Java内存模型调研结果

周宇强

June 10, 2018

1 概述

1.1 引出

Java memory model（以下简称为JMM）是隶属于java虚拟机（JVM）的一部分，它定义了JVM在计算机内存的工作方式。JMM的定义是为了屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java 程序在各种平台上都能达到一致的内存访问效果。从Java 5开始，Java内存模型成为Java语言规范的一部分。

上述内存访问差异主要表现在，不同架构下的物理机拥有不一样的内存模型，在特定操作协议下对特定的内存或者高速缓存进行读写访问的过程不同。

具体如C/C++语言，直接采用的物理机和操作系统的内存模型，所以会存在有不同平台下的内存访问差异，并发访问出错。而JMM的定义可以使得java程序可以在一次编写之后，在不同平台上运行。

1.2 目标

JMM的主要目标是定义了程序中各个变量的访问规则，及虚拟机中将变量存储到内存和从内存中取出变量的底层细节。

此处的变量不同于java编程时的变量，包括实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数这些线程私有，不被共享的元素。

1.3 要求

以下是JSR（Java Specification Requests）中对JMM做出的要求：

- 线程内部编译器转换是合法的
- 正确同步的程序具有顺序一致的行为
- 适应内存访问重排序和同步操作
- 标准处理器内存模型是合法的
- 无用的同步可以被忽略
- 为错误同步程序提供保证

1.4 介绍

介绍java内存模型之前，我们先看看两个内存模型，连续一致性模型（Sequential Consistency Memory Model）和先行发生模型（Happens-Before Memory Model ）。

前者就是简单的保证程序顺序与代码顺序一致，每次共享内存的写都会同步到主存，问题在于性能方面受频繁的读写影响很大。后者定义了一种happen-before关系，在后续我们会详细提到，先行发生模型主要的问题在于约束太弱，代码可能会产生不可期（或者说有着好几种可能）的结果。

java内存模型是建立在先行发生的内存模型之上的，并且再此基础上，增强了一些约束。因为先行发生在多线程竞争访问共享数据的时候，导致的不可预期的结果，有一些是java内存模型可以接受的，有一些是java内存模型不可以接受的。

2 基本结构与操作

2.1 主内存与工作内存

Java内存模型定义了主内存与工作内存[1] 两个概念。

规定，所有的变量都存储在主内存中。此处的主内存只是虚拟机内存的一部分，虚拟机内存也只是包括计算机物理内存为虚拟机进程分配的那一部分。

每条线程还有自己的工作内存。线程的工作内存中保存了被该线程用到的变量的主内存副本。线程对变量的所有操作（读取、赋值），都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

2.2 内存间相互操作

2.2.1 8种基本操作

Java内存模型中定义了8种操作来完成，虚拟机实现时必须保证这8 种操作都是原子的，不可再分的。（对于double和long 类型的变量来说，load，store，read，write操作在某些平台上允许有例外）

8种基本操作具体分别为:

- lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
- unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用。

[1]这里所讲的主内存、工作内存与Java 内存区域中的Java堆、栈、方法区等并不是同一个层次的内存划分。工作内存是JMM的一个抽象概念，并不真实存在。

主内存主要对应java堆中的对象实例数据部分，而工作内存则对应于虚拟机栈中的部分数据。从更低层次上说，主内存就直接对应于物理硬件的内存，工作内存涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

- load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
- assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write 的操作。
- write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

2.2.2 规则

对于上述8种基本操作，在使用的过程中，需要遵守以下规则：

- 不允许read和load，store和write操作之一单独出现;
- 不允许一个线程丢弃它的最近的assign操作;
- 不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中;
- 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化的变量;
- 一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复多次执行，多次执行lock 后，只有执行相同次数的unlock操作，变量才会被解锁;
- 如果对一个变量执行lock操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始变量的值;
- 如果一个变量事先没有被lock操作锁定，那就不需要对其执行unlock操作，也不允许去unlock一个被其他线程锁定住的变量;
- 对一个变量执行unlock操作前，必须下把此变量同步回主内存中（执行store，write操作）;

3 几个基本原理

3.1 重排序

我们都知道，在执行程序的过程中，为了提高性能，编译器或处理器常对指令进行重排序。重排序一般可分为3种：

- 编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。
- 指令级并行的重排序。现代处理器采用了指令级并行技术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。
- 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，会依次经历上述三种重排序。为了保证内存的可见性（共享内存在各线程之间相互可见），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。

3.2 as-if-serial语义

As-if-serial语义的意思是，所有的动作(Action)都可以为了优化而被重排序，但是必须保证它们重排序后的结果和程序代码本身的应有结果是一致的。Java编译器、运行时和处理器都会保证单线程下的as-if-serial语义。

比如，为了保证这一语义，重排序不会发生在有数据依赖的操作之中。

- int a = 1;
- int b = 2;
- int c = a + b;

将上面的代码编译成Java字节码或生成机器指令，可视为展开成了以下几步动作（实际可能会省略或添加某些步骤）：1）对a赋值1；2）对b赋值2；3）取a的值；4）取b的值；5）将取到两个值相加后存入c。

在上面5个动作中，动作1可能会和动作2、4重排序，动作2可能会和动作1、3重排序，动作3可能会和动作2、4重排序，动作4可能会和1、3重排序。但动作1和动作3、5不能重排序。动作2和动作4、5不能重排序。因为它们之间存在数据依赖关系，一旦重排，as-if-serial语义便无法保证。

此外，为保证as-if-serial语义，Java异常处理机制也会为重排序做一些特殊处理。如下列代码：

- try {
- x = 1;
- y = 0 / 0;
- } catch (Exception e) {
- } finally {
- System.out.println(”x = ” + x);
- }

y = 0 / 0可能会被重排序在x = 1之前执行，为了保证最终不致于输出错误结果，JIT在重排序时会在catch语句中插入错误代偿代码，将x赋值为1，将程序恢复到发生异常时应有的状态。这种做法的确将异常捕捉的逻辑变得复杂了，但是JIT的优化的原则是，尽力优化正常运行下的代码逻辑，哪怕以catch块逻辑变得复杂为代价，毕竟，进入catch块内是一种“异常”情况的表现。

3.3 内存屏障

内存屏障，又称内存栅栏，是一个CPU指令，基本上它是一条这样的指令：保证特定操作的执行顺序。影响某些数据（或则是某条指令的执行结果）的内存可见性。

JMM将内存屏障指令分为了loadload、loadstore、storeload及storestore四种：

- loadload 屏障：用在两个load指令（load1、load2）之间，确保load1指令数据的装载，在load2及后续所有load指令数据的装载之前。
- loadstore 屏障：用在load指令（load1）与store指令（store1）之间，确保load1指令数据的装载，在store1指令及后续所有store指令的数据刷新到内存（对其他所有处理器可见）之前。
- storeload 屏障：用在store指令（store1）与load指令（load1）之间，确保store1指令数据刷新到内存，在load1指令及后续所有load 指令之前。storeload屏障会保证该屏障之前的所有内存访问指令（包括load和store）执行完，再执行屏障后续内存访问指令。
- storestore 屏障：用在两个store指令（store1、store2）之间，确保store1指令数据刷新到内存，在store2及后续所有store指令数据刷新到内存之前。

3.4 happen-before

从jdk5开始，java使用新的JSR-133内存模型，基于happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果这两个操作之间存在happens-before关系，那么前一个操作的执行结果需要对后一个操作可见（但不一定在后一个之前执行），这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

happens-before关系的规则

1 在同一个线程里面，按照代码执行的顺序（也就是代码语义的顺序），前一个操作先于后面一个操作发生

2 对一个monitor对象的解锁操作先于后续对同一个monitor对象的锁操作

3 对volatile字段的写操作先于后面的对此字段的读操作

4 对线程的start操作（调用线程对象的start()方法）先于这个线程的其他任何操作

5 一个线程中所有的操作先于其他任何线程在此线程上调用join()方法

6 传递性规则：如果A happens-before B，且B happens-before C，那么A happens-before C。

7 如果线程T1中断线程T2，该中断操作，happen-before于其他线程的检测到线程T2中断这一操作

3.5 程序同步

程序必须正确同步，以避免代码重排序时可能出现的异常行为。

正确的同步不能确保程序的整体行为是正确的，但它允许程序员以简单的方式推断程序的可能行为。正确同步的程序的行为更少依赖于可能的重排序。未正确同步的程序可能产生非常奇怪的行为。

程序是否正确同步有两个关键点：

- 冲突访问：如果至少有一个访问是写操作，那么对同一个共享字段或数组元素的两次访问（读取或写入）将被认为是冲突的。
- happen-before关系：两个行为可以按照happen-before关系来排序。

当一个程序包含两个冲突访问，而这些访问并不是满足happen-before关系顺序的时候，据说它包含一个数据竞争。一个正确同步的程序是没有数据竞争的程序。

如果程序未正确同步，则会出现三种类型的问题：可见性，有序性和原子性。

3.5.1 可见性（Visible）

可见性就是指当一个线程修改了线程共享变量的值，其它线程能够立即得知这个修改。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方法来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是volatile的特殊规则保证了新值能立即同步到主内存，以及每使用前立即从内存刷新。因为我们可以说volatile保证了线程操作时变量的可见性，而普通变量则不能保证这一点。

除了volatile之外，Java还有两个关键字能实现可见性，它们是synchronized。同步块的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存中(执行store和write操作)”这条规则获得的，而final关键字的可见性是指：被final修饰的字段是构造器一旦初始化完成，并且构造器没有把“this”引用传递出去，那么在其它线程中就能看见final字段的值。

（可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如volatile int a = 0；之后有一个操作a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就这这个操作同样存在线程安全问题。）

3.5.2 有序性（ordering）

Java内存模型中的程序天然有序性可以总结为一句话：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存中主内存同步延迟”现象。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则来获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

先行发生原则：

如果Java内存模型中所有的有序性都只靠volatile和synchronized来完成，那么有一些操作将会变得很啰嗦，但是我们在编写Java并发代码的时候并没有感觉到这一点，这是因为Java语言中有上述提到的“先行发生”(Happen-Before)的原则。这个原则非常重要，它是判断数据是否存在竞争，线程是否安全的主要依赖。

3.5.3 原子性（ordering）

原子性是指在一个操作中就是cpu不可以在中途暂停然后再调度，既不被中断操作，要不执行完成，要不就不执行。

如果一个操作时原子性的，那么多线程并发的情况下，就不会出现变量被修改的情况

比如a=0；（a非long和double类型）这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++；这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。

非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

（由Java内存模型来直接保证的原子性变量操作包括read、load、use、assign、store和write六个，大致可以认为基础数据类型的访问和读写是具备原子性的。如果应用场景需要一个更大范围的原子性保证，Java内存模型还提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock与unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐匿地使用这两个操作，这两个字节码指令反映到Java代码中就是同步块－synchronized关键字，因此在synchronized块之间的操作也具备原子性。）

4 非形式化语义

4.1 定义

- 共享变量/堆内存：在线程之间共享的内存，所有变量（实例字段，静态字段和数组元素）都存储在堆内存中。方法的本地变量永远不会在线程之间共享。

- 线程间操作：线程间动作是由线程执行的动作，可以被另一个线程检测或直接影响。线程间动作包括读取和写入共享变量和同步动作，例如锁定或解锁监视器，读取或写入易失性变量或启动线程。我们不需要关心线程内动作（例如，添加两个局部变量并将结果存储在第三个局部变量中）。如前所述，所有线程都需要服从Java程序的正确的线程内语义。每个线程间操作都与关于该操作执行的信息相关联;我们将该信息称为注释。所有操作都使用它们所在的线程以及它们在该线程内发生的程序顺序进行注释。一些额外的注释包括：
  - write：准备写的变量和写入的值。
  - read：变量读取和可见的写值（从这里，我们可以确定可见值）。
  - lock：已锁定的监视器。
  - unlock：解锁的监视器。

  为了简洁起见，我们通常将线程间操作简称为操作。
- 程序顺序：在每个线程t执行的所有线程间操作中，t的程序顺序是这些操作执行的总顺序。
- 同步操作：除了读取和写入正常和最终变量以外的所有线程间操作都是同步操作。这些包括锁定，解锁，读取和写入易失性变量，启动线程的动作以及检测线程完成的动作。
- 同步顺序：在任何执行过程中，都有一个同步顺序，它是该执行过程的所有同步操作的总顺序。对于每个线程t，t中的同步动作的同步顺序与t的程序顺序一致。
- Happens-Before（HB）边：如果我们有两个动作x和y，我们使用$x \xrightarrow{hb} y$表示x在y之前。如果x和y是同一个线程的动作，并且x 按照程序顺序在y之前，那么$x \xrightarrow{hb} y$。
  - 同步动作之间若满足happen-before关系，也可以形成HB边。

4.2 规范

4.2.1 hb一致性

满足以下条件时，则我们说写操作w对读操作r可见：

- $r \xrightarrow{hb} w$ 不成立
- 不存在另一个写操作w'；满足$w \xrightarrow{hb} w'$ 和$w' \xrightarrow{hb} r$

4.2.2 动作（actions）和执行（executions）

动作a由元组(t，k，v，u)描述，其中包括：

- t - 执行动作的线程

- k - 类型的动作：易失性读取，易失性写入，（正常或非易失性）读取，（正常或非易失性）写入，锁定或解锁。易失性读取，易失性写入，锁定和解锁是同步操作。
- v - 行动中涉及的变量或监视器
- u - 行为的任意唯一标识符

执行E由元组(P，A，$\xrightarrow{po}$，$\xrightarrow{so}$，W，V，$\xrightarrow{sw}$，$\xrightarrow{hb}$)描述，包括：

- P - 一个程序
- A - 动作集合
- $\xrightarrow{po}$ - 程序顺序，对于某个线程t而言，代表了t所包含的A中动作的总顺序
- $\xrightarrow{so}$ - 同步顺序，A中所有同步动作的总顺序
- W - 可见写函数，对于A中的每个读取动作r，给出W（r）（对执行E中的读取动作r可见的写入动作）。
- V - 值写入的函数，对于A中的每个写入动作w，给出V（w）（E中的w动作的写入值）。
- $\xrightarrow{sw}$[2] - synchronizes-with,同步动作的局部顺序
- $\xrightarrow{hb}$[2] - happens-before,动作的局部顺序

4.2.3 定义

synchronizes-with :如果$x \xrightarrow{so} y$并且x是一个volatile类型的写动作或是一个解锁动作，y是对同一个volatile类型变量的读动作或是对同一个监视器的加锁动作，那么有$x \xrightarrow{sw} y$。volatile write和unlock称为release（释放），volatile read和lock称为acquire（请求）。

happens-before ：$\xrightarrow{hb}$相当于$\xrightarrow{sw} \cup \xrightarrow{po}$

Restrictions of partial orders and functions ：用$f\mid_d$表示将f域限制到d给出的函数。对于所有的$x \in d$，有$f(x) = f\mid_d(x)$，对于所有$x \notin d$，有$f(x) = \perp$。类似的，用$\xrightarrow{e}\mid_d$表示将局部顺序$\xrightarrow{e}$限制到只针对d中的元素：对于所有$x, y \in d$，当且仅当$x \xrightarrow{e}\mid_d y$时，有$x \xrightarrow{e} y$。当然，要是$x \notin d$或$y \notin d$，就不是$x \xrightarrow{e}\mid_d y$了。

4.2.4 完备的执行过程（well-form execution）

一个执行过程E满足下列条件时，是一个完备的执行过程：

[2]请注意，synchronizes-with和happen-before是由执行的其他组件和格式良好的执行规则唯一确定的。

- 每个read动作都可见一个执行过程中的write动作，所有volatile类型的read看见的是volatile类型的write，所有非volatile类型的read看见的是非volatile类型的write。(对于$r \in A$，有$W(r) \in A$，$W(r).v \in r.v$。若r.k是volatile read，W(r).k就是volatile write；若r.k是non-volatile read，W(r).k就是non-volatile write)
- 同步顺序与程序顺序一致。不存在$x, y \in A$，使得$x \xrightarrow{so} y \wedge y \xrightarrow{po} x$。同步顺序和程序顺序的传递闭包是不循环的。
- 执行过程要遵循单线程内的一致性。对于每个线程t，A中由t执行的动作与t在程序中独立生成的行为相同，每个写动作w写入值V(w)，每个读动作r看见值V(W(r))，t的程序顺序必须反映P(总程序)的程序顺序
- 执行过程要遵循happens-before的一致性。对于$r \in A$，不存在$r \xrightarrow{so} W(r)$或$\exists w \in A, w.v = r.v \wedge W(r) \xrightarrow{hb} w \xrightarrow{hb} r$
- 执行过程要遵循同步顺序的一致性。对于volatile类型read$r \in A$,不存在$r \xrightarrow{so} W(r)$或$\exists w \in A, w.v = r.v \wedge W(r) \xrightarrow{so} w \xrightarrow{so} r$
- 执行过程要为同步动作保持一个微弱的公平性。在同步顺序中发生给定同步操作之前，可能只发生有限次数的同步操作。

4.2.5 根据JMM的有效执行

一个完备的执行过程(P，A，$\xrightarrow{po}$，$\xrightarrow{so}$，W，V，$\xrightarrow{sw}$，$\xrightarrow{hb}$)通过A的动作来验证有效性。如果A中的所有动作都被认可提交，那么称该执行是基于Java内存模型有效执行。

从空集C0开始，我们模拟从A中取几个动作，加入到commited动作集C_i中来获得新的commited动作集$C_i + 1$。为了证明这是合理的，对于每个动作集C_i，我们需要证明包含C_i的执行E满足哪几个条件。

设有动作集C_0，C_1，C_2……C_n满足：

- C_0=
- $C_i \subseteq C_i + 1$
- C_n = A

有完备的执行过程E_0，E_1，E_2……E_n满足E_i = (P，A_i，$\xrightarrow{po_i}$，$\xrightarrow{so_i}$，W_i，V_i，$\xrightarrow{sw_i}$，$\xrightarrow{hb_i}$)

假定C_i中的每个动作都属于执行过程E_i，C_i中的所有动作在E_i和E中都共享相同的happens-before顺序和同步顺序关系。形式化：

1 $C_i \subseteq A_i$

2 $\xrightarrow{hb_i}|_{C_i} = \xrightarrow{hb}|_{C_i}$

3 $\xrightarrow{so_i}|_{C_i} = \xrightarrow{so}|_{C_i}$

每个C_i中的write动作在执行过程E和E_i写的值必须相同。形式化：

4 $V_i \mid_{C_i} = V \mid_{C_i}$

5 $W_i \mid_{C_i-1} = W \mid_{C_i-1}$

E_i中所有不在$C_i - 1$中的读取r，必须看见所有满足happen-before r的写动作write。所有C_i-$C_i - 1$中的读取在执行过程E和E_i中看见的write写动作必须属于$C_i - 1$。形式化：

6 对任意$r \in A_i - C_{i-1}$ 有$W_i(r) \xrightarrow{hb_i} r$

7 对任意$r \in C_i - C_{i-1}$ 有$W_i(r) \in C_{i-1}$ 且$W(r) \in C_{i-1}$

和程序顺序边一起，就足够用来决定程序中所有happen-before顺序边的最小同步顺序边集合我们称它为sufficient（足够）的。这个集合是惟一的。

给定执行过程E_i的一个sufficient同步顺序边集，如果有一个释放（release）-请求（acquire）动作对，与正在提交（commiting）的动作满足happen-before，那么这个动作对一定出现在所有执行过程F_j(j≥i),形式化：

8 令$\xrightarrow{ssw_i}$为$\xrightarrow{sw_i}$中满足在$\xrightarrow{hb_i}$ 子集中但不在$\xrightarrow{po_i}$中的边集
则$\xrightarrow{ssw_i}$为执行过程E_i的最小同步顺序边集。
若$x \xrightarrow{ssw_i} y \xrightarrow{hb_i} z$且$z \in C_i$,那么，对$\forall j \geq i$有$x \xrightarrow{sw_j} y$

4.3 final字段的语义

在java里面，如果一个类定义了一个final属性，那么这个属性在初始化之后就不可以在改变。一般认为final字段是不变的。在java内存模型里面，对final有一个特殊的处理。如果一个类C定义了一个非static的final属性A，以及非static final属性B，在C的构造器里面对A，B进行初始化，如果一个线程T1创建了类C的一个对象co，同一时刻线程T2访问co对象的A和B属性，如果t2获取到已经构造完成的co对象，那么属性A的值是可以确定的，属性B的值可能还未初始化

下面一段代码演示了这个情况

- public class FinalVarClass
- public final int a ;
- public int b = 0;
- static FinalVarClass co;
- public FinalVarClass()
- a = 1;
- b = 1;

- //线程1创建FinalVarClass对象co

- public static void create()
- if(co == null)
- co = new FinalVarClass();

- //线程2访问co对象的a，b属性
- public static void vistor()
- if(co != null)
- System.out.println(co.a);//这里返回的一定是1,a一定初始化完成
- System.out.println(co.b);//这里返回的可能是0，因为b还未初始化完成

发生这种情况的原因可能是处理器对创建对象的指令进行重新排序。正常情况下，对象创建语句co = new FinalVarClass()并不是原子的，简单来说，可以分为几个步骤，1 分配内存空间2 创建空的对象3 初始化空的对象4 把初始化完成的对象引用指向co ，由于这几个步骤处理器可能并发执行，比如3，4 并发执行，所以在create操作完成之后，co不一定马上初始化完成，所以在vistor方法的时候，b的值可能还未初始化。但是如果是final字段，必须保证在对应返回引用之前初始化完成。

4.4 long和double变量的特殊规则

允许虚拟机实现选择可以不保证64位数据类型的load，read，store，write这四个操作的原子性，即long和double的非原子性协定(备注:实际开发中，目前个平台下的商用虚拟机几乎都选择把64位数据的读写操作作为原子操作来对待，不需要将long和double类型变量专门声明为volatile)

4.5 Volatile

当一个变量被定义为volatile之后，将具备两种特性，第一是保证此变量对所有线程的可见性（这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的）

Volatile 使用原则:

运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值;

变量不需要与其他的状态变量共同参与不变约束;

满足上述规则的运算场景，使用volatile可以不用通过加锁来保证原子性

使用volatile变量的第二个语义是禁止指令重排序(volatile屏蔽指令重排序的语义在JDK1.5之后才被完全修复，此前的JDK中，即使将变量声明为volatile也仍然不能完全避免重排序所导致的问题（主要是在volatile变量前后的代码仍然存在重排序的问题），这也是JDK1.5之前的Java中无法安全的使用DCL来实现单例的原因)

References

[1] Making the Java Memory Model Safe ANDREAS LOCHBIHLER, Karlsruhe Institute of Technology

[2] JSR-133: JavaTM Memory Model and Thread Specification ,Proposed Final Draft

[3] Fixing the Java Memory Model ,William Pugh

[4] The Java Memory Model ,Jeremy Manson and William Pugh , Sarita V. Adve

[5] 其他网络（论坛、博客）来源