AQS 框架分享

北京 朝阳区 来广营中街甲一号 26-27 号楼

贺辰枫

AQS

基本概念 -Java 对象头

锁存在 Java 对象头里。如果对象是数组类型，则虚拟机用 3 个 Word （字宽）存储对象头，如果对象是非数组类型，则用 2 字宽存储对象头。在 32 位虚拟机中，一字宽等于四字节，即 32bit

长度	内容	说明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/64bit	Array length	数组的长度（如果当前对象是数组）

基本概念 -Java 对象头

Java 对象头里的 Mark Word 里默认存储对象的 HashCode ，分代年龄和锁标记位。 32 位 JVM 的 Mark Word 的默认存储结构如下

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象hashcode、对象分代年龄				01
轻量级锁	指向锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

基本概念 - 锁的分类

锁的分类（从虚拟机实现底层的角度来进行分类）

- 重量级锁
- 轻量级锁
- 偏向锁

基本概念 - 重量级锁

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础

1. 普通同步方法，锁是当前实例对象
2. 静态同步方法，锁是当前类的 class 对象
3. 同步方法块，锁是括号里面的对象

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性（我认为应该是同步开始从主内存中读取，同步结束刷新到主内存中），并且在正常退出或者抛出异常时自动释放锁

- 同步方法： synchronized 方法则会被翻译成普通的方法调用和返回指令如 :invokevirtual 、 areturn 指令，在 VM 字节码层面并没有任何特别的指令来实现被 synchronized 修饰的方法，而是在 Class 文件的方法表中将该方法的 access_flags 字段中的 synchronized 标志位置 1 ，表示该方法是同步方法并使用调用该方法的对象或该方法所属的 Class 在 JVM 的内部对象表示 Klass 做为锁对象

synchronized 是重量级锁，重量级锁通过对象内部的监视器 (monitor) 实现，其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高

基本概念 - 轻量级锁

引入轻量级锁主要目的是：减少传统的重量级锁使用操作系统互斥量产生的性能消耗

基于一种假设：基本没有多线程竞争

效果：将基于操作系统互斥量来进行的重量级加锁解锁操作转换为基于轻量级的 CAS 操作的加锁解锁操作

基本概念 - 轻量级锁

获取锁

1. 判断当前对象是否处于无锁状态 (锁标志位 01 ，偏向锁标志位 0)
- 若是，则 JVM 首先将在当前线程的栈帧中建立一个名为锁记录 (Lock Record) 的空间，用于存储锁对象目前的 Mark Word 的拷贝 (官方把这份拷贝加了一个 Displaced 前缀，即 Displaced Mark Word)
- 否则执行步骤 (3)

2. JVM 利用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指针
- 如果成功表示竞争到锁，则将锁标志位变成 00(表示此对象处于轻量级锁状态)，执行同步操作
- 如果失败则执行步骤 (3)

3. 判断当前对象的 Mark Word 是否指向当前线程的栈帧
- 如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块
- 否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁，锁标志位变成 10 ，后面等待的线程将会进入阻塞状态

基本概念 - 轻量级锁

释放锁

1. 取出在获取轻量级锁保存在 Displaced Mark Word 中的数据
2. 用 CAS 操作将取出的数据替换当前对象的 Mark Word 中
   - 如果成功，则说明释放锁成功
   - 否则执行 (3)
3. 如果 CAS 操作替换失败，说明有其他线程尝试获取该锁，则需要在释放锁的同时唤醒被挂起的线程

基本概念 - 偏向锁

引入偏向锁主要目的：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径 (CAS 原子指令)

基于一种假设：没有多线程竞争

效果：去除轻量级加锁解锁的 CAS 操作开销

基本概念 - 偏向锁

获取锁

1. 检测 Mark Word 是否为可偏向状态 (锁标识位 01 ，偏向锁标志位 1)
2. 若为可偏向状态，则测试线程 ID 是否为当前线程 ID
- 如果是，则执行步骤 (5)
- 否则执行步骤 (3)
3. 如果线程 ID 不为当前线程 ID ，则通过 CAS 操作竞争锁
- 竞争成功，将 Mark Word 的线程 ID 替换为当前线程 ID ，执行步骤 (5)
- 否则执行步骤 (4)
4. 通过 CAS 竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块
5. 执行同步代码块

基本概念 - 偏向锁

释放锁：偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要等待全局安全点 (这个时间点是上没有正在执行的代码) 。其步骤如下

1. 暂停拥有偏向锁的线程，判断锁对象石是否还处于被锁定状态
2. 撤销偏向锁，恢复到无锁状态 (01) 或者轻量级锁的状态

自旋锁

引入自旋锁主要目的是：避免线程阻塞和唤醒的开销（线程的阻塞和唤醒需要 CPU 从用户态转为核心态，频繁的阻塞和唤醒对 CPU 来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力）

基于一种假设：”阻塞”的时间很短

效果：相比于阻塞和唤醒具有更快的响应速度

局限

- 虽然避免了线程状态切换的开销，但是会占用 CPU 时间，当”阻塞”时间超过一定限度时，自旋锁反而会带来负增益
- 无法响应线程中断
- 无法满足公平性，即先进入睡眠的线程具有更高的优先级

自旋锁 -Demo


```
public class SpinLockDemo {

    private AtomicInteger state = new AtomicInteger();

    public void lock() {
        for (; ; ) {
            if (state.compareAndSet(0, 1)) {
         break;
            }
        }
    }

    public void unlock() {
        if (!state.compareAndSet(1, 0)) {
            throw new RuntimeException();
        }
    }
}
```

Ticket 锁

可以看成是自旋锁的一种优化

改进：满足公平性， FIFO 原则

局限：

- 占用 CPU 时间
- 仍然无法响应中断
- 在共享资源上自旋，有较高的同步代价

Ticket 锁 -Demo


```
public class TicketLockDemo {

    private AtomicInteger serviceNum = new AtomicInteger();// 当前服务号

    private AtomicInteger ticketNum = new AtomicInteger();// 排队号

    public int lock() {
        // 排队前拿个号
        int myTicketNum = ticketNum.getAndIncrement();

        while (serviceNum.get() != myTicketNum) {

        }

        return myTicketNum;
    }

    public void unlock(int myTicket) {
        int next = myTicket + 1;
        if (!serviceNum.compareAndSet(myTicket, next)) {
            throw new RuntimeException();
        }
    }
}
```

CLH 队列锁

可以看成是 Ticket 锁的一种优化

改进：在本地变量上自旋

局限：

- 占用 CPU 时间
- 仍然无法响应中断
- 一个节点仅能在加锁期间持有其前继节点的状态对象，

CLH 队列锁 -Demo


```
public class CLHLockDemo {
    AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());
    ThreadLocal<QNode> currNode;// 在本地变量上自旋

    public CLHLockDemo() {
        tail = new AtomicReference<QNode>(new QNode());
        currNode = new ThreadLocal<QNode>() {
            protected QNode initialValue() {
                return new QNode();
            }
        };
    }
    public void lock() {
        QNode curr = this.currNode.get();
        curr.locked = true;
        // 将当前节点通过 CAS 操作加到队列尾，返回原先的队列尾，作为它的前继节点
        QNode prev = tail.getAndSet(curr);
        while (prev.locked) {
            // 在本地变量 prev 上自旋
        }
    }
    public void unlock() {
        QNode qnode = currNode.get();
        qnode.locked = false;
    }
}
```

AQS 框架

并发大师 Doug Lea

抽象的队列式的同步器，AQS 定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的
ReentrantLock/Semaphore/CountDownLatch

AQS 是 CLH 队列锁的变体

扩展部分包括

- 真正维护一个队列（双向链表），提供可靠的 prev 指针以及非可靠的 next 指针
- 设计节点的状态，用于控制阻塞，可以实现 timesout 以及 cancellation

AQS 框架 - 概览


```
static final class Node {
    static final Node SHARED = new Node();
    static final Node EXCLUSIVE = null;
    static final int CANCELLED =  1;
    static final int SIGNAL    = -1;
    static final int CONDITION = -2;
    static final int PROPAGATE = -3;
    volatile int waitStatus;
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;
    …
}
```

AQS 框架 - 概览


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

Lock 接口

AQS 框架 - 源码详解


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

交给子类实现同步语义

AQS 框架 - 源码详解


```
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;
}
```

将节点通过 CAS 操作插入到队列尾部
注意 next 字段与 pred 字段的可靠性

AQS 框架 - 源码详解


```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

AQS 框架 - 源码详解


```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

若当前节点是头结点的后继，那么尝试获取资源
否则找到合适的位置，并睡眠

AQS 框架 - 源码详解


```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        return true;
    if (ws > 0) {
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

1、如果前继状态为 SIGNAL ，就去睡眠，这里会有个小问题
2、往前找到第一个有效节点
3、将有效节点设置为 SIGNAL ，在此过程中会加入适当的自旋

AQS 框架 - 源码详解


```
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

1、睡眠
2、返回是否被中断，如果是中断了，必须将中断状态传到外部，便于恢复中断现场

AQS 框架 - 源码详解


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

unlock 接口

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

交给子类实现同步语义

AQS 框架 - 源码详解


```
private void unparkSuccessor(Node node) {

    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    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);
}
```

尝试通过 next 定位后继，若失败，则从后往前找到后继

AQS 框架 - 源码详解

suspend() 和 resume() 方法：

- 这两个方法隶属于 Thread ，是 Thread 的非静态方法
- 两个方法配套使用， suspend() 使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态
- 典型地， suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复
- suspend 不能响应中断
- 但 suspend() 方法很容易引起死锁问题，已经不推荐使用了

λ 如果一个目标线程 t1 对某一关键系统资源进行了加锁操作，然后在该加锁区块执行 t1.suspend() ，那么除非执行 t1.resume() ，否则其它线程都将无法访问该系统资源

λ 如果另外一个线程 t2 想要占用资源，那么 t2 必须调用 t1.resume() ，如果 t2 调用 t1.resume() 之前需要获取该系统资源，那么造成死锁

AQS 框架 - 源码详解

wait() 和 notify() 方法：

- 这两个方法隶属于 Object ，是 Object 的非静态方法
- 两个方法配套使用， wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用
- **必须要在 synchronized 块内使用 (保证调用这两个方法时，获取该对象的锁)**，但是不用编译器也不会阻止，运行时可能抛出 IllegalMonitorStateException 异常
- wait 可以响应中断 (以抛出 InterruptedException 的方式，以类似方式相应中断的还有 join 方法与 sleep 方法)

AQS 框架 - 源码详解

LockSupport 类是 Java6(JSR166-JUC) 引入的一个类，提供了基本的线程同步原语。 LockSupport 实际上是调用了 Unsafe 类里的函数，归结到 Unsafe 里，只有两个函数

public native void unpark(Thread jthread);

public native void park(boolean isAbsolute, long time);

2 、 unpark 函数为线程提供 " 许可 (permit)" ，线程调用 park 函数则等待 " 许可 " 这个有点像信号量，但是这个 " 许可 " 是不能叠加的， " 许可 " 是一次性的

比如线程 B 连续调用了三次 unpark 函数，当线程 A 调用 park 函数就使用掉这个 " 许可 " ，如果线程 A 再次调用 park ，则进入等待状态

3 、 park 和 unpark 的灵活之处： unpark 函数可以先于 park 调用，这个正是它们的灵活之处

4 、 park 可以响应中断，但不是通过抛出 InterruptedException 的方式来中断，中断后中断标志位是 true

Unsafe 的使用有严格的限制，我们只能通过 LockSupport 来进行阻塞

AQS 框架 - 源码详解


```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

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

AQS 框架 - 源码详解


```
private void doAcquireShared(int arg) {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                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);
    }
}
```

AQS 框架 - 源码详解


```
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();
    }
}
```

AQS 框架 - 源码详解


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

AQS 框架 - 源码详解


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

AQS 框架 - 源码详解


```
private void doReleaseShared() {

    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;            // loop to recheck cases
                unparkSuccessor(h);
            }
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        if (h == head)                   // loop if head changed
            break;
    }
}
```

确保同一时刻只有一个线程唤醒 head 的后继
PROPAGATE: 代表后续节点会将唤醒动作传递下去 (一个短暂的状态，表明 unpark 链正在往后执行)

AQS 框架 - 源码详解


```
public class MyLockDemo {
    private final Sync sync=new Sync();

    private static final class Sync extends AbstractQueuedSynchronizer{
        private final AtomicInteger resources=new AtomicInteger();
        @Override
        protected boolean tryAcquire(int arg) {
            return resources.compareAndSet(0,arg);
        }
        @Override
        protected boolean tryRelease(int arg) {
            resources.set(0);
            return true;
        }
    }
    public void lock(){
        sync.acquire(1);
    }
    public void unlock(){
        sync.release(1);
    }
}
```

END&
THANKS