

【讲义】 第2讲：AQS和JUC

一、ReentrantLock重入锁

1.1> 概述

1.2> 中断响应 lockInterruptibly()

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

1.4> 公平锁和非公平锁

1.5> AQS源码解析

二、Condition重入锁的搭配类

三、Semaphore信号量

四、ReadWriteLock读写锁

五、CountDownLatch倒计时器

六、CyclicBarrier循环栅栏

七、LockSupport线程阻塞工具类

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(): 判断当前线程是否持有锁。

void unlock(): 释放锁。

- 下面是使用重入锁的简单示例：

```
10 public class ReentrantLockDemo {
11     public static int num;
12
13     @SneakyThrows
14     public static void main(String[] args) {
15         CountTask task = new CountTask();
16         Thread t1 = new Thread(task);
17         Thread t2 = new Thread(task);
18         t1.start();
19         t2.start();
20         t1.join();
21         t2.join();
22         System.out.println("num = " + num);
23     }
24
25     static class CountTask implements Runnable {
26         public ReentrantLock lock = new ReentrantLock();
27         @Override
28         public void run() {
29             System.out.println(Thread.currentThread().getName() + " start!");
30             for (int i = 0; i < 10000000; i++) {
31                 lock.lock();
32                 try {
33                     num++;
34                 } finally {
35                     lock.unlock();
36                 }
37             }
38             System.out.println(Thread.currentThread().getName() + " end!");
39         }
40     }
41 }
```

Run: ReentrantLockDemo x

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...

Thread-0 start!
Thread-1 start!
Thread-1 end!
Thread-0 end!
num = 20000000

- 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：

The screenshot shows an IDE with a Java file named `ReentrantLockDemo1.java`. The code defines a `ReentrantLockDemo1` class with a `main` method that starts a new thread with a `Task` object. The `Task` class implements `Runnable` and uses a `ReentrantLock` object. In the `run` method, it locks the lock, prints "is lock!", locks it again, prints "is lock!", then enters a `try` block where it prints "is coming!". The `finally` block contains two `unlock` calls, each followed by a print statement "is unlock!". The second `unlock` call and its corresponding print statement are highlighted with a red box. The output window shows the execution results: "Thread-0 is lock!", "Thread-0 is lock!", "Thread-0 is coming!", "Thread-0 is unlock!", and "Thread-0 is unlock!". A red box highlights an exception in the output: `Exception in thread "Thread-0" java.lang.IllegalMonitorStateException <3 internal lines> at com.muse.thread.day2.ReentrantLockDemo1$Task.run(ReentrantLockDemo1.java:26) <1 internal line>`. The process finished with exit code 0.

```
5 public class ReentrantLockDemo1 {
6     public static void main(String[] args) {
7         new Thread(new Task()).start();
8     }
9
10    static class Task implements Runnable {
11        public ReentrantLock lock = new ReentrantLock();
12
13        @Override
14        public void run() {
15            lock.lock();
16            System.out.println(Thread.currentThread().getName() + " is lock!");
17            lock.lock();
18            System.out.println(Thread.currentThread().getName() + " is lock!");
19            try {
20                System.out.println(Thread.currentThread().getName() + " is coming!");
21            } finally {
22                lock.unlock();
23                System.out.println(Thread.currentThread().getName() + " is unlock!");
24                lock.unlock();
25                System.out.println(Thread.currentThread().getName() + " is unlock!");
26                lock.unlock();
27                System.out.println(Thread.currentThread().getName() + " is unlock!");
28            }
29        }
30    }
31 }
```

Run: ReentrantLockDemo1

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...

Thread-0 is lock!
Thread-0 is lock!
Thread-0 is coming!
Thread-0 is unlock!
Thread-0 is unlock!

Exception in thread "Thread-0" java.lang.IllegalMonitorStateException <3 internal lines>
at com.muse.thread.day2.ReentrantLockDemo1\$Task.run(ReentrantLockDemo1.java:26) <1 internal line>

Process finished with exit code 0

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：

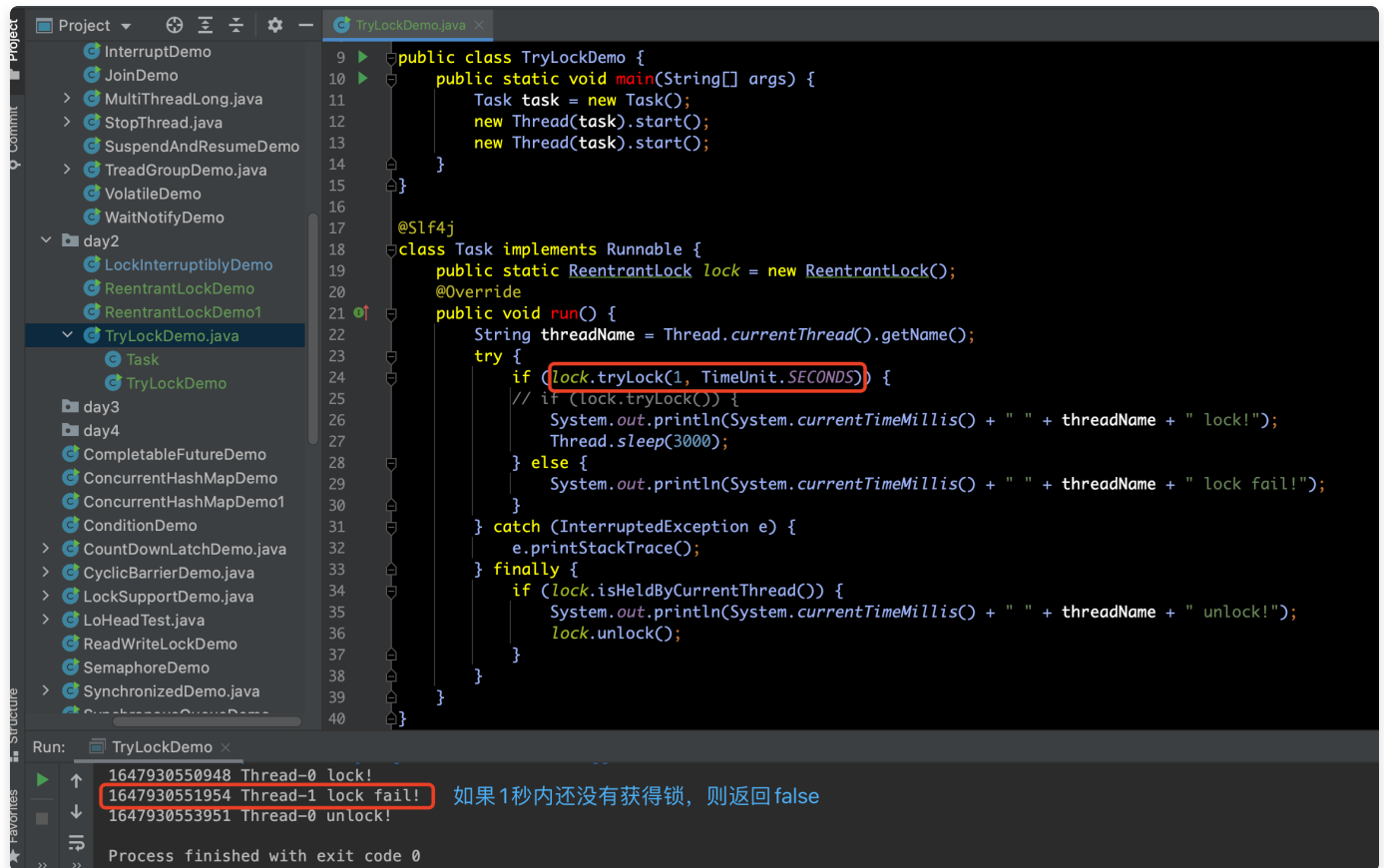
```
LockInterruptiblyDemo.java x
11 public class LockInterruptiblyDemo {
12     private static ReentrantLock lock1 = new ReentrantLock();
13     private static ReentrantLock lock2 = new ReentrantLock();
14
15     public static void main(String[] args) throws Throwable {
16         /** 可中断锁 */
17         Thread t1 = new Thread(new ReentrantLockThread(lock1, lock2)); 入参相反，可以导致死锁
18         Thread t2 = new Thread(new ReentrantLockThread(lock2, lock1));
19         t1.start();
20         t2.start();
21         TimeUnit.MILLISECONDS.sleep(100);
22         System.out.println("主线程开始沉睡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
23         TimeUnit.MILLISECONDS.sleep(1000);
24         System.out.println("主线程线程，在" + t1.getName() + "上开始执行interrupt()");
25         t1.interrupt(); 中断锁
26     }
27
28     /** ReentrantLock的lockInterruptibly实现死锁 */
29     static class ReentrantLockThread implements Runnable {
30         private ReentrantLock lock1, lock2;
31
32         public ReentrantLockThread(ReentrantLock lock1, ReentrantLock lock2) {
33             this.lock1 = lock1;
34             this.lock2 = lock2;
35         }
36         @Override
37         public void run() {
38             try {
39                 对lock1加锁 lock1.lockInterruptibly(); // 获得lock1的可中断锁
40                 System.out.println(Thread.currentThread().getName() + ", 加锁成功1-2!");
41                 TimeUnit.MILLISECONDS.sleep(100); // 等待lock1和lock2分别被两个线程获取。产生死锁现象
42                 对lock2加锁 lock2.lockInterruptibly(); // 获得lock2的可中断锁
43                 System.out.println(Thread.currentThread().getName() + ", 加锁成功2-2!");
44             } catch (InterruptedException e) {
45                 System.out.println(Thread.currentThread().getName() + ", 发生异常!");
46                 e.printStackTrace();
47             } finally {
48                 if (lock1.isHeldByCurrentThread()) {
49                     lock1.unlock(); 如果持有锁，则进行解锁
50                 }
51                 if (lock2.isHeldByCurrentThread()) {
52                     lock2.unlock();
53                 }
54                 System.out.println(Thread.currentThread().getName() + ", 解锁成功!");
55             }
56         }
57     }
58 }
```

```
Run: LockInterruptiblyDemo x
/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。



The screenshot shows an IDE with a project structure on the left and a code editor in the center. The code editor displays the following Java code:

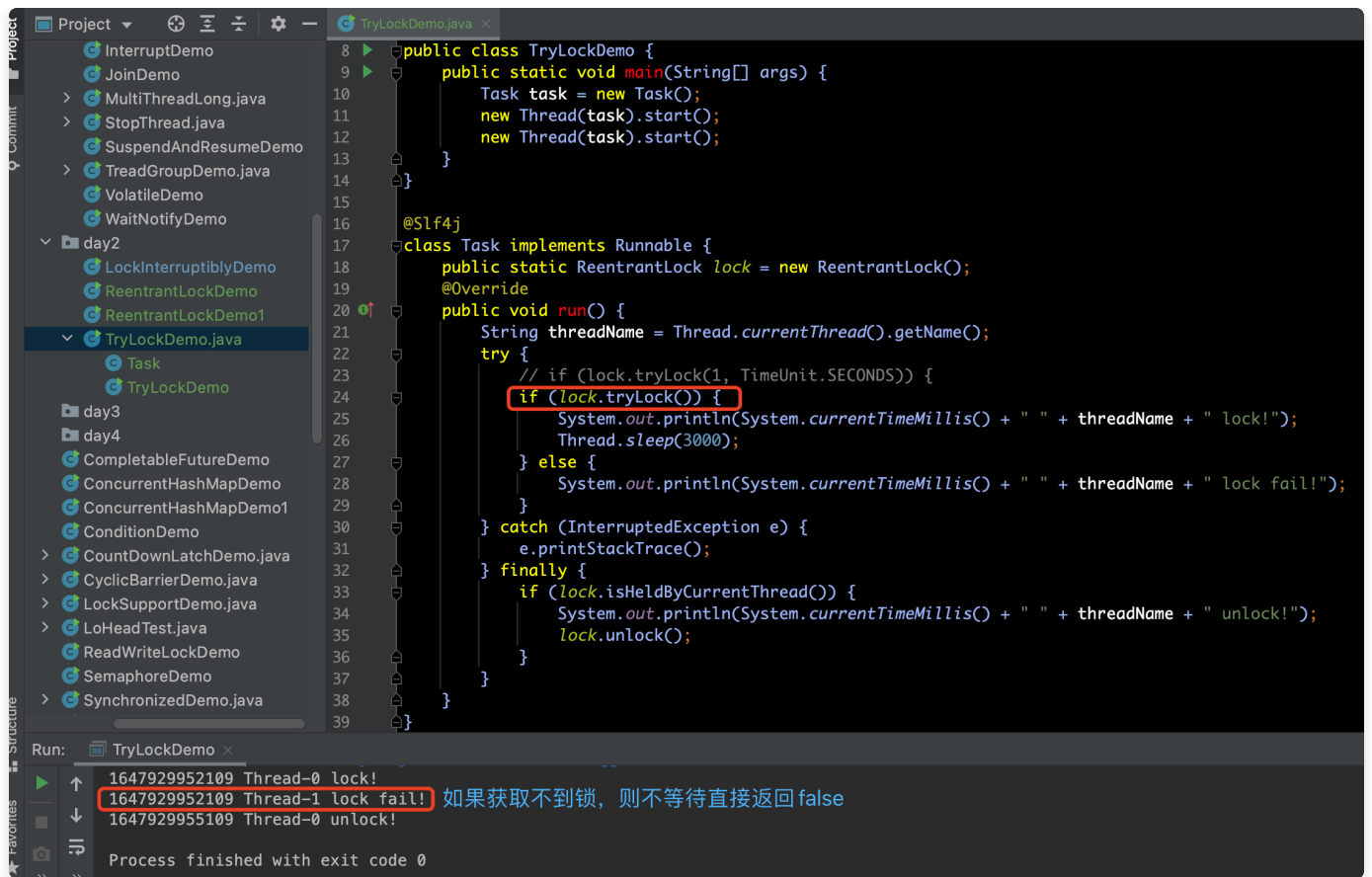
```
9 public class TryLockDemo {
10     public static void main(String[] args) {
11         Task task = new Task();
12         new Thread(task).start();
13         new Thread(task).start();
14     }
15 }
16
17 @Slf4j
18 class Task implements Runnable {
19     public static ReentrantLock lock = new ReentrantLock();
20     @Override
21     public void run() {
22         String threadName = Thread.currentThread().getName();
23         try {
24             if (lock.tryLock(1, TimeUnit.SECONDS)) {
25                 // if (lock.tryLock()) {
26                 System.out.println(System.currentTimeMillis() + " " + threadName + " lock!");
27                 Thread.sleep(3000);
28             } else {
29                 System.out.println(System.currentTimeMillis() + " " + threadName + " lock fail!");
30             }
31         } catch (InterruptedException e) {
32             e.printStackTrace();
33         } finally {
34             if (lock.isHeldByCurrentThread()) {
35                 System.out.println(System.currentTimeMillis() + " " + threadName + " unlock!");
36                 lock.unlock();
37             }
38         }
39     }
40 }
```

The output window at the bottom shows the following log messages:

```
Run: TryLockDemo
1647930550948 Thread-0 lock!
1647930551954 Thread-1 lock fail!
1647930553951 Thread-0 unlock!
Process finished with exit code 0
```

Red boxes highlight the `lock.tryLock(1, TimeUnit.SECONDS)` call in the code and the `lock fail!` message in the output. A note next to the output message states: "如果1秒内还没有获得锁，则返回 false" (If the lock is not obtained within 1 second, return false).

- 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。

```
10 public class FairLockDemo {
11     public static void main(String[] args) {
12         FairLockTask fairLockTask = new FairLockTask();
13         new Thread(fairLockTask, "T1").start();
14         new Thread(fairLockTask, "T2").start();
15         new Thread(fairLockTask, "T3").start();
16         new Thread(fairLockTask, "T4").start();
17         new Thread(fairLockTask, "T5").start();
18         new Thread(fairLockTask, "T6").start();
19     }
20 }
21
22 class FairLockTask implements Runnable {
23     // ReentrantLock lock = new ReentrantLock(true); // 开启公平锁
24     ReentrantLock lock = new ReentrantLock(false); // 开启非公平锁
25     @SneakyThrows
26     @Override
27     public void run() {
28         try {
29             lock.lock();
30             System.out.println(Thread.currentThread().getName() + " lock!");
31         } finally {
32             lock.unlock();
33         }
34     }
35 }
```

Run: FairLockDemo x

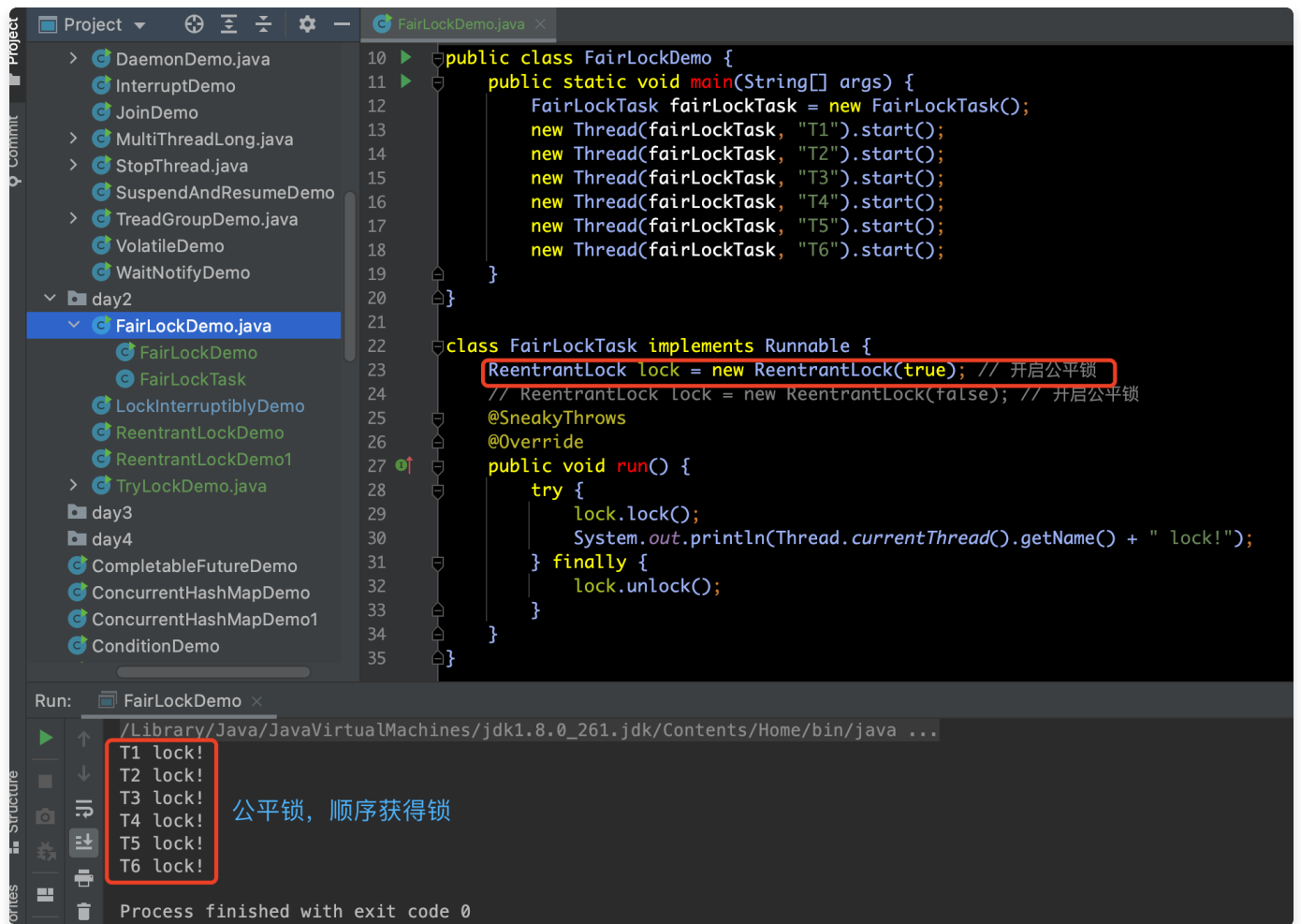
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...

T1 lock!
T5 lock!
T2 lock!
T6 lock!
T3 lock!
T4 lock!

非公平锁，乱序获得锁

Process finished with exit code 0

- 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。

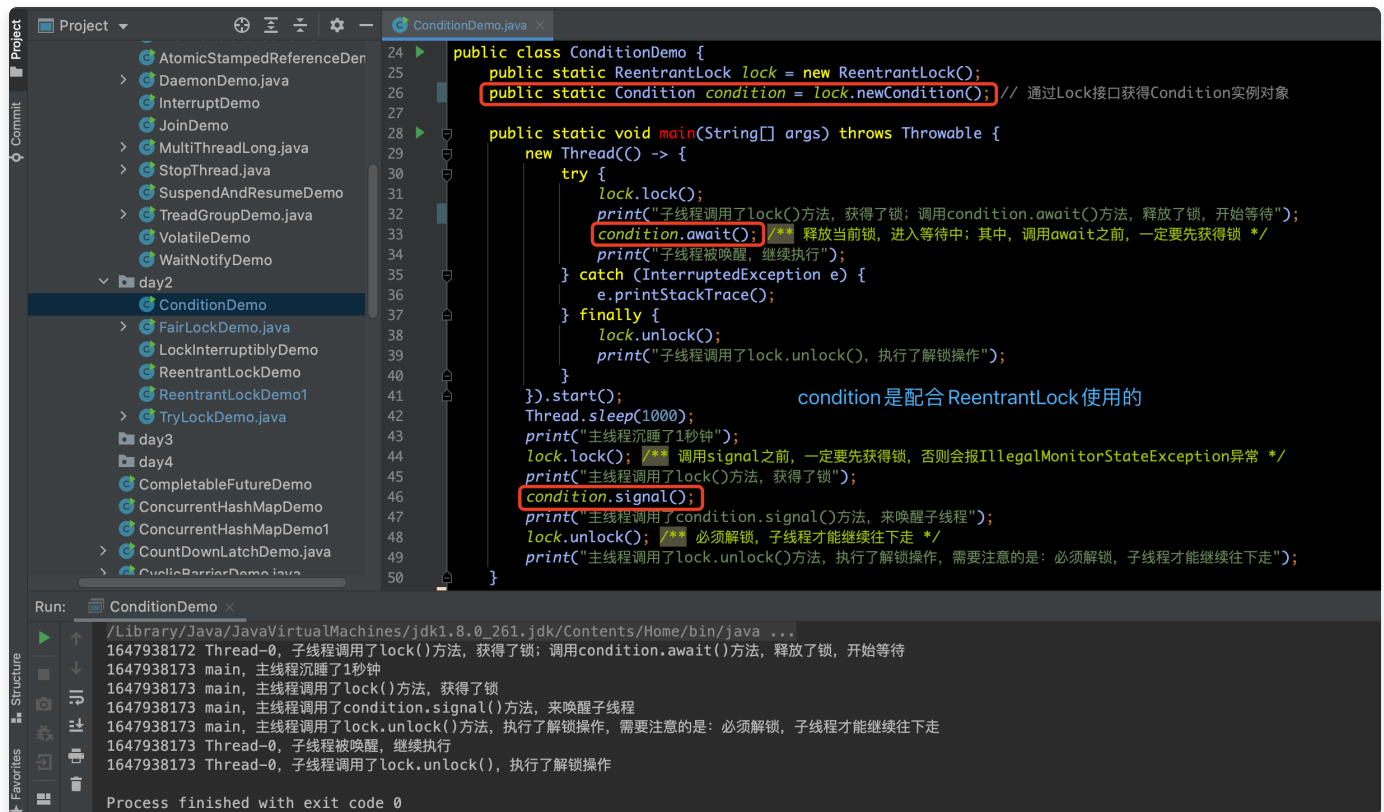
void signal():

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:



```
24 public class ConditionDemo {
25     public static ReentrantLock lock = new ReentrantLock();
26     public static Condition condition = lock.newCondition(); // 通过Lock接口获得Condition实例对象
27
28     public static void main(String[] args) throws Throwable {
29         new Thread(() -> {
30             try {
31                 lock.lock();
32                 print("子线程调用了lock()方法，获得了锁；调用condition.await()方法，释放了锁，开始等待");
33                 condition.await(); /** 释放当前锁，进入等待中；其中，调用await之前，一定要先获得锁 */
34                 print("子线程被唤醒，继续执行");
35             } catch (InterruptedException e) {
36                 e.printStackTrace();
37             } finally {
38                 lock.unlock();
39                 print("子线程调用了lock.unlock()，执行了解锁操作");
40             }
41         }).start();
42         Thread.sleep(1000);
43         print("主线程沉睡了1秒钟");
44         lock.lock(); /** 调用signal之前，一定要先获得锁，否则会报IllegalMonitorStateException异常 */
45         print("主线程调用了lock()方法，获得了锁");
46         condition.signal();
47         print("主线程调用了Condition.signal()方法，来唤醒子线程");
48         lock.unlock(); /** 必须解锁，子线程才能继续往下走 */
49         print("主线程调用了lock.unlock()方法，执行了解锁操作，需要注意的是：必须解锁，子线程才能继续往下走");
50     }
51 }
```

Run: ConditionDemo x

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...

1647938172 Thread-0, 子线程调用了lock()方法，获得了锁；调用condition.await()方法，释放了锁，开始等待

1647938173 main, 主线程沉睡了1秒钟

1647938173 main, 主线程调用了lock()方法，获得了锁

1647938173 main, 主线程调用了condition.signal()方法，来唤醒子线程

1647938173 main, 主线程调用了lock.unlock()方法，执行了解锁操作，需要注意的是：必须解锁，子线程才能继续往下走

1647938173 Thread-0, 子线程被唤醒，继续执行

1647938173 Thread-0, 子线程调用了lock.unlock()，执行了解锁操作

Process finished with exit code 0

三、Semaphore信号量

- 广义上说，Semaphore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。
```

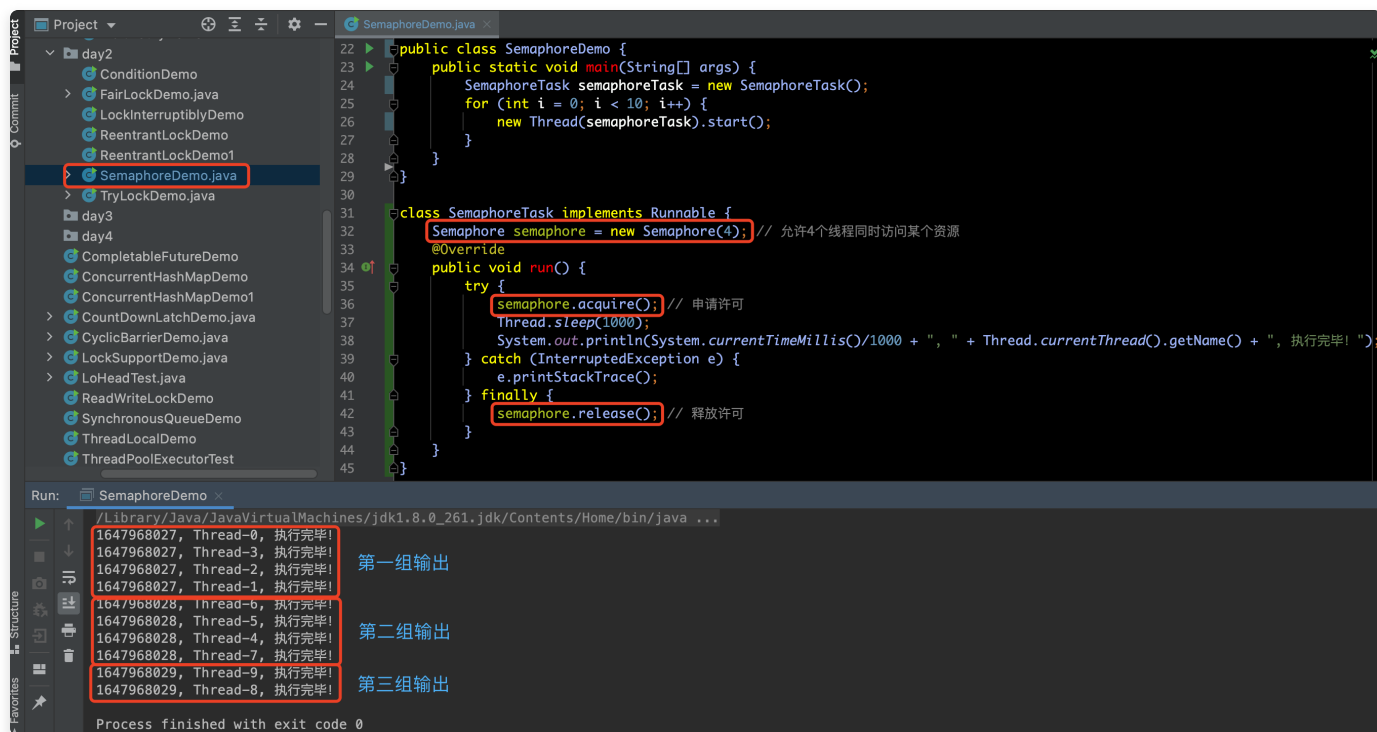
```
public void acquireUninterruptibly();    具有acquire一样的功能，但是不响应中断。
```

```
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



```
public class SemaphoreDemo {
    public static void main(String[] args) {
        SemaphoreTask semaphoreTask = new SemaphoreTask();
        for (int i = 0; i < 10; i++) {
            new Thread(semaphoreTask).start();
        }
    }
}

class SemaphoreTask implements Runnable {
    Semaphore semaphore = new Semaphore(4); // 允许4个线程同时访问某个资源
    @Override
    public void run() {
        try {
            semaphore.acquire(); // 申请许可
            Thread.sleep(1000);
            System.out.println(System.currentTimeMillis()/1000 + ", " + Thread.currentThread().getName() + ", 执行完毕!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release(); // 释放许可
        }
    }
}
```

Run: SemaphoreDemo

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1647968027, Thread-0, 执行完毕!
1647968027, Thread-3, 执行完毕!
1647968027, Thread-2, 执行完毕!
1647968027, Thread-1, 执行完毕!
1647968028, Thread-6, 执行完毕!
1647968028, Thread-5, 执行完毕!
1647968028, Thread-4, 执行完毕!
1647968028, Thread-7, 执行完毕!
1647968029, Thread-9, 执行完毕!
1647968029, Thread-8, 执行完毕!
```

第一组输出

第二组输出

第三组输出

Process finished with exit code 0

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：

```

12 public class ReadWriteLockDemo {
13     private static Lock lock = new ReentrantLock();
14     private static ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
15     private static Lock readLock = readWriteLock.readLock();
16     private static Lock writeLock = readWriteLock.writeLock();
17
18     public static void main(String[] args) {
19         boolean openRWLock = false;
20         /** 开启3个读操作线程 */
21         for (int i = 0; i < 3; i++) {
22             new Thread() -> {
23                 if (openRWLock) {
24                     execute(readLock, "开始执行【读】操作");
25                 } else {
26                     execute(lock, "开始执行【读】操作");
27                 }
28             }.start();
29         }
30
31         /** 开启3个写操作线程 */
32         for (int i = 0; i < 3; i++) {
33             new Thread() -> {
34                 if (openRWLock) {
35                     execute(writeLock, "开始执行【写】操作");
36                 } else {
37                     execute(lock, "开始执行【写】操作");
38                 }
39             }.start();
40         }
41     }
42
43     @
44     public static void execute(Lock lock, String msg) {
45         try {
46             lock.lock();
47             System.out.println(System.currentTimeMillis() / 1000 + " " + Thread.currentThread().getName() + ", " + msg);
48             TimeUnit.MILLISECONDS.sleep(1000);
49         } catch (InterruptedException e) {
50             e.printStackTrace();
51         } finally {
52             lock.unlock();
53         }
54     }
55 }

```

```

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：

The screenshot shows an IDE with two windows. The top window, titled 'ReadWriteLockDemo.java', contains the following Java code:

```
12 public class ReadWriteLockDemo {
13     private static Lock lock = new ReentrantLock();
14     private static ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
15     private static Lock readLock = readWriteLock.readLock();
16     private static Lock writeLock = readWriteLock.writeLock();
17
18     public static void main(String[] args) {
19         boolean openRWLock = true; 开启读写锁
20         /** 开启3个读操作线程 */
```

The bottom window, titled 'Run: ReadWriteLockDemo', shows the execution output:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1648088269 Thread-0, 开始执行【读】操作
1648088269 Thread-2, 开始执行【读】操作
1648088269 Thread-1, 开始执行【读】操作
1648088270 Thread-4, 开始执行【写】操作
1648088271 Thread-3, 开始执行【写】操作
1648088272 Thread-5, 开始执行【写】操作
Process finished with exit code 0
```

Annotations in the image include a red box around the line `boolean openRWLock = true;` in the code and a red box around the first three lines of the output. A blue text annotation '读锁同时执行，并未阻塞' is placed to the right of the first three lines of the output.

- 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。

```
CountDownLatchDemo.java x
12 public class CountDownLatchDemo {
13     public volatile static CountDownLatch countDownLatch = new CountDownLatch(3); 3个线程为1组
14
15     public static void main(String[] args) throws Throwable {
16         ExecutorService es = Executors.newCachedThreadPool();
17         for (int i = 0; i < 3; i++) { // 一共生产3个线程。
18             es.submit(new CountDownLatchRunning(countDownLatch, i));
19         }
20         countDownLatch.await(); // 主线程等待3个子线程执行完毕后，继续往下执行 主线程等待组内3个子线程执行完毕，才执行
21         System.out.println("3个子任务全部执行完毕!");
22         es.shutdown();
23     }
24 }
25
26 class CountDownLatchRunning implements Runnable {
27     private int i;
28     private CountDownLatch countDownLatch;
29
30     public CountDownLatchRunning(CountDownLatch countDownLatch, int i) {
31         this.i = i;
32         this.countDownLatch = countDownLatch;
33     }
34
35     @Override
36     public void run() {
37         try {
38             Random random = new Random();
39             Integer sleepTime = random.nextInt(1000);
40             TimeUnit.MILLISECONDS.sleep(sleepTime);
41             /** 子任务发生异常，也被算作子任务执行完毕。不会影响其他线程和CountDownLatch */
42             if (i == 1) {
43                 System.out.println(Thread.currentThread().getName() + ": 子任务发生异常!"); 发生异常
44                 throw new RuntimeException();
45             }
46             System.out.println(Thread.currentThread().getName() + ": 子任务执行完毕! sleepTime=" + sleepTime);
47         } catch (InterruptedException e) {
48             e.printStackTrace();
49         } finally {
50             countDownLatch.countDown(); 任务执行完毕后，调用 countDown
51         }
52     }
53 }
```

Run: CountDownLatchDemo x

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
pool-1-thread-1: 子任务执行完毕! sleepTime=587
pool-1-thread-3: 子任务执行完毕! sleepTime=688
pool-1-thread-2: 子任务发生异常!
3个子任务全部执行完毕!
Process finished with exit code 0
```

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。

```
CyclicBarrierDemo.java
16 public class CyclicBarrierDemo {
17     private static final int NUMS = 3; // 3个线程为1组
18     public static final CyclicBarrier cyclicBarrier = new CyclicBarrier(NUMS, new TeacherTask());
19
20     public static void main(String[] args) {
21         for (int i = 0; i < NUMS; i++) {
22             new Thread(new Student(i, cyclicBarrier)).start();
23         }
24     }
25
26     class Student implements Runnable {
27         private CyclicBarrier cyclicBarrier;
28         private volatile Integer studenNo;
29         public Student(Integer studenNo, CyclicBarrier cyclicBarrier) {
30             this.studenNo = studenNo;
31             this.cyclicBarrier = cyclicBarrier;
32         }
33
34         @SneakyThrows
35         @Override
36         public void run() {
37             System.out.println("【学生" + studenNo + "】：\"老师，我已经上巴士车了！\"");
38             cyclicBarrier.await(); // 等待所有线程完成第一组动作
39             System.out.println("【学生" + studenNo + "】：\"老师，我所在的巴士车已经到达目的地！\"");
40             cyclicBarrier.await(); // 等待所有线程完成第二组动作
41         }
42     }
43
44     /** barrierAction: 每当计数器一次计数完成后--CyclicBarrier.await()时，系统会执行的动作 */
45     class TeacherTask implements Runnable {
46         private static int step = 1;
47         @Override
48         public void run() {
49             if (step == 1) {
50                 System.out.println("【王老师】：\"同学们都已经上大巴士了，咱们出发！\"");
51             } else if (step == 2) {
52                 System.out.println("【王老师】：\"所有大巴车都到了，同学们开始春游！\"");
53             }
54             step++;
55         }
56     }
57 }
```

Run: CyclicBarrierDemo

```
【学生0】：\"老师，我已经上巴士车了！\"
【学生1】：\"老师，我已经上巴士车了！\"
【学生2】：\"老师，我已经上巴士车了！\"
【王老师】：\"同学们都已经上大巴士了，咱们出发！\"
【学生2】：\"老师，我所在的巴士车已经到达目的地！\"
【学生0】：\"老师，我所在的巴士车已经到达目的地！\"
【学生1】：\"老师，我所在的巴士车已经到达目的地！\"
【王老师】：\"所有大巴车都到了，同学们开始春游！\"
```

- 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。

```

27 class Student implements Runnable {
28     private CyclicBarrier cyclicBarrier;
29     private volatile Integer studenNo;
30     public Student(Integer studenNo, CyclicBarrier cyclicBarrier) {
31         this.studenNo = studenNo;
32         this.cyclicBarrier = cyclicBarrier;
33     }
34
35     @SneakyThrows
36     @Override
37     public void run() {
38         if (studenNo == 1) {
39             Thread.currentThread().interrupt(); 针对学号为1的线程执行interrupt操作
40         }
41         System.out.println("【学生" + studenNo + "】: \"老师, 我已经上巴士车了! \");
42         cyclicBarrier.await();
43         System.out.println("【学生" + studenNo + "】: \"老师, 我所在的巴士车已经到达目的地! \");
44         cyclicBarrier.await();
45     }
46 }

```

Run: CyclicBarrierDemo

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
【学生1】: "老师, 我已经上巴士车了! "
【学生2】: "老师, 我已经上巴士车了! "
【学生0】: "老师, 我已经上巴士车了! "
Exception in thread "Thread-1" Exception in thread "Thread-0" Exception in thread "Thread-2" java.util.concurrent
at com.muse.thread.day2.Student.run(CyclicBarrierDemo.java:42) <1 internal line>
java.util.concurrent.BrokenBarrierException <2 internal lines>
at com.muse.thread.day2.Student.run(CyclicBarrierDemo.java:42) <1 internal line>
java.lang.InterruptedException <2 internal lines>
at com.muse.thread.day2.Student.run(CyclicBarrierDemo.java:42) <1 internal line>
Process finished with exit code 0

```

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：

```
LockSupportDemo.java
8  /**
9   * 线程阻塞工具类测试示例
10  * 验证即使是先执行unpark, 随后执行park操作, 依然可以让park操作释放阻塞
11  */
12  public class LockSupportDemo {
13      public static void main(String[] args) throws Throwable {
14          System.out.println(System.currentTimeMillis() + " [主线程]开始执行");
15          Thread thread = new Thread(new LockSupportDemoTask());
16          thread.start();
17          System.out.println(System.currentTimeMillis() + " [主线程]执行了LockSupport.unpark(thread) , 释放对子线程的阻塞");
18          LockSupport.unpark(thread); /** unpark方法, 将thread的一个许可从[不可用状态]变为 [可用状态] */
19      } 首先: 执行unpark操作
20
21      static class LockSupportDemoTask implements Runnable {
22          @SneakyThrows
23          @Override
24          public void run() {
25              System.out.println(System.currentTimeMillis() + " [子线程]开始运行并且先sleep一秒");
26              TimeUnit.SECONDS.sleep(1);
27              System.out.println(System.currentTimeMillis() + " [子线程]执行LockSupport.park()方法, 阻塞本线程");
28              LockSupport.park(); /** park方法, 将thread的一个许可从[可用状态]变为 [不可用状态] */ 随后: 执行park操作
29              System.out.println(System.currentTimeMillis() + " [子线程]运行结束");
30          }
31      }
32  }
```

Run: LockSupportDemo

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
1648110884532 [主线程]开始执行
1648110884533 [主线程]执行了LockSupport.unpark(thread) , 释放对子线程的阻塞
1648110884534 [子线程]开始运行并且先sleep一秒
1648110885538 [子线程]执行LockSupport.park()方法, 阻塞本线程
1648110885538 [子线程]运行结束
Process finished with exit code 0
```

- 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 x
13 public class LockSupportDemo1 {
14     public static void main(String[] args) throws Throwable {
15         RunningDemo runningDemo = new RunningDemo();
16         Thread thread1 = new Thread(runningDemo, "T1");
17         Thread thread2 = new Thread(runningDemo, "T2");
18         thread1.start();
19         thread2.start();
20         TimeUnit.MILLISECONDS.sleep(100);
21         System.out.println("[主线程]执行interrupt()打断子线程" + thread1.getName());
22         thread1.interrupt(); 打断T1的park阻塞状态，不抛异常
23         TimeUnit.MILLISECONDS.sleep(100);
24         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
25         System.out.println("[主线程]针对子线程" + thread2.getName() + "执行LockSupport.unpark() 完毕! ");
26     }
27
28     static class RunningDemo implements Runnable {
29         private final Object object = new Object();
30         @Override
31         public void run() {
32             synchronized(object) {
33                 String threadName = Thread.currentThread().getName();
34                 System.out.println("[子线程]" + threadName + "开始运行! 执行LockSupport.park()方法，阻塞本线程!");
35                 /**
36                  * park()可以阻塞当前线程
37                  * 每一个线程有一个许可，该许可默认为[不可用]。
38                  * 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，将该许可消费更为[不可用]状态，流程代码可以继续执行。
39                  */
40                 LockSupport.park(); 执行线程阻塞
41
42                 /** 也可以使用Thread.currentThread().isInterrupted() */ 判断本线程是否被执行了interrupt方法
43                 if (Thread.interrupted()) {
44                     System.out.println("[子线程]" + threadName + "被interrupt()方法打断，但并不会抛出异常。");
45                 }
46                 System.out.println("[子线程]" + threadName + "运行结束! ");
47             }
48         }
49     }
50 }

Run: LockSupportDemo1 x
/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
```