**Qt学习笔记**

1、信号和槽机制是Qt编程的基础。它可以让应用程序编程人员把这些互不了解的对象绑定在一起。

2、槽和普通的C++成员函数几乎是一模一样的——可以是虚函数；可以被重载；可以是公有的、保护的或者私有的，并且可以被其他的C++成员函数直接调用；还有，它们的成员函数可以是任意类型。唯一不同的是：槽还可以和信号连接在一起，在这种情况下，每当发射信号的时候，就会自动调用这个槽。

3、connect()语句看起会是这个样子：

connect(sender, SIGNAL(signal), receiver, SLOT(slot));

4、 \* 一个信号可以连接多个槽，在发射这些信号的时候，会以**不确定的顺 序**一个接一个的调用这些槽。

\* 多个信号可以连接同一个槽，无论发射的是哪一个信号，都会调用这 个槽

\* 一个信号可以与另外一个信号相连接，例如：

connect(lineEdit, **SIGNAL(textChanged(const Qstring &))**, this, **SIGNAL(updateRecord(const Qstring &))**);

\* 连接可以被移除：disconnect(lcd, SIGNAL(overflow()), this, SLOT(handelMathError())); 但是这种情况**较少用到**，因为当删除对象的时候，Qt会自动解除和 这个对象相关的所有连接。

5、要把信号成功的连接到槽（或者连接到另外一个信号），它们的参数必须具有相同的顺序和相同的类型。但是有个例外：如果信号的参数比它所连接的槽的参数要多，那么多余的参数将会被简单的忽略掉；如果参数类型不匹配或者如果信号或者槽不存在，则当应用程序使用调试模式构建后，Qt会在运行的时候发出警告。

6、Qt的主要成就之一就是使用了一种机制对C++进行了扩展，并使用这种机制创建了独立的软件组成。这种机制称为**元对象系统，**它提供了关键的两项技术：**信号——槽以及内省**。内省功能对于实现信号和槽是必需的。

7、无论是使用手工编码还是使用Qt设计师，在创建对话框的时候总是要包含以下几个相同的基本步骤：

\* 创建并初始化字窗口部件

\* 把子窗口部件放到布局中

\* 设置Tab键顺序

\* 建立信号——槽之间的连接

\* 实现对话框中的自定义槽

8、在使用Qt设计师的设置快捷键的过程中，要注意一下一点：

\* 当使用&操作符和字符串连接起来的时候，&操作符后面连接的字符就是快捷键，但是要显示快捷键的时候，必须进入伙伴设计模式，这样才能将&的显示隐藏掉，不然的话，text文本将依然显示为&加上字符串。

9、可以通过**Qt命令行**来运行qmake，生成一个.pro文件和一个makefile文件。

10、改变形状的对话框：在某系情况下，人们希望能够提供一些改变形状的对话框。最常见的可改变形状的对话框有两种：

\* 扩展对话框

\* 多页对话框

在Qt中不论是用纯粹的代码，还是使用Qt设计师，都可以实现这两种对话框。

11、动态对话框：就是对话框在程序运行的时使用的从Qt设计师的.ui文件创建而来的那些对话框。动态对话框不需要通过uic把.ui文件转换成C++代码，相反，它是在程序运行的时候使用QUiLoader类载入该文件的。QuiLoader类放在一个独立的库中。为了在Qt应用程序中使用QuiLoader，必须在这个应用程序中的.pro文件中加入这一行：CONFIG += uitools;

12、动态对话框使不重新编译引用程序而可以改变窗体布局的做法成为可能。

13、Qt提供的4种类型的按钮：

\* QpushButton

\* QtoolButton

\* QcheckBox:复选框

\* QradioButton:单选按钮

14、Qt的容器窗口部件是一种可以包含其他窗口部件的窗口部件。例如QgroupBox和Qframe。

15、QTabWidget和QtoolBox是多页窗口部件。在多页窗口部件中，每一页都是一个子窗口部件，并从0开始编号这些页。滚动条机制是在QabstractScrollArea中实现的，它是所有项视图和其他类型的可滚动窗口部件的基类。

16、应用程序的主窗口提供了用于构建应用程序用户界面的框架。在绝大多数图形用户界面应用程序的后台，都有一套提供底层功能的代码——例如，用于读写文件或者用于处理用户界面中数据的代码。

17、子类化QmainWindow：通过子类化QmainWindow，可以创建一个应用程序的主窗口。Qdialog和QmainWindow都是**派生于Qwidget**。

18、**Qt信号与槽**

1、signals前面**不可加public、private和protected进行修饰**；**s**l**ots前面可以加**，因为Qt说槽函数可以当普通函数使用。

2、signals区域的函数必须是**void**类型，**而且这些信号函数没有函数体，也就是说不可以自己定义这些信号函数，你只要声明它就够了**，其它不用管，Qt内部自己弄。

3、**宏定义和函数指针不能用于信号和槽的参数，信号和槽也不能有缺省参数。**

**4、信号**

当某个信号对其客户或所有者发生的内部状态发生改动，信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时，和其相关联的槽将被即时执行，就象一个正常的函数调用相同。信号-槽机制完全独立于所有GUI事件循环。**只有当所有的槽返回以后发射函数（emit）才返回**。如果存在多个槽和某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，不过他们执行的顺序将会是**随机的、不确定的**，我们不能人为地指定哪个先执行、哪个后执行。   
信号的声明是在头文件中进行的，QT的signals关键字指出进入了信号声明区，随后即可声明自己的信号。例如，下面定义了三个信号：   
signals:   
void mySignal();   
void mySignal(int x);   
void mySignalParam(int x,int y);   
 在上面的定义中，signals是QT的关键字，而非C/C++的。接下来的一行void mySignal() 定义了信号mySignal，这个信号没有携带参数；接下来的一行void mySignal(int x)定义了重名信号mySignal，不过他携带一个整形参数，这有点类似于C++中的虚函数。**从形式上讲信号的声明和普通的C++函数是相同的，不过信号却没有函数体定义，另外，信号的返回类型都是void，不要指望能从信号返回什么有用信息。** 信号由moc自动产生，他们不应该在.cpp文件中实现。

5、槽

**槽是普通的C++成员函数，能被正常调用，他们唯一的特别性就是非常多信号能和其相关联**。当和其关联的信号被发射时，这个槽就会被调用。**槽能有参数，但槽的参数不能有缺省值**。   
 既然槽是普通的成员函数，因此和其他的函数相同，他们也有存取权限。槽的存取权限决定了谁能够和其相关联。同普通的C++成员函数相同，槽函数也分为三种类型，即public slots、private slots和protected slots。   
 public slots：在这个区内声明的槽意味着所有对象都可将信号和之相连接。这对于组件编程非常有用，你能创建彼此互不了解的对象，将他们的信号和槽进行连接以便信息能够正确的传递。   
 protected slots：在这个区内声明的槽意味着当前类及其子类能将信号和之相连接。这适用于那些槽，他们是类实现的一部分，不过其界面接口却面向外部。   
 private slots：在这个区内声明的槽意味着只有类自己能将信号和之相连接。这适用于联系非常紧密的类。   
 槽也能够声明为虚函数，这也是非常有用的。   
 槽的声明也是在头文件中进行的。例如，下面声明了三个槽：

public slots:   
 void mySlot();   
 void mySlot(int x);   
 void mySignalParam(int x,int y);

6、信号和槽的关联

通过**调用QObject对象的connect函数来将某个对象的信号和另外一个对象的槽函数相关联**，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：   
 bool QObject::connect ( const QObject \* sender, const char \* signal, const QObject \* receiver, const char \* member ) [static]   
 这个函数的作用就是将发射者sender对象中的信号signal和接收者receiver中的member槽函数联系起来。当指定信号signal时必须使用QT的宏SIGNAL()，当指定槽函数时必须使用宏SLOT()。**如果发射者和接收者属于同一个对象的话，那么在connect调用中接收者参数能省略。**   
 例如，下面定义了两个对象：标签对象label和滚动条对象scroll，并将valueChanged()信号和标签对象的setNum()相关联，另外信号还携带了一个整形参数，这样标签总是显示滚动条所处位置的值。   
QLabel \*label = new QLabel;   
QScrollBar \*scroll = new QScrollBar;   
QObject::connect( scroll, SIGNAL(valueChanged(int)), label, SLOT(setNum(int)) );   
**一个信号甚至能够和另一个信号相关联**，看下面的例子：   
class MyWidget : public QWidget   
{   
　　public:   
　　MyWidget();   
　　…   
　　signals:   
　　void aSignal();   
　　…   
　　private:   
　　…   
　　QPushButton \*aButton;   
};   
MyWidget::MyWidget()   
{   
　　aButton = new QPushButton( this );   
　　connect( aButton, SIGNAL(clicked()), SIGNAL(aSignal()) );   
}   
 在上面的构造函数中，MyWidget创建了一个私有的按钮aButton，按钮的单击事件产生的信号clicked()和另外一个信号aSignal() 进行了关联。这样一来，当信号clicked()被发射时，信号aSignal()也接着被发射。当然，你也能直接将单击事件和某个私有的槽函数相关联，然后在槽中发射aSignal()信号，这样的话似乎有点多余。   
 当信号和槽没有必要继续保持关联时，我们能**使用disconnect函数来断开连接**。其定义如下：   
 bool QObject::disconnect ( const QObject \* sender, const char \* signal, const Object \* receiver, const char \* member ) [static]   
这个函数断开发射者中的信号和接收者中的槽函数之间的关联。   
**有三种情况必须使用disconnect()函数：**   
 （1）断开和某个对象相关联的所有对象。这似乎有点不可理解，事实上，当我们在某个对象中定义了一个或多个信号，这些信号和另外若干个对象中的槽相关联，如果我们要切断这些关联的话，就能利用这个方法，非常之简洁。   
disconnect( myObject, 0, 0, 0 ) 或 myObject->disconnect()   
 （2）断开和某个特定信号的所有关联。   
 disconnect( myObject, SIGNAL(mySignal()), 0, 0 ) myObject->disconnect( SIGNAL(mySignal()) )  
 （3）断开两个对象之间的关联。   
 disconnect( myObject, 0, myReceiver, 0 ) 或 myObject->disconnect( myReceiver )

在disconnect函数中0能用作一个通配符，分别表示所有信号、所有接收对象、接收对象中的所有槽函数。不过发射者sender不能为0，其他三个参数的值能等于0。

**6、应注意的问题**

信号和槽机制是比较灵活的，但有些局限性我们必须了解，这样在实际的使用过程中做到有的放矢，避免产生一些错误。下面就介绍一下这方面的情况。   
 **（1）信号和槽的效率是非常高的**，不过同真正的回调函数比较起来，由于增加了灵活性，因此在速度上还是有所损失，当然这种损失相对来说是比较小的。**但如果我们要追求高效率的话，比如在实时系统中就要尽可能的少用这种机制。**  
 **（2）信号和槽机制和普通函数的调用相同**，如果使用不当的话，在程式执行时也有可能产生死循环。因此，在定义槽函数时一定要注意避免间接形成无限循环，即在槽中再次发射所接收到的同样信号。   
 **（3）如果一个信号和多个槽相联系**的话，那么，当这个信号被发射时，和之相关的槽**被激活的顺序将是随机的**。   
 **（4）宏定义不能用在signal和slot的参数中。**   
 既然moc工具不扩展#define，因此，在signals和slots中携带参数的宏就不能正确地工作，如果不带参数是能的。

**（5）构造函数不能用在signals或slots声明区域内。**  
 的确，将一个构造函数放在signals或slots区内有点不可理解，无论怎么，不能将他们放在private slots、protected slots或public slots区内。

**（6） 函数指针不能作为信号或槽的参数。**

**（7）信号和槽不能有缺省参数。**  
 既然signal->slot绑定是发生在运行时刻，那么，从概念上讲使用缺省参数是困难的。下面的用法是不合理的：   
class SomeClass : public QObject   
{   
Q\_OBJECT   
public slots:   
void someSlot(int x=100); // 将x的缺省值定义成100，在槽函数声明中使用是错误的

};

**（8）信号和槽也不能携带模板类参数。**

如果将信号、槽声明为模板类参数的话，即使moc工具不报告错误，也不可能得到预期的结果。

**（9）嵌套的类不能位于信号或槽区域内，也不能有信号或槽。**

**（10）友元声明不能位于信号或槽声明区内**。

19、在Qt中，创建菜单和工具栏包括以下步骤：

\* 创建并设置动作

\* 创建菜单并把动作添加到菜单上

\* 创建工具栏并把动作添加到工具栏上

20、模态窗口就是一个在得到调用的可以弹出并可以阻塞应用程序的窗口，从而会从调用发生开始起妨碍其他任意处理或者交互操作，直到关闭该窗口为止。前面使用的文件对话框和消息框就是模态的。

21、创建自定义窗口部件（包括两种方法）：

\* 通过对一个已经存在的Qt窗口部件进行子类化

\* 直接对Qwidget进行子类化

22、自定义Qt窗口部件：在某些情况下，我们发现Qt窗口部件需要更多的自定义定制，这些定制可能要比它在Qt设计师里面可设置的属性或者对他调用的那些函数更多一些。一个简单直接的解决方法就是对相关的窗口部件进行子类化 并且使它能够满足我们的需要。

23、许多自定义窗口部件是都是对现有窗口部件的简单组合，不论它们是内置的Qt窗口部件，还是其他一些像HexSpinBox这样的自定义窗口部件。

24、通过对现有窗口部件的组合构建而成的自定义窗口部件通常都可以在Qt设计师中开发出来：

\* 使用“Widget”模板创建一个新窗体。

\* 把一些必需的窗口部件添加到这个窗体上，并且对它们进行摆放。

\* 设置一些信号和槽的连接。

\* 如果通过信号和槽不能获得所需的行为，则只需要在类中添加一些必要的代码即可——这个类需要同时从Qwidget类和uic生成的类中派生出来。

25、要对这些现有的窗口部件进行组合，也完全可以通过手写代码方式来加以实现。但无论使用的是哪种方式，最终生成的类都会是Qwidget的一个子类。

26、如果窗口本身没有任何信号和槽，并且它也没有重新实现任何虚函数，那么我们甚至还是有可能通过对现有窗口部件的组合而不是通过子类化的方式来生成这样的窗口部件。

27、当我们在Qt设计师中使用窗口部件的时候，在Qt设计师属性编辑器里，那些继承于Qwidget的属性下面，将会显示这些自定义属性。这些属性可以是Qvariant所支持的任何类型。对于定义属性的类，Q\_OBJECT宏是必需的。

28、Qt提供了两种用存储颜色的类型：QRgb和Qcolor。虽然QRgb仅仅是一个用在Qimage中存储32位像素的类型别名，但Qcolor则是一个具有许多有用函数并且在Qt中广泛用于存储颜色的类。在QiconEditor窗口部件中，只有在处理QImage时，我们才是用QRgb，而对于其他任意东西，我们都指只使用Qcolor。

29、sizeHint()函数是从Qwidget中重新实现的，并且可以返回一个窗口部件的理想大小。

30、除了大小提示，窗口部件还有一个大小策略。他会告诉布局系统是否可以对这个窗口部件进行来场或者缩短。通过在构造函数中调用以QsizePolicy::Minimum为水平和垂直大小策略的setSizePolicy()，会告诉负责管理这个窗口部件的任意管理器，这个窗口部件的大小提示就是它的尺寸大小。

31、事件处理器，每一个都对应一种不同类型的事件。

32、没一个窗口部件都会配备一个调色板，由它来确定做什么事应该使用什么颜色。默认情况下，一个窗口部件的调色板会采用窗口部件系统的颜色主题。

33、一个窗口部件的调色板有三个颜色组构成：激活组（Active）、非激活组（Inactive）和不可用组（Disabled）。应该使用哪一个颜色组取决于该窗口部件的当前状态：

\* Active颜色组课用于当前激活窗口中的那些窗口部件。

\* Inactive颜色组可用于其他窗口中的那些窗口部件。

\* Disabled颜色组可用于任意窗口中的那些不可用窗口部件。

34、在Qt设计师中使用自定义窗口部件的之前，我们必须让Qt设计师先察觉到他们的存在。有两种方法可以完成这一任务：改进法和插件法。

35、改进法的缺点是：

\* 在Qt设计师中的，无法对自定义窗口部件的那些特定的属性进行访问

\* 无法对这个窗口部件自身进行绘制

36、插件法：插件法需要创建一个插件库，Qt设计师可以在运行时加载这个库，并且可以利用该库创建窗口部件的实例。

37、双缓冲：是一种图形用户界面编程技术，它包括把一个窗口部件渲染到一个脱屏像素映射中以及把这个像素映射复制到显示器上面。在Qt的早期版本中，这种技术通常用于消除屏幕的闪烁以及为用户提供一个漂亮的用户界面。在Qt4中，Qwidget会自动处理这些情况，所以很少考虑窗口部件的闪烁问题。

38、默认情况下，子窗口部件都会从它们的父窗口部件哪里继承相应的背景色。

39、  **QVector，是Qt对所有数组的封装**，比如我们想要一个int类型数组，我们原先会写int array[10]，我们在Qt里可以写QVector <int>  array(10)，赋值的时候，我们依然可以照旧array[5]=4;想获取某一项的值也还可以array[9]，也就是说，原来的特性我们还可以用。

40、一般情况下，在顶层窗口部件中调用show()足以显示所有的子窗口部件。但是，当明确地对子窗口部件调用hide()的时候，除非再次对它调用show()，否则它就一直会处于隐藏状态。

41、minimumSize()函数与sizeHint()函数相似，sizeHint()可以指定一个窗口部件的理想大小，minimumSize()可以只指定一个窗口部件理想的最小大小。布局绝不会把一个窗口部件的大小修改为比它的最小大小提示还要小的大小。

42、**QRect类定了平面上的矩形**：

一个矩形在内部是由**左上角和右下角**表示的，但是通常它所表达到是一个左上角和一个大小。

坐标类型是QCOORD（和int一样在qwindowdefs.h中定义的）。QCOORD的最小值是QCOORD\_MIN（-2147483648），最大值是QCOORD\_MAX（2147483647）。

注意矩形的大小（宽和高）也许与你通常所用的所有不同。**如果左上角和右下角相同，那么这个矩形的宽和高都为1。**

通常情况下，width = right - left + 1并且height = bottom - top + 1。我们这样设计是因为这样就可以使矩形空间可以用在绘制函数中，它的宽高就说明了所要绘制的象素。**例如，我们画一个宽和高都为1的矩形为一个单一点象素。**

默认坐标系统的原点(0,0)在左上角。y轴的正方向向下，并且x轴的正方向从左到右。

一个QRect可以用一组上、左、宽、高四个整数，或者从两个QPoint或者从一个QPoint和一个QSize来构造。创建之后，空间可以被改变，比如使用setLeft()、setRight()、setTop()和setBottom()，或者通过设置，比如setWidth()、setHeight()和setSize()。空间也可以通过移动函数来改变，比如moveBy()、moveCenter()和moveBottomRight()等等。你也可以用addCoords()给矩形加上坐标。

你也可以通过contains()测试来看一个QRect是否包含一个特定点。你也可以通过intersects()测试来看两个QRect是否相交（也可以参考intersect()）。可以使用unite()来获得两个QRect的边界长方形。

43、**Qt鼠标事件**:处理鼠标事件的频率不低于处理键盘事件的频率。包括按下、松开鼠标事件；移动鼠标到特定的区域或离开特定的区域；更改鼠标指针的形状等等。

**1.按下、松开鼠标按键**

    按下并释放鼠标按钮时，将调用以下方法：

* mousePressEvent (self,  event) - 鼠标键按下时调用;
* mouseReleaseEvent (self, event) - 鼠标键公开时调用;
* mouseDoubieCiickEvent (self, event) - 双击鼠标时调用。必须注意，在双击之前的其他事件。QApplicaption类的**setDoubleClickInterval( )方法可设置双击的时间间隔**；**doubleClickInterval( )方法返回双击的时间间隔**。

    event参数是QMouseEvent对象，存储事件的其他信息。有以下方法：

* x() 和 y() -返回相对于控件空间的鼠标坐标值;
* pos() - 返回相对于控件空间的QPoint对象;
* localPos()- 返回相对于控件空间的QPointF对象;
* globalX() 和 globalY() -  返回相对于屏幕的x,y 坐标值；
* globalPos() - 返回相对于屏幕的QPoint对象;
* windowPos() -  返回相对于窗口的QPointF对象;
* screenPos() - 返回相对于屏幕的QPointF对象;
* button() - 返回以下枚举值，用以判断是哪个鼠标健触发了事件。
  + QtCore.Qt.NoButton - 0 - 没有按下鼠标键。例如移动鼠标时的button()返回值；
  + QtCore.Qt.LeftButton -1 -按下鼠标左键；
  + QtCore.Qt.RightButton -2 -按下鼠标右键；
  + QtCore.Qt.Mion 或 QtCore.Qt.MiddleButton -4 -按下鼠标中键；
* buttons() - 返回前面所列枚举值的组合，用于判断同时按下了哪些键。
* **modifiers() - 判断按下了哪些修饰键（Shift,Ctrl , Alt,等等）,**详见键盘事件(18)中的modifiers()。
* **timestamp() - 返回事件发生的时间**；

    如果要让**父控件继续收到鼠标事件**，要调用事件的**ignore()**方法；否则，调用accept()。

    如果一个控件的QtCore.Qt.WA\_NoMousePropagation的属性设为True,则不会将事件传递给父控件。调用setAttribute( )方法可修改此参数：  
    button.setAttribute (QtCore.Qt.WA\_NoMousePropagation, True)

    缺省情况下，鼠标事件只拦截控件区域上的鼠标操作。如果可拦截控件区域以下的鼠标事件，必须调用grabMouse( )方法；释放时，调用releaseMouse( )。

**2.鼠标指针**

    要处理鼠标指针的移动，需要重载mouseMoveEvent(self，event)方法。**缺省情况下，只有按下鼠标键移动时，才会调用mouseMoveEvent( )。如果要处理包括普通的移动，需要以参数为True调用setMouseTracking() 方法。如果要处理窗口中鼠标移动的事件，需要调用grabMouse( )方法。**

     event对象的pos( )**返回值为相对控件的坐标**，要转换成**相对父控件或屏幕的坐标**，需要调用QWidget类的以下方法：

* mapToGlobal (QPoint) - 将窗口坐标转换成屏幕坐标;
* mapFromGlobal(QPoint) - 将屏幕坐标转换成窗口坐标;
* mapToParent(QPoint) - 将窗口坐标转换成父窗口坐标。如果没有父窗口，则相当于mapToGlobal (QPoint);
* mapFromParent(QPoint) - 将父窗口坐标转换成窗口坐标。如果没有父窗口，则相当于mapFromGlobal(QPoint);
* mapTo (QWidget, QPoint) - 将窗口坐标转换成 QWidget父窗口坐标;
* mapFrom (QWidget, QPoint) - 将 QWidget父窗口坐标转换成窗口坐标;

**3. 鼠标移进和移出控件**

鼠标移进和移出控件时，下列方法将被调用：

* enterEvent (self, event) -鼠标进入控件;
* leaveEvent (self, event) - 鼠标离开控件;

     event是一个QEvent对象，并不包括附加信息。

    4.滚动鼠标

    wheelEvent (self, event)方法可用来处理鼠标滚动事件。event是一个QWheelEvent对象，包含滚轮操作的相关信息。有以下方法可调用：

* angleDelta( ) - 返回QPoint对象，为滚轮转过的数值，单位为1/8度。例如：  
  angle=event.angleDelta( ) /8  
  angleX=angle.x()  
  angleY=angle.y()
* pixelDeita () - 返回QPoint对象，为滚轮转过的像素值。
* x() 和 y() - 返回相对于控件的当前鼠标的x,y位置;
* pos() - 返回相对于控件的当前鼠标位置的QPoint对象;
* posF() - 返回相对于控件的当前鼠标位置的QPoinFt对象;
* globalX() 和globalY() - 返回相对于屏幕的当前鼠标的x,y位置;
* globalPos() - 返回相对于屏幕的当前鼠标QPoint位置;
* globalPosF() - 返回相对于屏幕的当前鼠标QPointF位置;
* buttons(),modifiers()和timestamp()的用法参见本文“1.按下、松开鼠标按键”中的相关内容。

    如果要让父控件继续收到滚轮事件，要调用事件的ignore()方法；否则，调用accept()。

**5.更改鼠标指针形状**

    要修改鼠标进入控件后的形状，可调用QWidget的下列方法：  
    setCursor(QCursor qcr) - 参数qcr为QCursor对象或 Qtcore.Qt 类的枚举值,如：ArrowCursor(标准箭头)、upArrowCursor(向上箭头)、 CrossCursor(十字光标)、Waitcursor (沙漏)，等等。

    setCursor(QtCore.Qt.WaitCursor)

* unsetCursor() -  取消设置的鼠标形状。
* cursor() - 返回当前鼠标形状的QCursor对象，。

    使用QApplication类中的以下静态方法来控制整个应用程序的鼠标形状：

* setOverrideCursor(QCursor qcr) - 参数qcr为QCursor对象或 Qtcore.Qt 类的枚举值。
* restoreOverrideCursor() - 取消全局鼠标形状设置；
* changeOverrideCursor(QCursor qcr) - 将鼠标形状设置为qcr。**只有先调用setOverrideCursor( )了，该函数才起作用**。
* overrideCursor( ) - 返回当前鼠标形状的QCursor 对象；

**setOverrideCursor()和restoreOverrideCursor( )通常配合使用。**

44、Qt提供了两种用于控制鼠标光标形状的机制：

\* 当鼠标悬停在某个特殊的窗口部件的时，Qwidget::setCursor()可以设置它所使用的光标形状。如果没有为窗口部件专门设置光标吗，那么就会使用它的父窗口部件中的光标。顶层窗口部件的默认光标是箭头光标，即ArrowCursor。

\* 对于整个应用程序中使用所使用的光标形状，可以通过Qapplication::setOverrideCursor()进行设置，它会把不同窗口部件中的光标形状全部覆盖，直到调用restoreOverrideCursor()。

45、