一文搞懂到底什么是 AQS

前言

> 日常开发中，我们经常使用锁或者其他同步器来控制并发，那么它们的基础框架是什么呢？如何实现的同步功能呢？本文将详细讲解构建锁和同步器的基础框架--AQS，并根据源码分析其原理。

一、什么是 AQS?

1. AQS 简介

AQS（Abstract Queued Synchronizer），抽象队列同步器，它是用来构建锁或其他同步器的基础框架。虽然大多数程序员可能永远不会使用到它，但是知道 AQS 的原理有助于理解一些锁或同步器的是如何运行的。

那么有哪些同步器是基于 AQS 实现的呢？这里仅是简单介绍，详情后续会单独总结一篇文章。

同步器	说明
CountDownLatch	递减的计数器，直至所有线程的任务都执行完毕，才继续执行后续任务。
Semaphore	信号量，控制同时访问某个资源的数量。
CyclicBarrier	递增的计数器，所有线程达到屏障时，才会继续执行后续任务。
ReentrantLock	防止多个线程同时访问共享资源，类似 synchronized 关键

字。 |
ReentrantReadWriteLock	维护了读锁和写锁，读锁允许多线程访问，读锁阻塞所有线程。
Condition	提供类似 Object 监视器的方法，于 Lock 配合可以实现等待/通知模式。
FutureTask	当一个线程需要等待另一线程把某个任务执行完后才能继续执行，此时可以使用 FutureTask

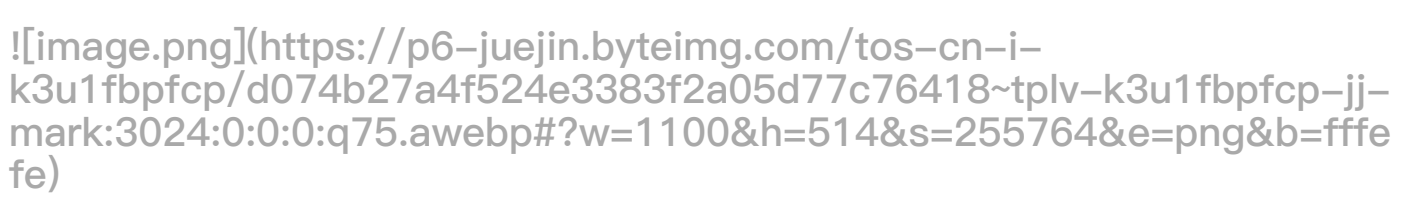
如果你理解了 AQS 的原理，也可以基于它去自定义一个同步组件。

2. AQS 数据结构

AQS 核心是通过对同步状态的管理，来完成线程同步，底层是**依赖一个双端队列来完成同步状态的管理**。

* 当前线程获取同步状态失败后，会构造成一个 Node 节点并加入队列末尾，同实阻塞线程。
* 当同步状态释放时，会把头节点中的线程唤醒，让其再次尝试获取同步状态

如下图，这里只是简单绘制，具体流程见下面原理分析：



这里的每个 Node 节点都存储着**当前线程、等待信息**等。

3. 资源共享模式

我们在获取共享资源时，有两种模式：

| 模式 | 说明 | 示例 |

| --- | --- | --- |
| 独占模式 | Exclusive，资源同一时刻只能被一个线程获取 | ReentrantLock |
| 共享模式 | Share，资源可同时被多个线程获取 | Semaphore、CountDownLatch |

二、AQS 原理分析

先简单说下原理分析的流程：

1. 同步状态相关源码；
2. 须重写的方法；
3. Node 节点结构分析；
4. 独占模式下的同步状态的获取与释放；
5. 共享模式下的同步状态的获取与释放；

1. 同步状态相关

上面介绍到， AQS 核心是通过对同步状态的管理，来完成线程同步，所以首先介绍管理同步状态的三个方法，在自定义同步组件时，需要通过它们获取和修改同步状态。

```
//保证可见性
private volatile int state

//获取当前同步状态。
protected final int getState() {
    return state;
}

//设置当前同步状态。
protected final void setState(int newState) {
    state = newState;
}

//使用 CAS 设置当前状态，保证原子性。
protected final boolean compareAndSetState(int expect, int update) {
    // See below for intrinsics setup to support this
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
```

}

```

## 2. 须重写的方法

AQS 是**基于模板方法模式**的，通过第一个 abstract 也可知道，AQS 是个抽象类，使用者需要继承 AQS 并重写指定方法。

以下这些方式是没有具体实现的，需要在使用 AQS 时在子类中去实现具体方法，等到介绍一些同步组件时，会详细说明如何重写。

```
//独占式获取同步状态，实现该方法须查询并判断当前状态是否符合预期，然后再进行CAS设置状态。
protected boolean tryAcquire (int arg)

//独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态。
protected boolean tryRelease (int arg)

//共享式获取同步状态，返回大于0的值表示获取成功，反之获取失败。
protected int tryAcquireShared (int arg)

//共享式释放同步状态。
protected boolean tryReleaseShared (int arg)

//当前同步器是否再独占模式下被线程占用，一般用来表示是否被当前线程独占。
protected boolean isHeldExclusively ()

```

## 3. Node 源码

Node 是双端队列中的节点，是数据结构的重要部分，线程相关的信息都存在每一个 Node 中。
```

3.1 Node 结构源码

源码如下：

```
static final class Node {
    //标记当前节点的线程在共享模式下等待。
    static final Node SHARED = new Node();

    //标记当前节点的线程在独占模式下等待。
    static final Node EXCLUSIVE = null;

    //waitStatus的值，表示当前节点的线程已取消（等待超时或被中断）
    static final int CANCELLED =  1;

    //waitStatus的值，表示后继节点的线程需要被唤醒
    static final int SIGNAL    = -1;

    //waitStatus的值，表示当前节点在等待某个条件，正处于condition等待队列中
    static final int CONDITION = -2;

    //waitStatus的值，表示在当前有资源可用，能够执行后续的acquireShared操作
    static final int PROPAGATE = -3;

    //等待状态，值如上，1、-1、-2、-3。
    volatile int waitStatus;

    //前趋节点
    volatile Node prev;

    //后继节点
    volatile Node next;

    //当前线程
    volatile Thread thread;

    //等待队列中的后继节点，共享模式下值为SHARED常量
    Node nextWaiter;

    //判断共享模式的方法
    final boolean isShared() {
        return nextWaiter == SHARED;
    }
```

```
    //返回前趋节点，没有报NPE
    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
        else
            return p;
    }

    //下面是三个构造方法
    Node() {}    // Used to establish initial head or SHARED marke

    Node(Thread thread, Node mode) {     // Used by addWaiter
        this.nextWaiter = mode;
        this.thread = thread;
    }
    Node(Thread thread, int waitStatus) { // Used by Condition
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}
```

3.2 设置头尾节点

Unsafe 类中，提供了一个基于 CAS 的设置头尾节点的方法，AQS 调用该方法进行设置头尾节点，保证并发编程中的线程安全。

```
//CAS自旋设置头节点
private final boolean compareAndSetHead(Node update) {
    return unsafe.compareAndSwapObject(this, headOffset, null, update);
}

//CAS自旋设置尾节点，expect为当前线程“认为”的尾节点，update为当前节点
private final boolean compareAndSetTail(Node expect, Node update) {
    return unsafe.compareAndSwapObject(this, tailOffset, expect,
update);
}
```

```

4. 独占模式
--------

资源同一时刻只能被一个线程获取，如 ReentrantLock。

### 4.1 获取同步状态

代码如下，调用 acquire 方法可以获取同步状态，底层就是调用须重写方法中的 tryAcquire。如果获取失败则进入同步队列中，即使后续对线程进行终端操作，线程也不会从同步队列中移除。

```
public final void acquire(int arg) {
    //调用须重写方法中的tryAcquire
    if (!tryAcquire(arg) &&
        //失败则进入同步队列中
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

获取失败会先调用 addWaiter 方法将当前线程封装成独占式模式的节点，添加到AQS的队列尾部，源码如下。

```
private Node addWaiter(Node mode) {
    //将当前线程封装成对应模式下的Node节点
    Node node = new Node(Thread.currentThread(), mode);

    Node pred = tail;//尾节点
    if (pred != null) {
        //双端队列需要两个指针指向
        node.prev = pred;
        //通过CAS方式
```
```

```
        if (compareAndSetTail(pred, node)) {
            //添加到队列尾部
            pred.next = node;
            return node;
        }
    }
    //等待队列中没有节点，或者添加队列尾部失败则调用end方法
    enq(node);
    return node;
}

//Node节点通过CAS自旋的方式被添加到队列尾部，直到添加成功为止。
private Node enq(final Node node) {
    //死循环，类似 while(1)
    for (;;) {
        Node t = tail;
        if (t == null) { // 须要初始化，代表队列的第一个元素
            if (compareAndSetHead(new Node()))
                //头节点就是尾节点
                tail = head;
        } else {
            //双端队列需要两个指针指向
            node.prev = t;
            //通过自旋放入队列尾部
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

此时，通过 addWaiter 已经将当前线程封装成独占模式的 Node 节点，并成功放入队列尾部。接下来会调用acquireQueued 方法在等待队列中排队。

```
final boolean acquireQueued(final Node node, int arg) {
    //获取资源失败标识
    boolean failed = true;
    try {
        //线程是否被中断标识
        boolean interrupted = false;
        //死循环，类似 while(1)
        for (;;) {
```

```
            //获取当前节点的前趋节点
            final Node p = node.predecessor();

            //前趋节点是head，即队列的第二个节点，可以尝试获取资源
            if (p == head && tryAcquire(arg)) {
                //资源获取成功将当前节点设置为头节点
                setHead(node);
                p.next = null; // help GC，表示head节点出队列
                failed = false;
                return interrupted;
            }
            //判断当前线程是否可以进入waitting状态，详解见下方
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())//阻塞当前线程，详解见下方
                interrupted = true;
        }
    } finally {
        if (failed)
            //取消获取同步状态，源码见下方的取消获取同步状态章节
            cancelAcquire(node);
    }
}

//将当前节点设置为头节点
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}

//判断当前线程是否可以进入waitting状态
private static boolean shouldParkAfterFailedAcquire(Node pred, Node
node) {
    //获取前趋节点的等待状态，含义见上方Node结构源码
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)//表示当前节点的线程需要被唤醒
        return true;
    if (ws > 0) {//表示当前节点的线程被取消

        //则当前节点一直向前移动，直到找到一个waitStatus状态小于或等于
0的节点
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        //排在这个节点的后面
        pred.next = node;
    } else {
        //通过CAS设置等待状态
```

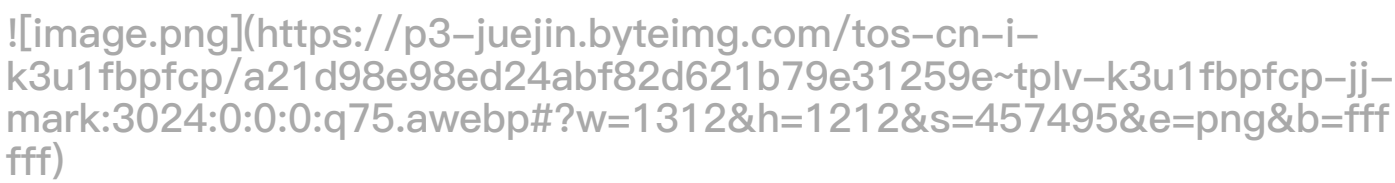
```
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }

    //阻塞当前线程
    private final boolean parkAndCheckInterrupt() {
        //底层调用的UnSafe类的方法 park：阻塞当前线程, unpark：使给定的线程停止阻塞
        LockSupport.park(this);
        //中断线程
        return Thread.interrupted();
    }
```

acquireQueued 方法中，**只有当前驱节点等于 head 节点时，才能够尝试获取同步状态**，这时为什么呢?

因为 head 节点是占有资源的节点，它释放后才会唤醒它的后继节点，所以需要检测。还有一个原因是因为如果遇到了非 head 节点的其他节点出队或因中断而从等待中唤醒，这时种情况则需要判断前趋节点是否为 head 节点，是才允许获取同步状态。

获取同步状态的整体流程图如下：



4.2 释放同步状态

调用须重写方法中的 tryAcquire 进行同步状态的释放，成功则唤醒队列中最前面的线程，具体如下。

```
public final boolean release(int arg) {
    //调用须重写方法中的tryRelease
```

```
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            //唤醒后继节点的线程，详情见下方
            unparkSuccessor(h);
        return true;
    }
    return false;
}

//唤醒后继节点的线程
private void unparkSuccessor(Node node) {
    //获取当前节点的等待状态
    int ws = node.waitStatus;
    if (ws < 0)
        //小于0则，则尝试CAS设为0
        compareAndSetWaitStatus(node, ws, 0);

    //获取后继节点
    Node s = node.next;

    //后继节点为空或者等待状态大于0，代表被节点被取消
    if (s == null || s.waitStatus > 0) {
        s = null;
        //将队列中的所有节点都向前移动
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    //不为空则进行唤醒操作
    if (s != null)
        //底层调用的UnSafe类的方法 park：阻塞当前线程, unpark：使给定的线程停止阻塞
        LockSupport.unpark(s.thread);
}
```

4.3 其他情况的获取同步状态

除此之外，独占模式下 AQS 还提供了两个获取同步状态的方法，**可中断的获取同步状态和超时获取同步状态。**

acquire 方法获取锁失败的线程是不能被 interrupt 方法中断的，所以提供了另一个方法 ，从而让获取锁失败等待的线程可以被中断。底层源码与

```
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())//中断则抛出异常
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}
```

通过调用 tryAcquireNanos 可以在超时时间内获取同步状态，可以理解为是上述中断获取同步状态的增强版。

```
public final boolean tryAcquireNanos(int arg, long nanosTimeout)
        throws InterruptedException {
    if (Thread.interrupted())//中断则抛出异常
        throw new InterruptedException();
    return tryAcquire(arg) ||
        doAcquireNanos(arg, nanosTimeout);
}
```

上面个两个方法的源码均与普通的独占获取同步状态的源码基本类似，感兴趣的话可以自行阅读，这里不做赘述。

5. 共享模式

资源可同时被多个线程获取，如 Semaphore、CountDownLatch。

5.1 获取同步状态

代码如下，调用 acquireShared 方法可以获取同步状态，底层就是先调用须重

写方法中的 tryAcquireShared。

tryAcquireShared 返回值的含义：

* 负数：表示获取资源失败
* 0：表示获取资源成功，但是没有剩余资源
* 正数：表示获取资源成功，还有剩余资源

```
public final void acquireShared(int arg) {
    //调用须重写方法中的tryAcquireShared
    if (tryAcquireShared(arg) < 0)
        //获取资源失败，将当前线程放入队列的尾部并阻塞
        doAcquireShared(arg);
}
```

若获取资源失败，调用如下方法将当前线程放入队列的尾部并阻塞，直到有其他线程释放资源并唤醒当前线程。

```
//部分方法与独占模式下的方法公用，这里不再重复说明，详情见独占模式下的获取同步状态源码。
private void doAcquireShared(int arg) {
    //将当前线程封装成独占式模式的节点，添加到AQS的队列尾部，源码在独占模式中已分析。
    final Node node = addWaiter(Node.SHARED);

    //获取资源失败标识
    boolean failed = true;
    try {
        //线程被打断表示
        boolean interrupted = false;

        //死循环，类似 while(1)
        for (;;) {
            //获取当前节点的前趋节点
            final Node p = node.predecessor();
            //前趋节点是head，即队列的第二个节点，可以尝试获取资源
            if (p == head) {
                int r = tryAcquireShared(arg);
```

```
                if (r >= 0) {
                    //将当前节点设置为头节点，若还有剩余资源，则继续唤醒队列
中后面的线程。
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC 表示head节点出队列
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
            //判断当前线程是否可以进入waitting状态，源码在独占模式中已分析
。
            if (shouldParkAfterFailedAcquire(p, node) &&
                //阻塞当前线程，源码在独占模式中已分析。
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            //取消获取同步状态，源码见下方的取消获取同步状态章节
            cancelAcquire(node);
    }
}

/*
 * propagate就是tryAcquireShared的返回值
 *● 负数：表示获取资源失败
 *● 0：表示获取资源成功，但是没有剩余资源
 *● 正数：表示获取资源成功，还有剩余资源
 */
private void setHeadAndPropagate(Node node, int propagate) {
    //将当前节点设置为头节点，源码在独占模式中已分析。
    Node h = head; //这时的h是旧的head
    setHead(node);

    // propagate > 0：还有剩余资源
    // h == null 和 h = head) == null: 不会成立，因为addWaiter已执行
    // waitStatus < 0：若没有剩余资源，但waitStatus又小于0，表示可能有
新资源释放
    // 括号中的 waitStatus < 0: 这里的 h 是此时的新的head（当前节点），
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {

        //获取当前节点的后继节点
        Node s = node.next;

        //后继节点不存在或者是共享锁都需要唤醒，可理解为只要后继节点不是
```

```
独占模式，都要唤醒
        //可能会导致不必要的唤醒
        if (s == null || s.isShared())
            //唤醒操作在此方法中，详情见下方的释放源码
            doReleaseShared();
    }
}

```

5.2 释放同步状态

代码如下，调用 releaseShared 方法可以释放同步状态，底层就是先调用须重写方法中的 tryReleaseShared。

```
public final boolean releaseShared(int arg) {
    //调用须重写方法中的tryReleaseShared
    if (tryReleaseShared(arg)) {
        //尝试释放资源成功，会继续唤醒队列中后面的线程。
        doReleaseShared();
        return true;
    }
    return false;
}

//唤醒队列中后面的线程
private void doReleaseShared() {

    //死循环，自旋操作
    for (;;) {
        //获取头节点
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            //signal表示后继节点需要被唤醒
            if (ws == Node.SIGNAL) {
                //自旋将头节点的waitStatus状态设置为0
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;            // loop to recheck cases

                //唤醒头节点的后继节点，源码见独占模式的释放
                unparkSuccessor(h);
            }
```

```
            //后继节点不需要唤醒，则把当前节点状态设置为PROPAGATE确保以后可以传递下去
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        //判断头节点是否变化，没有则退出循环。
        //有变化说明其他线程已经获取了同步状态，需要进行重试操作。
        if (h == head)                   // loop if head changed
            break;
    }
}

```

6. 取消获取同步状态

无论是独占模式还是共享模式，所有的程获取同步状态的过程中，如果发生异常或是超时唤醒等，都需要将当前的节点出队，源码如下。

```
//一般在获取同步状态方法的finally块中
private void cancelAcquire(Node node) {
    if (node == null)
        return;
    node.thread = null;//当前线程节点设为null
    Node pred = node.prev;//获取前驱节点

    //前趋节点为取消状态，向前遍历找到非取消状态的节点
    while (pred.waitStatus > 0)
        node.prev = pred = pred.prev;

    Node predNext = pred.next;//获取非取消节点的下一个节点

    node.waitStatus = Node.CANCELLED;//将当前节点的等待状态设为取消状态

    //当前节点是尾节点，则自旋将尾节点设置为前一个非取消节点
    if (node == tail && compareAndSetTail(node, pred)) {
        //将尾节点设为前一个非取消的节点，并将其后继节点设为null，help GC
        compareAndSetNext(pred, predNext, null);
    } else {
```

```
        int ws;//用于表示等待状态

        //pred != head：前一个非取消的节点非头节点也非尾节点
        //ws == Node.SIGNAL：当前等待状态为待唤醒
        //若不是待唤醒则CAS设置为待唤醒状态
        //前一个非取消的节点的线程不为null
        if (pred != head &&
            ((ws = pred.waitStatus) == Node.SIGNAL ||
             (ws <= 0 && compareAndSetWaitStatus(pred, ws,
Node.SIGNAL))) &&
            pred.thread != null) {
            //符合所有条件后，获取当前节点的后继节点
            Node next = node.next;
            if (next != null && next.waitStatus <= 0)
                //前一个非取消的节点的后继节点设为当前节点的后继节点
                //这样当前节点以及之前的已取消节点都会被移除
                compareAndSetNext(pred, predNext, next);
        } else {
            //前一个非取消的节点为头节点
            //唤醒后继节点的线程，详情见独占模式释放同步状态的源码
            //唤醒是为了执行shouldParkAfterFailedAcquire()方法，详解见上面
的acquireQueued源码
            //该方法中从后往前遍历找到第一个非取消的节点并将中间的移除队
列
            unparkSuccessor(node);
        }
        //移除当前节点
        node.next = node; // help GC
    }
}

```

三、总结
====

AQS 是用来构建锁或其他同步器的基础框架，底层是一个双端队列。支持独占和共享两种模式下的资源获取与释放，基于 AQS 可以自定义不同类型的同步组件。

在独占模式下，获取同步状态时，AQS 维护了一个双端队列，获取失败的线程都会被加入到队列中进行自旋，移出队列的条件就是前趋节点为 head 节点并成功获取同步状态。释放同步状态时，会唤醒 head 节点的后继节点。

在共享模式下，获取同步状态时，同样维护了一个双端队列，获取失败的的线程也会加入到队列中进行自旋，移除队列的条件也与独占模式一样。

但是在唤醒操作上，在资源数量足够的情况下，共享模式会将唤醒事件传递到后面的共享节点上，进行了后续节点的唤醒，解所成功后仍会唤醒后续节点。

关于 AQS 重要的几个组件的特点、原理以及对应的应用场景，后续会单独写一篇文章。若发现其他问题欢迎指正交流。

> 参考：
>
>
> [1] 翟陆续/薛宾田. Java 并发编程之美.
>
>
> [2] 方腾飞/魏鹏/程晓明. Java 并发编程的艺术.
>
>
> [3] Lev Vygotsky. Java 并发编程实践