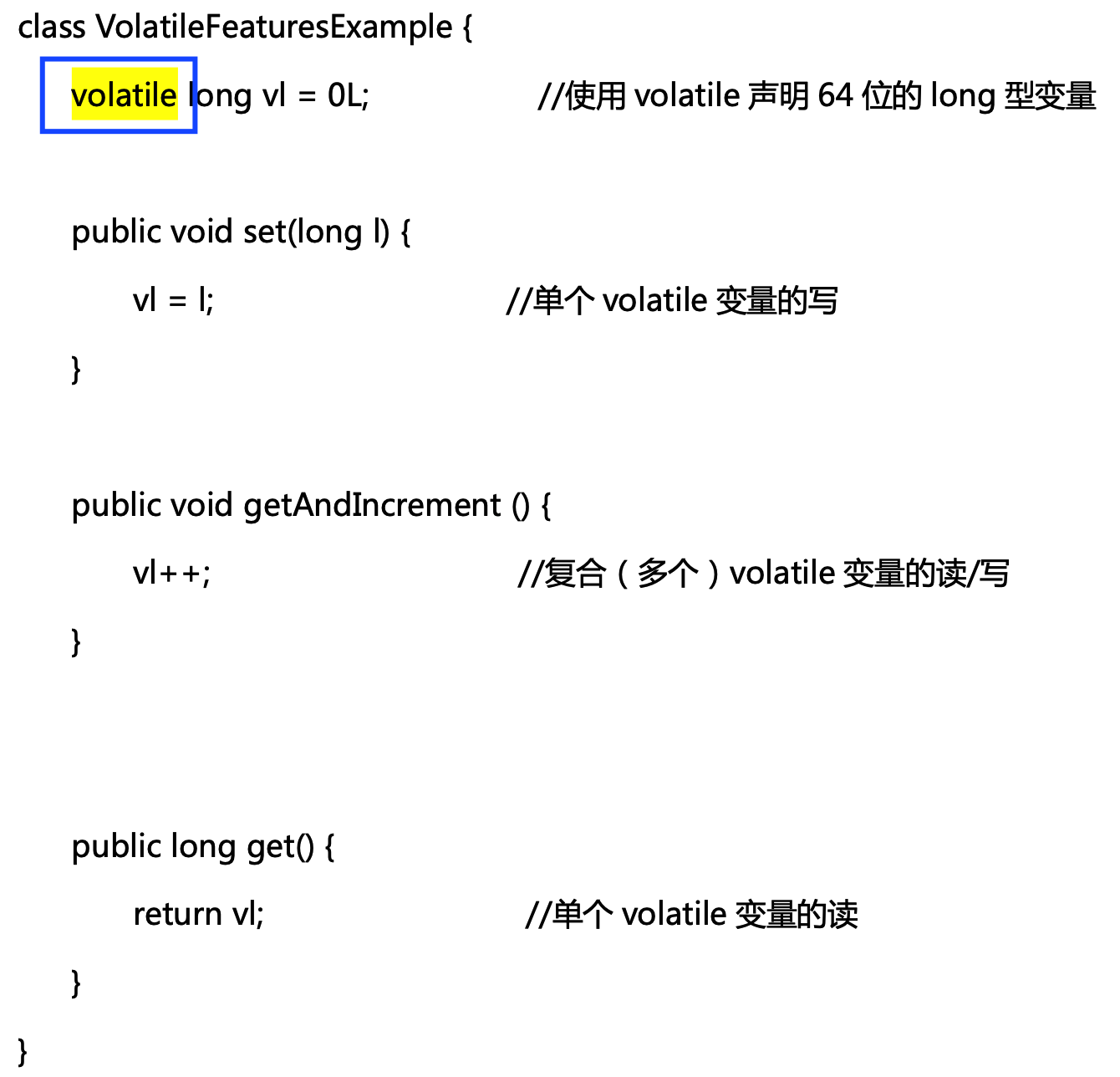
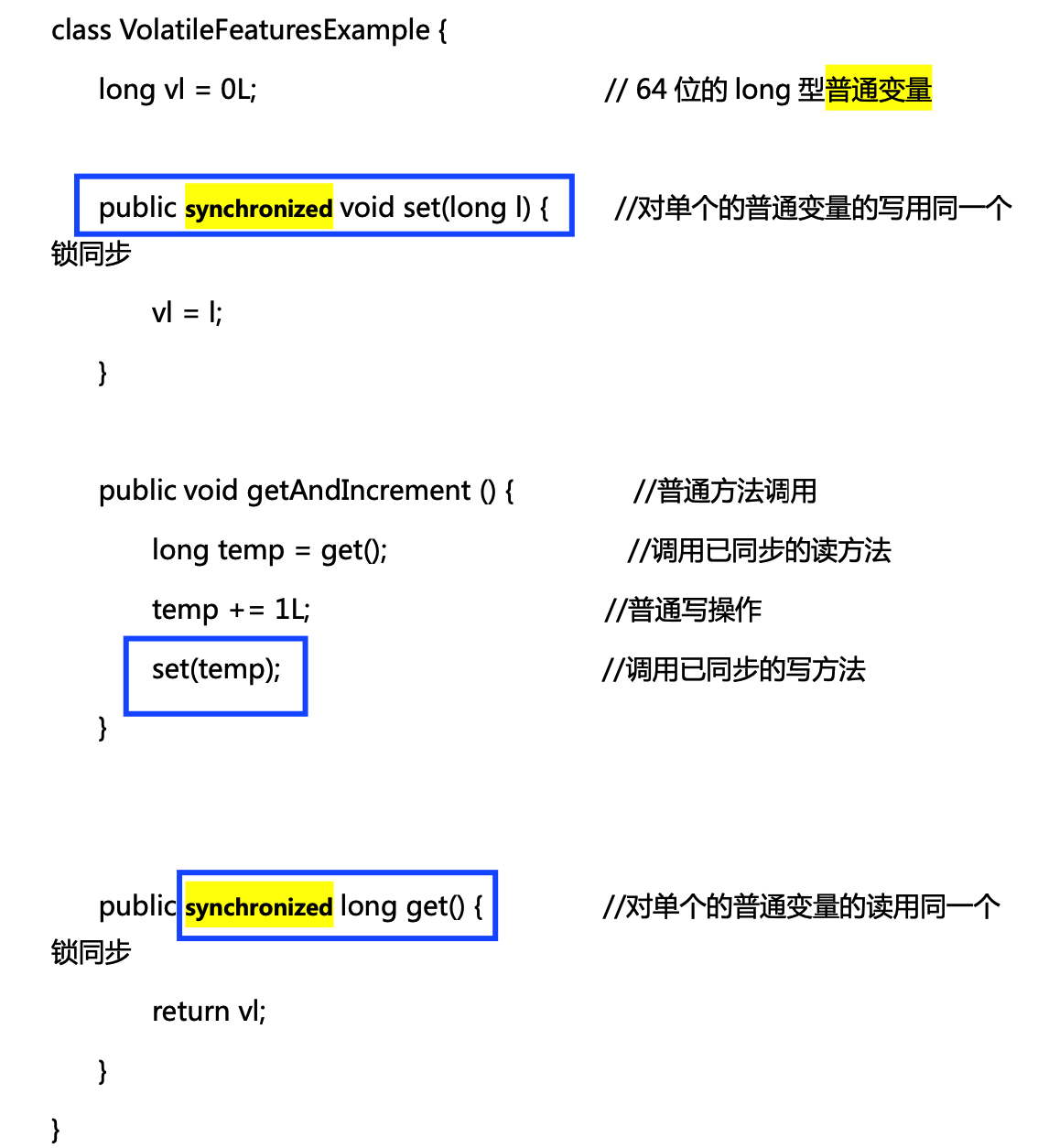
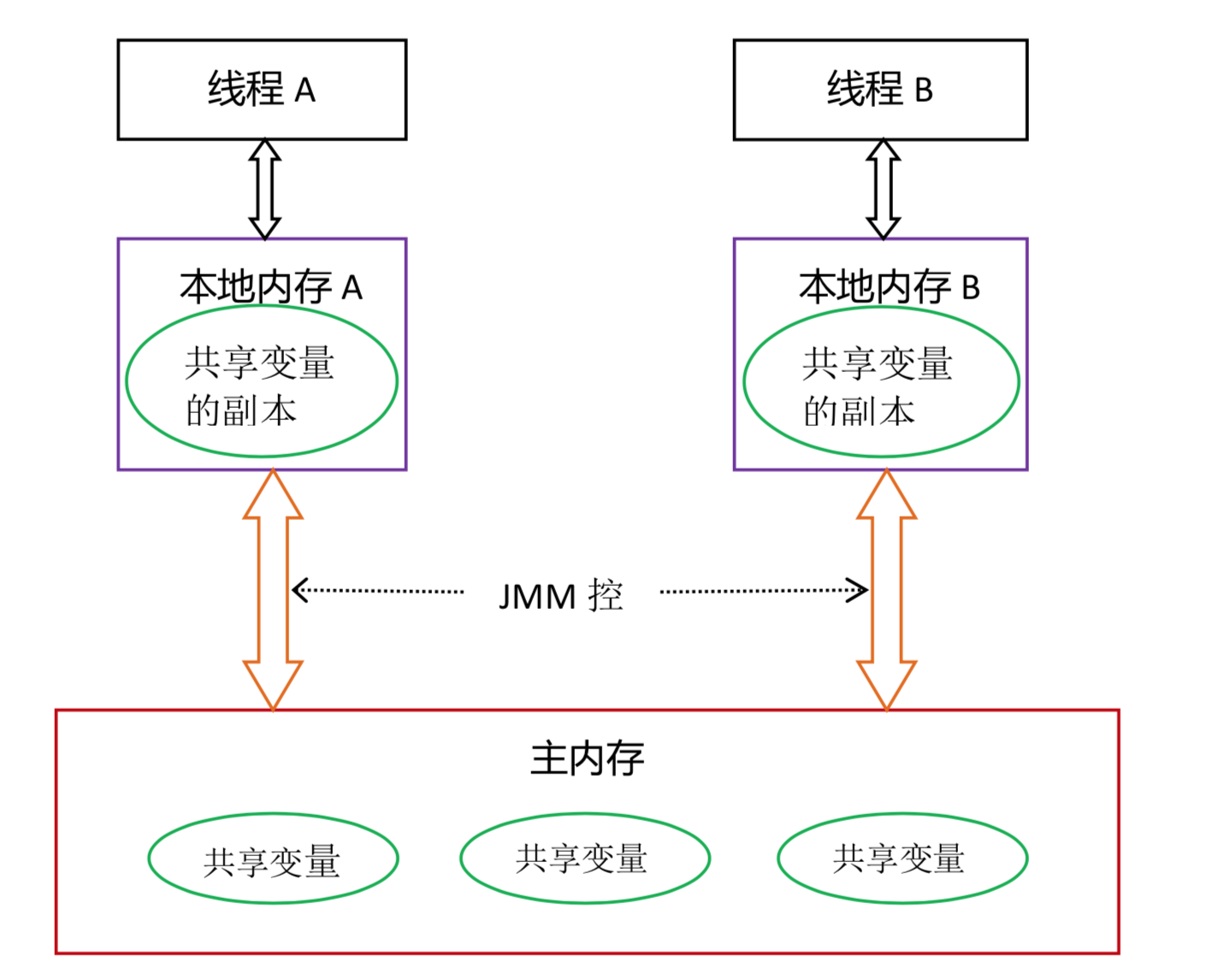
1. volatile: JVM提供的轻量级的同步机制。
   1. 保证可见性
   2. 不保证原子性，只保证单个volatile变量的读/写具有原子性，但类似volatile++这种复合操作不具有原子性
   3. 禁止指令重排序
2. 可以把对volatile变量的单个读/写，看成是用同一个锁对这些单个读/写操作做了同步



以上程序在语义上与下面程序相同



1. 内存可见性



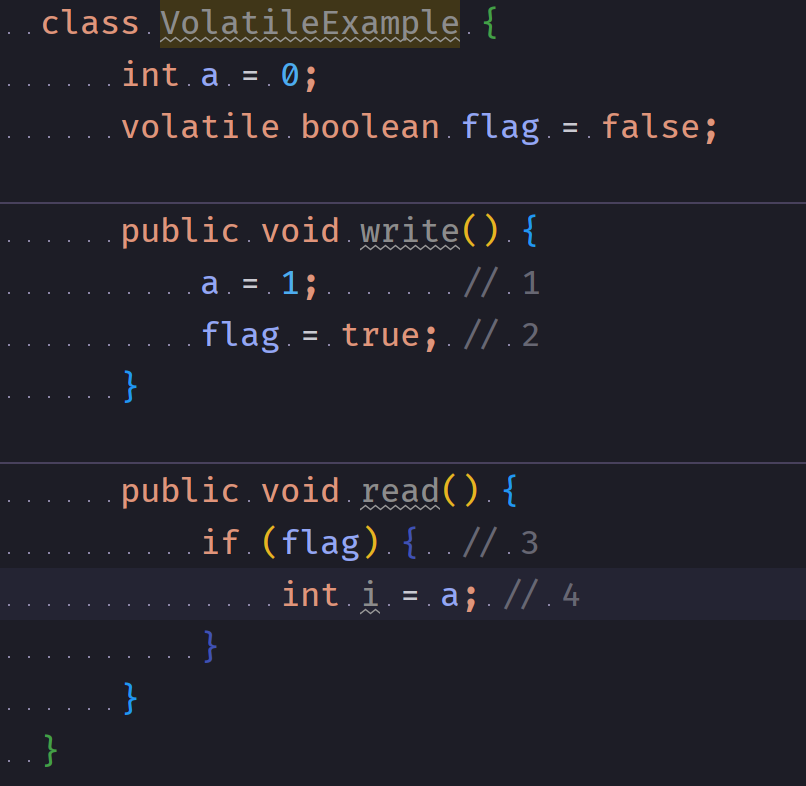
在当前的Java内存模型下，线程可以把变量保存在本地内存(比如机器的寄存器)中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致。要解决这个问题，只需要把该变量声明为volatile（不稳定的）即可，这就指示JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。一般说来，多任务环境下各任务间共享的标志都应该加volatile修饰。

volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值

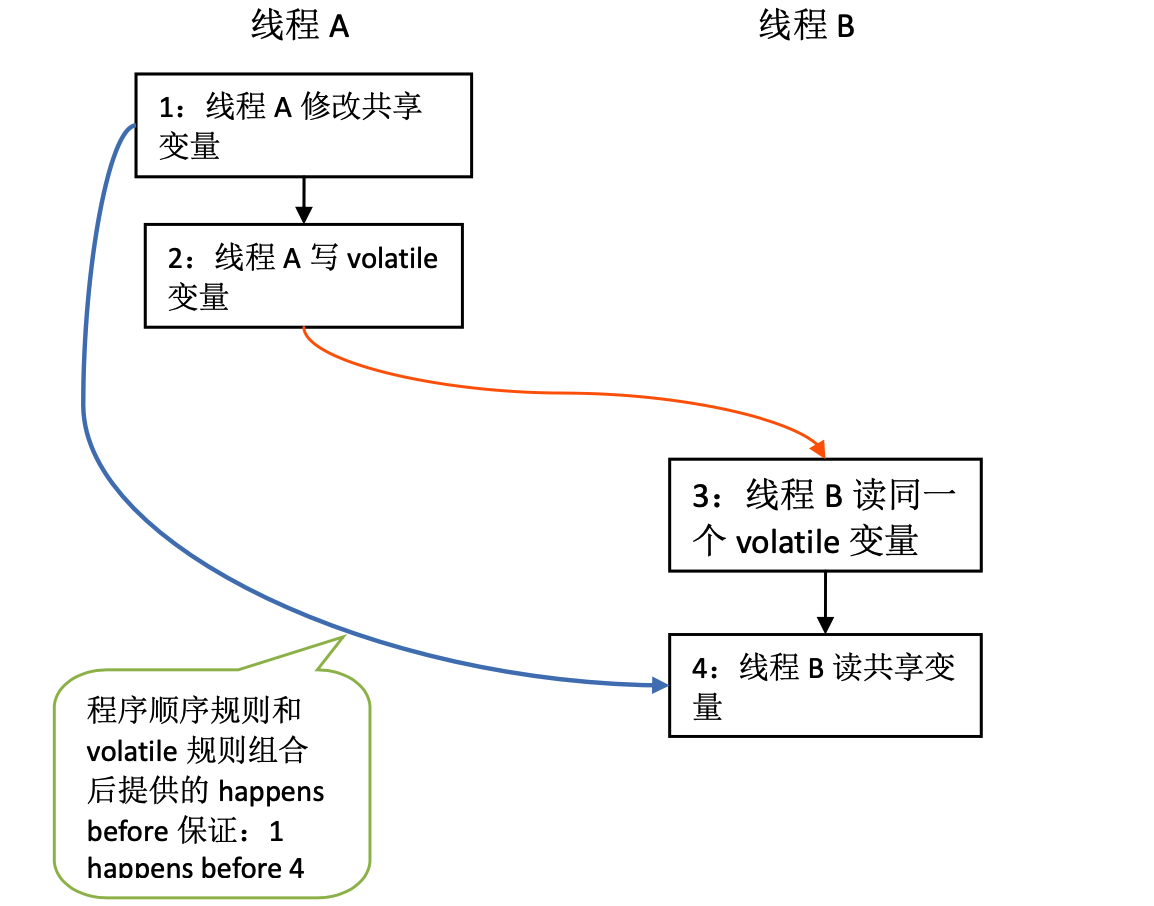
1. volatile写-读的内存语义，volatile的 写-读 与锁的 释放-获取 有相同的内存效果。volatile写和锁的释放有相同的内存语义，volatile读和锁的获取有相同的内存语义

**volatile写的内存语义**：当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。

**volatile读的内存语义**：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量

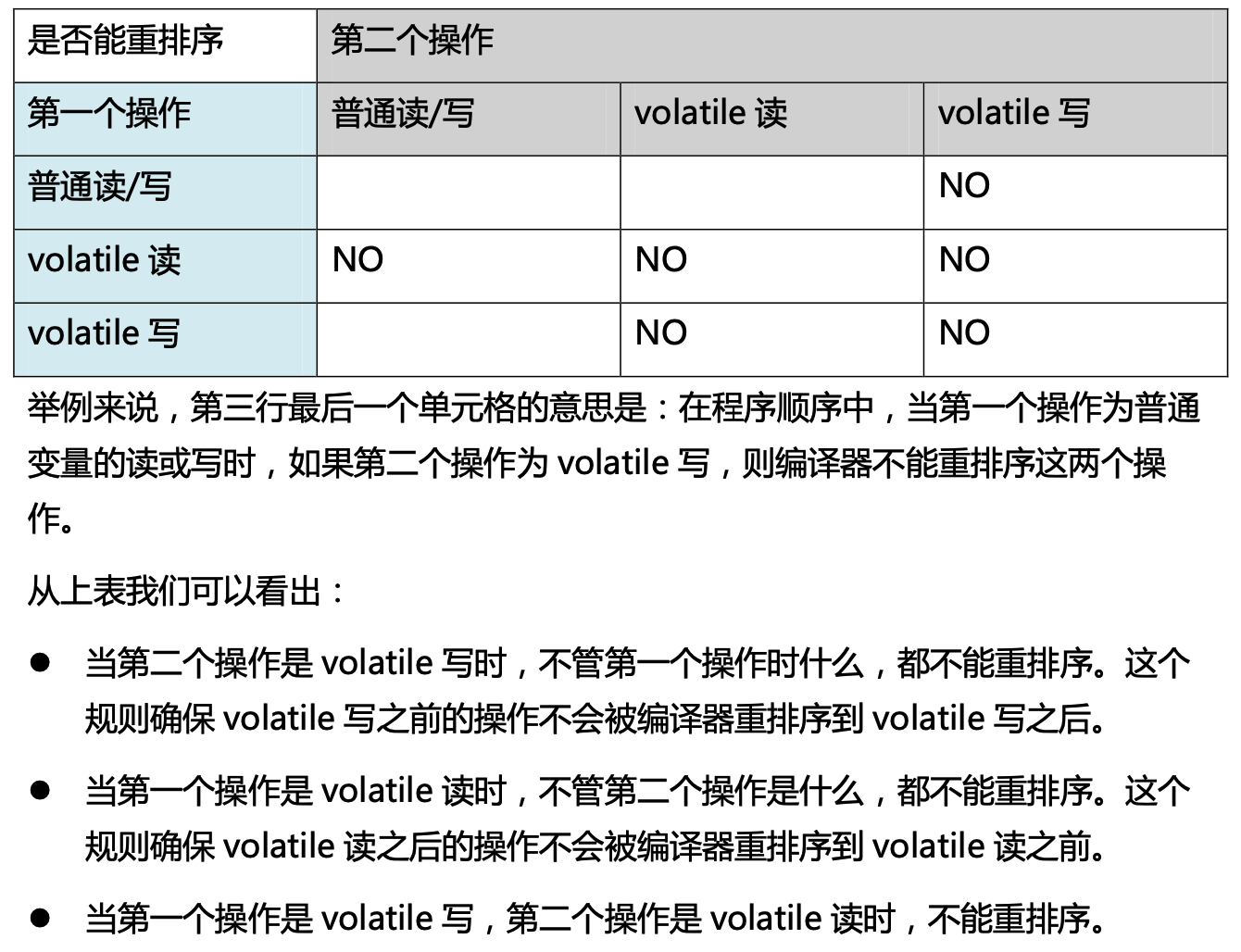


1. 根据程序顺序规则，1 happens before 2，3 happens before 4
2. 根据volatile规则，2 happens before 3
3. 根据happens before传递性规则，1 happens before 4

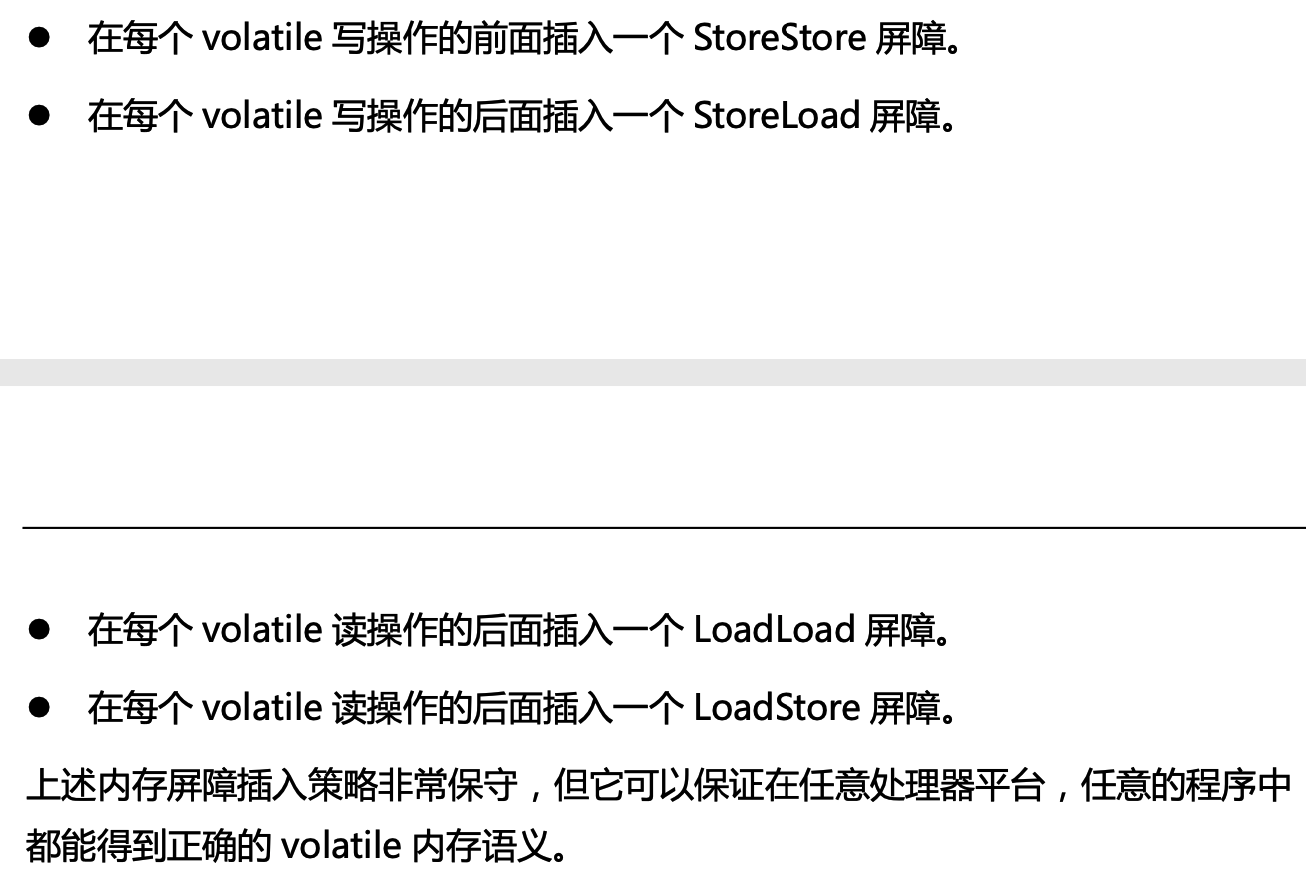


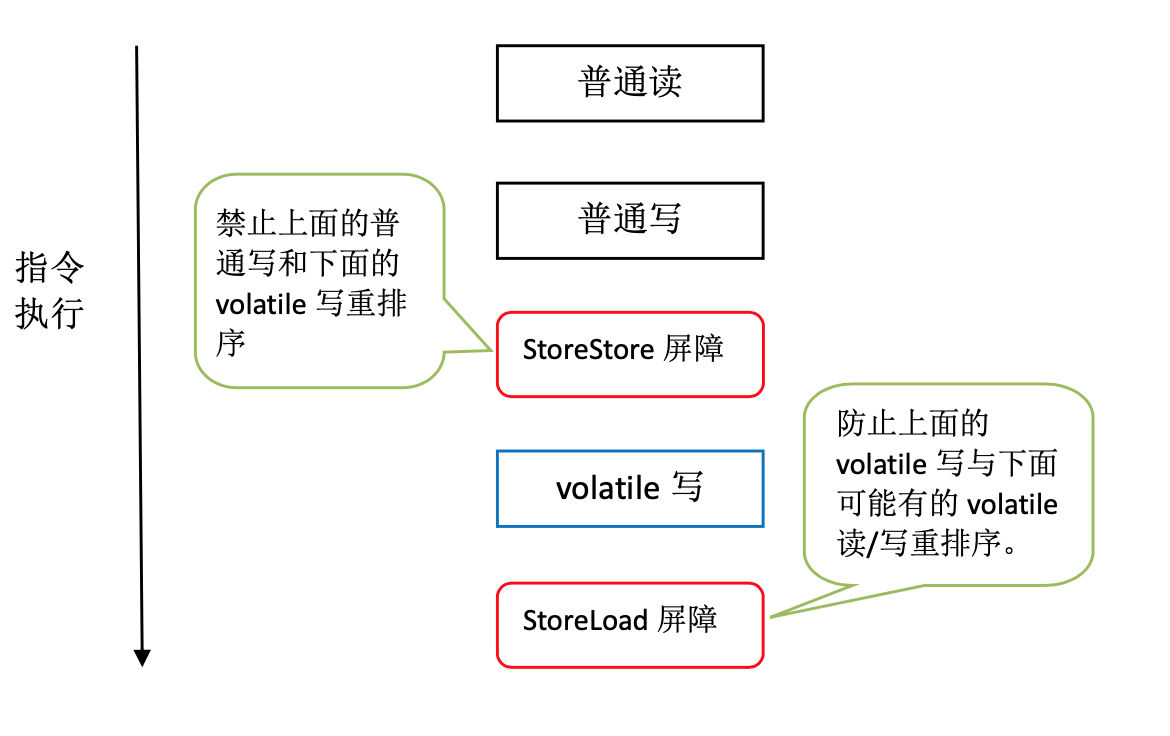
A线程在写volatile变量之前所有可见的共享变量，在B线程读同一个volatile变量后，将立即变得对B可见。

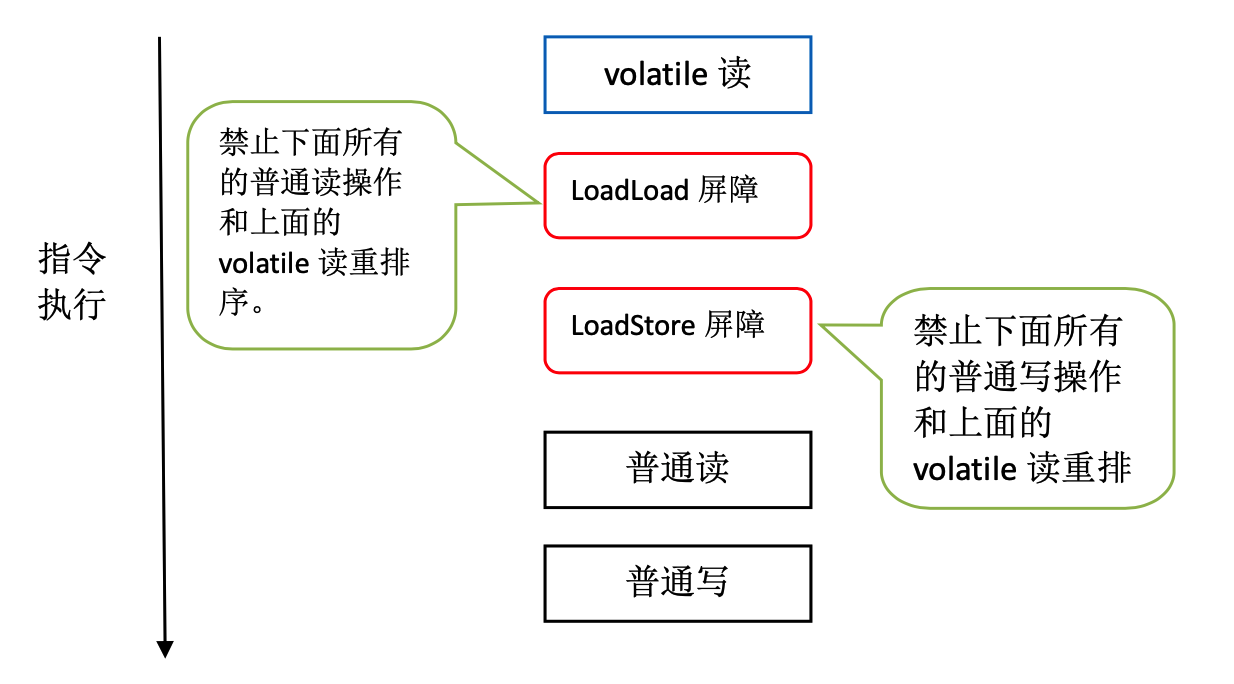
1. volatile内存语义的实现
   1. 针对编译器重排序，制定volatile重排序规则表限制编译器重排序



* 1. 针对处理器重排序，在生成字节码时，通过在指令序列中插入内存屏障来禁止特定类型的处理器重排序。JMM采取保守的内存屏障插入策略，如下



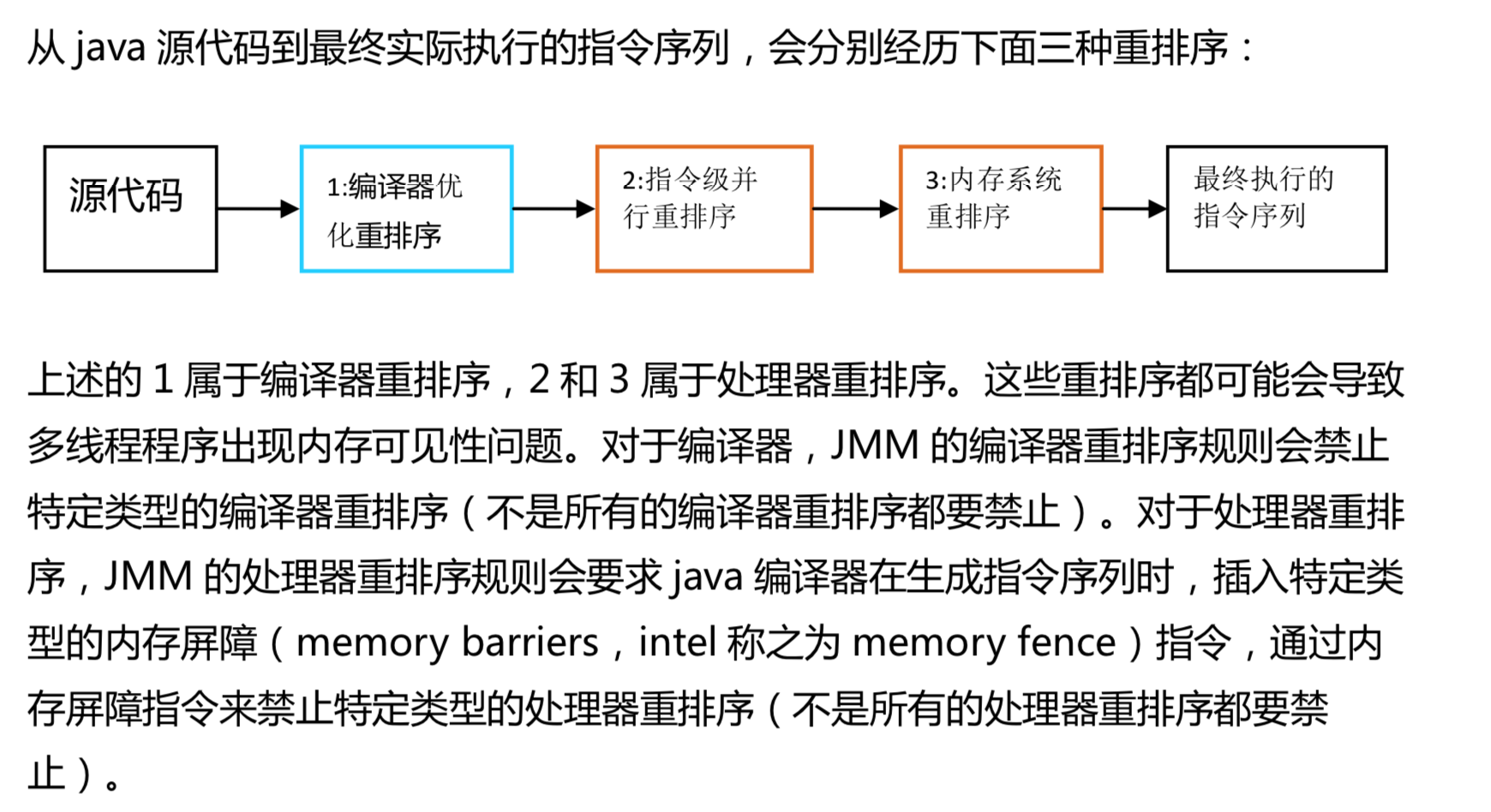




1. 原子性

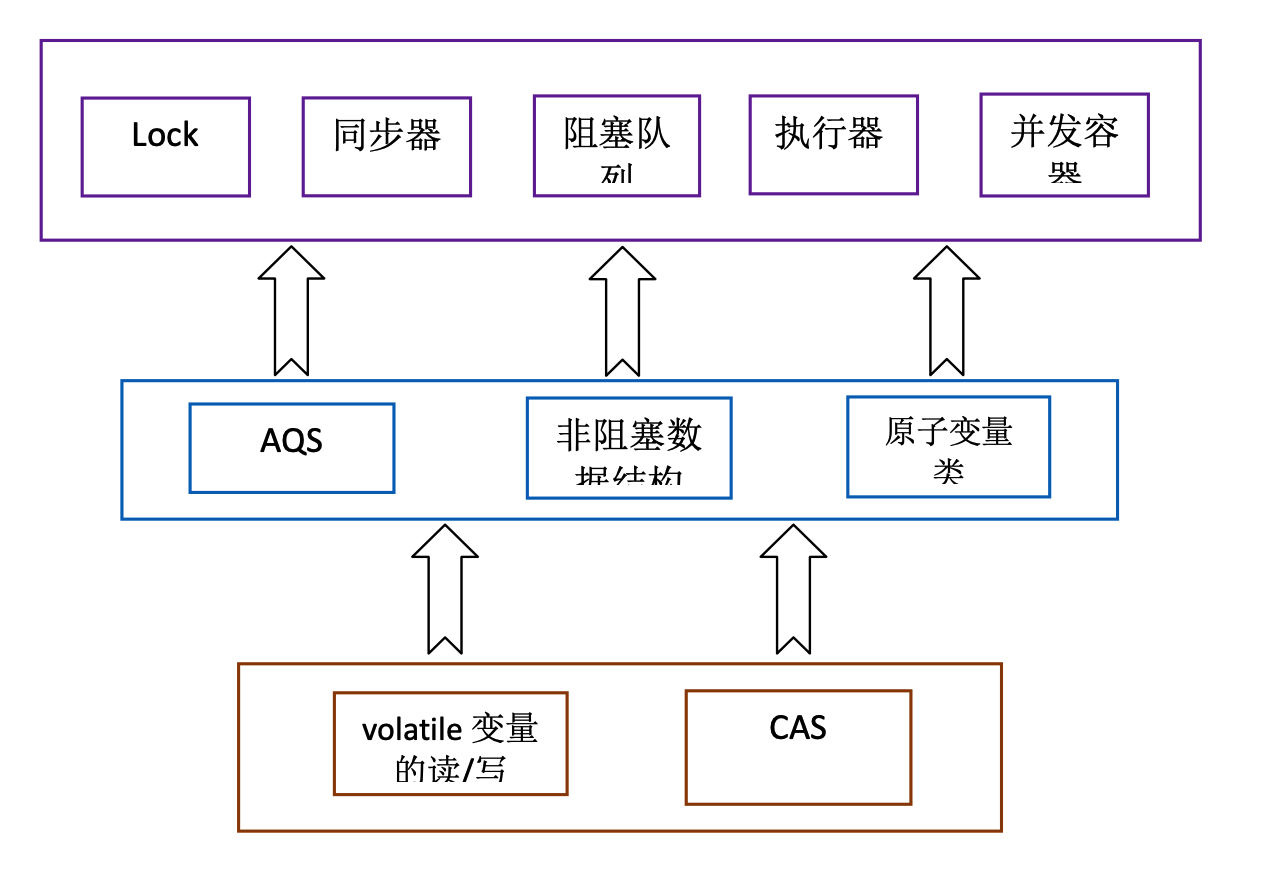
number++操作在多线程环境下是非线程安全的，需要使用AtomicInteger或者加锁解决

1. 重排序



1. 由于Java的CAS同时具有volatile读和volatile写的内存语义，因此Java线程之间的通信有以下四种方式
   1. A线程写volatile变量，随后B线程读这个volatile变量
   2. A线程写volatile变量，随后B线程用CAS更新这个volatile变量
   3. A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量
   4. A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量
2. 仔细分析JUC包的源代码实现，会发现一个通用化的实现模式
   1. 首先，声明共享变量为volatile
   2. 然后，使用CAS的原子条件更新来实现线程之间的同步
   3. 同时，配合volatile的读/写和CAS所具有的volatile读/写的内存语义来实现线程之间的通信

从整体看，JUC包的实现示意图如下



1. 案例
   1. 单例模式Double Check Lock写法instance必须采用volatile修饰防止重排序造成的类初始化未完成就使用的问题