基于CAS,compareAndSet 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*）]

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 |

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

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;  
 }  
}

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）

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

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

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