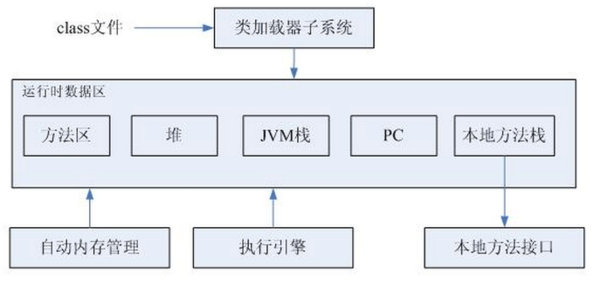
**JVM架构体系**

## 一、JVM概述

**1 JVM组成结构**



## 二、 JVM 类加载机制

**1 Bootstrap ClassLoader**是在JVM开始运行的时候加载java的核心类，是用C++编写的，它用来加载核心类库，在JVM源代码中这样写道  
static const char classpathFormat[] =   
"%/lib/rt.jar:"   
"%/lib/i18n.jar:"   
"%/lib/sunrsasign.jar:"   
"%/lib/jsse.jar:"   
"%/lib/jce.jar:"   
"%/lib/charsets.jar:"   
"%/classes";   
Extension ClassLoader是用来加载扩展类，即/lib/ext中的类。   
AppClassLoader用来加载Classpath的类，是和我们关系最密切的类。   
自定义加载：2种方式，1是URLClassLoader用来加载网络上远程的类，2是自己定义类继承 ClassLoader 。

**2 预加载和按需加载**

当 java.exe 虚拟机开始运行以后，它会找到安装在机器上的 JRE 环境，然后把控制权交给 JRE ， JRE 的类加载器会将 lib 目录下的 rt.jar 基础类别文件库加载进内存，这些文件是 Java 程序执行所必须的，所以系统在开始就将这些文件加载，避免以后的多次 IO 操作，从而提高程序执行效率。 相对于预先加载，我们在程序中需要使用自己定义的类的时候就要使用依需求加载方法（ load-on-demand ），就是在 Java 程序需要用到的时候再加载，以减少内存的消耗。

**3 ClassLoader工作原理：**

1) 线程需要用到某个类，于是contextClassLoader被请求来载入该类

2) contextClassLoader请求它的父ClassLoader来完成该载入请求

3) 如果父ClassLoader无法载入类，则contextClassLoader试图自己来载入

**4 父类委托机制 进行加载**

**保证安全，父类委托机制的优点是能够提高软件系统的安全性，假设我自己定义一个类加载器，然后随便伪造一个类，这个类不符合jvm规范，里面有不安全的代码，如果不适用父类委托机制，那么这个类就会被直接加载到内存里面了。 如果使用父类委托，那么就会被父加载器加载，它会按照jvm规范来加载，不符合规范就不会加载。**

**5 隐式加载**

不通过显示调用class loader来加载需要的类，而是通过JVM因需自动加载到内存当中的方式。比如加载一个类的时候会隐式加载它的父类。

**6 显示加载**

通过调用classLoader来加载类的方式，比如

this.getclass().getClassLoader().loadClass();

Class.forName("className");

**7 常见的错误**

ClassNotFoundException:这个异常发生在显示加载类的时候，没有找到对应类的字节码，显示加载的方式如下：通过类class中的forName()方法；通过ClassLoader中的loadClass()方法

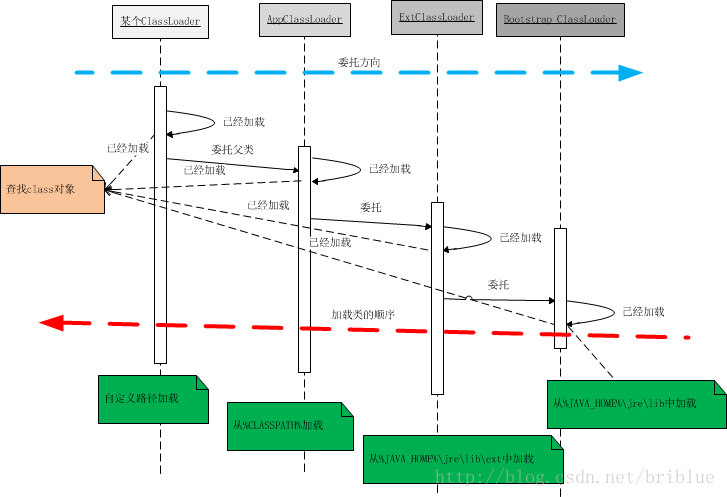
通过ClassLoader中的findSystemClass()方法

NoClassDefFoundError:隐式加载类时出现,涉及隐式加载的情景:

使用new关键字；属性引用加载某个类；继承了某个接口或类；以及方法的某个参数中引用了某个类

UnsatisfiedLinkError是一个在解析native标识的方法时出现的错误，是库文件缺失造成的，无法链接到本地的代码实现库(native实现)

**8 JVM初始化sun.misc.Launcher并创建Extension ClassLoader和AppClassLoader实例。**并将ExtClassLoader设置为AppClassLoader的父加载器。Bootstrap没有父加载器，但是它却可以作用一个ClassLoader的父加载器。比如ExtClassLoader。这也可以解释之前通过ExtClassLoader的getParent方法获取为Null的现象。



**9 ContextClassLoader**

通常情况下，JVM中的所有类加载器被组织成一个层次结构，使得每一个类加载器（除了原始类加载器）都有一个父加载器。当被要求加载一个类时，每一个类加载器都将先委托父加载器来加载，只有父加载器都不能成功加载时当前类加载器才会加载，双亲委派。

在这种情况下，如下的委派链中：

ClassLoader A -> System class loader -> Extension class loader -> Bootstrap class loader，

委派链左边的ClassLoader就可以很自然的使用右边的ClassLoader所加载的类。

但如果情况要反过来，是右边的ClassLoader所加载的代码需要反过来去找委派链靠左边的ClassLoader去加载东西怎么办呢？没办法反过来从右边找左边的~

以JNDI举例：它的核心内容（从J2SE1.3开始）在rt.jar中的引导类中实现了，但是这些JNDI核心类可能加载由独立厂商实现和部署在应用程序的classpath中的JNDI提供者。这个例子中的原始类加载器，即加载rt.jar的加载器去加载一个在它的子类加载器中可见的类。此时通常的J2SE委托机制不能工作，解决办法是让JNDI核心类使用线程上下文加载器，从而有效建立一条与类加载器层次结构相反方向的“通道”达到正确的委托。

## 调优工具

**1 基础知识**

**一台物理机或者虚拟机上面安装一个JVM，也可以安装多个 。**

**一台物理机或者虚拟机上面安装一个JVM，也就是JDK，上面可疑运行tomcat,eclipse等java进程，分别可以配置JVM的内存参数，但是总的内存不能大于物理机的内存值。**

**一个eclipse的main是一个进程，在eclipse.ini 中配置**

**一个tomcat是一个进程，里面有很多线程，wins在 catalina.bat，linux在catalina.sh中设置**

**根据进程号可以找到里面的参数设置**

**2 企业级应用开发中经常会遇到以下问题，可以使用工具对JVM进行监管，查找问题所在。**  
　　内存不足OutOfMemory（大对象没有gc等），内存泄露；  
　　线程死锁，线程数过多；  
　　锁争用（Lock Contention），资源未及时释放（数据库）；  
　　Java进程CPU消耗过高.

**3 Jps（JVM Process Status Tools） 虚拟机进程状态工具**

可以列举正在运行的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）

jps [option] [hostid默认为本机]

option选项：

-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

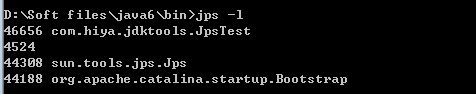
-v输出JVM的启动参数

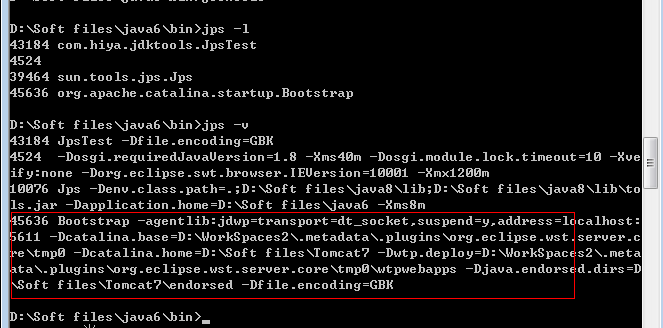
jps -q/

jps -m/

jps -l/

jps -v





**4 jstack看某个Java进程内的线程堆栈信息**

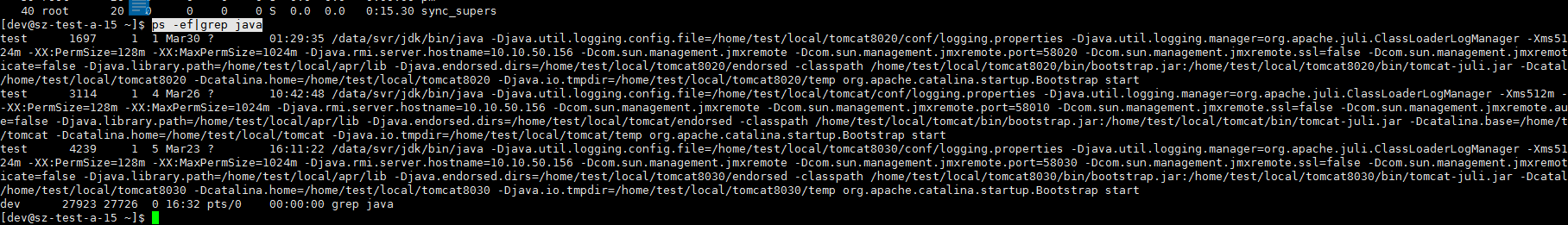
jstack用于生成java虚拟机当前时刻的线程快照，主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。

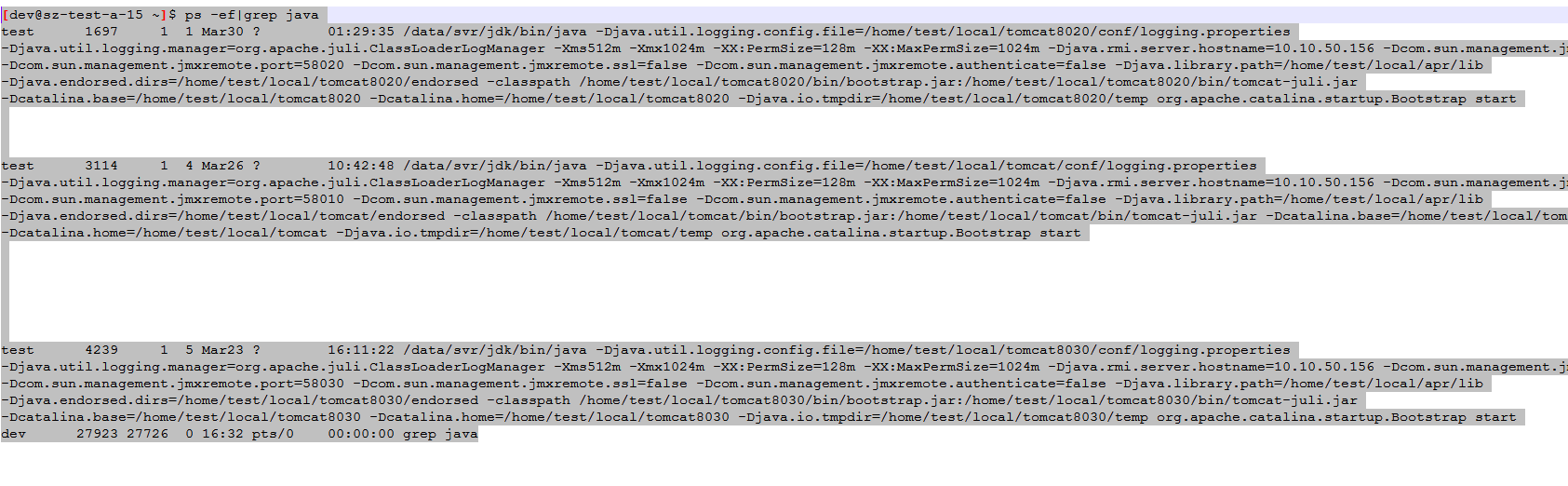
语法： jstack [ option ] pid

-F 当 jstack [-l] pid 没有响应的时候强制打印栈信息

-l 长列表. 打印关于锁的附加信息,例如属于java.util.concurrent的ownable synchronizers列表.

第一步先找出Java进程ID，我部署在服务器上的Java应用名称为mrf-center：





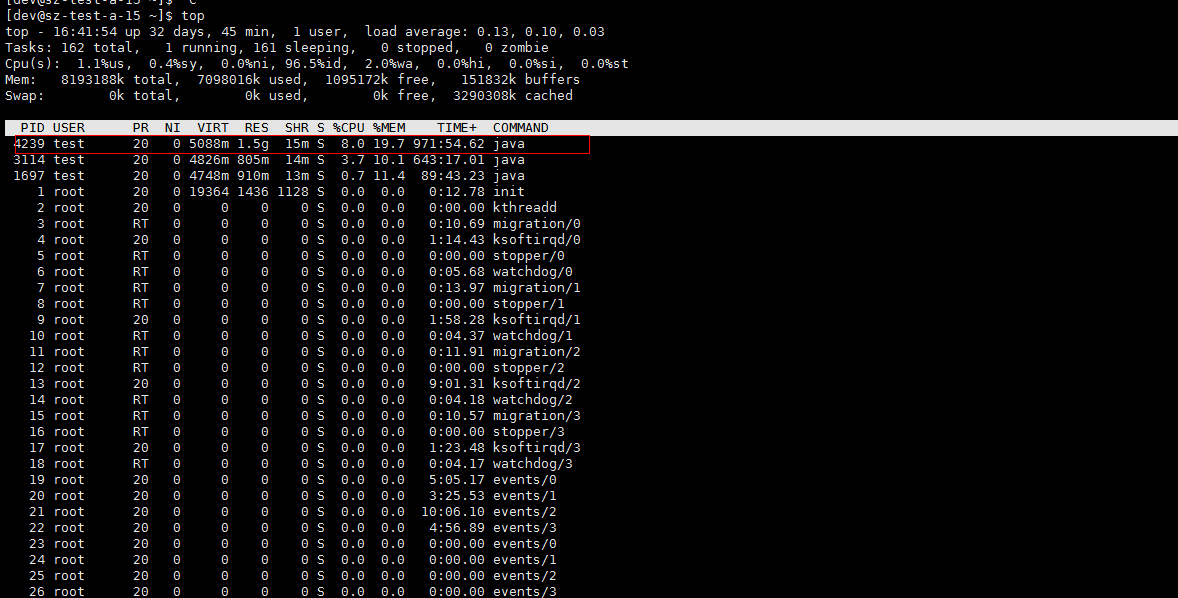
得到进程ID为21711，第二步找出该进程内最耗费CPU的线程，可以使用ps -Lfp pid或者ps -mp pid -o THREAD, tid, time或者top -Hp pid。

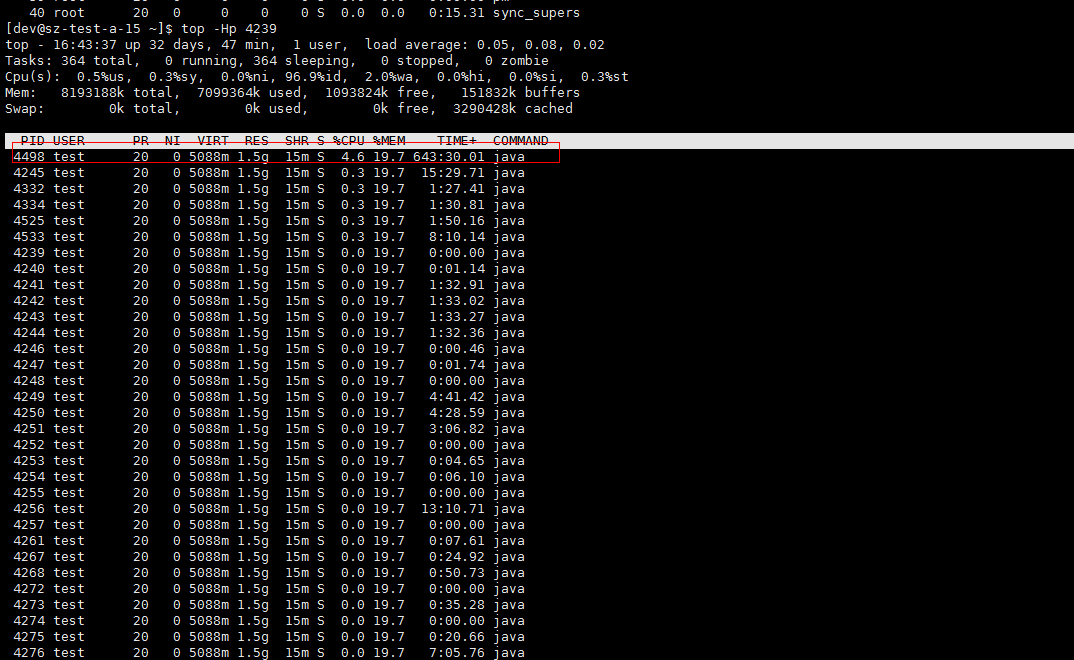
TIME列就是各个Java线程耗费的CPU时间，CPU时间最长的是线程ID为21742的线程，用得到21742的十六进制值为54ee。

 jstack 21711 | grep 54ee

"PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000 nid=0x54ee in Object.wait(

可以看到CPU消耗在PollIntervalRetrySchedulerThread这个类的Object.wait()，





4498的十六进制是1192

jstack 7957 > test.txt



**5 jstat 查看classloader，compiler，gc相关信息，实时监控资源和性能 。**

jstat工具特别强大，可以用来监视VM内存内的各种堆和非堆的大小及其内存使用量。

语法结构：jstat -<option> [-t] [-h<lines>] <pid> [<interval> [<count>]]

Options — 通常使用 -gcutil 查看gc情况

interval – 间隔时间，单位为秒或者毫秒

count — 打印次数，如果缺省则打印无数次

结果说明：

S0 — Heap上的 Survivor space 0 区已使用空间的百分比

S1 — Heap上的 Survivor space 1 区已使用空间的百分比

E — Heap上的 Eden space 区已使用空间的百分比

O — Heap上的 Old space 区已使用空间的百分比

P — Perm space 区已使用空间的百分比

YGC — 从应用程序启动到采样时发生 Young GC 的次数

YGCT— 从应用程序启动到采样时 Young GC 所用的时间(单位秒)

FGC — 从应用程序启动到采样时发生 Full GC 的次数

FGCT— 从应用程序启动到采样时 Full GC 所用的时间(单位秒)

GCT — 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

jstat –class <pid> : 显示加载class的数量，及所占空间等信息。

jstat -compiler <pid>: 显示VM实时编译的数量等信息。

jstat -gc <pid>: 显示gc的信息，查看gc的次数，及时间。

jstat -gccapacity <pid>: 显示VM内存中三代（young,old,perm）对象的使用和占用大小

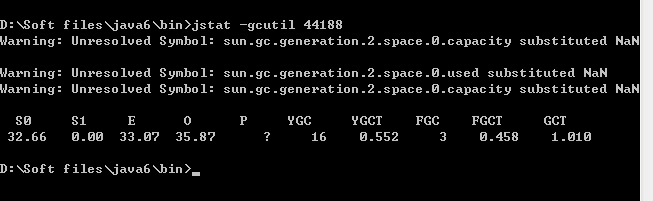
jstat -gcutil <pid>: 统计gc信息

jstat -gcnew / gcnewcapacity<pid>: 年轻代对象的信息(及其占用量)。

jstat -gcold / gcoldcapacity <pid> ：old代对象的信息(及其占用量)。

jstat -gcpermcapacity <pid>: perm对象的信息及其占用量。

jstat -printcompilation <pid>: 当前VM执行的信息。





**6 jconsole可以监控Java应用程序(如jar应用、tomcat等)**

但被监视的应用程序必须和jconsole是用同一个用户运行的。jvisualvm的使用和jconsole类似

本地监控： jconsole pid

远程监控： jconsole [ hostname:portNum ]

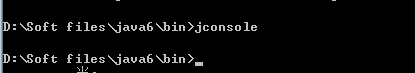
使用远程监控需要配置jmx代理信息，修改Tomcat的bin目录下的catalina.bat。

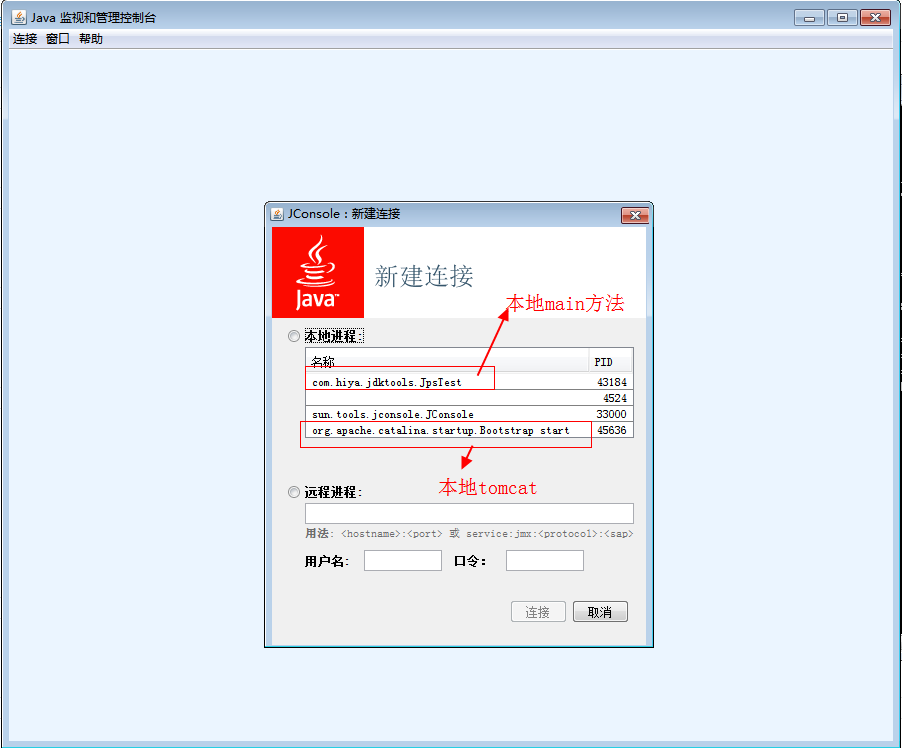
set JAVA\_OPTS= %JAVA\_OPTS% -Djava.rmi.server.hostname=HostIP

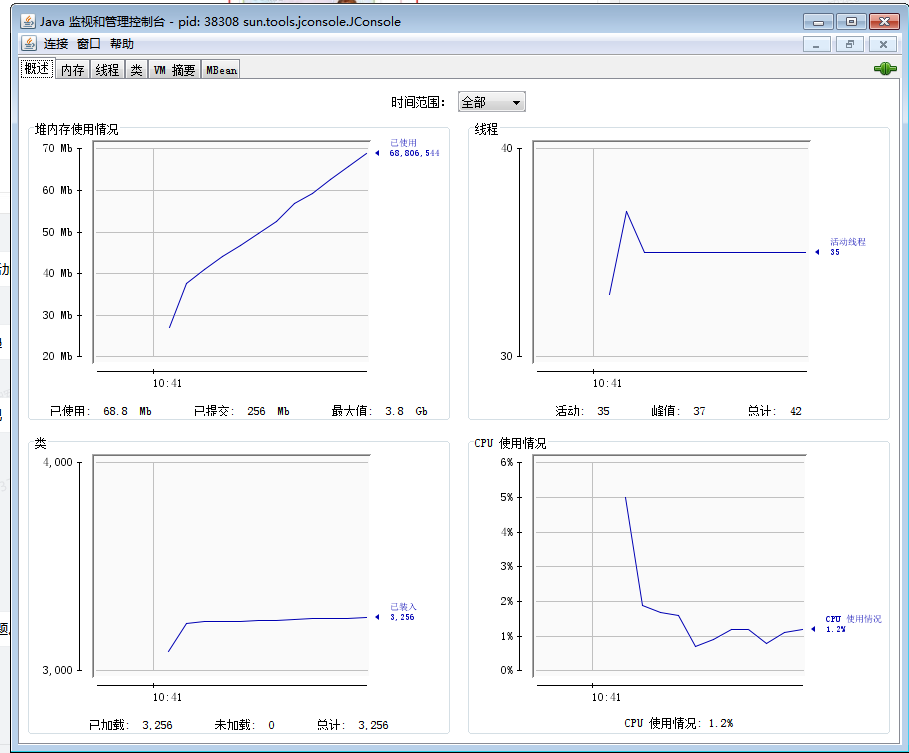
set JAVA\_OPTS= %JAVA\_OPTS% -Dcom.sun.management.jmxremote.port=8888

set JAVA\_OPTS= %JAVA\_OPTS% -Dcom.sun.management.jmxremote.ssl=false

set JAVA\_OPTS= %JAVA\_OPTS% -Dcom.sun.management.jmxremote.authenticate=false





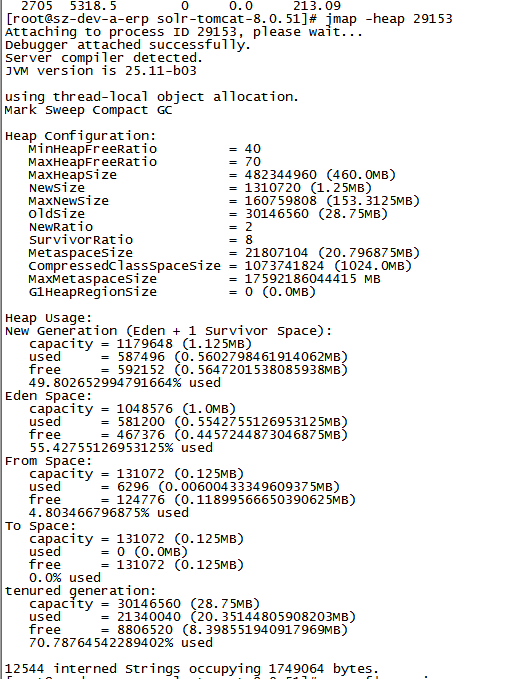


**7 Jmap 打印java进程的堆内存信息**

jmap -heap pid：查看heap的概要信息，GC使用的算法、heap的配置使用情况.

jmap -histo[:live] pid：查看堆内存中的每个类的类名、实例数量、内存占用大小

jmap -dump:live, format=b, file=fileName pid：将内存使用情况导出到文件中，再用jhat、MAT、VisualVM分析查看，以便查找内存溢出原因



**8 jhat可以对JVM中导出的文件进行分析**

使用命令 jhat fileName 即可以在浏览器中输入http://localhost:7000查看内存信息。如果Dump文件太大需要加上-J-Xmx512m指定最大堆内存，如 jhat -J-Xmx512m [-port 9998] tmp.bin分析内存还可以使用Eclipse的**Memory Analyze**r，插件地址http://download.eclipse.org/releases/juno，找到General Purpose Tools底下的Memory Analyzer并安装

## 四、执行引擎

**1 执行引擎**

JVM 加载 class 文件后，已经将 class 文件中的常量信息、类信息、方法代码等放入方法区中了。JVM 通过执行引擎来完成字节码的执行，在执行过程中 JVM 采用的是自己的一套指令系统，每个线程在创建后，都会产生一个程序计数器（pc）和栈（Stack），其中程序计数器中存放了下一条将要执行的指令，Stack 中存放 Stack Frame，表示的为当前正在执行的方法， 每个方法的执行都会产生 Stack Frame，Stack Frame 中存放了传递给方法的参数、方法内的局部变量以及操作数栈，操作数栈用于存放指令运算的中间结果，指令负责从操作数栈中弹出参与运算的操作数，指令执行完毕后再将计算结果压回到操作数栈，当方法执行完毕后则从 Stack 中弹出，继续其他方法的执行。

invokestatic：调用类的 static 方法

public class B{

public static void main(String[] args){ A.execute(“Hello World”);

}

}

编译后 B 中的代码转变为了如下字节码（通过调用 javap –c B 查看）：

public static void main(java.lang.String[]); Code:

0: ldc #16; //String Hello World

2: invokestatic #18; //Method sample/A.execute:(Ljava/lang/String;)V

5: return

当main 线程启动后，JVM 为此线程创建一个PC 和Stack，执行 ldc #16;//String Hello World，并同时将 invokestatic 放入 PC，接着将 Hello World 的引用压入 Stack 的参数变量中，

继续执行 invokestatic 指令，该指令后跟随的//Method 部分的内容称为<method-spec>内容，此内容的规范为：包名/类名.方法名:(参数类型)返回值类型，invokestatic 指令根据此

method-spec 找到此 Class 的方法，根据 method-spec 中的参数信息找到参数的个数，从 stack frame 中弹出这些参数，然后创建一个新的 stack frame，将参数信息压入，然后重

复上述过程，完成整个方法的执行。

invokevirtual：调用对象实例的方法 A a=new A()

invokeinterface：调用接口的方法 IA a=new A();

invokespecial：初始化对象，以及调用对象实例中的私有方法时

**2 三种方式执行：**

**（1）解释~解释执行方式，**也就是每次都由 JVM 来解释字节码，进行执行的方式，无疑这种方式会使得性能较低，毕竟每次执行都需要进行解释，然后调用机器指令完成执行。

**（2）即时编译~即时编译方式**，也就是每次执行字节码时，JVM 都将其首先转为机器码，然后进行执行， 固然这种方式性能较高，但每次转化为机器码，也使得系统执行会受到不小的影响。

**（3）自适应编译~自适应编译方式**，简称 Hotspot 方式，它是 Sun JDK 保证 Java 程序高性能执行的基础，方式的特点为在运行期根据代码的执行频率来决定是否要编译为机器码，如达到了执行次数的条件，那么就编译成机器码，这也就使得对于那些经常被执行的代码而言，执 行的性能是非常高的，并且做到尽可能少的影响整体应用性能，这个执行次数的条件可通过

-XX:CompileThreshold=10000 来设置，默认情况下为 10000 次，也可通过-XX:+PrintCompilation 参数来查看被编译成机器码的方法。

## 五、内存机制

**1 Runtime data area 的整体架构图**

**Runtime data area 主要包括五个部分：Heap (堆), Method Area(方法区域),**

**Java Stack(java 的栈), Program Counter(程序计数器), Native method**

**stack(本地方法栈)。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。**

**2 PC 寄存器** PC 寄存器是用于存储每个线程下一步将执行的 JVM 指令，如该方法为 native 的，则 PC寄存器中不存储任何信息。

3  **JVM 栈 线程私有的**，每个线程创建的同时都会创建 JVM 栈，JVM 栈中存放的为当前线程中八种基本类型，局域变量，非基本类型的对象在 JVM 栈上仅存放一个指向堆上的地址，因此 Java中基本类型的变量是值传递，而非基本类型的变量是引用传递。并且当线程运行完毕后， 这些内存也就被自动回收。当 JVM 栈的空间不足时，会抛出 StackOverflowError 的错误，当 JVM 参数设置为-Xss1K，运行后会报出类似下面的错误：Exception in thread "Thread-0" java.lang.StackOverflowError。

**4 堆（Heap）** 所有线程共享的，用来存储对象实例以及数组值，通过 new 创建的对象的内存都在此分配，其大小通过-Xms 和-Xmx 来控制，-Xms 为 最小 Heap 内存，默认为物理内存的。-Xmx 为 JVM 可申请的最大 Heap 内存，默认当空余堆内存小于 40% 时， JVM 会增大 Heap 的大小到-Xmx 指定的大小，通常都会将-Xms 和-Xmx 的值设成一样。

Heap 分为 New Generation 和 Old Generation两块：

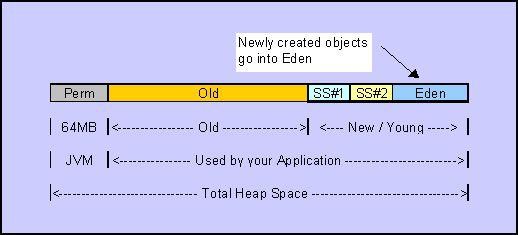
（1）.New Generation

新生代，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 构成，可通过-Xmn 参数来指定其大小，Eden Space 的大小和两块 Survivor

Space 的大小比例默认为 8，即当 New Generation 的大小为 10M 时，Eden Space 的大小为 8M，两块 Survivor Space 各占 1M，这个比例可通过-XX:SurvivorRatio 来指定。

（2）.Old Generation

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。



5 **方法区域（Method Area）**

方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final 类型的常量、类中的 Field 信息、类中的方法信息，当开发人员在程序中通过 Class 对象中的 getName、isInterface 等方法来获取信息时，这些数据都来源于方法区域，可见方法区域的重要性，同样，方法区域也是全局共享的，在一定的条件下它也会被 GC，当方法区域需要使用的内存超过其允许的大小时，会抛出 OutOfMemory 的错误信息。在 Sun JDK 中这块区域对应的为 Permanet Generation，又称为持久代，默认为 64M，可通过-XX:PermSize

以及-XX:MaxPermSize 来指定其大小。方法区包含运行时常量池（Runtime Constant Pool），String常量池。

**6 String内存分配**

**（1）使用关键字new，如：String s1 = new String(“myString”);**

第一种方式通过关键字new定义过程：在程序编译期，编译程序先去字符串常量池检查，是否存在“myString”,如果不存在，

则在常量池中开辟一个内存空间存放“myString”；如果存在的话，则不用重新开辟空间，保证常量池中只有一个“myString”常量，节省内存空间。

然后在内存堆中开辟一块空间存放new出来的String实例，在栈中开辟一块空间，命名为“s1”，存放的值为堆中String实例的内存地址，这个过程就是将引用s1指向new出来的String实例。

各位，最模糊的地方到了！堆中new出来的实例和常量池中的“myString”是什么关系呢？等我们分析完了第二种定义方式之后再回头分析这个问题。

**（2）直接定义，如：String s1 = “myString”;**

第二种方式直接定义过程：在程序编译期，编译程序先去字符串常量池检查，是否存在“myString”，如果不存在，则在常量池中开辟一个内存空间存放“myString”；如果存在的话，则不用重新开辟空间。

然后在栈中开辟一块空间，命名为“s1”，存放的值为常量池中“myString”的内存地址。常量池中的字符串常量与堆中的String对象有什么区别呢？为什么直接定义的字符串同样可以调用String对象的各种方法呢？

常量池中的字符串常量实质上是一个String实例，与堆中的String实例是克隆关系

**（3）String str=”kv”+”ill”+” “+”ans”;**

由于String类的immutable性质,这一说又要说很多，大家只 要知道String的实例一旦生成就不会再改变了，比如说：String str=”kv”+”ill”+” “+”ans”; 就是有4个字符串常量，

首先”kv”和”ill”生成了”kvill”存在内存中，然后”kvill”又和” ” 生成 “kvill “存在内存中，最后又和生成了”kvill ans”;并把这个字符串的地址赋给了str,就是因为String的

”不可变”产生了很多临时变量，这也就是为什么建议用StringBuffer的原 因了，因为StringBuffer是可改变的。

**7 举例分析**

**（1）堆和栈的初始化**

class BirthDate {

private int day;

private int month;

private int year;

public BirthDate(int d, int m, int y) {

day = d;

month = m;

year = y;

}

//省略get,set方法………

}

public class Test{

public static void main(String args[]){

int date = 9;

Test test = new Test();

test.change(date);

BirthDate d1= new BirthDate(7,7,1970);

}

public void change1(int i){

i = 1234;

}

　　对于以上这段代码，date为局部变量，i,d,m,y都是形参为局部变量，day，month，year为成员变量。下面分析一下代码执行时候的变化：

　　1. main方法开始执行：int date = 9;

　　　date局部变量，基础类型，引用和值都存在栈中。

　　2. Test test = new Test();

　　　test为对象引用，存在栈中，对象(new Test())存在堆中。

　　3. test.change(date);

　　　i为局部变量，引用和值存在栈中。当方法change执行完成后，i就会从栈中消失。

　　4. BirthDate d1= new BirthDate(7,7,1970);

　　　d1 为对象引用，存在栈中，对象(new BirthDate())存在堆中，其中d，m，y为局部变量存储在栈中，且它们的类型为基础类型，因此　　它们的数据也存储在栈中。 day,month,year为成员变量，它们存储在堆中(new BirthDate()里面)。当BirthDate构造方法执行完之　　　　后，d,m,y将从栈中消失。

　　5.main方法执行完之后，date变量，test，d1引用将从栈中消失，new Test(),new BirthDate()将等待垃圾回收。

**（2）String创建**

String str = new String("abc");

String str = "abc";

两种的形式来创建，第一种是用new()来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。而第二种是先在栈中创建一个对 String类的对象引用变量str，然后通过符号引用去字符串常量池 里找有没有"abc",如果没有，则将"abc"存放进字符串常量池 ，并令str指向”abc”，如果已经有”abc” 则直接令str指向“abc”。

**（3）String内容比较**

比较类里面的数值是否相等时，用equals()方法；当测试两个包装类的引用是否指向同一个对象时，用==，下面用例子说明上面的理论。

String str1 = "abc";

String str2 = "abc";

System.out.println(str1==str2); //true

可以看出str1和str2是指向同一个对象的。

String str1 =new String ("abc");

String str2 =new String ("abc");

System.out.println(str1==str2); // false

用new的方式是生成不同的对象。每一次生成一个。

**8 JVM 对象内存回收**

对新生代的对象的收集称为 minor GC，对旧生代的对象的收集称为 Full GC，调用 System.gc()为 Full GC，常见算法：

1）、 标记/清除算法

Heap 最明显的一种方法，对每一个对象都提供一个关联的引用计数，以此来标识该对象是否被使用，当这个计数为零时说明这个对象已经不再被使用了。标记和清除两个阶段的效率都不高，因为这两个阶段都需要遍历内存中的对象；标记清除之后会产生大量不连续的内存碎片，内存空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。因此 JVM 并没有采用引用计数。

2）、复制算法（新生代 Minor GC)

将可用内存按容量划分为大小相等的两块，每次使用其中的一块。当这一块的内存用完了，就将还存活的对象复制到另一块内存上，然后把这一块内存所有的对象一次性清理掉简单高效，优化了标记/清除算法的效率低、内存碎片多的问题。但是将内存缩小为原来的一半，浪费了一半的内存空间；如果对象的存活率很高，耗费的时间代价也是不可忽视的。由于新生代中的对象几乎都是朝生夕死的（达到98%），虚拟机都采用复制算法来回收新生代。由于新生代的对象存活率低，所以并不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的From Survivor空间、To Survivor空间，三者的比例为8：1：1。每次使用Eden和From Survivor区域，To Survivor作为保留空间。GC开始时，对象只会存在于Eden区和From Survivor区，To Survivor区是空的。GC进行时，Eden区中所有存活的对象都会被复制到To Survivor区，

接着清空Eden区和From Survivor区，新生代中存活的对象都在To Survivor区。

3）、 标记/整理算法（老年代 Full GC)

老年代中由于对象的存活率非常高，复制算法就不合适了。标记/整理算法。事实上，标记/整理算法的标记过程任然与标记/清除算法一样，但后续步骤不是直接对可回收对象进行回收，

而是让所有存活的对象都向一端移动，然后直接清理掉端边线以外的内存。缺点:效率也不高，不仅要标记存活对象，还要整理所有存活对象的引用地址，在效率上不如复制算法。

4）、分代收集算法

当前商业虚拟机都采用分代收集算法，终极算法，结合了前几种算法的优点，将算法组合使用进行垃圾回收。

分代收集算法的思想是按对象的存活周期不同将内存划分为几块，一般是把Java堆分为新生代和老年代、永久代（很少回收），这样就可以根据各个年代的特点采用最合适的收集算法。

新生代：朝生夕灭，存活时间很短，复制算法。

老年代：经过多次Minor GC而存活下来，存活周期长，标记/整理。

效率：复制算法 > 标记/整理算法 > 标记/清除算法（标记/清除算法有内存碎片问题，给大对象分配内存时可能会触发新一轮垃圾回收）

内存整齐率：复制算法 = 标记/整理算法 > 标记/清除算法

内存利用率：标记/整理算法 = 标记/清除算法 > 复制算法

9 **参数配置**

**（1）静态参数：**

**-Xmn**

新生代内存大小的最大值，包括 E 区和两个 S 区的总和，使用方法如：- Xmn65535，-Xmn1024k，-Xmn512m，-Xmn1g (-Xms,-Xmx 也是种写法)

**-Xms**

初始堆的大小，也是堆大小的最小值，默认值是总共的物理内存/64（且小于1G），默认情况下，当堆中可用内存小于 40%

(这个值可以用-XX:MinHeapFreeRatio 调整，如-X:MinHeapFreeRatio=30)时，堆内存会开始增加，一直增加到-Xmx 的大小；

**-Xmx**

堆的最大值，默认值是总共的物理内存/64（且小于 1G），如果 Xms 和 Xmx 都不设置，则两者大小会相同；

整个堆的大小=年轻代大小+年老代大小，堆的大小不包含持久代大小，如果增大了年轻代，年老代相应就会减小

线上生产环境，Xms 和 Xmx 设置的值必须一样，原因与年轻代一样——防止抖动；

**-Xss**

这个参数用于设置每个线程的栈内存，默认 1M，一般来说是不需要改的。除非代码不多，可以设置的小点，另外一个相似的参数是-XX:ThreadStackSize， 这两个参数在 1.6 以前，都是谁设置在后面，谁就生效；1.6 版本以后，-Xss 设置在后面，则以-Xss 为准，-XXThreadStackSize 设置在后面，则主线程以-Xss 为准，其它线程以-XX:ThreadStackSize 为准。

**（2）非 Stable静态参数：**

-XX:NewSize=2.125m 新生代对象生成时占用内存的默认值

-XX:MaxNewSize=size 新生成对象能占用内存的最大值

-XX:MaxPermSize=64m 方法区所能占用的最大内存

-XX:PermSize=64m 方法区分配的初始内存

-XX:MaxTenuringThreshold=15 对象在新生代存活区切换的次数

-XX:MaxHeapFreeRatio=70 GC 后 java 堆中空闲量占的最大比例，大于该值，则堆内存会减少

-XX:MinHeapFreeRatio=40 GC 后 java 堆中空闲量占的最小比例，小于该值，则堆内存会增加

-XX:NewRatio=2 新生代内存容量与老生代内存容量的比例

-XX:ReservedCodeCacheSize= 32m 保留代码占用的内存容量

-XX:ThreadStackSize=512 设置线程栈大小，若为 0 则使用系统默认值

XX:+ScavengeBeforeFullGC 新生代 GC 优先于 Full GC 执行

**（3）调试参数：**

-XX:-CITime 打印消耗在JIT 编译的时间

-XX:ErrorFile=./hs\_err\_pid<pid>.log 保存错误日志或者数据到文件中

-XX:-ExtendedDTraceProbes 开 启 solaris 特 有 的

-XX:HeapDumpPath=./java\_pid<pid>.hprof 指定导出堆信息时的路径或文件名

-XX:-HeapDumpOnOutOfMemoryError 当首次遭遇OOM 时导出此时堆中相关信息

-XX:OnError="<cmd args>;<cmd args>" 出现致命 ERROR 之后运行自定义命令

-XX:OnOutOfMemoryError="<cmd args>;<cmd args>" 当首次遭遇OOM 时执行自定义命令

-XX:-PrintClassHistogram 遇到 Ctrl-Break 后打印类实例的柱状信息，与 jmap -

-XX:-PrintConcurrentLocks 遇到 Ctrl-Break 后打印并发锁的相关信息，与jstack

-XX:-PrintCommandLineFlags 打印在命令行中出现过的标记

-XX:-PrintCompilation 当一个方法被编译时打印相关信息

-XX:-PrintGC 每次 GC 时打印相关信息

-XX:-PrintGC Details 每次 GC 时打印详细信息

-XX:-PrintGCTimeStamps 打印每次 GC 的时间戳

-XX:-TraceClassLoading 跟踪类的加载信息

-XX:-TraceClassLoadingPreorder 跟踪被引用到的所有类的加载信息

-XX:-TraceClassResolution 跟踪常量池

-XX:-TraceClassUnloading 跟踪类的卸载信息

-XX:-TraceLoaderConstraints 跟踪类加载器约束的相关

**10 调优方法**

一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1）.多数的 Java 应用不需要在服务器上进行 GC 优化；

2）.多数导致 GC 问题的 Java 应用，都不是因为我们参数设置错误，而是代码问题；

3）.在应用上线之前，先考虑将机器的 JVM 参数设置到最优（最适合）；

4）.减少创建对象的数量；

5）.减少使用全局变量和大对象；

6）.GC 优化是到最后不得已才采用的手段；

7）.在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多；

8）.将转移到老年代的对象数量降低到最小；

9）.减少 full GC 的执行时间；

10）.减少使用全局变量和大对象；

11）.调整新生代的大小到最合适；

12）.设置老年代的大小为最合适；

13）.选择合适的 GC 收集器；

**11 调优实例**

**实例 1：**

笔者昨日发现部分开发测试机器出现异常：java.lang.OutOfMemoryError:GC overhead limit exceeded，这个异常代表：GC 为了释放很小的空间却耗费了太多的时间，其原因一般有两个：

1，堆太小，

2，有死循环或大对象； 笔者首先排除了第 2 个原因，因为这个应用同时是在线上运行的，如果有问题，早就挂了。所以怀疑是这台机器中堆设置太小；使用 ps -ef |grep "java"查看，发现：该应用的堆区设置只有 768m，而机器内存有 2g，机器上只跑这一个 java 应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多；笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，于是改成下面的情况：跟踪运行情况发现，相关异常没有再出现；

**实例 2：**

一个服务系统，经常出现卡顿，分析原因，发现 Full GC 时间太长：

jstat -gcutil:

S0 S1 E O P YGC YGCT FGC FGCT GCT

12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993

发现 Young GC 执行了 54 次，耗时 2.047 秒，每次 Young GC 耗时 37ms，在正常范围。

而 Full GC 执行了 5 次，耗时 6.946 秒，每次平均 1.389s，数据显示出来的问题是：Full GC 耗时较长，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为 1:9，这就是问题的原因：

1，新生代太小，导致对象提前进入老年代，触发老年代发生 Full GC；

2，老年代较大，进行 Full GC 时耗时较大；

优化的方法是调整 NewRatio 的值，调整到 4，发现 Full GC 没有再发生，只有 Young GC 在执行。这就是把对象控制在新生代就清理掉，没有进入老年代。

**实例 3：**

一应用在性能测试过程中，发现内存占用率很高，Full GC 频繁，使用 sudo

-u admin -H jmap -dump:format=b,file=文件名.hprof pid 来 dump 内存，生成 dump 文件，并使用 Eclipse 下的 mat 差距进行分析，发现：

从图中可以看出，这个线程存在问题，队列 LinkedBlockingQueue 所引用的大量对象并未释放，导致整个线程占用内存高达 378m，此时通知开发人员进行代码优化，将相关对象释放掉即可。

**12 程序健壮性**

1）、尽早释放无用对象的引用。 好的办法是使用临时变量的时候，让引用变量在退出活动域后，自动设置为null，暗示垃圾收集器来收集该对象，防止发生内存泄露。对于仍然有指针指向的实例，jvm就不会回收该资源,因为垃圾回收会将值为null的对象作为垃圾，提高GC回收机制效率；

2）、定义字符串应该尽量使用 String str="hello"; 的形式 ，避免使用String str = new String("hello"); 的形式。因为要使用内容相同的字符串，不必每次都new一个String。

3）、我们的程序里不可避免大量使用字符串处理，避免使用String，应大量使用StringBuffer ，因为String被设计成不可变(immutable)类，所以它的所有对象都是不可变对象，请看下列代码；

4）、尽量少用静态变量 ，因为静态变量是全局的，GC不会回收的；

5）、尽量避免在类的构造函数里创建、初始化大量的对象，防止在调用其自身类的构造器时造成不必要的内存资源浪费，尤其是大对象，JVM会突然需要大量内存，

这时必然会触发GC优化系统内存环境；显示的声明数组空间，而且申请数量还极大。

以下是初始化不同类型的对象需要消耗的时间：

运算操作   示例    标准化时间

本地赋值    i = n 1.0

实例赋值    this.i = n 1.2

方法调用    Funct() 5.9

新建对象    New Object() 980

新建数组    New int[10] 3100

从表1可以看出，新建一个对象需要980个单位的时间，是本地赋值时间的980倍，是方法调用时间的166倍，而新建一个数组所花费的时间就更多了。

6）、尽量在合适的场景下使用对象池技术 以提高系统性能，缩减缩减开销，但是要注意对象池的尺寸不宜过大，及时清除无效对象释放内存资源，综合考虑应用运行环境的内存资源限制，避免过高估计运行环境所提供内存资源的数量。

7）、大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理 ，然后解决一块释放一块的策略。

8）、不要在经常调用的方法中创建对象 ，尤其是忌讳在循环中创建对象。可以适当的使用hashtable，vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次new之后又丢弃。

9）、一般都是发生在开启大型文件或跟数据库一次拿了太多的数据，造成 Out Of Memory Error 的状况，这时就大概要计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。

10）、尽量少用finalize函数 ，因为finalize()会加大GC的工作量，而GC相当于耗费系统的计算能力。

11）、不要过滥使用哈希表，有一定开发经验的开发人员经常会使用hash表来缓存一些数据，从而提高系统的运行速度。

## 六、JVM调优总结

**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：

1 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JVM内存大小=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。更具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。

2 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，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

**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进行判断。

**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，适用于科学技术和后台处理等。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。

**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

**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:+PrintGCDetails

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。

**3 常见配置汇总**

**（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。

**4 调优总结**

**（1）年轻代大小选择**

1）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。

2）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。

**（2）年老代大小选择**

1）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

2）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。

**（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：

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

2）-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