前言

CAS（Compare and Swap），即比较并替换，实现并发算法时常用到的一种技术，Doug lea大神在java同步器中大量使用了CAS技术，鬼斧神工的实现了多线程执行的安全性。

CAS的思想很简单：三个参数，一个当前内存值V、旧的预期值A、即将更新的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。

问题

一个 n ++ 的问题。

```
public class Case {

    public volatile int n;

    public void add() {
        n++;
    }
}
```

通过 javap - verbose Case 看看add方法的字节码指令

```
public void add();
    flags: ACC_PUBLIC
    Code:
      stack=3, locals=1, args_size=1
         0: aload_0
         1: dup
         2: getfield      #2                  // Field n:I
         5: iconst_1
         6: iadd
         7: putfield      #2                  // Field n:I
        10: return
```

n ++ 被拆分成了几个指令：

1. 执行 getfield 拿到原始n；
2. 执行 iadd 进行加1操作；
3. 执行 putfield 写把累加后的值写回n；

通过volatile修饰的变量可以保证线程之间的可见性，但并不能保证这3个指令的原子执行，在多线程并发执行下，无法做到线程安全，得到正确的结果，那么应该如何解决呢？

解决方案

在 add 方法加上synchronized修饰

```
public class Case {
```

```
    public volatile int n;

    public synchronized void add() {
        n++;
    }
}
```

这个方案当然可行，但是性能上差了点，还有其它方案么？

再来看一段代码

```
public int a = 1;
public boolean compareAndSwapInt(int b) {
    if (a == 1) {
        a = b;
        return true;
    }
    return false;
}
```

如果这段代码在并发下执行，会发生什么？

假设线程1和线程2都过了 `a == 1` 的检测，都准备执行对a进行赋值，结果就是两个线程同时修改了变量a，显然这种结果是无法符合预期的，无法确定a的最终值。

解决方法也同样暴力，在compareAndSwapInt方法加锁同步，变成一个原子操作，同一时刻只有一个线程才能修改变量a。

除了低性能的加锁方案，我们还可以使用JDK自带的CAS方案，在CAS中，比较和替换是一组原子操作，不会被外部打断，且在性能上更占有优势。

下面以 `AtomicInteger` 的实现为例，分析一下CAS是如何实现的。

```
public class AtomicInteger extends Number implements java.io.Serializable {
    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

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

1. Unsafe，是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地（native）方法来访问，Unsafe相当于一个后门，基于该类可以直接操作特定内存的数据。

2. 变量valueOffset，表示该变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的。
3. 变量value用volatile修饰，保证了多线程之间的内存可见性。

看看 AtomicInteger 如何实现并发下的累加操作：

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}
```

```
//unsafe.getAndAddInt
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
    return var5;
}
```

假设线程A和线程B同时执行getAndAdd操作（分别跑在不同CPU上）：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程A和线程B各自持有一份value的副本，值为3。
2. 线程A通过 getIntVolatile（var1，var2）拿到value值3，这时线程A被挂起。
3. 线程B也通过 getIntVolatile（var1，var2）方法获取到value值3，运气好，线程B没有被挂起，并执行 compareAndSwapInt 方法比较内存值也为3，成功修改内存值为2。
4. 这时线程A恢复，执行 compareAndSwapInt 方法比较，发现自己手里的值(3)和内存的值(2)不一致，说明该值已经被其它线程提前修改过了，那只能重新来一遍了。
5. 重新获取value值，因为变量value被volatile修饰，所以其它线程对它的修改，线程A总是能够看到，线程A继续执行 compareAndSwapInt 进行比较替换，直到成功。

整个过程中，利用CAS保证了对于value的修改的并发安全，继续深入看看Unsafe类中的compareAndSwapInt方法实现。

```
public final native boolean compareAndSwapInt(Object paramObject, long paramLong, int paramInt1, int
```

Unsafe类中的compareAndSwapInt，是一个本地方法，该方法的实现位于 unsafe.cpp 中

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offs
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

1. 先想办法拿到变量value在内存中的地址。
2. 通过 Atomic::cmpxchg 实现比较替换，其中参数x是即将更新的值，参数e是原内存的值。

如果是Linux的x86， Atomic :: cmpxchg 方法的实现如下：

```
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```

看到这汇编，内心崩溃 ☺

__asm__ 表示汇编的开始 volatile 表示禁止编译器优化 LOCK_IF_MP 是个内联函数

```
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

Window的x86实现如下：

```
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
    int mp = os::isMP(); //判断是否是多处理器
    _asm {
        mov edx, dest
        mov ecx, exchange_value
        mov eax, compare_value
        LOCK_IF_MP(mp)
        cmpxchg dword ptr [edx], ecx
    }
}

// Adding a lock prefix to an instruction on MP machine
// VC++ doesn't like the lock prefix to be on a single line
// so we can't insert a label after the lock prefix.
// By emitting a lock prefix, we can define a label after it.
#define LOCK_IF_MP(mp) __asm cmp mp, 0  \
                       __asm je L0      \
                       __asm _emit 0xF0 \
                       __asm L0:
```

LOCK_IF_MP 根据当前系统是否为多核处理器决定是否为cmpxchg指令添加lock前缀。

1. 如果是多处理器，为cmpxchg指令添加lock前缀。
2. 反之，就省略lock前缀。（单处理器会不需要lock前缀提供的内存屏障效果）

intel手册对lock前缀的说明如下：

1. 确保后续指令执行的原子性。
   在Pentium及之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其它处理器暂时无法通过总线访问内存，很显然，这个开销很大。在新的处理器中，Intel使用缓存锁定来保证指令执行的原子性，缓存锁定将

大大降低lock前缀指令的执行开销。
2. 禁止该指令与前面和后面的读写指令重排序。
3. 把写缓冲区的所有数据刷新到内存中。

上面的第2点和第3点所具有的内存屏障效果，保证了CAS同时具有volatile读和volatile写的内存语义。

CAS缺点

CAS存在一个很明显的问题，即ABA问题。

问题：如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？

如果在这段期间曾经被改成B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类 `AtomicStampedReference`，它可以通过控制变量值的版本来保证CAS的正确性。