链滴

(源码)JDK8-JUC-AQS 结合 ReentrantLock

作者：yazong

原文链接：https://ld246.com/article/1634480598089

来源网站：链滴

----选择最熟悉/简单的ReentrantLock作为突破口。

理论初步

前置知识:

公平锁和非公平锁

可重入锁

LockSupport

CAS

Volatile

自旋锁

数据结构之链表

模板模式

AQS是什么?

字面意思，抽象的队列同步器。不仅是简单的加锁、解锁。

技术解释，用来构建锁或者其他同步器组件的重量级基础框架及整个JUC体系的基石，通过内置的FIF队列来完成获取线程的排队工作，并通过一个int类型变量表示持有锁的状态。

栈:先进先出

队列:单链表、双链表。ArrayList最经典的链表结构。

队列:靠链表和相应的数据结构完成队列的内容，各个线程要抢锁，抢得到的用，抢不到的要安排排队

同步与同步器不同:

锁:面向锁的使用者。定义了程序员和锁交互的使用层API，隐藏了实现细节，直接调用即可。

同步器:面向锁的实现者。比如java并发大神DougLee，提出统一规范并简化了锁的实现，屏蔽了同状态管理、阻塞线程排队和通知、唤醒机制等。

AQS作用

AQS能干嘛?加锁会导致阻塞，有阻塞就需要排队，实现排队必然需要有某种形式的队列进行管理。

(AQS使用一个volatile的int类型的成员变量来表示同步状态，通知内置的FIFO队列来完成资源获取的队工作，将每条要去抢占资源的线程封装成一个Node节点来实现锁的分配，通过CAS完成对state值修改)

解释说明:抢到资源的线程直接使用处理业务逻辑，抢不到资源的必然涉及一种排队等候机制。抢占资失败的线程继续去等待(类似银行业务办理窗口都满了，暂时没有受理窗口的顾客只能去候客区排队等

)，但等候线程仍然保留获取锁的可能且获取锁流程仍在继续(候客区的顾客也在等着叫号，轮到了再受理窗口办理业务)。

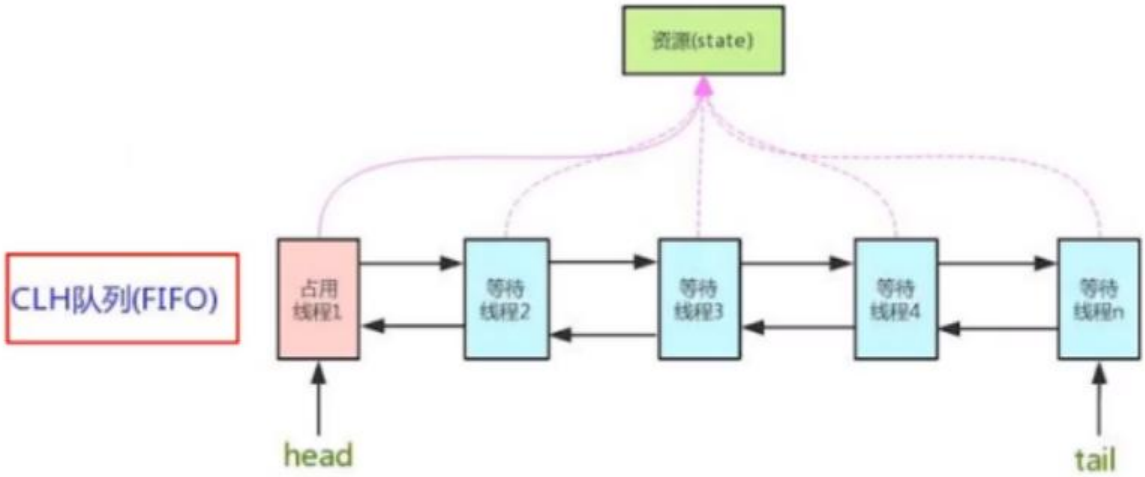
既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢?

如果共享资源被占用，那么就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是CLH列的变体实现的，将暂时获取不到锁的线程加入到队列中，这个队列就是AQS的抽象表现。它将请求享资源的线程封装成队列的节点(Node)，通过CAS、自旋以及LockSupport.park()的方式，维护stat变量的状态，使并发达到同步的控制效果。

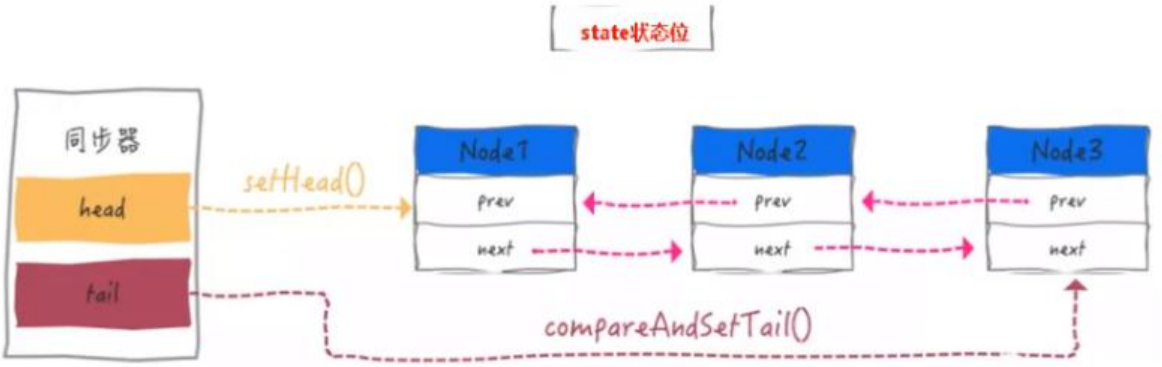
----AQS就是形成这样，唤醒，等待，加锁，释放，超时控制，取消等待的等等一系列操作。

怎样又快又快的实现上述功能呢?

AQS是用来构建锁或者其他同步器组件的重量级基础框架及整个JUC体系的基石，通过内置的FIFO队来完成资源获取线程的排队工作，并通过一个int类型变量表示持有锁的状态。

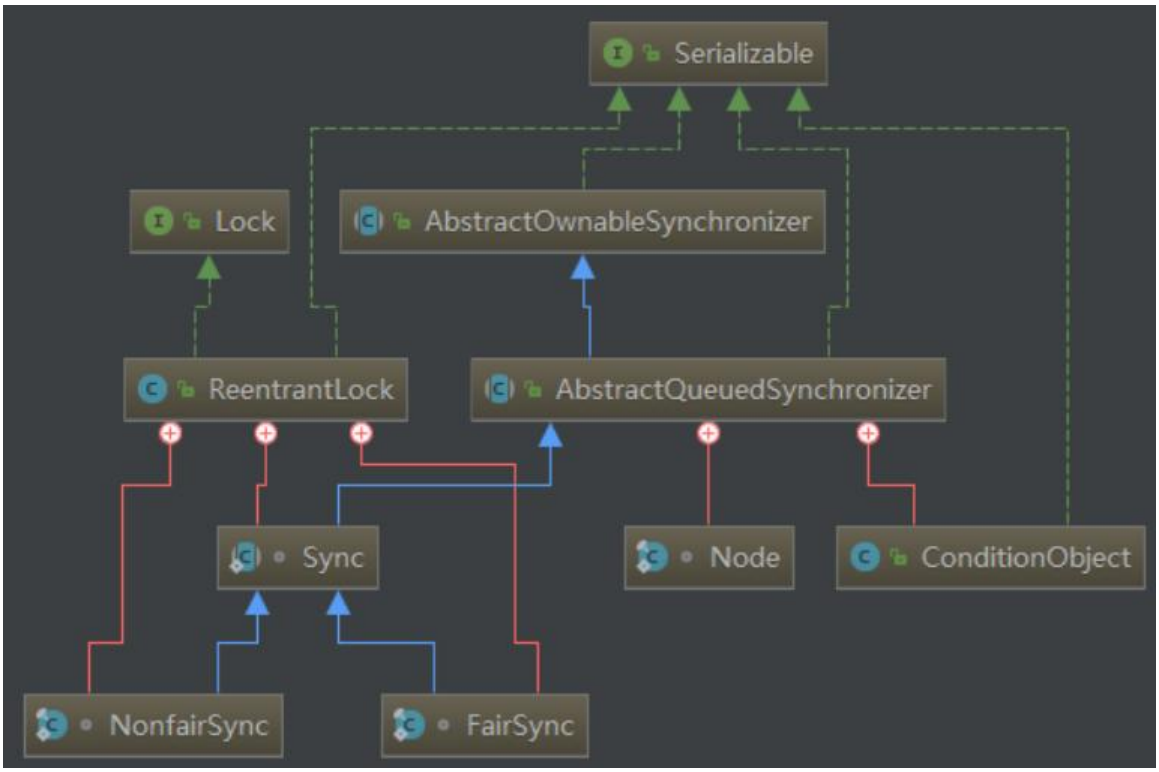


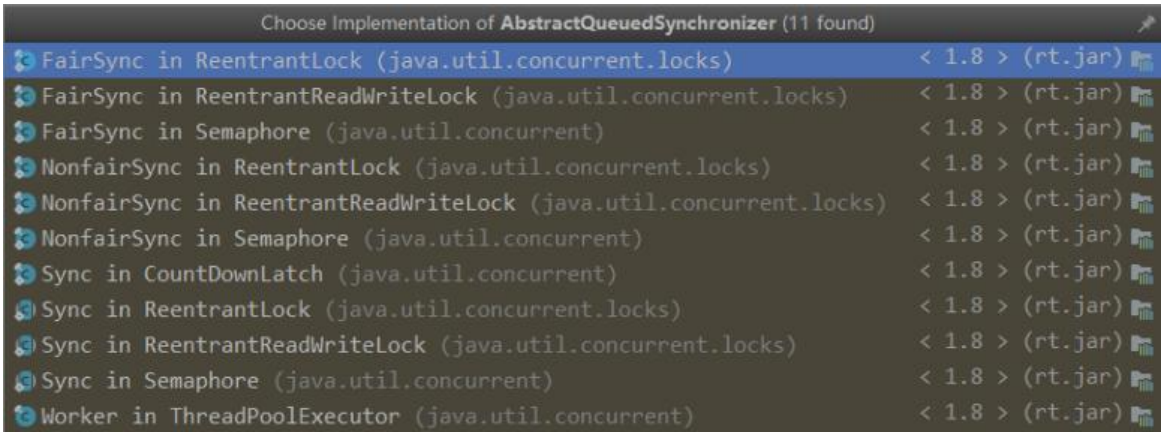
CLH:Craig、Landin and Hagersten队列(三个人名)，是一个单向链表，AQS中的队列是CLH变体的拟双向队列FIFO。



AQS核心类和重要属性

类结构关系





源码概要

```
----部分省略

public abstract class AbstractOwnableSynchronizer implements java.io.Serializable {
    private transient Thread exclusiveOwnerThread;
}

public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer imp
ements java.io.Serializable {
    static final class Node {
        static final Node SHARED = new Node();
        static final Node EXCLUSIVE = null;
        static final int CANCELLED =  1;
        static final int SIGNAL    = -1;
        static final int CONDITION = -2;
        static final int PROPAGATE = -3;
```

```
        volatile int waitStatus;
        volatile Node prev;
        volatile Node next;
        volatile Thread thread;
    }
    private transient volatile Node head;
    private transient volatile Node tail;
    private volatile int state;
}

public class ReentrantLock implements Lock, java.io.Serializable {
    private final Sync sync;
    abstract static class Sync extends AbstractQueuedSynchronizer {
        abstract void lock();
    }
    static final class NonfairSync extends Sync {}
    static final class FairSync extends Sync {}
    public ReentrantLock() {
        sync = new NonfairSync();
    }
    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
    public void lock() {
        sync.lock();
    }
    public void unlock() {
        sync.release(1);
    }
}
```

源码注释

```
public abstract class AbstractOwnableSynchronizer implements java.io.Serializable {
    //锁所属线程
private transient Thread exclusiveOwnerThread;
}

public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer imp
ements java.io.Serializable {
/**
等待队列节点类。

等待队列是“CLH”(Craig, Landin和Hagersten)锁队列的变体。
CLH锁通常用于自旋锁。
相反，我们使用它们来阻塞同步器，但使用相同的基本策略，即在其节点的前身中保存关于线程的一
控制信息。

每个节点中的“状态”字段跟踪线程是否应该阻塞。
节点在其前身发布时收到信号。
否则，队列的每个节点都充当持有单个等待线程的特定通知样式的监视器。
状态字段并不控制线程是否被授予锁等。
如果线程是队列中的第一个线程，则可以尝试获取。
```

但成为第一并不能保证成功;它只给予了竞争的权利。
因此，当前释放的竞争者线程可能需要重新等待。

要进入CLH锁队列，需要将其作为新的原子拼接进去尾巴。要退出队列，只需设置头部字段。
<pre>
    +------+ prev +-----+   +-----+
head | | <---- | | <---- | | tail
    +------+   +-----+   +-----+
</pre>

插入CLH队列只需要对“tail”进行单个原子操作，因此有一个简单的原子点来区分从未排队到已排
。
类似地，退出队列只涉及更新“头部”。
但是，节点需要更多的工作来确定谁是其后继者，部分原因是要处理由于超时和中断而可能的取消。
“prev”链接(未在原始CLH锁中使用)，主要用于处理取消。
如果一个节点被取消了，它的后续节点(通常)将被重新链接到一个未被取消的前任节点。
关于自旋锁的类似力学解释，请参阅Scott和Scherer在"http://www.cs.rochester.edu/u/scott/synchonization/"发表的论文。

我们还使用“next”链接去执行阻塞机制。
每个节点的线程id都保存在它自己的节点中，所以前任通过遍历下一个链接来确定它是哪个线程，从向下一个节点发出唤醒信号。
确定继承节点必须避免与新排队的节点竞争，以设置它们的前任节点的“下一个”字段

当一个节点的后继节点看起来是空的时候，可以通过从原子更新的“tail”向后检查来解决这个问题。
(或者换个说法，下一个链接是一种优化，所以我们通常不需要反向扫描。)

消去在基本算法中引入了一些保守性。
由于我们必须轮询其他节点的取消，我们可能无法注意到一个被取消的节点是在我们前面还是后面。
这个问题的处理方法是在取消后总是取消后继程序，允许它们稳定在一个新的前任程序上，除非我们确定一个未被取消的前任程序来承担这个责任。

CLH队列需要一个虚拟头节点来启动。
但是我们不是在建设的时候创造它们，因为如果没有争论，那就是白费力气。
相反，在第一次争用时构造节点并设置头和尾指针。

等待条件的线程使用相同的节点，但使用额外的链接。
条件只需要链接简单(非并发)链接队列中的节点，因为只有在独占持有时才会访问它们。
在等待时，节点被插入到条件队列中。
收到信号后，节点被转移到主队列。
状态字段的一个特殊值用于标记节点所在的队列。

*/
    static final class Node {
//标记,表示节点正在共享模式中等待
        static final Node SHARED = new Node();
//标记,表示节点正在独占模式中等待
        static final Node EXCLUSIVE = null;
/** waitStatus值表示线程已被取消 */
static final int CANCELLED = 1;
/** waitStatus值表示后续线程需要unparking */
static final int SIGNAL   = -1;
/** waitStatus值表示线程正在等待状态*/
static final int CONDITION = -2;

```
/** waitStatus值指示下一个获取的值应该无条件地传播 */
static final int PROPAGATE = -3;
/**
数值按数字排列以简化使用。
非负值意味着节点不需要发出信号。
因此，大多数代码不需要检查特定的值，只需要检查符号。
对于普通同步节点，该字段初始化为0，对于条件节点，该字段初始化为CONDITION。
可以使用CAS(或者在可能的情况下，无条件地写volatile)修改它。
*/
        volatile int waitStatus;
//等待队列节点的上一个节点
/**原文描述:
链接到当前节点/线程检查waitStatus所依赖的前一个节点。
在进入队列期间分配，仅在退出队列时为空(为了GC)。
此外，当取消一个前任时，我们在寻找一个始终存在的未取消的前任时进行短路，因为头节点永远不
被取消:
只有成功获取节点才会成为头节点。
被取消的线程永远不会成功获取，线程只会取消自己，不会取消任何其他节点。
*/
        volatile Node prev;
//等待队列节点的下一个节点
/**原文描述:
链接到当前节点/线程在释放时取消泊位的后续节点。
在排队时分配，在绕过被取消的前任时调整，在退出队列时为空(为了GC)。
查询操作直到附加之后才会分配前任的下一个字段，因此看到下一个字段为空并不一定意味着节点处
队列的末尾。
然而，如果下一个字段看起来是空的，我们可以从尾部扫描prev来再次检查。
取消节点的下一个字段被设置为指向节点本身而不是null，以使isOnSyncQueue的工作更容易。
*/
        volatile Node next;
//等待队列节点的所属线程
/**原文描述:
将此节点编入队列的线程。施工时初始化，使用后取消。
*/
        volatile Thread thread;
    }
//队列头结点,其实仅仅是一个指针,会指向队列中的哨兵节点,并非队列中真正的头结点.
/**
延迟初始化的等待队列头。
除了初始化，它只能通过方法setHead进行修改。
注意:如果head存在，它的waitStatus保证不会被CANCELLED。
*/
    private transient volatile Node head;
//队列尾结点,其实仅仅是一个指针,会指向队列中的真正的尾节点.
 / **
*等待队列的尾部，延迟初始化。仅通过方法enq修改以添加新的等待节点。
* /
    private transient volatile Node tail;
 / **
*同步状态。
* /
    private volatile int state;
}
```

```
public class ReentrantLock implements Lock, java.io.Serializable {
    private final Sync sync;
    //内部类Sync继承AQS
/**同步器提供所有实现机制*/
abstract static class Sync extends AbstractQueuedSynchronizer {
 / **
*执行{@link Lock# Lock}。子类化的主要原因是允许非公平版本的快速路径。
* /
abstract void lock();
    }
//非公平锁的同步对象,继承Sync,间接继承AQS.
    static final class NonfairSync extends Sync {}
//公平锁的同步对象,继承Sync,间接继承AQS.
    static final class FairSync extends Sync {}
    //默认实例化非公平锁
public ReentrantLock() {
        sync = new NonfairSync();
}
//标识位可切换公平锁还是非公平锁
public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
}
//内部类Sync调用加锁方法
 / **
*获取锁。
*如果锁没有被其他线程持有，则获取锁并立即返回，将锁持有计数设置为1。
*如果当前线程已经持有该锁，则持有计数加1，该方法立即返回。
*如果锁被另一个线程持有，那么当前线程将被禁用，并处于休眠状态，直到获得锁,当锁保持计数设
为1时。
* /
    public void lock() {
        sync.lock();
}
//内部类Sync调用解锁方法
 / **
*试图释放此锁。
*如果当前线程是这个锁的持有者，那么持有计数递减。
*如果持有计数现在是零，那么锁被释放。
*如果当前线程不是这个锁的持有者，那么{@link IllegalMonitorStateException}将被抛出。
* @throws IllegalMonitorStateException如果当前线程没有持有这个锁。
* /
    public void unlock() {
        sync.release(1);
    }
}
```

源码分析前提条件

锁被真正占用的两个状态:

state=1

ownerThread=currentThread

哨兵节点状态:

thread=null

prev=null

next=null或非null(有下一个节点)

关于各种状态和值的简单处理流程(省去了中间环节):

----先设置状态，再阻塞。

----先回溯状态，再解锁，再设置状态。

lock()

1)node0获取锁成功

设置state=1,ownerThread=node0

2.1)获取锁失败

设置上一个node节点的waitStatus=-1

2.2)执行park()阻塞当前节点node1

2.3)设置下一个线程节点wait=1,ownerThread=node1.并返回.

2.5)哨兵节点S1被GC回收,node1变成哨兵节点S。

unlock()

1)A0设置state=0,ownerThread=null

2)设置上一个node节点的waitStatus=0

3)执行unpark()唤醒下一个节点,执行lock()的2.3)

等待队列是“CLH”(Craig, Landin和Hagersten)锁队列的变体。

CLH锁通常用于自旋锁。

只有成功获取节点才会成为头节点。

被取消的线程永远不会成功获取，线程只会取消自己，不会取消任何其他节点。

下述源码分析中:

哨兵节点简称S

加入队列的节点/参数称为nodeX

源码 -业务总体流程案例图

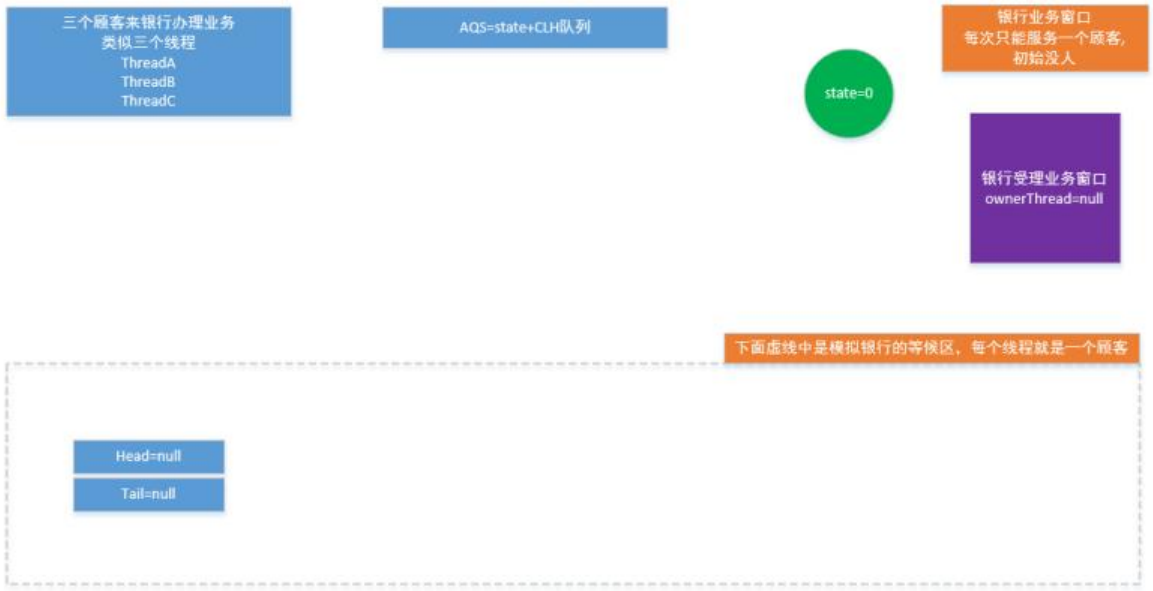
----业务结构流程图

----以非公平锁获取锁失败加入队列的情况为切入点画的流程图。

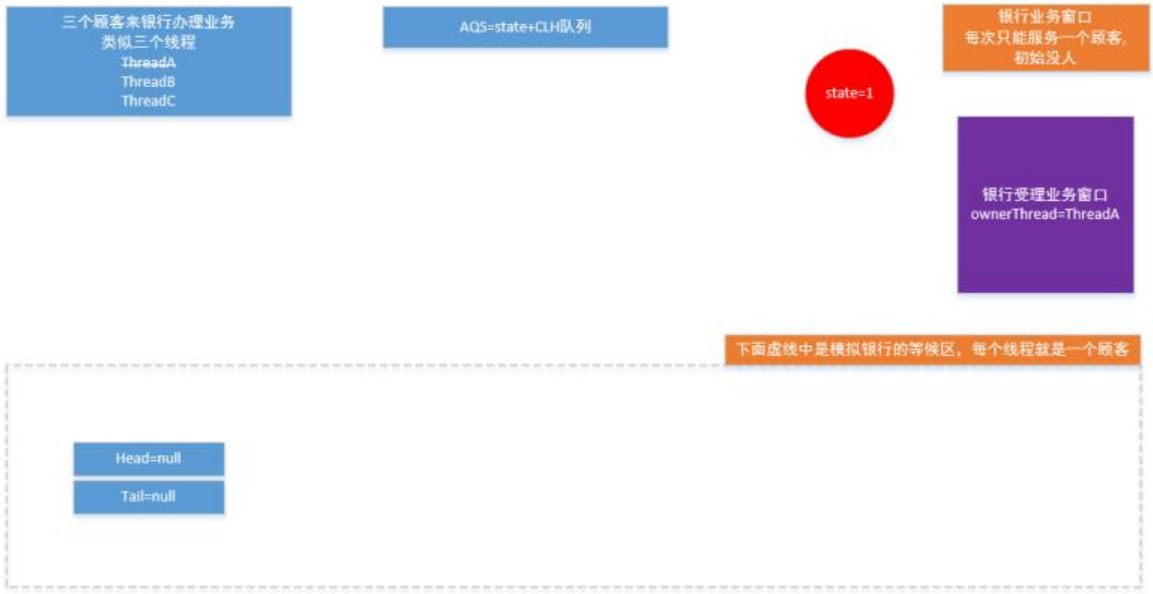
----以在银行办理业务为源码分析案例。

----这里只是简要说明,描述并不完善,更详细的参考不同方法的描述。

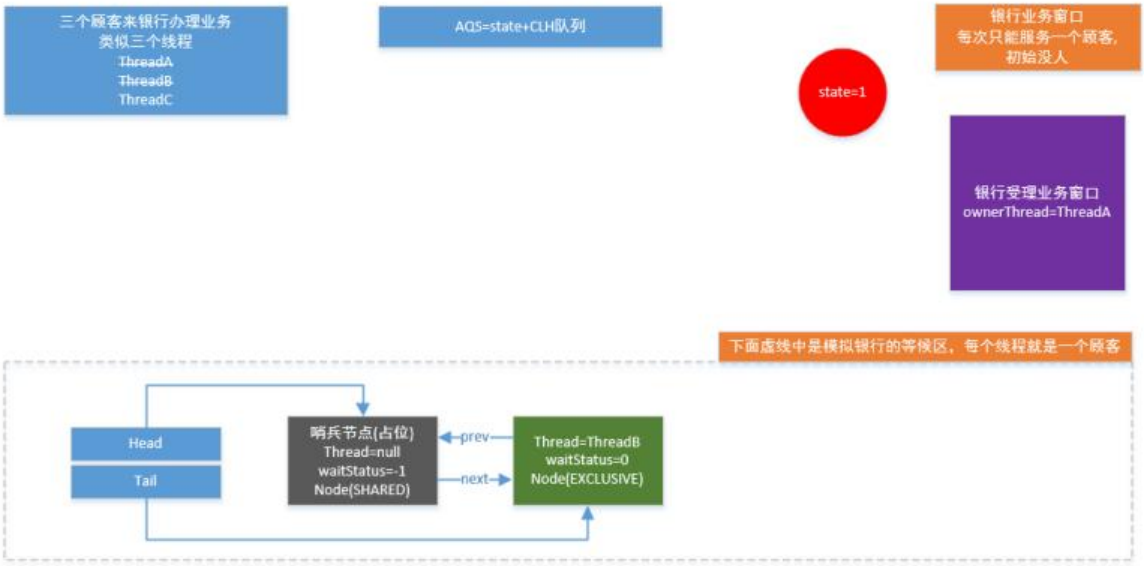
初始化队列模式



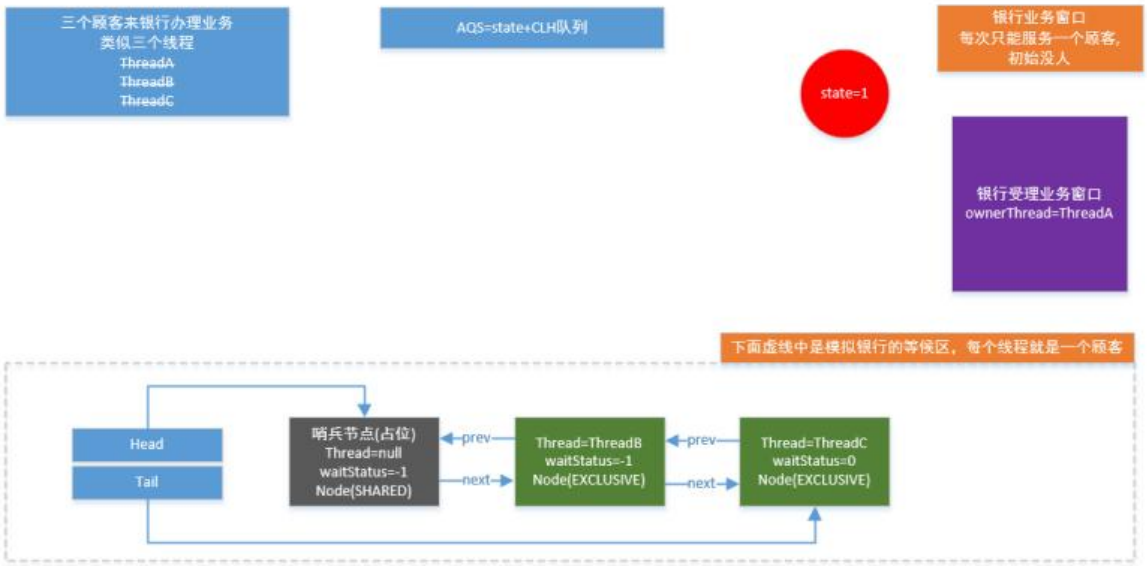
ThreadA 未抢到锁,入队,第一步,CAS 设置 state=1,锁所属线程 ownerThread=ThreadA.此时的 head 和 tail 都等于 null.



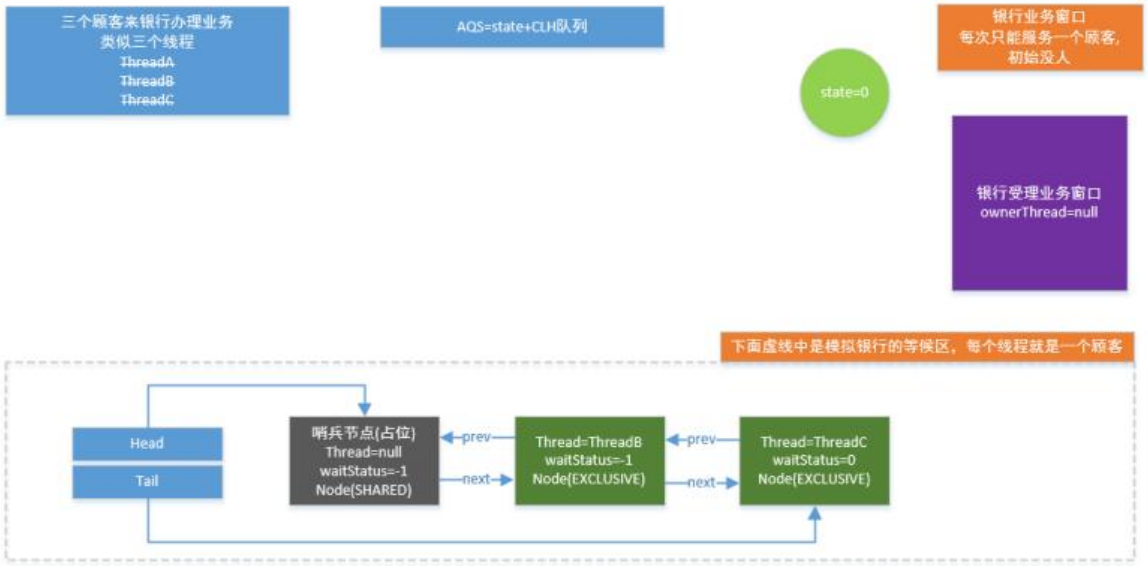
ThreadA 未抢到锁,入队,第三步,自旋生成加入阻塞队列的线程节点 ThreadB,此时 Node 类型为 EXCLUSIVE 独占类型,重置哨兵节点的 next、ThreadB 的 prev 和 tail 节点的指向,并给前一个节点,比如现在的哨兵节点的 waitStatus=-1,当前的 waitStatus 依然等于 0,并执行 park()阻塞当前 ThreadB 这个线程,等待被唤醒,要注意的是此时 for 自旋还一直在执行,其他线程依然可以依次加入此阻塞队列.



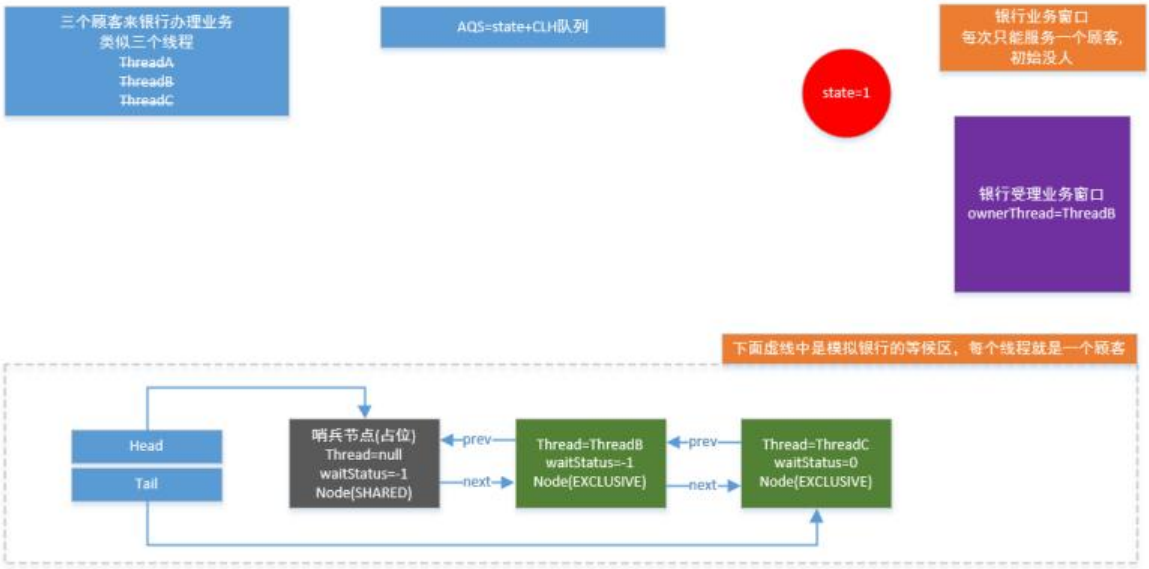
入队，ThreadC 同理。



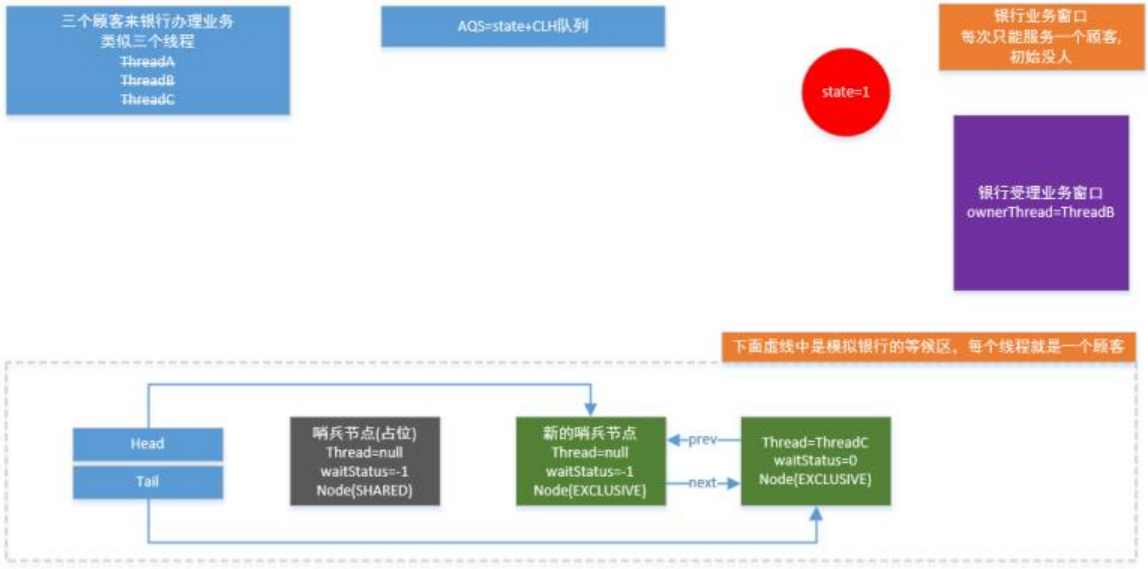
解锁出队,重置 ThreadA 释放锁(暂不考虑可重入锁),锁状态 state=0,ownerThread=null



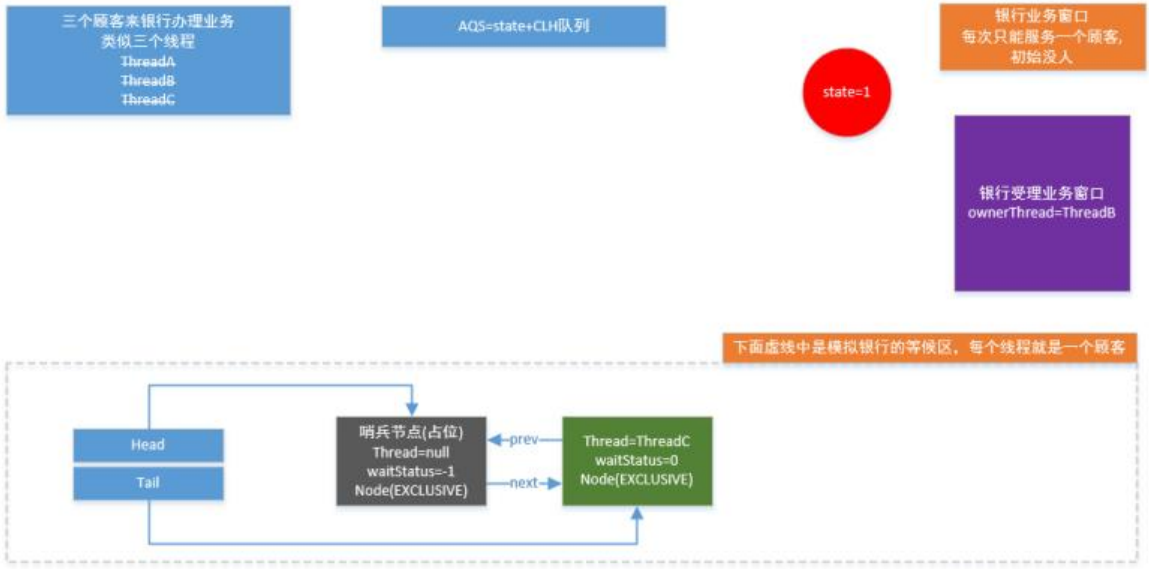
解锁出队,ThreadB 被 unpark() 唤醒,自旋,设置锁状态 state=1,锁所属线程 ownerThread=ThreadB,注意此时 ThreadB 还未真正出队,先设置状态,再出队.



解锁出队,自旋,ThreadB 设置锁状态 state=1,锁所属线程 ownerThread=ThreadB 后,设置队列中的 ThreadB 为哨兵节点,但 ThreadB 成为哨兵节点后,next 还是指向 ThreadC,waitStatus=-1,Node 类型由 SHARED 共享变成了 EXCLUSIVE 独占,tail 尾节点依然指向 ThreadC,head 头结点指向现在的哨兵节点,除非后续继续有线程入队.



解锁出队,自旋,原来的哨兵节点被 GC 收回.此时,才是 ThreadB 真正抢占锁成功.后续节点处理方式同理.



最终,可以还原成没有任何线程抢占锁,也没有阻塞队列的情况.即 state=0,ownerThread=null,
阻塞队列为空,哨兵节点也没有,head 和 tail 都等于 null.

案例 demo

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 可重入锁验证:显示锁ReentrantLock-案例05-非成对加锁解锁-多线程容易死锁
 */

public class ReEnterLockDemo06 {

    static Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {

        new Thread(()->{

            //System.out.println(Thread.currentThread().getName() + "\t unlock前无lock测试开始 .")

            //lock.unlock();
            //System.out.println(Thread.currentThread().getName() + "\t unlock前无lock测试结束 .")


            System.out.println(Thread.currentThread().getName() + "\t unlock前无lock测试-会抛异
.");

            lock.lock();
            lock.lock();
            try {
                System.out.println(Thread.currentThread().getName() + "\t 外层.");
                lock.lock();
                try {
```

```
                System.out.println(Thread.currentThread().getName() + "\t 内层.");
            }finally {
                lock.unlock();
            }
        }finally {
            lock.unlock();
            /**
             * 这里故意注释,实现加锁次数和释放次数不一样.
             * 由于加锁次数和释放次数不一样,第二个线程始终无法获取到,导致一直在等待.
             * 正常情况,加锁几次就要解锁几次.
             * 此时t1下面再加一个线程,t2就不输出了,t2无法获取锁了,这样就容易导致死锁和生产的一
问题.
             * --同一个线程可以多次获得属于自己的同一把锁,这样就可以一定程度的避免死锁.
             */
            //lock.unlock();

            System.out.println(Thread.currentThread().getName() + "\t 测试同一个线程未成对unl
ck接着lock是否阻塞-开始.");

            lock.lock();

            System.out.println(Thread.currentThread().getName() + "\t 测试同一个线程未成对unl
ck接着lock是否阻塞-结束.");

        }
    },"t1").start();

    Thread.sleep(2000);
    System.out.println(Thread.currentThread().getName() + "\t 开启另外线程测试上一个线程
成对unlock是否能lock开始 ." );

    lock.lock();

    System.out.println(Thread.currentThread().getName() + "\t 开启另外线程测试上一个线程
成对unlock是否能lock结束 ." );

    new Thread(()->{
        System.out.println(Thread.currentThread().getName() + "\t 之前线程未成对unlock()这
依然同样阻塞进不来lock.");
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "\t 内层.");
        }finally {
            lock.unlock();
        }
    },"t1").start();

  }

}

/**
输出结果:
 t1     unlock前无lock测试-会抛异常 .
```

```
t1    外层.
t1    内层.
t1    测试同一个线程未成对unlock接着lock是否阻塞-开始.
t1    测试同一个线程未成对unlock接着lock是否阻塞-结束.
main    开启另外线程测试上一个线程未成对unlock是否能lock开始 .
......
*/
```

加锁-lock.lock()

方法调用顺序

(以获取不到锁加入队列为例):

lock->acquire->tryAcquire->addWaiter(enq-自旋)->acquireQueued(自旋-shouldParkAfterFaildAcquire->parkAndCheckInterrupt)

方法-lock()

```
#公平锁
所属类:ReentrantLock$FairSync
public void lock() {
    sync.lock();
}
final void lock() {
    acquire(1);
}

#非公平锁
所属类:ReentrantLock$NonfairSync
final void lock() {
    //当前线程获取锁成功
    //CAS:把state从0改为1,成功则当前线程抢到锁.对应unlock()中compareAndSetState(1, 0).
    //如果CAS失败,那么有可能是当前线程再次获取锁,或者别的线程在占用锁.
    if (compareAndSetState(0, 1))//true
        //把ownerThread设置为currentThread.对应unlock()中setExclusiveOwnerThread(null);
        //exclusiveOwnerThread独占模式同步的当前所有者。
        setExclusiveOwnerThread(Thread.currentThread());
        //此时并未创建任何Node对象,只是说AQS有线程占用了AQS锁.
    else//false
        //当前线程获取锁失败
        //在独占模式下获取锁,忽略中断.失败时恢复正常.
        //可重入锁也在这里执行
        acquire(1);
}
```

方法(M1)-acquire

```
所属类:AQS
//尝试获取锁,如果获取不到就排队(阻塞当前线程)
//在独占模式下获取，忽略中断。
```

```
//通过至少调用一次{@link #tryAcquire}来实现，成功时返回。
//否则，线程将进入队列，可能会反复阻塞和解除阻塞，调用{@link #tryAcquire}直到成功。
public final void acquire(int arg) {
    //获取锁失败并且加入队列,则执行线程中断.
    //tryAcquire返回true抢锁成功(包括当前线程获取到锁和获取可重入锁),则不执行后面的加入队列作.acquireQueued是真正入队的入口.
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        /**
        parkAndCheckInterrupt()判断Thread.interrupted()线程是否会挂起.
        因为LockSupport.park挂起的线程不仅会被LockSupport.unpark方法唤醒，还会被中断唤醒所以线程被唤醒后调用了下Thread.interrupted()方法来返回下当前线程的中断状态，
        但是该方法会清楚掉线程的中断状态，所以在acquire()又一次的通过selfInterrupt()设置了下中状态。
        这也是ReentrantLock区别于synchronized关键字的一个地方。
        ReentrantLock支持等待队列中的线程中断，synchronized不支持。
        */
        selfInterrupt();
}
```

方法 (M1-1)-tryAcquire

```
//tryAcquire()尝试获取锁,导致失败的几种情况:
1)锁已经被其他线程获取.
2)锁没有被其他线程获取,但是需要排队.
3)CAS失败(可能过程中已经有其他线程获取到锁了).
锁为自由状态(state=0),并不能说明可以立即执行CAS获取锁,因为可能在当前线程获取锁之前,已经有他线程在排队了,必须遵循先来后到的原则获取锁.
所以还要调用hasQueuedPredecessors()查看自己是否需要排队。
```

公平锁与非公平锁的区别

在此tryAcquire()中，公平锁只比非公平锁多了hasQueuedPredecessors()的判断，判断等待队列中否存在有效的节点，即判断了是否需要排队，导致公平锁和非公平锁的差异如下:

公平锁:先来先到，线程在获取锁时，如果这个锁的等待队列总已经有线程在等待，那么当前线程就会入等待队列中。

非公平锁:不管是否有等待队列，如果可以获取锁，则立刻占有锁对象，也就是说队列的第一个排队线在unpark()，但之后还是需要竞争锁(存在线程竞争的情况)。

再者，公平锁每次获取到锁为同步队列中的第一个节点，保证请求资源时间上的绝对顺序，而非公平有可能刚释放锁的线程下次继续获取该锁，则有可能导致其他线程永远无法获取到锁，造成“饥饿”象。

公平锁为了保证时间上的绝对顺序，需要频繁的上下文切换，而非公平锁会降低一定的上下文切换，低性能开销。

因此，ReentrantLock默认选择的是非公平锁，则是为了减少一部分上下文切换，保证了系统更大的吐量。

非公平锁 -tryAcquire

```
所属类:AQS$NonfairSync
//非公平锁获取锁过程
//抢到锁返回true,表示获取到了锁/需要排队,线程逐级返回,加锁过程结束.
//抢不到锁返回false,表示无需排队.
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

非公平锁 - nonfairTryAcquire

```
所属类:AQS$Sync
final boolean nonfairTryAcquire(int acquires) {
    //当前线程
    final Thread current = Thread.currentThread();
    //state状态
    int c = getState();
    //c==0表示锁还未被占用或上个线程已经释放此锁,此线程可直接获取锁,不用加入队列直接加锁.
    if (c == 0) {
        //CAS:把state从0改为1,成功则抢到锁.对应unlock()解锁中compareAndSetState(1, 0).
        if (compareAndSetState(0, acquires)) {
            //把ownerThread设置为currentThread.对应unlock()解锁中setExclusiveOwnerThread(null
;
            setExclusiveOwnerThread(current);
            //抢到锁
            return true;//即tryAcquire()获取锁成功,但此线程并不入队.
        }
    }
    //current == getExclusiveOwnerThread()表示当前线程和已经被锁的线程对象是同一个,
    //已经占有锁了,但也是抢到锁了,只是可重复锁,不用加入队列,直接加锁的状态.
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        //锁状态是个int类型，这里应该是担心锁重复的次数太多然后 溢出变成负数
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        //可重入锁:相同线程每次lock()每次state加1,每次unlock(),每次state减1.必须先执行lock()才能
行unlock().
        setState(nextc);
        return true;//即tryAcquire()获取锁成功,但此线程并不入队.
    }
    return false;//即tryAcquire()获取锁失败,此线程执行后续的入队操作.
}
```

公平锁 -tryAcquire

```
所属类:AQS$FairSync
//公平锁获取锁的过程,只是比非公平锁多了"!hasQueuedPredecessors()"的逻辑.
//其他所有逻辑同非公平锁相同,看非公平锁的nonfairTryAcquire逻辑即可.
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //返回true,表示需要排队
        //返回false,表示无需排队
```

```
        if (!hasQueuedPredecessors() &&
            //无需排队就执行下述两种方法
            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;
}
```

公平锁 -hasQueuedPredecessors

所属类:AQS
//判断等待队列中是否存在有效节点
//返回true,表示需要排队
//返回false,表示无需排队
//注意，由于中断和超时导致的取消可能在任何时候发生，{@code true}返回并不保证其他线程会在前线程之前获得。
//有几下几种情况需要考虑:
1)未初始化:
1.1)下述的tail、head都为null,此时还未建立队列,h != t返回false,此时表示无需排队。整个hasQueuePredecessors()方法返回false.
2)已初始化(表示下述的tail、head都不为null):
2.1)head等于tail,那么h != t返回false,此时虽然已建立队列,但只初始化了哨兵节点,后续节点还未真正队。此时表示无需排队,整个hasQueuedPredecessors()方法返回false.
2.2)head不等于tail,那么h != t返回true,还分下述两种情况:
2.2.1)head的next节点等于null,即(s = h.next) == null返回true。注意此时2.2.2)的流程就不会再执。
此时虽然已建立队列,但现在可能在入队/出队的过程中。(要注意的是head和tail并不是同时初始化的,考addWaiter()和acquireQueued())
此时表示无需排队,整个hasQueuedPredecessors()方法返回false.
2.2.2)head的next节点不等于null,即(s = h.next) == null返回false,表示已建立等待队列并且已有其线程加入等待队列,继续分下述两种情况:
2.2.2.1)head的next节点的线程等于当前执行线程,即s.thread != Thread.currentThread()返回false此时表示无需排队,整个hasQueuedPredecessors()方法返回false.
2.2.2.2)head的next节点的线程不等于当前执行线程,即s.thread != Thread.currentThread()返回tru。此时表示需要排队,整个hasQueuedPredecessors()方法返回true.

```
public final boolean hasQueuedPredecessors() {
    //它的正确性取决于head在tail之前被初始化，并且如果当前线程在队列中处于首位，则head.nex是准确的。
    Node t = tail; // 以反向初始化顺序读取字段
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

方法 (M1-2)-addWaiter

```
所属类:AQS
//仅用于设置队列节点的位置,并非真正的加入到队列(在acquireQueued中设置waitStatus为-1和park
)阻塞线程才是真正加入队列)
//通过CAS和字段保证一定会加入成功.
private Node addWaiter(Node mode) {
    //要注意这个node对象为EXCLUSIVE独占模式,而哨兵节点S为SHARED共享模式(有且仅在初始建
队列时的哨兵节点成立,之后迭代的哨兵节点都是EXCLUSIVE独占模式).
    Node node = new Node(Thread.currentThread(), mode);
    //第二部分(初始化队列在第一部分):队列中已存在线程锁对象,后续再加入的执行部分.
    Node pred = tail;
    if (pred != null) {
        //此时的内容参考下述"第一部分"中的"多个节点案例"即可.
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            //返回的是最新加入队列的节点.
            return node;
        }
    }
    //第一部分:初始化队列,即加入第一个元素(包括哨兵节点和第一个加入队列的线程锁对象)
    enq(node);
    //返回的是最新加入队列的节点,这个node节点在enq()中并未处理.
    return node;
}
```

方法 (M1-2-1)-enq

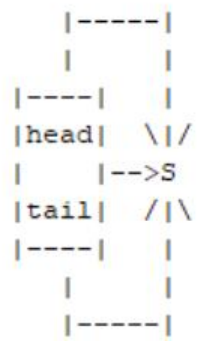
```
所属类:AQS
上面第一部分:调用enq方法初始化等待队列并通过cas及自旋保证成功加入到等待队列队尾.
//暂且把node当成第一个队列中的元素nodeX,哨兵节点除外.注意这个node参数节点为EXCLUSIVE
占模式.
private Node enq(final Node node) {
    //自旋锁处理方式
    for (;;) {
        Node t = tail;
        //第一次初始化(当队列为空时)(自旋第一次进入),仅处理哨兵节点,未处理真正将要加入队列中的
程锁对象nodeX.
        if (t == null) { // Must initialize
            //CAS:基于当前对象把head的值设置成S
            //设置哨兵节点S,实例化新node对象S(注意S对象标记为SHARED共享模式),而队列里的其他
点为EXCLUSIVE独占模式.
            if (compareAndSetHead(new Node()))
                //把head头结点和tail尾节点都指向S.
                //一旦设置head,那么head永远指向哨兵节点S(即使哨兵节点S会改变),tail会随着队列中加
的线程越来越多,一直往后移动直至指向最后一个线程节点.
                tail = head;
        } else {
            //第二次(自旋第二次进入,并返回),处理真正将要加入队列中的线程锁对象nodeX.
            //把nodeX的上一个节点prev指向tail尾节点
            //暂且把node当成第一个队列中的元素nodeX,哨兵节点除外.注意这个node参数节点为EXCL
SIVE独占模式.
```

```
            node.prev = t;
            //CAS:基于当前对象把tail尾节点的值由tail尾节点的Node对象设置成nodeX对象.
            //成功,第n次加入成功把当前线程节点加入到等待队列.
            //失败,加入到等待队列尾部的时候，有别的线程已经加入到等待队列了,再次自旋.
            if (compareAndSetTail(t, node)) {
                //设置tail尾节点的下一个节点next指向nodeX对象.
                t.next = node;
                //返回nodeX的上一个节点prev对象.
                //而这个enq()返回的t对象未在上述的addWaiter()中用到,只是用到了这个参数nodeX对象.
                return t;
            }
        }
    }
}
```

方法 (M1-2-图)-addWaiter/enq

----也可参考”源码-业务总体流程案例图”章节。

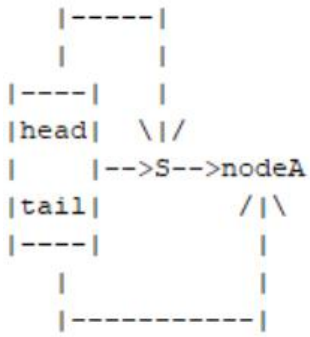
----上述 enq()中 if 部分初始化节点案例:此时的 CLH 队列格式为:



----上述 enq()中 if 部分初始化节点案例:此时的指针指向为:

tail->head->S

----上述 enq()中 else 部分单个节点案例:此时的 CLH 队列格式为:



S(thread=null,prev=null,next=nodeA)

nodeA(thread=nodeA,prev=S,next=null)

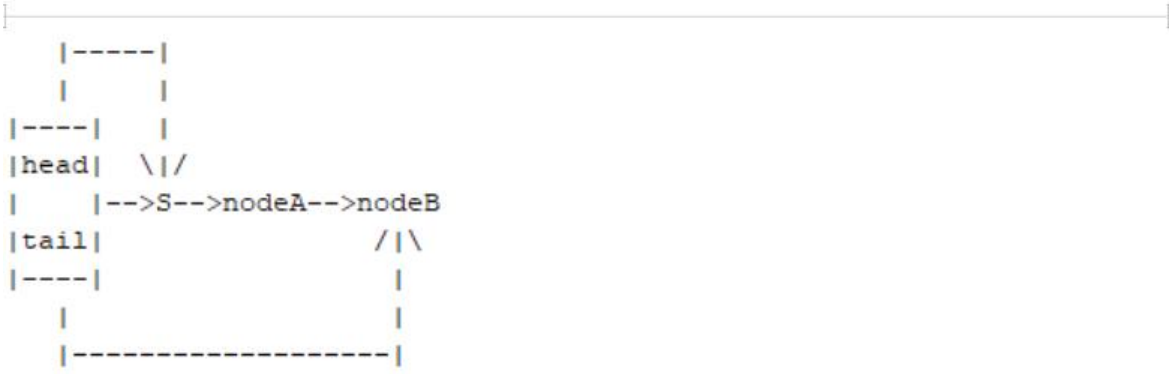
----上述 enq()中 else 部分单个节点案例:指针指向

nodeA.prev->t->tail->S
经 CASTail(t, node)
tail->nodeA
nodeA.prev->t->S
S.next->nodeA

AQS中有两个属性head和tail,分别用来保存AQS队列的头尾节点。
初始时，这两个属性都为null，当有一个线程进来时，队列如上。

上述队列表示，在节点nodeA插入队列之前，没有其他节点在排队。
注意，当前持有锁的线程，永远不会处于排队队列中。这是因为，当一个线程调用lock()获取锁时，有两种情况:
--1)锁处于自由状态(state==0)，且它不需要排队。通过CAS立即获取到了锁，这种情况，显然它不处于排队队列中。
--2)锁处于非自由状态，线程加入到了排队队列的队头(不包括head)等待锁。
将来某一时刻，锁被其他线程释放，并唤醒这个队列中的线程，线程唤醒后，执行tryAcquire，获取了锁，然后重新维护排队队列，将自己从队列移除(acquireQueued()).

----上述 addWaiter 第一部分多个节点案例:此时的 CLH 队列格式为:
此队列表示在节点 nodeB 插入之前，已经有一个节点 nodeA，在排队时已经正在排队获取锁的过程中了。



----上述 addWaiter 第一部分多个节点案例:指针指向

nodeB.prev->t->tail->nodeA
经 CASTail(t, node)
tail->nodeB
nodeB.prev->t->nodeA
nodeA.next->nodeB

方法 (M1-3)-acquireQueued

所属类:AQS
//线程真正加入CLH队列、队列重置并返回线程中断标识interrupted.
//参数:node-addWaiter()返回的尾节点指向的最后一个加入的nodeX(Node.EXCLUSIVE)对象
//参数:arg-1
//注意和doAcquireInterruptibly()的对比,二者主要区别在于发现线程被中断过后的处理逻辑.

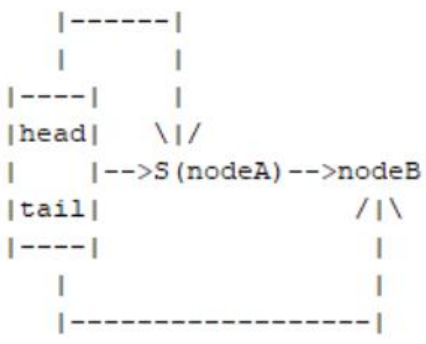
```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        //线程中断状态
        boolean interrupted = false;
        //注意:自旋一直跑着呢
        for (;;) {
            //获取nodeX的上一个节点prev
            final Node p = node.predecessor();
            //判断是否为head头节点对象,并通过当前线程再次获取一次锁(回看tryAcquire)
            //如果此时只有哨兵节点和一个后续的nodeX节点,即只有首节点(不包括哨兵节点),tryAcquir
成功,那么直接进入此if.
//这里可能会出现,比如nodeX刚释放了锁又再次抢到锁的情况.
            if (p == head && tryAcquire(arg)) {
                //移除当前哨兵节点S并GC回收,并把nodeA设置成哨兵节点S,nodeA后续的下一个节点nex
指向nodeX不变.
                //要注意经tryAcquire()处理后,此时的state=1,ownerThread=nodeA,nodeA就从此队列
失了,从而绑定了lock锁对象.
                setHead(node);
                //可以把这个GC回收的原哨兵节点S对象当成执行unlock()时处理的node对象.
                //前提是,有且仅当unlock()的线程作队列的节点中生效(存在CLH队列时生效)(因为第一次
获取锁成功时并不存在CLH队列,当然也不会存在各种node节点指向)
                p.next = null; // help GC
                failed = false;
                //注意:这里要返回,返回当前线程的中断状态,用于后续操作判断当前线程的是怎么被唤醒的.
                return interrupted;
            }
            //获取锁失败后设置当前节点nodeX的上一个节点prev的waitStatue为-1(包括哨兵节点)(-1
示节点被阻塞了)
            //只有发现当前节点不是首节点才会返回true
            if (shouldParkAfterFailedAcquire(p, node) &&
                //park()阻塞当前线程并在unpark()唤醒时返回当前线程的中断状态.
                //要注意的是:最终唤醒的一定是队头(并非哨兵节点,即一定是哨兵节点的后续next节点),被
醒后再次自旋进入最上面的if()处理新的队列逻辑.
                //这里不用考虑多次park()或unpark()的问题,因为这个LockSupport加锁解锁的许可证最
是1.
                parkAndCheckInterrupt()){
                //该线程是被中断唤醒并获取的锁,于辅助后续操作判断当前线程是被中断唤醒的.所以需要
新设置下中断位.
                interrupted = true;
            }

        }
    } finally {
        //注意这里最终都会执行别忘记了
        if (failed){
            //如果该方法因为某些特殊情况意外的退出(没有获取锁就退出了),那么就取消尝试获取锁.
            //设置节点状态为CANCELLED
            cancelAcquire(node);
        }
    }
}
```

方法 (M1-3图)-acquireQueued

----也可参考”源码-业务总体流程案例图”章节。

----上述 acquireQueued 第一个 if 案例:图形结构
此队列表示在节点 nodeB 插入之前，已经有一个节点 nodeA，在排队时已经正在排队获取锁的过程中了。



----上述 acquireQueued 第一个 if 案例:指针指向

head->S
tail->nodeB
nodeB->A
经 CASTail(t, node)
head->S(nodeA)
tail->nodeB
nodeB->S

方法 (M1-3-1)-setHead

```
所属类:AQS
//移除当前哨兵节点S并GC回收,设置nodeX为新的哨兵节点S.
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

方法 (M1-3-2)- predecessor

```
所属类:AQS$Node
//获取nodeX的上一个节点prev(可能会跟在addWaiter()中设置的有出入,即期间有节点位置变化)
final Node predecessor() throws NullPointerException {
    Node p = prev;
    if (p == null)
        throw new NullPointerException();
    else
        return p;
}
```

方法 (M1-3-3)-shouldParkAfterFailedAcquire

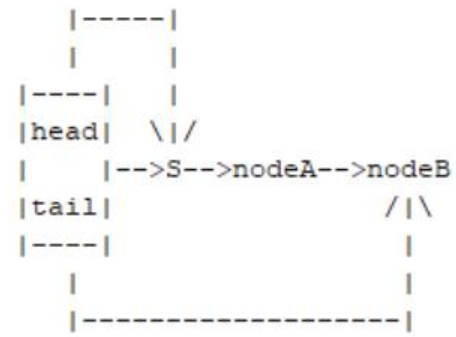
所属类:AQS

```
//获取锁失败后处理当前节点nodex的上一个节点prev的waitStatue为-1(-1表示节点被阻塞了)
//在for自旋中.
//pred-nodeX的前置节点,node-nodeX.
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    //判断当前节点nodex的上一个节点prev的waitStatue=-1
    if (ws == Node.SIGNAL)
        return true;
    if (ws > 0) {
    //ws==1,即CANCELLED,表示pred节点已经被取消了,将取消的pred节点从队列中移除,重新维护
排队链表关系.
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        //上一个节点的下一个节点=当前节点
        pred.next = node;
    } else {
    //程序第一次执行到这里返回false,还会进行外层第二次循环.
        //设置当前节点nodex的上一个节点prev的waitStatue=-1(包括哨兵节点).(-1表示节点被阻塞了)
        //没有condition的情况下,且是第一个加入到等待列表.如果在这期间头结点没有变的话,就把当
节点waitStatus设置为-1状态.
        //为何每个线程进入该方法后,修改的是上一个节点的waitStatus,而不是自己修改自己的?
        //因为线程调用park()后,无法设置自己的这个状态,若在调用park()前设置,存在不一致的问题,所
每个node的waitStatus,在后继节点加入时设置.
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

方法 (M1-3-3图)-shouldParkAfterFailedAcquire

----也可参考”源码-业务总体流程案例图”章节。

----如果此时的 CLH 队列格式为下述格式:
那么 nodeA 的 waitStatus 等于-1,依次往后,而 nodeX(即最后的 node 的 waitStatus 等于 0,前面 node 的 waitStatus 都等于-1)



方法 (M1-3-4)-parkAndCheckInterrupt

所属类:AQS

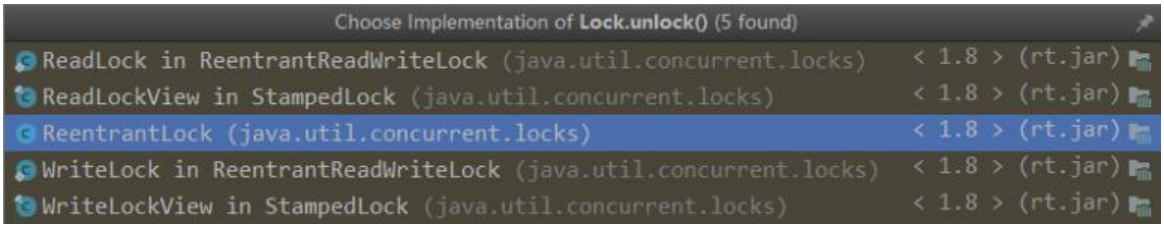
```
//真正加入CLH队列.park()阻塞当前线程并在unpark()唤醒时返回当前线程的中断状态.
private final boolean parkAndCheckInterrupt() {
    //park被唤醒的条件有:1.线程调用了unpark,2:其它线程中断了线程;3:发生了不可预料的事情
    LockSupport.park(this);
    //有两种情况,会导致阻塞结束:
    //1)持有锁的线程,释放锁后,将这个线程unpark()了.要注意的是:最终唤醒的一定是队头(并非哨兵节
),被唤醒后再次自旋处理.
    //2)线程被interrupt().(注意,在外部interrupt这个线程,不是抛出InterruptException,这一点和slee
及wait都不同)
        /**
        parkAndCheckInterrupt()判断Thread.interrupted()线程是否会挂起.
        因为LockSupport.park挂起的线程不仅会被LockSupport.unpark方法唤醒，还会被中断唤醒
所以线程被唤醒后调用了下Thread.interrupted()方法来返回下当前线程的中断状态，
        但是该方法会清楚掉线程的中断状态，所以在acquire()又一次的通过selfInterrupt()设置了下中
状态。
        这也是ReentrantLock区别于synchronized关键字的一个地方。
        ReentrantLock支持等待队列中的线程中断，synchronized不支持。
        */
    return Thread.interrupted();
}
```

解锁-lock.unlock()

方法调用顺序

unlock->release->tryRelease->unparkSuccessor

方法 -unlock()




```
所属类:ReentrantLock
//释放锁不区分非公平锁和公平锁
public void unlock() {
    sync.release(1);
}
```

方法(M2)-release

```
所属类:AQS
//释放当前线程的锁
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0){
            //当unlock()的线程作为队列的节点中生效(存在CLH队列时生效):唤醒当前线程的下一个线程
点
```

```
            unparkSuccessor(h);
        }
        return true;
    }
    return false;
}
```

方法 (M2-1)-tryRelease

```
所属类:ReentrantLock$Sync
//释放当前线程的锁
//release:释放数量
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    //当前执行unlock()的线程一定要和持有锁的线程是同一个才可以释放锁.
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    //c==0,直接释放锁,否则当前线程直接释放此线程可重入的数量.
    if (c == 0) {
        free = true;
        //设置当前线程持有的锁线程为null
        //要注意的是,持有锁的对象就是Lock本身,无论多少个线程来抢都不用关系这个Lock对象的GC
收问题.
        setExclusiveOwnerThread(null);
    }
    //设置当前线程持有的状态,0表示可以唤醒队列中其他线程,否则释放可重入的数量.
    setState(c);
    return free;
}
```

方法 (M2-2)-unparkSuccessor

```
所属类:AQS
//唤醒当前线程的下一个线程节点
private void unparkSuccessor(Node node) {

    int ws = node.waitStatus;
    if (ws < 0){
        //如果当前线程waitStatus=-1,表示此节点是阻塞节点线程,那么CAS设置当前线程的waitStatus
0
        //在释放下个线程之前先把自己当前线程的waitStatus设置成0
        compareAndSetWaitStatus(node, ws, 0);
    }
    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        //unpark()释放当前线程的下一个节点线程
```

```
        //当unlock()的线程作为队列的节点中生效(存在CLH队列时生效)(因为第一次就获取锁成功时并存在CLH队列,当然也不会存在各种node节点指向)
        //可以把这个unlock()的当前线程理解成CLH队列中的哨兵节点S,在acquireQueued()中被GC回了,否则当前unlock()的线程一直存在.
        LockSupport.unpark(s.thread);
}
```