AbstractQueuedSynchronizer超详细原理解析

2017-04-07 13:40 出处：清屏网 人气：97 评论（0）

今天我们来研究学习一下 `AbstractQueuedSynchronizer` 类的相关原理，`java.util.concurrent` 包中很多类都依赖于这个类所提供的队列式的同步器,比如说常用的 `ReentranLock` ， `Semaphore` 和 `CountDownLatch` 等.

为了方便理解,我们以一段使用 `ReentranLock` 的代码为例,讲解 `ReentranLock` 每个方法中有关 `AQS` 的使用.

ReentranLock示例

我们都知道 `ReentranLock` 的加锁行为和 `Synchronized` 类似,都是可重入的锁,但是二者的实现方式确实完全不同的,我们之后也会讲解 `Synchronized` 的原理. 除此之外,Synchronized的阻塞无法被中断,而ReentrantLock则提供了可中断的阻塞 下面的代码是 `ReentranLock` 的相关API,我们就以此为顺序,依次讲解.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
lock.unlock();
```

公平锁和非公平锁

ReentranLock分为公平锁和非公平锁.二者的区别就在获取锁是否和排队顺序相关.我们都知道,如果当前锁被另一个线程持有,那么当前申请锁的线程会被挂起等待,然后加入一个等待队列里.理论上,先调用 `lock` 函数被挂起等待的线程应该排在等待队列的前端,后调用的就排在后边.如果此时,锁被释放,需要通知等待线程再次尝试获取锁,公平锁会让最先进入队列的线程获得锁,而非公平锁则会唤醒所有线程,让它们再次尝试获取锁,所以可能会导致后来的线程先获得了锁,则就是非公平.

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

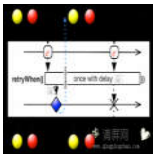
我们会发现 `FairSync` 和 `NonfairSync` 都继承了 `Sync` 类,而 `Sync` 的父类就是 `AbstractQueuedSynchronizer` .但是 `AQS` 的构造函数是空的,并没有任何操作.

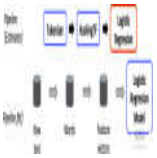
之后的源码分析,如果没有特别说明,就是指公平锁.

lock操作

`ReentranLock` 的 `lock` 函数如下所示,直接调用了 `sync` 的 `lock` 函数.也就是调用了 `FairSync` 的 `lock` 函数.

本类最热新闻


对RxJava中.repeatWhen()和.retr


MLlib1.6指南笔记

- RxDownload：基于RxJava打造的下载工具, 支持多...
- HoloLens开发手记：使用Visual Studio Using Vis...
- java集成pdf.js实现pdf文件在线预览
- RocketMQ源码三：Producer消息发送过程
- dotnet core搭建持续集成环境
- netty源码分析之writeAndFlush全解析
- Android教你一步步搭建MVP+Retrofit+RxJava网...
- Quartz.net开源job调度框架
- JavaWeb入门级项目实战：文章发布系统（第四节...
- A query was run and no Result Maps were foun...

48小时最热 7天最热 7天热评 月榜

小米5s顶配版/一加3t/华为p9哪款手机好？性价..

WPS备份管理在哪？

露台花园设计有哪些原则？

关于星巴克的“颜值正义”，以及你可能想知道..

```
    sync.lock();
}
//FairSync
final void lock() {
    acquire(1);//调用了AQS的acquire函数,这是关键函数之一
}
```

好,我们接下来就正式开始 **AQS** 相关的源码分析了，**acquire** 函数你可以将其理解为获取一个同一时间只能有一个函数获取的量,这个量就是锁概念的抽象化.我们先分析代码,你慢慢就会明白其中的含义.

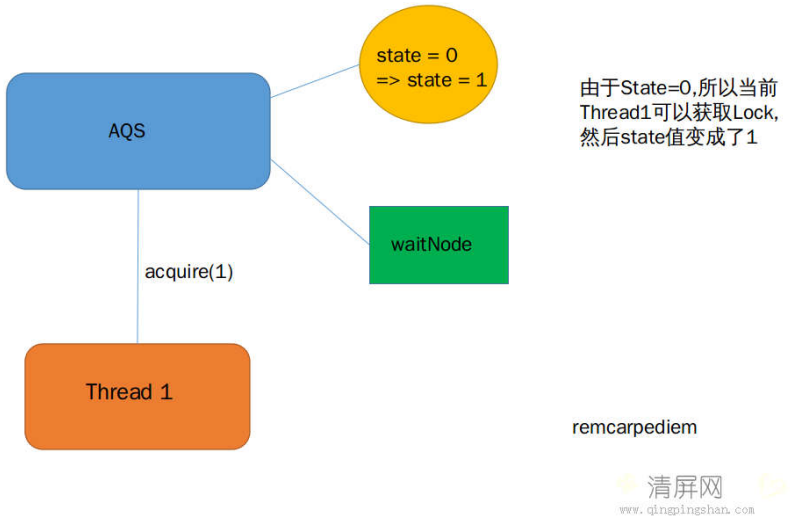
```
public final void acquire(int arg) {
    if (!tryAcquire(arg) && //tryAcquire先尝试获取"锁",//如果成功,直接返回,失败继续执行后续代码
        //addWaiter是给当前线程创建一个节点,并将其加入等待队列
        //acquireQueued是当线程已经加入等待队列之后的行为.
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

tryAcquire ， **addWaiter** 和 **acquireQueued** 都是十分重要的函数,我们接下来依次学习一下这些函数,理解它们的作用.

```
//AQS类中的变量.
private volatile int state;
//这是FairSync的实现,AQS中未实现,子类按照自己的需要实现该类
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    //获取AQS中的state变量,代表抽象概念的锁.
    int c = getState();
    if (c == 0) { //值为0,那么当前独占性变量还未被线程占有
        if (!hasQueuedPredecessors() && //如果当前阻塞队列上没有先来的线程在等待,UnfairSync这里的实现就不一致
            compareAndSetState(0, acquires)) {
            //成功cas,那么代表当前线程获得该变量的所有权,也就是说成功获得锁
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        //如果该线程已经获取了独占性变量的所有权,那么根据重入性
        //原理,将state值进行加1,表示多次lock
        //由于已经获得锁,该段代码只会被一个线程同时执行,所以不需要
        //进行任何并行处理
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //上述情况都不符合,说明获取锁失败
    return false;
}
```

Uber工程师涉嫌窃取谷歌技术：或被禁止参与...
如何选择Python版本2还是3？
苹果手机用户注意 这个漏洞能在WiFi芯片上执...
如何清洗羊绒衫？
用SQL注入穿IE沙箱
营业额达4813亿美元的沃尔玛 市值居然只有亚...

量 state .state的值表示其状态.如果是0,那么当前还没有线程独占此变量,否则就是已经有线程独占了这个变量,也就是代表已经有线程获得了锁.但是这个时候要再进行一次判断,看是否是当前线程自己获得的这个锁,如果是,那么就增加state的值.



这里有几点需要说明一下,首先是 `compareAndSetState` 函数,这是使用CAS操作来设置 `state` 的值,而且state值设置了 `volatile` 修饰符,通过这两点来确保修改state的值不会出现多线程问题.然后是公平锁和非公平锁的区别问题,
在 `UnfairSync` 的 `nonfairTryAcquire` 函数中不会在相同的位置上调用 `hasQueuedPredecessors` 来判断当前是否已经有线程在排队等待获得锁.

如果 `tryAcquire` 返回 `true` ,那么就是获取锁成功,如果返回false,那么就是未获得锁.需要加入阻塞等待队列.我们下面就来看一下 `addWaiter` 的相关操作

等待锁的阻塞队列

将保存当前线程信息的节点加入到等待队列的相关函数中涉及到了无锁队列的相关算法,由于在 `AQS` 中只是将节点添加到队尾,使用到的无锁算法也相对简单.真正的无锁队列的算法我们等到分析 `ConcurrentSkippedListMap` 时在进行讲解.

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    //先使用快速加列法来尝试一下,如果失败,则进行更加完备的加列算法.
    Node pred = tail;//列尾指针
    if (pred != null) {
        node.prev = pred; //步骤1:该节点的前趋指针指向tail
        if (compareAndSetTail(pred, node)){ //步骤二:cas将尾指针指向该节
点
            pred.next = node;//步骤三:如果成果,让旧列尾节点的next指针指向
该节点.
            return node;
        }
    }
    //cas失败,或在pred == null时调用enq
    enq(node);
    return node;
}
```

```
        Node t = tail;
        if (t == null) { //初始化
            if (compareAndSetHead(new Node())) //需要注意的是head是一个哨
兵的作用,并不代表某个要获取锁的线程节点
                tail = head;
        } else {
            //和addWaiter中一致,不过有了外侧的无限循环,不停的尝试,自旋锁
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

通过调用 `addWaiter` 函数，`AQS` 将当前线程加入到了等待队列,但是还没有阻塞当前线程的执行,接下来我们就来分析一下 `acquireQueued` 函数.

等待队列节点的操作

由于进入阻塞状态的操作会降低执行效率,所以，`AQS` 会尽力避免试图获取独占性变量的线程进入阻塞状态.所以,当线程加入等待队列之后，`acquireQueued` 会执行一个for循环,每次都判断当前节点是否应该获得这个变量(在队首了),如果不应该获取或在再次尝试获取失败,那么就调用 `shouldParkAfterFailedAcquire` 判断是否应该进入阻塞状态,如果当前节点之前的节点已经进入阻塞状态了,那么就可以判定当前节点不可能获取到锁,为了防止CPU不停的执行for循环,消耗CPU资源,调用 `parkAndCheckInterrupt` 函数来进入阻塞状态.

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) { //一直执行,知道获取锁,返回.
            final Node p = node.predecessor();
            //node的前驱是head,就说明,node是将要获取锁的下一个节点.
            if (p == head && tryAcquire(arg)) { //所以再次尝试获取独占性
变量
                setHead(node); //如果成果,那么就将自己设置为head
                p.next = null; // help GC
                failed = false;
                return interrupted;//此时,还没有进入阻塞状态,所以直接返回f
alse,表示不需要中断
            }
            //判断是否要进入阻塞状态.如果`shouldParkAfterFailedAcquire`返
回true,表示需要进入阻塞
            //调用parkAndCheckInterrupt,否在表示还可以再次尝试获取锁,继续
进行for循环
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                //调用parkAndCheckInterrupt进行阻塞,然后返回是否为中断状态
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
```

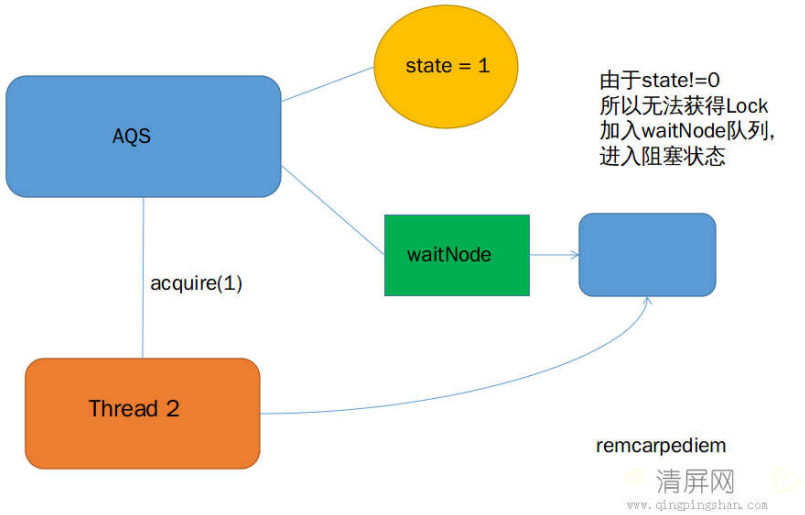
```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node nod
e) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL) //前一个节点在等待独占性变量释放的通知,所以,当前
节点可以阻塞
        return true;
    if (ws > 0) { //前一个节点处于取消获取独占性变量的状态,所以,可以跳过去
        //返回false
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        //将上一个节点的状态设置为signal,返回false,
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this); //将AQS对象自己传入
    return Thread.interrupted();
}
```

阻塞和中断

由上述分析,我们知道了 AQS 通过调用 LockSupport 的 park 方法来执行阻塞当前进程的操作.其实,这里的阻塞就是线程不再执行的含义.通过调用这个函数,线程进入阻塞状态,上述的 lock 操作也就阻塞了.等待中断或在独占性变量被释放.

```
public static void park(Object blocker) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);//设置阻塞对象,用来记录线程被谁阻塞的,用于线程监
控和分析工具来定位
    UNSAFE.park(false, 0L);//让当前线程不再被线程调度,就是当前线程不再执行.
    setBlocker(t, null);
}
```

关于中断的相关知识,我们以后再说,就继续沿着 AQS 的主线,看一下释放独占性变量的相关操作吧.



与 lock 操作类似, unlock 操作调用了 AQS 的 relase 方法,参数和调用 acquire 时一样,都是1.

```
public final boolean release(int arg) {
    if (tryRelease(arg)) { //释放独占性变量,起始就是将status的值减1,因为ac
quire时是加1
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);//唤醒head的后继节点
        return true;
    }
    return false;
}
```

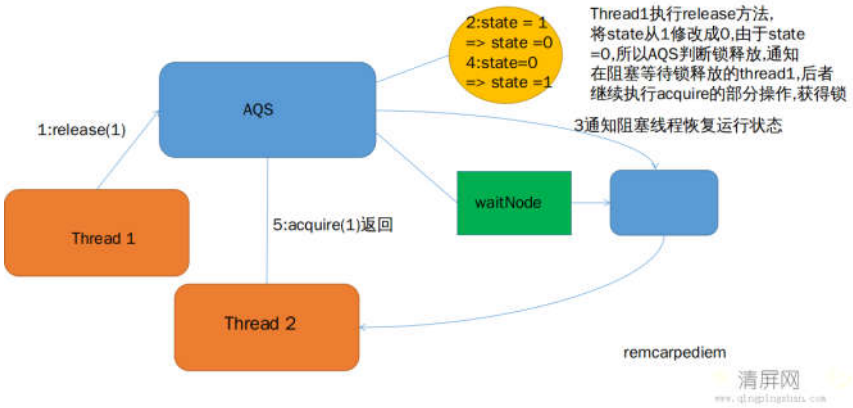
由上述代码可知,release就是先调用 tryRelease 来释放独占性变量,如果成果,那么就看一下是否有等待锁的阻塞线程,如果有,就调用 unparkSuccessor 来唤醒他们.

```
protected final boolean tryRelease(int releases) {
    //由于只有一个线程可以获得独占先变量,所以,所有操作不需要考虑多线程
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) { //如果等于0,那么说明锁应该被释放了,否在表示当前线程有多次l
ock操作.
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

我们可以看到 tryRelease 中的逻辑也体现了可重入锁的概念,只有等到 state 的值为1时,才代表锁真正被释放了.所以独占性变量 state 的值就代表锁的有无.
当 state=0 时,表示锁未被占有,否在表示当前锁已经被占有.

```
private void unparkSuccessor(Node node) {
    .....
     //一般来说,需要唤醒的线程就是head的下一个节点,但是如果它获取锁的操作被取
消,或在节点为null时
     //就直接继续往后遍历,找到第一个未取消的后继节点.
    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

们“acquireQueued”中的无限for循环中的操作,将从会试获取锁.



分享给小伙伴们：

本文标签： Java

相关文章

java反射如何提升性能	04.08	浅析Java中synchronized与static synchron...	04.08
Java自定义标签	04.08	Java中实例化一个抽象类对象的方式	04.08
serialization-protobuf	04.08	Java Level2学习的八大名著	04.08
利用java的多态性对hibernate方法的简单封...	04.08	贪心算法：最小代价生成树Java版详解	04.08
《Java NIO文档》非阻塞式服务器	04.08	Java七武器系列长生剑：Java虚拟机的显微镜..	04.08

发表评论

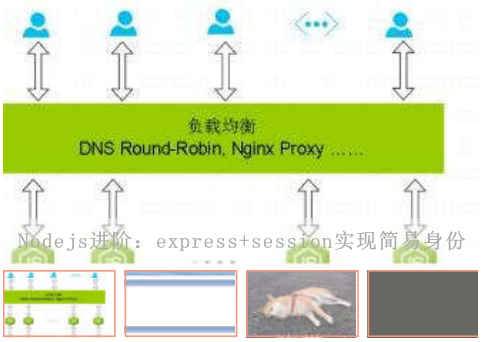
愿您的每句评论，都能给大家的生活添色彩，带来共鸣，带来思索，带来快乐。

登录

来说两句吧...

还没有评论，快来抢沙发吧！

畅言



热评话题

- 360Vulcan：NSA武器库之Etern...
- XP系统如何显示隐藏的文件夹？...
- 智联招聘验证码校验请求：MmE...
- Android7.0调用系统相机拍照、...
- To use the Pentestbox to creat...
- 在RHEL、CentOS及Fedora上安...
- Cross-site Request Forgery 簡...