

Java内存模型—JMM

2022年10月23日 20:53

首先：JVM和JMM是两个截然不同的东西

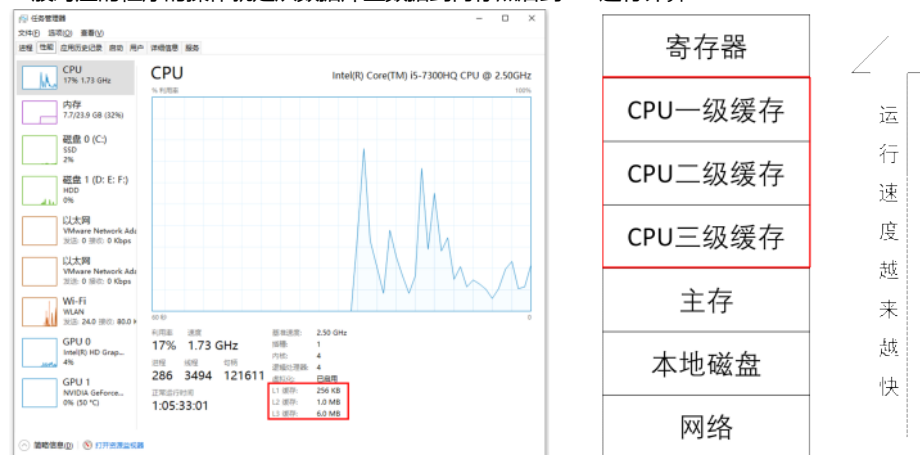
大厂面试题：

1. 你知道什么是Java内存模型JMM吗？
2. JMM与volatile它们两个之间的关系？
3. JMM有哪些特性 or 它的三大特性是什么？
4. 为什么要有JMM，它为什么出现？作用和功能是什么？
5. happens-before先行发生原则有了解过吗？

1、计算机硬件存储体系

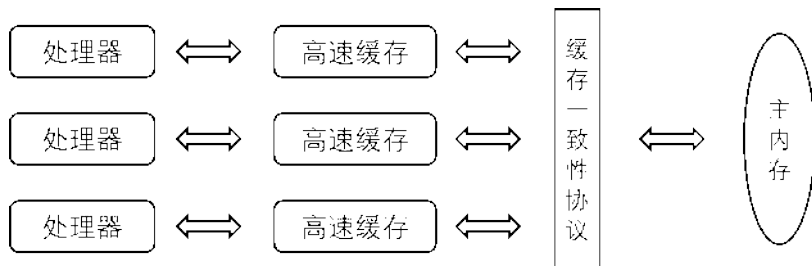
计算机存储结构，从本地磁盘到主存到CPU缓存，也就是从硬盘到内存，到CPU

一般对应的程序的操作就是从数据库查数据到内存然后到CPU进行计算



CPU和物理主内存的速度是不一致的，所以会有这么多级缓存

CPU的运行并不是直接操作内存而是先把内存里边的数据读到缓存，而内存的读和写操作的时候就会造成数据不一致的问题



处理器、高速缓存、主内存之间的交互关系

JVM规范中试图定义一种Java内存模型（Java Memory Model，简称JMM）来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。

2、Java内存模型Java Memory Model

JMM（Java内存模型Java Memory Model，简称JMM）本身是一种抽象的概念并不真实存在，它仅仅描述的是一组约定或规范，通过这组规范定义了程序中（尤其是多线程）各个变量的读写访问方式并决定一个线程对共享变量的写入，何时以及如何变成对另一个线程可见。关键技术点都是围绕多线程的原子性、可见性和有序性展开的。

原则：JMM的关键技术点都是围绕原子性、可见性和有序性展开的

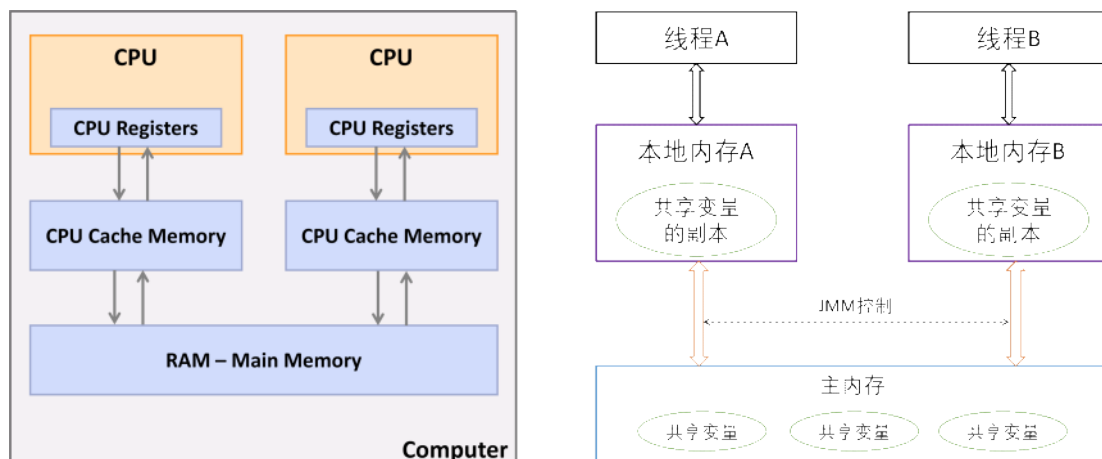
JMM能干啥？

1. 通过JMM来实现线程和主内存之间的抽象关系。
2. 屏蔽各个硬件平台和操作系统的内存访问差异以实现让Java程序在各种平台下都能达到一致的内存访问效果。

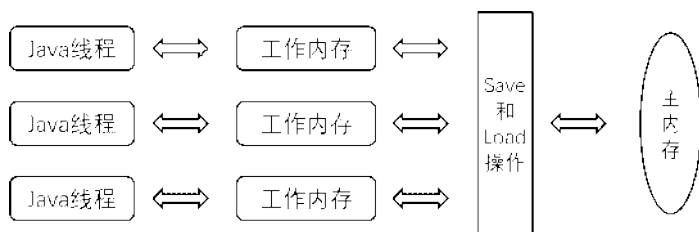
3、JMM规范下，三大特性

(1) 可见性

是指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道该变更，JMM规定了所有的变量都存储在**主内存**中。



系统主内存**共享变量**数据修改被写入的时机是不确定的，**多线程并发下很可能出现“脏读”**，所以每个线程都有自己的**工作内存**，线程自己的工作内存中保存了该线程使用到的变量的**主内存副本拷贝**，线程对变量的所有操作（读取，赋值等）都必须在线程自己的工作内存中进行，而不能够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。



线程、主内存、工作内存三者的交互关系

线程脏读举例：

1. 主内存有变量x，初始值位0
2. 线程A要将x加1，先将x=0拷贝到自己的私有内存中，然后更新x的值
3. 线程A将更新后的x值刷到主内存的时间是不固定的
4. 刚好在线程A没有回刷x到主内存时，线程B同样从主内存中读取x，此时为0，和线程A一样的操作，最后期盼的x=2就会变成x=1

(2) 原子性

指一个操作是不可打断的，及多线程环境下，操作不能被其他线程干扰。

(3) 有序性

对于一个线程的执行代码而言，我们总是习惯性的认为代码的执行总是从上到下，有序执行。但为了提升性能，编译器和处理器通常会对指令序列进行**重新排序**。

Java规范指定JVM线程内部维持**顺序化语义**，即只要程序的最终结果与它顺序化执行的结果相等，那么指令的执行顺序可以与代码顺序不一致，此过程叫**指令的重排序**。

优缺点：

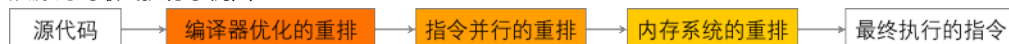
JVM能根据处理器特性（CPU多级缓存系统、多核处理器等）适当的对机器指令进行重排序，使机器指令更符合CPU的执行特性，最大限度地发挥机器性能。

但是，

指令重排可以保证串行语义一致，但**无法保证多线程间的语义也一致**（即可能产生“脏读”）

两行以上不相干的代码在执行的时候有可能先执行的不是第一条，**不见得是从上到下顺序执行，执行顺序会被优化。**

从源码到最终执行示例图：



```
public void mySort {
    int x = 11;    // 语句 1
    int y = 12;    // 语句 2
    x = x + 5;     // 语句 3
    y = x * x;     // 语句 4
}
```

以上四条语句可以重排序为：1234、2134、1324

问题：请问语句4可以重排后变成第一个执行吗？

答：不可以！要符合数据依赖性！

单线程环境里面确定程序最终执行结果和代码顺序执行的结果一致。

处理器在进行重排序时必须要考虑指令之间的数据依赖性

多线程环境中线程交替执行，由于编译器优化重排的存在，两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。

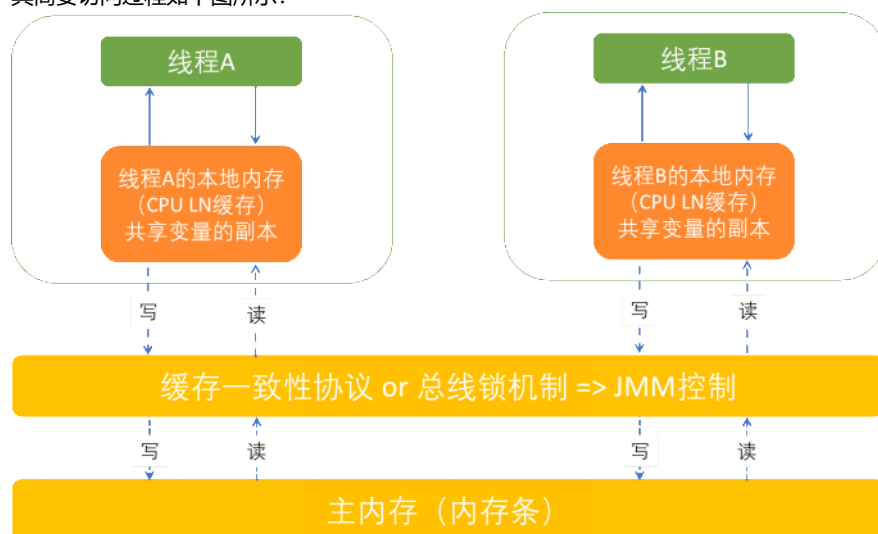
4、JMM规范下，多线程读变量的读写过程

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存（有些地方称为栈空间），工作内存是每个线程的私有数据区域。

而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，**但线程对变量的操作（读取赋值等）必须在工作内存中进行**，

首先要将变量从主内存拷贝到线程自己的工作内存空间，然后对变量进行操作，操作完成之后再将变量写回主内存，不能直接操作主内存中的变量。

各个线程中的工作内存中存储着主内存中的变量副本拷贝，因此不同的线程间无法访问对方的工作内存，线程间的通信（传值）必须通过主内存来完成其简要访问过程如下图所示：



JMM定义了线程和主内存之间的抽象关系

1. 线程之间的共享变量存储在主内存中（从硬件角度来说就是内存条）
2. 每个线程都有一个私有的本地工作内存，本地工作内存中存储了该线程用来读/写共享变量的副本（从硬件角度来说就是CPU的缓存，比如寄存器、L1/2/3缓存等）

小总结：

1. 定义的所有共享变量都存储在物理主内存中
2. 每个线程都有自己独立的工作内存，里面保存该线程使用到的变量的副本（主内存中该变量的一份拷贝）
3. 线程对共享变量所有的操作都必须先在线程自己的工作内存中进行后写回主内存，不能直接从主内存中读写（不能越级）
4. 不同线程之间也无法直接访问其他线程的工作内存中的变量，线程间变量值的传递需要通过主内存来进行（同级不能相互访问）

5、JMM规范下，多线程先行发生原则之happens-before

在JMM中，如果一个操作执行的结果需要对另一个操作具有可见性，或者代码重排序。

那么这两个操作之间必须存在happens-before(先行发生)原则。

(1) x、y案例说明

x = 5	线程A执行
y = x	线程B执行

上述操作称之为：写后读。问题：y是否等于5呢？

如果线程A的操作（x=5）happens-before（先行发生）线程B的操作（y=x），那么可以确定线程B执行后y=5一定成立；

如果他们不存在happens-before原则，那么y=5不一定成立。

happens-before **包含可见性和有序性的约束**

（2）先行发生原则说明

如果Java内存模型中所有有序性都仅靠volatile和synchronized来完成，那么有很多操作都将会变得非常啰嗦，但是我们在编写Java并发代码的时候并没有察觉到这一点。

我们没有时时、处处、次次，添加volatile和synchronized来完成程序，这是因为Java语言中JMM原则下有一个“先行发生”（happens-before）的原则限制和规矩，给你立好了规矩！

这个原则非常重要：

它是判断数据是否存在竞争，线程是否安全的非常有力的手段。依赖这个原则，可以通过几条简单规则一揽子解决并发环境下两个操作之间是否存在冲突的所有问题，而不需要陷入Java内存模型苦涩难懂的底层编译原理之中。

（3）happens-before总原则

- 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前
- 两个操作之间存在happens-before关系，并不意味着一定要按照happens-before原则制定的顺序来执行。如果重排序之后的执行结果与按照happens-before关系来执行的结果一致，那么这种重排序并不非法。

（4）happen-before之8条

次序规则：

一个线程内，按照代码顺序，写在前面的操作先行发生于写在后面的操作；前一个操作的结果可以被后续的操作获取。直白一点就是：前面一个操作把变量x赋值为1，那后面一个操作肯定能知道x已经变成了1

锁定原则：

一个unlock操作先行发生于后面（这里的后面，是指时间上的先后）对同一个锁的lock操作。

```
static Object objectLock = new Object();

public static void main(String[] args) throws InterruptedException {
    // 对于同一把锁objectLock, threadA一定要先unlock同一把锁后
    // threadB才能获取该锁，即：A先行发生于B
    synchronized(objectLock) {
    }
}
```

volatile变量规则：

对一个volatile变量的写操作先行发生于后面对这个变量的读操作；前面的写对后面的读是可见的，这里的“后面”同样指时间上的先后

传递规则：

如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则（Thread Start Rule）：

Thread对象的start方法先行发生于此线程的每一个动作

线程中断规则（Thread Interruption Rule）：

对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；可以通过Thread.interrupted()检测到是否发生中断

也就是说：你要先调用interrupt()方法设置过中断标志位，才能检测到中断发送

线程终止规则 (Thread Termination Rule) :

线程中的所有操作都先行发生于对此线程的终止检测

可以通过isAlive等手段检测线程是否已经终止执行

对象终结规则 (Finalizer Rule) :

一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize()方法的开始

就是：对象没有完成初始化之前，是不能调用finalized()方法的

(5) 小总结

在Java语言里面，happens-before的语义本质上是一种**可见性**

A happens-before B意味着A发生过的事情对B来说是可见的，无论A事件和B事件是否发生在同一个线程里

JMM的设计分为两部分：

- 一部分是面向程序员提供的，也就是happens-before规则，它通俗易懂的向我们程序员阐述了一个强内存模型，我们只要理解happens-before规则，就可以编写并发安全的程序了
- 另一部分是针对JVM实现的，为了尽可能少的对编译器和处理器做约束从而提高性能，JMM在不影响程序执行结果的前提下对其不做要求，即允许优化重排序。我们只需要关注前者就好了，也就是理解happens-before规则即可，其他繁琐的内容有JMM规范结合操作系统给我们搞定，我们只要写好代码即可。

6、案例说明

code:

```
private int value = 0;

public int getValue() {
    return value;
}

public int setValue() {
    return ++value;
}
```

假设存在线程A和B，线程A（时间上的先后）调用了setValue()，然后线程B调用了同一个对象的getValue()，那么线程B收到的返回值是什么？

使用这段简单的代码依次分析happens-before的规则（忽略后四条）：

1. 由于两个方法是由不同的线程调用，不在同一个线程，所以肯定不满足程序次序规则；
2. 两个方法都没有使用锁，所以不满足锁定规则；
3. 变量不是用volatile修饰的，所以volatile变量规则不满足；
4. 传递规则肯定不满足；

所以我们无法通过happens-before原则推导出线程A happens-before线程B，虽然可以确认在时间上线程A优先于线程B指定，但就是无法确认线程B获得的结果是什么，所以这段代码不是线程安全的。

修复01: 把getter/setter方法都定义为synchronized方法

```
private int value = 0;

public synchronized int getValue() {
    return value;
}

public synchronized int setValue() {
    return ++value;
}
```

修复02: 把value定义为volatile变量，由于setter方法对value的修改不依赖value的原值，满足volatile关键字的使用场景

```
private volatile int value = 0;
```

```
public int getValue() {  
    return value;  
}  
  
public synchronized int setValue() {  
    return ++value;  
}
```