参考：

[大白话聊聊Java并发面试问题之谈谈你对AQS的理解？【石杉的架构笔记】]

[深入浅出java同步器AQS（占小狼）]

前言

AQS是啥呢，他全名AbstractQueuedSynchronizer，可能不是太熟悉，但要提到锁，可能大多数人又有概念了。那么锁和AQS有什么关联呢，我们以最常用的锁ReentrantLock为例（本篇的讲解将均以ReentrantLock）来探究下锁与AQS的关联。

```
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    private final Sync sync;

    /**
     * Base of synchronization control for this lock. Subclassed
     * into fair and nonfair versions below. Uses AQS state to
     * represent the number of holds on the lock.
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = -5179523762034025860L;
```

可以看到，在我们的ReentrantLock中定义了一个内部类Sync继承了AQS。通读了下ReentrantLock源码，大致能够了解，ReentrantLock里面基本上都在基于AQS的子类在操作。

基本就得出这个结论，平常我们一直在用锁，类似ReentrantLock这样的，它们其实就是起到了一个封装的作用，真正底层在工作的就是AQS。

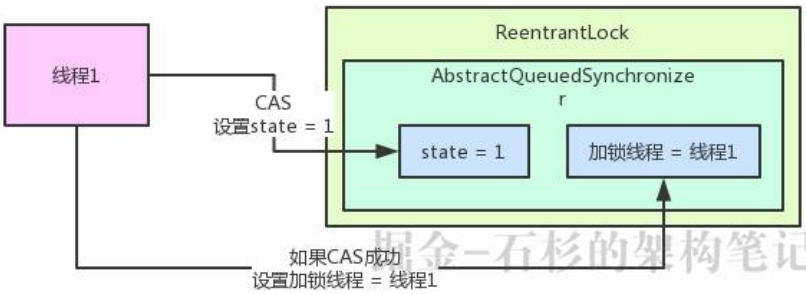
基于ReentrantLock理解AQS的加锁和释放锁的原理

长话短说，现在有一个线程过来尝试用ReentrantLock的lock()方法进行加锁， 会干些啥呢？在AQS中有一个核心的变量state，就代表了

加锁的状态，初始值为0。还有一个关键变量用来记录加锁的是哪个线程，初始值为null。

我们假设有个线程1，这时候它调用了lock()方法，它会直接用CAS操作将state的值从0变为1。（CAS操作是啥见相关笔记）

那么线程1就加锁成功了，此时应该是这样一个状态。

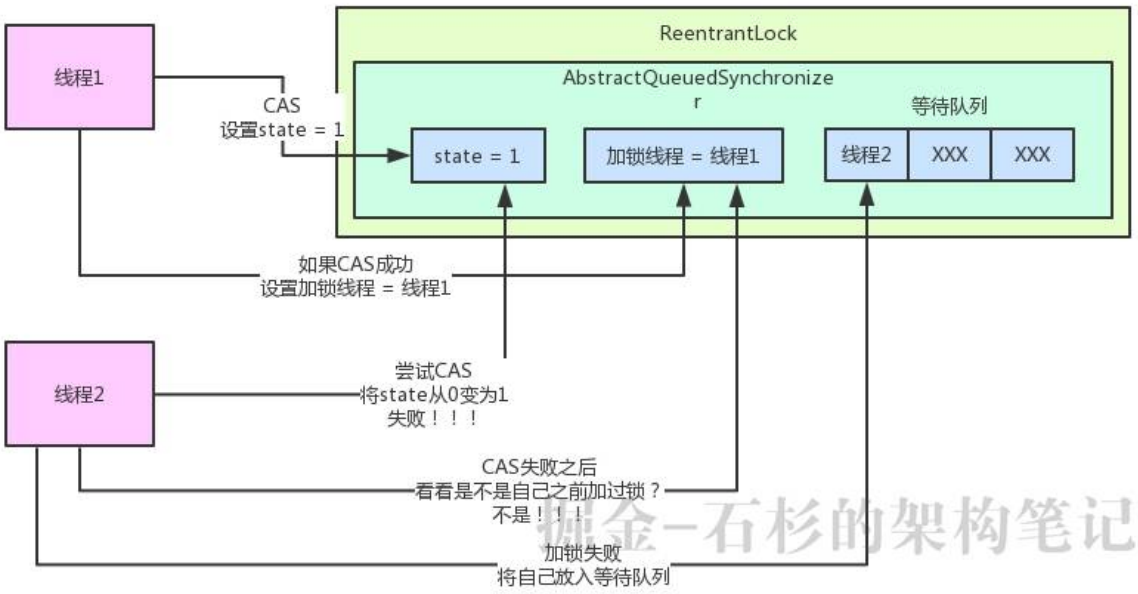


嗯，说白了，AQS就是并发包里的一个核心组件，里面有state变量、加锁线程变量等核心的东西，维护了加锁状态。

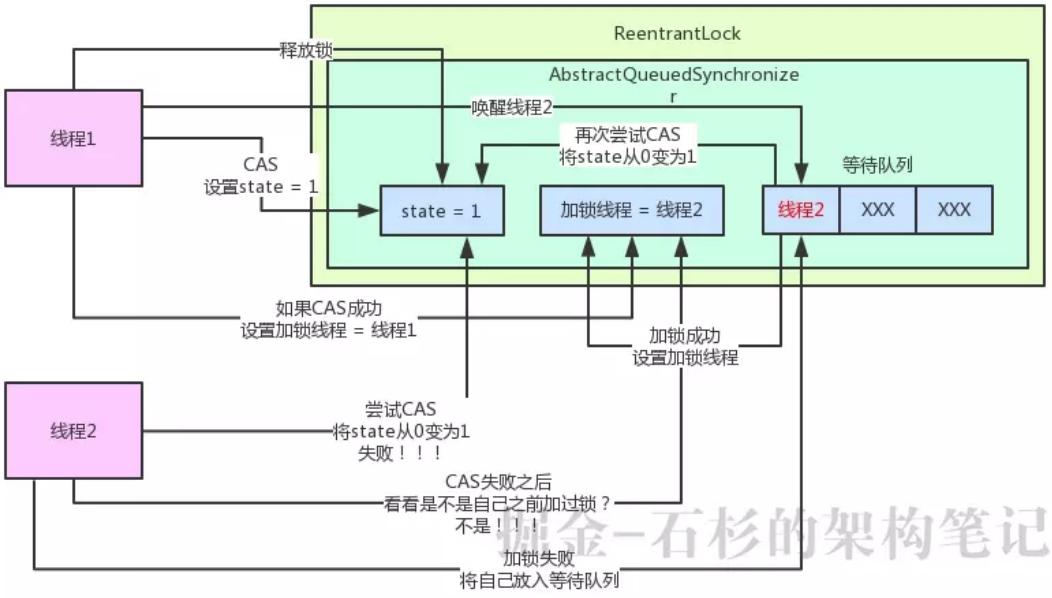
ReentrantLock之所以用Reentrant打头，意思就是他是一个可重入锁。可重入锁的意思，就是你可以对一个ReentrantLock对象多次执行lock()加锁和unlock()释放锁，也就是可以对一个锁加多次，叫做可重入加锁。大家看明白了那个state变量之后，就知道了如何进行可重入加锁！其实每次线程1可重入加锁一次，会判断一下当前加锁线程就是自己，那么他自己就可以可重入多次加锁，每次加锁就是把state的值给累加1，别的没啥变化。

嗯，线程1加锁成功还没演示完呢，我们需要线程2来实验下锁的威力，这时线程2也开始调用lock尝试加锁。那肯定是失败的，失败后会

咋样呢，放心，不会丢掉的，AQS提供了等待队列，这时候失败的线程2就会进去了。



这样我们就要释放线程1的锁了，线程1会咋样，线程2又会咋样呢？释放锁是个啥过程呢，其实就是将state减1，如果变为0就是彻底释放了，同时加锁线程也会设回null。这个时候就会去唤醒等待队列，我们的线程2就能去加锁了。大概就是这么个过程。



基于源码理解AQS

上面的解读总体来说还算是生动形象的，这里开始就可能要枯燥点了，深入到AQS的代码去看看这个流程是啥样的。

我们先来看看AQS中的主要内容定义。

```
public abstract class AbstractQueuedSynchronizer extends
    AbstractOwnableSynchronizer implements java.io.Serializable {
    //等待队列的头节点
    private transient volatile Node head;
    //等待队列的尾节点
    private transient volatile Node tail;
    //同步状态
    private volatile int state;
    protected final int getState() { return state;}
    protected final void setState(int newState) { state = newState;}
    ...
}
```

定义的head和tail就是等待队列了，而这个state就是核心的那个状态了，这个状态会通过CAS操作去累加和累减代表锁的状态 。

嗯，该怎么讲呢，算了，就从调用lock开始深入吧，以ReentrantLock为例，调用lock方法后，我们发现其实调用了ReentrantLock中写的AQS的子抽象类的lock。

```
public void lock() {
    sync.lock();
}
```

那么其实又调了ReentrantLock继承这个子抽象类中真正实现了lock方法的lock。我们发现其实最终就回到AQS中，其实整个的lock操作这一步正式算是开始了。

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

这一步啥意思呢，大概就是先tryAcquire进行尝试修改state，如果失败并且将线程加入等待队列，并让该线程中断。

那么tryAcquire就是去修改state的方法了。该方法AQS本身只抛出异常，具体实现要看子类的覆盖方法。

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

显然，这边调用了一个nonfairTryAcquire方法处理（这个非公平的前缀暂时不要纠结，这涉及到公平锁和非公平锁，等单独讲解），那我们就去看看这个方法。

```
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;
}
```

这里面干了什么呢，其实就和我们上述的图解差不多。先获取state，如果为0，那就直接CAS操作并设置加锁线程为当前线程，如果不是0就先判断当前线程是不是加锁线程，是则更新state值。

那么和图解的过程一样。尝试加锁失败了这个线程肯定不会丢掉，会把它塞到等待队列中，这就是addWaiter发挥作用的地方。

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

至于外面还套着的一层用了acquireQueued方法还有待研究。

那lock完接下来就是解锁操作了。我们会调用unlock方法，深入后第一个重要分支就是AQS中这个方法。

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

这里面首先就会调用一个与上文tryAcquire对应的方法tryRelease。它的实现实在ReentrantLock中。

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

首先必须确保当前操作线程与加锁线程一致，但不代表是加锁线程我执行这一次就结束了，因为这边是可重入锁ReentrantLock，所以可以lock许多次，那相对应的也得unlock同样的次数才能unlock完全，所以这边只有获取到的state在操作完为0时才最终确定unlock完毕，会将加锁线程移除。

回到之前的release方法，就清晰了，只有完全unlock后才会进入具体逻辑，干了啥呢，值得关注的其实就一点，取得头节点head后做了个uparkSuccessor操作，我们来看看。

```
private void unparkSuccessor(Node node) {
    /*
     * If status is negative (i.e., possibly needing signal) try
     * to clear in anticipation of signalling.  It is OK if this
     * fails or if status is changed by waiting thread.
     */
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    /*
     * Thread to unpark is held in successor, which is normally
     * just the next node.  But if cancelled or apparently null,
     * traverse backwards from tail to find the actual
     * non-cancelled successor.
     */
    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)
        LockSupport.unpark(s.thread);
}
```

总得来说，最重要的就是最后一句话，LockSupport.unpark(s.thread)；LockSupport是JDK中比较底层的类，用来创建锁和其他同步工具类的基本线程阻塞原语。这个unpark的作用就是让指定线程可用。之前不是说了如果竞争失败的线程会先进入等待队列嘛，这步的操作就相当于唤醒等待队列中的线程了。