# 并发编程中的线程通信

首先,线程通信又称之为线程的等待与唤醒，线程通信的目标是使线程间能够互相发送信号。另一方便面，线程通信使线程能够等待其他线程的信号，可以实现两个或多个线程在执行任务过程中互相配合协作。

**线程通信常用的方式有：**

1、volatile 内存共享

2、wait/notify 等待唤醒

3、使用 ReentrantLock 结合 Condition

4、CountDownLatch 并发工具

5、基于 LockSupport 实现线程间的阻塞和唤醒

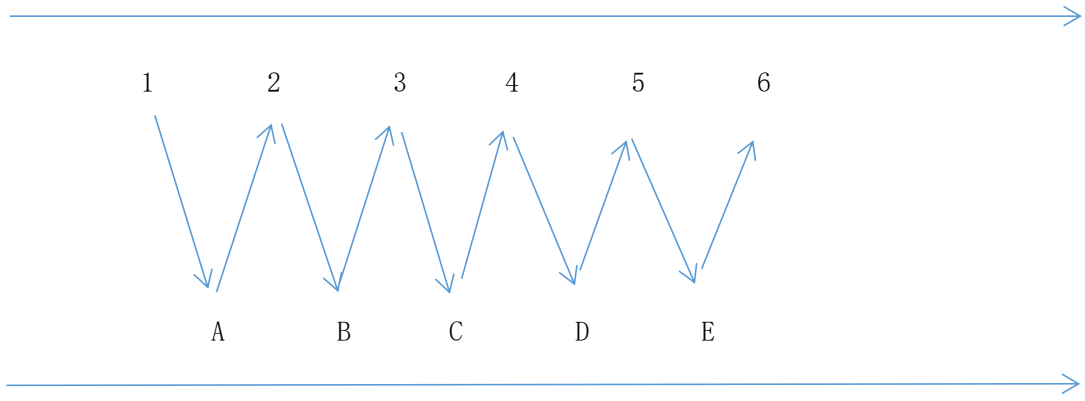
6、使用阻塞对列（BlockingQueue）控制线程通信

7、使用 ThreadLocal 机制

...

示例:

用两个线程，一个输出字母，一个输出数字，**交替输出** 1A2B3C4D5E6F...26Z



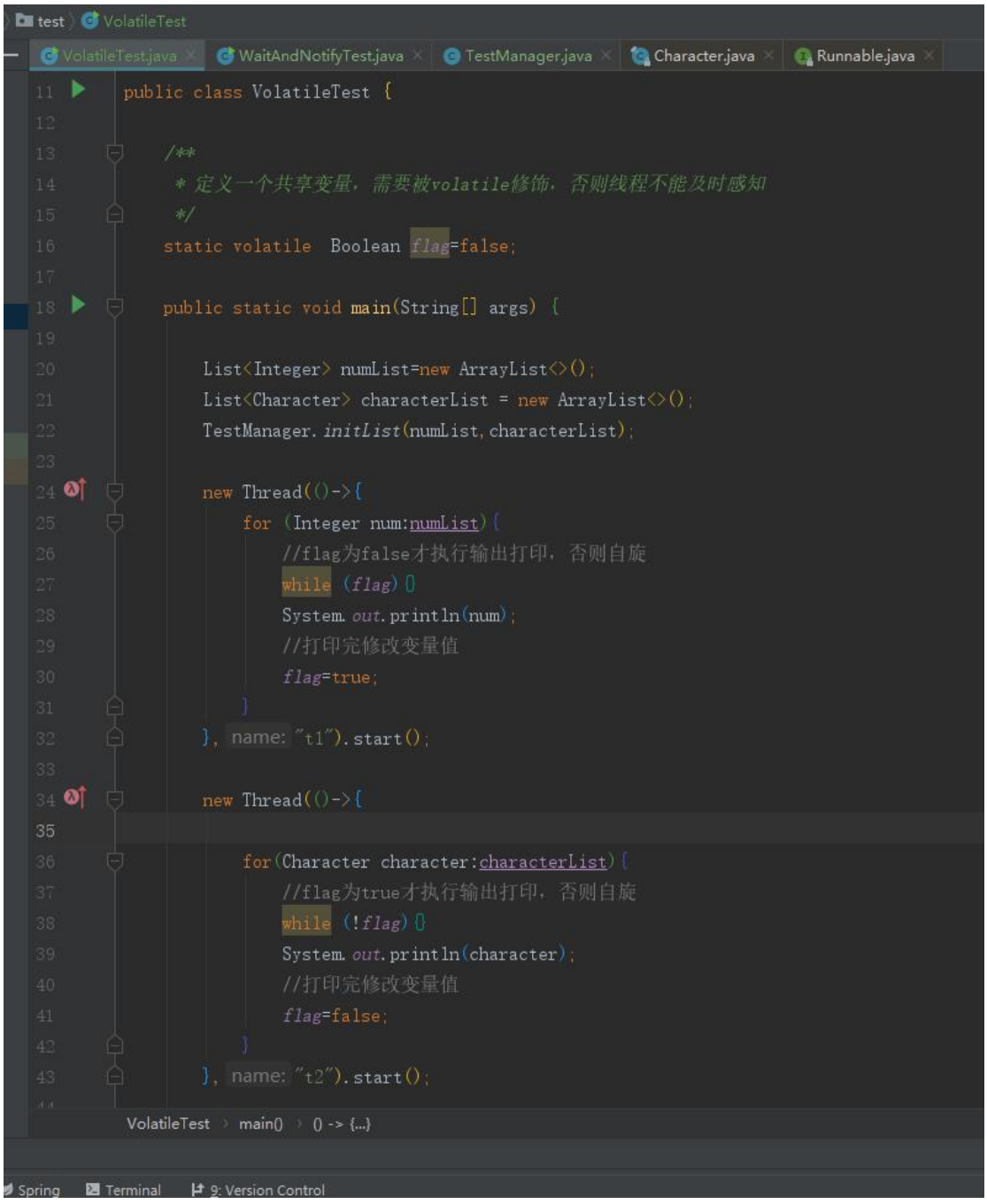

**方法一:使用 Volatile 关键字**

基于 volatile 关键字来实现线程间相互通信是使用共享内存的思想,大致意思就是多个线程同时监听一个被 volatile 关键字修饰的变量，当这个变量发生变化的时候 ，线程能够感知并执行相应的业务,避免出现**脏读**的现象，这也是最简单的一种实现方式。

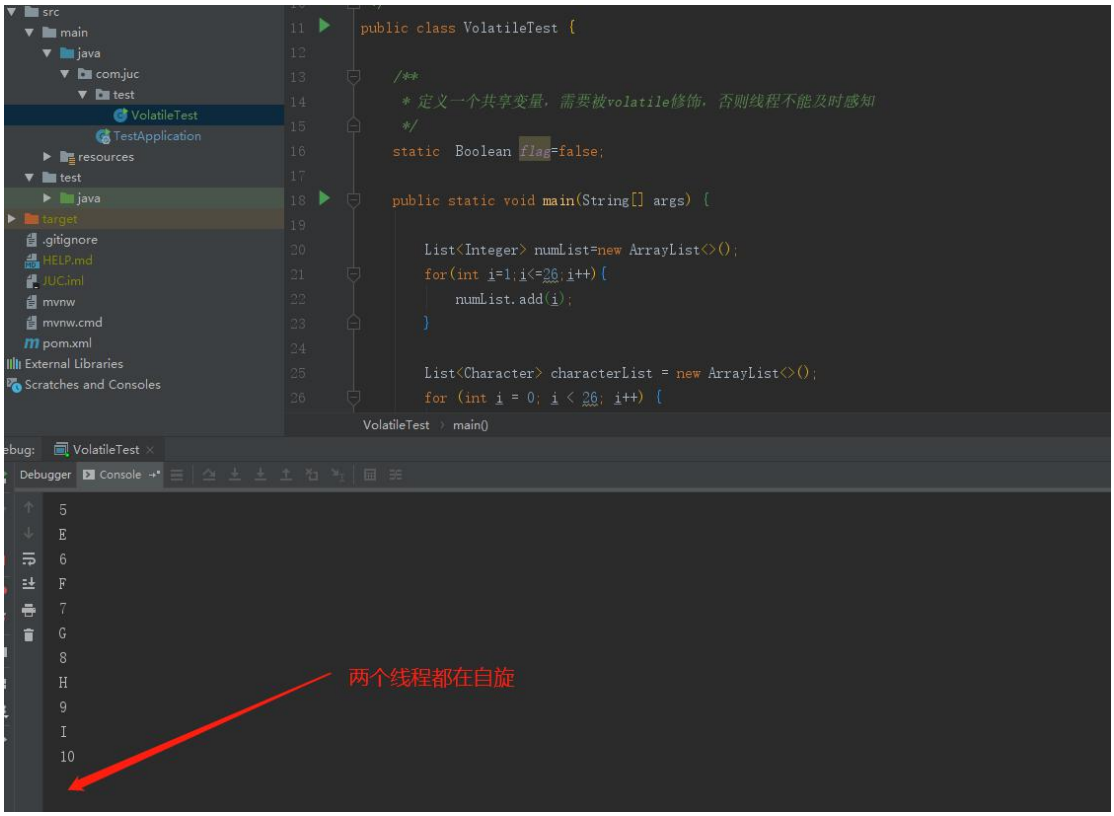
思考：为什么会出现脏读？

原因是因为 JAVA 内存模型规定所有的变量都是存在主内存中的，每个线程都有自己的工作内存（私有内存空间），线程对变量的所有操作都必须在自己工作内存中进行，而不能直接对主内存进行操作，且每个线程不能不能访问其他线程的工作内存。最重要的是，变量的值何时从线程的工作内存写回主内存是无法确定的。

代码实现：




思考：若不使用 Volatile 关键字修饰会出现什么情况？




**方法二:使用 Object 类的 wait()和 notify() 方法**

Object 类提供了线程间通信的**同步**方法:wait()、notify()、notifyAll(),他们是多线程通信的基础，为实现多线程协作提供了保证。

**需要注意的是：wait 和 notify 必须配合 synchronized 一起使用**

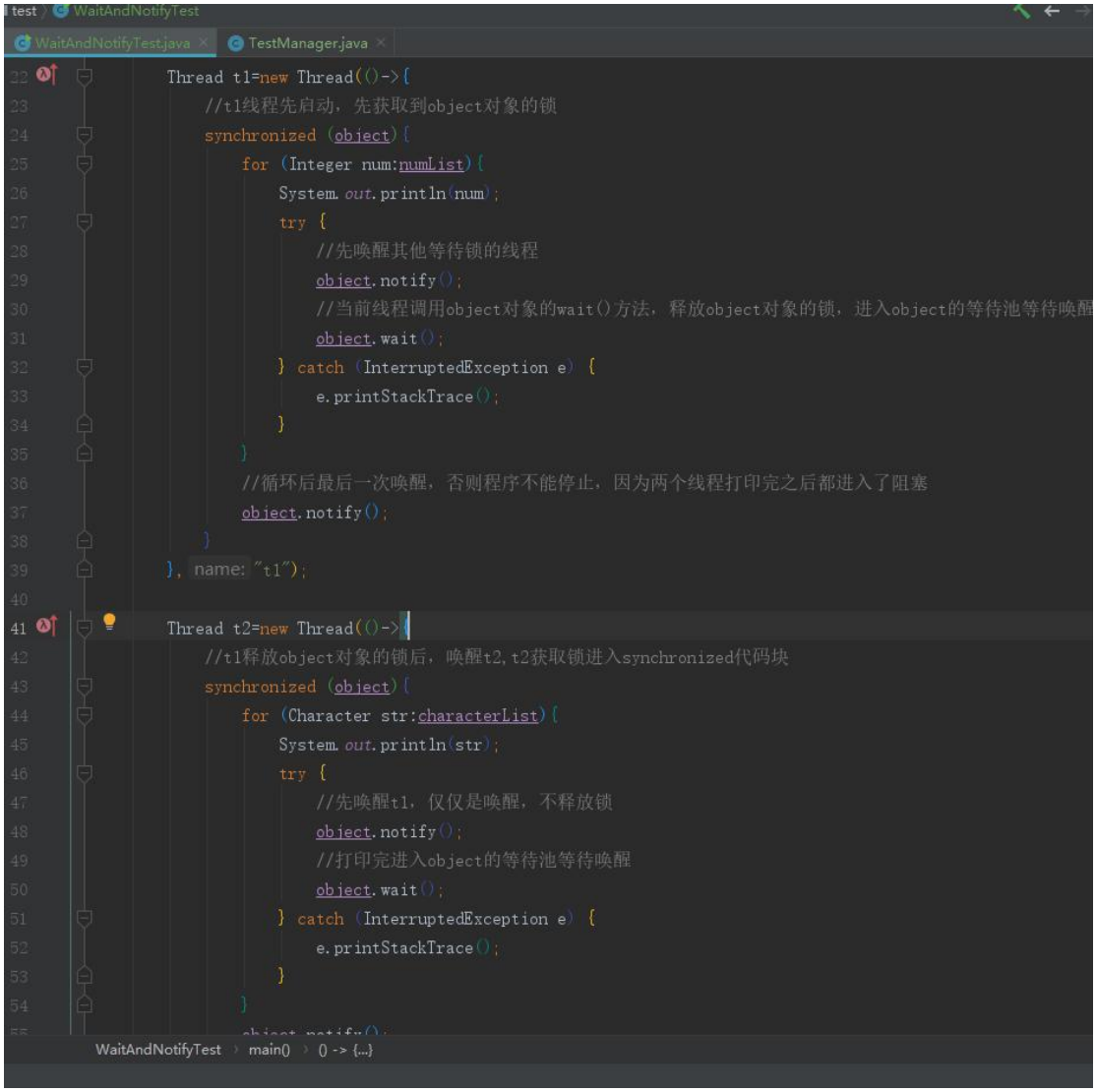
**原因：**

经查阅网上相关资料大致总结为：wait 和 notify 是针对已经获取了 object 锁的对象来进行操作的，object.wait()、object.notify()必须在 synchronize(object){...}语句块中，可以简单的理解为这是语法要求，否则会跑出 IllegalMonitorStateException 异常

拓展：object 锁放在哪呢？

object 锁是存在这个 object 对象的对象头里,对象头主要包括两部分数据：Mark Word（标记字段）和 Class Point（类型指针），其中 Mark Word 就存储了该对象的锁标志位，还存储对象的 hashcode 及分代年龄；（自行拓展）

代码实现：




**方法三:使用 ReentrantLock 结合 Condition**

ReetrantLock 是 JDK1.5 新增的同样用来实现线程之间的同步互斥，使用上与 Synchronize 类似，Synchronize 结合 Object 的 wait()和 notify()实现线程间通信，而 ReetrantLock 则可以借助 Condition 对象的 await()和 signal()方法实现同样的效果;

**主要相同点：**

1、都是独占锁，同一时间内只能有一个线程获取锁，未获取到锁的线程等待；

2、都是可重入锁，广义上解释是可重复可递归调用的锁，在外层获取并使用锁之后，在内层仍然可以使用，且不会造成死锁（前提是同一个对象）；

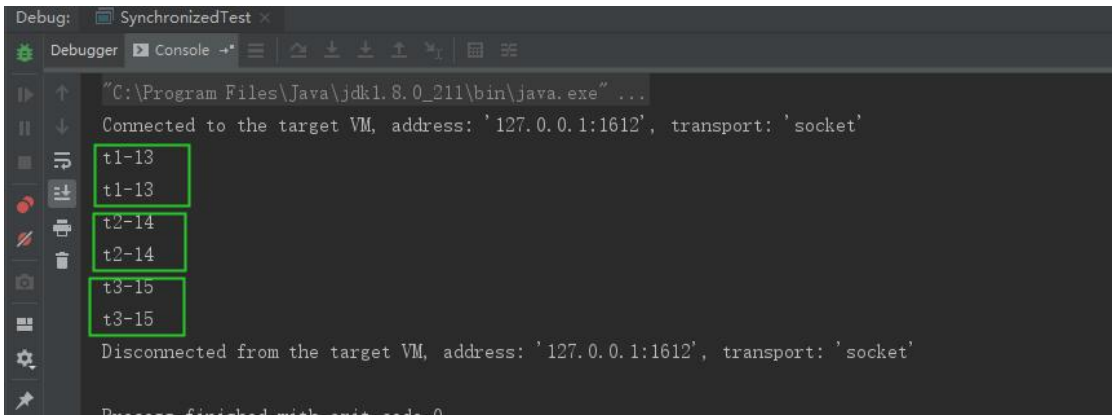
Synchronize 可重入测试:

```java
public class SynchronizedTest implements Runnable {

    public synchronized void get(){
        System.out.println(Thread.currentThread().getName()+"-"+Thread.currentThread().getId());
        //同一线程 外层函数获得锁之后 ， 内层函数仍然有获取该锁的代码，此时不受影响。可以再次获取锁而不会出现死锁
        set();
    }

    public synchronized void set(){
        System.out.println(Thread.currentThread().getName()+"-"+Thread.currentThread().getId());
    }


    @Override
    public void run() {
        get();
    }

    public static void main(String[] args) {
        SynchronizedTest test=new SynchronizedTest();
        new Thread(test, name: "t1").start();
        new Thread(test, name: "t2").start();
        new Thread(test, name: "t3").start();
    }
```

ReeTrantLock 可重入测试:

```java
    Lock lock=new ReentrantLock();

    public  void get(){
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()+"-"+Thread.currentThread().getId());
            //同一线程 外层函数获得锁之后 ， 内层函数仍然有获取该锁的代码，此时不受影响。可以再次获取锁而不会出现死锁
            set();
        }finally {
            lock.unlock();
        }
    }

    public  void set(){
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()+"-"+Thread.currentThread().getId());
        }finally {
            lock.unlock();
        }
    }

    @Override
    public void run() {
        get();
    }

    public static void main(String[] args) {
        ReentrantLockTest test=new ReentrantLockTest();
        new Thread(test, name: "t1").start();
        new Thread(test, name: "t2").start();
        new Thread(test, name: "t3").start();
    }
```

执行效果：同一个线程获取锁后，连续打印两次线程信息，说明可重入



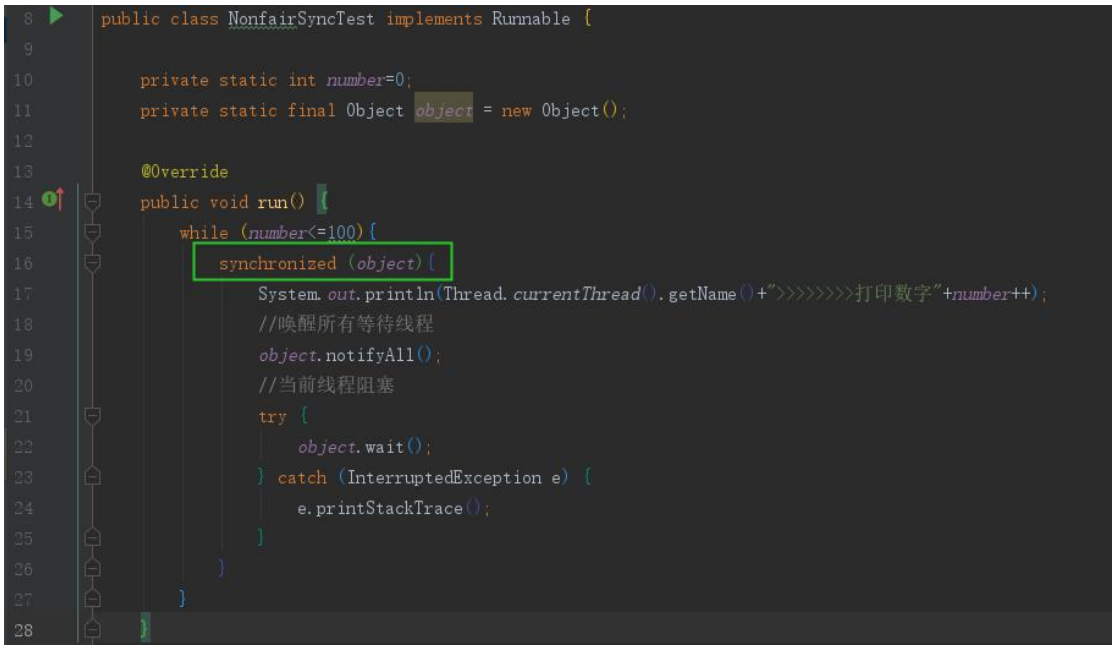

**主要不同点：**

1、Synchronize 对于加锁和释放锁都是自动的，即使加锁后发生异常系统能自动释放占用的锁，不会发生死锁现象；而 Lock 需要手动加锁和释放锁，因此使用 lock.unlock()必须放在 finally 代码块中；

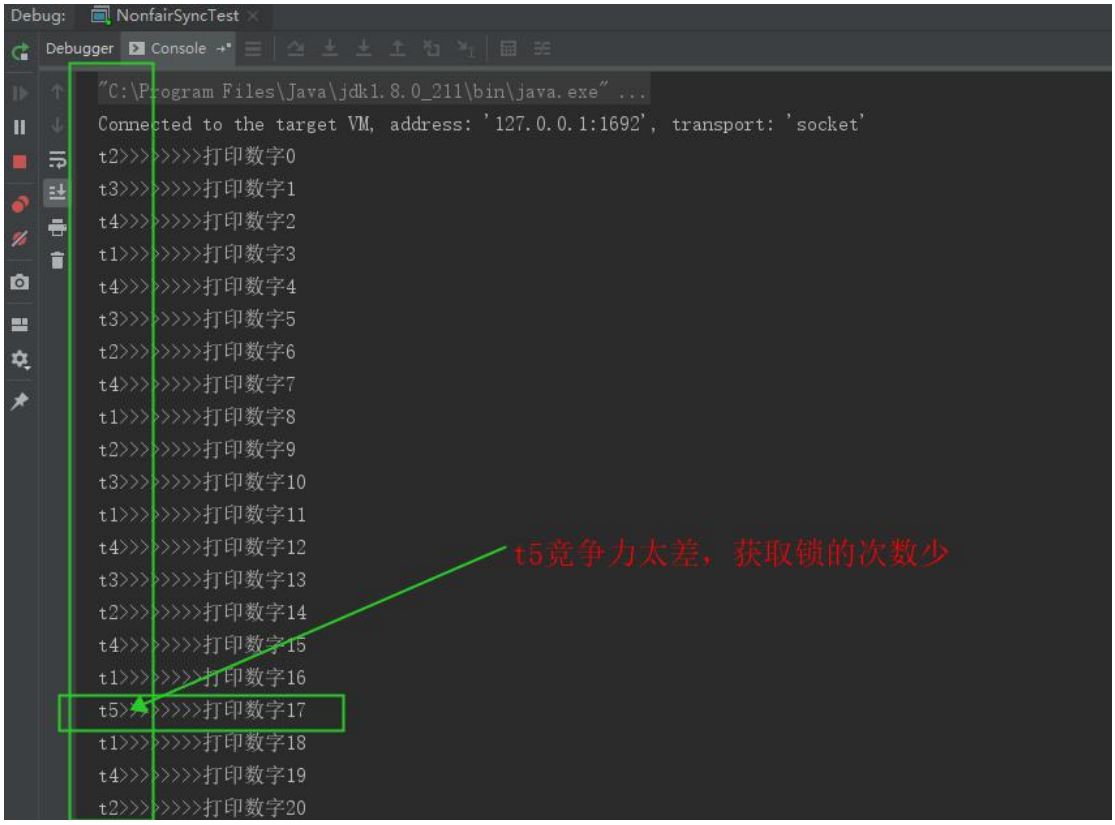
2、Synchronize 是非中断锁，lock 是中断锁；Lock 可以让等待锁的线程响应中断，而是同 Synchronize 只会让等待的线程一直等待下去，不能响应中断（阻塞）

3、Synchronize 是非公平锁，若存在多个等待获取的锁的线程，存在有的线程永远获取不到锁；而 Lock 可以在 Lock lock=new ReetrantLock（true）中指定 lock 实现公平锁机制，公平锁可以使等待时间最长的线程获取到锁；（先到先得）

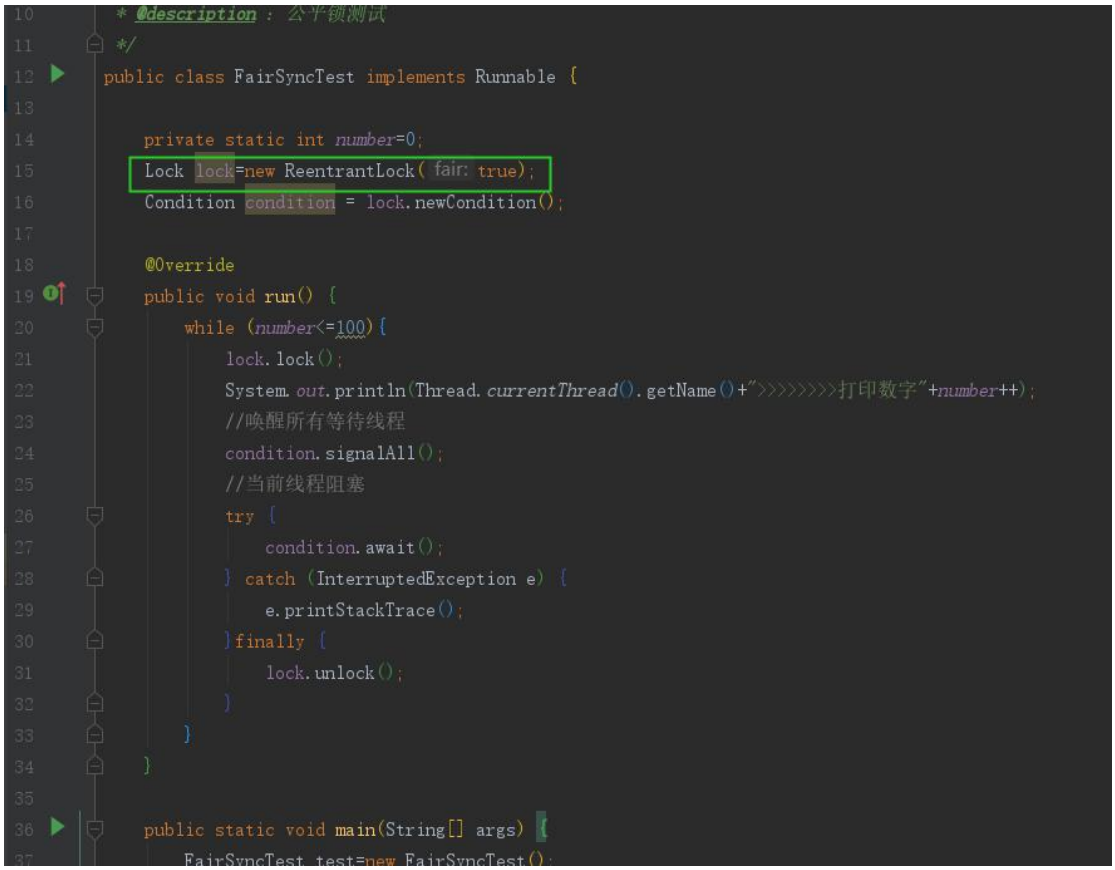
Synchronize 非公平锁测试：



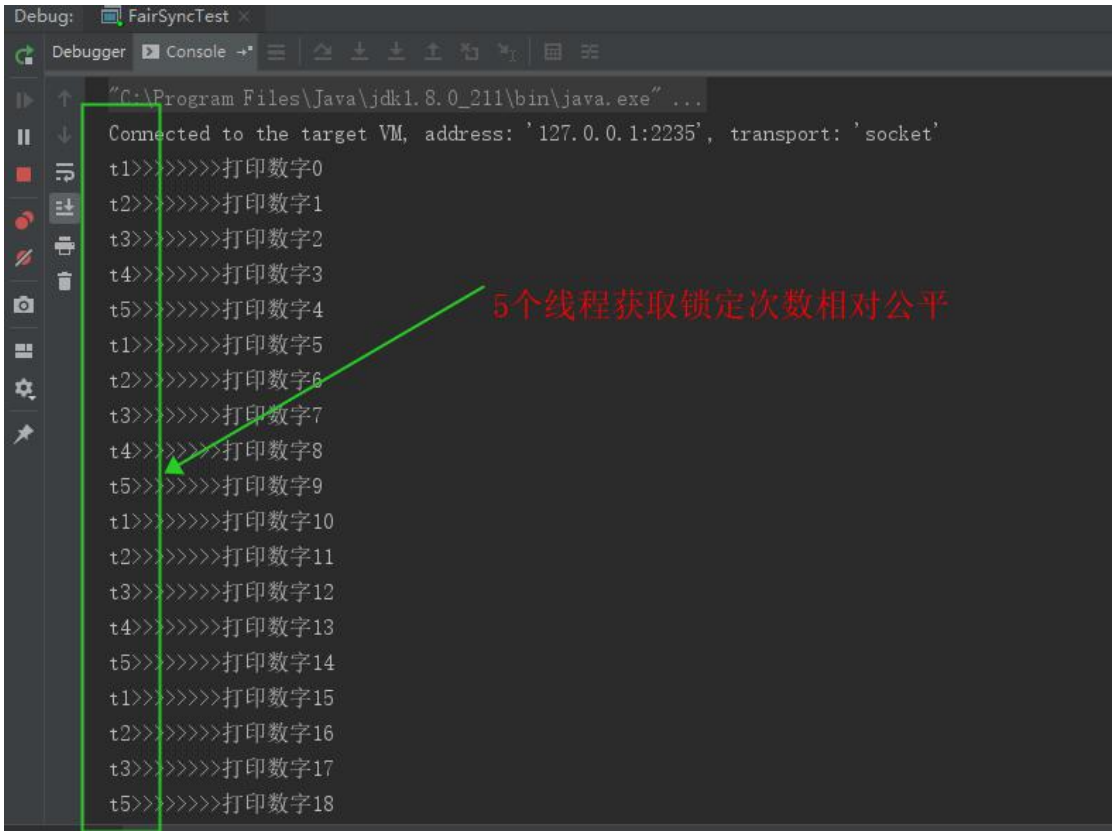
同时启动 5 个线程从 0 打印到 100，执行效果：




ReeTranLock 实现公平锁机制：



同时启动 5 个线程从 0 打印到 100，执行效果：




**性能比较：**

1、使用上 ReetrantLock 要优于 Synchronize，因为在加锁的细粒度和使用灵活度前者要高于后者；

2、JDK1.5 中，Synchronize 性能比较低，因为它是一个重量级锁，对性能最大的影响就是线程阻塞、线程挂起和唤醒线程的操作，都需要转入内核态中完成，这会给并发带来压力；

3、JDK1.6 中对 Synchronize 做了大量优化，比如加入了自适应自旋、锁消除、锁粗话、轻量级锁（自旋锁）、偏向锁，官方甚至提倡在 Synchronize 能实现需求的前提下，优先考虑 Synchronize 来进行同步；

简单介绍完 ReetrantLock 和 Synchronize 的异同点和性能比较后，看下使用 ReetrantLock 结合 Condition 如何实现：

```
SynchronizedTest.java  test\ReentrantLockTest.java  reentrant\ReentrantLockTest.java  NonfairSyncTest.java

16      public static void main(String[] args) {
17
18          List<Integer> numList=new ArrayList<>();
19          List<Character> characterList = new ArrayList<>();
20          TestManager.initList(numList,characterList);
21
22          //默认实现非公平锁机制，若需要实现公平锁传true即可
23          Lock lock=new ReentrantLock();
24
25          //condition对象用于线程的阻塞和唤醒功能,类似 object的 wait()和notify()方法
26          Condition condition = lock.newCondition();
27
28          Thread t1=new Thread(()->{
29              try {
30                  lock.lock();
31                  for (Integer num:numList){
32                      System.out.println(num);
33                      //唤醒等待线程
34                      condition.signal();
35                      //当前线程阻塞
36                      condition.await();
37                  }
38                  //最后打印完再次唤醒等待线程，否则程序无法停止
39                  condition.signal();
40
41              }catch (InterruptedException e){
42                  e.printStackTrace();
43              }finally {
44                  //必须放到finally代码块，不管是否发生异常都需手动释放锁
45                  lock.unlock();
46              }
47          }, name: "t1");
48
```

```
50          Thread t2=new Thread(()->{
51              try {
52                  lock.lock();
53                  for (Character str:characterList){
54                      System.out.println(str);
55                      //唤醒等待线程
56                      condition.signal();
57                      //当前线程阻塞
58                      condition.await();
59                  }
60                  condition.signal();
61
62              }catch (InterruptedException e){
63                  e.printStackTrace();
64              }finally {
65                  lock.unlock();
66              }
67
68          }, name: "t2");
```

**方法四:使用 JUC 工具类 CountDownLatch**

CountDownLatch 是 java 并发包下的一个辅助同步类，它能够使一个线程在等待**另一些**线程执行之后，再继续执行。

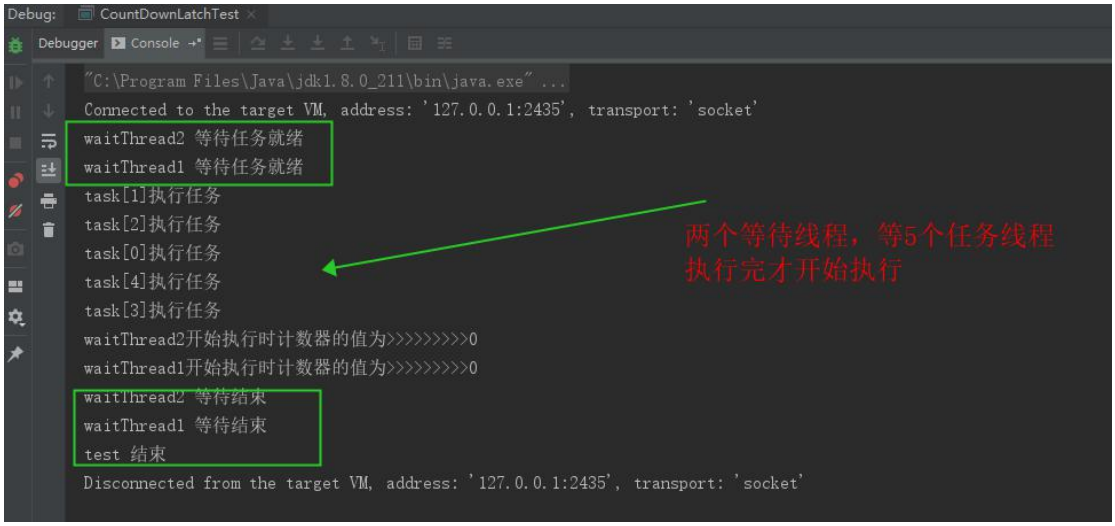
实现原理是使用一个计数器，计数器初始值为线程的数量，当每个线程执行完后，计数器的值会减 1（调用 countDownLatch.countDown()方法）。当计算器的值为 0 时，表示所有的线程都已经执行完了，然后在 CountDownLatch 上等待的线程（调用了 countDownLatch.await()方法的线程）就可以恢复执行接下来的任务;

CountDowlatch 使用示例：

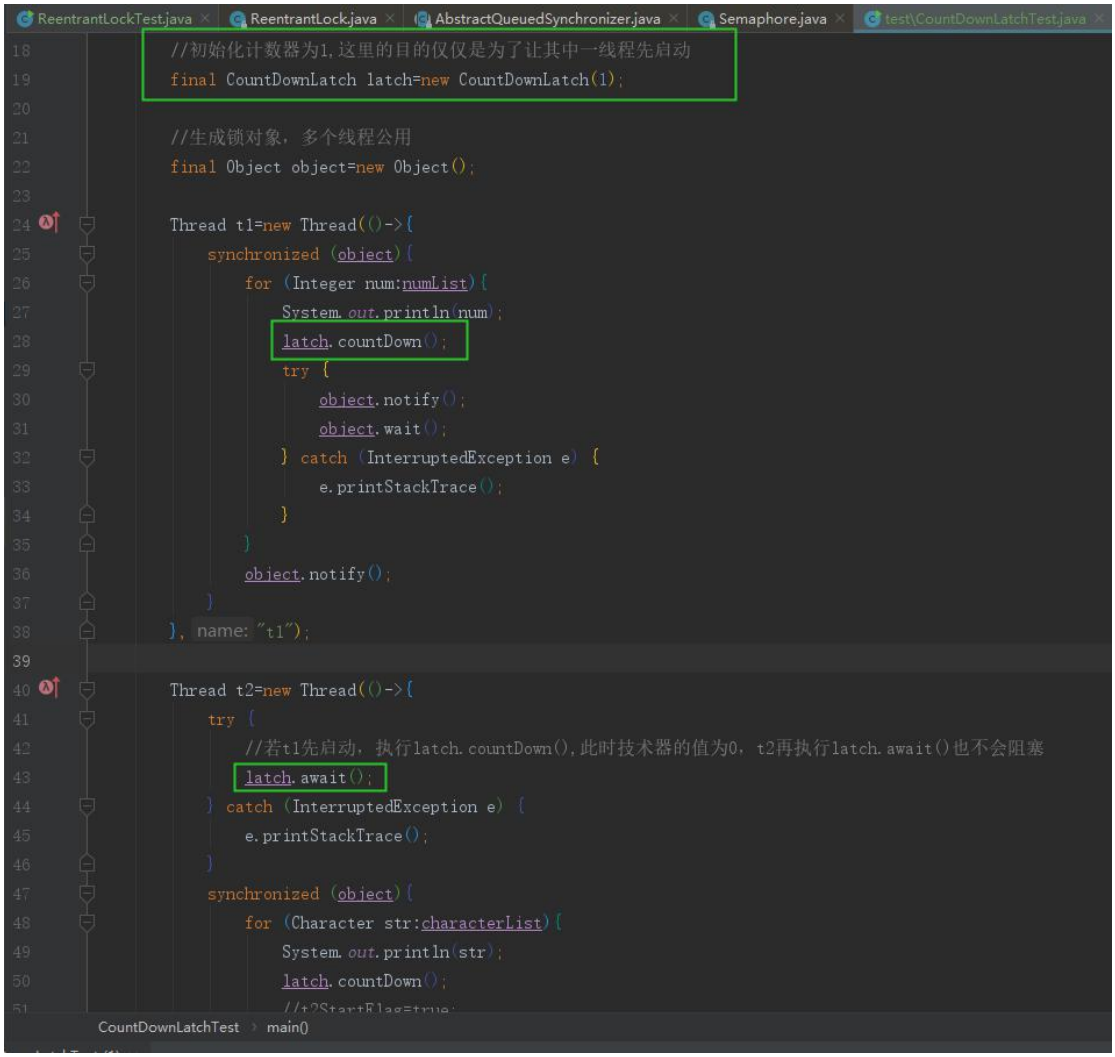
```
final CountDownLatch latch=new CountDownLatch(5);

Thread waitThread1=new Thread(()->{
    try {
        System.out.println("waitThread1 等待任务就绪");
        latch.await();
        System.out.println("waitThread1开始执行时计数器的值为>>>>>>>>>>"+latch.getCount());
        System.out.println("waitThread1 等待结束");
    }catch (InterruptedException e){
        e.printStackTrace();
    }
});

Thread waitThread2=new Thread(()->{
    try {
        System.out.println("waitThread2 等待任务就绪");
        latch.await();
        System.out.println("waitThread2开始执行时计数器的值为>>>>>>>>>>"+latch.getCount());
        System.out.println("waitThread2 等待结束");
    }catch (InterruptedException e){
        e.printStackTrace();
    }
});

waitThread1.start();
waitThread2.start();
```

```
//休眠一秒
Thread.sleep( millis: 1000);

//启动5个任务线程
for(int i=0;i<5;i++){
    final int taskId=i;
    Thread task=new Thread(()->{
        System.out.println("task["+taskId+"]执行任务");
        //执行完任务，计数器减1
        latch.countDown();
    });
    task.start();
}

//执行完5个任务线程后，计数器减为0，所有调用了latch的线程被唤醒
waitThread1.join();
waitThread2.join();
System.out.println("test 结束");
```

执行效果：




使用 CountLatch 实现题目示例（可优化）：



**方法五:基于 LockSupport 实现线程间的阻塞和唤醒**

LockSupport 是 jdk 中一个比较底层的类，用来创建锁和其他同步工具类（ReetrantLock、CountDownLatch）的基本线程阻塞原语，Java 锁和同步器框架（AQS 框架）AbstractQueuedSynchronizer 底层使用 LockSuport 类（还有一个 UnSafe 类（提供 CAS 操作））实现线程阻塞和唤醒；

LockSupport 最主要的两个方法就是 park 和 unpark，如果把线程 Thread 比喻成一辆车的话,park 就是让车停下,unpark 就是让车启动起来;这个和前面提到的 wait 和 notify 方法类似;

两者主要区别：

1、wait 和 notify 的执行顺序不能换，比如线程 A 要用 nonify 唤醒线程 B，那么线程 A 必须确保线程 B 已经调用过 wait 方法等待唤醒了，否则线程 B 可能会永远都在等待；

2、park 和 unpark 无执行顺序问题，unpark 可以优先于 park 调用；

代码实现：