目录

[一、Qt元对象系统（Meta-Object System） 2](#_Toc11954)

[1.元对象系统 2](#_Toc30989)

[2.信号与槽 4](#_Toc20665)

[3.动态属性系统 8](#_Toc24237)

[二、Qt对象树 10](#_Toc2643)

[三、窗口部件 12](#_Toc2447)

[1.默认部件基类 12](#_Toc15684)

[2.其他部件 15](#_Toc9756)

[四、布局管理器 17](#_Toc22302)

[五、容器 19](#_Toc23336)

[QString 类 19](#_Toc15563)

[QStringList类 25](#_Toc30160)

[QList类 QLinkedList类 26](#_Toc31586)

[QVector类 30](#_Toc14716)

[QMap类、 QHash类 31](#_Toc12465)

[QVariant类 35](#_Toc7349)

[六、 标准对话框 36](#_Toc28971)

[七、 正则表达式 43](#_Toc16883)

[八、 文件处理 47](#_Toc24081)

[九、附件 53](#_Toc13195)

# 一、Qt元对象系统（Meta-Object System）

## 1.元对象系统

元对象系统是一个基于*标准C++的扩展*，为Qt提供了信号与槽机制、实时类型信息、动态属性系统。

元对象系统的三个基本条件：类必须继承自QObject、类声明Q\_OBJECT宏（默认私有有）、元对象编译

器moc。

信号和槽机制是 QT 的核心机制，信号和槽是一种高级接口，应用于对象之间的通信，它是QT的核心特性，信号和槽是QT自行定义的一种通信机制，它独立于标准的 C/C++ 语言，要正确的处理信号和槽，必须借助一个称为 moc（Meta-Object Compiler），也就是“元对象编译器”。，它为高层次的事件处理自动生成所需要的必要代码。Qt 程序在交由标准编译器编译之前，先要使用 moc 分析 C++ 源文件。如果moc发现在一个类头文件中包含了宏 Q\_OBJECT，则会生成以moc\_className.cpp(自定义类名)的.cpp文件。这个源文件中包含了 Q\_OBJECT 宏的实现代码。新的文件同样将进入编译系统，与原文件一起参与编译。构建生成的.o文件包含moc生成的cpp文件。

Q\_OBJECT定义在qobjectdefs.h文件中：

|  |
| --- |
|  |

分析Q\_OBJECT后，发现都是在操作元对象，并没有所谓的信号和槽，属性等内容，很显然，QObject对象能够支持这些功能，必然是通过QMetaObject这个元对象来实现的。

QMetaObject定义：

|  |
| --- |
|  |

Qt元对象系统为何一定要继承自QObject?

分析QObject类：

|  |
| --- |
| … …    … … |

所以，QObject之所以为元对象系统的基类，因其提供了元对象系统很多支持工作：

信号与槽、事件处理、属性设置、国际化支持（翻译）、对象树资源管理等等。

使用QObject作为基类而不使用Q\_OBJECT宏和元对象代码是可以的，但是如果Q\_OBJECT宏没有被使用，那么这个类声明的信号和槽，以及其他特征描述都不会被调用。

元对象系统除实现信号与槽外，还有很多其他特性，比如：

QObject::metaObject()返回与该类绑定的meta-object对象。

QMetaObject::className()可以在运行时以字符串的形式返回类的名字，不需要C++编译器原生的运行时类型信息（RTTI）的支持。

QObject::inherits()函数返回继承信息：对象是否是QObject继承树上一个类的实例。

QObject::tr()和QObject::trUtf8()提供国际化支持，将字符串翻译成指定的语言。

QObject::setProperty()和QObject::property()通过名字动态设置和获取对象属性。

QMetaObject::newInstance()构造该类的一个新实例。

qObject\_cast()动态类型转换。

一般建议在QObject的所有子类中使用Q\_OBJECT宏，而不管它们是否使用了信号与槽。

## 2.信号与槽

GUI用户界面中，当用户操作一个窗口部件时，需要其他窗口部件响应，传统方式经常使用callback（回调机制）来实现。所谓回调即事先将函数指针作为参数传递另一个函数，然后在函数处理过程中适当地方调用回调函数回调机制有两个缺陷：类型不安全，不能保证调用函数过程中使用正确的参数；强耦合，处理函数必须知道调用哪个回调函数。Qt的信号与槽机制是类型安全的，松耦合，更灵活，更方便。

信号与槽（Signal & Slot）是 [Qt](http://c.biancheng.net/qt/" \t "_blank) 编程的基础，也是 Qt 的一大创新。因为有了信号与槽的编程机制，在 Qt 中处理界面各个组件的交互操作时变得更加直观和简单。  
 信号（Signal）就是在特定情况下被发射的事件，例如PushButton 最常见的信号就是鼠标单击时发射的 clicked() 信号。发射信号使用Qt的emit关键字。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。

定义信号，如：

|  |
| --- |
|  |

发射信号，如：

|  |
| --- |
|  |

槽（Slot）就是对信号响应的函数。槽就是一个函数，与一般的[C++](http://c.biancheng.net/cplus/" \t "_blank)函数是一样的，可以声明在类的任何部分（public、private 或 protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函数不同的是：槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

声明三个槽方法，如：

|  |
| --- |
|  |

槽方法中处理需要处理的工作，如：

|  |
| --- |
|  |

注意：使用qDebug()输出信息时注意添加头文件： 

GUI 程序设计的主要内容就是对界面上各组件的信号的响应，只需要知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。

信号与槽关联是用 QObject::connect() 函数实现：

**[QMetaObject::Connection](C:\\Users\\janny\\AppData\\Roaming\\Microsoft\\Word\\qmetaobject-connection.html)** QObject::**connect**(

const **[QObject](file:///C:\\Users\\janny\\AppData\\Roaming\\Microsoft\\Word\\qobject.html" \l "QObject)** \**sender*, //信号发送者

const **char** \**signal*, //发送的信号

const **[QObject](file:///C:\\Users\\janny\\AppData\\Roaming\\Microsoft\\Word\\qobject.html" \l "QObject)** \**receiver*, //信号接收者

const **char** \**method*, //表示与信号连接的方式的字符串，可以是槽或信号

**[Qt::ConnectionType](file:///C:\\Users\\janny\\AppData\\Roaming\\Microsoft\\Word\\qt.html" \l "ConnectionType-enum)** *type* = Qt::AutoConnection //连接方式，默认自动连接

)

常用格式：connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));

用来表示信号和槽的参数都是字符串，Qt提供了两个宏用于构造这样的字符串：对于信号使用SIGNAL，对于槽则使用SLOT，用它们将函数的原型包围起来即可。注意connect方法采用SIGNAL()及SLOT()时，这里的**函数原型只能写出类型，不能有任何参数名**，否则连接将会失败。

同时信号与槽连接方式：

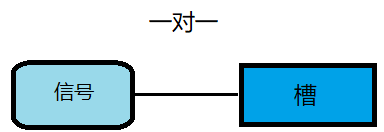
1）Qt::AutoConnection：(默认连接方式)自动方式，由系统自动选择连接方式。  
       2）Qt::DirectConnection：直接方式，信号发射时，立即调用槽。

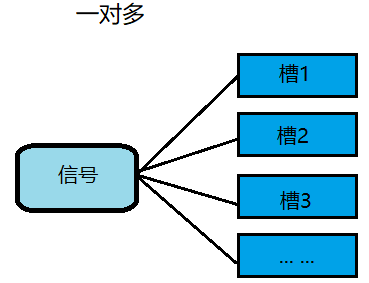
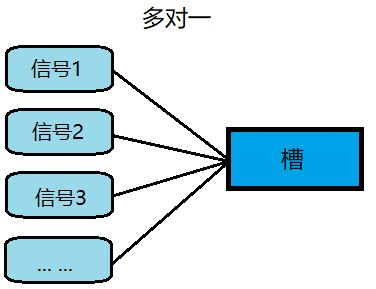
3）Qt::QueuedConnection：队列方式，信号发射时产生一个事件进入队列，事件被处理时槽才能调用。

4）Qt::BlockQueuedConnection：阻塞队列方式，信号发射时产生一个事件进入队列，然后当前线程进入阻塞状态，直到事件处理完毕，若接收方位于发送信号的线程中，则程序会死锁，故此连接方式仅用于多线程。

信号可以看做是特殊的函数，需要带括号，可带参数，信号无需实现也不能实现。槽函数需要带括号，有参数时还需要指明参数。当信号和槽函数带有参数时，在 connect()函数里，要写明参数的类型。信号的参数需与槽的参数列表一致,允许比槽参数多。如果不匹配或参数过少，会出现编译错误或运行错误。在使用信号与槽的类中，必须在类的定义中加入宏 Q\_OBJECT。当一个信号被发射时，与其关联的槽函数通常被立即执行，就像正常调用一个函数一样。只有当信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码。

**信号与槽的关联方式有如下特点：**

1. 一个信号连接一个槽：

connect(sender, SIGNAL(single1()), receiver, SLOT(slotFun()));

如：

|  |
| --- |
|  |

1. 一个信号连接一个信号：

connect(sender, SIGNAL(single1()), receiver, SIGNAL(single2()));

如：

|  |
| --- |
|  |

1. 一个信号连接多个槽，关联信号的槽函数按照建立连接时的顺序依次执行：

connect(sender, SIGNAL(single1()), receiver1, SLOT(slotFun()));  
connect(sender, SIGNAL(single1()), receiver2, SLOT(slotFun()));

connect(sender, SIGNAL(single1()), receiver3, SLOT(slotFun()));

如：

|  |
| --- |
|  |

1. 多个信号连接一个槽：

connect(sender1, SIGNAL(single1()), receiver, SLOT(slotFun()));  
connect(sender2, SIGNAL(single2()), receiver, SLOT(slotFun()));

connect(sender3, SIGNAL(single3()), receiver, SLOT(slotFun()));

如：

|  |
| --- |
|  |

1. 信号与槽的自动关联：

ui\_xxxx.h文件中 connectSlotsByName()方法通过对象名支持信号与槽的自动关联。不采用connect()函数而是采用on\_objectName\_signal命名方式命名槽达到自动关联的效果。

如类头文件中声明槽方法：

|  |
| --- |
|  |

**信号与槽的断开关联：**

QObject：：disconnect(const QObject\* sender,const char\* signal,const QObject \*receiver,const char\* method);

1. 断开与一个对象所有的信号的所有关联

disconnect(sender,0,0,0);

等价于:

sender->disconnect();

1. 断开与一个指定信号的所有关联

disconnect(sender, SIGNAL(single1()), 0, 0);

等价于：

sender->disconnect(SIGNAL(single1()));

1. 断开与一个指定接受者receiver的所有关联

disconnect(sender, 0, receiver, 0);

等价于：

sender->disconnect(SIGNAL(single1()));

1. 断开指定信号与槽的关联：

disconnect(sender, SIGNAL(single1()), receiver, SLOT(slotFun()));

等价于：

disconnect(myConnection); //myConnection为connect()的返回值

如：

xxx.h文件中添加m\_res变量用于保存connect()返回值。

|  |
| --- |
|  |

连接信号与槽：

|  |
| --- |
|  |

断开该连接：

|  |
| --- |
|  |

**信号与槽机制的优越性：**

1. 信号与槽机制是类型安全的，相关联的信号与槽参数必须匹配
2. 信号与槽是松耦合的，信号发送者不知道也不需知道接受者的信息。
3. 信号与槽可以使用任意类型的任意数量的参数。

## 3.动态属性系统

在标准C++中，为了保证封装性，我们经常声明一个私有变量，然后声明两个公有函数，即set函数和get函数。在Qt中我们可以使用宏Q\_PROPERTY()宏来实现这些。一个属性可以使用常规函数QObject::property()和QObject::setProperty()进行读写，不用知道属性所在类的任何细节，除了属性的名字。

Q\_PROPERTY（）原型：

**Q\_PROPERTY(type name**

**(READ getFunction [WRITE setFunction] |**

**MEMBER memberName [(READ getFunction | WRITE setFunction)])**

**[RESET resetFunction]**

**[NOTIFY notifySignal]**

**[REVISION int]**

**[DESIGNABLE bool]**

**[SCRIPTABLE bool]**

**[STORED bool]**

**[USER bool]**

**[CONSTANT]**

**[FINAL])**

示例：

1. 新建桌面应用程序TestProperty，父类QWidget,其他采用默认。
2. 右键单击项目添加自定义类MyPropertyClass,父类QObject.
3. mypropertyclass.h文件中Q\_OBJECT下方声明属性宏：

|  |
| --- |
|  |

1. 声明属性读取、设置函数，定义属性变更时发送的信号，定义成员变量m\_mask保存属性值。

|  |
| --- |
|  |

1. widget.h添加槽函数声明

|  |
| --- |
|  |

widget.cpp添加头文件：添加槽方法定义：

|  |
| --- |
|  |

1. Widget类构造函数中添加如下代码

|  |
| --- |
|  |

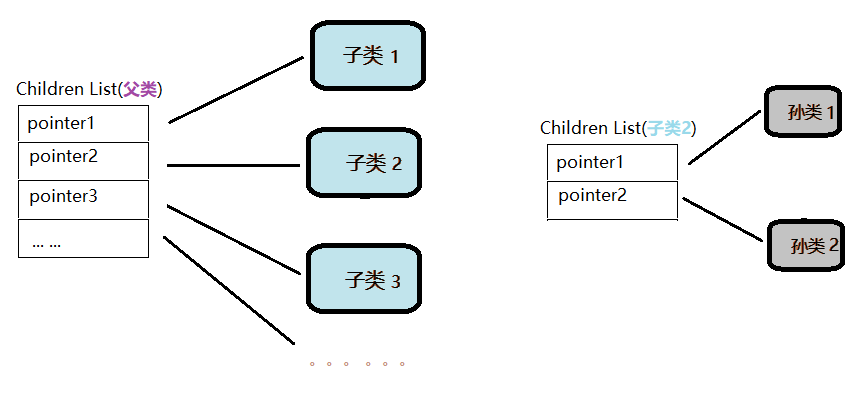
# 二、Qt对象树

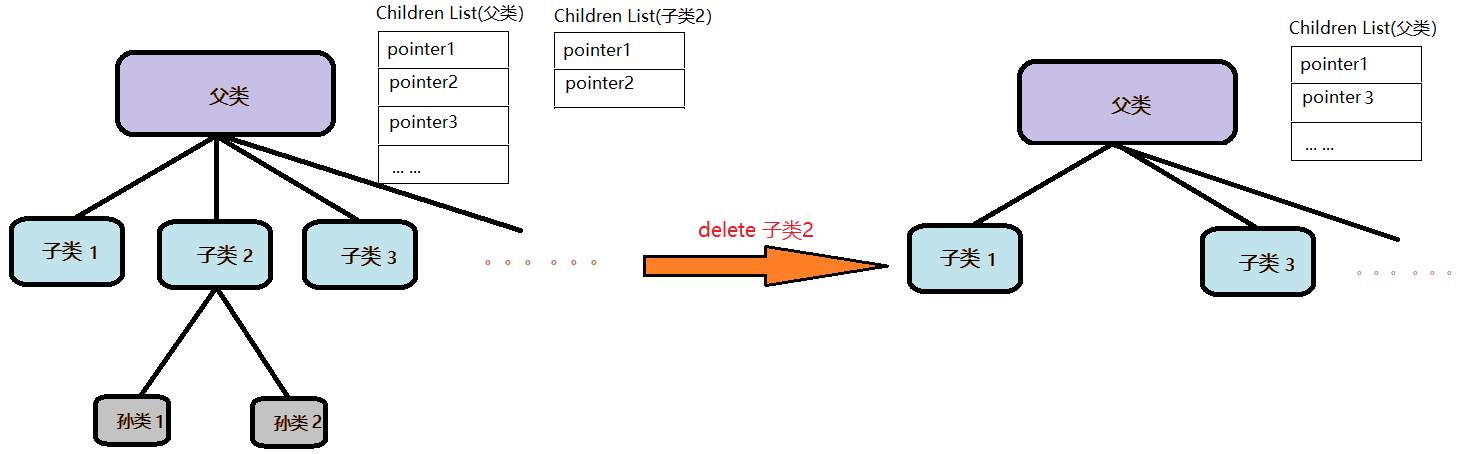
Qt提供了对象树机制，能够自动、有效的组织和管理继承自QObject的Qt对象。

每个继承自QObject类的对象通过它的对象链表（QObjectList）来管理子类对象，当用户创建一个子对象时，其对象链表相应更新子类对象信息，对象链表可通过children()获取。

当父对象析构的时候，其对象链表中的所有（子）对象也会被析构，父对象会自动将其从父对象列表中删除。Qt 保证没有对象会被 delete 两次。开发中手动回收资源时建议使用deleteLater代替delete,因deleteLater多次是安全的，而delete多次是不安全的。







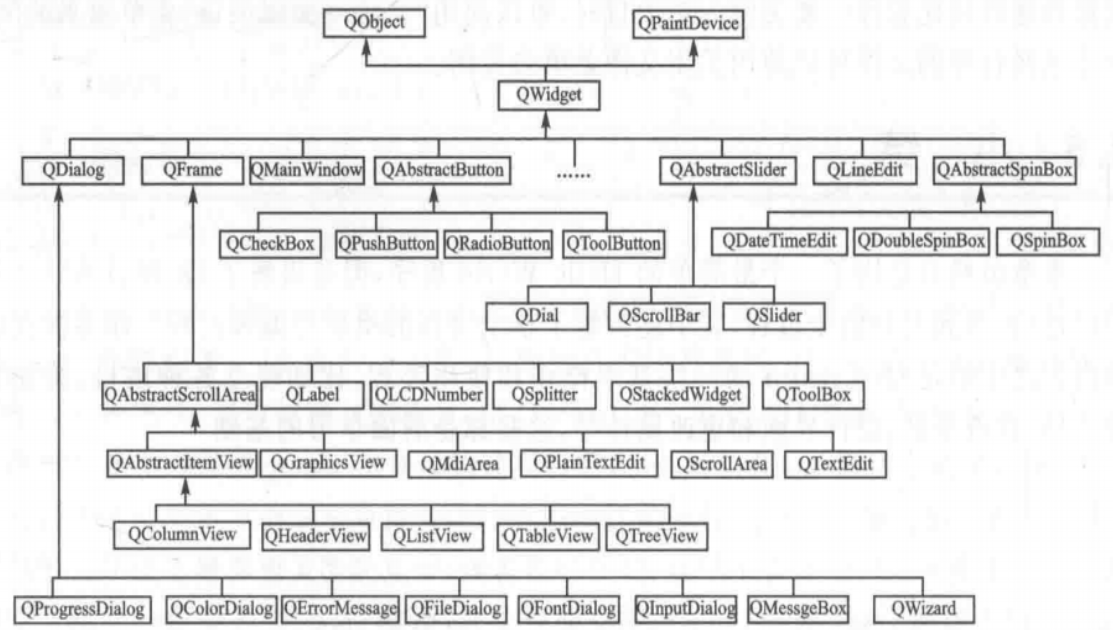
# 三、窗口部件

## 1.默认部件基类

QT提供的默认部件基类包括QMainWindow、QWidget、和QDialog这三种，这三个部件基类也是用的最多的。

QMainWindow是带有菜单栏、工具栏、状态栏的主窗口类，它有自己单独的布局。布局有一个中心区域，通常是标准的Qt部件，也可以是定制部件，且必须有一个中心小部件。setCentralWidget()方法可设置中心部件。

QWidget类是所有部件对象的基类，被称为基础窗口部件，继承关系详看QWidget类关系图。QWidget提供自我绘制和处理用户输入等基本功能，如接收鼠标、键盘和其它事件，并且在屏幕上绘制自己的表现。每一个窗口部件都是矩形，并且它们按Z轴顺序排列的。一个窗口部件可以被它的父窗口部件或者它前面的窗口部件盖住一部分。一个没有父窗口部件的窗口部件一直是顶级窗口部件。非顶级窗口部件是父窗口的子部件。QWidget构造函数有两个参数：QWidget \*parent = 0,Qt::WindowFlags f = 0。parent即父窗口，默认为0，即没有父窗口，是顶级窗口，如果指定parent值则当前窗体将会是一个子部件。Qt::WindowFlags是Qt::WindowType枚举值的组合，用来设置窗口的属性，f = 0表默认为Qt::Widget风格，其余窗口属性详见[附件表一](#_九、附件)。setWindowState（）可设置窗体的状态，参数由Qt::WindowStates枚举值指定，窗体状态如下表所示



QWidget类关系图

|  |  |
| --- | --- |
| **窗体状态** | **作用** |
| Qt::WindowNoState | 窗体为正常状态 |
| Qt::WindowMinimized | 窗体最小化 |
| Qt::WindowMaximized | 窗体最大化 |
| Qt::WindowFullScreen | 窗体全屏显示 |
| Qt::WindowActive | 窗体为活动窗体 |

QWidget设计模式属性设置：

* enabled 启用或禁用widget，默认启用。
* geometry widget的位置和尺寸。
* sizePolicy 设置widget在水平和垂直方向的伸缩策略以及伸缩因子(Stretch Factors)，所谓伸缩策略实际就是widget对待部件大小提示的策略，需结合布局管理器一起使用。伸缩策略可通过调用setSizePolicy()方法设置，大小提示可通过sizeHint()函数返回值获取，也可重载sizeHint()方法进行重置。

拉伸因子描述了各个部件在进行拉伸时，部件间以指定的比例进行拉伸，如水平布局的三个按钮其拉伸因子分别设为1,2,3则表示该3个按钮将会以1:2:3的比例进行拉伸。注意：1.当主窗口的大小不能按计算出来的比例容纳下所有子部件时，子部件不一定会按设计好的比例进行排列。

大小策略与拉伸因子之间的关系：

若部件的拉伸因子大于0，则按照拉伸因子的比例分配空间；若拉伸因子为0，则只有在其他部件不需要空间时才会获得空间；也就是说若一些部件拉伸因子大于0，而一些部件拉伸因子为0，则只有拉伸因子大于0的部件会被拉伸，而拉伸因子为0的部件不会被拉伸。若所有部件的拉伸因子都为0，则按照大小策略的规则对部件进行拉伸。注意：2.若部件的大小策略为Fixed，则即使设置了拉伸因子，该部件也不会被拉伸。故拉伸因子会使大小策略不起作用或失效(除了Fixed策略外)

|  |  |
| --- | --- |
| **属性值** | **作用** |
| Fixed | 尺寸不能改变，尺寸为sizeHint大小 |
| Minimum | 尺寸可以拉伸，尺寸可变范围：≥sizeHint |
| Maximum | 尺寸可以缩小，尺寸可变范围：minimumSizeHint ~ sizeHint |
| Preferred | 可以变大缩小，尺寸可变范围：≥minimumSizeHint |
| Expanding | 可以变大缩小，尺寸可变范围：≥minimumSizeHint，且部件有优先扩展权  (注：优先扩展权表部件将尽可能多的占用空间，如Preferred与Expanding同时存在则优先分配空间给Expanding) |
| MinimumExpanding | 尺寸可以拉伸，尺寸可变范围：≥minimumSizeHint，且部件有优先扩展权 |
| Ignored | 任意变大缩小，尺寸可变范围：≥minimumSizeHint  （若minimumSizeHint为0，则可缩小至0，此时部件不可见） |

* minimumSize 设置部件的最小尺寸，伸缩时不可继续缩小。
* maximumSize 设置部件的最大尺寸，伸缩时不可继续变大。
* palette 画板，设置部件的基本样式。
* mouseTracking 鼠标跟踪，鼠标离开部件范围后，部件仍可以接收到mousemove等事件。
* tabletTracking 平板或手机的陀螺仪、加速度传感器跟踪。
* focusPolicy 焦点策略，按钮可以通过NoFocus使虚线消失，lineedit这类文本编辑框必须能获得焦点。

|  |  |
| --- | --- |
| **属性值** | **作用** |
| NoFocus | 无法通过点击和键盘获得焦点 |
| TabFocus | 鼠标无法获得焦点，Tab键获得焦点 |
| ClickFocus | 鼠标点击获得焦点，Tab键无法获得焦点 |
| StrongFocus | 鼠标和Tab键都可以获得焦点 |
| WheelFocus | 通过滚轮获得焦点 |

* contextMenuPolicy 上下文菜单策略，指定菜单的显示方式。

|  |  |
| --- | --- |
| **属性值** | **作用** |
| NoContextMenu | 部件无菜单，菜单处理延迟到部件父亲 |
| PreventContextMenu | 部件无菜单，菜单处理不延迟到父亲，而是传递到本身。 |
| DefaultContextMenu | 调用部件的contextMenuEvent方法，默认处理方式为忽略上下文事件。 |
| ActionsContextMenu | 部件菜单由 actions定义构成 |
| CustomContextMenu | 部件菜单自定义，发送customContextMenuRequested信号 |

* acceptDrops 设置部件是否接受拖拽事件，默认启用。
* toolTip 设置部件的提示，鼠标悬浮时会显示。
* toolTipDuration 设置widgettoolTip的显示持续时间，毫秒为单位，默认为-1，会一直显示。
* statusTip 设置部件的状态提示，当窗口有statusBar时会显示在上面。
* whatsThis 某些窗口会有whatsThis按钮，选中后点击其他按钮会显示帮助信息
* accessibleName 辅助功能客户端应用程序所使用的控件名称。
* accessibleDescription主要用来为视力较差或盲人用户，提供更大的上下文，也可以使用上下文搜索或其他应用程序。
* layoutDirection widget布局的方向，LeftToRight，RightToLeft，LayoutDirectionAuto，字面意思
* autoFillBackground 设置widget背景是否被画板颜色自动填充，默认不勾选
* styleSheet 设置widget的qss样式表
* locale 设置widget的区域和语言，在将时间等信息转成字符串后，年月日几个字会显示不同语言
* inputMethodHint 设置widget输入时的屏幕键盘模式，有纯数字、纯字母等多种模式，用在手机端

QDialog是各种对话框的基类，其继承自QWidget,对话框有两种表现形式：模态对话框、非模态对话框。模态对话框就是阻塞同一应用程序中其它可视窗口的输入的对话框。用户必须完成当前对话框中的交互操作并且关闭窗口后才能操作应用程序中的其它窗口。模式对话框有它们自己的本地事件循环。exec()方法可使窗口以模态方式运行。当用户关闭这个对话框，exec()将提供一个可用的返回值并且这时流程控制继续从调用exec()的地方进行。通常，我们连接默认按钮，例如“OK”到accept()槽并且把“Cancel”连接到reject()槽，来使对话框关闭并且返回适当的值。另外我们也可以连接done()槽，传递给它Accepted或Rejected。

模态窗口运行：

（1）调用exec()方法，如

QDialog dlg;

dlg.exec();

（2）调用setModal()方法设置模态，如

QDialog dlg;

dlg.setModal(true);

dlg.show();

（3）QWidget widget;

widget.setWindowModality(Qt::ApplicationModal);

widget.show();

|  |  |
| --- | --- |
| **阻塞方式** | **阻塞效果** |
| Qt::ApplicationModal | 阻塞应用程序的所有窗口 |
| Qt::WindowModal | 阻塞阻塞父窗口、祖先窗口及它们的子窗口 |
| Qt::NoModal | 不阻塞，默认值 |

## 2.其他部件

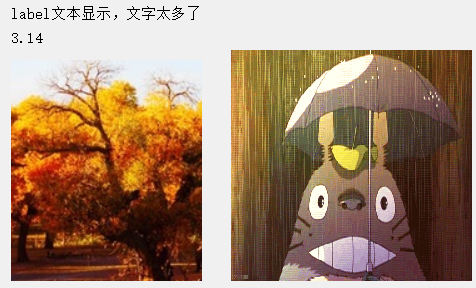
**QLabel**

常用来显示文本、数字、图片、gif动图。

新建桌面应用程序，项目名testQLabel,基类QWidget，类名Widget，勾选创建界面文件。

类构造函数中添加如下代码（准备好一个图片及gif格式动图）：

|  |
| --- |
|  |



可与其他部件设置伙伴关系，此时标签起到快捷键作用。如下图标签与按钮建立了伙伴关系，则按下快捷键Alt+P时，按钮便被按下了。



**QLineEdit**

接收用户输入。

显示模式

|  |  |
| --- | --- |
| **显示模式** | **效果** |
| QLineEdit::Normal | 正常显示 |
| QLineEdit::NoEcho | 不显示 |
| QLineEdit::Password | 密码显示 |
| QLineEdit::PasswordEchoOnEdit | 编辑时候正常显示，其他密码显示 |

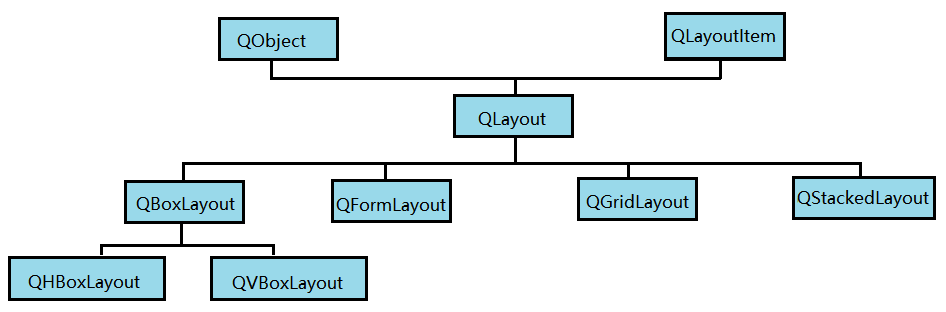
QLineEdit掩码字符

|  |  |  |
| --- | --- | --- |
| **字符（必须输入）** | **字符（可留空）** | **含义** |
| A | a | 只能输入A-Z,a-z |
| N | n | 只能输入A-Z,a-z,0~9 |
| X | x | 可以输入任意字符 |
| 9 | 0 | 只能输入0-9 |
| D | d | 只能输入1-9 |
|  | # | 只能输入+，-，0-9 |
| H | h | 只能输入十六进制字符(0-9,a-f,A-F) |
| B | b | 只能输入二进制字符(0,1） |
| > | | 后面的字母字符自动转换为大写 |
| < | | 后面的字母字符自动转换为小写 |
| ! | | 停止字母字符的大小写转换 |
| \ | | 将该表中的特殊字符正常显示用作分隔符 |

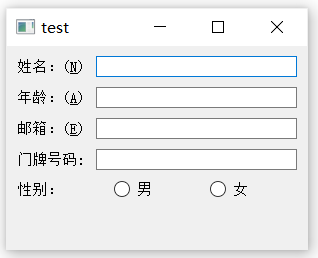
QPushButton 按钮类，常用显示文字、图标。

QRadioButton 单选按钮类，只能选一个，当将单选按钮加入一个分组框中时，加入的按钮便是一个分组，分组中只能选中一个，具有排他性。QCheckBox是多选按钮类，可多选。

# 四、布局管理器



利用布局管理器做出如下界面效果（标签与输入框设置伙伴关系）：



新建桌面应用程序，项目名TestLayout，基类QWidget,类名Widget,不勾选创建界面。

修改main函数：

|  |
| --- |
|  |

# 五、容器

## QString 类

隐式共享：隐式数据共享机制去最大化资源有效利用和最小化复制克隆操作。隐式共享类当作为函数参数传递的时候，不仅安全而且效率很高，因为传递的时候只是传递的数据的指针，数据本身只当自己被修改的时候才会去复制。简称写时复制。数据相同时，执行浅拷贝，仅复制指向数据块的指针，数据不同时则执行深拷贝。隐式共享技术可以大大降低对内存的开销及CPU资源的消耗，很大程度提高应用程序的运行效率。

QString类保存了16位的Unicode值，Unicode把所有语言都统一到一套编码里，Unicode提供[UTF-8](https://baike.baidu.com/item/UTF-8" \t "https://baike.baidu.com/item/UniCode/_blank)、[UTF-16](https://baike.baidu.com/item/UTF-16" \t "https://baike.baidu.com/item/UniCode/_blank)、[UTF-32](https://baike.baidu.com/item/UTF-32" \t "https://baike.baidu.com/item/UniCode/_blank)编码方式。UTF-8以字节为单位对Unicode进行编码，对不同范围的字符使用不同长度的编码。对于0x00-0x7F之间的字符，UTF-8编码与[ASCII编码](https://baike.baidu.com/item/ASCII%E7%BC%96%E7%A0%81" \t "https://baike.baidu.com/item/UniCode/_blank)完全相同。

QString s1 = s0;处设置断点后调试运行。

|  |  |
| --- | --- |
| **断点调试快捷键** | **功能** |
| F9 | 设置/取消断点 |
| F5 | 开始调试/调到下一个断点 |
| F10 | 单步 |
| F11 | 单步进入 |
| Shift + F11 | 单步跳出 |
| Ctrl + F10 | 调到指定行 |
| Shift + F5 | 结束断点调试 |

新建控制台应用程序，项目名TestQString,main函数中添加如下代码：

|  |
| --- |
|  |

构造s1:

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

QString常用初始化方式：

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

QString常用方法

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

## QStringList类

|  |
| --- |
|  |

遍历链表

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

## QList类 QLinkedList类

QList<T>是最常用的容器类。QList以列表形态存储并管理数据，并能进行基于快速索引的访问，也可以进行快速的数据删除操作。继承自QList类的子类有QItemSelection、QQueue、QStringList、QTestEventList.

可用运算符<< 、append()、prepend()进行添加元素，insert()进行插入。QList<T>维护了一个指针数组，数组元素指向每一个链表项，因此QList<T> 提供了基于下标的快速访问。

Java风格迭代器迭代点位置：第一个列表项前、两个列表项中间、最后一个列表项之后。STL风格迭代器迭代点位置：直接指向列表项。

Java风格迭代器

|  |  |  |
| --- | --- | --- |
| **容器类** | **只读迭代器** | **读写迭代器** |
| QList<T>,QQueue<T> | QListIterator<T> | QMutableListIterator<T> |
| QLinkedList<T> | QLinkedListIterator<T> | QMutableLinkedListIterator<T> |
| QVector<T>,QStack<T> | QVectorIterator<T> | QMutableVectorIterator<T> |

STL风格迭代器

|  |  |  |
| --- | --- | --- |
| **容器类** | **只读迭代器** | **读写迭代器** |
| QList<T>,QQueue<T> | QList<T>::const\_iterator | QList<T>::iterator |
| QLinkedList<T> | QLinkedList<T>::const\_iterator | QLinkedList<T>::iterator |
| QVector<T>,QStack<T> | QVector<T>::const\_iterator | QVector<T>::iterator |

QLinkedList<T>是一个链式列表，不能使用下标访问，与QList相比，当对一个很大的列表进行插入操作时，QLinkedList更高效。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

## QVector类

QVector在相邻的内存中存储给定数据类型T的一组数据。在QVector前部或中间位置插入操作速度都很慢，因会导致内存中大量的数据移动。访问数据可使用下标，也可使用迭代器。继承自QVector类的子类有QPolygon、QPolygonF、QStack。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

## QMap类、 QHash类

QMap与QHash差别：

①QHash比QMap查找速度更快。

②QHash以任意顺序存储，QMap以Key顺序存储数据。

③QHash的Key必须提供operator==()及一个全局的qHash(Key)函数，而QMap的Key必须提供operator<()函数。

存储一键多值数据时可使用QMultiMap<Key,T>容器或QMultiHash<Key,T>容器。

Java风格迭代器

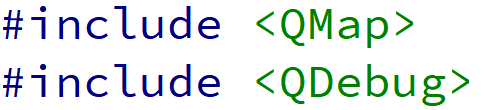
|  |  |  |
| --- | --- | --- |
| **容器类** | **只读迭代器** | **读写迭代器** |
| QMap<Key,T>,QMultiMap<Key,T> | QMapIterator<Key,T> | QMutableMapIterator<Key,T> |
| QHash<Key,T>,QMultiHash<Key,T> | QHashIterator<Key,T> | QMutableHashIterator<Key,T> |

STL风格迭代器

|  |  |  |
| --- | --- | --- |
| **容器类** | **只读迭代器** | **读写迭代器** |
| QMap<Key,T>,QMultiMap<Key,T> | QMap<Key,T>::const\_iterator | QMap<Key,T>::iterator |
| QHash<Key,T>,QMultiHash<Key,T> | QHash<Key,T>::const\_iterator | QHash<Key,T>::iterator |

新建控制台应用程序，项目名称为testQMap

main.cpp添加头文件



main函数中添加代码

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

STL风格迭代器

|  |
| --- |
|  |

**QMultiMap类**

|  |
| --- |
|  |

**QHash**

|  |
| --- |
|  |

## QVariant类

QVariant类是Qt的共用体union数据类型，不仅能保存很多Qt类型的值，包括QColor,QBrush,QFont,QRect,QString及QSize等等，而且能存放Qt的容器类型值。

可采用toT()将QVariant对象转换称为T类型并获取其值。如toInt(),toFloat(),toString()，可用type方法获取数据类型，判断数据是否可。对于GUI模块中定义的数据类型没有相应的转换函数，即没有toColor(),没有toImage()这样的函数提供，可通过value()方法实现转换，判断两种类型是否可以转换时可通过canConvert()方法的返回值判断。

新建桌面应用程序，基类选择QWidget，widget.cpp添加头文件



Widget类构造添加如下代码：

|  |
| --- |
|  |

# 标准对话框

|  |  |  |  |
| --- | --- | --- | --- |
| **对话框类** | **说明** | **静态函数** | **函数说明** |
| QFileDialog | 文件对话框 | getOpenFileName() | 选择打开一个文件 |
| getOpenFileNames() | 选择打开多个文件 |
| getSaveFileName() | 选择保存一个文件 |
| getExistingDirectory() | 选择一个己有的目录 |
| getOpenFileUrl() | 选择打幵一个文件，可选择远程网络文件 |
| QColorDialog | 颜色对话框 | getColor() | 选择颜色 |
| QFontDialog | 字体对话框 | QFont getFont() | 选择字体 |
| QInputDialog | 输入对话框 | getText() | 输入单行文字 |
| getlnt() | 输入整数 |
| getDouble() | 输入浮点数 |
| getltem() | 从一个下拉列表框中选择输入 |
| getMultiLineText() | 输入多行字符串 |
| QMessageBox | 消息框 | information() | 信息提示对话框 |
| question() | 询问并获取是否确认的对话框 |
| waming() | 警告信息提示对话框 |
| critical() | 错误信息提示对话框 |
| about() | 设置自定义信息的关于对话框 |
| aboutQt() | 关于Qt的对话框 |

新建桌面应用程序，项目名testStandardDialogs，类名Dialog,基类QDialog,不勾选创建界面文件。

编辑dialog.h文件

|  |
| --- |
|  |

编辑dialog.cpp编辑构造函数及添加相应成员函数的定义

|  |
| --- |
|  |

当前项目添加C++类，类名InputDlg,基类QDialog

编辑inputdlg.h文件

|  |
| --- |
|  |

编辑InputDlg构造函数及添加相应槽函数的定义

|  |
| --- |
|  |

项目添加添加C++类，类名MsgBoxDlg，基类QDialog

msgboxdlg.h文件

|  |
| --- |
|  |

msgboxdlg.cpp文件编辑构造函数，添加成员函数的定义

|  |
| --- |
|  |

# 正则表达式

正则表达式即一个文本匹配字符串的一种模式，Qt中QRegExp类实现使用正则表达式进行模式匹配，且完全支持Unicode，主要应用：字符串验证、搜索、查找替换、分割。

正则表达式中字符及字符集

|  |  |
| --- | --- |
| **元素** | **含义** |
| c | 匹配字符本身，如a匹配a |
| \c | 跟在\后面的字符匹配字符本身，但本表中下面指定的这些字符除外。 |
| \a | 匹配ASCII的振铃 |
| \f | 匹配ASCII的换页 |
| \n | 匹配ASCII的换行 |
| \r | 匹配ASCII的回车 |
| \t | 匹配ASCII的水平制表符 |
| \v | 匹配ASCII的垂直制表符 |
| \xhhhh | 匹配Unicode字符对应的十六进制数 |
| \0ooo | 匹配八进制的ASCII/Latin1字符 |
| . | 匹配任意字符 |
| \d | 匹配任意一个数字 |
| \D | 匹配一个非数字 |
| \s | 匹配一个空白字符，包括“\t”、“\n”、“\v”、“\f”、“\r”及“” |
| \S | 匹配一个非空白字符 |
| \w | 匹配一个单词字符，包括任意字符数字下划线，即A~Z,a~z,0~9中任意一个 |
| \W | 匹配一个非单词字符 |
| \n | 第n个反向引用 |

正则表达式中的量词

|  |  |
| --- | --- |
| **量词** | **含义** |
| E? | 匹配0次或1次等价于E{0,1} |
| E+ | 匹配1次或多次，等价于E{1，} |
| E\* | 匹配0次或多次，等价于E{0，} |
| E{n} | 匹配n次 |
| E{n,} | 匹配至少n次 |
| E{,m} | 匹配至多m次 |
| E{n,m} | 匹配至少n次，至多m次 |

正则表达式中的断言

|  |  |
| --- | --- |
| **断言** | **含义** |
| ^ | 标志字符串的开始。若匹配“^”则使用“\\^” |
| $ | 标志字符串的结尾。若匹配“$”则使用“\\$” |
| \b | 一个单词的边界 |
| \B | 一个非单词的边界。当\b为false则它为true |
| （?=E） | 表达式后紧跟E才匹配 |
| （?!E） | 表达式后不跟E才匹配 |

QRegExp同时支持通配符

|  |  |
| --- | --- |
| **通配符** | **含义** |
| c | 任意一个字符，表字符本身 |
| ？ | 任意一个字符，类似regexp中“.” |
| \* | 任意0个或多个字符 |
| [...] | 在[]中的字符集 |

新建控制台应用程序，项目名QRegExp

编辑main.cpp文件

|  |
| --- |
|  |

# 文件处理

QFile类用于文件操作，它提供了读写文件的接口，可以读写文件、二进制文件和Qt资源文件。

处理文本文件和二进制文件，可以使用QTextStream类和QDataStream类。处理临时文件可以使用QTemporaryFile,获取信息可以使用QFileInfo，处理目录可以使用QDir，监视文件和目录变化可以使用QFileSystemWatcher.

QTextStream的流操作符

|  |  |
| --- | --- |
| 操作符 | 作用描述 |
| bin | 设置读写的整数为 二进制数 |
| oct | 设置读写的整数为 八进制数 |
| dec | 设置读写的整数为十进制数 |
| hex | 设置读写的整数为十六进制数 |
| showbase | 强制显示进制前缀，如十六进制(0x)、八进制(0)、二进制(0b) |
| forcesign | 强制显示符号(+、-) |
| forcepoint | 强制显示小数点 |
| noshowbase | 强制不显示前缀 |
| noforcesign | 强制不显示符号 |
| uppercasebase | 显示大写的进制前缀 |
| lowercasebase | 显示小些的进制前缀 |
| uppercasedigits | 用大写字母表示 |
| lowercasedigits | 用小写字母表示 |
| fixed | 固定小数点表示 |  |
| scientific | 科学计数法表示 |  |
| left | 左对齐 |  |
| right | 右对齐 |  |
| center | 居中 |  |
| endl | 换行 |  |
| flush | 清除缓冲 |  |

QFile QTextStream操作文件示例：

新建控制台引用程序，编辑main函数,main.cpp添加头文件：

|  |
| --- |
|  |

读写二进制文件示例：

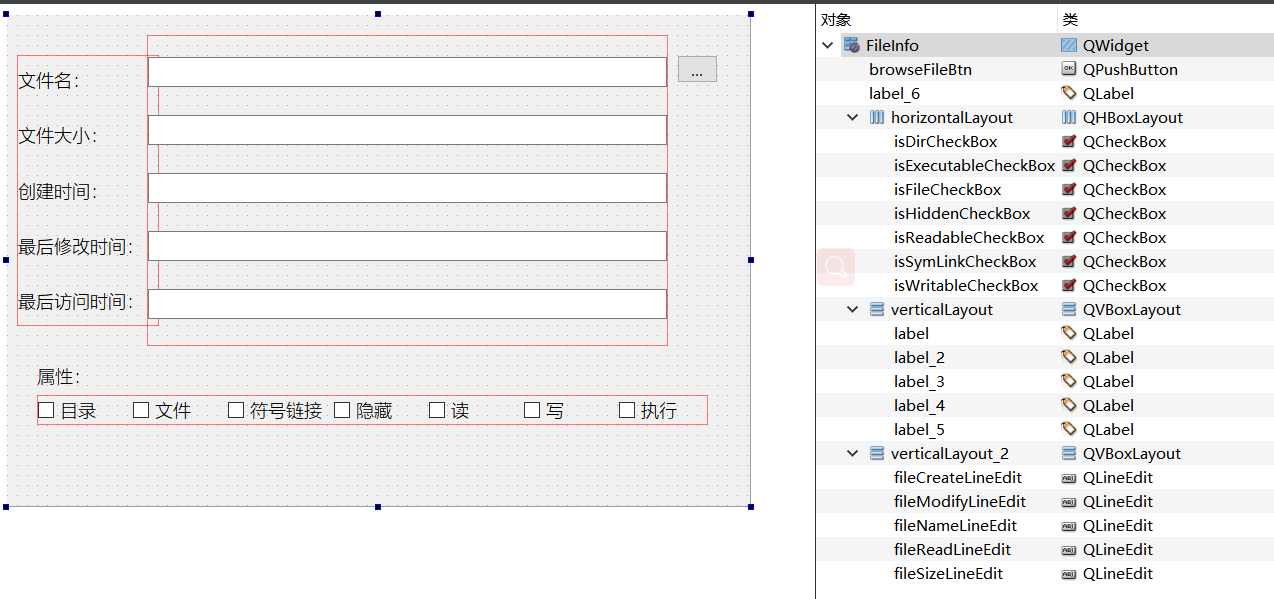
新建控制台应用程序，编辑main函数，main.cpp添加头文件

|  |
| --- |
|  |

QFileInfo类获取文件信息示例：

新建桌面应用程序，基类QWidget,类名FileInfo，勾选创建界面文件

设计模式下设计界面



fileinof.h文件添加成员函数及槽函数声明：

|  |
| --- |
| ... ...    ... ... |

fileinfo.cpp编辑构造函数并添加成员函数定义：

|  |
| --- |
|  |

|  |
| --- |
|  |

# 九、附件

表1：

|  |  |
| --- | --- |
| Qt:WindowFlags | 释义 |
| Qt::Widget | QWidget构造函数的默认值，如果新的窗口部件没有父窗口部件，则它是一个独立的窗口，否则就是一个子窗口部件。另请参见Qt :: Window和Qt :: SubWindow。 |
| Qt::Window | 无论是否有父窗口部件，新窗口部件都是一个窗口，通常有一个窗口边框和一个标题栏 。请注意，如果部件没有父控件，则无法取消设置此标记（即如果存在父控件，则可以取消此标记，否则，必须存在此标记用以标识此控件是个独立窗口）。 |
| Qt:: Dialog | 新窗口部件是一个对话框，它是QDialog构造函数的默认值。如果要将其用作模态对话框，则应从另一个窗口启动它，或者如果有父窗口，则与QWidget::windowModality属性一起使用。如果将其设置为模态，则对话框将阻止应用程序中的其他顶级窗口获取任何输入。我们将具有父级的顶级窗口称为辅助窗口。 |
| Qt::Sheet | 新窗口部件是一个Macintosh表单(sheet) 。由于使用工作表意味着窗口模态，推荐的方法是使用QWidget :: setWindowModality（）或QDialog :: open（）。 |
| Qt:: Drawer | 新窗口部件是一个Macintosh抽屉(drawer) |
| Qt::Popup | 新窗口部件是一个弹出式顶层窗口 |
| Qt::Tool | 新窗口部件是一个工具(tool)窗口，它通常是一个用于显示工具按钮的小窗口。如果一个工具窗口有父窗口部件，则它将显示在父窗口的部件上面，否则相当于使用了Qt::WindowStaysOnTopHint提示。如果窗口系统支持它，工具窗口可以用更轻的框架装饰。它也可以与Qt :: FramelessWindowHint结合使用。 |
| Qt::ToolTip | 表示窗口小部件是工具提示。 这在内部用于实现工具提示，没有标题栏和窗口边框。 |
| Qt:: Desktop | 新窗口部件是桌面，它是QDesktopWidget构造函数的默认值 |
| Qt::SplashScreen | 新窗口部件是一个欢迎窗口，它是SplashScreen构造函数的默认值。 |
| Qt::SubWindow | 新窗口部件是一个子窗口，而无论窗口部件是否有父窗口部件。此外，Qt还定义了一些控制窗口外观的窗口提示（这些窗口提示仅对顶层窗口有效） |
| Qt::ForeignWindow | 指示此窗口对象是表示由另一个进程创建的本机平台窗口或手动使用本机代码的句柄。 |
| Qt::CoverWindow | 表示该窗口代表一个封面窗口，例如，当应用程序在BlackBerry平台上最小化时显示。 |
| Qt::MSWindowFiredSizeDialogHint | 为Windows系统上的窗口装饰一个窄的对话框边框，通常这个提示用于固定大小的对话框 |
| Qt::MSWindowOwnDC | 为Windows系统上的窗口添加自身的显示上下文菜单 |
| Qt::BypassWindowManagerHint | 此标志可用于向平台插件指示应禁用“所有”窗口管理器协议。 此标志的行为会有所不同，具体取决于运行应用程序的操作系统以及正在运行的窗口管理器。 该标志可用于获取没有配置集的本机窗口。 |
| Qt::X11BypassWindowManagerHint | 完全忽视窗口管理器，它的作用是产生一个根本不被管理的无窗口边框的窗口(此时，用户无法使用键盘进行输入，除非手动调用QWidget::activateWindow()函数) |
| Qt::FramelessWindowHint: | 产生一个无窗口边框的窗口，此时用户无法移动该窗口和改变它的大小 |
| Qt::NoDropShadowWindowHint | 在支持平台上禁用窗口投影 |
| Qt::CustomizeWindowHint | 关闭默认的窗口标题提示 |
| Qt::WindowTitleHint | 为窗口装饰一个标题栏 |
| Qt::WindowSystemMenuHint | 为窗口添加一个窗口系统系统菜单，并尽可能地添加一个关闭按钮 |
| Qt::WindowMinimizeButtonHint | 为窗口添加一个“最小化”按钮 |
| Qt::WindowMaximizeButtonHint | 为窗口添加一个“最大化”按钮 |
| Qt::WindowCloseButtonHint | 为窗口添加一个“关闭”按钮 |
| Qt::WindowMinMaxButtonHint | 为窗口添加一个“最小化”按钮 和一个“最大化”按钮 |
| Qt::WindowContextHelpButtonHint | 为窗口添加一个“上下文帮助”按钮 |
| Qt::WindowStaysOnTopHint | 告知窗口系统，该窗口应该停留在所有其他窗口的上面 |
| Qt::WindowType\_Mask | 一个用于提示窗口标识的窗口类型部分的掩码 |