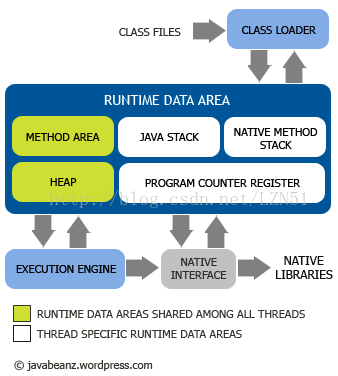
最近翻看了java线程相关的东西，书中有一边专门讲到java内存模型，读完之后边回想起java虚拟机模型，那时心中便在思考java内存模型（以下简称jmm）和java虚拟机模型（以下简称jvm）之间的关系，下面将详细讲述。

**一 jvm结构**

jvm的内部结构如下图所示，这张图很清楚形象的描绘了整个JVM的内部结构，以及各个部分之间的交互和作用。



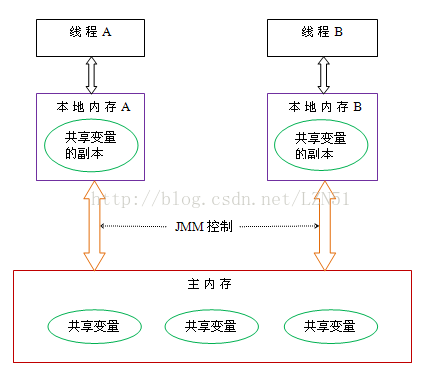
1. **Class Loader（类加载器）**就是将Class文件加载到内存，再说的详细一点就是，把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是类加载器的作用。
2. **Run Data Area（运行时数据区）** 就是我们常说的JVM管理的内存了，也是我们这里主要讨论的部分。运行数据区是整个JVM的重点。我们所有写的程序都被加载到这里，之后才开始运行。这部分也是我们这里将要讨论的重点。
3. **Execution engine（执行引擎）** 是Java虚拟机最核心的组成部分之一。执行引擎用于执行指令，不同的java虚拟机内部实现中，执行引擎在执行Java代码的时候可能有解释执行（解释器执行）和编译执行（通过即时编译器产生本地代码执行，例如BEA JRockit），也有可能两者兼备。任何JVM specification实现(JDK)的核心都是Execution engine，不同的JDK例如Sun 的JDK 和IBM的JDK好坏主要就取决于他们各自实现的Execution engine的好坏。
4. **Native interface 与native libraries交互**，是其它编程语言交互的接口。当调用native方法的时候，就进入了一个全新的并且不再受虚拟机限制的世界，所以也很容易出现JVM无法控制的native heap OutOfMemory。

**二 Run Data Area（运行时数据区）**

* Program Counter Register
  + 程序计数器，线程私有、指向下一条要很执行的指令
* Java Stack
  + Java虚拟机栈，线程私有，生命周期与线程相同。描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口
* Native Method Stack
  + 为虚拟机使用到的Native 方法服务
* Heap
  + 线程共享，由于现在收集器基本采用的分代收集算法，所以Java堆中还可以细分：新生代和老生代；更细致一点的有Eden空间、From Survivor空间、To Survivor空间等。所有的对象实例以及数组都要在堆上分配，是垃圾收集器管理的主要区域
* Method Area
  + 方法区，别名叫做非堆(Non-Heap)，线程共享的内存区域。目的是与Java堆区分开来，存储类信息、常量、静态变量、即时编译器编译后的代码。 方法区存放的信息包括：
    - **A 类的基本信息：**
      * 1.每个类的全限定名
      * 2.每个类的直接超类的全限定名(可约束类型转换)
      * 3.该类是类还是接口
      * 4.该类型的访问修饰符
      * 5.直接超接口的全限定名的有序列表
    - **B 已装载类的详细信息**
      * 1.运行时常量池：在方法区中，每个类型都对应一个常量池，存放该类型所用到的所有常量，常量池中存储了诸如文字字符串、final变量值、类名和方法名常量。它们以数组形式通过索引被访 问，是外部调用与类联系及类型对象化的桥梁。（存的可能是个普通的字符串，然后经过常量池解析，则变成指向某个类的引用）
      * 2.字段信息：字段信息存放类中声明的每一个字段的信息，包括字段的名、类型、修饰符。字段名称指的是类或接口的实例变量或类变量，字段的描述符是一个指示字段的类型的字符串，如private A a=null;则a为字段名，A为描述符，private为修饰符。
      * 3.方法信息：类中声明的每一个方法的信息，包括方法名、返回值类型、参数类型、修饰符、异常、方法的字节码。(在编译的时候，就已经将方法的局部变量、操作数栈大小等确定并存放在字节码中，在装载的时候，随着类一起装入方法区。)
      * 4.静态变量：就是类变量，类的所有实例都共享，我们只需知道，在方法区有个静态区，静态区专门存放静态变量和静态块。
      * 5.到类classloader的引用：到该类的类装载器的引用。 6.到类class 的引用：jvm为每个加载的类型(译者：包括类和接口)都创建一个java.lang.Class的实例。而jvm必须以某种方式把Class的这个实例和存储在方法区中的类型数据联系起来。

**三 jmm**

Java内存模型(Java Memory Model，JMM)JMM主要是为了规定了线程和内存之间的一些关系。根据JMM的设计，系统存在一个主内存(Main Memory)，Java中所有变量都储存在主存中，对于所有线程都是共享的。每条线程都有自己的工作内存(Working Memory)，工作内存中保存的是主存中某些变量的拷贝，线程对所有变量的操作都是在工作内存中进行，线程之间无法相互直接访问，变量传递均需要通过主存完成。



**四 jvm和jmm之间的关系**

jmm中的主内存、工作内存与jvm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。从更低层次上说，主内存就直接对应于物理硬件的内存，而为了获取更好的运行速度，虚拟机（甚至是硬件系统本身的优化措施）可能会让工作内存优先存储于寄存器和高速缓存中，因为程序运行时主要访问读写的是工作内存。