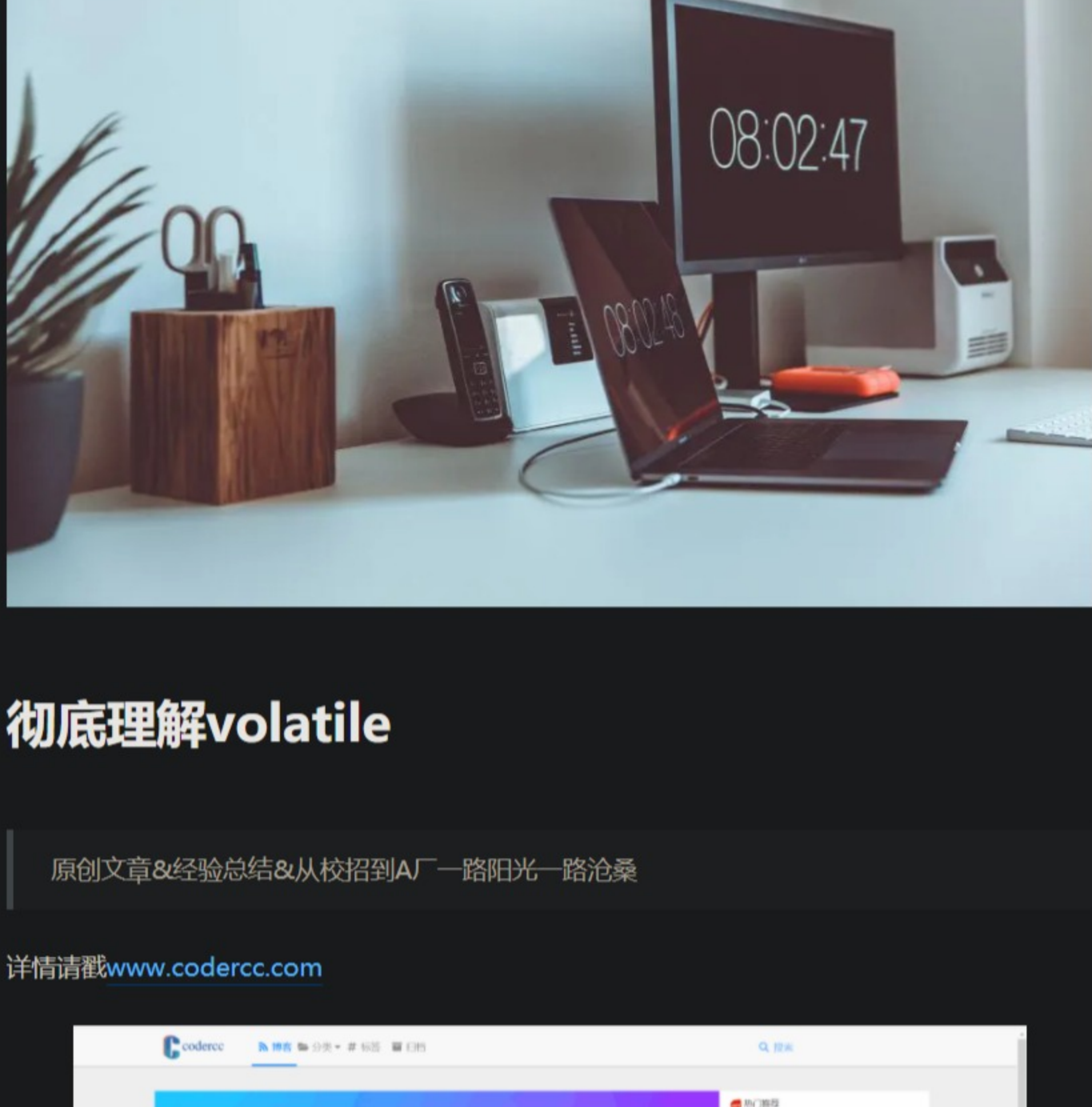


- 1. volatile简介
- 2. volatile实现原理
- 3. volatile的happens-before...  
文章阅读量 4,716
- 4. volatile的内存语义实现  
文章被阅读 429,033

相关文章

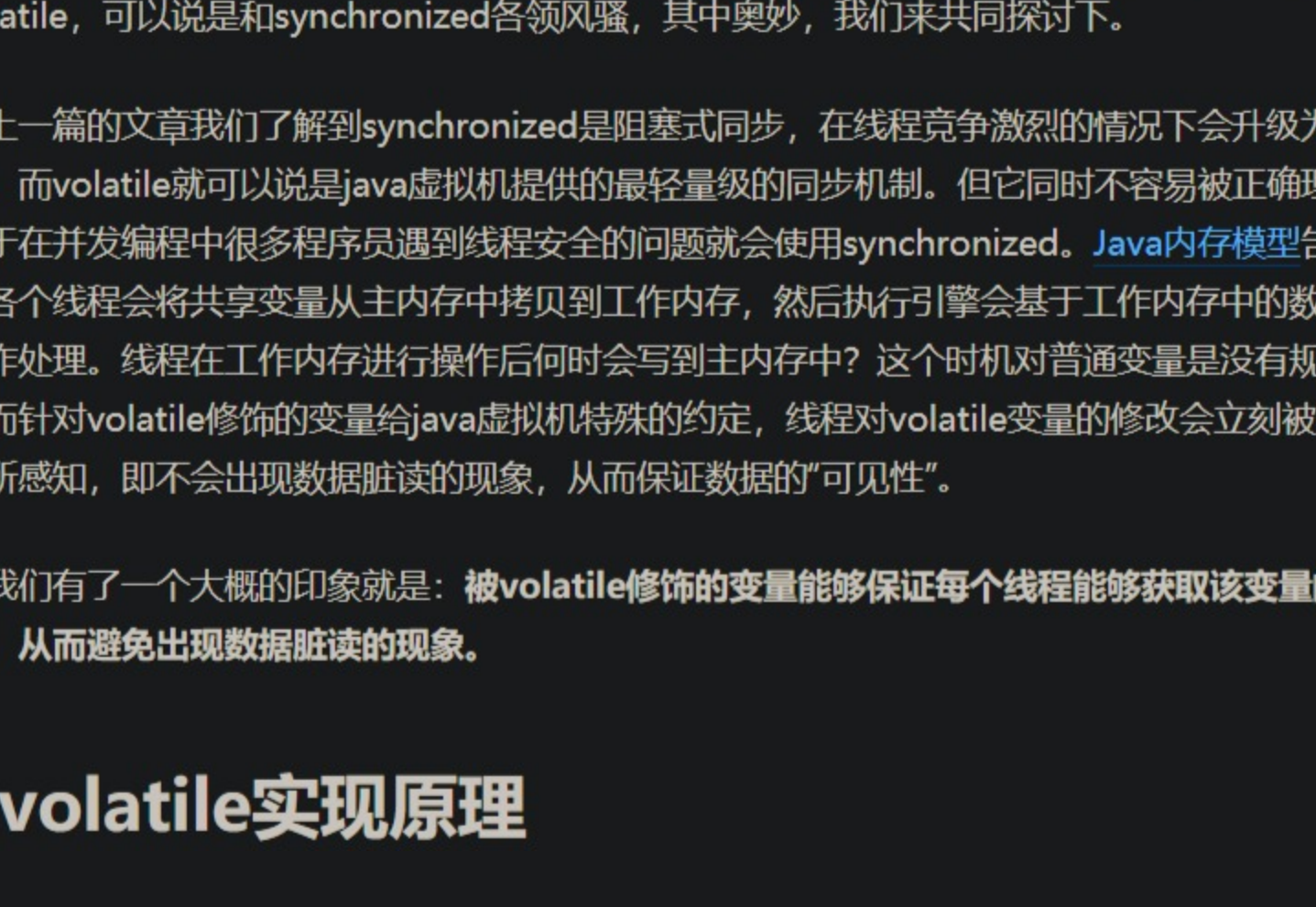
- git基本操作，一篇文章就够了！  
👍 996   👁 26
- MySQL命令，一篇文章替你全部搞定  
👍 1240   👁 16
- 彻底理解synchronized  
👍 317   👁 32
- Java内存模型以及happens-before原理  
👍 120   👁 12
- 深入理解AbstractQueuedSynchronizer(AQS)  
👍 120   👁 14



## 彻底理解volatile

原创文章&经验总结&从控制到大厂——阳光光一路沧桑

详情请戳[www.codercc.com](http://www.codercc.com)



### 1. volatile简介

在上一篇文章中我们深入理解了java关键字synchronized，我们知道在java中还有一个神器就是关键词volatile，可以说是和synchronized齐名双雄，其中 volatile，我们共同探讨一下。

通过上一篇文章我们了解到synchronized是阻塞式同步，在线程竞争激烈的情况下会升级为重量级锁，而volatile可以说是java虚拟机提供的轻量级的同步机制，但它同时还不容易被正确理解，也幸亏在开发编程中很多程序员遇到线程安全的问题就会使用synchronized。Java内存模型告诉我们，各个线程会共享变量从主存中拷贝到工作内存，然后运行引擎会基于工作内存中的数据进行操作处理，线程在工作内存进行操作时有时会写回主内存中；这个时机对普通变量没有规定的，而针对volatile修饰的变量给java虚拟机特殊的约定，线程对volatile变量的修改会立刻被其他线程所感知，即不会出现数据脏读的现象，从而保证数据的“可见性”。

现在我们有了一个大致的印象就是：被volatile修饰的变量能够保证每个线程都能够获取该变量的最新值，从而避免出现数据脏读的现象。

### 2. volatile实现原理

volatile是怎样实现的？比如一个很简单的Java代码：

```
instance = new Instance() //instance是volatile变量
```

在生成编译代码时会在volatile修饰的共享变量进行与操作的时候会多出**Lock前缀的指令**（具体的大家可以使用一些工具去看一下，这里我就只把结果说出来），我们把这个Lock指令肯定是有神功的地方，那么Lock前缀的指令在多线程环境下会发挥什么事情了？主要有这两个方面的影响：

1. 将当前处理器缓存中的数据与主内存中的数据同步；
2. 这个写回内存的操作会使得其他CPU里缓存了该内存地址的数据无效

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存（L1，L2或L3级）后再进行操作，但操作完不知道何时会与主内存同步。如果对声明了volatile的变量进行写操作，JVM就会向外界发送一条Lock前缀指令，将这个变量所在缓存中的数据与主内存同步。所以，在多线程下，为了保证各个处理器的缓存数据是一致的，就会实现**缓存一致性协议**，**每个处理器通过嗅探在总线上传播的数据来检查自己缓存数据是不是过期了**，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。因此，经过分析我们可以得出如下结论：

1. Lock前缀指令会引发处理器缓存与主内存同步；
2. 一个处理器的缓存与主内存同步会导致其他处理器缓存失效；
3. 当处理器发现本地缓存失效后，就会从主内存中重新读取数据，即可以获取当前最新值。

这样针对volatile变量通过这样的机制就使得每个线程都能够获得该变量的最新值。

### 3. volatile的happens-before关系

经过上面的分析，我们已经知道了volatile变量可以通过**缓存一致性协议**保证每个线程都能够获得最新值，即满足数据的“可见性”，我们继续延续上一篇文章的问题的方式（我一直认为思考问题的方式是属于自己的，也才最重要，也在不断培养这方面的能力），我一直将并发分析的切入点分为**两个核心，三大性质**。两大核心：JMM内存模型（主内存和工作内存）以及happens-before；三大性质：原子性、可见性、有序性（关于三大性质的总结以后博文文章会和大家共同探讨），废话不多说，先来看两个核心之一：volatile的happens-before关系。

在**Java的happens-before规则**中有一条是：“volatile变量规则：对一个volatile修饰的，happens-before于任意后续读对这个volatile修饰的。”下面我结合具体代码，我们利用这条规则推导一下：

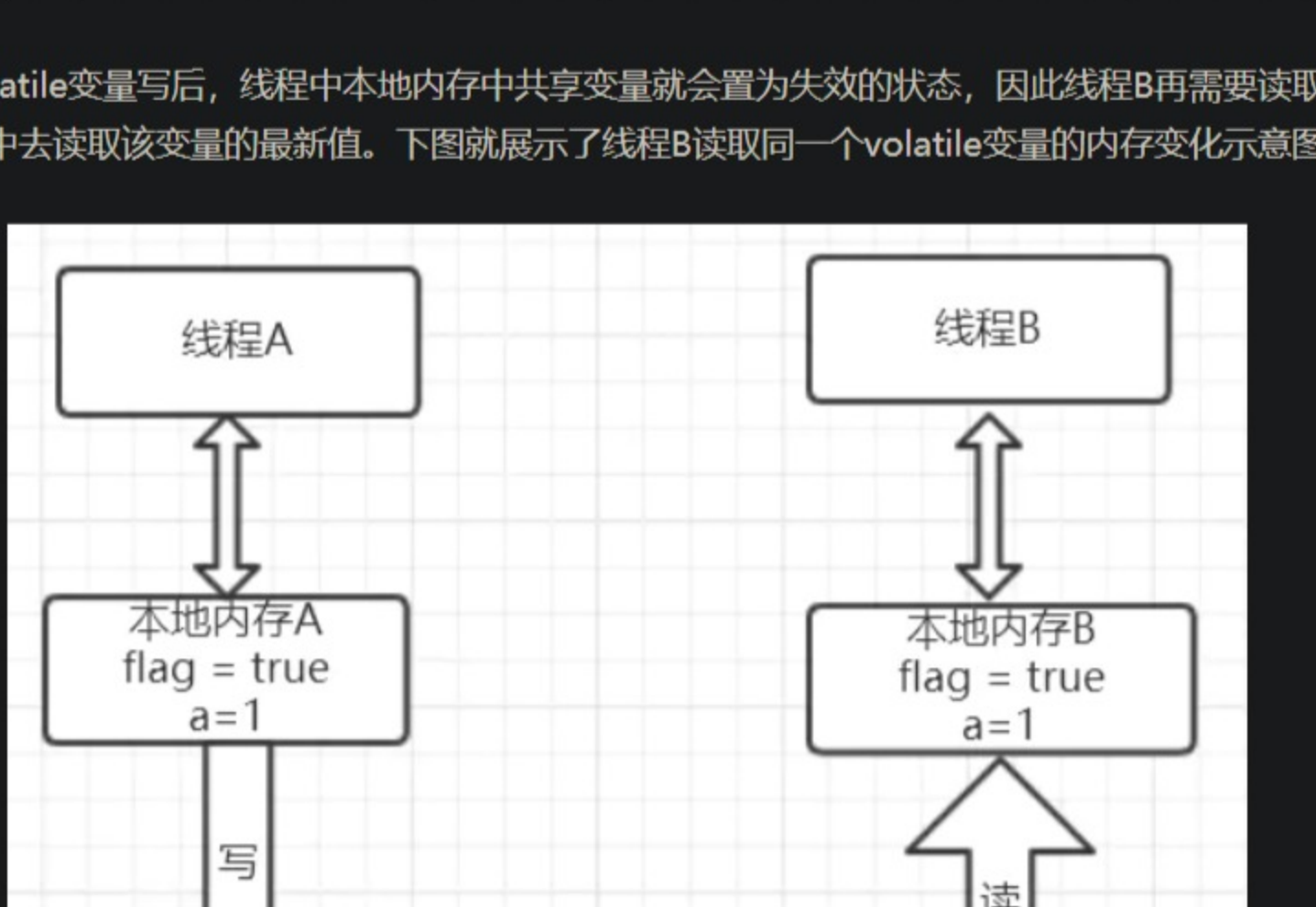
```
public class VolatileExample {
    private int a = 0;
    private volatile boolean flag = false;

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

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

复制代码

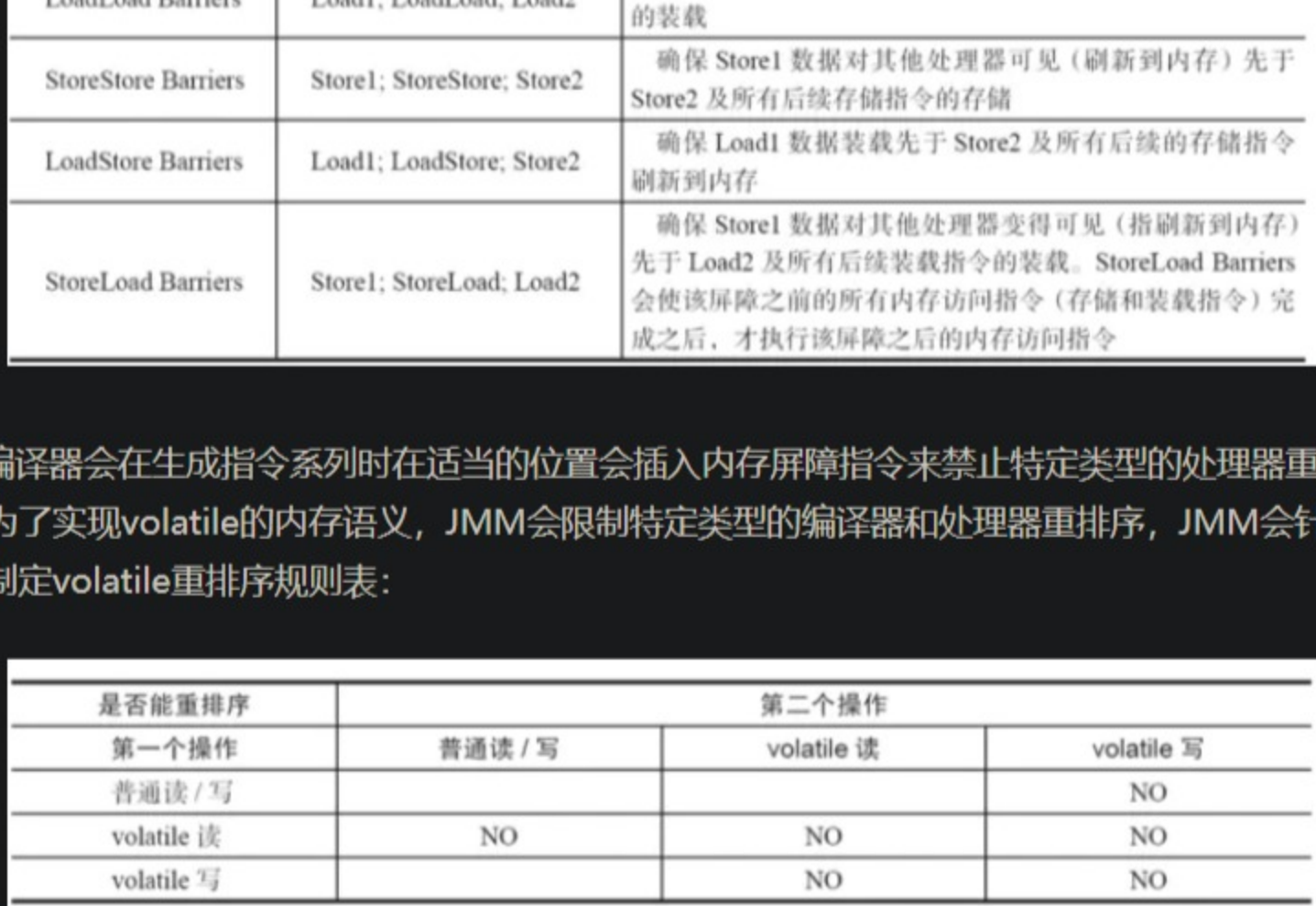
上面的代码对应对应的happens-before关系如下图所示：



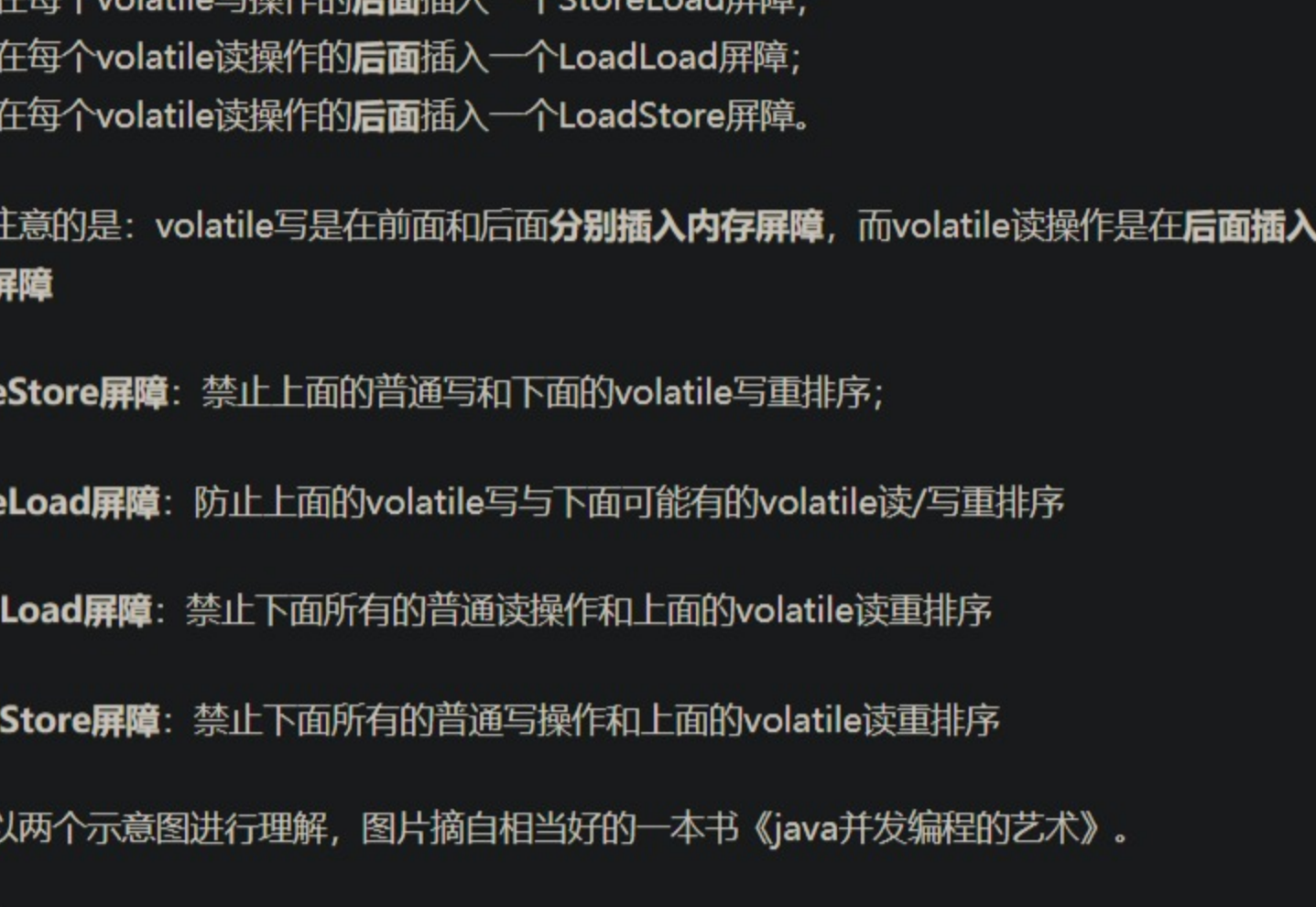
加载线程A先执行writer方法，然后线程B执行reader方法图中每一个箭头两个节点就代表一个happens-before关系，黑色的代表**按照程序顺序规则**推导出来，红色的就是根据**volatile变量的写happens-before于任意后续读对这个volatile修饰的**规则推导出来，而蓝色的就是根据**传播性规则**推导出来的。这里的happens-before 3，同样根据happens-before规则定义：如果A happens-before B,则A的执行结果对B可见，并且A的后续操作先于B的执行顺序，我们可以知道线程A之执行结果对线程B来说是可见的，也就是说当线程A将volatile变量 flag更改为true后线程B就能够迅速感知。

### 4. volatile的内存语义

还是按照**两个核心**的分析方式，分析完happens-before关系后我们现在就来进一步分析volatile的内存语义（按照这种方式去去学习，肯定不会让大家觉得知识能够多增多的知识，而在于去发现和创造，如果才去认同和接受这种方式，不如给个零，小弟在此谢过，对您是个鼓励），还是以上面的代码为例，假设线程A先执行writer方法，线程B随后执行reader方法，初始时线程的本地内存中flag都是初始状态，下图是线程A执行volatile写后的状态图。



当volatile变量写后，线程中本地内存中共享变量就会变为无效的状态，因此线程B再需要读取从主内存中去读取该变量的最新值。下图就展示了线程B读取下一个volatile变量的内存变化示意图。



从图上来看，线程A和线程B之间进行了一次通信，线程A在写volatile变量时，实际上就像是给B发送了一个消息告诉线程B你现在的值是正确的了，然后线程B读取这个volatile变量的时候就接收了线程A刚刚发送过的消息，既然正确的了，那线程B该怎么办了？自然而然就去主内存去读取。

好的，我们现在在**两个核心：happens-before**以及内存语义现在已经都了解清楚了，是不是还不够嘛，突然发现原来自己会这么爱学习（微笑脸），那我们下面再来一点干货——volatile内存语义的实现。

#### 4.1 volatile的内存语义实现

我们都知道，为了性能优化，JMM在不改变正确语义的前提下，会允许编译器和处理器对指令序列进行重排序，那如果想阻止重排序要怎么办呢？答案是可以添加内存屏障。

##### 内存屏障

JMM内存屏障分为四类见下图。

屏障类型	指令示例	说 明
LoadLoad Barrier	Load1; LoadLoad; Load2	前提 Load1 数据加载完成后先于 Load2 及所有后续数据加载指令的数据
StoreStore Barrier	Store1; StoreStore; Store2	前提 Store1 数据对其他处理器可见（刷新到内存）先于 Store2 及所有后续存储指令的数据
LoadStore Barrier	Load1; LoadStore; Store2	前提 Load1 数据加载完成后先于 Store2 及所有后续数据加载指令的数据
StoreLoad Barrier	Store1; StoreLoad; Load2	前提 Store1 数据对其他处理器可见（刷新到内存）先于 Load2 及所有后续数据加载指令的数据。StoreLoad Barrier 会使得该屏障之前的所有内存操作（存储和读取指令）完成之后，才执行屏障之后的内存操作指令

Java编译器在生成指令序列时在适当的位置会插入**禁止特定类型处理器和处理器重排序**，为了实现volatile的内存语义，JMM会限制特定类型的编译器和处理器重排序，JMM会对编译器和处理器重排序规则是：

第一个操作	第二个操作	volatile 读	volatile 写
普通读/写	普通读/写	NO	NO
volatile 读	NO	NO	NO
volatile 写	NO	NO	NO

“NO”表示禁止重排序，为了实现volatile内存语义时，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。对于编译原理来说，发现一个最分布重最小化插入屏障的总数几乎是不可能的，为此，JMM采取了保守策略：

1. 在每个volatile写操作的前面插入一个StoreStore屏障；
2. 在每个volatile写操作的后端插入一个StoreLoad屏障；
3. 在每个volatile读操作的前端插入一个LoadLoad屏障；
4. 在每个volatile读操作的后端插入一个LoadStore屏障。

需要注意的是：volatile写是在前面和后面分别插入**内存屏障**，而volatile读操作是在**后面插入两个内存屏障**

**StoreStore屏障**：禁止上面的普通写和下面的volatile写与重排序；

**StoreLoad屏障**：防止上面的volatile写与下面可能的volatile读/写重排序

**LoadLoad屏障**：禁止下面所有的普通读操作和上面的volatile读重排序

**LoadStore屏障**：禁止下面所有的普通写操作和上面的volatile读重排序

下面以两个示意图进行理解，图片摘自相当好的一本书《Java并发编程的艺术》。



### 5. 一个示例

我们现在已经理解volatile的精华了，文章开头的那个问题我想现在我们都能给出答案了，更正后的代码如下：

```
public class VolatileDemo {
    private static volatile boolean isOver = false;

    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (!isOver);
            }
        });
        thread.start();
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        isOver = true;
    }
}
```

复制代码

注意不同点，现在已经将**isOver设置成了volatile变量**，这样在main线程中将isOver改为了true后，thread的工作内存该变量值就会失效，从而需要再次从主内存中读取该值，现在能够读出isOver最新值为true从而能够结束死循环，从而能够顺利停止该thread线程。现在问题也解决了，知识也学到了！。（如果觉得还不错，请点赞，是对我的一个鼓励。）

##### 参考文献

《Java并发编程的艺术》

文章分类

后端

文章标签

后端

Java

程序员

编译器

你听...  
发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

关注

安装掘金浏览插件

打开新标签页及发现好内容，掘金、GitHub、Dribbble、ProductHunt等站内容轻松获取，快来安装掘金浏览器插件获取高质量内容吧！

输入评论...

发表评论

嘿嘿嘿哈哈哈

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

9月前

会发生了不能影响了1和4之间的关系了吗

👍 1

回复

嘿嘿嘿哈哈哈

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

9月前

因为StoreStore屏障不会发生重排序

👍

回复

我是小敏敏

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

9月前

有个小疑问：文中有关“为了保证各个处理器的缓存是一致的...”针对这句话，个人觉得有个漏洞没得到多体现：在单线程处理时，多线程应用是否还需要保证数据的可见性，即数据还是需要volatile修饰？

👍

回复

后浪Coder

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

12月前

需要，操作本身就是为了保证内存可见性

👍

回复

Hook技术

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

12月前

👍

回复

spilledyear

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

大佬，那个例子有点问题，为什么在while(true) 加上一条打印日志，就不成立了，如下，下面代码就可以跳出死循环

public class VolatileDemo {  
 private static boolean isOver = false;...

展开

👍 2

回复

GodsCoder

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

System.out.println("测试1: run1");  
while (!isOver);  
System.out.println("测试2: run2");  
}

👍 11

回复

之之之 高级工程师

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

回复 GodsCoder 哥

👍

回复

解解 代码刷子 @ weconex

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

👍

回复

Shely

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

啥才成立了呢？

👍

回复

tc7879665

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

1月前

回复 GodsCoder: 牛逼

👍

回复

左麟135度 风控系统 Java 工程师 @ Linzi

发布了60 篇专栏 · 获得点赞 4,716 · 获得阅读 429,013

2月前

@Override  
public void run() {  
 try {  
 Thread.sleep(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println("测试1: run1");  
 while (!isOver);  
 System.out.println("测试2: run2");  
 }  
}  
thread.start();  
try {  
 Thread.sleep(500);  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}  
isOver = true;  
这样也可以，是因为执行到while (!isOver);才去取值么

收起

👍

回复

相关推荐

MarkerHub · 4小时前 · 后端

我啊，2天前 · 程序员

Antalk · 16小时前 · Java

Java面试，1天前 · Java

谈谈我裸辞以及一周内找到工作的经历

华为云开发者社区 · 6小时前 · 后端

Golang：后端开发中的万能药吗？

黑小狼训练营 · 1天前 · 前端 / 后端 / Android / 程序员 / iOS

技术助力！掘金小册免费学

程序员鱼皮 · 1天前 · Java

整理完 140 多套 Java 完整实战项目，各个精品！

zimay · 7小时前 · Java

java并发编程工具类JUC第六篇SynchronousQueue同步队列

halla啊子 · 22小时前 · Java

Android基础系列（6）Java反射

程序员内参 · 23小时前 · Java / 后端

干掉前端！3分钟纯 Java 搭建个管理系统，我直接好家伙