深入理解读写锁ReentrantReadWriteLock

1.1读写锁简介

在并发场景中用于**解决线程安全问题**，我们几乎会高频的使用读写锁。通常使用java提供的关键字synchronized或者 concurrents包中实现了Lock接口的ReentrantLock。他们都是独占式获取锁，也就是在同一时刻只有一个线程能够获取到锁。而在一些业务的场景中，大部分都只是读数据，写数据写的少，如果仅仅式读数据的化并不影响数据正确性（出现脏读），而如果在这种业务场景下，依然使用独占锁的话，很显然这将是出现性能瓶颈的地方。针对这种读多写少的情况，java还提供了另一个实现Lock接口子类ReentrantReadWriteLock（读写锁）。**读写锁允许同一时刻被多个线程访问到，但是在写线程访问的时候，所有读线程和其他写线程都会被阻塞。**在分析WriteLock和ReadLock的互斥性时可以按照WriteLock与WriteLock之间，WriteLock与ReadLock之间以及ReadLock与ReadLock之间进行分析。这里做一个归纳总结：

1. **公平性选择：**支持非公性（默认）和公平的锁获取方式，**吞吐量上**还是非公平优于公平；
2. **重入性：**支持重入，读锁获取后能再次获取，写锁获取后能再次获取写锁，同时也能够获取读锁；

- [ ] **锁建降级：**遵循获取写锁，获取读锁再释放写锁的次序，写锁能够能够降级成为为读锁。什么意思

Lock ReentrantLock ReentrantWriteLcok （WriteLock ReadLock）共享锁：没有一个单独的子类

1.2写锁详解 (独占式获取)

1.2.1写锁的获取

同步组件的实现聚合了同步器（AQS）,并通过重写重写同步器（AQS）种的方法实现同步组件的同步语义因此，写锁的实现依然也是采用此方法。在同一时刻写锁是不能被多个线程所获取，很显然写锁是**独占式锁，**而实现**写锁的同步语义也是通过重写AQS中的tryAcquire方法实现的源码如下：**

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


Sync(默认非公共锁NonfairSync（）) in ReentrantReadWriteLock

//AQS的acquire
 public final void acquire(int arg) {
  if (!tryAcquire(arg) &&
  acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
```

```
 selfInterrupt();
}


// WriteLock - Sync- tryAcquire
//该方法在ReentrantReadWriteLock中（公平锁，非公平锁是相同的）

protected final boolean tryAcquire(int acquires) {
 /*
 * Walkthrough:
 * 1. If read count nonzero or write count nonzero
 * and owner is a different thread, fail.
 * 2. If count would saturate, fail. (This can only
 * happen if count is already nonzero.)
 * 3. Otherwise, this thread is eligible for lock if
 * it is either a reentrant acquire or
 * queue policy allows it. If so, update state
 * and set owner.
 */
 Thread current = Thread.currentThread();//当前线程
 int c = getState(); //当前读写锁当前的同步状态
 int w = exclusiveCount(c);//获取写锁获取的次数
 if (c != 0) {
 //当前有线程获得读写锁（写锁或者读锁）
 // (Note: if c != 0 and w == 0 then shared count != 0)
 if (w == 0 || current != getExclusiveOwnerThread()) //获取写锁的次数为0，且获取锁的线程不是当前线程
 //言外之意 当前锁被读线程拿着。写锁无法获取同步状态
 return false; //直接返回，写锁获取同步状态失败
 if (w + exclusiveCount(acquires) > MAX_COUNT) //写锁的重入次数达到最大
 throw new Error("Maximum lock count exceeded");//抛出锁重入最大次数错误
 // Reentrant acquire
 //写锁重入次数并没有达到最大，写锁的可重入，写锁重入（引用计数）+1
 setState(c + acquires);
 return true;//写锁重入成功 返回；
 }
 //当前读写锁没有线程获得,当前线程获取锁
 if (writerShouldBlock() ||
 !compareAndSetState(c, c + acquires))

 //(writerShouldBlock() :默认情况下返回false(非公平锁)，当前锁状态为);
```

```
53   //公平锁：当同步队列不为空 且当前线程在是在同步队列的头节点的下一个节点，当前写线程将被阻塞。
54   //此时读锁状态为0，写锁可以正常获取到同步状态
55   return false;
56   setExclusiveOwnerThread(current);//当前线程设置为只有写锁线程
57   return true;//写线程获得写锁。
58  }
59
60
```

1.2.2写锁阻塞的实现：非共公平锁/公平锁

```
1  ReentrantReadWeiteLock（）-sync中的抽象方法，放在其中的FairSync（公平锁）和NonfairSync（非公平锁）中实现
2  abstract boolean readerShouldBlock();
3
4  /**
5   * Returns true if the current thread, when trying to acquire
6   * the write lock, and otherwise eligible to do so, should block
7   * because of policy for overtaking other waiting threads.
8   */
9  abstract boolean writerShouldBlock();
```

```
1
2  //默认的写锁的阻塞-非公平式独占锁
3  final boolean writerShouldBlock() {
4   return false; // writers can always barge
5  }
6
7
8  //公平式独占锁-写锁的阻塞和读锁阻塞相同
9  final boolean writerShouldBlock() {
10   return hasQueuedPredecessors();
11  }
12
13  public final boolean hasQueuedPredecessors() {
14   // The correctness of this depends on head being initialized
15   // before tail and on head.next being accurate if the current
16   // thread is first in queue.
17   Node t = tail; // Read fields in reverse initialization order
18   Node h = head;
19   Node s;
20   return h != t &&
```

```
21    ((s = h.next) == null || s.thread != Thread.currentThread());
22    //返回true:同步队列不为空且，同步队列中头节点的下一个节点不为空，当前线程就是在头节点的下一个非空节点
23    //反之：返回false;
24  }
```

1.2.3需要注意的点：写状态（写锁重入计数）的标识

其中**EXCLUSIVE_MASK**为：

```
1  /** Returns the number of exclusive holds represented in count */
2  static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
```

EXCLUSIVE_MASK为1左移16为减一，即0x0000FFFF。

而exclusiveCount方法是将同步状态（state为int类型）与 0x0000FFFF相与，即取同步状态的低16位。

根据exclusiveCount方法的注释为独占式获取的次数即写锁被获取的次数，现在就可以得出来一个结论

:同步状态的低16位用来表示写锁的获取次数。

1.2.4同时还有 一个方法值得我们注意：读状态（读锁重入计数）的标识

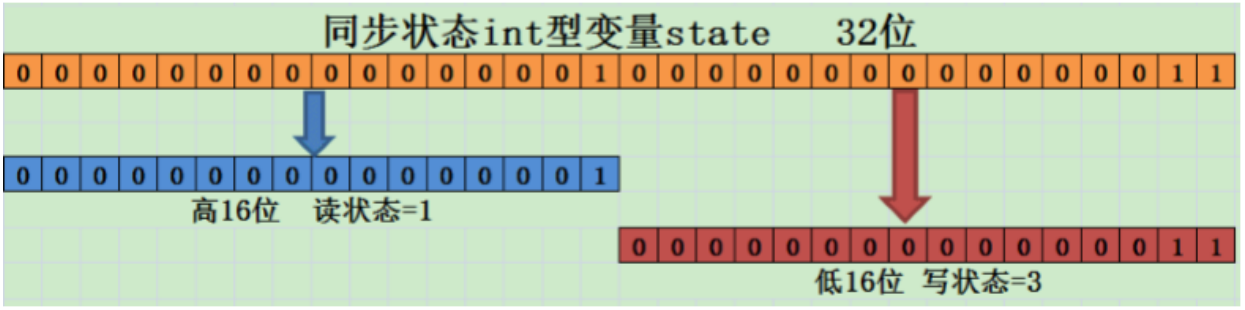
```
1  /** Returns the number of shared holds represented in count */
2  static int sharedCount(int c) { return c >>> SHARED_SHIFT; }
```

该方法式**获取读锁被获取的次数，是将同步状态（int c）右移16次，，即取同步状态的高16位**，现在我们可以得出另 外一个结论**同步状态的高16位用来表示读锁被获取的次数。**

```
1  //源码如下：
2
3  static final int SHARED_SHIFT = 16;
4  static final int SHARED_UNIT = (1 << SHARED_SHIFT);
5  static final int MAX_COUNT = (1 << SHARED_SHIFT) - 1;
6  static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
7
8  /** Returns the number of shared holds represented in count */
9  static int sharedCount(int c) { return c >>> SHARED_SHIFT; }//去除低16位影响
10  /** Returns the number of exclusive holds represented in count */
11  static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }//去除高16位影响
12
```

1.2.5读写锁同步状态的分别实现

如下是读写锁实现分别记录读锁和写锁的状态的图



1.3写锁的释放

写锁的释放通过重写AQS的tryReleasd方法

```
1  //WriteLock
2  public void unlock() {
3   sync.release(1);
4  }
5
6  Sync(默认非公共锁NonfairSync（）) in ReentrantReadWriteLock
7
8  public final boolean release(int arg) {
9   if (tryRelease(arg)) {
10   Node h = head;
11   if (h != null && h.waitStatus != 0)
12   unparkSuccessor(h);
13   return true;
14   }
15   return false;
16  }
17  //ReentrantReadWriteLock -Sync中的 -tryRlease（）写锁 独占式锁的释放
18
19  protected final boolean tryRelease(int releases) {
20   if (!isHeldExclusively())
21   //假如该线程并没有拿到写锁则抛出锁状态异常。
22   throw new IllegalMonitorStateException();
23   int nextc = getState() - releases; //写锁的同步状态-1
24   boolean free = exclusiveCount(nextc) == 0;//判断写锁状态是否为0，为0则释放
锁。
25   if (free)
26   setExclusiveOwnerThread(null); //释放持有锁的线程
27   setState(nextc); //写锁状态为0，则更新同步转台。
28   return free;//返回该线程是否释放了写锁。
```

```
29 }
30
31 //ReentrantReadWriteLock -Sync中的-isHeldExclusively()
32 //和ReentrantLock锁的isHeldExclusively()方法一样。
33
34 protected final boolean isHeldExclusively() {
35  // While we must in general read state before owner,
36  // we don't need to do so to check if current thread is owner
37  return getExclusiveOwnerThread() == Thread.currentThread();
38  //判断当前线程是否是 独占锁持有线程。
39 }
```

源码的实现逻辑请看注释，不难理解与ReentrantLock基本一致，这里需要注意的是，减少写状态int nextc = getState() - releases;只需要用**当前同步状态直接减去写状态的原因正是我们刚才所说的写状态是由同步状态的低16位表示的。 （写状态）**

1.4读锁的获取

读锁不是独占式锁，即同一时刻可以被多个线程获取也就是是一种共享锁。
读锁释放主要是tryReleaseShared 源码如下：

实现共享式同步组件的同步语义需要通过重写AQS的tryAcquireShared方法和tryReleaseShared 方法。读锁的获取实现方法为：

```
1 //ReadLcok
2 public void lock() {
3  sync.acquireShared(1);
4 }
5 //AQS-的同步组件 acquireShared()
6 public final void acquireShared(int arg) {
7  if (tryAcquireShared(arg) < 0)
8  doAcquireShared(arg); //读锁夺取失败执行
9 }
10
```

1.4.1tryAcquireShared(arg) 尝试获取共享资源。

说明：共享模式下获取资源/锁，忽略中断的影响。内部主要调用了两个个方法，其中tryAcquireShared需要自定义同步器实现。后面会对各个方法进行详细分析。acquireShared方法流程如下：

1. tryAcquireShared(arg) 尝试获取共享资源。**成功获取并且还有可用资源返回正数；成功获取但是没有可用资源时返回0；获取资源失败返回一个负数。**

- **说明：**尝试获取共享资源，需同步器自定义实现。有三个类型的返回值：
- 1：成功获取资源，并且还有剩余可用资源，可以唤醒下一个等待线程；
- -1：获取资源失败，准备进入等待队列；
- 0：获取资源成功，但没有剩余可用资源

2. 获取资源失败后调用doAcquireShared方法进入等待队列，获取资源后返回。

```
1
2  //sync -ReentrantReadWriteLock
3  //实现同步组件的同步语义，覆写AQS的tryAcquireShared方法
4
5  protected final int tryAcquireShared(int unused) {
6   /*
7   * Walkthrough:
8   * 1. If write lock held by another thread, fail.
9   * 2. Otherwise, this thread is eligible for
10   * lock wrt state, so ask if it should block
11   * because of queue policy. If not, try
12   * to grant by CASing state and updating count.
13   * Note that step does not check for reentrant
14   * acquires, which is postponed to full version
15   * to avoid having to check hold count in
16   * the more typical non-reentrant case.
17   * 3. If step 2 fails either because thread
18   * apparently not eligible or CAS fails or count
19   * saturated, chain to version with full retry loop.
20   */
21   Thread current = Thread.currentThread();//当前线程
22   int c = getState();//当前读写锁的状态
23   if (exclusiveCount(c) != 0 && //如写锁的获取的次数 即写锁重入次数
```

```
24   getExclusiveOwnerThread() != current) //当前线程独占锁（写锁）中线程不是当
前线程
25   return -1; //返回-即获取失败
26   int r = sharedCount(c); //当前读锁获取数量
27   if (!readerShouldBlock() && //读锁是否需要被阻塞（默认调用非公平锁覆写的方
法） 下面详解
28   r < MAX_COUNT && //读锁的获取次数小于读锁获取的最大计数
29   compareAndSetState(c, c + SHARED_UNIT)) { //则读锁的获得锁的数量+1
30   ----------------------------------------------------------------
31   // 新增关于读锁的一些功能，比如getReadHoldCount()方法返回
32   // 当前获取读锁的次数
33   if (r == 0) { //当前读锁没有任何线程获得，初始化firstReader和firstReaderHo
ldCount
34   firstReader = current; //当前度线程成为同步队列的头节点，获得锁
35   firstReaderHoldCount = 1; //头节点读锁的所状态 （有读线程获得锁）
36   } else if (firstReader == current) { //如果当前节点就是当前同步队列中的头节
点
37   firstReaderHoldCount++; //当前头读锁获得锁的数目+1；
38   } else {
39   HoldCounter rh = cachedHoldCounter; //更新cachedHoldCounter
40   if (rh == null || rh.tid != getThreadId(current))
41   cachedHoldCounter = rh = readHolds.get();
42   else if (rh.count == 0)
43   readHolds.set(rh);
44   rh.count++;//更新获取读锁的次数
45   }
46   return 1;
47   }
48   // CAS失败或者已经获取读锁的线程再次重入
49   return fullTryAcquireShared(current);
50  }
51  ----------------------------------------------------------------
52
```

说明：tryAcquireShared()的作用是尝试获取“读锁/共享锁”。函数流程如下：

1.如果“写锁”已经被持有，这时候可以继续获取读锁，但如果持有写锁的线程不是当前线程，直接返回-1（表示获取失败）；

2.如果在尝试获取锁时不需要阻塞等待（由公平性决定），并且读锁的共享计数小于最大数量MAX_COUNT，则直接通过CAS函数更新读取锁的共享计数，最后将当前线程获取读锁的次数+1。

3.如果第二步执行失败，则调用fullTryAcquireShared尝试获取读锁，源码如下：

1.4.2处理获取资源失败的线程

```
如果CAS失败或者已经获取读锁的线程再次获取读锁时，是靠fullTryAcquireShared方法实现的

final int fullTryAcquireShared(Thread current) {
 /*
 * This code is in part redundant with that in
 * tryAcquireShared but is simpler overall by not
 * complicating tryAcquireShared with interactions between
 * retries and lazily reading hold counts.
 */
 HoldCounter rh = null;
 for (;;) { //自旋
 int c = getState();
 //持有写线程的线程可以获取读锁，如果获取锁的线程不上当前线程，返回-1;
 if (exclusiveCount(c) != 0) { //写锁或取次数不等于0
 if (getExclusiveOwnerThread() != current)
 return -1;
 // else we hold the exclusive lock; blocking here
 // would cause deadlock.
 } else if (readerShouldBlock()) { //需要阻塞
 // Make sure we're not acquiring read lock reentrantly
 //当前线程如果是首个获取读锁的过程，初始化
 if (firstReader == current) {
 // assert firstReaderHoldCount > 0;
 } else {
 //更新读锁计时器
 if (rh == null) {
 rh = cachedHoldCounter;
 if (rh == null || rh.tid != getThreadId(current)) {
 rh = readHolds.get();
 if (rh.count == 0)
 readHolds.remove();//当前线程持有读锁数为0，移除计数器。
 }
 }
 if (rh.count == 0)
 return -1;
 }
```

```
37   }
38   if (sharedCount(c) == MAX_COUNT)//超过最大读锁数量
39   throw new Error("Maximum lock count exceeded");//抛出异常
40   if (compareAndSetState(c, c + SHARED_UNIT)) {//CAS更新读锁数量
41   if (sharedCount(c) == 0) {//首次获得读锁
42   firstReader = current;
43   firstReaderHoldCount = 1;
44   } else if (firstReader == current) {//当前线程节点是同步队列的头节点，持有
锁
45   firstReaderHoldCount++; //读锁计数+1
46   } else {
47   //更新所计数器
48   if (rh == null)
49   rh = cachedHoldCounter;
50   if (rh == null || rh.tid != getThreadId(current))
51   rh = readHolds.get();//更新为当前线程的计数器
52   else if (rh.count == 0)
53   readHolds.set(rh);
54   rh.count++;
55   cachedHoldCounter = rh; // cache for release
56   }
57   return 1;
58   }
59   }
60 }
```

代码的逻辑请看注释，需要注意的是 当写锁被其他线程获取后，读锁获取失败，否则获取成功利用CAS更新同步状 态。另外，当前同步状态需要加上SHARED_UNIT（(1 <<SHARED_SHIFT)即0x00010000）的原因这是我们在上面 所说的同步状态的高16位用来表示读锁被获取的次数。如果CAS失败或者已经获取读锁的线程再次获取读锁时，是靠fullTryAcquireShared方法实现的

1.4.3获取共享锁

```
1  //获取共享锁
2
3  private void doAcquireShared(int arg) {
4   // 添加一个共享模式Node到同步队列尾
5   final Node node = addWaiter(Node.SHARED);
6   //异常标记位
7   boolean failed = true;
8   try {
9   boolean interrupted = false;
10   for (;;) { //自旋 和acquireQueued很像
```

```
11  final Node p = node.predecessor();//获取前结点
12  if (p == head) {
13  int r = tryAcquireShared(arg);// 尝试获取 前结点位头节点，尝试获取资源
14  if (r >= 0) {
15  // 获取成功则前继出队，跟独占不同的是
16  // 会往后面结点传播唤醒的操作，保证剩下等待的线程能够尽快获取到剩下的资源。
17  setHeadAndPropagate(node, r);
18  p.next = null; // help GC
19  if (interrupted)
20  selfInterrupt(); //响应中断
21  failed = false;
22  return;//成果获取到资源且也执行了setHeadAndPropagate(node, r)方法
23  }
24  }
25
26  // p != head || r < 0
27  //如果当前节点不是头节点，或者获取资源失败尝试阻塞该节点
28  if (shouldParkAfterFailedAcquire(p, node) //让该节点前驱结点设置位signal表
示该节点应该阻塞
29  && parkAndCheckInterrupt())//阻塞
30  interrupted = true; //响应中断
31  }
32  }
33  finally {
34
35  //出现异常
36  if (failed)
37  cancelAcquire(node);
38  }
39 }
```

核心是这个doAcquireShared方法，跟独占模式的acquireQueued很像，主要区别在setHeadAndPropagate方法中， 这个方法会将node设置为head。如果当前结点acquire到了之后发现如果还有剩余资源，就调用这个方法继续唤醒并释放自己的后继节点，**后继会将这个操作传递下去。这就是PROPAGATE状态的含义。**

```
1  //将当前结点设置为头节点，如果有剩余资源可以再唤醒之后的线程
2  private void setHeadAndPropagate(Node node, int propagate) {
3   Node h = head; // Record old head for check below
4   setHead(node);
```

```
5   /*
6   * 尝试唤醒后继的结点：<br />
7   * propagate > 0说明许可还有能够继续被线程acquire;<br />
8   * 或者 之前的head被设置为PROPAGATE(PROPAGATE可以被转换为SIGNAL)说明需要往后传递;<br />
9   * 或者为null,我们还不确定什么情况。 <br />
10   * 并且 后继结点是共享模式或者为如上为null。
11   * <p>
12   * 上面的检查有点保守，在有多个线程竞争获取/释放的时候可能会导致不必要的唤醒。<br />
13   *
14   */
15   if (propagate > 0 || h == null || h.waitStatus < 0) {
16   Node s = node.next;
17   // 后继结是共享模式或者s == null（不知道什么情况）
18   // 如果后继是独占模式，那么即使剩下的许可大于0也不会继续往后传递唤醒操作
19   // 即使后面有结点是共享模式。
20   if (s == null || s.isShared())
21   // 唤醒后继结点
22   doReleaseShared();
23   }
24  }
25
26
27  //释放共享资源-唤醒后继线程并保证后继节点的资源传播 （只要还有剩余资源）
28  private void doReleaseShared() {
29   for (;;) {
30   Node h = head;
31   // 队列不为空且有后继结点
32   if (h != null && h != tail) {
33   int ws = h.waitStatus;
34   // 不管是共享还是独占只有结点状态为SIGNAL才尝试唤醒后继结点
35   if (ws == Node.SIGNAL) {
36   // 将waitStatus设置为0
37   if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
38   continue; // loop to recheck cases
39   unparkSuccessor(h);// 唤醒后继结点
40   // 如果状态为0则更新状态为PROPAGATE，更新失败则重试
41   } else if (ws == 0 && !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
42   continue; // loop on failed CAS
43   }
```

```
44   // 如果过程中head被修改了则重试。
45   if (h == head) // loop if head changed
46   break;
47   }
48  }
```

1.4.4读锁阻塞的实现：非共公平锁/公平锁

```
1  ReentrantReadWeiteLock（）-sync中的抽象方法，放在其中的FairSync（公平锁）和Nc
nfairSync（非公平锁）中实现
2  abstract boolean readerShouldBlock();
3
4  /**
5   * Returns true if the current thread, when trying to acquire
6   * the write lock, and otherwise eligible to do so, should block
7   * because of policy for overtaking other waiting threads.
8   */
9  abstract boolean writerShouldBlock();
```

```
1
2  //默认非公平锁的实现-读线程的阻塞
3  final boolean readerShouldBlock() {
4   /* As a heuristic to avoid indefinite writer starvation,
5   * block if the thread that momentarily appears to be head
6   * of queue, if one exists, is a waiting writer. This is
7   * only a probabilistic effect since a new reader will not
8   * block if there is a waiting writer behind other enabled
9   * readers that have not yet drained from the queue.
10   */
11   return apparentlyFirstQueuedIsExclusive();
12  }
13
14  final boolean apparentlyFirstQueuedIsExclusive() {
15   Node h, s; //? ?
16   return (h = head) != null &&
17   (s = h.next) != null &&
18   !s.isShared() &&
19   s.thread != null;
20   //如果头节点为非空，且头节点的下一个节点也为非空节点，且该头节点的
```

```
21   //下一个节点的状态不是共享锁模式的节点，且头节点的下一个节点的线程不为空，则返回true，
22   //当前线程需要被阻塞。
23   //反之不需要被阻塞。
24   //进行锁重入计数判断（判断是否大于读锁获取的最大值）
25  }
26
27  Node nextWaiter; //下一个节点
28
29  /**
30   * Returns true if node is waiting in shared mode.
31   */
32  final boolean isShared() {
33   return nextWaiter == SHARED;//返回当前节点的下一个节点的状态不是是共享锁模式的节点
34  }
35  ----------------------------------------------------------------------------
36
37  //公平锁实现-读线程的阻塞
38  final boolean readerShouldBlock() {
39   return hasQueuedPredecessors();
40  }//
41
42  public final boolean hasQueuedPredecessors() {
43   // The correctness of this depends on head being initialized
44   // before tail and on head.next being accurate if the current
45   // thread is first in queue.
46   Node t = tail; // Read fields in reverse initialization order
47   Node h = head;
48   Node s;
49   return h != t &&
50   ((s = h.next) == null || s.thread != Thread.currentThread());
51   //返回true:同步队列不为空且，同步队列中头节点的下一个节点不为空，当前线程就是在头节点的下一个非空节点
52   //反之：返回false;
53  }
```

1.5读锁的释放（共享锁）

```
1  //ReadLock -unlock()
```

```
//调用了AQS中的releaseShared组件
public void unlock() {
 sync.releaseShared(1);
}

public final boolean releaseShared(int arg) {
 if (tryReleaseShared(arg)) { //释放后如果允许后继等待线程获取资源返回true。
 doReleaseShared(); //反之返回false.
  return true;
  }
  return false;
 }
```

1.5.1tryReleaseShared（）方法

```
/**共享模式释放资源*/
protected boolean tryReleaseShared(int arg) {
 throw new UnsupportedOperationException();
}
```

说明：释放给定量的资源，需自定义同步器实现。释放后如果允许后继等待线程获取资源返回true。

tryReleaseShared在ReentrantReadWriteLock中由sync实现

说明：在`releaseShared`中，首先调用`tryReleaseShared`尝试释放锁，方法流程很简单，主要包括两**步：**

1. 更新当前线程计数器的锁计数；
2. CAS更新释放锁之后的state，这里使用了自旋，在state争用的时候保证了CAS的成功执行。

```
//tryReleaseShared在ReentrantReadWriteLock中由sync实现

protected final boolean tryReleaseShared(int unused) {
 Thread current = Thread.currentThread();
 if (firstReader == current) {//当前为第一个获取读锁的线程
 // assert firstReaderHoldCount > 0;
 //更新线程持有数
 if (firstReaderHoldCount == 1)
 firstReader = null; 读锁队列为空
```

```
 else
 firstReaderHoldCount--; 读锁计数-1
 } else {
 HoldCounter rh = cachedHoldCounter;
 if (rh == null || rh.tid != getThreadId(current))
 rh = readHolds.get();//获取当前线程的计数器
 int count = rh.count;
 if (count <= 1) {
 readHolds.remove();
 if (count <= 0) //计数器异常
 throw unmatchedUnlockException();
 }
 --rh.count;
 }
 for (;;) {//自旋
 int c = getState();
 int nextc = c - SHARED_UNIT; //获取剩余的资源/锁
 if (compareAndSetState(c, nextc))
 // Releasing the read lock has no effect on readers,
 // but it may allow waiting writers to proceed if
 // both read and write locks are now free.
 return nextc == 0; //是否
 }
}
```

1.5.2 doReleaseShared()释放公共资源，唤醒后继节。

、

//释放共享资源-唤醒后继线程并保证后继节点的资源传播 （只要还有剩余资源）

```
//释放共享资源-唤醒后继线程并保证后继节点的资源传播 （只要还有剩余资源）
private void doReleaseShared() {
 for (;;) {
 Node h = head;
 // 队列不为空且有后继结点
 if (h != null && h != tail) {
 int ws = h.waitStatus;
 // 不管是共享还是独占只有当前结点状态为SIGNAL才尝试唤醒后继结点
 if (ws == Node.SIGNAL) {
 // 将waitStatus设置为0
 if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
```

```
12  continue; // loop to recheck cases
13  unparkSuccessor(h);// 唤醒后继结点
14  // 如果状态为0则更新状态为PROPAGATE，更新失败则重试
15  } else if (ws == 0 && !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
16  continue; // loop on failed CAS
17  }
18  // 如果过程中head被修改了则重试。
19  if (h == head) // loop if head changed
20  break;
21  }
22 }
```

说明：在`tryReleaseShared`成功释放资源后，调用此方法唤醒后继线程并保证后继节点的release传播（通过设置head节点的`waitStatus`为`PROPAGATE`）。

doReleaseShared()解析：共享锁的所有释放于获取都有可能调用**doReleaseShared( )**
方法目的是让资源可尽可能得利用起来。
需要注意得是，在获得共享锁（读锁）的过程中如果遇到如下情况则不需要取唤醒其后继结点

1.后继结是共享模式或者s == null（不知道什么情况）
2.如果后继是独占模式，那么即使剩下的许可大于0也不会继续往后传递唤醒操作 即使后面有结点是共享模式。
3.没有剩余资源

1.5.3遇到的问题

- [x] ~~读写锁中 共享锁，跟独占锁，共用一个同步队列?~~

是

- [ ] 如何确保共享锁中读写锁的获得顺序。（默认非公平情况下）

- [ ]

共享锁的释放，只有当头节点被改变时才会跳出自旋？在没有改变头节点之前一直尝试唤醒下一个结点？