# GC机制

## GC概要

### JVM堆相关知识

    为什么先说JVM堆?  
    JVM的堆是[**Java**](http://lib.csdn.net/base/javase)对象的活动空间，程序中的类的对象从中分配空间，其存储着正在运行着的应用程序用到的所有对象。这些对象的建立方式就是那些new一类的操作，当对象无用后，是GC来负责这个无用的对象(地球人都知道)。

### JVM堆

#### 新域

存储所有新成生的对象

#### (2) 旧域

新域中的对象，经过了一定次数的GC循环后，被移入旧域

#### (3) 永久域

存储类和方法对象，从配置的角度看，这个域是独立的，不包括在JVM堆内。默认为4M。

新域会被分为3个部分：1.第一个部分叫Eden。(伊甸园？？可能是因为亚当和夏娃是人类最早的活动对象？)2.另两个部分称为辅助生存空间(幼儿园)，我这里一个称为A空间(From space)，一个称为B空间(To Space)。

## GC浅谈

GC的工作目的很明确：在堆中，找到已经无用的对象，并把这些对象占用的空间收回使其可以重新利用.大多数垃圾回收的 [**算法**](http://lib.csdn.net/base/datastructure)思路都是一致的：把所有对象组成一个集合，或可以理解为树状结构，从树根开始找，只要可以找到的都是活动对象，如果找不到，这个对象就是凋零的昨日黄花，应该被回收了。

在sun的文档说明中,对JVM堆的新域,是采用copying算法,该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收.它开始时把堆分成一个 对象面 和多个 空闲面 ,程序从对象面为对象分配空间,当对象面 满了, 基于copying算法的垃圾收集就从根集中扫描活动对象, 并将每个 活动对象 复制到 空闲面 (使得活动对象所占的内存之间没有空闲洞), 这样空闲面 就变成了 对象面, 原来的对象面变成了空闲面, 程序会在新的对象面中分配内存.

对于新生成的对象,都放在Eden中. 当Eden充满时 (小孩太多了), GC将开始工作, 首先停止应用程序的运行, 开始收集垃圾, 把所有可找到的对象 都复制到A空间中, 一旦A空间充满, GC就把在A空间中可找到的对象 都复制到B空间中 (会覆盖原有存储对象),当B空间充满的时候,GC就把在B空间中可找到的对象都复制到A空间中, AB在过程中互换角色. 在活动对象经过一定次数的GC操作之后, 这些活动对象就会被放到 旧域中. 对于这些活动对象, 新域的幼儿园生活结束了.新域为什么要这么折腾?起初在这块我也很迷糊,又查了些资料, 原来是这样: 应用程序生成的绝大部分对象都是短命的, copying算法最理想的状态是, 所有移出Eden的对象都会被收集, 因为这些都是短命鬼, 经历过一定次数的GC后应该被收集, 那么移入旧域的对象都是长命的. 这样可以防止AB空间的来回复制影响应用程序. 实际上这种理想状态是很难达到的, 应用程序中不可避免地存在长命的对象,copying算法的发明者要这些对象都尽量放在新域中, 以保证小范围的复制,压缩旧域的开销可比新域中的复制大的多(旧域在下面说).

对于旧域,采用的是tracing算法的一种,成为标记-清除-压缩收集器, 注意,这有一个压缩, 这是个开销挺大的操作.

垃圾回收主要是对 Young Generation块和Old Generation块内存进行回收, YG用来放新产生的对象,进过几次回收还没回收掉的对象往OG中移动,对YG进行来及回收又叫做MinorGC, 对OG垃圾回收又叫MajorGC, 两块内存回收互不干涉.

## 三. GC流程

[older generation][survivor 1][survivor 2][eden]

\*young generation=eden + survivor

1.当eden满了,触发young GC

2.young GC做了两件事:一, 去掉一部分没有用的object;二, 把老的还被引用的object发送到survior里, 等下几次GC之后survior再放到old里面,

3. 当Old满了,触发Full GC. Full GC很耗内存,把old young里面大部分垃圾回收掉. 这个时候用户线程会被block.

## 四. young generation比例越大不一定最好.

将young的大小设置为总堆大小的一半时会造成效率低下. 如果设置的过小,又会因为young generation收集程序不得不频繁运行而造成瓶颈.

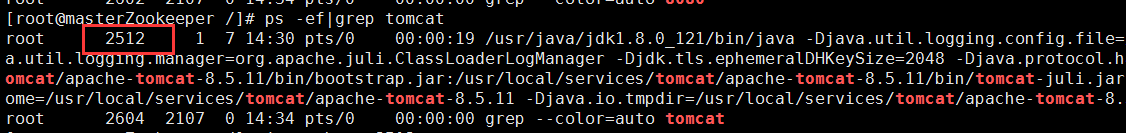
## 五. 总结

从上面的推导可以得出很多结论,下面是前辈的经验总结与自己的认识.

1. JVM堆得大小决定了GC的运行时间. 如果JVM堆的大小超过一定的限度. 那么GC的运行时间会很长.
2. 对象生存的时间越长, GC需要的回收时间也越长, 影响了回收速度.
3. 大多数对象都是短命的. 所以, 如果能让这些对象的生存期在GC的一次运行周期内,wonderful !
4. 应用程序中,建立与释放对象的速度决定了垃圾收集的频率.
5. 如果GC一次运行周期超过3-5秒,这会很影响应用程序的运行,如果可以,应该减少JVM堆的大小了.
6. 前辈经验之谈:通常情况下,JVM堆得大小应为物理内存的80%.

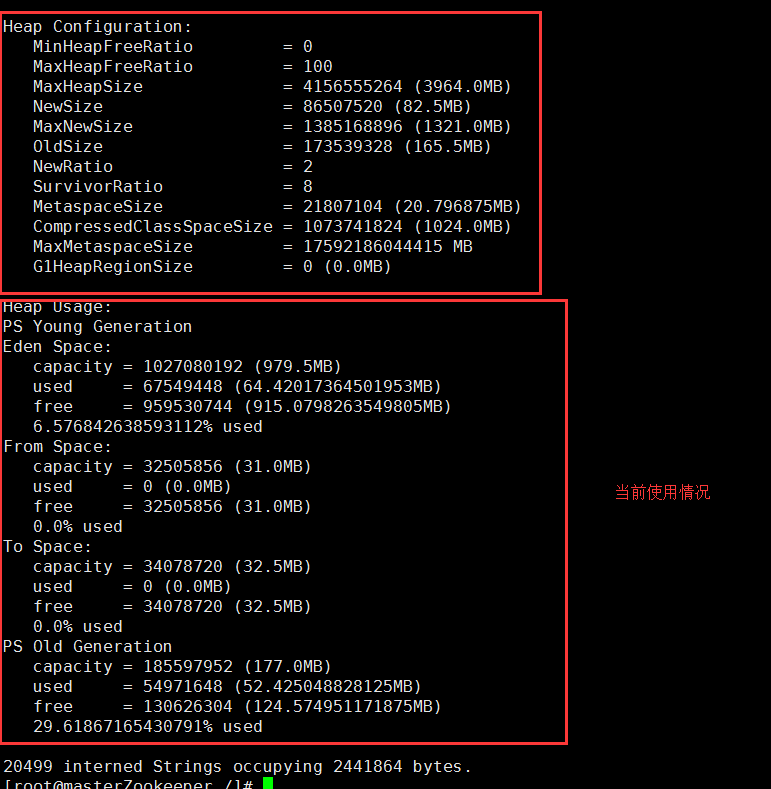
## 六. 看案例

找到java进程号



执行

Jmap –heap 2512



查看GC运行情况

jstat -gcutil -h5 2512 4s 100

### 原文案例

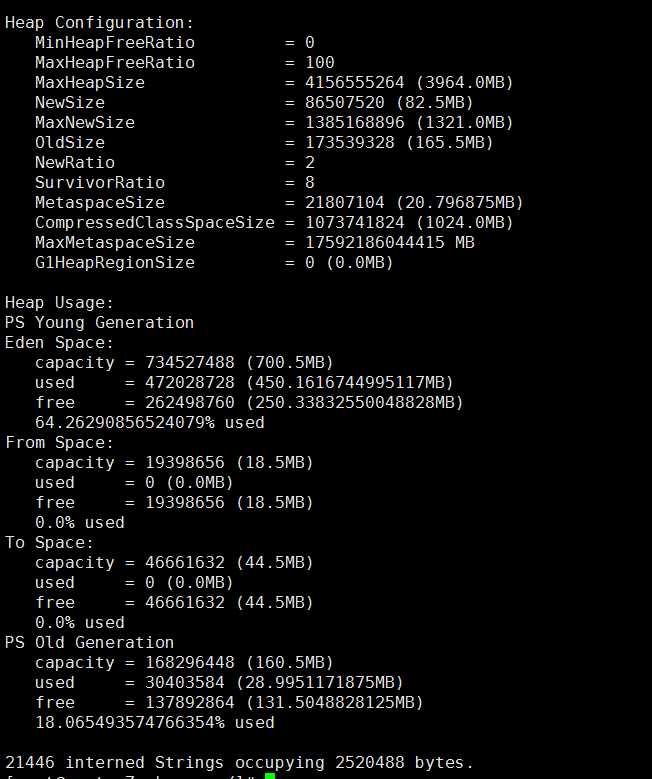
jmap -heap 2343  
Attaching to process ID 2343, please wait...  
Debugger attached successfully.  
Server compiler detected.  
JVM version is 11.0-b16  
   
using thread-local object allocation.  
Parallel GC with 8 thread(s)  
   
Heap Configuration:  
   MinHeapFreeRatio = 40 (空余堆内存小于40%时,JVM就会增大堆值到-Xmx的最大限制)

   MaxHeapFreeRatio = 70 (空余堆内存大于70%时,JVM就会降低堆值到-Xms的最小限制)

   MaxHeapSize      = 4294967296 (4096.0MB)  
   NewSize          = 2686976 (2.5625MB)  
   MaxNewSize       = -65536 (-0.0625MB)  
   OldSize          = 5439488 (5.1875MB)  
   NewRatio         = 2                  （YG，OG 大小比为1：2）  
   SurvivorRatio    = 8  
   PermSize         = 21757952 (20.75MB)  
   MaxPermSize      = 268435456 (256.0MB)  
   
Heap Usage:  
PS Young Generation  
Eden Space:  
   capacity = 1260060672 (1201.6875MB)  
   used     = 64868288 (61.86322021484375MB)  
   free     = 1195192384 (1139.8242797851562MB)  
   5.148028935546367% used  
From Space:  
   capacity = 85524480 (81.5625MB)  
   used     = 59457648 (56.70323181152344MB)  
   free     = 26066832 (24.859268188476562MB)  
   69.52120375359195% used  
To Space:  
   capacity = 85852160 (81.875MB)  
   used     = 0 (0.0MB)  
   free     = 85852160 (81.875MB)  
   0.0% used  
~~~~~~~~~~~~~~~~~~~~~~~~~~这三块为上面所说的YG大小和使用情况  
PS Old Generation  
   capacity = 2291138560 (2185.0MB)  
   used     = 1747845928 (1666.8757705688477MB)  
   free     = 543292632 (518.1242294311523MB)  
   76.28722062099989% used  
~~~~~~~~~~~~~~~~~~~~~~~~~~OG大小和使用情况  
PS Perm Generation  
   capacity = 108265472 (103.25MB)  
   used     = 107650712 (102.6637191772461MB)  
   free     = 614760 (0.5862808227539062MB)  
   99.43217353728436% used  
   
这台机器简单说YG内存1G，OG内存2G，总内存4G  
在这样的配置下，GC运行情况：  
jstat -gcutil -h5 2343 4s 100  
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT    
 79.82   0.00  75.34  78.55  99.44   7646 1221.668   398 2052.993 3274.661  
  0.00  79.52   0.62  78.63  99.44   7647 1221.782   398 2052.993 3274.775 这里发生了一次YG GC，也就是MinorGC，耗时0.12s  
  0.00  79.52  28.95  78.63  99.44   7647 1221.782   398 2052.993 3274.775  
  0.00  79.52  46.34  78.63  99.44   7647 1221.782   398 2052.993 3274.775  
   
同时可以看到总共进行了398次Major GC 总耗时2052.993 所以每次Major GC时间为：2052.993/398=5.16秒  
这是个很严重的问题，进行Major GC的时候程序会暂停，无法响应，居然会暂停5秒多，这谁都无法接受吧 :)  
同样Minor GC进行了7647次，总用时1221.782 平均时间为0.16秒，算是可以接受

再来看看修改配置后：  
jmap -heap 14103  
Attaching to process ID 14103, please wait...  
Debugger attached successfully.  
Server compiler detected.  
JVM version is 11.0-b16  
   
using parallel threads in the new generation.  
using thread-local object allocation.  
Concurrent Mark-Sweep GC  
   
Heap Configuration:  
   MinHeapFreeRatio = 40  
   MaxHeapFreeRatio = 70  
   MaxHeapSize      = 4294967296 (4096.0MB)  
   NewSize          = 536870912 (512.0MB)  
   MaxNewSize       = 536870912 (512.0MB)  
   OldSize          = 5439488 (5.1875MB)  
   NewRatio         =4                         YG：OG          1：4         
   SurvivorRatio    = 8  
   PermSize         = 268435456 (256.0MB)  
   MaxPermSize      = 268435456 (256.0MB)  
   
Heap Usage:  
New Generation (Eden + 1 Survivor Space):  
   capacity = 483196928 (460.8125MB)  
   used     = 428284392 (408.4438247680664MB)  
   free     = 54912536 (52.368675231933594MB)  
   88.63557841162434% used  
Eden Space:  
   capacity = 429522944 (409.625MB)  
   used     = 404788608 (386.0364990234375MB)  
   free     = 24734336 (23.5885009765625MB)  
   94.24144010337199% used  
From Space:  
   capacity = 53673984 (51.1875MB)  
   used     = 23495784 (22.407325744628906MB)  
   free     = 30178200 (28.780174255371094MB)  
   43.77499534970238% used  
To Space:  
   capacity = 53673984 (51.1875MB)  
   used     = 0 (0.0MB)  
   free     = 53673984 (51.1875MB)  
   0.0% used  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~YG 大小和使用状态  
concurrent mark-sweep generation:  
   capacity = 3758096384 (3584.0MB)  
   used     = 1680041600 (1602.2125244140625MB)  
   free     = 2078054784 (1981.7874755859375MB)  
   44.70459052494594% used  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~OG 大小和使用状态  
Perm Generation:  
   capacity = 268435456 (256.0MB)  
   used     = 128012184 (122.0819320678711MB)  
   free     = 140423272 (133.9180679321289MB)  
   47.688254714012146% used  
   
在这个配置下，GC运行情况：  
jstat -gcutil -h5 14103 4s 100  
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT    
 47.49   0.00  64.82  46.08  47.69  20822 2058.631    68   22.734 2081.365  
  0.00  37.91  38.57  46.13  47.69  20823 2058.691    68   22.734 2081.425 这里发生了一次YG GC，也就是MinorGC，耗时0.06s  
 46.69   0.00  15.19  46.18  47.69  20824 2058.776    68   22.734 2081.510  
 46.69   0.00  74.59  46.18  47.69  20824 2058.776    68   22.734 2081.510  
  0.00  40.29  19.95  46.24  47.69  20825 2058.848    68   22.734 2081.582  
   
MajorGC平均时间：22.734/68=0.334秒（上面是5秒多吧）  
MinorGC平均时间：2058.691/20823=0.099秒（比上面略少）

## 测试测试



上图是用tomcat默认启动 堆栈配置以及使用情况.

JAVA\_OPTS=”-server -XX:MetaspaceSize=50m”

|  |  |
| --- | --- |
| 参数 | 描述 |
|  |  |
|  |  |
| -XX:+UseSerialGC | Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收 |
| -XX:+UseParNewGC | 打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收 |
| -XX:+UseConcMarkSweepGC | 使用ParNew + CMS +  Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。 |
| -XX:+UseParallelGC | Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge +  Serial Old的收集器组合进行回收 |
| -XX:+UseParallelOldGC | 使用Parallel Scavenge +  Parallel Old的收集器组合进行回收 |
| -XX:SurvivorRatio | 新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1 |
| -XX:PretenureSizeThreshold | 直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| -XX:MaxTenuringThreshold | 晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代 |
| -XX:UseAdaptiveSizePolicy | 动态调整java堆中各个区域的大小以及进入老年代的年龄 |
| -XX:+HandlePromotionFailure | 是否允许新生代收集担保，进行一次minor gc后, 另一块Survivor空间不足时，将直接会在老年代中保留 |
| -XX:ParallelGCThreads | 设置并行GC进行内存回收的线程数 |
| -XX:GCTimeRatio | GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效 |
| -XX:MaxGCPauseMillis | 设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效 |
| -XX:CMSInitiatingOccupancyFraction | 设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70 |
| -XX:+UseCMSCompactAtFullCollection | 由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效 |
| -XX:+CMSFullGCBeforeCompaction | 设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用 |
| -XX:+UseFastAccessorMethods | 原始类型优化 |
| -XX:+DisableExplicitGC | 是否关闭手动System.gc |
| -XX:+CMSParallelRemarkEnabled | 降低标记停顿 |
| -XX:LargePageSizeInBytes | 内存页的大小不可设置过大，会影响Perm的大小，-XX:LargePageSizeInBytes=128m |