**并发编程**

# ReentrantReadWriteLock

## 概述

　ReentrantReadWriteLock是Lock的另一种实现方式，我们已经知道了ReentrantLock是一个排他锁，同一时间只允许一个线程访问，而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。相对于排他锁，提高了并发性。在实际应用中，大部分情况下对共享数据（如缓存）的访问都是读操作远多于写操作，这时ReentrantReadWriteLock能够提供比排他锁更好的并发性和吞吐量。

　　读写锁内部维护了两个锁，一个用于读操作，一个用于写操作。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。

　　ReentrantReadWriteLock支持以下功能：

　　　　1）支持公平和非公平的获取锁的方式；

　　　　2）支持可重入。读线程在获取了读锁后还可以获取读锁；写线程在获取了写锁之后既可以再次获取写锁又可以获取读锁；

　　　　3）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允许的；

　　　　4）读取锁和写入锁都支持锁获取期间的中断；

　　　　5）Condition支持。仅写入锁提供了一个 Conditon 实现；读取锁不支持 Conditon ，readLock().newCondition() 会抛出 UnsupportedOperationException。

# ReentrantLock

## 例子

|  |
| --- |
| ReentrantLock **reentrantLock** = **new** ReentrantLock();  **public void** test2() {  **for** (**int** i = 0; i < 3; i++) {  **reentrantLock**.lock();  }  **// 加锁三次，解锁两次，最终还是锁着的，另一个线程不会获取到锁**  **for** (**int** i = 0; i < 2; i++) {  **try** {   } **finally** {  **reentrantLock**.unlock();  }  } }  @Test **public void** test1() {  Thread t1 = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  test2();  }  });   Thread t2 = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  test2();  System.***out***.println(**"================="**);  }  });   t1.start();  t2.start(); } |

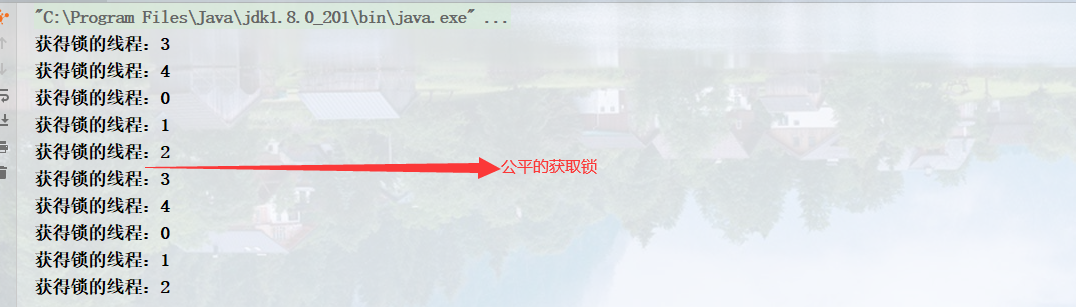
* ReentrantLock和synchronized都是独占锁,只允许线程互斥的访问临界区。但是实现上两者不同:synchronized加锁解锁的过程是隐式的,用户不用手动操作,优点是操作简单，但显得不够灵活。一般并发场景使用synchronized的就够了；ReentrantLock需要手动加锁和解锁,且解锁的操作尽量要放在finally代码块中,保证线程正确释放锁。ReentrantLock操作较为复杂，但是因为可以手动控制加锁和解锁过程,在复杂的并发场景中能派上用场。
* ReentrantLock和synchronized都是可重入的。synchronized因为可重入因此可以放在被递归执行的方法上,且不用担心线程最后能否正确释放锁；而ReentrantLock在重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样，否则可能导致其他线程无法获得该锁。

## ReentrantLock相比synchronized的额外功能

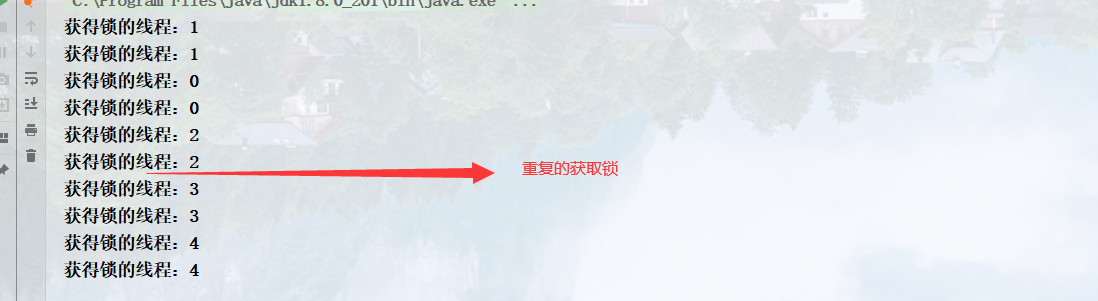
### ReentrantLock可以实现公平锁。

公平锁是指当锁可用时,在锁上等待时间最长的线程将获得锁的使用权。而非公平锁则随机分配这种使用权。和synchronized一样，默认的ReentrantLock实现是非公平锁,因为相比公平锁，非公平锁性能更好。当然公平锁能防止饥饿,某些情况下也很有用。在创建ReentrantLock的时候通过传进参数true创建公平锁,如果传入的是false或没传参数则创建的是非公平锁。

|  |
| --- |
| **static ReentrantLock *reentrantLock* = new ReentrantLock(true);   public static void main(String[] args) throws InterruptedException {   for(int i=0;i<5;i++){  new Thread(new ThreadDemo(i)).start();  }   }   static class ThreadDemo implements Runnable {  Integer id;   public ThreadDemo(Integer id) {  this.id = id;  }   @Override  public void run() {  try {  TimeUnit.*MILLISECONDS*.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  for(int i = 0;i < 2; i++){  *reentrantLock*.lock();  System.*out*.println("获得锁的线程："+id);  *reentrantLock*.unlock();  }  }  }** |



改为false



线程会重复获取锁。如果申请获取锁的线程足够多,那么可能会造成某些线程长时间得不到锁。这就是非公平锁的“饥饿”问题。

### 公平锁和非公平锁该如何选择

大部分情况下我们使用非公平锁，因为其性能比公平锁好很多。但是公平锁能够避免线程饥饿，某些情况下也很有用。

## ReentrantLock可响应中断

当使用synchronized实现锁时,阻塞在锁上的线程除非获得锁否则将一直等待下去，也就是说这种无限等待获取锁的行为无法被中断。而ReentrantLock给我们提供了一个可以响应中断的获取锁的方法lockInterruptibly()。该方法可以用来解决死锁问题。

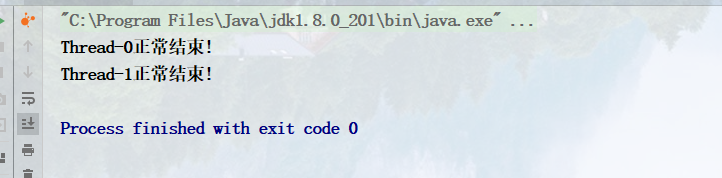
|  |
| --- |
| **static** Lock *lock1* = **new** ReentrantLock(); **static** Lock *lock2* = **new** ReentrantLock(); **public static void** main(String[] args) **throws** InterruptedException {   Thread thread = **new** Thread(**new** ThreadDemo(*lock1*, *lock2*));*//该线程先获取锁1,再获取锁2* Thread thread1 = **new** Thread(**new** ThreadDemo(*lock2*, *lock1*));*//该线程先获取锁2,再获取锁1* thread.start();  thread1.start();  thread.interrupt();*//是第一个线程中断* }  **static class** ThreadDemo **implements** Runnable {  Lock **firstLock**;  Lock **secondLock**;  **public** ThreadDemo(Lock firstLock, Lock secondLock) {  **this**.**firstLock** = firstLock;  **this**.**secondLock** = secondLock;  }  @Override  **public void** run() {  **try** {  **firstLock**.lockInterruptibly();  TimeUnit.***MILLISECONDS***.sleep(10);*//更好的触发死锁* **secondLock**.lockInterruptibly();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **firstLock**.unlock();  **secondLock**.unlock();  System.***out***.println(Thread.*currentThread*().getName()+**"正常结束!"**);  }  } } |



## 获取锁时限时等待

ReentrantLock还给我们提供了获取锁限时等待的方法tryLock(),可以选择传入时间参数,表示等待指定的时间,无参则表示立即返回锁申请的结果:true表示获取锁成功,false表示获取锁失败。我们可以使用该方法配合失败重试机制来更好的解决死锁问题。

|  |
| --- |
| **static** Lock *lock1* = **new** ReentrantLock(); **static** Lock *lock2* = **new** ReentrantLock(); **public static void** main(String[] args) **throws** InterruptedException {   Thread thread = **new** Thread(**new** ThreadDemo(*lock1*, *lock2*));*//该线程先获取锁1,再获取锁2* Thread thread1 = **new** Thread(**new** ThreadDemo(*lock2*, *lock1*));*//该线程先获取锁2,再获取锁1* thread.start();  thread1.start(); }  **static class** ThreadDemo **implements** Runnable {  Lock **firstLock**;  Lock **secondLock**;  **public** ThreadDemo(Lock firstLock, Lock secondLock) {  **this**.**firstLock** = firstLock;  **this**.**secondLock** = secondLock;  }  @Override  **public void** run() {  **try** {  **while**(!*lock1*.tryLock()){  TimeUnit.***MILLISECONDS***.sleep(10);  }  **while**(!*lock2*.tryLock()){  *lock1*.unlock();  TimeUnit.***MILLISECONDS***.sleep(10);  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **firstLock**.unlock();  **secondLock**.unlock();  System.***out***.println(Thread.*currentThread*().getName()+**"正常结束!"**);  }  } } |



线程通过调用tryLock()方法获取锁,第一次获取锁失败时会休眠10毫秒,然后重新获取，直到获取成功。第二次获取失败时,首先会释放第一把锁,再休眠10毫秒,然后重试直到成功为止。线程获取第二把锁失败时将会释放第一把锁，这是解决死锁问题的关键,避免了两个线程分别持有一把锁然后相互请求另一把锁。

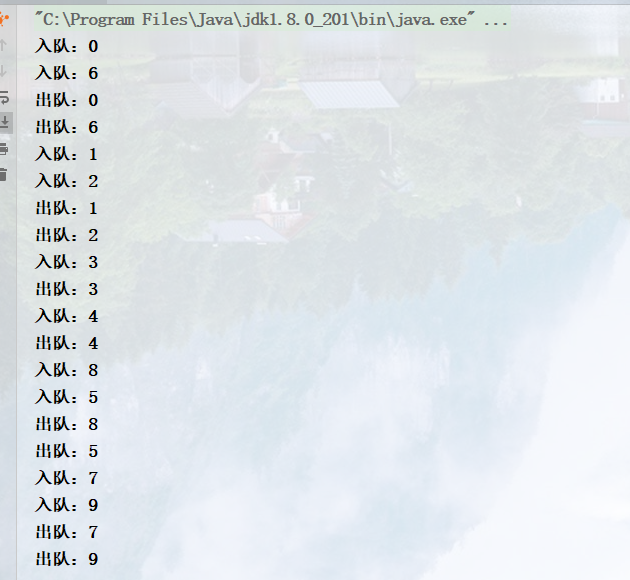
## 使用Condition实现简单的阻塞队列

阻塞队列是一种特殊的先进先出队列,它有以下几个特点

1.入队和出队线程安全

2.当队列满时,入队线程会被阻塞;当队列为空时,出队线程会被阻塞。

|  |
| --- |
| **int size**;*//阻塞队列最大容量* ReentrantLock **lock** = **new** ReentrantLock();  LinkedList<E> **list**=**new** LinkedList<>();*//队列底层实现* Condition **notFull** = **lock**.newCondition();*//队列满时的等待条件* Condition **notEmpty** = **lock**.newCondition();*//队列空时的等待条件* **public** LockTest(**int** size) {  **this**.**size** = size; }  **public void** enqueue(E e) **throws** InterruptedException {  **lock**.lock();  **try** {  **while** (**list**.size() ==**size**)*//队列已满,在notFull条件上等待* **notFull**.await();  **list**.add(e);*//入队:加入链表末尾* System.***out***.println(**"入队："** +e);  **notEmpty**.signal(); *//通知在notEmpty条件上等待的线程* } **finally** {  **lock**.unlock();  } }  **public** E dequeue() **throws** InterruptedException {  E e;  **lock**.lock();  **try** {  **while** (**list**.size() == 0)*//队列为空,在notEmpty条件上等待* **notEmpty**.await();  e = **list**.removeFirst();*//出队:移除链表首元素* System.***out***.println(**"出队："**+e);  **notFull**.signal();*//通知在notFull条件上等待的线程* **return** e;  } **finally** {  **lock**.unlock();  } }  **public static void** main(String[] args) {  LockTest<Integer> queue = **new** LockTest<>(2);  **for** (**int** i = 0; i < 10; i++) {  **int** data = i;  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **try** {  queue.enqueue(data);  } **catch** (InterruptedException e) {   }  }  }).start();   }  **for**(**int** i=0;i<10;i++){  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **try** {  Integer data = queue.dequeue();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }).start();  } } |

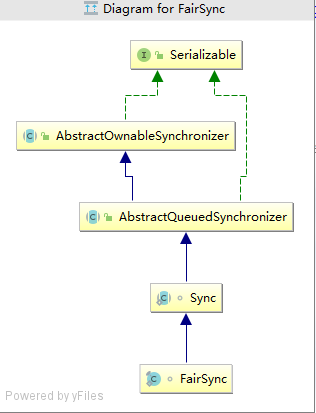


## 总结

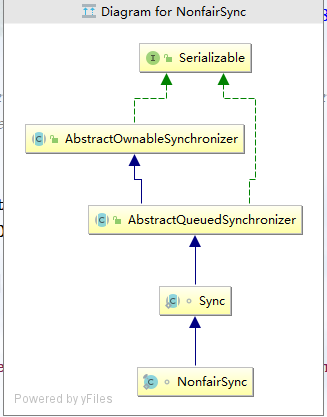
ReentrantLock是可重入的独占锁。比起synchronized功能更加丰富，支持公平锁实现，支持中断响应以及限时等待等等。可以配合一个或多个Condition条件方便的实现等待通知机制。

## 源码角度

### FairSync

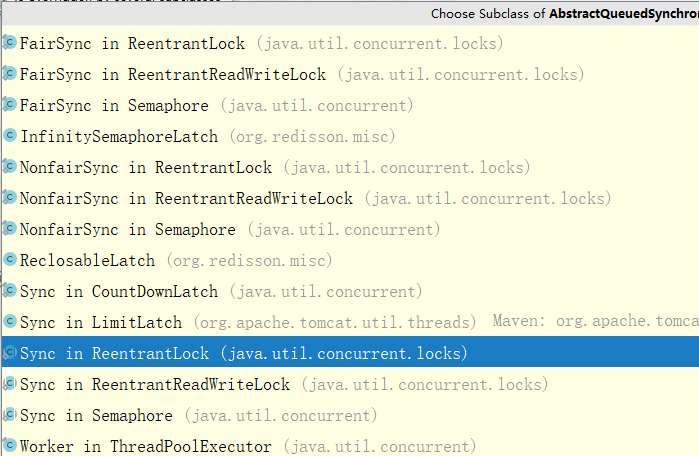


### NonfairSync



### AbstractQueuedSynchronizer介绍

AbstractQueuedSynchronizer,简称AQS,为构建不同的同步组件(重入锁,读写锁,CountDownLatch等)提供了可扩展的基础框架



AQS以模板方法模式在内部定义了获取和释放同步状态的模板方法,并留下钩子函数供子类继承时进行扩展,由子类决定在获取和释放同步状态时的细节,从而实现满足自身功能特性的需求。除此之外,AQS通过内部的同步队列管理获取同步状态失败的线程,向实现者屏蔽了线程阻塞和唤醒的细节。

#### AQS的内部结构(ReentrantLock的语境下)

AQS的内部结构主要由同步等待队列构成

##### 同步等待队列

AQS中同步等待队列的实现是一个带头尾指针(这里用指针表示引用是为了后面讲解源码时可以更直观形象,况且引用本身是一种受限的指针)且不带哨兵结点(后文中的头结点表示队列首元素结点,不是指哨兵结点)的双向链表。

|  |
| --- |
| */\*\*  \* Head of the wait queue, lazily initialized. Except for  \* initialization, it is modified only via method setHead. Note:  \* If head exists, its waitStatus is guaranteed not to be  \* CANCELLED.  \*/* **private transient volatile** Node **head**;  */\*\*  \* Tail of the wait queue, lazily initialized. Modified only via  \* method enq to add new wait node.  \*/* **private transient volatile** Node **tail**; |

head是头指针,指向队列的首元素;tail是尾指针,指向队列的尾元素。而队列的元素结点Node定义在AQS内部,主要有如下几个成员变量

volatile Node prev; //指向前一个结点的指针

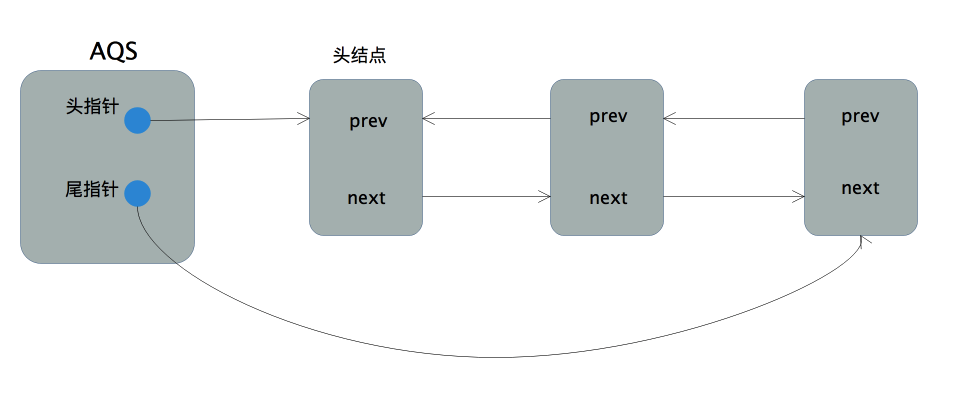
volatile Node next; //指向后一个结点的指针

volatile Thread thread; //当前结点代表的线程

volatile int waitStatus; //等待状态

* prev:指向前一个结点的指针
* next:指向后一个结点的指针
* thread:当前结点表示的线程,因为同步队列中的结点内部封装了之前竞争锁失败的线程,故而结点内部必然有一个对应线程实例的引用
* waitStatus:对于重入锁而言,主要有3个值。0:初始化状态；-1(SIGNAL):当前结点表示的线程在释放锁后需要唤醒后续节点的线程;1(CANCELLED):在同步队列中等待的线程等待超时或者被中断,取消继续等待。

**同步队列的结构如下图所示**

****

##### AQS中的其他数据结构(ReentrantLock的语境下)

（1）同步状态变量

|  |
| --- |
| /\*\*  \* The synchronization state.  \*/  private volatile int state; |

这是一个带volatile前缀的int值,是一个类似计数器的东西。在不同的同步组件中有不同的含义。以ReentrantLock为例,state可以用来表示该锁被线程重入的次数。当state为0表示该锁不被任何线程持有;当state为1表示线程恰好持有该锁1次(未重入);当state大于1则表示锁被线程重入state次。因为这是一个会被并发访问的量,为了防止出现可见性问题要用volatile进行修饰。

1. 持有同步状态的线程标志

|  |
| --- |
| /\*\*  \* The current owner of exclusive mode synchronization.  \*/  private transient Thread exclusiveOwnerThread; |

如注释所言,这是在独占同步模式下标记持有同步状态线程的。ReentrantLock就是典型的独占同步模式,该变量用来标识锁被哪个线程持有。

#### 获取锁和加锁的事项

|  |
| --- |
| **private static final** Unsafe ***unsafe*** = Unsafe.*getUnsafe*(); **private static final long *stateOffset***; **private static final long *headOffset***; **private static final long *tailOffset***; **private static final long *waitStatusOffset***; **private static final long *nextOffset***;  **static** {  **try** {  **// 获取某个属性的地址偏移值**  ***stateOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"state"**));  ***headOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"head"**));  ***tailOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"tail"**));  ***waitStatusOffset*** = ***unsafe***.objectFieldOffset  (Node.**class**.getDeclaredField(**"waitStatus"**));  ***nextOffset*** = ***unsafe***.objectFieldOffset  (Node.**class**.getDeclaredField(**"next"**));   } **catch** (Exception ex) { **throw new** Error(ex); } } |

#### 获取锁失败的线程如何安全的加入同步队列:addWaiter()

|  |
| --- |
| ***/\*\*  \* Creates and enqueues node for current thread and given mode.  \*  \* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared  \* @return the new node  \*/* 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; }**  **private** Node enq(**final** Node node) {  **for** (;;) {  Node t = **tail**;  **if** (t == **null**) { *// Must initialize*  ***// 头节点不存储有线程的信息,只是一个标志而已*** **if** (compareAndSetHead(**new** Node()))  **tail** = **head**;  } **else** {  node.**prev** = t;  **// 不断修改tail所指向的节点来插入新的节点**  **if** (compareAndSetTail(t, node)) {  t.**next** = node;  **return** t;  }  }  } } |

##### 线程加入同步队列后会做什么:acquireQueued()

|  |
| --- |
| **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;  }  if (*shouldParkAfterFailedAcquire*(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  } }** |

这段代码主要的内容都在for循环中,这是一个死循环,主要有两个if分句构成。第一个if分句中,当前线程首先会判断前驱结点是否是头结点,如果是则尝试获取锁,获取锁成功则会设置当前结点为头结点(更新头指针)。为什么必须前驱结点为头结点才尝试去获取锁？因为头结点表示当前正占有锁的线程,正常情况下该线程释放锁后会通知后面结点中阻塞的线程,阻塞线程被唤醒后去获取锁,这是我们希望看到的。然而还有一种情况,就是前驱结点取消了等待,此时当前线程也会被唤醒,这时候就不应该去获取锁,而是往前回溯一直找到一个没有取消等待的结点,然后将自身连接在它后面。一旦我们成功获取了锁并成功将自身设置为头结点,就会跳出for循环。否则就会执行第二个if分句:确保前驱结点的状态为SIGNAL,然后阻塞当前线程。

先来看shouldParkAfterFailedAcquire(p, node)，从方法名上我们可以大概猜出这是判断是否要阻塞当前线程的,方法内容如下

|  |
| --- |
| **private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  int ws = pred.waitStatus;  if (ws == Node.*SIGNAL*)  */\*  \* This node has already set status asking a release  \* to signal it, so it can safely park.  \*/* return true;  if (ws > 0) {  */\*  \* Predecessor was cancelled. Skip over predecessors and  \* indicate retry.  \*/* do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  */\*  \* waitStatus must be 0 or PROPAGATE. Indicate that we  \* need a signal, but don't park yet. Caller will need to  \* retry to make sure it cannot acquire before parking.  \*/  compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*);  }  return false; }** |

可以看到针对前驱结点pred的状态会进行不同的处理

1.pred状态为SIGNAL,则返回true,表示要阻塞当前线程。

2.pred状态为CANCELLED,则一直往队列头部回溯直到找到一个状态不为CANCELLED的结点,将当前节点node挂在这个结点的后面。

3.pred的状态为初始化状态,此时通过compareAndSetWaitStatus(pred, ws, Node.SIGNAL)方法将pred的状态改为SIGNAL。

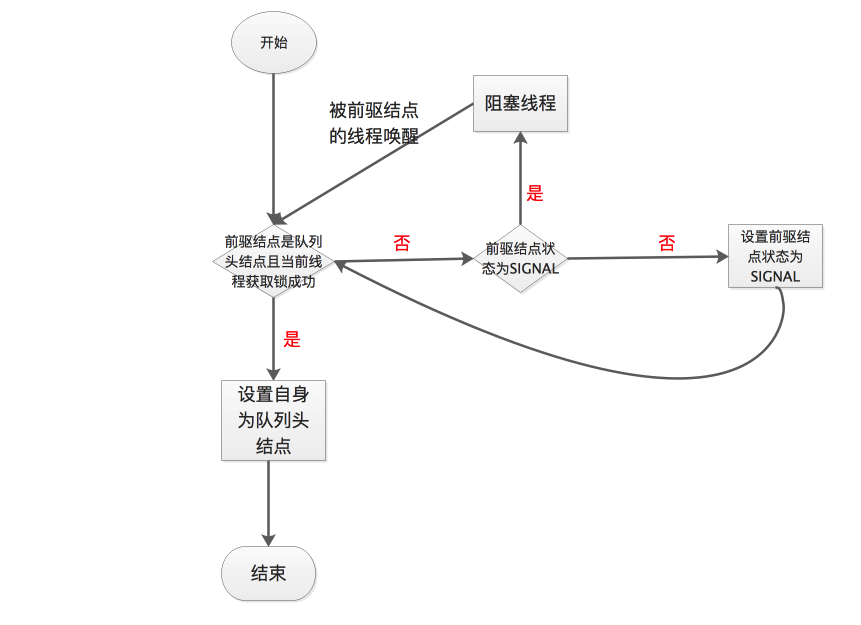
其实这个方法的含义很简单,就是确保当前结点的前驱结点的状态为SIGNAL,SIGNAL意味着线程释放锁后会唤醒后面阻塞的线程。毕竟,只有确保能够被唤醒，当前线程才能放心的阻塞。

但是要注意只有在前驱结点已经是SIGNAL状态后才会执行后面的方法立即阻塞,对应上面的第一种情况。其他两种情况则因为返回false而重新执行一遍for循环。这种延迟阻塞其实也是一种高并发场景下的优化,试想我如果在重新执行循环的时候成功获取了锁,是不是线程阻塞唤醒的开销就省了呢？

最后我们来看看阻塞线程的方法parkAndCheckInterrupt

shouldParkAfterFailedAcquire返回true表示应该阻塞当前线程,则会执行parkAndCheckInterrupt方法,这个方法比较简单,底层调用了LockSupport来阻塞当前线程,源码如下:

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



概括的说,线程在同步队列中会尝试获取锁,失败则被阻塞,被唤醒后会不停的重复这个过程,直到线程真正持有了锁,并将自身结点置于队列头部。

### 非公平模式解锁流程

### 为什么非公平锁性能好

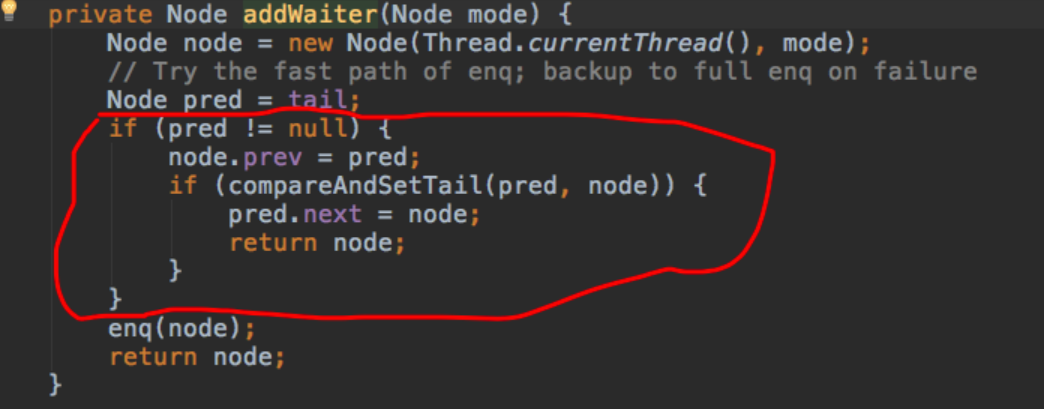
非公平锁对锁的竞争是抢占式的(队列中线程除外),线程在进入等待队列前可以进行两次尝试,这大大增加了获取锁的机会。这种好处体现在两个方面:

1.线程不必加入等待队列就可以获得锁,不仅免去了构造结点并加入队列的繁琐操作,同时也节省了线程阻塞唤醒的开销,线程阻塞和唤醒涉及到线程上下文的切换和操作系统的系统调用,是非常耗时的。在高并发情况下,如果线程持有锁的时间非常短,短到线程入队阻塞的过程超过线程持有并释放锁的时间开销,那么这种抢占式特性对并发性能的提升会更加明显。

2.减少CAS竞争。如果线程必须要加入阻塞队列才能获取锁,那入队时CAS竞争将变得异常激烈,CAS操作虽然不会导致失败线程挂起,但不断失败重试导致的对CPU的浪费也不能忽视。除此之外,加锁流程中至少有两处通过将某些特殊情况提前来减少CAS操作的竞争,增加并发情况下的性能。一处就是获取锁时将非重入的情况提前,如下图所示



另一处就是入队的操作,将同步队列非空的情况提前处理



这两部分的代码在之后的通用逻辑处理中都有,很显然属于重复代码,但因为避免了执行无意义的流程代码,比如for循环,获取同步状态等,高并发场景下也能减少CAS竞争失败的可能。