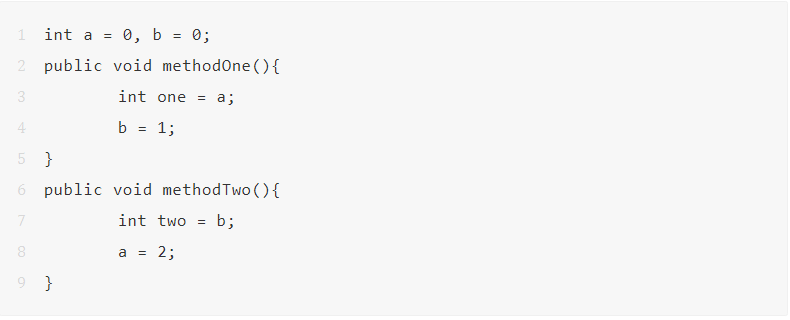
https://mp.weixin.qq.com/s/LSCqeYRKD0XqIsRmGYiACQ

# 为什么会有重排序？和 happens-before 有啥关系

## 举个例子

在讲重排序之前，先来看一个例子：



应该不难看出，在上面的例子中，我定义了两个共享变量 a 和 b ，以及两个方法。其中第一个方法是将局部变量 one 赋值为 a ，然后将 b 的值置为 1 。第二个方法则是将局部变量 two 赋值为 b ，然后将 a 的值置为 2 。

那么我在这里有个问题，  ( one , two ) 的值会是什么？

你可能会不假思索的告诉我，不是 ( 0 , 1 ) 就是 ( 2 , 0 ) ，这需要看我的 main 方法先执行哪个 method 方法。

不错，如果这个程序跑在了单线程上面，这样回答一点儿毛病都没有。

但是，如果是在多线程环境下呢？

假设，现在 methodOne 和 methodTwo 分别在两个不同的线程上执行，此时 Java 虚拟机在执行了任意一个方法的第一条赋值语句之后就切换线程，这个时候的 ( one , two ) 的值可能是 ( 0 , 0 )

看到这儿，有没有疑惑？为啥呢，怎么我写的程序好好的，到 Java 虚拟机这里了，它就给我变了呢？

就是因为在执行的过程中，发生了重排序。它可能是即时编译器的重排序，可能是处理器的乱序执行，或者是内存系统的重排序。

总之，在程序执行过程中，发生了重排序，然后得到的结果可能是 ( 0 ， 0 ) 这种情况。

## 为什么会重排序

看完上面，你可能会有疑问，为什么会有重排序呢？

我的程序按照我自己的逻辑写下来好好的没啥问题， Java 虚拟机为什么动我的程序逻辑？

你想想， CPU ，内存这些都是非常宝贵的资源， Java 虚拟机如果在重排序之后没啥效果，肯定也不会做这种费力不讨好的事情。

那么，重排序带来了什么好处呢？

## 重排序使得程序的性能得以提高

为了方便理解，我拿生活中的场景来举例子。

大早上起来，你会穿衣服，洗漱，做饭，吃饭对吧。那么在你起床之后，你是怎么做的呢？你是不是会在洗漱的时候，先把饭做上(比如让蒸蛋机帮你蒸个鸡蛋),然后呢等你洗漱完毕之后，就可以直接吃早饭了。

你为什么要这样做呢？还不是为了省时间，可以多睡那么一分钟，对不对。

同样的道理， Java 虚拟机之所以要进行重排序就是为了提高程序的性能。你写的程序，简简单单一行代码，到底层可能需要使用不同的硬件，比如一个指令需要同时使用 CPU 和打印机设备，但是此时 CPU 的任务完成了，打印机的任务还没完成，这个时候怎么办呢？不让 CPU 执行接下来的指令吗？CPU 的时间那么宝贵，你不让它工作，确定不是在浪费它的生命？

所以为了提高利用率以及程序的性能， Java 虚拟机会在你这个指令还没完全执行完毕的时候，就去执行另外一个指令。这就是流水线技术

流水线最怕的是啥？是我执行着命令，执行着命令，突然中断了，恢复中断的成本是很大的，所以就要想尽办法，绞尽脑汁不要让中断的情况发生。

即时编译器的重排序，处理器的乱序执行，以及内存系统的重排序的存在，都是为了减少中断。

到这里，你是不是对于 Java 虚拟机进行重排序这一点有了了解？

## 重排序带来的问题

回到文章刚开始举的那个例子，重排序提高了 CPU 的利用率没错，提高了程序性能没错，但是我的程序得到的结果可能是错误的啊，这是不是就有点儿得不偿失了？

因为**重排序可以保证串行语义一致，但是没有义务保证多线程间的语义也一致**

凡是问题，都有办法解决，要是没有，那就再想想。

它是怎么解决的呢？这就需要来说说，顺序一致性内存模型和 JMM ( Java Memory Model ， Java 内存模型)

## 顺序一致性内存模型与 JMM

要说数据一致性的话，就要说一说，数据竞争。

啥是数据竞争呢？在 Java 内存模型规范中给出了定义：

* 在一个线程中写一个变量
* 在另外一个线程中读同一个变量
* 写和读没有通过同步来排序

当代码中包含数据竞争时，程序的执行结果往往会超出你的想象，比如咱们刚开始说的那个例子，得到的结果可能是 ( 0 ， 0 ) 。但是如果一个多线程程序能够正确同步的话，那上面的结果就不会出现了。

Java 内存模型对于正确同步多线程程序的内存一致性做了下面的保证：

**如果程序是正确同步的，程序的执行也会具有顺序一致性**即，程序的执行结果与该程序在顺序一致性模型中执行的结果相同

这里面的同步包括了使用 volatile ， final ， synchronized 等关键字来实现多线程下的同步。那也就是说，如果没有正确使用这些同步， JMM 就不会有内存可见性的保证，这就会导致写的程序出错。

顺序一致性内存模型是一个**理想状态下的理论参考模型**，它为程序员提供了特别强的内存可见性保证，顺序一致性模型有两大特性：

* 一个线程中的所有操作必须按照程序的顺序来执行(也就是按照写的代码的顺序来执行)
* 不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序。也就是说，在顺序一致性模型中，每个操作必须是原子性的，而且立刻对所有线程都是可见的。

上面说了，顺序一致性内存模型是一个理想状态下的理论参考模型，因为顺序一致性内存模型要求操作对所有线程都是可见，只是这一点就会让 Java 虚拟机的性能降低。JMM 就是在顺序一致性内存模型的基础上，做了一些优化：

* 针对同步的多线程程序来说，也就是临界区内的代码， JMM 允许发生重排序(但是不允许临界区内的代码"逃逸"到临界区之外，因为如果允许的话，就会破坏锁的内存语义)
* 针对未同步的多线程程序来说， JMM 只提供最小安全性：线程读取到的值，要么是之前某个线程写入的值，要么是默认值，不会无中生有。

应该能够感觉到，相比于顺序一致性内存模型来说， JMM 给了编译器和处理器一些空间，允许它们发生重排序。

这时候就有冲突点了：程序员这边需要 JMM 提供一个强的内存模型来编写代码，也就是我代码写的顺序是什么样，那程序执行的时候就要是什么样;但是编译器和处理器则需要 JMM 对它们的约束越少越好，这样它们就可以尽可能多的去做优化，来提高性能。

作为 JMM 这个中介者来说，既要满足程序员的需求，又要满足编译器和处理器的需求，那就需要在这两者之间找一个平衡点，让程序员写的代码能够产生他期望的结果，同时呢，也让编译器和处理器能够做一些优化

JMM 提出的解决方案就是：对于程序员，提供 happens-before 规则，这样就满足了程序员的需求 ---> 简单易懂，而且提供了足够强的内存可见性保证；对于编译器和处理器来说，只要不改变程序的执行结果(前提是正确同步了多线程程序)，想怎么优化就怎么优化。

## happens-before

终于讲到了 happens-before 。

先来看 happens-before 关系的定义：

* 如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果就会对第二个操作可见
* 两个操作之间如果存在 happens-before 关系，并不意味着 Java 平台的具体实现就必须按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与按照 happens-before 关系来执行的结果一直，那么 JMM 也允许这样的重排序

看到这儿，你是不是觉得，这个怎么和 as-if-serial 语义一样呢。没错， happens-before 关系本质上和 as-if-serial 语义是一回事。

as-if-serial 语义保证的是单线程内重排序之后的执行结果和程序代码本身应该出现的结果是一致的， happens-before 关系保证的是正确同步的多线程程序的执行结果不会被重排序改变。

一句话来总结就是：如果操作 A happens-before 操作 B ，那么操作 A 在内存上所做的操作对操作 B 都是可见的，不管它们在不在一个线程。

在 Java 中，对于 happens-before 关系，有以下规定：

* 程序顺序规则：一个线程中的每一个操作， happens-before 于该线程中的任意后续操作
* 监视器锁规则：对一个锁的解锁， happens-before 于随后对这个锁的加锁
* volatile 变量规则：对一个 volatile 域的写， happens-before 与任意后续对这个 volatile 域的读
* 传递性：如果 A happens-before B ， 且 B happens-before C ，那么 A happens-before C
* start 规则：如果线程 A 执行操作 ThreadB。start() 启动线程 B ，那么 A 线程的 ThreadB。start() 操作 happens-before 于线程 B 中的任意操作
* join 规则：如果线程 A 执行操作 ThreadB。join() 并成功返回，那么线程 B 中的任意操作 happens-before 于线程 A 从 ThreadB。join() 操作成功返回。

写到这里，我感觉终于是写完这篇文章了，从为什么要重排序讲到 happens-before 。