AQS底层实现

AQS的非公平锁与同步队列的FIFO冲突吗?

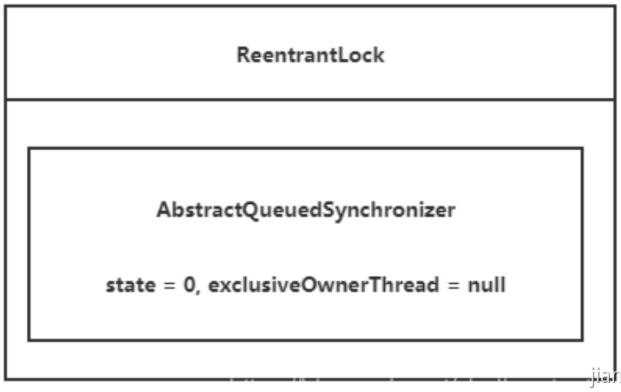
共享锁与独占锁的对比

AbstractQueuedSynchronizer，抽象队列同步器

JUC包下面很多API都是基于AQS来实现的加锁和释放锁等功能的，AQS是java并发包的基础类。

ReentrantLock、ReentrantReadWriteLock底层都是基于AQS来实现的。

ReentrantLock内部包含了一个AQS对象。



AQS定义两种资源共享方式：

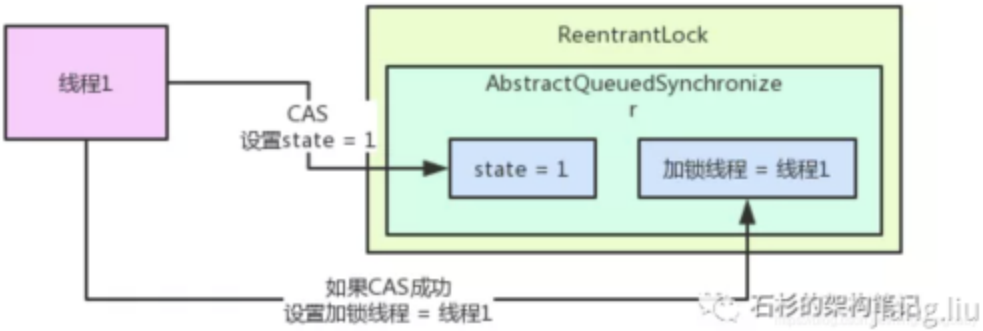
- **Exclusive（独占**，在特定时间内，只有一个线程能够执行，如ReentrantLock）
- **Share（共享**，多个线程可以同时执行，如ReadLock、Semaphore、CountDownLatch）

AQS对象内部有一个核心的变量叫做**state**，是int类型的，代表了**加锁的状态**。初始值为0

另外，这个AQS内部还有一个关键变量 **exclusiveOwnerThread**，用来记录**获得锁的线程**，初始化状态下，这个变量是null。

AQS中还有一个用来存储获取锁失败线程的队列，以及**head** 和 **tail** 结点。**是一个双向链表**

接着线程1跑过来进行加锁，这个加锁的过程，直接就是通过CAS操作将state值从0变为1。同时设置当前加锁线程是自己。



每次线程1可重入加锁一次，会判断一下当前加锁线程就是自己，那么他自己就可以可重入多次加锁，每次加锁就是把state的值给累加1。 **state+1 = 2，这就是可重入的核心原理！**

一旦state小于0表示可重入的次数大于int型最大值，产生溢出了。

接着线程2过来加锁，判断 exclusiveOwnerThread 发现不是自己加的锁，所以线程2此时就是加锁失败。

加锁失败后 ，此时就要将自己放入队列中来等待，等待线程1释放锁之后，自己就可以重新尝试加锁了。

> 如果等待队列里有其他的线程在等待了，就将自己的node设置为尾结点。**基于CAS的设置尾节点。并且通过“死循环”来保证节点的正确添加。**
> 如果之前的等待队列没有等待的线程，那么new一个node，让head和tail指向这个new出来的结点。
>
> 判断之前的结点是不是头结点 head，如果是头结点就尝试去获取锁，获取锁成功的话，就把当前线程设置为head
> 如果之前的不是头结点，那么就要等待了，等待过程中阻塞当前线程，并且通过**自旋**来一直尝试获取同步状态，只有前驱节点是头节点才能够尝试获取同步状态，等候之前的线程释放锁后，调用 LockSupport来唤醒。

线程锁释放：其实很简单 就是将 state-- 直到state = 0，则彻底释放锁，会将“加锁线程”变量也设置为null，然后通过 LockSupport.unpark()来唤醒等待队列中的下一个结点.

> 这块唤醒的是头结点的后继结点而不是头结点，
> 因为我们上面调用addWaiter方法的时候，如果等待队列里面没有等待线程，那么直接new 了一个Node 然后head 和 tail 都指向这个 node，换句话说这个头结点只是用来占位的，所以要从头结点的下一个结点开始唤醒。

ReentrantLock又分为**公平锁**和**非公平锁**。ReentrantLock是基于独占锁实现。

AQS的非公平锁与同步队列的FIFO冲突吗?

公平锁与非公平锁的含义都很明白，公平锁必须排队获取锁，锁的获取顺序完全根据排队顺序而来，而非公平就是谁抢到是谁的。但是！同步队列不是FIFO的吗，它们入队排好顺序并且按照顺序一个一个的醒来，只有醒来的才有机会去获取锁，才能去执行try方法，这不还是公平的吗?

解答：
线程在do方法中获取锁时，会先加入同步队列，之后根据情况再陷入阻塞。当阻塞后的节点一段时间后醒来时，这时候来了新的更多的线程来抢锁，这些新线程还没有加入到同步队列中去，也就是在try方法中获取锁。
在公平锁下，这些新线程会发现同步队列中存在节点等待，那么这些新线程将无法获取到锁，乖乖去排队；
而在非公平锁下，这些新线程会跟排队苏醒的线程进行锁争抢，失败的去同步队列中排队。
因此这里的公平与否，针对的其实是苏醒线程与还未加入同步队列的线程
而对于已经在同步队列中阻塞的线程而言，它们内部自身其实是公平的，因为它们是按顺序被唤醒的，这是根据AQS节点唤醒机制和同步队列的FIFO特性决定的。

共享锁与独占锁的对比

与AQS的独占功能不同，当锁被头节点获取后，独占功能是只有头节点获取锁，其余节点的线程继续沉睡，等待锁被释放后，才会唤醒下一个节点的线程，而共享功能是只要头节点获取锁成功，就在唤醒自身节点对应的线程的同时，继续唤醒AQS队列中的下一个节点的线程，每个节点在唤醒自身的同时还会唤醒下一个节点对应的线程，以实现共享状态的“向后传播”，从而实现共享功能。

CountDownLatch、emaphore、 ReentrantReadWriteLock都是共享锁的实现