深入理解Java内存模型

极客学院出版

前言

Java 线程之间的通信对程序员完全透明，内存可见性问题很容易困扰 Java 程序员，本文试图揭开 Java 内存模型神秘的面纱。本教程大致分三部分：重排序与顺序一致性；三个同步原语（lock，volatile，final）的内存语义，重排序规则及在处理器中的实现；Java 内存模型的设计目标，及其与处理器内存模型和顺序一致性内存模型的关系。

适用人群

适用于 Java 初学者的进阶学习，帮助读者更好的理解 Java 语言对内容的管理。

学习前提

学习本教程前，我们假设你已经掌握了 Java 编程语言的基础部分。

鸣谢：http://ifeve.com/java-memory-model-0/

目录

前言 1

第 1 章　基础 4

并发编程模型的分类 5

Java 内存模型的抽象 6

重排序 8

处理器重排序与内存屏障指令 9

happens-before 12

第 2 章　重排序 8

数据依赖性 15

as-if-serial 语义 16

程序顺序规则 17

重排序对多线程的影响 18

第 3 章　顺序一致性 21

数据竞争与顺序一致性保证 22

顺序一致性内存模型 23

同步程序的顺序一致性效果 26

未同步程序的执行特性 28

第 4 章　volatile 31

volatile 的特性 32

volatile 的写-读建立的 happens before 关系 34

volatile 写-读的内存语义 36

volatile 内存语义的实现 38

JSR-133 为什么要增强 volatile 的内存语义 43

第 5 章　锁 44
锁的释放-获取建立的 happens before 关系 45
锁释放和获取的内存语义 47
锁内存语义的实现 49
concurrent 包的实现 55
第 6 章　final 57
写 final 域的重排序规则 59
读 final 域的重排序规则 61
如果 final 域是引用类型 63
为什么 final 引用不能从构造函数内“逸出” 65
final 语义在处理器中的实现 67
JSR-133 为什么要增强 final 的语义 68
第 7 章　总结 69
处理器内存模型 70
JMM，处理器内存模型与顺序一致性内存模型之间的关系 72
JMM 的设计 73
JMM 的内存可见性保证 76
JSR-133 对旧内存模型的修补 78

1 基础

并发编程模型的分类

在并发编程中，我们需要处理两个关键问题：线程之间如何通信及线程之间如何同步（这里的线程是指并发执行的活动实体）。通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种：共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。

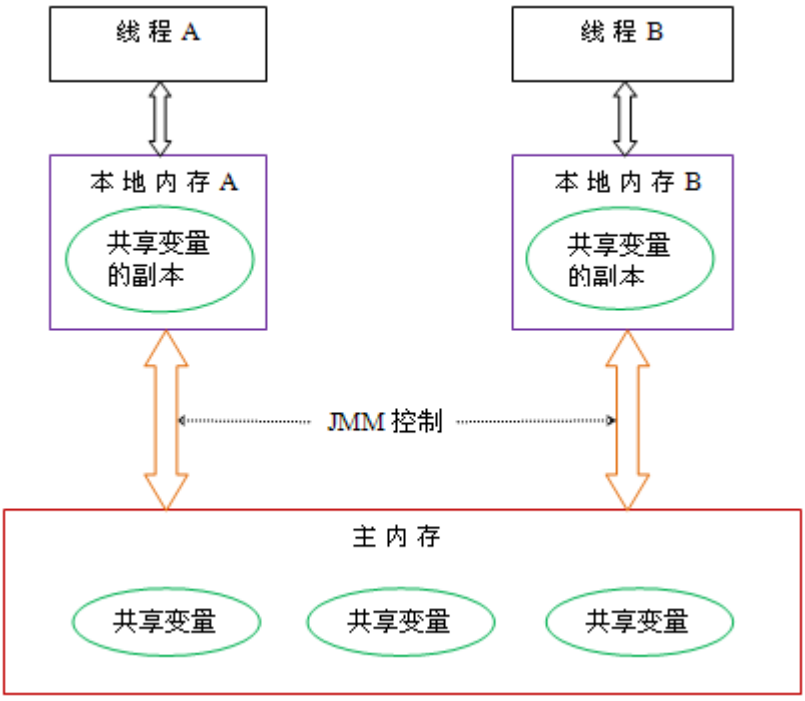
同步是指程序用于控制不同线程之间操作发生相对顺序的机制。在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java 的并发采用的是共享内存模型，Java 线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的 Java 程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

Java 内存模型的抽象

在 java 中，所有实例域、静态域和数组元素存储在堆内存中，堆内存在线程之间共享（本文使用“共享变量”这个术语代指实例域，静态域和数组元素）。局部变量（Local variables），方法定义参数（java 语言规范称之为 formal method parameters）和异常处理器参数（exception handler parameters）不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

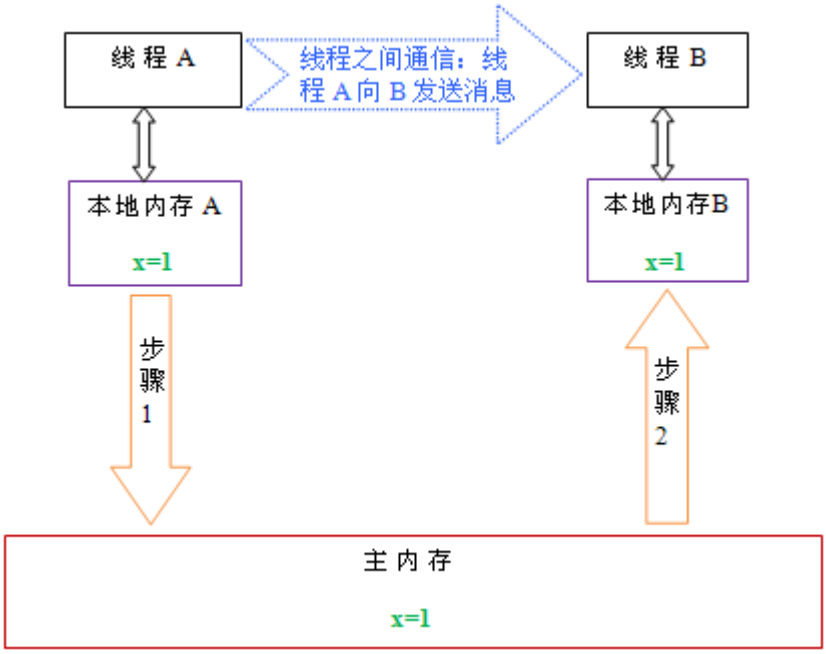
Java 线程之间的通信由 Java 内存模型（本文简称为 JMM）控制，JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java 内存模型的抽象示意图如下：



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 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。
2. 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。

下面通过示意图来说明这两个步骤：



如上图所示，本地内存 A 和 B 有主内存中共享变量 x 的副本。假设初始时，这三个内存中的 x 值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存 A 中。当线程 A 和线程 B 需要通信时，线程 A 首先会把自己本地内存中修改后的 x 值刷新到主内存中，此时主内存中的 x 值变为了1。随后，线程 B 到主内存中去读取线程 A 更新后的 x 值，此时线程 B 的本地内存的 x 值也变为了1。

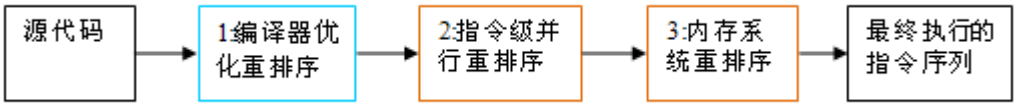
从整体来看，这两个步骤实质上是线程 A 在向线程 B 发送消息，而且这个通信过程必须要经过主内存。JMM 通过控制主内存与每个线程的本地内存之间的交互，来为 java 程序员提供内存可见性保证。

重排序

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从 java 源代码到最终实际执行的指令序列，会分别经历下面三种重排序：



上述的1属于编译器重排序，2和3属于处理器重排序。这些重排序都可能会导致多线程程序出现内存可见性问题。对于编译器，JMM 的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。对于处理器重排序，JMM 的处理器重排序规则会要求 java 编译器在生成指令序列时，插入特定类型的内存屏障（memory barriers，intel 称之为 memory fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序（不是所有的处理器重排序都要禁止）。

JMM 属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

处理器重排序与内存屏障指令

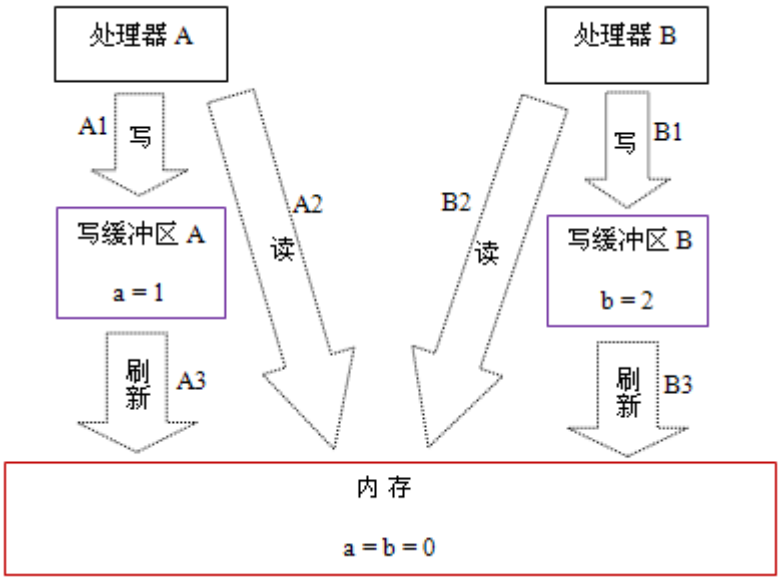
现代的处理器使用写缓冲区来临时保存向内存写入的数据。写缓冲区可以保证指令流水线持续运行，它可以避免由于处理器停顿下来等待向内存写入数据而产生的延迟。同时，通过以批处理的方式刷新写缓冲区，以及合并写缓冲区中对同一内存地址的多次写，可以减少对内存总线的占用。虽然写缓冲区有这么多好处，但每个处理器上的写缓冲区，仅仅对它所在的处理器可见。这个特性会对内存操作的执行顺序产生重要的影响：处理器对内存的读/写操作的执行顺序，不一定与内存实际发生的读/写操作顺序一致！为了具体说明，请看下面示例：

Processor A	Processor B

a = 1; //A1
x = b; //A2
	b = 2; //B1
	y = a; //B2

初始状态：a = b = 0 处理器允许执行后得到结果：x = y = 0	

假设处理器 A 和处理器 B 按程序的顺序并行执行内存访问，最终却可能得到 x = y = 0 的结果。具体的原因如下图所示：



这里处理器 A 和处理器 B 可以同时把共享变量写入自己的写缓冲区（A1，B1），然后从内存中读取另一个共享变量（A2，B2），最后才把自己写缓存区中保存的脏数据刷新到内存中（A3，B3）。当以这种时序执行时，程序就可以得到 x = y = 0 的结果。

从内存操作实际发生的顺序来看，直到处理器 A 执行 A3 来刷新自己的写缓存区，写操作 A1 才算真正执行了。虽然处理器 A 执行内存操作的顺序为：A1->A2，但内存操作实际发生的顺序却是：A2->A1。此时，处理器 A 的内存操作顺序被重排序了（处理器 B 的情况和处理器 A 一样，这里就不赘述了）。

这里的关键是，由于写缓冲区仅对自己的处理器可见，它会导致处理器执行内存操作的顺序可能会与内存实际的操作执行顺序不一致。由于现代的处理器都会使用写缓冲区，因此现代的处理器都会允许对写-读操作重排序。

下面是常见处理器允许的重排序类型的列表：

	Load-Load	Load-Store	Store-Store	Store-Load	数据依赖
sparc-TSO	N	N	N	Y	N
x86	N	N	N	Y	N
ia64	Y	Y	Y	Y	N
PowerPC	Y	Y	Y	Y	N

上表单元格中的 “N” 表示处理器不允许两个操作重排序，“Y” 表示允许重排序。

从上表我们可以看出：常见的处理器都允许 Store-Load 重排序；常见的处理器都不允许对存在数据依赖的操作做重排序。sparc-TSO 和 x86 拥有相对较强的处理器内存模型，它们仅允许对写-读操作做重排序（因为它们都使用了写缓冲区）。

※注1：sparc-TSO 是指以 TSO(Total Store Order)内存模型运行时，sparc 处理器的特性。

※注2：上表中的 x86 包括 x64 及 AMD64。

※注3：由于 ARM 处理器的内存模型与 PowerPC 处理器的内存模型非常类似，本文将忽略它。

※注4：数据依赖性后文会专门说明。

为了保证内存可见性，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。JMM 把内存屏障指令分为下列四类：

屏障类型	指令示例	说明
LoadLoad Barriers	Load1; LoadLoad; Load2	确保Load1数据的装载，之前于Load2及所有后续装载指令的装载。
StoreStore Barriers	Store1; StoreStore; Store2	确保Store1数据对其他处理器可见（刷新到内存），之前于Store2及所有后续存储指令的存储。
LoadStore Barriers	Load1; LoadStore; Store2	确保Load1数据装载，之前于Store2及所有后续的存储指令刷新到内存。
StoreLoad Barriers	Store1; StoreLoad; Load2	确保Store1数据对其他处理器变得可见（指刷新到内存），之前于Load2及所有后续装载指令的装载。StoreLoad Barriers会使该屏障之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令。

StoreLoad Barriers 是一个“全能型”的屏障，它同时具有其他三个屏障的效果。现代的多处理器大都支持该屏障（其他类型的屏障不一定被所有处理器支持）。执行该屏障开销会很昂贵，因为当前处理器通常要把写缓冲区中的数据全部刷新到内存中（buffer fully flush）。

happens-before

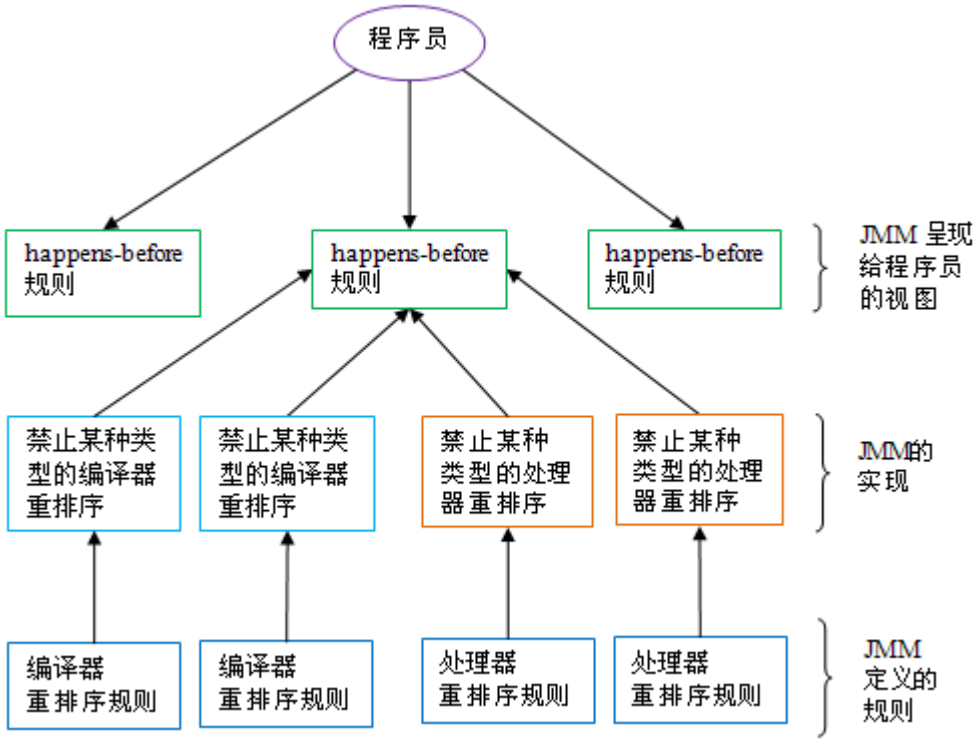
从 JDK5 开始，java 使用新的 JSR -133 内存模型（本文除非特别说明，针对的都是 JSR- 133 内存模型）。JSR-133 使用 happens-before 的概念来阐述操作之间的内存可见性。在 JMM 中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before 关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

与程序员密切相关的 happens-before 规则如下：

- 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。
- 监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。
- 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。
- 传递性：如果 A happens- before B，且 B happens- before C，那么 A happens- before C。

注意，两个操作之间具有 happens-before 关系，并不意味着前一个操作必须要在后一个操作之前执行！happens-before 仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前（the first is visible to and ordered before the second）。happens- before 的定义很微妙，后文会具体说明 happens-before 为什么要这么定义。

happens-before与JMM的关系如下图所示：



如上图所示，一个 happens-before 规则通常对应于多个编译器和处理器重排序规则。对于 java 程序员来说，happens-before 规则简单易懂，它避免 java 程序员为了理解 JMM 提供的内存可见性保证而去学习复杂的重排序规则以及这些规则的具体实现。

2

重排序

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性。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。

前面提到过，编译器和处理器可能会对操作做重排序。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。

注意，这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

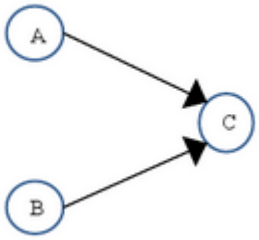
as-if-serial 语义

as-if-serial 语义的意思指：不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器，runtime 和处理器都必须遵守 as-if-serial 语义。

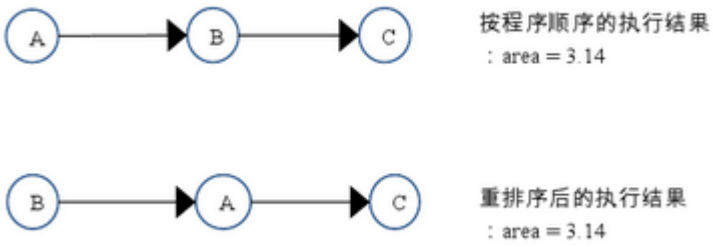
为了遵守 as-if-serial 语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作可能被编译器和处理器重排序。为了具体说明，请看下面计算圆面积的代码示例：

```
double pi  = 3.14;    //A
double r   = 1.0;     //B
double area = pi * r * r; //C
```

上面三个操作的数据依赖关系如下图所示：



如上图所示，A 和 C 之间存在数据依赖关系，同时 B 和 C 之间也存在数据依赖关系。因此在最终执行的指令序列中，C 不能被重排序到 A 和 B 的前面（C 排到 A 和 B 的前面，程序的结果将会被改变）。但 A 和 B 之间没有数据依赖关系，编译器和处理器可以重排序 A 和 B 之间的执行顺序。下图是该程序的两种执行顺序：



as-if-serial 语义把单线程程序保护了起来，遵守 as-if-serial 语义的编译器，runtime 和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。as-if-serial 语义使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。

程序顺序规则

根据 happens- before 的程序顺序规则，上面计算圆的面积的示例代码存在三个 happens- before 关系：

1. A happens- before B；
2. B happens- before C；
3. A happens- before C；

这里的第3个 happens- before 关系，是根据 happens- before 的传递性推导出来的。

这里 A happens- before B，但实际执行时 B 却可以排在 A 之前执行（看上面的重排序后的执行顺序）。在第一章提到过，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（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。这里操作 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 后的执行结果，与操作 A 和操作 B 按 happens- before 顺序执行的结果一致。在这种情况下，JMM 会认为这种重排序并不非法（not illegal），JMM 允许这种重排序。

在计算机中，软件技术和硬件技术有一个共同的目标：在不改变程序执行结果的前提下，尽可能的开发并行度。编译器和处理器遵从这一目标，从 happens- before 的定义我们可以看出，JMM 同样遵从这一目标。

重排序对多线程的影响

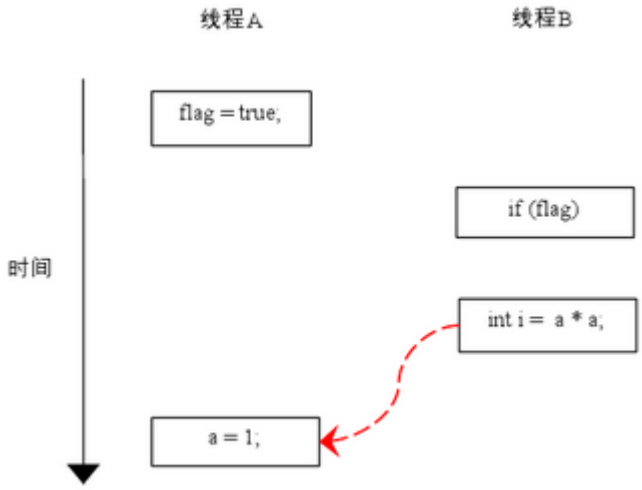
现在让我们来看看，重排序是否会改变多线程程序的执行结果。请看下面的示例代码：

```
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。

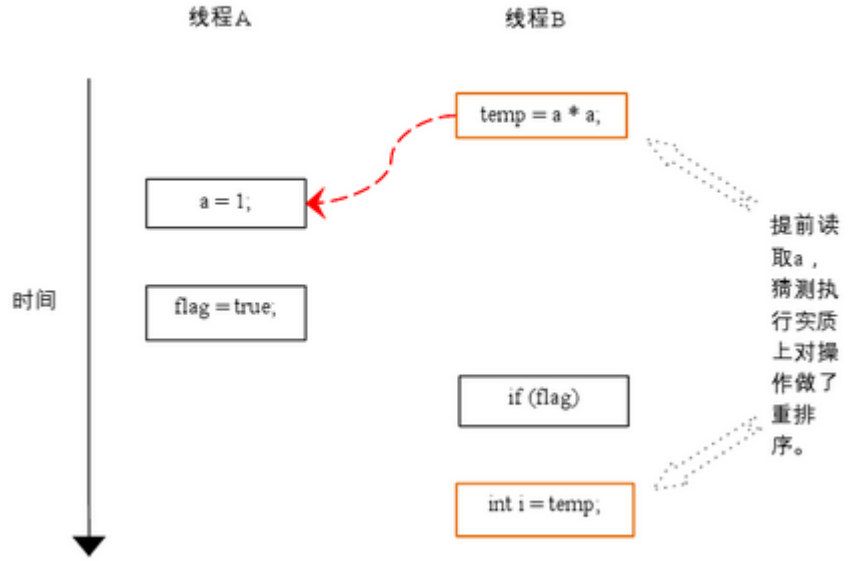
由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。让我们先来看看，当操作1和操作2重排序时，可能会产生什么效果？请看下面的程序执行时序图：



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！

※注：本文统一用红色的虚箭线表示错误的读操作，用绿色的虚箭线表示正确的读操作。

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，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。

3

顺序一致性

数据竞争与顺序一致性保证

当程序未正确同步时，就会存在数据竞争。java 内存模型规范对数据竞争的定义如下：

- 在一个线程中写一个变量，
- 在另一个线程读同一个变量，
- 而且写和读没有通过同步来排序。

当代码中包含数据竞争时，程序的执行往往产生违反直觉的结果（前一章的示例正是如此）。如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

JMM 对正确同步的多线程程序的内存一致性做了如下保证：

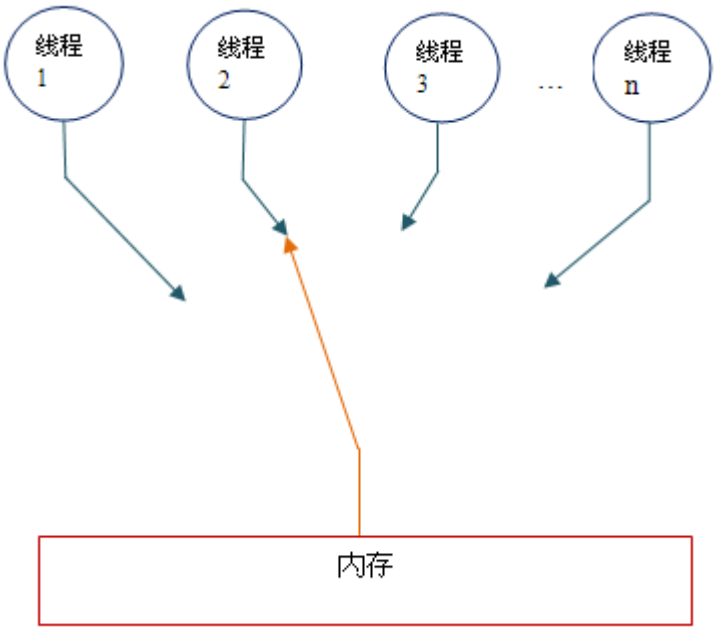
- 如果程序是正确同步的，程序的执行将具有顺序一致性（sequentially consistent）–即程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同（马上我们将会看到，这对于程序员来说是一个极强的保证）。这里的同步是指广义上的同步，包括对常用同步原语（lock，volatile 和 final）的正确使用。

顺序一致性内存模型

顺序一致性内存模型是一个被计算机科学家理想化了的理论参考模型，它为程序员提供了极强的内存可见性保证。顺序一致性内存模型有两大特性：

- 一个线程中的所有操作必须按照程序的顺序来执行。
- （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。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

顺序一致性内存模型为程序员提供的视图如下：

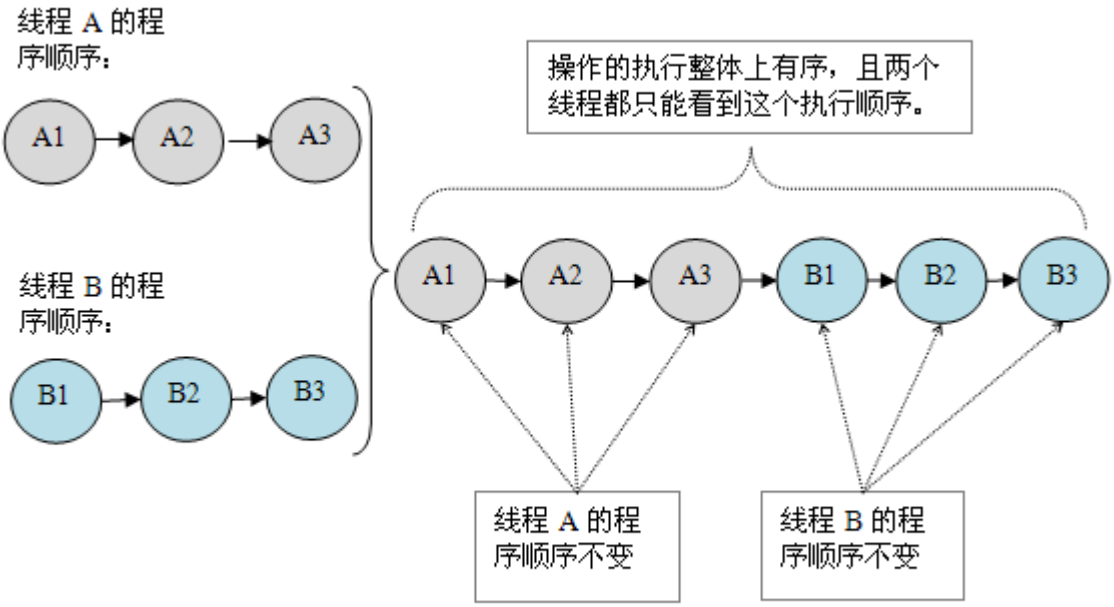


在概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关可以连接到任意一个线程。同时，每一个线程必须按程序的顺序来执行内存读/写操作。从上图我们可以看出，在任意时间点最多只能有一个线程可以连接到内存。当多个线程并发执行时，图中的开关装置能把所有线程的所有内存读/写操作串行化。

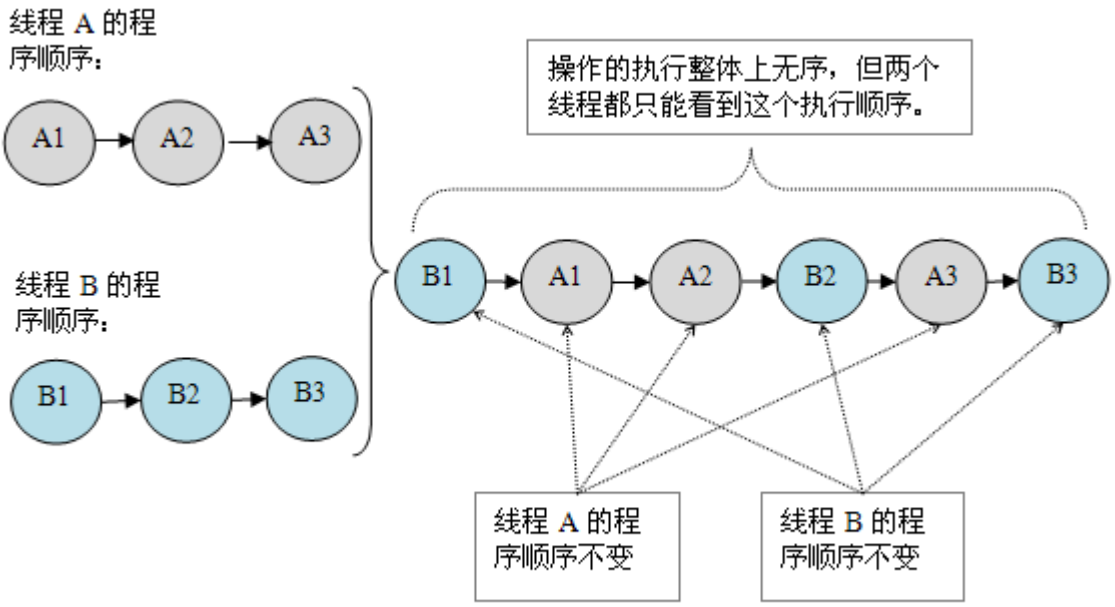
为了更好的理解，下面我们通过两个示意图来对顺序一致性模型的特性做进一步的说明。

假设有两个线程A和B并发执行。其中 A 线程有三个操作，它们在程序中的顺序是：A1->A2->A3。B线程也有三个操作，它们在程序中的顺序是：B1->B2->B3。

假设这两个线程使用监视器来正确同步：A 线程的三个操作执行后释放监视器，随后 B 线程获取同一个监视器。那么程序在顺序一致性模型中的执行效果将如下图所示：



现在我们再假设这两个线程没有做同步，下面是这个未同步程序在顺序一致性模型中的执行示意图：



未同步程序在顺序一致性模型中虽然整体执行顺序是无序的，但所有线程都只能看到一个一致的整体执行顺序。以上图为例，线程 A 和 B 看到的执行顺序都是：B1->A1->A2->B2->A3->B3。之所以能得到这个保证是因为顺序一致性内存模型中的每个操作必须立即对任意线程可见。

但是，在 JMM 中就没有这个保证。未同步程序在 JMM 中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。比如，在当前线程把写过的数据缓存在本地内存中，且还没有刷新到主内存之

前，这个写操作仅对当前线程可见；从其他线程的角度来观察，会认为这个写操作根本还没有被当前线程执行。只有当前线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，当前线程和其它线程看到的操作执行顺序将不一致。

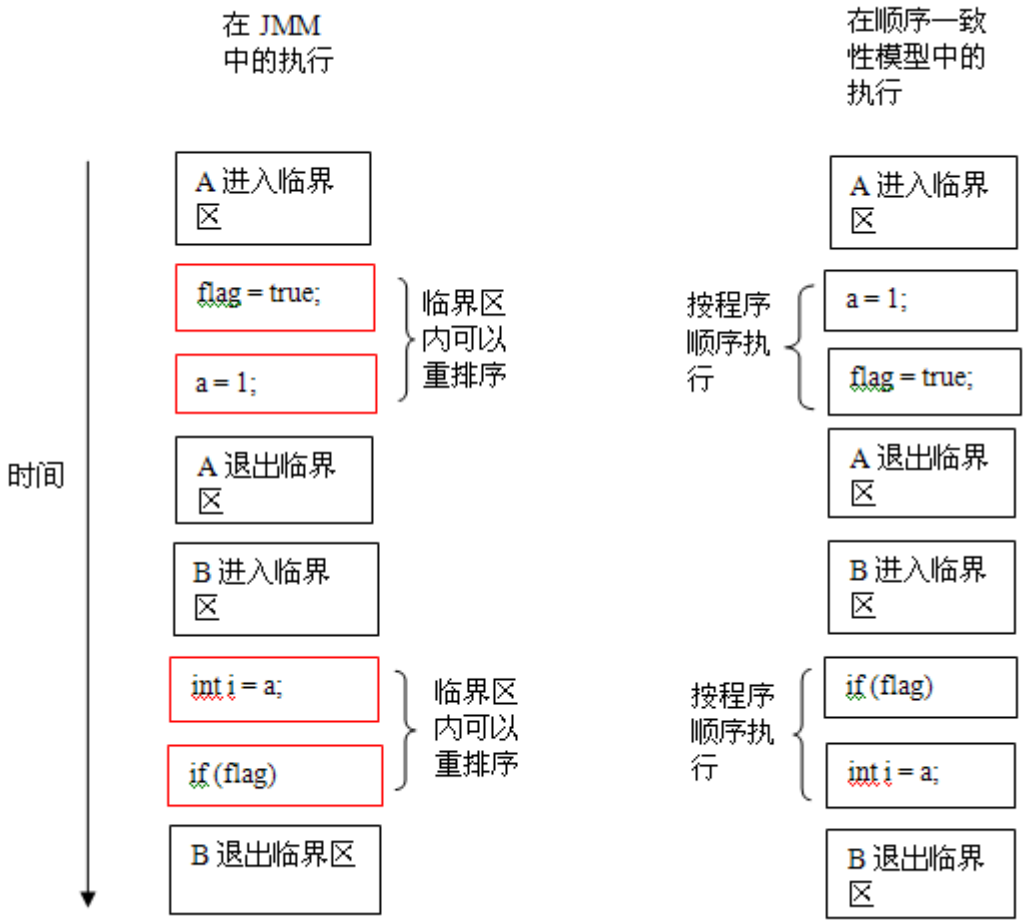
同步程序的顺序一致性效果

下面我们对前面的示例程序 ReorderExample 用监视器来同步，看看正确同步的程序如何具有顺序一致性。

请看下面的示例代码：

```
class SynchronizedExample {
int a = 0;
boolean flag = false;

public synchronized void writer() {
    a = 1;
    flag = true;
}

public synchronized void reader() {
    if (flag) {
        int i = a;
        ……
    }
}
}
```

上面示例代码中，假设 A 线程执行 writer() 方法后，B 线程执行 reader() 方法。这是一个正确同步的多线程程序。根据 JMM 规范，该程序的执行结果将与该程序在顺序一致性模型中的执行结果相同。下面是该程序在两个内存模型中的执行时序对比图：



在顺序一致性模型中，所有操作完全按程序的顺序串行执行。而在 JMM 中，临界区内的代码可以重排序（但 JMM 不允许临界区内的代码“逸出”到临界区之外，那样会破坏监视器的语义）。JMM会在退出监视器和进入监视器这两个关键时间点做一些特别处理，使得线程在这两个时间点具有与顺序一致性模型相同的内存视图（具体细节后文会说明）。虽然线程 A 在临界区内做了重排序，但由于监视器的互斥执行的特性，这里的线程 B 根本无法“观察”到线程 A 在临界区内的重排序。这种重排序既提高了执行效率，又没有改变程序的执行结果。

从这里我们可以看到 JMM 在具体实现上的基本方针：在不改变（正确同步的）程序执行结果的前提下，尽可能的为编译器和处理器的优化打开方便之门。

未同步程序的执行特性

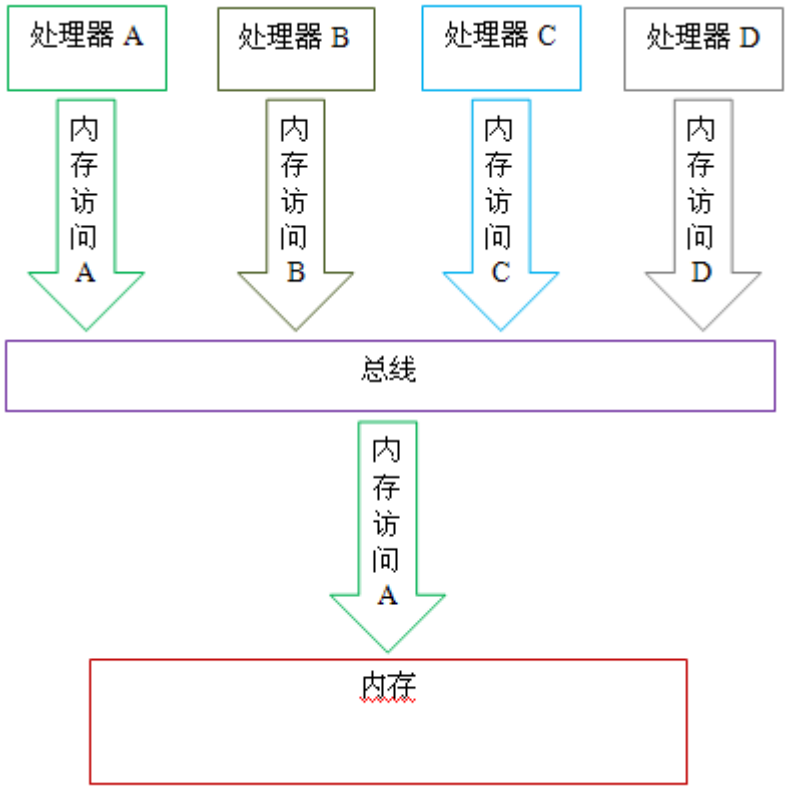
对于未同步或未正确同步的多线程程序，JMM 只提供最小安全性：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0，null，false），JMM 保证线程读操作读取到的值不会无中生有（out of thin air）的冒出来。为了实现最小安全性，JVM 在堆上分配对象时，首先会清零内存空间，然后才会在上面分配对象（JVM内部会同步这两个操作）。因此，在以清零的内存空间（pre-zeroed memory）分配对象时，域的默认初始化已经完成了。

JMM 不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致。因为未同步程序在顺序一致性模型中执行时，整体上是无序的，其执行结果无法预知。保证未同步程序在两个模型中的执行结果一致毫无意义。

和顺序一致性模型一样，未同步程序在 JMM 中的执行时，整体上也是无序的，其执行结果也无法预知。同时，未同步程序在这两个模型中的执行特性有下面几个差异：

1. 顺序一致性模型保证单线程内的操作会按程序的顺序执行，而 JMM 不保证单线程内的操作会按程序的顺序执行（比如上面正确同步的多线程程序在临界区内的重排序）。这一点前面已经讲过了，这里就不再赘述。
2. 顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而JMM不保证所有线程能看到一致的操作执行顺序。这一点前面也已经讲过，这里就不再赘述。
3. JMM不保证对64位的 long 型和 double 型变量的读/写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。

第3个差异与处理器总线的工作机制密切相关。在计算机中，数据通过总线在处理器和内存之间传递。每次处理器和内存之间的数据传递都是通过一系列步骤来完成的，这一系列步骤称之为总线事务（bus transaction）。总线事务包括读事务（read transaction）和写事务（write transaction）。读事务从内存传送数据到处理器，写事务从处理器传送数据到内存，每个事务会读/写内存中一个或多个物理上连续的字。这里的关键是，总线会同步试图并发使用总线的事务。在一个处理器执行总线事务期间，总线会禁止其它所有的处理器和 I/O 设备执行内存的读/写。下面让我们通过一个示意图来说明总线的工作机制：

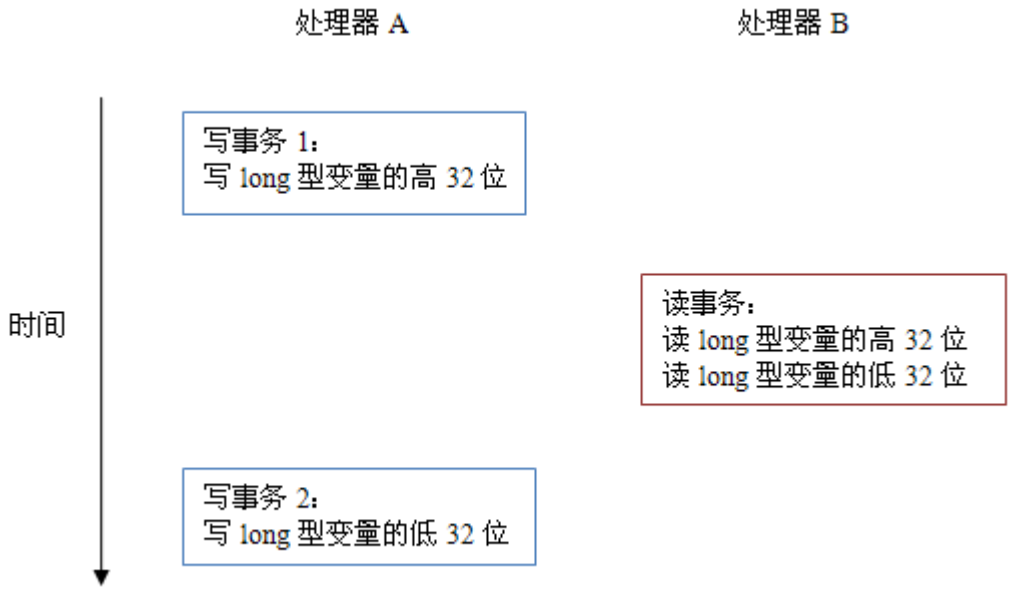


如上图所示，假设处理器 A，B 和 C 同时向总线发起总线事务，这时总线仲裁（bus arbitration）会对竞争作出裁决，这里我们假设总线在仲裁后判定处理器A在竞争中获胜（总线仲裁会确保所有处理器都能公平的访问内存）。此时处理器 A 继续它的总线事务，而其它两个处理器则要等待处理器A的总线事务完成后才能开始再次执行内存访问。假设在处理器 A 执行总线事务期间（不管这个总线事务是读事务还是写事务），处理器D向总线发起了总线事务，此时处理器 D 的这个请求会被总线禁止。

总线的这些工作机制可以把所有处理器对内存的访问以串行化的方式来执行；在任意时间点，最多只能有一个处理器能访问内存。这个特性确保了单个总线事务之中的内存读/写操作具有原子性。

在一些32位的处理器上，如果要求对64位数据的写操作具有原子性，会有比较大的开销。为了照顾这种处理器，java 语言规范鼓励但不强求 JVM 对64位的 long 型变量和 double 型变量的写具有原子性。当 JVM 在这种处理器上运行时，会把一个64位 long/ double 型变量的写操作拆分为两个32位的写操作来执行。这两个32位的写操作可能会被分配到不同的总线事务中执行，此时对这个64位变量的写将不具有原子性。

当单个内存操作不具有原子性，将可能会产生意想不到后果。请看下面示意图：



如上图所示，假设处理器 A 写一个 long 型变量，同时处理器 B 要读这个 long 型变量。处理器 A 中64位的写操作被拆分为两个32位的写操作，且这两个32位的写操作被分配到不同的写事务中执行。同时处理器B中64位的读操作被分配到单个的读事务中执行。当处理器 A 和 B 按上图的时序来执行时，处理器B将看到仅仅被处理器 A “写了一半“的无效值。

注意，在 JSR −133 之前的旧内存模型中，一个64位 long/ double 型变量的读/写操作可以被拆分为两个32位的读/写操作来执行。从 JSR −133 内存模型开始（即从 JDK5 开始），仅仅只允许把一个64位 long/ double 型变量的写操作拆分为两个32位的写操作来执行，任意的读操作在 JSR −133 中都必须具有原子性（即任意读操作必须要在单个读事务中执行）。

volatile

volatile 的特性

当我们声明共享变量为 volatile 后，对这个变量的读/写将会很特别。理解 volatile 特性的一个好方法是：把对 volatile 变量的单个读/写，看成是使用同一个锁对这些单个读/写操作做了同步。下面我们通过具体的示例来说明，请看下面的示例代码：

```
class VolatileFeaturesExample {
   //使用volatile声明64位的long型变量
   volatile long vl = 0L;

   public void set(long l) {
      vl = l;   //单个volatile变量的写
   }

   public void getAndIncrement () {
      vl++;   //复合（多个）volatile变量的读/写
   }

   public long get() {
      return vl;   //单个volatile变量的读
   }
}
```

假设有多个线程分别调用上面程序的三个方法，这个程序在语义上和下面程序等价：

```
class VolatileFeaturesExample {
   long vl = 0L;               // 64位的long型普通变量

   //对单个的普通 变量的写用同一个锁同步
   public synchronized void set(long l) {
      vl = l;
   }

   public void getAndIncrement () { //普通方法调用
      long temp = get();          //调用已同步的读方法
      temp += 1L;                 //普通写操作
      set(temp);                  //调用已同步的写方法
   }
   public synchronized long get() {
      //对单个的普通变量的读用同一个锁同步
      return vl;
```

```
  }
}
```

如上面示例程序所示，对一个 volatile 变量的单个读/写操作，与对一个普通变量的读/写操作使用同一个锁来同步，它们之间的执行效果相同。

锁的 happens-before 规则保证释放锁和获取锁的两个线程之间的内存可见性，这意味着对一个volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入。

锁的语义决定了临界区代码的执行具有原子性。这意味着即使是64位的 long 型和 double 型变量，只要它是 volatile 变量，对该变量的读写就将具有原子性。如果是多个 volatile 操作或类似于 volatile++ 这种复合操作，这些操作整体上不具有原子性。

简而言之，volatile 变量自身具有下列特性：

- 可见性：对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入。
- 原子性：对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++ 这种复合操作不具有原子性。

volatile 的写-读建立的 happens before 关系

上面讲的是 volatile 变量自身的特性，对程序员来说，volatile 对线程的内存可见性的影响比 volatile 自身的特性更为重要，也更需要我们去关注。

从 JSR-133 开始，volatile 变量的写-读可以实现线程之间的通信。

从内存语义的角度来说，volatile 与锁有相同的效果：volatile 写和锁的释放有相同的内存语义；volatile 读与锁的获取有相同的内存语义。

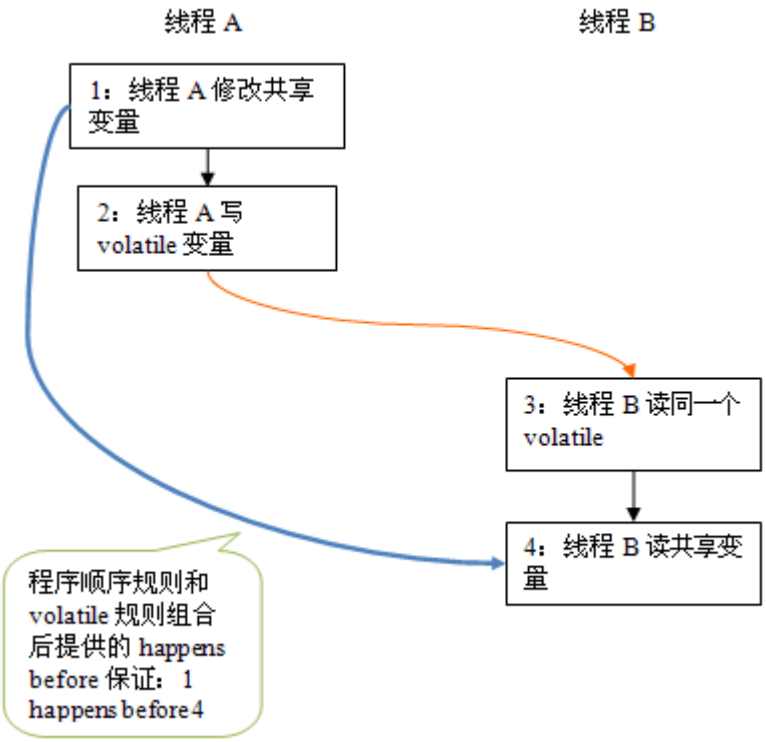
请看下面使用volatile变量的示例代码：

```
class VolatileExample {
   int a = 0;
   volatile boolean flag = false;

   public void writer() {
      a = 1;                  //1
      flag = true;               //2
   }

   public void reader() {
      if (flag) {                //3
         int i =  a;          //4
         ……
      }
   }
}
```

假设线程 A 执行 writer() 方法之后，线程 B 执行 reader() 方法。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。

上述 happens before 关系的图形化表现形式如下：



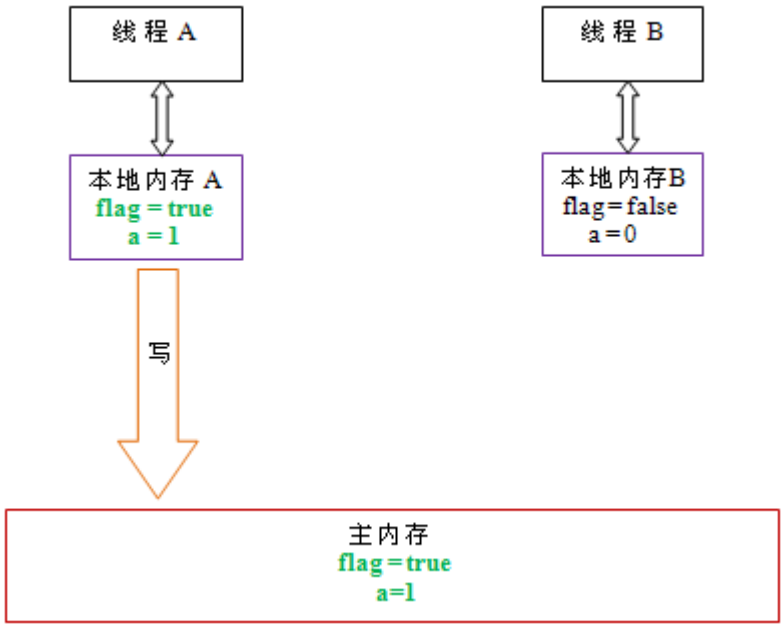
在上图中，每一个箭头链接的两个节点，代表了一个 happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示 volatile 规则；蓝色箭头表示组合这些规则后提供的 happens before 保证。

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。

volatile 写-读的内存语义

volatile 写的内存语义如下：

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刷新到主内存。以上面示例程序 VolatileExample 为例，假设线程 A 首先执行 writer() 方法，随后线程 B 执行 reader() 方法，初始时两个线程的本地内存中的 flag 和 a 都是初始状态。下图是线程 A 执行 volatile 写后，共享变量的状态示意图：

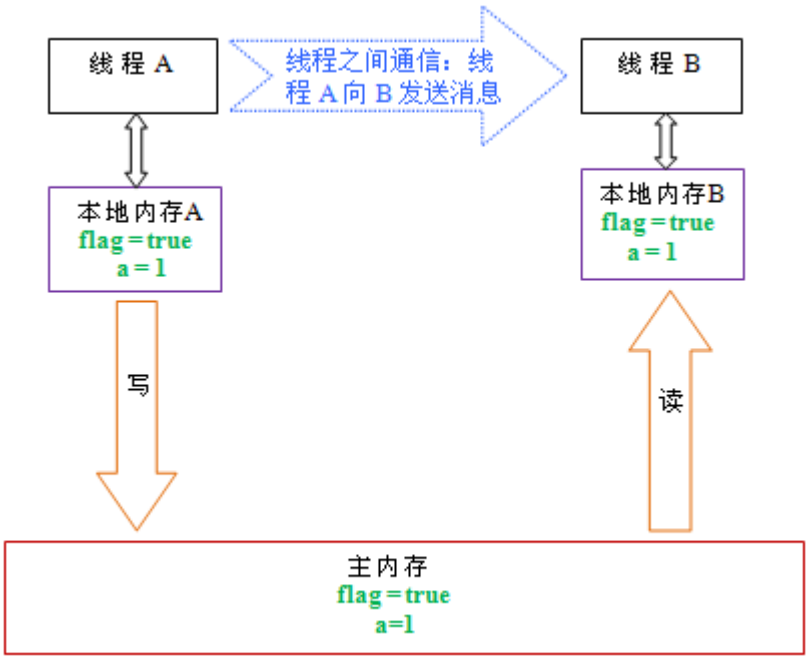


如上图所示，线程A在写flag变量后，本地内存A中被线程A更新过的两个共享变量的值被刷新到主内存中。此时，本地内存A和主内存中的共享变量的值是一致的。

volatile读的内存语义如下：

- 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。

下面是线程B读同一个 volatile 变量后，共享变量的状态示意图：



如上图所示，在读 flag 变量后，本地内存 B 已经被置为无效。此时，线程 B 必须从主内存中读取共享变量。线程 B 的读取操作将导致本地内存B与主内存中的共享变量的值也变成一致的了。

如果我们把 volatile 写和 volatile 读这两个步骤综合起来看的话，在读线程 B 读一个volatile 变量后，写线程 A 在写这个 volatile 变量之前所有可见的共享变量的值都将立即变得对读线程 B 可见。

下面对 volatile 写和 volatile 读的内存语义做个总结：

- 线程 A 写一个 volatile 变量，实质上是线程 A 向接下来将要读这个 volatile 变量的某个线程发出了（其对共享变量所在修改的）消息。
- 线程 B 读一个 volatile 变量，实质上是线程 B 接收了之前某个线程发出的（在写这个volatile 变量之前对共享变量所做修改的）消息。
- 线程A写一个 volatile 变量，随后线程 B 读这个 volatile 变量，这个过程实质上是线程A 通过主内存向线程 B 发送消息。

volatile 内存语义的实现

下面，让我们来看看 JMM 如何实现 volatile 写/读的内存语义。

前文我们提到过重排序分为编译器重排序和处理器重排序。为了实现 volatile 内存语义，JMM 会分别限制这两种类型的重排序类型。下面是 JMM 针对编译器制定的 volatile 重排序规则表：

是否能重排序	第二个操作		
第一个操作	普通读/写	volatile读	volatile写
普通读/写			NO
volatile读	NO	NO	NO
volatile写		NO	NO

举例来说，第三行最后一个单元格的意思是：在程序顺序中，当第一个操作为普通变量的读或写时，如果第二个操作为 volatile 写，则编译器不能重排序这两个操作。

从上表我们可以看出：

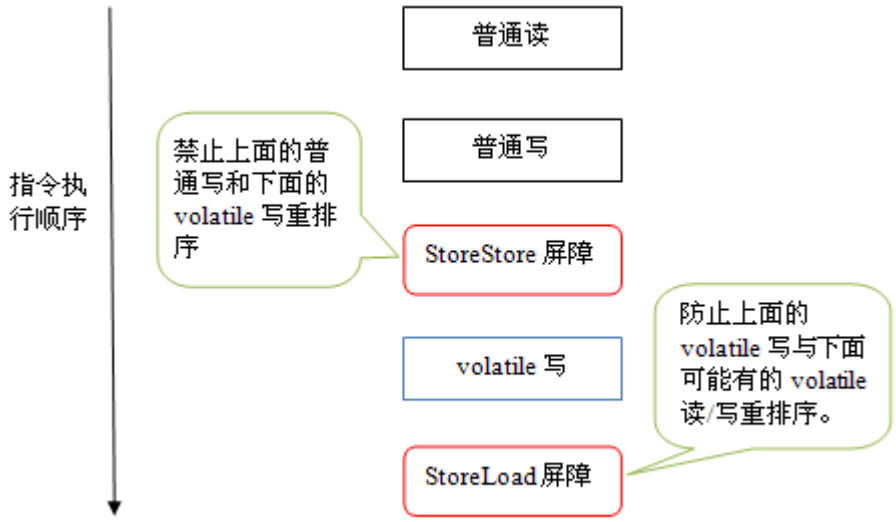
- 当第二个操作是 volatile 写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile 写之前的操作不会被编译器重排序到 volatile 写之后。
- 当第一个操作是 volatile 读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile 读之后的操作不会被编译器重排序到 volatile 读之前。
- 当第一个操作是 volatile 写，第二个操作是 volatile 读时，不能重排序。

为了实现 volatile 的内存语义，编译器在生成字节码时，会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。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，JMM 采取保守策略。下面是基于保守策略的 JMM 内存屏障插入策略：

- 在每个 volatile 写操作的前面插入一个 StoreStore 屏障。
- 在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。
- 在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。
- 在每个 volatile 读操作的后面插入一个 LoadStore 屏障。

上述内存屏障插入策略非常保守，但它可以保证在任意处理器平台，任意的程序中都能得到正确的volatile 内存语义。

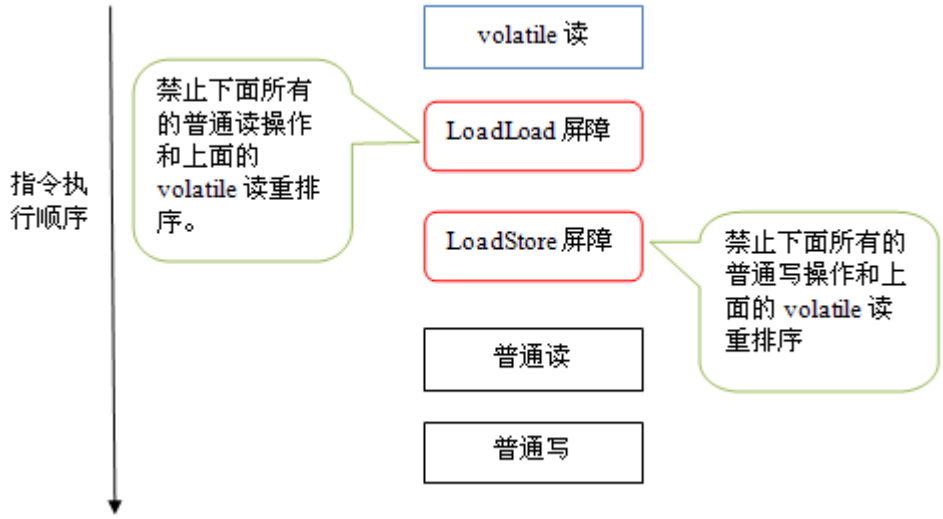
下面是保守策略下，volatile 写插入内存屏障后生成的指令序列示意图：



上图中的 StoreStore 屏障可以保证在 volatile 写之前，其前面的所有普通写操作已经对任意处理器可见了。这是因为 StoreStore 屏障将保障上面所有的普通写在 volatile 写之前刷新到主内存。

这里比较有意思的是 volatile 写后面的 StoreLoad 屏障。这个屏障的作用是避免 volatile写与后面可能有的 volatile 读/写操作重排序。因为编译器常常无法准确判断在一个 volatile写的后面，是否需要插入一个 StoreLoad 屏障（比如，一个 volatile 写之后方法立即return）。为了保证能正确实现 volatile 的内存语义，JMM 在这里采取了保守策略：在每个volatile 写的后面或在每个 volatile 读的前面插入一个 StoreLoad 屏障。从整体执行效率的角度考虑，JMM 选择了在每个 volatile 写的后面插入一个 StoreLoad 屏障。因为volatile 写-读内存语义的常见使用模式是：一个写线程写 volatile 变量，多个读线程读同一个 volatile 变量。当读线程的数量大大超过写线程时，选择在 volatile 写之后插入StoreLoad 屏障将带来可观的执行效率的提升。从这里我们可以看到 JMM 在实现上的一个特点：首先确保正确性，然后再去追求执行效率。

下面是在保守策略下，volatile 读插入内存屏障后生成的指令序列示意图：

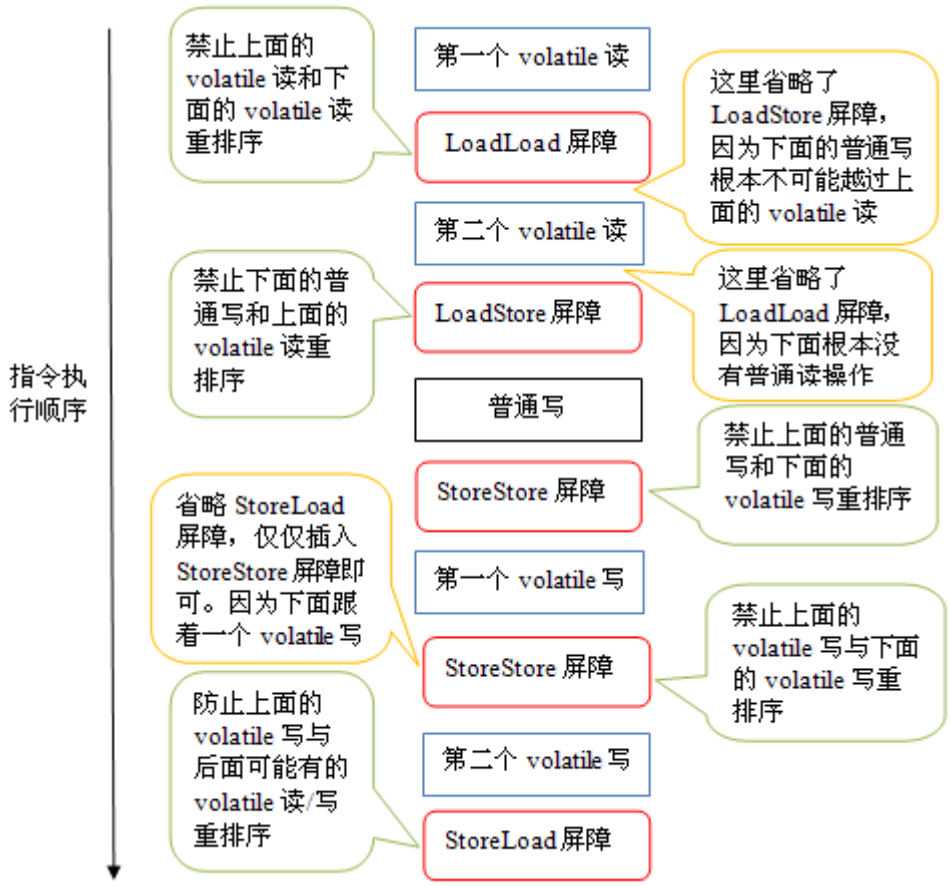


上图中的 LoadLoad 屏障用来禁止处理器把上面的 volatile 读与下面的普通读重排序。LoadStore 屏障用来禁止处理器把上面的 volatile 读与下面的普通写重排序。

上述 volatile 写和 volatile 读的内存屏障插入策略非常保守。在实际执行时，只要不改变volatile 写-读的内存语义，编译器可以根据具体情况省略不必要的屏障。下面我们通过具体的示例代码来说明：

```
class VolatileBarrierExample {
    int a;
    volatile int v1 = 1;
    volatile int v2 = 2;

    void readAndWrite() {
        int i = v1;         //第一个volatile读
        int j = v2;         // 第二个volatile读
        a = i + j;          //普通写
        v1 = i + 1;         // 第一个volatile写
        v2 = j * 2;         //第二个 volatile写
    }

    …                   //其他方法
}
```

针对 readAndWrite() 方法，编译器在生成字节码时可以做如下的优化：



注意，最后的 StoreLoad 屏障不能省略。因为第二个 volatile 写之后，方法立即 return。此时编译器可能无法准确断定后面是否会有 volatile 读或写，为了安全起见，编译器常常会在这里插入一个 StoreLoad 屏障。

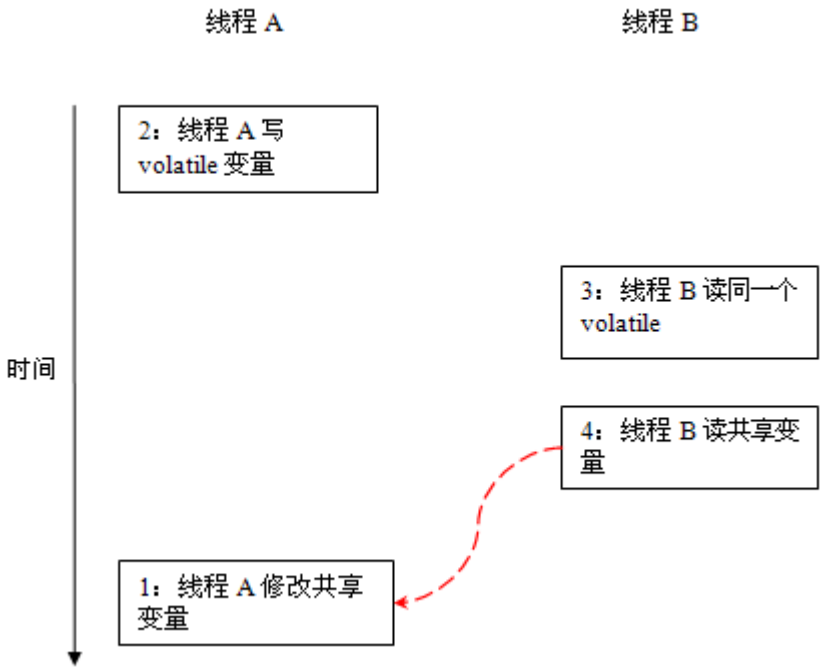
上面的优化是针对任意处理器平台，由于不同的处理器有不同“松紧度”的处理器内存模型，内存屏障的插入还可以根据具体的处理器内存模型继续优化。以 x86 处理器为例，上图中除最后的StoreLoad 屏障外，其它的屏障都会被省略。

前面保守策略下的 volatile 读和写，在 x86 处理器平台可以优化成：

前文提到过，x86 处理器仅会对写-读操作做重排序。X86 不会对读-读，读-写和写-写操作做重排序，因此在 x86 处理器中会省略掉这三种操作类型对应的内存屏障。在 x86 中，JMM 仅需在volatile 写后面插入一个 Store Load 屏障即可正确实现 volatile 写-读的内存语义。这意味着在 x86 处理器中，volatile 写的开销比 volatile 读的开销会大很多（因为执行StoreLoad 屏障开销会比较大）。

JSR-133 为什么要增强 volatile 的内存语义

在 JSR-133 之前的旧 Java 内存模型中，虽然不允许 volatile 变量之间重排序，但旧的Java 内存模型允许 volatile 变量与普通变量之间重排序。在旧的内存模型中，VolatileExample 示例程序可能被重排序成下列时序来执行：



在旧的内存模型中，当1和2之间没有数据依赖关系时，1和2之间就可能被重排序（3和4类似）。其结果就是：读线程B执行4时，不一定能看到写线程 A 在执行1时对共享变量的修改。

因此在旧的内存模型中，volatile 的写-读没有锁的释放-获所具有的内存语义。为了提供一种比锁更轻量级的线程之间通信的机制，JSR-133 专家组决定增强 volatile 的内存语义：严格限制编译器和处理器对 volatile 变量与普通变量的重排序，确保 volatile 的写-读和锁的释放-获取一样，具有相同的内存语义。从编译器重排序规则和处理器内存屏障插入策略来看，只要volatile 变量与普通变量之间的重排序可能会破坏 volatile 的内存语意，这种重排序就会被编译器重排序规则和处理器内存屏障插入策略禁止。

由于 volatile 仅仅保证对单个 volatile 变量的读/写具有原子性，而锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性。在功能上，锁比 volatile 更强大；在可伸缩性和执行性能上，volatile 更有优势。如果读者想在程序中用 volatile 代替监视器锁，请一定谨慎，具体细节请参阅参考文献5。

5

锁

锁的释放-获取建立的 happens before 关系

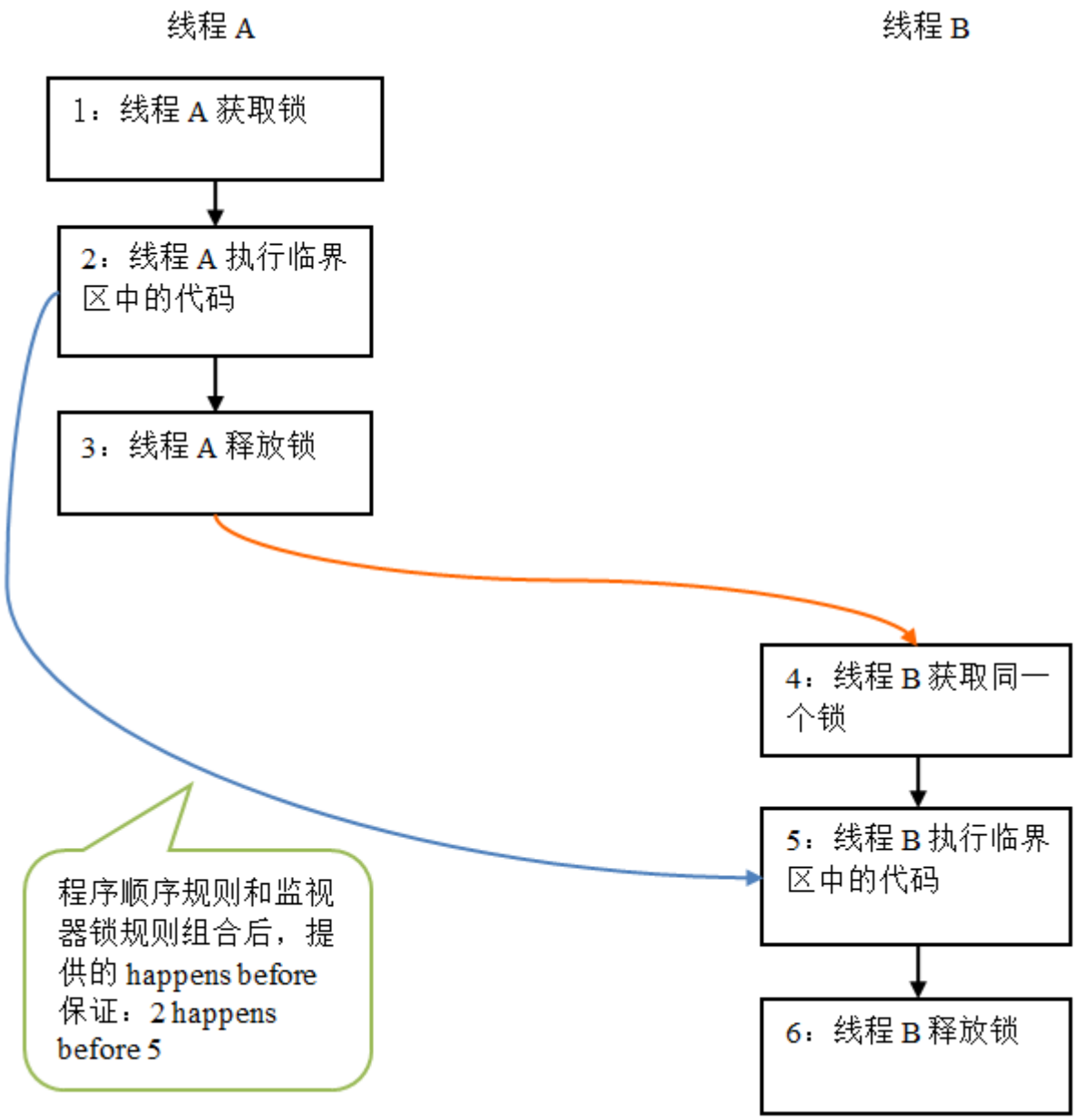
锁是 java 并发编程中最重要的同步机制。锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。下面是锁释放-获取的示例代码：

```
class MonitorExample {
   int a = 0;

   public synchronized void writer() {  //1
      a++;                              //2
   }                                    //3

   public synchronized void reader() {  //4
      int i = a;                        //5
      ……
   }                                    //6
}
```

假设线程 A 执行 writer() 方法，随后线程 B 执行 reader() 方法。根据 happens before 规则，这个过程包含的 happens before 关系可以分为两类：

1. 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
2. 根据监视器锁规则，3 happens before 4。
3. 根据 happens before 的传递性，2 happens before 5。

上述 happens before 关系的图形化表现形式如下：

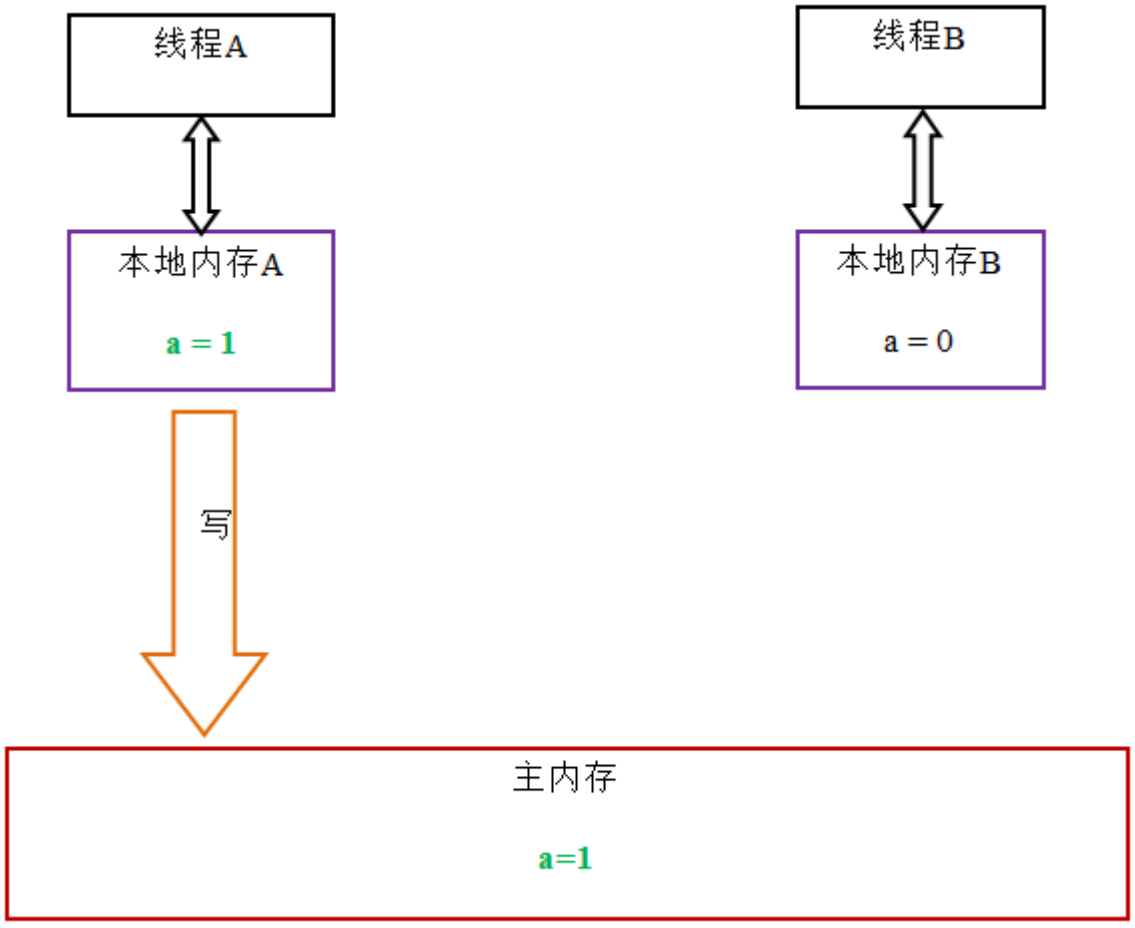


在上图中，每一个箭头链接的两个节点，代表了一个 happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的 happens before保证。

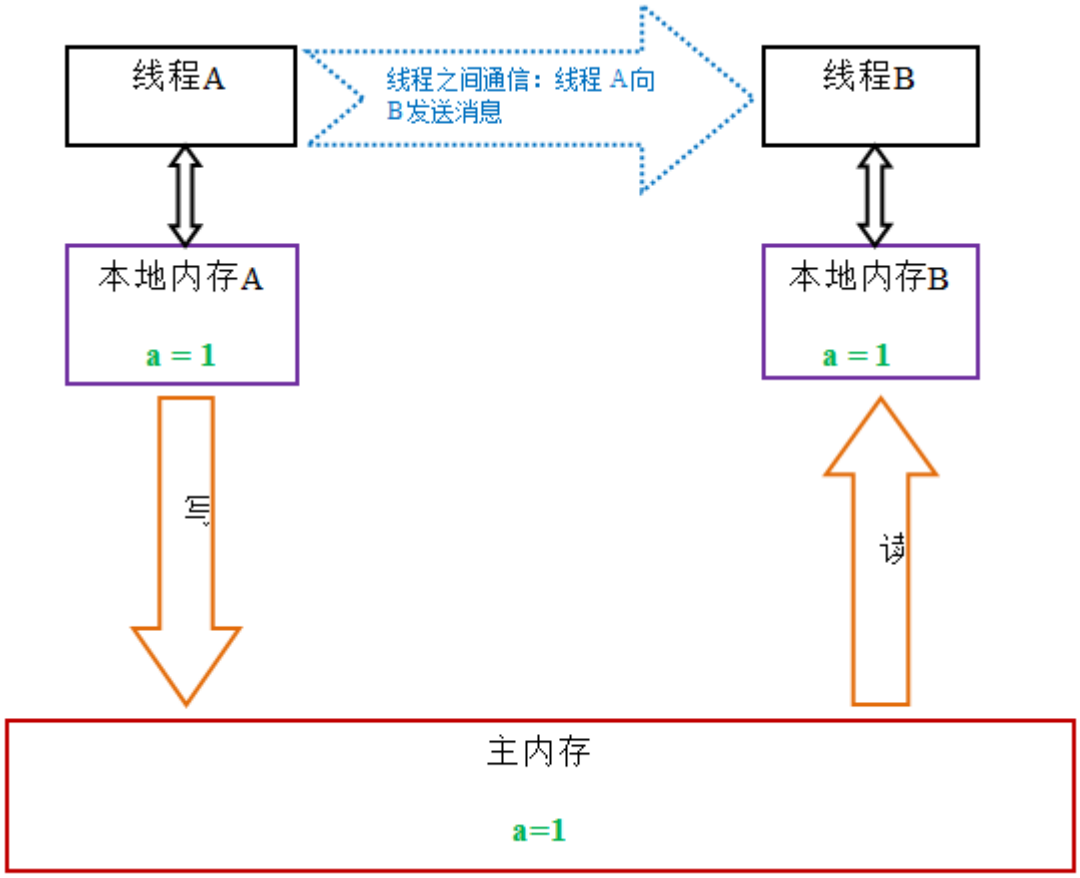
上图表示在线程A释放了锁之后，随后线程B获取同一个锁。在上图中，2 happens before 5。因此，线程A在释放锁之前所有可见的共享变量，在线程B获取同一个锁之后，将立刻变得对B线程可见。

锁释放和获取的内存语义

当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。以上面的MonitorExample程序为例，A线程释放锁后，共享数据的状态示意图如下：



当线程获取锁时，JMM 会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须要从主内存中去读取共享变量。下面是锁获取的状态示意图：



对比锁释放-获取的内存语义与 volatile 写-读的内存语义，可以看出：锁释放与 volatile 写有相同的内存语义；锁获取与 volatile 读有相同的内存语义。

下面对锁释放和锁获取的内存语义做个总结：

- 线程 A 释放一个锁，实质上是线程 A 向接下来将要获取这个锁的某个线程发出了（线程 A 对共享变量所做修改的）消息。
- 线程 B 获取一个锁，实质上是线程 B 接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
- 线程 A 释放锁，随后线程 B 获取这个锁，这个过程实质上是线程 A 通过主内存向线程 B 发送消息。

锁内存语义的实现

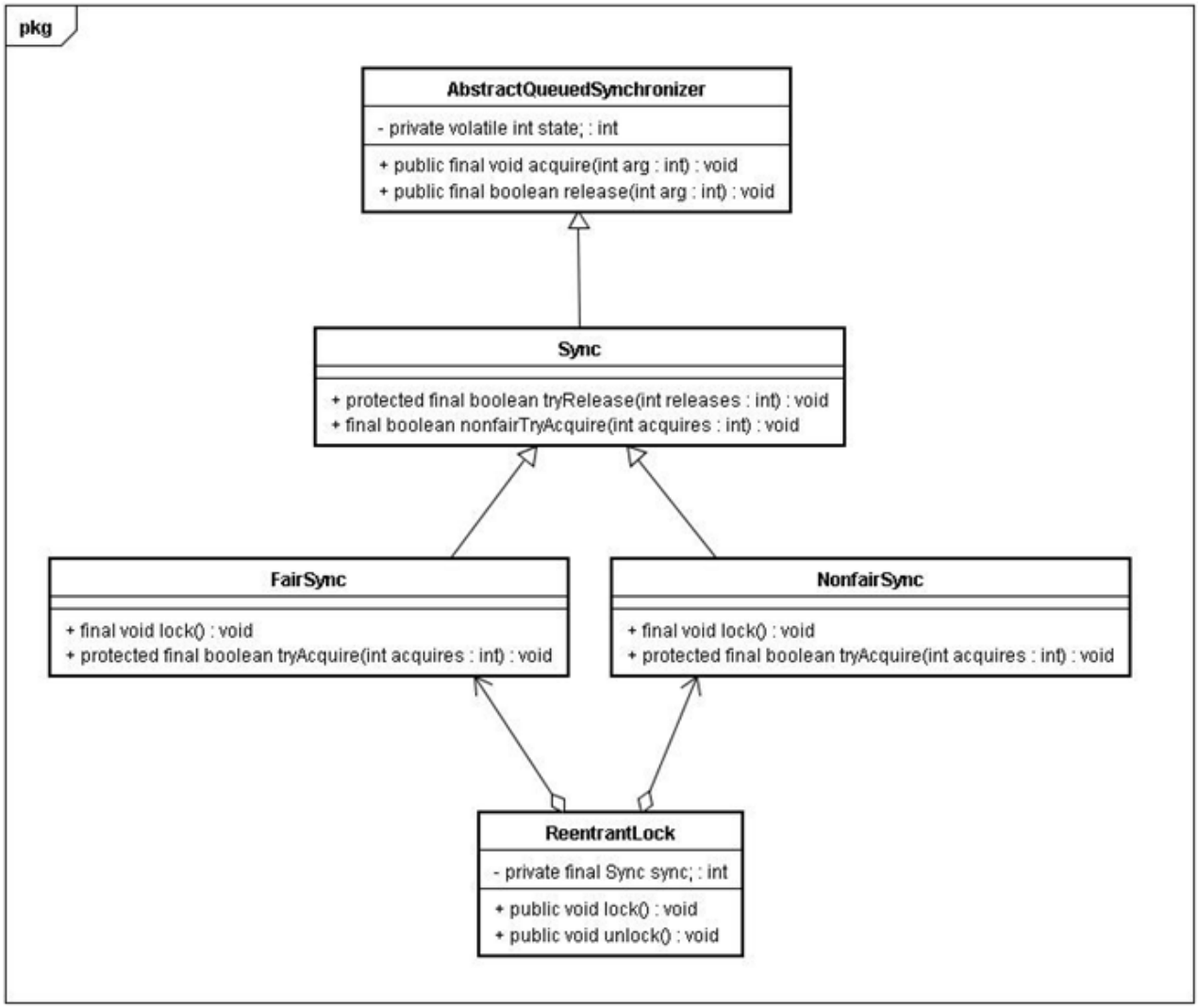
本文将借助 ReentrantLock 的源代码，来分析锁内存语义的具体实现机制。

请看下面的示例代码：

```
class ReentrantLockExample {
int a = 0;
ReentrantLock lock = new ReentrantLock();

public void writer() {
    lock.lock();        //获取锁
    try {
        a++;
    } finally {
        lock.unlock();  //释放锁
    }
}

public void reader () {
    lock.lock();        //获取锁
    try {
        int i = a;
        ……
    } finally {
        lock.unlock();  //释放锁
    }
}
}
```

在 ReentrantLock 中，调用 lock() 方法获取锁；调用 unlock() 方法释放锁。

ReentrantLock 的实现依赖于 java 同步器框架 AbstractQueuedSynchronizer（本文简称之为AQS）。AQS 使用一个整型的 volatile 变量（命名为 state）来维护同步状态，马上我们会看到，这个 volatile 变量是 ReentrantLock 内存语义实现的关键。 下面是ReentrantLock 的类图（仅画出与本文相关的部分）：



ReentrantLock 分为公平锁和非公平锁，我们首先分析公平锁。

使用公平锁时，加锁方法 lock() 的方法调用轨迹如下：

1. ReentrantLock : lock()
2. FairSync : lock()
3. AbstractQueuedSynchronizer : acquire(int arg)
4. ReentrantLock : tryAcquire(int acquires)

在第4步真正开始加锁，下面是该方法的源代码：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();  //获取锁的开始，首先读volatile变量state
    if (c == 0) {
        if (isFirst(current) &&
```

```
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

从上面源代码中我们可以看出，加锁方法首先读 volatile 变量 state。

在使用公平锁时，解锁方法 unlock() 的方法调用轨迹如下：

1. ReentrantLock : unlock()
2. AbstractQueuedSynchronizer : release(int arg)
3. Sync : tryRelease(int releases)

在第3步真正开始释放锁，下面是该方法的源代码：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);          //释放锁的最后，写volatile变量state
    return free;
}
```

从上面的源代码我们可以看出，在释放锁的最后写 volatile 变量 state。

公平锁在释放锁的最后写 volatile 变量 state；在获取锁时首先读这个 volatile 变量。根据 volatile 的 happens-before 规则，释放锁的线程在写 volatile 变量之前可见的共享变量，在获取锁的线程读取同一个 volatile 变量后将立即变的对获取锁的线程可见。

现在我们分析非公平锁的内存语义的实现。

非公平锁的释放和公平锁完全一样，所以这里仅仅分析非公平锁的获取。

使用非公平锁时，加锁方法 lock() 的方法调用轨迹如下：

1. ReentrantLock : lock()
2. NonfairSync : lock()
3. AbstractQueuedSynchronizer : compareAndSetState(int expect, int update)

在第3步真正开始加锁，下面是该方法的源代码：

```
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

该方法以原子操作的方式更新 state 变量，本文把 java 的 compareAndSet() 方法调用简称为 CAS。JDK 文档对该方法的说明如下：如果当前状态值等于预期值，则以原子方式将同步状态设置为给定的更新值。此操作具有 volatile 读和写的内存语义。

这里我们分别从编译器和处理器的角度来分析,CAS 如何同时具有 volatile 读和 volatile 写的内存语义。

前文我们提到过，编译器不会对 volatile 读与 volatile 读后面的任意内存操作重排序；编译器不会对 volatile 写与 volatile 写前面的任意内存操作重排序。组合这两个条件，意味着为了同时实现 volatile 读和 volatile 写的内存语义，编译器不能对 CAS 与 CAS 前面和后面的任意内存操作重排序。

下面我们来分析在常见的 intel x86 处理器中，CAS 是如何同时具有 volatile 读和 volatile 写的内存语义的。

下面是 sun.misc.Unsafe 类的 compareAndSwapInt() 方法的源代码：

```
public final native boolean compareAndSwapInt(Object o, long offset,
                                               int expected,
                                               int x);
```

可以看到这是个本地方法调用。这个本地方法在 openjdk 中依次调用的 C++ 代码为：unsafe.cpp，atomic.cpp 和 atomicwindowsx86.inline.hpp。这个本地方法的最终实现在 openjdk 的如下位置： `openjdk-7-fcs-src-b147-27jun2011\openjdk\hotspot\src\oscpu\windowsx86\vm\ atomicwindowsx86.inline.hpp` （对应于 windows 操作系统，X86 处理器）。下面是对应于 intel x86 处理器的源代码的片段：

```
// Adding a lock prefix to an instruction on MP machine
// VC++ doesn't like the lock prefix to be on a single line
// so we can't insert a label after the lock prefix.
// By emitting a lock prefix, we can define a label after it.
```

```
#define LOCK_IF_MP(mp) __asm cmp mp, 0  \

                    __asm je L0      \
                    __asm _emit 0xF0 \
                    __asm L0:

inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_value) {
  // alternative for InterlockedCompareExchange
  int mp = os::is_MP();
  __asm {
    mov edx, dest
    mov ecx, exchange_value
    mov eax, compare_value
    LOCK_IF_MP(mp)
    cmpxchg dword ptr [edx], ecx
  }
}
```

如上面源代码所示，程序会根据当前处理器的类型来决定是否为 cmpxchg 指令添加 lock 前缀。如果程序是在多处理器上运行，就为 cmpxchg 指令加上 lock 前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略 lock 前缀（单处理器自身会维护单处理器内的顺序一致性，不需要 lock 前缀提供的内存屏障效果）。

intel 的手册对 lock 前缀的说明如下：

1. 确保对内存的读-改-写操作原子执行。在 Pentium 及 Pentium 之前的处理器中，带有lock 前缀的指令在执行期间会锁住总线，使得其他处理器暂时无法通过总线访问内存。很显然，这会带来昂贵的开销。从 Pentium 4，Intel Xeon 及 P6 处理器开始，intel 在原有总线锁的基础上做了一个很有意义的优化：如果要访问的内存区域（area of memory）在 lock 前缀指令执行期间已经在处理器内部的缓存中被锁定（即包含该内存区域的缓存行当前处于独占或以修改状态），并且该内存区域被完全包含在单个缓存行（cache line）中，那么处理器将直接执行该指令。由于在指令执行期间该缓存行会一直被锁定，其它处理器无法读/写该指令要访问的内存区域，因此能保证指令执行的原子性。这个操作过程叫做缓存锁定（cache locking），缓存锁定将大大降低 lock 前缀指令的执行开销，但是当多处理器之间的竞争程度很高或者指令访问的内存地址未对齐时，仍然会锁住总线。
2. 禁止该指令与之前和之后的读和写指令重排序。
3. 把写缓冲区中的所有数据刷新到内存中。

上面的第2点和第3点所具有的内存屏障效果，足以同时实现 volatile 读和 volatile 写的内存语义。

经过上面的这些分析，现在我们终于能明白为什么 JDK 文档说 CAS 同时具有 volatile 读和volatile 写的内存语义了。

现在对公平锁和非公平锁的内存语义做个总结：

- 公平锁和非公平锁释放时，最后都要写一个 volatile 变量 state。
- 公平锁获取时，首先会去读这个 volatile 变量。
- 非公平锁获取时，首先会用 CAS 更新这个 volatile 变量,这个操作同时具有 volatile 读和 volatile 写的内存语义。

从本文对 ReentrantLock 的分析可以看出，锁释放-获取的内存语义的实现至少有下面两种方式：

1. 利用 volatile 变量的写-读所具有的内存语义。
2. 利用 CAS 所附带的 volatile 读和 volatile 写的内存语义。

concurrent 包的实现

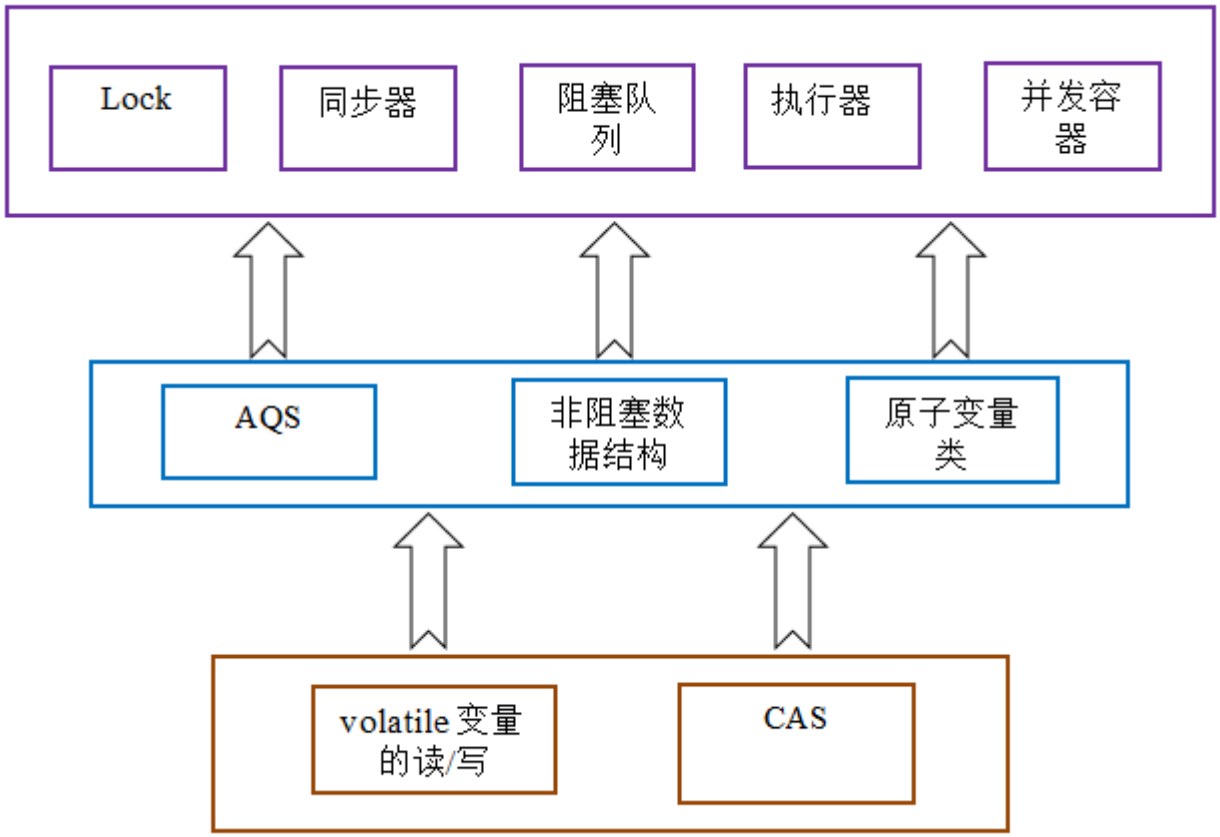
由于 java 的 CAS 同时具有 volatile 读和 volatile 写的内存语义，因此 Java 线程之间的通信现在有了下面四种方式：

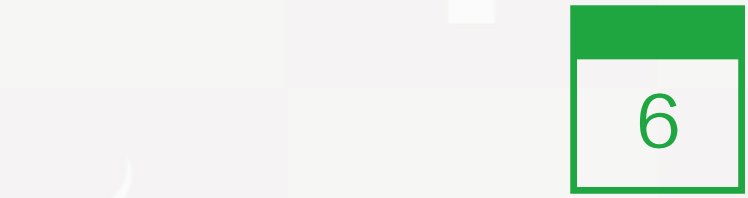
1. A 线程写 volatile 变量，随后 B 线程读这个 volatile 变量。
2. A 线程写 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量。
3. A 线程用 CAS 更新一个volatile变量，随后 B 线程用 CAS 更新这个 volatile 变量。
4. A 线程用 CAS 更新一个 volatile 变量，随后 B 线程读这个 volatile 变量。

Java 的 CAS 会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，volatile 变量的读/写和 CAS 可以实现线程之间的通信。把这些特性整合在一起，就形成了整个 concurrent 包得以实现的基石。如果我们仔细分析 concurrent 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，配合以 volatile 的读/写和 CAS 所具有的 volatile 读和写的内存语义来实现线程之间的通信。

AQS，非阻塞数据结构和原子变量类（java.util.concurrent.atomic 包中的类），这些 concurrent 包中的基础类都是使用这种模式来实现的，而 concurrent 包中的高层类又是依赖于这些基础类来实现的。从整体来看，concurrent 包的实现示意图如下：


Lock
同步器
阻塞队列
执行器
并发容器
AQS
非阻塞数据结构
原子变量类
volatile 变量的读/写
CAS



final

与前面介绍的锁和 volatile 相比较，对 final 域的读和写更像是普通的变量访问。对于final 域，编译器和处理器要遵守两个重排序规则：

1. 在构造函数内对一个 final 域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。
2. 初次读一个包含 final 域的对象的引用，与随后初次读这个 final 域，这两个操作之间不能重排序。

下面，我们通过一些示例性的代码来分别说明这两个规则：

```
public class FinalExample {
   int i;                          //普通变量
   final int j;                    //final变量
   static FinalExample obj;

   public void FinalExample () {     //构造函数
      i = 1;                       //写普通域
      j = 2;                       //写final域
   }

   public static void writer () {    //写线程A执行
      obj = new FinalExample ();
   }

   public static void reader () {       //读线程B执行
      FinalExample object = obj;       //读对象引用
      int a = object.i;                //读普通域
      int b = object.j;                //读final域
   }
}
```

这里假设一个线程 A 执行 writer() 方法，随后另一个线程 B 执行 reader() 方法。下面我们通过这两个线程的交互来说明这两个规则。

写 final 域的重排序规则

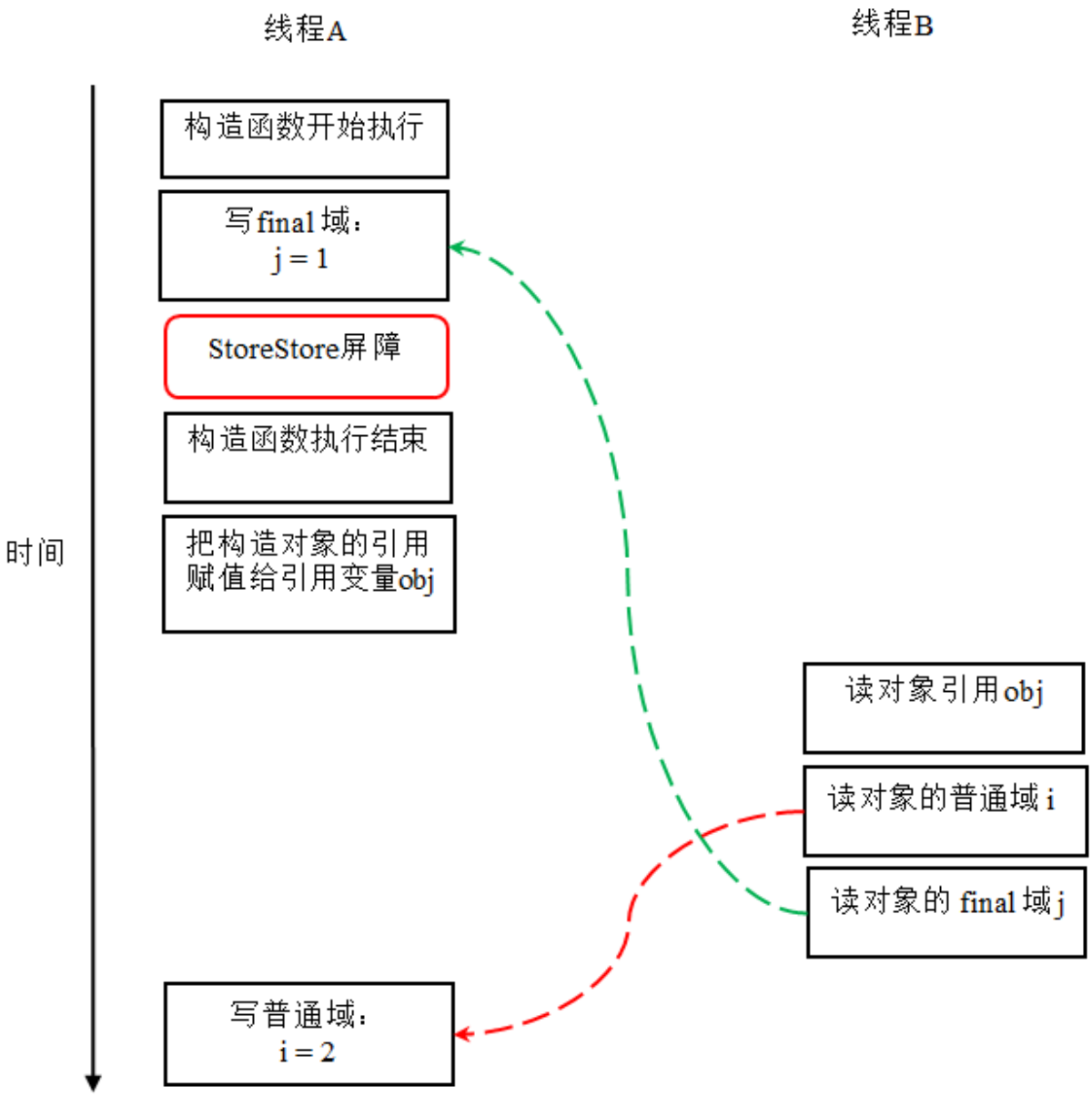
写 final 域的重排序规则禁止把 final 域的写重排序到构造函数之外。这个规则的实现包含下面2个方面：

- JMM 禁止编译器把 final 域的写重排序到构造函数之外。
- 编译器会在 final 域的写之后，构造函数 return 之前，插入一个 StoreStore 屏障。这个屏障禁止处理器把 final 域的写重排序到构造函数之外。

现在让我们分析 writer() 方法。writer() 方法只包含一行代码：finalExample = new FinalExample()。这行代码包含两个步骤：

1. 构造一个 FinalExample 类型的对象；
2. 把这个对象的引用赋值给引用变量 obj。

假设线程 B 读对象引用与读对象的成员域之间没有重排序（马上会说明为什么需要这个假设），下图是一种可能的执行时序：



在上图中，写普通域的操作被编译器重排序到了构造函数之外，读线程B错误的读取了普通变量i初始化之前的值。而写 final 域的操作，被写 final 域的重排序规则“限定”在了构造函数之内，读线程 B 正确的读取了 final 变量初始化之后的值。

写 final 域的重排序规则可以确保：在对象引用为任意线程可见之前，对象的 final 域已经被正确初始化过了，而普通域不具有这个保障。以上图为例，在读线程 B “看到”对象引用 obj 时，很可能 obj 对象还没有构造完成（对普通域i的写操作被重排序到构造函数外，此时初始值2还没有写入普通域i）。

读 final 域的重排序规则

读 final 域的重排序规则如下：

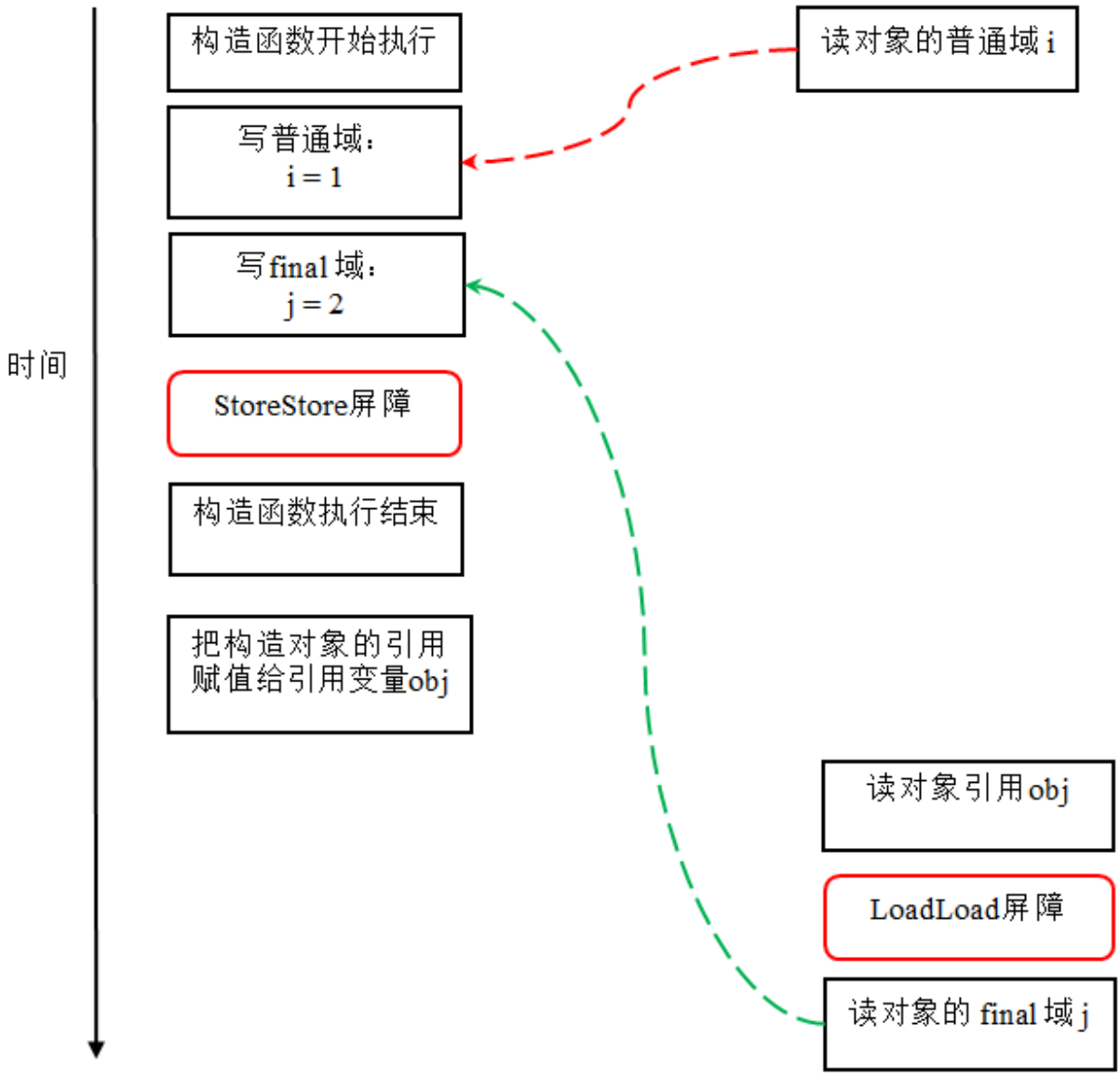
- 在一个线程中，初次读对象引用与初次读该对象包含的 final 域，JMM 禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读 final 域操作的前面插入一个 LoadLoad 屏障。

初次读对象引用与初次读该对象包含的 final 域，这两个操作之间存在间接依赖关系。由于编译器遵守间接依赖关系，因此编译器不会重排序这两个操作。大多数处理器也会遵守间接依赖，大多数处理器也不会重排序这两个操作。但有少数处理器允许对存在间接依赖关系的操作做重排序（比如 alpha 处理器），这个规则就是专门用来针对这种处理器。

reader() 方法包含三个操作：

1. 初次读引用变量 obj;
2. 初次读引用变量 obj 指向对象的普通域 j。
3. 初次读引用变量 obj 指向对象的 final 域 i。

现在我们假设写线程 A 没有发生任何重排序，同时程序在不遵守间接依赖的处理器上执行，下面是一种可能的执行时序：



在上图中，读对象的普通域的操作被处理器重排序到读对象引用之前。读普通域时，该域还没有被写线程A写入，这是一个错误的读取操作。而读 final 域的重排序规则会把读对象 final 域的操作“限定”在读对象引用之后，此时该 final 域已经被 A 线程初始化过了，这是一个正确的读取操作。

读 final 域的重排序规则可以确保：在读一个对象的 final 域之前，一定会先读包含这个 final 域的对象的引用。在这个示例程序中，如果该引用不为 null，那么引用对象的 final 域一定已经被 A 线程初始化过了。

如果 final 域是引用类型

上面我们看到的 final 域是基础数据类型，下面让我们看看如果 final 域是引用类型，将会有什么效果?

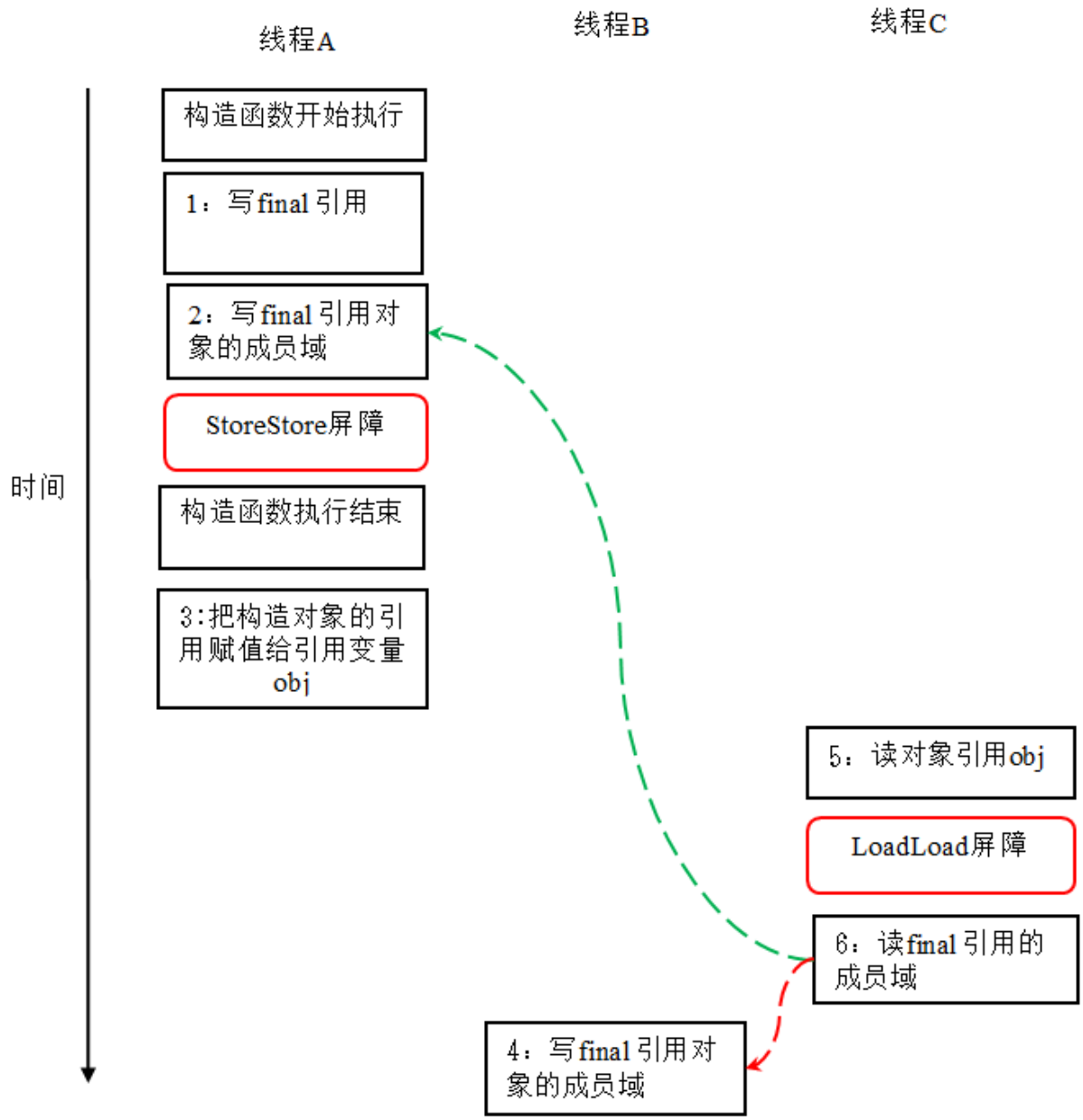
请看下列示例代码：

```
public class FinalReferenceExample {
final int[] intArray;                    //final是引用类型
static FinalReferenceExample obj;

public FinalReferenceExample () {        //构造函数
   intArray = new int[1];                //1
   intArray[0] = 1;                      //2
}

public static void writerOne () {        //写线程A执行
   obj = new FinalReferenceExample ();  //3
}

public static void writerTwo () {        //写线程B执行
   obj.intArray[0] = 2;                  //4
}

public static void reader () {           //读线程C执行
   if (obj != null) {                    //5
      int temp1 = obj.intArray[0];       //6
   }
}
}
```

这里 final 域为一个引用类型，它引用一个 int 型的数组对象。对于引用类型，写 final 域的重排序规则对编译器和处理器增加了如下约束：

1. 在构造函数内对一个 final 引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

对上面的示例程序，我们假设首先线程 A 执行 writerOne() 方法，执行完后线程 B 执行 writerTwo() 方法，执行完后线程 C 执行 reader() 方法。下面是一种可能的线程执行时序：



在上图中，1 是对 final 域的写入，2 是对这个 final 域引用的对象的成员域的写入，3是把被构造的对象的引用赋值给某个引用变量。这里除了前面提到的1不能和3重排序外，2和3也不能重排序。

JMM 可以确保读线程 C 至少能看到写线程 A 在构造函数中对 final 引用对象的成员域的写入。即 C 至少能看到数组下标 0 的值为 1。而写线程 B 对数组元素的写入，读线程 C 可能看的到，也可能看不到。JMM 不保证线程 B 的写入对读线程 C 可见，因为写线程 B 和读线程 C 之间存在数据竞争，此时的执行结果不可预知。

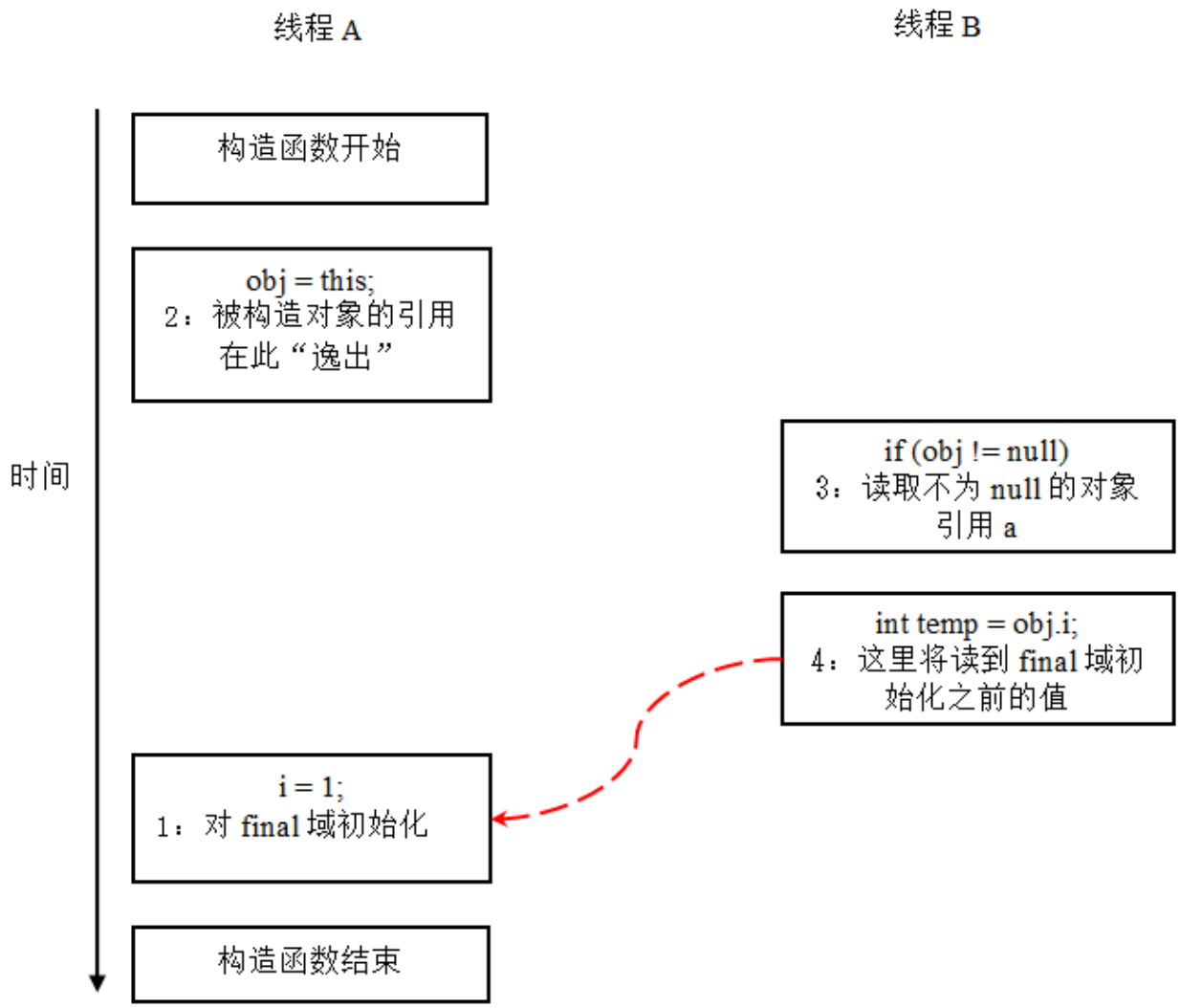
如果想要确保读线程 C 看到写线程 B 对数组元素的写入，写线程 B 和读线程 C 之间需要使用同步原语（lock 或 volatile）来确保内存可见性。

为什么 final 引用不能从构造函数内“逸出”

前面我们提到过，写 final 域的重排序规则可以确保：在引用变量为任意线程可见之前，该引用变量指向的对象的 final 域已经在构造函数中被正确初始化过了。其实要得到这个效果，还需要一个保证：在构造函数内部，不能让这个被构造对象的引用为其他线程可见，也就是对象引用不能在构造函数中“逸出”。为了说明问题，让我们来看下面示例代码：

```
public class FinalReferenceEscapeExample {
final int i;
static FinalReferenceEscapeExample obj;

public FinalReferenceEscapeExample () {
    i = 1;                          //1写final域
    obj = this;                       //2 this引用在此“逸出”
}

public static void writer() {
    new FinalReferenceEscapeExample ();
}

public static void reader {
    if (obj != null) {                  //3
        int temp = obj.i;               //4
    }
}
}
```

假设一个线程 A 执行 writer() 方法，另一个线程 B 执行 reader() 方法。这里的操作2使得对象还未完成构造前就为线程 B 可见。即使这里的操作 2 是构造函数的最后一步，且即使在程序中操作 2 排在操作 1 后面，执行 read() 方法的线程仍然可能无法看到 final 域被初始化后的值，因为这里的操作 1 和操作 2 之间可能被重排序。实际的执行时序可能如下图所示：



从上图我们可以看出：在构造函数返回前，被构造对象的引用不能为其他线程可见，因为此时的 final 域可能还没有被初始化。在构造函数返回后，任意线程都将保证能看到 final 域正确初始化之后的值。

final 语义在处理器中的实现

现在我们以 x86 处理器为例，说明 final 语义在处理器中的具体实现。

上面我们提到，写 final 域的重排序规则会要求译编器在 final 域的写之后，构造函数return 之前，插入一个 StoreStore 障屏。读 final 域的重排序规则要求编译器在读 final 域的操作前面插入一个 LoadLoad 屏障。

由于 x86 处理器不会对写-写操作做重排序，所以在 x86 处理器中，写 final 域需要的 StoreStore 障屏会被省略掉。同样，由于 x86 处理器不会对存在间接依赖关系的操作做重排序，所以在 x86 处理器中，读 final 域需要的 LoadLoad 屏障也会被省略掉。也就是说在 x86 处理器中，final 域的读/写不会插入任何内存屏障！

JSR-133 为什么要增强 final 的语义

在旧的 Java 内存模型中，最严重的一个缺陷就是线程可能看到 final 域的值会改变。比如，一个线程当前看到一个整形 final 域的值为 0（还未初始化之前的默认值），过一段时间之后这个线程再去读这个 final 域的值时，却发现值变为了 1（被某个线程初始化之后的值）。最常见的例子就是在旧的 Java 内存模型中，String 的值可能会改变（参考文献 2 中有一个具体的例子，感兴趣的读者可以自行参考，这里就不赘述了）。

为了修补这个漏洞，JSR-133 专家组增强了 final 的语义。通过为 final 域增加写和读重排序规则，可以为 java 程序员提供初始化安全保证：只要对象是正确构造的（被构造对象的引用在构造函数中没有“逸出”），那么不需要使用同步（指 lock 和 volatile 的使用），就可以保证任意线程都能看到这个 final 域在构造函数中被初始化之后的值。

7

总结

处理器内存模型

顺序一致性内存模型是一个理论参考模型，JMM 和处理器内存模型在设计时通常会把顺序一致性内存模型作为参照。JMM 和处理器内存模型在设计时会对顺序一致性模型做一些放松，因为如果完全按照顺序一致性模型来实现处理器和 JMM，那么很多的处理器和编译器优化都要被禁止，这对执行性能将会有很大的影响。

根据对不同类型读/写操作组合的执行顺序的放松，可以把常见处理器的内存模型划分为下面几种类型：

1. 放松程序中写-读操作的顺序，由此产生了 total store ordering 内存模型（简称为TSO）。
2. 在前面1的基础上，继续放松程序中写-写操作的顺序，由此产生了 partial store order 内存模型（简称为 PSO）。
3. 在前面1和2的基础上，继续放松程序中读-写和读-读操作的顺序，由此产生了 relaxed memory order 内存模型（简称为 RMO）和 PowerPC 内存模型。

注意，这里处理器对读/写操作的放松，是以两个操作之间不存在数据依赖性为前提的（因为处理器要遵守 as-if-serial 语义，处理器不会对存在数据依赖性的两个内存操作做重排序）。

下面的表格展示了常见处理器内存模型的细节特征：

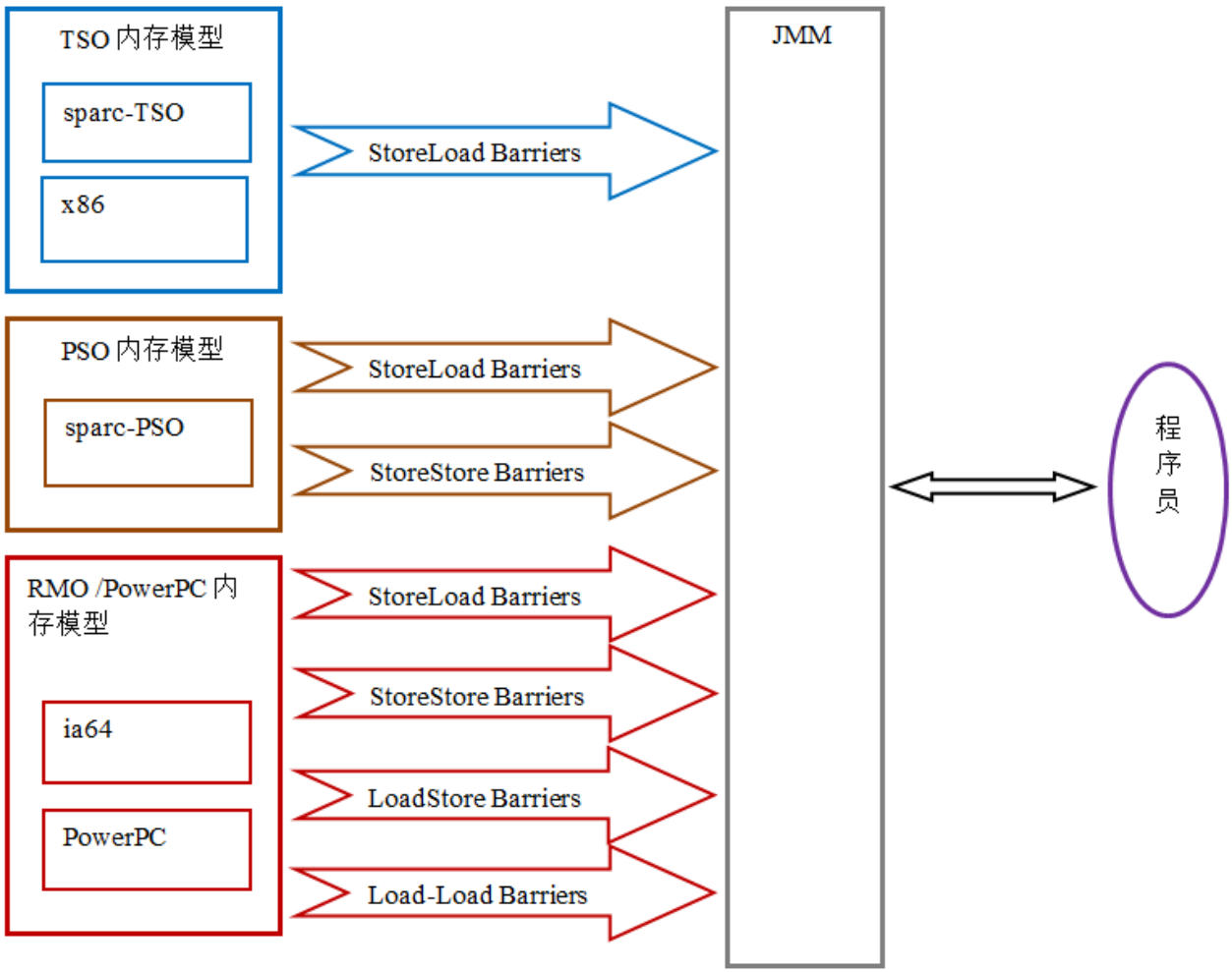
内存模型名称	对应的处理器	Store-Load 重排序	Store-Store 重排序	Load-Load 和Load-Store重排序	可以更早读取到其它处理器的写	可以更早读取到当前处理器的写
TSO	sparc-TSO X64	Y				Y
PSO	sparc-PSO	Y	Y			Y
RMO	ia64	Y	Y	Y		Y
PowerPC	PowerPC	Y	Y	Y	Y	Y

在这个表格中，我们可以看到所有处理器内存模型都允许写-读重排序，原因在第一章以说明过：它们都使用了写缓存区，写缓存区可能导致写-读操作重排序。同时，我们可以看到这些处理器内存模型都允许更早读到当前处理器的写，原因同样是因为写缓存区：由于写缓存区仅对当前处理器可见，这个特性导致当前处理器可以比其他处理器先看到临时保存在自己的写缓存区中的写。

上面表格中的各种处理器内存模型，从上到下，模型由强变弱。越是追求性能的处理器，内存模型设计的会越弱。因为这些处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。

由于常见的处理器内存模型比 JMM 要弱，java 编译器在生成字节码时，会在执行指令序列的适当位置插入内存屏障来限制处理器的重排序。同时，由于各种处理器内存模型的强弱并不相同，为了在不同的处理器平台向程序

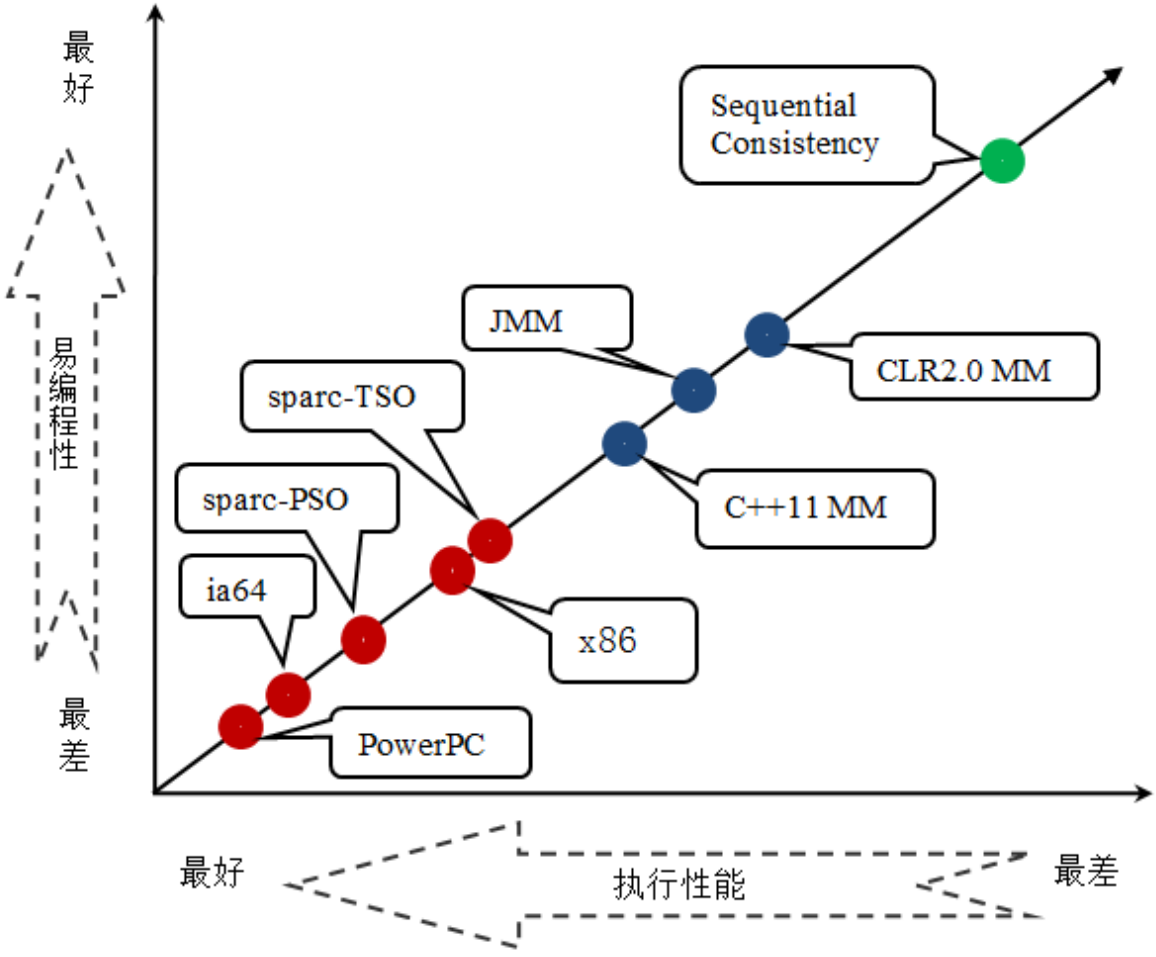
员展示一个一致的内存模型，JMM 在不同的处理器中需要插入的内存屏障的数量和种类也不相同。下图展示了 JMM 在不同处理器内存模型中需要插入的内存屏障的示意图：



如上图所示，JMM 屏蔽了不同处理器内存模型的差异，它在不同的处理器平台之上为 java 程序员呈现了一个一致的内存模型。

JMM，处理器内存模型与顺序一致性内存模型之间的关系

JMM 是一个语言级的内存模型，处理器内存模型是硬件级的内存模型，顺序一致性内存模型是一个理论参考模型。下面是语言内存模型，处理器内存模型和顺序一致性内存模型的强弱对比示意图：



从上图我们可以看出：常见的4种处理器内存模型比常用的3中语言内存模型要弱，处理器内存模型和语言内存模型都比顺序一致性内存模型要弱。同处理器内存模型一样，越是追求执行性能的语言，内存模型设计的会越弱。

JMM 的设计

从 JMM 设计者的角度来说，在设计 JMM 时，需要考虑两个关键因素：

- 程序员对内存模型的使用。程序员希望内存模型易于理解，易于编程。程序员希望基于一个强内存模型来编写代码。
- 编译器和处理器对内存模型的实现。编译器和处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。编译器和处理器希望实现一个弱内存模型。

由于这两个因素互相矛盾，所以 JSR-133 专家组在设计 JMM 时的核心目标就是找到一个好的平衡点：一方面要为程序员提供足够强的内存可见性保证；另一方面，对编译器和处理器的限制要尽可能的放松。下面让我们看看 JSR-133 是如何实现这一目标的。

为了具体说明，请看前面提到过的计算圆面积的示例代码：

```
double pi  = 3.14;    //A
double r   = 1.0;     //B
double area = pi * r * r; //C
```

上面计算圆的面积的示例代码存在三个 happens- before 关系：

1. A happens- before B；
2. B happens- before C；
3. A happens- before C；

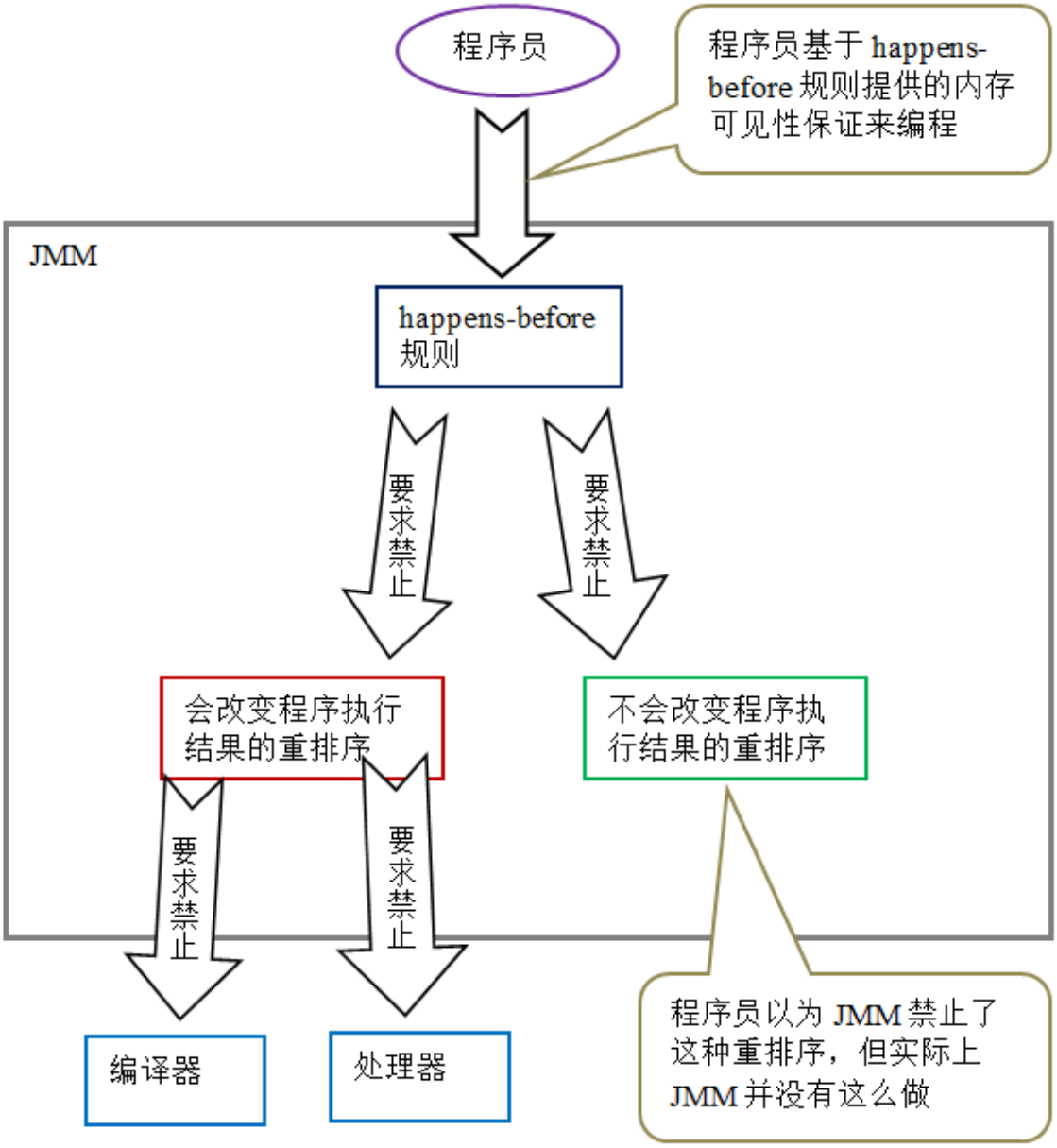
由于 A happens- before B，happens- before 的定义会要求：A 操作执行的结果要对B可见，且 A 操作的执行顺序排在B操作之前。 但是从程序语义的角度来说，对A和B做重排序即不会改变程序的执行结果，也还能提高程序的执行性能（允许这种重排序减少了对编译器和处理器优化的束缚）。也就是说，上面这3个 happens- before 关系中，虽然2和3是必需要的，但1是不必要的。因此，JMM 把 happens- before 要求禁止的重排序分为了下面两类：

- 会改变程序执行结果的重排序。
- 不会改变程序执行结果的重排序。

JMM 对这两种不同性质的重排序，采取了不同的策略：

- 对于会改变程序执行结果的重排序，JMM 要求编译器和处理器必须禁止这种重排序。
- 对于不会改变程序执行结果的重排序，JMM 对编译器和处理器不作要求（JMM 允许这种重排序）。

下面是JMM的设计示意图：



从上图可以看出两点：

- JMM 向程序员提供的 happens- before 规则能满足程序员的需求。JMM 的 happens- before 规则不但简单易懂，而且也向程序员提供了足够强的内存可见性保证（有些内存可见性保证其实并不一定真实存在，比如上面的 A happens- before B）。
- JMM 对编译器和处理器的束缚已经尽可能的少。从上面的分析我们可以看出，JMM 其实是在遵循一个基本原则：只要不改变程序的执行结果（指的是单线程程序和正确同步的多线程程序），编译器和处理器怎么优化都行。比如，如果编译器经过细致的分析后，认定一个锁只会被单个线程访问，那么这个锁可以被消

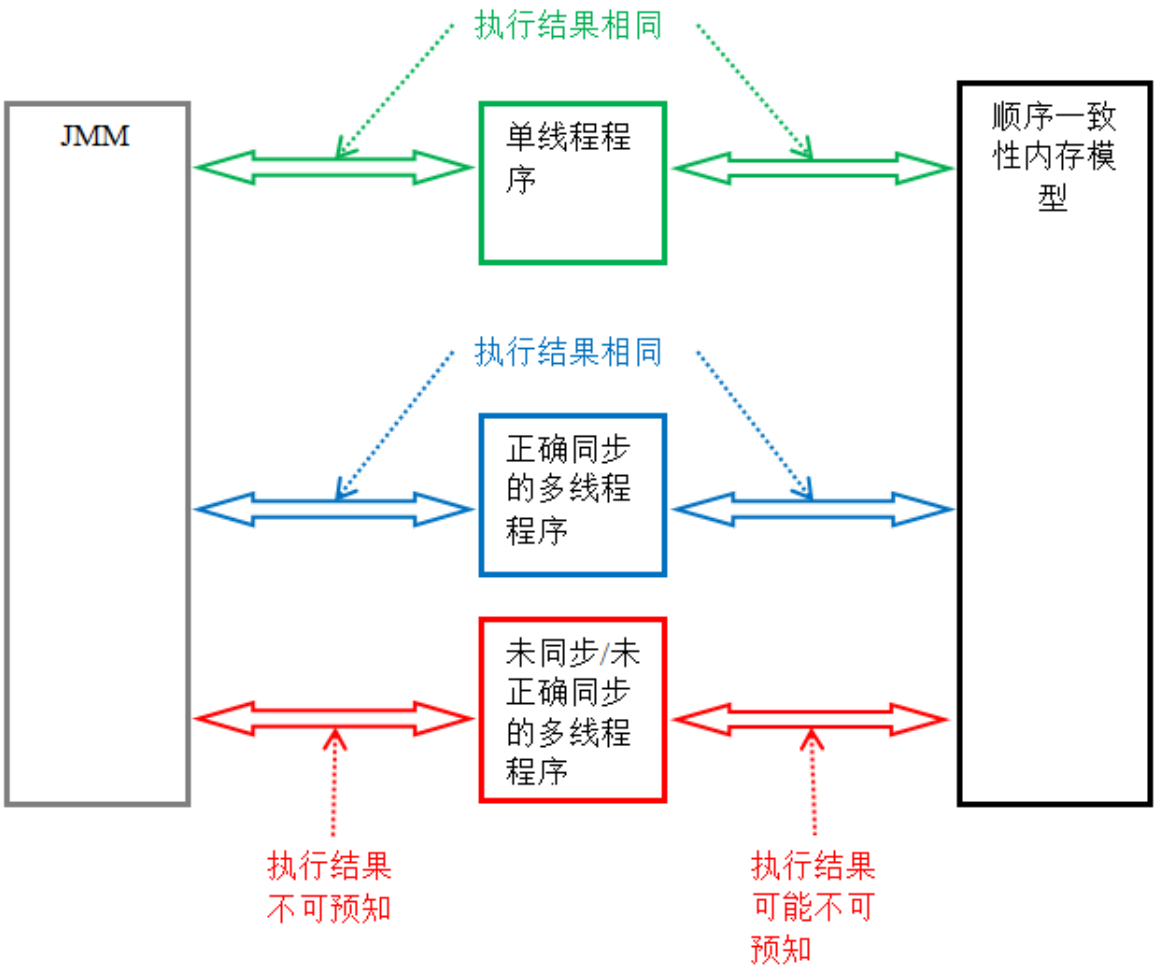
除。再比如，如果编译器经过细致的分析后，认定一个 volatile 变量仅仅只会被单个线程访问，那么编译器可以把这个 volatile 变量当作一个普通变量来对待。这些优化既不会改变程序的执行结果，又能提高程序的执行效率。

JMM 的内存可见性保证

Java 程序的内存可见性保证按程序类型可以分为下列三类：

1. 单线程程序。单线程程序不会出现内存可见性问题。编译器，runtime 和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。
2. 正确同步的多线程程序。正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）。这是 JMM 关注的重点，JMM 通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。
3. 未同步/未正确同步的多线程程序。JMM 为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0，null，false）。

下图展示了这三类程序在 JMM 中与在顺序一致性内存模型中的执行结果的异同：



只要多线程程序是正确同步的，JMM 保证该程序在任意的处理器平台上的执行结果，与该程序在顺序一致性内存模型中的执行结果一致。

JSR-133 对旧内存模型的修补

JSR-133 对 JDK5 之前的旧内存模型的修补主要有两个：

- 增强 volatile 的内存语义。旧内存模型允许 volatile 变量与普通变量重排序。JSR-133 严格限制 volatile 变量与普通变量的重排序，使 volatile 的写-读和锁的释放-获取具有相同的内存语义。
- 增强 final 的内存语义。在旧内存模型中，多次读取同一个 final 变量的值可能会不相同。为此，JSR-133 为 final 增加了两个重排序规则。现在，final 具有了初始化安全性。