**1、首先为何要指令重排序（instruction reordering）？**

**编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。**

也就是说，对于下面两条语句：

int a = 10;

int b = 20;

在计算机执行上面两句话的时候，**有可能**第二条语句会先于第一条语句执行。所以，千万不要随意假设指令执行的顺序。

**2、是不是所有的语句的执行顺序都可以重排呢？**

答案是否定的。为了讲清楚这个问题，先讲解另一个概念：数据依赖性

**2.1、什么是数据依赖性**

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖。数据依赖分下列三种类型：

|  |  |  |
| --- | --- | --- |
| 名称 | 代码示例 | 说明 |
| 写后读 | a = 1;b = a; | 写一个变量之后，再读这个位置。 |
| 写后写 | a = 1;a = 2; | 写一个变量之后，再写这个变量。 |
| 读后写 | a = b;b = 1; | 读一个变量之后，再写这个变量。 |

上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果将会被改变。所以，编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。也就是说：**在单线程环境下，指令执行的最终效果应当与其在顺序执行下的效果一致，否则这种优化便会失去意义。这句话有个专业术语叫做as-if-serial semantics (as-if-serial语义)**

**3、重排序对多线程的影响（**控制依赖关系对单线程无影响**）**

现在让我们来看看，重排序是否会改变多线程程序的执行结果。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

      }

    }  }

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，能否看到线程A在操作1对共享变量a的写入？

答案是：不一定能看到。

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：

上图的执行顺序是：2 -> 3 -> 4 -> 1 (这是完全存在并且合理的一种顺序，如果你不能理解，请先了解CPU是如何对多个线程进行时间分配的)

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！

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。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，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！

在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是as-if-serial语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。

#重排后，被插队，插队的又可以重排。

#主要是，多线程不管这个准则。（没有不同线程的数据依赖的一定要在前面，）