GC问题排查

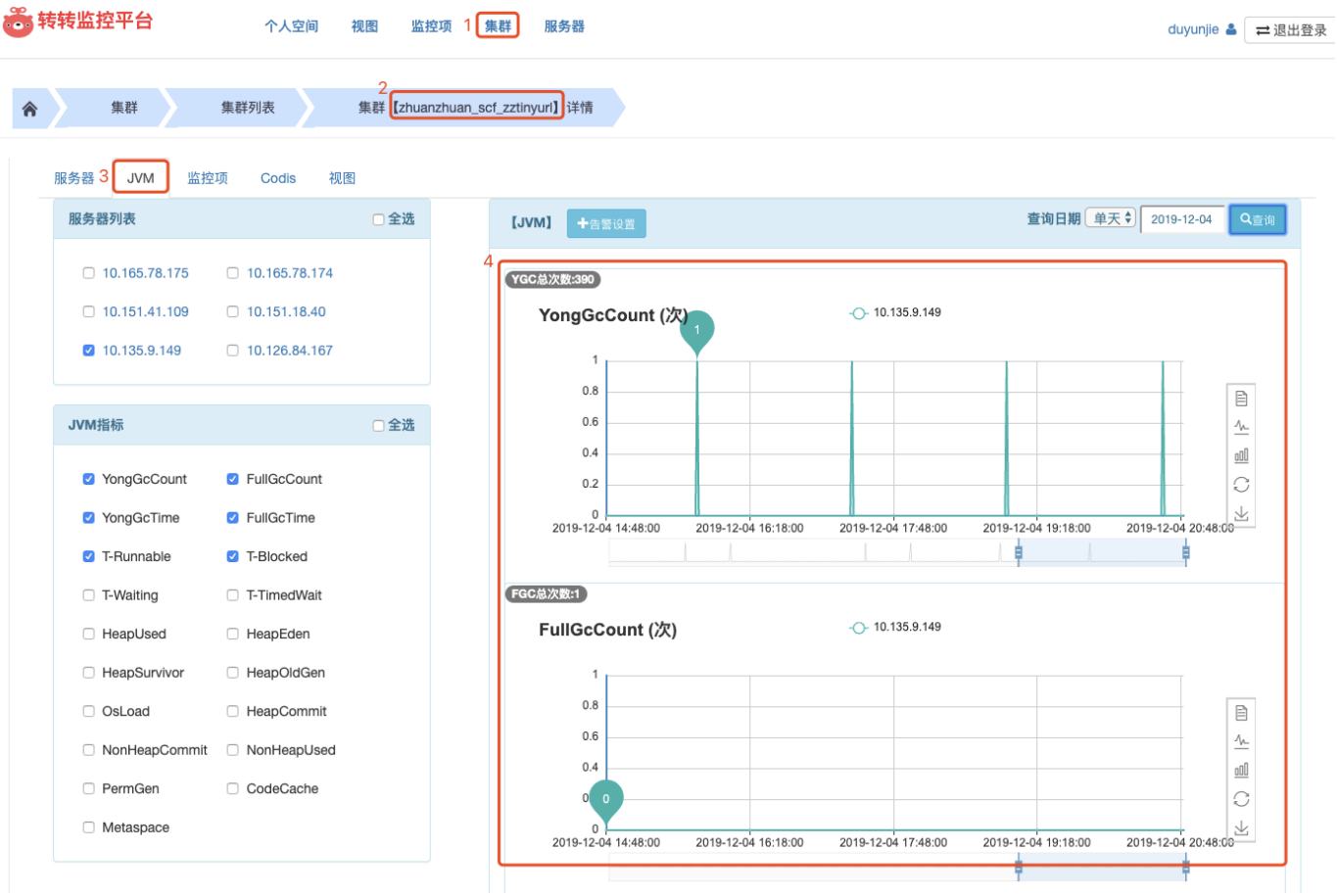
一、背景

线上Java服务偶尔会遇到内存回收不掉而频繁GC的问题，严重影响服务质量。那么遇到这种问题时，该如何排查？本文将就这个问题做一个简单的系统性分析。

二、如何排查

2.1 zzmonitor

目前大部分业务都接入了zzmonitor，zzmonitor会默认收集集群各实例的JVM监控信息。在zzmonitor平台，我们就可以看到这些实例的GC监控信息，并对GC进行报警设置。



2.2 jstat

登上服务器，通过jstat查看实时gc情况。

2.2.1 jstat -gc

查看各内存区域中对象的占用大小（单位：KB）。

```
#31514ID
jstat -gc -h20 31514 1000


#
jstat -gc -h20 31514 1000 | awk '{for(i=1;i<=NF;i++){printf("%10s",$i)}
printf("\n")}'
```

```
^C[work(duyunjie)@vm-bjdhj-153-28 springbootdemo]$ jstat -gc -h20 7276 1000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
43648.0 43648.0 43598.0  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  801    519.792  521.946
43648.0 43648.0 43604.1  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  801    519.792  521.946
43648.0 43648.0 43604.3  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  802    519.792  521.946
43648.0 43648.0 43604.5  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  803    520.265  522.419
43648.0 43648.0 43604.9  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  803    520.265  522.419
43648.0 43648.0 43605.1  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  804    520.265  522.419
43648.0 43648.0 43611.1  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  805    520.711  522.865
43648.0 43648.0 43611.5  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  805    520.711  522.865
43648.0 43648.0 43611.6  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  806    520.948  523.102
43648.0 43648.0 43611.9  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  807    521.579  523.733
43648.0 43648.0 43612.2  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  807    521.579  523.733
43648.0 43648.0 43618.2  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  808    521.579  523.733
43648.0 43648.0 43618.3  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  809    521.804  523.958
43648.0 43648.0 43618.6  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  809    522.429  524.583
43648.0 43648.0 43618.9  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  809    522.429  524.583
43648.0 43648.0 43619.1  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  811    522.676  524.830
43648.0 43648.0 43625.2  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  811    523.182  525.336
43648.0 43648.0 43625.4  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  811    523.182  525.336
43648.0 43648.0 43625.6  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  813    523.405  525.559
43648.0 43648.0 43626.0  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  813    523.949  526.103
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
43648.0 43648.0 43626.2  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  813    523.949  526.103
43648.0 43648.0 43632.2  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  814    524.180  526.334
43648.0 43648.0 43632.5  0.0   174848.0 174848.0  262144.0   262144.0  61440.0 58296.5 8192.0 7580.4     26    2.154  815    524.837  526.992
```

参数	描述
SOC	S0的容量（KB）
S1C	S1的容量（KB）
S0U	S0目前已使用空间（KB）
S1U	S1目前已使用空间（KB）
EC	Eden区的容量（KB）
EU	Eden区目前已使用空间（KB）
OC	Old区的容量（KB）
OU	Old区目前已使用空间（KB）
MC	元数据区的容量（KB）
MU	元数据区目前已使用空间（KB）
CCSC	压缩类的容量（KB）
CCSU	压缩类目前已使用空间（KB）
YGC	YGC总次数
YGCT	YGC所用总时间(s)
FGC	FGC总次数
FGCT	FGC所用总时间(s)
GCT	YGC、FGC所用总时间之和

2.2.2 jstat -gcutil

查看内存各区域的使用率。

```
#31514ID
jstat -gcutil -h20 31514 1000
```

```
[work(duyunjie)@vm-bjdhj-153-28 springbootdemo]$ jstat -gcutil -h20 7276 1000
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
100.00   0.00  66.17  90.18  95.07  92.85     16    2.154     18    4.191    6.344
100.00   0.00  68.31  90.18  95.07  92.85     16    2.154     20    4.339    6.493
100.00   0.00  70.46  90.18  95.07  92.85     16    2.154     20    4.730    6.883
100.00   0.00  72.64  90.18  95.07  92.85     16    2.154     20    4.730    6.883
100.00   0.00  74.79  90.18  95.07  92.85     16    2.154     22    4.924    7.078
100.00   0.00  76.93  90.18  95.07  92.85     16    2.154     22    5.279    7.433
100.00   0.00  78.01  90.18  95.07  92.85     16    2.154     22    5.279    7.433
100.00   0.00  79.34  90.18  95.07  92.85     17    2.154     24    5.460    7.614
  0.00   0.00  88.09 100.00  94.79  92.50     17    2.154     24    6.390    8.544
  0.00   0.00  90.90 100.00  94.79  92.50     17    2.154     24    6.390    8.544
  0.00   0.00  92.27 100.00  94.79  92.50     17    2.154     25    6.591    8.744
  0.00   0.00  93.80  86.25  94.79  92.50     17    2.154     26    7.115    9.269
  0.00   0.00  96.64  86.25  94.79  92.50     17    2.154     26    7.115    9.269
  0.00   0.00  98.08  86.25  94.79  92.50     17    2.154     27    7.115    9.269
  0.00   0.00  99.45  86.25  94.79  92.50     17    2.154     28    7.750    9.904
  3.83   0.00 100.00  86.25  94.79  92.50     17    2.154     28    7.750    9.904
 11.00   0.00 100.00  86.25  94.79  92.50     17    2.154     28    7.750    9.904
 11.01   0.00 100.00  86.25  94.79  92.50     17    2.154     30    7.945   10.099
 18.18   0.00 100.00  86.25  94.79  92.50     17    2.154     30    8.578   10.732
 25.36   0.00 100.00  86.25  94.79  92.50     17    2.154     30    8.578   10.732
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 32.53   0.00 100.00  86.25  94.79  92.50     17    2.154     32    8.976   11.130
 39.71   0.00 100.00  86.25  94.79  92.50     17    2.154     32    8.976   11.130
 46.89   0.00 100.00  86.25  94.79  92.50     17    2.154     33    8.976   11.130
```

参数	描述
S0	S0空间使用率
S1	S1空间使用率
E	Eden区空间使用率
O	Old区空间使用率
M	元数据区空间使用率
CCS	压缩类空间使用率
YGC	YGC总次数
YGCT	YGC所用总时间(s)
FGC	FGC总次数
FGCT	FGC所用总时间(s)
GCT	YGC、FGC所用总时间之和

2.2.3 总结

- **如果频繁YGC但很少FGC，那说明JVM在大量创建临时对象，但这些对象很快又被回收掉了；**
- **如果Old区使用率一直很高又频繁FGC，那基本说明有内存泄漏了。**

2.3 GC日志

我们也可以把GC日志保存下来，以便分析，其JVM参数如下：

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:/opt
/scf/log/zzsearch/gc_%p.log
```

具体分析可参考：

- CMS：https://www.cnblogs.com/zhangxiaoguang/p/5792468.html

2.4 确定造成内存泄漏的对象

通过上面的分析，我们知道当前JVM有内存泄漏。那么如何确定是哪个类的对象造成的呢？主要有两种：

- jmap -histo快速查看占用空间最多的类对象；
- jmap dump内存并用MAT进行分析。

2.4.1 jmap -histo

```
#7276ID
jmap -histo 7276|head -n20
```

```
[work(duyunjie)@vm-bjdhj-153-28 springbootdemo]$ jmap -histo 7276|head -n20
 排名     实例数         占用空间     类名
 num     #instances         #bytes  class name
----------------------------------------------
   1:      12542174      401349568  com.bj58.zhuanzhuan.springbootdemo.model.User
   2:         12631       57271280  [Ljava.lang.Object;
   3:         69388       10119080  [C
   4:        262144        6291456  org.apache.logging.log4j.core.async.AsyncLoggerConfigDisruptor$Log4jEventWrapper
   5:          2284        2517840  [B
   6:         68193        1636632  java.lang.String
   7:          4848        1607944  [I
   8:         12311        1366288  java.lang.Class
   9:         35249        1127968  java.util.concurrent.ConcurrentHashMap$Node
  10:          5744         505472  java.lang.reflect.Method
  11:         11334         453360  java.util.LinkedHashMap$Entry
  12:         28228         451648  java.lang.Object
  13:          5733         437792  [Ljava.util.HashMap$Node;
  14:         13512         432384  java.util.HashMap$Node
  15:           505         406432  [Ljava.util.concurrent.ConcurrentHashMap$Node;
  16:         12600         403200  java.lang.ref.WeakReference
  17:          6071         339976  java.util.LinkedHashMap
[work(duyunjie)@vm-bjdhj-153-28 springbootdemo]$
```

2.4.2 内存dump & MAT分析

我们也可以通过jmap把内存dump下来，并进行更全面的分析。

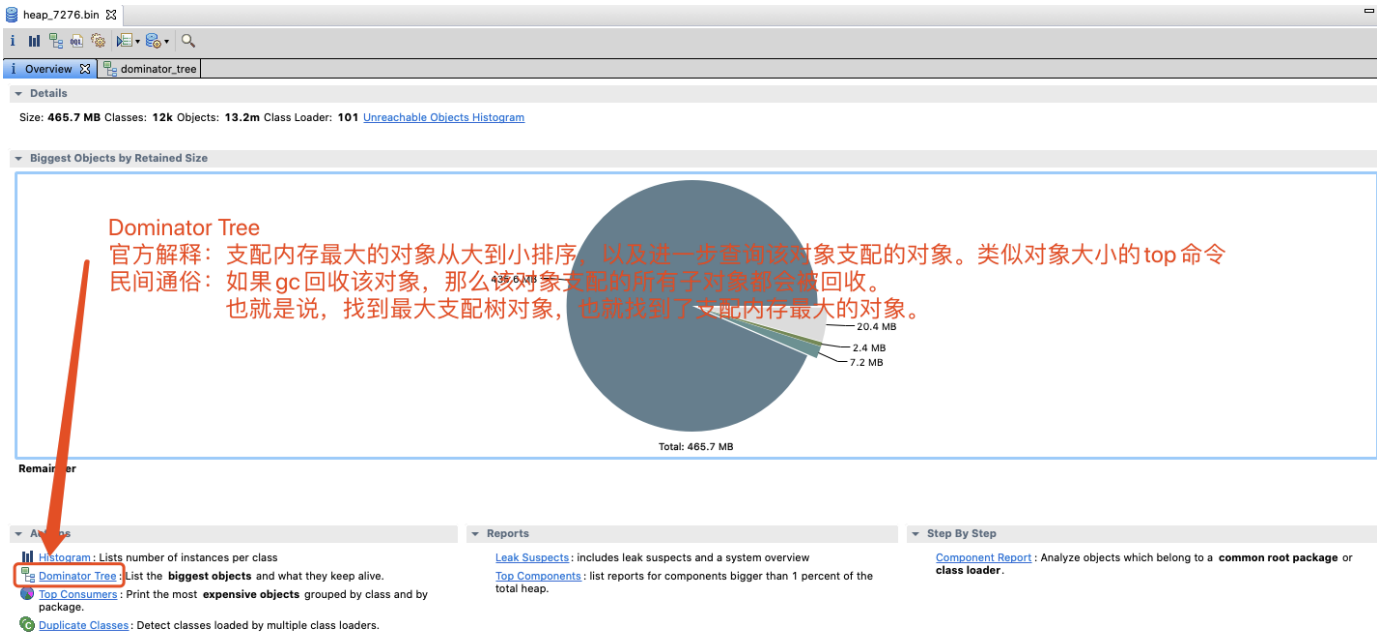
2.4.2.1 dump内存

```
jmap -dump:live,format=b,file=/tmp/heap_7276.bin 7276
```

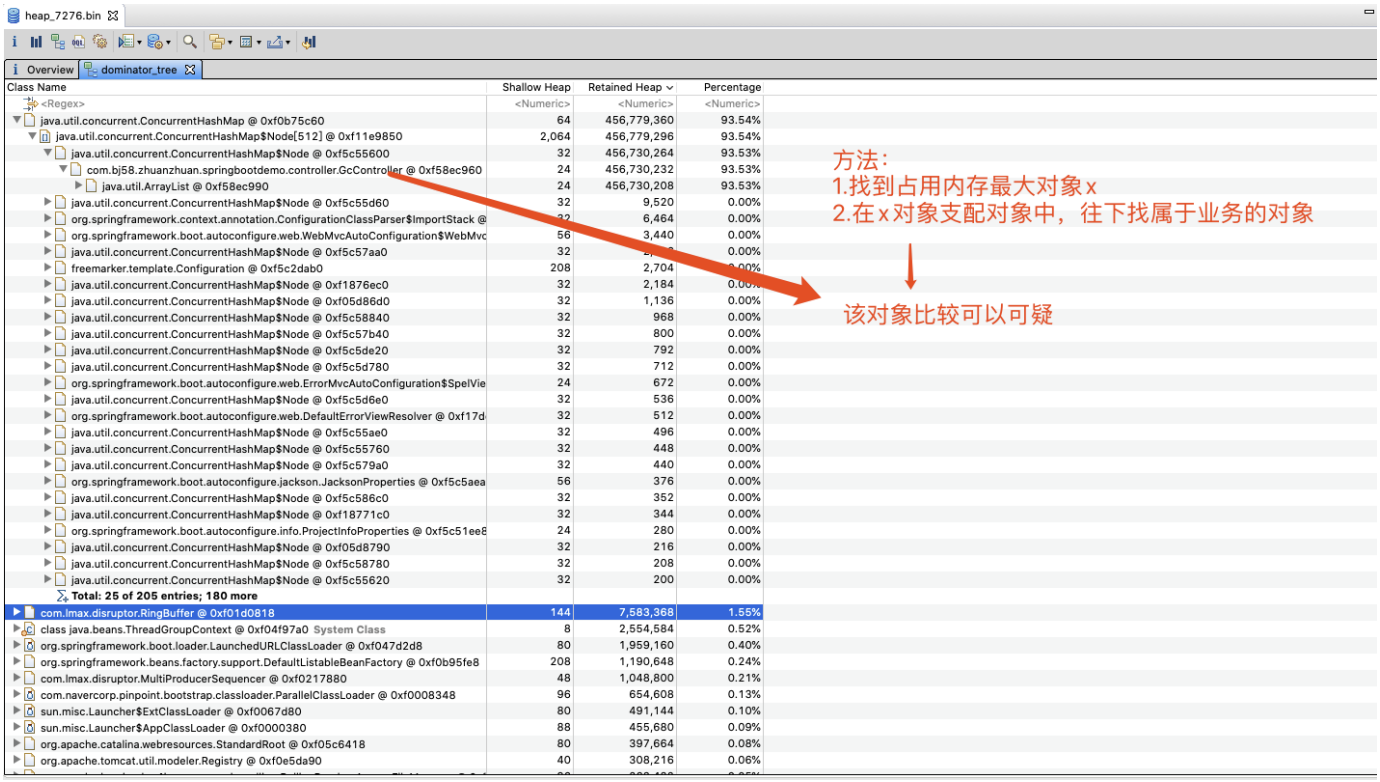
2.4.2.2 MAT分析

一招找出支配内存最大的对象，mat***Dominator Tree*** ：

Step 1 ：打开Dominator Tree



Step 2 ：分析支配内存最大对象



三、总结

出现内存泄漏，一般都是**因为长生命周期的对象引用了短生命周期的对象，从而造成短生命周期的对象也回收不掉**。所以我们在编码时一定要注意所创建对象的生命周期。

总结一下，遇到GC问题时，常规做法是：

1. **先通过zzmonitor、jstat确定是否内存泄漏**；
2. **再通过jstat dump内存，并使用MAT分析造成泄漏的类对象**。

本文针对常见GC问题的排查过程，作了一个简单实用的分析。在排查过程中如有问题，欢迎联系@梁会彬、@杜云杰。

四、FQA

4.1 常用的推荐JVM配置有哪些？

4.1.1 垃圾回收器

```
#CMS
-Xms2g -Xmx2g -Xmn1g -XX:MetaspaceSize=256m -XX:MaxMetaspaceSize=256m -XX:
+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -
XX:CMSInitiatingOccupancyFraction=80

#G1
-Xms2g -Xmx2g -XX:MetaspaceSize=256m -XX:MaxMetaspaceSize=256m -XX:
+UseG1GC -XX:MaxGCPauseMillis=100
```

4.1.2 GC日志

```
#%pID%tYYYY-MM-DD_HH-MM-SS.(gc_34299_2019-12-06_16-43-26.log)
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:/opt
/scf/log/zzproduct/gc_%p_%t.log
```

4.1.3 OOM Dump

```
#%pID%tYYYY-MM-DD_HH-MM-SS.(dump_34299_2019-12-06_16-43-26.log.hprof)
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/scf/log/zzproduct
/dump_%p_%t.log.hprof
```

4.2 为什么推荐Xms和Xmx配置成相等？

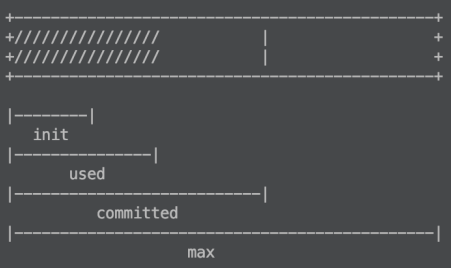
要解释Xms和Xmx相等，需要引入一些概念，init、used、committed、max。下图是jdk源码中的java.lang.management.MemoryUsage给我们做了很好的权威解释。

java.lang.management
public class **MemoryUsage**
extends Object

A MemoryUsage object represents a snapshot of memory usage. Instances of the MemoryUsage class are usually constructed by methods that are used to obtain memory usage information about individual memory pool of the Java virtual machine or the heap or non-heap memory of the Java virtual machine as a whole.
A MemoryUsage object contains four values:

init	represents the initial amount of memory (in bytes) that the Java virtual machine requests from the operating system for memory management during startup. The Java virtual machine may request additional memory from the operating system and may also release memory to the system over time. The value of init may be undefined.
used	represents the amount of memory currently used (in bytes).
committed	represents the amount of memory (in bytes) that is guaranteed to be available for use by the Java virtual machine. The amount of committed memory may change over time (increase or decrease). The Java virtual machine may release memory to the system and committed could be less than init. committed will always be greater than or equal to used.
max	represents the maximum amount of memory (in bytes) that can be used for memory management. Its value may be undefined. The maximum amount of memory may change over time if defined. The amount of used and committed memory will always be less than or equal to max if max is defined. A memory allocation may fail if it attempts to increase the used memory such that used > committed even if used <= max would still be true (for example, when the system is low on virtual memory).

Below is a picture showing an example of a memory pool:



MXBean Mapping

MemoryUsage is mapped to a CompositeData with attributes as specified in the from method.
Since: 1.5

而Xms对应的是init的值，Xmx对应的是max的值。从上图可知，如果init<max那么java的堆空间是一个动态变化的值！！！也就是根据需求，自动伸缩内存。

Xms<Xmx

好处：节省内存、提高资源利用率。

坏处：如果初始化内存设置较小，那么java进程启动的时候随着业务请求的增多，内存很快被占用完毕，触发fullgc，进而commited增大，然后又内存又被消耗完毕，形成内存消耗➡gc和增大committed的循环，**导致进程启动后的一段时间频繁fullgc**，直到committed到达一个稳定值为止。

线上case：

jvm参数：-Xms1g -Xmx2g -Xmn1g

解释：对应init=1g max=2g-Xmn新生代1g，那么jvm初始化的内存都分配给新生代了，老年代是不是就是0呢？带着疑问看一下jvm关于老年代初始化的代码逻辑(代码引用自genArguments.cpp)

结论：由于Xms和Xmx设置不一致，又指定了新生代大小，jvm默认老年代OldSize比较小，导致启动后一段时间频繁fullgc的悲剧

老年代初始化值

```
if (!FLAG_IS_CMDLINE(OldSize)) { //jvm-XX:OldSize
  MinOldSize = GenAlignment;     //6553664kfullgc
  initial_old_size = MIN2(MaxOldSize, MAX2(InitialHeapSize -
initial_young_size, MinOldSize));
                    //InitialHeapSizeXms initial_young_sizeXmn,old_size
65536
}
```

Xms==Xmx 可以很好的避免jvm内存伸缩问题，正常情况下也就不会遇见进程启动是频繁fullgc的问题。

综上所述：

在当下内存成本降低的情况下，把Xms和Xmx值设置成相等，可以很好的避免因jvm内存伸缩而带来的诡异的fullgc问题。

end