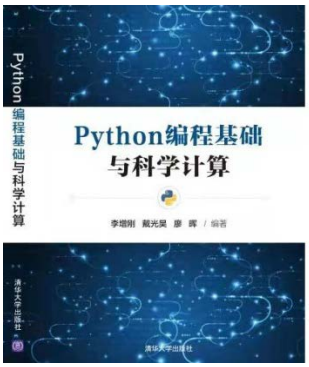
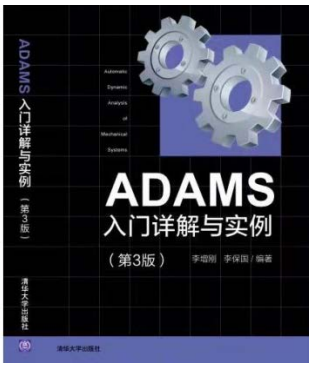
PyQt 中的事件及事件处理函数，节选自《Python 继承与 PyQt 可视化编程详解》，由清华大学出版社出版，该书正文部分有 610 页，讲解详细，在每个知识点有实例程序，另外也可参考《Qt for Python PySide6 GUI 界面开发详解与实例》和《Python 编程基础与科学计算》。





目录

第 10 章 事件及处理

事件（event）和前面经常用的信号一样，也是实现可视化控件之间联动的重要方法。事件是程序收到外界的输入，处于某种状态时自动发射的信号。事件有固定的类型，每种类型有自己的处理

函数，用户只要重写这些函数，即可达到特定的目的。通过事件可以用一个控件监测另外一个控件，并可过滤被监测控件发出的事件。

10.1 事件的类型

10.1.1 事件的概念

可视化应用程序在接受外界输入设备的输入时，例如鼠标、键盘等的操作，会对输入设备输入的信息进行分类，根据分类的不同，用不同的函数进行处理，做出不同的反应。外界对 PyQt 程序进行输入信息的过程称为事件，例如在窗口上单击鼠标、用鼠标拖动窗口、在输入框中输入数据等，这些都是外界对程序的输入，都可以称为事件。PyQt 程序对外界的输入进行处理的过程称为事件处理，根据外界输入信息的不同，处理事件的函数也不同。

前面建立的可视化程序中，在主程序中都会创建一个 QApplication 的应用程序实例对象，然后调用实例对象的 exec()函数，这将使应用程序进入一个循环，不断监听外界输入的信息，当输入的信息满足某种分类时，将会产生一个事件对象 QEvent()，事件对象中记录了外界输入的信息，并将事件对象发送给处理该事件对象的函数进行处理。

事件与前面讲过的信号与槽相似，但是又有不同。信号是指控件或窗口本身满足一定条件时，发射一个带数据的信息或不带数据的信息，需要编程人员为这个信息单独写处理这个信息的槽函数，并将信号和槽函数关联，发射信号时，自动执行与之关联的槽函数。而事件是外界对程序的输入，将外界的输入进行分类后交给函数处理，处理事件的函数是固定的，只需要编程人员把处理事件的函数重写，来达到处理外界输入的目的，不需要将事件与处理事件的函数进行连接，系统会自动调用能处理事件的函数，并把相关数据作为实参传递给处理事件的函数。

下面是一个处理鼠标单击事件的程序，在窗口的空白处单击鼠标左键，在 QLineEdit 控件上显示出鼠标单击点处的窗口坐标值，单击鼠标右键，显示右键单击处屏幕坐标值。单击鼠标左键或右键，将会产生 QMouseEvent 事件，QMouseEvent 事件的实例对象中有与鼠标事件相关的属性，如 button()方法获取单击的是左键还是右键，x()和 y()方法获取鼠标单击点处窗口坐标值，globalX()和 globalY()方法获取鼠标单击点处屏幕坐标值。QWidget 窗口处理 QMouseEvent 事件的函数有 mouseDoubleClickEvent(QMouseEvent)、mouseMoveEvent(QMouseEvent)、mousePressEvent(QMouseEvent)、mouseReleaseEvent(QMouseEvent)和 moveEvent(QMoveEvent)。

```
import sys    #Demo10_1.py
from PyQt5.QtWidgets import    QApplication,QWidget,QLineEdit
from PyQt5.QtCore import Qt


class MyWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.resize(500,500)
        self.lineEdit=QLineEdit(self)
```

```
            self.lineEdit.setGeometry(0,0,500,30)
        def mousePressEvent(self, event):    #重写处理 mousePress 事件的函数
            template1 ="单击点的窗口坐标是 x:{} y:{}"
            template2 = "单击点的屏幕坐标是 x:{} y:{}"
            if event.button() == Qt.LeftButton:    #button()获取左键或右键
                string = template1.format(event.x(),event.y()) #x()和 y()获取窗口坐标
                self.lineEdit.setText(string)
            if event.button() == Qt.RightButton:
                #globalX()和 globalY()获取全局坐标
                string = template2.format(event.globalX(), event.globalY())
                self.lineEdit.setText(string)

if __name__ == '__main__':
        app=QApplication(sys.argv)
        window = MyWindow()
        window.show()
        sys.exit(app.exec())
```

10.1.2 QEvent 类

QEvent 类是所有事件的基类，它在 QtCore 模块中。外界输入给程序的信息首先交给 QEvent 进行分类，得到不同类型的事件，然后将事件及相关信息交给控件或窗口的事件处理函数进行处理，得到对外界输入的响应。

QEvent 类的属性只有 accepted，方法如表 10-1 所示，主要方法介绍如下：

表 10-1 QEvent 类的方法

方法及参数类型	说　明	方法及参数类型	说　明
accept()	事件被接受	spontaneous()	获取事件是否立即被处理
ignore()	事件被拒绝	type()	获取事件的类型
isAccepted()	事件是否被接受	registerEventType ([hint=-1])	注册新的事件
setAccepted(bool)	设置事件是否被接受		

- 用 accept()或 setAccepted(True)方法接受一个事件，用 ignore()或 setAccepted(False)方法拒绝一个事件。被接受的事件不会再传递给其他对象；被拒绝的事件会传递给其他对象处理，如果没有对象处理，则该事件会被丢弃。
- 如果事件被 QWidget 的 event()函数进行了处理，则 spontaneous()方法的返回值是 True，否则返回 False。event()函数根据事件类型起到分发事件到指定处理函数的作用，可以在 event()函数中对事件进行处理。
- type()方法可以返回事件的类型。QEvent 中定义了事件的类型，QEvent 定义的主要事件类型如表 10-2 所示。

表 10-2 主要事件的类型

事件类型常量	值	所属事件类	说　明
QEvent.None	0		不是一个事件
QEvent.ActionAdded	114	QActionEvent	一个新 action 被添加
QEvent.ActionChanged	113	QActionEvent	一个 action 被改变
QEvent.ActionRemoved	115	QActionEvent	一个 action 被移除
QEvent.ActivationChange	99		顶层窗口激活状态发生变化
QEvent.ApplicationFontChange	36		程序的默认字体发生变化
QEvent.ApplicationPaletteChange	38		程序的默认调色板发生变化
QEvent.ApplicationStateChange	214		应用程序的状态发生变化
QEvent.ApplicationWindowIconChange	35		应用程序的图标发生变化
QEvent.ChildAdded	68	QChildEvent	一个对象获得孩子
QEvent.ChildPolished	69	QChildEvent	一个控件的孩子被抛光
QEvent.ChildRemoved	71	QChildEvent	一个对象失去孩子
QEvent.Clipboard	40		剪贴板的内容发生改变
QEvent.Close	19	QCloseEvent	Widget 被关闭
QEvent.ContentsRectChange	178		控件内容区外边距发生改变
QEvent.ContextMenu	82	QContextMenuEvent	上下文弹出菜单
QEvent.CursorChange	183		控件的鼠标发生改变
QEvent.DeferredDelete	52	QDeferredDeleteEvent	对象被清除后将被删除
QEvent.DragEnter	60	QDragEnterEvent	拖放操作时鼠标进入控件
QEvent.DragLeave	62	QDragLeaveEvent	拖放操作时鼠标离开控件
QEvent.DragMove	61	QDragMoveEvent	拖放操作正在进行
QEvent.Drop	63	QDropEvent	拖放操作完成
QEvent.DynamicPropertyChange	170		动态属性已添加、更改或删除
QEvent.EnabledChange	98		控件的 enabled 状态已更改
QEvent.Enter	10	QEnterEvent	鼠标进入控件的边界
QEvent.EnterEditFocus	150		编辑控件获得焦点进行编辑
QEvent.FileOpen	116	QFileOpenEvent	文件打开请求
QEvent.FocusIn	8	QFocusEvent	控件或窗口获得键盘焦点
QEvent.FocusOut	9	QFocusEvent	控件或窗口失去键盘焦点
QEvent.FocusAboutToChange	23	QFocusEvent	控件或窗口焦点即将改变
QEvent.FontChange	97		控件的字体发生改变
QEvent.Gesture	198	QGestureEvent	触发了一个手势
QEvent.GestureOverride	202	QGestureEvent	触发了手势覆盖
QEvent.GrabKeyboard	188		item 获得键盘抓取 （仅限 QGraphicsItem）
QEvent.GrabMouse	186		item 获得鼠标抓取 （仅限 QGraphicsItem）

QEvent.GraphicsSceneContextMenu	159	QGraphicsSceneContextMenuEvent	在图形场景上弹出菜单
QEvent.GraphicsSceneDragEnter	164	QGraphicsSceneDragDropEvent	拖放操作时鼠标进入场景
QEvent.GraphicsSceneDragLeave	166	QGraphicsSceneDragDropEvent	拖放操作时鼠标离开场景
QEvent.GraphicsSceneDragMove	165	QGraphicsSceneDragDropEvent	在场景上正在进行拖放操作
QEvent.GraphicsSceneDrop	167	QGraphicsSceneDragDropEvent	在场景上完成拖放操作
QEvent.GraphicsSceneHelp	163	QHelpEvent	用户请求图形场景的帮助
QEvent.GraphicsSceneHoverEnter	160	QGraphicsSceneHoverEvent	鼠标进入图形场景中的悬停项
QEvent.GraphicsSceneHoverLeave	162	QGraphicsSceneHoverEvent	鼠标离开图形场景一个悬停项
QEvent.GraphicsSceneHoverMove	161	QGraphicsSceneHoverEvent	鼠标在场景的悬停项内移动
QEvent.GraphicsSceneMouseDoubleClick	158	QGraphicsSceneMouseEvent	鼠标在图形场景中双击
QEvent.GraphicsSceneMouseMove	155	QGraphicsSceneMouseEvent	鼠标在图形场景中移动
QEvent.GraphicsSceneMousePress	156	QGraphicsSceneMouseEvent	鼠标在图形场景中按下
QEvent.GraphicsSceneMouseRelease	157	QGraphicsSceneMouseEvent	鼠标在图形场景中释放
QEvent.GraphicsSceneMove	182	QGraphicsSceneMoveEvent	控件被移动
QEvent.GraphicsSceneResize	181	QGraphicsSceneResizeEvent	控件已调整大小
QEvent.GraphicsSceneWheel	168	QGraphicsSceneWheelEvent	鼠标滚轮在图形场景中滚动
QEvent.Hide	18	QHideEvent	控件被隐藏
QEvent.HideToParent	27	QHideEvent	子控件被隐藏
QEvent.HoverEnter	127	QHoverEvent	鼠标进入悬停控件
QEvent.HoverLeave	128	QHoverEvent	鼠标离开悬停控件
QEvent.HoverMove	129	QHoverEvent	鼠标在悬停控件内移动
QEvent.IconDrag	96	QIconDragEvent	窗口的主图标被拖走
QEvent.InputMethod	83	QInputMethodEvent	正在使用输入法
QEvent.InputMethodQuery	207	QInputMethodQueryEvent	输入法查询事件
QEvent.KeyboardLayoutChange	169		键盘布局已更改
QEvent.KeyPress	6	QKeyEvent	键盘按下
QEvent.KeyRelease	7	QKeyEvent	键盘释放
QEvent.LanguageChange	89		应用程序翻译发生改变
QEvent.LayoutDirectionChange	90		布局的方向发生改变
QEvent.LayoutRequest	76		控件的布局需要重做
QEvent.Leave	11		鼠标离开控件的边界
QEvent.LeaveEditFocus	151		编辑控件失去编辑的焦点
QEvent.LeaveWhatsThisMode	125		程序离开“What’s This?”模式
QEvent.LocaleChange	88		系统区域设置发生改变
QEvent.NonClientAreaMouseMove	173		鼠标移动发生在客户区域外
QEvent.ModifiedChange	102		控件修改状态发生改变
QEvent.MouseButtonDblClick	4	QMouseEvent	鼠标再次按下
QEvent.MouseButtonPress	2	QMouseEvent	鼠标按下

QEvent.MouseButtonRelease	3	QMouseEvent	鼠标释放
QEvent.MouseMove	5	QMouseEvent	鼠标移动
QEvent.MouseTrackingChange	109		鼠标跟踪状态发生改变
QEvent.Move	13	QMoveEvent	控件的位置发生改变
QEvent.NativeGesture	197	QNativeGestureEvent	系统检测到手势
QEvent.Paint	12	QPaintEvent	需要屏幕更新
QEvent.PaletteChange	39		控件的调色板发生改变
QEvent.ParentAboutToChange	131		控件的 parent 将要更改
QEvent.ParentChange	21		控件的 parent 发生改变
QEvent.PlatformPanel	212		请求一个特定于平台的面板
QEvent.Polish	75		控件被抛光
QEvent.PolishRequest	74		控件应该被抛光
QEvent.ReadOnlyChange	106		控件 read-only 状态发生改变
QEvent.Resize	14	QResizeEvent	控件的大小发生改变
QEvent.ScrollPrepare	204	QScrollPrepareEvent	对象需要填充它的几何信息
QEvent.Scroll	205	QScrollEvent	对象需要滚动到提供的位置
QEvent.Shortcut	117	QShortcutEvent	快捷键处理
QEvent.ShortcutOverride	51	QKeyEvent	按下按键，用于覆盖快捷键
QEvent.Show	17	QShowEvent	控件显示在屏幕上
QEvent.ShowToParent	26		子控件被显示
QEvent.StatusTip	112	QStatusTipEvent	状态提示请求
QEvent.StyleChange	100		控件的样式发生改变
QEvent.TabletMove	87	QTabletEvent	Wacom 写字板移动
QEvent.TabletPress	92	QTabletEvent	Wacom 写字板按下
QEvent.TabletRelease	93	QTabletEvent	Wacom 写字板释放
QEvent.Timer	1	QTimerEvent	定时器事件
QEvent.ToolTip	110	QHelpEvent	一个 tooltip 请求
QEvent.ToolTipChange	184		控件的 tooltip 发生改变
QEvent.TouchBegin	194	QTouchEvent	触摸屏或轨迹板序列的开始
QEvent.TouchCancel	209	QTouchEvent	取消触摸事件序列
QEvent.TouchEnd	196	QTouchEvent	触摸事件序列结束
QEvent.TouchUpdate	195	QTouchEvent	触摸屏事件
QEvent.UngrabKeyboard	189	QGraphicsItem	Item 失去键盘抓取
QEvent.UngrabMouse	187		Item 失去鼠标抓取（QGraphicsItem QQuickItem）
QEvent.UpdateRequest	77		控件应该被重绘
QEvent.WhatsThis	111	QHelpEvent	控件显示“What’s This”帮助
QEvent.WhatsThisClicked	118		“What’s This”帮助链接被单击

QEvent.Wheel	31	QWheelEvent	鼠标滚轮滚动
QEvent.WindowActivate	24		窗口已激活
QEvent.WindowBlocked	103		窗口被模式对话框阻塞
QEvent.WindowDeactivate	25		窗户被停用
QEvent.WindowIconChange	34		窗口的图标发生改变
QEvent.WindowStateChange	105	QWindowStateChangeEvent	窗口的状态（最小化、最大化或全屏）发生改变
QEvent.WindowTitleChange	33		窗口的标题发生改变
QEvent.WindowUnblocked	104		一个模式对话框退出后，窗口将不被阻塞
QEvent.WinIdChange	203		窗口的系统标识符发生改变

10.1.3 event()函数

对于 GUI 应用程序，当捕捉到事件发生后，会首先发送到 QWidget 或子类的 event(QEvent)函数中进行数据处理，如果没有重写 event()函数进行事件处理，事件将会分发到事件默认的处理函数中，因此 event()函数是事件的集散地。如果重写了 event()函数，当 event()函数的返回值是 True 时，表示事件已经处理完毕，事件不会再发送给其他处理函数；当 event()函数的返回值是 False 时，表示事件还没有处理完毕。event()函数可以截获某些类型的事件，并处理事件。

下面的程序是将 10.1.2 小节中的例子做了改动，将鼠标的单击事件放到 event()函数中进行处理，只截获 QEvent.MouseButtonPress 事件，通过 super()函数调用父类的 event()函数，其他类型的事件仍交由 QWidget 的 event()函数处理和分发。

```
import sys    #Demo10_2.py
from PyQt5.QtWidgets import    QApplication,QWidget,QLineEdit
from PyQt5.QtCore import QEvent,Qt

class MyWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.resize(500,500)
        self.lineEdit=QLineEdit(self)
        self.lineEdit.setGeometry(0,0,500,30)
    def event(self, even):    #重写 event 函数
        if even.type() == QEvent.MouseButtonPress:    #按键的情况
            template1 = "单击点的窗口坐标是 x:{} y:{}"
            template2 = "单击点的屏幕坐标是 x:{} y:{}"
            if even.button() == Qt.LeftButton:     #按左键的情况
                string = template1.format(even.x(), even.y())
                self.lineEdit.setText(string)
```

```
                    return True
                elif even.button() == Qt.RightButton:   #按右键的情况
                    string = template2.format(even.globalX(), even.globalY())
                    self.lineEdit.setText(string)
                    return True
                else:   #按中键的情况
                    return True
            else:    #对于不是按鼠标键的事件，交给 QWidget 来处理
                finished = super().event(even)   #super()函数调用父类函数
                return finished

if __name__ == '__main__':
    app=QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

10.2 鼠标和键盘事件的类

键盘事件和鼠标事件是用得最多的事件，通过键盘和鼠标事件可以拖拽控件、弹出快捷菜单。

10.2.1 鼠标按键事件类

鼠标按键事件类 QMouseEvent 涉及鼠标按键的单击、释放和鼠标移动操作，与 QMouseEvent 关联的事件类型有 QEvent.MouseButtonDblClick、QEvent.MouseButtonPress、QEvent.MouseButtonRelease 和 QEvent.MouseMove。鼠标滚轮的滚动事件类是 QWheelEvent。当在一个窗口中按住鼠标按键或释放按键就会产生鼠标事件 QMouseEvent ，鼠标移动事件只会在按下鼠标按键的情况下才会发生，除非通过显式调用窗口的 setMouseTracking(bool)函数来开启鼠标轨迹跟踪，这种情况下只要鼠标指针移动，就会产生一系列鼠标事件。处理 QMouseEvent 类鼠标事件的函数有 mouseDoubleClickEvent(QMouseEvent)（双击鼠标按键）、mouseMoveEvent(QMouseEvent)（移动鼠标）mousePressEvent(QMouseEvent)（按下鼠标按键）mouseReleaseEvent(QMouseEvent)（释放鼠标按键），处理 QWheelEvent 类滚轮事件的函数是 wheelEvent(QWheelEvent)。

1. QMouseEvent 类的方法

当产生鼠标事件时，会生成 QMouseEvent 类的实例对象，并将实例对象作为实参传递给相关的处理函数。QMouseEvent 类包含了用于描述鼠标事件的参数。QMouseEvent 类在 QtGui 模块中，它的主要方法如表 10-3 所示。

- 用 button()方法可以获取引起鼠标事件的按钮，buttons()方法获取产生鼠标事件时被按住的按钮，返回值可以是 Qt.NoButton、Qt.AllButtons、Qt.LeftButton、Qt.RightButton、Qt.MidButton、Qt.MiddleButton、Qt.BackButton、Qt.ForwardButton、Qt.TaskButton 和 Qt.ExtraButtoni（i=1，

2，…，24）。

- 用 source()方法可以获取鼠标事件的来源，返回值可以是 Qt.MouseEventNotSynthesized（来自鼠标）、Qt.MouseEventSynthesizedBySystem（来自鼠标和触摸屏）、Qt.MouseEventSynthesizedByQt（来自触摸屏）和 Qt.MouseEventSynthesizedByApplication（来自应用程序）。
- 产生鼠标事件的同时，有可能按下了键盘上 Ctrl、Shitt 或 Alt 等修饰键，用 modifiers()方法可以获取这些键。modifiers()方法的返回值可以是 Qt.NoModifier（没有修饰键）、Qt.ShiftModifier（Shift 键）、Qt.ControlModifier（Ctrl 键）、Qt.AltModifier（Alt 键）、Qt.MetaModifier（Meta 键，Windows 系统为 window 键）、Qt.KeypadModifier（小键盘上的键）和 Qt.GroupSwitchModifier（Mode_switch 键）。

表 10-3 QMouseEvent 类的方法

方 法	返回值的类型	说 明
button()	Qt.MouseButton	获取产生鼠标事件的按键
buttons()	Qt.MouseButtons	获取产生鼠标事件时被按下的按键
flags()	Qt.MouseEventFlags	获取鼠标事件的标识
source()	Qt.MouseEventSource	获取鼠标事件的来源
modifiers()	Qt.KeyboardModifiers	获取修饰键
globalPos()	QPoint	获取全局的鼠标位置
globalX()	int	获取全局的 X 坐标
globalY()	int	获取全局的 Y 坐标
localPos()	QPointF	获取局部鼠标位置
screenPos()	QPointF	获取屏幕的鼠标位置
windowPos()	QPointF	获取相对于接受事件窗口的鼠标位置
pos()	QPoint	获取相对于控件的鼠标位置
x()	int	获取相对于控件的 X 坐标
y()	int	获取相对于控件的 Y 坐标

2. QWheelEvent 类的方法

QWheelEvent 类处理鼠标的滚轮事件，其方法如表 10-4 所示，大部分方法与 QMouseEvent 的方法相同，主要不同的方法如下所述。

- angleDelta().y()返回两次事件之间鼠标竖直滚轮旋转的角度，angleDelta().x()返回两次事件之间水平滚轮旋转的角度。如果没有水平滚轮，angleDetal().x()的值为 0，正数值表示滚轮相对于用户在向前滑动，负数值表示滚轮相对于用户是向后滑动。
- pixelDelta()方法返回两次事件之间控件在屏幕上的移动距离（单位是像素）。
- inverted()方法将 angleDelta()和 pixelDelta()的值与滚轮转向之间取值关系反向，即正数值表示滑轮相对于用户在向后滑动；负数值表示滑轮相对于用户在向前滑动。
- phase()方法返回设备的状态，返回值有 Qt.NoScrollPhase（不支持滚动）、Qt.ScrollBegin（开始位置）、Qt.ScrollUpdate（处于滚动状态）、Qt.ScrollEnd（结束位置）和 Qt.ScrollMomentum（不触碰设备，由于惯性仍处于滚动状态）。

表 10-4 QWheelEvent 类的方法

方 法	返回值的类型	方 法	返回值的类型
angleDelta()	QPoint	globalPosition()	QPointF
pixelDelta()	QPoint	globalX()	int
phase()	Qt.ScrollPhase	globalY()	int
inverted()	bool	pos()	QPoint
source()	Qt.MouseEventSource	posF()	QPointF
buttons()	Qt.MouseButtons	position()	QPointF
globalPos()	QPoint	x()	int
globalPosF()	QPointF	y()	int
modifiers()	Qt.KeyboardModifiers		

3. QMouseEvent 类和 QWheelEvent 类的应用实例

下面的程序涉及鼠标单击、拖拽、双击和滚轮滚动的事件，双击窗口的空白处，弹出打开图片的对话框，选择图片后，显示出图片，按住鼠标按键并拖动鼠标可以移动图片，滚动滚轮可以缩放图片。程序中通过控制绘图区域的中心位置来移动图像，通过控制图像区域的宽度和高度来缩放图像。

```
import sys   #Demo10_3.py
from PyQt5.QtWidgets import QApplication,QWidget,QFileDialog,QMenuBar
from PyQt5.QtGui import QPixmap,QPainter
from PyQt5.QtCore import QRect,QPoint

class MyWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.resize(600,600)
        self.pixmap = QPixmap()   #创建 QPixmap 图象
        self.pix_width = 0   #获取初始宽度
        self.pix_height = 0   #获取初始高度
        self.translate_x = 0   #用于控制 x 向平移
        self.translate_y = 0   #用于控制 y 向平移
        self.start=QPoint(0,0)   #鼠标单击时光标位置
        # 记录图像中心的变量，初始定义在窗口的中心
        self.center = QPoint(int(self.width() / 2), int(self.height() / 2))
        menuBar = QMenuBar(self)
        menuFile = menuBar.addMenu("文件(&F)")
        menuFile.addAction("打开(&O)").triggered.connect(self.actionOpen_triggered) #与槽连接
        menuFile.addSeparator()
        menuFile.addAction("退出(&E)").triggered.connect(self.close)   #动作与槽连接
    def paintEvent(self,event): #窗口绘制处理函数，当窗口刷新时调用该函数
        self.center = QPoint(self.center.x() + self.translate_x, self.center.y() + self.translate_y)
```

```
            #图像绘制区域的左上角点，用于缩放图像
            point_1 = QPoint(self.center.x() - self.pix_width, self.center.y() - self.pix_height)
            # 图像绘制区域的右下角点，用于缩放图像
            point_2 = QPoint(self.center.x() + self.pix_width, self.center.y() + self.pix_height)
            self.rect = QRect(point_1, point_2)   #图像绘制区域
            painter = QPainter(self)   # 绘图
            painter.drawPixmap(self.rect,self.pixmap)
        def mousePressEvent(self, event):   #鼠标按键按下事件的处理函数
            self.start=event.pos()   #鼠标位置
        def mouseMoveEvent(self,event): #鼠标移动事件的处理函数
            self.translate_x = event.x()-self.start.x()   #鼠标的移动量
            self.translate_y = event.y()-self.start.y()   #鼠标的移动量
            self.start = event.pos()
            self.update()
        def wheelEvent(self,event): #鼠标滚轮事件的处理函数
            self.pix_width = self.pix_width + int(event.angleDelta().y()/10)
            self.pix_height = self.pix_height + int(event.angleDelta().y()/10)
            self.update()
        def mouseDoubleClickEvent(self, event): #双击鼠标事件的处理函数
            self.actionOpen_triggered()
        def actionOpen_triggered(self):   #打开文件的动作
            fileDialog = QFileDialog(self)
            fileDialog.setNameFilter("图像文件(*.png *.jpeg *.jpg)")
            fileDialog.setFileMode(QFileDialog.ExistingFile)
            if fileDialog.exec():
                self.pixmap.load(fileDialog.selectedFiles()[0])
                self.pix_width = int(self.pixmap.width() / 2)   # 获取初始宽度
                self.pix_height = int(self.pixmap.height() / 2)   # 获取初始高度
                self.update()
if __name__ == '__main__':
    app=QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

10.2.2 键盘事件类

键盘事件类 QKeyEvent 涉及键盘键的按下和释放，与 QKeyEvent 关联的事件类型有 QEvent.KeyPress、QEvent.KeyRelease 和 QEvent.ShortcutOverride，处理键盘事件的函数是

keyPressEvent(QKeyEvent)和 keyReleaseEvent(QKeyEvent)。当发生键盘事件时，将创建 QKeyEvent 的实例对象，并将实例对象作为实参传递给处理函数。

键盘事件类 QKeyEvent 的常用方法如表 10-5 所示，主要方法介绍如下。

- 如果同时按下多个键，可以用 count()方法获取按键的数量。
- 如果按下一个键不放，将连续触发键盘事件，用 isAutoRepeat()方法可以获取某个事件是否是重复事件。
- 用 key()方法可以获取按键的 Qt.key 代码值，不区分大小写；可以用 text()方法获取按键的字符，区分大小写。
- 用 matches(QKeySequence.StandardKey)方法可以判断按下的键是否匹配标准的按钮，QKeySequence.StandardKey 中定义了常规的标准按键，例如 Ctrl+C 表示复制、Ctrl+V 表示粘贴、Ctrl+S 表示保存、Ctrl+O 表示打开、Ctrl+W 或 Ctrl+F4 表示关闭。

表 10-5 键盘事件类的常用方法

QKeyEvent 的方法	返回值的类型	说　明
count()	int	获取按键的数量
isAutoRepeat()	bool	获取是否是重复事件
key()	int	获取按键的代码
matches(QKeySequence.StandardKey)	bool	如果按键匹配标准的按钮，返回 True
modifiers()	Qt.KeyboardModifiers	获取修饰键
text()	str	返回按键上的字符

10.2.3 鼠标拖放事件类

可视化开发中经常会用鼠标拖放动作来完成一些操作，例如把一个 docx 文档拖到 word 中直接代开文件，把图片拖放到一个图片浏览器中打开图片，拖放一段文字到其他位置等。拖放事件包括鼠标进入、鼠标移动和鼠标释放事件，还可以有鼠标移出事件，对应的事件类型分别是 QEvent.DragEnter、QEvent.DragMove、QEvent.Drop 和 QEvent.DragLeave。拖放事件类分别为 QDragEnterEvent、QDragMoveEvent、QDropEvent 和 QDragLeaveEvent，其实例对象中保存着拖放信息，例如被拖放文件路径、被拖放的文本等。拖放事件的处理函数分别是 dragEnterEvent(QDragEnterEvent)、dragMoveEvent(QDragMoveEvent)、dropEvent(QDropEvent)和 dragLeaveEvent(QDragLeaveEvent)。

1. QDragEnterEvent、QDragMoveEvent、QDropEvent 和 QDragLeaveEvent 的方法

QDragEnterEvent 类是从 QDropEvent 类和 QDrageMoveEvent 类继承而来的，它没有自己特有的方法，QDragMoveEvent 类是从 QDropEvent 类继承而来的，它继承了 QDropEvent 类的方法，又添加了自己新的方法，QDragLeaveEvent 类是从 QEvent 类继承而来的，它没有自己特有的方法。

QDropEvent 类和 QDrageMoveEvent 类的方法分别如表 10-6 和表 10-7 所示，主要方法介绍如下。

- 要使一个控件或窗口接受拖放，必须用 setAcceptDrops(True)方法设置成接受拖放，在进入事件的处理函数 dragEnterEvent(QDragEnterEvent)中，需要把事件对象设置成 accept()，否则无法接受后续的移动和释放事件。

- 在拖放事件中，用 mimeData()方法获取被拖放物体的 QMimeData 数据，MIME（multipurpose internet mail extensions）是多用途互联网邮件扩展类型，关于 QMimeData 的介绍参见下面的内容。
- 在释放动作中，被拖拽的物体可以从原控件中被复制或移动到目标控件中，复制或移动动作可以通过 setDropAction(Qt.DropAction)来设置，其中 Qt.DropAction 可以取 Qt.CopyAction（复制）、Qt.MoveAction（移动）、Qt.LinkAction（链接）、Qt.IgnoreAction（什么都不做）或 Qt.TargetMoveAction（目标对象接管），另外系统也会推荐一个动作，可以用 proposedAction()方法获取推荐的动作，用 possibleActions()方法获取有可能实现的动作，用 dropAction()方法获取采取的动作。

表 10-6 QDropEvent 类的方法

QDropEvent 的方法	返回值的类型	说　明
keyboardModifiers()	Qt.KeyboardModifiers	获取修饰键
mimeData()	QMimeData	获取 mime 数据
mouseButtons()	Qt.MouseButtons	获取按下的鼠标按钮
pos()	QPoint	获取释放时的位置
posF()	QPointF	
dropAction()	Qt.DropAction	获取采取的动作
possibleActions()	Qt.DropActions	获取可能的动作
proposedAction()	Qt.DropAction	系统推荐的动作
acceptProposedAction()		接受推荐的动作
setDropAction(Qt.DropAction)		设置释放动作
source()	QObject	获取被拖对象

表 10-7 QDragMoveEvent 类的方法

QDragMoveEvent 的方法	说　明
accept()	在控件或窗口的边界内都可接受移动事件
accept(QRect)	在指定的区域内接受移动事件
answerRect()	返回可以释放的区域 Qrect
ignore()	在整个边界内部忽略移动事件
ignore(QRect)	在指定的区域内部忽略移动事件

2. QMimeData 类

QMimeData 类用于描述存放到粘贴板上的数据，并通过拖放事件传递粘贴板上的数据，从而在不同的程序间传递数据，也可以在同一个程序内传递数据。创建 QMimeData 实例对象的方法是 QMimeData()，它在 QtCore 模块中。

QMimeData 可以存储的数据有文本、图像、颜色和地址等。QMimeData 的方法如表 10-8 所示，可以分项设置和获取数据，也可以用 setData(str,QByteArray)方法设置数据。QMimeData 的数据格式、各种数据设置和获取的方法如表 10-9 所示。

表 10-8 QMimeData 类的方法

方法及参数类型	返回值的类型	说　明

formats()	List[str]	获取格式列表
hasFormat(str)	bool	获取是否有某种格式
removeFormat(str)		移除格式
setColorData(Any)		设置颜色数据
hasColor()	bool	获取是否有颜色数据
colorData()	Any	获取颜色数据
setHtml(str)		设置 Html 数据
hasHtml()	bool	判断是否有 Html 数据
html()	str	获取 Html 数据
setImageData(Any)		设置图像数据
hasImage()	bool	获取是否有图像数据
imageData()	Any	获取图像数据
setText(str)		设置文本数据
hasText()	bool	判断是否有文本数据
text()	str	获取文本数据
setUrls(Iterable[QUrl])		设置 Url 数据
hasUrls()	bool	判断是否有 Url 数据
urls()	List[QUrl]	获取 Url 数据
setData(str,QByteArray)		设置某种格式的数据
data(str)	QByteArray	获取某种格式的数据
clear()		清空格式和数据

表 10-9 QMimeData 的数据格式和数据方法

格　式	是否存在	获取方法	设置方法	举　例
text/plain	hasText()	text()	setText()	setText("拖动文本")
text/html	hasHtml()	html()	setHtml()	setHtml("拖动文本</ b>")
text/uri-list	hasUrls()	urls()	setUrls()	setUrls ([QUrl("www.qq.com/")])
image/ *	hasImage()	imageData()	setImageData()	setImageData(QImage("ix.png"))
application/x-color	hasColor()	colorData()	setColorData()	setColorData(QColor(23,56,53))

3. 拖放事件的应用实例

下面的程序是在上一个实例的基础上增加了拖拽功能，除了可以双击窗口、用菜单打开一个图像文件外，也可以把一个图像文件拖拽到窗口上打开。

```
import sys    #Demo10_4.py
from PyQt5.QtWidgets import QApplication,QWidget,QFileDialog,QMenuBar
from PyQt5.QtGui import QPixmap,QPainter
from PyQt5.QtCore import QRect,QPoint


class MyWindow(QWidget):
    def __init__(self,parent=None):
```

```
        super().__init__(parent)
        self.setAcceptDrops(True) #设置可接受拖放事件
        self.resize(600,600)
        self.pixmap = QPixmap()  #创建 QPixmap 图像
        self.pix_width = 0  #获取初始宽度
        self.pix_height = 0  #获取初始高度
        self.translate_x = 0  #用于控制 x 向平移
        self.translate_y = 0  #用于控制 y 向平移
        self.start=QPoint(0,0)  #鼠标单击时光标位置
        self.center = QPoint(int(self.width() / 2), int(self.height() / 2))
        menuBar = QMenuBar(self)
        menuFile = menuBar.addMenu("文件(&F)")
        menuFile.addAction("打开(&O)").triggered.connect(self.actionOpen_triggered) #动作与槽
        menuFile.addSeparator()
        menuFile.addAction("退出(&E)").triggered.connect(self.close)  #动作与槽连接
    def paintEvent(self,event):  #窗口绘制处理函数，当窗口刷新时调用该函数
        self.center = QPoint(self.center.x() + self.translate_x, self.center.y() + self.translate_y)
        point_1 = QPoint(self.center.x() - self.pix_width, self.center.y() - self.pix_height)
        point_2 = QPoint(self.center.x() + self.pix_width, self.center.y() + self.pix_height)
        self.rect = QRect(point_1, point_2)  #图像绘制区域
        painter = QPainter(self)  # 绘图
        painter.drawPixmap(self.rect,self.pixmap)
    def mousePressEvent(self, event):  #鼠标按键按下事件的处理函数
        self.start=event.pos()  #鼠标位置
    def mouseMoveEvent(self,event): #鼠标移动事件的处理函数
        self.translate_x = event.x()-self.start.x()  #鼠标的移动量
        self.translate_y = event.y()-self.start.y()  #鼠标的移动量
        self.start = event.pos()
        self.update()
    def wheelEvent(self,event): #鼠标滚轮事件的处理函数
        self.pix_width = self.pix_width + int(event.angleDelta().y()/10)
        self.pix_height = self.pix_height + int(event.angleDelta().y()/10)
        self.update()
    def mouseDoubleClickEvent(self, event): #双击鼠标事件的处理函数
        self.actionOpen_triggered()
    def actionOpen_triggered(self):  #打开文件的动作
        fileDialog = QFileDialog(self)
        fileDialog.setNameFilter("图像文件(*.png *.jpeg *.jpg)")
        fileDialog.setFileMode(QFileDialog.ExistingFile)
```

```
            if fileDialog.exec():
                self.pixmap.load(fileDialog.selectedFiles()[0])
                self.pix_width = int(self.pixmap.width() / 2)   # 获取初始宽度
                self.pix_height = int(self.pixmap.height() / 2)   # 获取初始高度
                self.update()
        def dragEnterEvent(self,event):   #拖动进入事件
            if event.mimeData().hasUrls():
                event.accept()
            else:
                event.ignore()
        def dropEvent(self,event):   #释放事件
            urls = event.mimeData().urls()   #获取被拖动文件的地址列表
            fileName = urls[0].path()     #将文件地址转成本地地址
            self.pixmap.load(fileName)
            self.pix_width = int(self.pixmap.width() / 2)   # 获取初始宽度
            self.pix_height = int(self.pixmap.height() / 2)
            self.update()
    if __name__ == '__main__':
        app=QApplication(sys.argv)
        window = MyWindow()
        window.show()
        sys.exit(app.exec())
```

4. QDrag 类

如果要在程序内部拖放控件，需要先把控件定义成可移动控件，可移动控件需要在其内部定义 QDrag 的实例对象。QDrag 类用于拖放物体，它继承自 QObject 类，创建 QDrag 实例对象的方法是 QDrag(QObject)，参数 QObject 表示只要是从 QObject 类继承的控件都可以。

QDrag 类的方法如表 10-10 所示，主要方法介绍如下。

- 创建 QDrag 实例对象后，用 exec(supportedActions)或 exec(supportedActions,defaultAction)方法开启拖放，参数是拖放事件支持的动作和默认动作。
- 用 setMimeData(QMimeData)方法设置 MIME 对象，传递数据，用 mimeData()方法获取 MIME 数据。
- 用 setPixmap(QPixmap)方法设置拖拽时显示的图像，用 setDragCursor(QPixmap,Qt.DropAction)设置鼠标的光标形状。
- 用 setHotSpot(QPoint)方法设置热点位置。热点位置是拖拽过程中，光标相对于控件左上角的位置。
- 为了防止误操作，可以用 QApplication 的 setStartDragDistance(int)方法和 setStartDragTime(msec)方法设置拖动开始一定距离或一段时间后才开始进行拖放事件。
- QDrag 有两个信号 actionChanged(Qt.DropAction)和 targetChanged(QObject)。

表 10-10 QDrag 类的方法

方法及参数类型	返回值的类型	说　明
exec(Qt.DropActions)	Qt.DropAction	开始拖动操作，并返回释放时的动作
exec(Qt.DropActions, Qt.DropAction)	Qt.DropAction	
defaultAction()	Qt.DropAction	返回默认的释放动作
setDragCursor(QPixmap,Qt.DropAction)		设置拖动时光标形状
dragCursor(Qt.DropAction)	QPixmap	获取拖动时的光标形状
setHotSpot(QPoint)		设置热点位置
hotSpot()	QPoint	获取热点位置
setMimeData(QMimeData)		设置拖放中传输的数据
mimeData()	QMimeData	获取数据
setPixmap(QPixmap)		设定拖动时鼠标显示的图像
pixmap()	QPixmap	获取图像
source()	QObject	返回被拖放物体的父控件
target()	QObject	返回目标控件
supportedActions()	Qt.DropActions	获取支持的动作
cancel()		取消拖放

下面实例先重写了 QPushButton 的 mousePressEvent()事件，在该事件中定义了 QDrag 的实例，这样 QPushButton 的实例对象就是可移动控件，然后又重新定义了 QFrame 框架，在内部定义了两个 QPushButton，重写了 dragEnterEvent()函数、dragMoveEvent()函数和 dropEvent()函数。程序运行后，可以随机用鼠标左键移动按钮的位置。

```
import sys   #Demo10_5.py
from PyQt5.QtWidgets import QApplication,QWidget,QPushButton,QFrame,QHBoxLayout
from PyQt5.QtGui import QDrag
from PyQt5.QtCore import QPoint,QMimeData,Qt

class myPushButton(QPushButton):
    def __init__(self, parent=None):
        super().__init__(parent)
    def mousePressEvent(self, event):   #按键事件
        if event.button() == Qt.LeftButton:
            drag = QDrag(self)
            drag.setHotSpot(event.pos()-self.rect().topLeft())
            mime = QMimeData()
            drag.setMimeData(mime)
            drag.exec()
class myFame(QFrame):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setAcceptDrops(True)
```

```
            self.setFrameShape(QFrame.Box)
            self.btn_1 = myPushButton(self)
            self.btn_1.setText("push button 1")
            self.btn_1.move(100,100)
            self.btn_2 = myPushButton(self)
            self.btn_2.setText("push button 2")
            self.btn_2.move(200,200)
        def dragEnterEvent(self,event):
            self.child = self.childAt(event.pos())    #获取指定位置的控件
            event.accept()
        def dragMoveEvent(self,event):
            if self.child:
                center = QPoint(int(self.child.width() / 2), int(self.child.height() / 2))
                self.child.move(event.pos() - center)
        def dropEvent(self,event):
            if self.child:
                center = QPoint(int(self.child.width() / 2), int(self.child.height() / 2))
                self.child.move(event.pos() - center)
class myWindow(QWidget):
        def __init__(self,parent=None):
            super().__init__(parent)
            self.setupUi()
            self.resize(600,400)
            self.setAcceptDrops(True)
        def setupUi(self):
            self.frame_1= myFame(self)
            self.frame_2 =myFame(self)
            H = QHBoxLayout(self)
            H.addWidget(self.frame_1)
            H.addWidget(self.frame_2)
if __name__ == '__main__':
        app=QApplication(sys.argv)
        window = myWindow()
        window.show()
        sys.exit(app.exec())
```

10.2.4 上下文菜单

1. 上下文菜单类的方法

上下文菜单通常通过单击鼠标右键后弹出。上下文菜单的事件类型是 QEvent.ContextMenu，处理函数是 contextMenuEvent (QContextMenuEvent)，其中上下文菜单类 QContextMenuEvent 的方法如表 10-11 所示。主要方法介绍如下。

表 10-11 上下文菜单类的方法

方　法	返回值的类型	说　明
globalPos()	QPoint	光标的全局坐标点
globalX()	int	全局坐标的 X 值
globalY()	int	全局坐标的 Y 值
pos()	QPoint	局部坐标点
x()	int	局部坐标的 x 值
y()	int	局部坐标的 y 值
reason()	QContextMenuEvent.Reason	上下文菜单产生的原因
modifiers()	Qt.KeyboardModifiers	获取修饰键

- 用 globalPos()方法、globalX()方法和 globalY()方法可以获得单击鼠标右键时的全局坐标位置，用 pos()方法、x()方法和 y()方法可以获得窗口的局部坐标点。
- 用 reason()方法可以获得产生上下文菜单的原因，返回值是 QContextMenuEvent.Reason 的枚举值，可能是 QContextMenuEvent.Mouse 、 QContextMenuEvent.Keyboard 和 QContextMenuEvent.Other，值分别是 0、1 和 2，分别表示上下文菜单来源于鼠标、键盘（Windows 系统是菜单键）或除鼠标键盘之外的其他情况。
- 在 contextMenuEvent (QContextMenuEvent)处理函数中，用菜单的 exec(QPoint)方法在指定位置显示菜单，菜单可以是在其他位置已经定义好的菜单，也可以是在处理函数中临时定义的。
- 只有在窗口或控件的 contextMenuPlolicy 属性为 Qt.DefaultContextMenu 时，单击鼠标右键才会执行处理函数，通常情况下 Qt.DefaultContextMenu 是默认值。如果不想弹出右键菜单，可以通过方法 setContextMenuPolicy(Qt.ContextMenuPolicy)将该属性设置为其他值，Qt.ContextMenuPolicy 的取值如表 10-12 所示。

表 10-12 Qt.ContextMenuPolicy 的取值

Qt.ContextMenuPolicy 取值	值	说　明
Qt.NoContextMenu	0	控件不具有上下文菜单，上下文菜单被推到控件的父窗口
Qt.DefaultContextMenu	1	控件或窗口的 contextMenuEvent()被调用
Qt.ActionsContextMenu	2	将控件 actions()方法返回的 QActions 当作上下文菜单项，单击鼠标右键后显示该菜单
Qt.CustomContextMenu	3	控件发射 customContextMenuRequested(Qpoint)信号，如果

		要自定义菜单，用这个枚举值，并自定义一个处理函数
Qt.PreventContextMenu	4	控件不具有上下文菜单，所有的鼠标右键事件都传递到 mousePressEvent()和 mouseReleaseEvent()函数

2. 上下文菜单应用实例

下面的程序建立一个空白窗口，在窗口单击鼠标右键，弹出上下文菜单，然后选择打开项，选择一幅图片后，在窗口上显示该图片。

```
import sys    #Demo10_6.py
from PyQt5.QtWidgets import QApplication,QWidget,QFileDialog,QMenu
from PyQt5.QtGui import QPixmap,QPainter

class myWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setAcceptDrops(True) #设置可接受拖放事件
        self.resize(600,400)
        self.pixmap = QPixmap()    #创建 QPixmap 图像
    def contextMenuEvent(self,event) :
        contextMenu = QMenu(self)
        contextMenu.addAction("打开(&O)").triggered.connect(self.actionOpen_triggered) #槽连接
        contextMenu.addSeparator()
        contextMenu.addAction("退出(&E)").triggered.connect(self.close)    # 动作与槽连接
        contextMenu.exec(event.globalPos())
    def paintEvent(self,event): #窗口绘制处理函数，当窗口刷新时调用该函数
        painter = QPainter(self)    # 绘图
        painter.drawPixmap(self.rect(),self.pixmap)
    def mouseDoubleClickEvent(self, event): #双击鼠标事件的处理函数
        self.actionOpen_triggered()
    def actionOpen_triggered(self):    #打开文件的动作
        fileDialog = QFileDialog(self)
        fileDialog.setNameFilter("图像文件(*.png *.jpeg *.jpg)")
        fileDialog.setFileMode(QFileDialog.ExistingFile)
        if fileDialog.exec():
            self.pixmap.load(fileDialog.selectedFiles()[0])
            self.update()
if __name__ == '__main__':
    app=QApplication(sys.argv)
    window = myWindow()
    window.show()
    sys.exit(app.exec())
```

10.2.5 剪贴板

剪贴板 QClipboard 类似于拖放，可以在不同的程序间用复制和粘贴操作来传递数据。QClipboard 位于 QtGui 模块中，继承自 QObject 类，用 QClipboard(parent=None)方式可以创建剪贴板对象。

可以直接往剪贴板中复制文本数据、QPixmap 和 QImage，其他数据类型可以通过 QMimeData 来传递数据，QClipboard 的常用方法如表 10-13 所示。

表 10-13 QClipboard 的常用方法

剪贴板的方法及参数类型	返回值的类型	说　明
setText(str)		将文本复制到剪贴板
text()	str	从剪贴板上获取文本
text(str)	Tuple[str,str]	从 str 指定的数据类型上获取文本，数据类型如 plain 或 html
setPixmap(QPixmap)		将 QPixmap 图像复制到剪贴板上
pixmap()	QPixmap	从剪贴板上获取 QPixmap 图像
setImage(QImage)		将 QImage 图像复制到剪贴板上
image()	QImage	从剪贴板上获取 QImage 图像
setMimeData(QMimeData)		将 QMimeData 数据赋值到剪贴板上
mimeData()	QMimeData	从剪贴板上获取 QMimeData 数据
clear()		清空剪贴板

剪贴板的主要信号是 dataChanged()，当剪贴板上的数据发生变化时发射该信号。

10.3 窗口常用事件

窗口常用事件涉及到窗口或控件的隐藏、显示、移动、缩放、重绘、关闭、获得和失去焦点等，通常需要重写这些事件的处理函数，以便达到特定的目的。

10.3.1 显示和隐藏事件

在用 show()方法或 setVisible(True)方法显示一个顶层窗口之前会发生 QEvent.Show 事件，调用 showEvent(QShowEvent)处理函数，显示事件类 QShowEvent 只有从 QEvent 继承的属性，没有自己特有的属性。在用 hide()方法或 setVisible(False)方法隐藏一个顶层窗口之前会发生 QEvent.Hide 事件，调用 hideEvent(QHideEvent)处理函数，隐藏事件类 QHideEvent 只有从 QEvent 继承的属性，没有自己特有的属性。利用显示和隐藏事件的处理函数，可以在窗口显示之前或被隐藏之前做一些预处理工作。

10.3.2 缩放和移动事件

当一个窗口或控件的宽度和高度发生改变时会触发 QEvent.Resize 事件，调用

resizeEvent(QResizeEvent)处理函数。缩放事件类 QResizeEvent 只有两个方法 oldSize()和 size()方法，分别返回缩放前和缩放后的窗口尺寸 QSize。

当改变一个窗口或控件的位置时会触发 QEvent.Move 事件，调用 moveEvent(QMoveEvent)处理函数。移动事件类 QMoveEvent 只有两个方法 oldPos()和 pos()方法，分别返回窗口左上角移动前和移动后的位置 QPoint。

10.3.3 绘制事件

绘制事件是窗体系统产生的，在一个窗口首次显示、隐藏后又显示、缩放窗口、移动控件，以及调用 update()、repaint()、resize()方法时都会触发 QEvent.Paint 事件，绘制事件发生时，会调用 paintEvent(QPaintEvent)处理函数，该函数是受保护的，不能直接用代码调用此函数，通常在 paintEvent(QPaintEvent)处理函数中处理一些与绘图、显示有关的事情。

绘制事件类 QPaintEvent 只有两个方法 rect()和 region()方法，分别返回被重写绘制的矩形区域 QRect 和裁剪区域 QRegion。

在前面内容中多次用到 paintEvent()处理函数，读者可参考前面的实例。

10.3.4 进入和离开事件

当鼠标的光标进入窗口时，会触发 QEvent.Enter 进入事件，进入事件的处理函数是 enterEvent(QEvent)；当鼠标的光标离开窗口时，会触发 QEvent.Leave 离开事件，离开事件的处理函数是 leaveEvent(QEvent)。可以重写这两个函数，以达到特定的目的。

10.3.5 获得和失去焦点事件

一个控件获得键盘焦点时，可以接受键盘的输入。控件获得键盘焦点的方法很多，例如按 Tab 键、鼠标、快捷键等。当一个控件获得和失去键盘输入焦点时，会触发 focusIn 和 focusOut 事件，这两个事件的处理函数分别是 focusInEvent(QFocusEvent)和 focusOutEvent(QFocusEvent)，焦点事件类 QFocusEvent 的方法有 gotFocus()、lostFocus()和 reason()。当事件类型 type()的值是 QEvent.FocusIn 时，gotFocus()方法的返回值是 True，当事件类型 type()的值是 QEvent.FocusOut 时，lostFocus()方法的返回值是 True；reason()方法返回获得焦点图像，其返回值的类型是 Qt.FocusReason，其值有 Qt.MouseFocusReason、Qt.TabFocusReason、Qt.BacktabFocusReason、Qt.ActiveWindowFocusReason、Qt.PopupFocusReason、Qt.ShortcutFocusReason、Qt.MenuBarFocusReason 和 Qt.OtherFocusReason。

10.3.6 关闭事件

当用户单击窗口右上角的×按钮或执行窗口的 close()方法时，会触发 QEvent.Close 事件，调用 closeEvent(QCloseEvent)处理该事件。如果事件用 ignore()方法忽略了，则什么也不会发生；如果事件用 accept()方法接收了，首先窗口被隐藏，如果窗口设置了 setAttibute(Qt.WA_DeleteOnClose,True)属性的情况下，窗口会被删除。窗口事件类 QCloseEvent 没有特殊的属性，只有从 QEvent 继承来的方

法。

10.3.7 计时器事件

从 QObject 类继承的窗口和控件都会有 startTimer(int,timerType = Qt.CoarseTimer)方法和 killTimer(int)方法。startTimer()方法会启动一个计时器，并返回计时器的 ID 号，如果不能启动计时器，则返回值是 0，参数 int 是计时器的事件间隔，单位是毫秒，timerType 是计时器的类型，可以取 Qt.PreciseTimer、Qt.CoarseTimer 或 Qt.VeryCoarseTimer。窗口或控件可以用 startTimer()方法启动多个计时器，启动计时器后，会触发 timerEvent(QTimerEvent)事件，QTimerEvent 是计时器事件类。用 QTimerEvent 的 timerId()方法可以获取触发计时器事件的计时器 ID；用 killTimer(int)方法可以停止计时器，参数是计时器的 ID。

下面的程序启动窗口上的两个计时器，这两个计时器的时间间隔不同，用计时器事件输出是哪个计时器触发了计时器事件，可用按钮停止计时器。

```
import sys   #Demo10_7.py
from PyQt5.QtWidgets import QApplication, QWidget,QPushButton,QHBoxLayout
from PyQt5.QtCore import Qt

class QmyWidget(QWidget):
    def __init__(self,parent = None):
        super().__init__(parent)
        self.ID_1 = self.startTimer(500,Qt.PreciseTimer)   #启动第 1 个计时器
        self.ID_2 = self.startTimer(1000,Qt.CoarseTimer)   #启动第 2 个计时器
        btn_1 = QPushButton("停止第 1 个计时器", self)
        btn_2 = QPushButton("停止第 2 个计时器", self)
        btn_1.clicked.connect(self.killTimer_1)
        btn_2.clicked.connect(self.killTimer_2)

        h=QHBoxLayout(self)
        h.addWidget(btn_1)
        h.addWidget(btn_2)
    def timerEvent(self, event):   #计时器事件
        print("我是第"+str(event.timerId())+"个计时器。")
    def killTimer_1(self):
        if self.ID_1:
            self.killTimer(self.ID_1)   #停止第 1 个计时器
    def killTimer_2(self):
        if self.ID_2:
            self.killTimer(self.ID_2)   #停止第 2 个计时器
if __name__ == "__main__":
```

```
app = QApplication(sys.argv)
myWindow = QmyWidget()
myWindow.show()
sys.exit(app.exec())
```

10.4 事件过滤和自定义事件

前面已经讲过，一个控件或窗口的 event()函数是所有事件的集合点，可以在 event()中设置某种类型的事件是接收还是忽略，另外还可以用事件过滤器把某种事件注册给其他控件或窗口进行监控、过滤和拦截。

10.4.1 事件的过滤

一个控件产生的事件可以交给其他控件进行处理，而不是由自身的处理函数处理，原控件称为被监测控件，进行处理事件的控件称为监测控件。要实现这个目的，需要将被监测控件注册给监测控件。

1. 事件过滤器的注册与删除

要把被监测对象的事件注册给监测控件，需要在被监测控件上安装监测器，被监测控件的监测器用 installEventFilter(QObject)方法定义，其中 QObject 是监测控件。如果一个控件上安装了多个事件过滤器，则后安装的过滤器先被使用，用 removeEventFilter(QObject)方法可以解除监测。

2. 事件的过滤

要实现对被监测对象事件的过滤，需要在监测对象上重写过滤函数 eventFilter(QObject,QEvent)，其中参数 QObject 是传递过来的被监测对象，QEvent 是被检测对象的事件类对象。过滤函数如果返回 True，表示事件已经过滤掉了，如果返回 False，表示事件没有被过滤。

3. 事件过滤器的应用实例

下面的程序在两个 QFrame 控件上分别定义了两个 QPushButton 按钮，把这两个按钮的事件注册到窗口上，监控按钮的移动事件，如果移动其中的一个按钮，另一个按钮也同步移动。

```
import sys    #Demo10_8.py
from PyQt5.QtWidgets import QApplication,QWidget,QPushButton,QFrame,QHBoxLayout
from PyQt5.QtGui import QDrag
from PyQt5.QtCore import QPoint,QMimeData,Qt,QEvent


class myPushButton(QPushButton):
    def __init__(self,name=None,parent=None):
        super().__init__(parent)
        self.setText(name)
    def mousePressEvent(self, event):    #按键事件
        if event.button() == Qt.LeftButton:
            drag = QDrag(self)
```

```
                drag.setHotSpot(event.pos()-self.rect().topLeft())
                mime = QMimeData()
                drag.setMimeData(mime)
                drag.exec()
class myFrame(QFrame):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setAcceptDrops(True)
        self.setFrameShape(QFrame.Box)
    def dragEnterEvent(self,event):
        self.child = self.childAt(event.pos())   #获取指定位置的控件
        if self.child:
            event.accept()
        else:
            event.ignore()
    def dragMoveEvent(self,event):
        if self.child:
            self.__center = QPoint(int(self.child.width() / 2), int(self.child.height() / 2))
            self.child.move(event.pos() - self.__center)
class myWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setupUi()
        self.resize(600,400)
        self.setAcceptDrops(True)
    def setupUi(self):
        self.frame_1 = myFrame(self)
        self.frame_2 = myFrame(self)
        H = QHBoxLayout(self)
        H.addWidget(self.frame_1)
        H.addWidget(self.frame_2)
        self.btn1 = myPushButton("button 1",self.frame_1)   #定义第 1 个按钮
        self.btn2 = myPushButton("button 2",self.frame_2)   #定义第 2 个按钮

        self.btn1.installEventFilter(self)   #将 btn1 的事件注册到窗口 self 上
        self.btn2.installEventFilter(self)   #将 btn2 的事件注册到窗口 self 上
    def eventFilter(self,watched,event):   #事件过滤函数
        if watched == self.btn1 and event.type()==QEvent.Move:
            self.btn2.move(event.pos())
```

```
                return True
            if watched == self.btn2 and event.type()==QEvent.Move:
                self.btn1.move(event.pos())
                return True
            return super().eventFilter(watched,event)
if __name__ == '__main__':
    app=QApplication(sys.argv)
    window = myWindow()
    window.show()
    sys.exit(app.exec())
```

10.4.2 自定义事件

除了可以直接使用 PyQt5 中标准的事件外，用户还可以自定义事件，指定事件产生的时机和事件的接受者。

1. 自定义事件类

定义自己的事件首先要创建一个继承自 QEvent 的类，并给自定义事件一个 ID 号（值），该 ID 号的值只能在 QEvent.User（值为 1000）和 QEvent.MaxUser（值为 65535）之间，且不能和已经有的 ID 号相同。为保证 ID 号的值不冲突，可以用 QEvent 类的 registerEventType([hint=-1])函数注册自定义事件的 ID 号，并检查给定的 ID 号是否合适，如果 ID 号合适，会返回指定的 ID 号值，如果不合适；则推荐一个 ID 号值。在自定义事件类中根据情况定义所有的属性和方法。

2. 自定义信号的发送

需要用 QCoreApplication 的 sendEvent(receiver,event)函数或 postEvent(receiver,event)函数发送自定义事件，其中 receiver 是自定义事件的接收者，event 是自定义事件的实例化对象。用 sendEvent(receiver,event)函数发送的自定义事件被 QCoreApplication 的 notify()函数直接发送给 receiver 对象，返回值是事件处理函数的返回值；用 postEvent(receiver,event)函数发送的自定义事件添加到事件队列中，它可以在多线程应用程序中用于在线程之间交换事件。

3. 自定义事件的处理函数

控件或窗口上都有个 customEvent(event)函数，用于处理自定义事件，自定义事件类的实例作为实参传递给形参 event，也可以用 event(event)函数处理，在 customEvent(event)函数或 event(event)函数中根据事件类型进行相应的处理，也可用事件过滤器来处理。

4. 自定义事件的应用实例

下面的程序是建立自定义事件的例子，读者可以根据这个例子了解建立自定义事件的过程。

```
import sys    #Demo10_9.py
from PyQt5.QtWidgets import QApplication,QWidget,QPushButton,QFrame,QHBoxLayout
from PyQt5.QtGui import QDrag
from PyQt5.QtCore import QPoint,QMimeData,Qt,QEvent,QCoreApplication


class myEvent(QEvent):      #自定义事件
```

```
    myID = QEvent.registerEventType(20000)   #注册 ID 号
    def __init__(self,position,object_name=None):
        QEvent.__init__(self,myEvent.myID)
        self.__pos = position     #位置属性，可对数据做其他处理
        self.__name = object_name   #名称属性
    def get_pos(self):   #事件的方法
        return self.__pos
    def get_name(self):     #事件的方法
        return self.__name


class myPushButton(QPushButton):
    def __init__(self,name=None,parent=None):
        super().__init__(parent)
        self.setText(name)
    def mousePressEvent(self, event):   #按键事件
        if event.button() == Qt.LeftButton:
            drag = QDrag(self)
            drag.setHotSpot(event.pos()-self.rect().topLeft())
            mime = QMimeData()
            drag.setMimeData(mime)
            drag.exec()
    def moveEvent(self, event):
        self.__customEvent = myEvent(event.pos(), self.objectName())   #自定义事件的实例化
        QCoreApplication.sendEvent(self.window(), self.__customEvent)   #发送事件
class myFrame(QFrame):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setAcceptDrops(True)
        self.setFrameShape(QFrame.Box)
    def dragEnterEvent(self,event):
        self.child = self.childAt(event.pos())   #获取指定位置的控件
        if self.child:
            event.accept()
        else:
            event.ignore()
    def dragMoveEvent(self,event):
        if self.child:
            self.__center = QPoint(int(self.child.width() / 2), int(self.child.height() / 2))
            self.child.move(event.pos() - self.__center)
```

```
class myWindow(QWidget):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.setupUi()
        self.resize(600,400)
        self.setAcceptDrops(True)
    def setupUi(self):
        self.frame_1 = myFrame(self)
        self.frame_2 = myFrame(self)
        H = QHBoxLayout(self)
        H.addWidget(self.frame_1)
        H.addWidget(self.frame_2)
        self.btn1 = myPushButton("PushButton 1",self.frame_1)   #定义第 1 个按钮
        self.btn1.setObjectName("button1")   #按钮的名称
        self.btn2 = myPushButton("PushButton 2",self.frame_1)   #定义第 2 个按钮
        self.btn2.setObjectName("button2")   #按钮的名称
        self.btn3 = myPushButton("PushButton 3", self.frame_2)   # 定义第 3 个按钮
        self.btn3.setObjectName("button3")   #按钮的名称
        self.btn4 = myPushButton("PushButton 4", self.frame_2)   # 定义第 4 个按钮
        self.btn4.setObjectName("button4")   #按钮的名称
    def customEvent(self,event) :   #自定义事件的处理函数
        if event.type() == myEvent.myID:
            if event.get_name() == "button1":
                self.btn3.move(event.get_pos())
            if event.get_name() == "button2":
                self.btn4.move(event.get_pos())
            if event.get_name() == "button3":
                self.btn1.move(event.get_pos())
            if event.get_name() == "button4":
                self.btn2.move(event.get_pos())
if __name__ == '__main__':
    app=QApplication(sys.argv)
    window = myWindow()
    window.show()
    sys.exit(app.exec())
```

推荐新书《Python 编程基础与科学计算》。

《Python 编程基础与科学计算》正文有 543 页，主要讲解 Python 语言的基本语法格式和 Python 科学计算包 NumPy（数组运算）、matplotlib（绘制各种二维和三维图表）、SciPy（科学计算方法）、SymPy（符号运算）、openpyxl（读写 Excel 数据）及 PyQt5 读写文本文件和二进制文件方面的内容。

本书重点讲解 Python 在科学计算方面的应用，包括数组的操作；绘制二维和三维数据图像；数值计算方法，如聚类算法、线性代数运算、稀疏矩阵、积分、微分、常微分方程组的求解、插值算法、优化算法、傅里叶变换、信号处理、图像处理、正交距离回归、空间算法等；符号运算，包括符号表达式简化、微分、积分、极限、泰勒展开、积分变换、代数方程、求解常微分和偏微分方程、求解非线性方程组、密集和稀疏矩阵运算等；用 Python 处理 Excel 数据；文本数据、二进制数据和原生数据的读写等内容。本书给出了每个命令的语法格式，对语法中的参数进行详细解释，在每个知识点配以实例程序。

主要内容有：(1)N 维数组的各种操作（一维数组是向量、二维数组是矩阵）和各种数组计算函数、各种类型的随机数组（如正态分布、二项式分布、F 分布等等）、数据统计；(2)多项式微积分与最小二乘法拟合(3)线性代数运算（范数、秩、行列式、逆矩阵、特征值和特征向量、线性方程组 Ax=b 的解、矩阵方程的解等等）；(3)矩阵分解（奇异值 SVD 分解、QR 分解、Cholesky 分解、LU 分解、LDL 分解、schur 分解、QZ 分解、Hessenberg 分解、极分解）；(4)稀疏矩阵的运算（逆矩阵、矩阵指数、特征值、SVD、方程组）；(5)数值积分和微分（一重、二重、三重和 n 重定积分）；(6)常微分方程组的解；(7)插值计算（样条插值、多项式插值、FFT 插值）；(8)聚类算法（k 平均、矢量量化、层次聚类）；(9)优化计算（单变量、多变量局部优化、全局最优差分优化、模拟退火优化、线性规划、曲线拟合、非线性方程的根等）；(10)傅里叶变换、逆变换、正弦余弦变换、各种窗函数、小波分析；(11)数字信号处理（卷积和相关计算、FIR 和 IIR 滤波器及其设计、非线性滤波器）；(12)图像处理（图像卷积、高斯滤波、边缘检测、图像变换、放射变换、形态学等）；(13)正交距离回归；(14)空间算法（旋转变换、kd 树近邻搜索、劳内德三角剖分、凸包等）；(15)绘制各种二维和三维数据图表；(16)符号运算和公式推导（符号表达式的运算和简化、极限、微积分、泰勒展开、积分变换（拉普拉斯变换、梅林变换、傅里叶变换、汉克尔变换、正余弦变换）、方程求解（代数方程、线性方程、非线性方程、常微分方程、偏微分方程）、矩阵的各种运算、稀疏矩阵、绘图）；(17)读写 Excel 表格和在 Excel 表格中绘图；(18)读写文本文件、二进制文件和原生数据、临时文件和内存临时存储；（19）各种单位之间的换算关系及 MATLAB、Fortran 文件的读写；(20)推荐用《Python 基础与 PyQt 可视化编程详解》编写复杂的 GUI 图形界面，该书正文有 610 页，讲解详细，实例丰富。