Java中CAS-比较并交换

# CAS的引入

在JDK 5之前Java语言是靠**synchronized关键字**保证同步的，锁机制存在以下问题：

（1）在多线程竞争下，加锁、释放锁会导致比较多的**上下文切换和调度延时**，引起性能问题。

（2）一个线程持有锁会导致其它所有需要此锁的线程挂起。

（3）如果一个优先级高的线程等待一个优先级低的线程释放锁会导致**优先级倒置**，**引起性能风险**。

**volatile**是不错的机制，但是volatile不能保证原子性。因此对于同步最终还是要回到锁机制上来。

独占锁是一种悲观锁，**synchronized就是一种独占锁**，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。**而另一个更加有效的锁就是乐观锁**。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**乐观锁用到的机制就是CAS，Compare and Swap**。

# 什么是CAS？

**CAS,Compare And Swap**的缩写，中文翻译成**比较并交换**。

**java.util.concurrent包**中借助CAS实现了区别于**synchronouse同步锁**的一种乐观锁。

我们都知道，在java语言之前，并发就已经广泛存在并在服务器领域得到了大量的应用。所以硬件厂商老早就在芯片中加入了大量直至并发操作的原语，从而在硬件层面提升效率。在intel的CPU中，使用**cmpxchg**指令。

在Java发展初期，java语言是不能够利用硬件提供的这些便利来提升系统的性能的。而随着java不断的发展,**Java本地方法(JNI)的出现**，使得java程序越过JVM直接调用本地方法提供了一种便捷的方式，因而java在并发的手段上也多了起来。而在Doug Lea提供的**concurrent**包中，CAS理论是它实现整个java包的基石。

**CAS 操作包含三个操作数 —— 内存值（V）、预期原值（A）和新值(B)。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做**。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新值B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行**读-修改-写操作**，而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法可以对该操作重新计算。

**非阻塞算法 （nonblocking algorithms）**：一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。

现代的CPU提供了特殊的指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。

# CAS的重要性

**java.util.concurrent包**完全建立在**CAS**之上的，没有CAS就不会有此包。可见CAS的重要性。

# CAS的目的

利用**CPU的CAS指令**，同时**借助JNI来完成Java的非阻塞算法**。其它原子操作都是利用类似的特性完成的。而整个J.U.C都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

# CAS的优点：效率高

**CAS 由于是在硬件层面保证的原子性，不会锁住当前线程，它的效率是很高的**。

# CAS存在的问题(缺点)

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。**ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作**

## ABA问题。

因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，操作成功，但是实际上却变化了。

**ABA问题的解决思路就是使用版本号**。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A－B－A 就会变成1A-2B-3A。

从Java1.5开始JDK的**atomic**包里提供了一个类**AtomicStampedReference**来解决ABA问题。这个类的**compareAndSet方法**作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

CAS 算法实现一个重要前提是需要取出内存中某时刻的数据，而在下一时刻把取出后的数据和内存中原始数据比较并替换，那么在这个时间差内会导致数据的变化。比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。如果链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。

因此前面提到的原子操作**AtomicStampedReference**/**AtomicMarkableReference** 就很有用了。这允许一对变化的元素进行原子操作。

ABA 问题带来的隐患，各种乐观锁的实现中通常都会用**版本号 version** 来对记录或对象标记，避免并发操作带来的问题。在 Java 中，**AtomicStampedReference<E>**也实现了这个作用，它通过包装**[E,Integer]的元组**来对对象标记**版本戳 stamp**，从而避免 ABA 问题。

## 循环时间长开销大。

**自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销**。

如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。**CAS 不适合竞争十分频繁的场景。**

## 只能保证一个共享变量的原子操作。

当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。**从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作**。

# concurrent包的实现

由于java的CAS同时具有 **volatile 读和volatile写**的内存语义，因此Java线程之间的通信现在有了下面四种方式：

A线程写volatile变量，随后B线程读这个volatile变量。

A线程写volatile变量，随后B线程用CAS更新这个volatile变量。

A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。

A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

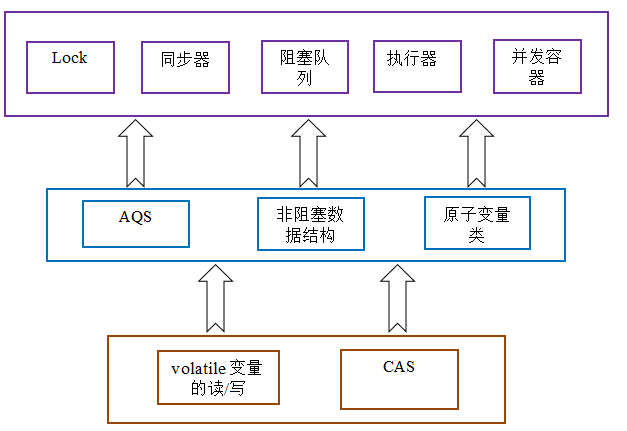
Java的CAS会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行**读-改-写操作**，这是在多处理器中实现同步的关键（从本质上来说，能够支持**原子性读-改-写指令**的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行**原子性读-改-写操作的原子指令**）。**同时，volatile变量的读/写和CAS可以实现线程之间的通信**。把这些特性整合在一起，就**形成了整个concurrent包得以实现的基石**。如果我们仔细分析concurrent包的源代码实现，会发现一个通用化的实现模式：

首先，声明共享变量为**volatile**；

然后，使用**CAS的原子条件**更新来实现线程之间的同步；

同时，配合以**volatile的读/写和CAS所具有的volatile读和写的内存语义**来实现线程之间的通信。

**AQS、非阻塞数据结构、原子变量类**（java.util.concurrent.atomic包中的类），这些concurrent包中的基础类都是使用这种模式来实现的，而**concurrent包中的高层类又是依赖于这些基础类来实现的**。从整体来看，concurrent包的实现示意图如下：



# 如何不用锁机制如何实现共享数据访问

如何**不用锁机制如何实现共享数据访问**？（不要用锁，不要用sychronized 块或者方法，也不要直接使用 jdk 提供的线程安全的数据结构，需要自己实现一个类来保证多个线程同时读写这个类中的共享数据是线程安全的，怎么办 ？）

**无锁化编程**的常用方法 ：硬件 CPU 同步原语 **CAS（Compare and Swap），**如无锁栈，无锁队列（ConcurrentLinkedQueue）等等。现在几乎所有的 CPU 指令都支持 CAS 的原子操作，X86 下对应的是 CMPXCHG 汇编指令，处理器执行 CMPXCHG 指令是一个原子性操作。有了这个原子操作，我们就可以用其来**实现各种无锁（lock free）的数据结构**。

CAS 实现了区别于 sychronized 同步锁的一种**乐观锁**，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改后的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。其实 CAS 也算是有锁操作，只不过是由 CPU 来触发，比 synchronized 性能好的多。**CAS 的关键点在于，系统在硬件层面保证了比较并交换操作的原子性**，处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。**CAS 是非阻塞算法的一种常见实现**。

一个线程间共享的变量，首先在主存中会保留一份，然后每个线程的工作内存也会保留一份副本。**这里说的预期值，就是线程保留的副本**。当该线程从主存中获取该变量的值后，主存中该变量可能已经被其他线程刷新了，但是该线程工作内存中该变量却还是原来的值，这就是所谓的预期值了。当你要用 CAS刷新该值的时候，**如果发现线程工作内存和主存中不一致了**，就会失败，如果一致，就可以更新成功。

## AtomicInteger分析

atomic包提供了一系列原子类。这些类可以保证多线程环境下，当某个线程在执行 atomic 的方法时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个线程执行。Atomic 类在软件层面上是非阻塞的，它的原子性其实是在硬件层面上借助相关的指令来保证的。

AtomicInteger 是一个支持原子操作的 Integer 类，就是保证对AtomicInteger 类型变量的增加和减少操作是原子性的，不会出现多个线程下的数据不一致问题。如果不使用 AtomicInteger，要实现一个按顺序获取的ID，就必须在每次获取时进行加锁操作，以避免出现并发时获取到同样的 ID的现象。Java 并发库中的 AtomicXXX 类均是基于这个原语的实现，拿出AtomicInteger 来研究在没有锁的情况下是如何做到数据正确性的：

来看看++i 是怎么做到的。

public final int incrementAndGet() {

for (;;) {  
int current = get();  
int next = current + 1;  
if (**compareAndSet(current, next))//如果返回的是false，就一直循环**

return next;  
}

}

在这里采用了 CAS 操作，每次从内存中读取数据然后将此数据和+1 后的结果进行 CAS 操作，如果成功就返回结果，否则重试直到成功为止。而 **compareAndSet** 利用 JNI 来完成 CPU 指令的操作，非阻塞算法。

public final boolean **compareAndSet**(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect,update);

}

其中，**unsafe.compareAndSwapInt()**是一个 native 方法，正是调用CAS 原语完成该操作。

首先假设有一个变量 i，i 的初始值为 0。每个线程都对 i 进行+1 操作。CAS是这样保证同步的：

假设有两个线程，线程 1 读取内存中的值为 0，current = 0，next = 1，然后挂起，然后线程 2 对 i 进行操作，将 i 的值变成了 1。线程 2 执行完，回到线程 1，进入 if 里的 compareAndSet 方法，该方法进行的操作的逻辑是，（1）如果操作数的值在内存中没有被修改，返回 true，然后 compareAndSet 方法返回 next 的值（2）如果操作数的值在内存中被修改了，则返回 false，重新进入下一次循环，重新得到 current 的值为 1，next 的值为 2，然后再比较，由于这次没有被修改，所以直接返回 2。

那么，为什么自增操作要通过 CAS 来完成呢？仔细观察

incrementAndGet()方法，发现自增操作其实拆成了两步完成的：

int current = get();

int next = current + 1;

由于 volatile 只能保证读取或写入的是最新值，那么可能出现以下情况：

1.A 线程执行 get()操作，获取 current 值（假设为 1）

2.B 线程执行 get()操作，获取 current 值（为 1）

3.B 线程执行 next = current + 1 操作，next = 2

4.A 线程执行 next = current + 1 操作，next = 2

这样的结果明显不是我们想要的，所以，**自增操作必须采用 CAS 来完成**。