参考：

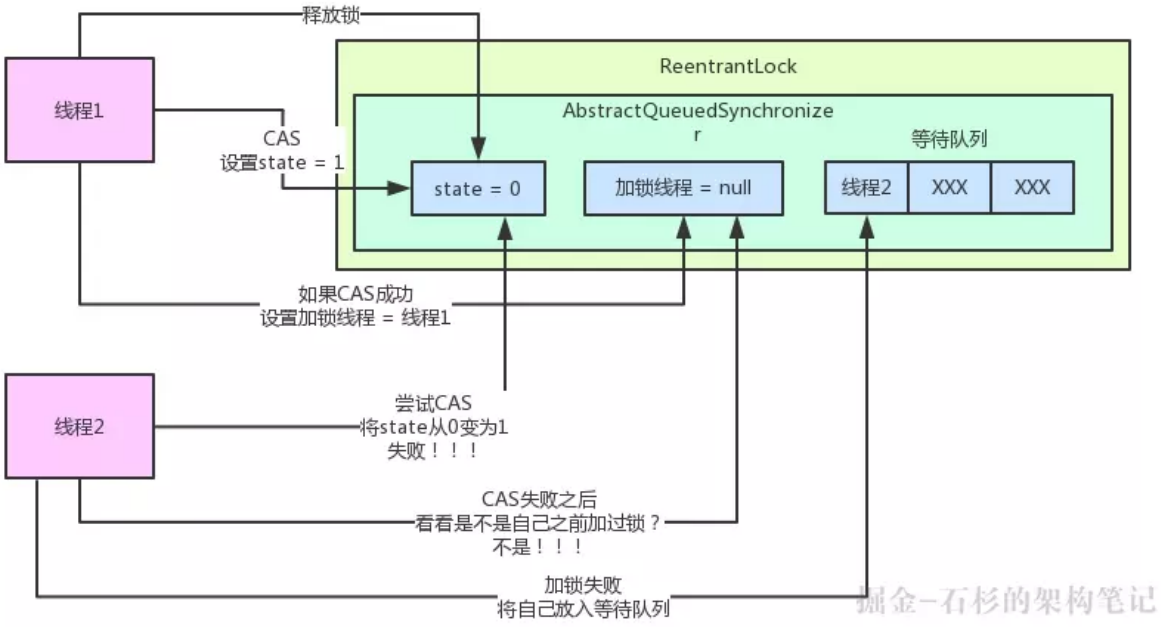
[大白话聊聊Java并发面试问题之公平锁与非公平锁是啥？【石杉的架构笔记】]

前言

AQS的笔记里面因为借用了ReentranLock做讲解，其中涉及了公平锁与非公平锁的区别，该篇单独对公平锁与非公平锁做一下讲解。

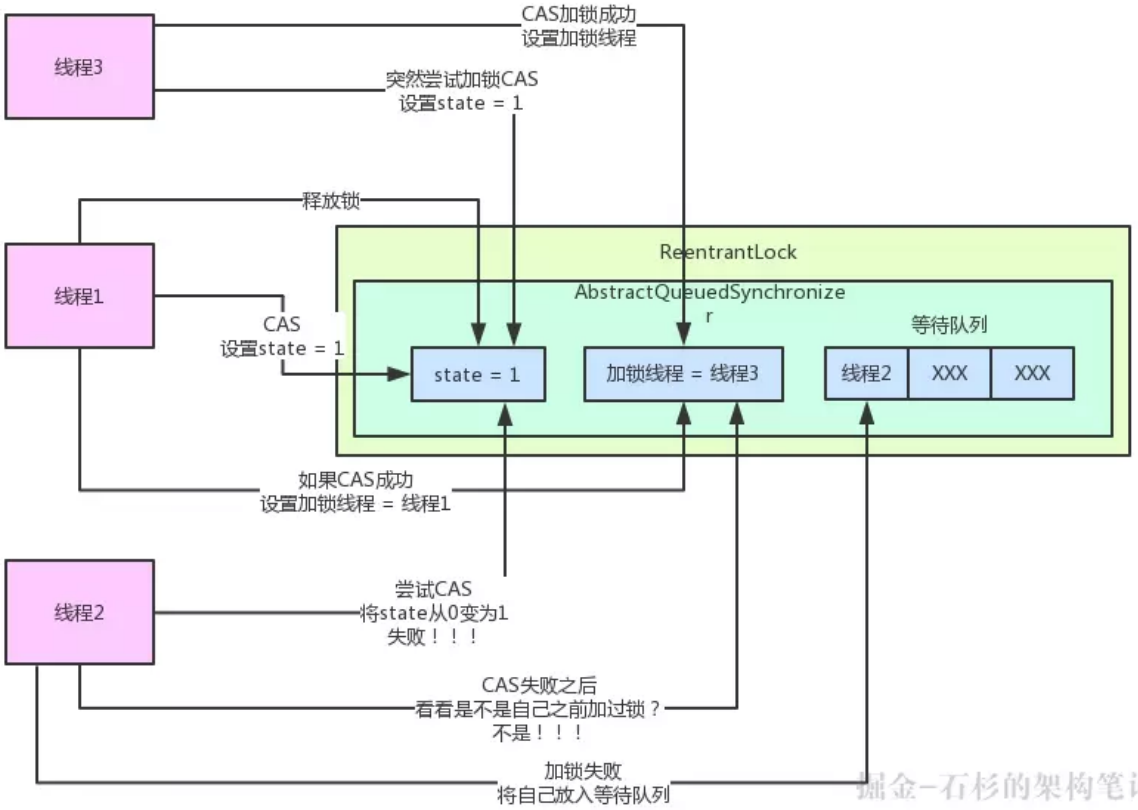
非公平锁

我们再把之前讲解CAS的图拿过来用一下。



我们来回顾一下，线程1通过CAS操作讲状态置设置为1并且加锁线程标记为线程1，通俗的讲线程1加锁成功了，这时候线程2妄图来加锁，显然失败了，但是失败不意味着被抛弃，线程2只是暂时存入等待队列。

好了，现在线程1解锁了，线程2感觉自己终于熬出头了，正准备走出等待队列尝试加锁，结果冒出了个线程3。

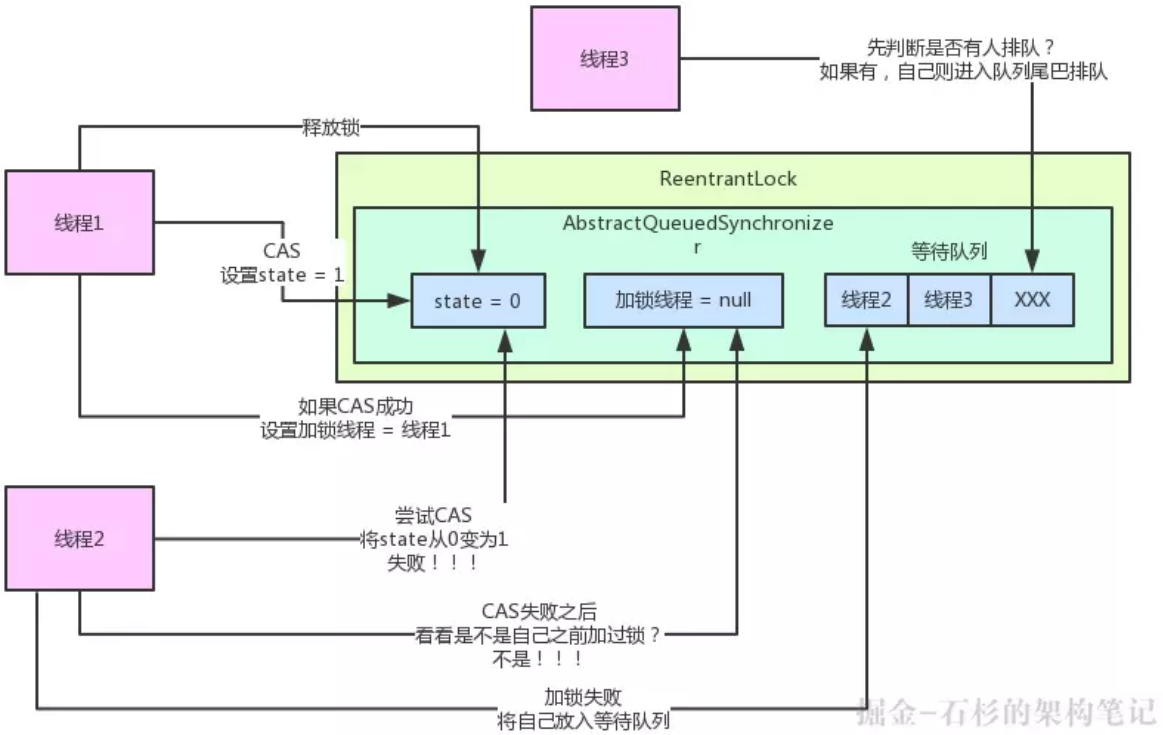


线程1刚解锁完，线程3也刚好来，结果线程3成功加锁，线程2失败只能继续呆着。这就是非公平锁的体现，没有先来后到，谁抢到归谁。

创建ReentranLock对象时默认就是创建的非公平锁。

公平锁

那显然，就是避免了非公平的插队现象。上张图



其实它对比非公平锁的区别就是在有新线程来的时候会先检测等待队列是否有等待线程，如果有就直接加入等待队列，没有才能直接去尝试加锁（这个在源码解析中用代码去讲解）。

源码解析区分公平锁非公平锁

上面再形象也只是大白话，程序猿还是要用代码说话的，具体代码里怎么区分的，AQS里我们提及过ReentranLock里有一个继承了AQS的抽象类Sync，其实其中还有两个子类又继承了Sync，就是区分公平锁与非公平锁的，从名字也可见一斑，NonfairSync和FairSync。

那么我们也讲过有一个操作state的重要方法tryAcquire，区分公平与非公平的关键就在这，NonfairSync的tryAcquire的实现代码。

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

为啥是nonfairTryAcquire方法，自己看前文去。

然后我们看看FairSync中的tryAcquire。

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            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;
}
```

红圈的地方就是关键处，这个方法判断了等待队列是否有线程。逻辑也不复杂，内容如下。

```
public final boolean hasQueuedPredecessors() {
    // The correctness of this depends on head being initialized
    // before tail and on head.next being accurate if the current
    // thread is first in queue.
    Node t = tail; // Read fields in reverse initialization order
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

多了一步判断性能自然也就降低了点，一搬情况下非公平的策略并不会有啥问题（线程2的感受其实不需要顾及，3先上还是2先上于总体来说没啥影响），所以说锁的策略默认都是非公平的。当然具体考量就看具体情况了。