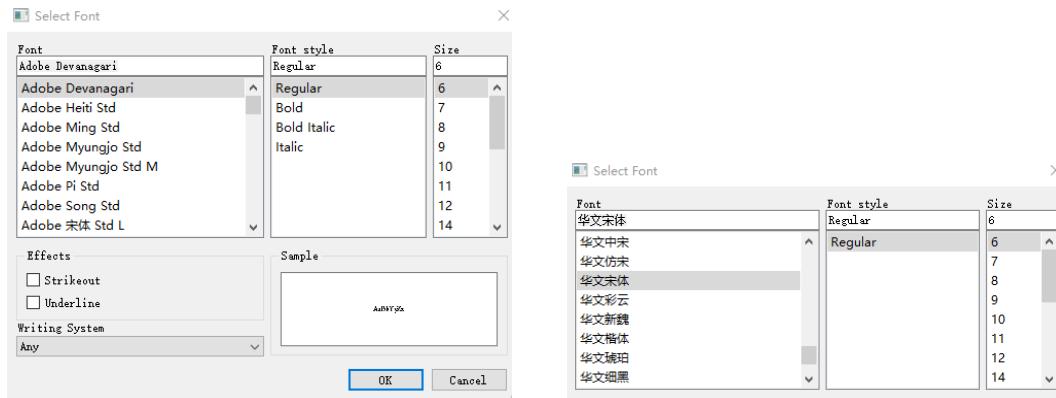


```
对象 = QFontDialog(父窗口) #字体选择模态对话框  
QFontDialog::show() #显示字体选择模态对话框
```

```
w = QWidget()  
w.resize(500,500)  
  
qf = QFontDialog(w) #创建字体选择模态对话框  
qf.show() #显示字体选择模态对话框
```

```
QFontDialog::exec() #阻塞字体模态对话框  
对象 = QFontDialog::selectedFont() #获取选择完的字体  
对象.family() #获取选择的是哪一类字体
```

```
w = QWidget()  
w.resize(500,500)  
  
qf = QFontDialog(w) #创建字体选择模态对话框  
qf.exec() #阻塞字体模态对话框，等选择完字体点击OK  
  
xfont = qf.selectedFont()#获取选择完的字体  
print(xfont) #选择完的字体生成的对象地址  
  
print(xfont.family()) #查看得到字体对象里面字体类型
```



```
<PyQt5.QtGui.QFont object at 0x00000241B9A5B208>  
华文宋体
```

## 模态对话框获取要打开的文件路径 QFileDialog

```
返回文件路径字符串 = QFileDialog.getOpenFileName(父对象, 对话框标题, 指定打开  
目录,(过滤字符串也),选择文件类型汇总, 默认选择文件) #获取选择的文件路径
```

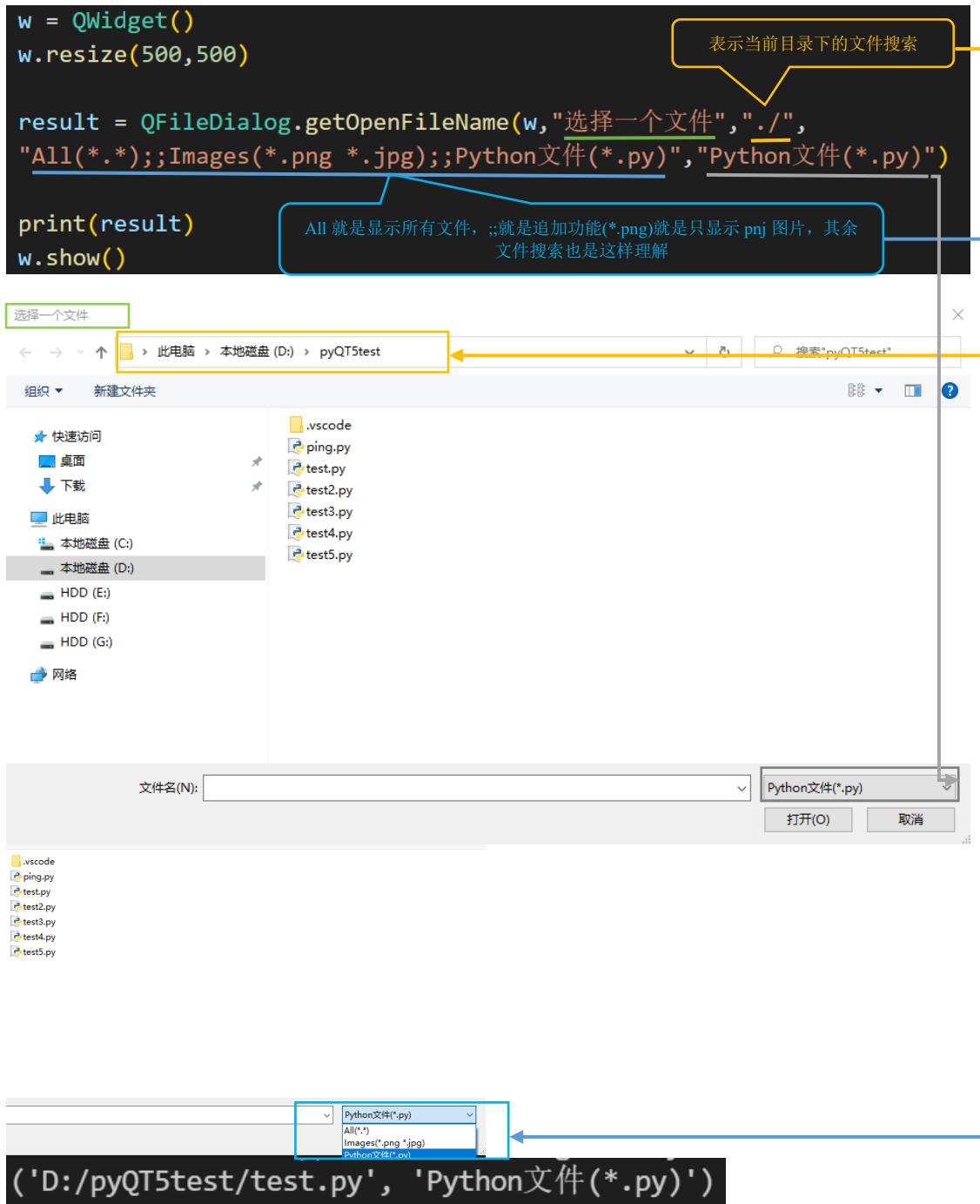
父对象： 嵌入的窗口

对话框标题： 弹出对话框左上角标题,填入字符串或者中文

指定打开目录： 就是弹出对话框显示哪一个目录下的文件，“./” 我们一般选择当前.py 文件  
下的目录

过滤字符串： 右下角就是显示哪一类文件如 png jpeg txt py

默认选择文件： 启动模态对话框右下角先显示哪一类文件



选择之后返回一个元组，第 1 个元素就是要打开的文件路径，然后将这个路径传给 open 就能打开文件了。

**QFileDialog.getOpenFileName 适合在按钮的槽函数中执行**

**QFileDialog.getSaveFileName(父对象, 对话框标题, 指定打开目录, (过滤字符串也), 选择文件类型汇总, 默认选择文件) #保存文件路径和打开文件参数是一样的**

```

w = QWidget()
w.resize(500,500)

result = QFileDialog.getSaveFileName(w,"选择一个文件","./",
"All(*.*);;Images(*.png *.jpg);;Python文件 (*.py)","Python文件 (*.py)")

print(result)
w.show()

```

( 'D:/pyQT5test/12345.py' , 'Python文件 (\*.py)' )

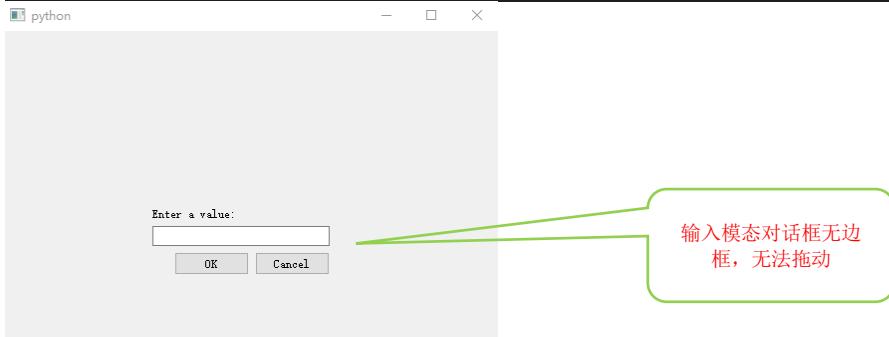
注意：这时候只是得到了保存文件的路径字符串，但是指定路径上没有 12345.py 文件，需要将路径字符串传递给 open 函数或者其它存储函数才行

要保存的文件路径得到了

## 输入模态对话框 QInputDialog

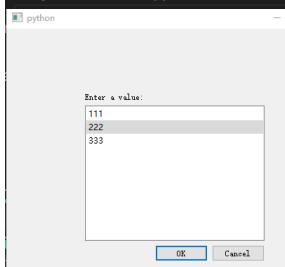
```
对象 = QInputDialog(父对象,参数) #创建模态对话框
```

```
input = QInputDialog(w,Qt.FramelessWindowHint) #创建输入模态对话框,  
#参数Qt.FramelessWindowHint对话框无边框  
input.show()
```



```
QInputDialog::setComboBoxItems(列表) #输入模态对话框增加条目，条目多少根据  
列表来算
```

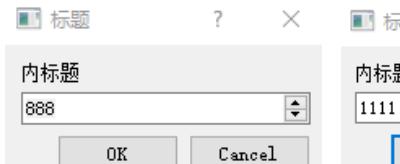
```
app = QApplication(sys.argv)  
  
w = QWidget()  
w.resize(500,500)  
  
input = QInputDialog(w,Qt.FramelessWindowHint) #创建输入模态对话框,  
#参数Qt.FramelessWindowHint对话框无边框  
input.setOption(QInputDialog.UseListViewForComboBoxItems)  
input.setComboBoxItems(["111","222","333"]) #增加输入模态对话框选择条目  
input.show()
```



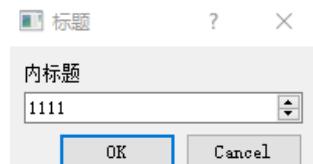
## 参数对话框

```
返回元组 = QInputDialog.getInt(父对象,外标题,内标题,默认值)
```

```
result = QInputDialog.getInt(w,"标题","内标题",888) #创建参数对话框  
print(result)
```



软件启动



点击 OK, 返回输入的值,True

(1111, True)



点击 cancel, 返回默认值, False

(888, False)

```
返回元组 = QInputDialog.getItem(父对象,外标题,内标题,列表)
```

```
result = QInputDialog.getItem(w,"标题","内标题",["1111","2222","3333"])
print(result)
```

```
返回 ('1111', True)
```



## QLabel 做展示数据使用

```
对象 = QLabel(默认显示字符串,父控件) #创建标签控件
```

```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)

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

我们发现字体永远是  
靠左边, 上下居中的



```
QLabel::setAlignment(对齐方式)
```

对齐方式: Qt.AlignRight #右对齐/垂直方向上对齐  
Qt.AlignVCenter #垂直方向居中

```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight) #右对齐
```



```
Qt.AlignVCenter #垂直方向居中
```

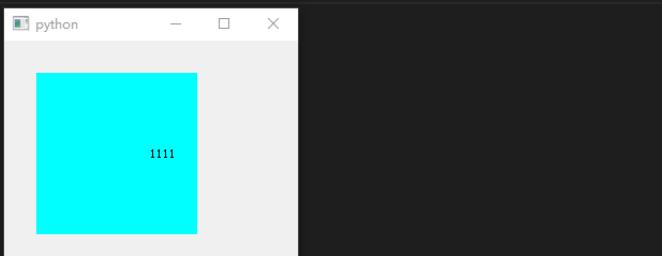
```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight | Qt.AlignVCenter) #右对齐 | 垂直方向居中

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



```
QLabel::setIndent(参数) #水平对齐方向保持相对间距设置，如果没有提起设置右对齐， 默认就是左对齐保持相对间距
```

```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色，亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight | Qt.AlignVCenter) #右对齐 | 垂直方向居中
label.setIndent(20) #水平对齐方向，相对距离保持20个像素
w.show()
sys.exit(app.exec_())
```



```
QLabel::setPixmap(父对象(可选), QPixmap 参数) #标签显示图片
```

QPixmap 与 QImage 的区别：

QPixmap 依赖于硬件， QImage 不依赖于硬件。

QPixmap 主要是用于绘图，针对屏幕显示而最佳化设计， QImage 主要是为图像 I/O、图片访问和像素修改而设计的。当图片小的情况下，直接用 QPixmap 进行加载，画图时无所谓，当图片大的时候如果直接用 QPixmap 进行加载，会占很大的内存，一般一张几十 K 的图片，用 QPixmap 加载进来会放大很多倍，所以一般图片大的情况下，用 QImage 进行加载，然后转乘 QPixmap 用户绘制。QPixmap 绘制效果是最好的。

QPixmap(传入图片路径字符串) #主要用于绘制图片



xzz.png  
ping.py  
test.py 应用程序跑的 test.py

```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色，亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight | Qt.AlignVCenter) #右对齐 | 垂直方向居中
label.setPixmap(QPixmap("xzz.png")) #当前py程序文件路径下的图片xzz.png
```



程序运行之后，发现 png 图片大于 QLabel 标签设置的长宽，显示不完

```
QLabel::adjustSize() #根据图片大小，自动调整标签控件长宽
```

```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色，亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight | Qt.AlignVCenter) #右对齐 | 垂直方向居中
label.setPixmap(QPixmap("xzz.png")) #当前py程序文件路径下的图片xzz.png
label.adjustSize() #根据图片大小，自动调整标签控件长宽
```



但是我不想这样，我想图片按照我标签 QLabel 控件的大小来缩放，就是图片服从控件尺寸

```
QLabel::setScaledContents(True) #图片服从标签尺寸
```

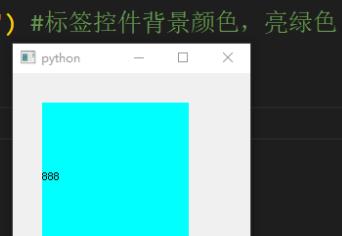
```
label = QLabel("1111",w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色，亮绿色
label.move(30,30)
label.resize(150,150)
label.setAlignment(Qt.AlignRight | Qt.AlignVCenter) #右对齐 | 垂直方向居中
label.setPixmap(QPixmap("xzz.png")) #当前py程序文件路径下的图片xzz.png
label.setScaledContents(True)
```



这就是符合标签尺寸自动修改图片大小。

```
QLabel::setNum(整形数/变量) #显示整形数据/浮点型数据
```

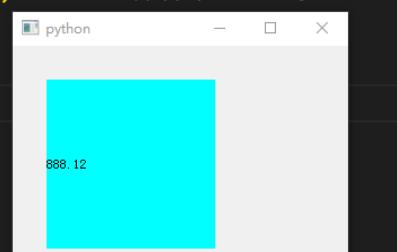
```
label = QLabel(w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色，亮绿色
label.move(30,30)
label.resize(150,150)
label.setNum(888) #传入整形变量/值，显示数据
w.show()
sys.exit(app.exec_())
```



```

label = QLabel(w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)
label.setNum(888.12) #支持小数显示/浮点型变量
w.show()
sys.exit(app.exec_())

```



得到动画对象 = QMovie(gif 动画路径字符串) #获取 gif 动画  
 QMovie::start() #启动动画, 不执行这句, gif 动画不会启动  
 QLabel::setMovie(传入 QMovie 动画对象) #在标签上显示 gif 动画

```

label = QLabel(w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)

movie = QMovie("xzzd.gif") #加载gif动画
label.setMovie(movie) #将gif动画赋值给标签显示
movie.start() #启动gif动画

```



感觉 gif 动画没有压缩到标签大小

其实和上页 setPixmap 一样的,gif 动画尺寸大于标签尺寸, 下面要放 gif 尺寸适应标签尺寸

```

label = QLabel(w) #创建label标签
label.setStyleSheet("background-color: cyan;") #标签控件背景颜色, 亮绿色
label.move(30,30)
label.resize(150,150)

movie = QMovie("xzzd.gif") #加载gif动画
label.setMovie(movie) #将gif动画赋值给标签显示
label.setScaledContents(True) #加入gif尺寸适应QLabel尺寸
movie.start() #启动gif动画

```



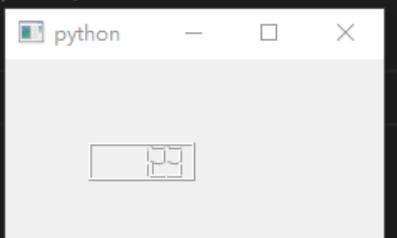
gif 尺寸符合标签尺寸

## 数码管显示 QLCDNumber

```
对象 = QLCDNumber(显示多少位数字(可选), 父对象) #创建数码管显示  
QLCDNumber::display(数值) #显示整数
```

```
lcd = QLCDNumber(w) #创建数码管显示  
lcd.move(50,50)  
lcd.display(123) #设置显示整数  
  
w.show()
```

我感觉数码管太小了

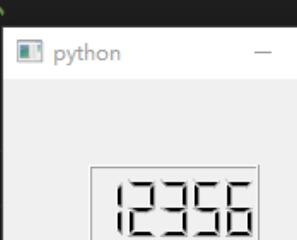


```
lcd = QLCDNumber(w) #创建数码管显示  
lcd.move(50,50)  
lcd.resize(100,50) #增大数码管尺寸  
lcd.display(123) #设置显示整数
```

数码管尺寸变大，更清楚

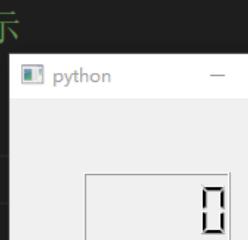


```
lcd = QLCDNumber(w) #创建数码管显示  
lcd.move(50,50)  
lcd.resize(100,50) #增大数码管尺寸  
lcd.display(12356) #设置显示整数
```



LCD 默认最大显示 5 位数

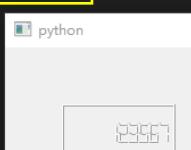
```
lcd = QLCDNumber(w) #创建数码管显示  
lcd.move(50,50)  
lcd.resize(100,50) #增大数码管尺寸  
lcd.display(123567) #设置显示整数
```



显示 7 位数就出问题了

QLCDNumber(显示多少位数字(设置), 父对象)

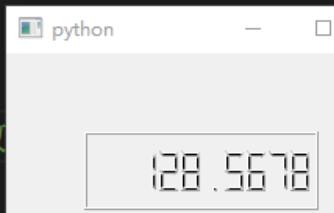
```
lcd = QLCDNumber([10,w]) #创建数码管显示  
lcd.move(50,50)  
lcd.resize(100,50) #增大数码管尺寸  
lcd.display(123567) #设置显示整数  
  
w.show()
```



创建 LCD 时设置最大显示数量

```
QLCDNumber::display(数值) #显示小数, 前提是显示数量位数够长
```

```
lcd = QLCDNumber(10,w) #创建数码管显示  
lcd.move(50,50)  
lcd.resize(150,50) #增大数码管尺寸  
lcd.display(128.5678) #也可以设置小数
```



## 进度条控件 QProgressBar

```
对象 = QProgressBar(父对象) #进度条创建
```

```
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
w.show()
```



进度条创建

```
QProgressBar::setMinimum(最小值) #设置进度条最小值  
QProgressBar::setMaximum(最大值) #设置进度条最大值  
QProgressBar::setValue(位置) #当前进度条显示的位置
```

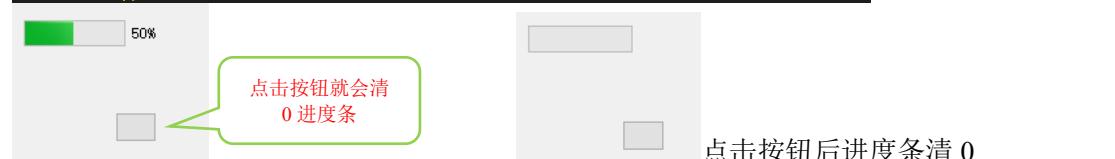
```
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置
```



```
QProgressBar::reset() #重置进度条, 也就是进度条清 0
```

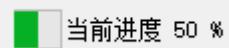
```
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
  
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置  
  
btn = QPushButton(w)  
btn.move(100,100)  
btn.clicked.connect(lambda :pb.reset()) #点击按钮, 重置进度条  
  
w.show()
```

记住, 进度条重置之后最小值是 -1



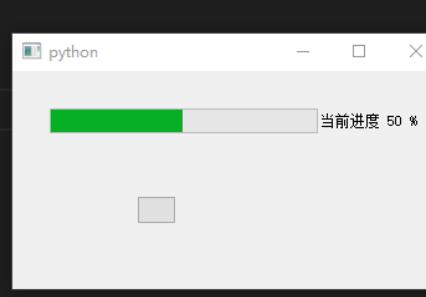
```
QProgressBar::setFormat(字符串传入) #给进度条设置中文提示,%v 显示进度条值
```

```
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
  
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置  
pb.setFormat("当前进度 %v %")
```



因为字符加入的原因，进度条尺寸被压缩了

```
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
pb.resize(300,20) #增大进度条尺寸  
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置  
pb.setFormat("当前进度 %v %")
```



增大进度条尺寸，解决问题。

```
QProgressBar::SetOrientation(参数) #设置进度条显示方向，默认水平  
参数: Qt.Vertical #垂直方向
```

```
w = QWidget()  
w.resize(500,500)  
  
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
pb.resize(20,300) #垂直方向显示进度条,尺寸长度减小,高度增加  
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置  
  
pb.setOrientation(Qt.Vertical) #设置进度条为垂直方向  
pb.setFormat("当前进度 %v %")
```

进度条虽然垂直了，但是没有字符显示

## 进度条实时显示

```
对象 = QTimer(父对象) #创建定时器  
QTimer::timeout.connect(传入槽函数) #定时时间到执行槽函数  
QTimer::start(定时时间) #启动定时器, 设置定时时间, 定时时间到执行  
timeout.connect 指定的槽函数
```

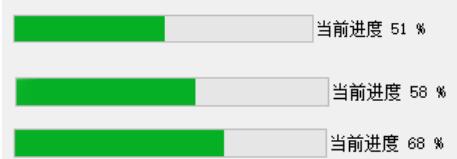
```
返回值 = QProgressBar::value() #获取进度条当前值
```

```
w = QWidget()  
w.resize(500,500)  
  
pb = QProgressBar(w) #创建进度条  
pb.move(30,30)  
pb.resize(300,20) #水平方向显示  
pb.setMinimum(0) #设置进度条最小值  
pb.setMaximum(100) #设置进度条最大值  
pb.setValue(50) #当前进度条显示50的位置  
pb.setFormat("当前进度 %v %")
```

```
timer = QTimer(pb) #创建定时器  
def cao():  
    pb.setValue(pb.value()+1) #进度条+1
```

```
timer.timeout.connect(cao) #定时时间到槽函数执行  
timer.start(1000) #启动定时器, 定时1秒溢出一次
```

```
w.show()
```



如果进度条累计到 100%了，防止进度条溢出，使用如下方法

```
def cao():  
    if pb.value() == pb.maximum(): #如果进度条累计满  
        timer.stop() #停止定时器  
    pb.setValue(pb.value()+1) #进度条+1
```

停止定时器工作，防止进度条溢出。

## 对话框控件 QErrorMessage，弹出一些警告，错误提示

对象 = QErrorMessage(父对象) #创建对话框，因为该对话框继承至 QDialog, 所以支持 exec, open, show 三种模态状态

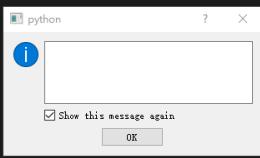
```
from PyQt5.QtWidgets import QComboBox, QDateTimeEdit, QDial, QDoubleSpinBox, QErrorMessage, QFile
import sys

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

em = QErrorMessage(w)
em.exec() #对话框控件继承至QDialog, 所以exec就是阻塞式弹出对话框

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

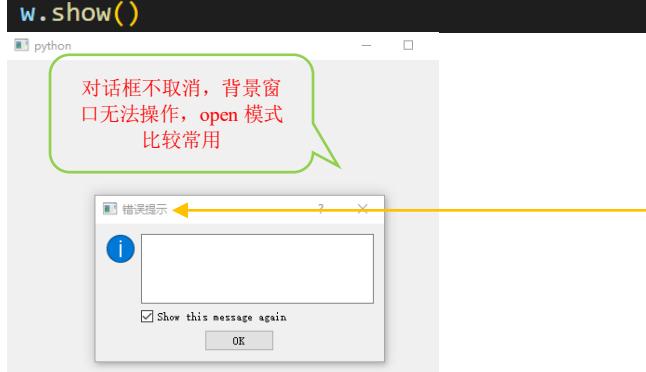


QErrorMessage::setTitle(传入显示的字符串) #对话框标题显示

```
w = QWidget()
w.resize(500,500)

em = QErrorMessage(w)
em.setWindowTitle("错误提示") ←
em.open() #对话框采用OPEN方式，程序向下执行，但是对话框不取消无法操作背景窗口

w.show()
```



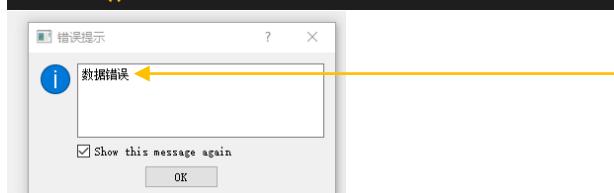
但是对话框不取消无法操作背景窗口

QErrorMessage::showMessage(传入显示的字符串) #对话框里面显示内容

```
w = QWidget()
w.resize(500,500)

em = QErrorMessage(w)
em.setWindowTitle("错误提示") #对话框标题显示内容
em.showMessage("数据错误") #对话框里面显示内容
em.open() #对话框采用OPEN方式，程序向下执行，但是对话框不取消无法操作背景窗口

w.show()
```



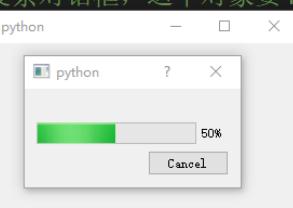
## 模态对话框进度条控件

```
对象 = QProgressDialog(父对象) #创建进度条对话框, 进度条对话框要 4 秒之后才显示, 奇怪  
QProgressDialog::setValue(进度值) #设置进度条对话框值
```

```
w = QWidget()  
w.resize(500,500)
```

```
pd = QProgressDialog(w) #创建进度条对话框, 这个对象要4秒之后才会显示, 很奇怪  
pd.setValue(50) #显示进度50%
```

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



QProgressDialog 创建的进度条对话框, 不需要 exec, show, open 去弹出显示, 进度条对话框会在 4 秒之后自动显示。这是因为 setMinimumDuration 函数设置了等待时间。

解决 4 秒之后显示的问题, 我要马上显示

```
QProgressDialog::setMinimumDuration(参数) #设置等待时间, 参数写 0 表示不等的
```

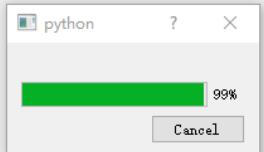
```
w = QWidget()  
w.resize(500,500)
```

```
pd = QProgressDialog(w) #创建进度条对话框, 这个对象要4秒之后才会显示, 很奇怪  
pd.setValue(50) #显示进度50%
```

```
pd.setMinimumDuration(0) #设置为马上显示
```

```
for i in range(1,100): #因为QProgressDialog是非阻塞, 所以for循环执行很快  
    pd.setValue(i)
```

```
w.show()
```

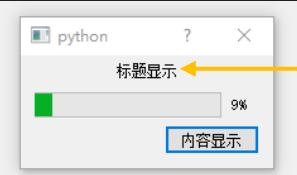


for 循环修改进度条参数, 马上触发进度条显示。

```
QProgressDialog(中间标题,取消按钮标题,最小值,最大值,父对象) #创建进度条对话框,设置内容
```

```
pd = QProgressDialog("标题显示","内容显示",1,1000,w) #创建进度条对话框,设置内容  
pd.setValue(50) #显示进度50%  
pd.setMinimumDuration(0) #设置为马上显示  
for i in range(1,100): #因为QProgressDialog是非阻塞, 所以for循环执行很快  
    pd.setValue(i)
```

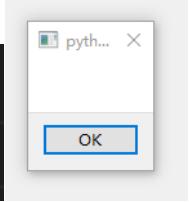
```
w.show()
```



## 消息弹出 QMessageBox

```
对象 = QMessageBox(显示图标, 显示标题, 显示内容, 显示按钮, 父对象) #消息对话框
```

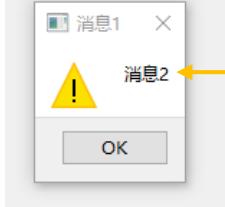
```
mb = QMessageBox(w) #创建消息框  
mb.show() #显示模态对话框
```



消息框显示多种内容

```
w = QWidget()  
w.resize(500,500)  
  
mb = QMessageBox(QMessageBox.Warning,"消息1","消息2",QMessageBox.Ok,w) #创建消息框  
mb.show() #显示模态对话框
```

```
w.show()
```



## 布局管理器

垂直布局管理器

```
对象 = QVBoxLayout() #创建垂直布局管理器  
窗口对象.setLayout(传入布局管理器对象) #将布局管理器嵌入进窗口
```

```
w = QWidget()  
w.resize(100,100)  
  
label1 = QLabel("标签1")  
label2 = QLabel("标签2")  
label3 = QLabel("标签3")  
  
vlayout = QVBoxLayout() #创建垂直布局管理器  
vlayout.addWidget(label1) #标签放入垂直布局管理器  
vlayout.addWidget(label2) #标签放入垂直布局管理器  
vlayout.addWidget(label3) #标签放入垂直布局管理器  
  
w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口  
  
w.show()
```



## 水平布局管理器

```
对象 = QBoxLayout() #创建水平布局管理器
```

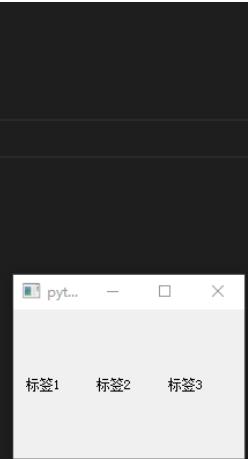
```
w = QWidget()
w.resize(100,100)

label1 = QLabel("标签1")
label2 = QLabel("标签2")
label3 = QLabel("标签3")

hlayout = QBoxLayout() #创建水平布局管理器
hlayout.addWidget(label1) #标签放入水平布局管理器
hlayout.addWidget(label2) #标签放入水平布局管理器
hlayout.addWidget(label3) #标签放入水平布局管理器

w.setLayout(hlayout) #将水平布局管理器模型放入父窗口

w.show()
```



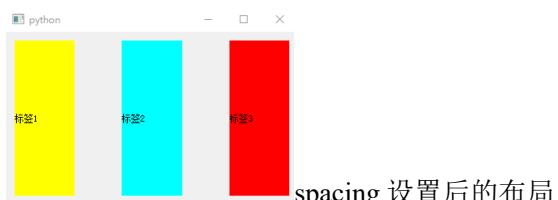
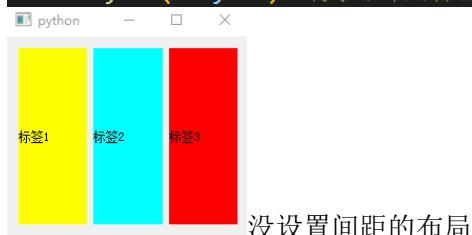
```
QBoxLayout::setSpacing(间距值) #设置布局管理间隔/垂直布局管理器间距同理
```

```
label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")

hlayout = QBoxLayout() #创建水平布局管理器
hlayout.addWidget(label1) #标签放入水平布局管理器

hlayout.addWidget(label2) #标签放入水平布局管理器
hlayout.addWidget(label3) #标签放入水平布局管理器
hlayout.setSpacing(60) #设置布局管理间隔

w.setLayout(hlayout) #将水平布局管理器模型放入父窗口
```



spacing 设置后的布局

## 布局嵌套布局

```
对象 = QVBoxLayout(布局方式) #盒子布局函数, 他有两个子类 QHBoxLayout 和  
QVBoxLayout, 它唯一的区别就是需要使用布局方式指定是垂直布局还是水平布局
```

```
布局方式: QVBoxLayout.LeftToRight #从左向右布局  
QVBoxLayout.RightToLeft #从右向左布局  
QVBoxLayout.TopToBottom #从上向下布局
```

```
QVBoxLayout::setLayout() #在垂直布局管理器窗口里面加入其它布局方式
```

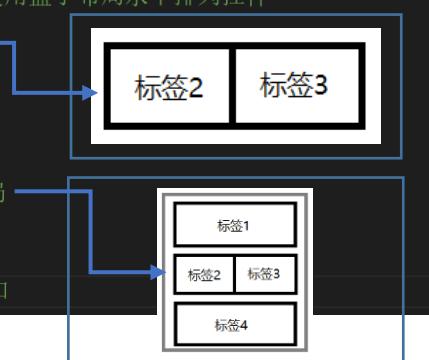
```
label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提现设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")
label4 = QLabel("标签4")
label4.setStyleSheet("background-color: green")

Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
Bxlayout.addWidget(label2) #加入盒子布局对象
Bxlayout.addWidget(label3) #加入盒子布局对象

vlayout = QVBoxLayout()
vlayout.addWidget(label1)
vlayout.addLayout(Bxlayout) #垂直布局中间是水平布局
vlayout.addWidget(label4)

w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```

从以上看, 就是 QHBoxLayout 盒子布局  
是寄生在垂直布局管理器里面的



### 布局管理器控件两边间距隔离

```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
Bxlayout.addWidget(label2) #加入盒子布局对象
Bxlayout.addWidget(label3) #加入盒子布局对象

vlayout = QVBoxLayout()
vlayout.addWidget(label1)
vlayout.addSpacing(100) #这样第1行label1和中间水平的两个标签就相隔100个像素间距
vlayout.addLayout(Bxlayout) #垂直布局中间是水平布局
vlayout.addWidget(label4)

w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```



### 两边挤压间距

```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
Bxlayout.addWidget(label2) #加入盒子布局对象
Bxlayout.addWidget(label3) #加入盒子布局对象

vlayout = QVBoxLayout()
vlayout.addWidget(label1)
vlayout.addSpacing(100) #这样第1行label1和中间水平的两个标签就相隔100个像素间距
vlayout.addLayout(Bxlayout) #垂直布局中间是水平布局
vlayout.addSpacing(50) #标签4与中间间距保持在50个像素
vlayout.addWidget(label4)
```



布局管理器指定某个控件比其它控件尺寸大，拉伸的时候大控件也按比例变大

```
QVBoxLayout::addWidget(控件对象,拉伸窗口比例) #拉伸窗口比例是按百分比来算的，如有3行控件,addWidget(控件对象,1) #表示该控件尺寸占窗口1倍  
addWidget(控件对象,2) #表示该控件尺寸为两倍控件大小  
addWidget(控件对象) #没有参数，表示该控件尺寸最小
```

```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
```

```
Bxlayout.addWidget(label2) #加入盒子布局对象  
Bxlayout.addWidget(label3) #加入盒子布局对象
```

```
vlayout = QVBoxLayout()  
vlayout.addWidget(label1,1)  
vlayout.addLayout(Bxlayout,2) #垂直布局中间是水平布局  
vlayout.addWidget(label4)
```

```
w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```

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



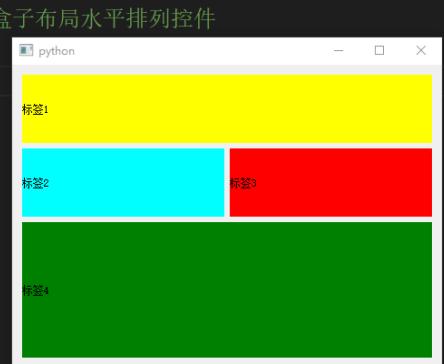
```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
```

```
Bxlayout.addWidget(label2) #加入盒子布局对象  
Bxlayout.addWidget(label3) #加入盒子布局对象
```

```
vlayout = QVBoxLayout()  
vlayout.addWidget(label1,1)  
vlayout.addLayout(Bxlayout,1) #垂直布局中间是水平布局  
vlayout.addWidget(label4,2)
```

```
w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```

```
w.show()
```



QVBoxLayout::addStretch(弹簧占比) #用弹簧来设置控件之间间隔，垂直布局和盒子布局也是一样的用法

```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
```

```
Bxlayout.addWidget(label2) #加入盒子布局对象  
Bxlayout.addWidget(label3) #加入盒子布局对象
```

```
vlayout = QVBoxLayout()  
vlayout.addWidget(label1,1)  
vlayout.addStretch(2) #弹簧宽度占2倍控件大小  
vlayout.addLayout(Bxlayout,1) #垂直布局中间是水平布局  
vlayout.addStretch(1) #弹簧宽度占1倍控件大小  
vlayout.addWidget(label4,1)
```

```
w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```



```
QVBoxLayout::SetStretchFactor(指定控件, 拉伸系数) #指定某个控件拉伸改变尺寸, 其余控件在拉伸的时候尺寸不变。
```

```
Bxlayout = QHBoxLayout(QBoxLayout.LeftToRight) #使用盒子布局水平排列控件
Bxlayout.addWidget(label2) #加入盒子布局对象
Bxlayout.addWidget(label3) #加入盒子布局对象

vlayout = QVBoxLayout()
vlayout.addWidget(label1)
vlayout.addStretch() #弹簧宽度占0倍控件大小
vlayout.addLayout(Bxlayout) #垂直布局中间是水平布局
vlayout.addStretch() #弹簧宽度占0倍控件大小
vlayout.addWidget(label4)
vlayout.setStretchFactor(label1,1) #拉伸的时候只修改指定控件尺寸, 其余控件尺寸不变

w.setLayout(vlayout) #将垂直布局管理器模型放入父窗口
```



## 表单布局管理器 QFormLayout

```
对象 = QFormLayout() #创建表单布局管理器
QFormLayout::addRow(控件对象 1, 控件对象 2) #第 1 种排列控件方式, 控件并列布局
```

```
w = QWidget()
w.resize(100,100)

label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")
label4 = QLabel("标签4")
label4.setStyleSheet("background-color: green")

flayout= QFormLayout() #创建表单布局管理器
flayout.addRow(label1,label2) #并列两个控件, 拉伸时第1个控件尺寸不变, 第2个控件尺寸改变

w.setLayout(flayout) #将表单布局管理器模型放入父窗口
w.show()
```



窗口拉伸值改变第 2 个控件尺寸

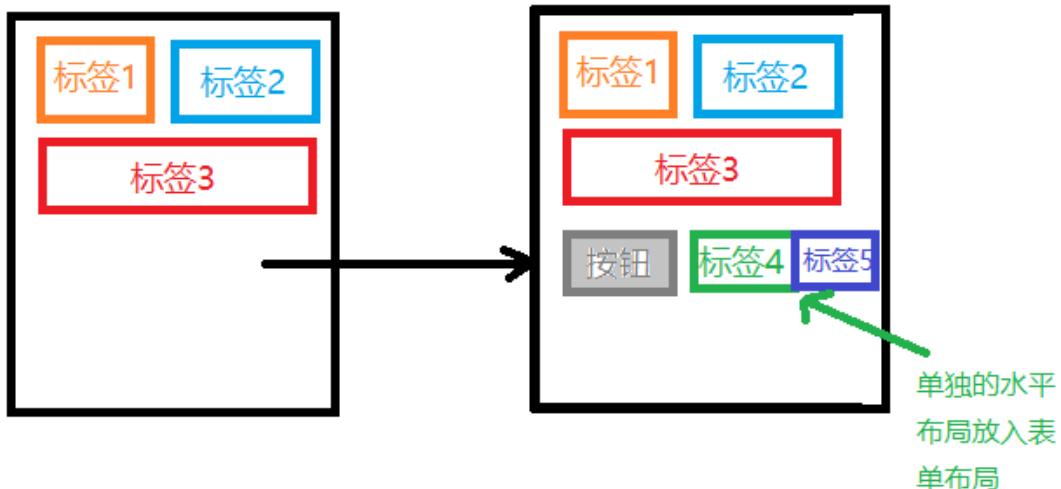
如果我突然想在第 2 排加 1 个控件

```
label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")
label4 = QLabel("标签4")
label4.setStyleSheet("background-color: green")

flayout= QFormLayout() #创建表单布局管理器
flayout.addRow(label1,label2) #并列两个控件, 拉伸时第1个控件尺寸不变, 第2个控件尺寸改变
flayout.addRow(label3) #如果我突然想再两个控件下加1个控件, 那么addRow直接再执行一次加1个控件就是

w.setLayout(flayout) #将表单布局管理器模型放入父窗口
```

## 表单布局内部加入水平布局



```
label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色，来提显设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")
label4 = QLabel("标签4")
label4.setStyleSheet("background-color: green")
label5 = QLabel("标签5")
label5.setStyleSheet("background-color: blue")

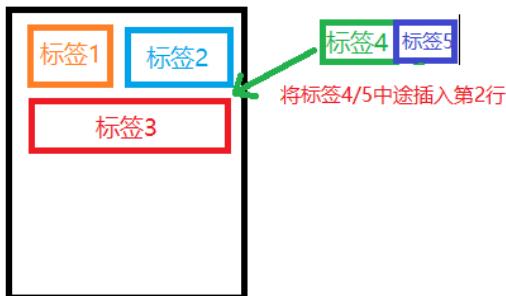
btn = QPushButton("按钮1")

hlyout = QHBoxLayout()
hlyout.addWidget(label4)
hlyout.addWidget(label5)

layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #并列两个控件，拉伸时第1个控件尺寸不变，第2个控件尺寸改变
layout.addRow(label3) #如果我突然想再两个控件下加1个控件，那么addRow直接再执行一次加1个控件就是
layout.addRow(btn,hlyout) #在第3行表单标签中加入水平标签
```



`QFormLayout::insertRow(插入某行,控件/布局对象) #在表单布局管理器某行插入新控件/布局对象`



```

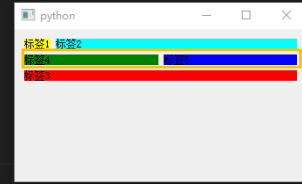
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")
label4 = QLabel("标签4")
label4.setStyleSheet("background-color: green")
label5 = QLabel("标签5")
label5.setStyleSheet("background-color: blue")

hlyout = QHBoxLayout()
hlyout.addWidget(label4)
hlyout.addWidget(label5)

layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #第1次addRow代表第0行
layout.addRow(label3) #如果我突然想再两个控件下加1个控件，那么addRow直接再执行一次加1个控件就是
layout.insertRow(1,hlyout) #插入布局/控件在指定的第1行 ←

w.setLayout(layout) #将表单布局管理器模型放入父窗口

```



`QFormLayout::rowCount()` #获取表单布局管理器行数

```

hlyout = QHBoxLayout()
hlyout.addWidget(label4)
hlyout.addWidget(label5)

layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #第1次addRow代表第0行
layout.addRow(label3) #如果我突然想再两个控件下加1个控件，那么addRow直接再执行一次加1个控件就是
layout.insertRow(1,hlyout) #插入布局/控件在指定的第1行
print(layout.rowCount()) #获取表单布局管理器里面有几行控件/布局对象，防止行数太多，越界

w.setLayout(layout) #将表单布局管理器模型放入父窗口
w.show()
sys.exit(app.exec_())

```



3

得到 3 行控件/布局对象

`QFormLayout::removeRow(传入布局对象/控件对象/行号)` #删除布局管理器的某行布局

```

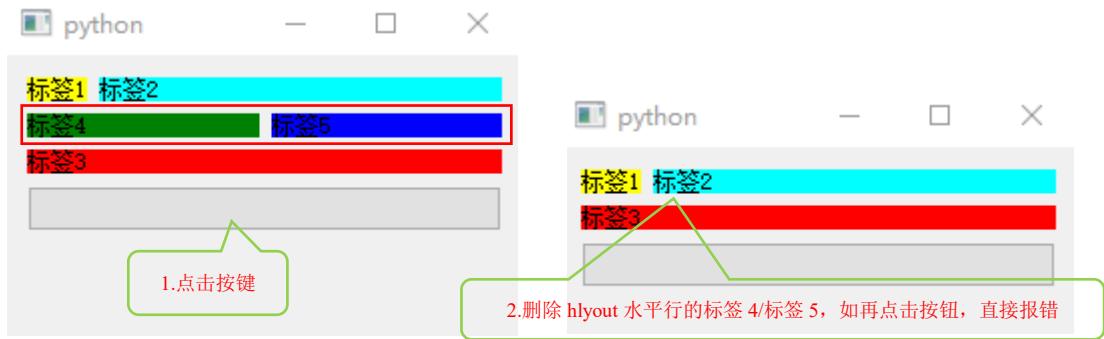
hlyout = QHBoxLayout()
hlyout.addWidget(label4)
hlyout.addWidget(label5)

layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #第1次addRow代表第0行
layout.addRow(label3) #如果我突然想再两个控件下加1个控件，那么addRow直接再执行一次加1个控件就是
layout.insertRow(1,hlyout) #插入布局/控件在指定的第1行
btn = QPushButton() #创建按钮加入表单布局，用于删除某行布局
layout.addRow(btn)
print(layout.rowCount()) #获取表单布局管理器里面有几行控件/布局对象，防止行数太多，越界

btn.clicked.connect(lambda :layout.removeRow(hlyout)) #删除水平平行布局

w.setLayout(layout) #将表单布局管理器模型放入父窗口
w.show()
sys.exit(app.exec_())

```



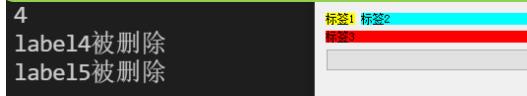
解决再次点击按钮删除指定行布局报错问题

```
QLabel::destroyed.connect() #实时监听标签控件是否被删除
```

```
layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #第1次addRow代表第0行
layout.addRow(label3) #如果我突然想再两个控件下加1个控件, 那么addRow直接再执行一次加1个控件就是
layout.insertRow(1,hlyout) #插入布局/控件在指定的第1行
btn = QPushButton() #创建按钮加入表单布局, 用于删除某行布局
layout.addRow(btn)
print(layout.rowCount()) #获取表单布局管理器里面有几行控件/布局对象, 防止行数太多, 越界
label4.destroyed.connect(lambda :print("label4被删除")) #监听控件是否被删除
label5.destroyed.connect(lambda :print("label5被删除"))

btn.clicked.connect(lambda :layout.removeRow(hlyout)) #删除水平行布局, 同时也删除了控件
```

如果第二次执行删除行布局按钮, 在槽函数加入标志位, 发现标志位不符合要求, 就不执行 removeRow 函数

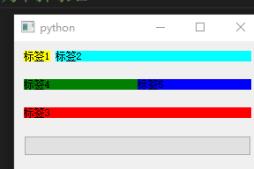


在控件监听槽函数事件中, 发现控件被删除, 下次就不执行 removeRow 函数就可以了。

```
QFormLayout::setVerticalSpacing(间距值) #表单布局里面的控件垂直间距设置
```

```
layout= QFormLayout() #创建表单布局管理器
layout.addRow(label1,label2) #第1次addRow代表第0行
layout.addRow(label3) #如果我突然想再两个控件下加1个控件, 那么addRow直接再执行一次加1个控件就是
layout.insertRow(1,hlyout) #插入布局/控件在指定的第1行
btn = QPushButton() #创建按钮加入表单布局, 用于删除某行布局
layout.addRow(btn)
print(layout.rowCount()) #获取表单布局管理器里面有几行控件/布局对象, 防止行数太多, 越界
layout.setVerticalSpacing(20) #设置表单控件之间垂直方向间距
```

```
w.setLayout(layout) #将表单布局管理器模型放入父窗口
w.show()
sys.exit(app.exec_())
```



```
QFormLayout::setHorizontalSpacing(间距值) #表单布局内部控件之间水平方向间距
```

```
layout.setVerticalSpacing(20) #设置表单控件之间垂直方向间距
layout.setHorizontalSpacing(50) #设置表单控件之间水平方向间距
```

```
w.setLayout(layout) #将表单布局管理器模型放入父窗口
w.show()
```



## 网格布局管理器 QGridLayout

```
对象 = QGridLayout() #创建网格布局管理器  
QGridLayout::addWidget(控件对象) #加载布局控件默认是垂直布局
```

```
w = QWidget()  
w.resize(100,100)  
  
label1 = QLabel("标签1")  
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔  
label2 = QLabel("标签2")  
label2.setStyleSheet("background-color: cyan")  
label3 = QLabel("标签3")  
label3.setStyleSheet("background-color: red")  
  
glayout = QGridLayout() #创建网格布局管理器  
  
glayout.addWidget(label1) #用addWidget 网格默认是垂直布局  
glayout.addWidget(label2)  
glayout.addWidget(label3)  
  
w.setLayout(glayout) #将表单布局管理器模型放入父窗口  
w.show()
```

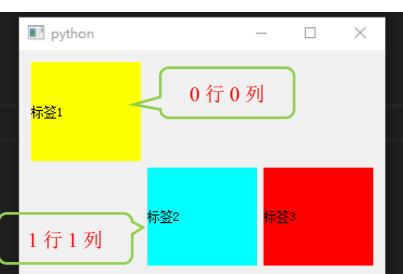
QGridLayout::addWidget(父对象(可以不填), 控件对象, 布局某行, 布局某列)

```
w = QWidget()  
w.resize(100,100)  
  
label1 = QLabel("标签1")  
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔  
label2 = QLabel("标签2")  
label2.setStyleSheet("background-color: cyan")  
label3 = QLabel("标签3")  
label3.setStyleSheet("background-color: red")  
  
glayout = QGridLayout() #创建网格布局管理器  
  
glayout.addWidget(label1,0,0) #标签1加入0行0列  
glayout.addWidget(label2,0,1) #标签2加入0行1列  
glayout.addWidget(label3,0,2) #标签3加入0行2列  
  
w.setLayout(glayout) #将表单布局管理器模型放入父窗口  
w.show()
```

标签全部都在 0 行, 当然是垂直布局效果了。

```
label3.setStyleSheet("background-color: red")
```

```
glayout = QGridLayout() #创建网格布局管理器  
  
glayout.addWidget(label1,0,0) #标签1加入0行0列  
glayout.addWidget(label2,1,1) #标签2加入1行1列  
glayout.addWidget(label3,1,2) #标签3加入1行2列
```



```
QGridLayout::addWidget(父对象(可以不填), 控件对象, 布局某行, 布局某列, 多占几行, 多占几列)
```

```
glayout = QGridLayout() #创建网格布局管理器  
glayout.addWidget(label1, 0, 0) #标签1加入0行0列  
glayout.addWidget(label2, 0, 1) #标签2加入0行1列  
glayout.addWidget(label3, 1, 0)
```



本来第1行第2列是没有的

```
glayout = QGridLayout() #创建网格布局管理器  
glayout.addWidget(label1, 0, 0) #标签1加入0行0列  
glayout.addWidget(label2, 0, 1) #标签2加入0行1列  
glayout.addWidget(label3, 1, 0, 1, 2) #标签3加入1行0列, 再多占第1行第2列
```



还可以扩展更多行更多列

```
label1 = QLabel("标签1")  
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提现设置布局间隔  
label2 = QLabel("标签2")  
label2.setStyleSheet("background-color: cyan")  
label3 = QLabel("标签3")  
label3.setStyleSheet("background-color: red")  
  
glayout = QGridLayout() #创建网格布局管理器  
  
glayout.addWidget(label1, 0, 0) #标签1加入0行0列  
glayout.addWidget(label2, 0, 1)  
glayout.addWidget(label3, 1, 0, 3, 3) #标签3加入1行0列, 再多占第3行第3列
```



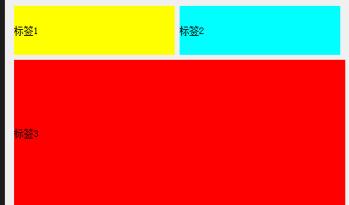
所以网格布局管理器可以多加 addWidget 布局很多行, 很多列。

```
QGridLayout::setColumnStretch(某列, 拉伸系数) #网格布局管理器控件列弹簧
```

```
glayout = QGridLayout() #创建网格布局管理器  
  
glayout.addWidget(label1, 0, 0) #标签1加入0行0列  
glayout.setColumnStretch(0, 1) #0行0列标签1拉伸系数占用很宽的列  
glayout.addWidget(label2, 0, 1) #标签2加入0行1列  
glayout.addWidget(label3, 1, 0, 3, 3) #标签3加入1行0列, 再多占第3行第3列
```



```
glayout.addWidget(label1, 0, 0) #标签1加入0行0列  
glayout.setColumnStretch(0, 1) #0行0列标签1拉伸系数占用很宽的列  
glayout.addWidget(label2, 0, 1) #标签2加入0行1列  
glayout.setColumnStretch(1, 1) #1列标签2拉伸系数占用很宽的列  
glayout.addWidget(label3, 1, 0, 3, 3) #标签3加入1行0列, 再多占第3行第3列
```



标签 1 占 1/2 拉伸系数, 标签 2 占 1/2 拉伸系数就均匀了

```

glayout.addWidget(label1,0,0) #标签1加入0行0列
glayout.setColumnStretch(0,1) #0列标签1拉伸系数占用很宽的列
glayout.addWidget(label2,0,1) #标签2加入0行1列
glayout.setColumnStretch(1,2) #1列标签2拉伸系数占用很宽的列
glayout.addWidget(label3,1,0,3,3) #标签3加入1行0列, 再多占第3行第3列

```



标签 1 占 1/3 拉伸系数, 标签 2 占 2/3 拉伸系数, 所以标签 2 要长一些

`QGridLayout::setRowStretch(某行, 拉伸系数) #设置网格布局某行控件高度伸缩系数`

```

glayout = QGridLayout() #创建网格布局管理器

glayout.addWidget(label1,0,0) #标签1加入0行0列
glayout.setColumnStretch(0,1) #0列标签1拉伸系数占用很宽的列
glayout.addWidget(label2,0,1) #标签2加入0行1列
glayout.setColumnStretch(1,2) #1列标签2拉伸系数占用很宽的列
glayout.addWidget(label3,1,0,3,3) #标签3加入1行0列, 再多占第3行第3列

glayout.setRowStretch(1,1) #1行拉伸系数1/1

```



`对象 = QStackedLayout(父对象<可以不填后面用 setLayout 载入>) #创建栈式布局管理器`

```

w = QWidget()
w.resize(100,100)

label1 = QLabel("标签1")
label1.setStyleSheet("background-color: yellow") #给标签加入颜色, 来提显设置布局间隔
label2 = QLabel("标签2")
label2.setStyleSheet("background-color: cyan")
label3 = QLabel("标签3")
label3.setStyleSheet("background-color: red")

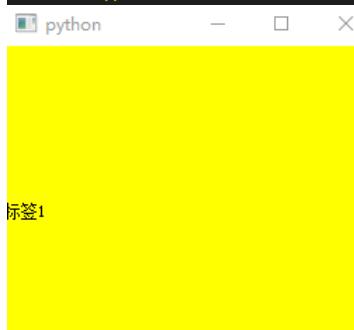
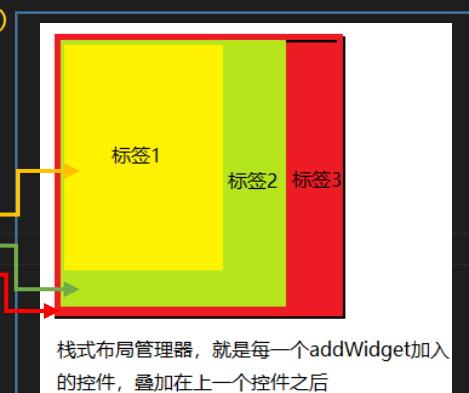
sl = QStackedLayout() #创建栈式布局管理器

sl.addWidget(label1) #将标签1加入栈式管理器
sl.addWidget(label2) #将标签2加入栈式管理器
sl.addWidget(label3) #将标签3加入栈式管理器

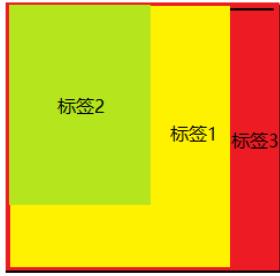
w.setLayout(sl) #将栈式模型放入父窗口

w.show()

```



执行的时候默认显示第 0 层, 标签 1



如果我想让第 2 层控件，标签 2 显示

```
QStackedLayout::setCurrentIndex(传入层号) #选择指定层控件
```

```
sl = QStackedLayout() #创建栈式布局管理器  
sl.addWidget(label1) #将标签1加入栈式管理器  
sl.addWidget(label2) #将标签2加入栈式管理器  
sl.addWidget(label3) #将标签3加入栈式管理器  
w.setLayout(sl) #将栈式模型放入父窗口  
sl.setCurrentIndex(1) #选择第2层(序号1)控件  
  
w.show()
```



```
QStackedLayout::currentChanged.connect(槽函数) #栈式布局管理器，界面切换，槽函数启动。
```

界面不管是手动还是自动切换，都会执行槽函数

```
QStackedLayout::currentChanged.connect(lambda val:print(val)) #界面切换的时候，切换到第几层了，会将当前切换层传递给槽函数 val
```

```
QStackedLayout::removeWidget(传入控件) #栈式布局得到的控件，指定某层/某控件移除
```

这里就不再实际演示了。

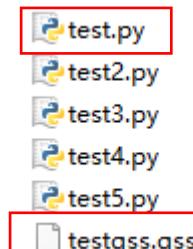
```
控件.setFixedSize(长度,宽度) #布局管理器不管怎么伸缩，设置最小的显示尺寸，该尺寸无法因为伸缩再缩小
```

```
w = QWidget()  
w.resize(400,400)  
  
label1 = QLabel("标签1")  
label1.setFixedSize(200,200) #控件最小尺寸，不受伸缩影响  
label1.setStyleSheet("background-color: yellow") #给标签加入颜色，来提显设置布局间隔  
label2 = QLabel("标签2")  
label2.setFixedSize(100,100) #控件最小尺寸，不受伸缩影响  
label2.setStyleSheet("background-color: cyan")  
label3 = QLabel("标签3")  
label3.setStyleSheet("background-color: red")  
  
ql = QVBoxLayout() #垂直布局管理器  
ql.addWidget(label1)  
ql.addWidget(label2)  
ql.addWidget(label3)  
w.setLayout(ql)
```



# QSS 使用

## 如何加载调用 QSS 文件



在主程序同级目录下创建 QSS 文件

```
test.py 2 testqss.qss
3 testqss.qss > QLabel
1 QLabel{
2     background-color: orange;
3 }
4
5 QPushButton{
6     background-color: red;
7 }
8
9
```

给标签控件和按钮控件加背景色

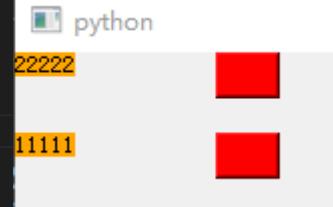
```
app = QApplication(sys.argv)
w = QWidget()
w.resize(400,400)

with open("testqss.qss","r") as xfile:
    textqss = xfile.read()      #读取QSS文件内容
    app.setStyleSheet(textqss)   #将QSS内容写入整个项目

L1 = QLabel(w)
L1.setText("11111")
L1.move(0,20)
L2 = QLabel(w)
L2.setText("22222")
L2.move(0,40)

b1 = QPushButton(w)
b1.move(100,0)
b2 = QPushButton(w)
b2.move(100,40)

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



QSS 文件加载成功。

我发现一种控件类型只能设置一种背景色，这是不对的。下面我们使用控件选择器来解决这个问题。

插入内容：如果 QSS 加载出现 gbk 错误

UnicodeDecodeError: 'gbk' codec can't decode byte 0xaf in position 49: illegal multibyte sequence

```
with open("testqss.qss","r",encoding = 'utf-8') as xfile:  
    xtext = xfile.read()#读取QSS文件内容  
    app.setStyleSheet(xtext)#将QSS内容写入整个项目
```

那就在 open qss 文件的时候，强制指定编码方式 encoding = 'utf-8'

问题得到解决

### III 选择器使用

控件大名 #创建的对象 ID{.....}

如下：

```
QPushButton#b1 {  
    background-color: red;  
}  
} 
```

b1 是创建某个按钮时，给某个按钮设置的 ID

设置 b1 这个按钮为红色背景

```
b1 = QPushButton(w)  
b1.move(100,0)  
b2 = QPushButton(w)  
b2.move(100,40)
```

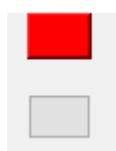


按钮背景色不见了？

这是因为我们 QSS 只使用了按钮的 ID，但是在对象中没有设置按钮 ID

控件.setObjectName(设置 ID 字符串) #设置控件 ID，方便 QSS 单独设置样式

```
b1 = QPushButton(w)  
b1.move(100,0)  
b1.setObjectName("b1") #设置b1按钮ID  
b2 = QPushButton(w)  
b2.move(100,40)
```



```
label1 = QLabel(w)  
label1.setText("11111")  
label1.move(0,10)  
label1.setObjectName("label1")#设置label1标签ID  
L2 = QLabel(w)  
L2.setText("22222")  
L2.move(0,40)  
L2.setObjectName("L2") #设置L2标签ID  
  
b1 = QPushButton(w)  
b1.move(100,0)  
b1.setObjectName("b1") #设置b1按钮ID  
b2 = QPushButton(w)  
b2.move(100,40)  
b2.setObjectName("BT2")#设置b2按钮ID(ID名不一定非要和对象一样)
```

```
testqss.qss x  
+---+  
+ QPushbutton#b1  
+ QLabel#label1 {  
+     background-color: orange;  
+ }  
+ QLabel#L2 {  
+     background-color: rgb(200, 255, 0);  
+ }  
+ QPushbutton#b1 {  
+     background-color: red;  
+ }  
+ QPushbutton#BT2 {  
+     background-color: rgb(255, 102, 0);  
+ }
```

## 伪状态使用

控件#选择器:hover{...} #鼠标移动到控件执行



```
label1 = QLabel(w)
label1.setText("11111")
label1.move(0,10)
label1.setObjectName("label1")#设置label1标签ID
```

鼠标移动前

```
QLabel#label1:hover {
    background-color: orange;
}
```

鼠标移动到标签后

## 通配符选择器

\*{...} #让所有控件的样式都执行\*号里面的内容



```
label1 = QLabel(w)
label1.setText("11111")
label1.move(0,10)
label1.setObjectName("label1")#设置label1标签ID
```

```
btn1 = QPushButton(w)
btn1.move(100,0)
btn1.setObjectName("btn1") #设置b1按钮ID
```

```
background-color: orange;
```

所有控件的样式都变一样了

## 类选择器

点 . 控件 {...} #类选择器，比如控件创建的子类，子类控件就不会被样式修改

如 QLabel 控件，只修改 QLabel 控件的样式，如果用 QLabel 建立了子类控件，这个子类控件样式不会被 QLabel 样式修改



```
label1 = QLabel(w)
label1.setText("11111")
label1.move(0,10)
```

```
.QLabel{
    background-color: orange;
}
```

所有控件的样式都变一样了

image(传入图片路径) #加载图片



```
QLabel{
    image:url("xzz.png") /*在QSS文件路径下找到xzz.png图片进行加载*/
}
```

```
label1 = QLabel(w)
label1.setText("11111")
label1.move(0,10)
```

尺寸太小了，图片显示不完

如果图片在其它路径，写法就是 image:url(..//xzz/xzz.png) #这就是跳转到 qss 文件上一级目录，然后进入 xzz 目录，加载图片。

设置图片长宽

```
label1 = QLabel(w)
label1.setText("11111")
label1.resize(100,100) #我先设置标签大小
label1.move(0,10)
```

先设置标签大小

```
QLabel{
    image:url("xzz.png"); /*加入分号，才能写下边的内容*/
    width: 100px;
    height: 100px;
}
```

在设置图片长宽



设置成功

### 子控件选择器

复选框，切换选择状态，图片显示 QCheckBox

控件::indicator{...} #指示器设置，一般设置些默认参数，只支持 QCheckBox,QRadioButton

控件::indicator:checked{...} /\*控件选中执行的样式内容\*/

控件::indicator:unchecked{...} /\*控件选中被取消样式内容\*/

```
cb = QCheckBox("11111",w) #创建勾选框
cb.move(100,100)
cb.resize(100,100)
```

```
QCheckBox::indicator{
    width: 100;
    height: 100;
}

QCheckBox::indicator:checked{ /*CheckBox被勾选执行*/
    image:url(xzz.png);
}

QCheckBox::indicator:unchecked{ /*CheckBox取消勾选执行*/
    image:url("xzz22.png");
}
```



勾选框没选中， 默认显示 unchecked

不管勾选没勾选，显示尺寸都是一样的，类似这些参数就可以先让指示器默认设置

勾选显示得图片，没有”冒号也可以导入

取消勾选显示得图片

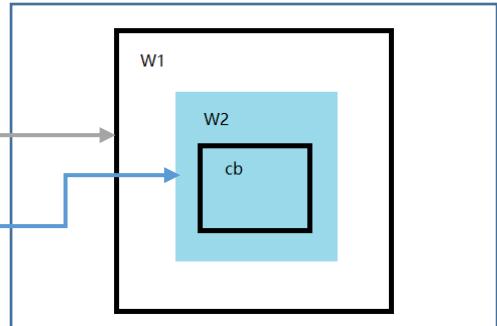


勾选选择执行 checked

## 子选择器

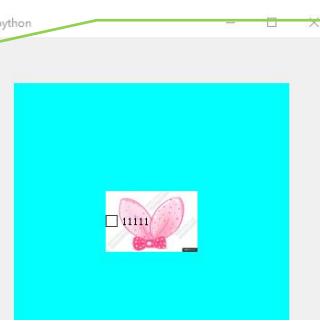
控件#选择器>子控件{.....}

```
w2 = QWidget(w) #创建小窗口  
w2.move(50,50)  
w2.resize(300,300)  
w2.setObjectName("w2")  
  
cb = QCheckBox("11111",w2) #创建勾选框  
cb.move(100,100)  
cb.resize(100,100)
```



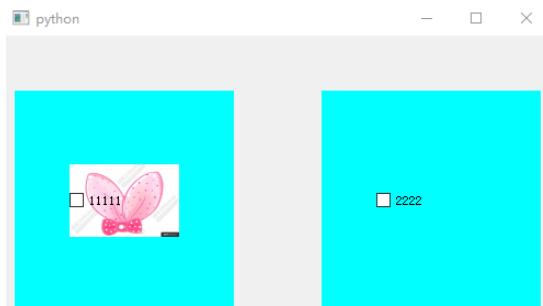
```
QWidget#w2{  
    background-color: cyan;  
}  
  
QWidget#w2>QCheckBox{  
    width: 100px;  
    height: 100px;  
    image:url(xzz.png);  
}
```

> 指定, 只修改 w2 窗口下 QCheckBox 的样  
式, 其余窗口的 QCheckBox 样式不改



这就是子选择器的用处，批量修改指定对象之下的控件。

```
QWidget#w2{  
    background-color: cyan;  
}  
  
QWidget#w2>QCheckBox{  
    width: 100px;  
    height: 100px;  
    image:url(xzz.png);  
}  
  
QWidget#w3{  
    background-color: cyan;  
}  
  
w2 = QWidget(w) #创建小窗口1  
w2.move(10,50)  
w2.resize(200,200)  
w2.setObjectName("w2")  
  
w3 = QWidget(w) #创建小窗口2  
w3.move(290,50)  
w3.resize(200,200)  
w3.setObjectName("w3")  
  
cb = QCheckBox("11111",w2) #创建勾选框1  
cb.move(50,50)  
cb.resize(100,100)  
  
cb2 = QCheckBox("2222",w3) #创建勾选框2  
cb2.move(50,50)  
cb2.resize(100,100)
```



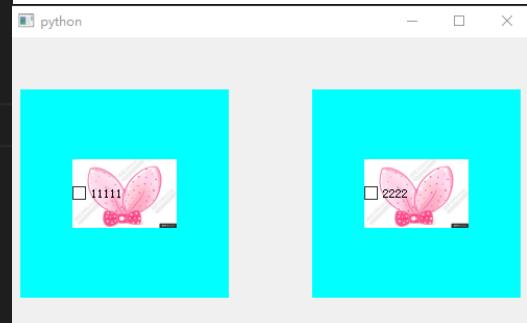
如果我想 w2,w3 窗口下的 QCheckBox 都修改同样的样式呢?

#控件,#控件,#控件{....} #多控件选择, 对多控件进行样式修改, 修改多少个控件, 就用多少个 ‘,’逗号隔离开

```
QWidget#w2{
    background-color: cyan;
}

QWidget#w3{
    background-color: cyan;
}

#w2>QCheckBox,#w3>QCheckBox{
    width: 100px;
    height: 100px;
    image:url(xzz.png);
}
```



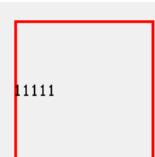
用伪状态给控件加边框

border #创建矩形边框  
solid 实线, dotted 虚线

```
QLabel:hover{
    border:2px solid red; /*border 加入边框, 边框线宽2px, solid实心线, 红色线*/
}
```

```
label1 = QLabel("1111",w) #创建标签
label1.move(20,20)
label1.resize(100,100)
```

1111



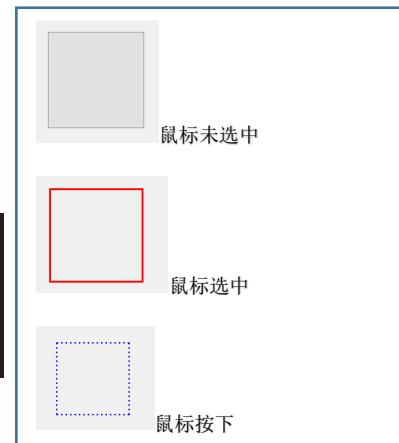
实心边框尺寸默认是根据控件 resize 来的。注意因为控件是标签, 所以 pressed 鼠标按下状态不支持。

控件:pressed{....} #鼠标按下伪状态

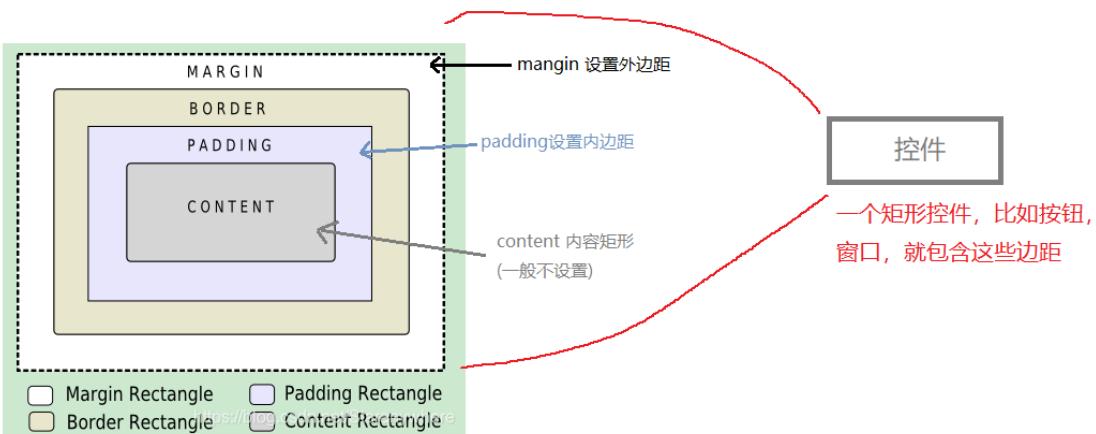
```
QPushButton:hover{
    border:2px solid red;
}

QPushButton:pressed{
    border:2px dotted red;
}

btn1 = QPushButton(w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
```



## 盒子模型



`border-width: 填入边框线宽`

`border-style: 上边框样式 右边框样式 下边框样式 左边框样式 #填入边框线的类型`



设置上右下左, 每条边框线



`border-top-style: 设置上边框样式 # border-style 一般用于同一设置, 而 border-top-style 只设置上边框`

```
QLabel{  
    background-color: cyan;  
    border-width: 20px; /*设置边框线宽*/  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-top-style: groove;  
}
```



`border-top-style: none;` #none 为无边框设置

```
QLabel{  
    background-color: cyan;  
    border-width: 20px; /*设置边框线宽*/  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-top-style: none;  
}
```



`border-width: 上边宽度 右边宽度 下边宽度 左边宽度` #设置边框各边宽度

```
QLabel{  
    background-color: cyan;  
    border-width: 2px 4px 8px 30px; /*上边宽度 右边宽度 下边宽度 左边宽度 */  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
}
```



`border-bottom-width: 下边宽度` #单独设置边框下边宽度

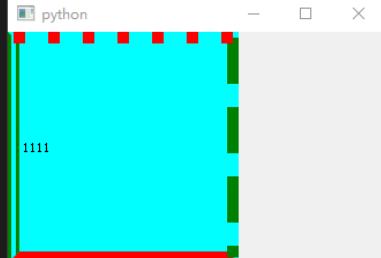
其它几个边单独设置请查阅网上资料，这里不再描述

```
QLabel{  
    background-color: cyan;  
    border-width: 2px 4px 8px 30px; /*上边宽度 右边宽度 下边宽度 左边宽度 */  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-bottom-width: 100; /*可以单独设置边框下边宽度*/  
}
```



```
border-color: 颜色1 颜色2 #设置边框对边颜色
```

```
QLabel{  
    background-color: cyan;  
    border-width: 10px;  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-color: red green; /*两个参数，设置边框对边颜色*/  
}
```



```
border-color: 上边颜色 右边颜色 下边颜色 左边颜色 #设置边框4边颜色
```

```
QLabel{  
    background-color: cyan;  
    border-width: 10px;  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-color: red green black orange; /*上 右 下 左 颜色*/  
}
```



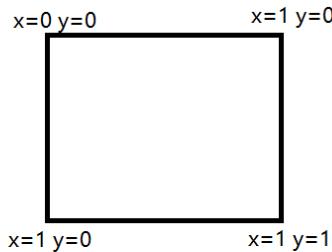
```
border-left-color: 颜色 #单独设置边框左边颜色
```

```
QLabel{  
    background-color: cyan;  
    border-width: 10px;  
    border-style: dotted dashed solid double; /*点状 虚线 实线 双线*/  
    border-color: red green black orange; /*上 右 下 左 颜色*/  
    border-left-color: #rgb(0, 140, 255); /*单独设置左边颜色*/  
}
```

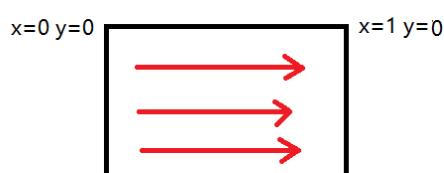


## 颜色渐变

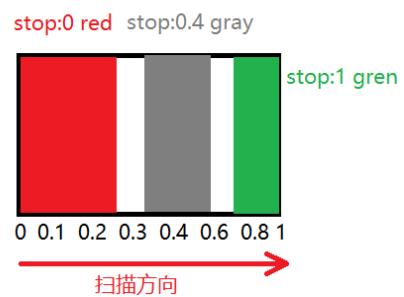
`qlineargradient(扫描的 x 起始位置, 扫描的 y 起始位置, 扫描结束的 x 位置, 扫描结束的 y 位置, stop:扫描起始位置 颜色, stop:扫描中间位置 颜色, stop:扫描结束位置 颜色) #颜色渐变设置`



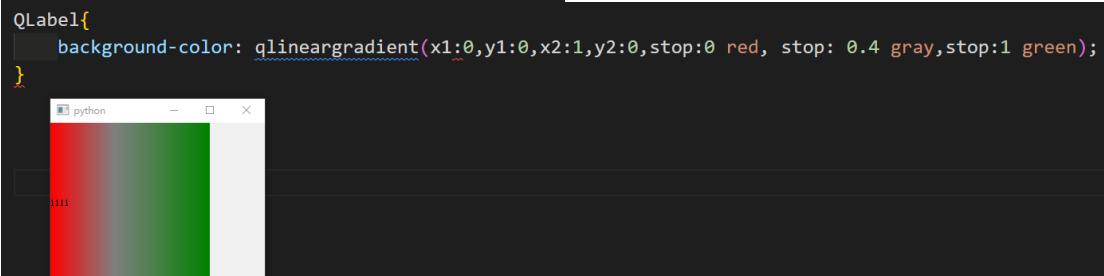
如果我从左向右扫描



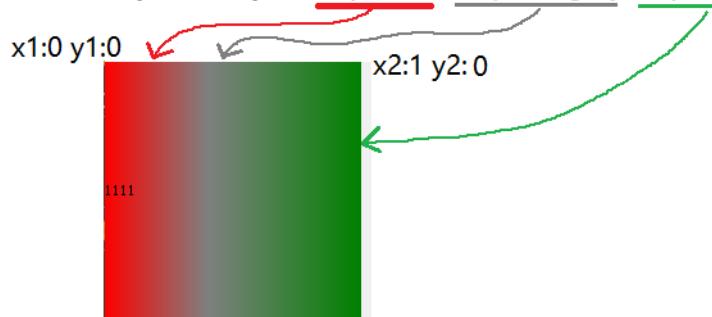
如果我还要加入颜色



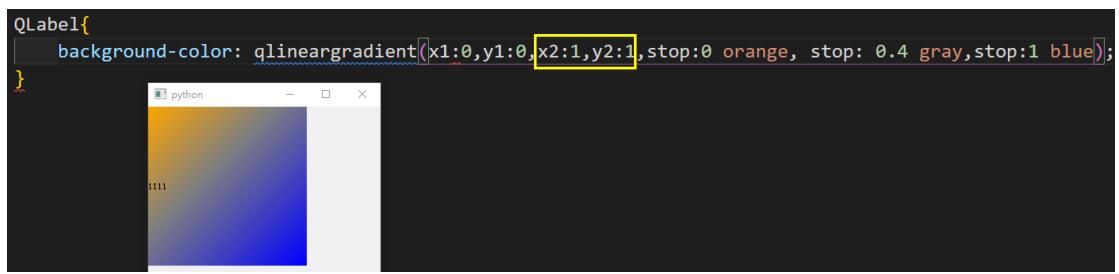
```
label = QLabel("1111", w)
label.resize(200, 200)
```



```
qlineargradient(x1:0,y1:0,x2:1,y2:0,stop:0 red, stop: 0.4 gray,stop:1 green);
```



颜色从红色经过灰色过渡到绿色

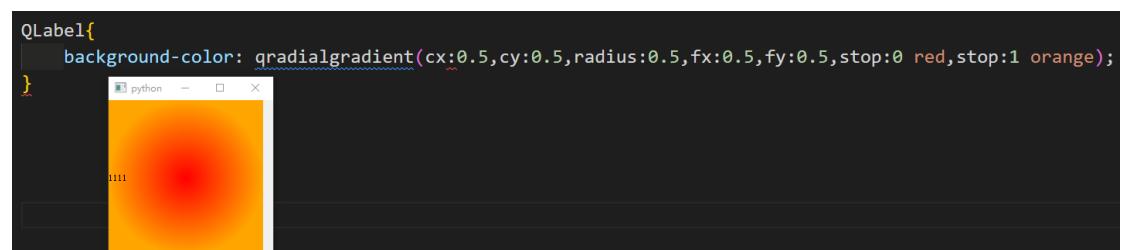
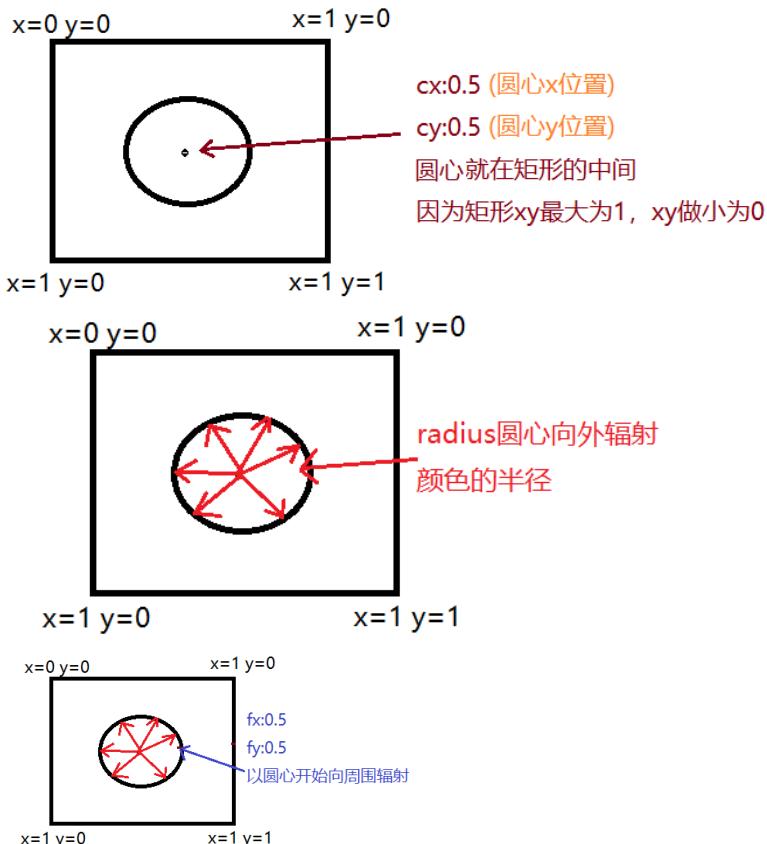


向右下角扫描

## 辐射渐变

从控件中间某一点,向四周 360 度辐射颜色

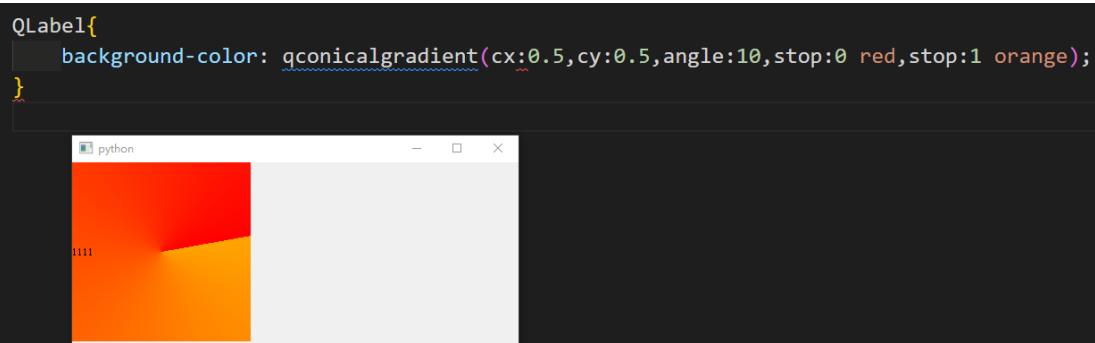
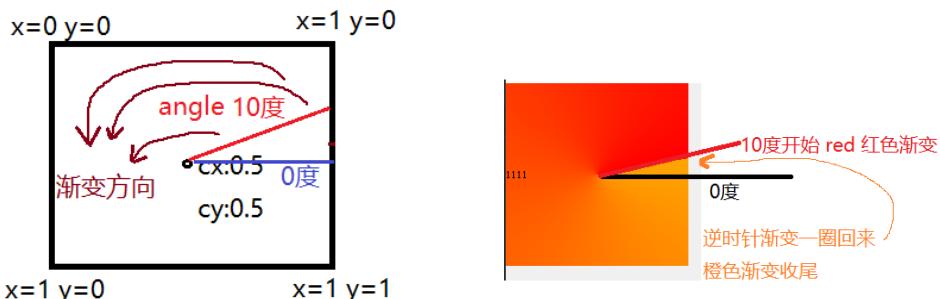
```
qradialgradient(圆心 x 位置,圆心 y 位置,radius:圆心向外辐射的半径,光源辐射方向  
x 位置,光源辐射方向 y 位置,起始颜色,结束颜色); #圆形辐射渐变
```



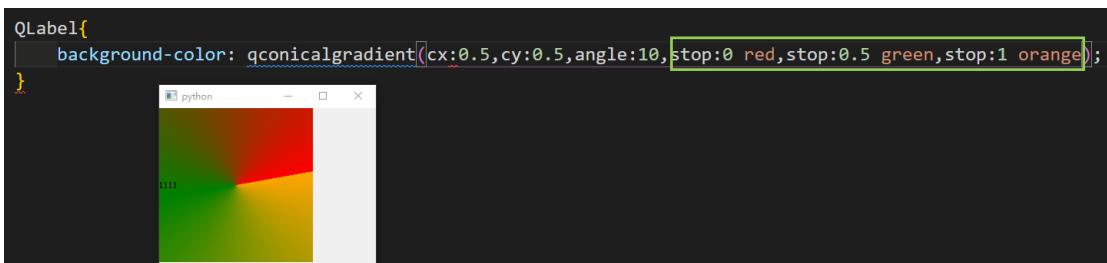


## 角度渐变

```
QLabel{
    background-color: qconicalgradient(cx:0.5, cy:0.5, angle:10, stop:0 red, stop:1 orange); #角度渐变
```



角度颜色渐变实验成功。



三段颜色渐变

```
border-image:url(传入图片路径) #加载图片
```

```
QLabel{
    background-color: cyan; /*背景色*/
    border-image: url("xzz.png");/*图片将覆盖背景色*/
}
```

图片覆盖了背景颜色



```
background-image:url(传入图片路径) #设置图片到背景显示
```



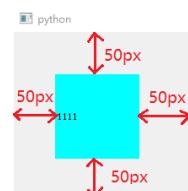
```
margin:矩形缩放多少个像素 #外边距就是让控件向内缩放
```

```
QLabel{  
    background-color: cyan; /*背景色*/  
}  
  
label = QLabel("1111",w)  
label.resize(200,200)
```

记住现在控件是 resize 设置的大小

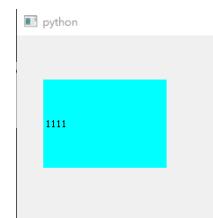


```
QLabel{  
    background-color: cyan; /*背景色*/  
    margin:50px; /*外边距内缩50个像素*/  
}
```



你看，控件是向内缩放 50 个像素

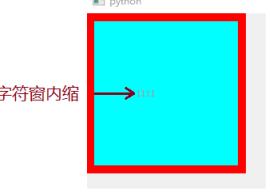
```
QLabel{  
    background-color: cyan; /*背景色*/  
    margin:50px 30px ; /*外边距对边缩放*/  
}
```



外边距和 border 设置方式一样，只不过是内缩而已，我这里就不一一演示了。

```
padding: 前景控件内缩距离 #内边距设置
```

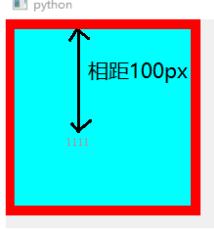
```
QLabel{  
    color: #rosybrown; /*前景色设置, QLabel就是字符的颜色*/  
    background-color: cyan; /*背景色*/  
    border: 10px solid red;  
    padding: 50px; /*内边距缩放*/  
}
```



内边距就是让前景色的矩形框内缩

**padding-top:** 前景控件内缩相差顶部距离 #内边距顶部距离设置

```
QLabel{  
    color: #rosybrown; /*前景色设置, QLabel就是字符的颜色*/  
    background-color: cyan; /*背景色*/  
    border: 10px solid red;  
    padding: 50px; /*内边距缩放*/  
    padding-top: 100px; /*前景字符距离顶边100px*/  
}
```

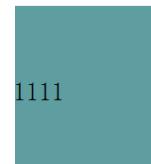


内边距就是设置控件内部字符或者前景色的位置。

设置字体

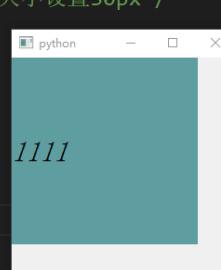
**font-size:** 字体大小 px #设置字体大小

```
label = QLabel("1111", w)  
label.resize(200, 200)  
  
QLabel{  
    background-color: #cadetblue;  
    font-size: 30px; /*字体大小设置30px*/  
}
```



**font-style:** italic; #字体为斜体

```
QLabel{  
    background-color: #cadetblue;  
    font-size: 30px; /*字体大小设置30px*/  
    font-style: italic;  
}
```

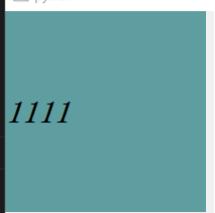


**font-weight:** 100~900; #设置字体粗细

```
QLabel{  
    background-color: #cadetblue;  
    font-size: 30px; /*字体大小设置30px*/  
    font-style: italic;  
    font-weight: 100; /*设置字体粗细*/  
}
```



```
QLabel{  
    background-color: #cadetblue;  
    font-size: 30px; /*字体大小设置30px*/  
    font-style: italic;  
    font-weight: 900; /*设置字体粗细*/  
}
```

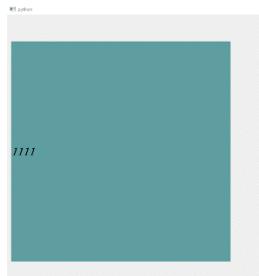


针对布局管理器的控件，在 QSS 设置控件尺寸最大和最小。

```
min-width: 控件最小宽度 #针对布局管理器，限制控件尺寸  
min-height: 控件最小高度  
max-width: 控件最大宽度  
max-height: 控件最大高度
```

```
QLabel{  
    background-color: #cadetblue;  
    font-size: 30px; /*字体大小设置30px*/  
    font-style: italic;  
    font-weight: 900; /*设置字体粗细*/  
  
    min-width: 200px; /*针对布局管理器，设置QLabel标签创建的对象，控件尺寸最小缩放到200*/  
    min-height: 200px; /*针对布局管理器，设置QLabel标签创建的对象，控件尺寸最小缩放到200*/  
    max-width: 600px; /*针对布局管理器，设置QLabel标签创建的对象，控件尺寸最大缩放到600*/  
    max-height: 600px; /*针对布局管理器，设置QLabel标签创建的对象，控件尺寸最大缩放到600*/  
}
```

控件拉伸到最小状态

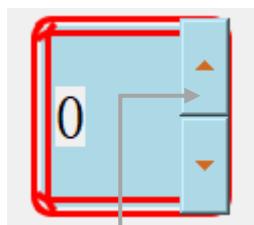


控件拉伸到最大状态

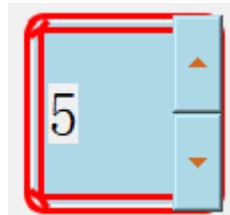


## QSpinBox 控件 QSS 设置

```
sb = QSpinBox(w)  
sb.move(50, 50)  
sb.resize(100, 100)
```



```
QSpinBox{  
    font-size: 30px; /*字体大小*/  
    color: #chocolate; /*字体颜色*/  
    border: 10px double #red; /*线宽 双线 红色*/  
    border-radius: 10px; /*设置控件四角圆弧*/  
    background-color: #lightblue; /*背景色*/  
}
```



我想让向上的按钮设置在左边，向下按钮在右边

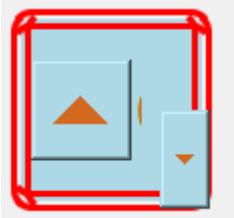
```
QSpinBox::up-button #设置向上按钮尺寸大小
```

```
QSpinBox{  
    font-size: 30px;      /*字体大小*/  
    color: chocolate; /*字体颜色*/  
    border: 10px double red; /*线宽 双线 红色*/  
    border-radius: 10px; /*设置控件四角圆弧*/  
    background-color: lightblue; /*背景色*/  
}  
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/  
    width: 50px;  
    height: 50px;  
}
```



你看，向上按钮尺寸发生了变化

```
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/  
    width: 50px;  
    height: 50px;  
}  
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/  
    subcontrol-position:left center; /*向上按钮左对齐,居中*/  
}
```



```
subcontrol-position #QSS 设置控件位置
```

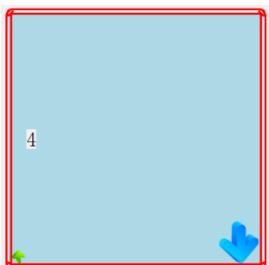
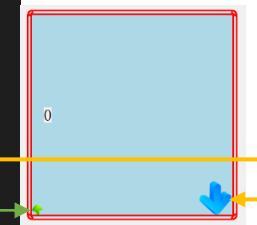
```
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/  
    width: 50px;  
    height: 50px;  
}  
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/  
    subcontrol-origin: padding;  
    subcontrol-position:left bottom; /*向上按钮左对齐,居中*/  
}  
QSpinBox::down-button{ /*用子选择器里面的down-button设置向下按钮*/  
    subcontrol-origin: padding;  
    subcontrol-position:right bottom;  
}
```



向 QSpinBox 上下按键加入图标

```
sb = QSpinBox(w)
sb.move(50,50)
sb.resize(400,400) #控件尺寸加大
```

```
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/
    width: 20px;
    height: 20px;
}
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/
    subcontrol-origin: padding;
    subcontrol-position:left bottom; /*向上按钮左对齐,居中*/
    image:url(xzzup.png)
}
QSpinBox::down-button{ /*用子选择器里面的down-button设置向下按钮*/
    subcontrol-origin: padding;
    subcontrol-position:right bottom;
    image:url(xzzdown.png)
}
```



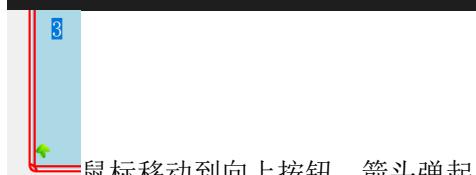
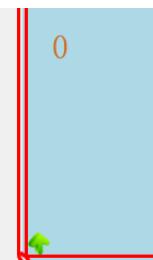
上下按键点击有效果，只是没有动态，感觉效果很死板。

```
QSpinBox::up-button:hover{...} #加入向上按钮伪状态，鼠标移动到向上按钮控件执行
```

```
QSpinBox::up-button{ /*用子选择器里面的up-button设置向上按钮位置*/
    subcontrol-origin: padding;
    subcontrol-position:left bottom; /*向上按钮左对齐,居中*/
    image:url(xzzup.png)
}

QSpinBox::up-button:hover{ /*加入向上按钮伪状态*/
    bottom: 10px; /*向上按钮收到鼠标移动，向上跑10px*/
}

QSpinBox::down-button{ /*用子选择器里面的down-button设置向下按钮*/
    subcontrol-origin: padding;
    subcontrol-position:right bottom;
    image:url(xzzdown.png)
}
```



鼠标移动到向上按钮，箭头弹起

```
QSpinBox::down-button:hover{.....} #加入向下按钮伪状态，鼠标移动到向下按钮控件  
执行
```

```
QSpinBox::up-button:hover{ /*加入向上按钮伪状态*/  
    bottom: 10px; /*向上按钮收到鼠标移动，向上跑10px，也就是向上控件距离底部10px*/  
}
```

```
QSpinBox::down-button{ /*用子选择器里面的down-button设置向下按钮*/  
    subcontrol-origin: padding;  
    subcontrol-position:right bottom;  
    image:url(xzzdown.png)  
}
```

```
QSpinBox::down-button:hover{  
    top: 10px; /*向下按钮收到鼠标移动，向下跑10px，也就是向下控件距离底部10px*/  
}
```



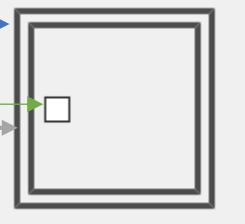
向下键鼠标未选择

向下键鼠标选择

## QCheckBox 勾选按钮 QSS 设置

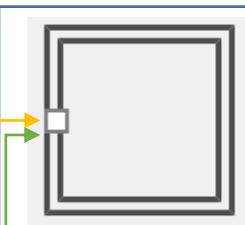
```
cb = QCheckBox(w) #创建勾选按钮  
cb.move(50,50)  
cb.resize(100,100) #控件尺寸
```

```
QCheckBox{  
    color: gray; /*前景色为灰色*/  
    border: 10px double gray(76, 76, 76);  
    padding: 5px;  
}
```

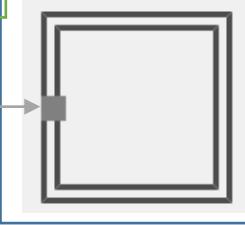


```
QCheckBox{  
    color: gray; /*前景色为灰色*/  
    border: 10px double gray(76, 76, 76);  
    padding: 5px;  
}
```

```
QCheckBox::indicator{ /*勾选框，未点击，默认执行*/  
    subcontrol-origin:border;  
    subcontrol-position:left center; /*勾选框，靠左边居中*/  
    background: white; /*勾选框未点击显示白色*/  
    border: 2px solid gray;  
}
```



```
QCheckBox::indicator:checked{ /*勾选框点击后执行*/  
    background-color: gray;  
}
```



## PyQT5 自定义信号

自定义信号必须以类属性的方式才能定义，所以你必须先创建个类

```
对象 = pyqtSignal() #创建一个信号对象  
pyqtSignal().emit() #发送信号，该信号发送到 pyqtSignal 所在的类创建的对象
```

```
class Btn(QPushButton): #继承自按钮  
    xClicked = pyqtSignal() #类里面的变量就是类属性,用pyqtSignal定义一个信号  
  
    def mousePressEvent(self,evt): #只有借助鼠标按下事件，去触发emit  
        self.xClicked.emit()  
  
btn1 = Btn(w) #创建按钮  
btn1.move(100,100)  
btn1.resize(100,100)  
  
btn1.xClicked.connect(lambda : print("槽函数执行"))
```

槽函数执行  
槽函数执行  
槽函数执行  
槽函数执行  
槽函数执行  
槽函数执行



点击按钮，槽函数执行。

```
对象 = pyqtSignal(数据类型) #信号发送数据类型的值给槽函数，数据类型就接收数据  
的定义(可以是字符串，整形，浮点型)  
pyqtSignal().emit(数据/变量) #将变量和数据传递给信号，信号将发出去
```

```
class Btn(QPushButton): #继承自按钮  
    xClicked = pyqtSignal(str) #信号发送字符串给槽函数←  
  
    def mousePressEvent(self,evt): #只有借助鼠标按下事件，去触发emit  
        self.xClicked.emit("12345") #发射的时候可以加入字符串变量  
  
btn1 = Btn(w) #创建按钮  
btn1.move(100,100)  
btn1.resize(100,100)  
  
btn1.xClicked.connect(lambda v: print(v)) #槽函数接收emit发来的数据，用v变量接收
```

12345  
12345  
12345  
12345  
12345  
12345



信号传参操作成功

信号发送多个参数实现

```
class Btn(QPushButton): #继承自按钮
    xClicked = pyqtSignal(str,int) #信号传入两个参数

    def mousePressEvent(self,evt): #只有借助鼠标按下事件，去触发emit
        self.xClicked.emit("12345",255) #发射两个参数

btn1 = Btn(w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)

def cao(s,num): #用s和num接收emit发来的字符串和整形
    print("%s = %d" %(s,num))

btn1.xClicked.connect(cao) #槽函数接收emit发来的数据
```

w.show()

```
12345 = 255
12345 = 255
12345 = 255
12345 = 255
12345 = 255
12345 = 255
12345 = 255
```



测试成功

```
对象 = pyqtSignal([数据类型],[数据类型]) #信号用[]括号设置多个参数类型，但是只能发射其中一个类型
```

```
class Btn(QPushButton): #继承自按钮
    xClicked = pyqtSignal([str],[int]) #信号只发送1个类型数据，字符串类型或者整形

    def mousePressEvent(self,evt): #只有借助鼠标按下事件，去触发emit
        self.xClicked.emit("12345") #发射1个数据，我先发送字符串
```

```
btn1 = Btn(w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
```

```
def cao(data): #用data接收数据
    print(data)
```

```
btn1.xClicked[str].connect(cao) #槽函数接收emit发来的数据
```

```
12345
12345
12345
12345
12345
```



```

class Btn(QPushButton): #继承自按钮
    xClicked = pyqtSignal([str],[int]) #信号只发送1个类型数据, 字符串类型或者整形

    def mousePressEvent(self,evt): #只有借助鼠标按下事件, 去触发emit
        self.xClicked[int].emit(888) #发射1个数据, 我发送整形, 记住, 一定要指定[int]

btn1 = Btn(w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)

def cao(data): #用data接收数据
    print(data)

btn1.xClicked[int].connect(cao) #槽函数接收emit发来的数据
#emit发射的数据是整形, 那么槽函数接收也必须是[int] 不然会报错

```

888      

上面字符串可以默认发送, 但是 int  
整形发送就必须指定

## 控件动画实现

```

对象 = QPropertyAnimation(父对象) #属性动画类, 创建动画
QPropertyAnimation::setTargetObject(传入控件) #将要用于做动画的控件对象传入
QPropertyAnimation::setProperty(参数) #做动画要用什么属性。
参数: pos 就是控件坐标移动,
      size 就是控件尺寸变化,
      geometry 就是控件坐标变化同时控件大小也变化
QPropertyAnimation::setStartValue(坐标参数) #动画在窗口开始位置
QPropertyAnimation::setEndValue(坐标参数) #动画在窗口结束位置
QPropertyAnimation::setDuration(ms 毫秒) #设置动画从开始到结束运行时间
QPropertyAnimation::start() #启动动画

```

```

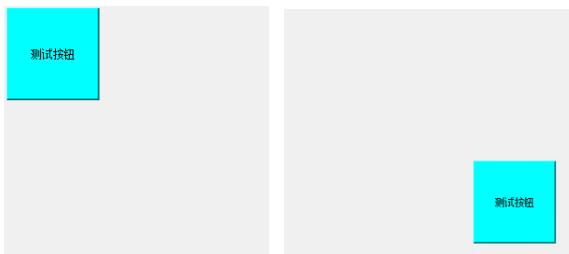
w = QWidget()
w.resize(500,500)

btn1 = QPushButton("测试按钮",w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
btn1.setStyleSheet("background-color:cyan;")

animation = QPropertyAnimation(w) #创建属性动画
animation.setTargetObject(btn1) #将按钮做成功动画
animation.setProperty(b"pos") #我默认用pos
animation.setStartValue(QPoint(0,0)) #动画在窗口开始的坐标
animation.setEndValue(QPoint(300,300)) #动画在窗口结束坐标
animation.setDuration(3000) #动画从开始坐标移动到结束坐标, 一共用3秒
animation.start() #启动动画

w.show()
sys.exit(app.exec_())

```

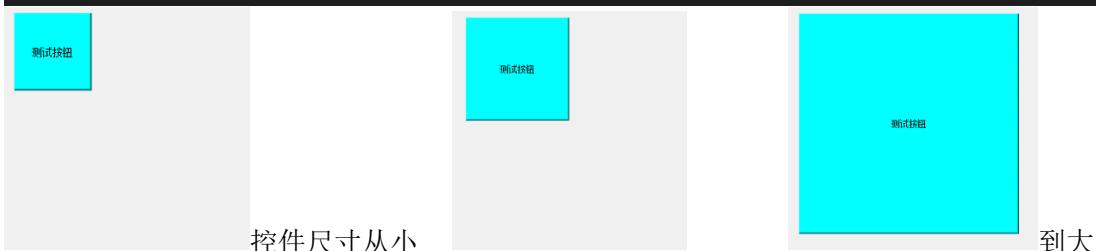


动画从 x=0,y=0 的位置向 x=300,y=300 滑动

```
QPropertyAnimation::setPropertyName(b"size") #使用 size 就是控件尺寸大小动画变化
```

```
btn1 = QPushButton("测试按钮",w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
btn1.setStyleSheet("background-color:cyan;")

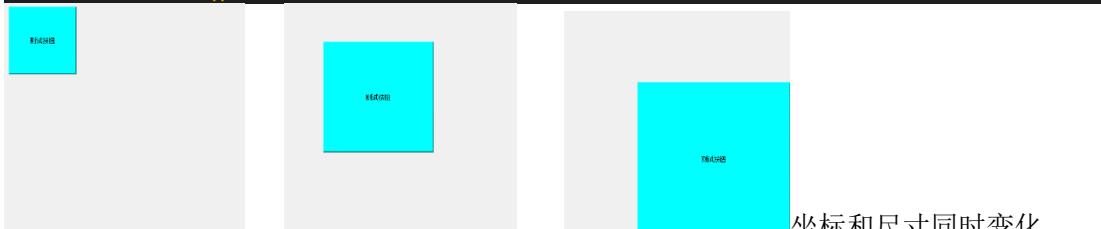
animation = QPropertyAnimation(w) #创建属性动画
animation.setTargetObject(btn1) #将按钮做成功能
animation.setPropertyNames(b"size") #使用size 就是控件尺寸大小动画变化
animation.setStartValue(QSize(0,0)) #动画在窗口开始的坐标
animation.setEndValue(QSize(300,300)) #动画在窗口结束坐标
animation.setDuration(3000) #动画从开始坐标移动到结束坐标，一共用3秒
animation.start() #启动动画
```



```
QPropertyAnimation::setPropertyNames(b"geometry") #使用 geometry 设置控件坐标变化，也可以同时尺寸变化
```

```
QRect(控件 x 坐标, 控件 y 坐标, 控件宽度尺寸, 控件高度尺寸) #QRect 即设置控件位置，也设置控件尺寸大小
```

```
animation = QPropertyAnimation(w) #创建属性动画
animation.setTargetObject(btn1) #将按钮做成功能
animation.setPropertyNames(b"geometry") #使用geometry 设置控件坐标变化，也可以同时尺寸变化
animation.setStartValue(QRect(0,0,100,100)) #用QRect设置动画开始坐标和控件开始尺寸
animation.setEndValue(QRect(200,200,300,300)) #用QRect设置动画结束坐标和控件结束尺寸
animation.setDuration(3000) #动画从开始坐标移动到结束坐标，一共用3秒
animation.start() #启动动画
```



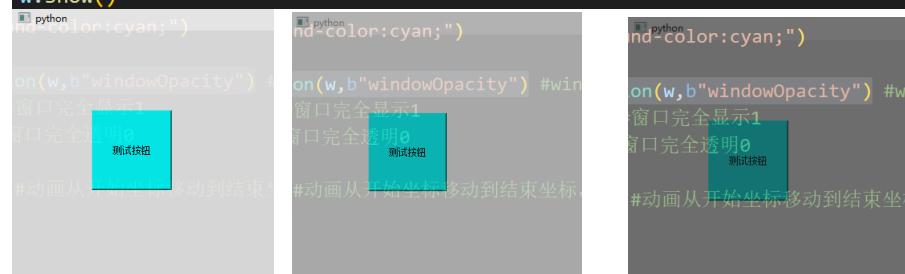
```
QPropertyAnimation(父对象(一定是窗口类 QWidget), b"windowOpacity") #设置窗口为显示~透明模式 0 为完全显示, 1 为完全透明
```

```
btn1 = QPushButton("测试按钮", w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
btn1.setStyleSheet("background-color:cyan;")

animation = QPropertyAnimation(w,b"windowOpacity") #windowOpacity设置窗口透明度, 取值范围0~1
animation.setStartValue(1) #窗口完全显示1
animation.setEndValue(0) #窗口完全透明0

animation.setDuration(3000) #动画窗口完全显示到窗口完全透明需要3秒
animation.start() #启动动画

w.show()
```



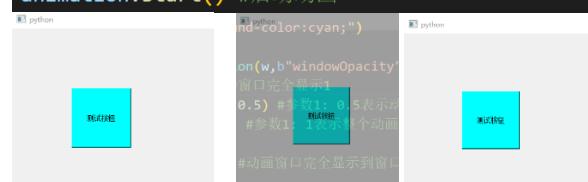
从完全显示，到完全透明用了 3 秒。

## 设置窗口透明与不透明来回闪烁

```
QPropertyAnimation::setKeyValueAt(填入动画时间区域,透明值) #在动画运行过程中插入控制
```

```
animation = QPropertyAnimation(w,b"windowOpacity") #windowOpacity设置窗口透明度, 取值范围0~1
animation.setStartValue(1) #窗口完全显示1
animation.setKeyValueAt(0.5,0.5) #参数1: 0.5表示动画在1.5秒的时候透明度达到参数2:0.5 = 50%
animation.setKeyValueAt(1,1) #参数1: 1表示整个动画结束的时候透明度达到参数2: 1 = 不透明

animation.setDuration(3000) #动画窗口完全显示到窗口完全透明需要3秒
animation.start() #启动动画
```



动画从不透明变成半透明再到不透明

## 动画移动节奏控制

```
QPropertyAnimation::setEasingCurve(运动参数)
```

```
对象 = QPropertyAnimation(控件对象,b"pos") #使用 pos 模式父窗口支持控件对象
```

```
btn1 = QPushButton("测试按钮",w) #创建按钮
btn1.move(100,100)
btn1.resize(100,100)
btn1.setStyleSheet("background-color:cyan;")

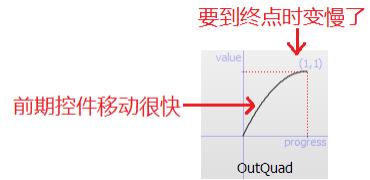
animation = QPropertyAnimation(btn1,b"pos") #设置成pos位置移动模式
animation.setStartValue(QPoint(0,0)) #控件起始移动位置
animation.setEndValue(QPoint(300,300))#控件移动结束位置

animation.setDuration(3000) #动画窗口完全显示到窗口完全透明需要3秒
animation.setEasingCurve(QEasingCurve.Linear) #设置动画移动速度,线性匀速运动
animation.start() #启动动画
```

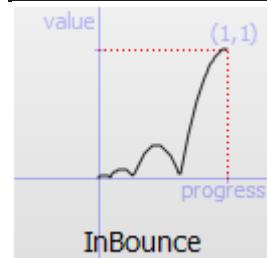


```
QPropertyAnimation::setEasingCurve(QEasingCurve::OutQuad) #设置动画由快到慢
```

```
animation = QPropertyAnimation(btn1, b"pos") #设置成pos位置移动模式  
animation.setStartValue(QPoint(0,0)) #控件起始移动位置  
animation.setEndValue(QPoint(300,300))#控件移动结束位置  
  
animation.setDuration(3000) #动画窗口完全显示到窗口完全透明需要3秒  
animation.setEasingCurve(QEasingCurve::OutQuad) #设置动画有快到慢  
animation.start() #启动动画
```



```
QPropertyAnimation::setEasingCurve(QEasingCurve::InBounce) #设置动画弹簧效果，开始移动是弹簧，移动结尾是匀速
```



前面一段距离来回弹，后面一段距离直线运动。

# PyQT5 多线程使用

## 多线程创建

```
from PyQt5.QtCore import QThread #加入 pyQt5 多线程库
对象 = 类(QThread) #类继承至 QThread, 实现 run 函数, 就形成多线程了
class worker(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 1
    def __init__(self): #默认写法
        super(worker, self).__init__()

    def run(self): #子线程运行
        while True:
            print("worker...")
            time.sleep(1)

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

thread1 = worker() #创建线程对象
thread1.start() #启动线程

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



线程创建成功

两个线程同时运行

```
class xthread(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 2
    def __init__(self):
        super(xthread, self).__init__()

    def run(self):
        while True:
            print("xthread...")
            time.sleep(1)

class worker(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 1
    def __init__(self): #默认写法
        super(worker, self).__init__()

    def run(self): #子线程运行
        while True:
            print("worker...")
            time.sleep(1)
```

两个线程, 创建两个不同的类。

```

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

thread1 = worker() #创建线程对象 1
thread1.start() #启动线程

thread2 = xthread() #创建线程 2
thread2.start()

w.show()
sys.exit(app.exec_())

```

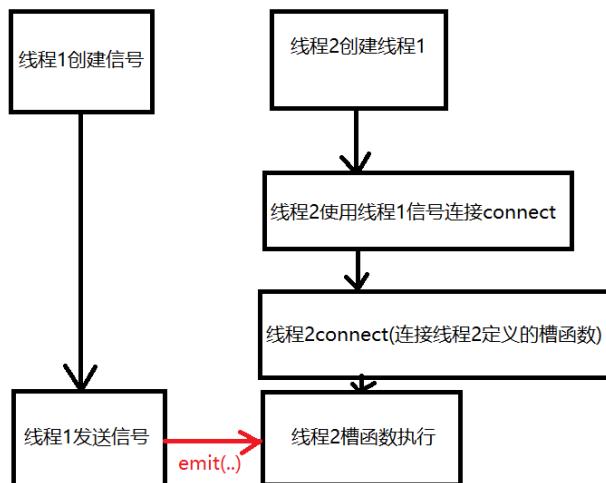
xthread...  
worker...  
xthread...  
worker...  
xthread...  
worker...  
xthread...  
worker...  
xthread...

python

两个线程同时执行。

## 线程自定义信号，用于界面实时更新

第 1 种方式



这是线程之间耦合的信号传递数据方式

```

class xthread(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 2
    xsignal = pyqtSignal([str])
    def __init__(self):
        super(xthread, self).__init__()

    def run(self):
        while True:
            print("xthread...")
            self.xsignal.emit("send12345") #发送信号给 worker 类建立的槽函数
            time.sleep(1)

```

```

class worker(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 1

    def __init__(self): #默认写法
        super(worker, self).__init__()
        self.thread2 = xthread() #线程类中, 创建另外线程 线程耦合方式
        self.thread2.start() #启动线程
        self.thread2.xsignal[str].connect(self.xcao) #连接本类的槽函数,
接收另外线程发来的数据

    def xcao(self, data): #槽函数 必须加 self, data 就是接收的信号数据
        print(data)

    def run(self): #子线程运行
        while True:
            print("worker...")
            time.sleep(1)

```

```

app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

thread1 = worker() #创建线程对象 1
thread1.start() #启动线程

w.show()
sys.exit(app.exec_())

```

```

worker...
send12345
xthread...
worker...
send12345
xthread...
worker...
send12345
worker...
xthread...
send12345

```

## QMutex 互斥锁(线程锁)与全局变量在多线程读写的安全

注意:在 pyQT5 多线程使用全局变量 global 只能在 run 函数里面声明

```

from PyQt5.QtCore import QThread, QMutex #加入互斥锁(线程锁)

qm1 = QMutex() #创建互斥锁(线程锁)
gdata = 50 #共享的全局数据

```

```
class xthread(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 2
    def __init__(self):
        super(xthread, self).__init__()
    def run(self):
        global gdata #全局变量只能在 run 里面声明
        while True:
            qm1.lock() #上锁
            gdata += 1 #加 1
            print("xthread = %d" %gdata)
            qm1.unlock()#解锁
            time.sleep(1)
```

```
class worker(QThread): #类继承至 QThread, 那么该类的创建的对象就是线程 1

    def __init__(self): #默认写法
        super(worker, self).__init__()

    def run(self): #子线程运行
        global gdata #全局变量只能在 run 里面声明
        while True:
            qm1.lock()
            gdata = gdata + 100 #加 100
            print("worker = %d" %gdata)
            qm1.unlock()
            time.sleep(1)
```

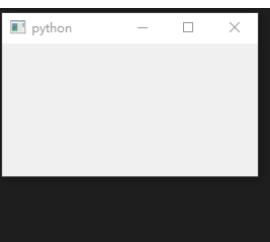
```
app = QApplication(sys.argv)

w = QWidget()
w.resize(500,500)

thread1 = worker() #创建线程对象 1
thread1.start() #启动线程

thread2 = xthread() #创建线程对象 2
thread2.start() #启动线程

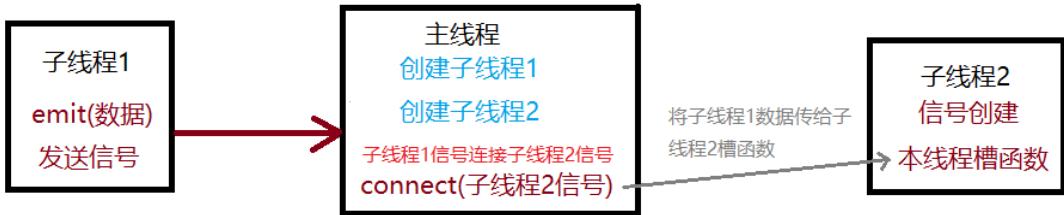
w.show()
sys.exit(app.exec_())
```



```
xthread = 51
worker = 151
xthread = 152
worker = 252
xthread = 253
xthread = 354
worker = 454
xthread = 455
worker = 555
xthread = 556
```

运行正常

## 两个线程相互通信或者多个线程之间相互通信传递数据，使用信号连接信号



```
发送端对象 = pyqtSignal(int) #比如创建个信号，发送 int 数据  
接收端对象 = pyqtSignal(int) #接收线程创建个信号，接收 int 数据  
发送端对象.connect(接收线程.接收端对象[int]) #发送端信号用 connect 连接接收端  
信号
```

```
from PyQt5.QtCore import QThread

class xthread(QThread): #类继承至 QThread，那么该类的创建的对象就是线程 2

    xthreadsig = pyqtSignal(int) #发送信号线程
    def __init__(self):
        super(xthread, self).__init__()

    def run(self):
        cont = 0
        while True:
            cont = cont + 1
            print("xthread...")
            self.xthreadsig[int].emit(cont) #发送计时数据
            time.sleep(1)

class worker(QThread): #类继承至 QThread，那么该类的创建的对象就是线程 1
    workerig = pyqtSignal(int) #创建信号，用于接收其它线程信号发来的数据

    def __init__(self): #默认写法
        super(worker, self).__init__()
        self.workerig.connect(self.workerCAO) #接收数据槽函数

    def run(self): #子线程运行
        while True:
            print("worker....")
            time.sleep(1)

    def workerCAO(self,value): #线程里面的槽函数一定要加 self 接收数据
        value
        print("workerCAO = %d " %value)

def appCAO(value):
    print("appCAO value = %d" %value)

app = QApplication(sys.argv)
```

```
w = QWidget()
w.resize(500,500)

thread1 = worker() #创建线程对象 1
thread1.start() #启动线程

thread2 = xthread() #创建线程对象 2
thread2.xthreadsig.connect(appCAO) #尝试子线程外部连接槽函数
thread2.xthreadsig.connect(thread1.workerig[int]) #信号连接信号，线程
xthread 发送数据给 worker 线程
thread2.start() #启动线程

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

```
xthread...
appCAO value = 9
workerCAO = 9
worker....
xthread...
appCAO value = 10
workerCAO = 10
```

thread2 发数据， thread1 接收

xthread 累加的数据， thread1 的 workerCAO 函数接收成功

## 综合应用案例，程序打包 exe

应用案例实现



```
from PyQt5.Qt import *
from PyQt5.QtWidgets import QComboBox, QDateTimeEdit, QDial, QDoubleSpinBox, QFormLayout, QRubberBand
import sys

#注册界面类
class RegiseterUser():
    def __init__(self, xobj): #创建对象的时候，类名传入父句柄给xobj
        lb1 = QLabel("用户名")
        lb2 = QLabel("密码")
        lb3 = QLabel("密码确认")
        le1 = QLineEdit()
        le2 = QLineEdit()
        le3 = QLineEdit()
        BtnAdd = QPushButton("注册")
        flayout = QFormLayout() #创建表单
        flayout.addRow(lb1,le1) #将控件放入表单，然后将表单放入网格布局管理器
        flayout.addRow(lb2,le2)
        flayout.addRow(lb3,le3)
        flayout.addRow(BtnAdd)

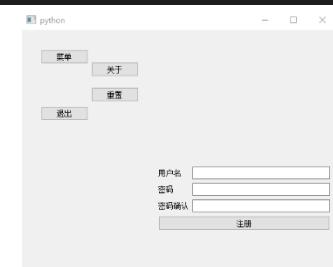
        w2 = QWidget() #该类不需要父句柄，因为下面网格布局管理器会设置父句柄
        w2.setFixedSize(200,200) #一定要固定左上角控件尺寸，不然因为网格布局管理器原因，会被注册控件覆盖
        menubtn = QPushButton("菜单",w2)
        menubtn.move(20,20)
        asbtn = QPushButton("关于",w2)
        asbtn.move(100,40)
        resbtn = QPushButton("重置",w2)
        resbtn.move(100,80)
        exitbtn = QPushButton("退出",w2)
        exitbtn.move(20,110)

        glayout=QGridLayout(xobj) #网格布局管理器嵌入传进来的父对象
        glayout.addWidget(w2,0,0)
        glayout.addLayout(flayout,1,1)

    app = QApplication(sys.argv)

    w = QWidget()
    w.resize(500,500)
    w1 = RegiseterUser(w) #创建注册界面

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



## 给注册界面增加样式

```
#注册界面类
class RegiseterUser():
    def __init__(self,xobj): #创建对象的时候，类名传入父句柄给xobj

        lb1 = QLabel("用户名")
        lb1.setObjectName("Rlb") #修改标签控件字体
        lb2 = QLabel("密码")
        lb2.setObjectName("Rlb") #修改标签控件字体
        lb3 = QLabel("密码确认")
        lb3.setObjectName("Rlb") #修改标签控件字体
        le1 = QLineEdit()
        le1.setObjectName("edit") #修改编辑栏样式
        le2 = QLineEdit()
        le2.setObjectName("edit") #修改编辑栏样式
        le3 = QLineEdit()
        le3.setObjectName("edit") #修改编辑栏样式

        BtnAdd = QPushButton("注册")
        BtnAdd.setObjectName("BtnADD") #修改注册按钮样式

        layout = QFormLayout() #创建表单
        layout.addRow(lb1,le1) #将控件放入表单，然后将表单放入网格布局管理器
        layout.addRow(lb2,le2)
        layout.addRow(lb3,le3)
        layout.addRow(BtnAdd)

    app = QApplication(sys.argv)

    with open("prj.qss","r",encoding='utf-8') as xfile:
        textqss = xfile.read()
        app.setStyleSheet(textqss)

    w = QWidget()
    w.resize(500,500)
    w.setObjectName("w") #修改主窗口样式
    w1 = RegiseterUser(w) #创建注册界面

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

```
QWidget#w{
    background-color: #blanchedalmond; /*主界窗口背景色*/
}

QLabel#Rlb{
    color: #blueviolet; /*字体颜色/
    font: 16pt "方正舒体"; /*字体大小 字体书法(记住在open QSS文件的时候用'utf-8' 才能设置字体书法)*/
}

QLineEdit#edit{
    background-color: transparent; /*修改编辑框为透明背景*/
    color: #brown; /*设置编辑框字体颜色*/
}

QPushButton#BtnADD{
    width: 100px; /*按钮长高*/
    height: 50px;
    background-color: #cornflowerblue; /*按钮背景色*/
    color: #rgb(100, 0, 139);/*"注册" 字体颜色*/
    border-radius: 10px;/*按钮边框圆角设置*/
}
```



我觉得编辑框有边框好难看，我只需要底部边框

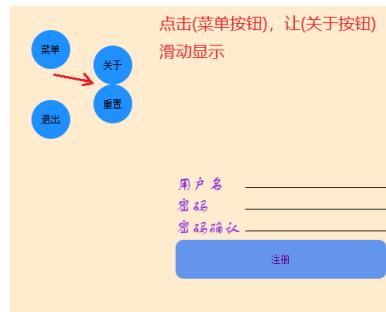
```
border:none; #取消控件四周边框
```

```
QLineEdit#edit{  
    background-color: transparent; /*修改编辑框为透明背景*/  
    color: brown; /*设置编辑框字体颜色*/  
    border:none; /*取消所有边框*/  
    border-bottom: 1px solid; /*增加底部边框*/  
}
```



你看，底部边框黑黑的一条直线。

## 按钮滑动弹出

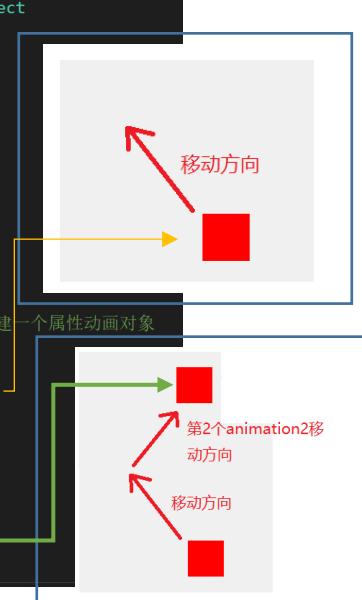


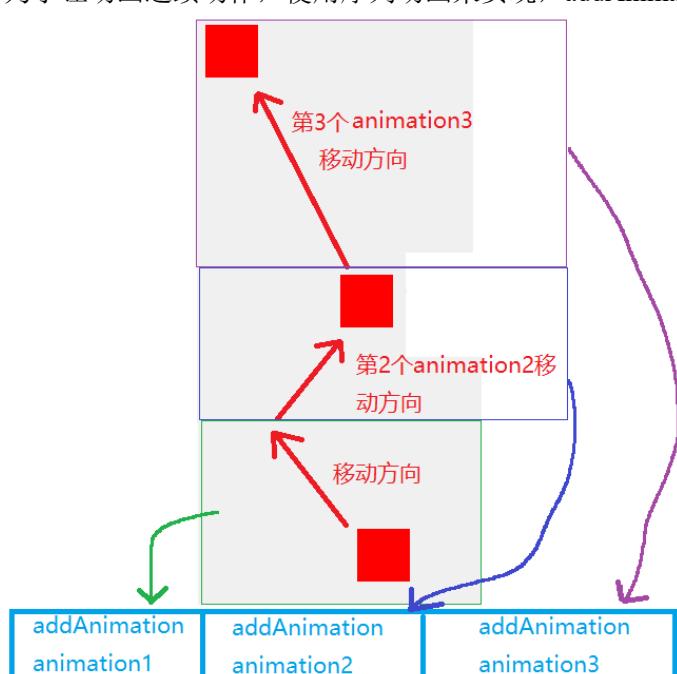
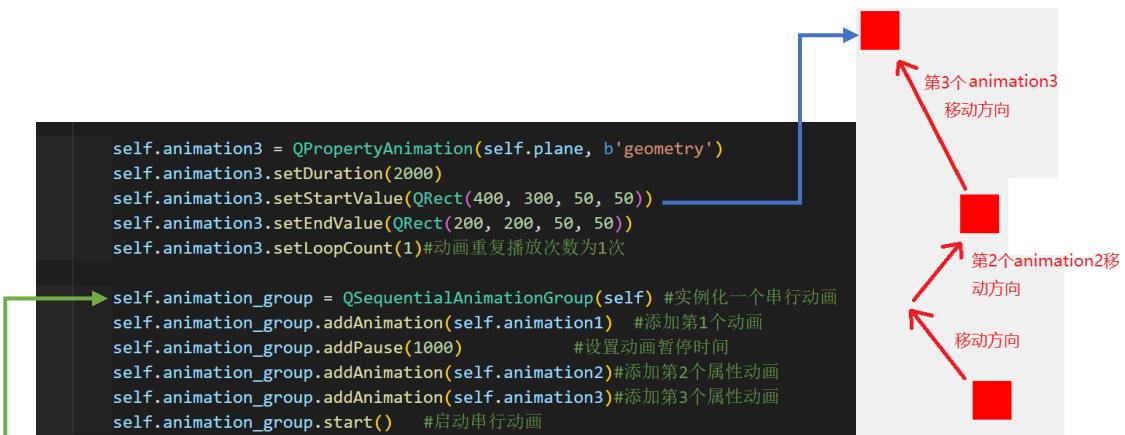
先讲讲序列动画类

```
对象 = QSequentialAnimationGroup(传入要显示序列动画的窗口对象)  
QSequentialAnimationGroup::addAnimation(添加 QPropertyAnimation 创建的动画  
对象) #按照添加顺序，显示创建的属性动画  
QPropertyAnimation::setLoopCount(传入重复次数) #属性动画重复播放次数，如果  
填入-1 表示无限重复播放  
QSequentialAnimationGroup::addPause(传入停止时间 ms 毫秒) #序列动画暂停  
QSequentialAnimationGroup::start() #启动动画
```

```
import sys  
from PyQt5.QtCore import QPropertyAnimation, QSequentialAnimationGroup, QRect  
from PyQt5.QtWidgets import QApplication, QWidget, QLabel
```

```
class Demo(QWidget):  
    def __init__(self):  
        super(Demo, self).__init__()  
        self.resize(600, 600)  
  
        self.plane = QLabel(self)  
        self.plane.resize(50, 50)  
        self.plane.setStyleSheet("background-color:red;")  
  
        self.animation1 = QPropertyAnimation(self.plane, b'geometry') #创建一个属性动画对象  
        #动画目标标签 位置和大小  
        self.animation1.setDuration(2000)#动画从开始到结束位置用多少ms毫秒  
        self.animation1.setStartValue(QRect(300, 500, 50, 50))#动画开始位置  
        self.animation1.setEndValue(QRect(200, 400, 50, 50))#动画结束位置  
        self.animation1.setLoopCount(1)#动画重复播放次数为1次  
  
        self.animation2 = QPropertyAnimation(self.plane, b'geometry')  
        self.animation2.setDuration(2000)  
        self.animation2.setStartValue(QRect(200, 400, 50, 50))  
        self.animation2.setEndValue(QRect(400, 300, 50, 50))
```





将动画按照addAnimation加入的先后顺序进行执行

这就是执行原理

```

if __name__ == '__main__':
    app = QApplication(sys.argv)
    demo = Demo()
    demo.show()
    sys.exit(app.exec_())

```

执行成功

## 按钮滑动弹出实现

```
animimalarray = [] #因为无法将__init__创建的窗口传递给槽函数，所以用列表做全局传递
def __init__(self,xobj): #创建对象的时候，类名传入父句柄给xobj

    lb1 = QLabel("用户名")
    lb1.setObjectName("Rlb") #修改标签控件字体
    lb2 = QLabel("密码")
    lb2.setObjectName("Rlb") #修改标签控件字体
    lb3 = QLabel("密码确认")
    lb3.setObjectName("Rlb") #修改标签控件字体
    le1 = QLineEdit()
    le1.setObjectName("edit") #修改编辑栏样式
    le2 = QLineEdit()

w2 = QWidget() #该类不需要父句柄，因为下面网格布局管理器会设置父句柄
w2.setFixedSize(200,200) #一定要固定左上角控件尺寸，不然因为网格布局管理器原因，会被注册控件覆盖
self.menubtn = QPushButton("菜单",w2)
self.menubtn.setObjectName("menuBTN") #修改左上角按钮控件样式
self.menubtn.move(20,20)
self.menubtn.clicked.connect(self.MenuCao)
self.asbtn = QPushButton("关于",w2)
self.asbtn.setObjectName("menuBTN")
self.asbtn.move(100,40)
resbtn = QPushButton("重置",w2)
resbtn.setObjectName("menuBTN")
resbtn.move(100,90)
exitbtn = QPushButton("退出",w2)
exitbtn.setObjectName("menuBTN")
exitbtn.move(20,110)

glayout=QGridLayout(xobj) #网格布局管理器嵌入传进来的父对象
glayout.addWidget(w2,0,0)
glayout.addLayout(flayout,1,1)

self.animimalarray.append(w2) #将窗口w2传递给列表，列表传递给槽函数
def MenuCao(self): #类中槽函数需要加self
    print("MenuCao....")
    animaGroup = QSequentialAnimationGroup(self.animimalarray[0]) #创建序列动画组
    animation = QPropertyAnimation() #创建属性动画
    animation.setTargetObject(self.asbtn) #将其中一个要做动画的按钮对象放入该目标
    animation.setProperty(b"pos") #移动对象模式
    animation.setStartValue(self.menubtn.pos()) #开始位置为主菜单按钮位置
    animation.setEndValue(self.asbtn.pos()) #结束位置为“关于”按钮位置
    animation.setDuration(1000)

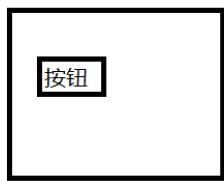
    animaGroup.addAnimation(animation)
    animaGroup.start(QAbstractAnimation.DeleteWhenStopped) #动画停止后删除
```

按钮的槽函数无法传递窗口对象，所以用(类)变量来传递

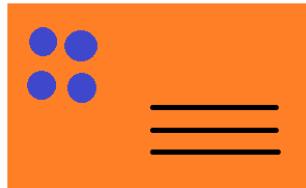
必须指定序列组在哪一个窗口对象上面滑动



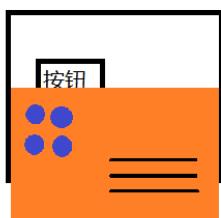
## 界面滑动切换



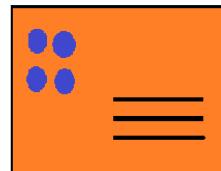
主界面



注册界面



注册界面开始覆盖



主界面  
注册界面完全覆盖

```
#注册界面类
class RegiseterUser():

    animimalarray = [] #因为无法将__init__创建的窗口传递给槽函数，所以用列表做全局传递
    def __init__(self,xobj): #创建对象的时候，类名传入父句柄xobj
        lb1 = QLabel("用户名")
        lb1.setObjectName("Rlb") #修改标签控件字体
        lb2 = QLabel("密码")
        lb2.setObjectName("Rlb") #修改标签控件字体
        lb3 = QLabel("密码确认")
        lb3.setObjectName("Rlb") #修改标签控件字体
        le1 = QLineEdit()
        le1.setObjectName("edit") #修改编辑栏样式
        le2 = QLineEdit()
        le2.setObjectName("edit") #修改编辑栏样式
        le3 = QLineEdit()
        le3.setObjectName("edit") #修改编辑栏样式

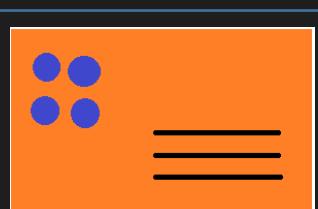
        BtnAdd = QPushButton("注册")
        BtnAdd.setObjectName("BtnADD") #修改注册按钮样式

        flayout = QFormLayout() #创建表单
        flayout.addRow(lb1,le1) #将控件放入表单，然后将表单放入网格布局管理器
        flayout.addRow(lb2,le2)
        flayout.addRow(lb3,le3)
        flayout.addRow(BtnAdd)

        w2 = QWidget() #该类不需要父句柄，因为下面网格布局管理器会设置父句柄
        w2.setFixedSize(200,200) #一定要固定左上角控件尺寸，不然因为网格布局管理器原因，会被注册控件覆盖
        self.menubtn = QPushButton("菜单",w2)
        self.menubtn.setObjectName("menuBTN") #修改左上角按钮控件样式
        self.menubtn.move(20,20)
        self.menubtn.clicked.connect(self.MenuCao)
        self.asbtn = QPushButton("关于",w2)
        self.asbtn.setObjectName("menuBTN")
        self.asbtn.move(100,40)
        resbtn = QPushButton("重置",w2)
        resbtn.setObjectName("menuBTN")
        resbtn.move(100,90)
        exitbtn = QPushButton("退出",w2)
        exitbtn.setObjectName("menuBTN")
        exitbtn.move(20,110)

        glayout=QGridLayout(xobj) #网格布局管理器嵌入传进来的父对象
        glayout.addWidget(w2,0,0)
        glayout.addLayout(flayout,1,1)

        self.animimalarray.append(w2) #将窗口w2传递给列表，列表传递给槽函数
```



```

def MenuCao(self): #类中槽函数需要加self
    print("MenuCao....")
    animaGroup = QSequentialAnimationGroup(self.animalarray[0]) #创建序列动画组
    animation = QPropertyAnimation() #创建属性动画
    animation.setTargetObject(self.asbtn) #将其中一个要做动画的按钮对象放入该目标
    animation.setProperty(b"pos") #移动对象模式
    animation.setStartValue(self.menubtn.pos()) #开始位置为主菜单按钮位置
    animation.setEndValue(self.asbtn.pos())
    animation.setDuration(1000)

    animaGroup.addAnimation(animation)
    animaGroup.start(QAbstractAnimation.DeleteWhenStopped) #动画停止后删除

```

注册界面代码不变

```

class MainInterface():
    list = []
    def __init__(self,xobj): #创建对象的时候，类名传入父句柄给xobj

        Btn = QPushButton(xobj) #主界面按钮切换窗口
        Btn.move(100,100)
        Btn.clicked.connect(self.CAO)
        self.list.append(xobj)
    def CAO(self):
        print("1111")
        w2 = QWidget(self.list[0])
        w2.resize(500,500)
        w2.setObjectName("regwindow")
        regisw = RegiseterUser(w2) #创建注册界面，将注册界面嵌入进w2窗口
        w2.move(100,100) #移动到斜角方便区分
        w2.show() #显示注册界面

```

主界面代码

```

app = QApplication(sys.argv)

with open("prj.qss","r",encoding='utf-8') as xfile:
    textqss = xfile.read()
    app.setStyleSheet(textqss)

w = QWidget()
w.resize(500,500)
w.setObjectName("w") #修改主窗口样式
xmain = MainInterface(w) #创建主界面嵌入进w窗口

w.show()
sys.exit(app.exec_())

```

开始运行



修改主界面类如下：

```
class MainInterface():
    list = [] #用来放槽函数执行完之后需要存放的变量，对象，数据
    def __init__(self,xobj): #创建对象的时候，类名传入父句柄给xobj

        Btn = QPushButton(xobj) #主界面按钮切换窗口
        Btn.move(100,100)
        Btn.clicked.connect(self.CAO)
        self.list.append(xobj)

    def CAO(self):
        print("1111")
        w2 = QWidget(self.list[0])
        w2.resize(500,500)
        w2.setObjectName("regwindow")
        regisw = RegiseterUser(w2) #创建注册界面，将注册界面嵌入进w2窗口
        #w2.move(100,100) #移动到斜角方便区分
        w2.show() #显示注册界面
        self.list.append(regisw) #因为槽函数执行完之后，会释放槽函数创建的对象和数据，所以必须在槽函数最后执行完之前把注册界面放入全局变量，或者类变量
```



点击注册界面里面的  
按钮没有按钮滑动效  
果，改善

按钮有滑动效果

下面实现界面切换，主要还是修改主界面槽函数

```
class MainInterface():
    list = [] #用来放槽函数执行完之后需要存放的变量，对象，数据
    def __init__(self,xobj): #创建对象的时候，类名传入父句柄给xobj

        Btn = QPushButton(xobj) #主界面按钮切换窗口
        Btn.move(100,100)
        Btn.clicked.connect(self.CAO)
        self.list.append(xobj)

    def CAO(self):
        print("1111")
        w2 = QWidget(self.list[0])
        w2.resize(500,500)
        w2.setObjectName("regwindow")
        regisw = RegiseterUser(w2) #创建注册界面，将注册界面嵌入进w2窗口
        #w2.move(100,100) #移动到斜角方便区分
        w2.show() #显示注册界面
        self.list.append(regisw) #因为槽函数执行完之后，会释放槽函数创建的对象和数据，

        w2.move(0,w2.height()) #窗口.height() 得到窗口高度，意思就是让注册窗口移动到主界面窗口最底部
        w2.show()#显示注册界面
        anima = QPropertyAnimation(w2)#创建窗口动画对象
        anima.setTargetObject(w2) #滑动w2注册窗口
        anima.setPropertyNames(b"pos") #窗口坐标移动模式，就是滑动模式
        anima.setStartValue(w2.pos()) #窗口滑动启动坐标，w2窗口当前位置，在底部
        anima.setEndValue(QPoint(0,0)) #w2窗口滑动到主窗口顶部
        anima.setDuration(500) #500ms毫秒时间滑动
        anima.start(QAbstractAnimation.DeleteWhenStopped) #启动滑动，滑动完成后删除动画
```



窗口滑动实现

## 将程序打包成 exe 文件

全功能打包工具 pyinstaller

该工具将写好的 python QT5 应用程序，应用程序依赖的库，跑应用程序的解释器，全部打包在一个目录下。直接将目录给用户就可以使用。

但是请注意，在什么平台上开发的 pyQT5，那么要求客户也必须是和开发者一样的平台  
比如开发者是 windows 64 位，客户也必须是 windows 64 位。如果客户是 32 位，可能会出些 bug。

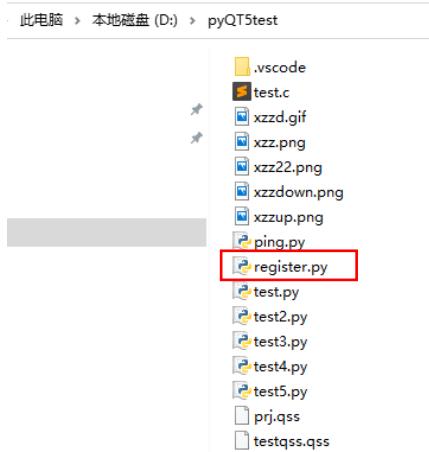
使用 pip3 安装 pyinstaller

确保网络正常

```
C:\Users\XZ666>cd C:\Users\XZ666\AppData\Local\Programs\Python\Python37\Scripts
C:\Users\XZ666\AppData\Local\Programs\Python\Python37\Scripts>pip3.7.exe install pyinstaller
Collecting pyinstaller
  Downloading pyinstaller-4.5.1-py3-none-win_amd64.whl (1.9 MB)
[██████████] | 1.9 MB 652 kB/s
Requirement already satisfied: setuptools in c:\users\xz666\appdata\local\programs\python\python37\lib\site-packages (from pyinstaller) (47.1.0)
Collecting pywin32-ctypes>=0.2.0
  Downloading pywin32_ctypes-0.2.0-py3.7-none-any.whl (28 kB)
Collecting altgraph
  Downloading altgraph-0.17.2-py2.py3-none-any.whl (21 kB)

  Running setup.py install for future ... done
  Running setup.py install for pefile ... done
  WARNING: The scripts pyi-archive_viewer.exe, pyi-bindepend.exe, pyi-grab_version.exe, pyi-makespec.exe, pyi-set_version.exe and pyinstaller.exe are installed in 'c:\users\xz666\appdata\local\programs\python\python37\Scripts' which is not on PATH.
    Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed altgraph-0.17.2 future-0.18.2 importlib-metadata-4.8.1 pefile-2021.9.3 pyinstaller-4.5.1 pyinstaller-hooks-contrib-2021.3 pywin32-ctypes-0.2.0 typing-extensions-3.10.0.2 zipp-3.5.0
```

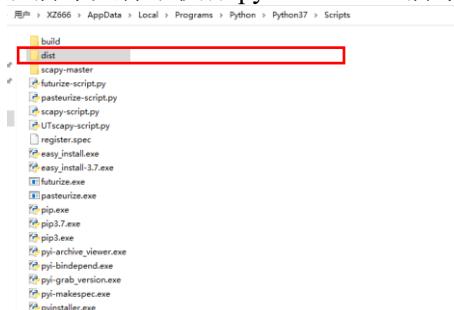
pyinstaller.exe –version 查看版本，必须在 3.3.1 之上  
出现版本号，表示安装成功。



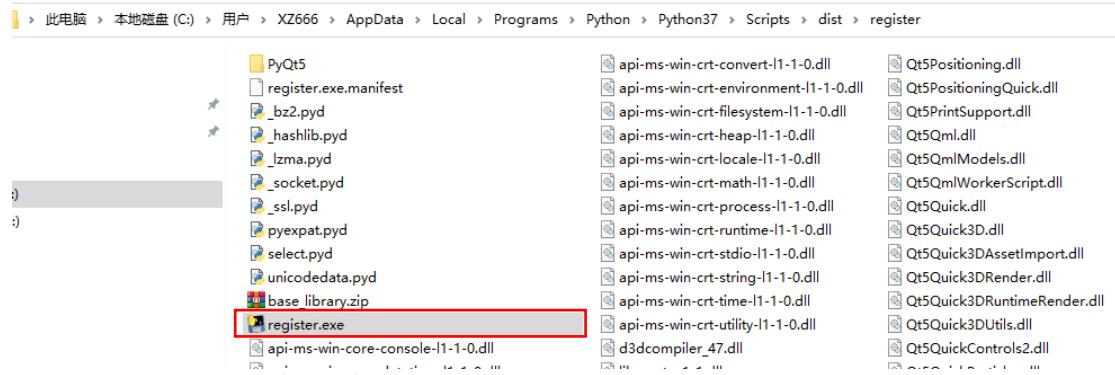
register.py 就是我的主程序，我打包试试看行不行

```
C:\Users\XZ666\AppData\Local\Programs\Python\Python37\Scripts>pyinstaller.exe D:\pyQT5test\register.py
```

在脚本文件下执行 pyinstaller.exe 脚本，指定应用程序路径，开始打包



打包完成后会在脚本文件目录下生成 dist



执行 register.exe 不成功，黑框闪退

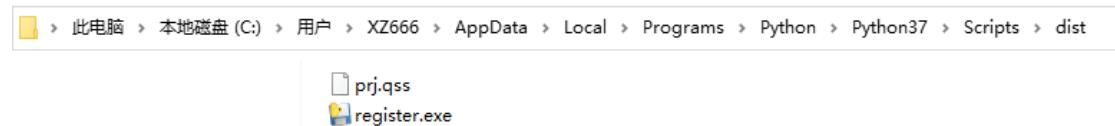
这是因为 windows 主机并没有用 pip3 装 PyQt5，我是用的 vscode 环境下的 PyQt5。

第 1 种解决方案：在本 windows 环境下装 PyQt5，在客户机器上也要装 PyQt5。

第 2 种解决方案：pyinstaller.exe -F -w D:\pyQT5test\register.py 。加入-F -w 将 vscode 的 PyQt5 环境压缩到 exe 程序中。

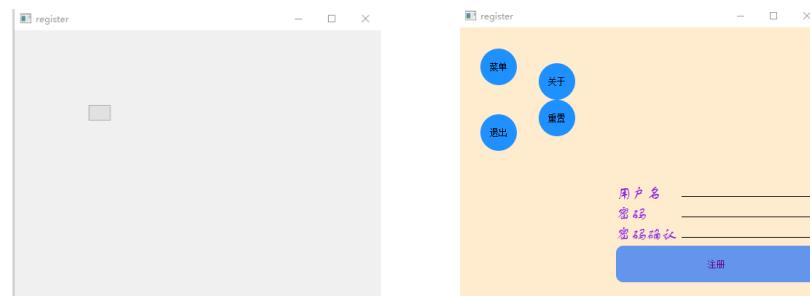
我选择第 2 中方案

```
C:\Users\XZ666\AppData\Local\Programs\Python\Python37\Scripts>pyinstaller.exe -F -w D:\pyQT5test\register.py
```



生成应用程序，记住要在应用程序同级目录下加入 QSS 文件。因为 QSS 是应用程序直接调用。QSS 文件无法编译进应用程序。

这个一次性打包的应用程序占用 60M 空间。所以一次性打包应用程序很大。



程序运行成功。