对 volatile、compareAndSet、weakCompareAndSet 的一些思考

2017-12-20 ImportNew

> 来源：tomas家的小拨浪鼓 ，
> www.jianshu.com/p/55a66113bc54

最近在看AtomicIntegerFieldUpdater的时候看到了两个很有意思的方法：compareAndSet 和 weakCompareAndSet。下面主要针对这两个方法展开讨论。

基于 JDK 8

首先，我们知道AtomicIntegerFieldUpdater是一个基于反射的功能包，它可以实现针对于指定类中volatile int 字段的原子更新。

『 compareAndSet 』：

```
/**
 * Atomically sets the field of the given object managed by this updater
 * to the given updated value if the current value {@code ==} the
 * expected value. This method is guaranteed to be atomic with respect to
 * other calls to {@code compareAndSet} and {@code set}, but not
 * necessarily with respect to other changes in the field.
 *
 * @param obj An object whose field to conditionally set
 * @param expect the expected value
 * @param update the new value
 * @return {@code true} if successful
 * @throws ClassCastException if {@code obj} is not an instance
 * of the class possessing the field established in the constructor
 */
public abstract boolean compareAndSet(T obj, int expect, int update);
```

以原子的方式更新这个更新器所管理的对象(obj)的成员变量，并且将这个成员变量更新为给定的更新后的值(update)如果当前值等于期望值(expect)时。

当存在其他使用‘compareAndSet’或者‘set’的情况下，这个方法可以确保是原子的，但如果你用其他的方式去改变这个成员变量时(如，使用直接赋值的方式 field=newField)，那么它是不会遵循这个原子性的。

嗯，这个方法好理解，compareAndSet保证了：a) 只有field的值为expect时；b) 将field的值修改为update的值；这两步是原子完成的。同时field一定为一个volatile属性，而volatile保证了属性在线程间的可见性，以及防止了指令的重排序。(关于volatile下面还会进一步展开)。嗯，一切看起来都挺美好的。

然后，我们来看下另一个方法『weakCompareAndSet』：

```
/**
 * Atomically sets the field of the given object managed by this updater
 * to the given updated value if the current value {@code ==} the
 * expected value. This method is guaranteed to be atomic with respect to
 * other calls to {@code compareAndSet} and {@code set}, but not
 * necessarily with respect to other changes in the field.
 *
 * <p><a href="package-summary.html#weakCompareAndSet">May fail
 * spuriously and does not provide ordering guarantees</a>, so is
 * only rarely an appropriate alternative to {@code compareAndSet}.
 *
 * @param obj An object whose field to conditionally set
 * @param expect the expected value
 * @param update the new value
 * @return {@code true} if successful
 * @throws ClassCastException if {@code obj} is not an instance
 * of the class possessing the field established in the constructor
 */
public abstract boolean weakCompareAndSet(T obj, int expect, int update);
```

以原子的方式更新这个更新器所管理的对象(obj)的成员变量，并且将这个成员变量更新为给定的更新后的值(update)如果当前值等于期望值(expect)时。

当存在其他使用‘compareAndSet’或者‘set’的情况下，这个方法可以确保是原子的，但如果你用其他的方式去改变这个成员变量时(如，使用直接赋值的方式 field=newField)，那么它是不会遵循这个原子性的。

该方法可能可能虚假的失败并且不会提供一个排序的保证，所以它在极少的情况下用于代替compareAndSet方法。

第一次看weakCompareAndSet doc文档的说明时，我是困惑的。我并不清楚你说的“fail spuriously”和“not provide ordering guarantees”的确切含义。于是我查询了些相关资料。

首先，我从jdk 8 的官方文档的java.util.concurrent.atomic上找到这么二段话：

The atomic classes also support method weakCompareAndSet, which has limited applicability. On some platforms, the weak version may be more efficient than compareAndSet in the normal case, but differs in that any given invocation of the weakCompareAndSet method may return false spuriously (that is, for no apparent reason). A false return means only that the operation may be retried if desired, relying on the guarantee that repeated invocation when the variable holds expectedValue and no other thread is also attempting to set the variable will eventually succeed. (Such spurious failures may for example be due to memory contention effects that are unrelated to whether the expected and current values are equal.) Additionally weakCompareAndSet does not provide ordering guarantees that are usually needed for synchronization control. However, the method may be useful for updating counters and statistics when such updates are unrelated to the other happens-before orderings of a program. When a thread sees an update to an atomic variable caused by a weakCompareAndSet, it does not necessarily see updates to any other variables that occurred before the weakCompareAndSet. This may be acceptable when, for example, updating performance statistics, but rarely otherwise.

一个原子类也支持weakCompareAndSet方法，该方法有适用性的限制。在一些平台上，在正常情况下weak版本比compareAndSet更高效，但是不同的是任何给定的weakCompareAndSet方法的调用都可能会返回一个虚假的失败(无任何明显的原因)。一个失败的返回意味着，操作将会重新执行如果需要的话，重复操作依赖的保证是当变量持有expectedValue的值并且没有其他的线程也尝试设置这个值将最终操作成功。(一个虚假的失败可能是由于内存冲突的影响，而和预期值(expectedValue)和当前的值是否相等无关)。此外weakCompareAndSet并不会提供排序的保证，即通常需要用于同步控制的排序保证。然而，这个方法可能在修改计数器或者统计，这种修改无关于其他happens-before的程序中非常有用。当一个线程看到一个通过weakCompareAndSet修改的原子变量时，它不被要求看到其他变量的修改，即便该变量的修改在weakCompareAndSet操作之前。

weakCompareAndSet atomically reads and conditionally writes a variable but does not create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.

weakCompareAndSet实现了一个变量原子的读操作和有条件的原子写操作，但是它不会创建任何happen-before排序，所以该方法不提供对weakCompareAndSet操作的目标变量以外的变量的在之前或在之后的读或写操作的保证。

这二段话是什么意思了，也就是说weakCompareAndSet底层不会创建任何happen-before的保证，也就是不会对volatile字段操作的前后加入内存屏障。因为就无法保证多线程操作下对除了weakCompareAndSet操作的目标变量(该目标变量一定是一个volatile变量)之其他的变量读取和写入数据的正确性。

这里，需要对volatile进行一个较为详细的说明。这样大家就能更深刻的明白上面这段话的语义了。

volatile

volatile 的特性

volatile变量自身具有下列特性：

① 可见性/一致性：对一个 volatile 变量的读，总是能看到(任意线程)对这个 volatile 变量最后的写入。
② 原子性：对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++这种复合操作不具有原子性。

Q：volatile是如何保证可见性的了？

A：在多核处理器中，当进行一个volatile变量的写操作时，JIT编译器生成的汇编指令会在写操作的指令前在上一个“lock”前缀。“lock”前缀的指令在多核处理器下会引发了两件事情：

① 将当前处理器缓存行的数据会写回到系统内存。
② 这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效。

因此更确切的来说，因为操作缓存的最小单位为一个缓存行，所以每次对volatile变量自身的操作，都会使其所在缓存行的数据会写回到主存中，这就使得其他任意线程对该缓存行中变量的读操作总是能看到最新写入的值(会从主存中重新载入该缓存行到线程的本地缓存中)。当然，也正是因为缓存每次更新的最小单位为一个缓存行，这导致在某些情况下程序可能出现“伪共享”的问题。嗯，好像有些个跑题，“伪共享”并不属于本文范畴，这里就不进行展开讨论。

好了，目前为止我们已经了解volatile变量自身所具有的特性了。注意，这里只是volatile自身所具有的特性，而volatile对线程的内存可见性的影响比volatile自身的特性更为重要。

volatile 写-读建立的 happens before 关系

happens-before 规则中有这么一条：
volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

happens-before的这个规则会保证volatile写-读具有如下的内存语义：

volatile写的内存语义：

当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值刷新到主内存。

volatile读的内存语义：

当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

为了实现 volatile 的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。因为内存屏障是一组处理器指令，它并不由JVM直接暴露，因此JVM会根据不同的操作系统插入不同的指令以达成我们所要内存屏障效果。

从整体执行效率的角度考虑，JMM 选择了在每个 volatile 写的后面插入一个 StoreLoad 屏障。

StoreLoad屏障
指令示例：Store1; StoreLoad; Load2
确保Store1数据对其他处理器变得可见(指刷新到内存)先于Load2及所有后续装载指令的装载。StoreLoad Barriers会使该屏障之前的所有内存访问指令(存储和装载指令)完成之后，才执行该屏障之后的内存访问指令。StoreLoad Barriers是一个“全能型”的屏障，它同时具有其他3个屏障的效果(LoadLoad Barriers、StoreStore Barriers、LoadStore Barriers)

好了，到现在我们知道了volatile的内存语义(happens-before关系)会保证volatile写操作之前的读写操作不会被重排序到volatile写操作之后，并且保证了写操作后将线程本地内存(可能包含了多个缓存行)中所有的共享变量值都刷新到主内存中。这样其他线程总是能在volatile写操作后的读取操作中得到该线程中所有共享变量的正确值。这是volatile的happens-before关系(通过内存屏障实现)带给我们的结果。注意，这个和volatile变量自身的特性是不同的，volatile自身仅仅是保证了volatile变量本身的可见性。而volatile的happens-before关系则保证了操作不会被重排序同时保证了线程本地内存中所有共享变量的可见性。

好了，讨论到这里，我们重新来理解下weakCompareAndSet的实现语义。也就是说，weakCompareAndSet操作仅保留了volatile自身变量的特性，而出去了happens-before规则带来的内存语义。也就是说，weakCompareAndSet无法保证处理操作目标的volatile变量外的其他变量的执行顺序(编译器和处理器为了优化程序性能而对指令序列进行重新排序)，同时也无法保证这些变量的可见性。

源码实现

目前为止，我们已经能够明白compareAndSet方法和weakCompareAndSet方法的不同之处了。那么，接下来我们来看看这两个方法的具体实现：

```
public boolean compareAndSet(T obj, int expect, int update) {
    if (obj == null || obj.getClass() != tclass || cclass != null) fullCheck(obj);
    return unsafe.compareAndSwapInt(obj, offset, expect, update);
}

public boolean weakCompareAndSet(T obj, int expect, int update) {
    if (obj == null || obj.getClass() != tclass || cclass != null) fullCheck(obj);
    return unsafe.compareAndSwapInt(obj, offset, expect, update);
}
```

是的，你没有看错。这两个方法的实现完全一样。『unsafe.compareAndSwapInt(obj, offset, expect, update);』中就是调用native方法了：

```
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;
}
```

由此可见，在JDK8乃至之前的版本，weakCompareAndSet方法并没有被真是意义上的实现，目前该方法所呈现出来的效果与compareAndSet方法是一样的。

基于JDK 9

在JDK 9中 compareAndSet 和 weakCompareAndSet方法的实现有些许的不同

```
/**
 * Atomically updates Java variable to {@code x} if it is currently
 * holding {@code expected}.
 *
 * <p>This operation has memory semantics of a {@code volatile} read
 * and write.  Corresponds to C11 atomic_compare_exchange_strong.
 *
 * @return {@code true} if successful
 */
```

```
@HotSpotIntrinsicCandidate
public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);
```

① 底层调用的native方法的实现中，cmpxchgb指令前都会有“lock”前缀了(在JDK 8中，程序会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。只有在CPU是多处理器(multi processors)的时候，会添加一个lock前缀)。

② 同时多了一个@HotSpotIntrinsicCandidate注解，该注解是特定于Java虚拟机的注解。通过该注解表示的方法可能(但不保证)通过HotSpot VM自己来写汇编或IR编译器来实现该方法以提供性能。
它表示注释的方法可能（但不能保证）由HotSpot虚拟机内在化。如果HotSpot VM用手写汇编和/或手写编译器IR（编译器本身）替换注释的方法以提高性能，则方法是内在的。

也就是说虽然外面看到的在JDK9中weakCompareAndSet和compareAndSet底层依旧是调用了一样的代码，但是不排除HotSpot VM会手动来实现weakCompareAndSet真正含义的功能的可能性。

后记

嗯，关于compareAndSet与weakCompareAndSet两个方法的不同，看似可能是个“简单”的问题，但当我真的去探究它们的不同时，还是话费了我不少的时间，同时也让我对volatile有了更加深入的理解。这里关于CAS还有不少值得深入探讨的地方，值得再用一篇文章好好的进行叙说。关于JDK9的改变也是值得以后慢慢去探索的。

参考

- 《Java 并发编程的艺术》