JVM 独家剖析

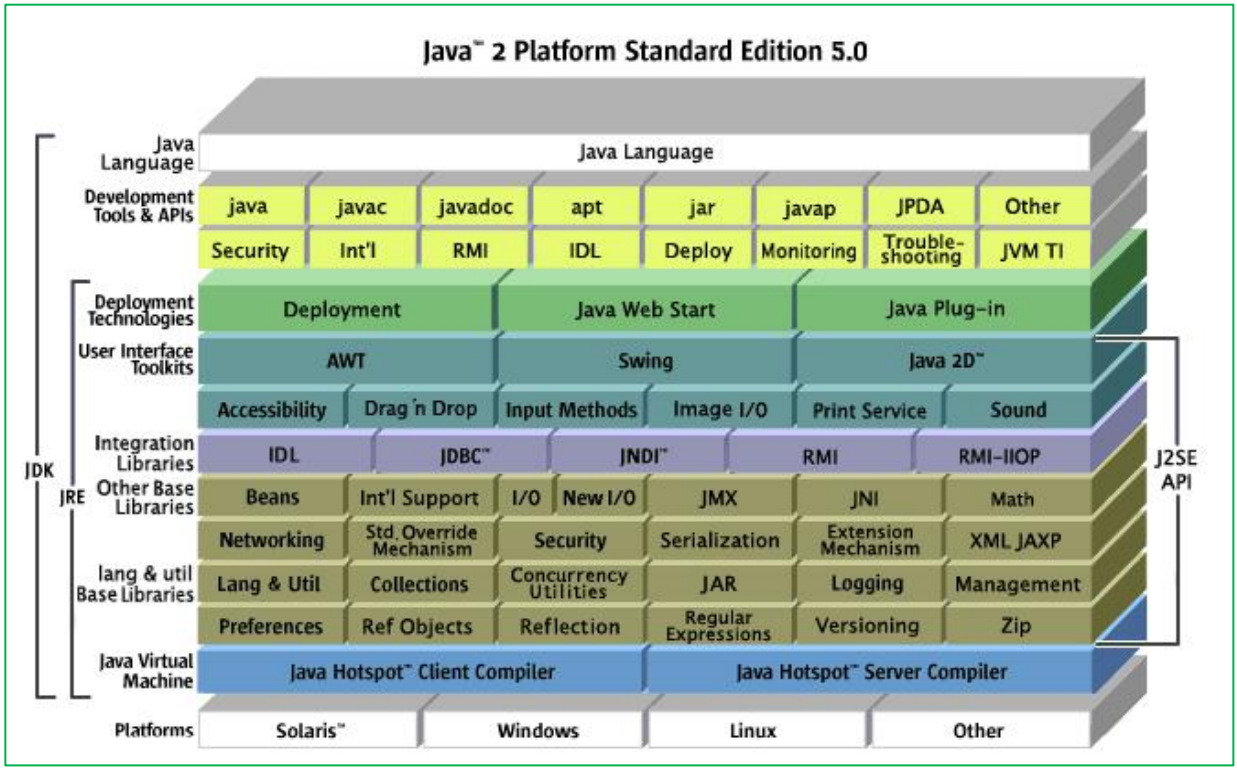
一、JVM 概述

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，JVM 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java 语言的一个非常重要的特点就是与平台的无关性，“一次编译，到处运行”。而使用 Java 虚拟机是实现这一特点的关键。

那么，JVM 的底层实现原理究竟什么呢？下面，博主就以《Java 虚拟机规范（Java SE 7 版）》一书为主要依据，结合部分网络资料，带着大家一起来解读 JVM!

二、JVM 结构

1. Java 平台的逻辑结构

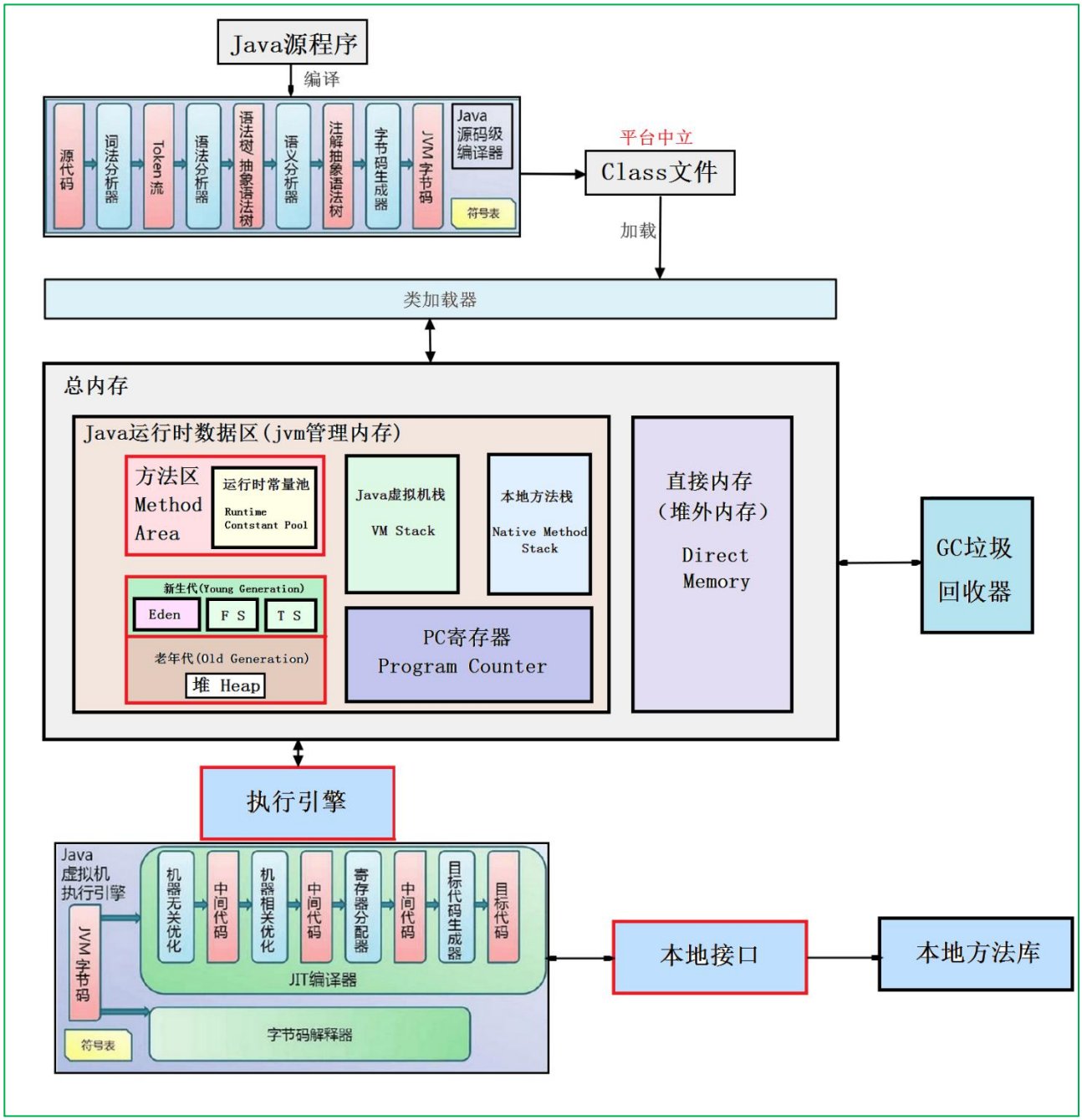


从上图的关系，我们可以简单理解为：

JRE = JVM + 类库。

JDK = JRE + JAVA 的开发工具。

3. JVM 的物理结构



JVM 逻辑结构主要包括两个子系统和两个组件。两个子系统分别是 Classloader(类加载器)子系统和 Executionengine(执行引擎)子系统；两个组件分别是 Runtimedataarea(运行时数据区域)组件和 Nativeinterface(本地接口)组件。

下面博主将对图上的每一个部分，逐一介绍：

（1）Classloader 子系统

根据给定的全限定名类名(如 java.lang.Object)来装载 class 文件的内容到运行时数据域中的方法区域。Java 程序员可以继承 ClassLoader 类来写自己 Classloader。

（2）Executionengine 子系统

执行 classes 中的指令。任何 JVMspecification 实现(JDK)的核心都是 Executionengine，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 Executionengine

的好坏。

（3）Nativeinterface 组件

与 nativelibraries 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 nativeheapOutOfMemory。

（4）RuntimeDataArea 组件

这就是我们常说的 JVM 的内存了。它主要分为五个部分：

1）PC 寄存器

Java 虚拟机可以支持多条线程同时执行，而每一条线程都有自己的 PC（Program Counter）寄存器，简称“线程私有”。在任意时刻，一条 Java 虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法（Current Method）。

如果这个方法不是 native 的，那 PC 寄存器就保存 Java 虚拟机正在执行的字节码指令的地址；如果该方法是 native 的，那 PC 寄存器的值是 undefined。

2）Java 虚拟机栈

线程私有，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，每个方法执行都会创建一个栈帧。

栈帧(Stack Frame)：，每一个栈帧都有自己的局部变量表（Local Variables）、操作数栈（Operand Stack）和动态链接（Dynamic Linking）。

局部变量表：存放编译时的 8 中基本数据类型、引用类型和 returnAddress 类型（指向一条字节码指令地址）。

操作数栈：是一个后进先出 （Last-In-First-Out，LIFO）栈。

动态链接：是一个指向当前方法所属的类的运行时常量池的引用。

方法区有两种异常：

线程请求栈深度大于虚拟机栈深度，抛出 StackOverflowError 异常。

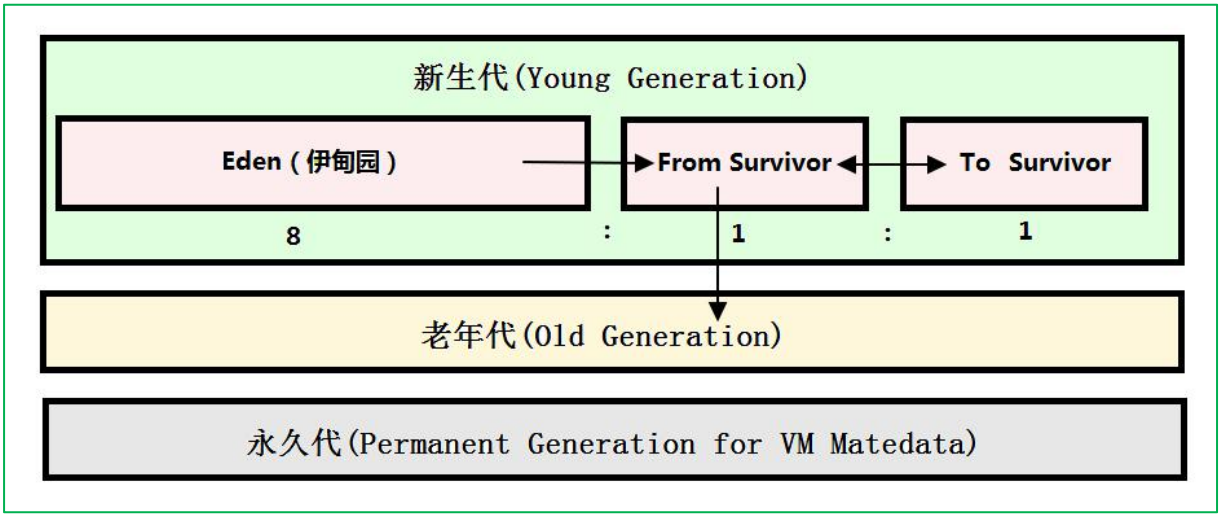
动态拓展时无法申请到足够内存，抛出 OutOfMemoryError 异常。

3）Java 堆

Java 堆（Java Heap）是 Java 虚拟机管理的最大内存区域，虚拟机启动时创建，所有线程共享该内存。该内存唯一目的就是存放对象实例，几乎所有的对象实例都在此分配内存。

Java 堆是垃圾收集器管理的主要区域，也被成为“GC 堆”。在对 GC 堆的划分上，JDK1.7 及以前的版本，和 JDK1.8 是有明显不同的。

JDK1.7 及之前，堆内存通常被分为三块区域：新生代(Young Generation)、老年代(Old Generation)、永久代(Permanent Generation for VM Matedata)



新生代： 用来存放生命周期较短的对象，而新生代又使用复制算法进行 GC ，又将其按照 8:1:1 的比例分为一块较大的 Eden 空间和 2 个较小的 From Survivor 和 To Survivor 空间。

老年代：用来存放生命周期较长的对象。

永久内存：用来存放对象的方法、变量等元数据信息。

Xms Java 堆初始内存，默认值为物理内存的 1/64，当可用的 Java 堆内存小于 40%时，JVM 会将内存调整至-Xmx 所允许的最大值

Xmx Java 堆最大内存，默认值为物理内存的 1/4，当可用的 Java 堆内存大于 70%时，JVM 会将内存调整至-Xms 所指定的初始值

一个对象被创建后，首先被放到新生代的 Eden 内存中，如果存活期超两个 Survivor 之后,就会被转移到长时内存(Old Generation)中；

通过如果永久内存不够，我们就会得到如下错误：

Java.lang.OutOfMemoryError: PermGen

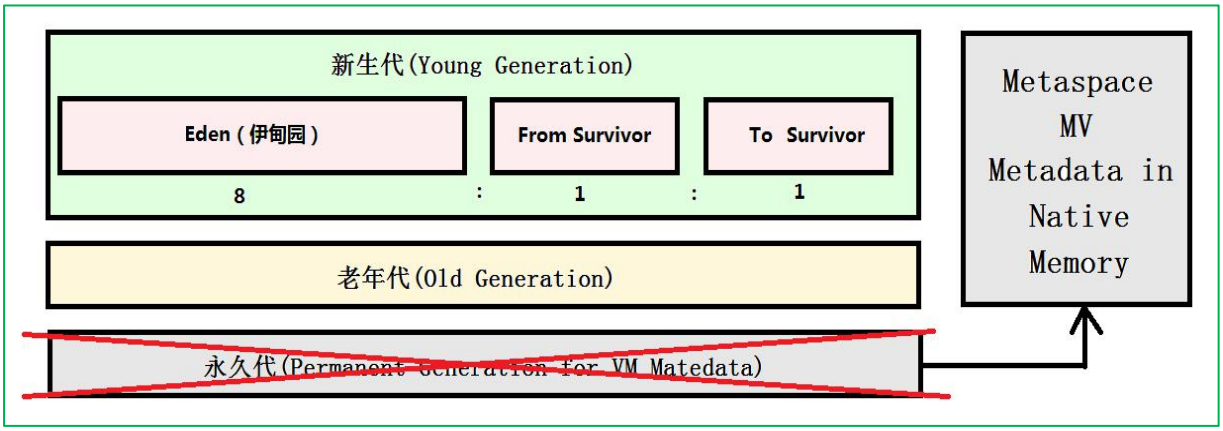
解决：

Eclipse 中，点击“Run”→"Run Configurations",在打开的窗口中点击“Arguments”选项卡。在 VM arguments 中内容最下边输入如下内容后重启：

-Xms256m -Xmx512m -XX:MaxNewSize=256m -XX:MaxPermSize=256m

而在 JDK8 中情况发生了明显的变化，就是一般情况下你都不会得到这个错误，原因在于 JDK8 中把存放元数据中的永久内存从堆内存中移到了本地内存（Native Memory）中了。

JDK8 中 JVM 堆内存结构就变成了如下：



这样永久内存就不再占用堆内存，它可以通过自动增长来避免 JDK7 以及前期版本中，常见的永久内存错误(java.lang.OutOfMemoryError: PermGen)，也许这个就是你的 JDK 升级到 JDK8 的理由之一吧。当然 JDK8 也提供了一个新的设置 Matespace 内存大小的参数，通过这个参数可以设置 Matespace 内存大小，这样我们可以根据自己项目的实际情况，避免过度浪费本地内存，达到有效利用。

-XX:MaxMetaspaceSize=128m 设置最大的元内存空间 128 兆

注意：如果不设置 JVM 将会根据一定的策略自动增加本地元内存空间。如果你设置的元内存空间过小，你的应用程序可能得到以下错误：

java.lang.OutOfMemoryError: Metadata space

4）方法区

方法区和 Java 堆一样，是各个线程的共享的区域，它存储了每一个**类的结构信息**，例如**运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容**、还包括**一些在类、实例、接口初始化时用到的特殊方法**。

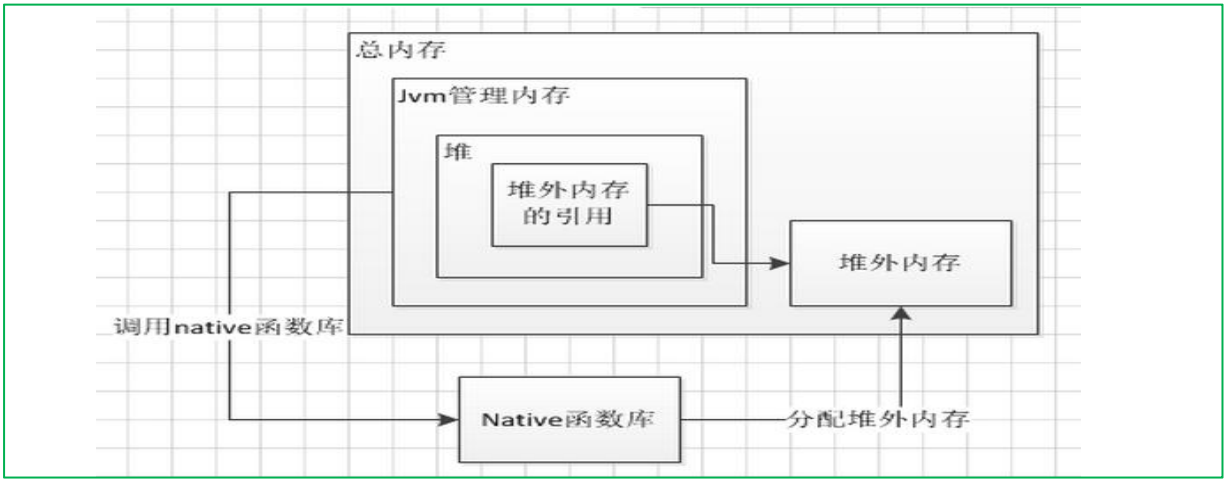
运行时常量池存放 Class 文件中的常量池（存放编译期生成的各种字面量和符号引用）；翻译出来的直接引用；运行期间产生的新的常量（譬如 String 类的 intern() 方法）。

方法区的垃圾收集比较少见，主要针对常量池的回收和类型的卸载。当方法区无法满足内存分配需求时，会抛出 OutOfMemoryError 异常。

5）本地方法栈

与虚拟机栈功能类似，但虚拟机栈为 Java 方法服务，而本地方法栈为 Native 方法服务。也有 StackOverflowError 和 OutOfMemoryError 异常。

（5）直接内存



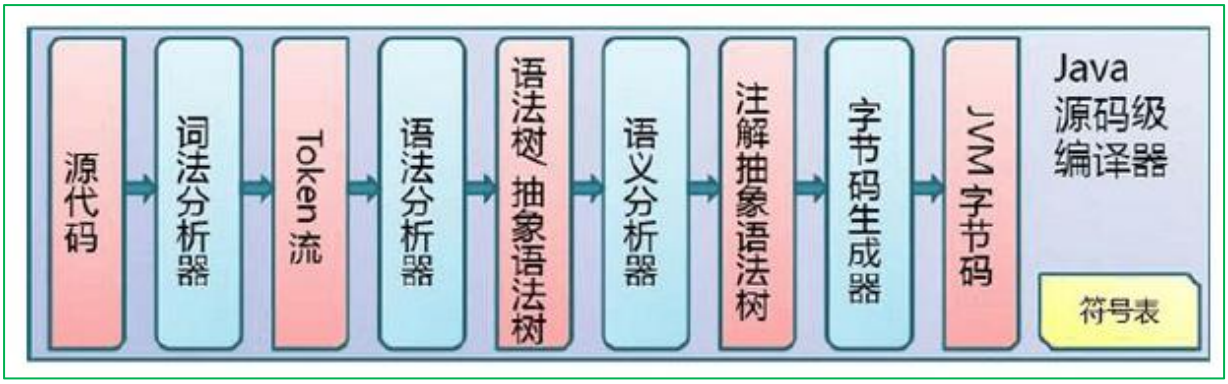
直接内存并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范定义中的内存区域。但这部分区域被频繁使用并可能引起 OutOfMemoryError 异常。

NIO（New Input/Output）类中 ，可用使用 Native 函数库直接分配堆外内存，然后通过一个存储在 java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作，避免了在 Java 堆和 Native 堆中来回复制数据。

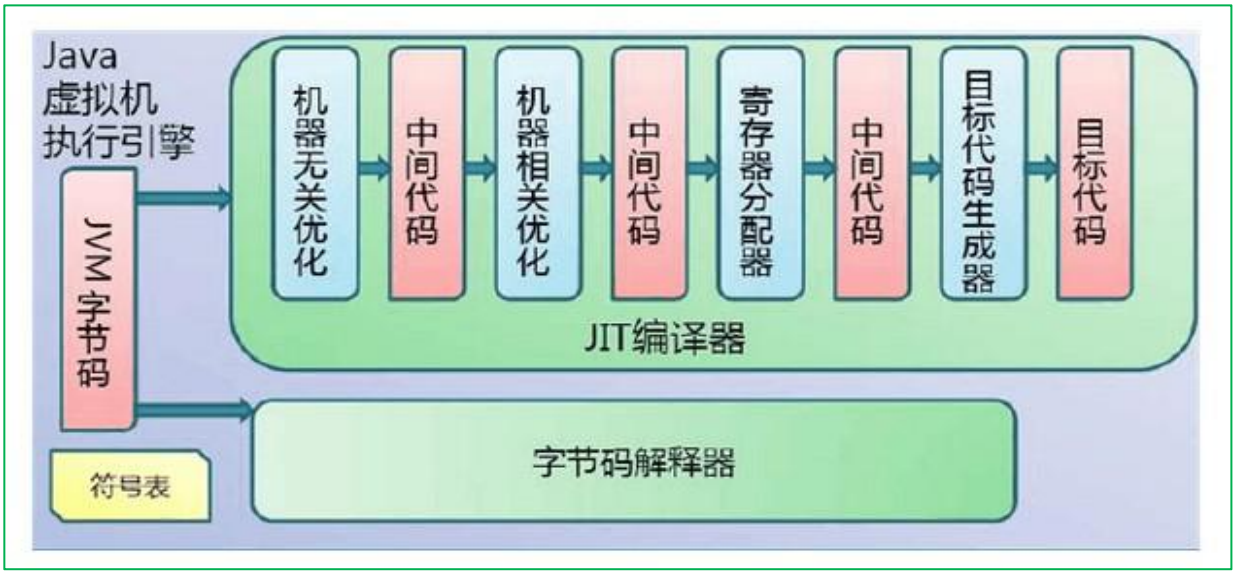
不受 java 堆大小的限制，但受本机总内存的大小及处理器寻址空间的限制，会抛出 OutOfMemoryError 异常。

4. Java 代码编译和执行的整个过程

Java 代码编译是由 Java 源码编译器来完成，流程图如下所示：



Java 字节码的执行是由 JVM 执行引擎来完成，流程图如下所示：



Java 代码编译和执行的整个过程包含了以下三个重要的机制：

（1）Java 源码编译机制

Java 源码编译由以下三个过程组成：分析和输入到符号表、注解处理、语义分析和生成 class 文件；

最后生成的 class 文件由以下部分组成：

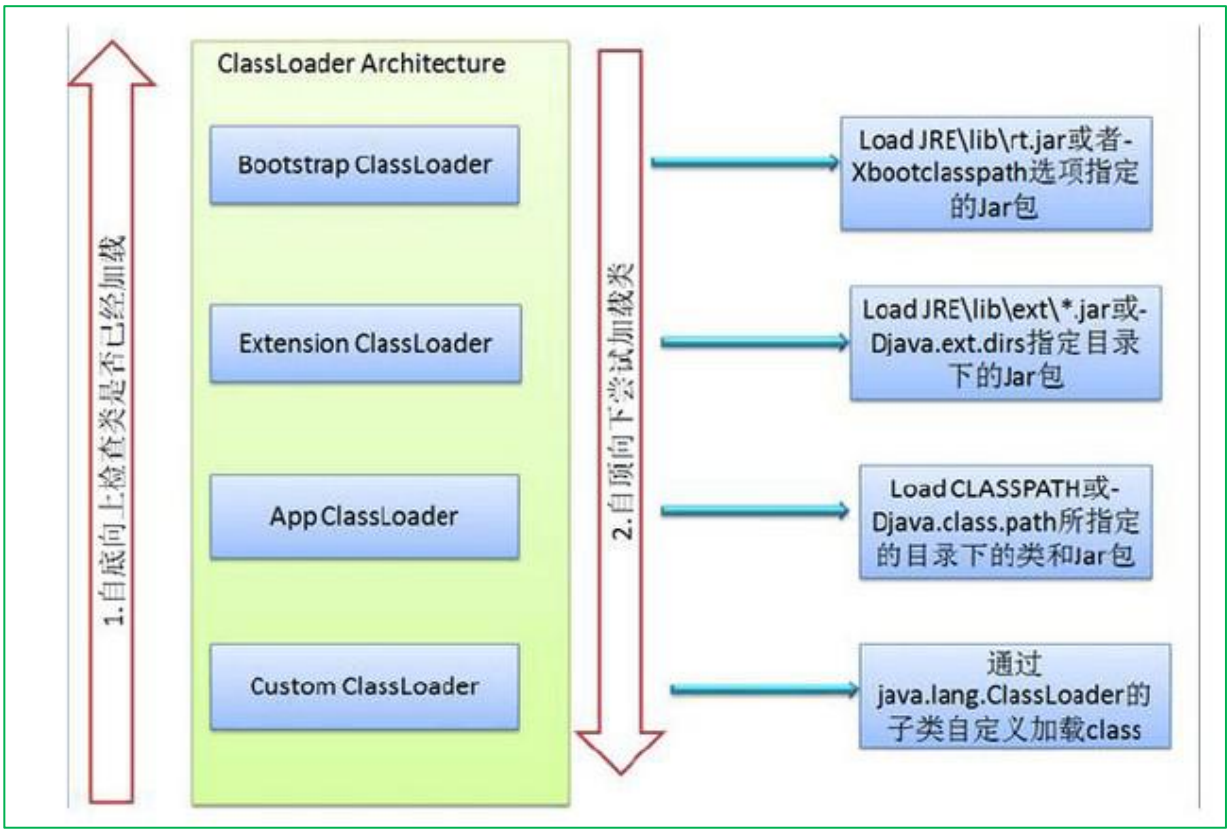
1）结构信息：包括 class 文件格式版本号及各部分的数量与大小的信息；

2）元数据：对应于 Java 源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池；

3）方法信息：对应 Java 源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息。

（2）类加载机制

JVM 的类加载是通过 ClassLoader 及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



1）Bootstrap ClassLoader /启动类加载器

$JAVA_HOME 中 jre/lib/rt.jar 里所有的 class，由 C++实现，不是 ClassLoader 子类

2）Extension ClassLoader/扩展类加载器

负责加载 java 平台中扩展功能的一些 jar 包，包括$JAVA_HOME 中 jre/lib/*.jar 或 -Djava.ext.dirs 指定目录下的 jar 包

3）App ClassLoader/ 系统类加载器

负责记载 classpath 中指定的 jar 包及目录中 class

4）Custom ClassLoader/用户自定义类加载器(java.lang.ClassLoader 的子类)

属于应用程序根据自身需要自定义的 ClassLoader，如 tomcat、jboss 都会根据 j2ee 规范自行实现 ClassLoader

加载过程中会先检查类是否被已加载，**检查顺序是自底向上**，从 Custom ClassLoader 到 BootStrap ClassLoader 逐层检查，只要某个 classloader 已加载就视为已加载此类，保证此类只所有 ClassLoader 加载一次。而**加载顺序是自顶向下**，也就是由上层来逐层尝试加载此类。

（3）类加载双亲委派机制介绍和分析

在这里，需要着重说明的是，JVM 在加载类时默认采用的是双亲委派机制。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

（4）类执行机制

JVM 是基于栈的体系结构来执行 class 字节码的。线程创建后，都会产生程序计数器(PC)和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈

帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。