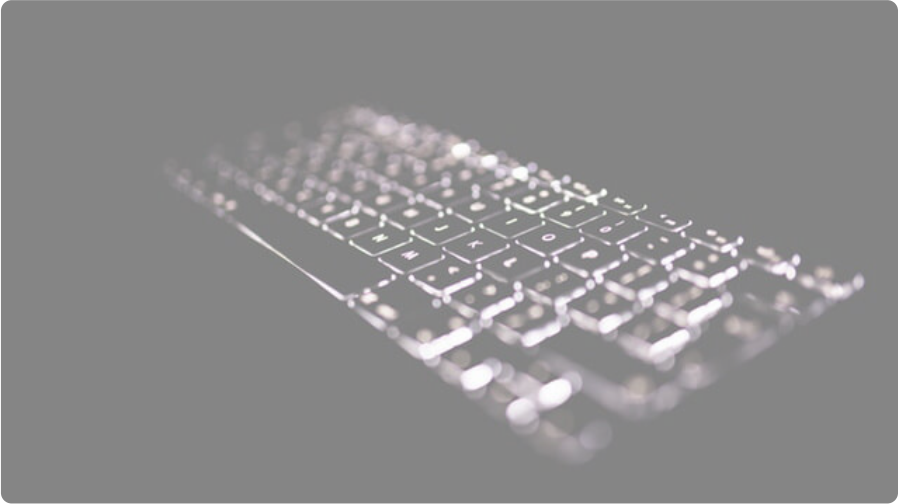


目录	
第1章 基础	
01 开篇词：为什么学习本专栏	
02 String、Long 源码解析和面试题	
03 Java 常用关键字理解	
04 Arrays、Collections、Objects 常用方法源码解析	
第2章 集合	
05 ArrayList 源码解析和设计思路	
06 LinkedList 源码解析	
07 List 源码会问哪些面试题	
08 HashMap 源码解析	
09 TreeMap 和 LinkedHashMap 核心源码解析	
10 Map源码会问哪些面试题	
11 HashSet、TreeSet 源码解析	
12 彰显细节：看集合源码对我们实际工作的帮助和应用	
13 差异对比：集合在 Java 7 和 8 有何不同和改进	
14 简化工作：Guava Lists Maps 实际工作运用和源码	
第3章 并发集合类	
15 CopyOnWriteArrayList 源码解析和设计思路	
16 ConcurrentHashMap 源码解析和设计思路	
17 并发 List、Map源码面试题	
18 场景集合：并发 List、Map的应用	

30 AbstractQueuedSynchronizer 源码解析（上）

更新时间：2019-11-07 10:11:55



“

不想当将军的士兵，不是好士兵。

——拿破仑

”

引导语

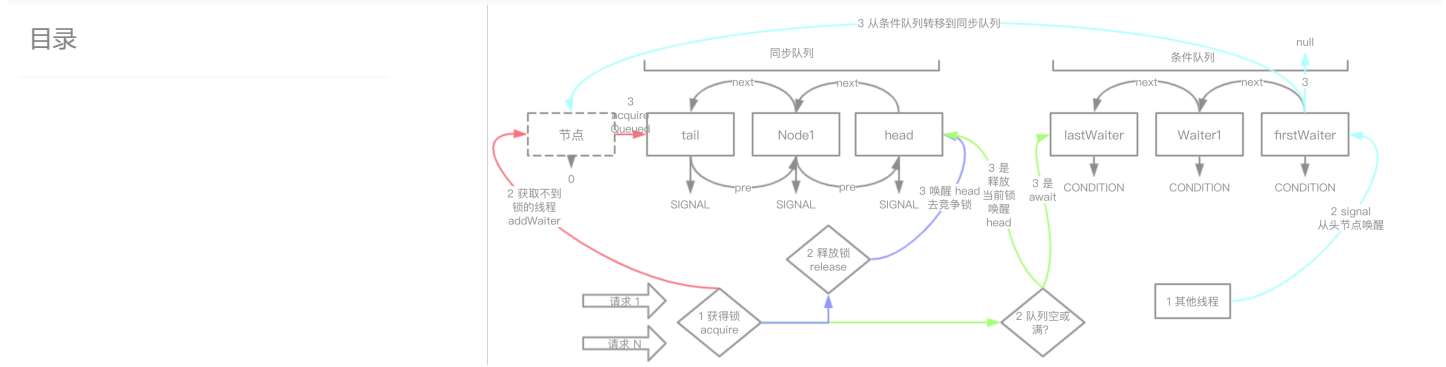
AbstractQueuedSynchronizer 中文翻译叫做同步器，简称 AQS，是各种各样锁的基础，比如说 ReentrantLock、CountDownLatch 等等，这些我们经常用的锁底层实现都是 AQS，所以学好 AQS 对于后面理解锁的实现是非常重要的。

锁章节的内容是这么安排的：

- 1：AQS 源码非常多，我们会分成两个小节来说，先把底层原理弄清楚；
- 2：我们平时用不到 AQS，只会接触到 ReentrantLock、CountDownLatch 这些锁，我们以两个锁为例子，讲解下源码，因为 AQS 只要弄懂了，所有的锁你只要清楚锁的目的，就能够利用 AQS 去实现它；
- 3：总结一下锁的面试题；
- 4：总结一下锁在工作中有哪些使用场景，举几个实际的例子，看看锁使用时，有哪些注意事项；
- 5：最后我们自己来实现一个锁，看看如果我们自己来实现锁，有哪些步骤，需要注意哪些事项。

ps：本章内容需要大量队列基础知识，没有看过第四章队列的同学，建议先阅读下队列章节。

1 整体架构



这个图总结了 AQS 整体架构的组成，和部分场景的动态流向，图中两个点说明一下，方便大家观看。

- 1. AQS 中队列只有两个：同步队列 + 条件队列，底层数据结构两者都是链表；
- 2. 图中有四种颜色的线代表四种不同的场景，1、2、3 序号代表看的顺序。

AQS 本身就是一套锁的框架，它定义了获得锁和释放锁的代码结构，所以如果要新建锁，只要继承 AQS，并实现相应方法即可。

接下来我们一起来看下这个图中各个细节点。

1.1 类注释

首先我们来看一下，从 AQS 类注释上，我们可以得到哪些信息：

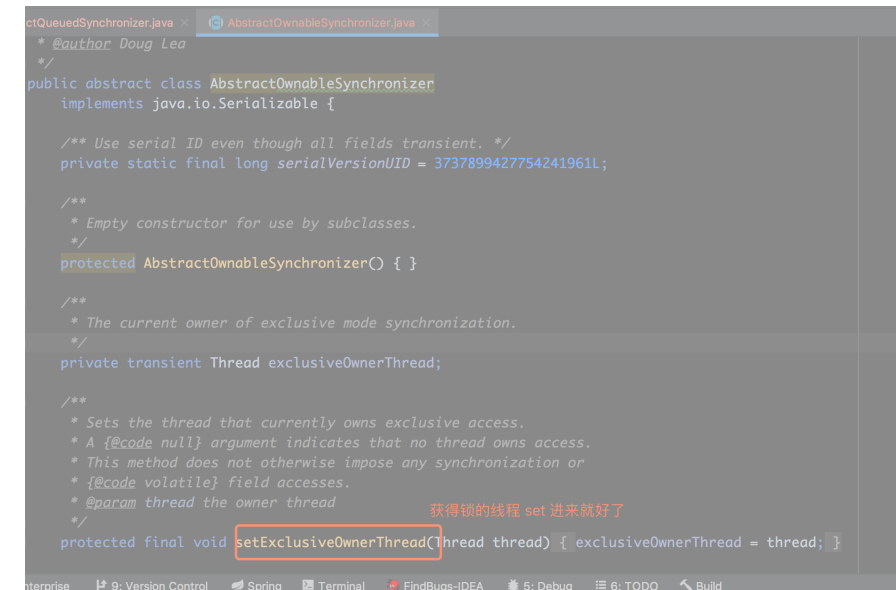
- 1. 提供了一种框架，自定义了先进先出的同步队列，让获取不到锁的线程能进入同步队列中排队；
- 2. 同步器有个状态字段，我们可以通过状态字段来判断能否得到锁，此时设计的关键在于依赖安全的 atomic value 来表示状态（虽然注释是这个意思，但实际上是通过把状态声明为 volatile，在锁里面修改状态值来保证线程安全的）；
- 3. 子类可以通过给状态 CAS 赋值来决定能否拿到锁，可以定义那些状态可以获得锁，哪些状态表示获取不到锁（比如定义状态值是 0 可以获得锁，状态值是 1 就获取不到锁）；
- 4. 子类可以新建非 public 的内部类，用内部类来继承 AQS，从而实现锁的功能；
- 5. AQS 提供了排它模式和共享模式两种锁模式。排它模式下：只有一个线程可以获得锁，共享模式可以让多个线程获得锁，子类 ReadWriteLock 实现了两种模式；
- 6. 内部类 ConditionObject 可以被用作 Condition，我们通过 new ConditionObject () 即可得到条件队列；
- 7. AQS 实现了锁、排队、锁队列等框架，至于如何获得锁、释放锁的代码并没有实现，比如 tryAcquire、tryRelease、tryAcquireShared、tryReleaseShared、isHeldExclusively 这些方法，AQS 中默认抛 UnsupportedOperationException 异常，都是需要子类去实现的；
- 8. AQS 继承 AbstractOwnableSynchronizer 是为了方便跟踪获得锁的线程，可以帮助监控和诊断工具识别是哪些线程持有了锁；
- 9. AQS 同步队列和条件队列，获取不到锁的节点在入队时是先进先出，但被唤醒时，可能并不会按照先进先出的顺序执行。

AQS 的注释还有很多很多，以上 9 点是挑选出来稍微比较重要的注释总结。

目录	AQS 类定义代码如下：
----	--------------

```
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {
```

- 可以看出两点：
1. AQS 是个抽象类，就是给各种锁子类继承用的，AQS 定义了很多如何获得锁，如何释放锁的抽象方法，目的就是为了让子类去实现；
 2. 继承了 AbstractOwnableSynchronizer，AbstractOwnableSynchronizer 的作用就是为了知道当前是那个线程获得了锁，方便监控用的，代码如下：



```
ctQueuedSynchronizer.java x AbstractOwnableSynchronizer.java x
* @author Doug Lea
*/
public abstract class AbstractOwnableSynchronizer
    implements java.io.Serializable {

    /** Use serial ID even though all fields transient. */
    private static final long serialVersionUID = 3737899427754241961L;

    /**
     * Empty constructor for use by subclasses.
     */
    protected AbstractOwnableSynchronizer() { }

    /**
     * The current owner of exclusive mode synchronization.
     */
    private transient Thread exclusiveOwnerThread;

    /**
     * Sets the thread that currently owns exclusive access.
     * A {@code null} argument indicates that no thread owns access.
     * This method does not otherwise impose any synchronization or
     * {@code volatile} field accesses.
     * @param thread the owner thread
     */
    protected final void setExclusiveOwnerThread(Thread thread) { exclusiveOwnerThread = thread; }
```

1.3 基本属性

AQS 的属性可简单分为四类：同步器简单属性、同步队列属性、条件队列属性、公用 Node。

1.3.1 简单属性

首先我们来看一下简单属性有哪些：

```
// 同步器的状态，子类会根据状态字段进行判断是否可以获得锁
// 比如 CAS 成功给 state 赋值 1 算得到锁，赋值失败为得不到锁，CAS 成功给 state 赋值 0 算释放
// 可重入锁，每次获得锁 +1，每次释放锁 -1
private volatile int state;

// 自旋超时阈值，单位纳秒
// 当设置等待时间时才会用到这个属性
static final long spinForTimeoutThreshold = 1000L;
```

最重要的就是 state 属性，是 int 属性的，所有继承 AQS 的锁都是通过这个字段来判断能不能获得锁，能不能释放锁。

1.3.2 同步队列属性

释放锁时，就会从同步队列头开始释放一个排队的线程，让线程重新去竞争锁。

所以同步队列的主要作用阻塞获取不到锁的线程，并在适当时机释放这些线程。

同步队列底层数据结构是个双向链表，我们从源码中可以看到链表的头尾，如下：

```
// 同步队列的头。  
private transient volatile Node head;  
  
// 同步队列的尾  
private transient volatile Node tail;
```

源码中的 Node 是同步队列中的元素，但 Node 被同步队列和条件队列公用，所以我们在说完条件队列之后再说 Node。

1.3.3 条件队列的属性

首先我们介绍下条件队列：条件队列和同步队列的功能一样，管理获取不到锁的线程，底层数据结构也是链表队列，但条件队列不直接和锁打交道，但常常和锁配合使用，是一定的场景下，对锁功能的一种补充。

条件队列的属性如下：

```
// 条件队列，从属性上可以看出是链表结构  
public class ConditionObject implements Condition, java.io.Serializable {  
    private static final long serialVersionUID = 1173984872572414699L;  
    // 条件队列中第一个 node  
    private transient Node firstWaiter;  
    // 条件队列中最后一个 node  
    private transient Node lastWaiter;  
}
```

ConditionObject 我们就称为条件队列，我们需要使用时，直接 new ConditionObject () 即可。

ConditionObject 是实现 Condition 接口的，Condition 接口相当于 Object 的各种监控方法，比如 Object#wait ()、Object#notify、Object#notifyAll 这些方法，我们可以先这么理解，后面会细说。

1.3.4 Node

Node 非常重要，即是同步队列的节点，又是条件队列的节点，在入队的时候，我们用 Node 把线程包装一下，然后把 Node 放入两个队列中，我们看下 Node 的数据结构，如下：

```
static final class Node {  
    /**  
     * 同步队列单独的属性  
     */  
    //node 是共享模式  
    static final Node SHARED = new Node();  
  
    //node 是排它模式  
    static final Node EXCLUSIVE = null;
```

目录	<pre>volatile Node prev; // 当前节点的下一个节点 volatile Node next; /** * 两个队列共享的属性 */ // 表示当前节点的状态，通过节点的状态来控制节点的行为 // 普通同步节点，就是 0，条件节点是 CONDITION -2 volatile int waitStatus; // waitStatus 的状态有以下几种 // 被取消 static final int CANCELLED = 1; // SIGNAL 状态的意义：同步队列中的节点在自旋获取锁的时候，如果前一个节点的状态是 SIGNAL static final int SIGNAL = -1; // 表示当前 node 正在条件队列中，当有节点从同步队列转移到条件队列时，状态就会被更改成 C static final int CONDITION = -2; // 无条件传播,共享模式下，该状态的进程处于可运行状态 static final int PROPAGATE = -3; // 当前节点的线程 volatile Thread thread; // 在同步队列中，nextWaiter 并不真的是指向其下一个节点，我们用 next 表示同步队列的下一个 // 但在条件队列中，nextWaiter 就是表示下一个节点元素 Node nextWaiter; }</pre>
----	--

从 Node 的结构中，我们需要重点关注 waitStatus 字段，Node 的很多操作都是围绕着 waitStatus 字段进行的。

Node 的 pre、next 属性是同步队列中的链表前后指向字段，nextWaiter 是条件队列中下一个节点的指向字段，但在同步队列中，nextWaiter 只是一个标识符，表示当前节点是共享还是排它模式。

1.3.5 共享锁和排它锁的区别

排它锁的意思是同一时刻，只能有一个线程可以获得锁，也只能有一个线程可以释放锁。

共享锁可以允许多个线程获得同一个锁，并且可以设置获取锁的线程数量。

1.4 Condition

刚才我们看条件队列 ConditionObject 时，发现其是实现 Condition 接口的，现在我们一起来看下 Condition 接口，其类注释上是这么写的：

1. 当 lock 代替 synchronized 来加锁时，Condition 就可以用来代替 Object 中相应的监控方法了，比如 Object#wait ()、Object#notify、Object#notifyAll 这些方法；
2. 提供了一种线程协作方式：一个线程被暂停执行，直到被其它线程唤醒；
3. Condition 实例是绑定在锁上的，通过 Lock#newCondition 方法可以产生该实例；

目录

类注释上甚至还给我们举了一个例子：

假设我们有一个有界边界的队列，支持 put 和 take 方法，需要满足：

- 1: 如果试图往空队列上执行 take，线程将会阻塞，直到队列中有可用的元素为止；
- 2: 如果试图往满的队列上执行 put，线程将会阻塞，直到队列中有空闲的位置为止。

1、2 中线程阻塞都会到条件队列中去阻塞。

take 和 put 两种操作如果依靠一个条件队列，那么每次只能执行一种操作，所以我们可以新建两个条件队列，这样就可以分别执行操作了，看了这个需求，是不是觉得很像我们第三章学习的队列？实际上注释上给的 demo 就是我们学习过的队列，篇幅有限，感兴趣的可以看看 ConditionDemo 这个测试类。

除了类注释，Condition 还定义出一些方法，这些方法奠定了条件队列的基础，方法主要有：

```
void await() throws InterruptedException;
```

这个方法的主要作用是：使当前线程一直等待，直到被 signalled 或被打断。

当以下四种情况发生时，条件队列中的线程将被唤醒

1. 有线程使用了 signal 方法，正好唤醒了条件队列中的当前线程；
2. 有线程使用了 signalAll 方法；
3. 其它线程打断了当前线程，并且当前线程支持被打断；
4. 被虚假唤醒 (即使没有满足以上 3 个条件，wait 也是可能被偶尔唤醒，虚假唤醒定义可以参考：https://en.wikipedia.org/wiki/Spurious_wakeup)。

被唤醒时，有一点需要注意的是：线程从条件队列中苏醒时，必须重新获得锁，才能真正被唤醒，这个我们在说源码的时候，也会强调这个。

await 方法还有带等待超时时间的，如下：

```
// 返回的 long 值表示剩余的给定等待时间，如果返回的时间小于等于 0，说明等待时间过了
// 选择纳秒是为了避免计算剩余等待时间时的截断误差
long awaitNanos(long nanosTimeout) throws InterruptedException;

// 虽然入参可以是任意单位的时间，但底层仍然转化成纳秒
boolean await(long time, TimeUnit unit) throws InterruptedException;
```

除了等待方法，还是唤醒线程的两个方法，如下：

```
// 唤醒条件队列中的一个线程，在被唤醒前必须先获得锁
void signal();

// 唤醒条件队列中的所有线程
void signalAll();
```

至此，AQS 基本的属性就已经介绍完了，接着让我们来看一看 AQS 的重要方法。

2 同步器的状态

目录

1. state 是锁的状态，是 int 类型，子类继承 AQS 时，都是要根据 state 字段来判断有无得到锁，比如当前同步器状态是 0，表示可以获得锁，当前同步器状态是 1，表示锁已经被其他线程持有，当前线程无法获得锁；
2. waitStatus 是节点（Node）的状态，种类很多，一共有初始化 (0)、CANCELLED (1)、SIGNAL (-1)、CONDITION (-2)、PROPAGATE (-3)，各个状态的含义可以见上文。

这两个状态我们需要牢记，不要混淆了。

3 获取锁

获取锁最直观的感受就是使用 Lock.lock () 方法来获得锁，最终目的是想让线程获得对资源的访问权。

Lock 一般是 AQS 的子类，lock 方法根据情况一般会选择调用 AQS 的 acquire 或 tryAcquire 方法。

acquire 方法 AQS 已经实现了，tryAcquire 方法是等待子类去实现，acquire 方法制定了获取锁的框架，先尝试使用 tryAcquire 方法获取锁，获取不到时，再入同步队列中等待锁。tryAcquire 方法 AQS 中直接抛出一个异常，表明需要子类去实现，子类可以根据同步器的 state 状态来决定是否能够获得锁，接下来我们详细看下 acquire 的源码解析。

acquire 也分两种，一种是排它锁，一种是共享锁，我们——来看下：

3.1 acquire 排它锁

```
// 排它模式下，尝试获得锁
public final void acquire(int arg) {
    // tryAcquire 方法是需要实现类去实现的，实现思路一般都是 cas 给 state 赋值来决定是否能获得
    if (!tryAcquire(arg) &&
        // addWaiter 入参代表是排他模式
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

以上代码的主要步骤是（流程见整体架构图中红色场景）：

1. 尝试执行一次 tryAcquire，如果成功直接返回，失败走 2；
2. 线程尝试进入同步队列，首先调用 addWaiter 方法，把当前线程放到同步队列的队尾；
3. 接着调用 acquireQueued 方法，两个作用，1：阻塞当前节点，2：节点被唤醒时，使其能够获得锁；
4. 如果 2、3 失败了，打断线程。

3.1.1 addWaiter

代码很少，每个方法都是关键，接下来我们先来看下 addWaiter 的源码实现：

```
// 方法主要目的：node 追加到同步队列的队尾
// 入参 mode 表示 Node 的模式（排它模式还是共享模式）
// 出参是新增的 node
// 主要思路：
```

目录

```
// 初始化 Node
Node node = new Node(Thread.currentThread(), mode);
// 这里的逻辑和 enq 一致，enq 的逻辑仅仅多了队尾是空，初始化的逻辑
// 这个思路在 java 源码中很常见，先简单的尝试放一下，成功立马返回，如果不行，再 while 循环
// 很多时候，这种算法可以帮忙解决大部分的问题，大部分的入队可能一次都能成功，无需自旋
Node pred = tail;
if (pred != null) {
    node.prev = pred;
    if (compareAndSetTail(pred, node)) {
        pred.next = node;
        return node;
    }
}
//自旋保证node加入到队尾
enq(node);
return node;
}

// 线程加入同步队列中方法，追加到队尾
// 这里需要重点注意的是，返回值是添加 node 的前一个节点
private Node enq(final Node node) {
    for (;;) {
        // 得到队尾节点
        Node t = tail;
        // 如果队尾为空，说明当前同步队列都没有初始化，进行初始化
        // tail = head = new Node();
        if (t == null) {
            if (compareAndSetHead(new Node()))
                tail = head;
        }
        // 队尾不为空，将当前节点追加到队尾
        else {
            node.prev = t;
            // node 追加到队尾
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
}
```

如果之前学习过队列的同学，对这个方法应该感觉毫不吃力，就是把新的节点追加到同步队列的队尾。

其中有一点值得我们学习的地方，是在 addWaiter 方法中，并没有进入方法后马上就自旋，而是先尝试一次追加到队尾，如果失败才自旋，因为大部分操作可能一次就会成功，这种思路在我们写自旋的时候可以借鉴。

3.1.2 acquireQueued

下一步就是要阻塞当前线程了，是 acquireQueued 方法来实现的，我们来看下源码实现：

```
// 主要做两件事情：
// 1：通过不断的自旋尝试使自己前一个节点的状态变成 signal，然后阻塞自己。
// 2：获得锁的线程执行完成之后，释放锁时，会把阻塞的 node 唤醒,node 唤醒之后再次自旋，尝试
// 返回 false 表示获得锁成功，返回 true 表示失败
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
```


目录

```
for (;;) {
    // 选上一个节点
    final Node p = node.predecessor();
    // 有两种情况会走到 p == head:
    // 1:node 之前没有获得锁, 进入 acquireQueued 方法时, 才发现他的前置节点就是头节点,
    // 2:node 之前一直在阻塞沉睡, 然后被唤醒, 此时唤醒 node 的节点正是其前一个节点, 也
    // 如果自己 tryAcquire 成功, 就立马把自己设置成 head, 把上一个节点移除
    // 如果 tryAcquire 失败, 尝试进入同步队列
    if (p == head && tryAcquire(arg)) {
        // 获得锁, 设置成 head 节点
        setHead(node);
        //p被回收
        p.next = null; // help GC
        failed = false;
        return interrupted;
    }

    // shouldParkAfterFailedAcquire 把 node 的前一个节点状态置为 SIGNAL
    // 只要前一个节点状态是 SIGNAL了, 那么自己就可以阻塞(park)了
    // parkAndCheckInterrupt 阻塞当前线程
    if (shouldParkAfterFailedAcquire(p, node) &&
        // 线程是在这个方法里面阻塞的, 醒来的时候仍然在无限 for 循环里面, 就能再次自旋尝试
        parkAndCheckInterrupt())
        interrupted = true;
    }
} finally {
    // 如果获得node的锁失败, 将 node 从队列中移除
    if (failed)
        cancelAcquire(node);
}
}
```

此方法的注释还是很清楚的, 我们接着看下此方法的核心: `shouldParkAfterFailedAcquire`, 这个方法的主要目的就是把前一个节点的状态置为 `SIGNAL`, 只要前一个节点的状态是 `SIGNAL`, 当前节点就可以阻塞了 (`parkAndCheckInterrupt` 就是使节点阻塞的方法), 源码如下:

```
// 当前线程可以安心阻塞的标准, 就是前一个节点线程状态是 SIGNAL 了。
// 入参 pred 是前一个节点, node 是当前节点。

// 关键操作:
// 1: 确认前一个节点是否有效, 无效的话, 一直往前找到状态不是取消的节点。
// 2: 把前一个节点状态置为 SIGNAL。
// 1、2 两步操作, 有可能一次就成功, 有可能需要外部循环多次才能成功 (外面是个无限的 for 循环)
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    // 如果前一个节点 waitStatus 状态已经是 SIGNAL 了, 直接返回, 不需要在自旋了
    if (ws == Node.SIGNAL)
        /*
         * This node has already set status asking a release
         * to signal it, so it can safely park.
         */
        return true;
    // 如果当前节点状态已经被取消了。
    if (ws > 0) {
        /*
         * Predecessor was cancelled. Skip over predecessors and
         * indicate retry.
         */
    }
}
```

目录

```
node.prev = pred = pred.prev;
} while (pred.waitStatus > 0);
pred.next = node;
// 否则直接把节点状态置 为SIGNAL
} else {
    /*
     * waitStatus must be 0 or PROPAGATE. Indicate that we
     * need a signal, but don't park yet. Caller will need to
     * retry to make sure it cannot acquire before parking.
     */
    compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
}
return false;
}
```

acquire 整个过程非常长，代码也非常多，但注释很清楚，可以一行一行仔细看看代码。

总结一下，acquire 方法大致分为三步：

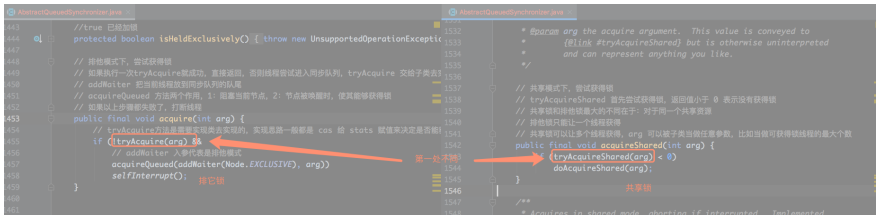
1. 使用 tryAcquire 方法尝试获得锁，获得锁直接返回，获取不到锁的走 2；
2. 把当前线程组装成节点（Node），追加到同步队列的尾部（addWaiter）；
3. 自旋，使同步队列中当前节点的前置节点状态为 signal 后，然后阻塞自己。

整体的代码结构比较清晰，一些需要注意的点，都用注释表明了，强烈建议阅读下源码。

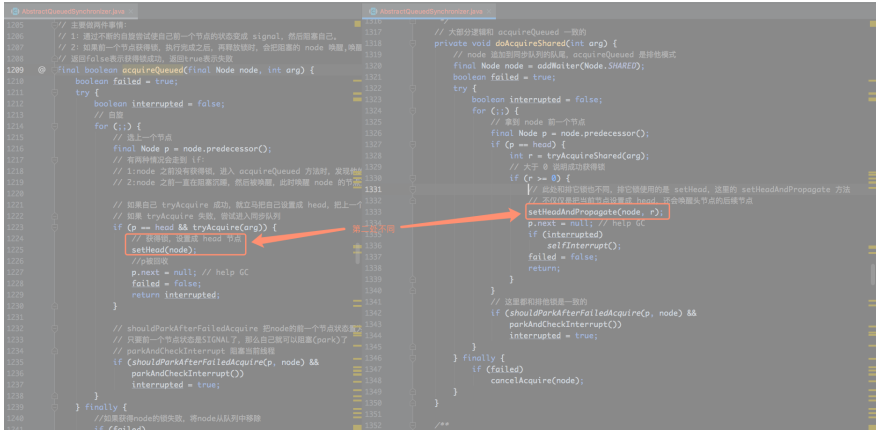
3.2 acquireShared 获取共享锁

acquireShared 整体流程和 acquire 相同，代码也很相似，重复的源码就不贴了，我们就贴出来不一样的代码来，也方便大家进行比较：

1. 第一步尝试获得锁的地方，有所不同，排它锁使用的是 tryAcquire 方法，共享锁使用的是 tryAcquireShared 方法，如下图：



2. 第二步不同，在于节点获得排它锁时，仅仅把自己设置为同步队列的头节点即可（setHead 方法），但如果是共享锁的话，还会去唤醒自己的后续节点，一起来获得该锁（setHeadAndPropagate 方法），不同之处如下（左边排它锁，右边共享锁）：



目录

```
// 主要做两件事情
// 1:把当前节点设置成头节点
// 2:看看后续节点有无正在等待，并且也是共享模式的，有的话唤醒这些节点
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    // 当前节点设置成头节点
    setHead(node);
    /*
     * Try to signal next queued node if:
     *   Propagation was indicated(表示指示) by caller,
     *   or was recorded (as h.waitStatus either before
     *   or after setHead) by a previous operation
     *   (note: this uses sign-check of waitStatus because
     *   PROPAGATE status may transition to SIGNAL.)
     * and
     *   The next node is waiting in shared mode,
     *   or we don't know, because it appears null
     *
     * The conservatism(保守) in both of these checks may cause
     * unnecessary wake-ups, but only when there are multiple
     * racing acquires/releases, so most need signals now or soon
     * anyway.
     */
    // propagate > 0 表示已经有节点获得共享锁了
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        //共享模式，还唤醒头节点的后置节点
        if (s == null || s.isShared())
            doReleaseShared();
    }
}

// 释放后置共享节点
private void doReleaseShared() {
    /*
     * Ensure that a release propagates, even if there are other
     * in-progress acquires/releases. This proceeds in the usual
     * way of trying to unparkSuccessor of head if it needs
     * signal. But if it does not, status is set to PROPAGATE to
     * ensure that upon release, propagation continues.
     * Additionally, we must loop in case a new node is added
     * while we are doing this. Also, unlike other uses of
     * unparkSuccessor, we need to know if CAS to reset status
     * fails, if so rechecking.
     */
    for (;;) {
        Node h = head;
        // 还没有到队尾，此时队列中至少有两个节点
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 如果队列状态是 SIGNAL，说明后续节点都需要唤醒
            if (ws == Node.SIGNAL) {
                // CAS 保证只有一个节点可以运行唤醒的操作
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue; // loop to recheck cases
                // 进行唤醒操作
                unparkSuccessor(h);
            }
        }
        else if (ws == 0 &&
            !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
            continue; // loop on failed CAS
    }
}
```

目录	<pre>// 加上共享锁的特性就是可以多个线程获得锁，也可以释放锁，这就导致头节点可能会发生变化 // 如果头节点发生了变化，就继续循环，一直循环到头节点不变化时，结束循环。 if (h == head) // loop if head changed break; } }</pre>
----	--

这个就是共享锁独特的地方，当一个线程获得锁后，它就会去唤醒排在它后面的其它节点，让其它节点也能够获得锁。

4 总结

AQS 的内容实在太多了，这只是 AQS 的上篇，但内容长度已经超过了我们平时章节的三倍了，所以不得不分节，下一章仍然是 AQS，主要讲解锁的释放和条件队列两大部分。

精选留言 7

欢迎在这里发表留言，作者筛选后可公开显示

慕粉1150563265

队列满或者空，直接加入同步队列，看上去也没啥问题呀

👍 0 回复

2019-12-02

文贺 回复 慕粉1150563265

同学你好，你问的这三个问题，我自己理解应该还是没有弄清楚已经有了同步队列，为什么还需要条件队列，你可以看看下一篇解析，这个是需要锁+队列的场景下，才需要同步队列和条件队列的相互配合。

回复

7天前

慕粉1150563265 回复 慕粉1150563265

是的，后来想了下，如果加入到了同步队列中，后面其他线程就会永远阻塞

回复

7天前

慕粉1150563265

队列满或者空的时候，为什么不直接加入到同步队列中，反而加入了条件队列，这点没说，求指教

👍 0 回复

2019-12-02

慕粉1150563265

其实，有一点我还是没理解，为什么加入条件队列之后，又要移动到同步队列中呢？

目录	<div><div>qq_铂协成员_0</div><div>线程在条件队列和同步队列之间的转移没理清，能解释一下吗？</div><div><div>👍 0</div><div>回复</div><div>2019-11-27</div></div><div><div>文贺 回复 qq_铂协成员_0</div><div>可以结合 AbstractQueuedSynchronizer 源码解析（下）中的 singal、singalAll 方法看看。</div><div><div>回复</div><div>2019-11-30 13:18:49</div></div></div></div>
	<div><div>慕盖茨4571687</div><div>老师我问一下，为啥是当前节点的前驱是头节点能获取锁，不应该是当前节点是头节点才能获取锁吗？</div><div><div>👍 1</div><div>回复</div><div>2019-11-21</div></div><div><div>文贺 回复 慕盖茨4571687</div><div>因为头节点肯定是已经，或者曾经获得锁了，现在由头节点去唤醒后续的锁。</div><div><div>回复</div><div>2019-11-23 16:40:50</div></div></div></div>
	<div><div>daygoodgoodstudy</div><div>原来看网上的文章都是迷迷糊糊的，这位大佬一在分析之余还各个点小结一番，看得很舒服</div><div><div>👍 0</div><div>回复</div><div>2019-11-21</div></div><div><div>文贺 回复 daygoodgoodstudy</div><div>谢谢同学，一起进步学习。</div><div><div>回复</div><div>2019-11-23 16:39:43</div></div></div></div>
	<div><div>慕盖茨4571687</div><div>老师我想问一下同步队列和条件队列啥区别没太理解</div><div><div>👍 0</div><div>回复</div><div>2019-11-15</div></div><div><div>文贺 回复 慕盖茨4571687</div><div>两者底层原理相近，条件队列可以看做是对同步队列的功能补充。</div><div><div>回复</div><div>2019-11-17 10:40:54</div></div></div></div>

千学不如一看，千看不如一练