# 什么是CAS？

## 并发

他首先他运用的场合一定是并发的，他的全程呢叫做compare and swap，是一种思想，它是一种算法，用来实现线程安全的算法，同时他也是一种CPU指令，如说compare and swap，这一条指令就可以完成比较并操作这样的一个组合操作，不会被打断，所以啊CAS最主要的就是被用在了并发编程领域，实现那些不能被打断的数据交换操作，从而避免我们多线程情况下，由于我们执行顺序是不能确定的，因为是多线程嘛，所以执行顺序不能确定，或者是某个线程被暂停了，这种不可预知的这种问题。

## CAS思路

我认为A的值应该是A，如果使得话那我就把它改成B，如果不是A(说明被别人修改过了)，那我就不修改了，避免多人同时修改导致出错。

我认为的这个A是我之前读取出来的，读取出来之后我在此基础上进行修改，然后我修改完之后呢，我想写回去之后呢，我会先去判断，他是不A，如果是的话我就改成B，B呢是我刚才计算出来的值，如果不是A，这就表明在我计算的过程中，这个值被别人修改了，那我就不能修改了，因为别人修改过了，在这种情况下，如果我再修改，相当于我并发的写了，那么这种时候，这种写入的结果呢是错误的。

所以用了这样一个最主要的思路之后呢就可以避免多人同时修改的时候的问题。

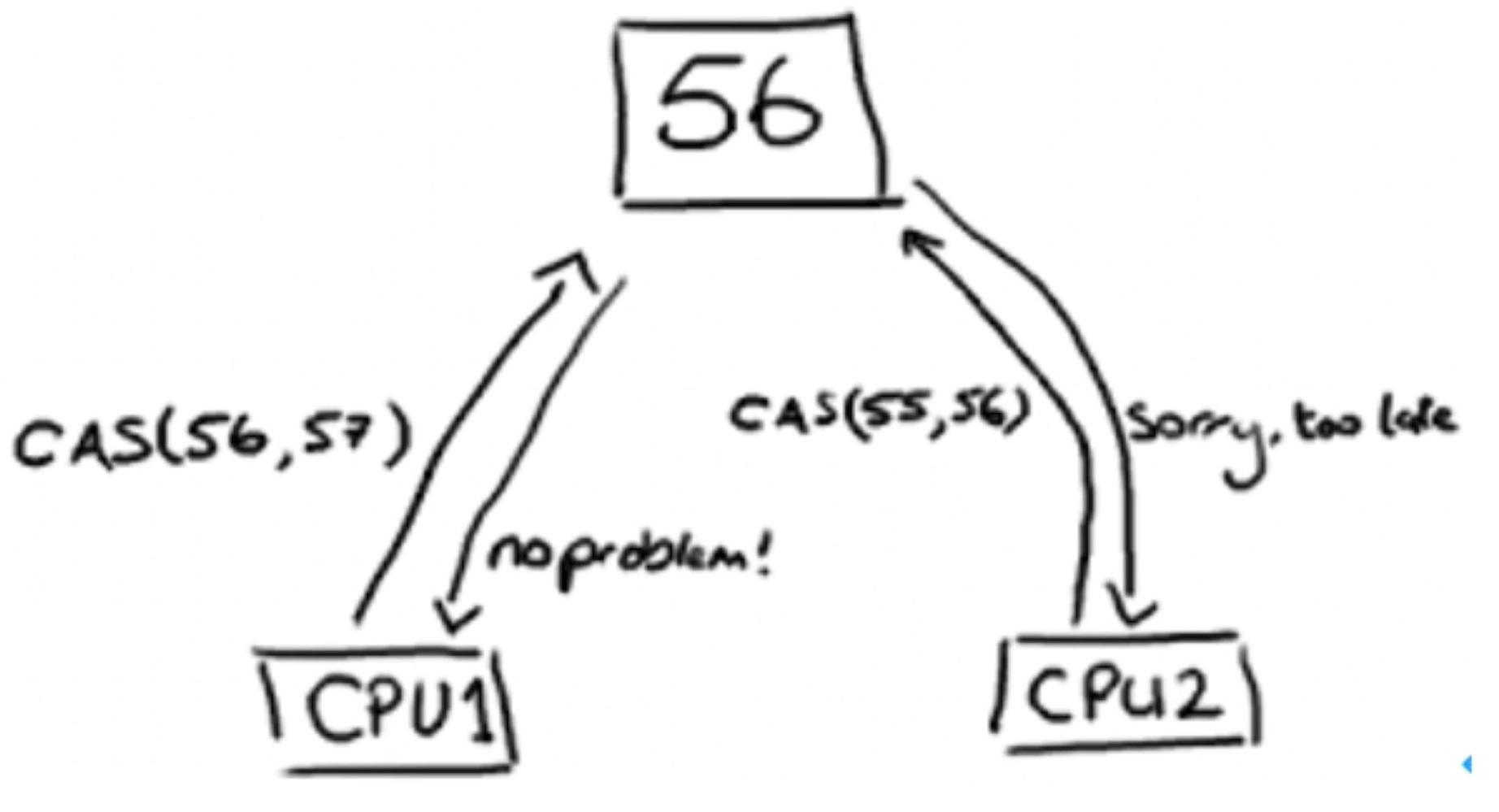
这就像我们在谈判的时候，一方是一个非常乐观的去协商的方式，我会给你商量，会看你有没有改，如果你改了，那么我就不该，他们之间非常友好，要比我们之前的那种互斥锁，要好的多，我们互斥锁是我才不跟你握手呢，我把这个资源呢牢牢地攥在我的手里，那后你也拿不到，所以呢我也可以保证安全，所以这是两种不同思路，

## CAS的三个操作数

内存值V、预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才将内存修改为B，否则什么都不做。最后返回现在的V值。

内存值指得是现在的这个值是多少，也就是说，我希望你是A，然后呢再去看看你现在V是不是A，如果是的话，满足我的期望，我就把他改成B，否则，也就意味着说如果A和B不相等，我就什么都不做，最后呢会返回我先的V。那这是CAS的一个基本的思想。

## 图解什么是CAS？



比如说我们现在是有两个线程，分别用两个CPU，它们都想改动56这个值，我们先来看CPU1，他去执行的时候，cas期望是56，并且想改成57，然后他去看是不是56，果然是56，于是呢他就可以改动成功，而当他改完之后。我们的CPU2他已经很慢了，他这会刚刚轮到他，他的CAS希望是55，可是这个时候他要么是56，要么是cpu已经改过的57，他肯定不是55了，所以这个时候虽然CPU2想把值从55改成56，但是它发现了你的这个值不是55，以他不会去真正的把这个55改成56，也就是说这个操作呢整个没有效果的，这个时候他没有修改成功嘛，所以是sorry，too late，他可以有其他的操作，比如说重试啊，或者是报错啊，这写都可以的，但是这一次CAS定是失败的。

## CPU的特殊指令

CAS实际上最终呢是要利用一些CPU的特殊指令，因为这些指令呢，首先一个，由CPU保证了他的原子性，其次一个指令就可以做几件事情，比如说这一个指令，可以让他先比较，然后呢再更新，而且这个指令由于有CPU保证了他的原子性，所以不会出现线程安全问题。

## CAS的等价代码(语义)

# 案例演示

## 两个线程竞争，其中一个落败

两个线程去修改信息其中一个或落败，其实就是后面去修改的回落败。

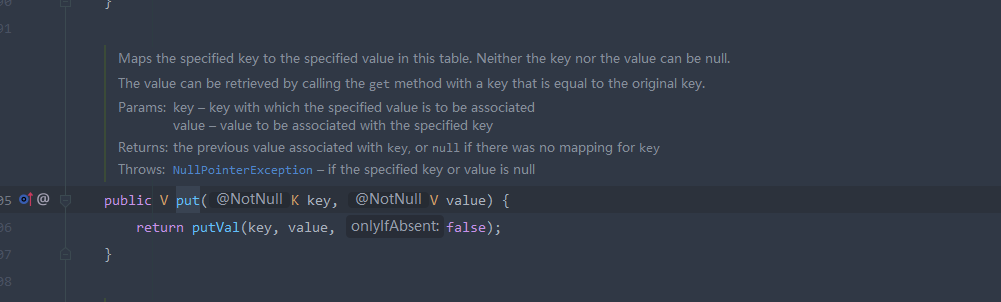
# 应用场景

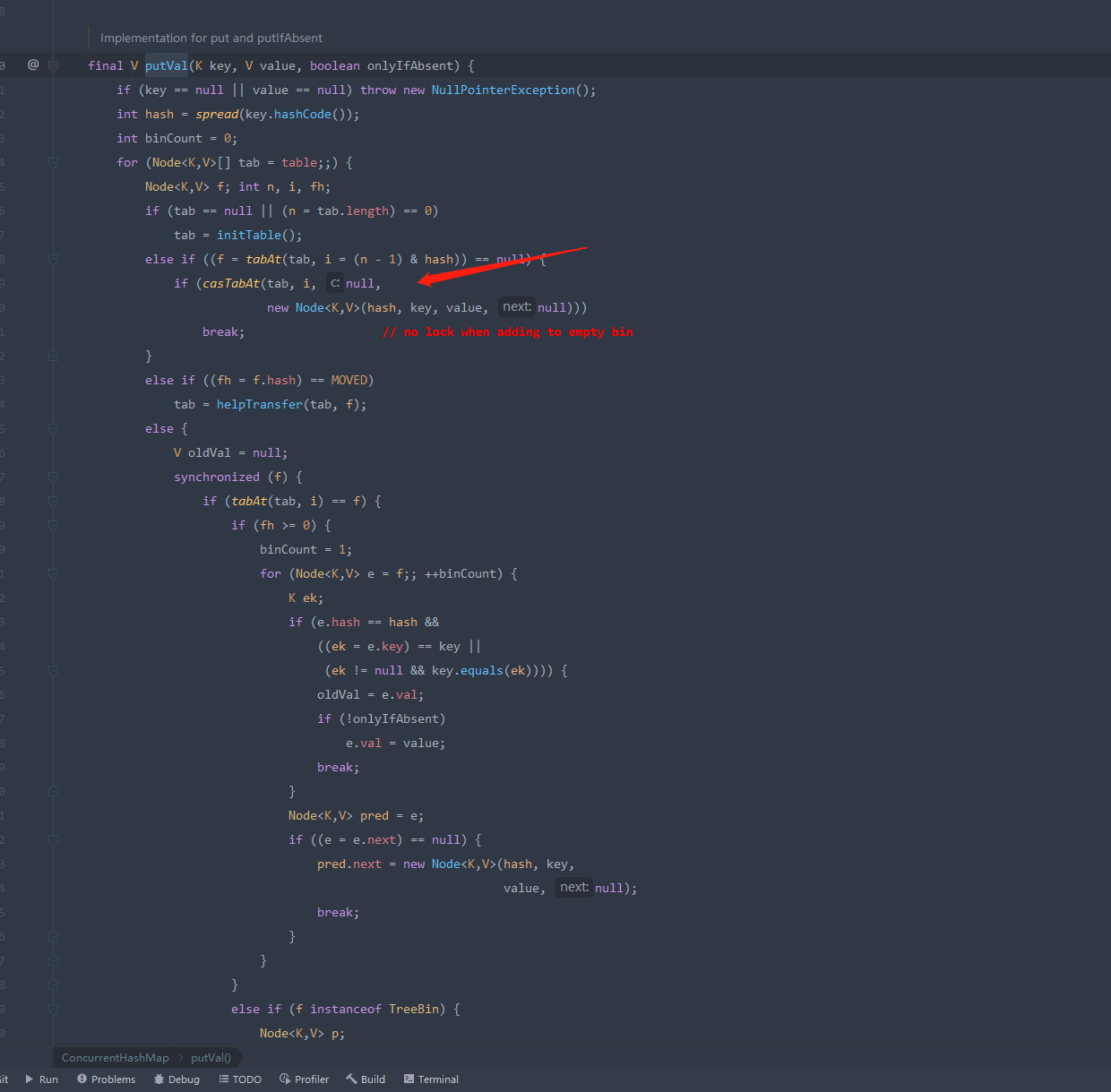
## 乐观锁

我们再讲乐观锁的时候我们知道他底层也是利用CAS实现的，比如说我们的数据库，他在修改的时候那么，它是可以用版本号的方式，去进行修改，而不需要锁住这个表，这样的思想正式CAS思想的体现，

## 2、并发容器

我们的并发容器中也有很多CAS的体现，比如说我们的concurrenthashmap，我们来看他的put().



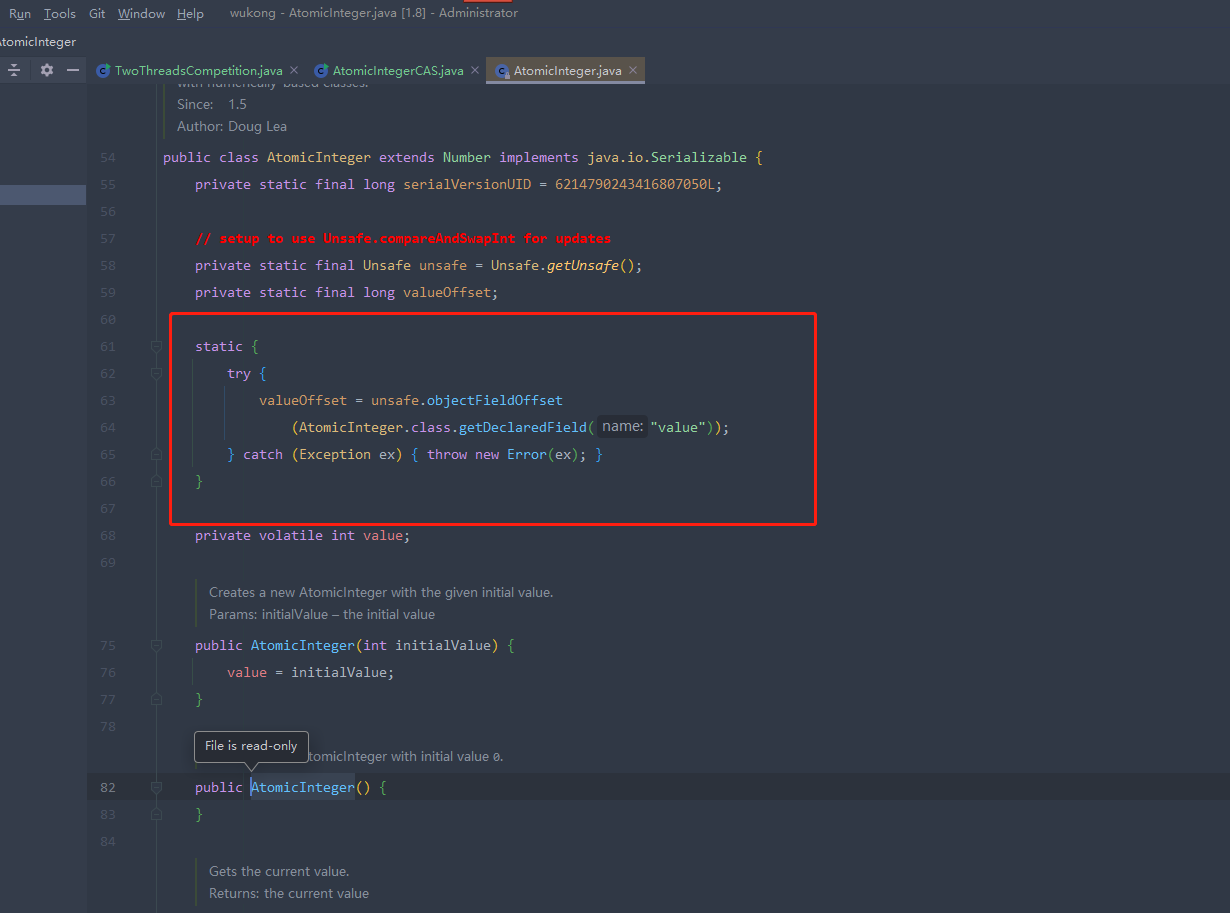


这里面非常明显有一个CAS 操作，他的方法名就叫做CAS，所以在我们的concurrenthashmap中，也有对这种思想的运用。

## 3、原子类

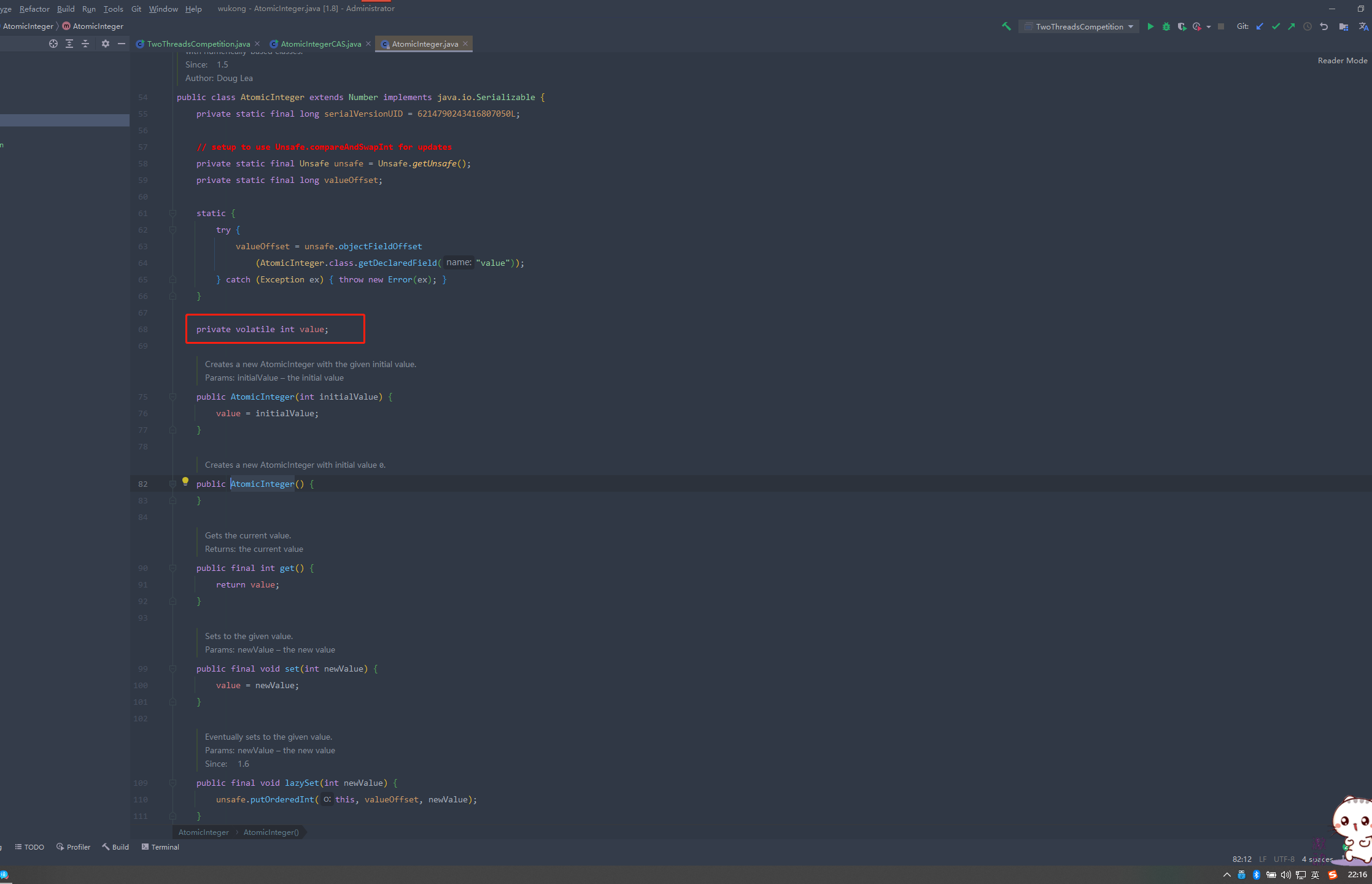
# 以AtomicInteger为例，分析在java中是如何利用CAS实现原子操作的？

## AtomicInteger加载Unsafe工具，用来直接操作内存数据



在这个AtomicInteger里面有一个初始静态代码块，所以这个代码块会在你使用AtomicInteger的时间就会执行。

在里面有一个值是valueOffset，而他正是利用我们的unSafe去拿到的，他拿到的是我们AtomicInteger这个类中value这个字段的地址，那我们的value字段在哪里呢？

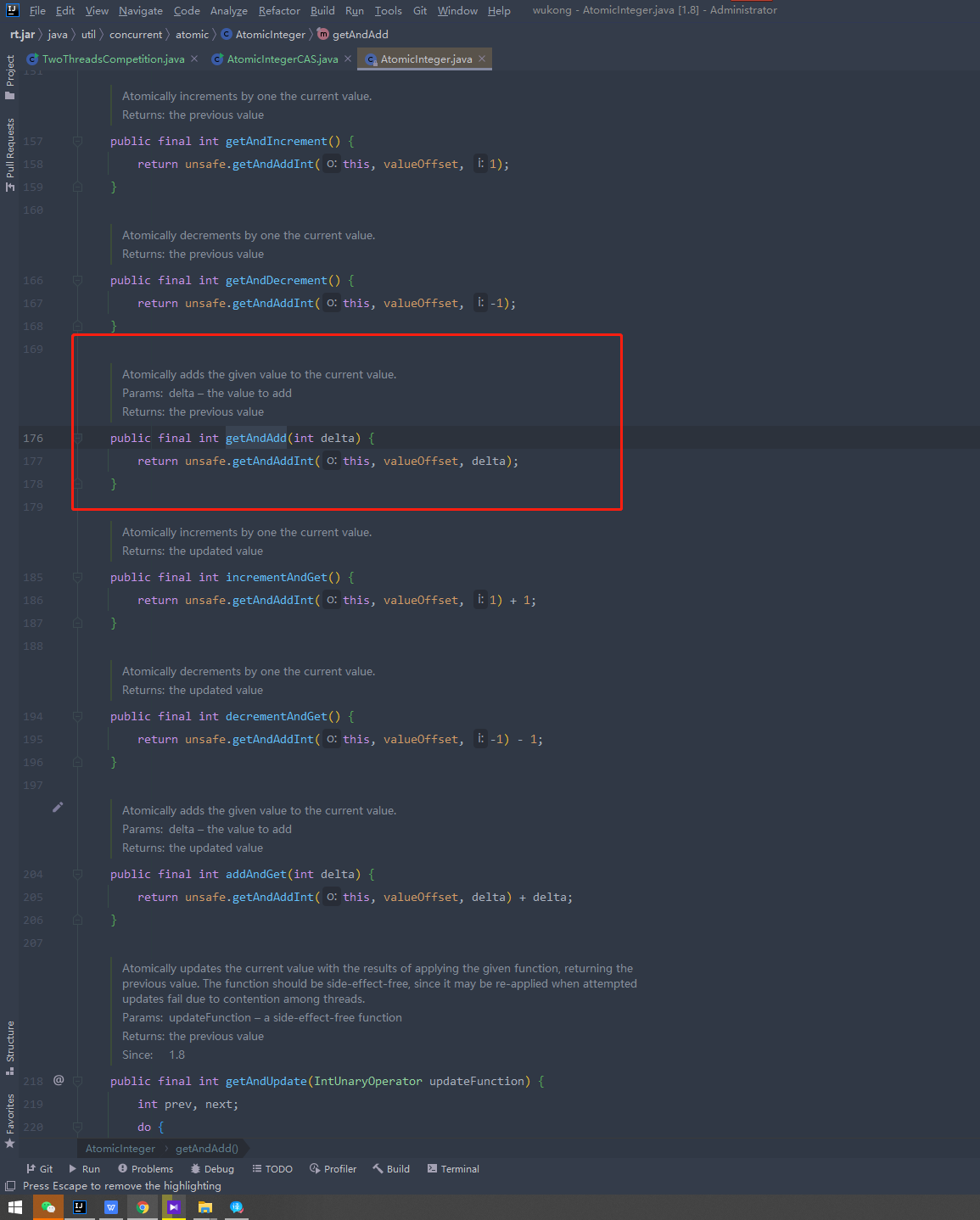


就在这个静态代码块的下面，那么这个value就是对应我们下面的第三点，也就是用volatile修饰的字段，保证可见性。

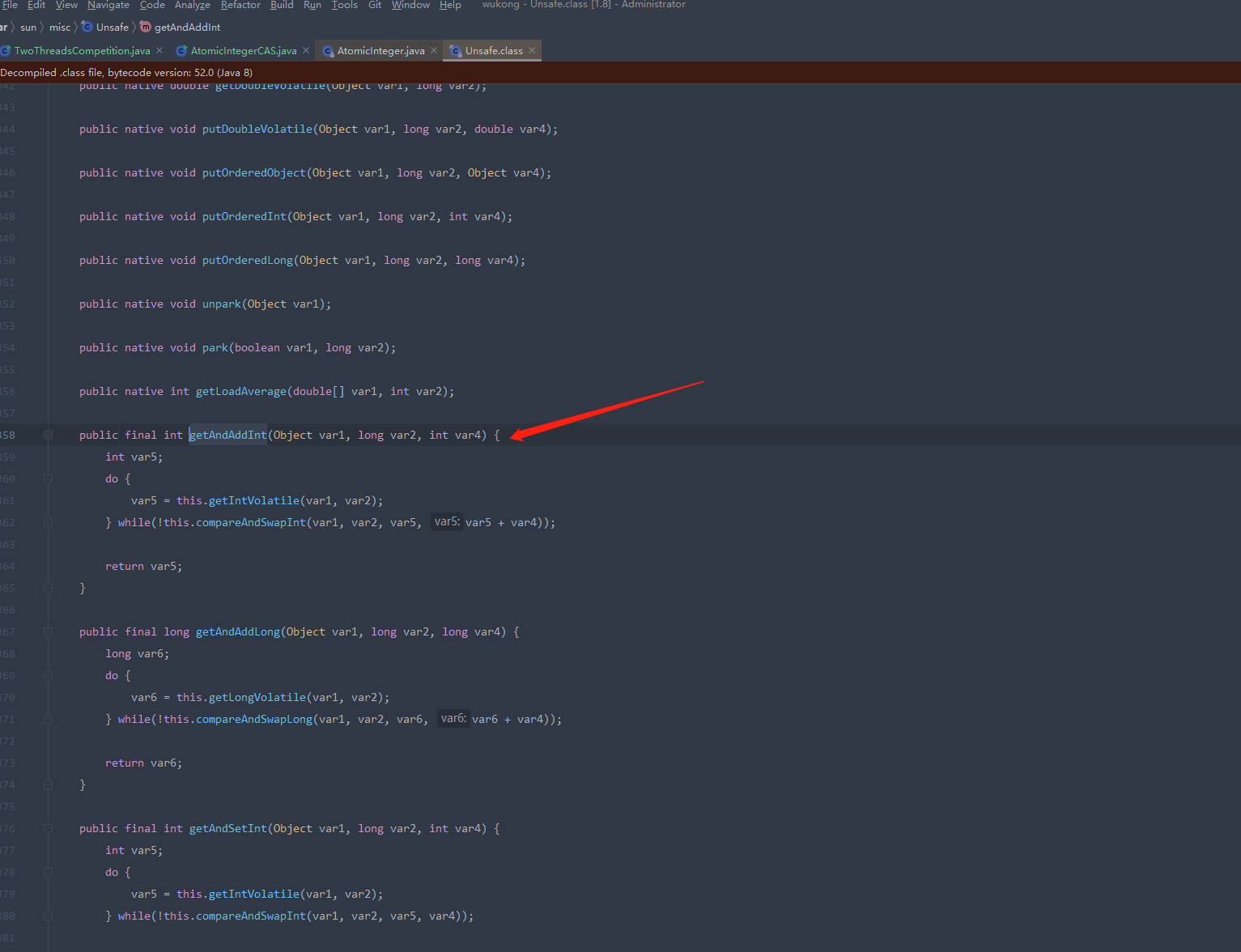
## 用Unsafe来实现底层操作

## 用volitile修饰value字段，保证可见性

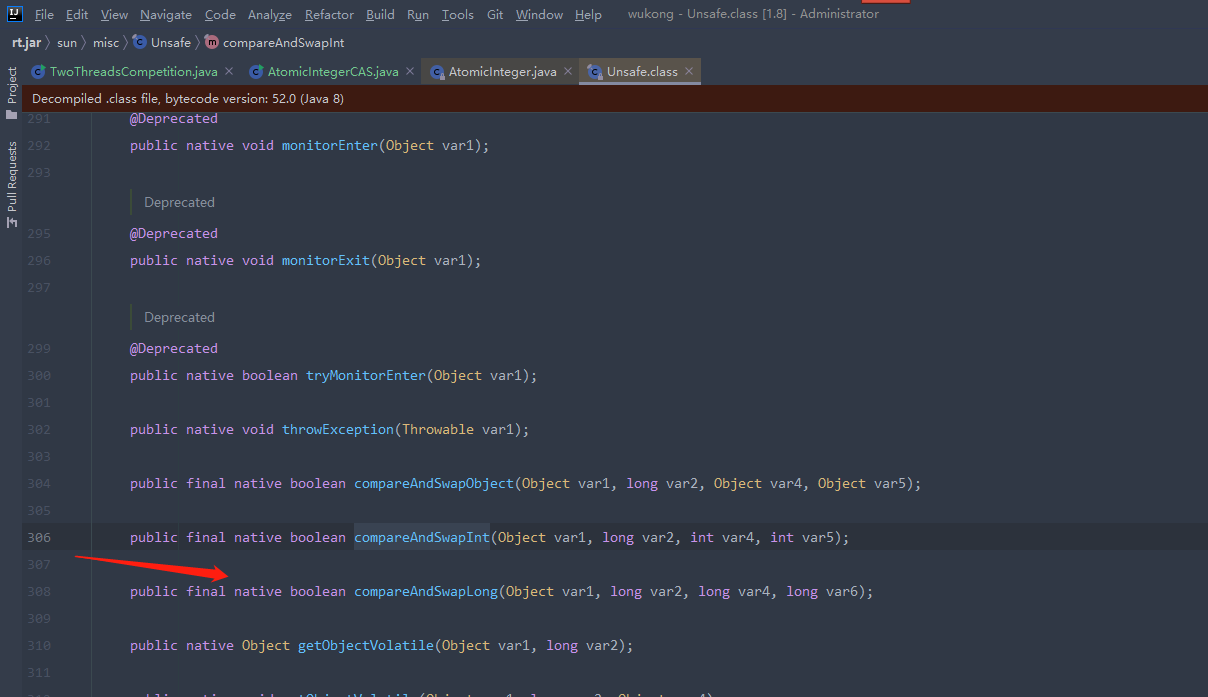
## getAndAddInt()方法分析



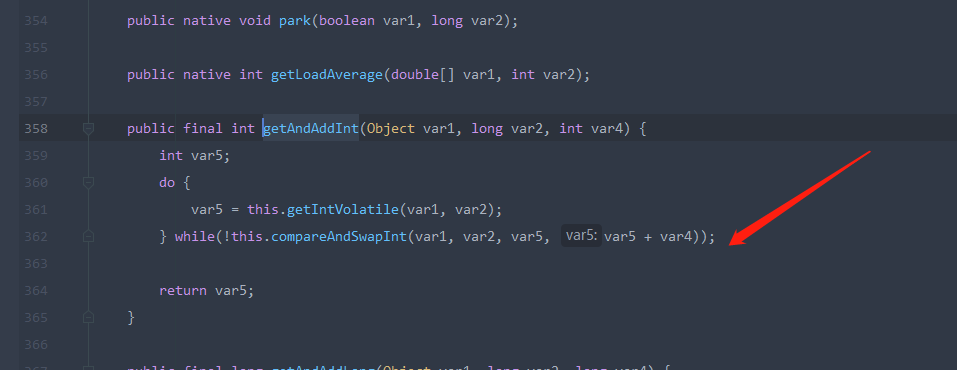
那这个getAndAdd下面调用的是unsafe，他的getAndAddInt()，我们来到这个方法。



这个类呢是Unsafe类，已经跳到Unsafe这个方法中去了，那我们看下这个代码，首先很明显的看到它是一个do..while..循环，那么do..while..循环呢，在这个while的判断条件里面，他就明显用了这个CompareAndSwapInt()，这正是我们CAS思想的体现，也就是CompareAndSwap，我们再来看这个CompareAndSwapInt()。



这个方法是一个native方法，我们等会去看下这个代码。

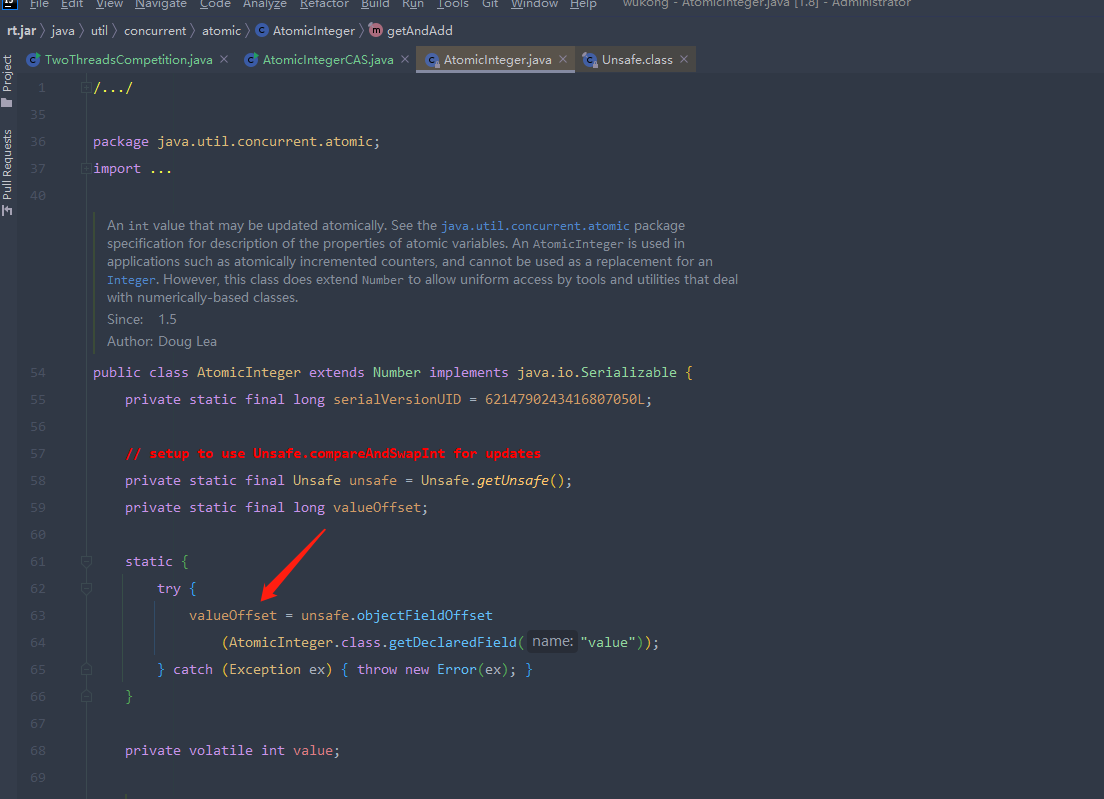


在这个循环中，如果说他这次的一个CAS失败的话，他就会进行重试，这个var5呢是我们最后将要返回的那个旧值，所以啊，while判断条件里面的CAS加上我们这边的一个自旋，do..while..是一个自旋，整个呢就形成了一个乐观锁，如果这次更新失败了，他也会自动的去进行重试。

## Unsafe类

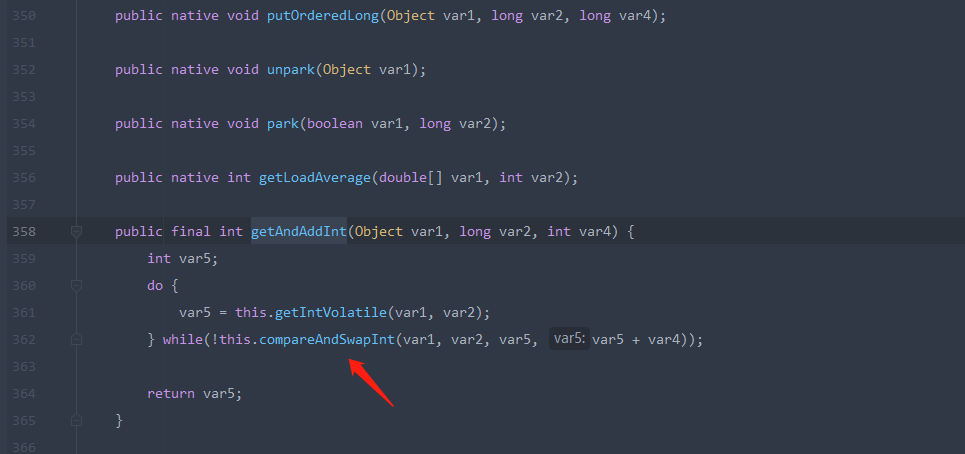
### Unsafe是CAS的核心类。Java无法直接访问底层操作系统，而是通过本地(native)方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，他提供了硬件级别的原子操作。

### valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址来获取数据的原值的，这样我们就能通过unsafe来实现CAS了。

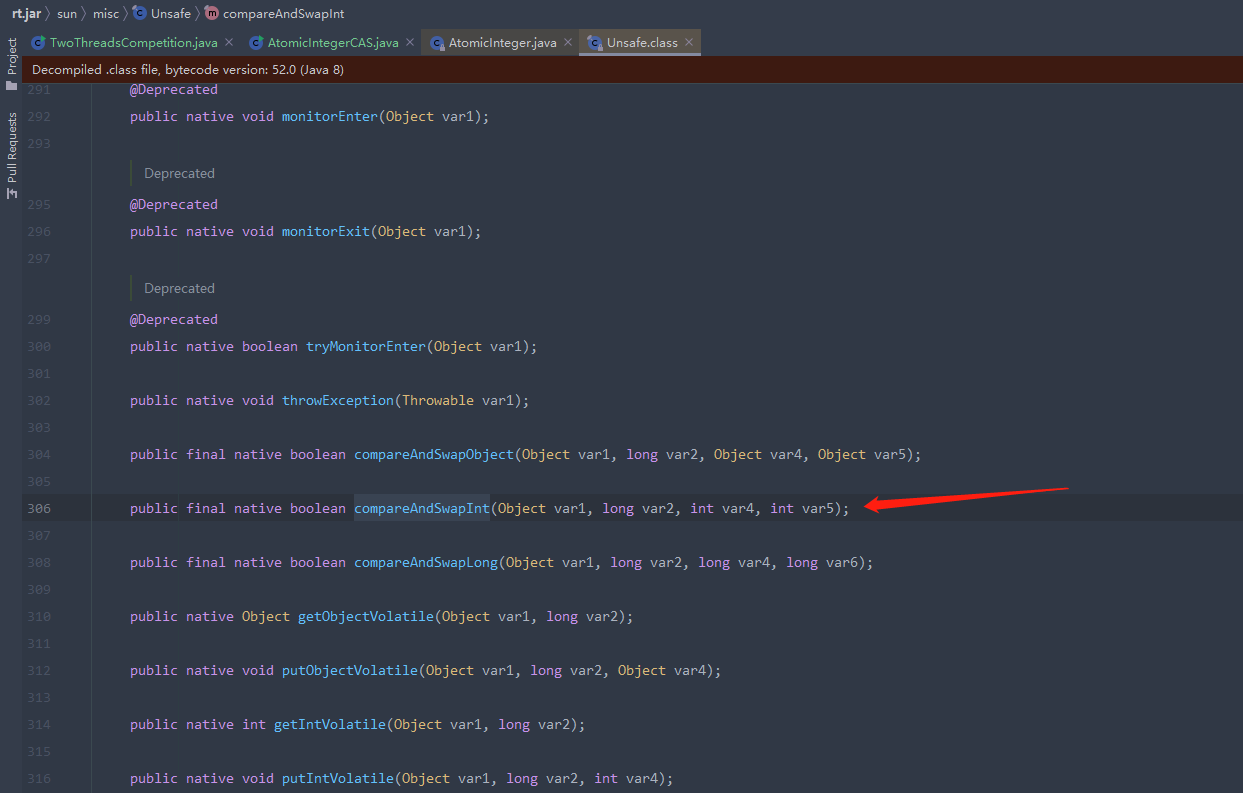


我们在AtomicInteger里面拿到一个valueOffset,它是什么意思呢？他就是我们的AtomicInteger类里面的”value”的地址，所以我们就有了这个变量在内存中的一个偏移地址，有了这个地址之后，我们就可以用Unsafe直接操作这个地址了，通过这个地址我们可以获取原值，也可以写入新值。

那么这里还有一个关键点就是，你看unsafe，看到哪里就截止了？看到这里



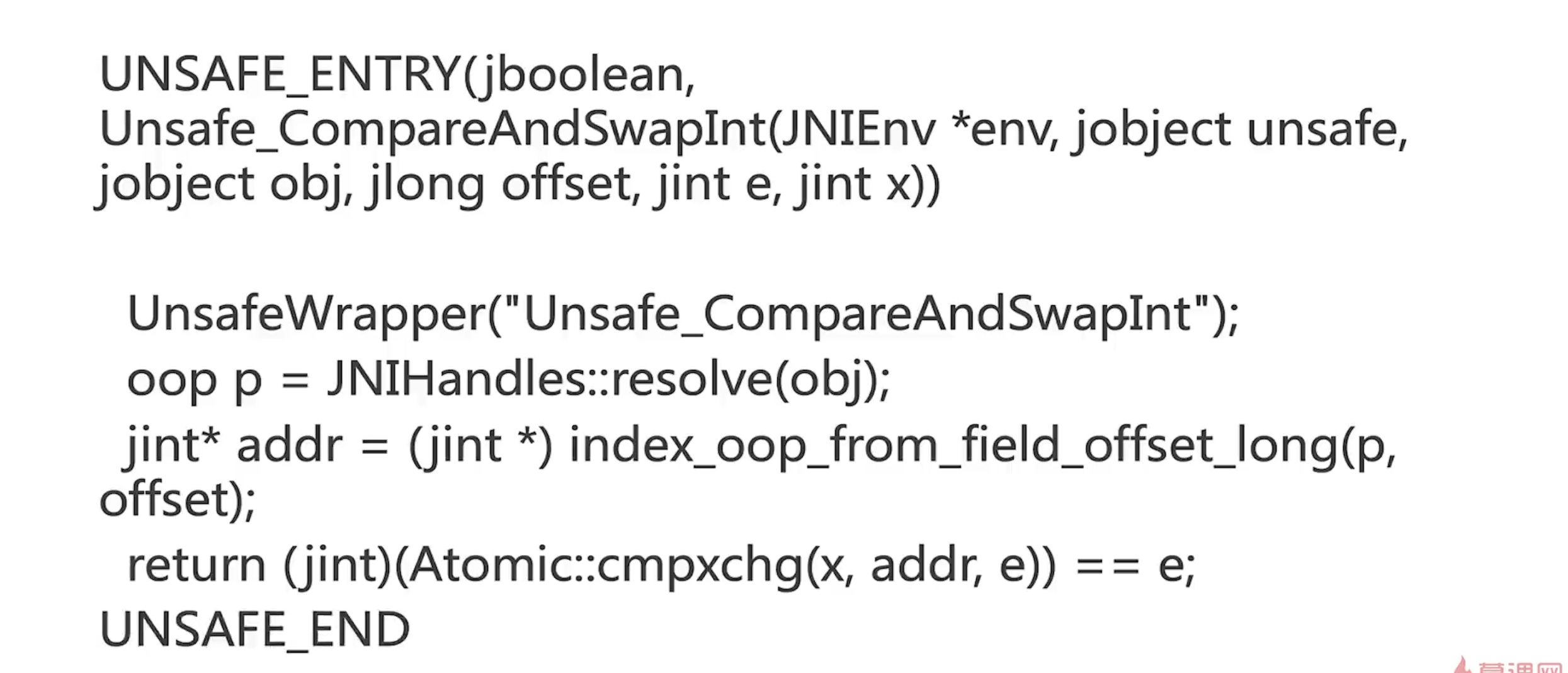
CompareAndSwapInt().



然后呢，你看到它是native的，那你这个native确实是优点难受啊，如果你还想进一步探索这个native做了什么事情的话，。

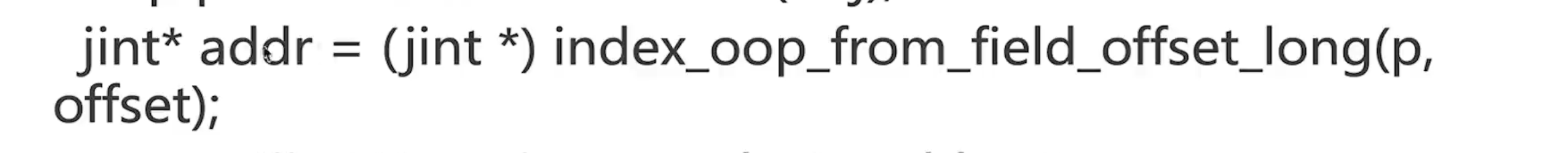
### Unsafe类中的compareAndSwapInt

Unsafe的这个compareAndSwapInt()，它是一个native的方法，他的实现呢在我们Unsafe.cpp中，这是一个C++代码，有关于如何去从java代码对应到C++代码这个过程，以及这个方法啊，我们在java并发精讲中，有一个彩蛋进行了介绍，那么在此呢我把这个C++代码贴出来，



这个就是他的代码，具体来讲，大家看可能会很困难，因为不仅是内容，包括语法可能我们都不熟悉，这毕竟不是同一个语言，我们来对它进行一下简单地介绍。

首先呢它是想办法拿到我们的内存地址



这一行语句中利用我们的偏移量offset拿到了地址，并且有了地址之后



利用了这行语句并且传入了我们的地址，去进行CAS操作，而其中最关键的应该是这里，



也就是Atomic::cmpxchg()，这个语句啊，实现了原子性操作的比较和替换，其中这里的X呢，这个X参数是即将更新的值，而这个E呢，使我们的原值，然后呢再加上他的这个addr内存地址，他就额已完成这个操作，如果还想进一步深究cmpxchg()方法是怎么实现的，那就对应到机器指令和汇编了，那我认为呢，我们没有必要去深入到如此底层，因为在平时开发中，我们深入到native实现，就已经足够了。

## 6、总结

### 1、Unsafe类中的compareAndSwapInt()方法。

在我们的AtomicInteger中，他首先是利用了Unsafe，这个类来帮助我们来操作层，尤其是利用了里面的compareAndSwapInt()方法。

在这个方法中呢我们先是想办法拿到在内存中的地址(方法中线想办法拿到变量value在内存中的地址)，有了地址之后，我们就可以通过Atomic::cmpxchg实现原子性的比较和替换，其中参数x是即将更新的值，参数e是原内存的值。至此，最终完成了CAS的全过程。

# 缺点

## ABA问题

首先在compareAndSwap的时候，它是先检查，但是它检查，他只是检查是不是和我这个值相等，它并不是检查再次期间是不是真的被修改过，那么我们举一个例子啊，比如说我拿到这个值，然后呢我发现他现在是5，OK，我就去进行计算了，那在我计算的这个过程中，有第二个线程把我改成了7，又有第三个线程把7又改回来了5，等到我第一个线程计算完毕，我去看它现在是不是5，我会发现真的是5，于是呢我就以为在此期间，没有任何线程去对它进行修改，所以啊，我就会所当然的把这个5改成我所期待的那个值，但实际上呢，此过程中已经有其他人把它修改过了，那么这样就会发生问题，比如说我就会认为没有其他人修改过那可能我接下来的一些操作，一些逻辑，包括一些计数，都会出现问题。

如何解决这个问题呢？ 我们可以沿用数据库的方式，用乐观锁的时候呢，我们添加一个版本号，这样一来啊，我们原来的那个路径，原来是5，变成7变成5，你可能由于最开始和最后都是5，可能看不清楚，但是我们加了版本号之后呢就会变成版本1，版本2，版本3，这样一来我们直接去对比这个版本号要比我们直接去对比这个值，来的更靠谱，所以我们可以用这样的思路来解决ABA问题。

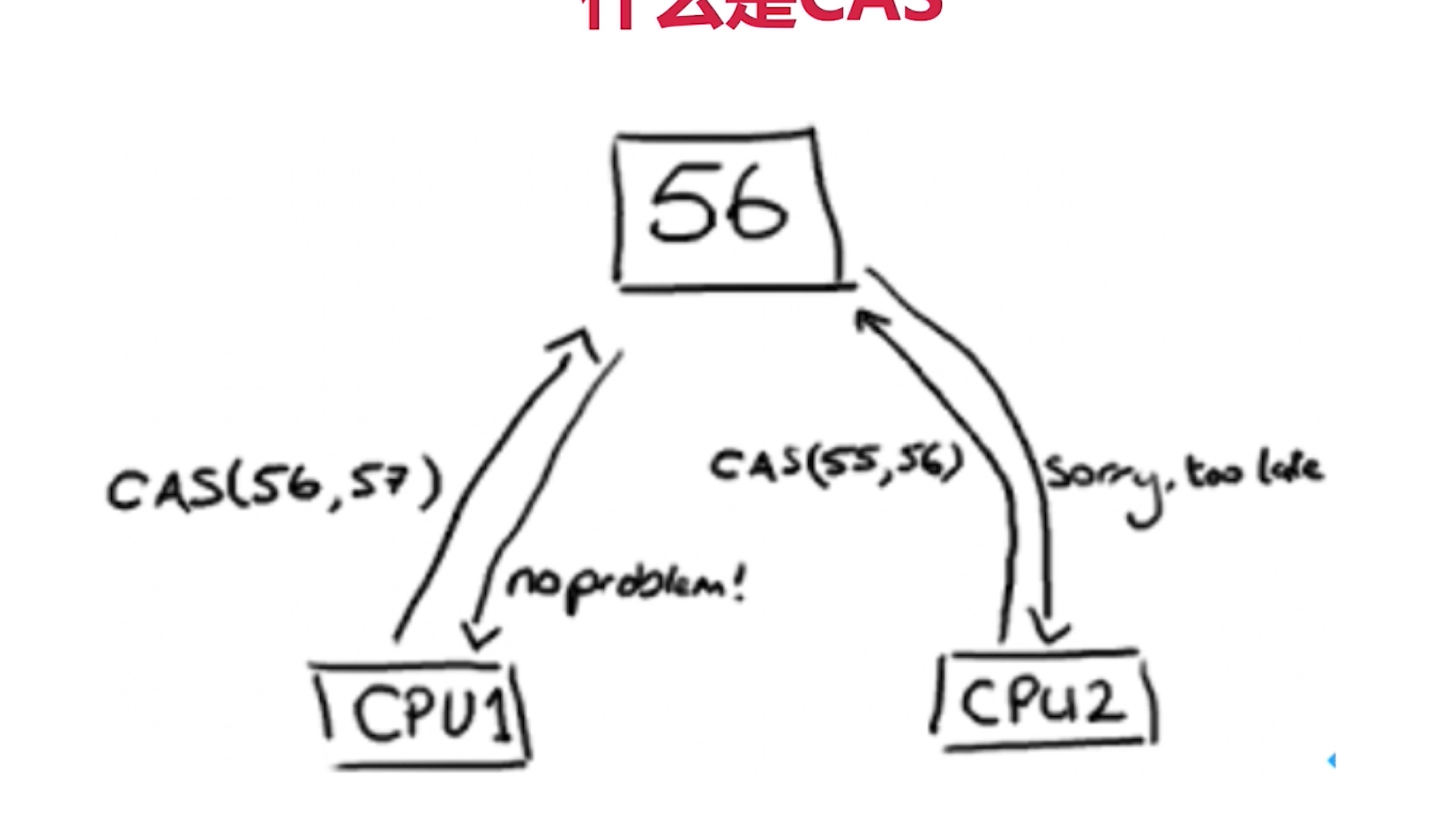
这个ABA，第一个A和第三个A实际上指的就是我们开始的5和最后的5，而这里的B呢就是我们刚才所说的7，这是CAS的明显的一个缺点。

## 自旋时间过长

我们看到肯能刚才有一些操作呢，它是死循环的，这种情况下，相当于是一个很长时间的一个自旋，那如果在此期间，他的竞争非常激烈，或者，锁他还一直都拿不到，那这个自旋啊它是消耗CPU的，他的性能是一种很大的消耗。所以这也是CAS的一个缺点。

# 6、CAS原理总结

## 1、什么是CAS



CAS啊实际上用这幅图是比较明显的看出来的，他就是先去比较，然后呢再去回过来操作。

## 2、案例演示

### 1、两个线程竞争，其中一个会落败

两个线程去竞争的时候呢，第一个线程成功了，第二个线程去比较的时候发现已经被改过了，所以第二个线程落败，同时我们也通过这个案例明白了CAS的等价代码，以及它内部的工作原理。

## 应用场景

### 乐观锁

### 并发容器

### 原子类

## 以AtomicInteger为例，分析在java中是如何利用CAS实现案子操作的？

我们以AtomicInteger为突破口，逐层分析了他的源代码，从最顶层的代码一直到Unsafe的代码，以及还看到了Unsafe他的native实现，明白了他是如何一步一步利用CAS实现原子操作的，当然最本质的是要CPU支持这样的指令，因为最后我们还是要操作CPU的，利用CPU的原子指令来保证他的原子性。

## 缺点

### ABA问题

### 自旋带来的性能消耗