Trace跟踪参数:

* -verbose:gc
* -XX:+printGC
* 可以打印GC的简要信息
  + [GC 4790K->374K(15872K), 0.0001606 secs]
  + [GC 4790K->374K(15872K), 0.0001474 secs]
  + [GC 4790K->374K(15872K), 0.0001563 secs]
  + [GC 4790K->374K(15872K), 0.0001682 secs]

如上所示我们为jvm配置-verbose:gc和-XX:printGC可以看到jvm做GC打印出来的参数

如上所示做了四次GC分别表示回收前的堆内存空间与回收后的内存空间可以看出大概回收

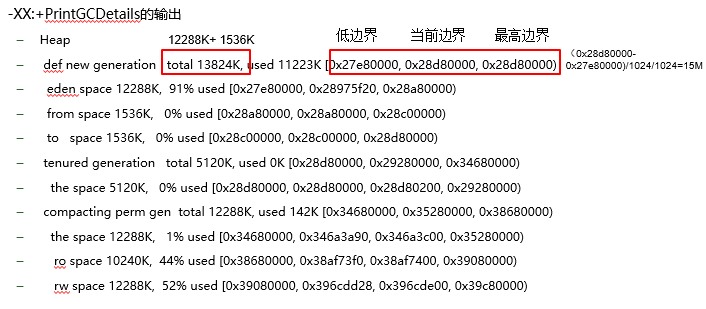
了4M左右的内存，和回收所需要的时间。

* -XX:+PrintGCDetails
  + 打印GC详细信息
* -XX:+PrintGCTimeStamps
  + 打印CG发生的时间戳
* 例
  + [GC[DefNew: 4416K->0K(4928K), 0.0001897 secs] 4790K->374K(15872K), 0.0002232 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

温故一下java堆内存模型：



然后我们可以看下面GC打出的详细信息



如上图所示的是

我们可以看出新生代的低边界与高边界大概有15M的空间

Eden区使用:12M

From幸存区:1.5M

To 幸存区: 1.5 M

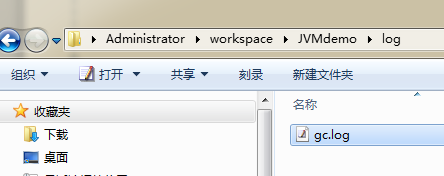
Tenured genetation 老年代5M

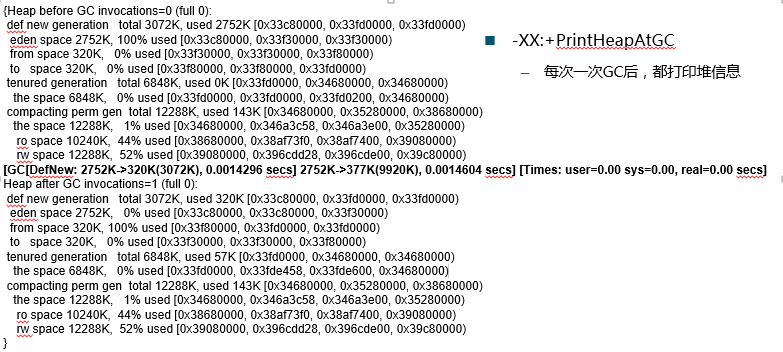
Compacting perm gen 永久代 12M

通过如上的GC参数我们可以看出当前java内存的使用情况。

当然以后我们也要对java内存进行分析，所以希望把gc的日志文件输出到指定路径去通过如下所示的GC日志配置，我们可以把gc的日志输出到文件中。

* Xloggc:log/gc.log
  + 指定GC log的位置，以文件输出
  + 帮助开发人员分析问题





如上图黑色部分的信息所显示的，做了一次GC大概回收了5M的空间回收以前

内存中新生代eden区中的内存2752k经过一次GC以后剩余377k，其中幸存区中剩余320k

有57k被GC识别为老年代。

* -XX:+TraceClassLoading
  + 监控类的加载
    - [Loaded java.lang.Object from shared objects file]
    - [Loaded java.io.Serializable from shared objects file]
    - [Loaded java.lang.Comparable from shared objects file]
    - [Loaded java.lang.CharSequence from shared objects file]
    - [Loaded java.lang.String from shared objects file]
    - [Loaded java.lang.reflect.GenericDeclaration from shared objects file]
    - [Loaded java.lang.reflect.Type from shared objects file]

配置如上的参数，可以到到jvm的类加载的日志。

如下所示配置了如下所示的参数以后可以打印累的信息。

* -XX:+PrintClassHistogram
  + 按下Ctrl+Break后，打印类的信息：
* num #instances #bytes class name
* ----------------------------------------------
* 1: 890617 470266000 [B
* 2: 890643 21375432 java.util.HashMap$Node
* 3: 890608 14249728 java.lang.Long
* 4: 13 8389712 [Ljava.util.HashMap$Node;
* 5: 2062 371680 [C
* 6: 463 41904 java.lang.Class
  + 分别显示：序号、实例数量、总大小、类型

堆的分配参数:

* Xmx –Xms
  + 指定最大堆和最小堆
  + -Xmx20m -Xms5m 运行代码：
    - System.out.print("Xmx=");
    - System.out.println(Runtime.getRuntime().maxMemory()/1024.0/1024+"M");
    - System.out.print("free mem=");
    - System.out.println(Runtime.getRuntime().freeMemory()/1024.0/1024+"M");
    - System.out.print("total mem=");
    - System.out.println(Runtime.getRuntime().totalMemory()/1024.0/1024+"M");

输出结果:

Xmx=19.375M

free mem=4.342750549316406M

total mem=4.875M

* Xmx20m -Xms5m 运行代码
  + - byte[] b=new byte[1\*1024\*1024];
    - System.out.println("分配了1M空间给数组");
    - System.out.print("Xmx=");
    - System.out.println(Runtime.getRuntime().maxMemory()/1024.0/1024+"M");
    - System.out.print("free mem=");
    - System.out.println(Runtime.getRuntime().freeMemory()/1024.0/1024+"M");
    - System.out.print("total mem=");
    - System.out.println(Runtime.getRuntime().totalMemory()/1024.0/1024+"M");
* 分配了1M空间给数组
* Xmx=19.375M
* free mem=3.4791183471679688M //java尽可能的维持最小堆。
* total mem=4.875M
* Xmx20m -Xms5m 运行代码
  + - b=new byte[4\*1024\*1024];
    - System.out.println("分配了4M空间给数组");
    - System.out.print("Xmx=");
    - System.out.println(Runtime.getRuntime().maxMemory()/1024.0/1024+"M");
    - System.out.print("free mem=");
    - System.out.println(Runtime.getRuntime().freeMemory()/1024.0/1024+"M");
    - System.out.print("total mem=");
    - System.out.println(Runtime.getRuntime().totalMemory()/1024.0/1024+"M");

分配了4M空间给数组

Xmx=19.375M

free mem=3.5899810791015625M

total mem=9.00390625M 总内存变多了

* -Xmx20m -Xms5m 运行代码
  + - System.gc();
    - System.out.println("回收内存");
    - System.out.print("Xmx=");
    - System.out.println(Runtime.getRuntime().maxMemory()/1024.0/1024+"M");
    - System.out.print("free mem=");
    - System.out.println(Runtime.getRuntime().freeMemory()/1024.0/1024+"M");
    - System.out.print("total mem=");
    - System.out.println(Runtime.getRuntime().totalMemory()/1024.0/1024+"M");
* 回收内存
* Xmx=19.375M
* free mem=6.354591369628906M //做一次GC之后自由使用内存又增多了。
* total mem=10.75390625M

可以通过如下所示的jvm参数来调整，最小新生代，新生代与老年代，eden与幸存区的堆区大小与比例。

* -Xmn
  + 设置新生代大小
* -XX:NewRatio
  + 新生代（eden+2\*s）和老年代（不包含永久区）的比值
  + 4 表示 新生代:老年代=1:4，即年轻代占堆的1/5
* -XX:SurvivorRatio
  + 设置两个Survivor区和eden的比
  + 8表示 两个Survivor :eden=2:8，即一个Survivor占年轻代的1/10

如下面一段代码，给堆区分配10M的空间

public static void main(String[] args) {

byte[] b=null;

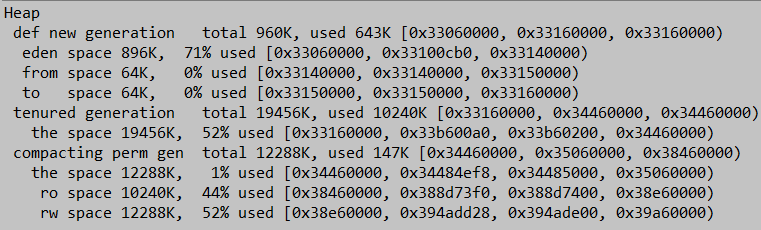
for(int i=0;i<10;i++)

b=new byte[1\*1024\*1024];

}

配置参数一:”

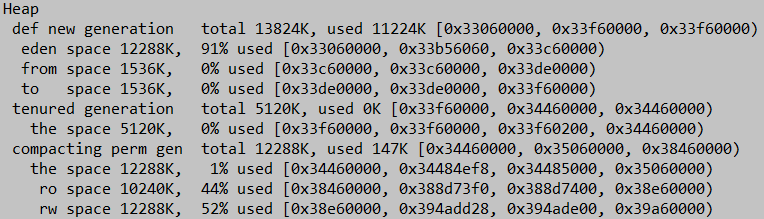
-Xmx20m -Xms20m -Xmn1m -XX:+PrintGCDetails



可以看出新生代大小于1M，剩下的全部放到了老年代几乎整个byte[]都存储到了老年代了

配置参数二:

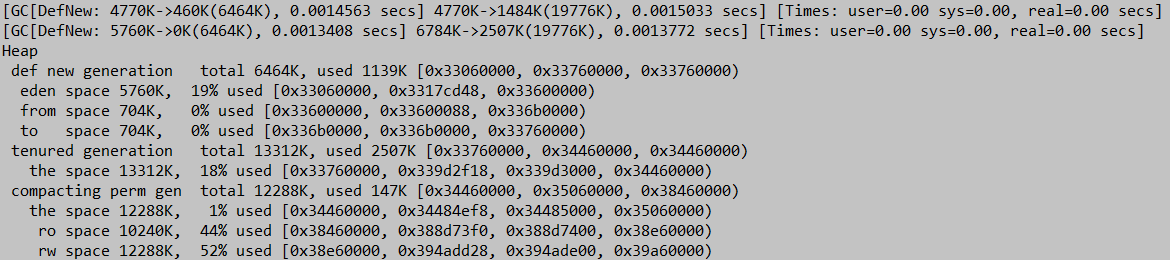
-Xmx20m -Xms20m -Xmn15m -XX:+PrintGCDetails



可以看出整个byte[]都分配到了eden区，并没有触发GC

配置参数三

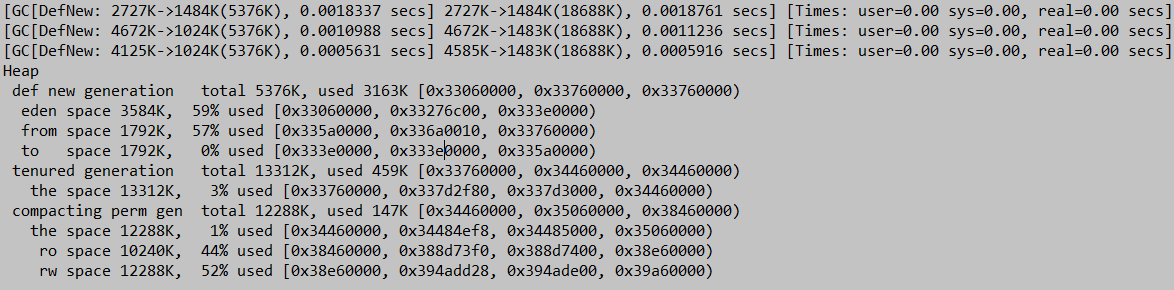
-Xmx20m -Xms20m –Xmn7m -XX:+PrintGCDetails



如图所示做了两次GC，其中幸存区的GC需要老年代来担保。

配置参数四:

-Xmx20m -Xms20m -Xmn7m -XX:SurvivorRatio=2 -XX:+PrintGCDetails

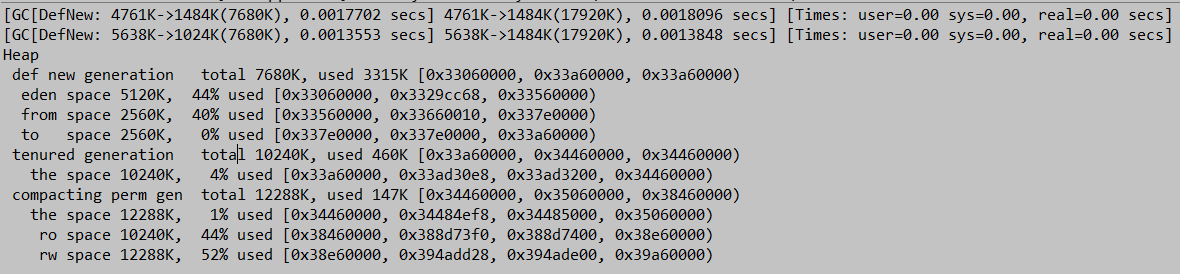


首先如上所示配置了sur:eden=2:2=1也就是所让两个幸存区的大小等于eden区

如上图可以看出来，当幸存区的堆增大之后，做了3次GC

配置参数四:

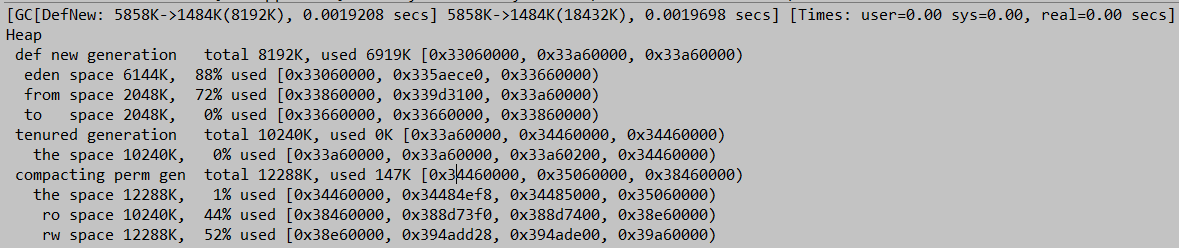
-Xmx20m -Xms20m -XX:NewRatio=1 -XX:SurvivorRatio=2 -XX:+PrintGCDetails



如上所示配置了新生代玉老年代的比例相等

配置参数五:

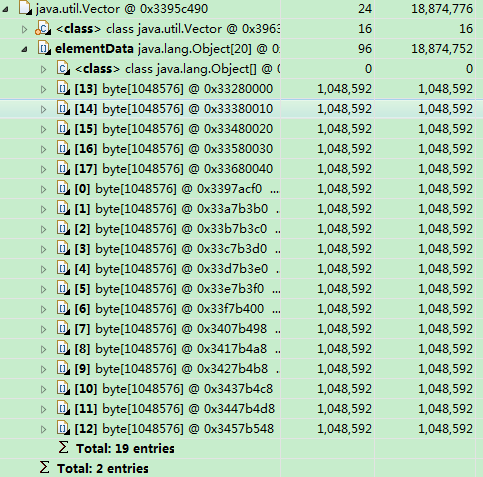
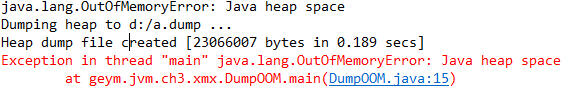
-Xmx20m -Xms20m -XX:NewRatio=1 -XX:SurvivorRatio=3 -XX:+PrintGCDetails



如上所示配置sur:end:2:3做了一次GC

堆溢出的情况:

* -XX:+HeapDumpOnOutOfMemoryError
  + OOM时导出堆到文件
* -XX:+HeapDumpPath
  + 导出OOM的路径
* -Xmx20m -Xms5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump
  + Vector v=**new Vector();**
  + **for(int i=0;i<25;i++)**
  + v.add(**new byte[1\*1024\*1024]);**



如上参数配置所示，我们分配最大堆内存20m但是我们创建了一个25m的数组，

这种情况下肯定是会出现堆溢出的，这时候我们可以配置如上参数，当堆溢出时把

溢出的日志文件进行绝对路径保存，可以看到如上图所示的，日志文件中只存放了

19个byte[]数组，达到了我们参数的最高限制，所以不能再分配了。

急救措施：如下比如在堆dump的时候执行脚本发邮件或者重启来拯救系统，保证系统的

正常运行。

* XX:OnOutOfMemoryError
  + 在OOM时，执行一个脚本
  + "-XX:OnOutOfMemoryError=D:/tools/jdk1.7\_40/bin/printstack.bat %p“
  + 当程序OOM时，在D:/a.txt中将会生成线程的dump
  + 可以在OOM时，发送邮件，甚至是重启程序

综上所述：

* 根据实际事情调整新生代和幸存代的大小
* 官方推荐新生代占堆的3/8
* 幸存代占新生代的1/10
* 在OOM时，记得Dump出堆，确保可以排查现场问题

永久区参数配置:

* XX:PermSize -XX:MaxPermSize
  + 设置永久区的初始空间和最大空间
  + 他们表示，一个系统可以容纳多少个类型

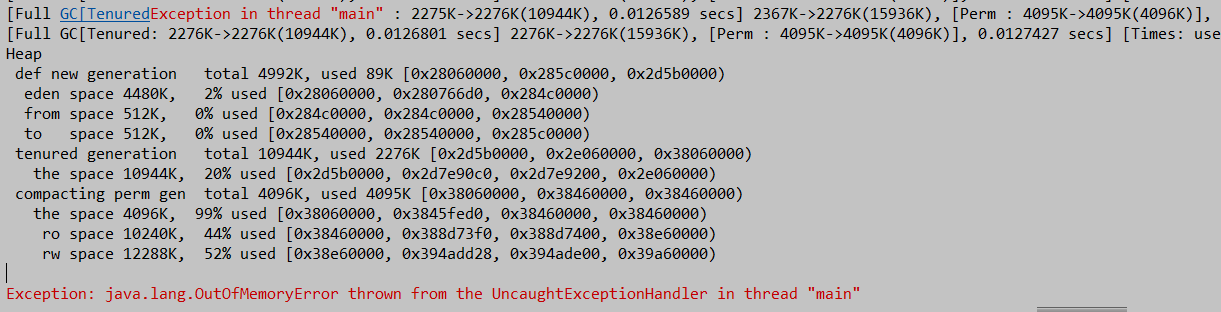
使用CGLIB等库的时候，可能会产生大量的类，这些类，有可能撑爆永久区导致OOM

for(int i=0;i<100000;i++){

CglibBean bean = new CglibBean("geym.jvm.ch3.perm.bean"+i,new HashMap());

}

如上所示，不断的产生新的类。



由于永久区内存爆满已至于我们系统产生了OOM.

如果堆空间没有用完也抛出了OOM，有可能是永久区导致的

栈内存参数分配:

* Xss
  + 通常只有几百K
  + 决定了函数调用的深度
  + 每个线程都有独立的栈空间
  + 局部变量、参数 分配在栈上

如下代码所示

public class TestStackDeep {

private static int count=0;

public static void recursion(long a,long b,long c){

long e=1,f=2,g=3,h=4,i=5,k=6,q=7,x=8,y=9,z=10;

count++;

recursion(a,b,c);

}

public static void main(String args[]){

try{

recursion(0L,0L,0L);

}catch(Throwable e){

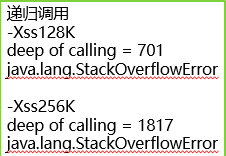
System.out.println("deep of calling = "+count);

e.printStackTrace();

}

}

}



如上所示是通过为栈内存分配了内存之后，决定了函数的调用深度。因此当我们

碰到SOF的的时候后可以适当的调整栈的内存大小。由于GC会及时回收栈内存，因此

栈内存也不益过大。