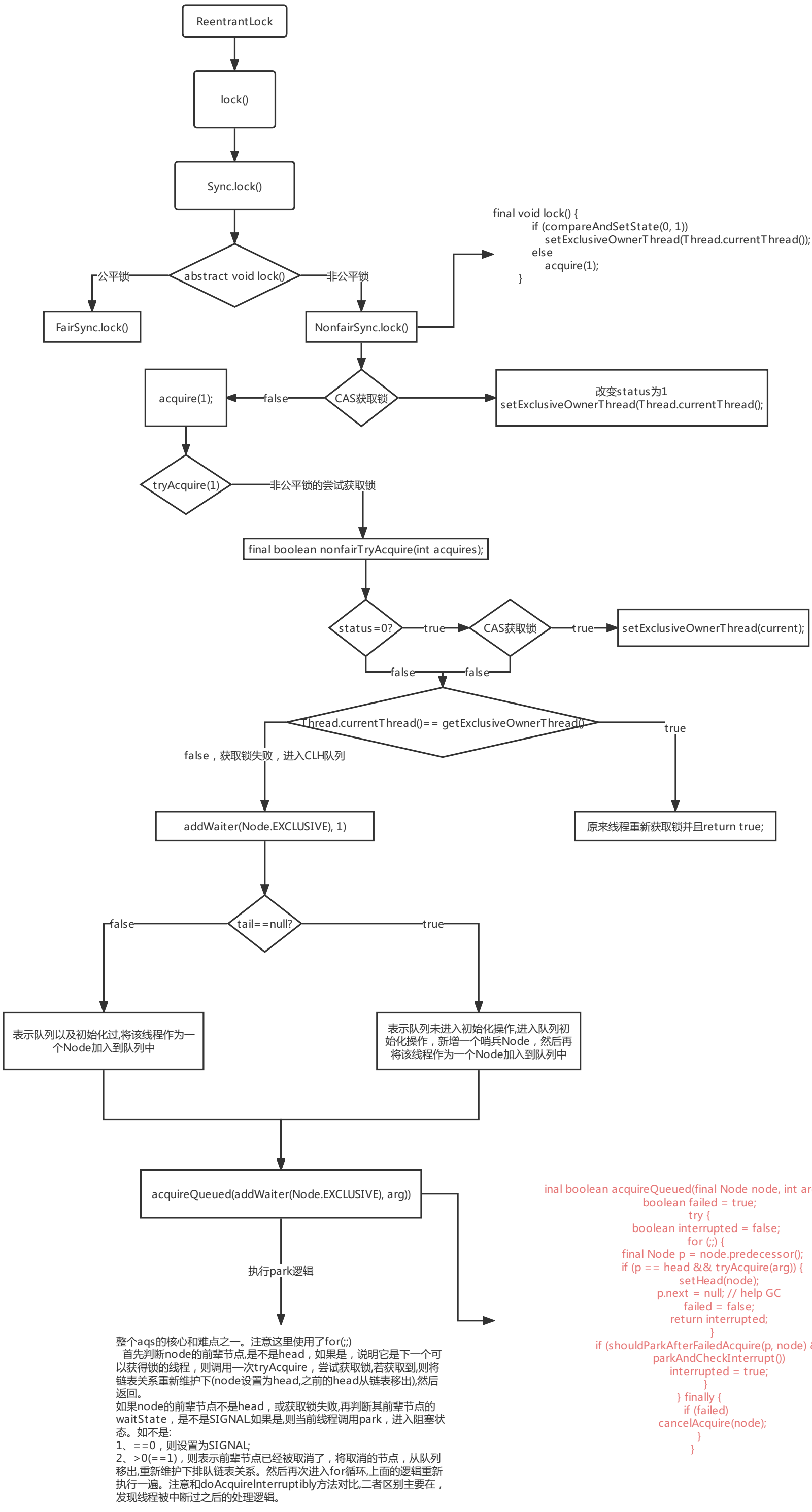
ReentrantLock
lock()
Sync.lock()
abstract void lock()
公平锁
非公平锁
FairSync.lock()
NonfairSync.lock()
final void lock() {
if (compareAndSetState(0, 1))
setExclusiveOwnerThread(Thread.currentThread());
else
acquire(1);
}
CAS获取锁
false
acquire(1);
改变status为1
setExclusiveOwnerThread(Thread.currentThread();
tryAcquire(1)
非公平锁的尝试获取锁
final boolean nonfairTryAcquire(int acquires);
status=0?
true
CAS获取锁
true
setExclusiveOwnerThread(current);
false
false
Thread.currentThread()== getExclusiveOwnerThread()
true
false，获取锁失败，进入CLH队列
addWaiter(Node.EXCLUSIVE), 1)
原来线程重新获取锁并且return true;
tail==null?
false
true
表示队列以及初始化过,将该线程作为一个Node加入到队列中
表示队列未进入初始化操作,进入队列初始化操作，新增一个哨兵Node，然后再将该线程作为一个Node加入到队列中
acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
执行park逻辑
inal 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);
}
}
整个aqs的核心和难点之一。注意这里使用了for(;;)
首先判断node的前辈节点,是不是head，如果是，说明它是下一个可以获得锁的线程，则调用一次tryAcquire，尝试获取锁,若获取到,则将链表关系重新维护下(node设置为head,之前的head从链表移出),然后返回。
如果node的前辈节点不是head，或获取锁失败,再判断其前辈节点的waitState，是不是SIGNAL.如果是,则当前线程调用park，进入阻塞状态。如不是:
1、==0，则设置为SIGNAL;
2、>0(==1)，则表示前辈节点已经被取消了，将取消的节点，从队列移出,重新维护下排队链表关系。然后再次进入for循环,上面的逻辑重新执行一遍。注意和doAcquireInterruptibly方法对比,二者区别主要在，发现线程被中断过之后的处理逻辑。