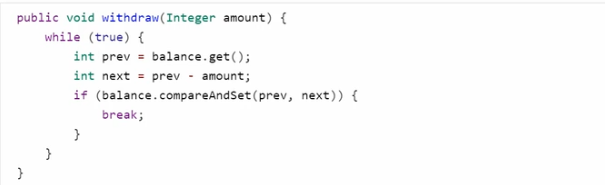
CAS

前面看到的 AtomicInteger 的解决方法，内部并没有用锁来保护共享变量的线程安全。那么它是如何实现的呢？

使用do...while

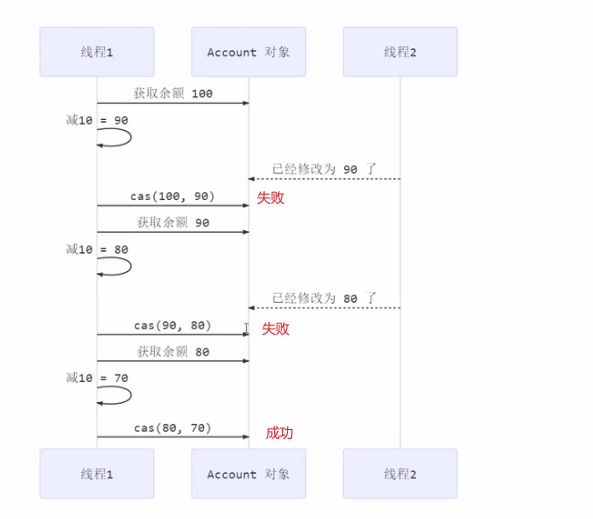
*/\*\*  
 \* 存在线程安全问题：  
 \* （1）使用synchronized解决  
 \* （2）使用ReentrantLock解决、  
 \* （3）使用CAS+volatile解决(无锁)  
 \*/***public void** cost(**int** x) {  
 **int** t;  
 **do** {  
 t = **balance**.get(); *//快照* } **while** (!**balance**.compareAndSet(t, t - x)); *//CAS*}

使用while



其中的关键是 compareAndSet，它的简称就是 CAS （也有 Compare And Swap 的说法），它必须是原子操作。

是一种CPU指令级的原子性。



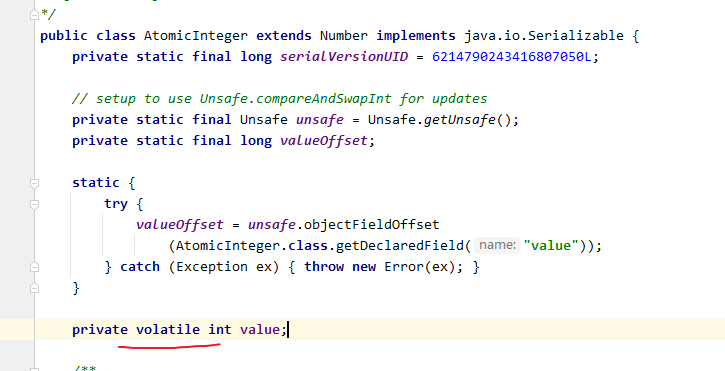
其实 CAS 的底层是 lock cmpxchg 指令（X86 架构），在单核 CPU 和多核 CPU 下都能够保证【比较-交换】的原子性。

在多核状态下，某个核执行到带 lock 的指令时，CPU 会让总线锁住，当这个核把此指令执行完毕，再开启总线。这个过程中不会被线程的调度机制所打断，保证了多个线程对内存操作的准确性，是原子的。

volatile

CAS操作需要volatile支持。

源码上可以看出：



获取共享变量时，为了保证该变量的可见性，需要使用 volatile 修饰。

它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。即一个线程对 volatile 变量的修改，对另一个线程可见。

volatile 仅仅保证了共享变量的可见性，让其它线程能够看到最新值，但不能解决指令交错问题（不能保证原子性）

CAS 必须借助 volatile 才能读取到共享变量的最新值来实现【比较并交换】的效果。

为什么无锁效率高

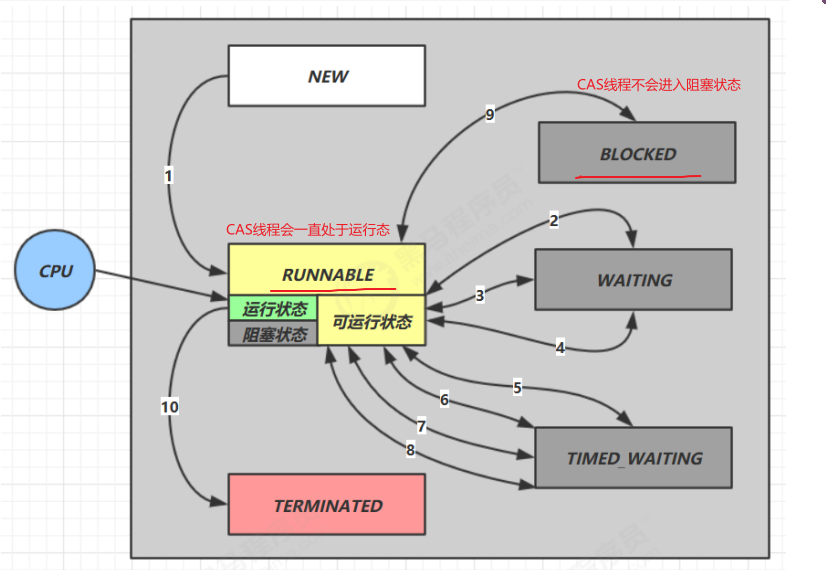
使用CAS的优点是吞吐量高。

缺点是需要频繁使用cpu，资源开销大。

无锁情况下，即使重试失败，线程始终在高速运行，没有停歇，而 synchronized 会让线程在没有获得锁的时候，发生上下文切换，进入阻塞。

打个比喻线程就好像高速跑道上的赛车，高速运行时，速度超快，一旦发生上下文切换，就好比赛车要减速、熄火，等被唤醒又得重新打火、启动、加速... 恢复到高速运行，代价比较大。

但无锁情况下，因为线程要保持运行，需要额外 CPU 的支持，CPU 在这里就好比高速跑道，没有额外的跑道，线程想高速运行也无从谈起，虽然不会进入阻塞，但由于没有分到时间片，仍然会进入可运行状态，还是会导致上下文切换。



CAS特点

结合 CAS 和 volatile 可以实现无锁并发，适用于线程数少、多核 CPU 的场景下。

当线程数大于CPU的最大核心数，CAS效率不会提升。

* CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
* synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

CAS 体现的是无锁并发（不会加锁，使用while循环）、无阻塞并发（CAS线程会一直运行，不会进入阻塞态发生上下文切换），请仔细体会这两句话的意思。

因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一。

但如果竞争激烈，可以想到重试必然频繁发生，反而效率会受影响。