链滴

[转]Java CAS 原理剖析

作者：boolean-dev

原文链接：https://ld246.com/article/1565572474956

来源网站：链滴

Java CAS 原理剖析

本文转载来自卡巴拉的树的Java CAS 原理剖析

在Java并发中，我们最初接触的应该就是synchronized关键字了，但是synchronized属于重量级锁很多时候会引起性能问题，volatile也是个不错的选择，但是volatile不能保证原子性，只能在某些场下使用。

像synchronized这种独占锁属于**悲观锁**，它是在假设一定会发生冲突的，那么加锁恰好有用，除此之，还有**乐观锁**，乐观锁的含义就是假设没有发生冲突，那么我正好可以进行某项操作，如果要是发生突呢，那我就重试直到成功，乐观锁最常见的就是CAS。

我们在读Concurrent包下的类的源码时，发现无论是**ReenterLock内部的AQS，还是各种Atomic头的原子类**，内部都应用到了CAS，最常见的就是我们在并发编程时遇到的i++这种情况。传统的方肯定是在方法上加上synchronized关键字:

```
public class Test {

    public volatile int i;

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

```
}
```

但是这种方法在性能上可能会差一点，我们还可以使用AtomicInteger，就可以保证i原子的++了。

```
public class Test {

    public AtomicInteger i;

    public void add() {
        i.getAndIncrement();
    }
}
```

我们来看getAndIncrement的内部：

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

再深入到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;
}
```

这里我们见到compareAndSwapInt这个函数，它也是CAS缩写的由来。那么仔细分析下这个函数做什么呢?

首先我们发现compareAndSwapInt前面的this，那么它属于哪个类呢，我们看上一步getAndAddInt前面是unsafe。这里我们进入的Unsafe类。这里要对Unsafe类做个说明。结合AtomicInteger的定来说：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // 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;
    ...
```

在AtomicInteger数据定义的部分，我们可以看到，其实实际存储的值是放在value中的，除此之外们还获取了unsafe实例，并且定义了valueOffset。再看到static块，懂类加载过程的都知道，static的加载发生于类加载的时候，是最先初始化的，这时候我们调用unsafe的objectFieldOffset从Atomic文件中获取value的偏移量，那么valueOffset其实就是记录value的偏移量的。

再回到上面一个函数getAndAddInt，我们看var5获取的是什么，通过调用unsafe的getIntVolatile(va1, var2)，这是个native方法，具体实现到JDK源码里去看了，其实就是获取var1中，var2偏移量处的。var1就是AtomicInteger，var2就是我们前面提到的valueOffset,这样我们就从内存里获取到现在valueOffset处的值了。

现在重点来了，compareAndSwapInt（var1, var2, var5, var5 + var4）其实换成compareAndSwapnt（obj, offset, expect, update）比较清楚，意思就是如果obj内的value和expect相等，就证明没其他线程改变过这个变量，那么就更新它为update，如果这一步的CAS没有成功，那就采用自旋的方继续进行CAS操作，取出乍一看这也是两个步骤了啊，其实在JNI里是借助于一个CPU指令完成的。以还是原子操作。

1. CAS底层原理

CAS底层使用JNI调用C代码实现的，如果你有Hotspot源码，那么在Unsafe.cpp里可以找到它的实现：

```
static JNINativeMethod methods_15[] = {
    //省略一堆代码...
    {CC"compareAndSwapInt",  CC"("OBJ"J""I""I"")Z",      FN_PTR(Unsafe_CompareAndSwapInt)}

    {CC"compareAndSwapLong", CC"("OBJ"J""J""J"")Z",      FN_PTR(Unsafe_CompareAndSwapL
ng)},
    //省略一堆代码...
};
```

我们可以看到compareAndSwapInt实现是在Unsafe_CompareAndSwapInt里面，再深入到Unsafe_ompareAndSwapInt:

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject ob
, jlong offset, jint e, jint x))
  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
```

p是取出的对象，addr是p中offset处的地址，最后调用了Atomic::cmpxchg(x, addr, e), 其中参数x即将更新的值，参数e是原内存的值。代码中能看到cmpxchg有基于各个平台的实现，这里我选择Linx X86平台下的源码分析：

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_v
lue) {
  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;
}
```

这是一段小汇编，__asm__说明是ASM汇编，__volatile__禁止编译器优化

```
// Adding a lock prefix to an instruction on MP machine
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

os::is_MP判断当前系统是否为多核系统，如果是就给总线加锁，所以同一芯片上的其他处理器就暂时能通过总线访问内存，保证了该指令在多处理器环境下的原子性。

在正式解读这段汇编前，我们来了解下嵌入汇编的基本格式：

```
asm ( assembler template
    : output operands                /* optional */
    : input operands                 /* optional */
    : list of clobbered registers    /* optional */
    );
```

- **template**就是cmpxchgl %1,(%3)表示汇编模板
- **output operands**表示输出操作数,=a对应eax寄存器
- **input operand** 表示输入参数，%1 就是exchange_value, %3是dest, %4就是mp， r表示任意寄器，a还是eax寄存器
- **list of clobbered registers**就是些额外参数，cc表示编译器cmpxchgl的执行将影响到标志寄存器,emory告诉编译器要重新从内存中读取变量的最新值，这点实现了volatile的感觉。

那么表达式其实就是cmpxchgl exchange_value ,dest，我们会发现%2也就是compare_value没有上，这里就要分析cmpxchgl的语义了。cmpxchgl末尾l表示操作数长度为4，上面已经知道了。cmpxhgl会默认比较eax寄存器的值即compare_value和exchange_value的值，**如果相等，就把dest的值值给exchange_value,否则，将exchange_value赋值给eax。**具体汇编指令可以查看Intel手册CMP CHG

最终，JDK通过CPU的cmpxchgl指令的支持，实现AtomicInteger的CAS操作的原子性。

2. CAS 的问题

1. **ABA问题**

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

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。这就是CAS的ABA问题。 常见的解决思路是使用版本号。在变量前面追加上版本号，每次变量更新时候把版本号加一，那么A-B-A 就会变成1A-2B-3A。 目前在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为给定的更新值。

1. **循环时间长开销大**

上面我们说过如果CAS不成功，则会原地自旋，如果长时间自旋会给CPU带来非常大的执行开销。