# 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 但是会有一个 ABA的问题， 假如V 被其他线程改了 然后又改了回来 ， 此时 对于当前线程来说 v 还是没有变化的 。 在 jdk 1.7之后通过在修改时 加版本号 解决了，这个问题

Unsafe 实现 (本地方法 直接跟硬件打交道)

本质就是 调用 Unsafe 的getAndAddInt() 方法

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

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

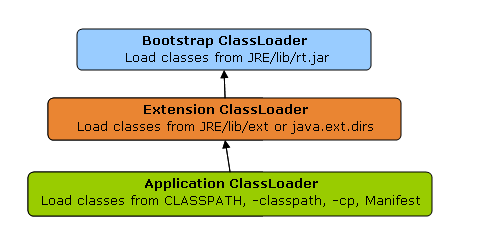
rt.jar 在 java/bin/中 是JAVA基础类库，也就是 java文档中所有的类的class文件  
dt.jar在 java/bin/中是关于运行环境的类库

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

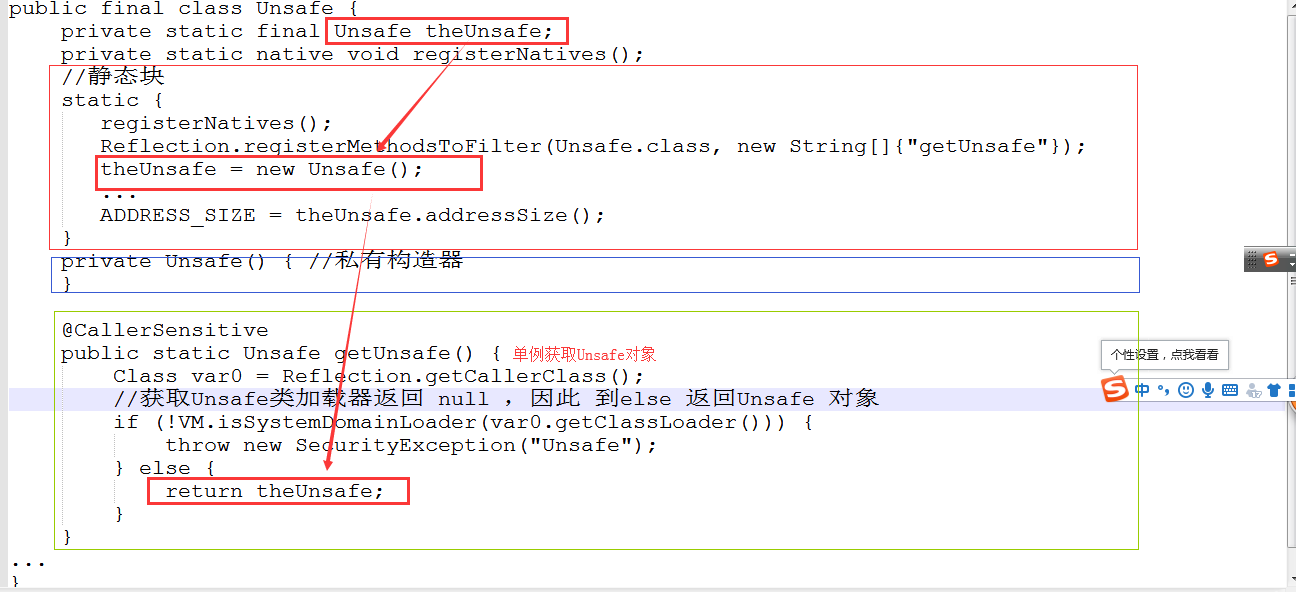


注意 ： 他们是自上而下的 继承关系， 也是是双亲委派模式 。 双亲委派模式：底层的加载器 在加载class文件的时候首先会到 调用父类中去查找相应的class文件 ，如果没则 自己查找并返回 。

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、unPark、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：

**private transient volatile** Node **head**; 队首一直为 null  
 **private transient volatile** Node **tail**; 队尾

**private volatile int state**; 锁的状态

4、Node线程都被看成一个个node ，其中node三个属性 用于标识当前node的状态

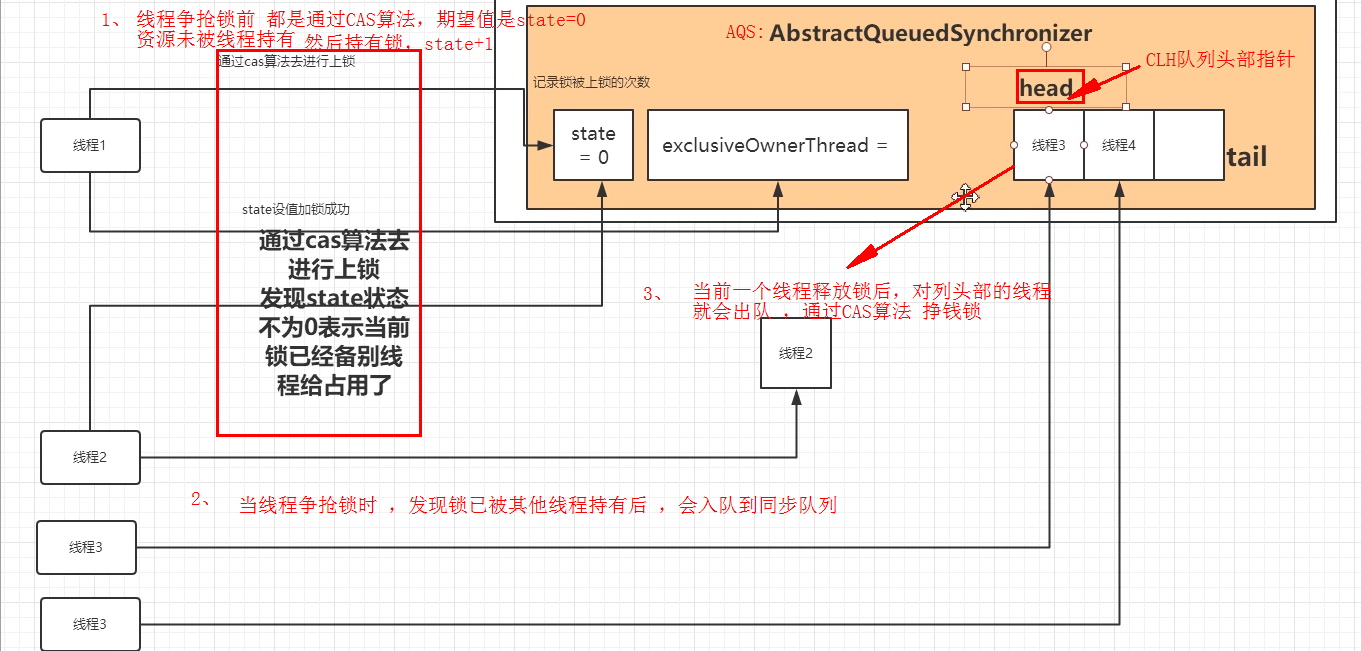
（1）**volatile** Node **prev**; 前一个node节点

（2）**volatile** Node **next**; 后一个node节点

（3）**volatile** Thread **thread**; 当前节点中的 线程

（4）Node **nextWaiter**; 链接等待状态下的节点

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 尝试获取node节点 的前一个节点的前一个 node节点
2. 判断获取到的节点是否是头节点 ，如果是然后调用tryAcquire 尝试获取锁 ，如果成功 就将当前node设置成CLH的头节点 ， 下一个节点 设置为null 然后return；
3. 如果不是头结点，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