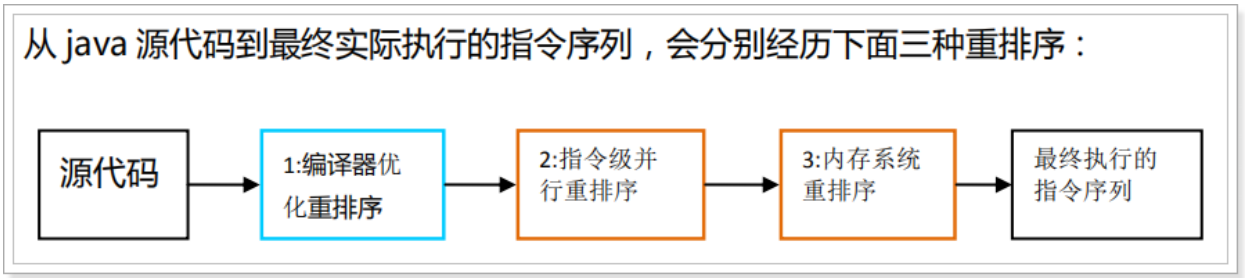
重排序是对内存访问操作（读写）的一种优化，在不影响单线程正确执行的情况下，提升程序的性能。

重排序类型	表现	来源
指令重排序	字节码顺序和源代码顺序不一致	编译器
	执行顺序与字节码顺序不一致	JIT编译器、处理器
存储子系统重排序	处理器感知其他处理器对数据的操作乱序	处理器（高速缓存、写缓

java平台包括两种编译器：静态编译器（javac）和动态编译器（JIT）。javac的作用是将java源码编译成.class字节码文件。JIT的作用是将字节码动态编译为java虚拟机宿主机的本地代码（机器码），是在程序运行过程中执行的。



重排序需要遵循一定的规则（As-If-serial语义），而不是对指令、内存操作的结果进行随意排序。从而给单线程程序带来的一种现象是：指令是按照源代码顺序执行的。

As-If-serial语义：

不管怎么重排序，单线程程序的执行结果不能被改变。**保证了重排序不会影响单线程程序执行的正确性，但是可能会导致多线程程序出现非预期的结果。**

As-If-serial规定了存在数据依赖关系的语句不会被重排序，只有不存在数据依赖的语句才会被重排序。As-If-serial使得程序员在单线程编程的情况下，无需担心重排序会干扰程序正确运行，也无需担心内存可见性问题。

数据依赖关系类型

类型	代码示例	说明
写后读	x=1; y=x+1;	y依赖x的值。若重排序，导致y的
读后写	y=x; x=1;	读一个变量x后，再写x的值
写后写	x=1; x=2;	写x之后，再写x的值

存在控制依赖关系的语句可以被重排序(注意此处是控制依赖，不是数据依赖)

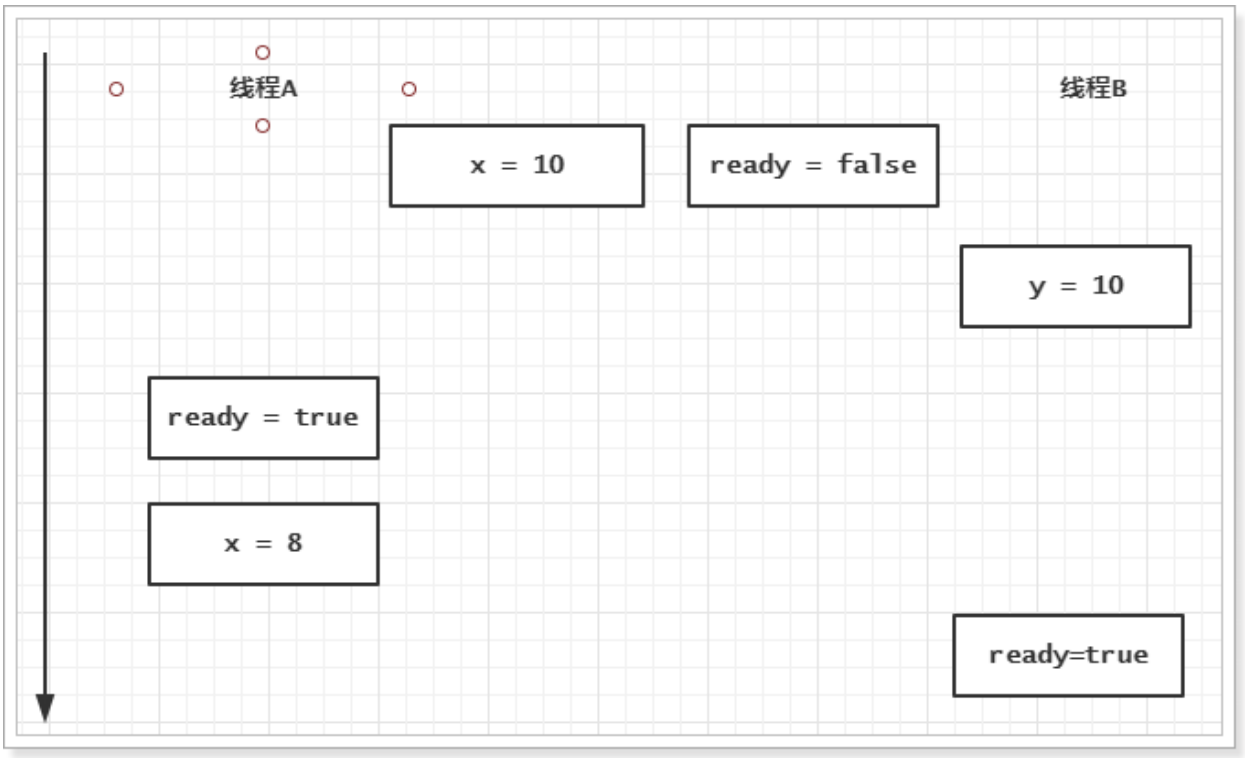
```
1  public class Reorder {
2   private int x = 10;
3   private boolean ready = false;
4
5   public int read() {
6   int y = 0;
7   if (ready) { //第7行
8   y = x; //第8行
9   }
10   if (y == 10) {
11   System.out.println(y); //y==10,说明执行了y=x,ready为true
12   }
13   return y;
14   }
15
16   public void write() {
17   x = 8; //第17行
18   ready = true; //第18行
19   }
20
21   public static void main(String[] args) {
22   ExecutorService executor = Executors.newCachedThreadPool();
23   while (true) {
24   //每次都创建了新的reorder对象，重新开始，避免受上次执行的影响。
25   Reorder reorder = new Reorder();
26   executor.submit(() -> reorder.write());
```

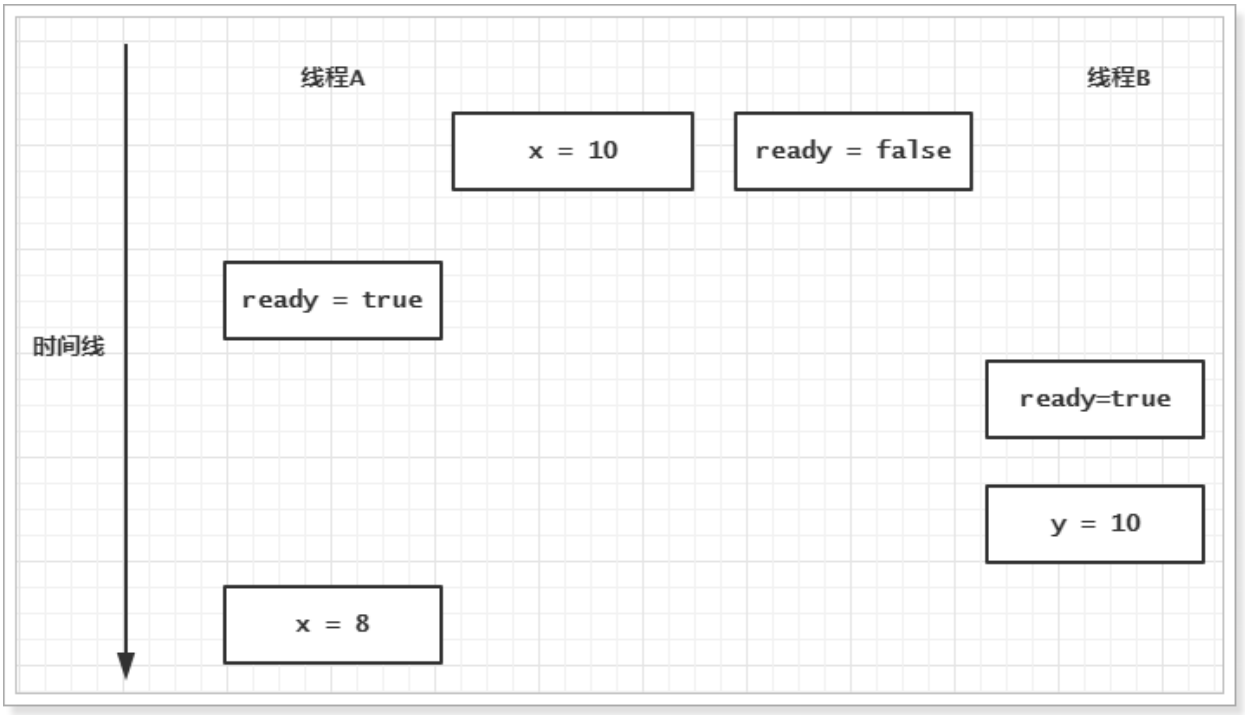
```
27    executor.submit(() -> reorder.read());
28    }
29    }
30  }
```

上面代码中的第8行的执行依赖ready的值，但是第8行代码可能会被重排序到第7行的判断前执行，这是因为处理器采取了一种猜测执行（Speculation）的技术。处理器可能先执行了 y = x; 并将结果存放于ROB（重排序缓冲器）中。接着再去读取变量ready的值。如果ready的值为true，那么会将ROB中存放的y写回主内存。如果ready的值为false，将会丢弃ROB中x的值，达到y=x;没有执行的效果。从单线程角度来看，并不会影响程序的正确性。但是它可能会导致多线程出现非预期的结果。

对于上述代码结果出现y=10进行分析：

1. 重排序导致读线程的第8行代码先于第7行执行，此时x==10。然后写线程将ready设置为true，读线程判断ready的值为true，将10赋值给y。在单线程情况下，7、8行重排序不会影响程序正确性（参考上面的分析），但是此处是多线程，线程的情况下，if中的判断条件可能会被其他线程改变，猜测执行的结果并不一定准确了。此处读线程的处理器可以提前读取x的值，然后把计算结果临时保存到一个名为重排序缓冲 （ROB）的硬件缓存中。当接下来判断ready为真时，就把该结果赋值给y。
2. 写线程在执行write()方法时，由于第17行和第18行没有数据依赖，可能会重排序。也就是写线程先设置了ready为true，还未设置x=8，此时x==10，然后读线程发现ready是true，将10赋值给y。





进一步测试：

- 使用synchronized修饰 write()

因为这里只有一个线程执行write()方法，并且读线程没有使用synchronized来修饰read()方法。那样使用synchronized并没有什么用，上述结果y还可能是

10，同理，只使用synchronized修饰read()方法也一样，使用synchronized保证线程安全需要对共享变量的所有操作方法都需要使用synchronized修饰。synchronized保证共享变量可见性的原因是读取共享变量的时候会刷新缓存（工作内存），从而读取到共享变量最新值。如果不使用synchronized来读取共享变量，并不能保证一定能读取到最新的值。当使用synchronized修饰write()方法时，写线程获取锁之后，仍然是有可能对第17、18行代码进行重排序（不会重排序到临界区外）。锁的获取能够保证临界区内共享变量的值是最新值，锁的释放会将共享变量写到主存。不使用锁当然也会将共享变量的值写回主内存，只是这个时间不能确定，由操作系统调用。总结：对共享变量的读和写都需要使用锁才能保证共享变量的可见性。

- 使用volatile修饰 ready

使用volatile修饰ready之后，会禁止第17行和18行的重排序。也就不会出现上面原因分析的第2点。volatile关键字会禁止volatile写操作与该操作之前的任何读、写操作重排序，保证了 volatile写操作之前的读、写操作会先提交（写到主内存）。其他线程在看到volatile变量的更新时，volatile变量之前的操作也是可见的。也就是在第7行和第8行未重排序时，读到的ready为true时，x一定等于8。如果第7行和第8行能够发生重排序，则x还是有可能读到的是10。但是volatile关键字禁止了volatile读操作与后面的任何读、写操作重排序，也就保证了读到的ready为true时，x一定等于8。

happens-before原则

happens-before 用来描述不同操作之间的内存可见性。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要存在 happens-before 关系。

与程序员密切相关的规则：

- 程序顺序规则：一个线程中的每个操作，happens-before 于该线程中的任意后续操作。
- 传递性：如果 A happens-before B，且 B happens-before C，那么 A happens-before C。

- volatile 变量规则：对一个 volatile 域的写，happens-before 于任意后续对这个 volatile 域的读。
- 监视器锁规则：对一个监视器的解锁，happens-before 于随后对这个监视器的加锁。

如果 A happens-before B，JMM并不要求A一定要在B之前执行。JMM仅仅要求前一个操作A的执行结果对后一个操作B可见，且操作A的顺序排在B操作之前。

```
//重排序前
int a = 1;//操作A
int b = 2;//操作B
int c = a + b;//操作C

//重排序后
int b = 2;//操作B
int a = 1;//操作A
int c = a + b;//操作C
```

上述代码满足 A happens-before B，操作A的执行结果不需要对操作B可见，而且重排序操作A和操作B后的的执行结果，与未重排序时执行的结果一致。在这种情况下，JMM允许这种重排序。