

主讲老师: Fox

【有道云笔记】8.1 Semaphore源码分析

<https://note.youdao.com/s/BtfAwtQe>

## Semaphore源码分析

Semaphore基于 AQS + CAS 实现的，可根据构造参数的布尔值，选择使用公平锁，还是非公平锁。Semaphore默认使用非公平锁。

Semaphore详情如下：

The image shows a screenshot of the Semaphore class in the Java standard library, with several methods highlighted by red boxes and annotated with red arrows and text. The annotations are as follows:

- Semaphore实现了公平锁、非公平锁**: Points to the `Sync`, `NonfairSync`, and `FairSync` inner classes in the left sidebar.
- 阻塞获取一个信号量**: Points to the `acquire(): void` method.
- 归还一个信号量给Semaphore**: Points to the `release(): void` method.
- 阻塞获取指定的信号量**: Points to the `acquire(int): void` method.
- 归还指定信号量给Semaphore**: Points to the `release(int): void` method.
- 是否是公平锁**: Points to the `isFair(): boolean` method.
- 获取等待队列中挂起线程信息**: Points to the `getQueuedThreads(): Collection<Thread>` method.

The code snippet on the right shows the following lines:

```
150 * actions following a successful "
151 * in another thread.
152 *
153 * @since 1.5
154 * @author Doug Lea
155
156 public class Semaphore implements java.util.concurrent.locks.Lock {
157     private static final long serialVersionUID = 1L;
158     /** All mechanics via AbstractQueuedSynchronizer. */
159     private final Sync sync;
160
161     /**
162      * Synchronization implementation for this Semaphore. Subclasses
163      * must implement this method.
164      *
165      * @return the Sync object for this Semaphore
166      */
167     abstract Sync getSync();
168
169     Sync(int permits) {
170         setState(permits);
171     }
172 }
```

## 构造函数

```

1 // AQS的实现
2 private final Sync sync;
3
4 // 默认使用非公平锁
5 public Semaphore(int permits) {
6     sync = new NonfairSync(permits);
7 }
8
9 // 根据fair布尔值选择使用公平锁还是非公平锁
10 public Semaphore(int permits, boolean fair) {
11     sync = fair ? new FairSync(permits) : new NonfairSync(permits);
12 }

```

## 公平锁与非公平锁

Semaphore中公平锁与非公平锁的实现，可以在tryAcquireShared()方法中找到两种锁的区别。

### NonfairSync

Semaphore#NonfairSync#tryAcquireShared() 详情如下

```

1 // 非公平锁 获取信号量
2 protected int tryAcquireShared(int acquires) {
3     return nonfairTryAcquireShared(acquires);
4 }

```

Semaphore#Sync#nonfairTryAcquireShared() 详情如下

```

1 // 非公平锁 获取信号量
2 final int nonfairTryAcquireShared(int acquires) {
3     // 自旋
4     for (;;) {
5         // 获取Semaphore中可用的信号量数
6         int available = getState();
7         // 当前可用信号量数 - acquires
8         int remaining = available - acquires;
9         // 可用信号量数不足 或 CAS操作获取信号量失败, 返回 当前可用信号量数 - acquires
10        if (remaining < 0 ||
11            compareAndSetState(available, remaining))
12            return remaining;
13    }
14 }

```

## FairSync

Semaphore#FairSync#tryAcquireShared() 详情如下

```

1 protected int tryAcquireShared(int acquires) {
2     // 自旋
3     for (;;) {
4         // 等待队列中挂起线程, 返回-1 (根据返回的-1, 将当前线程添加到等待队列中)
5         if (hasQueuedPredecessors())
6             return -1;
7         // 尝试获取Semaphore的信号量, 下面与非公平锁逻辑相同
8         int available = getState();
9         int remaining = available - acquires;
10        if (remaining < 0 ||
11            compareAndSetState(available, remaining))
12            return remaining;
13    }
14 }

```

## 总结

不难看出, 公平锁与非公平锁的区别在于当线程尝试获取Semaphore中的信号量时:

- 公平锁，优先判断等待队列中是否有挂起的线程，如果有，则将当前线程添加到等待队列中，等待唤醒后抢夺信号量；
- 非公平锁，不管等待队列中是否有挂起线程，优先尝试获取信号量，获取失败，将当前线程添加到等待队列。

## acquire()

Semaphore默认实现的是非公平锁，acquire()按非公平锁的实现进行源码分析。

Semaphore 中获取一个信号量，Semaphore#acquire() 详情如下：

```
1 // Semaphore 中无信号量，阻塞
2 public void acquire() throws InterruptedException {
3     // 获取 Semaphore 信号量
4     sync.acquireSharedInterruptibly(1);
5 }
```

AbstractQueuedSynchronizer#acquireSharedInterruptibly() 详情如下：

```
1 public final void acquireSharedInterruptibly(int arg)
2     throws InterruptedException {
3     // 线程中断，抛出异常
4     if (Thread.interrupted())
5         throw new InterruptedException();
6     // 尝试获取Semaphore的信号量
7     if (tryAcquireShared(arg) < 0)
8         // 尝试获取信号量失败，再次获取Semaphore信号量
9         doAcquireSharedInterruptibly(arg);
10 }
```

```

1 private void doAcquireSharedInterruptibly(int arg)
2     throws InterruptedException {
3     final Node node = addWaiter(Node.SHARED);
4     boolean failed = true;
5     try {
6         // 自旋
7         for (;;) {
8             final Node p = node.predecessor();
9             // 当前节点的前驱节点为等待队列头节点
10            if (p == head) {
11                // 尝试获取信号量
12                int r = tryAcquireShared(arg);
13                // 获取信号量成功
14                if (r >= 0) {
15                    // 唤醒等待队列中的待唤醒线程
16                    setHeadAndPropagate(node, r);
17                    p.next = null;
18                    failed = false;
19                    return;
20                }
21            }
22            // 获取信号量失败，挂起线程 ==> 线程阻塞，待唤醒进行下一轮自旋
23            if (shouldParkAfterFailedAcquire(p, node) &&
24                // 若当前线程被中断，抛出InterruptedException异常
25                parkAndCheckInterrupt())
26                throw new InterruptedException();
27        }
28    } finally {
29        if (failed)
30            cancelAcquire(node);
31    }
32 }

```

AbstractQueuedSynchronizer#setHeadAndPropagate()

```

1 // node: 当前节点; propagate 剩余资源
2 private void setHeadAndPropagate(Node node, int propagate) {
3     // 获取等待队列中的头节点
4     Node h = head;
5     // 将当前Node节点设置为等待队列的头节点
6     setHead(node);
7     // 剩余资源大于0 || 原等待队列中的头节点为null || 原等待队列中 Node 的 ws 为 -1 或者
    -3(共享锁)
8     if (propagate > 0 || h == null || h.waitStatus < 0 || (h = head) == null ||
    h.waitStatus < 0) {
9         // 获取当前等待队列头节点的后继节点
10        Node s = node.next;
11        // 当前节点的后继节点为null 或 当前节点的后继节点为共享锁
12        if (s == null || s.isShared())
13            doReleaseShared();
14    }
15 }

```

## release()

Semaphore默认实现的是非公平锁，release()按非公平锁的实现进行源码分析。

归还Semaphore的信号量，Semaphore#release() 详情如下：

```

1 // 归还Semaphore的信号量
2 public void release() {
3     sync.releaseShared(1);
4 }

```

归还信号量，Semaphore#Sync#releaseShared() 详情如下：

```

1 public final boolean releaseShared(int arg) {
2     // 尝试归还信号量
3     if (tryReleaseShared(arg)) {
4         // 归还信号量
5         doReleaseShared();
6         // 归还成功
7         return true;
8     }
9     // 归还失败
10    return false;
11 }

```

归还信号量，Semaphore#Sync#releaseShared() 详情如下：

```

1 // 尝试归还信号量
2 protected final boolean tryReleaseShared(int releases) {
3     // 自旋
4     for (;;) {
5         // 获取Semaphore中可用的信号量数
6         int current = getState();
7         // 当前可用信号量数 + 归还的信号量 releases
8         int next = current + releases;
9         // 超出了int的最大值，变成了负数
10        if (next < current)
11            throw new Error("Maximum permit count exceeded");
12        // cas操作，将信号量归还给Semaphore
13        if (compareAndSetState(current, next))
14            return true;
15    }
16 }

```

归还信号量成功，唤醒等待队列中的挂起线程，AbstractQueuedSynchronizer#doReleaseShared()：

```
1 private void doReleaseShared() {
2     // 自旋
3     for (;;) {
4         // 获取等待队列头节点
5         Node h = head;
6         // 等待队列中有排队的线程
7         if (h != null && h != tail) {
8             int ws = h.waitStatus;
9             // 等待队列头节点ws = -1, 说明其后继节点中有待唤醒的线程
10            if (ws == Node.SIGNAL) {
11                // cas 操作, 等待队列头节点的 ws 由 -1 更新为 0 , cas失败, 继续下一次自旋
12                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
13                    continue;
14                // 唤醒头节点的后继节点中待唤醒线程
15                unparkSuccessor(h);
16            }
17            // 解决共享锁JDK1.5的bug, 头节点的 ws 为0, 将头节点的 ws 设置为 -3 , 代表后继节
            // 点中可能有待唤醒的线程
18            else if (ws == 0 &&
19                    !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
20                continue;
21        }
22        if (h == head)
23            break;
24    }
25 }
```