java并发编程笔记--Java内存模型

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | | 性能分析工具：Lmbench3  linux监控工具：vmstat  jvm线程分析：jstack 进程号 |   Reference：参考文献   |  | | --- | | [分布式中的一致性模型](http://blog.sina.com.cn/s/blog_3fe961ae0101g24g.html) | |

目录

[1 Java内存模型 3](#_Toc453679695)

[1.1 示例1：没有进行正确同步的程序可能产生非预期结果 3](#_Toc453679696)

[2 JSR-133 FAQ 3](#_Toc453679697)

[2.1 什么是内存模型？ 3](#_Toc453679698)

[2.2 其它语言比如c++是否也提供了内存模型？ 3](#_Toc453679699)

[2.3 什么是JSR-133? 3](#_Toc453679700)

[3 运维经验 4](#_Toc453679701)

# Java内存模型基本概念

## 并发编程模型的两个关键问题

1. 线程之间如何通信；（内存共享&消息传递，Java的并发采用共享内存模型）
2. 线程之间如何同步；

## 内存模型抽象结构

|  |
| --- |
| 线程的同步通过JMM进行控制 |

|  |
| --- |
| 内存访问的同步过程，可以看做是线程的通信过程 |

1. JMM，java内存模型，主要用来保证共享内存的可见性；
2. Java内存模型可以分为共享内存和线程私有内存两个部分；其中，局部变量，函数参数，异常对象参数等都属于线程私有内存，不存在内存可见性问题；实例域、静态域和数组元素存放在堆中，属于线程共享内存；
3. 多个线程对共享内存的读写可以看做是线程之间的间接通信过程；
4. JMM通过控制主内存&每个线程的本地内存的交互，来保证共享内存可见性；

|  |
| --- |
| 结论：不存在共享内存的多线程程序是安全的；（Immutable） |

## 问题原因：重排序

|  |
| --- |
| 源码到实际运行指令，可能经过的重排序流程 |

1. **重排序**：是编译器或者处理器为了优化程序性能而对指令序列进行重新排序的操作；
2. 重排序主要发生在编译、执行两个过程中，分别由编译器和处理器控制；可以分为3种类型：
   1. 编译器优化重排序：编译器在不改变单线程程序语义的前提下，调整优化代码结构；
   2. 指令级并行重排序：为满足ILP（Instruction-Level-Parallelism，指令级并行技术），可以同时执行多条指令，处理器对不存在数据依赖的指令进行重排序，改变指令的执行顺序；
   3. 内存系统重排序：内存&读写缓冲区&高速缓存之间的数据读写可能改变执行顺序；
3. JMM通过禁止特定类型的重排序，来保证内存可见性；
   1. 编译器重排序：引入happens-before规则，禁止编译器进行特定类型的重排序；
   2. 处理器重排序：在生成的指令序列中，（编译器）插入特定类型的内存屏障指令，通过内存屏障指令禁止处理器进行特定类型的重排序；

|  |
| --- |
| 观点：缓存的引入，提高了数据访问效率，但也引入了读写数据不一致的可能性； |

## 控制处理器重排序：内存屏障

1. **内存屏障：**用于提醒处理器禁止特定类型重排序的指令；内存屏障指令用来实现对内存操作顺序的控制；常见内存屏障类型如下：
2. 大多数处理器都实现了StoreLoad内存屏障指令，因为大多数处理器都有写缓存，为保证读取数据的正确性，需要通过StoreLoad指令保证读操作之前，所有数据已写入内存；
3. StoreLoad为全能型指令，能够实现其余类型指令的功能；但指令开销较大，需要将写缓冲区中的数据全部写入内存；
4. 常见的内存屏障类型包括如下几种类型：

|  |  |  |  |
| --- | --- | --- | --- |
| **内存屏障类型** | **重排序类型** | **指令示例** | **语义** |
| LoadLoad | 读-读 | Load1;  LoadLoad;  Load2; | 确保Load1数据的读取先于Load2及其后所有数据的读取； |
| StoreStore | 写-写 | Store1;  StoreStore;  Store2; | 确保Store1数据的写入先于Store2及其后所有数据的写入； |
| LoadStore | 读-写 | Load1;  LoadStore;  Load2; | 确保Load1数据的读取先于Store2及其后续所有数据的写入； |
| StoreLoad | 写-读 | Store1;  StoreLoad;  Load2; | 1. 确保Store1数据的写入先于Load2及其后续所有数据的读取； 2. 等待Store1及其之前所有读写指令执行完成后，再执行Load2及其之后的指令；（保证了StoreLoad指令能够具备前三种指令的功能） |

|  |
| --- |
| 观点：内存屏障是Java控制处理器重排序的底层实现机制； |

## 控制编译器重排序：happens-before规则

|  |
| --- |
| happens-before规则示意 |

**happens-before规则：**编译器重排序，通过约定一系列规则来进行控制，这些规则称为happens-before规则。JSR-133通过确定一些列happens-before规则来确保一个操作结果需要对另一个操作可见；常见的happens-before规则如下：

|  |
| --- |
| 1. 程序顺序规则：一个线程中的每个操作，happens-before于该线程的任意后续操作； 2. 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁； 3. 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； 4. 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C； |

|  |
| --- |
| 注：对happens-before的理解；  happens-before规则并不要求前一个操作一定要在后一个操作之前执行，仅要求签一个操作的结果对后一个操作可见；happens-before规则简化了对内存可见性控制的理解； |

|  |
| --- |
| 观点：happens-before规则是Java控制编译器重排序的底层实现机制； |

## 单线程程序重排序约束：as-if-serial语义

1. 数据依赖：如果单个线程中的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，此时这两个操作之前就存在数据依赖性。
2. as-if-serail语义：不管怎么重排序，单线程程序的执行结果不能被改变；
3. 控制依赖（判断-执行）在单线程中重排序不会改变执行结果，但在多线程中重排序可能会改变执行结果；（JMM对此无约束）

|  |
| --- |
| 观点：在计算机中，软件技术和硬件技术有一个共同目标：在不改变程序执行结果的前提下，尽可能提高并行度；所以happens-before规则的限制仅是对是否改变单线程程序执行结果的限制，而不限制具体的实现过程； |

## 数据一致性保证：顺序一致性模型

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 知识点：分布式中的一致性模型  进行数据复制主要出于两个目的：可靠性和性能。数据一旦被复制，就会带来一致性的问题。  **以数据为中心的一致性模型**  **1.严格一致性（strict consistency）**  1）对于数据项x的任何读操作将返回最近一次对x进行写操作的结果所对应的值；  2）严格一致性是限制性最强的模型，但是在分布式系统中实现这种模型代价太大，所以在实际系统中运用有限；    **2.顺序一致性**  1）任何执行结果都是相同的，就好像所有进程对数据存储的读、写操作是按某种序列顺序执行的，并且每个进程的操作按照程序所制定的顺序出现在这个序列中；  2）任何读、写操作的交叉都是可接受的，但是所有进程都看到相同的操作交叉。顺序一致性由Lamport（1979）在解决多处理器系统的共享存储器时首次提出的；    **3.因果一致性**  1）所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上的进程可以以不同的顺序看到并发的写操作（Hutto和Ahamad 1990）；  2）假设P1和P2是有因果关系的两个进程，例如P2的写操作信赖于P1的写操作，那么P1和P2对x的修改顺序，在P3和P4看来一定是一样的。但如果P1和P2没有关系，那么P1和P2对x的修改顺序，在P3和P4看来可以是不一样的。    下表列出一些一致性模型，以限制性逐渐降低的顺序排列。   |  |  | | --- | --- | | 一致性 | 描述 | | 严格 | 所有共享访问按绝对时间排序 | | 线性化 | 所有进程以相同的顺序看到所有的共享访问。而且，访问是根据（非唯一的）全局时间戮排序的 | | 顺序 | 所有进程以相同顺序看到所有的共享访问。访问不是按时间排序的 | | 因果 | 所有的进程以相同的顺序看到困果相关的共享访问 | | FIFO | 所有进程以不同的进程提出写操作的顺序相互看到写操作。来自不同的进程的写操作可以不必总是以同样的顺序出现 |     另一种不同的方式是引入显式的同步变量，下表是这样的一致性模型的总结。   |  |  | | --- | --- | | 一致性 | 描述 | | 弱 | 只有在执行一次同步后，共享数据才被认为是一致的 | | 释放 | 退出临界区时，使共享数据成为一致 | | 入口 | 进入临界区时，使属于一个临界区的共享数据成为一致 |     **以客户为中心的一致性模型**  **1.最终一致性**  最终一致性指的是在一段时间内没有数据更新操作的话，那么所有的副本将逐渐成为一致的。例如OpenStack Swift就是采用这种模型。以一次写多次读的情况下，这种模型可以工作得比较好；    **2.单调读**  如果一个进程读取数据项x的值，那么该进程对x执行的任何后续读操作将总是得到第一次读取的那个值或更新的值；    **3.单调写**  一个进程对数据x执行的写操作必须在该进程对x执行任何后续写操作之前完成；    **4.写后读**  一个进程对数据x执行一次写操作的结果总是会被该进程对x执行的后续读操作看见；    **5.读后写**  同一个进程对数据项x执行的读操作之后的写操作，保证发生在与x读取值相同或比之更新的值上； |

# JSR-133 FAQ

## 什么是内存模型？

在多处理器环境下，处理器通常会有一层或者多层高速缓存；高速缓存能够提升数据访问的性能，因为数据离处理器更”近”了。同时，也可以降低共享内存总线的流量压力，因为许多数据操作可以通过高速缓存进行，减少不必要的内存访问；高速缓存显著的提升了计算性能，同时也引入了新的问题：如何保障多处理器访问共享数据的原子性，可见性，有序性；比如：当两个处理器同时访问相同内存位置的数据，将如何处理？在什么条件下，多个处理器可以看到相同的数据？

在处理器层面，内存模型定义了必要的条件确保处理器之间共享数据的可见性；有些处理器使用”强内存模型”，即同一时间，不管共享数据当前处于哪个位置，都可以被其它处理器看到；有些处理器使用”弱内存模型”，即使用称为”内存屏障”的特殊指令，来清理高速缓存中的数据和控制高速缓存数据失效，用以确保当前处理器可以读到其它处理器的写，或者当前处理器的写可以被其它处理器读到；这些内存屏障通常通过”加锁”和”解锁”实现，并且对于较高层次的语言（不能直接访问硬件的语言）是不可见的；

基于”强内存模型”的编程，将更加容易，因为可以减少内存屏障的使用，但即使在”强内存模型”中，有时内存屏障也是必要的；目前的处理器设计倾向于使用”弱内存模型”，因为对高速缓存一致性的宽松限制，能够使增强多处理器的可伸缩性，同时能够使用更大的内存；

一个线程的写在什么时候对其它线程可见，这取决于编译器的代码重排序；比如：编译器优化代码时，可能认为将一个写操作放到后面执行将更有效率；只要重排序不影响程序执行的语义（即：程序执行的结果不会因为代码执行顺序的改变而不同）；

java对并发编程提供了语言层面的支持，包括volitile、synchronized、final等，同时确保程序在不同的处理器上都可以正确运行；

## 其它语言比如c++是否也提供了内存模型？

c/c++并没有提供语言层面的多线程支持，而是通过第三方库比如pthread来保障线程的同步；

## 什么是JSR-133?

Java内存模型，是一个用来在语言层面提供对跨平台并发编程提供一致性语义；JSR-133定义了一个新的内存模型，用来修复旧内存模型的缺陷；为确保语义的一致性，final和volitile关键字的语义需要改变；