Java并发之AQS

**一、AQS是什么？有什么用？**

AQS全称 AbstractQueuedSynchronizer ，即抽象的队列同步器，是一种用来构建锁和同步器的框架。

基于AQS构建同步器：

ReentrantLock Semaphore CountDownLatch ReentrantReadWriteLock SynchronusQueue FutureTask

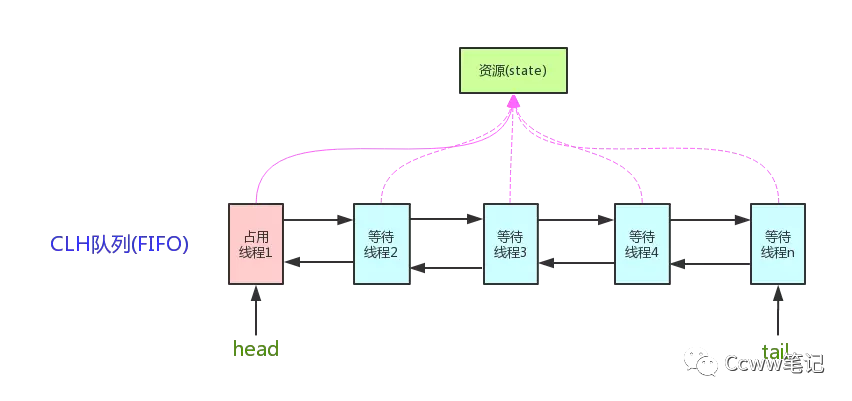
优势：

AQS 解决了在实现同步器时涉及的大量细节问题，例如自定义标准同步状态、FIFO 同步队列。基于 AQS 来构建同步器可以带来很多好处。它不仅能够极大地减少实现工作，而且也不必处理在多个位置上发生的竞争问题。

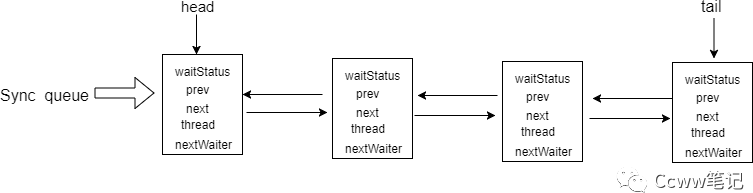
# 二、AQS核心知识

## AQS核心思想

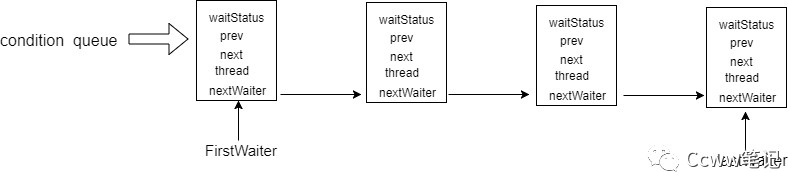
如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设 置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机 制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。如图所示：



Sync queue： 同步队列，是一个双向列表。包括head节点和tail节点。head节点主要用作后续的调度。



Condition queue： 非必须，单向列表。当程序中存在cindition的时候才会存在此列表。



## AQS设计思想

AQS使用一个int成员变量来表示同步状态

使用Node实现FIFO队列，可以用于构建锁或者其他同步装置

AQS资源共享方式：独占Exclusive（排它锁模式）和共享Share（共享锁模式）

AQS它的所有子类中，要么实现并使用了它的独占功能的api，要么使用了共享锁的功能，而不会同时使用两套api，即便是最有名的子类ReentrantReadWriteLock也是通过两个内部类读锁和写 锁分别实现了两套api来实现的

## state状态

state状态使用volatile int类型的变量，表示当前同步状态。state的访问方式有三种: getState()

setState()

compareAndSetState()

## AQS中Node常量含义

CANCELLEDwaitStatus值为1时表示该线程节点已释放（超时、中断），已取消的节点不会再阻 塞。

SIGNALwaitStatus为-1时表示该线程的后续线程需要阻塞，即只要前置节点释放锁，就会通知标 识为 SIGNAL 状态的后续节点的线程

CONDITIONwaitStatus为-2时，表示该线程在condition队列中阻塞（Condition有使用）

PROPAGATEwaitStatus为-3时，表示该线程以及后续线程进行无条件传播（CountDownLatch中 有使用）共享模式下， PROPAGATE 状态的线程处于可运行状态

## 同步队列为什么称为FIFO呢？

因为只有前驱节点是head节点的节点才能被首先唤醒去进行同步状态的获取。当该节点获取到同步状态时，它会清除自己的值，将自己作为head节点，以便唤醒下一个节点。

## Condition队列

除了同步队列之外，AQS中还存在Condition队列，这是一个单向队列。调用ConditionObject.await() 方法，能够将当前线程封装成Node加入到Condition队列的末尾，然后将获取的同步状态释放（即修改同步状态的值，唤醒在同步队列中的线程）。

Condition队列也是FIFO。调用ConditionObject.signal()方法，能够唤醒ﬁrstWaiter节点，将其添 加到同步队列末尾。

## 自定义同步器的实现

在构建自定义同步器时，只需要依赖AQS底层再实现共享资源state的获取与释放操作即可。自定义同步器实现时主要实现以下几种方法：

isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用 资 源 ； 正 数 表 示 成 功 ， 且 有 剩 余 资 源 。 tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true， 否则返回false。

# 三 AQS实现细节

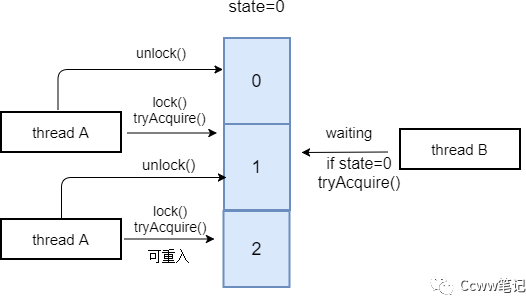
线程首先尝试获取锁，如果失败就将当前线程及等待状态等信息包装成一个node节点加入到FIFO 队列中。 接着会不断的循环尝试获取锁，条件是当前节点为head的直接后继才会尝试。如果失败就会阻塞自己直到自己被唤醒。而当持有锁的线程释放锁的时候，会唤醒队列中的后继线程。

## 独占模式下的AQS

所谓独占模式，即只允许一个线程获取同步状态，当这个线程还没有释放同步状态时，其他线程是获取 不了的，只能加入到同步队列，进行等待。

很明显，我们可以将state的初始值设为0，表示空闲。当一个线程获取到同步状态时，利用CAS操作让state加1，表示非空闲，那么其他线程就只能等待了。释放同步状态时，不需要CAS操作，因为独占模式下只有一个线程能获取到同步状态。ReentrantLock、CyclicBarrier正是基于此设计 的。

例如，ReentrantLock，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该 锁并将state+1。



独占模式下的AQS是不响应中断的，指的是加入到同步队列中的线程，如果因为中断而被唤醒的话，不 会立即返回，并且抛出InterruptedException。而是再次去判断其前驱节点是否为head节点，决定是否 争抢同步状态。如果其前驱节点不是head节点或者争抢同步状态失败，那么再次挂起。

# 独占模式获取资源-acquire方法

acquire以独占exclusive方式获取资源。如果获取到资源，线程直接返回，否则进入等待队列，直到获 取到资源为止，且整个过程忽略中断的影响。源码如下：

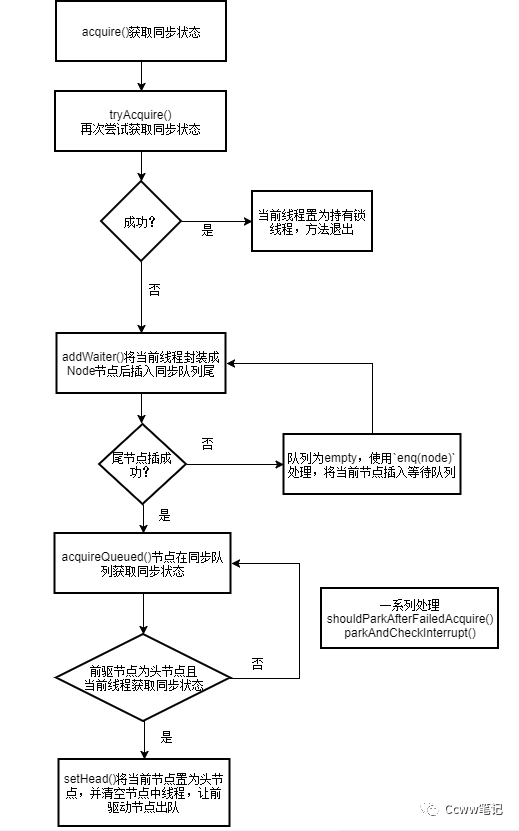
public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) selfInterrupt();

}

流程图：



调用自定义同步器的tryAcquire() 尝试直接去获取资源，如果成功则直接返回； 没成功，则addWaiter() 将该线程加入等待队列的尾部，并标记为独占模式；

acquireQueued() 使线程在等待队列中休息，有机会时（轮到自己，会被unpark() ）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。

如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断

selfInterrupt() ，将中断补上。

## 独占模式获取资源-tryAcquire方法

tryAcquire 尝试以独占的方式获取资源，如果获取成功，则直接返回true ，否则直接返回false ， 且具体实现由自定义AQS的同步器实现的。

protected boolean tryAcquire(int arg) {

throw new

UnsupportedOperationException(); }

## 独占模式获取资源-addWaiter方法

根据不同模式( Node.EXCLUSIVE 互斥模式、 Node.SHARED 共享模式)创建结点并以CAS的方式将当前线程节点加入到不为空的等待队列的末尾(通过compareAndSetTail() 方法)。如果队列为空，通过enq(node) 方法初始化一个等待队列，并返回当前节点。

/\*\* \* 参数

\* 返回值

\* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared

\* @return the new node \*/ private Node addWaiter(Node mode) {

//

将当前线程以指定的模式创建节点node Node node = new Node(Thread.currentThread(),

mode); // Try the fast path of enq; backup to full enq on failure // 获

取当前同队列的尾节点

if (pred != null) {

Node pred = tail;

//队列不为空，将新的node加入等待队列中

node.prev = pred;

入队列中

node;

失败时会调用enq方法处理

if (compareAndSetTail(pred, node)) {

return node;

enq(node);

}

}

//CAS方式将当前节点尾插

pred.next =

//当队列为empty或者CAS

return node;

}

其中，队列为empty，使用enq(node) 处理，将当前节点插入等待队列，如果队列为空，则初始化当前队列。所有操作都是CAS自旋的方式进行，直到成功加入队尾为止。

private Node enq(final Node node) {

//不断自旋

Node t = tail; initialize

load，在第一次用的时候new

//当前队列为empty

for (;;) {

if (t == null) { // Must

//完成队列初始化操作，头结点中不放数据，只是作为起始标记，lazy-

if (compareAndSetHead(new Node()))

tail = head; } else {

//不断将当前节点使用CAS尾插入队列中直到成功为止

(compareAndSetTail(t, node)) {

node.prev = t; if

t.next = node;

return t; } } } }

## 独占模式获取资源-acquireQueued方法

acquireQueued 用于已在队列中的线程以独占且不间断模式获取state状态，直到获取锁后返回。主要流程：

结点node进入队列尾部后，检查状态；

调用park()进入waiting状态，等待unpark()或interrupt()唤醒；

被唤醒后，是否获取到锁。如果获取到，head指向当前结点，并返回从入队到获取锁的整个过程中是否被中断过；如果没获取到，继续流程1

final boolean acquireQueued(final Node node, int arg) {

认为true 即为尚未

中断过的标记

//获取前节点

boolean failed = true;

boolean interrupted = false;

try {

//是否已获取锁的标志，默

//等待中是否被for (;;) {

final Node p = node.predecessor();

当前节点已经成为头结点，尝试获取锁（tryAcquire）成功，然后返回

//如果

if (p ==

head && tryAcquire(arg)) { setHead(node);

p.next = null; // help GC failed = false;

return interrupted; } //shouldParkAfterFailedAcquire根据对当前节点的前一个节点的状态进行判断，对当前节点做出不同的操作

//parkAndCheckInterrupt让线程进入等待状态，并检查当前线程是否被可以被中断 if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt()) interrupted = true;

} finally { //将当前节点设置为取消状态；取消状态设置为1

}

if (failed)

cancelAcquire(node); } }

# 独占模式释放资源-release方法

release方法是独占exclusive模式下线程释放共享资源的锁。它会调用tryRelease()释放同步资源， 如果全部释放了同步状态为空闲（即state=0）,当同步状态为空闲时，它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock().

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

unparkSuccessor(h);

if (h != null && h.waitStatus != 0)

return true;

}

return false; }

## 独占模式释放资源-tryRelease方法

tryRelease() 跟tryAcquire() 一样实现都是由自定义定时器以独占exclusive模式实现的。因为其是独占模式，不需要考虑线程安全的问题去释放共享资源，直接减掉相应量的资源即可(state-=arg)。而且tryRelease() 的返回值代表着该线程是否已经完成资源的释放，因此在自定义同步器的tryRelease() 时，需要明确这条件，当已经彻底释放资源(state=0)，要返回true，否则返回false。

protected boolean tryRelease(int arg) {

throw new

UnsupportedOperationException(); }

ReentrantReadWriteLock的实现：

protected final boolean tryRelease(int releases) {

if

(!isHeldExclusively()) throw new IllegalMonitorStateException();

//减掉相应量的资源(state-=arg) int nextc = getState() - releases;

//是否完全释放资源

if (free) setState(nextc);

boolean free = exclusiveCount(nextc) == 0;

setExclusiveOwnerThread(null);

return free;

}

## 独占模式释放资源-unparkSuccessor方法

unparkSuccessor 用unpark()唤醒等待队列中最前驱的那个未放弃线程，此线程并不一定是当前节点的next节点，而是下一个可以用来唤醒的线程，如果这个节点存在，调用unpark()方法唤醒。

private void unparkSuccessor(Node node) {

//当前线程所在的结点node

int ws

= node.waitStatus; //置零当前线程所在的结点状态，允许失败 if (ws < 0)

compareAndSetWaitStatus(node, ws, 0); //找到下一个需要唤醒的结点Node s = node.next; if (s == null || s.waitStatus > 0) { s = null; // 从后向前找

node; t = t.prev)

if (t.waitStatus <= 0)

for (Node t = tail; t != null && t !=

//从这里可以看出，<=0的结点，都是还有效的结点

s = t;

}

if (s != null)

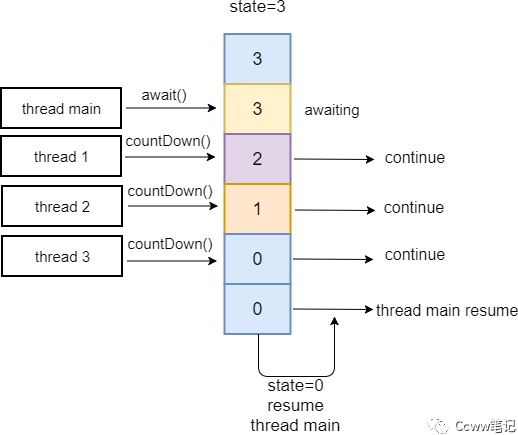
//唤醒 LockSupport.unpark(s.thread); }

## 共享模式下的AQS

共享模式，当然是允许多个线程同时获取到同步状态,共享模式下的AQS也是不响应中断的.

很明显，我们可以将state的初始值设为N（N > 0），表示空闲。每当一个线程获取到同步状态时，就利用CAS操作让state减1，直到减到0表示非空闲，其他线程就只能加入到同步队列，进行等待。释放同步状态时，需要CAS操作，因为共享模式下，有多个线程能获取到同步状态。CountDownLatch、Semaphore正是基于此设计的。

例如，CountDownLatch，任务分为N个子线程去执行，同步状态state也初始化为N（注意N要与线程 个数一致）：



## 共享模式获取资源-acquireShared方法

acquireShared 在共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg); }

流程：

先通过tryAcquireShared()尝试获取资源，成功则直接返回；

失败则通过doAcquireShared()中的park()进入等待队列，直到被unpark()/interrupt()并成功获取到资源才返回(整个等待过程也是忽略中断响应)。

## 共享模式获取资源-tryAcquireShared方法

tryAcquireShared() 跟独占模式获取资源方法一样实现都是由自定义同步器去实现。但AQS规范中已定义好tryAcquireShared() 的返回值：

负值代表获取失败；

0代表获取成功，但没有剩余资源；

正数表示获取成功，还有剩余资源，其他线程还可以去获取。

protected int tryAcquireShared(int arg) {

throw new

UnsupportedOperationException(); }

## 共享模式获取资源-doAcquireShared方法

doAcquireShared() 用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。

private void doAcquireShared(int arg) {

//加入队列尾部

final Node node

= addWaiter(Node.SHARED); //是否成功标志

try { false;

取前驱节点

//等待过程中是否被中断过的标志

for (;;) {

boolean failed = true;

boolean interrupted =

final Node p = node.predecessor();//获

if (p == head) {//如果到head的下一个，因为head是拿到资源的线程，

此时node被唤醒，很可能是head用完资源来唤醒自己的 int r =

tryAcquireShared(arg);//尝试获取资源 if (r >= 0) {//成功setHeadAndPropagate(node, r);//将head指向自己，还有剩余资源可以再唤醒之后

的线程 p.next = null; // help GC if (interrupted)//如果等待过程中被打断过，此时将中断补上。

selfInterrupt(); failed = false;

return; } }

适合位置，进入waiting状态，等着被unpark()或interrupt()

(shouldParkAfterFailedAcquire(p, node) &&

//判断状态，队列寻找一个

if

parkAndCheckInterrupt())

} finally {

}

if (failed)

interrupted = true;

cancelAcquire(node);

}

}

## 共享模式释放资源-releaseShared方法

releaseShared() 用于共享模式下线程释放共享资源，释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。

public final boolean releaseShared(int arg) {

//尝试释放资源

if

(tryReleaseShared(arg)) {

return true; }

//唤醒后继结点

return false;

doReleaseShared();

}

独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这 主要是基于独占下可重入的考量；而共享模式下的releaseShared()则没有这种要求，共享模式实 质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等 待结点。 https://[www.cnblogs.com/waterystone/p/4920797.html](http://www.cnblogs.com/waterystone/p/4920797.html)

## 共享模式释放资源-doReleaseShared方法

doReleaseShared() 主要用于唤醒后继节点线程,当state为正数，去获取剩余共享资源；当state=0时去获取共享资源。

private void doReleaseShared() { if (h != null && h != tail) {

if (ws == Node.SIGNAL) {

for (;;) {

Node h = head;

int ws = h.waitStatus;

if (!compareAndSetWaitStatus(h,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Node.SIGNAL,  继 | 0)) | unparkSuccessor(h); | | continue; | } | //唤醒后  else if (ws |
| == 0 && | continue; | | !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  } // head发生变化 if (h == | | | |
| head) | break; | | } } | | | |