[**Java内存区域（运行时数据区域）和内存模型（JMM）**](https://www.cnblogs.com/czwbig/p/11127124.html)

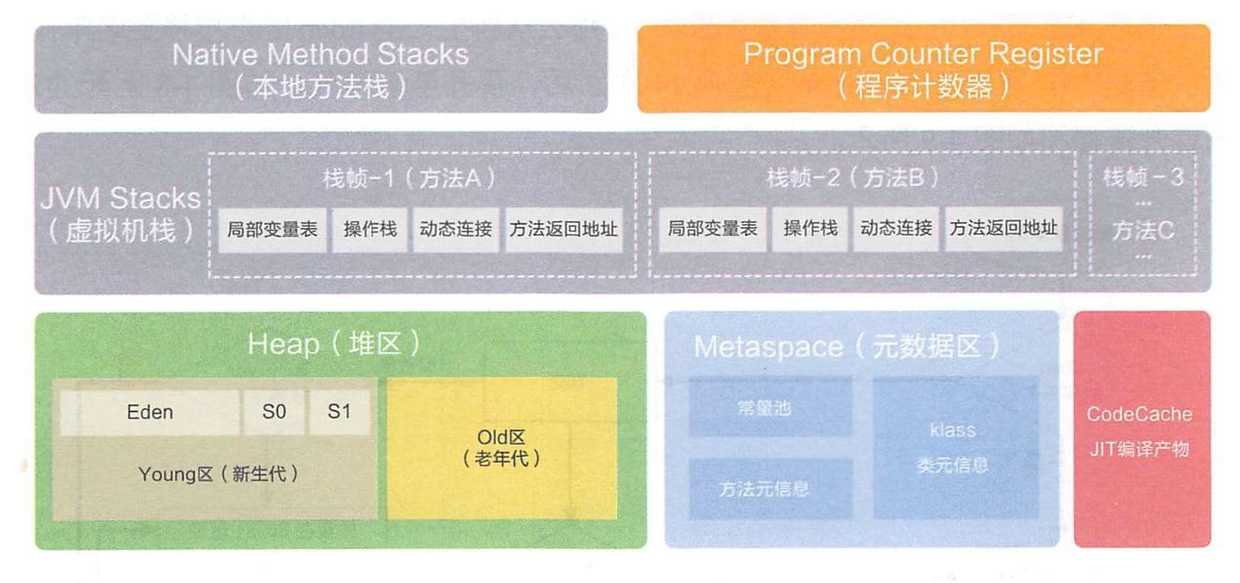
Java 内存区域和内存模型是不一样的东西，内存区域是指 Jvm 运行时将数据分区域存储，强调对内存空间的划分。

而内存模型（Java Memory Model，简称 JMM ）是定义了线程和主内存之间的抽象关系，即 JMM 定义了 JVM 在计算机内存(RAM)中的工作方式，如果我们要想深入了解Java并发编程，就要先理解好Java内存模型。

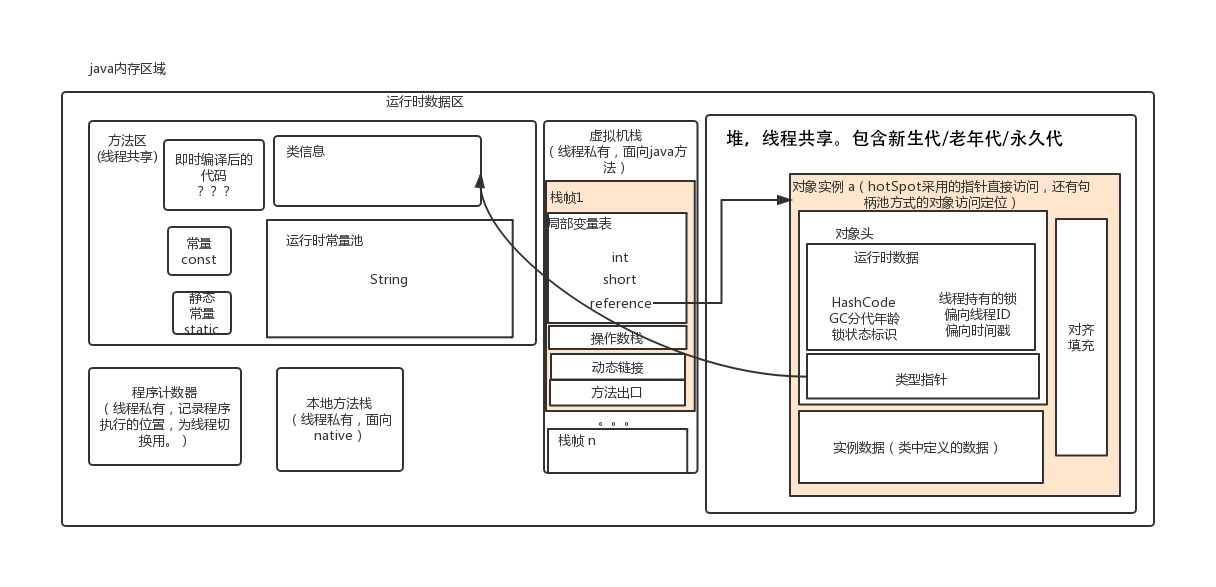
**Java运行时数据区域**

众所周知，Java 虚拟机有自动内存管理机制，如果出现内存泄漏和溢出方面的问题，排查错误就必须要了解虚拟机是怎样使用内存的。

下图是 JDK8 之后的 JVM 内存布局。



JDK8 之前的内存区域图如下:



在 HotSpot JVM 中，永久代中用于存放类和方法的元数据以及常量池，比如Class和Method。每当一个类初次被加载的时候，它的元数据都会放到永久代中。  
永久代是有大小限制的，因此如果加载的类太多，很有可能导致永久代内存溢出，即万恶的 *java.lang.OutOfMemoryError: PermGen* ，为此我们不得不对虚拟机做调优。  
那么，Java 8 中 PermGen 为什么被移出 HotSpot JVM 了？我总结了两个主要原因:

1. 由于 PermGen 内存经常会溢出，引发恼人的 *java.lang.OutOfMemoryError: PermGen*，因此 JVM 的开发者希望这一块内存可以更灵活地被管理，不要再经常出现这样的 OOM
2. 移除 PermGen 可以促进 HotSpot JVM 与 JRockit VM 的融合，因为 JRockit 没有永久代。  
   根据上面的各种原因，PermGen 最终被移除，**方法区移至 Metaspace，字符串常量移至 Java Heap**。

引用自<https://www.sczyh30.com/posts/Java/jvm-metaspace/>

**程序计数器**

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器内核都只会执行一条线程中的指令。

因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

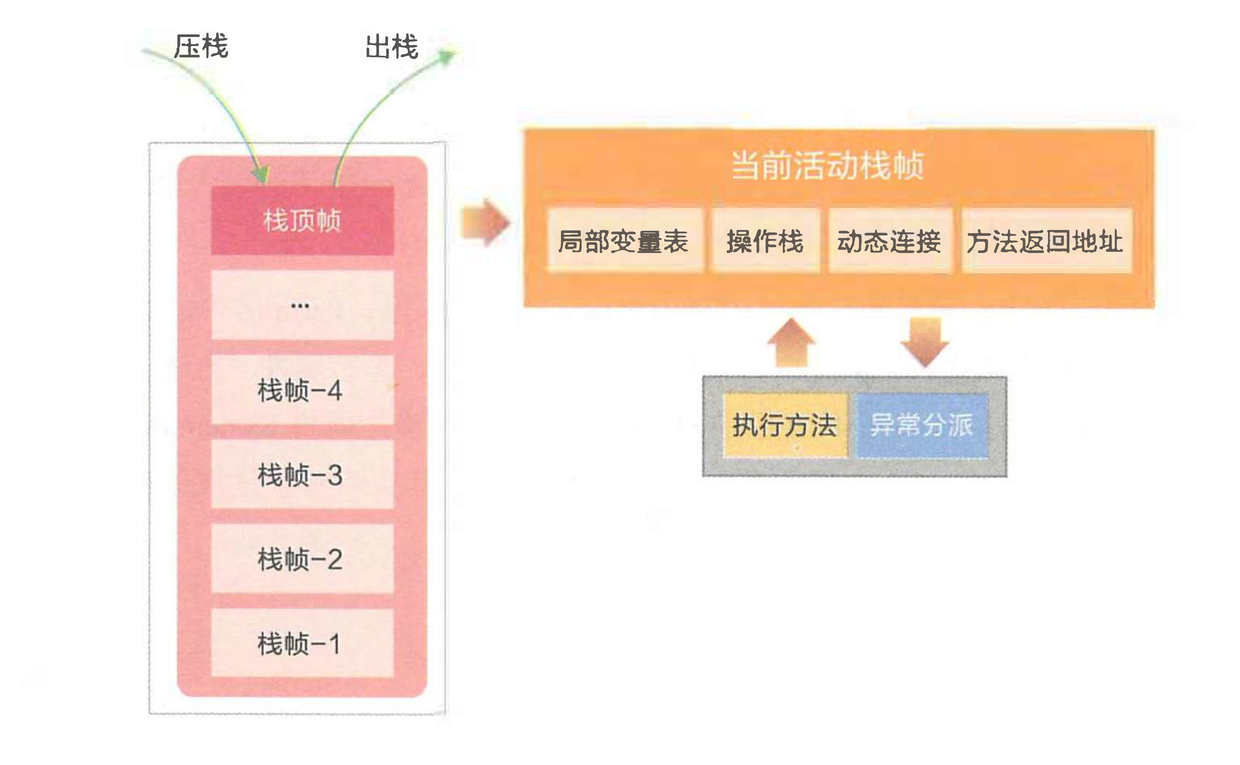
如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

**Java虚拟机栈**

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。

虚拟机栈描述的是 Java 方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame，是方法运行时的基础数据结构）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

在活动线程中，只有位千栈顶的帧才是有效的，称为当前栈帧。正在执行的方法称为当前方法，栈帧是方法运行的基本结构。在执行引擎运行时，所有指令都只能针对当前栈帧进行操作。



**1. 局部变量表**

局部变量表是存放方法参数和局部变量的区域。 局部变量没有准备阶段， 必须显式初始化。如果是非静态方法，**则在 index[0] 位置上存储的是方法所属对象的实例引用**，一个引用变量占 4 个字节，随后存储的是参数和局部变量。字节码指令中的 STORE 指令就是将操作栈中计算完成的局部变呈写回局部变量表的存储空间内。

虚拟机栈规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展），如果扩展时无法申请到足够的内存，就会抛出 OutOfMemoryError 异常。

**2. 操作栈**

操作栈是个初始状态为空的桶式结构栈。在方法执行过程中， 会有各种指令往  
栈中写入和提取信息。JVM 的执行引擎是基于栈的执行引擎， 其中的栈指的就是操  
作栈。字节码指令集的定义都是基于栈类型的，栈的深度在方法元信息的 stack 属性中。

**i++ 和 ++i 的区别：**

1. i++：从局部变量表取出 i 并压入操作栈(load memory)，然后对局部变量表中的 i 自增 1(add&store memory)，将操作栈栈顶值取出使用，如此线程从操作栈读到的是自增之前的值。
2. ++i：先对局部变量表的 i 自增 1(load memory&add&store memory)，然后取出并压入操作栈(load memory)，再将操作栈栈顶值取出使用，线程从操作栈读到的是自增之后的值。

之前之所以说 i++ 不是原子操作，即使使用 volatile 修饰也不是线程安全，就是因为，可能 i 被从局部变量表（内存）取出，压入操作栈（寄存器），操作栈中自增，使用栈顶值更新局部变量表（寄存器更新写入内存），其中分为 3 步，volatile 保证可见性，保证每次从局部变量表读取的都是最新的值，但可能这 3 步可能被另一个线程的 3 步打断，产生数据互相覆盖问题，从而导致 i 的值比预期的小。

**3. 动态链接**

每个栈帧中包含一个在常量池中对当前方法的引用， 目的是支持方法调用过程的动态连接。

**4.方法返回地址**

方法执行时有两种退出情况：

1. 正常退出，即正常执行到任何方法的返回字节码指令，如 RETURN、IRETURN、ARETURN 等；
2. 异常退出。

无论何种退出情况，都将返回至方法当前被调用的位置。方法退出的过程相当于弹出当前栈帧，退出可能有三种方式：

1. 返回值压入上层调用栈帧。
2. 异常信息抛给能够处理的栈帧。
3. PC计数器指向方法调用后的下一条指令。

**本地方法栈**

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。Sun HotSpot 虚拟机直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

线程开始调用本地方法时，会进入 个不再受 JVM 约束的世界。本地方法可以通过 JNI(Java Native Interface)来访问虚拟机运行时的数据区，甚至可以调用寄存器，具有和 JVM 相同的能力和权限。 当大量本地方法出现时，势必会削弱 JVM 对系统的控制力，因为它的出错信息都比较黑盒。对内存不足的情况，本地方法栈还是会抛出 nativeheapOutOfMemory。

JNI 类本地方法最著名的应该是 System.currentTimeMillis() ，JNI使 Java 深度使用操作系统的特性功能，复用非 Java 代码。 但是在项目过程中， 如果大量使用其他语言来实现 JNI , 就会丧失跨平台特性。

**Java堆**

对于大多数应用来说，Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”（Garbage Collected Heap）。从内存回收的角度来看，由于现在收集器基本都采用分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等。从内存分配的角度来看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer,TLAB）。

Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，当前主流的虚拟机都是按照可扩展来实现的（通过 -Xmx 和 -Xms 控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异常。

**方法区**

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然  
Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

Java 虚拟机规范对方法区的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。垃圾收集行为在这个区域是比较少出现的，其内存回收目标主要是针对常量池的回收和对类型的卸载。当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

**JDK8 之前，Hotspot 中方法区的实现是永久代（Perm），JDK8 开始使用元空间（Metaspace），以前永久代所有内容的字符串常量移至堆内存，其他内容移至元空间，元空间直接在本地内存分配。**

为什么要使用元空间取代永久代的实现？

1. 字符串存在永久代中，容易出现性能问题和内存溢出。
2. 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3. 永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。
4. 将 HotSpot 与 JRockit 合二为一。

**运行时常量池**

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

一般来说，除了保存 Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只有编译期才能产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern() 方法。

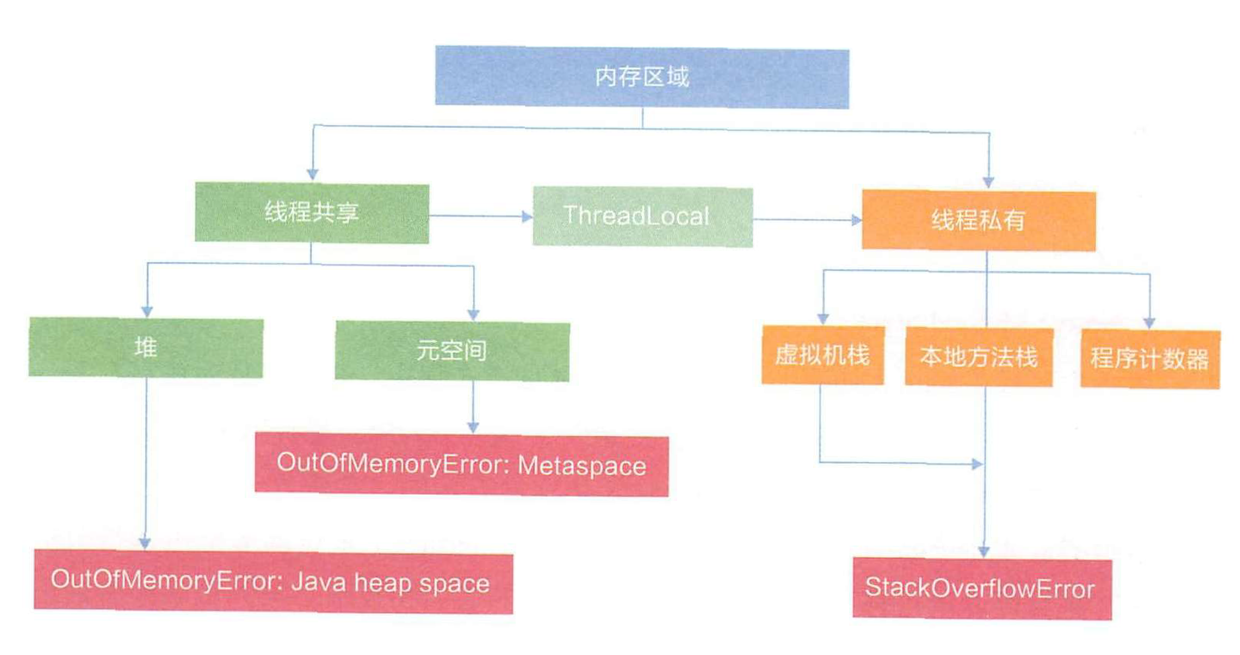
既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

**直接内存**

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域。

在 JDK 1.4 中新加入了 NIO，引入了一种基于通道（Channel）与缓冲区（Buffer）的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

显然，本机直接内存的分配不会受到 Java 堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存（包括 RAM 以及 SWAP 区或者分页文件）大小以及处理器寻址空间的限制。服务器管理员在配置虚拟机参数时，会根据实际内存设置 -Xmx 等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），从而导致动态扩展时出现 OutOfMemoryError 异常。



**Java内存模型**

Java内存模型是共享内存的并发模型，线程之间主要通过读-写共享变量（堆内存中的实例域，静态域和数组元素）来完成隐式通信。

Java 内存模型（JMM）控制 Java 线程之间的通信，决定一个线程对共享变量的写入何时对另一个线程可见。

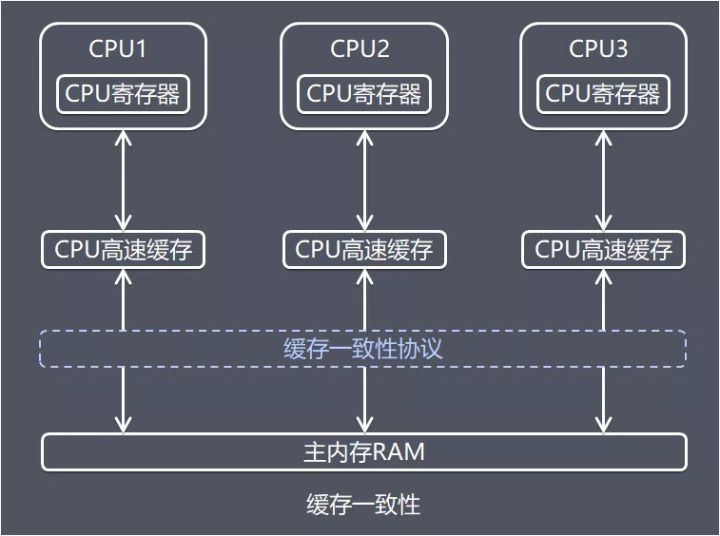
**计算机高速缓存和缓存一致性**

计算机在高速的 CPU 和相对低速的存储设备之间使用高速缓存，作为内存和处理器之间的缓冲。将运算需要使用到的数据复制到缓存中，让运算能快速运行，当运算结束后再从缓存同步回内存之中。

在多处理器的系统中(或者单处理器多核的系统)，每个处理器内核都有自己的高速缓存，它们有共享同一主内存(Main Memory)。

当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致。

为此，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议进行操作，来维护缓存的一致性。



**JVM主内存与工作内存**

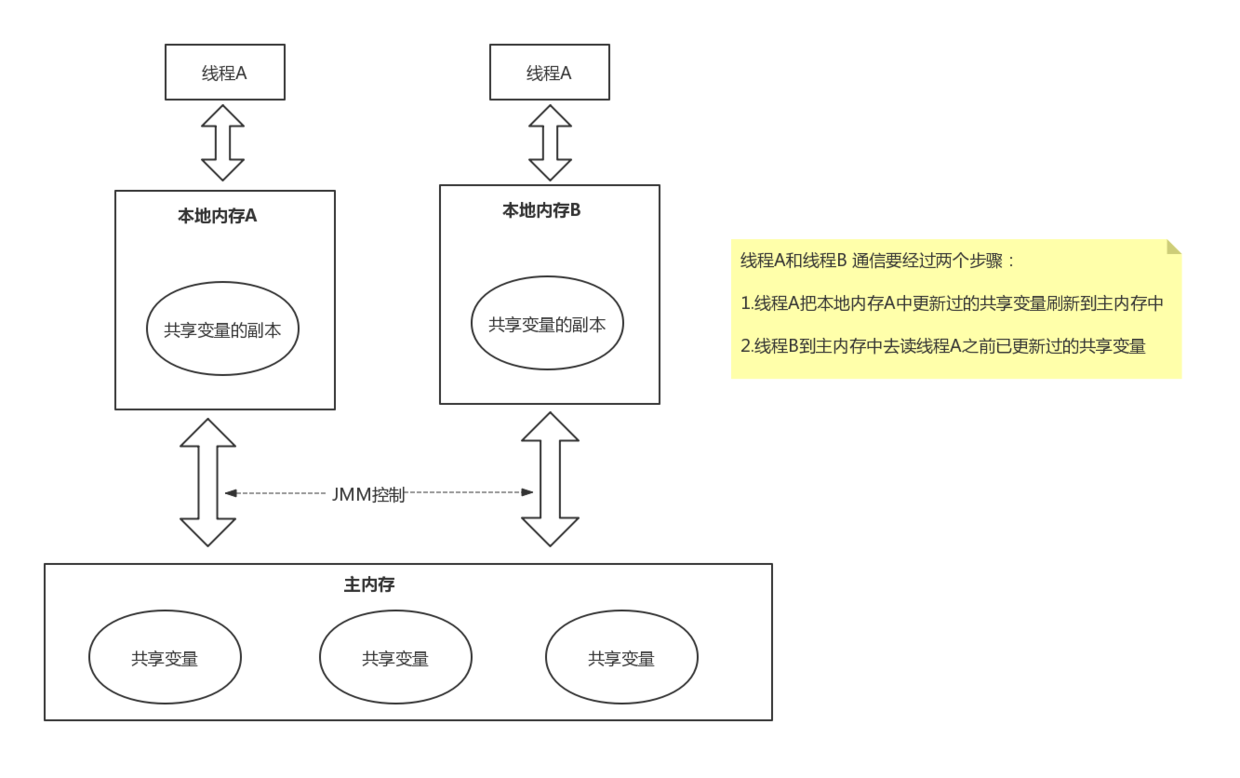
Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量（线程共享的变量）存储到内存和从内存中取出变量这样底层细节。

Java内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程对变量的所有操作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。

这里的工作内存是 JMM 的一个抽象概念，也叫本地内存，其存储了该线程以读 / 写共享变量的副本。

**就像每个处理器内核拥有私有的高速缓存，JMM 中每个线程拥有私有的本地内存。**

不同线程之间无法直接访问对方工作内存中的变量，线程间的通信一般有两种方式进行，一是通过消息传递，二是共享内存。Java 线程间的通信采用的是共享内存方式，线程、主内存和工作内存的交互关系如下图所示：



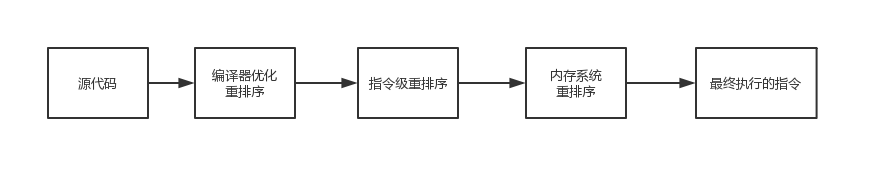
这里所讲的主内存、工作内存与 Java 内存区域中的 Java 堆、栈、方法区等并不是同一个层次的内存划分，这两者基本上是没有关系的，如果两者一定要勉强对应起来，那从变量、主内存、工作内存的定义来看，主内存主要对应于Java堆中的对象实例数据部分，而工作内存则对应于虚拟机栈中的部分区域。

**重排序和happens-before规则**

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读 / 写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从 java 源代码到最终实际执行的指令序列，会分别经历下面三种重排序：



JMM 属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

java 编译器禁止处理器重排序是通过在生成指令序列的适当位置会插入内存屏障（重排序时不能把后面的指令重排序到内存屏障之前的位置）指令来实现的。

**happens-before**

从 JDK5 开始，java 内存模型提出了 happens-before 的概念，通过这个概念来阐述操作之间的内存可见性。

如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在 happens-before 关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

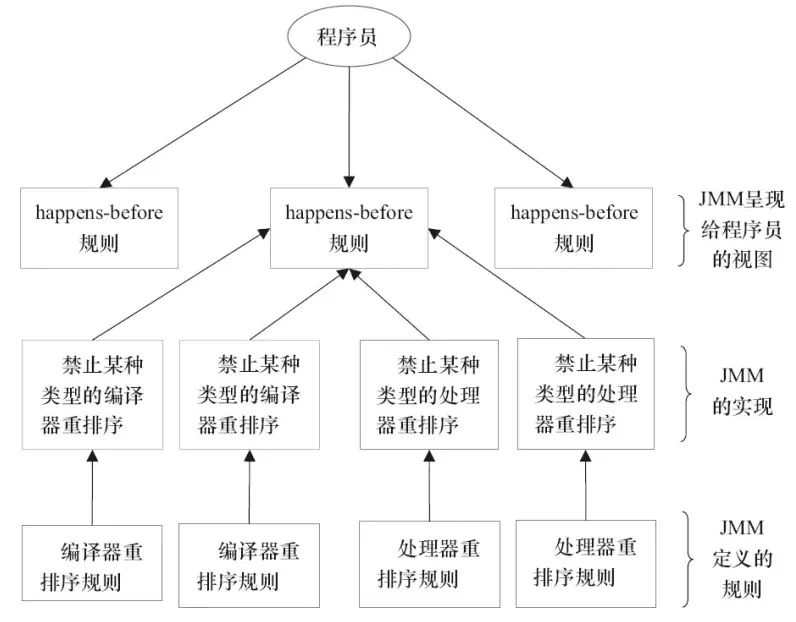
**这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。**

如果 A happens-before B，那么 Java 内存模型将向程序员保证—— A 操作的结果将对 B 可见，且 A 的执行顺序排在 B 之前。

重要的 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 于该线程中的任意后续操作。
2. 监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。
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。

下图是 happens-before 与 JMM 的关系



**volatile关键字**

volatile 可以说是 JVM 提供的最轻量级的同步机制，当一个变量定义为volatile之后，它将具备两种特性：

1. **保证此变量对所有线程的可见性。**而普通变量不能做到这一点，普通变量的值在线程间传递均需要通过主内存来完成。

注意，volatile 虽然保证了可见性，但是 Java 里面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。而 synchronized 关键字则是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得线程安全的。

1. **禁止指令重排序优化。**普通的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，推荐与感谢：  
[深入理解Java虚拟机（第2版）](https://book.douban.com/subject/24722612/)  
[码出高效：Java开发手册](https://book.douban.com/subject/30333948/)  
[Java内存模型原理，你真的理解吗？](https://zhuanlan.zhihu.com/p/51613784))  
[深入理解 Java 内存模型](https://www.infoq.cn/article/java-memory-model-1)

如果觉得本文有所帮助，欢迎点【推荐】！文章错误之处烦请留言。 转载说明：转载后必须在文章开头明显地给出作者和原文链接；引用必须注明出处；需要二次修改发布请联系作者征得同意。

分类: [JVM](https://www.cnblogs.com/czwbig/category/1495542.html)