# QThread Re-entrant & thread-Safe

deepwaterooo

April 18, 2015

## Contents



# 1 QT 的信号与槽机制

## 1.1 概述

- 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。

## 1.2 信号

- 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，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。

## 1.3 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);
```

## 1.4 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。

- 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 Object * 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。

## 1.5 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 $Q_{OBJECT}$ ， $_{C}$++ 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。

## 1.6 examples

```
//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 (, intint)
    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, (5100)
    emit mySignalParam(5, 100);
}
```

## 1.7 应注意的问题

- 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注意避免间接形成无限循环，即在槽中再次发射所接收到的同样信号。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：

```
#ifdef 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); // 将的缺省值定义成，在槽函数声明中使用是错误的x100
};
```

- 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 location);
public signals:
    void MyObject::moved (pair location);
```

  - 但是，你可以使用 typedef 语句来绕过这个限制。如下所示：

```
typedef pair 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
};
```

# 2 Qt 多线程之可重入与线程安全

- 可重入：假如一个类的任何函数在此类的多个不同的实例上，可以被多个线程同时调用，那么这个类被称为是“可重入”的。
- 线程安全：假如不同的线程作用在同一个实例上仍可以正常工作，那么称之为“线程安全”的。

## 2.1 QObject 可重入性

- QObject 是可重入的。它的大多数非 GUI 子类，像 QTimer,QTcpSocket,QUdpSocket,QHttp,QFtp,QProcess 也是可重入的，在多个线程中同时使用这些类是可能的。
- 需要注意的是，这些类被设计成在一个单线程中创建与使用，因此，在一个线程中创建一个对象，而在另外的线程中调用它的函数，这样的行为不能保证工作良好。
- 有三种约束需要注意：
  - QObject 的孩子总是应该在它父亲被创建的那个线程中创建。这意味着，你绝不应该传递 QThread 对象作为另一个对象的父亲 (因为 QThread 对象本身会在另一个线程中被创建)
  - 事件驱动对象仅仅在单线程中使用。明确地说，这个规则适用于" 定时器机制 "与" 网格模块 "，举例来讲，你不应该在一个线程中开始一个定时器或是连接一个套接字，当这个线程不是这些对象所在的线程。
  - 你必须保证在线程中创建的所有对象在你删除 QThread 前被删除。这很容易做到: 你可以 run() 函数运行的栈上创建对象。
- 尽管 QObject 是可重入的，但 GUI 类，特别是 QWidget 与它的所有子类都是不可重入的。它们仅用于主线程。正如前面提到过的，QCoreApplication::exec() 也必须从那个线程中被调用。实践上，不会在别的线程中使用 GUI 类，它们工作在主线程上，把一些耗时的操作放入独立的工作线程中，当工作线程运行完成，把结果在主线程所拥有的屏幕上显示。

## 2.2 逐线程事件循环

- 每个线程可以有它的事件循环，初始线程开始它的事件循环需使用 QCoreApplication::exec(), 别的线程开始它的事件循环需要用 QThread::exec(). 像 QCoreApplication 一样，QThread 提供了 exit(int) 函数，一个 quit() slot。
- 线程中的事件循环，使得线程可以使用那些需要事件循环的非 GUI 类 (如，QTimer,QTcpSocket,QProcess)。也可以把任何线程的 signals 连接到特定线程的 slots，也就是说信号 -槽机制是可以跨线程使用的。对于在 QApplication 之前创建的对象，QObject::thread() 返回 0, 这意味着主线程仅为这些对象处理投递事件，不会为没有所属线程的对象处理另外的事件。
- 可以用 **QObject::moveToThread()** 来改变它和它孩子们的线程亲缘关系，假如对象有父亲，它不能移动这种关系。在另一个线程 (而不是创建它的那个线程) 中 delete QObject 对象是不安全的。除非你可以保证在同一时刻对象不在处理事件。可以用 QObject::deleteLater(), 它会投递一个 DeferredDelete 事件，这会被对象线程的事件循环最终选取到。
- 假如没有事件循环运行，事件不会分发给对象。举例来说，假如你在一个线程中创建了一个 QTimer 对象，但从没有调用过 exec(), 那么 QTimer 就不会发射它的 timeout() 信号. 对 deleteLater() 也不会工作。(这同样适用于主线程)。你可以手工使用线程安全的函数 QCoreApplication::postEvent()，在任何时候，给任何线程中的任何对象投递一个事件，事件会在那个创建了对象的线程中通过事件循环派发。事件过滤器在所有线程中也被支持，不过它限定被监视对象与监视对象生存在同一线程中。类似地，QCoreApplication::sendEvent(不是 postEvent()), 仅用于在调用此函数的线程中向目标对象投递事件。

## 2.3 从别的线程中访问 QObject 子类

- QObject 和所有它的子类是非线程安全的。这包括整个的事件投递系统。需要牢记的是，当你正从别的线程中访问对象时，事件循环可以向你的 QObject 子类投递事件。假如你调用一个不生存在当前线程中的 QObject 子类的函数时，你必须用 mutex 来保护 QObject 子类的内部数据，否则会遭遇灾难或非预期结果。像其它的对象一样，QThread 对象生存在创建它的那个线程中 —不是当 QThread::run() 被调用时创建的那个线程。一般来讲，在你的 QThread 子类中提供 slots 是不安全的，除非你用 mutex 保护了你的成员变量。
- 另一方面，你可以安全的从 QThread::run() 的实现中发射信号，因为信号发射是线程安全的。

## 2.4 跨线程的信号 -槽

- Qt 支持三种类型的信号 -槽连接：
  - 1，直接连接，当 signal 发射时，slot 立即调用。此 slot 在发射 signal 的那个线程中被执行 (不一定是接收对象生存的那个线程 (?))

- 2，队列连接，当控制权回到对象属于的那个线程的事件循环时，slot 被调用。此 slot 在接收对象生存的那个线程中被执行
- 3，自动连接 (缺省)，假如信号发射与接收者在同一个线程中，其行为如直接连接，否则，其行为如队列连接。

- 连接类型可能通过以向 connect() 传递参数来指定。注意的是，当发送者与接收者生存在不同的线程中，而事件循环正运行于接收者的线程中，使用直接连接是不安全的。同样的道理，调用生存在不同的线程中的对象的函数也是不是安全的。QObject::connect() 本身是线程安全的。

### 2.5 多线程与隐含共享

- Qt 为它的许多值类型使用了所谓的隐含共享 (implicit sharing) 来优化性能。原理比较简单，共享类包含一个指向共享数据块的指针，这个数据块中包含了真正原数据与一个引用计数。把深拷贝转化为一个浅拷贝，从而提高了性能。这种机制在幕后发生作用，程序员不需要关心它。如果深入点看，假如对象需要对数据进行修改，而引用计数大于 1，那么它应该先 detach()。以使得它修改不会对别的共享者产生影响，既然修改后的数据与原来的那份数据不同了，因此不可能再共享了，于是它先执行深拷贝，把数据取回来，再在这份数据上进行修改。例如：

```
void QPen::setStyle(Qt::PenStyle style){
     detach();           // detach from common data
     d->stylestyle = style;   // set the style member
}
void QPen::detach(){
    if (d->ref != 1) {
     ...              // perform a deep copy
     }
}
```

- 一般认为，隐含共享与多线程不太和谐，因为有引用计数的存在。对引用计数进行保护的方法之一是使用 mutex, 但它很慢，Qt 早期版本没有提供一个满意的解决方案。从 4.0 开始，隐含共享类可以安全地跨线程拷贝，如同别的值类型一样。它们是 完全可重入 的。隐含共享真的是"implicit"。它使用汇编语言实现了原子性引用计数操作，这比用 mutex 快多了。
- 假如你在多个线程中同进访问相同对象，你也需要用 mutex 来串行化访问顺序, 就如同其他可重入对象那样。总的来讲，隐含共享真的给”隐含“掉了，在多线程程序中，你可以把它们看成是一般的，非共享的，可重入的类型，这种做法是安全的。

## 3 QThread 使用方法: QThread 中的 slots 在那个线程中执行？

Reference: http://mobile.51cto.com/symbian-268690.htm

### 3.1 QThread::run

- run 对于线程的作用相当于 main 函数对于应用程序。它是线程的入口，run 的开始和结束意味着线程的开始和结束。原文如下: The run() implementation is for a thread what the main() entry point is for the application. All code executed in a call stack that starts in the run() function is executed by the new thread, and the thread finishes when the function returns.

```
class Thread : public QThread {
    Q_OBJECT
public:
    Thread(QObject* parent = 0)
    : QThread(parent) {
    }
public slots:
    void slot() { ... }
signals:
    void sig();
protected:
    void run() { ...}
};
```

```
int main(int argc, char** argv) {
    ...
    Thread thread;
    ...
}
```

- 对照前面的定理，run 函数中的代码时确定无疑要在次线程中运行的，那么其他的呢？比如 slot 是在次线程还是主线程中运行？

## 3.2 QObject::connect

### 3.2.1 three connection types

1. 自动连接 (Auto Connection)
   - 这是默认设置
   - 如果发送者和接收者处于同一线程，则等同于直接连接
   - 如果发送者和接受者位于不同线程，则等同于队列连接
   - 也就是这说，只存在下面两种情况
2. 直接连接 (Direct Connection)
   - 当信号发射时，槽函数将直接被调用。
   - 无论槽函数所属对象在哪个线程，槽函数都在发射者所在线程执行。
3. 队列连接 (Queued Connection)
   - 当控制权回到接受者所在线程的事件循环时，槽函数被调用。
   - 槽函数在接收者所在线程执行。

### 3.2.2 explain by examples

- 不妨继续拿前面的例子来看，slot 函数是在主线程还是次线程中执行呢？
- 定理二强调两个概念：发送者所在线程和接收者所在线程。而 slot 函数属于我们在 main 中创建的对象 thread，即 thread 属于主线程
  - 队列连接告诉我们：槽函数在接受者所在线程执行。即 slot 将在主线程执行
  - 直接连接告诉我们：槽函数在发送者所在线程执行。发送者在那个线程呢？？不定！
  - 自动连接告诉我们：二者不在同一线程时，等同于队列连接。即 slot 在主线程执行
- 要彻底理解这几句话，你可能需要看 **Qt meta-object** 系统 和 **Qt event** 系统
  - QThread 是用来管理线程的，它所处的线程和它管理的线程并不是同一个东西
  - QThread 所处的线程，就是执行 QThread t(0) 或 QThread * t=new QThread(0) 的线程。也就是咱们这儿的主线程
  - QThread 管理的线程，就是 run 启动的线程。也就是次线程
  - 因为 QThread 的对象在主线程中，所以他的 slot 函数会在主线程中执行，而不是次线程。除非：QThread 对象在次线程中
  - slot 和信号是直接连接，且信号所属对象在次线程中
- 但上两种解决方法都不好，因为 QThread 不是这么用的 (Bradley T. Hughes)

### 3.3 主线程 (信号) ~ QThread(槽)

- 这是 Qt Manual 和例子中普遍采用的方法。但由于 manual 没说槽函数是在主线程执行的，所以不少人都认为它应该是在次线程执行了。
  - 定义一个 Dummy 类，用来发信号
  - 定义一个 Thread 类，用来接收信号
  - 重载 run 函数，目的是打印 threadid

```
#include <QtCore/QCoreApplication>
#include <QtCore/QObject>
#include <QtCore/QThread>
#include <QtCore/QDebug>

class Dummy : public QObject {
    Q_OBJECT
public:
    Dummy(){}
public slots:
    void emitsig() {
    emit sig();
    }
signals:
    void sig();
};

class Thread : public QThread {
    Q_OBJECT
public:
    Thread(QObject* parent = 0)
    : QThread(parent) {
    //moveToThread(this);
    }
public slots:
    void slot_main () {
    qDebug() << "from thread slot_main:" << currentThreadId();
    }
protected:
    void run() {
    qDebug() << "thread thread:" << currentThreadId();
    exec();
    }
};

//#include "main.moc"
int main(int argc, char *argv[]) {
    QCoreApplication a(argc, argv);
    qDebug() << "main thread:" << QThread::currentThreadId();
    Thread thread;
    Dummy dummy;
    QObject::connect(&dummy, SIGNAL(sig()), &thread, SLOT(slot_main()));
    thread.start();
    dummy.emitsig();
    return a.exec();
}
```

- 然后看到结果 (具体值每次都变，但结论不变)

```
main thread:           0x1a40
from thread slot_main: 0x1a40
thread thread:         0x1a48

Mine here:
Starting /home/jenny/480/qt/build-dummyThread-Unnamed-Debug/dummyThread...
```

```
main thread:           140534496016256
from thread slot_main: 140534496016256
thread thread:         140534421948160
```

- 看到了吧，槽函数的线程和主线程是一样的！
- 如果你看过 Qt 自带的例子，你会发现 QThread 中 slot 和 run 函数共同操作的对象，都会用 QMutex 锁住。为什么？因为 slot 和 run 处于不同线程，需要线程间的同步！
- 如果想让槽函数 slot 在次线程运行 (比如它执行耗时的操作，会让主线程死掉)，怎么解决呢？
- 注意：发送 dummy 信号是在主线程，接收者 thread 也在主线程中。参考我们前面的结论，很容易想到: 将 thread 放到次线程中不就行了这也是代码中注释掉的 moveToThread(this) 所做的，去掉注释，你会发现 slot 在次线程中运行

```
main thread:           0x13c0
thread thread:         0x1de0
from thread slot_main: 0x1de0

Mine here:
Starting /home/jenny/480/qt/build-dummyThread-Unnamed-Debug/dummyThread...
main thread:           140371166443392
thread thread:         140371092375296
from thread slot_main: 140371092375296
```

- 这可以工作，但这是 Bradley T. Hughes 强烈批判的用法。推荐的方法后面会给出。

## 3.4 run 中信号与 QThread 中槽

Reference: http://mobile.51cto.com/symbian-268690_1.htm

- examples:
  - 定义一个 Dummy 类，在 run 中发射它的信号
  - 也可以在 run 中发射 Thread 中的信号，而不是 Dummy（效果完全一样），QThread 定义槽函数，重载 run 函数

```
#include <QtCore/QCoreApplication>
#include <QtCore/QObject>
#include <QtCore/QThread>
#include <QtCore/QDebug>

class Dummy : public QObject {
    Q_OBJECT
public:
    Dummy(QObject* parent = 0)
    : QObject(parent) {}
public slots:
    void emitsig() {
    emit sig();
    }
signals:
    void sig();
};

class Thread : public QThread {
    Q_OBJECT
public:
    Thread(QObject* parent = 0)
    : QThread(parent) {
    //moveToThread(this);
    }
public slots:
    void slot_thread() {
```

```
    qDebug() << "from thread slot_thread:"  << currentThreadId();
    }
signals:
    void sig();
protected:
    void run() {
    qDebug() << "thread thread:" << currentThreadId();
    Dummy dummy;
    connect(&dummy, SIGNAL(sig()), this, SLOT(slot_thread()));
    dummy.emitsig();
    exec();
    }
};

//#include "main.moc"
int main(int argc, char *argv[]) {
    QCoreApplication a(argc, argv);
    qDebug() << "main thread:" << QThread::currentThreadId();
    Thread thread;
    thread.start();
    return a.exec();
}
```

- 想看结果么？

```
main thread:              0x15c0
thread thread:            0x1750
from thread slot_thread: 0x15c0

Mine here:
Starting /home/jenny/480/qt/build-dummyThread-Unnamed-Debug/dummyThread...
main thread:              140388248221568
thread thread:            140388174153472
from thread slot_thread: 140388248221568
```

- 其实没悬念，肯定是主线程
- thread 对象本身在主线程。所以它的槽也在要在主线程执行，如何解决呢？
  - （方法一）前面提了 moveToThread, 这儿可以用，而且可以解决问题。当同样，是被批判的对象。

```
uncomment moveToThread(this); line :
Starting /home/jenny/480/qt/build-dummyThread-Unnamed-Debug/dummyThread...
main thread:              140217092188032
thread thread:            140217018119936
from thread slot_thread: 140217018119936
```

  - （方法二）注意哦，这儿我们的信号时次线程发出的，对比 connect 连接方式，会发现：
    * 采用直接连接，槽函数将在次线程 (信号发出的线程) 执行
    * 这个方法不太好，因为你需要处理 slot 和它的对象所在线程的同步。需要 QMutex 一类的东西 (have **NOT** tried this method yet~!!)
  - （方法三）推荐的方法，其实，这个方法太简单，太好用了。定义一个普通的 QObject 派生类，然后将其对象 move 到 QThread 中。使用信号和槽时根本不用考虑多线程的存在。也不用使用 QMutex 来进行同步，Qt 的事件循环会自己自动处理好这个。

```
#include <QtCore/QCoreApplication>
#include <QtCore/QObject>
#include <QtCore/QThread>
#include <QtCore/QDebug>

class Dummy : public QObject {
    Q_OBJECT
public:
    Dummy(QObject* parent = 0)
```

```
    : QObject(parent) {}
public slots:
    void emitsig() {
    emit sig();
    }
signals:
    void sig();
};

class Object : public QObject {
    Q_OBJECT
public:
    Object(){}
public slots:
    void slot() {
    qDebug() << "from thread slot:"  << QThread::currentThreadId();
    }
};

//#include "main.moc"
int main(int argc, char *argv[]) {
    QCoreApplication a(argc, argv);
    qDebug() << "main thread:" << QThread::currentThreadId();
    QThread thread;
    Object obj;
    Dummy dummy;
    obj.moveToThread(&thread);
    QObject::connect(&dummy, SIGNAL(sig()), &obj, SLOT(slot()));
    thread.start();
    dummy.emitsig();
    return a.exec();
}
```

* 结果：恩，slot 确实不在主线程中运行（这么简单不值得欢呼么？）

```
main thread:      0x1a5c
from thread slot: 0x186c

Mine here:
Starting /home/jenny/480/qt/build-dummyThread-Unnamed-Debug/dummyThread...
main thread:      139964716550016
from thread slot: 139964642481920
```

# 4 QT 中关于信号与槽机制的实现原理：源代码分析

本文介绍的内容是 QT 中关于信号与槽机制的实现原理，每个对象都有一个相应的纪录该对象的元对象，关于元对象的类在本文中有所介绍。

## 4.1 每个对象都有一个相应的纪录该对象的元对象

关于元对象的类：下面介绍有两种

### 4.1.1 QMetaObject 类

```
/*生成元对象需要的输入参数************************************/
// 类名
const char * const class_name;

// 父类名
QMetaObject *superclass;

// 记录信息slot
const QMetaData * const slot_data;
```

```
// 记录槽的个数
int n_slots;

// 记录信息signal
const QMetaData * const signal_data;

// 记录信号的个数
int n_signals;

/****************** 元对象类提供的方法*************************/
int numSlots( bool super = FALSE ) const;    // 返回槽的个数
int numSignals( bool super = FALSE ) const; // 返回信号的个数
int findSlot( const char *, bool super = FALSE ) const;    // 查找槽
int findSignal( const char *, bool super = FALSE ) const; // 查找信号

// 返回指定位置的槽
const QMetaData *slot( int index, bool super = FALSE ) const;

// 返回指定位置的信号
const QMetaData *signal( int index, bool super = FALSE ) const;

// 所有槽名字的列表
QStrList slotNames( bool super = FALSE ) const;

// 所有信号名字的列表
QStrList signalNames( bool super = FALSE ) const;

// 槽的起始索引
int slotOffset() const;

// 信号的起始索引
int signalOffset() const;

/*两个获取类的元对象的方法*************************************/
static QMetaObject *metaObject( const char *class_name );
static bool hasMetaObject( const char *class_name );
```

### 4.1.2 QMetaData 类

```
  // 记录元对象数据for 信号与槽
struct QMetaData {
    const char *name;        // 名称
    const QUMethod* method; // 详细描述信息
    enum Access { Private, Protected, Public };
    Access access;           // 访问权限
};
```

## 4.2 QObject 类实现了信号与槽机制

它利用元对象纪录的信息，实现了信号与槽机制.

### 4.2.1 信号与槽建立连接的实现

1. 接口函数:

```
    // 连接
    // 参数发送对象，信号，接收对象，处理信号的信号槽(/)
    static bool connect(const QObject *sender, const char *signal,
                        const QObject *receiver, const char *member );
    bool connect(const QObject *sender, const char *signal,
                 const char *member ) const;

    static bool disconnect(const QObject *sender, const char *signal,
                        const QObject *receiver, const char *member);
    bool disconnect(const char *signal = 0,
```

```
                const QObject *receiver = 0, const char *member = 0 );
bool disconnect(const QObject *receiver, const char *member = 0 );

// 连接的内部实现
// 发送对象，信号的索引，接收对象，处理信号的类型，处理信号信号槽的索引(/)
static void connectInternal(const QObject *sender, int signal_index,
                            const QObject *receiver, int membcode, int member_index );
static bool disconnectInternal(const QObject *sender, int signal_index,
                            const QObject *receiver, int membcode, int member_index );
```

2. 信号与槽连接的实现原理：

```
// 一阶段
bool QObject::connect(const QObject *sender,    // 发送对象
              const char *signal,       // 信号
              const QObject *receiver, // 接收对象
              const char *member        // 槽
              ) {
     // 检查发送对象，信号，接收对象，槽不为null
    if ( sender == 0 || receiver == 0 || signal == 0 || member == 0 ) {
        return false;
    }

    // 获取发送对象的元对象
    QMetaObject *smeta = sender->metaObject();
    // 检查信号
    if ( !check_signal_macro( sender, signal, "connect", "bind" ) )
        return false;
    // 获取信号的索引
    int signal_index = smeta->findSignal( signal, true );
    if ( signal_index < 0 ) {                // normalize and retry
        nw_signal = qt_rmWS( signal-1 ); // remove whitespace
        signal = nw_signal.data()+1;         // skip member type code
        signal_index = smeta->findSignal( signal, true );
    }
    // 如果信号不存在，则退出
    if ( signal_index < 0  ) {                    // no such signal
        return false;
    }

    // 获取信号的元数据对象
    const QMetaData *sm = smeta->signal( signal_index, true );
    // 获取信号名字
    signal = sm->name;
    // 获取处理信号的类型（是信号槽）/
    int membcode = member[0] - '0';         // get member code    // **** membcode
    // 发送信号对象
    QObject *s = (QObject *)sender;         // we need to change them
    // 接收信号对象
    QObject *r = (QObject *)receiver;       //   internally
    // 获取接收对象的元对象
    QMetaObject *rrmeta = r->metaObject();
    int member_index = -1;

    switch ( membcode ) {                // get receiver member
    case QSLOT_CODE:// 如果是槽
        // 获取槽索引
        member_index = rmeta->findSlot( member, true );
        if ( member_index < 0 ) {            // normalize and retry
            nw_member = qt_rmWS(member);     // remove whitespace
            member = nw_member;
            member_index = rmeta->findSlot( member, true );
        }
        break;
    case QSIGNAL_CODE:// 如果是信号
```

```
        // 获取信号索引
        member_index = rmeta->findSignal( member, true );
        if ( member_index < 0 ) {           // normalize and retry
            nw_member = qt_rmWS(member);     // remove whitespace
            member = nw_member;
            member_index = rmeta->findSignal( member, true );
        }
        break;
    }
    // 如果接收对象不存在相应的信号或槽，则退出
    if ( member_index < 0  ) {
        return false;
    }
    // 检查连接的参数发送的信号，接收对象，处理信号的槽或信号()
    if ( !s->checkConnectArgs(signal,receiver,member) ) {
        return false;
    } else {
    // 获取处理信号的元数据对象
        const QMetaData *rm = membcode == QSLOT_CODE ?
        rmeta->slot( member_index, true ) :
        rmeta->signal( member_index, true );
        if ( rm ) {
            // 建立连接
            // 发送信号的对象，信号的索引，接收信号的对象，处理信号的类型，处理信号的索引()
            connectInternal( sender, signal_index, receiver, membcode, member_index );
        }
    }
    return true;
}

// 二阶段
// 建立连接
// 发送信号的对象，信号的索引，接收信号的对象，处理信号的类型，处理信号的索引()
void QObject::connectInternal( const QObject *sender, int signal_index,
                  const QObject *receiver, int membcode, int member_index )   {
    // 发送信号的对象
    QObject *s = (QObject*)sender;
    // 接收信号的对象
    QObject *r = (QObject*)receiver;
    // 如果发送对象的连接查询表为，则建立null
    if ( !s->connections ) {                // create connections lookup table
        s->connections = new QSignalVec( signal_index+1 );
        Q_CHECK_PTR( s->connections );
        s->connections->setAutoDelete( true );
    }
    // 获取发送对象的相应信号的连接列表
     QConnectionList *clist = s->connections->at( signal_index );
    if ( !clist ) {                         // create receiver list
        clist = new QConnectionList;
        Q_CHECK_PTR( clist );
        clist->setAutoDelete( true );
        s->connections->insert( signal_index, clist );
    }
    QMetaObject *rrmeta = r->metaObject();
    const QMetaData *rm = 0;
    switch ( membcode ) {               // get receiver member
    case QSLOT_CODE:
        rm = rmeta->slot( member_index, true );
        break;
    case QSIGNAL_CODE:
        rm = rmeta->signal( member_index, true );
        break;
    }
    // 建立连接
```

```
    QConnection *c = new QConnection( r, member_index, rm ? rm->name : "qt_invoke", memb
    Q_CHECK_PTR( c );
    // 把连接添加到发送对象的连接列表中
    clist->append( c );
    // 判断接收对象的发送对象列表是否为null
    if ( !r->senderObjects ) {                  // create list of senders
        // 建立接收对象的发送对象列表
        r->senderObjects = new QSenderObjectList;
    }
    // 把发送对象添加到发送对象列表中
    r->senderObjects->append( s );              // add sender to list
}
```

3. 信号发生时激活的操作函数。激活 slot 的方法

```
// 接口:
void QObject::activate_signal( int signal ) {
    #ifndef QT_NO_PRELIMINARY_SIGNAL_SPY
    if ( qt_preliminary_signal_spy ) {
        //信号没有被阻塞
        //信号>=0
        //连接列表不为空，或者信号对应的连接存在
        if ( !signalsBlocked() && signal >= 0 &&
             ( !connections || !connections->at( signal ) ) ) {
            //
            QUObject o[1];
            qt_spy_signal( this, signal, o );
            return;
        }
    }
    #endif
    if ( !connections || signalsBlocked() || signal < 0 )
        return;
    //获取信号对应的连接列表
    QConnectionList *clist = connections->at( signal );
    if ( !clist )
        return;
    QUObject o[1];
    //
    activate_signal( clist, o );
}

void QObject::activate_signal( QConnectionList *clist, QUObject *o )   {
    if ( !clist )
        return;
#ifndef QT_NO_PRELIMINARY_SIGNAL_SPY
    if ( qt_preliminary_signal_spy )
        qt_spy_signal( this, connections->findRef( clist), o );
#endif
    QObject *object;
    //发送对象列表
    QSenderObjectList* sol;
    //旧的发送对象
    QObject* oldSender = 0;
    //连接
    QConnection *c;
    if ( clist->count() == 1 ) { // save iterator
        //获取连接
        c = clist->first();
        //
        object = c->object();
        //获取发送对象列表
        sol = object->senderObjects;
        if ( sol ) {
        //获取旧的发送对象
```

```
            oldSender = sol->currentSender;
            //
            sol->ref();
            //设置新的发送对象
            sol->currentSender = this;
            }
            if ( c->memberType() == QSIGNAL_CODE )//如果是信号，则发送出去
                object->qt_emit( c->member(), o );
            else
                object->qt_invoke( c->member(), o );//如果是槽，则执行
            //
            if ( sol ) {
                //设置恢复为旧的发送对象
                sol->currentSender = oldSender;
                if ( sol->deref() )
                    delete sol;
            }
        } else {
            QConnection *cd = 0;
            QConnectionListIt it(*clist);
            while ( (c=it.current()) ) {
                ++it;
                if ( c == cd )
                    continue;
                ccd = c;
               object = c->object();
                //操作前设置当前发送对象
                sol = object->senderObjects;
                if ( sol ) {
                    oldSender = sol->currentSender;
                    sol->ref();
                    sol->currentSender = this;
                }
                //如果是信号，则发送出去
                if ( c->memberType() == QSIGNAL_CODE ){
                    object->qt_emit( c->member(), o );
                }
                //如果是槽，则执行
                else {
                    object->qt_invoke( c->member(), o );
                }
                //操作后恢复当前发送对象
                if (sol ) {
                    sol->currentSender = oldSender;
                    if ( sol->deref() )
                        delete sol;
                }
            } // while
        }
    }
```

# 5 QT 中实现 Thread 与 GUI 主线程连通方法

http://mobile.51cto.com/symbian-270684.htm

## 5.1

# 6 Other Reference

- 了解 Qt 多线程编程新手必学 (1)

http://mobile.51cto.com/symbian-269482.htm

- QT 核心编程之 Qt 线程（3）

http://mobile.51cto.com/symbian-270589.htm

- 浅谈 Qt 中多线程编程

http://mobile.51cto.com/symbian-268343.htm

- 解析 QT 多线程程序详细设计上篇

http://mobile.51cto.com/symbian-270667.htm

- 解析 QT 多线程程序详细设计之 QObject 可重入性下篇

http://mobile.51cto.com/symbian-270674.htm

- Qt 的插件机制 (1)

http://mobile.51cto.com/symbian-268027.htm

- Qt 和 KDE 在未来将面临新的挑战和机遇

http://mobile.51cto.com/hot-247522.htm

- QT 源码之 Qt 信号槽机制与事件机制的联系

http://mobile.51cto.com/symbian-270997.htm

- QT 进程间通信详细介绍

http://mobile.51cto.com/symbian-270726.htm

- 解析 QT 静态库和动态库

http://mobile.51cto.com/symbian-267846.htm

- Linux 下 QT 实现串口通讯小实例

http://mobile.51cto.com/symbian-270754.htm

- Linux 虚拟串口及 Qt 串口通信实例

http://mobile.51cto.com/symbian-270768.htm