Java内存模型

关键字： `Java` 内存模型 并发 重排序 顺序一致性

> Java并发编程中涉及的JMM内存模型备忘。

内存模型

内存模型描述的是程序中各变量（实例域、静态域和数组元素）之间的关系，以及在实际计算机系统中将变量存储到内存和从内存取出变量这样的低层细节。对象最终存储在内存中，但编译器、运行库、处理器或缓存可以有特权定时地在变量的指定内存位置存入或取出变量值。例如，编译器为了优化一个循环索引变量，可能会选择把它存储到一个寄存器中，或者缓存会延迟到一个更适合的时间，才把一个新的变量值存入主存。所有的这些优化是为了帮助实现更高的性能，通常这对于用户来说是透明的，但是对多处理系统来说，这些复杂的事情可能有时会完全显现出来。

显式内存模型的好处

像 C 和 C++ 这些语言就没有显示的内存模型 —— 但 C 语言程序继承了执行程序处理器的内存模型（尽管一个给定体系结构的编译器可能知道有关底层处理器的内存模型的一些情况，并且保持一致性的一部分责任也落到了该编译器的头上）。这意味着并发的 C 语言程序可以在一个，而不能在另一个处理器体系结构上正确地运行。虽然一开始 JMM 会有些混乱，但这有个很大的好处 —— 根据 JMM 而被正确同步的程序能正确地运行在任何支持 Java 的平台上。

内存模型的充要条件

在处理器层面上，内存模型定义了一个充要条件，“让当前的处理器可以看到其他处理器写入到内存的数据” 以及 “其他处理器可以看到当前处理器写入到内存的数据”。有些处理器有很强的内存模型(strong memory model)，能够让所有的处理器在任何时候任何指定的内存地址上都可以看到完全相同的值。而另外一些处理器则有较弱的内存模型（weaker memory model），在这种处理器中，必须使用内存屏障（一种特殊的指令）来刷新本地处理器缓存并使本地处理器缓存无效，目的是为了让当前处理器能够看到其他处理器的写操作或者让其他处理器能看到当前处理器的写操作。这些内存屏障通常在lock和unlock操作的时候完成。内存屏障在高级语言中对程序员是不可见的。

CPU内存指令重排序(Memory Reordering)

对主存的一次访问一般花费硬件的数百次时钟周期。处理器通过缓存（caching）能够从数量级上降低内存延迟的成本，这些缓存为了性能会重新排列待定内存操作的顺序。也就是说，程序的读写操作不一定会按照它要求处理器的顺序执行。当数据是不可变的，同时/或者数据限制在线程范围内，这些优化是无害的。如果把这些优化与对称多处理（symmetric multi-processing）和共享可变状

态（shared mutable state）结合，那么就是一场噩梦。当基于共享可变状态的内存操作被重新排序时，程序可能行为不定。一个线程写入的数据可能被其他线程可见，原因是数据写入的顺序不一致。适当的放置内存屏障，通过强制处理器顺序执行待定的内存操作来避免这个问题。

重排序的背景

现代CPU的主频越来越高，与cache的交互次数也越来越多。当CPU的计算速度远远超过访问cache时，会产生cache wait，过多的cache wait就会造成性能瓶颈。

针对这种情况，多数架构（包括X86）采用了一种将cache分片的解决方案，即将一块cache划分成互不关联地多个 slots (逻辑存储单元，又名 Memory Bank 或 Cache Bank)，CPU可以自行选择在多个 idle bank 中进行存取。这种 SMP 的设计，显著提高了CPU的并行处理能力，也回避了cache访问瓶颈。

重排序的种类

- `编译期重排`：编译源代码时，编译器依据对上下文的分析，对指令进行重排序，以之更适合于CPU的并行执行。
- `运行期重排`：CPU在执行过程中，动态分析依赖部件的效能，对指令做重排序优化。

重排序原理实例

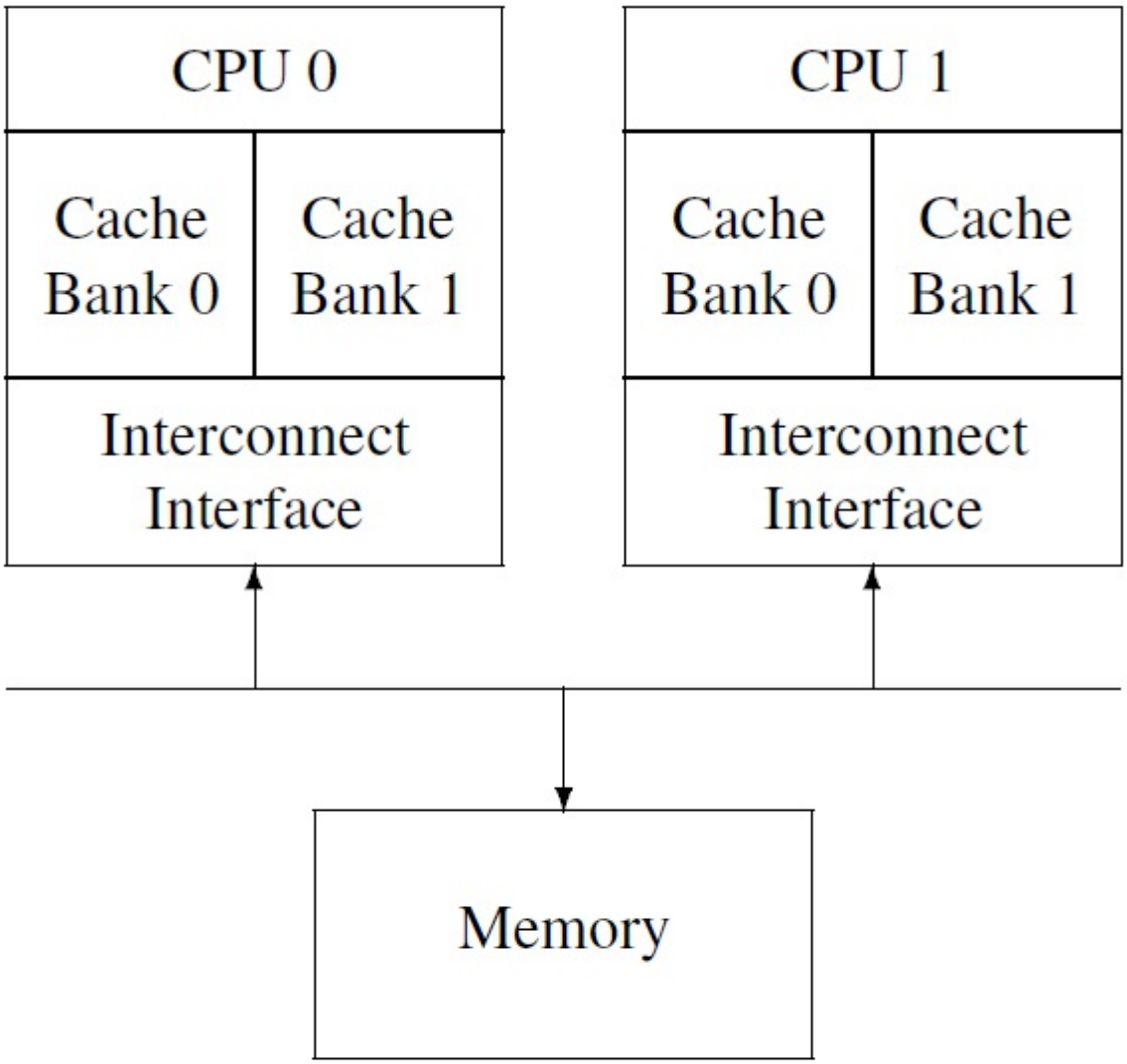


Figure 2: Hardware Parallelism

从图中可以看到，这是一台配备双CPU的计算机，cache 按地址被分成了两块 cache banks，分别是 cache bank0 和 cache bank1。

- Memory Bank的划分

> 一般 Memory bank 是按cache address来划分的。比如偶数adress 0x12345000 分到 bank 0, 奇数address 0x12345100 分到 bank1。

- 理想的内存访问指令顺序：

  i. CPU0 往 cache address 0x12345000 写入一个数字1。因为address 0x12345000是偶数，所以值被写入 bank0。
  ii. CPU1 读取 bank0 address 0x12345000 的值，即数字1。
  iii. CPU0 往 cache 地址 0x12345100 写入一个数字2。因为address 0x12345100 是奇数，所以值被写入 bank1.

iv. CPU1 读取 bank1 address 0x12345100 的值，即数字2。

- 重排序后的内存访问指令顺序：
  i. CPU0 准备往 bank0 address 0x12345000 写入数字1。
  ii. CPU0 检查 bank0 的可用性，发现 bank0 处于 busy 状态。
  iii. **CPU0 为了防止 cache等待，发挥最大效能，将内存访问指令重排序。** 即先执行后面的 bank1 address 0x12345100 数字2的写入请求。
  iv. CPU0 检查 bank1 可用性，发现bank1处于 idle 状态。
  v. CPU0 将数字2 写入 bank 1 address 0x12345100。
  vi. **CPU1来读取 0x12345000，未读到数字1，出错。**
  vii. CPU0 继续检查 bank0 的可用性，发现这次 bank0 可用了，然后将数字1 写入 0x12345000。
  viii. CPU1 读取 0x12345100，读到数字2，正确。

从上述触发步骤中，可以看到第 3 步发生了指令重排序，并导致第 6 步读到错误的数据。

> 重排序引发多线程的内存可见性问题，JMM会禁止特定类型的编译器重排序，同时在Java编译器在生成的指令序列的适当位置插入内存屏障以此来禁止特定类型的处理器从排序。

重排序规则

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。

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`

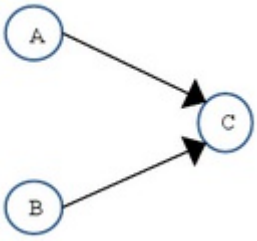
程序的执行结果不能被改变。编译器，runtime 和处理器都必须遵守as-if-serial语义。

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：

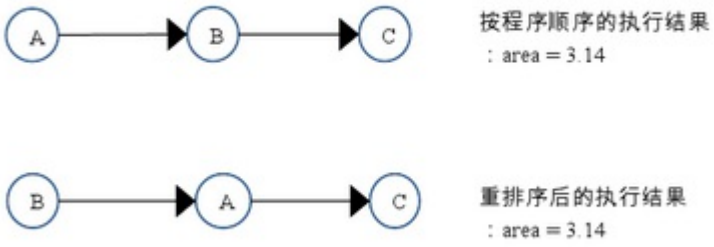
```
double pi  = 3.14;        //A
double r   = 1.0;         //B
double area = pi * r * r; //C
```

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（ happens-before ）

> happens-before 定义：如果操作A happends-before 操作B，那么操作A的结果将对操作B **可见** 。

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：

```
A happens-before B；
B happens-before C；
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。

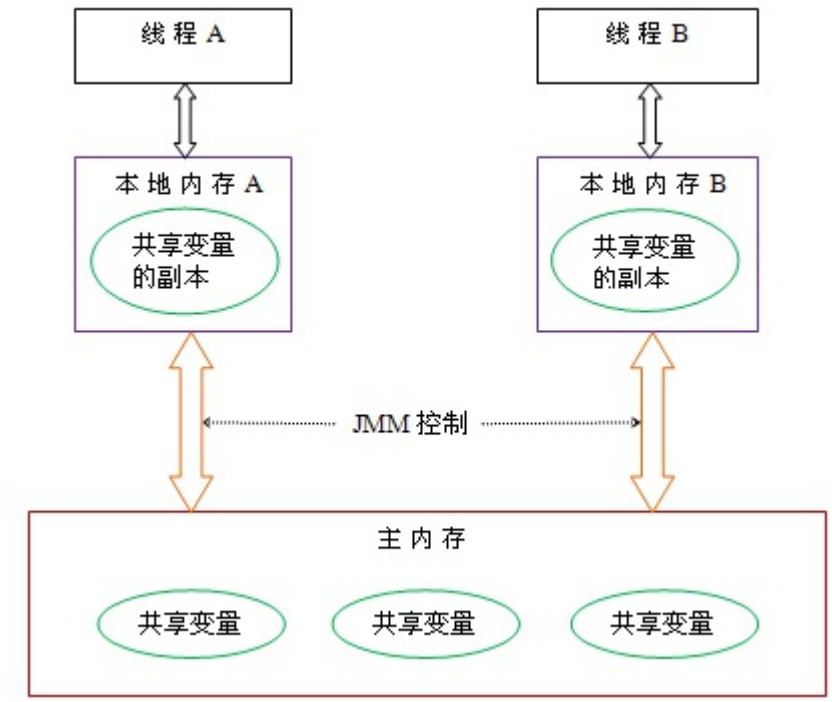
这里A happens-before B，但实际执行时B却可以排在A之前执行（看上面的重排序后的执行顺序）。如果A happens-before B，JMM并不要求A一定要在B之前执行。JMM仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。这里操作A的执行结果不需要对操作B可见，且重排序操作A和操作B后的执行结果，与操作A和操作B按happens-before顺序执行的结果一致。在这种情况下，JMM会认为这种重排序并不非法（not illegal），JMM允许这种重排序。

> 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。

在单线程程序中，对存在控制依赖`if (flag)`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。

Java内存模型

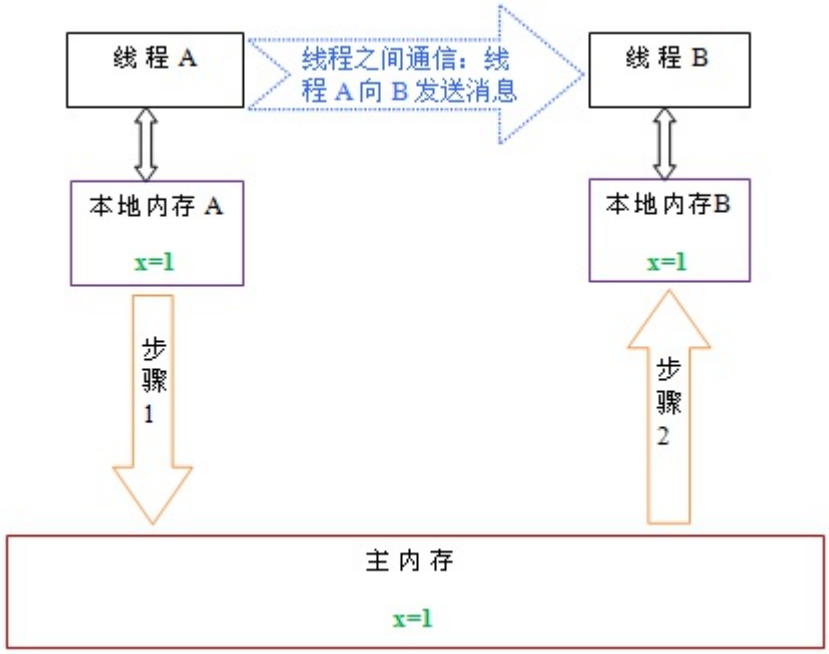
Java内存模型的抽象示意图



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：



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。

同步

同步包括两方面的含义：`独占性`和`可见性`。

根据Java Language Specification中的说明, JVM系统中存在一个主内存(Main Memory或Java Heap Memory)，Java中所有变量都储存在主存中，对于所有线程都是共享的。

每条线程都有自己的工作内存(Working Memory)，工作内存中保存的是主存中某些变量的拷贝，`线程对所有变量的操作都是在工作内存中进行`，线程之间无法相互直接访问，变量传递均需要通过主存完成。

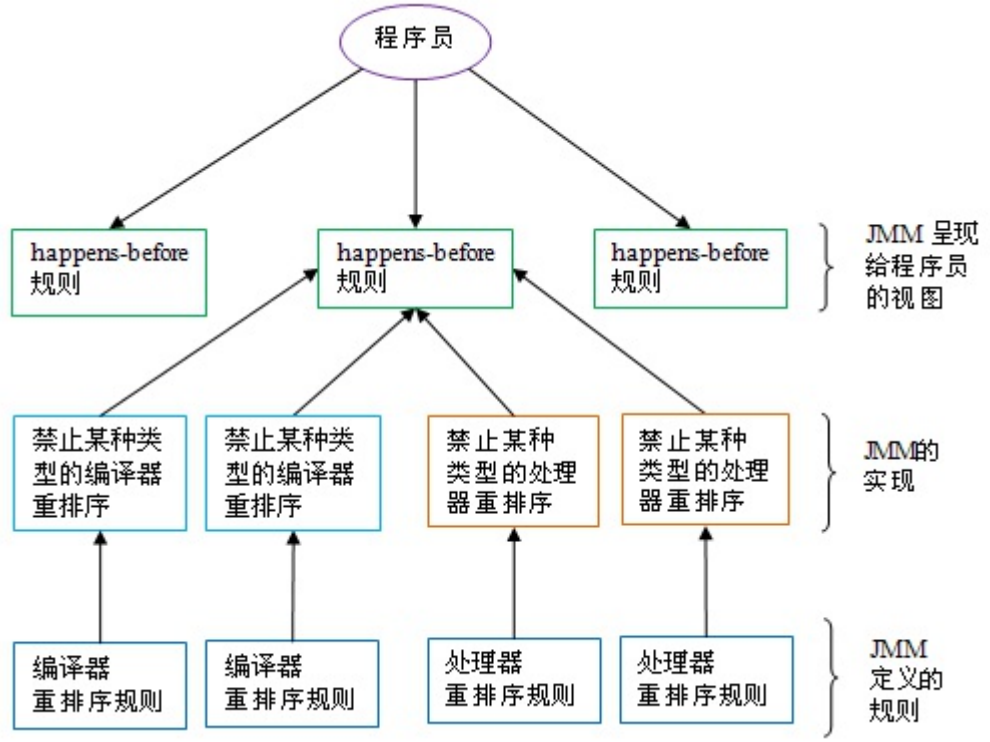
> 变量可见性——如果线程t1与线程t2分别被安排在了不同的处理器上面，那么t1与t2对于变量A的修改时相互不可见，如果t1给A赋值，然后t2又赋新值，那么t2的操作就将t1的操作覆盖掉了，这样会产生不可预料的结果。所以，即使有些操作时原子性的，但是如果不具有可见性，那么多个处理器中备份的存在就会使原子性失去意义。

happens-before 关系

- 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：对一个监视器的解锁 happens-before 于随后对该监视器（同一个锁）的加锁。
- 传递性：如果 A happens-before B 且 B happends-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。

happens-before与JMM的关系如下图所示：



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（因为重排序导致的）。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（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。

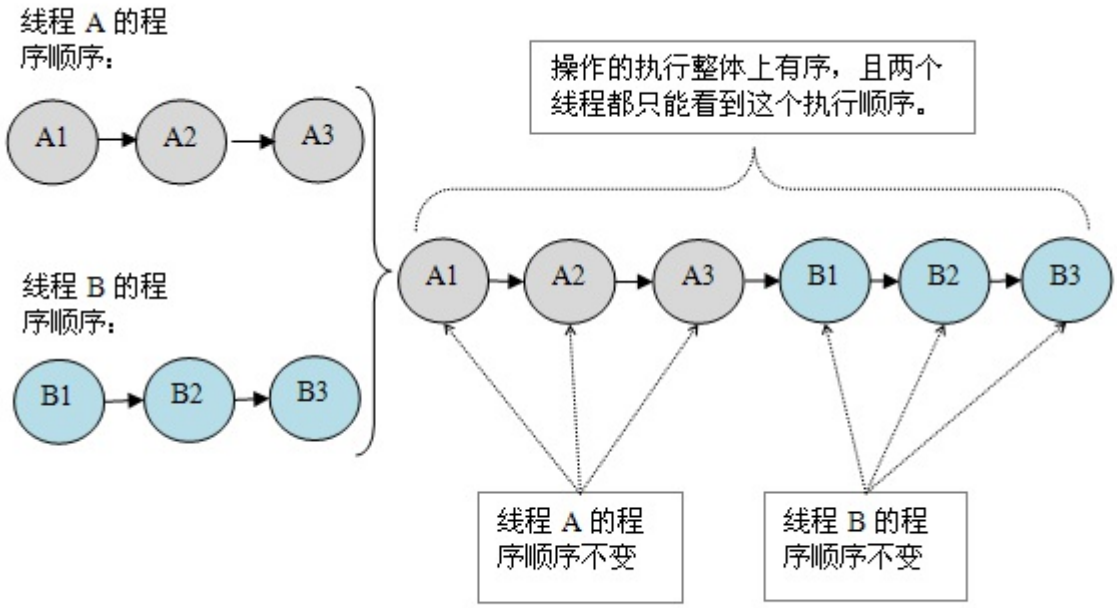
顺序一致性内存模型

顺序一致性内存模型有两大特性：

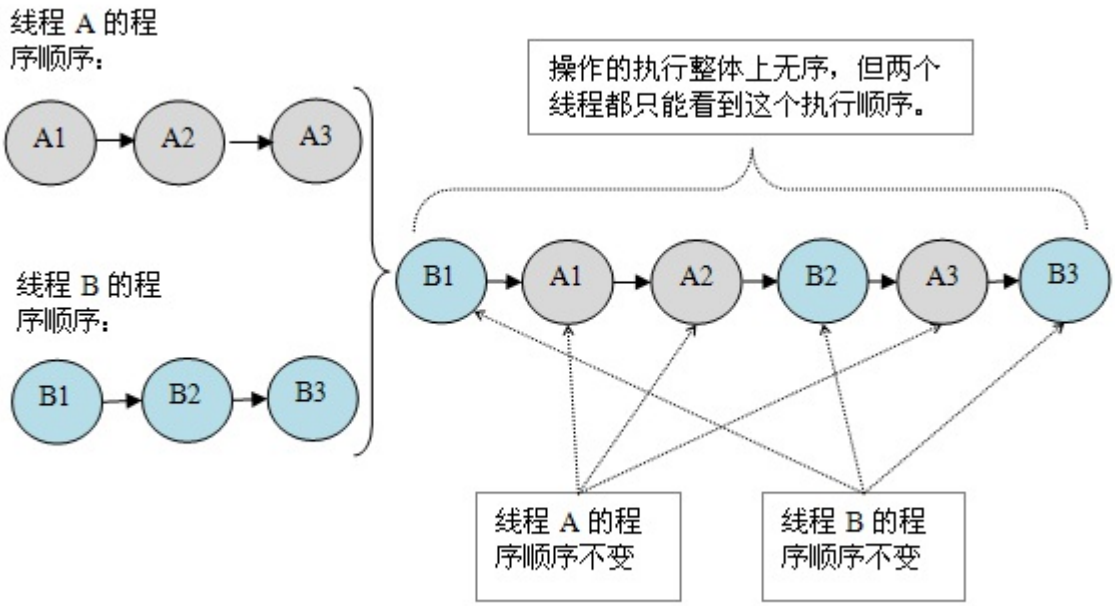
- 一个线程中的所有操作必须按照程序的顺序来执行。
- 不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

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：



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。

未同步程序在顺序一致性模型和JMM中的执行有下面几个差异：

- 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。
- 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。
- 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。

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。

volatile

`volatile 具有可见性`，对一个volatile变量的读，总是能看到对这个volatile变量最后的写入。同

时volaile具有原子性，对任意单个volatile变量的读/写具有原子性（ 对long、double类型很有意义，对v++这种复合操作不具原子性 ）。

volatile 的写-读与锁的释放-获取有相同的内存效果（写==释放锁 ，读==获取锁）。

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。

volatile 读/写的内存语义：当写一个volatile变量时，JMM会把对应的本地内存中的共享变量刷新到主存，实际是向接下来将要读取这个变量的某个线程发出消息。当读一个voltaile变量时，JMM会把该线程对应的本地缓存置为无效，线程接下来将从主存中读取共享变量。实际是接收到了之前某个线程发出的消息。因此 volatile读/写实际是两个线程之间通过主存进行通信 。

volatile写和volatile读的内存语义总结：

- 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。

为了实现volatile的内存语义，JMM针对编译器制定了以下3条重排序规则：

- 当第二个操作是volatile写时，不管第一个操作（volatile操作，普通操作）是什么，都不能重排序。
- 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。
- 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

final 域

对于final域，编译器和处理器遵守两条重排序规则：

- 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，这两个操作不能重排序。

- 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。

JMM禁止把final域的写重排序到构造函数之外，由此可保证在对象引用为任意线程可见之前，对象final域已经被正确初始化了（普通域不具有这个保障），当然前提条件是final引用不能从构造函数内“逸出”（对象未构造完成之前就将其引用暴露出去）。编译器会在final域写之后，构造函数返回之前插入StoreStore屏障，由此便确保了final域的写不会重排序到构造函数之外。

通过为final域增加读/写重排序规则，可以为Java程序员提供初始化安全保证： 只要对象是正确初始化的（被构造对象的引用在构造函数中没“逸出”），那么不需要同步就可以保证任意线程都能看到final域在构造函数中被初始化的值。

synchronized（内置锁）

synchronized为一段操作或内存进行加锁，它具有互斥性。当线程要操作被synchronized修饰的内存或操作时，必须首先获得锁才能进行后续操作，但是在同一时刻只能有一个线程获得相同的一把锁（对象监视器），所以它只允许一个线程进行操作。

JMM关于synchronized的两条规定：

- 线程解锁前，必须把共享变量的最新值刷新到主内存中
- 线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。

下面是锁释放-获取的示例代码：

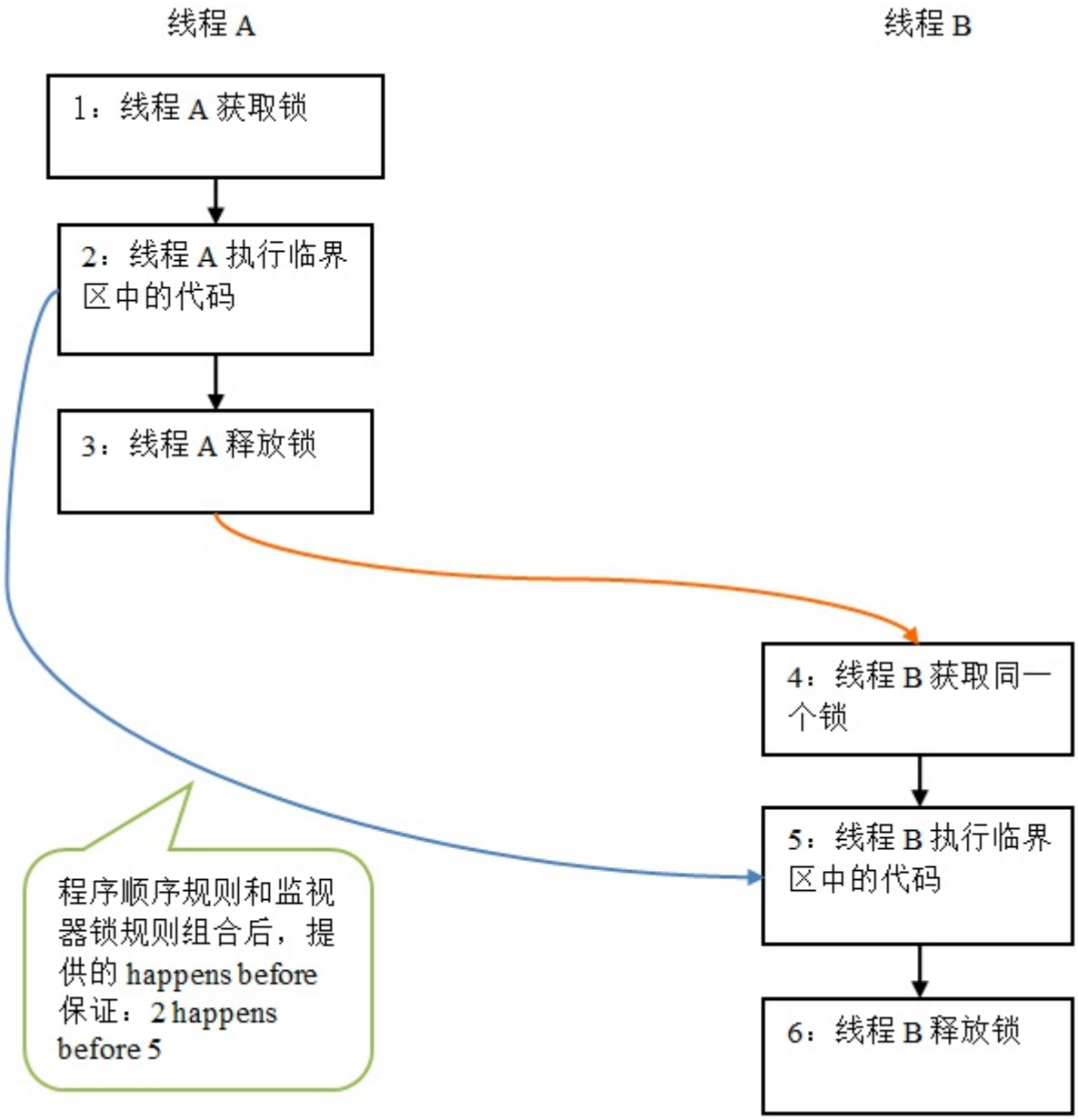
```
class MonitorExample {
    int a = 0;

    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关系可以分为三类：

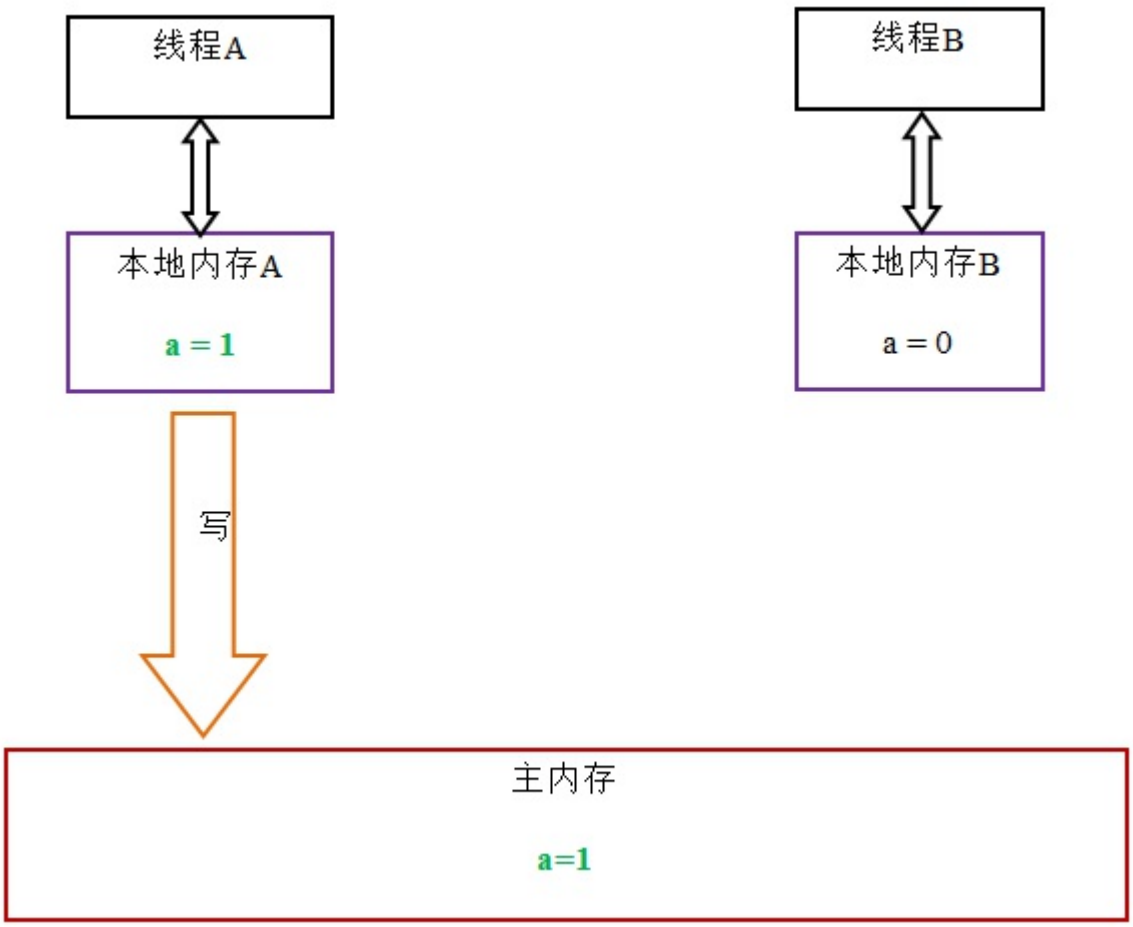
- 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
- 根据监视器锁规则，3 happens before 4。
- 根据happens before的传递性，2 happens before 5。



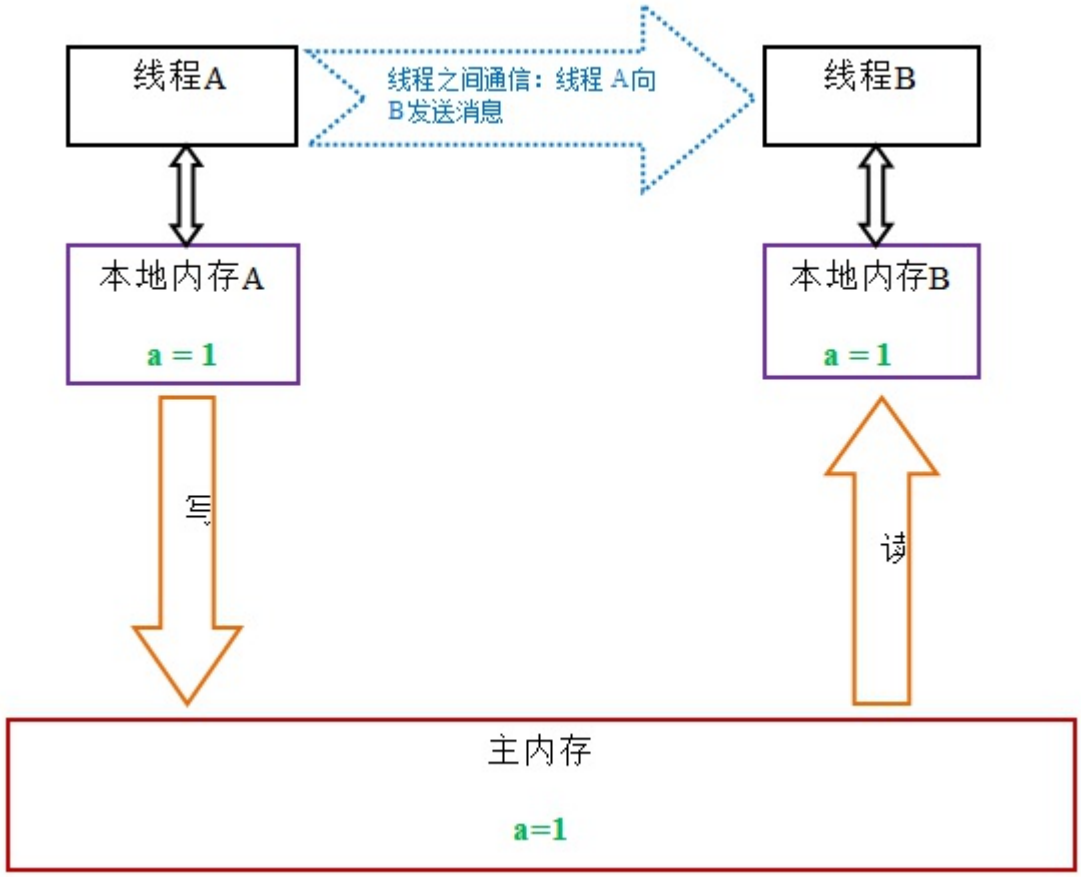
在上图中，每一个箭头链接的两个节点，代表了一个happens before关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的happens before保证。

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 （内置锁）。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。

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。



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。

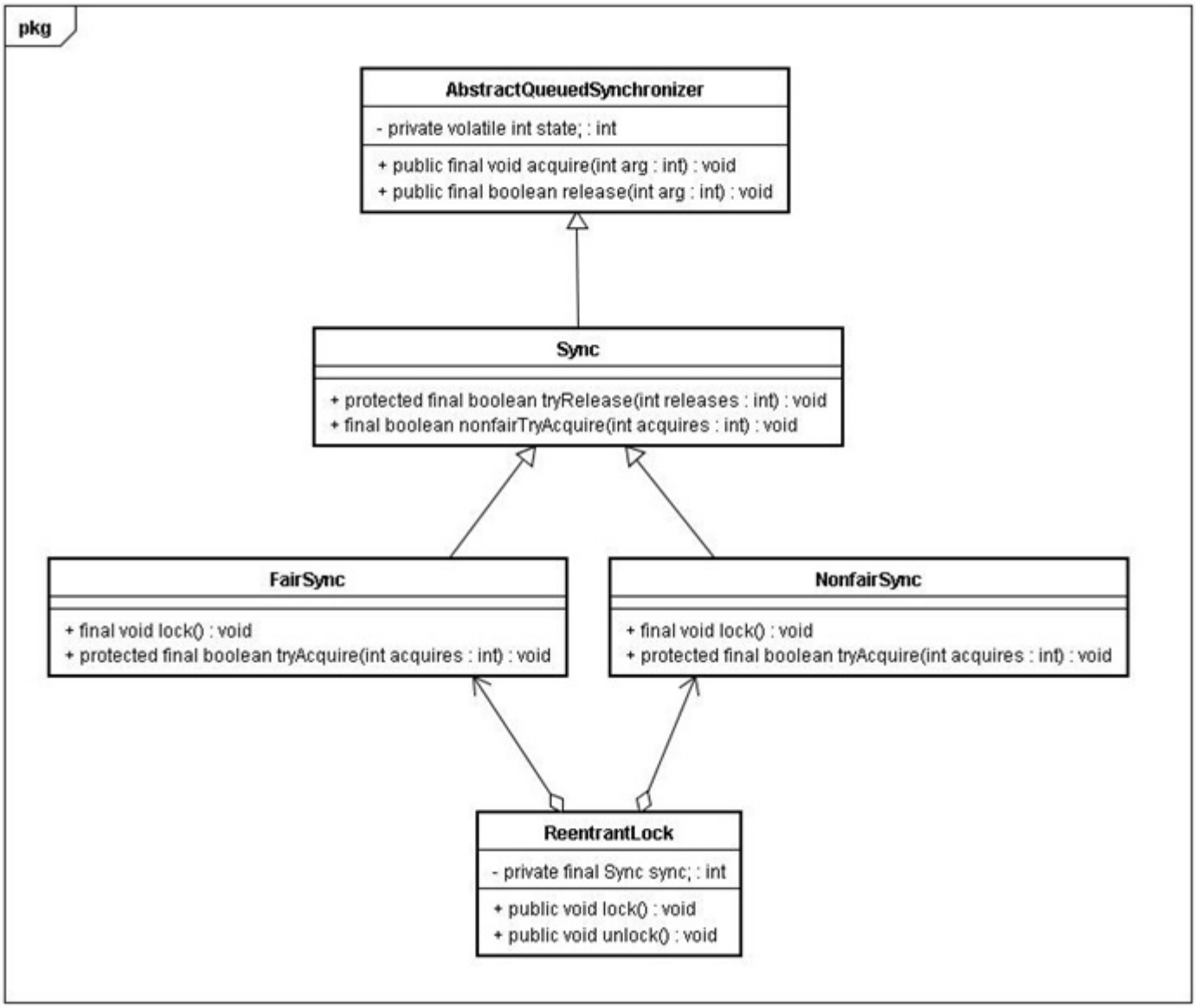


锁释放和锁获取的内存语义总结：

- 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

Java同步器框架 AbstractQueuedSynchronizer

AQS使用一个整型的volatile变量（命名为state）来维护同步状态。



锁释放-获取的内存语义的实现至少有下面两种方式：

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

CAS（Compare And Swap）

CAS 指的是现代 CPU 广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。这个指令会对内存中的共享数据做原子的读写操作。

CAS指令的操作过程：首先，CPU 会将内存中将要被更改的数据与期望的值做比较。然后，当这两个值相等时，CPU 才会将内存中的数值替换为新的值。否则便不做操作。最后，CPU 会将旧的数值返回。

这一系列的操作是原子的。它们虽然看似复杂，但却是 Java 5 并发机制优于原有锁机制的根本。简单来说，CAS 的含义是“我认为原有的值应该是什么，如果是，则将原有的值更新为新值，否则不做修改，并告诉我原来的值是多少”。（这段描述引自《Java并发编程实战》）

简单的来说，`CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B`。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则返回V。这是一种乐观锁的思路，它相信在它修改之前，没有其它线程去修改它；而Synchronized是一种悲观锁，它认为在它修改之前，一定会有其它线程去修改它，悲观锁效率很低。

CAS模拟实现:

```
public class SimulatedCAS {

    private int value;

    public SimulatedCAS(int value) {
        this.value = value;
    }

    public synchronized int get(){
        return value;
    }

    public synchronized int compareAndSwap(int expectedValue, int newValue){
        int oldValue = value;//获取旧值
        if(oldValue == expectedValue){//如果期望值与当前V位置的值相同就给予新值
            value = newValue;
        }
        return oldValue;//返回V位置原有的值
    }

    public synchronized boolean compareAndSet(int expectedValue, int newValue){
        return ( expectedValue == compareAndSwap(expectedValue, newValue));
    }
}
```

CAS使用，以AtomicInteger为例:

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}

public final int decrementAndGet() {
    for (;;) {
        int current = get();
        int next = current - 1;
        if (compareAndSet(current, next))
            return next;
```

```
        }
    }
```

concurrent包

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：

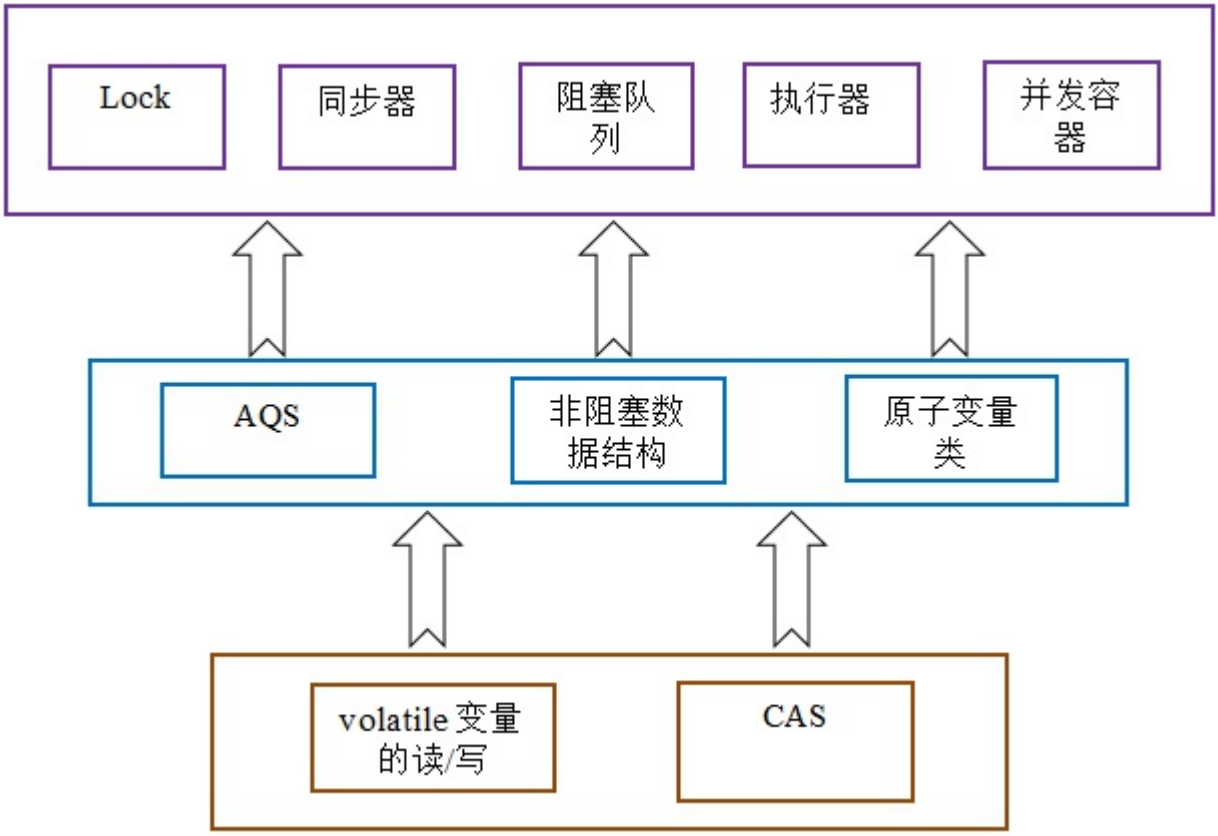
- A线程写volatile变量，随后B线程读这个volatile变量。
- A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
- A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。
- A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

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。

concurrent包实现模式：

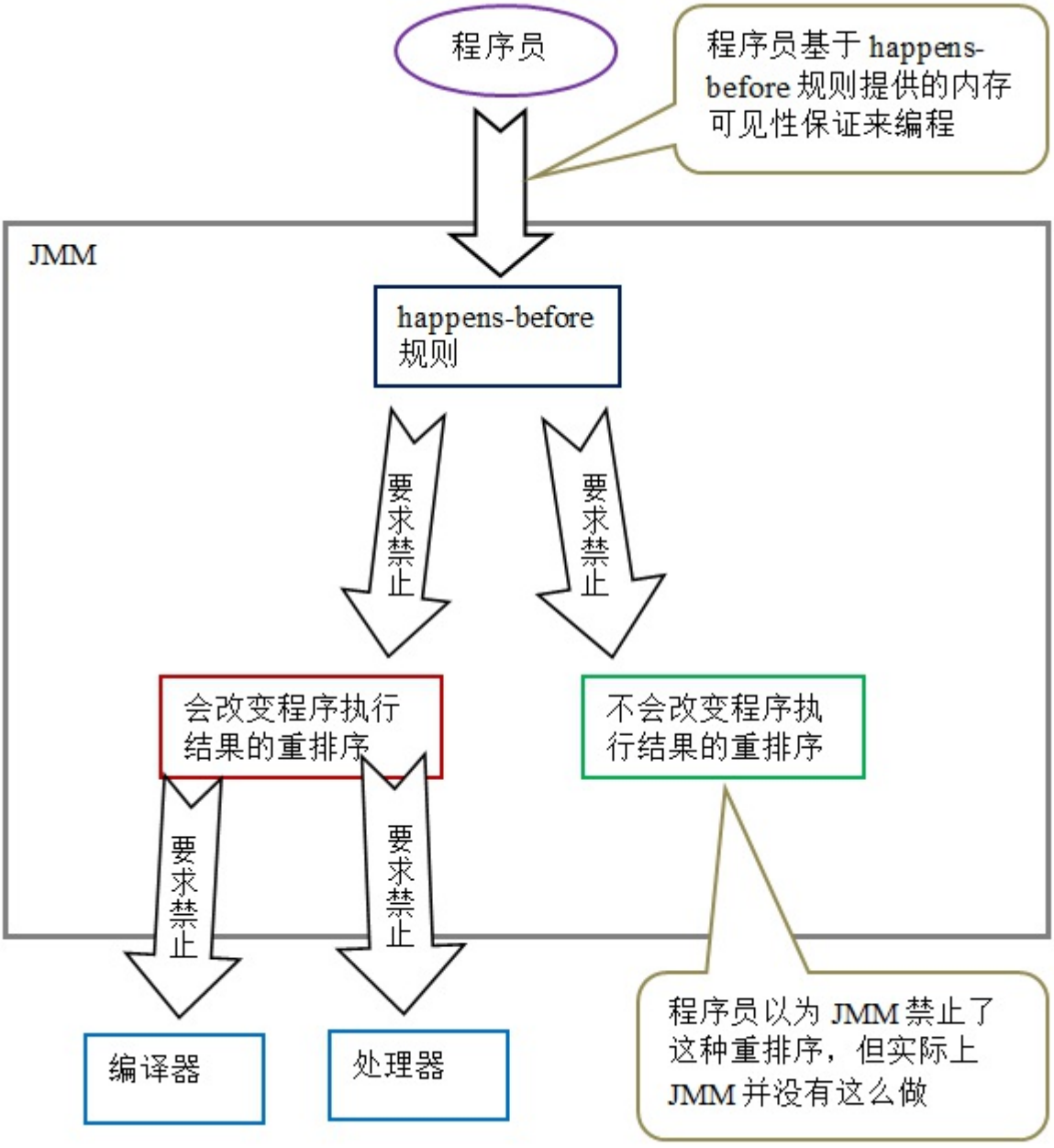
- 首先，声明共享变量为volatile；
- 然后，使用CAS的原子条件更新来实现线程之间的同步；
- 同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

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：



总结

JMM的设计示意图：



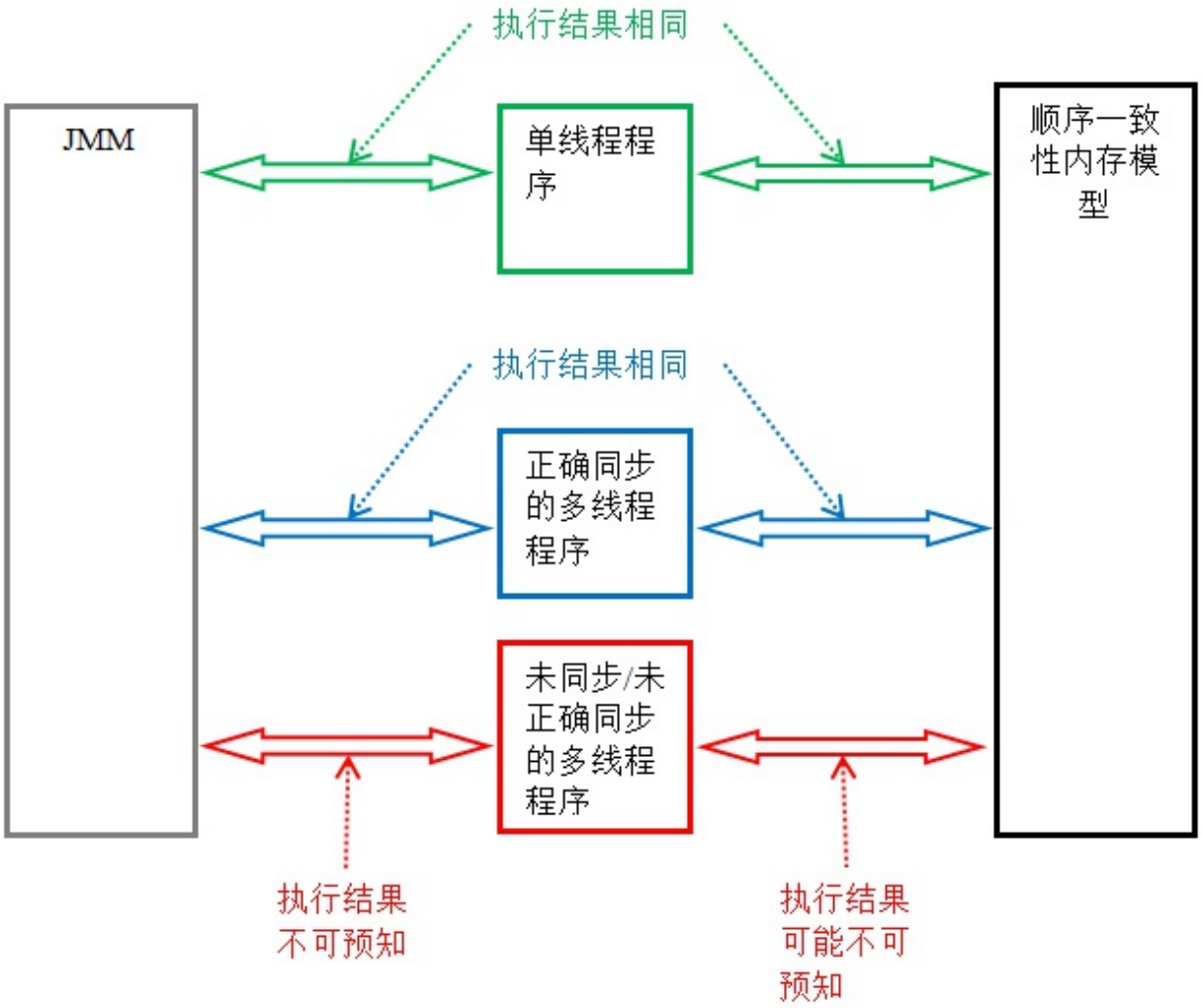
从上图可以看出两点：

- 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：

- 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。
- 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。
- 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）。



参考资料

- 深入理解Java内存模型
- Java内存模型FAQ
- 《Java并发编程实战》