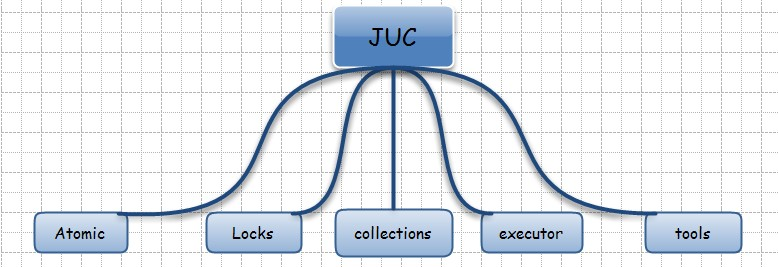
[**JUC 基础内容概述**](http://www.goldendoc.org/2011/05/juc/)

2011 年 05 月 13 日, 上午 12:31

*Concurrent Programming in Java*的作者 Doug Lea 编写了一个极其优秀的、免费的并发实用程序包，它包括并发应用程序的锁、互斥、队列、线程池、轻量级任务、有效的并发集合、原子的算术操作和其它基本构件。我们一般称这个包为 J.U.C。

**1. JUC概况**

以下是Java JUC包的主体结构：  


* Atomic : AtomicInteger
* Locks : Lock, Condition, ReadWriteLock
* Collections : Queue, ConcurrentMap
* Executer : Future, Callable, Executor
* Tools : CountDownLatch, CyclicBarrier, Semaphore

**2. 原子操作**

多个线程执行一个操作时，其中任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤，那么这个操作就是原子的。出现原因: synchronized的代价比较高。

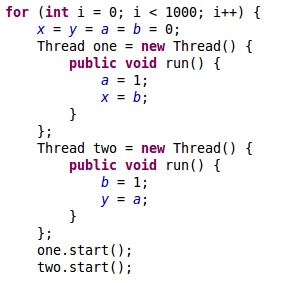
以下以AtomicInteger为例：

* int addAndGet(int delta)：以原子方式将给定值与当前值相加。 实际上就是等于线程安全版本的i =i+delta操作。
* boolean compareAndSet(int expect, int update)：如果当前值 == 预期值，则以原子方式将该值设置为给定的更新值。 如果成功就返回true，否则返回false，并且不修改原值。
* int decrementAndGet()：以原子方式将当前值减 1。 相当于线程安全版本的–i操作。
* int getAndAdd(int delta)：以原子方式将给定值与当前值相加。 相当于线程安全版本的t=i;i+=delta;return t;操作。
* int getAndDecrement()：以原子方式将当前值减 1。 相当于线程安全版本的i–操作。
* int getAndIncrement()：以原子方式将当前值加 1。 相当于线程安全版本的i++操作。
* int getAndSet(int newValue)：以原子方式设置为给定值，并返回旧值。 相当于线程安全版本的t=i;i=newValue;return t;操作。
* int incrementAndGet()：以原子方式将当前值加 1。 相当于线程安全版本的++i操作。

**3. 指令重排**

你的程序并不能总是保证符合CPU处理的特性。

要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。



多核CPU，大压力下，两个线程交替执行，x，y输出结果不确定。可能结果：

|  |  |
| --- | --- |
| 1  2  3  4 | x = 0， y = 1  x = 1， y = 1  x = 1， y = 0  x = 0， y = 0 |

**4. Happens-before法则：（Java 内存模型）**

如果动作B要看到动作A的执行结果（无论A/B是否在同一个线程里面执行），那么A/B就需要满足happens-before关系。

Happens-before的几个规则：

* Program order rule：同一个线程中的每个Action都happens-before于出现在其后的任何一个Action。
* Monitor lock rule：对一个监视器的解锁happens-before于每一个后续对同一个监视器的加锁。
* Volatile variable rule：对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作。
* Thread start rule：Thread.start()的调用会happens-before于启动线程里面的动作。
* Thread termination rule：Thread中的所有动作都happens-before于其他线程检查到此线程结束或者Thread.join（）中返回或者Thread.isAlive()==false。
* Interruption rule：一个线程A调用另一个另一个线程B的interrupt（）都happens-before于线程A发现B被A中断（B抛出异常或者A检测到B的isInterrupted（）或者interrupted()）。
* Finalizer rule：一个对象构造函数的结束happens-before与该对象的finalizer的开始
* Transitivity：如果A动作happens-before于B动作，而B动作happens-before与C动作，那么A动作happens-before于C动作。  
  因为CPU是可以不按我们写代码的顺序执行内存的存取过程的，也就是指令会乱序或并行运行， 只有上面的happens-before所规定的情况下，才保证顺序性。

**JMM的特性：**

多个CPU之间的缓存也不保证实时同步；  
JMM不保证创建过程的原子性，读写并发时，可能看到不完整的对象。（so D-check）

**volatile语义：**

volatile实现了类似synchronized的语义，却又没有锁机制。它确保对  volatile字段的更新以可预见的方式告知其他的线程。

1. Java 存储模型不会对volatile指令的操作进行重排序：这个保证对volatile变量的操作时按照指令的出现顺序执行的。
2. volatile变量不会被缓存在寄存器中（只有拥有线程可见），每次总是从主存中读取volatile变量的结果。

ps：volatile并不能保证线程安全的，也就是说volatile字段的操作不是原子性的，volatile变量只能保证可见性。

**5. CAS操作**

Compare and Swap

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

实现简单的非阻塞算法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private volatile int value;// 借助volatile原语，保证线程间的数据是可见的    public final int get() {      return value;  }    public final int incrementAndGet() {      for (;;) {          int current = get();          int next = current + 1;          if (compareAndSet(current, next))              return next;      }//Spin自旋等待直到返为止置  } |

整个J.U.C都是建立在CAS之上的，对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。会出现所谓的“ABA”问题

**6. Lock 锁**

Synchronized属于独占锁，高并发时性能不高，JDK5以后开始用JNI实现更高效的锁操作。

Lock—->

ReentrantLock—->

ReentrantReadWriteLock.ReadLock / ReentrantReadWriteLock.writeLock

ReadWriteLock—-> ReentrantReadWriteLock

LockSupport

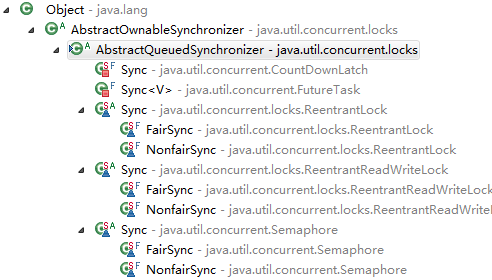
Condition

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| void lock() | 获取锁。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。 |
| void lockInterruptibly() throws InterruptedException; | 如果当前线程未被中断，则获取锁。如果锁可用，则获取锁，并立即返回。 |
| Condition newCondition(); | 返回绑定到此 Lock 实例的新 Condition 实例 |
| boolean tryLock(); | 仅在调用时锁为空闲状态才获取该锁 |
| boolean tryLock(long time, TimeUnit unit) throws InterruptedException; | 如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁 |
| void unlock(); | 释放锁 |

PS : 一般来说，获取锁和释放锁是成对儿的操作，这样可以避免死锁和资源的浪费。

**7. AQS**

锁机制实现的核心所在。AbstractQueuedSynchronizer是Lock/Executor实现的前提。



**AQS实现：**

基本的思想是表现为一个同步器，AQS支持下面两个操作：

**acquire[[1]](#footnote-2)：**

|  |  |
| --- | --- |
| 1  2  3  4  5 | while(synchronization state does not allow acquire){      enqueue current thread if not already queued;      possibly block current thread;  }  dequeue current thread if it was queued; |

**release：**

|  |  |
| --- | --- |
| 1  2  3 | update synchronization state;  if(state may permit[[2]](#footnote-3) a blocked thread to acquire)      unlock one or more queued threads; |

要支持这两个操作，需要实现的三个条件：

* Atomically managing synchronization state（原子性操作同步器的状态位）
* Blocking and unblocking threads（阻塞和唤醒线程）
* Maintaining queues（维护一个有序的队列）

**Atomically managing synchronization state**

使用一个32位整数来描述状态位：private volatile int state; 对其进行CAS操作，确保值的正确性。

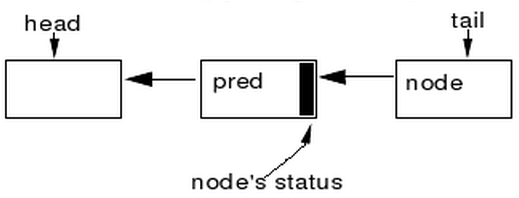
**Blocking and unblocking threads**

JDK 5.0以后利用JNI在LockSupport类中实现了线程的阻塞和唤醒。

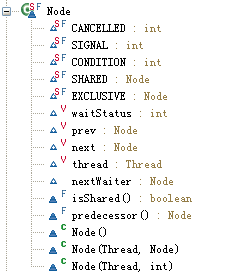
LockSupport.park() //在当前线程中调用，导致线程阻塞  
LockSupport.park(Object)  
LockSupport.unpark(Thread)

**Maintaining queues**

在AQS中采用CHL列表来解决有序的队列的问题。（CHL= Craig, Landin, and Hagersten）



Node里面是什么结构？



WaitStatus –>节点的等待状态，一个节点可能位于以下几种状态：

* CANCELLED = 1： 节点操作因为超时或者对应的线程被interrupt。节点不应该不留在此状态，一旦达到此状态将从CHL队列中踢出。
* SIGNAL = -1： 节点的继任节点是（或者将要成为）BLOCKED状态（例如通过LockSupport.park()操作），因此一个节点一旦被释放（解锁）或者取消就需要唤醒（LockSupport.unpack()）它的继任节点。
* CONDITION = -2：表明节点对应的线程因为不满足一个条件（Condition）而被阻塞。
* 0： 正常状态，新生的非CONDITION节点都是此状态。

非负值标识节点不需要被通知（唤醒）。  
队列管理操作：

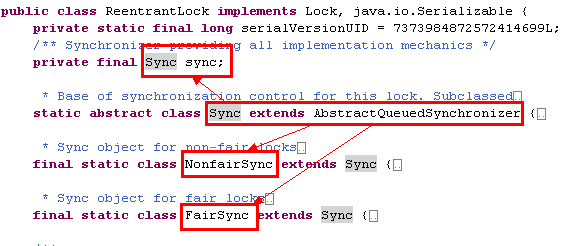
入队enqueue：

采用CAS操作，每次比较尾结点是否一致，然后插入的到尾结点中。

|  |  |
| --- | --- |
| 1  2  3 | do {      pred = tail;  } while ( !compareAndSet(pred,tail,node) ); |

出队dequeue：

|  |  |
| --- | --- |
| 1  2 | while (pred.status != RELEASED) ;      head  = node; |



**加锁操作：**

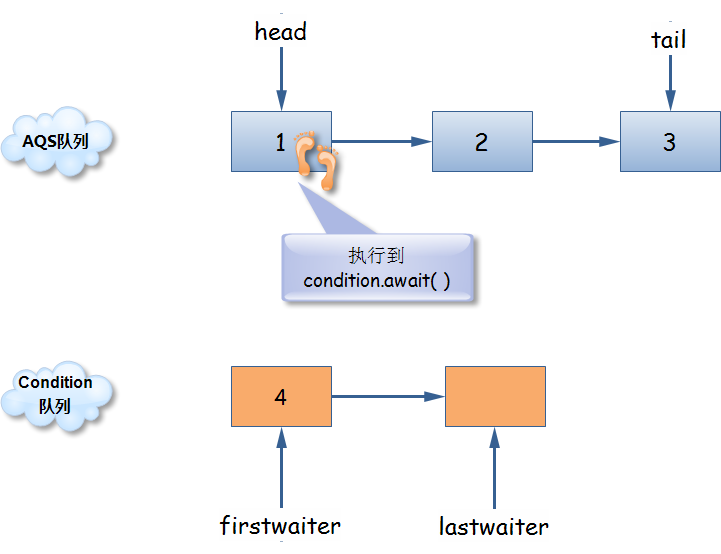
|  |  |
| --- | --- |
| 1  2  3  4  5 | public final void acquire(int arg) {      if (!tryAcquire(arg)          acquireQueued(addWaiter(Node.EXCLUSIVE), arg))      selfInterrupt();  } |

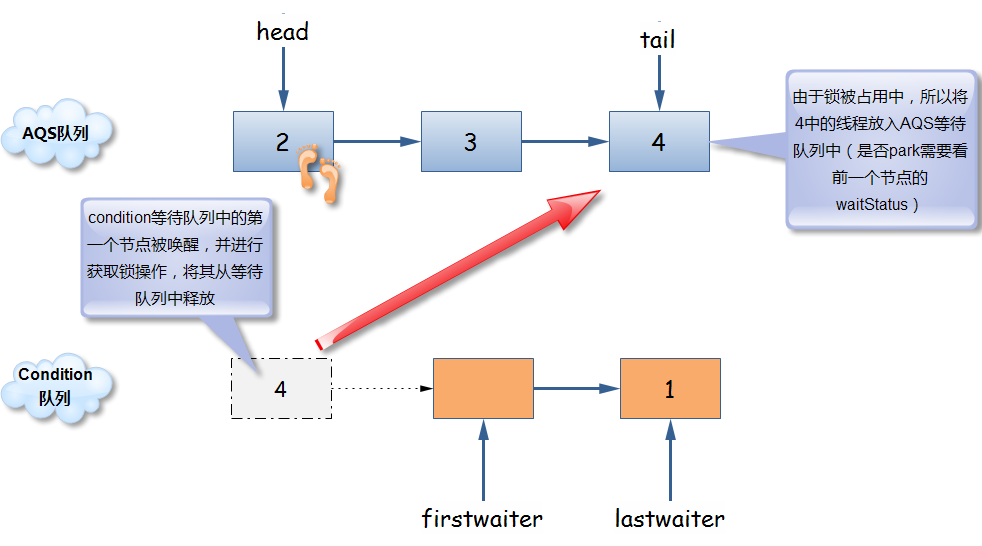
**释放操作：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public final boolean release(int arg) {      if (tryRelease(arg)) {          Node h = head;          if (h != null &amp;&amp; h.waitStatus != 0)              unparkSuccessor(h);          return true;      }      return false;  } |

The synchronizer framework provides a ConditionObject class for use by synchronizers that maintain exclusivesynchronization and conform to the Lock interface.     —— Doug Lea《 The java.util.concurrent Synchronizer Framework 》

以下是AQS队列和Condition队列的出入结点的示意图，可以通过这几张图看出线程结点在两个队列中的出入关系和条件。





## [Java并发编程J.U.C之锁的获取与释放](http://www.goldendoc.org/2011/06/lock_acquire_release/)

2011 年 06 月 13 日, 下午 2:05

上一篇文章中，我们对[J.U.C的一些大概的情况](http://www.goldendoc.org/2011/05/juc/)做了了解，在这一篇文章我们将来以ReentrantLock为例，来分析一下锁的获取和释放的过程，让大家能够对锁的获取和释放的整体过程有一个了解。

#### 一、锁的获取

先看下ReentrantLock的lock()方法，整个方法只有一行，调用acquire方法，看看acquire方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public final void acquire(int arg) {      if (!tryAcquire(arg) &&          acquireQueued(addWaiter(Node.EXCLUSIVE), arg))          selfInterrupt();  } |

这段代码的实现也是比较简洁，先尝试一次tryAcquire操作，如果失败，则把当前线程加入到同步队列中去，这个时候可能会反复的阻塞与唤醒这个线程，直到后续的tryAcquire（看acquireQueued的实现）操作成功。

再看看tryAcquire的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | protected final boolean tryAcquire(int acquires) {      final Thread current = Thread.currentThread();      int c = getState();      if (c == 0) {          if (isFirst(current) &&              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;      }      return false;  } |

这段代码是尝试获取锁的过程，它先判断当前的AQS的state值，如果为0，则表示该锁没有被持有过，如果这个时候同步队列是空的或者当前线程就是在同步队列的头部，那么修改state的值，并且设置排他锁的持有线程为当前线程

如果大于0，则判断当前线程是否是排他锁的持有线程，如果是，那么把state值加1（注意state是int类型的，所以state的最大值是就是int的最大值）

如果第一次tryAcquire()操作失败，那么就把当前线程加入到等待队列中去，看addWaiter()方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private Node addWaiter(Node mode) {      Node node = new Node(Thread.currentThread(), mode);      // Try the fast path of enq; backup to full enq on failure      Node pred = tail;      if (pred != null) {          node.prev = pred;          if (compareAndSetTail(pred, node)) {              pred.next = node;              return node;          }      }      enq(node);      return node;  } |

这段代码中先尝试了一下了下enq()方法中等待队列不为空的情况，如果失败，再调用enq()方法将当前线程加入等待队列，enq()的过程我们已经在上一篇文章中讲过了，不再赘述。

最后在当前线程被加入到等待队列中去以后，再调用acquireQueued去获取锁，看看acquireQueued的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | final boolean acquireQueued(final Node node, int arg) {      try {          boolean interrupted = false;          for (;;) {              final Node p = node.predecessor();              if (p == head && tryAcquire(arg)) {                  setHead(node);                  p.next = null; // help GC                  return interrupted;              }              if (shouldParkAfterFailedAcquire(p, node) &&                  parkAndCheckInterrupt())                  interrupted = true;          }      } catch (RuntimeException ex) {          cancelAcquire(node);          throw ex;      }  } |

这段代码中拿到当前线程在同步队列中的前面一个节点，如果这个节点是是头部，那么马上进行一次tryAcquire操作，如果操作成功，那么把当前线程弹出队列，整个操作就此结束。如果这个节点不是头部或者说tryAcquire操作失败的话，那么就判断是不是要将当前线程给阻塞掉（shouldParkAfterFailedAcquire）方法：判断当前线程是否应该被阻塞掉，实际上判断的是当前线程的前一个节点的状态，如果前一个节点的状态小于0（condition或者signal），那么返回true，阻塞当前线程；如果前一个节点的状态大于0（cancelled），则向前遍历，直到找到一个节点状态不大于0的节点，并且将中间的cancelled状态的节点全部踢出队列；如果前一个节点的状态等于0，那么将其状态置为-1（signal），并且返回false，等待下一次循环的时候再阻塞。

**整个锁的获取过程就是这样，我们再来总结一下整个过程：**acquire()方法会先调用一次tryAcquire方法获取一次锁，如果失败，则把当前线程加入到等待队列中去，然后再调用acquireQueued获取锁，acquireQueued在当前节点不在头部的时候会把当前线程的前一个结点的状态置为SIGNAL，然后阻塞当前线程。当当前线程到了队列的头部的时候，那么获取锁的操作就会成功返回。

#### 二、锁的释放

首先，我们知道在acquireQueued方法中，如果一个线程成功获取到了锁，那么它就应该是整个等待队列的head节点，然后，我们再来看一看unlock()方法，和lock()方法一样，unlock()方法也是只有一行代码，直接调用release()方法，我们看看release()方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public final boolean release(int arg) {      if (tryRelease(arg)) {          Node h = head;          if (h != null && h.waitStatus != 0)              unparkSuccessor(h);          return true;      }      return false;  } |

这个过程首先调用tryRelease方法，如果锁已经完全释放，那么就唤醒下一个节点，先来看看tryRelease方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | protected final boolean tryRelease(int releases) {      int c = getState() - releases;      if (Thread.currentThread() != getExclusiveOwnerThread())          throw new IllegalMonitorStateException();      boolean free = false;      if (c == 0) {          free = true;          setExclusiveOwnerThread(null);      }      setState(c);      return free;  } |

这段代码首先获取当前AQS的state状态并且将其值减一，如果结果等于0（锁已经被完全释放），那么将排他锁的持有线程置为null。将AQS的state状态置为减一后的结果。

然后再看看唤醒继任节点的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private void unparkSuccessor(Node node) {      compareAndSetWaitStatus(node, Node.SIGNAL, 0);      Node s = node.next;      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;      }      if (s != null)          LockSupport.unpark(s.thread);  } |

这段代码先清除当前节点的waitStatus为0，然后判断下一个节点是不是null或者cancelled的状态，如果是，则从队列的尾部往前开始找，找到一个非cancelled状态的节点，最后唤醒这个节点。

**最后，总结一下释放操作的整个过程：**其实整个释放过程就做了两件事情，一个是将state值减1，然后就是判断锁是否被完全释放，如果被完全释放，则唤醒继任节点。

#### 三、整体过程描述

看了上面的锁的获取与释放操作以后，整体过程还是比较清晰的，在文章的最后，我们把获取与释放操作串在一起在简单看一下：

* 获取锁的时候将当前线程放入同步队列，并且将前一个节点的状态置为signal状态，然后阻塞
* 当这个节点的前一个节点成功获取到锁，前一个节点就成了整个同步队列的head。
* 当前一个节点释放锁的时候，它就唤醒当前线程的这个节点，然后当前线程的节点就可以成功获取到锁了
* 这个时候它就到整个队列的头部了，然后release操作的时候又可以唤醒下一个。

## [Java并发编程J.U.C之Condition](http://www.goldendoc.org/2011/06/juc_condition/)

2011 年 06 月 17 日, 下午 4:29

在上一篇中，我们了解了下[J.U.C的锁的获取与释放的过程](http://www.goldendoc.org/2011/06/lock_acquire_release/)，这个过程主要通过在A.Q.S中维持一个等待队列来实现，其中我们也提到了，在A.Q.S中除了一个等待队列之外，还有一个Condition队列，在了解Condition队列之前，先来看一下Condition是怎么回事：

The synchronizer framework provides a ConditionObject class for use by synchronizers that maintain exclusive synchronization and conform to the Lock interface. Any number of condition objects may be attached to a lock object, providing classic monitor-style await, signal, and signalAll operations, including those with timeouts, along with some inspection and monitoring methods.

上面的这一段内容摘自Doug Lea的[AQS论文](http://gee.cs.oswego.edu/dl/papers/aqs.pdf)，从上面这一段话可以看出，Condition主要是为了在J.U.C框架中提供和Java传统的监视器风格的wait，notify和notifyAll方法类似的功能，那么先来解释一下这三个方法的作用：

* Object.wait()方法：使当前线程释放Object上的监视器并且挂起，直到有另外的线程调用Object.notify()方法或者Object.notifyAll()方法唤醒当前线程，当被唤醒后，Object.wait()方法会尝试重新获取监视器，成功获取后继续往下执行。注意Object.wait()方法只有在当前线程持有Object的监视器的时候才能够调用，不然会抛出异常。
* Object.notify()方法：用于唤醒另外一个调用了Object.wait()方法的线程，如果有多个都调用了Object.wait()方法，那么就会选择一个线程去notify()，具体选择哪一个和具体的实现有关，当前线程在调用Object.notify()方法以后会就释放Object的监视器，和wait()方法一样，Object.notify()方法只有在当前线程只有Object的监视器的时候才能够调用，不然就会抛出异常。
* Object.notifyAll()方法：唤醒所有调用了Object.wait()方法的线程，如果有多个线程调用了Object.wait()方法，那么就会引发这些线程之间的竞争，最后谁成功获取到Object的监视器和具体的实现有关，当前线程在调用Object.notifyAll()方法以后会就释放Object的监视器，和wait()方法一样，Object.notifyAll()方法只有在当前线程只有Object的监视器的时候才能够调用，不然就会抛出异常。

那么Condition是如何实现wait，notify和notifyAll方法的功能呢？我们接下来看：

在Condition中，wait，notify和notifyAll方法分别对应了await，signal和signalAll方法，当然Condition也提供了超时的、不可被中断的await()方法，不过我们主要还是看一看await，notify和notifyAll的实现，先看await：

#### await方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public final void await() throws InterruptedException {      if (Thread.interrupted())          throw new InterruptedException();      Node node = addConditionWaiter();      int savedState = fullyRelease(node);      int interruptMode = 0;      while (!isOnSyncQueue(node)) {          LockSupport.park(this);          if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)              break;      }      if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)          interruptMode = REINTERRUPT;      if (node.nextWaiter != null)          unlinkCancelledWaiters();      if (interruptMode != 0)          reportInterruptAfterWait(interruptMode);  } |

整个await的过程如下：

1. 在第2行处，如果当前线程被中断，则抛出中断异常。
2. 在第4行处，将节点加入到Condition队列中去，这里如果lastWaiter是cancel状态，那么会把它踢出Condition队列。
3. 在第5行处，调用tryRelease，释放当前线程的锁
4. 在第7行处，判断节点是否在等待队列中（signal操作会将Node从Condition队列中拿出并且放入到等待队列中去），如果不在等待队列中了，就park当前线程，如果在，就退出循环，这个时候如果被中断，那么就退出循环
5. 在第12行处，这个时候线程已经被signal()或者signalAll()操作给唤醒了，退出了4中的while循环，尝试再次获取锁，调用acquireQueued方法。

可以看到，这个await的操作过程和Object.wait()方法是一样，只不过await()采用了Condition队列的方式实现了Object.wait()的功能。

#### signal和signalAll方法：

在了解了await方法的实现以后，signal和signalAll方法的实现就相对简单了，先看看signal方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public final void signal() {      if (!isHeldExclusively())          throw new IllegalMonitorStateException();      Node first = firstWaiter;      if (first != null)          doSignal(first);  } |

这里先判断当前线程是否持有锁，如果没有持有，则抛出异常，然后判断整个condition队列是否为空，不为空则调用doSignal方法来唤醒线程，看看doSignal方法都干了一些什么：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private void doSignal(Node first) {      do {          if ( (firstWaiter = first.nextWaiter) == null)              lastWaiter = null;          first.nextWaiter = null;      } while (!transferForSignal(first) &&               (first = firstWaiter) != null);  } |

这个while循环的作用就是将firstWaiter往Condition队列的后面移一位，并且唤醒first，看看while循环中tranferForSignal：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | final boolean transferForSignal(Node node) {      /\*       \* If cannot change waitStatus, the node has been cancelled.       \*/      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).       \*/      Node p = enq(node);      int c = p.waitStatus;      if (c > 0 || !compareAndSetWaitStatus(p, c, Node.SIGNAL))          LockSupport.unpark(node.thread);      return true;  } |

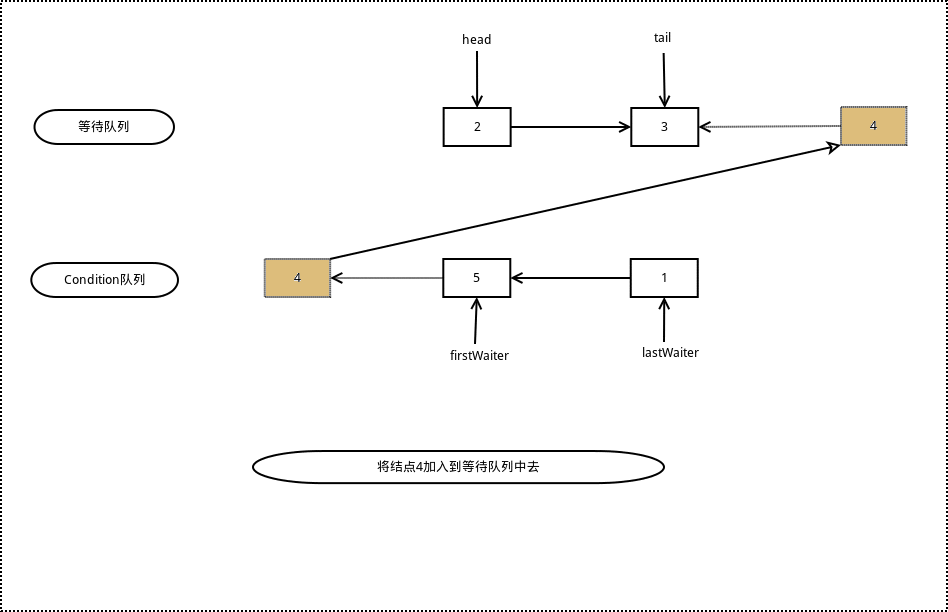
这段代码的作用就是修改Node的waitStatus为0，然后将Node插入到等待队列中，并且唤醒Node。

signalAll和signal方法类似，主要的不同在于它不是调用doSignal方法，而是调用doSignalAll方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private void doSignalAll(Node first) {      lastWaiter = firstWaiter  = null;      do {          Node next = first.nextWaiter;          first.nextWaiter = null;          transferForSignal(first);          first = next;      } while (first != null);  } |

这个方法就相当于把Condition队列中的所有Node全部取出插入到等待队列中去。

#### 总结：

在了解了await()，signal()和signalAll方法的实现以后，我们再来通过一副gif动画来看一看这一个整体的过程：  
[](http://www.goldendoc.org/wp-content/uploads/2011/06/animation.gif)

1. acquire 获得，取得 [↑](#footnote-ref-2)
2. permit 许可、允许 [↑](#footnote-ref-3)