

第六章 ReentrantLock源码解析2--释放锁unlock()

最常用的方式：



```
int a = 12;
//注意：通常情况下，这个会设置成一个类变量，比如说Segment中的段锁与copyOnWriteArrayList中的全局锁
final ReentrantLock lock = new ReentrantLock();

lock.lock();//获取锁
try {
    a++;//业务逻辑
} catch (Exception e) {
}finally{
    lock.unlock();//释放锁
}
```



注：关于lock()方法的源码解析，请参照 第五章 ReentrantLock源码解析1 --获得非公平锁与公平锁lock()"


释放锁：unlock()


步骤：

- 1 ) 获取当前的锁数量，然后用这个锁数量减去解锁的数量（这里为1），最后得出结果c
- 2 ) 判断当前线程是不是独占锁的线程，如果不是，抛出异常
- 3 ) 如果c==0，说明锁被成功释放，将当前的独占线程置为null，锁数量置为0，返回true
- 4 ) 如果c!=0，说明释放锁失败，锁数量置为c，返回false
- 5 ) 如果锁被释放成功的话，唤醒距离头节点最近的一个非取消的节点


源代码：

ReentrantLock：unlock()







```
/**
 * 释放这个锁
 *1 ) 如果当前线程持有这个锁，则锁数量被递减
 *2 ) 如果递减之后锁数量为0，则锁被释放。
 *如果当前线程不持久有这个锁，抛出异常
 */
public void unlock() {
    sync.release(1);
}
```



AbstractQueuedSynchronizer：release(int arg)





```
/**
 * 释放锁（在独占模式下）
 */
public final boolean release(int arg) {
    if (tryRelease(arg)) { //如果成功释放锁
        Node h = head; //获取头节点：（注意：这里的头节点就是当前正在释放锁的节点）
        if (h != null && h.waitStatus != 0) //头结点存在且等待状态不是取消
            unparkSuccessor(h); //唤醒距离头节点最近的一个非取消的节点
        return true;
    }
}
```

```
        return false;
    }
}
```



Sync : tryRelease(int releases)



```
/**
 * 释放锁
 */
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;//获取现在的锁数量-传入的解锁数量（这里为1）
    if (Thread.currentThread() != getExclusiveOwnerThread())//当前线程不持有锁
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) { //锁被释放
        free = true;
        setExclusiveOwnerThread(null);
    } //如果不为0，怎么办，不释放了吗？
    setState(c);
    return free;
}
```



AbstractQueuedSynchronizer : unparkSuccessor(Node node)



```
/**
 * 唤醒离头节点node最近的一个非取消的节点
 * @param node 头节点
 */
private void unparkSuccessor(Node node) {

    int ws = node.waitStatus;
    if (ws < 0) //将ws设为0状态（即什么状态都不是）
        compareAndSetWaitStatus(node, ws, 0);

    /*
     * 获取头节点的下一个等待状态不是cancel的节点
     */
    Node s = node.next; //头节点的下一个节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        /*
         * 注意：从后往前遍历找到离头节点最近的一个非取消的节点，从后往前遍历据说是在入队（enq()）的时候，可能
         nodeX.next==null，但是在读源码的时候没看出来
         */
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread); //唤醒离头节点最近的一个非取消的节点
}
```



注意：

在程序的注释部分有一些疑问，整理成下边这样：

- 如果按照开头的那个程序的话，成功的获取一个锁之后，就会在finally里边解一次锁，可重入性怎么体现？
- 在找到离头节点最近的一个非取消的节点，是以从后往前的方式进行的，原因是"从后往前遍历据说是在入队（ enq() ）的时候，可能 nodeX.next==null"，但是在读源码的时候没看出来

第一个问题答案：

可重入性体现在下边这个程序（就是锁套锁，最常见的就是在递归中）：

```
final ReentrantLock lock = new ReentrantLock();
public void add() {
    lock.lock(); //获取锁
    try {
        add(); //业务逻辑
    } catch (Exception e) {
    } finally {
        lock.unlock(); //释放锁
    }
}
```

注意：

- 上边这个程序只是一个示例，在递归的使用中，一定要有递归结束的条件
- 每有一个lock()方法，就有一个unlock()与之对应，所以在解锁的时候，只需要把传递解锁数量为1就可以。

第二个问题答案：

记住：如果顺着节点头一直next下去可能会不正确。

举个例子：A 1 -> A 2 -> A 3

现在我们从A 1开始往下走，当我们走到A 3的时候，就在这时，一个新节点A 4入队，会走下面的入队代码：

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            Node h = new Node(); // Dummy header
            h.next = node;
            node.prev = h;
            if (compareAndSetHead(h)) {
                tail = node;
                return h;
            }
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) { 1
                t.next = node; 2
                return t;
            }
        }
    }
}
```

可以看到步骤1 和步骤2 整体并不是原子的，也就是说，当执行完C A S的时候但是2 还没执行，这时候队列为：A 1 --> A 2 --> A 4，如果你使用next的话，可能就把A 3 给没了，但是node.prev = t（即A 4.prev = A 3），也就是说前驱节点是已经赋过值的了，如果你从队列结尾A 4.prev就会是A 3，即A 3也丢不了。