Condition等待队列

1.Condition 简介

任何一个java对象都天然继承于Object类，在线程间通信的往往会应用到Object的几个方法，比如wait(),wait(long timeout),wait(long timeout, int nanos)与notify(),notifyAll()几个方法实现等待/通知机制，同样的，在java Lock（）体系下依然会由同样的方法实现等待/通信机制，从整体上来讲**Object的wait和notify/notifyAll是与对象监视器配合完成线程间的通信，而Condition与Lock配合完成通知机制，前者是java底层级别的，后者是语言级别的，具有更高的可控制性和扩展性。两者除了在使用方法上不同外，在功能特性上还是有很多的不同：**

1.Condition能狗支持不响应中断，而通过使用Object方法不支持；
2.Condition能够支持多个等待队列（new多个Condition对象），而Object方式只能支持一个；
3.Condition能够支持设置截至时间，而Object不支持，Obejct支支持超时时间

2.等待方法await( )

Condition await( ) 相当于Object的wait( ) 方法

1.`I.void await() throws InterruptedException;`当线程进入等待状态，如果**其他线程调用condition的signalAll方法并且当前线程获取到Lock从await方法返回，如果在等待状态中被中断会抛出被中断异常；**

2`long awaitNanos(long nanosTimeout) throws InterruptedException;`：**当前线程进入等待状态直到被通知，中断或者超时**

3.`void awaitUninterruptibly()`：特性1：等待过程中不响应中断

4. **`boolean await(long time, TimUnit unit) throws InterruptedException:` 同第二种，支持自定义时间单位**

5.**`boolean awaitUntil(Date deadline) throws InterruptedException`：当前线程进入等待状态直到被通知，中断或者到了 某个时间**

3.唤醒方法 signal() signalAll（）

Condition signal（） signalAll()

1. void signal()：**唤醒一个等待在condition上的线程，将该线程从等待队列中转移到同步队列中，如果在同步队列中 能够竞争到Lock则可以从等待方法中返回。**
2. void signalAll()：**与1的区别在于能够唤醒所有等待在condition上的线程.**

4.Condition的实现

4.1Condition 等待队列的产生+队列的数据结构

```
Lock lock = new ReetrantLock();
//与锁绑定的Condition对象 -即等待队列
Condition condition = lock.newCondition();
```

4.1.1等待队列产生总结

创建一个 condition对象是通过lock.newCondition(),而这个方法实际上是会new出一个ConditionObject对象，该类是AQS的一 个内部类。前面我们说过condition是要和lock配合使用的也就是condition和Lock是绑定在一起的，而lock的实现原 理又依赖于AQS。

AQS内部维护一个同步队列，如果是独占锁的话，所有获取失败的线程尾插如**同步队列**，同样的，condition内部也是使用同样的方式，内部维护了一个**等待队列**，所有调用**condition.await()方法的线程就会加入到等待队列中**去，并且**线程状态转换为等待状态。**另外注意到ConditionObject中有两个成员变量：

```
/** First node of condition queue. */
private transient Node firstWaiter;
/** Last node of condition queue. */
private transient Node lastWaiter;
```

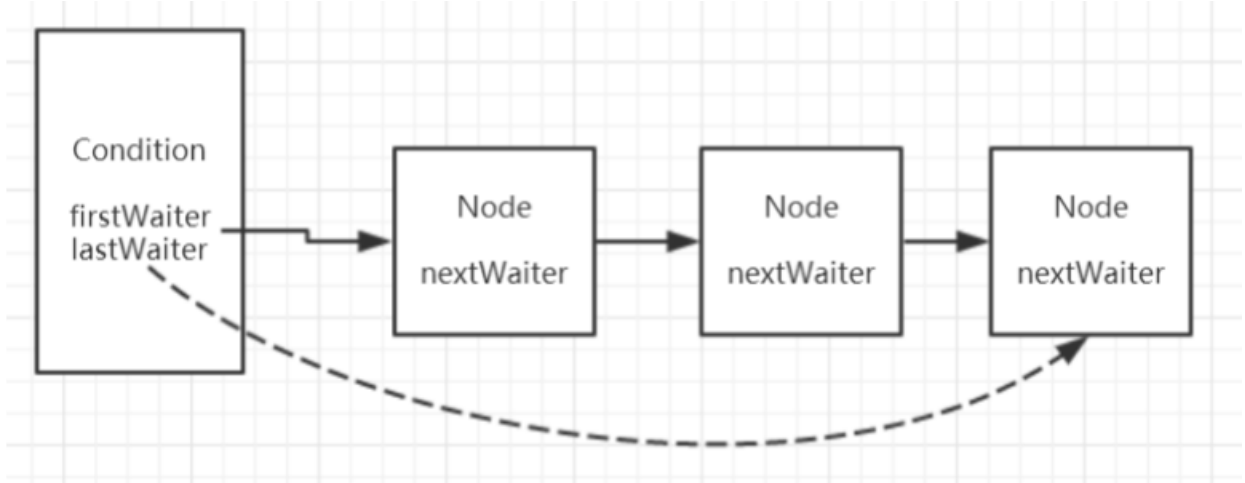
ConditionObject通过持有等待队列的头尾指针来管理等待队列。主要注意的是Node类复用了 在AQS中的Node类，Node类有这样一个属性：

```
//后继节点
Node nextWaiter;
```

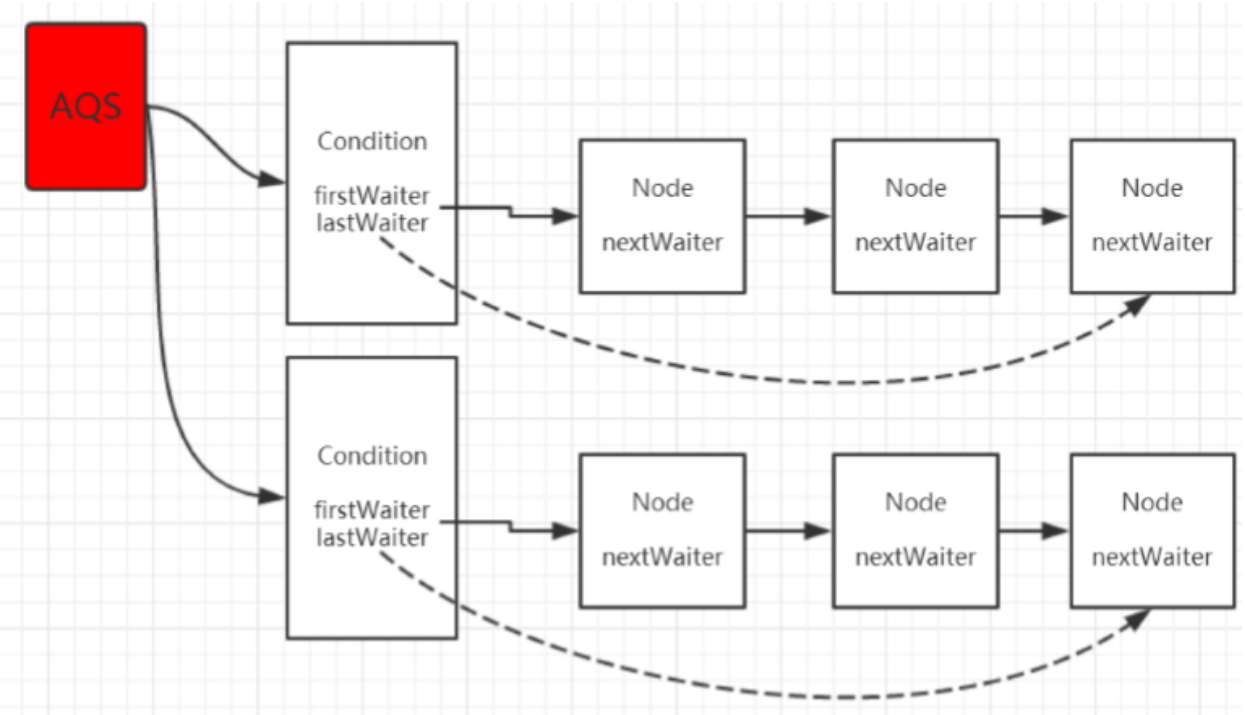
进一步说明，等待队列是一个单向队列，而在之前说AQS时知道同步队列是一个双向队列。

4.1.2结构总结：

1.队列是一个拥有头尾指针的单向队列。（不带头节点）



2.且调用多次lock.newCondition()方法创建多个condition对象，也就式一个lock可以持有多个等待队列。。而在之前利用Object的方式实际上是指在**对象Object对象监视器上只能拥有一个同步队列和 一个等待队列，而并发包中的Lock拥有一个同步队列和多个等待队列。**示意图如下：



如图所示，ConditionObject是AQS的内部类，因此每一个ConditionObject能狗方法到AQS提供的方法，相当于每个Condition都拥有同步器的引用。

4.2Condition实现有界队列

学完类集补充

5.await()实现原理

当调用condition.awit()方法后会使当前获取Lock锁的线程进入到等待队列中，如果该线程能够从await（）方法中返回的话一定是该线程获取了与condition相关联的锁。

```
public final void await() throws InterruptedException {
 if (Thread.interrupted()) //该方法响应中断
 throw new InterruptedException();
 //1.将当前获得锁的线程包装成一个当前锁的等待队列的结点，并尾插到等待队列中去
 Node node = addConditionWaiter();
 // 2.释放当前线程所占用的lock，释放后会唤醒同步队列中的下一个节点
 int savedState = fullyRelease(node);
 int interruptMode = 0;
 //判断该节点是否在同步队列中
  while (!isOnSyncQueue(node)) {
 //当线程不在同步队列后将其，阻塞（在等待队列中），置为WAIT状态
  LockSupport.park(this);
  //假如
  if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
  break;
  }

  //被唤醒，进入同步队列竞争获取锁
  if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
  interruptMode = REINTERRUPT;
  if (node.nextWaiter != null) // clean up if cancelled
  unlinkCancelledWaiters();
  //处理中断状况
  if (interruptMode != 0)
  reportInterruptAfterWait(interruptMode);
 }
```

5.1addConditionWaiter():包装

将当前获得锁的线程包装成一个 当前锁的等待队列 的结点

```
private Node addConditionWaiter() {
 Node t = lastWaiter; //获得等待队列的最后一个节点
 // If lastWaiter is cancelled, clean out.
 if (t != null && t.waitStatus != Node.CONDITION) { //如果尾接点不为空，且该节点状态不是取消状态
 //清空所有等待队列中状态不为CONDITION的节点
 unlinkCancelledWaiters();
 t = lastWaiter;//确定清空完的队列中的最后一个节点
 }
```

```
 Node node = new Node(Thread.currentThread(), Node.CONDITION); //将当前线程包装为Node节点且状态为Condition
  if (t == null)
  firstWaiter = node;//初始化等待队列
  else
  t.nextWaiter = node;//如果等待队列不为空，尾插到等待队列中
  lastWaiter = node;
  return node; //返回该节点（尾插到等待队列中的节点）
 }
```

5.2unlinkCancelledWaiters()：清空状态不为CONDITION的节点

//单链表向后遍历

//清空所有等待队列中状态不为CONDITION的节点

```
private void unlinkCancelledWaiters() {
  Node t = firstWaiter; //等待队列的第一个节点
  Node trail = null;
  while (t != null) { //从第一个节点向后遍历
  Node next = t.nextWaiter; //该节点的下一个节点
  if (t.waitStatus != Node.CONDITION) { //如果该节点状态不是CONDITION状态
  t.nextWaiter = null;//该节点的下一个节点设置空
  if (trail == null)//如果尾节点为空
  firstWaiter = next; //第一个节点等于该节点的下一个节点
  else
  //该节点不为空，尾接点的下一个节点为该节点的下一个节点
  trail.nextWaiter = next;
  if (next == null) //如果该节点的下一个节点为空，则将最后一个节点设置为该节点
  lastWaiter = trail; //
  }
  else //如果该节点状态是CONDITION状态，将尾节点设置为该节点，该节点向后遍历。
  trail = t;
  t = next;
  }
 }
```

5.3 fullyRelease(Node node)释放锁

释放锁的过程，将线程包装为Node节点尾插到等待队列后，线程释放锁的过程.

```
final int fullyRelease(Node node) {
  boolean failed = true;
  try {
```

```
//获取当前同步状态
int savedState = getState();
//调用release释放当前同步状态，唤醒下一个在同步队列中阻塞（pake的）节点
if (release(savedState)) {
failed = false;
return savedState;//为0状态，release()方法将其中状态改为0
 } else {
 //释放同步状态失败抛出异常
 throw new IllegalMonitorStateException();
 }
 } finally {//最后将当前线程节点状态设置为取消
 if (failed)
 node.waitStatus = Node.CANCELLED;
 }
}
```

5.4 isOnSyncQueue(Node node)：判断该节点是否在同步队列中

```
final boolean isOnSyncQueue(Node node) {
 if (node.waitStatus == Node.CONDITION || node.prev == null)
 return false;
 if (node.next != null) // If has successor, it must be on queue
 return true;
 /*
 * node.prev can be non-null, but not yet on queue because
 * the CAS to place it on queue can fail. So we have to
 * traverse from tail to make sure it actually made it. It
 * will always be near the tail in calls to this method, and
 * unless the CAS failed (which is unlikely), it will be
 * there, so we hardly ever traverse much.
 */
 return findNodeFromTail(node);
}

//从后向前遍历，看该节点是否在同步队列中
private boolean findNodeFromTail(Node node) {
 Node t = tail;
 for (;;) {
 if (t == node)
 return true;
 if (t == null)
 return false;
```

```
 t = t.prev;
 }
}
```

5.5被唤醒后进入同步队列竞争获取锁源码分析

```

```

5.6 await()的退出时机

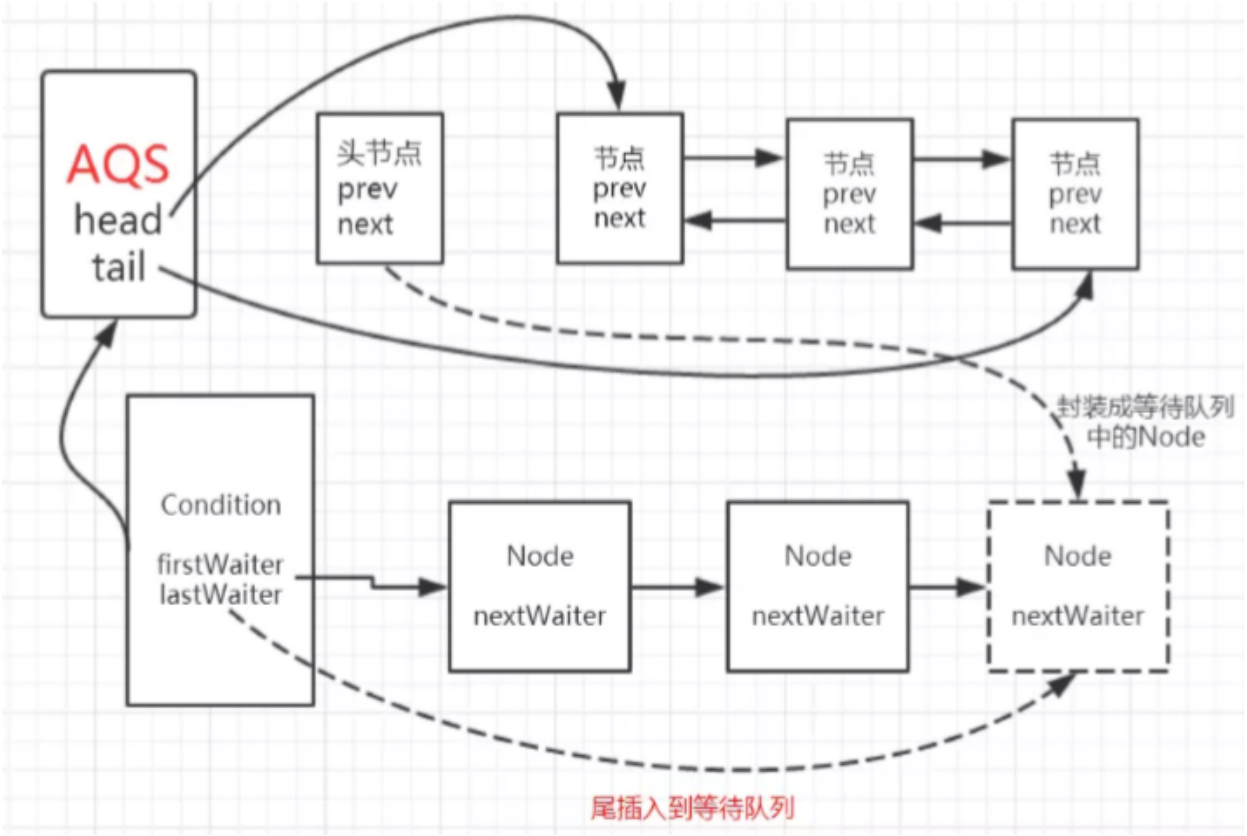
```
while (!isOnSyncQueue(node)) {
 //判断该节点是否在同步队列中
 LockSupport.park(this);
 if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
 break;
}
I.在等待时被中断，通过break退出循环
II.被唤醒后置入同步队列，退出循环。

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;
 }

/** Mode meaning to reinterrupt on exit from wait */
private static final int REINTERRUPT = 1;
/** Mode meaning to throw InterruptedException on exit from wait */
private static final int THROW_IE = -1;

```

结论：

> 很显然，当线程第一次调用condition.await()方法时，会进入到这个while()循环中，然后通过LockSupport.park(this)方 法使得当前线程进入等待状态，那么要想退出这个await方法第一个前提条件自然而然的是要先退出这个while循 环，出口就只剩下两个地方：
> 1. 逻辑走到break退出while循环；
> 2. while循环中的逻辑判断为false。
> 再看代码出现第 1种情况的条件是当前等待的线程被中断后代码会走到break退出，

> 第二种情况是当前节点被移动到了同步队列中 （即另外线程调用的condition的signal或者signalAll方法），while中逻辑判断为false后结束while循环。
> **总结下**，就是 当前线程被中断或者调用condition.signal/condition.signalAll方法当前节点移动到了同步队列后 ，这是当前线程退 出await方法的前提条件。当退出while循环后就会调用acquireQueued(node, savedState)，这个方法在介绍AQS的 底层实现时说过了，该方法的作用是在自旋过程中线程不断尝试获取同步状态，直至成功（线程获取到lock）。这样也说明了调用await方法必须是已经获得了condition引用（关联）的lock。

5.7await()流程图：



如图，调用condition.await方法的线程必须是已经获得了lock，也就是当前线程是同步队列中的头结点。调用该方法 后会使得当前线程所封装的Node尾插入到等待队列中

6.signal实现原理

调用condition的signal或者signalAll方法可以将等待队列中等待时间最长的节点移动到同步队列中，使得该节点能 够有机会获得lock。按照等待队列是先进先出（FIFO）的，所以等待队列的头节点必然会是等待时间长的节点， 也就是每次调用condition的signal方法是将头节点移动到同步队列中。

6.1signal

```
1  public final void signal() {
```

```
 //当前线程是否已经获取到lock
 if (!isHeldExclusively()) //如果获取到了则抛出异常
 throw new IllegalMonitorStateException();
 Node first = firstWaiter;//获取等待队列的第一个元素，对其操作
 if (first != null)
 doSignal(first);
}
```

6.2 doSignal(Node first) --ransferForSignal(first)对等待队列头节点唤醒，放置同步队列中

```
private void doSignal(Node first) {
 do {
 if ( (firstWaiter = first.nextWaiter) == null)//第一个节点的下一个节点为空
 lastWaiter = null;//等待队列的最后一个节点为空
 first.nextWaiter = null;//第一个节点的下一个节点为空，移除头节点
 } while (!transferForSignal(first) && // transferForSignal方法对头结点做真正的处理
 (first = firstWaiter) != null);
}


final boolean transferForSignal(Node node) {
 /*
 * If cannot change waitStatus, the node has been cancelled.
 */
 //首先将节点状态更新为0，失败返回false;
 if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
 return false;

 /*
 * Splice onto queue and try to set waitStatus of predecessor to
 * indicate that thread is (probably) waiting. If cancelled or
 * attempt to set waitStatus fails, wake up to resync (in which
 * case the waitStatus can be transiently and harmlessly wrong).
 */
 //将当前节点enq（）方法尾插到同步队列中去。
 Node p = enq(node);
 int ws = p.waitStatus;
 if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))//如果竞争获取锁失败则插入同步队列队尾，且将其状态设置为SIGNAL
 LockSupport.unpark(node.thread);//阻塞该线程
```

```
30   return true;//唤醒成功
31  }
32
```

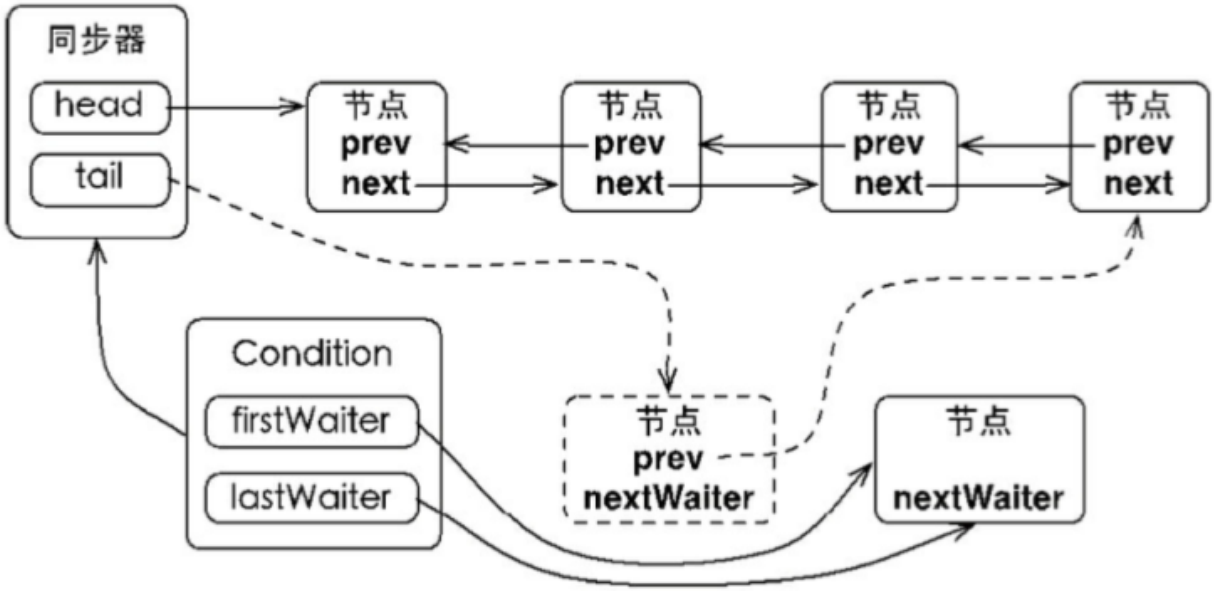
这段代码主要做了两件事情

1.将头结点的状态更改为CONDITION；

2.调用enq方法，将该节点 尾插入到同步队列中。

现在我们可以得出结论：调**用condition的signal的前提条件是当前线程已经获取了lock，该 方法会使得等待队列中的头节点即等待时间最长的那个节点移入到同步队列，而移入到同步队列后才有机会使得等 待线程被唤醒**，即从await方法中的LockSupport.park(this)方法中返回，从而才有机会使得调用await方法的线程成 功退出。

6.3signal执行流程图



7. signalAll实现原理

与sigal方法的区别体现在**doSignalAll方法上**，doSignal方法只会对等待队列的头节点进行操作：该方法只不过是将等待队列中的每一个节点都移入到同步队列中，**即“通知”当前调用condition.await()方法的每一个线程。**

```
1  public final void signalAll() {
2   //当前线程是否已经获取到lock
3   if (!isHeldExclusively()) //如果获取到了则抛出异常
4   throw new IllegalMonitorStateException();
5   Node first = firstWaiter;
6   if (first != null)
7   doSignalAll(first);
```

```
}
```

7.1 doSignalAll(first)循环式从头到尾唤醒节点

```
private void doSignalAll(Node first) {
 lastWaiter = firstWaiter = null; //头尾节点设置为空
 do {
 Node next = first.nextWaiter;
 first.nextWaiter = null;
 transferForSignal(first); //循环式从头到尾唤醒节点，仍到同步队列里面去
 first = next;
 } while (first != null); //头节点不为空
}


final boolean transferForSignal(Node node) {
 /*
 * If cannot change waitStatus, the node has been cancelled.
 */
 //首先将节点状态更新为0，失败返回false;
 if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
 return false;

 /*
 * Splice onto queue and try to set waitStatus of predecessor to
 * indicate that thread is (probably) waiting. If cancelled or
 * attempt to set waitStatus fails, wake up to resync (in which
 * case the waitStatus can be transiently and harmlessly wrong).
 */
 //将当前节点enq（）方法尾插到同步队列中去。
 Node p = enq(node);
 int ws = p.waitStatus;
 if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))//如果竞争获取锁失败则插入同步队列队尾，且将其状态设置为SIGNAL
 LockSupport.unpark(node.thread);//阻塞该线程
 return true;//唤醒成功
}
```