# ThreadLocal

线程利用ThreadLocal存储的原理是每一个线程类里面都有一个ThreadLocal.ThreadLocalMap变量,即ThreadLocal类内部的ThreadLocalMap内部类。该内部类还有一个内部类为Entry，作为map（ThreadLocalMap简称）的存储单元，每一个Entry接口ThreadLocal和要存储的Value。其中ThreadLocal被标记为弱引用。

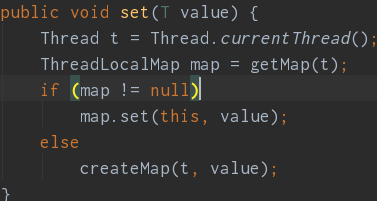
线程存储利用ThreadLocal的set方法存储变量信息的时候，会调用其内部类Map的set方法，如果map为null，则初始化，会new map赋值给thread内的map变量，因此每个线程内部都有一个自己的map实例，并非共享的。

## set方法

使用ThreadLocal存储线程局部变量的时候：

ThreadLocal<Integer> threadLocal = new ThreadLocal<>();  
threadLocal.set(5);

此时会调用ThreadLocal类的set()方法，



其中getMap(Thread thread)方法，返回该thread中的threadLocals变量，该变量为ThreadLocal中的静态内部类ThreadLocalMap，可以将其理解为一个如同Map的结构的一个对象。

如果Map为空，则会初始化，默认大小为16，会将传入的值保存。注意：Map保存的为ThreadLocal对象和要保存的值。但是每个线程的ThreadLocal对象有不一样，是怎么实现保存的呢？

1. private void set(ThreadLocal<?> key, Object value) {
3. // We don't use a fast path as with get() because it is at
4. // least as common to use set() to create new entries as
5. // it is to replace existing ones, in which case, a fast
6. // path would fail more often than not.
8. Entry[] tab = table;
9. **int** len = tab.length;
10. **int** i = key.threadLocalHashCode & (len-1);
12. **for** (Entry e = tab[i];
13. e != **null**;
14. e = tab[i = nextIndex(i, len)]) {
15. ThreadLocal<?> k = e.get();
17. **if** (k == key) {
18. e.value = value;
19. **return**;
20. }
22. **if** (k == **null**) {
23. replaceStaleEntry(key, value, i);
24. **return**;
25. }
26. }
28. tab[i] = **new** Entry(key, value);
29. **int** sz = ++size;
30. **if** (!cleanSomeSlots(i, sz) && sz >= threshold)
31. rehash();
32. }

首先查找传入的ThreadLocal所处的位置（NO.10）,然后从所处位置往后循环(只会往后)。如果发现了存在有原ThreadLocal 值，则将旧值替换为新值。若所处的位置ThreadLocal值为null，则调用replaceStaleEntry函数（稍后再讲解）。若直到entry数组最后一个，还是没有发现的话，则直接新建一个entry，然后进入No.30代码处。

这里若发现key值之前没有key为null 的节点，则不会触发清除。

replaceStaleEntry（）操作代码如下：

### replaceStaleEntry

主要功能：找到第一个threadlocal为null的节点

此时入参为 key = threadLocal(this), value = value, staleSlot = ThreadLocal哈希后的位置或者是往后搜索时的任意的第一个key为null 的位置。注意！ 进入这里的条件为哈希后的位置或往后的位置有一处存在key为null 的entry。key为null即弱引用ThreadLocal为null，表明这里应该需要被垃圾回收的，但是由于value内存泄漏，导致未被垃圾回收器回收内存。

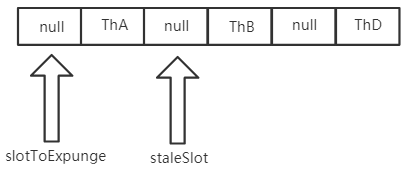
1. private **void** replaceStaleEntry(ThreadLocal<?> key, Object value,
2. **int** staleSlot) {
3. Entry[] tab = table;
4. **int** len = tab.length;
5. Entry e;
7. **int** slotToExpunge = staleSlot;
8. **for** (**int** i = prevIndex(staleSlot, len);
9. (e = tab[i]) != **null**;
10. i = prevIndex(i, len))
11. **if** (e.get() == **null**)
12. slotToExpunge = i;
14. **for** (**int** i = nextIndex(staleSlot, len);
15. (e = tab[i]) != **null**;
16. i = nextIndex(i, len)) {
17. ThreadLocal<?> k = e.get();

20. **if** (k == key) {
21. e.value = value;
23. tab[i] = tab[staleSlot];
24. tab[staleSlot] = e;
26. // Start expunge at preceding stale entry if it exists
27. **if** (slotToExpunge == staleSlot)
28. slotToExpunge = i;
29. cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
30. **return**;
31. }
33. **if** (k == **null** && slotToExpunge == staleSlot)
34. slotToExpunge = i;
35. }
37. // If key not found, put new entry in stale slot
38. tab[staleSlot].value = **null**;
39. tab[staleSlot] = **new** Entry(key, value);
41. // If there are any other stale entries in run, expunge them
42. **if** (slotToExpunge != staleSlot)
43. cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
44. }

slotToExpunge：表示为null 处table的下标，用于后续清除。

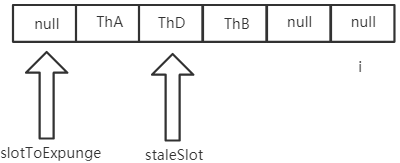
进入replaceStaleEntry方法之前，假设threadLocal对象hash值在table中的位置为下标1，即为下图的第一个ThA处。为null往后循环一次，找到第二个null处。

No.8用于往前找到最前面的为空的null的下标，赋予slotToExpunge：



若此时，我们保存的ThreadLocal为ThD，从下标为3处开始，往前找到下标为0的节点为null，然后将0赋值给slotToExpunge，然后进入第二个循环No.14，第二个循环则是往后找到要保存的ThreadLocal对象，若找到即：if (k == key) ,此时i=5，然后tab[i] = tab[staleSlot];

tab[staleSlot] = e,（e为ThD的Entry对象，保存的为ThreadLocal和value）这两步使的map发生如下变化：



完成了即：从为null的节点处往后循环，找到目标ThreadLocal对象，然后赋新值并与进入函数的那个null处的节点交换。然后进入到if (slotToExpunge == staleSlot)，即图一的staleSlot之前没有null节点，并且后面也没有null节点（No.33代码），此时slotToExpunge节点为第一个为null的节点。然后调用No.29代码，对slotToExpunge之后的null节点进行清除。怎么保证slotToExpunge为第一个null节点呢？若之前有null节点，这个判断if (k == null && slotToExpunge == staleSlot)，也无法满足，这里意思为从传入null的节点开始往后查找，找到一个为null的节点，并且之前都没有null节点，就将该null节点赋值给slotToExpunge，用于后续直接清除。注意：往后循环if (k == null && slotToExpunge == staleSlot)都不给slotToExpunge赋值了，因为他不满足等于staleSlot的条件了，即找到第一个null节点赋值给slotToExpunge。

跳出for循环，意为没有找到目标ThreadLocal对象，因此直接

1. tab[staleSlot].value = **null**;
2. tab[staleSlot] = **new** Entry(key, value);
4. // If there are any other stale entries in run, expunge them
5. **if** (slotToExpunge != staleSlot)
6. cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);

将传入的null节点赋值为目标值。然后清除slotToExpunge处往后的null节点。因为经过上一节描述的代码之后，slotToExpunge处一定为第一个为null 的节点的下标。

replaceStaleEntry()方法内部，可以看到：

1. 首先往前循环找到第一个为null 的节点赋值为slotToExpunge。因此循环中slotToExpunge == staleSlot就一直都不会满足。
2. 若往前循环不存在null节点，则程序会保证slotToExpunge为第一个null节点的下标。
3. 然后往后循环找到key值，找到的话与当前staleSlot处的null节点交换，并清除slotToExpunge往后的为null 的节点（下节介绍）。
4. 若找不到则新建节点，判断如果包含null节点则清除。

ThreadLocalMap.Entry 继承了弱引用类 WeakReference 类，而且弱引用类包裹了key值。这意味着key值是一个弱引用。一旦key值所指向的ThreadLocal没有了强引用指向，那么便会被下一次的GC回收。然后key值便会为null，但是对应的Entry对象还在，并没有释放内存，即上面的为null都是判断的ThreadLocal对象是否为null。

上述代码无论是否找到目标key所对应的Entry，都替换掉staleSlot位置的过时Entry，换上新的Entry。

找到为null的节点后，调用cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);

首先来看expungeStale代码：

### expungeStaleEntry()方法

主要功能：往后循环清理threadlocal为null的节点，碰到entry[i]=null,则调用cleanSomeSlots方法。

1. private **int** expungeStaleEntry(**int** staleSlot) {
2. Entry[] tab = table;
3. **int** len = tab.length;
4. // expunge entry at staleSlot
5. tab[staleSlot].value = **null**;
6. tab[staleSlot] = **null**;
7. size--;
9. Entry e;
10. **int** i;
11. **for** (i = nextIndex(staleSlot, len);
12. (e = tab[i]) != **null**;
13. i = nextIndex(i, len)) {
14. ThreadLocal<?> k = e.get();
15. **if** (k == **null**) {
16. e.value = **null**;
17. tab[i] = **null**;
18. size--;
19. } **else** {
20. **int** h = k.threadLocalHashCode & (len - 1);
21. **if** (h != i) {
22. tab[i] = **null**;
24. **while** (tab[h] != **null**)
25. h = nextIndex(h, len);
26. tab[h] = e;
27. }
28. }
29. }
30. **return** i;
31. }

首先往后循环，若发现ThreadLocal为null，则设置该处为null，且value为null，这里设置value为null，就释放了value的强引用，使得能够回收table[i]处的内存。如果不为null，则对该处的ThreadLocal进行再hash，如果得到的下标h不是当前下标i（即该节点是通过开放地址法推移到这里来的），则先将此处table[i]设置为null，然后从h处开始，往后遍历找到为null的地方（这里默认肯定是h<i），很有可能找到一开始传入被赋值为null的那个staleSlot节点或者i处的节点。

最后返回第一个table[i]为null的那个点（注意不是ThreadLocal为null而是table[i]为null）。

然后将为null的table[i]的下标传给cleanSomeSlots()函数：

### cleanSomeSlots

1. **private** **boolean** cleanSomeSlots(**int** i, **int** n) {
2. **boolean** removed = **false**;
3. Entry[] tab = table;
4. **int** len = tab.length;
5. **do** {
6. i = nextIndex(i, len);
7. Entry e = tab[i];
8. **if** (e != **null** && e.get() == **null**) {
9. n = len;
10. removed = **true**;
11. i = expungeStaleEntry(i);
12. }
13. } **while** ( (n >>>= 1) != 0);
14. **return** removed;
15. }

该函数执行对数级别的扫描，介于不扫描和全扫描之间。他快速找到为null的table之后ThreadLocal为null的节点，然后再次传递给expungeStale函数执行清理Value工作。

总结一下：

首先往后找，若找到threadlocal值，则替换。若找到的途中发现有threadlocal为null 的节点标记为null1，则触发replacestaleentry方法，它首先会往前循环，找到第一个为null 的节点并标记为tag，若没找到，则往后循环，找到threadlocal 的值，若找到则与null1交换，若直至最后一个都没找到，则新建一个节点并且在这途中，若之前前面没找到null节点此时会把往后第一个为null 的节点赋值给tag。然后将tag节点往后循环进行清除。清除途中发现entry[i]为null，会直接返回然后调用celanSomeSlots进行快速循环再次清理。

## get方法

1. **public** T get() {
2. Thread t = Thread.currentThread();
3. ThreadLocalMap map = getMap(t);
4. **if** (map != **null**) {
5. ThreadLocalMap.Entry e = map.getEntry(**this**);
6. **if** (e != **null**) {
7. @SuppressWarnings("unchecked")
8. T result = (T)e.value;
9. **return** result;
10. }
11. }
12. **return** setInitialValue();
13. }

首先拿到Map对象，然后调用Entry对象：

1. **private** Entry getEntry(ThreadLocal<?> key) {
2. **int** i = key.threadLocalHashCode & (table.length - 1);
3. Entry e = table[i];
4. **if** (e != **null** && e.get() == key)
5. **return** e;
6. **else**
7. **return** getEntryAfterMiss(key, i, e);
8. **}**

这里又会判断，如果e为null或者ThreadLocal为null，会进一步处理：

1. private Entry getEntryAfterMiss(ThreadLocal<?> key, **int** i, Entry e) {
2. Entry[] tab = table;
3. **int** len = tab.length;
5. **while** (e != **null**) {
6. ThreadLocal<?> k = e.get();
7. **if** (k == key)
8. **return** e;
9. **if** (k == **null**)
10. expungeStaleEntry(i);
11. **else**
12. i = nextIndex(i, len);
13. e = tab[i];
14. }
15. **return** **null**;
16. }

这里会判断如果是ThreadLocal为null，则会调用expungeStaleEntry方法进行清除。

## rehash方法

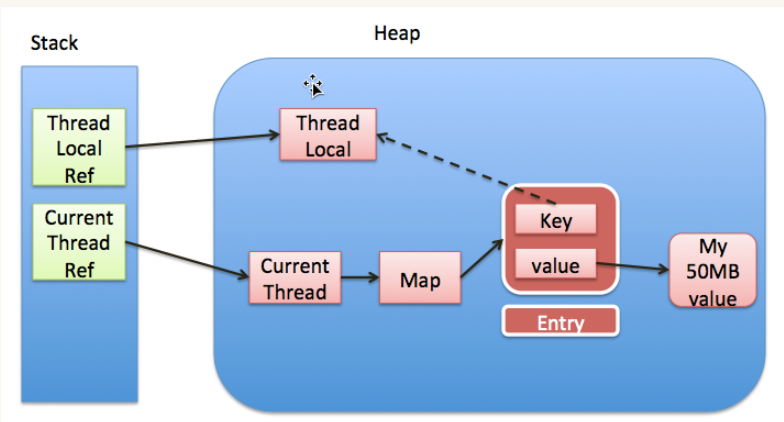
rehash会调用expungeStaleEntries方法，依次对ThreadLocal为null 的对象进行expungeStaleEntry操作。

1. private void expungeStaleEntries() {
2. Entry[] tab = table;
3. **int** len = tab.length;
4. **for** (**int** j = 0; j < len; j++) {
5. Entry e = tab[j];
6. **if** (e != **null** && e.get() == **null**)
7. expungeStaleEntry(j);
8. }
9. }

expungeStaleEntry会往后循环为null 的会清理掉，不为null的再次hash，然后位置不同的，从hash指定位置处依次往后遍历直至找到为null的空空填进去。

## 总结

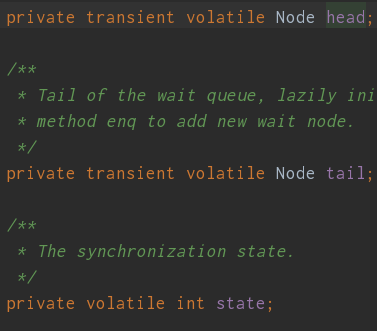
ThreadLocal保存变量的过程中，由于ThreadLocal变量是弱引用，所以只要对于ThreadLocal的强引用被消除了，那么该对象可能就会被回收掉，然而，由于Map的Entry中存储的为键值对，值此时还没有被消除，就会造成内存泄漏。



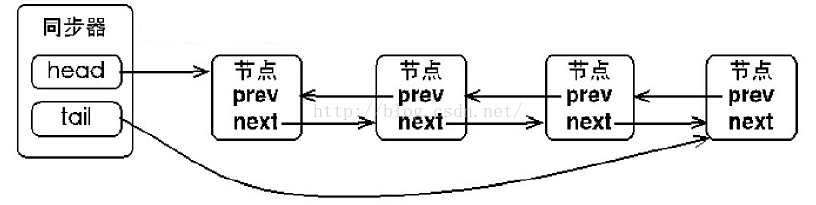
但是Map中，set、get、rehash方法都会对ThreadLocal为null的entry进行处理，消除value值，保证了ThreadLocal为null 的entry，value也为null，并且table[i]也会设置为null，此时Entry对象就可以被回收。

# ReentrantLock中的AQS

ReentrantLock中使用Sync来实现的AQS，AQS即：AbstractQueueSynchronized，抽象队列同步器，使用队列来实现同步机制。



在AbstractQueuedSynchronizer中，有三个变量，分别表示队列中的头节点和为节点以及状态（为0表示可以获取到锁，为1表示不能）。



其中Sync的实现类包括公平锁和非公平锁。首先来看非公平锁：

## NonfairSync的lock方法

例如一个ReentrantLock调用lock()想要获取锁，会发生下面过程：

**final void** lock() {  
1 **if** (compareAndSetState(**0**, **1**))  
2 setExclusiveOwnerThread(**Thread**.**currentThread**());  
3 **else  
4** acquire(**1**);  
}

第一行，首先会使用CAS修改同步器中的status，如果成功将状态从0修改为1，则代表当前线程获取到了锁，因此执行第2行，第2行方法只有一行代码：

### tryAcquire和nonfairTryAcquire

exclusiveOwnerThread = *thread*;

表示当前排外锁的拥有着为当前线程。

但是如果修改同步器状态失败，则运行第四行。

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

acquire主要执行三个方法，首先看第一个tryAcquire()方法，他直接返回nonfairTryAcquire(acquires)，接着往里看：

**final boolean** nonfairTryAcquire(**int** *acquires*) {  
 **final Thread** current = **Thread**.**currentThread**();  
1 **int** c = getState();  
2 **if** (c == **0**) {  
3 **if** (compareAndSetState(**0**, *acquires*)) {  
4 setExclusiveOwnerThread(current);  
5 **return true**;  
6 }  
 }  
7 **else if** (current == getExclusiveOwnerThread()) {  
8 **int** nextc = c + *acquires*;  
9 **if** (nextc < **0**) // overflow  
 **throw new** Error(**"Maximum lock count exceeded"**);  
10 setState(nextc);  
 **return true**;  
 }  
 **return false**;  
}

首先第一行，getStatus()方法为父类AbstractQueuedSynchronizer类的方法，返回同步器状态。第2行，如果状态为0，则表示又可以尝试获取锁，因此再次调用CAS尝试修改锁，成功则直接返回true。获取到锁，结束。注意同步器状态不为0，有两种情况，一种是当前锁拥有着不是目前线程，还有一种为当前锁拥有着为本线程。注意前面的图看lock()方法，CAS0到1，不成功，同样也有可能是当前线程已经拥有了锁，所以到第8行，判断如果锁拥有着为本线程，则进行锁的重入实现，即nextc数量加1。如果状态不为0，并且当前线程不是锁拥有着，这返回false。

### addWaiter和acquireQueued方法：

再次尝试获取锁失败后，首先进入进入addWaiter()方法，传入null，

**private Node** addWaiter(**Node** *mode*) {  
 **Node** node = **new** Node(**Thread**.**currentThread**(), *mode*);  
 // Try the fast path of enq; backup to full enq on failure  
 **Node** pred = tail;  
 **if** (pred != **null**) {  
 node.prev = pred;  
 **if** (compareAndSetTail(pred, node)) {  
 pred.next = node;  
 **return** node;  
 }  
 }   
 enq(node);  
 **return** node;  
}

这个方法将当前线程放入等待队列中，并且设置为尾节点。compareAndSetTail(pre,node)方法，看名称意为将尾节点从pre设置为node.

Enq(node):

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

为初始化队列，并将tail设置为node节点。Enq()之后，addWauter方法返回node节点。（**这里为什么要设置为死循环呢？**

因为前面addWaiter里面，CAS将当前节点设置为尾节点，但是如果失败了，就还是会进入end方法，这里会进行死循环直至将当前节点放入到队列中。）

此时node节点：



然后acquire（1）就到了acquireQueued(node,arg)方法中，此时node为当前线程的节点，arg为1；

**final boolean** acquireQueued(**final Node** *node*, **int** *arg*) {  
 **boolean** failed = **true**;  
 **try** {  
 **boolean** interrupted = **false**;  
 **for** (;;) {  
 **final Node** p = *node*.predecessor();  
1 **if** (p == head && tryAcquire(*arg*)) {  
 setHead(*node*);  
 p.next = **null**; // help GC  
 failed = **false**;  
 **return** interrupted;  
 }  
 **if** (**shouldParkAfterFailedAcquire**(p, *node*) &&  
 parkAndCheckInterrupt())  
 interrupted = **true**;  
 }  
 } **finally** {  
 **if** (failed)  
 cancelAcquire(*node*);  
 }  
}

predecessor()方法：

**final Node** predecessor() **throws NullPointerException** {  
 **Node** p = prev;  
 **if** (p == **null**)  
 **throw new** NullPointerException();  
 **else  
 return** p;  
}

返回node节点的前置节点。然后返回到acqyureQueued方法。

final Node p = node.predecessor();第一遍读起来感觉这个死循环没用，每一次都获取的同一个节点，但其实死循环不是为了一个一个循环队列节点，而是保证能够阻塞。并且每个线程执行的节点不一样，各个线程管理各个的节点，自己判断前面的节点是否为头节点，不是阻塞就好了。

判断p是否是head节点，如果是head节点，会尝试获取锁，tryAcquire方法在上面已经分析过。获取锁成功之后，sethead(node)会把node节点置为头节点，p.next = null将之前的head节点指向断掉，帮助jvm触发GC。最后返回当前线程在获取锁过程中是否曾经被中断。

如果node.prev不是头节点，不会尝试获取锁，这也就是AQS内部链表的作用，会从链表的头部开始尝试获取锁，达到一个FIFO的作用。获取锁失败或者node.prev不是头节点，则会执行shouldParkAfterFailedAcquire：

**private static boolean** shouldParkAfterFailedAcquire(**Node** pred, **Node** *node*) {  
 **int** ws = *pred*.waitStatus;  
 **if** (ws == **Node**.***SIGNAL***)  
 */\*  
 \* This node has already set status asking a release  
 \* to signal it, so it can safely park.  
 \*/* **return true**;  
 **if** (ws > **0**) {  
 */\*  
 \* Predecessor was cancelled. Skip over predecessors and  
 \* indicate retry.  
 \*/* **do** {  
 *node*.prev = pred = pred.prev;  
 } **while** (pred.waitStatus > **0**);  
 pred.next = *node*;  
 } **else** {  
 */\*  
 \* waitStatus must be 0 or PROPAGATE. Indicate that we  
 \* need a signal, but don't park yet. Caller will need to  
 \* retry to make sure it cannot acquire before parking.  
 \*/* **compareAndSetWaitStatus**(pred, ws, **Node**.***SIGNAL***);  
 }  
 **return false**;  
}

shouldParkAfterFailedAcquire方法有三个作用：1、若pred.waitStatus状态位大于0，说明这个节点已经取消了获取锁的操作，doWhile循环会递归删除掉这些放弃获取锁的节点。2、若状态位不为Node.SIGNAL,且没有取消操作，则会尝试将状态位修改为Node.SIGNAL。3、状态位是Node.SIGNAL，表明线程是否已经准备好被阻塞并等待唤醒。

最终，只有在pred.waitStatus已经等于Node.SIGNAL时才会返回true。其他情况返回false，然后acquireQueued会继续循环。

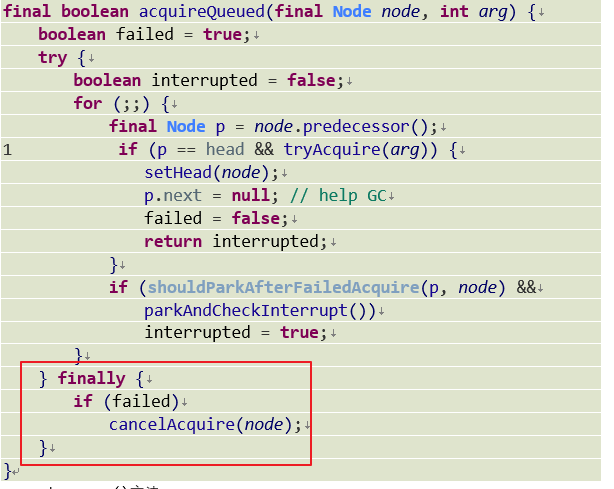
**注意:在默认情况下，new的node的waitStatus为0，循环一次就将前面一个节点的waitStatus改为Node.SIGNAL，再次进入循环判断前面一个节点的waitStatus，如果为SIGNAL则阻塞当前线程。**

在shouldParkAfterFailedAcquire返回true之后，acquireQueued方法体内继续执行parkAndCheckInterrupt()：

**private final boolean** parkAndCheckInterrupt() {  
 **LockSupport**.**park**(**this**);  
 **return Thread**.**interrupted**();  
}

该方法调用LockSupport.park()方法使线程阻塞。注意，ReentrantLock.lock()获取锁阻塞就是在这一步实现。阻塞的线程在其他线程释放锁之后会被LockSupport.unpark()唤醒。LockSupport.park(),LockSupport.unpark()最终都是调用了UNSAFE的native方法，这里不做分析。整个ReentrantLock.lock方法就分析到这里。

LockSupport.park()会导致当前线程阻塞(park(Object object)暂时也这么理解)，然后调用unpark的时候就会通知传入的线程。



最后这里，在线程lock的时候，如果取消了的话，会进入这里，会将等待队列里面的当前节点删除掉。

## NonFairSync的Unlock()方法

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

会调用起父类AbstractQueuedSynchronizer的release()方法：

### release

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

首先会尝试释放锁，即调用方法tryRelease:

**protected final boolean tryRelease**(**int** releases) {  
 **int** c = **getState**() - releases;  
 **if** (Thread.**currentThread**() != **getExclusiveOwnerThread**())  
 **throw new IllegalMonitorStateException**();  
 **boolean** free = **false**;  
 **if** (c == 0) {  
 free = **true**;  
 **setExclusiveOwnerThread**(**null**);  
 }  
 **setState**(c);  
 **return** free;  
}

该方法会对重用锁数量进行判断，若status -1 = 0，则表示当前只有一个线程锁，调用此函数后就已经释放完毕锁，因此设置锁的线程拥有着为null，修改status，然后返回true。但若不为零，则表示还没有完全释放锁，只是释放了“一层”，所以只修改状态返回false。然后返回上层release()方法。tryRelease()返回false则返回。

若释放了锁则进入if语句，将调用unparkSuccessor()方法。

### unparkSuccessor

**private void unparkSuccessor**(Node node) {  
**int** ws = node.waitStatus;  
 **if** (ws < 0)  
 **compareAndSetWaitStatus**(node, ws, 0);  
Node s = node.next;  
 **if** (s == **null** || s.waitStatus > 0) {  
 s = **null**;  
 **for** (Node t = tail; t != **null** && t != node; t = t.prev)  
 **if** (t.waitStatus <= 0)  
 s = t;  
 }  
 **if** (s != **null**)  
 LockSupport.**unpark**(s.thread);  
}

设置node状态为0，然后获取下一个节点，若s为null，或者waitStatus>0（表示取消状态），则从尾节点开始往前遍历，找到状态小于0的节点，然后将该节点释放（unPark(thread)）。这样就实现了从节点lock到获取锁失败到加入阻塞队列到被LockSupport阻塞到unLock()到释放下一节点的全程。

## FairSync的Lock()方法

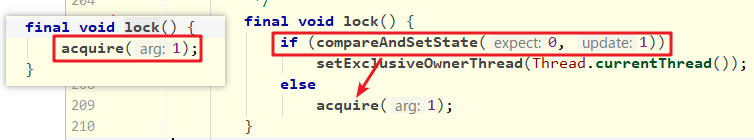
同样是首先调用AbstractQueuedSynchronizer的acquire()方法：

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

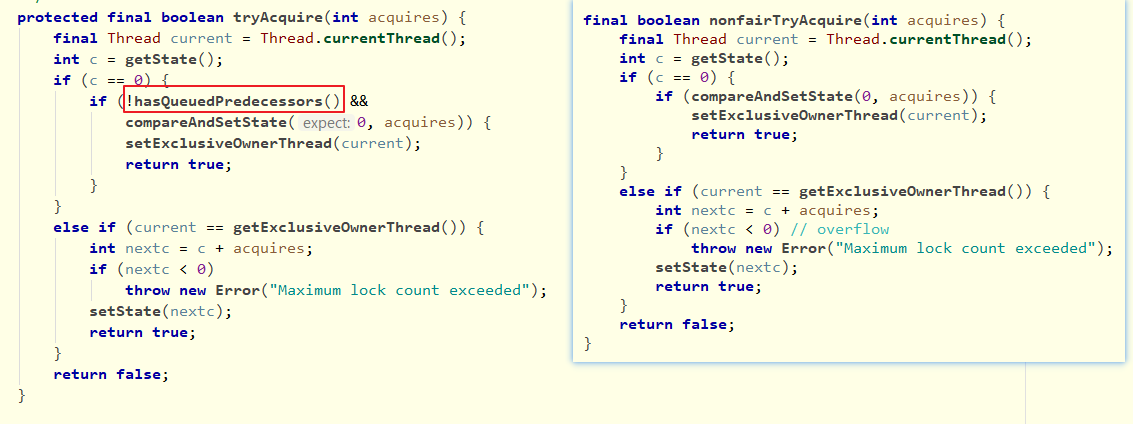
首先调用tryAcquire(arg)方法:

protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

与NonFairSync的lock方法不同有两个，NonFairSync在调用 acquire()之前会首先尝试获得锁。而FairSync则直接调用acquire()。



非公平锁进入acquire时如果state=0，则会直接CAS修改。非公平锁在state=0的时候，会首先判断等待队列是否为空，若不为空才会CAS status.



# LockSupport：

补充：LockSupport用法：

public static void main(String[] args) throws InterruptedException {  
 Thread parkThread = new Thread(new ParkThread());  
 parkThread.start();  
 System.*out*.println("开始线程唤醒");  
 LockSupport.*unpark*(parkThread);  
 System.*out*.println("结束线程唤醒");  
}  
  
static class ParkThread implements Runnable {  
  
 @Override  
 public void run() {  
 System.*out*.println("开始线程阻塞");  
 LockSupport.*park*();  
 System.*out*.println("结束线程阻塞");  
 }  
}

可以先唤醒再阻塞也能够正常执行。

LockSupport有一个变量permit标记是否阻塞，

1. 调用park的时候，如果当前线程的permit为1，则将permit设置为0，并立即返回（并不会阻塞），如果当前线程的permit为0，则将当前线程阻塞，知道别的线程将permit设置为1（调用unPark()），park会将permit方法再次设置为0，并返回。也就是说，park方法返回后permit的值一定会是0的。

2. 调用unPark()的时候，会将permit置位1（注意多次调用unPark方法pemrit的值还是会为1）

# Callable和Future与线程池的关系

@FunctionalInterface  
public interface Callable<V> {  
 */\*\*  
 \* Computes a result, or throws an exception if unable to do so.  
 \*  
 \** ***@return*** *computed result  
 \** ***@throws*** *Exception if unable to compute a result  
 \*/* V call() throws Exception;  
}

Callable仅有一个方法，call()方法。可以返回计算结果。

public interface Future<V> {boolean cancel(boolean mayInterruptIfRunning);boolean isCancelled();boolean isDone();V get() throws InterruptedException, ExecutionException;V get(long timeout, TimeUnit unit)  
 throws InterruptedException, ExecutionException, TimeoutException;  
}

Future也是一个接口，FutureTask实现了该接口，并且也实现了Runable。构造方法接收Callable或Runnable:

public FutureTask(Callable<V> callable) {  
 if (callable == null)  
 throw new NullPointerException();  
 this.callable = callable;  
 this.state = *NEW*; // ensure visibility of callable  
}

public FutureTask(Runnable runnable, V result) {  
 this.callable = Executors.*callable*(runnable, result);  
 this.state = *NEW*; // ensure visibility of callable  
}

目前就可以将Callable传给FutureTask，然后调用get()方法获取到线程的返回值

线程池是为了解决线程切换带来的影响的一个工具，线程池中会限定线程的个数（或者不会），有空闲的线程就会让空闲的线程来执行任务。线程是稀缺资源，使用线程池可以减少创建和先回线程的次数，每个工作线程都可以重复使用。可以根据系统的承受能力，调整线程池中工作线程的数量，防止因为消耗过多内存导致服务器崩溃。

## 线程池的创建



**corePoolSize**:线程中核心线程的最大值。核心线程默认情况下会一直存活在线程池中，即使这个核心线程啥也不干(闲置状态)。超过核心线程数量后新创建的线程为非核心线程。

**maximumPoolSize**：线程池中的最大线程数。

**keepAliveTime**：非核心线程超过这个时间没有使用就会被销毁。但是如果设置了allowCoreThreadTimeOut =true，就会作用于核心线程。

**Unit**：时间单位。

**workQueue**：当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果队列满了，则新建非核心线程执行任务。

**threadFactory**：创建线程的方式，这是一个接口，new它的时候需要实现他的Thread newThread(Runnable r)方法。

**RejectedExecutionHandler handler** ：这个主要是用来抛异常的。

## 处理任务的流程

**线程池新添加了任务，那么线程池是如何处理这些批量任务？**

1：如果线程数量未达到corePoolSize，则新建一个线程(核心线程)执行任务

2：如果线程数量达到了corePools，则将任务移入队列等待

3：如果队列已满，新建线程(非核心线程)执行任务

4：如果队列已满，总线程数又达到了maximumPoolSize，就会由RejectedExecutionHandler抛出异常

# CAS关键方法

**boolean** compareAndSwapObject(Object var1, **long** var2, Object var4, Object var5);

意为:对象var1的var2的内存块由旧值var4赋值为var5

**boolean** compareAndSwapInt(Object var1, **long** var2, **int** var4, **int** var5);

同理；

**boolean** compareAndSwapLong(Object var1, **long** var2, **long** var4, **long** var6);

同理；

# CountDownLatch

countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。调用await方法使当前线程阻塞直到规定数量的其他线程调用countDown()到计数器为0。最主要就是两个方法,await()和countDown()方法。

该类基于aqs进行阻塞和非阻塞。主要使用到了aqs的state变量，表示还有多少个线程没有调用countDown()方法。

## Await()方法 ：

**public void await**() **throws** InterruptedException {  
 sync.**acquireSharedInterruptibly**(1);  
}

进入如下方法：

1. **public** **final** **void** acquireSharedInterruptibly(**int** arg)
2. **throws** InterruptedException {
3. **if** (Thread.interrupted())
4. **throw** **new** InterruptedException();
5. **if** (tryAcquireShared(arg) < 0)
6. doAcquireSharedInterruptibly(arg);
7. }

直接看5行代码，**tryAcquireShared**里面判断state变量的值是否为0，如果为0，则代表没有线程在执行，则返回1，否则返回-1，因此如果有线程还未执行完毕，则会进入if方法体。该方法如下：

1. **private** **void** doAcquireSharedInterruptibly(**int** arg)
2. **throws** InterruptedException {
3. **final** Node node = addWaiter(Node.SHARED);
4. **boolean** failed = **true**;
5. **try** {
6. **for** (;;) {
7. **final** Node p = node.predecessor();
8. **if** (p == head) {
9. **int** r = tryAcquireShared(arg);
10. **if** (r >= 0) {
11. setHeadAndPropagate(node, r);
12. p.next = **null**; // help GC
13. failed = **false**;
14. **return**;
15. }
16. }
17. **if** (shouldParkAfterFailedAcquire(p, node) &&
18. parkAndCheckInterrupt())
19. **throw** **new** InterruptedException();
20. }
21. } **finally** {
22. **if** (failed)
23. cancelAcquire(node);
24. }
25. }

**addWaiter()**方法会将当前线程加入到aqs队列中（作为尾节点）。然后判断前一个节点是否为头结点。若不为头结点，则阻塞(第17行，其实实际会执行两次循环之后才阻塞，第一次循环由于node是新添加的节点，所以会将node节点的waiteStatus设置为SIGNAL，然后就退出，第二次如果node的waiteStatus为SIGNAL，则会阻塞返回true，然后执行parkAndCheckInterrupt方法，该方法会阻塞当前线程)，若前一个节点是头结点，则再次调用第9行的**tryAcquireShared()**,该方法会检测state，若等于0(则返回1)，即所有线程都执行完毕了，则会执行setHeadAndPropagate()方法，

**private void setHeadAndPropagate**(Node node, **int** propagate) {  
 Node h = head; // Record old head for check below  
 **setHead**(node);**if** (propagate > 0 || h == **null** || h.waitStatus < 0 ||  
 (h = head) == **null** || h.waitStatus < 0) {  
 Node s = node.next;  
 **if** (s == **null** || s.**isShared**())  
 **doReleaseShared**();  
 }  
}

该方法会将当前节点设置为头结点,并且判断，如果下一个节点被表明为共享锁的节点(在调用第三行addWaiter方法的时候，传入的Node.SHARE就表明该节点为共享锁节点)，则会调用 **doReleaseShared()**。会唤醒AQS中的下一个节点。核心代码如下：

Node s = node.next;  
**if** (s == **null** || s.waitStatus > 0) {  
 s = **null**;  
 **for** (Node t = tail; t != **null** && t != node; t = t.prev)  
 **if** (t.waitStatus <= 0)  
 s = t;  
}  
**if** (s != **null**)  
 LockSupport.**unpark**(s.thread);

唤醒下一个队列之后就会回到**doAcquireSharedInterruptibly**()方法的自选循环中，直到该线程等待的其他线程执行完毕。(实际上一个线程在CountDownLatch的await中被唤醒，也就意味着别的线程都执行完毕了即state=0，也就不用一直自旋了)

若返回-1，即还有线程未完成，则进行**if** (**shouldParkAfterFailed**

**Acquire**(p, node) && **parkAndCheckInterrupt**())，也就是判断将ws设置为Signal，然后**parkAndCheckInterrupt**方法调用LockSupport.park()使得当前线程阻塞。

发现的问题：

1. 考虑到目前等待的线程被唤醒，他只会唤醒下一个的等待线程，那么万一有n个线程都在等待这个CountDownLatch，那其他的线程会怎么被唤醒。（所以就要详细的看唤醒的过程）

回答：当目前这个线程被唤醒的时候会首先调用**setHeadAndPropagate()方法，该方法会唤醒后面等待的共享锁节点。**

1. ReentranLock中的AQS中状态与唤醒的关系

**setHeadAndPropagate方法最后会判断后续节点的nextWaite是否为Share节点，如果是才唤醒**

## countDown()方法

1. **public** **void** countDown() {
2. sync.releaseShared(1);
3. }

代码1

可以看到首先调用sync中的releaseShare()方法.

1. **public** **final** **boolean** releaseShared(**int** arg) {
2. **if** (tryReleaseShared(arg)) {
3. doReleaseShared();
4. **return** **true**;
5. }
6. **return** **false**;
7. }

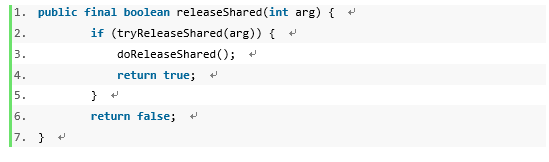
代码2

首先看第2行的treRealXXX()方法。

1. **protected** **boolean** tryReleaseShared(**int** releases) {
2. // Decrement count; signal when transition to zero
3. **for** (;;) {
4. **int** c = getState();
5. **if** (c == 0)
6. **return** **false**;
7. **int** nextc = c-1;
8. **if** (compareAndSetState(c, nextc))
9. **return** nextc == 0;
10. }
11. }

进来就进入死循环，然后如果state=0，即初始化的时候=0。或者还有其他线程未执行完毕，就返回false，否则返回true。

这时候回到代码2



如果还有其他线程在执行，那么当前线程将state-1后就返回了。但是如果当前线程是最后一个线程，即state-1==0了，则会进行第三行代码。

1. **private** **void** doReleaseShared() {
2. **for** (;;) {
3. Node h = head;
4. **if** (h != **null** && h != tail) {
5. **int** ws = h.waitStatus;
6. **if** (ws == Node.SIGNAL) {
7. **if** (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
8. **continue**;            // loop to recheck cases
9. unparkSuccessor(h);
10. }
11. **else** **if** (ws == 0 &&
12. !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
13. **continue**;                // loop on failed CAS
14. }
15. **if** (h == head)                   // loop if head changed
16. **break**;
17. }
18. }

代码4

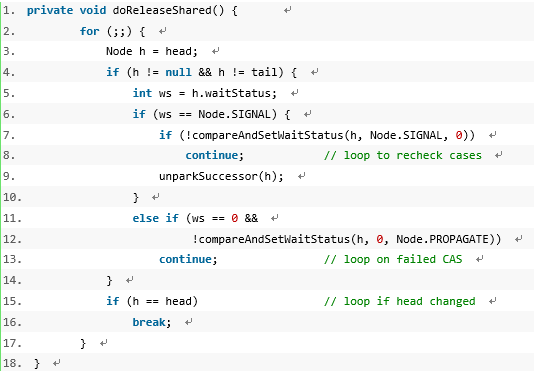
如果等待队列里面head!=null 并且head!= tail（表示内部>=1个线程在等待）则会进循环体，第五行代码。

如果h的waitStatus等于SIGNAL的话(**这里之前一直有疑问，新建的Node的时候，head的waitStatus不是初始化为0吗？这里为什么判断==-1，其实因为上面提到了，新加入节点会把前一个节点的waitStatus设置为-1，所以只要这里判断当前节点的waitStatus为-1的话，就会唤醒下一个节点**)。就将head的waiteStatus设置为0，然后调用unparkSuccess()方法，并将头结点传入。

1. **private** **void** unparkSuccessor(Node node) {
2. **int** ws = node.waitStatus;
3. **if** (ws < 0)
4. compareAndSetWaitStatus(node, ws, 0);
5. Node s = node.next;
6. **if** (s == **null** || s.waitStatus > 0) {
7. s = **null**;
8. **for** (Node t = tail; t != **null** && t != node; t = t.prev)
9. **if** (t.waitStatus <= 0)
10. s = t;
11. }
12. **if** (s != **null**)
13. LockSupport.unpark(s.thread);
14. }

代码5

获取头结点的下一个节点，如果下一个节点被中断（第6行）则会从后往前便利取出没有被中断的第一个节点，然后将该节点的线程唤醒。之后回到代码4的15行：



此时判断如果h==head，则跳出循环。回到await方法中。

**问题1：代码4中的15行代码：if(h == head)的作用**

**问题2：代码4中的11行代码将头结点状态设置为PROPAGATE作用**

## 简单描述：

await()方法：如果还有未运行完毕，则将当前线程new一个节点，放到aqs队列中去，并且要经历两次循环，则将前面一个节点的waitStatus设置为-1，第二次进行阻塞。并且两次循环都要判断前面一个节点是不是头结点，如果是，则查询等待的线程运行完没有，如果没完那么往下进行循环。如果完了，则 讲当前节点设置为head，然后唤醒nextWaiter=SHARED的head.next节点。

countDown()方法：计数器减一，如果减一之后，计数为0，则唤醒下一个waiteStatus状态=-1的节点。此时就回到了await两次循环中的第二次被阻塞的时候。

# CountDownLatch与ReentranLock加锁和解锁的时候的相同与不同之处

加锁的时候：

1. 创建等待节点的时候，前者节点的waitStatus状态为SHARED，后者为EXCLUSIVE，这里设置的状态在后续通知的时候会用到。
2. 前者被唤醒的时候，会将当前节点设置为头结点并继续唤醒队列中的后续waiteStatus状态为SHARED的节点线程。后者被唤醒的时候，就只是将当前线程设置为head就完了。

解锁的时候：

1. 前者在头结点waiteStatus=0的时候（初始化？），将其状态设置为PROPAGATE。后者是==0，就直接返回，不等于0才唤醒head的next节点，

# Semaphore

构造函数：

1. **public** Semaphore(**int** permits) {
2. sync = **new** NonfairSync(permits);
3. }

内部还是使用的AQS，这里将AQS的state设置为入参。

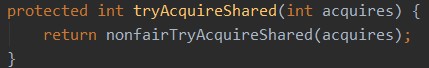
## Acquire

获取对象锁的方法：

1. **public** **void** acquire() **throws** InterruptedException {
2. sync.acquireSharedInterruptibly(1);
3. }
4. **public** **final** **void** acquireSharedInterruptibly(**int** arg)
5. **throws** InterruptedException {
6. **if** (Thread.interrupted())
7. **throw** **new** InterruptedException();
8. **if** (tryAcquireShared(arg) < 0)
9. doAcquireSharedInterruptibly(arg);
10. }

代码1

首先判断是否被中断，然后调用tryAcquireShared()。



1. **final** **int** nonfairTryAcquireShared(**int** acquires) {
2. **for** (;;) {
3. **int** available = getState();
4. **int** remaining = available - acquires;
5. **if** (remaining < 0 ||
6. compareAndSetState(available, remaining))
7. **return** remaining;
8. }
9. }

该方法将状态值减1，然后判断如果小于0则直接返回减1后的值。如果不小于1，则将减1后的值赋给旧值。并返回减1后的值。

然后就返回到代码1的第5行，判断如果小于0，则代表当前锁已经被占了，则进入doAcquireSharedInterruptibly()方法。

1. **private** **void** doAcquireSharedInterruptibly(**int** arg)
2. **throws** InterruptedException {
3. **final** Node node = addWaiter(Node.SHARED);
4. **boolean** failed = **true**;
5. **try** {
6. **for** (;;) {
7. **final** Node p = node.predecessor();
8. **if** (p == head) {
9. **int** r = tryAcquireShared(arg);
10. **if** (r >= 0) {
11. setHeadAndPropagate(node, r);
12. p.next = **null**; // help GC
13. failed = **false**;
14. **return**;
15. }
16. }
17. **if** (shouldParkAfterFailedAcquire(p, node) &&
18. parkAndCheckInterrupt())
19. **throw** **new** InterruptedException();
20. }
21. } **finally** {
22. **if** (failed)
23. cancelAcquire(node);
24. }
25. }

该方法先添加一个节点，然后设置waitStatus的值，然后阻塞。被唤醒后，判断如果减1后的state>0，则说明还允许线程加入，则设置头结点为当前节点，然后唤醒后面的节点。

**这是一个典型的共享锁的结构**

## Release

Release方法调用该方法，传入1：

1. **public** **final** **boolean** releaseShared(**int** arg) {
2. **if** (tryReleaseShared(arg)) {
3. doReleaseShared();
4. **return** **true**;
5. }
6. **return** **false**;
7. }

代码1

首先调用tryReleaseShared方法：

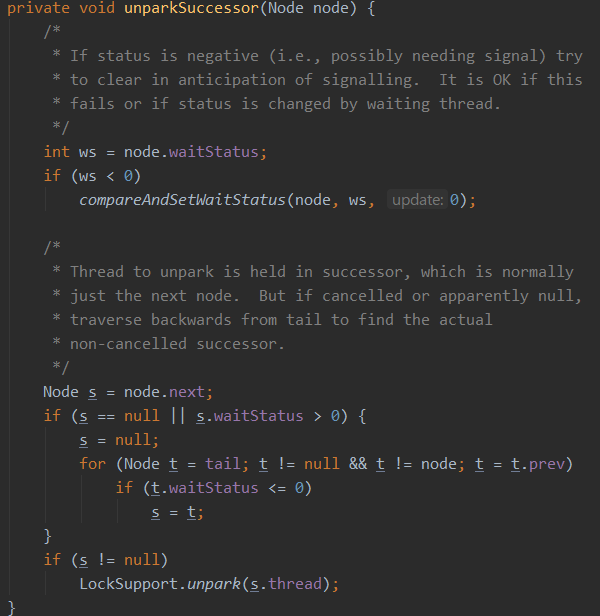
1. **protected** **final** **boolean** tryReleaseShared(**int** releases) {
2. **for** (;;) {
3. **int** current = getState();
4. **int** next = current + releases;
5. **if** (next < current) // overflow
6. **throw** **new** Error("Maximum permit count exceeded");
7. **if** (compareAndSetState(current, next))
8. **return** **true**;
9. }
10. }

他会将AQS的state加1，然后将值赋给旧的state就返回true。

这时返回到代码1的第3行，调用doReleaseShared方法：

1. **private** **void** doReleaseShared() {
2. **for** (;;) {
3. Node h = head;
4. **if** (h != **null** && h != tail) {
5. **int** ws = h.waitStatus;
6. **if** (ws == Node.SIGNAL) {
7. **if** (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
8. **continue**;            // loop to recheck cases
9. unparkSuccessor(h);
10. }
11. **else** **if** (ws == 0 &&
12. !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
13. **continue**;                // loop on failed CAS
14. }
15. **if** (h == head)                   // loop if head changed
16. **break**;
17. }
18. }

该方法会调用unparkSuccessor()方法，

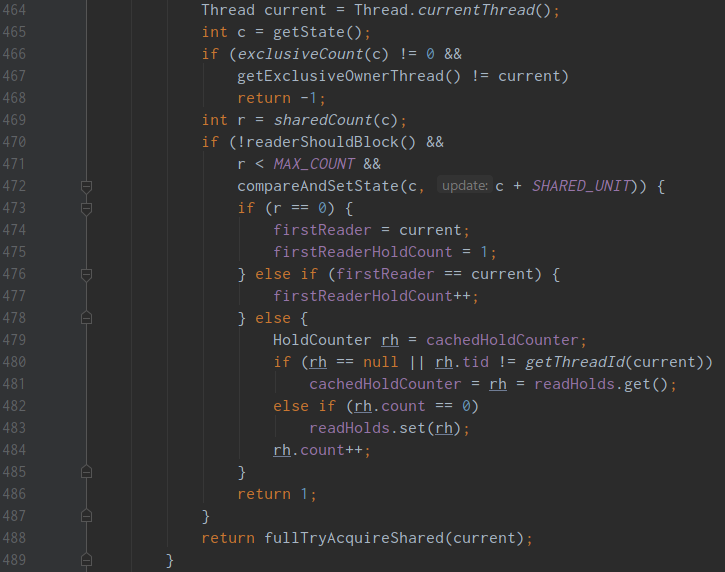


这个方法主要就是唤醒头结点的下一个节点的线程。

# ReadWriteLock

## 读锁获取：

获取读锁代码和获取共享锁代码差不多，重写了tryAcquireShared()方法用于判断是否能够获得写锁。

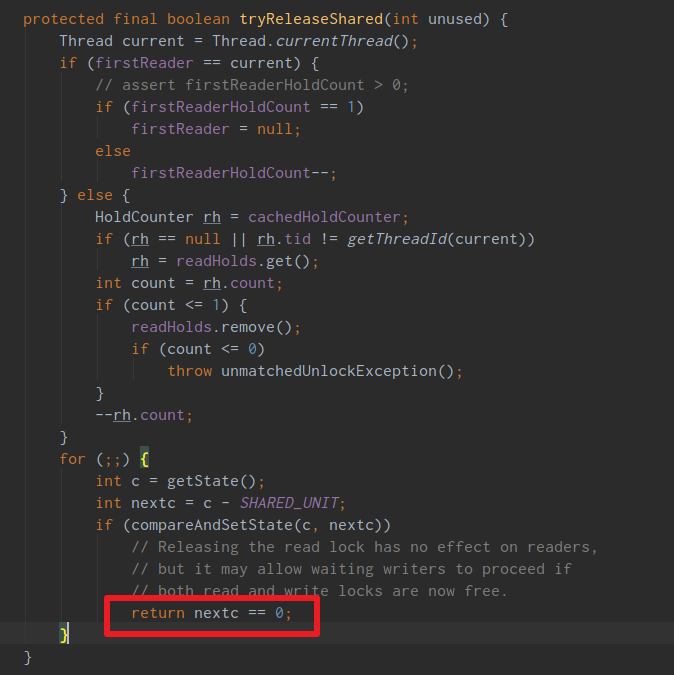


简单来说读锁获取失败，只有下面情况： 持有写锁的线程不是本线程(466,467代码) 或者读锁线程数已达到最大数量。

如果都不满足，即可以获得读锁后，将state读锁数量加一，如果加之前读锁=0，则设置firstReader为当前线程，fiestReaderHoldCount=1。如果当前firstReader为当前线程，则只设置fiestReaderHoldCount++,**即**如果第一个读的线程为自己，则设置fiestReaderHoldCount++。但如果不止一个线程获的了该读锁并且第一个读线程不为自己，则进入478行代码。cachedHoldCounter表示最后一个线程读锁的数量。readHolds用来记录当前线程持有的读锁数量(**存疑**，482,483代码看不懂。不知道具体考虑的哪种场景)。

firstReader用来记录第一个获取到读锁的线程，firstReadHoldCount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。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，

## 读锁释放



写锁释放完毕才会唤醒等待队列。释放等待队列第一个节点。

## 写锁获取

获取写锁失败：1.有读锁，写线程为空或写线程不为当前线程。

有读锁且写锁为当前线程就成功了(这种情况一定是先获取了写锁在获取读锁的，参考获的写锁失败的第2种情况)。

### 公平锁：

获取写锁失败：没有读写锁的情况下，等待队列的第一个等待节点的线程不是当前线程。

### 非公平锁：

获取写锁成功：1. 没有读写锁的情况下，直接获取。

获的读锁失败后，会加入AQS，当读线程运行完毕后，会唤醒后面线程，如果后面是读线程，则被唤醒的线程会依次唤醒后面的非独占线程（读线程）。因此在AQS队列中，会有顺序性。

## 写锁释放

如果独占锁数量为0则唤醒等待队列第一个线程。

