Java 并发底层硬件原理探究 🧠

- 0️⃣ 何为JMM？
  - 不同硬件架构有不同的内存模型
  - JMM是在硬件内存模型之上的一层抽象
    - 定义了一系列内存访问的规则（可见性、有序性）
    - 屏蔽了底层不同硬件架构的内存访问差异
  - 面对JMM编程，而无需关注底层硬件内存模型，实现"一次编写，到处运行"
- 1️⃣ 为何学习JMM要了解一定的硬件知识
  - 硬件层面复杂性
    - 硬件层面出于性能考虑会进行数据缓存、指令重排序等操作
    - 影响可见性、有序性
  - Java 开发者为何需要了解硬件
- 2️⃣ CPU多级缓存架构
  - 缓存的原理 — 时间/空间局部性原理，提升访问性能
  - 组成
    - L1/L2/L3 Cache
    - 内存
  - Cache缓存的单位 — "缓存行"
  - 伪共享问题 ⚠️ — 多个CPU核心共享同一个缓存行，即使各自修改不同的变量也会导致对方缓存行失效
  - 多核缓存一致性挑战
    - 缓存一致性问题 ❓ — 当多个核心同时操作共享数据时，如何保证所有核心看到的都是这份数据的最新、最一致的版本
    - MESI 协议 🔄
      - 状态
        - M（Modified）
        - E（Exclusive）
        - S（Shared）
        - I（Invalid）
      - 核心思路 — 一个核心修改数据会通过总线告知其他核心，其他核心将副本置为失效，从而避免读到过期数据
    - 几乎所有现代通用CPU 都保证缓存一致性 ✅
  - Store Buffer 与有序性挑战
    - Store Buffer 作用
      - 相当于是CPU核心和L1 Cache之间的又一层缓存
      - 用于缓存写(Store)操作，减少写操作延迟
      - 后台异步提交到L1 Cache，提交时可以对写操作进行合并
    - Store Buffer 清空时机
      - 持续尝试清空
      - Store Buffer满
      - 特定指令执行
        - 内存屏障（x86 MFENSE/SFENCE）
        - LOCK前缀指令（x86）
        - ...
    - Store Buffer 的副作用
      - 延迟可见性
        - 对当前核心可见 — ✨ Store Forwarding
          - ✅ 大多数CPU都支持
          - 当前CPU核心可以从Store Buffer读取未刷到L1的修改
        - 对其他核心不可见 — 暂存到Store Buffer的写操作不参与缓存一致性协议，在刷到L1 Cache前对其他核心不可见
      - 有序性
        - 某些架构的CPU中，Store Buffer中的内容不按顺序刷到L1 Cache
        - ✅ x86架构不存在有序性问题
- 3️⃣ 指令重排序
  - 为什么要重排序 ❓ — 提升性能 🚀
  - 重排序发生的地方 🏞️
    - 编译器重排序
    - CPU重排序
  - 重排序的依据
    - 遵循as-if-serial语义 — 单线程重排序后执行结果和不重排序结果相同
    - 遵循内存模型的规定
  - 带来的问题 — 有序性
- 4️⃣ 硬件内存模型
  - tradeoff — 硬件层面优化空间 vs 软件层面并发程序易编程性
  - 顺序一致性模型（SC）❌
    - 特点 🌟
      - 一个线程中的所有操作必须按照程序的顺序来执行
      - 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。
    - 缺点 👎
      - 限制了很多硬件层面的优化空间，性能差
      - 实际硬件架构中没有用该模型的 ❌
  - 放松的一致性模型 ✅
    - 特点 🌟
      - 放松一些一致性要求，例如允许某些类型的重排序 🔄
      - 给硬件设计者更多优化空间，提升性能 🚀
      - 主流CPU架构多采用放松的一致性模型 ✅
    - 代表 💼
      - x86 TSO 模型 🔥
        - 仅允许StoreLoad重排序 — why? — 为了enable Store Buffer优化。如果后面的Load和前面的Store不是访问同一个内存地址，则后面的Load可以直接从缓存读取，无需等待前面的Store刷到L1缓存
        - 不允许StoreStore/LoadLoad/LoadStore重排序
      - ARM 弱内存模型 🌪️
  - 对JMM实现的影响
- 5️⃣ 内存屏障 Memory Barrier 🧱
  - 内存屏障的作用 — 显式控制有序性和可见性 🎮
  - 类型分类
    - 编译器内存屏障 — 阻止编译器的重排序
    - 硬件内存屏障(以x86为例)
      - 完全内存屏障
        - 指令 — MFENCE/LOCK前缀/XCHG
        - 作用
          - 有序性 — 阻止向屏障之前/之后的重排序
          - 可见性 — 强制Store Buffer排空，确保屏障之前的写入对屏障之后可见
      - 写内存屏障
        - 指令 — SFENCE
        - 作用
          - 有序性 — 限制SFENCE前后的写指令重排序；不过x86本身就不会发生StoreStore重排序，因此这类指令的作用主要体现在可见性
          - 可见性 — 强制Store Buffer排空，确保屏障之前的写入对屏障之后可见
      - 读内存屏障
        - 指令 — LFENCE
- 6️⃣ 硬件对于JMM实现的影响
  - JVM在不同硬件平台上实现同步原语（volatile、synchronized等）时，必须替我们屏蔽底层的差异，根据底层硬件一致性模型，插入合适的内存屏障指令，从而对外提供一个统一的JMM模型。