JAVA原子性操作：

我们看下面一段代码

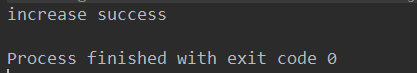
public void increase() {  
 int prevCount = counter;   
 counter = prevCount + 1;  
 if (counter - prevCount == 1) {  
 System.*out*.println("increase success");  
 } else {  
 System.*out*.println("increase failed");  
 }  
}

这是一段简单的给counter + 1 的操作，并且判断counter相比之前操作是否被加一了

下面我们来看这段代码是否存在问题？

首先获取counter值，赋值给prevCount，然后prevCount加一赋给counter。

在程序里面运行一下：



看上去没什么问题

下面我们给他加上一些限制

public void increase(long delayed) {  
 try {  
 int prevCount = counter;  
 counter = prevCount + 1;

//假设这里休眠了几秒  
 Thread.*sleep*(delayed);  
 if (counter - prevCount == 1) {  
 System.*out*.println("increase success");  
 } else {  
 System.*out*.println("increase failed");  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

中间休眠了几秒

那么问题可想而知，多线程情况下，counter在某个线程休眠时间内被修改了值。那么就导致counter前后不一致，这个increase方法就表示操作失败了

这里我们可能会考虑使用synchronized来就可以满足该操作的一致性了

就好像这样的操作：

public synchronized void increase(long delayed)

又好比这样：

public void increase(long delayed) {  
 synchronized (this) {  
 }  
}

鉴于synchronized的特性，我们可能会遇到这样的情况，线程ABCDE相同情况存在竞争该对象，结果ABCDE都休眠了3s，导致ABCDE5个线程执行完成需要15s时间，这个时间太久了，如果不用synchronized该怎么解决这个问题呢，我们可以使用循环[这里叫自旋锁]，来保证数据被正确+1 了：

代码如下：

public void increase(long delayed) {  
 // 自旋锁  
 for (;;) {  
 int prevCount = counter;  
 counter = prevCount + 1;  
 //假设这里休眠了几秒  
 //...  
 if (counter - prevCount == 1) {  
 System.*out*.println("increase success");  
 break;  
 } else {  
 System.*out*.println("increase failed");  
 }  
 }  
}

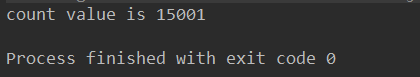
使用for循环，如果这次操作失败，那么重新执行该方法

使用了自旋锁，大大降低了出错的概率，是否这个代码就可以满足下面的需求了呢，我们测试一下

WorkThread workThread = new WorkThread(myObject, 2000);  
WorkThread workThread1 = new WorkThread(myObject, 1000);  
WorkThread workThread2 = new WorkThread(myObject, 1000);  
try {  
 workThread.start();  
 workThread1.start();  
 workThread2.start();  
 workThread.join();  
 workThread1.join();  
 workThread2.join();  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}  
System.*out*.println("count value is " + myObject.counter);

启动3个线程，每个线程调用5000次该自增方法：

结果如下：



确实解决了我们的问题。使用自旋锁for来实现保证数据一致性【相比synchronized，确实省去线程等待与竞争不确定的问题】

上面这个方法，Java提供了相关的原子性操作类：

java.util.concurrent.atomic

上面的代码我们可以更好的写下这个方法：

private AtomicInteger counter = new AtomicInteger(1);  
counter.incrementAndGet();

atomic提供了很多原子性操作，使用unSafe类直接操作对象地址内存，确保中间过程不会被打断，保证修改的一致性

比如我们需要比较与修改一个数：

如果counter之前是5，那么我们把它修改为10

counter.compareAndSet(5,10);

这个操作也是原子性的，在比较内存值是5之后，中间其他线程无法修改该值，等待赋值为10之后才算完成该操作，方法返回boolean表示是否比较设置成功

我们把compareAndSet这个方法称为JAVA的 CAS操作，原子性

前面简单介绍了CAS操作，原理使用unSafe直接比较修改内存，下面我们看下JAVA的AQS锁的实现

基于Java原子性操作compareAndSet[CAS]的AbstractQueuedSynchronizer [AQS]

[ 提供一个框架来实现阻塞锁和相关同步器（信号量，事件，等），这个框架依赖于先入先出等待队列（*FIFO*），JAVA里面所有线程安全的数据定义，集合，都是基于此AQS原理 + 自旋锁 + CAS操作来确保线程同步 ]

如何使用与实现：

AbstractQueuedSynchronizer一般作为一个自定义锁的内部实现类，比如CountDownLatch，ReentrantLock，CyclicBarrier，Semaphore 等，详细讲解AbstractQueuedSynchronizer之后讲仔细分析其源码的实现

实现方式：

AbstractQueuedSynchronizer提供了共享锁与独占锁

重要相关方法：

final void setState(int newState) state是一个int值，用于子类当成任何标志，实现tryAcquire，tryRelease，tryAcquireShared，tryReleaseShared方法的判断值锁

|  |  |  |
| --- | --- | --- |
|  | 独占方法 | 共享方法 |
| 公开方法 | void acquire(int arg) 尝试获取锁，如果被其他占用那么进入等待，等待过程不可以被中断（当前Thread#Interrupt） | void acquireShared(int arg) //尝试获取共享锁，如果被其他占用那么进入等待，等待过程不可以被中断（当前Thread#Interrupt） |
| 公开方法 | void acquireInterruptibly(int arg) 尝试获取锁，如果被其他占用那么进入等待，等待过程可以被中断（当前Thread#Interrupt） | void acquireSharedInterruptibly(int arg) //尝试获取共享锁，如果被其他占用那么进入等待，等待过程可以被中断（当前Thread#Interrupt） |
| 公开方法 | boolean tryAcquireNanos(int arg, long nanosTimeout) 尝试获取锁等待时间nanosTimeout，如果等待超时，线程将返回继续，等待过程可以被中断（当前Thread#Interrupt） | boolean tryAcquireSharedNanos(int arg, long nanosTimeout) //尝试获取共享锁等待时间nanosTimeout，如果等待超时，线程将返回继续，等待过程可以被中断（当前Thread#Interrupt） |
| 需要子类实现 | boolean tryAcquire(int arg) //IMPL内部类实现，表示是否获取锁成功，返回true，表示获取成功，程序正常执行，返回false表示获取失败，进入等待状态（LockSupport.park） | int tryAcquireShared(int arg) //IMPL内部类实现 ，返回>0表示获取成功，返回<0 表示线程阻塞，等待线程被释放 |
| 公开方法 | boolean release(int arg) 释放线程占用锁 | boolean releaseShared(int arg) 释放该共享锁，相当于UnLock,释放成功将当前共享锁线程释放 |
| 需要子类实现 | boolean tryRelease(int arg) //IMPL内部类实现，表示释放占用锁，，返回true表示释放该锁，AQS尝试解锁阻塞的线程(LockSupport.unPark) | boolean tryReleaseShared(int arg) //IMPL内部类实现，表示释放共享锁，返回true表示释放该锁，AQS尝试解锁阻塞的线程(LockSupport.unPark) |

总结，内部类需要实现下面4 + 2方法：

boolean tryAcquire(int arg) // 尝试获取独占锁，返回false将阻塞

boolean tryRelease(int arg) // 尝试解锁当前线程，返回true将其他队列线程尝试唤醒继续获取独占锁

boolean tryReleaseShared(int arg) // 尝试解锁当前线程，返回true将其他队列线程尝试唤醒继续获取独占锁

int tryAcquireShared(int arg) //尝试获取共享锁，返回false将阻塞

setExclusiveOwnerThread(Thread) //设置独占线程

boolean isHeldExclusively() //是否是独占线程，一般固定实现为（return getExclusiveOwnerThread() == Thread.currentThread();）

AQS公开方法：

|  |  |
| --- | --- |
| hasQueuedPredecessors() | 返回相比于当前线程是否存在其他线程在队列前面等待，用于判断是否需要独占锁，查询前置节点线程（常用于公平锁） |
| boolean hasQueuedThreads() | 是否队列里面有节点在等待 |
| Thread getFirstQueuedThread() | 返回队列里面第一个的线程 |
| Collection<Thread> getExclusiveQueuedThreads() | 返回等待队列里面的非共享节点的线程 |
| Collection<Thread> getSharedQueuedThreads() | 返回队列里面所有共享节点的线程 |
| Collection<Thread> getQueuedThreads() | 获取队列里面所有节点的线程 |
| boolean isQueued(Thread thread) | 当前线程是否在等待队列里面 |
| int getQueueLength() | 获取队列长度 |
| boolean hasContended() | 头节点是否为null |
| Collection<Thread> getWaitingThreads(ConditionObject condition) | 获取condition的等待线程 |
| int getWaitQueueLength(ConditionObject condition) | 获取condition的等待线程长度 |

独占锁源码解析：

获取锁从以下开始：

AbstractQueuedSynchronizer#acquire

public final void acquire(int arg) {  
 if (!tryAcquire(arg) &&  
 acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 *selfInterrupt*();  
}

方法分4步：

1. 调用子类的覆盖tryAcquire方法

1.1 如果tryAcquire返回ture，表示获取锁成功，程序正常执行，线程不会被阻塞

1.2 如果tryAcquire返回false那么执行acquireQueued(addWaiter(Node.EXCLUSIVE), arg)方法

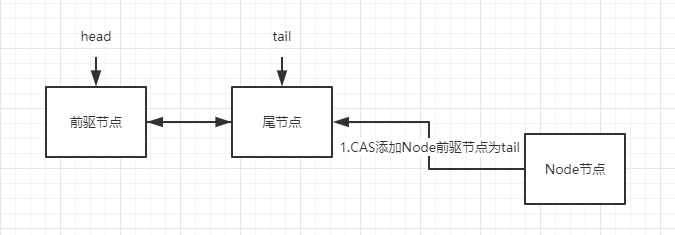
2. 执行addWaiter(Node.EXCLUSIVE)方法

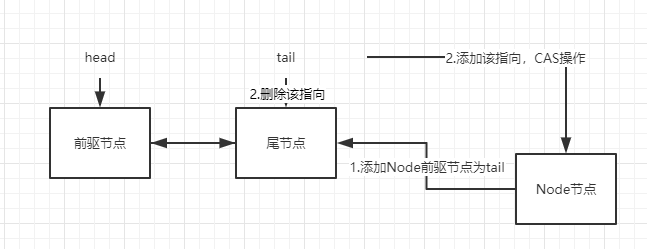
private Node addWaiter(Node mode) {  
 Node node = new Node(mode);

// 0 需要循环保证自旋锁数据CAS成功  
 for (;;) {

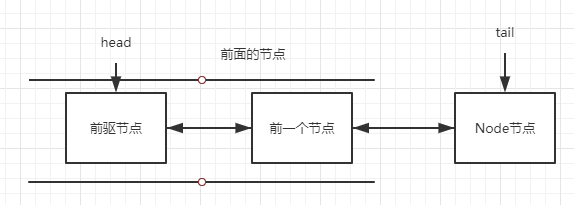
// 1 获取尾部node  
 Node oldTail = tail;  
 if (oldTail != null) {  
 *U*.putObject(node, Node.*PREV*, oldTail);  
 if (compareAndSetTail(oldTail, node)) {  
 oldTail.next = node;  
 return node;  
 }  
 } else {  
 initializeSyncQueue();  
 }  
 }  
}

2.1 new一个Node节点，获取队列尾部node节点，如果尾部node节点不是null，将当前节点设置为尾部节点，使用CAS操作，new的节点的前节点为上一个尾部节点，设置当前节点为尾部节点，如果CAS成功那么返回，如果CAS失败，表示可能当前线程被其他线程修改了尾部Node，所以需要执行注释0，使用自旋锁 保证CAS操作成功，设置尾节点成功





结果：



注意：这里不需要关注head节点

2.2 如果尾部节点是null，表示当前还没有生成一个队列，这个时候调用initializeSyncQueue生成一个队列[链表]，实现如下：

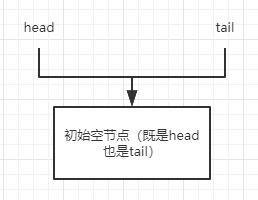
private final void initializeSyncQueue() {  
 Node h;  
 if (*U*.compareAndSwapObject(this, *HEAD*, null, (h = new Node())))  
 tail = h;  
}

创建一个节点并设置成头节点，成功通知作为tail尾部节点，然后使用for循环再次将生成节点加入上一次for创建的尾节点生成链表

执行成功将：

1. 当前Node加入之前链表尾部，如果之前链表不为null

2. 创建一个空Node的头链表，当前Node加入链表（初始化的时候head表示newNode，tail指向head，然后第二次for循环tail指向新node。连接成链表）



3. 调用acquireQueue方法：线程等待核心实现

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

// 1变量表示线程是否被中断  
 boolean interrupted = false;  
 for (;;) {

// 2.获取前驱节点  
 final Node p = node.predecessor();

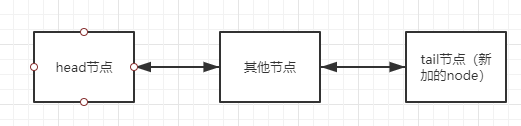
// 3.如果前驱节点是head节点，那么释放前驱节点锁，前驱节点调用tryAcquire尝试获取锁，返回false，当前节点设置为head节点  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);  
 p.next = null; // help GC  
 return interrupted;  
 }

// 4.如果不是前驱结点，或者第一个节点获取锁失败，判断是否需要执行线程锁

// 5. *shouldParkAfterFailedAcquire方法*  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())

// 如果当前线程被唤醒了，那么判断是否中断，自选返回执行是否中断

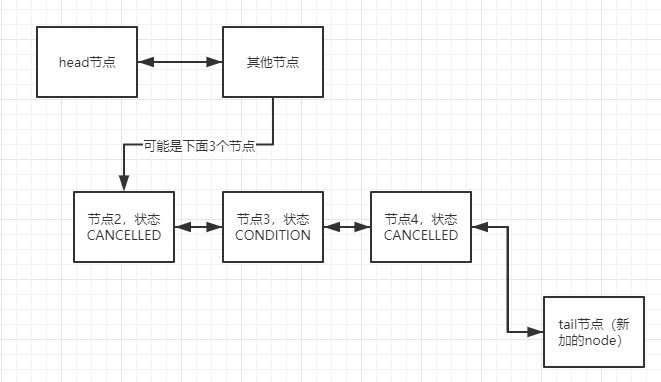
// 如果是响应可中断，那么就应该直接抛出异常（唯一与doAcquireInterruptibly的区别）  
 interrupted = true;  
 }  
 } catch (Throwable t) {  
 cancelAcquire(node);  
 throw t;  
 }  
}



获取tail节点（新加的node节点）的前置节点

1. 可能是head节点，如果是head节点，那么尝试唤醒head节点

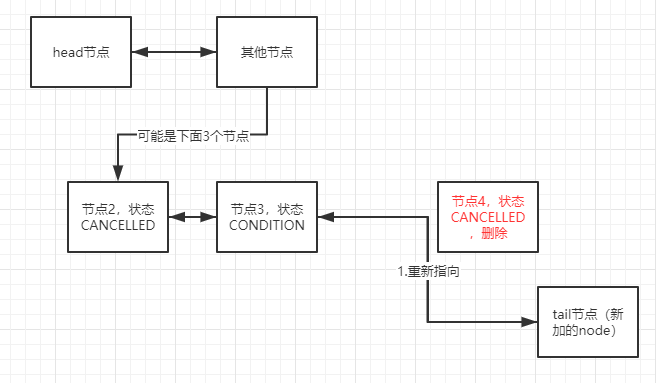
2. 可能是其他节点，如果是其他节点，那么当前tail节点需要做以下事情：



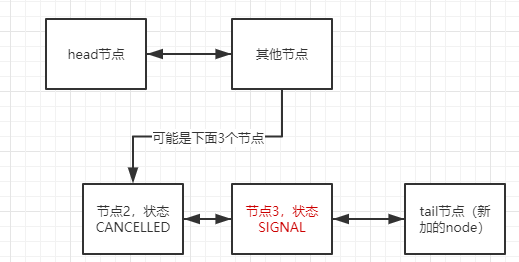
5. shouldParkAfterFailedAcquire将执行几个操作，如下：

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
 int ws = pred.waitStatus;  
 if (ws == Node.*SIGNAL*)  
 return true; // 如果前驱节点是SIGNAL,返回那么就会执行Park锁当前线程  
 if (ws > 0) { // 如果是cancelled那么重新刷新链表，删除这个状态的  
 do {  
 node.prev = pred = pred.prev;  
 } while (pred.waitStatus > 0);  
 pred.next = node;  
 } else { // 否则将其他状态设置为signal，3里面的代码使用自旋锁重新走这个方法，锁住当前线程  
 pred.compareAndSetWaitStatus(ws, Node.*SIGNAL*);  
 }  
 return false;  
}

这个方法就是排除cancel的节点，并自旋设置前置节点为signal状态，然后自身执行parkAndCheckInterrupt锁住当前线程，等待前置节点Signal状态下才会通知当前节点（只有在前置节点是signal状态才能通知当前节点解锁）



跳出循环，[再次for执行判断节点3是否是head节点，是则尝试唤醒](#获取解锁head节点)，否则设置节点3为signal状态，此时这样：

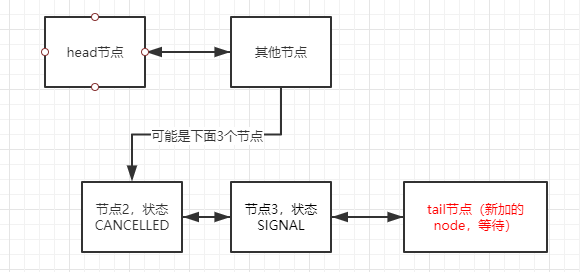


将其他节点重新排序，删除其他节点内部状态为cancel状态的节点，第二点将前置节点状态设置为singal，这样才能确保前置节点在这个状态一定能唤醒自己，然后使用自旋锁确保操作完整，最后当前节点线程阻塞，如果节点3不是head节点而且是Signal状态，执行下面

private final boolean parkAndCheckInterrupt() {  
 LockSupport.*park*(this);  
 return Thread.*interrupted*();  
}

返回是否中断，只有前置解锁，当前获取锁之后才能执行最初的selfInterrupt打断该线程，所以之前的acquire方法是不会响应线程中断，只有唤醒才能执行中断，上面就是独占原理的实现

当前tail节点等待：



acquireInterruptibly可响应中断方法，唯一区别就是解锁的线程直接抛出异常，响应中断，在3里面黄色部分介绍

private void doAcquireInterruptibly(int arg)  
 throws InterruptedException {  
 final Node node = addWaiter(Node.*EXCLUSIVE*);  
 try {  
 for (;;) {  
 final Node p = node.predecessor();  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);  
 p.next = null; // help GC  
 return;  
 }

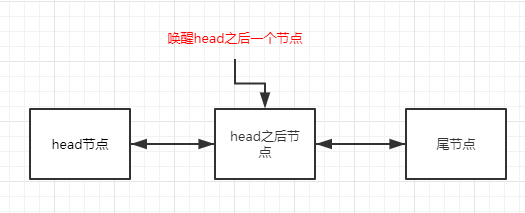
// parkAndCheckInterrupt在thread#interrupt下面中断，直接抛出异常  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 throw new InterruptedException();  
 }  
 } catch (Throwable t) {  
 cancelAcquire(node);  
 throw t;  
 }  
}

[唤醒acquireInterruptibly方法，线程唤醒抛出interrupt异常](#独占可中断)

解锁源码分析：

public final boolean release(int arg) {  
 if (tryRelease(arg)) {  
 Node h = head;  
 if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);  
 return true;  
 }  
 return false;  
}

尝试解锁tryRelease，如果解锁成功，那么唤醒head头部节点线程，如果头部节点状态不是0，那么 将head节点唤醒，此时前面的方法：（acquireQueued）将唤醒前驱节点线程，for自旋一次设置head为下一个并且走出循环

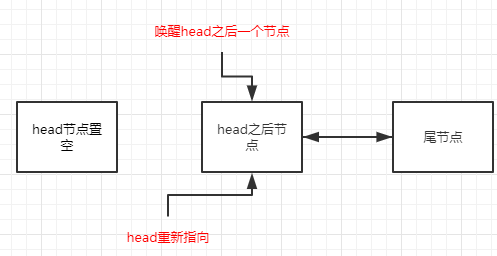


之后一个节点，假设节点2解锁，[再走上面for自旋锁流程](#解锁之后自旋走唤醒)，[解锁之后自旋设置head节点](#解锁之后自旋尝试获取锁)

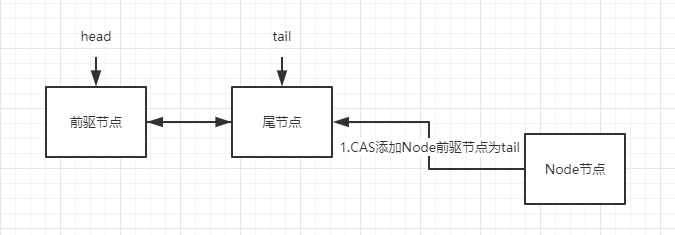
总流程图：

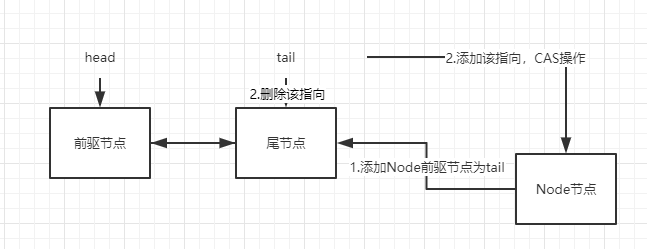
添加：

如果前驱节点是head节点，尝试唤醒成功：

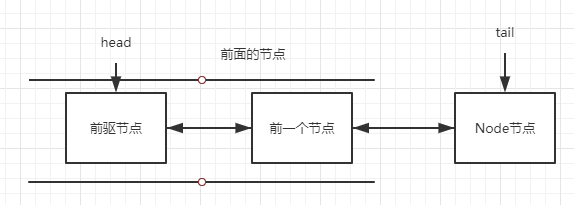


否则添加

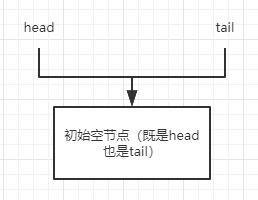




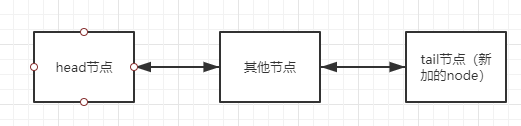
结果：

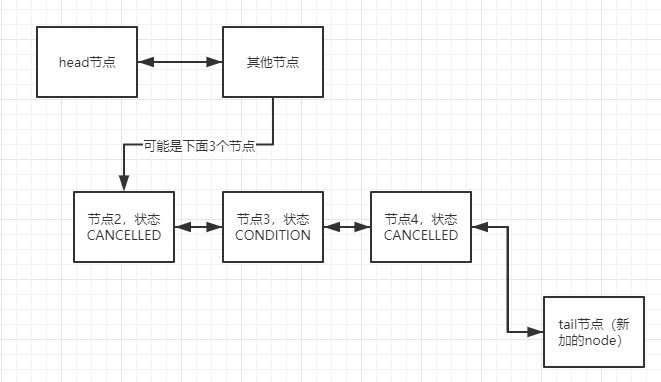


创建空head与node：

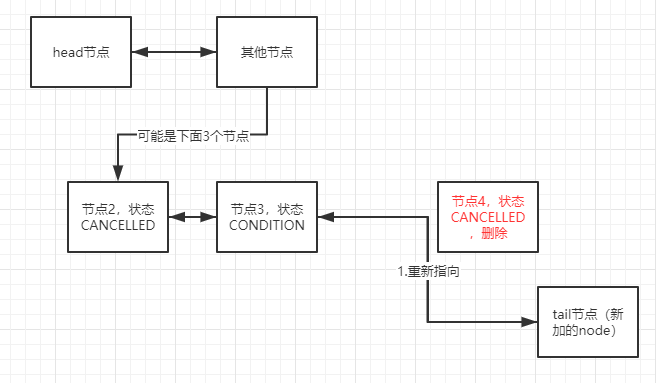


重新排序：

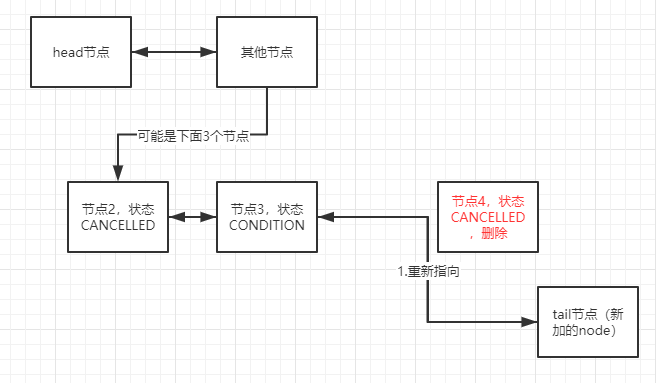




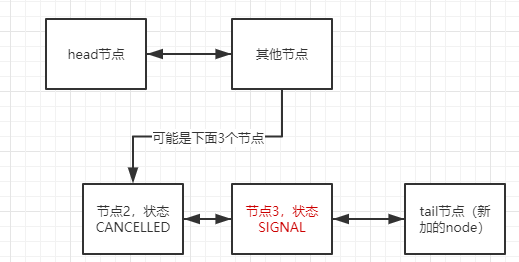
更新前驱节点状态



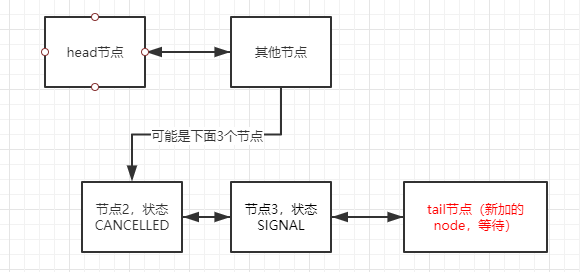
删除无效节点：



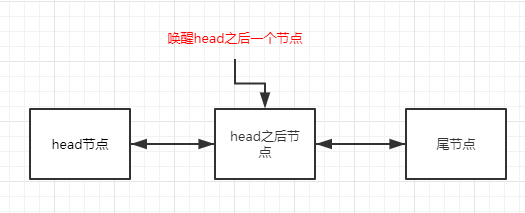
跳出循环，[再次for执行判断节点3是否是head节点，是则尝试唤醒](#获取解锁head节点)，否则设置节点3为signal状态，此时这样：



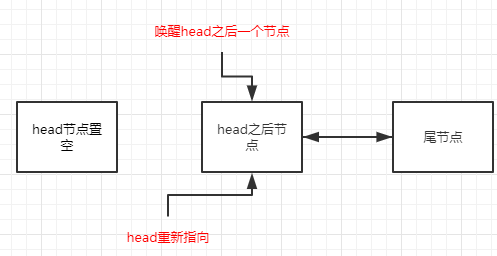
阻塞：



唤醒：



最前节点唤醒，执行剔除head节点：



上面我们分析了独占锁的实现，注意我们这里队列 休眠和唤醒都是线程级别的，所以在前面的acquireQueued使用for来确保自身能够在被正确唤醒的时候return解锁，独占锁在解锁的时候只会解锁唤醒head节点，后面节点继续等待下一次唤醒head指向，传递如下：

当前获取锁线程A执行 release方法，通知唤醒head之后一个节点的线程（unparkSuccessor(Node head)），当前线程退出节点，head指向head之后节点，线程A唤醒head所占线程B，head指向下一个，线程B解锁阻塞进入运行状态，然后循环，B再次解锁，head指向C，C进入运行状态，所以这就独占线程的问题，需要成对 获取锁 与 解锁， 如果不解锁，那么head等待就造成了 一直等待现象

[当所有阻塞都解锁之后，那么如下，head = tail节点，等于初始化节点了](#初始化节点)

各节点变更时机：

Head节点是在阻塞节点线程解锁之后重新指向该解锁节点

Tail是在调用addWaiter之后修改指向add的waiter

下面我们分析共享锁：

关于共享锁，首先，我们实现tryAcquireShared返回是否能被共享获取，如果不行，流程和前面一样，节点生成与阻塞

解锁，在共享锁解锁的时候，代码如下：

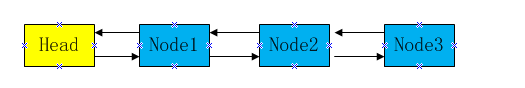
for (;;) {  
 Node h = head;  
 if (h != null && h != tail) {  
 int ws = h.waitStatus;  
 if (ws == Node.*SIGNAL*) {  
 if (!h.compareAndSetWaitStatus(Node.*SIGNAL*, 0))  
 continue; // loop to recheck cases  
 unparkSuccessor(h);  
 }  
 else if (ws == 0 &&  
 !h.compareAndSetWaitStatus(0, Node.*PROPAGATE*))  
 continue; // loop on failed CAS  
 }  
 if (h == head) // loop if head changed  
 break;  
}

这里调用unparkSuccessor解锁下一个线程B，下一个线程B被唤醒，acquireShared方法唤醒放通进入自旋，执行下面方法

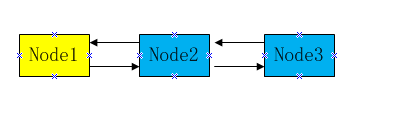
private void setHeadAndPropagate(Node node, int propagate) {  
 Node h = head; // Record old head for check below  
 setHead(node);  
 if (propagate > 0 || h == null || h.waitStatus < 0 ||  
 (h = head) == null || h.waitStatus < 0) {  
 Node s = node.next;  
 if (s == null || s.isShared())  
 doReleaseShared();  
 }  
}

头指针更换B节点，判断下个节点是否是shared状态，如果是继续解锁B下一个节点C，这会让下个节点C解锁，

程序执行C的acquireShared方法唤醒放通进入自旋。也回调设置当前节点为head，再次唤醒C的下一个节点



Node1调用tryAcquireShared成功后，更换了头节点



Node1变成了头节点然后调用unparkSuccessor()方法唤醒了Node2,Node2继续设置head节点判断节点Node3是否是shared，是则唤醒Node3，这就产生了链式唤醒，这便是共享锁的唤醒方式

区别：

独占锁，unlock只会唤醒head下一个节点，成功则head指向该下一个节点

共享锁，unlock解锁会链式唤醒head之后节点，如果成功，那么head将链式指向，直到最后一个唤醒的节点

ReentrantLock独占锁

ReentrantLock公平锁与非公平锁，公平锁确保前驱节点顺序唤醒，否则排队阻塞

非公平锁在获取锁的时候，优先尝试获取锁并设置为当前线程

final void lock() {

//先尝试获取  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}

final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();

//再次尝试获取  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }

// 重入锁  
 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;  
}

公平锁

final void lock() {  
 acquire(1);  
}  
  
*/\*\*  
 \* Fair version of tryAcquire. Don't grant access unless  
 \* recursive call or no waiters or is first.  
 \*/*// Android-removed: @ReservedStackAccess from OpenJDK 9, not available on Android.  
// @ReservedStackAccess  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {

// 如果没有前驱节点（head下一个节点）并且获取锁成功，那么设置当前线程为占用锁的线程  
 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;  
}

Semaphore：

使用共享锁，可以使用公平与非公平方式获取锁：

final int nonfairTryAcquireShared(int acquires) {  
 for (;;) {  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 if (hasQueuedPredecessors())  
 return -1;  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

信号量相当于CountDownLatch的总数量，每个线程可以获取一定数据量acquires值，如果不足，那么阻塞，等待下次释放锁重新尝试获取一定数量的信号量

public void acquireUninterruptibly(int permits) {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.acquireShared(permits);  
}

获取permits个信号量，如果不足，那么等待，否则获取成功

释放锁的时候，采取共享锁，会依次唤醒后继节点，每个节点尝试获取锁，直到某个节点失败

CountDownLatch

Sync(int count) {  
 setState(count);  
}

protected int tryAcquireShared(int acquires) {  
 return (getState() == 0) ? 1 : -1;  
}  
  
protected boolean tryReleaseShared(int releases) {  
 // Decrement count; signal when transition to zero  
 for (;;) {  
 int c = getState();  
 if (c == 0)  
 return false;  
 int nextc = c - 1;  
 if (compareAndSetState(c, nextc))  
 return nextc == 0;  
 }  
}

构造函数，初始化一个数字，每次调用lock的时候，执行sharedAcquire，判断是否是0，否则加入链表等待，在某个线程都执行unlock回调tryReleaseShared之后链表依次对节点尝试sharedAcquire，此时，如果线程释放了unlock次数为初始化数字的次数，那么之前等待节点都被依次唤醒

举个例子，初始化构造函数传入5，线程Main执行lock，但是被阻塞。线程ABCDE有5个线程，线程A执行依次release方法，5减一成四，此时Main尝试唤醒，但是不是0，唤醒失败，继续等待，BCDE都release，此时Main的lock成功，Main线程唤醒继续执行。

只有在构造函数传入的数字由其他线程改变成0的时候，之前尝试获取lock而等待的线程都会唤醒

CyclicBarrier

private int dowait(boolean timed, long nanos)  
 throws InterruptedException, BrokenBarrierException,  
 TimeoutException {  
 final ReentrantLock lock = this.lock;

// 线程锁  
 lock.lock();  
 try {

// 数量自减  
 int index = --count;  
 if (index == 0) { // 如果已经达到阻塞个数那么执行runnable  
 boolean ranAction = false;  
 try {  
 final Runnable command = barrierCommand;  
 if (command != null)  
 command.run();  
 ranAction = true;  
 return 0;  
 } finally {  
 if (!ranAction)  
 breakBarrier();  
 }  
 }  
  
 // loop until tripped, broken, interrupted, or timed out  
 for (;;) {  
 try {  
 if (!timed)  
 trip.await();  
 else if (nanos > 0L)  
 nanos = trip.awaitNanos(nanos);  
 } catch (InterruptedException ie) {  
 }  
 }  
 } finally {  
 lock.unlock();  
 }  
}

同步栅栏，当有一定数量线程等待的时候，就会执行一个runnable方法。原理是使用线程等待+锁数量并执行runnable

注意一点：

Condition执行await会释放占用锁（Release当前锁），所以后面线程调用lock可以获取该锁

查询可以得知，await方法和Object的wait（必须是Owner线程才能指向wait）都是释放锁占用，但是这里释放占用不等同于我们AQS主动调用unlock方法，unlock方法是释放阻塞，而wait/await方法是释放占用，比如synchronized锁住object对象，这个object方法里面调用wait只是等待，此时别的Thread就可以获取该object对象

Condition原理，await调用的时候内部维护了链表，用于临时释放该锁并锁住当前线程，后续被signal的时候会加入AQS链表，进入独占锁的链表