# 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）

基于CAS实现

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）

Synchronize 、lock 、readWriteLock

## 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。

当多线程操作同一变量之前 ，由于共享变量（V）时在主内存的 会先各自拷贝一分到自己的内存（E） ， 然后拿 E 和 V 就进行比较 ，如果 V=E 则将V的值=N ， 然后刷新到主内存。 当另一个线程 修改时 也拿刚才 拷贝的 E 和当前主内存的值进行比较 ， E!=V 。 则说明值已经修改过了 。此时不进行操作 而是当前线程 自旋 。重复之前的 先读取主内存数据 ，比较再修改值的操作

自旋的作用 ： 并不能保证第一次修改 就能成功 ，虽然自旋是while循环 ，但也不会总成死锁 。Jdk 如果自旋多次不能修改成功 则 自动终止

并发下线程在操作锁时 有两种状态 ：

A、阻塞

B、不阻塞

1)自旋 ： 自旋多次失败 ，终止

1. 适应性自旋 ：获取 锁 自旋次数， 来判断下一次自旋的时候获取锁的概率有多大

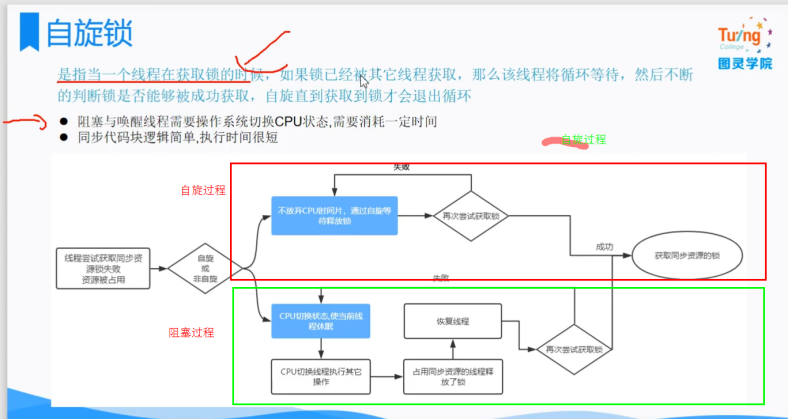
然后 自适应调整 自旋的 次数（包括上调次数 和 下调次数）

(1)线程在在未争抢到锁 时的状态

1)阻塞 (性能低)

2)自旋 (性能高)

(2)自旋锁

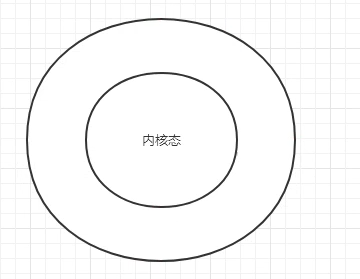
、

自旋的特点 ： 不释放cpu资源 ，只是一直循环尝试获取锁 ，当获取不到锁 循环到一定次数， 就不在争抢 （1.7后 出现自适应 自旋锁进行 优化）

阻塞的特点 ： 直接通过cpu 切换线程状态 ，获取不到锁的 线程 则强制休眠 ， 获取到锁的 线程 执行 ，由于 cpu 只有一个 并且在执行线程时 要来回切换 所有性能会很低

1. 阻塞 性能低的原因 ： 涉及到cpu 切换线程 上下文

线程 运行 存储空间 ： 内核态 ，外层是用户态



线程切换的过程 ： 就是 用户态 和 内核态 之间数据的切换过程

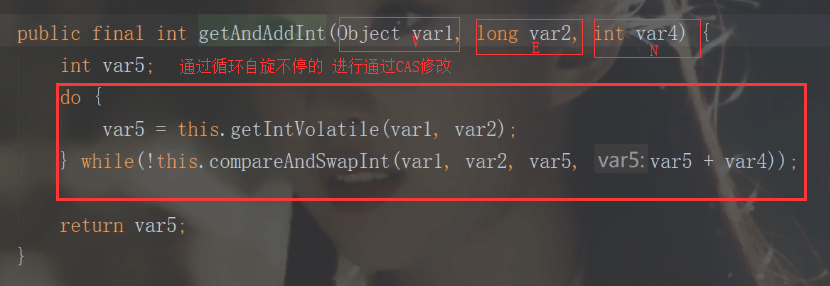
如果涉及到 线程 上下文切换的时候 ， 需要将 待运行 线程的用户态（用户态：阻塞的线程中的运行结果或涉及到的信息保存到PCB内存）的信息 拷贝的 到内核态（内核态：线程内部的中间变量、数据运行结果、寄存器中的运行结果、线程栈中保存的指令）

然后才能 运行

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。

Jdk中 很多原子类在java.util.current.atomic包下 如： AutomicInteger 都是基于CAS实现的

atomicIntrnger 的getAndAddInt方法

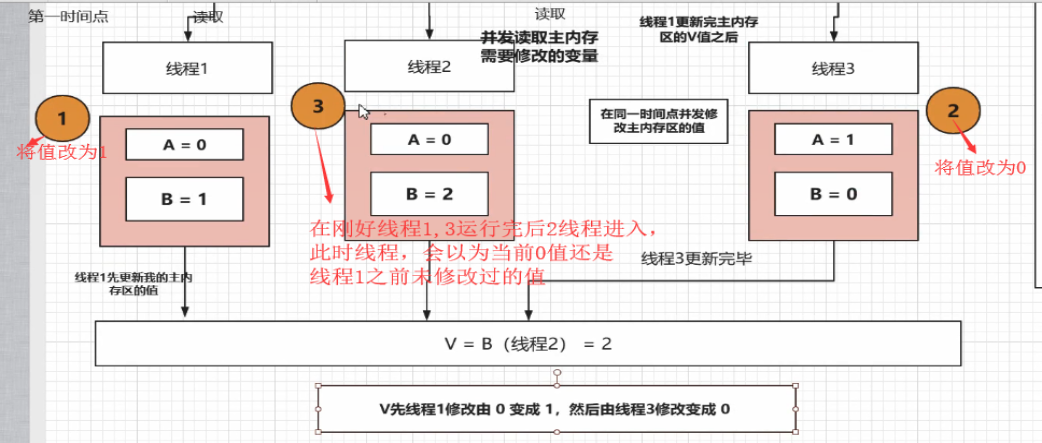


**优点 ：** 性能较高

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

缺点：

1. ABA问题



最终的错觉结果就是 线程3 没有操作 。 只是线程1 和 线程2 运行的结果

解决 ：在jdk1.6后 给操作的变量前加个版本号 AtomicStampRefrence类，每次更新变量都会加1

1. 循环时间长 ，开销大
2. 只能保证 一个 共享变量的 原子操作 AtomicStampRefrence类可以把多个变量放在一个对象里进行cas操作

## 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。

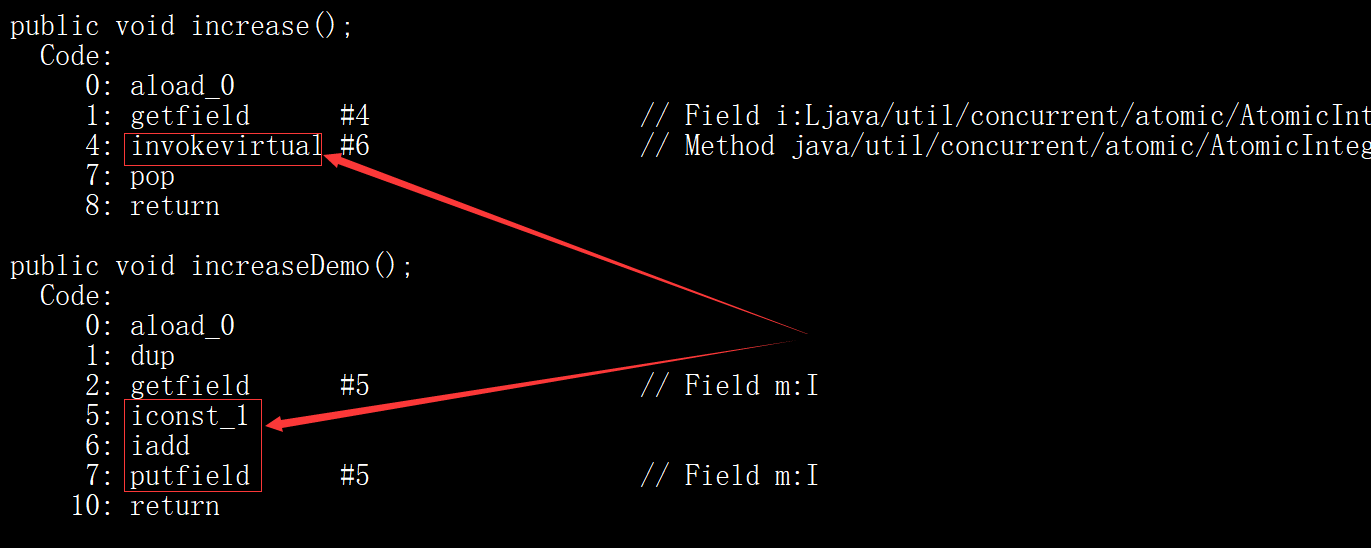
1. 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用总线锁定。

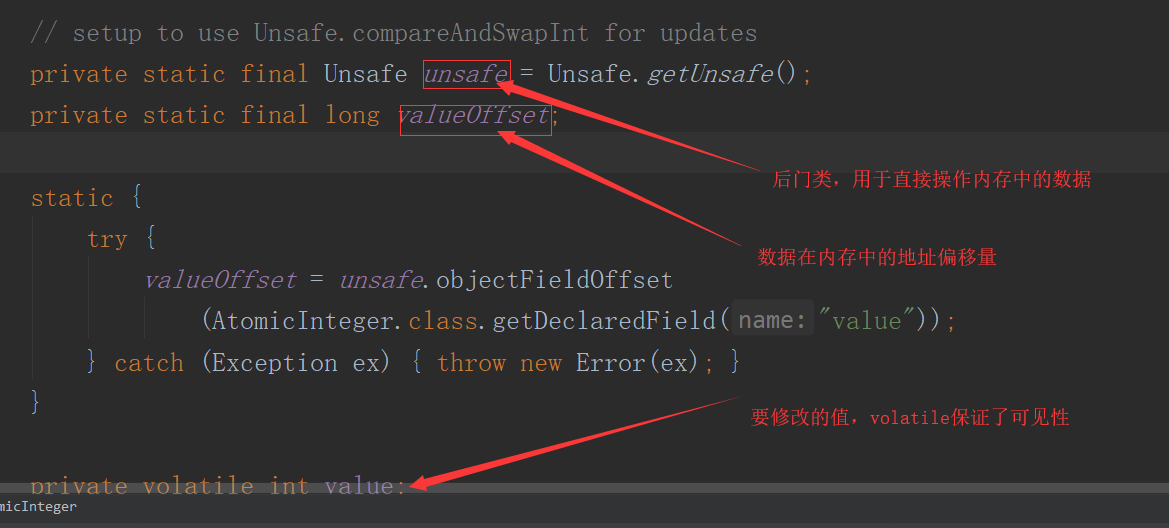
2. 有些处理器不支持缓存锁定，对于 Intel 486 和 Pentium 处理器，就是锁定的内存区域在处理器的缓存行也会调用总线锁定。

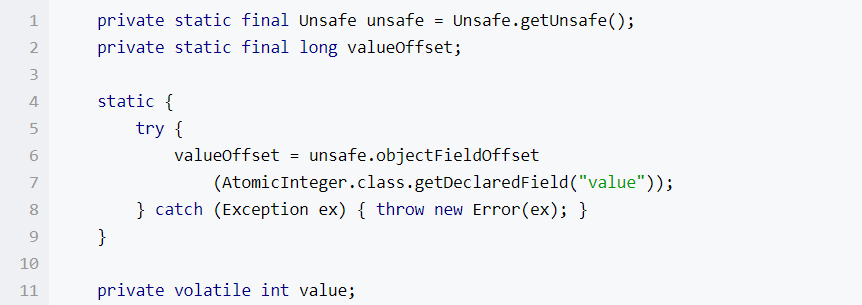
## CAS举例

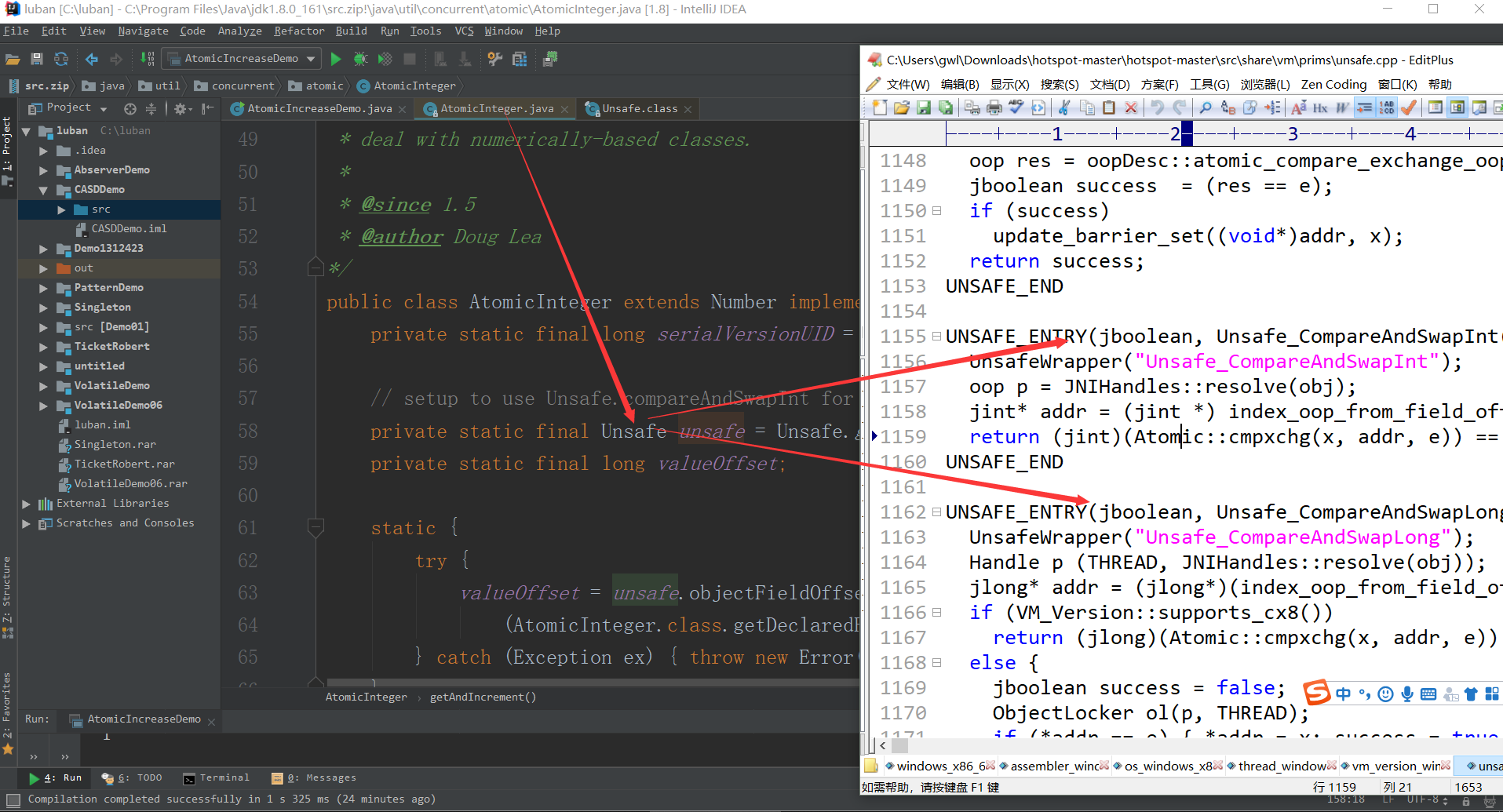
## CAS源码分析

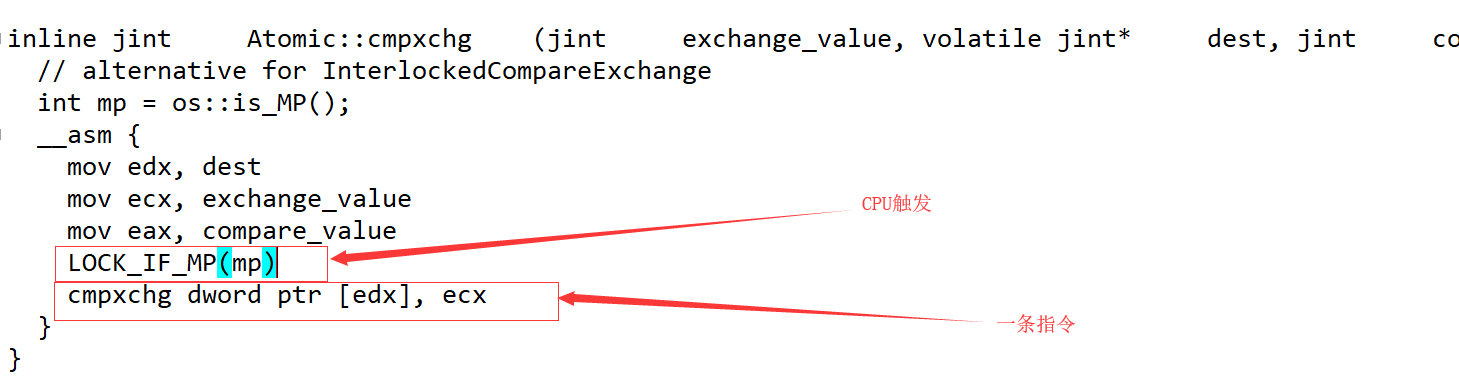
JUC下的atomic类都是通过CAS来实现的，下面就以AtomicInteger为例来阐述CAS的实现。如下：



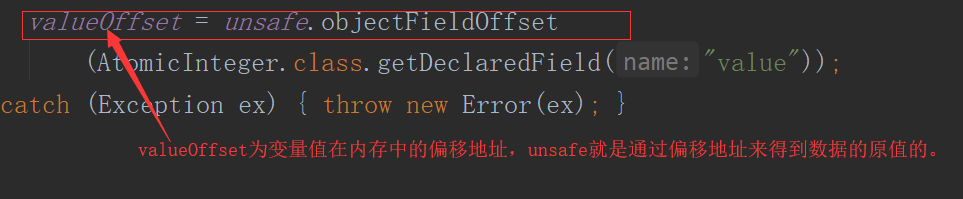


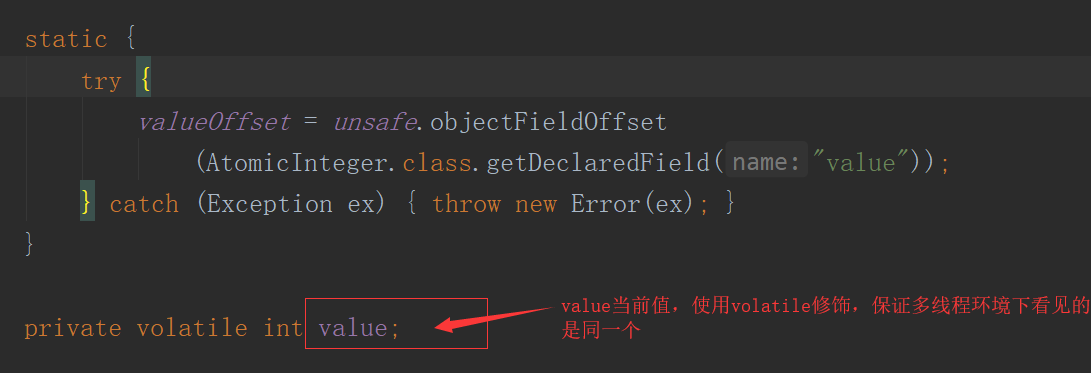






Unsafe是CAS的核心类，Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门：Unsafe，它提供了硬件级别的原子操作。





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. Unsafe

Java jdk中的对象调用 .getClass.getClassLoader()返回null的问题

Jdk中的 rt.jar 、 dt.jar

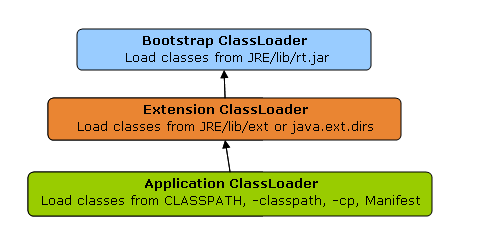
rt.jar 在 java/bin/中 是JAVA基础类库，也就是 java文档中所有的类的class文件  
dt.jar在 java/bin/中是关于运行环境的类库

java自带三种类加载器

A、BootstrapClassLoader启动类：是JVM启动的时候负责加载jdk 中 jre/lib/rt.jar . C++写的 不是java类

B、ExtentionClassLoader 扩展类加载器： java编写 扩展了bootStrap 主要加载 java/lib/ext 目录下 calss文件 。 如果自定义类加载器的话就是 实现这个类

C、AppClassLoade系统类加载器: 负责加载 项目中 自己的 classpath目录下calss文件

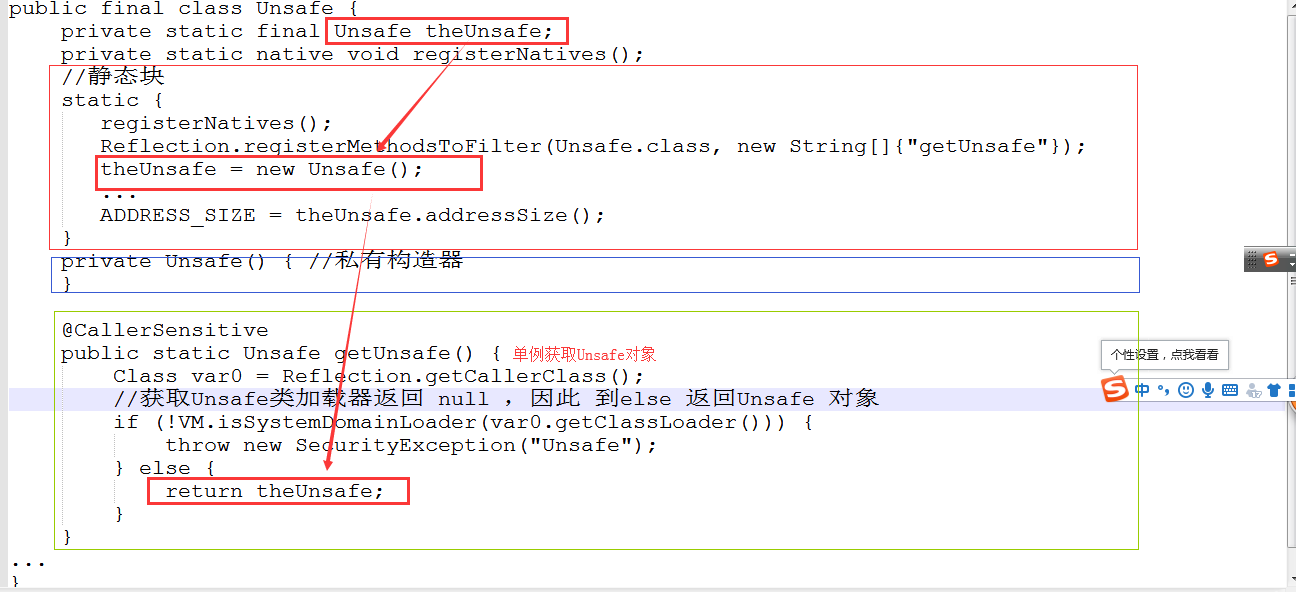


注意 ： 他们是自上而下的 继承关系， 也是是双亲委派模式 。 双亲委派模式：底层的加载器 在加载class文件的时候首先会到 调用父类中去查找相应的class文件 ，如果没则 自己查找并返回 。

总结 ： 如果当 java rt.jar中的 jdk 对象 调用 getClass.getClassLoader() 获取类加载器时返回 null (rt.jar 的类加载是bootStrap ，而他不是java 自然就返回null)

java不能直接访问操作系统底层，而是通过本地方法来访问。Unsafe类提供了硬件级别的原子操作（基本都是 native方法） 包括 数组操作、对象操作、CAS操作、线程park操作、另外也是 juc 并发包的底层基础实现。

获取 Unsafe ：



注意 ： 只有在启动类下 调用Unsafe.getUnsafe() 能够创建unsafe对象

if (!VM.isSystemDomainLoader(var0.getClassLoader())) {

throw new SecurityException("Unsafe");

} else {

return theUnsafe;

}

因为 自己的写的类获取获取的classLoader 不是Unsafe的类加载器返回false

如果想使用必须通过反射：

Field f = Unsafe.class.getDeclaredField("theUnsafe");

f.setAccessible(true);

Unsafe unsafe = (Unsafe) f.get(null);

Unsafe主要提供了以下功能 :

（1）Info相关。主要返回某些低级别的内存信息：addressSize(), pageSize()

（2）Objects相关。主要提供Object和它的域操纵方法：allocateInstance(),objectFieldOffset()

（3）Class相关。主要提供Class和它的静态域操纵方法：staticFieldOffset(),defineClass(),defineAnonymousClass(),ensureClassInitialized()

（4）Arrays相关。数组操纵方法：arrayBaseOffset(),arrayIndexScale()

（5）Synchronization相关。主要提供低级别同步原语（如基于CPU的CAS（Compare-And-Swap）原语）：monitorEnter(),tryMonitorEnter(),monitorExit(),compareAndSwapInt(),putOrderedInt()

1. Memory相关。直接内存访问方法（绕过JVM堆直接操纵本地内存）：allocateMemory(),copyMemory(),freeMemory(),getAddress(),getInt(),putInt()
2. AQS ：

<https://www.cnblogs.com/waterystone/p/4920797.html>

1、AQS: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

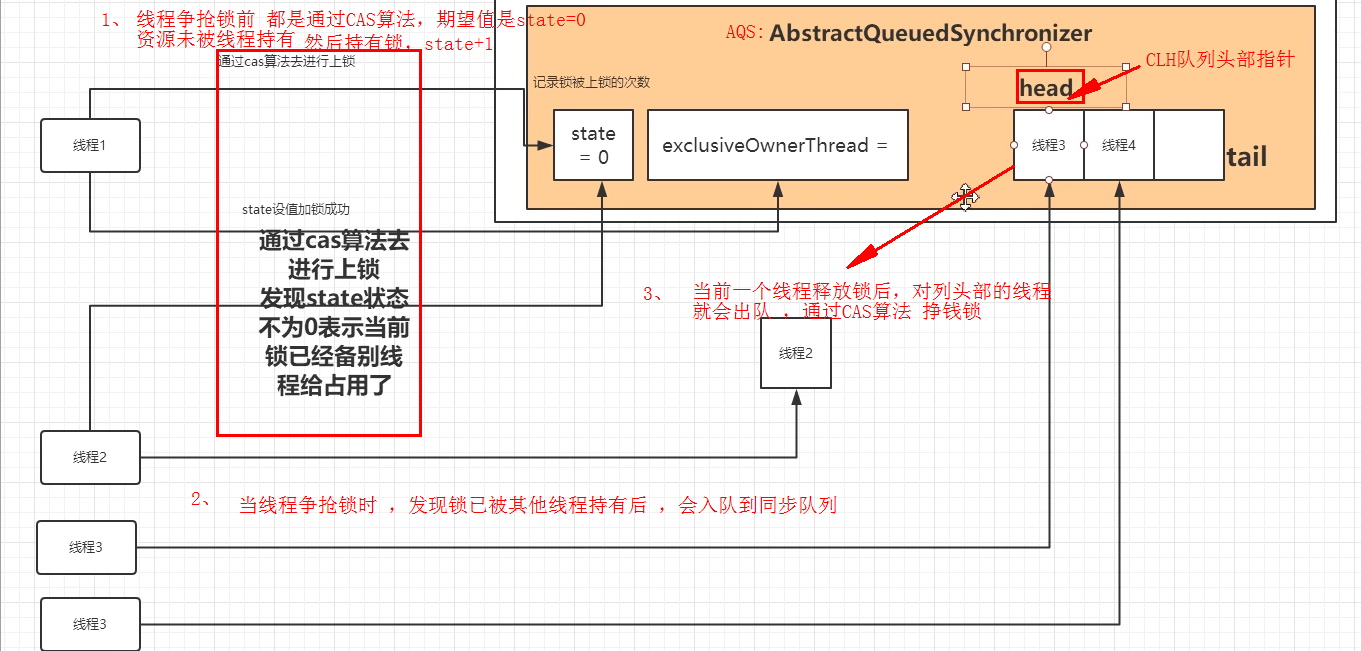
AbstractQueuedSynchronizer 类的三个成员：

A、State 被加锁的次数 （State =0 表示该资源没被线程持有）

B、eclusiveOwnerThread ：当前持有锁的线程

C、clh同步队列 ： 用于存放 线程 （根据线程争抢资源的顺序 一次从队首放到队尾）

具体 步骤 ：



1. State 是 volitil 修饰的
2. 线程在 获取锁时 通过getState() 获取当前锁的状态 ，如果state=0 则占有锁 ，如果 未获取到锁 则放到队列 调用tryAcquire自旋



总结 ：

1/AQS 包含

Node(CLH队列即同步队列：储存着所有等待锁的线程)

独占锁

共享锁

Condition条件

2/acquireQueue() 方法获取锁 的步骤

**final boolean** acquireQueued(**final** Node node, **int** arg) {  
 **boolean** failed = **true**;  
 **try** {  
 **boolean** interrupted = **false**;  
 **for** (;;) { //自旋  
 **final** Node p = node.predecessor(); //获取前一个节点，判断是否是head  
 **if** (p == **head** && tryAcquire(arg)) { //并且获取到锁  
 setHead(node);  
 p.**next** = **null**; *// help GC* failed = **false**;  
 **return** interrupted;  
 }  
 **if** (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = **true**;  
 }  
 } **finally** {  
 **if** (failed)  
 cancelAcquire(node);  
 }  
}

1. 通过 循环自旋
2. 尝试获取node节点 ，的前一个节点的前一个 node节点 ，然后调用tryAcquire 尝试获取锁 ，如果成功 就将当前node设置成CLH的头节点
3. 然后调用shouldParkAfterFailedAcquire方法，来决定是否要阻塞当前线程，如果是 则调用 parkAndCheckInterrupt 阻塞当前线程
4. 如果当前线程发生异常，非正常退出，那么会在finally模块中调用cancelAcquire(node)方法，取消当前节点状态。

总结 : AQS 是 锁的 基础 ， 一个锁类 必须先 一个AQS 内部类 来实现锁机制