简介

Java中的大部分同步类（Lock、Semaphore、ReentrantLock等）都是基于AbstractQueuedSynchronizer（简称为AQS）实现的。AQS是一种提供了原子式管理同步状态、阻塞和唤醒线程功能以及队列模型的简单框架。

在AQS中的锁类型有两种：分别是**「Exclusive(独占锁)*「和」*Share(共享锁)」**。

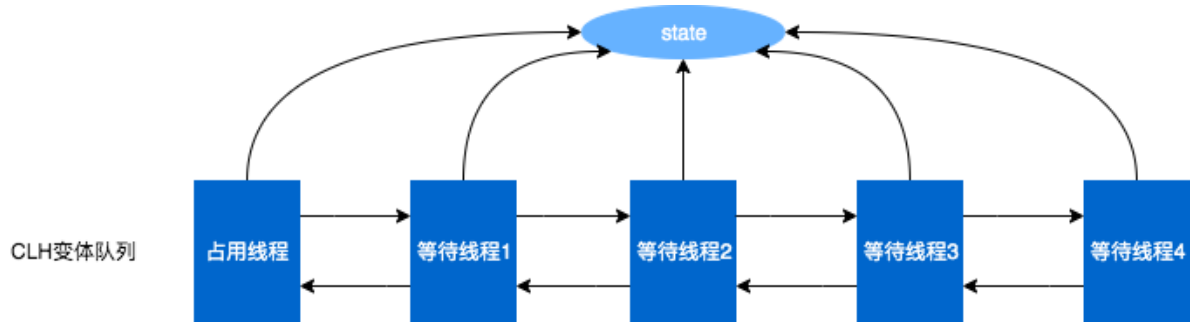
「独占锁」就是**「每次都只有一个线程运行」**，例如`ReentrantLock`。

「共享锁」就是**「同时可以多个线程运行」**，如`Semaphore、CountDownLatch、ReentrantReadWriteLock`。

原理

CLH：Craig、Landin and Hagersten队列，是单向链表，AQS中的队列是CLH变体的虚拟双向队列（FIFO），AQS是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。

主要原理图如下：



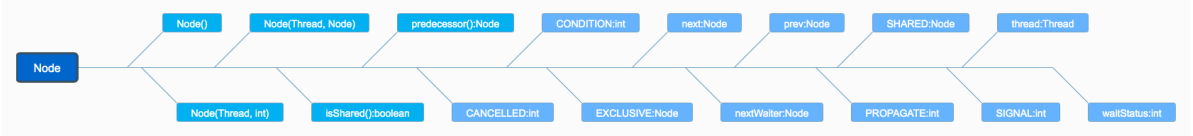
AQS使用一个Volatile的int类型的成员变量来表示同步状态，通过内置的FIFO队列来完成资源获取的排队工作，通过CAS完成对State值的修改。

在FIFO队列中，**「头节点占有锁」**，也就是头节点才是锁的持有者，尾指针指向队列的最后一个等待线程节点，除了头节点和尾节点，节点之间都有**「前驱指针」**和**「后继指针」**

在AQS中维护了一个**「共享变量state」**，标识当前的资源是否被线程持有，多线程竞争的时候，会去判断state是否为0，尝试的去把state修改为1

1. AQS数据结构

AQS中最基本的数据结构——Node，Node即为上面CLH变体队列中的节点。



解释一下几个方法和属性值的含义：

方法和属性值	含义
waitStatus	当前节点在队列中的状态
thread	表示处于该节点的线程
prev	前驱指针
predecessor	返回前驱节点，没有的话抛出npe
nextWaiter	指向下一个处于CONDITION状态的节点（由于本篇文章不讲述Condition Queue队列，这个指针不多介绍）
next	后继指针

线程两种锁的模式：

模式	含义
SHARED	表示线程以共享的模式等待锁
EXCLUSIVE	表示线程正在以独占的方式等待锁

waitStatus有下面几个枚举值：

枚举	含义
0	当一个Node被初始化的时候的默认值
CANCELLED	为1，表示线程获取锁的请求已经取消了
CONDITION	为-2，表示节点在等待队列中，节点线程等待唤醒
PROPAGATE	为-3，当前线程处在SHARED情况下，该字段才会使用
SIGNAL	为-1，表示线程已经准备好了，就等资源释放了

2. 同步状态State

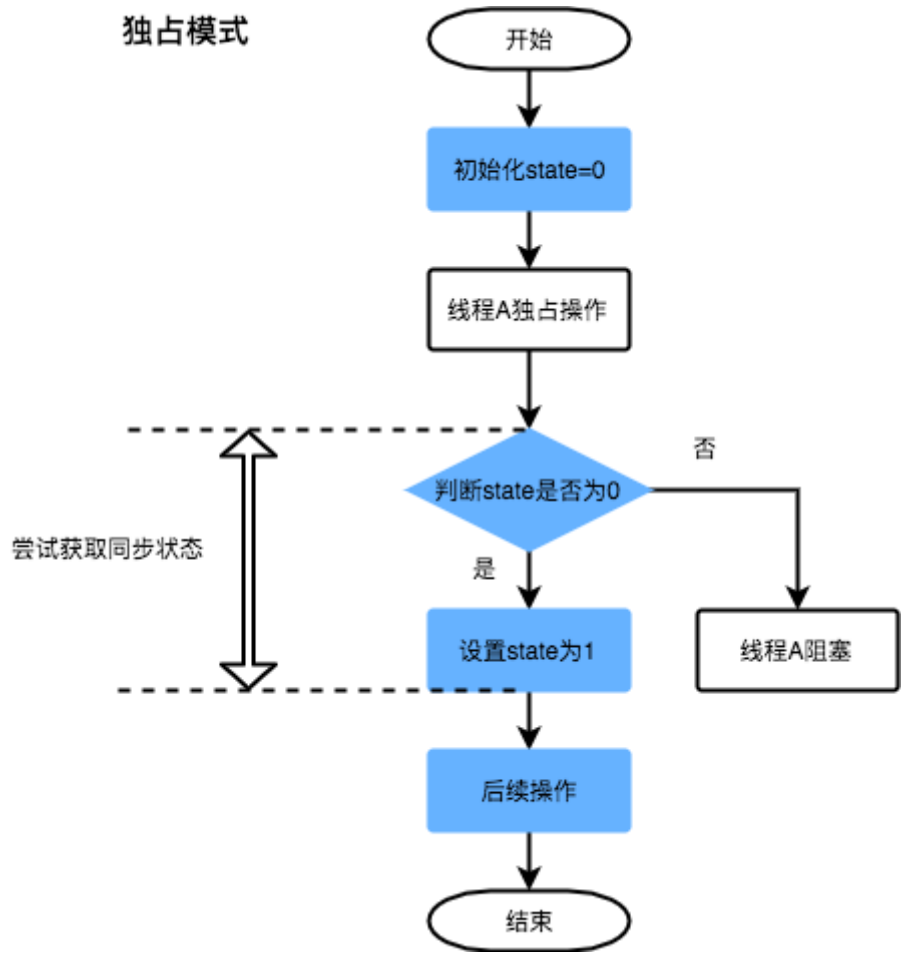
了解一下AQS的同步状态——State。AQS中维护了一个名为state的字段，意为同步状态，是由Volatile修饰的，用于展示当前临界资源的获锁情况。

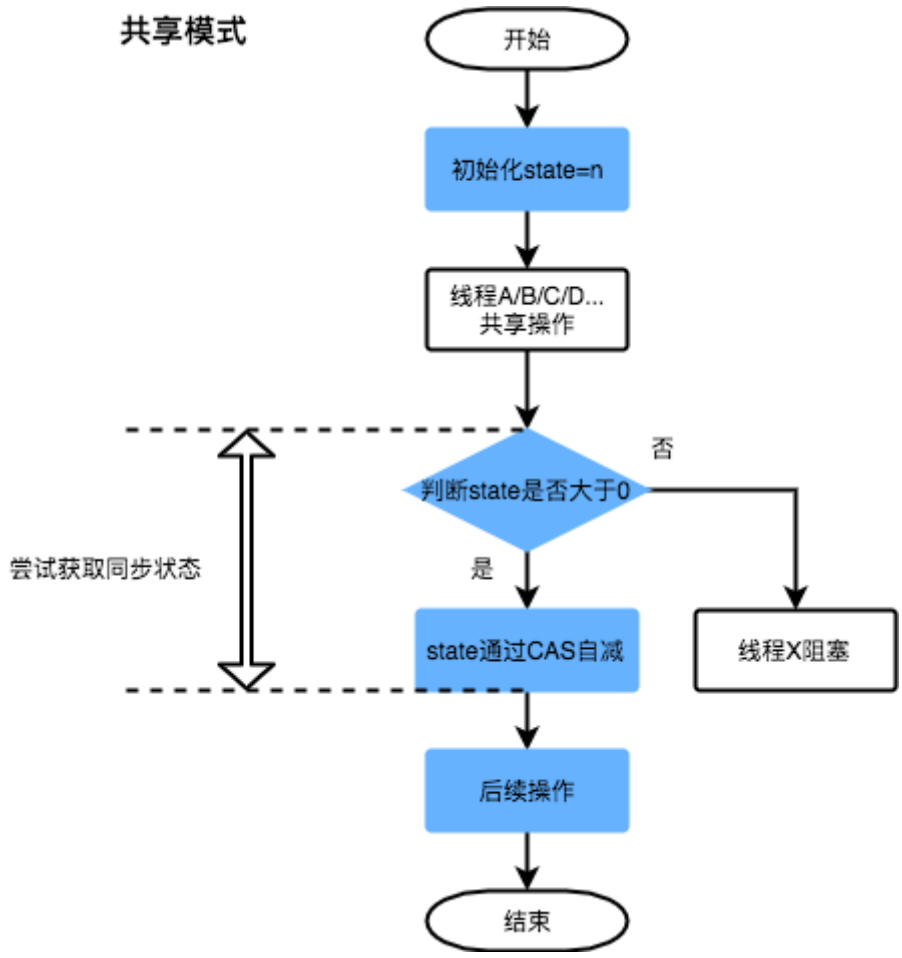
```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private volatile int state;
```

下面提供了几个访问这个字段的方法：

方法名	描述
protected final int getState()	获取State的值
protected final void setState(int newState)	设置State的值
protected final boolean compareAndSetState(int expect, int update)	使用CAS方式更新State

这几个方法都是Final修饰的，说明子类中无法重写它们。我们可以通过修改State字段表示的同步状态来实现多线程的独占模式和共享模式（加锁过程）。





3. 线程加入等待队列

ReentrantLock中公平锁和非公平锁在底层是相同的，这里以非公平锁为例进行分析。

在非公平锁中，有一段这样的代码：

```
// java.util.concurrent.locks.ReentrantLock

static final class NonfairSync extends Sync {
    ...
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
  ...
}
```

看一下这个Acquire是怎么写的：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

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

首先会调用 tryAcquire(arg) 方法，这个方法是需要同步组件自己实现的。 该方法保证线程安全的获取同步状态， tryAcquire(arg) 返回 true 表示获取成功也就正常退出了。否则会 构造同步节点（独占式 Node.EXCLUSIVE）并通过 `addWaiter(Node mode)` 方法将加入到同步队列的尾部，最后调用 `acquireQueued(final Node node, int arg)` 通过 “死循环”的方式获取同步状态。如果获取不到则阻塞节点中对应的线程，而被阻塞后的唤醒只能依靠前驱节点出队或者阻塞线程被中断来实现。

再看一下tryAcquire方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

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

可以看出，这里只是AQS的简单实现，具体获取锁的实现方法是由各自的公平锁和非公平锁单独实现的（以ReentrantLock为例）。如果该方法返回了True，则说明当前线程获取锁成功，就不用往后执行了；如果获取失败，就需要加入到等待队列中。

加入队列的时机

当执行Acquire(1)时，会通过tryAcquire获取锁。在这种情况下，如果获取锁失败，就会调用addWaiter加入到等待队列中去。

如 何加入队列

获取锁失败后，会执行addWaiter(Node.EXCLUSIVE)加入等待队列，具体实现方法如下：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

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;
}
private final boolean compareAndSetTail(Node expect, Node update) {
    return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
}
```

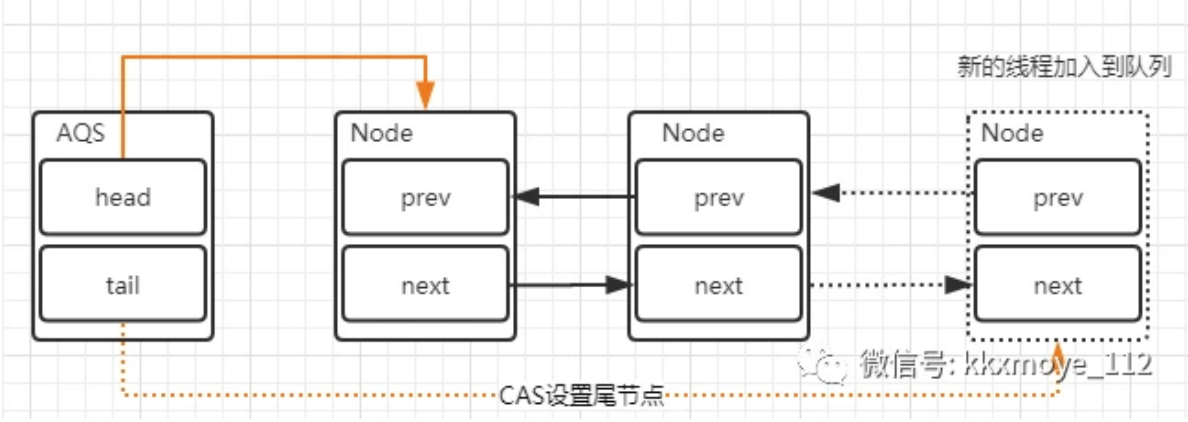
主要的流程如下：

- 通过当前的线程和锁模式新建一个节点。
- Pred指针指向尾节点Tail。
- 将New中Node的Prev指针指向Pred。
- 通过compareAndSetTail方法，完成尾节点的设置。这个方法主要是对tailOffset和Expect进行比较，如果tailOffset的Node和Expect的Node地址是相同的，那么设置Tail的值为Update的值。

当出现锁竞争以及释放锁的时候，AQS同步队列中的节点会发生变化，首先看一下添加节点的场景。

这里会涉及到两个变化

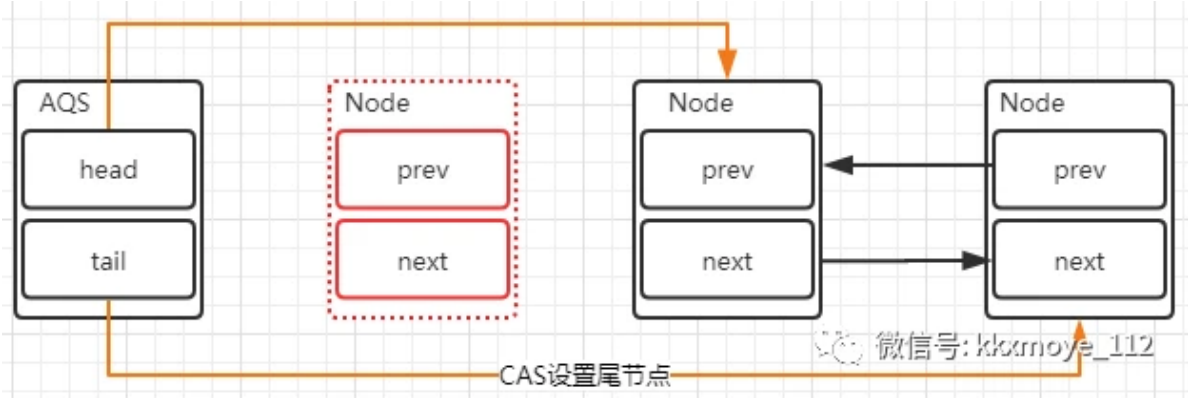
- 新的线程封装成Node节点追加到同步队列中，设置prev节点以及修改当前节点的前置节点的next节点指向自己
- 通过CAS讲tail重新指向新的尾部节点



4. 等待队列中线程出队列时机

前驱是头结点，就获取到了同步状态。

head节点表示获取锁成功的节点，当头结点在释放同步状态时，会唤醒后继节点，如果后继节点获得锁成功，会把自己设置为头结点，节点的变化过程如下



这个过程也是涉及到两个变化

- 修改head节点指向下一个获得锁的节点
- 新的获得锁的节点，将prev的指针指向null

这里有一个小的变化，就是设置head节点不需要用CAS，原因是设置head节点是由获得锁的线程来完成的，而同步锁只能由一个线程获得，所以不需要CAS保证，只需要把head节点设置为原首节点的后继节点，并且断开原head节点的next引用即可

代码设计

AQS的设计模式采用的模板方法模式，子类通过继承的方式，实现它的抽象方法来管理同步状态，对于子类而言它并没有太多的活要做，AQS提供了大量的模板方法来实现同步，主要是分为三类：独占式获取和释放同步状态、共享式获取和释放同步状态、查询同步队列中的等待线程情况。自定义子类使用AQS提供的模板方法就可以实现自己的同步语义。

独占式同步状态获取

acquire(int arg)方法为AQS提供的模板方法，该方法为独占式获取同步状态，但是该方法对中断不敏感，也就是说由于线程获取同步状态失败加入到CLH同步队列中，后续对线程进行中断操作时，线程不会从同步队列中移除。代码如下：

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

各个方法定义如下：

1. tryAcquire：去尝试获取锁，获取成功则设置锁状态并返回true，否则返回false。该方法自定义同步组件自己实现，该方法必须要保证线程安全的获取同步状态。
2. addWaiter：如果tryAcquire返回FALSE（获取同步状态失败），则调用该方法将当前线程加入到CLH同步队列尾部。
3. acquireQueued：当前线程会根据公平性原则来进行阻塞等待（自旋）,直到获取锁为止；并且返回当前线程在等待过程中有没有中断过。
4. selfInterrupt：产生一个中断。

独占式同步状态释放

当线程获取同步状态后，执行完相应逻辑后就需要释放同步状态。AQS提供了release(int arg)方法释放同步状态：

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

该方法同样是先调用自定义同步器自定义的tryRelease(int arg)方法来释放同步状态，释放成功后，会调用unparkSuccessor(Node node)方法唤醒后继节点（如何唤醒LZ后面介绍）。 这里稍微总结下：

> 在AQS中维护着一个FIFO的同步队列，当线程获取同步状态失败后，则会加入到这个CLH同步队列的对尾并一直保持着自旋。在CLH同步队列中的线程在自旋时会判断其前驱节点是否为首节点，如果为首节点则不断尝试获取同步状态，获取成功则退出CLH同步队列。当线程执行完逻辑后，会释放同步状态，释放后会唤醒其后继节点。

共享式同步状态获取

AQS提供acquireShared(int arg)方法共享式获取同步状态：

```
    public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0)
            //获取失败，自旋获取同步状态
            doAcquireShared(arg);
    }
```

从上面程序可以看出，方法首先是调用tryAcquireShared(int arg)方法尝试获取同步状态，如果获取失败则调用doAcquireShared(int arg)自旋方式获取同步状态，共享式获取同步状态的标志是返回 >= 0 的值表示获取成功。

获取同步状态如下：

```
    private void doAcquireShared(int arg) {
        /共享式节点
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                //前驱节点
                final Node p = node.predecessor();
                //如果其前驱节点，获取同步状态
                if (p == head) {
                    //尝试获取同步
                    int r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        if (interrupted)
                            selfInterrupt();
                        failed = false;
                        return;
                    }
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                        parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

tryAcquireShared(int arg)方法尝试获取同步状态，返回值为int，当其 >= 0 时，表示能够获取到同步状态，这个时候就可以从自旋过程中退出。 acquireShared(int arg)方法不响应中断，与独占式相似，AQS也提供了响应中断、超时的方法，分别是：acquireSharedInterruptibly(int arg)、tryAcquireSharedNanos(int arg,long nanos)，这里就不做解释了。

共享式同步状态释放

获取同步状态后，需要调用release(int arg)方法释放同步状态，方法如下：

```
    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }
```

因为可能会存在多个线程同时进行释放同步状态资源，所以需要确保同步状态安全地成功释放，一般都是通过CAS和循环来完成的。

疑问

Q：某个线程获取锁失败的后续流程是什么呢?

A：存在某种排队等候机制，线程继续等待，仍然保留获取锁的可能，获取锁流程仍在继续。

Q：既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢?

A：是CLH变体的FIFO双端队列。

Q：处于排队等候机制中的线程，什么时候可以有机会获取锁呢?

A：前驱结点是头结点，并且当前线程获取锁成功

Q：如果处于排队等候机制中的线程一直无法获取锁，需要一直等待么？还是有别的策略来解决这一问题?

A：线程所在节点的状态会变成取消状态，取消状态的节点会从队列中释放

Q：Lock函数通过Acquire方法进行加锁，但是具体是如何加锁的呢?

A：AQS的Acquire会调用tryAcquire方法，tryAcquire由各个自定义同步器实现，通过tryAcquire完成加锁过程。

- 那AQS只能用来实现独占且公平锁吗？显然不是，AQS又是如何实现非公平锁和共享锁的呢？ 其实AQS无论用来实现什么锁，这些锁本质的区别就是在于获取共享资源访问权的方式不同 ，而独占且公平的锁很明显获取访问权的方式是通过FIFO队列的顺序（即请求访问共享资源的顺序），而共享锁也是一样，只是可以获取访问权的线程数多了些；那么非公平锁是如何实现的呢？其实也很简单，就是舍弃队列的FIFO特性，只要持有共享资源的线程释放了锁，所有的在同步队列中的线程都会通过CAS操作去竞争锁；

ReentrantLock

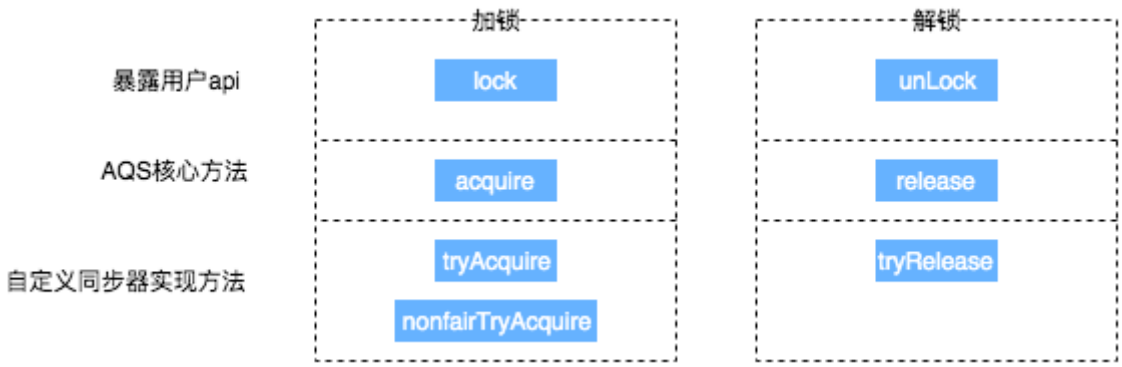
加锁：

- 通过ReentrantLock的加锁方法Lock进行加锁操作。
- 会调用到内部类Sync的Lock方法，由于Sync#lock是抽象方法，根据ReentrantLock初始化选择的公平锁和非公平锁，执行相关内部类的Lock方法，本质上都会执行AQS的Acquire方法。
- AQS的Acquire方法会执行tryAcquire方法，但是由于tryAcquire需要自定义同步器实现，因此执行了ReentrantLock中的tryAcquire方法，由于ReentrantLock是通过公平锁和非公平锁内部类实现的tryAcquire方法，因此会根据锁类型不同，执行不同的tryAcquire。
- tryAcquire是获取锁逻辑，获取失败后，会执行框架AQS的后续逻辑，跟ReentrantLock自定义同步器无关。

解锁：

- 通过ReentrantLock的解锁方法Unlock进行解锁。
- Unlock会调用内部类Sync的Release方法，该方法继承于AQS。
- Release中会调用tryRelease方法，tryRelease需要自定义同步器实现，tryRelease只在ReentrantLock中的Sync实现，因此可以看出，释放锁的过程，并不区分是否为公平锁。
- 释放成功后，所有处理由AQS框架完成，与自定义同步器无关。

通过上面的描述，大概可以总结出ReentrantLock加锁解锁时API层核心方法的映射关系。



非公平锁

非公平锁则没有这些规则，是**抢占模式**，每来一个人不会去管队列如何，直接尝试获取锁。

```
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    final void lock() {
        // 不管是否有线程在AQS的FIFO队列中排队等待，直接执行一次CAS操作竞争锁
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
        // CAS失败，则准备进入FIFO队列，在进入队列之前，还有一次机会，
        // AQS的acquire方法通过调用tryAcquire再给当前线程一次机会，此时再失败则进入队列等待
            acquire(1);
    }

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

非公平模式下每个线程都有2次机会(CAS操作)插队竞争锁，2次均失败之后才会进入FIFO队列等待，然后公平锁模式下，线程是不允许插队竞争锁的， 只要FIFO队列中有线程在等待，则当前竞争锁的线程必须进入队列等待，这就是为什么公平锁的吞吐比非公平锁低的原因。

重要的区别是在尝试获取锁时`tryAcquire(arg)`，非公平锁是不需要判断队列中是否还有其他线程，也是直接尝试获取锁：

```
    final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            //没有 !hasQueuedPredecessors() 判断
            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;
    }
```

公平锁

首先看下获取锁的过程：

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

可以看到是使用 `sync` 的方法，而这个方法是一个抽象方法，具体是由其子类(`FairSync`)来实现的，以下是公平锁的实现:

```
        final void lock() {
            acquire(1);
        }

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

第一步是尝试获取锁(`tryAcquire(arg)`),这个也是由其子类实现：

```
        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;
        }
    }
```

首先会判断 `AQS` 中的 `state` 是否等于 0，0 表示目前没有其他线程获得锁，当前线程就可以尝试获取锁。

注意:尝试之前会利用 `hasQueuedPredecessors()` 方法来判断 AQS 的队列中中是否有其他线程，如果有则不会尝试获取锁(**这是公平锁特有的情况**)。

如果队列中没有线程就利用 CAS 来将 AQS 中的 state 修改为1，也就是获取锁，获取成功则将当前线程置为获得锁的独占线程(`setExclusiveOwnerThread(current)`)。

如果 `state` 大于 0 时，说明锁已经被获取了，则需要判断获取锁的线程是否为当前线程(`ReentrantLock` 支持重入)，是则需要将 `state + 1`，并将值更新。

写入队列

如果 `tryAcquire(arg)` 获取锁失败，则需要用 `addWaiter(Node.EXCLUSIVE)` 将当前线程写入队列中。

写入之前需要将当前线程包装为一个 `Node` 对象(`addWaiter(Node.EXCLUSIVE)`)。

释放锁

公平锁和非公平锁的释放流程都是一样的：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    // 非持有锁的线程调用此方法直接抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 状态为0，表示锁完全释放，此时需清除AOS中的线程记录
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

首先会判断当前线程是否为获得锁的线程，由于是重入锁所以需要将 `state` 减到 0 才认为完全释放锁。

释放之后需要调用 `unparkSuccessor(h)` 来唤醒被挂起的线程。

参考

https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html

https://xie.infoq.cn/article/7e9a2689d223acaab1636f93d

http://cmsblogs.com/?hmsr=toutiao.io&p=2197&utm_medium=toutiao.io&utm_source=toutiao.io

https://blog.csdn.net/zl1zl2zl3/article/details/82215563

https://youendless.com/post/reentrantlock/