【讲义】第1讲：并发编程基础知识

一、Java内存模型

- Java内存模型，即：JMM。当程序执行并行操作时，如果对数据的访问和操作不加以控制，那么必然会对程序的正确性造成破坏。因此，我们需要在深入了解并行机制的前提下，再定义一种规则，来保证多个线程间可以有效地、正确地协同工作。而JMM就是为此而生的。
- JMM的关键技术点都是围绕着多线程的原子性、可见性和有序性来创建的。所以，下面我们来一一介绍这三种特性。

1.1> 原子性（Atomicity）

- 原子性

> 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程所干扰。

- 比如一个int a，线程A对其赋值1，线程B对其赋值2，无论什么情况，a的值要么是1，要么是2；不会被线程A或线程B干扰。但是，如果是在**32位操作系统**中，操作64位的long类型数据的时候，就无法保证原子性了。因为赋值操作需要执行2次32位的操作，而在多线程的情况下，可能会出现“意想不到”的最终结果。如下所示：

```java
/**
 * 对32位操作系统进行long操作的时候，会因为读写不是原子性的，而产生问题
 */
public class MultiThreadLong {
    public static long t = 0;
    public static void main(String[] args) {
        new Thread(new WriteThread(111L)).start();
        new Thread(new WriteThread(-999L)).start();
        new Thread(new WriteThread(333L)).start();
        new Thread(new WriteThread(-444L)).start();
        new Thread(new ReadThread()).start();
    }
}

/** 赋值线程 */
class WriteThread implements Runnable {
    private long to;
    public WriteThread(long to) {
        this.to = to;
    }
    @Override
    public void run() {
        while (true) {
            MultiThreadLong.t = to;
            Thread.yield();
        }
    }
}

/** 读取线程 */
class ReadThread implements Runnable {
    @Override
    public void run() {
        while (true) {
            long tmp = MultiThreadLong.t;
            // 由于long是64位的，在32位操作系统中，不是原子性操作，所以，if判断有可能是true
            if (tmp != 111L && tmp != -999L && tmp != 333L && tmp != -444L) {
                System.out.println(tmp);
            }
            Thread.yield();
        }
    }
}
```

【解释】

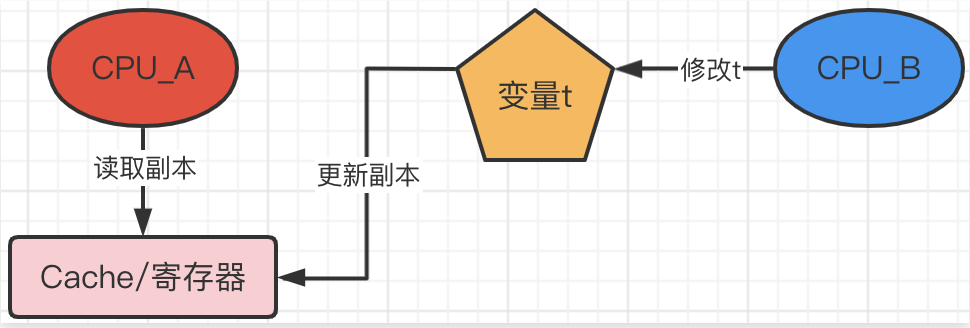
> 由于在32位操作系统中，对long赋值是要执行两步的，所以，在并发赋值时，就有可能最终long的赋值结果“让人很意外”，即：不是111L、-999L、333L和-444L中的任意一个。（例如：long的高32位由线程A赋值了，低32位由线程B赋值了）

1.2> 可见性（Visibility）

- 可见性

> 是指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道这个修改。

- 如果在CPU_A和CPU_B上各运行了一个线程，它们共享变量t，由于编译器优化或者硬件优化的缘故， 在CPU_A上的线程将变量t进行了优化，将其缓存在cache中或者寄存器里。这种情况下，如果在CPU_B上的某个线程修改了变量t的实际值，那么CPU_A上的线程可能并无法意识到这个改动，依然会读取cache中或者寄存器里的数据。



- 可见性问题是一个综合性问题。除了上述提到的缓存优化或者硬件优化（有些内存读写可能不会立即触发，而会先进入一个硬件队列等待）会导致可见性问题外，**指令重排**以及**编译器的优化**，都有可能导致一个线程的修改不会立即被其他线程察觉。

1.3> 有序性（Ordering）

- 因为指令流水线的存在，CPU才能真正高效的执行。但是，流水线总是害怕被中断的。流水线满载时，性能确实相当不错，但是一旦中断，所有的硬件设备都会进入一个停顿期，再次满载又需要几个周期，因此，性能损失会比较大。所以，我们必须要想办法尽量不让流水线中断。

1.3.1> 指令重排

- A=B+C的执行过程

A=B+C

把变量B值加载到R1寄存器中	IF	ID	EX	MEM	WB				
把变量C值加载到R2寄存器中		IF	ID	EX	MEM	WB			
R1+R2，并存放到R3寄存器中			IF	ID	X	EX	MEM	WB	
将R3寄存器的值保存到变量A中				IF	X	ID	EX	MEM	WB

IF：取指
ID：译码和取寄存器操作数
EX：执行或者有效地址计算
MEM：存储器访问
WB：写回

- 重排前指令执行过程（红叉表示停顿）

a=b+c	把变量b值加载到Rb寄存器中	IF	ID	EX	MEM	WB								
	把变量c值加载到Rc寄存器中		IF	ID	EX	MEM	WB							
	Rb+Rc，并存放到Ra寄存器中			IF	ID	X	EX	MEM	WB					
	将Ra寄存器的值保存到变量a中				IF	X	ID	EX	MEM	WB				
d=e-f	把变量e值加载到Re寄存器中					X	IF	ID	EX	MEM	WB			
	把变量f值加载到Rf寄存器中							IF	ID	EX	MEM	WB		
	Re-Rf，并存放到Rd寄存器中								IF	ID	X	EX	MEM	WB
	将Rd寄存器的值保存到变量d中									IF	X	ID	EX	MEM	WB

- 重排后的指令（已经没有停顿了）

	把变量b值加载到Rb寄存器中	IF	ID	EX	MEM	WB								
	把变量c值加载到Rc寄存器中		IF	ID	EX	MEM	WB							
	把变量e值加载到Re寄存器中			IF	ID	EX	MEM	WB						
a=b+c	Rb+Rc，并存放到Ra寄存器中				IF	ID	EX	MEM	WB					
	把变量f值加载到Rf寄存器中					IF	ID	EX	MEM	WB				
	将Ra寄存器的值保存到变量a中						IF	ID	EX	MEM	WB			
d=e-f	Re-Rf，并存放到Rd寄存器中							IF	ID	EX	MEM	WB		
	将Rd寄存器的值保存到变量d中								IF	ID	EX	MEM	WB	

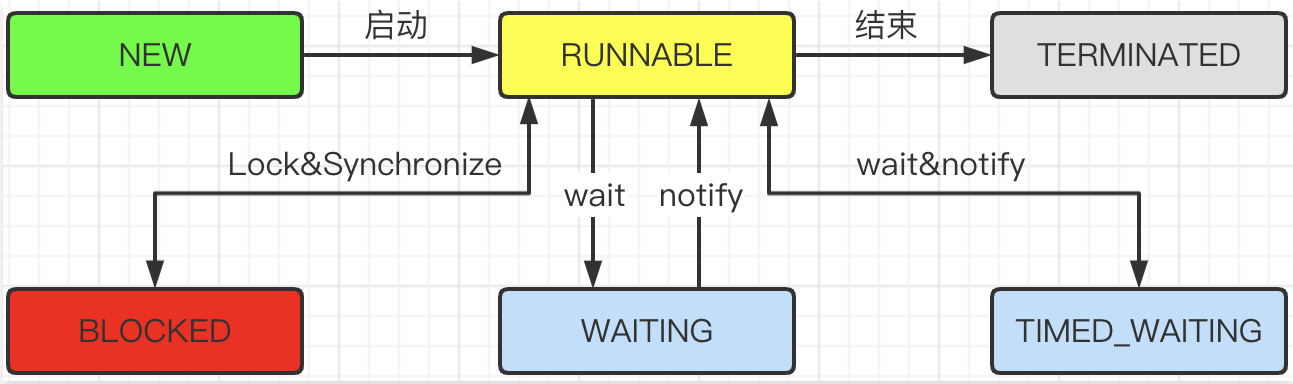
1.3.2> Happen-Before规则

- 程序顺序原则：一个线程内保证语义的串行性。
- volatile规则：volatile变量的写，先发生于读，这保证了volatile变量的可见性。
- 锁规则：解锁（unlock）必然发生在随后的加锁（lock）前。
- 传递性：A先于B，B先于C，那么A必然先于C。
- 线程的start()方法先于它的每一个动作。
- 线程的所有操作先于线程的终结Thread.join()。
- 线程的中断interrupt()先于被中断线程的代码。
- 对象的构造函数执、结束先于finalize()方法。

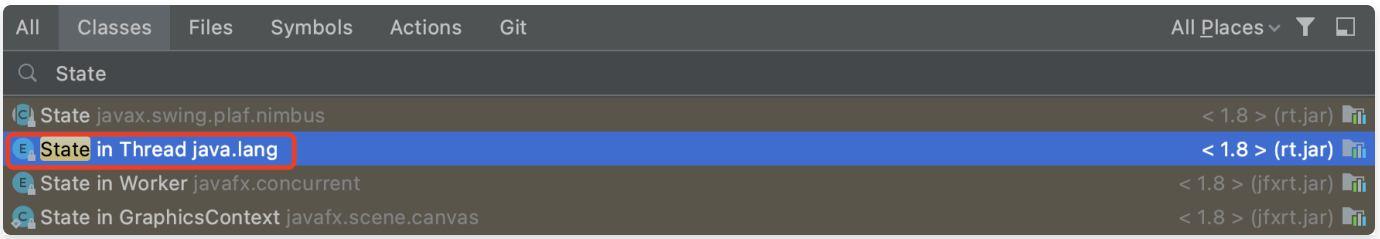
二、多线程基本操作

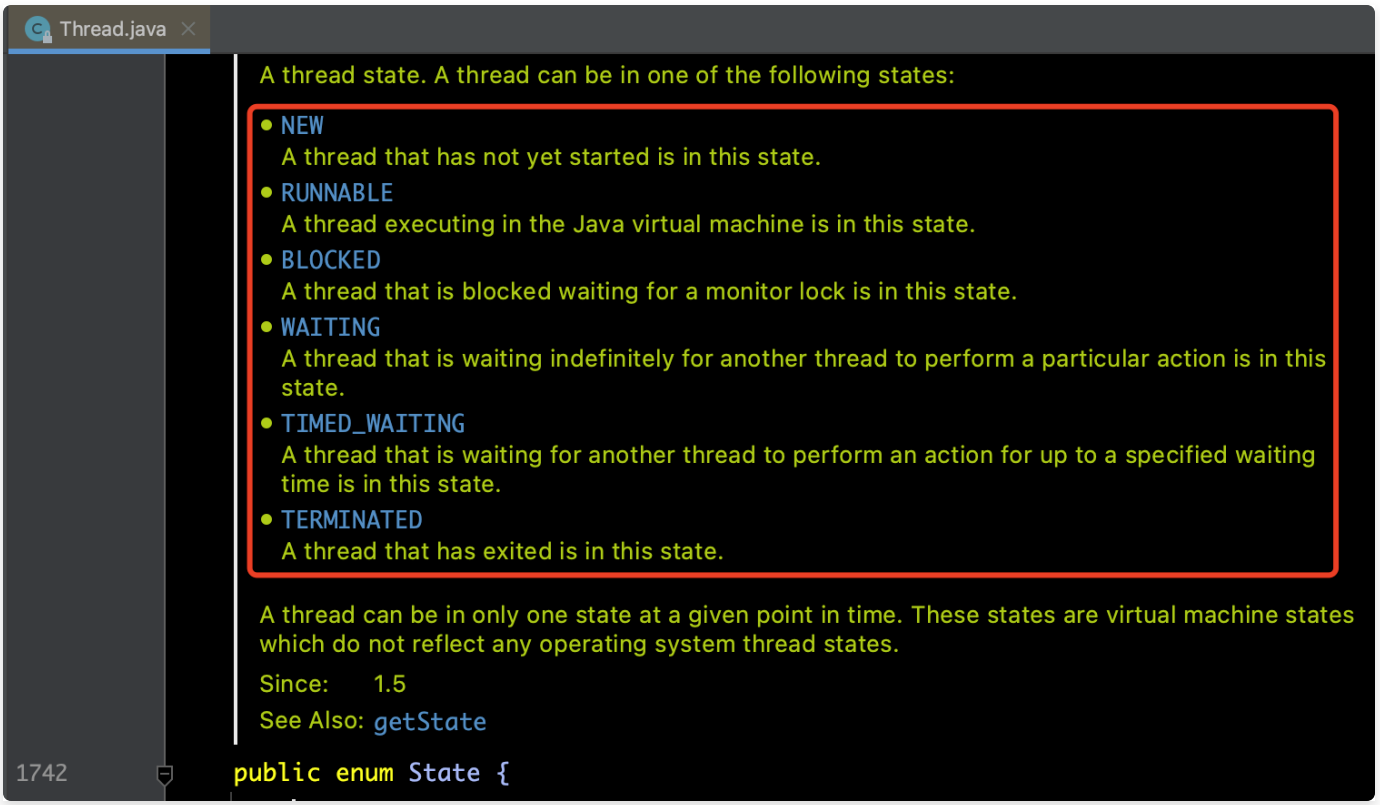
2.1> 线程状态

- 线程状态图



- 线程的所有状态都在Thread中的State枚举中定义的。如下所示：





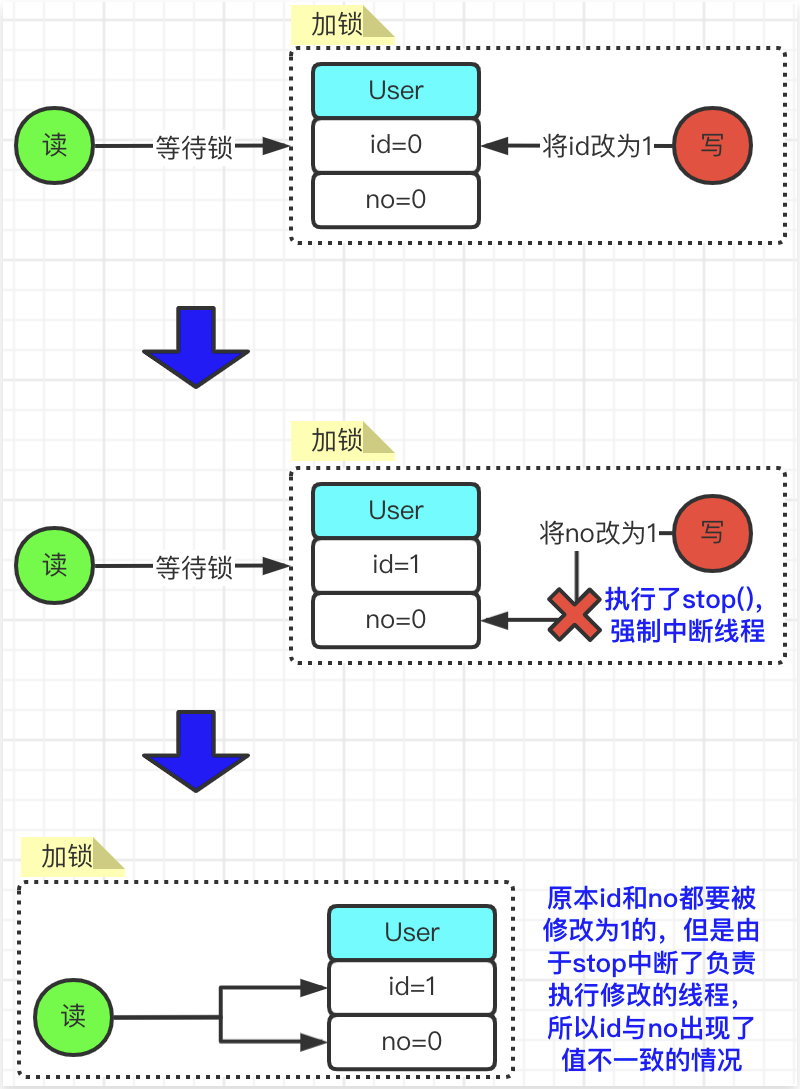
【解释】

> - NEW：表示刚刚创建的线程，这种线程还没开始执行。
> - RUNNABLE：当调用start()方法时，处于该状态，表示线程所需的一切资源都已经准备好了。
> - BLOCKED：如果线程在执行过程中遇到了锁，就会进入该状态。
> - WAITING：处于无时间限制的等待状态。
> - TIMED_WAITING：处于有限的等待状态。

> - TERMINATED：当线程执行完毕，就进入结束状态。
>
> 【注意】从NEW状态出发后，线程不能再回到NEW状态，同理，处于TERMINATED的线程也不能再回到RUNNABLE状态。

2.2> stop（被废弃）

- JDK提供了stop方法用来关闭线程，但是该方法由于太过于暴力了，强行把执行到一半的线程终止，所以可能会引起一些数据不一致的问题。所以stop方法被标记为废弃方法。如下所示：



- 代码实现

StopThread.java

```java
public class StopThread {
    public static User user = new User();
    @SneakyThrows
    public static void main(String[] args) {
        new ReadObjectThread().start(); // 开启读线程
        while (true) { // 开启写线程
            Thread changeObjectThread = new ChangedObjectThread();
            changeObjectThread.start();
            Thread.sleep(150);
            changeObjectThread.stop(); // 执行stop，强制终止changeObjectThread线程，并且没有任何错误中断信息输出
        }
    }
    /** 改变变量值的线程 */
    public static class ChangedObjectThread extends Thread {
        @SneakyThrows
        @Override
        public void run() {
            while (true) {
                synchronized(user) {
                    int v = (int) (System.currentTimeMillis()/1000);
                    user.setId(v);
                    Thread.sleep(100); // 强制要求沉睡100毫秒
                    user.setNo(v);
                }
                Thread.yield();
            }
        }
    }
    /** 读取变量值的线程 */
    public static class ReadObjectThread extends Thread {
        @Override
        public void run() {
            while (true) {
                synchronized(user) {
                    if (user.getId() != user.getNo()) { // 只有当user对象的id和no不同的时候，才输出
                        System.out.println(user.toString());
                    }
                }
                Thread.yield();
            }
        }
    }
}

@Data
@ToString
class User {
    private int id = 0;
    private int no = 0;
}
```

加大在此处被 stop() 的概率，这样 id 和 no 值就会出现不同！

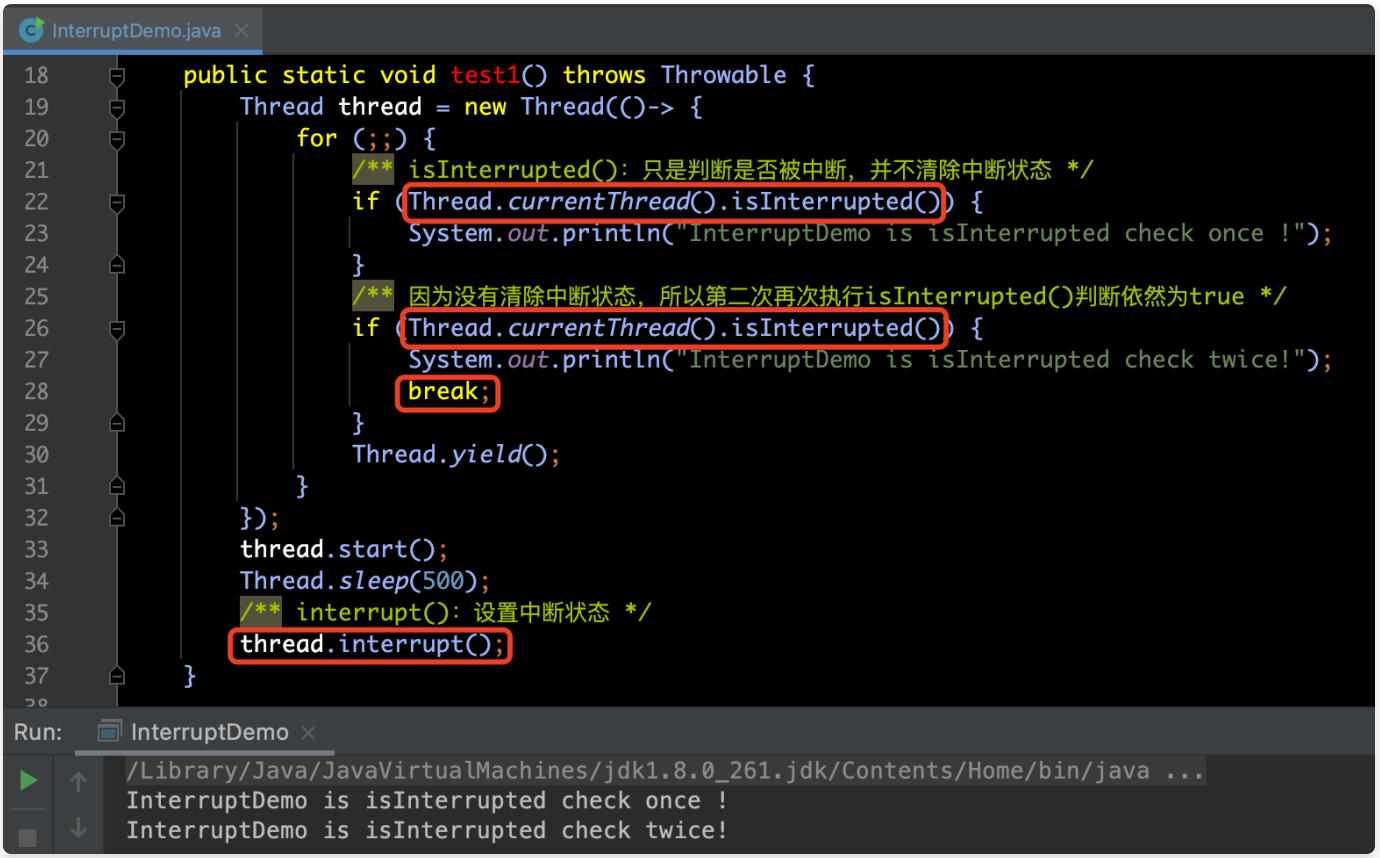
2.3> interrupt&isInterrupt&interrupted

- 线程中断并不会使线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出啦！至于目标线程接到通知后如何处理，则完全**由目标线程自行决定**。
- “完全由目标线程自行决定”这一点非常重要，如果中断后，线程立即无条件退出，那么又会遇到跟stop()方法一样的老问题了。
- JDK中关于线程中断一共提供了3种方法：

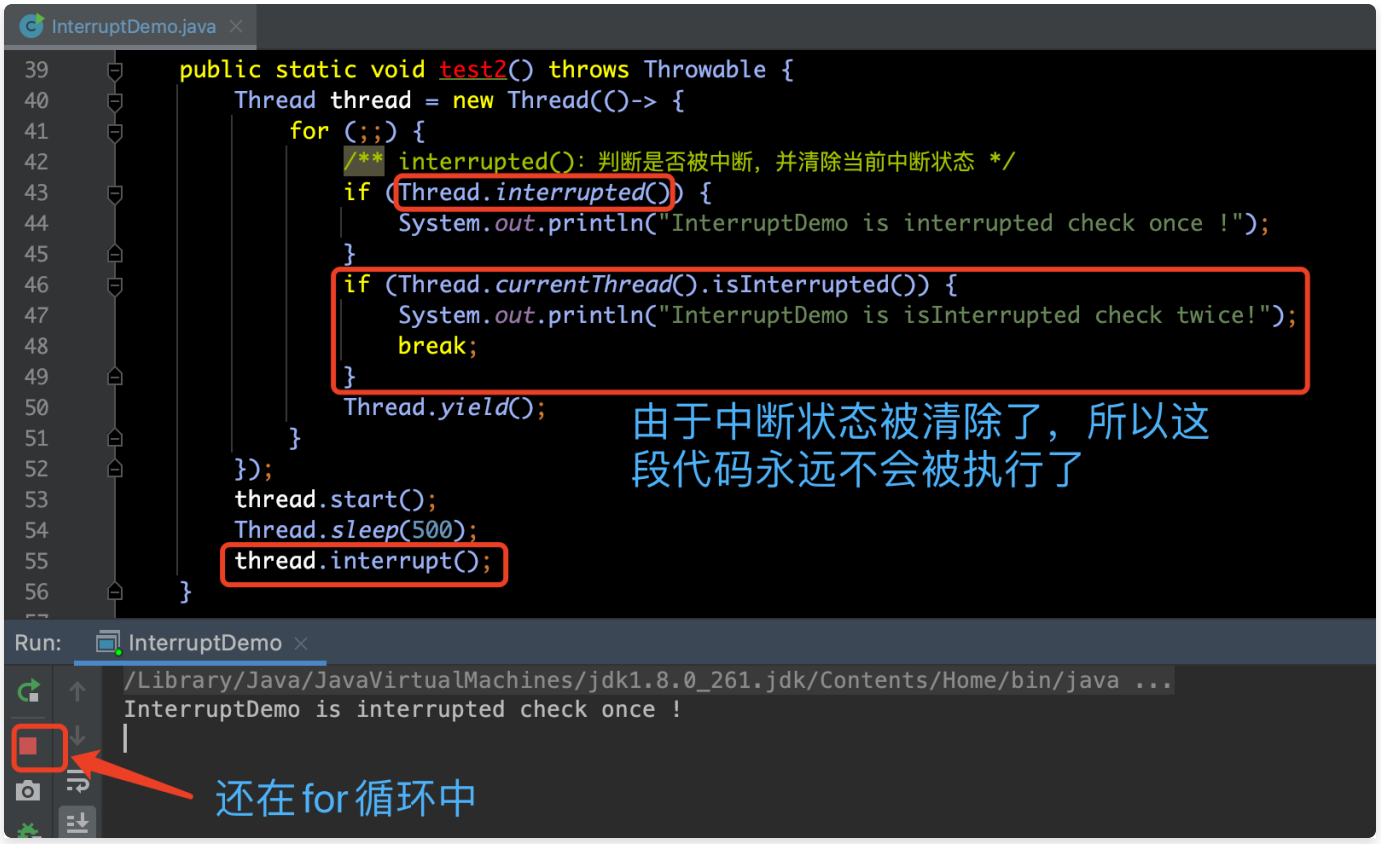
> - interrupt()　　中断线程，**添加中断状态**

- isInterrupted()　判断线程是否被中断
- interrupted()　判断线程是否被中断，并**清除当前中断状态**

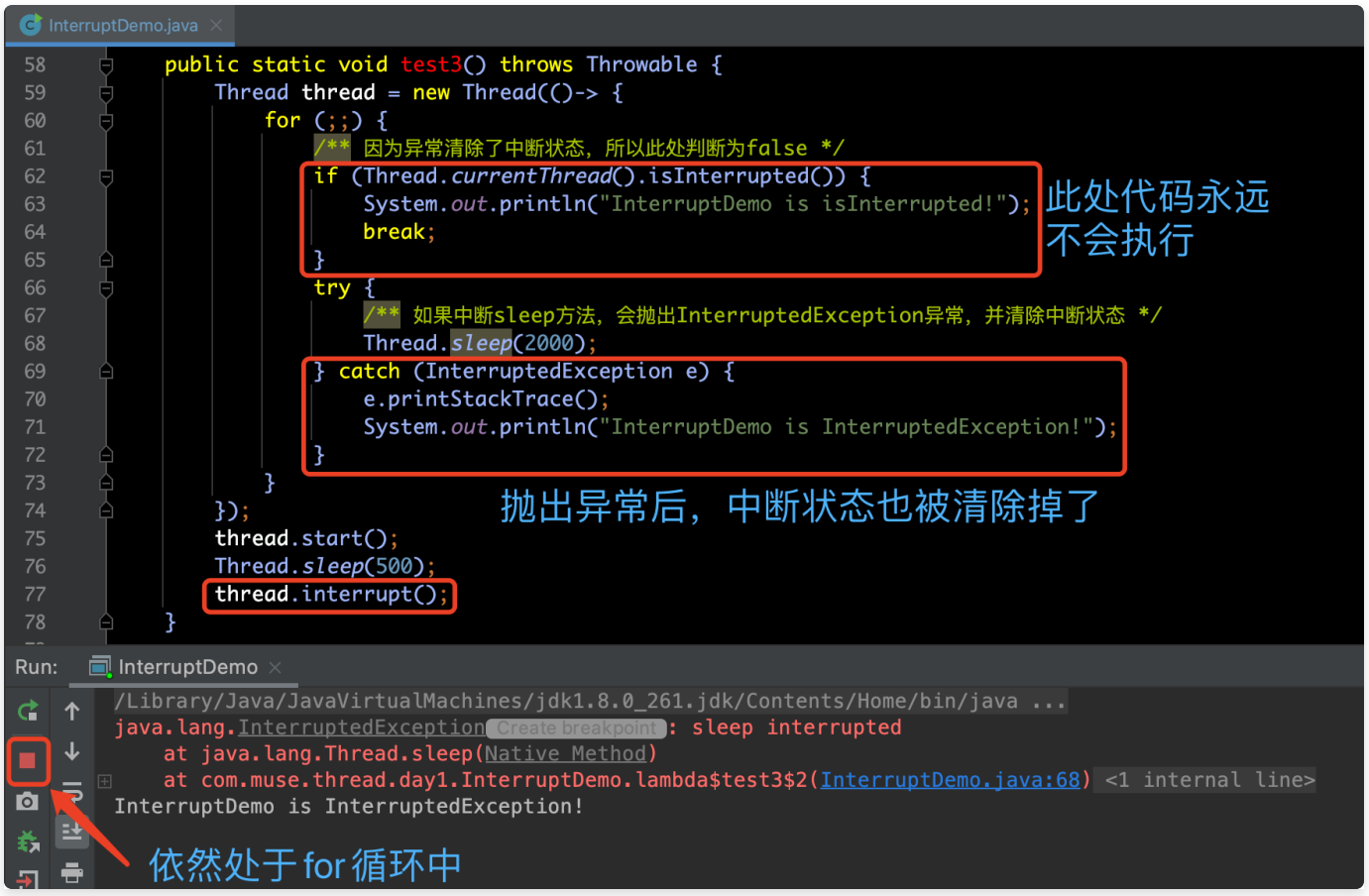
- 实验一：测试isInterrupted()只判断线程是否被中断，不会清除中断状态



- 实验二：测试interrupted()会清除掉当前中断状态



- 实验三：如果中断sleep方法，会抛出InterruptedException异常，并清除中断状态



- 实验四：修复由于异常后，线程中断状态被清除掉导致无限循环的问题

```java
public static void test4() throws Throwable {
    Thread thread = new Thread(()-> {
        for (;;) {
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("InterruptDemo is isInterrupted!");
                break;
            }
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                System.out.println("InterruptDemo is InterruptedException!");
                /** 因为中断标识被异常清除了，所以再次设置中断标识，这样第二次循环的时候，isInterrupted判断就可以break */
                Thread.currentThread().interrupt();
            }
        }
    });
    thread.start();
    Thread.sleep(500);
    thread.interrupt();
}
```

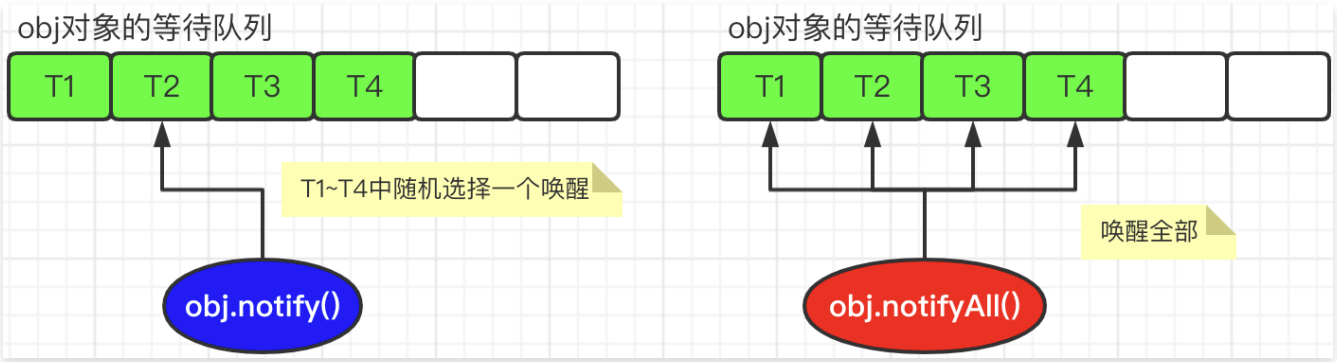
由于异常中添加了中断状态，所以该段代码会被执行

由于异常清除掉了中断状态，所以此处再次调用interrupt方法，再次给线程添加中断状态

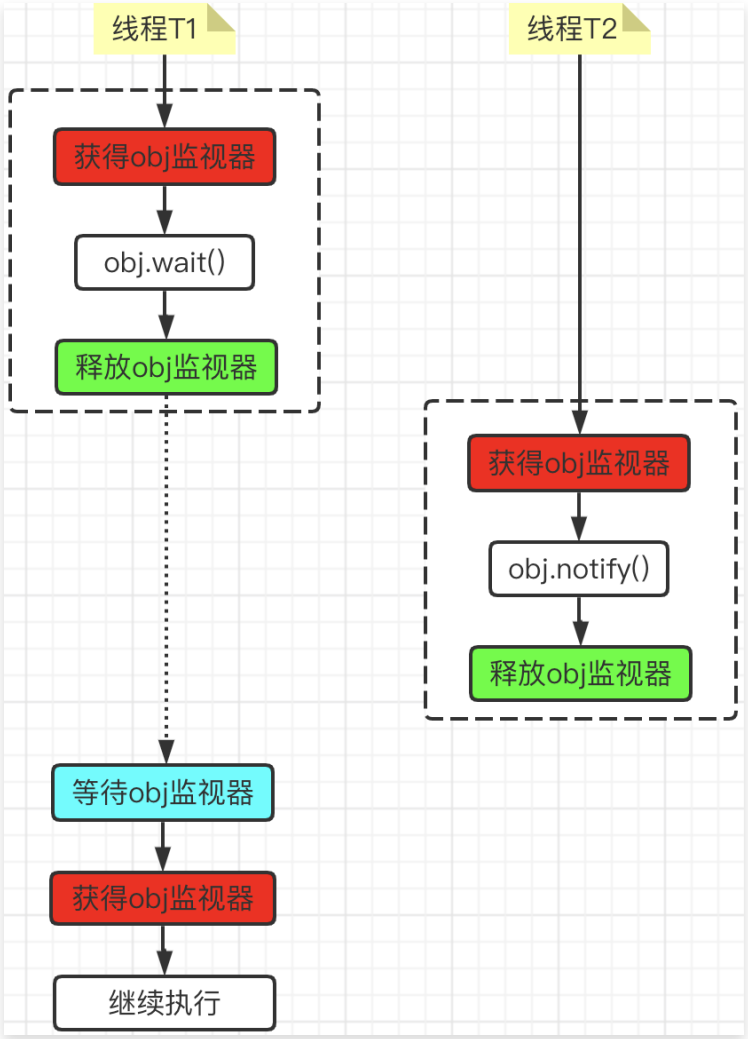
```
Run: InterruptDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
java.lang.InterruptedException Create breakpoint : sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.muse.thread.day1.InterruptDemo.lambda$test4$3(InterruptDemo.java:88) <1 internal line>
InterruptDemo is InterruptedException!
InterruptDemo is isInterrupted!

Process finished with exit code 0
```

2.4> wait&notify

- 这两个方法是**Object类**提供的方法，也就是说，任何对象都可以调用这两个方法。用于支持多线程之间的协作操作。
- 线程A调用了obj.wait()方法，那么线程A就会停止继续执行，而转为等待状态。*那么等待到何时才能结束呢？* 即：线程A会一直等到其他线程调用了obj.notify()方法为止。
- 如果一个线程调用了object.wait()，那么它就会**进入object对象的等待队列**。这个等待队列中，可能会有多个线程，因为系统运行多个线程同时等待某一个对象。当object.notify()被调用时，它就会从这个等待队列中，随机选择一个线程，并将其唤醒。这种选择是不公平的，是**完全随机**的。
- notify()和notifyAll()唤醒等待的线程的区别



- Object的wait()方法并不是可以随便调用的。它**必须包含在对应的synchronized语句中**，无论是wait()或notify()都需要首先获得目标对象的一个监视器。



- 演示操作

```java
public class WaitNotifyDemo {
    final static Object obj = new Object();

    public static void main(String[] args) {
        new T1().start();
        new T2().start();
    }

    public static class T1 extends Thread {
        @Override
        public void run() {
            synchronized(obj) {
                System.out.println(System.currentTimeMillis() + ":T1 start!");
                try {
                    /** Step1: 释放obj对象锁，进入obj对象的wait队列 */
                    System.out.println(System.currentTimeMillis() + ":T1 wait for object!");
                    obj.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                /** Step3: T2执行完毕，T1需要获得obj锁，才可以继续执行 */
                System.out.println(System.currentTimeMillis() + ":T1 end!");
            }
        }
    }

    public static class T2 extends Thread {
        @Override
        public void run() {
            // 由于Step1释放obj的锁，所以T2可以获得锁，然后运行
            synchronized(obj) {
                System.out.println(System.currentTimeMillis() + ":T2 start! notify one thread");
                /** Step2: 唤醒T1，但是由于T2没有执行完毕而释放obj的锁，所以T1等待中 */
                obj.notify();
                System.out.println(System.currentTimeMillis() + ":T2 notify end!");
                try {
                    Thread.sleep(2000);
                    System.out.println(System.currentTimeMillis() + ":T2 sleep 2000ms end!");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
Run:  WaitNotifyDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1646892620803:T1 start!
1646892620803:T1 wait for object!
1646892620804:T2 start! notify one thread
1646892620804:T2 notify end!
1646892622807:T2 sleep 2000ms end!
1646892622808:T1 end!

Process finished with exit code 0
```

- 如果不在T2中执行obj.notify();则T1一直处于阻塞状态，输入如下所示：

```
    public static class T2 extends Thread {
        @Override
        public void run() {
            // 由于Step1释放obj的锁，所以T2可以获得锁，然后运行
            synchronized(obj) {
                System.out.println(System.currentTimeMillis() + ":T2 start! notify one thread");
                /** Step2: 唤醒T1，但是由于T2没有执行完毕而释放obj的锁，所以T1等待中 */
                //obj.notify();
                System.out.println(System.currentTimeMillis() + ":T2 notify end!");
                try {
                    Thread.sleep(2000);
                    System.out.println(System.currentTimeMillis() + ":T2 sleep 2000ms end!");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
Run: WaitNotifyDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1647935764635:T1 start!
1647935764636:T1 wait for object!
1647935764637:T2 start! notify one thread
1647935764637:T2 notify end!
1647935766641:T2 sleep 2000ms end!
```

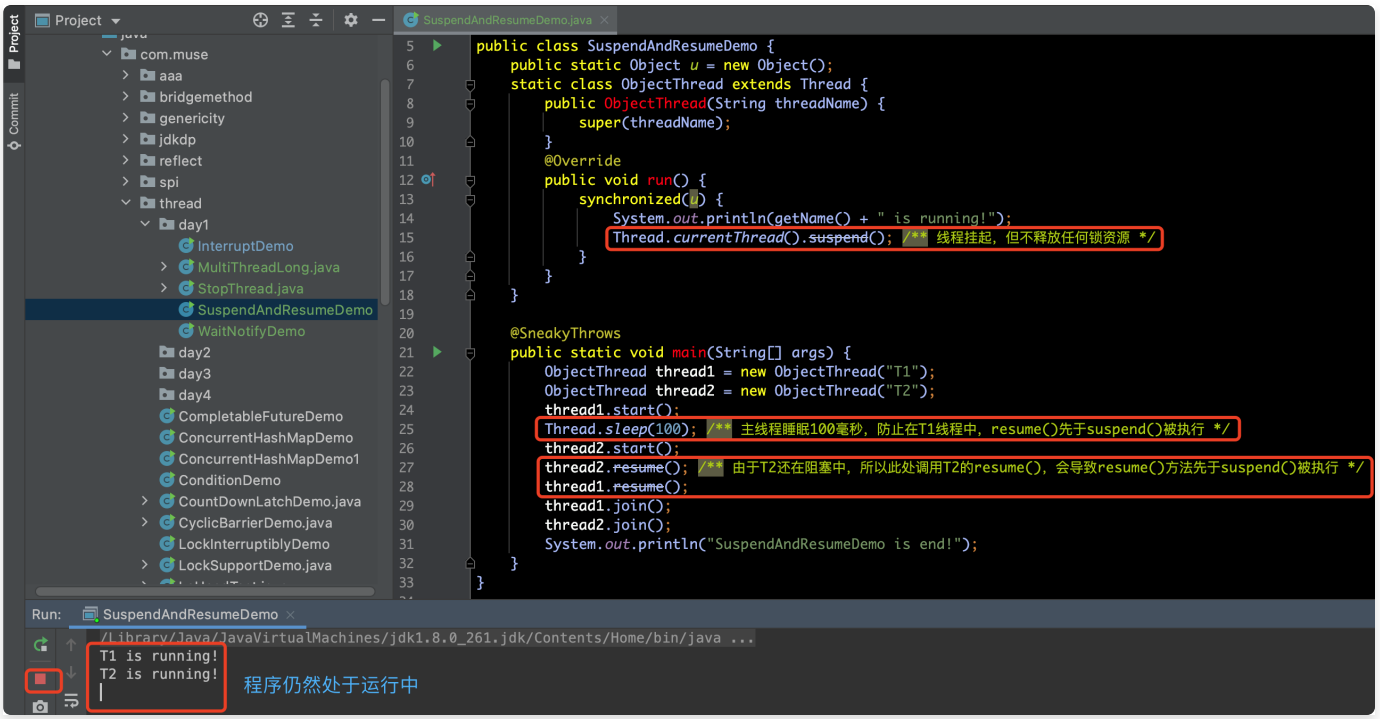
- 此时如果还希望T1可以解锁，则可以使用wait(long timeout)方式，即：如果T1等待timeout时间依然没有被执行notify，则自动解除等待状态。

```
WaitNotifyDemo.java
    public static class T1 extends Thread {
        @Override
        public void run() {
            synchronized(obj) {
                System.out.println(System.currentTimeMillis() + ":T1 start!");
                try {
                    /** Step1: 释放obj对象锁，进入obj对象的wait队列 */
                    System.out.println(System.currentTimeMillis() + ":T1 wait for object!");
                    obj.wait(1000);  等待1秒钟，如果依然没有被执行notify，则解除等待，试图再次获得锁
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                /** Step3: T2执行完毕，T1需要获得obj锁，才可以继续执行 */
                System.out.println(System.currentTimeMillis() + ":T1 end!");
            }
        }
    }

    public static class T2 extends Thread {
        @Override
        public void run() {
            // 由于Step1释放obj的锁，所以T2可以获得锁，然后运行
            synchronized(obj) {
                System.out.println(System.currentTimeMillis() + ":T2 start! notify one thread");
                /** Step2: 唤醒T1，但是由于T2没有执行完毕而释放obj的锁，所以T1等待中 */
                //obj.notify();
                System.out.println(System.currentTimeMillis() + ":T2 notify end!");
                try {
                    Thread.sleep(2000);
                    System.out.println(System.currentTimeMillis() + ":T2 sleep 2000ms end!");
```

```
Run: WaitNotifyDemo
1647936160351:T1 start!
1647936160352:T1 wait for object!
1647936160352:T2 start! notify one thread
1647936160352:T2 notify end!
1647936162357:T2 sleep 2000ms end!
1647936162357:T1 end!

Process finished with exit code 0
```

2.5> suspend&resume（被废弃）

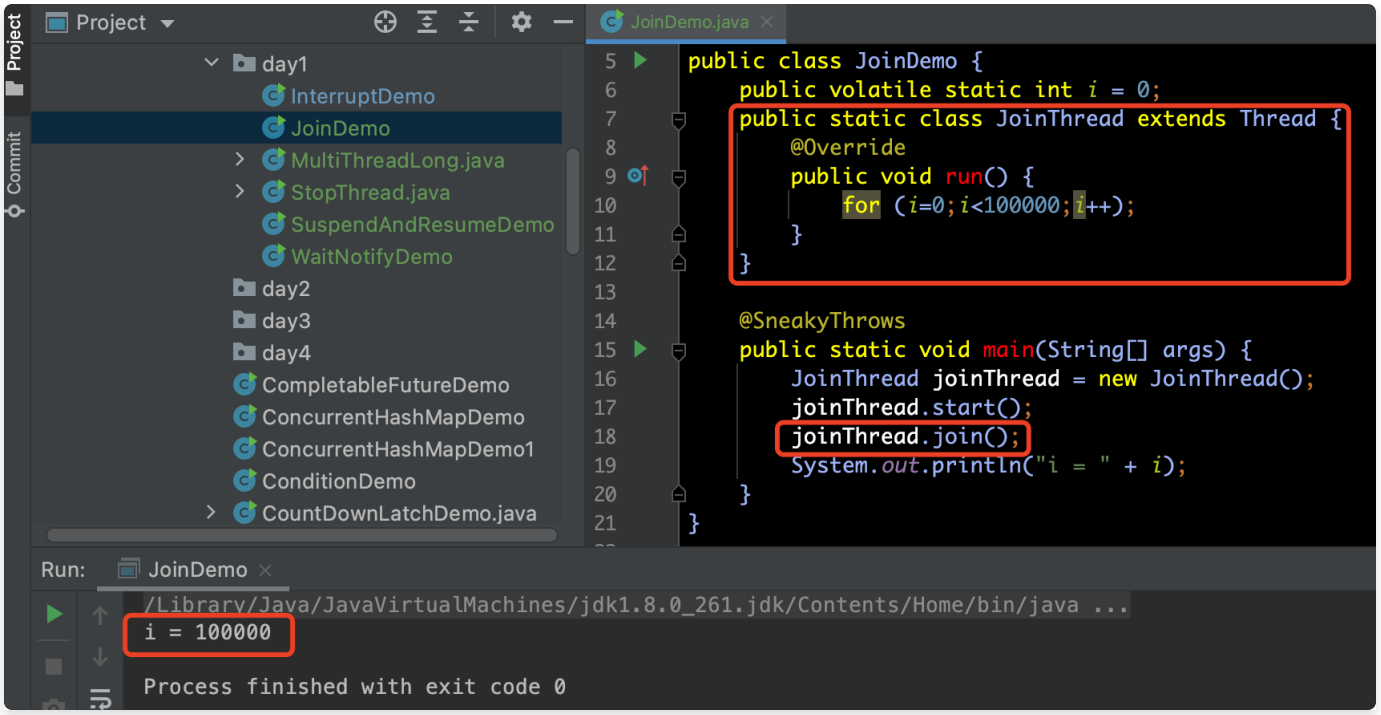
- suspend用于操作线程挂起，它会在暂停线程的同时，并锁住资源而不去释放。
- resume用于解除线程挂起，它会让线程继续执行。
- 如果一个线程的resume先于suspend被执行了，那么这个线程就会永久被挂起。并且，从它的线程状态上看，居然还是Runnable状态！如下图所示：

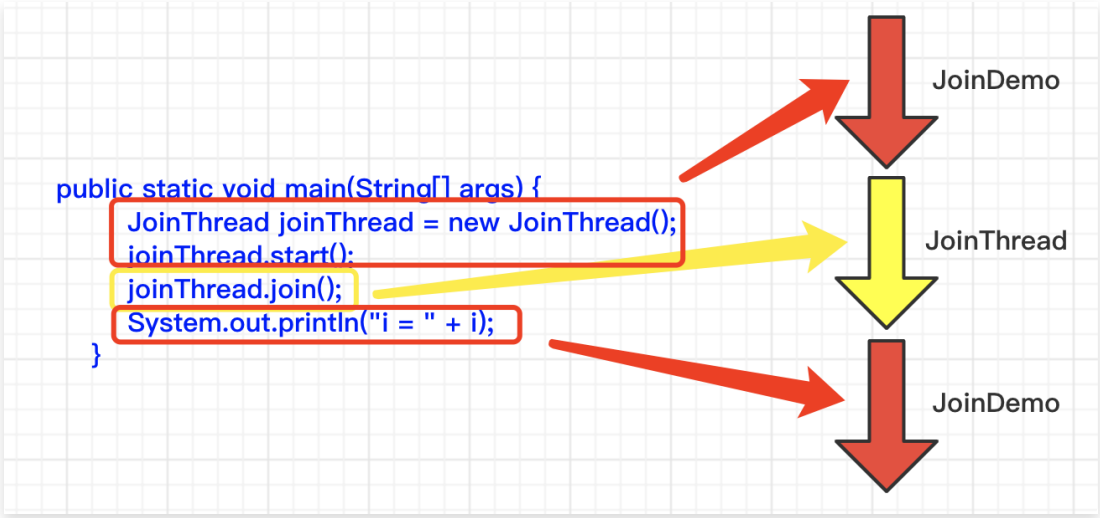



```
muse@muse:/Users/muse> jps
55116 Launcher
55117 SuspendAndResumeDemo
90796
55326 Jps
muse@muse:/Users/muse> jstack 55117
2022-03-15 09:46:45
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.261-b12 mixed mode):

"Attach Listener" #13 daemon prio=9 os_prio=31 tid=0x00007fb09c032800 nid=0x5507 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"T2" #12 prio=5 os_prio=31 tid=0x00007fb09a864000 nid=0xa803 runnable [0x000070000537b000]
   java.lang.Thread.State: RUNNABLE
        at java.lang.Thread.suspend0(Native Method)
        at java.lang.Thread.suspend(Thread.java:1032)
        at com.muse.thread.day1.SuspendAndResumeDemo$ObjectThread.run(SuspendAndResumeDemo.java:15)
        - locked <0x00000007156b0830> (a java.lang.Object)
```

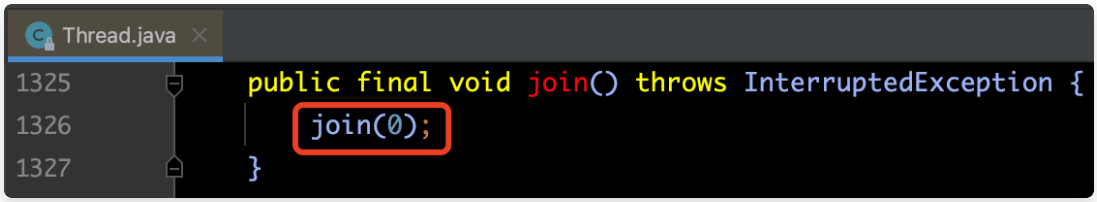
2.6> join&yield

- 当一个线程的输入非常依赖于另外一个或者多个线程的输出，此时，这个线程就需要等待依赖线程执行完毕，才能继续执行。JDK提供了join()操作来实现这个功能





- join()的本质就是让调用线程wait()在当前线程对象实例上。下面为相关代码：



Thread.java

```
1241    public final synchronized void join(long millis)
1242    throws InterruptedException {
1243        long base = System.currentTimeMillis();
1244        long now = 0;
1245
1246        if (millis < 0) {
1247            throw new IllegalArgumentException("timeout value is negative");
1248        }
1249
1250        if (millis == 0) {
1251            while (isAlive()) {
1252                wait(0);
1253            }
1254        } else {
1255            while (isAlive()) {
1256                long delay = millis - now;
1257                if (delay <= 0) {
1258                    break;
1259                }
1260                wait(delay);
1261                now = System.currentTimeMillis() - base;
1262            }
1263        }
1264    }
```

Object.java

```
382    public final native void wait(long timeout) throws InterruptedException;
```

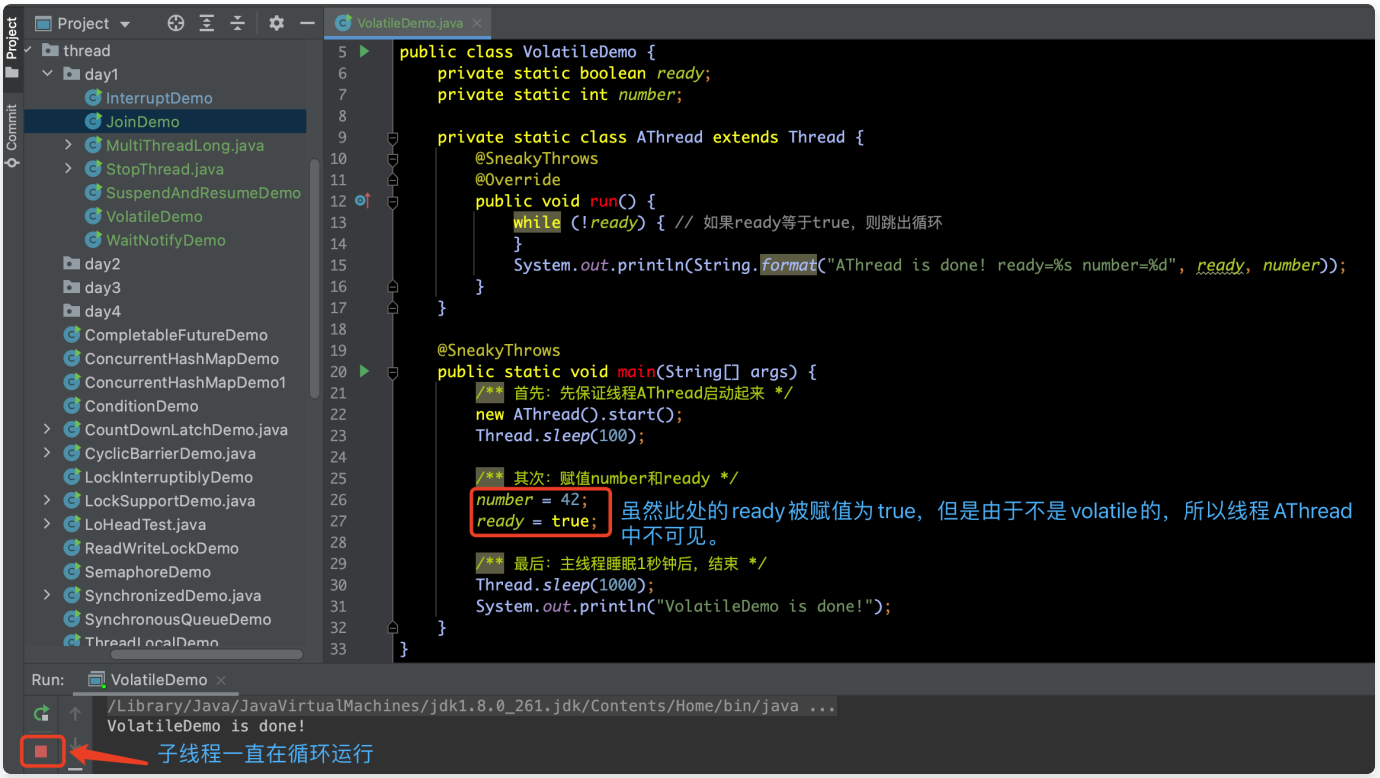
- 从上面源码中可以看到，它让调用线程在当前线程对象上进行等待。当线程执行完成后，被等待的线程会在退出前调用notifyAll()通知所有的等待线程继续执行。因此，值得注意的一点是：**不要在应用程序中，在Thread对象实例上使用类似wait()或者notify()等方法**，因为这很有可能会影响系统API的工作，或者被系统API所影响。
- Thread.yield()方法会使当前线程让出CPU。让出CPU并不表示当前线程不执行了。当前线程在让出CPU后，还会进行CPU资源的争夺，但是是否能够再次被分配到，就不一定了。因此，对Thread.yield()的调用就好像是在说：我已经完成一些最重要的工作了，我应该是可以休息一下了。可以给其他线程一些工作机会了。

三、volatile

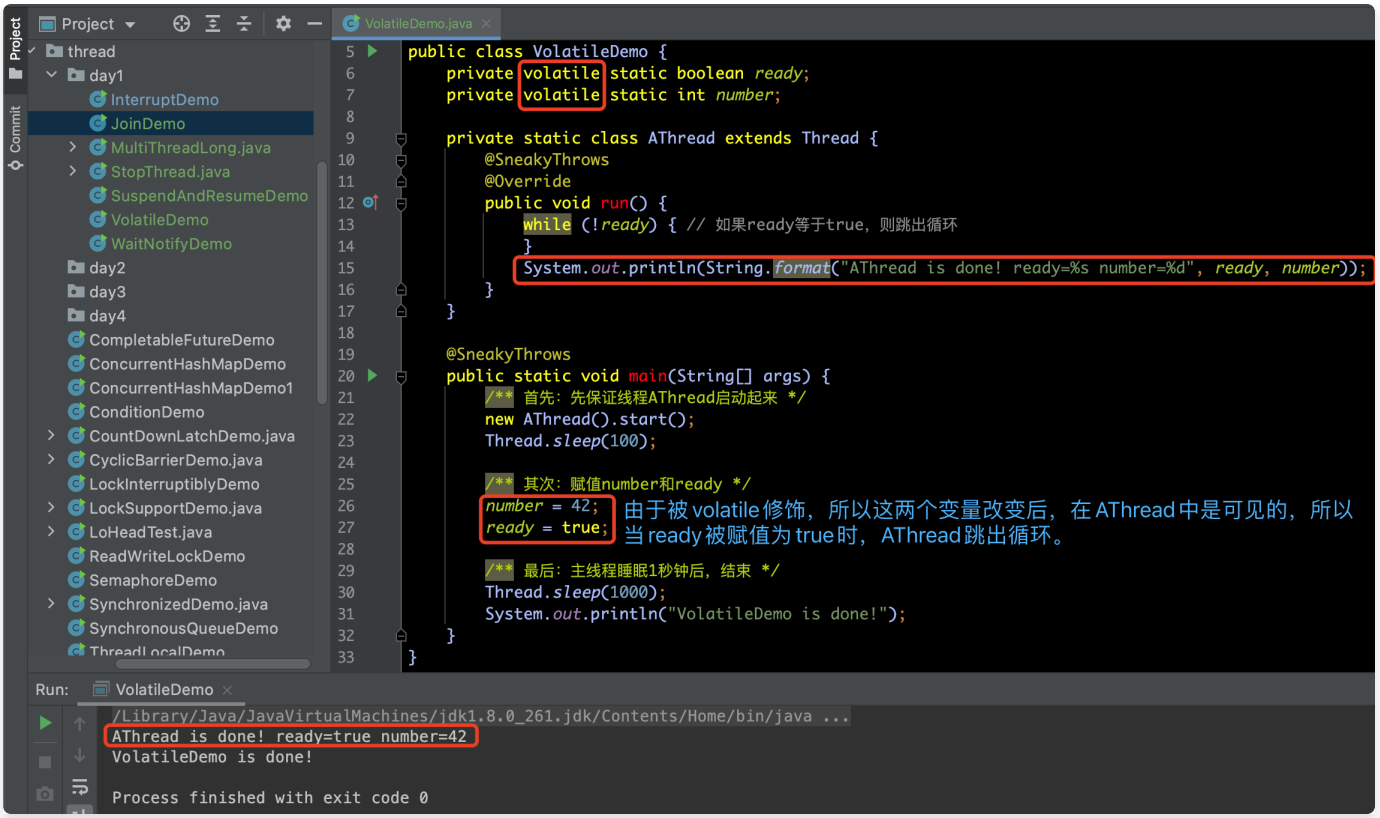
- 正常情况下，如果我们不使用volatile，那么每条线程都会有自己的缓存，当全局变量被修改时，其

他线程可能并不会被通知到。

- **volatile并不能真正的保证线程安全。它只能确保一个线程修改了数据后，其他线程能够看到这个改动。**
- 如下所示，即使ready在主线程中被赋值为true，依然无法在子线程中获得最新修改的值，从而结束while循环：



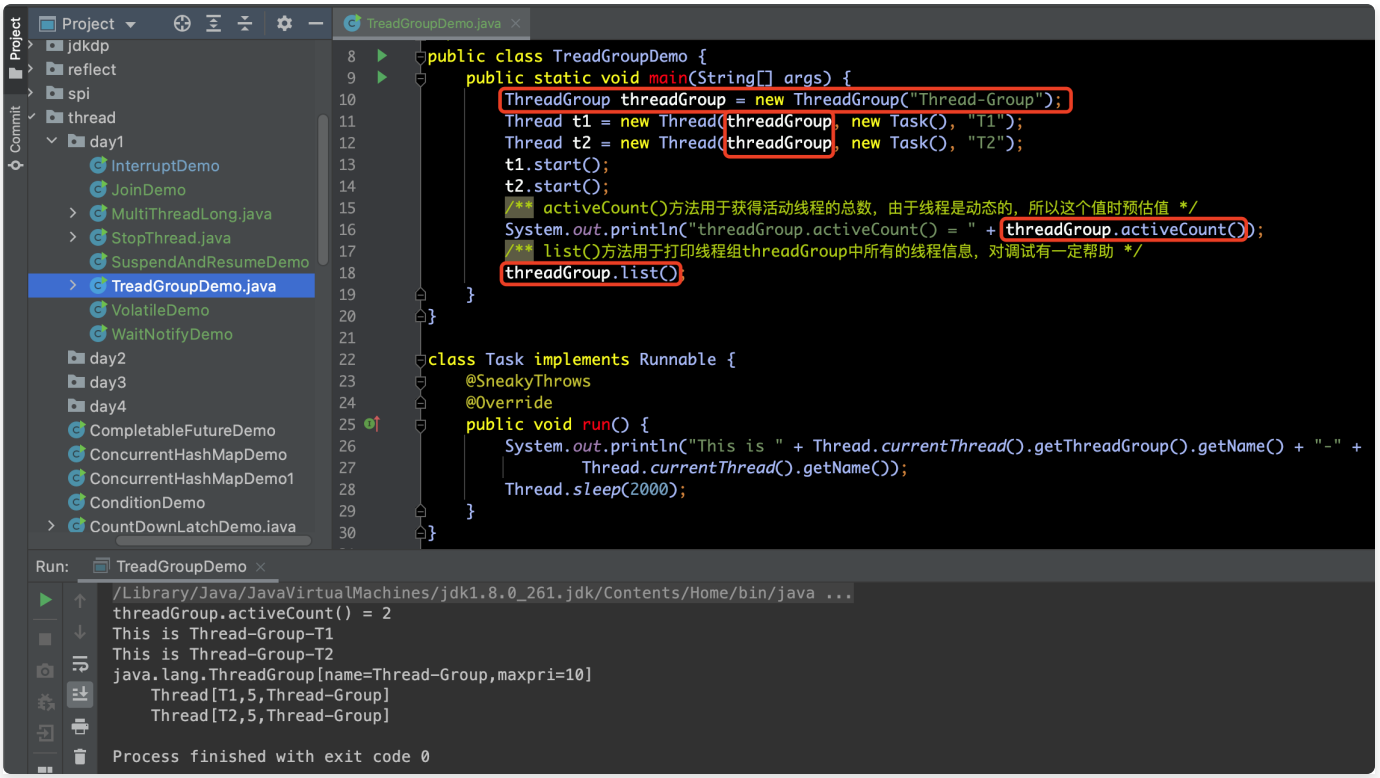
- 当我们使用volatile去申明变量时，就等于告诉了虚拟机，这个变量极有可能会被某些程序或者线程修改。为了确保这个变量被修改后，应用程序范围内的所有线程都能够“看到”这个改动，虚拟机就必须采用一些特殊的手段，保证这个变量的可见性等特点。如下所示：



- volatile并不能代替锁，它也无法保证一些复合操作的原子性。比如通过volatile是无法保证i++的原子性操作的。

四、ThreadGroup

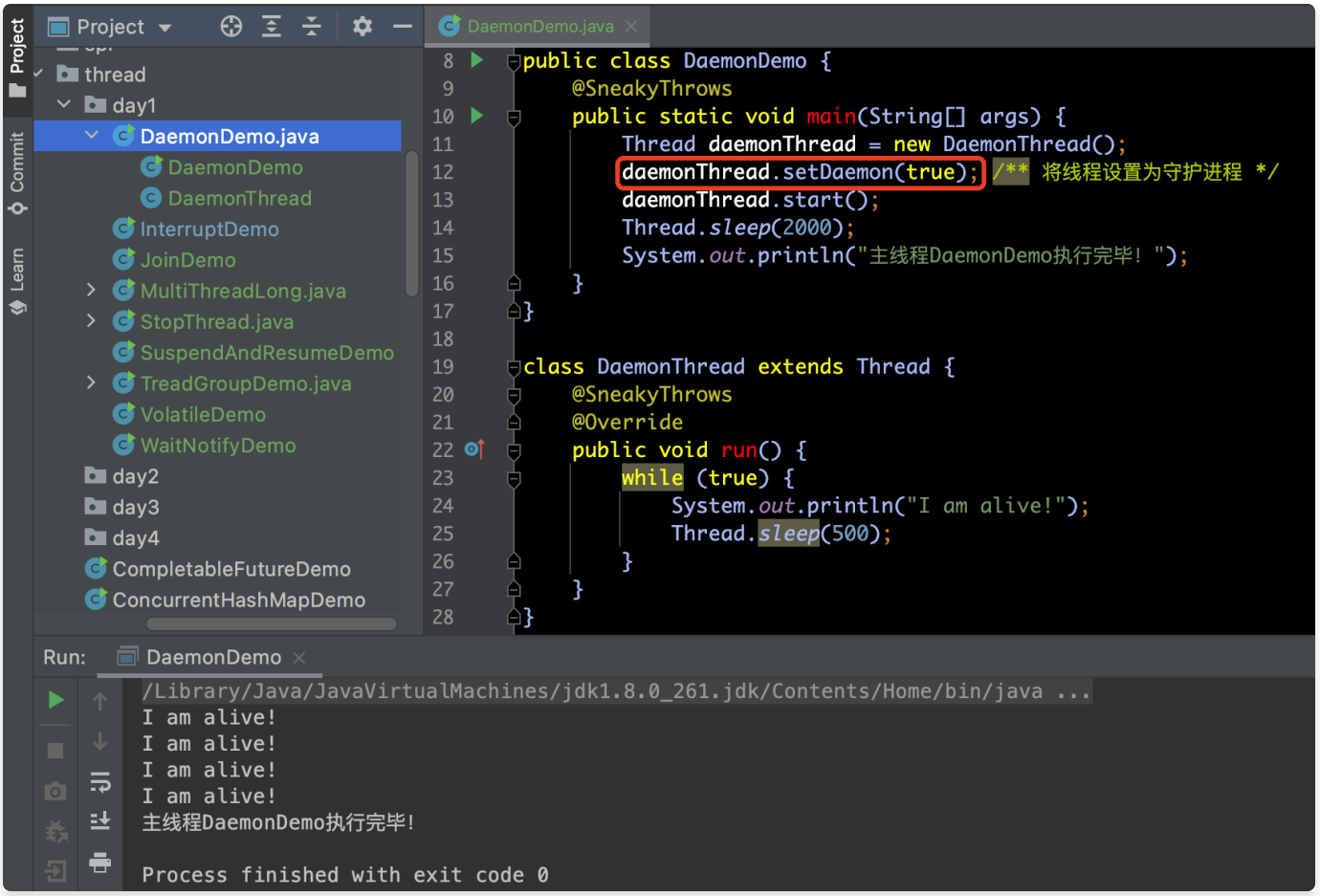
- 在一个系统中，如果线程数量很多，而且功能分配比较明确，就可以将相同功能的线程放置在一个线程组里。如下图所示：



- 建议大家在创建线程和线程组的时候，给它们取一个好听的名字。

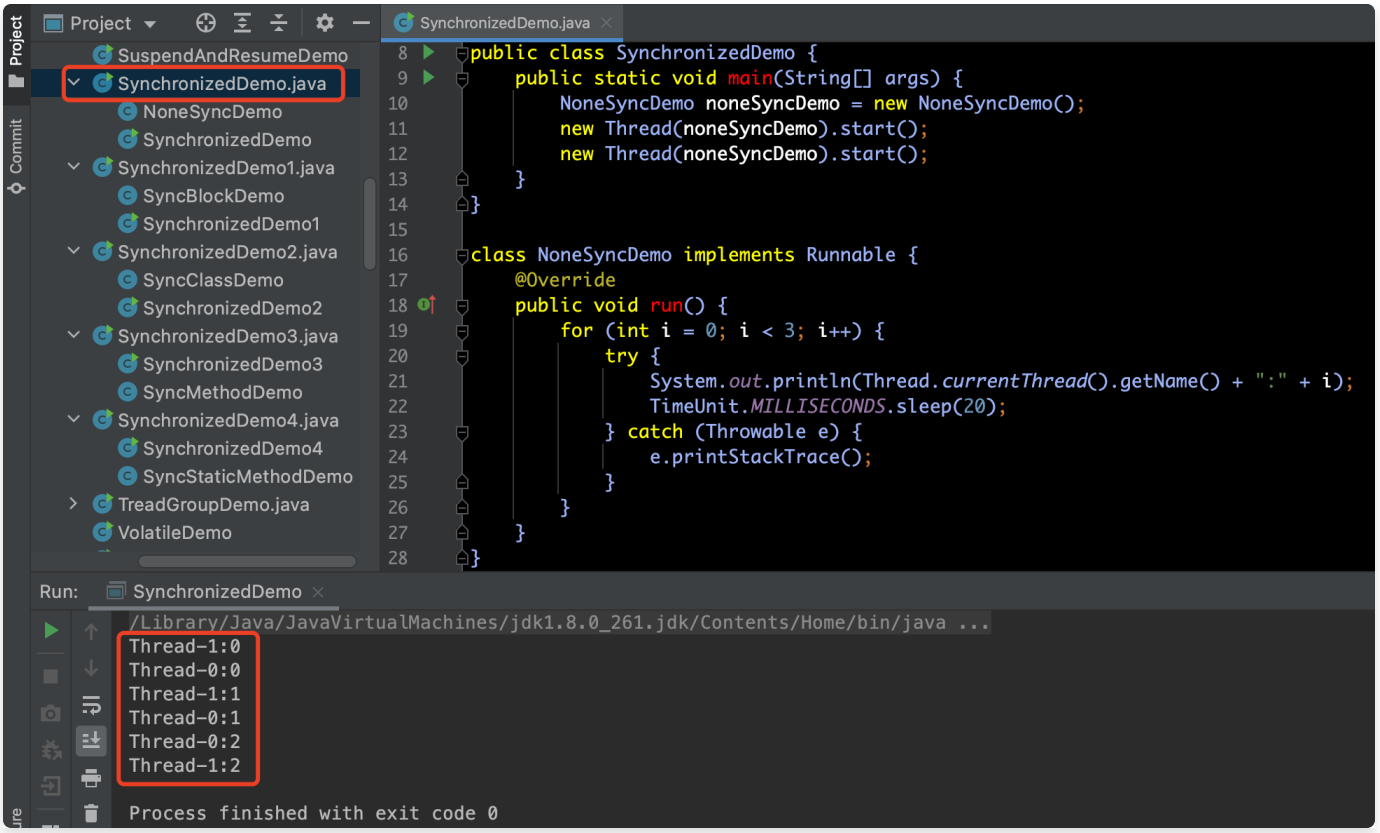
五、Daemon

- 守护线程是一种特殊的线程，它会在后台默默地完成一些系统性的服务。当一个Java应用内，只有守护线程时，Java虚拟机就自然退出了。我们可以通过调用setDaemon(true)的方式，设置线程为守护线程。这里需要注意，**设置守护线程必须在线程调用start()方法之前设置setDaemon(true)**，否则会报IllegalThreadStateException。

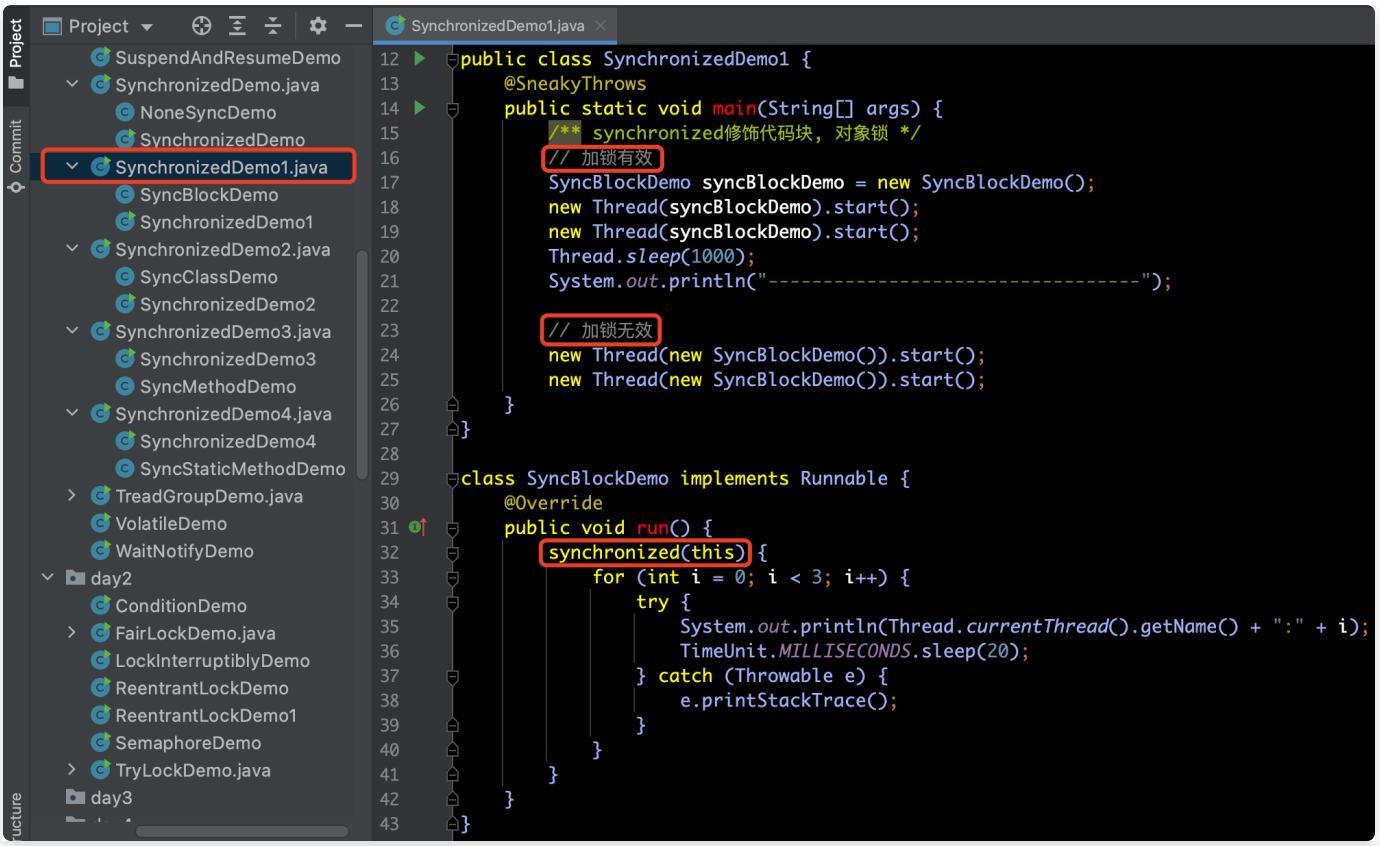


六、synchronized

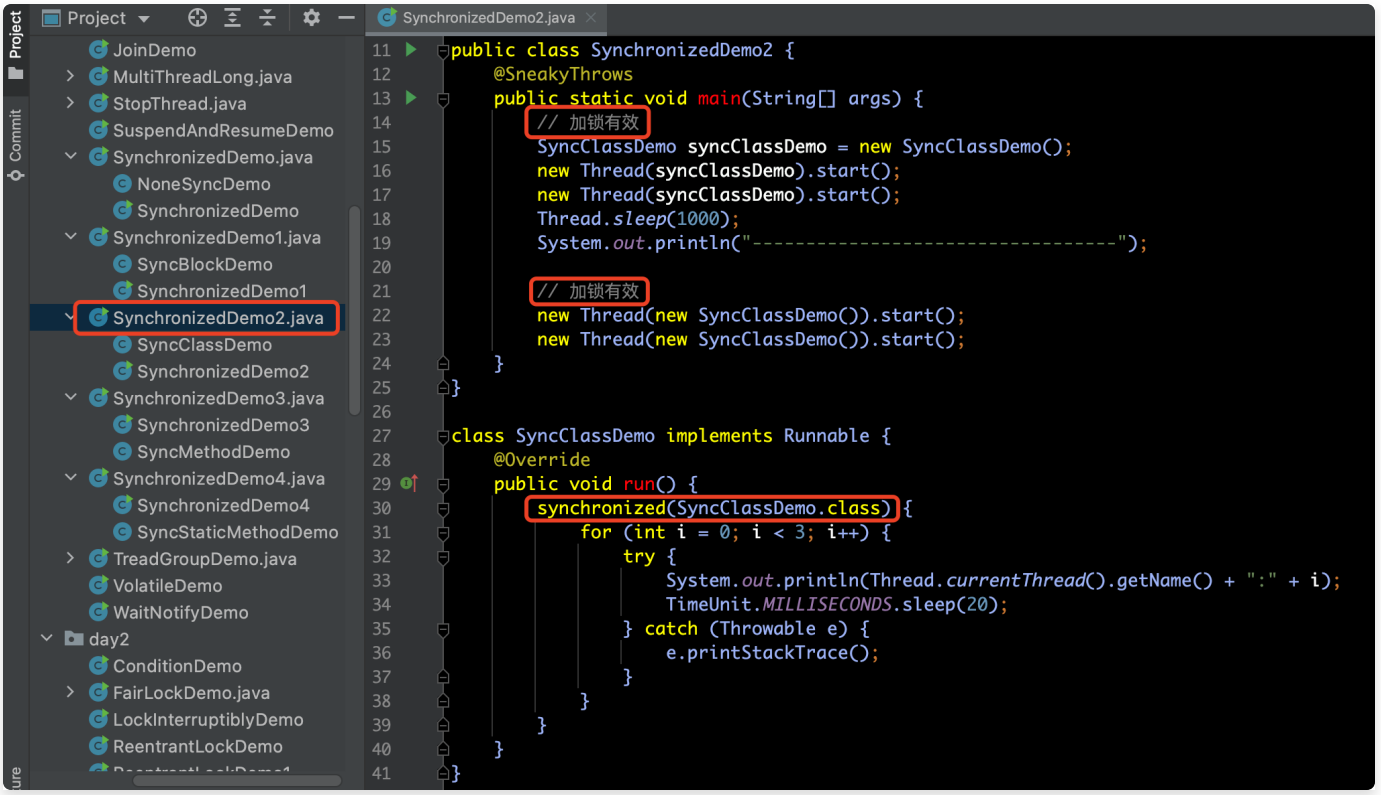
- 情况1：**不使用**synchronized修饰的对象，乱序输出。

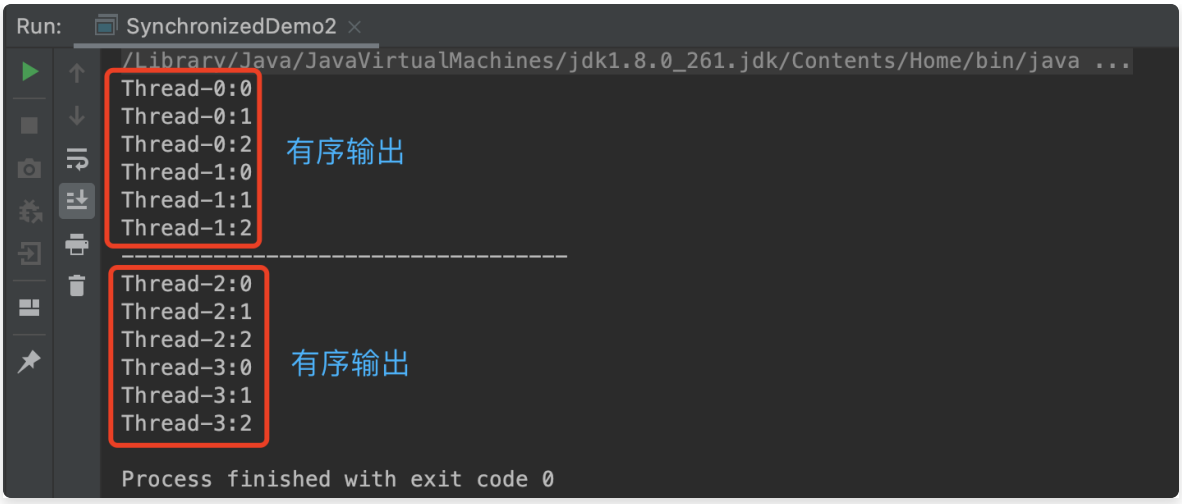


- 情况2：synchronized**修饰代码块——对象**

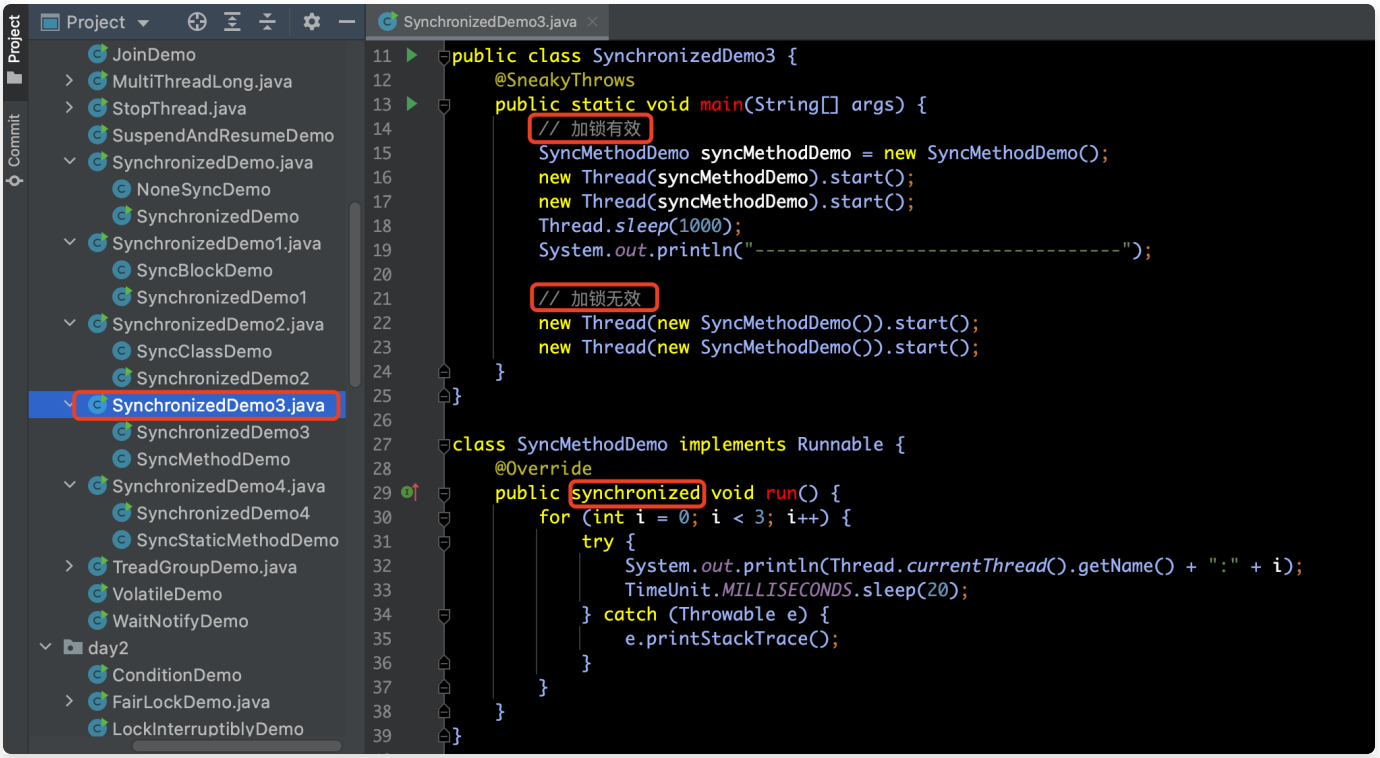


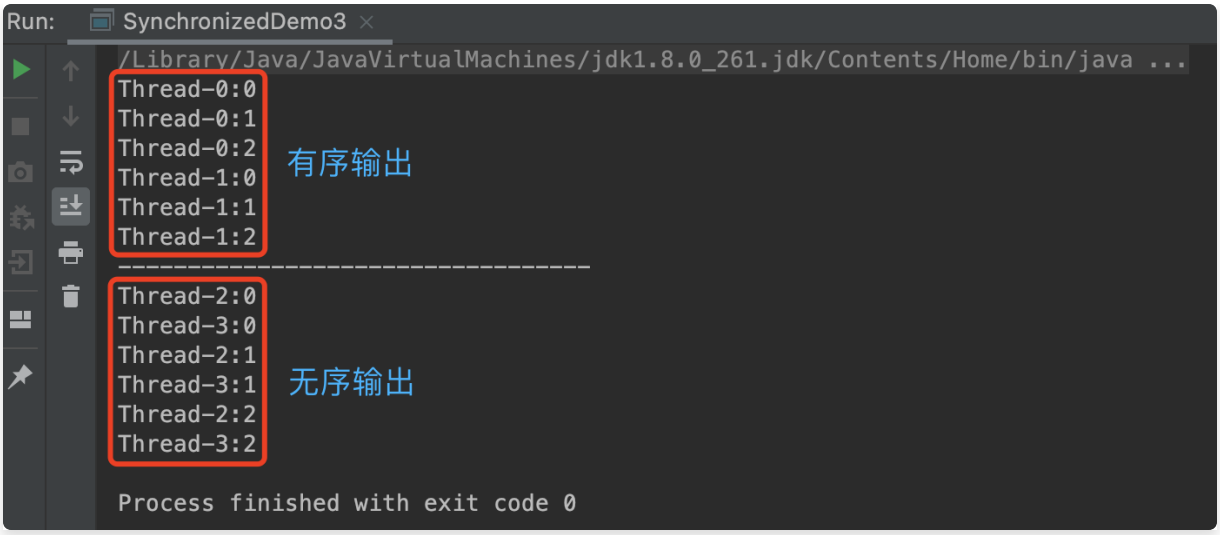
- 情况3：synchronized**修饰代码块——类**



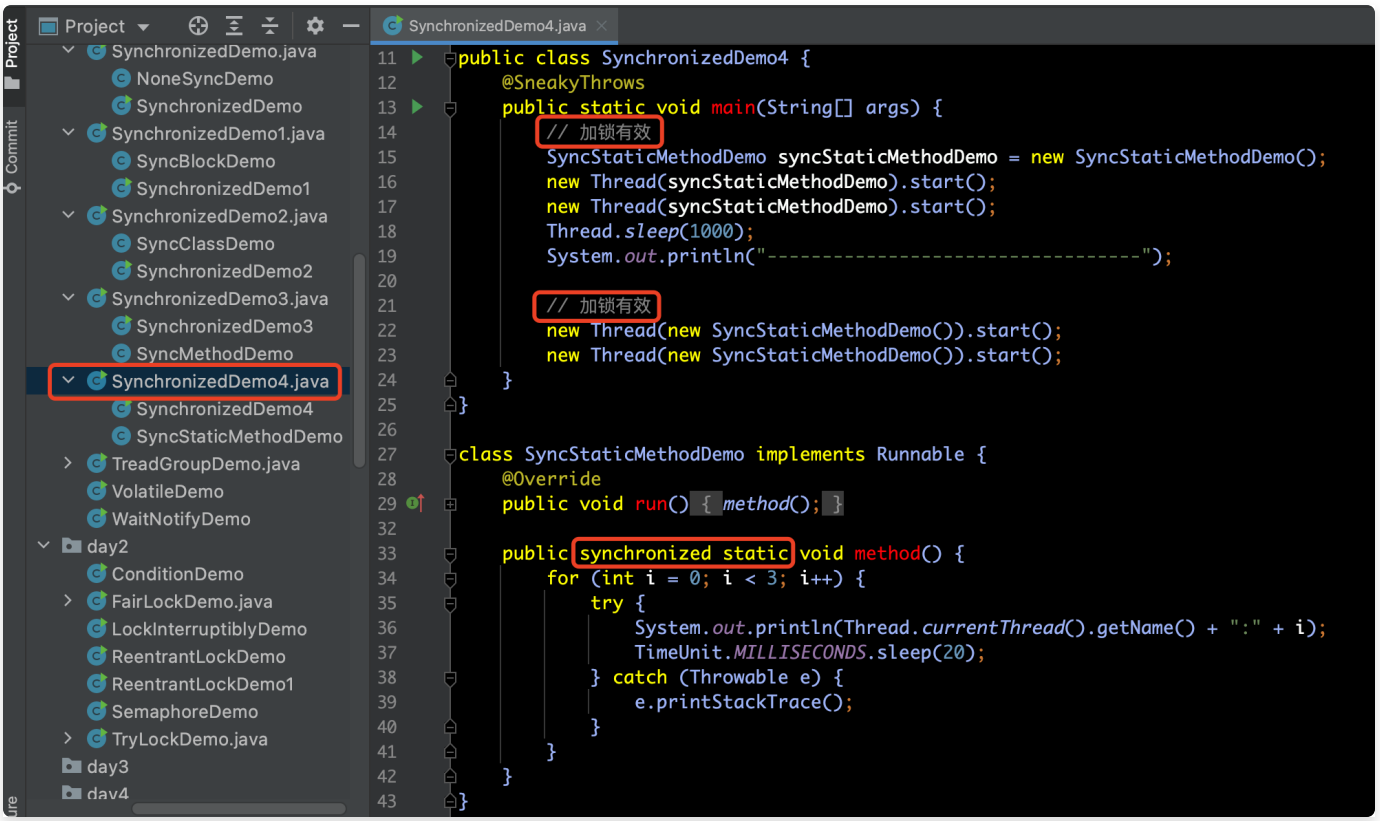


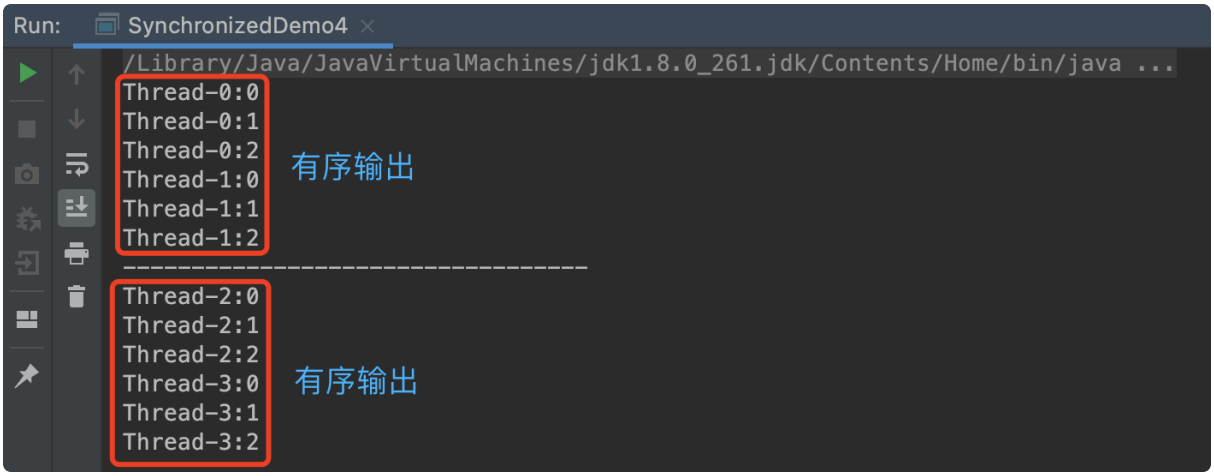
- 情况4：synchronized修饰**方法**





- 情况5：直接作用于**静态方法**——相当于对当前类加锁，进入同步代码前要获得当前**类的锁**。





七、锁的优化策略

- 偏向锁

  - 如果一个线程获得了锁，那么锁就进入偏向模式。当这个线程再次请求锁时，无须在做任何同步操作。这样就节省了大量有关锁申请的操作，从而提高了程序性能。

- 轻量级锁

  - 如果偏向锁失败,虚拟机并不会立即挂起线程.它还会使用一种称为轻量级锁的优化手段。
  - 轻量级锁的操作也很轻便，它只是**简单地将对象头部作为指针，指向持有锁的线程堆栈的内部，来判断一个线程是否持有对象锁**。
  - 如果线程获得轻量级锁成功，则可以顺利进入临界区。如果轻量级锁加锁失败，则表示其他线程抢先争夺到了锁，那么当前线程的锁请求就会膨胀为重量级锁。

- 自旋锁

  - 锁膨胀后，虚拟机为了避免线程真实地在操作系统层面挂起，虚拟机还会再做最后的努力——自旋锁。
  - 系统会进行一次赌注：它会假设在不久的将来，线程可以得到这把锁。因此，虚拟机会**让当前线程做几个空循环**（这也是自旋的含义），在经过若干次循环后，如果可以得到锁，那么就顺利进入临界区。如果还不能获得锁，才会真实地将线程在操作系统层面挂起。

- 锁消除

  - 锁消除是一种更彻底的锁优化。Java虚拟机在JIT编译时，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁。通过锁消除，可以节省毫无意义的请求锁时间。
  - 比如，我们有可能在一个不可能存在并发竞争的场合使用Vector，而Vector内部使用了synchronized请求锁。例如如下代码：

```java
public String[] createStrings() {
    Vector<String> vector = new Vector<>();
    for (int i=0; i<100; i++) {
        vector.add(Integer.toString(i));
    }
    return vector.toArray(new String[]{});
}
```

```java
// Vector.java
785     public synchronized boolean add(E e) {
786         modCount++;
787         ensureCapacityHelper(elementCount + 1);
788         elementData[elementCount++] = e;
789         return true;
790     }
```

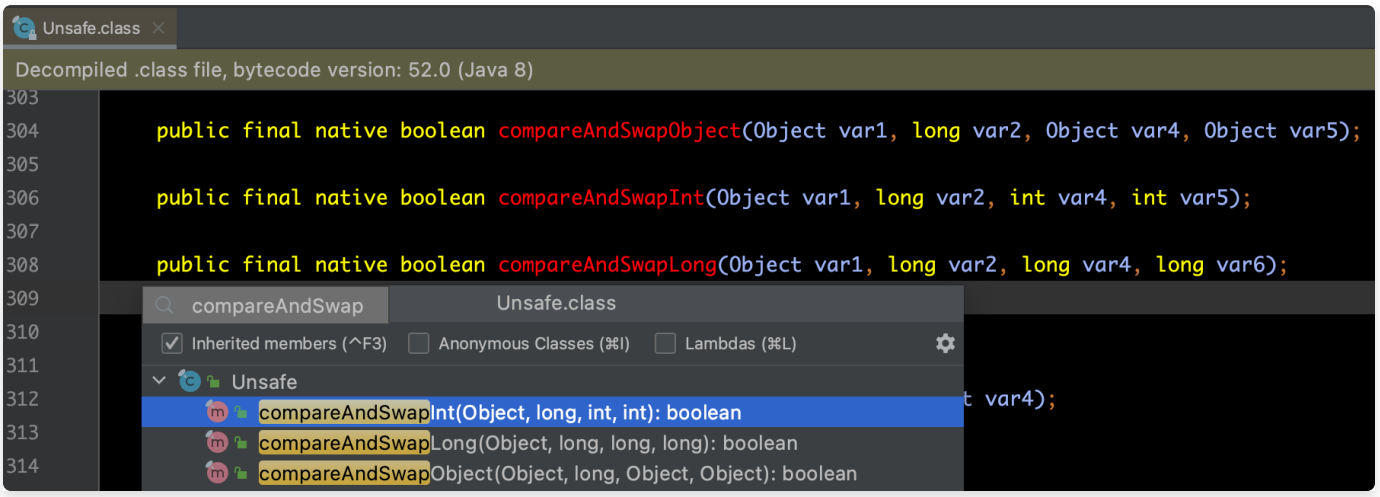
【注】在上述代码中的Vector，由于变量vector只在createStrings()函数中使用，因此，它只是一个单纯的局部变量。局部变量是在线程栈上分配的，属于线程私有的数据，因此不可能被其他线程访问。所以，如果虚拟机检测到这种情况，就**会将这些无用的锁操作去除掉**。

- 锁消除涉及的一项关键技术为**逃逸分析**。所谓逃逸分析就是观察某一个变量是否会逃出某一个作用域。

八、无锁

8.1> CAS

- 在Unsafe类里，包含着CAS的操作函数。它采用无锁的乐观策略，由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小得多。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销。因此，它要比基于锁的方式拥有更优越的性能。



- CAS算法的过程

- 它包含四个参数CAS(object,offset,expectdValue,newValue)，分别是：
  object：待更新的对象
  offset：待更新变量的offset偏移量
  expectdValue：表示预期值，
  newValue：表示新值
- 那么，仅当object+offset定位到的值等于expectdValue值时，才会将其值设为newValue，如果与expectdValue值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。
- 在硬件层面，大部分的现代处理器都已经支持原子化的CAS指令。

8.2> AtomicInteger

- 通过使用AtomicInteger，我们可以得益于CAS等CPU指令，保证Integer操作的原子性。如下所示：

```
AtomicIntegerDemo.java
/**
 * 无锁的线程安全整数AtomicInteger
 */
public class AtomicIntegerDemo {
    static AtomicInteger atomicInteger = new AtomicInteger();
    static Demo demo = new Demo();

    @SneakyThrows
    public static void main(String[] args) {
        for (int i=0; i< 100; i++) {
            new Thread(() -> {
                for (int j=0; j<1000;j++) {
                    atomicInteger.incrementAndGet();
                    demo.score++;
                }
            }).start();
        }
        Thread.sleep(1000);
        System.out.println("atomicInteger = " + atomicInteger);
        System.out.println("demo.score = " + demo.score);
    }
}

class Demo {
    volatile int score;
}
```

```
Run: AtomicIntegerDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
atomicInteger = 100000
demo.score = 99683

Process finished with exit code 0
```

- 我们来看一下AtomicInteger的incrementAndGet()方法是怎么实现的（基于JDK1.8）

AtomicInteger.java

```java
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

AtomicInteger.java

```java
    public final int incrementAndGet() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }
```

【注】由于针对incrementAndGet() 方法，是先执行increment，然后再执行get，所以获取的是increment之后的值；而unsafe.getAndAddInt先返回get到的值，然后内部在执行addInt，所以在整体的方法这里需要"+1";

- 此处使用了CAS的方式，来进行原子的+1操作

Unsafe.class

Decompiled .class file, bytecode version: 52.0 (Java 8)

```java
    public final int getAndAddInt(Object var1, long var2, int var4) {
        int var5;
        do {
            var5 = this.getIntVolatile(var1, var2);
        } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

        return var5;
    }
```

8.3> Unsafe

- Unsafe封装了一些类似指针的操作。因为指针是不安全的，如果指针指错了位置，或者计算指针偏移量时出错，结果可能是灾难性的，你很有可能覆盖别人放入内存中的数据，导致系统崩溃。

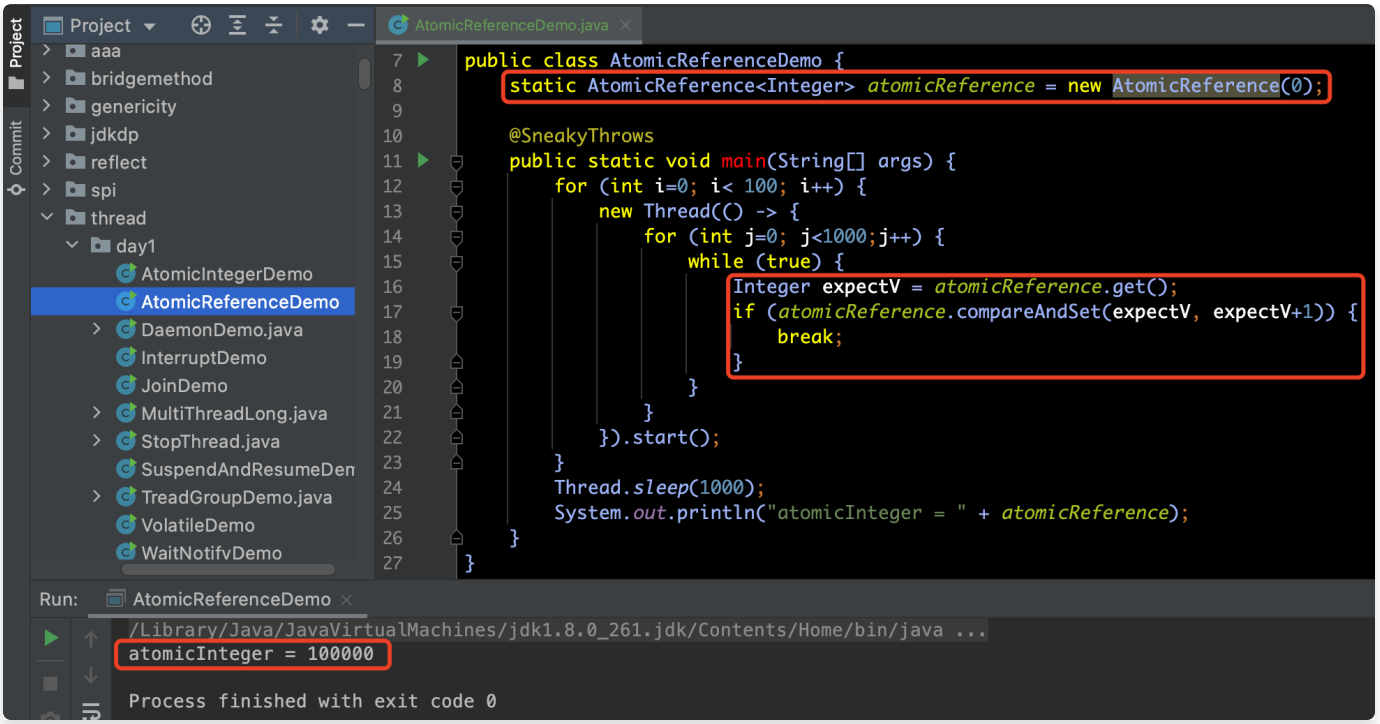
- Unsafe类还提供了一些常用的方法，如下所示：

```java
// 获得给定对象偏移量上的int值
public native int getInt(Object o, long offset);

// 设置给定对象偏移量上的int值
public native void putInt(Object o, long offset, int x);

// 获得字段在对象中的偏移量
public native long objectFieldOffset(Field f);

// 设置给定对象的int值，使用volatile语义
public native void putIntVolatile(Object o, long offset, int x);

// 获得给定对象的int值，使用volatile语义
public native int getIntVolatile(Object o, long offset);

// 和putIntVolatile()一样，但是它要求被操作字段就是volatile类型的
public native void putOrderedInt(Object o, long offset, int x);
```

- 但是，如果我们要使用Unsafe类，需要调用getUnsafe()函数，如果这个类的ClassLoader不为null，就直接抛出异常，拒绝工作。因为我们知道，只有系统类加载器才会返回null。因此，这也使得**我们自己的应用程序无法直接使用Unsafe类**。它是一个JDK内部使用的专属类。

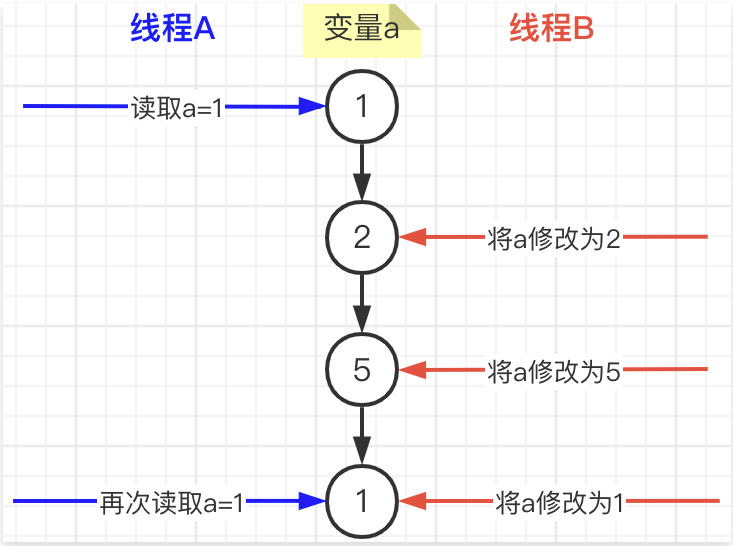
8.4> AtomicReference

- AtomicReference和AtomicInteger非常类似，只是AtomicReference对应普通的对象引用。如下所示：



8.5> AtomicStampedReference

- 有一点需要注意的是，当你获得对象当前数据后，在准备修改为新值前，对象的值被其他线程连续修改了N次，而最终对象的值又被恢复为旧值。这样，当前线程就无法正确判断这个对象究竟是否被修改过。如下所示：



- 那么，针对上面的问题，我们可以通过使用AtomicStampedReference这种带有时间戳的对象引用来解决，因为它包含了一个stamp参数，类似版本或时间戳的概念，如果想要CAS成功，就必须ref和stamp都满足期待值，如下所示：

AtomicStampedReferenceDemo.java

```java
public class AtomicStampedReferenceDemo {
    public static AtomicStampedReference<Integer> reference = new AtomicStampedReference(1, 0);
    public static void main(String[] args) {
        new AThread().start(); // 开启A线程
        new BThread().start(); // 开启B线程
    }
    /** A线程（将值加1） */
    public static class AThread extends Thread {
        @SneakyThrows
        @Override
        public void run() {
            Integer ref = reference.getReference();
            Integer stamp = reference.getStamp();
            System.out.println("[AThread] Before Sleep: ref=" + ref + ", stamp=" + stamp);
            Thread.sleep(1000);
            System.out.println("[AThread] After Sleep: ref=" + reference.getReference() + ", stamp=" + reference.getStamp());
            if (reference.compareAndSet(ref, ref + 1, stamp, stamp + 1)) {
                System.out.println("[AThread] Add one success!");
            } else {
                System.out.println("[AThread] Add one fail!");
            }
        }
    }
    /** B线程（先加2，再还原为原值） */
    public static class BThread extends Thread {
        private boolean flag = false;
        private Integer originRef = 0;

        @SneakyThrows
        @Override
        public void run() {
            while (true) {
                Integer ref = reference.getReference();
                Integer stamp = reference.getStamp();
                if (flag) {
                    if (reference.compareAndSet(ref, originRef, stamp, stamp + 1)) {
                        System.out.println("[BThread] ref=" + reference.getReference() + ",stamp=" + reference.getStamp());
                        break;
                    }
                } else {
                    if (reference.compareAndSet(ref, ref + 2, stamp, stamp + 1)) {
                        System.out.println("[BThread] ref=" + reference.getReference() + ",stamp=" + reference.getStamp());
                        originRef = ref;
                        flag = true;
                    }
                }
                Thread.sleep(100);
            }
        }
    }
```

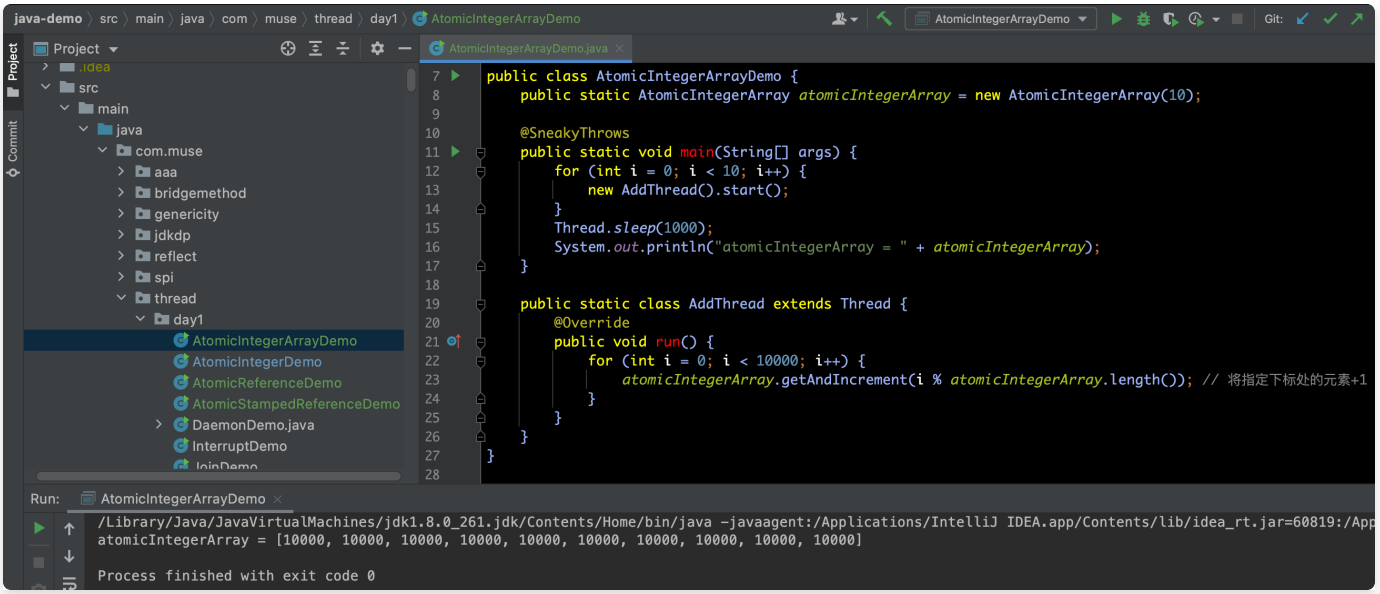
构造函数中：
arg0表示ref的初始值
arg1表示stamp的初始值

在sleep的这1000毫秒内，虽然ref值又被修改为原值，但是stamp已经变更了

```
Run: AtomicStampedReferenceDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
[AThread] Before Sleep: ref=1, stamp=0
[BThread] ref=3,stamp=1
[BThread] ref=1,stamp=2
[AThread] After Sleep: ref=1, stamp=2
[AThread] Add one fail!

Process finished with exit code 0
```

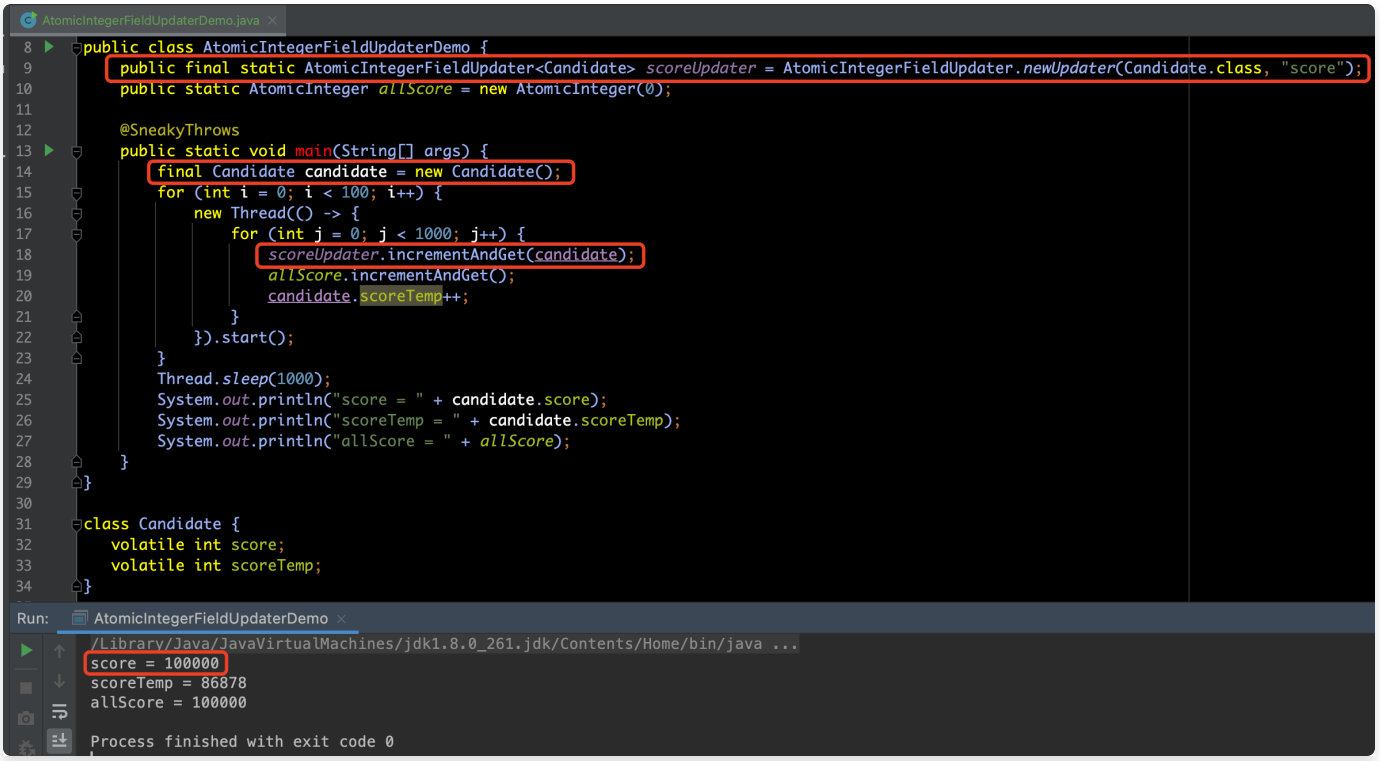
8.6> AtomicIntegerArray

- 除了基本数据类型之外，JDK还为我们提供了数组这种复合类型结构。当前可用的原子数组有AtomicIntegerArray、AtomicLongArray和AtomicReferenceArray。它们本质都是对数组类型进行封装，使用Unsafe类通过CAS的方式控制数组在多线程下的安全性。下面以AtomicIntegerArray为例进行演示：



8.7> AtomicIntegerFieldUpdater

- 它的作用是，让普通变量也能享受原子操作，并且在不改动或极少改动原有代码的基础上，让普通的变量也能享受CAS操作带来的线程安全性。
- 根据数据类型不同，这个Updater有三种。分别是AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater，顾名思义，它们分别可以对int、long和Object进行CAS修改。
- allScore是通过AtomicInteger来进行递增计算的；score是通过AtomicIntegerFieldUpdater进行封装执行CAS操作的；scoreTemp就是普通的字段通过自增计算的。循环了100*1000次，总计数应该是100000。如下所示；



- 使用Updater的注意事项：

> - Updater只能修改它可见范围内的变量。因为Updater使用反射得到这个变量，如果变量不可见，就会出错。比如**不能将score声明为private**。
> - 为了确保变量被正确的读取，它**必须是volatile类型**的。
> - 由于CAS操作会通过对象实例中的偏移量直接进行赋值，因此，它**不支持static字段**（即：Unsafe.objectFieldOffset()不支持静态变量）

九、ThreadLocal

- 详情请见：*【源码解析】ThreadLocal.pdf*