第 5 章

Lock 锁

Lock 锁在 JDK 1.5 时被推出，同期推出的并发包为 Java 多线程奠定了坚实的基础，Lock 锁提供了比 synchronized 锁更广泛的锁操作，它们允许更灵活的锁结构并且可以关联多个 Condition 对象。

Lock 锁和 synchronized 锁的对比见表 5-1。

表 5-1　Lock 锁和 synchronized 锁

锁　特　性	Lock 锁	synchronized 锁
可重入	√	√
自动释放	×	√
可中断	√	×
可尝试拿锁，有等待时间	√	×
可尝试立即拿锁	√	×
同一个锁可以有多个唤醒及等待操作对象	√	×
灵活的锁资源获取	√	×
公平锁	√	×
读写分离锁	√	×

7min

5.1　Lock 接口

Lock 接口为锁提供了标准规范的方法，子类必须实现这些方法。

1. lock()

获得锁，阻塞当前执行线程直到获得锁为止。

2. lockInterruptibly()

获得锁，阻塞当前执行线程直到获得锁为止，或者如果当前执行线程中断，则抛出 InterruptedException 异常并且清除当前执行线程的中断状态。

3. newCondition()

创建与此锁相关联的新 Condition 对象，Condition 对象可以使当前执行线程唤醒或等待，类似 Object 对象监视器功能。

4. tryLock()

尝试拿锁，获得可用的锁并立即返回 true，如果锁不可用，则立即返回 false。

5. tryLock(long time, TimeUnit unit)

尝试拿锁并最大等待给定的时间段，如果获得可用的锁，则立即返回 true，否则超过等待的时间段返回 false。接收 long 入参，作为最大等待时间；接收 TimeUnit 入参，作为时间的单位。

6. unlock()

释放锁，只有锁的持有者才能释放锁。如果当前执行线程不是锁的持有者，则会引发 IllegalMonitorStateException 异常。

5.2 ReentrantLock


17min

ReentrantLock 实现了 Lock 接口，不仅提供了 Lock 接口标准规范的锁方法，还提供了灵活的锁资源获取。

5.2.1 构造器


7min

ReentrantLock 构造器见表 5-2。

表 5-2 ReentrantLock 构造器

构 造 器	描 述
ReentrantLock()	构造新的对象，默认无参构造器
ReentrantLock(boolean fair)	构造新的对象，指定是否使用公平锁

5.2.2 常用方法


7min

1. lock()


11min

获得锁，如果锁未被持有，则获得该锁，并将锁保持计数设置为 1；如果当前执行线程已经持有此锁，则保持计数将增加 1；如果锁由另一个执行线程持有，则当前执行线程将阻塞等待，直到获得锁为止。这里提到的保持计数就是可重入的一种标记。

标准使用示例，代码如下：

```
//第 5 章/one/OneMain.java
public class OneMain {

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
```

```
        Thread threadB = new Thread(lockRunnable, "B");
        threadB.start();
        threadA.start();
    }

static class LockRunnable implements Runnable{
        private final ReentrantLock lock = new ReentrantLock();
                            //创建 ReentrantLock 对象,默认构造器为非公平锁
        @Override
        public void run() {
            lock.lock();                    //获得锁
            try {
                System.out.println(Thread.currentThread().getName());
                Thread.sleep(2000);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                lock.unlock();              //释放锁
            }
        }
    }
}
```

执行结果如下：

```
B
A
```

2. unlock()

释放锁，如果当前执行线程是该锁的持有者，则保持计数将递减 1，若保持计数为 0，则释放锁；如果当前执行线程不是此锁的持有者，则会引发 IllegalMonitorStateException 异常。

3. lockInterruptibly()

获得锁，但比 lock()方法多了一个响应中断，代码如下：

```
//第 5 章/one/OneMain.java
public class OneMain {

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        Thread threadB = new Thread(lockRunnable, "B");
        threadB.start();
        threadA.start();
        Thread.sleep(200);
        threadB.interrupt();                    //中断
        threadA.interrupt();                    //中断
    }
```

```
    static class LockRunnable implements Runnable{

        private final ReentrantLock lock = new ReentrantLock();
                            //创建 ReentrantLock 对象,默认构造器为非公平锁
        @Override
        public void run() {
            try {
                lock.lockInterruptibly();          //获得锁,此方法响应中断
                System.out.println(Thread.currentThread().getName()
                                                    + ":lock");
                while (true){//死循环,为了演示中断效果
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if(lock.isHeldByCurrentThread()){//查询此锁是否由当前线程持有
                    System.out.println(Thread.currentThread().getName()
                                                      + ":unlock");
                    lock.unlock();                 //释放锁
                }
            }
        }
    }
}
```

执行结果如下：

```
A:lock
java.lang.InterruptedException
    at
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly
(AbstractQueuedSynchronizer.java:959)
    at
java.base/java.util.concurrent.locks.ReentrantLock$Sync.lockInterruptibly(ReentrantLock.
java:161)
    at
java.base/java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.java:
372)
    at
cn.kungreat.book.five.one.OneMain$LockRunnable.run(OneMain.java:25)
    at
java.base/java.lang.Thread.run(Thread.java:833)
```

4. isHeldByCurrentThread()

查询锁是否由当前执行线程持有，如果是，则返回值为 true，否返回值为 false。在释放锁时应配合使用，以防止出现 IllegalMonitorStateException 异常。

5. tryLock()

如果锁未被持有，则获得该锁，将锁保持计数设置为 1 并返回 true；如果当前执行线程已经持有该锁，则保持计数将增加 1 并返回 true；如果锁由另一个执行线程持有，则该方法

将立即返回 false。

即使此锁已设置使用公平策略，如果该锁可用，则调用此方法将立即获取该锁，无论其他执行线程是否正在等待该锁。这种行为在某些情况下是有用的，尽管它破坏了公平策略。

如果想遵守此锁的公平策略，则可使用 tryLock(0，TimeUnit. SECONDS)，代码如下：

```
//第 5 章/two/TwoMain.java
public class TwoMain {

    public static void main(String[] args) {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        Thread threadB = new Thread(lockRunnable, "B");
        threadA.start();
        threadB.start();
    }

    static class LockRunnable implements Runnable {

        private final ReentrantLock lock = new ReentrantLock();

        @Override
        public void run() {

            if (lock.tryLock()) {//尝试拿锁
                System.out.println(Thread.currentThread().getName()
                                                        + ":getLock");
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock(); //释放锁
                }
            } else {
                System.out.println(Thread.currentThread().getName()
                                                        + ":noLock");
            }

        }
    }
}
```

执行结果如下：

```
A:getLock
B:noLock
```

6. tryLock(long timeout, TimeUnit unit)

尝试拿锁并最大等待给定的时间段，如果获得可用的锁，则立即返回 true，否则超过最

大的等待时间段后返回 false。接收 long 入参,作为等待锁的最大时间;接收 TimeUnit 入参,作为时间的单位。该方法比 tryLock()多了最大等待给定的时间、响应中断、公平策略,代码如下:

```
//第 5 章/two/TwoMain.java
public class TwoMain {

    public static void main(String[] args) {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        Thread threadB = new Thread(lockRunnable, "B");
        threadA.start();
        threadB.start();
    }

    static class LockRunnable implements Runnable {

        private final ReentrantLock lock = new ReentrantLock();

        //最大等待时间充足,两个线程都可以获得锁
        @Override
        public void run() {
            try {
                if (lock.tryLock(5,TimeUnit.SECONDS)) {
                    System.out.println(Thread.currentThread().getName()
                                                            + ":getLock");
                    try {
                        Thread.sleep(2000);             //睡眠 2000ms
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        lock.unlock();                  //释放锁
                    }
                } else {
                    System.out.println(Thread.currentThread().getName()
                                                            + ":noLock");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

执行结果如下:

```
B:getLock
A:getLock
```

7. getHoldCount()

获得当前执行线程持有此锁的计数器,可以理解为锁重入的次数。如果此锁未由当前

执行线程持有，则为 0，代码如下：

```
//第 5 章/two/OtherMethod.java
public class OtherMethod {

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        threadA.start();
        Thread.sleep(500);                    //睡眠 500ms
        System.out.println("main:" + lockRunnable.lock.getHoldCount());
                                        //如果此锁未由当前线程持有，则为 0

    }
    static class LockRunnable implements Runnable {

        private final ReentrantLock lock = new ReentrantLock();

        @Override
        public void run() {
            lock.lock();
            try {
                System.out.println("run:" + lock.getHoldCount());
                                            //获得当前线程对此锁的计数器
                addCount();
            } finally {
                lock.unlock();
            }
            System.out.println("end:" + lock.getHoldCount());
                                           //获得当前线程对此锁的计数器
        }

        public void addCount(){
            lock.lock();
            try {
                Thread.sleep(2000);
                System.out.println("addCount:" + lock.getHoldCount());
                                            //获得当前线程对此锁的计数器
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
run:1
main:0
```

```
addCount:2
end:0
```

修改 OtherMethod 类，代码如下：

```
//第 5 章/two/OtherMethod.java
public class OtherMethod {

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        threadA.start();
        Thread.sleep(500);                    //睡眠 500ms
        lockRunnable.lock.lock();
        try {
            System.out.println("main:" + lockRunnable.lock.getHoldCount());
                                              //获得当前线程对此锁的计数器
        } finally {
            lockRunnable.lock.unlock();
        }
    }
    static class LockRunnable implements Runnable {

        private final ReentrantLock lock = new ReentrantLock();

        @Override
        public void run() {
            lock.lock();
            try {
                System.out.println("run:" + lock.getHoldCount());
                                             //获得当前线程对此锁的计数器
                addCount();
            } finally {
                lock.unlock();
            }
            System.out.println("end:" + lock.getHoldCount());
                                              //获得当前线程对此锁的计数器
        }

        public void addCount(){
            lock.lock();
            try {
                Thread.sleep(2000);
                System.out.println("addCount:" + lock.getHoldCount());
                                                    //获得当前线程对此锁的计数器
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

执行结果如下：

```
run:1
addCount:2
end:1
```

注意：观察上方的输出结果，并思考为什么主线程没有输出内容，而且JVM也没有结束。

8. getQueueLength()

返回等待获得此锁的执行线程数量的估计值，代码如下：

```
//第5章/two/MethodAll.java
public class MethodAll {

    private static final ReentrantLock LOCK = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        threadA.start();                    //启动线程
        Thread.sleep(200);                  //睡眠200ms
        LOCK.lock();                        //拿锁
        try {
            System.out.println(Thread.currentThread().getName());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            LOCK.unlock();                  //释放锁
        }
    }

    static class LockRunnable implements Runnable {

        @Override
        public void run() {
            LOCK.lock();                    //拿锁
            try {
                System.out.println(Thread.currentThread().getName());
                Thread.sleep(1000);         //睡眠1000ms
                System.out.println("getQueueLength:"
                                                    + LOCK.getQueueLength());
                                            //返回等待获得此锁的线程数量的估计值
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                LOCK.unlock();              //释放锁
            }
        }

    }
}
```

执行结果如下：

```
A
getQueueLength:1
main
```

9. hasQueuedThread(Thread thread)

查询给定线程对象是否正在等待获得此锁，返回 boolean 值。接收 Thread 入参，作为给定线程对象，代码如下：

```
//第 5 章/two/MethodAll.java
public class MethodAll {

    private static final ReentrantLock LOCK = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        LockRunnable lockRunnable = new LockRunnable();
        Thread threadA = new Thread(lockRunnable, "A");
        threadA.start();                    //启动线程
        Thread.sleep(200);                  //睡眠 200ms
        LOCK.lock();                        //拿锁
        try {
            System.out.println(Thread.currentThread().getName());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            LOCK.unlock();                  //释放锁
        }
    }

    static class LockRunnable implements Runnable {

        @Override
        public void run() {
            LOCK.lock();                    //拿锁
            try {
                System.out.println(Thread.currentThread().getName());
                                                        //当前执行线程对象名称
                Thread.sleep(1000);
                ThreadGroup threadGroup =
                                    Thread.currentThread().getThreadGroup();
                                //当前执行线程对象所属的线程组对象
                Thread[] threads = new Thread[threadGroup.activeCount()];
                threadGroup.enumerate(threads);
                                    //将组内活动线程对象复制到指定数组中
                for (int i = 0; i < threads.length; i++) {
                    if(threads[i].getName().equals("main")){
                        System.out.println("hasQueuedThread:"
                                + LOCK.hasQueuedThread(threads[i]));
                                //查询给定线程对象是否正在等待获得此锁
```

```
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                LOCK.unlock(); //释放锁
            }
        }

    }
}
```

执行结果如下：

```
A
hasQueuedThread:true
main
```

10. hasQueuedThreads()

查询是否有任何执行线程正在等待获得此锁，返回 boolean 值。

11. isFair()

获得此锁是否为公平锁，返回 boolean 值。

5.2.3 公平锁或非公平锁

此类的构造器接收一个可选的公平参数 boolean 值。当设置为 true 时，即公平锁，在并发拿锁的情况下倾向于等待时间最长的执行线程优先拿锁，否则此锁不保证任何特定的并发拿锁顺序。

使用公平锁的程序一般显示出比使用非公平锁的程序更低的吞吐量（通常更慢），但获得锁的时间差异较小，保证极端情况下不会出现饥饿状态（一个等待时间较久的执行线程，一直没有获得锁）。需要注意的是，此锁的公平性并不能保证系统 CPU 线程调度的公平性。

公平锁演示代码如下：

```
//第 5 章/two/FairTest.java
public class FairTest {

    private static final ReentrantLockMy LOCK = new ReentrantLockMy(true);

    public static void main(String[] args) throws InterruptedException {
        RunnableMy runnableMy = new RunnableMy();
        for (int i = 1; i < 6; i++) {
            new Thread(runnableMy,"线程" + i).start();
            Thread.sleep(100);              //保证线程启动的顺序
        }
    }
```

```
    static class RunnableMy implements Runnable{

        @Override
        public void run() {
            //循环两次,公平锁拿锁时需要排队,非公平锁可以直接拿锁
            for (int i = 0; i < 2; i++) {
                LOCK.lock(); //拿锁
                try {
                    Thread.sleep(1500);
                    System.out.println(LOCK.getQueuedThreads());
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    LOCK.unlock(); //释放锁
                }
            }
        }
    }

    static final class ReentrantLockMy extends ReentrantLock{
        public ReentrantLockMy(boolean fair) {
            super(fair);
        }


        @Override
        protected Collection<Thread> getQueuedThreads() {
            return super.getQueuedThreads();
        }
    }
}
```

执行结果如下：

```
[Thread[线程5,5,main], Thread[线程4,5,main], Thread[线程3,5,main], Thread[线程2,5,main]]
[Thread[线程1,5,main], Thread[线程5,5,main], Thread[线程4,5,main], Thread[线程3,5,main]]
[Thread[线程2,5,main], Thread[线程1,5,main], Thread[线程5,5,main], Thread[线程4,5,main]]
[Thread[线程3,5,main], Thread[线程2,5,main], Thread[线程1,5,main], Thread[线程5,5,main]]
[Thread[线程4,5,main], Thread[线程3,5,main], Thread[线程2,5,main], Thread[线程1,5,main]]
[Thread[线程5,5,main], Thread[线程4,5,main], Thread[线程3,5,main], Thread[线程2,5,main]]
[Thread[线程5,5,main], Thread[线程4,5,main], Thread[线程3,5,main]]
[Thread[线程5,5,main], Thread[线程4,5,main]]
[Thread[线程5,5,main]]
[]
```

注意：将上方代码修改为非公平锁实现，并运行主方法观察输出结果。

5.2.4 自旋锁

自旋锁是指当一个执行线程在尝试拿锁时，如果此锁已经被其他执行线程占有，则该执

行线程将循环一定的次数不断地尝试拿锁，直到获得锁或者超过一定的循环尝试次数，代码如下：

```
//第 5 章/two/SpinLock.java
public class SpinLock {
    public static final ReentrantLock REENTRANT_LOCK = new ReentrantLock();

    public static void main(String[] args) {
        ReentrantLockSpin reentrantLockSpin = new ReentrantLockSpin();
        new Thread(reentrantLockSpin).start();
        new Thread(reentrantLockSpin).start();
        new Thread(reentrantLockSpin).start();
        new Thread(reentrantLockSpin).start();
        new Thread(reentrantLockSpin).start();
    }

    static final class ReentrantLockSpin implements Runnable {

        @Override
        public void run() {
            //循环一定的次数拿锁，可以理解为自旋次数
            for (int i = 0; i < 100; i++) {
                if(REENTRANT_LOCK.tryLock()){//尝试拿锁
                    try {
                        System.out.println("获得锁后执行的操作");
                        return; //已经获得锁做完任务退出
                    } finally {
                        REENTRANT_LOCK.unlock();
                    }
                }
            }
        }
    }
}
```

注意：以上代码每次运行后的输出结果可能不相同，自旋锁的好处是在合理的范围内使用可以降低 CPU 上下文的切换，坏处是当使用不当时也会造成 CPU 资源的损耗、浪费。

30min

5.3 Condition

Condition 是一个接口，Lock 锁将 Object 对象监视器（wait、notify、notifyAll）方法分解为不同的 Condition 对象，Lock 实例对象可以通过 newCondition()方法创建多个 Condition 对象，每个 Condition 对象都可以提供类似等待、唤醒的效果。如果 Lock 锁取代了 synchronized 锁语句的使用，则 Condition 对象取代了 Object 对象监视器的使用。

1. await()

使当前执行线程阻塞等待，直到它被唤醒或中断，具有释放当前锁的特性。

2. await(long time, TimeUnit unit)

使当前执行线程阻塞等待，直到它被唤醒、中断或者超过最大等待时间，如果超过最大等待时间，则返回值为 false，否则返回值为 true，具有释放当前锁的特性。接收 long 入参，作为最大等待时间；接收 TimeUnit 入参，作为时间单位。代码如下：

```
//第 5 章/three/ConditionTest.java
public class ConditionTest {
    final Lock lock = new ReentrantLock();
    final Condition condition = lock.newCondition();

    public void testTime() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName());
            System.out.println(condition.await(2, TimeUnit.SECONDS));
            System.out.println(Thread.currentThread().getName() + ":end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void test() {
        lock.lock();
        try {
            System.out.println(System.currentTimeMillis());
                                                        //当前系统时间的毫秒数
            Thread.sleep(5000);
            System.out.println(System.currentTimeMillis());
                                                        //当前系统时间的毫秒数
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ConditionTest conditionTest = new ConditionTest();
        new Thread(new Runnable() {
            @Override
            public void run() {
                conditionTest.testTime();
            }
        }, "A").start();
        Thread.sleep(200);
        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                conditionTest.test();
            }
        }, "B").start();
    }
}
```

执行结果如下：

```
A
1668237496111
1668237501115
false
A:end
```

3. awaitNanos(long nanosTimeout)

使当前执行线程阻塞等待，直到它被唤醒、中断或者超过最大等待时间，返回纳秒数等于入参时间减去等待时间的估计值，大于 0 表示提前唤醒，小于或等于 0 表示没有剩余时间，具有释放当前锁的特性。接收 long 入参，作为最大等待时间纳秒数，代码如下：

```
//第 5 章/three/ConditionTestTwo.java
public class ConditionTestTwo {
    final Lock lock = new ReentrantLock();
    final Condition condition = lock.newCondition();

    public void testTime() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName());
            System.out.println(condition.awaitNanos(10000000000L)); //10s
            System.out.println(Thread.currentThread().getName() + ":end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void test() {
        lock.lock();
        try {
            System.out.println(System.currentTimeMillis());
                                                        //当前系统时间的毫秒数
            Thread.sleep(3000);
            System.out.println(System.currentTimeMillis());
                                                        //当前系统时间的毫秒数
            condition.signal(); //唤醒单个等待此 Condition 对象的线程
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
            } finally {
                lock.unlock();
            }
        }



        public static void main(String[] args) throws InterruptedException {
            ConditionTest conditionTest = new ConditionTest();
            new Thread(new Runnable() {
                @Override
                public void run() {
                    conditionTest.testTime();
                }
            }, "A").start();
            Thread.sleep(200);
            new Thread(new Runnable() {
                @Override
                public void run() {
                    conditionTest.test();
                }
            }, "B").start();
        }
    }
```

执行结果如下：

```
A
1668238608253
1668238611256
6797669700
A:end
```

4. awaitUninterruptibly()

使当前执行线程阻塞等待，直到它被唤醒，具有释放当前锁的特性。

5. signal()

唤醒单个等待此 Condition 对象的执行线程。

6. signalAll()

唤醒所有等待此 Condition 对象的执行线程。

此处模拟生产和消费的实例，数据存放在一个固定大小的缓冲区中。需要可以精准控制到具体方法中执行线程的等待或唤醒，即在缓冲区空间可用时通知到指定方法中的执行线程，就需要通过两个 Condition 实例来配合实现，代码如下：

```
//第 5 章/three/ConditionTest.java
public class ConditionTest<E> {
```

```
final Lock lock = new ReentrantLock();
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();

final Object[] items = new Object[100]; //数据缓冲区
int putptr, takeptr, count;

public void put(E x) throws InterruptedException {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;
        notEmpty.signal();
        System.out.println(Thread.currentThread().getName() + "put:" + x);
    } finally {
        lock.unlock();
    }
}

public E take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        E x = (E) items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}

public static void main(String[] args) {
    ConditionTest<Integer> conditionTest = new ConditionTest<>();
    Random random = new Random();
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                while (true){
                    Thread.sleep(1000);
                    int i = random.nextInt(9999);
                    conditionTest.put(i);
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
```

```
                }
            }
        },"A").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (true){
                        System.out.println(Thread.currentThread().getName()
                                + ":" + conditionTest.take());
                    }
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        },"B").start();
    }
}
```

注意：观察上方代码，并思考可以精准控制到具体方法中执行线程的等待或唤醒。

5.4 ReentrantReadWriteLock

此类对象可以创建一个读锁和一个写锁，读锁、写锁都实现了 Lock 接口。在多线程并发时读锁之间可以共享（可同时拿锁），在多线程并发时读锁和写锁之间互斥。此类适用于读多写少的场景，读锁不支持 newCondition()，默认抛出 UnsupportedOperationException 异常。

5.4.1 构造器

ReentrantReadWriteLock 构造器见表 5-3。

表 5-3 ReentrantReadWriteLock 构造器

构 造 器	描 述
ReentrantReadWriteLock()	构造新的对象，默认为无参构造器
ReentrantReadWriteLock(boolean fair)	构造新的对象，指定是否使用公平锁

5.4.2 共享锁和互斥锁

1. 共享锁

在多线程并发时读锁之间可以共享（可同时拿锁），代码如下：

```
//第 5 章/four/ReadWriteMain.java
public class ReadWriteMain {
```

```
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                    new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                  REENTRANT_READ_WRITE_LOCK.readLock(); //获得读锁

    public static void main(String[] args) {
        RunnableMy runnableMy = new RunnableMy();
        Thread threadA = new Thread(runnableMy, "A");
        Thread threadB = new Thread(runnableMy, "B");
        threadA.start();
        threadB.start();
    }

    static class RunnableMy implements Runnable{

        @Override
        public void run() {
            readLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                        + ":" + System.currentTimeMillis());
                                        //输出当前执行线程名称:当前系统时间(ms)
                Thread.sleep(5000);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
B:1668914471426
A:1668914471427
```

2. 互斥锁

在多线程并发时读锁和写锁之间互斥，代码如下：

```
//第 5 章/four/ReadWriteMain.java
public class ReadWriteMain {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                      new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                              REENTRANT_READ_WRITE_LOCK.readLock();
                                                          //获得读锁
    static final ReentrantReadWriteLock.WriteLock writeLock =
                              REENTRANT_READ_WRITE_LOCK.writeLock();
                                                          //获得写锁
```

```
    public static void main(String[] args) {
        RunnableMy runnableMy = new RunnableMy();
        Thread threadA = new Thread(runnableMy, "A");
        Thread threadB = new Thread(new Runnable() {
            @Override
            public void run() {
                runnableMy.writeTest();
            }
        }, "B");
        threadA.start();
        threadB.start();
    }

    static class RunnableMy implements Runnable{

        @Override
        public void run() {
            readLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                    + ":" + System.currentTimeMillis());
                                    //输出当前执行线程名称:当前系统时间(ms)
                Thread.sleep(5000);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void writeTest(){
            writeLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                    + ":" + System.currentTimeMillis());
                                    //输出当前执行线程名称:当前系统时间(ms)
                Thread.sleep(5000);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                writeLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
A:1668914799127
B:1668914804139
```

5.4.3 重入特性

1. 持有写锁时可以再获得读锁

代码如下：

```
//第 5 章/four/ReentryMain.java
public class ReentryMain {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                        new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                            REENTRANT_READ_WRITE_LOCK.readLock();
    static final ReentrantReadWriteLock.WriteLock writeLock =
                            REENTRANT_READ_WRITE_LOCK.writeLock();

    public static void main(String[] args) {
        RunnableMy runnableMy = new RunnableMy();
        runnableMy.writeTest();
    }

    static class RunnableMy implements Runnable{

        @Override
        public void run() {
            readLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                                            + ":run");
                                                    //输出当前执行线程名称
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void writeTest(){
            writeLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                                                + ":writeTest");
                                                    //输出当前执行线程名称
                run();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                writeLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
main:writeTest
main:run
```

2. 持有读锁时不可以再获得写锁

代码如下：

```
public class ReentryMain {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                        new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                            REENTRANT_READ_WRITE_LOCK.readLock();
    static final ReentrantReadWriteLock.WriteLock writeLock =
                            REENTRANT_READ_WRITE_LOCK.writeLock();

    public static void main(String[] args) {
        RunnableMy runnableMy = new RunnableMy();
        runnableMy.run();
    }

    static class RunnableMy implements Runnable{

        @Override
        public void run() {
            readLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                                            + ":run");
                                                    //输出当前执行线程名称
                writeTest();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void writeTest(){
            writeLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                                        + ":writeTest");
                                                //输出当前执行线程名称
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                writeLock.unlock();
            }
        }
    }
}
```

注意：观察上方代码结构，一定要避免在持有读锁时再次尝试获得写锁，以上代码会造成执行线程无限期阻塞。

3. 锁降级

观察官方文档示例代码，如图 5-1 所示。

```
class CachedData {
   Object data;
   boolean cacheValid;
   final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

   void processCachedData() {
     rwl.readLock().lock();//获得读锁
     if (!cacheValid) {
       //在获得写锁之前必须释放读锁
       rwl.readLock().unlock();//释放读锁
       rwl.writeLock().lock();//获得写锁
       try {
         if (!cacheValid) {
           data = ...;
           cacheValid = true;
         }
         //在释放写锁之前获取读锁
         rwl.readLock().lock();
       } finally {
         rwl.writeLock().unlock(); //释放写锁，如果仍保持读取，则称之为锁降级
       }
     }

     try {
       use(data);//处理数据
     } finally {
       rwl.readLock().unlock();//释放读锁
     }
   }
 }
```

图 5-1　官方文档示例代码

5.4.4　常用方法

1. isWriteLockedByCurrentThread()

查询此对象写锁是否由当前执行线程持有，如果是，则返回值为 true，否则返回值为 false，代码如下：

```
//第 5 章/four/MethodMain.java
public class MethodMain {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                    new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.WriteLock writeLock =
                            REENTRANT_READ_WRITE_LOCK.writeLock();

    public static void main(String[] args) {
```

```
            writeLock.lock();
            try {
                System.out.println(REENTRANT_READ_WRITE_LOCK
                                                .isWriteLockedByCurrentThread());
                                                //查询此对象写锁是否由当前执行线程持有
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                writeLock.unlock();
            }
        }
}
```

执行结果如下：

```
true
```

2. getQueueLength()

返回等待获得此对象读写锁的执行线程数量的估计值，代码如下：

```
//第 5 章/four/MethodMain.java
public class MethodMain {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                        new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                                REENTRANT_READ_WRITE_LOCK.readLock();
    static final ReentrantReadWriteLock.WriteLock writeLock =
                                REENTRANT_READ_WRITE_LOCK.writeLock();

    public static void main(String[] args) throws InterruptedException {
        RunnableMy runnableMy = new RunnableMy();
        new Thread(runnableMy,"A").start();
        new Thread(runnableMy,"B").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                runnableMy.writeTest();
            }
        }, "C").start();
        runnableMy.writeTest();
    }

    static class RunnableMy implements Runnable{
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                readLock.lock();
                System.out.println(Thread.currentThread().getName()
                                                                + ":run");
```

```
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void writeTest(){
            writeLock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                                                        +":writeTest");
                Thread.sleep(1500);
              System.out.println(REENTRANT_READ_WRITE_LOCK
                                                          .getQueueLength());
                                              //返回等待获得读写锁的执行线程数量的估计值
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                writeLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
main:writeTest
3
C:writeTest
2
A:run
B:run
```

注意：由于多线程并发原因，所以输出结果可能不同，但是第 1 次输出的 getQueueLength()方法一定是 3。

3. getReadHoldCount()

获得当前执行线程持有此对象读锁的计数器，可以理解为读锁重入的次数。如果读锁未由当前执行线程持有，则为 0，代码如下：

```
//第 5 章/four/MethodMainOther.java
public class MethodMainOther {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                          new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                              REENTRANT_READ_WRITE_LOCK.readLock();
```

```
    public static void main(String[] args) throws InterruptedException {
        RunnableMy runnableMy = new RunnableMy();
        new Thread(runnableMy,"A").start();
        new Thread(runnableMy,"B").start();
    }
    static class RunnableMy implements Runnable{
        @Override
        public void run() {
            readLock.lock();
            try {
                Thread.sleep(500);
                readTest();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void readTest(){
            readLock.lock();
            try {
                Thread.sleep(100);
                System.out.println(REENTRANT_READ_WRITE_LOCK
                                                        .getReadHoldCount());
                                                //获得当前执行线程对此对象读锁的计数器
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
2
2
```

4. getReadLockCount()

获得此对象读锁的计数器，可以理解为读锁重入的次数，代码如下：

```
//第 5 章/four/MethodMainOther.java
public class MethodMainOther {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                                new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.ReadLock readLock =
                            REENTRANT_READ_WRITE_LOCK.readLock();
```

```
    public static void main(String[] args) throws InterruptedException {
        RunnableMy runnableMy = new RunnableMy();
        new Thread(runnableMy,"A").start();
        new Thread(runnableMy,"B").start();
    }
    static class RunnableMy implements Runnable{
        @Override
        public void run() {
            readLock.lock();
            try {
                Thread.sleep(500);
                readTest();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }

        public void readTest(){
            readLock.lock();
            try {
                Thread.sleep(100);
                System.out.println(REENTRANT_READ_WRITE_LOCK
                                                        .getReadLockCount());
                                                        //获得此对象读锁的计数器
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                readLock.unlock();
            }
        }
    }
}
```

执行结果如下：

```
4
4
```

5. getWriteHoldCount()

获得当前执行线程持有此对象写锁的计数器，可以理解为写锁重入的次数。如果写锁未由当前执行线程持有，则为 0。

6. getWaitQueueLength(Condition condition)

返回与此对象写锁相关联的 Condition 条件下的执行线程等待数量的估计值。接收 Condition 入参，作为给定条件对象，代码如下：

```
//第 5 章/four/MethodMainWait.java
public class MethodMainWait {
    static final ReentrantReadWriteLock REENTRANT_READ_WRITE_LOCK =
                                        new ReentrantReadWriteLock();
    static final ReentrantReadWriteLock.WriteLock WRITE_LOCK =
                                REENTRANT_READ_WRITE_LOCK.writeLock();
    static final Condition condition = WRITE_LOCK.newCondition();

    public static void main(String[] args) throws InterruptedException {
        RunnableMy runnableMy = new RunnableMy();
        new Thread(runnableMy,"A").start();
        new Thread(runnableMy,"B").start();
        Thread.sleep(1000);
        WRITE_LOCK.lock();
        try {
            System.out.println(REENTRANT_READ_WRITE_LOCK
                                        .getWaitQueueLength(condition));
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            WRITE_LOCK.unlock();
        }
    }

    static class RunnableMy implements Runnable{
        @Override
        public void run() {
            WRITE_LOCK.lock();
            try {
                condition.await(); //使当前执行线程阻塞等待
                System.out.println(Thread.currentThread().getName());
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                WRITE_LOCK.unlock();
            }
        }

    }
}
```

执行结果如下：

```
2
B
A
```

7. hasQueuedThread(Thread thread)

查询给定线程对象是否正在等待获得此对象读锁或写锁，返回 boolean 值。接收

Thread 入参，作为给定线程对象。

8. hasQueuedThreads()

查询是否有任何执行线程正在等待获得此对象读锁或写锁，返回 boolean 值。

9. hasWaiters(Condition condition)

查询与此对象写锁相关联的 Condition 条件下的执行线程是否有正在等待获得此写锁，如果有，则为 true，否则为 false。接收 Condition 入参，作为给定条件对象。

10. isFair()

获得此锁是否为公平锁，返回 boolean 值。

小结

Lock 锁提供了比 synchronized 锁更强大、更灵活的功能，但是使用起来的复杂度也比 synchronized 锁会更高一些，需要根据具体的业务场景选择合适的锁方案，并不是说哪一方一定强，哪一方一定弱。

习题

1. 判断题

(1) Lock 锁出现异常时会自动释放锁。(　　)

(2) synchronized 锁出现异常时会自动释放锁。(　　)

(3) ReentrantReadWriteLock 读写锁可以在持有读锁时再获得写锁。(　　)

(4) ReentrantReadWriteLock 读写锁可以在持有写锁时再获得读锁。(　　)

(5) 公平锁相较非公平锁往往更慢。(　　)

(6) ReentrantReadWriteLock 读写锁读锁之间多线程并发共享。(　　)

(7) ReentrantReadWriteLock 读写锁读锁、写锁之间多线程并发互斥。(　　)

(8) ReentrantLock 对象可以创建多个 Condition 对象。(　　)

(9) ReentrantReadWriteLock 读写锁读锁不支持创建 Condition 对象。(　　)

2. 选择题

(1) Lock 锁拿锁时可以立即返回的方法是(　　)。(单选)

A. lock()　　B. lockInterruptibly()

C. tryLock()　　D. unlock()

(2) Lock 锁最大等待时间拿锁的方法是(　　)。(单选)

A. tryLock()　　B. unlock()

C. tryLock(long time, TimeUnit unit)　　D. newCondition()

(3) ReentrantLock 查询此锁是否由当前执行线程持有的方法是(　　)。(单选)

A. isLocked()　　B. hasQueuedThreads()

C. isHeldByCurrentThread()　　D. isFair()

(4) 以下(　　)类实现了 Lock 接口。(多选)

A. ReentrantLock

B. ReentrantReadWriteLock

C. ReentrantReadWriteLock. ReadLock

D. ReentrantReadWriteLock. WriteLock

(5) Lock 锁在公平模式下,以下(　　)方法不支持公平策略。(单选)

A. lock()

B. lockInterruptibly()

C. tryLock()

D. tryLock(long time, TimeUnit unit)

3. 填空题

(1) 根据业务要求补全代码,多线程并发拿锁时要求不阻塞并立即返回,然后输出线程名称,代码如下:

```
//第 5 章/answer/TryLock.java
public class TryLock {

    private static final Lock LOCK = ____________________________;

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        new Thread(myRunnable,_________).start();
        new Thread(myRunnable,_________).start();
        new Thread(myRunnable,_________).start();
        new Thread(myRunnable,_________).start();
    }

    private static final class MyRunnable implements Runnable{

        @Override
        public void run() {
            if(_________________){
                try {
                    System.out.println(_____________________+":获得锁");
                } finally {
                    LOCK.unlock();
                }
            }else{
                System.out.println(_____________________+":没获得锁");
            }
        }
    }
}
```

(2) 根据业务要求补全代码,参考官方文档完成锁降级,代码如下:

```
//第 5 章/answer/CachedData.java
public class CachedData {
    static final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    static boolean cacheValid = false;

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        new Thread(myRunnable,"A").start();
        new Thread(myRunnable,"B").start();
        new Thread(myRunnable,"C").start();
        new Thread(myRunnable,"D").start();
    }

    private static final class MyRunnable implements Runnable{

        @Override
        public void run() {
            rwl.readLock().lock();
            if (!cacheValid) {
                ____________________________;
                rwl.writeLock().lock();
                try {
                    if (!cacheValid) {
                        System.out.println("模拟数据写操作...");
                        cacheValid = true;
                    }
                    //通过在释放写锁之前获得读锁来完成锁降级
                    rwl.readLock().lock();
                } finally {
                    ____________________________;
                }
            }
            try {
                System.out.println("模拟数据读操作...");
            } finally {
                rwl.readLock().unlock();
            }
        }
    }
}
```

（3）根据业务要求补全代码，参考官方文档完成读写锁，代码如下：

```
//第 5 章/answer/RWDictionary.java
public class RWDictionary {
    private final Map<String, Object> m = new TreeMap<>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();
```

```
    public Object get(String key) {
        _____;
        try {
            return m.get(key);
        } finally {
            _____;
        }
    }

    public List < String > allKeys() {
        r.lock();
        try {
            return new ArrayList <>(m.keySet());
        } finally {
            r.unlock();
        }
    }

    public Object put(String key, Object value) {
        _____;
        try {
            return m.put(key, value);
        } finally {
            _____;
        }
    }

    public void clear() {
        _____
        try {
            m.clear();
        } finally {
            _____;
        }
    }
}
```

（4）根据业务要求补全代码，A、B两个执行线程将按照1～100的顺序轮流打印数字，代码如下：

```
//第 5 章/answer/LoopPrint.java
public class LoopPrint {
    private static final ReentrantLock REENTRANT_LOCK = new ReentrantLock();
    private static final Condition CONDITION = REENTRANT_LOCK.newCondition();

    public static void main(String[] args) throws InterruptedException {
        LoopRunnable loopRunnable = new LoopRunnable();
        new Thread(loopRunnable, "A").start();
        new Thread(loopRunnable, "B").start();
    }
```

```
    static class LoopRunnable implements Runnable {
        private volatile int num = 1;

        @Override
        public void run() {
            REENTRANT_LOCK.lock();
            try {
                for (; num <= 100; ) {
                    System.out.println(Thread.currentThread().getName()
                                                                + ":" + num);
                    num++;
                    _____________________________;
                    _____________________________;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                _____________________________;
                REENTRANT_LOCK.unlock();
            }
        }
    }
}
```

（5）根据业务要求补全代码，启动 3 个执行线程 A、B、C，每个执行线程将自己的名称轮流打印 5 遍，打印顺序是 ABCABC…，代码如下：

```
//第 5 章/answer/ThreePrint.java
public class ThreePrint {

    private static final ReentrantLock LOCK = new ReentrantLock();
    private static final Condition CONDITION = _____________________________;

    public static void main(String[] args) {
        LoopRunnable loopRunnable = new LoopRunnable();
        new Thread(loopRunnable, "A").start();
        new Thread(loopRunnable, "B").start();
        new Thread(loopRunnable, "C").start();
    }

    static class LoopRunnable implements Runnable {
        private volatile int loopIndex = 0;
        private final String[] loopNames = {"A", "B", "C"};

        @Override
        public void run() {
            for (int x = 0; x < 5; x++) {
                LOCK.lock();
                try {
                    String name = Thread.currentThread().getName();
```

```
                    //名称不匹配时一直循环
                    while (!name.equals(loopNames[loopIndex])) {
                        _____________________________;
                    }
                    System.out.print(name);              //消费名称
                    loopIndex++;
                    if (loopIndex == loopNames.length) {
                        loopIndex = 0;                   //重置指针
                    }
                    _____________________________;
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    LOCK.unlock();
                }
            }
        }
    }
}
```

（6）模拟从1累加到100，启动10个执行线程平分此任务，主线程需要等待其他执行线程任务完成，然后统计计算结果并输出，代码如下：

```
//第5章/answer/TenCount.java
public class TenCount {

    public static void main(String[] args) {
        FutureRun[] futureRuns = new FutureRun[10];
        for (int i = 0; i < futureRuns.length; i++) {
            FutureRun futureRun = new FutureRun(i * 10 + 1);
            futureRuns[i] = futureRun;
            new Thread(futureRun).start();
        }
        int numCount = 0;
        for (int i = 0; i < futureRuns.length; i++) {
            numCount = numCount + futureRuns[i].getCountNum();
        }
        System.out.println("总结果:" + numCount);
    }

    static final class FutureRun implements Runnable {

        private final ReentrantLock reentrantLock = new ReentrantLock();
        private final Condition condition = reentrantLock.newCondition();
        private final int startNum;
        private boolean isEnd = false;
        private int countNum;

        public FutureRun(int startNum) {
            this.startNum = startNum;
```

```
        }

        @Override
        public void run() {
            int endNum = startNum + 10;
            reentrantLock.lock();
            try {
                for (int i = startNum; i < endNum; i++) {
                    countNum = countNum + i;
                    Thread.sleep(1000);
                }
                System.out.println(Thread.currentThread().getName()
                                                                + ":done");
            } catch (InterruptedException e) {
                countNum = 0;
                e.printStackTrace();
            } finally {
                isEnd = true;
                ____________________________;
                reentrantLock.unlock();
            }
        }

        public int getCountNum() {
            reentrantLock.lock();
            try {
                while (!isEnd) {
                    ____________________________;
                }
                return countNum;
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                reentrantLock.unlock();
            }
            return 0;
        }
    }
}
```

注意：此案例都在演示执行线程之间如何协作，FutureRun 对象的数据并没有多线程并发操作。