【讲义】第2讲：AQS和JUC

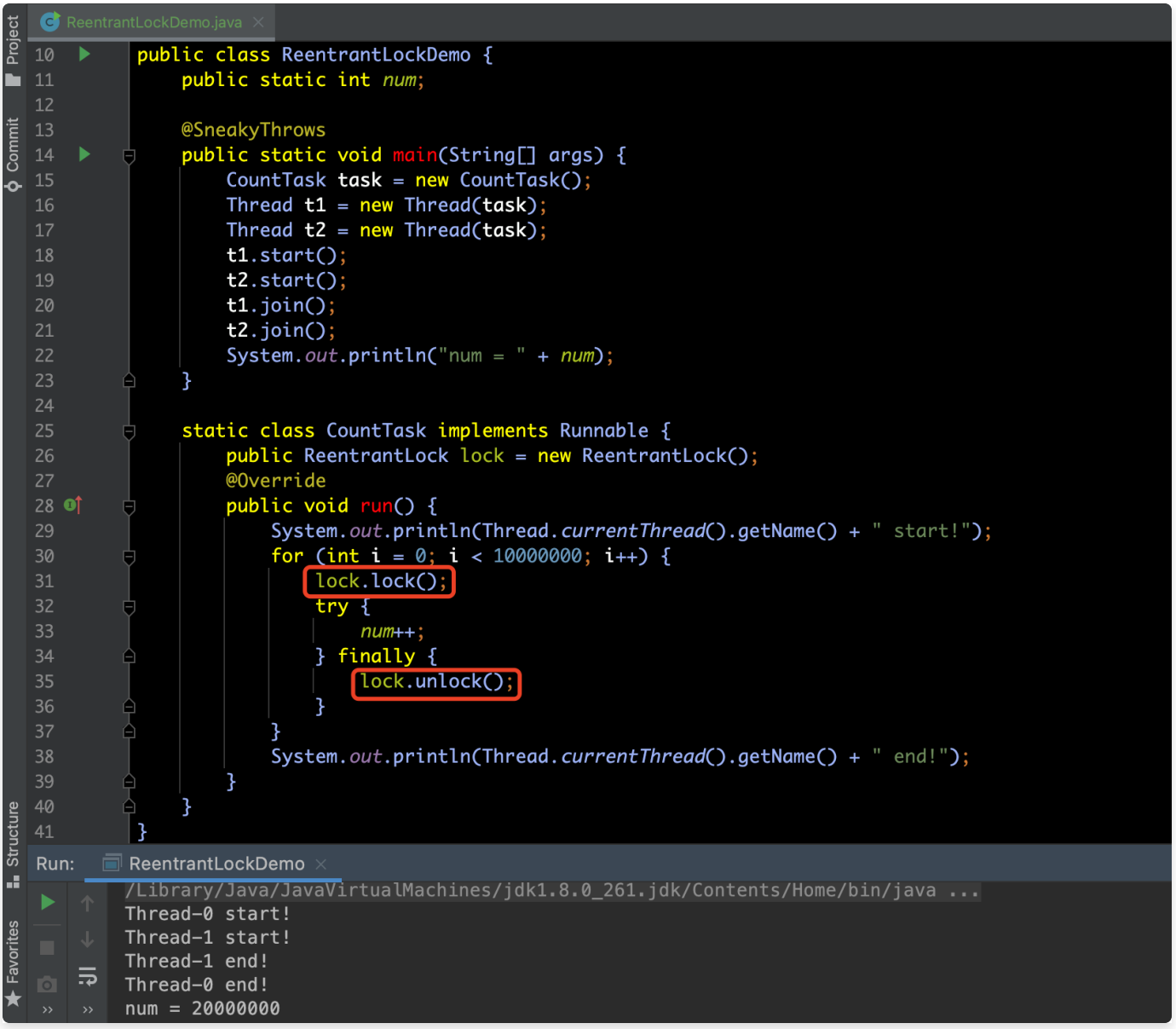
一、ReentrantLock重入锁

1.1> 概述

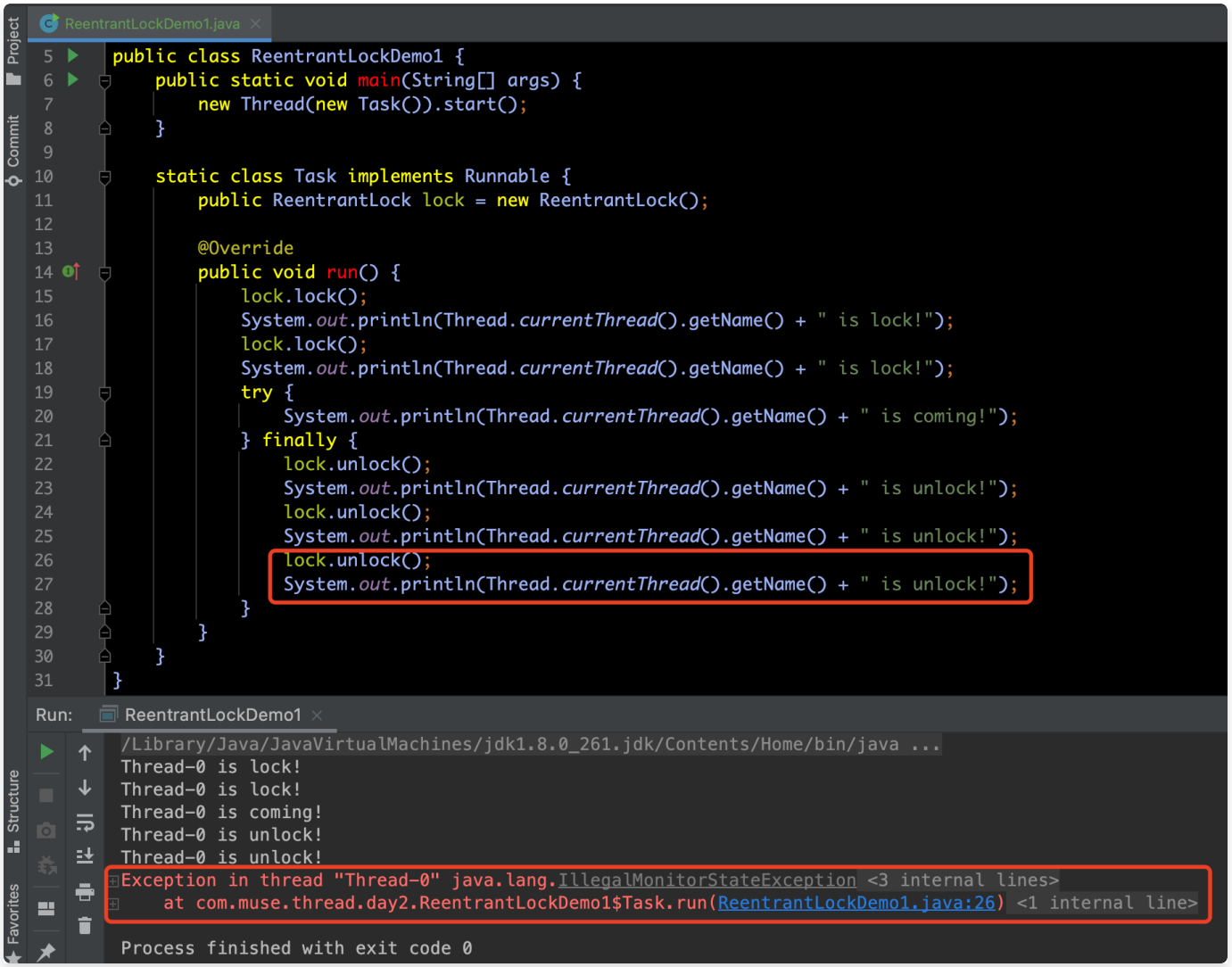
- 重入锁可以完全替代synchronized关键字。在JDK5.0的早期版本中，重入锁的性能远远好于synchronized，但从JDK6.0开始，JDK在synchronized上做了大量的优化，使得两者的性能差距并不大。重入锁对逻辑控制的灵活性要远远好于synchronized。
- 重入锁常用方法

> void lock()：获得锁，如果锁已经被占用，则等待。
> void lockInterruptibly()：获得锁，但优先响应中断。
> boolean tryLock()：尝试获得锁，如果成功，返回true；如果失败则返回false；获得不到锁，则不进行等待，立即返回。
> boolean tryLock(long time, TimeUnit unit)：在给定时间内尝试获得锁。
> boolean isHeldByCurrentThread()：判断担心线程是否持有锁。
> void unlock()：释放锁。

- 下面是使用重入锁的简单示例：



- 之所以称之为**重入锁**，就是一个线程允许反复进入。当然，这里的反复仅仅局限于一个线程；如果同一个线程多次获锁，那么在释放锁的时候，也必须释放相同次数。如果释放锁的次数多，那么会得到一个java.lang.IllegalMonitorStateException异常，反之，如果释放锁的次数少，那么相当于线程还持有这个锁。如下所示：



1.2> 中断响应 lockInterruptibly()

- 如果使用synchronized，要么获得锁，要么保持等待。而如果使用了重入锁，则提供了另一种可能，那就是线程可以被中断。也就是在等待锁的过程中，程序可以根据需要取消对锁的请求。即：**如果一个线程正在等待锁，那么它依然可以收到一个通知，被告知无须再等待，可以停止工作了**。可以很好的应对死锁问题。示例如下所示：

LockInterruptiblyDemo.java

```java
public class LockInterruptiblyDemo {
    private static ReentrantLock lock1 = new ReentrantLock();
    private static ReentrantLock lock2 = new ReentrantLock();

    public static void main(String[] args) throws Throwable {
        /** 可中断锁 */
        Thread t1 = new Thread(new ReentrantLockThread(lock1, lock2));
        Thread t2 = new Thread(new ReentrantLockThread(lock2, lock1));
        t1.start();
        t2.start();
        TimeUnit.MILLISECONDS.sleep(100);
        System.out.println("主线程开始沉睡第1秒！"); //主线程睡眠1秒，避免线程t1直接响应run方法中的睡眠中断
        TimeUnit.MILLISECONDS.sleep(1000);
        System.out.println("主线程线程，在"+ t1.getName() +"上开始执行interrupt()");
        t1.interrupt();
    }

    /** ReentrantLock的lockInterruptibly实现死锁 */
    static class ReentrantLockThread implements Runnable {
        private ReentrantLock lock1, lock2;

        public ReentrantLockThread(ReentrantLock lock1, ReentrantLock lock2) {
            this.lock1 = lock1;
            this.lock2 = lock2;
        }
        @Override
        public void run() {
            try {
                lock1.lockInterruptibly(); // 获得lock1的可中断锁
                System.out.println(Thread.currentThread().getName() + ", 加锁成功1-2! ");
                TimeUnit.MILLISECONDS.sleep(100); // 等待lock1和lock2分别被两个线程获取。产生死锁现象
                lock2.lockInterruptibly(); // 获得lock2的可中断锁
                System.out.println(Thread.currentThread().getName() + ", 加锁成功2-2! ");
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + ", 发生异常! ");
                e.printStackTrace();
            } finally {
                if (lock1.isHeldByCurrentThread()) {
                    lock1.unlock();
                }
                if (lock2.isHeldByCurrentThread()) {
                    lock2.unlock();
                }
                System.out.println(Thread.currentThread().getName() + ", 解锁成功! ");
            }
        }
    }
```

入参相反，可以导致死锁

中断锁

对lock1加锁

对lock2加锁

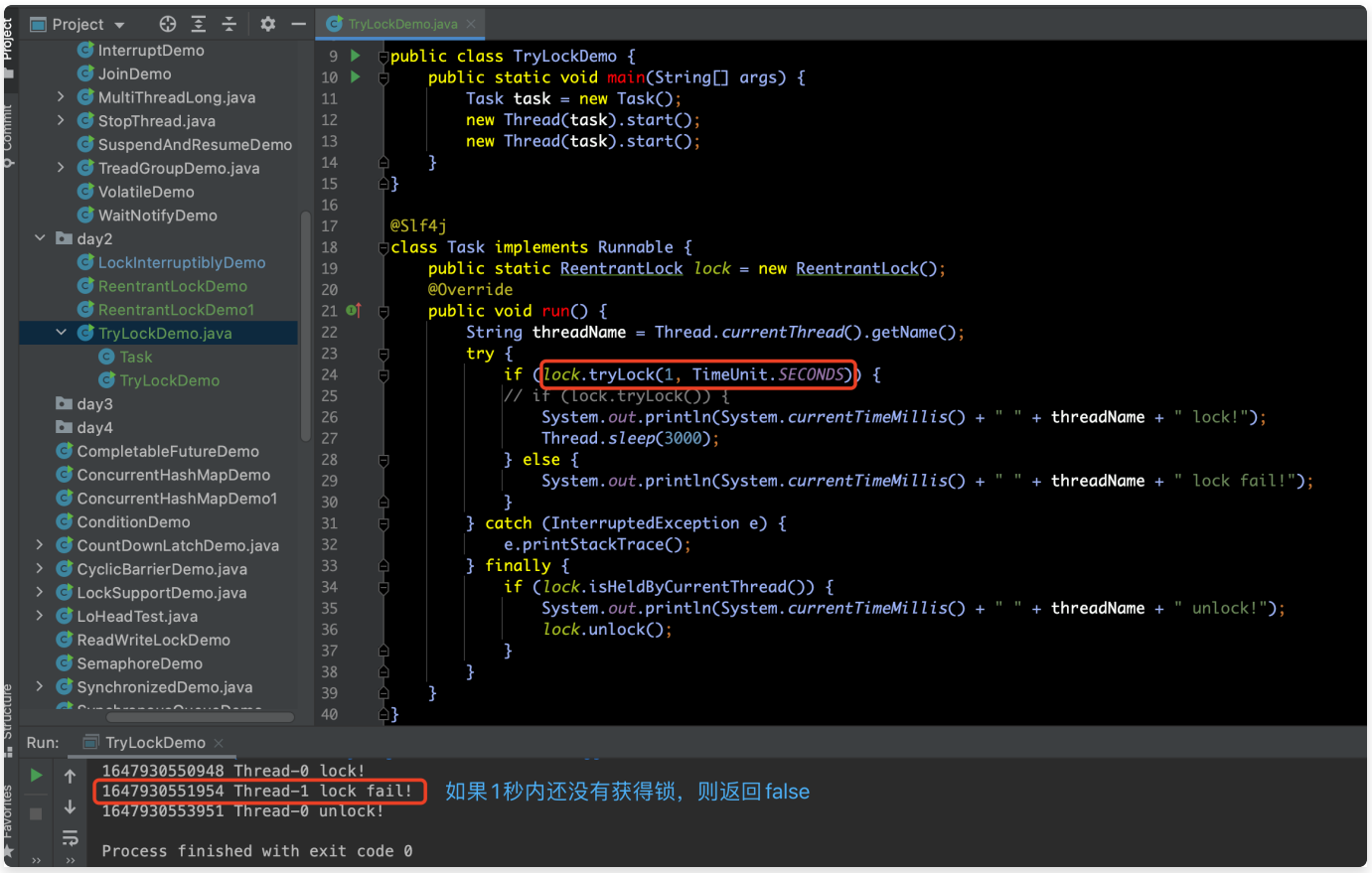
如果持有锁，则进行解锁

Run: LockInterruptiblyDemo

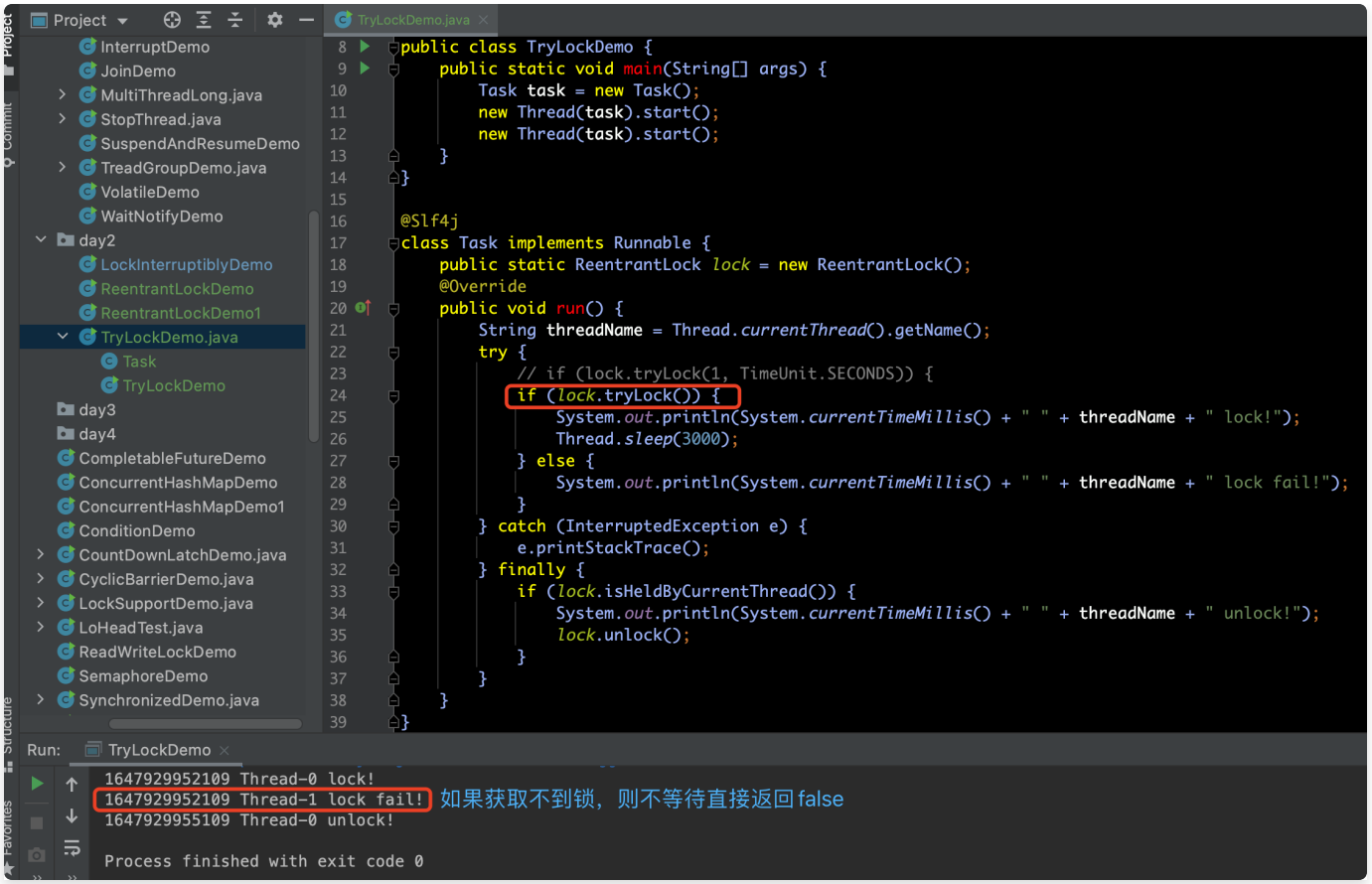
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
Thread-0, 加锁成功1-2!
Thread-1, 加锁成功1-2!
主线程开始沉睡第1秒!
主线程线程，在Thread-0上开始执行interrupt()
Thread-0, 发生异常!
Thread-0, 解锁成功!
Thread-1, 加锁成功2-2!
Thread-1, 解锁成功!
java.lang.InterruptedException <3 internal lines>
    at com.muse.thread.day2.LockInterruptiblyDemo$ReentrantLockThread.run(LockInterruptiblyDemo.java:62)

Process finished with exit code 0
```

1.3> 锁申请等待限时 tryLock(long time, TimeUnit unit)

- 除了等待外部通知之外，要避免死锁还有另外一种方式，就是**限时等待**。以下面为例，线程尝试获得锁，如果没有获得锁，则等待5秒钟。如果5秒钟之后依然没有获得锁，则返回false，表示获得锁失败。

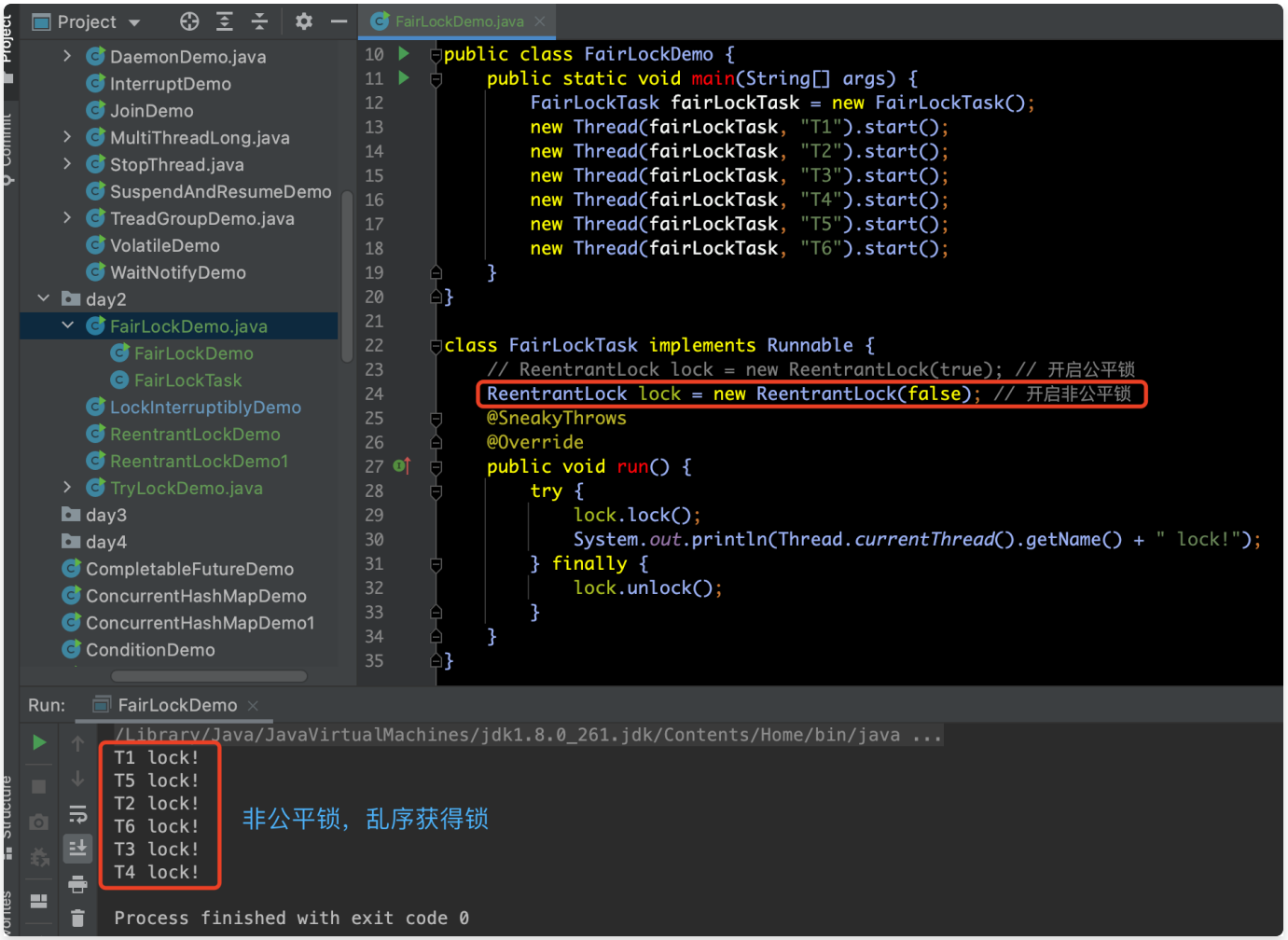


- tryLock()方法也可以不带参数直接运行。在这种情况下，当前线程会尝试获得锁，如果锁并未被其他线程占用，则申请锁会成功，并立即返回true。如果锁被其他线程占用，则**当前线程不会进行等待，而是立即返回false**。这种模式不会引起线程等待，因此也不会产生死锁。

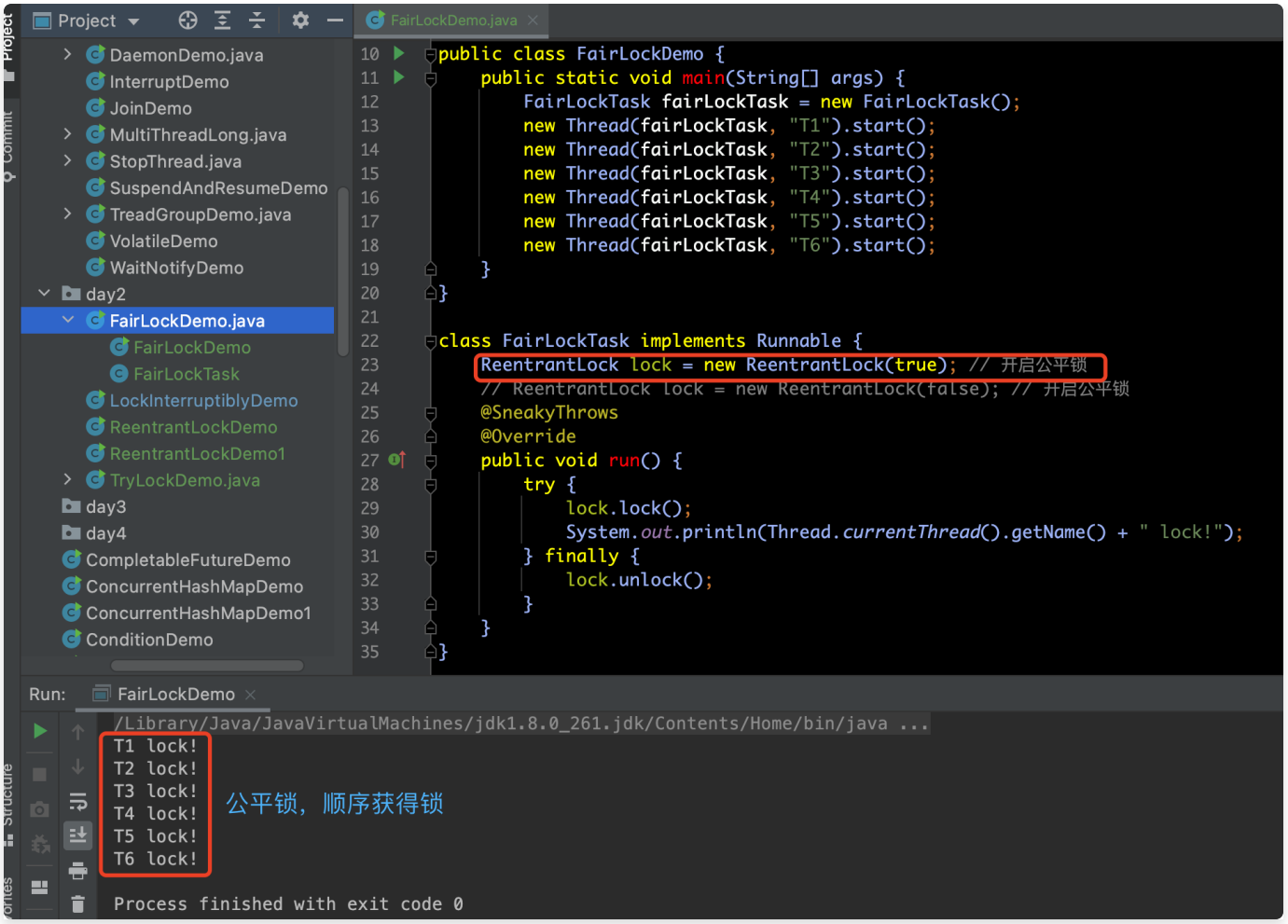


1.4> 公平锁和非公平锁

- 在大多数情况下，锁的申请都是非公平锁。系统只是会从这个锁的等待线程中随机选择一个。类似大家买票不去排队，乱哄哄的围在售票窗口前，售票员忙得焦头烂额，也顾不及谁先谁后，随便找一个人出票就完事儿了。



- 当入参为true时，则采用公平锁方式。要求系统维护一个有序队列，因此公平锁的实现成本比较高，性能相对也非常低下。因此，默认情况下，锁是非公平的。如果没有特别的需求，也不需要使用公平锁。



1.5> AQS源码解析

- 详情请见：*【源码解析】AbstractQueuedSynchronizer.pdf*

二、Condition重入锁的搭配类

- 相同点

> Condition和Object的wait()、notify()方法的作用是大致相同的，对应关系如下所示：
>
> - Condition.await()--->Object.wait()
> - Condition.signal()--->Object.notify()
> - Condition.signalAll()--->Object.notifyAll()

- 不同点

> Object.wait()和Object.notify()方法是和Synchronized关键字合作使用的；而Condition是与ReentrantLock相关联的。

- Condition常用方法

void await()：

会使当前线程等待，同时释放当前锁，当其他线程中使用signal()或者signalAll()方法时，线程会重新获得锁并继续执行。或者当线程被中断是，也能跳出等待。这和Object.wait()方法很相似。

void awaitUninterruptibly()：

与await()方法基本相同，区别是它不会在等待过程中响应中断。

long awaitNanos(long nanosTimeout)：

如果nanosTimeout时间内，没有被执行signal，则解除等待状态。

boolean await(long time, TimeUnit unit)：

如果time时间内，没有被执行signal，则解除等待状态。

boolean awaitUntil(Date deadline)：

如果deadline时间内，没有被执行signal，则解除等待状态。

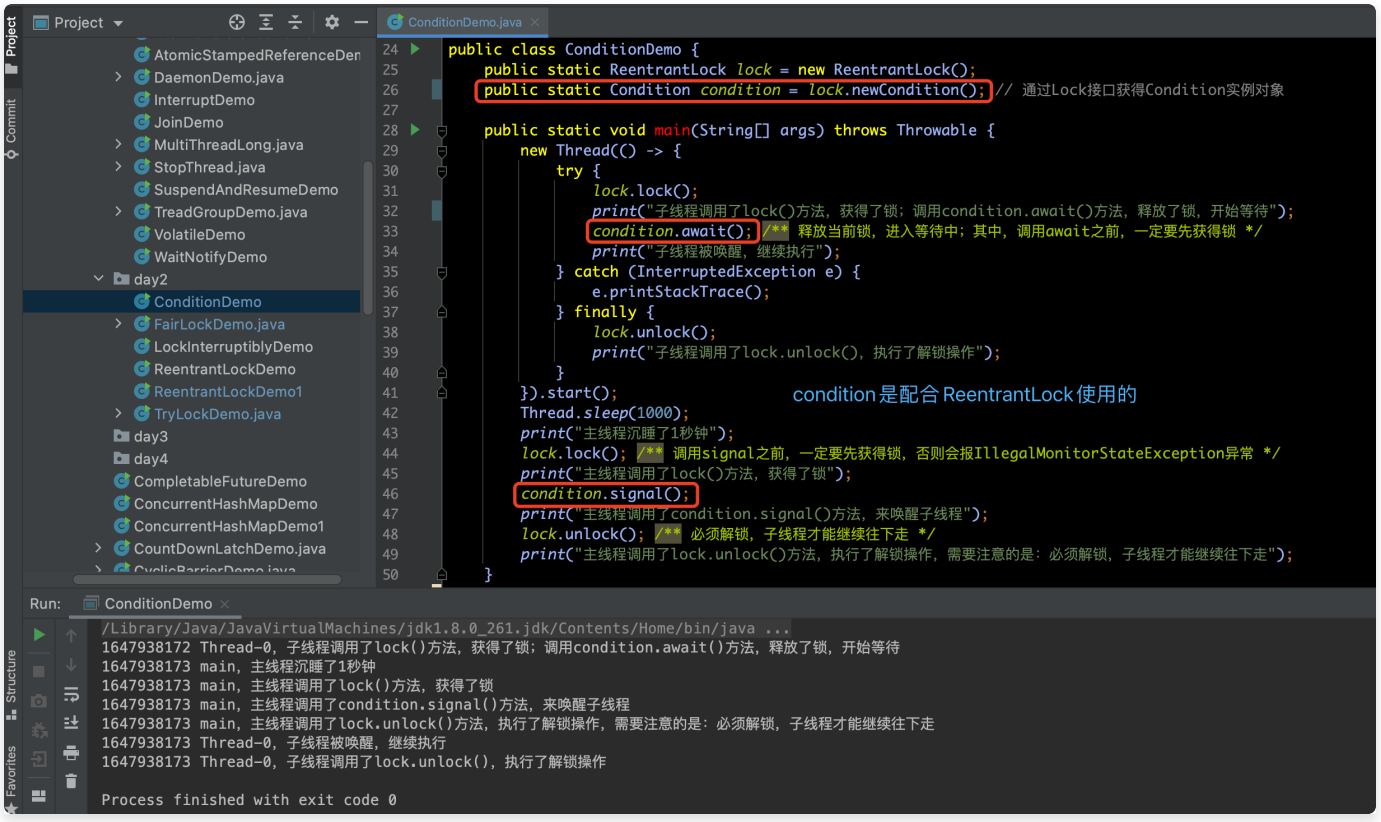
void signal()：

用于唤醒一个正在等待中的线程。

void signalAll()：

用于唤醒所有正在等待中的线程。

- Condition使用示例：



三、Semaphore信号量

- 广义上说，Semapore信号量是对锁的一种扩展；因为无论是内部锁synchronized，还是重入锁

ReentrantLock，一次都只允许一个线程访问某一资源，而信号量却可以**指定多个线程同时访问某一个资源**。

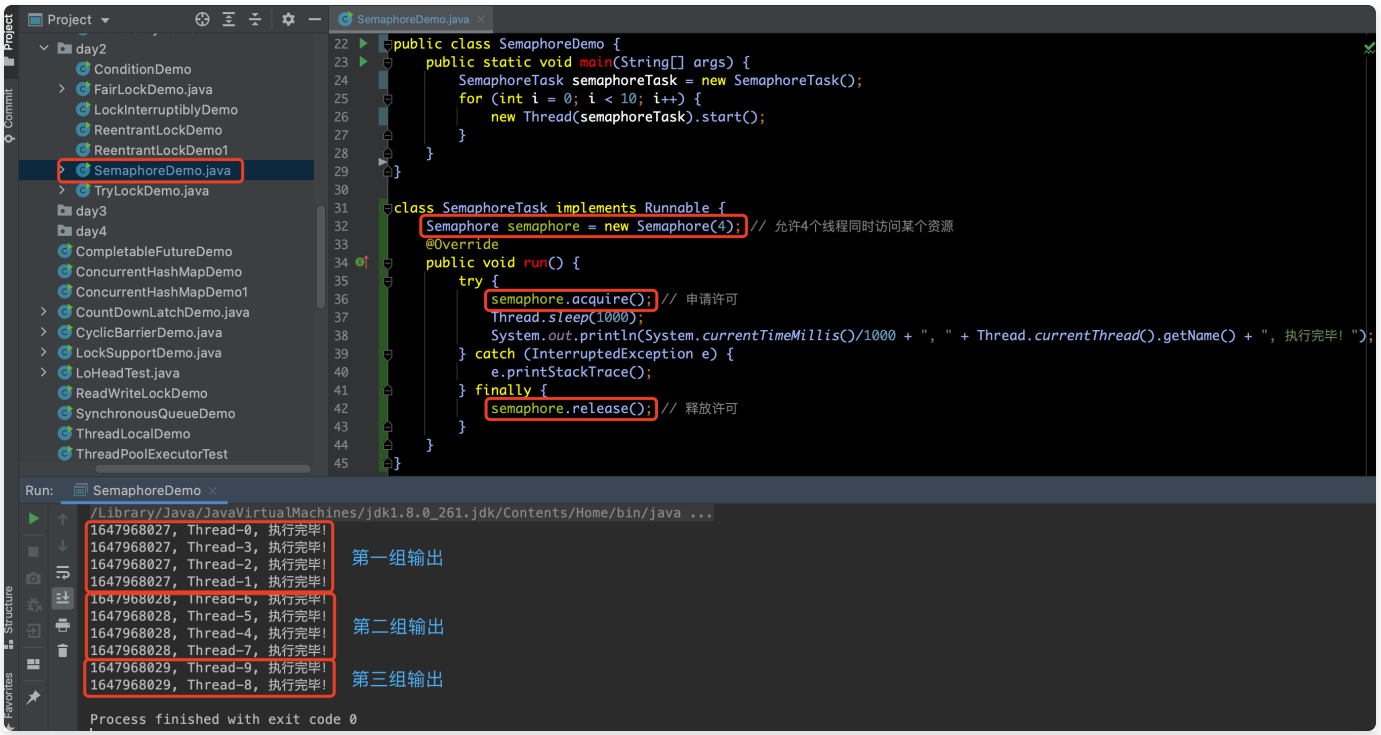
- 信息量主要提供了一下构造函数，必须要指定信号量的准入数，即：同时能申请多少个许可

```
public Semaphore(int permits); // permits：准入数
public Semaphore(int permits, boolean fair); // permits：准入数，fair：是否公平获得锁
```

- 信息量主要方法如下所示：

```
public void acquire();      尝试获得一个准入的许可。若无法获得，则线程会等待，直到有线程释放一个许可或者当前线程被中断。
public void acquireUninterruptibly();      具有acquire一样的功能，但是不响应中断。
public void tryAcquire();      尝试获得一个许可，如果成功就返回true，失败则返回false。
public void tryAcquire(long timeout, TimeUnit unit); 在指定时间内，尝试获得一个许可，如果成功就返回true，失败则返回false。
public void release();      资源访问结束后，释放一个许可。
```

- 申请了4个准入，循环10个线程，那么将会以4个线程一组为单位进行执行输出



四、ReadWriteLock读写锁

- ReadWriteLock是JDK5中提供的读写分离锁。它允许多个线程同时读。但是考虑到数据的完整性，写写操作和读写操作间依然是需要相互等待和持有锁的。读写锁的访问约束情况如下所示：

	读	写
读	非阻塞	阻塞
写	阻塞	阻塞

- 下面例子中，我们使用**普通锁**，执行读写操作：

ReadWriteLockDemo.java

```
public class ReadWriteLockDemo {
    private static Lock lock = new ReentrantLock();
    private static ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    private static Lock readLock = readWriteLock.readLock();
    private static Lock writeLock = readWriteLock.writeLock();

    public static void main(String[] args) {
        boolean openRWLock = false;
        /** 开启3个读操作线程 */
        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                if (openRWLock) {
                    execute(readLock, "开始执行【读】操作");
                } else {
                    execute(lock, "开始执行【读】操作");
                }
            }).start();
        }

        /** 开启3个写操作线程 */
        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                if (openRWLock) {
                    execute(writeLock, "开始执行【写】操作");
                } else {
                    execute(lock, "开始执行【写】操作");
                }
            }).start();
        }
    }

    public static void execute(Lock lock, String msg) {
        try {
            lock.lock();
            System.out.println(System.currentTimeMillis() / 1000 + " " + Thread.currentThread().getName() + ", " + msg);
            TimeUnit.MILLISECONDS.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

从ReentrantReadWriteLock中获得读锁和写锁

是否打开读写锁

读锁

普通锁

写锁

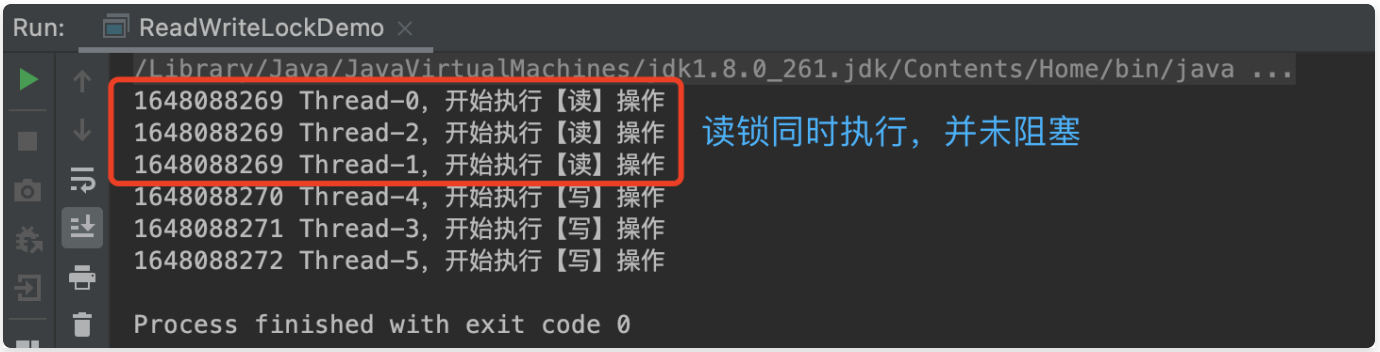
普通锁

```
Run:  ReadWriteLockDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1648087825 Thread-0, 开始执行【读】操作
1648087826 Thread-1, 开始执行【读】操作
1648087827 Thread-2, 开始执行【读】操作
1648087828 Thread-3, 开始执行【写】操作
1648087829 Thread-4, 开始执行【写】操作
1648087830 Thread-5, 开始执行【写】操作

Process finished with exit code 0
```

每隔1秒，执行一次

- 下面例子中，我们使用**读写锁**（只需要将上面例子中openRWLock=true即可）执行读写操作，如

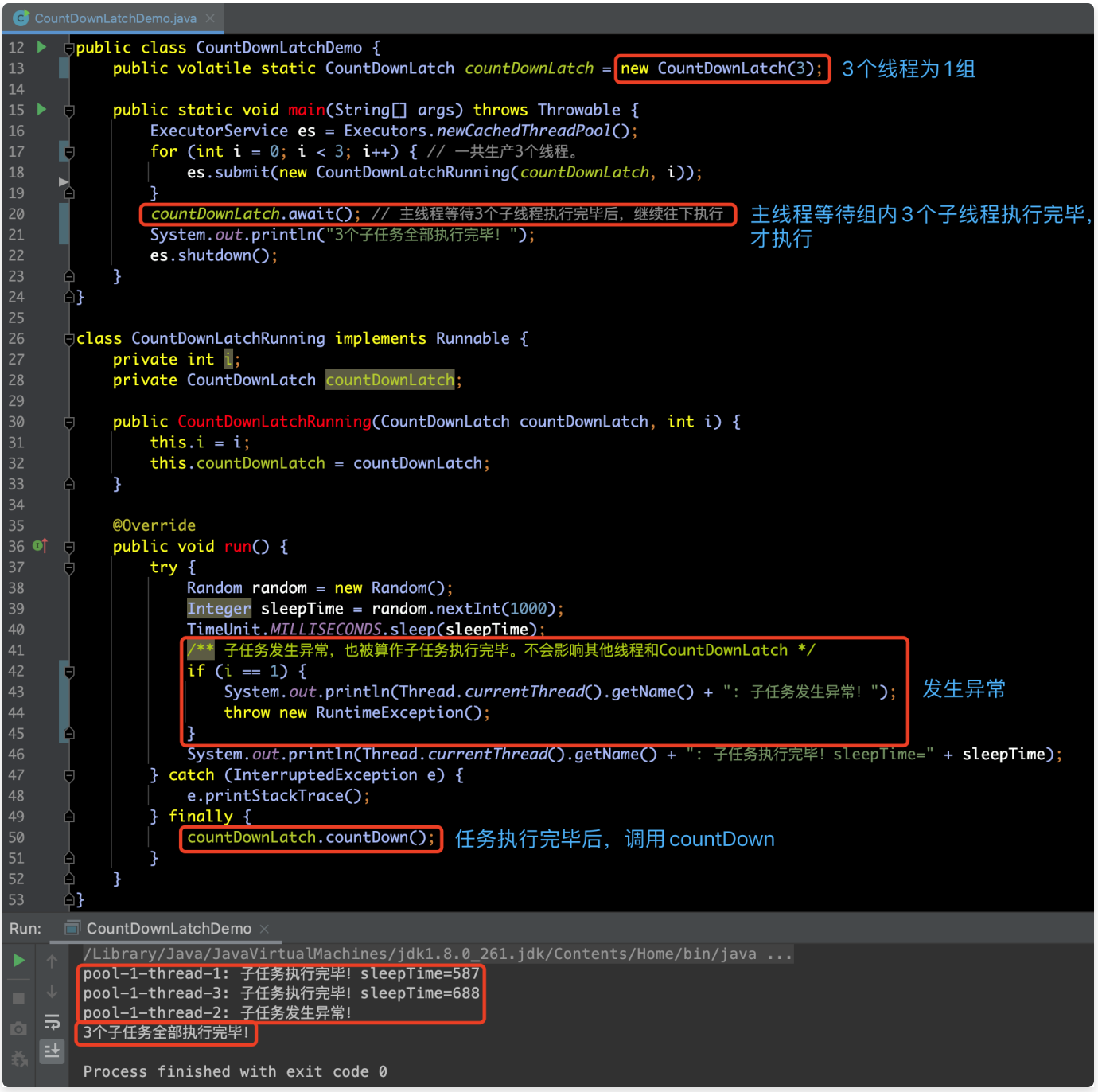
下所示：



- 所以，如果在系统中，读操作的次数远远大于写操作，那么读写锁就可以发挥最大的效果，提升系统的性能。

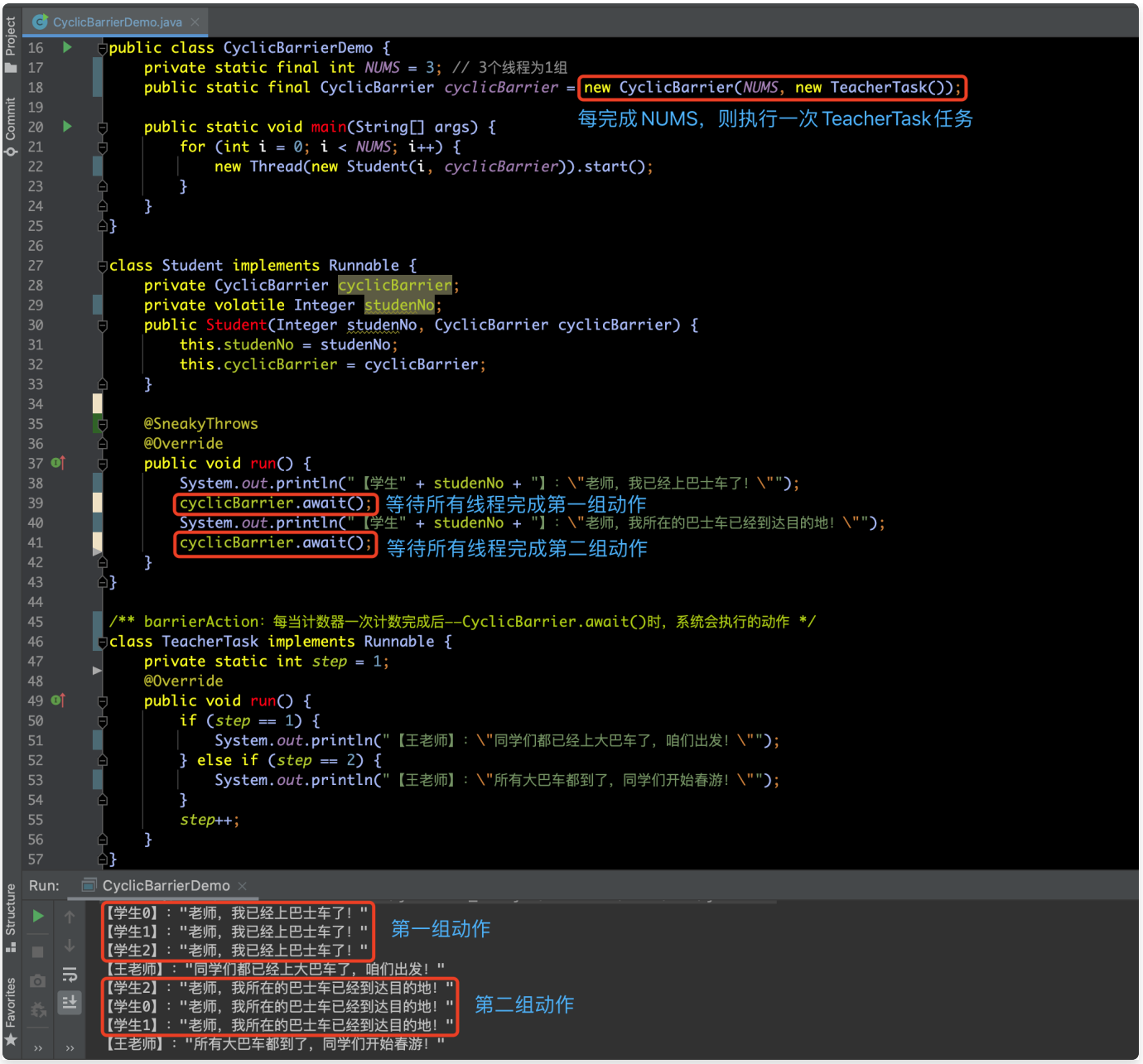
五、CountDownLatch倒计时器

- CountDownLatch是一个多线程控制工具。用来控制线程的等待。设置需要countDown的数量num，然后每一个线程执行完毕后，调用countDown()方法，而主线程调用await()方法执行等待，直到num个子线程执行了countDown()方法 ，则主线程开始继续执行。

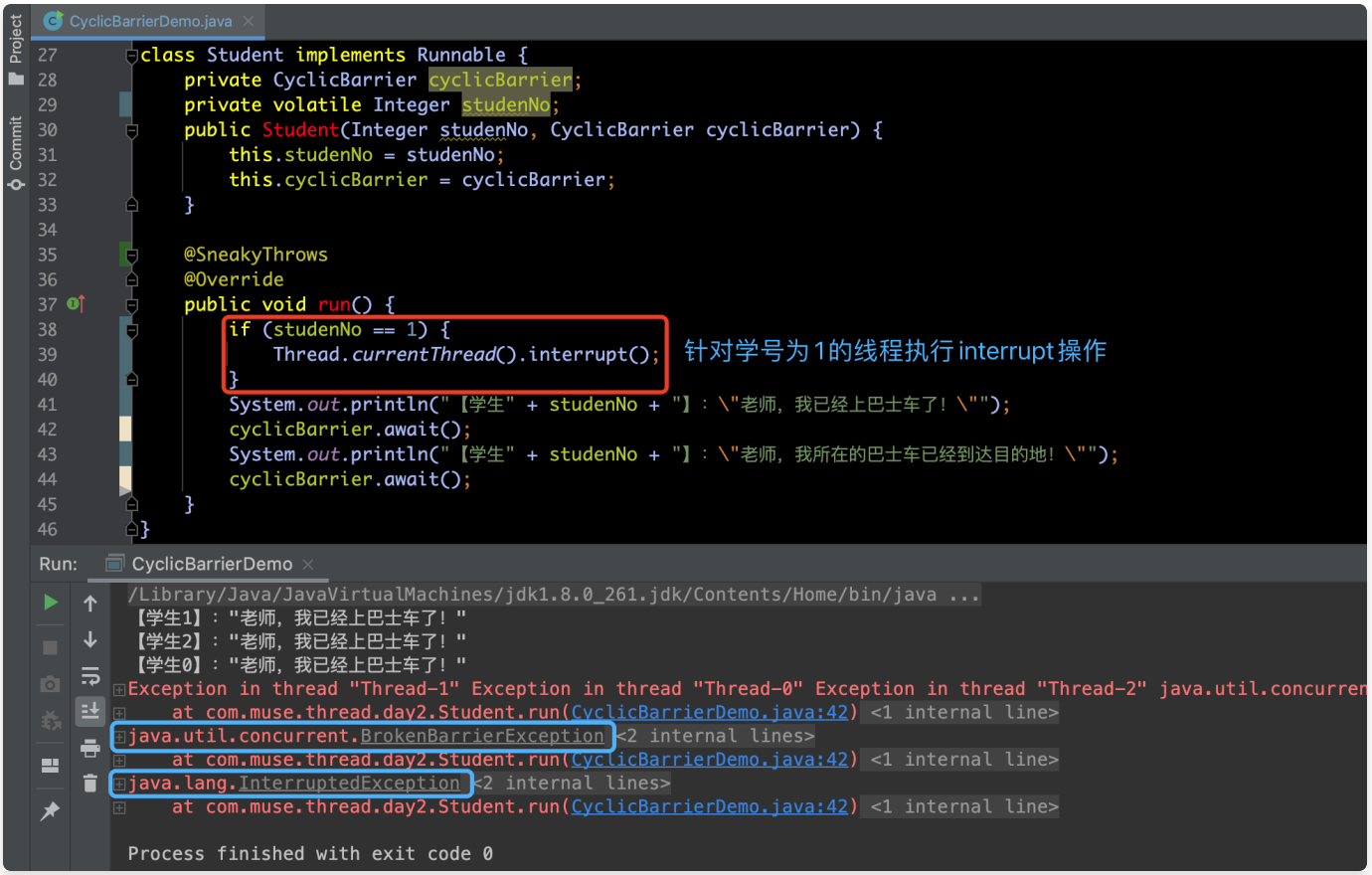


六、CyclicBarrier循环栅栏

- CyclicBarrier与CountDownLatch非常类似，它支持计数器的反复使用，CyclicBarrier可以理解为循环栅栏。CyclicBarrier可以接收一个参数作为Runnable barrierAction，每当计数器一次计数完成后——CyclicBarrier.await()时，系统会执行的动作。



- CyclicBarrier.await()方法可能会**抛出两种异常**：一个是**InterruptedException**，也就是在等待过程中，线程被中断，应该说这是一个非常通用的异常，大部分迫使线程等待的方法都可能会抛出这个异常，使得线程在等待时依然可以响应外部紧急事件。另外一个异常则是CyclicBarrier特有的**BrokenBarrierException**，一旦遇到这个异常，则表示当前的CyclicBarrier已经破损了，可能系统已经没有办法等待所有线程到齐了。如果继续等待，可能就是徒劳无功的，因此就此结束吧。

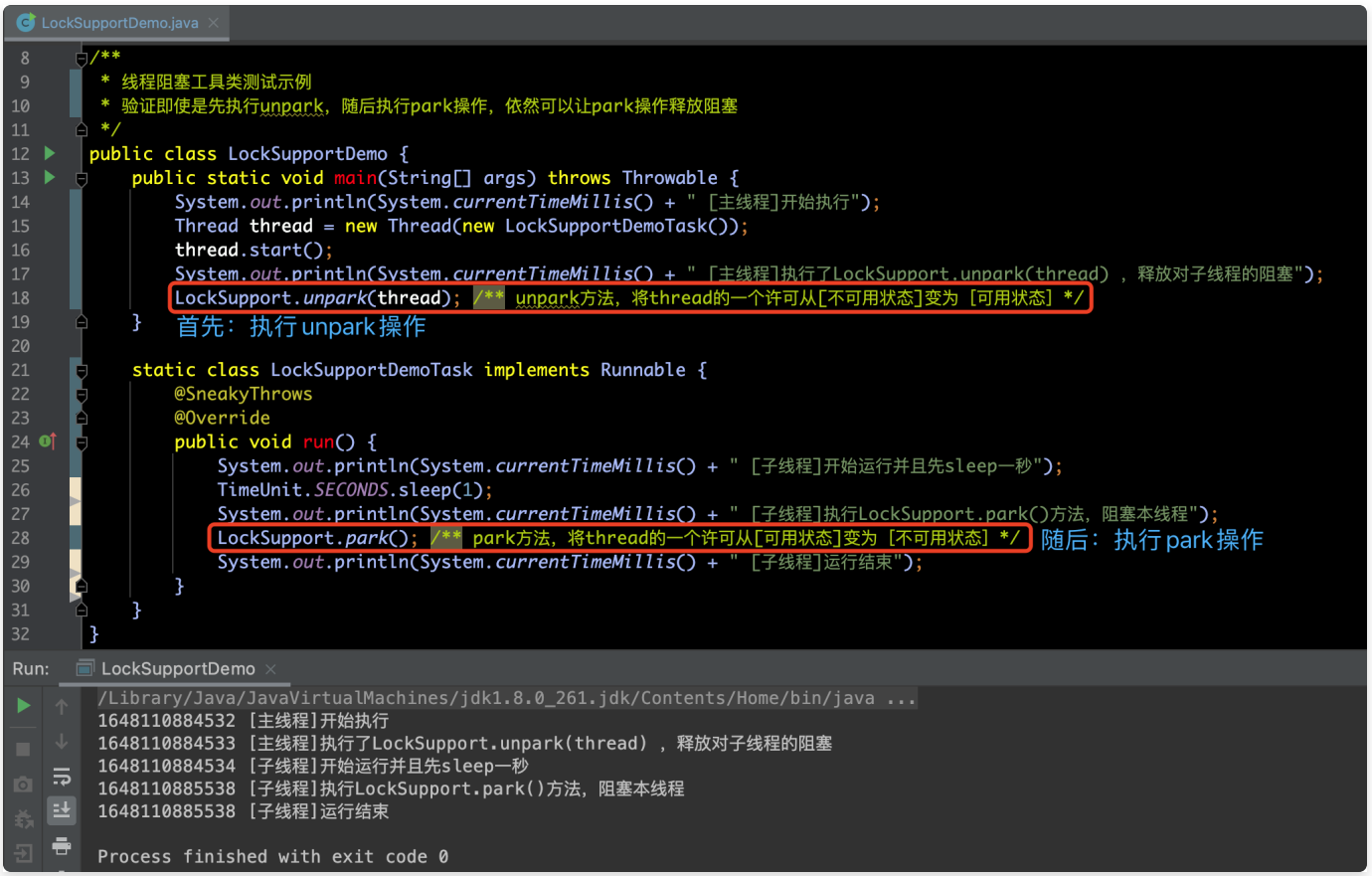


七、LockSupport线程阻塞工具类

- LockSupport是一个非常方便实用的**线程阻塞工具**，它可以在线程内任意位置让线程阻塞。和Thread.suspend()相比，它弥补了由于resume()在前发生，导致线程无法继续执行的情况。和Object.wait()方法相比，它**不需要先获得某个对象的锁**，也不会抛出InterruptedException异常。
- park()可以阻塞当前线程，其中每一个线程都有一个许可，该许可**默认为[不可用]**。如果该许可是[可用]状态，那么park()方法会立即返回，消费这个许可，将该许可**变更为[不可用]**状态，流程代码可以继续执行。**如果该许可是[不可用]状态，那么park()方法将会阻塞**；unpark方法，将指定线程的一个许可**变为[可用]**状态。如下表所示：

	当前许可状态	调用后许可状态
park()	可用	将可用变为不可用
	不可用	阻塞
unpark()	可用	不变化
	不可用	变为可用

- 示例一：先执行unpark()方法再执行park()方法，也不会造成永久卡死线程。如下所示：



- 示例二：LockSupport.park()还能支持中断。但是它**不会抛InterruptedException异常**。它只会默默的返回，但是我们可以从Thread.interrupted()等方法获得中断标记。

LockSupportDemo1.java

```java
public class LockSupportDemo1 {
    public static void main(String[] args) throws Throwable {
        RunningDemo runningDemo = new RunningDemo();
        Thread thread1 = new Thread(runningDemo, "T1");
        Thread thread2 = new Thread(runningDemo, "T2");
        thread1.start();
        thread2.start();
        TimeUnit.MILLISECONDS.sleep(100);
        System.out.println("[主线程]执行interrupt()打断子线程" + thread1.getName());
        thread1.interrupt(); 打断T1的park阻塞状态，不抛异常
        TimeUnit.MILLISECONDS.sleep(100);
        LockSupport.unpark(thread2); /** unpark方法，将thread1的一个许可变为［可用］状态*/通过unpark打断T2的park阻塞状态
        System.out.println("[主线程]针对子线程" + thread2.getName() + "执行LockSupport.unpark() 完毕！");
    }

    static class RunningDemo implements Runnable {
        private final Object object = new Object();
        @Override
        public void run() {
            synchronized(object) {
                String threadName = Thread.currentThread().getName();
                System.out.println("[子线程]" + threadName + "开始运行！执行LockSupport.park()方法，阻塞本线程！");
                /**
                 * park()可以阻塞当前线程
                 * 每一个线程有一个许可，该许可默认为[不可用]。
                 * 如果该许可是[可用]状态，那么park()方法会立即返回，消费这个许可，将该许可消变更为[不可用]状态，流程代码可以继续执行。
                 */
                LockSupport.park(); 执行线程阻塞

                /** 也可以使用Thread.currentThread().isInterrupted() */  判断本线程是否被执行了interrupt方法
                if (Thread.interrupted()) {
                    System.out.println("[子线程]" + threadName + "被interrupt()方法打断，但并不会抛出异常。");
                }
                System.out.println("[子线程]" + threadName + "运行结束！");
            }
        }
    }
}
```

Run: LockSupportDemo1

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
[子线程]T1开始运行！执行LockSupport.park()方法，阻塞本线程！
[主线程]执行interrupt()打断子线程T1
[子线程]T1被interrupt()方法打断，但并不会抛出异常。
[子线程]T1运行结束！
[子线程]T2开始运行！执行LockSupport.park()方法，阻塞本线程！
[主线程]针对子线程T2执行LockSupport.unpark() 完毕！
[子线程]T2运行结束！

Process finished with exit code 0
```