Qt 对象模型

标准的 C++对象模型为对象范例提供了十分有效的运行时刻支持。但是这种 C++对象模型的静态性质在一定的问题领域是不够灵活的。图形用户界面编程就是一个同时需要运行时刻的效率和高水平的灵活性的领域。Qt 通过结合 C++的速度为这一领域提供了 Qt 对象模型的灵活性。

Qt 把下面这些特性添加到了 C++当中：

- 一种关于无缝对象通讯被称为信号和槽的非常强大的机制，
- 可查询和可设计的属性，
- 强大的事件和事件过滤器，
- 根据上下文进行国际化的字符串翻译，
- 完善的时间间隔驱动的计时器使得在一个事件驱动的图形界面程序中很好地集成许多任务成为可能。
- 以一种自然的方式组织对象所有权的分层次和可查询的对象树。
- 被守护的指针，QGuardedPtr，当参考对象被破坏时，可以自动地设置为无效，不像正常的C++指针在它们的对象被破坏的时候变成了“摇摆指针”。

许多Qt的特性是基于QObject的继承，通过标准C++技术实现的。其他的，比如对象通讯机制和虚拟属性系统，都需要Qt自己的元对象编译器(moc) 提供的元对象系统。

元对象系统是一种可以使语言更加适用于真正的组件图形用户界面程序的C++扩展。尽管模板也可以用来扩展C++，元对象系统提供给标准C++而模板所不能提供的益处，请看为什么Qt不用模板来实现信号和槽？ 。

对象树和对象所有权

QObject在对象树中组织它们自己。当你以另外一个对象作为父对象来创建一个QObject时，它就被添加到父对象的children() 列表中，并且当父对象被删除的时候，它也会被删除。这种机制很好的适合了图形用户界面应用对象的需要。例如，一个QAccel(键盘快捷键)是相关窗口的子对象，当用户关闭该窗口的时候，这个快捷键也被删除了。

静态函数QObject::objectTrees()提供了访问当前存在的所有跟对象的方法。

QWidget，在屏幕上显示的任何东西的基类，扩展着父-子对象关系。一个子对象通常就是一个子窗口部件，也就是说，它被显示在父对象的坐标系统中并且在图象上由父对象的边界夹住。例如，当一个应用程序在一个消息框被关闭之后删除这个消息框时，消息框的按钮和标签正如我们所想要的也被删除了，因为这些按钮和标签都是消息框的子对象。

你也可以自己删除子对象，这样它们就会把它们自己从它们的父对象中移除。例如，当用户移除工具条可以导致应用程序删除它的一个QToolBar对象，在这种情况工具条的QMainWindow父对象会检测到这种变化并因此而重新构成屏幕空间。

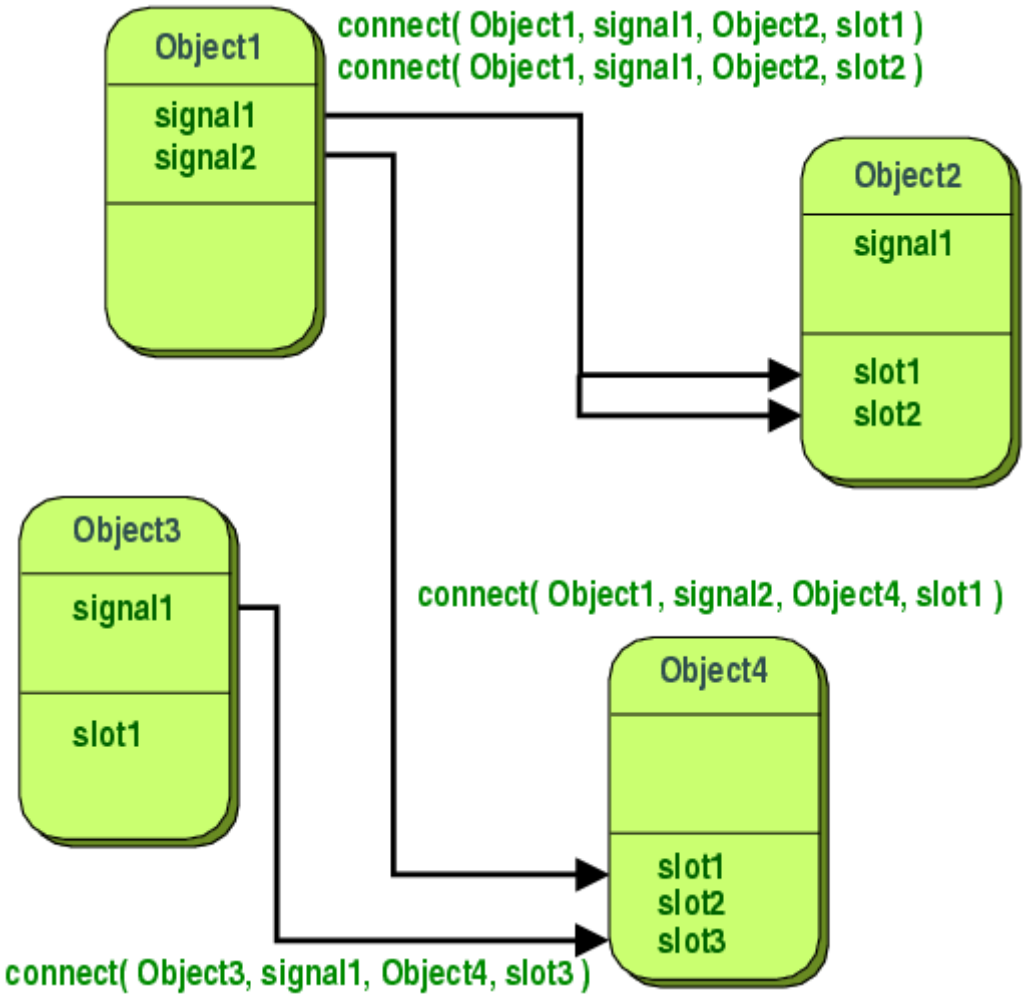
当一个应用程序的看起来或者运行起来有些奇怪的时候，调试函数 QObject::dumpObjectTree() 和QObject::dumpObjectInfo()是经常有用的。

信号和槽

信号和槽用于对象间的通讯。信号/槽机制是 Qt 的一个中心特征并且也许是 Qt 与其它工具包的最不相同的部分。

在图形用户界面编程中，我们经常希望一个窗口部件的一个变化被通知给另一个窗口部件。更一般地，我们希望任何一类的对象可以和其它对象进行通讯。例如，如果我们正在解析一个 XML 文件，当我们遇到一个新的标签时，我们也许希望通知列表视图我们正在用来表达 XML 文件的结构。

较老的工具包使用一种被称作回调的通讯方式来实现同一目的。回调是指一个函数的指针，所以如果你希望一个处理函数通知你一些事件，你可以把另一个函数（回调）的指针传递给处理函数。处理函数在适当的时候调用回调。回调有两个主要缺点。首先他们不是类型安全的。我们从来都不能确定处理函数使用了正确的参数来调用回调。其次回调和处理函数是非常强有力地联系在一起的，因为处理函数必须知道要调用哪个回调。

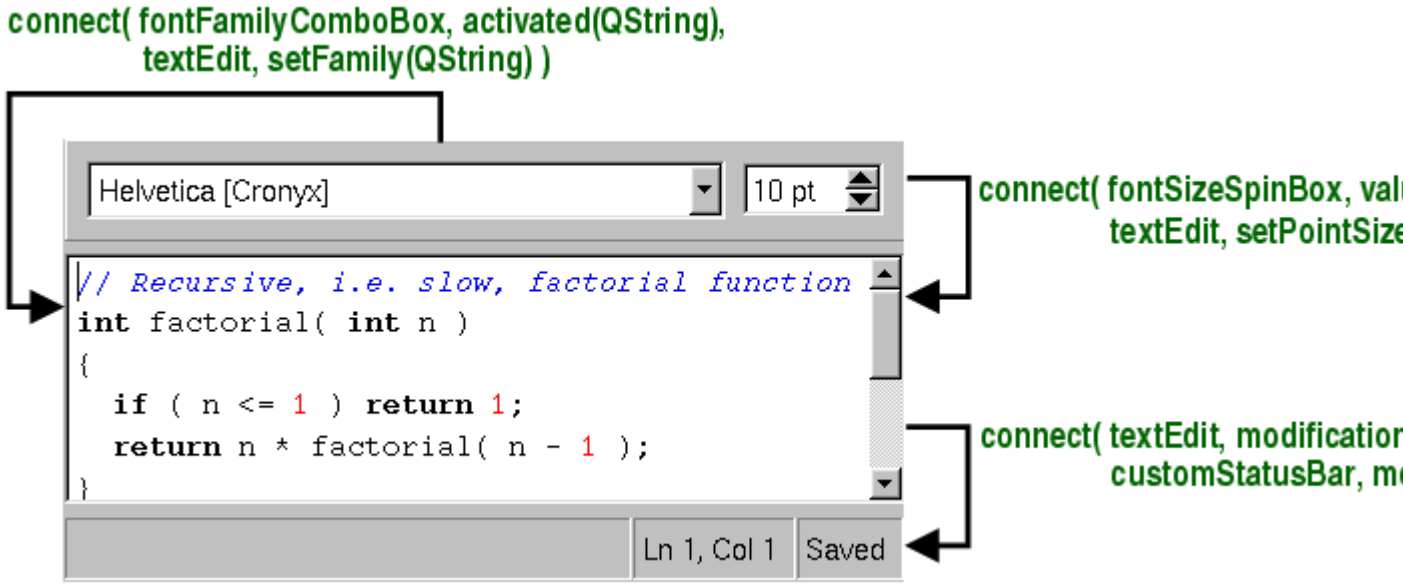


一个关于一些信号和槽连接的摘要图

在 Qt 中我们有一种可以替代回调的技术。我们使用信号和槽。当一个特定事件发生的时候，一个信号被发射。Qt 的窗口部件有很多预定义的信号，但是我们总是可以通过继承来加入我们自己的信号。槽就是一个可以被调用处理特定信号的函数。Qt 的窗口部件又很多预定义的槽，但是通常的习惯是你可以加入自己的槽，这样你就可以处理你所感兴趣的信号。

信号和槽的机制是类型安全的：一个信号的签名必须与它的接收槽的签名相匹配。（实际上一个槽的签名可以比它接收的信号的签名少，因为它可以忽略额外的签名。）因为签名是一致的，编译器就可以帮助我们检测类型不匹配。信号和槽是宽松地联系在一起的：一个发射信号的类不用知道也不用注意哪个槽要接收这个信号。Qt 的信号和槽的机制可以保证如果你把一个信号和一个槽连接起来，槽会在正确的时间使用信号的参数而被调用。信号和槽可以使用任何数量、任何类型的参数。它们是完全类型安全的：不会再有回调核心转储(core dump)。

从QObject类或者它的一个子类（比如QWidget类）继承的所有类可以包含信号和槽。当对象改变它们的状态的时候，信号被发送，从某种意义上讲，它们也许对外面的世界感兴趣。这就是所有的对象通讯时所做的一切。它不知道也不注意无论有没有东西接收它所发射的信号。这就是真正的信息封装，并且确保对象可以用作一个软件组件。



一个信号和槽连接的例子

槽可以用来接收信号，但它们是正常的成员函数。一个槽不知道它是否被任意信号连接。此外，对象不知道关于这种通讯机制和能够被用作一个真正的软件组件。

你可以把许多信号和你所希望的单一槽相连，并且一个信号也可以和你所期望的许多槽相连。把一个信号和另一个信号直接相连也是可以的。（这时，只要第一个信号被发射时，第二个信号立刻就被发射。）

总体来看，信号和槽构成了一个强有力的组件编程机制。

一个小例子

一个最小的 C++类声明如下：

```
class Foo
{
public:
    Foo();
    int value() const { return val; }
    void setValue( int );
private:
    int val;
};
```

一个小的 Qt 类如下：

```
class Foo : public QObject
{
    Q_OBJECT
public:
    Foo();
    int value() const { return val; }
public slots:
    void setValue( int );
signals:
    void valueChanged( int );
private:
    int val;
};
```

这个类有同样的内部状态，和公有方法来访问状态，但是另外它也支持使用信号和槽的组件编程：这个类可以通过发射一个信号，valueChanged()，来告诉外面的世界它的状态发生了变化，并且它有一个槽，其它对象可以发送信号给这个槽。

所有包含信号和/或者槽的类必须在它们的声明中提到 Q_OBJECT。

槽可以由应用程序的编写者来实现。这里是 Foo::setValue()的一个可能的实现：

```
void Foo::setValue( int v )
{
    if ( v != val ) {
        val = v;
        emit valueChanged(v);
    }
}
```

emit valueChanged(v)这一行从对象中发射valueChanged信号。正如你所能看到的，你通过使用emit signal(arguments)来发射信号。

下面是把两个对象连接在一起的一种方法：

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
b.setValue( 11 ); // a == undefined  b == 11
a.setValue( 79 ); // a == 79         b == 79
b.value();
```

调用a.setValue(79)会使a发射一个valueChanged() 信号，b将会在它的setValue()槽中接收这个信号，也就是b.setValue(79) 被调用。接下来b会发射同样的valueChanged()信号，但是因为没有槽被连接到b的valueChanged()信号，所以没有发生任何事（信号消失了）。

注意只有当v != val的时候setValue()函数才会设置这个值并且发射信号。这样就避免了在循环连接的情况下（比如b.valueChanged() 和a.setValue()连接在一起）出现无休止的循环的情况。

这个例子说明了对象之间可以在互相不知道的情况下一起工作，只要在最初的时在它们中间建立连接。

预处理程序改变或者移除了signals、slots和emit 这些关键字，这样就可以使用标准的C++编译器。

在一个定义有信号和槽的类上运行moc。这样就会生成一个可以和其它对象文件编译和连接成引用程序的C++源文件。

信号

当对象的内部状态发生改变，信号就被发射，在某些方面对于对象代理或者所有者也许是很有趣的。只有定义了一个信号的类和它的子类才能发射这个信号。

例如，一个列表框同时发射highlighted()和activated()这两个信号。绝大多数对象也许只对activated()这个信号感兴趣，但是有时想知道列表框中的哪个条目在当前是高亮的。如果两个不同的类对同一个信号感兴趣，你可以把这个信号和这两个对象连接起来。

当一个信号被发射，它所连接的槽会被立即执行，就像一个普通函数调用一样。信号/槽机制完全不依赖于任何一种图形用户界面的事件回路。当所有的槽都返回后 emit也将返回。

如果几个槽被连接到一个信号，当信号被发射时，这些槽就会被按任意顺序一个接一个地执行。

信号会由moc自动生成并且一定不要在.cpp文件中实现。它们也不能有任何返回类型（比如使用void）。

关于参数需要注意。我们的经验显示如果信号和槽*不*使用特殊的类型，它们都可以多次使用。如果QScrollBar::valueChanged() 使用了一个特殊的类型，比如hypothetical QRangeControl::Range，它就只能被连接到被设计成可以处理QRangeControl的槽。简单的和教程 1 的第 5 部分一样的程序将是不可能的。

槽

当一个和槽连接的信号被发射的时候，这个操被调用。槽也是普通的 C++函数并且可以像它们一样被调用；它们唯一的特点就是它们可以被信号连接。槽的参数不能含有默认值，并且和信号一样，为了槽的参数而使用自己特定的类型是很不明智的。

因为槽就是普通成员函数，但却有一点非常有意思的东西，它们也和普通成员函数一样有访问权限。一个槽的访问权限决定了谁可以和它相连：

一个public slots:区包含了任何信号都可以相连的槽。这对于组件编程来说非常有用：你生成了许多对象，它们互相并不知道，把它们的信号和槽连接起来，这样信息就可以正确地传递，并且就像一个铁路模型，把它打开然后让它跑起来。

一个protected slots:区包含了之后这个类和它的子类的信号才能连接的槽。这就是说这些槽只是类的实现的一部分，而不是它和外界的接口。

一个private slots:区包含了之后这个类本身的信号可以连接的槽。这就是说它和这个类是非常紧密的，甚至它的子类都没有获得连接权利这样的信任。

你也可以把槽定义为虚的，这在实践中被发现也是非常有用的。

信号和槽的机制是非常有效的，但是它不像“真正的”回调那样快。信号和槽稍微有些慢，这是因为它们所提供的灵活性，尽管在实际应用中这些不同可以被忽略。通常，发射一个和槽相连的信号，大约只比直接调用那些非虚函数调用的接收器慢十倍。这是定位连接对象所需的开销，可以安全地重复所有地连接（例如在发射期间检查并发接收器是否被破坏）并且可以按一般的方式安排任何参数。当十个非虚函数调用听起来很多时，举个例子来说，时间开销只不过比任何一个“new”或者 “delete”操作要少些。当你执行一个字符串、矢量或者列表操作时，需要“new”或者 “delete”，信号和槽仅仅对一个完整函数调用地时间开销中的一个非常小的部分负责。无论何时你在一个槽中使用一个系统调用和间接地调用超过十个函数的时间是相同的。在一台 i585-500 机器上，你每秒钟可以发射 2，000，000 个左右连接到一个接收器上的信号，或者发射 1，200，000 个左右连接到两个接收器的信号。信号和槽机制的简单性和灵活性对于时间的开销来说是非常值得的，你的用户甚至察觉不出来。

元对象信息

元对象编译器（moc）解析一个C++文件中的类声明并且生成初始化元对象的C++代码。元对象包括所有信号和槽函数的名称，还有这些函数的指针。（要获得更多的信息，请看为什么Qt不用模板来实现信号和槽？）

元对象包括一些额外的信息，比如对象的类名称。你也可以检查一个对象是否继承了一个特定的类，比如：

```
if ( widget->inherits("QButton") ) {
      // 是的，它是一个 Push Button、Radio Button 或者其它按钮。
}
```

一个真实的例子

这是一个注释过的简单的例子（代码片断选自qlcdnumber.h）。

```
#include "qframe.h"
#include "qbitarray.h"

class QLCDNumber : public QFrame
```

QLCDNumber通过QFrame和QWidget，还有#include这样的相关声明继承了含有绝大多数信号/槽知识的QObject。

```
{
    Q_OBJECT
```

Q_OBJECT是由预处理器展开声明几个由moc来实现的成员函数，如果你得到了几行“virtual function QButton::className not defined”这样的编译器错误信息，你也许忘记运行moc或者忘记在连接命令中包含moc输出。

```
public:
    QLCDNumber( QWidget *parent=0, const char *name=0 );
    QLCDNumber( uint numDigits, QWidget *parent=0, const char *name=0 );
```

它并不和moc直接相关，但是如果你继承了QWidget，你当然想在你的构造器中获得*parent*和*name*这两个参数，而且把它们传递到父类的构造器中。

一些解析器和成员函数在这里省略掉了，moc 忽略了这些成员函数。

```
signals:
    void    overflow();
```

当QLCDNumber被请求显示一个不可能值时，它发射一个信号。

如果你没有留意溢出，或者你认为溢出不会发生，你可以忽略 overflow()信号，也就是说你可以不把它连接到任何一个槽上。

另一方面如果当数字溢出时，你想调用两个不同的错误函数，很简单地你可以把这个信号和两个不同的槽连接起来。Qt 将会两个都调用（按任意顺序）。

```
public slots:
    void    display( int num );
    void    display( double num );
    void    display( const char *str );
    void    setHexMode();
    void    setDecMode();
    void    setOctMode();
    void    setBinMode();
    void    smallDecimalPoint( bool );
```

一个槽就是一个接收函数，用来获得其它窗口部件状态变或的信息。QLCDNumber 使用它，就像上面的代码一样，来设置显示的数字。因为display()是这个类和程序的其它的部分的一个接口，所以这个槽是公有的。

几个例程把QScrollBar的newValue信号连接到display槽，所以LCD数字可以继续显示滚动条的值。

请注意 display()被重载了，当你把一个信号和这个槽相连的时候，Qt 将会选择适当的版本。如果使用回调，你会发现五个不同的名字并且自己来跟踪类型。

一些不相关的成员函数已经从例子中省略了。

```
};
```

元对象系统

Qt 中的元对象系统是用来处理对象间通讯的信号/槽机制、运行时的类型信息和动态属性系统。

它基于下列三类：

1. QObject类；
2. 类声明中的私有段中的 Q_OBJECT 宏；
3. 元对象编译器（moc）。

*moc*读取C++源文件。如果它发现其中包含一个或多个类的声明中含有Q_OBJECT宏，它就会给含有Q_OBJECT宏的类生成另一个含有元对象代码的C++源文件。这个生成的源文件可以被类的源文件包含（#include）到或者和这个类的实现一起编译和连接。

除了提供对象间通讯的信号和槽机制之外（介绍这个系统的主要原因），QObject中的元对象代码实现其它特征：

- className()函数在运行的时候以字符串返回类的名称，不需要C++编译器中的本地运行类型信息（RTTI）的支持。
- inherits()函数返回这个对象是否是一个继承于QObject继承树中一个特定类的类的实例。
- tr()和trUtf8() 两个函数是用于国际化中的字符串翻译。
- setProperty()和property()两个函数是用来通过名称动态设置和获得对象属性的。
- metaObject()函数返回这个类所关联的元对象。

虽然你使用QObject作为一个基类而不使用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，那么这里的信号和槽以及其它特征描述都不会被提供。根据元对象系统的观点，一个没有元代码的QObject的子类和它含有元对象代码的最近的祖先相同。举例来说就是，className()将不会返回你的类的实际名称，返回的是它的这个祖先的名称。我们*强烈*建议QObject 的所有子类使用Q_OBJECT宏，而不管它们是否实际使用了信号、槽和属性。

属性

Qt提供了一套和一些编译器提供商也提供的属性系统类似的完善的属性系统。然而，作为一个不依赖编译器和平台的库，Qt不能依赖像__property或者[property]那样的非标准编译器特征。我们的解决方案可以在我们支持的每一个平台上和*任何*标准的C++编译器一起工作。它基于元对象系统，元对象系统也通过信号和槽提供对象通讯。

在类声明中的Q_PROPERTY宏声明了一个属性。属性只能在继承于QObject的子类中声明。第二个宏，Q_OVERRIDE，可以用来覆盖一些子类中由继承得到的属性。

对于外面的世界，属性看起来和一个数据成员非常类似。然而，属性和普通的数据成员还是有一下一些不同点：

- 一个读函数。这是一直存在的。
- 一个写函数。这个是可选的：像QWidget::isDesktop()这样的只读的属性就没有写函数。
- “存储”特征需要说明持续性。绝大多数属性是被存储的，但是有一小部分的虚拟属性却不用。举个例子，QWidget::minimumWidth()是不用存储的，因为它只是QWidget::minimumSize()的一种查看，没有自己的数据。
- 一个复位函数用来把属性设置回它根据上下文的特定缺省值。这个用法还是比较罕见的，但是举个例子，QWidget::font()需要这个函数，因为没有调用QWidget::setFont()意味着“复位到根据上下文特定的字体”。
- “可设计”特征说明它是否可以被一个图形用户界面生成器（例如Qt设计器）设置属性。对于大多数属性都有这个特征，但不是所有，例如QButton::isDown()。用户可以按按钮，并且应用程序设计人员可以让程序来按它自己的按钮，但是一个图形用户界面设计工具不能按按钮。

读、写和复位函数就像任何成员函数一样，继承或不继承，虚或不虚。只有一个例外就是，在多重继承的情况下，成员函数必须从第一个被继承类继承。

属性可以在不知道被使用的类的任何情况的时候通过QObject中的一般函数进行读写。下面两个函数调用是等效的：

```
// QButton *b 和 QObject *o 指向同一个按钮
b->setDown( TRUE );
o->setProperty( "down", TRUE );
```

等效的是指，除了第一个函数要快一些，在编译的时候提供了更好的诊断信息。在实际应用中，第一个函数更好些。然而，因为我们可以通过QMetaObject获得任何一个QObject的所有有用属性的一个列表，QObject::setProperty()可以让你控制类中那些在编译时不可用的属性。

像QObject::setProperty()一样，还有一个相应的QObject::property()函数。QMetaObject::propertyNames()返回所有可用属性的名称。QMetaObject::property()返回一个指定属性的属性数据：一个QMetaProperty对象。

这里有一个简单的例子说明了可以应用的绝大多数重要属性函数：

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

这个类有一个名为“priority”的还不被元对象系统所知的属性。为了让这个属性被元对象系统知道，你必须用Q_PROPERTY宏来声明它。声明语法如下：

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
            [RESET resetFunction] [DESIGNABLE bool]
            [SCRIPTABLE bool] [STORED bool] )
```

为了声明是有效的，读函数必须是常量函数并且返回值的类型是它本身或者是指向它的指针，或者是它的一个引用。可选的写函数必须返回 void 并且必须带有一个正确的参数，类型必须是它本身或者是指向它的指针，或者是它的一个常量引用。元对象编译器强迫这样的。

属性的类型可以是任何一个QVariant支持的类型或者是一个自己在类中已经定义的枚举类型。因为MyClass中的属性使用了枚举类型Priority，这个类型必须也向属性系统注册。这样的话，像如下方式通过名称来设置值是可行的：

```
obj->setProperty( "priority", "VeryHigh" );
```

枚举类型必须使用 Q_ENUMS 宏来进行注册。这里是一个包含属性相关声明的最终的类声明：

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

另外一个类似的宏是 Q_SETS。像 Q_ENUMS 一样，它注册了一个枚举类型，但是它额外的加了一个“set”的标记，也就是说，这个枚举数据可以被一起读或写。一个输入输出类也许有枚举数据“读”和“写”和接收“读|写”：这时最好用 Q_SETS 来声明一个枚举类型，而不是 Q_ENUMS。

Q_PROPERTY 段剩余的关键字是 RESET、DESIGNABLE、SCRIPTABLE 和 STORED。

RESET 指定一个函数可以设置属性到缺省状态（这个缺省状态可能和初始状态不同）。这个函数必须返回 void 并且不带有参数。

DESIGNABLE 声明这个属性是否适合被一个图形用户界名设计工具修改。缺省的 TRUE 是说这个属性可写，否则就是 FALSE 说明不能。你可以定义一个布尔成员函数来替代 TRUE 或 FALSE。

SCRIPTABLE 声明这个属性是否适合被一个脚本引擎访问。缺省是 TRUE，可以。你可以定义一个布尔成员函数来替代 TRUE 或 FALSE。

STORED声明这个属性的值是否必须作为一个存储的对象状态而被记得。STORED只对可写的属性有意义。缺省是TRUE。技术上多余的属性（比如，如果QRect的geometry已经是一个属性了的QPoint的pos）定义为FALSE。

连接到属性系统是一个附加宏，“Q_CLASSINFO”，它可以用来把名称/值这样一套的属性添加到一个类的元对象中，例如：

```
Q_CLASSINFO( "Version", "3.0.0" )
```

和其它元数据一样，类信息在运行时是可以通过元对象访问的，具体请看 QMetaObject::classInfo()。

使用元对象编译器

元对象编译器，朋友中的moc，是处理Qt的C++扩展的程序。

元对象编译器读取一个C++源文件。如果它发现其中的一个或多个类的声明中含有Q_OBJECT宏，它就会给这个使用Q_OBJECT宏的类生成另外一个包含元对象代码的C++源文件。尤其是，元对象代码对信号/槽机制、运行时类型信息和动态属性系统是需要的。

一个被元对象编译器生成的 C++源文件必须和这个类的实现一起被编译和连接（或者它被包含到（#include）这个类的源文件中）。

如果你是用qmake来生成你的Makefile文件，当需要的时候，编译规则中需要包含调用元对象编译器，所以你不需要直接使用元对象编译器。关于元对象编译器的更多的背景知识，请看为什么Qt不用模板来实现信号和槽？。

用法

元对象编译器很典型地和包含下面这样情况地类声明地输入文件一起使用：

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

signals:
    void mySignal();

public slots:
    void mySlot();

};
```

除了上述提到地信号和槽，元对象编译器在下一个例子中还将实现对象属性。Q_PROPERTY 宏声明了一个对象属性，而Q_ENUMS 声明在这个类中的属性系统中可用的枚举类型的一个列表。在这种特殊的情况下，我们声明了一个枚举类型属性Priority，也被称为“priority”，并且读函数为priority()，写函数为setPriority()。

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

属性可以通过 Q_OVERRIDE 宏在子类中进行修改。Q_SETS 宏声明了枚举变量可以进行组合操作，也就是说可以一起读或写。另外一个宏，Q_CLASSINFO，用来给类的元对象添加名称/值这样一组数据：

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO( "Author", "Oscar Peterson")
    Q_CLASSINFO( "Status", "Very nice class")
```

```
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();
};
```

这三个概念：信号和槽、属性和元对象数据是可以组合在一起的。

元对象编译器生成的输出文件必须被编译和连接，就像你的程序中的其它的 C++代码一样；否则你的程序的连编将会在最后的连接阶段失败。出于习惯，这种操作是用下述两种方式之一解决的：

方法一：类的声明放在一个头文件（*.h文件*）中

如果在上述的文件*myclass.h*中发现类的声明，元对象编译器的输出文件将会被放在一个叫*moc_myclass.cpp*的文件中。这个文件将会像通常情况一样被编译，作为对象文件的结果是*moc_myclass.o*（在Unix下）或者*moc_myclass.obj*（在Windows下）。这个对象接着将会被包含到一个对象文件列表中，它们将会在程序的最后连编阶段被连接在一起。

方法二：类的声明放在一个实现文件（*.cpp文件*）中

如果上述的文件*myclass.cpp*中发现类的声明，元对象编译器的输出文件将会被放在一个叫*myclass.moc*的文件中。这个文件需要被实现文件包含（#include），也就是说*myclass.cpp*需要包含下面这行

```
#include "myclass.moc"
```

放在所有的代码之后。这样，元对象编译器生成的代码将会和*myclass.cpp*中普通的类定义一起被编译和连接，所以方法一中的分别编译和连接就是不需要的了。

方法一是常规的方法。方法二用在你想让实现文件自包含，或者 Q_OBJECT 类是内部实现的并且在头文件中不可见的这些情况下使用。

Makefile 中自动使用元对象编译器的方法

除了最简单的测试程序之外的任何程序，建议自动使用元对象编译器。在你的程序的Makefile文件中加入一些规则，*make*就会在需要的时候运行元对象编译器和处理元对象编译器的输出。

我们建议使用Trolltech的自由makefile生成工具，qmake，来生成你的Makefile。这个工具可以识别方法一和方法二风格的源文件，并建立一个可以做所有必要的元对象编译操作的Makefile。

另一方面如果，你想自己建立你的 Makefile，下面是如何包含元对象编译操作的一些提示。

对于在头文件中声明了 Q_OBJECT 宏的类，如果你只使用 GNU 的 make 的话，这是一个很有用的 makefile 规则：

```
moc_%.cpp: %.h
        moc $< -o $@
```

如果你想更方便地写 makefile，你可以按下面的格式写单独的规则：

```
moc_NAME.cpp: NAME.h
        moc $< -o $@
```

你必须记住要把*moc_NAME.cpp*添加到你的SOURCES（你可以用你喜欢的名字替代）变量中并且把*moc_NAME.o*或者*moc_NAME.obj*添加到你的OBJECTS变量中。

（当我们给我们的 C++源文件命名为.cpp 时，元对象编译器并不留意，所以只要你喜欢，你可以使用.C、.cc、.CC、.cxx 或者甚至.c++。）

对于在实现文件（.cpp 文件）中声明 Q_OBJECT 的类，我们建议你使用下面这样的 makefile 规则：

```
NAME.o: NAME.moc

NAME.moc: NAME.cpp
        moc -i $< -o $@
```

这将会保证make程序会在编译*NAME.cpp*之前运行元对象编译器。然后你可以把

```
#include "NAME.moc"
```

放在*NAME.cpp*的末尾，这样在这个文件中的所有的类声明被完全地知道。

调用元对象编译器 moc

这里是元对象编译器 moc 所支持地命令行选项：

-o *file*
: 将输出写到*file*而不是标准输出。

-f
: 强制在输出文件中生成#include声明。文件的名称必须符合正则表达式\.[hH][^.]*（也就是说扩展名必须以H或h开始）。这个选项只有在你的头文件没有遵循标准命名法则的时候才有用。

-i
: 不在输出文件中生成#include 声明。当一个 C++文件包含一个或多个类声明的时候你也许应该这样使用元对象编译器。然后你应该在.cpp 文件中包含（#include）元对象代码。如果-i 和-f 两个参数都出现，后出现的有效。

-nw
: 不产生任何警告。不建议使用。

-ldbg
: 把大量的 lex 调试信息写到标准输出。

-p *path*
: 使元对象编译器生成的（如果有生成的）#include声明的文件名称中预先考虑到*path*/。

-q *path*
: 使元对象编译器在生成的文件中的qt #include文件的名称中预先考虑到*path*/。

你可以明确地告诉元对象编译器不要解析头文件中的成分。它可以识别包含子字符串 MOC_SKIP_BEGIN 或者 MOC_SKIP_END 的任何 C++注释（//）。它们正如你所期望的那样工作并且你可以把它们划分为若干层次。元对象编译器所看到的最终结果就好像你把一个 MOC_SKIP_BEGIN 和一个 MOC_SKIP_END 当中的所有行删除那样。

诊断

元对象编译器将会警告关于学多在 Q_OBJECT 类声明中危险的或者不合法的构造。

如果你在你的程序的最后连编阶段得到连接错误，说 YourClass::className()是未定义的或者 YourClass 缺乏 vtbl，某样东西已经被做错。绝大多数情况下，你忘记了编译或者#include 元对象编译器产生的 C++代码，或者（在前面的情况下）没有在连接命令中包含那个对象文件。

限制

元对象编译器并不展开#include 或者#define，它简单地忽略它所遇到的所有预处理程序指示。这是遗憾的，但是在实践中它通常情况下不是问题。

元对象编译器不处理所有的 C++。主要的问题是类模板不能含有信号和槽。这里是一个例子：

```
class SomeTemplate<int> : public QFrame {
    Q_OBJECT
    ...
signals:
    void bugInMocDetected( int );
};
```

次重要的是，后面的构造是不合法的。所有的这些都可以替换为我们通常认为比较好的方案，所以去掉这些限制对于我们来说并不是高优先级的。

多重继承需要把QObject放在第一个

如果你使用多重继承，元对象编译器假设*首先*继承的类是QObject的一个子类。也就是说，确信*仅仅*首先继承的类是QObject。

```
class SomeClass : public QObject, public OtherClass {
    ...
};
```

（这个限制几乎是不可能去掉的；因为元对象编译器并不展开#include 或者#define，它不能发现基类中哪个是 QObject。）

函数指针不能作为信号和槽的参数

在你考虑使用函数指针作为信号/槽的参数的大多数情况下，我们认为继承是一个不错的替代方法。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
```

```
    Q_OBJECT
    ...
public slots:
    // 不合法的
    void apply( void (*apply)(List *, void *), char * );
};
```

你可以在这样一个限制范围内工作：

```
typedef void (*ApplyFunctionType)( List *, void * );

class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    void apply( ApplyFunctionType, char * );
};
```

有时用继承和虚函数、信号和槽来替换函数指针是更好的。

友声明不能放在信号部分或者槽部分中

有时它也许会工作，但通常情况下，友声明不能放在信号部分或者槽部分中。把它们替换到私有的、保护的或者公有的部分中。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
signals:
    friend class ClassTemplate<char>; // 错的
};
```

信号和槽不能被升级

把继承的成员函数升级为公有状态这一个 C++特征并不延伸到包括信号和槽。这里是一个不合法的例子：

```
class Whatever : public QButtonGroup {
    ...
public slots:
    void QButtonGroup::buttonPressed; // 错的
    ...
};
```

QButtonGroup::buttonPressed()槽是保护的。

C++测验：如果你试图升级一个被重载的保护成员函数将会发生什么？

1. 所有的函数都被重载。
2. 这不是标准的 C++。

类型宏不能被用于信号和槽的参数

因为元对象编译器并不展开#define，在信号和槽中类型宏作为一个参数是不能工作的。这里是一个不合法的例子：

```
#ifdef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif

class Whatever : public QObject {
    ...
signals:
    void someSignal( SIGNEDNESS(int) );
    ...
};
```

不含有参数的#define 将会像你所期望的那样工作。

嵌套类不能放在信号部分或者槽部分，也不能含有信号和槽

这里是一个例子：

```
class A {
    Q_OBJECT
public:
    class B {
    public slots:   // 错的
        void b();
        ...
    };
signals:
    class B {       // 错的
        void b();
        ...
    };
};
```

构造函数不能用于信号部分和槽部分

为什么一个人会把一个构造函数放到信号部分或者槽部分，这对于我们来说都是很神秘的。你无论如何也不能这样做（除去它偶尔能工作的情况）。请把它们放到私有的、保护的或者公有的部分中，它们本该属于的地方。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
public slots:
    SomeClass( QObject *parent, const char *name )
        : QObject( parent, name ) { } // 错的
    ...
};
```

属性的声明应该放在含有相应的读写函数的公有部分之前

在包含相应的读写函数的公有部分之中和之后声明属性的话，读写函数就不能像所期望的那样工作了。元对象编译器会抱怨不能找到函数或者解析这个类型。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
public:
    ...
    Q_PROPERTY( Priority priority READ priority WRITE setPriority ) // 错的
    Q_ENUMS( Priority ) // 错的
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};
```

根据这个限制，你应该在 Q_OBJECT 之后，在这个类的声明之前声明所有的属性：

```
class SomeClass : public QObject {
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    ...
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};
```

为什么 Qt 不用模板来实现信号和槽？

一个简单的答案是，当初 Qt 被设计的时候，因为各种各样的编译器的不充分，所以在多平台应用程序中完全使用模板机制是不可能的。甚至今天，许多被广泛使用的 C++编译器在使用高级模板的时候还是有问题的。例如，你不能安全地依靠部分模板的示例，这对一些不平常的问题领域是必要的。因此 Qt 中模板的用法不得不保守。记住 Qt 是一个多平台的工具包，在 Linux/g++平台上的进步不一定能够在其它情况下获得改进。

那些在模板执行上比较弱的编译器终将得到改进。但是，即使我们所有的用户以极好的模板支持接近一个完全现代的C++编译器的标准，我们也不会放弃通过使用我们的元对象编译器的基于字符串的途径。这里是为什么这样做的五个原因：

1. 语法问题

语法不是糖：我们用来表达我们的运算法则的语法较大程度上影响我们的代码的可读性和可维护性。Qt 中信号和槽所用的语法在实践中被证明是非常成功的。这种语法是直观的、容易使用的和容易读的。人们在学习 Qt 时发现这种语法帮助他们理解和使用信号和槽的概念——而不管它们的高度抽象和通用的性质。此外，在类定义中信号声明保证了信号就像被保护的 C++成员函数一样被保护。这帮助了程序员在刚开始的时候就获得了他们的设计权力，而不用不得不考虑设计模式。

2. 预编译程序是好的

Qt的moc（元对象编译器）提供了一种的方式除了那些编译语言的工具。它可以生成任何一个标准的C++编译器都能编译的额外的C++代码。元对象编译器读取C++源文件。如果它发现其中有一个或多个类的声明中含有“Q_OBJECT”这个宏，它就会为这些类生成另外一个包含元对象代码的C++源文件。这个由元对象编译器生成的C++源文件必须和它的类实现一起编译和连接（或者它也可以被#included到个类的源文件中）。有特色的是元对象编译器不是用手工来调用的，它可以自动地被连编系统调用，所以它不需要程序员额外的付出努力。

这里有一些其它的预编译程序，比如，rpc和idl，它们使程序或者对象能够通过进程或者machine boundaries来进行通讯。预编译程序的选择是编写编译器，专有的语言或者使用对话框或向导这些图形编程工具来生成晦涩的代码。我们能使我们的客户使用他们所喜欢的工具，而不是把他们锁定在一个专有的C++编译器或者一个特殊的集成开发环境。我们不强迫程序员把生成的代码添加到源程序仓库中，而是鼓励他们把我们的工具加入到他们的连编系统中：更加干净，更加安全和更加富有UNIX精神。

3. 灵活性为王

C++是一种标准化的、强大的和精心制作的多用途语言。它只是用来开发很多领域的软件项目的一种语言，生成许多种应用程序，从整个操作系统、数据库服务器和高性能的图形应用程序到普通的桌面应用程序。C++成功的关键之一是它着重于最大效能和最小内存占用同时保持 ANSI-C 的的兼容性的可伸缩语言设计

在这些优势当中，也有一些不利方面。对于C++，当它用来构成基于组件的图形用户界面编程的时候，静态的对象模型在使用Objective C途径的动态消息机制方面是明显的劣势。对于一个高端数据库服务器或者一个操作系统使用正确的图形用户界面前端工具的这一设计选择不是必须的。使用元对象编译器，我们可以把这一劣势转化为优势并且会加入当我们遇到安全的和有效的图形用户界面程序编程这一挑战的时候所需要的灵活性。

我们的方法比你用模板所能做到的一切更好。比如，我们有对象属性。并且我们可以重载信号和槽，当你在使用可以重载这一关键理念的语言进行程序设计的时候你会感到很自然。我们的信号只对一个类实例的大小增加了零个字节，也就是说我们能在不破坏二进制程序的兼容性的同时加入新的信号。因为我们不像模板那样过多地依靠内嵌，我们可以使得代码变得更小。添加一个新的连接就是增加一个简单地函数调用而不是一个复杂地模板函数。

另外一个好处就是我们可以在运行时探测对象的信号和槽。我们可以通过使用类型安全的名称调用而不用我们知道我们要连接的对象的确切类型来建立连接。这在一个基于模板的解决方案中是不可能的。这种运行时的自我检测扩充了一种新的功能，比如我们可以使用Qt设计器的XML格式的ui文件来生成和连接图形用户界面。

4. 调用性能不是一切

Qt 的信号和槽的执行没有基于模板的解决方案那样快。发射一个信号的时间大约和普通模板实现中的四个普通函数调用的时间差不多，Qt 要求努力控制到和十个普通函数调用差不多。这也不必惊讶，因为 Qt 机制中包括了一个通用调度器，自我测量和基本的脚本化的能力。它不过分依赖内嵌和代码扩展，并且它提供了运行时得无比安全性。Qt 的迭代（iterator）是安全的而那些基于模板的更快的系统确不是。甚至在你发射一个信号到多个接收器的过程中，那些接收器可以被安全地删除而不会导致你的程序崩溃。没有了这种安全，你的程序在调试自由的内存读或写错误这种困难情况下最终会崩溃。

虽然如此，一个基于模板的解决方案不是能比使用信号和槽更加提高应用程序的性能吗？虽然 Qt 通过一个信号调用槽的时候会增加一点时间开销是真的，这个调用的开销只占整个槽调用的开销的很小比例。以上的情况都是基于 Qt 的信号和槽系统使用典型的空槽。一旦你在槽里面做任何有意义的事情时，比如一些简单的字符串操作，调用的时间开销就可以忽略不计了。Qt 的系统非常的优化了，以至于任何东西都要求操作符 new 或者 delete（比如，字符串操作或者从一个模板容器插入/删除一些东西）的时间开销要比发射一个信号多的多。

另外：如果你在一个性能为关键的任务中的一个严紧的内部回路中使用信号和槽并且你认为这种连接是瓶颈的话，建议你使用标准的监听接口模式来替代信号和槽。当这种情况发生时，总之你也许只需要一个一对一的连接。比如，你有一个对象从网络上下载数据，你使用信号来说明所需要的数据已经到达的这种设计是非常明智的。但是如果你需要向接收者一个字节一个字节地发送数据，使用监听接口要比信号和槽好。

5. 没有限制

因为我们有元对象编译器来处理信号和槽，我们可以向它添加一些其它模板不能做的但很有用的东西。在这之中，有我们利用生成的tr()函数进行作用域翻译，和一个自我测量和扩展的运行时的类型信息的先进的属性系统。属性系统有一个独一无二的优势：没有一个

强大的和自我测量的属性系统——如果这不是不可能的——一个Qt设计器这样的强大的和通用的用户界面设计工具就很难被写出来。

带有元对象编译器预处理器的C++从本质上给我们带来对象的C的灵活性或一个Java的运行时环境，当保持C++的唯一特性和可伸缩的优点。它使得Qt成为我们今天拥有的灵活和舒适的工具。