24 | 如何优化JVM内存分配？

2019-07-16 刘超

你好，我是刘超。

JVM调优是一个系统而又复杂的过程，但我们知道，在大多数情况下，我们基本不用去调整JVM内存分配，因为一些初始化的参数已经可以保证应用服务正常稳定地工作了。

但所有的调优都是有目标性的，JVM内存分配调优也一样。没有性能问题的时候，我们自然不会随意改变JVM内存分配的参数。那有了问题呢？有了什么样的性能问题我们需要对其进行调优呢？又该如何调优呢？这就是我今天要分享的内容。

JVM内存分配性能问题

谈到JVM内存表现出的性能问题时，你可能会想到一些线上的JVM内存溢出事故。但这方面的事故往往是应用程序创建对象导致的内存回收对象难，一般属于代码编程问题。

但其实很多时候，在应用服务的特定场景下，JVM内存分配不合理带来的性能表现并不会像内存溢出问题这么突出。可以说如果你没有深入到各项性能指标中去，是很难发现其中隐藏的性能损耗。

JVM内存分配不合理最直接的表现就是频繁的GC，这会导致上下文切换等性能问题，从而降低系统的吞吐量、增加系统的响应时间。因此，如果你在线上环境或性能测试时，发现频繁的GC，且是正常的对象创建和回收，这个时候就需要考虑调整JVM内存分配了，从而减少GC所带来的性能开销。

对象在堆中的生存周期

了解了性能问题，那需要做的势必就是调优了。但先别急，在了解JVM内存分配的调优过程之前，我们先来看看一个新创建的对象在堆内存中的生存周期，为后面的学习打下基础。

在第20讲中，我讲过JVM内存模型。我们知道，在JVM内存模型的堆中，堆被划分为新生代和老年代，新生代又被进一步划分为Eden区和Survivor区，最后Survivor由From Survivor和To Survivor组成。

当我们新建一个对象时，对象会被优先分配到新生代的Eden区中，这时虚拟机会给对象定义一个对象年龄计数器（通过参数-XX:MaxTenuringThreshold设置）。

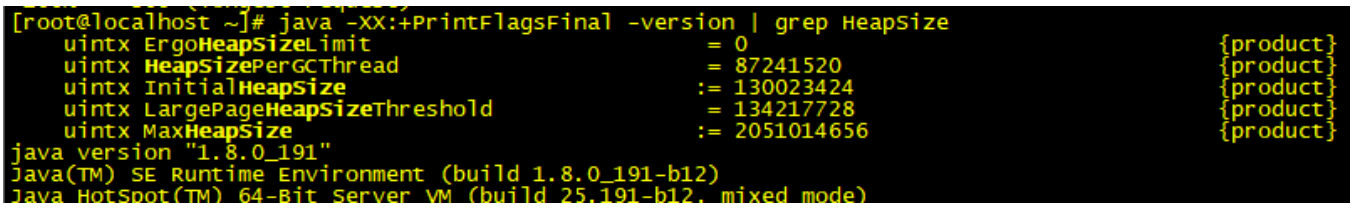
同时，也有另外一种情况，当Eden空间不足时，虚拟机将会执行一个新生代的垃圾回收（Minor GC）。这时JVM会把存活的对象转移到Survivor中，并给对象的年龄+1。对象在Survivor中同样也会经历MinorGC，每经过一次MinorGC，对象的年龄将会+1。

当然了，内存空间也是有设置阈值的，可以通过参数-XX:PetenureSizeThreshold设置直接被分配到老年代的最大对象，这时如果分配的对象超过了设置的阀值，对象就会直接被分配到老年代，这样做的好处就是可以减少新生代的垃圾回收。

查看JVM堆内存分配

我们知道了一个对象从创建至回收到堆中的过程，接下来我们再来了解下JVM堆内存是如何分配的。在默认不配置JVM堆内存大小的情况下，JVM根据默认值来配置当前内存大小。我们可以通过以下命令来查看堆内存配置的默认值：

```
java -XX:+PrintFlagsFinal -version | grep HeapSize
jmap -heap 17284
```



```
[root@localhost ~]# ps -ef|grep java
root     17284 10321  0 17:19 pts/1    00:00:11 java -jar heapTest-0.0.1-SNAPSHOT.jar
root     17741 17723  0 19:07 pts/2    00:00:00 grep --color=auto java
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# jmap -heap 17284
Attaching to process ID 17284, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2051014656 (1956.0MB)
   NewSize                  = 42991616 (41.0MB)
   MaxNewSize               = 683671552 (652.0MB)
   OldSize                  = 87031808 (83.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 254803968 (243.0MB)
   used     = 19794256 (18.877273559570312MB)
   free     = 235009712 (224.1227264404297MB)
   7.768425333156507% used
From Space:
   capacity = 8388608 (8.0MB)
   used     = 0 (0.0MB)
   free     = 8388608 (8.0MB)
   0.0% used
To Space:
   capacity = 9961472 (9.5MB)
   used     = 0 (0.0MB)
   free     = 9961472 (9.5MB)
   0.0% used
PS Old Generation
   capacity = 70254592 (67.0MB)
   used     = 17130840 (16.337242126464844MB)
   free     = 53123752 (50.662757873535156MB)
   24.383943472335588% used
```

通过命令，我们可以获得在这台机器上启动的JVM默认最大堆内存为1953MB，初始化大小为124MB。

在JDK1.7中，默认情况下年轻代和老年代的比例是1:2，我们可以通过-XX:NewRatio重置该配置项。年轻代中的Eden和To Survivor、From Survivor的比例是8:1:1，我们可以通过-XX:SurvivorRatio重置该配置项。

在JDK1.7中如果开启了-XX:+UseAdaptiveSizePolicy配置项，JVM将会动态调整Java堆中各个区域的大小以及进入老年代的年龄，-XX:NewRatio和-XX:SurvivorRatio将会失效，而JDK1.8是默认开启-XX:+UseAdaptiveSizePolicy配置项的。

还有，在JDK1.8中，不要随便关闭UseAdaptiveSizePolicy配置项，除非你已经对初始化堆内存/最大堆内存、年轻代/老年代以及Eden区/Survivor区有非常明确的规划了。否则JVM将会分配最小堆内存，年轻代和老年代按照默认比例1:2进行分配，年轻代中的Eden和Survivor则按照默认比例8:2进行分配。这个内存分配未必是应用服务的最佳配置，因此可能会给应用服务带来严重

的性能问题。

JVM内存分配的调优过程

我们先使用JVM的默认配置，观察应用服务的运行情况，下面我将结合一个实际案例来讲述。现模拟一个抢购接口，假设需要满足一个5W的并发请求，且每次请求会产生20KB对象，我们可以通过千级并发创建一个1MB对象的接口来模拟万级并发请求产生大量对象的场景，具体代码如下：

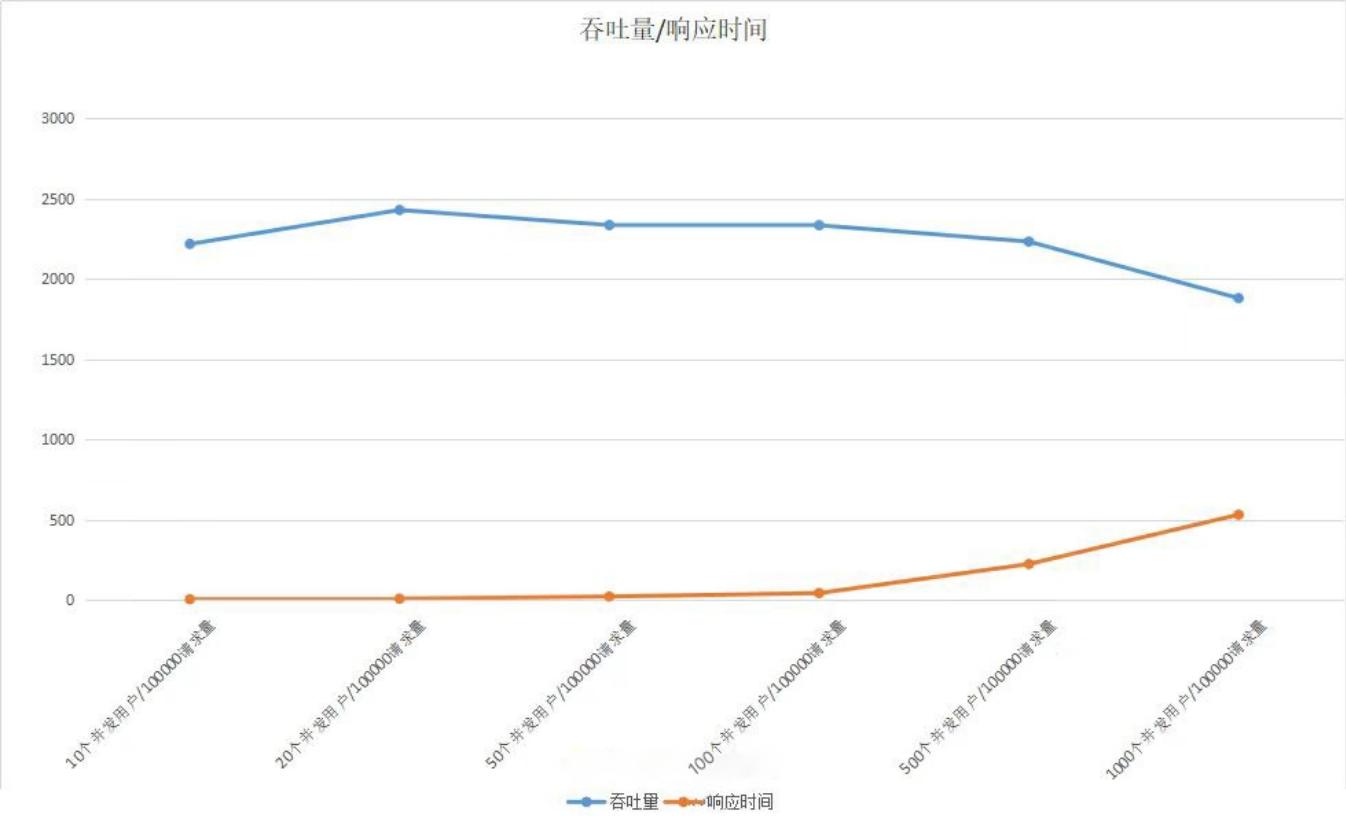
```
@RequestMapping(value = "/test1")
public String test1(HttpServletRequest request) {
 List<Byte[]> temp = new ArrayList<Byte[]>();

 Byte[] b = new Byte[1024*1024];
 temp.add(b);

 return "success";
}
```

AB压测

分别对应用服务进行压力测试，以下是请求接口的吞吐量和响应时间在不同并发用户数下的变化情况：



可以看到，当并发数量到了一定值时，吞吐量就上不去了，响应时间也迅速增加。那么，在JVM内部运行又是怎样的呢？

分析GC日志

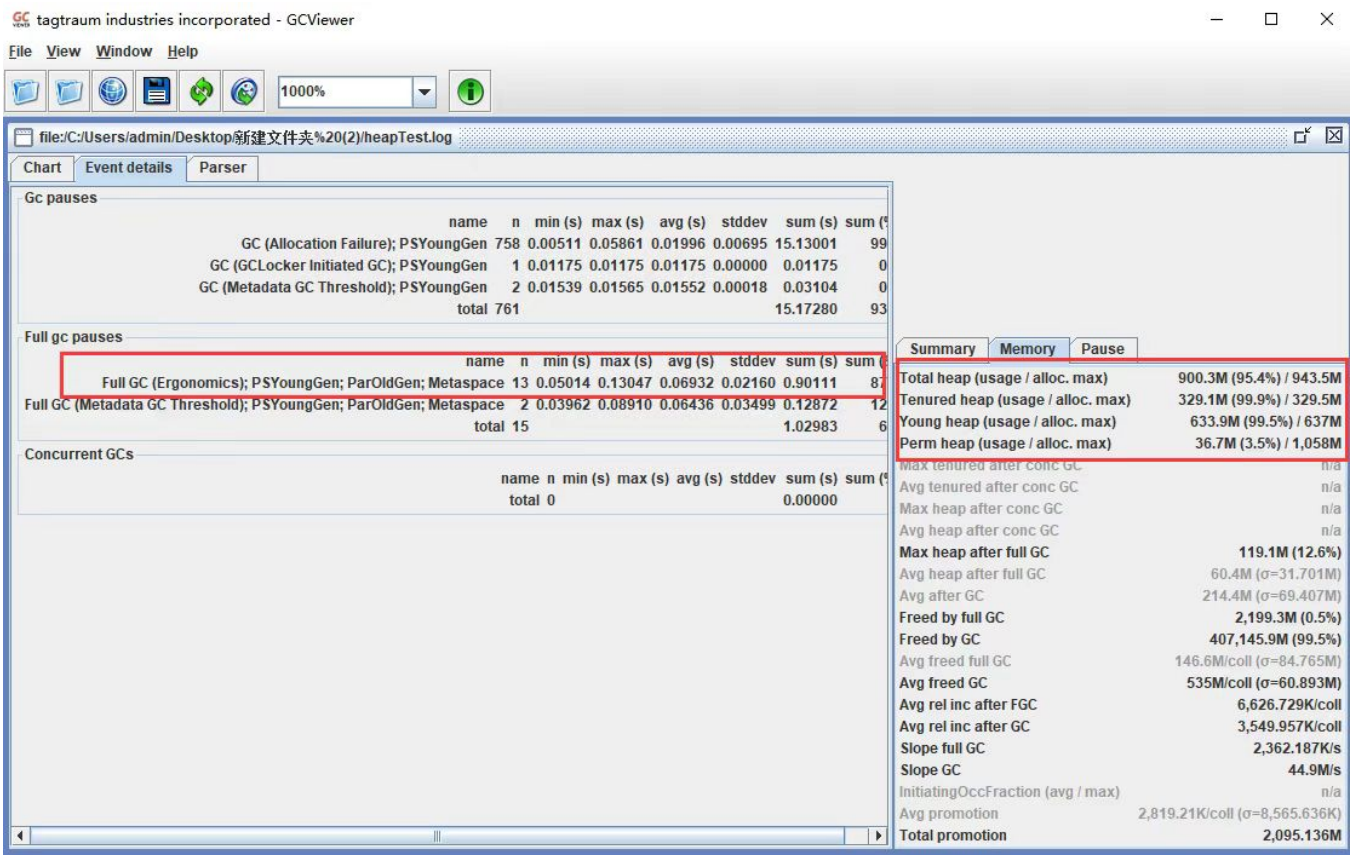
此时我们可以通过GC日志查看具体的回收日志。我们可以通过设置VM配置参数，将运行期间的GC日志 dump下来，具体配置参数如下：

```
-XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:/log/heapTest.log
```

以下是各个配置项的说明：

- -XX:PrintGCTimeStamps：打印GC具体时间；
- -XX:PrintGCDetails ：打印出GC详细日志；
- -Xloggc: path：GC日志生成路径。

收集到GC日志后，我们就可以使用第22讲中介绍过的GCViewer工具打开它，进而查看到具体的GC日志如下：



主页面显示FullGC发生了13次，右下角显示年轻代和老年代的内存使用率几乎达到了100%。而FullGC会导致stop-the-world的发生，从而严重影响到应用服务的性能。此时，我们需要调整堆内存的大小来减少FullGC的发生。

参考指标

我们可以将某些指标的预期值作为参考指标，上面的GC频率就是其中之一，那么还有哪些指标可以为我们提供一些具体的调优方向呢？

GC频率：高频的FullGC会给系统带来非常大的性能消耗，虽然MinorGC相对FullGC来说好了许多，但过多的MinorGC仍会给系统带来压力。

内存：这里的内存指的是堆内存大小，堆内存又分为年轻代内存和老年代内存。首先我们要分析堆内存大小是否合适，其实是分析年轻代和老年代的比例是否合适。如果内存不足或分配不均匀，会增加FullGC，严重的将导致CPU持续爆满，影响系统性能。

吞吐量：频繁的FullGC将会引起线程的上下文切换，增加系统的性能开销，从而影响每次处理的线程请求，最终导致系统的吞吐量下降。

延时：JVM的GC持续时间也会影响到每次请求的响应时间。

具体调优方法

调整堆内存空间减少FullGC：通过日志分析，堆内存基本被用完了，而且存在大量FullGC，这意味着我们的堆内存严重不足，这个时候我们需要调大堆内存空间。

```
java -jar -Xms4g -Xmx4g heapTest-0.0.1-SNAPSHOT.jar
```

以下是各个配置项的说明：

- -Xms：堆初始大小；
- -Xmx：堆最大值。

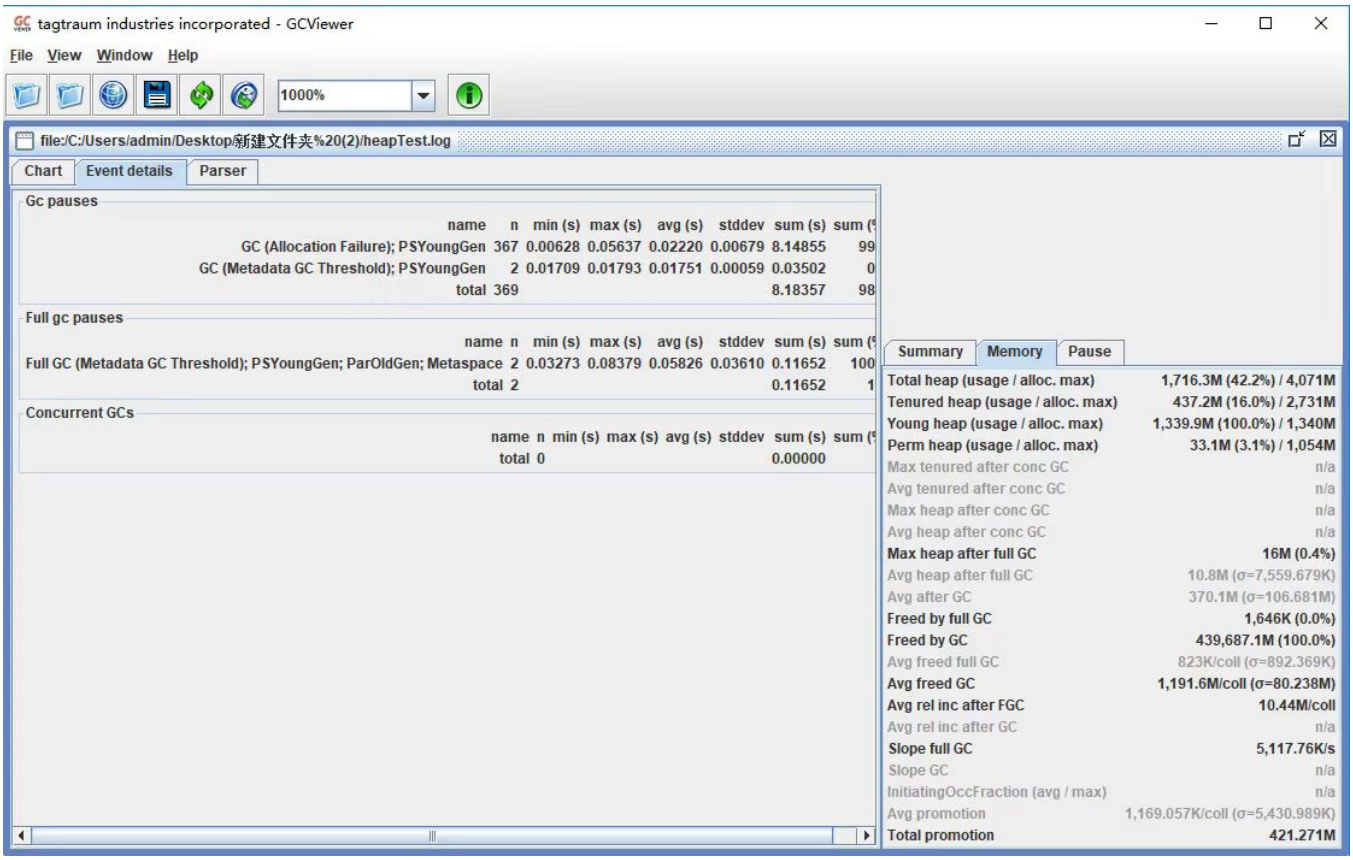
调大堆内存之后，我们再来测试下性能情况，发现吞吐量提高了40%左右，响应时间也降低了将近50%。

```
Concurrency Level:      1000
Time taken for tests:   37.677 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      13900000 bytes
HTML transferred:       700000 bytes
Requests per second:    2654.12 [#/sec] (mean)
Time per request:       376.773 [ms] (mean)
Time per request:       0.377 [ms] (mean, across all concurrent requests)
Transfer rate:          360.28 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  232 773.0      0    7021
Processing:     1  134 120.9    102    1729
Waiting:        1  125 102.3     99    1729
Total:          1  366 783.4    117    8033

Percentage of the requests served within a certain time (ms)
  50%    117
  66%    183
  75%    246
  80%    328
  90%   1092
  95%   1220
  98%   3096
  99%   3224
 100%   8033 (longest request)
```

再查看GC日志，发现FullGC频率降低了，老年代的使用率只有16%了。



调整年轻代减少MinorGC：通过调整堆内存大小，我们已经提升了整体的吞吐量，降低了响应时间。那还有优化空间吗？我们还可以将年轻代设置得大一些，从而减少一些MinorGC（第22讲有通过降低Minor GC频率来提高系统性能的详解）。

```
java -jar -Xms4g -Xmx4g -Xmn3g heapTest-0.0.1-SNAPSHOT.jar
```

再进行AB压测，发现吞吐量上去了。

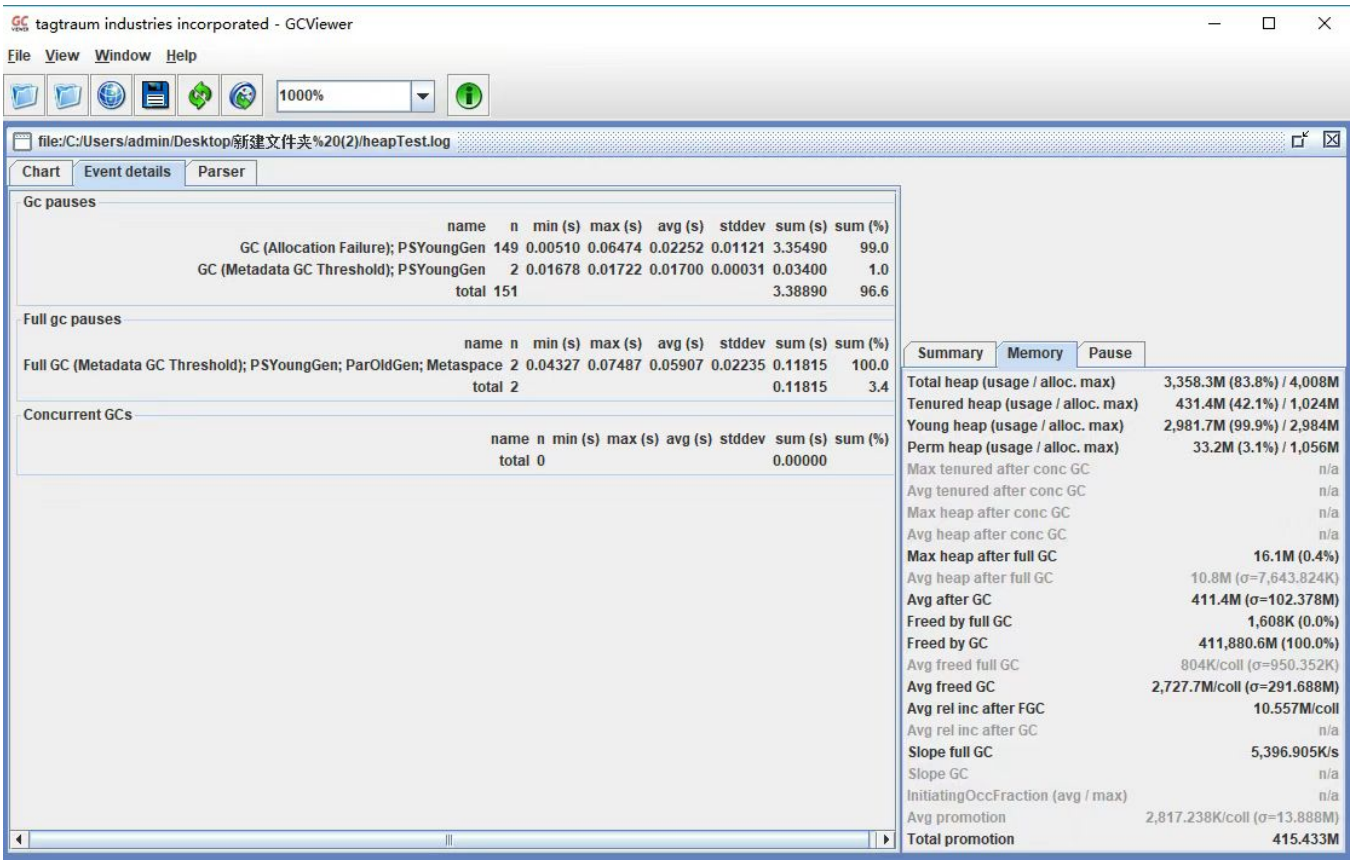
```
Document Path:          /test2
Document Length:        7 bytes

Concurrency Level:      1000
Time taken for tests:   34.157 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      13900000 bytes
HTML transferred:       700000 bytes
Requests per second:    2927.68 [#/sec] (mean)
Time per request:       341.568 [ms] (mean)
Time per request:       0.342 [ms] (mean, across all concurrent requests)
Transfer rate:          397.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  208 812.5      0   15040
Processing:     1  126 117.2     97    1766
Waiting:        1  120 106.9     94    1683
Total:          1  334 824.0    114   15225

Percentage of the requests served within a certain time (ms)
  50%    114
  66%    166
  75%    216
  80%    266
  90%   1076
  95%   1184
  98%   3066
  99%   3160
 100%  15225 (longest request)
```

再查看GC日志，发现MinorGC也明显降低了，GC花费的总时间也减少了。



设置Eden、Survivor区比例：在JVM中，如果开启 AdaptiveSizePolicy，则每次 GC 后都会重新计算 Eden、From Survivor和 To Survivor区的大小，计算依据是 GC 过程中统计的 GC 时间、吞吐量、内存占用量。这个时候SurvivorRatio默认设置的比例会失效。

在JDK1.8中，默认是开启AdaptiveSizePolicy的，我们可以通过-XX:-UseAdaptiveSizePolicy关闭该项配置，或显示运行-XX:SurvivorRatio=8将Eden、Survivor的比例设置为8:2。大部分新对象都是在Eden区创建的，我们可以固定Eden区的占用比例，来调优JVM的内存分配性能。

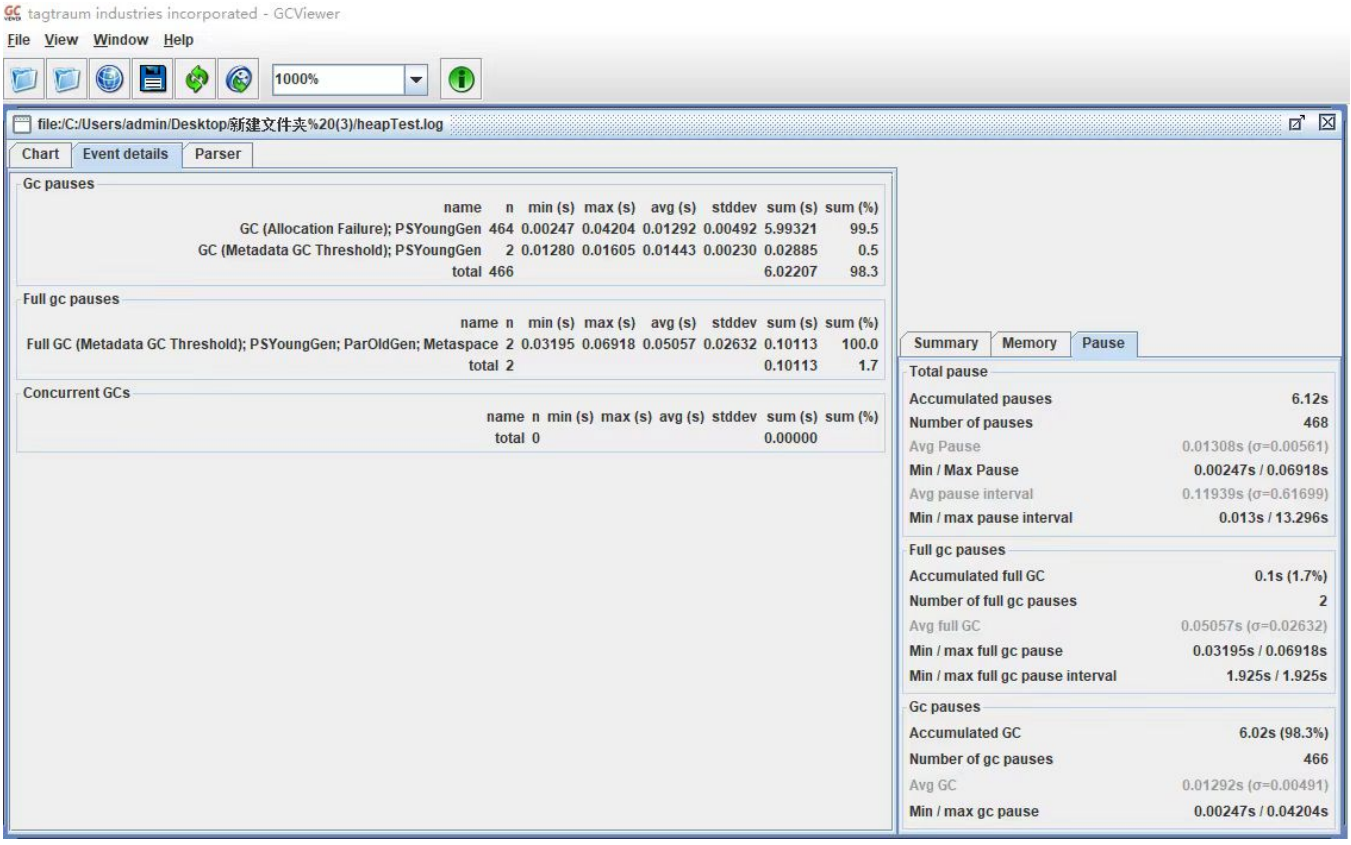
再进行AB性能测试，我们可以看到吞吐量提升了，响应时间降低了。

```
Document Path:          /test2
Document Length:        7 bytes

Concurrency Level:      1000
Time taken for tests:   33.322 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      13900000 bytes
HTML transferred:       700000 bytes
Requests per second:    3001.03 [#/sec] (mean)
Time per request:       333.219 [ms] (mean)
Time per request:       0.333 [ms] (mean, across all concurrent requests)
Transfer rate:          407.37 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  222 909.2      0   31044
Processing:     1  101  79.2     82     975
Waiting:        1   94  64.5     80     975
Total:          1  323 913.2     94   31135

Percentage of the requests served within a certain time (ms)
  50%     94
  66%    133
  75%    171
  80%    210
  90%   1069
  95%   1155
  98%   3063
  99%   3136
 100%  31135 (longest request)
```



总结

JVM内存调优通常和GC调优是互补的，基于以上调优，我们可以继续对年轻代和堆内存的垃圾回收算法进行调优。这里可以结合上一讲的内容，一起完成JVM调优。

虽然分享了一些JVM内存分配调优的常用方法，但我还是建议你在进行性能压测后如果没有发现突出的性能瓶颈，就继续使用JVM默认参数，起码在大部分的场景下，默认配置已经可以满足我们的需求了。但满足不了也不要慌张，结合今天所学的内容去实践一下，相信你会有新的收获。

思考题

以上我们都是基于堆内存分配来优化系统性能的，但在NIO的Socket通信中，其实还使用到了堆外内存来减少内存拷贝，实现Socket通信优化。你知道堆外内存是如何创建和回收的吗？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

bro. 10

堆外内存创建有两种方式:1.使用ByteBuffer.allocateDirect()得到一个DirectByteBuffer对象,初始化堆外内存大小,里面会创建Cleaner对象,绑定当前this.DirectByteBuffer的回收,通过put,get传递进去Byte数组,或者序列化对象,Cleaner对象实现一个虚引用(当内存被回收时,会受到一个系统通知)当Full GC的时候,如果DirectByteBuffer标记为垃圾被回收,则Cleaner会收到通知调用clean()方法,回收改堆外内存DirectByteBuffer

2019-07-22

作者回复

回答很全面，赞！

2019-07-24

QQ怪 3

盲目增大堆内存可能会让吞吐量不增反减，堆内存大了，每次gc扫描对象也就越多也越需要花费时间，反而会适得其反

2019-07-16

作者回复

对的。合理设置堆内存大小，根据实际业务调整，不宜过大，也不宜过小。

2019-07-17

青梅煮酒 2

超哥好，我们经常发现生产环境内存使用超过90%持续3分钟，没有outofmer, dump下来堆没有发现问题，这种情况每不确定几小时就会一次，求解答

2019-07-16

作者回复

你好，某一时间段高峰值的访问可能会有这种情况，JVM会最大可能进行对象的回收，防止内存溢出异常的发生。如果不是内存泄漏，或者瞬时并发量大大超过预期并发量的情况，几乎很少发生内存溢出异常。

建议结合内存持续占用率以及Full GC发生的频率来分析调优。

2019-07-17

迎风劲草 1

老师，你的这个抢购场景下我理解是不是新生代越大越好，因为对象都是生命周期较短的对象。尽量在新生代中被回收掉。

2019-07-18

作者回复

也不是越大越好，因为新生代过大，会导致minor gc的停顿时间过长。

我们知道，如果新生代很快就满了，会以担保的方式将新增的对象直接分配到老年代，这样增加了老年代回收的成本，这个成本跟具体的垃圾收集器相关。所以我们需要适当的调大年轻代，将对象尽量留在年轻代回收。

如果调整太大，我们知道每次Minor GC分为对象标记和复制两个阶段，并且都是STW的，如果对象过于庞大，有可能标记时间要大于复制时间，这样反而适得其反。

2019-07-22

殷传宁 0

吞吐量那个图是怎么查看的？方法能说一下吗？

2019-10-13

作者回复

使用AB工具运行后会自动打印出这些信息，回顾下加餐篇《加餐 | 推荐几款常用的性能测试工具》

2019-10-19

没有小名的曲儿 0

超哥好，我们线上的的服务器经常会进行图片处理，内存15G，堆内存设置的-Xms与-Xmx都是10G，经常在进行图片处理时，用top查看java进程，占用率高达79.1%。
等传完之后，观察内存一直都是这么多。jmap也看了，是正常回收的。并且各个代free率很高，但是top查看java进程内存一直占用79.1%。
是不是因为设置了-Xms为10G，尽管GC了，也不会降低占用整个分配的物理内存呢？

2019-09-30

作者回复

如果内存一直没有释放，我想跟内存设置比例没有关系，可能是引用没有释放，尝试在传完之后手动释放内存试试

2019-10-02

SDL 0

老师 为什么我用这个java -XX查看某个参数都没有相关信息输出的？就只有版本号那些信息呢

2019-09-26

作者回复

java -X可以查看部分JVM参数信息

2019-10-02

小笨蛋 0

请问堆内存的分配有没有一个大概的标准既然都提到了不能太大也不能太小

2019-09-17

作者回复

需要根据自己的项目来具体做配置，如果不清除具体需要的配置大小，使用默认配置就可以了

2019-09-18

godtrue 0

课后思考及问题
1：JVM 内存分配不合理最直接的表现就是频繁的 GC，这会导致上下文切换等性能问题，从而降低系统的吞吐量、增加系统的响应时间。
频繁的GC，GC线程和应用线程会频繁的切入切出，所以，降低了系统的性能。
2：老师好，现在有这么一个问题，我们有一个定时任务跑一次大概会有2亿条数据一条数据大概40kb大小，一次大概7.4TB多的数据，分布式任务50台机器需要刷新2个多小时，我们需要持久化，为了提高性能做了异步发送MQ到另外的机器来持久化，不过MQ积压严重，数据跑一次耗时太长，有什么建议的优化思路嘛？拆分消息会加剧业务处理的复杂度，目前我能想到的是加机器加带宽。请老师给个优化的思考？

2019-09-11

作者回复

优化传输性能，例如使用特定的数据结构序列化与反序列化传输数据（protobuff序列化），并且提高单台服务并行处理能力。

2019-09-15

风轻扬 0

老师，如果允许分配担保机制失败。那即使老年代的空间不足以吃下年轻代的对象。jvm也会冒险进行minor gc的。gc之后，如果老年代还是吃不下对象，这个时候才会Full GC。那关闭这个分配担保机制，感觉好一点啊，反正有Full GC兜底呢

2019-09-10

作者回复

打开分配担保机制，是为了避免Full GC过于频繁。

2019-09-10

疯狂咸鱼 0

ab压测是什么工具的

2019-09-03

作者回复

是一种简单的压力测试工具，可以网上查询下资料

2019-09-04

涛哥迷妹 0

你好，请问G1调优能不能也讲讲。主要应该注意些什么和cms这种调优的差异

2019-08-29

作者回复

嗯，在后面的答疑课堂中讲到了，有问题欢迎提出

2019-09-02

高鑫 0

大大，有个问题请教。如果survivor区不能容纳eden区的活跃对象，那么这些对象会直接晋升到oldgen么？能否详述一下对象晋升的流程

2019-07-25

晓杰 0

可以通过directBuffer创建堆外内存，full gc可以对堆外内存进行回收

2019-07-17

晓杰 0

full gc会对堆外内存进行回收

2019-07-17

歪曲丶 0

Unsafe DirectByteBuffer都可以直接开辟堆外内存 啥时候回收 可以在full gc的时候回收 难的是堆外的阈值设定 监控堆外内存 jmx好像取不到堆外的大小了吧 之前看到R大在1.7的时候粗略的

算下的 有种可能是老年代引用堆外的引用 但是old gc或者full gc迟迟不gc 那堆外就有可能oom

2019-07-17

我又不乱来 0

超哥，有两个疑问。
当第一次创建对象的时候 eden 空间不足会进行一次minor gc把存活的对象放到from s区。如果这个时候from s放不下。会发生一次担保进入老年代吗？
当一次创建对象的时候eden空间不足进入from s区。当第二次创建对象的时候eden空间又不足了，这个时候会把，eden和第一次存在from s 区的对象进行gc 存活的放在 to s区，to s区空间不足，进行担保放入老年代？这样的理解对吗。

2019-07-16

作者回复

对的，细节把握的很好！

前提是老年代有容量这些对象的空间，才会进行分配担保。如果老年代剩余空间小于每次minor gc晋升到老年代的平均值，则会发起一次Full GC。

2019-07-17

LW 0

老师的压测结果图形化是用什么工具做的？

2019-07-16

作者回复

Excel

2019-07-17

杨俊 0

印象中本地内存分配堆外内存，c语言用到的内存就是这样，回收是通过GC自动扫描directbyte buffer对象回收

2019-07-16

作者回复

对的，可以手动回收掉，如果不手动回收，则会通过FullGC来回收。

2019-07-17

-W.LI- 0

老师好!堆外缓存实在FGC的时候回收的吧。
AdaptiveSizePolicy这个参数是不是不太智能啊?我项目4G内存默认开启的AdaptiveSizePolicy。发现只给年轻代分配了136M内存。平时运行到没啥问题，没到定时任务的点就频繁FGC。每次定时任务执行完，都会往老年代推40多M，一天会堆300多M到老年代，也不见它把年轻代调大。用的parNew+CMS。后来把年轻代调整到1G(单次YGC耗时从20ms增加到了40ms)，每天老年代内存涨20M左右。

2019-07-16

作者回复

这个会根据我们的内存创建大小合理分配内存，并不仅仅考虑对象晋升的问题，还会综合考虑

回收停顿时间等因素。

针对某些特殊场景，我们可以手动来配置调优。

2019-07-17