《深入理解 Java 内存模型》读书总结

概 要

文章是《深入理解 Java 内容模型》读书笔记，该书总共包括了 3 部分的知识。

第 1 部分，**基本概念**

包括“并发、同步、主内存、本地内存、重排序、内存屏障、happens before 规则、as-if-serial 规则、数据依赖性、顺序一致性模型、JMM 的含义和意义”。

第 2 部分，**同步机制**

该部分中就介绍了“同步”的 3 种方式：volatile、锁、final。对于每一种方式，从该方式的“特性”、“建立的 happens before 关系”、“对应的内存语义”、“实现方式”等几个方面进行了分析说明。实际上，JMM 保证“如果程序正确同步，则执行结果与顺序一致性内存模型的结果相同”的机制；而这部分这是确保程序正确同步的机制。

第 3 部分，**JMM 总结**

第 1 部分 基本概念

1. 并发

定义：即，并发(同时)发生。在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行，但任一个时刻点上只有一个程序在处理机上运行。

并发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。**

(01) 通信 —— 是指线程之间如何交换信息。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。

(02) 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 Java 中，可以通过 volatile，synchronized, 锁等方式实现同步。

2.主内存和本地内存

主内存 —— 即 *main memory*。在 java 中，实例域、静态域和数组元素是线程之间共享的数据，它们存储在**主内存**中。

本地内存 —— 即 *local memory*。 局部变量，方法定义参数 和 异常处理器参数是不会在线程之间共享的，它们存储在线程的**本地内存**中。

3.重排序

定义：重排序是指“编译器和处理器”为了提高性能，而在程序执行时会对程序进行的重排序。

说明：重排序分为——“编译器”和“处理器”两个方面，而“处理器”重排序又包括“指令级重排序”和“内存的重排序”。

关于重排序，我们需要理解它的思想：为了提高程序的并发度，从而提高性能！但是对于多线程程序，重排序可能会导致程序执行的结果不是我们需要的结果！因此，就需要我们通过“volatile，synchronize，锁等方式”作出正确的实现同步。

4.内存屏障

定义：包括 LoadLoad, LoadStore, StoreLoad, StoreStore 共 4 种内存屏障。内存屏障是与相应的内存重排序相对应的。

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。

作用：通过内存屏障可以禁止特定类型处理器的重排序，从而让程序按我们预想的流程去执行。

5. happens-before

定义：JDK5(JSR-133)提供的概念，用于描述多线程操作之间的内存可见性。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存在 happens-before 关系。

作用：描述多线程操作之间的内存可见性。

[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。

6. 数据依赖性

定义：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。

作用：编译器和处理器不会对“存在数据依赖关系的两个操作”执行重排序。

7.as-if-serial

定义：不管怎么重排序，程序的执行结果不能被改变。

8. 顺序一致性内存模型

定义：它是理想化的内存模型。有以下规则：

(01) 一个线程中的所有操作必须按照程序的顺序来执行。

(02) 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。

9. JMM

定义：Java Memory Mode，即 *Java 内存模型*。它是 Java 线程之间通信的控制机制。

说明：JMM 对 Java 程序作出保证——如果程序是正确同步的，程序的执行将具有顺序一致性。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。

10. 可见性

可见性一般用于指不同线程之间的数据是否可见。

在 java 中， 实例域、静态域和数组元素这些数据是线程之间共享的数据，它们存储在**主内存**中；主内存中的所有数据对该内存中的线程都是可见的。而局部变量，方法定义参数 和 异常处理器参数这些数据是不会在线程之间共享的，它们存储在线程的**本地内存**中；它们对其它线程是不可见的。

此外，对于主内存中的数据，在本地内存中会对应的创建该数据的副本(相当于缓冲)；这些副本对于其它线程也是不可见的。

11. 原子性

是指一个操作是按原子的方式执行的。要么该操作不被执行；要么以原子方式执行，即执行过程中不会被其它线程中断。

第 2 部分 同步机制

1.volatile

1.1 作用

如果一个变量是 volatile 类型，则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。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。

1.2 volatile 的内存语义

volatile 写：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。

volatile 读：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。

1.3 JMM 中的实现方式

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

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。

1.4 volatile 和 synchronize 对比

在功能上，监视器锁比 volatile 更强大；在可伸缩性和执行性能上，volatile 更有优势。

volatile 仅仅保证对单个 volatile 变量的读/写具有原子性；而 synchronize 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。

2.锁

2.1 作用

锁是 java 并发编程中最重要的同步机制。

2.2 锁的内存语义

(01) 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。

(02) 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。

(03) 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。

2.3 JMM 如何实现锁

公平锁

公平锁是通过“volatile”实现同步的。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。

非公平锁

通过 CAS 实现的，CAS 就是 compare and swap。CAS 实际上调用的 JNI 函数，也就是 CAS 依赖于本地实现。以 Intel 来说，对于 CAS 的 JNI 实现函数，它保证：(01)禁止该 CAS 之前和之后的读和写指令重排序。(02)把写缓冲区中的所有数据刷新到内存中。

3.final

3.1 特性

对于**基本类型**的 final 域，编译器和处理器要遵守两个重排序规则：

(01) final 写：“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。

(02) final 读：“初次读一个包含 final 域的对象的引用”，与“随后初次读对象的 final 域”，这两个操作之间不能重排序。

对于**引用类型**的 final 域，除上面两条之外，还有一条规则：

(03) final 写：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。

注意：

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”。

3.2 JMM 如何实现 final

通过“内存屏障”实现。

在 final 域的写之后，构造函数 return 之前，插入一个 StoreStore 障屏。在读 final 域的操作前面插入一个 LoadLoad 屏障。

第 3 部分 JMM 总结

JMM 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 。

JMM 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：

(01) 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(程序尽可能的顺序执行)来编写代码。

(02) 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(对程序重排序，做尽可能多的并发)来提高性能。编译器和处理器希望实现一个弱内存模型。

JMM 设计就需要在这两者之间作出协调。JMM 对程序采取了不同的策略：

(01) 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。

(02) 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）。