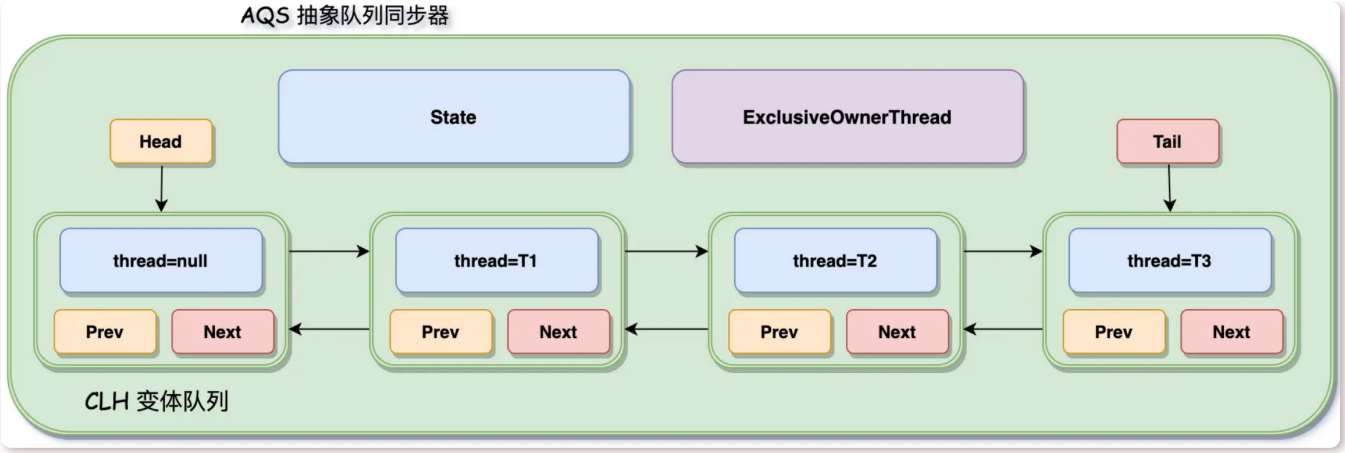
前言

在本文中并不会去很深入的去全面的了解 *AQS* 和 *ReentrantLock* 的源码，旨在能够简单直接的去理解 *AQS* 的思想和 *ReentrantLock* 中这些思想的具体体现形式，并且主要以 *ReentrantLock* 中默认的非公平锁为例子进行介绍，公平锁的差距会略微提及，详细的可查看参考的资料进行查看，相信各位小伙伴看完这一块之后多少会有一点收获和帮助。

> 🌈
>
> **参考资料：**
>
> 1. 万字图文 | 聊一聊 ReentrantLock 和 AQS 那点事
> 2. 深入剖析ReentrantLock公平锁与非公平锁源码实现
> 3. 线程中断：Thread类中interrupt（）、interrupted（）和 isInterrupted（）方法详解

1、什么是AQS

在 **Java** 中，*AQS* 是 *AbstractQueuedSynchronizer* 的简称，直译过来是抽象队列同步器。*AbstractQueuedSynchronizer* 是一个提供了基于 *FIFO* 等待队列实现的**同步器框架**，是 **Java** 并发库中锁和同步器的核心实现之一。它允许开发人员通过**继承** *AQS* 类来实现自定义同步器，从而为多线程程序提供可靠的同步机制。



`AQS` 的**核心思想**是，将等待共享资源的线程封装在一个 `FIFO` 队列中，然后用 `CAS` 操作等原子操作来修改该队列中的头结点和尾结点。对于独占式同步器（例如 `ReentrantLock`），`AQS` 还提供了一个 `state` 变量，用于记录当前占用该同步器的线程数。每次执行 `acquire` 操作时，线程会尝试获取同步器的状态。如果成功获取，则该线程可以继续执行；否则，需要一定的阻塞等待唤醒机制来保证锁的分配，`AQS` 中会将竞争共享资源失败的线程添加到一个变体的 `CLH` 队列中。

```
1  public abstract class AbstractQueuedSynchronizer
2    extends AbstractOwnableSynchronizer implements
   java.io.Serializable {
3      // CLH 变体队列头、尾节点
4      private transient volatile Node head;
5      private transient volatile Node tail;
6      // AQS 同步状态
7      private volatile int state;
8      // CAS 方式更新 state
9      protected final boolean compareAndSetState(int expect, int
   update) {
10         return unsafe.compareAndSwapInt(this, stateOffset, expect,
   update);
11     }
12 }
```

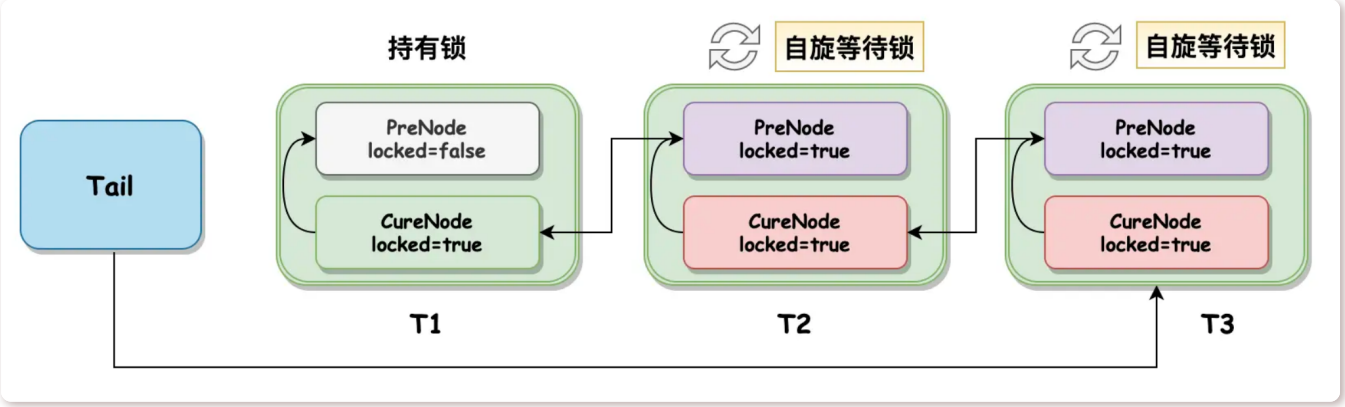
> 🌈
>
> 简单来说，`AQS` 是Java中的一个抽象类，为开发者提供了一种非常灵活的**同步机制**，可以适用于多种场景，相比较于传统的 `synchronized` 关键字更加高效和可定制化。

2、谈谈CLH队列

`CLH(Craig、Landin and Hagersten)` 队列，是 **单向链表实现的队列**。申请线程只在本地变量上自旋，**它不断轮询前驱的状态**，如果发现 **前驱节点释放了锁就结束自旋**，其主要有以下特点：

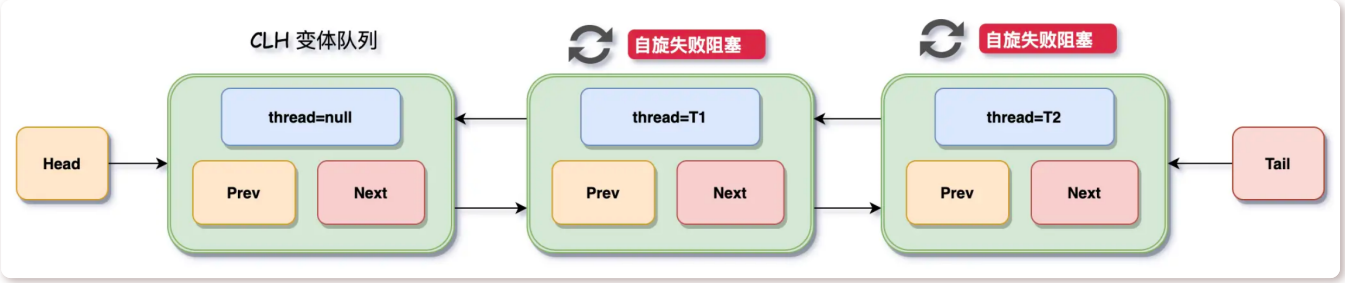
1. `CLH` 队列是一个**单向链表**，保持 `FIFO` 先进先出的队列特性；

2. 通过 `tail` 尾节点（原子引用）来构建队列，总是指向**最后一个节点**；
3. 未获得锁节点会进行**自旋**，而不是切换线程状态；
4. **并发高时性能较差**，因为未获得锁节点不断轮训前驱节点的状态来查看是否获得锁。



`AQS` 中的队列是 `CLH` 变体的**虚拟双向队列**，通过将每条请求共享资源的线程封装成一个节点来实现锁的分配，相对于普通的 `CLH` 队列来说，其主要有以下特点：

1. `AQS` 中队列是个**双向链表**，也是 `FIFO` 先进先出的特性；
2. 通过 `Head` 、 `Tail` 头尾两个节点来组成队列结构，通过 `volatile` 修饰保证可见性；
3. `Head` 指向节点为已获得锁的节点，是一个**虚拟节点**，节点本身不持有具体线程；
4. 获取不到同步状态，会将节点进行**自旋**获取锁，自旋一定次数失败后会将线程**阻塞**，相对于 `CLH` 队列性能较好。



3、独占锁与共享锁

独占锁也叫排它锁，是指该锁一次只能被一个线程所持有，如果别的线程想要获取锁，只有等到持有锁线程释放。获得排它锁的线程即能读数据又能修改数据，与之对立的就是共享锁。

共享锁是指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。

	独占锁	共享锁
独占锁	不可共存	不可共存
共享锁	不可共存	可共存

4、什么是ReentrantLock

`ReentrantLock` 翻译为 **可重入锁**，指的是一个线程能够对 **临界区共享资源进行重复加锁**，确保线程安全最常见的做法是利用锁机制如 `Lock、sychronized` 来对 **共享数据做互斥同步**，这样在同一个时刻，只有 **一个线程可以执行某个方法或者某个代码块**，那么操作必然是 **原子性的，线程安全的**，与 `sychronized` 主要有以下区别：

	Synchronized	ReentrantLock
锁实现机制	对象头监视器模式	依赖 AQS
灵活性	不灵活	支持响应中断、超时、尝试获取锁
释放锁形式	自动释放锁	显示调用 unlock()
支持锁类型	非公平锁	公平锁 & 非公平锁
条件队列	单条件队列	多个条件队列
是否支持可重入	支持	支持

5、认识AOS

抽象类 `AQS` 同样继承自抽象类 `AOS（AbstractOwnableSynchronizer）`，其内部只有一个 Thread 类型的变量，提供了获取和设置当前独占锁线程的方法，主要作用是 **记录当前占用独占锁（互斥锁）的线程实例**。

```
1  public abstract class AbstractOwnableSynchronizer implements
   java.io.Serializable {
2      // 独占线程（不参与序列化）
3      private transient Thread exclusiveOwnerThread;
4      // 设置当前独占的线程
5      protected final void setExclusiveOwnerThread(Thread thread) {
6          exclusiveOwnerThread = thread;
7      }
8      // 返回当前独占的线程
9      protected final Thread getExclusiveOwnerThread() {
10         return exclusiveOwnerThread;
11     }
12 }
```

6、公平锁与非公平锁

6.1、公平锁FairSync

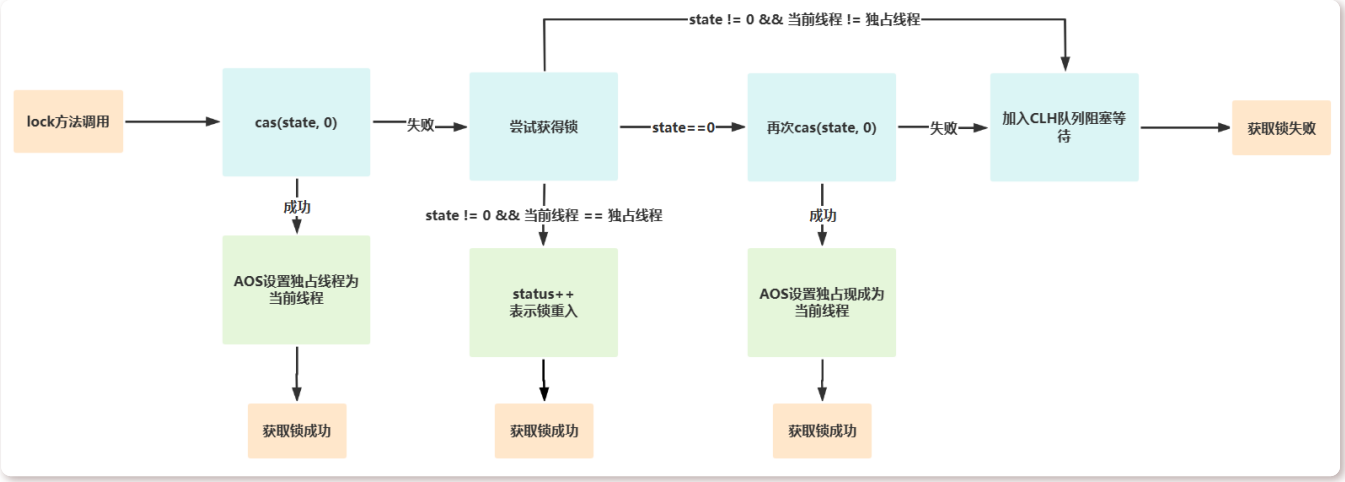
1. 公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁
2. 公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大

6.2、非公平锁NonfairSync

1. 非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁
2. 非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU 不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁

7、理解加锁过程

到这里的之前六点都是为了这里的加锁和后续的解锁过程进行一个前置准备，从这里开始将重点介绍独占锁，也就是 `ReentrantLock` 的非公平锁的加锁和解锁过程。开始之前先放一张我个人理解的加锁流程图：



下面将从创建 `ReentrantLock` 到底层 `AQS` 实现的顺序，和大家一起一步步的查看其中的源码：

1. 创建锁并调用上锁方法，需要注意的是， `ReentrantLock` 具有公平锁和非公平锁的实现，默认是非公平锁，如果后续需要尝试非公平锁的话可以通过构造器 `public ReentrantLock(boolean fair)` 进行创建：

```
public static void main(String[] args) {
    // 创建非公平锁
    ReentrantLock lock = new ReentrantLock();
    // 获取锁操作
    lock.lock();
    try {
        // 执行代码逻辑
```

```
    } catch (Exception ex) {
        // 异常处理逻辑
    } finally {
        // 解锁操作
        lock.unlock();
    }
}
```

2. 通过点击 `lock()` 方法进去可以发现是调用了内部类实现的同步器的上锁方法 `lock()`，我们继续点进去选择非公平锁实现就能找到对应的上锁逻辑：

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

--- 选择 NonfairSync --

final void lock() {
    // 使用CAS尝试获得锁，非公平的体现
    if (compareAndSetState(0, 1))
        // 为前面提及的AOS方法，获取锁成功便设置独占线程为当前线程
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // AQS思想的体现
        acquire(1);
}
```

3. **if** 块中应该还算是很简单的逻辑的，由于是**非公平锁**，所以能够直接尝试去获得锁而不会直接被安排去阻塞入队，如果对 `CAS` 不了解并且感兴趣的小伙伴可以前往之前的文章[【并发编程】CAS到底是什么]。

   接下来我们主要看 **else** 块中的 `acquire()` 方法，其对整个 `AQS` 做到了**承上启下**的作用，通过 `tryAcquire()` 模版方法进行尝试获取锁，获取锁失败包装当前线程为 `Node` 节点加入等待队列排队：

```
// 为了方便查看我给 if 块加了大括号，并调整了if中的换行，源码中是没有的(可能是JDK开发者的风格如此，我看着挺难受的)
public final void acquire(int arg) {
    // 再次尝试获得锁，如果失败了取反之后为真，便会执行后面的 acquireQueued 方法将当前线程包装入队
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) {
        // if块逻辑，当再次获取锁失败和包装入队成功后，将当前线程标记为中断状态
        selfInterrupt();
    }
}

```

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

4. 接下来我们按照顺序查看 `tryAcquire()` 方法，看一下非公平锁的这个方法是怎么实现的，点进去之后会发现是 `AQS` 中的方法，默认实现是抛出一个异常的，我们选择非公平锁实现即可：

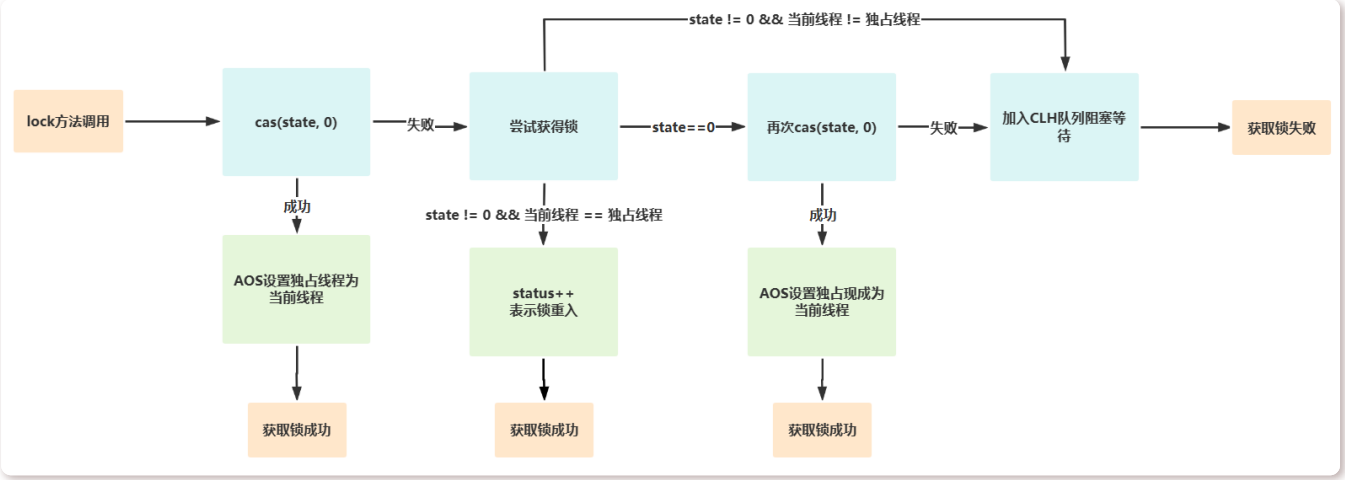
```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}

// 选中 NonfairSync 实现
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

// 继续查看 nonfairTryAcquire 实现
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // state == 0证明无锁状态，可直接争夺锁，体现了非公平特性
    if (c == 0) {
        // CAS 尝试获得锁
        if (compareAndSetState(0, acquires)) {
            // 争夺锁成功调用AOS中方法设置独占线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // state ! = 0 证明有锁占据，需要判断独占线程是否是当前线程，体现了锁可重入特性
    else if (current == getExclusiveOwnerThread()) {
        // 增加重入次数
        int nextc = c + acquires;
        // 同步状态值达到整形最大值，再增加则会整形溢出，由最大值转变成负数，如果继续执行将永远无法获取到锁，造成死锁问题，因此在这里抛出异常
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 同步状态值正常，更新该值，获取锁成功
        setState(nextc);
        return true;
    }
    // 争夺锁失败
    return false;
}
```

5. 按照顺序接下来应该是方法 `acquireQueued()` 了，这个方法在这里就不会展开进行讲解(主要是我也没完全理清逻辑，打一个 `TODO` 吧)，感兴趣的小伙伴可以前往参考资料里面第一篇文章进行查看，里面讲得十分清楚，如果只是简单理解的话，那就是这个方法中，会根据 `CLH` 队列 `FIFO` 特性将当前线程封装成 `Node` 数据结构从队列尾部插入到队列中。

```
// 为了方便查看我给 if 块加了大括号，并调整了if中的换行，源码中是没有的(可能是JDK开发者的风格如此，我看着挺难受的)
public final void acquire(int arg) {
    // 再次尝试获得锁，如果失败了取反之后为真，便会执行后面的 acquireQueued 方法将当前线程包装入队
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) {
        // if块逻辑，当再次获取锁失败和包装入队成功后，将当前线程标记为中断状态
        selfInterrupt();
    }
}
```

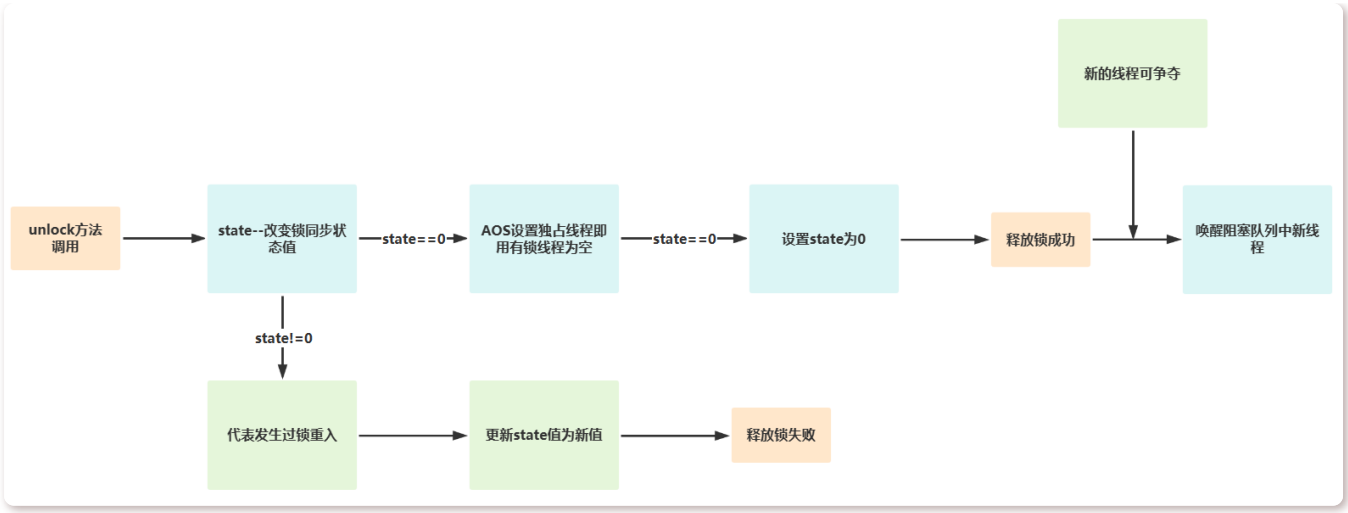
6. 最后的 `selfInterrupt()` 方法则是通过 `interrupt()` 方法对线程进行标记，具体方法理解网上资料也很多，可以查看参考资料中的相关文章进行查看。

> 到这里加锁的大概流程就此结束啦，再次将开始放出来自己绘画的流程图贴在这里，就当做是总结吧。



8、理解解锁过程

解锁过程相对于加锁过程来说简单了许多，因为步骤相对较少，下面是我个人理解画的流程图：



下面将开始通过源码步骤一点点来理解整个流程：

1. 通过调用 `unlock()` 方法执行释放锁流程，这里一般处于final块中，进去后同样是调用内部类实例的同步器实现的 `release()` 方法，我们继续点进去：

```
public void unlock() {
    sync.release(1);
}

public final boolean release(int arg) {
    // 尝试释放锁
    if (tryRelease(arg)) {
        // 获取头结点
        Node h = head;
        // 唤醒头结点
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

2. 从上面的源码我们可以看出，释放锁的核心应该是在 `tryRelease()` 方法中，我们点进去会发现这是 `AQS` 中的方法，默认实现为抛异常，因此我们直接前往其在 `ReentrantLock` 中的具体实现：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    // 如果当前线程不等于拥有锁线程，抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 如果减去 releases 后为0，证明锁释放成功，否则说明发生了锁重入
    if (c == 0) {
        free = true;
        // 将拥有锁线程设置为空
```

```
            setExclusiveOwnerThread(null);
        }
        // 更新state状态为0代表无锁，或重入次数减一
        setState(c);
        // 返回解锁情况
        return free;
    }
```

3. 在锁释放成功之后，我们回到 `release()` 方法中，会发现获取了阻塞队列中的头结点并尝试将其唤醒，当然这一块同样不深入探索，简单理解为，阻塞队列为遵循 `FIFO` 规则的 `CLH` 变体队列，一般情况下唤醒的是头结点：

```
public final boolean release(int arg) {
    // 尝试释放锁
    if (tryRelease(arg)) {
        // 获取头结点
        Node h = head;
        // 唤醒头结点
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

9、解锁之后唤醒线程补充

9.1、基本流程

唤醒线程的基本流程如下：

1. 当线程释放锁时，它会检查等待队列中是否有等待线程；
2. 如果等待队列中没有等待线程，则本次释放锁的操作完毕，其他线程可以继续尝试获取锁；
3. 如果等待队列中存在等待线程，则从**队列的头部**选择一个等待线程，并将其从等待队列中删除（此时，如果等待队列只有这一个等待线程，则等待队列为空）；
4. 选中的等待线程被唤醒，并尝试获取锁。如果获取成功，则该线程成为新的持有锁的线程；如果获取失败，则该线程会再次加入等待队列中等待下一次唤醒。

9.2、头结点一定存在吗

从前面的源码中我们可以看到其实有一步是判断头结点是否为空的 `h != null && h.waitStatus != 0`，那么**什么情况下头节点为空**呢，当线程还在争夺锁，队列还未初始化，头节点必然是为空的，当头节点等待状态等于0，证明后继节点还在自旋，不需要进行后继节点唤醒。

9.3、唤醒的一定是头结点吗

在 `ReentrantLock` 的非公平锁实现中，当唤醒阻塞队列中的节点时，会**优先选择队首节点进行唤醒**。如果队首节点的**等待状态为0**（即已经被唤醒），则**继续向后查找**，直到找到一个等待状态**不为0**的节点进行唤醒。因此，在 `ReentrantLock` 的非公平锁实现中，唤醒的节点可能不是头结点，而是任意一个等待状态不为0的节点。这种行为可以减少线程唤醒的数量，从而提高性能。

9.4、唤醒的线程一定会持有锁吗

其实不然，在上面的流程图中也有提及，在锁释放之后，state的值其实已经变成了0，此时**其他线程是可以进来争夺锁的**，可别忘了我们说的是非公平锁。

- 如果在唤醒等待队列中的线程的过程中，**没有其他线程进来争夺并持有锁**，那么成功唤醒的线程就可以成功占有锁的坑位；
- 但是！如果这个时候**有其它线程进行争夺锁**，那么唤醒的线程只能够和加锁其中的逻辑相同，和其它线程各凭本事争夺锁资源了，如果获取锁失败后再次尝试过后还是获取不到就只能够回到阻塞队列里面呆着了。

> 🌈
>
> 扯到这里已经是很长了，在做笔记的同时自己对 `AQS` 的理解也更进了一步，希望对阅读本文的小伙伴也有帮助和得到对应的收获，再次感谢和贴上参考资料：
>
> 1. 万字图文 | 聊一聊 ReentrantLock 和 AQS 那点事
> 2. 深入剖析ReentrantLock公平锁与非公平锁源码实现
> 3. 线程中断：Thread类中interrupt()、interrupted()和isInterrupted()方法详解