**J.U.C Lock源码分析**

**八两俊**

**2014.12**

锁是控制多个线程对共享资源进行访问的工具。通常，锁提供了对共享资源的独占访问。一次只能有一个线程获得锁，对共享资源的所有访问都需要首先获得锁。不过，某些锁可能允许对共享资源并发访问，如ReadWriteLock的读取锁。

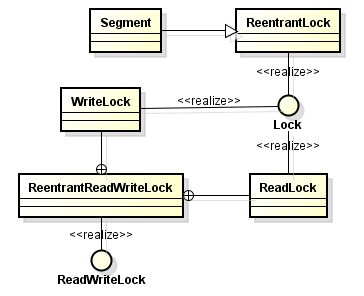
synchronized是Java内置的一种锁机制，它是独占锁。synchronized方法或语句的使用提供了对与每个对象相关的隐式监视器锁的访问，但却强制所有锁获取和释放均要出现在一个块结构中：当获取了多个锁时，它们必须以相反的顺序释放，且必须在与所有锁被获取时相同的词法范围内释放所有锁。

Lock 实现提供了使用 synchronized 方法和语句所没有的其他功能，包括提供了一个非块结构的获取锁尝试 (tryLock())、一个获取可中断锁的尝试 (lockInterruptibly()) 和一个获取超时失效锁的尝试 (tryLock(long, TimeUnit))。

Lock 类还可以提供与隐式监视器锁完全不同的行为和语义，如保证排序、非重入用法或死锁检测。如果某个实现提供了这样特殊的语义，则该实现必须对这些语义加以记录。

注意，Lock 实例只是普通的对象，其本身可以在 synchronized 语句中作为目标使用。获取 Lock 实例的监视器锁与调用该实例的任何 [lock()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html#lock()) 方法没有特别的关系。为了避免混淆，建议除了在其自身的实现中之外，决不要以这种方式使用 Lock 实例。

synchronized采用了阻塞算法，而Lock采用了基于处理器的CAS操作的非阻塞算法，Lock提供了更细粒度的锁实现。



Lock类关系图

# ReentrantLock

一个可重入的互斥锁 [Lock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html)，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。

ReentrantLock 将由最近成功获得锁，并且还没有释放该锁的线程所拥有。当锁没有被另一个线程所拥有时，调用 lock 的线程将成功获取该锁并返回。如果当前线程已经拥有该锁，此方法将立即返回。可以使用 isHeldByCurrentThread() 和 getHoldCount() 方法来检查此情况是否发生。

此类的构造方法接受一个可选的公平参数。当设置为true时，在多个线程的争用下，这些锁倾向于将访问权授予等待时间最长的线程。否则此锁将无法保证任何特定访问顺序。与采用默认设置（使用不公平锁）相比，使用公平锁的程序在许多线程访问时表现为很低的总体吞吐量（即速度很慢，常常极其慢），但是在获得锁和保证锁分配的均衡性时差异较小。不过要注意的是，公平锁不能保证**线程调度**的公平性。因此，使用公平锁的众多线程中的一员可能获得多倍的成功机会，这种情况发生在其他活动线程没有被处理并且目前并未持有锁时。还要注意的是，未定时的 tryLock 方法并没有使用公平设置。因为即使其他线程正在等待，只要该锁是可用的，此方法就可以获得成功。

1、默认构造器实例化

**public** ReentrantLock() {

sync = **new** NonfairSync();// 默认使用不公平锁

}

2、创建一个具有给定公平策略的 ReentrantLock

**public** ReentrantLock(**boolean** fair) {

sync = (fair)? **new** FairSync() : **new** NonfairSync();// 根据入参判断是创建不公平锁还是公平锁

}

## ReentrantLock.lock()

从构造器的定义上来看，看以看出ReentrantLock支持公平锁和非公平锁两种锁机制，接下来，我们从这两种锁机制的源码进行分析。

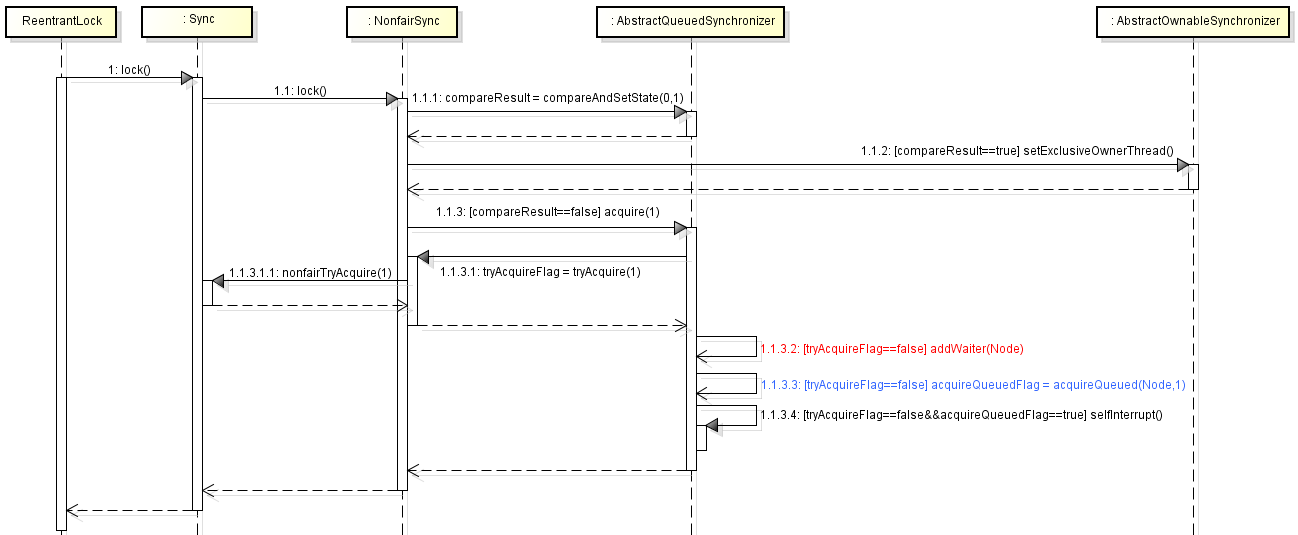
### ReentrantLock.NonfairSync.lock()

获取锁。

如果该锁没有被另一个线程保持，则获取该锁并立即返回，将锁的保持计数设置为 1。

如果当前线程已经保持该锁，则将保持计数加1，并且该方法立即返回。

如果该锁被另一个线程保持，则出于线程调度的目的，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，此时锁保持计数被设置为1。



ReentrantLock 非公平锁lock处理时序图

处理时序

1、调用ReentrantLock的lock()方法，在此方法中会根据初始化ReentrantLock的锁机制(公平所还是非公平锁)进行进一步的锁控制处理，这里以非公平锁[NonfairSync]为例。

2、NonfairSync.lock()方法处理：

final void lock() {

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

compareAndSetState方法是AbstractQueuedSynchronizer抽象类提供的，采用处理器CAS操作设置锁的同步状态。

判断当前锁的状态是否为0[未锁状态]，如果是则将锁的状态设置为1[锁定状态]，并调用AbstractOwnableSynchronizer的setExclusiveOwnerThread方法将当前锁的拥有者设置为当前调用线程。

如果compareAndSetState方法返回false[锁已经被其他线程独占]，则调用AbstractQueuedSynchronizer的acquire方法获取锁。

3、AbstractQueuedSynchronizer. acquire(int)方法处理

public final void acquire(int arg) {// arg 值为1

// 尝试获得锁，如果无法获得锁则将请求线程挂起

if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

3.1、tryAcquire(arg)：回调NonfairSync的tryAcquire方法尝试获得锁，其内部将进一步调用其匪类Sync的nonfairTryAcquire方法获得锁。

final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

} else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0) // overflow

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

采用不公平方式所得锁。

判断当前锁的状态是否为0[未锁状态]，如果为零则独占锁，并调用AbstractOwnableSynchronizer的setExclusiveOwnerThread方法将当前锁的拥有者设置为当前调用线程，返回true；如果当前线程已经是锁的拥有者，则将同步状态state加1，返回true；否则返回false。

3.2、addWaiter(Node.EXCLUSIVE)方法。

tryAcquire失败就意味着入队列了。此时AQS的队列中节点Node就开始发挥作用了。一般情况下AQS支持独占锁和共享锁，而独占锁在Node中就意味着条件（Condition）队列为空。在java.util.concurrent.locks.AbstractQueuedSynchronizer.Node中有两个常量，

static final Node EXCLUSIVE = null; //独占节点模式

static final Node SHARED = new Node(); //共享节点模式

private Node addWaiter(Node mode) {// mode为Node. EXCLUSIVE

Node node = new Node(Thread.currentThread(), mode);

Node pred = tail;

if (pred != null) {

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

private Node enq(final Node node) {

for (;;) {

Node t = tail;

if (t == null) { // Must initialize

if (compareAndSetHead(new Node()))

tail = head;

} else {

node.prev = t;

if (compareAndSetTail(t, node)) {

t.next = node;

return t;

}

}

}

}

用当前线程构建Node节点，将Node加入到CLH队列末端。

Node类中定义的等待状态

/\*\*因为超时或中断，该线程已经被取消\*/

static final int CANCELLED = 1;//

/\*\*线程的后继线程正/已被阻塞，当该线程release或cancel时要重新这个后继线程(unpark) \*/

static final int SIGNAL = -1;

/\*\*表明该线程被处于条件队列，就是因为调用了Condition.await而被阻塞 \*/

static final int CONDITION = -2;

/\*传播共享锁\*/

static final int PROPAGATE = -3;

3.3、acquireQueued：**自旋请求锁**，如果可能的话挂起线程，直到得到锁，返回当前线程是否中断过（如果park()过并且中断过的话有一个interrupted中断位）。

**final** **boolean** acquireQueued(**final** Node node, **int** arg) {

**boolean** failed = **true**;

**try** {

**boolean** interrupted = **false**;

**for** (;;) {

**final** Node p = node.predecessor();

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return** interrupted;

}

// shouldParkAfterFailedAcquire:如果Node的waitStatus为-1，表示需要park

// parkAndCheckInterrupt:将当前线程park，并返回当前线程的中断状态位,并清空中断状态位

**if** (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())

interrupted = **true**;

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

1、如果当前节点是AQS队列的头结点（如果第一个节点是DUMP节点也就是傀儡节点，那么第二个节点实际上就是头结点了），就尝试在此获取锁tryAcquire(arg)。如果成功就将头结点设置为当前节点（不管第一个结点是否是DUMP节点），返回中断位。否则进行2。

2、检测当前节点是否应该park()，如果应该park()就挂起当前线程并且返回当前线程中断位。进行操作1。

shouldParkAfterFailedAcquire方法：

**private** **static** **boolean** shouldParkAfterFailedAcquire(Node pred, Node node) {

**int** ws = pred.waitStatus;

// 如果节点的waitStatus为-1则表明这个节点已经设置状态要求释放信号，则可以被安全的park

**if** (ws == Node.*SIGNAL*)

**return** **true**;

**if** (ws > 0) {

// 如果waitStatus大于0，表明前一个节点已经被取消，所以要移除前一个节点

**do** {

node.prev = pred = pred.prev;

} **while** (pred.waitStatus > 0);

pred.next = node;

} **else** {

// 这个时候waitStatus为0，-2(Node.CONDITION)， -3(Node.PROPAGATE)，设置其前一个节点的等待状态为-1，然后在acquireQueued中再次自旋将其挂起。

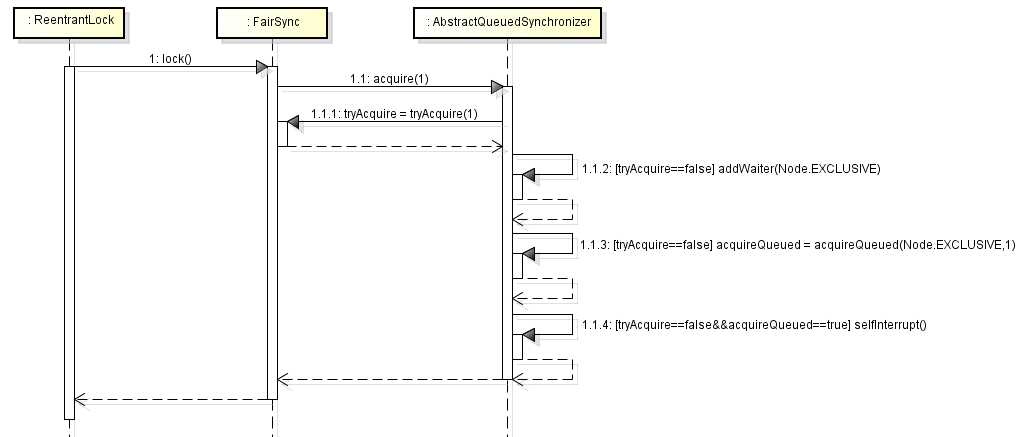
*compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*);

}

**return** **false**;

}

### ReentrantLock.FairSync.lock()



ReentrantLock公平锁lock处理时序图

对比公平锁和非公平锁的时序图，对于非公平锁，首先比较当前锁状态，如果为0则独占锁，并立即返回；否则将请求交给AbstractQueuedSynchronizer的acquire(int)去尝试获得锁。而对于公平锁，他首先不会去判断锁的状态，而是直接将请求交给AbstractQueuedSynchronizer的acquire(int)方法去尝试获得锁。

从AbstractQueuedSynchronizer中的acquire(1)方法中可以看出，在处理尝试获得锁的逻辑时，AbstractQueuedSynchronizer都是交给子类的tryAcquire(1)方法处理，所以，公平锁和不公平锁的获得锁的不同性在tryAcquire(1)中体现。以下是公平锁FairSync的tryAcquire(1)处理逻辑。

**protected** **final** **boolean** tryAcquire(**int** acquires) {

**final** Thread current = Thread.*currentThread*();

**int** c = getState();

**if** (c == 0) {

**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**;

}

**return** **false**;

}

1、首先判断锁的状态是否为0(未占用)，如果未占用则调用hasQueuedPredecessors()判断AQS的CLH队列中是否还有等待节点，判断锁的状态是否为0，如果上述两个条件则当前线程独占锁，返回true。

2、如果上一步锁已经被占用，则判断占用锁的线程是否为当前线程，如果是则将锁的状态state加1。

然后我们返回回来看AbstractQueuedSynchronizer的acquire(1)方法。

**public** **final** **void** acquire(**int** arg) {

**if** (!tryAcquire(arg) && acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))

*selfInterrupt*();

}

这样我们可以发现，不管是公平锁还是不公平锁最后在tryAcquire方法未获得锁之后的处理逻辑是一致的。

## ReentrantLock.lockInterruptibly()

### ReentrantLock.NonfairSync.lockInterruptibly()

如果当前线程未被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，则获取锁。

如果锁可用，则获取锁，并立即返回。

如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态：

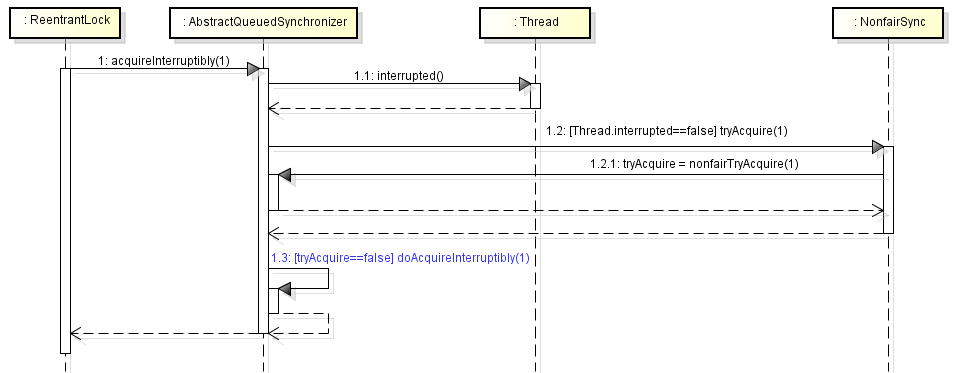
a、锁由当前线程获得；

b、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程，并且支持对锁获取的中断。

如果当前线程：

1. 在进入此方法时已经设置了该线程的中断状态；
2. 在获取锁时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，并且支持对锁获取的中断，

则将抛出 [InterruptedException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\InterruptedException.html)，并清除当前线程的已中断状态。



ReentrantLock lockInterruptibly()时序图

核心方法：acquireInterruptibly

**public** **final** **void** acquireInterruptibly(**int** arg) **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**if** (!tryAcquire(arg))

doAcquireInterruptibly(arg);

}

通过Thread.interrupted()判断当前线程是否被中断，如果被中断则停止获得锁并抛出异常清空中断状态。

如果没有被中断，则调用tryAcquire(1)获得锁，如果没有获得锁，则调用doAcquireInterruptibly(1)方法。

**private** **void** doAcquireInterruptibly(**int** arg) **throws** InterruptedException {

**final** Node node = addWaiter(Node.*EXCLUSIVE*);

**boolean** failed = **true**;

**try** {

**for** (;;) {

**final** Node p = node.predecessor();

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return**;

}

**if** (*shouldParkAfterFailedAcquire*(p, node) && parkAndCheckInterrupt())

**throw** **new** InterruptedException();

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

在doAcquireInterruptibly方法先尝试获得锁，如果没有获得锁则将当前线程对应的Node加入到CLH队列的末端，并以轮询的形式在shouldParkAfterFailedAcquire()获取并更新Node节点的状态，如果在节点的waitStatus为-1(Node.SIGNAL)，则调用parkAndCheckInterrupt挂起当前线程，如果当前线程被中断，则抛出InterruptedException，终止锁获取。

### ReentrantLock.FairSync.lockInterruptibly()

ReentrantLocK的公平锁FairSync在处理lockInterruptibly是的逻辑与非公平锁NonfairSync几乎一致，就是在tryAcquire方法的处理不一致，可以参考公平锁lock方法对tryAcquire的处理。

这个方法在获取锁的时候，支持可中断。

## ReentrantLock.tryLock()

仅在调用时锁为空闲状态才获取该锁。

如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。

**public** **boolean** tryLock() {

**return** sync.nonfairTryAcquire(1);

}

从方法的内部来看，其实他还是调用了ReentrantLock.Sync.nonfairTryAcquire()方法，这个方法在前面[ReentrantLock.NonfairSync.lock()](#_ReentrantLock.NonfairSync.lock())中有介绍，他采用的是不公平的获得锁机制，不管获得了锁还是没获得锁都立即返回。如果没有获得锁，也不会挂起线程。

## ReentrantLock.tryLock(long, TimeUnit)

### ReentrantLock.NonfairSync.tryLock(long, TimeUnit)

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。

如果锁可用，则此方法将立即返回值 true。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下三种情况之一前，该线程将一直处于休眠状态：

a、锁由当前线程获得；

b、其他某个线程中断当前线程，并且支持对锁获取的中断；

c、已超过指定的等待时间

如果获得了锁，则返回值 true。

如果当前线程：

b、在进入此方法时已经设置了该线程的中断状态；

c、在获取锁时被中断，并且支持对锁获取的中断，

则将抛出 InterruptedException，并会清除当前线程的已中断状态。

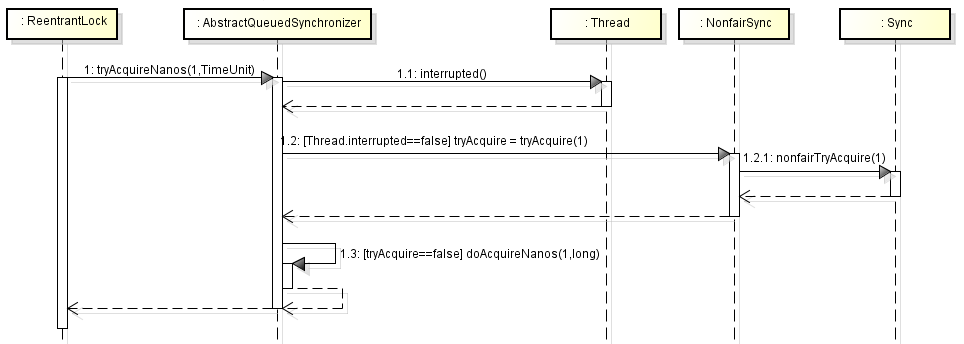
如果超过了指定的等待时间，则将返回值 false。如果time小于等于 0，该方法将完全不等待。

**实现注意事项**

在某些实现中可能无法中断锁获取，即使可能，该操作的开销也很大。程序员应该知道可能会发生这种情况。在这种情况下，该实现应该对此进行记录。

相对于普通方法返回而言，实现可能更喜欢响应某个中断，或者报告出现超时情况。

Lock 实现可能可以检测锁的错误用法，例如，某个调用可能导致死锁，在特定的环境中可能抛出（未经检查的）异常。该 Lock 实现必须对环境和异常类型进行记录。



ReentrantLock.tryLock(Long,TimeUnit)时序图

**public** **boolean** tryLock(**long** timeout, TimeUnit unit) **throws** InterruptedException {

**return** sync.tryAcquireNanos(1, unit.toNanos(timeout));

}

从上面的方法可以看出，整个tryLock方法的实现将交由AbstractQueuedSynchronizer.tryAcquireNanos实现完成。

**public** **final** **boolean** tryAcquireNanos(**int** arg, **long** nanosTimeout) **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**return** tryAcquire(arg) || doAcquireNanos(arg, nanosTimeout);

}

首先判断判断当前线程是否已经中断，如果中断则清除中断状态位，并抛出InterruptedException，终止获得锁逻辑。

如果线程未中断，调用tryAcquire尝试获得锁，如果获得锁则立即返回，否则调用doAcquireNanos完成超时时间内获得锁操作。

**private** **boolean** doAcquireNanos(**int** arg, **long** nanosTimeout) **throws** InterruptedException {

**long** lastTime = System.*nanoTime*();

**final** Node node = addWaiter(Node.*EXCLUSIVE*);

**boolean** failed = **true**;

**try** {

**for** (;;) {

**final** Node p = node.predecessor();

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return** **true**;

}

**if** (nanosTimeout <= 0)

**return** **false**;

// 判断节点等待状态(waitStatus)，如果waitStatus为-1且等待的时间超过1000L则表示需要park当前线程

// 此方法支持中断。

**if** (*shouldParkAfterFailedAcquire*(p, node) && nanosTimeout > *spinForTimeoutThreshold*)

LockSupport.*parkNanos*(**this**, nanosTimeout);

**long** now = System.*nanoTime*();

nanosTimeout -= now - lastTime;

lastTime = now;

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

### ReentrantLock.FairSync.tryLock(long, TimeUnit)

此方法的处理逻辑跟NonfairSync的处理逻辑只是在tryAcquire方法上不一致，可以参考对应FairSync中对tryAcquire方法的处理逻辑。

获得锁的过程也是可中断的。

## ReentrantLock.unLock()

释放锁。

**实现注意事项**

Lock 实现通常对哪个线程可以释放锁施加了限制（通常只有锁的保持者可以释放它），如果违背了这个限制，可能会抛出（未经检查的）异常。该 Lock 实现必须对所有限制和异常类型进行记录。

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);

return true;

}

return false;

}

整个tryRelease操作是这样的：

1、判断持有锁的线程是否是当前线程，如果不是就抛出IllegalMonitorStateExeception()，因为一个线程是不能释放另一个线程持有的锁（否则锁就失去了意义）。否则进行2。

2、将AQS状态位减少要释放的次数（对于独占锁而言总是1），如果剩余的状态位0（也就是没有线程持有锁），那么当前线程就是最后一个持有锁的线程，清空AQS持有锁的独占线程。进行3。

3、将剩余的状态位写回AQS，如果没有线程持有锁就返回true，否则就是false。

**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;

}

释放锁后，唤醒继任者节点线程。

**private** **void** unparkSuccessor(Node node) {

**int** ws = node.waitStatus;

**if** (ws < 0)// -1,-2,-3

*compareAndSetWaitStatus*(node, ws, 0);

Node s = node.next;

**if** (s == **null** || s.waitStatus > 0) {

s = **null**;

// 取消被取消的后继节点

**for** (Node t = tail; t != **null** && t != node; t = t.prev)

**if** (t.waitStatus <= 0)

s = t;

}

**if** (s != **null**)// 如果找到有效节点，则唤醒其对应的线程

LockSupport.*unpark*(s.thread);  
 }

## ReentrantLock.getHoldCount()

查询当前线程保持此锁的次数。

方法内部逻辑先判断当前线程是否占有锁，如果没有则返回0；如果占有锁则获取AbstractQueuedSynchronizer中锁的状态为state。

## ReentrantLock.newCondition()

返回一个Condition实例。

# ReadWriteLock

ReadWriteLock 维护了一对相关的[锁](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html)，一个用于**只读**操作，另一个用于写入操作。只要没有 writer，[读取锁](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\ReadWriteLock.html#readLock())可以由多个 reader 线程同时保持。[写入锁](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\ReadWriteLock.html#writeLock())是独占的。

所有ReadWriteLock实现都必须保证 writeLock 操作的内存同步效果也要保持与相关 readLock 的联系。也就是说，成功获取读锁的线程会看到写入锁之前版本所做的所有更新。

与互斥锁相比，读-写锁允许对共享数据进行更高级别的并发访问。虽然一次只有一个线程（writer 线程）可以修改共享数据，但在许多情况下，任何数量的线程可以同时读取共享数据（reader 线程），读-写锁利用了这一点。从理论上讲，与互斥锁相比，使用读-写锁所允许的并发性增强将带来更大的性能提高。在实践中，只有在多处理器上并且只在访问模式适用于共享数据时，才能完全实现并发性增强。

与互斥锁相比，使用读-写锁能否提升性能则取决于读写操作期间读取数据相对于修改数据的频率，以及数据的争用——即在同一时间试图对该数据执行读取或写入操作的线程数。例如，某个最初用数据填充并且之后不经常对其进行修改的collection，因为经常对其进行搜索（比如搜索某种目录），所以这样的 collection 是使用读-写锁的理想候选者。但是，如果数据更新变得频繁，数据在大部分时间都被独占锁，这时，就算存在并发性增强，也是微不足道的。更进一步地说，如果读取操作所用时间太短，则读-写锁实现（它本身就比互斥锁复杂）的开销将成为主要的执行成本，在许多读-写锁实现仍然通过一小段代码将所有线程序列化时更是如此。最终，只有通过分析和测量，才能确定应用程序是否适合使用读-写锁。

尽管读-写锁的基本操作是直截了当的，但实现仍然必须作出许多决策，这些决策可能会影响给定应用程序中读-写锁的效果。这些策略的例子包括：

1、在 writer 释放写入锁时，reader 和 writer 都处于等待状态，在这时要确定是授予读取锁还是授予写入锁。Writer 优先比较普遍，因为预期写入所需的时间较短并且不那么频繁。Reader 优先不太普遍，因为如果 reader 正如预期的那样频繁和持久，那么它将导致对于写入操作来说较长的时延。公平或者“按次序”实现也是有可能的。

2、在 reader 处于活动状态而 writer 处于等待状态时，确定是否向请求读取锁的 reader 授予读取锁。Reader 优先会无限期地延迟 writer，而 writer 优先会减少可能的并发。

3、确定是否重新进入锁：可以使用带有写入锁的线程重新获取它吗？可以在保持写入锁的同时获取读取锁吗？可以重新进入写入锁本身吗？

4、可以将写入锁在不允许其他 writer 干涉的情况下降级为读取锁吗？可以优先于其他等待的 reader 或 writer 将读取锁升级为写入锁吗？

当评估给定实现是否适合您的应用程序时，应该考虑所有这些情况。

# ReentrantReadWriteLock

**获取顺序**

此类不会将读取者优先或写入者优先强加给锁访问的排序。但是，它确实支持可选的公平 策略。

**非公平模式（默认）**

当非公平地（默认）构造时，未指定进入读写锁的顺序，受到 reentrancy [重入]约束的限制。连续竞争的非公平锁可能无限期地推迟一个或多个 reader 或 writer 线程，但吞吐量通常要高于公平锁。

**公平模式**

当公平地构造线程时，线程利用一个近似到达顺序的策略来争夺进入。当释放当前保持的锁时，可以为等待时间最长的单个 writer 线程分配写入锁，如果有一组等待时间大于所有正在等待的 writer 线程的 reader 线程，将为该组分配写入锁。

如果保持写入锁，或者有一个等待的 writer 线程，则试图获得公平读取锁（非重入地）的线程将会阻塞。直到当前最旧的等待 writer 线程已获得并释放了写入锁之后，该线程才会获得读取锁。当然，如果等待 writer 放弃其等待，而保留一个或更多 reader 线程为队列中带有写入锁自由的时间最长的 waiter，则将为那些 reader 分配读取锁。

试图获得公平写入锁的（非重入地）的线程将会阻塞，除非读取锁和写入锁都自由（这意味着没有等待线程）。（注意，非阻塞ReentrantReadWriteLock.ReadLock.tryLock() 和 ReentrantReadWriteLock.WriteLock.tryLock() 方法不会遵守此公平设置，并将获得锁（如果可能），不考虑等待线程）。

**重入**

此锁允许 reader 和 writer 按照 [ReentrantLock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\ReentrantLock.html) 的样式重新获取读取锁或写入锁。在写入线程保持的所有写入锁都已经释放后，才允许重入 reader 使用它们。

此外，writer可以获取读取锁，但反过来则不成立。在其他应用程序中，当在调用或回调那些在读取锁状态下执行读取操作的方法期间保持写入锁时，重入很有用。如果 reader 试图获取写入锁，那么将永远不会获得成功。

**锁降级**

重入还允许从写入锁降级为读取锁，其实现方式是：先获取写入锁，然后获取读取锁，最后释放写入锁。但是，从读取锁升级到写入锁是不可能的。

**锁获取的中断**

读取锁和写入锁都支持锁获取期间的中断。

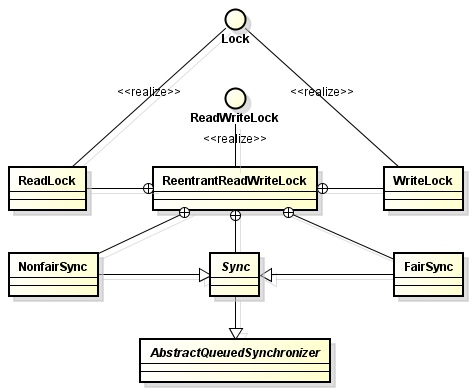
[**Condition**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html)**支持**

写入锁提供了一个 [Condition](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html) 实现，对于写入锁来说，该实现的行为与 [ReentrantLock.newCondition()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\ReentrantLock.html#newCondition()) 提供的 [Condition](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html) 实现对 [ReentrantLock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\ReentrantLock.html) 所做的行为相同。当然，此 [**Condition**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html)**只能用于写入锁**。读取锁不支持 [Condition](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html)，readLock().newCondition() 会抛出 UnsupportedOperationException。

**监测**

此类支持一些确定是保持锁还是争用锁的方法。这些方法设计用于监视系统状态，而不是同步控制。

此类行为的序列化方式与内置锁的相同：反序列化的锁处于解除锁状态，无论序列化该锁时其状态如何。



ReentrantReadWriteLock类图

## 实例化

**public** ReentrantReadWriteLock() {

**this**(**false**);

}

**public** ReentrantReadWriteLock(**boolean** fair) {

sync = (fair)? **new** FairSync() : **new** NonfairSync();

readerLock = **new** ReadLock(**this**);

writerLock = **new** WriteLock(**this**);

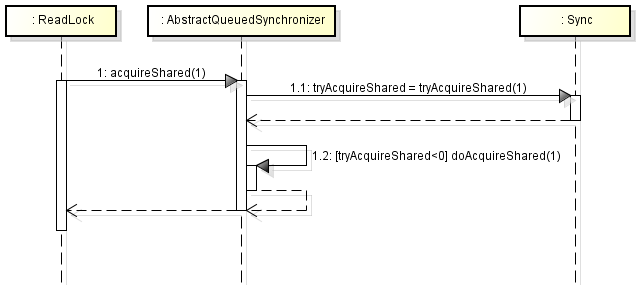
}

ReentrantReadWriteLock提供一个无参的构造器和一个有参的构造器，无参的构造器默认采用非公平模式策略；有参构造器可改变锁获得策略，为true时采用公平策略。

创建ReentrantReadWriteLock同时会实例化一个读锁(ReadLock)和一个写锁(WriteLock)。

# ReentrantReadWriteLock.ReadLock

## ReentrantReadWriteLock.ReadLock.lock()



ReentrantReadWriteLock.ReadLock.lock处理时序图

**public** **void** lock() {

sync.acquireShared(1);

}

从方法调用上来看，获得锁的处理还是交给了AbstractQueuedSynchronizer的acquireShared(1)方法，如下：

**public** **final** **void** acquireShared(**int** arg) {

**if** (tryAcquireShared(arg) < 0)// 在共享模式下尝试获得锁

doAcquireShared(arg);//

}

**protected** **final** **int** tryAcquireShared(**int** unused) {

Thread current = Thread.*currentThread*();

**int** c = getState();

// 判断当前读取锁保持计数是否不为0，且拥有锁的线程是否为当前线程，满足条件在返回-1

**if** (*exclusiveCount*(c) != 0 && getExclusiveOwnerThread() != current)

**return** -1;

// MAX\_COUNT = (1 << SHARED\_SHIFT) – 1 🡪65535,判断读取锁的数量是否达到最大数量

**if** (*sharedCount*(c) == *MAX\_COUNT*)

**throw** **new** Error("Maximum lock count exceeded");

// readerShouldBlock 判断是否需要阻塞线程

**if** (!readerShouldBlock() && compareAndSetState(c, c + *SHARED\_UNIT*)) {

HoldCounter rh = cachedHoldCounter;

**if** (rh == **null** || rh.tid != current.getId())

cachedHoldCounter = rh = readHolds.get();

rh.count++;

**return** 1;

}

**return** fullTryAcquireShared(current);

}

1. 公平锁机制和不公平锁机制对readerShouldBlock的处理是不一样的，如下：
2. 公平锁FairSync

**final** **boolean** readerShouldBlock() {

**return** hasQueuedPredecessors();

}

**final** **boolean** hasQueuedPredecessors() {

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

// 头节点的下一个节点为空或头节点的下一个节点对应的线程不是当前线程，则返回true

**return** h != t && ((s = h.next) == **null** || s.thread != Thread.*currentThread*());

}

1. 不公平锁NonfairSync

**final** **boolean** readerShouldBlock() {

**return** apparentlyFirstQueuedIsExclusive();

}

**final** **boolean** apparentlyFirstQueuedIsExclusive() {

Node h, s;

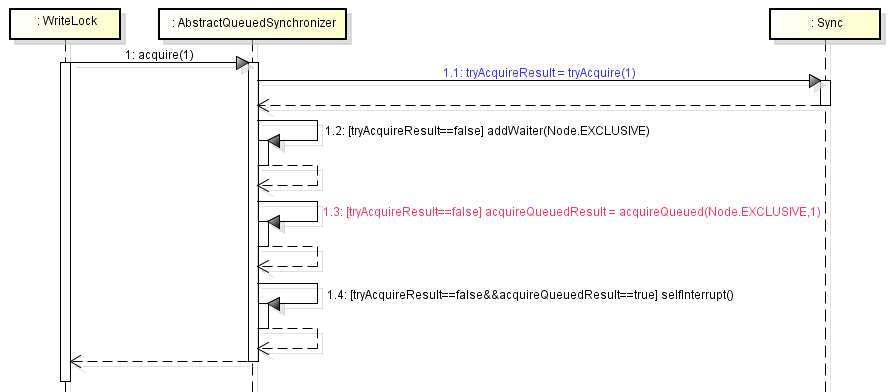
// 头节点不为空，头节点的下一个节点不为空、不是共享模式，且对应线程不为空，则返回true

**return** (h = head) != **null** && (s = h.next) != **null** && !s.isShared() && s.thread != **null**;

}

# ReentrantReadWriteLock.WriteLock

## ReentrantReadWriteLock.WriteLock.lock()



# Condition

Condition将Object 监视器方法（[wait](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Object.html#wait())、[notify](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Object.html#notify()) 和 [notifyAll](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Object.html#notifyAll())）分解成截然不同的对象，以便通过将这些对象与任意 [Lock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html) 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock替代了 synchronized 方法和语句的使用，Condition 替代了Object监视器方法的使用。

条件(也称为条件队列 或条件变量)为线程提供了一个含义，以便在某个状态条件可能为 true 的另一个线程通知它之前，一直挂起该线程（即让其“等待”）。因为访问此共享状态信息发生在不同的线程中，所以它必须受保护，因此要将某种形式的锁与该条件相关联。等待提供一个条件的主要属性是：以原子方式释放相关的锁，并挂起当前线程，就像 Object.wait 做的那样。

Condition 实例实质上被绑定到一个锁上。要为特定 [Lock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html) 实例获得 Condition 实例，请使用其 [newCondition()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html#newCondition()) 方法。

Condition实现可以提供不同于 Object 监视器方法的行为和语义，比如受保证的通知排序，或者在执行通知时不需要保持一个锁。如果某个实现提供了这样特殊的语义，则该实现必须记录这些语义。

注意，Condition 实例只是一些普通的对象，它们自身可以用作 synchronized 语句中的目标，并且可以调用自己的 [wait](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Object.html#wait(long)) 和 [notification](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Object.html#notify()) 监视器方法。获取 Condition 实例的监视器锁或者使用其监视器方法，与获取和该 Condition 相关的 [Lock](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Lock.html) 或使用其 waiting 和 signalling 方法没有什么特定的关系。为了避免混淆，建议除了在其自身的实现中之外，切勿以这种方式使用 Condition 实例。

除非另行说明，否则为任何参数传递 null 值将导致抛出 [NullPointerException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\NullPointerException.html)。

## 获取实例

ReentrantLock lock = new ReentrantLock();

Condition condition = lock.newCondition();

**final** ConditionObject newCondition() {

**return** **new** ConditionObject();

}

在newCondition的内部，其实创建了一个ConditionObject的对象，ConditionObject是AbstractQueuedSynchronizer中定义的一个成员内部类，此类实现了Condition接口。

## Condition.await()

造成当前线程在接到信号或被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())之前一直处于等待状态。

与此 Condition 相关的锁以原子方式释放，并且出于线程调度的目的，将禁用当前线程，且在发生以下四种情况之一以前，当前线程将一直处于休眠状态：

a、其他某个线程调用此Condition 的 [signal()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html#signal()) 方法，并且碰巧将当前线程选为被唤醒的线程；

b、其他某个线程调用此Condition 的 [signalAll()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\locks\Condition.html#signalAll()) 方法；

c、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程，且支持中断线程的挂起；

d、发生“虚假唤醒”

在所有情况下，在此方法可以返回当前线程之前，都必须重新获取与此条件有关的锁。在线程返回时，可以保证它保持此锁。

如果当前线程：

a、在进入此方法时已经设置了该线程的中断状态；

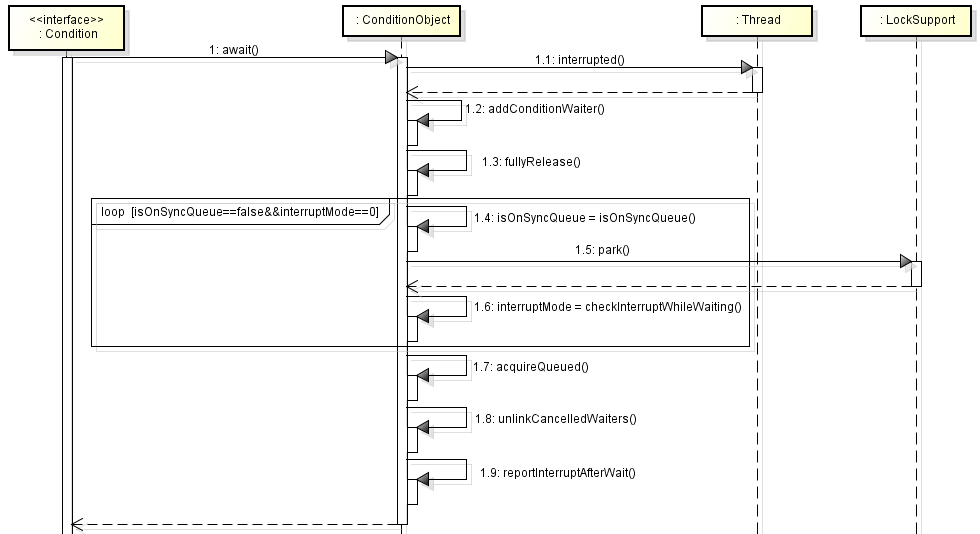
b、在支持等待和中断线程挂起时，线程被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，

则抛出 [InterruptedException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\InterruptedException.html)，并清除当前线程的中断状态。在第一种情况下，没有指定是否在释放锁之前发生中断测试。

实现注意事项

假定调用此方法时，当前线程保持了与此 Condition 有关联的锁。这取决于确定是否为这种情况以及不是时，如何对此作出响应的实现。通常，将抛出一个异常（比如 [IllegalMonitorStateException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\IllegalMonitorStateException.html)）并且该实现必须对此进行记录。

与响应某个信号而返回的普通方法相比，实现可能更喜欢响应某个中断。在这种情况下，实现必须确保信号被重定向到另一个等待线程（如果有的话）。



Lock await方法处理时序图

**public** **final** **void** await() **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

Node node = addConditionWaiter();

**int** savedState = fullyRelease(node);// 释放当前线程所占有的锁

**int** interruptMode = 0;

**while** (!isOnSyncQueue(node)) {

LockSupport.*park*(**this**);// 挂起线程

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)// 判断当前线程是否被中断，允许阻塞被中断

**break**;

}

**if** (acquireQueued(node, savedState) && interruptMode != *THROW\_IE*)// 被中断或被唤醒都需要将锁拿回来

interruptMode = *REINTERRUPT*;

**if** (node.nextWaiter != **null**) // clean up if cancelled

unlinkCancelledWaiters();

**if** (interruptMode != 0)

reportInterruptAfterWait(interruptMode);// 挂起线程或抛出异常

}

1、如果当前线程已经被中断，则清空中断状态位，抛出InterruptedException异常(抛出异常之后中断状态为由回复为true).

2、添加一个新的等待节点到等待队列中，其中可能会踢出等待队列中已取消的等待节点。见[ConditionObject.addConditionWaiter](#_ConditionObject.addConditionWaiter)。

3、判断当前线程是否占有锁，如果未占有锁则抛出IllegalMonitorStateException异常；否则释放当前线程所占有的锁。

4、isOnSyncQueue方法会判断当前节点的状态，逻辑如下：

a、如果当前节点状态为Node.CONDITION或前一个节点为空，则返回false。

b、如果条件a不满足，且当前节点下一个节点(node.next)不为空则返回true。

// 未完，待续…

4、acquireQueued方法采用不间断的方式去获得锁，返回当前线程是否被中断过。如果当前线程被中断过且中断模式不为THROW\_IE(-1)则将中断模式重置为REINTERRUPT(1).

5、如果当前节点存在下一个节点，则迭代同步等待队列，清空已取消节点。

6、根据不同的中断模式返回不同的结果，如果中断模式是THROW\_IE则抛出InterruptedException异常，如果中断模式是REINTERRUPT，则重新interrupt当前线程。

## ConditionObject.addConditionWaiter

**private** Node addConditionWaiter() {

Node t = lastWaiter;

**if** (t != **null** && t.waitStatus != Node.*CONDITION*) {

unlinkCancelledWaiters();

t = lastWaiter;

}

Node node = **new** Node(Thread.*currentThread*(), Node.*CONDITION*);

**if** (t == **null**)

firstWaiter = node;

**else**

t.nextWaiter = node;

lastWaiter = node;

**return** node;

}

如果等待队列中最后一个节点的状态不为Node.CONDITION，那么则调用unlinkCancelledWaiters()方法清除等待队列中已取消的节点。最后将当前节点插入到队列的末端。

**private** **void** unlinkCancelledWaiters() {

Node t = firstWaiter;

Node trail = **null**;

**while** (t != **null**) {

Node next = t.nextWaiter;

**if** (t.waitStatus != Node.*CONDITION*) {// 节点已经被取消

t.nextWaiter = **null**;// 断开当前节点也下一个节点的联系，相当于将当前节点踢出等待队列,有利于GC

**if** (trail == **null**)

firstWaiter = next;

**else**

trail.nextWaiter = next;

**if** (next == **null**)

lastWaiter = trail;

} **else**

trail = t;

t = next;

}

}

## Condition.signal()

唤醒一个等待线程。

如果所有的线程都在等待此条件，则选择其中的一个唤醒。在从 await 返回之前，该线程必须重新获取锁。

public final void signal() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

if (first != null)

doSignal(first);

}

**private** **void** doSignal(Node first) {

**do** {

**if** ( (firstWaiter = first.nextWaiter) == **null**)// 没有下一个等待节点,需要将lastWaiter标识[最后一个节点]为null

lastWaiter = **null**;

first.nextWaiter = **null**;// 将下一个当代节点标识为null

} **while** (!transferForSignal(first) && (first = firstWaiter) != **null**);// 如果节点取消，且first节点的存在下一个节点。

}

**final** **boolean** transferForSignal(Node node) {

**if** (!*compareAndSetWaitStatus*(node, Node.*CONDITION*, 0))// 如果节点已经取消

**return** **false**;

Node p = enq(node);

**int** ws = p.waitStatus;

**if** (ws > 0 || !*compareAndSetWaitStatus*(p, ws, Node.*SIGNAL*))

LockSupport.*unpark*(node.thread);// 唤醒线程

**return** **true**;

}

# LockSupport

用来创建锁和其他同步类的基本线程阻塞原语。

此类以及每个使用它的线程与一个许可关联（从 Semaphore 类的意义上说）。如果该许可可用，并且可在进程中使用，则调用 park 将立即返回；否则可能阻塞。如果许可尚不可用，则可以调用 unpark 使其可用。（**但与 Semaphore 不同的是，许可不能累积，并且最多只能有一个许可**。）

park和 unpark 方法提供了阻塞和解除阻塞线程的有效方法，并且不会遇到导致过时方法 Thread.suspend 和 Thread.resume 因为以下目的变得不可用的问题：由于许可的存在，调用 park 的线程和另一个试图将其 unpark 的线程之间的竞争将保持活性。此外，如果调用者线程被中断，并且支持超时，则 park 将返回。park 方法还可以在其他任何时间“毫无理由”地返回，因此通常必须在重新检查返回条件的循环里调用此方法。从这个意义上说，park 是“忙碌等待”的一种优化，它不会浪费这么多的时间进行自旋，但是必须将它与 unpark 配对使用才更高效。

三种形式的 park 还各自支持一个 blocker 对象参数。此对象在线程受阻塞时被记录，以允许监视工具和诊断工具确定线程受阻塞的原因。（这样的工具可以使用方法 getBlocker(java.lang.Thread) 访问 blocker。）建议最好使用这些形式，而不是不带此参数的原始形式。在锁实现中提供的作为blocker 的普通参数是 this。

这些方法被设计用来作为创建高级同步实用工具的工具，对于大多数并发控制应用程序而言，它们本身并不是很有用。park 方法仅设计用于以下形式的构造：

while (!canProceed()) { ... LockSupport.park(this); }

在这里，在调用 park 之前，canProceed 和其他任何动作都不会锁定或阻塞。因为每个线程只与一个许可关联，park 的任何中间使用都可能干扰其预期效果。