# JDK

## JDK1.5 2004

泛型

注解

装箱

枚举

可变长参数

Foreach循环

## JDK1.6

脚本语言支持

JDBC4.0

Java编译器API

## JDK7 2011

G1

动态语言增强

64位系统中的压缩指针

NIO 2.0

## JDK8 2014

Lambda

语法增强，Java类型注解

## JDK9

模块化

# Java语言规范

语法，词法

变量

类型

文法

内存模型

类加载连接过程

# JVM规范

Class文件类型

数字的内部表示和存储

ReturnAddress数据类型定义

运行时数据

帧栈

虚拟机的启动

虚拟机的指令集

PC，堆，栈，方法区

Groovy

Clojure

Scala

一些特殊方法

clinit

init

VM指令

类型转化

l2i

栈 aload astore

运算 iadd isub

对java的支持

反射java.lang.reflect

ClassLoader

Java.security

JVM的编译

JavaP

# 蓝山老师：java背后的秘密笔

<http://itboba.com/taxonomy/term/974>

## 视屏-01

讲座内容：

java运行机理，开发高性能java原则与方法

1.简术环境搭建

2.jvm的体系结构：结构，classloader，gc

3.开发高性能java：原则和方法

4.(未录制)配置调优jvm：内存配置，gc配置

实例：电信系统

参考资料：

《java深度历险》，虚拟机架构白皮书

环境搭建：

JAVA\_HOME 安装主目录

path

classpath:java平台类库，第3方类库，自定义类库路径

安装jdk后有2个jre目录

jdk中的jre

无论是class还是jvm都需要jre支持

jdk中的jre提供java的开发环境，包含了jdk中很多工具需要的运行的环境如javac等

如何寻找jre 定位虚拟机

1查找自己(java.exe)目录下的jre

2 查询父目录下是否有jre

3 查询系统配置，注册表HKEY\_LOCAL\_MACHINE\JavaSoft\Java Runtime

Environment

## 视屏-02

JVM有自己完善的硬件架构：处理器，堆栈，寄存器，指令系统。

class字节码文件就是控制cpu的指令集。

sun规定，每个字节码指令8bit，一共可以有256条字节码指令。

Jvm由两个子系统和两个组件组成：

两个子系统：class loader + execution engine

两个组件：runtime data area + native interface

编写自己的classloader extends java.lang.ClassLoader,主要应用与中间件服务器上。

主要考虑安全性和效率

## 视屏-03

Exceution engine：执行class中的指令，相当与jvm的CPU，决定了jvm的好坏。

Native interface：与其他语言交互的接口，从此脱离jvm控制。容易出现jvm无法控制的

native heap OutofMmory

Runtime Data Area组件：就是我们常说的JVM内存了。由五部分组成：

HEAP

Method Area

Java Stack

Program Count

Native Method Stack

其中HEAP,Method Area是全局共享的，其他是线程私有的。

---------------------------------

## 视屏-04

java的类装载器

C语言静态语言：编译后无法改变

classloader使得java成为动态性语言，动态加载需要的类

两种载入方式：：

预先载入：基础类库，常驻内存，不被gc回收。

按需载入：自定义类库，使用的时候载入，能被gc释放。

按需载入：

隐式载入：new关键字

显示载入：2种方式:

Class c=Class.forName(“”);//可从参数获取

ClassLoader.loadClass();

java -verbose:class Main

实例：查看java运行细节：

|  |
| --- |
| public class Main{  public static void main(String[] args){  A a = new A();  a.print();  B b = new B();  b.print();  }  }  class A{  public void print(){  System.out.println("this is a");  }  }  class B{  public void print(){  System.out.println("this is b");  }  } |

javac Main.java

java -verbose:class Main

|  |
| --- |
| [Opened /usr/java/jdk1.6.0\_24/jre/lib/rt.jar]  [Loaded java.lang.Object from /usr/java/jdk1.6.0\_24/jre/lib/rt.jar]  [Loaded java.io.Serializable from /usr/java/jdk1.6.0\_24/jre/lib/rt.jar]  …//loaded ….. from rt.jar  [Loaded Main from file:/home/hch/c\_code/]  [Loaded A from file:/home/hch/c\_code/]  this is a  [Loaded B from file:/home/hch/c\_code/]  this is b  [Loaded java.lang.Shutdown from /usr/java/jdk1.6.0\_24/jre/lib/rt.jar]  [Loaded java.lang.Shutdown$Lock from /usr/java/jdk1.6.0\_24/jre/lib/rt.jar] |

## 视屏-05

Classoader按照继承体系，从上到下依次载入类，直到所有的祖先类都载入才轮到自己。

gc时正好相反。所以java类继承体系不宜过长，否则载入、释放性能会受到影响。

gc收集按需载入的对象。

直接使用类装载器：

先获取对象类，再

o.getClass()

我们无法构建class类，是在其第一次加载时创建。其构造方法是私有的。

获取Class的两种方法，1 从类的静态域获取，2从对象的getClass()方法获取

Class c = String.class;

c= str.getClass();

ClassLoader cl.getClassLoader();

获取classloader两种方法

1 获取class再从class获取classloader

2 直接从类获得，Integer.getClassLoader

获取classloader后就载入需要的类了：

Class c = classLoader.loadClass("ClassName");

从类获取对象O o = c. newInstance();

类有一个属性，指向classLoader。

|  |
| --- |
| //获取class  Person p = **new** Person();  Class<? **extends** Person>clazz0 = p.getClass();  Class<Person> clazz1 = Person.**class**;    //获取ClassLoader  ClassLoader classloader = clazz1.getClassLoader();  //加载class  Class<?> clazz = classloader.loadClass("com.hc.study.Persion");    //反射创建对象  Object person = clazz.newInstance(); |

## 视屏-06

Java.exe依照底下的顺序来查找JRE：

自己的目录下有没有JRE；

父目录有没有JRE；

查询注册表：

[HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment]

1运行java.exe

5 BootstrapLoader

3定位JVM

7 AppClassLoader

parent

2定位JRE

6 ExtClassLoader

parent

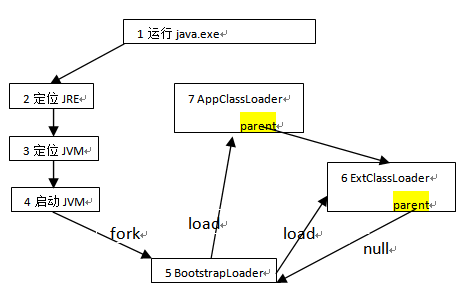
4启动JVM

null

load

load

fork



Java.exe执行的时候，选择jre，激活jre中的JVM，JVM一旦激活，会首先做一些初始化工作，然后就会初始化类装载器，并形成类装载器层次体系。

JVM初始化后就生成一个特殊的进程加载Bootstrap Loader

Bootstrap Loader做两件事：

BootstrapLoader会加载一个特殊的装载器：ExtClassLoader,并设定其parent=null;因为在java的环境中，BootstrapLoader是不存在的。

BootstrapLoader再加载另一个装载器：AppClassLoader，并设定其parent=ExtClassLoader;

BootstrapLoader又叫初始类装载器，也叫系统引导类装载器，首先会搜索系统参数sun.boot.class.path设定的路径并加载其中的类，然后会加载ExtClassLoader和AppClassLoader并启动他们，同时设定他们各自的parent指针。

ExtClassLoader又叫扩展类装载器，ExtClassLoader会搜索java.ext.dirs系统参数所指定的路径，并加载其中的类

AppClassLoader又叫系统类加载器(这个名字容易混淆)，其搜索java.class.path所指定的路径，并加载其中的类.加载应用真正需要的类，开始应用程序

|  |
| --- |
| **sun.boot.class.path=/usr/local/jdk1.6.0\_20/jre/lib/resources.jar:/usr/local/jdk1.6.0\_20/jre/lib/rt.jar:/usr/local/jdk1.6.0\_20/jre/lib/sunrsasign.jar:/usr/local/jdk1.6.0\_20/jre/lib/jsse.jar:/usr/local/jdk1.6.0\_20/jre/lib/jce.jar:/usr/local/jdk1.6.0\_20/jre/lib/charsets.jar:/usr/local/jdk1.6.0\_20/jre/classes**  **java.ext.dirs=/usr/local/jdk1.6.0\_20/jre/lib/ext:/usr/java/packages/lib/ext**  **java.class.path=.:/usr/local/jdk1.6.0\_20/lib.jar:/usr/local/jdk1.6.0\_20/tools.jar** |
| Eclipse启动  **sun.boot.class.path**=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\resources.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\rt.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\sunrsasign.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jsse.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jce.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\charsets.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\modules\jdk.boot.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\classes  **java.ext.dirs**=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\ext;C:\Windows\Sun\Java\lib\ext  **java.class.path**=D:\workspace-stu\t\bin |
| **cmd启动**  **sun.boot.class.path=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\resources.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\rt.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\sunrsasign.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jsse.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jce.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\charsets.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\modules\jdk.boot.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\classes**  **java.ext.dirs=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\ext;C:\Windows\Sun\Java\lib\ext**  **java.class.path=.** |

完成上述过程后就形成了类装载器层次体系。

之所以要存在层次体系，适因为要实现所谓的类装载委托模型。该模型就是类装载器载入类时，首先请求其parant使用其搜索路径载入，如果parent找不到，才有自己依照自己的搜索路径来搜索类。

类装载器为什么要这样设计，除了动态性目的外，主要目的就是为了安全(java安全架构是由华裔科学家宫力博士设计).

## 视屏-07类装载器

委托模型：

AppClassLoader parent

ExtClassLoader parent

Bootstrap parent

User2ClassLoader parent

User1ClassLoader parent

用户自定义的类装载器不可能装载应该由父装载器装载的可靠类，从而防止不可靠代码替代父装载器中的可靠代码。

隔离性：

两个位于不同分支的类装载器具有隔离性，这种隔离性使得他们装载同一个类时会在内存出现两个Class类，但互不可见。

命名空间：

每个类装载器有自己的命名空间；不同命名空间的两个类是不可见的，但只要得到类所对应的Class对象的reference，还是可以反问另一命名空间的类。

？？下面代码需要补齐：

|  |
| --- |
| **publicclass** MyClassLoader **extends** ClassLoader{  @Override  **public** Class<?> loadClass(String name) **throws** ClassNotFoundException {  **returnsuper**.loadClass(name);  }  } |

Java虚拟机类装载：原理、实现与应用

http://developer.51cto.com/art/200508/1380.htm

## 视屏-08 GC机制

Java的特性：跨平台，自动垃圾收集。

垃圾收集的实现

1引用计数，早版本

优：快；交织在程序过程中快

缺：不能识别循环引用，由于计数额外开销；

2跟踪收集

优：可以识别循环引用，不会交织在程序过程中

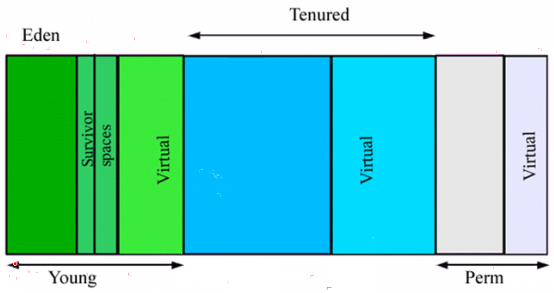
缺：需要维护对象引用全景图，增加了内存的消耗。

3基于对象**跟踪**的**分代增量**收集

优：

缺：算法复杂

三个区域：YongGen，OldGen，PermGen



一般来说重型对象声明后不会马上清除，所以在OldGen中声明。

Eden：伊甸园，万物初始

survivor：n幸存者

Tenure: 享有

Tenured：享有终身职位的

## 视屏-10-16：编写高性能java程序的原则

原则：设计原则，编码原则

方法

设计原则：

很多性能和java本身无关，是由于不当的设计，拙劣的编码引起的。

1）先把焦点放在设计，数据结构，算法上。

2）不要依赖于编译器优化技术，正确理解并使用运行期优化

javac -o 编译时会优化代码。

jit 运行期优化，转换成机器码并优化。IBM最优化。

好的jit技术加上合理使用可使某些java代码达到甚至超过C++，逼近C性能。

3) 使用StringBuffer

4）降低对象的创建成本（慎用重型对象，对象池）

5）降低同步化带来的影响

6）尽可能使用stack变量，基本类型完成任务。

常量池访问

常量池转化为栈变量访问

静态属性访问

静态属性转化为栈变量访问

7）使用static，final，private函数以促使inlining

(编译时或运行时)jdk会判断，如果函数体比较短，可能变成inlining

缺点：不能被集成，只有多次调用inline才会提升性能。

编译期inling不能动态跟新被inlining的类

8）实例变量初始化一次就好

分配内存空间

默认值初始化

实例初始化，静态初始化

构造函数

9) 集群类对象的处理(数组，集合等)

使用正确的合适的集群类

复制：循环复制还是 System.arraycopy

遍历：循环，各种迭代器：Iteretor，Listerator, Enumeration; 循环比较高效。

优先选择数组，然后选择集群类；

10) 重用对象

11）使用缓式评估技术()

合理做到功能分离，方法尽量短小

采用合理代码编写技巧，编译期生成高效字节码，分担jit负担

常量池访问

常量池转化为栈变量访问

静态属性访问

静态属性转化为栈变量访问

## 视屏-17

未录制完成

## java -h

|  |
| --- |
| [admin@localhost test]$ java -h  Usage: java [-options] class [args...]  (to execute a class)  or java [-options] -jar jarfile [args...]  (to execute a jar file)  where options include:  -d32 use a 32-bit data model if available  -d64 use a 64-bit data model if available  -server to select the "server" VM  The default VM is server.  -cp <class search path of directories and zip/jar files>  -classpath <class search path of directories and zip/jar files>  A : separated list of directories, JAR archives,  and ZIP archives to search for class files.  -D<name>=<value>  set a system property  -verbose[:class|gc|jni]  enable verbose output  -version print product version and exit  -version:<value>  require the specified version to run  -showversion print product version and continue  -jre-restrict-search | -jre-no-restrict-search  include/exclude user private JREs in the version search  -? -help print this help message  -X print help on non-standard options  -ea[:<packagename>...|:<classname>]  -enableassertions[:<packagename>...|:<classname>]  enable assertions  -da[:<packagename>...|:<classname>]  -disableassertions[:<packagename>...|:<classname>]  disable assertions  -esa | -enablesystemassertions  enable system assertions  -dsa | -disablesystemassertions  disable system assertions  -agentlib:<libname>[=<options>]  load native agent library <libname>, e.g. -agentlib:hprof  see also, -agentlib:jdwp=help and -agentlib:hprof=help  -agentpath:<pathname>[=<options>]  load native agent library by full pathname  -javaagent:<jarpath>[=<options>]  load Java programming language agent, see java.lang.instrument  -splash:<imagepath>  show splash screen with specified image |

## 视屏-06 系统属性系统环境变量

下面是windows和linux下系统环境变量和系统属性

|  |
| --- |
| **package** com.hc.study;  **import** java.util.Map;  **import** java.util.Properties;  **publicclass** JavaSystemProperties {  **publicstaticvoid** main(String[] args) {  System.*out*.println("System.getenv():");  Map<String, String> env = System.*getenv*();  **for**(String key : env.keySet() ){  System.*out*.println(key + "=" + env.get(key));  }  System.*out*.println("-------------------");  System.*out*.println("System.getProperties():");  Properties properties = System.*getProperties*();  **for**(Object key : properties.keySet()){  System.*out*.println(key + "=" + properties.get(key));  }  }  } |
| **D:\workspace-stu\t\bin>java -cp . com.hc.study.JavaSystemProperties**  **System.getenv():**  **USERPROFILE=C:\Users\Administrator**  **ProgramData=C:\ProgramData**  **PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC**  **JAVA\_HOME=D:\0\_java\jdk1.6.0\_23\_32**  **ProgramFiles(x86)=C:\Program Files (x86)**  **=ExitCode=00000001**  **TEMP=C:\Users\ADMINI~1\AppData\Local\Temp**  **SystemDrive=C:**  **ProgramFiles=C:\Program Files (x86)**  **Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ICBCEbankTools\Gemplus\GemSafe Libraries\BIN\;D:\0\_java\jdk1.6.0\_23\_32\bin;D:\0\_java\maven-2.2.1\bin;C:\Program Files (x86)\IDM Computer Solutions\UltraEdit\**  **HOMEDRIVE=C:**  **PROCESSOR\_REVISION=0f0a**  **\_DFX\_INSTALL\_UNSIGNED\_DRIVER=1**  **USERDOMAIN=HCH**  **ALLUSERSPROFILE=C:\ProgramData**  **ProgramW6432=C:\Program Files**  **PROCESSOR\_IDENTIFIER=Intel64 Family 6 Model 15 Stepping 10, GenuineIntel**  **SESSIONNAME=Console**  **TMP=C:\Users\ADMINI~1\AppData\Local\Temp**  **CommonProgramFiles=C:\Program Files (x86)\Common Files**  **=::=::\**  **LOGONSERVER=\\HCH**  **M2\_HOME=D:\0\_java\maven-2.2.1**  **PROCESSOR\_ARCHITECTURE=x86**  **FP\_NO\_HOST\_CHECK=NO**  **OS=Windows\_NT**  **PROCESSOR\_ARCHITEW6432=AMD64**  **HOMEPATH=\Users\Administrator**  **PROMPT=$P$G**  **PROCESSOR\_LEVEL=6**  **CommonProgramW6432=C:\Program Files\Common Files**  **=D:=D:\workspace-stu\t\bin**  **LOCALAPPDATA=C:\Users\Administrator\AppData\Local**  **COMPUTERNAME=HCH**  **windir=C:\Windows**  **SystemRoot=C:\Windows**  **NUMBER\_OF\_PROCESSORS=2**  **USERNAME=Administrator**  **PUBLIC=C:\Users\Public**  **PSModulePath=C:\Windows\system32\WindowsPowerShell\v1.0\Modules\**  **CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files**  **ComSpec=C:\Windows\system32\cmd.exe**  **APPDATA=C:\Users\Administrator\AppData\Roaming**  **-------------------**  **System.getProperties():**  **java.runtime.name=Java(TM) SE Runtime Environment**  **sun.boot.library.path=D:\0\_java\jdk1.6.0\_23\_32\jre\bin**  **java.vm.version=19.0-b09**  **java.vm.vendor=Sun Microsystems Inc.**  **java.vendor.url=http://java.sun.com/**  **path.separator=;**  **java.vm.name=Java HotSpot(TM) Client VM**  **file.encoding.pkg=sun.io**  **sun.java.launcher=SUN\_STANDARD**  **user.country=CN**  **sun.os.patch.level=**  **java.vm.specification.name=Java Virtual Machine Specification**  **user.dir=D:\workspace-stu\t\bin**  **java.runtime.version=1.6.0\_23-b05**  **java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment**  **java.endorsed.dirs=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\endorsed**  **os.arch=x86**  **java.io.tmpdir=C:\Users\ADMINI~1\AppData\Local\Temp\**  **line.separator=**  **java.vm.specification.vendor=Sun Microsystems Inc.**  **user.variant=**  **os.name=Windows 7**  **sun.jnu.encoding=GBK**  **java.library.path=D:\0\_java\jdk1.6.0\_23\_32\bin;.;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ICBCEbankTools\Gemplus\GemSafe Libraries\BIN\;D:\0\_java\jdk1.6.0\_23\_32\bin;D:\0\_java\maven-2.2.1\bin;C:\Program Files (x86)\IDM Computer Solutions\UltraEdit\**  **java.specification.name=Java Platform API Specification**  **java.class.version=50.0**  **sun.management.compiler=HotSpot Client Compiler**  **os.version=6.1**  **user.home=C:\Users\Administrator**  **user.timezone=**  **java.awt.printerjob=sun.awt.windows.WPrinterJob**  **file.encoding=GBK**  **java.specification.version=1.6**  **sun.boot.class.path=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\resources.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\rt.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\sunrsasign.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jsse.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\jce.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\charsets.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\lib\modules\jdk.boot.jar;D:\0\_java\jdk1.6.0\_23\_32\jre\classes**  **java.ext.dirs=D:\0\_java\jdk1.6.0\_23\_32\jre\lib\ext;C:\Windows\Sun\Java\lib\ext**  **java.class.path=.**  **user.name=Administrator**  **java.vm.specification.version=1.0**  **java.home=D:\0\_java\jdk1.6.0\_23\_32\jre**  **sun.arch.data.model=32**  **user.language=zh**  **java.specification.vendor=Sun Microsystems Inc.**  **awt.toolkit=sun.awt.windows.WToolkit**  **java.vm.info=mixed mode, sharing**  **java.version=1.6.0\_23**  **java.vendor=Sun Microsystems Inc.**  **file.separator=\**  **java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport.cgi**  **sun.io.unicode.encoding=UnicodeLittle**  **sun.cpu.endian=little**  **sun.desktop=windows**  **sun.cpu.isalist=pentium\_pro+mmx pentium\_pro pentium+mmx pentium i486 i386 i86** |
| **[admin@localhost test]$ java JavaSystemProperties**  **System.getenv():**  **TERM=linux**  **OLDPWD=/home/admin**  **SHELL=/bin/bash**  **INPUTRC=/etc/inputrc**  **HISTSIZE=1000**  **JAVA\_HOME=/usr/local/jdk1.6.0\_20**  **SHLVL=1**  **CLASSPATH=.:/usr/local/jdk1.6.0\_20/lib.jar:/usr/local/jdk1.6.0\_20/tools.jar**  **XFILESEARCHPATH=/usr/dt/app-defaults/%L/Dt**  **PATH=/usr/kerberos/bin:/usr/local/jdk1.6.0\_20/bin:/usr/local/bin:/bin:/usr/bin:/home/admin/bin**  **MAIL=/var/spool/mail/admin**  **USER=admin**  **HOSTNAME=localhost.localdomain**  **PWD=/home/admin/test**  **HOME=/home/admin**  **LOGNAME=admin**  **LESSOPEN=|/usr/bin/lesspipe.sh %s**  **CVS\_RSH=ssh**  **G\_BROKEN\_FILENAMES=1**  **LS\_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=01;32:\*.cmd=01;32:\*.exe=01;32:\*.com=01;32:\*.btm=01;32:\*.bat=01;32:\*.sh=01;32:\*.csh=01;32:\*.tar=01;31:\*.tgz=01;31:\*.arj=01;31:\*.taz=01;31:\*.lzh=01;31:\*.zip=01;31:\*.z=01;31:\*.Z=01;31:\*.gz=01;31:\*.bz2=01;31:\*.bz=01;31:\*.tz=01;31:\*.rpm=01;31:\*.cpio=01;31:\*.jpg=01;35:\*.gif=01;35:\*.bmp=01;35:\*.xbm=01;35:\*.xpm=01;35:\*.png=01;35:\*.tif=01;35:**  **\_=/usr/local/jdk1.6.0\_20/bin/java**  **NLSPATH=/usr/dt/lib/nls/msg/%L/%N.cat**  **LD\_LIBRARY\_PATH=/usr/local/jdk1.6.0\_20/jre/lib/amd64/server:/usr/local/jdk1.6.0\_20/jre/lib/amd64:/usr/local/jdk1.6.0\_20/jre/../lib/amd64**  **LANG=en\_US.UTF-8**  **-------------------**  **System.getProperties():**  **java.runtime.name=Java(TM) SE Runtime Environment**  **java.home=/usr/local/jdk1.6.0\_20/jre**  **sun.boot.library.path=/usr/local/jdk1.6.0\_20/jre/lib/amd64**  **sun.boot.class.path=/usr/local/jdk1.6.0\_20/jre/lib/resources.jar:/usr/local/jdk1.6.0\_20/jre/lib/rt.jar:/usr/local/jdk1.6.0\_20/jre/lib/sunrsasign.jar:/usr/local/jdk1.6.0\_20/jre/lib/jsse.jar:/usr/local/jdk1.6.0\_20/jre/lib/jce.jar:/usr/local/jdk1.6.0\_20/jre/lib/charsets.jar:/usr/local/jdk1.6.0\_20/jre/classes**  **java.ext.dirs=/usr/local/jdk1.6.0\_20/jre/lib/ext:/usr/java/packages/lib/ext**  **java.class.path=.:/usr/local/jdk1.6.0\_20/lib.jar:/usr/local/jdk1.6.0\_20/tools.jar**  **java.library.path=/usr/local/jdk1.6.0\_20/jre/lib/amd64/server:/usr/local/jdk1.6.0\_20/jre/lib/amd64:/usr/local/jdk1.6.0\_20/jre/../lib/amd64:/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib**  **java.vm.version=16.3-b01**  **java.vm.vendor=Sun Microsystems Inc.**  **java.vendor.url=http://java.sun.com/**  **path.separator=:**  **java.vm.name=Java HotSpot(TM) 64-Bit Server VM**  **file.encoding.pkg=sun.io**  **sun.java.launcher=SUN\_STANDARD**  **user.country=US**  **sun.os.patch.level=unknown**  **java.vm.specification.name=Java Virtual Machine Specification**  **user.dir=/home/admin/test**  **java.runtime.version=1.6.0\_20-b02**  **java.awt.graphicsenv=sun.awt.X11GraphicsEnvironment**  **java.endorsed.dirs=/usr/local/jdk1.6.0\_20/jre/lib/endorsed**  **os.arch=amd64**  **java.io.tmpdir=/tmp**  **line.separator=**  **java.vm.specification.vendor=Sun Microsystems Inc.**  **os.name=Linux**  **sun.jnu.encoding=UTF-8**  **java.specification.name=Java Platform API Specification**  **java.class.version=50.0**  **sun.management.compiler=HotSpot 64-Bit Server Compiler**  **os.version=2.6.18-164.el5**  **user.home=/home/admin**  **user.timezone=**  **java.awt.printerjob=sun.print.PSPrinterJob**  **file.encoding=UTF-8**  **java.specification.version=1.6**  **user.name=admin**  **java.vm.specification.version=1.0**  **sun.arch.data.model=64**  **user.language=en**  **java.specification.vendor=Sun Microsystems Inc.**  **java.vm.info=mixed mode**  **java.version=1.6.0\_20**  **java.vendor=Sun Microsystems Inc.**  **file.separator=/**  **java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport.cgi**  **sun.io.unicode.encoding=UnicodeLittle**  **sun.cpu.endian=little**  **sun.cpu.isalist=** |

# 深入理解Java虚拟机：JVM高级特性与最佳实践1.周志明

## 概述

本书示例使用JDK1.6

版本问题：Jdk1.5以后，

公开版本号(Product Version)是：JDK5,JDK6,JDK7

程序员内部使用的开发版本号(Developer Version)是：1.5,1.6,1.7

即java -verison 的输出

## 参考资料

勘误表：<http://icyfenix.iteye.com/blog/1119214>

莫

《Java虚拟机规范》

## 编译JDK1.7

下载<http://download.java.net/openjdk/jdk7/>

openjdk-7-fcs-src-b147-27\_jun\_2011.zip

参考

<http://wendal.net/420.html>

成功编译OpenJDK 7u2 ! 哦也！

# ClassLoader

过程：加载----->验证------>准备------>解析------>初始化

<http://blog.csdn.net/woshichuanqihan/article/details/49229641>

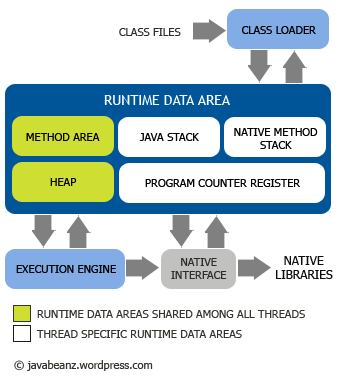
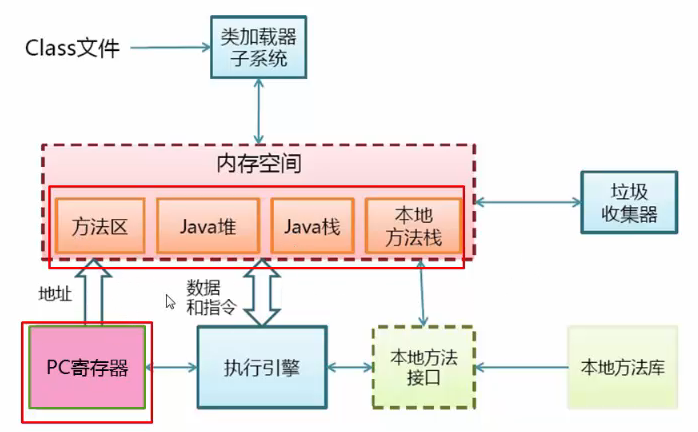
# 《分布式java应用基础与实践》笔记

作者博客：http://bluedavy.com/

# 内容

java内存模型，分配回收算法，跟踪JVM内存使用情况，判断程序运行状况，性能调优

# JVM的内部结构



JVM主要包括两个子系统和两个组件。两个子系统分别是Class loader子系统和Execution engine(执行引擎) 子系统；两个组件分别是Runtime data area (运行时数据区域)组件和Native interface(本地接口)组件。

## Class loader子系统

Class loader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 Runtime data area中的method area(方法区域)。

JVM运行时会产生三个ClassLoader

ClassLoader根装载器，C++，装载jre核心类库；

ExtClassLoader扩展类装载器，装载jre扩展目录ext下的类；

AppClassLoader系统装载器，装载classpath下的类包。

JVM装载类时使用“**全盘负责委托机制**”

全盘负责：当classloader装载一个类的时候，除非显示使用另一个classloader，该类所依赖的其它类也由这个classloader装载。

委托机制：先委托父装载器寻找目标类，只有在找不到的情况下才从自己的类路劲中查找并装载目标类。

## Execution engine子系统

Execution engine子系统的作用：执行classes中的指令。任何JVM specification实现(JDK)的核心都是Execution engine，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。

## Native interface组件

Native interface组件：与native libraries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native heap OutOfMemory。

## Runtime Data Area组件

Runtime Data Area组件：就是我们常说的JVM的内存了。它主要分为五个部分：  
1、Method Area(方法区域)：

存放被装载的class的信息(名称，修饰符等)、类中的静态变量、常量、Filed信息、方法信息。当调用Class对象的getName，isInterface等方法获取信息时，这些数据来源于方法区。

方法区也叫持久带，Permanet Generation。默认最小值16M，最大值64M。

可通过-XX:PermSize 及 -XX:MaxPermSize来指定。

方法区全局共享，一定条件下也会被GC，（GC运行时加载的类）。

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。

2、Heap (堆)：

一个Java虚拟机实例中只存在一个堆空间。

堆存储对象实例，数组值。可以认为new创建的对象都分配在堆中。

32位最大2G，64位没有上限。

默认最大、最小值是物理内存的1/64、1/4,且均小于1GB。

通过-Xms，-Xmx来控制。

默认JVM空闲空间小于40%时，增大到最大值；大于70%时减小到最小值。

可通过参数指定：-XX:MinHeapFreeRatio=40,-XX:MaxHeapFreeRatio=70

垃圾回收就是对堆的回收，分代回收

|  |  |  |  |
| --- | --- | --- | --- |
| Eden | S0/from/survivor | S1/to | Old Ganetation |
| 包括tlab |  |  |  |
| New Generation | | | Tenuring Generation |

Eden space和survivor可动态调整

-Xms指定新生代大小

3、Java Stack(java的栈)：

虚拟机只会直接对Java stack执行两种操作：以帧为单位的压栈或出栈

5、Native method stack(本地方法栈)：

保存native方法进入区域的地址

4、Program Counter(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。

以上五部分只有Heap 和Method Area是被所有线程的共享使用的；而Java stack, Program counter 和Native method stack是以线程为粒度的，每个线程独自拥有自己的部分。

# JVM内存管理

## PC程序计数器

每个线程一个，总是指向下一条指令地址

执行本地方法是，指向未定义

## Java栈

线程私有

栈由一系列帧组成(因此Java栈也叫作帧栈)

帧保存一个方法的局部变量，操作数栈，常量池指针

每一次方法调用创建一个帧，并压栈

Java没有寄存器，所有参数传递使用操作数栈

局部变量表，槽位：

静态方法，第0个槽位是第1个参数；

对象方法，第0个槽位是referent this；

小对象(一般几十个bytes)在没有逃逸的情况下，可以直接分配在栈上；

大对象或者逃逸对象，无法栈上分配；

栈上分配永远不会出现内存泄漏，可以自动回收，减轻GC压力。

## eg:栈上分配示例

|  |
| --- |
| /\*\*  \* 栈上分配测试用例  \*  \* 1 栈上分配运行，没有GC  \* java -server -Xmx10m -Xms10 -XX:+DoEscapeAnalysis -XX:+PringGC  \* 输出5  \*  \* 2 堆上分配运行，大量GC  \* java -server -Xmx10m -Xms10 -XX:-DoEscapeAnalysis -XX:+PringGC  \* 输出500  \*  \* **@author** hc  \*/  **public** **class** OnStackTest {  **public** **static** **void** alloc(){  **byte**[] b = **new** **byte**[2];  b[0] = 1;  }    **public** **static** **void** main(String[] args) {  System.***out***.println("start");  **long** start = System.*currentTimeMillis*();  **for**(**int** i=0; i<1000000; i++){  *alloc*();  }  **long** end = System.*currentTimeMillis*();  System.***out***.println("end tack time="+(end-start));  }  } |

## 本地方法栈

HotSpot的本地方法栈和虚拟机栈合二为一；

## 方法区(永久代x)

保持类的信息，常量

通常与永久区关联在一起

\*PermGen space: Permannent Generation space

\*也叫永久代，但对于其他的虚拟机，没有永久代的说法；

\*HotSpot也打算放弃永久代，采用Native Memory来实现方法区；

\*线程共享

\*存储类信息，常量，静态变量，即时编译器编译后的代码；

\*java虚拟机规范把方法区描述为堆的一个逻辑部分

\*别名NonHeap(非堆)

\*这个区可以不做垃圾回收，回收的目的主要是回收常量池和对类型的卸载；

JDK6的时候，String等常量在方法区总

JDK7的时候，在堆中了

方法区通常和永久区Perm关联在一起；

永久区只是保存一些相对稳定的东西，不代表里面的东西永远不会被回收

## 运行时常量池

\*常量池Constant PoolTable；

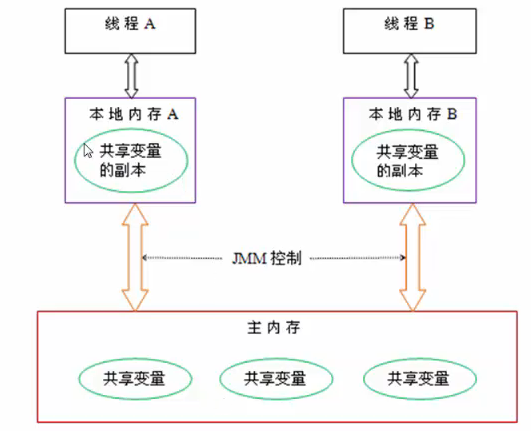
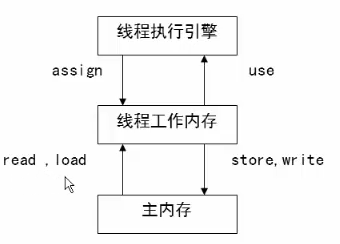
\*方法区的一部分，存放编译期生成的各种字面量和符号引用；

## Java堆

\*虚拟机规范中描述：所有的对象实例及数组都要在堆上分配；

\*但随着JIT编译器的发展，逃逸分析技术的成熟，栈上分配，标量替换等优化技术导致所有对象堆

# JVM内存模型



每个线程都有一个工作内存和主存独立；

工作内存存放主存中变量值的拷贝；

一个线程更新的值不会马上被其它线程可见，需要使用volatile

Volatile不能替代锁，一般来说性能比锁好，但不绝对；

可见性：

一个线程修改了变量后，其它线程可以立即知道

保证可见性的3种方法

volatile；

synchronized；

final(常量，一旦初始化完成，其它线程可见)

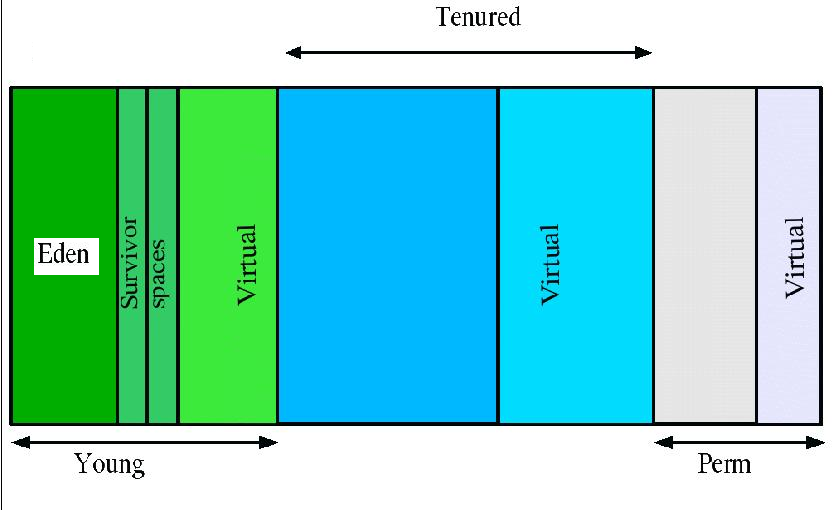
## 指令重排

指令重排，会破坏并行线程间的有序性；

加同步后串行可以避免

|  |
| --- |
| **int** a=0;  **boolean** flag = **false**;  **public** **void** write(){  a=1;  flag=**true**;  }  /\*\*  \* 两线程同时运行write，read时，read有可能返回0  \* **@return**  \*/  **public** **int** read(){  **if**(flag){  **return** a;  };  **return** -1;  } |

# JVM内存划分



Java堆中的各代分布（perm不属于堆）

年轻代，新生代

年老代

Sun的JVM Generational Collecting(垃圾回收)把内存分为

新生代/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)

-Xmx设置的大小 = PSYoungGen + ParOldGen

PSYoungGen = Eden + from(没有to)

比如下列日志

java -XX:+PrintGCDetails -Xmx512m -Xms512m T1

Heap

PSYoungGen total 174592K, used 27979K [0x00000000f5500000, 0x0000000100000000, 0x0000000100000000)

eden space 174080K, 16% used [0x00000000f5500000,0x00000000f7052d88,0x00000000fff00000)

from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)

to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)

ParOldGen total 349696K, used 468K [0x00000000dff80000, 0x00000000f5500000, 0x00000000f5500000)

object space 349696K, 0% used [0x00000000dff80000,0x00000000dfff5070,0x00000000f5500000)

PSPermGen total 21504K, used 2531K [0x00000000dad80000, 0x00000000dc280000, 0x00000000dff80000)

object space 21504K, 11% used [0x00000000dad80000,0x00000000daff8e60,0x00000000dc280000)

不同的GC会显示不同的结构，有时候也如下显示：

def new generation

tenured generation

compacting perm gen

the space 12288K

ro space 10240K 制度

rw space 12288k jdk5以后，会有共享区间，所有的JVM可以使用；

## Young

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(需要注意，Survivor的两个区是对称的，没先后关系，而且，Survivor区总有一个是空的。  )

## Tenured

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。

## Perm

Permanent Generation space，存放Class和Meta信息。

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。

## 例子

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，一部分幸存的对象会被拷贝至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Method Area，不属于Heap。

## 建议

以下转载热衷于钻研技术的哥们Richen Wang关于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。  
3、JVM调优：如果在没有出现内存泄露且上面两种办法都不能保证内存的回收时，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。

推荐的两款内存检测工具  
1、jconsole JDK自带的内存监测工具，路径jdk bin目录下jconsole.exe，双击可运行。连接方式有两种，第一种是本地方式如调试时运行的进程可以直接连，第二种是远程方式，可以连接以服务形式启动的进程。远程连接方式是：在目标进程的jvm启动参数中添加“-Dcom.sun.management.jmxremote.port=1090 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false”。1090是监听的端口号具体使用时要进行修改，然后使用IP加端口号连接即可。通过该工具可以监测到当时内存的大小，CPU的使用量以及类的加载，还提供了手动gc的功能。优点是效率高，速度快，在不影响进行运行的情况下监测产品的运行。缺点是无法看到类或者对象之类的具体信息。使用方式很简单点击几下就可以知道功能如何了，确实有不明白之处可以上网查询文档。

2、JProfiler 收费的工具，但是到处都有破解办法。安装好以后按照配置调试的方式配置好一个本地的session即可运行。可以监测当时的内存、CPU、线程等，能具体的列出内存的占用情况，还可以就某个类进行分析。优点很多，缺点太影响速度，而且有的类可能无法被织入方法，例如我使用jprofiler时一直没有备份成功过，总会有一些类的错误。

# 《深入java虚拟机学习》笔记

Java的体系结构：

Java程序设计语言

Java class文件格式

Java应用编程接口API

Java虚拟机

|  |  |
| --- | --- |
|  |  |
|  |  |

本书针对Java SDK 1.2

## 术语

类型：java中每一个变量和表达式都有编译时可以确认的类型，或基本类型，或引用类型，变量或表达式的类型决定了其所拥有的值的范围、种类、支持的操作及操作的含义。、

接口

类

类型：包含类和接口

对象

## Java体系结构介绍

Java体系结构包含4个独立的相关技术：

Java程序设计语言

Java class文件格式

Java应用编程接口API

Java虚拟机

Java API和java虚拟机组成一个java平台，也叫java运行时

Java虚拟机主要任务就是装载class文件并执行其中的字节码。

Java虚拟机的执行引擎各不相同：

一次性解释字节码，最简单

即时编译器(just-in-time compiler)：第一次被执行时，编译成本地机器码，执行快，耗内存。

自适应优化器，jvm开始时解释执行，对活动频繁的代码编译成本机代码。基本上80%的代码会被编译。

硬件虚拟机

有时候java虚拟机也叫java解释器，但并不确切，不代表任何技术含义。

Java虚拟机是有主机上软件实现的。

启动(bootstrap)类装载器是系统中唯一的，是jvm实现的一部分，和jvm使用相同语言实现。

用户类装载器，java语言编写。

每个类装载器有自己的独立命名空间。实现类隔离

P28…

## 可移植性

## 安全性

## 网络

## Java虚拟机内部体系结构详解

## Class文件

## 类生命周期

## 连接模型，class装载

## 垃圾收集

## Jvm指令

## 类型转换

## 整数运算

## 位运算，逻辑运算

## 浮点数

## 创建和操作对象、数组

## 条件，分支操作指令

## 异常

## Finally

## 方法调用

## 监视器(锁、解锁)

# Jvm启动流程

java命令启动，后面跟启动类

装在配置，在当前路径寻找jvm.cfg

根据cfg寻找JVM.dll(JVM主要实现)

初始化JVM，获得JNIEnv接口(JNIEnv是JVM的接口，findClass等操作通过它实现)

找到main方法并运行

# Jvm启动参数

## JVM参数

java启动参数共分为三类；

1是标准参数(-)，所有的JVM实现都必须实现这些参数的功能，而且向后兼容；

2是非标准参数（-X），默认jvm实现这些参数的功能，但是并不保证所有jvm实现都满足，且不保证向后兼容；

3是Stable(非稳定)参数（-XX），此类参数各个jvm实现会有所不同，将来可能会随时取消，需要慎重使用；

* 布尔型参数选项：-XX:+ 打开， -XX:- 关闭。（比如-XX:+PrintGCDetails）

## Trace跟踪参数

-verbose:gc

-XX:+printGC 打印简单GC信息

-XX:+PrintGCDetails 打印详细GC,程序末尾会打印堆栈信息

-XX:+PrintGCTimeStamps 打印GC发生的时间戳

-Xloggc:log/gc.log 指定gc log的位置，

-XX:+PrintHeapAtGC 在GC前后打印堆信息，就是GCDetails中的信息

-XX:+TraceClassLoading 监控类的加载

-XX:+PrintClassHistogram 当Ctrl+Break时，打印类的信息:序号,实例数量,总大小,类型

## 堆分配参数

-Xmx20m

-Xms5m

-Xmn10m 新生代大小

-XX:NewRatio=4 新生代(eden+2\*s)和年老代的比值,4表示年轻代占堆的1/5

-XX:SurvivorRatio=8 survivor和eden的比值，如1:8,一个survivor占年轻代1/10

在32Bit操作系统上有1.5G-2G的限制，而64Bit的就没有。

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64但小于1G。

JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4但小于1G。

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制，可以由-XX:MinHeapFreeRatio=指定。

默认空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制，可以由-XX:MaxHeapFreeRatio=指定。

服务器一般设置-Xms、-Xmx相等，上面的两个参数没啥用。

System.***out***.println("maxMemory ="+Runtime.*getRuntime*().maxMemory()/1024.0/1024 + "M");

System.***out***.println("freeMemory ="+Runtime.*getRuntime*().freeMemory()/1024.0/1024 + "M");

System.***out***.println("totalMemory="+Runtime.*getRuntime*().totalMemory()/1024.0/1024 + "M");

maxMemory =19.375M

freeMemory =4.345982513427734M

totalMemory=4.875M

## OOM

-XX:+HeapDumpOnOutOfMemoryError 当OOM时导出堆到文件

-XX:+HeapDumpPath=d:/a.dump 导出堆的文件路径

-XX:OnOutOfMemoryError=d:/tools/jdk1.7/bin/printstack.bat %p OOM是执行一个脚本

%p是当前进程的PID; printstack.bat = jstack -F %1 > D:/a.txt

脚本中：可以生成线程的dump文件，发送邮件，重启程序。。。

## eg: HeapDumpOnOutOfMemoryError

|  |
| --- |
| /\*\*  \* -Xmx20m -Xms20m -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump  \*  \* **@author** hc  \*  \*/  **public** **class** OnStackTest {  **public** **static** **void** main(String[] args) {  List<**byte**[]> list = **new** ArrayList<**byte**[]>();  **long** start = System.*currentTimeMillis*();  **for**(**int** i=0; i<100; i++){  list.add(**new** **byte**[1024\*1024]);  }  **long** end = System.*currentTimeMillis*();  System.***out***.println("end tack time="+(end-start));  }  }  [GC (Allocation Failure) [PSYoungGen: 4852K->504K(6144K)] 4852K->4664K(19968K), 0.0221493 secs] [Times: user=0.02 sys=0.01, real=0.02 secs]  [GC (Allocation Failure) [PSYoungGen: 5688K->488K(6144K)] 9848K->9776K(19968K), 0.0107310 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]  [Full GC (Ergonomics) [PSYoungGen: 488K->0K(6144K)] [ParOldGen: 9288K->9720K(13824K)] 9776K->9720K(19968K), [Metaspace: 2514K->2514K(1056768K)], 0.0178757 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]  [Full GC (Ergonomics) [PSYoungGen: 5275K->1024K(6144K)] [ParOldGen: 9720K->13816K(13824K)] 14995K->14840K(19968K), [Metaspace: 2517K->2517K(1056768K)], 0.0296804 secs] [Times: user=0.01 sys=0.00, real=0.03 secs]  [Full GC (Ergonomics) [PSYoungGen: 5147K->5120K(6144K)] [ParOldGen: 13816K->13816K(13824K)] 18964K->18936K(19968K), [Metaspace: 2517K->2517K(1056768K)], 0.0062843 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [Full GC (Allocation Failure) [PSYoungGen: 5120K->5120K(6144K)] [ParOldGen: 13816K->13793K(13824K)] 18936K->18914K(19968K), [Metaspace: 2517K->2517K(1056768K)], 0.0221666 secs] [Times: user=0.05 sys=0.00, real=0.02 secs]  java.lang.OutOfMemoryError: Java heap space  Dumping heap to d:/a.dump ...  Heap dump file created [20033924 bytes in 0.061 secs]  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  at com.hc.stu.OnStackTest.main(OnStackTest.java:18)  Heap  PSYoungGen total 6144K, used 5251K [0x00000000ff980000, 0x0000000100000000, 0x0000000100000000)  eden space 5632K, 93% used [0x00000000ff980000,0x00000000ffea0dc8,0x00000000fff00000)  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)  to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)  ParOldGen total 13824K, used 13793K [0x00000000fec00000, 0x00000000ff980000, 0x00000000ff980000)  object space 13824K, 99% used [0x00000000fec00000,0x00000000ff9787c0,0x00000000ff980000)  Metaspace used 2549K, capacity 4486K, committed 4864K, reserved 1056768K  class space used 273K, capacity 386K, committed 512K, reserved 1048576K |

官方推荐；

新生代占堆的3/8，幸存代占新生代的1/10

常用参数组合

-Xmx20m -Xms20m -Xmn7m -XX:NewRatio=1 -XX:SurvivorRatio=3 -XX:+PrintGCDetails

## 永久区分配参数

Permanent,持久带，

-XX:PermSize 默认64m

-XX:MaxpermSize=128m

## eg:永久区OOM1

for(int i=0; i<1000000; i++){

CglibBean bwan = new CglibBean(“xxx”+i, new HashMap() );

}

## eg:永久区OOM2

<http://www.cnblogs.com/zhuxing/articles/1247621.html>

Java内存溢出之PermGen OOM深入分析

|  |
| --- |
| try {  //准备url  URL url = new File("D:/classes").toURL();  URL[] urls = {url};  //获取有关类型加载的JMX接口  ClassLoadingMXBean loadingBean = ManagementFactory.getClassLoadingMXBean();  //用于缓存类加载器  List<ClassLoader> classLoaders = new ArrayList<ClassLoader>();  while (true) {  //加载类型并缓存类加载器实例  ClassLoader classLoader = new URLClassLoader(urls);  classLoaders.add(classLoader);  classLoader.loadClass("ZhuXing");  //显示数量信息（共加载类型数目，当前有效类型数目，被卸载类型数目）  System.out.println("total: " + loadingBean.getTotalLoadedClassCount());  System.out.println("active: " + loadingBean.getLoadedClassCount());  System.out.println("unloaded: " + loadingBean.getUnloadedClassCount());  }  } catch (Exception e) {  e.printStackTrace();  } |

## 栈分配参数

-Xss128k

通常只有几百K

## 标准参数(-)

-，所有的JVM实现都必须实现这些参数的功能，而且向后兼容

### -client

以client模式启动jvm，这种方式启动速度快，但运行时性能和内存管理效率不高，适合客户端程序或者开发调试。

### -server

以server模式启动jvm，与client情况恰好相反。适合生产环境。

64位的jvm自动以server模式启动。

### -classpath -cp

告诉jvm类搜索路径。如果指定了-classpath，则jvm就忽略CLASSPATH中指定的路径。各路径之间以分号隔开。如果-classpath和CLASSPATH都没有指定，则jvm从当前路径寻找class。

Jvm搜索class的方式和顺序为：Bootstrap(jvm自带的)->Extension（JRE\_HOME/lib/ext中的）->User（应用程序自己的）。如下语句会输出上述三种方式的路径信息：

System.getProperty("sun.boot.class.path");

System.getProperty("java.ext.dirs");

System.getProperty("java.class.path");

### -D

-Dproperty=value

设置系统属性名/值对。如果value中有空格，则需要使用双引号。

### -jar

指定以jar包的形式启动应用程序

### -javaagent

-javaagent:<jarpath>[=<options>]

指定jvm启动时装入java语言设备代理。关于设备代理是jdk1.5新推出的概念。进一步的信息可以参考java.lang.instrument.Instrumentation接口

### -verbose

-verbose[:class|gc|jni]

分别输出：

Jvm载入类的信息。诊断ClassNotFound错误时有用

Jvm每次gc的信息。

Native方法调用的情况，用于诊断jni调用。

## 非标准参数(-X)

-X，默认JVM实现此功能，但是不保证所有JVM都实行，而且不保证向后兼容

### -Xbootclasspath

-Xbootclasspath:bootclasspath

让jvm从指定的路径中加载bootclass，用来替换jdk的rt.jar。一般不会用到。

-Xbootclasspath/a:path

被指定的文件追加到默认的bootstrap路径中。

-Xbootclasspath/p:path

让jvm优先于默认的bootstrap去加载path中指定的class

### -Xincgc

-Xincgc 开启增量gc，一般也不会用到。

### -Xloggc

-Xloggc:<file>

输出gc日志，在没有专业的监控工具时，这个显得很有用。

### -Xmx

-Xmx16m

指定jvm堆的最大值。默认是物理内存的1/4

|  |
| --- |
| Eg:T1.java  System.out.println("maxMemory="+Runtime.getRuntime().maxMemory()/1024.0/1024.0+"M");  System.out.println("freeMemory="+Runtime.getRuntime().freeMemory()/1024.0/1024.0+"M");  System.out.println("totalMemory="+Runtime.getRuntime().totalMemory()/1024.0/1024.0+"M");  D:\std\03>java -Xmx512m -Xms512m T1  maxMemory=512.0M  freeMemory=402.0382766723633M  totalMemory=512.0M  D:\std\03>java -Xmx512m T1  maxMemory=455.0M  freeMemory=72.89435577392578M  totalMemory=153.5M  hello word! |

### -Xms

指定jvm堆的初始大小。默认是物理内存的1/64

### -Xmn

设置新生代大小

-Xmn1m

表示Eden+from+to = 1m

### -Xss

设置单个线程栈的大小。

### -Xprof

跟踪正在运行的程序，并将跟踪数据输出到标准输出，适合开发调试。

## 非稳定参数(-XX)

-XX，此类参数各个JVM的实现会有不同，而且随时可能消失

### -XX:+ScavengeBeforeFullGC

新生代GC优先于Full GC执行

### -XX:+UseGCOverheadLimit

在抛出OOM之前，jvm消耗在gc上的时间比例。

### -XX:-UseConcMarkSweepGC

采用并发标记交换算法进行GC。并发是指过个线程同时gc，适合多核处理器的情况，并发一定满足并行，反之不成立。

### -XX:-UseParallelGC

启用并行GC。并行是指有独立的线程进行gc，gc时不会打断应用线程。

### -XX:-UseSerialGC

启用串行GC。新版本jdk不会这么干了。

### -XX:NewRatio

### -XX:MaxNewSize=size

Java堆中新生成对象占用堆的大小

### -XX:MaxPermSize=size

Java堆中年老代占用堆的最大值

### -XX:HeadDumpPath=./java\_pid<pid>.hprof

指定导出堆信息时的路径或者文件名

### -XX:+HeadDumpOnOutOfMemoryError

当首次出现OOM时导出此时堆中相关信息。

### -XX:+PrintGC

每次GC时打印相关信息

|  |
| --- |
| D:\std\03>java -XX:+PrintGC T1  [GC 16384K->520K(61952K), 0.0130297 secs]  [GC 16904K->504K(78336K), 0.0047305 secs]  [GC 33272K->504K(78336K), 0.0066194 secs]  [GC 33272K->560K(111104K), 0.0024796 secs]  [GC 66096K->488K(111104K), 0.0151663 secs]  [GC 66024K->504K(175104K), 0.0060366 secs] |

### -XX:+PrintGCDetails

每次GC时打印详细信息

|  |
| --- |
| D:\std\03>java -XX:+PrintGCDetails T1  [GC [PSYoungGen: 16378K->520K(18944K)] 16378K->520K(61952K), 0.0175810 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]  [GC [PSYoungGen: 16898K->504K(35328K)] 16898K->504K(78336K), 0.0056236 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [GC [PSYoungGen: 33272K->504K(35328K)] 33272K->504K(78336K), 0.0045910 secs] [Times: user=0.00 sys=0.02, real=0.00 secs]  [GC [PSYoungGen: 33272K->576K(68096K)] 33272K->576K(111104K), 0.0032187 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC [PSYoungGen: 66112K->504K(68096K)] 66112K->504K(111104K), 0.0144939 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  [GC [PSYoungGen: 66040K->504K(132096K)] 66040K->504K(175104K), 0.0062213 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]  [GC [PSYoungGen: 131576K->32K(132096K)] 131576K->500K(175104K), 0.0044729 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [GC [PSYoungGen: 131104K->32K(263168K)] 131572K->500K(306176K), 0.0023564 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC [PSYoungGen: 262176K->0K(263168K)] 262644K->476K(306176K), 0.0032350 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC [PSYoungGen: 262144K->0K(343040K)] 262620K->476K(386048K), 0.0017479 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC [PSYoungGen: 342016K->0K(343040K)] 342492K->476K(386048K), 0.0016565 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  Heap  PSYoungGen total 343040K, used 300992K [0x00000000eb000000, 0x0000000100000000, 0x0000000100000000)  eden space 342016K, 92% used [0x00000000eb000000,0x00000000fe34c670,0x00000000ffe00000)  from space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)  to space 1024K, 0% used [0x00000000fff00000,0x00000000fff00000,0x0000000100000000)  ParOldGen total 43008K, used 476K [0x00000000c1000000, 0x00000000c3a00000, 0x00000000eb000000)  object space 43008K, 1% used [0x00000000c1000000,0x00000000c1077070,0x00000000c3a00000)  PSPermGen total 21504K, used 2533K [0x00000000bbe00000, 0x00000000bd300000, 0x00000000c1000000)  object space 21504K, 11% used [0x00000000bbe00000,0x00000000bc079540,0x00000000bd300000)  [GC [PSYoungGen: 342016K->32K(343040K)] 342492K->508K(386048K), 0.0018837 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] |

### -XX:+PrintGCTimeStamps

打印GC的时间戳,这个选项记录的是jvm启动时间为起点的相对时间，可读性较差，不利于定位问题

### -XX:+PrintGCDateStamps

打印GC的时间戳,记录的是系统时间

### -XX:+TraceClassLoading

跟踪类的加载信息

|  |
| --- |
| D:\std\03>java -XX:+TraceClassLoading T1  [Opened D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.lang.Object from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.io.Serializable from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.lang.Comparable from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.lang.String from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.lang.Class from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  。。。 。。。  [Loaded java.security.UnresolvedPermission from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.security.BasicPermissionCollection from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded T1 from file:/D:/std/03/]  [Loaded java.lang.Void from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  hello word!  [Loaded java.lang.Shutdown from D:\java\jdk1.7.0\_79\jre\lib\rt.jar]  [Loaded java.lang.Shutdown$Lock from D:\java\jdk1.7.0\_79\jre\lib\rt.jar] |

### -XX:+TraceClassLoadingPreorder

跟踪被引用到的所有类的加载信息

### -XX:+TraceClassUnloading

跟踪类的卸载信息

### -XX:+TraceClassResolution

跟踪常量池

## -XX:+Prin**tClassHistogram**

按下Ctrl+Break ,打印类的信息

参考

Java 6 JVM参数选项大全（中文版）

http://kenwublog.com/docs/java6-jvm-options-chinese-edition.htm

+http://lavasoft.blog.51cto.com/62575/25492

http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

-XX:+DisableExplicitGC标志自动将System.gc()调用转换成一个空操作

-XX:PermSize 及 -XX:MaxPermSize

-Xms，-Xmx

-XX:MinHeapFreeRatio=40

专家指导如何进行JVM参数配置

http://developer.51cto.com/art/201009/227903.htm

http://developer.51cto.com/art/201009/227750.htm

http://developer.51cto.com/art/200907/134761.htm

设置JVM参数之堆设置

-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

设置JVM参数之收集器设置

-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

设置JVM参数之垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

设置JVM参数之并行收集器设置

-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)

设置JVM参数之并发收集器设置

-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。

# GC日志分析

参考：<http://swcdxd.iteye.com/blog/1859858>

JVM的GC日志的主要参数包括如下几个：

-XX:+PrintGC 输出GC日志

-XX:+PrintGCDetails 输出GC的详细日志

-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:../logs/gc.log 日志文件的输出路径

eg:

D:\std\03>java -Xmx512m -Xms512m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log T1

|  |
| --- |
| D:\std\03>java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xmx512m -Xms512m T1  0.331: [GC [PSYoungGen: 132096K->552K(153600K)] 132096K->552K(503296K), 0.0246810 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]  0.436: [GC [PSYoungGen: 132648K->504K(153600K)] 132648K->504K(503296K), 0.0051794 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  … …  2.881: [GC [PSYoungGen: 173056K->0K(174080K)] 173524K->468K(523776K), 0.0018212 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  2.990: [GC [PSYoungGen: 173056K->0K(174592K)] 173524K->468K(524288K), 0.0231249 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]  hello word!  Heap 开始边界 当前边界 最高边界  PSYoungGen total 174592K, used 24397K [0x00000000f5500000, 0x0000000100000000, 0x0000000100000000)  eden space 174080K, 14% used [0x00000000f5500000,0x00000000f6cd3688,0x00000000fff00000)  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)  to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)  ParOldGen total 349696K, used 468K [0x00000000dff80000, 0x00000000f5500000, 0x00000000f5500000)  object space 349696K, 0% used [0x00000000dff80000,0x00000000dfff5070,0x00000000f5500000)  PSPermGen total 21504K, used 2531K [0x00000000dad80000, 0x00000000dc280000, 0x00000000dff80000)  object space 21504K, 11% used [0x00000000dad80000,0x00000000daff8e60,0x00000000dc280000) |

**解释:**

2.990时间戳:

[GC [PSYoungGen: 173056K年轻代垃圾回收前的大小->0K年轻代垃圾回收以后的大小(174592K) 年轻代的总大小]

173524K堆区垃圾回收前的大小->468K堆区垃圾回收后的大小(524288K) 堆区总大小, 0.0231249 secs回收时间]

[Times: user=0.03用户耗时 sys=0.00系统耗时, real=0.02 secs实际耗时]

**进一步说明**

-Xmx512m 堆大小512m(524288K)

-Xmx512m(524288K) = PSYoungGen 174592K + ParOldGen 349696K

PSYoungGen 174592K = eden space 174080K + from space 512K

**PSPermGen不包含在堆里面**

# GC

## 算法

引用计数法

Java不适用

标记清除

两个阶段：标记阶段，清除阶段

标记压缩

比标记清除优

复制算法

新生代使用的算法

两块空间，每次将所有存活对象复制到另一个空间；

比标记压缩相对比较高效，适合每次回收大量对象；

对空间有一定的浪费，浪费一般的空间

## 可达/不可达/复活

可达/可触及

从根节点可触及(现阶段)

可复活

一旦所有引用被释放，就是可复活状态；

在调用finalize()方法之前都是可复活的；

不可触及

调用finalize()方法后，可能进入不可触及状态，

该状态的对象不可复活，

可以被回收；

## finalize

只会调用一次，

在被调用时可以复活对象，如set static obj = this;

当obj=null时后，再次GC时，是不调用的，不会复活对象，该对象被回收；

应该避免使用finalize()

## 根

栈中引用的对象；

方法区中静态成员或常量引用的对象；

JNI方法栈中引用的对象；

## Stop-The-World

Java中全局暂停现象；

多半由于GC引起:Dump线程，死锁检查；

所有Java代码停止，native代码可以执行，但不能和JVM交互；

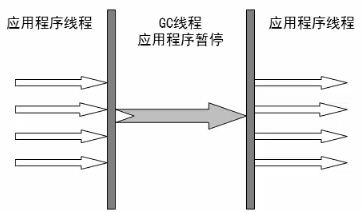
## 串行回收器SerialGC

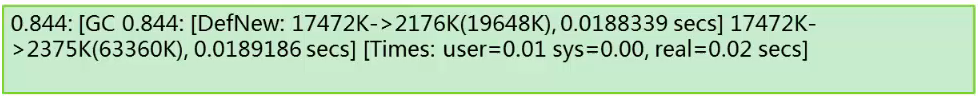
最古老最稳定，效率高；可能产生较长停顿

-XX:+UseSerialGC

新生代，老年代使用串行回收，

新生代复制算法，老年代标记-压缩算法；





## 并行回收器ParNewGC

-XX:+UseParNewGC

-XX:ParallelGCThreads 限制线程数量

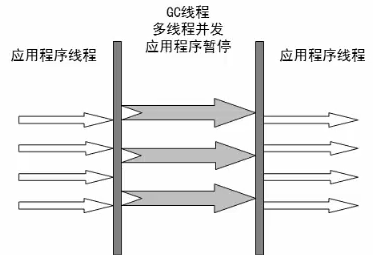
串行收集器在新生代的并行版本；

**新生代**并行回收， 老年代串行回收，

多线程，需要多核支持。多线程不一定快，单核不建议使用；

Stop-the-word

GC日志有ParNew





## 并行回收器ParallelGC

类似ParNew

串行收集器在新生代和老年代的并行版本；

更加关注吞吐量；

应用程序暂停；

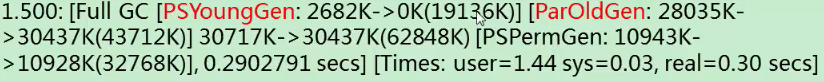
-XX:+UseParallelGC 新生代并行+老年代串行

-XX:+UseParallelOldGC 新生代并行+老年代并行

-XX:MaxGCPauseMills 最大停顿时间，毫秒，GC只是尽力保证不超过该时间

-XX:GCTimeRatio 0-100，GC占总时间的比。默认99，占用1/100CPU时间；

停顿时间和吞吐量不可能同时调优



## 并发回收器CMS

-XX:+UseConcMarkSweepGC

Concurrent Mark Sweep;

**并发标记清除**，前面都是标记压缩；

只在**老年代**使用，新生代ParNew

主要工作与用户线程一起执行，关键步骤需要独占CPU

与用户线程一起运行，减少停顿时间，吞吐量下降

GC 时会影响系统性能

清理不彻底

因为与用户线程一起运行，不能在空间快满时才清理

-XX:CMSInitiaingOccupancyFraction 设置触发GC的阈值

如果不幸内存空间不够，就会引起concurrent mode failure,使用串行GC作为后备

CMS后会产生碎片，需要整理

-XX:+UseCMSCompactAtFullCollection

FullGCh后进行一次整理，整理是独占的

-XX:+CMSFullGCsBeforeCompaction

设置进行几次FullGC后，进行一次碎片整理

-XX:ParallelCMSThreads

设定CMS的线程数量，一般等于CPU数量

-XX:+CMSClassUnloadingEnabled

允许对类元数据进行回收

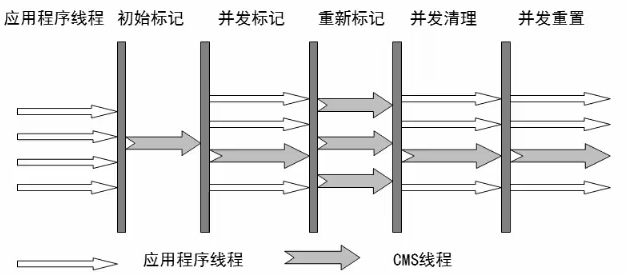
当永久区OOM时，要么增大Perm空间，要么允许对该空间GC

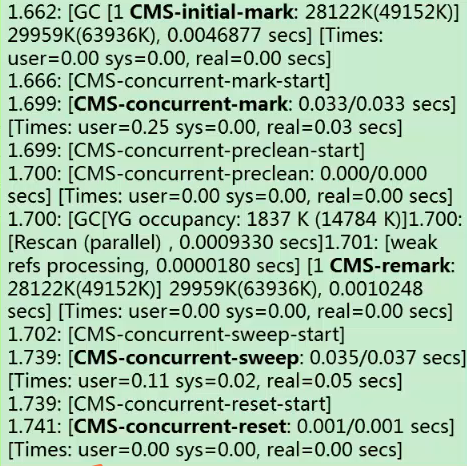
-XX:CMSInitiaingPermOccupancyFraction

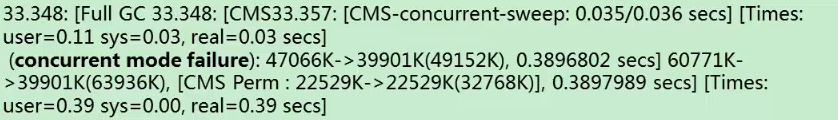
当永久区占用率达到这一百分比时启动CMS

-XX:UseCMSInitiaingOccupancyOnly

只在到达阈值时，才进行CMS回收







CMS过程比较复杂，着重实现了标记的过程：

1 初始标记, 停顿

根可以直接关联的对象，速度快，停顿

2 并发标记

和用户线程一起

标记全部对象

3 重新标记, 停顿

由于并发标记时，用户线程也在运行，这里做些修正

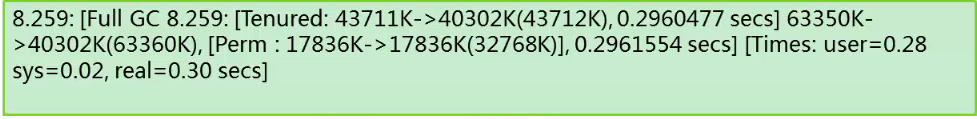
4 并发清除

基于标记结果，直接清理对象;

5 并发重置

清理工作，为下一次GC准备

## FullGC



## 减轻GC压力

# JVM调优示例

## 示例 tomcat设置

|  |
| --- |
| JAVA\_OPTS="-server -Xms2048M -Xmx2048M -Xmn1024M -XX:SurvivorRatio=4 -Xss256k -XX:PermSize=512m -XX:MaxPermSize=512m -XX:-DisableExplicitGC -verbosegc -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:-CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:ParallelCMSThreads=8 -XX:ParallelGCThreads=8 -XX:MaxTenuringThreshold=5 -XX:-UseAdaptiveSizePolicy -XX:TargetSurvivorRatio=90 -XX:+ScavengeBeforeFullGC -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/tomcat/logs/oom-error.log -Xloggc:/usr/local/tomcat/logs/gc.log" |

# 参考

<http://blog.csdn.net/column/details/javavirtualmachine.html>

Java虚拟机学习系列