1. 线程的创建方式及几种状态

1、创建方式

实现线程的3种方式：继承Thread类、实现Runnable接口、通过实现Callable接口和Future接收返回结果创建线程。

2、线程的状态

①新建(new)

②就绪 (调用了start()方法，表示等待资源分配)

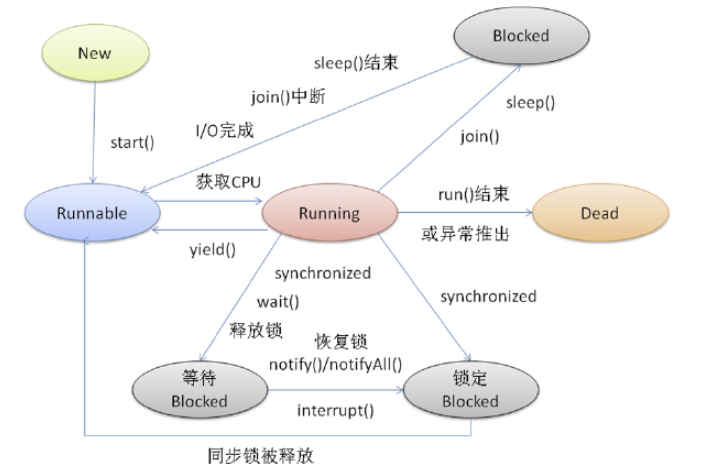
③运行(表示分配到了cpu资源)

④阻塞(等待阻塞：执行wait()

同步阻塞：获取对象的同步锁时，锁被别的线程占用

其他阻塞：执行了sleep()或者join()

⑤死亡(线程销毁)



2、sleep()与wait()区别？

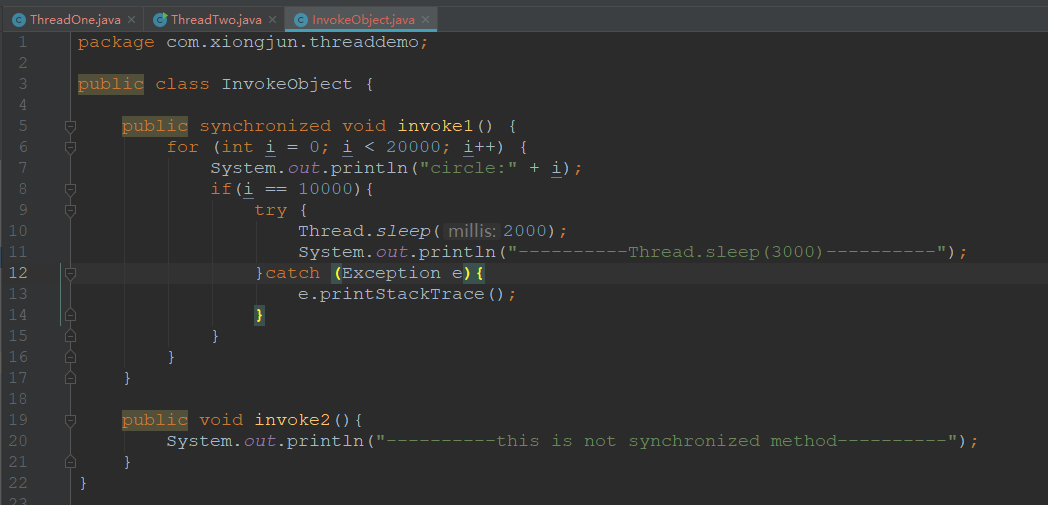
线程在调用sleep()后主动让出cpu，但是还会占有锁，当sleep指定的时间过后，该线程进入可执行状态，要看cpu是否空闲，空闲才会立即继续执行sleep()后面的程序；

wait()方法是指在一个已经进入同步锁的县城内，让自己让出同步锁，需要其他线程调用notify()或者notifyAll()才唤起当前线程，但是其他线程不会释放锁，此时，当前线程进入可执行状态，可以参与锁的竞争，当前得到锁之后才会继续执行wait()后的程序。

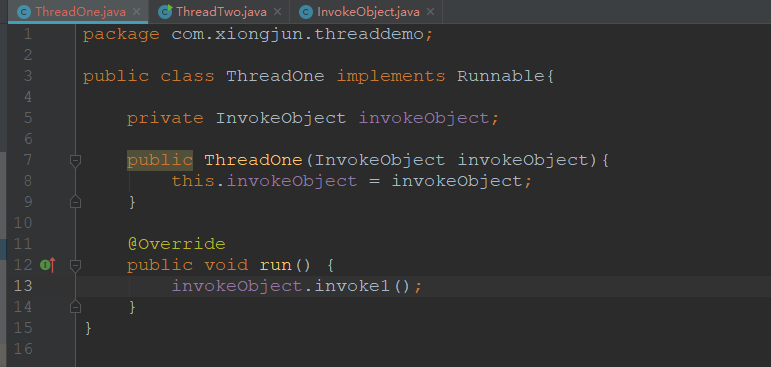
3、当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法？

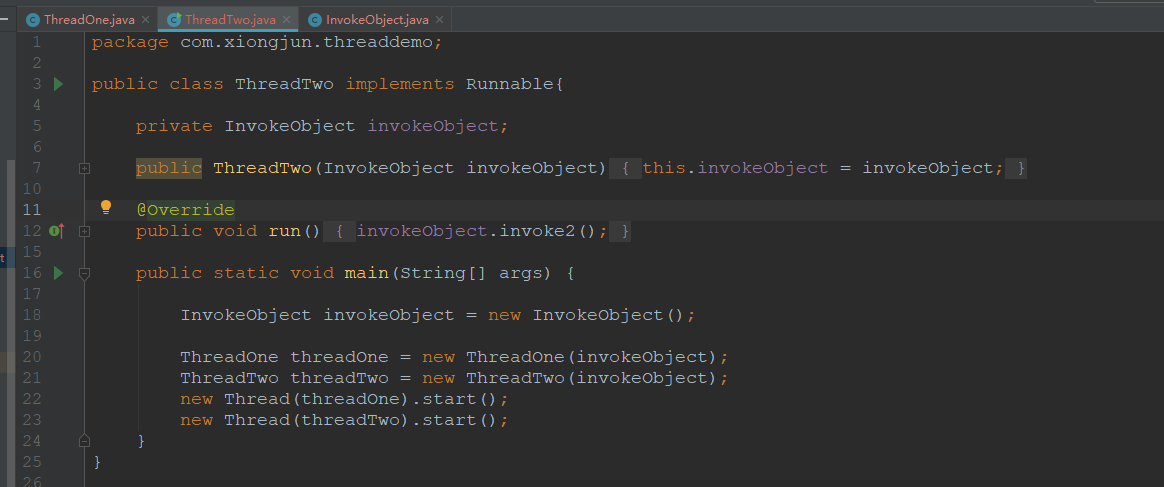
分三种情况：

情况①：其他方法不是synchronized时，其他线程可以进入此对象的其他方法，此时，是异步执行，比如：

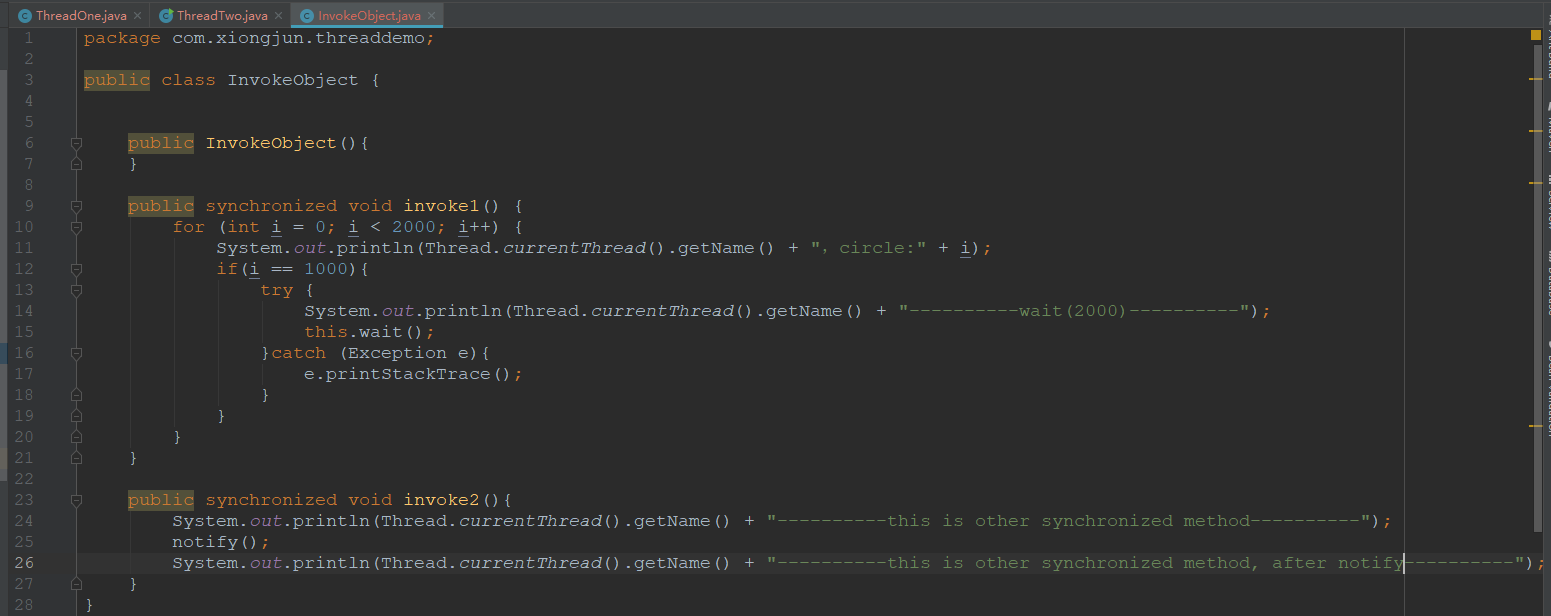


两个线程类：

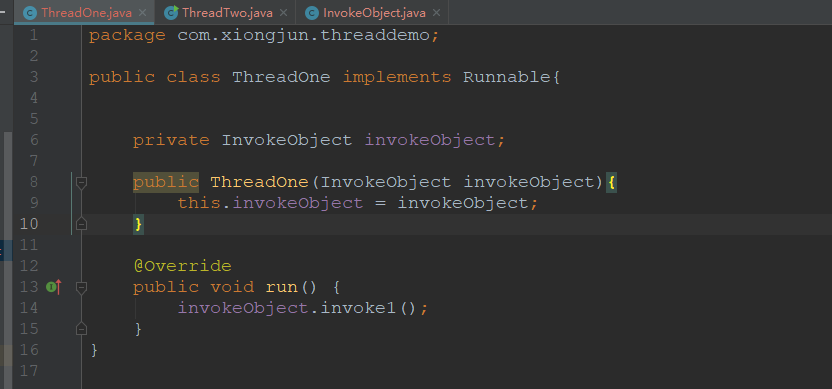


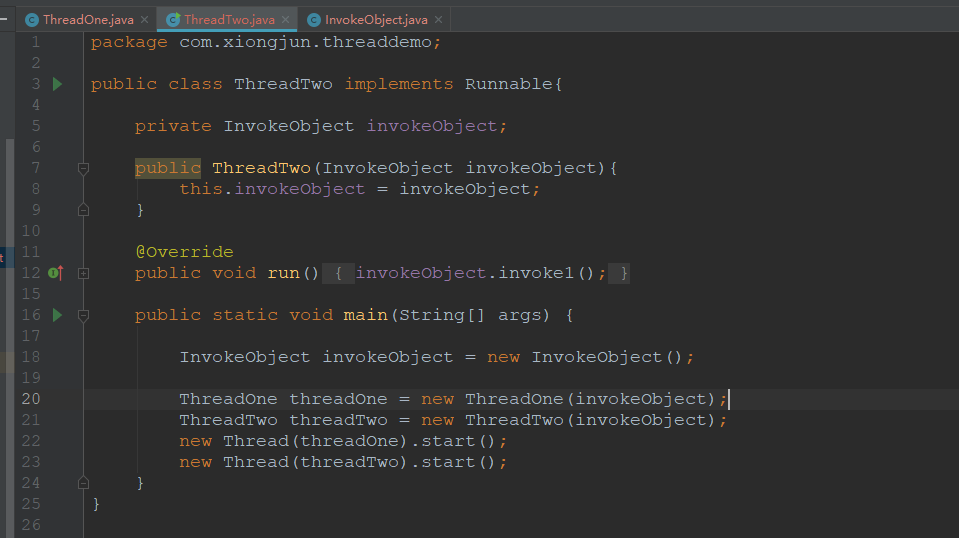


情况②：如果这个方法内部调用了wait，则可以进入其他的synchronized方法，此时是同步执行。如果没有调用wait,则不能进入其他的synchronized方法。如下：

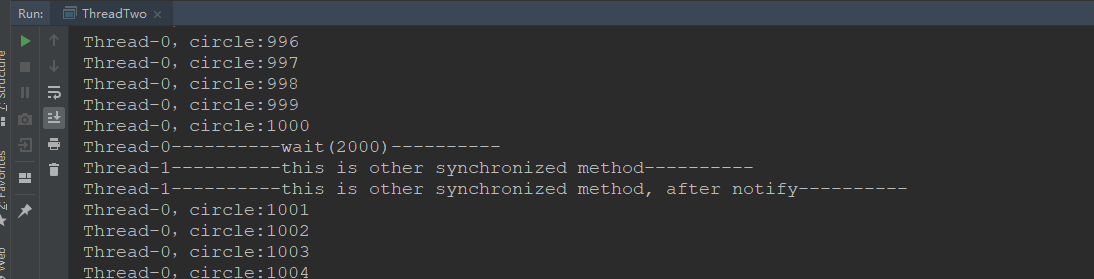


两个线程类，调用不同的synchronized方法：



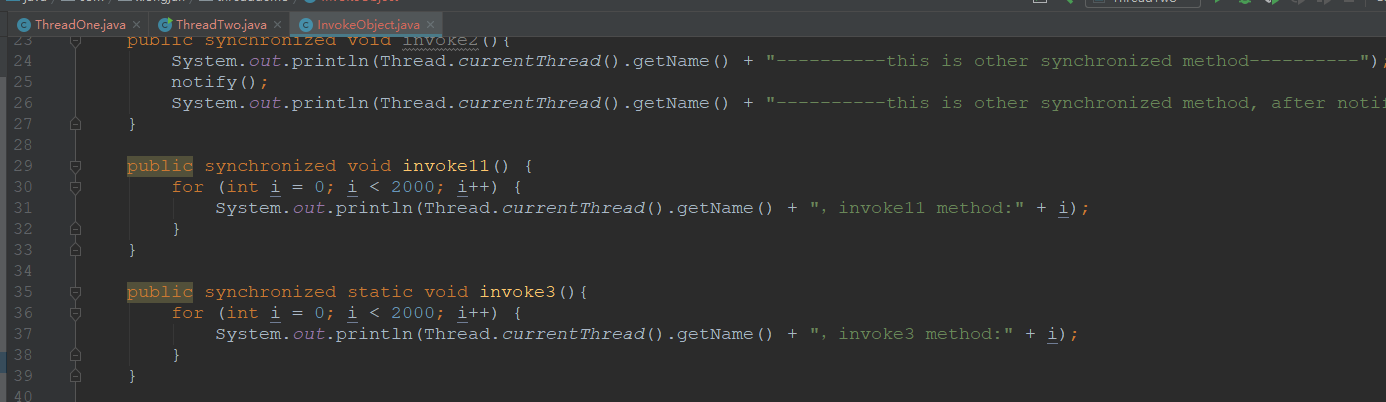


执行结果如下：

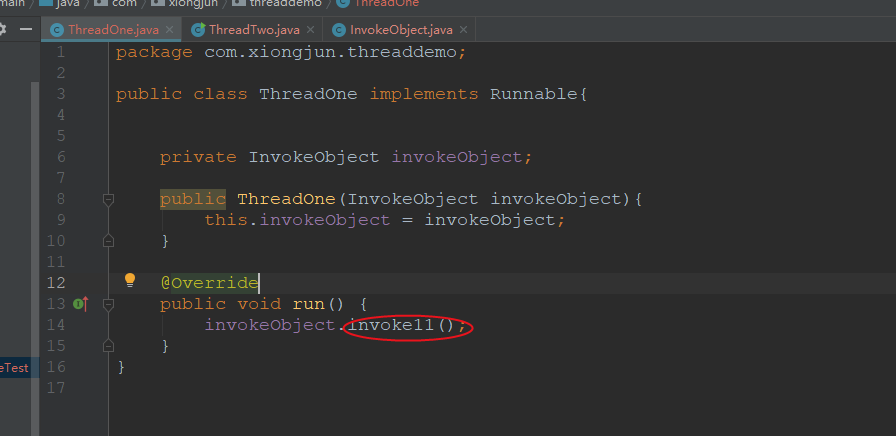


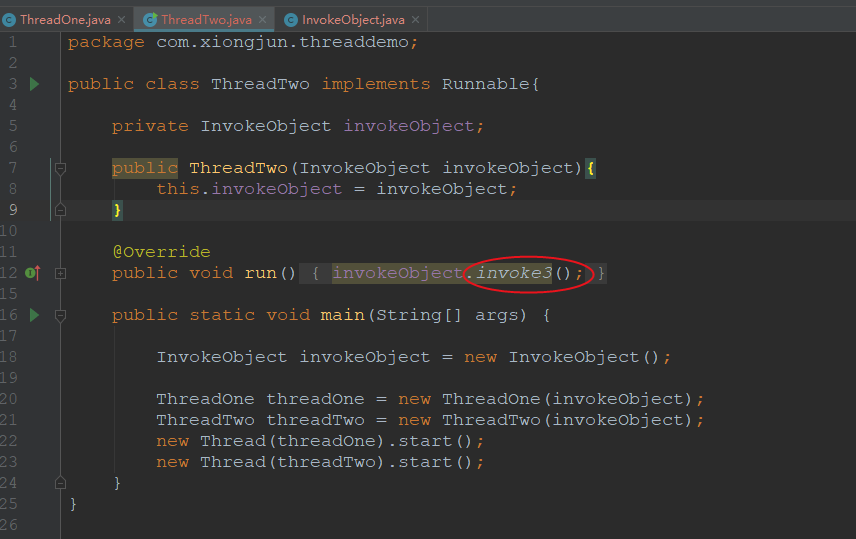
解析：threadOne线程先获取到invokeObject对象锁，执行invoke1()同步方法，当执行wait(),threadOne线程释放锁，threadTwo线程此时获取到invokeObject对象锁，执行invoke2()同步方法，执行notify()之后，threadOne线程此时变成唤醒状态，可以竞争invokeObject对象锁了，等invoke2()同步方法执行完之后，threadTwo线程释放invokeObject对象锁，threadOne线程获取到invokeObject对象锁之后继续执行完invoke1()同步方法。

情况③：如果当前方法没有执行wait,而另外一个方法是static的synchronized方法，此时两个方法能同时进入，他们是异步执行，因为一个方式是使用的示例锁，一个是使用的全局锁。如果不是static的synchronized方法就不能同时执行。另外，当前方法不能是static的synchronized方法，否则也不能同时执行。如下图：

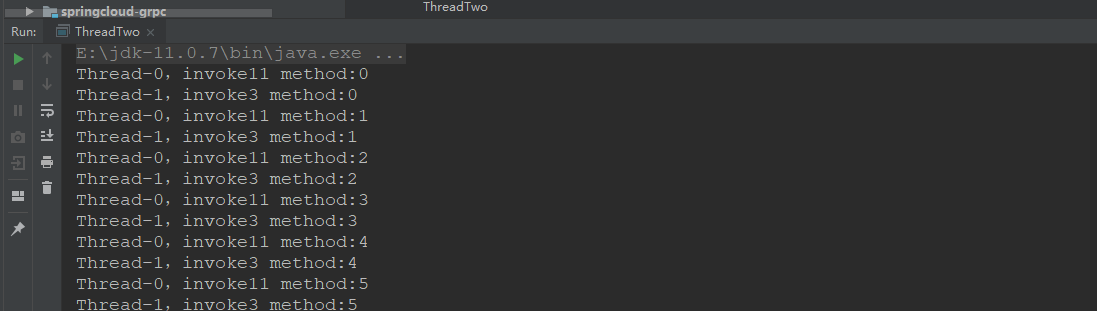


两个线程类：





执行结果：



执行结果表明static的synchronized方法能够获取到锁可以异步执行。

1. 线程的优先级和让步yield()

java中线程优先级priority范围是1~10，默认为5，数值越大，优先级越高。优先级越高，线程分配到CPU资源的概率越大。无论是优先级相同或不同，线程调用都不糊i绝对按照优先级执行。

参考（股票分析软件）：<https://www.cnblogs.com/bclshuai/p/10245165.html>

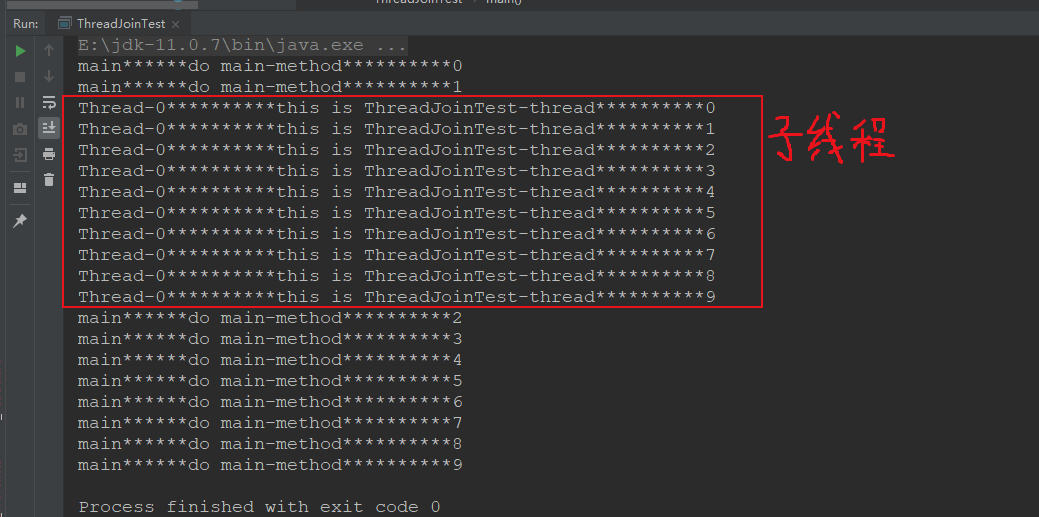
线程让步是调用yield()方法，表示当前线程愿意让出CPU，但是不能保证当前线程一定不继续执行，即不能保证其他线程线执行。

1. 线程的join()方法

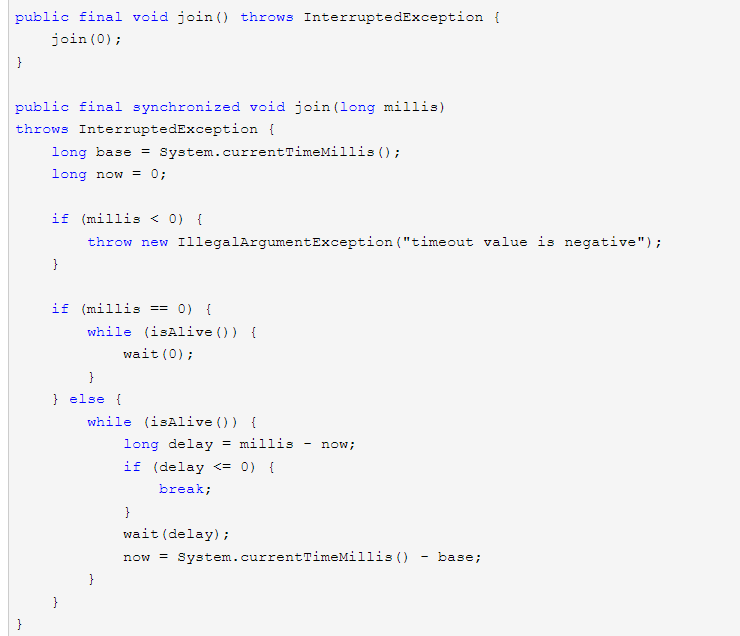
Join()是在“主线程”中插入“子线程”，等“子线程”结束之后“主线程”才能继续运行。示例如下图：



执行结果：



我们来看下join放的源码：



当mills==0时，会进入while(isAlive())循环，即只要子线程是活的，就使用wait(0)让主线程不停等待0s。

为什么isAlive()是判断的子线程而wait(0)就是让主线程等待呢?

因为wait()的作用是让“当前线程”等待，而这里的“当前线程”是指当前在CPU上运行的线程。所以虽然是调用的子线程的wait()方法，但是它是通过主线程去调用的，所以wait的是主线程而不是子线程。

1. 实例锁与全局锁(类锁)

实例锁：就锁住某个实例对象。比如：

public synchronized void doMethod(){

doSomething...;

}

全局锁：锁住整个类。比如：

public static synchronized void doMethod(){

doSomething...;

}

或者

public class A{

public void doMethod(){

synchronized(A.class){

doSomething...;

}

}

}

1. synchronized原理

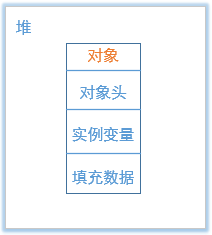
synchronized是由JVM实现互斥同步的一种方式。synchronized的语义底层是通过一个监视器锁(monitor)的对象来完成的。每个对象都有一个monitor。每个synchronized修饰过的代码，当它的monitor被占用时，就会处于锁定状态并且尝试获取monitor的所有权。过程如下：

1. 如果monitro的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者；
2. 如果线程已经占有该monitor，只是重新进入(执行monitorenter指令)，则进入monitor的进入数加1,如果释放锁(或者叫做退出monitor，即执行monitorexit指令)，进入数减1；

3、如果其他下称已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

同步代码块是由monitorenter和monitorexit指令来实现同步的，他们隐式的执行了lock和unLock操作，用于提供原子性保证。monitorenter指令插入到同步代码块开始的位置、monitorexit指令插入到同步代码块结束位置。而同步方法是由方法调用指令读取运行时常量池中方法的ACC\_SYNCHRONIZED标志来隐式实现的。

锁是加在对象上的，无论是类对象还是实例对象，每个对象主要由一个对象头、实例变量、填充数据三部分组成，结构如图：



synchronized使用的锁对象是存储在对象头里的，jvm中采用2个bit来存储对象头(如果是数组对象则会分配3个字节存储对象头，多出来的1bit存放数组的长度),对象头的主要结构是由Mark Word和Class Metadata Address组成，结构如下图：



其中Mark Word在默认情况下存储着对象的hashCode、锁信息(锁状态、锁标志位、是否偏向锁)、分代年龄或GC标志等信息，以下是32位JVM的Mark Word默认存储结构：



考虑到JVM的存储空间效率，Mark Word被设计程一个非固定的数据结构，一边存储更多的数据，如下图：



Synchronized是重量级锁，标志位为10，其中指针指向的是monitor对象的起始地址，每个对象都存在着一个monitor与之关联。

参考文档：<https://www.cnblogs.com/little-sheep/p/9909111.html>

<https://www.cnblogs.com/wuzhenzhao/p/10250801.html>

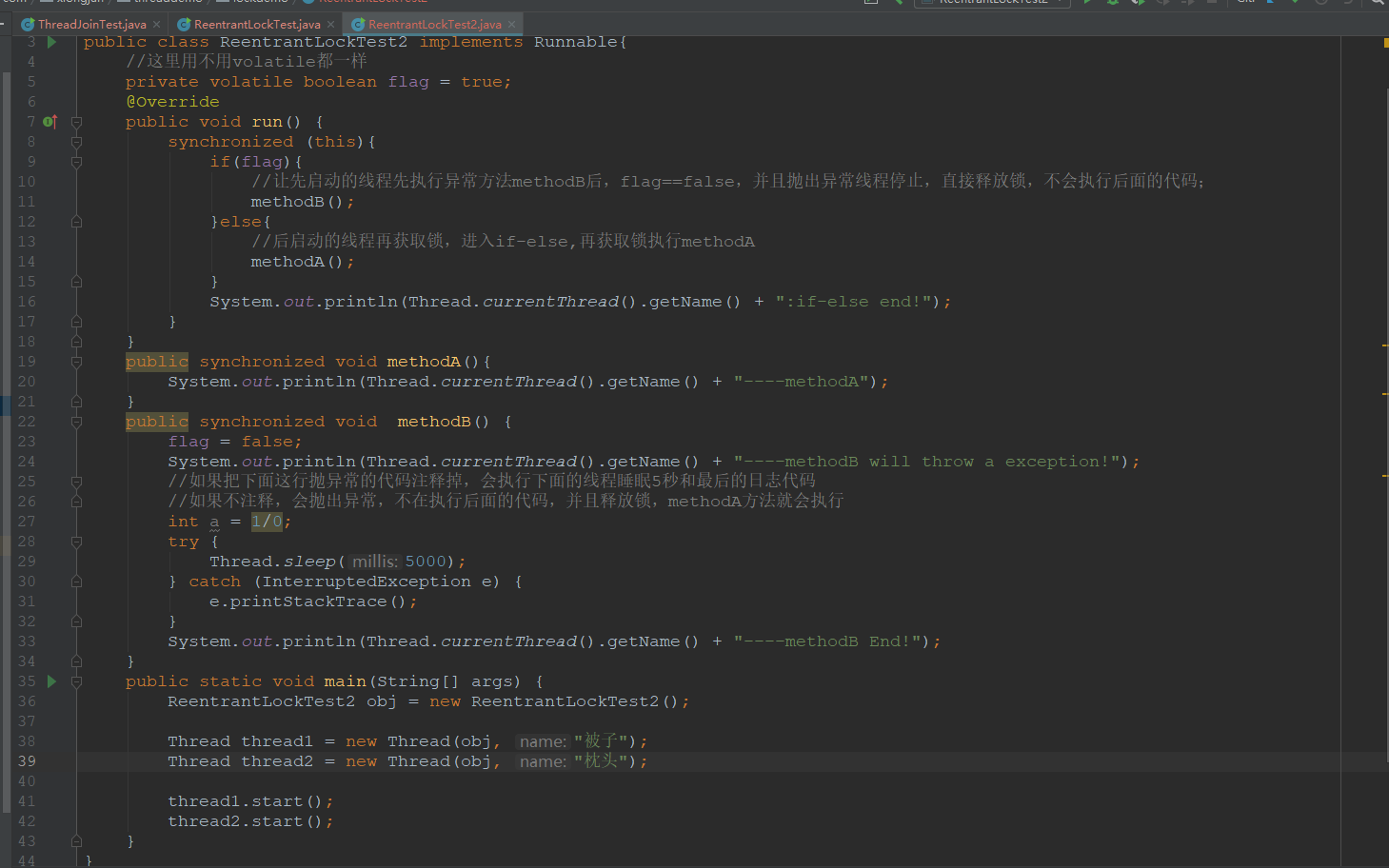
1. 锁的相关概念

重入锁：

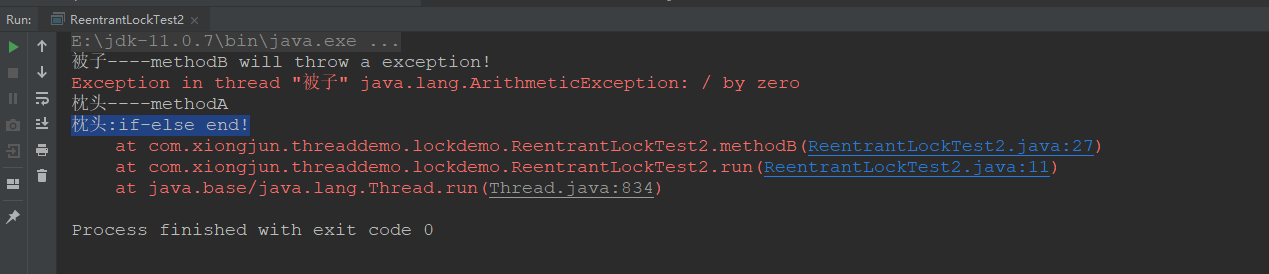
是一种递归无阻塞的同步机制，也叫做递归锁,它可以有效防止死锁。指的是同一线程万层函数获得锁之后，内层递归函数任然由获取该所的代码，但不受影响。Java中synchronized和ReentrantLock是可重入锁。

原理：通过为每个锁关联一个请求计数器和一个占有它的线程。当计数为0时，认为锁是未被占有的；线程请求一个未被占有的锁时，JVM将记录锁的占有者，并且将请求计数器置为1 。如果同一个线程再次请求这个锁，计数器将递增；每次占用线程退出同步块，计数器值将递减。直到计数器为0,锁被释放。

当可重入锁其中一个线程执行synchronized代码抛出异常，会释放锁。如下图：



执行结果如下：



分析：执行main方法，假设“被子”线程先获取到obj的对象锁(这里不一定谁先调用start()方法就是谁先获取到锁)，此时flag=true,调用methodB()方法,会子啊此获取锁(因为synchronized是可重入锁)，此时锁的计数器变成2，然后进入methodB()方法，在methodB()中flag改为false,然后打印“被子----methodB will throw a exception!”，接下来执行int a = 1/0,此时会抛出异常（没有捕获异常），同时锁的计数器直接置为0，释放锁。此时“枕头”线程获取到锁，锁的计数器为1，由于此时flag=false,所以，“枕头”线程走else逻辑，调用methodA()方法，打印”枕头----methodA”,因为此时不抛异常，所以，执行下一条打印语句”枕头:if-else end!”，最后释放锁。

**在执行synchronized代码过程中，如果抛出异常但是未捕获异常，线程会立即停止，退出同步代码块，并且将锁的计数器直接置未0，释放锁，不会执行后续代码。**

自旋锁：

自旋是指循环，一般情况下，线程获取不到锁，就会进入阻塞状态，而自旋锁是在获取不到锁即将阻塞时，进入while循环体内，判断当前线程是否获取到锁，如果没有获取到就一直循环，这样做的好处是，避免了线程的阻塞,减少了CPU资源的消耗提升了性能，因为阻塞和唤醒线程需要占用大量的CPU资源。

但是自旋锁也有缺点，如果持有锁的线程占用锁的时间过长或者锁的竞争过于激烈时，线程在自旋过程中会长时间获取不到所资源，将引起CPU的浪费。所以自旋锁适用于并发度不是特别高的场景，已经释放锁时间比较短的场景。

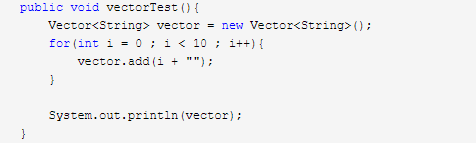
JDK1.6中，java虚拟机提供-XX:+UseSpinning参数来开启自旋锁，使用-XX:PreBlockSpin参数来设置自旋锁等待的次数。

从JDK1.7开始，自旋锁的参数被取消，虚拟机不再支持由用户配置自旋锁，自旋锁总是会执行，自旋锁次数也由虚拟机自动调整。

随着jdk的更新，在1.6的时候，又出现了“自适应自旋锁”。所谓自适应，就是对于同一个锁对象，线程的自旋时间是根据上一个持有该锁的自旋时间以及状态来确定的。比如，如果一个线程刚刚通过自旋得到了锁，并且该线程也在运行中，那么JVM会认为此次自旋操作也是有很大机会可以拿到锁，因此他会让自旋的时间相对演唱，但是，如果线程自旋操作很少成功，JVM可能直接忽略自旋操作。因此，自适应自旋锁是一个更加智能的锁。

锁消除：

锁消除的依据是jvm检测到不可能存在锁竞争，比如下面代码：



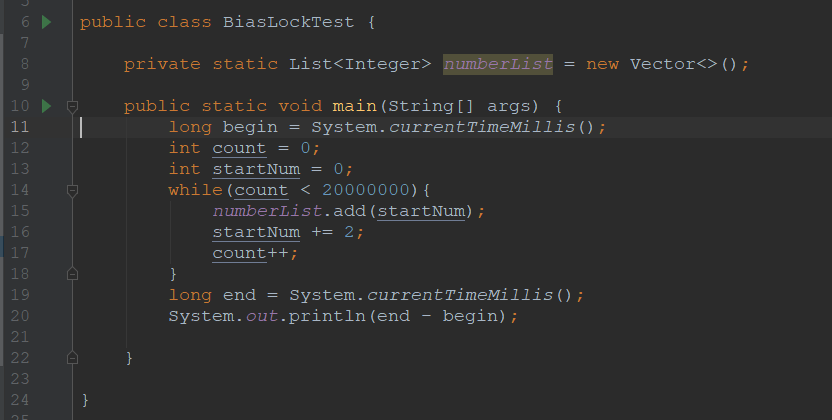
Vector是线程安全的，它的add()方法是synchronized的，但是对于上述代码中，vector是不可能被多个线程操作的，所以，vector对象不可能存在锁竞争。所以JVM会将vector对象内部的加锁操作消除。

锁粗化：

如果连续执行加锁、释放锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。锁粗化就是将多个连续的加锁、解锁操作链接在一起，扩展成一个范围更大的锁，增大锁的作用域。如上面(锁消除)的代码示例，vector对象每次add都需要加锁、释放锁操作，会合并程一个更大返回的加锁、释放锁操作，即执行完for循环才会执行释放锁。

偏向锁：

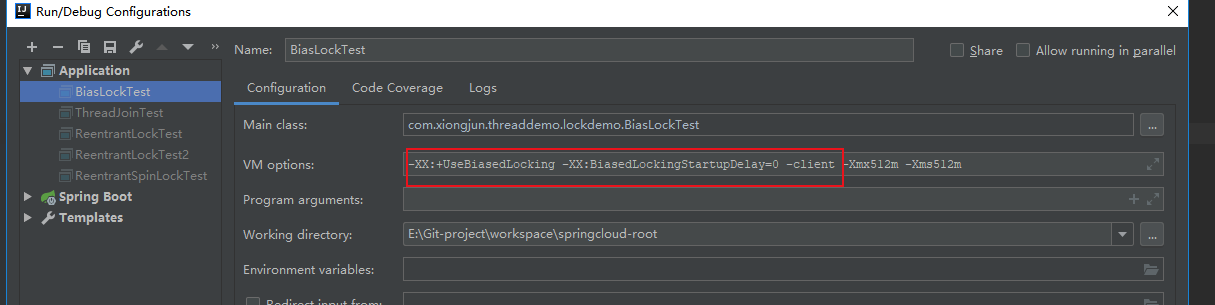
他的核心思想是，如果程序没有竞争，则取消之前已经取得锁的线程同步操作。也就是说，若对象锁被线程获取后，便进入偏向模式，当线程再次请求这个锁时，就无需再进行相关的同步操作了，从而节约了操作时间，如果在此过程中有其他线程进行了锁请求，则退出偏向模式。使用场景比如，单线程访问同步方法：



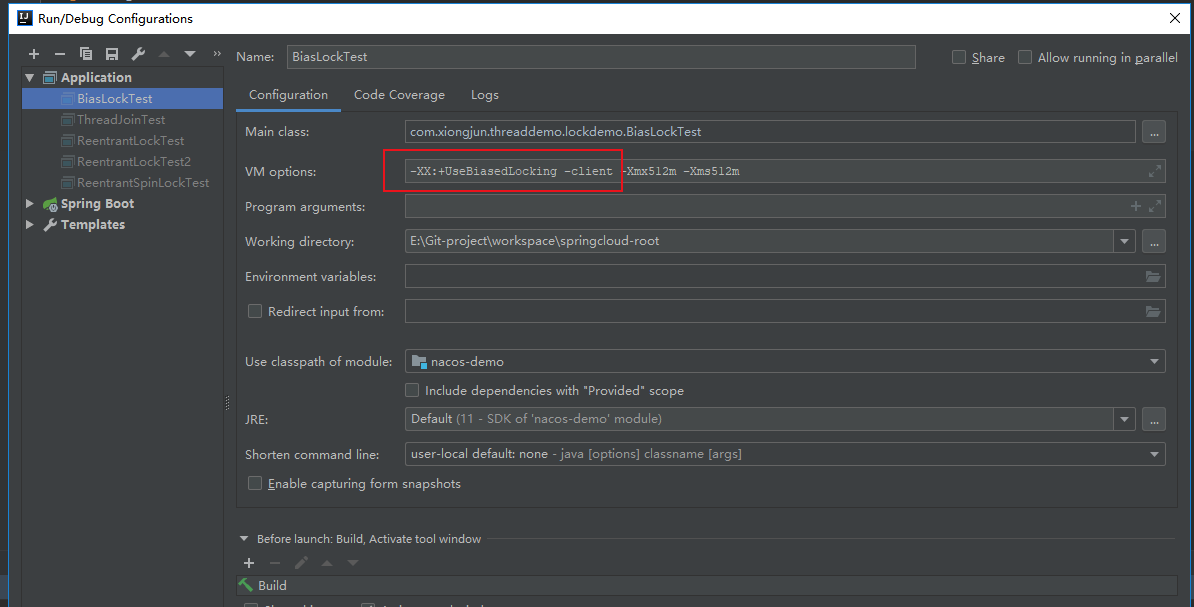
同时需要配置jvm参数开启偏向锁，

-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0 -client -Xmx512m -Xms512m

如下图：



关闭偏向锁，-XX:+UseBiasedLocking -client -Xmx512m -Xms512m如下图：



解析：当锁对象第一次被线程获取的时候，虚拟机会将该对象头部的mardword中的标志位设置为01,即偏向模式。将该线程的id 存在markword中，成功之后，持有该锁的线程以后就不在进行任何的同步操作。但是，在另外一个线程尝试获取该锁的时候，这种偏向模式宣告结束，升级为轻量级锁。

轻量级锁：

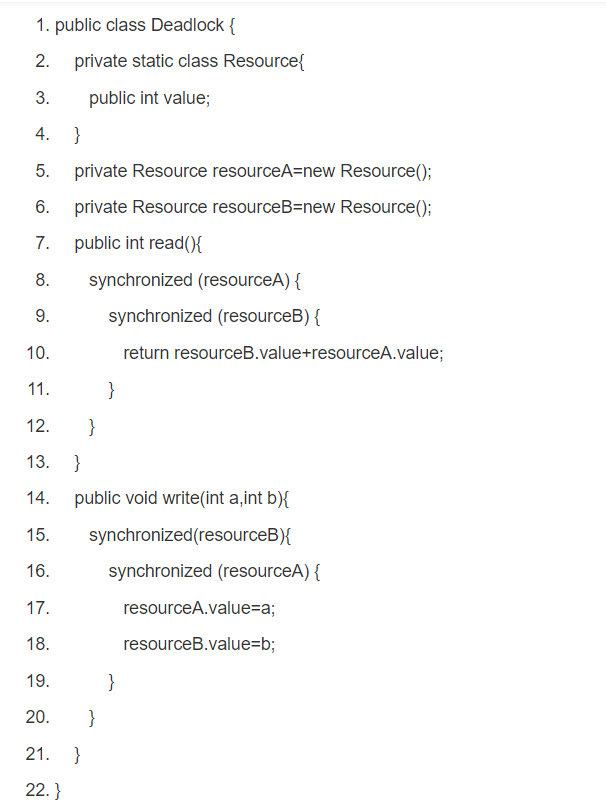
是由偏向锁升级而来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁竞争的时候，偏向锁就会升级成轻量级锁。它通过CAS来避免进入开销较大的互斥操作。

重量级锁：

重量级锁是synchronized，是java虚拟机中最基础的锁实现。在这种状态下，java虚拟机会阻塞获取锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。

线程死锁：

当两个线程被阻塞，每个线程都在等待另一个线程释放锁，就会发生死锁。示例如下：



假设read()方法由线程A启动，write()由线程Bqidong ,A线程将拥有resourceA锁，写线程将拥有resourceB锁，两者都坚持等待的话就会出现死锁。实际上，上面的代码发生死锁的概率很小，因为在代码内的某个点，CPU必须从读线程切换到写线程，所以死锁基本上不会发生，但是无论代码中发生死锁的概率多小，一旦发生死锁，程序就死掉。

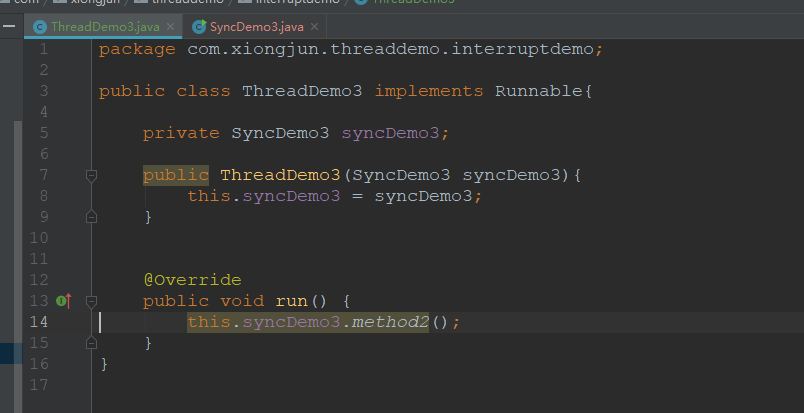
1. 线程的interrupt()方法使用

表示中断线程。如果是中断正在执行的线程，则当前线程的状态会立即变成interrupted，中断的结果线程是死亡、还是等待新的任务或是继续运行至下一步，就取决于这个程序本身。线程会不时地检测这个中断标示位，以判断线程是否应该被中断（中断标示值是否为true）。它并不像stop方法那样会中断一个正在运行的线程。

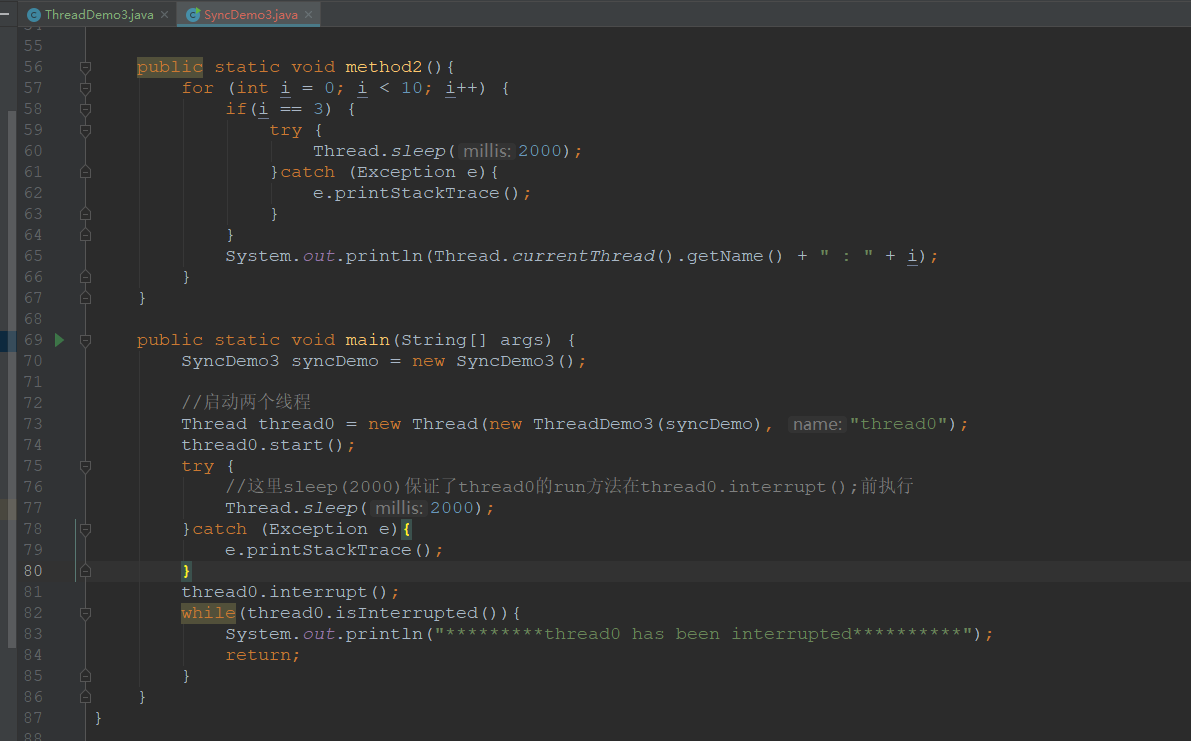
①sleep() &interrupt()一起使用

线程A正在sleep()休眠着，此时当前线程A调用interrupt()方法使线程A中断休眠操作，线程A的状态会马上变成interrupted,且会抛出异常(因为sleep()会抛InterruptException),但是线程A会继续往下执行。示例如下：

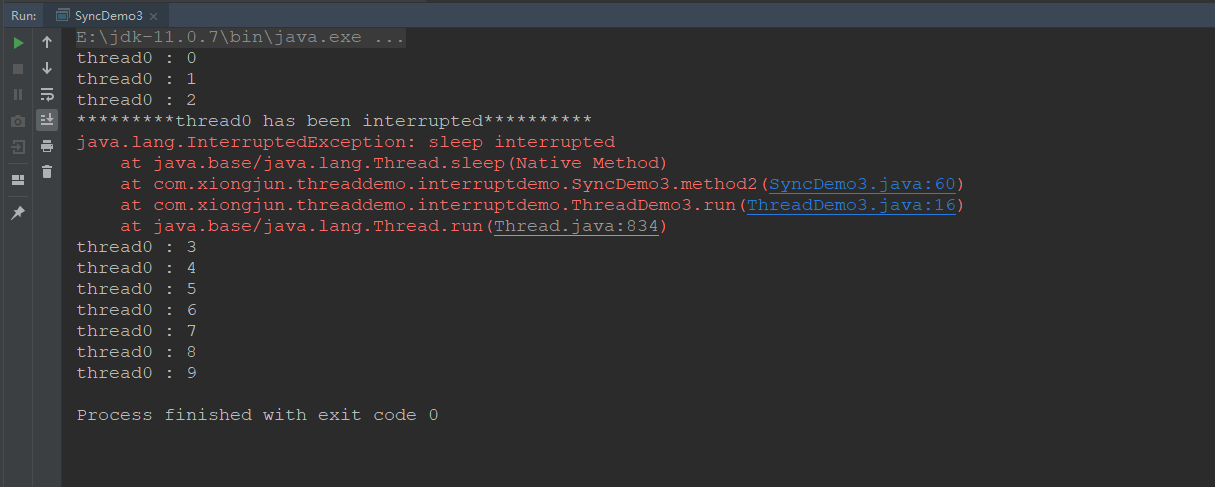
线程类：



操作类：



执行结果如下，在interrupt之后，线程A会中断sleep(),抛出异常，并且执行完剩下的for循环：



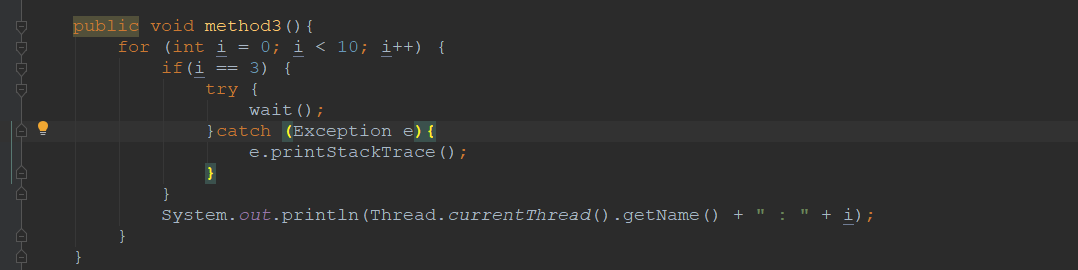
②wait() &interrupt()一起使用

wait()、notify()和notifyAll()必须要synchronized修饰的方法中调用，否则会抛异常IllegalMonitorStateException。这里讲一下这三个方法为什么必须要在synchronized修饰的方法中调用。

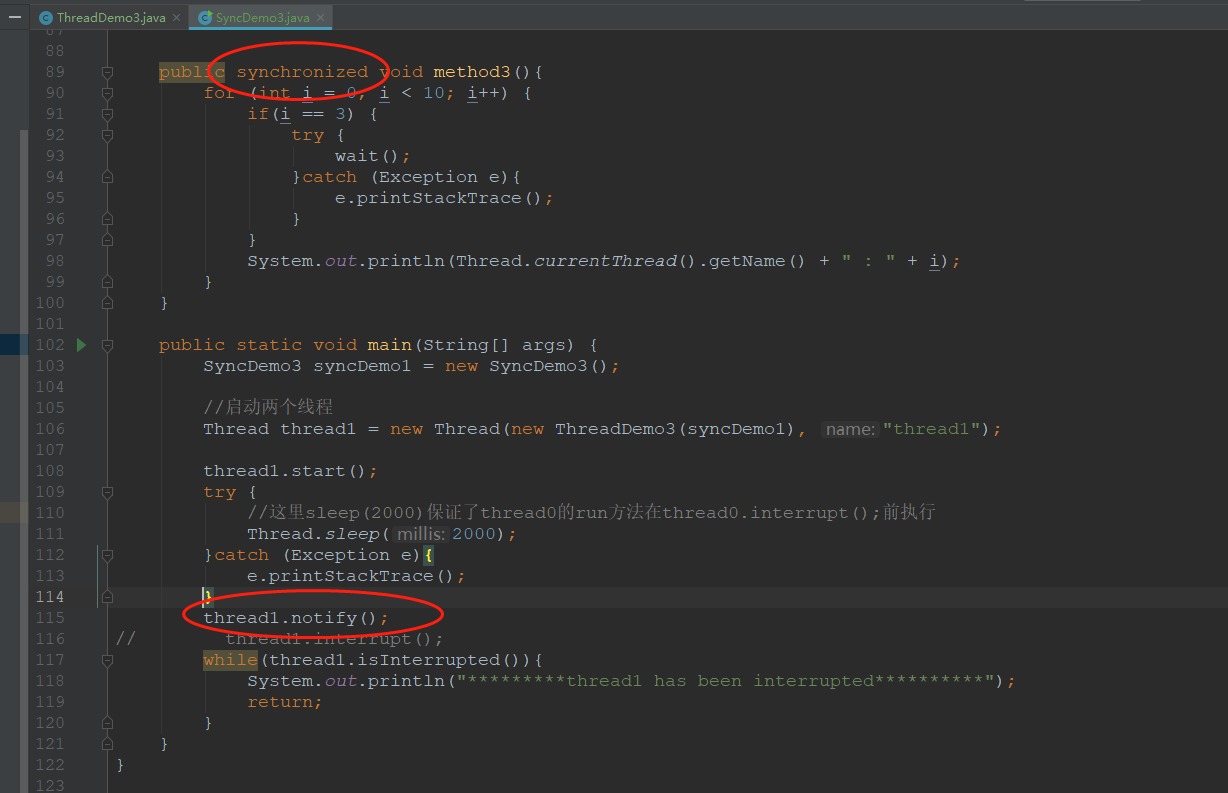
调用wait()方法就是释放锁，释放锁的前提使必须要先获得锁，而只有synchronized修饰的方法才会有对象锁；

nofity()和notifyAll()都是将锁交给含有wait()方法的线程，让线程继续执行下去，如果滋生没有锁，也无法把锁交给其他线程。像下图两种情况都会抛IllegalMonitorStateException。

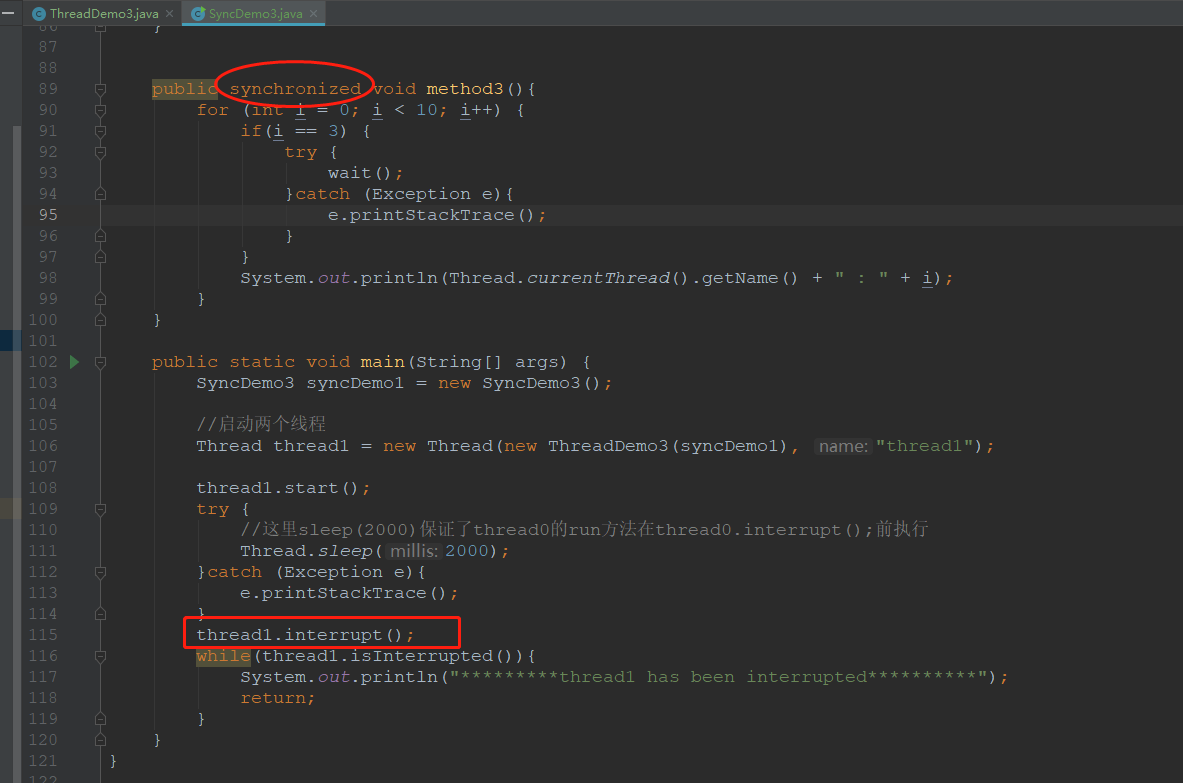
情况1：wait()在没有synchronized修饰的方法中调用



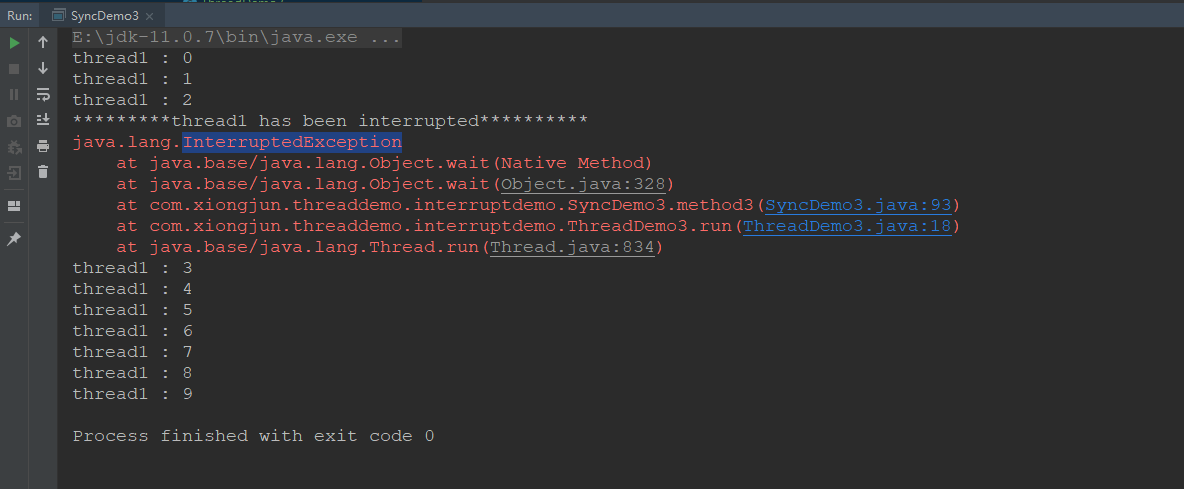
情况2：notify()或notifyAll()在没有synchronized修饰的方法中调用



先在回归正题，在synchronized方法中调用wait之后，在其他线程中中断当前线程，也会抛InterruptedException，类似①sleep() &interrupt()一起使用.如下图：



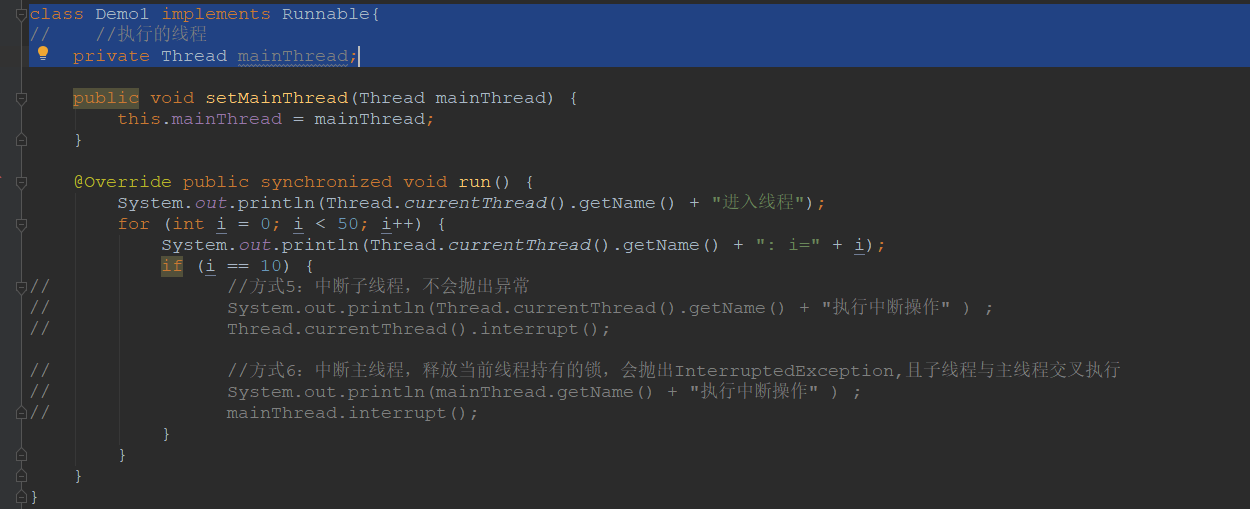
执行结果：



③join() &interrupt()一起使用

如下图的代码：

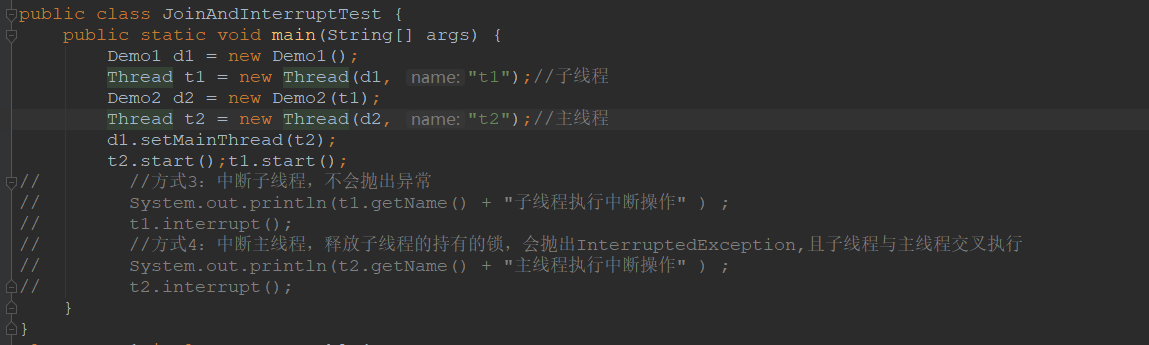
线程1：



线程2：



Main方法：



上图代码中，有六种方式执行interrupt()方法(或者叫有6个地方可以执行interrupt方法)，只有方式4和方式6分别单独执行，会抛异常，这两种方式都是 执行了主线程的interrupt()方法，且执行的地方不在主线程。

1. 停止线程的几种方法

①Thread.stop()

该方法无法预料到线程执行到哪一步就被意外终止，属于非法终止线程。该方法已经被废弃。

②Thread.interrupt()

该方法并不能保证线程不被执行，不建议采用来停止线程。

③volatile

可以通过设置volatile修饰的退出标志位来保证线程安全退出。



1. 守护线程

调用线程对象的方法setDaemon(true),即可将线程设置为守护线程。当正在运行的线程只有守护线程时，JVM退出，即所有守护线程退出。

守护线程使用情况较少，但并非无用，比如，JVM的垃圾回收，内存管理等线程都是守护线程。

该方法必须在启动线程前调用start()。

1. Java中用到的线程调度算法是什么？

分时调度模型：

让所有线程轮流获得cpu的使用权，并且平均分配每个线程占用的CPU的时间片。

抢占式调度模型：

让优先级高的线程先占用CPU,如果优先级相同给，那么就随机选择一个线程，十七占用CPU。处于运行状态的线程会一直运行，直到它不得的放弃CPU。

1. 线程类的构造函数、静态块是被哪个线程调用的？

线程类的构造方法、静态块是被 new这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了Thread1，main 函数中 new 了 Thread2，那么：

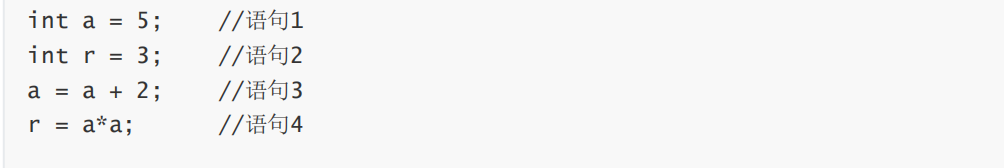
（1）Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是Thread2 自己调用的。

（2）Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是Thread1 自己调用的。

1. 重排序

程序执行的顺序按照代码的先后顺序执行。

一般来说处理器为了提高程序运行效率，可能会对输入代码进行优化，进行重新排序（重排序），它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。比如：



则因为重排序，他还可能执行顺序为（这里标注的是语句的执行顺序） 2-1-3-4，1-3-2-4 但绝不可能 2-1-4-3，因为这打破了依赖关系。

显然重排序对单线程运行是不会有任何问题，但是多线程就不一定了，所以我们在多线程编程时就得考虑这个问题了。

重排序遵守的规则（as-if-serial）：

1. 不管怎么排序，结果不能改变；
2. 不存在数据依赖的可以被编译器和处器其重排序；
3. 一个操作依赖两个操作，这两个操作如果不存在依赖可以重排序
4. 单线程根据此规则不会有问题，但是重排序后多线程会有问题。
5. Synchronized、volatile、CAS比较？

Synchronized是悲观锁，属于抢占式，会引起线程阻塞。

既能保证可见性，又能保证原子性.

volatile提供多线程共享变量可见性和禁止指令重排序优化。

只能保证可见性，无法保证原子性,所以volatile变量的运算再并发下是不安全的。

对于volatile变量，java内存模型定义了8中内存键操作：

lock与unlock

给对象上锁和释放锁。

read和write

把一个变量值从主内存传输到线程的工作内存，以便load。

把store操作从工作内存得到的变量的值，放入主内存的变量中。

load和store

 把read操作从主内存得到的变量值放入工作内存的变量副本中。

 把工作内存的变量值传送到主内存，以便write。

use和assgin

 把工作内存变量值传递给执行引擎。

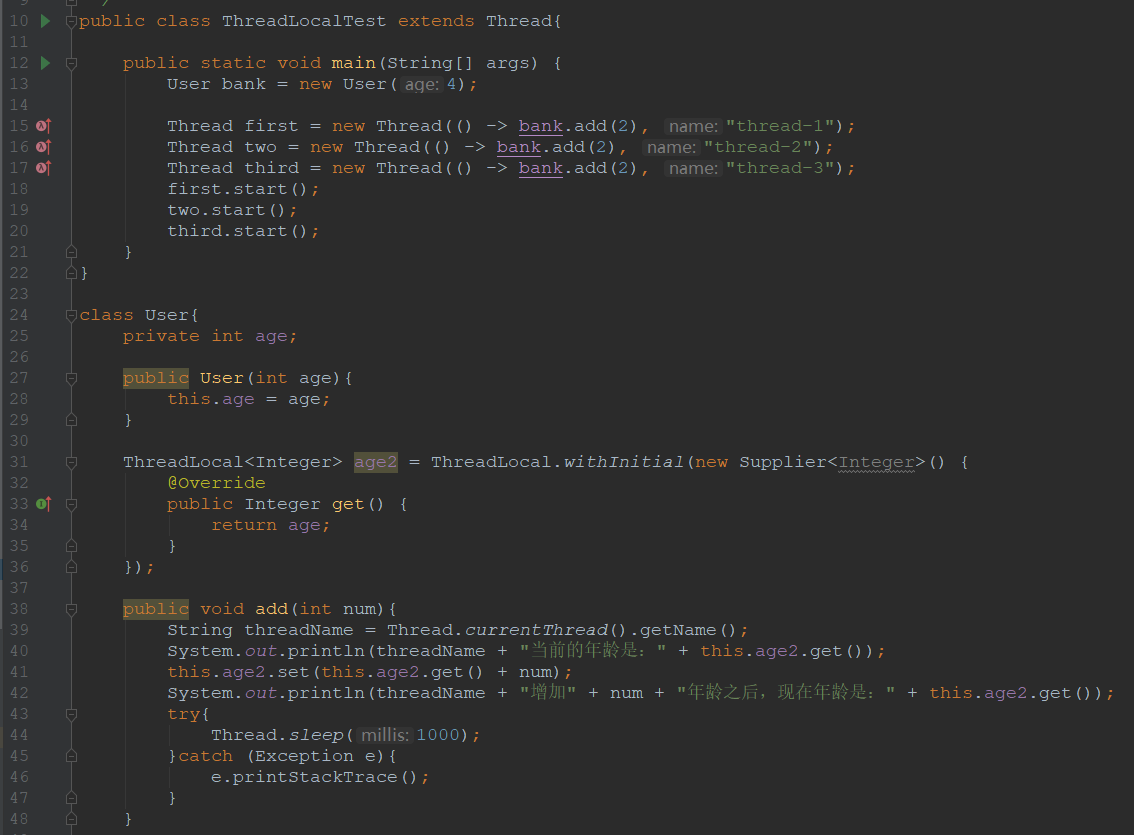
 将执行引擎值传递给工作内存变量值。

以上8中操作保证了volatile变量的修改，一定会被另外的线程看见。

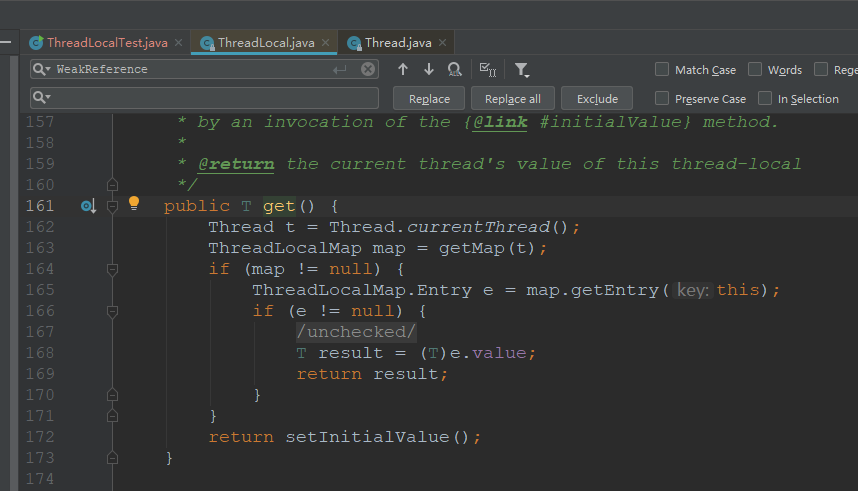
CAS是基于乐观锁的操作，非阻塞。它的操作包含三个操作数--内存位置(V)、预期原值(A)和新值(B)。如果内存地址的值和A的值一样，那么就将内存里面的值更新成B的值。CAS是通过无限循环来获取数据的，如果在第一轮循环中，a线程获取地址里面的值被b线程修改了，那么a线程需要自旋，到下次循环才有可能有机会执行。它会引发ABA问题。

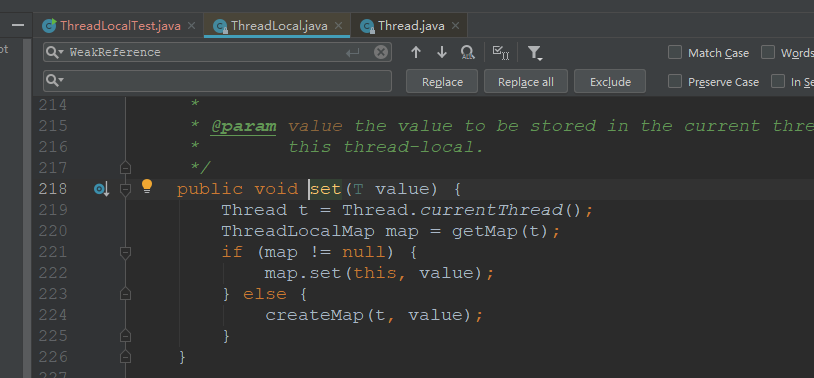
1. ThreadLocal类

ThreadLocal创建的变量只能被当前线程访问，其他线程无法访问和修改。



看一下源码：

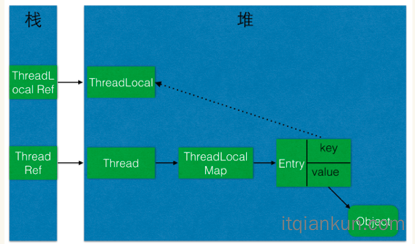




ThreadLocal类中是通过ThreadLocalMap类来存放数据的，而ThreadLocalMap是通过Entry[]实现的。

ThreadLocal可能会造成OOM(OutOfMemeryError)。

首先ThreadLocal里面的ThreadLocalMap类的Entry类继承了所引用WeakReference,所以在ThreadLocalMap类的keu值是使用弱引用的方式来链接ThreadLocal的，就像下面这样：



如上图，ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用引用他，那么系统gc的时候，这个ThreadLocal势必会被回收，这样依赖，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry，如果当前线程再不迟迟结束，这些key为null的Entru的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value,永远无法回收，造成内存泄露。

参考文档：<https://www.itqiankun.com/article/1564891332>

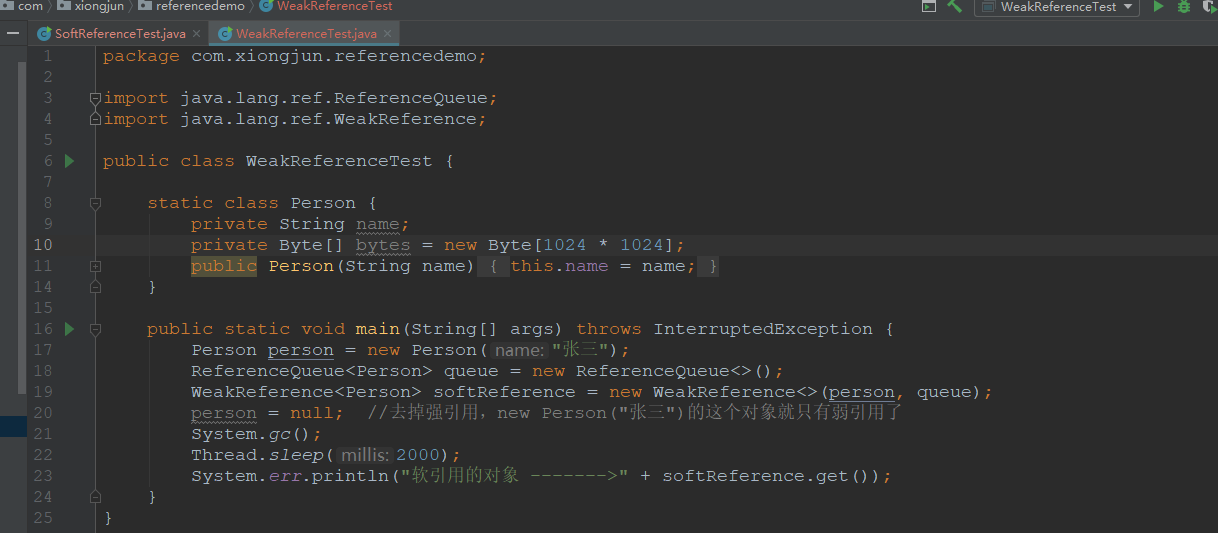
1. 强引用、弱引用、软引用、虚引用

1、强引用(StrongReference)

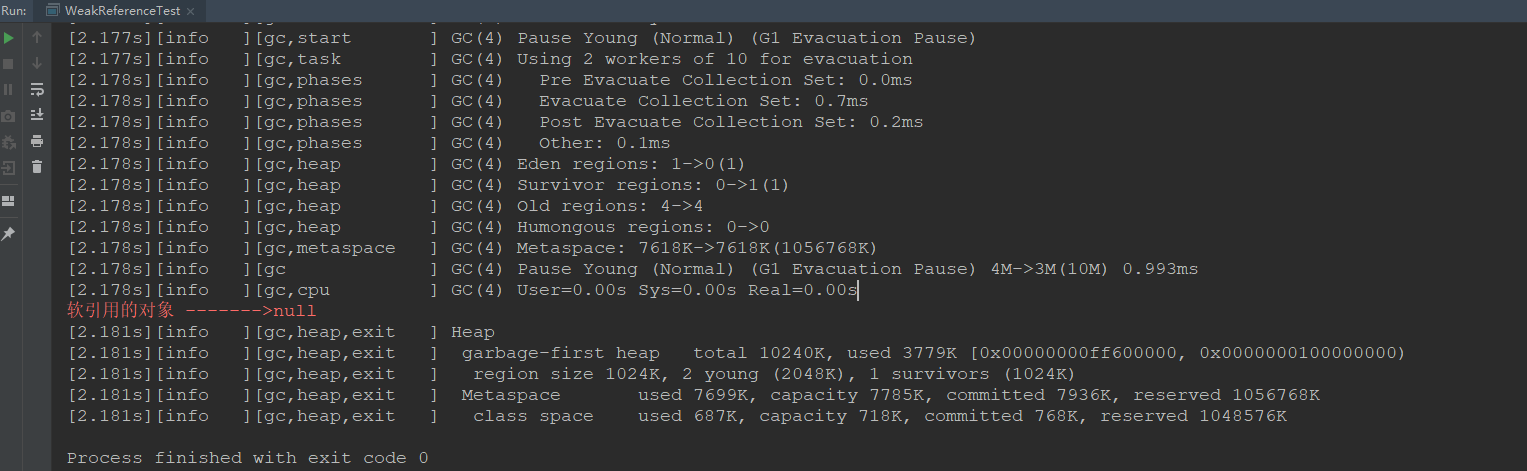
强引用是最普遍的引用，如果一个对象具有强引用，垃圾回收器不会回收该对象，当内存空间不足时，JVM 宁愿抛出 OutOfMemoryError异常；只有当这个对象没有被引用时，才有可能会被回收。

2、弱引用(WeakReference)

只有弱引用的独享拥有更短暂的生命周期，他只能生存到下一次垃圾收集发生之前。当垃圾收集器扫描到只具有弱引用的对象时，无论当前内存空间是否足够，都会回收对引用对象。如下图：



执行结果：



上图代码中，如果没有执行System.gc(),还是能获取到对象，执行GC操作之后，

3、软引用(SoftReference)

软引用具有以下特征：

①当内存空间足够，垃圾回收器就不会回收它。

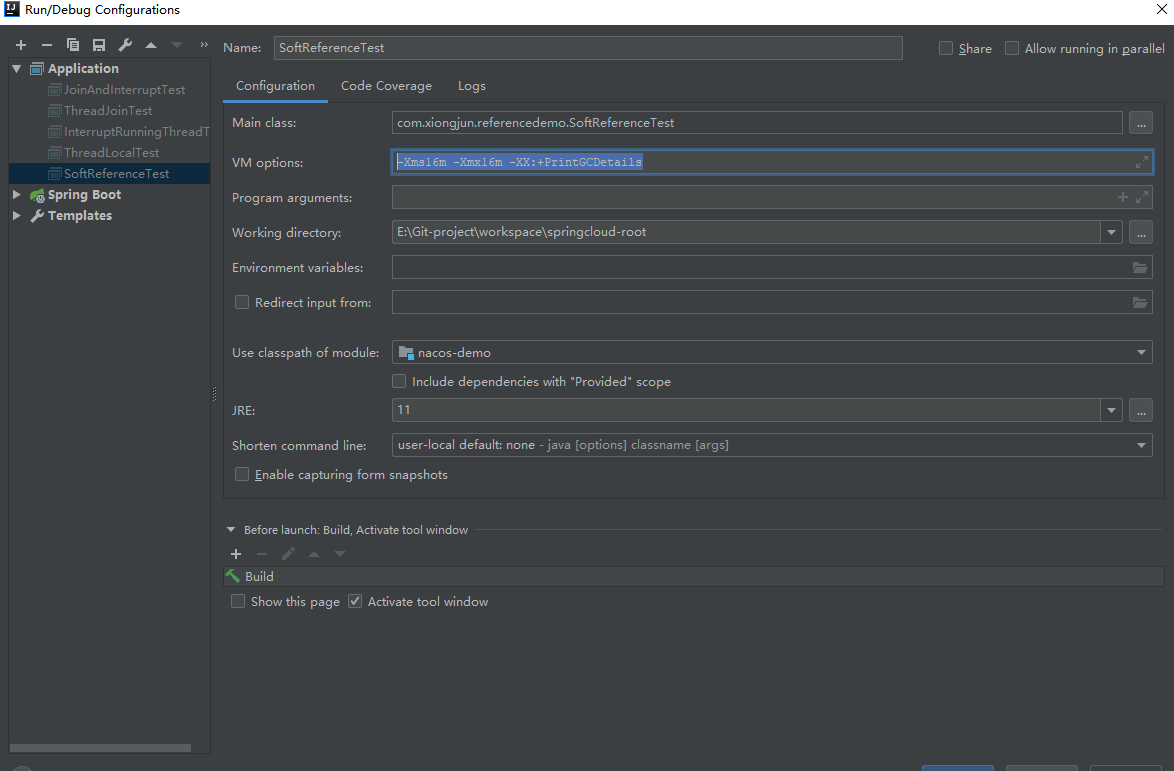
②当内存空间不足了，就会回收该对象。

③JVM会优先回收长时间闲置不用的软引用的对象，对那些刚刚构建的或刚刚使用过的“新”软引用对象会尽可能保留。

④如果回收完还没有足够的内存，才会抛出内存溢出异常。只要垃圾回收器没有回收它，该对象就可以被程序使用。

软引用是用来描述一些有用但并不必须的对象，适合用来实现缓存。我们来测试以下，为了方便测试，将JVM运行内存参数设置为16M,如图：

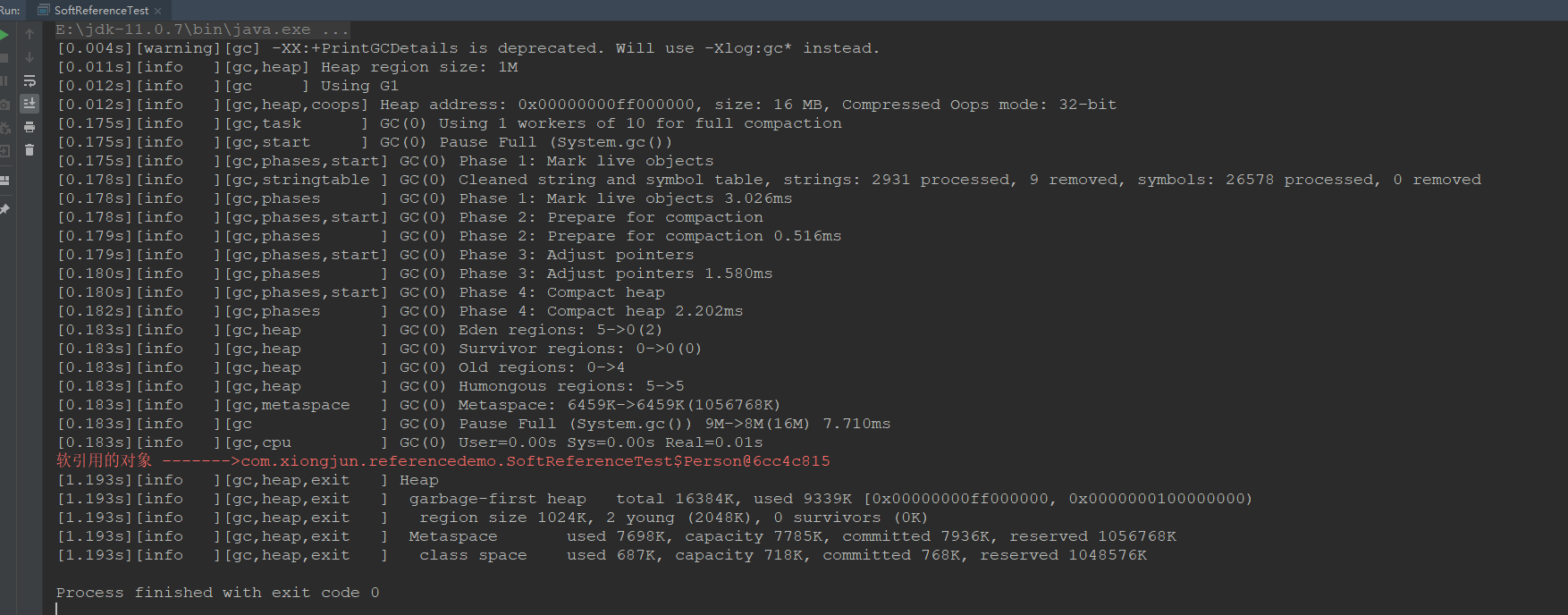
-Xms16m -Xmx16m -XX:+PrintGCDetails



代码：

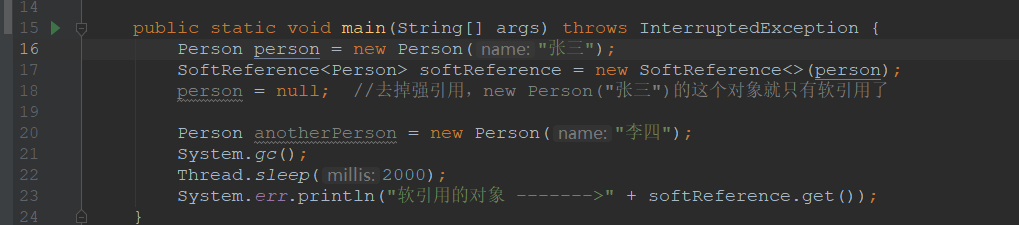


打印：

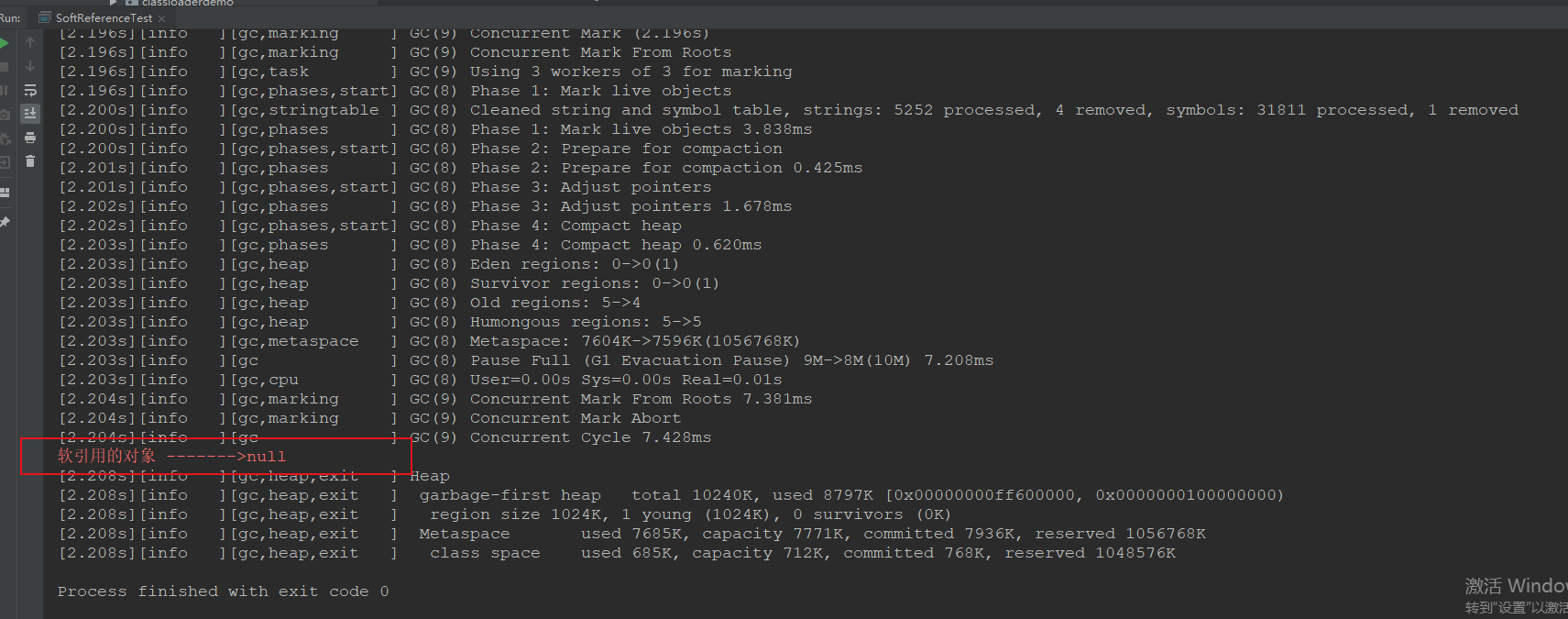


虽然调用了System.gc()后JVM不一定立刻执行GC操作，但从上述执行结果看，已经执行了GC，但是软引用的对象并没有被回收掉，说明现在内存空间还足够，JVM暂时不会回收软引用的对象。

修改main方法如下图：



执行结果：



因为没有足够的空间同时保留两个Person对象，所以再new Person("李四")时，也会触发JVM的GC,同时因为前面的new Person("张三")只有软引用了，它会被回收掉。

4、虚引用(PhantomReference)

虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

使用引用类可以避免在程序执行期间将对象留在内存中。在实际程序设计中，一般很少使用弱引用和虚引用，使用软引用的情况较多，因为软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止OOM问题。

ReferenceQueue<T>，引用队列，垃圾回收之后，垃圾回收器将引用对象添加到该队列中。

1. 原子类AtomicXXX

1、Atomic原理

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以像自旋锁一样，继续尝试，一直等到执行成功为止。

2、原子化基本数据类型

有三个实现类AtomicInteger、AtomicBoolean、AtomicLong，使用以AtomicInteger为例，如下图：



3、原子化对象引用类型

实现类分别有：AtomicReference、AtomicStampedReference、AtomicMarkableReference,其中后两个可以实现解决ABA问题的方案。

①什么是ABA问题？

比如下列代码：



t1,t2都读取到内存值是5,t2让出执行权,t1先执行,执行多次,修改过内存值,但是最终值和初始内存值一样;此时交到线程2执行,线程2执行成功。这就是ABA。

②ABA的解决思路

就是使用版本号，在变量前面追加版本号，每次变量更新的时候把版本号加1，那么A-B-A就会变成1A-2B-3A。如下图:

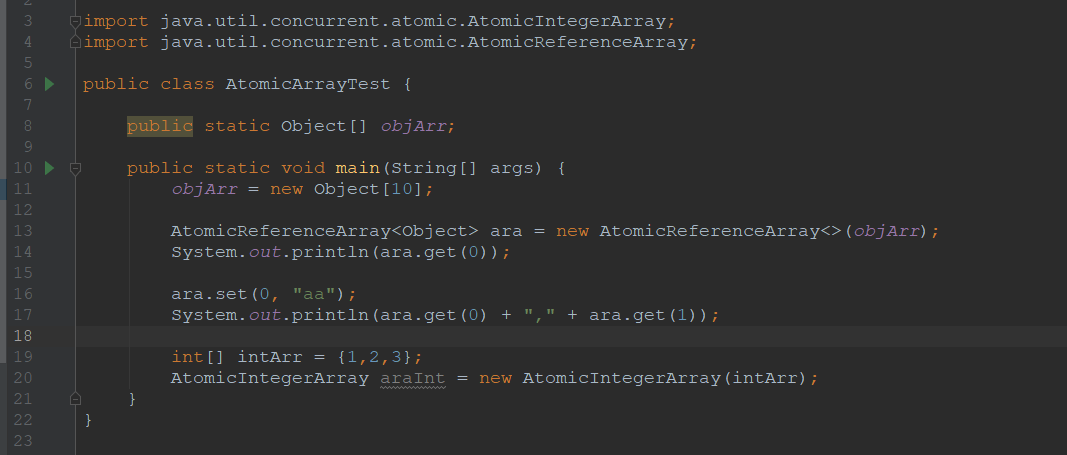


③什么是CAS(Compare And Swap)?

CAS就是乐观锁的实现机制,可用于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同事改写某一数据时由于执行顺序不确定性及中断的不可预知性产生的数据不一致问题。该操作通过将内存中的值与指定数据进行比较，当数值一样时将内存中的数据替换为新的值。

4、原子化数组类型

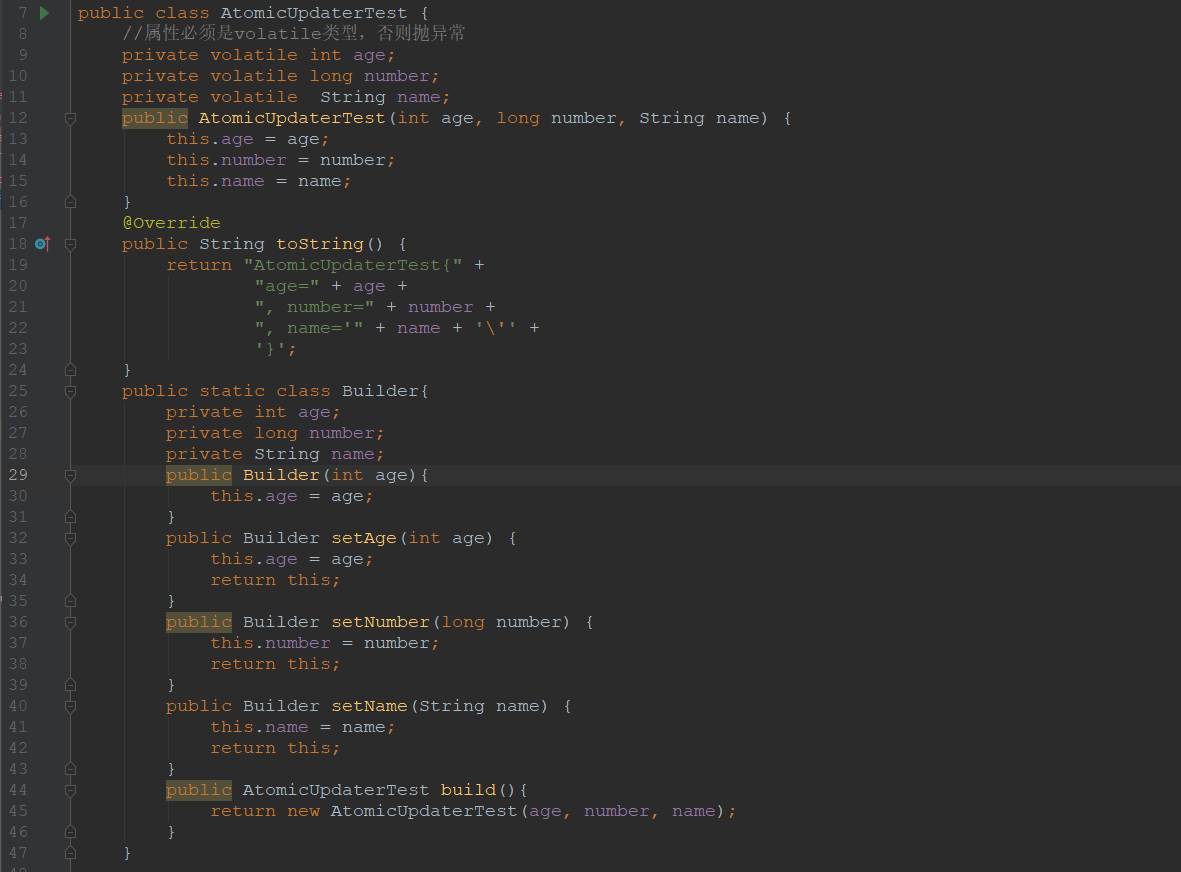
实现类分别有AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray，使用和原子化基本类型差不多。



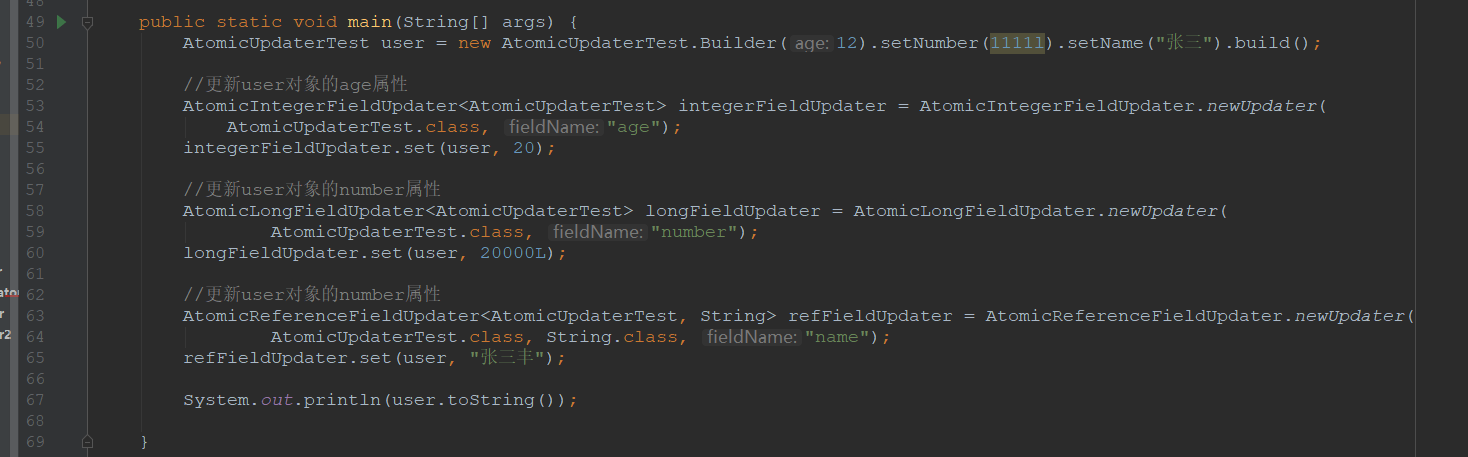
5、原子化对象属性更新器

实现类AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

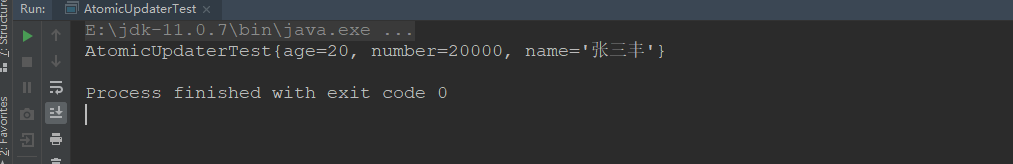
这三个类都是例用java的反射机制实现的，并且为了保证原子性，要求被更新的对象的属性必须是volatile类型的。使用如下：



Main方法：



执行结果：



6、原子化累加器

实现类有四个：DoubleAder、DoubleAccumulator、LongAdder、LongAccumulator

这几个功能仅用来执行累加操作，速度非常快，以DoubleAdder和DoubleAccumulator使用为例，示例如下：



1. Lock相关类

1、关于Condition类

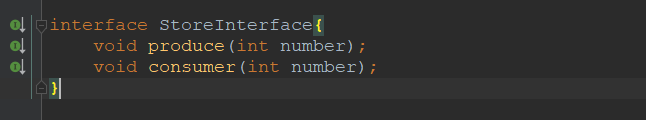
Condition是在java 1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。因此通常来说比较推荐使用Condition，阻塞队列实际上是使用了Condition来模拟线程间协作。

Codition依赖于Lock接口，使用lock.newCondition生成一个Condition对象，调用Condition的await()、signal()、signalAll()必须要在lock()与unlock()之间使用。

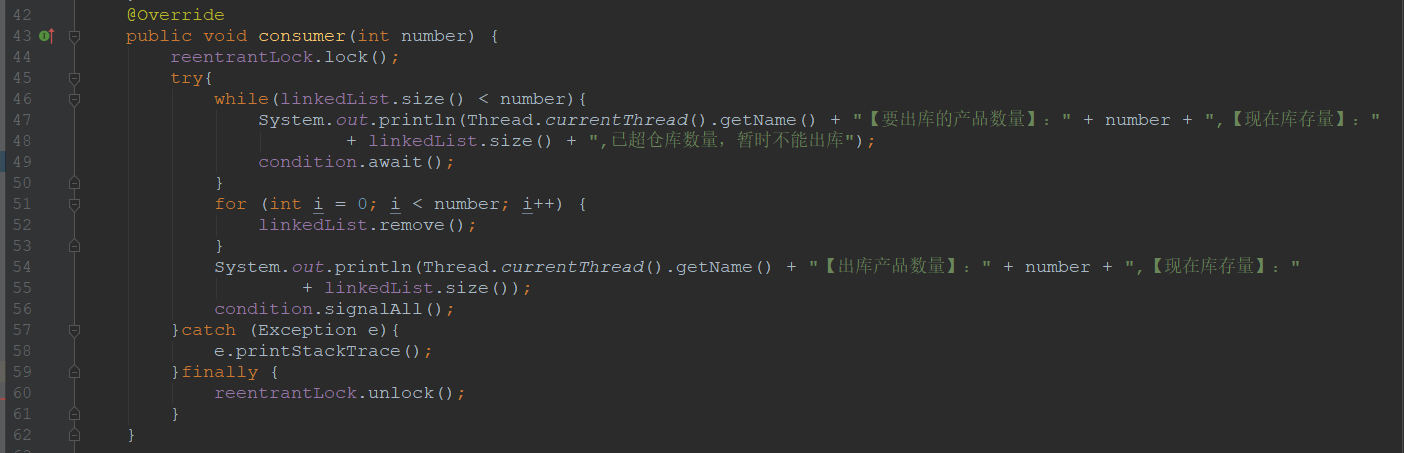
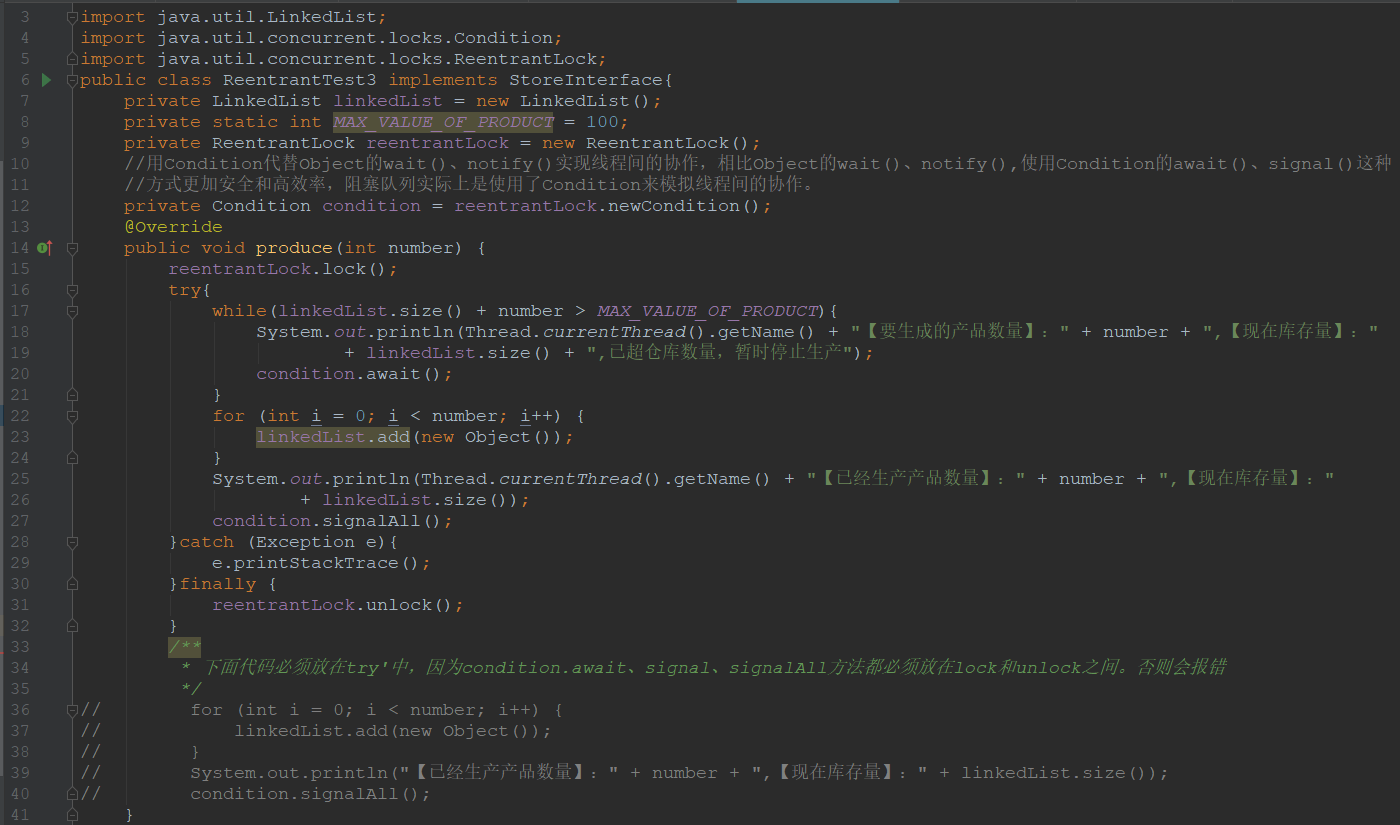
2、ReentrantLock与Condition相关使用

下面来看示例代码：

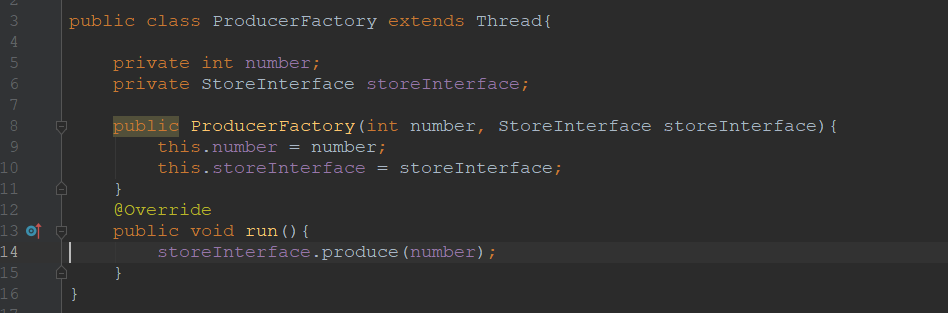
仓库接口，用于消费和生产：



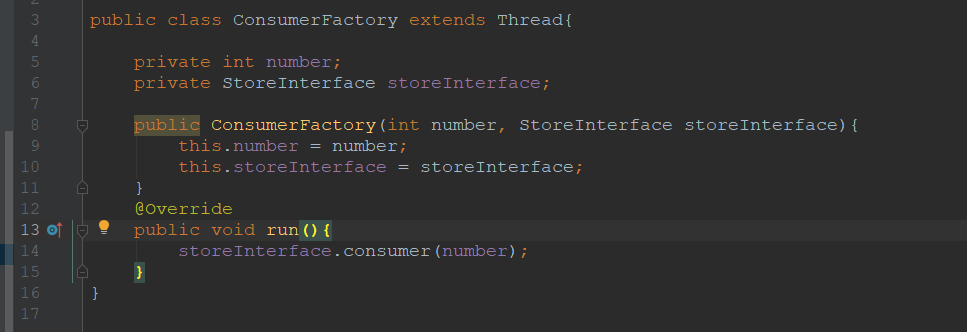
实现仓库接口的消费和生产功能：



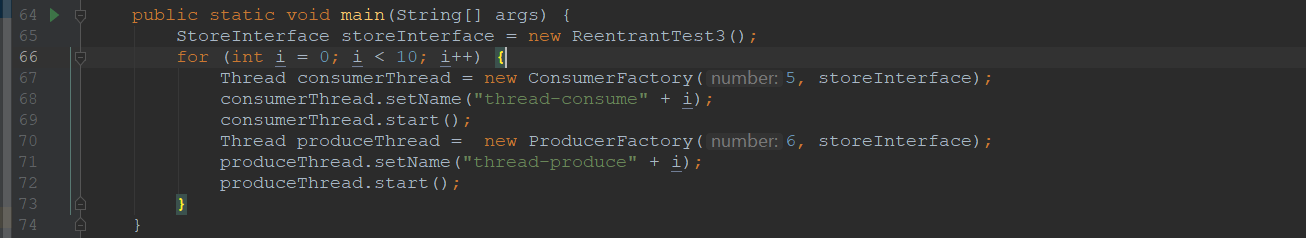
生产者：



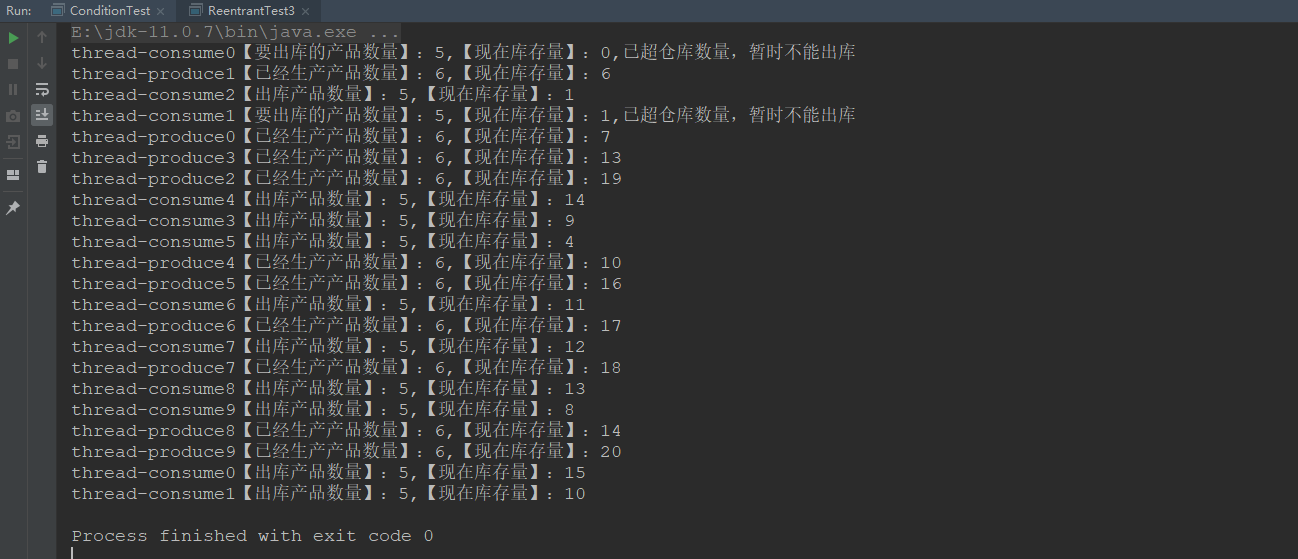
消费者：



Main方法:生产、消费各10次，此处每次生产数量6>消费者数量5，执行完之后线程不会阻塞，如下图：



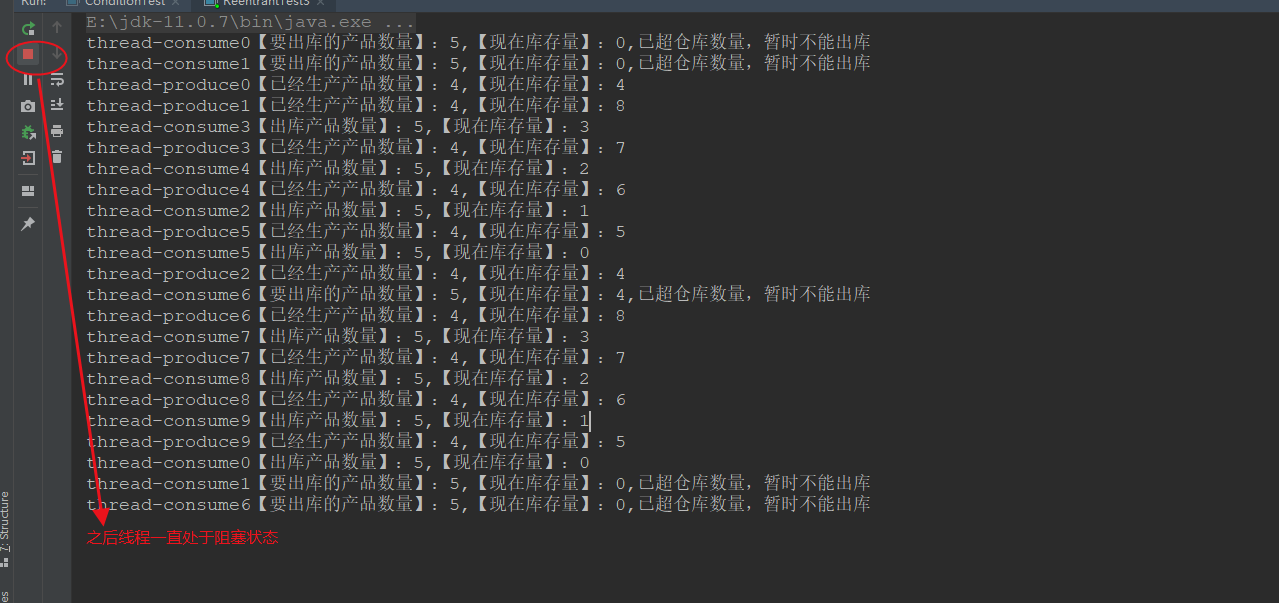
执行结果：



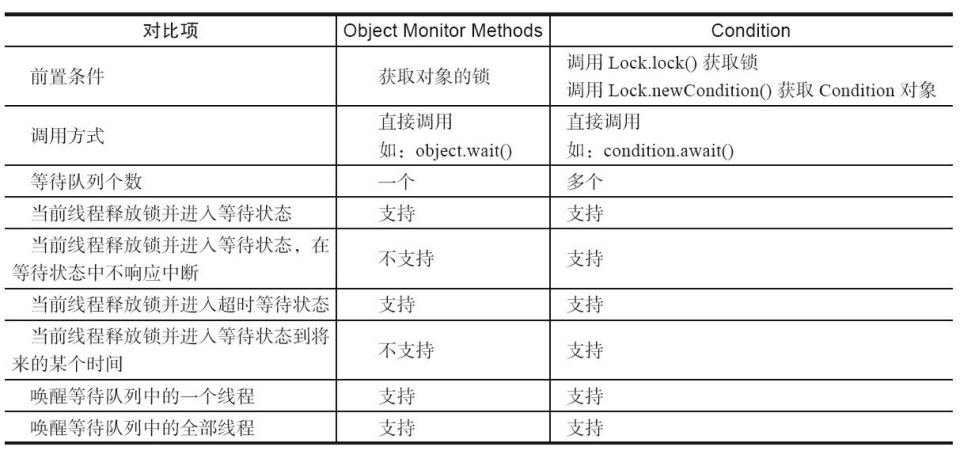
Main方法:生产、消费各10次，此处每次生产数量4>消费者数量5，执行完之后线程会一直处于阻塞，如下图：



执行结果：



1. Lock与synchronized对比



另外，synchronized只支持非公平锁，非公平主要表现在获取锁的行为上，并非是按照申请锁的时间前后给等待线程分配锁的，每当锁被释放后，任何一个线程都有机会竞争到锁，这样做的目的是为了提高执行性能，缺点是可能会产生线程饥饿现象。而Lock还可以支持公平锁。

1. ReentrantLock与synchronized实现原理有什么区别

锁的实现原理基本是为了达到一个目的：让所有的线程都能看到某种标记。

Synchronized通过在对象头中设置标记实现了这一目的，是一种JVM原生的锁实现方式，而ReentrantLock以及所有的基于Lock接口的实现类，都是通过用一个volitile修饰的int型变量，并保证每个线程都能拥有对该int的可见性和原子修改，其本质是基于所谓的AQS框架。

1. AQS是什么

AQS)AbstractQueuedSynchronizer类)是一个用来构建锁和同步器的框架，各种Lock包中的锁(常用的有ReentrantLock、ReadWriteLock)，以及其他如Semaphore、CountDownLatch，甚至是早期的FutureTask等，都是基于AQS来构建。

1. 并发协作之生产者-消费者-仓储模型

对此模型，应该满足一下几点：

1.生产者近几年在仓储未满时候生产，仓满则停止生产；

2.消费者仅仅在仓储有产品时才消费，仓空则等待；

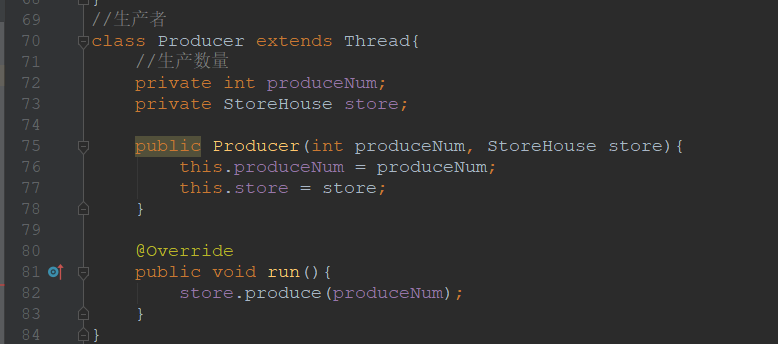
3.当消费者发现仓库没产品时会通知生产者生产；

4.当生产者在生产产品时，应该通知等待的消费者去消费。

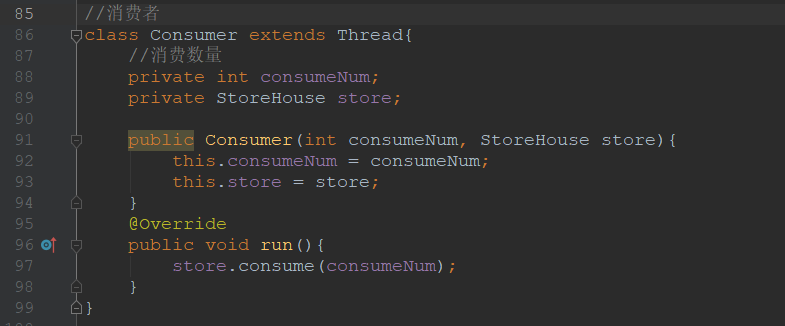
仓库类：



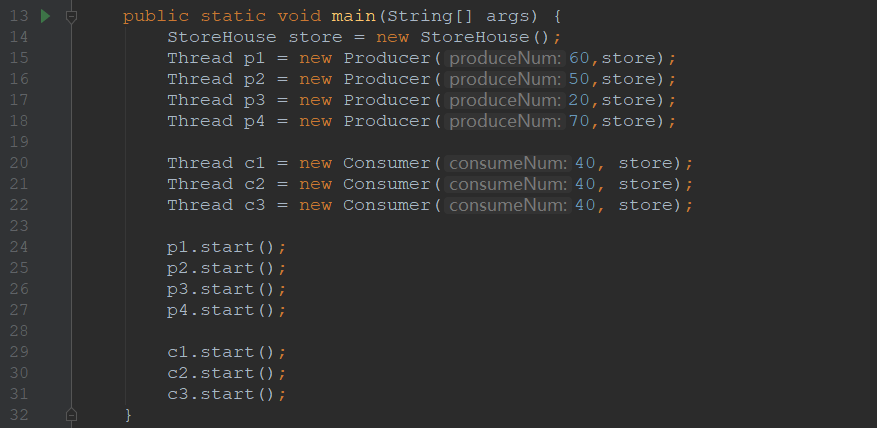
生产者类：



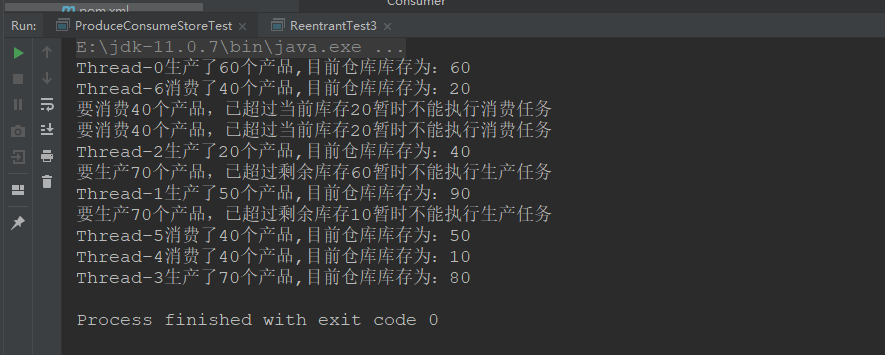
消费者类：



Main方法:



执行结果：



1. 线程池
2. 线程池的优点

①提高效率

线程池时为了突然大量爆发的线程设计的，通过有限的几个固定线程为大量的操作服务，减少了创建和销毁线程所需要的资源和时间，从而提高效率。

②还提高了线程的管理

线程是稀缺资源。如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配，调优和监控。

如果一个线程所需要执行的时间非常长，就没有必要使用线程池了(不是线程池的线程不能长时间操作，是不宜)。

1. 线程池的状态

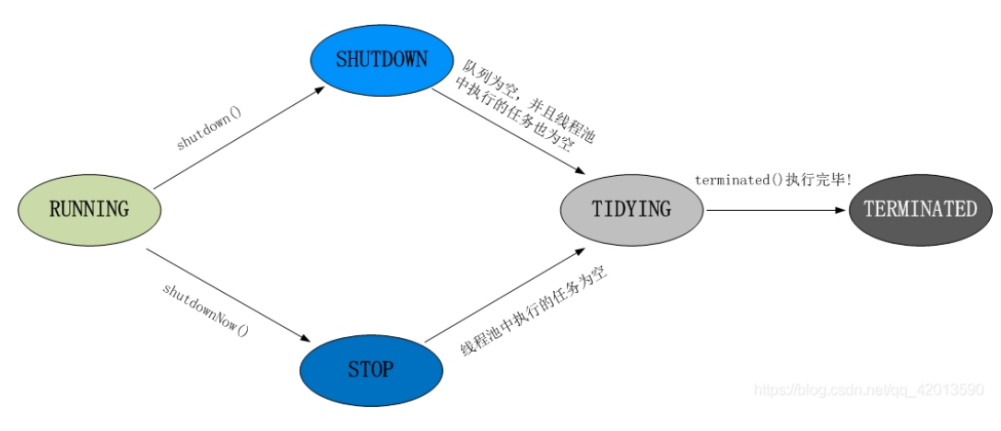
Running:正常状态，能接收新的任务，处理队列中的任务；

Shutdown：不接受新的任务提交，但是会继续处理等待队列中的任务；

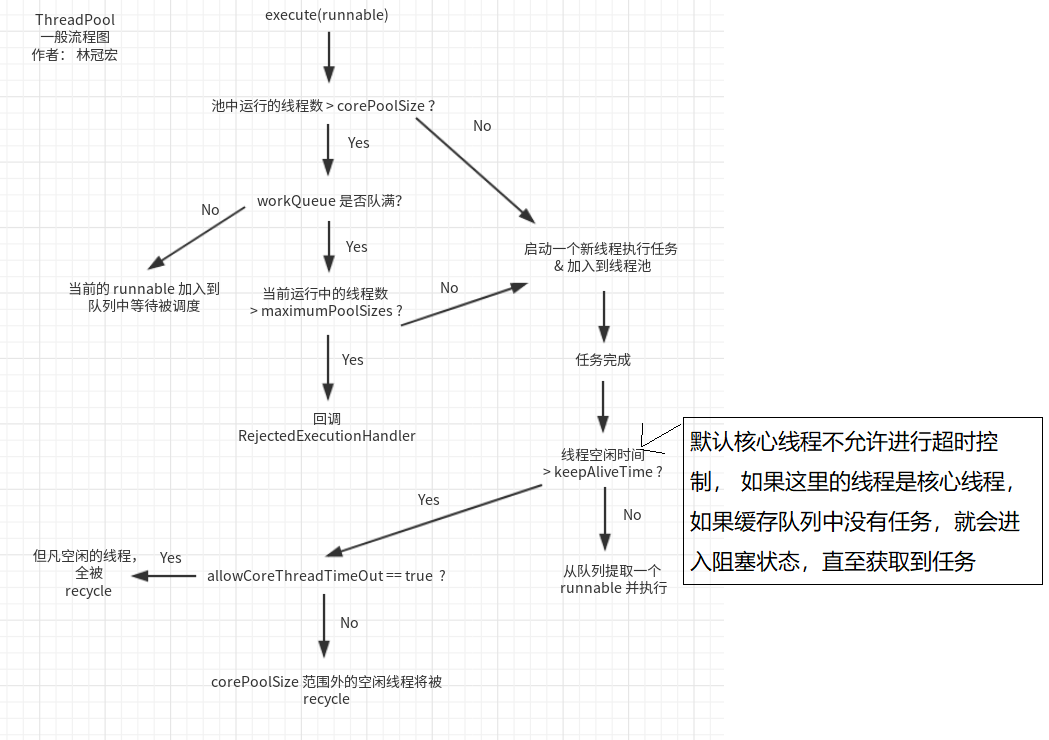
Stop:不接受新的任务提交，不在处理等待队列中的任务，中断正在执行任务的线程；调用线程池的shutdownNow()方法的时候，线程池由(RUNNING或者SHUTDOWN ) -> STOP。

Tidying：所有任务都销毁了，workCount为0，线程池的状态在转换为tidying状态时，会执行钩子方法terminated();erminated()方法在ThreadPoolExecutor类中是空的，没有任何实现。

Terminated:terminated()方法结束后，线程池的状态就会变成这个。



1. 线程池的原理/执行过程



提交一个任务到线程池中，线程池的处理流程如下：

1. 判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2. 线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3. 判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

“核心线程”、“非核心线程”是一个虚拟的概念，是为了方便描述而虚拟出来的概念，在代码中**并没有哪个线程被标记为“核心线程”或“非核心线程”，所有线程都是一样的，只是当线程池中的线程多于指定的核心线程数量时，会将多出来的线程销毁掉，池中只保留指定个数的线程。那些被销毁的线程是随机的，可能是第一个创建的线程，也可能是最后一个创建的线程，或其它时候创建的线程**。

参考文档：

<https://www.cnblogs.com/DDiamondd/p/11362164.html>

1. 线程池的使用

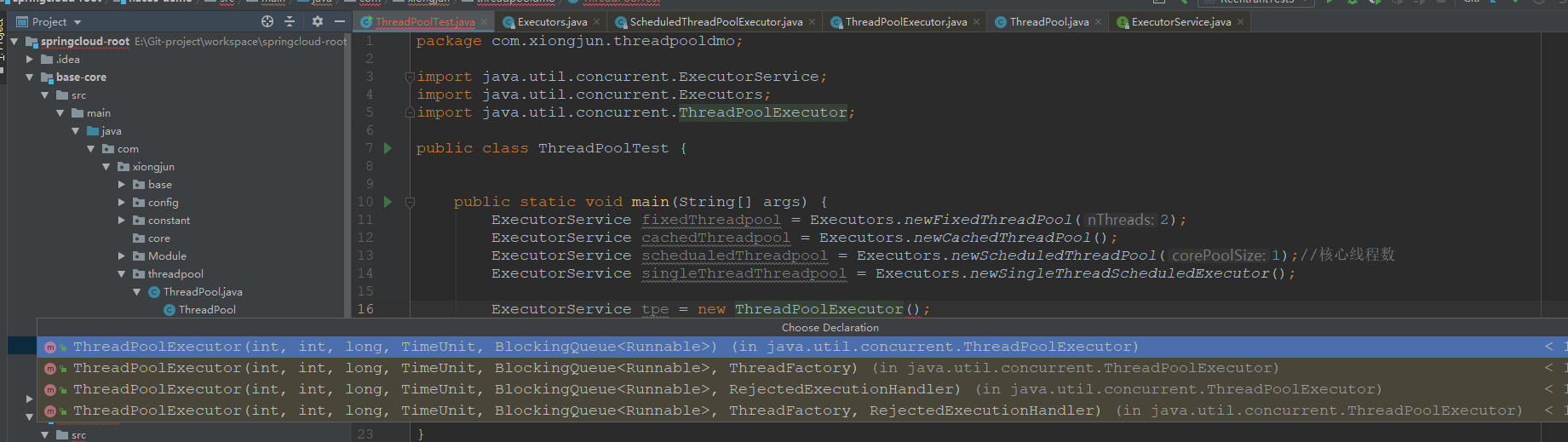
①Executors类

Executors是创建线程的基本工具类，它有5种方法来创建线程池：newFixedThreadPool、newCachedThreadPool、newSingleThreadExecutor、newScheduledThreadPool，也可以直接通过new ThreadPoolExecutor创建。其中，newFixedThreadPool与newCachedThreadPool是直接通过new ThreadpoolExecutor来创建线程的，而newScheduledThreadPool是先new newScheduledThreadPoolExecutor，然后再通过ThreadpoolExecutor创建线程；newSingleThreadExecutor是通过new FinalizableDelegatedExecutorService(new

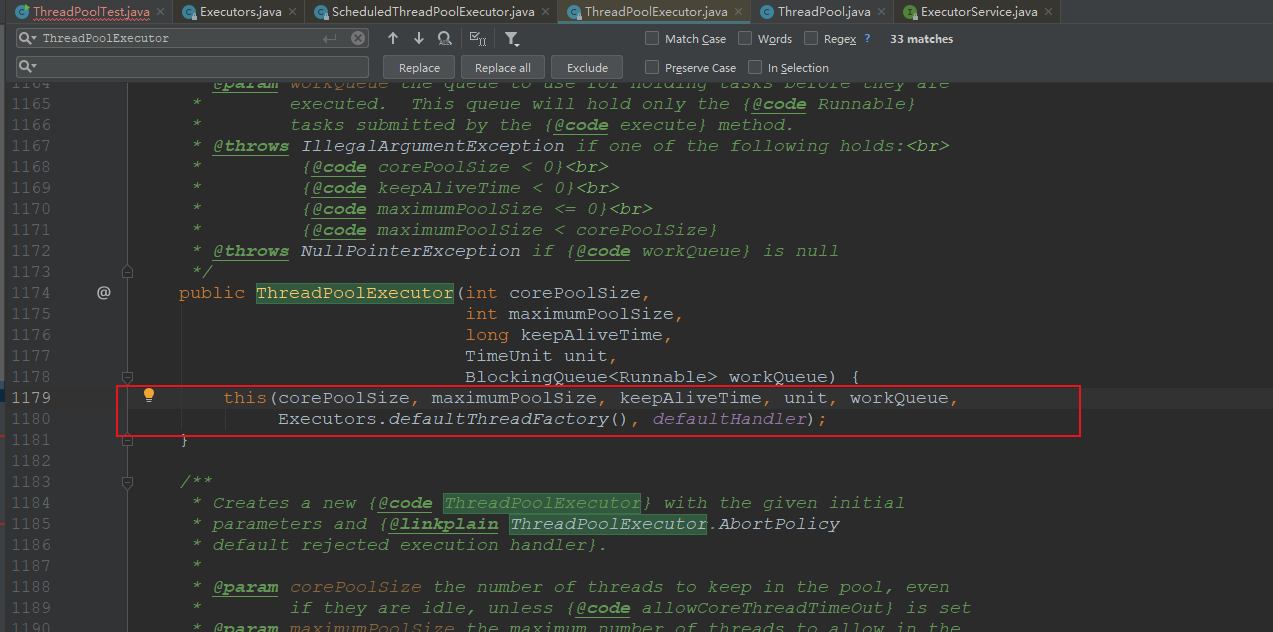
ThreadPoolExecutor(1,1...))创建的，底层还是通过new ThreadpoolExecutor创建。

②ThreadPoolExecutor类

我们来看下它的构造方法：



从上图中我们可以看出ThreadPoolExecutor类有4个构造方法，我们来看一下参数最少的构造方法：



从上图可以看出所有的构造函数都是掉了同一个构造函数，只是ThreadFactory与RejectedExecutionHandler是否是默认。下面来讲一下这些参数：

·corePoolSize 核心线程数

·maximumPoolSize 最大线程数量

·keepAliveTime 当线程数大于corePoolSize时，多余的空闲线程等待新任务的最长时间，超过这个时间，多于核心线程数量的线程就会被销毁

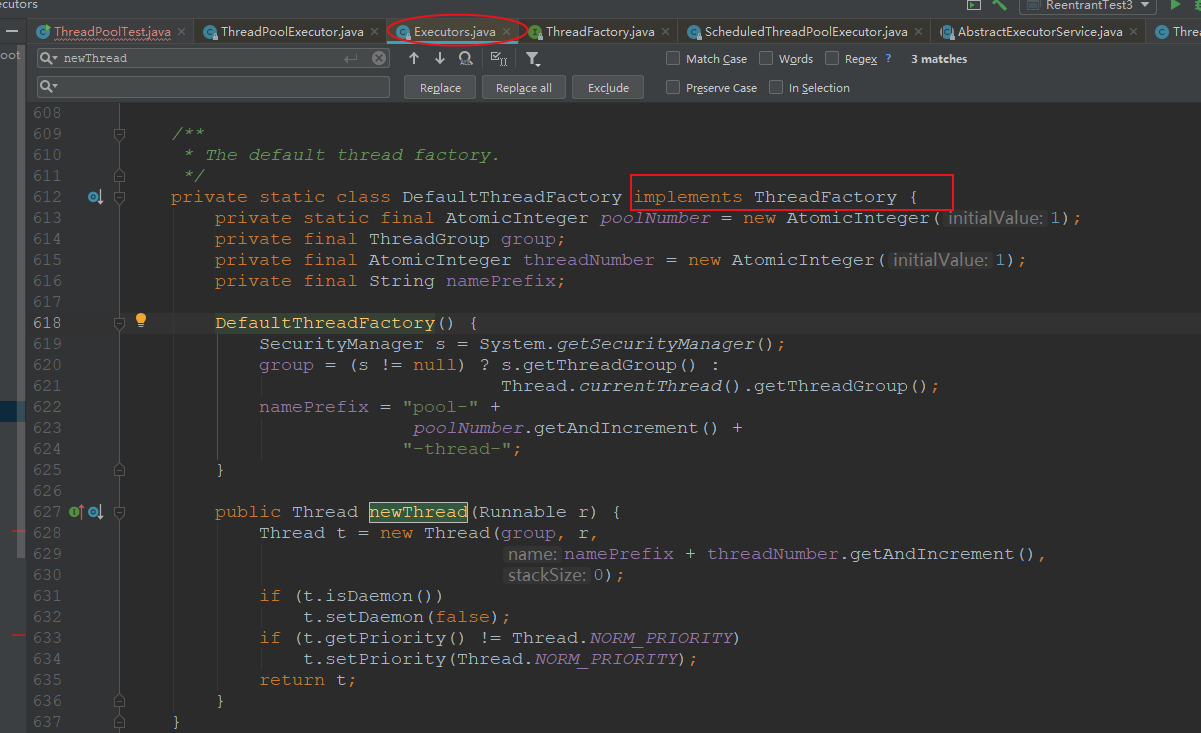
·unit 时间单位(比如秒、分等)

·workQueue 阻塞队列

·threadFactory 线程工厂

·handler 线程池拒绝策略

其中，默认的线程工厂DefaultThreadFactory类，他是Executors的静态内部类，它实现了ThreadFactory接口的newThread(Runnable r)方法。实现如下图：



newThread()方法中会通过isDaemon()方法判断r线程是否是守护线程，如果是守护线程，就改为非守护线程，然后判断线程的优先级是否为5，如果不是5就修改成5。

默认的拒绝策略是AbortPolicy.

③线程池4种拒绝策略

线程池种有三个重要参数，决定了拒绝策略：corePoolSize核心线程数(也是最小线程数)、workQueue阻塞队列、maximumPoolSize最大线程数。

当线程池中任务队列已满且maximumPoolSize最大线程数达到最大，此时如果还有任务过来，那么将会触发拒绝策列。他们都实现了RejectedExecutionHandler接口

·AbortPolicy

丢弃任务并抛出RejectedExecutionException异常。

·DiscardPolicy

丢弃任务，但是不抛出异常。

·DiscardOldestPolicy

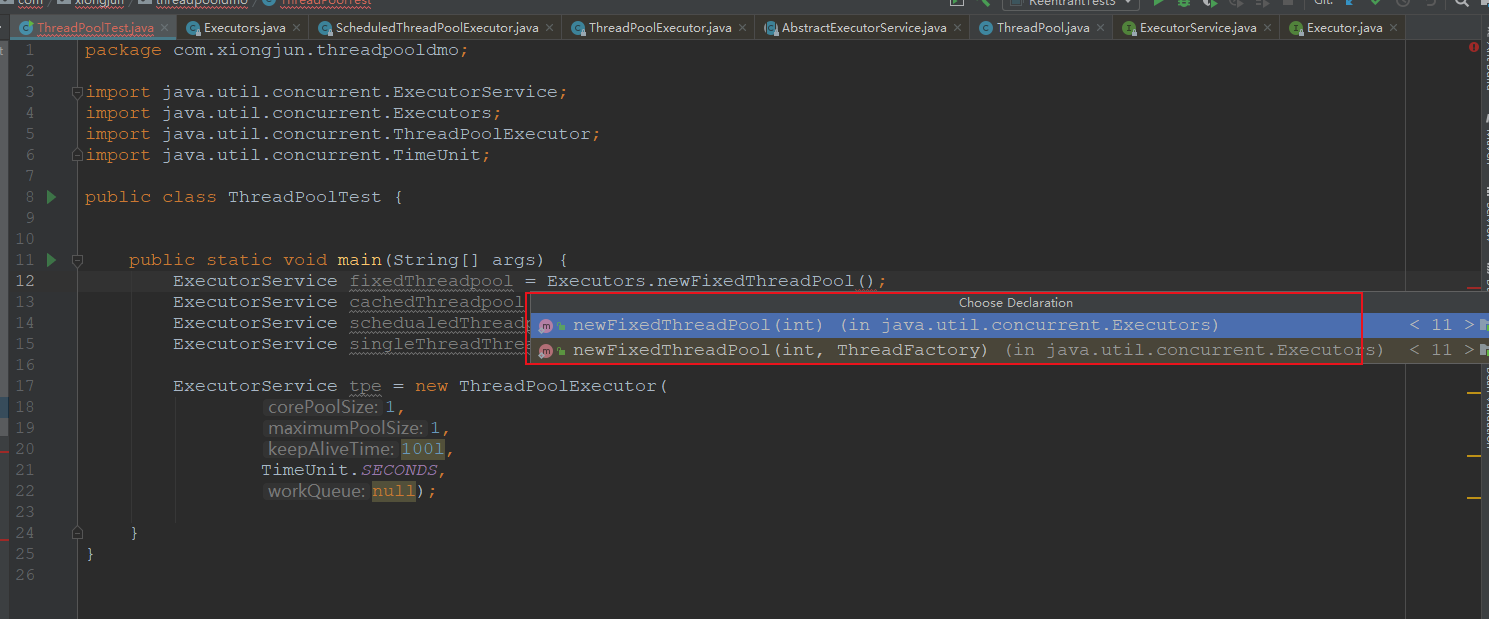
丢弃队列最前面的任务，然后重新提交被拒绝的任务。

·CallerRunsPolicy

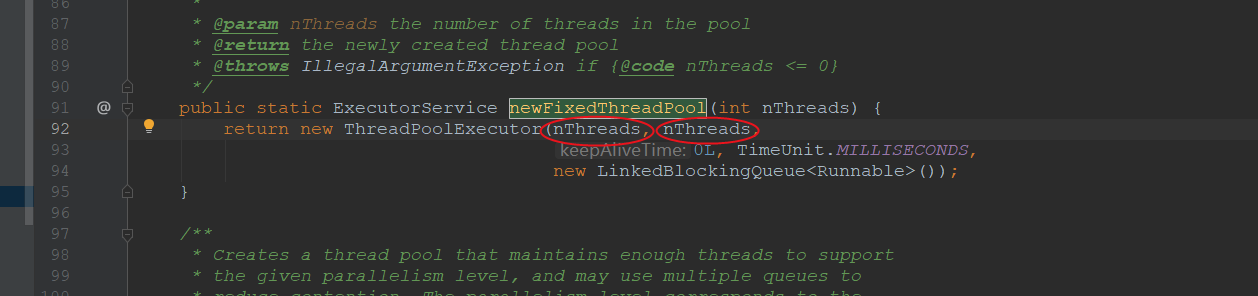
由调用线程(提交任务的线程)处理该任务

④newFixedThreadPool(int corePoolSize)

表示创建一个定长线程池，核心线程数与最大线程数相等，超出的线程会在队列中等待。



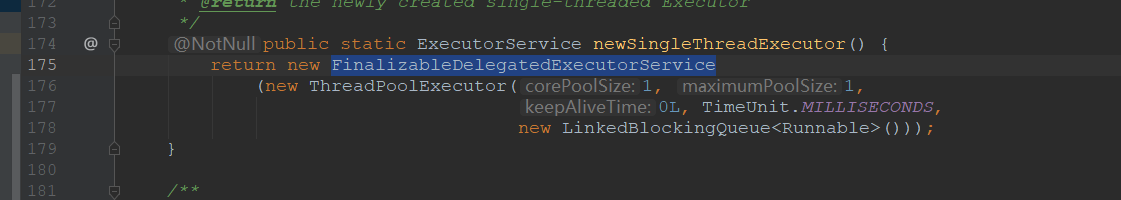
它有两个构造函数，两者相比，其中一个只是使用自建的ThreadFactofy。看一下他的构造函数实现：



这里使用的LinkedBlockingQueue<Runnable>,是一个无界阻塞队列，因此，如果有很多请求积压，会导致队列越来越长，容易导致OOM。

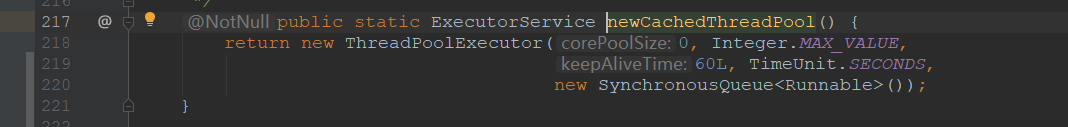
⑤newSingledThreadPool()

创建一个只有一个线程的线程池,即核心线程数和最大线程数都是1。底层如下：



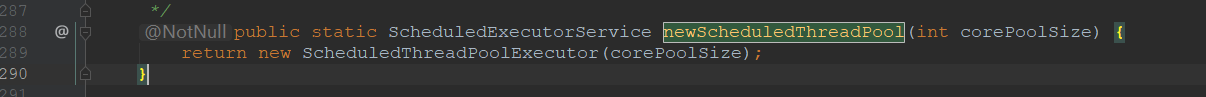
⑤newCachedThreadPool()

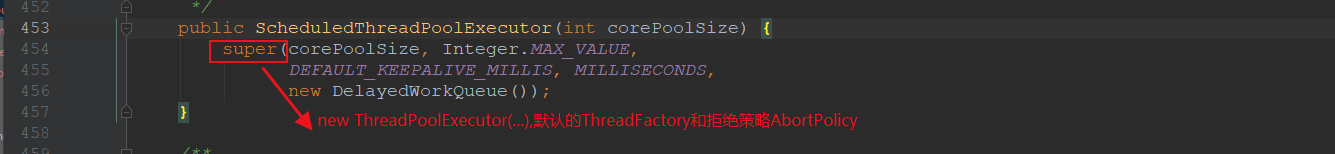
在有任务时才创建新线程，空闲线程被保留60s，最多能创建Integer.MAX\_VALUE个线程。使用了SynchronousQueue无界阻塞队列。



⑥newScheduledThreadPool(int corePoolSize)

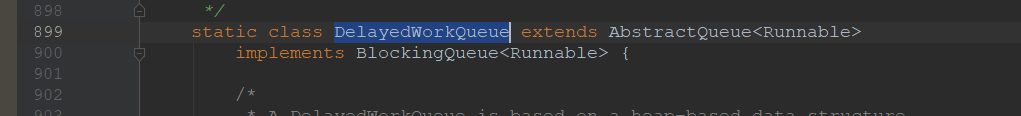
线程池能按时间计划来执行任务，允许用户设定计划执行任务的时间。参数corePoolSize设定线程池中线程的最小数目。当任务较多，线程池可能会创建更多的工作线程。底层实现如下：



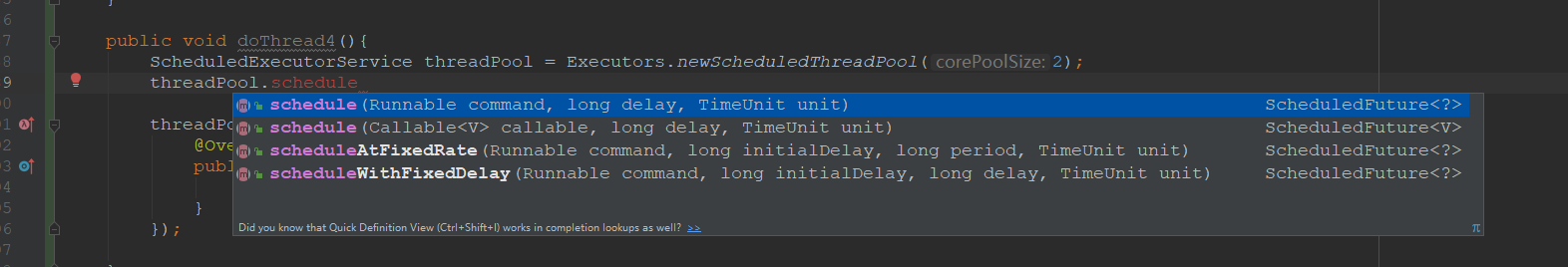


其中最大线程数时Integer.MAX\_VALUE，空闲线程被保留10ms,使用的队列是

new DelayedWorkQueue();它实现了BlockingQueue阻塞队列。



通过下列4种方式实现定时任务：



对于第一种和第二种方法中，long delay是从现在开始延迟单位时间后执行线程；

long initialDela:初始化延迟时间，即从现在开始延迟单位时间后第一次执行线程

long period:连续执行之间的时间间隔，是基于上一次任务的开始执行时间上增加period

long delay:下一次任务的执行时间都是上一次任务执行完后(如果在线程执行完业务之后，又睡眠了n秒，那就得再加n秒)，延迟delay时间后执行下一次任务。

⑦newSingleThreadScheduledPool()

线程池中只有一个工作线程，即相当于newScheduledThreadPool(1);他能按时间来执行任务。

1. 如何合理分配线程池大小？

要合理的分配线程池的大小要根据CPU密集和IO密集和分配。

①什么是CPU密集？

CPU密集的意思是该任务需要大量的运算，而没有阻塞，CPU一直全速运行。CPU密集任务只有在真正的多核CPU上才可能得到加速，而在单核CPU上，无论你开几个模拟的多线程，该任务都不可能得到加速，因为单核CPU总的运算能力就那样。

②什么是IO密集?

IO密集就是该任务需要大量的IO，即大量的阻塞。在单线程上运行IO密集型的任务会导致浪费大量的CPU运算能力浪费在等待，所以在IO密集型任务中使用多线程可以大大的加速程序运行，即使在单核CPU上，这种加速主要就是利用了被浪费掉的阻塞时间。

③分配CPU和IO

CPU密集型时，任务可以少配置线程数，大概和机器的cpu核数相当，这样可以是的每个线程都在执行任务。

IO密集型时，大部分线程都阻塞，需要多分配线程数，2\*cpu核数。

1. 并发容器与并发队列
2. 进程使用的并发容器：

Vector、ConcurrentHashMap、HashTable,这几个容器都是线程安全的。在多线程开发中需要使用的容器必须是加锁(同步)的。

1. 常用并发工具类：

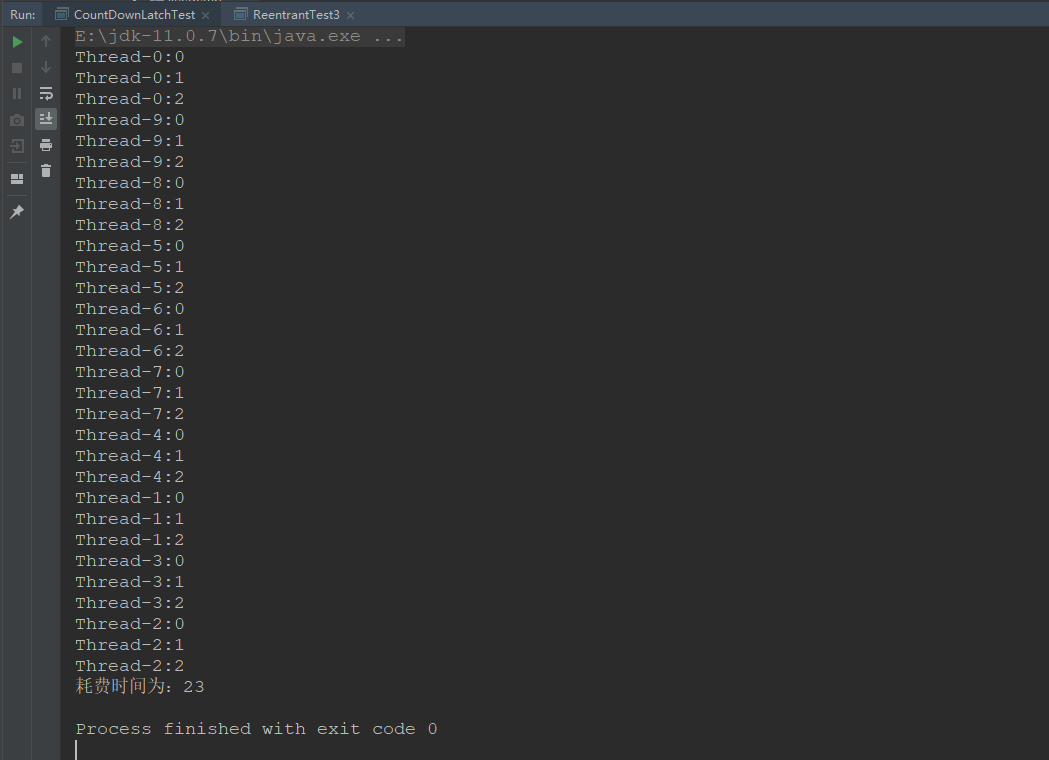
①CountDownLanch

CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

与CountDownLatch的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。使用示例如下：



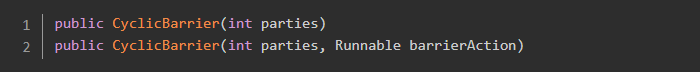
执行结果如下：



②CyclicBarrier(回环栅栏)

他是一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点（common barrier point）,再全部同时执行。阻塞子线程，当阻塞数量达到定义的参与线程后，才可以继续向下执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。叫做栅栏，大概是描述所有线程被栅栏挡住了，当都达到时，一起跳过栅栏执行，，我们把这个状态叫做barrier。

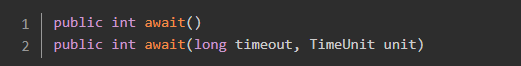
CyclicBarrier有两个构造方法，如下图：



第一个参数表示一起执行的线程个数；

第二个参数表示所有子线程都处于barrier时，先执行一次线程barrierAction,执行完之后然后再一起同时接着执行所有子线程(包括之前被阻塞的线程)。

让线程处于barrier状态的方法时await();如下图：

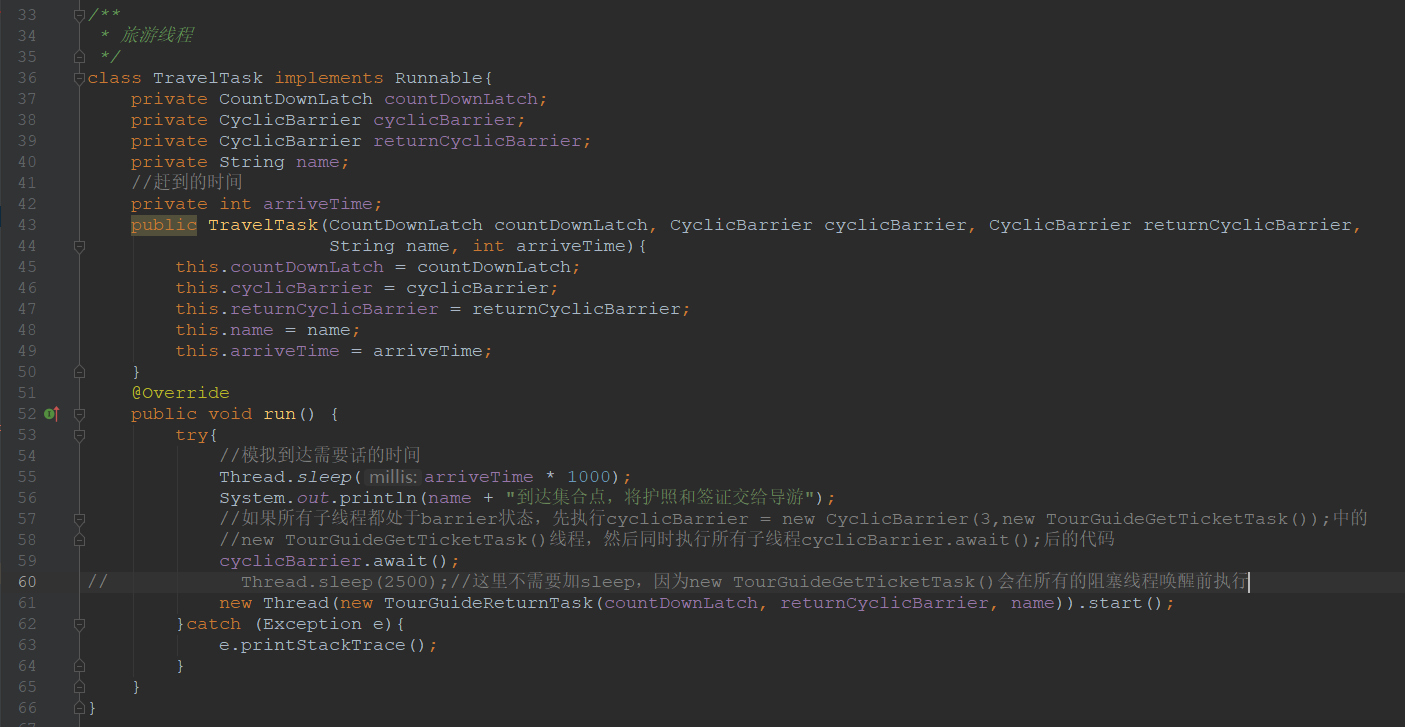


第一个是默认方法，表示等到所有线程都处于barrier状态，才一起执行。

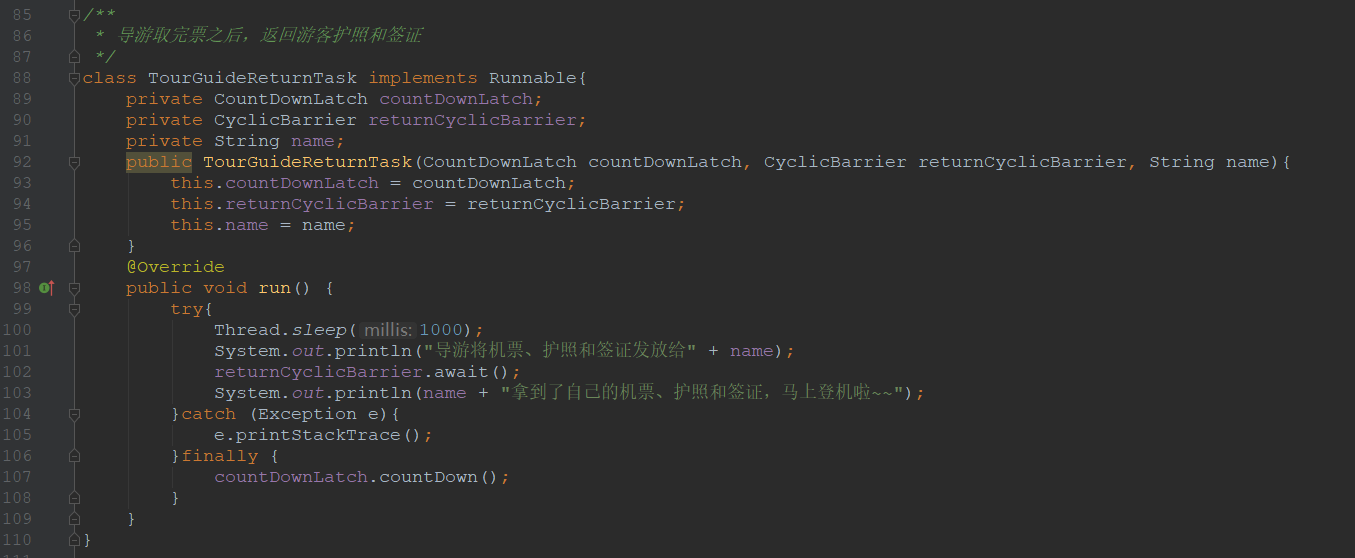
第二个方法，指定了等待的时间，当所有线程没有都处于barrier状态，又到了指定的时间，所有阻塞的线程可以继续执行了。

**使用场景**：(报旅行团旅行)

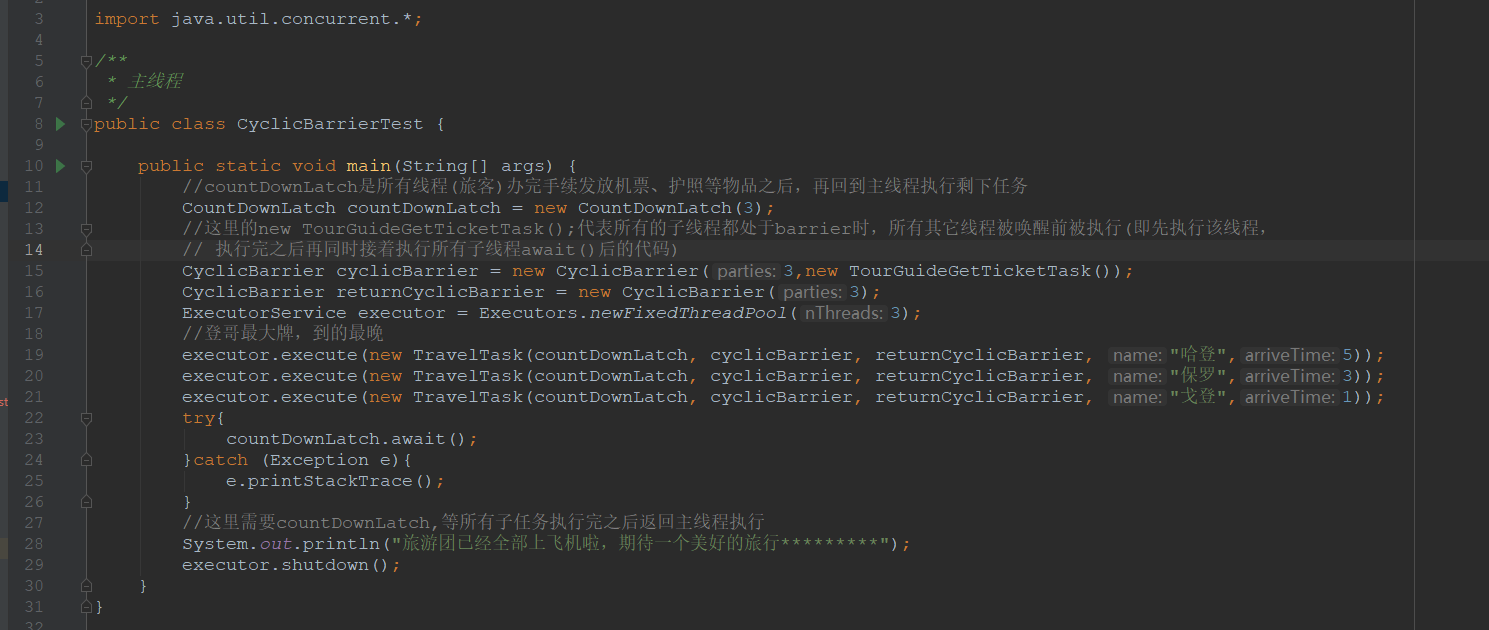
出发时，导游会在机场收了所有人的护照和签证，办理集体出境手续，办完手续之后，再把机票、护照和签证等发到大家手里，然后大家就可以登机旅行啦。具体代码如下：



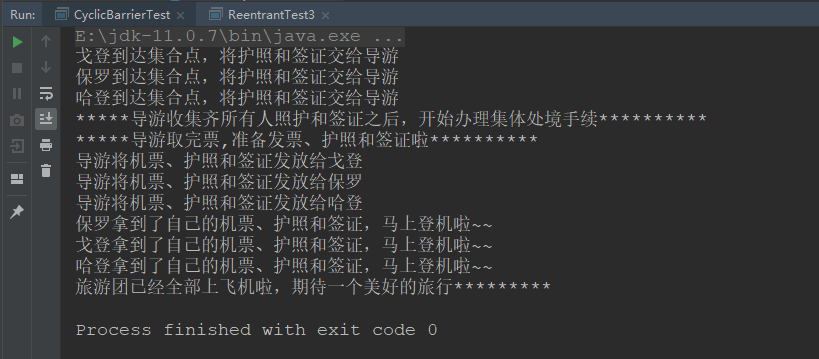




Main方法



执行结果：



③Semaphore(信号量)

Semaphore是synchronized的加强版，作用是控制线程的并发数量(允许自定义多少线程同时访问)。这个是synchronized实现不了的。

构造方法：



permits表示允许并发的线程数；

fair表示公平性，如果这个设为ture，表示下次执行的线程会是等待最久的线程。

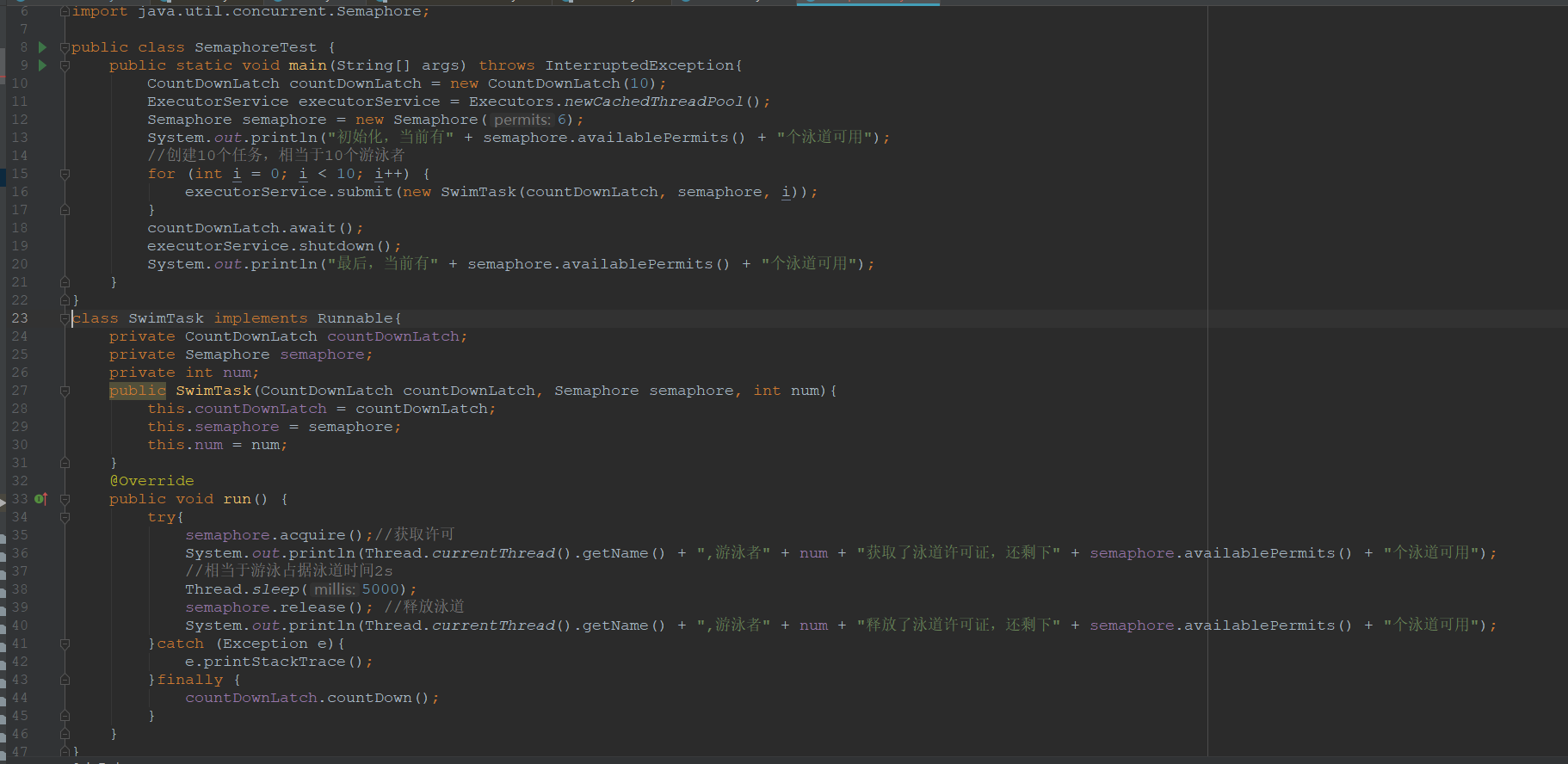
重要方法：



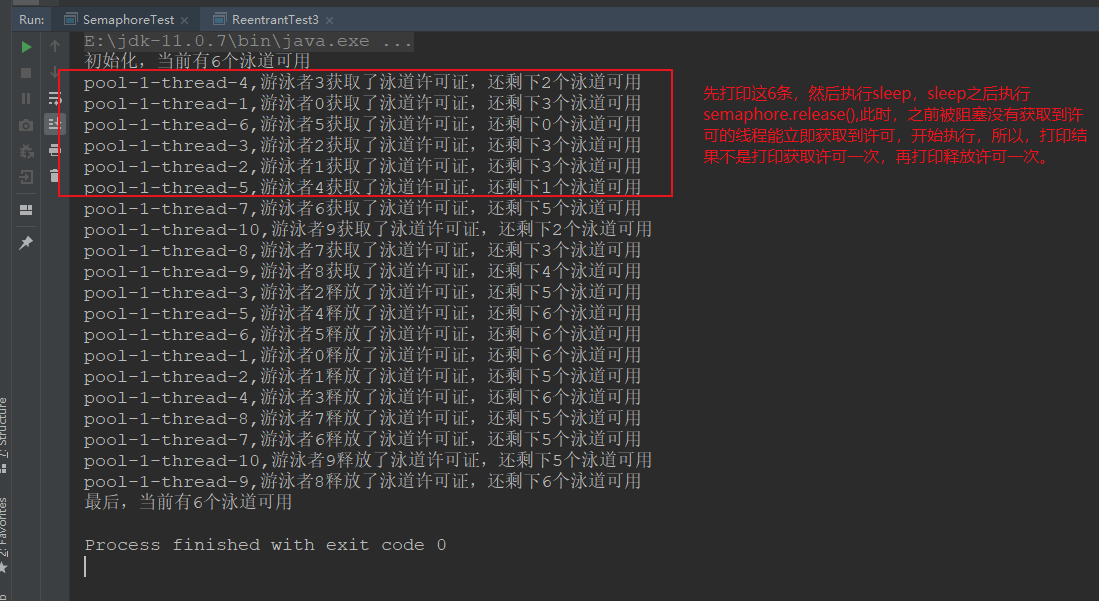
acquire()表示阻塞并获取许可

release()表示释放许可。

使用示例如下：



执行结果：



Semaphore 只是对资源并发访问的线程数进行监控，并不会保证线程安全。

④总结

CountdownLatch的计数器只能用一次，而CyclicBarrier得计数器可以使用reset()重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

CountdownLatch和CyclicBarrier得await()方法放在哪里哪个线程就被阻塞，直到达到释放许可的条件。

CyclicicBarrier实现线程同步阻塞使用的是ReentrantLock,可重入锁。实现线程间通信使用的Condition。

**使用场景对比**：

需要等待某个条件达到要求后才能做后面的事情；同时当线程都完成后也会触发事件，可以使用CountDownLatch

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的应用场景。

Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限，作用是限制某段代码块的并发数。

1. Java中各个线程是怎么彼此看到对方的变量的呢？

Java中定义了主内存和工作内存。所有的变量都存储再主内存，每条线程还有自己的工作内存，保存了被该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作（读写）都必须再工作内从中运行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

1. volatile关键字？