

JUC

1 java.util.concurrent

Callable

ArrayBlockingQueue

ConcurrentHashMap

CopyOnWriteArrayList

CountDownLatch

CyclicBarrier

Exchanger

Executors

FutureTask

LinkedBlockingQueue

Semaphore

ThreadPoolExecutor

2 java.util.concurrent.locks

AbstractQueuedSynchronizer

Condition

Lock

LockSupport

ReentrantLock

ReadWriteLock

ReentrantReadWriteLock

3 java.util.concurrent.atomic

AtomicInteger

AtomicReference

4 synchronized关键字 (底层也是cas+park实现的)

1 用户态和内核态

2 cas

cas会引起aba问题，解决方案:AtomicStampedReference

cas怎么解决原子性问题：lock

3 对象的内存布局

4 锁升级

5 volatile关键字

6 aqs

基于ch1队列的state共享变量的锁模式，通过模版方法模式实现

AQS实现的排它锁：ReentrantLock

AQS实现的共享锁: ReadWriteLock\Semaphore\CountDownLatch\CyclicBarrier

Condition接口的主要实现类是AQS的内部类 `ConditionObject`，每个Condition对象都包含一个等待队列。该队列是Condition对象实现等待/通知的关键。AQS中同步队列与等待队列的关系如下：

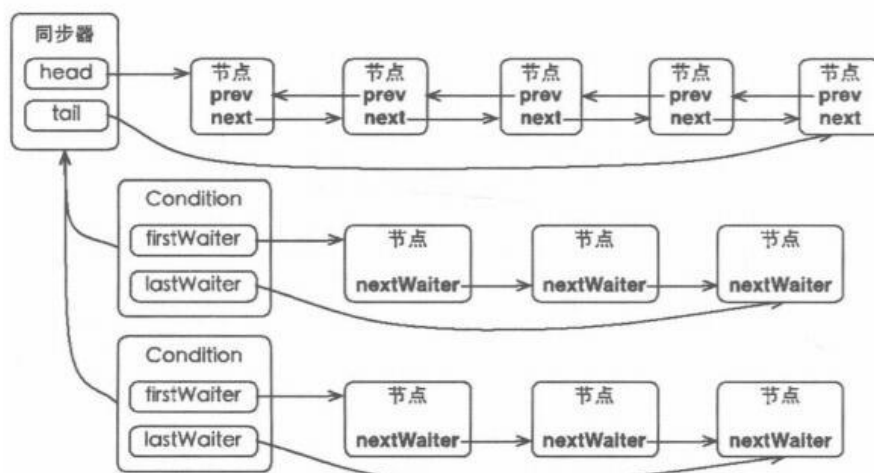


图 5-10 同步队列与等待队列 https://blog.csdn.net/gs_albb

在Object的监视器模型上，一个对象拥有一个同步队列与一个等待队列，而AQS拥有一个同步队列和多个等待队列。

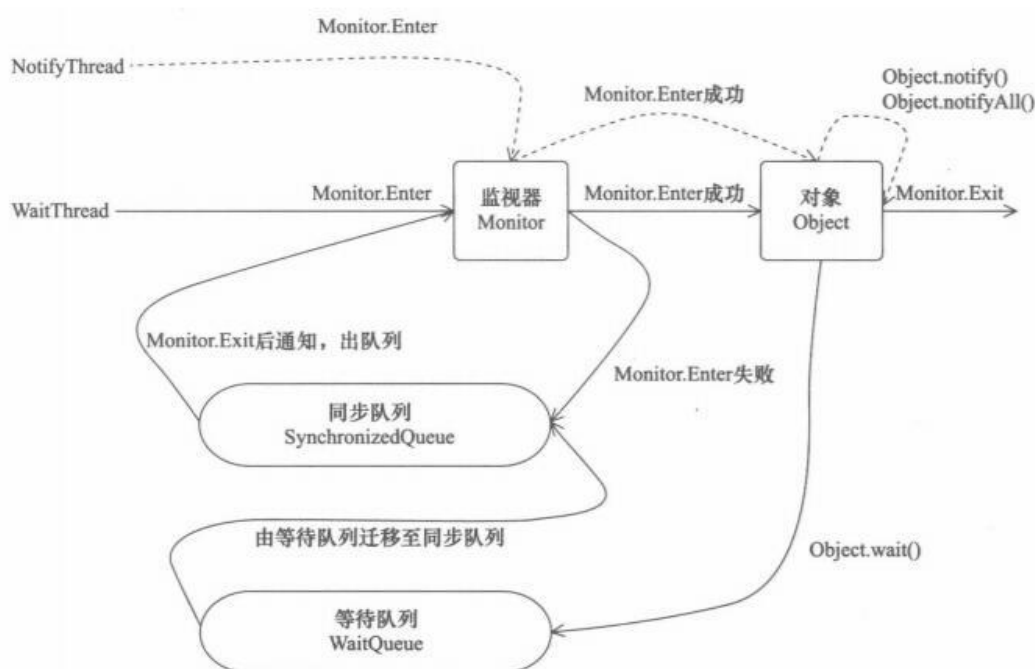


图 4-3 WaitNotify.java 运行过程 https://blog.csdn.net/gs_albb

调用condition的await方法，将会使当前线程进入等待队列并释放锁(先加入等待队列再释放锁)，同时线程状态转为等待状态。

从同步队列和阻塞队列的角度看，调用await方法时，相当于**同步队列的首节点移到condition的等待队列中**

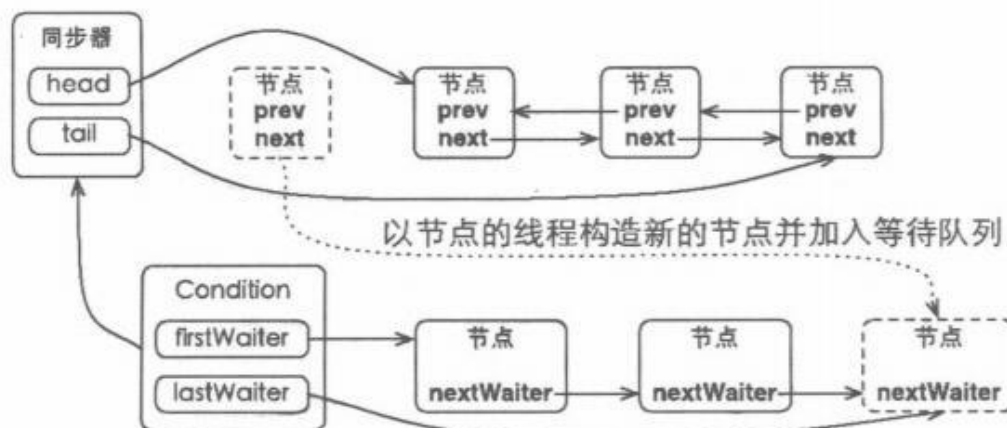


图 5-11 当前线程加入等待队列 https://blog.csdn.net/gs_albb

调用condition的signal方法时，将会把等待队列的首节点移到等待队列的尾部，然后唤醒该节点。被唤醒，并不代表就会从await方法返回，也不代表该节点的线程能获取到锁，它一样需要加入到锁的竞争acquireQueued方法中去，只有成功竞争到锁，才能从await方法返回。

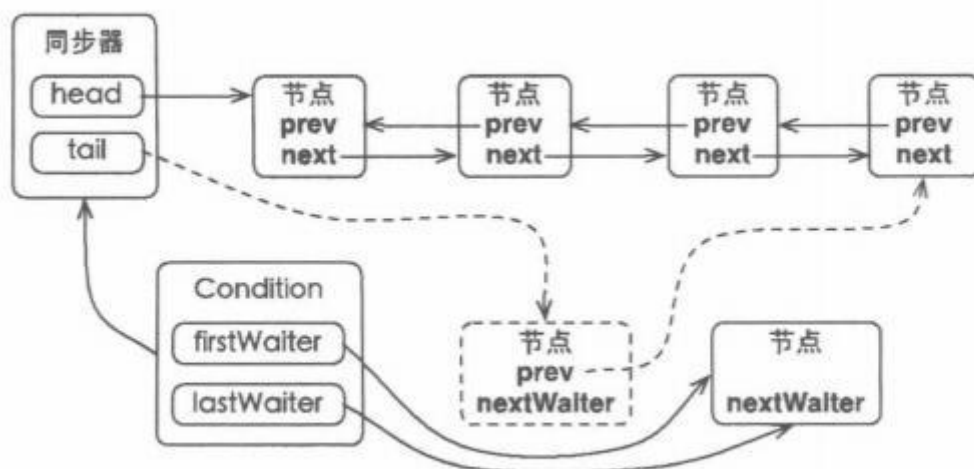
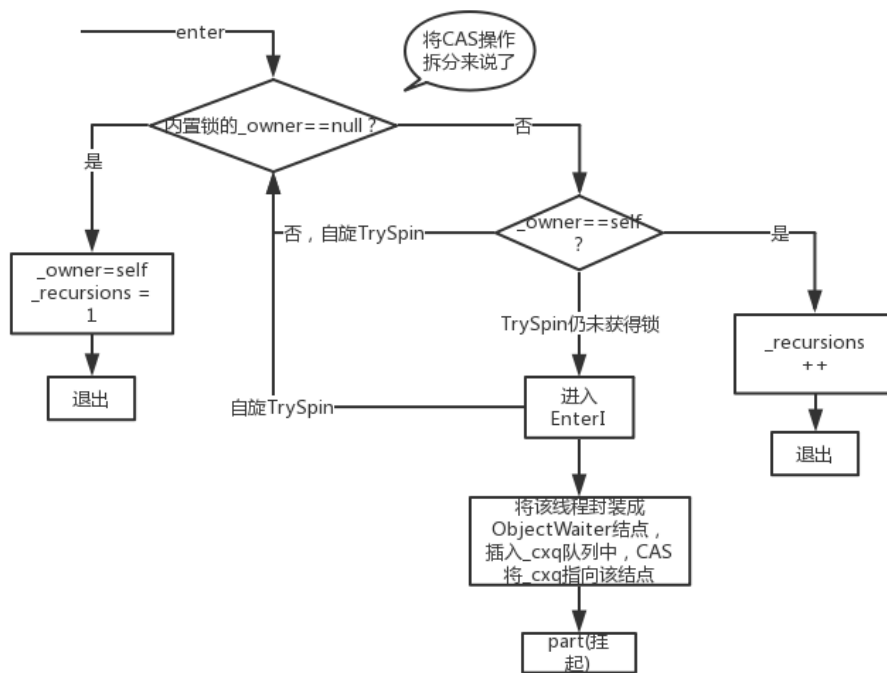


图 5-12 节点从等待队列移动到同步队列 blog.csdn.net/gs_albb

SYNC重量级锁的获取过程



讲解SYNC

<https://blog.csdn.net/zwjy1203/article/details/106217887>

讲解aqs

<https://segmentfault.com/a/1190000016058789>

AQS非公平锁底层原理代码分析：

调用端代码样例

```

Lock lock = new ReentrantLock();
lock.lock();
try {
    //do buss
} finally {
    lock.unlock();
}
  
```

```

//构造方法中指定了是非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}
  
```

```
//lock.lock调用非公平锁的lock方法
final void lock() {
    //并没有检查当前队列是否有排队情况，直接尝试获取锁，此处显示出非公平锁，此处并没有采用重试，
    失败之后尝试acquire入队列
    if (compareAndSetState(0, 1))
        //如果CAS把state变成1，表示获取锁成功，把当前线程设置成独占线程
        setExclusiveOwnerThread(Thread.currentThread());
    else
        //调用AQS的acquire方法
        acquire(1);
}
```

```
//获取锁的逻辑-主要包含三个方法
//tryAcquire-aqs子类实现的方法
//addwaiter-把当前线程封装成Node对象放入队列，注意此处的Node类型是Node.EXCLUSIVE
//acquireQueued-更新前节点的状态
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        //如果是被别的线程中断唤醒，获取到锁之后，会自己打断
        selfInterrupt();
}
```

```
//非公平锁中tryAcquire方法的具体实现
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //如果当前锁没有被占用，通过CAS操作尝试获取锁，此处也体现出非公平锁，此处并没有采用重试，
        失败之后返回false
        if (compareAndSetState(0, acquires)) {
            //设置排他线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //线程重入的情况
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //非上述两种情况，返回false，后续进入CLH队列
    return false;
}
```

```
//把Node节点放入队列
private Node addwaiter(Node mode) {
    //构建新Node
    /*Node(Thread thread, Node mode) {
        *      this.nextwaiter = mode;
        *      this.thread = thread;
        *}
    */
}
```

```

    */
    Node node = new Node(Thread.currentThread(), mode);
    // 尝试快速入队
    Node pred = tail;
    if (pred != null) {
        //当前线程的节点的prev值变成尾结点, 考虑如果有多个线程同时进入, 会有多个线程的节点的prev值会变成尾结点
        node.prev = pred;
        //如果tail节点不是空的, 即队列中有值, 采用CAS操作把AQS的尾节点变成当前线程的节点, 此处只会有一个线程会成功(思考尾分叉), 此处并没有采用重试, 失败之后采用别的方式进入队列
        if (compareAndSetTail(pred, node)) {
            // 把上一个尾节点的next值, 指向新的尾节点
            pred.next = node;
            return node;
        }
    }
    //如果快速入队失败, 尝试正常入队
    enq(node);
    return node;
}

```

```

//快速入队失败之后尝试正常入队
private Node enq(final Node node) {
    //采用死循环重试的方式
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            //如果尾节点是空, 表明是第一次进入队列, 则new Node()对象, 此对象没有Thread属性, 称为空节点。把空节点置成CLH队列的头节点
            //所以, 注意CLH队列的头节点要么是NULL, 要么是没有Thread属性的空节点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            //采用CAS的方式把当前节点设置成尾结点, 如果失败, 会一直重试, 直至设置成功
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}

```

```

// 修改Node节点状态
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        //采用死循环的方式一直重试
        for (;;) {
            //获取当前节点的前一个节点
            final Node p = node.predecessor();
            //如果当前节点的前一个节点是头节点, 则调用上面的tryAcquire方法, 继续尝试获取锁
            if (p == head && tryAcquire(arg)) {
                //如果获取锁成功, 则把当前节点设置成头节点, 并把当前节点的thread属性置null, 当前节点的prev属性置null
            }
        }
    }
}

```

```

        setHead(node);
        //把当前节点的前节点的next属性变成null, 为了方便GC回收
        p.next = null; // help GC
        failed = false;
        return interrupted;
    }
    //获取锁失败的处理
    if (shouldParkAfterFailedAcquire(p, node) &&
        parkAndCheckInterrupt())
        interrupted = true;
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
}

```

```

//设置当前节点为头节点, 并把当前节点的thread属性置null, 当前节点的prev属性置null
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}

```

```

// 如果获取锁失败, 进入park状态
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    //查看当前节点的前一个节点, 如果waitStatus == -1, 直接返回
    if (ws == Node.SIGNAL)
        return true;
    //查看当前节点的前一个节点, 如果waitStatus == 1, 则一直往前找, 直到找到waitStatus <= 0
    //的节点作为它的前节点
    if (ws > 0) {
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        //如果waitStatus <= 0, 独占锁只有waitStatus == 0的场景, 则通过CAS把前节点的
        //waitStatus变成-1
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

```

```

//如果设置了前节点waitStatus = -1之后, 当前节点的线程park
private final boolean parkAndCheckInterrupt() {
    //未获取到锁的线程会一直park在这个位置, 直到被上个节点的线程唤醒或者被别的线程中断
    LockSupport.park(this);
    //如果被唤醒, 则返回是否是中断被唤醒的标识, 如果返回true, 表明是被别的线程中断, 如果返回
    //false, 表明是被前节点唤醒
    return Thread.interrupted(); //此方法会消除中断标识
}

```

```
//如果是被别的线程中断之后获取到锁，并不是从前节点唤醒的，则线程自己中断
static void selfInterrupt() {
    Thread.currentThread().interrupt();
}
```

AQS公平锁底层原理代码分析：

```
//调用端代码
Lock lock = new ReentrantLock(true);
lock.lock();
try {
    //do buss
} finally {
    lock.unlock();
}
```

```
//构造其中指定使用公平锁
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

```
//公平锁的lock实现
//此处并没有采用CAS直接获取锁
final void lock() {
    acquire(1);
}
```

```
//此处的获取锁的主要逻辑和非公平锁的逻辑一致，只是tryAcquire方法是公平锁的实现
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

```
//公平锁的尝试获取锁的逻辑
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //如果state==0&&队列中没有正在等待的线程，则尝试CAS获取锁
        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;
    }
    //如果state!=0或者CAS获取锁失败则当前线程进入队列
    return false;
}

```

```

public final boolean hasQueuedPredecessors() {
    Node t = tail;
    Node h = head;
    Node s;
    //如果h=null t=null表明队列中没有等待的线程，则返回false，整个Node进入队列中
    //如果h非空，t非空但是t==h表明队列中没有等待的线程，则返回false，整个Node进入队列中
    //如果h非空 t是空 s = null表明 整个队里刚开始是空的，第一个线程进入队列时，刚把head空节点设置好，并未设置tail节点
    //如果h非空 t非空，s != null表明 整个队列中至少有一个值，如果s的线程是当前线程，则返回false，s是第一个等待的线程，尝试获取锁
    //如果h非空 t非空 s != null 但是s的线程不是当前线程，返回true，尝试进入队列
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}

```

公平锁和非公平锁的解锁流程都是一样的

```

//此处的release方法调用的是AQS中的方法，所以，公平锁和非公平锁都是一个流程
public void unlock() {
    sync.release(1);
}

```

```

//释放锁
public final boolean release(int arg) {
    //调用AQS实现类中的方法，此处调用的是ReentrantLock内部类Sync中的方法，公平锁和非公平锁都一样
    if (tryRelease(arg)) {
        Node h = head;
        //释放锁成功之后，如果头节点不是空&&头节点状态!=0，表明后续由需要头节点唤醒的线程
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

```

```

//尝试释放锁
protected final boolean tryRelease(int releases) {
    //state值，每释放一次就减1
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    //如果state=0，表明该线程不再使用锁，需要把exclusiveOwnerThread变成null

```

```

//如果state!=0,表明该线程还需要继续使用锁,但是只是释放了一次而已
if (c == 0) {
    free = true;
    setExclusiveOwnerThread(null);
}
setState(c);
return free;
}

```

//如果上个线程完全释放锁,即state=0&&CLH队列中头节点非空&&头结点的waitStatus!=0则唤醒CLH队列中的下一个线程

```

private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    //如果头节点状态<0,则采用CAS方法把statue变成0
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    Node s = node.next;
    //如果头节点的下一个节点为空或者waitStatus >0,则从尾节点开始往前遍历,找出
    waitStatus<=0的最前面的那一个节点。
    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;
    }
    //如果找到了该节点,则唤醒该节点上的线程
    if (s != null)
        LockSupport.unpark(s.thread);
}

```

共享锁的获取源码分析

```

//使用端代码逻辑
Semaphore semaphore = new Semaphore(5);
semaphore.acquire();
semaphore.release();

```

```

//锁的构造方法
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

```

```

//非公平锁的实现
NonfairSync(int permits) {
    super(permits);
}

```

```

//Semaphore内部类Sync的构造方法
Sync(int permits) {
    setState(permits);
}

```

```

}

//设置AQS的state属性的值
protected final void setState(int newState) {
    state = newState;
}

```

```

//调用获取共享锁的方法
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}

```

```

//获取可打断的共享锁
public final void acquireSharedInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //如果>=0表明获取锁成功，不需要等待队列，如果方法返回值<0表示线程获取锁失败，则需要进入CLH
    队列等待锁
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}

```

```

//Semaphore内部NonfairSync非公平锁的实现
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}

```

```

//Semaphore内部Sync的方法
final int nonfairTryAcquireShared(int acquires) {
    //此处是一个死循环
    for (;;) {
        //获取当前的state值
        int available = getState();
        int remaining = available - acquires;
        //如果剩余的state的值<0,则直接返回负值
        //如果大于或者等于0，尝试CAS更新状态--此处会一直重试，直至成功或者直接返回负值
        //返回负值就会进入CLH队列
        if (remaining < 0 ||
            !compareAndSetState(available, remaining))
            return remaining;
    }
}

```

```

//线程进入队列&更新waitStatus状态，对比独占锁的acquireQueued
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    //此处的逻辑跟独占锁的一直，同一个方法
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        //此处是一个死循环，会保证进入队列一定成功
        for (;;) {
            final Node p = node.predecessor();

```

```

        //如果当前节点的前一个节点是head节点
        if (p == head) {
            //再次尝试获取锁
            int r = tryAcquireShared(arg);
            //如果获取锁成功
            if (r >= 0) {
                //设置头节点
                setHeadAndPropagate(node, r);
                p.next = null; // help GC
                failed = false;
                return;
            }
        }
        if (shouldParkAfterFailedAcquire(p, node) &&
            parkAndCheckInterrupt())
            throw new InterruptedException();
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
}

```

```

//设置头节点&释放共享锁
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    //设置头节点
    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();
    }
}

```

```

//设置头节点，并把thread值为空，并且prev的值为空
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}

```

共享锁的释放源码分析

```

//释放共享锁逻辑
public void release() {
    sync.releaseShared(1);
}

```

```
//AQS中的释放共享锁逻辑
public final boolean releaseShared(int arg) {
    //尝试释放锁，释放成功返回true，否则返回false
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

```
//Semaphore中Sync中的方法，采用CAS+自旋的方式，一直尝试释放锁，直至成功，每次释放state值就+1
protected final boolean tryReleaseShared(int releases) {
    //死循环
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            return true;
    }
}
```

```
//AQS中的方法，同样是采用CAS+自旋的方式
private void doReleaseShared() {
    //死循环
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                //如果头节点状态是SIGNAL，则使用CAS把头节点状态变成0，此处只有一个线程可以完成此操作
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue; // loop to recheck cases
                //完成变更的线程，可以唤醒后续的线程，没有变更成功的线程则继续循环
                unparkSuccessor(h);
            }
            //PROPAGATE为了解决线程无法唤醒的BUG
            else if (ws == 0 &&
                !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue; // loop on failed CAS
        }
        //如果头结点没有发生过变化，则可以退出循环，这个是循环退出的唯一方式，否自会一直继续，形成调用风暴，增加后续节点唤醒的速度
        if (h == head) // loop if head changed
            break;
    }
}
```

Condition接口源码解析

```
//Condition是个接口
//ConditionObject是AQS的内部类,ConditionObject实现了Condition接口
//ConditionObject中有firstwaiter\lastwaiter两个属性
//condition必须和lock配合使用
//Lock接口有个方法newCondition, 返回Condition对象
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();
condition.await();
condition.signal();
condition.signalAll();
```

```
//调用的是ReentrantLock内部类Sync中的newCondition方法
public Condition newCondition() {
    return sync.newCondition();
}
```

```
//在该方法中直接new ConditionObject
final ConditionObject newCondition() {
    return new ConditionObject();
}
```

await方法源码解析

```
//AQS中ConditionObject内部类中的方法
//能调用此方法的必定是持有锁的线程--此处的锁必须是实现了Lock的锁
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //添加一个新的Node节点到等待队列中(注意, 此处并不是把同步队列中的Node节点加入等待队列, 而是创建了一个新的Node)
    Node node = addConditionWaiter();
    //释放锁, 此处的释放锁, 会释放该线程所有的锁, (即, 如果一个线程加了两次锁, state=2, 则此处直接state=0) 并返回持有的锁的个数
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    //查看当前节点是否在同步队列中
    while (!isOnSyncQueue(node)) {
        //当前线程挂起
        LockSupport.park(this);
        //-----
        //以下是线程被唤醒之后的逻辑--包括了被中断和被signal两种唤醒方式
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

```

//新建Node节点并加入到等待队列中
private Node addConditionWaiter() {
    Node t = lastWaiter;
    //如果Node的waitStatus非CONDITION, 则从firstwaiter开始, 往后遍历找出最后一个状态是
    //CONDITION节点作为尾部节点
    //等待队列中的状态有三种, CONDITION(初始状态), 0, 和CANCELLED
    //同步队列中的状态有三种, 0(初始状态), SIGNAL和取消, 以及PROPAGATE
    if (t != null && t.waitStatus != Node.CONDITION) {
        //从前往后遍历找出最后一个状态是CONDITION节点作为尾部节点
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    //创建一个新的节点(注意, 节点状态是CONDITION)
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    //如果头节点是空, 则当前节点为头节点
    if (t == null)
        firstWaiter = node;
    //否则, 把尾节点的nextwaiter指向当前节点
    else
        t.nextwaiter = node;
    //尾节点指向当前节点
    lastWaiter = node;
    return node;
}

```

```

//释放线程拥有的所有锁, 即state=0, 此处返回值表明的是该线程持有的锁的个数(重入的个数)
final int fullyRelease(Node node) {
    boolean failed = true;
    try {
        int savedState = getState();
        //排他锁释放线程方法一样
        if (release(savedState)) {
            failed = false;
            return savedState;
        } else {
            throw new IllegalMonitorStateException();
        }
    } finally {
        if (failed)
            node.waitStatus = Node.CANCELLED;
    }
}

```

```

//判定Node是否在同步队列中
final boolean isOnSyncQueue(Node node) {
    //在等待队列中
    if (node.waitStatus == Node.CONDITION || node.prev == null)
        return false;
    //在同步队列中
    if (node.next != null) // If has successor, it must be on queue
        return true;
    //从尾节点开始查询Node是否在同步队列中
    return findNodeFromTail(node);
}

```

```

//从尾节点开始查询Node是否在同步队列中
private boolean findNodeFromTail(Node node) {
    Node t = tail;
    for (;;) {
        if (t == node)
            return true;
        if (t == null)
            return false;
        t = t.prev;
    }
}

```

signal方法源码解析

```

//AQS中ConditionObject内部类中的方法
//能调用此方法的必定是持有锁的线程--此处的锁必须是实现了Lock的锁
public final void signal() {
    //检查当前线程是否持有锁
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        //通知第一个waiter
        doSignal(first);
}

```

```

//通知第一个waiter
private void doSignal(Node first) {
    do {
        //如果头节点的下一个节点是空，则把等待队列置空
        if ((firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
    } while (!transferForSignal(first) &&
        (first = firstWaiter) != null);
}

```

```

final boolean transferForSignal(Node node) {
    /*
     * If cannot change waitStatus, the node has been cancelled.
     */
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    /*
     * 加入到同步队列中，同获取锁中的enq方法是同一个方法，返回值是当前节点的前一个节点
     */
    Node p = enq(node);
    int ws = p.waitStatus;
    //如果前一个节点取消或者更改状态失败，则唤醒首节点
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        LockSupport.unpark(node.thread);
    //返回true，则退出循环，表明唤醒节点
}

```



```
        return true;
    }
```

await方法被唤醒之后的逻辑

```
private int checkInterruptWhileWaiting(Node node) {
    return Thread.interrupted() ?
        (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
        0;
}
```

```
final boolean transferAfterCancelledWait(Node node) {
    if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
        enq(node);
        return true;
    }
    /*
     * If we lost out to a signal(), then we can't proceed
     * until it finishes its enq().  Cancelling during an
     * incomplete transfer is both rare and transient, so just
     * spin.
     */
    while (!isOnSyncQueue(node))
        Thread.yield();
    return false;
}
```

```
private void reportInterruptAfterWait(int interruptMode)
    throws InterruptedException {
    if (interruptMode == THROW_IE)
        throw new InterruptedException();
    else if (interruptMode == REINTERRUPT)
        selfInterrupt();
}
```