> 主要讲解Java的并发编程的基础知识，包括原子性、可见性、有序性，以及内存模型JMM。

Java系列说明

从这篇文章开始，我就要正式开始学习Java了，之所以说是从现在开始，是因为前两个月一直在纠结是否转技术栈（细心的同学可以发现，我之前写的文章，其实和Java并没有什么关系），现在已经想清楚了，既然确定要转Java技术栈，那就踏踏实实从头开始学吧。

目前的我，可以说是Java小白，刚转团队不久，也就接触了2个月的Java，代码没写几行，既然发现自己Java很菜，那就要列个学习计划，将这块知识好好补补。目前给自己定了一年的学习计划，希望能通过一年的学习，将Java的技能从初阶直接晋级到高阶水平，可能有同学会问“我学习Java都几年的，都还是中级水平，你花一年就可以晋级到高阶？”，我只想说，我想试试，毕竟工作这么长时间，也掌握了一定的学习方法，相信跟着自己的学习节奏走，应该不会离目标太远，今天立个Flag，希望一年后不会啪啪打脸【捂脸】~~

Java系列的内容主要包括并发编程、Spring、SpringBoost、SpringCloud、Tomcat、MyBatis、Dubbo和虚拟机，然后一些经典书籍的读书笔记等，当这些都掌握到一定深度后，我想我的Java技能应该也就差不多了。

最后想说的是，Java很多系列文章，很大一部分是内容整理，之所以要通过文章的形式再写一遍，是因为看过的内容，如果自己不整理一遍，或者不让程序跑跑，很容易遗忘，所以写文章其实不是目的，主要是重新整理和回顾学习内容的过程，一方面印象深刻，另一方面，也便于自己后续查阅。

今天废话有点多，我们就从并发编程开始吧！

并发编程基本概念

原子性

> 一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

原子性是拒绝多线程操作的，不论是多核还是单核，具有原子性的量，同一时刻只能有一个线程来对它进行操作。简而言之，在整个操作过程中不会被线程调度器中断的操作，都可认为是原子性。例如a=1是原子性操作，但是a++和a +=1就不是原子性操作。Java中的原子性操作包括：

- 基本类型的读取和赋值操作，且赋值必须是值赋给变量，变量之间的相互赋值不是原子性操作；
- 所有引用reference的赋值操作；
- java.concurrent.Atomic.* 包中所有类的一切操作。

可见性

> 指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

在多线程环境下，一个线程对共享变量的操作对其他线程是不可见的。Java提供了volatile来保证可见性，当一个变量被volatile修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会立即被更新到主内存中，其他线程读取共享变量时，会直接从主内存中读取。当然，synchronize和Lock都可以保证可见性。synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

有序性

> 即程序执行的顺序按照代码的先后顺序执行。

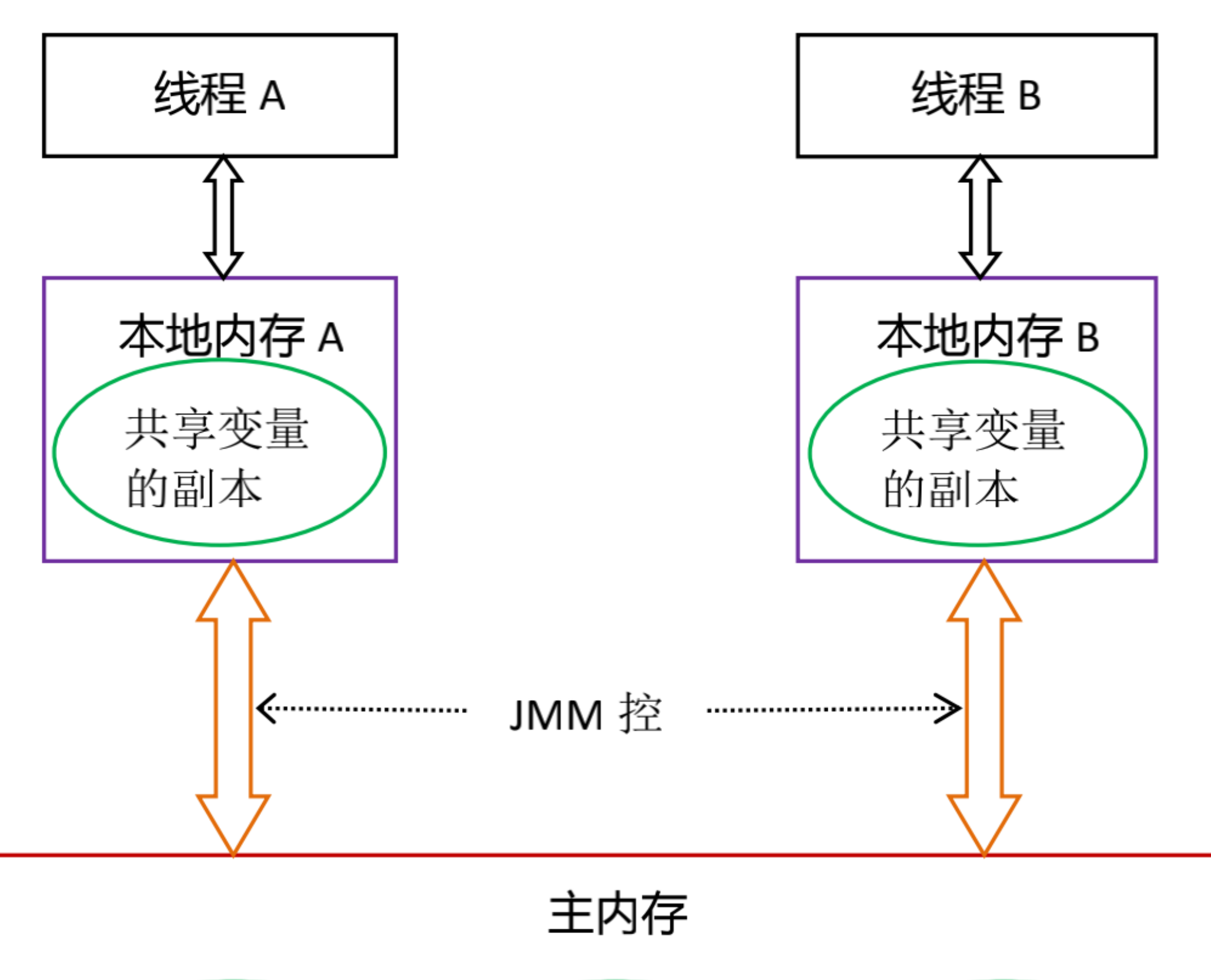
Java内存模型中的有序性可以总结为：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存主主内存同步延迟”现象。

在Java内存模型中，为了效率是允许编译器和处理器对指令进行重排序，当然重排序不会影响单线程的运行结果，但是对多线程会有影响。Java提供volatile来保证一定的有序性。最著名的例子就是单例模式里面的DCL（双重检查锁）。另外，可以通过synchronized和Lock来保证有序性，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

> 为了让大家更好理解可见性和有序性，这个就不得不了解“内存模型”、“重排序”和“内存屏障”，因为这三个概念和他们关系非常密切。

内存模型

JMM决定一个线程对共享变量的写入何时对另一个线程可见，JMM定义了线程和主内存之间的抽象关系：共享变量存储在主内存(Main Memory)中，每个线程都有一个私有的本地内存（Local Memory），本地内存保存了被该线程使用到的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。



对于普通的共享变量来讲，线程A将其修改为某个值发生在线程A的本地内存中，此时还未同步到主内存中去；而线程B已经缓存了该变量的旧值，所以就导致了共享变量值的不一致。解决这种共享变量在多线程模型中的不可见性问题，可以使用volatile、synchronized、final等，此时A、B的通信过程如下：

- 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去；
- 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

> JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证，需要注意的是，JMM是个抽象的内存模型，所以所谓的本地内存，主内存都是抽象概念，并不一定就真实的对应cpu缓存和物理内存。

总结一句话，内存模型JMM控制多线程对共享变量的可见性！！！

重排序

> 重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。

重排序需要遵守一定规则：

- 重排序操作不会对存在数据依赖关系的操作进行重排序。比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
- 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变。 比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

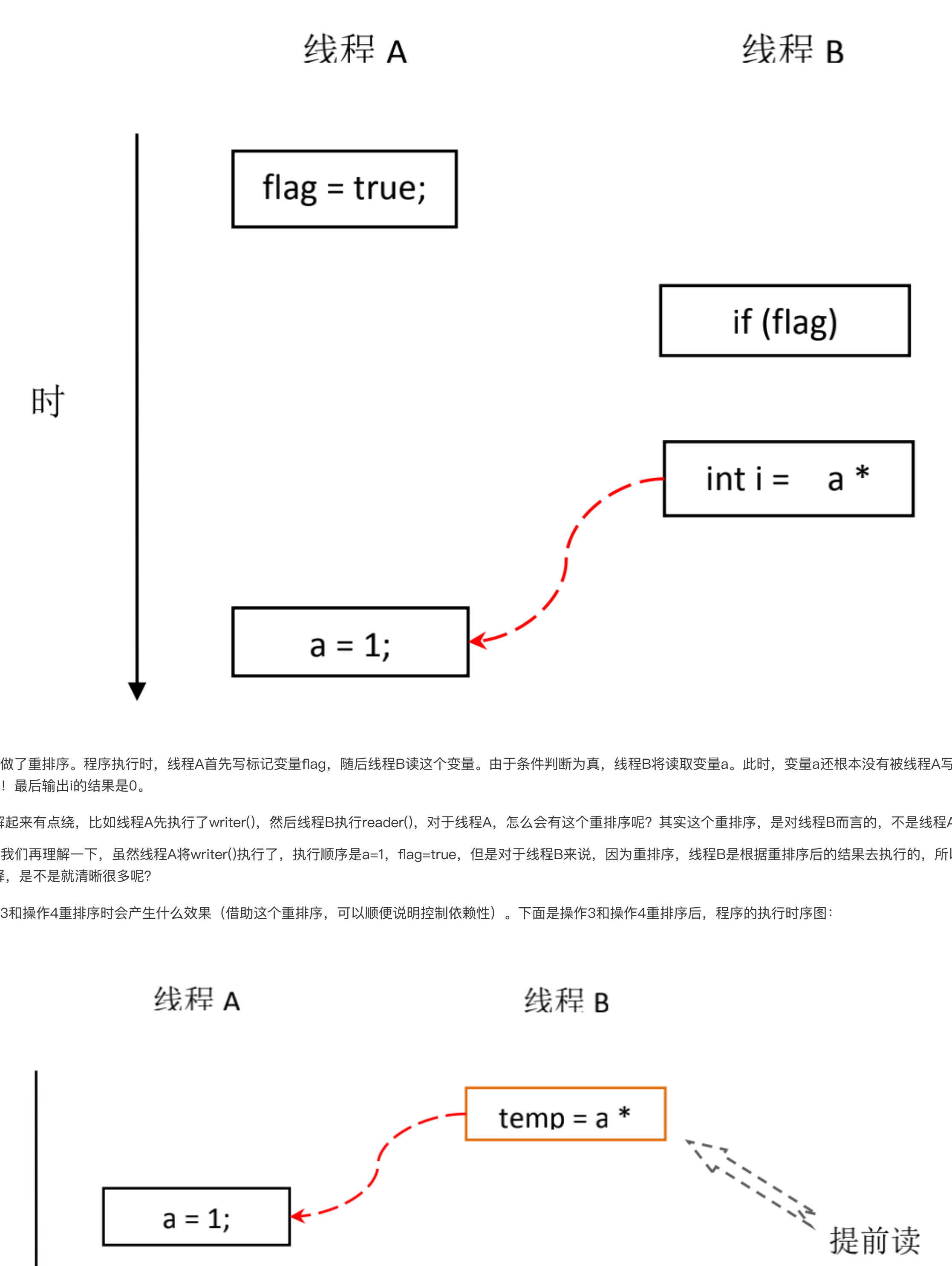
重排序在单线程下一定能保证结果的正确性，但是在多线程环境下，可能发生重排序，影响结果，请看下面的示例代码：

```
class ReorderExample {
  int a = 0;
  boolean flag = false;
  public void writer() {
      a = 1;                   //1
      flag = true;             //2
  }
  Public void reader() {
      if (flag) {                //3
          int i =  a * a;        //4
          System.out.println(i);
      }
  }
}
```

flag变量是个标记，用来标识变量a是否已被写入。这里假设有两个线程A和B，A首先执行writer()方法，随后B线程接着执行reader()方法。线程B在执行操作4时，输出是多少呢?

答案是：可能是0，也可能是1。

由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。让我们先来看看，当操作1和操作2重排序时，可能会产生什么效果？请看下面的程序执行时序图：



如上图所示，操作1和操作2做了重排序。程序执行时，线程A首先写标记变量flag，随后线程B读这个变量。由于条件判断为真，线程B将读取变量a。此时，变量a还根本没有被线程A写入，在这里多线程程序的语义被重排序破坏了！最后输出i的结果是0。

> 温馨提示：这里其实理解起来有点绕，比如线程A先执行了writer()，然后线程B执行reader()，对于线程A，怎么会有这个重排序呢？其实这个重排序，是对线程B而言的，不是线程A哈！
>
> 有了线程B这第一视角，我们再理解一下，虽然线程A将writer()执行了，执行顺序是a=1，flag=true，但是对于线程B来说，因为重排序，线程B是根据重排序后的结果去执行的，所以才会出现上述异常情况，这么给大家解释，是不是就清晰很多呢?

下面再让我们看看，当操作3和操作4重排序时会产生什么效果（借助这个重排序，可以顺便说明控制依赖性）。下面是操作3和操作4重排序后，程序的执行时序图：

在程序中，操作3和操作4存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。以处理器的猜测执行为例，执行线程B的处理器可以提前读取并计算a*a，此时结果为0，然后把计算结果临时保存到一个名为重排序缓冲（reorder buffer ROB）的硬件缓存中。当接下来操作3的条件判断为真时，就把该计算结果写入变量i中。

从图中我们可以看出，猜测执行实质上对操作3和4做了重排序。重排序在这里破坏了多线程程序的语义！因为temp的值为0，所以最后输出i的结果是0。

那如何避免重排序对多线程的影响呢，答案是“内存屏障”！

内存屏障

为了保证内存可见性，可以通过volatile、final等修饰变量，java编译器在生成指令序列的适当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。内存屏障主要有3个功能：

- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 它会强制将对缓存的修改操作立即写入主存；
- 如果是写操作，它会导致其他CPU中对应的缓存行无效。

假如我对上述示例的falg变量通过volatile修饰：

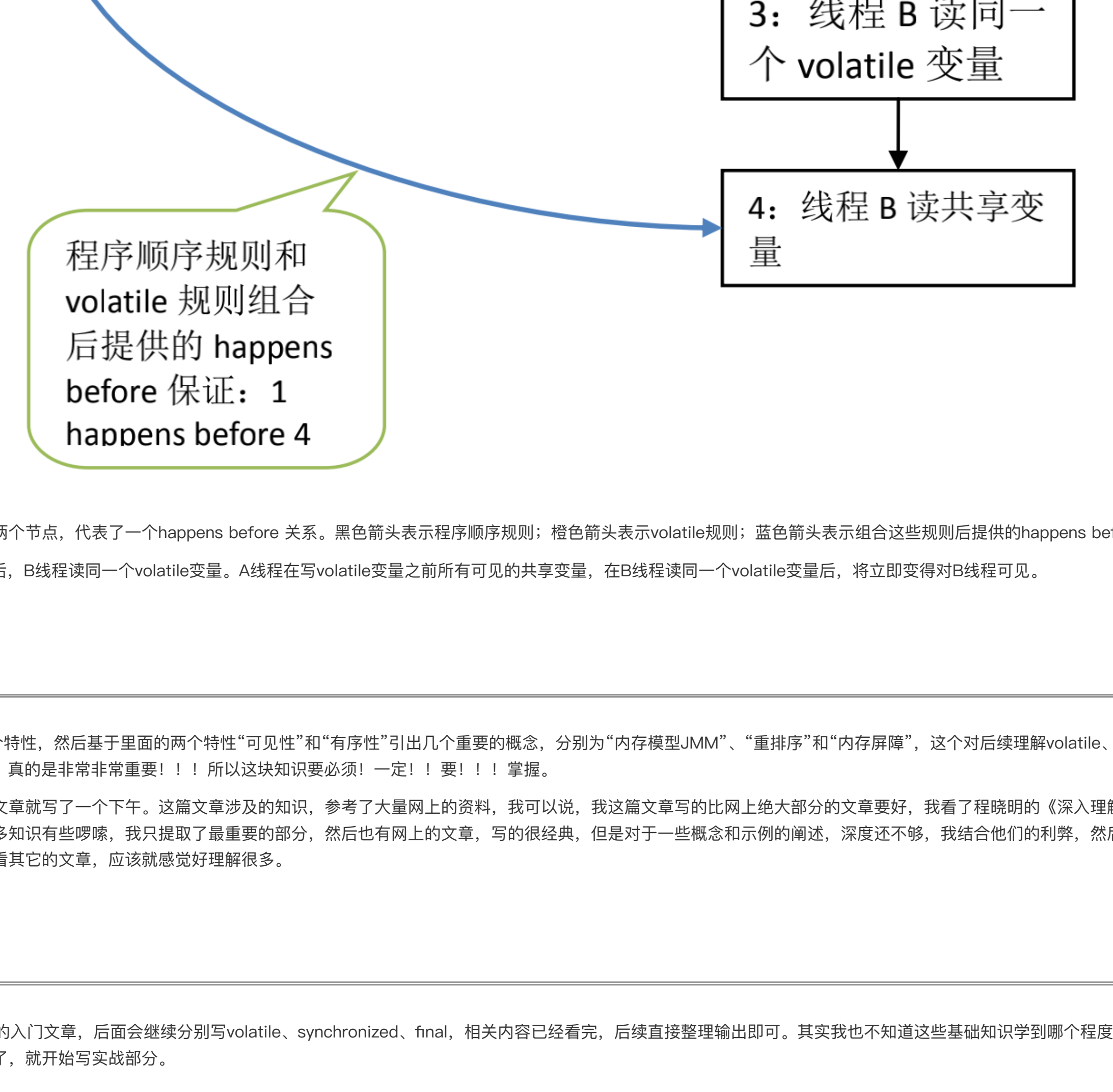
```
class ReorderExample {
  int a = 0;
  boolean volatile flag = false;
  public void writer() {
      a = 1;                   //1
      flag = true;             //2
  }
  Public void reader() {
      if (flag) {                //3
          int i =  a * a;        //4
          System.out.println(i);
      }
  }
}
```

这个时候，volatile禁止指令重排序也有一些规则，因为篇幅原因，改规则将会在下一章讲解，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。

> happens before规则，其实就是重排序规则建立的代码前后依赖关系。
>
> 温馨提示：这里大家可能会有疑问，1、3的规则我理解，但是对于2，为什么“2 happens before 3”，还记得前面讲的“内存模型”么？因为你对变量flag指定了volatile，所以当线程A执行完后，变量flag=true会直接刷到内存中，然后B马上可见，所以说2一定是在3前面，不可能因为重排序，导致3在2前面执行。（然后还要提示一下，这里执行时有个前提条件，就是线程A执行完，才能执行线程B里面的逻辑，因为线程A不执行完，flag一直是false，线程B根本就进不到主流程，所以你也可以直接理解为线程A执行完后，再执行线程B，才有这么个先后关系。）

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。

总结

今天讲解了Java并发编程的3个特性，然后基于里面的两个特性“可见性”和“有序性”引出几个重要的概念，分别为“内存模型JMM”、“重排序”和“内存屏障”，这个对后续理解volatile、synchronized、final，以及避免使用的各种坑，真的是非常非常重要！！！所以这块知识要必须！一定！！要！！！掌握。

不算之前看的内容，光写这篇文章就写了一个下午。这篇文章涉及的知识，参考了大量网上的资料，我可以说，我这篇文章写的比网上绝大部分的文章要好，我看了程晓明的《深入理解Java内存模型》，里面的内容虽然很好，但是很多知识有些啰嗦，我只提取了最重要的部分，然后也有网上的文章，写的很经典，但是对于一些概念和示例的阐述，深度还不够，我结合他们的利弊，然后整理了这篇文章，详细大家看完这篇文章后，再看其它的文章，应该就感觉好理解很多。

后记

这篇文章是我对Java并发编程的入门文章，后面会继续分别写volatile、synchronized、final，相关内容已经看完，后续直接整理输出即可。其实我也不知道这些基础知识学到哪个程度才算OK，那就边写边学，等基础知识写的差不多了，就开始写实战部分。

参考文章：
《深入理解Java内存模型》
《Java并发编程实战》

学习交流

可以扫下面二维码，关注「楼仔」公众号。

长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏👏👏

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。