[QregExp 正则表达式的表示类 3](#_Toc466191342)

[控件 3](#_Toc466191343)

[按钮组 Buttons 3](#_Toc466191344)

[输入部件组 Input Widgets 3](#_Toc466191345)

[显示控件组 Display Widgets 4](#_Toc466191346)

[Spacers 空间间隔组 4](#_Toc466191347)

[Layouts 布局管理组 4](#_Toc466191348)

[容器组 Containers 4](#_Toc466191349)

[项目视图组 Item Views 5](#_Toc466191350)

[MVC设计模式：三元素 5](#_Toc466191351)

[项目控件组 Item Widgets 5](#_Toc466191352)

[Qlabel 5](#_Toc466191353)

[QWidget 6](#_Toc466191354)

[QComboBox 6](#_Toc466191355)

[QTranslator 6](#_Toc466191356)

[QListWidget 7](#_Toc466191357)

[QListWidgetItem 7](#_Toc466191358)

[问题 9](#_Toc466191359)

[在Qt中作用中文的问题 9](#_Toc466191360)

[designer无法启动的问题 9](#_Toc466191361)

[按F1提示你没有相关文档 9](#_Toc466191362)

[关于Qt4widget内存分配和setObjectName作用的问题(父类会自动delete子类) parent-child内存管理机制 10](#_Toc466191363)

[Qt Designer中自定义控件的使用(提升法与插件法) 10](#_Toc466191364)

[使用designer 11](#_Toc466191365)

[自定义控件 11](#_Toc466191366)

[提升 vs 插件 11](#_Toc466191367)

[还有点什么... 12](#_Toc466191368)

## QApplication

QApplication类管理GUI应用程序的控制流和主要设置。

QApplication用一些基于QWidget的应用程序所需的功能专门化QGuiApplication。它处理部件特定的初始化,终结。

对于任何使用Qt的 GUI应用程序，无论同时具有1个、2个、或者更多个窗口，它都有且仅有一个QApplication对象。基于non-QWidget 的Qt应用程序，使用QGuiApplication代替QApplication，因为它不依赖于QtWidgets的库。

一些GUI应用程序提供一个特殊的批处理模式. 为执行任务提供不需要人工干预的命令行参数。在这种非gui模式, 实例化一个纯QCoreApplication通常就足够了，这样做避免了初始化不必要的图形用户界面所需的资源。下面的例子展示了如何动态地创建一个适当的类型的应用程序实例:

[**QCoreApplication**](../qtcore/qcoreapplication.html)**\* createApplication(int &argc, char \*argv[])**

**{**

**for (int i = 1; i < argc; ++i)**

**if (!qstrcmp(argv[i], "-no-gui"))**

**return new** [**QCoreApplication**](../qtcore/qcoreapplication.html)**(argc, argv);**

**return new** [**QApplication**](qapplication.html#QApplication)**(argc, argv);**

**}**

**int main(int argc, char\* argv[])**

**{**

[**QScopedPointer**](../qtcore/qscopedpointer.html)**<**[**QCoreApplication**](../qtcore/qcoreapplication.html)**> app(createApplication(argc, argv));**

**if (qobject\_cast<**[**QApplication**](qapplication.html#QApplication) **\*>(app.data())) {**

**// start GUI version...**

**} else {**

**// start non-GUI version...**

**}**

**return app->exec();**

**}**

QApplication可以通过instance ()函数实例化, 它返回一个指向应用程序的QCoreApplication(或QGuiApplication / QApplication)的实例。如果没有分配实例,返回null。

[QApplication](qapplication.html)的主要责任:

* 初始化应用程序与用户的桌面设置，诸如：[palette](qapplication.html#palette)(), [font](qapplication.html#font)() and [doubleClickInterval](qapplication.html#doubleClickInterval-prop)().它在用户改变这些全局桌面时跟踪这些属性，例如通过某种形式的控制面板.
* 它执行事件处理,也就是说,它从底层窗口系统接收事件并将它们分派到相关部件. 通过使用sendEvent()和postEvent()你可以发送自己的事件到窗口。
* 它解析常见的命令行参数,并相应地设置其内部状态
* 它通过一个封装在QStyle对象定义应用程序的外观和感觉,通过setStyle()可以在运行时改变。.
* 它指定了应用程序是如何分配颜色，详情见setColorSpec()。
* 它提供了本地化的字符串,可见用户通过翻译()，通过translate ()对用户可见。
* 它提供了一些神奇的对象，如[desktop](file:///C:\Users\JG\Desktop\qapplication.html#desktop) ()和[clipboard](file:///C:\Users\JG\qtgui\qguiapplication.html#clipboard) ()
* 它知道应用程序的窗口. 你可以通过它知道窗口在什么位置，通过使用[widgetAt](qapplication.html#widgetAt)(),[topLevelWidgets](qapplication.html#topLevelWidgets)() and [closeAllWindows](qapplication.html#closeAllWindows)(), 等.
* 它管理应用程序的鼠标光标处理, 见 [setOverrideCursor](../qtgui/qguiapplication.html#setOverrideCursor)()

因为QApplication对象要做这么多的初始化, 它必须在其他与用户界面相关的对象之前创建. QApplication也处理常见的命令行参数. 因此,在创建它之前解释或修改应用程序的argv是一个好方法。

| **Groups of functions 函数功能分组** | |
| --- | --- |
| System settings  系统设置 | [desktopSettingsAware](../qtgui/qguiapplication.html#desktopSettingsAware)(), [setDesktopSettingsAware](../qtgui/qguiapplication.html#setDesktopSettingsAware)(), [cursorFlashTime](qapplication.html#cursorFlashTime-prop)(), [setCursorFlashTime](qapplication.html#cursorFlashTime-prop)(), [doubleClickInterval](qapplication.html#doubleClickInterval-prop)(), [setDoubleClickInterval](qapplication.html#doubleClickInterval-prop)(), [setKeyboardInputInterval](qapplication.html#keyboardInputInterval-prop)(), [wheelScrollLines](qapplication.html#wheelScrollLines-prop)(), [setWheelScrollLines](qapplication.html#wheelScrollLines-prop)(), [palette](qapplication.html#palette)(), [setPalette](qapplication.html#setPalette)(), [font](qapplication.html#font)(), [setFont](qapplication.html#setFont)(), [fontMetrics](qapplication.html#fontMetrics)(). |
| Event handling  事件处理 | [exec](qapplication.html#exec)(), processEvents(), exit(), quit(). sendEvent(), postEvent(), sendPostedEvents(), removePostedEvents(), hasPendingEvents(), [notify](qapplication.html#notify)(). |
| GUI StylesGUI样式 | [style](qapplication.html#style)(), [setStyle](qapplication.html#setStyle)(). |
| Color usage  颜色用法 | [colorSpec](qapplication.html#colorSpec)(), [setColorSpec](qapplication.html#setColorSpec)(). |
| Text handling  文本处理 | installTranslator(), removeTranslator() translate(). |
| Widgets  部件 | [allWidgets](qapplication.html#allWidgets)(), [topLevelWidgets](qapplication.html#topLevelWidgets)(), [desktop](qapplication.html#desktop)(), [activePopupWidget](qapplication.html#activePopupWidget)(), [activeModalWidget](qapplication.html#activeModalWidget)(), [clipboard](../qtgui/qguiapplication.html#clipboard)(), [focusWidget](qapplication.html#focusWidget)(), [activeWindow](qapplication.html#activeWindow)(), [widgetAt](qapplication.html#widgetAt)(). |
| Advanced cursor handling  高级的光标处理 | [overrideCursor](../qtgui/qguiapplication.html#overrideCursor)(), [setOverrideCursor](../qtgui/qguiapplication.html#setOverrideCursor)(), [restoreOverrideCursor](../qtgui/qguiapplication.html#restoreOverrideCursor)(). |
| Miscellaneous  杂项 | [closeAllWindows](qapplication.html#closeAllWindows)(), startingUp(), closingDown(). |

**See also** [QCoreApplication](../qtcore/qcoreapplication.html), [QAbstractEventDispatcher](../qtcore/qabstracteventdispatcher.html), [QEventLoop](../qtcore/qeventloop.html), and [QSettings](../qtcore/qsettings.html).

## QregExp 正则表达式的表示类

表达式（expressions）: E

量词(quantifiers)：E? E+ E\* E[n] E[n,] E[,m] E[n,m] ----E表示表达式

断言(assertions): ^ $ \b \B (?=E) (?!E)

## Horizontal水平

## Vertical垂直

## Graphics 图形

# 控件

## 按钮组 Buttons

Push Button 按钮

Tool Button 工具按钮

Radio Button 单选按钮

Check Box 复选按钮

Connand Link Button 命令链接按钮

Button Box 按钮盒

## 输入部件组 Input Widgets

Combo Box 组合框

Font Combo Box 字体组合框

Line Edit 行编辑

Text Edit 文本编辑

Plain Text Edit 纯文本编辑

Spin Box 数字显示框（自旋盒）

Double Spin Box 双自旋盒

Time Edit 时间编辑

Date Edit 日期编辑

Date/Time Edit 日期/时间编辑

Dial 拨号

Horizontal Scroll Bar 横向滚动条

Vertical Scroll Bar 垂直滚动条

Horizontal Slider 横向滑块

Vertical Slider 垂直滑块

Key sequence Edit 按键序列编辑

## 显示控件组 Display Widgets

Label 标签

Text Browser 文本浏览器

Calendar 日历

LCD Number 液晶数字

Progress Bar 进度条

Horizontal Line 水平线

Vertical Line 垂直线

Open GL Widget 开放式图形库开具

QquickWidget 嵌入QML工具

QwebView Web视图

## Spacers 空间间隔组

Horizontal Spacer 水平间隔

Vertical Spacer 垂直间隔

## Layouts 布局管理组

Vertical Layout 垂直布局---- QVBoxLayout

Horizontal Layout 横向（水平）布局---QHBoxLayout

Grid Layout 网格布局---QGridLayout

Form Layout 表单布局---QFormLayout

## 容器组 Containers

Group Box 组框

Scroll Area 滚动区域

Tool Box 工具箱

Tab Widget 标签小部件

Stacked Widget 堆叠部件

Frame 帧

Widget 小部件

MdiArea MDI区域

Dock Widget 停靠窗体部件

Qax Widget 封装Flash的ActiveX控件

## 项目视图组 Item Views

List View 清单视图

Tree View 树视图

Table View 表视图

Column View 列视图

## MVC设计模式：三元素

模型 Model ---表示数据(所有模型都基于QabstractItemModel抽象基类)

视图 View ---用户界面(所有视图都继承QabstractItemView抽象基类)

控制 Controller ---定义了用户在界面上的操作

MVC模式有效地分离了数据和用户界面

proxy代理, standard 标准, column列、专栏, abstract 摘要

Strategy 策略

Manual 手动 Submit 提交

## 项目控件组 Item Widgets

List Widget 清单控件

Tree Widget 树形控件---Qt的树形控件QtreeWidget--QtreeWidgeItem树形控件的树节点

Table Widget 表控件

Item 项目

QVariant

Qbrush

QtAlgorithms QtGlobal 提供了常用的算法

### Qlabel

QLabel imageLabel =new QLabel;

imageLabel->setScaledContents(true);//设置为填充全部有效空间

scaledContents : bool

这个属性保存标签的其内容是否填满所有可用空间。

当启用标签显示一个象素映射时，它将象素映射填充标签的全部可用的空间。这个属性默认是false.

Access functions:

|  |  |
| --- | --- |
| bool | hasScaledContents() const |
| void | setScaledContents(bool) |

### QPixmap与QLabel的显示问题

在QLabel中通过QPixmap显示图片。使图片按原比例显示在QLabel中。

想法一（实测行不通）：期望通过[**QPixmap**](qpixmap.html#QPixmap-3)(const QSize &*size*)在构建QPixmap时指定大小为lab的大小。之后再加载图片。----该方法行不通：因为在load时QPixmap对象的大小会发生改变。

qDebug() << lab.size(); //

QPixmap photo\_temp = QPixmap(lab.size()); //希望指定photo\_temp的大小为lab.size()

qDebug() << photo\_temp.size(); //此处输出的结果确实与相同

photo1\_temp.load(photo\_name); //加载图片（photo\_name为图片的全路径）

qDebug() << photo\_temp.size(); //此处输出的结果与不同，输出的大小为所加载的图片的大小 lab.setPixmap(photo\_temp);

可行的方法：

先加载图片到QPixmap对象，再对QPixmap对象进行缩放。并用重写*resizeEvent*事件，以保证lab大小变化时，显示的图片大小跟随变化。

QPixmap photo\_temp = QPixmap(photo\_name);

QPixmap photo = photo\_temp ->scaled(lab.size(),

Qt::KeepAspectRatio); // Qt::KeepAspectRatio表示保持图片的长宽比例

void Widget::*resizeEvent*(QResizeEvent \*event)

{

Q\_UNUSED(event); //避免因未使用event而产生警告。

QPixmap photo\_temp = photo1->scaled(ui->lab\_photo1->size(),

Qt::KeepAspectRatio);

ui->lab\_photo1->setPixmap(photo\_temp);

update();

}

### [Q](file:///C:\Users\JG\Desktop\qwidget.html#QWidget)Widget

[Q](qwidget.html" \l "QWidget)Widget \*QMainWindow::centralWidget() const

返回中央主窗口小部件。如果没有设置中央小部件这个函数返回0。

void QMainWindow::setCentralWidget([QWidget](qwidget.html#QWidget) \*widget)

设置给定部件为主窗口的中心部件。

Note: [QMainWindow](file:///C:\Users\JG\Desktop\qmainwindow.html)获得小部件的指针并在在适当的时候删除它。

### QFrame

setFrameStyle(QFrame::Box);//设置边框

### QComboBox

QComboBox listComboBox =new QComboBox;

listComboBox->addItem("Standard");

listComboBox->addItem("QTextListFormat::ListDisc");

listComboBox->addItem("QTextListFormat::ListCircle");

### QScrollArea

## QTranslator

提供国际化支持文本输出

这个类的一个对象包含一组从源语言到目标语言的翻译。QTranslator提供函数来在翻译文件中查找翻译。翻译文件使用Qt的语言学家创建。

最常见的使用QTranslator:: load加载一个翻译文件,安装使用QCoreApplication::installTranslator(),并通过QObject:tr()使用。这里有一个例子main()函数使用QTranslator

int main(int argc, char \*argv[])

{

QApplication app(argc, argv);

QTranslator translator;

// look up e.g. :/translations/myapp\_de.qm

if (translator.load(QLocale(), QLatin1String("myapp"), QLatin1String("\_"), QLatin1String(":/translations")))

app.installTranslator(&translator);

QPushButton hello(QCoreApplication::translate("main", "Hello world!"));

hello.resize(100, 30);

hello.show();

return app.exec();

}

## QListWidget

QListWidget类提供了一个基于项目列表小部件。

QListWidget使用一个内部的模型管理列表中的每个QListWidgetItem。

创建一个QListWidget

**QListWidget \*listWidget = new QListWidget(this);**

selectionMode()保存了在列表中能同时选中的选项个数，以及是否可以创建复杂的选择项。

可以通过 [setSelectionMode](qabstractitemview.html#selectionMode-prop)() 改变这些设置.

有如下方法可以将项目添加到列表:

**1．**

**new** [**QListWidgetItem**](qlistwidgetitem.html)**(tr("Oak"), listWidget);**

**new** [**QListWidgetItem**](qlistwidgetitem.html)**(tr("Fir"), listWidget);**

**new** [**QListWidgetItem**](qlistwidgetitem.html)**(tr("Pine"), listWidget);**

**2.**

[**QListWidgetItem**](qlistwidgetitem.html) **\*newItem = new** [**QListWidgetItem**](qlistwidgetitem.html)**;**

**newItem->setText(itemText);**

**listWidget->insertItem(row, newItem);**

**3. listWidget .addItem( );**

对于第二种方式：如果您需要插入一个新项目在一个特定的位置,那么它应该没有父部件构造。insertItem()函数应该被用来把它在列表中。小部件将所有权的项目列表。

[count](qlistwidget.html#count-prop)() ：得到列表项目数. 从列表中删除条目,使用 [takeItem](qlistwidget.html#takeItem)().

在列表中找当前项[currentItem](qlistwidget.html#currentItem)(), 改变当前项 [setCurrentItem](qlistwidget.html#setCurrentItem)().用户也可以通过导航键盘或点击不同的项目改变当前项. 当当前项发生变化时,currentItemChanged()信号发射。

## QListWidgetItem

QListWidgetItem类提供一个QListWidget项目的视图类。

QListWidgetItem代表一个QListWidget的单一项目。每个项目可以容纳几条信息,并适当地将显示它们。

列表项通常作用[text](file:///C:\Users\JG\Desktop\qlistwidgetitem.html#text)()显示文本，[icon](qlistwidgetitem.html#icon)()一个图标()。这些设置与setText()和setIcon()函数相关。文本和外观可以通过setFont()、[setForeground](file:///C:\Users\JG\Desktop\qlistwidgetitem.html#setForeground)(), and [setBackground](file:///C:\Users\JG\Desktop\qlistwidgetitem.html#setBackground)()设置。

文本列表项可以使用setTextAlignment ()函数保持一致 Tooltips, status tips and "What's This?" help can be added to list items with [setToolTip](qlistwidgetitem.html#setToolTip)(), [setStatusTip](qlistwidgetitem.html#setStatusTip)(), and [setWhatsThis](qlistwidgetitem.html#setWhatsThis)().

By default, items are enabled, selectable, checkable, and can be the source of drag and drop operations.

Each item's flags can be changed by calling [setFlags](qlistwidgetitem.html#setFlags)() with the appropriate value (see [Qt::ItemFlags](../qtcore/qt.html#ItemFlag-enum)). Checkable items can be checked, unchecked and partially checked with the [setCheckState](qlistwidgetitem.html#setCheckState)() function. The corresponding [checkState](qlistwidgetitem.html#checkState)() function indicates the item's current check state.

The [isHidden](qlistwidgetitem.html#isHidden)() function can be used to determine whether the item is hidden. To hide an item, use [setHidden](qlistwidgetitem.html#setHidden)().

## [QLocale](qlocale.html)

QLocale类表示在各种语言的数字和字符串之间的转换。

QLocale在它的构造函数中初始化语言-国家对，并提供与QSting中number-to-string 和 string-to-number转换函数类似的函数。

Example:

[**QLocale**](qlocale.html#QLocale) **egyptian(**[**QLocale**](qlocale.html#QLocale)**::Arabic,** [**QLocale**](qlocale.html#QLocale)**::Egypt);**

[**QString**](qstring.html) **s1 = egyptian.toString(1.571429E+07, 'e');**

[**QString**](qstring.html) **s2 = egyptian.toString(10);**

**double d = egyptian.toDouble(s1);**

**int i = egyptian.toInt(s2);**

### QSvgRenderer

QSvgRenderer类是用来绘制SVG文件的内容到绘图设备。使用QSvgRenderer,可缩放矢量图形(SVG)可以被呈现到任何QPaintDevice子类,包括QWidget、[QImage](file:///C:\Users\JG\qtgui\qimage.html)、QGLWidget。

QSvgRenderer提供一个API,它支持SVG渲染的基本特性,诸如加载和渲染的静态图纸,和交互更好的特点，如动画。场景渲染通过[QPainter](file:///C:\Users\JG\qtgui\qpainter.html)来完成，SVG绘图可以呈现在QPaintDevice的任何子类。

SVG绘图在QSvgRenderer构造时加载，或在构造后使用load()函数。

数据由序列化的XML直接提供，或间接使用文件名提供。如果一个有效的文件已经被加载，[isValid](qsvgrenderer.html#isValid)() returns true，否则返回false. QSvgRenderer提供的[render](file:///C:\Users\JG\Desktop\qsvgrenderer.html#render) ()槽通过指定的painter来呈现当前文件、或一个动画文件的当前帧。

defaultSize()函数提供了呈现当前加载的SVG文件所需的空间信息。它作用于绘图设备，如[QWidget](file:///C:\Users\JG\qtwidgets\qwidget.html)，经常需要parent layout提供一个大小提示，图的默认大小可能不同于它的可见区域,发现使用viewBox属性。

支持SVG动画图纸，可以用一个简单的控制功能和属性的集合。

* [animated](file:///C:\Users\JG\Desktop\qsvgrenderer.html#animated) ()函数显示一幅画是否包含动画信息.
* framesPerSecond属性包含动画播放的速度.

QSvgRenderer类提供了repaintNeeded()在渲染文件需要更新时发射。

### update()

**void** QPaintDeviceWindow::**update**(const [QRect](../qtcore/qrect.html) &re*ct*)

标记窗口的矩形区域将调试重绘。--将重绘这个区域

**void** QPaintDeviceWindow::**update**()

将重绘整个窗口。

## QDomElement

## enum Qt::ItemDataRole

模型中的每一项都有一组与之关联的数据元素，它们都有自己的角色。角色用来指示视图所显示的模型需要哪种类型的数据。自定义模型应该返回这些类型的数据。

常用的角色（与关联的类型）有：

|  |  |  |
| --- | --- | --- |
| **Constant 常量** | **Value** | **Description** 描述 |
| Qt::DisplayRole | 0 | 关键数据以文本的形式呈现. ([QString](qstring.html)) |
| Qt::DecorationRole | 1 | 要呈现的数据以一个图标的形式装饰. ([QColor](../qtgui/qcolor.html), [QIcon](../qtgui/qicon.html) or [QPixmap](../qtgui/qpixmap.html)) |
| Qt::EditRole | 2 | The data in a form suitable for editing in an editor. ([QString](qstring.html)) |
| Qt::ToolTipRole | 3 | The data displayed in the item's tooltip. ([QString](qstring.html)) |
| Qt::StatusTipRole | 4 | The data displayed in the status bar. ([QString](qstring.html)) |
| Qt::WhatsThisRole | 5 | The data displayed for the item in "What's This?" mode. ([QString](qstring.html)) |
| Qt::SizeHintRole | 13 | The size hint for the item that will be supplied to views. ([QSize](qsize.html)) |

角色描述外观和元数据(与相关的类型):

|  |  |  |
| --- | --- | --- |
| **Constant** | **Value** | **Description** |
| Qt::FontRole | 6 | The font used for items rendered with the default delegate. ([QFont](../qtgui/qfont.html)) |
| Qt::TextAlignmentRole | 7 | The alignment of the text for items rendered with the default delegate. ([Qt::AlignmentFlag](qt.html#AlignmentFlag-enum)) |
| Qt::BackgroundRole | 8 | The background brush used for items rendered with the default delegate. ([QBrush](../qtgui/qbrush.html)) |
| Qt::BackgroundColorRole | 8 | This role is obsolete. Use BackgroundRole instead. |
| Qt::ForegroundRole | 9 | The foreground brush (text color, typically) used for items rendered with the default delegate. ([QBrush](../qtgui/qbrush.html)) |
| Qt::TextColorRole | 9 | This role is obsolete. Use ForegroundRole instead. |
| Qt::CheckStateRole | 10 | This role is used to obtain the checked state of an item. ([Qt::CheckState](qt.html#CheckState-enum)) |
| Qt::InitialSortOrderRole | 14 | This role is used to obtain the initial sort order of a header view section. ([Qt::SortOrder](qt.html#SortOrder-enum)). This role was introduced in Qt 4.8. |

可访问性角色(与相关的类型):

|  |  |  |
| --- | --- | --- |
| **Constant** | **Value** | **Description** |
| Qt::AccessibleTextRole | 11 | The text to be used by accessibility extensions and plugins, such as screen readers. ([QString](qstring.html)) |
| Qt::AccessibleDescriptionRole | 12 | A description of the item for accessibility purposes. ([QString](qstring.html)) |

用户角色:

|  |  |  |
| --- | --- | --- |
| **Constant** | **Value** | **Description** |
| Qt::UserRole | 0x0100 | The first role that can be used for application-specific purposes. |

对用户角色，它是由开发人员决定使用哪个类型并且在访问和设置数据时确保使用正确的组件类型

## [QVariant](qvariant.html)

## QMainWindow

## 事件

accept()

### QMouseEvent

### QWheelEvent

## 信号与槽signals--slots

emit

## 多线程[QThread](qthread.html)

QThread类提供了一种独立于平台的方式来管理线程。一个QThread对象管理控制程序内的一个线程。

QThreads在 run()中开始执行。

默认，run()通过调用exec()开始事件循环并且在线程内运行Qt事件循环。

A [QThread](qthread.html) object manages one thread of control within the program. QThreads begin executing in [run](qthread.html#run)(). By default, [run](qthread.html#run)() starts the event loop by calling [exec](qthread.html#exec)() and runs a Qt event loop inside the thread.

可以通过工作对象调用 [QObject::moveToThread](qobject.html#moveToThread)()移除线程。

**class Worker : public** [**QObject**](qobject.html#QObject)

**{**

**Q\_OBJECT**

**public slots:**

**void doWork(const** [**QString**](qstring.html) **&parameter) {**

[**QString**](qstring.html) **result;**

**/\* ...** 这是费时的或阻塞操作**... \*/**

**emit resultReady(result);**

**}**

**signals:**

**void resultReady(const** [**QString**](qstring.html) **&result);**

**};**

**class Controller : public** [**QObject**](qobject.html#QObject)

**{**

**Q\_OBJECT**

[**QThread**](qthread.html#QThread) **workerThread;**

**public:**

**Controller() {**

**Worker \*worker = new Worker;**

**worker->moveToThread(&workerThread);**

**connect(&workerThread, &****[QThread](qthread.html" \l "QThread)::finished, worker, &**[**QObject**](qobject.html#QObject)**::deleteLater);**

**connect(this, &Controller::operate, worker, &Worker::doWork);**

**connect(worker, &Worker::resultReady, this, &Controller::handleResults);**

**workerThread.start();**

**}**

**~Controller() {**

**workerThread.quit();**

**workerThread.wait();**

**}**

**public slots:**

**void handleResults(const** [**QString**](qstring.html) **&);**

**signals:**

**void operate(const** [**QString**](qstring.html) **&);**

**};**

这段代码中Worker类的doWork槽内的代码将执行在一个单独的线程中。虽然， 你可以在任何线程中将任何对象的任何信号connect到它，但是，它在不同的线程仍是安全的signals and slots连接，因为队列连接机制（mechanism called [queued connections](qt.html#ConnectionType-enum)）。

另一种使代码运行在一个单独线程的方法是，在QThread的子类重写run()。

For example:

**class WorkerThread : public** [QThread](qthread.html#QThread)

**{**

**Q\_OBJECT**

**void run() Q\_DECL\_OVERRIDE {**

[QString](qstring.html) **result;**

**/\* ...** 这是费时的或阻塞操作**... \*/**

**emit resultReady(result);**

**}**

**signals:**

**void resultReady(const** [QString](qstring.html) **&s);**

**};**

**void MyObject::startWorkInAThread()**

**{**

**WorkerThread \*workerThread = new WorkerThread(this);**

**connect(workerThread, &WorkerThread::resultReady, this, &MyObject::handleResults);**

**connect(workerThread,&WorkerThread::finished,workerThread,&**[QObject](qobject.html#QObject)**::deleteLater);**

**workerThread->start();**

**}**

在这个例子中，这个线程退出之后运行函数将返回。这个运行的线程中没有任何事件循环，除非你调用[exec](file:///C:\Users\JG\Desktop\qthread.html#exec)()。

需要记住的是, QThread建议在旧线程中实例化它,而不是在新线程调用[run](file:///C:\Users\JG\Desktop\qthread.html#run) ()。这意味着所有QThread槽队列（queued slots）将执行在的旧线程中。因此、开发人员希望在新线程中调用槽必须使用worker-object的方法，新槽不应直接在QThread的子类中实现。

在子类化[QThread](file:///C:\Users\JG\Desktop\qthread.html)时，记住, 虽然[run](file:///C:\Users\JG\Desktop\qthread.html#run) ()垫在新线程中，但构造函数执行在旧的线程。如果来自两个不同的线程的两个函数都要访问成员变量, 那么要检查它是安全的。

（继承自QThread类实现的所谓线程类只是run函数里的部分在新线程执行，类的其他函数都是在旧线程执行的--QThread的子类是在旧线程中构造的）

Note: 必须注意被不同的线程影响的对象。

## [QObject](file:///C:\Users\JG\Desktop\qobject.html)

详细描述

[QObject](file:///C:\Users\JG\Desktop\qobject.html)是所有qt类的基类。

QObject是Qt的核心对象模型。这个模型的主要功能是一个非常强大的无缝对象间通信机制--称为信号与槽机制。你可以通过[connect](file:///C:\Users\JG\Desktop\qobject.html#connect) ()连接连接一个信号与槽，通过 [disconnect](file:///C:\Users\JG\Desktop\qobject.html#disconnect) ()断开信号与槽的连接。为了避免陷入永无止境的循环通知，你可以通过blockSignals()暂时阻止信号通过blockSignals()。protected 函数connectNotify()和disconnectNotify()能够跟踪connections。

QObjects在对象树组织它们自己。当您以另一个对象作为parent创建一个QObject时，这个对象将自动添加到parent对象的[children](file:///C:\Users\JG\Desktop\qobject.html#children)() list中。父对象的能够在其析构函数中自动删除它的孩子。你可以根据名称寻找对象的孩子，使用findChild()或findChildren()。

每一个对象都有一个[objectName](file:///C:\Users\JG\Desktop\qobject.html#objectName-prop) ()并且它的类名称可以通过相应的metaObject()返回的元对象来获取。

在QObject继承层次结构你可以使用[inherits](file:///C:\Users\JG\Desktop\qobject.html#inherits) ()函数来知道一个类是否继承另一个类。

当一个对象被删除时，它会发出[destroyed](file:///C:\Users\JG\Desktop\qobject.html#destroyed) ()信号。您可以捕获这个信号来避免qobject悬空的引用。

qobject可以通过[event](file:///C:\Users\JG\Desktop\qobject.html#event) ()接收事件并过滤其他对象的事件。

When an object is deleted, it emits a [destroyed](qobject.html#destroyed)() signal. You can catch this signal to avoid dangling references to QObjects. 详细说明查看 [installEventFilter](qobject.html#installEventFilter)() and [eventFilter](qobject.html#eventFilter)() 。一个便利的处理是：通过重写[childEvent](file:///C:\Users\JG\Desktop\qobject.html#childEvent)()来捕获其孩子的事件。

最后但同样重要的：QObject提供了Qt支持的基本定时器。QTimer。

注意：任何实现的信号与槽的对象必须有Q\_OBJECT宏。并且您还需要源文件上运行元对象编译器（[Meta Object Compiler](file:///C:\Users\JG\qtdoc\moc.html)）。我极力建议在所有的[QObject](file:///C:\Users\JG\Desktop\qobject.html)子类中使用[Q\_OBJECT](file:///C:\Users\JG\Desktop\qobject.html#Q_OBJECT)宏，不管它是否需要使用信号与槽。因为没有这样做的结果可能导致某些功能表现出奇怪的行为。

所有Qt widgets都继承自QObject。[isWidgetType](file:///C:\Users\JG\Desktop\qobject.html#isWidgetType)()函数返回一个对象是否是一个widget。它比qobject\_cast < QWidget \* >(obj)或obj - >[inherits](file:///C:\Users\JG\Desktop\qobject.html#inherits) (“QWidget”) 更快。一些QObject函数,比如[children](qobject.html#children)()返回一个QObjectList。QObjectList是typedef QList< QObject \* >。

### 线程关联性

据说QObject一每个实例都与一个线程关联, 或者它居住在一个特定的线程中。当一个QObject接收到一个信号队列或发布一个事件，它的槽或事件处理程序将运行的线程对象居住的线程中。

注意:如果一个QObject没有与线程关联(即,如果[thread](file:///C:\Users\JG\Desktop\qobject.html#thread) ()返回零), 或者它居住的线程没有运行事件循环，

那么它将不能接收信号队列或发布事件。默认情况下, QObject居住在创建它的线程中。一个对象可以使用[thread](file:///C:\Users\JG\Desktop\qobject.html#thread)()查询它关联的线程，并使用[moveToThread](file:///C:\Users\JG\Desktop\qobject.html#moveToThread)()改变它关联的线程。

所有qobject必须和他们的parent居住在同一个线程中（这里的parent不是指类的继承关系，而是指Qt部件之间的关系）。 因此:

**•**如果两个qobject生活在不同的线程则setParent()调用将会失败。

**•**当QObject搬到另一个线程**,** 它所有的孩子也会被自动移动**.**

**•**如果QObject 有parent 则moveToThread()调用将失败.

**•**如果在QThread::run()中创建一个QObjects对象，不能指定QThread对象为它的父类（如不能 mywidget = new mywidegt(this)这么创建），因为QThread并不住在调用QThread::run()的线程中，也是住在旧的线程中。

例：

class A\_Thread :public QThread

{

public:

      void run()

      {

               QWidget \*widget1 = new QWidget (this); //@1 ---error

  QWidget \*widget2 = new QWidget ; //@2 -- right

       }

}

我的理解：A\_Thread是QThread的子类，在对这个类进行实例化时其构造函数运行在旧的线程中，A\_Thread的对象住在住在实例化Thread的线程中，但Thread对象的run()函数却运行在一个新的线程中，

对于@1创建的widget1作为this的孩子，但他们分属两个线程—所以这是错误的（违背的所有qobject必须和他们的parent住在同一个线程的要求）。

对于@2 widget2

注意:QObject对象的成员变量不自动成为它的孩子。parent-child关联只能由以下两种方式建立。

1. 在要作为child的对象的构造函数中设置一个指向其父类的指针.
2. 在构造函数之后，通过调用setParent()设置其parent。

没有parent-child关联，当调用[moveToThread](file:///C:\Users\JG\Desktop\qobject.html#moveToThread)()时对象的成员变量仍将在旧的线程。

解释：

**class MyWidget : public** [QWidget](../qtwidgets/qwidget.html) **//**[QWidget](file:///C:\Users\JG\qtwidgets\qwidget.html) **是**QObject的子类

**{**

**Q\_OBJECT**

**public:**

**MyWidget();**

**private:**

[QPushButton](../qtwidgets/qpushbutton.html) **\*myButton1;**

[QPushButton](file:///C:\Users\JG\qtwidgets\qpushbutton.html) **\*myButton2;**

[QPushButton](C:\\Users\\JG\\qtwidgets\\qpushbutton.html) **\*myButton3;**

**};**

**MyWidget::MyWidget()**

**{**

**myButton1 = new** [QPushButton](../qtwidgets/qpushbutton.html)**(this); //在构造函数中指定this为它的**parent

**myButton2 = new** [QPushButton](../qtwidgets/qpushbutton.html)**; //在构造函数中没有指定**parent，但之后通过

**myButton2 ->**setParent(this); // setParent设置**this为它的**parent

**myButton3 = new** [QPushButton](../qtwidgets/qpushbutton.html)**; // myButton3不会自动成为this的了**child

**}**

# 问题

## 在Qt中使用中文的问题

使用带bom的UTF8文件格式,然后 在代码中添加

#pragma execution\_character\_set("utf-8")

## designer无法启动的问题

在使用VS+Qt 开发时，在VS是打开Qt 的.ui文件时没有弹出designer界面，而是出现一个代码页。

网上的解决方法如下：



但是网上的方法没有解决我的问题。

我的解决方法是以集成图形处理器方式打开。这个问题和我以前在Matlab上遇到的一样。将designer.exe以集成图形处理器的方式打开时，designer就可以正常启动了。但是在VS中还是不能打开.ui文件。这时我继续改VS的打开方式，最后成功解决。

## 按F1提示你没有相关文档

手册有时你按F1提示你没有相关文档．这时候你可以先按住Ctrl然后点击你要查找的那个方法．跳到定义处后，然后在按F1一般就回有了．或者你直接在帮助里面搜索也可以的．

## 关于Qt4widget内存分配和setObjectName作用的问题(父类会自动delete子类) parent-child内存管理机制

2个比较菜的问题：

1.由propdialog.ui文件生成的头文件ui\_propdialog.h中，可以找到为各个widget分配内存的代码，在函数setupUi中，例如layoutWidget = new QWidget(Dialog);

但是，找不到释放内存的代码，即没有delete，而这个properties dialog需要多次调用，会不会引起内存泄漏呢？是qt内部有自动释放的机制还是必须自己写代码delete掉？

2.layoutWidget->setObjectName(QString::fromUtf8("layoutWidget"));这一些语句去掉的话有什么不好的影响？我不需要根据ObjectName查找子控件。

只看该作者1楼 发表于: 2006-08-10

1, 如果指明parent后.当parent delete时,子部件会自动delete

2,应该没有影响吧.

只看该作者2楼 发表于: 2006-08-10

多谢释疑，但是关于第一点，我不是很理解内部的机制，这与以前编程的习惯不合。

如果说系统在parent析构的时候自动释放子控件，那么用户在析构函数中再使用delete不就会出现错误吗？实际上我这样做不会报错。

我做了一个实验，在某个对话框中new了一大堆控件而没有delete，然后多次调用该对话框，运行时用KSysGuard监视占用虚存大小，似乎没有什么变化。难道系统释放子控件的时候还检查一下是否已经释放过了？

只看该作者3楼 发表于: 2006-08-10

如果说系统在parent析构的时候自动释放子控件，那么用户在析构函数中再使用delete不就会出现错误吗？实际上我这样做不会报错。

//在parent未被delete时， 你可以delete子部件， 也可以不delete子部件。

但到最后parent 被delete 之前， 会检查parent的 reconstructor中有无delete子部件sentence，

若有，则执行delete子部件。若无， 编译系统也会自动给你加上delete子部件的sentence再执行。

给个例子你：

你写个class，可以写reconstructor function.也可以不写。 不写编译系统会自动给你加上。

## [Qt Designer中自定义控件的使用(提升法与插件法)](https://www.oschina.net/question/234345_52719)

准备乱写一点Qt自定义Widget在Designer中的使用。可是又不想重复提升法(promotion)及插件法基本用法，因为Manual中[Using Custom Widgets with Qt Designer](http://doc.qt.nokia.com/4.7/designer-using-custom-widgets.html)已经说的很清楚了。

### 使用designer

Qt用户可能会经历这样的3个阶段：

刚接触Qt，在designer中拖拖拽拽，一个界面就形成了。相当方便(不过不少网友初次接触时对layout相当不适应)

比较熟悉之后，感觉designer是个累赘，比如：

感觉在designer中拖拽控件，完全没有在代码中写的易于控制。尽管都是用QLayout

接触到自定义控件后，发现在designer中使用自定义控件比较麻烦。

就是不太喜欢desinger生成.ui文件，然后uic生成 ui\_xxx.h文件的这套东西。

...

再过一段时间，发觉，designer确实很方便。而且通过提升法在designer中使用自定义控件是非常简单的。而且前一阶段的问题其实也都不算什么问题。

### 自定义控件

考虑下列情况：

* 从QWidget派生了一个led灯控件：HLed
* 从QSpinBox派生了一个进制可变的：HBaseSpinBox
* 从QLabel派生了一个发射但双击信号的:HLabel
* ...

恩，没什么特别的哈

* 如果在代码中使用的话，包含头文件直接用即可。和标准控件没有任何区别。
* 如果在designer中使用，分别拖放QWidget、QSpinBox、QLabel，然后右键点击提升...

### 提升 vs 插件

两种方法有什么区别么？什么区别?

先考虑我们是如何使用designer的（此处不考虑.ui动态加载）：

* 打开designer，拖放控件，应用布局，设置属性
* 保存，生成 XXX.ui 文件
* 调用 uic 将 XXX.ui 预处理成 ui\_XXX.h
* 调用C++编译器编译这些 \*.h、\*.cpp

注意，如果你写了插件，那么它只在第一步中有用。有什么用：

1. 插件包含类名name(),头文件includeFile()信息。（对比，用提升法时，手动输入这两项信息）
2. 插件包含控件的实现
   * 所以在designer中可以看到控件的真实样子。(对比，提升法时，需要一个替身，一般就是其基类)
   * 可以感知其有哪些属性，并显示在属性编辑器中。（对比，你可以直接在属性编辑器中添加属性，又一个大大的加号，不是么？）
3. ...

归根到底，插件法 和 提升法想比，其实没有本质的区别。二者都是要生成一个.ui文件，而这个文件的使用，和插件一点关系都没有。

### 还有点什么...

前面的东西都似乎没什么意思，因为Manual中介绍很清楚，只要用过提升法和插件法就能得出上面这些的东西。可是，还有一点可能稍微有点意思，考虑：

我从QStackedWidget派生类了一个HDoubleSpinBox。可是在designer中，我想看到一个QDoubleSpinBox，而不是一个stacked widget.

我从QLabel派生了一个HLabelButton。可是在designer中，我想看到一个QToolButton，而不是一个 QLabel

...

插件法能不能做到？

能，只要让插件的createWidget()创建一个QDoubleSpinBox或者QToolButton

提升法能不能做到？

能，注意到提升时我们我们需要选择一个基类。而这个基类，我们可以随便选，选择哪一个，哪一个就会作为替身显示出来。

# Qt的一些宏和机制

## Q\_INIT\_RESOURCE

Q\_INIT\_RESOURCE是Qt的资源机制（resource mechanism)，它使程序在编译时将图片存储在.cpp文件中，运行时连接它。这要求你建立一个Qt资源文件\*\*\*.qrc，在\*\*\*.qrc中指定图片位置。编译时编译器将\*\*\*.qrc中指定的图片以二进制数的形式存储到Qt自动建立的名为qrc\_\*\*\*.cpp的文件中，这里的\*\*\*就是你建立\*\*\*.qrc时的名字，如果你的资源中没有这个资源文件，则没必要用（也不能用，因为找不到qrc\_\*\*\*这个文件）。

## Q\_PROPERTY

资源文件

qrc  qml svg

svg图片的信息可以通过浏览器的F12开发者选项来查看。

## [QPixmap](qpixmap.html)

详细描述

QPixmap类是一种离屏图像表示,可以用作绘制设备。

Qt 提供了4个类来处理图像数据：[QImage](file:///C:\Users\JG\Desktop\qimage.html), [QPixmap](file:///C:\Users\JG\Desktop\qpixmap.html), [QBitmap](file:///C:\Users\JG\Desktop\qbitmap.html) and [QPicture](file:///C:\Users\JG\Desktop\qpicture.html).

[QImage](file:///C:\Users\JG\Desktop\qimage.html)为I/O设计和优化，和直接访问和操纵像素，而QPixmap是为在屏幕上显示图像而进行设计和优化。QBitmap只是一个继承自QPixmap的深度1方便的类, isQBitmap()函数返回true,如果QPixmap对象是一个位图，否则返回假。最后，QPicture类是一个绘制设备用于记录和回放QPainter命令。

QPixmap可以很容易地显示在屏幕上通过使用QLabel或QAbstractButton的子类(如QPushButton和QToolButton)。QLabel象素映射属性,而QAbstractButton图标属性。

QPixmap对象可以通过值传递，因为QPixmap类使用隐式数据共享。QPixmap对象也可以流化。

注意, 象素映射中的像素数据是内置的并且由底层窗口系统管理。由于QPixmap类是 QPaintDevice的子类，QPainter可以直接用来绘制到pixmap上。

像素只能通过QPainter函数或通过转换QPixmap到一个[QImage](file:///C:\Users\JG\Desktop\qimage.html)。然而, [fill](qpixmap.html#fill) ()函数用于通过给定的颜色初始化整个象素映射。

有函数来实现[QImage](file:///C:\Users\JG\Desktop\qimage.html) and [QPixmap](file:///C:\Users\JG\Desktop\qpixmap.html) 两者间的转换。典型地，[QImage](file:///C:\Users\JG\Desktop\qimage.html)类用于加载一个图像文件,在 [QImage](file:///C:\Users\JG\Desktop\qimage.html)对象转换为QPixmap显示到屏幕上之前，它可以对图像数据进行一些操作。如果不需要进行操作的话,图像文件可以直接加载到QPixmap。

QPixmap提供了一组函数,可以用来获取各种各样的信息象图，此外,有几个函数, 能变换象素映射。

## **int** **qmlRegisterType**(const **char** \**uri*, **int** *versionMajor*, **int** *versionMinor*, const **char** \**qmlName*)

这个模板函数在QML系统中用qmlName名称注册一个c++类型，在库中导入来自的uri的版本由*versionMajor* 和 *versionMinor*组成（主版本、副版本）。

返回：QML类型的id。

有两种形式的这个模板函数:

template<typename T>

int qmlRegisterType(const char \*uri, int versionMajor, int versionMinor, const char \*qmlName);

template<typename T, int metaObjectRevision>

int qmlRegisterType(const char \*uri, int versionMajor, int versionMinor, const char \*qmlName);

前者是标准形式，注册一个类型为T的新类型。后者允许为注册的类修改为一个特定的版本。 (see [Type Revisions and Versions](qtqml-cppintegration-definetypes.html#type-revisions-and-versions)).

例如, 将c++的MySliderItem类注册为QML的Slider类，并对应"com.mycompany.qmlcomponents"命名空间的1.0版本。

**#include <QtQml>**

**...**

**qmlRegisterType<MySliderItem>("com.mycompany.qmlcomponents", 1, 0, "Slider");**

**// qmlRegisterType<C++类>(“注册成为QML类型后的命名空间”, 主版本号,副版本号, “注册为QML的类名”)**

一旦注册完成, 这个类型就能够通过在QML中通过导入指定类型的名称空间和版本号来使用。

**import com.mycompany.qmlcomponents 1.0**

**Slider {**

**// ...**

**}**

## The Property System 属性系统

Qt提供了一个复杂的类似与一些编译器供应商提供的属性系统。然而,它作为一个编译器且是一个独立与平台的库，Qt不依赖于非标准编译器特性（如\_\_property或[property] ）。Qt的解决方案对它所支持的平台下的任何标准c++编译器都适用。它是基于元对象系统的，因此还提供了通过信号与槽进行inter-object（对象间）通信。

声明属性的要求

通过使用[Q\_PROPERTY()](file:///C:\Users\JG\Desktop\qobject.html#Q_PROPERTY)宏在继承于[QObject](file:///C:\Users\JG\Desktop\qobject.html)的类中声明一个属性。

**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])**

下面是取自[QWidget](file:///C:\Users\JG\qtwidgets\qwidget.html)的一些属性声明的典型的例子。

**Q\_PROPERTY(bool focus READ hasFocus)**

**Q\_PROPERTY(bool enabled READ isEnabled WRITE setEnabled)**

**Q\_PROPERTY(**[**QCursor**](../qtgui/qcursor.html) **cursor READ cursor WRITE setCursor RESET unsetCursor)**

在下面这一个例子,演示如何使用Qt属性关键字导出成员变量。请注意, NOTIFY信号必须指定允许QML属性绑定。

**Q\_PROPERTY(**[**QColor**](../qtgui/qcolor.html) **color MEMBER m\_color NOTIFY colorChanged)**

**Q\_PROPERTY(**[**qreal**](qtglobal.html#qreal-typedef) **spacing MEMBER m\_spacing NOTIFY spacingChanged)**

**Q\_PROPERTY(**[**QString**](qstring.html) **text MEMBER m\_text NOTIFY textChanged)**

**...**

**signals:**

**void colorChanged();**

**void spacingChanged();**

**void textChanged(const** [**QString**](qstring.html) **&newText);**

**private:**

[**QColor**](../qtgui/qcolor.html) **m\_color;**

[**qreal**](qtglobal.html#qreal-typedef) **m\_spacing;**

[**QString**](qstring.html) **m\_text;**

一个属性的行为像类的数据成员，但它可以通过元对象系统（[Meta-Object System](file:///C:\Users\JG\Desktop\metaobjects.html)）附加功能。

* 如果没有指定MEMBER变量，则READ访问器函数是必需的。它是用于读取属性值。理想情况下,一个常量函数是用于此目的，并且它必须返回属性的类型或类型的常量引用。如[QWidget::focus](../qtwidgets/qwidget.html#focus-prop)是一个与只读属性,用READ 指定读取函数为QWidget:hasFocus()。
* 写入访问器函数是可选的，它用于设置属性值。它必须返回void, 有且仅有一个参数，且参数的类型为属性的类型或指针或引用。e.g., [QWidget::enabled](../qtwidgets/qwidget.html#enabled-prop)有一个WRITE函数[QWidget::setEnabled](file:///C:\Users\JG\qtwidgets\qwidget.html#enabled-prop)().只读属性不需要WRITE函数。e.g., [QWidget::focus](../qtwidgets/qwidget.html#focus-prop) 没有 WRITE 函数.
* MEMBER关联的变量在没有指定READ访问器函数时是必需的。这使得给定的成员变量可读和可写而不需要创建READ and WRITE访问器函数。如果你需要控制的变量的访问，对于其他没有使用MEMBER关联的变量仍然可以使用READ or WRITE访问器函数(但是READ or WRITE访问器与MEMBER不能同时使用)，
* RESET功能是可选的，它使属性返回到了上下文特定的默认值。如, [QWidget::cursor](file:///C:\Users\JG\qtwidgets\qwidget.html#cursor-prop)具有典型的READ and WRITE函数[QWidget::cursor](file:///C:\Users\JG\qtwidgets\qwidget.html#cursor-prop)() and [QWidget::setCursor](file:///C:\Users\JG\qtwidgets\qwidget.html#cursor-prop)()，并且它也有一个复位功能[QWidget::unsetCursor](file:///C:\Users\JG\qtwidgets\qwidget.html#cursor-prop)()，因为没有调用QWidget:setCursor()意味着重置到上下文特定的光标。重置函数必须返回void,不带任何参数。
* NOTIFY信号是可选的，如果定义,它应该指定一个现有类的信号，当的值属性变更时发出信号。

NOTIFY信号为MEMBER变量时必须有零个或一个参数，必须是相同类型的属性。参数将属性 的新值。NOTIFY信号只能在属性值真的发生改变时发出。为了避免绑定在QML不必要的重新评

估。例如:当需要一个MEMBER属性时Qt自动发出信号，因为它没有一个明确的setter。(for example. Qt emits automatically that signal when needed for MEMBER properties that do not have an explicit setter.)

* A REVISION number is optional. If included, it defines the property and its notifier signal to be used in a particular revision of the API (usually for exposure to QML). If not included, it defaults to 0.
* The DESIGNABLE attribute indicates whether the property should be visible in the property editor of GUI design tool (e.g., [Qt Designer](../qtdesigner/qtdesigner-manual.html)). Most properties are DESIGNABLE (default true). Instead of true or false, you can specify a boolean member function.
* The SCRIPTABLE attribute indicates whether this property should be accessible by a scripting engine (default true). Instead of true or false, you can specify a boolean member function.
* The STORED attribute indicates whether the property should be thought of as existing on its own or as depending on other values. It also indicates whether the property value must be saved when storing the object's state. Most properties are STORED (default true), but e.g., [QWidget::minimumWidth](../qtwidgets/qwidget.html#minimumWidth-prop)() has STORED false, because its value is just taken from the width component of property [QWidget::minimumSize](../qtwidgets/qwidget.html#minimumSize-prop)(), which is a [QSize](qsize.html).
* The USER attribute indicates whether the property is designated as the user-facing or user-editable property for the class. Normally, there is only one USER property per class (default false). e.g., [QAbstractButton::checked](../qtwidgets/qabstractbutton.html#checked-prop) is the user editable property for (checkable) buttons. Note that [QItemDelegate](../qtwidgets/qitemdelegate.html) gets and sets a widget's USER property.
* The presence of the CONSTANT attibute indicates that the property value is constant. For a given object instance, the READ method of a constant property must return the same value every time it is called. This constant value may be different for different instances of the object. A constant property cannot have a WRITE method or a NOTIFY signal.
* The presence of the FINAL attribute indicates that the property will not be overridden by a derived class. This can be used for performance optimizations in some cases, but is not enforced by moc. Care must be taken never to override a FINAL property.

The READ, WRITE, and RESET functions can be inherited. They can also be virtual. When they are inherited in classes where multiple inheritance is used, they must come from the first inherited class.

The property type can be any type supported by [QVariant](qvariant.html), or it can be a user-defined type. In this example, class [QDate](qdate.html) is considered to be a user-defined type.

**Q\_PROPERTY(**[**QDate**](qdate.html) **date READ getDate WRITE setDate)**

Because [QDate](qdate.html) is user-defined, you must include the <QDate> header file with the property declaration.

For historical reasons, *QMap* and *QList* as property types are synonym of *QVariantMap* and *QVariantList*.

Q\_SLOTS

Q\_SIGNALS

property与

## Qt中的数据库操作

1：Qt中如何连接到数据库

2：连接到数据库之后如何进行数据库操作，（增、删、改、查）

3：如何得到数据库返回的信息

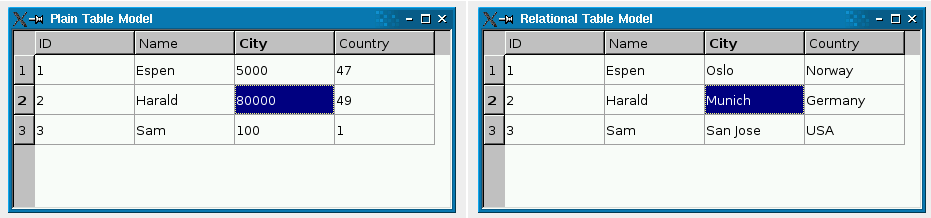
4：如何对数据库进行可视化操作。主/从视图如何应用。

5：当某个模型与数据库中的表关联后，修改模型的数据是否会对数据库的表有影响。

## QSqlRelationalTableModel

QSqlRelationalTableModel类为一个数据库表提供了一个可编辑的数据模型，并且它支持外键。

QSqlRelationalTableModel类似于QSqlTableModel,但QSqlRelationalTableModel允许设置外键列到其他数据库表。



The screenshot on the left shows a plain [QSqlTableModel](qsqltablemodel.html) in a [QTableView](../qtwidgets/qtableview.html). Foreign keys (city and country) aren't resolved to human-readable values. The screenshot on the right shows a [QSqlRelationalTableModel](qsqlrelationaltablemodel.html), with foreign keys resolved into human-readable text strings.

The following code snippet shows how the [QSqlRelationalTableModel](qsqlrelationaltablemodel.html) was set up:

**model->setTable("employee");**

**model->setRelation(2,** [**QSqlRelation**](qsqlrelation.html)**("city", "id", "name"));**

**model->setRelation(3,** [**QSqlRelation**](qsqlrelation.html)**("country", "id", "name"));**

The [setRelation](qsqlrelationaltablemodel.html#setRelation)() function calls establish a relationship between two tables. The first call specifies that column 2 in table employee is a foreign key that maps with field id of table city, and that the view should present the city's name field to the user. The second call does something similar with column 3.

If you use a read-write [QSqlRelationalTableModel](qsqlrelationaltablemodel.html), you probably want to use [QSqlRelationalDelegate](qsqlrelationaldelegate.html) on the view. Unlike the default delegate, [QSqlRelationalDelegate](qsqlrelationaldelegate.html) provides a combobox for fields that are foreign keys into other tables. To use the class, simply call [QAbstractItemView::setItemDelegate](../qtwidgets/qabstractitemview.html#setItemDelegate)() on the view with an instance of [QSqlRelationalDelegate](qsqlrelationaldelegate.html):

[**QTableView**](../qtwidgets/qtableview.html) **\*view = new** [**QTableView**](../qtwidgets/qtableview.html)**;**

**view->setModel(model);**

**view->setItemDelegate(new** [**QSqlRelationalDelegate**](qsqlrelationaldelegate.html)**(view));**

The [relationaltablemodel](qtsql-relationaltablemodel-example.html) example illustrates how to use [QSqlRelationalTableModel](qsqlrelationaltablemodel.html) in conjunction with [QSqlRelationalDelegate](qsqlrelationaldelegate.html) to provide tables with foreign key support.

Notes:

* The table must have a primary key declared.
* The table's primary key may not contain a relation to another table.
* If a relational table contains keys that refer to non-existent rows in the referenced table, the rows containing the invalid keys will not be exposed through the model. The user or the database is responsible for keeping referential integrity.
* If a relation's display column name is also used as a column name in the relational table, or if it is used as display column name in more than one relation it will be aliased. The alias is the relation's table name, display column name and a unique id joined by an underscore (e.g. tablename\_columnname\_id). [QSqlRecord::fieldName](qsqlrecord.html#fieldName)() will return the aliased column name. All occurrences of the duplicate display column name are aliased when duplication is detected, but no aliasing is done to the column names in the main table. The aliasing doesn't affect [QSqlRelation](qsqlrelation.html), so [QSqlRelation::displayColumn](qsqlrelation.html#displayColumn)() will return the original display column name.
* The reference table name is aliased. The alias is the word "relTblAl" and the relationed column index joined by an underscore (e.g. relTblAl\_2). The alias can be used to filter the table (For example, [setFilter](qsqltablemodel.html#setFilter)("relTblAl\_2='Oslo' OR relTblAl\_3='USA'")).
* When using [setData](qsqlrelationaltablemodel.html#setData)() the role should always be [Qt::EditRole](../qtcore/qt.html#ItemDataRole-enum), and when using [data](qsqlrelationaltablemodel.html#data)() the role should always be [Qt::DisplayRole](../qtcore/qt.html#ItemDataRole-enum).

**See also** [QSqlRelation](qsqlrelation.html), [QSqlRelationalDelegate](qsqlrelationaldelegate.html), and [Relational Table Model Example](qtsql-relationaltablemodel-example.html).

## QSqlTableModel

QSqlTableModel类提供了一个可为数据库表编辑的数据模型。QSqlTableModel是读写数据库表记录的高层接口。它是建立在低层SqlQuery类之上，并且可用于提供查看数据的QTableView类。

For example:

[**QSqlTableModel**](qsqltablemodel.html#QSqlTableModel) **\*model = new** [**QSqlTableModel**](qsqltablemodel.html#QSqlTableModel)**(parentObject, database);**

**model->setTable("employee");**

**model->setEditStrategy(**[**QSqlTableModel**](qsqltablemodel.html#QSqlTableModel)**::OnManualSubmit);**

**model->select();**

**model->setHeaderData(0,** [**Qt**](../qtcore/qt.html)**::Horizontal, tr("Name"));**

**model->setHeaderData(1,** [**Qt**](../qtcore/qt.html)**::Horizontal, tr("Salary"));**

[**QTableView**](../qtwidgets/qtableview.html) **\*view = new** [**QTableView**](../qtwidgets/qtableview.html)**;**

**view->setModel(model);**

**view->hideColumn(0); // don't show the ID**

**view->show();**

我们设置SQL表的名称和编辑策略, 然后我们设置在视图标题的标签显示. 编辑策略决定将用户在视图中所做的改变应用到实际的数据库中。可选的策略有：[OnFieldChange](qsqltablemodel.html#EditStrategy-enum), [OnRowChange](qsqltablemodel.html#EditStrategy-enum), and [OnManualSubmit](qsqltablemodel.html#EditStrategy-enum).

没有绑定到视图时，QSqlTableModel还可以用于以编程方式访问数据库：

[**QSqlTableModel**](qsqltablemodel.html#QSqlTableModel) **model;**

**model.setTable("employee");**

**model.select();**

**int salary = model.record(4).value("salary").toInt();**

上面的代码片段中通过SELECT 命令提取第4条记录的结果集中的工资字段。

可以使用setFilter()设置过滤器，或使用setSort()修改排序。最后,你必须调用select()将数据填充到模型。

tablemodel的例子说明了如何使用QSqlTableModel作为QTableView的数据来源。

QSqlTableModel没有提供直接支持外键。使用QSqlRelationalTableModel和QSqlRelationalDelegate。

[QSqlTableModel](qsqltablemodel.html) provides no direct support for foreign keys. 如果你决定使用外键那么你需要使用[QSqlRelationalTableModel](qsqlrelationaltablemodel.html) and [QSqlRelationalDelegate](qsqlrelationaldelegate.html).

**See also** [QSqlRelationalTableModel](qsqlrelationaltablemodel.html), [QSqlQuery](qsqlquery.html), [Model/View Programming](../qtwidgets/model-view-programming.html), [Table Model Example](qtsql-tablemodel-example.html), and [Cached Table Example](qtsql-cachedtable-example.html).

## QSqlRecord

QSqlRecord类封装了一条数据库记录(通常是数据库中的表或视图中的一行)。

QSqlRecord类封装了一条数据库记录的功能和特点。QSqlRecord支持添加和删除字段以及设置和检索字段值。

记录的字段的值可以使用[setValue](qsqlrecord.html#setValue)()通过名称或position设置；如果你想设置一个字段为null使用setNull()。

根据名称查找一个字段的position使用indexOf(),通过position找到字段的名称使用[fieldName](file:///C:\Users\JG\Desktop\qsqlrecord.html#fieldName) ()。使用[field](file:///C:\Users\JG\Desktop\qsqlrecord.html#field) ()来检索QSqlField对象的给定字段。使用[contains](file:///C:\Users\JG\Desktop\qsqlrecord.html#contains) ()查看记录包含的特定字段名。

When queries are generated to be executed on the database only those fields for which [isGenerated](qsqlrecord.html#isGenerated)() is true are included in the generated SQL.

在记录添加字段的函数[append](file:///C:\Users\JG\Desktop\qsqlrecord.html#append)() or [insert](file:///C:\Users\JG\Desktop\qsqlrecord.html#insert)(), 替换字段的函数[replace](file:///C:\Users\JG\Desktop\qsqlrecord.html#replace)(),移除字段的函数[remove](qsqlrecord.html#remove)()。移除所有字段使用[clear](file:///C:\Users\JG\Desktop\qsqlrecord.html#clear)()。[count](qsqlrecord.html#count)()返回记录中字段的数目。可以清除所有的值(设置为null)使用clearValues()。**See also** [QSqlField](qsqlfield.html) and [QSqlQuery::record](qsqlquery.html#record)().

## QModelIndex

## QDomDocument

## QDomNodeList

## QItemSelectionModel

QItemSelectionModel类跟踪一个视图的选择项。QItemSelectionModel跟踪一个模型中选定的项目。同一个模型可能一个视图或多个视图。它总是跟踪当前操作的视图中的选中项。

QItemSelectionModel是一个模型/视图类 并且它是Qt的模型/视图框架的一部分。

所选项的存储的使用范围，当你想修改选中的项时使用select() 并且提供参数[QItemSelection](file:///C:\Users\JG\Desktop\qitemselection.html)或 [QModelIndex](file:///C:\Users\JG\Desktop\qmodelindex.html) 和 a [QItemSelectionModel::SelectionFlag](file:///C:\Users\JG\Desktop\qitemselectionmodel.html#SelectionFlag-enum)。

[QItemSelectionModel](file:///C:\Users\JG\Desktop\qitemselectionmodel.html)需要一个两层的方法来选择管理，处理两个选择项委托和当前选择的项目的一部分。当前选中的项是当前交互式选择的一部分(例如用rubber-band选择或keyboard-shift选择)。

更新当前选中的项，使用按位或[QItemSelectionModel::Current](file:///C:\Users\JG\Desktop\qitemselectionmodel.html#SelectionFlag-enum)和其他SelectionFlags 。如果您省略QItemSelectionModel::Current命令，将创建一个新的当前选择，和前一个添加到整个选择。所有功能操作两层; 例如,设置selecteditem()从两个层将返回items。

**Note:** Since 5.5, [model](qitemselectionmodel.html#model), [hasSelection](qitemselectionmodel.html#hasSelection), and [currentIndex](qitemselectionmodel.html#currentIndex) are meta-object properties.

**See also** [Model/View Programming](../qtwidgets/model-view-programming.html), [QAbstractItemModel](qabstractitemmodel.html), and [Chart Example](../qtwidgets/qtwidgets-itemviews-chart-example.html).

## SQL

### [**QSqlDatabase**](file:///C:\Users\JG\Desktop\qsqldatabase.html#QSqlDatabase)

[static] [**QSqlDatabase**](qsqldatabase.html#QSqlDatabase) QSqlDatabase::**addDatabase**(const [**QString**](../qtcore/qstring.html) &*type*, const [**QString**](../qtcore/qstring.html) &*connectionName* = QLatin1String( defaultConnection ))

添加一个数据库到数据库连接列表，使用*type*驱动程序类型和connectionName连接的名字。如果已经存在一个叫做connectionName数据库连接，该连接将被删除。

数据库连接是指通过connectionName。返回新添加的数据库连接。如果类型不可用或无法加载,isValid()返回false。

如果没有指定connectionName，新连接就默认为应用程序连接，并且随后调用没有连接名参数的[database](file:///C:\Users\JG\Desktop\qsqldatabase.html#database) ()返回默认的连接。如果提供了connectionName，使用数据库(connectionName)来获取连接。

警告:如果您添加一个连接具有相同名称的现有连接, 新的连接将取代旧的连接。如果你多次调用没有指定connectionName的**addDatabase()**默认的连接将被取代。

在使用连接之前它必须被初始化。e.g., call some or all of [setDatabaseName](qsqldatabase.html#setDatabaseName)(), [setUserName](qsqldatabase.html#setUserName)(), [setPassword](qsqldatabase.html#setPassword)(), [setHostName](qsqldatabase.html#setHostName)(), [setPort](qsqldatabase.html#setPort)(), and [setConnectOptions](qsqldatabase.html#setConnectOptions)(), and, finally, [open](qsqldatabase.html#open)().

**Note:** This function is [thread-safe](../qtdoc/threads-reentrancy.html)

**See also** [database](qsqldatabase.html#database)(), [removeDatabase](qsqldatabase.html#removeDatabase)(), and [Threads and the SQL Module](../qtdoc/threads-modules.html#threads-and-the-sql-module).

//连接数据库

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");

db.setDatabaseName(":memory:");

if (!db.open())

return db.lastError();

//查看数据库中是否存在"books"和"authors"表

QStringList tables = db.tables();

if (tables.contains("books", Qt::CaseInsensitive)

&& tables.contains("authors", Qt::CaseInsensitive))

return QSqlError();

// QSqlQuery对象用来执行和操作的SQL语句

QSqlQuery q;

//建表

if (!q.exec(QLatin1String("create table books(id integer primary key, title varchar, author integer, genre integer, year integer, rating integer)")))

return q.lastError();

//为在表中插入记录作准备

if (!q.prepare(QLatin1String("insert into books(title, year, author, genre, rating) values(?, ?, ?, ?, ?)")))

return q.lastError();

q.addBindValue(title);

q.addBindValue(year);

q.addBindValue(authorId);

q.addBindValue(genreId);

q.addBindValue(rating);

q.exec(); //在表中插入记录

// 创建数据模型

QSqlRelationalTableModel \*model;

model = new QSqlRelationalTableModel(parent);

model->setEditStrategy(QSqlTableModel::OnManualSubmit);

model->setTable("books");

// 记录列的索引

int authorIdx = model->fieldIndex("author");

int genreIdx = model->fieldIndex("genre");

// 设置关系到另一个数据库表 （"authors"和"genres"是数据库中的另外两个表）

model->setRelation(authorIdx, QSqlRelation("authors", "id", "name"));

model->setRelation(genreIdx, QSqlRelation("genres", "id", "name"));

// 设置本地化标题标题

model->setHeaderData(authorIdx, Qt::Horizontal, tr("Author Name"));

model->setHeaderData(genreIdx, Qt::Horizontal, tr("Genre"));

model->setHeaderData(model->fieldIndex("title"), Qt::Horizontal, tr("Title"));

model->setHeaderData(model->fieldIndex("year"), Qt::Horizontal, tr("Year"));

model->setHeaderData(model->fieldIndex("rating"), Qt::Horizontal, tr("Rating"));

// 填充模型

if (!model->select()) {

showError(model->lastError());

return;

}

// 设置模型和隐藏ID列

ui.bookTable->*setModel*(model); //为view设置模型

ui.bookTable->setItemDelegate(new BookDelegate(ui.bookTable));

ui.bookTable->setColumnHidden(model->fieldIndex("id"), true);

ui.bookTable->setSelectionMode(QAbstractItemView::SingleSelection);

问题：在layout中显示的文本重叠在一起了（代码如下：）

g\_layout = new QGridLayout(this);

v\_layout->addLayout(g\_layout);

QLabel \*disk\_mode\_static = new QLabel("mode:", this);

g\_layout->addWidget(disk\_mode\_static,0,0,Qt::AlignRight);

QLabel \*disk\_size\_static = new QLabel("size:", this);

g\_layout->addWidget(disk\_size\_static,1,0,Qt::AlignRight);

QLabel \*part\_table\_type\_static = new QLabel("type", this);

g\_layout->addWidget(part\_table\_type\_static,2,0,Qt::AlignRight);

原因：

我的这个类继承于QMainWindow,这个类有自己默认的布局管理器,所以我这个布局管理就无效,需要进行指定。

QpushButton设置当鼠标移到其上时的提示

鼠标移到按钮的区域时显示提示信息，可用setToolTip（）函数