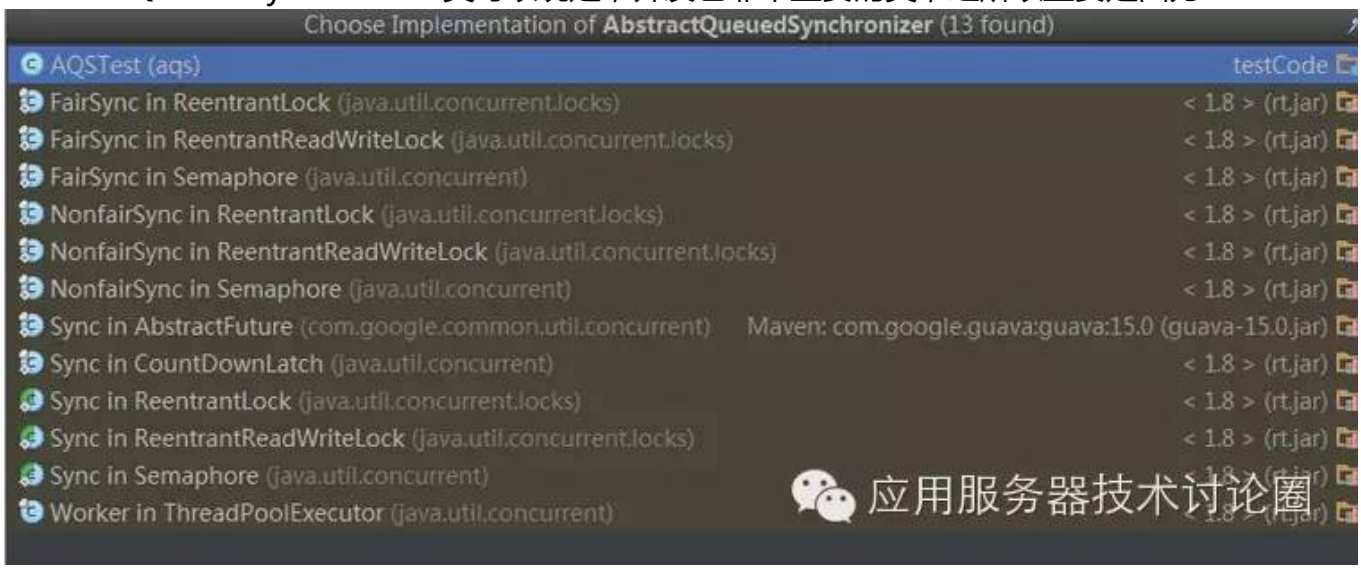
最不能忽略的AbstractQueuedSynchronizer类源码分析（必看）

2015-10-20 03:06 feiying 0 阅读 85

1.AbstractQueuedSynchronizer（AQS）类不能被忽略的重要性

AbstractQueuedSynchronizer类可以说是，并发包非常重要的类，之所以重要是因为：

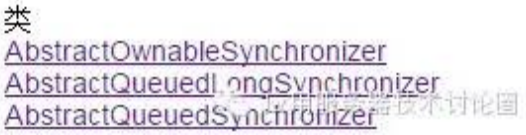


在上述的类图中，它占据着撑起并发工具包的半壁的江山，

这个类是ReentrantLock, ReentrantReadWriteLock,Condition,CountDownLatch,Semaphore，

ThreadPoolExecutor,AbstarctFuture,FutureTask等类的底层实现，可以这么说，研究清楚这一个类，并发工具包基本可以懂一大半；

首先来看一下AQS的类的继承关系，在lock包中的AQS一共有三个类，分别为：



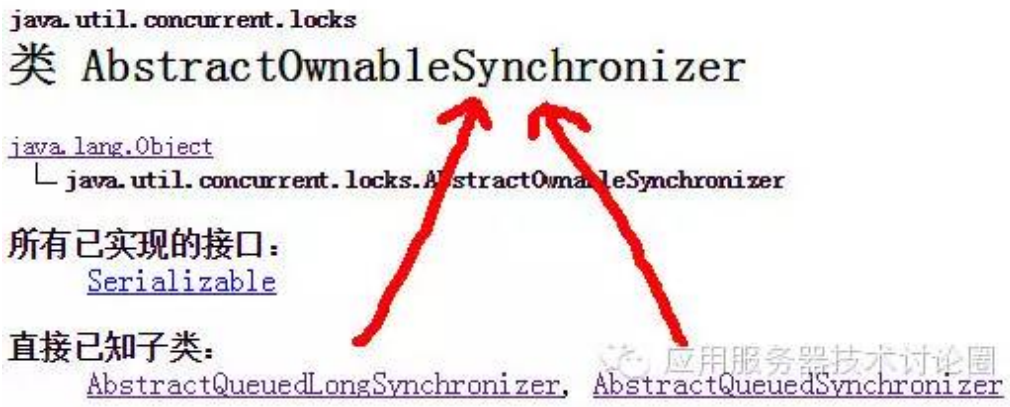
其实AQS还有一个Long的版本，也就是AbstractQueuedLongSynchronizer，这个类和AQS的区别是一些属性和内部类的字段不是int的，

以Node节点的state为例，是long的

以 long 形式维护同步状态的一个 AbstractQueuedSynchronizer 版本。此类具有的结构、属性和方法与 AbstractQueuedSynchronizer 完全相同，但所有与状态相关的参数和结果都定义为 long 而不是 int。当创建需要 64 位状态的多级别锁和屏障等同步器时，此类很有用。

可以总结的一点是AbstractQueuedLongSynchronizer就是AQS的64位的版本；

对于AbstractOwnableSynchronizer类，这个类是其父类，如下面的类图所示



这个AbstractOwnableSynchronizer父类的作用就是接口，它是提供了一个线程独占的方法：

对于这个类的子类，也就是AQS（或者是AbstractQueuedLongSynchronizer），才是做出一些数据结构，监视，上锁这些操作，

对于AQS是如何使用的，在其它的并发工具包都已经分析过了，这里就不再缀余；

下面主要是更细粒度的分解一下AQS的源码。

```
 */
public class ConditionObject implements Condition, java.io.Serializable {
    private static final long serialVersionUID = 1173984872572414699L;
    /** First node of condition queue. */
    private transient Node firstWaiter;
    /** Last node of condition queue. */
    private transient Node lastWaiter;
```

2.AQS的数据结构与线程操作原理解析

AQS实质就是一个队列，队列中的每一个Node就是一个节点，只不过这个Node比较特殊，它一般会装载具体的线程，

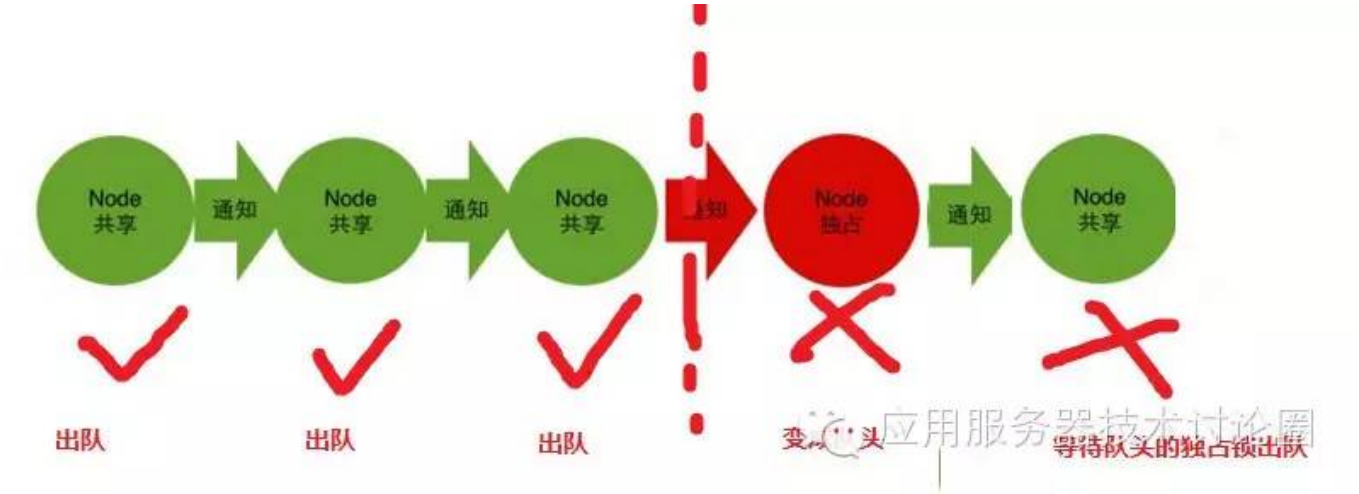
如果该线程没有到出队的时机，那么它会一直进行阻塞，直到Node节点到了时机；

那什么是时机呢？

Node节点中有标识能识别是当前Node节点再等待什么锁，具体如下面的源码：

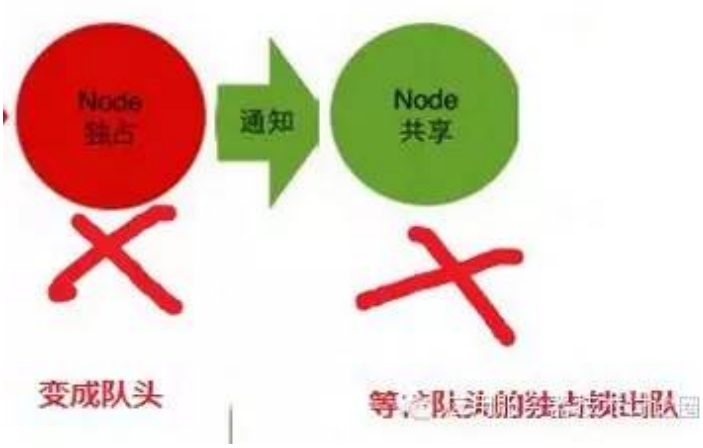
```
*/
static final class Node {
    /** Marker to indicate a node is waiting in shared mode */
    static final Node SHARED = new Node();
    /** Marker to indicate a node is waiting in exclusive mode */
    static final Node EXCLUSIVE = null;
```

例如下面的队列，第一个Node当为共享锁的时候，Node队列从出队位置一直传导到整个队列中，



一直遇到为独占模式的节点，这个时候上图中的前三个Node节点代表的线程恰好都是等待共享锁的，所以都出队，

而当这三个线程执行完毕，独占Node节点等待到线程执行的时机，这个独占线程才开始出队；

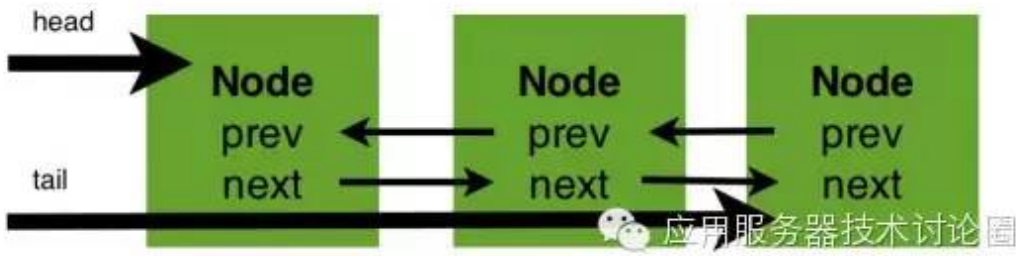


对于这个模式下，只能有一个线程独占，因此往后传播也没有用（代码中独占模式中也没有传播）

一直等到独占模式结束，下一个Node节点才继续重复上述的模式；

来看一下Node节点的数据结构：

prev是节点的前任，next是节点的后续，Thread是当前Node节点对应的线程，如下图所示

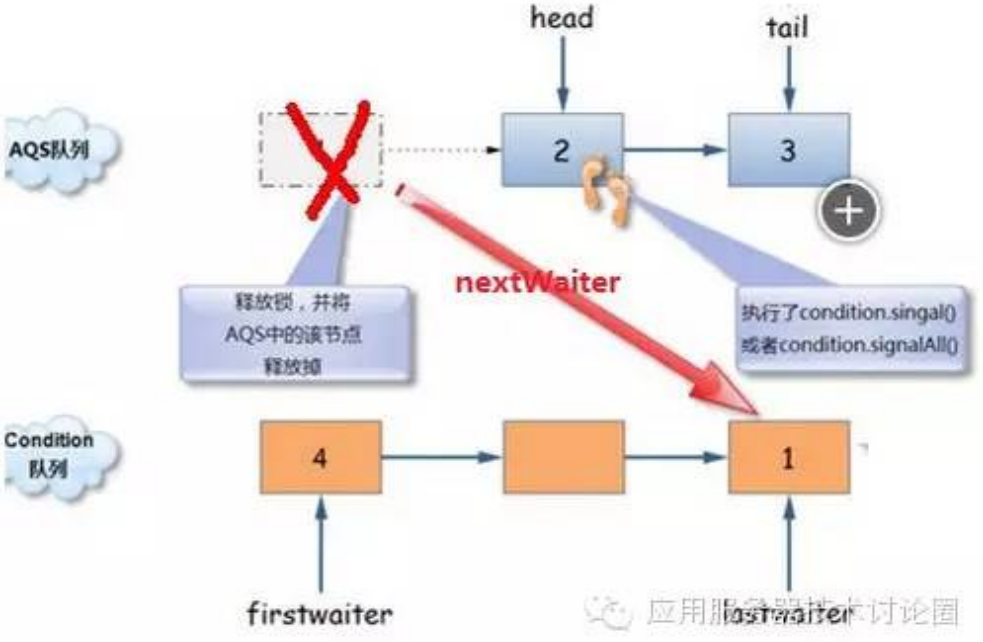


head和tail是AQS的两个指针，指向整个队列的头Node，和尾Node，其作用就是遍历；

上述属性中的nextWaiter，这个是给Condition类用的，Condition在源码分析的时候，其有两个队列

一个队列是AQS的队列，也就是上面讲述的这个，

另外一个是Condition队列，当调用Condition.await的时候，将这个Node节点加u到这个AQS队列中



如上图所示，当节点出队后，也就是调用Condition.signal的时候，该节点通过这个nextWaiter的指针，将该节点加入到AQS队列的尾部

这个nextWaiter的作用就是找到Condition队列中的Node节点的；

waitstatus是Node节点的状态

表示节点的状态。其中包含的状态有：

1. CANCELLED，值为1，表示当前的线程被取消；
2. SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark；
3. CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中；
4. PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；
5. 值为0，表示当前节点在sync队列中，等待着获取锁。

这里不应该混淆的是，state是整个AQS队列的状态，代表着整个队列进入独占模式，还是共享模式，

而这里应该叫节点的等待状态，与对应节点相关的；

AQS队列中Node节点持有的Thread对象，因此其线程操作都是采用LockSupport工具类来做的，

例如看到，当时机到了，需要唤醒下一个Node节点的线程，那么就需要调用LockSupport

```
/**
 * Wakes up node's successor, if one exists.
 *
 * @param node the node
 */
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);
}
```


首先判断waitstatus的状态，最后通过LockSupport进行释放线程；

LockSupport实质上就是一个object,wait的Thread线程版本的再封装，屏蔽对象和锁，就是直接操控线程Thread休眠和直接唤醒哪一个线程提供了方便的方法；

3.独占锁acquire/release方法解析

对于获取所acquire来说：

整体思路：

a.首先,调用回调函数tryAcquire（通常是子类实现），看看是否这把锁直接就能获得到，如果获得到直接return；

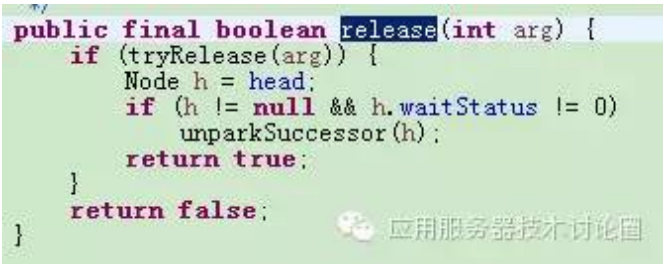
b.如果获得不到的话，按照规矩调用addWaiter方法，将独占锁加入到队尾

c.调用acquireQueued方法，判断当前线程的Node == head，如果不是，调用park方法，让当前线程进入休眠状态，等待唤醒；

当当前的Node是头的时候，再次回调tryAcquire，子类业务逻辑修改整个AQS队列的state的状态，

设置为头节点，next指针==null，方便GC，返回执行对应的业务逻辑；

release锁就很简单了：



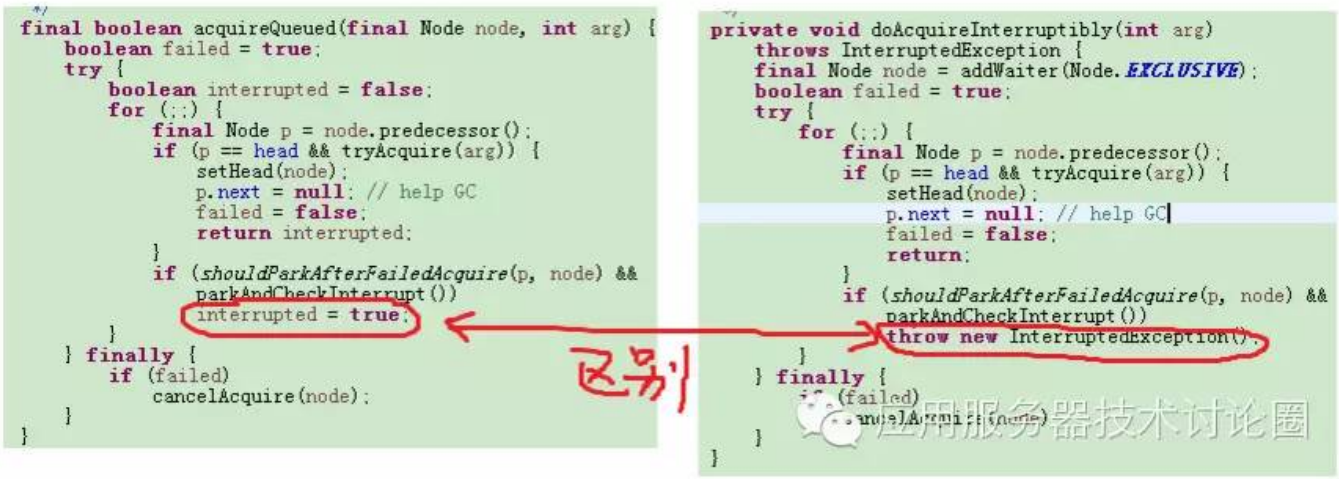
a.首先回调父类的tryRelease方法修改一下状态，一般这个都会成功，不像获取那么需要等待

b.调用前面分析过的unparkSuccessor方法，修改waitstatus，因为通过Node可以拿到对应的Thread，

最后调用LockSupport.unpark可以对该线程进行唤醒；

独占锁的思路就是如此，对于上述是基本的实现，对于独占锁还有两个其余的方法，大体上和acquire类似

一个是带中断的acquireInterrupt，对比如下：



上述的唯一区别就在于当parkAndCheckInterrupt方法中，这个方法最终返回的是

当前线程是否为中断；如果当前线程在执行过程中，出现一些异常，或者一些cancel等方式导致线程运行不下去了

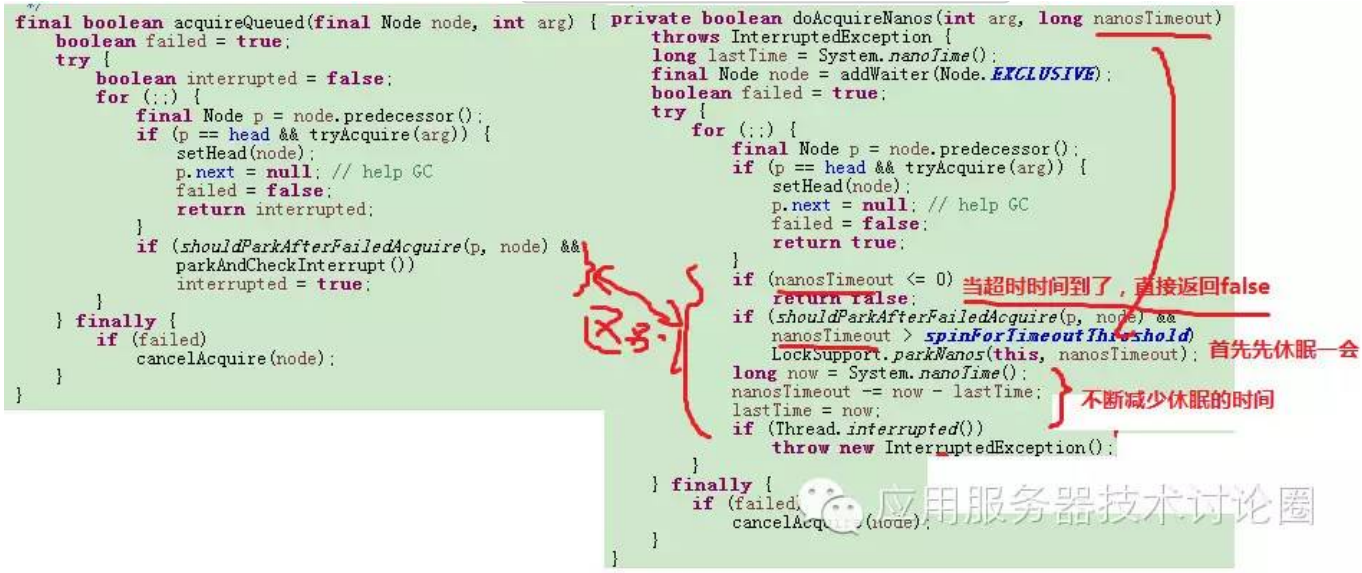
如果是正常的（如上述左图中所示），那么直接将interrupted属性置为true而已，也就是标识置为true，但是Node仍然会排队，等待锁

直到拿到锁之后，调用finally中才发现，是失败的，进行cancel；

而看到右图，也就是当检测到线程已经中断后，直接抛出异常，提前返回，

这么做的好处是，省略了无用的Node等锁的后续时间，因为等待锁也是无意义的，线程已经中止了；

另外一个版本，就是带有超时时间的版本：



具体的和aquire的对比如上图所示，这个超时时间的参数意思是，加上代表如果超过超时时间的话，直接返回false，步骤如下：

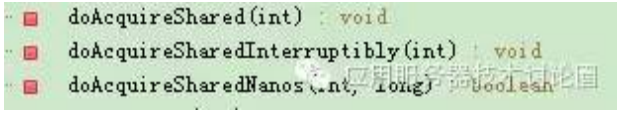
首先，调用LockSupport.parkNanos，让当前线程休眠足够的秒数，

然后，比对当前的超时时间，如果确实过了超时时间的话，直接返回false，返回给客户端，告诉已经超时了，CAS不会再做了，

否则，还继续CAS空转，一直到超时时间到位置，再返回false;

4.共享锁tryAcquireShared/tryReleaseShared方法解析

对于共享锁的acquire，可以看到也是带有三个方法，获得共享锁的方法，带有中断的版本，带有超时时间的版本：



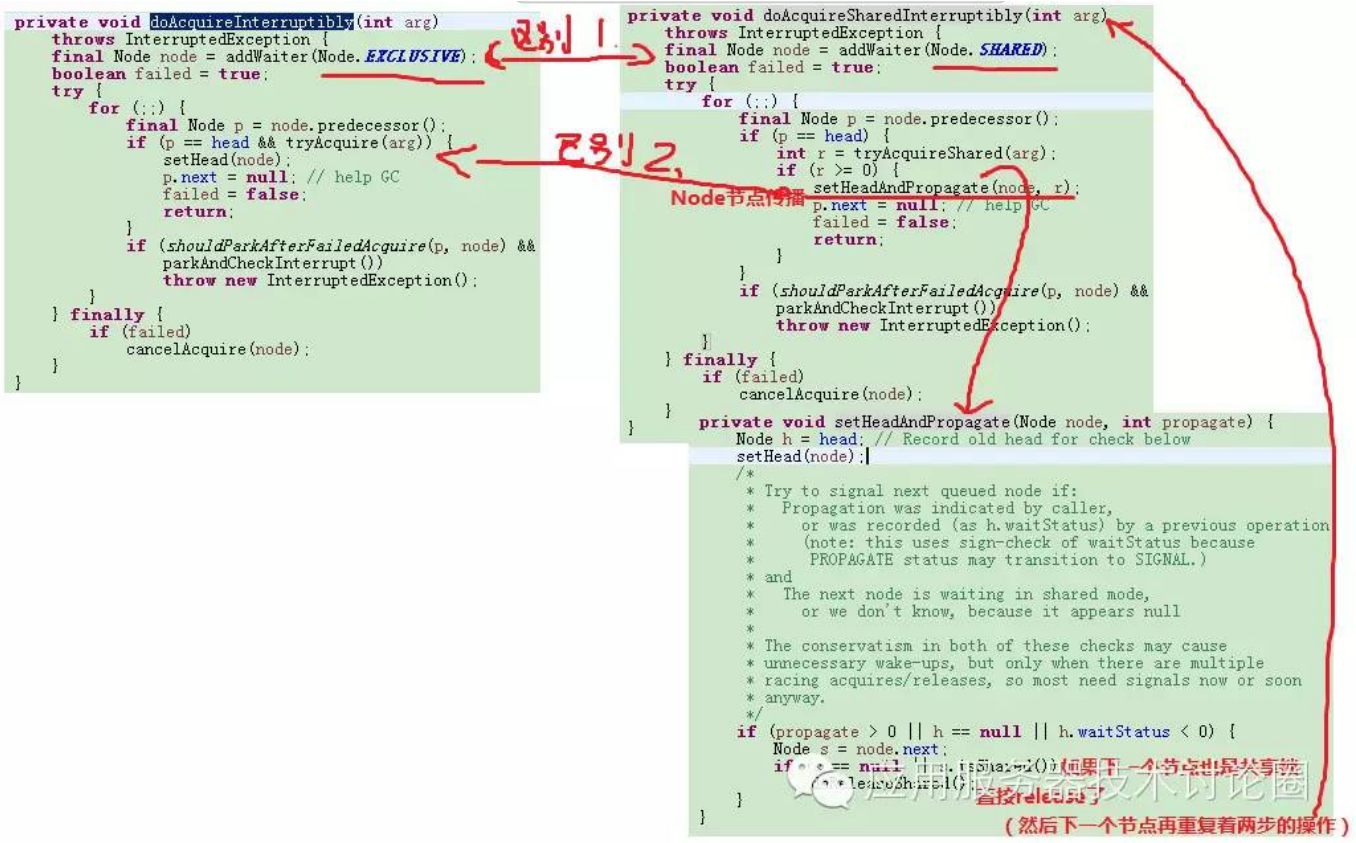
和独占锁其实区别不是很大，

主要的区别：

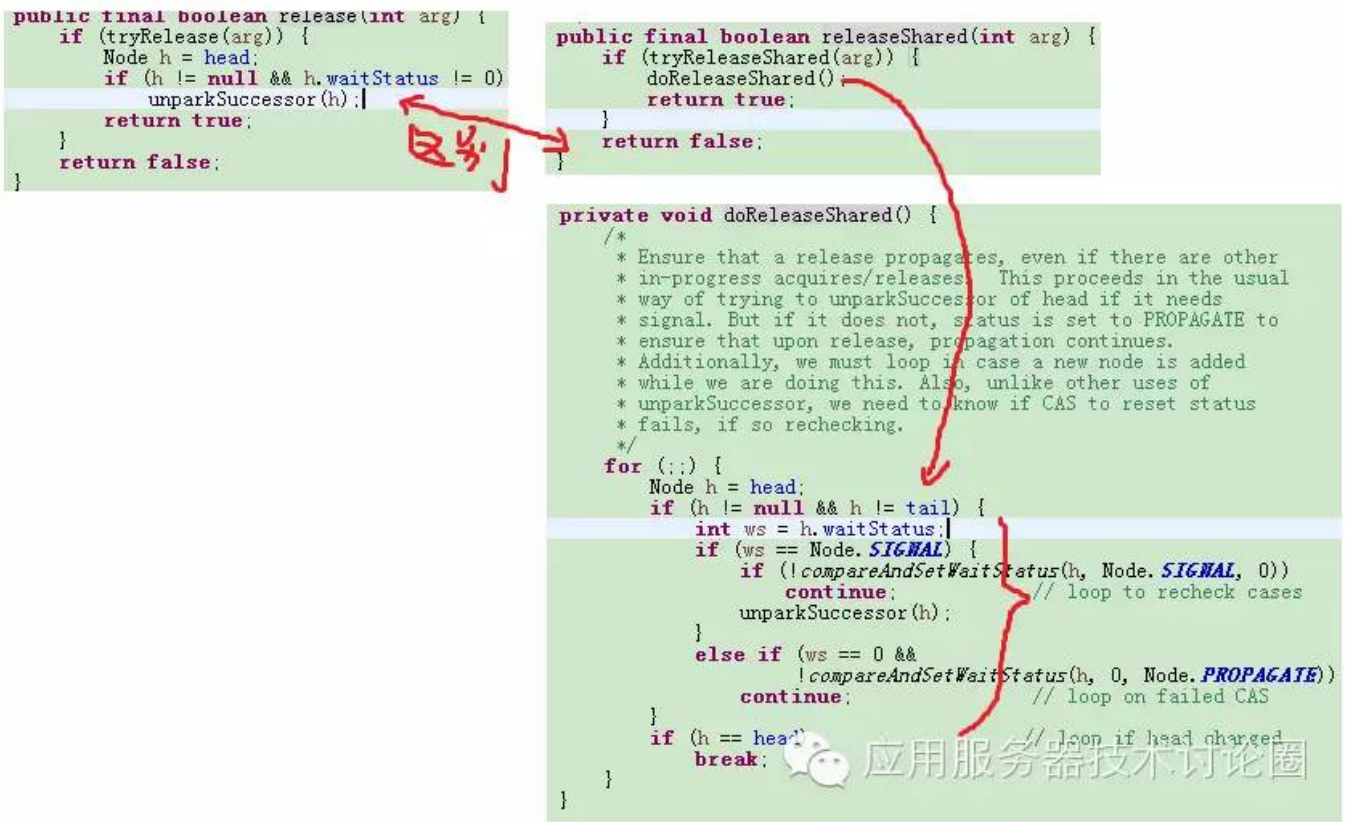
一个是锁的类型，

一个是共享锁特有的传播的特性，

以两种锁的中断的版本进行对比：



对于释放锁，二者的对比为：

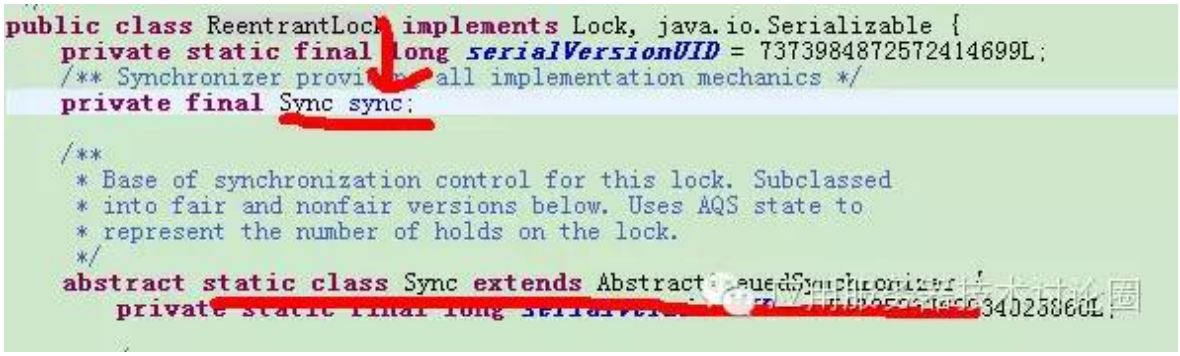


对于独占锁，因为就是队头的操作，所以直接调用unparkSuccessor这个方法即可；

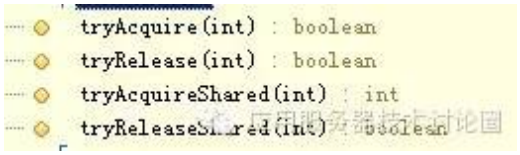
对于共享锁，因为有传播的问题，所以还需要进行更加细致的判断才行，如右图的状态变迁和CAS的对于status的状态的操作；

5.回调函数tryAcquire/tryRelease方法，state属性，与Sync内部类

对于AQS队列来说，一般的用法是在并发工具中声明一个内部Sync同步类，这个同步类继承的是AQS，如ReentrantLock所示：



对于上述分析的acquire/release独占锁，或者共享锁的过程中，发现多次调用了tryXX的方法，



这四个方法都是protected的，也就是需要作为回调函数，在AQS的子类中，根据自己的业务逻辑进行自定义的；

这几个方法无非都是做一件事，通过自己的业务逻辑修改自己的state属性，

对于State状态，是AQS队列实现的重头戏，这个状态实质是推动着整个队列出队的时机；

a.以ReentrantLock为例，因为就是独占锁，不存在传播，所以state就是0，1两种状态，

当0的时候，说明没有锁，那么队列中开始出队，如果是1，那么正在出队中；

b.对于ReentrantReadWriteLock中的Write锁和ReentrantLock一样，对于Read锁就是AQS队列中的共享锁的模式,

state代表着读锁，也就是传播，类似于上面图中前3个Node节点都是读锁，所以线程都能解锁；

c.CountDownLatch来讲，就比较复杂一些，它首先是共享锁，其次相比读锁，可以更细粒度的控制可以出队的Node的个数，

也就是state代表count，例如设置count为2，即使前面3个Node节点都能出队，到2个Node也就卡住了；

state状态的修改，通常都是AQS中的tryAquire子类的一些重载方法，基于子类的具体的业务进行state的定制，而这个也是整个AQS的核心；

6.Lock vs Sychronized优缺点对比

AQS通过构造一个基于阻塞的CLH队列容纳所有的阻塞线程，而对该队列的操作均通过Lock-Free（CAS）操作，
但对已经获得锁的线程而言，ReentrantLock实现了偏向锁的功能。

而synchronized关键字的底层实现也是一个基于CAS操作的等待队列，但JVM实现的更精细，把等待队列分为ContentionList和EntryList，目的是为了降低线程的出列速度；
当然synchronized关键字也实现了偏向锁，从数据结构来说二者设计没有本质区别。
但synchronized还实现了自旋锁，并针对不同的系统和硬件体系进行了操作系统级别的优化，性能更为优秀，而Lock则完全依靠系统阻塞挂起等待线程，这个是AQS的一大弱项。

当然Lock比synchronized更适合在应用层扩展，可以继承AbstractQueuedSynchronizer定义各种实现，比如实现读写锁（ReadWriteLock），公平或不公平锁；同时，Lock对应的Condition也比wait/notify要方便的多、灵活的多。

总结：

AQS队列是并发工具包的底层实现，它采用队列的形势，Node节点关联Thread，

模拟独占锁和共享锁，通过高效的CAS指令，达到同步控制的作用，堪比JVM的性能，

AQS的线程同步使用了LockSupport，比较方便的操纵unpark/park Thread,

AQS队列涉及的类太多，不能忽略，必看！

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

Linux GCC 常用命令 (/html/281/201607/2666539297/1.html)

Android嵌入式底层课程 (/html/466/201605/2247483948/1.html)

开源免费的文件同步工具-FreeFileSync (/html/299/201507/208172574/1.html)

Java设计模式（九）桥接模式 (/html/166/201605/2648885555/1.html)

Linux内核测试套件LTP初探-服务器篇 (/html/340/201609/2650278437/1.html)