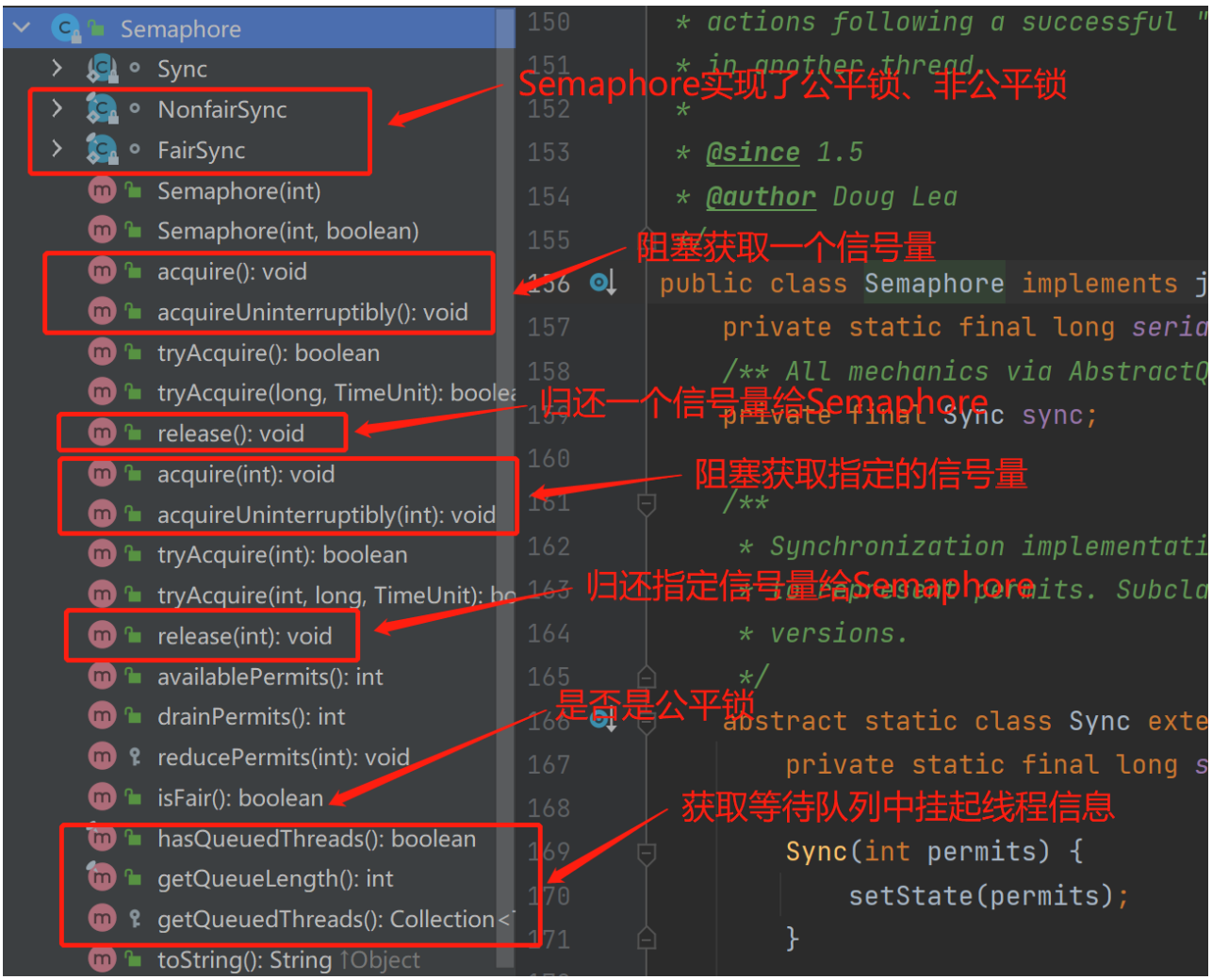
主讲老师：Fox

> 【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：



构造函数

```
// AQS的实现
private final Sync sync;

// 默认使用非公平锁
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

// 根据fair布尔值选择使用公平锁还是非公平锁
public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

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

```
// 非公平锁  获取信号量
protected int tryAcquireShared(int acquires) {
     return nonfairTryAcquireShared(acquires);
}
```

Semaphore#Sync#nonfairTryAcquireShared() 详情如下

```
// 非公平锁 获取信号量
final int nonfairTryAcquireShared(int acquires) {
    // 自旋
    for (;;) {
        // 获取Semaphore中可用的信号量数
        int available = getState();
        // 当前可用信号量数 - acquires
        int remaining = available - acquires;
        // 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
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

FairSync

Semaphore#FairSync#tryAcquireShared() 详情如下

```
protected int tryAcquireShared(int acquires) {
   // 自旋
    for (;;) {
        // 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)
        if (hasQueuedPredecessors())
            return -1;
        // 尝试获取Semaphore的信号量，下面与非公平锁逻辑相同
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

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：

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

AbstractQueuedSynchronizer#acquireSharedInterruptibly() 详情如下：

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

```java
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        // 自旋
        for (;;) {
            final Node p = node.predecessor();
             // 当前节点的前驱节点为等待队列头节点
            if (p == head) {
                // 尝试获取信号量
                int r = tryAcquireShared(arg);
                // 获取信号量成功
                if (r >= 0) {
                    // 唤醒等待队列中的待唤醒线程
                    setHeadAndPropagate(node, r);
                    p.next = null;
                    failed = false;
                    return;
                }
            }
            // 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
            if (shouldParkAfterFailedAcquire(p, node) &&
                // 若当前线程被中断，抛出InterruptedException异常
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

AbstractQueuedSynchronizer#setHeadAndPropagate()

```java
// node:  当前节点；propagate 剩余资源
private void setHeadAndPropagate(Node node, int propagate) {
    // 获取等待队列中的头节点
    Node h = head;
    // 将当前Node节点设置为等待队列的头节点
    setHead(node);
    // 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)
    if (propagate > 0 || h == null || h.waitStatus < 0 ||  (h = head) == null || h.waitStatus < 0) {
        // 获取当前等待队列头节点的后继节点
        Node s = node.next;
        // 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
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

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：

```java
// 归还Semaphore的信号量
 public void release() {
     sync.releaseShared(1);
 }
```

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

```
public final boolean releaseShared(int arg) {
    // 尝试归还信号量
    if (tryReleaseShared(arg)) {
        // 归还信号量
        doReleaseShared();
        // 归还成功
        return true;
    }
    // 归还失败
    return false;
}
```

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

```
// 尝试归还信号量
protected final boolean tryReleaseShared(int releases) {
    // 自旋
    for (;;) {
        // 获取Semaphore中可用的信号量数
        int current = getState();
        // 当前可用信号量数 + 归还的信号量 releases
        int next = current + releases;
        // 超出了int的最大值，变成了负数
        if (next < current)
            throw new Error("Maximum permit count exceeded");
        // cas操作，将信号量归还给Semaphore
        if (compareAndSetState(current, next))
            return true;
    }
}
```

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

```java
private void doReleaseShared() {
    // 自旋
    for (;;) {
        // 获取等待队列头节点
        Node h = head;
        // 等待队列中有排队的线程
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 等待队列头节点ws = -1，说明其后继节点中有待唤醒的线程
            if (ws == Node.SIGNAL) {
                // 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
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;
                // 唤醒头节点的后继节点中待唤醒线程
                unparkSuccessor(h);
            }
            // 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
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;
        }
        if (h == head)
            break;
    }
}
```