# 并发编程之 CAS 的原理

锁的分类

乐观锁 ： 多线程中 读的操作多 写少 ，每个操作都不加锁 （实现：不加锁 往往通过版本控制）

悲观锁 ： 多线程中 写操作较多 读少， 每个操作都加锁 （实现：加 lock）

## 一、什么是CAS（乐观锁）

CAS （compareAndSwap），中文叫比较交换，是一种无锁原子算法。

它包含 3 个参数 CAS（V，E，N）

V表示要更新变量的值，E表示预期值

N表示新值。

仅当 V值等于E值时，才会将V的值设为N

如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程则什么都不做。最后，CAS 返回当前V的真实值。CAS 操作时抱着乐观的态度进行的，它总是认为自己可以成功完成操作。

总结：v = e ， 将 v = n 返回 v

V！=e ， 直接 将 v 返回

CAS的全称为Compare And Swap，直译就是比较交换。是一条CPU的原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子地更新某个位置的值，其实现方式是基于硬件平台的汇编指令，在intel的CPU中，使用的是cmpxchg指令，就是说CAS是靠硬件实现的，从而在硬件层面提升效率。

在通过CAS 操作变量时 需要提供一个期望值，与当前变量进行比较 。如果变量不等于 期望值 ，哪说明它已经被别人修改过了。你就需要重新读取，再次尝试修改就好了。

**优点 ：** 性能较高

1. 无锁 ，所以是非阻塞的
2. 无锁 ，性能高 提高cpu 的吞吐量 减少系统开销

## CAS底层原理

这样归功于硬件指令集的发展，实际上，我们可以使用同步将这两个操作变成原子的，但是这么做就没有意义了。所以我们只能靠硬件来完成，硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成。这类指令常用的有：

1. 测试并设置（Tetst-and-Set）

2. 获取并增加（Fetch-and-Increment）

3. 交换（Swap）

4. 比较并交换（Compare-and-Swap）

5. 加载链接/条件存储（Load-Linked/Store-Conditional）

CPU 实现原子指令有2种方式：

**1. 通过总线锁定来保证原子性。**

总线锁定其实就是处理器使用了总线锁，所谓总线锁就是使用处理器提供的一个 LOCK# 信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。但是该方法成本太大。因此有了下面的方式。

**2、通过缓存锁定来保证原子性。**

所谓 缓存锁定 是指内存区域如果被缓存在处理器的缓存行中，并且在Lock 操作期间被锁定，那么当他执行锁操作写回到内存时，处理器不在总线上声言 LOCK# 信号，而时修改内部的内存地址，并允许他的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改两个以上处理器缓存的内存区域数据（这里和 volatile 的可见性原理相同），当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

注意：有两种情况下处理器不会使用缓存锁定。

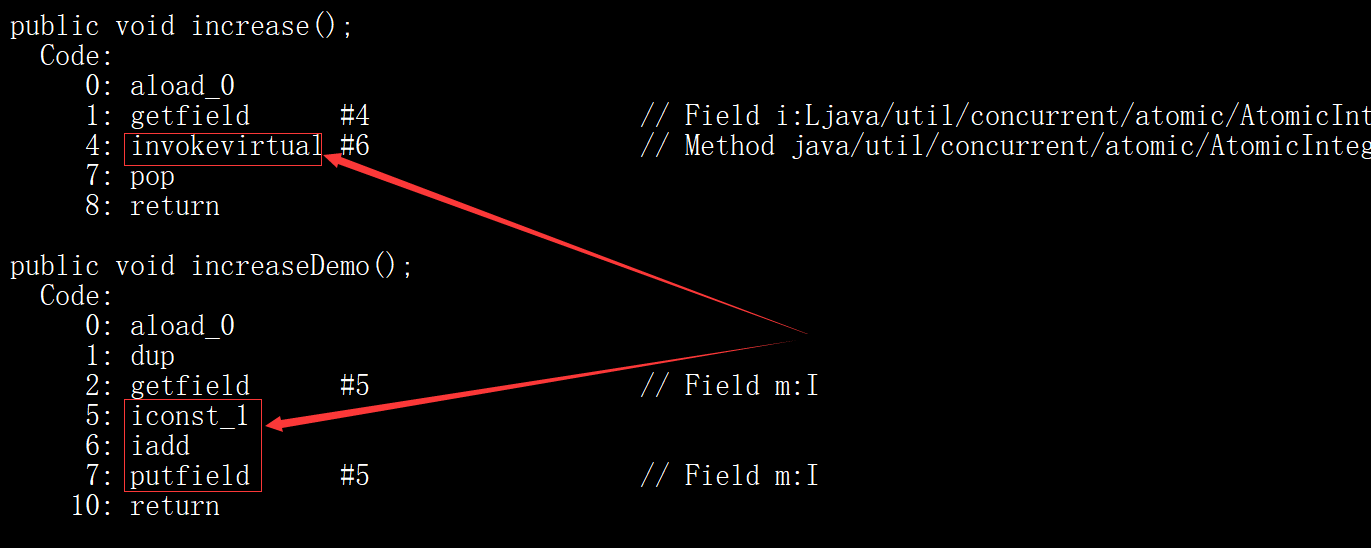
1. 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用总线锁定。

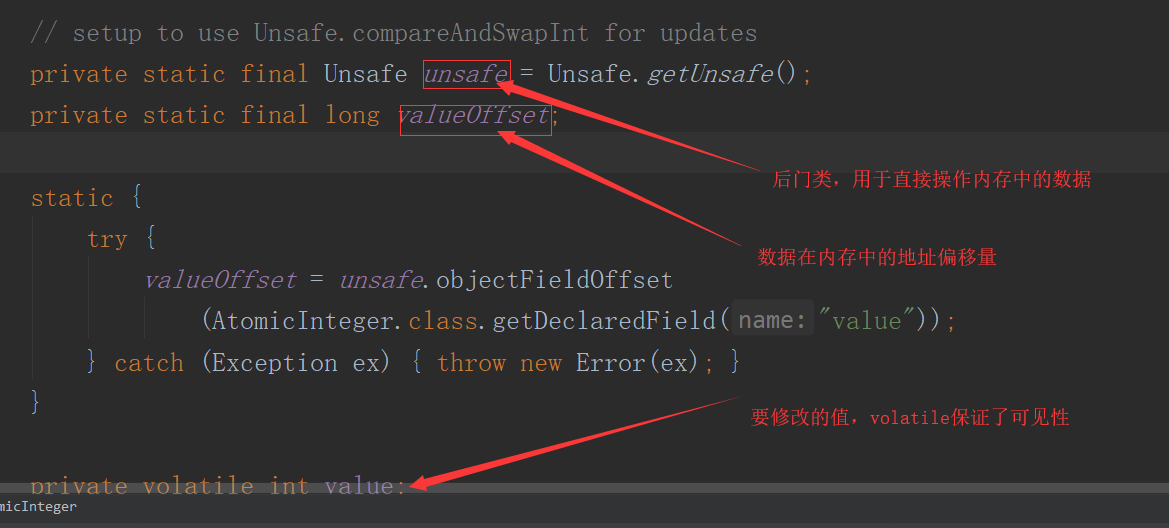
2. 有些处理器不支持缓存锁定，对于 Intel 486 和 Pentium 处理器，就是锁定的内存区域在处理器的缓存行也会调用总线锁定。

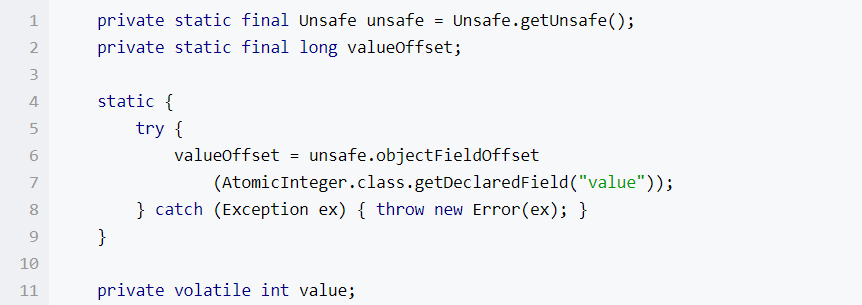
## CAS举例

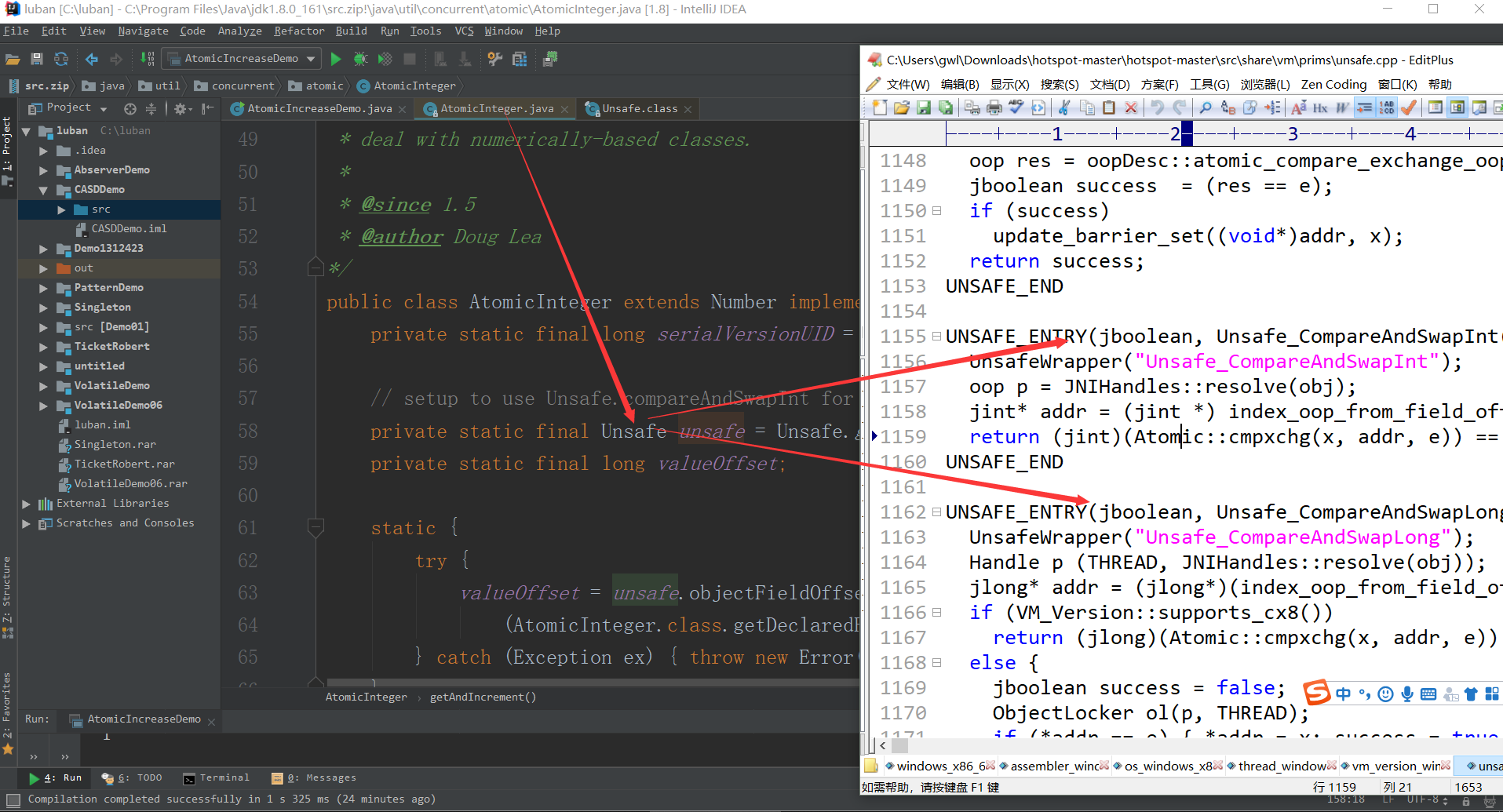
## CAS源码分析

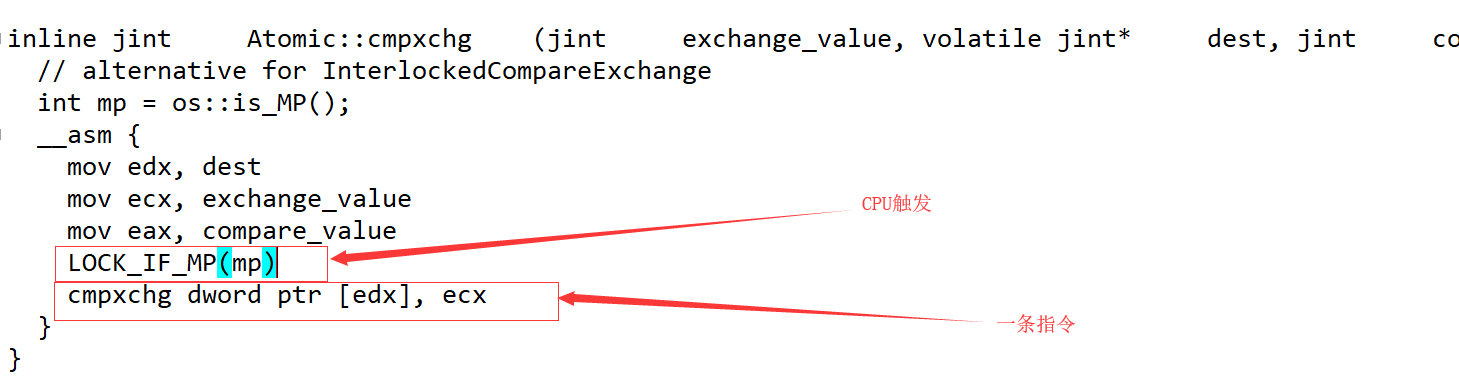
JUC下的atomic类都是通过CAS来实现的，下面就以AtomicInteger为例来阐述CAS的实现。如下：



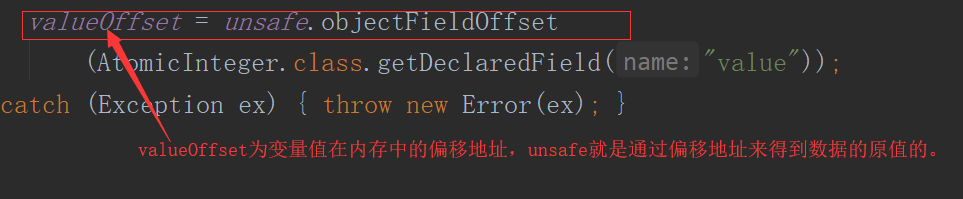


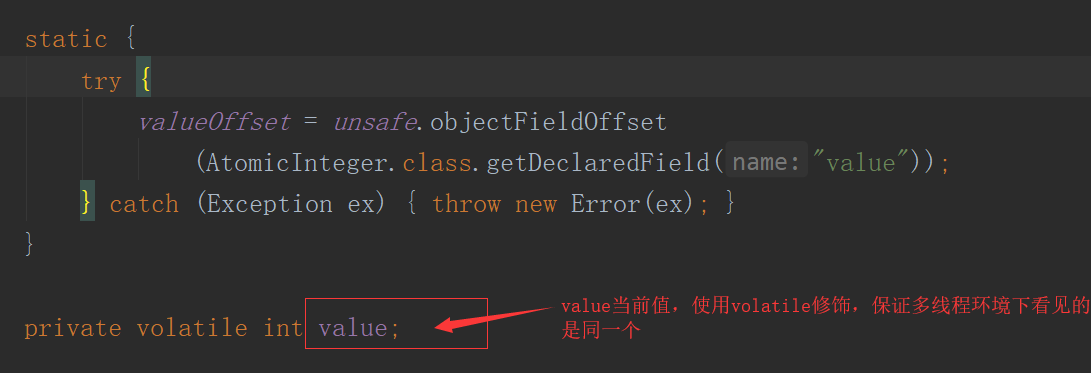






Unsafe是CAS的核心类，Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门：Unsafe，它提供了硬件级别的原子操作。





内部调用unsafe的getAndAddInt方法，在getAndAddInt方法中主要是看compareAndSwapInt方法：

CAS可以保证一次的读-改-写操作是原子操作，在单处理器上该操作容易实现，但是在多处理器上实现就有点儿复杂了。

缓存加锁：其实针对于上面那种情况我们只需要保证在同一时刻对某个内存地址的操作是原子性的即可。缓存加锁就是缓存在内存区域的数据如果在加锁期间，当它执行锁操作写回内存时，处理器不在输出LOCK#信号，而是修改内部的内存地址，利用缓存一致性协议来保证原子性。缓存一致性机制可以保证同一个内存区域的数据仅能被一个处理器修改，也就是说当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行

CAS缺点

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方法：循环时间太长、只能保证一个共享变量原子操作、ABA问题。

CAS 使用场景 ：

1. 适用于 简单的数据计算
2. 适合 线程冲突少的场合

循环时间太长

如果CAS一直不成功呢？这种情况绝对有可能发生，如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。在JUC中有些地方就限制了CAS自旋的次数，例如BlockingQueue的SynchronousQueue。

只能保证一个共享变量原子操作

看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了，当然如果你有办法把多个变量整成一个变量，利用CAS也不错。例如读写锁中state的高地位

****ABA问题****

CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，这就是所谓的ABA问题。对于ABA问题其解决方案是加上版本号，即在每个变量都加上一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A。

1. AQS

1、AbstractQueuedSynchronizer 同步发生器，用于构建lock。在juc包下是 reentrantLock 、 readWriteLock的基础

量级 ： cas < aqs< Synchronized

2、原理 ：通过内置的FIFO（利用clh队列先进先出的特性）同步对列 ，来实现线程争夺资源的 同步工作