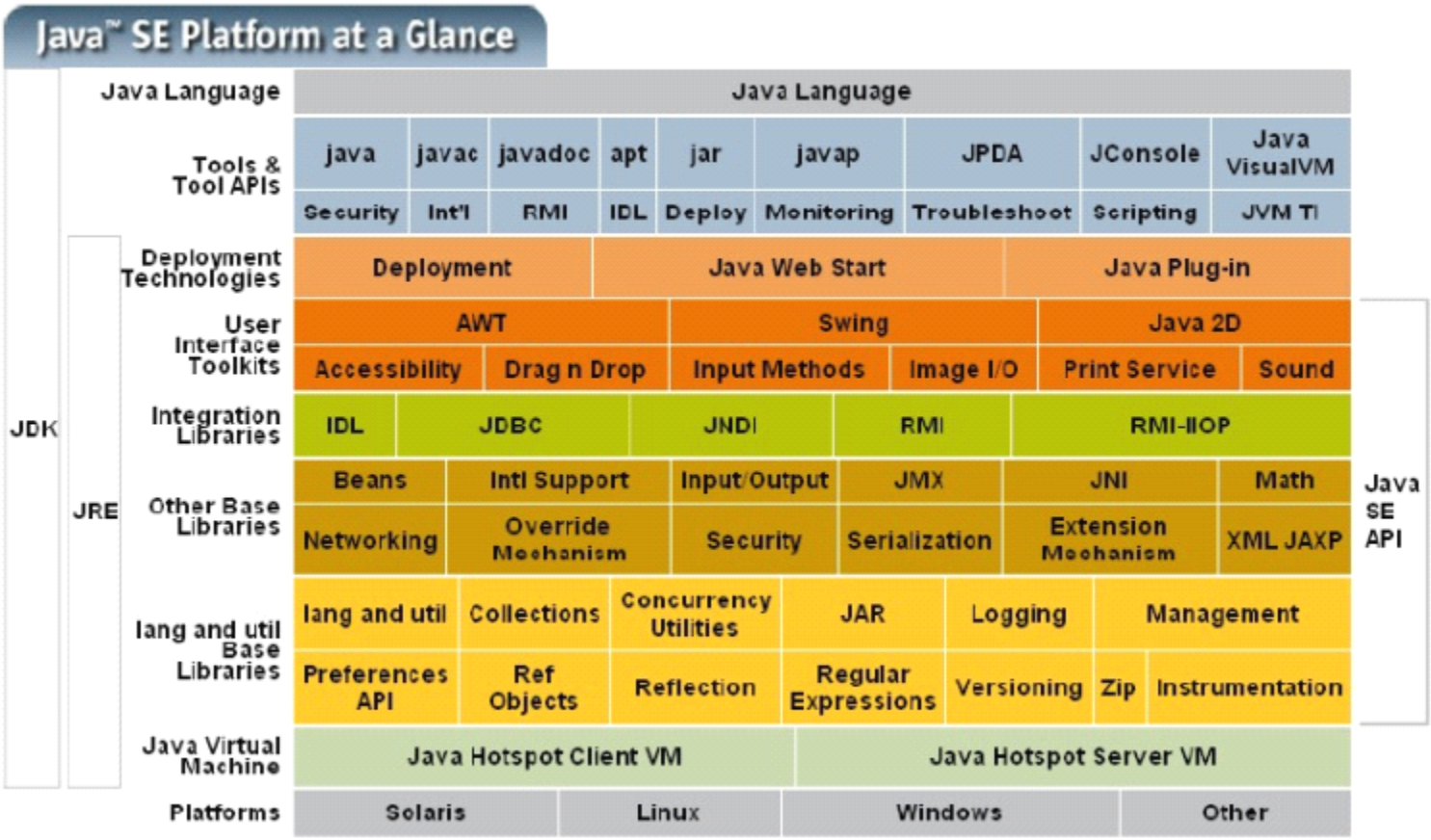
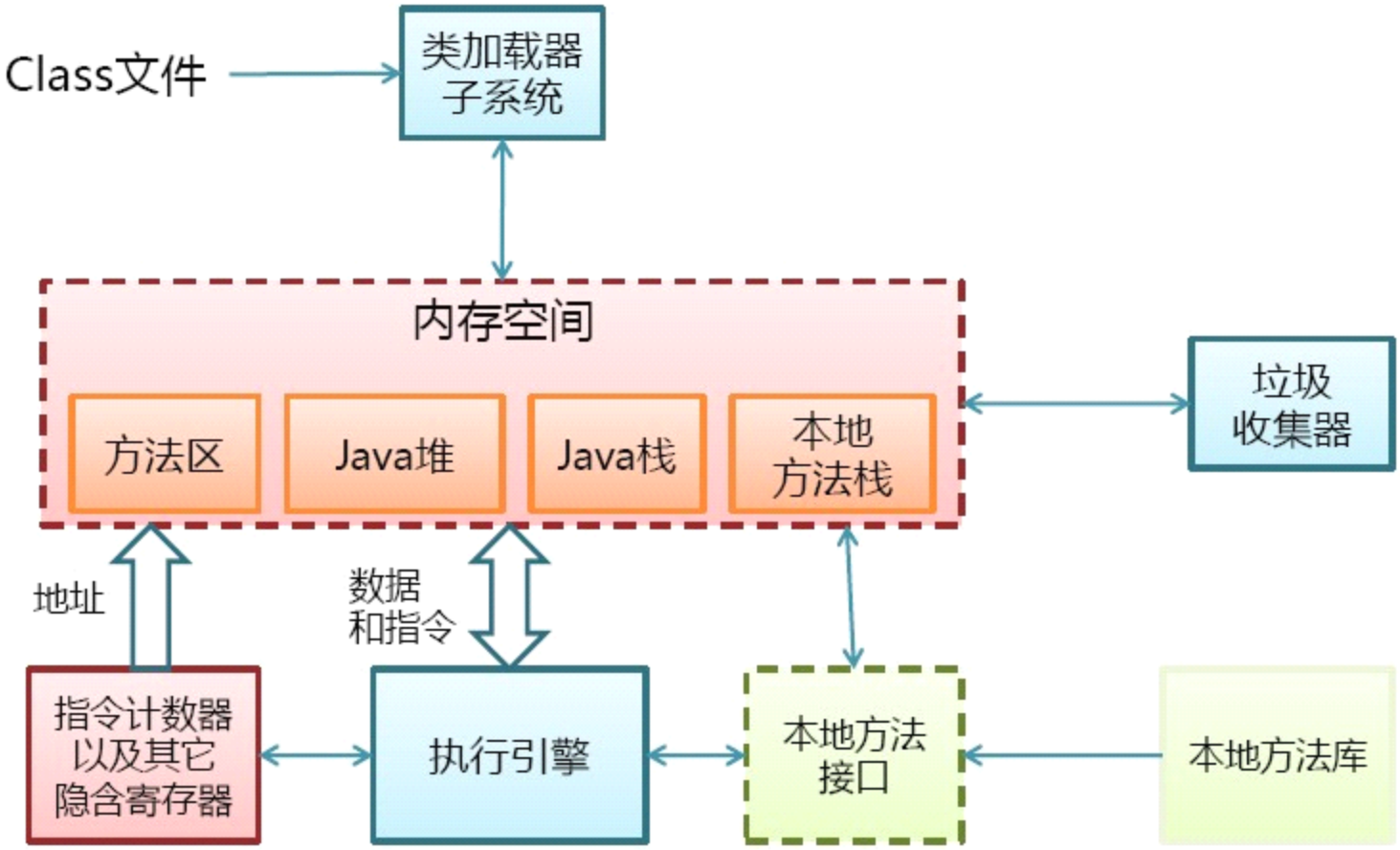
JVM 学习笔记（一）------基本结构 收藏

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：



对于 JVM 的学习，在我看来这么几个部分最重要：

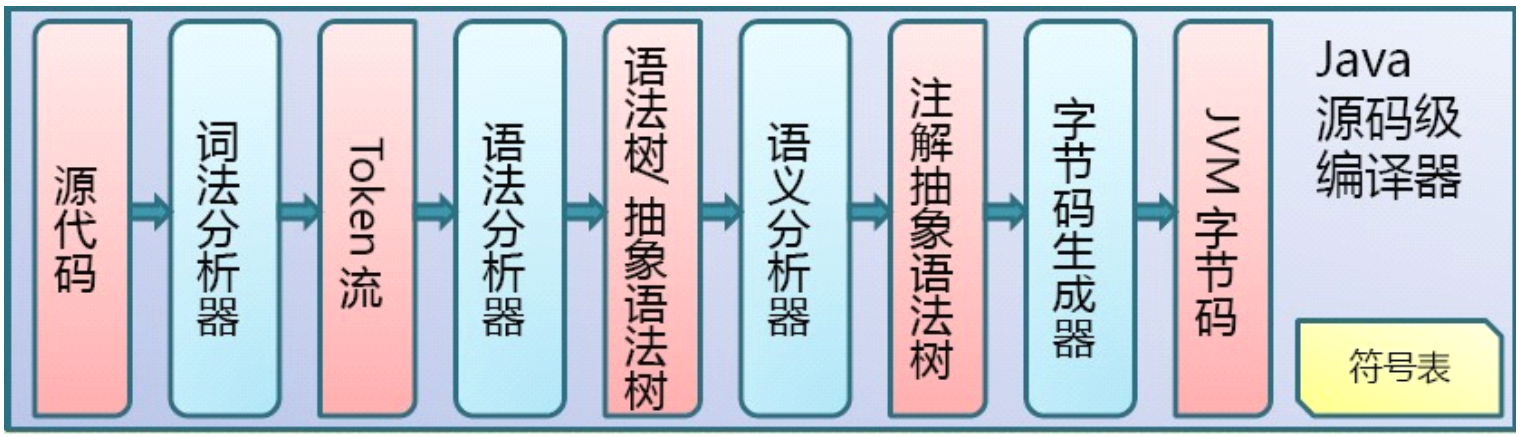
Java 代码编译和执行的整个过程

JVM 内存管理及垃圾回收机制

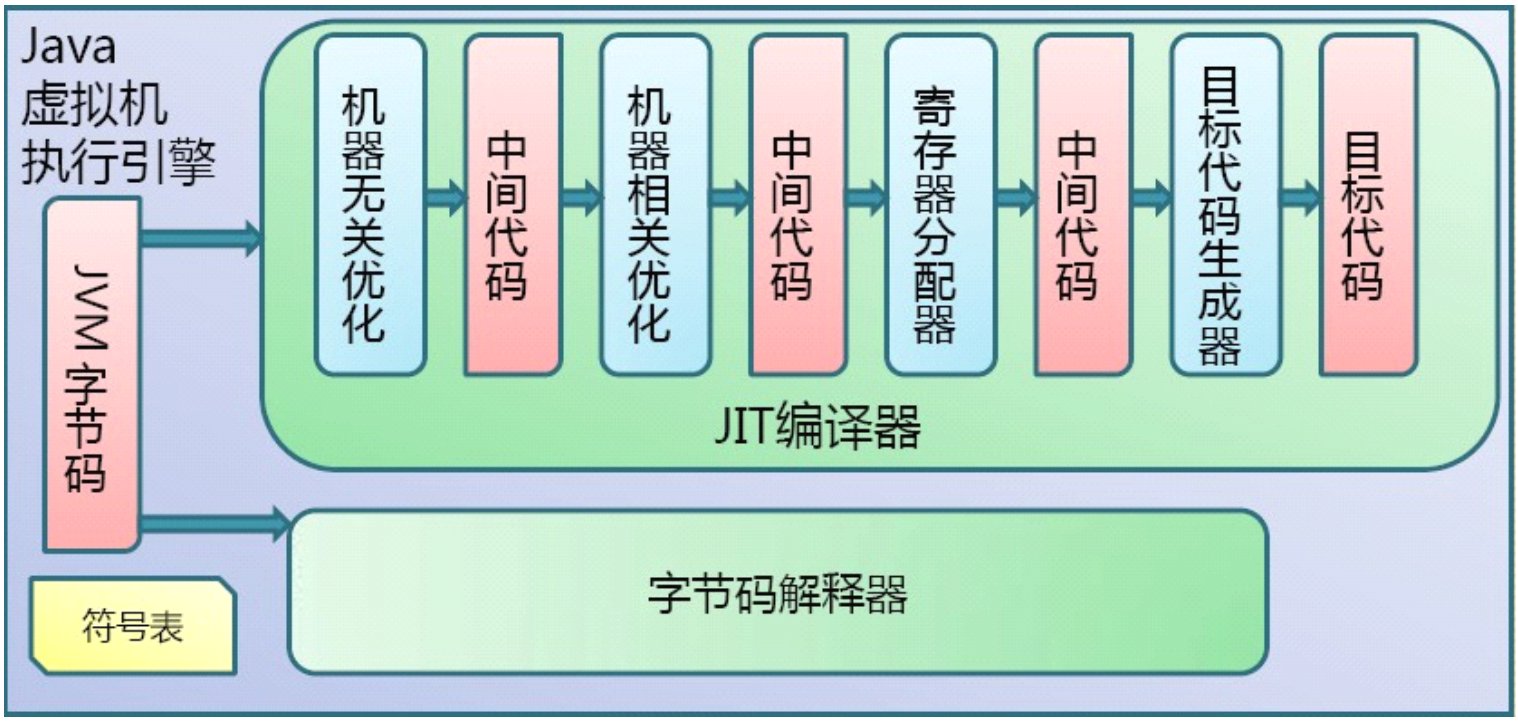
下面将这两个部分进行详细学习

JVM 学习笔记（二）------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：

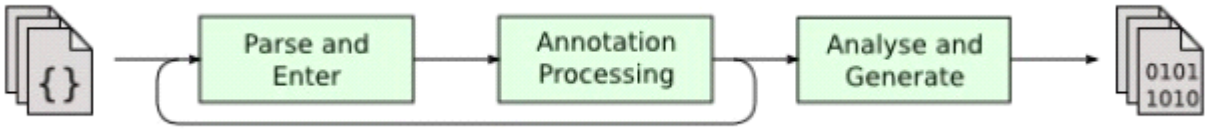
- Java 源码编译机制
- 类加载机制
- 类执行机制

Java 源码编译机制

Java 源码编译由以下三个过程组成：

- 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：

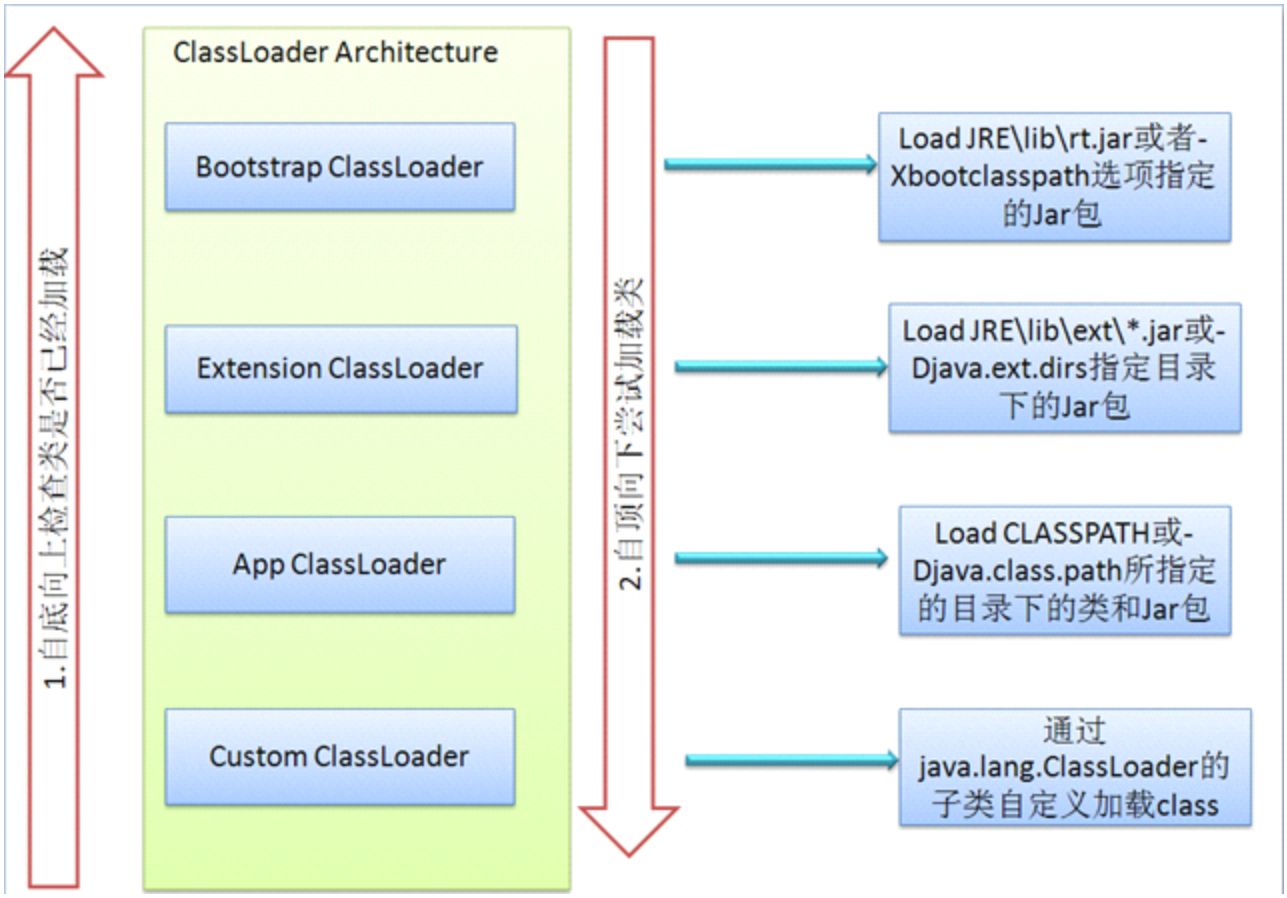


最后生成的 class 文件由以下部分组成：

- 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

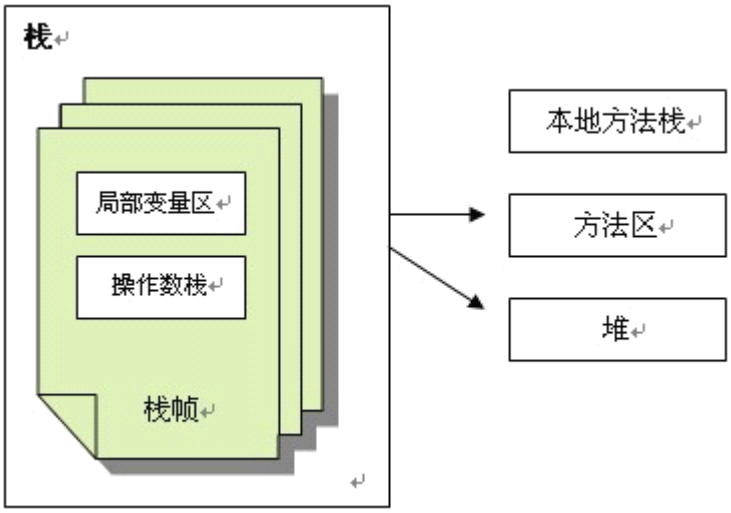
JVM 是基于栈的体系结构来执行 class 字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中

间结果。栈的结构如下图所示：

原 JVM 学习笔记（三）------内存管理和垃圾回收

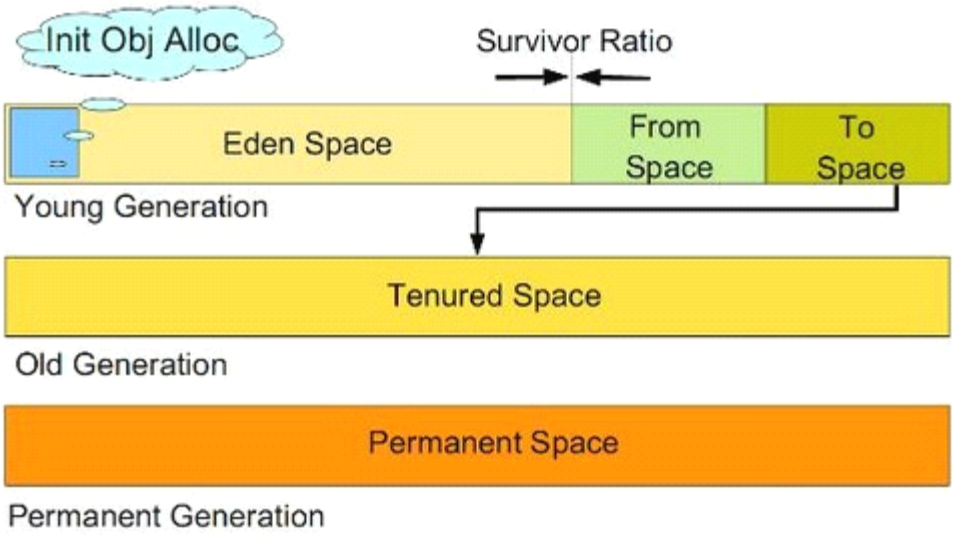
JVM 内存组成结构

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，其大小可以通过-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
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

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

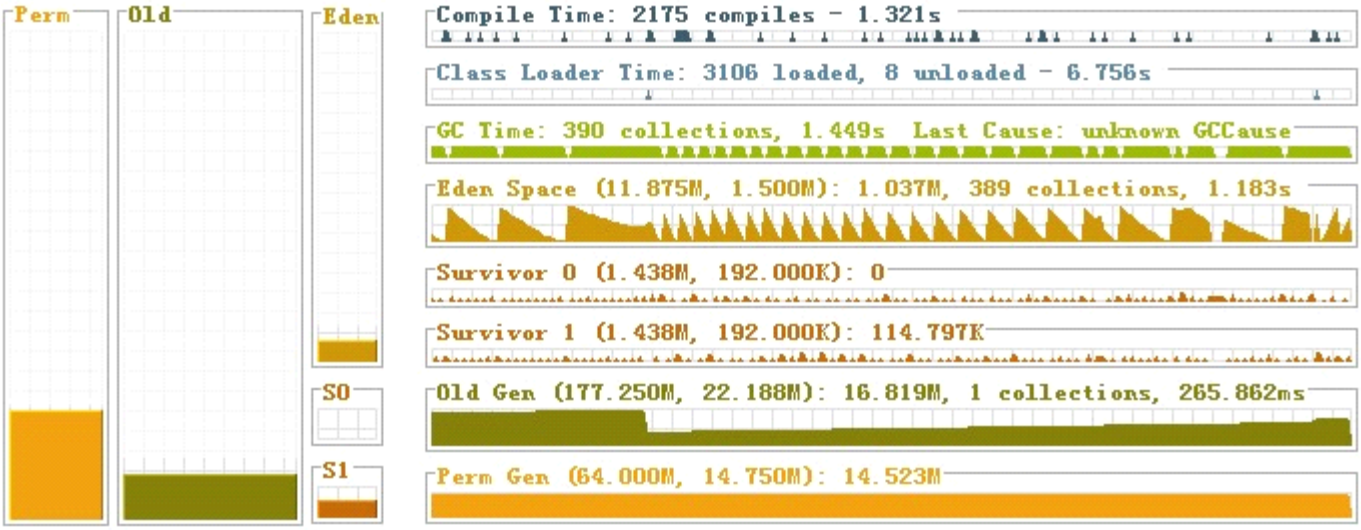
垃圾回收机制

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，

用 java visualVM 来查看，能明显观察到新生代满了后，会把对象转移到旧生代，然后清空继续装载，当旧生代也满了后，就会报 outofmemory 的异常，如下图所示：



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 -XX:+UseParNewGC	串行GC	并发GC
不支持的组合	1、-XX:+UseParNewGC -XX:+UseParallelOldGC 2、-XX:+UseParNewGC -XX:+UseSerialGC	

原 JVM 学习笔记（四）------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

由上一篇博文 JVM 学习笔记（三）------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

1. 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

2. 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
3. 垃圾回收统计信息
   - -XX:+PrintGC
   - -XX:+PrintGCDetails
   - -XX:+PrintGCTimeStamps
   - -Xloggc:filename
4. 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)
5. 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。