CAS学习

什么是CAS

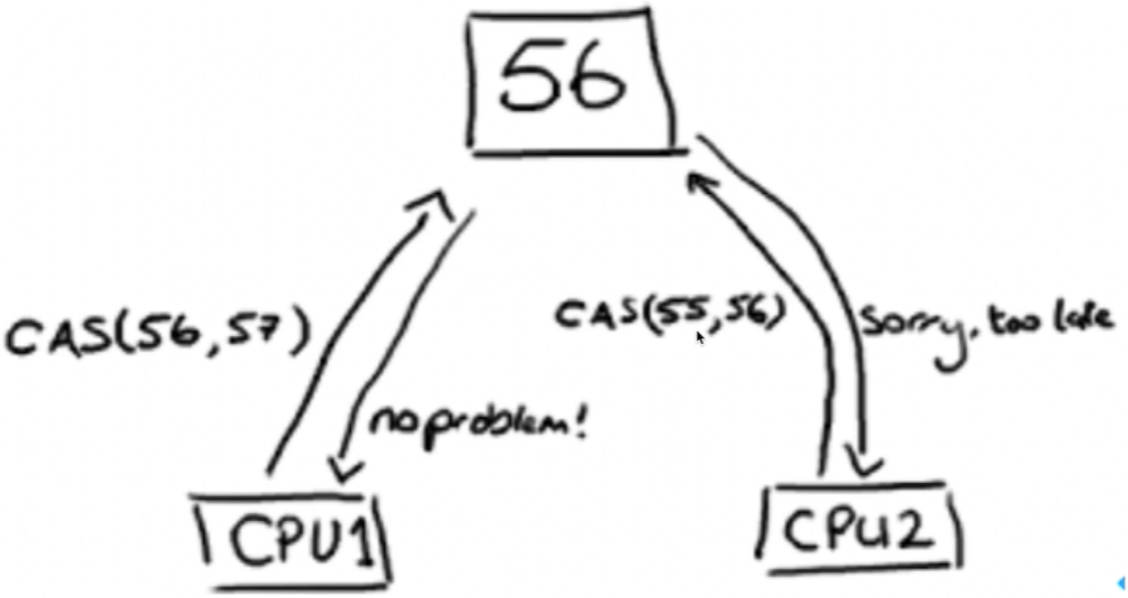
CAS：Compare and Swap，即比较再交换。CAS更应该理解为一种思想，java中的乐观锁、数据库中的乐观锁都使用到了这种思想。

CAS的思想

CAS有三个操作数：内存值V、预期值A、要修改的值B，**当且仅当预期值A和内存值V相同时**，才将内存值修改为B，否则什么都不做。最后返回现在的V值。

简单的说，CAS 需要你额外给出一个期望值，也就是你认为这个内存中的变量现在应该是什么样子的。如果变量值不是你想象的那样，就说明要操作的变量已经被别人修改过了。你就需要重新读取，再次尝试修改就好了。（这里说的变量稍显的有些险隘，现实中也可是其他的，比如数据库中的一行记录）

因此，当多个线程同时CAS 一个变量时，**只会有一个线程胜出**，并成功更新，其余均会失败。**失败的线程不会挂起（阻塞），仅是被告知失败**，并且允许再次尝试，基于这样的思想，CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰。



CAS等价代码：我们用synchronized来保证CAS的原子性，模拟CPU中CAS的原子性

```
package cas;

/**
 * 描述：       模拟CAS操作，等价代码
 */
public class SimulatedCAS {
    private volatile int value;

    public synchronized int compareAndSwap(int expectedValue, int newValue)
{
        int oldValue = value;
```

```
        if (oldValue == expectedValue) {
            value = newValue;
        }
        return oldValue;
    }
}
```

CAS 底层原理

CAS 的底层原理归功于硬件指令集的发展，实际上，我们可以使用同步将这两个操作变成原子的，但是这么做就失去了CAS的意义了。所以我们只能靠硬件来完成，硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成。这类指令常用的有：

1. 测试并设置（Tetst-and-Set）
2. 获取并增加（Fetch-and-Increment）
3. 交换（Swap）
4. **比较并交换（Compare-and-Swap）**
5. 加载链接/条件存储（Load-Linked/Store-Conditional）

其中，前面的3条是20世纪时，大部分处理器已经有了，后面的2条是现代处理器新增的。而且这两条指令的目的和功能是类似的，在IA64，x86 指令集中有 cmpxchg 指令完成 CAS 功能，在 sparc-TSO 也有 casa 指令实现，而在 ARM 和 PowerPC 架构下，则需要使用一对 ldrex/strex 指令来完成 LL/SC 的功能。

CPU 实现原子指令有2种方式：

1. 通过总线锁定来保证原子性。 总线锁定其实就是处理器使用了总线锁，所谓总线锁就是使用处理器提供的一个 LOCK# 信号，当一个处理器咋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共享内存。但是该方法成本太大。因此有了下面的方式。
2. 通过缓存锁定来保证原子性。 所谓 缓存锁定 是指内存区域如果被缓存在处理器的缓存行中，并且在Lock 操作期间被锁定，那么当他执行锁操作写回到内存时，处理器不在总线上声言 LOCK# 信号，而时修改内部的内存地址，并允许他的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改两个以上处理器缓存的内存区域数据（这里和 volatile 的可见性原理相同），当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

注意：有两种情况下处理器不会使用缓存锁定。

1. 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用**总线锁定。**
2. 有些处理器不支持缓存锁定，对于 Intel 486 和 Pentium 处理器，就是锁定的内存区域在处理器的缓存行也会调用总线锁定。

以AtomicInteger为例，分析在Java中是如何利用CAS实现原子操作的?

AtomicInteger加载Unsafe工具，用来直接操作内存数据，这里是JDK13中的实现

```
private static final jdk.internal.misc.Unsafe U =
jdk.internal.misc.Unsafe.getUnsafe();

// 获取下面value的内存偏移量
private static final long VALUE = U.objectFieldOffset(AtomicInteger.class,
"value");
 // AtomicInteger的值，value是用volatile修饰的，保证了多线程之间看到的value值是同一份。
private volatile int value;
```

进入`Unsafe#getUnsafe()`

```
private Unsafe() {}
private static final Unsafe theUnsafe = new Unsafe();

public static Unsafe getUnsafe() {
    return theUnsafe;
}
```

Unsafe是CAS的核心类。Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它**提供了硬件级别的原子操作。**

无论是AtomicInteger中的`getAndAdd(int delta)`方法还是`getAndIncrement()`等方法，发现只要是与修改值相关的最后都调用了Unsafe类的`getAndAddInt(this, VALUE, delta)`

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

public final int getAndDecrement() {
    return U.getAndAddInt(this, VALUE, -1);
}


public final int getAndAdd(int delta) {
    return U.getAndAddInt(this, VALUE, delta);
}

public final int incrementAndGet() {
    return U.getAndAddInt(this, VALUE, 1) + 1;
}


public final int decrementAndGet() {
    return U.getAndAddInt(this, VALUE, -1) - 1;
}

public final int addAndGet(int delta) {
    return U.getAndAddInt(this, VALUE, delta) + delta;
}
```

`VALUE`表示的是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原值的，这样我们就能通过unsafe来实现CAS了。

进入`Unsafe`中的`getAndAddInt(Object o, long offset, int delta)`

```
/**
* @param o object/array to update the field/element in  要更新的对象：这里就是AtomicInteger
* @param offset field/element offset  要更改的变量的内存偏移量（内存地址）
* @param delta the value to add  要增加的值
* @return the previous value
* @since 1.8
*/
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

可以发现Unsafe的 getAndAddInt利用了自旋+CAS的方式,在这个过程中,通过compareandswapInt比较并更新 value值,如果更新失败,重新获取,然后再次更新,直到更新成功，进入`weakCompareAndSetInt`

```
public final boolean weakCompareAndSetInt(Object o, long offset,
                                          int expected,
                                          int x) {
    // 调用下面的native方法
    return compareAndSetInt(o, offset, expected, x);
}
// 参数含义分别是：要更新的对象、变量的内存偏移量（通过offset查看内存中的值）、期望值（与内存中的值去比较）、x：最终要被更新为的值
public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);
```

进入Unsafe类中的compareAndSwapInt方法（cpp代码）

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSetInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x)) {
  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
```

方法中先想办法拿到变量value在内存中的地址：变量 `addr`。

然后通过`Atomic::cmpxchg(x, addr, e)`实现原子性的比较和替换，其中参数x是即将更新的值，参数e是原内存的值。至此，最终完成了CAS的全过程。

缺点

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。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，每次变量更新的时候把版本号加一，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。

自旋时间过长

从`Unsafe#getAndAddInt(Object o, long offset, int delta)`的源码中可以看到：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。

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CAS能保证其原子性，如果对多个共享变量进行操作,CAS就不能保证其原子性，因为多个变量之间是独立的。有一个解决方案是利用对象整合多个共享变量，即一个类中的成员变量就是这几个共享变量。然后将这个对象做CAS操作就可以保证其原子性。atomic中提供了**AtomicReference**来保证引用对象之间的原子性