Java 并发编程之美：并发编程高级篇之三

一、前言

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

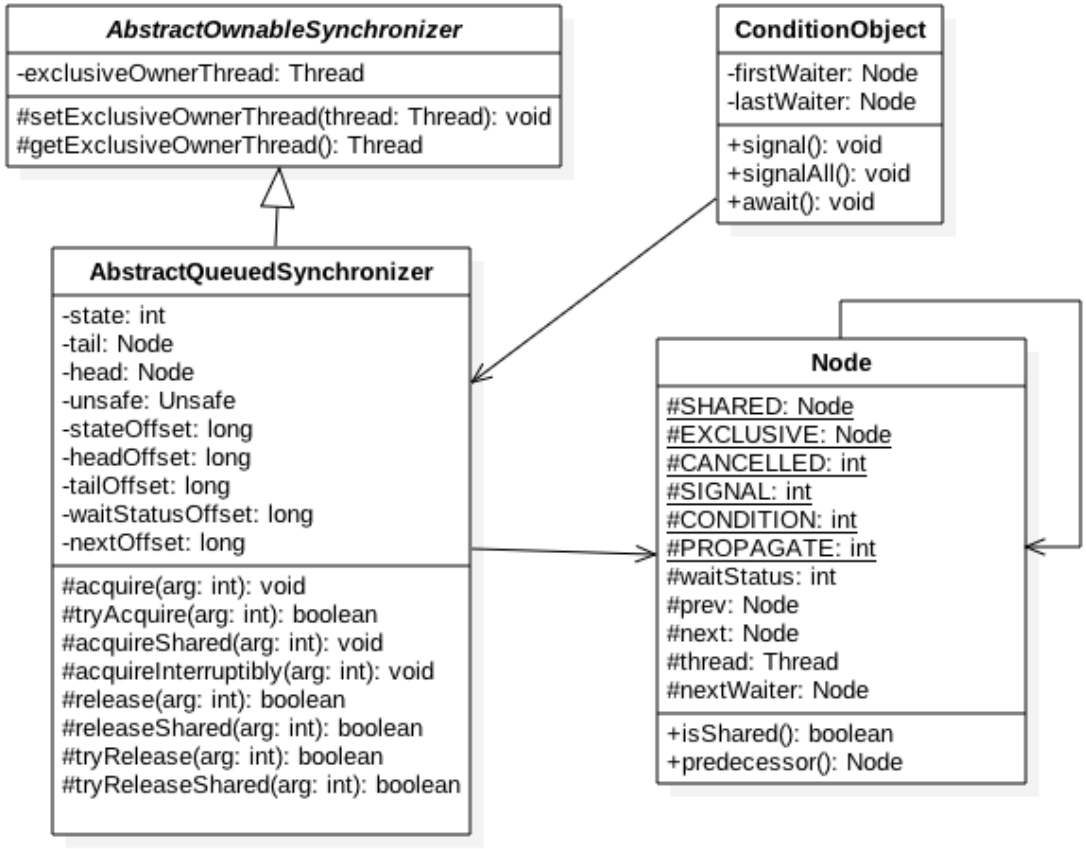
本 Chat 作为 Java 并发编程之美系列的高级篇之三，主要讲解锁，内容如下：（建议先阅读 Java 并发编程之美：并发编程高级篇之二 ）

- 抽象同步队列 AQS (AbstractQueuedSynchronizer) 概述，AQS 是实现同步的基础组件，并发包中锁的实现底层就是使用 AQS 实现，虽然大多数开发者可能从来不会直接用到 AQS，但是知道其原理对于架构设计还是很有帮助的。
- 独占锁 ReentrantLock 原理探究，ReentrantLock 是可重入的独占锁或者叫做排它锁，同时只能有一个线程可以获取该锁，其实现分为公平与非公平的独占锁。
- 读写锁 ReentrantReadWriteLock 原理，ReentrantLock 是独占锁，同时只有一个线程可以获取该锁，而实际情况下会有写少读多的场景，显然 ReentrantLock 满足不了需求，所以 ReentrantReadWriteLock 应运而生，本文来介绍读写分离锁的实现。
- StampedLock 锁原理探究，StampedLock 是并发包里面 jdk8 版本新增的一个锁，该锁提供了三种模式的读写控制。

二、抽象同步队列 AQS 概述

2.1 AQS - 锁的底层支持

AbstractQueuedSynchronizer 抽象同步队列, 简称 AQS，是实现同步器的基础组件，并发包中锁的实现底层就是使用 AQS 实现，另外大多数开发者可能从来不会直接用到 AQS，但是知道其原理对于架构设计还是很有帮助的，下面看下 AQS 的类图结构：



AQS 是一个 FIFO 的双向队列，内部通过节点 head 和 tail 记录队首和队尾元素，队列元素类型为 Node。其中 Node 中 thread 变量用来存放进入 AQS 队列里面的线程；Node 节点内部 SHARED 用来标记该线程是获取共享资源时候被阻塞挂起后放入 AQS 队列，EXCLUSIVE 标示线程是获取独占资源时候被挂起后放入 AQS 队列；waitStatus 记录当前线程等待状态，分别为 CANCELLED（线程被取消了），SIGNAL（线程需要被唤醒），CONDITION（线程在条件队列里面等待），PROPAGATE（释放共享资源时候需要通知其它节点）；prev 记录当前节点的前驱节点，next 记录当前节点后继节点。

AQS 中维持了一个单一的状态信息 state, 可以通过 getState,setState,compareAndSetState 函数修改其值；对于 ReentrantLock 的实现来说，state 可以用来表示当前线程获取锁的可重入次数；对应读写锁 ReentrantReadWriteLock 来说 state 的高 16 位表示读状态也就是获取该读锁的次数，低 16 位表示获取到写锁的线程的可重入次数；对于 semaphore 来说 state 用来表示当前可用信号的个数；对于 FutuerTask 来说，state 用来表示任务状态（例如还没开始，运行，完成，取消）；对应 CountDownlatch 和 CyclicBarrie 来说 state 用来表示计数器当前的值。

AQS 有个内部类 ConditionObject 是用来结合锁实现线程同步，ConditionObject 可以直接访问 AQS 对象内部的变量，比如 state 状态值和 AQS 队列；ConditionObject 是条件变量，每个条件变量对应着一个条件队列 (单向链表队列)，用来存放调用条件变量的 await() 方法后被阻塞的线程，如类图，这个条件队列的头尾元素分别为 firstWaiter 和 lastWaiter。

对于 AQS 来说线程同步的关键是对状态值 state 进行操作，根据 state 是否属于一个线程，操作 state 的方式分为独占模式和共享模式。

独占方式下获取和释放资源使用方法为：

```
void acquire(int arg)
void acquireInterruptibly(int arg)
boolean release(int arg)
```

共享模式下获取和释放资源方法为：

```
void acquireShared(int arg)
void acquireSharedInterruptibly(int arg)
boolean releaseShared(int arg)
```

对于独占方式获取的资源是与具体线程绑定的，就是说如果一个线程获取到了资源，就会标记是那个线程获取到了，其它线程尝试操作 state 获取资源时候发现当前该资源不是自己持有的，就会获取失败后被阻塞；比如独占锁 ReentrantLock 的实现，当一个线程获取了 ReentrantLock 的锁后，AQS 内部会首先使用 CAS 操作把 state 状态值从 0 变为 1，然后设置当前锁的持有者为当前线程，当该线程再次获取锁时候发现当前线程就是锁的持有者则会把状态值从 1 变为 2 也就是设置可重入次数，当另外一个线程获取锁的时候发现自己并不是该锁的持有者就会被放入 AQS 阻塞队列后挂起。

对应共享操作方式资源是与具体线程不相关的，多个线程去请求资源时候是通过 CAS 方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次获取时候如果当前资源还能满足它的需要，则当前线程只需要使用 CAS 方式进行获取即可，共享模式下并不需要记录那个线程获取了资源；比如 Semaphore 信号量，当一个线程通过 acquire() 方法获取一个信号量时候，会首先看当前信号量个数是否满足需要，不满足则把当前线程放入阻塞队列，如果满足则通过自旋 CAS 获取信号量。

对应独占模式的获取与释放资源流程：

1）当一个线程调用 acquire(int arg) 方法获取独占资源时候，会首先使用 tryAcquire 尝试获取资源，具体是设置状态变量 state 的值，成功则直接返回。失败则将当前线程封装为类型为 Node.EXCLUSIVE 的 Node 节点后插入到 AQS 阻塞队列尾部，并调用 LockSupport.park(this) 挂起当前线程。

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

2）当一个线程调用 release(int arg) 时候会尝试使用, tryRelease 操作释放资源，这里是设置状态变量 state 的值，然后调用 LockSupport.unpark(thread) 激活 AQS 队列里面最早被阻塞的线程 (thread)。被激活的线程则使用 tryAcquire 尝试看当前状态变量 state 的值是

否能满足自己的需要，满足则该线程被激活然后继续向下运行，否者还是会被放入 AQS 队列并被挂起。

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

需要注意的 AQS 类并没有提供可用的 tryAcquire 和 tryRelease，正如 AQS 是锁阻塞和同步器的基础框架，tryAcquire 和 tryRelease 需要有具体的子类来实现。子类在实现 tryAcquire 和 tryRelease 时候要根据具体场景使用 CAS 算法尝试修改状态值 state, 成功则返回 true, 否者返回 false。子类还需要定义在调用 acquire 和 release 方法时候 state 状态值的增减代表什么含义。

比如继承自 AQS 实现的独占锁 ReentrantLock，定义当 status 为 0 的时候标示锁空闲，为 1 的时候标示锁已经被占用，在重写 tryAcquire 时候，内部需要使用 CAS 算法看当前 status 是否为 0，如果为 0 则使用 CAS 设置为 1，并设置当前线程的持有者为当前线程，并返回 true, 如果 CAS 失败则 返回 false。

比如继承自 AQS 实现的独占锁实现 tryRelease 时候，内部需要使用 CAS 算法把当前 status 值从 1 修改为 0，并设置当前锁的持有者为 null，然后返回 true, 如果 cas 失败则返回 false。

对应共享资模式的获取与释放流程：

1）当线程调用 acquireShared(int arg) 获取共享资源时候，会首先使用 tryAcquireShared 尝试获取资源，具体是设置状态变量 state 的值，成功则直接返回。失败则将当前线程封装为类型为 Node.SHARED 的 Node 节点后插入到 AQS 阻塞队列尾部，并使用 LockSupport.park(this) 挂起当前线程。

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

2）当一个线程调用 releaseShared(int arg) 时候会尝试使用, tryReleaseShared 操作释放资源，这里是设置状态变量 state 的值，然后使用 LockSupport.unpark（thread）激活 AQS 队列里面最早被阻塞的线程 (thread)。被激活的线程则使用 tryReleaseShared 尝试看当前状态变量 state 的值是否能满足自己的需要，满足则该线程被激活然后继续向下运行，否者还是会被放入 AQS 队列并被挂起。

```
public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }
```

同理需要注意的 AQS 类并没有提供可用的 tryAcquireShared 和 tryReleaseShared，正如 AQS 是锁阻塞和同步器的基础框架，tryAcquireShared 和 tryReleaseShared 需要有具体的子类来实现。子类在实现 tryAcquireShared 和 tryReleaseShared 时候要根据具体场景使用 CAS 算法尝试修改状态值 state, 成功则返回 true, 否者返回 false。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryAcquireShared 时候，首先看写锁是否被其它线程持有，如果是则直接返回 false，否者使用 cas 递增 status 的高 16 位，在 ReentrantReadWriteLock 中 status 的高 16 为获取读锁的次数。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryReleaseShared 时候，内部需要使用 CAS 算法把当前 status 值的高 16 位减一，然后返回 true, 如果 cas 失败则返回 false。

基于 AQS 实现的锁除了需要重写上面介绍的方法，还需要重写 isHeldExclusively 方法用来判断锁是被当前线程独占还是被共享。

另外也许你会好奇独占模式下的

```
void acquire(int arg)
void acquireInterruptibly(int arg)
```

和共享模式下获取资源的：

```
void acquireShared(int arg)
void acquireSharedInterruptibly(int arg)
```

这两套函数都有一个带有 Interruptibly 关键字的函数，那么带有这个关键字的和不带的有什么区别那?

其实不带 Interruptibly 关键字的方法是说不对中断进行响应，也就是线程在调用不带 Interruptibly 关键字的方法在获取资源的时候或者获取资源失败被挂起时候，其他线程中断了该线程，那么该线程不会因为被中断而抛出异常，还是继续获取资源或者被挂起，也就是不对中断进行响应，忽略中断。

带 Interruptibly 关键字的方法是说对中断进行响应，也就是也就是线程在调用带 Interruptibly 关键字的方法在获取资源的时候或者获取资源失败被挂起时候，其他线程中

断了该线程，那么该线程会抛出 InterruptedException 异常而返回。

本节最后我们来看看 AQS 提供的队列是如何维护的，主要看入队操作

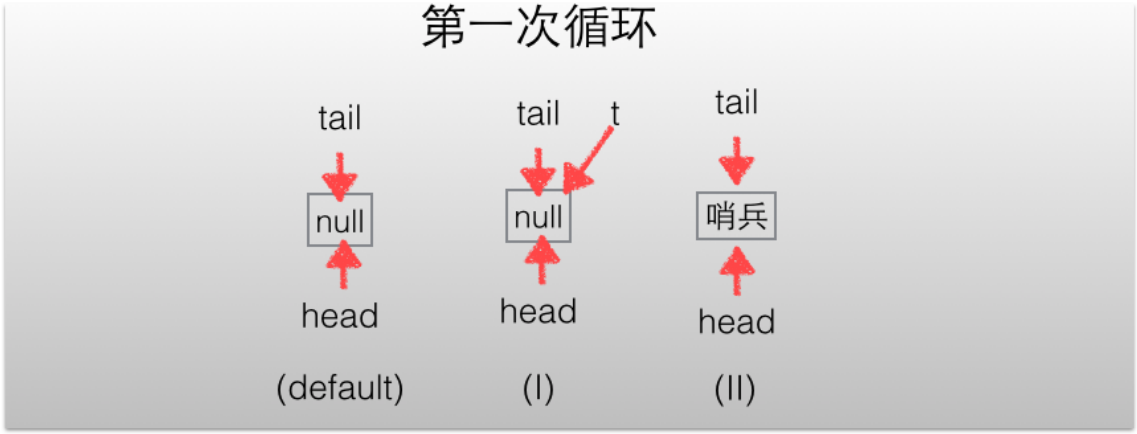
- 入队操作： 当一个线程获取锁失败后该线程会被转换为 Node 节点，然后就会使用 enq(final Node node) 方法插入该节点到 AQS 的阻塞队列，

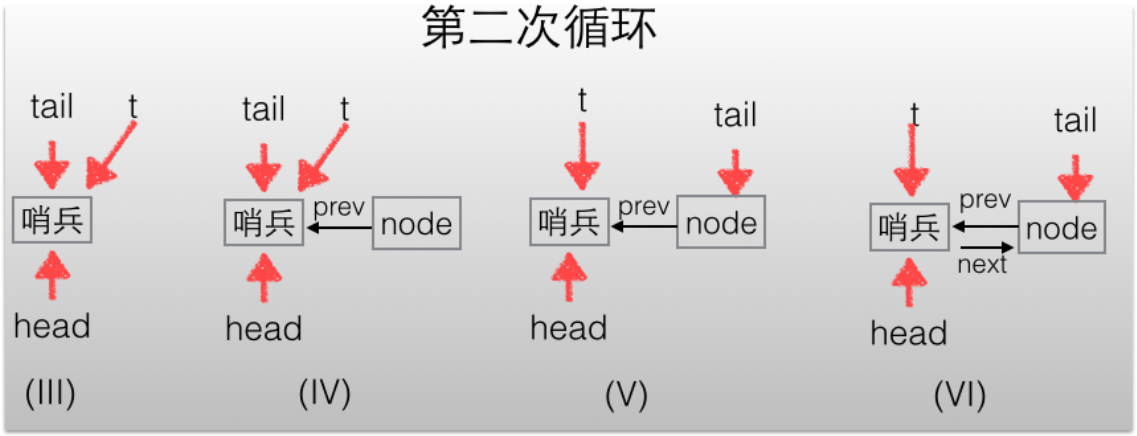
```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;//(1)
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))//(2)
                tail = head;
        } else {
            node.prev = t;//(3)
            if (compareAndSetTail(t, node)) {//(4)
                t.next = node;
                return t;
            }
        }
    }
}
```

下面结合代码和下面的节点图来讲解下，如上代码第一次循环当要在 AQS 队列尾部插入元素时候，AQS 队列状态为图（default), 也就是队列头尾节点都指向 null；当执行代码（1）后节点 t 指向了尾部节点，这时候队列状态如图（I）。

可知这时候 t 为 null，则执行代码（2）使用 CAS 算法设置一个哨兵节点为头结点，如果 CAS 设置成功，然后让尾部节点也指向哨兵节点，这时候队列状态如图（II）。

到现在只是插入了一个哨兵节点，还需要插入的 node 节点，所以第二次循环后执行到步骤（1），这时候队列状态如图（III）；然后执行代码（3）设置 node 的前驱节点为尾部节点，这时候队列状态图如图（IV）；然后通过 CAS 算法设置 node 节点为尾部节点，CAS 成功后队列状态图为（V）；CAS 成功后在设置原来的尾部节点的后驱节点为 node, 这时候就完成了双向链表的插入了，这时候队列状态为图（VI）。





2.2 AQS - 条件变量的支持

正如基础篇讲解的 notify 和 wait 是配合 synchronized 内置锁实现线程间同步基础设施，条件变量的 signal 和 await 方法是用来配合锁（使用 AQS 实现的锁）实现线程间同步的基础设施。

在基础篇讲解了在调用共享变量的 notify 和 wait 方法前必须先获取该共享变量的内置锁，同理在调用条件变量的 signal 和 await 方法前必须先获取条件变量对应的锁。

说了那么多，到底什么是条件变量那？如何使用那？不急，下面看一个例子：

```
ReentrantLock lock = new ReentrantLock();//(1)
Condition condition = lock.newCondition();//(2)

lock.lock();//(3)
try {
    System.out.println("begin wait");
    condition.await();//(4)
    System.out.println("end wait");

} catch (Exception e) {
    e.printStackTrace();

} finally {
```

```
        lock.unlock();//(5)
    }


    lock.lock();//(6)
    try {
        System.out.println("begin signal");
        condition.signal();//(7)
        System.out.println("end signal");
    } catch (Exception e) {
        e.printStackTrace();

    } finally {
        lock.unlock();//(8)
    }
```

如上代码（1）创建了一个独占锁 ReentrantLock 的对象，ReentrantLock 是基于 AQS 实现的锁。

代码（2）使用创建的 lock 对象的 newCondition（）方法创建了一个 ConditionObject 变量，这个变量就是 lock 锁对应的一个条件变量。需要注意的是一个 Lock 对象可以创建多个条件变量。

代码（3）首先获取了独占锁，代码（4）则调用了条件变量的 await（）方法阻塞挂起了当前线程，当其它线程调用了条件变量的 signal 方法时候，被阻塞的线程才会从 await 处返回，需要注意的是和调用 Object 的 wait 方法一样，如果在没有获取到锁前调用了条件变量的 await 方法会抛出 java.lang.IllegalMonitorStateException 异常。

代码（5）则释放了获取的锁。

其实这里的 lock 对象等价于 synchronized 加上共享变量，当调用 lock.lock（）方法就相当于进入了 synchronized 块（获取了共享变量的内置锁），当调用 lock.unLock() 方法时候就相当于退出了 synchronized 块。 当调用条件变量的 await() 方法时候就相当于调用了共享变量的 wait() 方法，当调用了条件变量的 signal 方法时候就相当于调用了共享变量的 notify() 方法。当调用了条件变量的 signalAll（）方法时候就相当于调用了共享变量的 notifyAll() 方法。

有了上面的解释相信大家对条件变量是什么，用来做什么用的有了一定的认识了。

上面通过 lock.newCondition() 作用其实是 new 了一个 AQS 内部声明的 ConditionObject 对象，ConditionObject 是 AQS 的内部类，可以访问到 AQS 内部的变量（例如状态变量 status 变量）和方法。对应每个条件变量内部维护了一个条件队列，用来存放当调用条件变量的 await() 方法被阻塞的线程。注意这个条件队列和 AQS 队列不是一回事情。

如下代码，当线程调用了条件变量的 await() 方法时候（事先必须先调用了锁的 lock() 方法获取锁），内部会构造一个类型为 Node.CONDITION 的 node 节点，然后插入该节点到条件队列末尾，然后当前线程会释放获取的锁（也就是会操作锁对应的 status 变量的值），并被阻塞挂起。这时候如果有其它线程调用了 lock.lock() 尝试获取锁时候，就会

有一个线程获取到锁，如果获取到锁的线程有调用了条件变量的 await（）方法，则该线程也会被放入条件变量的阻塞队列，然后释放获取到的锁，阻塞到 await() 方法处。

```
    public final void await() throws InterruptedException {
                if (Thread.interrupted())
                    throw new InterruptedException();
                 //创建新的node,并插入到条件队列末尾（9）
                Node node = addConditionWaiter();
                //释放当前线程获取的锁（10）
                int savedState = fullyRelease(node);
                int interruptMode = 0;
                //调用park方法阻塞挂起当前线程（11）
                while (!isOnSyncQueue(node)) {
                    LockSupport.park(this);
                    if ((interruptMode =
checkInterruptWhileWaiting(node)) != 0)
                        break;
                }
                ...
            }
```

如下代码，当另外一个线程调用了条件变量的 signal 方法时候（事先必须先调用了锁的 lock() 方法获取锁），内部会把条件队列里面队头的一个线程节点从条件队列里面移除后放入到 AQS 的阻塞队列里面，然后激活这个线程。

```
    public final void signal() {
          if (!isHeldExclusively())
              throw new IllegalMonitorStateException();
          Node first = firstWaiter;
          if (first != null)
             //移动条件队列队头元素到AQS队列
              doSignal(first);
      }
```

需要注意的是 AQS 只提供了 ConditionObject 的实现，并没有提供 newCondition 函数来 new 一个 ConditionObject 对象，需要由 AQS 的子类来提供 newCondition 函数。

下面来看下当一个线程调用条件变量的 await() 方法被阻塞后，如何放入的条件队列。

```
            private Node addConditionWaiter() {
                Node t = lastWaiter;
                ...
                //(1)
                Node node = new Node(Thread.currentThread(),
Node.CONDITION);
                //(2)
                if (t == null)
```

```
                firstWaiter = node;
            else
                t.nextWaiter = node;//(3)
            lastWaiter = node;//(4)
            return node;
        }
```

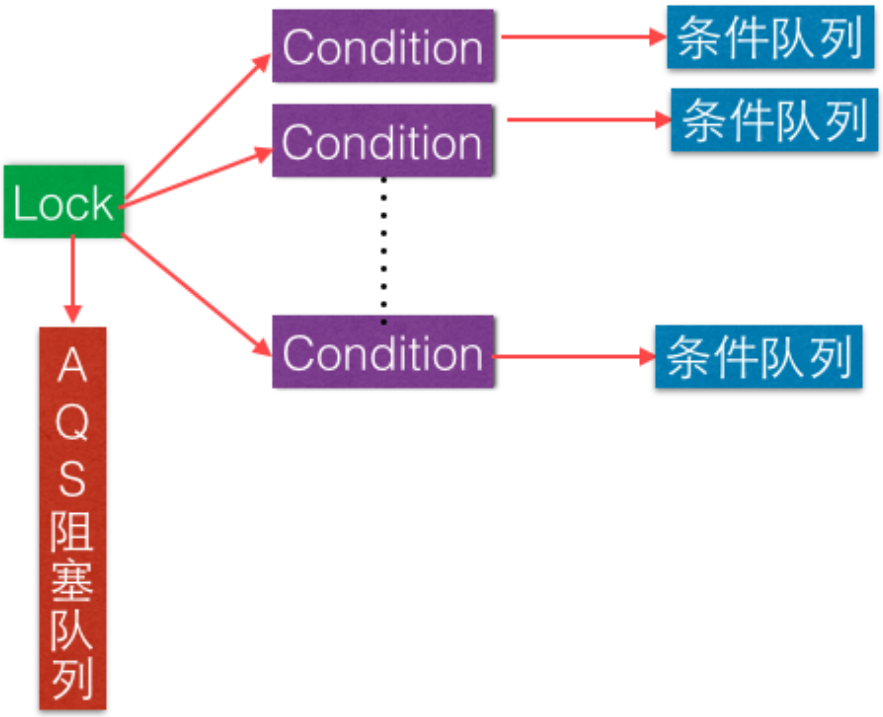
如上代码（1）首先根据当前线程创建一个类型为 Node.CONDITION 的节点，然后通过步骤（2）（3）（4）在单向条件队列尾部插入一个元素。

注：当多个线程同时调用 lock.lock() 获取锁的时候，同时只有一个线程获取到了该锁，其他线程会被转换为 Node 节点插入到 lock 锁对应的 AQS 阻塞队列里面，并做自旋 CAS 尝试获取锁；

如果获取到锁的线程又调用了对应的条件变量的 await() 方法，则该线程会释放获取到的锁，并被转换为 Node 节点插入到条件变量对应的条件队列里面；

这时候因为调用 lock.lock() 方法被阻塞到 AQS 队列里面的一个线程会获取到被释放的锁，如果该线程也调用了条件变量的 await（）方法则该线程也会被放入条件变量的条件队列。

当另外一个线程调用了条件变量的 signal() 或者 signalAll() 方法时候，会把条件队列里面的一个或者全部 Node 节点移动到 AQS 的阻塞队列里面，等待时机获取锁。

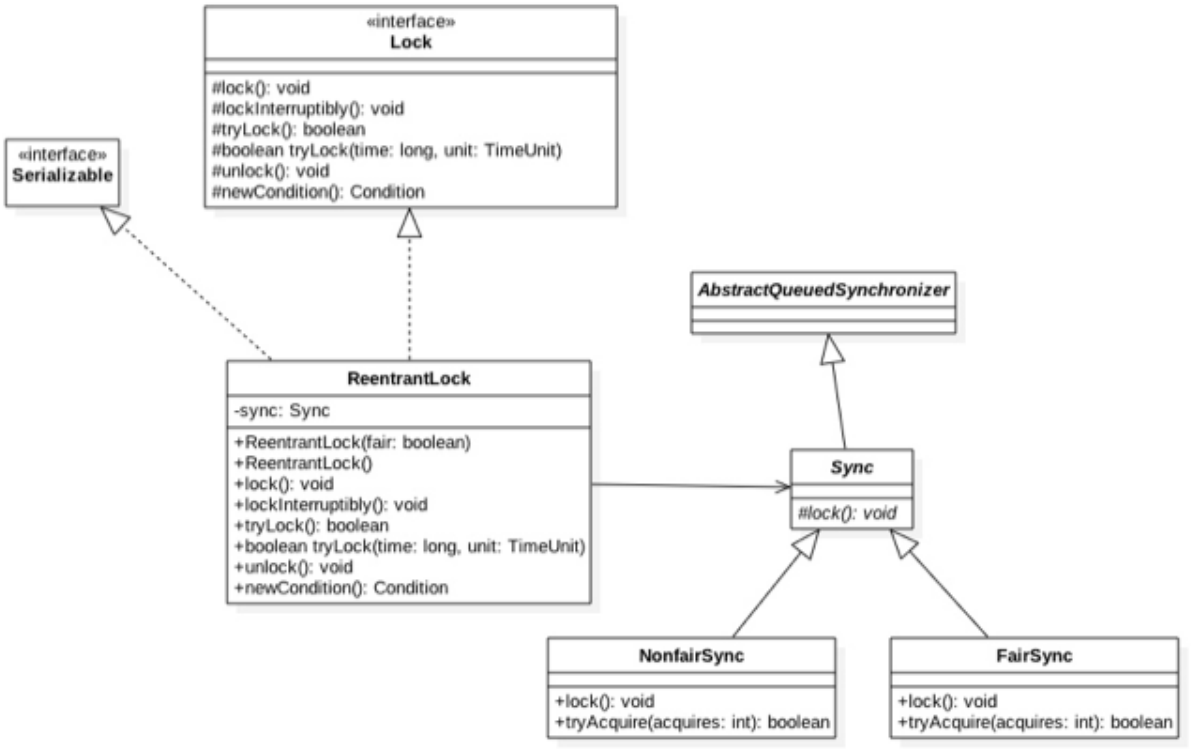


最后一个图总结下，一个锁对应有一个 AQS 阻塞队列，对应多个条件变量，每个条件变量有自己的一个条件队列。

三、独占锁 ReentrantLock 原理

3.1 类图结构简介

ReentrantLock 是可重入的独占锁，同时只能有一个线程可以获取该锁，其它获取该锁的线程会被阻塞后放入该锁的 AQS 阻塞队列里面。首先一览 ReentrantLock 的类图以便对它的实现有个大致了解



从类图可知 ReentrantLock 最终还是使用 AQS 来实现，并且根据参数决定内部是公平还是非公平锁，默认是非公平锁：

```
public ReentrantLock() {
        sync = new NonfairSync();
    }

    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

其中类 Sync 直接继承自 AQS，它的子类 NonfairSync 和 FairSync 分别实现了获取锁的公平和非公平策略。

在这里 AQS 的状态值 state 代表线程获取该锁的可重入次数，默认情况下 state 的值为 0 标示当前锁没有被任何线程持有，当一个线程第一次获取该锁时候会使用尝试使用 CAS 设置 state 的值为 1，如果 CAS 成功则当前线程获取了该锁，然后记录该锁的持有者为当前线程，在该线程没有释放锁第二次获取改锁后状态值被为 2，这就是可重入次数，在

该线程释放该锁的时候，会尝试使用 CAS 让状态值减一，如果减一后状态值为 0 则当前线程释放该锁。

3.2 获取锁

- void lock()
  当一个线程调用该方法，说明该线程希望获取该锁，如果锁当前没有被其它线程占用并且当前线程之前没有获取该锁，则当前线程会获取到该锁，然后设置当前锁的拥有者为当前线程，并设置 AQS 的状态值为 1 后直接返回。
  如果当前线程之前已经获取过该锁，则这次只是简单的把 AQS 的状态值 status 加 1 后返回。
  如果该锁已经被其它线程持有，则调用该方法的线程会被放入 AQS 队列后阻塞挂起。

```
public void lock() {
        sync.lock();
    }
```

如上代码 ReentrantLock 的 lock() 是委托给了 sync 类，根据创建 ReentrantLock 时候构造函数选择 sync 的实现是 NonfairSync 或者 FairSync，这里先看 sync 的子类 NonfairSync 的情况，也就是非公平锁的时候：

```
final void lock() {
  //（1）CAS设置状态值
  if (compareAndSetState(0, 1))
      setExclusiveOwnerThread(Thread.currentThread());
  else
  //（2）调用AQS的acquire方法
      acquire(1);
}
```

如上代码（1）因为默认 AQS 的状态值为 0，所以第一个调用 Lock 的线程会通过 CAS 设置状态值为 1，CAS 成功则表示当前线程获取到了锁，然后 setExclusiveOwnerThread 设置了该锁持有者是当前线程。

如果这时候有其它线程调用 lock 方法企图获取该锁执行代码（1）CAS 会失败，然后会调用 AQS 的 acquire 方法，这里注意传递参数为 1，这里在贴下 AQS 的 acquire 骨干代码：

```
    public final void acquire(int arg) {
        //(3)调用ReentrantLock重写的tryAcquire方法
        if (!tryAcquire(arg) &&
            // tryAcquiref返回false会把当前线程放入AQS阻塞队列
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

之前说过 AQS 并没有提供可用的 tryAcquire 方法，tryAcquire 方法需要子类自己定制化，所以这里代码（3）会调用 ReentrantLock 重写的 tryAcquire 方法代码，这里先看下非公平锁的代码如下：

```
    protected final boolean tryAcquire(int acquires) {
                return nonfairTryAcquire(acquires);
    }

   final boolean nonfairTryAcquire(int acquires) {
     final Thread current = Thread.currentThread();
     int c = getState();
     //（4）当前AQS状态值为0
     if (c == 0) {
         if (compareAndSetState(0, acquires)) {
             setExclusiveOwnerThread(current);
             return true;
         }
     }//(5)当前线程是该锁持有者
     else if (current == getExclusiveOwnerThread()) {
         int nextc = c + acquires;
         if (nextc < 0) // overflow
             throw new Error("Maximum lock count exceeded");
         setState(nextc);
         return true;
     }//(6)
     return false;
   }
```

如上代码（4）会看当前锁的状态值是否为 0，为 0 则说明当前该锁空闲，那么就尝试 CAS 获取该锁（尝试将 AQS 的状态值从 0 设置为 1），并设置当前锁的持有者为当前线程返回返回 true。

如果当前状态值不为 0 则说明该锁已经被某个线程持有，所以代码（5）看当前线程是否是该锁的持有者，如果当前线程是该锁持有者，状态值增加 1 然后返回 true。

如果当前线程不是锁的持有者则返回 false, 然后会被放入 AQS 阻塞队列。

这里介绍完了非公平锁的实现代码，回过头来看看非公平在这里是怎么体现的，首先非公平是说先尝试获取锁的线程并不一定比后尝试获取锁的线程优先获取锁。

这里假设线程 A 调用 lock（）方法时候执行到了 nonfairTryAcquire 的代码（4）发现当前状态值不为 0，所以执行代码（5）发现当前线程不是线程持有者，则执行代码（6）返回 false，然后当前线程会被放入了 AQS 阻塞队列。

这时候线程 B 也调用了 lock() 方法执行到 nonfairTryAcquire 的代码（4）时候发现当前状态值为 0 了（假设占有该锁的其它线程释放了该锁）所以通过 CAS 设置获取到了该锁。

而明明是线程 A 先请求获取的该锁那，这就是非公平锁的实现，这里线程 B 在获取锁前并没有看当前 AQS 队列里面是否有比自己请求该锁更早的线程，而是使用了抢夺策略。

那么下面看看公平锁是怎么实现公平的，公平锁的话只需要看 FairSync 重写的 tryAcquire 方法

```
        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            //（7）当前AQS状态值为0
            if (c == 0) {
             //（8）公平性策略
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            //（9）当前线程是该锁持有者
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count
exceeded");

                setState(nextc);
                return true;
            }//(10)
            return false;
        }
    }
```

如上代码公平性的 tryAcquire 策略与非公平的类似，不同在于代码（8）处在设置 CAS 前添加了 hasQueuedPredecessors 方法，该方法是实现公平性的核心代码，代码如下：

```
    public final boolean hasQueuedPredecessors() {

        Node t = tail; // Read fields in reverse initialization
order
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread !=
Thread.currentThread());
    }
```

如上代码如果当前线程节点有前驱节点则返回 true，否者如果当前 AQS 队列为空或者当前线程节点是 AQS 的第一个节点则返回 false.

其中如果 h==t 则说明当前队列为空则直接返回 false，如果 h!=t 并且 s==null 说明有一个元素将要作为 AQS 的第一个节点入队列，那么返回 true, 如果 h!=t 并且 s!=null 并且 s.thread != Thread.currentThread() 则说明队列里面的第一个元素不是当前线程则返回 true.

- void lockInterruptibly()
  与 lock() 方法类似，不同在于该方法对中断响应，就是当前线程在调用该方式时候，如果其它线程调用了当前线程线程的 interrupt（）方法，当前线程会抛出 InterruptedException 异常然后返回

```
public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
}

public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    //当前线程被中断则直接抛出异常
    if (Thread.interrupted())
        throw new InterruptedException();
    //尝试获取资源
    if (!tryAcquire(arg))
        //调用AQS可被状态的方法
        doAcquireInterruptibly(arg);
}
```

- boolean tryLock()
  尝试获取锁，如果当前该锁没有被其它线程持有则当前线程获取该锁并返回 true, 否者返回 false，注意该方法不会引起当前线程阻塞。

```
public boolean tryLock() {
    return sync.nonfairTryAcquire(1);
}

final boolean nonfairTryAcquire(int acquires) {
  final Thread current = Thread.currentThread();
  int c = getState();
  if (c == 0) {
      if (compareAndSetState(0, acquires)) {
          setExclusiveOwnerThread(current);
          return true;
      }
  }
  else if (current == getExclusiveOwnerThread()) {
      int nextc = c + acquires;
      if (nextc < 0) // overflow
```

```
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
```

如上代码与非公平锁的 tryAcquire() 方法类似，所以 tryLock() 使用的是非公平策略。

- boolean tryLock(long timeout, TimeUnit unit)
  尝试获取锁与 tryLock（）不同在于设置了超时时间，如果超时没有获取该锁则返回 false。

```
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
            //调用AQS的tryAcquireNanos方法。
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
```

3.3 释放锁

- void unlock()
  尝试释放锁，如果当前线程持有该锁，调用该方法会让该线程对该线程持有的 AQS 状态值减一，如果减去 1 后当前状态值为 0 则当前线程会释放对该锁的持有，否者仅仅减一而已。如果当前线程没有持有该锁调用了该方法则会抛出 IllegalMonitorStateException 异常，代码如下：

```
    public void unlock() {
        sync.release(1);
    }

    protected final boolean tryRelease(int releases) {
        //(11)如果不是锁持有者调用UNlock则抛出异常。
        int c = getState() - releases;
        if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
        boolean free = false;
        //(12)如果当前可重入次数为0，则清空锁持有线程
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
        //(13)设置可重入次数为原始值-1
        setState(c);
```

```
            return free;
        }
```

如上代码（11）如果当前线程不是该锁持有者则直接抛出异常，否者看状态值剩余值是否为 0，为 0 则说明当前线程要释放对该锁的持有权，则执行（12）把当前锁持有者设置为 null。如果剩余值不为 0，则仅仅让当前线程对该锁的可重入次数减一。

3.4 案例介绍

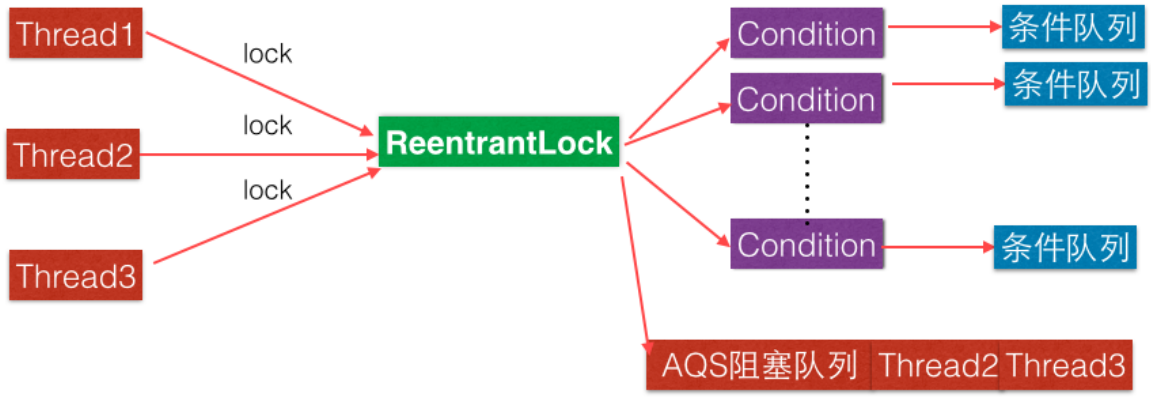
下面使用 ReentrantLock 来实现一个简单的线程安全的 list：

```
public static class ReentrantLockList {

        //线程不安全的list
        private ArrayList<String> array = new ArrayList<String>
();
        //独占锁
        private volatile ReentrantLock lock = new
ReentrantLock();

        //添加元素
        public void add(String e) {

            lock.lock();
            try {
                array.add(e);

            } finally {
                lock.unlock();

            }
        }
        //删元素
        public void remove(String e) {

            lock.lock();
            try {
                array.remove(e);

            } finally {
                lock.unlock();

            }
        }

        //获取数据
        public String get(int index) {

            lock.lock();
```

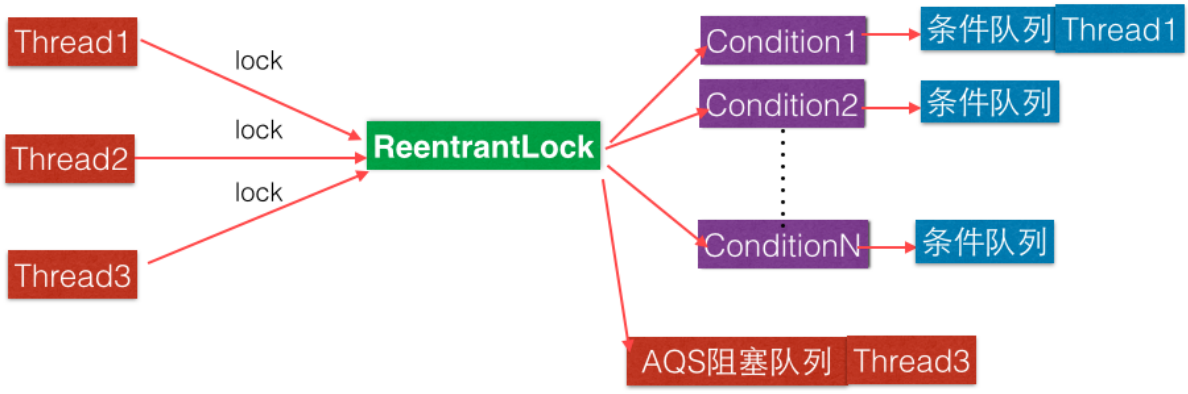
```
            try {
                return array.get(index);

            } finally {
                lock.unlock();

            }
        }
```

如上代码通过在操作 array 元素前进行加锁保证同时只有一个线程可以对 array 数组进行修改，但是同时也只能有一个线程对 array 元素进行访问。

同理最后使用几个图来加深理解：



如上图，假如线程 Thread1,Thread2,Thread3 同时尝试获取独占锁 ReentrantLock，假设 Thread1 获取到了，则 Thread2 和 Thread3 就会被转换为 Node 节点后放入 ReentrantLock 对应的 AQS 阻塞队列后阻塞挂起。



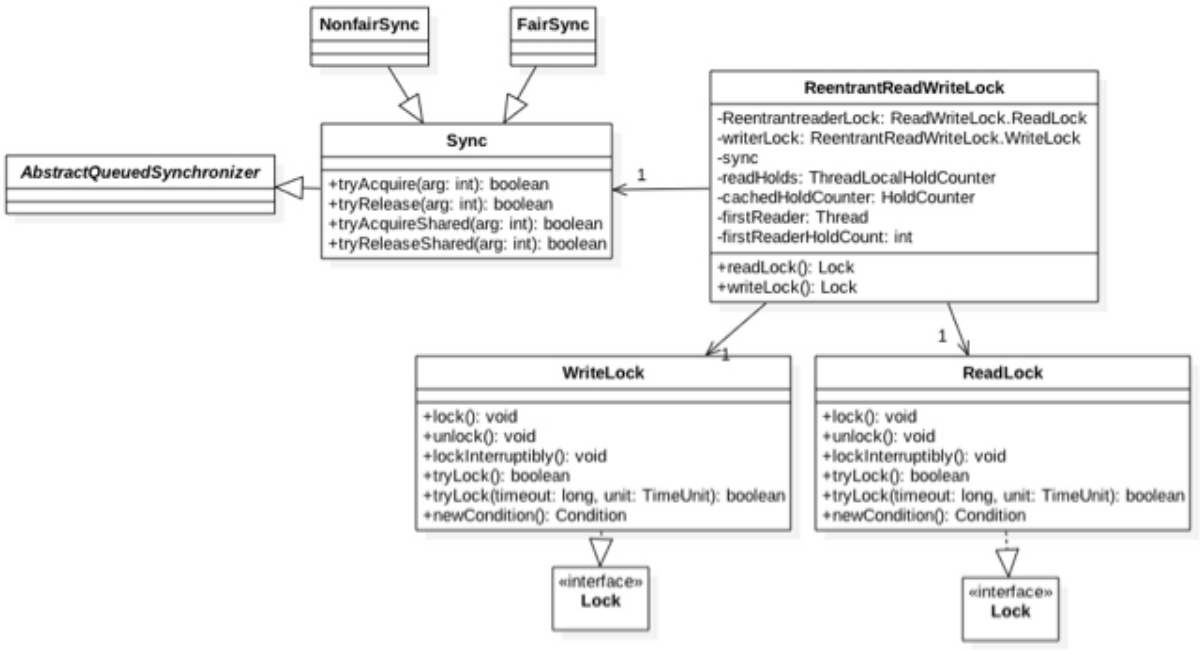
如上图，假设 Thread1 获取锁后调用了对应的锁创建的条件变量 1，那么 Thread1 就会释放获取到的锁，然后当前线程就会被转换为 Node 节点后插入到条件变量 1 的条件队列，由于 Thread1 释放了锁，所以阻塞到 AQS 队列里面 Thread2 和 Thread3 就有机会获取到该锁，假如使用的公平策略，那么这时候 Thread2 会获取到该锁，会从 AQS 队列里面移除 Thread2 对应的 Node 节点。

四、读写锁 ReentrantReadWriteLock 原理

在解决线程安全问题上使用 ReentrantLock 就可以，但是 ReentrantLock 是独占锁，同时只有一个线程可以获取该锁，而实际情况下会有写少读多的场景，显然 ReentrantLock 满足不了需求，所以 ReentrantReadWriteLock 应运而生，ReentrantReadWriteLock 采用读写分离，多个线程可以同时获取读锁。

4.1 类图结构介绍

为了一览 ReentrantReadWriteLock 内部构造先看下它的类图结构：



如图读写锁内部维护了一个 ReadLock 和 WriteLock，并且也提供了公平和非公平的实现，下面只介绍下非公平的读写锁实现。我们知道 AQS 里面只维护了一个 state 状态，而 ReentrantReadWriteLock 则需要维护读状态和写状态，一个 state 是无法表示写和读状态的。ReentrantReadWriteLock 巧妙的使用 state 的高 16 位表示读状态也就是获取该读锁的线程个数，低 16 位表示获取到写锁的线程的可重入次数。

```
static final int SHARED_SHIFT   = 16;

//共享锁（读锁）状态单位值65536
static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
//共享锁线程最大个数65535
static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;

//排它锁(写锁)掩码 二进制 15个1
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

/** 返回读锁线程数  */
static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
```

```
/** 返回写锁可重入个数 */
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

类图中 firstReader 用来记录第一个获取到读锁的线程，firstReaderHoldCount 则记录第一个获取到读锁的线程获取读锁的可重入数。cachedHoldCounter 用来记录最后一个获取读锁的线程获取读锁的可重入次数：

```
static final class HoldCounter {
    int count = 0;
    //线程id
    final long tid = getThreadId(Thread.currentThread());
}
```

readHolds 是 ThreadLocal 变量，用来存放除去第一个获取读锁线程外的其它线程获取读锁的可重入数, 可知 ThreadLocalHoldCounter 继承了 ThreadLocal，里面 initialValue 方法返回一个 HoldCounter 对象。

```
static final class ThreadLocalHoldCounter
    extends ThreadLocal<HoldCounter> {
    public HoldCounter initialValue() {
        return new HoldCounter();
    }
}
```

4.1.1 写锁的获取与释放

ReentrantReadWriteLock 中写锁是使用的 WriteLock 来实现的。

- void lock()
  写锁是个独占锁，同时只有一个线程可以获取该锁。
  如果当前没有线程获取到读锁和写锁则当前线程可以获取到写锁然后返回。
  如果当前已经有线程取到读锁和写锁则当前线程则当前请求写锁线程会被阻塞挂起。
  另外写锁是可重入锁，如果当前线程已经获取了该锁，再次获取的只是简单的把可重入次数加一后直接返回。

```
public void lock() {
    sync.acquire(1);
}
```

```
 public final void acquire(int arg) {
     // sync重写的tryAcquire方法
     if (!tryAcquire(arg) &&
         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
```

```
            selfInterrupt();
        }
```

如上代码 lock() 内部调用了 AQS 的 acquire 方法，其中的 tryAcquire 是 ReentrantReadWriteLock 内部 sync 类重写的，代码如下：

```
    protected final boolean tryAcquire(int acquires) {

                Thread current = Thread.currentThread();
                int c = getState();
                int w = exclusiveCount(c);
                //（1） c!=0说明读锁或者写锁已经被某线程获取
                if (c != 0) {
                     （2）//w=0说明已经有线程获取了读锁或者w!=0并且当前线程不
    是写锁拥有者，则返回false
                    if (w == 0 || current !=
    getExclusiveOwnerThread())
                        return false;
                    （3）//说明某线程获取了写锁，判断可重入个数
                    if (w + exclusiveCount(acquires) > MAX_COUNT)
                        throw new Error("Maximum lock count
    exceeded");

                    （4）// 设置可重入数量(1)
                    setState(c + acquires);
                    return true;
                }

                （5）//第一个写线程获取写锁
                if (writerShouldBlock() ||
                    !compareAndSetState(c, c + acquires))
                    return false;
                setExclusiveOwnerThread(current);
                return true;
            }
```

如上代码（1）如果当前 AQS 状态值不为 0 则说明当前已经有线程获取到了读锁或者写锁，代码（2）如果 w==0 说明状态值的低 16 位为 0，而状态值不为 0，则说明高 16 位不为 0，这暗示已经有线程获取了读锁，所以直接返回 false。

如果 w!=0 说明当前已经有线程获取了该写锁，则看当前线程是不是该锁的持有者，如果不是则返回 false。

执行到代码（3）说明当前线程之前已经获取到了该锁，则判断该线程的可重入此时是不是超过了最大值，是则抛出异常，否者执行点（4）增加当前线程的可重入次数然后返回 true.

如果 AQS 的状态值等于 0 则说明目前没有线程获取到读锁和写锁，则执行代码（5），其中对于 writerShouldBlock 方法非公平锁的实现为：

```
final boolean writerShouldBlock() {
    return false; // writers can always barge
}
```

如代码对于非公平锁来说固定返回 false，则说明代码（5）抢占式执行 CAS 尝试获取写锁，获取成功则设置当前锁的持有者为当前线程返回 true, 否者返回 false。

对于公平锁的实现为：

```
final boolean writerShouldBlock() {
  return hasQueuedPredecessors();
}
```

可知还是使用 hasQueuedPredecessors 来判断当前线程节点是否有前驱节点，如果有则当前线程放弃获取写锁的权限直接返回 false。

- void lockInterruptibly()
  类似 lock() 方法，不同在于该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

```
        public void lockInterruptibly() throws
InterruptedException {
            sync.acquireInterruptibly(1);
        }
```

- boolean tryLock()
  尝试获取写锁，如果当前没有其它线程持有写锁或者读锁，则当前线程获取写锁会成功，然后返回 true
  如果当前已经其它线程持有写锁或者读锁则该方法直接返回 false，当前线程并不会被阻塞。
  如果当前线程已经持有了该写锁则简单增加 AQS 的状态值后直接返回 true

```
public boolean tryLock( ) {
    return sync.tryWriteLock();
}

final boolean tryWriteLock() {
    Thread current = Thread.currentThread();
    int c = getState();
    if (c != 0) {
        int w = exclusiveCount(c);
```

```
            if (w == 0 || current != getExclusiveOwnerThread())
                return false;
            if (w == MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
        }
        if (!compareAndSetState(c, c + 1))
            return false;
        setExclusiveOwnerThread(current);
        return true;
    }
```

如上代码与 tryAcquire 方法类似这里不再讲述，不同在于这里使用的非公平策略。

- boolean tryLock(long timeout, TimeUnit unit)
  与 tryAcquire（）不同在于多了超时时间的参数，如果尝试获取写锁失败则会把当前线程挂起指定时间，待超时时间到后当前线程被激活，如果还是没有获取到写锁则返回 false。另外该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

```
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
```

- void unlock()
  尝试释放锁，如果当前线程持有该锁，调用该方法会让该线程对该线程持有的 AQS 状态值减一，如果减去 1 后当前状态值为 0 则当前线程会释放对该锁的持有，否者仅仅减一而已。如果当前线程没有持有该锁调用了该方法则会抛出 IllegalMonitorStateException 异常，代码如下：

```
    public void unlock() {
        sync.release(1);
    }

    public final boolean release(int arg) {
    //调用ReentrantReadWriteLock中sync实现的tryRelease方法
    if (tryRelease(arg)) {
        //激活阻塞队列里面的一个线程
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

```
        protected final boolean tryRelease(int releases) {
            //（6） 看是否是写锁拥有者调用的unlock
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            //（7）获取可重入值，这里没有考虑高16位，因为写锁时候读锁状态值
肯定为0
            int nextc = getState() - releases;
            boolean free = exclusiveCount(nextc) == 0;
            //（8）如果写锁可重入值为0则释放锁，否者只是简单更新状态值。
            if (free)
                setExclusiveOwnerThread(null);
            setState(nextc);
            return free;
        }
```

如上代码 tryRelease 首先通过 isHeldExclusively 判断是否当前线程是该写锁的持有者，如果不是则抛出异常，否者执行代码（7）说明当前线程持有写锁，持有写锁说明状态值的高 16 位为 0，所以这里 nextc 值就是当前线程写锁的剩余可重入次数。

代码（8）判断当前可重入次数是否为 0，如果 free 为 true 说明可重入次数为 0，则当前线程会释放对写锁的持有，当前锁的持有者设置为 null。如果 free 为 false 则简单更新可重入次数。

4.1.2 读锁的获取与释放

ReentrantReadWriteLock 中写锁是使用的 ReadLock 来实现的。

- void lock()
  获取读锁，如果当前没有其它线程持有写锁，则当前线程可以获取读锁，AQS 的高 16 位的值会增加 1，然后方法返回。否者如果其它有一个线程持有写锁，则当前线程会被阻塞。

```
    public void lock() {
        sync.acquireShared(1);
    }

     public final void acquireShared(int arg) {
         //调用ReentrantReadWriteLock中的sync的tryAcquireShared方法
         if (tryAcquireShared(arg) < 0)
             //调用AQS的doAcquireShared方法
             doAcquireShared(arg);
     }
```

如上代码读锁的 lock 方法调用了 AQS 的 acquireShared 方法，内部调用了 ReentrantReadWriteLock 中的 sync 重写的 tryAcquireShared 方法代码如下：

```
protected final int tryAcquireShared(int unused) {

    //(1)获取当前状态值
     Thread current = Thread.currentThread();
     int c = getState();

     //(2)判断是否写锁被占用
     if (exclusiveCount(c) != 0 &&
         getExclusiveOwnerThread() != current)
         return -1;

     //（3）获取读锁计数
     int r = sharedCount(c);
     //（4）尝试获取锁，多个读线程只有一个会成功，不成功的进入下面
fullTryAcquireShared进行重试
     if (!readerShouldBlock() &&
         r < MAX_COUNT &&
         compareAndSetState(c, c + SHARED_UNIT)) {
         //(5)第一个线程获取读锁
         if (r == 0) {
             firstReader = current;
             firstReaderHoldCount = 1;
         //(6)如果当前线程是第一个获取读锁的线程
         } else if (firstReader == current) {
             firstReaderHoldCount++;
         } else {
             //（7）记录最后一个获取读锁的线程或记录其它线程读锁的可重入数
             HoldCounter rh = cachedHoldCounter;
             if (rh == null || rh.tid != current.getId())
                 cachedHoldCounter = rh = readHolds.get();
             else if (rh.count == 0)
                 readHolds.set(rh);
             rh.count++;
         }
         return 1;
     }
     //(8)类似tryAcquireShared，但是是自旋获取
     return fullTryAcquireShared(current);
}
```

如上代码首先获取了当前 AQS 的状态值，然后代码（2）看是否有其它线程获取到了写锁，如果是则直接返回了 - 1，然后调用 AQS 的 doAcquireShared 方法把当前线程放入阻塞队列。

否者执行到代码（3）得到获取到读锁的线程个数，到这里说明目前没有线程获取到写锁，但是还是可能有线程持有读锁，然后执行代码（4），非公平锁的 readerShouldBlock 实现代码如下：

```
    final boolean readerShouldBlock() {
         return apparentlyFirstQueuedIsExclusive();
    }

    final boolean apparentlyFirstQueuedIsExclusive() {
    Node h, s;
    return (h = head) != null &&
        (s = h.next)  != null &&
        !s.isShared()         &&
        s.thread != null;
     }
```

如上代码作用是如果队列里面存在一个元素，则判断第一个元素是不是正在尝试获取写锁，如果不是的话，则当前线程使用判断当前获取读锁线程是否达到了最大值，最后执行 CAS 操作设置 AQS 状态值的高 16 位值增加 1。

代码（5）（6）记录第一个获取读锁的线程并统计该线程获取读锁的可重入数，代码（7）使用 cachedHoldCounter 记录最后一个获取到读锁的线程并同时该线程获取读锁的可重入数，另外 readHolds 记录了当前线程获取读锁的可重入数。

如果 readerShouldBlock 返回 true 则说明有线程正在获取写锁，则执行代码（8）fullTryAcquireShared 代码与 tryAcquireShared 类似，不同在于前者是通过循环自旋获取。

```
final int fullTryAcquireShared(Thread current) {
                     HoldCounter rh = null;
            for (;;) {
                int c = getState();
                if (exclusiveCount(c) != 0) {
                    if (getExclusiveOwnerThread() != current)
                        return -1;
                    // else we hold the exclusive lock; blocking
here
                    // would cause deadlock.
                } else if (readerShouldBlock()) {
                    // Make sure we're not acquiring read lock
reentrantly
                    if (firstReader == current) {
                        // assert firstReaderHoldCount > 0;
                    } else {
                        if (rh == null) {
                            rh = cachedHoldCounter;
                            if (rh == null || rh.tid !=
getThreadId(current)) {
                                rh = readHolds.get();
                                if (rh.count == 0)
                                    readHolds.remove();
                            }
```

```
                        }
                        if (rh.count == 0)
                            return -1;
                    }
                }
                if (sharedCount(c) == MAX_COUNT)
                    throw new Error("Maximum lock count
exceeded");
                if (compareAndSetState(c, c + SHARED_UNIT)) {
                    if (sharedCount(c) == 0) {
                        firstReader = current;
                        firstReaderHoldCount = 1;
                    } else if (firstReader == current) {
                        firstReaderHoldCount++;
                    } else {
                        if (rh == null)
                            rh = cachedHoldCounter;
                        if (rh == null || rh.tid !=
getThreadId(current))
                            rh = readHolds.get();
                        else if (rh.count == 0)
                            readHolds.set(rh);
                        rh.count++;
                        cachedHoldCounter = rh; // cache for
release
                    }
                    return 1;
                }
            }
        }
```

- void lockInterruptibly()
  类似 lock() 方法，不同在于该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

- boolean tryLock()
  尝试获取读锁，如果当前没有其它线程持有写锁，则当前线程获取写锁会成功，然后返回 true

如果当前已经其它线程持有写锁则该方法直接返回 false，当前线程并不会被阻塞。

如果其它获取当前线程已经持有了该读锁则简单增加 AQS 的状态值高 16 位后直接返回 true。代码类似 tryLock 这里不再讲述。

- boolean tryLock(long timeout, TimeUnit unit)
  与 tryLock（）不同在于多了超时时间的参数，如果尝试获取读锁失败则会把当前线程挂起指定时间，待超时时间到后当前线程被激活，如果还是没有获取到读锁则返回 false。另外该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

- void unlock()

```
        public void unlock() {
            sync.releaseShared(1);
        }

        public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }
```

```
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    //如果当前线程是第一个获取读锁线程
    if (firstReader == current) {
        //如果可重入次数为1
        if (firstReaderHoldCount == 1)
            firstReader = null;
        else//否者可重入次数减去1
            firstReaderHoldCount--;
    } else {
        //如果当前线程不是最后一个获取读锁线程，则从threadlocal里面获取
        HoldCounter rh = cachedHoldCounter;
        if (rh == null || rh.tid != current.getId())
            rh = readHolds.get();
        //如果可重入次数<=1则清除threadlocal
        int count = rh.count;
        if (count <= 1) {
            readHolds.remove();
            if (count <= 0)
                throw unmatchedUnlockException();
        }
        //可重入次数减去一
        --rh.count;
    }

    //循环直到自己的读计数-1 cas更新成功
    for (;;) {
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))

            return nextc == 0;
    }
}
```

4.1.3 案例介绍

上节介绍了使用 ReentrantLock 实现的线程安全的 list, 但是由于 ReentrantLock 是独占锁所以在读多写少的情况下性能很差，下面使用 ReentrantReadWriteLock 来改造为如下代码：

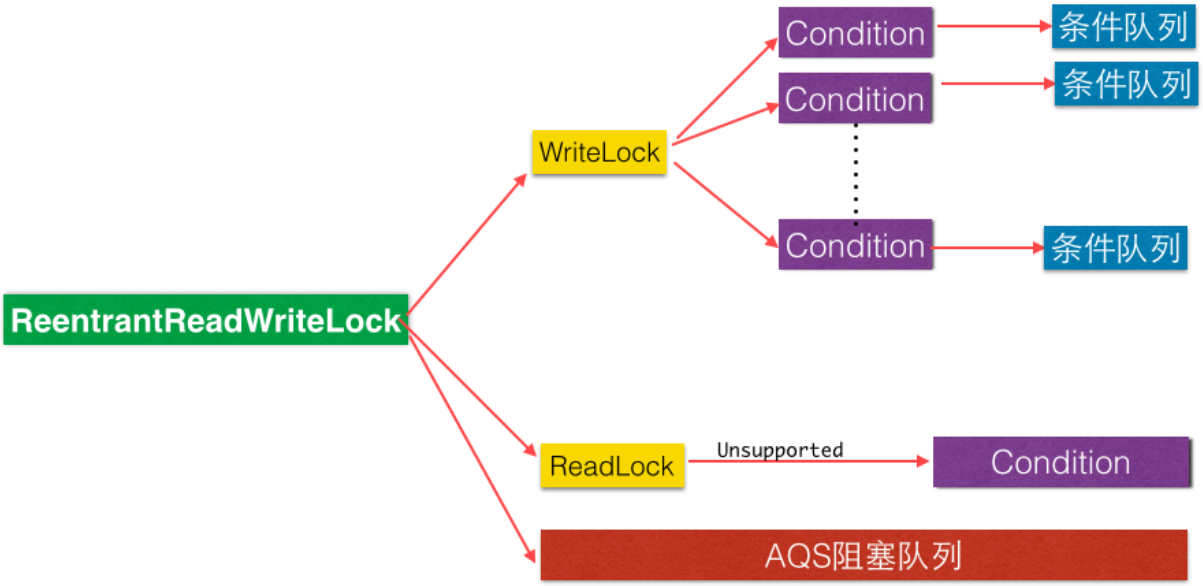
```
public static class ReentrantLockList {

        //线程不安全的list
        private ArrayList<String> array = new ArrayList<String>
();
        //独占锁
        private final  ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
        private final Lock readLock = lock.readLock();
        private final Lock writeLock = lock.writeLock();

        //添加元素
        public void add(String e) {

            writeLock.lock();
            try {
                array.add(e);

            } finally {
                writeLock.unlock();

            }
        }
        //删元素
        public void remove(String e) {

            writeLock.lock();
            try {
                array.remove(e);

            } finally {
                writeLock.unlock();

            }
        }

        //获取数据
        public String get(int index) {

            readLock.lock();
            try {
                return array.get(index);

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

```
                }
            }

        }
```

如代码调用 get 方法适合使用的是读锁，这样运行多个读线程同时访问 list 的元素，在读多写少的情况下性能相比 ReentrantLock 会很好。

注：本节介绍了读写锁 ReentrantReadWriteLock 的原理，可知底层还是使用 AQS 实现的，ReentrantReadWriteLock 巧妙的使用 AQS 的状态值的高 16 位表示获取读锁的线程个数，低 16 位表示获取写锁的线程的可重入次数，并通过 CAS 对其进行操作实现了读写分离，在读多写少的场景下比较适用。

最后一张图加深对 ReentrantReadWriteLock 的理解：



4.2 JDK8 新增的 StampedLock 锁探究

StampedLock 是并发包里面 jdk8 版本新增的一个锁，该锁提供了三种模式的读写控制，当调用获取锁的系列函数时候，会返回一个 long 型的变量，我们称之为戳记（stamp），这个戳记代码了锁的状态。其中 try 系列获取锁的函数，当获取锁失败后会返回为 0 的 stamp 值。当调用释放锁和转换锁的方法时候需要传入获取锁时候返回的 stamp 值。

StampedLock 提供的三种读写模式的锁分别如下：

- 写锁 writeLock，是个排它锁或者叫独占锁，同时只有一个线程可以获取该锁，当一个线程获取该锁后，其它请求读锁和写锁的线程必须等待，类似于 ReentrantReadWriteLock 的写锁（不同在于这里的写锁是不可重入锁）；当目前没

有线程持有读锁或者写锁的时候才可以获取到该锁，请求该锁成功后会返回一个 stamp 票据变量用来表示该锁的版本，当释放该锁时候需要调用 unlockWrite 方法并传递获取锁时候的 stamp 参数。并且提供了非阻塞的 tryWriteLock 方法。

- 悲观读锁 readLock，是个共享锁，在没有线程获取独占写锁的情况下，同时多个线程可以获取该锁; 如果已经有线程持有写锁，其它线程请求获取该读锁会被阻塞，这类似 ReentrantReadWriteLock 的读锁（不同在于这里的读锁是不可重入锁）。这里说的悲观是指在具体操作数据前悲观的认为其它线程可能要对自己操作的数据进行修改，所以需要先对数据加锁，这是在读少写多的情况下的一种考虑, 请求该锁成功后会返回一个 stamp 票据变量用来表示该锁的版本，当释放该锁时候需要 unlockRead 并传递参数 stamp。并且提供了非阻塞的 tryReadLock。

- 乐观读锁 tryOptimisticRead，是相对于悲观锁来说的，在操作数据前并没有通过 CAS 设置锁的状态，仅仅是通过位运算测试；如果当前没有线程持有写锁，则简单的返回一个非 0 的 stamp 版本信息，获取该 stamp 后在具体操作数据前还需要调用 validate 验证下该 stamp 是否已经不可用，也就是看当调用 tryOptimisticRead 返回 stamp 后，到当前时间间是否有其它线程持有了写锁，如果是那么 validate 会返回 0，否者就可以使用该 stamp 版本的锁对数据进行操作。由于 tryOptimisticRead 并没有使用 CAS 设置锁状态，所以不需要显示的释放该锁。该锁的一个特点是适用于读多写少的场景，因为获取读锁只是使用位操作进行检验，不涉及 CAS 操作，所以效率会高很多，但是同时由于没有使用真正的锁，在保证数据一致性上需要拷贝一份要操作的变量到方法栈，并且在操作数据时候可能其它写线程已经修改了数据，而我们操作的是方法栈里面的数据，也就是一个快照，所以最多返回的不是最新的数据，但是一致性还是得到保障的。

StampedLock 还支持这三种锁在一定条件下进行相互转换：
例如 `long tryConvertToWriteLock(long stamp)` 期望把 stamp 标示的锁升级为写锁，这个函数会在下面几种情况下返回一个有效的 stamp（也就是晋升写锁成功）：

1）当前锁已经是写锁模式了。

2）当前锁处于读锁模式，并且没有其他线程是读锁模式

3）当前处于乐观读模式，并且当前写锁可用。

另外 StampedLock 的读写锁都是不可重入锁，所以当获取锁后释放锁前不应该在调用会获取锁的操作，以避免产生死锁。当多个线程同时尝试获取读锁和写锁时候，谁写获取锁没有一定的规则，完全都是尽力而为，是随机的。并且该锁不是直接实现 Lock 或 ReadWriteLock 接口，而是内部自己维护了一个双向阻塞队列。

下面通过 JDK8 里面提供的一个管理二维点的例子讲解来加深对上面讲解的理解。

```
class Point {

    // 成员变量
    private double x, y;
```

```
    // 锁实例
    private final StampedLock sl = new StampedLock();

    // 排它锁-写锁（writeLock）
    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    // 乐观读锁（tryOptimisticRead）
    double distanceFromOrigin() {

        // 尝试获取乐观读锁（1）
        long stamp = sl.tryOptimisticRead();
        // 将全部变量拷贝到方法体栈内（2）
        double currentX = x, currentY = y;
        // 检查在（1）获取到读锁票据后，锁有没被其它写线程排它性抢占（3）
        if (!sl.validate(stamp)) {
            // 如果被抢占则获取一个共享读锁（悲观获取）（4）
            stamp = sl.readLock();
            try {
                // 将全部变量拷贝到方法体栈内（5）
                currentX = x;
                currentY = y;
            } finally {
                // 释放共享读锁（6）
                sl.unlockRead(stamp);
            }
        }
        // 返回计算结果（7）
        return Math.sqrt(currentX * currentX + currentY *
currentY);
    }

    // 使用悲观锁获取读锁，并尝试转换为写锁
    void moveIfAtOrigin(double newX, double newY) {
        // 这里可以使用乐观读锁替换（1）
        long stamp = sl.readLock();
        try {
            // 如果当前点在原点则移动（2）
            while (x == 0.0 && y == 0.0) {
                // 尝试将获取的读锁升级为写锁（3）
                long ws = sl.tryConvertToWriteLock(stamp);
                // 升级成功，则更新票据，并设置坐标值，然后退出循环（4）
                if (ws != 0L) {
                    stamp = ws;
                    x = newX;
```

```
                    y = newY;
                    break;
                } else {
                    // 读锁升级写锁失败则释放读锁，显示获取独占写锁，然后
循环重试（5）
                    sl.unlockRead(stamp);
                    stamp = sl.writeLock();
                }
            }
        } finally {
            // 释放锁（6）
            sl.unlock(stamp);
        }
    }
}
```

如上代码 Point 类里面有两个成员变量（x,y) 来标示一个点的二维坐标，和三个操作坐标变量的方法，另外实例化了一个 StampedLock 对象用来保证操作的原子性。

首先分析下 move 方法，该函数作用是使用参数的增量值，改变当前 point 坐标的位置；代码先获取到了写锁，然后对 point 坐标进行修改，然后释放锁。该锁是排它锁，这保证了其它线程调用 move 函数时候会被阻塞，也保证了其它线程不能获取读锁，读取坐标的值，直到当前线程显示释放了写锁，也就是保证了对变量 x,y 操作的原子性和数据一致性。

然后看下 distanceFromOrigin 方法，该方法作用是计算当前位置到原点（坐标为 0,0）的距离，代码（1）首先尝试获取乐观读锁，如果当前没有其它线程获取到了写锁，那么（1）会返回一个非 0 的 stamp 用来表示版本信息，代码（2）拷贝坐标变量到本地方法栈里面。

代码（3）检查在（1）获取到的 stamp 值是否还有效，之所以还要在此校验是因为代码（1）获取读锁时候并没有通过 CAS 操作修改锁的状态，而是简单的通过与或操作返回了一个版本信息，这里校验是看在在获取版本信息到现在的时间段里面是否有其它线程持有了写锁，如果有则之前获取的版本信息就无效了。

这里如果校验成功则执行（7）使用本地方法栈里面的值进行计算然后返回。需要注意的是在代码（3) 校验成功后，代码（7）计算期间，其它线程可能获取到了写锁并且修改了 x,y 的值，而当前线程执行代码（7）进行计算时候采用的还是修改前值的拷贝，也就是操作的值是对之前值的一个拷贝，一个快照，并不是最新的值。

另外还有个问题，代码（2) 和（3）能否互换，答案是不能。假设位置换了，那么首先执行 validate，假如验证通过了，要拷贝 x,y 值到本地方法栈，而在拷贝的过程中很有可能其它线程已经修改了 x,y 中的一个，这就造成了数据的不一致性了。那么你可能会问，即使不交换代码 (2) 和（3），在拷贝 x,y 值到本地方法栈里面时，也会存在其它线程修改了 x,y 中的一个值那，这不也会存在问题？这个确实会存在，但是，别忘了拷贝后还有一道 validate, 如果这时候有线程修改了 x,y 中的值，那么肯定是有线程在调用 validate 前，调用 sl.tryOptimisticRead 后获取了写锁，那么进行 validate 时候就会失败。

现在应该明白了吧，这也是乐观读设计的精妙之处也是使用时候容易出问题的地方。下面继续分析 validate 失败后会执行代码（4）获取悲观读锁，如果这时候其他线程持有写锁则代码（4）会导致的当前线程阻塞直到其它线程释放了写锁。如果这时候没有其他线程获取到写锁，那么当前线程就可以获取到读锁，然后执行代码（5）重新拷贝新的坐标值到本地方法栈，然后就是代码（6）释放了锁，拷贝的时候由于加了读锁，所以拷贝期间其它线程获取写锁时候会被阻塞，这保证了数据的一致性，另外这里 x,y 没有被声明为 volatie，会不会存在内存不可见性问题那？答案是不会，因为加锁的语义保证了内存可见性，关系内存可见性问题可以参考 chat(http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f)。

最后代码（7）使用方法栈里面数据计算返回，同理这里在计算时候使用的数据也可能不是最新的，其它写线程可能已经修改过原来的 x,y 值了。

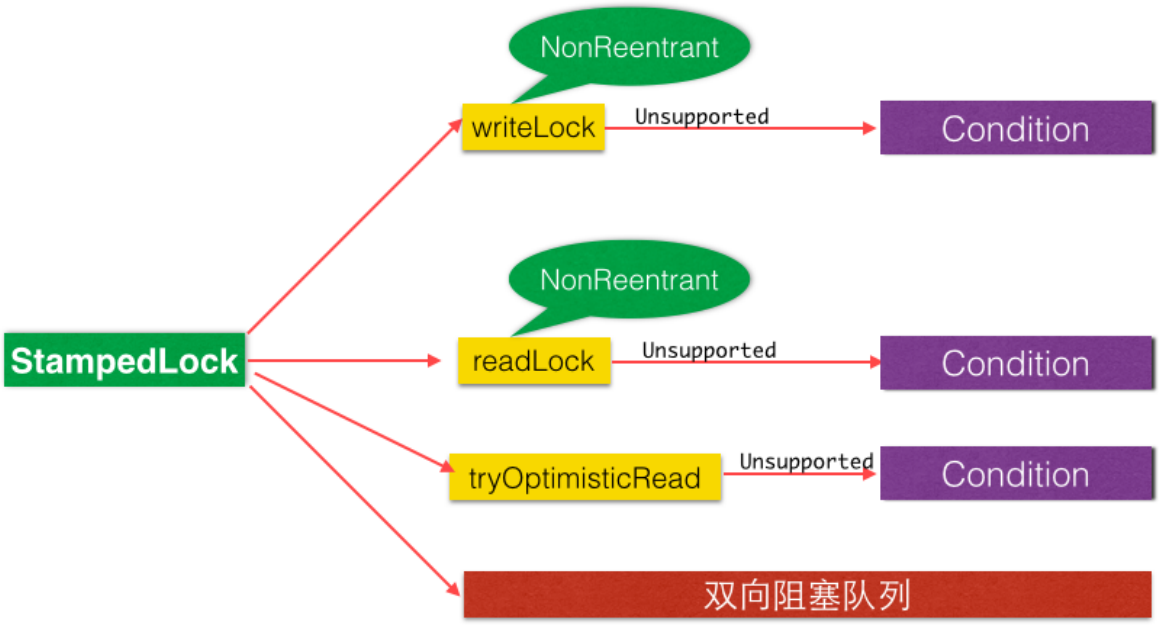
最后一个方法 moveIfAtOrigin 作用是如果当前坐标为原点则移动到指定的位置。代码（1）获取悲观读锁，保证其它线程不能获取写锁修改 x,y 值，然后代码（2）判断如果当前点在原点则更新坐标，代码（3) 尝试升级读锁为写锁，这里升级不一定成功，因为多个线程都可以同时获取悲观读锁，当多个线程都执行到（3）时候只有一个可以升级成功，升级成功则返回非 0 的 stamp，否非返回 0。这里假设当前线程升级成功，然后执行步骤（4）更新 stamp 值和坐标值，然后退出循环。如果升级失败则执行步骤（5）首先释放读锁然后申请写锁，获取到写锁后在循环重新设置坐标值。最后步骤（6) 释放锁。

使用乐观读锁还是很容易犯错误的，必须要小心，必须要保证如下的使用顺序：

```
long stamp = lock.tryOptimisticRead(); //非阻塞获取版本信息
copyVaraibale2ThreadMemory();//拷贝变量到线程本地堆栈
if(!lock.validate(stamp)){ // 校验
    long stamp = lock.readLock();//获取读锁
    try {
        copyVaraibale2ThreadMemory();//拷贝变量到线程本地堆栈
     } finally {
       lock.unlock(stamp);//释放悲观锁
    }

}

useThreadMemoryVarables();//使用线程本地堆栈里面的数据进行操作
```

总结： StampedLock 提供的读写锁与 ReentrantReadWriteLock 类似，只是前者的都是不可重入锁。但是前者通过提供乐观读锁在多线程多读的情况下提供更好的性能，这是因为获取乐观读锁时候不需要进行 CAS 操作设置锁的状态，而只是简单的测试状态。

最后通过一个图来一览 StampedLock 的组成：



六、总结

本文首先对 AQS 进行概述，讲述了 AQS 做为同步器的基础设施，为子类抽象了哪些能力。然后讲解了独占锁 ReentrantLock 原理探究，讲解了如何基于 AQS 的能力实现了公平与非公平的独占锁，然后讲解了读写锁 ReentrantReadWriteLock 原理，ReentrantReadWriteLock 使用读写分离锁，在读多写少的情景下的比较适用。最后讲解了 JDK8 新增的 StampedLock 锁原理探究，该锁提供了三种模式的读写控制。锁是 JUC 包中最难理解的一块，希望读者能够在本文指导下结合 JUC 包源码进行深入思考，以便加深理解。