

## 34 | JVM GC原理及调优的基本思路

2019-07-30 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 08:13 大小 7.54M



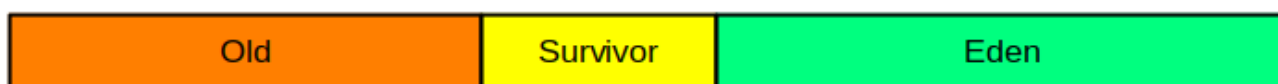
和 Web 应用程序一样，Tomcat 作为一个 Java 程序也跑在 JVM 中，因此如果我们要对 Tomcat 进行调优，需要先了解 JVM 调优的原理。而对于 JVM 调优来说，主要是 JVM 垃圾收集的优化，一般来说是因为有问题才需要优化，所以对于 JVM GC 来说，如果你观察到 Tomcat 进程的 CPU 使用率比较高，并且在 GC 日志中发现 GC 次数比较频繁、GC 停顿时间长，这表明你需要对 GC 进行优化了。

在对 GC 调优的过程中，我们不仅需要知道 GC 的原理，更重要的是要熟练使用各种监控和分析工具，具备 GC 调优的实战能力。CMS 和 G1 是时下使用率比较高的两款垃圾收集器，从 Java 9 开始，采用 G1 作为默认垃圾收集器，而 G1 的目标也是逐步取代 CMS。所以今天我们先来简单回顾一下两种垃圾收集器 CMS 和 G1 的区别，接着通过一个例子帮你提高 GC 调优的实战能力。

## CMS vs G1

CMS 收集器将 Java 堆分为**年轻代**（ Young ）或**年老代**（ Old ）。这主要是因为有研究表明，超过 90% 的对象在第一次 GC 时就被回收掉，但是少数对象往往会存活较长的时间。

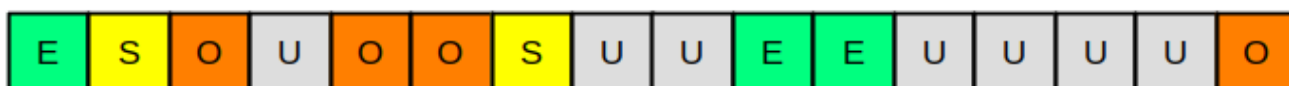
CMS 还将年轻代内存空间分为**幸存者空间**（ Survivor ）和**伊甸园空间**（ Eden ）。新的对象始终在 Eden 空间上创建。一旦一个对象在一次垃圾收集后还幸存，就会被移动到幸存者空间。当一个对象在多次垃圾收集之后还存活时，它会移动到年老代。这样做的目的是在年轻代和年老代采用不同的收集算法，以达到较高的收集效率，比如在年轻代采用复制 - 整理算法，在年老代采用标记 - 清理算法。因此 CMS 将 Java 堆分成如下区域：



与 CMS 相比，G1 收集器有两大特点：

G1 可以并发完成大部分 GC 的工作，这期间不会 “Stop-The-World” 。

G1 使用**非连续空间**，这使 G1 能够有效地处理非常大的堆。此外，G1 可以同时收集年轻代和年老代。G1 并没有将 Java 堆分成三个空间（ Eden、Survivor 和 Old ），而是将堆分成许多（通常是几百个）非常小的区域。这些区域是固定大小的（默认情况下大约为 2MB ）。每个区域都分配给一个空间。G1 收集器的 Java 堆如下图所示：



图上的 U 表示 “未分配” 区域。G1 将堆拆分成小的区域，一个最大的好处是可以做局部区域的垃圾回收，而不需要每次都回收整个区域比如年轻代和年老代，这样回收的停顿时间会比较短。具体的收集过程是：

将所有存活的对象将从**收集的区域**复制到**未分配的区域**，比如收集的区域是 Eden 空间，把 Eden 中的存活对象复制到未分配区域，这个未分配区域就成了 Survivor 空间。理想

情况下，如果一个区域全是垃圾（意味着一个存活的对象都没有），则可以直接将该区域声明为“未分配”。

为了优化收集时间，G1 总是优先选择垃圾最多的区域，从而最大限度地减少后续分配和释放堆空间所需的工作量。这也是 G1 收集器名字的由来——Garbage-First。

## GC 调优原则

GC 是有代价的，因此我们调优的根本原则是**每一次 GC 都回收尽可能多的对象**，也就是减少无用功。因此我们在做具体调优的时候，针对 CMS 和 G1 两种垃圾收集器，分别有一些相应的策略。

### CMS 收集器

对于 CMS 收集器来说，最重要的是**合理地设置年轻代和年老代的大小**。年轻代太小的话，会导致频繁的 Minor GC，并且很有可能存活期短的对象也不能被回收，GC 的效率就不高。而年老代太小的话，容纳不下从年轻代过来的新对象，会频繁触发单线程 Full GC，导致较长时间的 GC 暂停，影响 Web 应用的响应时间。

### G1 收集器

对于 G1 收集器来说，我不推荐直接设置年轻代的大小，这一点跟 CMS 收集器不一样，这是因为 G1 收集器会根据算法动态决定年轻代和年老代的大小。因此对于 G1 收集器，我们需要关心的是 Java 堆的总大小（`-Xmx`）。

此外 G1 还有一个较关键的参数是`-XX:MaxGCPauseMillis = n`，这个参数是用来限制最大的 GC 暂停时间，目的是尽量不影响请求处理的响应时间。G1 将根据先前收集的信息以及检测到的垃圾量，估计它可以立即收集的最大区域数量，从而尽量保证 GC 时间不会超出这个限制。因此 G1 相对来说更加“智能”，使用起来更加简单。

## 内存调优实战

下面我通过一个例子实战一下 Java 堆设置得过小，导致频繁的 GC，我们将通过 GC 日志分析工具来观察 GC 活动并定位问题。

1. 首先我们建立一个 Spring Boot 程序，作为我们的调优对象，代码如下：

```

1 @RestController
2 public class GcTestController {
3
4     private Queue<Greeting> objCache = new ConcurrentLinkedDeque<>();
5
6     @RequestMapping("/greeting")
7     public Greeting greeting() {
8         Greeting greeting = new Greeting("Hello World!");
9
10        if (objCache.size() >= 200000) {
11            objCache.clear();
12        } else {
13            objCache.add(greeting);
14        }
15        return greeting;
16    }
17 }
18
19 @Data
20 @AllArgsConstructor
21 class Greeting {
22     private String message;
23 }

```

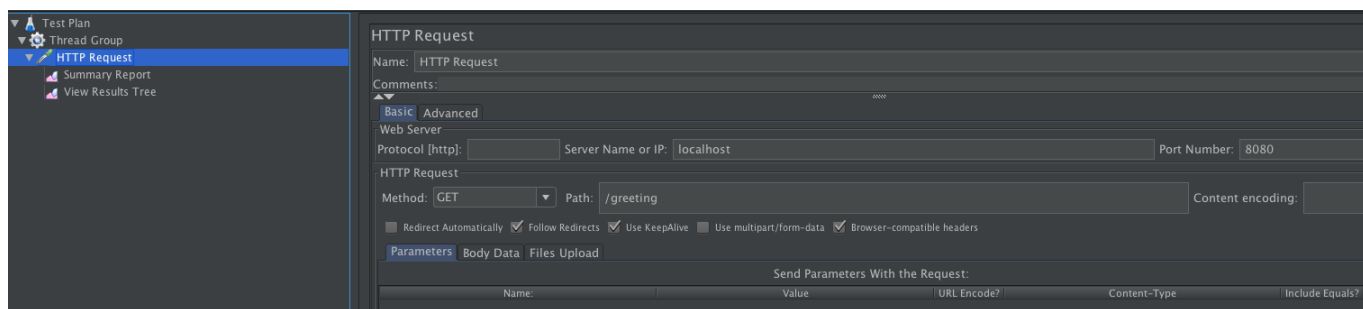
上面的代码就是创建了一个对象池，当对象池中的对象数到达 200000 时才清空一次，用来模拟年老对象。

2. 用下面的命令启动测试程序：

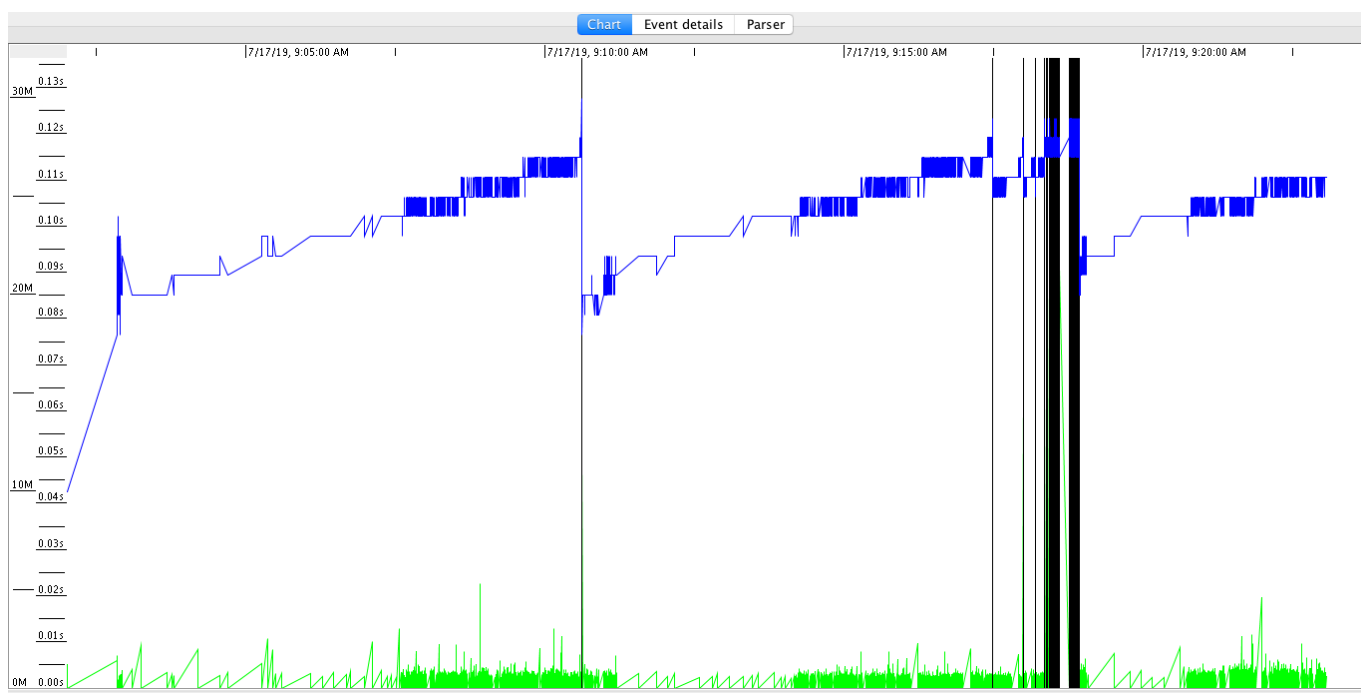
```
1 java -Xmx32m -Xss256k -verbosegc -Xlog:gc*,gc+ref=debug,gc+heap=debug,gc+age=trace:file:
```

我给程序设置的堆的大小为 32MB，目的是能让我们看到 Full GC。除此之外，我还打开了 verbosegc 日志，请注意这里我使用的版本是 Java 12，默认的垃圾收集器是 G1。

3. 使用 JMeter 压测工具向程序发送测试请求，访问的路径是 /greeting。



4. 使用 GCViewer 工具打开 GC 日志，我们可以看到这样的图：



我来解释一下这张图：

图中上部的蓝线表示已使用堆的大小，我们看到它周期的上下震荡，这是我们的对象池要扩展到 200000 才会清空。

图底部的绿线表示年轻代 GC 活动，从图上看到当堆的使用率上去了，会触发频繁的 GC 活动。

图中的竖线表示 Full GC，从图上看到，伴随着 Full GC，蓝线会下降，这说明 Full GC 收集了年老代中的对象。

基于上面的分析，我们可以得出一个结论，那就是 Java 堆的大小不够。我来解释一下为什么得出这个结论：


GC 活动频繁：年轻代 GC（绿色线）和年老代 GC（黑色线）都比较密集。这说明内存空间不够，也就是 Java 堆的大小不够。

Java 的堆中对象在 GC 之后能够被回收，说明不是内存泄漏。

我们通过 GCViewer 还发现累计 GC 暂停时间有 55.57 秒，如下图所示：

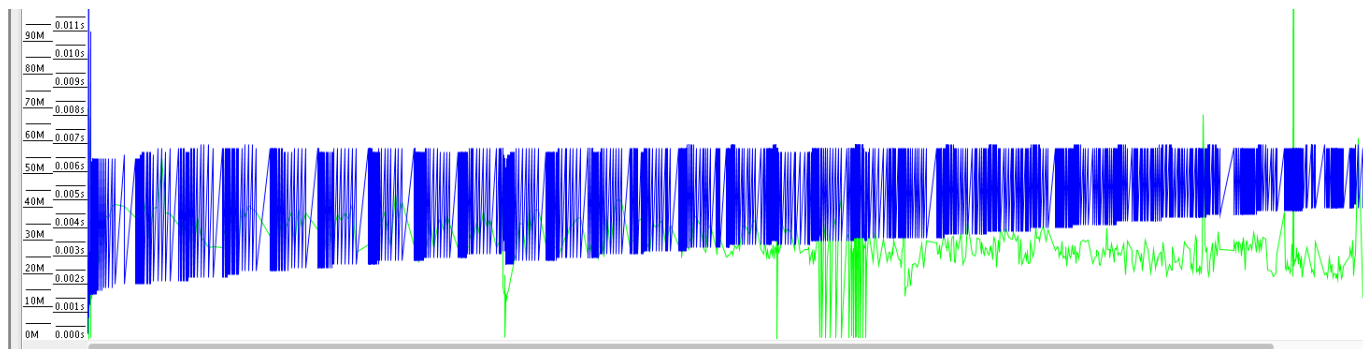
Total pause	
Accumulated pauses	55.57s
Number of pauses	24170
Avg Pause	0.0023s ( $\sigma=0.0061$ )
Min / Max Pause	0.00005s / 0.1364s
Avg pause interval	0.07312s ( $\sigma=0.45212$ )
Min / max pause interval	0.003s / 49.493s
Full gc pauses	
Accumulated full GC	12.45s (22.4%)
Number of full gc pauses	180
Avg full GC	0.06915s ( $\sigma=0.01831$ )
Min / max full gc pause	0.03581s / 0.1364s
Min / max full gc pause interval	0.055s / 412.117s
Gc pauses	
Accumulated GC	43.12s (77.6%)
Number of gc pauses	23990
Avg GC	0.0018s ( $\sigma=0.00107$ )
Min / max gc pause	0.00005s / 0.02876s

因此我们的解决方案是调大 Java 堆的大小，像下面这样：

 复制代码

```
1 java -Xmx2048m -Xss256k -verbosegc -Xlog:gc*,gc+ref=debug,gc+heap=debug,gc+age=trace:fi:
```

生成的新的 GC log 分析图如下：



你可以看到，没有发生 Full GC，并且年轻代 GC 也没有那么频繁了，并且累计 GC 暂停时间只有 3.05 秒。

Summary		Memory	Pause
Total pause			
Accumulated pauses		3.05s	
Number of pauses		938	
Avg Pause	0.00325s ( $\sigma=0.00131$ )		
Min / Max Pause	0.00007s / 0.03054s		
Avg pause interval	1.6564s ( $\sigma=1.73862$ )		
Min / max pause interval	0.026s / 15.604s		
Full gc pauses			
Accumulated full GC		0s (0.0%)	
Number of full gc pauses		0	
Avg full GC		n/a	
Min / max full gc pause		n/a	
Min / max full gc pause interval		n/a	
Gc pauses			
Accumulated GC		3.05s (100.0%)	
Number of gc pauses		938	
Avg GC	0.00325s ( $\sigma=0.00131$ )		
Min / max gc pause	0.00007s / 0.03054s		

## 本期精华

今天我们首先回顾了 CMS 和 G1 两种垃圾收集器背后的设计思路以及它们的区别，接着分析了 GC 调优的总体原则。

对于 CMS 来说，我们要合理设置年轻代和年老代的大小。你可能会问该如何确定它们的大小呢？这是一个迭代的过程，可以先采用 JVM 的默认值，然后通过压测分析 GC 日志。

如果我们看年轻代的内存使用率处在高位，导致频繁的 Minor GC，而频繁 GC 的效率又不高，说明对象没那么快能被回收，这时年轻代可以适当调大一点。

如果我们看年老代的内存使用率处在高位，导致频繁的 Full GC，这样分两种情况：如果每次 Full GC 后年老代的内存占用率没有下来，可以怀疑是内存泄漏；如果 Full GC 后年老代的内存占用率下来了，说明不是内存泄漏，我们要考虑调大年老大。

对于 G1 收集器来说，我们可以适当调大 Java 堆，因为 G1 收集器采用了局部区域收集策略，单次垃圾收集的时间可控，可以管理较大的 Java 堆。

## 课后思考

如果把年轻代和年老代都设置得很大，会有什么问题？



不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。



# 深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 特别放送 | 如何持续保持对学习的兴趣？

下一篇 35 | 如何监控Tomcat的性能？

## 精选留言 (16)

写留言



-W.LI-

2019-07-30

年轻代设置过大:

- 1.生命周期长的对象会长时间停留在年轻代，在S0和S1来回复制，增加复制开销。
- 2.年轻代太大会增加YGC每次停顿的时间，不过通过根节点遍历，OopMap，old scan等优化手段这一部分的开销其实比较少。
- 3.浪费内存。内存也是钱啊虽然现在租的很便宜...

展开



8





**-W.LI-**

2019-07-30

李老师好!感觉老师今天偷懒了,CMS负责老年代回收,年轻代一般配合parNew使用。大概啥情况下使用G1比较好啊?之前看见网上说,大堆多核,jdk9以及以上可以使用G1,jdk8的话除非cms满足不了需求不然不建议使用G1。

G1不太了解老师能推荐下资料么?

我觉得工具,可以提高效率,初学者优先搞清楚原理扎实基础比较好。

展开 ∨

作者回复: g1实现很复杂,有人专门写了本书来讲,原理入门可以看看这篇文章:

<https://yq.aliyun.com/articles/444436>

1

3



**业余草**

2019-07-30

需要实际操作一遍,光看是记不住的,过一段时间就忘记了。

1

2



**nightmare**

2019-07-30

分情况,如果是G1大年轻代和大老年代没什么问题 如果是cms parnew的话 也需要看情况 如果你的并发比较大并且很快占满eden区 或者 用jstat监控 supervisor区占比一直高于百分之70这个时候 这个时候加大新生代就没有什么问题 如果要很久才占满eden区 或者 supervisor区占比比较小 这个时候就要把 新生代 设置小一点 减少新生代回收时间 老年代也要看年轻代晋升到老年代平均占多大 如果晋升很快并且对象占比较大 大一点没问题 否...

展开 ∨

1

1



**QQ怪**

2019-07-30

设置过大回收频率降低,单次回收的对象量大,回收stw时间过长,设置大也不好,过小也不好,设置适合的才是最好的

展开 ∨

1

1



**Geek\_ebda96**

2019-08-03

老师您好,请问对于新生代的内存,supervisor和eden区域,大小比例怎样设置合理,这个比例是否对GC性能有影响呢?

展开 ▾



**CN8818**

2019-08-01

执行命令 : java -Xmx2048m -Xss256k -verbosegc -Xlog:gc\*,gc+ref=debug,gc+heap=debug,gc+age=trace:file=gc-%p-%t.log:tags,uptime,time,level:filecount=2,filesize=100m -jar target/demo-0.0.1-SNAPSHOT.jar...

展开 ▾

作者回复: 要采用最新jvm版本12



**月如钩**

2019-08-01

实操很重要呀，朋友们，纸上谈兵很容易忘

展开 ▾



**xj\_zh**

2019-07-31

老师，可以讲一讲undertow吗，为什么spring boot 2.0 选择undertow做为默认WEB容器。



**双月鸟**

2019-07-30

CMS默认开启-XX:+UseAdaptiveSizePolicy，所以G1收集器会根据算法动态决定年轻和年老代的大小不能成为G1的优势

展开 ▾



**弃**

2019-07-30

谢谢老师。

展开 ▾



许童童

2019-07-30

老师讲得好啊，虽然工作中没有用到Java，但读了这篇文章也基本懂了！



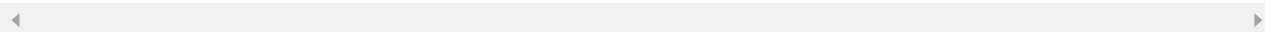
锦

2019-07-30

对于CMS来说，设置很大的堆内存，在导致单次STW时间长，会导致服务不可用，定时器出问题？对响应敏感的系统来说不太友好，但堆内存设置太小又会导致频道GC，所以需要综合评估。那么如何使用超大机器内存呢？可以使用集群方式部署，单个应用设置较小的堆内存。

对于G1来说，文中有提到可以设置较大内存，因为G1是局部收集，但极端情况下，区域...  
展开

作者回复: 随着java9 普及开，就默认用g1了



2

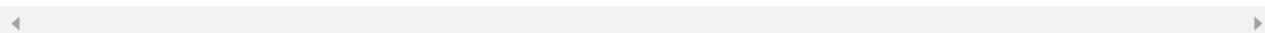


弃

2019-07-30

老师，我想问个问题:在docker中运行的springboot(使用默认的tomcat容器)，如何查看tomcat的gc日志？

作者回复: 可以登到容器内去看，不过一般会把容器的日志目录mount到主机的某个目录



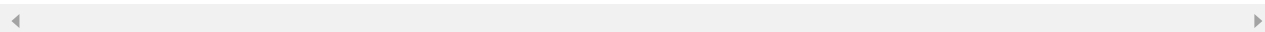
新世界

2019-07-30

设置过大，回收频率会降低，导致单次回收时间过长，因为需要回收的对象更多，导致GC stop the world时间过长，卡顿明显，导致请求无法及时处理

展开

作者回复: 对的，主要是会引起gc停顿时间过长



1





a、

2019-07-30

年老代如果设置过大，会导致full gc时间过长,full gc需要stop-the-world，程序会出现长时间停顿。如果年轻代设置过大，因为年轻代用的是标记-复制算法，所以会出现需要复制大量数据的情况，也需要stop-the-world，所以也会出现长时间停顿

展开 ∨



1

