Unsafe与CAS - 五月的仓颉 - 博客园

Unsafe与CAS

Unsafe

简单讲一下这个类。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**。

这个类尽管里面的方法都是public的，但是并没有办法使用它们，JDK API文档也没有提供任何关于这个类的方法的解释。总而言之，对于Unsafe类的使用都是受限制的，只有授信的代码才能获得该类的实例，当然JDK库里面的类是可以随意使用的。

从第一行的描述可以了解到Unsafe提供了硬件级别的操作，比如说获取某个属性在内存中的位置，比如说修改对象的字段值，即使它是私有的。不过Java本身就是为了屏蔽底层的差异，对于一般的开发而言也很少会有这样的需求。

举两个例子，比方说：

```
public native long staticFieldOffset(Field paramField);
```

这个方法可以用来获取给定的paramField的内存地址偏移量，这个值对于给定的field是唯一的且是固定不变的。再比如说：

```
public native int arrayBaseOffset(Class paramClass);
public native int arrayIndexScale(Class paramClass);
```

前一个方法是用来获取数组第一个元素的偏移地址，后一个方法是用来获取数组的转换因子即数组中元素的增量地址的。最后看三个方法：

```
public native long allocateMemory(long paramLong);
public native long reallocateMemory(long paramLong1, long paramLong2);
public native void freeMemory(long paramLong);
```

分别用来分配内存，扩充内存和释放内存的。

当然这需要有一定的C/C++基础，对内存分配有一定的了解，这也是为什么我一直认为C/C++开发者转行做Java会有优势的原因。

CAS

CAS，Compare and Swap即比较并交换，设计并发算法时常用到的一种技术，java.util.concurrent包全完建立在CAS之上，没有CAS也就没有此包，可见CAS的重要性。

当前的处理器基本都支持CAS，只不过不同的厂家的实现不一样罢了。**CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false**。

CAS也是通过Unsafe实现的，看下Unsafe下的三个方法：

public final native boolean compareAndSwapObject(Object paramObject1, long paramLong, Object paramObject2, Object paramObject3);

public final native boolean compareAndSwapInt(Object paramObject, long paramLong, int paramInt1, int paramInt2);

public final native boolean compareAndSwapLong(Object paramObject, long paramLong1, long paramLong2, long paramLong3);

就拿中间这个比较并交换Int值为例好了，如果我们不用CAS，那么代码大致是这样的：

```
1 public int i = 1;
2
3 public boolean compareAndSwapInt(int j)
4 {
5    if (i == 1)
6    {
7       i = j;
8       return true;
9    }
10    return false;
11 }
```

当然这段代码在并发下是肯定有问题的，有可能线程1运行到了第5行正准备运行第7行，线程2运行了，把i修改为10，线程切换回去，线程1由于先前已经满足第5行的if了，所以导致两个线程同时修改了变量i。

解决办法也很简单，给compareAndSwapInt方法加锁同步就行了，这样，compareAndSwapInt方法就变成了一个原子操作。CAS也是一样的道理，比较、交换也是一组原子操作，不会被外部打断，先根据paramLong/paramLong1获取到内存当中当前的内存值V，在将内存值V和原值A作比较，要是相等就修改为要修改的值B，由于CAS都是硬件级别的操作，因此效率会高一些。

由CAS分析AtomicInteger原理

java.util.concurrent.atomic包下的原子操作类都是基于CAS实现的，下面拿AtomicInteger分析一下，首先是AtomicInteger类变量的定义：

```
1 private static final Unsafe unsafe = Unsafe.getUnsafe();
2 private static final long valueOffset;
3
4 static {
5  try {
6     valueOffset = unsafe.objectFieldOffset
7        (AtomicInteger.class.getDeclaredField("value"));
8   } catch (Exception ex) { throw new Error(ex); }
9 }
10
11 private volatile int value;
```

关于这段代码中出现的几个成员属性：

1、Unsafe是CAS的核心类，前面已经讲过了

2、valueOffset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

3、value是用volatile修饰的，这是非常关键的

下面找一个方法getAndIncrement来研究一下AtomicInteger是如何实现的，比如我们常用的addAndGet方法：

```
1 public final int addAndGet(int delta) {
2     for (;;) {
3         int current = get();
4         int next = current + delta;
5         if (compareAndSet(current, next))
6             return next;
7     }
8 }
```

```
1 public final int get() {
2     return value;
3 }
```

这段代码如何在不加锁的情况下通过CAS实现线程安全，我们不妨考虑一下方法的执行：

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1和线程2各自持有一份value的副本，值为3

2、线程1运行到第三行获取到当前的value为3，线程切换

3、线程2开始运行，获取到value为3，利用CAS对比内存中的值也为3，比较成功，修改内存，此时内存中的value改变比方说是4，线程切换

4、线程1恢复运行，利用CAS比较发现自己的value为3，内存中的value为4，得到一个重要的结论-->**此时value正在被另外一个线程修改，所以我不能去修改它**

5、线程1的compareAndSet失败，循环判断，因为value是volatile修饰的，所以它具备可见性的特性，线程2对于value的改变能被线程1看到，只要线程1发现当前获取的value是4，内存中的value也是4，说明**线程2对于value的修改已经完毕并且线程1可以尝试去修改它**

6、最后说一点，比如说此时线程3也准备修改value了，没关系，因为比较-交换是一个原子操作不可被打断，线程3修改了value，线程1进行compareAndSet的时候必然返回的false，这样线程1会继续循环去获取最新的value并进行compareAndSet，直至获取的value和内存中的value一致为止

整个过程中，利用CAS机制保证了对于value的修改的线程安全性。

CAS的缺点

CAS看起来很美，但这种操作显然无法涵盖并发下的所有场景，并且CAS从语义上来说也不是完美的，存在这样一个逻辑漏洞：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了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。这个漏洞称为CAS操作的"ABA"问题。java.util.concurrent包为了解决这个问题，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。不过目前来说这个类比较"鸡肋"，大部分情况下ABA问题并不会影响程序并发的正确性，如果需要解决ABA问题，使用传统的互斥同步可能回避原子类更加高效。

==========

我不能保证写的每个地方都是对的，但是至少能保证不复制、不黏贴，保证每一句话、每一行代码都经过了认真的推敲、仔细的斟酌。每一篇文章的背后，希望都能看到自己对于技术、对于生活的态度。

我相信乔布斯说的，只有那些疯狂到认为自己可以改变世界的人才能真正地改变世界。面对压力，我可以挑灯夜战、不眠不休；面对困难，我愿意迎难而上、永不退缩。

其实我想说的是，我只是一个程序员，这就是我现在纯粹人生的全部。

==========