**1、CAS（Compare and set）：**比较和替换是设计**并发算法时用到的一种技术**。简单来说，比较和替换是使用一个期望值和一个变量的当前值进行比较，如果当前变量的值与我们期望的值相等，就使用一个新值替换当前变量的值。**CAS是一种系统原语**（所谓原语属于操作系统用语范畴。**原语由若干条指令组成的，用于完成一定功能的一个过程**。primitive or atomic action 是由若干个机器指令构成的完成某种特定功能的一段程序，具有**不可分割性**·即**原语的执行必须是连续的，在执行过程中不允许被中断**）。CAS是Compare And Set的缩写。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值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。

     在x86 平台上，CPU提供了在**指令执行期间对总线加锁**的手段。CPU芯片上有一条引线#HLOCK pin，如果汇编语言的程序中在一条指令前面加上前缀"LOCK"，经过汇编以后的机器代码就使CPU在执行这条指令的时候把#HLOCK pin的电位拉低，持续到这条指令结束时放开，从而把总线锁住，这样同一总线上别的CPU就暂时不能通过总线访问内存了，**保证了这条指令在多处理器环境中的原子性**。

CAS的全称为Compare-And-Swap，是一条**CPU的原子指令**，其作用是让CPU比较后原子地更新某个位置的值，经过调查发现，其实现方式是**基于硬件平台的汇编指令**，就是说**CAS是靠硬件实现**的，JVM只是封装了汇编调用，那些**AtomicInteger类便是使用了这些封装后的接口**。

**CAS用作原子操作**

现在CPU内部已经执行原子的CAS操作。Java5以来，你可以使用java.util.concurrent.atomic包中的一些原子类来使用CPU中的这些功能。

使用Java5+提供的CAS特性而不是使用自己实现的的好处是Java5+中**内置的CAS特性**可以让你利用**底层的程序运行机器的CPU的CAS特性**。这会使含有CAS的代码运行更快。

**java.util.concurrent包中借助CAS实现了区别于synchronouse同步锁的一种乐观锁。**  
​​CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许java调用其他语言。​而compareAndSwapInt就是借助C来**调用CPU底层指令实现**的。

**​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。

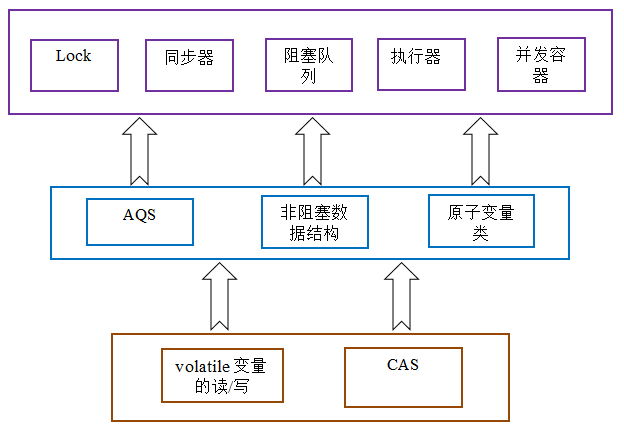
Java的CAS会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，**volatile变量的读/写和CAS可以实现线程之间的通信**。把这些特性整合在一起，就形成了整个concurrent包得以实现的基石。如果我们仔细分析concurrent包的源代码实现，会发现一个通用化的实现模式：

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：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/004msBIGzy74jZFZxxZ87)

volatile与CAS是CPU内部的特性，处理器提供了很多**LOCK前缀的指令**来实现，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使用共享内存。

**2、volatile没办法保证对变量的操作的原子性。**

　　在前面已经提到过，**自增操作是不具备原子性**的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

　　假如某个时刻变量inc的值为10，线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

　　然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的**缓存行无效**，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

　　然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

　　那么两个线程分别进行了一次自增操作后，inc只增加了1。

**​3、volatile能保证有序性**

　　在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

　　volatile关键字禁止指令重排序有两层意思：

　　1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

　　2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行

**​4、原子操作类**

​在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。**atomic是利用CAS来实现原子性操作**的（Compare And Swap），**CAS实际上是利用处理器提供的CMPXCHG指令实现**的，而处理器执行CMPXCHG指令是一个原子性操作。

**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**