

# QT4.7 多线程编程和程序架构

## 作者:向仔州

多线程创建.....	3
objectName()	
setObjectName()	
terminate()	
多线程同步有什么用 ? .....	8
wait()	
多线程的互斥 , 也就是两个线程或者多个线程同时读写一个变量的问题.....	11
Qmutex	
Qmutex::lock()	
Qmutex::unlock()	
信号量也是一种特殊的线程锁 , 信号量使用.....	19
Qsemaphore	
Qsemaphore::acquire()	
Qsemaphore::release()	
用银行家算法来练习多线程和互斥锁操作.....	22
QT 多线程使用信号与槽.....	25
线程类定义的槽函数怎么没有在自己的线程类里面执行 ? .....	29
moveToThread()	
exec()	
quit()	
open()	
write()	
flush()	
close()	
QT 的 connect 链接信号到槽的函数有几个功能没有设置 , 我们一直用的是默认功能 , connect 有五个形参 , 我们只用了四个。 .....	35
Qt::QueuedConnection	
Qt::DirectConnection	
Qt::BlockingQueuedConnet	
对象创建的线程生命周期是否会随着对象执行完后线程跟着消失 , 该问题出现在子函数里面创建对象的情况.....	41
Qt 另一种创建线程方式.....	45



## 多线程创建

在linux C 语言多线程编程中，线程是一个函数，一个线程一个函数

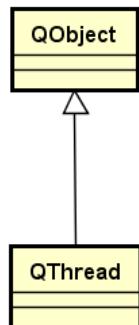
```
void func(int *arg) {  
    char buf[50];  
    while(1)  
    {  
        memset(buf, 0, sizeof(buf));  
        read(0, buf, 50);  
        printf("读出键盘内容 : %s\n", buf);  
    }  
}  
  
void main()  
{  
    int ret = -1, fd = -1;  
    pthread_t th = -1;  
  
    ret = pthread_create(&th, NULL, (void *)func, NULL);  
    if(ret != 0)  
        printf("func creat pthread error\n");  
}
```

线程函数创建

可以给线程函数传入参数

创建的线程函数

在 QT 中多线程是一个对象



注意：

1. Qt 中的线程以对象的形式被创建和使用
2. 每一个线程对应着一个 QThread 对象

```
class MyThread : public QThread // 创建线程类  
{  
protected:  
    void run() // 线程入口函数  
    {  
        for(int i=0; i<5; i++)  
        {  
            qDebug() << objectName() << " : " << i;  
  
            sleep(1); // 暂停 1 秒  
        }  
    }  
};
```

QT 的多线程就是自己创建一个类，这个类里面的 run 函数执行的代码就是线程

而且

而且一个类可以创建多个对象，每个对象都有一个 run，也就是每个对象就是个线程

```

#include <QThread> // 创建线程需要的类
#include <QDebug>

class mythread : public QThread
{
protected:
    void run()
    {
        while(1)
        {
            qDebug() << "thread";
        }
    }
};

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);

    mythread thread1; // 创建线程 thread1
    thread1.setObjectName("thread1"); // 设置线程名
    thread1.start(); // 启动线程

    return a.exec();
}

```

这就是自己定义的线程类，在 run 写自己想要的代码

线程类创建好后要在主函数创建对象才行

每个线程都可以传入一个字符串数据

然后启动线程

```

thread

```

线程执行过程

```

class mythread : public QThread
{
protected:
    void run()
    {
        while(1)
        {
            qDebug() << objectName(); //将setObjectName获取的外部字符串传入进来
            sleep(1); //延时1秒
        }
    }
};

```

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    mythread thread1; //创建线程thread1
    thread1.setObjectName("thread1"); //将字符串传入线程1
    thread1.start(); //启动线程

    mythread thread2; //创建线程thread2, 用的是同一个线程类的run函数
    thread2.setObjectName("thread2"); //将字符串传入线程2
    thread2.start(); //启动线程2

    return a.exec();
}

```

"thread1"  
"thread2"  
"thread1"  
"thread2"  
"thread2"  
"thread1"

线程 1 和线程 2 谁先执行是随机的

这就是 QT 的多线程，和 linux C 语言线程不一样，C 语言是一个函数对应一个线程，但是 QT 可以一个函数对应多个对象创建出来的线程。

## 我们如何关闭线程

在工程开发中 terminate() 是禁止使用的！ terminate() 会

使得操作系统暴力终止线程，而不会考虑数据完整性，资源

释放等问题！

```

class mythread : public QThread
{
protected:
    volatile bool flag;
    void run()
    {
        for(int i = 0; i < 10; i++)
        {
            qDebug() << objectName() << "=" << i; //将setObjectName获取的外部字符串传入进来
            msleep(500); //延时500毫秒
        }
    }
};

```

我发现一个问题，除了创建的线程在执行外，主程序 main 也是个线程，也在并行执行

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    qDebug() << "thread start";

    mythread thread1; // 创建线程 thread1
    thread1.setObjectName("thread1"); // 将字符串传入线程1
    thread1.start(); // 启动线程

    qDebug() << "thread end";
    return a.exec();
}

```

```

thread start
thread end
"thread1" = 0
"thread1" = 1
"thread1" = 2
"thread1" = 3
"thread1" = 4
"thread1" = 5
"thread1" = 6
"thread1" = 7
"thread1" = 8
"thread1" = 9

```

因为主线程 main 没有延时，所以主线程先执行完，然后子线程 thread1 后执行完，但是这里有个问题就是子线程没有被主线程终止，是自动执行完的，这样其实有些时候不安全

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);

    qDebug() << "thread start";

    mythread thread1; // 创建线程 thread1
    thread1.setObjectName("thread1"); // 将字符串传入线程1
    thread1.start(); // 启动线程

    qDebug() << "thread end";

    thread1.terminate(); // 线程1终止函数

    return a.exec();
}

```

```

thread start
thread end
"thread1" = 0

```

加入了线程终止函数导致子线程才开始执行，主线程就执行完了，因而导致主线程的 terminate 函数先执行，关闭掉了子线程

增加子线程终止函数

那么问题来了，如果子线程申请了堆空间，或者打开了某些功能，但是你主线程关闭了子线程，导致子线程还没有执行到释放堆空间和关闭某些功能的程序段就自动退出了。这样是不是就会引起设备故障呢？所以我们要禁止在工程开发中用 terminate() 函数

我们要在使用的线程类中增加标志位来判别线程状态

```
class mythread : public QThread
{
protected:
    volatile bool flag; //类成员函数无法定义变量的时候初始化
    void run()
    {
        for(int i = 0; !flag && i < 10; i++)
        {
            qDebug ()<<objectName ()<<"="<<i; //将setObjectName()方法名与对象名连接起来
            msleep(500); //延时500毫秒
        }
        qDebug ()<<"run end";
    }

public:
    mythread() //建立构造函数初始化标志位
    {
        flag = false;
    }

    void stop()
    {
        flag = true;
    }
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qDebug ()<<"thread start";

    mythread thread1; //创建线程thread1
    thread1.setObjectName("thread1"); //将字符串传入线程1
    thread1.start(); //启动线程

    qDebug ()<<"thread end";

    for(int i = 0; i<10000; i++)
    {
        for(int i=0; i<100000; i++);
    }
    thread1.stop();
}

return a.exec();
```

如果 flag 为 false 就继续循环

增加标志位，在创建对象的时候就自动执行构造函数来初始化 flag 变量

这里延时让主程序关闭子线程不要这么快，可以观察子线程现象

执行对象的 stop 函数让线程跳出循环

```
thread start
thread end
"thread1" = 0
"thread1" = 1
"thread1" = 2
"thread1" = 3
"thread1" = 4
run end
```

你看，虽然我没有执行完子线程循环就退出了，但是子线程循环结束后还执行了 run end 这条语句，所以这就证明了子线程不是 terminate 强制结束的。而是要执行完整个 run 函数才结束，那么在 run 后面的程序段就可以写释放内存，关闭功能的代码。

## 多线程同步有什么用？

```
void run1()
{
    qDebug() << "run 11111 start";
    qDebug() << "run 11111 begin";

    for(int i=0;i<2000;i++) //延时函数
    {
        for(int i=0;i<100000;i++);
    }
}

void run2()
{
    qDebug() << "run 22222 start";
    qDebug() << "run 22222 begin";

    for(int i=0;i<2000;i++) //延时函数
    {
        for(int i=0;i<100000;i++);
    }
}

void run3()
{
    qDebug() << "run 33333 start";
    qDebug() << "run 33333 begin";

    for(int i=0;i<2000;i++) //延时函数
    {
        for(int i=0;i<100000;i++);
    }
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    run1();
    run2();
    run3();

    return a.exec();
}
```

建立 3 个功能函数，  
run1(),run2(),run3()

```
"xxxx1111" : run() begin
"xxxx1111" : run() end
"xxxx2222" : run() begin
"xxxx2222" : run() end
"xxxx3333" : run() begin
"xxxx3333" : run() end
```

单线程情况下是按照顺序  
run1,run2,run3 执行完

```

void run1()
{
    qDebug() << "run 11111 start";
    qDebug() << "run 11111 begin";
}
class xzz1 : public QThread
{
public:
    void run()
    {
        run1();
        msleep(500); //毫秒延时函数
    }
    xzz1()
    {

    }
};

```

功能类 1

```

void run2()
{
    qDebug() << "run 22222 start";
    qDebug() << "run 22222 begin";
}
class xzz2 : public QThread
{
public:
    void run()
    {
        run2();
        msleep(500); //毫秒延时函数
    }
    xzz2()
    {

    }
};

```

功能类 2

```

void run3()
{
    qDebug() << "run 33333 start";
    qDebug() << "run 33333 begin";
}
class xzz3 : public QThread
{
public:
    void run()
    {
        run3();
        msleep(500); //毫秒延时函数
    }
    xzz3()
    {

    }
};

```

功能类 3

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

```

xzz1 x1; //run1() 功能函数的类  
xzz2 x2; //run2() 功能函数的类  
xzz3 x3; //run3() 功能函数的类

```

/*如果这是3个不同功能的类*/
x1.start();
x2.start();
x3.start();
return a.exec();
}

```

主函数启动这 3 个  
不同功能的类和里  
面的函数

```

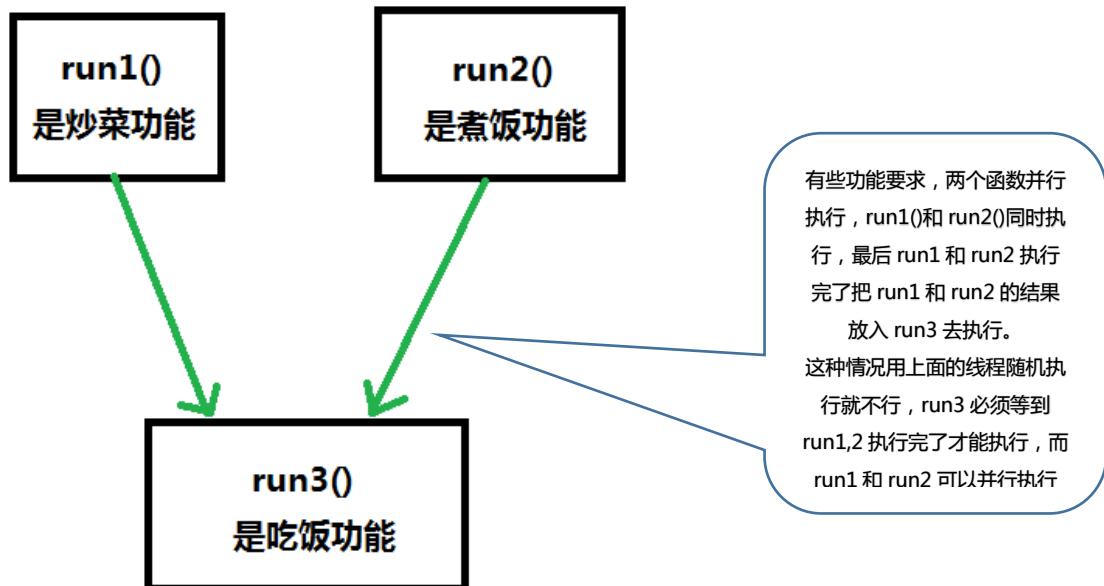
run 11111 start
run 22222 start
run 22222 begin
run 33333 start
run 11111 begin
run 33333 begin

```

我们上一节就知道了，线程间的功能执行是并行的，  
但是哪个线程先执行和哪个线程后执行是随机的

如果这 3 个功能类里面的函数各自执行各自的，相互没有数据往来，函数相互没有功能依赖，那么这样做没有问题。

如果这3个功能类的函数相互之间有数据交互或者函数之间必须满足run1()run2()run3()执行顺序，那么就需要线程同步了。如果是这样我直接按照顺序执行3个类的函数就是了，为什么还需要线程同步呢？



比如吃饭功能，在没有饭的情况下不能先执行吃饭函数(run3)，必须先等饭做好了才能吃。然后菜和饭是可以同时做的。所以炒菜和煮饭是同时执行最后执行完了才允许执行吃饭功能。

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    xzz1 x1; //run1() 功能函数的类
    xzz2 x2; //run2() 功能函数的类
    xzz3 x3; //run2() 功能函数的类

    /*如果这是3个不同功能的类*/
    x1.start();
    x2.start();

    x1.wait();
    x2.wait();

    x3.start();

    return a.exec();
}
```

x1.wait 表示必须 x1 执行完了才能向下执行，x2.wait 也是一样。

D:\QT\bin\qtcreator\_pr  
run 22222 start  
run 11111 start  
run 22222 begin  
run 11111 begin  
run 33333 start  
run 33333 begin

用wait函数等待线程执行完，这样不管你是先执行run1还是run2，反正run1和run2执行完了才能执行run3。

## 多线程的互斥，也就是两个线程或者多个线程同时读写一个变量的问题

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

static QString q_store;//两个线程要同时访问的变量
```

q\_store 全局变量

```
class produter : public QThread //写数据进变量的线程
{
protected:
    void run()
    {
        int count = 0;
        while(true)
        {
            q_store.append(QString::number((count++)%10));
            qDebug()<<objectName() << ":" + q_store;
            msleep(1);
        }
    }
};

class customer : public QThread//从变量中读取出数据的线程
{
protected:
    void run()
    {
        while(true)
        {
            if(q_store != "")
            {
                q_store.remove(0,1);
                qDebug()<<objectName() << ":" + q_store;
            }
            msleep(1);
        }
    }
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    produter pro;
    customer cus;

    pro.setObjectName("pro");
    cus.setObjectName("cus");

    pro.start(); //启动写变量线程
    cus.start(); //启动读变量线程
    return a.exec();
}
```

程序在正常运行一段时间后崩溃了。

这是因为有时候写线程在向变量写数据的时候，读线程正好没有读变量中的数据。一旦写线程向变量赋值写数据，这时候读线程又在读相同变量的数据就会发送崩溃

Produter 线程写数  
据进变量

q\_store 全局变量

A yellow box labeled "Produter 线程写数 据进变量" has a yellow arrow pointing to an orange box labeled "q\_store 全局变量".

customer 线程读取变量数据

q\_store 全局变量

A blue box labeled "customer 线程读取变量数据" has a blue arrow pointing to an orange box labeled "q\_store 全局变量".

The screenshot shows two windows. The top window is titled 'D:\QT\bin\qtcreator\_process\_stub.exe' and lists repeated entries: 'pro' : 6789. The bottom window is titled 'mutual.exe' and displays a message: 'mutual.exe 已停止工作' (The program has stopped working) with the sub-message: '出现了一个问题，导致程序停止正常工作。请关闭。' (A problem occurred, causing the program to stop working normally. Please close.) and a 'Close Program' button.

11 / 51

所以我们 QT 多线程和 UCOSII 系统，linux 系统一样，都发明了互斥锁这个东西。



这就是多线程同时访问一个变量造成的互斥问题。

我们要给线程加入线程锁，也就是所谓的互斥锁

```
QMutex mutex;  
  
mutex.lock();  
  
// do something with critical resource  
  
mutex.unlock();
```

这里是要执行读写的全局变量，在读写之前先上锁

这就是线程锁的用法

```
#include <QDebug>  
#include <QMutex>/> //增加线程锁，也就是我们说的互斥锁  
  
static QString q_store; //两个线程要同时访问的变量  
static QMutex x_mutex; //定义一个全局的互斥锁，因为是两个线程在用同一个锁  
  
class produter : public QThread //写数据进变量的线程  
{  
protected:  
    void run()  
    {  
        int count = 0;  
        while(true)  
        {  
            x_mutex.lock(); //上锁  
            q_store.append(QString::number((count++)%10));  
            qDebug()<<objectName() << ":" + q_store;  
            x_mutex.unlock(); //解锁  
            msleep(1);  
        }  
    }  
};  
  
class customer : public QThread //从变量中读取出数据的线程  
{  
protected:  
    void run()  
    {  
        while(true)  
        {  
            x_mutex.lock(); //上锁  
            if(q_store != "")  
            {  
                q_store.remove(0,1);  
                qDebug()<<objectName() << ":" + q_store;  
            }  
            x_mutex.unlock(); //解锁  
            msleep(1);  
        }  
    }  
};
```

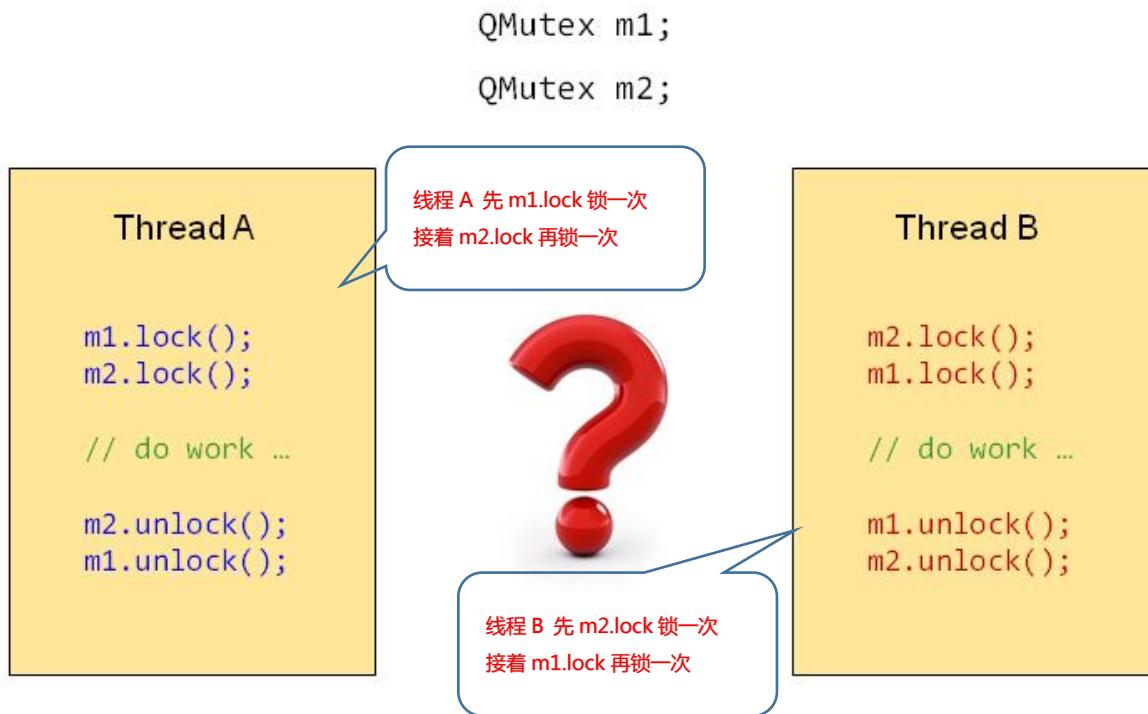
在线程写 q\_store 变量之前先判断是否有其它线程调用了 x\_mutex  
这个变量的 lock 函数，如果其它线程调用了，那么该线程就在这里  
阻塞等待，如果其它线程没有调用那么该线程就直接上锁 lock

这是告诉其它使用 x\_mutex 变量的线程，我这里已经解锁了

这里有个要注意的地方，为什么我的 x\_mutex 是全局的呢？这是因为这两个线程是读写的同一个 q\_store 变量，那么就要用同一个互斥锁（线程锁）变量才能让两个线程知道锁的是同一个变量



## 线程锁产生死锁，及解决方法



---

```
#include <QMutex> //增加线程锁，也就是我们说的互斥锁
```

```
QMutex g_mutex_1; // 定义2个线程锁
QMutex g_mutex_2; // 定义2个线程锁

class ThreadA : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            g_mutex_1.lock();
            qDebug() << objectName() << "ThreadA1";
            g_mutex_2.lock();

            qDebug() << objectName() << "A do work ...";

            g_mutex_2.unlock();
            g_mutex_1.unlock();
            sleep(1);
        }
    }
};
```

假如线程 A 先启动,那么线程 A 会先锁住 g\_mutex\_1 , 这时候如果线程调度同时启动了线程 B

```

class ThreadB : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            g_mutex_2.lock();
            qDebug() << objectName() << "ThreadB1";
            g_mutex_1.lock();

            qDebug() << objectName() << "B do work ...";
            g_mutex_1.unlock();
            g_mutex_2.unlock();

            sleep(1);
        }
    }
};

这种情况就是所谓的死锁

```

线程 B 会把 g\_mutex\_2 锁住

然后又想锁 g\_mutex\_1，但是发现这个 g\_mutex\_1 锁已经被线程 A 锁住了，那么线程 B 只有在这里等待，等待线程 A 解锁

这个时候线程 A 又想锁 g\_mutex\_2，发现 mutex2 倍  
线程 B 锁住，所以线程 A 只有等待

```

void run()
{
    while( true )
    {
        g_mutex_1.lock();
        qDebug() << objectName() << "ThreadA1";
        g_mutex_2.lock();

        qDebug() << objectName() << "A do v";
    }
}

```

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

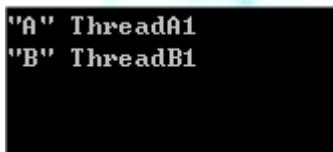
    ThreadA A;
    ThreadB B;

    A.setObjectName("A");
    B.setObjectName("B");

    A.start();
    B.start();

    return a.exec();
}

```



你看两个线程执行一下就不执行了，因为线程 A 和 B 相互锁住了。

我们可以让锁按照 1 , 2 , 3 , 4 这种顺序的方式加锁

```
class ThreadA : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            g_mutex_1.lock();
            qDebug() << objectName() << "ThreadA1";
            g_mutex_2.lock();

            qDebug() << objectName() << "A do work ...";

            g_mutex_2.unlock();
            g_mutex_1.unlock();
            sleep(1);
        }
    }
};
```

我们线程 A 的锁是按照顺序加锁的，先锁 1，然后在锁 2

解锁也是按照倒序解锁的，先解锁最后一个加锁的锁 2，然后  
再解锁倒数第二个加锁的锁 1，以此类推....

```
class ThreadB : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            g_mutex_2.lock();
            qDebug() << objectName() << "ThreadB1";
            g_mutex_1.lock();

            qDebug() << objectName() << "B do work ...";
            g_mutex_1.unlock();
            g_mutex_2.unlock();

            sleep(1);
        }
    }
};
```

但是我们发现线程 B 的加锁是乱的 和线程 A 不一致，先加锁了 2 在加锁 1，而且解锁也是没按照倒序的方法解锁，而是乱选择锁编号解锁

```
while( true )
{
    g_mutex_1.lock();
    qDebug() << objectName() << "ThreadB1";
    g_mutex_2.lock();

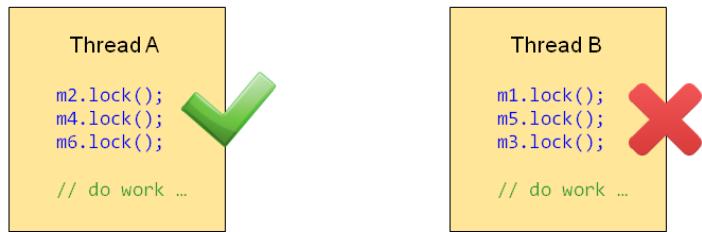
    qDebug() << objectName() << "B do work ...";
    g_mutex_2.unlock();
    g_mutex_1.unlock();

    sleep(1);
}
```

我们将线程 B 解锁顺序修改成线程 A 的  
解锁顺序，按照顺序解锁就可以了

```
"A" ThreadA1
"A" A do work ...
"B" ThreadB1
"B" B do work ...
"B" ThreadB1
"B" B do work ...
"A" ThreadA1
"A" A do work ...
"A" ThreadA1
"A" A do work ...
"B" ThreadB1
"B" B do work ...
```

## 系统中的每个线程按照严格递增的次序请求资源



就是上面这种加锁解锁方法可以用在多个

全局变量上，有多少个全局变量就定义多少个锁，按照顺序加锁解锁。

```
unsigned char a1;
unsigned char b2;
unsigned char c3;
```

我定义了3个全局变量，每个全局变量给了一把锁

```
QMutex g_mutex_1;//定义1个线程锁
QMutex g_mutex_2;//定义2个线程锁
QMutex g_mutex_3;//定义3个线程锁
```

```
class ThreadA : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread A";
            g_mutex_1.lock();
            a1 = 10;
            g_mutex_1.unlock();

            g_mutex_2.lock();
            b2 = 20;
            g_mutex_2.unlock();

            g_mutex_3.lock();
            c3 = 30;
            g_mutex_3.unlock();
        }
    }
}
```

按照全局变量的赋值顺序，锁也按照顺序加，一个锁管一个全局变量

```
class ThreadB : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread B";
            g_mutex_1.lock();
            qDebug() << "a1 = " << a1;
            g_mutex_1.unlock();

            g_mutex_2.lock();
            qDebug() << "b2 = " << b2;
            g_mutex_2.unlock();

            g_mutex_3.lock();
            qDebug() << "c3 = " << c3;
            g_mutex_3.unlock();
        }
    }
}
```

线程 B 要读取全局变量的内容，那么加锁解锁顺序也必须按照线程 A 的加锁解锁顺序来，因为这几个全局变量是先按照 A 线程的方式锁的。

```
Thread A
Thread B
a1 = 10
b2 = 20
c3 = 30
Thread A
Thread B
a1 = 10
b2 = 20
c3 = 30
Thread A
Thread B
a1 = 10
b2 = 20
c3 = 30
```

还有一种方法就是所有多线程之间使用的多个全局变量只用一把锁

```
unsigned char a1;
unsigned char b2;
unsigned char c3;
```

我定义的3个全局变量只用一把锁

```
QMutex g_mutex_1; // 定义1个线程锁
```

```
class ThreadA : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread A";
            g_mutex_1.lock();
            a1 = 10;
            b2 = 20;
            c3 = 30;
            g_mutex_1.unlock();
        }
    }
}
```

线程A把3个全局变量锁起来全部赋值

```
class ThreadB : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread B";

            g_mutex_1.lock();
            qDebug() << "a1 = " << a1;
            qDebug() << "b2 = " << b2;
            qDebug() << "c3 = " << c3;
            g_mutex_1.unlock();

            sleep(1);
        }
    }
}
```

线程B把三个全局变量锁起来，读取三个全局变量数据

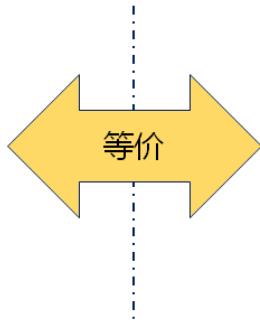
```
Thread A
Thread B
a1 = 10
b2 = 20
c3 = 30
Thread A
Thread B
a1 = 10
b2 = 20
c3 = 30
Thread B
a1 = 10
b2 = 20
c3 = 30
Thread A
```

这种多个全局变量用一把锁的方式也可以解决线程死锁问题，但是在服务器多线程编程环境下，程序运行速度就变慢了，而且效率很低。但是在客户端多线程情况下还是可以接受的

## 信号量也是一种特殊的线程锁，信号量使用 QS\_semaphore 使用示例

```
QS_semaphore sem(1);
sem.acquire();
// do something with
// critical resource
sem.release();

QS_mutex mutex;
mutex.lock();
// do something with
// critical resource
mutex.unlock();
```



- QS\_semaphore 对象中维护了一个整型值
- acquire() 使得该值减 1, release() 使得该值加 1
- 当该值为 0 时, acquire() 函数将阻塞当前线程

```
#include <QS_semaphore>/信号量使用

unsigned char a1;

QS_semaphore sem(1); // 定义1个信号量初始值是1
我们定义了一个信号量

class ThreadA : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread A";
            sem.acquire();
            // 获取线程锁看看sem信号量里面的值是不是0那么sem变量减1,如果是0线程A就卡死在这里
            a1 = 10;
            sem.release(); // 释放信号量让sem加1

            sleep(1);
        }
    }
};

class ThreadB : public QThread
{
protected:
    void run()
    {
        while( true )
        {
            qDebug() << "Thread B";
            sem.acquire();
            // 获取线程锁看看sem信号量里面的值是不是0那么sem变量减1,如果是0线程B就卡死在这里
            qDebug() << "a1 = " << a1;
            sem.release(); // 释放信号量让sem加1
            sleep(1);
        }
    }
};

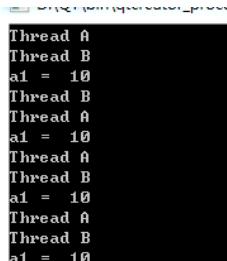
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    ThreadA A;
    ThreadB B;

    A.setObjectName("A");
    B.setObjectName("B");

    A.start();
    B.start();

    return a.exec();
}
```



信号量使用方法和互斥锁类似

信号量使用方法和互斥锁类似

程序运行正常，那信号量到底有什么用？

如果是上一页这样使用信号量，我不知道直接使用互斥锁啊，我拿信号量有什么用？

下面看看信号量真正的应用场景



假如:定义了5个箱  
子用来放物品



工厂A:依次向5  
个箱子放物品

工厂B:在A工厂向箱  
子里面放东西的同  
时，工厂B又在取箱  
子里面的东西

解决方案:

- 1.这时候我定义一个数组变量代表箱子
- 2.给A工厂创建线程A，B工厂创建线程B
- 3.因为箱子里面是5个物品，所以我用5个信号量来管理这5个物品，每个物品对应一个信号量

```
#include <QSSemaphore> //信号量使用
```

```
unsigned char buffer[5] = {0}; //箱子
```

```
QSSemaphore A(5); //工厂A信号量,初始值是5  
QSSemaphore B(0); //工厂B信号量,初始值是0
```

```
class ThreadA : public QThread  
{  
protected:  
    void run()  
    {  
        while( true )  
        {  
            A.acquire(); //让A信号量减1, 现在A信号量值就是4了  
            for(int i = 0; i < 5; i++)  
            {  
                buffer[i] = i;  
                qDebug() << "Thread A put in"; //工厂A向箱子依次放物品  
            }  
            B.release(); //你看我加1的不是A信号量,而是B信号量  
            sleep(1);  
        }  
    }  
}  
  
class ThreadB : public QThread  
{  
protected:  
    void run()  
    {  
        while( true )  
        {  
            B.acquire(); //判断B信号量释放为1的方式来判断A工厂是否向箱子放入了物品  
            for(int i = 0; i < 5; i++)  
            {  
                qDebug() << "buffer = " << buffer[i]; //B工厂同时获取物品  
            }  
            A.release(); //获取了A放入的5个物品后才能释放A的信号量让A加回5  
            sleep(1);  
        }  
    }  
};
```

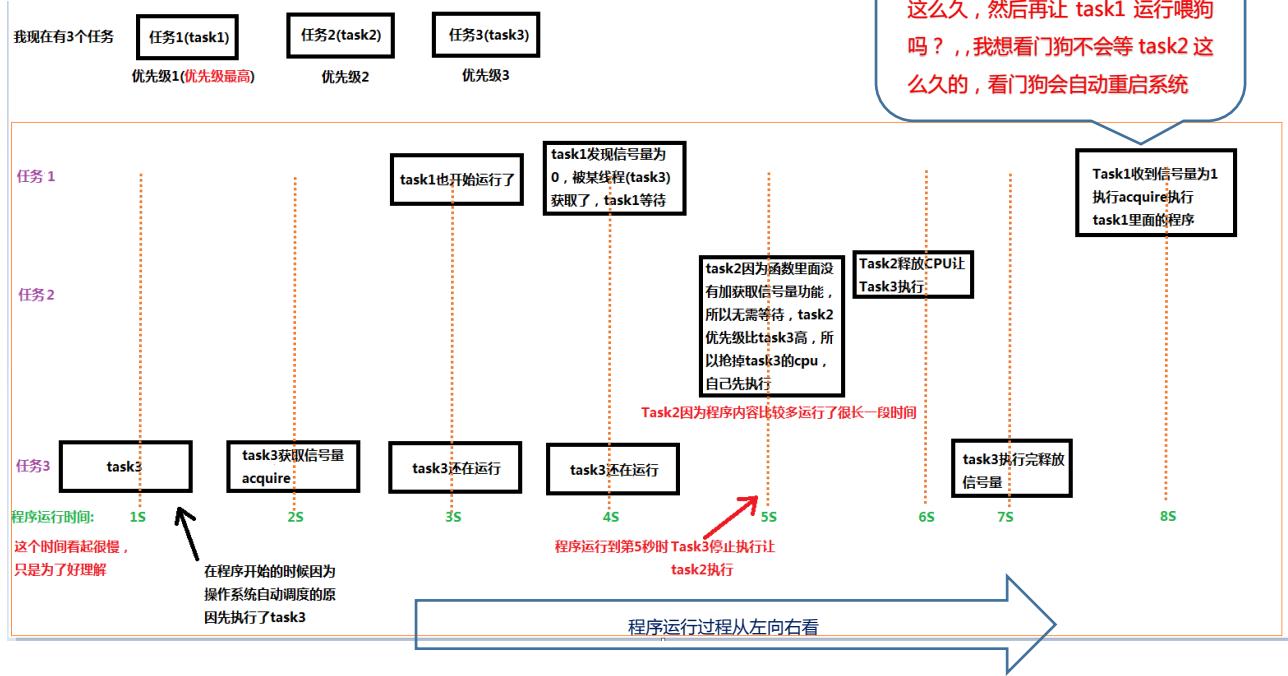
这是因为在多线程情况下，A 工厂在向箱子放物品的时候，  
B 工厂会去拿箱子的物品。如果我这里 B 信号量为 0，就表  
示 A 工厂还没有向箱子里面放物品时，B 线程开始执行了，  
那么下面的 B 工厂发现箱子没有物品，线程就会等待。

```
buffer = 0  
Thread A put in  
buffer = 0  
buffer = 1  
buffer = 2  
buffer = 3  
buffer = 4  
Thread A put in  
buffer = 0  
buffer = 1  
buffer = 2  
buffer = 3  
buffer = 4
```

启动 A 工厂和 B 工厂线程，执行没有问题。

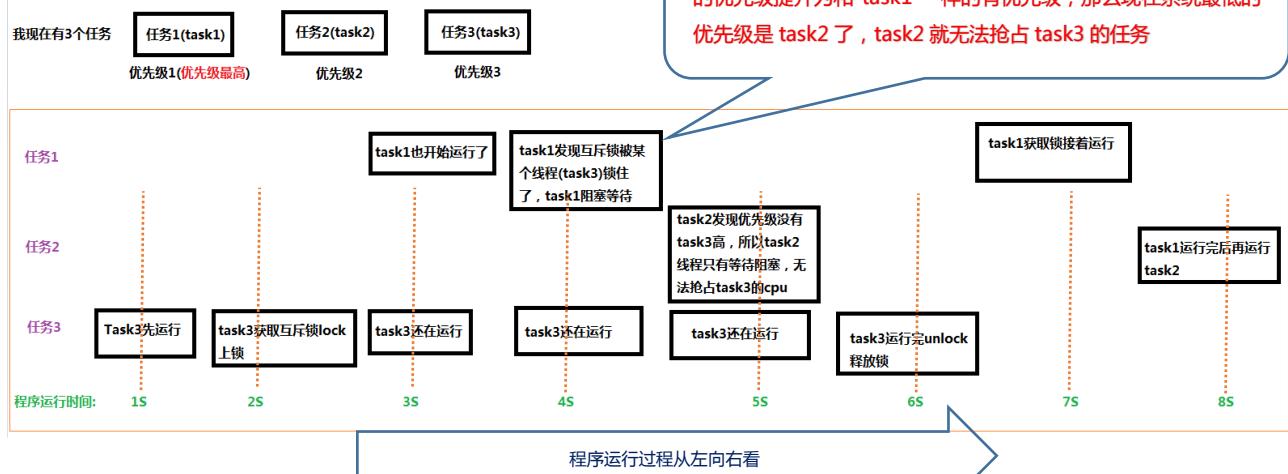
## 信号量和互斥锁区别

信号量有一个优先级翻转问题



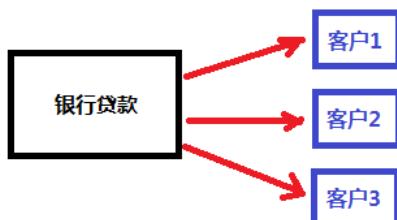
信号量只阻塞执行到信号量函数的线程，不会去阻止没有执行信号量函数的线程，所以该线程抢占其它优先级低的线程是可以的。所以信号量使用场景要自己衡量。

## 互斥锁解决了信号量翻转问题



这就是互斥锁的好处，一旦线程上锁，其它没有执行互斥锁的线程，就是优先级再高也无法打断优先级低的线程。这样 task1 就可以正常喂狗。

## 用银行家算法来练习多线程和互斥锁操作



银行家算法就是银行不是同一时间一次性同时贷款给3个客户

银行家算法案例



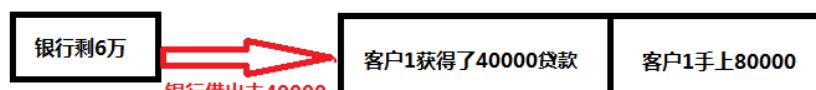
现在银行只有10万，怎么操作呢？



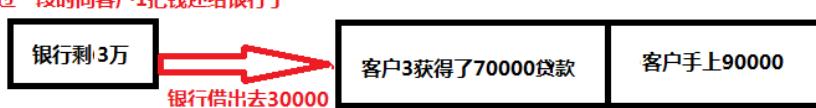
这时候银行先贷款给资金要求最少的客户，然后让其它两个客户等待



过一段时间客户2和其它客户完成交易，把钱还给银行了



过一段时间客户1把钱还给银行了



所以银行家算法也是串行操作的。

## 设计思路:

- 1.首先每个客户就是一个线程，所以要创建3个线程
- 2.银行也是一个线程，所以客户加上银行是4个线程
- 3.客户贷款成功后，执行还款，然后关闭还款成功客户的线程。

```
class customer : public QThread //创建客户类，1个客户就是一个线程
{
protected:
    int m_need; //客户需要资金多少
    volatile int m_current;//客户手上的资金有多少
    QMutex m_mutex;

    void run()
    {
        bool condition = false;
        qDebug()<<objectName()<<"apply money";//客户线程向银行申请借钱

        do
        {
            m_mutex.lock(); //银行线程修改m_current值的时候，要加线程锁这样才安全
            condition = (m_current < m_need); //如果当前客户手上的资金 < 需要的资金 就继续循环
            m_mutex.unlock();
            msleep(1); //继续等待1ms等待银行放款

        }while(condition);

        qDebug()<<objectName()<<"apply money";//客户线程向银行借到了钱
    }

public:
    customer(int current,int need)
    {
        m_current = current; //客户从银行借了多少钱
        m_need = need; //客户需要的资金总额是多少
    }

    void addmoney(int m) //银行给这个对象客户放款
    {
        m_mutex.lock();
        m_current += m;
        m_mutex.unlock();
    }

    int current()
    {
        int ret = 0;
        m_mutex.lock();
        ret = m_current; //返回现在客户手上的资金
        m_mutex.unlock();

        return ret;
    }

    int need()
    {
        return m_need; //返回当前客户手上的资金总额
    }

    int backmoney() //还钱的函数
    {
        int ret = 0;

        m_mutex.lock();
        ret = m_current;
        m_current = 0;
        m_mutex.unlock();

        return ret;
    }
};
```



银行线程发现某个客户线程的交易已经完成了，就会执行客户线程里面的backmoney 函数要求客户还钱，backmoney 返回 ret 变量就证明还钱了

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    customer cli1(40000,80000); //创建客户1,客户1手上有40000现金,再借40000总和就是80000
    customer cli2(20000,30000); //创建客户2,客户2手上有20000现金,再借10000总和就是30000
    customer cli3(20000,90000); //创建客户3,客户3手上有20000现金,再借70000总和就是90000

    /*让3个借款客户的线程跑起来*/
    cli1.setObjectName("cli1");
    cli2.setObjectName("cli2");
    cli3.setObjectName("cli3");
    cli1.start();
    cli2.start();
    cli3.start();
    return a.exec();
}

```

```

"cli1" apply money
"cli2" apply money
"cli3" apply money

```



程序执行到这里就不向下执行了，等待银行线程启动放款

## QT 多线程使用信号与槽

线程类是自带信号发送功能的

### - 关键信号：

- `void started()`  
✓ 线程开始运行时发射该信号
- `void finished()`  
✓ 线程完成运行时发射该信号
- `void terminated()`  
✓ 线程被异常终止时发射该信号

这几个函数就是信号发送函数，在用 `QObject::connect` 函数时填入。

```
#ifndef TESTTHREAD_H
#define TESTTHREAD_H

#include <QThread>

class Testthread : public QThread
{
    Q_OBJECT
protected:
    void run(); //线程运行的函数run
public:
    explicit Testthread(QObject *parent = 0);
signals:
    void testsignal(); //添加信号
protected slots:
    void testsolt(); //添加槽函数
};

#endif // TESTTHREAD_H
```

线程运行函数来模拟线程启动和结束

线程开始要发送信号，所以定义个发送信号  
函数在 connect 中占发送信号的位置

一旦信号发送了，就有对应的槽函数来接收

```

#include "Testthread.h"
#include <QDebug>
Testthread::Testthread(QObject *parent) :
    QThread(parent)
{
    connect(this, SIGNAL(testsignal()), this, SLOT(testsolt()));
    //发送本身这个类里面的信号函数到本身这个类里面的槽函数
}

```

头文件定义的 testsignal 函数填入 connect , 然后槽函数也填入 connect

```

void Testthread::run()
{
    qDebug() << "test run";

    int i = 5;
    while(i--)
    {
        qDebug() << "i = " << i << endl;
        sleep(1);
    }
    emit testsignal();
    //emit发送信号,指定发送哪个函数做信号,这里是testsignal
}

```

线程执行完了如果想通知别人,就用 emit 关键字发送信号 testsignal 这个空函数

```

void Testthread::testsolt()
{
    qDebug() << "testsolt";
}

```

线程发送信号后对应 connect 连接的槽函数就会响应

```

#include <QtCore/QCoreApplication>
#include "Testthread.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Testthread t;
    t.start();
    return a.exec();
}

```

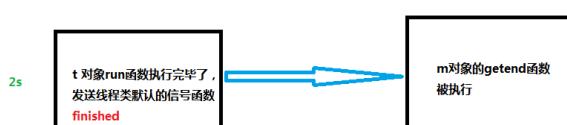
```

test run
i = 4
i = 3
i = 2
i = 1
i = 0
testsolt

```

线程循环  
执行完发  
送了信号,  
导致槽函  
数打印了

下面用另外一个不是线程的类创建的对象,来接收线程类创建的线程对象发送过来的信号。



沿用上页的 Testthread 类函数和对象 t

下面创建一个普通类来接收 Testthread 对象发过来的信号

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <QObject>

class myObject : public QObject
{
    Q_OBJECT
public:
    explicit myObject(QObject *parent = 0);

signals:

protected slots:
    void getstart(); //接受另外一个类TestThread创建的对象里面线程运行开始的信号
    void getend(); //接受另外一个类TestThread创建的对象里面线程运行完成的信号
    void getterminate(); //接受另外一个类TestThread创建的对象里面线程运行异常终止的信号
};

#endif // MYOBJECT_H
#include "myobject.h"
#include <QDebug>

myObject::myObject(QObject *parent) :
    QObject(parent)
{ }

void myObject::getstart()
{
    qDebug() << "getstart";
}

void myObject::getend()
{
    qDebug() << "getend";
}

void myObject::getterminate()
{
    qDebug() << "getterminate";
}

#include <QtCore/QCoreApplication>
#include <QThread> //线程自带started finished terminate信号发送函数
#include "Testthread.h"
#include "myobject.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Testthread t;
    myObject m;

    QObject::connect(&t, SIGNAL(started()), &m, SLOT(getstart()));
    QObject::connect(&t, SIGNAL(finished()), &m, SLOT(getend()));
    QObject::connect(&t, SIGNAL(terminated()), &m, SLOT(getterminate()));
    t.start();
    return a.exec();
}
```

普通类定义的三个槽函数用来响应 testthread 类对象发过来的线程信号，记得用 connect 链接了才有效

实现三个槽函数

用 connect 将 testthread 对象 t 发送的信号和 myobject 普通类 m 实现的槽函数链接起来

记住这里填的是线程默认信号函数

普通类 m 收到了 t 对象的线程启动信号

普通类 m 收到了 t 对象的线程结束信号

所以普通类定义的槽函数可以收到其它线程类对象发送过来的信号，只要你用 connect 链接了。

如果程序中有多个线程，槽函数是在哪个

线程中执行的？

答案是：

- 进程中存在栈空间的概念（区别于栈数据结构）
- 栈空间专用于函数调用（保存函数参数，局部变量，等）
- 线程拥有独立的栈空间（可调用其它函数）

只要函数体中没有访问临界资源的代码，同一个函数

可以被多个线程同时调用，且不会产生任何副作用！

但是线程类创建的槽函数不一定在该线程类中执行

```
1 int main(int argc, char *argv[])
2 {
3     QCOREAPPLICATION a(argc, argv);
4
5     qDebug() << "main id = " << QThread::currentThreadId();
6     //QThread::currentThreadId() 函数用来打印线程ID
7
8     Testthread t;
9     myObject m;
10
11    QObject::connect(&t, SIGNAL(started()), &m, SLOT(getstart()));
12    QObject::connect(&t, SIGNAL(finished()), &m, SLOT(getend()));
13    QObject::connect(&t, SIGNAL(terminated()), &m, SLOT(getterminate()));
14    t.start();
15    return a.exec();
16 }
17
18 #include "myobject.h"
19 #include <QDebug>
20 #include <QThread>
21 myObject::myObject(QObject *parent) :
22     QObject(parent)
23 {
24 }
25
26 void myObject::getstart()
27 {
28     qDebug() << "getstart";
29     qDebug() << "myObject::getstart slot ID = " << QThread::currentThreadId();
30 }
31
32 void myObject::getend()
33 {
34     qDebug() << "getend";
35     qDebug() << "myObject::getend slot ID = " << QThread::currentThreadId();
36 }
37
38 ...
39
40 void Testthread::testsolt()
41 {
42     qDebug() << "testsolt";
43     qDebug() << "Testthread::testsolt ID= " << QThread::currentThreadId(); //打印槽函数线程ID
44 }
```

打印主函数的线程 ID

打印 myObject 类函数里面的线程 ID

打印另外一个主函数创建出来的独立的线程类的线程 ID

```
main id = 0x2c3c
test run
i = 4
getstart
myObject::getstart slot ID = 0x2c3c
i = 3
i = 2
i = 1
i = 0
testsolt
Testthread::testsolt ID= 0x2c3c
getend
myObject::getend slot ID = 0x2c3c
```

主线程 ID 是 0x2c3c

其它类创建的槽函数也是主线程的线程 ID

重点是独立创建的线程类，线程 ID 和主线程不一样，但是线程类自己创建的槽函数怎么和主线程 ID 是一样的？这个子线程创建的槽函数不应该是子线程类的 ID 吗。

## 线程类定义的槽函数怎么没有在自己的线程类里面执行？

### 第1问：对象依附于哪个线程？

答：默认情况下对象依附于创建自己对象的线程，比如我是主线程创建的对象 A，那么对象 A 就依附于主线程，但是对象 A 创建的线程有自己的线程 ID。

### 第2问：那么槽函数是在哪个线程执行呢？

答：槽函数是在创建自己槽函数的对象中执行，但是对象又是主线程创建的，对象依附于主线程，那么槽函数也就跟着对象混，槽函数也依附于主线程。如果对象是子线程创建的，那么槽函数就依附于子线程。

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();
    //QThread::currentThreadId() 函数用来打印线程ID

    Testthread t;
    myObject m;

    QObject::connect(&t, SIGNAL(started()), &m, SLOT(getstart()));
    QObject::connect(&t, SIGNAL(finished()), &m, SLOT(getend()));
    QObject::connect(&t, SIGNAL(terminated()), &m, SLOT(getterminate()));

    t.start();
    return a.exec();
}
```

比如主线程 main 函数创建了对象 t 和 m

那么 t 和 m 对象里面创建的槽函数就依附于主线程执行

那么我们能不能修改 t 和 m 对象的线程依附性，就是让 t 和 m 对象依附在其它线程。这样 t 和 m 的槽函数就在指定的线程中执行，而不是默认在主线程执行。

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();
    //QThread::currentThreadId() 函数用来打印线程ID

    Testthread t;
    myObject m;

    m.moveToThread(&t); //让m对象创建的三个槽函数在t对象的线程中执行

    QObject::connect(&t, SIGNAL(started()), &m, SLOT(getstart()));
    QObject::connect(&t, SIGNAL(finished()), &m, SLOT(getend()));
    QObject::connect(&t, SIGNAL(terminated()), &m, SLOT(getterminate()));

    t.start();
    return a.exec();
}
```

我们让 m 对象依附于线程类对象 t，这样槽函数就不在主线程执行，而在 t 线程

```
main id = 0x3688
test run
esthread::run ID= 0x37d0
i = 4
i = 3
i = 2
i = 1
i = 0
testsolt
Testthread::testsolt ID= 0x3688
```

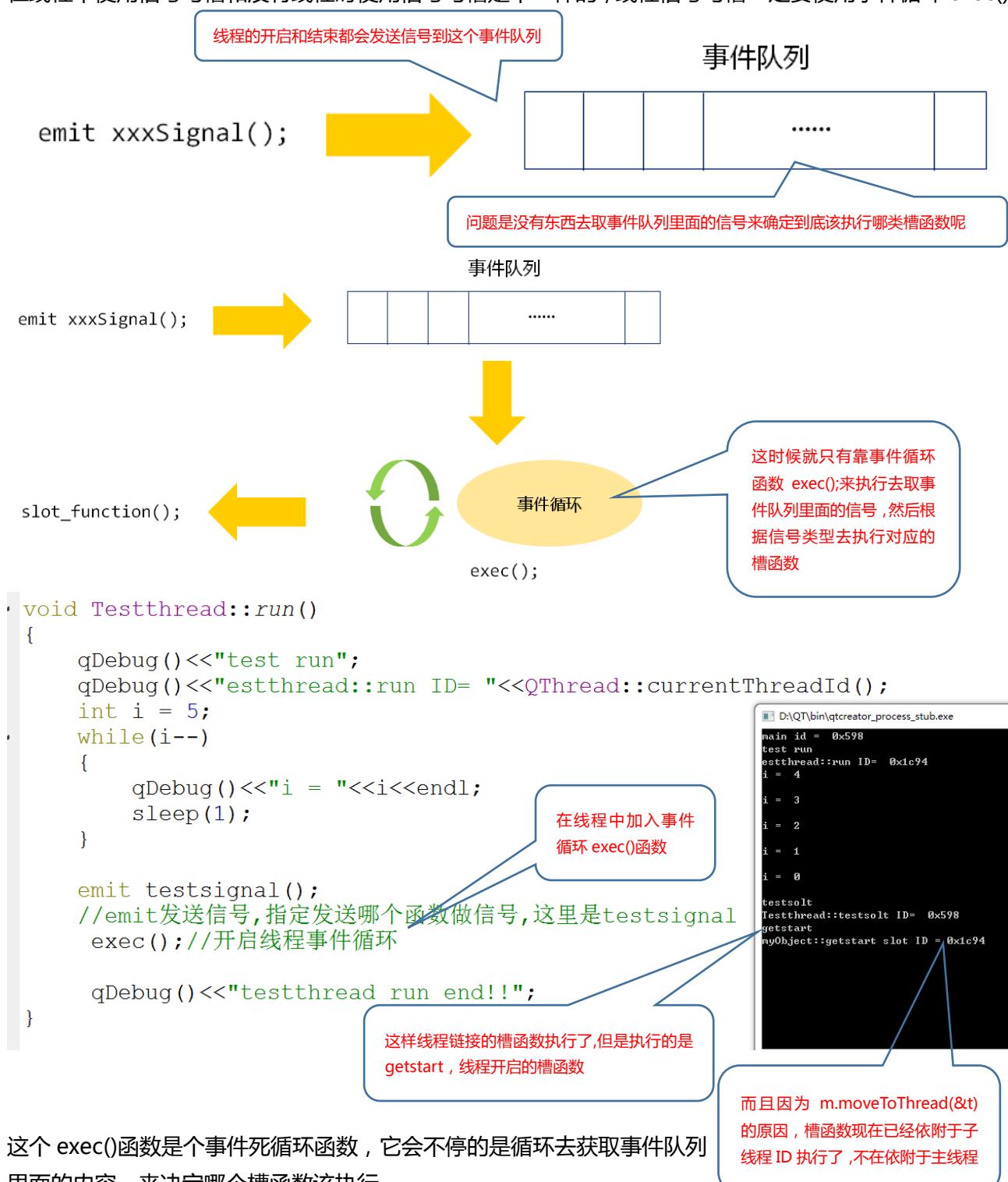
我们运行发现 t 线程槽函数执行了

但是另外三个 m 对象的槽函数没有执行，m 对象的槽函数是接收 t 线程开始信号和 t 线程结束信号的

这是为什么呢？m.moveToThread(&t)修改了线程依附关系就出事了吗？

这是因为线程中的事件循环函数没有启动

在线程中使用信号与槽和没有线程时使用信号与槽是不一样的，线程信号与槽一定要使用事件循环 exec()；



这个 `exec()` 函数是个事件死循环函数，它会不停的循环去获取事件队列里面的内容，来决定哪个槽函数该执行。

如何处理 `exec()` 事件死循环？

用 `quit` 函数来结束指定线程。

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();
    //QThread::currentThreadId() 函数用来打印线程ID

    Testthread t;
    myObject m;

    m.moveToThread(&t); //让m对象创建的三个槽函数在t对象的线程中执行

    QObject::connect(&t, SIGNAL(started()), &m, SLOT(getstart()));
    QObject::connect(&t, SIGNAL(finished()), &m, SLOT(getend()));
    QObject::connect(&t, SIGNAL(terminated()), &m, SLOT(getterminate()));

    t.start();

    t.moveToThread(&t); //让t对象依附于t线程
    t.wait(4000); //wait函数是阻塞在这里，等待该t线程执行多少毫秒后再向下执行剩余程序
    t.quit(); //quit函数是退出线程
    return a.exec();
}

main id = 0x22d4
getstart
myObject::getstart slot ID = 0xb4c
test run
estthread::run ID= 0xb4c
i = 4
i = 3
i = 2
i = 1
i = 0
testsolt
Testthread::testsolt ID= 0xb4c
testthread run end!!
getend
myObject::getend slot ID = 0xb4c

```

要用 quit 结束 t 对象的线程，必须让 t 对象依附于自己的线程，而不能让 t 对象依附于默认的主线程

加入 wait 等待 4 秒，然后执行 quit 线程结束

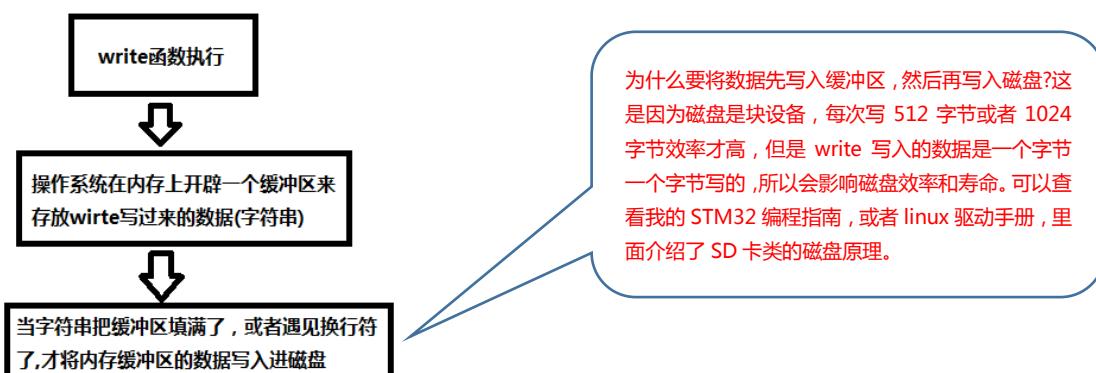
t 对象依附于 t 线程了

t 线程的 exec() 循环结束，执行线程结束槽函数。

但是我的 exec 必须靠主线程的 quit 来结束好麻烦

一般事件循环 exec 用来处理 IO 操作，比如 write 文件，read 文件。

exec 操作流程:先开启事件循环，在写 write 文件的时候，发起写信号函数，然后写 write 程序在子线程中进行 write 写文件操作系统是如何执行的?



## 案例：我们做一个向文件写数据的系统

```
#include <QObject>
#include <QFile> //文件操作函数头文件

class filewrite : public QObject
{
    Q_OBJECT
    QFile m_file;
public:
    explicit filewrite(QString file, QObject *parent = 0);
    bool open(); //打开文件
    void write(QString text); //向文件写数据
    void close(); //关闭文件

signals:

public slots:

};

#include "filewrite.h"

filewrite::filewrite(QString file, QObject *parent) :
    QObject(parent), m_file(file)
{ }

bool filewrite::open()
{
    return m_file.open(QIODevice::WriteOnly | QIODevice::Text); //打开文件设置文件为只可写
}

void filewrite::write(QString text)
{
    m_file.write(text.toAscii()); //将String字符串转成Ascii码
    m_file.flush(); //我们写完文件后在没有关闭文件的情况下用flush将内存的字符串写入磁盘
}

void filewrite::close()
{
    m_file.close();
}
```

我们将打开文件，写文件数据，关闭文件用类和函数封装起来

```
#include "filewrite.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    qDebug() << "main pid = " << QThread::currentThread();

    filewrite xwriter("E:/QTtest/xzzwrite.txt");

    if (xwriter.open())
    {
        xwriter.write("写入数据到磁盘1111\r\n");
        xwriter.write("写入数据到磁盘2222\r\n");
        xwriter.write("写入数据到磁盘3333\r\n");
        xwriter.close();
    }
    return a.exec();
}
```



数据写入磁盘

这里就有个问题了，就算程序运行没有问题，但是在主线程每次向文件写数据都是用 write，导致 flush 函数执行，flush 函数是向磁盘写数据，所以每行 write 函数都执行得比较慢，这样就会影响主线程执行其它程序的效率

所以文件写入磁盘存储的程序需要做优化，下面就来优化

```
class filewrite : public QObject
{
    Q_OBJECT
    class writethread : public QThread
    {
        protected:
            void run();
    };

    QFile m_file;
    writethread m_writethread; //用内部类创建线程对象

public:
    explicit filewrite(QString file, QObject *parent = 0);
    bool open(); //打开文件
    void Write(QString text); //向文件写数据
    void close(); //关闭文件

signals:
    void doWrite(QString text); //写文件的时候发送信号
    void doClose(); //关闭文件也要发送信号，因为close函数是个消耗时间的函数

protected slots:
    void Writeslot(QString text); //接受写文件信号的槽函数
    void Closeslot(); //关闭信号槽函数
};

#endif // FILEWRITE_H
```

我们在类里面创建一个内部类来做线程

我们把 write 和 close 里面实现的写文件和关闭文件这种费时间的函数提取出来，我们只在 write 和 close 里面发信号

这个信号就是给 write 和 close 函数使用的，这样 write 和 close 在主进程中执行得很快

要向磁盘写文件的时候，主进程的 write 和 close 函数发信号，然后导致 write 和 close 对应的槽函数执行

在槽函数里面实现向磁盘写数据的 write , flush , close

```
#include "filewrite.h"
#include <QDebug>
filewrite::filewrite(QString file, QObject *parent) :
    QObject(parent), m_file(file)
{
    connect(this, SIGNAL(doWrite(QString)), this, SLOT(Writeslot(QString)));
    connect(this, SIGNAL(doClose()), this, SLOT(Closeslot()));
    moveToThread(&m_writethread); //我们让槽函数在内部类的线程对象下执行
    m_writethread.start(); //启动线程
}

bool filewrite::open()
{
    return m_file.open(QIODevice::WriteOnly | QIODevice::Text); //打开文件设置文件为只可写
}

void filewrite::Write(QString text)
{
    emit doWrite(text);
    //将写文件的write从写文件到执行flush修改成发射信号，交给对应的writeslot槽函数去执行flush
}
```

连接写文件信号到槽函数里面，close 关闭文件也是

打开文件不耗时间，所以交给主进程 open 执行

写文件要耗时间，主进程 write 发信号交个槽函数去处理

```

void filewrite::close()
{
    emit doClose(); //将close函数修改成发射信号，交给对应的closeslot槽函数去执行close
}

void filewrite::writethread::run()
{
    qDebug() << "writethread run pid = " << QThread::currentThread();
    exec(); //线程run函数执行事件循环
    qDebug() << "writethread end pid = " << QThread::currentThread();
}

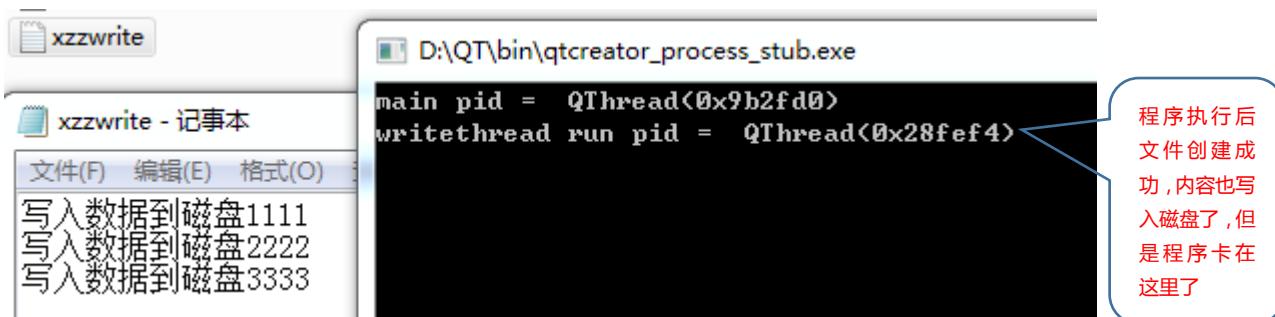
void filewrite::Writeslot(QString text)
{
    m_file.write(text.toAscii()); //将String字符串转成Ascii码
    m_file.flush(); //我们写完文件后在没有关闭文件的情况下用flush将内存的字符串写入磁盘
}

void filewrite::Closeslot()
{
    m_file.close();
}

```

我们知道线程中的槽函数执行，必须用 exec()去获取队列的信号来执行槽函数，这样槽函数才能执行。上一节讲过的，线程的信号与槽和普通信号与槽触发方式不一样。

这就是我们的槽函数，执行 write , flush , close 这些耗 CPU 时间的程序，为什么槽函数来执行消耗 CPU 时间的程序不会影响主线程效率呢？这是因为槽函数用 moveTo... 映射到内部内线程去执行了，所以槽函数和 main 主线程是并行的程序

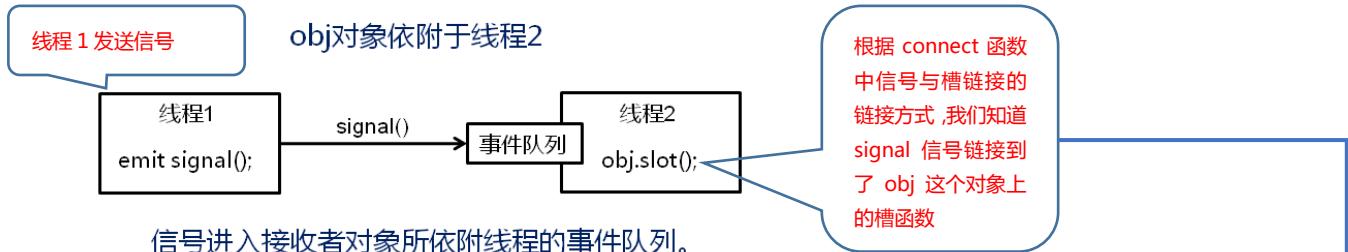


程序卡住是因为事件循环 exec() 执行了，不是事件循环执行了的话，你的槽函数是没法执行的，你的 txt 文件也不会创建出来。我们只有用 quit() 函数来结束 exec() 事件循环  
未完待续.....

QT 的 connect 链接信号到槽的函数有几个功能没有设置，我们一直用的是默认功能，connect 有五个形参，我们只用了四个。

我们用 emit 发信号的时候，我们知道信号发给了哪一个线程的事件队列吗？

- 每一个线程都有自己的事件队列
- 线程通过事件队列接收信号
- 信号在事件循环中被处理

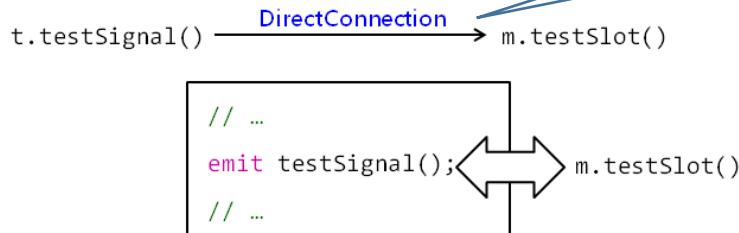


根据线程的依附性，obj 这个对象依附于线程 2，那么 obj 对象的槽函数也就属于线程 2，那么 signal 发送的信号就存放在线程 2 的时间队列里面。

所以要求线程 2 开启事件循环，因为有了 exec 才能去线程 2 的事件队列里面取出信号，执行 obj 定义的槽函数。

直接在发送信号的线程中调用槽函数，等价于槽函数的实时调用！

Connect 的 5 个链接方式, 我们先使用 DirectConnection 的链接方式，看看有什么后果



- Qt::DirectConnection (立即调用)
- Qt::QueuedConnection (异步调用)
- Qt::BlockingQueuedConnection (同步调用)
- Qt::AutoConnection (默认连接)
- Qt::UniqueConnection (单一连接)



下面我们来做实验

```
#ifndef TESTTHREAD_H
#define TESTTHREAD_H

#include <QThread>

class Testthread : public QThread
{
    Q_OBJECT
protected:
    void run(); //线程运行的函数run
public:
    explicit Testthread(QObject *parent = 0);

signals:
    void testsignal(); //添加信号
};

#endif // TESTTHREAD_H
```

创建发送信号类头文件

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <QObject>

class myObject : public QObject
{
    Q_OBJECT
public:
    explicit myObject(QObject *parent = 0);

signals:

protected slots:
    void testslot();
};

#endif // MYOBJECT_H
```

创建接受槽函数头文件

## 实现发送信号类函数

```
#include "Testthread.h"
#include <QDebug>
#include <QThread>
Testthread::Testthread(QObject *parent) :
    QThread(parent)
{
}

void Testthread::run()
{
    qDebug() << "Testthread::run ID= " << QThread::currentThreadId();
    int i = 5;
    while(i--)
    {
        qDebug() << "i = " << i << endl;
        sleep(1);
    }

    emit testsignal();
    //emit发送信号,指定发送哪个函数做信号,这里是testsignal
    exec(); //开启线程事件循环

    qDebug() << "Testthread run end!!";
}
```

## 实现接受信号类函数

```
#include "myobject.h"
#include <QDebug>
#include <QThread>
myObject::myObject(QObject *parent) :
    QObject(parent)
{
}

void myObject::testslot()
{
    qDebug() << "myObject::testslot slot ID =" << QThread::currentThreadId();
}

#include "Testthread.h"
#include "myobject.h"
#include <QDebug>

void direct_connection()
{
    static Testthread t;
    static myObject m;
    QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()), Qt::DirectConnection);
    t.start(); //启动线程

    t.wait(4000); //等待4秒
    t.quit(); //quit函数是退出线程
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();

    direct_connection(); //调用direct_connect方式的信号与槽
    return a.exec();
}
```

我们使用 connect 链接信号和槽的时候用

Qt::DirectConnection 模式链接

- Qt::DirectConnection (立即调用)

- Qt::QueuedConnection (异步调用)

- Qt::BlockingQueuedConnection (同步调用)

```

main id = 0xbff0
Testthread::run ID= 0x2414
i = 4
i = 3
i = 2
i = 1
i = 0
myObject::tests slot ID = 0x2414
Testthread run end!!

```

t 线程执行，然后发送信号

```

void direct_connection()
{
    static Testthread t;
    static myObject m;
    QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()));
    t.start(); //启动线程

    t.wait(4000); //等待4秒
    t.quit(); //quit函数是退出线程
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();

    direct_connection(); //调用direct_connect方式的信号与槽
    return a.exec();
}

```

因为我的 m 对象是在主线程创建的

发现槽函数的线程 ID 是 t 线程的 ID，而不是主线程的 ID，

我虽然主线程调用的子函数，但是子函数展开就是在主线程创建的 m 对象

这时候 m 对象依附于主线程，那么 m 对象创建的槽函数也依附于 m 也就间接依附于主线程了，但是为什么 m 是依附于子线程呢？

`QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()), Qt::DirectConnection);`

就是因为 connect 的时候使用了 `Qt :: DirectConnection` 属性，这个属性是立即调用

意思是：

```

void Testthread::run()
{
    qDebug() << "Testthread::run ID= " << QThread::currentThreadId();
    int i = 5;
    while(i--)
    {
        qDebug() << "i = " << i << endl;
        sleep(1);
    }

    emit testsignal(); // ==直接将myObject::testslot()赋值到这行
    //emit发送信号,指定发送哪个函数做信号,这里是testsingal
    exec(); //开启线程事件循环

    qDebug() << "Testthread run end!!";
}

```

`Qt :: DirectConnection` 就是信号发送的同时执行槽函数，也就是把槽函数放在信号所处的线程来执行

而且槽函数依附于子线程 t，会出现一个现象，就是线程 t 发送信号后，要等待槽函数执行完才能向下继续执行。这种 `DirectConnection` 要求发送信号的线程和接受信号(slot 槽函数)线程不是在同一个线程，否则会产生 bug。

像槽函数创建在 m 对象，信号在 t 线程对象，这种情况就算 t 线程使用 `emit` 内部将 slot 放入 t 线程执行也不会出现 bug，因为创建信号和槽的是两个对象

```

1 #include "myobject.h"
2 #include <QDebug>
3 #include <QThread>
4 myObject::myObject(QObject *parent) :
5     QObject(parent)
6 {
7 }
8
9 void myObject::testslot()
10 {
11     int i=5;
12     while(i--)
13     {
14         qDebug ()<<"i--";
15     }
16     qDebug ()<<"myObject::testslot slot ID =" <<QThread::currentThreadId();
17 }
18

```

```

main id = 0x260c
Testthread::run ID= 0x790
i = 4

i = 3

i = 2

i = 1

i = 0

i--
i--
i--
i--
i--
i--
myObject::testslot slot ID = 0x790
Testthread run end!!

```

我在 m 对象槽函数加入循环，我们看看线程 t 是不是要等待 m 对象的槽函数执行完了，才回到 t 线程继续向下执行。

你看 t 线程最后一个函数要等槽函数执行完了才能执行，所以槽函数依附于线程也会牺牲线程效率的

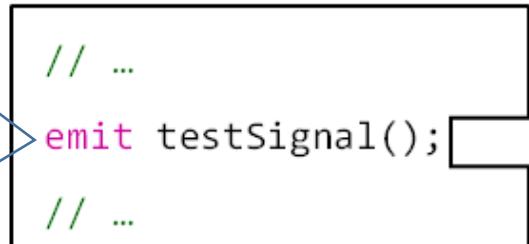
## Qt::QueuedConnection 异步调用和立即调用的区别

- Qt::DirectConnection ( 立即调用 )
- Qt::QueuedConnection ( 异步调用 )
- Qt::BlockingQueuedConnection ( 同步调用 )
- Qt::AutoConnection ( 默认连接 )
- Qt::UniqueConnection ( 单一连接 )



t.testSignal() ————— **QueuedConnection** —————> m.testSlot()

Qt::QueuedConnection 属性的意思就是 t 线程发送了 emit 信号，然后 t 线程不会等待槽函数执行完了后继续向下执行，而是发送信号之后就直接向下执行



目标线程  
事件队列

```

void queued_connection()
{
    static Testthread t;
    static myObject m;
    QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()), Qt::QueuedConnection);
    t.start(); //启动线程

    t.wait(4000); //等待4秒
    t.quit(); //quit函数是退出线程
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();

    queued_connection(); //调用queued_connection方式的信号与槽
    return a.exec();
}

```

```

main id = 0x2a7c
Testthread::run ID= 0x21f0
i = 4
i = 3
i = 2
i = 1
i = 0
Testthread run end!!
myObject::testslot slot ID = 0x2a7c

```

你看 t 线程发送信号 emit 之后直接执行到 t 线程结束 ,至于槽函数什么时候执行是槽函数依附的对象线程决定的

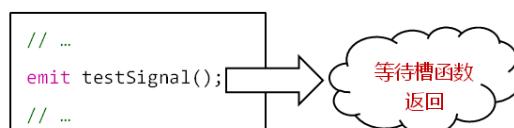
Qt::QueuedConnection 还有个功能就是让 emit 不能向 Qt :: DirectConnection 这样直接将槽函数替换到发送信号的线程里面来执行 ,这样发送信号的线程要等待槽函数执行完了才能继续向下执行 ,比如 t 线程。所以 Qt::QueuedConnection 让槽函数依附于创建它的对象线程 ,这里是 m 对象 ,m 对象是主线程创建的 ,所以是主线程来决定什么时候执行槽函数

## 下面看看同步调用

- Qt::DirectConnection (立即调用)
- Qt::QueuedConnection (异步调用)
- Qt::BlockingQueuedConnection (同步调用)**
- Qt::AutoConnection (默认连接)
- Qt::UniqueConnection (单一连接)

BlockingQueuedConnection 的功能和 DirectConnection 的功能差不多 ,都是线程发送信号后 ,要等待槽函数执行完了 ,才回来执行该线程 ,但是这里一个唯一的特点 ,看下面

t.testSignal() —————— BlockingQueuedConnection ——————> m.testSlot()



```

void myObject::testslot()
{
    int i=10;
    while(i--)
    {
        qDebug() << "i-- = " << i;
    }
    qDebug() << "myObject::testslot slot ID =" << QThread::currentThreadId();
}

```

打印 10 次 ,这样好和上面的 DirectConnection 例程区分

```

void Blockqueued_connection()
{
    static Testthread t;
    static myObject m;
    QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()), Qt::BlockingQueuedConnection);
    t.start(); //启动线程

    t.wait(4000); //等待4秒
    t.quit(); //quit函数是退出线程
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

main id = 0x12c
Testthread::run ID= 0xa28
i = 4
i = 3
i = 2
i = 1
i = 0
i-- = 9
i-- = 8
i-- = 7
i-- = 6
i-- = 5
i-- = 4
i-- = 3
i-- = 2
i-- = 1
i-- = 0
myObject::testslot slot ID = 0x12c
Testthread run end!!

```

Connect 使用 Qt::BlockingQueuedConnection 同步调用属性

DirectConnection 例程虽然信号发送之后要等待槽函数执行完才能继续执行当前线程，这是因为槽函数依附于子线程 ID。但是 BlockingQueuedConnection 可不是依附于子线程 ID，而是依附于创建对象的线程 ID，testslot 是 m 对象创建的，m 对象是主线程创建的，所以槽函数依附于主线程 ID，只不过 BlockingQueuedConnection 让槽函数在分离 t 线程 ID 的同时又具备信号的跟踪性，就是不管哪个信号，只要 emit 了 testslot 这个槽函数，那么该 emit 的线程就要等待槽函数执行完了才能接着向下执行

AutoConnection 就是我们常用的 connect 信号与槽链接，最后形参不加 Qt:: 属性

- Qt::DirectConnection (立即调用)
- Qt::QueuedConnection (异步调用)
- Qt::BlockingQueuedConnection (同步调用)
- Qt::AutoConnection (默认连接)**
- Qt::UniqueConnection (单一连接)

如果发送 emit 信号的线程和槽函数 slot 执行的线程是同一个线程 ID  
那么 QT 就自动在 connect 后面帮你加入 DirectConnection 参数

AutoConnection = {  
 DirectConnection , 发送线程 = 接收线程  
 QueuedConnection , 发送线程 != 接收线程
}

如果发送 emit 信号的线程和槽函数 slot 执行的线程不是同一个线程 ID 那么 QT 就自动在 connect 后面帮你加入 QueuedConnection 参数

```

void auto_connection()
{
    static Testthread t;
    static myObject m;
    QObject::connect(&t, SIGNAL(testsignal()), &m, SLOT(testslot()));
    t.start(); //启动线程

    t.wait(4000); //等待4秒
    t.quit(); //quit函数是退出线程
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "main id = " << QThread::currentThreadId();
    auto_connection(); //调用Blockqueued_connection方式的信号与槽
    return a.exec();
}

```

不加参数就是 Autoconnection ,  
QT 系统自动选择

```

main id = 0x29b4
Testthread::run ID= 0x3194
i = 4
i = 3
i = 2
i = 1
i = 0
i-- = 9
i-- = 8
i-- = 7
i-- = 6
i-- = 5
i-- = 4
i-- = 3
i-- = 2
i-- = 1
i-- = 0
Testthread run end!!
myObject::testslot slot ID = 0x29b4

```

这很明显槽函数还没执行完线程就先执行完了，所以线程不用等待槽函数执行完，证明 QT 选择的是 QueuedConnection

对象创建的线程生命周期是否会随着对象执行完后线程跟着消失，该问题出现在子函数里面创建对象的情况

```
1 #include <QtCore/QCoreApplication>
2 #include <QThread>
3 #include <QDebug>
4
5 class mythread : public QThread
6 {
7 protected:
8     void run()
9     {
10         while(1)
11         {
12             qDebug() << "mythread run!!";
13             sleep(1);
14         }
15     }
16 }
17
18 int main(int argc, char *argv[])
19 {
20     QApplication a(argc, argv);
21     mythread mt;
22     mt.start();
23     while(1)
24     {
25         qDebug() << "main run ...";
26         for(int i=500000000;i>0;i--) //延时函数
27     }
28     return a.exec();
29 }
```

```
main run ...
mythread run!!
```

主线程和 mt 对象的线程同时执行正常

下面我们来修改下线程对象创建方式

```

class mythread :public QThread
{
protected:
    void run()
    {
        while(1)
        {
            qDebug ()<<"mythread run!!";
            sleep(1);
        }
    }
};

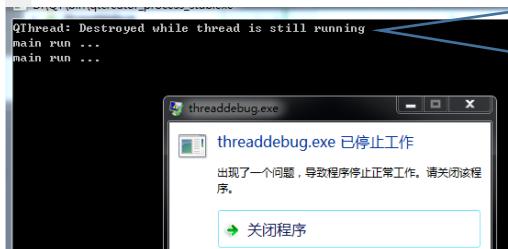
void creat_thread(void) //定义个子函数创建对象
{
    mythread mt;           在子函数里面创建对象
    mt.start();
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    creat_thread(); //在子函数创建对象

    while(1)           执行到这里的时候子函数 creat_thread  
已经执行完了，子函数里面的对象会自  
动释放，这时候子 mt 线程还在吗？
    {
        qDebug ()<<"main run ...";
        for(int i=500000000;i>0;i--); //延时函数
    }
    return a.exec();
}

```



主函数的子函数 creat\_thread 已经执行完了，但是子函数的 run 还在执行，子函数创建的 mt 对象里面的栈都已经释放了，如果这时候 mt 对象的 run 函数还在执行就是非法操作栈空间，程序就会报错

如何解决对象比对象对应的线程先执行完成的情况呢？

编写多线程有一条准则

## 线程对象生命周期 > 对应的线程生命周期

就是你要评估你的对象存在时间一定要大于你对象创建的线程运行时间，必须是对象的线程运行先结束，然后对象才能结束。

所以我们要做一个在线程对象销毁的同时，自动把对象对应的线程强制执行结束。

```
class mythread :public QThread
{
protected:
    void run()
    {
        for(int i=5;i>0;i--)
        {
            qDebug() << "mythread run!!";
            sleep(1);
        }
    }
public:
    ~mythread()
    {
        wait();
        qDebug() << "~mythread wait end";
    }
};

void creat_thread(void) //定义子函数创建对象
{
    mythread mt;
    mt.start();
}

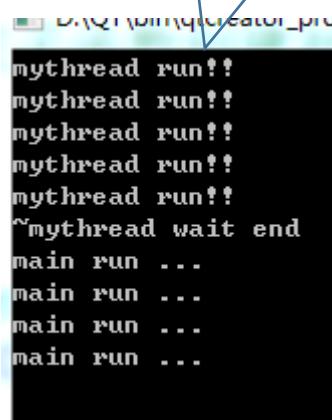
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    creat_thread(); //在子函数创建对象

    while(1)
    {
        qDebug() << "main run ...";
        for(int i=500000000;i>0;i--) //延时函数
    }
    return a.exec();
}
```

在析构函数中加入 wait，让对象在执行完之前执行析构函数，发现有 wait()函数的存在，那么创建 mt 对象的线程（主线程）就会卡在 wait()函数这里，让对象的线程函数执行完后，wait()释放继续向下执行，这时候对象才执行完

这种 wait 函数因为要等对象的线程执行完，才向下继续执行，所以 creat\_thread 函数卡了 5 秒钟，就是等待 run 函数执行完，然后才轮到主线程执行



```
mythread run!!
mythread run!!
mythread run!!
mythread run!!
mythread run!!
~mythread wait end
main run ...
```

所以 wait 来增加线程执行时间大于对象执行时间的方式，必须要求对象线程里面执行的函数在几个毫秒甚至几个微妙内执行完，这样 wait 才有意义，如果对象的线程执行几秒几十秒，那么其它线程就只有卡起等到了。

如果线程执行时间很长，那么如何保证对象和对象对应的线程同时销毁，而不影响其它线程的效率呢？  
这就要用**异步性线程设计**，上一页是同步性线程设计。

异步性线程就是线程执行完成后，通知系统销毁线程对象。异步性线程只能在堆中创建线程对象。

```
class mythread :public QThread
{
protected:
    explicit mythread()
    {

    }
    ~mythread()
    {
        qDebug ()<<"Async ~mythread";
    }
    void run()
    {
        for(int i=5;i>0;i--)
        {
            qDebug ()<<"mythread run!!";
            sleep(1);
        }
        deleteLater(); //Qt库函数deleteLater表示摧毁当前对象
    }
public:
    static mythread* NewInstance()
    {
        return new mythread(); //创建一个对象返回该对象地址
    }
};

void creat_thread(void) //定义个子函数创建对象
{
    mythread* mt = mythread::NewInstance(); //创建对象得到地址赋值给mt
    mt->start(); //启动对象

    delete mt;
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    creat_thread(); //在子函数创建对象

    while(1)
    {
        qDebug ()<<"main run ...";
        for(int i=500000000;i>0;i--); //延时函数
    }
    return a.exec();
}
```

1. 在 run 函数执行完之前调用系统的 deleteLater 函数，这个函数就是叫系统销毁当前对象，也就是创建这个线程的对象。

2. 在堆空间创建对象，其实不用用函数封装返回地址，直接 new mythread()是一样的，这里不过是为了规范。

3. 在创建线程的子函数中，执行 NewInstance 在堆空间创建 mt 对象。

4. 这里执行 delete 会报错，因为析构函数是 protected 的，就是不准你在这里释放掉堆空间对象，如果这里释放掉堆空间对象，那么和前面的问题一样了，creat\_thread 函数先执行完，释放 mt 对象，然后 mt 对象的线程没有执行完，系统崩溃

```

void creat_thread(void) // 定义子函数创建对象
{
    mythread* mt = mythread::NewInstance(); // 创建对象得到地址赋值给mt
    mt->start(); // 启动对象
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    creat_thread(); // 在子函数创建对象

    while(1)
    {
        qDebug() << "main run ...";
        for(int i=500000000;i>0;i--) // 延时函数
    }
    return a.exec();
}

```

```

main run ...
mythread run!!
mythread run!!
main run ...

```

你看主线程和 mt 线程是并行执行的。  
这样就没有 wait 那样麻烦了。

```

void run()
{
    for(int i=5;i>0;i--)
    {
        qDebug() << "mythread run!!";
        sleep(1);
    }
    deleteLater(); // Qt 库函数 deleteLater 表示摧毁当前对象
}

```

这里是因为 mt 对象的 run 函数执行 5 次循环后，线程就结束了，  
用 deleteLater 释放掉了 mt 对象。所以就只剩下主线程执行了。

所以异步性线程设计方法可以用于多线程长时间执行的应用程序。

## Qt 另一种创建线程方式

```

class QThread : public Qt
{
    // .....
    // .....

protected:
    virtual void run() = 0;
    // .....
    // .....
};

```

我们前面用类继承的 QThread 实现多  
线程时，QThread 的原形就是这样

那么既然 QThread 原形是虚函数来构建的 run，那么有什么问题呢？

```

class mythread : public QThread
{
protected:
    void run()
    {
        for(int i=5;i>0;i--)
        {
            qDebug() << "mythread run!!";
            sleep(1);
    }
}

```

比如我想建立个类，实现一个新的功能，那么就要去继承至 Qthread

这样貌似也没什么问题啊。

比如我要增加新的功能，也可以把以前的线程 A 类拿来，然后新建立一个 B 类，让 B 类继承至 A 类，这种开发方式。

现在我要实现不用继承 Qthread，也能实现多线程

```

class QThread : public QObject
{
    Q_OBJECT

// .....

protected:
    virtual void run()
    {
        (void)exec();
    }

// .....
};

```

QT4.7 版本改进后的 Qthread 不再是虚函数，里面有个 exec 函数来帮你执行多线程的槽函数，当然同时也支持老版本的虚函数 run

```

1 ifndef THREAD2_H
2 define THREAD2_H

3 include <QObject>
4 include <QThread>

5 class thread2 : public QObject
6 {
7     Q_OBJECT
8     QThread m_thread; //没有继承Qthread，但是还是要建立个Qthread对象来实现子线程

public:
    explicit thread2(QObject *parent = 0);
    void start();
protected slots:
    void tmain(); //定义一个槽函数当做线程run函数
signals:
public slots:
};

# endif // THREAD2_H

```

这个普通类没有继承 Qthread 线程类，但是也能做到子线程类的效果

1. 创建个线程对象

2. 在头文件建立槽函数，这个槽函数就是线程入口函数

```

#include "thread2.h"
#include <QDebug>

thread2::thread2(QObject *parent) :
    QObject(parent)
{
    moveToThread(&m_thread); //让线程对象m_thread依附在普通类thread2的对象下。
    connect(&m_thread, SIGNAL(started()), this, SLOT(tmain())); //链接线程对象到槽
}

void thread2::tmain()
{
    for(int i=5;i>0;i--)
    {
        qDebug() << "tmain ID = "<<QThread::currentThreadId();
        for(int i=100000000;i>0;i--) //延时
    }
}

void thread2::start()
{
    m_thread.start();
}

```

4.这就是让普通类变成线程类的关键，将线程对象依附在普通类，因为普通类要有线程类的效果也得靠线程对象间接驱动

6.这样槽函数就类似 run 函数了

5.把槽函数链接上这个线程对象

7.启动 m\_thread 线程，就会触发信号 started，因为信号链接的是槽函数，所以槽函数会被执行

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qDebug() << "main ID = "<<QThread::currentThreadId();

    thread2 t2;
    t2.start();

    return a.exec();
}

```

创建普通类 thread2 的对象 t2  
启动线程对象 m\_thread

```

main ID = 0x2564
tmain ID = 0xb28

```

你看 thread 对象线程执行成功，如果看不太直观，我们再修改一下代码

```

void thread2::tmain()
{
    for(int i=5;i>0;i--)
    {
        qDebug() << objectName() << "ID = "<<QThread::currentThreadId();
        for(int i=100000000;i>0;i--) //延时
    }
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qDebug() << "main id" << QThread::currentThreadId();

    thread2 t2;
    t2.setObjectName("t2 run");
    t2.start();

    thread2 t3;
    t3.setObjectName("t3 run");
    t3.start();

    return a.exec();
}

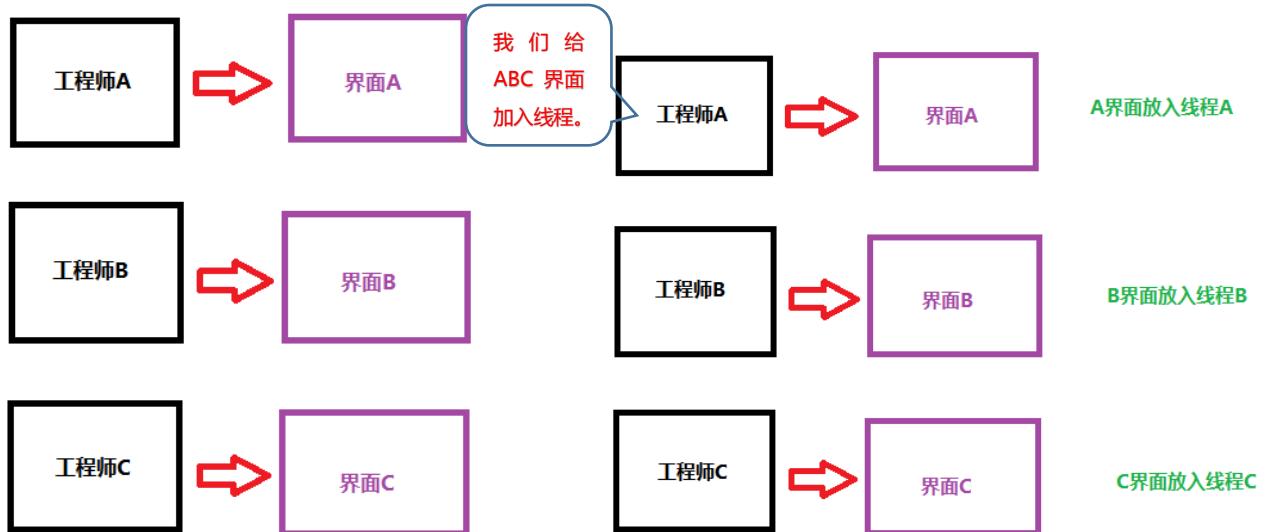
```

打印线程对象名称

你看两个对象同时执行，根据线程 ID 就能看出来，t2，t3 是并行执行的。所以这种非继承 Qthread 的类也是多线程实现的一种方法。这种方法是固定的，你就按照一个线程一个对象就是了，不需要考虑到父类继承了 Qthread 这种麻烦事

## QT 多线程发送数据给界面显示

我们在子线程中创建一个界面，这样的话一个子线程就是一个界面，那么每个人都可以用一个子线程将自己的界面包含进来，进行协同开发，这样是否可行呢？



```
1 ifndef TESTTHREAD_H
2 define TESTTHREAD_H
3
4 #include <QThread>
5
6 class Testthread : public QThread
7 {
8     Q_OBJECT
9 public:
10     explicit Testthread(QObject *parent = 0);
11
12 signals:
13
14 public slots:
15
16 protected:
17     void run(); // 创建run函数
18 };
19
20 endif // TESTTHREAD_H
```

```
#include "testthread.h"
#include "widget.h"
#include <QDebug>

Testthread::Testthread(QObject *parent) :
    QThread(parent)
{
}

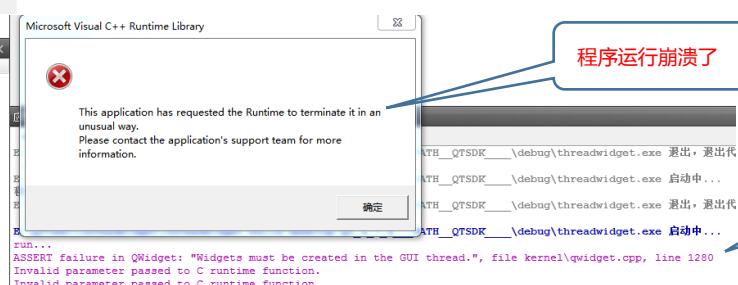
void Testthread::run()
{
    qDebug() << "run...";

    QWidget w; // 在线程里面创建一个窗口
    w.show();
    exec();
}
```

```
----- -----
#include "testthread.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Testthread *ptest = new Testthread();
    ptest->start();
    return a.exec();
}
```

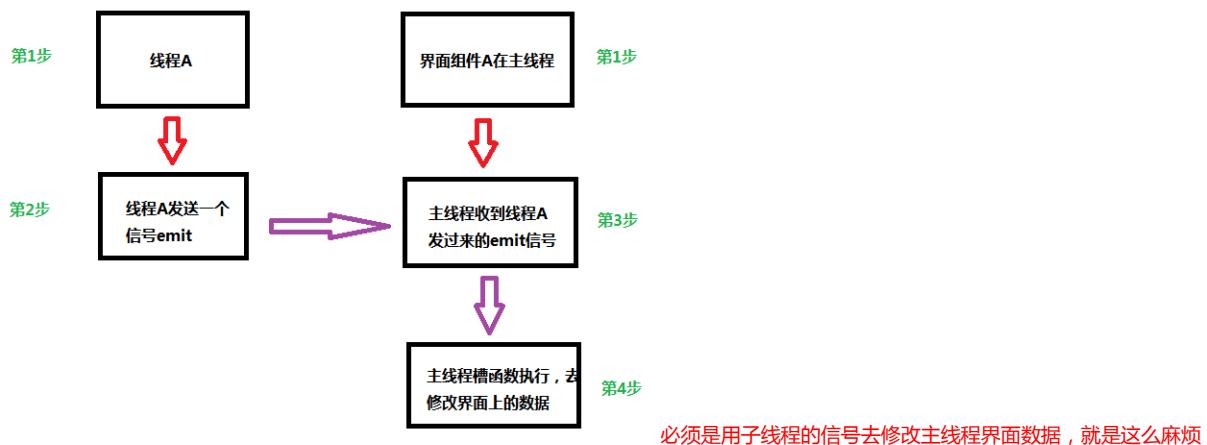


程序运行崩溃了

错误打印要求界面创建必须在 GUI 线程里面创建，意思就是要创建界面，必须在主线程里面创建

从以上情况我们知道了，我们不能直接给 ABC 三个工程师的界面各自分配一个子线程。必须将 ABC 工程师的界面放在主线程创建，也就是 ABC 三个工程师三个对象必须在主线程执行。

所以如果你想用子线程去修改界面上的数据必须遵循以下方法：



先修改我们的 Testthread 线程类，这个线程我假设是来接受底层传感器或者串口数据的。

```
#ifndef TESTTHREAD_H
#define TESTTHREAD_H

#include <QThread>

class Testthread : public QThread
{
    Q_OBJECT
public:
    explicit Testthread(QObject *parent = 0);

signals:
    void updateUI(QString text); // 定义一个发往主线程的信号函数
public slots:

protected:
    void run(); // 创建run函数
};

#endif // TESTTHREAD_H
#include <QDebug>

Testthread::Testthread(QObject *parent) :
    QThread(parent)
{
}

void Testthread::run()
{
    emit updateUI("Begin");

    for(int i=0; i<10; i++)
    {
        emit updateUI(QString::number(i));

        sleep(1);
    }

    emit updateUI("End");
}
```



```

class Widget : public QWidget
{
    Q_OBJECT
    Testthread m_thread;
    QPlainTextEdit textEdit;
public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();
protected slots:
    void appendText(QString text); //主线程界面线程定义一个槽函数接受子线程的数据变化
private:
    Ui::Widget *ui;
};

```

主界面创建子线程对象

用文本框来显示子线程对象发送过来的底层数据

定义一个槽函数来接受子线程发送过来的emit信号数据

```

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);

    textEdit.setParent(this);
    textEdit.move(20, 20);
    textEdit.resize(200, 150);
    textEdit.setReadOnly(true);

    connect(&m_thread, SIGNAL(updateUI(QString)), this, SLOT	appendText(QString)));
    m_thread.start();
}

```

文本框设置

子线程信号函数

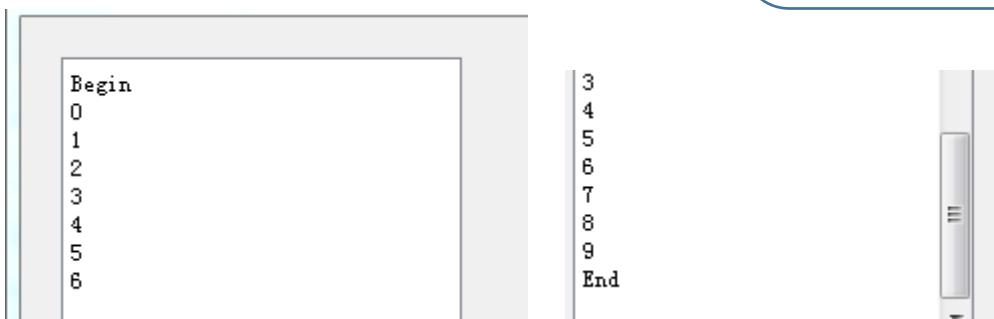
放入子线程对象

启动子线程，发送数据

第3步  
主线程收到线程A发过来的emit信号

第4步  
主线程槽函数执行，去修改界面上的数据

槽函数接受到 m\_thread 线程数据，去更新界面



这就是效果，子线程不能直接修改主界面，必须把数据用信号的方式发送给主界面槽函数，主界面槽函数将子线程的数据拿起来去更新界面。

