6.2 抽象同步队列 AQS 概述

6.2.1 AQS——锁的底层支持

AbstractQueuedSynchronizer 抽象同步队列简称 AQS，它是实现同步器的基础组件，并发包中锁的底层就是使用 AQS 实现的。另外，大多数开发者可能永远不会直接使用 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，如图 6-1 所示。

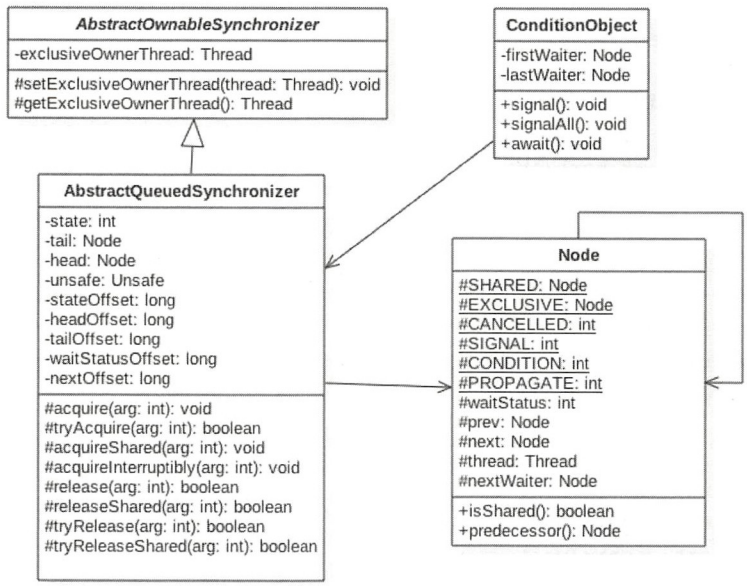

图 6-1

由该图可以看到，AQS 是一个 FIFO 的双向队列，其内部通过节点 head 和 tail 记录队首和队尾元素，队列元素的类型为 Node。其中 Node 中的 thread 变量用来存放进入 AQS 队列里面的线程；Node 节点内部的 SHARED 用来标记该线程是获取共享资源时被阻塞挂起后放入 AQS 队列的，EXCLUSIVE 用来标记线程是获取独占资源时被挂起后放入 AQS 队列的；waitStatus 记录当前线程等待状态，可以为 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（释放共享资源时需要通知其他节点）；prev 记录当前节点的前驱节点，next 记录当前节点的后继节点。

在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；对于读写锁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CountDownlatch来说，state用来表示计数器当前的值。

AQS有个内部类ConditionObject，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，每个条件变量对应一个条件队列（单向链表队列），其用来存放调用条件变量的await方法后被阻塞的线程，如类图所示，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的方式分为独占方式和共享方式。 在独占方式下获取和释放资源使用的方法为：void acquire(int arg) void acquireInterruptibly(int arg) boolean release(int arg)。

在共享方式下获取和释放资源的方法为：void acquireShared(int arg) void acquireSharedInterruptibly(int arg) boolean releaseShared(int arg)。

使用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，就会标记是这个线程获取到了，其他线程再尝试操作state获取资源时会发现当前该资源不是自己持有的，就会在获取失败后被阻塞。比如独占锁ReentrantLock的实现，当一个线程获取了ReentrantLock的锁后，在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发现它就是锁的持有者，则会把状态值从1变为2，也就是设置可重入次数，而当另外一个线程获取锁时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。

对应共享方式的资源与具体线程是不相关的，当多个线程去请求资源时通过CAS方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次去获取时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。比如Semaphore信号量，当一个线程通过acquire()方法获取信号量时，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。

在独占方式下，获取与释放资源的流程如下：

（1）当一个线程调用 acquire(int arg) 方法获取独占资源时，会首先使用 tryAcquire 方法尝试获取资源，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 阻塞队列的尾部，并调用 LockSupport.park(this) 方法挂起自己。

```
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) 方法时会尝试使用 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) 方法激活 AQS 队列里面被阻塞的一个线程 (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，否则还是会被放入 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;
}
```

需要注意的是，AQS 类并没有提供可用的 tryAcquire 和 tryRelease 方法，正如 AQS 是锁阻塞和同步器的基础框架一样，tryAcquire 和 tryRelease 需要由具体的子类来实现。子类在实现 tryAcquire 和 tryRelease 时要根据具体场景使用 CAS 算法尝试修改 state 状态值，成功则返回 true, 否则返回 false。子类还需要定义，在调用 acquire 和 release 方法时 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 时表示锁空闲，为 1 时表示锁已经被占用。在重写 tryAcquire 时，在内部需要使用 CAS 算法查看当前 state 是否为 0，如果为 0 则使用 CAS 设置为 1，并设置当前锁的持有者为当前线程，而后返回

true，如果 CAS 失败则返回 false。

比如继承自 AQS 实现的独占锁在实现 tryRelease 时，在内部需要使用 CAS 算法把当前 state 的值从 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。

在共享方式下，获取与释放资源的流程如下：

（1）当线程调用 acquireShared(int arg) 获取共享资源时，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 阻塞队列的尾部，并使用 LockSupport.park(this) 方法挂起自己。

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

（2）当一个线程调用 releaseShared(int arg) 时会尝试使用 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 队列里面被阻塞的一个线程 (thread)。被激活的线程则使用 tryReleaseShared 查看当前状态变量 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，否则还是会被放入 AQS 队列并被挂起。

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

同样需要注意的是，AQS 类并没有提供可用的 tryAcquireShared 和 tryReleaseShared 方法，正如 AQS 是锁阻塞和同步器的基础框架一样，tryAcquireShared 和 tryReleaseShared 需要由具体的子类来实现。子类在实现 tryAcquireShared 和 tryReleaseShared 时要根据具体场景使用 CAS 算法尝试修改 state 状态值，成功则返回 true，否则返回 false。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryAcquireShared 时，首先查看写锁是否被其他线程持有，如果是则直接返回 false，否则

使用 CAS 递增 state 的高 16 位 (在 ReentrantReadWriteLock 中，state 的高 16 位为获取读锁的次数)。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryReleaseShared 时，在内部需要使用 CAS 算法把当前 state 值的高 16 位减 1，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外，还需要重写 isHeldExclusively 方法，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，独占方式下的 void acquire(int arg) 和 void acquireInterruptibly(int arg)，与共享方式下的 void acquireShared(int arg) 和 void acquireSharedInterruptibly(int arg)，这两套函数中都有一个带有 Interruptibly 关键字的函数，那么带这个关键字和不带有什么区别呢？我们来讲讲。

其实不带 Interruptibly 关键字的方法的意思是不对中断进行响应，也就是线程在调用不带 Interruptibly 关键字的方法获取资源时或者获取资源失败被挂起时，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，它还是继续获取资源或者被挂起，也就是说不对中断进行响应，忽略中断。

而带 Interruptibly 关键字的方法要对中断进行响应，也就是线程在调用带 Interruptibly 关键字的方法获取资源时或者获取资源失败被挂起时，其他线程中断了该线程，那么该线程会抛出 InterruptedException 异常而返回。

最后，我们来看看如何维护 AQS 提供的队列，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) 方法将该节点插入到 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;
                }
            }
        }
    }
```

下面结合代码和节点图（见图 6-2）来讲解入队的过程。如上代码在第一次循环中，当要在 AQS 队列尾部插入元素时，AQS 队列状态如图 6-2 中（default) 所示。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 6-2 中（I）所示。

这时候 t 为 null，故执行代码（2），使用 CAS 算法设置一个哨兵节点为头节点，如果 CAS 设置成功，则让尾部节点也指向哨兵节点，这时候队列状态如图 6-2 中（II）所示。

到现在为止只插入了一个哨兵节点，还需要插入 node 节点，所以在第二次循环后执行到代码（1），这时候队列状态如图 6-2（III）所示；然后执行代码（3）设置 node 的前驱节点为尾部节点，这时候队列状态如图 6-2 中（IV）所示；然后通过 CAS 算法设置 node 节点为尾部节点，CAS 成功后队列状态如图 6-2 中（V）所示；CAS 成功后再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，此时队列状态如图 6-2 中（VI）所示。

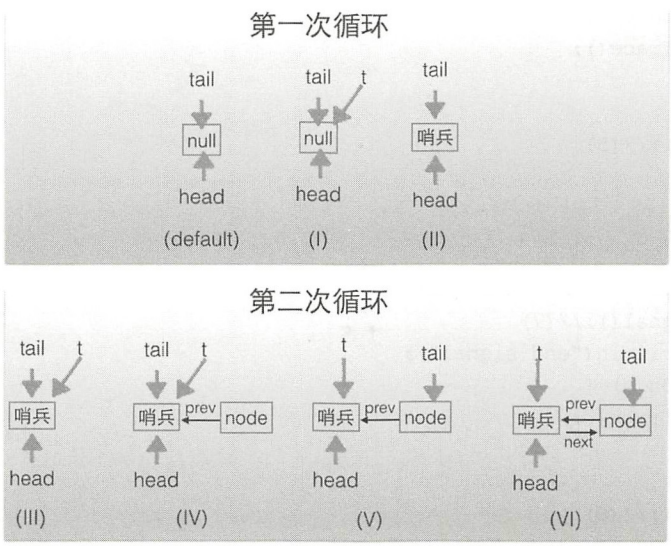

图 6-2

6.2.2 AQS——条件变量的支持

正如在基础篇中讲解的，notify 和 wait，是配合 synchronized 内置锁实现线程间同步的基础设施一样，条件变量的 signal 和 await 方法也是用来配合锁（使用 AQS 实现的锁）实现线程间同步的基础设施。

它们的不同在于，synchronized 同时只能与一个共享变量的 notify 或 wait 方法实现同步，而 AQS 的一个锁可以对应多个条件变量。

在基础篇中讲解了，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 方法前也必须先获取条件变量对应的锁。

那么，到底什么是条件变量呢？如何使用呢？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)
}
```

代码（1）创建了一个独占锁 ReentrantLock 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。当其他线程调用条件变量的 signal 方法时，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 方法则会抛出 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，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），调用 lock.unLock() 方法就相当于退出 synchronized 块。 调用条件变量的 await() 方法就相当于调用共享变量的 wait() 方法，调用条件变量的 signal 方法就相当于调用共享变量的 notify() 方法。调用条件变量的 signalAll（）方法就相当于调用共享变量的 notifyAll() 方法。

经过上面解释，相信大家已经知道条件变量是什么，它是用来做什么的了。

在上面代码中，lock.newCondition() 的作用其实是 new 了一个在 AQS 内部声明的 ConditionObject 对象，ConditionObject 是 AQS 的内部类，可以访问 AQS 内部的变量（例如状态变量 state）和方法。在每个条件变量内部都维护了一个条件队列，用来存放调用条件变量的 await() 方法时被阻塞的线程。注意这个条件队列和 AQS 队列不是一回事。

在如下代码中，当线程调用条件变量的 await() 方法时（必须先调用锁的 lock() 方法获取锁），在内部会构造一个类型为 Node.CONDITION 的 node 节点，然后将该节点插入条件队列末尾，之后当前线程会释放获取的锁（也就是会操作锁对应的 state 变量的值），并被阻塞挂起。这时候如果有其他线程调用 lock.lock() 尝试获取锁，就会有一个线程获取到锁，如果获取到锁的线程调用了条件变量的 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，在 await() 方法处阻塞。

```
public final void await() throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
             //创建新的node节点,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;
            }
            ...
        }
```

在如下代码中，当另外一个线程调用条件变量的 signal 方法时（必须先调用锁的 lock() 方法获取锁），在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并放入 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)
            //将条件队列头元素移动到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，该函数用来 new 一个 ConditionObject 对象。需要由 AQS 的子类来提供 newCondition 函数。

下面来看当一个线程调用条件变量的 await() 方法而被阻塞后，如何将其放入条件队列。

```
        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;
        }
```

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

注意：当多个线程同时调用 lock.lock() 方法获取锁时，只有一个线程获取到了锁，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里面。

当另外一个线程调用条件变量的 signal() 或者 signalAll() 方法时，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。

最后使用一个图（见图 6-3）总结如下：一个锁对应一个 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。

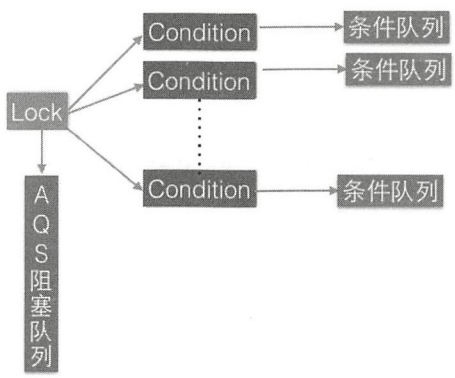

图 6-3

6.2.3 基于 AQS 实现自定义同步器

本节我们基于 AQS 实现一个不可重入的独占锁，正如前文所讲的，自定义 AQS 需要重写一系列函数，还需要定义原子变量 state 的含义。这里我们定义，state 为 0 表示目前

锁没有被线程持有，state 为 1 表示锁已经被某一个线程持有，由于是不可重入锁，所以不需要记录持有锁的线程获取锁的次数。另外，我们自定义的锁支持条件变量。

1. 代码实现

如下代码是基于 AQS 实现的不可重入的独占锁。

```
class NonReentrantLock implements Lock, java.io.Serializable {

    // 内部帮助类
    private static class Sync extends AbstractQueuedSynchronizer {
        // 是否锁已经被持有
        protected boolean isHeldExclusively() {
            return getState() == 1;
        }

        //如果state为0 则尝试获取锁
        public boolean tryAcquire(int acquires) {
            assert acquires == 1; //
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        // 尝试释放锁，设置state为0
        protected boolean tryRelease(int releases) {
            assert releases == 1; //
            if (getState() == 0)
                throw new IllegalMonitorStateException();
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }

        // 提供条件变量接口
        Condition newCondition() {
            return new ConditionObject();
        }

    }
```

```
    //创建一个Sync来做具体的工作
    private final Sync sync = new Sync();

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

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

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

    public Condition newCondition() {
        return sync.newCondition();
    }

    public boolean isLocked() {
        return sync.isHeldExclusively();
    }

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

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

在如上代码中，NonReentrantLock 定义了一个内部类 Sync 用来实现具体的锁的操作，Sync 则继承了 AQS。由于我们实现的是独占模式的锁，所以 Sync 重写了 tryAcquire、tryRelease 和 isHeldExclusively 3 个方法。另外，Sync 提供了 newCondition 这个方法用来支持条件变量。

2. 使用自定义锁实现生产—消费模型

下面我们使用上节自定义的锁实现一个简单的生产—消费模型，代码如下。

```
final static NonReentrantLock lock = new NonReentrantLock();
final static Condition notFull = lock.newCondition();
final static Condition notEmpty = lock.newCondition();

final static Queue<String> queue = new LinkedBlockingQueue<String>();
final static  int queueSize = 10;


public static void main(String[] args) {

    Thread producer = new Thread(new  Runnable() {
        public void run() {
            //获取独占锁
            lock.lock();
            try{

                //(1)如果队列满了，则等待
                while(queue.size() == queueSize){
                    notEmpty.await();
                }

                //(2)添加元素到队列
                queue.add("ele");

                //(3)唤醒消费线程
                notFull.signalAll();

            }catch(Exception e){
                e.printStackTrace();
            }finally {
                //释放锁
                lock.unlock();
            }
        }
    });

    Thread consumer = new Thread(new  Runnable() {
        public void run() {
            //获取独占锁
            lock.lock();
            try{
                //队列空，则等待
                while(0 == queue.size() ){
```

```
                    notFull.await();;
                }

                //消费一个元素
                String ele = queue.poll();
                //唤醒生产线程
                notEmpty.signalAll();

            }catch(Exception e){
                e.printStackTrace();
            }finally {
                //释放锁
                lock.unlock();
            }
        }
    });

    //启动线程
    producer.start();
    consumer.start();
}
```

如上代码首先创建了 NonReentrantLock 的一个对象 lock，然后调用 lock.newCondition 创建了两个条件变量，用来进行生产者和消费者线程之间的同步。

在 main 函数里面，首先创建了 producer 生产线程，在线程内部首先调用 lock.lock() 获取独占锁，然后判断当前队列是否已经满了，如果满了则调用 notEmpty.await() 阻塞挂起当前线程。需要注意的是，这里使用 while 而不是 if 是为了避免虚假唤醒。如果队列不满则直接向队列里面添加元素，然后调用 notFull.signalAll() 唤醒所有因为消费元素而被阻塞的消费线程，最后释放获取的锁。

然后在 main 函数里面创建了 consumer 生产线程，在线程内部首先调用 lock.lock() 获取独占锁，然后判断当前队列里面是不是有元素，如果队列为空则调用 notFull.await() 阻塞挂起当前线程。需要注意的是，这里使用 while 而不是 if 是为了避免虚假唤醒。如果队列不为空则直接从队列里面获取并移除元素，然后唤醒因为队列满而被阻塞的生产线程，最后释放获取的锁。

6.3 独占锁 ReentrantLock 的原理

6.3.1 类图结构

ReentrantLock 是可重入的独占锁，同时只能有一个线程可以获取该锁，其他获取该锁的线程会被阻塞而被放入该锁的 AQS 阻塞队列里面。首先看下 ReentrantLock 的类图以便对它的实现有个大致了解，如图 6-4 所示。

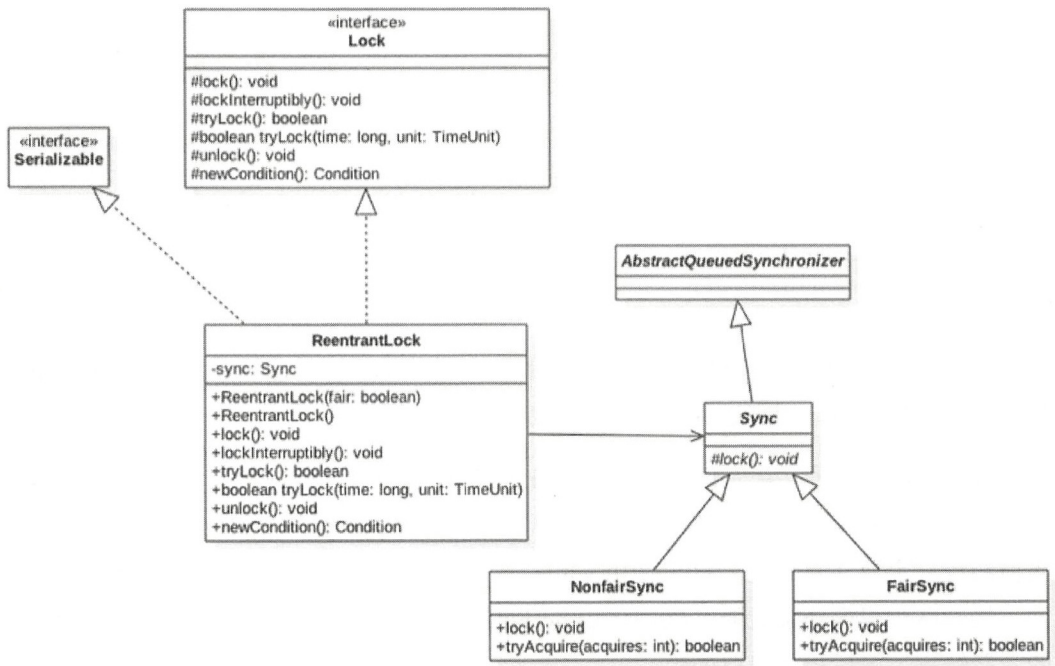

图 6-4

从类图可以看到，ReentrantLock 最终还是使用 AQS 来实现的，并且根据参数来决定其内部是一个公平还是非公平锁，默认是非公平锁。

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

    public ReentrantLock(boolean fair) {
```

```
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

其中 Sync 类直接继承自 AQS，它的子类 NonfairSync 和 FairSync 分别实现了获取锁的非公平与公平策略。

在这里，AQS 的 state 状态值表示线程获取该锁的可重入次数，在默认情况下，state 的值为 0 表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会尝试使用 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的情况下第二次获取该锁后，状态值被设置为 2，这就是可重入次数。在该线程释放该锁时，会尝试使用 CAS 让状态值减 1，如果减 1 后状态值为 0，则当前线程释放该锁。

6.3.2　获取锁

1. void lock() 方法

当一个线程调用该方法时，说明该线程希望获取该锁。如果锁当前没有被其他线程占用并且当前线程之前没有获取过该锁，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，然后直接返回。 如果当前线程之前已经获取过该锁，则这次只是简单地把 AQS 的状态值加 1 后返回。 如果该锁已经被其他线程持有，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();
    }
```

在如上代码中，ReentrantLock 的 lock() 委托给了 sync 类，根据创建 ReentrantLock 构造函数选择 sync 的实现是 NonfairSync 还是 FairSync，这个锁是一个非公平锁或者公平锁。这里先看 sync 的子类 NonfairSync 的情况，也就是非公平锁时。

```
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);
}
```

在代码（1）中，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设置该锁持有者是当前线程。

如果这时候有其他线程调用lock方法企图获取该锁，CAS会失败，然后会调用AQS的acquire方法。注意，传递参数为1，这里再贴下AQS的acquire的核心代码。

```
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。我们先看下非公平锁的代码。

```
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;
}
```

首先代码（4）会查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，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然后返回，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）查看当前线程是否是该锁的持有者，如果当前线程是该锁的持有者，则状态值加1，然后返回true，这里需要注意，nextc<0说明可重入次数溢出了。如果当前线程不是锁的持有者则返回false，然后其会被放入AQS阻塞队列。

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（）方法时执行到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，然后当前线程被放入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），发现当前状态值为0了（假设占有该锁的其他线程释放了该锁），所以通过CAS设置获取到了该锁。明明是线程A先请求获取该锁呀，这就是非公平的体现。这里线程B在获取锁前并没有查看当前AQS队列里面是否有比自己更早请求该锁的线程，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;
        }
    }
```

如以上代码所示，公平的 tryAcquire 策略与非公平的类似，不同之处在于，代码（8）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());
    }
```

在如上代码中，如果当前线程节点有前驱节点则返回 true，否则如果当前 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，直接返回 false；如果 h!=t 并且 s==null 则说明有一个元素将要作为 AQS 的第一个节点入队列（回顾前面的内容，enq 函数的第一个元素入队列是两步操作：首先创建一个哨兵头节点，然后将第一个元素插入哨兵节点后面），那么返回 true, 如果 h!=t 并且 s!=null 和 s.thread != Thread.currentThread() 则说明队列里面的第一个元素不是当前线程，那么返回 true。

2. void lockInterruptibly() 方法

该方法与 lock() 方法类似，它的不同在于，它对中断进行响应，就是当前线程在调用该方法时，如果其他线程调用了当前线程的 interrupt（）方法，则当前线程会抛出 InterruptedException 异常，然后返回。

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

public final void acquireInterruptibly(int arg)
       throws InterruptedException {
```

```
    //如果当前线程被中断，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可被中断的方法
        doAcquireInterruptibly(arg);
}
```

3. boolean tryLock() 方法

尝试获取锁，如果当前该锁没有被其他线程持有，则当前线程获取该锁并返回 true，否则返回 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() 方法代码类似，所以 tryLock() 使用的是非公平策略。

4. boolean tryLock(long timeout, TimeUnit unit) 方法

尝试获取锁，与 tryLock（）的不同之处在于，它设置了超时时间，如果超时时间到

没有获取到该锁则返回 false。

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

6.3.3 释放锁

1. void unlock() 方法

尝试释放锁，如果当前线程持有该锁，则调用该方法会让该线程对该线程持有的 AQS 状态值减 1，如果减去 1 后当前状态值为 0，则当前线程会释放该锁，否则仅仅减 1 而已。如果当前线程没有持有该锁而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;
}
```

如代码（11）所示，如果当前线程不是该锁持有者则直接抛出异常，否则查看状态值是否为 0，为 0 则说明当前线程要放弃对该锁的持有权，则执行代码（12）把当前锁持有者设置为 null。如果状态值不为 0，则仅仅让当前线程对该锁的可重入次数减 1。

6.3.4 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();

            }
        }
        //删除元素
        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一时间只有一个线程可以对 array 数组进行修改，但是也只能有一个线程对 array 元素进行访问。

同样最后使用图（见图 6-5）来加深理解。

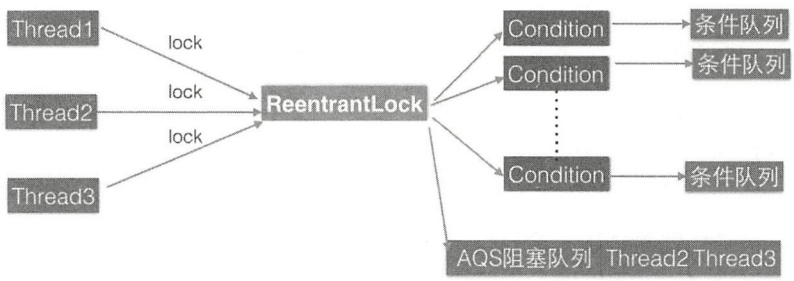

图 6-5

如图 6-5 所示，假如线程 Thread1、Thread2 和 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 节点并被放入 ReentrantLock 对应的 AQS 阻塞队列，而后被阻塞挂起。

如图 6-6 所示，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 节点插入条件变量 1 的条件队列。由于 Thread1 释放了锁，所以阻塞到 AQS 队列里面的 Thread2 和 Thread3 就有机会获取到该锁，假如使用的是公平策略，那么这时候 Thread2 会获取到该锁，从而从 AQS 队列里面移除 Thread2 对应的 Node 节点。

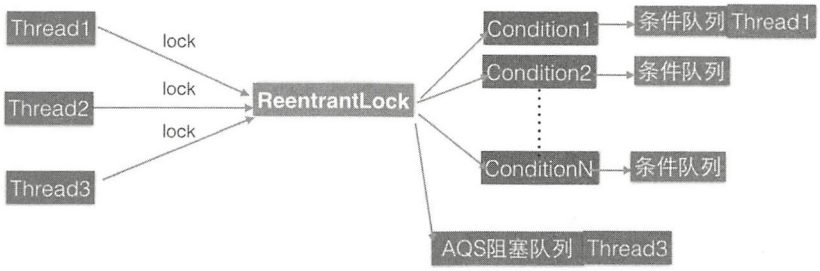

图 6-6

6.3.5 小结

本节介绍了 ReentrantLock 的实现原理，ReentrantLock 的底层是使用 AQS 实现的可重入独占锁。在这里 AQS 状态值为 0 表示当前锁空闲，为大于等于 1 的值则说明该锁已经被占用。该锁内部有公平与非公平实现，默认情况下是非公平的实现。另外，由于该锁是独占锁，所以某时只有一个线程可以获取该锁。

6.4 读写锁 ReentrantReadWriteLock 的原理

解决线程安全问题使用 ReentrantLock 就可以，但是 ReentrantLock 是独占锁，某时只有一个线程可以获取该锁，而实际中会有写少读多的场景，显然 ReentrantLock 满足不了这个需求，所以 ReentrantReadWriteLock 应运而生。ReentrantReadWriteLock 采用读写分离的策略，允许多个线程可以同时获取读锁。

6.4.1 类图结构

为了了解 ReentrantReadWriteLock 的内部构造，我们先看下它的类图结构，如图 6-7 所示。

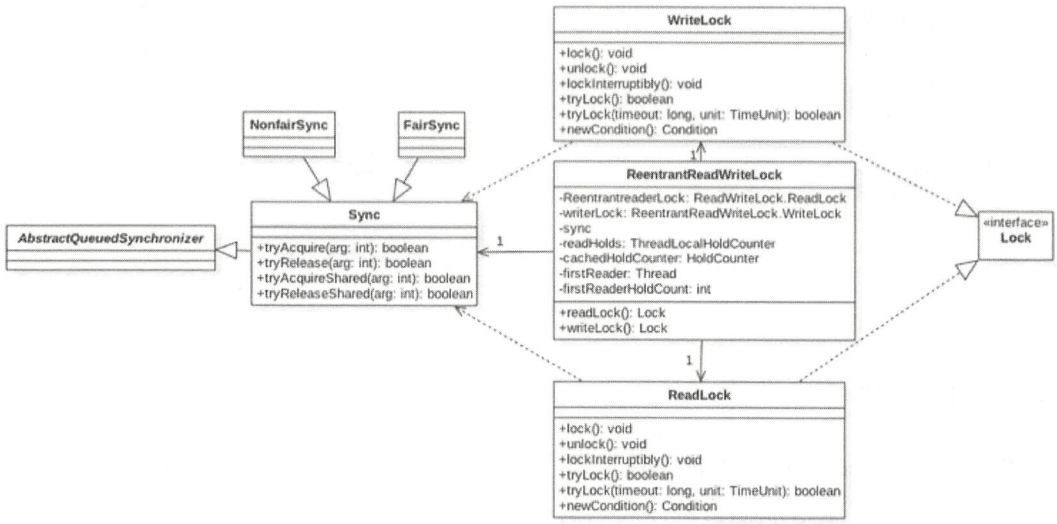

图 6-7

读写锁的内部维护了一个 ReadLock 和一个 WriteLock，它们依赖 Sync 实现具体功能。而 Sync 继承自 AQS，并且也提供了公平和非公平的实现。下面只介绍非公平的读写锁实现。我们知道 AQS 中只维护了一个 state 状态，而 ReentrantReadWriteLock 则需要维护读状态和写状态，一个 state 怎么表示写和读两种状态呢 ?ReentrantReadWriteLock 巧妙地使用 state 的高 16 位表示读状态，也就是获取到读锁的次数；使用低 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; }
```

其中 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次数。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他线程获取读锁的可重入次数。ThreadLocalHoldCounter 继承了 ThreadLocal，因而 initialValue 方法返回一个 HoldCounter 对象。

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

6.4.2 写锁的获取与释放

在 ReentrantReadWriteLock 中写锁使用 WriteLock 来实现。

1. void lock()

写锁是个独占锁，某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获取到读锁和写锁，则当前请求写锁的线程会被阻塞挂起。 另外，写锁是可重入锁，如果当前线程已经获取了该锁，再次获取只是简单地把可重入次数加 1 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();
```

```
    }
```

如以上代码所示，在 lock() 内部调用了 AQS 的 acquire 方法，其中 tryAcquire 是 ReentrantReadWriteLock 内部的 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，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)说明当前线程获取了写锁，判断可重入次数
                if (w + exclusiveCount(acquires) > MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");

                // (4)设置可重入次数(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;
        }
```

在代码（1）中，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。在代码（2）中，如果 w==0 说明状态值的低 16 位为 0，而 AQS 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。

而如果 w!=0 则说明当前已经有线程获取了该写锁，再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，所以判断该线程的可重入次数是不是超过了最大值，是则抛出异常，否则执行代码（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，所以执行代码（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
}
```

如果代码对于非公平锁来说总是返回 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并返回 true，否则返回 false。

公平锁的实现为

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

这里还是使用 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。

2. void lockInterruptibly()

类似于 lock() 方法，它的不同之处在于，它会对中断进行响应，也就是当其他线程调用了该线程的 interrupt() 方法中断了当前线程时，当前线程会抛出异常 InterruptedException 异常。

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

3. boolean tryLock()

尝试获取写锁，如果当前没有其他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。 如果当前已经有其他线程持有写锁或者读锁则该方法直接返回 false，且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是非公平策略。

4. boolean tryLock(long timeout, TimeUnit unit)

与 tryAcquire（）的不同之处在于，多了超时时间参数，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，该方法会对中断进行响应，也就是当其他线程调用了该线程的 interrupt() 方法中断了当前线程时，当前线程会抛出 InterruptedException 异常。

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

5. 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 1，如果减去 1 后当前状态值为 0 则当前线程会释放该锁，否则仅仅减 1 而已。如果当前线程没有持有该锁而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，因为获取写锁时读锁状态值肯定为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，否则只是简单地更新状态值
            if (free)
                setExclusiveOwnerThread(null);
            setState(nextc);
            return free;
        }
```

在如上代码中，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，否则执行代码（7），这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 则说明可重入次数为 0，所以当前线程会释放写锁，将当前锁的持有者设置为 null。如果 free 为 false 则简单地更新可重入次数。

6.4.3 读锁的获取与释放

ReentrantReadWriteLock 中的读锁是使用 ReadLock 来实现的。

1. void lock()

获取读锁，如果当前没有其他线程持有写锁，则当前线程可以获取读锁，AQS 的状态值 state 的高 16 位的值会增加 1，然后方法返回。否则如果其他一个线程持有写锁，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);
 }
```

在如上代码中，读锁的 lock 方法调用了 AQS 的 acquireShared 方法，在其内部调用了 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他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）查看是否有其他线程获取到了写锁，如果是则直接返回 –1，而后调用 AQS 的 doAcquireShared 方法把当前线程放入 AQS 阻塞队列。

如果当前要获取读锁的线程已经持有了写锁，则也可以获取读锁。但是需要注意，当一个线程先获取了写锁，然后获取了读锁处理事情完毕后，要记得把读锁和写锁都释放掉，不能只释放写锁。

否则执行代码（3），得到获取到的读锁的个数，到这里说明目前没有线程获取到写锁，但是可能有线程持有读锁，然后执行代码（4）。其中非公平锁的 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;
 }
```

如上代码的作用是，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，则当前线程判断当前获取读锁的线程是否达到了最大值。最后执行 CAS 操作将 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和该线程获取读锁的可重入

数，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，所以执行代码（8）。fullTryAcquireShared 的代码与 tryAcquireShared 类似，它们的不同之处在于，前者通过循环自旋获取。

```
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;
        }
    }
}
```

2. void lockInterruptibly()

类似于 lock() 方法，不同之处在于，该方法会对中断进行响应，也就是当其他线程调用了该线程的 interrupt() 方法中断了当前线程时，当前线程会抛出 InterruptedException 异常。

3. boolean tryLock()

尝试获取读锁，如果当前没有其他线程持有写锁，则当前线程获取读锁会成功，然后返回 true 。如果当前已经有其他线程持有写锁则该方法直接返回 false，但当前线程并不会被阻塞。 如果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。其代码类似 tryLock 的代码，这里不再讲述。

4. boolean tryLock(long timeout, TimeUnit unit)

与 tryLock（）的不同之处在于，多了超时时间参数，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，如果此时还没有获取到读锁则返回 false。另外，该方法对中断响应 , 也就是当其他线程调用了该线程的 interrupt() 方法中断了当前线程时，当前线程会抛出 InterruptedException 异常。

5. void unlock()

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

如上代码具体释放锁的操作是委托给 Sync 类来做的，sync.releaseShared 方法的代码如下：

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

其中 tryReleaseShared 的代码如下。

```
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    ....

    //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;
    }
}
```

如以上代码所示，在无限循环里面，首先获取当前 AQS 状态值并将其保存到变量 c，然后变量 c 被减去一个读计数单位后使用 CAS 操作更新 AQS 状态值，如果更新成功则查看当前 AQS 状态值是否为 0，为 0 则说明当前已经没有读线程占用读锁，则 tryReleaseShared 返回 true。然后会调用 doReleaseShared 方法释放一个由于获取写锁而被阻塞的线程，如果当前 AQS 状态值不为 0，则说明当前还有其他线程持有了读锁，所以 tryReleaseShared 返回 false。如果 tryReleaseShared 中的 CAS 更新 AQS 状态值失败，则自旋重试直到成功。

6.4.4 案例介绍

上节介绍了如何使用 ReentrantLock 实现线程安全的 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 来改造它，代码如下。

```
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();

    }
}
//删除元素
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();

    }
}
```

```
}
```

以上代码调用 get 方法时使用的是读锁，这样运行多个读线程来同时访问 list 的元素，这在读多写少的情况下性能会更好。

最后使用一张图（见图 6-8）来加深对 ReentrantReadWriteLock 的理解。

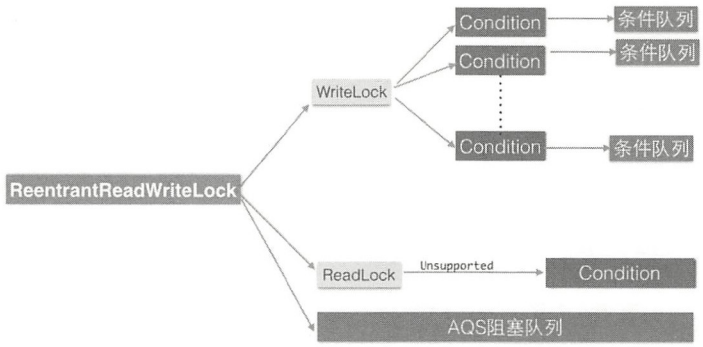

图 6-8

6.4.5 小结

本节介绍了读写锁 ReentrantReadWriteLock 的原理，它的底层是使用 AQS 实现的。ReentrantReadWriteLock 巧妙地使用 AQS 的状态值的高 16 位表示获取到读锁的个数，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，这在读多写少的场景下比较适用。

6.5 JDK 8 中新增的 StampedLock 锁探究

6.5.1 概述

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，会返回一个 long 型的变量，我们称之为戳记（stamp），这个戳记代表了锁的状态。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需要传入获取锁时返回的 stamp 值。

StampedLock 提供的三种读写模式的锁分别如下。

- **写锁 writeLock**：是一个排它锁或者独占锁，某时只有一个线程可以获取该锁，当一个线程获取该锁后，其他请求读锁和写锁的线程必须等待，这类似于 ReentrantReadWriteLock 的写锁（不同的是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时才可以获取到该锁。请求该锁成功后会返回一个 stamp 变量用来表示该锁的版本，当释放该锁时需要调用 unlockWrite 方法并传递获取锁时的 stamp 参数。并且它提供了非阻塞的 tryWriteLock 方法。
- **悲观读锁 readLock**：是一个共享锁，在没有线程获取独占写锁的情况下，多个线程可以同时获取该锁。如果已经有线程持有写锁，则其他线程请求获取该读锁会被阻塞，这类似于 ReentrantReadWriteLock 的读锁（不同的是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其会悲观地认为其他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 变量用来表示该锁的版本，当释放该锁时需要调用 unlockRead 方法并传递 stamp 参数。并且它提供了非阻塞的 tryReadLock 方法。
- **乐观读锁 tryOptimisticRead**：它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，仅仅通过位运算测试。如果当前没有线程持有写锁，则简单地返回一个非 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 方法验证该 stamp 是否已经不可用，也就是看当调用 tryOptimisticRead 返回 stamp 后到当前时间期间是否有其他线程持有了写锁，如果是则 validate 会返回 0，否则就可以使用该 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，所以不需要显式地释放该锁。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复制一份要操作的变量到方法栈，并且在操作数据时可能其他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）：

- 当前锁已经是写锁模式了。
- 当前锁处于读锁模式，并且没有其他线程是读锁模式
- 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，所以在获取锁后释放锁前不应该再调用会获取锁的操作，以避免造成调用线程被阻塞。当多个线程同时尝试获取读锁和写锁时，谁先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，而是其在内部自己维护了一个双向阻塞队列。

6.5.2 案例介绍

下面通过 JDK 8 里面提供的一个管理二维点的例子来理解以上介绍的概念。

```
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() {

        // （1）尝试获取乐观读锁
        long stamp = sl.tryOptimisticRead();
        // （2）将全部变量复制到方法体栈内
        double currentX = x, currentY = y;
        // （3）检查在（1）处获取了读锁戳记后，锁有没被其他写线程排它性抢占
        if (!sl.validate(stamp)) {
```

```
            // （4）如果被抢占则获取一个共享读锁（悲观获取）
            stamp = sl.readLock();
            try {
                // （5）将全部变量复制到方法体栈内
                currentX = x;
                currentY = y;
            } finally {
                //（6） 释放共享读锁
                sl.unlockRead(stamp);
            }
        }
        // （7）返回计算结果
        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) {
        // （1）这里可以使用乐观读锁替换
        long stamp = sl.readLock();
        try {
            // （2）如果当前点在原点则移动
            while (x == 0.0 && y == 0.0) {
                // （3）尝试将获取的读锁升级为写锁
                long ws = sl.tryConvertToWriteLock(stamp);
                // （4）升级成功，则更新戳记，并设置坐标值，然后退出循环
                if (ws != 0L) {
                    stamp = ws;
                    x = newX;
                    y = newY;
                    break;
                } else {
                    // （5）读锁升级写锁失败则释放读锁，显式获取独占写锁，然后循环重试
                    sl.unlockRead(stamp);
                    stamp = sl.writeLock();
                }
            }
        } finally {
            // （6）释放锁
            sl.unlock(stamp);
        }
    }
}
```

在如上代码中，Point 类里面有两个成员变量（x,y) 用来表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，该方法的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，而后释放锁。该锁是排它锁，这保证了其他线程调用 move 函数时会被阻塞，也保证了其他线程不能获取读锁，来读取坐标的值，直到当前线程显式释放了写锁，保证了对变量 x,y 操作的原子性和数据一致性。

然后看 distanceFromOrigin 方法，该方法的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，如果当前没有其他线程获取到了写锁，那么代码（1）会返回一个非 0 的 stamp 用来表示版本信息，代码（2）复制坐标变量到本地方法栈里面。

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

如果校验成功则执行代码（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）中校验成功后，在代码（7）计算期间，其他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）进行计算时采用的还是修改前的值的副本，也就是操作的值是之前值的一个副本，一个快照，并不是最新的值。

另外还有个问题，代码（2）和代码（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 validate 通过了，要复制 x,y 值到本地方法栈，而在复制的过程中很有可能其他线程已经修改了 x,y 中的一个值，这就造成了数据的不一致。那么你可能会问，即使不交换代码（2）和代码（3），在复制 x,y 值到本地方法栈时，也会存在其他线程修改了 x,y 中的一个值的情况，这不也会存在问题吗？这个确实会存在，但是，别忘了复制后还有 validate 这一关呢，如果这时候有线程修改了 x,y 中的某一值，那么肯定是有线程在调用 validate 前，调用 sl.tryOptimisticRead 后获取了写锁，这样进行 validate 时就会失败。

现在你应该明白了，这也是乐观读设计的精妙之处，而且也是在使用时容易出问题的地方。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）会使当前线程阻塞直到其他线程释放了写锁。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）重新复制新的坐标值到本地方法栈，再然后就是代码（6）释放了锁。复制时由于加了读锁，所以在复制期间如果有其他线程获取写锁会被阻塞，这保证了数据的一致性。另外，这里的 x,y 没有被声明为 volatie 的，会不会存在内存不可见性问题呢？答案是不会，因为加锁的语义保证了内存的可见性。

最后代码（7）使用方法栈里面的数据计算并返回，同理，这里在计算时使用的数据也可能不是最新的，其他写线程可能已经修改过原来的 x,y 值了。

最后一个方法 moveIfAtOrigin 的作用是，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，保证其他线程不能获取写锁来修改 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代码（3）时只有一个可以升级成功，升级成功则返回非 0 的 stamp，否则返回 0。这里假设当前线程升级成功，然后执行代码（4）更新 stamp 值和坐标值，之后退出循环。如果升级失败则执行代码（5）首先释放读锁，然后申请写锁，获取到写锁后再循环重新设置坐标值。最后代码（6）释放锁。

使用乐观读锁还是很容易犯错误的，必须要小心，且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();//复制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();//复制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
```

最后通过一张图（见图 6-9）来一览 StampedLock 的组成。

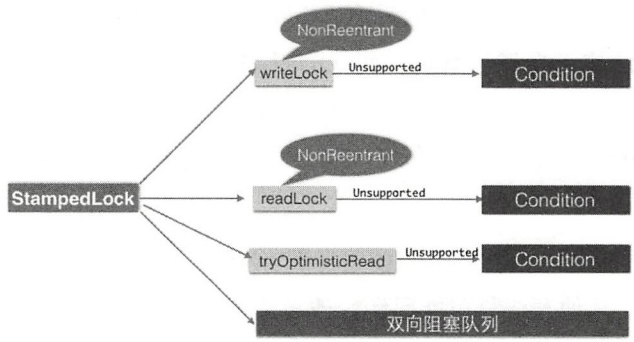

图 6-9

6.5.3 小结

StampedLock 提供的读写锁与 ReentrantReadWriteLock 类似，只是前者提供的是不可重入锁。但是前者通过提供乐观读锁在多线程多读的情况下提供了更好的性能，这是因为获取乐观读锁时不需要进行 CAS 操作设置锁的状态，而只是简单地测试状态。

第7章

Java并发包中并发队列原理剖析

JDK 中提供了一系列场景的并发安全队列。总的来说，按照实现方式的不同可分为阻塞队列和非阻塞队列，前者使用锁实现，而后者则使用 CAS 非阻塞算法实现。

7.1 ConcurrentLinkedQueue 原理探究

ConcurrentLinkedQueue 是线程安全的无界非阻塞队列，其底层数据结构使用单向链表实现，对于入队和出队操作使用 CAS 来实现线程安全。下面我们来看具体实现。

7.1.1 类图结构

为了能从全局直观地了解 ConcurrentLinkedQueue 的内部构造，先简单介绍 ConcurrentLinkedQueue 的类图结构，如图 7-1 所示。

ConcurrentLinkedQueue 内部的队列使用单向链表方式实现，其中有两个 volatile 类型的 Node 节点分别用来存放队列的首、尾节点。从下面的无参构造函数可知，默认头、尾节点都是指向 item 为 null 的哨兵节点。新元素会被插入队列末尾，出队时从队列头部获取一个元素。

```
public ConcurrentLinkedQueue() {
   head = tail = new Node<E>(null);
}
```

在 Node 节点内部则维护一个使用 volatile 修饰的变量 item，用来存放节点的值；next 用来存放链表的下一个节点，从而链接为一个单向无界链表。其内部则使用 UNSafe 工具类提供的 CAS 算法来保证出入队时操作链表的原子性。

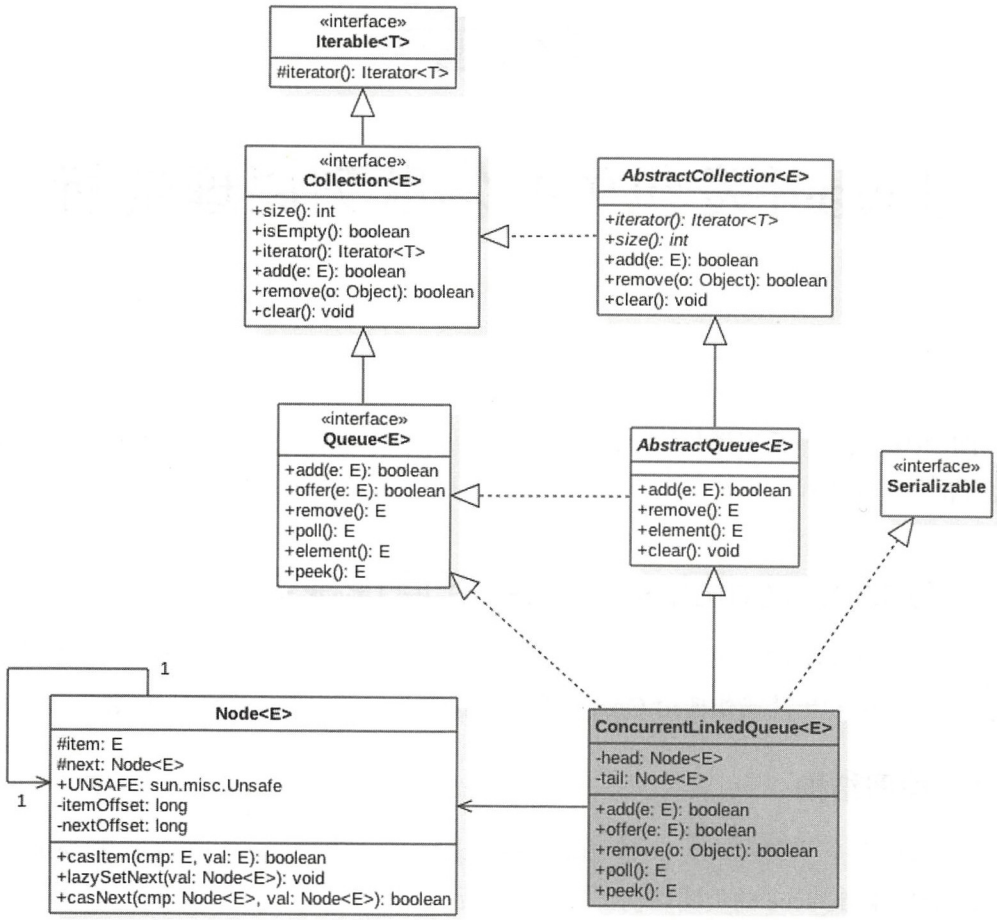

图 7-1

7.1.2 ConcurrentLinkedQueue 原理介绍

本节介绍 ConcurrentLinkedQueue 的几个主要方法的实现原理。

1. offer 操作

offer 操作是在队列末尾添加一个元素，如果传递的参数是 null 则抛出 NPE 异常，否则由于 ConcurrentLinkedQueue 是无界队列，该方法一直会返回 true。另外，由于使用 CAS 无阻塞算法，因此该方法不会阻塞挂起调用线程。下面具体看下实现原理。