JVM学习

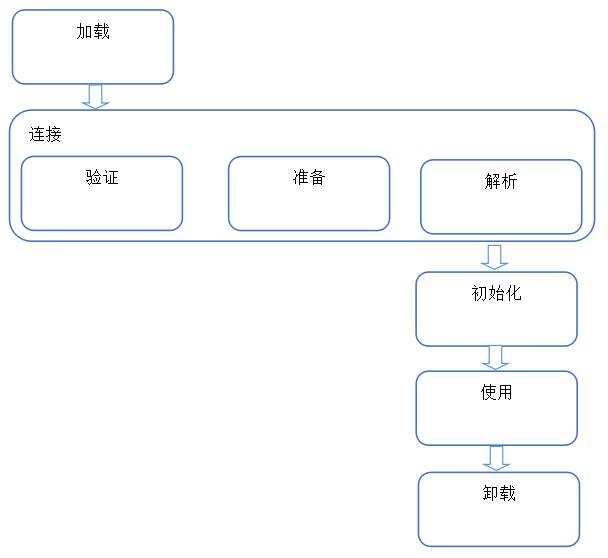
# JVM运行基本过程

Java源码==》javac编译==》java运行

## 类加载机制

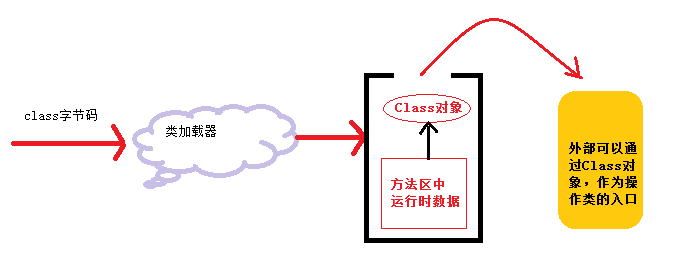
JVM将class文件加载到内存中，并对数据及逆行校验、解析、初始化，最终形成JVM可以直接使用的java类型的过程。

加载（loading）==》（验证（verification）==》准备（pregaration）==》解析（resolution））（链接）==》初始化（initializtion）==》使用（use）==》unloading（卸载）



**加载**

将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，在堆中生成一个代表这个类的java.lang.Class对象，作为方法区类数据的访问入口，这个过程需要类加载器参与。



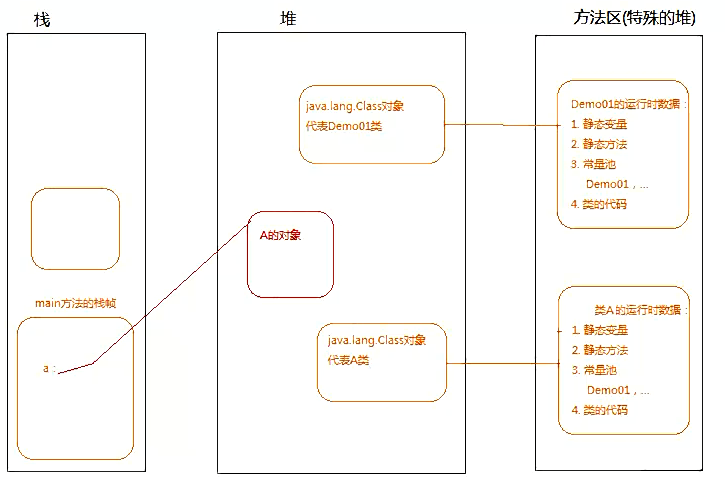
**链接**

将java类的二进制代码合并到JVM的运行状态之中的过程,包括三个阶段：

1. 验证：确保加载的类信息符合JVM规范，没有安全方面的问题
2. 准备：正式为类变量（static变量）分配内存并设置类变量初始值的阶段，这些内存都将在方法去中进行分配
3. 解析：虚拟机常量池的符号引用替换为字节引用过程

**初始化**

1. 初始化阶段是执行类构造器<clinit>（）方法的过程。类构造器<clinit>（）方法是由编译器自动收藏类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生
2. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化
3. 虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确加锁和同步
4. 当范围一个Java类的静态域时，只有真正声名这个域的类才会被初始化



**内存中存在栈、堆（放创建好的对象）、方法区（实际也是一种特殊堆）**

* 1. JVM加载Demo01时候，首先在方法区中形成Demo01类对应静态数据（类变量、类方法、代码…），同时在堆里面也会形成java.lang.Class对象（反射对象），代表Demo01类，通过对象可以访问到类二进制结构。然后加载变量A类信息，同时也会在堆里面形成a对象，代表A类。
  2. main方法执行时会在栈里面形成main方法栈帧，一个方法对应一个栈帧。如果main方法调用了别的方法，会在栈里面挨个往里压，main方法里面有个局部变量A类型的a，一开始a值为null，通过new调用类A的构造器，栈里面生成A（）方法同时堆里面生成A对象，然后把A对象地址付给栈中的a,此时a拥有A对象地址。
  3. 当调用A.width时，调用方法区数据。

**类的主动引用（一定会发生类的初始化）**

1. new一个类的对象
2. 调用类的静态成员（除了final常量）和静态方法
3. 使用java.lang.reflect包的方法对类进行反射调用
4. 当虚拟机启动，java Demo01,则一定会初始化Demo01类，说白了就是先启动main方法所在的类
5. 当初始化一个类，如果其父类没有被初始化，则先初始化它父类

**类的被动引用（不会发生类的初始化）**

1. 当访问一个静态域时，只有真正声名这个域的类才会被初始化
2. 通过子类引用父类的静态变量，不会导致子类初始化
3. 通过数组定义类的引用，不会触发此类初始化
4. 引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

## 类加载器

作用：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，在堆中生成一个代表这个类的java.lang.Class。

### 类加载器层次结构：

1. 引导类加载器（Bootstrap类加载器 – JRE/lib/rt.jar）（用c实现的）

Bootstrap类加载器负责加载rt.jar中的JDK类文件，它是所有类加载器的父加载器。Bootstrap类加载器没有任何父类加载器，如果你调用String.class.getClassLoader()，会返回null，任何基于此的代码会抛出NUllPointerException异常。Bootstrap加载器被称为初始类加载器。

1. Extension类加载器（扩展类加载器） – JRE/lib/ext或者java.ext.dirs指向的目录（用java实现）java9中貌似移除了ExtClassloader类。

而Extension将加载类的请求先委托给它的父加载器，也就是Bootstrap，如果没有成功加载的话，再从jre/lib/ext目录下或者java.ext.dirs系统属性定义的目录下加载类。Extension加载器由sun.misc.Launcher$ExtClassLoader实现。

1. Application类加载器（应用程序类加载器） – CLASSPATH环境变量, 由-classpath或-cp选项定义,或者是JAR中的Manifest的classpath属性定义.（用java实现）

第三种默认的加载器就是System类加载器（又叫作Application类加载器）了。它负责从classpath环境变量中加载某些应用相关的类，classpath环境变量通常由-classpath或-cp命令行选项来定义，或者是JAR中的Manifest的classpath属性。Application类加载器是Extension类加载器的子加载器。通过sun.misc.Launcher$AppClassLoader实现。

1. 自定义类加载器（用java实现）

开发人员可以通过继承java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求。

除引导类加载器外，其他类加载器都必须继承自java.lang.ClassLoader类。

java.lang.ClassLoader（抽象类） API:

public final [ClassLoader](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassLoader.html" \o "class in java.lang) getParent()：得到父类加载器

public Class<?> loadClass(String name) throws ClassNotFoundException:加载名为name类

protected [Class](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Class.html)<?> findClass([String](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/String.html) name) throws [ClassNotFoundException](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassNotFoundException.html)：查找名为name的类。

protected final Class<?> findLoadedClass(String name)：查找已经加载的名为name的类

（@Deprecated）protected final [Class](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Class.html)<?> defineClass(byte[] b, int off, int len)throws [ClassFormatError](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassFormatError.html)：把字节数组中的内容转化为java类

### 类加载器的双亲委托机制（代理模式）

交给其他的加载器来加载指定的类（优先由父类加载，依次追溯，直到最高的类加载器，如果最高父类加载器能完成加载，则直接加载，返回成功，只有父类加载器无法加载时才自己加载。这种方式即双亲委托机制，保证java核心库的类型安全，保证不会出现用户自定义的java.lang.Object类、java.lang.String类等）

并不是所有的类加载器都使用双亲委托机制，如tomcat 的类加载器。

### 自定义类加载器

1. extends java.lang.ClassLoader
2. 检查请求的类型是否已经被这个类加载器加载了，如果已经加载，直接返回
3. 委派类加载请求给父类加载器，父类加载器如果能完成，则返回父类加载器加载的Class实例。
4. 调用本类的类加载器findClass（）方法，视图获取字节码
5. 如果获取不到，调用definClass（…）方法返回异常给loadClass（…）, loadClass（…）抛出异常则终止加载过程。

注意：被两个类加载器同时加载的类，java认为是两个不同的类。

拓展：代码加密：使用类加载器进行类加密解密

## 线程上下文类加载器

不使用双亲委托机制实现类加载，使用线程类加载器实现。

每个线程都有一个关联的上下文类加载器，如果使用new Thread（）方式生成的新的线程，新线程将继承其父线程的上下文类加载器，如果程序对线程上下文类加载器没有任何改动，程序中所有的线程都将使用系统类加载器作为上下文类加载器。

可以使用如下方式获取或设置线程上下文类加载器（抛弃双亲委托机制）：

Thread.currentThread().getContextClassLoader()

Thread.currentThread().setContextClassLoader(classLoader)

## Tomcat类加载器机制

为了保证安全，tomcat不能使用系统默认的类加载器。

如果tomcat在运行web项目时使用系统的类加载器，用户可以直接操作系统中的各个目录，这是相当危险的。

对于运行在java EE容器中的web应用来说，类加载器的实现方式与一般的java应用有所不同。

每个web应用都有一个对应的类加载器实例。该类加载器也是用代理模式（非双亲委托机制），不同的是它首先尝试去加载某个类，如果找不到才代理给其父类加载器。为了保证安全，这种方式下核心库不在查询范围之内。

Tomcat可以限制用户只能把类卸载指定的地方，否则不加载类。

## OSGi(Open Service Gateway Initiative)

面向java的动态模块系统，Java动态化模块化系统的一系列规范。它为开发人员提供了面向服务和基于组件的运行环境。并提供标准的方式来管理软件的声明周期。Eclipse的插件机制就是基于OSGi技术来构建的。

原理：

每个模块（bundle）都包含类，模块可以声明它所依赖的需要导入（import）的其他模块的java包和类（通过Import-Package），也可以声明导出（export）自己的包和类，供其他模块使用（通过Export-Package）。这些功能通过OSGi特有的类加载机制来实现的。OSGi的每个模块都有一个对应的类加载器。它负责加载模块自己好汉的java包和类。需要加载java核心库的类时会代理给其父类（通过启动类加载器Bootstrap）当它需要加载锁导入的java 类时，他会代理给导出此java类的模块来完成加载。模块可以显示声明某些java包和类，必须由父类加载器来加载。

# jvm结构



方法区 - 所有的类级数据将被存储在这里，包括静态变量。每个JVM只有一个方法区域，它是一个共享资源。

堆区 – 虚拟机启动时建立。所有对象及其相应的实例变量和数组将存储在此处。每个JVM只有一个堆区。由于 方法 和堆区域 共享多个线程的内存，所存储的数据不是线程安全的。

直接内存：Java堆外的直接向系统申请的内存空间。Java NIO库允许直接使用直接内存。

垃圾回收系统- 垃圾回收器可以对方法区、java堆和直接内存进行回收。Java堆是垃圾回收的重点。Java中所有的对象空间的释放时隐式的。垃圾回收式JVM优化的重点。

Java栈 – 每个java虚拟机线程都有一个java栈，一个线程的java栈在线程创建时被创建，Java栈中保存着帧信息，java栈中保存着局部变量、方法参数，同时和java方法的调用，返回密切相关。

本地方法栈 – 本地方法栈和java栈类似，不同的是java栈用于方法的调用，本地栈用于本地方法（native method）调用，作为java虚拟机的重要扩展，Java虚拟机允许java直接调用本地方法（通常使用c编写，这也是java能屏蔽操作系统差异的重要原因）

PC寄存器（program counter） - 每个线程都有独立的PC寄存器，Java虚拟机为每个Java线程创建PC寄存器。在任意时刻，一个java线程总在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC寄存器就会指向当前正在被执行的指令。如果当前的方法是本地方法，那么PC寄存器的值就是undefined。一旦执行指令就保存当前执行指令的地址，PC寄存器将用下一条指令更新。

执行引擎：执行引擎是java虚拟机的最核心的组件之一，它负责执行虚拟机的字节码，现代虚拟机为了提高执行效率，会使用即时编译（just in time）技术将方法编译成机器码后再执行。

Java hotspot client

Java hotspot server

# jvm堆结构图及分代

## 内存分代策略

Java虚拟机根据对象存活周期不同，把堆内存划分为几块，一般为新生代、年老代和永久代（永久代是对hotspot虚拟机而言）。

为什么分代

堆内存是虚拟机内存管理最大的一块，也是垃圾回收最频繁的一块区域，程序所有的对象实例都放在堆内存中。给堆内存分代是为了提高对内存分配和垃圾回收的效率。如果堆内存没有区域划分，所有的新创建的对象和声明周期很长的对象放在一起。随着程序的执行，堆内存需要频繁进行垃圾回收，每次回收都要便利所有对象，代价巨大。

有了内存分代，新创建的对象会在新生代中分配内存，经过多次回收仍然存活下来的对象存在在老年代中，静态属性、类信息等存放在永久代中，新生代对象存活时间短，只需要在新生代区域中频繁进行GC，年老代中对象生命周期长，内存回收的频率相对较低，不需要频繁的进行回收，永久代中回收效果太差，一般不进行垃圾回收（永久代是hotspot特有的分代，hotspot使用永久代的方法来实现方法区，hotspot JDK1.7开始去永久代）。

可以根据不同年代的特点采用合适的垃圾回收算法，分代收集大大的提升了收集效率，这些都是内存分代带来的好处。

堆的结构图



## 新生代

Eden：伊甸园，对象出生区域（大对象会直接放在老年代）,垃圾回收最频繁区域。当Eden区域没有足够空间时，虚拟机会发起一次Minor GC。GC开始时对象只会存在于Eden和from区，to区时空的（作为保留区）。GC进行时，Eden区所有存活对象都会被复制到to区域，而Form区中，仍然存活的对象会根据他们的年龄值决定去向。年龄值达到（默认15，新生代中的对象每熬过一轮垃圾回收，年龄值就加1，GC分代年龄存储在对象header中。年龄值达到阈值时会移入老年代，没有达到阈值的会被放在To区。接着From和To区会交换角色，即To就是上次清空的From区，保证To区域为空。

From（s0）和To(s1)：合称survivor

Eden :From:To 空间占比8:1:1,使用这种划分方式是为了让hotspot采用复制算法回收新生代。

## 年老代

在新生代中经历多次GC后仍然存活下来的对象会进入年老代，年老代对象生命周期较长，存活率高，在年老代中进行GC的频道相对而言较低，回收速度也较慢。

## 永久代

永久代的存储类信息、常量、静态变量、即使编译器编译后的代码等数据。对这一区域java虚拟机规范指出可以不进行垃圾手机，一般而言不会进行垃圾回收。

# 垃圾回收算法

## 引用计数算法（reference counting）

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。该算法最致命的时无法处理循环引用的问题。

## 复制（copying）

复制算法把内存空间分为两个相等区域，每次只用一个区域。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一个区域中（新生代From到To，该算法每次只处理正在使用中的对象，因此复制成本小，同时复制过去后还能进行内存整理，不会出现碎片化的问题。该算法缺点是需要两倍的存储空间。

## 标记清除法（mark-sweep）

此算法分两个阶段执行，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。该算法需要暂停整个应用，同时会产生内存碎片（优化的方向是缩短暂停时间）。

## 标记整理法（mark-compact）

此算法结合标记清除与复制算法，分两个阶段，第一阶段从根节点开始标记所有的被引用对象，第二阶段遍历整个堆，清除未标记并把存活对象压缩到堆的其中一块，按顺序排放，此算法避免了编辑清除随拍你问题，同时也避免了复制算法的空间问题。

# 垃圾收集器

分类：

ScavengeGC（次收集）和FullGC（全收集）

新生代GC（ScavengeGC）是发生在新生代的GC，因为新生代的java对象大多数都是朝生夕死，所以ScavengeGC十分频繁，速度很快，当Eden空间不足时会触发ScavengeGC。

一般情况下新对象生成在Eden空间申请空间失败时会出发ScavengeGC，堆Eden区域进行GC并清除非存活对象，把村会对象移到survivor区。然后整理survivor两个区。这种方式的GC时堆年轻代的Eden区进行。

老年代GC（FullGC/MajorGC） 指发生在老年代的GC，出现FullGC一般会伴随至少一次的MajorGC，比如分配担保失败，FullGC比MajorGC慢十倍以上，当老年代不足或显示调用System.gc（）方法时，会触发FullGC。

次收集器：当年轻代堆空间紧张会被触发，相对全收集器而言，收集间隔短。

全收集器：当老年代或者持久堆空间满了，就会触发全收集操作。可以System.gc()显示启动全收集器。

分代回收器:

七种分代回收器



Serial收集器（串行收集器），单线程收集器，JDK1.3唯一的一个次生代收集器。Client模式下默认新生代收集器,执行时必须暂停其他所有工作线程（stop the world，SWT）。使用-xx:+UseSerialGC.

CMS收集器（并行） 以获取最短回收停顿时间为目标的收集器只能与serial和Parnew（新生代）收集器组合。

并行收集器（ParNew）并行收集器，使用复制算法。单核CPU新能没有serial性能好。多核CPU效率更高。CMS降低的拥堵线程等待的时间，没有完全解决SWT的问题。

Parallel scavenge，并行多线程收集器，使用复制算法，更关注系统吞吐量的问题。

Serial Old ：老年代单线程收集器，使用标记整理算法（能与所有的新生代收集器组合）

Parallel Old ：多线程收集器，使用标记整理算法，吞吐量优先（只能与parallel新生代收集器组合。）。

G1 分类收集器，面向服务端应用的收集器，主要目标时配备多颗CPU服务器治理大内存。

并行：并行的关键是有同时处理多个任务的能力。

并发：并发的关键是有处理多个任务的能力，不一定要同时

# 性能监测工具

**Jps（JVM Process Status Tools）**

Jps是参照Unix系统的取名规则命名的，而他的功能和ps的功能类似，可以列举正在运行的饿虚拟机进程并显示虚拟机执行的主类以及这些进程的唯一ID（ＬＶＭＩＤ，对应本机来说和PID相同），他的用法如下：

**Jps [option] [hostid]**

其中hostid默认为本机，而option选项包含以下选项

|  |  |
| --- | --- |
| Option | Function |
| -q | 只输出LVMID |
| -m | 输出JVM启动时传给主类的方法 |
| -l | 输出主类的全名，如果是Jar则输出jar的路径 |
| -v | 输出JVM的启动参数 |

**jstat（JVM Statistics Monitoring Tools）**

Jstat主要用于监控虚拟机的各种运行状态信息，如类的装载、内存、垃圾回收、JIT编译器等，在没有GUI的服务器上，这款工具是首选的一款监控工具。其用法如下：

jstat [option vmid [interval [s|ms] [vount] ] ]

参数interval和count分别表示查询间隔和查询次数，如每1毫秒查询一次进程20445的垃圾回收情况，监控20次，命令如下所示：

**jstat –gc 20445 1 20**

Jinfo的作用是实时查看虚拟机的各项参数信息jps –v可以查看虚拟机在启动时被显式指定的参数信息，但是如果你想知道默认的一些参数信息呢？除了去查询对应的资料以外，jinfo就显得很重要了。jinfo的用法如下：

**Jinfo [option] pid**

如 jinfo –sysprops {pid}

Jmap用于生成堆快照（heapdump）。当然我们有很多方法可以取到对应的dump信息，如我们通过JVM启动时加入启动参数 –XX:HeapDumpOnOutOfMemoryError参数，可以让JVM在出现内存溢出错误的时候自动生成dump文件，亦可以通过-XX:HeapDumpOnCtrlBreak参数，在运行时使用ctrl+break按键生成dump文件，当然我们也可以使用kill -3 pid的方式去恐吓JVM生成dump文件。Jmap的作用不仅仅是为了获取dump文件，还可以用于查询finalize执行队列、Java堆和永久带的详细信息，如空间使用率、垃圾回收器等。其运行格式如下：

Jmap [option] vmip

jhat（JVM Heap Analysis Tool）

Jhat是用来分析dump文件的一个微型的HTTP/HTML服务器，它能将生成的dump文件生成在线的HTML文件，让我们可以通过浏览器进行查阅，然而实际中我们很少使用这个工具，因为一般服务器上设置的堆、栈内存都比较大，生成的dump也比较大，直接用jhat容易造成内存溢出，而是我们大部分会将对应的文件拷贝下来，通过其他可视化的工具进行分析。启用法如下：

**Jhat {dump\_file}**

执行命令后，我们看到系统开始读取这段dump信息，当系统提示Server is ready的时候，用户可以通过在浏览器键入[http://ip地址:7000](http://yhjhappy234.blog.163.com/" \t "_blank)进行查询。

我们可以看到刚才生成的dump文件有多大

**jstack（JVM Stack Trace for java）**

Jstack用于JVM当前时刻的线程快照，又称threaddump文件，它是JVM当前每一条线程正在执行的堆栈信息的集合。生成线程快照的主要目的是为了定位线程出现长时间停顿的原因，如线程死锁、死循环、请求外部时长过长导致线程停顿的原因。通过jstack我们就可以知道哪些进程在后台做些什么？在等待什么资源等！其运行格式如下：

Jstack [option] vmid

**JConsole（JVM Monitoring and management console）**

在JDK的bin目录下，我们很容易找到jconsole.exe这个程序，双击即可启动！

VisualVM被成为是more in one的工具集，它可以实现以下功能点：

1、  显示虚拟机的进程以及进程的配置信息和环境信息(jps、jinfo)

2、  监视应用程序的CPU、内存、堆、方法区和线程信息(jstat、jstack)

3、  Dump以及分析dump的功能(jmap、jhat)

4、  离线程序快照：离线dump分析

5、  方法运行性能分析，找出调用最多，运行最长的方法块

6、  Plugings动态扩展功能

**VisualVM**

是基于netBean开发，因此天生就具有plug大量扩展的能力，我们可以通过他的插件页面轻松安装所需要的插件！

启用VisualVM工具会很醒目的告诉我们检测到一个死锁，而不需要我们在Jconsole下手动启用检测。

# 常见参数

**堆配置：**

-Xms:初始堆大小 -Xms：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表示年轻代和年老代比值为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： 3 Survivor：2，一个Survivor区占整个年轻代的1/5 -XX:MaxPermSize=n:设置持久代大小

1、一般初始堆和最大堆设置一样，因为：现在内存不是什么稀缺的资源，但是如果不一样，从初始堆到最大堆的过程会有一定的性能开销，所以一般设置为初始堆和最大堆一样。64位系统理论上可以设置为无限大，但是一般设置为**4G**,因为如果再大，JVM进行垃圾回收出现的暂停时间会比较长，这样全GC过长，影响JVM对外提供服务，所以不能太大。一般设置为4G。  
2、-XX:NewRaio和-XX:SurvivorRatio这两个参数，都是设置年轻代和年老代的大小的，设置一个即可，第一是设置年轻代的大小，第二个是设置比值，理论上设置一个既可以满足需求

**收集器:**

-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

打印GC回收的过程日志信息

-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:设置并行收集最大的暂停时间（如果到这个时间了，垃圾回收器依然没有回收完，也会停止回收）

-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)

-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手机方式为并行收集时，使用的CPU数。并行收集线程数

**回收器的选择**

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，一般不考虑使用了，所以这里只针对并行收集器和并发收集器。默认情况下，JDK5.0以前是使用的串行收集器，如果想使用其他收集器需要在启动时加入相应的参数，JDK5.0以后，JVM会根据系统当前的配置进行判断

**吞吐量优先的并行收集器**并行收集器主要以到达一定的吞吐量为目标，适用于后台处理

java -Xmx3550m-Xms3550m-Xss128k-XX:+UseParallelGC

-XX:ParallelGCThreads=20

-XX:+UseParallelGC:选择垃圾收集器为并行收集器。次配置仅对年轻代有效。即上述配置下，年轻代使用并行收集，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同。

-XX:+UseParallelOldGC:配置年老代来及收集方式为并行收集，JDK6.0支持对年老代并行收集

-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

-XX:+UseAdaptiveSizePolicy:设置此选项以后，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响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

**响应时间优先的并发收集器**

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。

-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

**调优总结：**

**年轻代大小选择**

响应时间优先的应用：尽可能设置大，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的成都，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8核CPU以上应用。

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，可能会造成内存碎片、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，一般需要参考一下数据获得：  
1、并发垃圾收集信息  
2、持久代并发收集次数  
3、传统GC信息  
4、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，减少中期对象，而年老代尽存放长期存活的对象

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FullGc后，对年老代进行压缩

调优方法

**调优工具**

Jconsole,jProfile,VisualVM

Jconsole:jdk自带， 功能简单，但是可以再系统有一定负荷的情况下使用，对垃圾回收算法有很详细的跟踪。  
JProfiler：商业软件，需要付费，但是功能强大  
VisualVM:JDK自带，功能强大，与Jprofiler类似，推荐

如何调优

观察内存释放情况、集合类检查，对象树  
上面这些调优工具都提供了强大的功能，但是总的来说一般分为以下几类功能：

1、堆的信息查看（年轻代、年老代、持久代分配）  
2、提供即时的垃圾回收功能呢  
3、垃圾监控，长时间监控

**内存泄露检查**

一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

持久代沾满处理：  
1、-XX:MaxPermSize=16m  
2、换JDK比如：JRocket

系统内存被沾满：  
一般是因为没有足够的资源产生线程造成的，系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大的一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，出现异常。  
分配给Java虚拟机的内存越多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比。同事，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共生产的线程数。

java程序内存问题的诊断方法：

1、jstat可以查看垃圾回收情况：jstat -gcutil pid  
2、jmap可以将java内存dump出来  
3、jstack -l 进程id  
4、eclipse插件MAT可以有效的分析内存占用情况

# JVM优化

三个点：

1.选择JVM版本，确定使用client模式还是server模式

2对于堆区的大小可以通过参数大小自己指定。

3.垃圾收集器的选择