第22讲 | AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

2018-06-26 杨晓峰

第22讲 | AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

杨晓峰

00:26 / 11:03

在今天这一讲中，我来分析一下并发包内部的组成，一起来看看各种同步结构、线程池等，是基于什么原理来设计和实现的。

今天我要问你的问题是，AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

典型回答

AtomicIntger是对int类型的一个封装，提供原子性的访问和更新操作，其原子性操作的实现是基于CAS（compare-and-swap）技术。

所谓CAS，表征的是一些列操作的集合，获取当前数值，进行一些运算，利用CAS指令试图进行更新。如果当前数值未变，代表没有其他线程进行并发修改，则成功更新。否则，可能出现不同的选择，要么进行重试，要么就返回一个成功或者失败的结果。

从AtomicInteger的内部属性可以看出，它依赖于Unsafe提供的一些底层能力，进行底层操作；以volatile的value字段，记录数值，以保证可见性。

```
private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();
private static final long VALUE = U.objectFieldOffset(AtomicInteger.class, "value");
private volatile int value;
```

具体的原子操作细节，可以参考任意一个原子更新方法，比如下面的getAndIncrement。

Unsafe会利用value字段的内存地址偏移，直接完成操作。

```
public final int getAndIncrement() {
    return U.getAndAddInt(this, VALUE, 1);
}
```

因为getAndIncrement需要返归数值，所以需要添加失败重试逻辑。

```
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

而类似compareAndSet这种返回boolean类型的函数，因为其返回值表现的就是成功与否，所以不需要重试。

```
public final boolean compareAndSet(int expectedValue, int newValue)
```

CAS是Java并发中所谓lock-free机制的基础。

考点分析

Java　　CAS　　Java

今天的问题有点偏向于　　并发机制的底层了，虽然我们在开发中未必会涉及　　的实现层面，但是理解其机制，掌握如何在　　中运用该技术，还是十分有必要的，尤其是这也是个并发编程的面试热点。

有的同学反馈面试官会问CAS更加底层是如何实现的，这依赖于CPU提供的特定指令，具体根据体系结构的不同还存在着明显区别。比如，x86 CPU提供cmpxchg指令；而在精简指令集的体系架构中，则通常是靠一对儿指令（如"load and reserve"和"store conditional"）实现的，在大多数处理器上CAS都是个非常轻量级的操作，这也是其优势所在。

大部分情况下，掌握到这个程度也就够用了，我认为没有必要让每个Java工程师都去了解到指令级别，我们进行抽象、分工就是为了让不同层面的开发者在开发中，可以尽量屏蔽不相关的细节。

如果我作为面试官，很有可能深入考察这些方向：

- 在什么场景下，可以采用CAS技术，调用Unsafe毕竟不是大多数场景的最好选择，有没有更加推荐的方式呢？毕竟我们掌握一个技术，cool不是目的，更不是为了应付面试，我们还是希望能在实际产品中有价值。
- 对ReentrantLock、CyclicBarrier等并发结构底层的实现技术的理解。

知识扩展

关于CAS的使用，你可以设想这样一个场景：在数据库产品中，为保证索引的一致性，一个常见的选择是，保证只有一个线程能够排他性地修改一个索引分区，如何在数据库抽象层面实现呢？

可以考虑为索引分区对象添加一个逻辑上的锁，例如，以当前独占的线程ID作为锁的数值，然后通过原子操作设置lock数值，来实现加锁和释放锁，伪代码如下：

```
public class AtomicBTreePartition {
private volatile long lock;
public void acquireLock(){}
public void releaseeLock(){}
}
```

那么在Java代码中，我们怎么实现锁操作呢？Unsafe似乎不是个好的选择，例如，我就注意到类似Cassandra等产品，因为Java 9中移除了Unsafe.moniterEnter()/moniterExit()，导致无法平滑升级到新的JDK版本。目前Java提供了两种公共API，可以实现这种CAS操作，比如使用java.util.concurrent.atomic.AtomicLongFieldUpdater，它是基于反射机制创建，我们需要保证类型和字段名称正确。

```
private static final AtomicLongFieldUpdater<AtomicBTreePartition> lockFieldUpdater =
        AtomicLongFieldUpdater.newUpdater(AtomicBTreePartition.class, "lock");

private void acquireLock(){
    long t = Thread.currentThread().getId();
    while (!lockFieldUpdater.compareAndSet(this, 0L, t)){
        // 等待一会儿，数据库操作可能比较慢
         …
    }
}
```

Atomic包提供了最常用的原子性数据类型，甚至是引用、数组等相关原子类型和更新操作工具，是很多线程安全程序的首选。

我在专栏第七讲中曾介绍使用原子数据类型和Atomic*FieldUpdater，创建更加紧凑的计数器实现，以替代AtomicLong。优化永远是针对特定需求、特定目的，我这里的侧重点是介绍可能的思路，具体还是要看需求。如果仅仅创建一两个对象，其实完全没有必要进行前面的优化，但是如果对象成千上万或者更多，就要考虑紧凑性的影响了。而atomic包提供的LongAdder，在高度竞争环境下，可能就是比AtomicLong更佳的选择，尽管它的本质是空间换时间。

回归正题，如果是Java 9以后，我们完全可以采用另外一种方式实现，也就是Variable Handle API，这是源自于JEP 193，提供了各种粒度的原子或者有序性的操作等。我将前面的代码修改为如下实现：

```
private static final VarHandle HANDLE = MethodHandles.lookup().findStaticVarHandle
        (AtomicBTreePartition.class, "lock");

private void acquireLock(){
    long t = Thread.currentThread().getId();
    while (!HANDLE.compareAndSet(this, 0L, t)){
        // 等待一会儿，数据库操作可能比较慢
        …
    }
}
```

过程非常直观，首先，获取相应的变量句柄，然后直接调用其提供的CAS方法。

一般来说，我们进行的类似CAS操作，可以并且推荐使用Variable Handle API去实现，其提供了精细粒度的公共底层API。我这里强调公共，是因为其API不会像内部API那样，发生不可预测的修改，这一点提供了对于未来产品维护和升级的基础保障，坦白说，很多额外工作量，都是源于我们使用了Hack而非Solution的方式解决问题。

CAS也并不是没有副作用，试想，其常用的失败重试机制，隐含着一个假设，即竞争情况是短暂的。大多数应用场景中，确实大部分重试只会发生一次就获得了成功，但是总是有意外情况，所以在有需要的时候，还是要考虑限制自旋的次数，以免过度消耗CPU。

另外一个就是著名的ABA问题，这是通常只在lock-free算法下暴露的问题。我前面说过CAS是在更新时比较前值，如果对方只是恰好相同，例如期间发生了 A -> B -> A的更新，仅仅判断数值是A，可能导致不合理的修改操作。针对这种情况，Java提供了AtomicStampedReference工具类，通过为引用建立类似版本号（stamp）的方式，来保证CAS的正确性，具体用法请参考这里的介绍。

前面介绍了CAS的场景与实现，幸运的是，大多数情况下，Java开发者并不需要直接利用CAS代码去实现线程安全容器等，更多是通过并发包等间接享受到lock-free机制在扩展性上的好处。

下面我来介绍一下AbstractQueuedSynchronizer（AQS），其是Java并发包中，实现各种同步结构和部分其他组成单元（如线程池中的Worker）的基础。

AQS

学习　　，如果上来就去看它的一系列方法（下图所示），很有可能把自己看晕，这种似懂非懂的状态也没有太大的实践意义。

我建议的思路是，尽量简化一下，理解为什么需要AQS，如何使用AQS，至少要做什么，再进一步结合JDK源代码中的实践，理解AQS的原理与应用。

Doug Lea曾经介绍过AQS的设计初衷。从原理上，一种同步结构往往是可以利用其他的结构实现的，例如我在专栏第19讲中提到过可以使用Semaphore实现互斥锁。但是，对某种同步结构的倾向，会导致复杂、晦涩的实现逻辑，所以，他选择了将基础的同步相关操作抽象在AbstractQueuedSynchronizer中，利用AQS为我们构建同步结构提供了范本。

AQS内部数据和方法，可以简单拆分为：

- 一个volatile的整数成员表征状态，同时提供了setState和getState方法

```
private volatile int state;
```

- 一个先入先出（FIFO）的等待线程队列，以实现多线程间竞争和等待，这是AQS机制的核心之一。
- 各种基于CAS的基础操作方法，以及各种期望具体同步结构去实现的acquire/release方法。

利用AQS实现一个同步结构，至少要实现两个基本类型的方法，分别是acquire操作，获取资源的独占权；还有就是release操作，释放对某个资源的独占。

以ReentrantLock为例，它内部通过扩展AQS实现了Sync类型，以AQS的state来反映锁的持有情况。

```
private final Sync sync;
abstract static class Sync extends AbstractQueuedSynchronizer { …}
```

下面是ReentrantLock对应acquire和release操作，如果是CountDownLatch则可以看作是await()/countDown()，具体实现也有区别。

```
public void lock() {
    sync.acquire(1);
}
public void unlock() {
    sync.release(1);
}
```

排除掉一些细节，整体地分析acquire方法逻辑，其直接实现是在AQS内部，调用了tryAcquire和acquireQueued，这是两个需要搞清楚的基本部分。

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

首先，我们来看看tryAcquire。在ReentrantLock中，tryAcquire逻辑实现在NonfairSync和FairSync中，分别提供了进一步的非公平或公平性方法，而AQS内部tryAcquire仅仅是个接近未实现的方法（直接抛异常），这是留个实现者自己定义的操作。

我们可以看到公平性在ReentrantLock构建时如何指定的，具体如下：

```
public ReentrantLock() {
        sync = new NonfairSync(); // 默认是非公平的
    }
    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

以非公平的tryAcquire为例，其内部实现了如何配合状态与CAS获取锁，注意，对比公平版本的tryAcquire，它在锁无人占有时，并不检查是否有其他等待者，这里体现了非公平的语义。

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();// 获取当前AQS内部状态量
    if (c == 0) { // 0表示无人占有，则直接用CAS修改状态位，
        if (compareAndSetState(0, acquires)) {// 不检查排队情况，直接争抢
            setExclusiveOwnerThread(current);  //并设置当前线程独占锁
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) { //即使状态不是0，也可能当前线程是锁持有者，因为这是再入锁
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

接下来我再来分析acquireQueued，如果前面的tryAcquire失败，代表着锁争抢失败，进入排队竞争阶段。这里就是我们所说的，利用FIFO队列，实现线程间对锁的竞争的部分，算是是AQS的核心逻辑。

当前线程会被包装成为一个排他模式的节点（EXCLUSIVE），通过addWaiter方法添加到队列中。acquireQueued的逻辑，简要来说，就是如果当前节点的前面是头节点，则试图获取锁，一切顺利则成为新的头节点；否则，有必要则等待，具体处理逻辑请参考我添加的注释。

```
final boolean acquireQueued(final Node node, int arg) {
      boolean interrupted = false;
      try {
        for (;;) {// 循环
            final Node p = node.predecessor();// 获取前一个节点
            if (p == head && tryAcquire(arg)) { // 如果前一个节点是头结点，表示当前节点合适去tryAcquire
                setHead(node); // acquire成功，则设置新的头节点
                p.next = null; // 将前面节点对当前节点的引用清空
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node)) // 检查是否失败后需要park
                interrupted |= parkAndCheckInterrupt();
        }
       } catch (Throwable t) {
        cancelAcquire(node);// 出现异常，取消
        if (interrupted)
              selfInterrupt();
        throw t;
      }
}
```

到这里线程试图获取锁的过程基本展现出来了，tryAcquire是按照特定场景需要开发者去实现的部分，而线程间竞争则是AQS通过Waiter队列与acquireQueued提供的，在release方法中，同样会对队列进行对应操作。

今天我介绍了Atomic数据类型的底层技术CAS，并通过实例演示了如何在产品代码中利用CAS，最后介绍了并发包的基础技术AQS，希望对你有所帮助。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天布置一个源码阅读作业，AQS中Node的waitStatus有什么作用？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

wenxueliu

2018-06-26

建议；

1. 希望能有推外内存的主题，范型部分希望能与cpp比较讲解。
2. 一些主题如果已经有公开的比较好的资料，可以提供链接，对重点强调即可。希望能看到更多公开资料所没有的信息，这也是老鸟们付费的初衷。
同意的点赞

Cui

2018-06-26

老师，看了AQS的实现原理后，我再回顾了您之前关于synchronized的文章，心中有些疑问：
1、synchronized在JVM中是会进行锁升级和降级的，并且是基于CAS来掌握竞争的情况，在竞争不多的情况下利用CAS的轻量级操作来减少开销。
2、而AQS也是基于CAS操作队列的，位于队列头的节点优先获得锁，其他的节点会被LockSupport.park()起来（这个好像依赖的是操作系统的互斥锁，应该也是个重量级操作）。
我觉得这两种方式都是基于CAS操作的，只是操作的对象不同（一个是Mark Word，一个是队列节点），当竞争较多时，还是不可避免地会使用到操作系统的互斥锁。然而，我再测试这两者的性能时，在无竞争的情况下，两者性能相当，但是，当竞争起来后，AQS的性能明显比synchronized要好（测试案例是8个线程并发对一个int递增，每个线程递增1000万次，AQS的耗时大概要少30%），这是为什么呢？

作者回复

2018-06-28

Locksupport的实现据说速度快，我也没具体对比过；不过jdk9里，monitor相关操作也加快了，可以看看jep143

二木��

2018-06-28

一直很好奇，为何CAS指令在发现内容未变的时候就能判断没有其他线程修改呢？可能被修改后的值与比较的值一样呀

OneThin

2018-07-16

能否出一节讲一下unsafe，感觉这个才是最基础的。另外unsafe为什么叫unsafe呢

王胖小子

2018-07-09

CAS有部分实现是解决ABA问题，可以讲一下ABA问题是如何解决的，除了version外，还有没有其他的方式

卡斯瓦德

2018-07-05

老师请教个问题，acquireQueued的源代码中，使用for（；；）做了个自旋锁吧，作者为什么不用while（true），这种方式呢，是因为开销不一样吗？

作者回复

2018-07-07

也许，这个我不知道具体原因，看上去while会比for多一个变量

黄明恩

2018-06-28

老师可否分析下Object.wait和notify的原理

爱新觉罗老流氓

2018-06-27

ReentrantLock的非公平锁，其实只有一次非公平的机会！那一次就是在lock方法中，非公平锁的实现有if else分支，在if时就进行一次cas state，成功的线程去执行任务代码去了。那么失败的线程就会进入else逻辑，就是AQS#acquired，从这里开始非公平锁和公平锁就完全一样了，只是公平锁被欺负了一次，它的lock方法是直接调acquired方法。

为什么只有这一次呢？先看AQS#acquired，第一个逻辑是tryAcquired，公平锁和非公平锁实现略有区别。但记住，在这个时刻下，即使你看到公平锁trtAcquired实现中多一个hasQueuedPredecessors判断，无关紧要，重要的是这个时刻，还没有执行后面的addWaiter逻辑，根本没有入队，那么公平锁进入这个hasXXX方法，当然也是马上出来，执行后面的cas state，跟非公平锁没有不同...

如果，AQS#acquired的第一个tryAcquired失败了，都会进入acquiredQueued，此方法中有个强制的逻辑，就是无限for循环中的 final Node p = node.predecessors(); 在这个逻辑下，非平锁锁也要乖乖排队……

以上只是分析了lock方法，带超时的tryLock方法还没有具体看代码。如果我的lock分析有误，欢迎指出批评！

三口先生

2018-06-26

大于0取消状态，小于0有效状态，表示等待状态四种cancelled，signal，condition，propagate

作者回复

2018-06-28

不错

antipas

2018-06-26

看AQS源码过程中产生了新问题，它对线程的挂起唤醒是通过locksupport实现的，那么它与wait/notify又有何不同，使用场景有何不同。我的理解是使用 wait/notify需要synchronized锁，而且wait需要条件触发

作者回复

2018-06-28

这是两种方式，wait基于monitor，一般用并发库就不用Object.wait、notify之类了

I.am DZX

2018-06-26

CANCELLED 1 因为超时或中断设置为此状态，标志节点不可用
SIGNAL -1 处于此状态的节点释放资源时会唤醒后面的节点
CONDITION -2 处于条件队列里，等待条件成立(signal signalall) 条件成立后会置入获取资源的队列里
PROPAGATE -3 共享模式下使用，头节点获取资源时将后面节点设置为此状态，如果头节点获取资源后还有足够的资源，则后面节点会尝试获取，这个状态主要是为了共享状态下队列里足够多的节点同时获取资源
0 初始状态

作者回复

2018-06-28

好

三木子

2018-06-26

最近遇到配置tomcat连接池，导致cpu过高问题，最后发现配置连接池数过大导致上下文切换次数过多
，也就是线程池中任务数过少，空闲的线程过多，我想问为什么会导致上下文切换过多？

TonyEasy

2018-06-26

老师，说实话这一期的对我来说有点难度了，钦佩老师对知识理解的深入，请问老师可以指点下java学习的路线图吗，或者您分享下您自己的学习路线。

作者回复

2018-06-28

大家基础不一样，以后被问到不生疏也好；关于路线，不知道你的兴趣和规划是什么，通常来说Java只是技能树中的一项，项目经验，领域知识，综合起来才能要到高价

极客时间

极客时间

极客时间