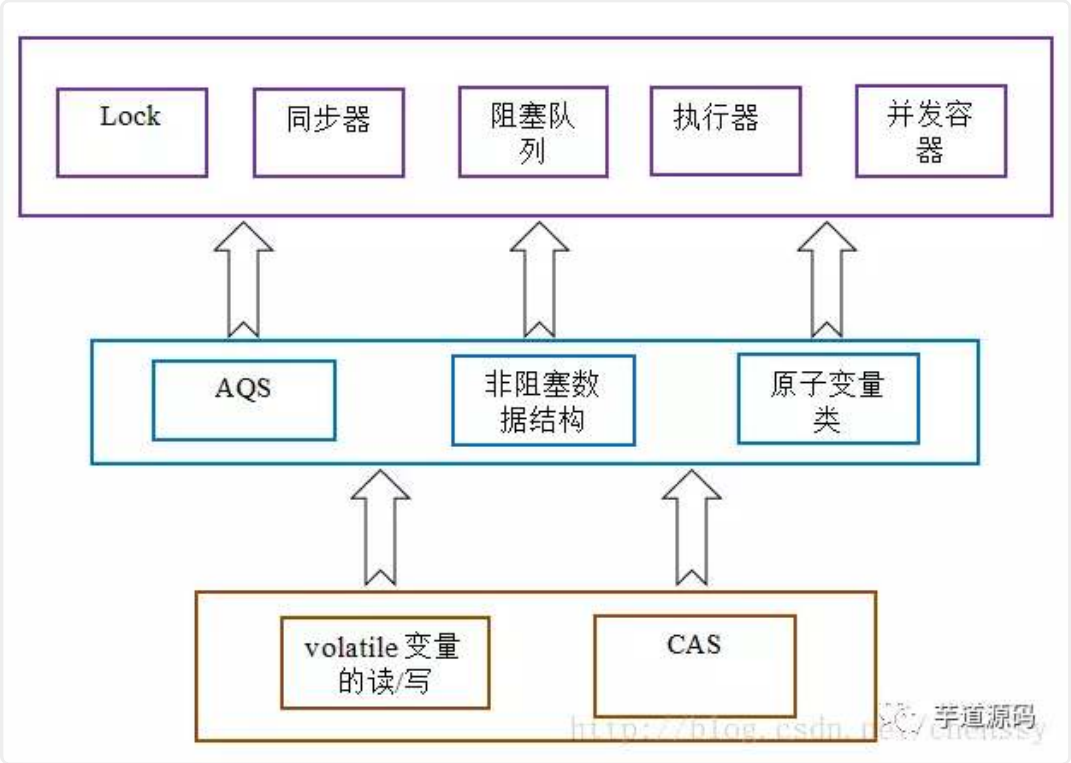
【死磕Java并发】—- 深入分析CAS

2018-02-27 大明哥 芋道源码

CAS，Compare And Swap，即比较并交换。Doug lea大神在同步组件中大量使用CAS技术鬼斧神工地实现了Java多线程的并发操作。整个AQS同步组件、Atomic原子类操作等等都是以CAS实现的，甚至ConcurrentHashMap在1.8的版本中也调整为了CAS+Synchronized。可以说CAS是整个JUC的基石。



这里写图片描述

CAS分析

在CAS中有三个参数：内存值V、旧的预期值A、要更新的值B，当且仅当内存值V的值等于旧的预期值A时才会将内存值V的值修改为B，否则什么都不干。其伪代码如下：

```
if(this.value == A){
    this.value = B
    return true;
}else{
    return false;
}
```

JUC下的atomic类都是通过CAS来实现的，下面就以AtomicInteger为例来阐述CAS的实现。如下：

```
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;
```

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：Unsafe，它提供了硬件级别的原子操作。

valueOffset为变量值在内存中的偏移地址，unsafe就是通过偏移地址来得到数据的原值的。

value当前值，使用volatile修饰，保证多线程环境下看见的是同一个。

我们就以AtomicInteger的addAndGet()方法来做说明，先看源代码：

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

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;
}
```

内部调用unsafe的getAndAddInt方法，在getAndAddInt方法中主要是看compareAndSwapInt方法：

```
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

该方法为本地方法，有四个参数，分别代表：对象、对象的地址、预期值、修改值（有位伙伴告诉我他面试的时候就问到这四个变量是啥意思...+_+）。该方法的实现这里就不做详细介绍了，有兴趣的伙伴可以看看openjdk的源码。

CAS可以保证一次的读-改-写操作是原子操作，在单处理器上该操作容易实现，但是在多处理器上实现就有点儿复杂了。

CPU提供了两种方法来实现多处理器的原子操作：总线加锁或者缓存加锁。

总线加锁：总线加锁就是就是使用处理器提供的一个LOCK#信号，当一个处理器在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使用共享内存。但是这种处理方式显得有点儿霸道，不厚道，他把CPU和内存之间的通信锁住了，在锁定期间，其他处理器都不能其他内存地址的数据，其开销有点儿大。所以就有了缓存加锁。

缓存加锁：其实针对于上面那种情况我们只需要保证在同一时刻对某个内存地址的操作是原子性的即可。缓存加锁就是缓存在内存区域的数据如果在加锁期间，当它执行锁操作写回内存时，处理器不在输出LOCK#信号，而是修改内部的内存地址，利用缓存一致性协议来保证原子性。缓存一致性机制可以保证同一个内存区域的数据仅能被一个处理器修改，也就是说当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行。

CAS缺陷

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方法：循环时间太长、只能保证一个共享变量原子操作、ABA问题。

循环时间太长

如果CAS一直不成功呢？这种情况绝对有可能发生，如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。在JUC中有些地方就限制了CAS自旋的次数，例如BlockingQueue的SynchronousQueue。

只能保证一个共享变量原子操作

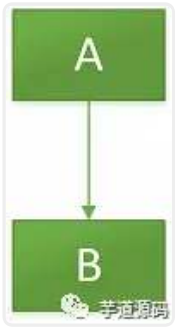
看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了，当然如果你有办法把多个变量整成一个变量，利用CAS也不错。例如读写锁中state的高地位

ABA问题

CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，这就是所谓的ABA问题。对于ABA问题其解决方案是加上版本号，即在每个变量都加上一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A。

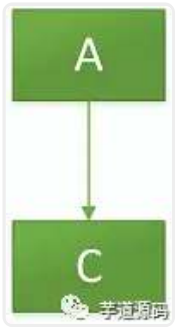
用一个例子来阐述ABA问题所带来的影响。

有如下链表



这里写图片描述

假如我们想要把B替换为A，也就是compareAndSet(this,A,B)。线程1执行B替换A操作，线程2主要执行如下动作，A 、B出栈，然后C、A入栈，最终该链表如下：



这里写图片描述

完成后线程1发现仍然是A，那么compareAndSet(this,A,B)成功，但是这时会存在一个问题就是B.next = null,compareAndSet(this,A,B)后，会导致C丢失，改栈仅有一个B元素，平白无故把C给丢失了。

CAS的ABA隐患问题，解决方案则是版本号，Java提供了AtomicStampedReference来解决。AtomicStampedReference通过包装[E,Integer]的元组来对对象标记版本戳stamp，从而避免ABA问题。对于上面的案例应该线程1会失败。

AtomicStampedReference的compareAndSet()方法定义如下：

```
public boolean compareAndSet(V   expectedReference,
                             V   newReference,
                             int expectedStamp,
                             int newStamp) {
    Pair<V> current = pair;
    return
        expectedReference == current.reference &&
        expectedStamp == current.stamp &&
        ((newReference == current.reference &&
          newStamp == current.stamp) ||
         casPair(current, Pair.of(newReference, newStamp)));
}
```

compareAndSet有四个参数，分别表示：预期引用、更新后的引用、预期标志、更新后的标志。源码部门很好理解预期的引用 == 当前引用，预期的标识 == 当前标识，如果更新后的引用和标志和当前的引用和标志相等则直接返回true，否则通过Pair生成一个新的pair对象与当前pair CAS替换。Pair为AtomicStampedReference的内部类，主要用于记录引用和版本戳信息（标识），定义如下：

```
private static class Pair<T> {
    final T reference;
    final int stamp;
    private Pair(T reference, int stamp) {
        this.reference = reference;
        this.stamp = stamp;
    }
    static <T> Pair<T> of(T reference, int stamp) {
        return new Pair<T>(reference, stamp);
    }
}

private volatile Pair<V> pair;
```

Pair记录着对象的引用和版本戳，版本戳为int型，保持自增。同时Pair是一个不可变对象，其所有属性全部定义为final，对外提供一个of方法，该方法返回一个新建的Pari对象。pair对象定义为volatile，保证多线程环境下的可见

性。在AtomicStampedReference中，大多方法都是通过调用Pair的of方法来产生一个新的Pair对象，然后赋值给变量pair。如set方法：

```
    public void set(V newReference, int newStamp) {
        Pair<V> current = pair;
        if (newReference != current.reference || newStamp != current.stamp)
            this.pair = Pair.of(newReference, newStamp);
    }
```

下面我们将通过一个例子可以可以看到AtomicStampedReference和AtomicInteger的区别。我们定义两个线程，线程1负责将100 —> 110 —> 100，线程2执行 100 —>120，看两者之间的区别。

```
public class Test {
    private static AtomicInteger atomicInteger = new AtomicInteger(100);
    private static AtomicStampedReference atomicStampedReference = new AtomicStampedReference(100,1);

    public static void main(String[] args) throws InterruptedException {

        //AtomicInteger
        Thread at1 = new Thread(new Runnable() {
            @Override
            public void run() {
                atomicInteger.compareAndSet(100,110);
                atomicInteger.compareAndSet(110,100);
            }
        });

        Thread at2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(2);      // at1,执行完
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("AtomicInteger:" + atomicInteger.compareAndSet(100,120));
            }
        });

        at1.start();
        at2.start();

        at1.join();
        at2.join();

        //AtomicStampedReference

        Thread tsf1 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    //让 tsf2先获取stamp，导致预期时间戳不一致
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 预期引用：100，更新后的引用：110，预期标识getStamp() 更新后的标识getStamp() + 1
                atomicStampedReference.compareAndSet(100,110,atomicStampedReference.getStamp(),atomicStam
                atomicStampedReference.compareAndSet(110,100,atomicStampedReference.getStamp(),atomicStam
            }
        });
```

```
        Thread tsf2 = new Thread(new Runnable() {
            @Override
            public void run() {
                int stamp = atomicStampedReference.getStamp();

                try {
                    TimeUnit.SECONDS.sleep(2);       //线程tsf1执行完
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("AtomicStampedReference:" +atomicStampedReference.compareAndSet(100,12
            }
        });

        tsf1.start();
        tsf2.start();
    }

}
```

运行结果：

这里写图片描述

运行结果充分展示了AtomicInteger的ABA问题和AtomicStampedReference解决ABA问题。