9 Java 内存模型

Java 平台把线程和多处理技术集成到了语言中，这种集成程度比以前的大多数编程语言都要强很多。该语言对于平台独立的并发及多线程技术的支持是野心勃勃并且是具有开拓性的，或许并不奇怪，这个问题要比 Java 体系结构设计者的原始构想要稍微困难些。关于同步和线程安全的许多底层混淆是 Java 内存模型 (Java Memory Model,JMM)的一些难以直觉到的细微差别。

例如，并不是所有的多处理器系统都表现出缓存一致性（cache coherency）；假如有一个处理器有一个更新了的变量值位于其缓存中，但还没有被存入主存，这样别的处理器就可能会看不到这个更新的值。在缓存缺乏一致性的情况下，两个不同的处理器可以看到在内存中同一位置处有两种不同的值。这听起来不太可能，但是这却是故意的 —— 这是一种获得较高的性能和可伸缩性的方法 —— 但是这加重了开发者和编译器为解决这些问题而编写代码的负担。

9.1 Java 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。

JMM 允许编译器和缓存对数据在处理器特定的缓存（或寄存器）和主存之间移动的次序拥有重要的特权，除非程序员已经使用 synchronized 或 final 明确地请求了某些可见性保证。这意味着在缺乏同步的情况下，从不同的线程角度来看，内存的操作是以不同的次序发生的。

许多没有正确同步的程序在某些情况下似乎工作得很好，例如在轻微的负载下、在单处理器系统上，或者在具有比 JMM 所要求的更强的内存模型的处理器上。

“重新排序”(reordering)这个术语用于描述几种对内存操作的真实明显的重新排序的类型：

1) 当编译器不会改变程序的语义时，作为一种优化它可以随意地重新排序某些指令。

2) 在某些情况下，可以允许处理器以颠倒的次序执行一些操作。

3) 通常允许缓存以与程序写入变量时所不相同的次序把变量存入主存。

从另一线程的角度来看，任何这些条件都会引发一些操作以不同于程序指定的次序发生 —— 并且忽略重新排序的源代码时，内存模型认为所有这些条件都是同等的。

9.1.1 可见性

1. 同步与可见性（visibility）

大多数程序员都知道，synchronized 关键字强制实施一个互斥锁（互相排斥），这个互斥锁防止每次有多个线程进入一个给定监控器所保护的同步语句块。但是同步还有另一个方面：正如 JMM 所指定，它强制实施某些**内存可见性规则**。它确保了当存在一个同步块时缓存被更新，当输入一个同步块时缓存失效。因此，在一个由给定监控器保护的同步块期间，一个线程所写入的值对于其余所有的执行由同一监控器所保护的同步块的线程来说是可见的。它也确保了编译器不会把指令从一个同步块的内部移到外部（虽然在某些情况下它会把指令从同步块的外部移到内部）。JMM 在缺乏同步的情况下不会做这种保证 —— 这就是只要有多个线程访问相同的变量时必须使用同步（或者它的同胞，易失性）的原因。

2. 不可变对象的问题

不可变对象似乎可以改变它们的值（这种对象的不变性旨在通过使用 final 关键字来得到保证）。让一个对象的所有字段都为 final 并不一定使得这个对象不可变 —— 所有的字段还必须是原语类型或是对不可变对象的引用。不可变对象（如 String ）被认为不要求同步。但是，因为在将内存写方面的更改从一个线程传播到另一个线程时存在潜在的延迟，所以有可能存在一种竞态条件，即允许一个线程首先看到不可变对象的一个值，一段时间之后看到的是一个不同的值。

3. Volatile 与可见性

可见性——如何知道当线程 A 执行 someVariable=3 时，其他线程是否可以看到线程 A 所写的值 3？有一些原因使其他线程不能立即看到 someVariable 的值 3：可能是因为编译器为了执行效率更高而重新排序了指令，也可能是 someVariable 缓存在寄存器中，或者

它的值写到写处理器的缓存中、但是还没有刷新到主存中，或者在读处理器的缓存中有一个老的（或者无效的）值。内存模型决定什么时候一个线程可以可靠地“看到”由其他线程对变量的写入。特别是，内存模型定义了保证内存操作跨线程的可见性的 volatile 、synchronized 和 final 的语义。

在多个线程访问同一个变量时，必须使用同步或者 volatile。volatile 语义保证 volatile 字段的读写直接在主存而不是寄存器或者本地处理器缓存中进行，并且代表线程对 volatile 变量进行的这些操作是按线程要求的顺序进行的。换句话说，这意味着老的内存模型保证正在读或写的变量的可见性，不保证写入其他变量的可见性。Volatile 变量可以与对非 volatile 变量的读写一起重新排序。

如果当线程 A 写入 volatile 变量 V，而线程 B 读取 V 时，那么在写入 V 时，A 可见的所有变量值现在都可以保证对 B 是可见的。代价是访问 volatile 字段时会对性能产生更大的影响。

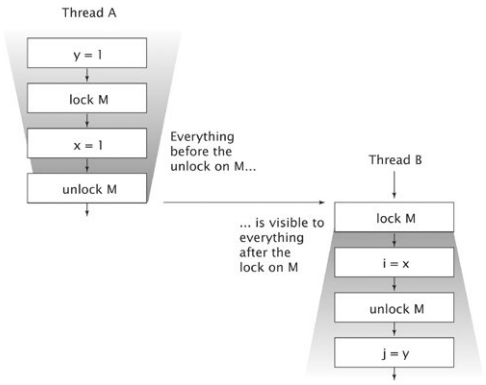
9.1.2 发生前关系（happen-before）

像对变量的读写这样的操作，在线程中是根据所谓的“程序顺序”——程序的语义声明它们应当发生的顺序——排序的。在不同线程中的操作完全不一定要彼此排序——如果启动两个线程并且它们对任何公共监视器都不用同步执行、或者不涉及任何公共 volatile 变量，则完全无法准确地预言一个线程中的操作（或者对第三个线程可见）相对于另一个线程中操作的顺序。

排序保证是在线程启动、一个线程参与另一个线程、一个线程获得或者释放一个监视器（进入或者退出一个同步块）、或者一个线程访问一个 volatile 变量时创建的。JMM 描述了程序使用同步或者 volatile 变量以协调多个线程中的活动时所进行的顺序保证。新的 JMM 非正式地定义了一个名为 happens-before 的排序，它是程序中所有操作的部分顺序。

1) 一个线程中的每个操作“发生之前”于这个线程程序规定的其他后续出现的操作。
2) 对监视器的解锁“发生之前”于同一监视器上的所有后续锁定。
3) 对 volatile 变量的写“发生之前”于同一 volatile 变量 的每一个后续读。
4) 一个线程的 Thread.start()调用“发生之前”于这个启动后的线程的其他操作。
5) 线程中的所有操作“发生之前” 从这个线程的 Thread.join() 成功返回的所有其他线程。

例如，下面是用同步保证线程内存写的可见性。



9.2 初始化安全性

JMM 还寻求提供一种新的初始化安全性保证——只要对象是正确构造的(意即不会在构造函数完成之前发布对这个对象的引用)，然后所有线程都会看到在构造函数中设置的 final 字段的值，不管是否使用同步在线程之间传递这个引用。而且，所有可以通过正确构造的对象的 final 字段可及的变量，如用一个 final 字段引用的对象的 final 字段，也保证对其他线程是可见的。这意味着如果 final 字段包含，比如说对一个 LinkedList 的引用，除了引用的正确的值对于其他线程是可见的外，这个 LinkedList 在构造时的内容在不同步的情况下，对于其他线程也是可见的。

可以不用同步安全地访问这个 final 字段，编译器可以假定 final 字段将不会改变，因而可以优化多次提取。

在构造函数的 final 字段的写与在另一个线程中对这个对象的共享引用的初次装载之间有一个类似于 happens-before 的关系。当构造函数完成任务时，对 final 字段的所有写（以及通过这些 final 字段间接可及的变量）变为“冻结”，所有在冻结之后获得对这个对象的引用的线程都会保证看到所有冻结字段的冻结值。初始化 final 字段的写将不会与构造函数关联的冻结后面的操作一起重新排序。

初始化安全性保证了在多线程共享情况下不可变对象的正确构造。

参考文献

1） 修 Java 内 存 模 型 （ 1 ）， http://www.ibm.com/developerworks/cn/java/j-jtp02244/

2） 修 Java 内 存 模 型 （ 2 ）， http://www.ibm.com/developerworks/cn/java/j-jtp03304/

3） 实 现 一 个 不 受 约 束 的 不 变 性 模 型， http://www.ibm.com/developerworks/cn/java/j-immutability.html

4） 一种新的 Java 存储模型 L2JMM，计算机研究与发展，2006，43（4）