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

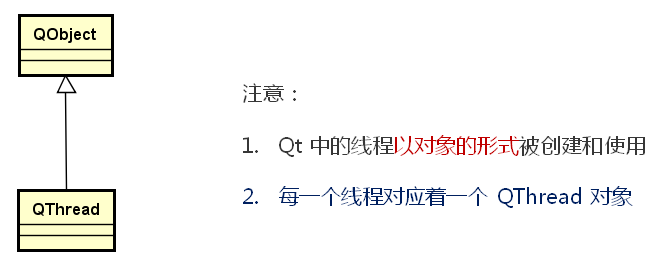
Qt另一种创建线程方式………………………………………………………………………………………………………..45

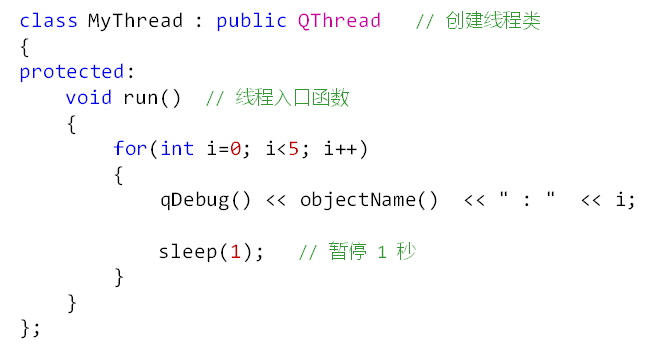
**多线程创建**

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



在QT中多线程是一个对象



而且

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



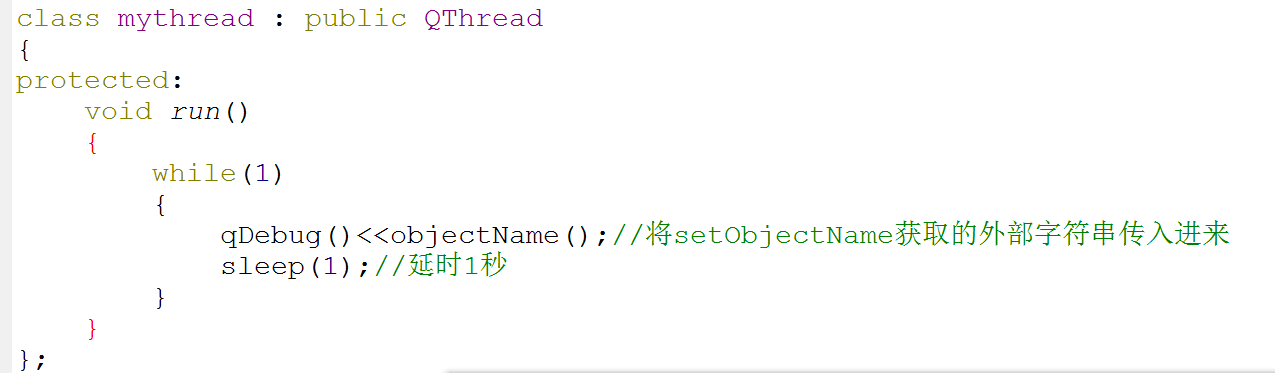
然后启动线程

每个线程都可以传入一个字符串数据

线程类创建好后要在主函数创建对象才行

这就是自己定义的线程类，在run写自己想要的代码

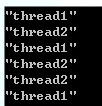
线程执行过程



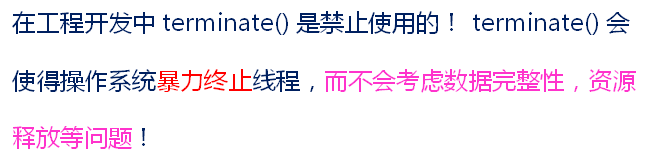
我创建了两个线程，但是都是用的同一个类，那么第二个线程就会复制第一个线程的run函数的内容和线程1并行执行

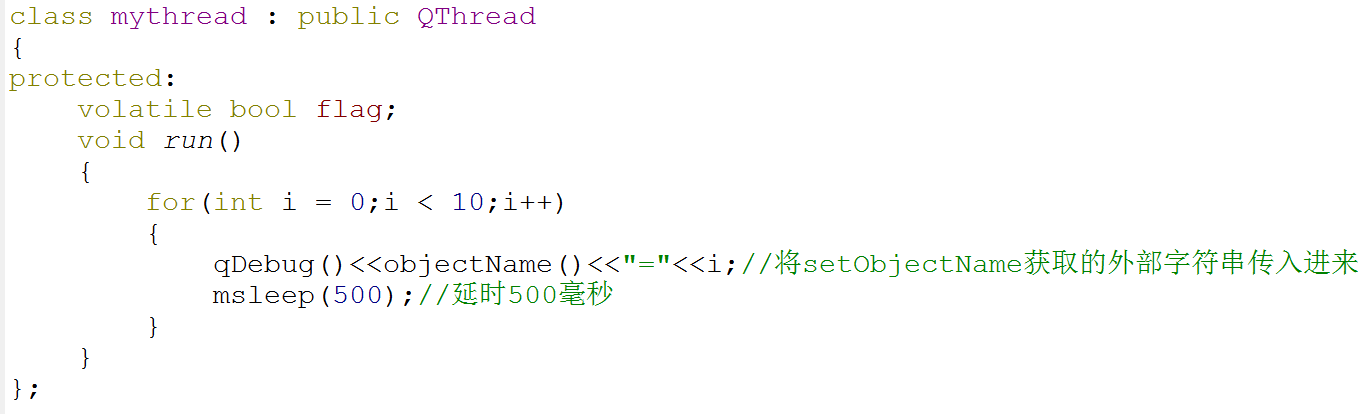


线程1和线程2谁先执行是随机的

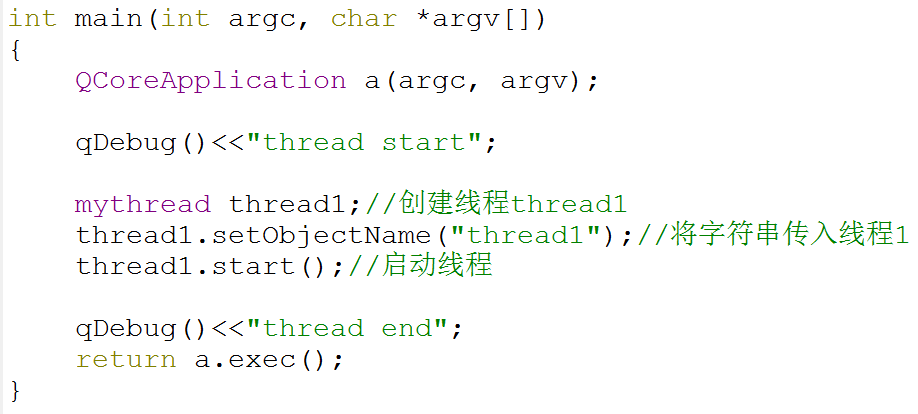
这就是QT的多线程，和linux C语言线程不一样，C语言是一个函数对应一个线程，但是QT可以一个函数对应多个对象创建出来的线程。

我们如何关闭线程



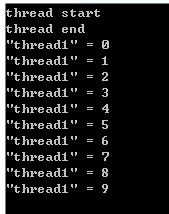


我发现一个问题，除了创建的线程在执行外，主程序main也是个线程，也在并行执行



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

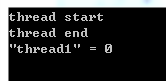
主程序main和创建的thread1线程在并行执行，创建打印测试看看





加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？

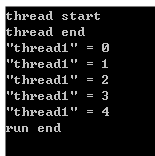
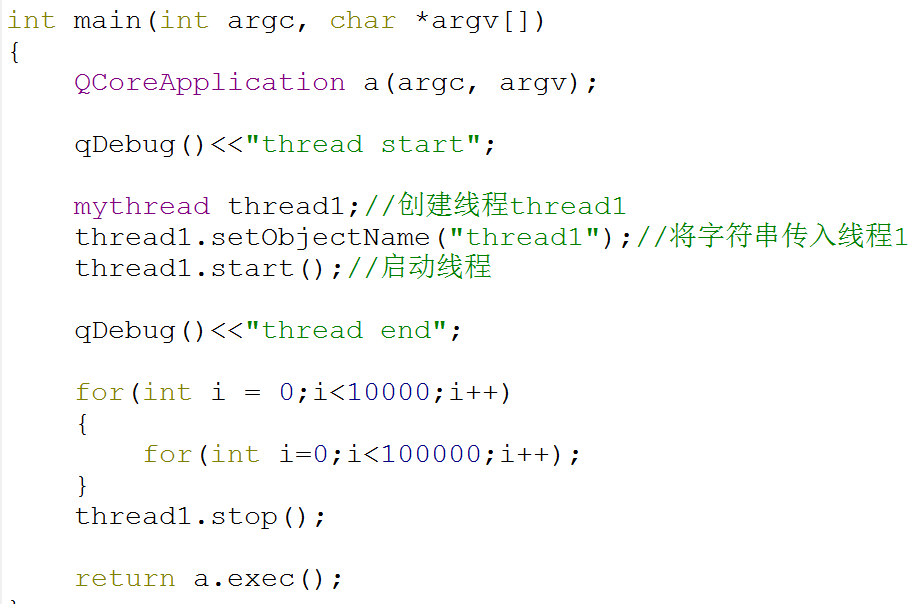
所以我们要禁止在工程开发中用terminate()函数

我们要在使用的线程类中增加标志位来判别线程状态



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

如果flag为false就继续循环

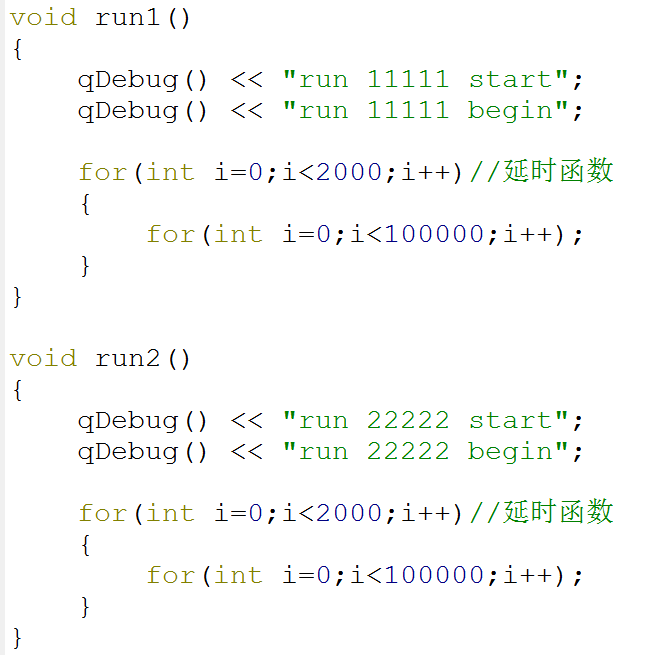


执行对象的stop函数让线程跳出循环

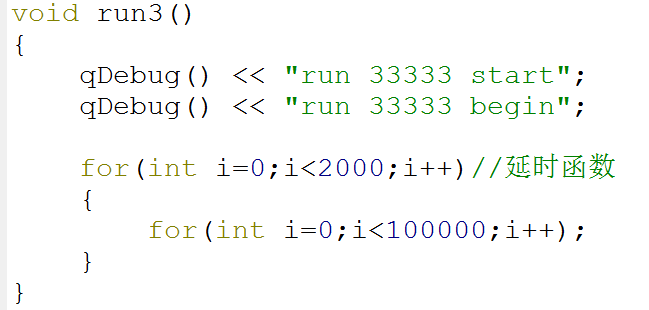
这里延时让主程序关闭子线程不要这么快，可以观察子线程现象

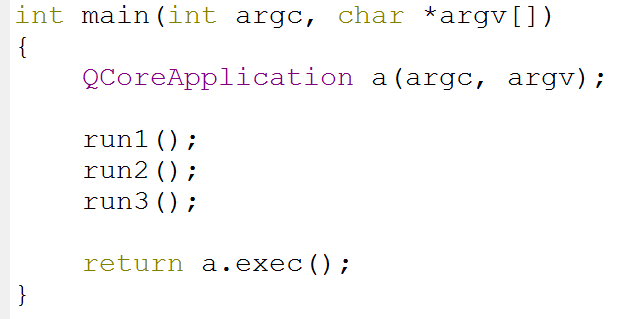
你看，虽然我没有执行完子线程循环就退出了，但是子线程循环结束后还执行了run end这条语句，所以这就证明了子线程不是terminate强制结束的。而是要执行完整个run函数才结束，那么在run后面的程序段就可以写释放内存，关闭功能的代码。

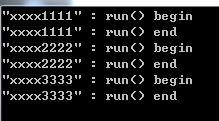
**多线程同步有什么用？**



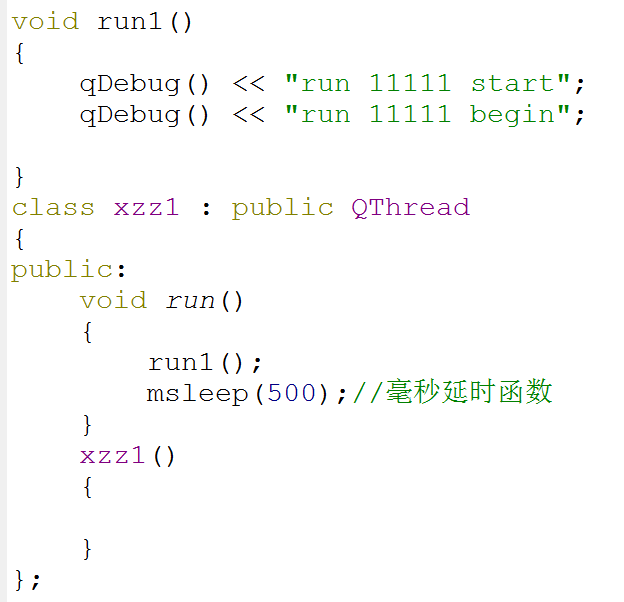
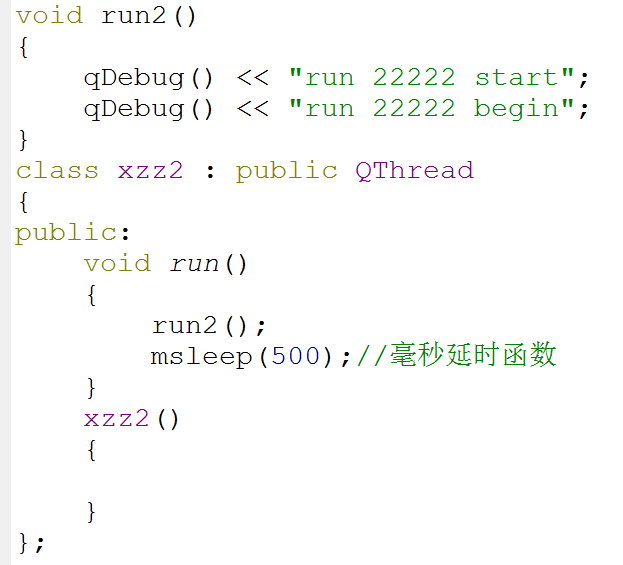
建立3个功能函数，run1(),run2(),run3()







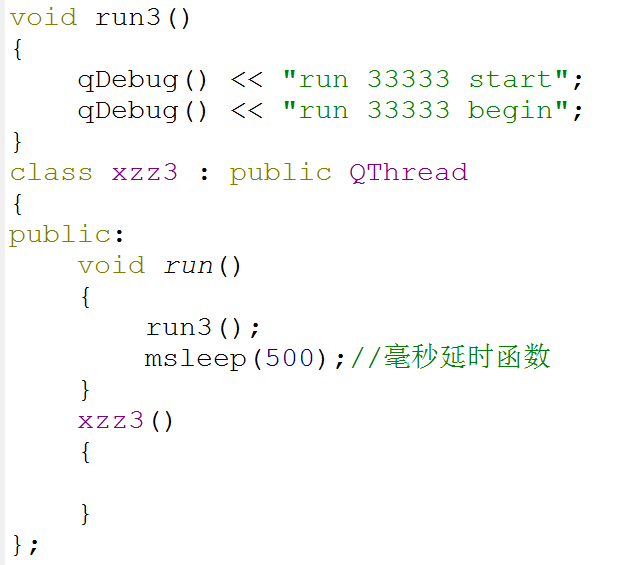
单线程情况下是按照顺序run1,run2,run3执行完

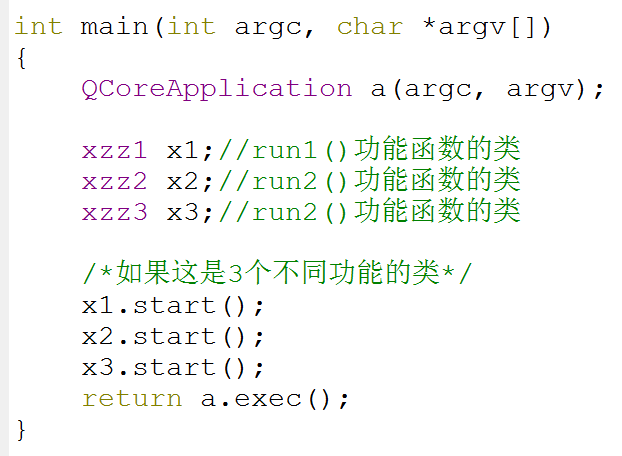
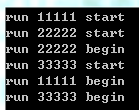
 

功能类2

功能类1

功能类3



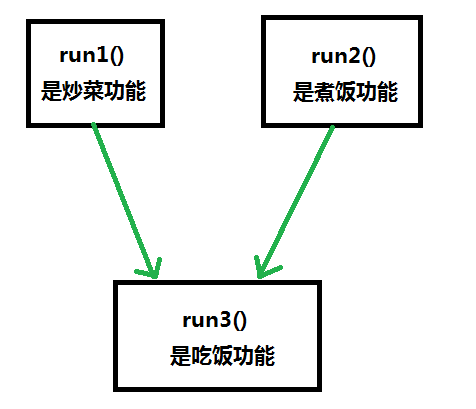
 

我们上一节就知道了，线程间的功能执行是并行的，但是哪个线程先执行和哪个线程后执行是随机的

主函数启动这3个不同功能的类和里面的函数

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run 1()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？



有些功能要求，两个函数并行执行，run1()和run2()同时执行，最后run1和run2执行完了把run1和run2的结果放入run3去执行。

这种情况用上面的线程随机执行就不行，run3必须等到run1,2执行完了才能执行，而run1和run2可以并行执行

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。



x1.wait表示必须x1执行完了才能向下执行，x2.wait也是一样。

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**



Produter线程写数据进变量

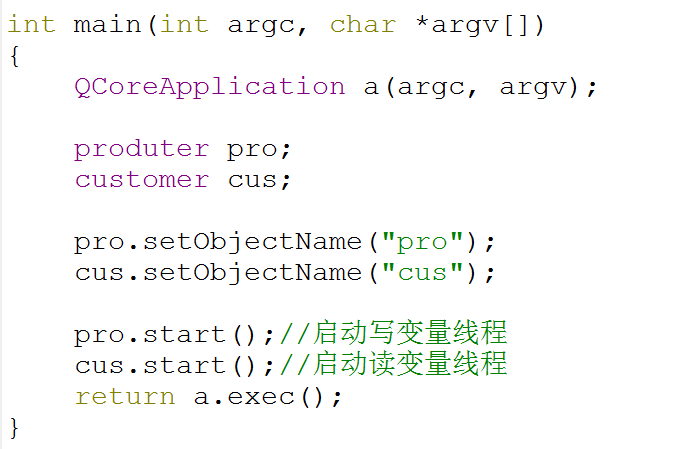
q\_store全局变量

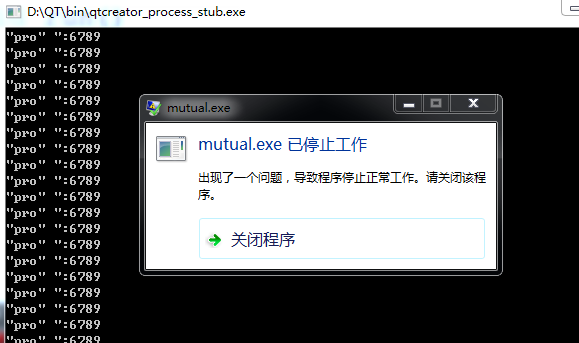
q\_store全局变量



customer线程读取变量数据

q\_store全局变量





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

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。

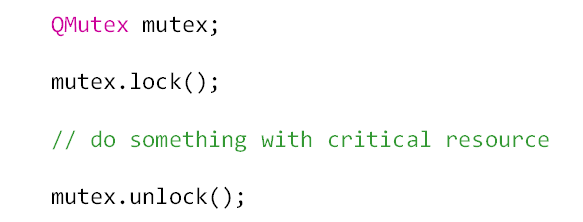
customer线程读取变量数据

q\_store全局变量

Produter线程写数据进变量

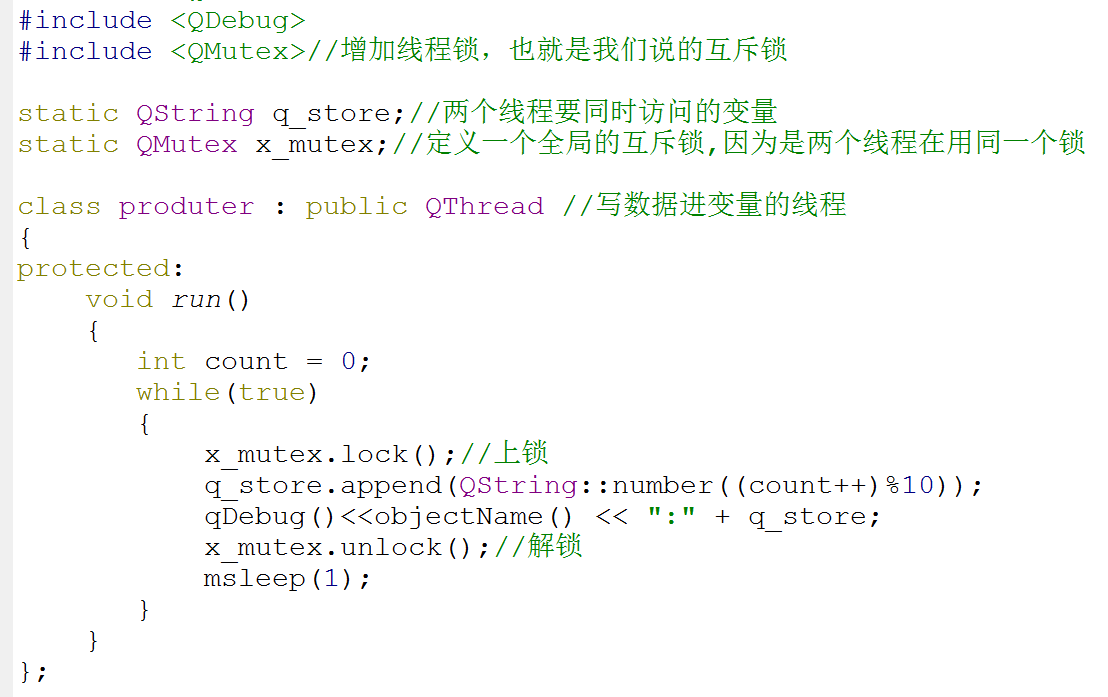
这就是多线程同时访问一个变量造成的互斥问题。

我们要给线程加入线程锁，也就是所谓的互斥锁



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



这是告诉其它使用x\_mutex变量的线程，我这里已经解锁了

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

如果我pro线程定义个A\_mutex，cus线程也定义个不同的B\_mutex，那么全局变量的保护就失去了意义。程序照样崩溃。因为pro使用的锁，cus根本不知道，因为cus使用的是另外一个锁。

Mutex使用注意事项1:

Mutex.lock

……..读写变量的代码…….

Mutex.unlock

Mutex.lock

……..读写变量的代码…….

如果线程只执行mutex.lock，然后unlock交给其它线程去执行就会出现程序BUG

线程锁必须在线程里面成对出现，就是锁线程和解锁线程都必须是成对出现在线程里面，如果线程里面执行了lock那么也必须在该线程找其它地方执行unlock，这样就是正确的

Mutex使用注意事项2:

Mutex.unlock

……..读写变量的代码…….

Mutex.lock

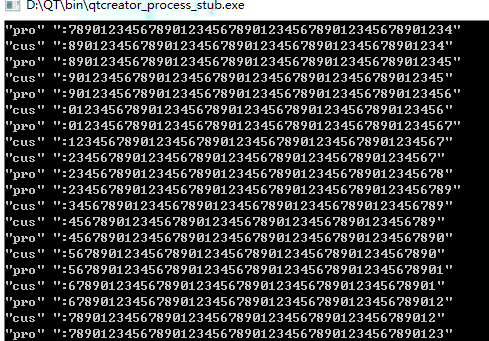
Mutex.lock

……..读写变量的代码…….

Mutex.unlock

如果先执行unlock，然后再执行lock这样程序也会出现BUG，所以lock和unlock执行是有先后顺序要求的

在读写变量代码时，必须先执行mutex.lock，然后再执行mutex.unlock

有了线程锁之后多线程读写同一个变量的程序就没有问题了

线程锁产生死锁，及解决方法

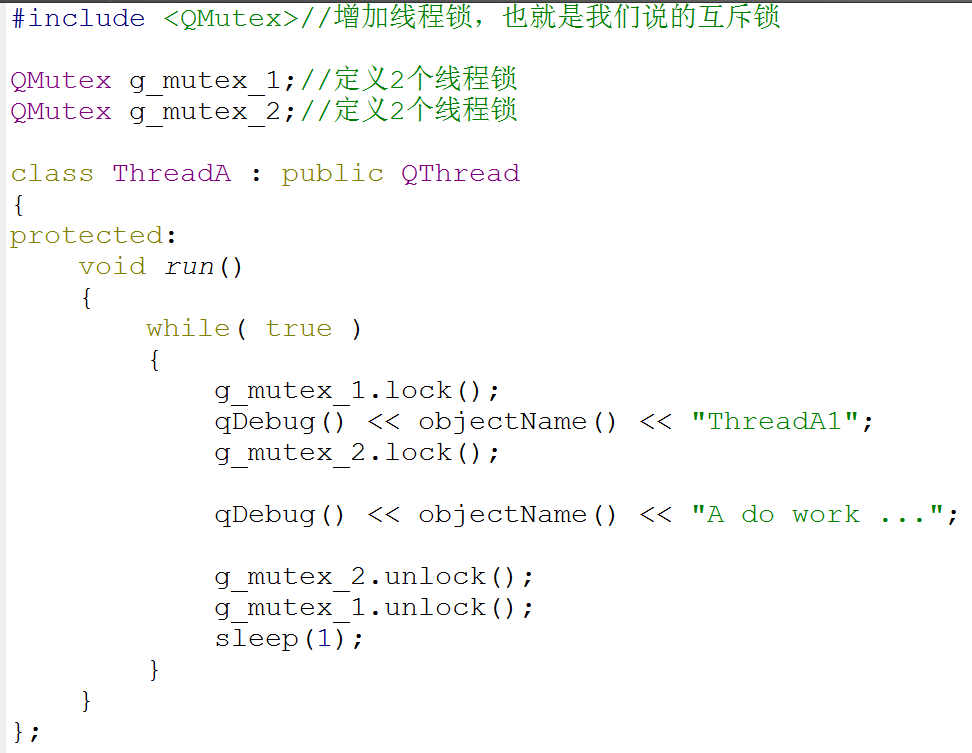


线程B 先m2.lock锁一次

接着m1.lock再锁一次

线程A 先m1.lock锁一次

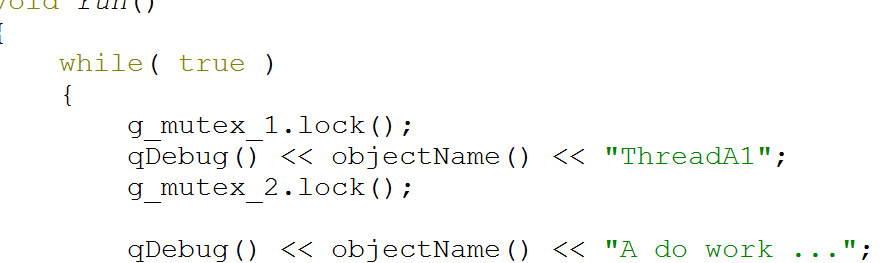
接着m2.lock再锁一次



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



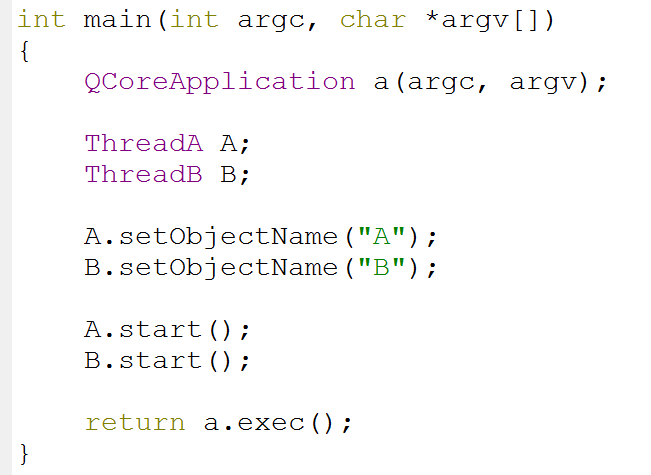
这个时候线程A又想锁q\_mutex\_2，发现mutex2倍线程B锁住，所以线程A只有等待

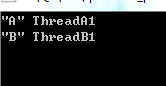


然后又想锁g\_mutex\_1，但是发现这个g\_mutex\_1锁已经被线程A锁住了，那么线程B只有在这里等待，等待线程A解锁

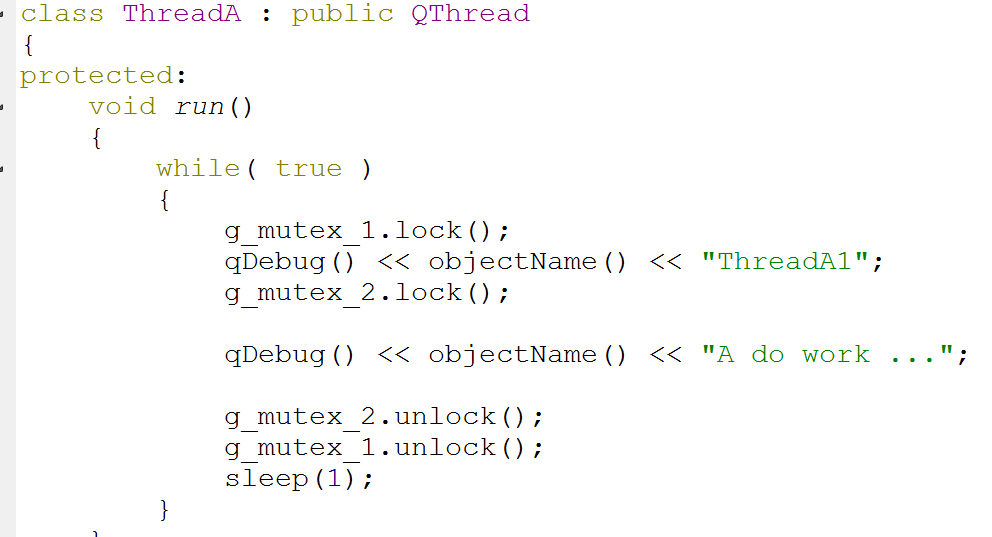
线程B会把g\_mutex\_2锁住

这种情况就是所谓的死锁



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



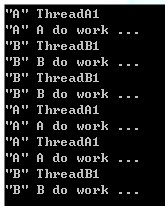
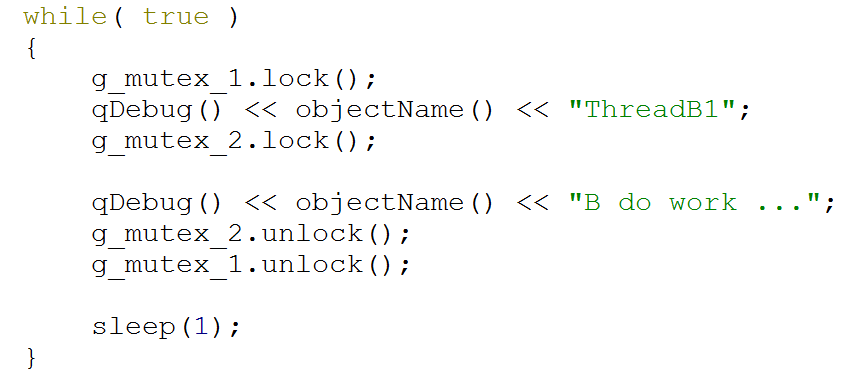
解锁也是按照倒序解锁的，先解锁最后一个加锁的锁2，然后再解锁倒数第二个加锁的锁1，以此类推….

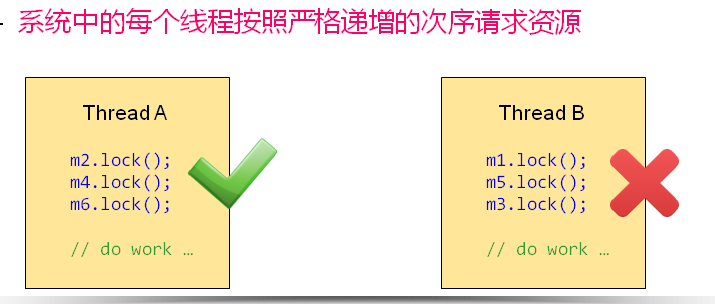
我们线程A的锁是按照顺序加锁的，先锁1，然后在锁2

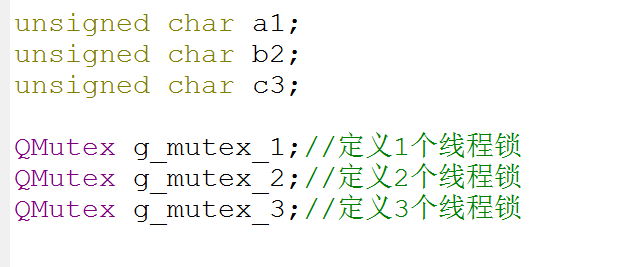


但是我们发现线程B的加锁是乱的，和线程A不一致，先加锁了2在加锁1，而且解锁也是没按照倒序的方法解锁，而是乱选择锁编号解锁

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



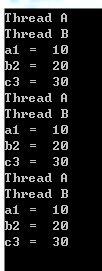
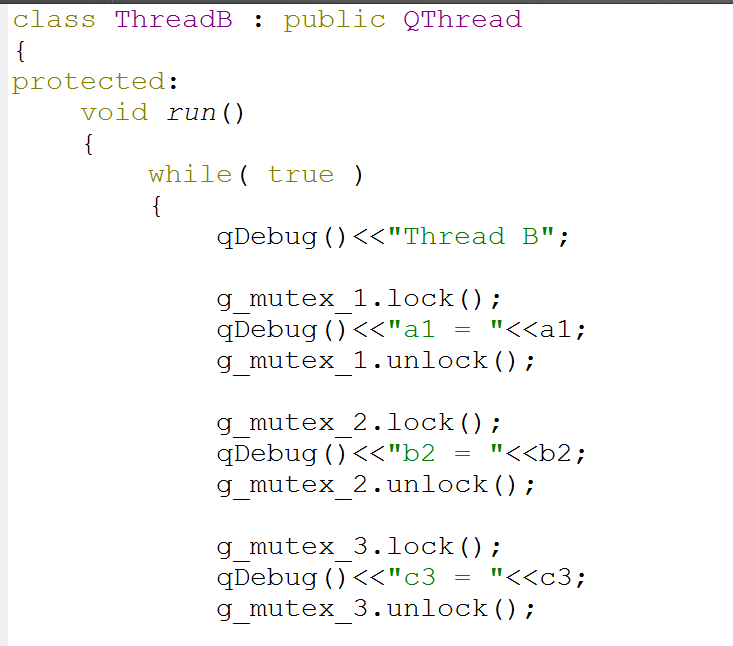
就是上面这种加锁解锁方法可以用在多个全局变量上，有多少个全局变量就定义多少个锁，按照顺序加锁解锁。



我定义了3个全局变量，每个全局变量给了一把锁



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



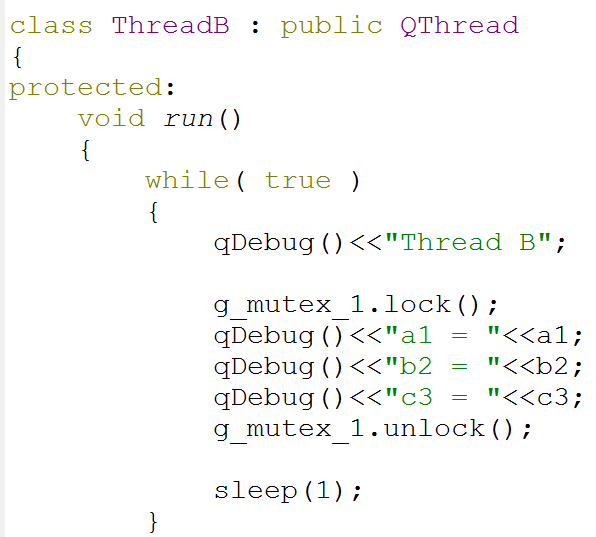
线程B要读取全局变量的内容，那么加锁解锁顺序也必须按照线程A的加锁解锁顺序来，因为这几个全局变量是先按照A线程的方式锁的。

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

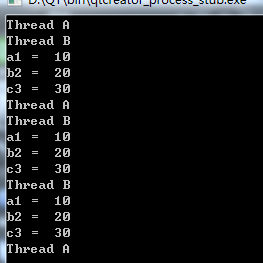


线程A把3个全局变量锁起来全部赋值

我定义的3个全局变量只用一把锁

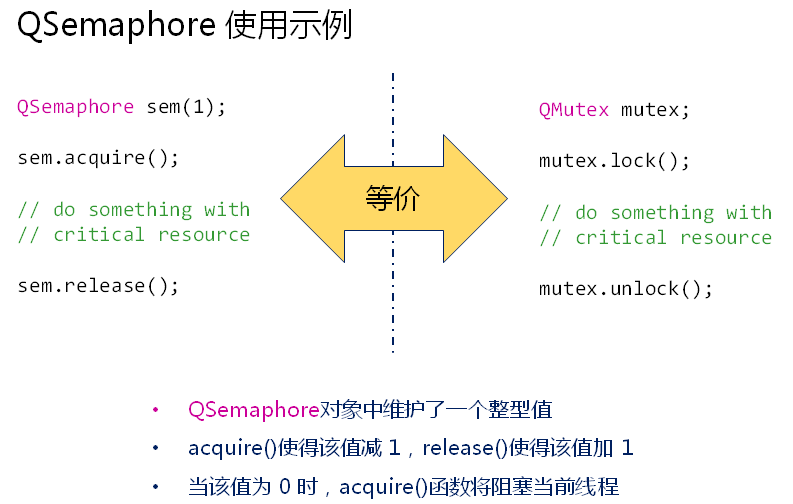


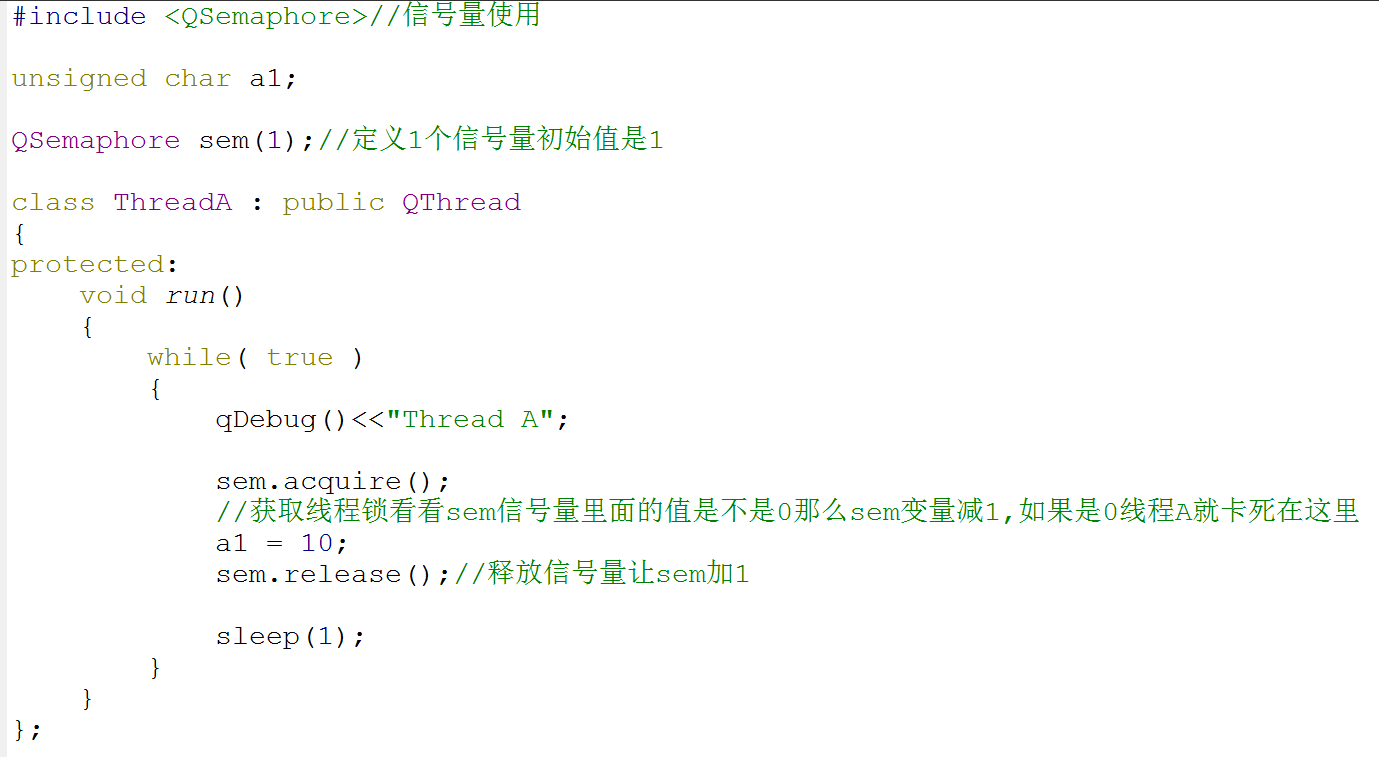
线程B把三个全局变量锁起来，读取三个全局变量数据



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

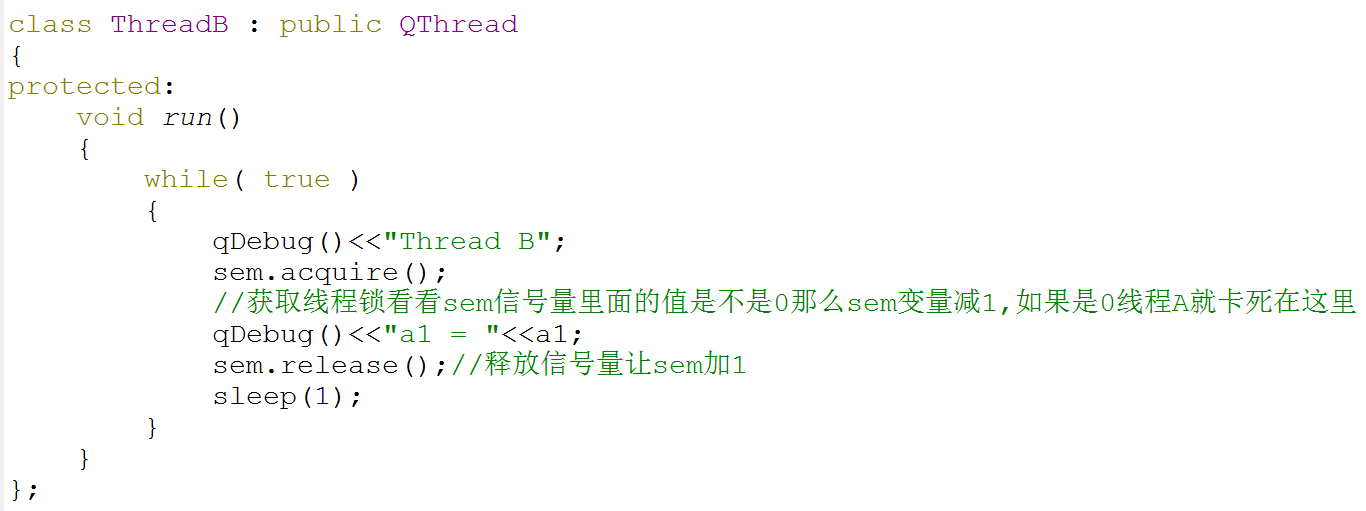
**信号量也是一种特殊的线程锁，信号量使用**



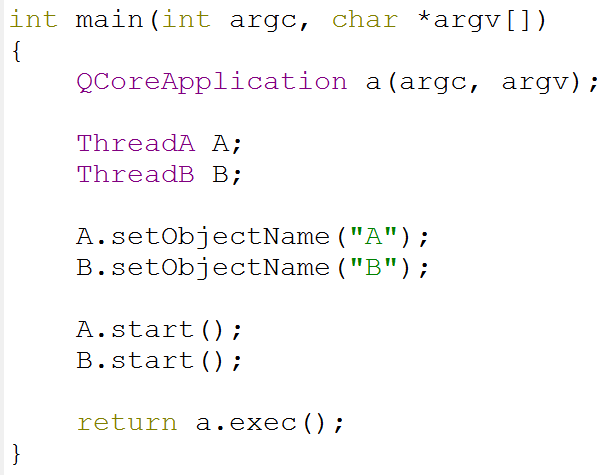
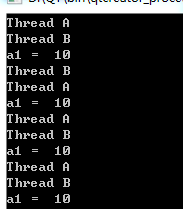


信号量使用方法和互斥锁类似

我们定义了一个信号量

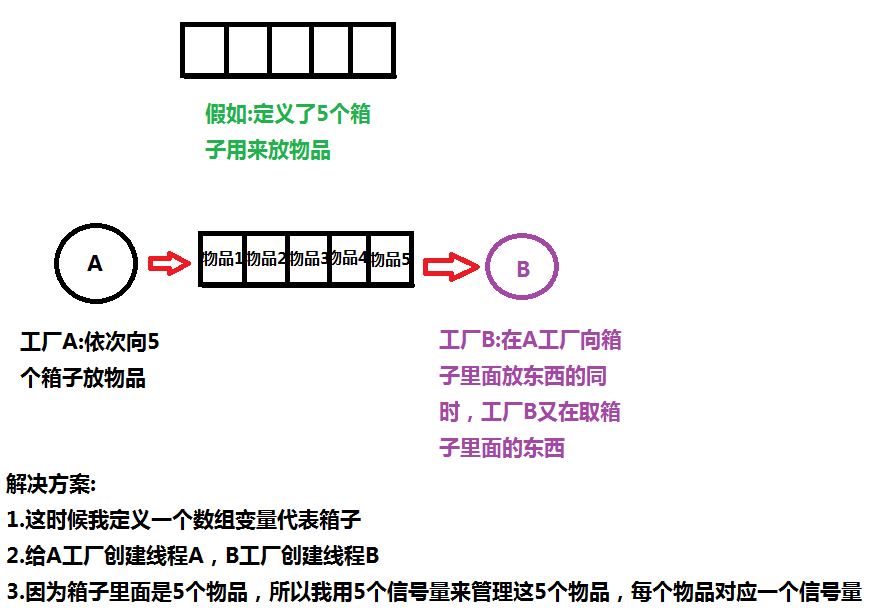


信号量使用方法和互斥锁类似

 程序运行正常，那信号量到底有什么用?

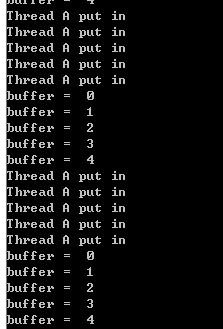
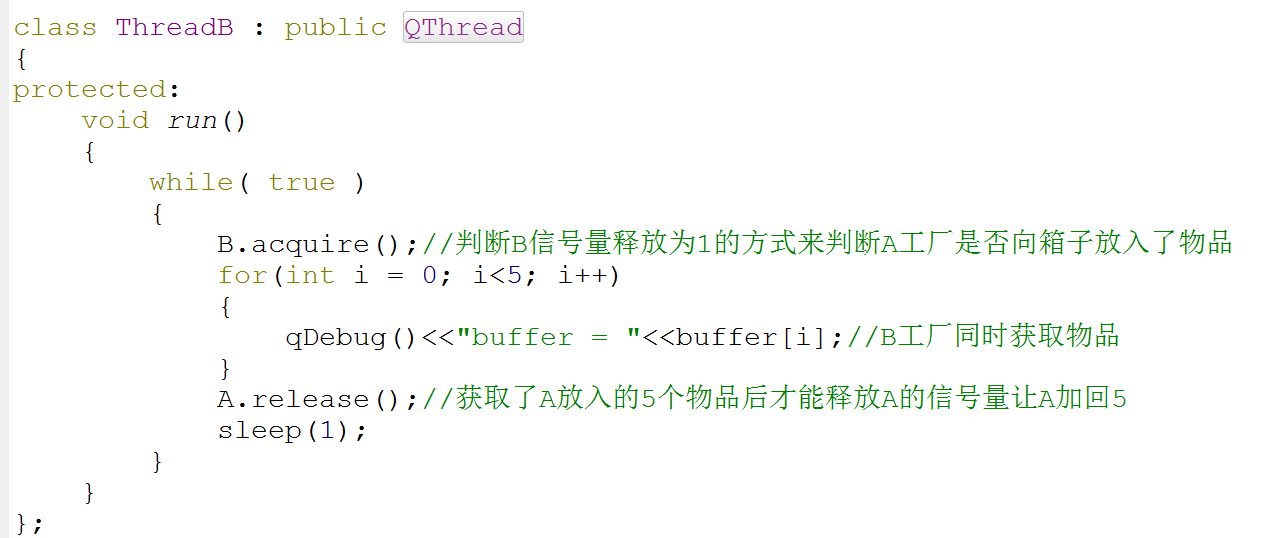
如果是上一页这样使用信号量，我不知道直接使用互斥锁啊，我拿信号量有什么用？

下面看看信号量真正的应用场景





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。

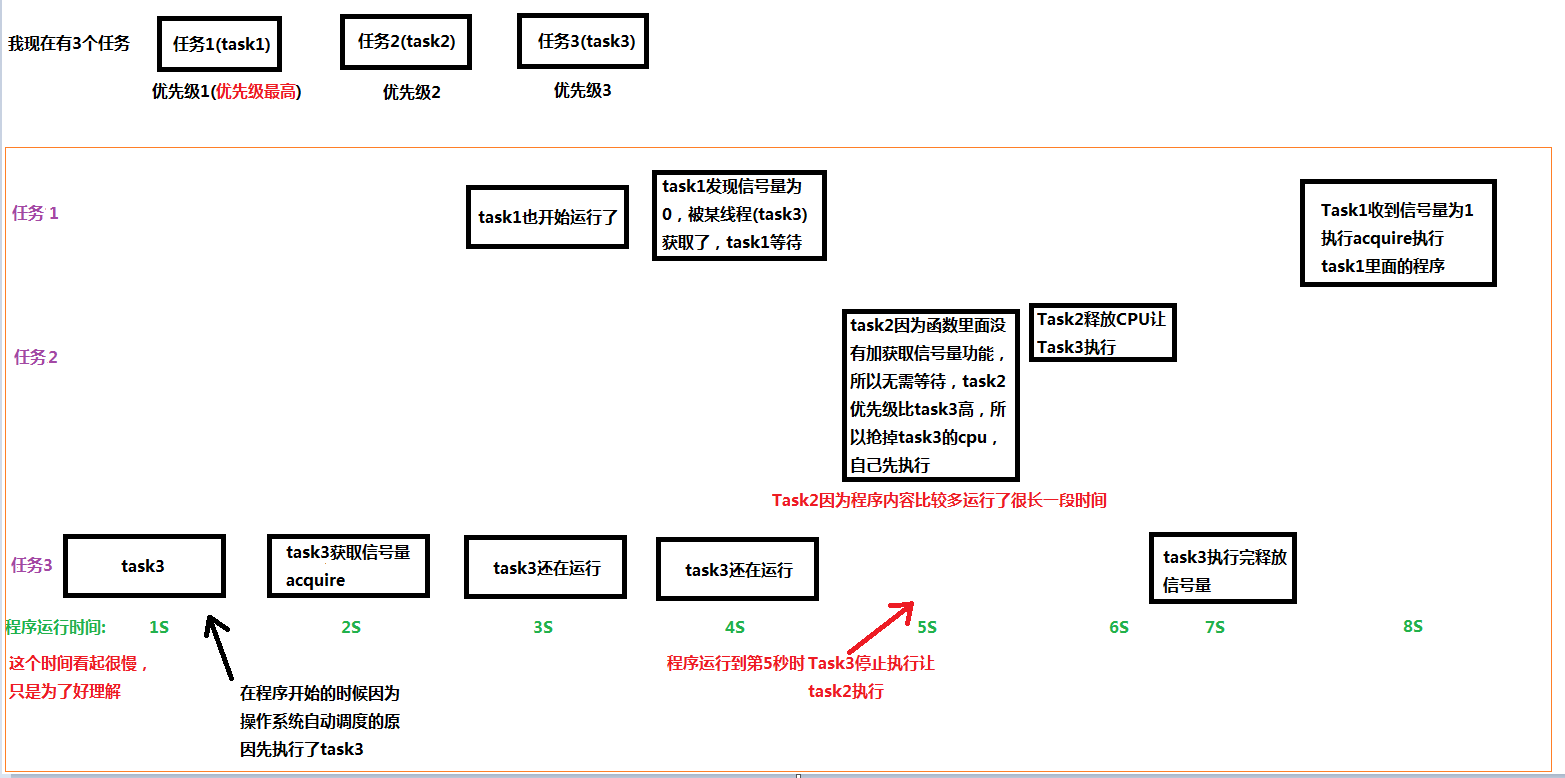


启动A工厂和B工厂线程，执行没有问题。

信号量和互斥锁区别

如果Task1里面是看门狗喂狗程序，那么看门狗运行你task2运行这么久，然后再让task1运行喂狗吗？，，我想看门狗不会等task2这么久的，看门狗会自动重启系统

信号量有一个优先级翻转问题

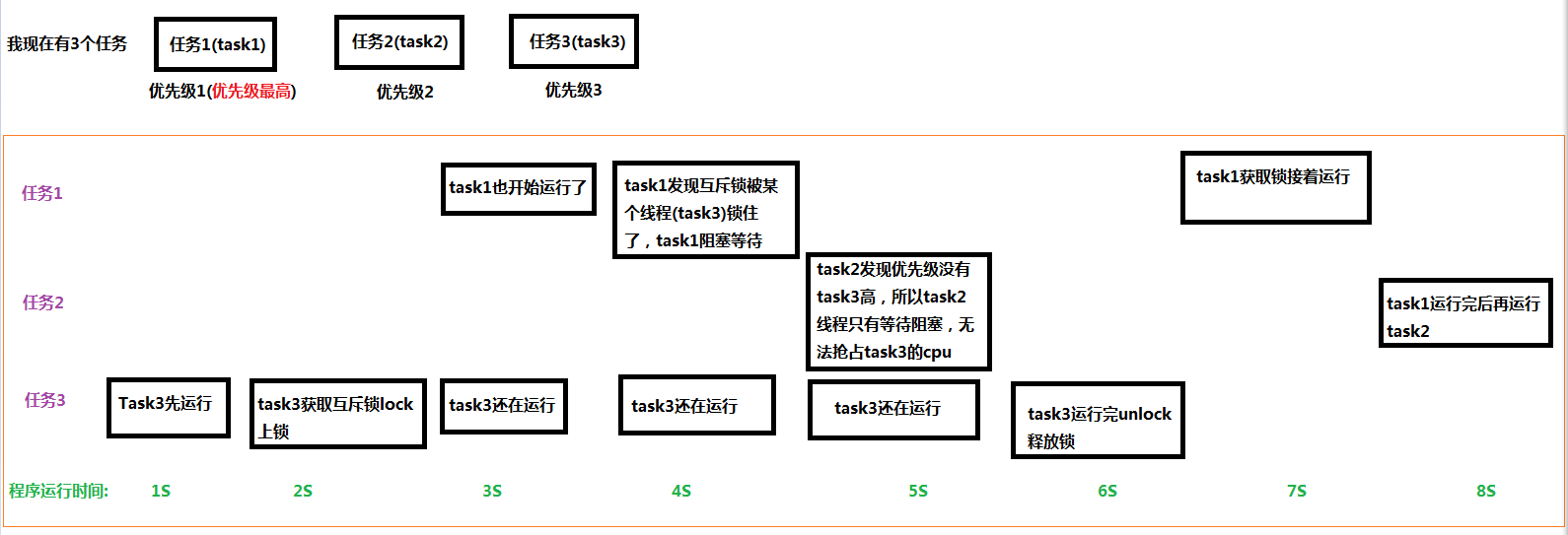


程序运行过程从左向右看

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。

注意:这里task1发现互斥锁被task3锁住了，那么系统会将task3的优先级提升为和task1一样的有优先级，那么现在系统最低的优先级是task2了，task2就无法抢占task3的任务

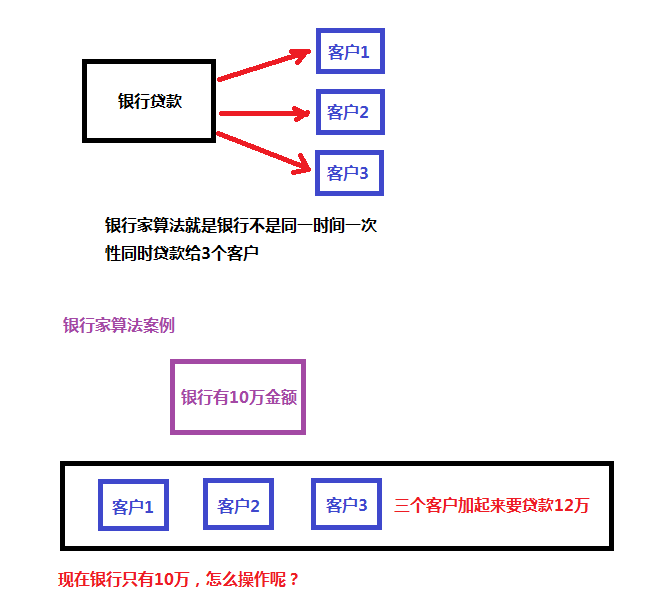
互斥锁解决了信号量翻转问题

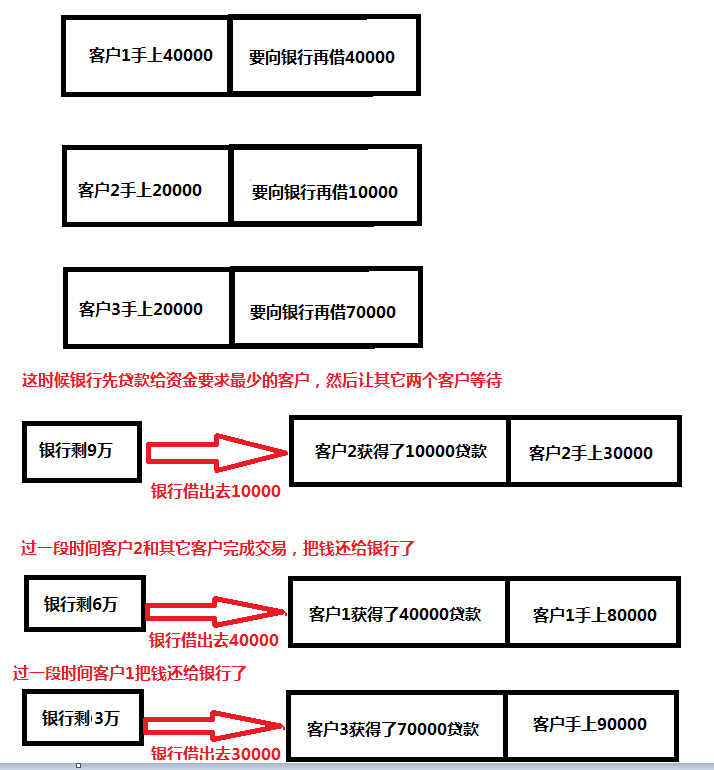


程序运行过程从左向右看

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**





所以银行家算法也是串行操作的。

设计思路:

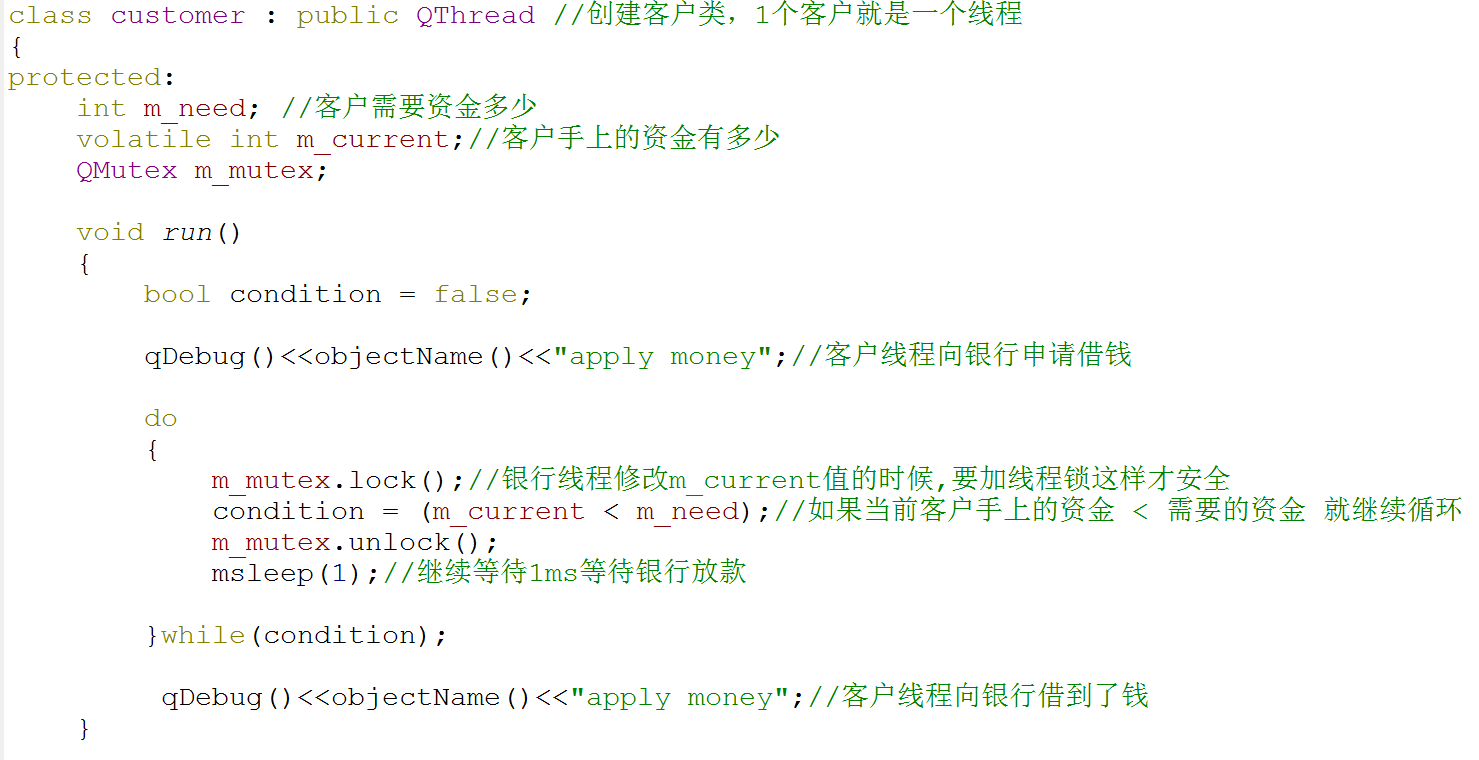
1.首先每个客户就是一个线程，所以要创建3个线程

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。

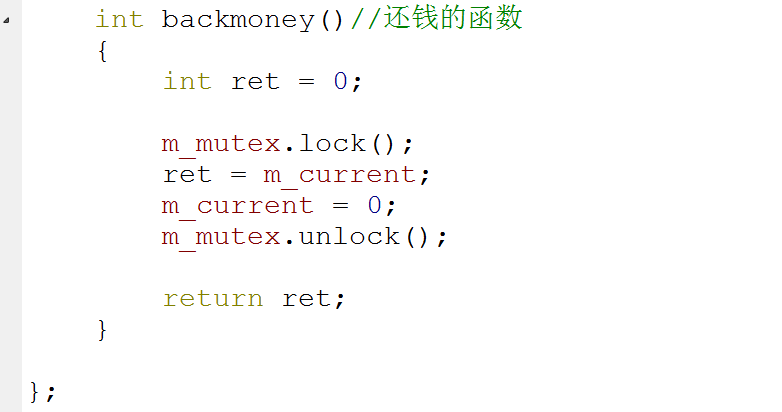


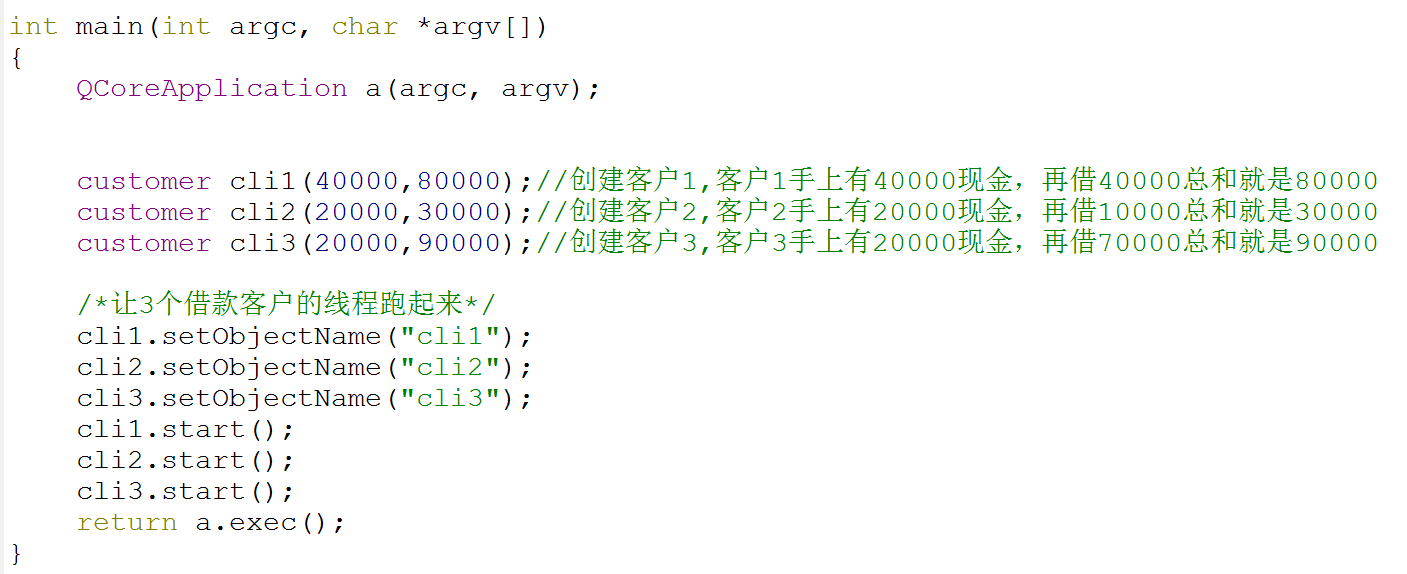
给每个客户创建一个这种类的对象，这样每个客户就是一个线程





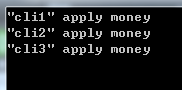
银行线程发现某个客户线程的交易已经完成了，就会执行客户线程里面的backmoney函数要求客户还钱，backmoney返回ret变量就证明还钱了







启动三个线程向银行借款，但是现在银行线程没有启动，所以三个线程都处于run里面的while死循环

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



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



一旦信号发送了，就有对应的槽函数来接收

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

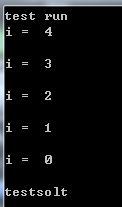
线程运行函数来模拟线程启动和结束



线程发送信号后对应connect连接的槽函数就会响应

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

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

线程循环执行完发送了信号，导致槽函数打印了

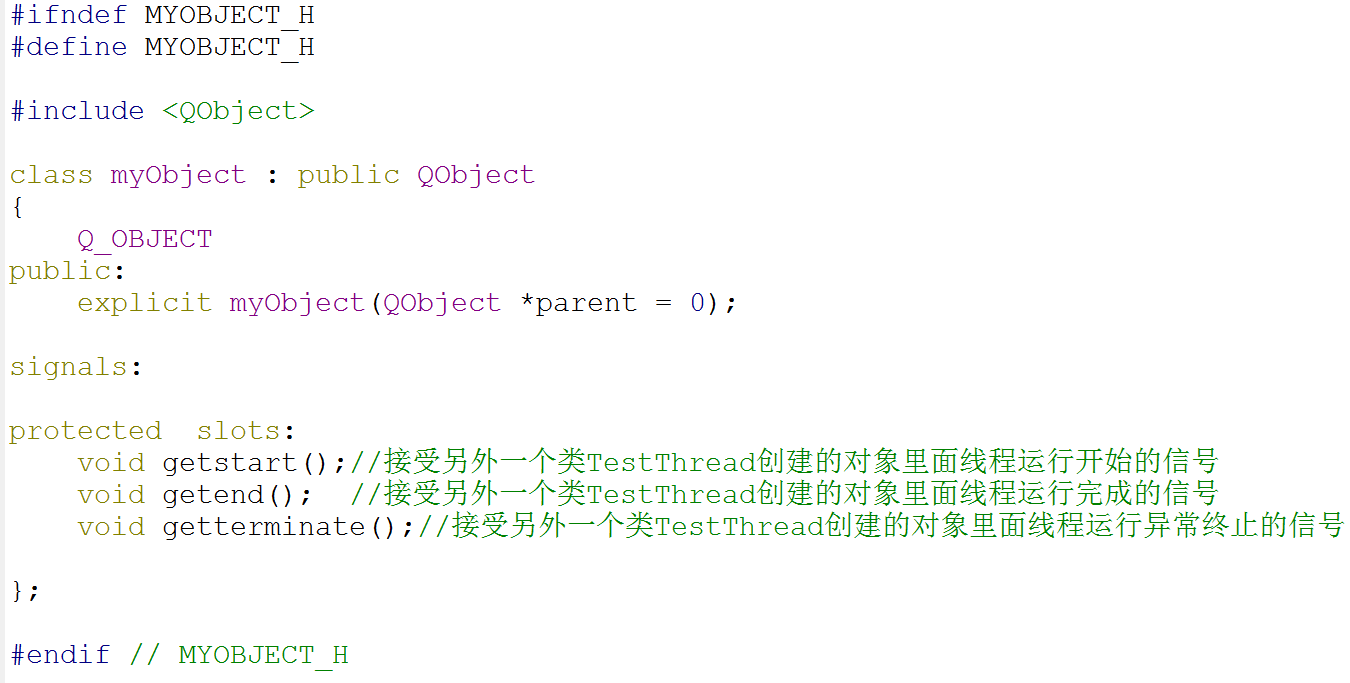
启动线程

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。

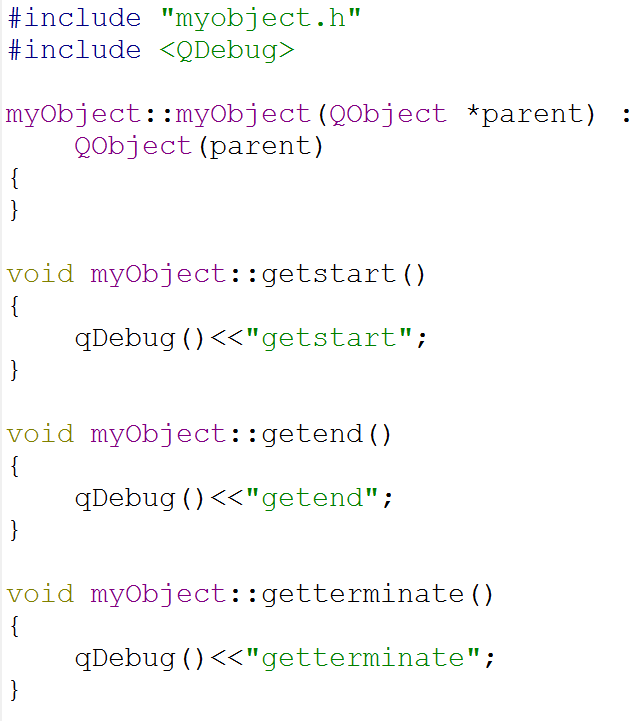


沿用上页的Testthread类函数和对象t

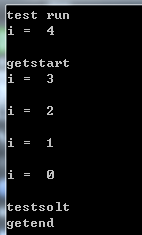
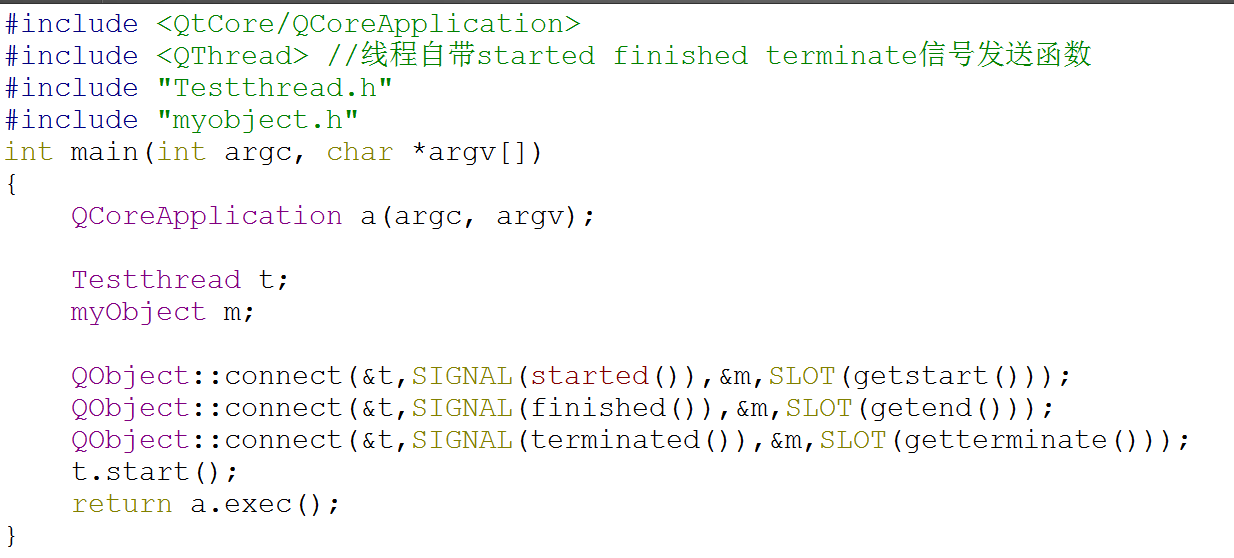
下面创建一个普通类来接收Testthread对象发过来的信号



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



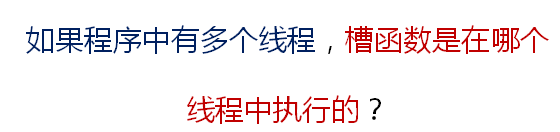
普通类m收到了t对象的线程结束信号

普通类m收到了t对象的线程启动信号

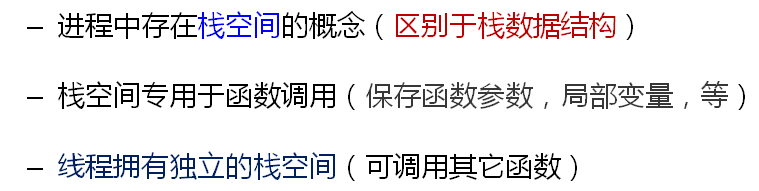
记住这里填的是线程默认信号函数

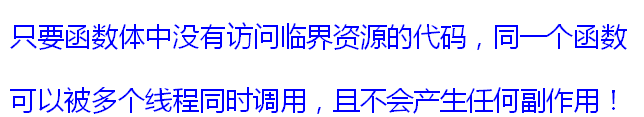
用connect将testthread对象t发送的信号和myobject普通类m实现的槽函数链接起来

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。

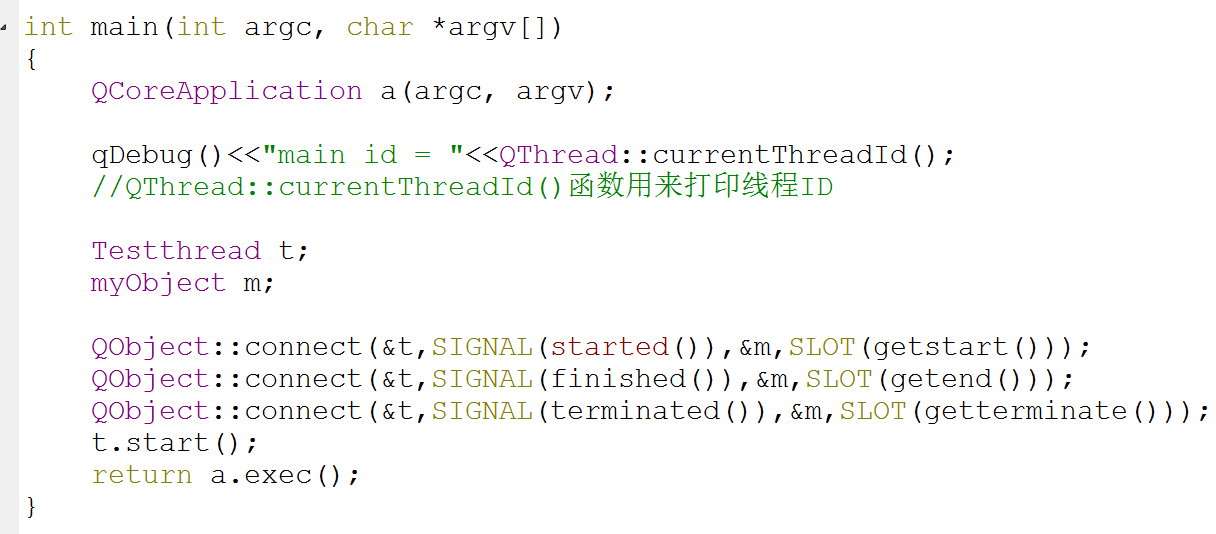


答案是:

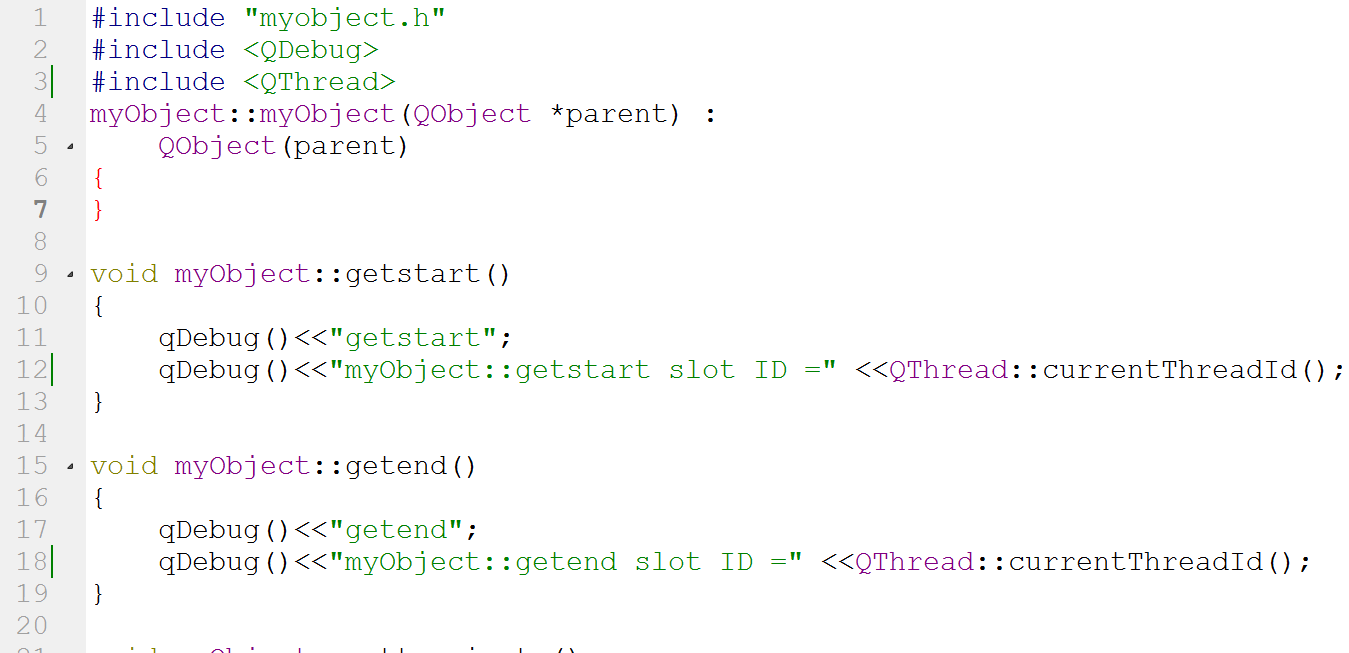




但是线程类创建的槽函数不一定在该线程类中执行

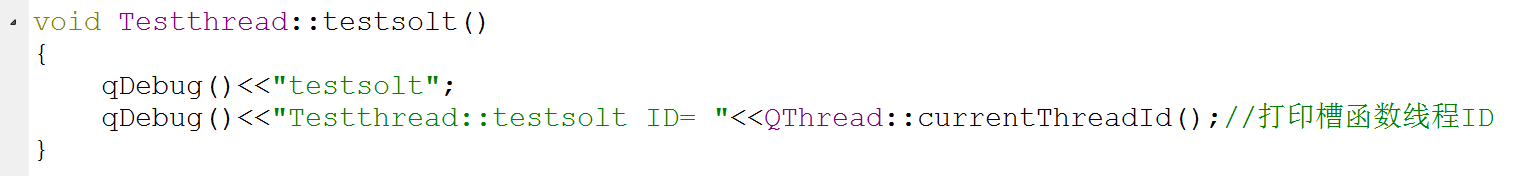


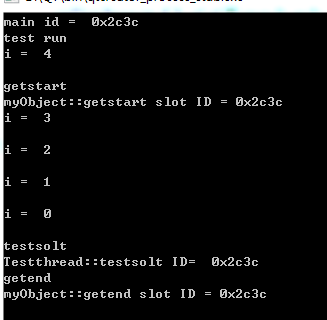
打印主函数的线程ID



打印另外一个主函数创建出来的独立的线程类的线程ID

打印myObject类函数里面的线程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。

其它类创建的槽函数也是主线程的线程ID

主线程ID是0x2c3c

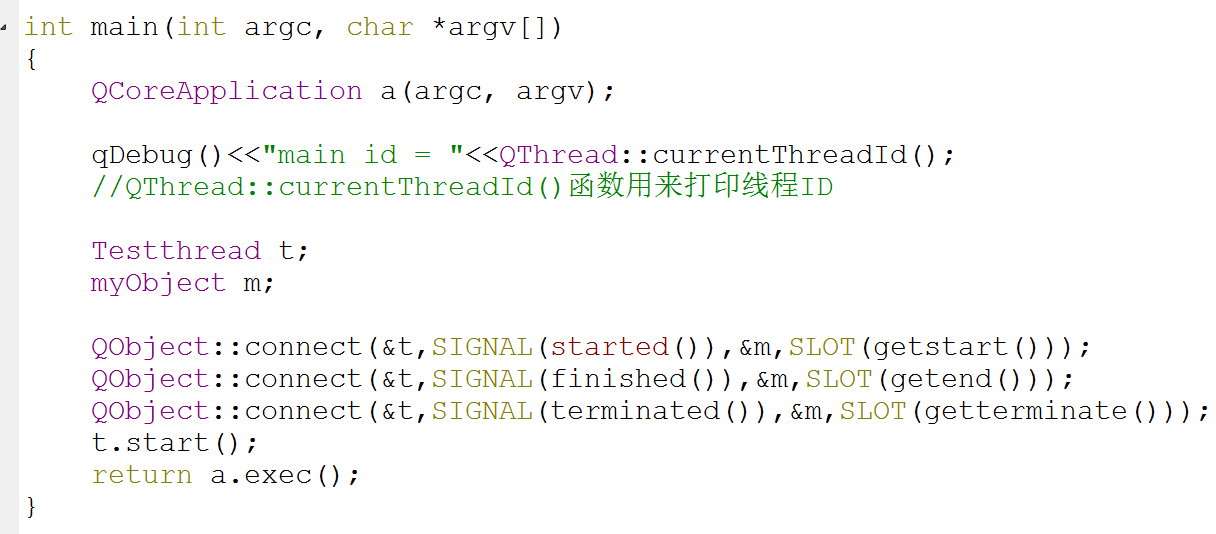
**线程类定义的槽函数怎么没有在自己的线程类里面执行？**

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。



那么t和m对象里面创建的槽函数就依附于主线程执行

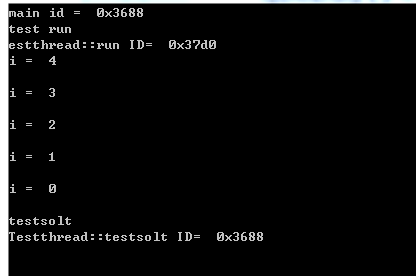
比如主线程main函数创建了对象t和m

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。



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了，而在t线程



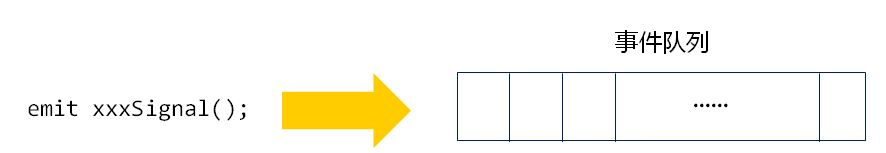
但是另外三个m对象的槽函数没有执行，m对象的槽函数是接收t线程开始信号和t线程结束信号的

我们运行发现t线程槽函数执行了

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

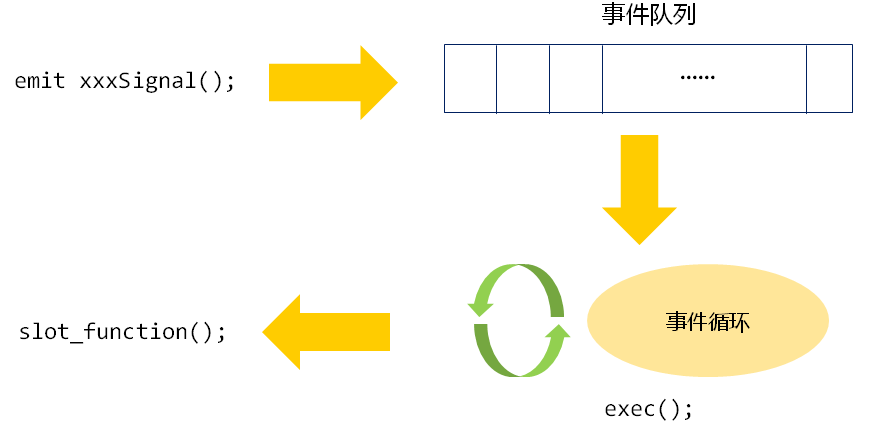
这是因为线程中的事件循环函数没有启动

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()；

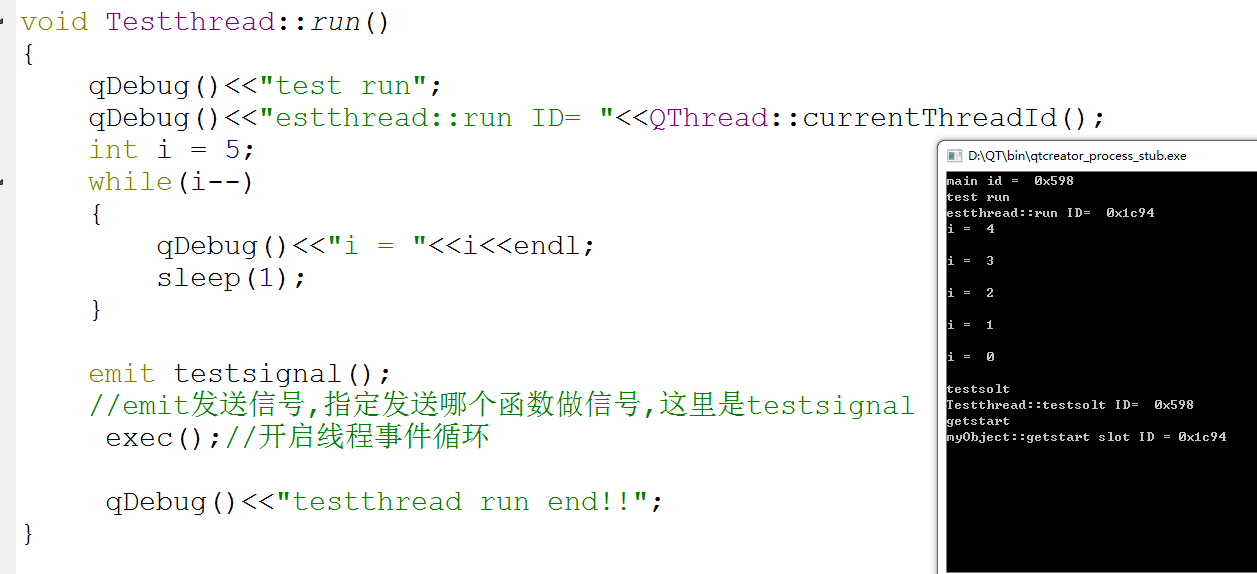


线程的开启和结束都会发送信号到这个事件队列

问题是没有东西去取事件队列里面的信号来确定到底该执行哪类槽函数呢



这时候就只有靠事件循环函数exec();来执行去取事件队列里面的信号，然后根据信号类型去执行对应的槽函数



这样线程链接的槽函数执行了,但是执行的是getstart，线程开启的槽函数

在线程中加入事件循环exec()函数

而且因为m.moveToThread(&t)的原因，槽函数现在已经依附于子线程ID执行了，不在依附于主线程

这个exec()函数是个事件死循环函数，它会不停的是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。

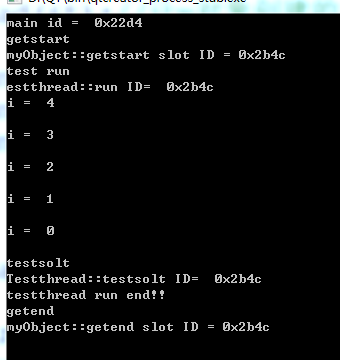
如何处理exec()事件死循环?

用quit函数来结束指定线程。



加入wait等待4秒，然后执行quit线程结束

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

但是我的exec必须靠主线程的quit来结束好麻烦

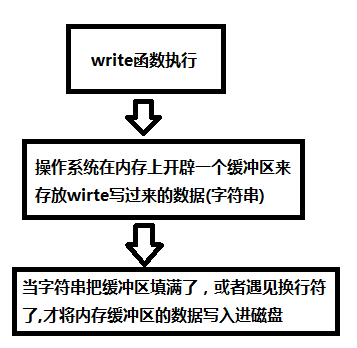
t线程的exec()循环结束，执行线程结束槽函数。

t对象依附于t线程了

一般事件循环exec用来处理IO操作，比如write文件，read文件。

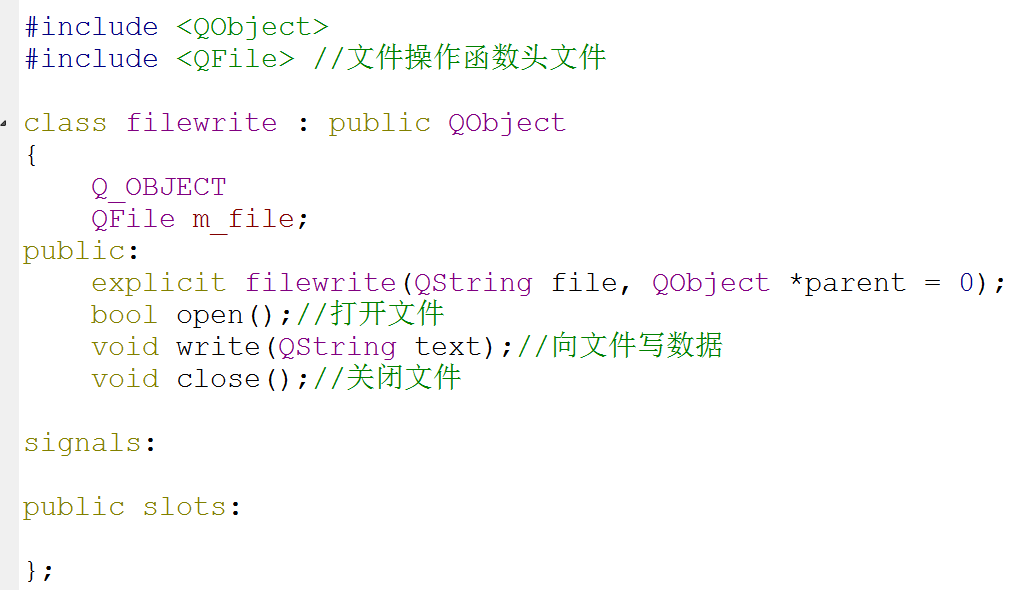
exec操作流程:先开启事件循环，在写write文件的时候，发起写信号函数，然后写write程序在子线程中进行

write写文件操作系统是如何执行的?



为什么要将数据先写入缓冲区，然后再写入磁盘?这是因为磁盘是块设备，每次写512字节或者1024字节效率才高，但是write写入的数据是一个字节一个字节写的，所以会影响磁盘效率和寿命。可以查看我的STM32编程指南，或者linux驱动手册，里面介绍了SD卡类的磁盘原理。

案例: 我们做一个向文件写数据的系统



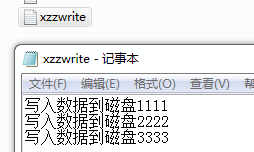
我们将打开文件，写文件数据，关闭文件用类和函数封装起来



我们每次写文件到内存后，直接就将内存的文件数据写入磁盘



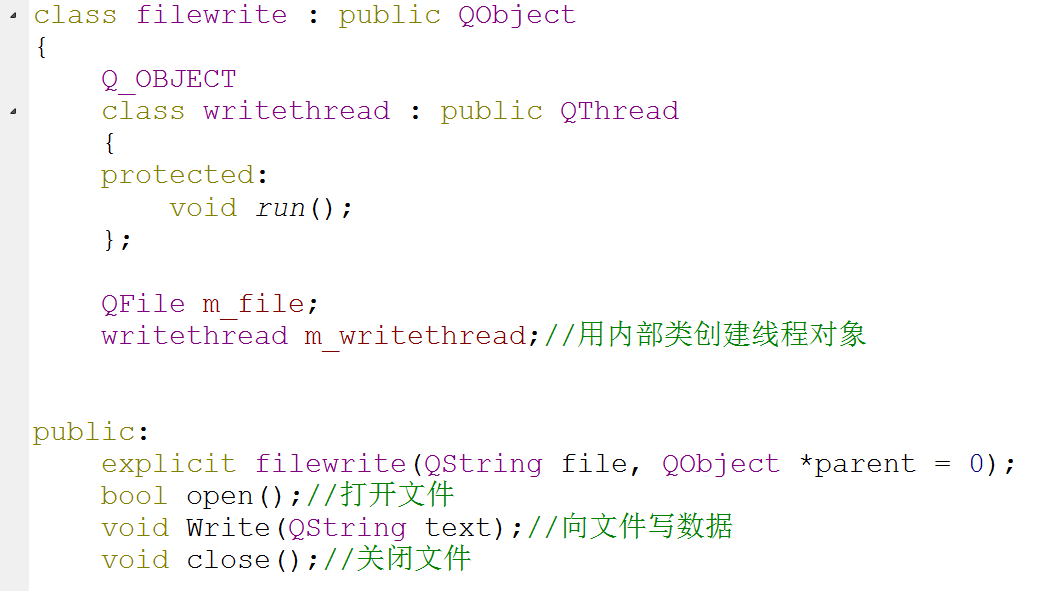
这里就有个问题了，就算程序运行没有问题，但是在主线程每次向文件写数据都是用write，导致flush函数执行，flush函数是向磁盘写数据，所以每行write函数都执行得比较慢，这样就会影响主线程执行其它程序的效率

数据写入磁盘



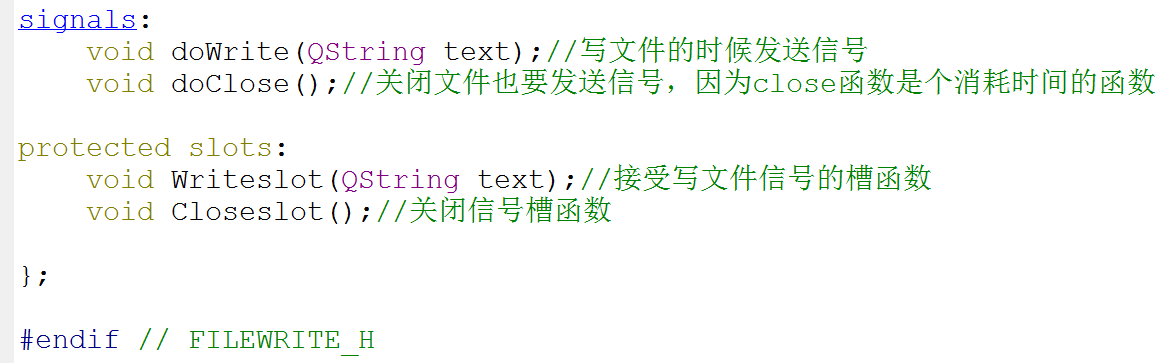
程序执行成功

所以文件写入磁盘存储的程序需要做优化，下面就来优化



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

我们在类里面创建一个内部类来做线程



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

这个信号就是给write和close函数使用的，这样write和close在主进程里面就执行得很快

在槽函数里面实现向磁盘写数据的write，flush，close

连接写文件信号到槽函数里面，close关闭文件也是



写文件要耗时间，主进程write发信号交个槽函数去处理

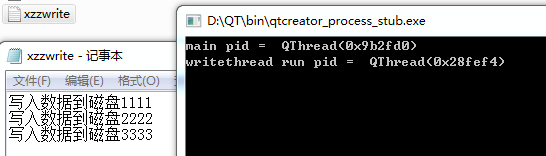
打开文件不耗时间，所以交给主进程open执行



我们知道要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

Close也是消耗时间的函数，所以发信号交给槽函数去处理



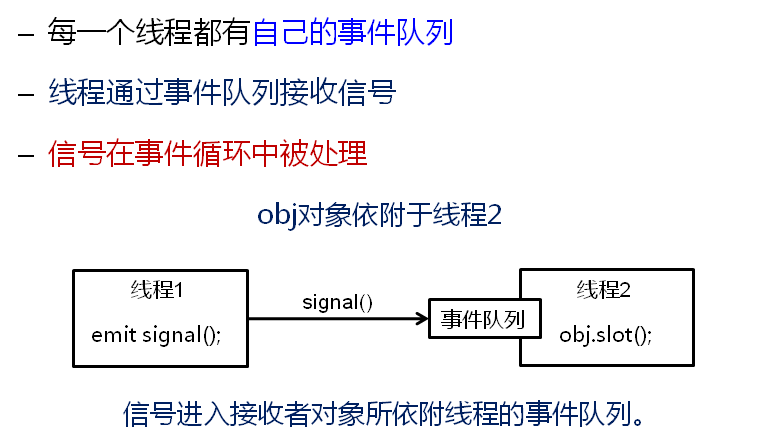
程序执行后文件创建成功，内容也写入磁盘了，但是程序卡在这里了

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？

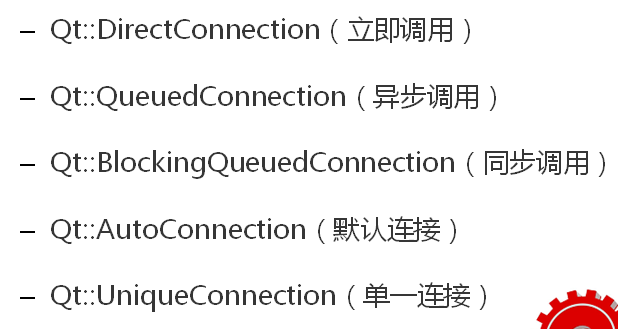
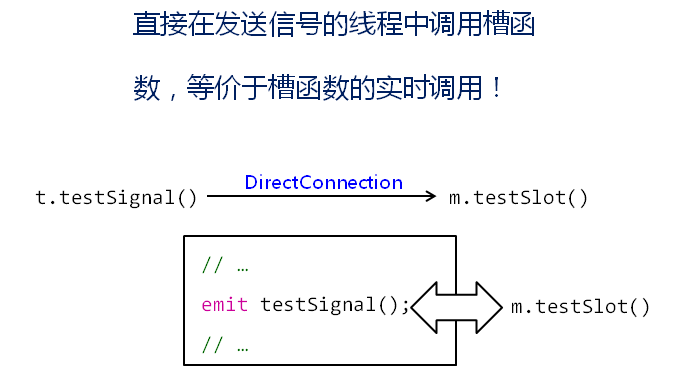


线程1发送信号

根据connect函数中信号与槽链接的链接方式，我们知道signal信号链接到了obj这个对象上的槽函数

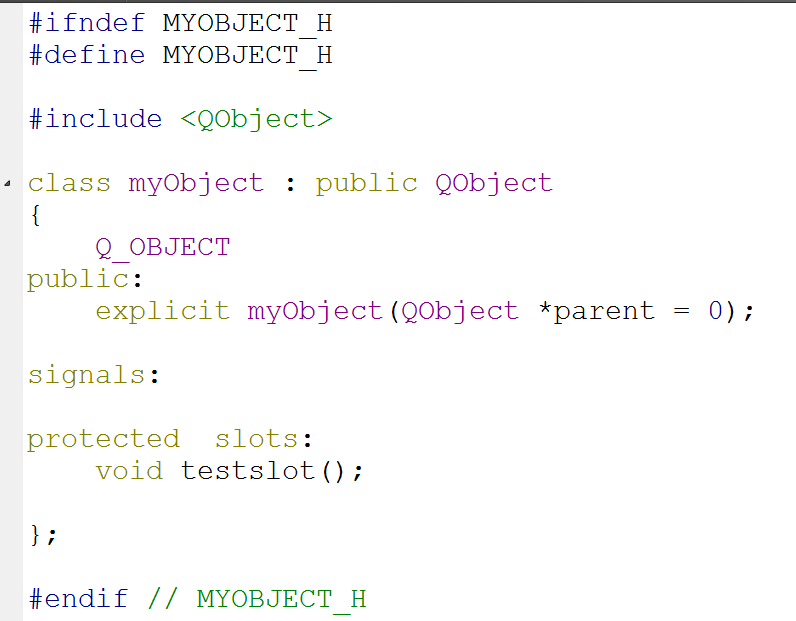
根据线程的依附性，obj这个对象依附于线程2，那么obj对象的槽函数也就属于线程2，那么signal发送的信号就存放在线程2的时间队列里面。

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。



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

下面我们来做实验

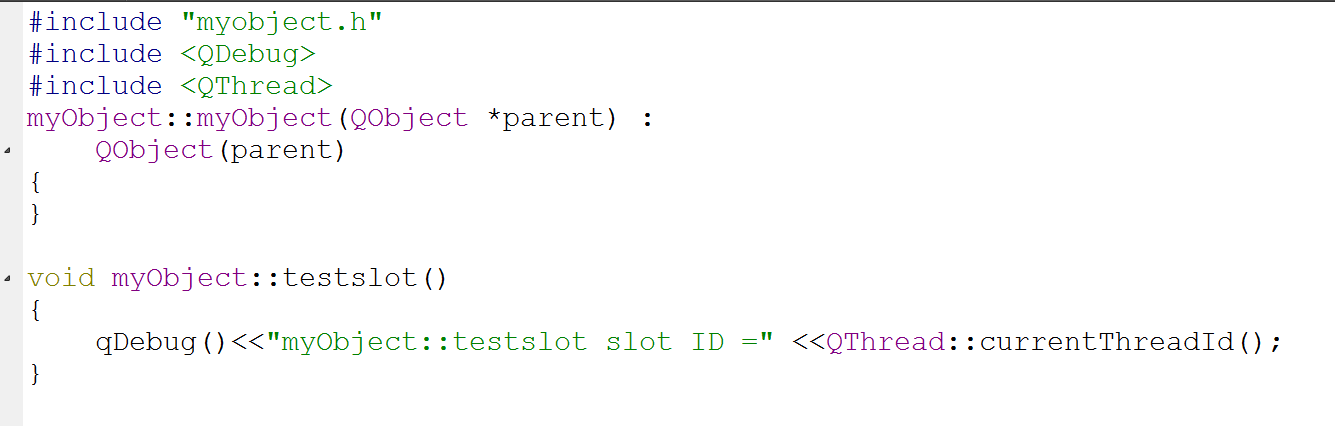
 

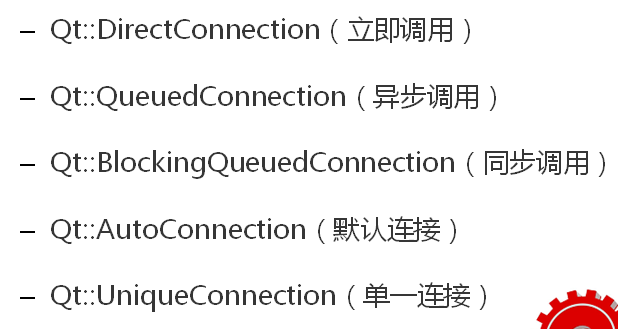
创建发送信号类头文件 创建接受槽函数头文件

实现发送信号类函数

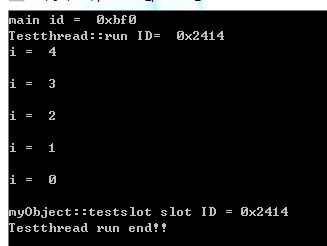


实现接受信号类函数



我们使用connect链接信号和槽的时候用Qt::DirectConnection模式链接





发现槽函数的线程ID是t线程的ID，而不是主线程的ID，

t线程执行，然后发送信号



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

因为我的m对象是在主线程创建的

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？



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

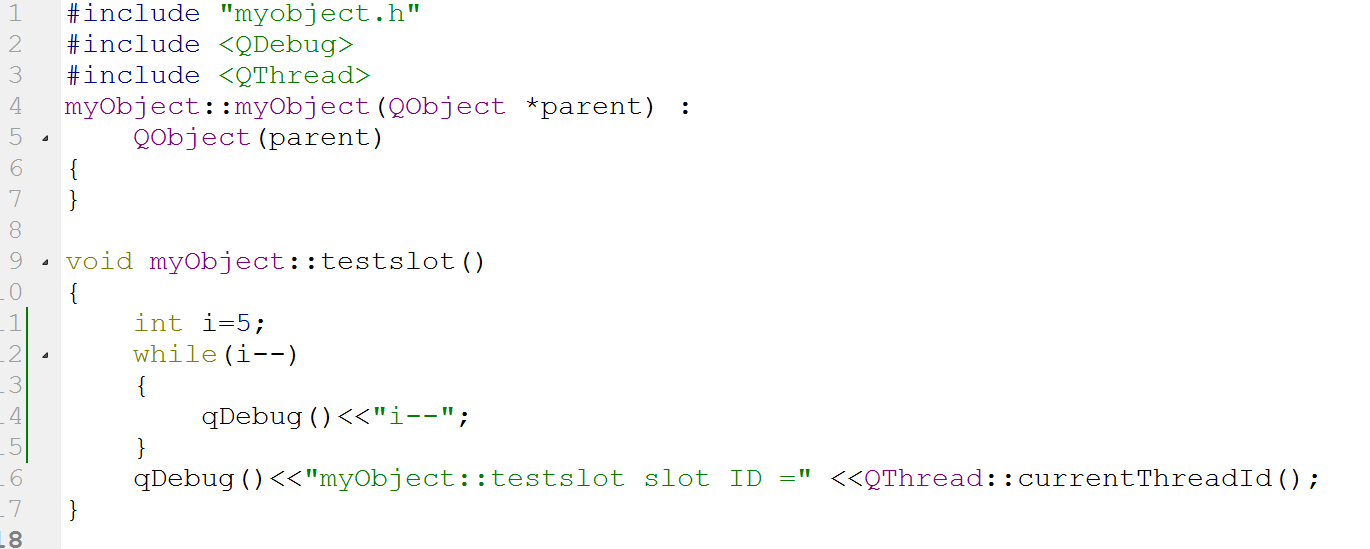
意思就是:



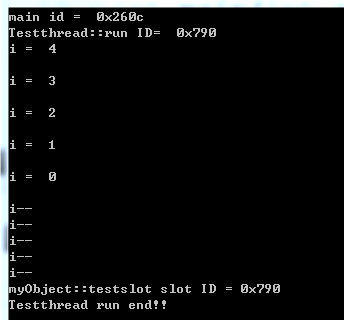
Qt：：DirectConnection就是信号发送的同时执行槽函数，也就是把槽函数放在信号所处的线程来执行

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。

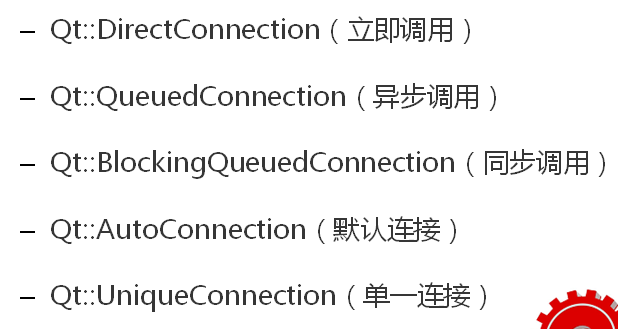
像槽函数创建在m对象，信号在t线程对象，这种情况就算t线程使用emit内部将slot放入t线程执行也不会出现bug，因为创建信号和槽的是两个对象 

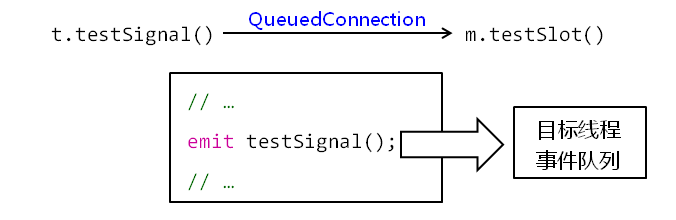
我在m对象槽函数加入循环，我们看看线程t是不是要等待m对象的槽函数执行完了，才回到t线程继续向下执行。



你看t线程最后一个函数要等槽函数执行完了才能执行，所以槽函数依附子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

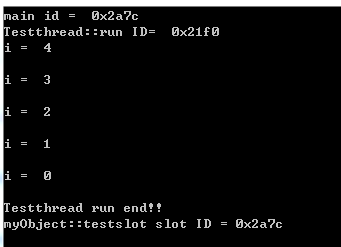




Qt::QueuedConnection属性的意思就是t线程.发送了emit信号，然后t线程不会等待槽函数执行完了后继续向下执行，而是发送信号之后就直接向下执行



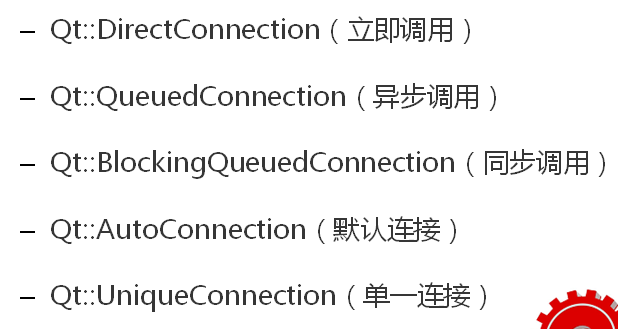
Connect同步调用



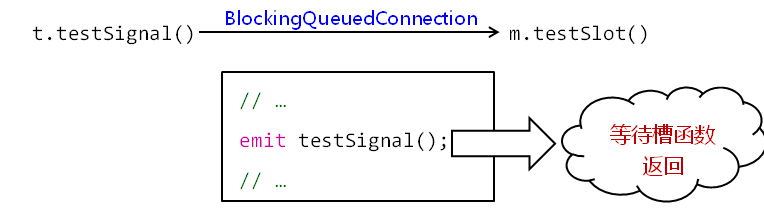
Qt::QueuedConnection还有个功能就是让emit不能向Qt：：DirectConnection这样直接将槽函数替换到发送信号的线程里面来执行，这样发送信号的线程要等待槽函数执行完了才能继续向下执行，比如t线程。所以Qt::QueuedConnection让槽函数依附于创建它的对象线程，这里是m对象，m对象是主线程创建的，所以是主线程来决定什么时候执行槽函数

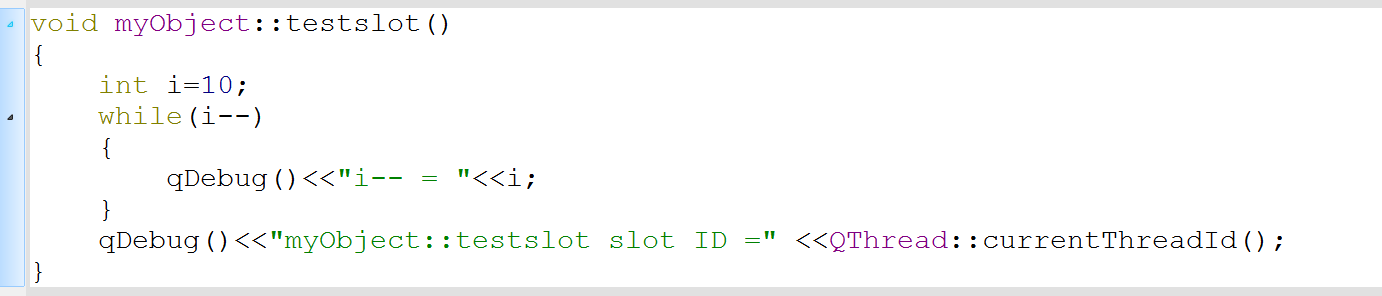
你看t线程发送信号emit之后直接执行到t线程结束，至于槽函数什么时候执行是槽函数依附的对象线程决定的

下面看看同步调用

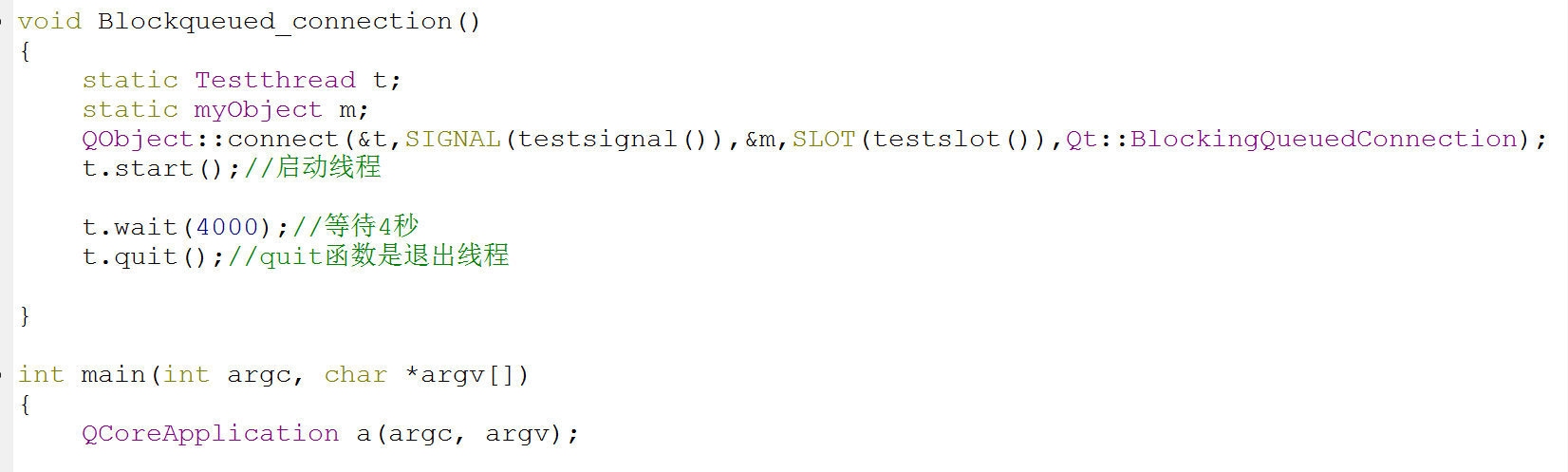


BlockingQueuedConnet的功能和DirectConnection的功能差不多，都是线程发送信号后，要等待槽函数执行完了，才回来执行该线程，但是这里一个唯一的特点，看下面

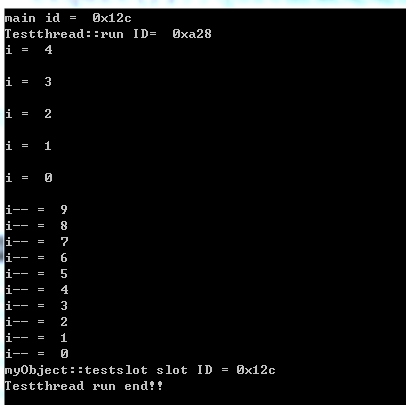




打印10次，这样好和上面的DirectConnection例程区分



Connect使用Qt::BlockingQueuedConnection同步调用属性

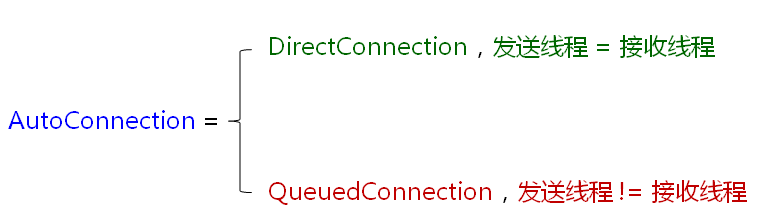


DirectConnection例程虽然信号发送之后要等待槽函数执行完才能继续执行当前线程，这是因为槽函数依附于子线程ID。但是BlockingQueuedConnection可不是依附于子线程ID，而是依附于创建对象的线程ID，testslot是m对象创建的，m对象是主线程创建的，所以槽函数依附于主线程ID，只不过BlockingQueuedConnection让槽函数在分离t线程ID的同时又具备信号的跟踪性，就是不管哪个信号，只要emit了testslot这个槽函数，那么该emit的线程就要等待槽函数执行完了才能接着向下执行

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



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



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



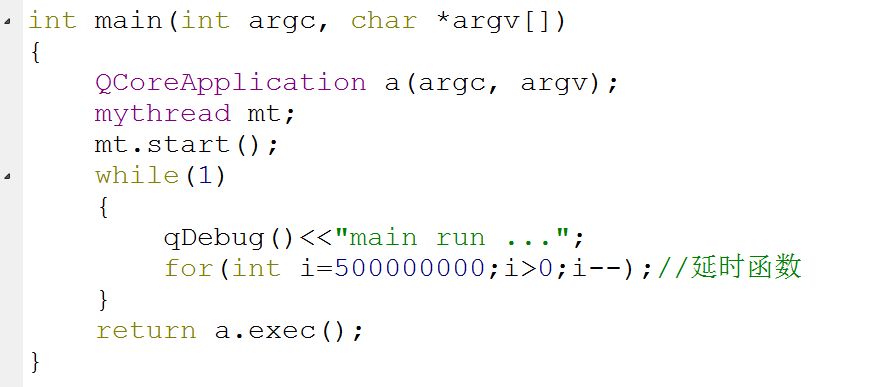
这很明显槽函数还没执行完线程就先执行完了，所以线程不用等待槽函数执行完,证明QT选择的是QueuedConnection

不加参数就是Autoconnection，QT系统自动选择

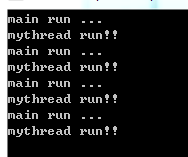
**对象创建的线程生命周期是否会随着对象执行完后线程跟着消失，该问题出现在子函数里面创建对象的情况**



我们按照常规方法设计个线程类



在主函数创建线程对象

下面我们来修改下线程对象创建方式

主线程和mt对象的线程同时执行正常



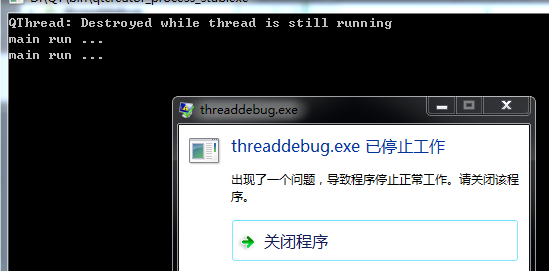
在子函数里面创建对象



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

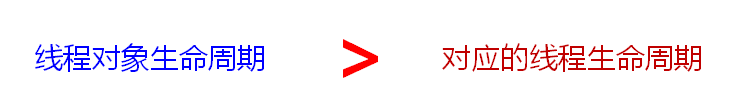
执行到这里的时候子函数creat\_thread已经执行完了，子函数里面的对象会自动释放，这时候子mt线程还在吗？

主线程调用子函数，子函数入栈



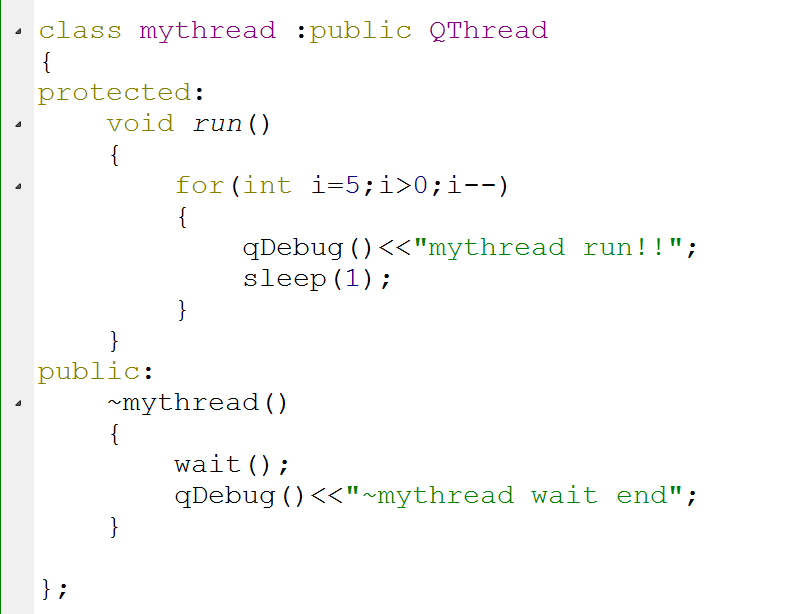
如何解决对象比对象对应的线程先执行完成的情况呢？

编写多线程有一条准则



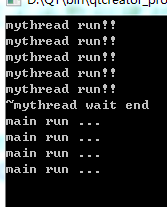
就是你要评估你的对象存在时间一定要大于你对象创建的线程运行时间，必须是对象的线程运行先结束，然后对象才能结束。

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。



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

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



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。



3.在创建线程的子函数中，执行NewInstance在堆空间创建mt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。

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。

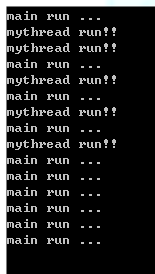


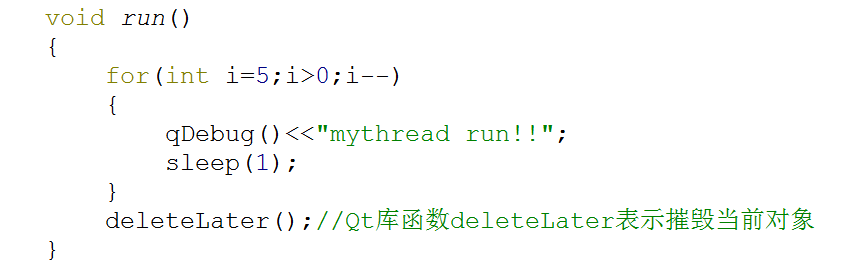
4.这里执行delete会报错，因为析构函数是protected的，就是不准你在这里释放掉堆空间对象，如果这里释放掉堆空间对象，那么和前面的问题一样了，creat\_thread函数先执行完，释放mt对象，然后mt对象的线程没有执行完，系统崩溃



6.creat\_thread()执行完后，mt对象在堆空间没有被释放，那么mt线程和主线程并行执行

5.取消delect，就这样执行creat\_thread函数，这个函数执行完后，mt对象因为在堆空间的，所以没有释放掉。Creat\_thread只释放了栈区的数据。



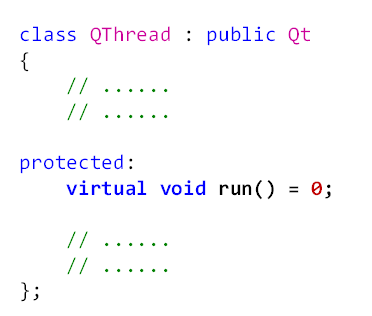


这里是因为mt对象的run函数执行5次循环后，线程就结束了，用deleteLater释放掉了mt对象。所以就只剩下主线程执行了。

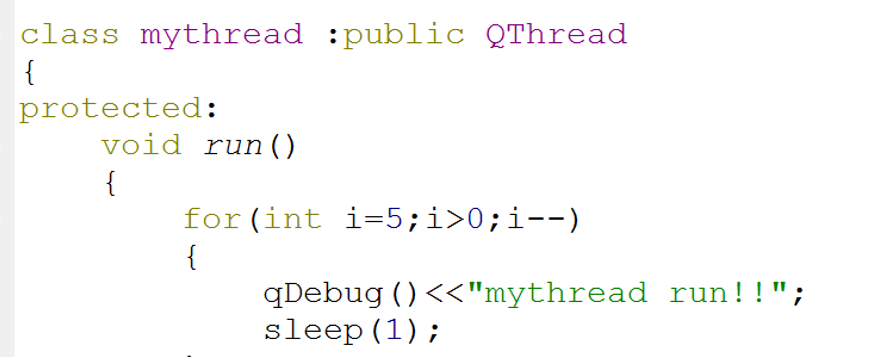
你看主线程和mt线程是并行执行的。这样就没有wait那样麻烦了。

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**

那么既然Qthread原形是虚函数来构建的run，那么有什么问题呢？

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

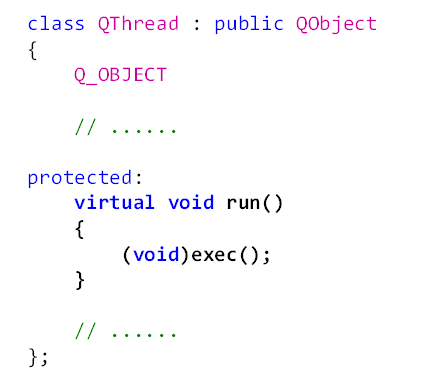


比如我想建立个类，实现个新的功能，那么就要去继承至Qthread

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



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



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

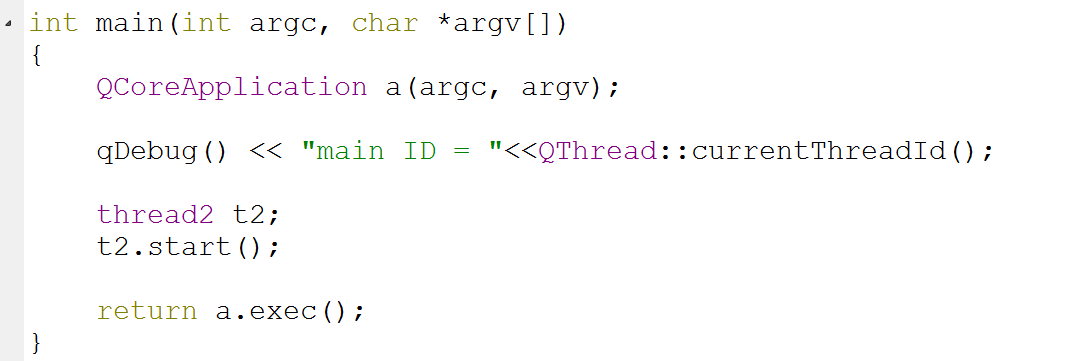


7.启动m\_thread线程，就会触发信号started，因为信号链接的是槽函数，所以槽函数会被执行

6.这样槽函数就类似run函数了

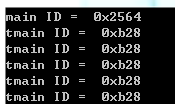
5.把槽函数链接上这个线程对象

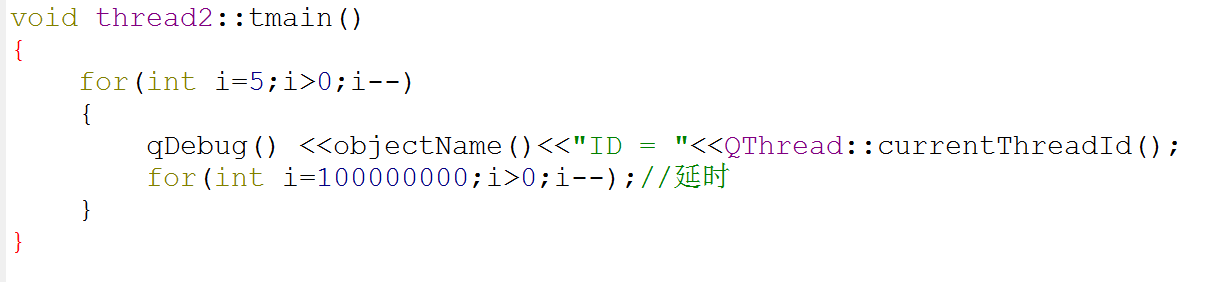
4.这就是让普通类变成线程类的关键，将线程对象依附在普通类，因为普通类要有线程类的效果也得靠线程对象间接驱动



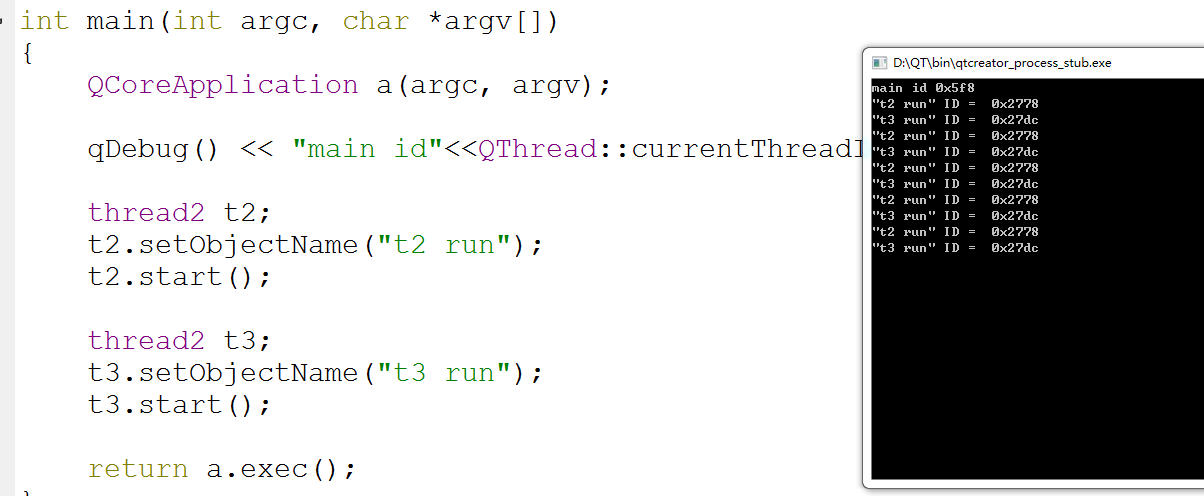
创建普通类thread2的对象t2

启动线程对象m\_thread

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



打印线程对象名称

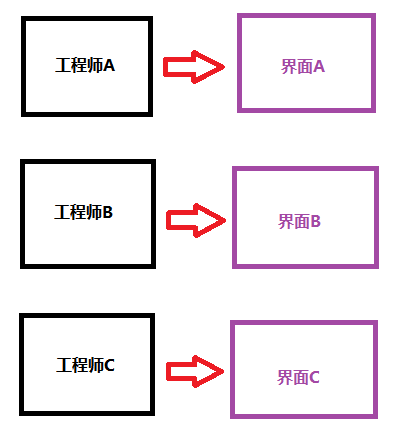
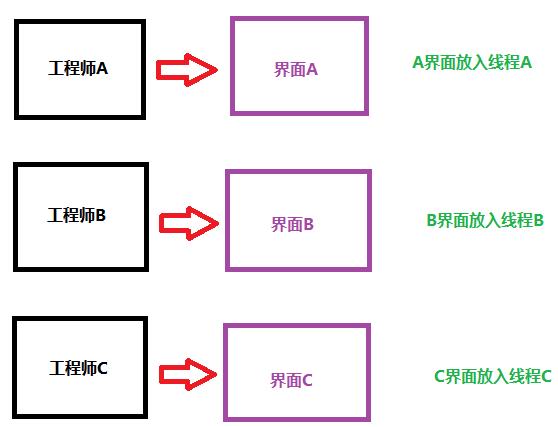


你看两个对象同时执行，根据线程ID就能看出来，t2，t3是并行执行的。

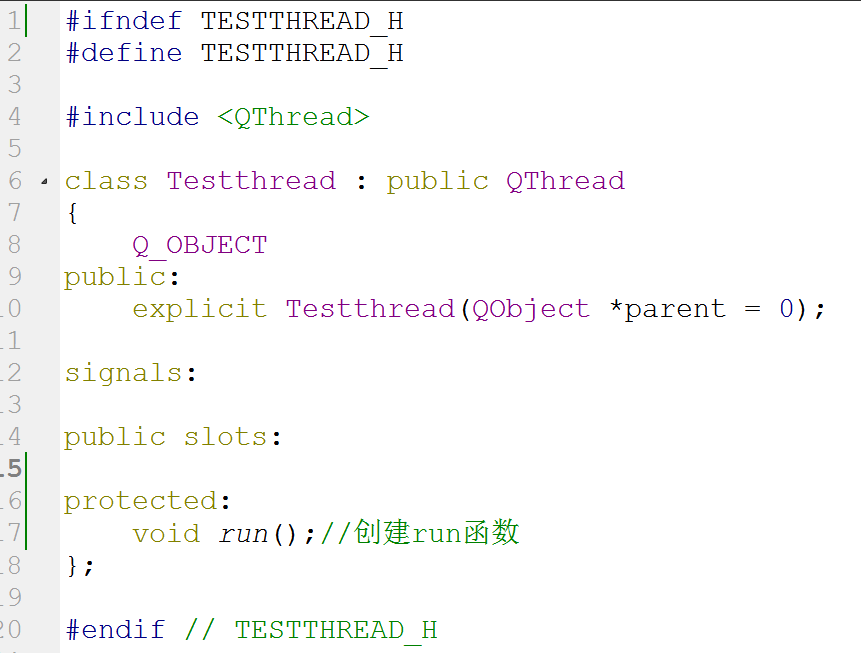
所以这种非继承Qthread的类也是多线程实现的一种方法。这种方法是固定的，你就按照一个线程一个对象就是了，不需要考虑到父类继承了Qthread这种麻烦事

**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？

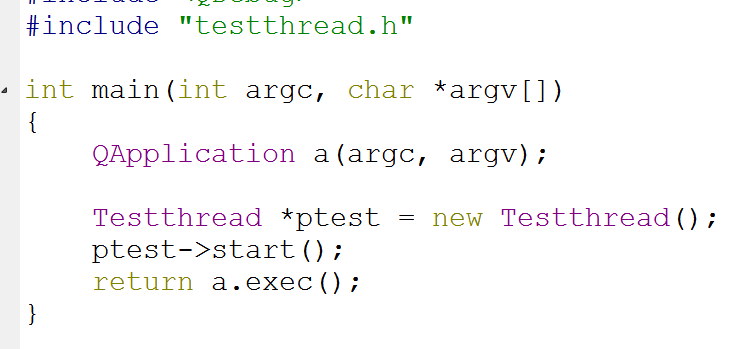
 

我们给ABC界面加入线程。

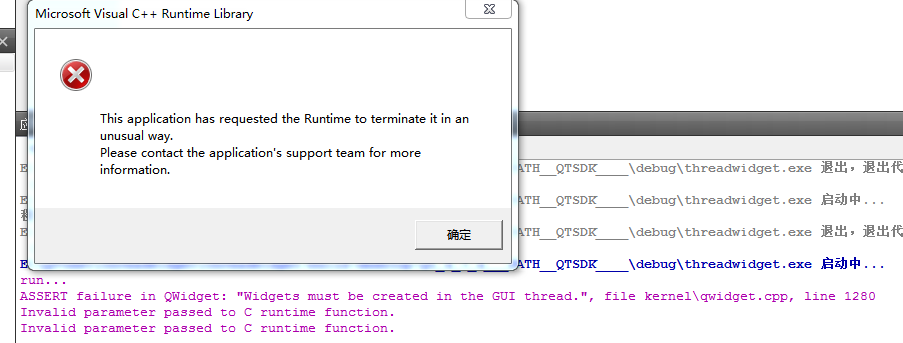
 

在这个子线程里面创建一个界面，这就是我们要给A工程师的界面开辟的线程

头文件创建一个线程



启动子线程来创建界面

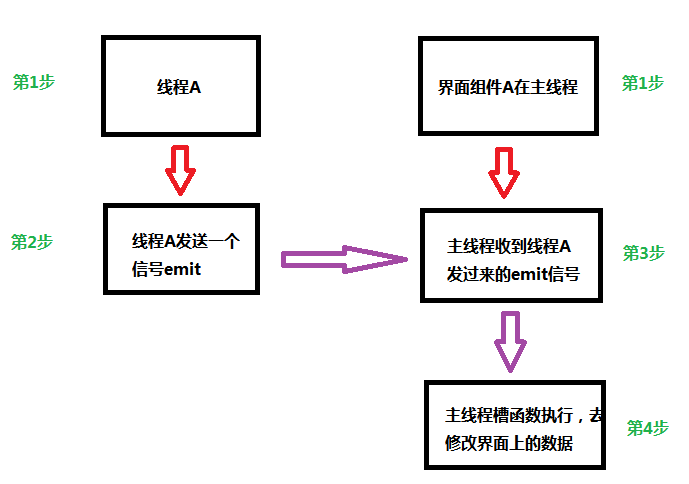


错误打印要求界面创建必须在GUI线程里面创建，意思就是要创建界面，必须在主线程里面创建

程序运行崩溃了

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。

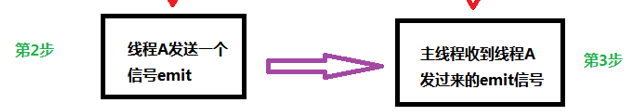
所以如果你想用子线程去修改界面上的数据必须遵循以下方法:

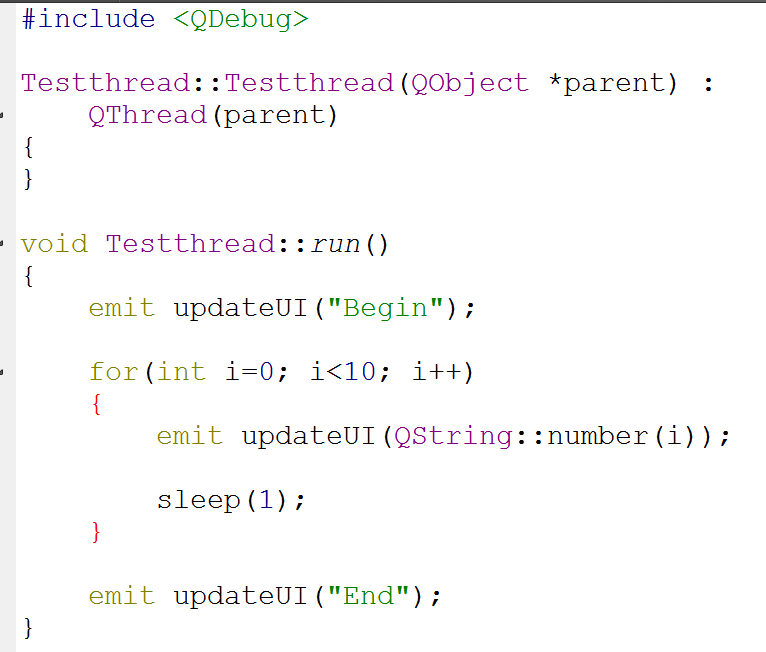
必须是用子线程的信号去修改主线程界面数据，就是这么麻烦

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。

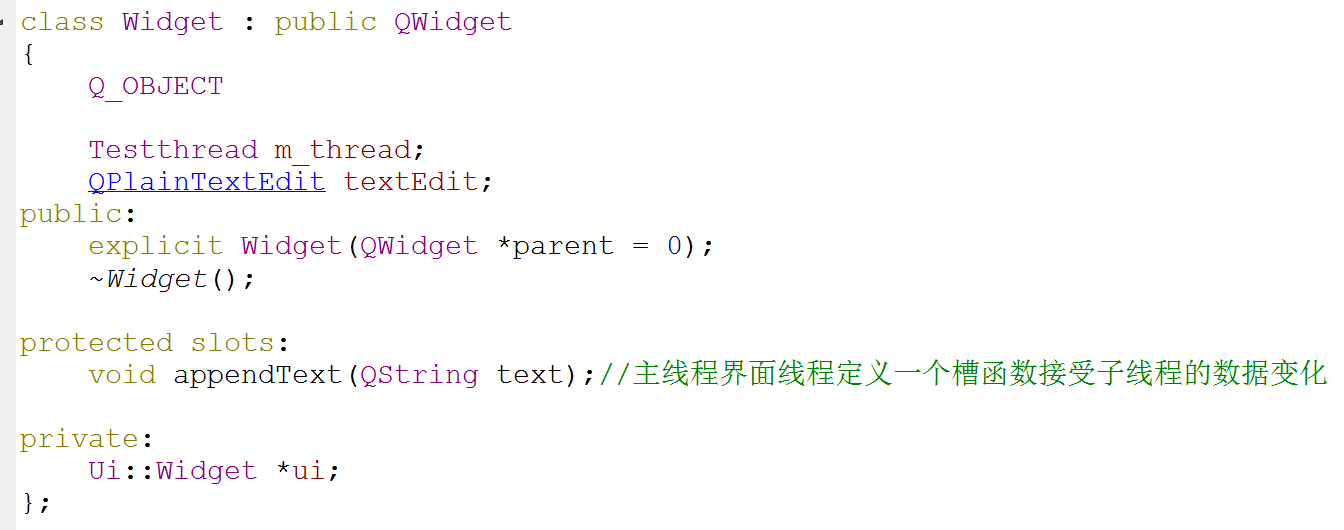


给Testthread类增加了一个发送信号给主界面的函数





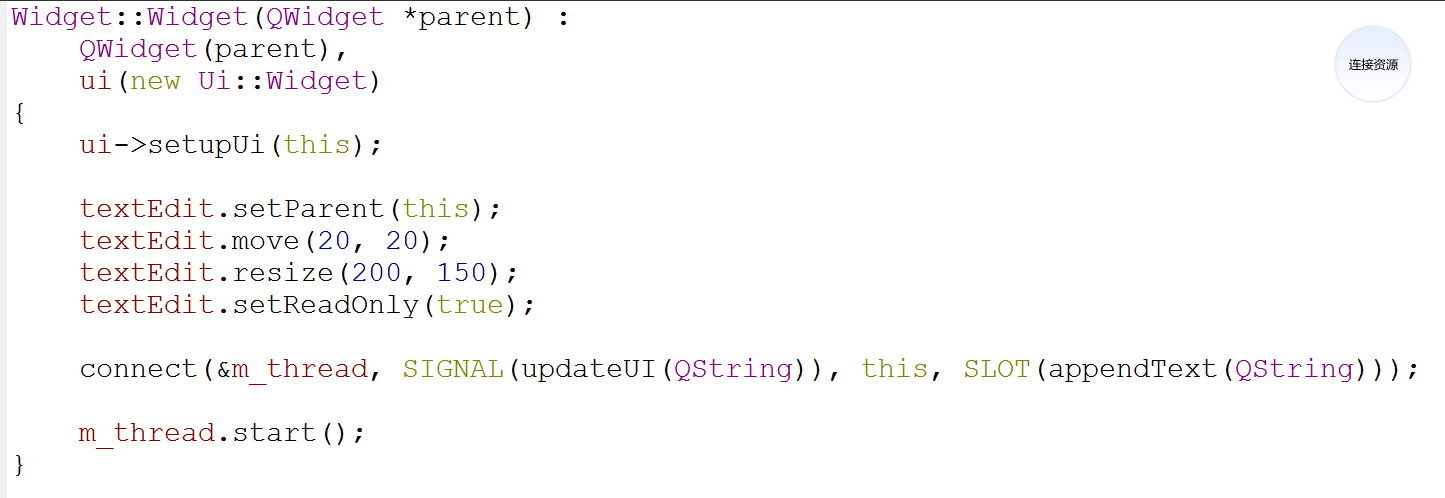
用emit不停的发数据给主界面(主线程)

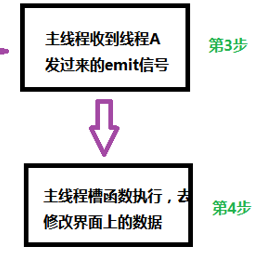


用文本框来显示子线程对象发送过来的底层数据

定义一个槽函数来接受子线程发送过来的emit信号数据

主界面创建子线程对象





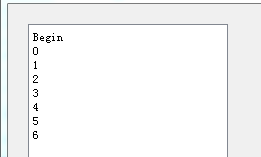
槽函数接受到m\_thread线程数据，去更新界面

子线程信号函数

放入子线程对象

文本框设置

启动子线程，发送数据

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。