

漫画：什么是CAS机制？（进阶篇）

原创：永远爱大家的

点击上方“[程序员小灰](#)”，选择“置顶公众号”

有趣有内涵的文章第一时间送达！

上一期为大家讲解的CAS机制的基本概念，没看过的小伙伴们可以点击下面的链接：

[漫画：什么是 CAS 机制？](#)

这一期我们来深入介绍之前遗留的两个问题：

1. Java当中CAS的底层实现
2. CAS的ABA问题和解决方法

大黄，上一次听你讲解 CAS，
收获还蛮大的。不过还有几个
问题我没太搞清楚。



有什么问题你说说吧，我尽量给你解答。



第一个问题，CAS 的底层究竟是怎么来实现的？比如 `AtomicInteger`，是怎么做到原子性的比较和更新一个值？



要回答这个问题，我们得先来看看 `AtomicInteger` 的源代码。



首先看一看AtomicInteger当中常用的自增方法 **incrementAndGet**:

```
public final int incrementAndGet() {  
  
    for (;;) {  
  
        int current = get();  
  
        int next = current + 1;  
  
        if (compareAndSet(current, next))  
  
            return next;  
  
    }  
  
}  
  
private volatile int value;  
  
public final int get() {  
  
    return value;  
  
}
```

这段代码是一个无限循环，也就是CAS的自旋。循环体当中做了三件事：

1. 获取当前值。

2. 当前值+1，计算出目标值。

3. 进行CAS操作，如果成功则跳出循环，如果失败则重复上述步骤。

这里需要注意的重点是 `get` 方法，这个方法的作用是获取变量的当前值。

如何保证获得的当前值是内存中的最新值呢？很简单，用**`volatile`**关键字来保证。有关**`volatile`**关键字的知识，我们之前有介绍过，这里就不详细阐述了。

外面的自旋操作我看懂了，可是
`compareAndSet` 方法是如何保证原
子性操作的呢？



这就需要我们深入看一看
compareAndSet 方法内部的实现。



接下来看一看compareAndSet方法的实现，以及方法所依赖对象的来历：

```
public final boolean compareAndSet(int expect, int update) {  
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
}  
  
private static final Unsafe unsafe = Unsafe.getUnsafe();  
private static final long valueOffset;  
  
static {  
    try {  
        valueOffset = unsafe.objectFieldOffset  
            (AtomicInteger.class.getDeclaredField("value"));  
    } catch (Exception ex) { throw new Error(ex); }  
}
```

compareAndSet方法的实现很简单，只有一行代码。这里涉及到两个重要的对象，一个是**unsafe**，一个是**valueOffset**。

什么是unsafe呢？Java语言不像C，C++那样可以直接访问底层操作系统，但是JVM为我们提供了一个后门，这个后门就是unsafe。unsafe为我们提供了**硬件级别的原子操作**。

至于valueOffset对象，是通过unsafe.objectFieldOffset方法得到，所代表的是**AtomicInteger对象value成员变量在内存中的偏移量**。我们可以简单地把valueOffset理解为value变量的内存地址。

我们在上一期说过，CAS机制当中使用了3个基本操作数：**内存地址V，旧的预期值A，要修改的新值B**。

而unsafe的compareAndSwapInt方法参数包括了这三个基本元素：valueOffset参数代表了V，expect参数代表了A，update参数代表了B。

正是unsafe的compareAndSwapInt方法保证了Compare和Swap操作之间的原子性操作。

OK，这个问题我大致懂了。还有另一个问题，CAS 当中的 ABA 是怎么回事呢？



所谓 ABA 问题，就是一个变量的值从 A 改成了 B，又从 B 改成了 A。



什么是ABA呢？假设内存中有一个值为A的变量，存储在地址V当中。



内存地址V

此时有三个线程想使用CAS的方式更新这个变量值，每个线程的执行时间有略微的偏差。线程1和线程2已经获得当前值，线程3还未获得当前值。



内存地址V

线程1: 获取当前值A, 期望更新为B
线程2: 获取当前值A, 期望更新为B
线程3: 期望更新为A

接下来, 线程1先一步执行成功, 把当前值成功从A更新为B; 同时线程2因为某种原因被阻塞住, 没有做更新操作; 线程3在线程1更新之后, 获得了当前值B。



内存地址V

线程1: 获取当前值A, **成功更新为B**
线程2: 获取当前值A, 期望更新为B, **BLOCK**
线程3: **获取当前值B**, 期望更新为A

再之后, 线程2仍然处于阻塞状态, 线程3继续执行, 成功把当前值从B更新成了A。



内存地址V

线程1: 获取当前值A, 成功更新为B, 已返回
线程2: 获取当前值A, 期望更新为B, BLOCK
线程3: 获取当前值B, 成功更新为A

最后, 线程2终于恢复了运行状态, 由于阻塞之前已经获得了“当前值”A, 并且经过compare检测, 内存地址V中的实际值也是A, 所以成功把变量值A更新成了B。



内存地址V

线程1: 获取当前值A, 成功更新为B, 已返回
线程2: 获取“当前值”A, 成功更新为B
线程3: 获取当前值B, 成功更新为A, 已返回

这个过程中, 线程2获取到的变量值A是一个旧值, 尽管和当前的实际值相同, 但内存地址V中的变量已经经历了A->B->A的改变。

可是这个例子看起来没毛病啊，
本来不就是要把 A 更新成 B 吗？



表面看起来没什么问题，但如果
我们结合应用场景，就可以
看出它的问题所在。



当我们举一个提款机的例子。假设有一个遵循CAS原理的提款机，小灰有100元存款，要用这个提款机来提款50元。

存款余额：100元



由于提款机硬件出了点小问题，小灰的提款操作被同时提交两次，开启了两个线程，两个线程都是获取当前值100元，要更新成50元。

理想情况下，应该一个线程更新成功，另一个线程更新失败，小灰的存款只被扣一次。

存款余额：100元



线程1(提款机): 获取当前值100，期望更新为50
线程2(提款机): 获取当前值100，期望更新为50

线程1首先执行成功，把余额从100改成50。线程2因为某种原因阻塞了。这时候，小灰的妈妈刚好给小灰汇款50元。

存款余额：50元



线程1(提款机): 获取当前值100, 成功更新为50
线程2(提款机): 获取当前值100, 期望更新为50, BLOCK
线程3(小灰妈): 获取当前值50, 期望更新为100

线程2仍然是阻塞状态, 线程3执行成功, 把余额从50改成100。

存款余额：100元



线程1(提款机): 获取当前值100, 成功更新为50, 已返回
线程2(提款机): 获取当前值100, 期望更新为50, BLOCK
线程3(小灰妈): 获取当前值50, 成功更新为100

线程2恢复运行，由于阻塞之前已经获得了“当前值” 100，并且经过compare检测，此时存款实际值也是100，所以成功把变量值100更新成了50。

存款余额：**50**元



线程1(提款机): 获取当前值100，成功更新为50，已返回
线程2(提款机): 获取“当前值” 100，**成功更新为50**
线程3(小灰妈): 获取当前值50，成功更新为100，已返回

流氓，把钱还我！



这个举例改编自《java特种兵》当中的一段例子。原本线程2应当提交失败，小灰的正确余额应该保持为100元，结果由于ABA问题提交成功了。

这下子总算明白了。那么 ABA 问题应该怎么来解决呢？



解决方法很简单，加个版本号就行。



什么意思呢？真正要做到严谨的CAS机制，我们在Compare阶段不仅要比较期望值A和地址V中的实际值，还要比较变量的版本号是否一致。

我们仍然以最初的例子来说明一下，假设地址V中存储着变量值A，当前版本号是01。线程1获得了当前值A和版本号01，想要更新为B，但是被阻塞了。

版本号01



内存地址V

线程1: 获取当前值A, 版本号01, 期望更新为B

这时候, 内存地址V中的变量发生了多次改变, 版本号提升为03, 但是变量值仍然是A。

版本号03



内存地址V

线程1: 获取当前值A, 版本号01, 期望更新为B

随后线程1恢复运行, 进行Compare操作。经过比较, 线程1所获得的值和地址V的实际值都是A, 但是版本号不相等, 所以这一次更新失败。

版本号03



内存地址V

线程1: 获取当前值A, 版本号01, 期望更新为B

A == A

01 != 03

更新失败!

在Java当中, **AtomicStampedReference**类就实现了用版本号做比较的CAS机制。

原来如此, 还真是个好方法!



最后，让我们回顾一下今天的所学：



1. Java语言CAS底层如何实现？

利用unsafe提供了原子性操作方法。

2. 什么是ABA问题？ 怎么解决？

当一个值从A更新成B，又更新会A，普通CAS机制会误判通过检测。

利用版本号比较可以有效解决ABA问题。

—————END—————

喜欢本文的朋友们，欢迎长按下图关注订阅号**程序员小灰**，收看更多精彩内容

