深入理解Java内存模型（四）——volatile

作者 程晓明 程晓明 关注 82 他的粉丝 发布于 2013年2月6日. 估计阅读时间: 20 分钟 来 QCon上海2018 关注大数据平台技术选型、搭建、系统迁移和优化的经验。 72 讨论

volatile的特性

当我们声明共享变量为volatile后，对这个变量的读/写将会很特别。理解volatile特性的一个好方法是：把对volatile变量的单个读/写，看成是使用同一个监视器锁对这些单个读/写操作做了同步。下面我们通过具体的示例来说明，请看下面的示例代码：

```
class VolatileFeaturesExample {
    volatile long vl = 0L;  //使用volatile声明64位的long型变量

    public void set(long l) {
        vl = l;   //单个volatile变量的写
    }

    public void getAndIncrement () {
        vl++;    //复合（多个）volatile变量的读/写
    }


    public long get() {
        return vl;   //单个volatile变量的读
    }
}
```

假设有多个线程分别调用上面程序的三个方法，这个程序在语意上和下面程序等价：

```
class VolatileFeaturesExample {
    long vl = 0L;               // 64位的long型普通变量

    public synchronized void set(long l) {     //对单个的普通 变量的写用同一个监视器同步
        vl = l;
    }

    public void getAndIncrement () { //普通方法调用
        long temp = get();           //调用已同步的读方法
        temp += 1L;                  //普通写操作
        set(temp);                   //调用已同步的写方法
    }
    public synchronized long get() {
    //对单个的普通变量的读用同一个监视器同步
        return vl;
    }
}
```

如上面示例程序所示，对一个volatile变量的单个读/写操作，与对一个普通变量的读/写操作使用同一个监视器锁来同步，它们之间的执行效果相同。

监视器锁的happens-before规则保证释放监视器和获取监视器的两个线程之间的内存可见性，这意味着对一个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。

监视器锁的语义决定了临界区代码的执行具有原子性。这意味着即使是64位的long型和double型变量，只要它是volatile变量，对该变量的读写就将具有原子性。如果是多个volatile操作或类似于volatile++这种复合操作，这些操作整体上不具有原子性。

简而言之，volatile变量自身具有下列特性：

- 可见性。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。
- 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。

volatile写-读建立的happens before关系

上面讲的是volatile变量自身的特性，对程序员来说，volatile对线程的内存可见性的影响比volatile自身的特性更为重要，也更需要我们去关注。

从JSR-133开始，volatile变量的写-读可以实现线程之间的通信。

从内存语义的角度来说，volatile与监视器锁有相同的效果：volatile写和监视器的释放有相同的内存语义；volatile读与监视器的获取有相同的内存语义。

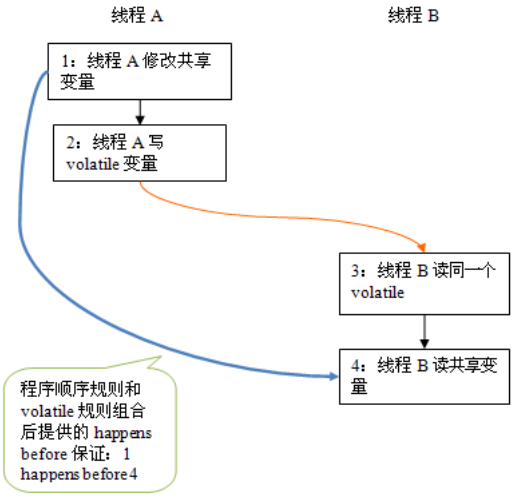
请看下面使用volatile变量的示例代码：

```
class VolatileExample {
    int a = 0;
    volatile boolean flag = false;

    public void writer() {
        a = 1;                   //1
        flag = true;               //2
    }

    public void reader() {
        if (flag) {                //3
            int i =  a;            //4
            ……
        }
    }
}
```

假设线程A执行writer()方法之后，线程B执行reader()方法。根据happens before规则，这个过程建立的happens before 关系可以分为两类：

1. 根据程序次序规则，1 happens before 2; 3 happens before 4。
2. 根据volatile规则，2 happens before 3。
3. 根据happens before 的传递性规则，1 happens before 4。

上述happens before 关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示volatile规则；蓝色箭头表示组合这些规则后提供的happens before保证。

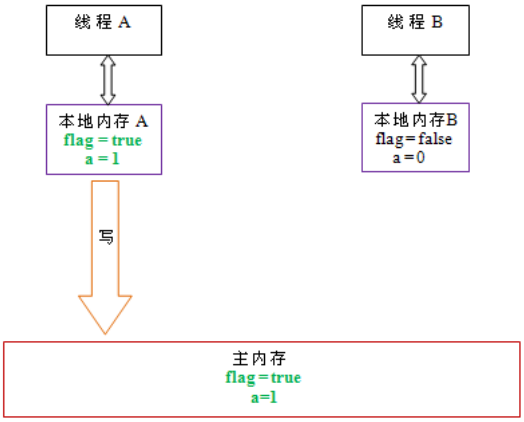
这里A线程写一个volatile变量后，B线程读同一个volatile变量。A线程在写volatile变量之前所有可见的共享变量，在B线程读同一个volatile变量后，将立即变得对B线程可见。

volatile写-读的内存语义

volatile写的内存语义如下：

- 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刷新到主内存。

以上面示例程序VolatileExample为例，假设线程A首先执行writer()方法，随后线程B执行reader()方法，初始时两个线程的本地内存中的flag和a都是初始状态。下图是线程A执行volatile写后，共享变量的状态示意图：

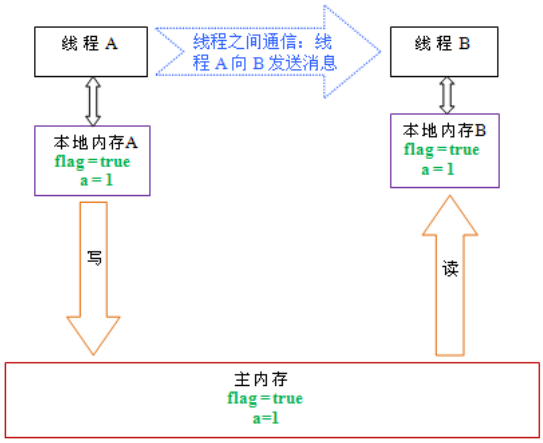


如上图所示，线程A在写flag变量后，本地内存A中被线程A更新过的两个共享变量的值被刷新到主内存中。此时，本地内存A和主内存中的共享变量的值是一致的。

volatile读的内存语义如下：

- 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。

下面是线程B读同一个volatile变量后，共享变量的状态示意图：



如上图所示，在读flag变量后，本地内存B已经被置为无效。此时，线程B必须从主内存中读取共享变量。线程B的读取操作将导致本地内存B与主内存中的共享变量的值也变成一致的了。

如果我们把volatile写和volatile读这两个步骤综合起来看的话，在读线程B读一个volatile变量后，写线程A在写这个volatile变量之前所有可见的共享变量的值都将立即变得对读线程B可见。

下面对volatile写和volatile读的内存语义做个总结：

- 线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所在修改的）消息。
- 线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。
- 线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。

volatile内存语义的实现

下面，让我们来看看JMM如何实现volatile写/读的内存语义。

前文我们提到过重排序分为编译器重排序和处理器重排序。为了实现volatile内存语义，JMM会分别限制这两种类型的重排序类型。下面是JMM针对编译器制定的volatile重排序规则表：

是否能重排序	第二个操作		
第一个操作	普通读/写	volatile读	volatile写
普通读/写			NO
volatile读	NO	NO	NO
volatile写		NO	NO

举例来说，第三行最后一个单元格的意思是：在程序顺序中，当第一个操作为普通变量的读或写时，如果第二个操作为volatile写，则编译器不能重排序这两个操作。

从上表我们可以看出：

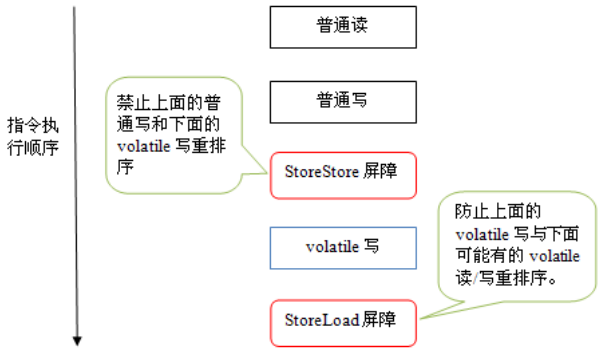
- 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。
- 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。
- 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

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。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，JMM采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

- 在每个volatile写操作的前面插入一个StoreStore屏障。
- 在每个volatile写操作的后面插入一个StoreLoad屏障。
- 在每个volatile读操作的后面插入一个LoadLoad屏障。
- 在每个volatile读操作的后面插入一个LoadStore屏障。

上述内存屏障插入策略非常保守，但它可以保证在任意处理器平台，任意的程序中都能得到正确的volatile内存语义。

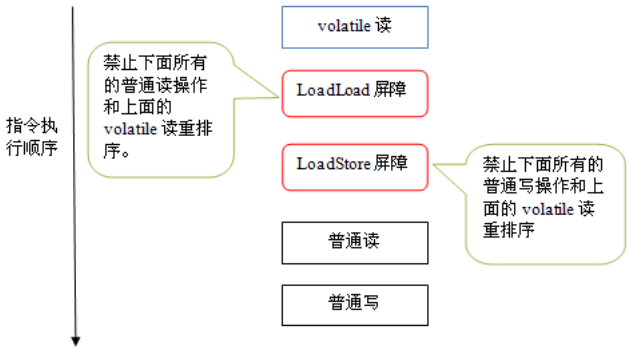
下面是保守策略下，volatile写插入内存屏障后生成的指令序列示意图：



上图中的StoreStore屏障可以保证在volatile写之前，其前面的所有普通写操作已经对任意处理器可见了。这是因为StoreStore屏障将保障上面所有的普通写在volatile写之前刷新到主内存。

这里比较有意思的是volatile写后面的StoreLoad屏障。这个屏障的作用是避免volatile写与后面可能有的volatile读/写操作重排序。因为编译器常常无法准确判断在一个volatile写的后面，是否需要插入一个StoreLoad屏障（比如，一个volatile写之后方法立即return）。为了保证能正确实现volatile的内存语义，JMM在这里采取了保守策略：在每个volatile写的后面或在每个volatile读的前面插入一个StoreLoad屏障。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。因为volatile写-读内存语义的常见使用模式是：一个写线程写volatile变量，多个读线程读同一个volatile变量。当读线程的数量大大超过写线程时，选择在volatile写之后插入StoreLoad屏障将带来可观的执行效率的提升。从这里我们可以看到JMM在实现上的一个特点：首先确保正确性，然后再去追求执行效率。

下面是在保守策略下，volatile读插入内存屏障后生成的指令序列示意图：

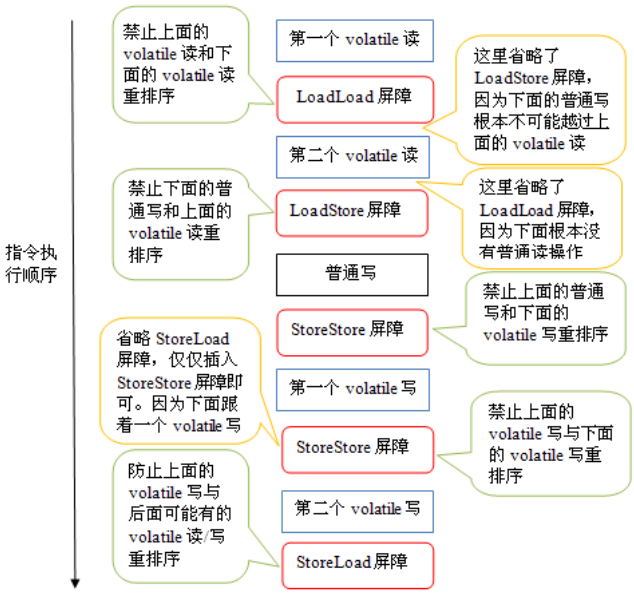


上图中的LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。

上述volatile写和volatile读的内存屏障插入策略非常保守。在实际执行时，只要不改变volatile写-读的内存语义，编译器可以根据具体情况省略不必要的屏障。下面我们通过具体的示例代码来说明：

```
class VolatileBarrierExample {
    int a;
    volatile int v1 = 1;
    volatile int v2 = 2;

    void readAndWrite() {
        int i = v1;           //第一个volatile读
        int j = v2;           // 第二个volatile读
        a = i + j;            //普通写
        v1 = i + 1;          // 第一个volatile写
        v2 = j * 2;          //第二个 volatile写
    }

    …                     //其他方法
}
```

针对readAndWrite()方法，编译器在生成字节码时可以做如下的优化：



注意，最后的StoreLoad屏障不能省略。因为第二个volatile写之后，方法立即return。此时编译器可能无法准确断定后面是否会有volatile读或写，为了安全起见，编译器常常会在这里插入一个StoreLoad屏障。

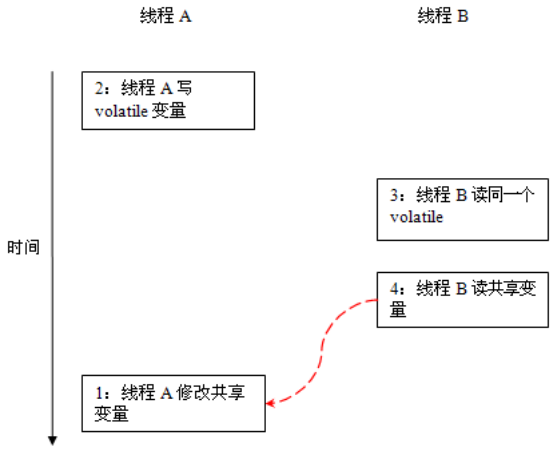
上面的优化是针对任意处理器平台，由于不同的处理器有不同“松紧度”的处理器内存模型，内存屏障的插入还可以根据具体的处理器内存模型继续优化。以x86处理器为例，上图中除最后的StoreLoad屏障外，其它的屏障都会被省略。

前面保守策略下的volatile读和写，在 x86处理器平台可以优化成：

前文提到过，x86处理器仅会对写-读操作做重排序。X86不会对读-读，读-写和写-写操作做重排序，因此在x86处理器中会省略掉这三种操作类型对应的内存屏障。在x86中，JMM仅需在volatile写后面插入一个StoreLoad屏障即可正确实现volatile写-读的内存语义。这意味着在x86处理器中，volatile写的开销比volatile读的开销会大很多（因为执行StoreLoad屏障开销会比较大）。

JSR-133为什么要增强volatile的内存语义

在JSR-133之前的旧Java内存模型中，虽然不允许volatile变量之间重排序，但旧的Java内存模型允许volatile变量与普通变量之间重排序。在旧的内存模型中，VolatileExample示例程序可能被重排序成下列时序来执行：



在旧的内存模型中，当1和2之间没有数据依赖关系时，1和2之间就可能被重排序（3和4类似）。其结果就是：读线程B执行4时，不一定能看到写线程A在执行1时对共享变量的修改。

因此在旧的内存模型中 ，volatile的写-读没有监视器的释放-获所具有的内存语义。为了提供一种比监视器锁更轻量级的线程之间通信的机制，JSR-133专家组决定增强volatile的内存语义：严格限制编译器和处理器对volatile变量与普通变量的重排序，确保volatile的写-读和监视器的释放-获取一样，具有相同的内存语义。从编译器重排序规则和处理器内存屏障插入策略来看，只要volatile变量与普通变量之间的重排序可能会破坏volatile的内存语意，这种重排序就会被编译器重排序规则和处理器内存屏障插入策略禁止。

由于volatile仅仅保证对单个volatile变量的读/写具有原子性，而监视器锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性。在功能上，监视器锁比volatile更强大；在可伸缩性和执行性能上，volatile更有优势。如果读者想在程序中用volatile代替监视器锁，请一定谨慎。

参考文献

1. Concurrent Programming in Java™: Design Principles and Pattern
2. JSR 133 (Java Memory Model) FAQ
3. JSR-133: Java Memory Model and Thread Specification
4. The JSR-133 Cookbook for Compiler Writers
5. Java 理论与实践: 正确使用 Volatile 变量
6. Java theory and practice: Fixing the Java Memory Model, Part 2

作者简介

程晓明，Java软件工程师，国家认证的系统分析师、信息项目管理师。专注于并发编程，就职于富士通南大。个人邮箱：asst2003@163.com。

感谢张龙对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家通过新浪微博（@InfoQ）或者腾讯微博（@InfoQ）关注我们，并与我们的编辑和其他读者朋友交流。