# 一、JVM 概念

JVM 是 Java Virtual Machine (Java 虚拟机)，JVM 是一个虚构出来的计算机 是通过在实际的计算上仿真模拟各种计算机功能来实现

NOTE:

JAVA 语言最重要的一个特点 就是 与平台的无关性（跨平台）

Write Once Run AnyWhere，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 虚拟机后 Java 语言在不同的平台上运行并且不需要重新编译

Java 语言使用Java 虚拟机屏蔽了与具体平台相关的信息

使用Java 语言编译程序生成在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进行修改运行

Java 虚拟机在执行字节码的时候把字节码解释成具体平台上的机器指令执行

这就是 “Write Once Run AnyWhere” 的原因（Java 语言本身不是跨平台，而是虚拟机跨平台）

# 二、Java 程序

（1） 编写程序源码（HelloWorld.java）

public class HelloWorld{

public static void main(String[] args){

System.out.println("Hello World");

}

}

（2） 编译程序源码

javac HelloWorld.java --- .java（源码）--->编译--->.class（字节码）

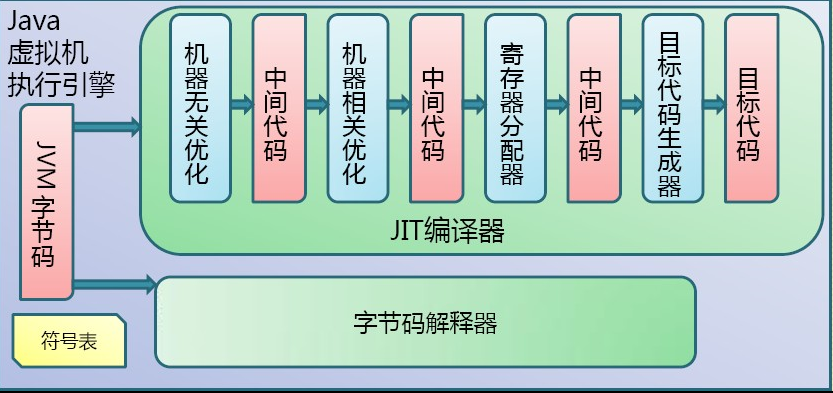
（3） 运行程序

java HelloWorld

（4）查看Java 程序执行流程图

a. .java（源码）--->编译--->.class

（5）查看 Java 虚拟机装载 字节码流程图



字节码执行流程

a. 分析和输入到符号表

b. 注解处理

c. 语义解析和生成class文件

（6）class文件:

结构信息：

包含class 文件格式版本号以及各部分数量与大小的文件

元数据：

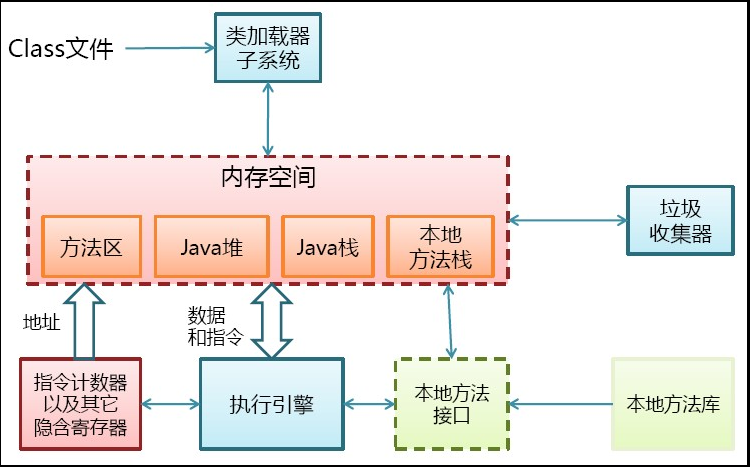
对应于Java 源码中声明与常量的相关信息

包含类 继承的类 实现的接口 等等

方法信息：

对应 java 源码中的语句和表达式对应的信息

# 三、JVM 体系架构



JVM物理结构

## 1、类加载机制

（1）Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar 里的class 由c++实现

（2）Extension ClassLoader

负责加载 Java 平台中扩展功能的一些jar 包含$JAVA\_HOME中jre/lib/\*.jar

（3）App ClassLoader

负责加载Classpath 中指定的jar包及目录中的class(程序运行的时候 所加载的jar包及 class)

（4）Customer ClassLoader

属于应用程序根据自身需要自定义的ClassLoader 如Tomcat 都会根据J2EE 规范自行实现ClassLoader 加载

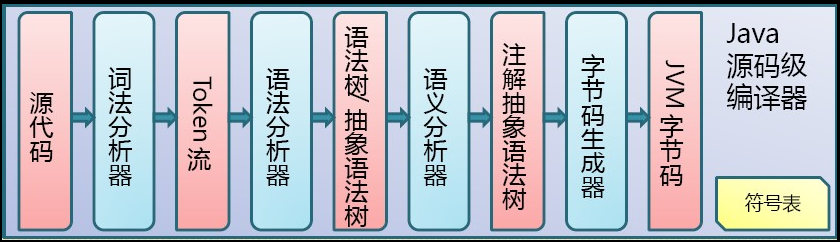
加载过程中会去检查类是否被装及检查顺序

ClassLoader 加载的顺序是自顶向下

从Bootstrap ClassLoader ---> Customer ClassLoader（由上层来逐层尝试加载这个类）

## 2、类执行机制

JVM 是基于栈的体系机构来执行class 字节码 ，线程创建完后，都会产生计数器（PC）和栈（Stack） 程序计算器存放下一条要执行的指定在方法内的偏移量，栈中存放的是一个栈帧 每个栈帧对应这每个方法的每一次调用



JVM执行流程

## 3、内存空间

栈：

是程序的运行区，是在线程创建的时候创建 它的生命周期跟随线程生命周期，线程结束内存也就释放对于栈来说不存在垃圾回收机器 只要线程结束 栈就结束

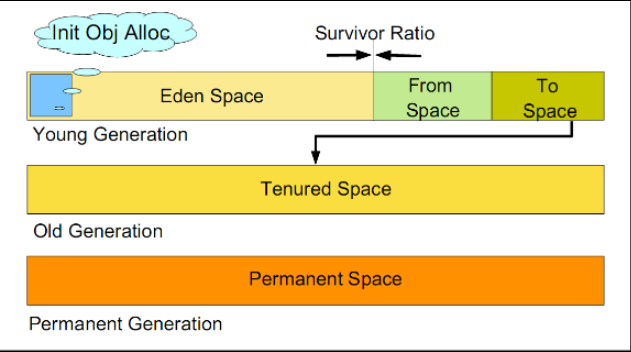
堆：

虚拟机用于存放对象与数组实例的地方，垃圾回收的主要区域就是这里（可能方法区）



堆整体结构

## 4、 堆空间结构：



堆年代划分

整体是按照分代划分的

why 按照分代划分的?

分代回收思想的提出源自于2个假设：

（1） Most allocated objects will die young

大量分配内存的对象都会在比较快的时间内释放（死在年轻代上）

（2） Few reference older to younger objects exist少量的老年代对年轻代的引用

### 4.1、虚拟机年代划分：

#### 年轻代（Young Generation， New Generation）

又称为新生代，程序中新建的对象都将分配到新生代中，新生代又由Eden Space和两块Survivor Space（s0,s1）构成.

年轻代主要存放新创建的对象，内存大小相对会比较小，垃圾回收会比较频繁。年轻代分成1个Eden Space和2个Suvivor

Space（命名为A和B）。当对象在堆创建时，将进入年轻代的Eden Space

#### 年老代（Old Generation）:

年轻代和年老代的划分是对垃圾收集影响比较大的

又称为旧生代，用于存放程序中经过几次垃圾回收还存活的对象，例如缓存的对象等，旧生代所占用的内存大小即为-Xmx指定的大小减去-Xmn指定的大小

年老代主要存放JVM认为生命周期比较长的对象（经过几次的Young Gen的垃圾回收后仍然存在），内存大小相对会比较大，垃圾回收也相对没有那么频繁（譬如可能几个小时一次）。年老代主要采用压缩的方式来避免内存碎片（将存活对象移动到内存片的一边，也就是内存整理）。当然，有些垃圾回收器（譬如CMS垃圾回收器）出于效率的原因，可能会不进行压缩

#### 持久代（PermanentGeneration）：

一个常驻内存区域，用于存放 JDK 自身所携带的 Class， Interface 的元数据，就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存（主要存放类定义、字节码和常量等很少会变更的信息）主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大

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。

### 4.2、HotSpot虚拟机GC算法采用分代收集算法：

（1）、一个人（对象）出来（new 出来）后会在Eden Space（伊甸园）无忧无虑的生活，直到GC到来打破了他们平静的生活。GC会逐一问清楚每个对象的情况，有没有钱（此对象的引用）啊，因为GC想赚钱呀，有钱的才可以敲诈嘛。然后富人就会进入Survivor Space（幸存者区），穷人的就直接kill掉。

（2）、并不是进入Survivor Space（幸存者区）后就保证人身是安全的，但至少可以活段时间。GC会定期（可以自定义）会对这些人进行敲诈，亿万富翁每次都给钱，GC很满意，就让其进入了Genured Gen(养老区)。万元户经不住几次敲诈就没钱了，GC看没有啥价值啦，就直接kill掉了。

（3）、进入到养老区的人基本就可以保证人身安全啦，但是亿万富豪有的也会挥霍成穷光蛋，只要钱没了，GC还是kill掉。

================================================================

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的收集掉那些生命周期短的对象。年轻代分三个区。一个Eden区，两个Survivor区(一般而言)。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区也满了的时候，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。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

Eden--->GC--->Survior(s0)--->GC--->Survior(s1)--->Old

================================================================

### 4.3、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：

（1）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

（2）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

（3）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，

### 4.4、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

（5）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

（6）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

### 4. 5、通常参数设置：

（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

JVM 把年轻代 整体划分为三个部分(对前面关于年轻代的补充与修改)

1、Eden

2、2个Survior(From、To 默认8:1)

一般创建对象的时候首先会被分配到 Eden 区，

这些对象经过第一次GC后，仍然存活 将会被移动到Survior区，对象在每熬过意GC 年龄就增加1岁 当它的年龄增加到一定的程度时 就会被移交到年老代中

因为在年轻代中对象都是在朝生夕死的（80%以上） 所以采用的算法是复制算法

复制算法的基本思想就是将内存分为两块 每次只用其中的一块 当这一块内存用完 就将还活着的对象复制移动另一个块上面

复制算法不会产生内存碎片（采用空间换时间的方式来加速内存垃圾回收）

在GC开始，对象只会存在于Eden区和名为Survior（From）区中的某一块， survior区“To”是空的，紧接着进行GC Eden区中所有存活的对象都会复制到"To",而在"From"区中仍然存活的对象会根据它们的年龄值来决定去向.

年龄达到一定的阈值 才会移动到年老代中，没有达到阈值的对象会被复制到“To”区域，经过这次GC 后 Eden 和From区 已经是被清空的，这个时候会“From”和“To” 会交换它们的角色，也就是说新的“To” 就是上次GC前的From 新的From 就是上次GC前的To,不管怎么样都会Survior区始终有一块是空的,GC会一直重复这样的过程

# 四、垃圾回收机制及优化

## 一、垃圾回收器机制

GC(Garbage Collection)

垃圾回收器机制是由垃圾回收器Garbage Collection实现的

GC 是后台的守护进程，它的特别之处是一个低优先级进程

可以根据内存的使用情况动态调整他的优先级

因此它在内存中低到一定限度时才会自动运行 从而实现对内存的回收

--垃圾回收的运行时间不确定的原因

## 二、执行流程详解

程序运行期间 所有对象实例存储在运行时数据区域的heap 中 当一个对象不再被引用的时候 它就需要被回收，在GC过程中 这些不再被使用的对象从heap中回收掉 这样空间可以循环利用

## 三、垃圾回收器常用算法：

### a. 引用计数（Reference Counting）:

比较古老的回收算法 原理此对象有一个引用，即增加一个计数，删除一个引用就是减少一个计数，垃圾回收的时候 只回收计数为0的对象

问题：无法处理循环引用

A--->B

B--->A

### b. 标记-清除（Mark-Sweep）

算法分为两个阶段:

第一阶段是从引用根几点开始标记所有被引用的对象，

第二阶段遍历整个heap 把未标记的对象清除

缺点：这个算法需要暂停整个应用 同时会产生内存的碎片

### c. 复制（Coppying）

此算法把内存空间划分为两个相等的区域 每次只使用其中一个区域，垃圾回收的时候 遍历当前使用的区域 把正在使用的对象复制到另一个区域中 算法每次只处理正在使用的对象因此复制成本较小同时复制过去以后能进行相应的调整 不会出“碎片”问题

缺点：需要两倍内存空间

### d. 标记-整理（Mark-Compact）:

结合上面“标记-清除”和“复制”两个算法的优点 也是分为两个解读那

第一阶段是从引用根几点开始标记所有被引用的对象，

第二阶段遍历整个heap 把清除未标记对象并且把存活对象压缩到堆的其中某一块 按顺序排放

此算法 避免了“标记-清除”算法的碎片问题 和 “复制”算法空间问题

### e. 增量收集（Incremental Collecting）

实施垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

### f. 分代（Generational Collecting）

基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

## 四、GC 种类：

Sun GC 主要方式有CMS 和 G1

CMS(Concurrent Mark-Sweep)是JDK 1.4 后期引用的新的GC算法

应对场景：响应时间的重要性比较高的应用

并且预期这部分应用能够承受垃圾回收线程和应用线程共享处理器资源

是以牺牲吞吐量为代价来获取最短回收停顿时间的垃圾回收器

针对年老代的回收

JVM 参数：

-XX:+UseConcMarkSweepGC

### 1 Scavenge GC

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，堆Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。

### 2 Full GC

对整个堆进行整理，包括Young、Tenured(old)和Perm。Full GC比Scavenge GC要慢，因此应该尽可能减少Full GC。有如下原因可能导致Full GC：

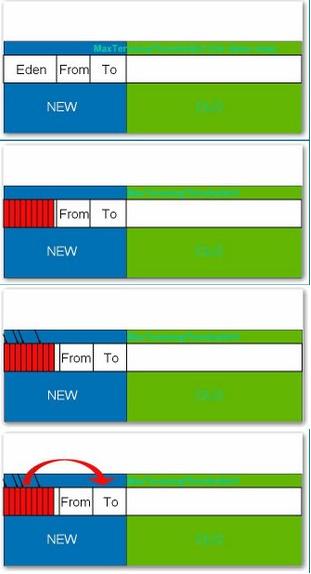
• Tenured被写满

• Perm域被写满

• System.gc()被显示调用

• 上一次GC之后Heap的各域分配策略动态变化

## 五、分代垃圾回收过程演示

[](http://static.oschina.net/uploads/space/2011/0411/123815_4irb_84549.jpg)

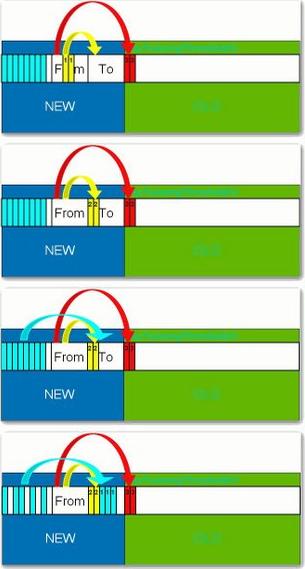
图表 1‑2 回收过程(1)

[](http://static.oschina.net/uploads/space/2011/0411/123858_3Zft_84549.jpg)

图表 1‑3 回收过程(2)

[](http://static.oschina.net/uploads/space/2011/0411/123943_qZQx_84549.jpg)

图表 1‑4 回收过程(3)

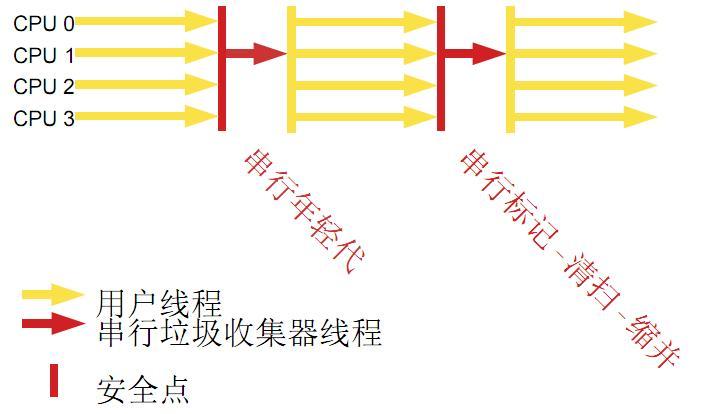
[](http://static.oschina.net/uploads/space/2011/0411/124028_xY4r_84549.jpg)

图表 1‑4 回收过程(4)

## 六、垃圾回收器

目前的收集器主要有三种：串行收集器、并行收集器、并发收集器。

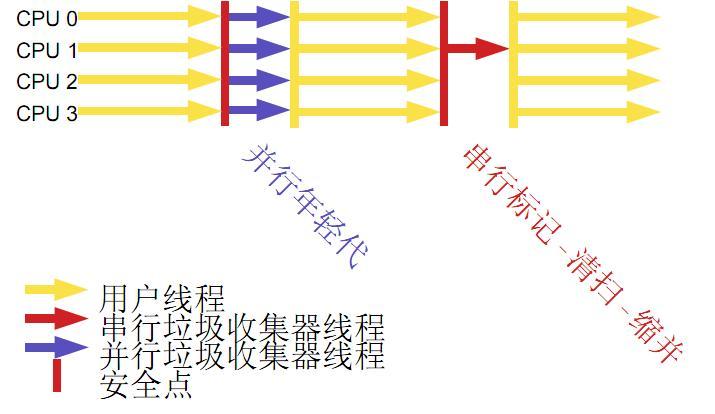
### 6.1串行收集器

[](http://static.oschina.net/uploads/space/2011/0411/124110_1HTr_84549.jpg)

图表 2‑1 串行收集器

使用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，无法使用多处理器的优势，所以此收集器适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。可以使用-XX:+UseSerialGC打开。

### 6.2并行收集器

[](http://static.oschina.net/uploads/space/2011/0411/124154_PEt0_84549.jpg)

图表 2‑2 并行收集器

对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。使用-XX:+UseParallelGC.打开。并行收集器在J2SE5.0第六更新上引入，在Java SE6.0中进行了增强--可以堆年老代进行并行收集。如果年老代不使用并发收集的话，是使用单线程进行垃圾回收，因此会制约扩展能力。使用-XX:+UseParallelOldGC打开。

#### 6.2.1使用-XX:ParallelGCThreads=设置并行垃圾回收的线程数

此值可以设置与机器处理器数量相等。

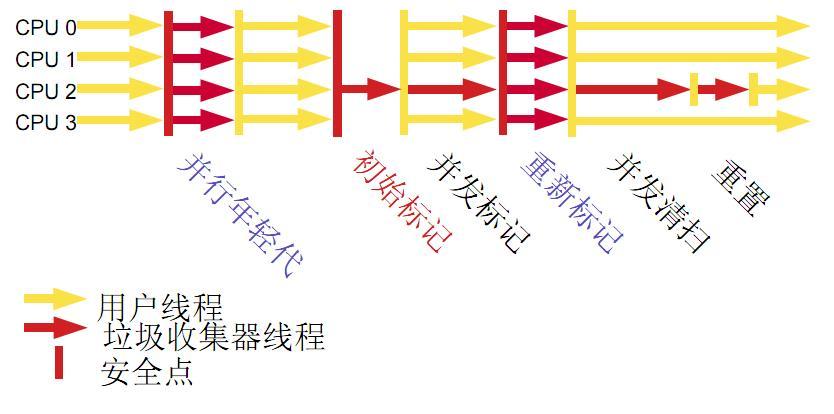
#### 6.2.2收集器配置

§ 最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=指定。为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。

§ 吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=来设定，公式为1/（1+N）。例如，-XX:GCTimeRatio=19时，表示5%的时间用于垃圾回收。默认情况为99，即1%的时间用于垃圾回收。

### 6.3并发收集器(CMS)

可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。使用-XX:+UseConcMarkSweepGC打开。

[](http://static.oschina.net/uploads/space/2011/0411/124228_sfNT_84549.jpg)

图表 2‑3 并发收集器

#### 6.3.1回收机制

并发收集器主要减少年老代的暂停时间，他在应用不停止的情况下使用独立的垃圾回收线程，跟踪可达对象。在每个年老代垃圾回收周期中，在收集初期并发收集器会 对整个应用进行简短的暂停，在收集中还会再暂停一次。第二次暂停会比第一次稍长，在此过程中多个线程同时进行垃圾回收工作。

#### 6.3.2并发收集器使用处理器换来短暂的停顿时间

在一个N个处理器的系统上，并发收集部分使用K/N个可用处理器进行回收，一般情况下1<=K<=N/4。

#### 6.3.3一个处理器的主机上使用并发收集器

在只有一个处理器的主机上使用并发收集器，设置为incremental mode模式也可获得较短的停顿时间。

#### 6.3.4浮动垃圾

由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要20%的预留空间用于这些浮动垃圾。

#### 6.3.5Concurrent Mode Failure

并发收集器在应用运行时进行收集，所以需要保证堆在垃圾回收的这段时间有足够的空间供程序使用，否则，垃圾回收还未完成，堆空间先满了。这种情况下将会发生“并发模式失败”，此时整个应用将会暂停，进行垃圾回收。

#### 6.3.6启动并发收集器

因为并发收集在应用运行时进行收集，所以必须保证收集完成之前有足够的内存空间供程序使用，否则会出现“Concurrent Mode Failure”。通过设置-XX:CMSInitiatingOccupancyFraction=指定还有多少剩余堆时开始执行并发收集

### 6.4垃圾回收器小结

#### 6.4.1串行处理器

 --适用情况：数据量比较小（100M左右）；单处理器下并且对响应时间无要求的应用。  
 --缺点：只能用于小型应用

#### 6.4.2并行处理器

 --适用情况：“对吞吐量有高要求”，多CPU、对应用响应时间无要求的中、大型应用。举例：后台处理、科学计算。

 --缺点：应用响应时间可能较长

#### 6.4.3并发处理器

 --适用情况：“对响应时间有高要求”，多CPU、对应用响应时间有较高要求的中、大型应用。举例：Web服务器/应用服务器、电信交换、集成开发环境。

### 6.5常见配置举例

#### 6.5.1堆大小设置

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统 下，一般限制在1.5G~2G；64为操作系统对内存无限制。我在Windows Server 2003系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

典型设置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置JVM最大可用内存为3550M。

-Xms3550m：设置JVM初始内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g：设置年轻代大小为2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss128k： 设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m:设置持久代大小为16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率。

#### 6.5.2回收器选择

JVM给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前[系统配置](http://java.sun.com/j2se/1.5.0/docs/guide/vm/server-class.html)进行判断。

##### 6.5.2.1吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学计数和后台处理等。

典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

##### 6.5.2.2响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

##### 6.5.2.3辅助信息

JVM提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

-XX:+PrintGC

输出形式：

|  |
| --- |
| [GC 118250K->113543K(130112K), 0.0094143 secs]  [Full GC 121376K->10414K(130112K), 0.0650971 secs] |

-XX:+Printetails

输出形式：

|  |
| --- |
| [GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]  [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs] |

-XX:+PrintGCTimeStamps -XX:+PrintGC：PrintGCTimeStamps可与上面两个混合使用

输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

-XX:+PrintGCApplicationConcurrentTime:打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用

输出形式：Application time: 0.5291524 seconds

-XX:+PrintGCApplicationStoppedTime：打印垃圾回收期间程序暂停的时间。可与上面混合使用

输出形式：Total time for which application threads were stopped: 0.0468229 seconds

-XX:PrintHeapAtGC:打印GC前后的详细堆栈信息

输出形式：

|  |
| --- |
| 34.702: [GC {Heap before gc invocations=7:   def new generation   total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000) eden space 49152K,  99% used [0x1ebd0000, 0x21bce430, 0x21bd0000)  from space 6144K,  55% used [0x221d0000, 0x22527e10, 0x227d0000)    to   space 6144K,   0% used [0x21bd0000, 0x21bd0000, 0x221d0000)   tenured generation   total 69632K, used 2696K [0x227d0000, 0x26bd0000, 0x26bd0000)  the space 69632K,   3% used [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)   compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs] 55264K->6615K(124928K)Heap after gc invocations=8:  def new generation   total 55296K, used 3433K [0x1ebd0000, 0x227d0000, 0x227d0000)  eden space 49152K,   0% used [0x1ebd0000, 0x1ebd0000, 0x21bd0000)  from space 6144K,  55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)  to   space 6144K,   0% used [0x221d0000, 0x221d0000, 0x227d0000)  tenured generation   total 69632K, used 3182K [0x227d0000, 0x26bd0000, 0x26bd0000)  the space 69632K,   4% used [0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)   compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000) }, 0.0757599 secs] |

-Xloggc:filename:与上面几个配合使用，把相关日志信息记录到文件以便分析。

#### 6.5.3常见配置汇总

##### 6.5.3.1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

##### 6.5.3.2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

##### 6.5.3.3垃圾回收统计信息

-XX:+PrintGC

-XX:+Printetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

##### 6.5.3.4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)

##### 6.5.3.5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。

#### 6.5.4调优总结

##### 6.5.4.1年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

##### 6.5.4.2年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

并发垃圾收集信息

持久代并发收集次数

传统GC信息

花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

##### 6.5.4.3较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间 较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出 现“碎片”，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

-XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

#### 6.5.5参考文献

* [Java 理论与实践: 垃圾收集简史](http://www.ibm.com/developerworks/cn/java/j-jtp10283/)
* [Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning](http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html#resources)
* [Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1](http://developers.sun.com/mobility/midp/articles/garbagecollection2/#16.2.6)
* [Hotspot memory management whitepaper](https://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf)
* [Java Tuning White Paper](http://java.sun.com/performance/reference/whitepapers/tuning.html)
* [Diagnosing a Garbage Collection problem](http://java.sun.com/docs/hotspot/gc1.4.2/example.html)
* [Java HotSpot VM Options](http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp)
* [A Collection of JVM Options](http://blogs.sun.com/watt/resource/jvm-options-list.html)
* [Frequently Asked Questions about Garbage Collection in the HotspotTM JavaTM Virtual Machine](http://java.sun.com/docs/hotspot/gc1.4.2/faq.html)