https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933166&idx=1&sn=15e614500676170b76a329efd3255c12&chksm=88621b10bf1592064befc5c9f0d78c56cda25c6d003e1711b85e5bfeb56c9fd30d892178db87&token=1033016931&lang=zh\_CN&scene=21#wechat\_redirect

# java中的CAS，你需要知道的东西

## **本文主要内容**

1. 从网站计数器实现中一步步引出CAS操作
2. 介绍java中的CAS及CAS可能存在的问题
3. 悲观锁和乐观锁的一些介绍及数据库乐观锁的一个常见示例
4. 使用java中的原子操作实现网站计数器功能

## **我们需要解决的问题**

**需求：我们开发了一个网站，需要对访问量进行统计，用户每次发一次请求，访问量+1，如何实现呢？**

下面我们来模仿有100个人同时访问，并且每个人对咱们的网站发起10次请求，最后总访问次数应该是1000次。实现访问如下。

### 方式1

代码如下：



输出：



代码中的count用来记录总访问次数，request()方法表示访问一次，内部休眠5毫秒模拟内部耗时，request方法内部对count++操作。程序最终耗时1秒多，执行还是挺快的，但是count和我们期望的结果不一致，我们期望的是1000，实际输出的是973（每次运行结果可能都不一样）。

分析一下问题出在哪呢？

代码中采用的是多线程的方式来操作count，count++会有线程安全问题，count++操作实际上是由以下三步操作完成的：

获取count的值，记做A：A=count

将A的值+1，得到B：B = A+1

让B赋值给count：count = B

如果有A、B两个线程同时执行count++，他们同时执行到上面步骤的第1步，得到的count是一样的，3步操作完成之后，count只会+1，导致count只加了一次，从而导致结果不准确。

那么我们应该怎么做的呢？

对count++操作的时候，我们让多个线程排队处理，多个线程同时到达request()方法的时候，只能允许一个线程可以进去操作，其他的线程在外面候着，等里面的处理完毕出来之后，外面等着的再进去一个，这样操作count++就是排队进行的，结果一定是正确的。

我们前面学了synchronized、ReentrantLock可以对资源加锁，保证并发的正确性，多线程情况下可以保证被锁的资源被串行访问，那么我们用synchronized来实现一下。

### 方式二：使用synchronized实现

代码如下：



输出：



程序中request方法使用synchronized关键字，保证了并发情况下，request方法同一时刻只允许一个线程访问，request加锁了相当于串行执行了，count的结果和我们预期的结果一致，只是耗时比较长，5秒多。

### 方式3

我们在看一下count++操作，count++操作实际上是被拆分为3步骤执行：

1. 获取count的值，记做A：A=count

2. 将A的值+1，得到B：B = A+1

3. 让B赋值给count：count = B

方式2中我们通过加锁的方式让上面3步骤同时只能被一个线程操作，从而保证结果的正确性。

我们是否可以只在第3步加锁，减少加锁的范围，对第3步做以下处理：

1、获取锁

2、第三步获取一下count最新的值，记做LV

3、判断LV是否等于A，如果相等，则将B的值赋给count，并返回true，否者返回false；

4、释放锁

如果我们发现第3步返回的是false，我们就再次去获取count，将count赋值给A，对A+1赋值给B，然后再将A、B的值带入到上面的过程中执行，直到上面的结果返回true为止。

我们用代码来实现，如下：



输出：



代码中用了volatile关键字修饰了count，可以保证count在多线程情况下的可见性。关于volatile关键字的使用，也是非常非常重要的，前面有讲过，不太了解的朋友可以去看一下：volatile与Java内存模型

咱们再看一下代码，compareAndSwap方法，我们给起个简称吧叫CAS，这个方法有什么作用呢？这个方法使用synchronized修饰了，能保证此方法是线程安全的，多线程情况下此方法是串行执行的。方法由两个参数，expectCount：表示期望的值，newCount：表示要给count设置的新值。方法内部通过getCount()获取count当前的值，然后与期望的值expectCount比较，如果期望的值和count当前的值一致，则将新值newCount赋值给count。

再看一下request()方法，方法中有个do-while循环，循环内部获取count当前值赋值给了expectCount，循环结束的条件是compareAndSwap返回true，也就是说如果compareAndSwap如果不成功，循环再次获取count的最新值，然后+1，再次调用compareAndSwap方法，直到compareAndSwap返回成功为止。

代码中相当于将count++拆分开了，只对最后一步加锁了，减少了锁的范围，此代码的性能是不是比方式2快不少，还能保证结果的正确性。大家是不是感觉这个compareAndSwap方法挺好的，这东西确实很好，java中已经给我们提供了CAS的操作，功能非常强大，我们继续向下看。

## **CAS**

CAS,compare and swap的缩写，中文翻译成比较并交换。

**CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该 位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前 值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”**

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新 值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

很多地方说CAS操作是非阻塞的，其实系统底层进行CAS操作的时候，会判断当前系统是否为多核系统，如果是就给总线加锁，所以同一芯片上的其他处理器就暂时不能通过总线访问内存，保证了该指令在多处理器环境下的原子性。总线上锁的，其他线程执行CAS还是会被阻塞一下，只是时间可能会非常短暂，所以说CAS是非阻塞的并不正确，只能说阻塞的时间是非常短的。

java中提供了对CAS操作的支持，具体在sun.misc.Unsafe类中，声明如下：

public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);

public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);

上面三个方法都是类似的，主要对4个参数做一下说明。

var1：表示要操作的对象

var2：表示要操作对象中属性地址的偏移量

var4：表示需要修改数据的期望的值

var5：表示需要修改为的新值

JUC包中大部分功能都是依靠CAS操作完成的，所以这块也是非常重要的，有关Unsafe类，下篇文章会具体讲解。

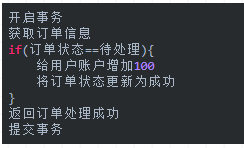
synchronized、ReentrantLock这种独占锁属于**悲观锁**，它是在假设需要操作的代码一定会发生冲突的，执行代码的时候先对代码加锁，让其他线程在外面等候排队获取锁。悲观锁如果锁的时间比较长，会导致其他线程一直处于等待状态，像我们部署的web应用，一般部署在tomcat中，内部通过线程池来处理用户的请求，如果很多请求都处于等待获取锁的状态，可能会耗尽tomcat线程池，从而导致系统无法处理后面的请求，导致服务器处于不可用状态。

除此之外，还有**乐观锁**，乐观锁的含义就是假设系统没有发生并发冲突，先按无锁方式执行业务，到最后了检查执行业务期间是否有并发导致数据被修改了，如果有并发导致数据被修改了 ，就快速返回失败，这样的操作使系统并发性能更高一些。cas中就使用了这样的操作。

关于乐观锁这块，想必大家在数据库中也有用到过，给大家举个例子，可能以后会用到。

如果你们的网站中有调用支付宝充值接口的，支付宝那边充值成功了会回调商户系统，商户系统接收到请求之后怎么处理呢？假设用户通过支付宝在商户系统中充值100，支付宝那边会从用户账户中扣除100，商户系统接收到支付宝请求之后应该在商户系统中给用户账户增加100，并且把订单状态置为成功。

处理过程如下：



由于网络等各种问题，可能支付宝回调商户系统的时候，回调超时了，支付宝又发起了一笔回调请求，刚好这2笔请求同时到达上面代码，最终结果是给用户账户增加了200，这样事情就搞大了，公司蒙受损失，严重点可能让公司就此倒闭了。

那我们可以用乐观锁来实现，给订单表加个版本号version，要求每次更新订单数据，将版本号+1，那么上面的过程可以改为：

获取订单信息,将version的值赋值给V\_A

if(订单状态==待处理){

开启事务

给用户账户增加100

update影响行数 = update 订单表 set version = version + 1 where id = 订单号 and version = V\_A;

if(update影响行数==1){

提交事务

}else{

回滚事务

}

}

返回订单处理成功

上面的update语句相当于我们说的CAS操作，执行这个update语句的时候，多线程情况下，数据库会对当前订单记录加锁，保证只有一条执行成功，执行成功的，影响行数为1，执行失败的影响行数为0，根据影响行数来决定提交还是回滚事务。上面操作还有一点是将事务范围缩小了，也提升了系统并发处理的性能。这个知识点希望你们能get到。

## **CAS 的问题**

cas这么好用，那么有没有什么问题呢？还真有。

### ABA问题

**CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了**。这就是CAS的ABA问题。常见的解决思路是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A-B-A 就会变成1A-2B-3A。目前在JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

### 循环时间长开销大

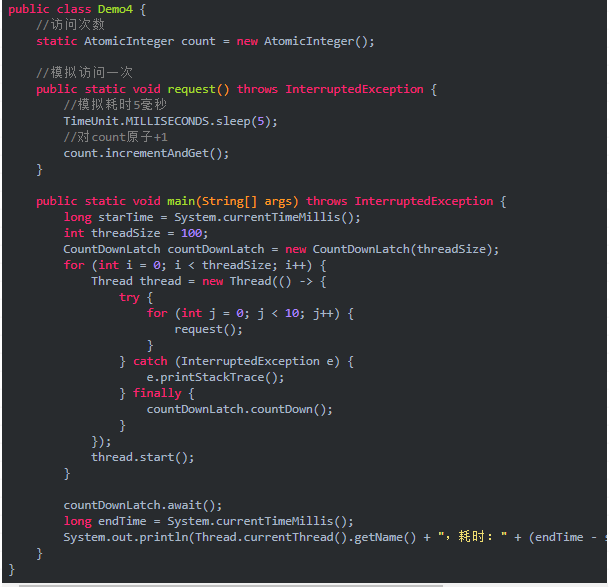
上面我们说过如果CAS不成功，则会原地循环（自旋操作），如果长时间自旋会给CPU带来非常大的执行开销。并发量比较大的情况下，CAS成功概率可能比较低，可能会重试很多次才会成功。

## **使用JUC中的类实现计数器**

juc框架中提供了一些原子操作，底层是通过Unsafe类中的cas操作实现的。通过原子操作可以保证数据在并发情况下的正确性。

此处我们使用java.util.concurrent.atomic.AtomicInteger类来实现计数器功能，AtomicInteger内部是采用cas操作来保证对int类型数据增减操作在多线程情况下的正确性。

计数器代码如下：



输出：



耗时很短，并且结果和期望的一致。

关于原子类操作，都位于java.util.concurrent.atomic包中，下篇文章我们主要来介绍一下这些常用的类及各自的使用场景。