# JVM 原理讲解和调优

转载 2016 年 12 月 27 日 15:56:34

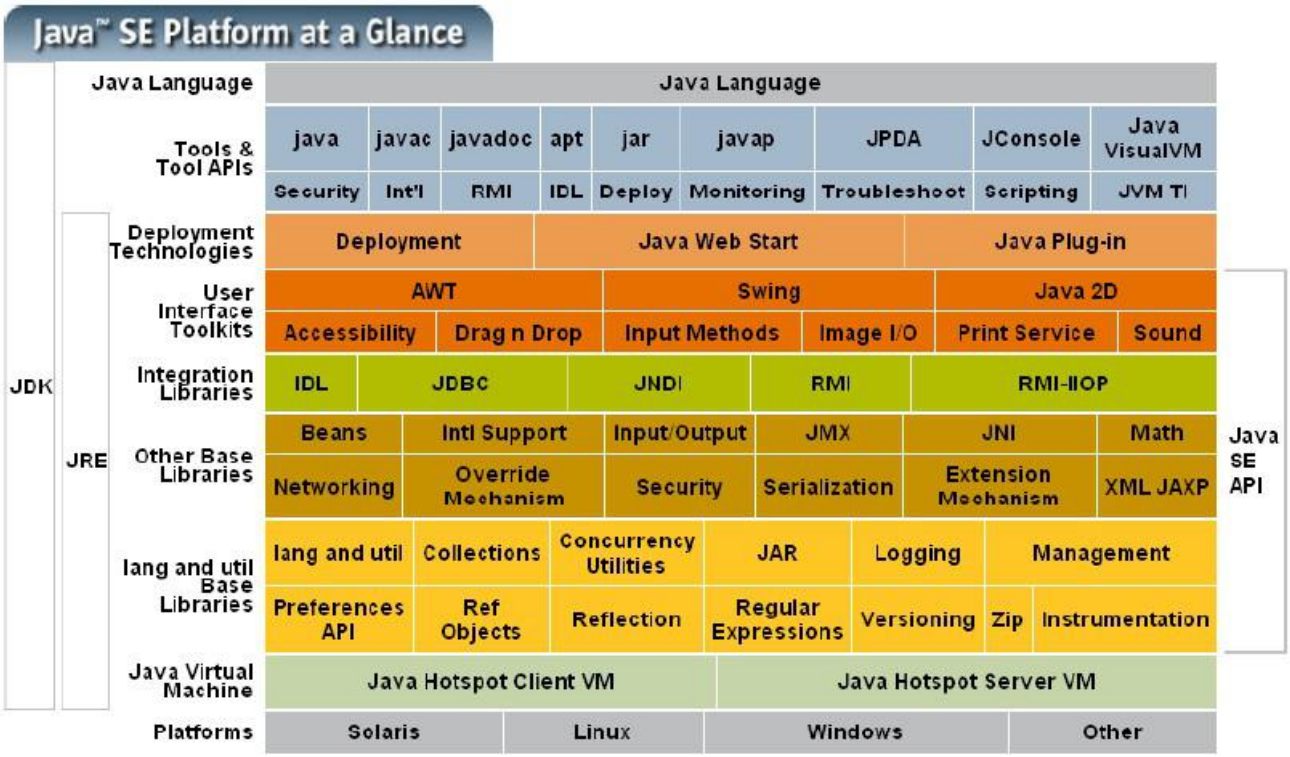
8125

## 一、什么是 JVM

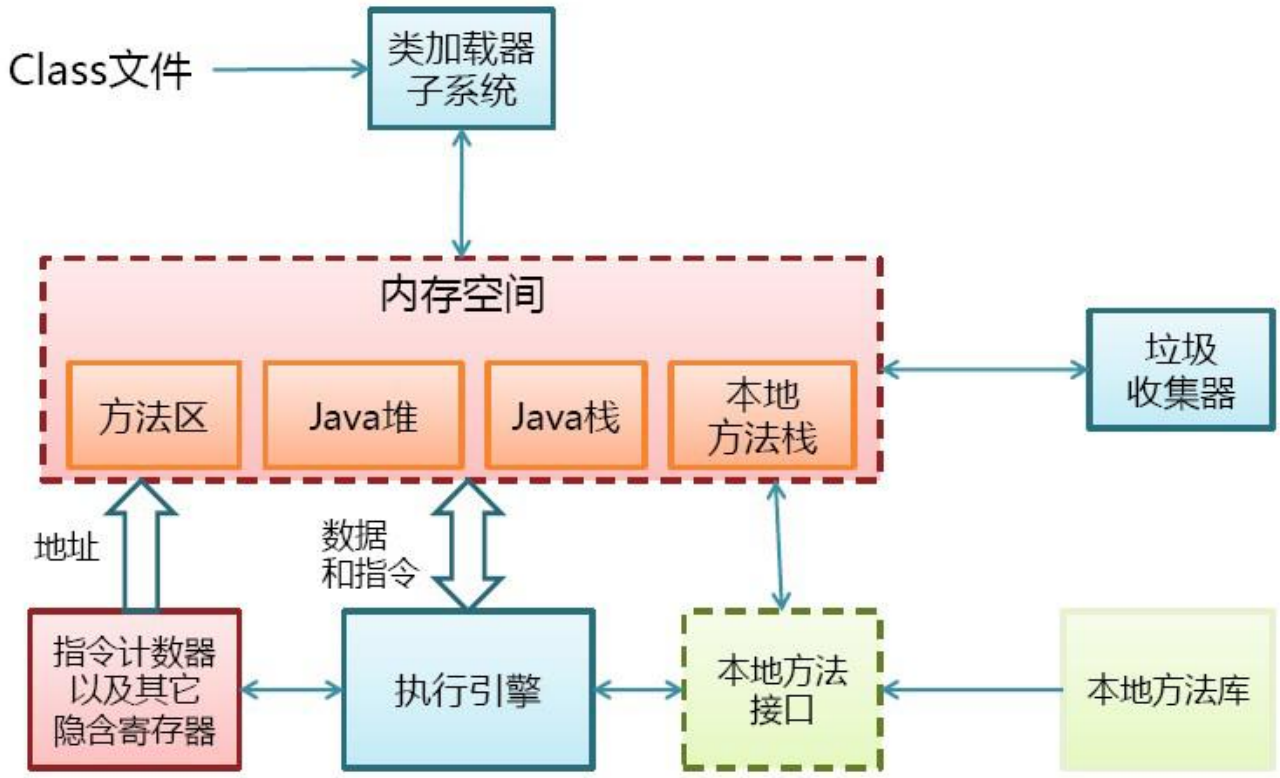
JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，JVM 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是 Java 的能够“一次编译，到处运行”的原因。

从 Java 平台的逻辑结构上来看，我们可以从下图来了解 JVM：

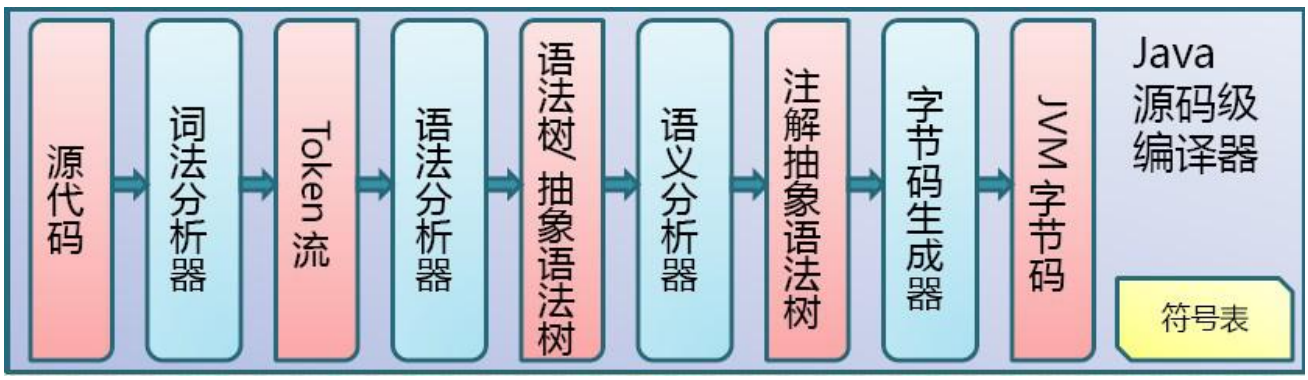



从上图能清晰看到 Java 平台包含的各个逻辑模块，也能了解到 JDK 与 JRE 的区别，对于 JVM 自身的物理结构，我们可以从下图鸟瞰一下：

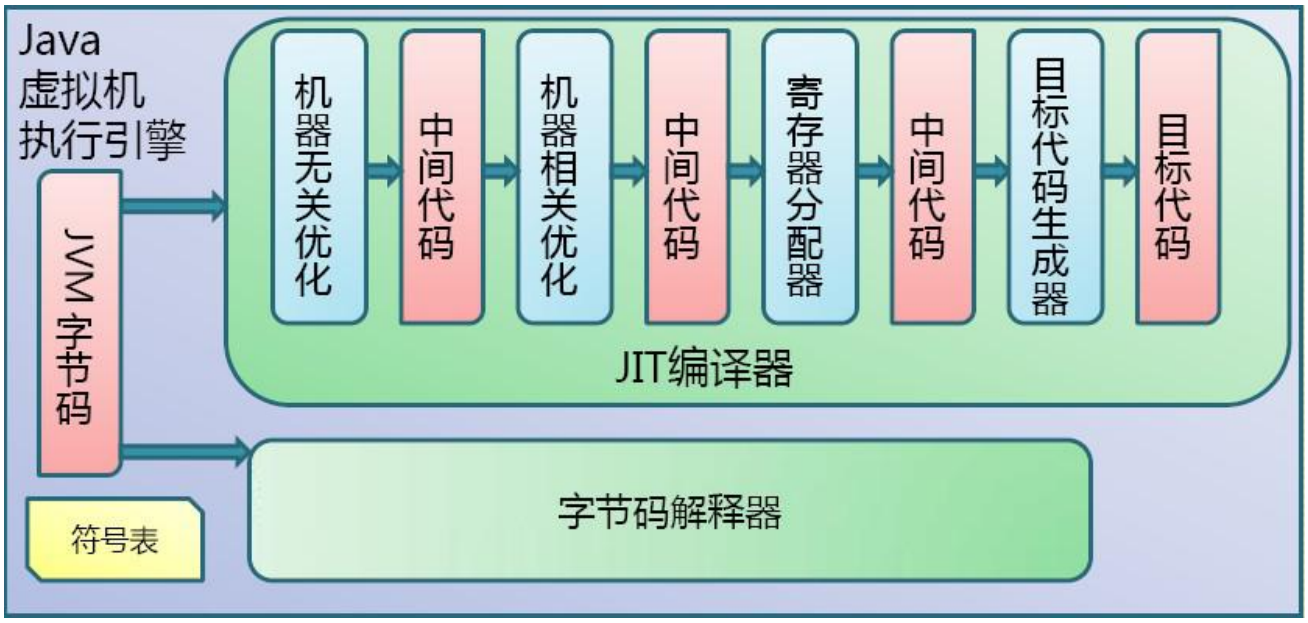



## 二、JAVA 代码编译和执行过程

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：




ava 代码编译和执行的整个过程包含了以下三个重要的机制：

- 

Java 源码编译机制

- 
- 

类加载机制

- 
- 

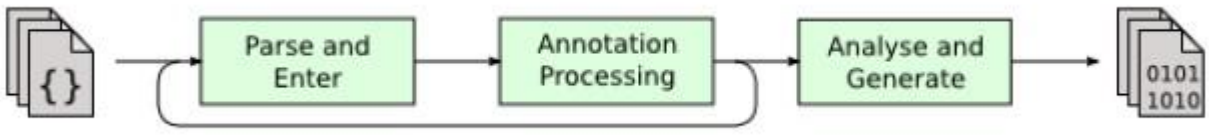
类执行机制

- 

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
- 流程图如下所示：




最后生成的 class 文件由以下部分组成：

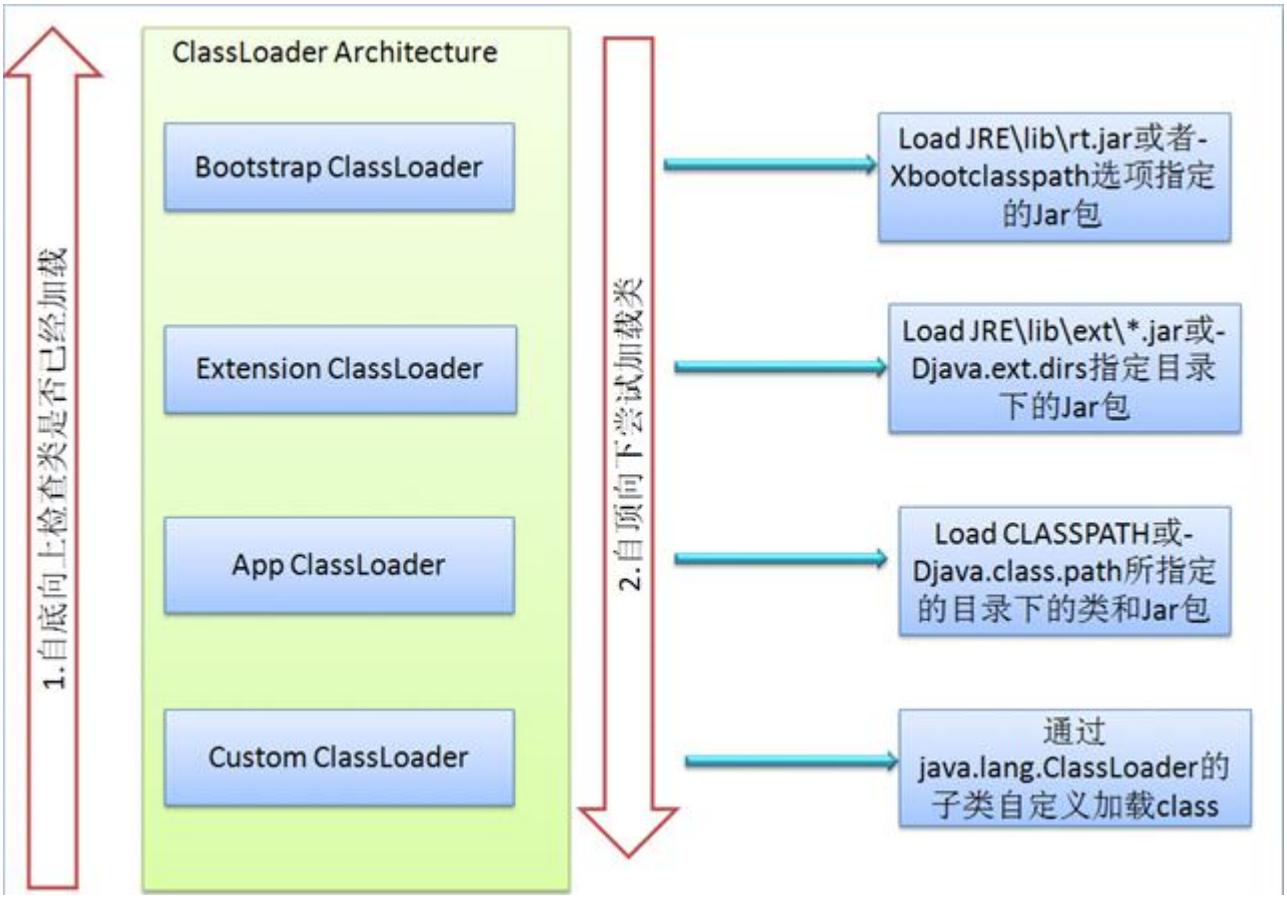
- 结构信息。包括 class 文件格式版本号及各部分的数量与大小的信息

- 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

- 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

- 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

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

2）Extension ClassLoader

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

负责记载 classpath 中指定的 jar 包及目录中 class

4）Custom ClassLoader

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。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

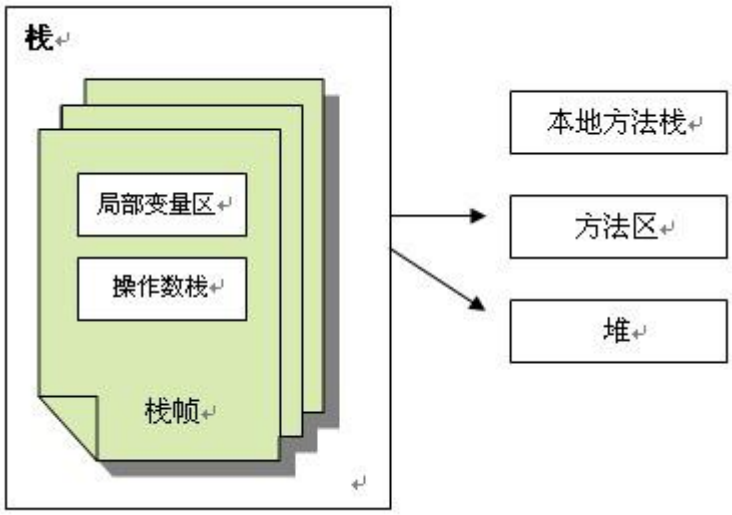
类执行机制

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。栈的结构如下图所示：

# 三、JVM 内存管理和垃圾回收

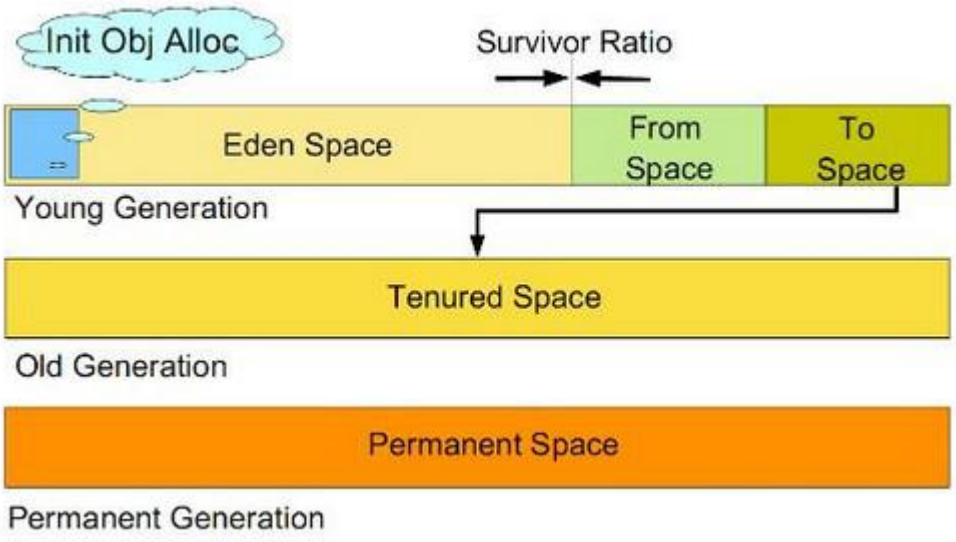
JVM 内存组成结构

JVM 栈由堆、栈、本地方法栈、方法区等部分组成，结构图如下所示：




1）堆

所有通过 new 创建的对象的内存都在堆中分配，堆的大小可以通过-Xmx 和-Xms 来控制。堆被划分为新生代和旧生代，新生代又被进一步划分为 Eden 和 Survivor 区，最后 Survivor 由 From Space 和 To Space 组成，结构图如下所示：




- 

新生代。新建的对象都是用新生代分配内存，Eden 空间不足的时候，会把存活的对象转移到 Survivor 中，新生代大小可以由-Xmn 来控制，也可以用-XX:SurvivorRatio 来控制 Eden 和 Survivor 的比例

- 

- 

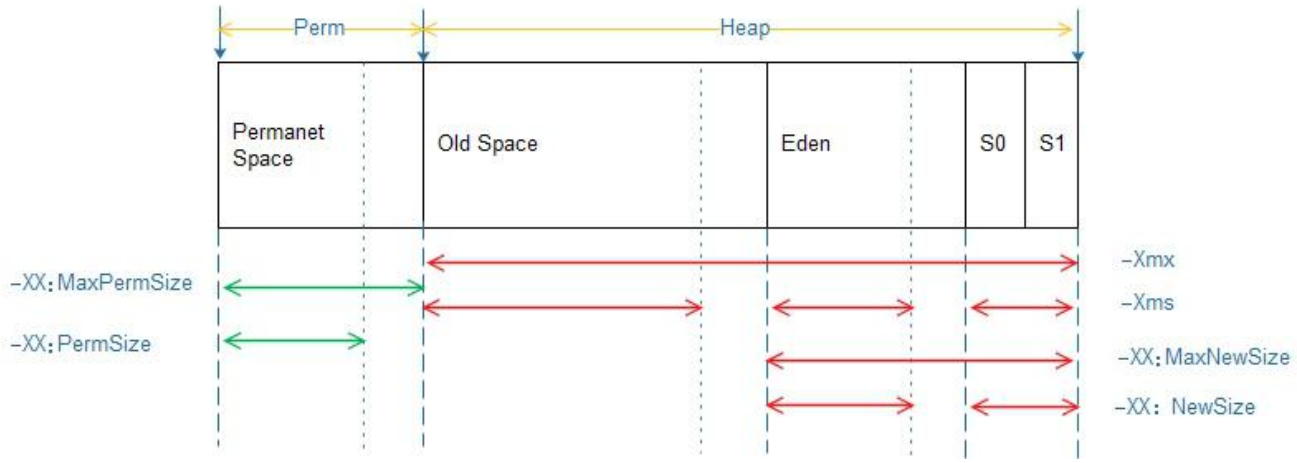
旧生代。用于存放新生代中经过多次垃圾回收仍然存活的对象

- 

- 

持久带（Permanent Space）实现方法区，主要存放所有已加载的类信息，方法信息，常量池等等。可通过-XX:PermSize 和-XX:MaxPermSize 来指定持久带初始化值和最大值。Permanent Space 并不等同于方法区，只不过是 Hotspot JVM 用 Permanent Space 来实现方法区而已，有些虚拟机没有 Permanent Space 而用其他机制来实现方法区。

- 




-Xmx:最大堆内存,如：-Xmx512m

-Xms:初始时堆内存,如：-Xms256m

-XX:MaxNewSize:最大年轻区内存

-XX:NewSize:初始时年轻区内存.通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90%

-XX:MaxPermSize:最大持久带内存

-XX:PermSize:初始时持久带内存

-XX:+PrintGCDetails。打印 GC 信息

-XX:NewRatio 新生代与老年代的比例，如 -XX:NewRatio=2，则新生代占整个堆空间的 1/3，老年代占 2/3

-XX:SurvivorRatio 新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10

2）栈

每个线程执行每个方法的时候都会在栈中申请一个栈帧,每个栈帧包括局部变量区和操作数栈，用于存放此次方法调用过程中的临时变量、参数和中间结果。

-xss:设置每个线程的堆栈大小.  JDK1.5+ 每个线程堆栈大小为 1M,一般来说如果栈不是很深的话， 1M 是绝对够用了的。

3）本地方法栈

用于支持 native 方法的执行，存储了每个 native 方法调用的状态

4）方法区

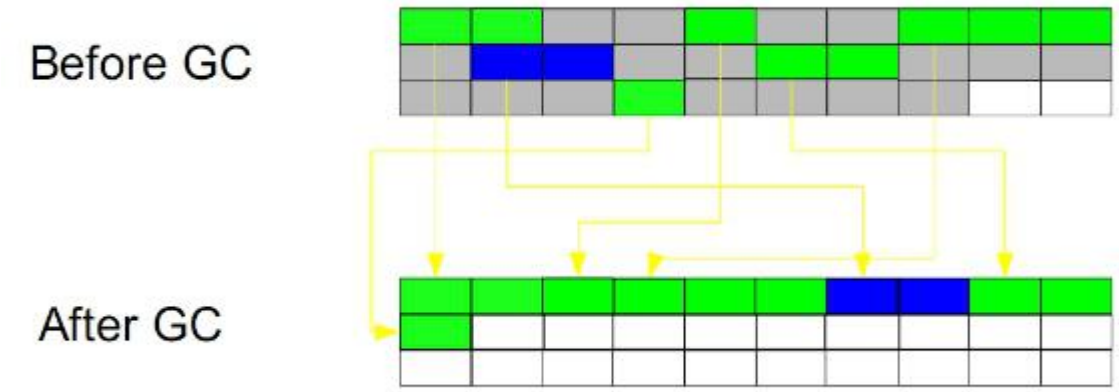
存放了要加载的类信息、静态变量、final 类型的常量、属性和方法信息。JVM 用持久代（Permanet Generation）来存放方法区，可通过-XX:PermSize 和-XX:MaxPermSize 来指定最小值和最大值

## 垃圾回收按照基本回收策略分

引用计数（Reference Counting）：

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。

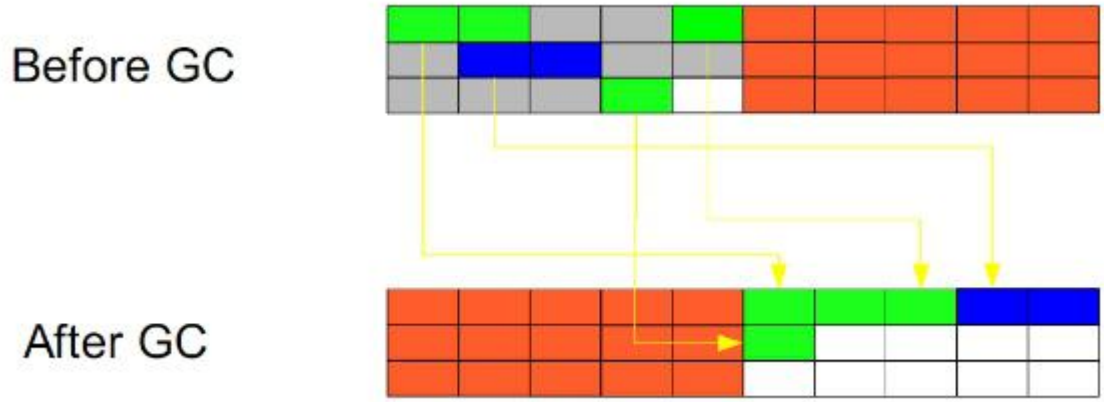
标记-清除（Mark-Sweep）：




---

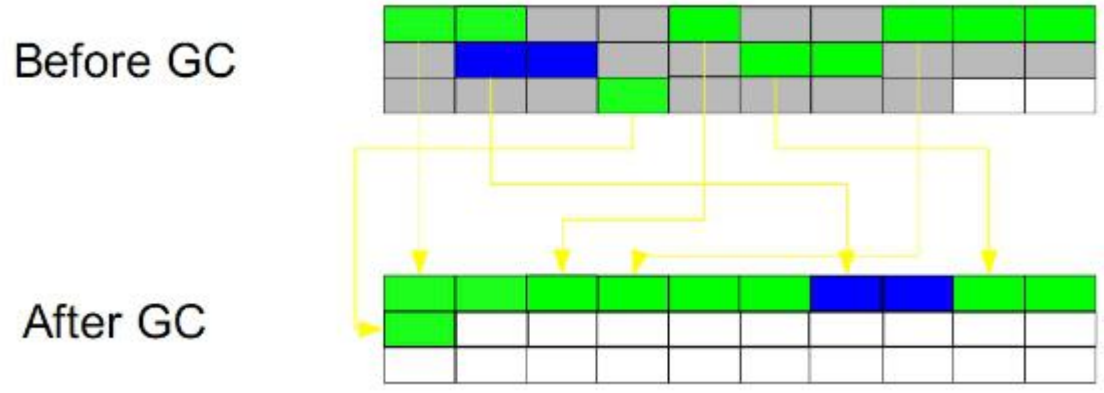
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）：




此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）：



此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

JVM 分别对新生代和旧生代采用不同的垃圾回收机制

新生代的 GC：

新生代通常存活时间较短，因此基于 Copying 算法来进行回收，所谓 Copying 算法就是扫描出存活的对象，并复制到一块新的完全未使用的空间中，对应于新生代，就是在 Eden 和 From Space 或 To Space 之间 copy。新生代采用空闲指针的方式来控制 GC 触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发 GC。当连续分配对象时，对象会逐渐从 eden 到 survivor，最后到旧生代。

在执行机制上 JVM 提供了串行 GC（Serial GC）、并行回收 GC（Parallel Scavenge）和并行 GC（ParNew）

1）串行 GC

在整个扫描和复制过程采用单线程的方式来进行，适用于单 CPU、新生代空间较小及对暂停时间要求不是非常高的应用上，是 client 级别默认的 GC 方式，可以通过-XX:+UseSerialGC 来强制指定

2）并行回收 GC

在整个扫描和复制过程采用多线程的方式来进行，适用于多 CPU、对暂停时间要求较短的应用上，是 server 级别默认采用的 GC 方式，可用-XX:+UseParallelGC 来强制指定，用-XX:ParallelGCThreads=4 来指定线程数

3）并行 GC

与旧生代的并发 GC 配合使用

旧生代的 GC：

旧生代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记（Mark）算法来进行回收，所谓标记就是扫描出存活的对象，然后再进行回收未被标记的对象，回收后对用空出的空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。在执行机制上 JVM 提供了串行 GC（Serial MSC）、并行 GC（parallel MSC）和并发 GC（CMS），具体算法细节还有待进一步深入研究。

以上各种 GC 机制是需要组合使用的，指定方式由下表所示：

| 指定方式 | 新生代 GC 方式 | 旧生代 GC 方式 |
| --- | --- | --- |
| -XX:+UseSerialGC | 串行 GC | 串行 GC |
| -XX:+UseParallelGC | 并行回收 GC | 并行 GC |
| -XX:+UseConeMarkSweepGC | 并行 GC | 并发 GC |
| -XX:+UseParNewGC | 并行 GC | 串行 GC |
| -XX:+UseParallelOldGC | 并行回收 GC | 并行 GC |
| -XX:+ UseConeMarkSweepGC<br>-XX:+UseParNewGC | 串行 GC | 并发 GC |
| 不支持的组合 | 1、-XX:+UseParNewGC -XX:+UseParallelOldGC<br>2、-XX:+UseParNewGC -XX:+UseSerialGC | |

# 四、JVM 内存调优

首先需要注意的是在对 JVM 内存调优的时候不能只看操作系统级别 Java 进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为 GC 过后这个值是不会变化的，因此内存调优的时候要更多地使用 JDK 提供的内存查看工具，比如 JConsole 和 Java VisualVM。

对 JVM 内存的系统级的调优主要的目的是减少 GC 的频率和 Full GC 的次数，过多的 GC 和 Full GC 是会占用很多的系统资源（主要是 CPU），影响系统的吞吐量。特别要关注 Full GC，因为它会对整个堆进行整理，导致 Full GC 一般由于以下几种情况：

旧生代空间不足

调优时尽量让对象在新生代 GC 时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

Pemanet Generation 空间不足

增大 Perm Gen 空间，避免太多静态对象

统计得到的 GC 后晋升到旧生代的平均大小大于旧生代剩余空间

控制好新生代和旧生代的比例

System.gc()被显示调用

垃圾回收不要手动触发，尽量依靠 JVM 自身的机制

调优手段主要是通过控制堆内存的各个部分的比例和 GC 策略来实现，下面来看看各部分比例不良设置会导致什么后果

1）新生代设置过小

一是新生代 GC 次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发 Full GC

2）新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发 Full GC；二是新生代 GC 耗时大幅度增加

一般说来新生代占整个堆 1/3 比较合适

3）Survivor 设置过小

导致对象从 eden 直接到达旧生代，降低了在新生代的存活时间

4）Survivor 设置过大

导致 eden 过小，增加了 GC 频率

另外，通过-XX:MaxTenuringThreshold=n 来控制新生代存活时间，尽量让对象在新生代被回收

由内存管理和垃圾回收可知新生代和旧生代都有多种 GC 策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM 提供两种较为简单的 GC 策略的设置方式

1）吞吐量优先

JVM 以吞吐量为指标，自行选择相应的 GC 策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n 来设置

2）暂停时间优先

JVM 以暂停时间为指标，自行选择相应的 GC 策略及控制新生代与旧生代的大小比例，尽量保证每次 GC 造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n 来设置

最后汇总一下 JVM 常见配置

堆设置

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为 3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4

-XX:SurvivorRatio=n:年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor 区占整个年轻代的 1/5

-XX:MaxPermSize=n:设置持久代大小

收集器设置

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的 CPU 数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

# java 中 JVM 的原理

2014 年 06 月 05 日 10:50:22

120590

## 一、java 虚拟机的生命周期：

Java 虚拟机的生命周期 一个运行中的 Java 虚拟机有着一个清晰的任务：执行 Java 程序。程序开始执行时他才运行，程序结束时他就停止。你在同一台机器上运行三个程序，就会有三个运行中的 Java 虚拟机。 Java 虚拟机总是开始于一个 main()方法，这个方法必须是公有、返回 void、直接受一个字符串数组。在程序执行时，你必须给 Java 虚拟机指明这个包换 main()方法的类名。 Main()方法是程序的起点，他被执行的线程初始化为程序的初始线程。程序中其他的线程都由他来启动。Java 中的线程分为两种：守护线程 （daemon）和普通线程（non-daemon）。守护线程是 Java 虚拟机自己使用的线程，比如负责垃圾收集的线程就是一个守护线程。当然，你也可 以把自己的程序设置为守护线程。包含 Main()方法的初始线程不是守护线程。 只要 Java 虚拟机中还有普通的线程在执行，Java 虚拟机就不会停止。如果有足够的权限，你可以调用 exit()方法终止程序。

## 二、java 虚拟机的体系结构：

在 Java 虚拟机的规范中定义了一系列的子系统、内存区域、数据类型和使用指南。这些组件构成了 Java 虚拟机的内部结构，他们不仅仅为 Java 虚拟机的实现提供了清晰的内部结构，更是严格规定了 Java 虚拟机实现的外部行为。

每一个 Java 虚拟机都由一个类加载器子系统（class loader subsystem），负责加载程序中的类型（类和接口），并赋予唯一的名字。每一个 Java 虚拟机都有一个执行引擎（execution engine）负责执行被加载类中包含的指令。

程序的执行需要一定的内存空间，如字节码、被加载类的其他额外信息、程序中的对象、方法的参数、返回值、本地变量、处理的中间变量等等。Java 虚拟机将 这些信息统统保存在数据区（data areas）中。虽然每个 Java 虚拟机的实现中都包含数据区，但是 Java 虚拟机规范对数据区的规定却非常的抽象。许多结构上的细节部分都留给了 Java 虚拟机实现者自己发挥。不同 Java 虚拟机实现上的内存结构千差万别。一部分实现可能占用很多内存，而其他以下可能只占用很少的内存；一些实现可 能会使用虚拟内存，而其他的则不使用。这种比较精炼的 Java 虚拟机内存规约，可以使得 Java 虚拟机可以在广泛的平台上被实现。

数据区中的一部分是整个程序共有，其他部分被单独的线程控制。每一个 Java 虚拟机都包含方法区（method area）和堆（heap），他们都被整个程序共享。Java 虚拟机加载并解析一个类以后，将从类文件中解析出来的信息保存与方法区中。程序执行时创建的 对象都保存在堆中。

当一个线程被创建时，会被分配只属于他自己的 PC 寄存器“pc register”（程序计数器）和 Java 堆栈（Java stack）。当线程不掉用本地方法时，PC 寄存器中保存线程执行的下一条指令。Java 堆栈保存了一个线程调用方法时的状态，包括本地变量、调用方法的参数、返回值、处理的中间变量。调用本地方法时的状态保存在本地方法堆栈中（native method stacks），可能再寄存器或者其他非平台独立的内存中。

Java 堆栈有堆栈块（stack frames（or frames））组成。堆栈块包含 Java 方法调用的状态。当一个线程调用一个方法时，Java 虚拟机会将一个新的块压到 Java 堆栈中，当这个方法运行结束时，Java 虚拟机会将对应的块弹出并抛弃。

Java 虚拟机不使用寄存器保存计算的中间结果，而是用 Java 堆栈在存放中间结果。这是的 Java 虚拟机的指令更紧凑，也更容易在一个没有寄存器的设备上实现 Java 虚拟机。

图中的 Java 堆栈中向下增长的，PC 寄存器中线程三为灰色，是因为它正在执行本地方法，他的下一条执行指令不保存在 PC 寄存器中。

## 三、类加载器子系统：

Java 虚拟机中的类加载器分为两种：原始类加载器（primordial class loader）和类加载器对象（class loader objects）。原始类加载器是 Java 虚拟机实现的一部分，类加载器对象是运行中的程序的一部分。不同类加载器加载的类被不同的命名空间所分割。

类加载器调用了许多 Java 虚拟机中其他的部分和 java.lang 包中的很多类。比如，类加载对象就是 java.lang.ClassLoader 子类 的实例，ClassLoader 类中的方法可以访问虚拟机中的类加载机制；每一个被 Java 虚拟机加载的类都会被表示为一个 java.lang.Class

类的实例。像其他对象一样，类加载器对象和 Class 对象都保存在堆中，被加载的信息被保存在方法区中。

1、加载、连接、初始化（Loading, Linking and Initialization）

类加载子系统不仅仅负责定位并加载类文件，他按照以下严格的步骤作了很多其他的事情：（具体的信息参见第七章的“类的生命周期”）

1）、加载：寻找并导入指定类型（类和接口）的二进制信息

2）、连接：进行验证、准备和解析

①验证：确保导入类型的正确性

②准备：为类型分配内存并初始化为默认值

③解析：将字符引用解析为直接饮用

3）、初始化：调用 Java 代码，初始化类变量为合适的值

2、原始类加载器（The Primordial Class Loader）

每个 Java 虚拟机都必须实现一个原始类加载器，他能够加载那些遵守类文件格式并且被信任的类。但是，Java 虚拟机的规范并没有定义如何加载类，这由 Java 虚拟机实现者自己决定。对于给定类型名的类型，原始莱加载器必须找到那个类型名加“.class”的文件并加载入虚拟机中。

3、类加载器对象

虽然类加载器对象是 Java 程序的一部分，但是 ClassLoader 类中的三个方法可以访问 Java 虚拟机中的类加载子系统。

1）、protected final Class defineClass(…)：使用这个方法可以出入一个字节数组，定义一个新的类型。

2）、protected Class findSystemClass(String name)：加载指定的类，如果已经加载，就直接返回。

3）、protected final void resolveClass(Class c)：defineClass()方法只是加载一个类，这个方法负责后续的动态连接和初始化。

具体的信息，参见第八章“连接模型”（ The Linking Model）。

4、命名空间

当多个类加载器加载了同一个类时，为了保证他们名字的唯一性，需要在类名前加上加载该类的类加载器的标识。具体的信息，参见第八章“连接模型”（ The Linking Model）。

## 四、方法区：

在 Java 虚拟机中，被加载类型的信息都保存在方法区中。这写信息在内存中的组织形式由虚拟机的实现者定义，比如，虚拟机工作在一个“little- endian”的处理器上，他就可以将信息保存为“little-endian”格式的，虽然在 Java 类文件中他们是以“big-endian”格式保 存的。设计者可以用最适合并地机器的表示格式来存储数据，以保证程序能够以最快

的速度执行。但是，在一个只有很小内存的设备上，虚拟机的实现者就不会占用 很大的内存。

程序中的所有线程共享一个方法区，所以访问方法区信息的方法必须是线程安全的。如果你有两个线程都去加载一个叫 Lava 的类，那只能由一个线程被容许去加载这个类，另一个必须等待。

在程序运行时，方法区的大小是可变的，程序在运行时可以扩展。有些 Java 虚拟机的实现也可以通过参数也订制方法区的初始大小，最小值和最大值。

方法区也可以被垃圾收集。因为程序中的内由类加载器动态加载，所有类可能变成没有被引用（unreferenced）的状态。当类变成这种状态时，他就可 能被垃圾收集掉。没有加载的类包括两种状态，一种是真正的没有加载，另一个种是“unreferenced”的状态。详细信息参见第七章的类的生命周期 （The Lifetime of a Class）。

1、类型信息（Type Information）

每一个被加载的类型，在 Java 虚拟机中都会在方法区中保存如下信息：

1）、类型的全名（The fully qualified name of the type）

2）、类型的父类型的全名（除非没有父类型，或者弗雷形式 java.lang.Object）（The fully qualified name of the typeís direct superclass）

3）、给类型是一个类还是接口（class or an interface）（Whether or not the type is a class ）

4）、类型的修饰符（public，private，protected，static，final，volatile，transient 等）（The typeís modifiers）

5）、所有父接口全名的列表（An ordered list of the fully qualified names of any direct superinterfaces）

类型全名保存的数据结构由虚拟机实现者定义。除此之外，Java 虚拟机还要为每个类型保存如下信息：

1）、类型的常量池（The constant pool for the type）

2）、类型字段的信息（Field information）

3）、类型方法的信息（Method information）

4）、所有的静态类变量（非常量）信息（All class (static) variables declared in the type, except constants）

5）、一个指向类加载器的引用（A reference to class ClassLoader）

6）、一个指向 Class 类的引用（A reference to class Class）

1）、类型的常量池（The constant pool for the type）

常量池中保存中所有类型是用的有序的常量集合，包含直接常量（literals）如字符串、整数、浮点数的常量，和对类型、字段、方法的符号引用。常量池 中每一个保存的常量都有一个索引，就像数组中的字段一样。因为常量池中保存中所有类型使用到的类型、

字段、方法的字符引用，所以它也是动态连接的主要对 象。详细信息参见第六章“The Java Class File”。

2）、类型字段的信息（Field information）

字段名、字段类型、字段的修饰符（public，private，protected，static，final，volatile，transient 等）、字段在类中定义的顺序。

3）、类型方法的信息（Method information）

方法名、方法的返回值类型（或者是 void）、方法参数的个数、类型和他们的顺序、字段的修饰符（public，private，protected，static，final，volatile，transient 等）、方法在类中定义的顺序

如果不是抽象和本地本法还需要保存

方法的字节码、方法的操作数堆栈的大小和本地变量区的大小（稍候有详细信息）、异常列表（详细信息参见第十七章“Exceptions”。）

4）、类（静态）变量（Class Variables）

类变量被所有类的实例共享，即使不通过类的实例也可以访问。这些变量绑定在类上（而不是类的实例上），所以他们是类的逻辑数据的一部分。在 Java 虚拟机使用这个类之前就需要为类变量（non-final）分配内存

常量（final）的处理方式于这种类变量（non-final）不一样。每一个类型在用到一个常量的时候，都会复制一份到自己的常量池中。常量也像类变 量一样保存在方法区中，只不过他保存在常量池中。（可能是，类变量被所有实例共享，而常量池是每个实例独有的）。Non-final 类变量保存为定义他的 类型数据（data for the type that declares

them）的一部分，而 final 常量保存为使用他的类型数据（data for any type that uses them）的一部分。详情参见第六章“The Java Class FileThe Java Class File”

5）、指向类加载器的引用（A reference to class ClassLoader）

每一个被 Java 虚拟机加载的类型，虚拟机必须保存这个类型是否由原始类加载器或者类加载器加载。那些被类加载器加载的类型必须保存一个指向类加载器的引 用。当类加载器动态连接时，会使用这条信息。当一个类引用另一个类时，虚拟机必须保存那个被引用的类型是被同一个类加载器加载的，这也是虚拟机维护不同命 名空间的过程。详情参见第八章“The Linking Model”

6）、指向 Class 类的引用（A reference to class Class）

Java 虚拟机为每一个加载的类型创建一个 java.lang.Class 类的实例。你也可以通过 Class 类的方法：public static Class forName(String className)来查找或者加载一个类，并取得相应的 Class 类的实例。通过这个 Class 类的实例，我们可以访问 Java 虚拟机方法区中的信息。具体参照 Class 类的 JavaDoc。

2、方法列表（Method Tables）

为了更有效的访问所有保存在方法区中的数据，这些数据的存储结构必须经过仔细的设计。所有方法区中，除了保存了上边的那些原始信息外，还有一个为了加快存 取速度而设计的数据结构，比如方法列表。每一个被加载的非抽象类，Java 虚拟机都会为他们产生一个方法列表，这个列表中保存了这个类可能调用的所有实例 方法的引用，报错那些父类中调用的方法。详情参见第八章“The Linking Model”

## 五、堆：

当 Java 程序创建一个类的实例或者数组时，都在堆中为新的对象分配内存。虚拟机中只有一个堆，所有的线程都共享他。

1、垃圾收集（Garbage Collection）

垃圾收集是释放没有被引用的对象的主要方法。它也可能会为了减少堆的碎片，而移动对象。在 Java 虚拟机的规范中没有严格定义垃圾收集，只是定义一个 Java 虚拟机的实现必须通过某种方式管理自己的堆。详情参见第九章“Garbage Collection”。

2、对象存储结构（Object Representation）

Java 虚拟机的规范中没有定义对象怎样在堆中存储。每一个对象主要存储的是他的类和父类中定义的对象变量。对于给定的对象的引用，虚拟机必须嫩耨很快的 定位到这个对象的数据。另为，必须提供一种通过对象的引用方法对象数据的方法，比如方法区中的对象的引用，所以一个对象保存的数据中往往含有一个某种形式 指向方法区的指针。

一个可能的堆的设计是将堆分为两个部分：引用池和对象池。一个对象的引用就是指向引用池的本地指针。每一个引用池中的条目都包含两个部分：指向对象池中对 象数据的指针和方法区中对象类数据的指针。这种设计能够方便 Java 虚拟机堆碎片的整理。当虚拟机在对象池中移动一个对象的时候，只需要修改对应引用池中 的指针地址。但是每次访问对象的数据都需要处理两次指针。下图演示了这种堆的设计。在第九章的“垃圾收集”中的 HeapOfFish Applet 演示了这种设计。

另一种堆的设计是：一个对象的引用就是一个指向一堆数据和指向相应对象的偏移指针。这种设计方便了对象的访问，可是对象的移动要变的异常复杂。下图演示了这种设计

当程序试图将一个对象转换为另一种类型时，虚拟机需要判断这种转换是否是这个对象的类型，或者是他的父类型。当程序适用 instanceof 语句的时候也 会做类似的事情。当程序调用一个对象的方法时，虚拟机需要进行动态绑定，他必须判断调用哪一个类型的方法。这也需要做上面的判断。

无论虚拟机实现者使用哪一种设计，他都可能为每一个对象保存一个类似方法列表的信息。因为他可以提升对象方法调用的速度，对提升虚拟机的性能非常重要，但 是虚拟机的规范中比没有要求必须实现类似的数据结构。下图描述了这种结构。图中显示了一个对象引用相关联的所有的数据结构，包括：

1）、一个指向类型数据的指针

2）、一个对象的方法列表。方法列表是一个指向所有可能被调用对象方法的指针数组。方法数据包括三个部分：操作码堆栈的大小和方法堆栈的本地变量区；方法的字节码；异常列表。

每一个 Java 虚拟机中的对象必须关联一个用于同步多线程的 lock(mutex)。同一时刻，只能有一个对象拥有这个对象的锁。当一个拥有这个这个对象 的锁，他就可以多次申请这个锁，但是也必须释放相应次数的锁才能真正释放这个对象锁。很多对象在整个生命周期中都不会被锁，所以这个信息只有在需要时才需 要添加。很多 Java 虚拟机的实现都没有在对象的数据中包含“锁定数据”，只是在需要时才生成相应的数据。除了实现对象的锁定，每一个对象还逻辑关联到一 个“wait set”的实现。锁定帮组线程独立处理共享的数据，不需要妨碍其他的线程。“wait set”帮组线程协作完成同一个目标。“wait set”往往通过 Object 类的 wait()和 notify()方法来实现。

垃圾收集也需要堆中的对象是否被关联的信息。Java 虚拟机规范中指出垃圾收集一个运行一个对象的 finalizer 方法一次，但是容许 finalizer 方法重新引用这个对象，当这个对象再次不被引用时，就不需要再次调用 finalize 方法。所以虚拟机也需要保存 finalize 方法 是否运行过的信息。更多信息参见第九章的“垃圾收集”

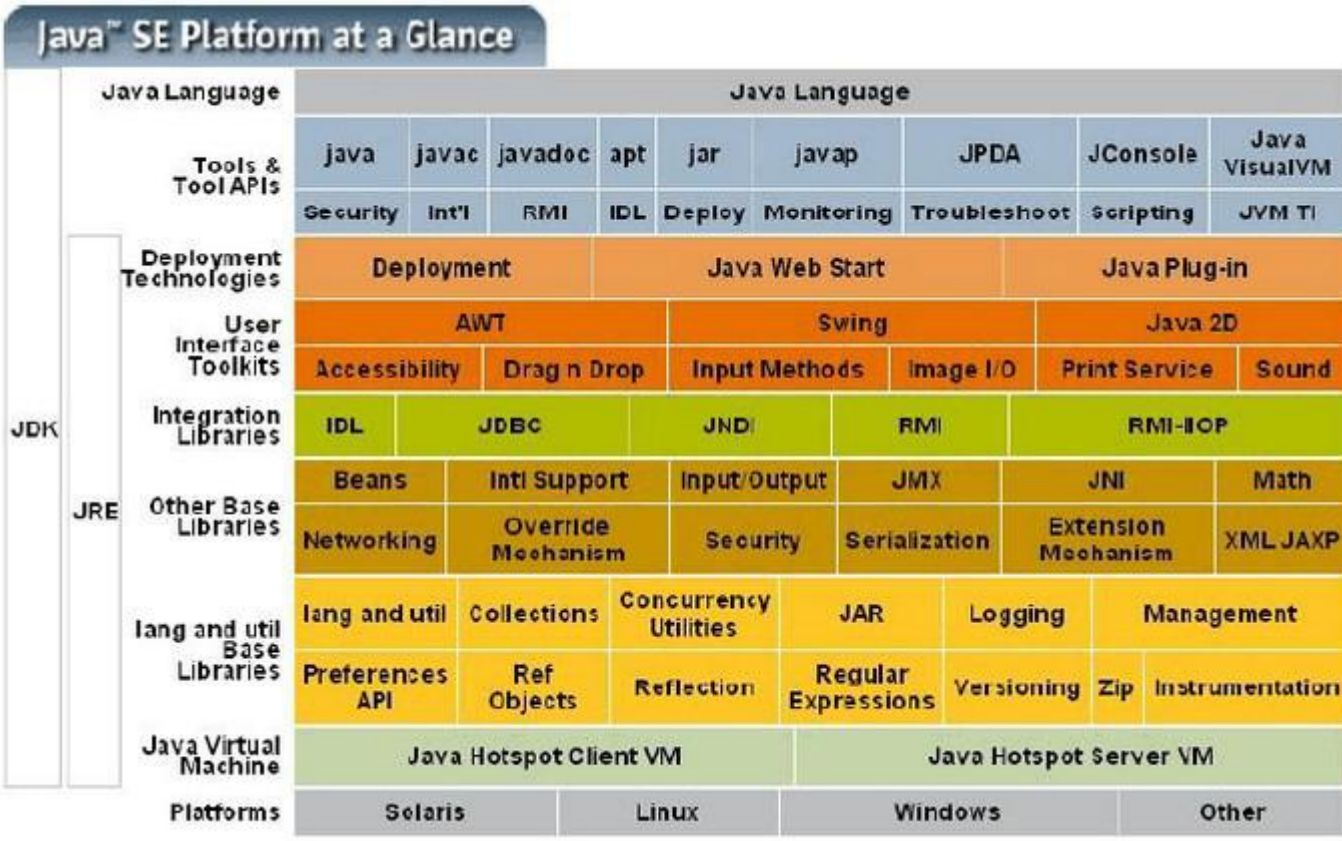
3、数组的保存（Array Representation）

在 Java 中，数组是一种完全意义上的对象，他和对象一样保存在堆中、有一个指向 Class 类实例的引用。所有同一维度和类型的数组拥有同样的 Class，数组的长 度不做考虑。对应 Class 的名字表示为维度和类型。比如一个整型数据的 Class 为“[I”，字节型三维数组 Class 名为“[[[B”，两维对象数据 Class 名为“[[Ljava.lang.Object”。

数组必须在堆中保存数组的长度，数组的数据和一些对象数组类型数据的引用。通过一个数组引用的，虚拟机应该能够取得一个数组的长度，通过索引能够访问特定 的数据，能够调用 Object 定义的方法。Object 是所有数据类的直接父类。更多信息参见第六章“类文件”。

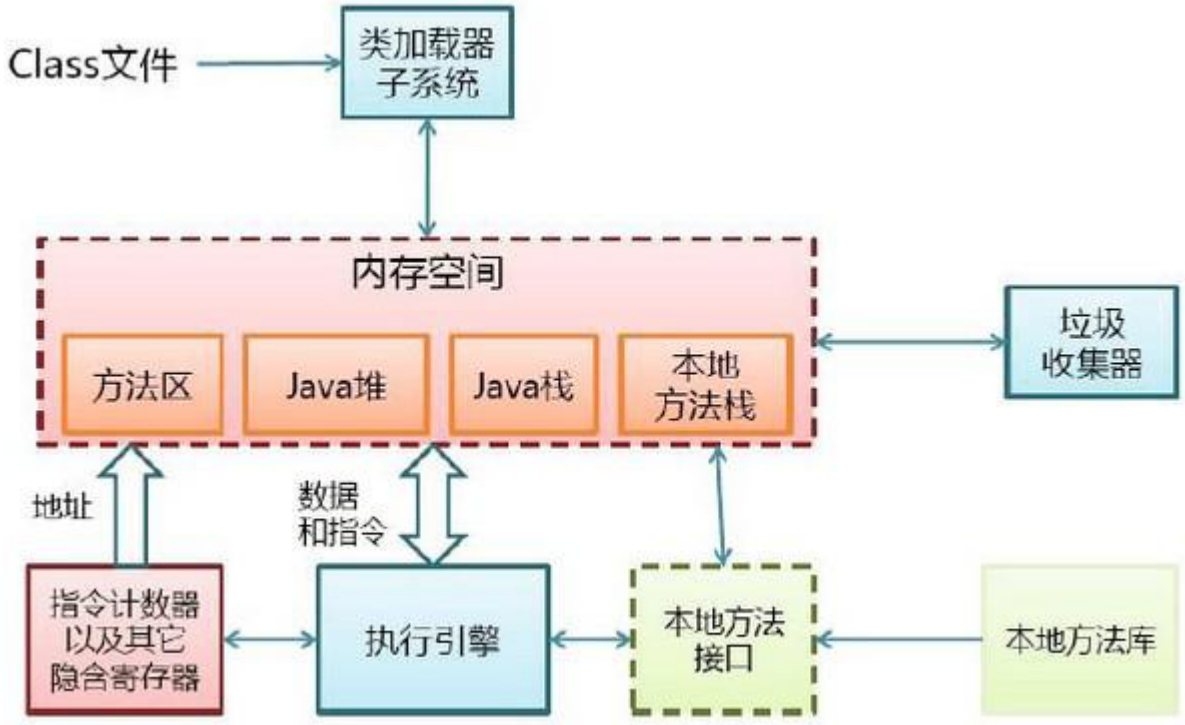
## 六、基本结构：

从 Java 平台的逻辑结构上来看，我们可以从下图来了解 JVM：




从上图能清晰看到 Java 平台包含的各个逻辑模块，也能了解到 JDK 与 JRE 的区别。

JVM 自身的物理结构




此图看出 jvm 内存结构

JVM 内存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 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。

**Classloader 子系统的作用：**

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 Runtimedataarea 中的 methodarea(方法区域)。Java 程序员可以 extendsjava.lang.ClassLoader 类来写自己的 Classloader。

**Executionengine 子系统的作用：**

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 Execution engine 的好坏。

**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。

**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、Heap(堆)：一个 Java 虚拟实例中只存在一个堆空间

2、MethodArea(方法区域)：被装载的 class 的信息存储在 Methodarea 的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的 class 文件，然后读入这个 class 文件内容并把它传输到虚拟机中。

3、JavaStack(java 的栈)：虚拟机只会直接对 Javastack 执行两种操作：以帧为单位的压栈或出栈

4、ProgramCounter(程序计数器)：每一个线程都有它自己的 PC 寄存器，也是该线程启动时创建的。PC 寄存器的内容总是指向下一条将被执行指令的饿地址，这里的地址可以是一个本地指针，也可以是在方法区中相对应于该方法起始指令的偏移量。

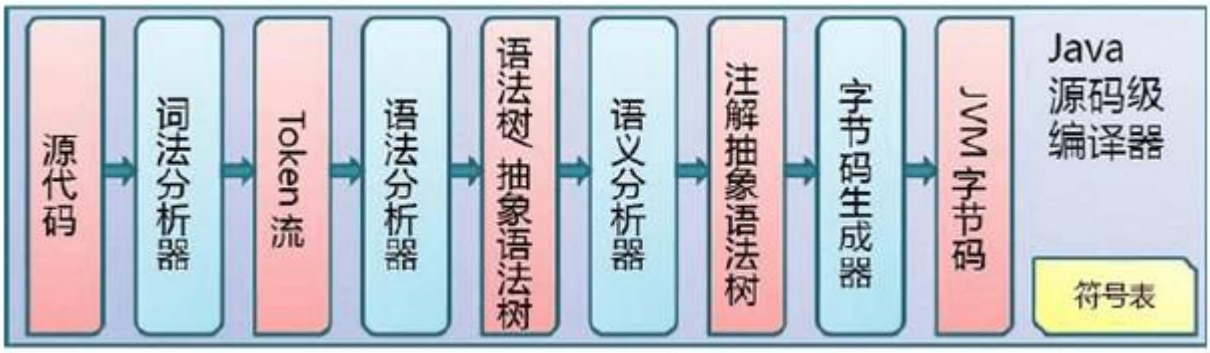
5、Nativemethodstack(本地方法栈)：保存 native 方法进入区域的地址

对于 JVM 的学习，在我看来这么几个部分最重要：

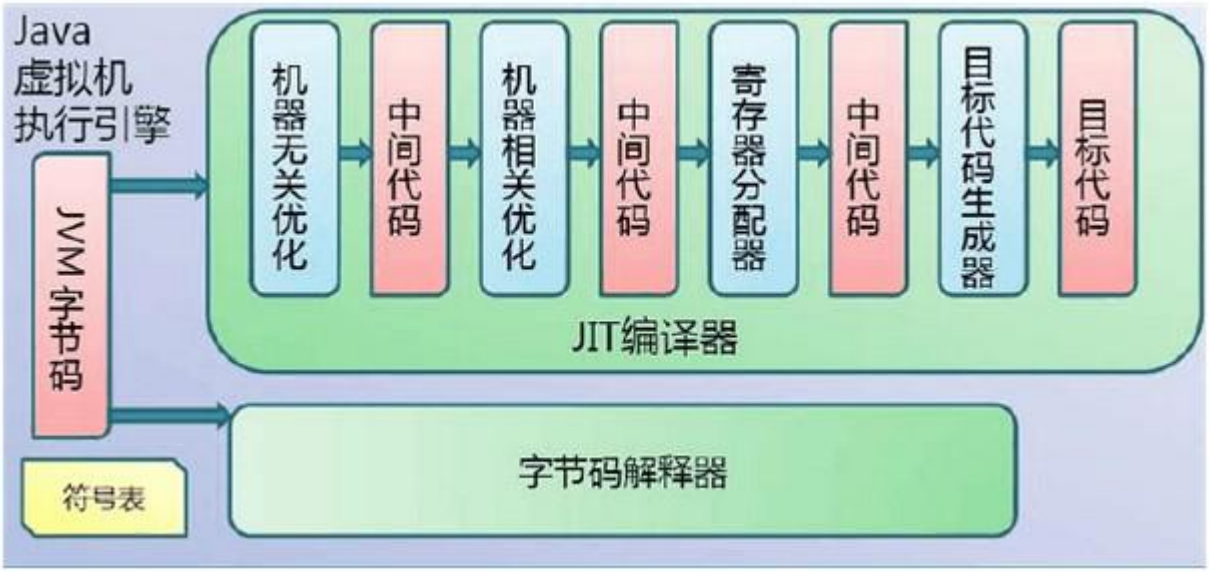
- Java 代码编译和执行的整个过程
- JVM 内存管理及垃圾回收机制

## 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：



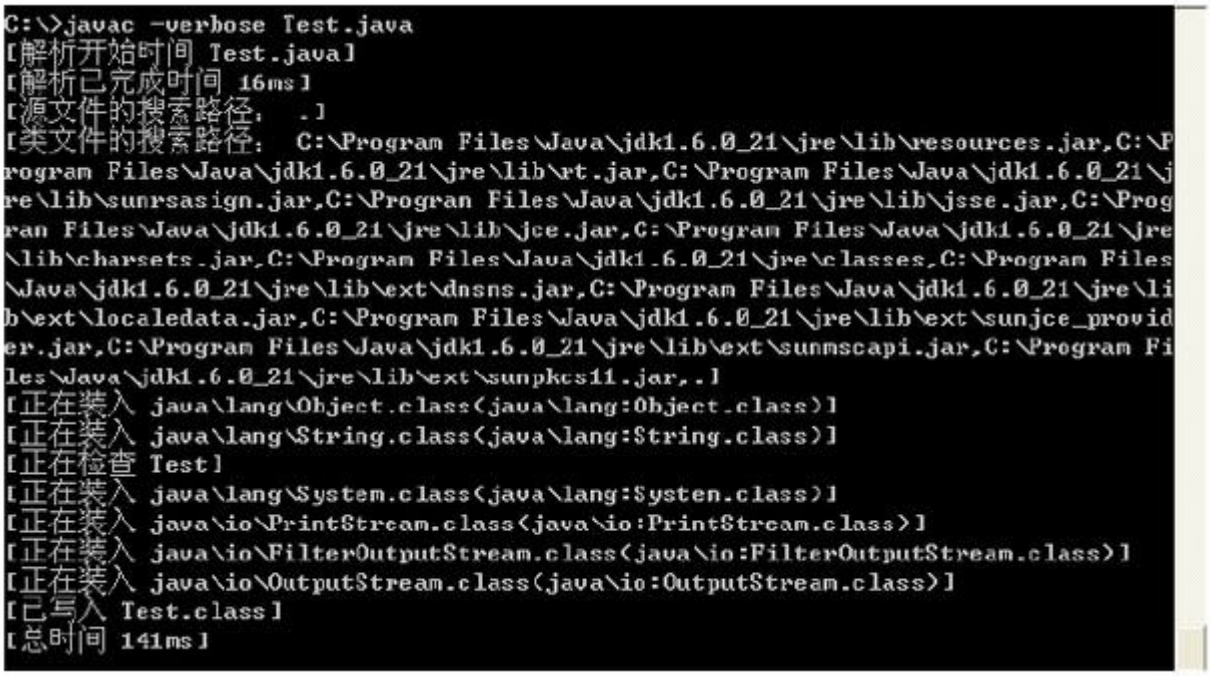

Java 代码编译和执行的整个过程包含了以下三个重要的机制：

- Java 源码编译机制
- 类加载机制
- 类执行机制

# Java 源码编译机制

Java 源码编译由以下三个过程组成：（javac -verbose 输出有关编译器正在执行的操作的消息）

- 分析和输入到符号表
- 注解处理
- 语义分析和生成 class 文件



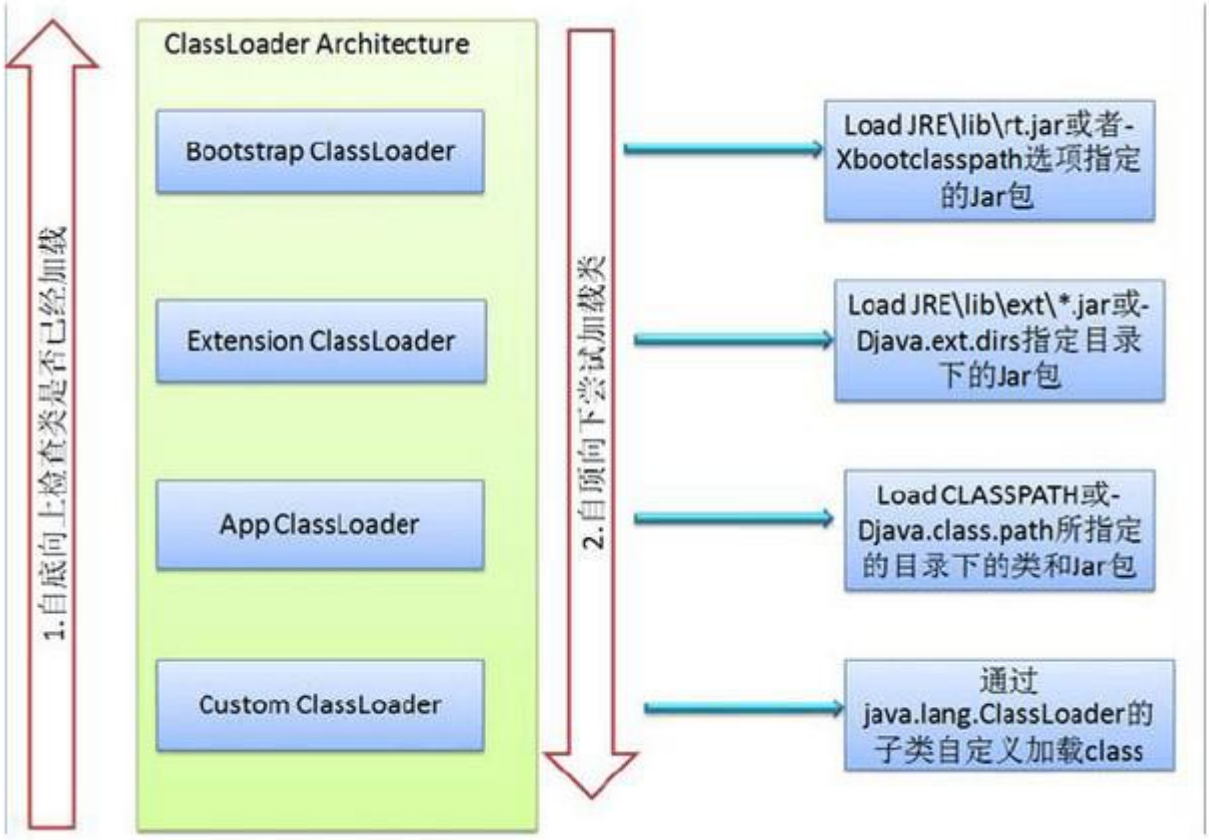

最后生成的 class 文件由以下部分组成：

- 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
- 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
- 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

、

# 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-D java.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。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

## 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。

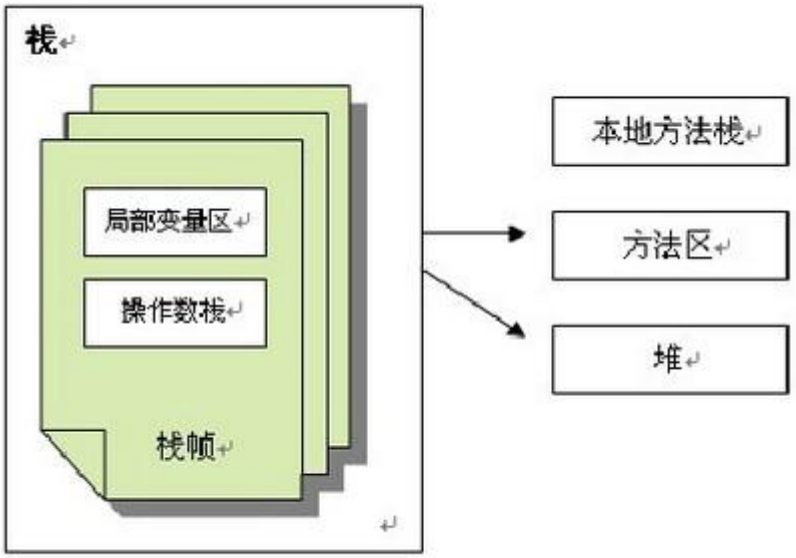
## 类执行机制

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。

# 内存管理和垃圾回收

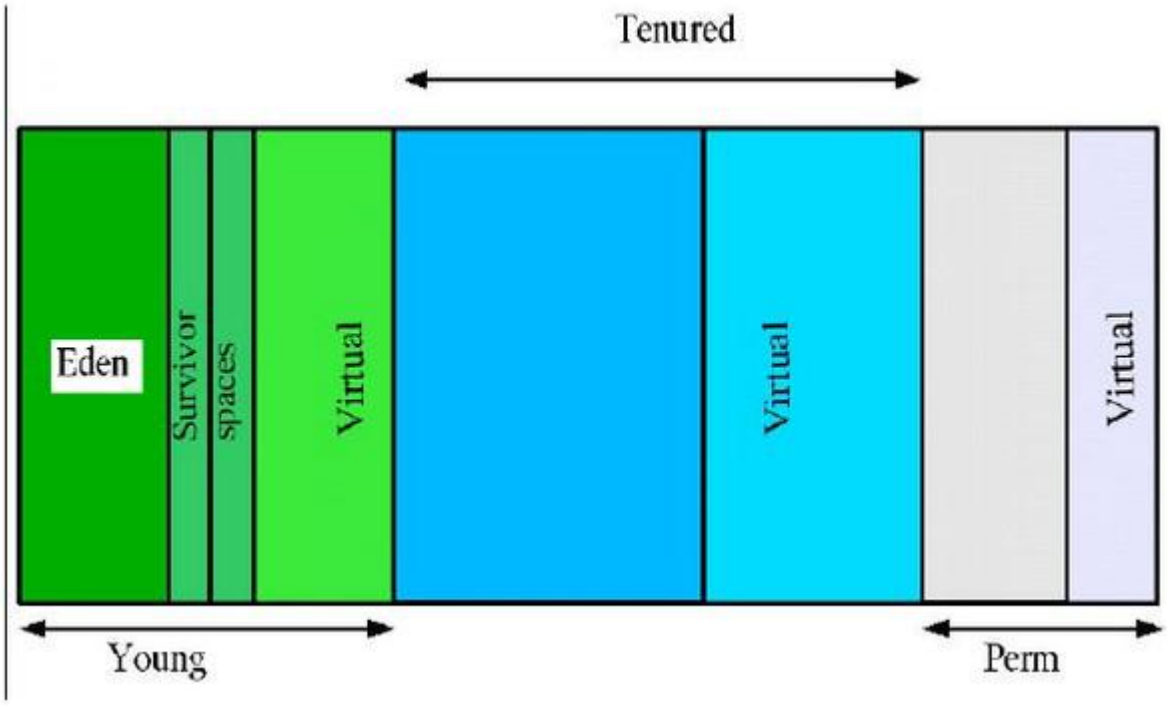
## JVM 内存组成结构

JVM 栈由堆、栈、本地方法栈、方法区等部分组成，结构图如下所示：




# JVM 内存回收

Sun 的 JVMGenerationalCollecting(垃圾回收)原理是这样的：把对象分为年青代(Young)、年老代(Tenured)、持久代(Perm)，对不同生命周期的对象使用不同的算法。(基于对对象生命周期分析)




1.Young（年轻代）

年轻代分三个区。一个 Eden 区，两个 Survivor 区。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制年老区(Tenured。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。

2.Tenured（年老代）

年老代存放从年轻代存活的对象。一般来说年老代存放的都是生命期较长的对象。

3.Perm（持久代）

用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=进行设置。

举个例子：当在程序中生成对象时，正常对象会在年轻代中分配空间，如果是过大的对象也可能会直接在年老代生成(据观测在运行某程序时候每次会生成一个十兆的空间用收发消息，这部分内存就会直接在年老代分配）。年轻代在空间被分配完的时候就会发起内存回收，大部分内存会被回收，一部分幸存的内存会被拷贝至 Survivor 的 from 区，经过多次回收以后如果 from 区内存也分配完毕，就会也发生内存回收然后将剩余的对象拷贝至 to 区。等到 to 区也满的时候，就会再次发生内存回收然后把幸存的对象拷贝至年老区。

通常我们说的 JVM 内存回收总是在指堆内存回收，确实只有堆中的内容是动态申请分配的，所以以上对象的年轻代和年老代都是指的 JVM 的 Heap 空间，而持久代则是之前提到的 MethodArea，不属于 Heap。

# 关于 JVM 内存管理的一些建议

1、手动将生成的无用对象，中间对象置为 null，加快内存回收。

2、对象池技术如果生成的对象是可重用的对象，只是其中的属性不同时，可以考虑采用对象池来较少对象的生成。如果有空闲的对象就从对象池中取出使用，没有再生成新的对象，大大提高了对象的复用率。

3、JVM 调优通过配置 JVM 的参数来提高垃圾回收的速度，如果在没有出现内存泄露且上面两种办法都不能保证 JVM 内存回收时，可以考虑采用 JVM 调优的方式来解决，不过一定要经过实体机的长期测试，因为不同的参数可能引起不同的效果。如-Xnoclassgc 参数等。