第 3 章　Java 内存模型

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，本章将揭开 Java 内存模型神秘的面纱。本章大致分 4 部分：Java 内存模型的基础，主要介绍内存模型相关的基本概念；Java 内存模型中的顺序一致性，主要介绍重排序与顺序一致性内存模型；同步原语，主要介绍 3 个同步原语（synchronized、volatile 和 final）的内存语义及重排序规则在处理器中的实现；Java 内存模型的设计，主要介绍 Java 内存模型的设计原理，及其与处理器内存模型和顺序一致性内存模型的关系。

3.1　Java 内存模型的基础

3.1.1　并发编程模型的两个关键问题

在并发编程中，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，通过写-读内存中的公共状态进行隐式通信。在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过发送消息来显式进行通信。

同步是指程序中用于控制不同线程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。

3.1.2 Java 内存模型的抽象结构

在 Java 中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享（本章用“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（ExceptionHandler 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如图 3-1 所示。

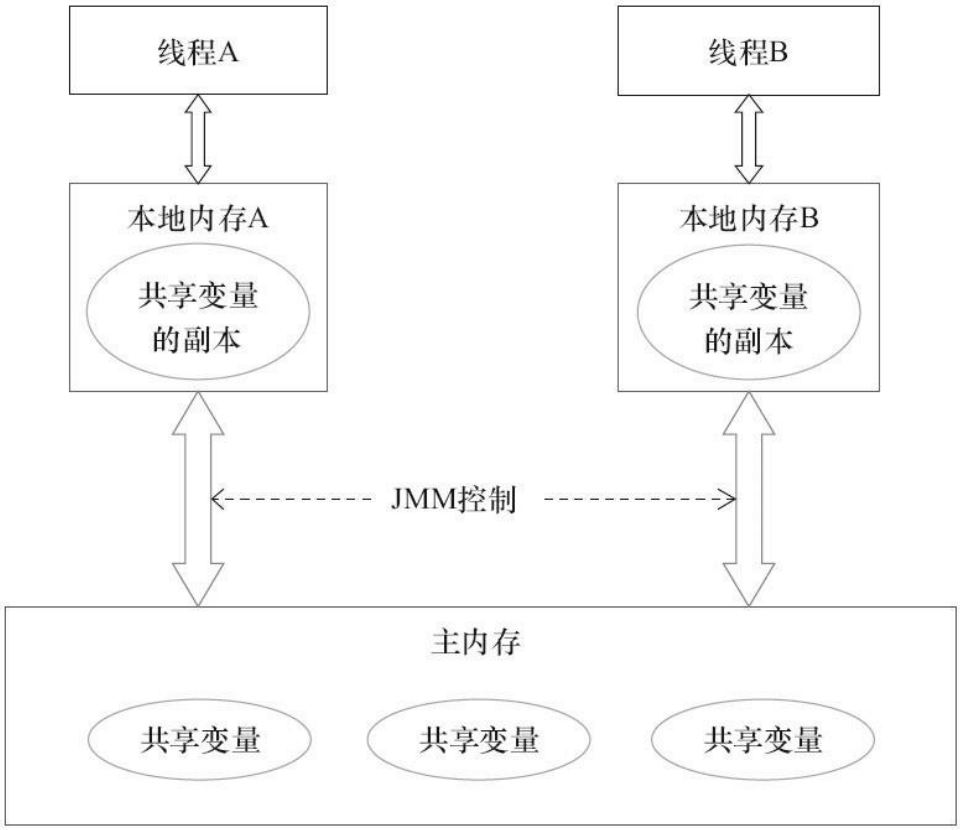

图 3-1

从图 3-1 来看，如果线程 A 与线程 B 之间要通信的话，必须要经历下面 2 个步骤。

- 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。
- 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（见图 3-2）来说明这两个步骤。

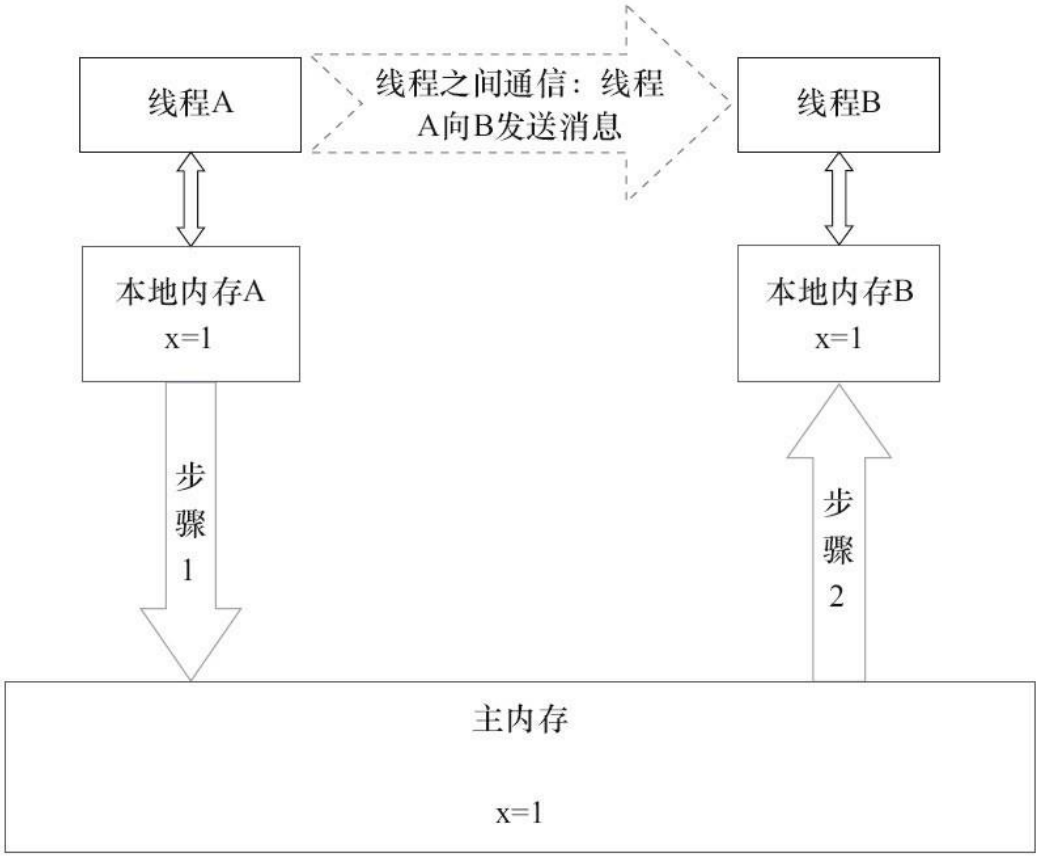

图 3-2 线程之间的通信图

如图 3-2 所示，本地内存 A 和本地内存 B 由主内存中共享变量 x 的副本。假设初始时，这 3 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。

3.1.3 从源代码到指令序列的重排序

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3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-LevelParallelism，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3种重排序，如图3-3所示。

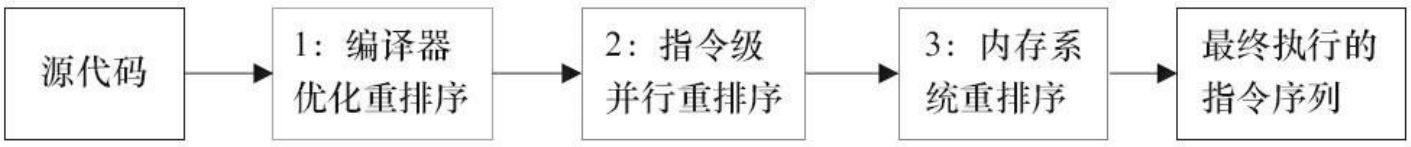

图3-3　从源码到最终执行的指令序列的示意图

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。

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。

3.1.4 并发编程模型的分类

现代的处理器使用写缓冲区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，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，请看下面的表 3-1。

处理器 / 示例项	Processor A	Processor B

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

运行结果	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，最终可能得到 x=y=0 的结果。具体的原因如图 3-4 所示。

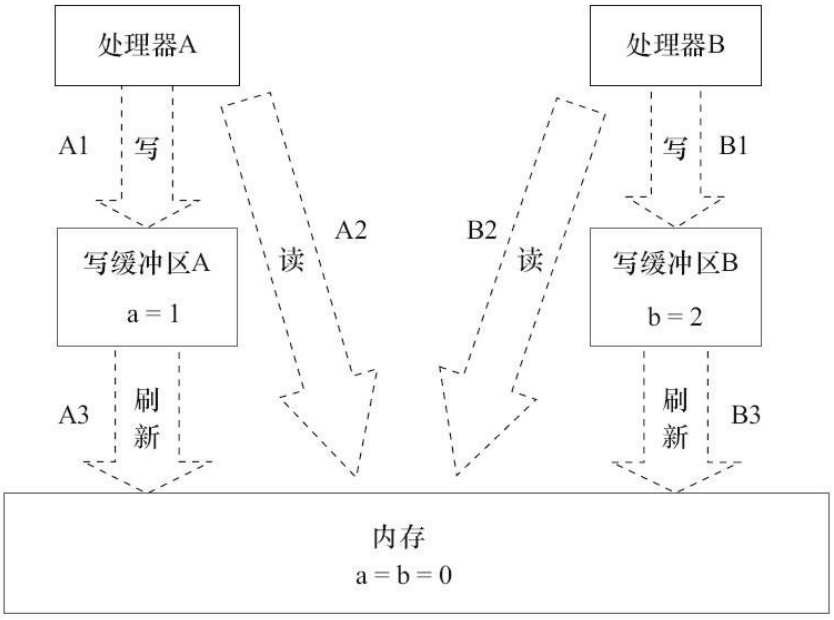

图 3-4 处理器和内存的交互