dreamcatcher-cx

why is more important than what.

Java并发包基石-AQS详解

目录

Java并发包（JUC）中提供了很多并发工具，这其中，很多我们耳熟能详的并发工具，譬如ReentrangLock、Semaphore，它们的实现都用到了一个共同的基类--**AbstractQueuedSynchronizer**,简称AQS。AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

本章我们就一起探究下这个神奇的东东，并对其实现原理进行剖析理解

基本实现原理

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

状态信息通过procted类型的**getState**，**setState**，**compareAndSetState**进行操作

AQS支持两种同步方式：

1.独占式

2.共享式

这样方便使用者实现不同类型的同步组件，独占式如ReentrantLock，共享式如Semaphore，CountDownLatch，组合式的如ReentrantReadWriteLock。总之，AQS为使用提供了底层支撑，如何组装实现，使用者可以自由发挥。

同步器的设计是基于**模板方法模式**的，一般的使用方式是这样：

1.使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）

2.将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这其实是**模板方法模式**的一个很经典的应用。

我们来看看AQS定义的这些可重写的方法：

protected boolean tryAcquire(int arg)：独占式获取同步状态，试着获取，成功返回true，反之为false

protected boolean tryRelease(int arg)：独占式释放同步状态，等待中的其他线程此时将有机会获取到同步状态；

protected int tryAcquireShared(int arg)：共享式获取同步状态，返回值大于等于0，代表获取成功；反之获取失败；

protected boolean tryReleaseShared(int arg)：共享式释放同步状态，成功为true，失败为false

protected boolean isHeldExclusively()：是否在独占模式下被线程占用。

关于AQS的使用，我们来简单总结一下：

如何使用

首先，我们需要去继承AbstractQueuedSynchronizer这个类，然后我们根据我们的需求去重写相应的方法，比如要实现一个独占锁，那就去重写tryAcquire，tryRelease方法，要实现共享锁，就去重写tryAcquireShared，tryReleaseShared；最后，在我们的组件中调用AQS中的模板方法就可以了，而这些模板方法是会调用到我们之前重写的那些方法的。也就是说，我们只需要很小的工作量就可以实现自己的同步组件，重写的那些方法，仅仅是一些简单的对于共享资源state的获取和释放操作，至于像是获取资源失败，线程需要阻塞之类的操作，自然是AQS帮我们完成了。

设计思想

对于使用者来讲，我们无需关心获取资源失败，线程排队，线程阻塞/唤醒等一系列复杂的实现，这些都在AQS中为我们处理好了。我们只需要负责好自己的那个环节就好，也就是获取/释放共享资源state的姿势T_T。很经典的模板方法设计模式的应用，AQS为我们定义好顶级逻辑的骨架，并提取出公用的线程入队列/出队列，阻塞/唤醒等一系列复杂逻辑的实现，将部分简单的可由使用者决定的操作逻辑延迟到子类中去实现即可。

自定义同步器

同步器代码实现

上面大概讲了一些关于AQS如何使用的理论性的东西，接下来，我们就来看下实际如何使用，直接采用JDK官方文档中的小例子来说明问题

```
1 package juc;
2
3 import java.util.concurrent.locks.AbstractQueuedSynchronizer;
4
5 /**
6  * Created by chengxiao on 2017/3/28.
7  */
8 public class Mutex implements java.io.Serializable {
9     //静态内部类，继承AQS
10    private static class Sync extends AbstractQueuedSynchronizer {
11        //是否处于占用状态
12        protected boolean isHeldExclusively() {
13            return getState() == 1;
14        }
15        //当状态为0的时候获取锁，CAS操作成功，则state状态为1，
16        public boolean tryAcquire(int acquires) {
17            if (compareAndSetState(0, 1)) {
18                setExclusiveOwnerThread(Thread.currentThread());
19                return true;
20            }
21            return false;
22        }
23        //释放锁，将同步状态置为0
24        protected boolean tryRelease(int releases) {
25            if (getState() == 0) throw new IllegalMonitorStateException();
26            setExclusiveOwnerThread(null);
27            setState(0);
28            return true;
29        }
30    }
31        //同步对象完成一系列复杂的操作，我们仅需指向它即可
32        private final Sync sync = new Sync();
33        //加锁操作，代理到acquire（模板方法）上就行，acquire会调用我们重写的tryAcquire方法
34        public void lock() {
```

Oracle(4)
并发编程(8)
分布式系统
数据结构(2)
算法(5)

随笔档案(20)
2017年7月 (2)
2017年6月 (1)
2017年5月 (2)
2017年4月 (1)
2017年3月 (1)
2017年2月 (1)
2017年1月 (1)
2016年12月 (3)
2016年11月 (4)
2016年10月 (2)
2016年9月 (2)

积分与排名
积分 - 45393
排名 - 8329

最新评论
1. Re:HashMap实现原
@云淡wy谢谢指正，改
--dr
2. Re:HashMap实现原
如果定位到的数组包含链
作，其时间复杂度依然为
的Entry会插入链表头部

```
35              sync.acquire(1);
36          }
37          public boolean tryLock() {
38              return sync.tryAcquire(1);
39          }
40          //释放锁，代理到release（模板方法）上就行，release会调用我们重写的tryRelease方法。
41          public void unlock() {
42              sync.release(1);
43          }
44          public boolean isLocked() {
45              return sync.isHeldExclusively();
46          }
47  }
```

同步器代码测试

测试下这个自定义的同步器，我们使用之前文章中做过的并发环境下a++的例子来说明问题（a++的原子性其实最好使用原子类AtomicInteger来解决，此处用Mutex有点大炮打蚊子的意味，好在能说明问题就好）

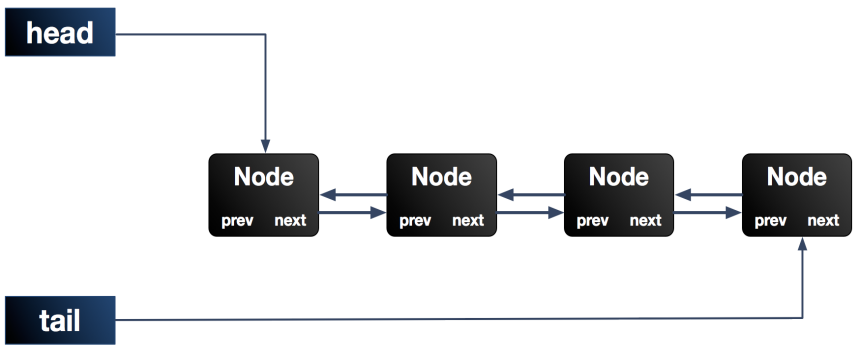
```
TestMutex
```

测试结果：

```
加锁前，a=279204
加锁后，a=300000
```

源码分析

我们先来简单描述下AQS的基本实现，前面我们提到过，AQS维护一个共享资源state，通过内置的FIFO来完成获取资源线程的排队工作。（这个内置的同步队列称为"CLH"队列）。该队列由一个一个的Node结点组成，每个Node结点维护一个prev引用和next引用，分别指向自己的前驱和后继结点。AQS维护两个指针，分别指向队列头部head和尾部tail。



其实就是个**双端双向链表**。

当线程获取资源失败（比如tryAcquire时试图设置state状态失败），会被构造成一个结点加入CLH队列中，同时当前线程会被阻塞在队列中（通过LockSupport.park实现，其实是等待态）。当持有同步状态的线程释放同步状态时，会唤醒后继结点，然后此结点线程继续加入到对同步状态的争夺中。

Node结点

Node结点是AbstractQueuedSynchronizer中的一个静态内部类，我们捡Node的几个重要属性来说一下

```
1 static final class Node {
2         /** waitStatus值，表示线程已被取消（等待超时或者被中断）*/
3         static final int CANCELLED =  1;
4         /** waitStatus值，表示后继线程需要被唤醒（unpaking）*/
5         static final int SIGNAL    = -1;
6         /**waitStatus值，表示结点线程等待在condition上，当被signal后，会从等待队列转移到同步到队列中 */
7         /** waitStatus value to indicate thread is waiting on condition */
8         static final int CONDITION = -2;
9        /** waitStatus值，表示下一次共享式同步状态会被无条件地传播下去
10        static final int PROPAGATE = -3;
11        /** 等待状态，初始为0 */
```

```
        volatile int waitStatus;
        /**当前结点的前驱结点 */
        volatile Node prev;
        /** 当前结点的后继结点 */
        volatile Node next;
        /** 与当前结点关联的排队中的线程 */
        volatile Thread thread;
        /** ..... */
    }
```

独占式

获取同步状态--acquire()

来看看acquire方法，lock方法一般会直接代理到acquire上

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

我们来简单理一下代码逻辑：

a.首先，调用使用者重写的tryAcquire方法，若返回true，意味着获取同步状态成功，后面的逻辑不再执行；若返回false，也就是获取同步状态失败，进入b步骤；

b.此时，获取同步状态失败，构造独占式同步结点，通过addWatiter将此结点添加到同步队列的尾部（此时可能会有多个线程结点试图加入同步队列尾部，需要以线程安全的方 式添加）；

c.该结点以在队列中尝试获取同步状态，若获取不到，则阻塞结点线程，直到被前驱结点唤醒或者被中断。

addWaiter

为获取同步状态失败的线程，构造成一个Node结点，添加到同步队列尾部

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);//构造结点
        //指向尾结点tail
        Node pred = tail;
        //如果尾结点不为空，CAS快速尝试在尾部添加，若CAS设置成功，返回；否则，enq。
        if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        enq(node);
        return node;
    }
```

先cas快速设置，若失败，进入enq方法

将结点添加到同步队列尾部这个操作，同时可能会有多个线程尝试添加到尾部，是非线程安全的操作。

以上代码可以看出，使用了compareAndSetTail这个cas操作保证安全添加尾结点。

enq方法

```
private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            if (t == null) { //如果队列为空，创建结点，同时被head和tail引用
                if (compareAndSetHead(new Node()))
                    tail = head;
```

推荐排行榜

1. HashMap实现原理及
2. 图解排序算法(三)之堆
3. 图解排序算法(四)之归
4. 图解排序算法(一)之三择，冒泡，直接插入)(9
5. Oracle体系结构详解

```
            } else {
                node.prev = t;
                if (compareAndSetTail(t, node)) {//cas设置尾结点，不成功就一直重试
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

enq内部是个死循环，通过CAS设置尾结点，不成功就一直重试。很经典的CAS自旋的用法，我们在之前关于原子类的源码分析中也提到过。这是一种**乐观的并发策略**。

最后，看下acquireQueued方法

acquireQueued

```
final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {//死循环
                final Node p = node.predecessor();//找到当前结点的前驱结点
                if (p == head && tryAcquire(arg)) {//如果前驱结点是头结点，才tryAcquire，其他结点是没有机会tryAcquire的。
                    setHead(node);//获取同步状态成功，将当前结点设置为头结点。
                    p.next = null; // 方便GC
                    failed = false;
                    return interrupted;
                }
                // 如果没有获取到同步状态，通过shouldParkAfterFailedAcquire判断是否应该阻塞，parkAndCheckInterrupt用来阻塞线
程
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

acquireQueued内部也是一个死循环，只有前驱结点是头结点的结点，也就是老二结点，才有机会去tryAcquire；若tryAcquire成功，表示获取同步状态成功，将此结点设置为头结点；若是非老二结点，或者tryAcquire失败，则进入shouldParkAfterFailedAcquire去判断判断当前线程是否应该阻塞，若可以，调用parkAndCheckInterrupt阻塞当前线程，直到被中断或者被前驱结点唤醒。若还不能休息，继续循环。

shouldParkAfterFailedAcquire

shouldParkAfterFailedAcquire用来判断当前结点线程是否能休息

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        //获取前驱结点的wait值
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)//若前驱结点的状态是SIGNAL，意味着当前结点可以被安全地park
            return true;
        if (ws > 0) {
        // ws>0，只有CANCEL状态ws才大于0。若前驱结点处于CANCEL状态，也就是此结点线程已经无效，从后往前遍历，找到一个非CANCEL状态的结
点，将自己设置为它的后继结点
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
```

```
            // 若前驱结点为其他状态，将其设置为SIGNAL状态
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

若shouldParkAfterFailedAcquire返回true，也就是当前结点的前驱结点为SIGNAL状态，则意味着当前结点可以放心休息，进入parking状态了。parkAncCheckInterrupt阻塞线程并处理中断。

```
private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);//使用LockSupport使线程进入阻塞状态
        return Thread.interrupted();// 线程是否被中断过
    }
```

至此，关于acquire的方法源码已经分析完毕，我们来简单总结下

a.首先tryAcquire获取同步状态，成功则直接返回；否则，进入下一环节；

b.线程获取同步状态失败，就构造一个结点，加入同步队列中，这个过程要保证线程安全；

c.加入队列中的结点线程进入自旋状态，若是老二结点（即前驱结点为头结点），才有机会尝试去获取同步状态；否则，当其前驱结点的状态为SIGNAL，线程便可安心休息，进入阻塞状态，直到被中断或者被前驱结点唤醒。

释放同步状态--release()

当前线程执行完自己的逻辑之后，需要释放同步状态，来看看release方法的逻辑

```
public final boolean release(int arg) {
        if (tryRelease(arg)) {//调用使用者重写的tryRelease方法，若成功，唤醒其后继结点，失败则返回false
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);//唤醒后继结点
            return true;
        }
        return false;
    }
```

unparkSuccessor：唤醒后继结点

```
1 private void unparkSuccessor(Node node) {
2         //获取wait状态
3         int ws = node.waitStatus;
4         if (ws < 0)
5             compareAndSetWaitStatus(node, ws, 0);// 将等待状态waitStatus设置为初始值0
6         Node s = node.next;//后继结点
7         if (s == null || s.waitStatus > 0) {//若后继结点为空，或状态为CANCEL（已失效），则从后尾部往前遍历找到一个处于正常阻塞状态的结点          进行唤醒
8             s = null;
9             for (Node t = tail; t != null && t != node; t = t.prev)
10                if (t.waitStatus <= 0)
11                    s = t;
12        }
13        if (s != null)
14            LockSupport.unpark(s.thread);//使用LockSupprot唤醒结点对应的线程
15    }
```

release的同步状态相对简单，需要找到头结点的后继结点进行唤醒，若后继结点为空或处于CANCEL状态，从后向前遍历找寻一个正常的结点，唤醒其对应线程。

共享式

共享式：共享式地获取同步状态。对于独占式同步组件来讲，同一时刻只有一个线程能获取到同步状态，其他线程都得去排队等待，其待重写的尝试获取同步状态的方法tryAcquire返回值为boolean，这很容易理解；对于共享式同步组件来讲，同一时刻

可以有多个线程同时获取到同步状态，这也是“共享”的意义所在。其待重写的尝试获取同步状态的方法tryAcquireShared返回值为int。

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

1.当返回值大于0时，表示获取同步状态成功，同时还有剩余同步状态可供其他线程获取；

2.当返回值等于0时，表示获取同步状态成功，但没有可用同步状态了；

3.当返回值小于0时，表示获取同步状态失败。

获取同步状态--acquireShared

```
public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0)//返回值小于0，获取同步状态失败，排队去；获取同步状态成功，直接返回去干自己的事儿。
            doAcquireShared(arg);
    }
```

doAcquireShared

```
 1  private void doAcquireShared(int arg) {
 2          final Node node = addWaiter(Node.SHARED);//构造一个共享结点，添加到同步队列尾部。若队列初始为空，先添加一个无意义的傀儡结点，再将新节点添加到队列尾部。
 3          boolean failed = true;//是否获取成功
 4          try {
 5              boolean interrupted = false;//线程parking过程中是否被中断过
 6              for (;;) {//死循环
 7                  final Node p = node.predecessor();//找到前驱结点
 8                  if (p == head) {//头结点持有同步状态，只有前驱是头结点，才有机会尝试获取同步状态
 9                      int r = tryAcquireShared(arg);//尝试获取同步装填
10                      if (r >= 0) {//r>=0,获取成功
11                          setHeadAndPropagate(node, r);//获取成功就将当前结点设置为头结点，若还有可用资源，传播下去，也就是继续唤醒后继结点
12                          p.next = null; // 方便GC
13                          if (interrupted)
14                              selfInterrupt();
15                          failed = false;
16                          return;
17                      }
18                  }
19                  if (shouldParkAfterFailedAcquire(p, node) &&//是否能安心进入parking状态
20                      parkAndCheckInterrupt())//阻塞线程
21                      interrupted = true;
22              }
23          } finally {
24              if (failed)
25                  cancelAcquire(node);
26          }
27      }
```

大体逻辑与独占式的acquireQueued差距不大，只不过由于是共享式，会有多个线程同时获取到线程，也可能同时释放线程，空出很多同步状态，所以当排队中的老二获取到同步状态，如果还有可用资源，会继续传播下去。

setHeadAndPropagate

```
private void setHeadAndPropagate(Node node, int propagate) {
        Node h = head; // Record old head for check below
        setHead(node);
        if (propagate > 0 || h == null || h.waitStatus < 0) {
            Node s = node.next;
            if (s == null || s.isShared())
                doReleaseShared();
```

```
        }
    }
```

释放同步状态--releaseShared

```
public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();//释放同步状态
            return true;
        }
        return false;
    }
```

doReleaseShared

```
private void doReleaseShared() {
        for (;;) {//死循环，共享模式，持有同步状态的线程可能有多个，采用循环CAS保证线程安全
            Node h = head;
            if (h != null && h != tail) {
                int ws = h.waitStatus;
                if (ws == Node.SIGNAL) {
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                        continue;
                    unparkSuccessor(h);//唤醒后继结点
                }
                else if (ws == 0 &&
                         !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                    continue;
            }
            if (h == head)
                break;
        }
    }
```

代码逻辑比较容易理解，需要注意的是，共享模式，释放同步状态也是多线程的，此处采用了CAS自旋来保证。

总结

关于AQS的介绍及源码分析到此为止了。

AQS是JUC中很多同步组件的构建基础，简单来讲，它内部实现主要是状态变量state和一个FIFO队列来完成，同步队列的头结点是当前获取到同步状态的结点，获取同步状态state失败的线程，会被构造成一个结点（或共享式或独占式）加入到同步队列尾部（采用自旋CAS来保证此操作的线程安全），随后线程会阻塞；释放时唤醒头结点的后继结点，使其加入对同步状态的争夺中。

AQS为我们定义好了顶层的处理实现逻辑，我们在使用AQS构建符合我们需求的同步组件时，只需重写tryAcquire，tryAcquireShared，tryRelease，tryReleaseShared几个方法，来决定同步状态的释放和获取即可，至于背后复杂的线程排队，线程阻塞/唤醒，如何保证线程安全，都由AQS为我们完成了，这也是非常典型的模板方法的应用。AQS定义好顶级逻辑的骨架，并提取出公用的线程入队列/出队列，阻塞/唤醒等一系列复杂逻辑的实现，将部分简单的可由使用者决定的操作逻辑延迟到子类中去实现。

分类： 并发编程