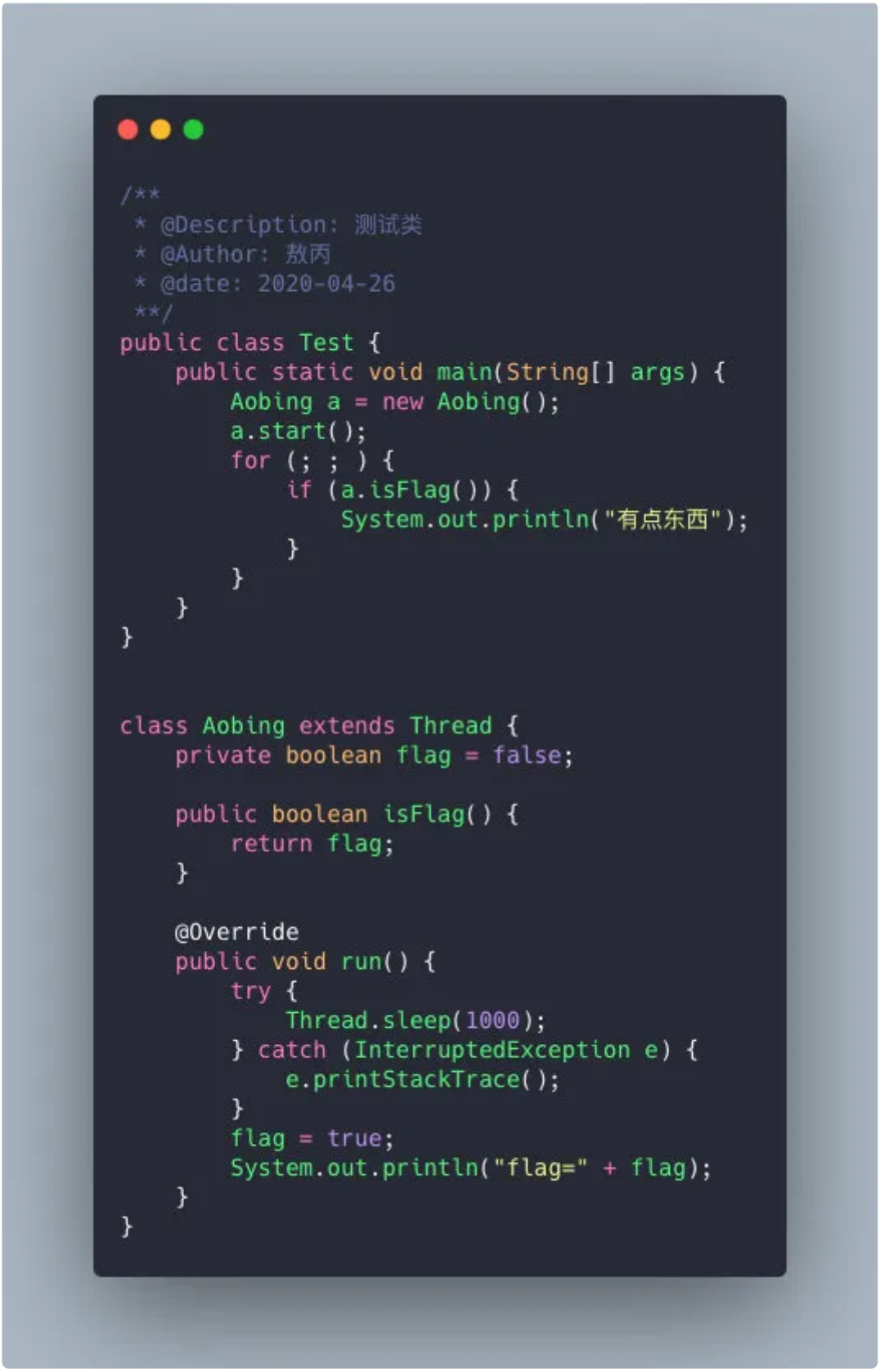
面试官想到，一个Volatile，敖丙都能吹半小时

原创 三太子敖丙 三太子敖丙 4月29日

收录于话题
#多线程 253 #程序员 1905

Volatile可能是面试里面必问的一个话题吧，对他的认知很多朋友也仅限于会用阶段，今天我们换个角度去看看。

先来跟着丙丙来看一段demo的代码：



```java
/**
 * @Description: 测试类
 * @Author: 敖丙
 * @date: 2020-04-26
 **/
public class Test {
    public static void main(String[] args) {
        Aobing a = new Aobing();
        a.start();
        for (; ; ) {
            if (a.isFlag()) {
                System.out.println("有点东西");
            }
        }
    }
}


class Aobing extends Thread {
    private boolean flag = false;

    public boolean isFlag() {
        return flag;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        flag = true;
        System.out.println("flag=" + flag);
    }
}
```

你会发现，永远都不会输出有点东西这一段代码，按道理线程改了flag变量，主线程也能访问到的呀？

为会出现这个情况呢？那我们就需要聊一下另外一个东西了。

JMM（JavaMemoryModel）

JMM：Java内存模型，是java虚拟机规范中所定义的一种内存模型，Java内存模型是标准化的，屏蔽掉了底层不同计算机的区别（注意这个跟JVM完全不是一个东西，只有还有小伙伴搞错的）。

那正式聊之前，丙丙先大概科普一下现代计算机的内存模型吧。

现代计算机的内存模型

其实早期计算机中cpu和内存的速度是差不多的，但在现代计算机中，cpu的指令速度远超内存的存取速度，由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（Cache）来作为内存与处理器之间的缓冲。

将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新的问题：缓存一致性（CacheCoherence）。

在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（MainMemory）。

然后我们可以聊一下JMM了。

JMM

`Java内存模型(JavaMemoryModel)` 描述了Java程序中各种变量(线程共享变量)的访问规则，以及在JVM中将变量，存储到内存和从内存中读取变量这样的底层细节。

JMM有以下规定：

所有的共享变量都存储于主内存，这里所说的变量指的是实例变量和类变量，不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。

每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。

`线程对变量的所有的操作(读，取)都必须在工作内存中完成，而不能直接读写主内存中的变量` 。

不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存中转来完成。

本地内存和主内存的关系：

正是因为这样的机制，才导致了可见性问题的存在，那我们就讨论下可见性的解决方案。

可见性的解决方案

加锁

为啥加锁可以解决可见性问题呢？

因为某一个线程进入synchronized代码块前后，线程会获得锁，清空工作内存，从主内存拷贝共享变量最新的值到工作内存成为副本，执行代码，将修改后的副本的值刷新回主内存中，线程释放锁。

而获取不到锁的线程会阻塞等待，所以变量的值肯定一直都是最新的。

Volatile修饰共享变量

开头的代码优化完之后应该是这样的：

Volatile做了啥？

每个线程操作数据的时候会把数据从主内存读取到自己的工作内存，如果他操作了数据并且写会了，他其他已经读取的线程的变量副本就会失效了，需要都数据进行操作又要再次去主内存中读取了。

volatile保证不同线程对共享变量操作的可见性，也就是说一个线程修改了volatile修饰的变量，当修改写回主内存时，另外一个线程立即看到最新的值。

是不是看着加一个关键字很简单，但实际上他在背后含辛茹苦默默付出了不少，我从计算机层面的缓存一致性协议解释一下这些名词的意义。

之前我们说过当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致，举例说明变量在多个CPU之间的共享。

如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？

为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有MSI、`MESI（IllinoisProtocol）`、MOSI、Synapse、Firefly及DragonProtocol等。

聊一下Intel的MESI吧

MESI（缓存一致性协议）

当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

至于是怎么发现数据是否失效呢？

嗅探

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的值是不是过期了，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。

嗅探的缺点不知道大家发现了没有？

总线风暴

由于Volatile的MESI缓存一致性协议，需要不断的从主内存嗅探和cas不断循环，无效交互会导致总线带宽达到峰值。

所以不要大量使用Volatile，至于什么时候去使用Volatile什么时候使用锁，根据场景区分。

我们再来聊一下 指令重排序 的问题

禁止指令重排序

什么是重排序?

为了提高性能，编译器和处理器常常会对既定的代码执行顺序进行指令重排序。

重排序的类型有哪些呢？源码到最终执行会经过哪些重排序呢？

一个好的内存模型实际上会放松对处理器和编译器规则的束缚，也就是说软件技术和硬件技术都为同一个目标，而进行奋斗：在不改变程序执行结果的前提下，尽可能提高执行效率。

JMM对底层尽量减少约束，使其能够发挥自身优势。

因此，在执行程序时，为了提高性能，编译器和处理器常常会对指令进行重排序。

一般重排序可以分为如下三种：

- 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序;
- 指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序;
- 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行的。

这里还得提一个概念， `as-if-serial` 。

as-if-serial

不管怎么重排序，单线程下的执行结果不能被改变。

编译器、runtime和处理器都必须遵守as-if-serial语义。

那**Volatile**是怎么保证不会被执行重排序的呢？

内存屏障

java编译器会在生成指令系列时在适当的位置会插入 内存屏障 指令来禁止特定类型的处理器重排序。

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会针对编译器制定volatile重排序规则表：

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。

写

读

上面的我提过重排序原则，为了提高处理速度，JVM会对代码进行编译优化，也就是指令重排序优化，并发编程下指令重排序会带来一些安全隐患：如指令重排序导致的多个线程操作之间的不可见性。

如果让程序员再去了解这些底层的实现以及具体规则，那么程序员的负担就太重了，严重影响了并发编程的效率。

从JDK5开始，提出了 happens-before 的概念，通过这个概念来阐述操作之间的内存可见性。

happens-before

如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在happens-before关系。

volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

如果现在我的变了flag变成了false，那么后面的那个操作，一定要知道我变了。

聊了这么多，我们要知道Volatile是没办法保证原子性的，一定要保证原子性，可以使用其他方法。

无法保证原子性

就是一次操作，要么完全成功，要么完全失败。

假设现在有N个线程对同一个变量进行累加也是没办法保证结果是对的，因为读写这个过程并不是原子性的。

要解决也简单，要么用原子类，比如AtomicInteger，要么加锁(记得关注Atomic的底层)。

应用

单例有8种写法，我说一下里面比较特殊的一种，涉及Volatile的。

> 大家可能好奇为啥要双重检查？如果不用**Volatile**会怎么样？

我先讲一下 禁止指令重排序 的好处。

对象实际上创建对象要进过如下几个步骤：

- 分配内存空间。
- 调用构造器，初始化实例。
- 返回地址给引用

上面我不是说了嘛，是可能发生指令重排序的，那有可能构造函数在对象初始化完成前就赋值完成了，在内存里面开辟了一片存储区域后直接返回内存的引用，这个时候还没真正的初始化完对象。

但是别的线程去判断instance！=null，直接拿去用了，其实这个对象是个半成品，那就有空指针异常了。

可见性怎么保证的？

因为可见性，线程A在自己的内存初始化了对象，还没来得及写回主内存，B线程也这么做了，那就创建了多个对象，不是真正意义上的单例了。

上面提到了volatile与synchronized，那我聊一下他们的区别。

volatile与synchronized的区别

volatile只能修饰实例变量和类变量，而synchronized可以修饰方法，以及代码块。

volatile保证数据的可见性，但是不保证原子性(多线程进行写操作，不保证线程安全);而synchronized是一种排他(互斥)的机制。

volatile用于禁止指令重排序：可以解决单例双重检查对象初始化代码执行乱序问题。

volatile可以看做是轻量版的synchronized，volatile不保证原子性，但是如果是对一个共享变量进行多个线程的赋值，而没有其他的操作，那么就可以用volatile来代替synchronized，因为赋值本身是有原子性的，而volatile又保证了可见性，所以就可以保证线程安全了。

总结

1. volatile修饰符适用于以下场景：某个属性被多个线程共享，其中有一个线程修改了此属性，其他线程可以立即得到修改后的值，比如booleanflag;或者作为触发器，实现轻量级同步。
2. volatile属性的读写操作都是无锁的，它不能替代synchronized，因为它没有提供原子性和互斥性。因为无锁，不需要花费时间在获取锁和释放锁_上，所以说它是低成本的。
3. volatile只能作用于属性，我们用volatile修饰属性，这样compilers就不会对这个属性做指令重排序。
4. volatile提供了可见性，任何一个线程对其的修改将立马对其他线程可见，volatile属性不会被线程缓存，始终从主 存中读取。
5. volatile提供了happens-before保证，对volatile变量v的写入happens-before所有其他线程后续对v的读操作。
6. volatile可以使得long和double的赋值是原子的。
7. volatile可以在单例双重检查中实现可见性和禁止指令重排序，从而保证安全性。

注：以上所有的内容如果能全部掌握我想**Volatile**在面试官那是很加分了，但是我还没讲到很多关于计算机内存那一块的底层，那大家就需要后面去补课了，如果等得及，也可以等到我写计算机基础章节。

絮叨

因为更新文章和视频，丙丙已经半年多的周末没休息了，都是在公司那个工位冲冲冲，一直想找时间出去玩，想着年假一天没用，就请了两天出去玩一下。

这样五一就可以早点回来，准备恢复视频的更新，你在看的时候呢，敖丙应该在出游的列车上了，是的我就背了这个包，到写完的时候，我还没确定去哪里，提前祝大家节日愉

快。

我是敖丙，一个在互联网苟且偷生的工具人。

你知道的越多，你不知道的越多，人才们的 【三连】 就是丙丙创作的最大动力，我们下期见！