

<http://www.cnblogs.com/smyhvae/p/4736162.html>

本文主要内容：

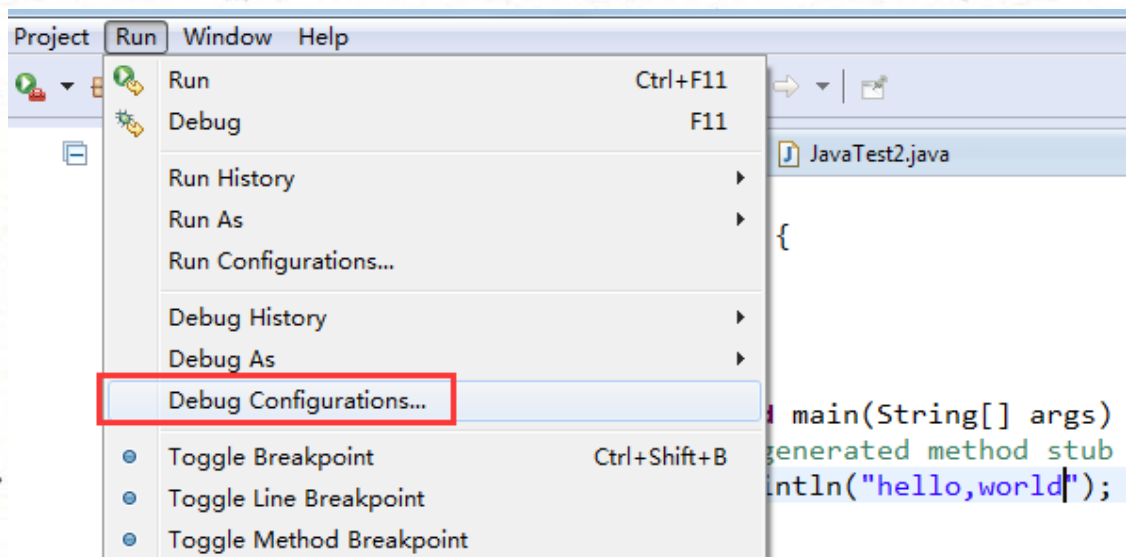
- Trace跟踪参数
- 堆的分配参数
- 栈的分配参数

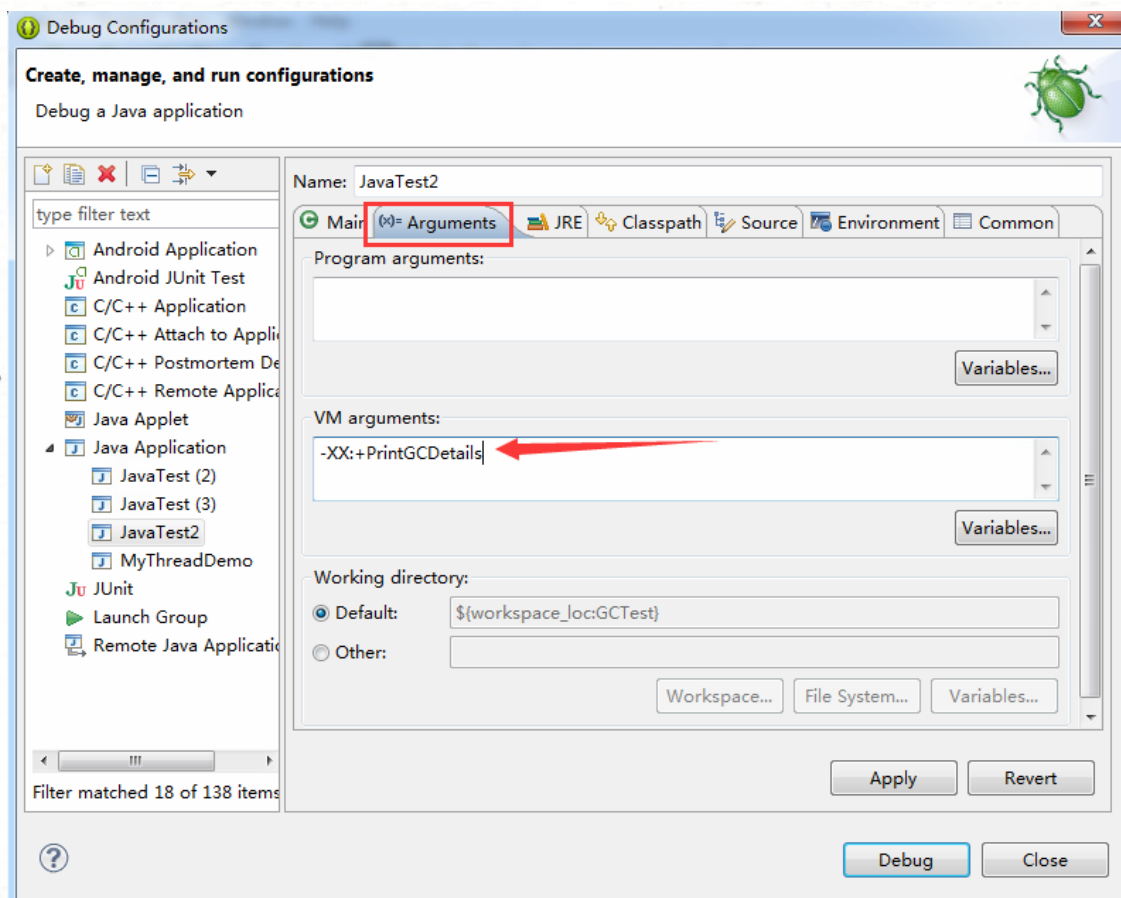
零、在**IDE**的后台打印**GC**日志：

既然学习JVM，阅读GC日志是处理Java虚拟机内存问题的基础技能，它只是一些人为确定的规则，没有太多技术含量。

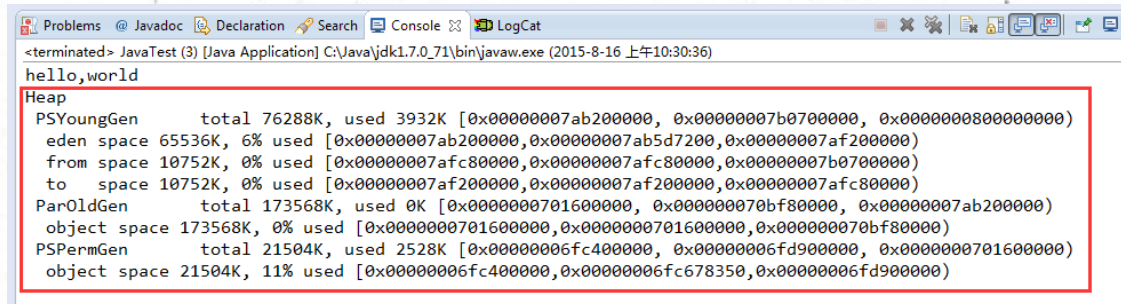
既然如此，那么在IDE的控制台打印GC日志是必不可少的了。现在就告诉你怎么打印。

(1) 如果你用的是Eclipse，打印GC日志的操作如下：

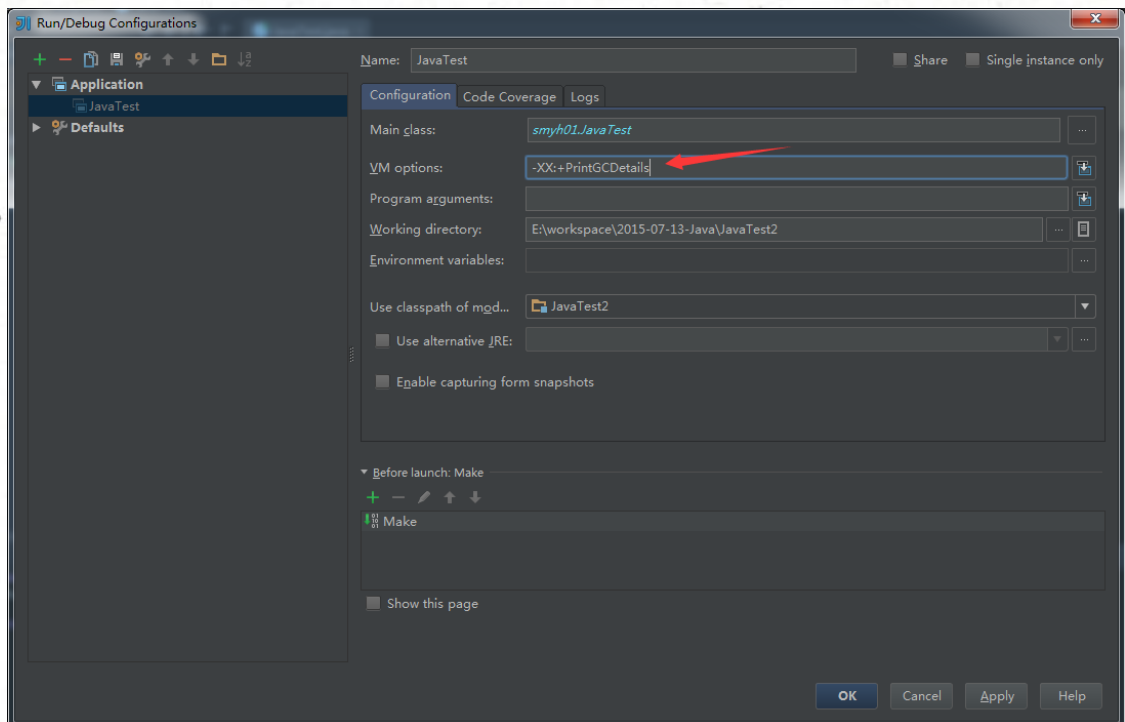
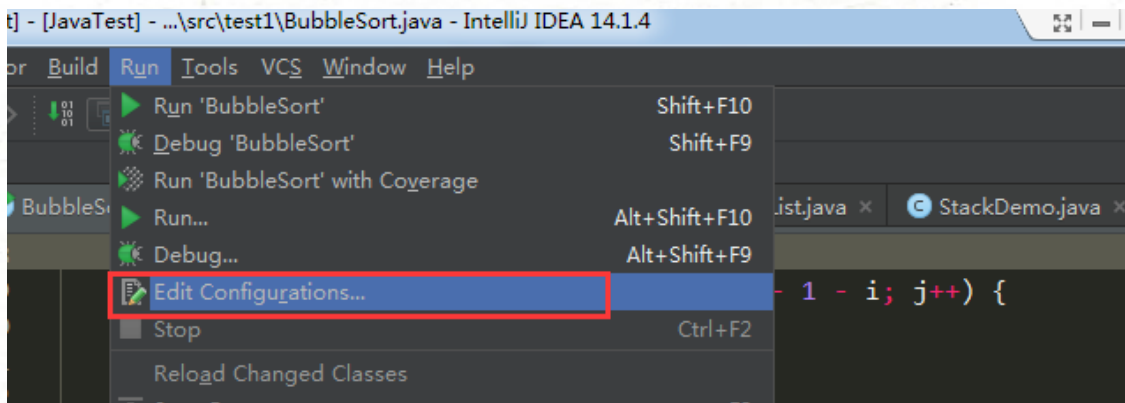




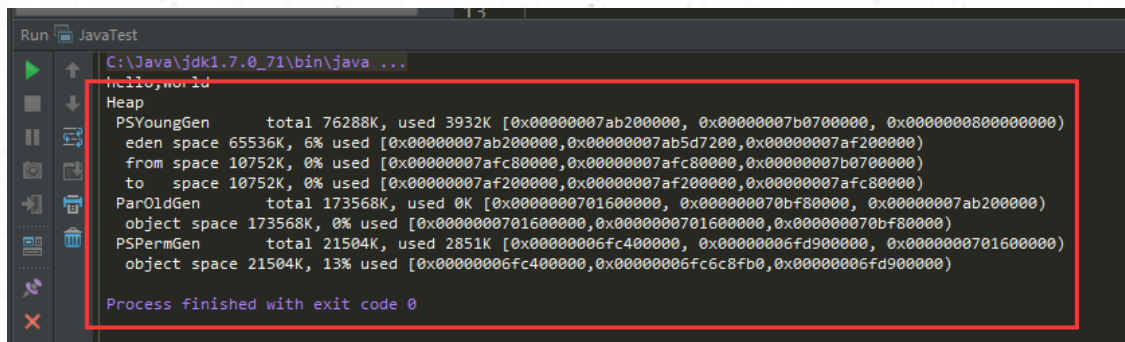
在上图的箭头处加上**-XX:+PrintGCDetails**这句话。于是，运行程序后，GC日志就可以打印出来了：



(2) 如果你用的是IntelliJ IDEA，打印GC日志的操作如下：



在上图的箭头处加上**-XX:+PrintGCDetails**这句话。于是，运行程序后，GC日志就可以打印出来了：



当然了，光有**-XX:+PrintGCDetails**这一句参数肯定是不够的，下面我们详细介绍一下更多的参数配置。

一、Trace跟踪参数：

1、打印GC的简要信息：

```
-verbose:gc  
-XX:+printGC
```

解释：可以打印GC的简要信息。比如：

```
[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]
```

上方日志的意思是说，GC之前，用了4M左右的内存，GC之后，用了374K内存，一共回收了将近4M。内存大小一共是16M左右。

2、打印GC的详细信息：

```
-XX:+PrintGCDetails
```

解释：打印GC详细信息。

```
-XX:+PrintGCTimeStamps
```

解释：打印CG发生的时间戳。

理解GC日志的含义：

例如下面这段日志：

```
[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]
```

上方日志的意思是说：这是一个新生代的GC。方括号内部的“4416K->0K(4928K)”含义是：“GC前该内存区域已使用容量->GC后该内存区域已使用容量（该内存区域总容量）”。而在方括号之外的“4790K->374K(15872K)”表示“GC前Java堆已使用容量->GC后Java堆已使用容量（Java堆总容量）”。

再往后看，“0.0001897 secs”表示该内存区域GC所占用的时间，单位是秒。

再比如下面这段GC日志：

-XX:+PrintGCDetails的输出

	12288K+ 1536K	低边界	当前边界	最高边界
- Heap				
- def new generation	total 13824K, used 11223K	[0x27e80000, 0x28d80000, 0x28d80000]		(0x28d80000-0x27e80000)/1024/1024=15M
- eden space 12288K, 91% used		[0x27e80000, 0x28975f20, 0x28a80000]		
- from space 1536K, 0% used		[0x28a80000, 0x28a80000, 0x28c00000]		
- to space 1536K, 0% used		[0x28c00000, 0x28c00000, 0x28d80000]		
- tenured generation total 5120K, used 0K		[0x28d80000, 0x29280000, 0x34680000]		
- the space 5120K, 0% used		[0x28d80000, 0x28d80000, 0x28d80200, 0x29280000]		
- compacting perm gen total 12288K, used 142K		[0x34680000, 0x35280000, 0x38680000]		
- the space 12288K, 1% used		[0x34680000, 0x346a3a90, 0x346a3c00, 0x35280000]		
- ro space 10240K, 44% used		[0x38680000, 0x38af73f0, 0x38af7400, 0x39080000]		
- rw space 12288K, 52% used		[0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000]		

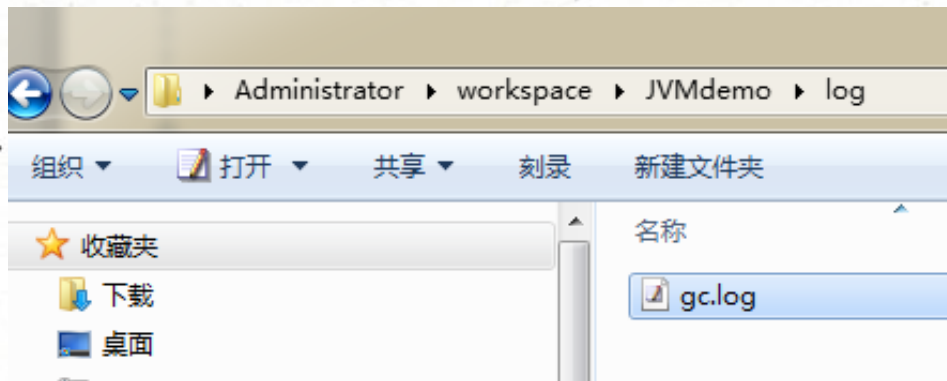
上图中，我们先看一下用红框标注的“[0x27e80000, 0x28d80000, 0x28d80000)”的含义，它表示新生代在内存当中的位置：第一个参数是申请到的起始位置，第二个参数是申请到的终点位置，第三个参数表示最多能申请到的位置。上图中的例子表示新生代申请到了15M的控件，而这个**15M是等于：（eden space的12288K）+（from space的1536K）+（to space的1536K）**。

疑问：分配到的新生代有**15M**，但是可用的只有**13824K**，为什么会有这个差异呢？等我们在后面的文章中学习到了GC算法之后就明白了。

3、指定GC log的位置：

-Xloggc:log/gc.log

解释：指定**GC log**的位置，以文件输出。帮助开发人员分析问题。



-XX:+PrintHeapAtGC

解释：每一次GC前和GC后，都打印堆信息。

例如：


```

{Heap before GC invocations=0 (full 0):
def new generation total 3072K, used 2752K [0x33c80000, 0x33fd0000, 0x33fd0000)
eden space 2752K, 100% used [0x33c80000, 0x33f30000, 0x33f30000)
from space 320K, 0% used [0x33f30000, 0x33f80000, 0x33f80000)
to space 320K, 0% used [0x33f80000, 0x33f80000, 0x33fd0000)
tenured generation total 6848K, used 0K [0x33fd0000, 0x34680000, 0x34680000)
the space 6848K, 0% used [0x33fd0000, 0x33fd0000, 0x33fd0200, 0x34680000)
compacting perm gen total 12288K, used 143K [0x34680000, 0x35280000, 0x38680000)
the space 12288K, 1% used [0x34680000, 0x346a3c58, 0x346a3e00, 0x35280000)
ro space 10240K, 44% used [0x38680000, 0x38af73f0, 0x38af7400, 0x39080000)
rw space 12288K, 52% used [0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000)
[GC]DefNew: 2752K->320K(3072K), 0.0014296 secs] 2752K->377K(9920K), 0.0014604 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap after GC invocations=1 (full 0):
def new generation total 3072K, used 320K [0x33c80000, 0x33fd0000, 0x33fd0000)
eden space 2752K, 0% used [0x33c80000, 0x33c80000, 0x33f30000)
from space 320K, 100% used [0x33f80000, 0x33fd0000, 0x33fd0000)
to space 320K, 0% used [0x33f30000, 0x33f30000, 0x33f80000)
tenured generation total 6848K, used 57K [0x33fd0000, 0x34680000, 0x34680000)
the space 6848K, 0% used [0x33fd0000, 0x33fde458, 0x33fde600, 0x34680000)
compacting perm gen total 12288K, used 143K [0x34680000, 0x35280000, 0x38680000)
the space 12288K, 1% used [0x34680000, 0x346a3c58, 0x346a3e00, 0x35280000)
ro space 10240K, 44% used [0x38680000, 0x38af73f0, 0x38af7400, 0x39080000)
rw space 12288K, 52% used [0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000)
}

```

■ -XX:+PrintHeapAtGC

- 每次一次GC后，都打印堆信息

上图中，红框部分正好是一次GC，红框部分的前面是GC之前的日志，红框部分的后面是GC之后的日志。

-XX:+TraceClassLoading

解释：监控类的加载。

例如：

```

[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]

```

-XX:+PrintClassHistogram

解释：按下Ctrl+Break后，打印类的信息。

例如：

序号 num	实例数量 #instances	总大小 #bytes	类型 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

二、堆的分配参数：

1、-Xmx -Xms：指定最大堆和最小堆

举例、当参数设置为如下时：

```
-Xmx20m -Xms5m
```

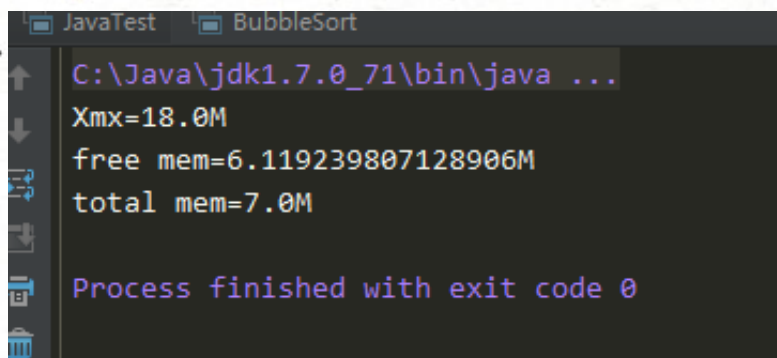
然后我们在程序中运行如下代码：

```
System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 /  
1024 + "M");    //系统的最大空间
```

```
System.out.println("free mem=" + Runtime.getRuntime().freeMemory() /  
1024.0 / 1024 + "M");    //系统的空闲空间
```

```
System.out.println("total mem=" + Runtime.getRuntime().totalMemory() /  
1024.0 / 1024 + "M");    //当前可用的总空间
```

运行效果：



```
JavaTest BubbleSort  
C:\Java\jdk1.7.0_71\bin\java ...  
Xmx=18.0M  
free mem=6.119239807128906M  
total mem=7.0M  
  
Process finished with exit code 0
```

保持参数不变，在程序中运行如下代码：（分配1M空间给数组）

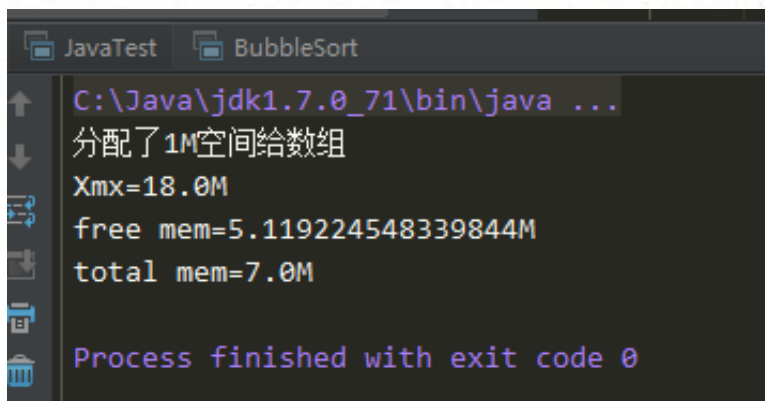
```
byte[] b = new byte[1 * 1024 * 1024];  
System.out.println("分配了1M空间给数组");
```

```
System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 /  
1024 + "M");    //系统的最大空间
```

```
System.out.println("free mem=" + Runtime.getRuntime().freeMemory() /  
1024.0 / 1024 + "M");    //系统的空闲空间
```

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

运行效果：



```
JavaTest BubbleSort
C:\Java\jdk1.7.0_71\bin\java ...
分配了1M空间给数组
Xmx=18.0M
free mem=5.119224548339844M
total mem=7.0M
Process finished with exit code 0
```

注：Java会尽可能将total mem的值维持在最小堆。

保持参数不变，在程序中运行如下代码：（分配10M空间给数组）



```
byte[] b = new byte[10 * 1024 * 1024];
System.out.println("分配了10M空间给数组");

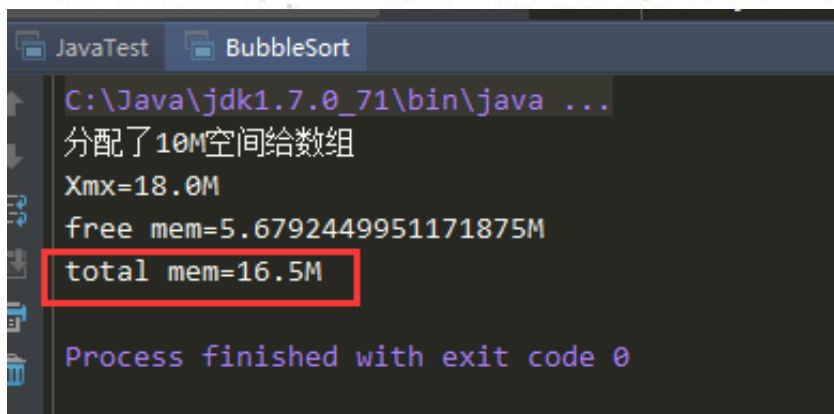
System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 /
1024 + "M"); //系统的最大空间

System.out.println("free mem=" + Runtime.getRuntime().freeMemory() /
1024.0 / 1024 + "M"); //系统的空闲空间

System.out.println("total mem=" + Runtime.getRuntime().totalMemory() /
1024.0 / 1024 + "M"); //当前可用的总空间
```



运行效果：



```
JavaTest BubbleSort
C:\Java\jdk1.7.0_71\bin\java ...
分配了10M空间给数组
Xmx=18.0M
free mem=5.6792449951171875M
total mem=16.5M
Process finished with exit code 0
```

如上图红框所示：此时，total mem 为7M时已经不能满足需求了，于是total mem 涨成了16.5M。

保持参数不变，在程序中运行如下代码：（进行一次GC的回收）



```
System.gc();

System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 /
1024 + "M");    //系统的最大空间

System.out.println("free mem=" + Runtime.getRuntime().freeMemory() /
1024.0 / 1024 + "M");    //系统的空闲空间

System.out.println("total mem=" + Runtime.getRuntime().totalMemory() /
1024.0 / 1024 + "M");    //当前可用的总空间
```



运行效果：

```
JavaTest  BubbleSort
C:\Java\jdk1.7.0_71\bin\java ...
Xmx=18.0M
free mem=6.245208740234375M
total mem=7.0M
Process finished with exit code 0
```

问题1: -Xmx (最大堆空间) 和 -Xms (最小堆空间) 应该保持一个什么关系, 可以让系统的性能尽可能的好呢?

问题2: 如果你要做一个Java的桌面产品, 需要绑定JRE, 但是JRE又很大, 你如何做一下JRE的瘦身呢?

2、-Xmn、-XX:NewRatio、-XX:SurvivorRatio:

- -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

例如: 4, 表示新生代:老年代=1:4, 即新生代占整个堆的1/5

- -XX:SurvivorRatio (幸存代)

设置两个Survivor区和eden的比值

例如: 8, 表示两个Survivor:eden=2:8, 即一个Survivor占年轻代的1/10

现在运行如下这段代码:

```

public class JavaTest {
    public static void main(String[] args) {
        byte[] b = null;
        for (int i = 0; i < 10; i++)
            b = new byte[1 * 1024 * 1024];
    }
}

```

我们通过设置不同的jvm参数，来看一下GC日志的区别。

(1) 当参数设置为如下时：（设置新生代为1M，很小）

```
-Xmx20m -Xms20m -Xmn1m -XX:+PrintGCDetails
```

运行效果：

```

JavaTest
C:\Java\jdk1.7.0_71\bin\java ...
Heap
PSYoungGen      total 512K, used 0K [0x00000000fff00000, 0x0000000100000000, 0x0000000100000000)
  eden space 0K, -2147483648% used [0x00000000fff00000, 0x00000000fff00000, 0x00000000fff00000)
    from space 512K, 0% used [0x00000000fff80000, 0x00000000fff80000, 0x0000000100000000)
    to   space 512K, 0% used [0x00000000fff00000, 0x00000000fff00000, 0x00000000fff80000)
ParOldGen       total 19456K, used 11061K [0x0000000fec000000, 0x00000000fff00000, 0x00000000fff00000)
  object space 19456K, 56% used [0x0000000fec000000, 0x00000000fff6cd508, 0x00000000fff00000)
PSPermGen       total 21504K, used 2938K [0x00000000f9a00000, 0x00000000faf00000, 0x00000000fec00000)
  object space 21504K, 13% used [0x00000000f9a00000, 0x00000000f9cdeb98, 0x00000000faf00000)

Process finished with exit code 0

```

总结：

没有触发GC

由于新生代的内存比较小，所以全部分配在老年代。

(2) 当参数设置为如下时：（设置新生代为15M，足够大）

```
-Xmx20m -Xms20m -Xmn15m -XX:+PrintGCDetails
```

运行效果：

```

14
JavaTest
C:\Java\jdk1.7.0_71\bin\java ...
Heap
PSYoungGen      total 13824K, used 12017K [0x00000000ff100000, 0x0000000100000000, 0x0000000100000000)
  eden space 12288K, 97% used [0x00000000ff100000, 0x00000000ffc5a0, 0x00000000ffd00000)
    from space 1536K, 0% used [0x00000000ffe80000, 0x00000000ffe80000, 0x0000000100000000)
    to   space 1536K, 0% used [0x00000000ffd00000, 0x00000000ffd00000, 0x00000000ffe80000)
ParOldGen       total 5120K, used 0K [0x00000000fea00000, 0x00000000fef00000, 0x00000000ff100000)
  object space 5120K, 0% used [0x00000000fea00000, 0x00000000fea00000, 0x00000000fef00000)
PSPermGen       total 21504K, used 2938K [0x00000000f9800000, 0x00000000fad00000, 0x00000000fea00000)
  object space 21504K, 13% used [0x00000000f9800000, 0x00000000f9adeb98, 0x00000000fad00000)

Process finished with exit code 0

```

上图显示：

没有触发GC

全部分配在eden (蓝框所示)

老年代没有使用 (红框所示)

(3) 当参数设置为如下时: (设置新生代为7M, 不大不小)

`-Xmx20m -Xms20m -Xmn7m -XX:+PrintGCDetails`

运行效果:

```
JavaTest
C:\Java\jdk1.7.0_71\bin\java ...
[GC [PSYoungGen: 1178K->504K(6656K)] 1178K->504K(19968K), 0.0015688 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC [PSYoungGen: 504K->0K(6656K)] [ParOldGen: 56K->528K(13312K)] 560K->528K(19968K) [PSPermGen: 2932K->2931K(21504K)], 0.0207332 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [PSYoungGen: 5611K->64K(6656K)] 6140K->1616K(19968K), 0.0008724 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 PSYoungGen      total 6656K, used 5287K [0x00000000ff900000, 0x0000000100000000, 0x0000000100000000)
  eden space 6144K, 85% used [0x00000000ff900000, 0x00000000ffe19fa8, 0x00000000ff900000)
   from space 512K, 12% used [0x00000000fff80000, 0x00000000fff90000, 0x0000000100000000)
   to   space 512K, 0% used [0x00000000fff00000, 0x00000000fff00000, 0x00000000fff80000)
 ParOldGen       total 13312K, used 1552K [0x00000000fec00000, 0x00000000ff900000, 0x00000000ff900000)
  object space 13312K, 11% used [0x00000000fec00000, 0x00000000fad84208, 0x00000000ff900000)
 PSPermGen       total 21504K, used 3107K [0x00000000f9a00000, 0x00000000faf00000, 0x00000000fec00000)
  object space 21504K, 14% used [0x00000000f9a00000, 0x00000000f9d08c78, 0x00000000faf00000)

Process finished with exit code 0
```

总结:

进行了2次新生代GC

s0 s1 太小, 需要老年代担保

(4) 当参数设置为如下时: (设置新生代为7M, 不大不小; 同时, 增加幸存代大小)

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

运行效果:

```
JavaTest
C:\Java\jdk1.7.0_71\bin\java ...
[GC [PSYoungGen: 958K->664K(5632K)] 958K->664K(18944K), 0.0009464 secs] [Times: user=0.00 sys=0.03, real=0.00 secs]
[Full GC [PSYoungGen: 664K->0K(5632K)] [ParOldGen: 0K->526K(13312K)] 664K->526K(18944K) [PSPermGen: 2845K->2844K(21504K)], 0.0087324 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [PSYoungGen: 3399K->1088K(5632K)] 3926K->1614K(18944K), 0.0005779 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4244K->1088K(5632K)] 4770K->1614K(18944K), 0.0004354 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4215K->1088K(5632K)] 4741K->1614K(18944K), 0.0003558 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 PSYoungGen      total 5632K, used 2171K [0x00000000ff900000, 0x0000000100000000, 0x0000000100000000)
  eden space 4096K, 26% used [0x00000000ff900000, 0x00000000ffa0ed80, 0x00000000ffd00000)
   from space 1536K, 70% used [0x00000000ffe80000, 0x00000000fff90010, 0x0000000100000000)
   to   space 1536K, 0% used [0x00000000ffd00000, 0x00000000ffd00000, 0x00000000ffe80000)
 ParOldGen       total 13312K, used 526K [0x00000000fec00000, 0x00000000ff900000, 0x00000000ff900000)
  object space 13312K, 3% used [0x00000000fec00000, 0x00000000fec839c8, 0x00000000ff900000)
 PSPermGen       total 21504K, used 2940K [0x00000000f9a00000, 0x00000000faf00000, 0x00000000fec00000)
  object space 21504K, 13% used [0x00000000f9a00000, 0x00000000f9cdf3b8, 0x00000000faf00000)

Process finished with exit code 0
```

总结:

进行了至少3次新生代GC

s0 s1 增大

(5) 当参数设置为如下时:

`-Xmx20m -Xms20m -XX:NewRatio=1`

`-XX:SurvivorRatio=2 -XX:+PrintGCDetails`

运行效果：

```
Process finished with exit code 0
C:\Java\jdk1.7.0_71\bin\java ...
[GC [PSYoungGen: 1026K->632K(7680K)] 1026K->632K(17920K), 0.0010158 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC [PSYoungGen: 632K->0K(7680K)] [ParOldGen: 0K->526K(10240K)] 632K->526K(17920K) [PSPermGen: 2864K->2864K(21504K)], 0.0087632 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4506K->1056K(7680K)] 5032K->1582K(17920K), 0.0007632 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5256K->1088K(7680K)] 5783K->1614K(17920K), 0.0005614 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 PSYoungGen      total 7680K, used 3256K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
  eden space 5120K, 42% used [0x00000000ff600000, 0x00000000ff81e138, 0x00000000ff800000)
    from space 2560K, 42% used [0x00000000ff800000, 0x00000000ffc10010, 0x00000000ffd80000)
    to   space 2560K, 0% used [0x00000000ffd80000, 0x00000000ffd80000, 0x0000000100000000)
 ParOldGen       total 10240K, used 526K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  object space 10240K, 5% used [0x00000000fec00000, 0x00000000fec83b08, 0x00000000ff600000)
 PSPermGen       total 21504K, used 2940K [0x00000000f9a00000, 0x00000000faf00000, 0x00000000fec00000)
  object space 21504K, 13% used [0x00000000f9a00000, 0x00000000f9cdf3b8, 0x00000000faf00000)
Process finished with exit code 0
```

(6) 当参数设置为如下时： 和上面的 (5) 相比，适当减小幸存代大小，这样的话，能够减少GC的次数

-Xmx20m -Xms20m -XX:NewRatio=1

-XX:SurvivorRatio=3 -XX:+PrintGCDetails

```
Process finished with exit code 0
C:\Java\jdk1.7.0_71\bin\java ...
[GC [PSYoungGen: 1055K->600K(8192K)] 1055K->600K(18432K), 0.0013448 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC [PSYoungGen: 600K->0K(8192K)] [ParOldGen: 0K->526K(10240K)] 600K->526K(18432K) [PSPermGen: 2847K->2846K(21504K)], 0.02659 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [PSYoungGen: 5611K->1120K(8192K)] 6138K->1646K(18432K), 0.0091442 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 8192K, used 6427K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
  eden space 6144K, 86% used [0x00000000ff600000, 0x00000000ffb2ed38, 0x00000000ffc00000)
    from space 2048K, 54% used [0x00000000ffc00000, 0x00000000fff18010, 0x0000000100000000)
    to   space 2048K, 0% used [0x00000000fff18010, 0x00000000fff18010, 0x00000000ffe00000)
 ParOldGen       total 10240K, used 526K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  object space 10240K, 5% used [0x00000000fec00000, 0x00000000fec83ad8, 0x00000000ff600000)
 PSPermGen       total 21504K, used 2940K [0x00000000f9a00000, 0x00000000faf00000, 0x00000000fec00000)
  object space 21504K, 13% used [0x00000000f9a00000, 0x00000000f9cdf3b8, 0x00000000faf00000)
Process finished with exit code 0
```

3、-XX:+HeapDumpOnOutOfMemoryError、-XX:+HeapDumpPath

- -XX:+HeapDumpOnOutOfMemoryError

OOM时导出堆到文件

根据这个文件，我们可以看到系统dump时发生了什么。

- -XX:+HeapDumpPath

导出OOM的路径

例如我们设置如下的参数：

-Xmx20m -Xms5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump

上方意思是说，现在给堆内存最多分配20M的空间。如果发生了OOM异常，那就把dump信息导出到d:/a.dump文件中。

然后，我们执行如下代码：

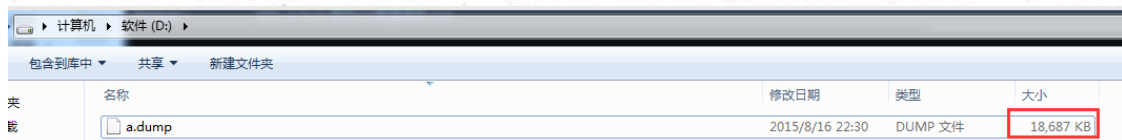
```
Vector v = new Vector();
for (int i = 0; i < 25; i++)
    v.add(new byte[1 * 1024 * 1024]);
```

上方代码中，需要利用25M的空间，很显然会发生OOM异常。现在我们运行程序，控制台打印如下：

```
JavaTest
C:\Java\jdk1.7.0_71\bin\java ...
java.lang.OutOfMemoryError: Java heap space
Dumping heap to d:/a.dump ...
Heap dump file created [19135003 bytes in 0.059 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at test03.JavaTest.main(JavaTest.java:12) <5 internal calls>

Process finished with exit code 1
```

现在我们去D盘看一下dump文件：



名称	修改日期	类型	大小
a.dump	2015/8/16 22:30	DUMP 文件	18,687 KB

上图显示，一般来说，这个文件的大小和最大堆的大小保持一致。

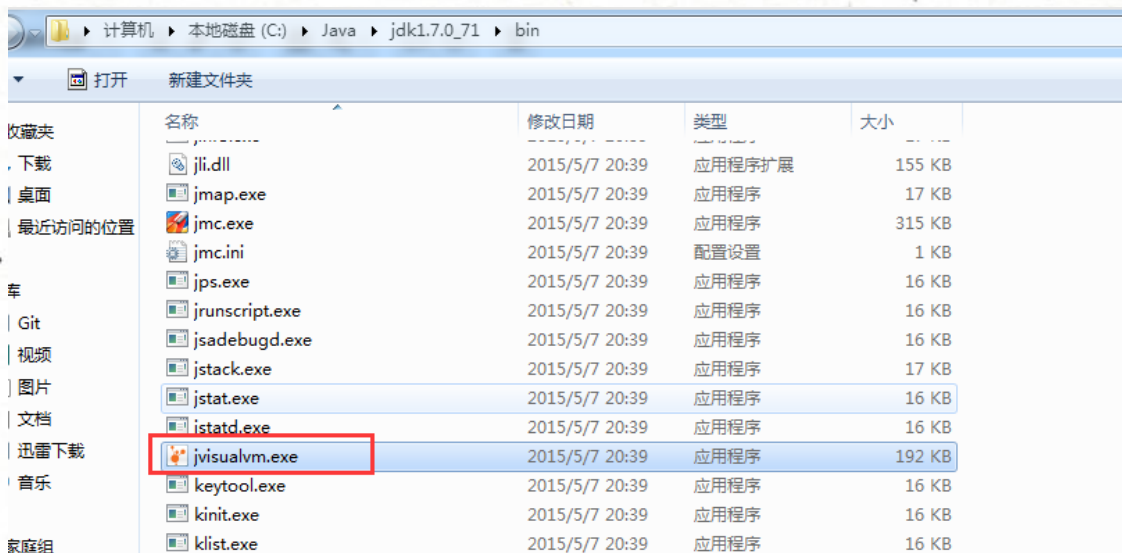
我们可以用VisualVM打开这个dump文件。

注：关于VisualVM的使用，可以参考下面这篇博客：

使用 VisualVM 进行性能分析及调

优：<http://www.ibm.com/developerworks/cn/java/j-lo-visualvm/>

或者使用Java自带的Java VisualVM工具也行：



名称	修改日期	类型	大小
jli.dll	2015/5/7 20:39	应用程序扩展	155 KB
jmap.exe	2015/5/7 20:39	应用程序	17 KB
jmc.exe	2015/5/7 20:39	应用程序	315 KB
jmc.ini	2015/5/7 20:39	配置设置	1 KB
jps.exe	2015/5/7 20:39	应用程序	16 KB
jrunscript.exe	2015/5/7 20:39	应用程序	16 KB
jsadefbugd.exe	2015/5/7 20:39	应用程序	16 KB
jstack.exe	2015/5/7 20:39	应用程序	17 KB
jstat.exe	2015/5/7 20:39	应用程序	16 KB
jstatd.exe	2015/5/7 20:39	应用程序	16 KB
jvisualvm.exe	2015/5/7 20:39	应用程序	192 KB
keytool.exe	2015/5/7 20:39	应用程序	16 KB
kinit.exe	2015/5/7 20:39	应用程序	16 KB
klist.exe	2015/5/7 20:39	应用程序	16 KB

java.util.Vector @ 0x3395c490	24	18,874,776
<class> class java.util.Vector @ 0x3963	16	16
elementData java.lang.Object[20] @ 0	96	18,874,752
<class> class java.lang.Object[] @ 0	0	0
[13] byte[1048576] @ 0x33280000	1,048,592	1,048,592
[14] byte[1048576] @ 0x33380010	1,048,592	1,048,592
[15] byte[1048576] @ 0x33480020	1,048,592	1,048,592
[16] byte[1048576] @ 0x33580030	1,048,592	1,048,592
[17] byte[1048576] @ 0x33680040	1,048,592	1,048,592
[0] byte[1048576] @ 0x3397acf0 ...	1,048,592	1,048,592
[1] byte[1048576] @ 0x33a7b3b0 ..	1,048,592	1,048,592
[2] byte[1048576] @ 0x33b7b3c0 ..	1,048,592	1,048,592
[3] byte[1048576] @ 0x33c7b3d0 ..	1,048,592	1,048,592
[4] byte[1048576] @ 0x33d7b3e0 ..	1,048,592	1,048,592
[5] byte[1048576] @ 0x33e7b3f0 ...	1,048,592	1,048,592
[6] byte[1048576] @ 0x33f7b400 ...	1,048,592	1,048,592
[7] byte[1048576] @ 0x3407b498 ..	1,048,592	1,048,592
[8] byte[1048576] @ 0x3417b4a8 ..	1,048,592	1,048,592
[9] byte[1048576] @ 0x3427b4b8 ..	1,048,592	1,048,592
[10] byte[1048576] @ 0x3437b4c8	1,048,592	1,048,592
[11] byte[1048576] @ 0x3447b4d8	1,048,592	1,048,592
[12] byte[1048576] @ 0x3457b548	1,048,592	1,048,592
Σ Total: 19 entries		
Σ Total: 2 entries		

上图中就是dump出来的文件，文件中可以看到，一共有19个byte已经被分配了。

4、-XX:OnOutOfMemoryError:

- -XX:OnOutOfMemoryError

在OOM时，执行一个脚本。

可以在OOM时，发送邮件，甚至是重启程序。

例如我们设置如下的参数：

-XX:OnOutOfMemoryError=D:/tools/jdk1.7_40/bin/printstack.bat %p //p代表的
是当前进程的pid

上方参数的意思是说，执行printstack.bat脚本，而这个脚本做的事情是：

D:/tools/jdk1.7_40/bin/jstack -F %1 > D:/a.txt，即当程序OOM时，在
D:/a.txt中将会生成线程的dump。

5、堆的分配参数总结：

- 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

6、永久区分配参数：

- -XX:PermSize -XX:MaxPermSize

设置永久区的初始空间和最大空间。也就是说，jvm启动时，永久区一开始就占用了PermSize大小的空间，如果空间还不够，可以继续扩展，但是不能超过MaxPermSize，否则会OOM。

他们表示，一个系统可以容纳多少个类型

代码举例：

我们知道，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。于是，我们运行下面这段代码：

```
for(int i=0;i<100000;i++){
    CglibBean bean = new CglibBean("geym.jvm.ch3.perm.bean"+i,new
    HashMap());
}
```

上面这段代码会在永久区不断地产生新的类。于是，运行效果如下：

```
[Full GC(TenuredException in thread "main" : 2275K->2276K(10944K), 0.0126589 secs) 2367K->2276K(15936K), [Perm : 4095K->4095K(4096K)],
[Full GC(Tenured: 2276K->2276K(10944K), 0.0126801 secs) 2276K->2276K(15936K), [Perm : 4095K->4095K(4096K)], 0.0127427 secs] [Times: use
Heap
def new generation      total 4992K, used 89K [0x28060000, 0x285c0000, 0x2d5b0000)
eden space 4480K,       2% used [0x28060000, 0x280766d0, 0x284c0000)
from space 512K,        0% used [0x284c0000, 0x284c0000, 0x28540000)
to space 512K,          0% used [0x28540000, 0x28540000, 0x285c0000)
tenured generation      total 10944K, used 2276K [0x2d5b0000, 0x2e060000, 0x38060000)
the space 10944K,       20% used [0x2d5b0000, 0x2d7e90c0, 0x2d7e9200, 0x2e060000)
compacting perm gen      total 4096K, used 4095K [0x38060000, 0x38460000, 0x38460000)
the space 4096K,        99% used [0x38060000, 0x3845fed0, 0x38460000, 0x38460000)
ro space 10240K,        44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
rw space 12288K,        52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main"
```

总结：

如果堆空间没有用完也抛出了OOM，有可能是永久区导致的。

堆空间实际占用非常少，但是永久区溢出 一样抛出OOM。

三、栈的分配参数：

1、Xss：

设置栈空间的大小。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

注：栈空间是每个线程私有的区域。栈里面的主要内容是栈帧，而栈帧存放的是局部变量表，局部变量表的内容是：局部变量、参数。

我们来看下面这段代码：（没有出口的递归调用）



```
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();
        }
    }
}

```



上方这段代码是没有出口的递归调用，肯定会出现OOM的。

如果设置栈大小为128k:

-Xss128K

运行效果如下：（方法被调用了294次）

```

C:\Java\jdk1.7.0_71\bin\java ...
deep of calling = 294
java.lang.StackOverflowError

```

如果设置栈大小为256k：（方法被调用748次）

```

C:\Java\jdk1.7.0_71\bin\java ...
java.lang.StackOverflowError
deep of calling = 748
at test03.TestStackDeep.recursion(TestStackDeep.java:10)
at test03.TestStackDeep.recursion(TestStackDeep.java:12)

```

意味着函数调用的次数太深，像这种递归调用就是个典型的例子。

总结：

我们在本文中介绍了jvm的一些最基本的参数，还有很多参数（如GC参数等）将在后续的系列文章中进行介绍。我们将在接下来的文章中介绍GC算法。