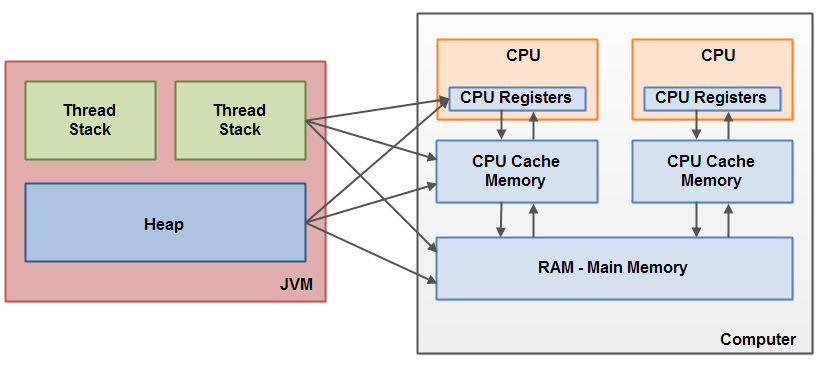
**概述**

在《实例说明JVM内存分布》一文中，我们由JVM内存结构，初步引申得出哪些是线程安全的，哪些是非线程安全的。然而，Java在实际多线程工作时，要远比这复杂的多。Java通过Java内存模型与线程规范（jsr-133）定义了一系列规范和约束，JVM通过实现这些规范和约束保障了Java多线程时候的高效与稳定。

JVM通过JMM规范屏蔽掉了硬件和操作系统之间的差异。用户只需要知道JVM提供的内存和多线程规范，无需再关心平台差异。JVM更是提供统一的字节码指令和默认访问使用底层的方式，从而不需要通过修改代码来适配平台。JMM是Java实现跨平台的重要保证之一。

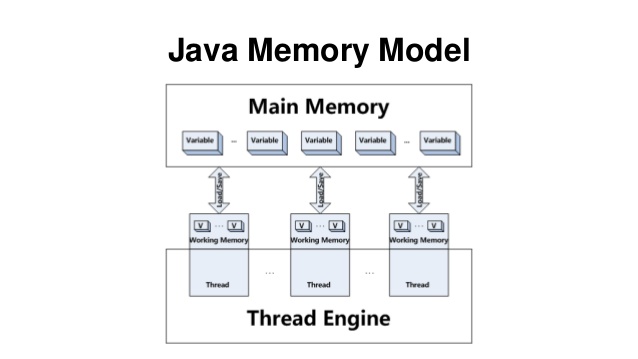
**什么是JMM**

在实际情况当中，程序变量在内存中可能是这样映射的（图一）：



图一

堆和栈中的数据必定会在主存中，但同时也可能映射到各种缓存及CPU寄存器中。不同的操作系统和硬件之间处理方式有差异，如果交给用户去处理，可以想象难度和复杂度，最重要的是，这也违背了Java的初衷：跨平台。于是JMM将其简化成如下图（图二）：



图二

JMM并不关心有多少缓存和寄存器，所有的缓存和寄存器统一看作工作内存（Working Memory），计算器内存就当做主内存（Main Memory）。然后JMM定义了一系列主存和工作内存的关系、原语以及操作原则来规范JVM的实现。

**JMM与多线程**

JMM主要是围绕并发编程中的问题进行约定的：

1）原子性

原子操作是计算机和程序语言必须提供和保证的机制，否则，在多线程的时候我们无法保证数据的完整性。JMM定义了8种基本的原子操作供JVM使用，JVM通过这8种基本的原子操作来实现更大范围的原子操作。这8种原子操作是：

* **lock（锁定）**：作用于主内存的变量，把一个变量标识为一条线程独占状态。
* **unlock（解锁）**：作用于主内存变量，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他线程锁定。
* **read（读取）**：作用于主内存变量，把一个变量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，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
* **assign（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值给工作内存的变量，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，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
* **write（写入）**：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

例如，我们一个线程新使用一个int变量，那么，JVM会将这个变量从主存加载到工作内存当中，这个过程调用了read，那么read必须是一个原子性的操作，不能出现read到一半被其他线程打断的情况。

lock和unlock也同理，要么获取锁或者释放锁成功，要么失败，不能出现中间状态。然后JVM通过字节码指令monitorenter和monitorexit封装了lock和unlock提供给用户，来实现更大范围的原子操作。

2）可见性

可见性是指我们一个线程修改了一个变量，想让其他变量也知道，即如何将我们执行的结果同步到主存或者将一个被修改的变量从主存同步到工作内存。 Java语言提供了以下几种方式：synchronized、volatile和final。

synchronized主要通过monitorenter和monitorexit实现，底层对应的是lock和unlock原子操作。JMM规定，在执行lock操作时，将会清空工作内存变量的值，在执行引擎使用的时候，再重新执行load和assign操作进行重新加载；同样，在执行unlock前，必须把工作内存中的变量同步（store、write）到主存中。再结合原子性，这样就保证了变量的可见性和一致性。

volatile和final并没有任何字节码指令与其对应，这两个关键字仅仅是类型修饰符。例如，当JVM检测到变量有volatile修饰时候，JVM会从主存中读取数据，写入数据时候也会写入到主存当中。final关键字则是在JVM检测到该字时会把变量的初始化结果同步到主存，然后不再发生变化。

3）有序性

在Java中，synchronized通过每一次lock只允许一个线程操作来进行有序性的控制。而volatile则通过禁止指令重排来进行执行顺序的保证。

除此之外，JVM运行时候还约定了一些规则来进行顺序性保证，叫Happens-Before原则，“Before”之前的数据在Before之后必须确保已经同步到主存，即“Before”之后的线程获取的主存内的值肯定是最新的：

* **程序次序规则：**一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。
* **管程锁定规则：**一个unLock操作先行发生于后面对同一个锁的lock操作。
* **volatile变量规则：**对一个变量的写操作先行发生于后面对这个变量的读操作。
* **线程启动规则：**Thread对象的start()方法先行发生于此线程的每个一个动作。
* **线程终结规则：**线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行。
* **线程中断规则：**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。
* **对象终结规则：**一个对象的初始化完成先行发生于他的finalize()方法的开始。
* **传递性：**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。

JMM通过一个模型和一系列规范来指导JVM的实现，隔离了用户层和操作系统及硬件层，用户再也不需要关心什么是缓存一致性，什么是MESI以及如何处理这些问题。最终，Java开发人员只需要简单的使用几个关键字即可实现多线程的并发。

**其他情况**

JMM规定允许JVM将没有被volatile修饰的64位数据的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和write这四个操作的原子性，即如果有多个线程共享一个并未声明为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。尤其在32位的JVM上对long和double类型数据进行操作，因为这两者数据都是占有8个字节，操作系统无法将该数据一次性全部读取到寄存器，所以，需要分两步完成。但由于目前各种平台下的JVM几乎都选择把64位数据的读写操作作为原子操作来对待，因此在编写代码时一般也不需要将用到的long和double变量专门声明为volatile。但我们不能忘记这个问题的存在。

**总结**

JMM（JSR-133）看起来非常抽象，类似于《Java虚拟机规范》，因为这是一个JVM多线程实现的指南，算是虚拟机规范的一部分，而且后续《Java内存模型与线程规范》也会合并到虚拟机规范当中，绝大多数情况下Java开发人员都不需要关心其实现细节。尤其在Spring和Servlet大行其道的今天，Spring和Servlet容器提供的功能解决了大多数场景的并发问题，但业务量上来之后，我们终究是绕不开这些最根本的问题。