

 **BlueMix** 点击按钮，开始云上的开发！

开始您的试用

developerWorks 中国 技术主题 Linux 文档库

QT 的信号与槽机制介绍

信号与槽作为 QT 的核心机制在 QT 编程中有着广泛的应用，本文介绍了信号与槽的一些基本概念、元对象工具以及在实际使用过程中应注意的一些问题。

唐新华 软件工程师。Email: xhsmart@263.net

2010 年 9 月 20 日

QT 是一个跨平台的 C++ GUI 应用构架，它提供了丰富的窗口部件集，具有面向对象、易于扩展、真正的组件编程等特点，更为引人注目的是目前 Linux 上最为流行的 KDE 桌面环境就是建立在 QT 库的基础之上。QT 支持下列平台：MS/WINDOWS-95、98、NT 和 2000；UNIX/X11-Linux、Sun Solaris、HP-UX、Digital Unix、IBM AIX、SGI IRIX；EMBEDDED- 支持 framebuffer 的 Linux 平台。伴随着 KDE 的快速发展和普及，QT 很可能成为 Linux 窗口平台上进行软件开发时的 GUI 首选。

概述

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。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，也是 QT 区别于其它工具包的重要地方。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，因此要正确的处理信号和槽，必须借助一个称为 moc (Meta Object Compiler) 的 QT 工具，该工具是一个 C++ 预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

在我们所熟知的很多 GUI 工具包中，窗口小部件 (widget) 都有一个回调函数用于响应它们能触发的每个动作，这个回调函数通常是一个指向某个函数的指针。但是，在 QT 中信号和槽取代了这些凌乱的函数指针，使得我们编写这些通信程序更为简洁明了。信号和槽能携带任意数量和任意类型的参数，他们是类型完全安全的，不会像回调函数那样产生 core dumps。

所有从 QObject 或其子类 (例如 QWidget) 派生的类都能够包含信号和槽。当对象改变其状态时，信号就由该对象发射 (emit) 出去，这就是对象所要做的全部事情，它不知道另一端是谁在接收这个信号。这就是真正的信息封装，它确保对象被当作一个真正的软件组件来使用。槽用于接收信号，但它们是普通的对象成员函数。一个槽并不知道是否有任何信号与自己相连接。而且，对象并不了解具体的通信机制。

你可以将很多信号与单个的槽进行连接，也可以将单个的信号与很多的槽进行连接，甚至于将一个信号与另外一个信号相连接也是可能的，这时无无论第一个信号什么时候发射系统都将立刻发射第二个信号。总之，信号与槽构造了一个强大的部件编程机制。

信号

当某个信号对其客户或所有者发生的内部状态发生改变，信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时，与其相关联的槽将被立刻执行，就象一个正常的函数调用一样。信号 - 槽机制完全独立于任何 GUI 事件循环。只有当所有的槽返回以后发射函数 (emit) 才返回。如果存在多个槽与某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，



在 IBM BlueMix 云平台上
开发并部署您的下一个应用。

开始您的试用

但是它们执行的顺序将会是随机的、不确定的，我们不能人为地指定哪个先执行、哪个后执行。

信号的声明是在头文件中进行的，QT 的 `signals` 关键字指出进入了信号声明区，随后即可声明自己的信号。例如，下面定义了三个信号：

```
signals:
void mySignal();
void mySignal(int x);
void mySignalParam(int x,int y);
```

在上面的定义中，`signals` 是 QT 的关键字，而非 C/C++ 的。接下来的一行 `void mySignal()` 定义了信号 `mySignal`，这个信号没有携带参数；接下来的一行 `void mySignal(int x)` 定义了重名信号 `mySignal`，但是它携带一个整形参数，这有点类似于 C++ 中的虚函数。从形式上讲信号的声明与普通的 C++ 函数是一样的，但是信号却没有函数体定义，另外，信号的返回类型都是 `void`，不要指望能从信号返回什么有用信息。

信号由 moc 自动产生，它们不应该在 .cpp 文件中实现。

槽

槽是普通的 C++ 成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值。

既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++ 成员函数一样，槽函数也分为三种类型，即 `public slots`、`private slots` 和 `protected slots`。

`public slots`：在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。

`protected slots`：在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。

`private slots`：在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

槽也能够声明为虚函数，这也是非常有用的。

槽的声明也是在头文件中进行的。例如，下面声明了三个槽：

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

信号与槽的关联

通过调用 `QObject` 对象的 `connect` 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：

```
bool QObject::connect ( const QObject * sender, const char * signal,
const QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 `sender` 对象中的信号 `signal` 与接收者 `receiver` 中的 `member` 槽函数联系起来。当指定信号 `signal` 时必须使用 QT 的宏 `SIGNAL()`，当指定槽函数时必须使用宏 `SLOT()`。如果发射者与接收者属于同一个对象的话，那么在 `connect` 调用中接收者参数可以省略。

例如，下面定义了两个对象：标签对象 `label` 和滚动条对象 `scroll`，并将 `valueChanged()` 信号与标签对象的 `setNum()` 相关联，另外信号还携带了一个整形参数，这样标签总是显示滚动条所处位置的值。

```
QLabel *label = new QLabel;
QScrollBar *scroll = new QScrollBar;
QObject::connect( scroll, SIGNAL(valueChanged(int)),
label, SLOT(setNum(int)) );
```

一个信号甚至能够与另一个信号相关联，看下面的例子：

```
class MyWidget : public QWidget
{
public:
    MyWidget();
    ...
signals:
    void aSignal();
    ...
private:
    ...
    QPushButton *aButton;
};
MyWidget::MyWidget()
{
    aButton = new QPushButton( this );
    connect( aButton, SIGNAL(clicked()), SIGNAL(aSignal()) );
}
```

在上面的构造函数中，MyWidget 创建了一个私有的按钮 aButton，按钮的单击事件产生的信号 clicked() 与另外一个信号 aSignal() 进行了关联。这样一来，当信号 clicked() 被发射时，信号 aSignal() 也接着被发射。当然，你也可以直接将单击事件与某个私有的槽函数相关联，然后在槽中发射 aSignal() 信号，这样的话似乎有点多余。

当信号与槽没有必要继续保持关联时，我们可以使用 disconnect 函数来断开连接。其定义如下：

```
bool QObject::disconnect ( const QObject * sender, const char * signal,
                           const QObject * receiver, const char * member ) [static]
```

这个函数断开发射者中的信号与接收者中的槽函数之间的关联。

有三种情况必须使用 disconnect() 函数：

断开与某个对象相关联的任何对象。这似乎有点不可理解，事实上，当我们在某个对象中定义了一个或者多个信号，这些信号与另外若干个对象中的槽相关联，如果我们要切断这些关联的话，就可以利用这个方法，非常之简洁。

```
disconnect( myObject, 0, 0, 0 )
或者
myObject->disconnect()
```

断开与某个特定信号的任何关联。

```
disconnect( myObject, SIGNAL(mySignal()), 0, 0 )
或者
myObject->disconnect( SIGNAL(mySignal()) )
```

断开两个对象之间的关联。

```
disconnect( myObject, 0, myReceiver, 0 )
或者
myObject->disconnect( myReceiver )
```

在 disconnect 函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数。但是发射者 sender 不能为 0，其它三个参数的值可以等于 0。

元对象工具

元对象编译器 moc (meta object compiler) 对 C++ 文件中的类声明进行分析并产生用于初始化元对象的 C++ 代码，元对象包含全部信号和槽的名字以及指向这些函数的指针。

moc 读 C++ 源文件，如果发现有 Q_OBJECT 宏声明的类，它就会生成另外一个 C++ 源文件，这个新生成的文件中包含有该类的元对象代码。例如，假设我们有一个头文件 mysignal.h，在这个文件中包含有信号或槽的声明，那么在编译之前 moc 工具就会根据该文件自动生成一个名为 mysignal.moc.h 的 C++ 源文件并将其提交给编译器；类似地，对应于 mysignal.cpp 文件 moc 工具将自动生成一个名为 mysignal.moc.cpp 文件提交给编译器。

元对象代码是 signal/slot 机制所必须的。用 moc 产生的 C++ 源文件必须与类实现一起进行编译和连接，或者用 #include 语句将其包含到类的源文件中。moc 并不扩展 #include 或者 #define 宏定义，它只是简单

的跳过所遇到的任何预处理指令。

程序样例

这里给出了一个简单的样例程序，程序中定义了三个信号、三个槽函数，然后将信号与槽进行了关联，每个槽函数只是简单的弹出一个对话框窗口。读者可以用 kdevelop 生成一个简单的 QT 应用程序，然后将下面的代码添加到相应的程序中去。

信号和槽函数的声明一般位于头文件中，同时在类声明的开始位置必须加上 Q_OBJECT 语句，这条语句是不可缺少的，它将告诉编译器在编译之前必须先应用 moc 工具进行扩展。关键字 signals 指出随后开始信号的声明，这里 signals 用的是复数形式而非单数，signals 没有 public、private、protected 等属性，这点不同于 slots。另外，signals、slots 关键字是 QT 自己定义的，不是 C++ 中的关键字。

信号的声明类似于函数的声明而非变量的声明，左边要有类型，右边要有括号，如果要向槽中传递参数的话，在括号中指定每个形式参数的类型，当然，形式参数的个数可以多于一个。

关键字 slots 指出随后开始槽的声明，这里 slots 用的也是复数形式。

槽的声明与普通函数的声明一样，可以携带零或多个形式参数。既然信号的声明类似于普通 C++ 函数的声明，那么，信号也可采用 C++ 中虚函数的形式进行声明，即同名但参数不同。例如，第一次定义的 void mySignal() 没有带参数，而第二次定义的却带有参数，从这里我们可以看到 QT 的信号机制是非常灵活的。

信号与槽之间的联系必须事先用 connect 函数进行指定。如果要断开二者之间的联系，可以使用函数 disconnect。

```
//tsignal.h
...
class TsignalApp:public QMainWindow
{
    Q_OBJECT
    ...
    // 信号声明区
    signals:
        // 声明信号 mySignal()
        void mySignal();
        // 声明信号 mySignal(int)
        void mySignal(int x);
        // 声明信号 mySignalParam(int,int)
        void mySignalParam(int x,int y);
    // 槽声明区
    public slots:
        // 声明槽函数 mySlot()
        void mySlot();
        // 声明槽函数 mySlot(int)
        void mySlot(int x);
        // 声明槽函数 mySignalParam (int , int)
        void mySignalParam(int x,int y);
}
...
//tsignal.cpp
...
TsignalApp::TsignalApp()
{
    ...
    // 将信号 mySignal() 与槽 mySlot() 相关联
    connect(this,SIGNAL(mySignal()),SLOT(mySlot()));
    // 将信号 mySignal(int) 与槽 mySlot(int) 相关联
    connect(this,SIGNAL(mySignal(int)),SLOT(mySlot(int)));
    // 将信号 mySignalParam(int,int) 与槽 mySlotParam(int,int) 相关联
    connect(this,SIGNAL(mySignalParam(int,int)),SLOT(mySlotParam(int,int)));
}
// 定义槽函数 mySlot()
void TsignalApp::mySlot()
{
    QMessageBox::about(this,"Tsignal", "This is a signal/slot sample without
parameter.");
}
// 定义槽函数 mySlot(int)
void TsignalApp::mySlot(int x)
{
    QMessageBox::about(this,"Tsignal", "This is a signal/slot sample with one
parameter.");
}
// 定义槽函数 mySlotParam(int,int)
void TsignalApp::mySlotParam(int x,int y)
{
    char s[256];
    sprintf(s,"x:%d y:%d",x,y);
    QMessageBox::about(this,"Tsignal", s);
}
```

```
void TsignalApp::slotFileNew()
{
    // 发射信号 mySignal()
    emit mySignal();
    // 发射信号 mySignal(int)
    emit mySignal(5);
    // 发射信号 mySignalParam(5, 100)
    emit mySignalParam(5,100);
}
```

应注意的问题

信号与槽机制是比较灵活的，但有些局限性我们必须了解，这样在实际的使用过程中做到有的放矢，避免产生一些错误。下面就介绍一下这方面的情况。

1. 信号与槽的效率是非常高的，但是同真正的回调函数比较起来，由于增加了灵活性，因此在速度上还是有所损失，当然这种损失相对来说是比较小的，通过在一台 i586-133 的机器上测试是 10 微秒（运行 Linux），可见这种机制所提供的简洁性、灵活性还是值得的。但如果我们要追求高效率的话，比如在实时系统中就要尽可能的少用这种机制。
2. 信号与槽机制与普通函数的调用一样，如果使用不当的话，在程序执行时也有可能产生死循环。因此，在定义槽函数时一定要避免间接形成无限循环，即在槽中再次发射所接收到的同样信号。例如，在前面给出的例子中如果在 mySlot() 槽函数中加上语句 emit mySignal() 即可形成死循环。
3. 如果一个信号与多个槽相联系的话，那么，当这个信号被发射时，与之相关的槽被激活的顺序将是随机的。
4. 宏定义不能用在 signal 和 slot 的参数中。

既然 moc 工具不扩展 #define，因此，在 signals 和 slots 中携带参数的宏就不能正确地工作，如果不带参数是可以的。例如，下面的例子中将带有参数的宏 SIGNEDNESS(a) 作为信号的参数是不合语法的：

```
#ifndef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif
class Whatever : public QObject
{
    [...]
signals:
    void someSignal( SIGNEDNESS(a) );
    [...]
};
```

5. 构造函数不能用在 signals 或者 slots 声明区域内。

的确，将一个构造函数放在 signals 或者 slots 区内有点不可理解，无论如何，不能将它们放在 private slots、protected slots 或者 public slots 区内。下面的用法是不合语法要求的：

```
class SomeClass : public QObject
{
    Q_OBJECT
public slots:
    SomeClass( QObject *parent, const char *name )
        : QObject( parent, name ) {} // 在槽声明区内声明构造函数不合语法
    [...]
};
```

6. 函数指针不能作为信号或槽的参数。

例如，下面的例子中将 void (*applyFunction)(QList*, void*) 作为参数是不合语法的：

```
class someClass : public QObject
{
    Q_OBJECT
    [...]
public slots:
    void apply(void (*applyFunction)(QList*, void*), char*); // 不合语法
};
```

你可以采用下面的方法绕过这个限制：

```
typedef void (*ApplyFunctionType)(QList*, void*);
class someClass : public QObject
{
    Q_OBJECT
    [...]
```

```
public slots:
void apply( ApplyFunctionType, char *);
};
```

7. 信号与槽不能有缺省参数。

既然 signal->slot 绑定是发生在运行时刻，那么，从概念上讲使用缺省参数是困难的。下面的用法是不合理的：

```
class SomeClass : public QObject
{
    Q_OBJECT
public slots:
    void someSlot(int x=100); // 将 x 的缺省值定义成 100，在槽函数声明中使用是错误的
};
```

8. 信号与槽也不能携带模板类参数。

如果将信号、槽声明为模板类参数的话，即使 moc 工具不报告错误，也不可能得到预期的结果。例如，下面的例子中当信号发射时，槽函数不会被正确调用：

```
[...]
public slots:
    void MyWidget::setLocation (pair<int,int> location);
[...]
```

```
[...]
public signals:
    void MyObject::moved (pair<int,int> location);
```

但是，你可以使用 typedef 语句来绕过这个限制。如下所示：

```
typedef pair<int,int> IntPair;

[...]
```

```
public slots:
    void MyWidget::setLocation (IntPair location);
[...]
```

```
public signals:
    void MyObject::moved (IntPair location);
```

这样使用的话，你就可以得到正确的结果。

9. 嵌套的类不能位于信号或槽区域内，也不能有信号或者槽。

例如，下面的例子中，在 class B 中声明槽 b() 是不合语法的，在信号区内声明槽 b() 也是不合语法的。

```
class A
{
    Q_OBJECT
public:
    class B
    {
        public slots: // 在嵌套类中声明槽不合语法
            void b();
        [...]
```

```
};
signals:
    class B
    {
        // 在信号区内声明嵌套类不合语法
        void b();
        [...]
```

```
};
};
```

10. 友元声明不能位于信号或者槽声明区内。

相反，它们应该在普通 C++ 的 private、protected 或者 public 区内进行声明。下面的例子是不合语法规范的：

```
class someClass : public QObject
{
    Q_OBJECT
[...]
```

```
signals: // 信号定义区
    friend class ClassTemplate; // 此处定义不合语法
};
```

参考资料

[QT](#)

[KDevelop](#)



IBM Bluemix 资源中心

文章、教程、演示，帮助您构建、部署和管理云应用。

developerWorks 中文社区

立即加入来自 IBM 的专业 IT 社交