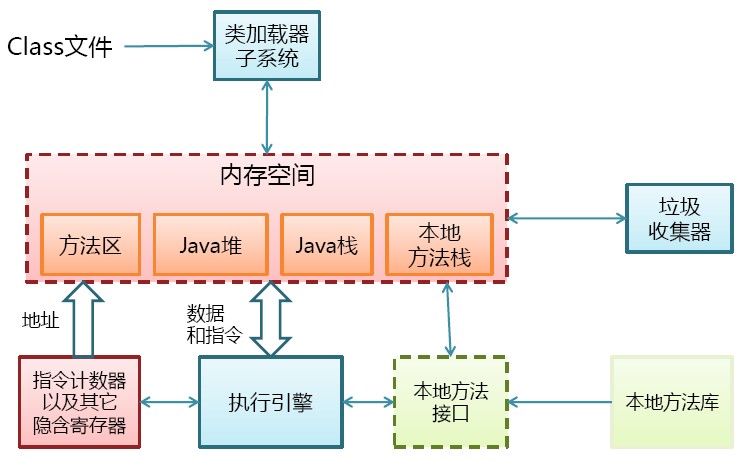
# 什么是JVM

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, 相当于虚构的计算机。

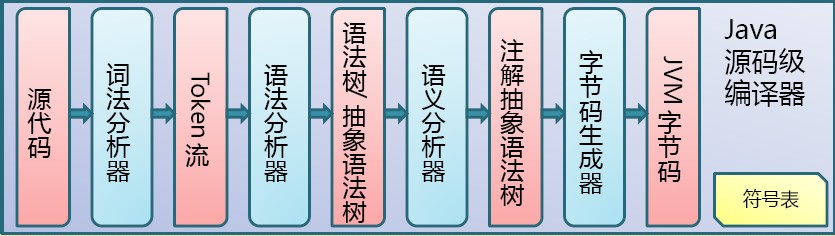
一般高级语言如果在不同的平台运行，可能需要编译成不同的目标代码，引入Java语言虚拟机后，Java语言不需要重新编译。虚拟机在执行字节码时，把字节码解释成具体平台上的指令执行。

下图为jvm大致物理结构：

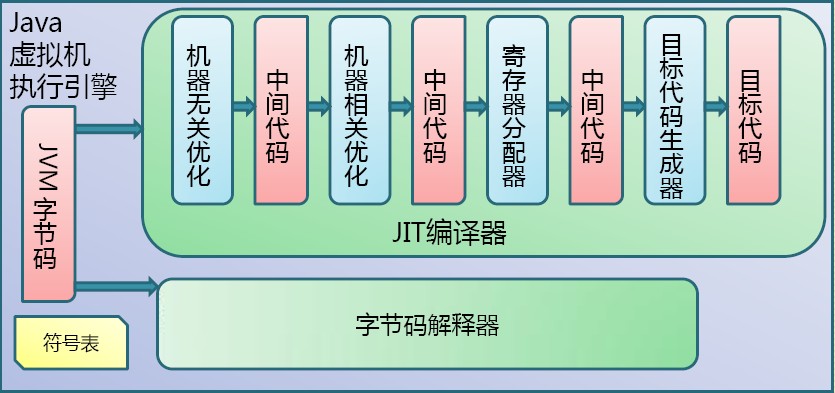


# JAVA代码编译和执行过程

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：



Java代码编译和执行的整个过程包含了一下三个重要机制：

1 Java源码编译机制

2类加载机制

3类执行机制

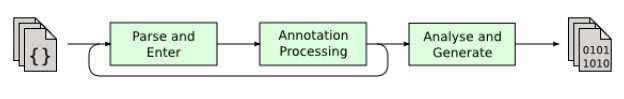
Java源码编译机制（一下三个过程组成）：

1分析和输入到符号表

2注解处理

3语义分析和生成class文件

流程图如下：



最后生成的class文件由以下部分构成：

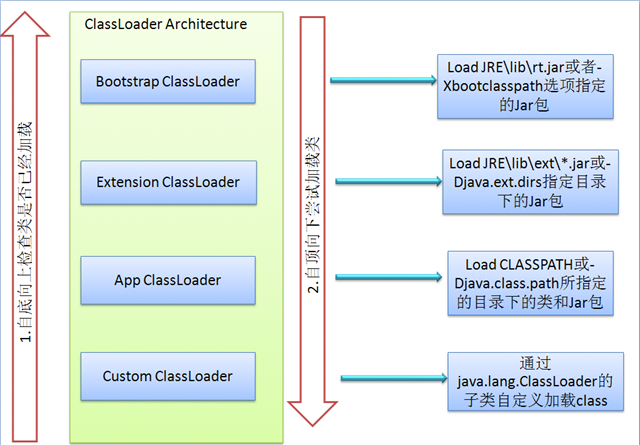
结构信息：class文件格式版本号及各部分的数量与大小的信息。

元数据。对应于JAVA源码中声明与常亮的信息。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亮池。

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（包含很多常用的类，例如String,Long）里所有的class，由C++实现，不是ClassLoader子类。

1. 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.dires指定目录下的jar包

1. App ClassLoader

负责记载classpath中指定的jar包及目录中class

1. 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BootStrapClassLoader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

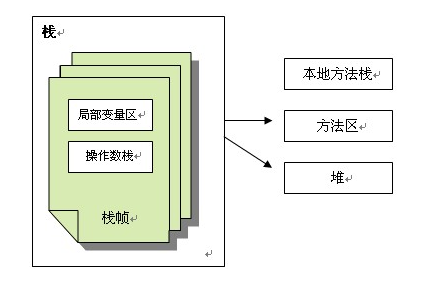
JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生计数器(pc)和栈(stack),程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧。每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量和操作数栈两部分组成，局部变量区用于存放方法中的局部变量数，操作数栈中用于存放方法执行过程中的中间结果。栈结构如下所示：



# JVM内存管理和垃圾回收

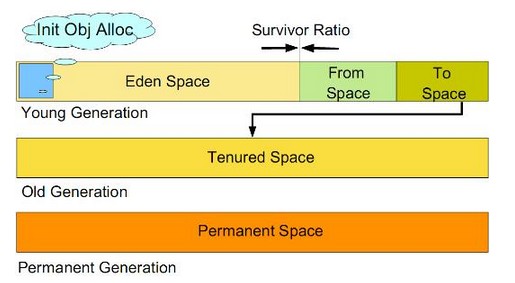
JVM内存组成结构

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:

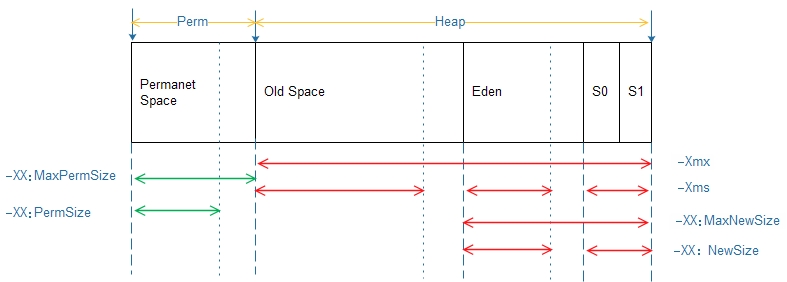
所有通过new创建的对象的内存都在堆中分配，堆的大小可以通过-Xmx和-Xms来控制。堆被划分为新生代和旧生代，新生代又被进一步划分为Eden和Survivor区，最后Survivor由From Space和To Space组成，结构图如下：



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，新生代大小可由-Xmn来控制，也可以用-XX:SurvivoRatio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。

持久代（永久代）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。可以通过-XX:PermSize和MaxPermSize来指定持久代初始化值和最大值。持久代并不等同于方法区，只不过是Hotspot JVM用永久代来实现方法区而已，有些虚拟机没有Permanent Space而用其他机制来实现方法区。



-Xmx:最大对内存，如：-Xmx512m

-Xms：初始时堆内存，如：-Xms256m

-XX:MaxNewSize: 最大新生代内存

-XX:PermSize:初始时新生代内存，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=Eden+1个Survivor即90%

-XX:MaxPermSize: 最大持久代内存。

-XX:Permsize: 初始时持久代内存。

-XX:+PrintGCDetails.打印GC信息。

1. 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。

1. 本地方法栈

用于支持native方法的执行，存储了native方法的调用。

1. 方法区

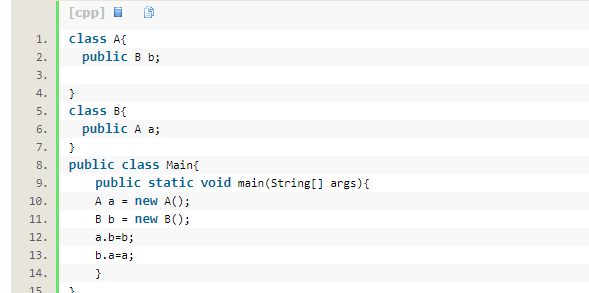
存放了要加载的类信息，静态变量，final类型的常亮，属性和方法信息。JVM永久代来存放方法区，可通过 -XX:PermSize和-XX:MaxPermSize来指定最小值和最大值

垃圾回收按照基本回收策略：

简述：垃圾回收的循环引用问题：

产生原因：每个内存对象维护一个引用计数。

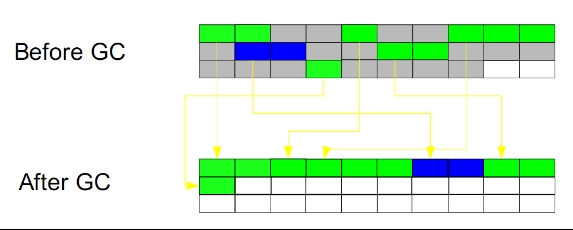
当有新的引用指向某对象时就该将对象的引用计数+1，当指向该对象的某一引用被销毁是-1，当计数归零时，就回收该对象所占用的内存资源。导致循环引用时某些垃圾回收算法无法检测，引发内存泄漏。例如：



在函数结尾，a和b的计数均为2

先撤销a，然后a的计数为1，再等待b.a对a的引用的撤销，也就是在等待b的撤销，对于b来讲也是同理两个对象都在等待对方撤销，资源都不被释放。

1. 引用计数：比较古老的回收算法。原理是对此对象有一个引用，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。此算法无法处理循环引用的问题。
2. 标记清除：



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。

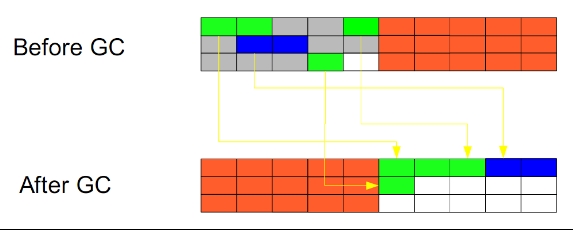
该算法一般应用于老年代，因为老年代的对象生命周期比较长。

优点：可以解决循环引用的问题。

必要时才回收（内存不足时）

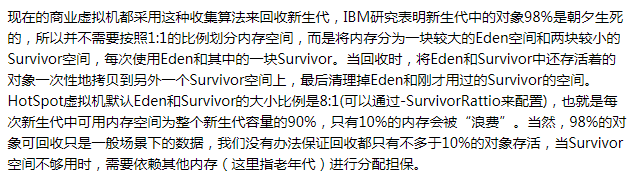
缺点：回收时应用需要挂起。效率不高。会造成内存碎片。

1. 复制：

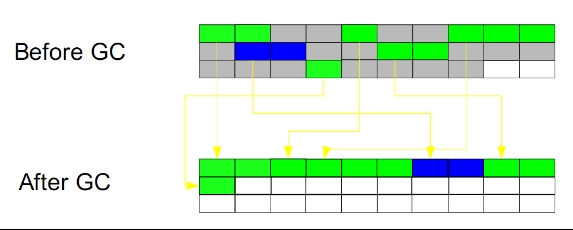


此算法吧内存空间划分为两个相等的区域，每次只是用其中一个

区域。垃圾回收时，遍历当前使用区域，把正在使用中（存活）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，同是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。缺点是会浪费空间。

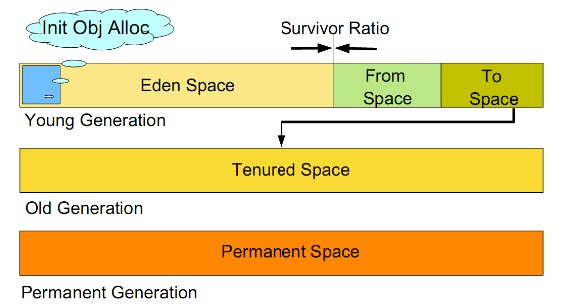


1. 标记-整理



此算法结合了”标记-清除”和”复制”两个算法的有点。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”到堆的一块，按顺序排放。(老年代)

1. generation算法（Generational Collector）



分代的垃圾回收策略，基于一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。

年轻代（Young Generation）

1 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速收集掉那些生命周期短的对象。

2 新生代内存按照8:1:1的比例分为一个eden区和两个survivor（survivor0,survivor1）区。一个Eden区，两个Survivor区（一般而言）。大部分对象在Eden区中生成。回收时先将Eden区存活的对象复制到一个survivor0区，然后清空eden区，当这个survivor0区放满时，则将eden区和survivor0区和survivor1区交换，即保持survivor1区为空，如此往复。

3 当survivor1区不足以存放eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC,也就是新生代，老年代都进行回收。

年老代（Old Generation）

1 在年轻代中经历了N次垃圾回收后仍然存活的对象，会被放到老年代中。因此可以认为老年代中存放的都是一些生命周期比较长的对象。

2 内存比新生代大很多(大概比例是1:2)，当老年代内存满时触发Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率比较高。

持久代（Permanent Generation）

用于存放静态文件，如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。

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（也就是复制算法）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法来控制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rvivor，会逐渐从eden到su最后到旧生代。

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。

1. 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

1. 并行GC

旧生代的GC：

旧生代与新生代不同，对象存活时间比较长，比较稳定，因此采用标记算法进行回收，该算法先扫描出存活的对象，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，减少碎片。在执行机制下JVM提供了串行GC（Serial MSC）,

并行GC（parallel MSC）和并发GC（CMS）。

JAVA内存内存泄漏引起的原因：

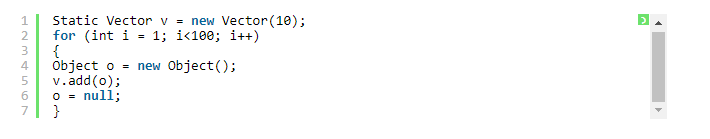
内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。内存泄漏有时不严重且不易察觉，这样开发者就不知道存在内存泄漏，但有时会很严重，会提示Out of memory。

内存泄漏的根本原因是，长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是Java中内存泄漏的发生场景。主要有以下几类：

1. 静态集合类引起内存泄漏：

像HashMap，Vector等的使用最容易出现内存泄漏，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。

例如：



在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个Vector中，如果仅仅释放引用本身（o=null），那么Vector仍然引用该对象，所以这个对象对GC来说是不可回收的。因此，如果对象加入到Vector后，还必须从Vector中删除，最简单的方法就是将Vector对象设置为null。

1. 监听器

通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

1. 各种连接

数据库连接，网络连接和io连接，除非显示的调用close()方法，

将其连接关闭，否则是不会自动被GC回收的。对于Resultset和 Statement对象可以不进行显示回收，因为Connection一旦回收，上 两者会变为null。单如果使用连接池，除了要显示的关闭连接，还必 须显示地关闭Resultset Statement对象（关闭其中一个，另外一个也 会关闭），否则就会造成大量的Statement对象无法释放，从而引起 内存泄漏。