**\* 第12章 Java内存模型与线程**

衡量一个服务性能的高低好坏，**每秒事务处理数（TPS）**是最重要的指标之一。它代表着**一秒内服务端平均能响应的请求总数**，而TPS值与程序的并发能力又有着非常密切的关系。对于计算量相同的任务，程序线程并发协调得越有条不紊，效率自然就会越高；反之，线程之间频繁阻塞甚至死锁，将会大大降低程序的并发能力。

“高效并发”是本书讲解JVM的最后一部分，将会介绍虚拟机如何实现多线程、多线程之间由于共享和竞争数据而导致的一系列问题及解决方案。

一、硬件的效率与一致性

**高速缓存**

**高速缓存**

**高速缓存**

**缓**

**存**

**一**

**致**

**性**

**协**

**议**

**处理器**

**处理器**

**处理器**

**二、Java内存模型（JMM）**

JVM规范中定义一种Java内存模型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。

（1）主内存与工作内存

JMM的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。这里的变量（Variables）不同于Java编程中所描述的**变量**，它**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，因为后者是线程私有**的，不会被共享，自然就不会存在竞争问题。

JMM规定**所有的变量都存储在主内存**（Main Memory，此处指虚拟机内存的一部分，并非物理硬件的主存）中。**每条线程还有自己的工作内存**（Working Memory），线程的**工作内存保存了被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）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。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递均需要通过主内存来完成。

需要注意：

（1）如果局部变量是一个reference类型，它引用的对象在Java堆中可被各个线程共享，但是reference本身在Java栈的局部变量表中，它是线程私有的。

（2）另外关于工作内存的副本拷贝，如果线程访问一个对象，只会将该对象的引用、对象中某个在线程访问到的字段进行拷贝，但不会有虚拟机实现成把整个对象拷贝一次。

（3）根据JVM规范规定，volatile变量也是有工作内存的拷贝，但是由于其特殊的操作顺序性规定，所以操作起来如同直接在主内存中读写访问一般。

**类比虚拟机栈部分区域**

**类比Java堆中对**

**象实例数据部分**

**工作内存**

**工作内存**

**工作内存**

**Save**

**和**

**Load**

**操作**

**Java线程**

**Java线程**

**Java线程**

线程、主内存、工作内存三者交互关系

（2）内存间交互操作

主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存，如何从工作内存同步回主内存之类的实现细节，JMM定义了以下8种操作来完成，每一种操作都是**原子的**、不可再分的。\_

自JSR-133文档后，已经放弃采用这8种操作去定义Java内存模型的访问协议，因为这种定义相当严谨且十分繁琐，实践起来很麻烦。

lock（锁定）：作用于**主内存的变量**，把一个变量表示为一条线程独占状态。

unlock（解锁）：作用于**主内存的变量**，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它线程锁定。

read（读取）：作用于主内存的变量，把一个变量的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。

load（载入）：作用于工作内存的变量，它把之前通过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，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。

assigin（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，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。

write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

JMM模型规定执行上述8种基本操作必须满足以下规则：

（1）把一个变量从主内存复制到工作内存：read->load；

把变量从工作内存同步回主内存：store->write；

JMM模型只要求上述两个操作必须按顺序执行，成对搭配。但没有保证是连续执行。即read与load之间，store与write之间是可插入其他指令的。

（2）不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。

（3）不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。

（4）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。

（5）一个新的变量只能在主内存中“诞生”，且不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说，就是对一个变量实施use、store操作之前，必须先经过了assign和load操作。

（6）一个变量在同一个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执行多次，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。

（7）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操作初始化变量的值。

（8）如果一个变量事先没有被lock操作锁定，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定住的变量。

（9）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（store->write）

（3）对于volatile型变量的特殊规则

关键字volatile是JVM提供的最轻量级的同步机制。当一个变量被定义为volatile之后，该变量将具备两种特性。

**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。而普通变量无法做到这一点，因为普通变量的值在线程间传递均需要通过主内存来完成。例如，线程A在工作内存中修改了一个普通变量的值，然后向主内存进行回写，另外一条线程B等线程A回写完成后再从主内存进行读取操作，新变量值才会对线程B可见。

然而，基于volatile变量的运算在并发下并不是安全的。虽然volatile变量在各个线程的工作内存中不存在一致性问题，但是由于Java里的运算并非原子操作，导致**volatile变量**的运算在并发下一样是不安全的。因此在**不符合以下两条规则**的运算场景中，仍然**需要通过加锁**（使用synchronized或java.util.concurrent中的原子类）来保证原子性。

（1）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。

（2）变量不需要与其他的状态变量共同参与不变约束。

**二是禁止指令重排序优化。**

普通变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。

最后总结一下JMM对volatile变量定义的特殊规则：

设T为线程，V和W分别表示两个volatile型变量，那么在进行read、load、use、assign、store和write操作时需要满足如下规则：

（1）只有当线程T对变量V执行的**前**一个动作是**load**时，线程T才能对变量V**后**执行**use**操作；并且，只有当线程T对变量V执行的后一个动作是use的时候，线程T才能对变量V执行load操作。线程T对变量V的use动作可以认为是和线程T对变量V的load、read动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次使用V前都必须先从主内存刷新最新的值，用于保证能看见其他线程对变量V所做的修改后的值）

（2）只有当线程T对变量V执行的**前**一个动作是**assign**时，线程T才能**后**对变量V执行**store**动作；并且，只有当线程T对变量V执行的后一个动作是store时，线程T才能对变量V执行assign动作。线程T对变量V的assign动作可以认为是和线程T对变量V的store、write动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次修改V后都必须同步回主内存中，用于保证其他线程可以看到自己对变量V所做的修改）。

（3）假定动作A是线程T对变量V实施的use或assign动作，假定动作F是和动作A相关联的load或store动作，假定动作P是和动作F相应的对变量V的read或write动作；类似的，假定动作B是线程T对变量W实施的use或assign动作，假定动作G是和动作B相关联的load或store动作，假定动作Q是和动作G相应的对变量W的read或write动作。如果A先于B，那么P先于Q（这条规则要求volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同）

（4）对于long和double型变量的特殊规则

JMM规定了8个原子操作，但对于64位数据类型（long和double），在模型中特别定义了一条相对宽松的规则：

允许虚拟机将没有被volatile修饰的64位数据读写操作划分为两次32位的操作来进行，即允许虚拟机实现选择可以不保证64位数据类型的load、store、read、write这4个操作的原子性。这就是所谓long和double的非原子协定。

如果有多个线程共享一个未被声明为volatile的long或double类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了“半个变量”的数值。但是目前**主流商用虚拟机**都选择把**64位数据的读写操作作为原子操作来对待**，所以程序员在编写代码时一般**不需要特别将用到的long和double变量专门声明为volatile**。