CAS原理分析

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（后面的章节还会谈到锁）。

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。

CAS 操作

上面的乐观锁用到的机制就是CAS，Compare and Swap。

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。

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。

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。

```
private volatile int value;
```

首先毫无以为，在没有锁的机制下可能需要借助volatile原语，保证线程间的数据是可见的（共享的）。这样才获取变量的值的时候才能直接读取。

```
public final int get() {
    return value;
  }
```

然后来看看++i是怎么做到的。

```
public final int incrementAndGet() {
  for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next))
      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。

```
public final boolean compareAndSet(int expect, int update) {
  return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
```

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看起来很爽，但是会导致“**ABA问题**”。

CAS算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

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。

==========

http://www.cnblogs.com/maxupeng/archive/2011/07/01/2096035.html

并发中CAS的含义及Java中AtomicXXX类的分析

维基百科：

In computer science, the **compare-and-swap** CPU instruction ("**CAS**") (or the Compare & Exchange - CMPXCHG instruction in the x86 and Itanium architectures) is a special instruction that atomically (regarding intel x86, lock prefix should be there to make it really atomic) compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple Boolean response (this variant is often called **compare-and-set**), or by returning the value read from the memory location (*not* the value written to it). Compare-and-Swap (and Compare-and-Swap-Double) has been an integral part of the IBM 370(and all successor) architectures since 1970. The operating systems which run on these architectures make extensive use of Compare-and-Swap (and Compare-and-Swap-Double) to facilitate process (i.e., system and user tasks) and processor (i.e., central processors) parallelism while eliminating, to the greatest degree possible, the "disabled spin locks" which were employed in earlier IBM operating systems. In these operating systems, new units of work may be instantiated "globally", into the Global Service Priority List, or "locally", into the Local Service Priority List, by the execution of a single Compare-and-Swap instruction. This dramatically improved the responsiveness of these operating systems.

==========

总结：CAS是硬件CPU提供的元语，它的原理：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。

Java并发库中的AtomicXXX类均是基于这个元语的实现，以AtomicInteger为例：

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}

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元语完成该操作。