在JDK 5之前Java语言是靠synchronized关键字保证同步的，这会导致有锁

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。**

一、什么是CAS

**CAS,compare and swap**的缩写，中文翻译成**比较并交换**。

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cucurenct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 —— 内存位置（V）、预期原值（A）和新值(B)。** 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。”**（CAS保存了V位置的原来A值，当再次去访问时，若还是旧A，则替换成B，否则，不替换）**

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。

二、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。**

三、CAS存在的问题

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**

1.  **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，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一**，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。

**2. 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。

**3. 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。**

四、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：

1. A线程写volatile变量，随后B线程读这个volatile变量。
2. A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
3. 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。
4. 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：

1. **首先，声明共享变量为volatile；**
2. **然后，使用CAS的原子条件更新来实现线程之间的同步；**
3. **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，非阻塞数据结构和原子变量类（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：

