# 前言

实战jvm调优，其中参数启动参数是：

java -jar -Xms1536M -Xmx1536M -Xmn512M -Xss256K -XX:SurvivorRatio=6 -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly **jvm-full-gc-0.0.1-SNAPSHOT.jar**

java -jar -Xms1536M -Xmx1536M **-Xmn1024M** -Xss256K -XX:SurvivorRatio=6 -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction**=92** -XX:+UseCMSInitiatingOccupancyOnly **jvm-full-gc-0.0.3-SNAPSHOT.jar**

# 系统分析

## 5机器配置

* 2核4G
* JVM内存大小：2G

## **查看java启动参数(jinfo -flags pid)**

**Command line配置参数:**

**-Xms1536M** --最小堆内存

**-Xmx1536M** --最大堆内存

**-Xmn512M** --年轻代内存

**-Xss256K**  --线程栈内存

**-XX:SurvivorRatio=6**  --年轻代分配比例

**-XX:MetaspaceSize=256M**  -元空间

**-XX:MaxMetaspaceSize=256M** -元空间

**-XX:+UseParNewGC**  -新生代使用ParNewGC

**-XX:+UseConcMarkSweepGC** --老年代用CMS

**-XX:CMSInitiatingOccupancyFraction=75** --老年代超过75%就要gc

**-XX:+UseCMSInitiatingOccupancyOnly** --该参数启用后，参数CMSInitiatingOccupancyFraction才会生效。默认关闭。

**Non-default VM flags 其他默认参数：**

**-XX:CICompilerCount=3**  --jvm处理编译队列的总线程数，设置的相对较大可以一定程度提升JIT编译的速度，默认为3

**-XX:CompressedClassSpaceSize=260046848** --compressed class space(存放Klass对象)空间的大小

**-XX:InitialHeapSize=1610612736**  --初始化堆内存

**-XX:MaxHeapSize=1610612736**  --最大堆内存

**-XX:MaxNewSize=536870912**  -最大新生代内存

**-XX:MinHeapDeltaBytes=196608** --为了防止频繁扩展内存代空间,每次扩展内存代时都有一个最小值,其默认值为128KB

**-XX:NewSize=536870912** - -新生代内存

**-XX:OldPLABSize=16** - -默认值为1024，指老生代PLAB缓存大小。在32位JVM中PLAB为4KB，64位JVM中为8KB，表示对象从Eden复制到Old时，每次请求4KB作为分配缓存，提高分配效率。增大该值可以提高分配的效率，但是可能增加内存碎片；通常来说Old分区空间更大，实际调优中可以尝试先增大该值。

**-XX:OldSize=1073741824**  --老生代堆空间的默认值

**-XX:ThreadStackSize=256**  --线程栈内存

**-XX:+UseCompressedClassPointers** --表示是否启用类指针压缩，因为对于任何一个jvm中的对象而言，其内部都有一个指向自己对应类(属于哪个class)的指针(Java习惯叫引用)，在64位的Java虚拟机中，默认是启动压缩的；

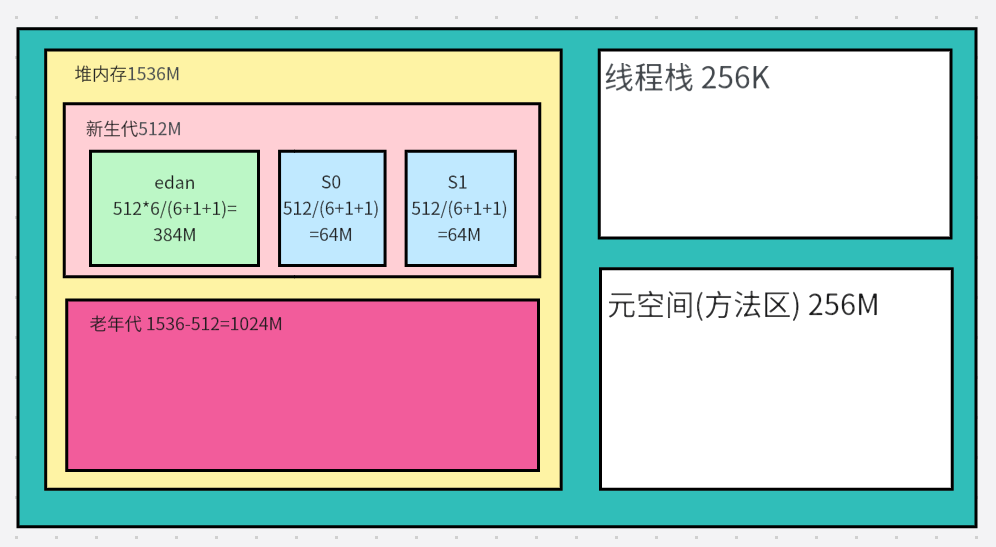
**-XX:+UseCompressedOops**  --表示是否使用普通对象指针压缩，Oops是Ordinary object pointers的缩写，就是任何指向一个在堆中的对象(非简单类型)的指针，默认也是启动压缩的。

**-XX:+UseFastUnorderedTimeStamps** --是 JDK 8 中引入的一个 JVM 参数，用于控制 JVM 如何获取时间戳。默认情况下，JVM 使用有序的时间戳，即每个时间戳都大于前面一个时间戳，但这种方法可能会带来一些性能开销。

开启 UseFastUnorderedTimeStamps 后，JVM 将使用无序的时间戳，也就是说，时间戳的顺序可能与它们实际产生的顺序不同。这种无序的时间戳可以提高一些多线程场景下的性能，例如 Actor 模型等。

需要注意的是，使用无序的时间戳可能会影响某些应用程序的正确性，因此在使用该参数前请仔细评估其对应用程序的潜在影响。

## **整理jvm图**



## **查看程序开机时间**

通过ps -p pid -o lstart查看，得知系统运行时间：**7天**

## **查看程序GC概况 (**jstat -gc pid**)**

### YoungGC总结

YoungGC次数(**YGC**)和耗时(**YGCT**)：**1万多次，500多秒。**意思是每天会发生**10000/7=1400**多次Young GC，每分钟会发生**10000/7/24/60=1**次，每次Young GC在**500\*1000/10000=50**毫秒左右。

### **触发Young GC条件**

1. 每一分钟就有64M对象挤爆Survivor区。
2. 触发对象动态年龄判断，意思是在Young GC后Survivor区依然超过50%(可配置)，假设第N次为临界线，那么N及N+1次的存活对象再触发一次YGC然后都加入老年代(4.1更详细)

### FullGC总结

FullGC次数(**FGC**)和耗时(**FGCT**)：**500多次，200多秒**，意思是每天会发生**500/7=70**多次Full GC，平均每小时**500/7/24=3**次，每次Full GC在**200\*1000/500=400**毫秒左右。

### **触发Young GC条件**

1. 每20分钟就有1024M对象挤爆老年区。
2. 触发老年代空间分配担保机制。意思是YGC后年轻代存活对象大于老年代可用空间，就会触发fullGC。(4.2更详细)

# 模拟优化

## 模拟高负载

通过模拟web调用来耗损系统资源，通过 jstat -gc pid查看发现频繁Young GC和Full gc。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S0C | S1C | S0U | S1U | EC | EU | OC | OU | MC | MU | CCSC | CCSU | YGC | YGCT | FGC | FGCT | GCT |
| 65536 | 65536 | 0 | 65536 | 393216 | 75112.5 | 1048576 | 380483.7 | 37504 | 35557.8 | 4992 | 4560 | 31 | 27.824 | 16 | 1.464 | 29.288 |
| 65536 | 65536 | 0 | 0 | 393216 | 301603.2 | 1048576 | 217799.1 | 37504 | 35589.5 | 4992 | 4560 | 33 | 27.926 | 17 | 1.584 | 29.511 |
| 65536 | 65536 | 65536 | 0 | 393216 | 22729.2 | 1048576 | 931921.3 | 37504 | 35590.7 | 4992 | 4560 | 35 | 28.104 | 18 | 1.585 | 29.690 |
| 65536 | 65536 | 0 | 65536 | 393216 | 250827 | 1048576 | 703996.3 | 37504 | 35591.1 | 4992 | 4560 | 37 | 28.190 | 19 | 1.813 | 3003 |
| 65536 | 65536 | 65536 | 0 | 393216 | 393128.5 | 1048576 | 270530.9 | 37504 | 35591.9 | 4992 | 4560 | 39 | 28.190 | 20 | 1.901 | 3091 |
| 65536 | 65536 | 0 | 0 | 393216 | 205970.8 | 1048576 | 316843.2 | 37504 | 35593.2 | 4992 | 4560 | 41 | 28.387 | 22 | 222 | 30.409 |

## 优化step1

### 优化参数

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 优化前 | 优化后 | 备注 |
| Xms | 1536M | 1536M |  |
| Xmx | 1536M | 1536M |  |
| Xmn | 512M | 1024M | 新生代扩张大1G |
| Xss | 256K | 256K |  |
| SurvivorRatio | 6 | 6 |  |
| MetaspaceSize | 256m | 256m |  |
| MaxMetaspaceSize | 256M | 256M |  |
| UseParNewGC |  |  |  |
| UseConcMarkSweepGC |  |  |  |
| CMSInitiatingOccupancyFraction | 75 | 92 | 老年代超过92%就要gc |
| UseCMSInitiatingOccupancyOnly |  |  |  |

### 优化后GC

**jstat -gc pid**查看优化后的gc情况。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S0C | S1C | S0U | S1U | EC | EU | OC | OU | MC | MU | CCSC | CCSU | YGC | YGCT | FGC | FGCT | GCT |
| 131072.0 | 131072 | 0 | 0 | 786432 | 527494 | 524288 | 502091.5 | 37248 | 35496.9 | 4992 | 4577.2 | 8 | 1.147 | 14 | 0.794 | 1.941 |
| 131072.0 | 131072 | 0 | 0 | 786432 | 769570.4 | 524288 | 273488.9 | 37248 | 35496.9 | 4992 | 4577.2 | 9 | 1.147 | 15 | 0.927 | 2.075 |
| 131072.0 | 131072 | 0 | 0 | 786432 | 554273.6 | 524288 | 32385.3 | 37248 | 35496.9 | 4992 | 4577.2 | 10 | 1.147 | 18 | 1.048 | 2.196 |
| 131072.0 | 131072 | 0 | 0 | 786432 | 736446.8 | 524288 | 310410.2 | 37248 | 35499.1 | 4992 | 4577.7 | 11 | 1.285 | 19 | 1.155 | 2.44 |
| 131072.0 | 131072 | 0 | 0 | 786432 | 183137.8 | 524288 | 338821.3 | 37248 | 35499.2 | 4992 | 4577.7 | 13 | 1.285 | 23 | 1.397 | 2.682 |
| 131072.0 | 131072 | 0 | 0 | 786432 | 694561.1 | 524288 | 338801.8 | 37248 | 35499.2 | 4992 | 4577.7 | 13 | 1.285 | 25 | 1.418 | 2.703 |

### 优化总结

优化后发现频繁full gc，且笔youngGC还频繁，表示本次**优化失败**，还不如优化前。为什么这样呢？

有可能是以下三点：

1. 元空间不够导致的多余full gc。(本例MU变化不大，排除)
2. 显示调用System.gc()造成多余的full gc，这种一般线上尽量通过­XX:+DisableExplicitGC参数禁用，如果加上了这个JVM启动参数，那么代码中调用System.gc()没有任何效果。(本例无此代码，排除)
3. 老年代空间分配担保机制。(深度怀疑)

## 优化step2

### 问题分析

#### jmap 分析

**jmap -histo pid**分析得知系统有创建大量**com.jvm.User**

|  |  |  |  |
| --- | --- | --- | --- |
| num | #instances | #bytes | class name |
| 1: | 6344 | 512487480 | [B |
| 2: | 73647 | 8728456 | [C |
| 3: | 2889 | 7891536 | [I |
| 4: | 47727 | 1145448 | java.lang.String |
| 5: | 7363 | 816824 | java.lang.Class |
| 6: | 19413 | 621216 | java.util.concurrent.ConcurrentHashMap$Node |
| 7: | 10004 | 480192 | java.nio.HeapCharBuffer |
| 8: | 6715 | 418944 | [Ljava.lang.Object; |
| 9: | 9012 | 288384 | java.util.HashMap$Node |
| 10: | 2926 | 248904 | [Ljava.util.HashMap$Node; |
| 11: | 6106 | 244240 | java.util.LinkedHashMap$Entry |
| 12: | 2695 | 237160 | java.lang.reflect.Method |
| 13: | 14654 | 234464 | java.lang.Object |
| 14: | 113 | 188112 | [Ljava.util.concurrent.ConcurrentHashMap$Node; |
| 15: | 2649 | 148344 | java.util.LinkedHashMap |
| 16: | 5000 | 120000 | com.jvm.User |
| 17: | 1245 | 76560 | [Ljava.lang.String; |

#### Top&Jstack分析

1. 执行 **top -p pid** 查看当前pid cpu占用情况

top - 14:23:32 up 1:04, 6 users, load average: 1.00, 1.03, 0.87

Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie

%Cpu(s): 24.6 us, 0.0 sy, 0.0 ni, 75.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem : 3863572 total, 2033672 free, 1033712 used, 796188 buff/cache

KiB Swap: 2097148 total, 2097148 free, 0 used. 2491504 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

4229 root 20 0 3471168 37632 11724 S 106.7 1.0 27:53.85 java

1. 按**H**查看此pid里面每一个线程的cpu占用情况

top - 14:23:45 up 1:04, 6 users, load average: 1.00, 1.03, 0.87

Threads: 16 total, 1 running, 15 sleeping, 0 stopped, 0 zombie

%Cpu(s): 25.0 us, 0.1 sy, 0.0 ni, 74.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem : 3863572 total, 2033572 free, 1033828 used, 796172 buff/cache

KiB Swap: 2097148 total, 2097148 free, 0 used. 2491404 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

9127 root 20 0 3471168 37632 11724 R 99.9 1.0 28:05.98 java

4243 root 20 0 3471168 37632 11724 S 0.3 1.0 0:01.15 java

4229 root 20 0 3471168 37632 11724 S 0.0 1.0 0:00.00 java

1. 根据cpu占用最高的PID查看堆栈情况

**jstack pid | grep -A 10 $(printf "%x" 子线程pid)，确认代码中com.jvm.User创建5000个，建议减少创建。**

## 优化step3

如果通过jstack 发现线程众多，不要处理，可以通过arthas定位。

### 问题分析

#### trace分析

trace com.jvm.\*Controller \* -n 5 查看5次Controller 所有方法发现：

`---ts=2023-11-06 15:33:29;thread\_name=http-nio-8189-exec-8;id=17;is\_daemon=true;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebappClassLoader@4461c7e3

`---[98.33075ms] com.jvm.IndexController:processUserData()

`---[76.66% 75.37696ms ] com.jvm.IndexController:queryUsers() #12

`---[99.94% 75.334687ms ] com.jvm.IndexController:queryUsers()

`---[82.37% min=0.004223ms,max=0.169358ms,total=62.050756ms,count=5000] com.jvm.User:<init>() #41

#### jad反编译分析

jad --source-only \*.IndexController queryUsers

发现源码40行有5000条创建，建议见啥

private ArrayList<User> queryUsers() {

ArrayList<User> users = new ArrayList<User>();

/\*40\*/ for (int i = 0; i < 5000; ++i) {

/\*41\*/ users.add(new User(i, "zhuge"));

}

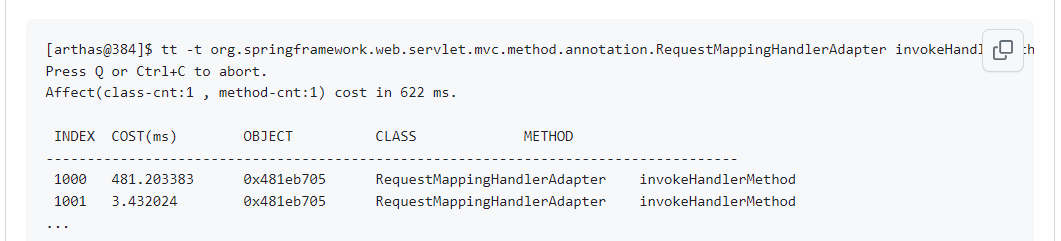
/\*43\*/ return users;

}

## springmvc相关

SpringMVC应用，所有请求都会被RequestMappingHandlerAdapter拦截，我们通过tt命令，监听invokeHandlerMethod的执行，然后在页面随便点点，就会得到以下内容：

tt -t org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter invokeHandlerMethod -n 10



# GC概念

## 对象动态年龄判断

jdk8中如果minor gc后，现存经过1、2、3、4次minor gc的对象存活，且1、2、3的总内存超过survivor区的50%,则3、4以上的对象都直接加入老年代。

意思是N次后面进来的对象超过超过50%,则N及N次以上的都加入到老年代。

可以通过-XX:TargetSurvivorRatio 可以指定百分比。

对象动态年龄判断机制一般是在minor gc之后触发的。

## 老年代空间分配担保机制

年轻代每次minor gc之前JVM都会计算下老年代剩余可用空间

如果这个可用空间小于年轻代里现有的所有对象大小之和(包括垃圾对象)，就会看一个“-XX:-HandlePromotionFailure”(jdk1.8.0\_162默认无设置)的参数是否设置。

如果有这个参数，就会看看老年代的可用内存大小，是否大于之前每一次minor gc后进入老年代的对象的平均大小。

如果上一步结果是小于或者之前说的参数没有设置，那么就会触发一次Full gc，对老年代和年轻代一起回收一次垃圾。

如果回收完还是没有足够空间存放新的对象就会发生"OOM"。

当然，如果minor gc之后剩余存活的需要挪动到老年代的对象大小还是大于老年代可用空间，那么也会触发full gc。

fullgc完之后如果还是没有空间放minor gc之后的存活对象，则也会发生"OOM"。