1. 线程的几种状态

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。

新建(new)

就绪 (调用了start()方法，表示等待资源分配)

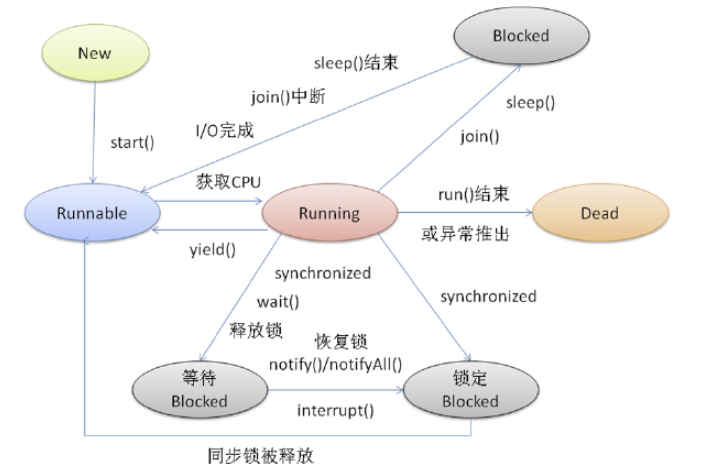
运行(表示分配到了cpu资源)

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())

死亡(线程销毁)



2、sleep()与wait()区别？

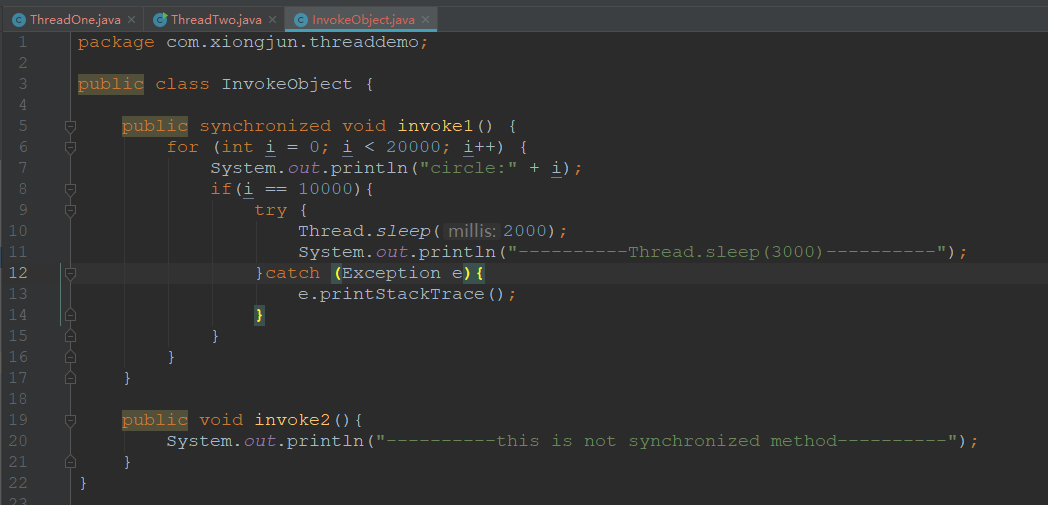
线程在调用sleep()后主动让出cpu，但是还会占有锁，当sleep指定的时间过后，该线程进入可执行状态，要看cpu是否空闲，空闲才会立即继续执行sleep()后面的程序；

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。

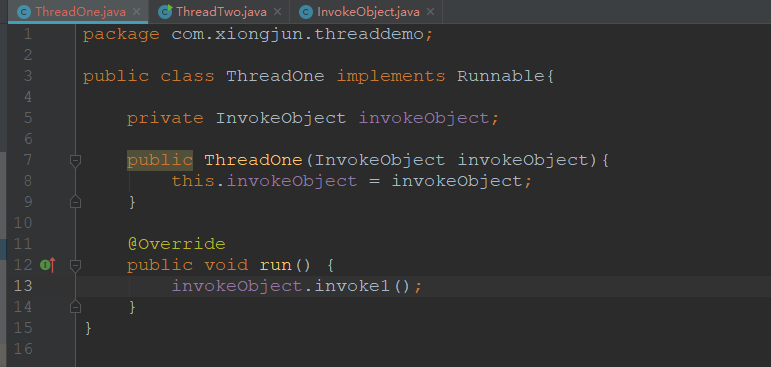
3、当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法？

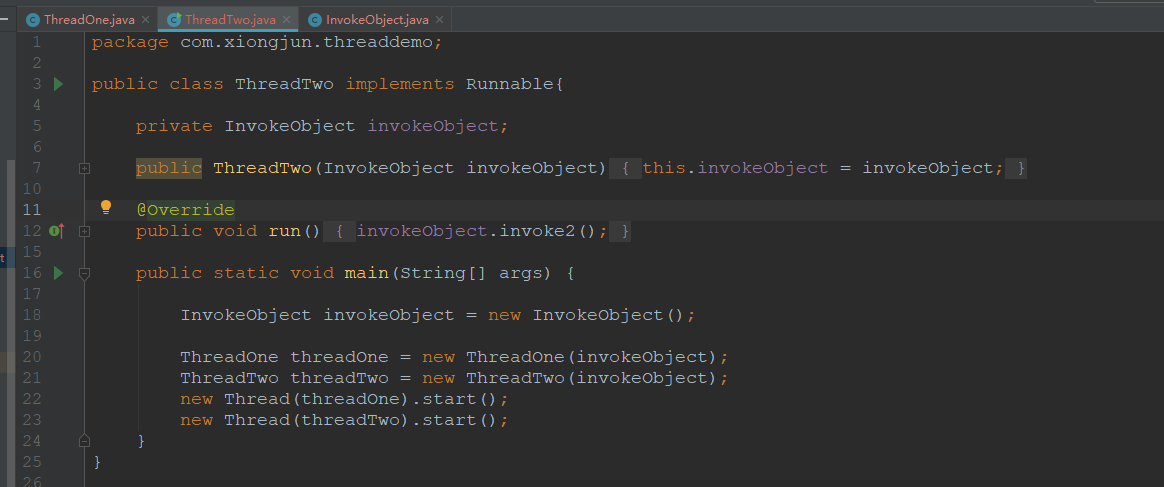
分三种情况：

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：

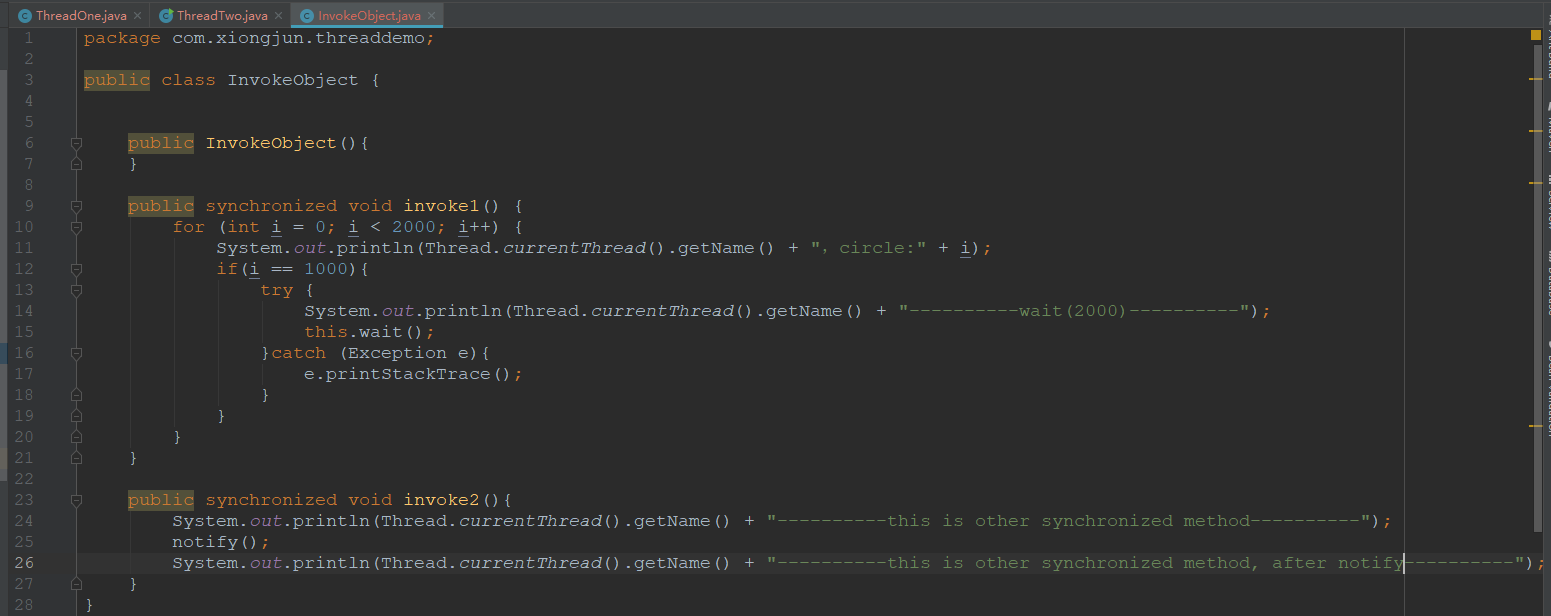


两个线程类：

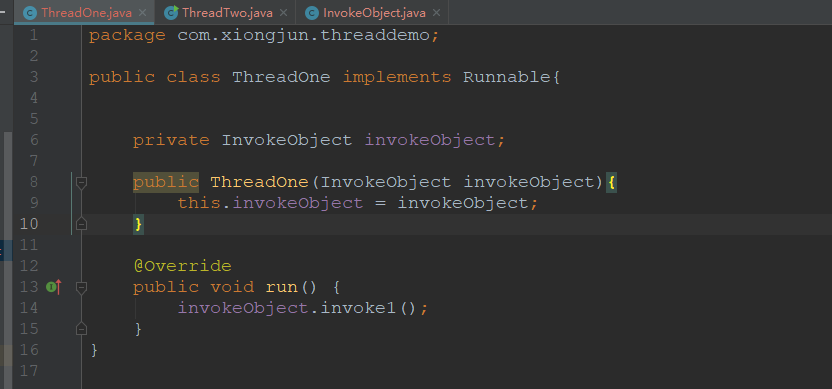


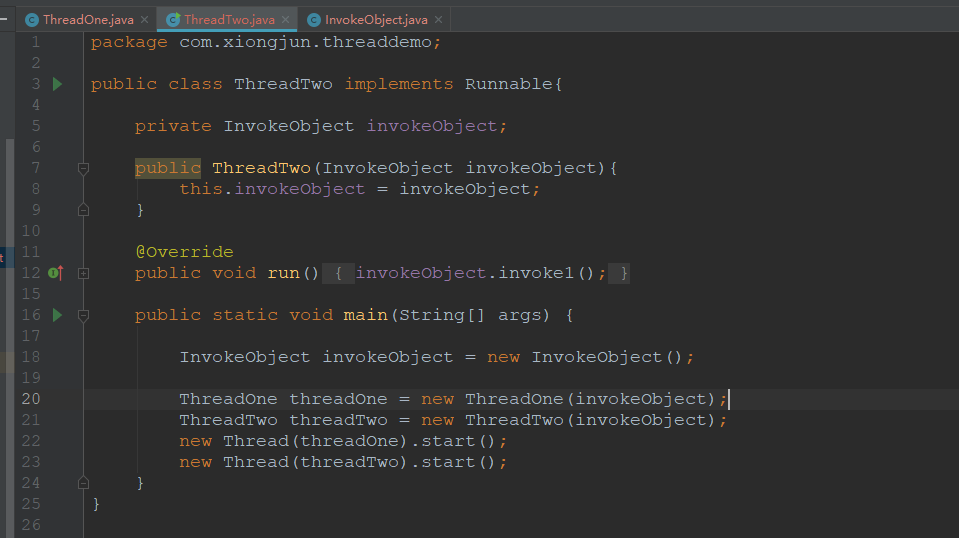


情况②：如果这个方法内部调用了wait，则可以进入其他的synchronized方法，此时是同步执行。如果没有调用wait,则不能进入其他的synchronized方法。如下：

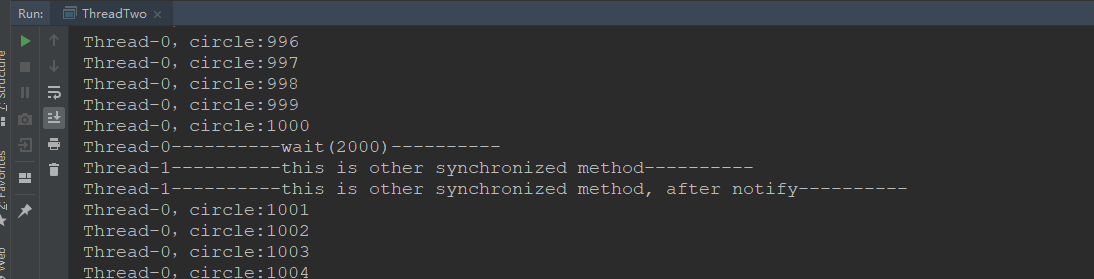


两个线程类，调用不同的synchronized方法：



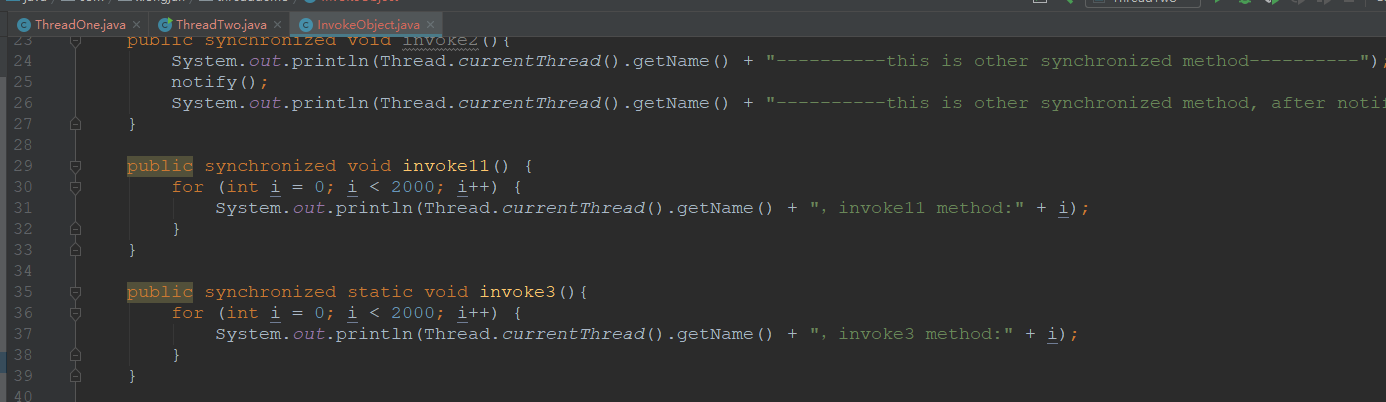


执行结果如下：

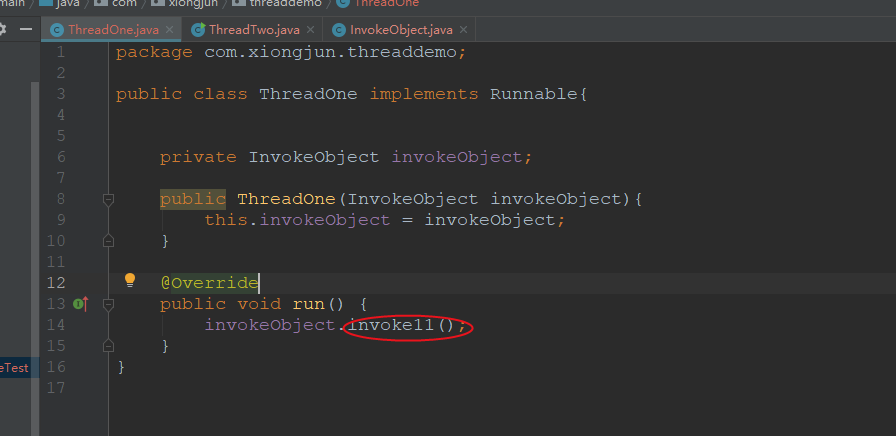


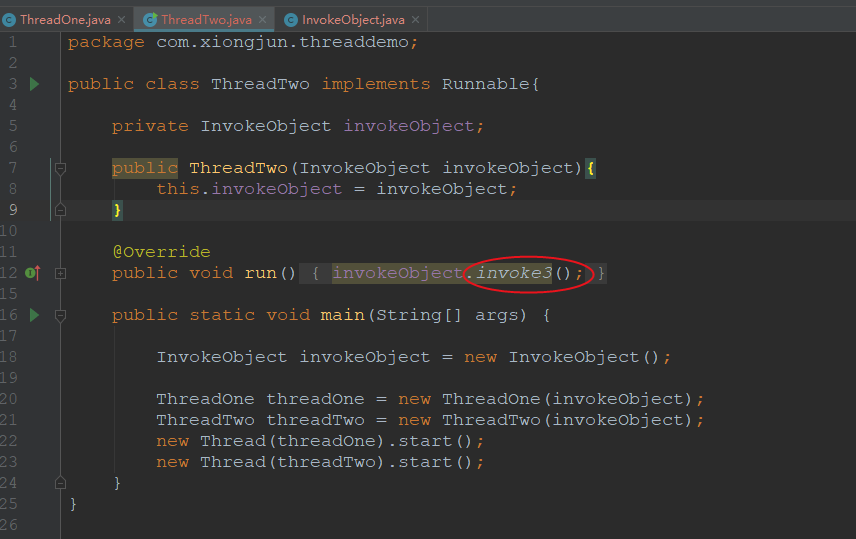
解析：threadOne线程先获取到invokeObject对象锁，执行invoke1()同步方法，当执行wait(),threadOne线程释放锁，threadTwo线程此时获取到invokeObject对象锁，执行invoke2()同步方法，执行notify()之后，threadOne线程此时变成唤醒状态，可以竞争invokeObject对象锁了，等invoke2()同步方法执行完之后，threadTwo线程释放invokeObject对象锁，threadOne线程获取到invokeObject对象锁之后继续执行完invoke1()同步方法。

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：

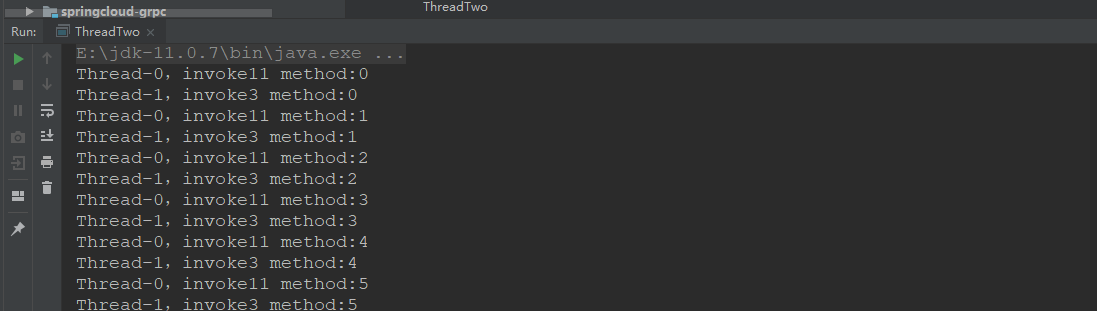


两个线程类：





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()

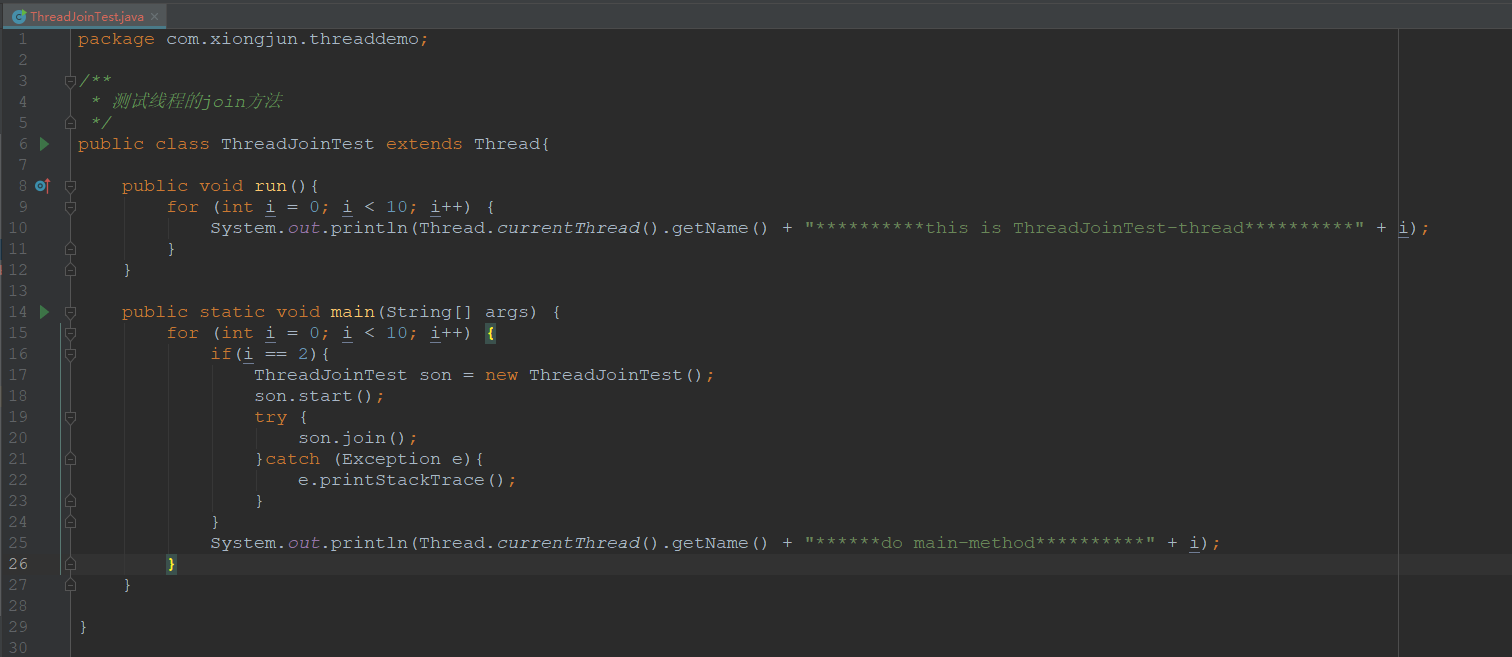
java中线程优先级priority范围是1~10，默认为5，数值越大，优先级越高。优先级越高，线程分配到CPU资源的概率越大。无论是优先级相同或不同，线程调用都不糊i绝对按照优先级执行。

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

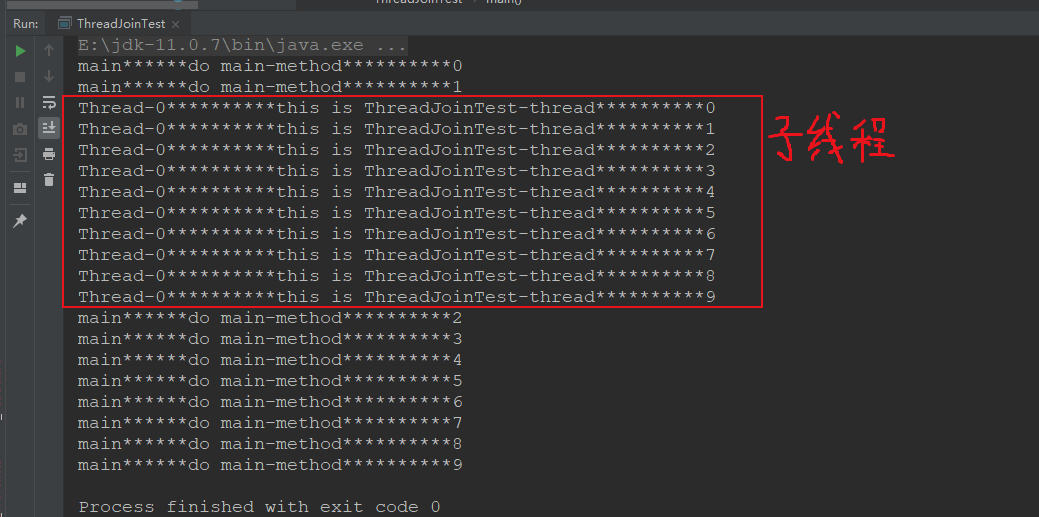
线程让步是调用yield()方法，表示当前线程愿意让出CPU，但是不能保证当前线程一定不继续执行，即不能保证其他线程线执行。

1. 线程的join()方法

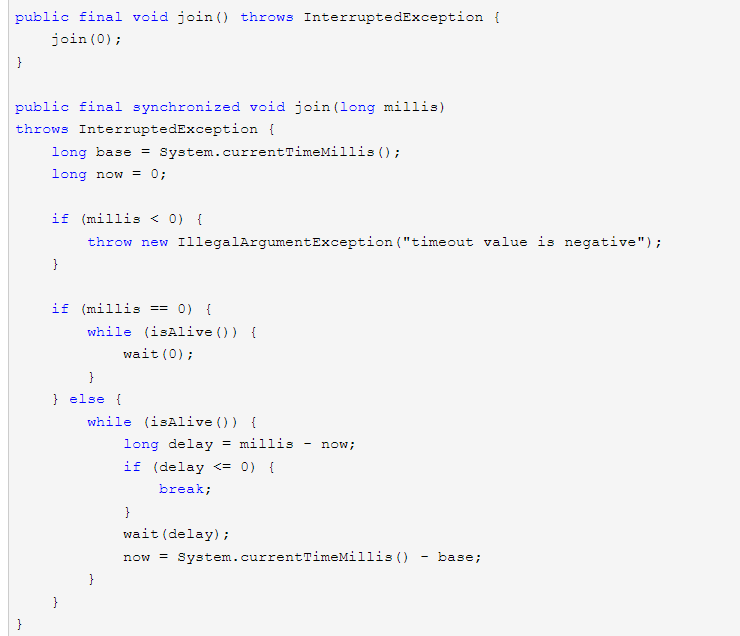
Join()是在“主线程”中插入“子线程”，等“子线程”结束之后“主线程”才能继续运行。示例如下图：



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...;

}

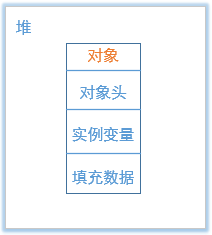
}

}

1. synchronized原理

synchronized是由JVM实现互斥同步的一种方式。Java虚拟机中的同步是基于进入和退出管程(Monitor)对象实现的，无论是显示同步(有明确的monitorenter和monitorexit指令)还是隐式同步都是如此。同步代码块是由monitorenter和monitorexit指令来实现同步的，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>

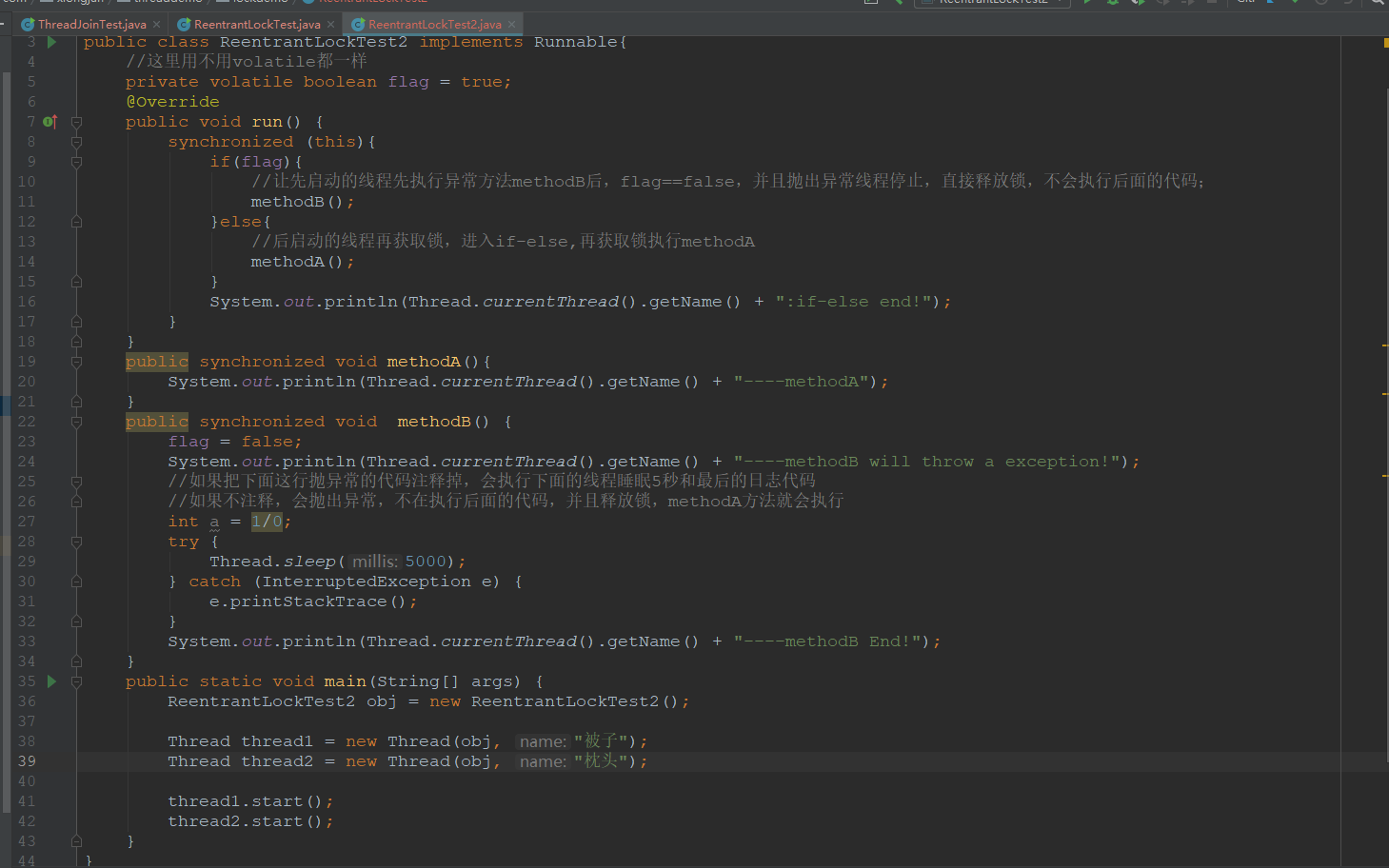
1. 锁的相关概念

重入锁：

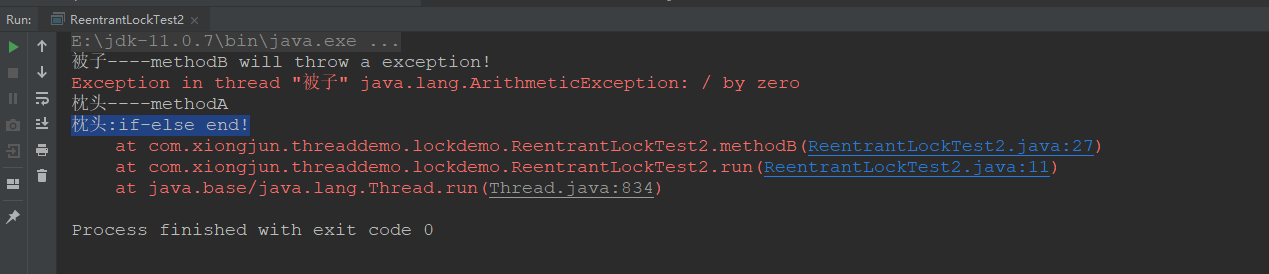
是一种递归无阻塞的同步机制，也叫做递归锁,它可以有效防止死锁。指的是同一线程万层函数获得锁之后，内层递归函数任然由获取该所的代码，但不受影响。Java中synchronized和ReentrantLock是可重入锁。

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。

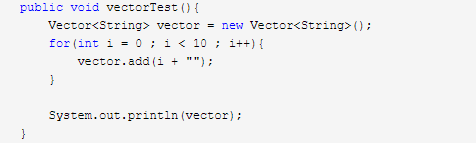
JDK1.6中，java虚拟机提供-XX:+UseSpinning参数来开启自旋锁，使用-XX:PreBlockSpin参数来设置自旋锁等待的次数。

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：



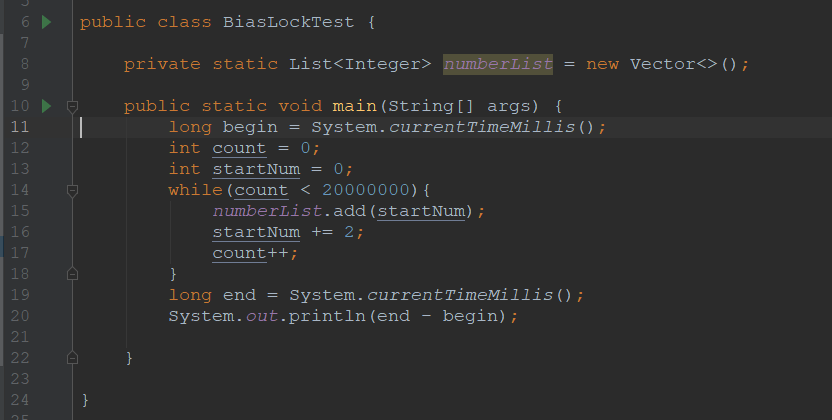
Vector是线程安全的，它的add()方法是synchronized的，但是对于上述代码中，vector是不可能被多个线程操作的，所以，vector对象不可能存在锁竞争。所以JVM会将vector对象内部的加锁操作消除。

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。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：

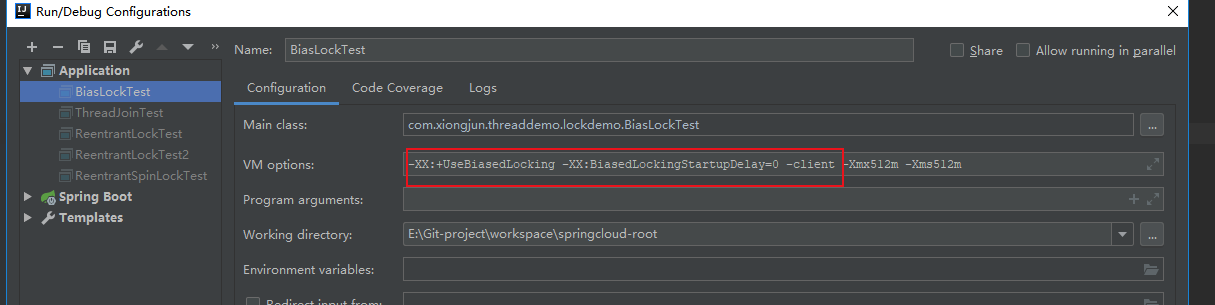
他的核心思想是，如果程序没有竞争，则取消之前已经取得锁的线程同步操作。也就是说，若对象锁被线程获取后，便进入偏向模式，当线程再次请求这个锁时，就无需再进行相关的同步操作了，从而节约了操作时间，如果在此过程中有其他线程进行了锁请求，则退出偏向模式。使用场景比如，单线程访问同步方法：



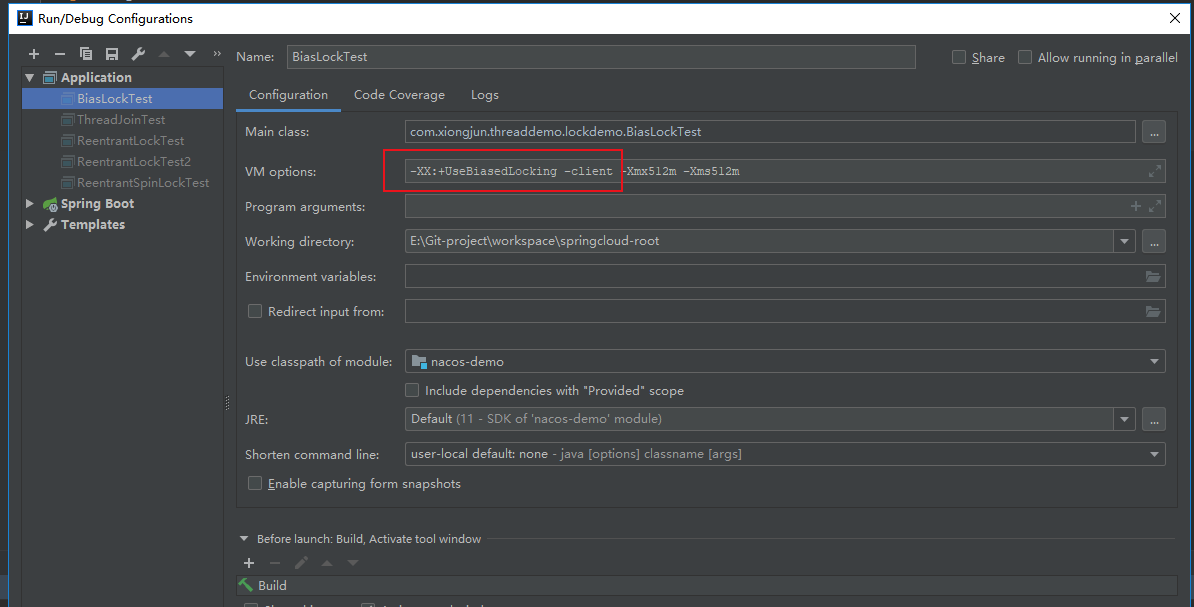
同时需要配置jvm参数开启偏向锁，

-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。

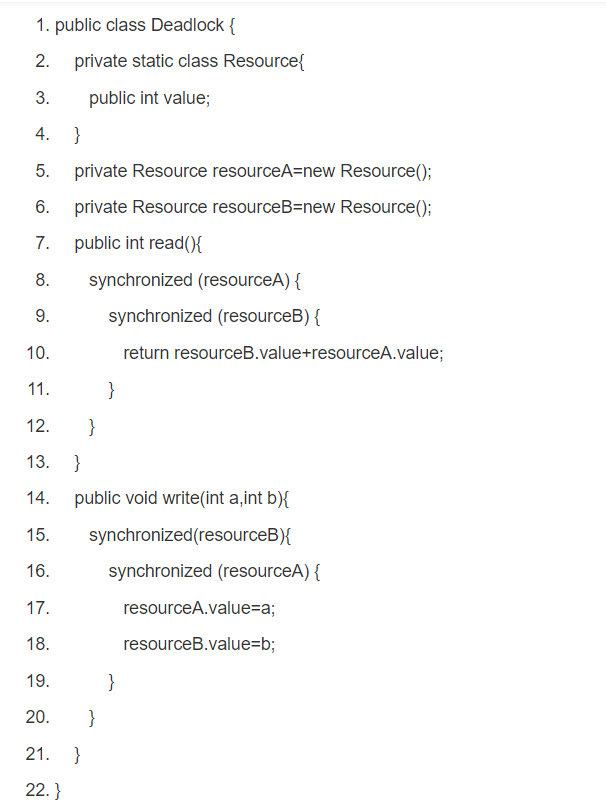
轻量级锁：

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：

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。

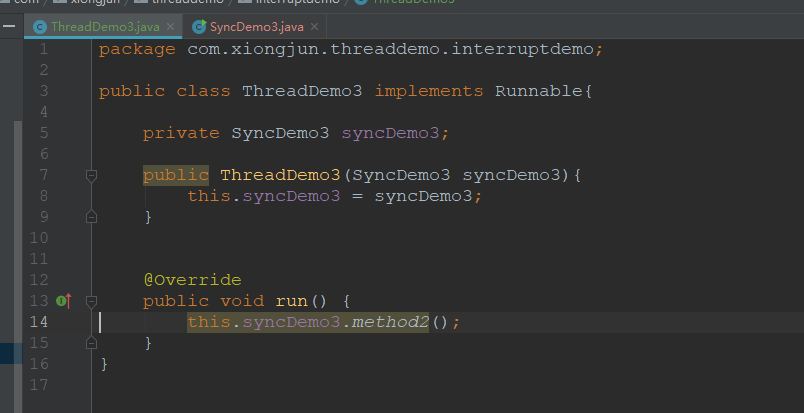
1. 线程的interrupt()方法使用

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，但是会走完当前线程，不会抛异常；

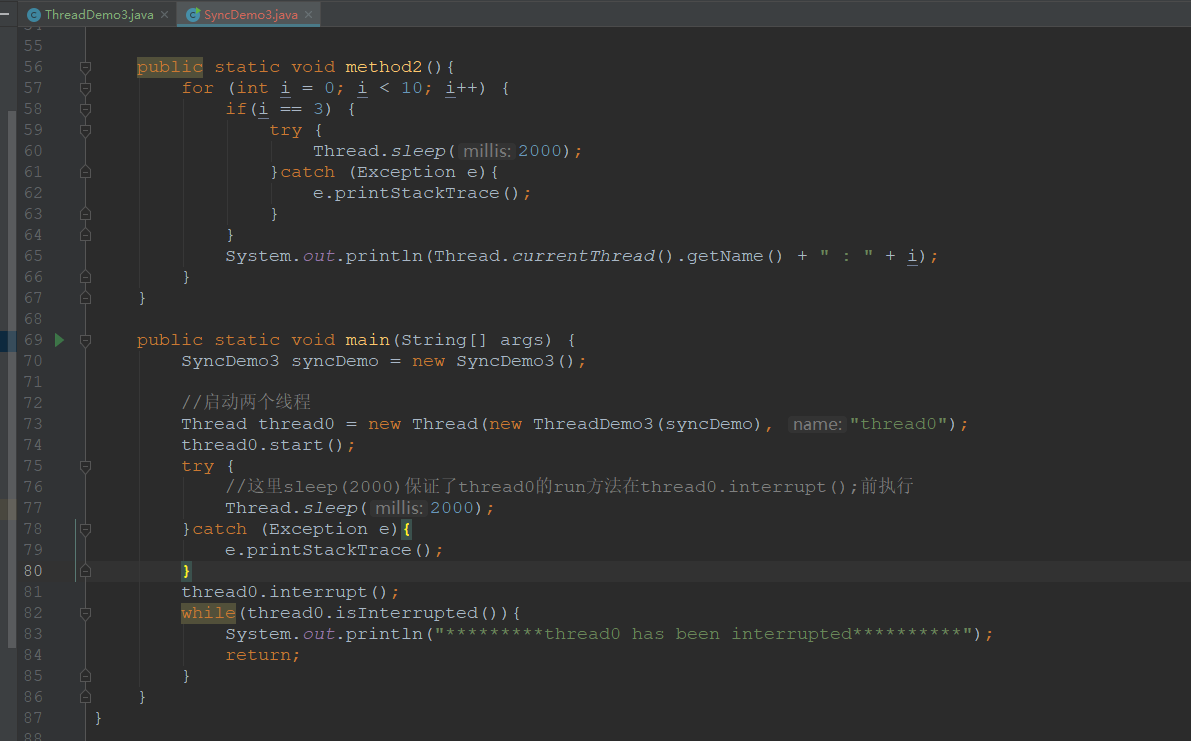
①sleep() &interrupt()一起使用

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：

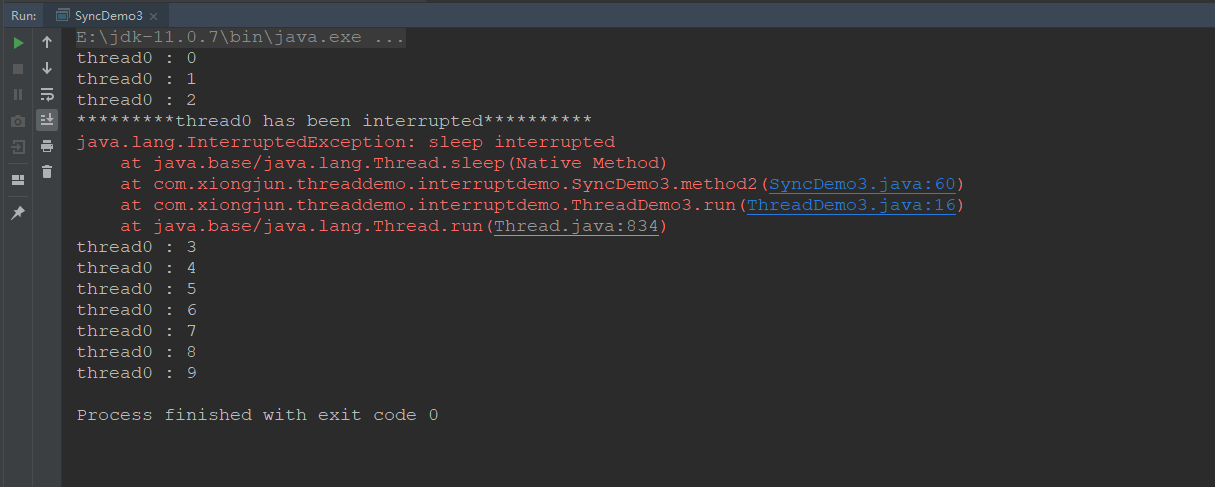
线程类：



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：



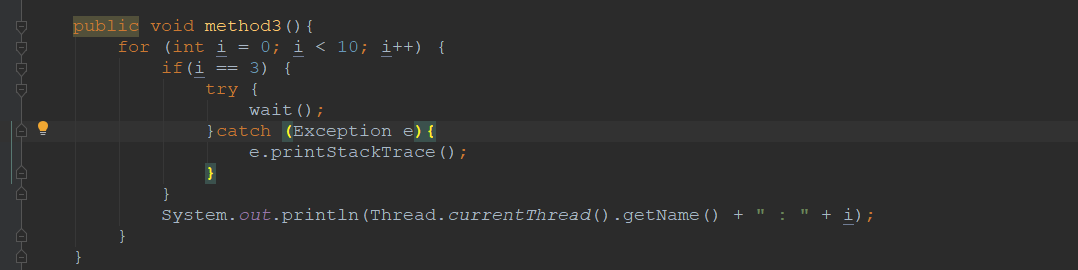
②wait() &interrupt()一起使用

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。

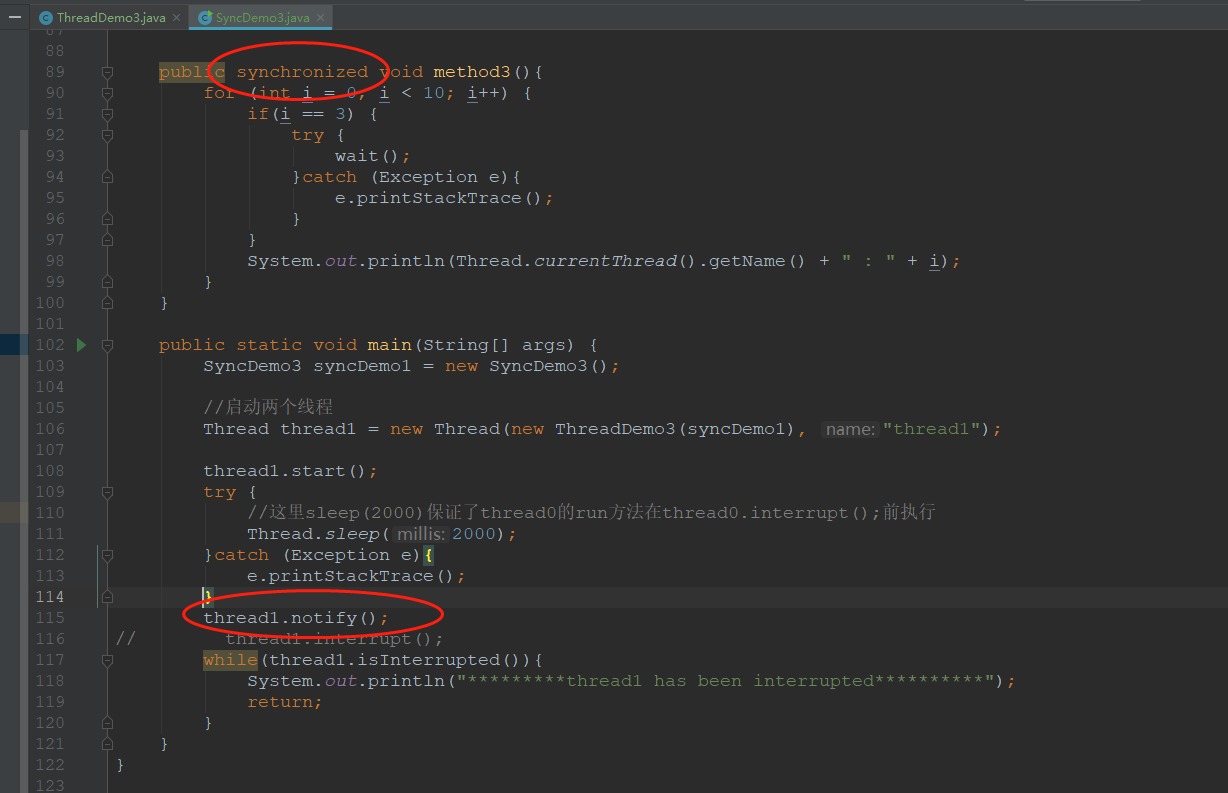
调用wait()方法就是释放锁，释放锁的前提使必须要先获得锁，而只有synchronized修饰的方法才会有对象锁；

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。

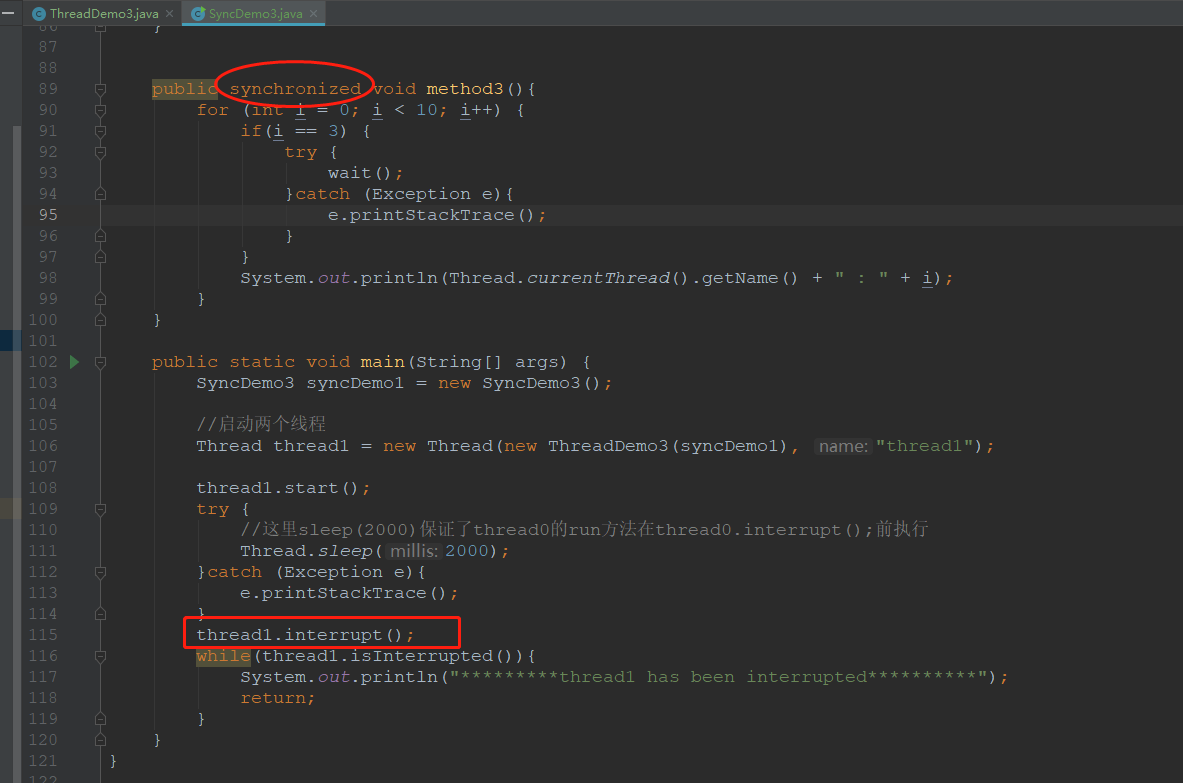
情况1：wait()在没有synchronized修饰的方法中调用



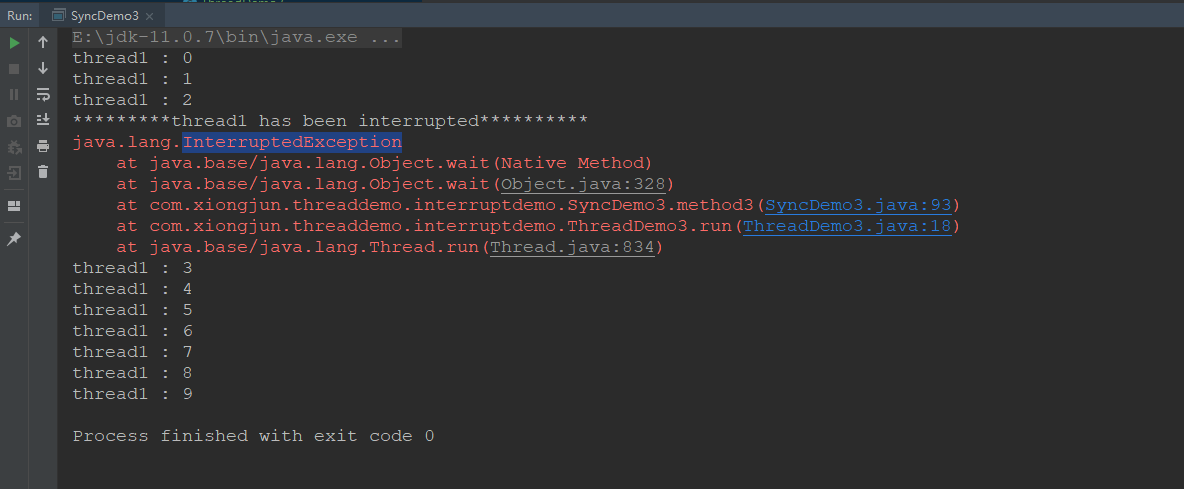
情况2：notify()或notifyAll()在没有synchronized修饰的方法中调用



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：



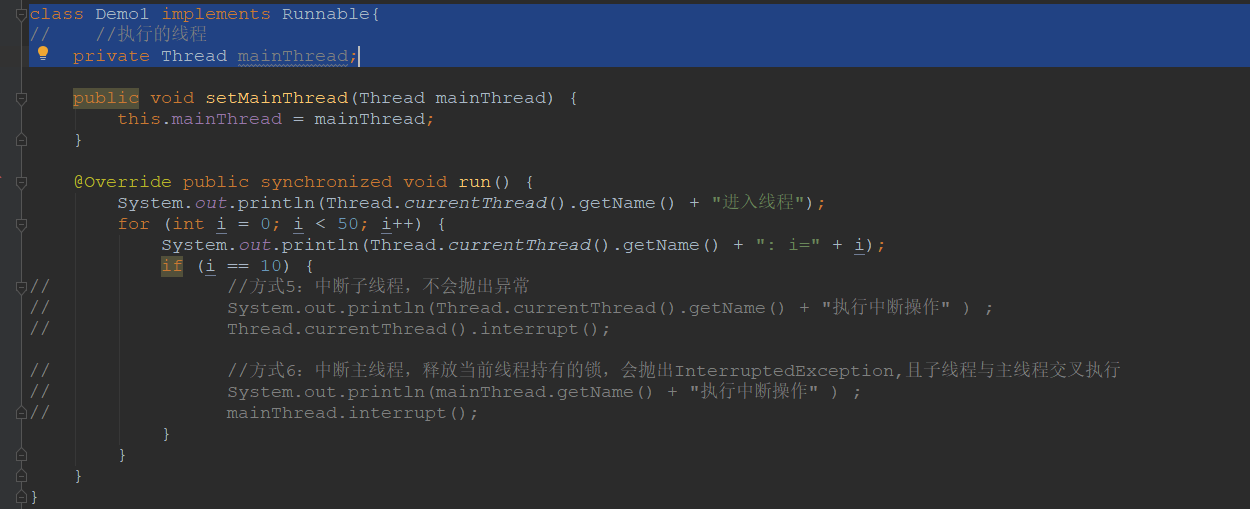
执行结果：



③join() &interrupt()一起使用

如下图的代码：

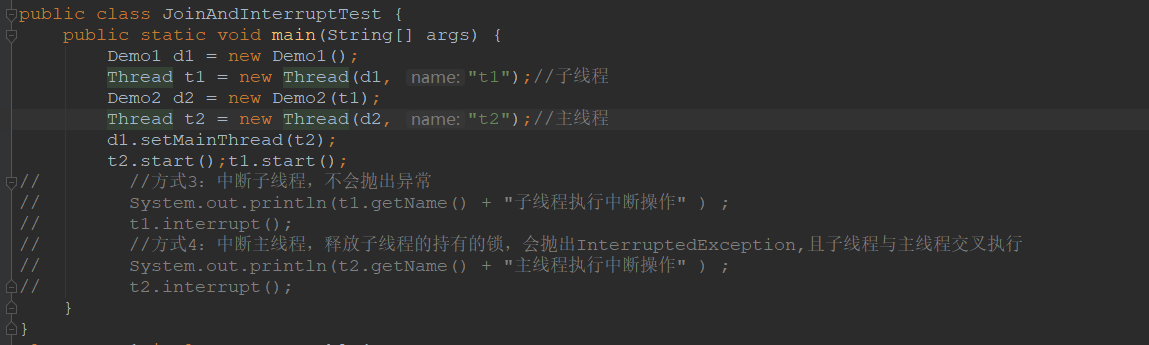
线程1：



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。

<https://www.cnblogs.com/onlywujun/p/3565082.html>

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

<https://www.cnblogs.com/rwxwsblog/p/6046034.html>

<https://www.hollischuang.com/archives/1716>

<https://blog.csdn.net/yjsz2010/article/details/82729562>

<https://www.cnblogs.com/wxd0108/p/5479442.html>