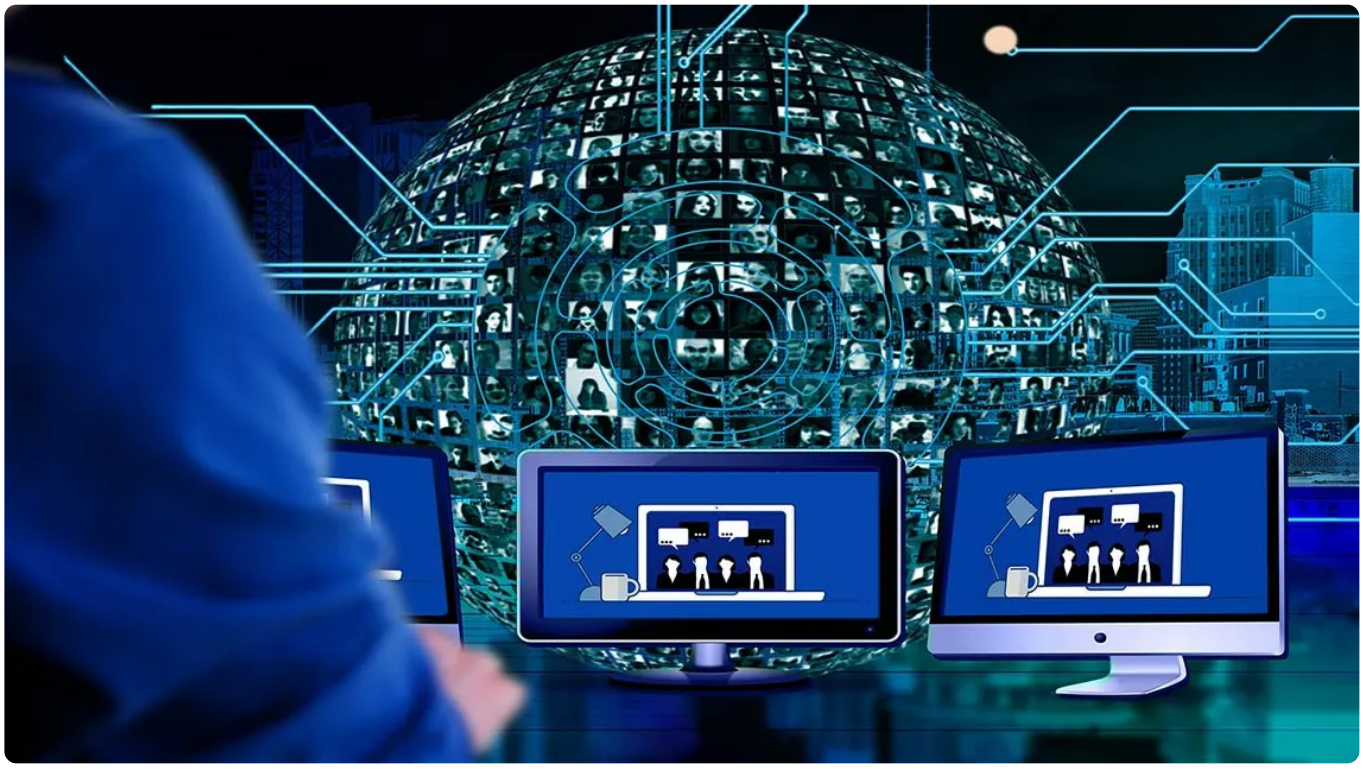
18 | Java内存模型：Java中的volatile有什么用？

2021-12-08 海纳

《编程高手必学的内存知识》 课程介绍 >



讲述：海纳

时长 19:27 大小 17.82M

你好，我是海纳。

随着这节课的开始，我们将进入到专栏的最后一个模块：**自动内存管理篇**。在这个模块，你将会了解到，以 Java 为代表的托管型语言是如何自动进行内存管理和回收整理的，这将提高你使用 Java、Python、 Go 等托管型语言的能力。

为什么我要把自动内存管理篇放到最后才讲呢？因为要理解这一篇的内容，需要软件篇和硬件篇的知识做铺垫。比如说，在面试时，有一个问题面试官问到的频率非常高，但几乎没有人能回答正确，因为它需要的前置知识太多。这个问题是：Java 中的 volatile 有什么用？如何正确地使用它？

这个问题之所以会频繁出现在面试中，是因为 Java 并发库中大量使用了 volatile 变量，在 JVM 的研发历史上，它在各种不同的体系结构下产生了很多典型的问题。那么，在开发并发程序的时候，深刻地理解它的作用是非常有必要的。

幸运的是，前面硬件篇的知识已经帮我们打好了足够的基础，今天我们就可以深入讨论这个问题了。由于在这个问题中，volatile 的语义是由 Java 内存模型定义的，我们就先从 Java 内存模型这个话题聊起。

Java 内存模型

我们知道在不同的架构上，缓存一致性问题是不同的，例如 x86 采用了 TSO 模型，它的**写后写（StoreStore）和读后读（LoadLoad）**完全不需要软件程序员操心，但是 Arm 的弱内存模型就要求我们自己在合适的位置添加 StoreStore barrier 和 LoadLoad barrier。例如下面这个例子：

复制代码

```
public class MemModel {
    static int x = 0;
    static int y = 0;
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                x = 1;
                y = 1;
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                while (y == 0);
                if (x != 1) {
                    System.out.println("Error!");
                }
            }
        });
        t2.start();
        t1.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
29      }
30  }
```

上面这个例子在 x86 机器上运行是没有问题的，但是在 Arm 机器就有概率打印出 Error。原因是第一个线程 t1 对变量 x 和 y 的写操作的顺序是不能保证顺序的，同时，第二个线程 t2 读取 x 和 y 的时候也不保证顺序。这一点我们在第 15 节课和第 16 节课 已经分析过了。

为了解决这个问题，Java 语言在规范中做出了明确的规定，也就是在 JSR 133 文档中规定了 Java 内存模型。**内存模型是用来描述编程语言在支持多线程编程中，对共享内存访问的顺序。**所以显然，在上面例子中，线程间变量共享的情况，就可以借此来解决。

在 JSR133 文档中，这个内存模型有一个专门的名字，叫 **Happens-before**，它规定了一些同步动作的先后顺序。当然，这个规范也不是一蹴而就的，它也是经过了几次讨论和更新之后才最终定稿。所以，在早期的 JVM 实现中仍然存在一些弱内存相关的问题。这些问题我们很难称其为 bug，因为标准里的规定就有问题，虚拟机实现只是遵从了标准而已。

接下来，我们探寻一下 Happens-before 模型究竟会带来什么样的问题，这样你就能深刻体会到 volatile 存在的意义了。

Happens-before 模型

Java 内存模型（Java Memory Model， JMM）是通过各种操作来定义的，包括对变量的读写操作、加锁和释放锁，以及线程的启动和等待操作。JMM 为程序中的动作定义了一种先后关系，这些关系被称为 Happens-Before 关系。**要想保证操作 B 可以看到操作 A 的结果，A 和 B 就必须满足 Happens-Before 关系**，这个结论与 A 和 B 是否在同一线程中执行无关。

我们先来看 Happens-Before 的规则，然后再对它的特点进行分析。我们要明确 Happens-Before 模型所讨论的都是同步动作，包括加锁、解锁、读写 volatile 变量、线程启动和线程完成等。下面这几条规则中所说的操作都是指同步动作。见下面的表格：

规则名称	规则具体描述
程序顺序规则	同步动作的执行顺序和代码顺序一致，也就是说，如果程序中动作A在动作B之前，那么在运行时动作A将在动作B之前执行。这一点可以类比顺序一致性内存模型
监视器(Monitor)规则	同一个monitor，解锁发生在加锁前面
volatile变量规则	对volatile的写操作在对该变量的读操作之前执行
线程启动规则	线程启动是每个线程的第一个操作
线程结束规则	线程Thread2检测到线程Thread1已经结束，那么Thread1中的所有动作都一定已经执行
中断规则	当线程Thread2在线程Thread1上调用interrupt时，必须在Thread1检测到interupt之前执行
终结器规则	对象的构造函数必须在该对象的finalize方法之前执行完成
传递性规则	如果动作A先于动作B发生，B又先于动作C发生，那么可以确定A先于C发生

极客时间

Happens-Before 模型强调的是同步动作的先后关系，对于非同步动作，就没有任何的限制了。这节课的第一个例子，它里面的读写操作都是非同步动作，所以它在不同的体系结构上运行，会得到不同的结果。但这并不违反 JMM 的规定。

你要牢记一点的是，**JMM 是一种理论上的内存模型，并不是真实存在的。它是以具体的 CPU 的内存模型为基础的。**可能我这么说，你还是觉得比较抽象，现在，我们来看 JSR 133 文档中的两个令人费解的例子，你就能理解了。

第一个例子是控制流依赖，例子中包括了两个线程且变量 x 和 y 的初值都是 0。第一个线程的代码是：

复制代码

```
// Thread1
r1 = x;
if (r1 != 0)
  y = 1;
```

第二个线程的代码是：

复制代码

```
// Thread 2
r2 = y;
if (r2 != 0)
   x = 1;
```

由于存在控制流依赖，这两段代码中，第 4 行都不能提前到第 2 行之前执行。换句话说，到目前为止，所有的主流的 CPU 中，上面两段代码都会按照代码顺序执行。你可以推演一下，最终的运行结果一定是 r1 和 r2 的值都是 0。

但是 Happens-before 是一种理论模型，如果线程 1 中，y=1 先于 r1=x 执行，同时线程 2 中，x=1 先于 r2=y 执行，最后的结果，存在 r1 和 r2 的值都是 1 的可能性。理论上确实可能存在一种 CPU，当它进行分支预测投机执行的时候，投机的结果被其他 CPU 观察到。当然，实际中绝对不可能出现这样的 CPU，因为这意味着厂家花费了更多的精力为软件开发者带来了一个巨大的麻烦，而且由于核间同步通讯的要求，CPU 的性能还会下降。

第二个例子是数据流依赖。假设 x 和 y 的初值是 0，而 r1 和 r2 的初值是 42。线程 1 的代码是：

复制代码

```
r1 = x;
y = r1;
```

线程 2 的代码是：

复制代码

```
r2 = y;
x = r2;
```

因为每个线程内部的第 2 行和第 1 行之间都存在数据依赖，所以这里是无法产生乱序执行的，所以无论你以怎样的顺序对这两个线程进行调度，都不可能出现 r1=r2=42 的情况的。但是 r1=r2=42 在 Happens-before 模型中却是合理的，因为它没有对数据流依赖进行规定。

也就是说，**普通的变量读写在 JMM 是允许乱序的**，如果真的有人造出这么愚蠢的 CPU，运行出这种结果却是符合 Happens-before 的规定的。

但是这两个问题在现实中并不存在。我这里特别想讲这两个例子的原因，是因为 JSR 133 文档花费了大量的篇幅在介绍本不应该存在的两个问题，这导致这个文档极其晦涩难懂。

从实用的角度，我建议你**在理解 JMM 时，一定要结合具体的 CPU 体系结构**。大体上讲，JMM 加上每一种体系结构都有的控制流依赖和数据流依赖，才是一种比较实用的内存模型。纯粹的 JMM 本身的实用性并不强。

JMM 是一种标准规定，它并不管实现者是如何实现它的。具体到 Java 语言虚拟机的实现，当 Java 并发库的核心开发者 Doug Lea 将 JMM 简化之后，就变得更容易理解一些。我们来看 Doug Lea 的描述。

JVM 的具体实现

Doug Lea 为了方便虚拟机开发人员理解 Java 内存模型，编写了一个名为《Java 内存模型 Cookbook》的小册子。在这个小册子中，他给出一个表格，**现代的 JVM 基本都是按照这个表格来实现的**。

	普通读	普通写	volatile读 进入monitor	volatile写 退出monitor
普通读				LoadStore
普通写				StoreStore
volatile读 进入monitor	LoadLoad	LoadStore	LoadLoad	LoadStore
volatile写 退出monitor			StoreLoad	StoreStore

极客时间

这个表格描述了连续的两个读写动作，JVM 应该如何处理。表格的最左列代表了第一个动作，第一行代表了第二个动作。表格中的内容使用了 LoadLoad、LoadStore、StoreStore、StoreLoad 四种内存屏障，分别表示第一个动作和第二个动作之间应该插入什么类型的内存屏障。

在上一节课中，我们知道了，在不同的体系结构上，这四类 barrier 的实际含义并不相同。因为 x86 采用了 **TSO 模型**，所以它根本没有定义 LoadLoad、LoadStore 和 StoreStore 这三种 barrier，x86 上只有 StoreLoad barrier 是有意义的。

而 Arm 上，由于存在单向的 barrier，所以 LoadLoad 和 LoadStore barrier 就可以使用 acquire load 代替，LoadStore 和 StoreStore barrier 也可以使用 release store 代替，而 StoreLoad barrier 就只能使用 dmb 代替了。

我们可以看到，表格的第三行刚好就对应了 arm 的 acqure load barrier，所以我们就知道在 arm 上，JVM 在实现 volatile 读的时候就必然会使用 acquire load 来代替。表格的第四列则刚好对应 arm 的 release store barrier，同时，arm 上的 JVM 在实现 volatile 写的时候，就可以使用 release store 来代替。

回到这节课开始时的例子，可见，只要将变量 y 改成 volatile，就相当于在第 8 行和第 9 行之前增加了 StoreStore barrier，同时在第 15 行和第 16 行处增加了 LoadLoad barrier，那么这段 Java 代码在 Arm 上的效果就与上一节课所分析的内存屏障的示例代码逻辑是一致的了。

只要 JVM 遵守了 JMM 的规定，那么不管在什么平台上，最后的运行结果都是一样。在这节课刚开始的那个例子中，只要把变量 y 修改成 volatile 修饰的，就不会再出现在 x86 上不会打印 Error，而在 Arm 有机率打印 Error 的情况了。**所有平台运行结果的一致性是由 JVM 遵守 JMM 来保证的**。

到这里，我们就知道了，Happens-before 模型是一种理论模型，它没有规定控制流依赖和数据流依赖。但是在实际的 CPU 中，这两种依赖都是存在的，这是 JVM 实现的基础。所以在 JVM 的实现中，主要是参考了 Doug Lea 所写的 Cookbook 中的建议。**从实用的角度，Java 程序员就可以从 Doug Lea 所给出的表格去理解 volatile 的意义，而不必再去参考 JSR 133 文档。**

接下来，我们通过两个例子来进一步加深对 Java 内存模型的理解，看看 Java 内存模型在实际场景中是如何应用的。

JMM 应用举例一：AQS

与这节课第一个例子相似，JDK 的源代码中有很多使用 volatile 变量的读写来保证代码执行顺序的例子，我们以 CountDownLatch 来举例，它有一个内部类是 Sync，它的定义如下所示：

复制代码

```
private static final class Sync extends AbstractQueuedSynchronizer {
    Sync(int count) {
        setState(count);
    }

    int getCount() {
        return getState();
    }

    protected int tryAcquireShared(int acquires) {
        return (getState() == 0) ? 1 : -1;
    }

    protected boolean tryReleaseShared(int releases) {
        // Decrement count; signal when transition to zero
        for (;;) {
            int c = getState();
            if (c == 0)
                return false;
            int nextc = c-1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
}
```

我们看到代码里的 tryAcquireShared 代表这个方法具有 acquire 语义，而 tryReleaseShared 则代表了这个方法具有 release 语义。从 tryAcquireShared 的代码里，我们可以推测 getState 里面应该会有 acquire 语义，我们继续看 AbstractQueuedSynchronizer 的代码。

复制代码

```
public abstract class AbstractQueuedSynchronizer
```

```
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {

    /**
     * The synchronization state.
     */
    private volatile int state;

    /**
     * Returns the current value of synchronization state.
     * This operation has memory semantics of a {@code volatile} read.
     * @return current state value
     */
    protected final int getState() {
        return state;
    }

    /**
     * Sets the value of synchronization state.
     * This operation has memory semantics of a {@code volatile} write.
     * @param newState the new state value
     */
    protected final void setState(int newState) {
        state = newState;
    }

    /**
     * Atomically sets synchronization state to the given updated
     * value if the current state value equals the expected value.
     * This operation has memory semantics of a {@code volatile} read
     * and write.
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful. False return indicates that the act
     *         value was not equal to the expected value.
     */
    protected final boolean compareAndSetState(int expect, int update) {
        // See below for intrinsics setup to support this
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
    }
```

从上面的代码中可以看到，state 是一个 volatile 变量，根据 JMM 模型，我们可以知道 getState 方法是一种带有 acquire 语义的读。

在为 state 变量赋值的时候，AbstractQueuedSynchronizer(AQS) 提供了两个方法，一个是 setState，另一个是 compareAndSetState。其中 setState 是一个带有 release 语

义的写。那为什么还要提供 comareAndSet 方法呢？

这是因为 compareAndSetState，不仅是强调 release 语义，它还有原子性语义。这个操作中包含了**取值，比较和赋值三个动作**，如果比较操作不成功，则赋值操作不会发生。

通过这个例子，我们就可以得到一个结论，**内存屏障与原子操作是两个不同的概念。内存屏障强调的是可见性，而原子操作则是强调多个步骤要么都完成，要么都不做。也就是说一个操作中的多个步骤是不能存在有些完成了，有些没完成的状态的。**

接下来，我们再举一个与内存模型相关的并发例子。这是一道十几年来经久不衰的面试题，也是 Java 面试官最喜欢问的。这道题是：如何高效地实现线程安全的单例模式？

JMM 应用举例二：线程安全的单例模式

我们知道单例模式是设计模式的一种，它的主要特点是全局只能生成唯一的对象。如何才能写出线程安全的单例模式代码呢？

我们从单线程最基本的单例模式开始讲起，它的代码是这样的：

复制代码

```
class Singleton {
    private static Singleton instance;

    public int a;

    private Singleton() {
      a = 1;
    }

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

这个类的特点是，**构造函数是私有的**。这意味着，除了在 getInstance 这个静态方法里，可以使用“new Singleton”的方式进行对象的创建，整个工程中的其他任意位置都不能再使用这种方法进行创建。

要想得到 Singleton 的实例就只能使用 getInstance 这个静态方法。而这个方法每一次都会返回同一个对象。所以这就保证了全局只能产生一个 Singleton 实例。

这个单例模式看上去写得很正确，但是面试题中的要求是写出线程安全的单例模式。上面的写法显然不是线程安全的。为什么我这么说呢？

假设第一个线程调用 getInstance 时，看到 instance 变量的值为 null，它就会创建一个新的 Singleton 对象，然后将其赋值给 instance 变量。当第二个线程随后调用 getInstance 时，它仍然有可能看到 instance 变量的值为 null，然后也创建一个新的 Singleton 对象。更具体的过程希望你可以自己进行分析，因为这是并发程序的相关内容，不是我们这节课的重点，所以我就不啰嗦了。

为了解决这个问题，我们可以将 getInstance 方法改为同步方法，这样就为调用这个方法加上了锁，从而可以保证线程安全：

复制代码

```
class Singleton {
    private static Singleton instance;
    public int a;
    private Singleton() {
        a = 1;
    }
    public synchronized static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

显然，上面的代码是线程安全的，我们之前分析过，在线程 1 还未执行完 getInstance，线程 2 就开始执行的情况，在加锁以后就不会再出现了。但是这样会带来新的问题：**访问加锁的方法是非常低效的**。

所以，又有另外一种实现方式被提出：

复制代码

```
class Singleton {
    private static Singleton instance = new Singleton();
```

```
3
4       public int a;
5       private Singleton() {
6           a = 1;
7       }
8
9       public static Singleton getInstance() {
10          return instance;
11      }
12  }
```

上面代码的第二行是在类加载的时候执行的，而类加载过程是线程安全的，所以不管有多少线程调用 getInstance 方法，它的返回值都是第二行所创建的对象。

这种创建方式有别于第一种。**第一种单例模式的实现是在第一次调用 getInstance 时，它是在不得不创建的时候才去创建新的对象，所以这种方式被称为懒汉式；第二种实现则是在类加载时就将对象创建好了，所以这种方式被称为饿汉式。**

还有的人既想使用懒汉式进行创建，又希望程序的效率比较好，所以提出了双重检查 (Double Check)，它的具体实现方案如下：

复制代码

```
1  class Singleton {
2      // 非核心代码略
3
4      public static Singleton getInstance() {
5          if (instance == null) {
6              synchronized (Singleton.class) {
7                  if (instance == null)
8                      instance = new Singleton();
9              }
10         }
11         return instance;
12     }
13 }
```

大多数情况下，instance 变量的值都不为 null，所以这个方法大多数时候都不会走到加锁的分支里。如果 instance 变量值为 null，则通过在 Singleton.class 对象上进行加锁，来保证对象创建的正确性，看上去这个实现非常好。

但是经过我们这节课的讲解，你就能理解这个写法在多核体系结构上还是会出现问题的。假设线程 1 执行到第 8 行，在创建 Singleton 变量的时候，由于没有 Happens-Before 的约束，所以 instance 变量和 instance.a 变量的赋值的先后顺序就不能保证了。

如果这时线程 2 调用了 getInstance，它可能看到 instance 的值不是 null，但是 instance.a 的值仍然是一个未初始化的值，也就是说线程 2 看到了 instance 和 instance.a 这两个变量的赋值乱序执行了。

这显然是一个写后写的乱序执行，所以修改的办法很简单：**只需要将 instance 变量加上 volatile 关键字，即可把这个变量的读变成 acquire 读，写变成 release 写**。这样，我们才真正地正确实现了饿汉式和懒汉式的单例模式。

总结

好啦，今天这节课就结束啦，这节课我们学习了 Java 内存模型。从这节课中，我们了解到 JSR 133 中所描述的 Java 内存模型是一种理论模型，它的规则非常少，以至于连控制流依赖和数据流依赖都没有规定，这导致 JSR133 文档讨论了很多在现实中根本不存在的情况。

而我们在讨论 JMM 的实现时，必然会与具体的 CPU 相联系。Doug Lea 将 JMM 做了简单的翻译，使用软件程序员可以看懂的语言重新阐释了 JSR 133 文档。

到这里，这节课开始处所讲的 volatile 的机制，其答案也就明晰了。它的作用是为变量的读写增加 happens before 关系，结合具体的 CPU 实现，就相当于是为变量的读增加 acquire 语义，为变量的写增加 release 语义。

接下来，我们用两个具体的例子来解释可见性、原子性和 volatile 的用法。

第一个例子是 Java 并发库中的核心数据结构 AbstractQueuedSynchronizer(AQS)，它通过使用 volatile 变量和原子操作来维护对象的状态。

第二个例子是实现线程安全的单例模式。我们梳理了单例模式的各种实现方式，并详细介绍了 double check 实现方式的问题，以及如何使用 volatile 来修复这个问题。

思考题

请你思考一下，volatile 能替代锁（或者 CAS 操作）的能力吗？比如，下面这个例子的写法，sum 的最终结果一定是 80000 吗？如果不是的话，应该怎么做才能保证呢？欢迎你在留言区分享你的想法和收获，我在留言区等你。

复制代码

```
class AddThread extends Thread {
    public void run() {
        for (int i = 0; i < 40000; i++)
            Main.sum += 1;
    }
}

class Main {
    public static volatile int sum = 0;
    public static void main(String[ ] args) throws Exception {
        AddThread t1 = new AddThread();
        AddThread t2 = new AddThread();
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(sum);
    }
}
```

吊打面试官

- 对比一下Java内存模型和C++内存模型？

我们这节课重点讲解了Java内存模型，本节课的Happens-Before模型和volatile自带的读写语义是Java内存模型的核心。所以在面试时如果遇到这个问题，这节课的内容就足以回答这个问题的前半部分。

而C++内存模型中最重要的也是理解acquire语义(memory_order_acquire)和release语义(memory_order_release)。这两个语义我们在上一节课介绍arm上的单向屏障时已经详细介绍过了。

C++内存模型里还有一点要注意的是acq_rel和csq的区别。acq_rel只保证单个线程中的变量读写顺序。但是csq还要保证多个线程中的变量读写顺序在所有其他线程看来都是一致的。这句话比较拗口，如果感觉很难理解，请参考C++标准（扫描右下角二维码跳转）中的最后一个例子，它比较生动地展示了这两种语义的区别。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 17 | NUMA：非均匀访存带来了哪些提升与挑战？

下一篇 19 | 垃圾回收：如何避免内存泄露？

精选留言 (5)

写留言

送过快递的码农
2021-12-09

我尝试对之前的知识和今天的文章我来梳理一下。volatile 关键字修饰的变量要遵循happen-bofore模型。由于cpu缓存的情况，我们会出现读滞后数据的情况。前面的知识，cpu通过缓存一致性协议，对缓存状态进行管理，一旦失效，会通过总线同步给其他核心。但是由于，这样性能成本太高，所以出现写缓存区+失效队列+内存屏障来进行补充？又由于我理解这个是jmm内存模型提出的规范，由于不同平台，不同cpu缓存架构不一样，cpu所...

展开

作者回复: 我觉得你这个总结还是感性上的一种总结，还不够精细。大体方向是对的。但细节还要再多抠一下。

共 2 条评论 >

raisecomer
2021-12-08

“happen-before模型”的表格中“对volatile的写操作在对该变量的读操作之前执行”，太令人费解了

.

2021-12-08

volatile作用的自我总结：
1.给编译器看在编译层面禁止重拍
2.给虚拟机看让其在对应的指令加入屏障。防止cpu级别的重排序与缓存一致性问题

思考题:...

展开

作者回复: 总结得非常好！15，16课已经讲过了啊，arm上就是采用了dmb, arquire, release这三种。

一子三木
2021-12-08

volatile 只能保证可见性和有序性。不能保证原子性。

作者回复: Bingo~

大豆
2021-12-08

1、volatile不能代替锁，它的作用是绕过高速缓存，直接与内存进行交互并读写数据。
2、当多线程时，会存在线程A将新值写入内存前，线程B又从内存读取了旧值，这样就会导致sum的值不会是80000。
3、要想sum的值为80000，还得给Main.sum += 1;这句代码加锁。
4、volatile还有一个作用是防止指令重排，对吗？老师。

展开

作者回复: 太对了，同学！

共 2 条评论 >