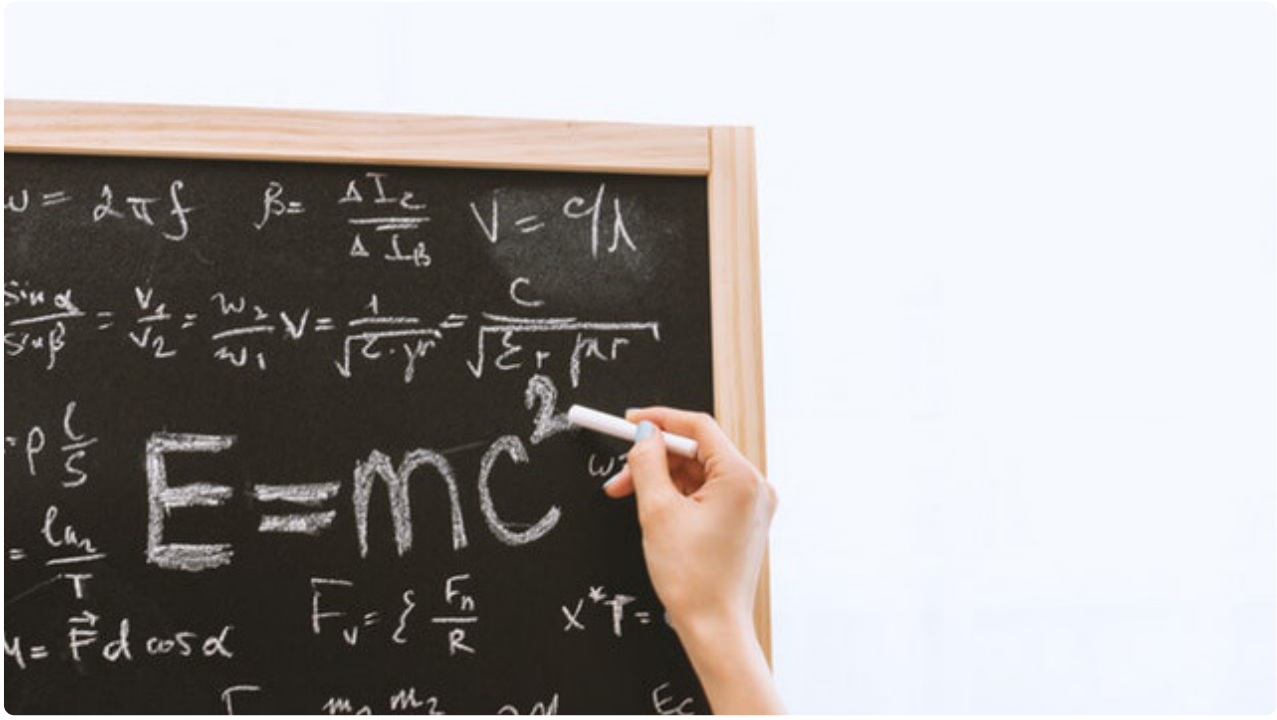
15 原子性轻量级实现—深入理解Atomic与CAS

更新时间：2019-10-17 10:45:37



> 构成我们学习最大障碍的是已知的东西，而不是未知的东西。
>
> —— 贝尔纳

在上一章介绍了并发的三大特性，即原子性、可见性和有序性。从本节起，我们将学习如何在多线程开发中确保这三大特性。首先，最简单的方式就是使用 synchronized 关键字或者其它加锁。这种方式最大的好处是–简单！是的，无需动脑子，在需要的地方加锁就好了。同步方式在并发时包治百病，但治病的手段却是让多线程程序转为串行执行，这相当于自毁武功。如果滥用同步，那么程序就是去了多线程的意义。因此，只有在必要的时候才使用同步。比如对共享资源的访问。而且尽量控制同步代码块的范围，不需要使用同步的代码，尽量不要放入同步代码块。

那么除了使用 synchronized 实现同步，还有其它手段保证三大特性吗？答案是肯定的，Java 还提供了轻量级的实现，来解决特定的问题。这些实现方式不像 synchronized 能够包治百病，但是对症下药，疗效更好。对于程序来说，在解决问题的同时，还能保证代码的效率。所以我们需要掌握好 synchronized 同步之外的这些方法，遇到并发问题时，采用更为合适的手段解决问题，而不是一股脑的都用 synchronized 或者其它显式锁的方式实现同步。这样才是一位合格的攻城狮！

本节我们来看看原子性的轻量级实现–Atomic。

1. Atomic 简介

Atomic 相关类在 java.util.concurrent.atomic 包中。针对不同的原生类型及引用类型，有 AtomicInteger、AtomicLong、AtomicBoolean、AtomicReference 等。另外还有数组对应类型 AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray。由于 Atomic 提供的功能类似，就不一个个过了。我们以 AtomicInteger 为例，看看 Atomic 类型变量所能提供的功能。

我们先看一个简单的例子，运算逻辑是对变量 count 的累加。假如 count 为 int 类型，多个线程并发时，可能各自读取到了同样的值，也可能 A 线程读到 2，但由于某种原因更新晚了，count 已经被其它线程更新为了 4，但是线程 A 还是继续执行了 count+1 的操作，count 反而被更新为更小的值 3。现在的多线程程序是不安全的。要处理此问题，按照我们已经学习过的知识，需要把 count=count+1 放入 synchronized 代码块中。这样做肯定能够解决问题。但是这种同步操作是悲观锁的方式，每次都认为有其它线程在和它并发操作，所以每次都要对资源进行锁定，而加锁这个操作自身就有很大消耗。而且不是每一次 count+1 时都有并发发生，无并发发生时的加锁并无必要。直接用 synchronized 进行同步，效率并不高。

下面我们看看怎么用 AtomicInteger 解决这个问题。使用 AtomicInteger 很简单，我们在声明 count 的时候，将其声明为 AtomicInteger 即可，然后把 count=count+1 的语句改为 count.incrementAndGet ()。问题就完美解决了。

接下来我们看看 Atomic 实现原子操作的原理。我们首先看看 AtomicInteger 的 incrementAndGet 方法注释：

```
/**
 * Atomically increments by one the current value.
 *
 * @return the updated value
 */
```

可以看到此方法以原子操作在当前 value 上加 1。count=count+1 这行语句其实隐含了两步操作，第一步取得 count 的值，第二步为 count 加 1 。而在这两步操作中间，count 的值可能已经改变了。而 AtomicInteger 提供的 incrementAndGet () 方法，则把这两步操作作为一个原子性操作来完成，则不会出现线程安全问题。

Atomic 变量的操作是如何保证原子性的呢？其实是使用了 CAS 算法。

2. CAS 算法分析

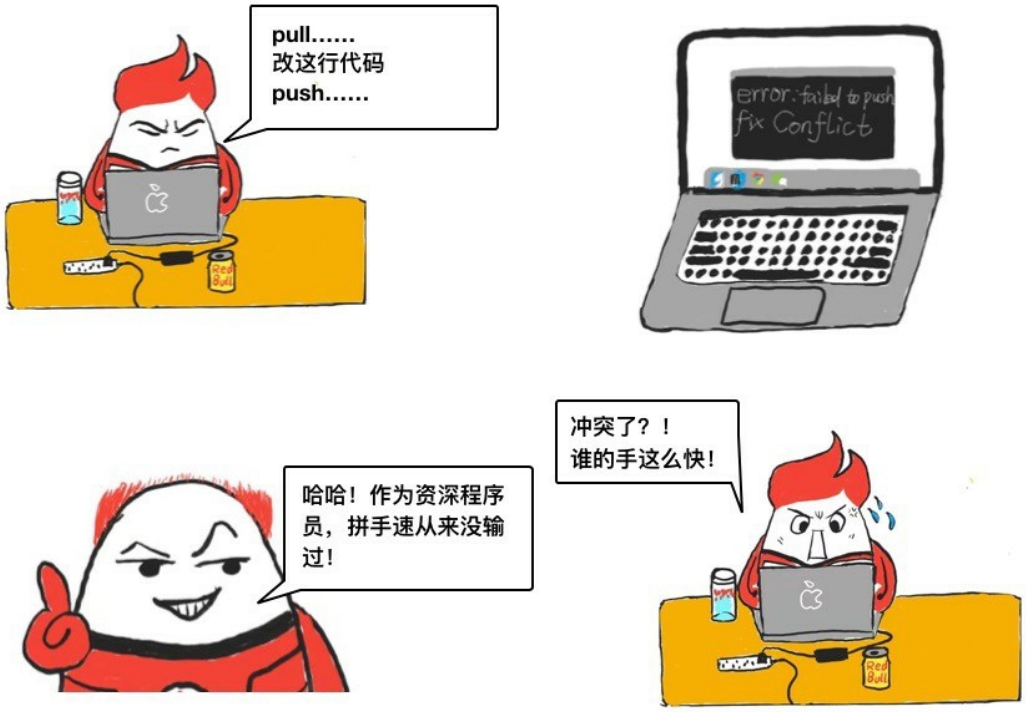
CAS 是 Compare and swap 的缩写，翻译过来就是比较替换。其实 CAS 是乐观锁的一种实现。而 Synchronized 则是悲观锁。这里的乐观和悲观指的是当前线程对是否有并发的判断。

悲观锁–认为每一次自己的操作大概率会有其它线程在并发，所以自己在操作前都要对资源进行锁定，这种锁定是排他的。悲观锁的缺点是不但把多线程并行转化为了串行，而且加锁和释放锁都会有额外的开支。

乐观锁–认为每一次操作时大概率不会有其它线程并发，所以操作时并不加锁，而是在对数据操作时比较数据的版本，和自己更新前取得的版本一致才进行更新。乐观锁省掉了加锁、释放锁的资源消耗，而且在并发量并不是很大的时候，很少会发生版本不一致的情况，此时乐观锁效率会更高。

Atomic 变量在做原子性操作时，会从内存中取得要被更新的变量值，并且和你期望的值进行比较，期望的值则是你要更新操作的值。如果两个值相等，那么说明没有其它线程对其更新，本线程可以继续执行。如果不等，说明有线程已经先于此线程进行了更新操作。那么则继续取得该变量的最新值，重复之前的逻辑，直至操作成功。这保证了每个线程对 Atomic 变量操作是线程安全的。

这里举个例子，我们每天都会向代码库提交代码，不知道你是否遇到过如下场景。你发现代码中有个 bug，只需要修改一行代码就可以修复，于是你先 pull，改好这行代码后立刻 push，但是 git 告诉你由于落后远程代码库的版本，push 失败了。很不巧，就在你 pull 和 push 之间这短短的几秒钟，有其它开发 push 了代码。那你只能再次 pull，和你这次修改做合并，然后再次 push。仔细想想，这不就是 CAS 吗？只不过除了数据提交前的版本比较 git 帮你做外，pull、merge、push 需要你手动执行。



3. Atomic 源代码分析

下面我们看看 AtomicInteger 的源代码。首先，AtomicInteger 中有 3 个重要的成员变量：

```
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;
private volatile int value;
```

第一个 Unsafe 对象，Atomic 中的原子操作都是借助 unsafe 对象所实现的；

第二个是 AtomicInteger 包装的变量在内存中的地址；

第三个是 AtomicInteger 包装的变量值，并且用 volatile 修饰，以确保变量的变化能被其它线程看到。

其实 valueOffset 就是 value 的内存地址。

AtomicInteger 中有一段静态代码块如下：

```
static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}
```

这段代码中 unsafe 对象获取了 AtomicInteger 类中 value 这个字段的 offset。unsafe.objectFieldOffset () 是一个 native 的方法。

AtomicInteger 有一个构造函数如下：

```
public AtomicInteger(int initialValue) {
    value = initialValue;
}
```

可以看到对它所包装的 int 变量 value 进行了赋值。

通过以上分析，我们来总结一下目前对 AtomicInteger 的了解：

1. AtomicInteger 对象包装了通过构造函数传入的一个初始 int 值；
2. AtomicInteger 持有这个 int 变量的内存地址；
3. AtomicInteger 还有一个用来做原子性操作的 unsafe 对象。

接下来我们以文章前面提到的 incrementAndGet 方法为例，来看看 Atomic 原子性的实现。代码如下：

```
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

代码很简单，调用了 *unsafe*.getAndAddInt(this, *valueOffset*, 1) 后，对其返回 +1，然后 return。

那么原子性实现的秘密就全在 *unsafe*.getAndAddInt () 这个方法中了。随便翻看一下 AtomicInteger 的源代码，这个方法被各种调用，其实我们搞清楚 *unsafe*.getAndAddInt () 的实现，谜底也就揭晓了。我们继续看 *unsafe*.getAndAddInt () 的实现：

```
public final int getAndAddInt(Object obj, long valueOffset, int var) {
        int expect;
        // 利用循环，直到更新成功才跳出循环。
        do {
            // 获取value的最新值
            expect = this.getIntVolatile(obj, valueOffset);
            // expect + var表示需要更新的值，如果compareAndSwapInt返回false，说明value值被其他线程更改了。
            // 那么就循环重试，再次获取value最新值expect，然后再计算需要更新的值expect + var。直到更新成功
        } while(!this.compareAndSwapInt(obj, valueOffset, expect, expect + var));

        // 返回当前线程在更改value成功后的，value变量原先值。并不是更改后的值
        return expect;
    }
```

为了帮助理解，我加了一些注释。三个入参，第一个 obj 传入的是 AtomicInteger 对象自己，第二个是 value 变量的内存地址，第三个则是要增加的值。

程序体中是一个循环，循环中通过 AtomicInteger 对象和 value 属性的 offset，取得到当前的 value 值，接下来调用 this.compareAndSwapInt (obj, valueOffset, expect, expect + var)。这个方法名仔细看下，是不是很熟悉？是的，就是 CAS。调用前我们已经获取到了期望值，所以在这个方法中会把期望值和你要替换掉的值做比较，如果一直则替换，否则重复 while 循环，也就是再此获取最新的期望值，然后再比较替换，直至替换成功。

你现在一定很好奇 compareAndSwapInt 的方法是如何实现的。我们点开此方法后，可以看到是一个 native 方法，native 方法使用 C 语言编写。由于 JDK 并未开源，我们只能下载开源版本的 OpenJDK。

可以看到在 compareAndSwapInt 源代码的最后，调用了 Atomic::cmpxchg (x,addr,e)。这个方法在不同的平台会有不同的实现。不过总的思想如下：

1. 判断当前系统是否为多核处理器；
2. 执行 CPU 指令 cmpxchg，如果为多核则在 cmpxchg 加 lock 前缀。

可以看到最终是通过 CPU 指令 cmpxchg 来实现比较交换。那么 Lock 前缀起到什么作用呢？加了 Lock 前缀的操作，在执行期间，所使用的缓存会被锁定，其他处理器无法读写该指令要访问的内存区域，由此保证了比较替换的原子性。而这个操作过程称之为缓存锁定。

4. CAS 的缺点

CAS 最终通过 CPU 指令实现，把无谓的同步消耗降到最低，但是没有银弹，CAS 也有着几个致命的缺点：

1. 比较替换如果失败，则会一直循环，直至成功。这在并发量很大的情况下对 CPU 的消耗将会非常大；
2. 只能保证一个变量自身操作的原子性，但多个变量操作要实现原子性，是无法实现的；
3. ABA 问题。

前两个问题比较简单，我们重点看一下第三个 ABA 问题。

假如本线程更新前取得期望值为 A，和更新操作之间的这段时间内，其它线程可能把 value 改为了 B 又改回了 A。而本线程更新时发现 value 和期望值一样还是 A，认为其没有变化，则执行了更新操作。但其实此时的 A 已经不是彼时的 A 了。

大多数情况下 ABA 不会造成业务上的问题。但是如果你认为 ABA 问题对你的程序业务有问题，那么就需要解决。JDK 提供了 AtomicStampedReference 类，通过对 Atomic 包装的变量增加版本号，来解决 ABA 问题，即使 value 还是 A，但如果版本变化了，也认为比较失败。

5. 总结

本节我们学习了轻量级的原子性实现–Atomic。并且以 AtomicInteger 为例进行了源代码的讲解，Atomic 的类很多，但是大同小异，感兴趣的话，可以自己读一下其它 Atomic 类的源代码。本节最后介绍了 CAS，一定要深入理解，这也是面试中经常会问到的问题之一。我们经过本节的学习，了解了 Atomic 的优点，也知道了它的局限性。在以后的多线程开发中，可以有选择的使用 Atomic 变量，以使程序达到更好的效率。

}

14 僵持不下—死锁详解

16 让你眼见为实—volatile详解