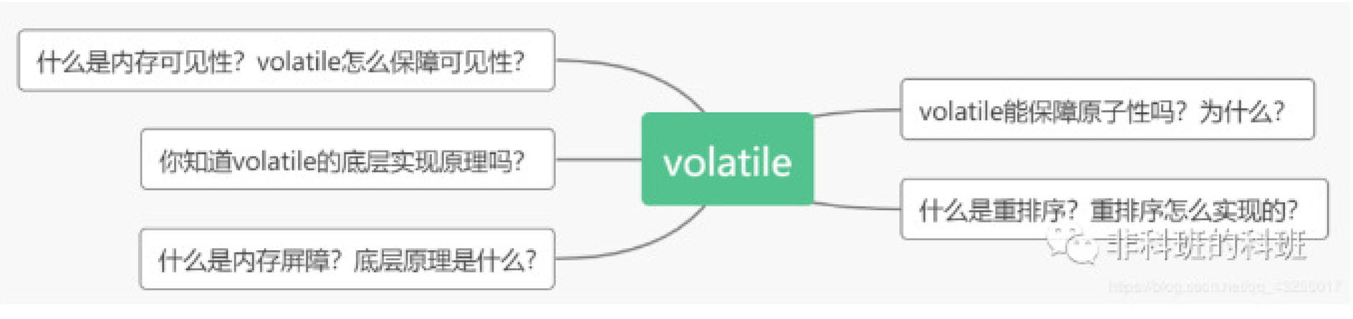
面试造飞机系列：volatile面试的连环追击，你还好吗？

程序员面试 5月13日

本文脑图

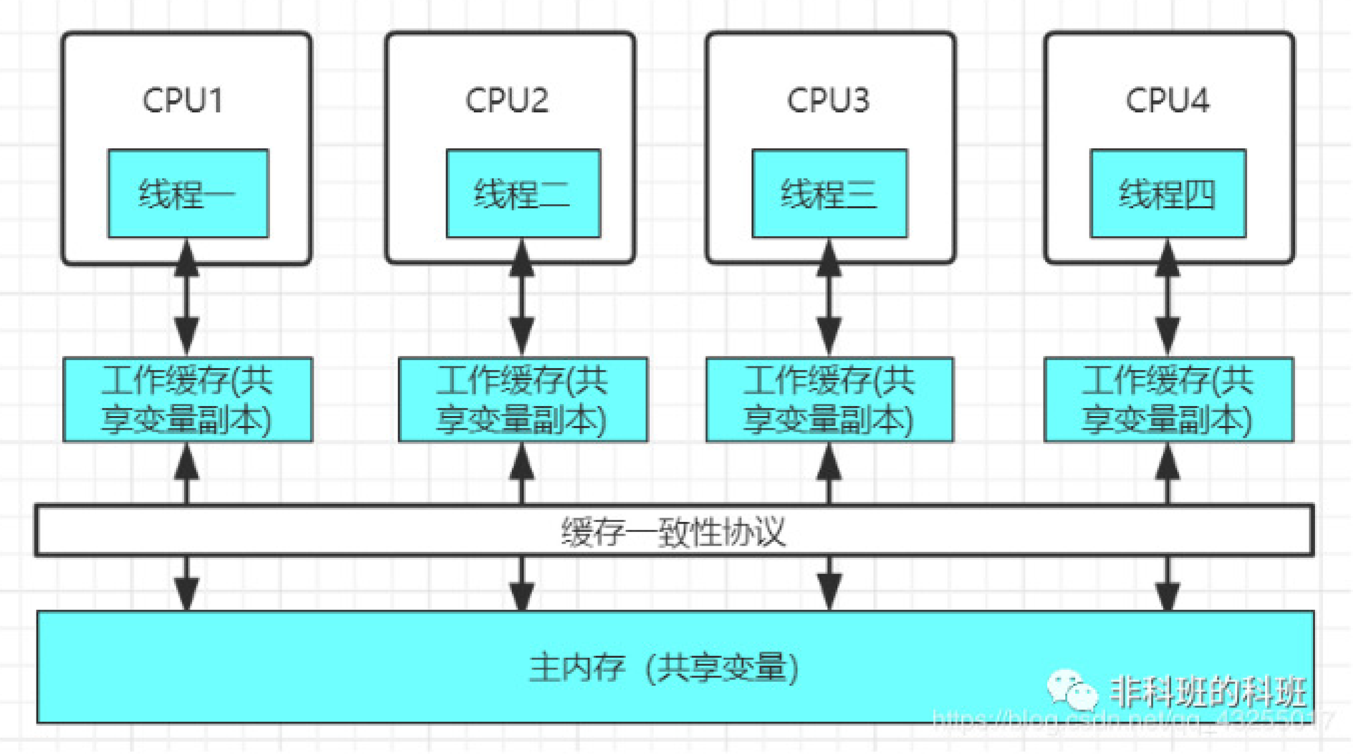


`volatile` 是 `java` 中热门**关键字**，也是面试中的**高频问点**，今天就来深入的从各种 `volatile` 面试题中剖析它的底层原理实现，并通过简单的代码去证明。

在深入 `volatile` 之前，我们先从原理入手，然后层层深入，逐步剖析它的底层原理，使用过 `volatile` 关键字的程序员都知道，在多线程并发场景中 `volitile` 能够保障共享变量的**可见性。**

那么问题来了，什么是可见性呢？volatile是怎么保障共享变量的可见性的呢？

在说**可见性**之前，我们先来了解在多线程的条件下，线程与线程之间是怎么通信的，我们先来看看一张图：

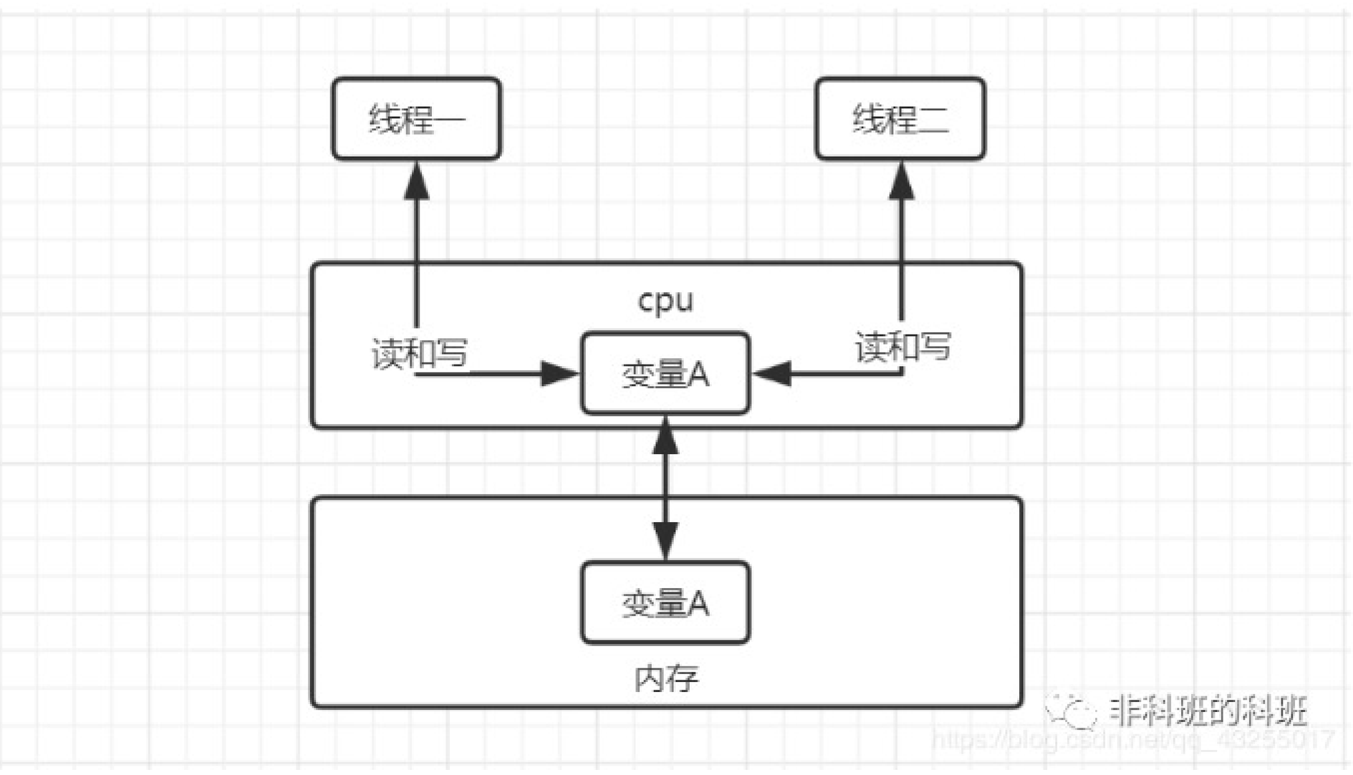


在Java线程中每次的**读取**和**写入**不会直接操作主内存，因为 `cpu` 的速度远快于主内存的速度，若是直接操作主内存，大大限制了cpu的性能，对性能有很大的影响，所以每条线程都有各自的**工作内存。**

这里的**工作内存**类似于**缓存**，并非实际存在的，因为缓存的读取和写入的速度远大于主内存，这样就大大提高了 `cpu` 与**数据交互的性能。**

所有的**共享变量**都是直接存储于**主内存**中，工作内存保存线程在使用主内存共享变量的**副本**，当操作完工作内存的变量，会写入主内存，完成对共享变量的读取和写入。

在单线程时代，不存在数据一致性的的问题，线程都是排队的顺序执行，前面的线程执行完才会到后面的线程执行。



随着计算机的发展，到了**多核多线程**的时代，**缓存**的出现虽然提升了 `cpu` 的执行效率，但是却出现了**缓存一致性**的问题，为了解决数据的一致性问题，提出两种解决方案：

1. **总线上加Lock#锁**：该方法简单粗暴，在总线上加锁，其它cpu的线程只能排队等候，效率低下。
2. **缓存一致性协议**：该方案是JMM中提出的解决方案，通过对变量地址加锁，减小锁的粒度，执行变得更加高效。

为了提高程序的执行效率，设计者们提出了底层对编译器和执行器（处理器）的优化方案，分别是**编译器**和**处理器**的**重排序**

那么什么是编译器重排序和处理器啊重排序呢？

编译器重排序就是在不改变单线程的语义的前提下，可以重新排列语句的执行顺序。

处理器排序是在机器指令的层面，假如不存在数据依赖，处理器可以改变机器指令的执行顺序，为了提高程序的执行效率，在多线程中假如两行的代码存在数据依赖，将会被禁止重排序。

不管是**编译器重排序**和**处理器的重排序**，前提条件都不能改变单线程语义的前提下进行重排序，说白了就是最后的**执行结果要准确无误。**

学过大学的计算机基础课都知道，我们的程序用**高级语言**写完后是不能被各大平台的机器所执行的，需要执行**编译**，然后将编译后的**字节码文件**处理成**机器指令**，才能被计算机执行。

从java源代码到最终的机器执行指令，分别会经过下面三种重排序：

在这里插入图片描述

前面说到了数据依赖的特性，什么是数据依赖呢？

数据依赖就是假设一句代码中对一个变量 `a++` 自增，然后后一句代码 `b=a` 将a的值赋值给b，便表示这两句代码存在数据依赖，两句代码执行顺序不能互换。

前面提到编译器和处理器的重排序，在编译器和处理器进行重排序的时候，就会遵守数据的依赖性，编译器和处理器就会禁止存在数据依赖的两个操作进行重排序，保证了数据的准确性。

在 `JDK5` 开始，为了保证程序的有序性，便提出了 `happen-before` 原则，假如两个操作符合该原则，那么这两个操作可以随意的进行重排序，并不会影响结果的正确性。

具体 `happen-before` 原则有6条，具体原则如下所示：

1. 同一个线程中前面的操作先于后续的操作（但是这个并不是绝对的，假如在单线程的环境下，重排序后不会影响结果的准确性，是可以进行重排序，不按代码的顺序执行）。
2. `Synchronized` 规则中解锁操作先于后续的加锁操作。
3. `volatile` 规则中写操作先于后续的读取操作，保证数据的可见性。
4. 一个线程的 `start()` 方法先于任何该线程的所有后续操作。
5. 线程的所有操作先于其他该线程在该线程上调用join返回成功的操作。
6. 如果操作a先于操作b，操作b先于操作c，那么操作a先于操作c,传递性原理。

我们来看重点第三条，也就是我们今天所了解的重点volatile关键字，为了实现volatile内存语义，规定有volatile修饰的共享变量在机器指令层面会出出现Lock前缀的指令。

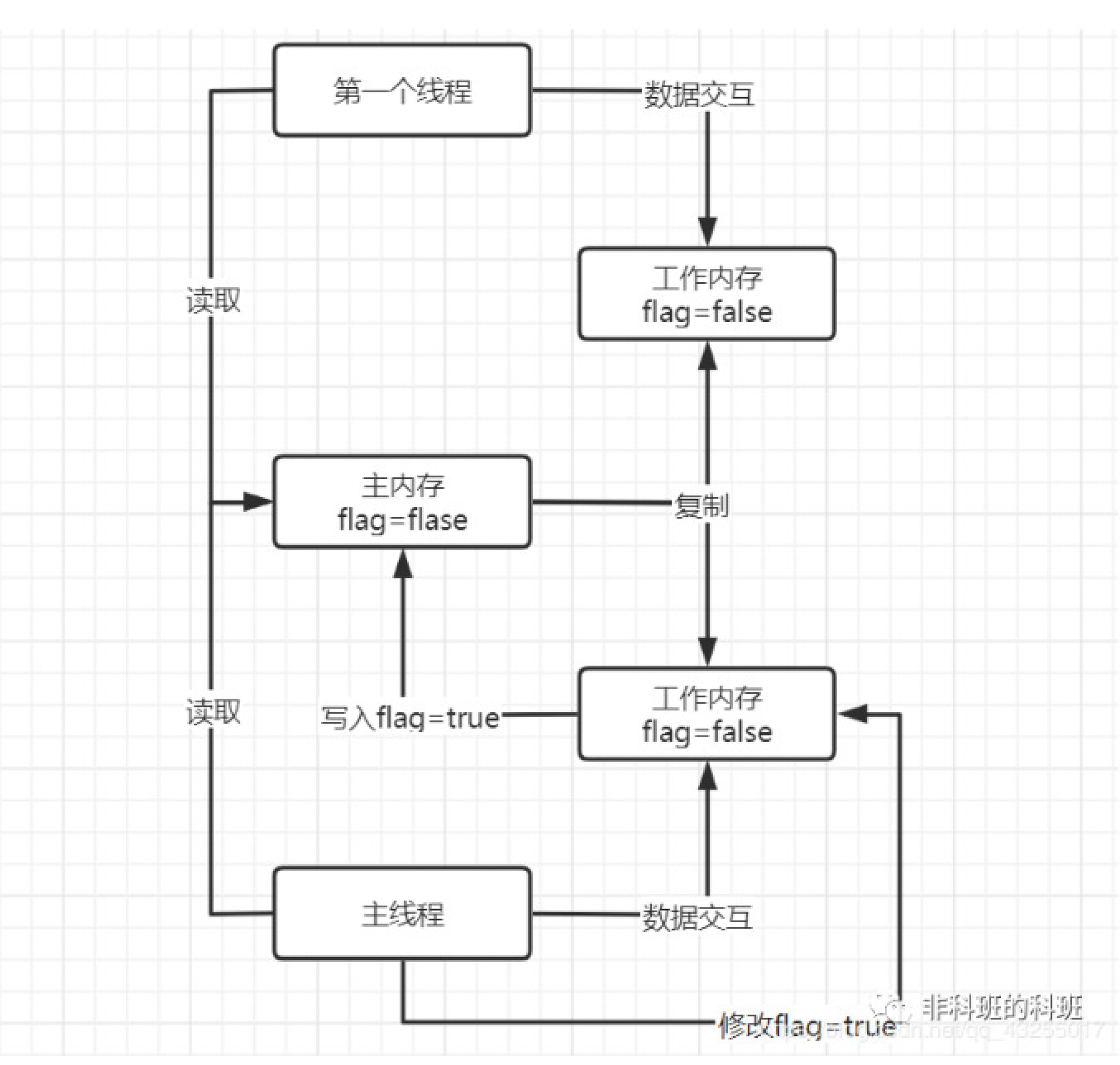
我们来看看一个例子经典的例子，具体的代码如下：

```java
public class TestVolatile extends Thread {
    private static boolean flag = false;

    public void run() {
        while (!flag) ;
        System.out.println("run方法退出了")
    }

    public static void main(String[] args) throws Exception {
        new TestVolatile().start();
        Thread.sleep(5000);
        flag = true;
    }
}
```

看上面的代码执行run方法能执行退出吗？是不能的，因为对于这两个线程来说，首先 `new TestVolatile().start()` 线程拿到 `flag` 共享变量的值为false，并存储在于自己的工作内存中。

第一个线程到while循环中，就直接进入死循环，即使主线程读取flag的值，然后改变该值为true。

但是对于第一个线程来说并不知道，flag的值已经被修改，在第一个线程的工作内存中flag仍然为false。具体的执行原理图如下：



这样对于共享变量flag，主线程修改后，对于线程1来说是不可见的，然后我们加上volatile变量修饰该变量，修改代码如下：

```java
private static volatile boolean flag = false;
```

输出的结果中，就会输出run方法退出了，具体的原理假如一个共享变量被 `Volatile` 修饰，该指令在多核处理器下会引发两件事情。

1. 将当前处理器缓存行数据写回主内存中。
2. 这个写入的操作会让其它处理器中已经缓存了该变量的内存地址失效，当其它处理器需求再次使用该变量时，必须从主内存中重新读取该值。

让我们具体从idea的输出的**汇编指令**中可以看出，我们看到红色线框里面的那行指令：`putstatic flag`，将静态变量 `flag` 入栈，注意观察add指令前面有一个 `lock` 前缀指令。

> 注意：让idea输出程序的汇编指令，在启动程序的时候，可以加上 `-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly` 作为启动参数，就可以查看汇编指令。

简单的说被volatile修饰的共享变量，在lock指令后是一个原子操作，该原子操作不会被其它线程的调度机制打断，该原子操作一旦执行就会运行到结束，中间不会切换到任意一个线程。

当使用lock前缀的机器指令，它会向cpu发送一个LOCK#信号，这样能保证在多核多线程的情况下互斥的使用该共享变量的内存地址。直到执行完毕，该锁定才会消失。

volatile的底层就是通过**内存屏障**来实现的，**lock前缀指令**就相当于一个**内存屏障。**

那么什么又是内存屏障呢？

内存屏障是一组 `CPU` 指令，为了提高程序的运行效率，编译器和处理器运行对指令进行重排序，JMM为了保证程序运行结果的准确性，规定存在**数据依赖**的机器指令**禁止重排序。**

通过插入特定类型的**内存屏障**（例如**lock前缀指令**）来禁止特定类型的编译器重排序和处理器重排序，插入一条内存屏障会告诉编译器和CPU：不管什么指令都不能和这条 `Memory Barrier` 指令重排序。

所以为了保证每个cpu的数据一致性，每一个cpu会通过嗅探总线上传播的数据来检查自己数据的有效性，当发现自己缓存的数据的内存地址被修改，就会让自己缓存该数据的缓存行失效，重新获取数据，保证了数据的可见性。

那么既然volatile可以保证可见性，它可以保证数据的原子性吗？

什么是**原子性**呢？原子性就是即**不可再分**了，不能分为多步操作。在Java中只有对基本类型变量的赋值和读取才是原子操作。

如 `i = 1`，但是像 `j = i` 或者 `i++` 都不是原子操作，因为他们都进行了多次原子操作，比如先读取i的值，再将i的值赋值给j，两个原子操作加起来就不是原子操作了。

所以假如一个 `volatile` 的 `integer` 自增 `(i++)`，其实要分成3步：

1. 读取主内存中volatile变量值到工作内存；
2. 在工作内存中增加变量的值；
3. 把工作内存的值写主内存。

假如有两个线程都要执行a变量的自增操作，当线程1执行a++;语句时，先是读入a的值为0，此时a线程的执行时间被让出。

线程2获得执行，线程2会重新从主内存中，读入a的值还是0，然后线程2执行+1操作，最后把a=1刷新到主内存中；

线程2执行完后，线程1又开始执行，但之前已经读取的a的值0，因为前面的读取原子操作已经结束了，所以它还是在0的基础上执行+1操作，也就是还是等于1，并刷新到主内存中。所以最终的结果是a变量的值为1