

JVM- 技术专题 -GCViewer 调优 GC



李博@Alex

[+关注](#)

发布于: 2020 年 08 月 22 日



使用GCViewer调优GC

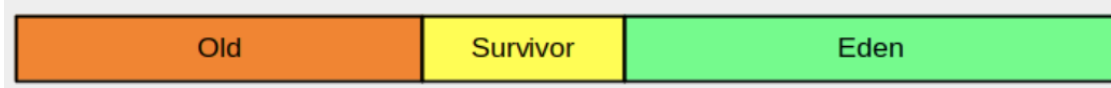
在对 GC 调优的过程中，我们不仅需要知道 GC 的原理，更重要的是要熟练使用各种监控和分析工具，具备 GC 调优的实战能力。

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：

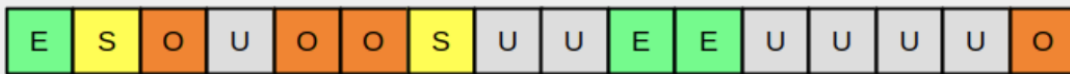


image.png

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：

 复制代码

```
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    }
16 }
```

```
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=gc-%p-%t.lc
```

[复制代码](#)

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：

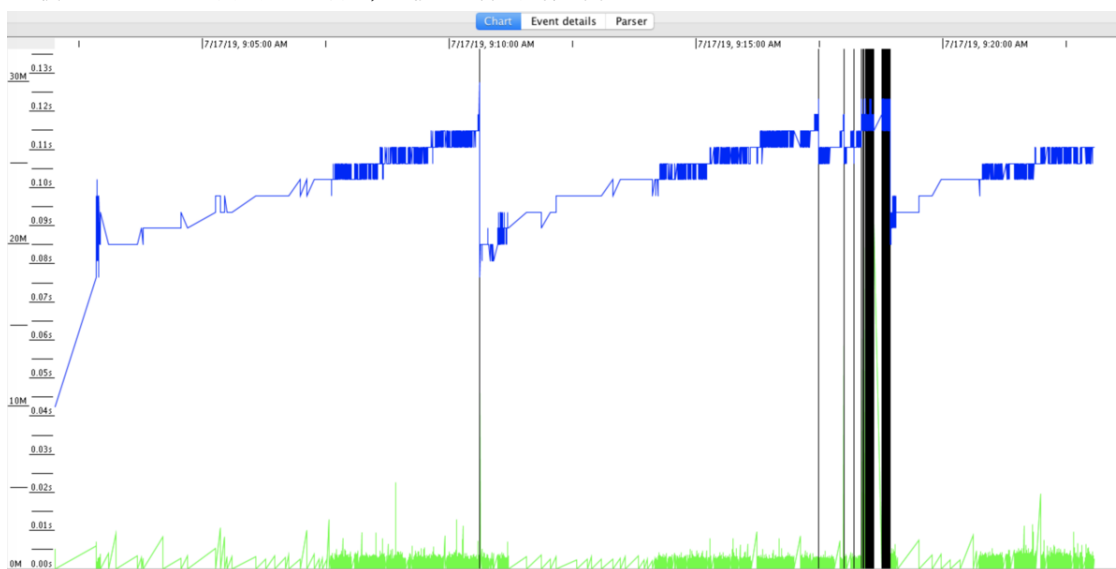


image.png

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

image.png

因此我们的解决方案是调大 Java 堆的大小，像下面这样：

```
1 java -Xmx2048m -Xss256k -verbosegc -Xlog:gc*,gc+ref=debug,gc+heap=debug,gc+age=trace:file=gc-%p-%t.  
2
```

 复制代码

生成的新的 GC log 分析图如下：

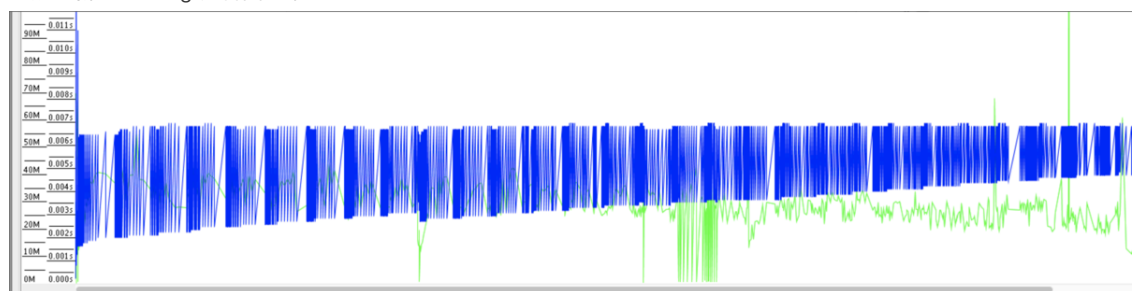


image.png

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。

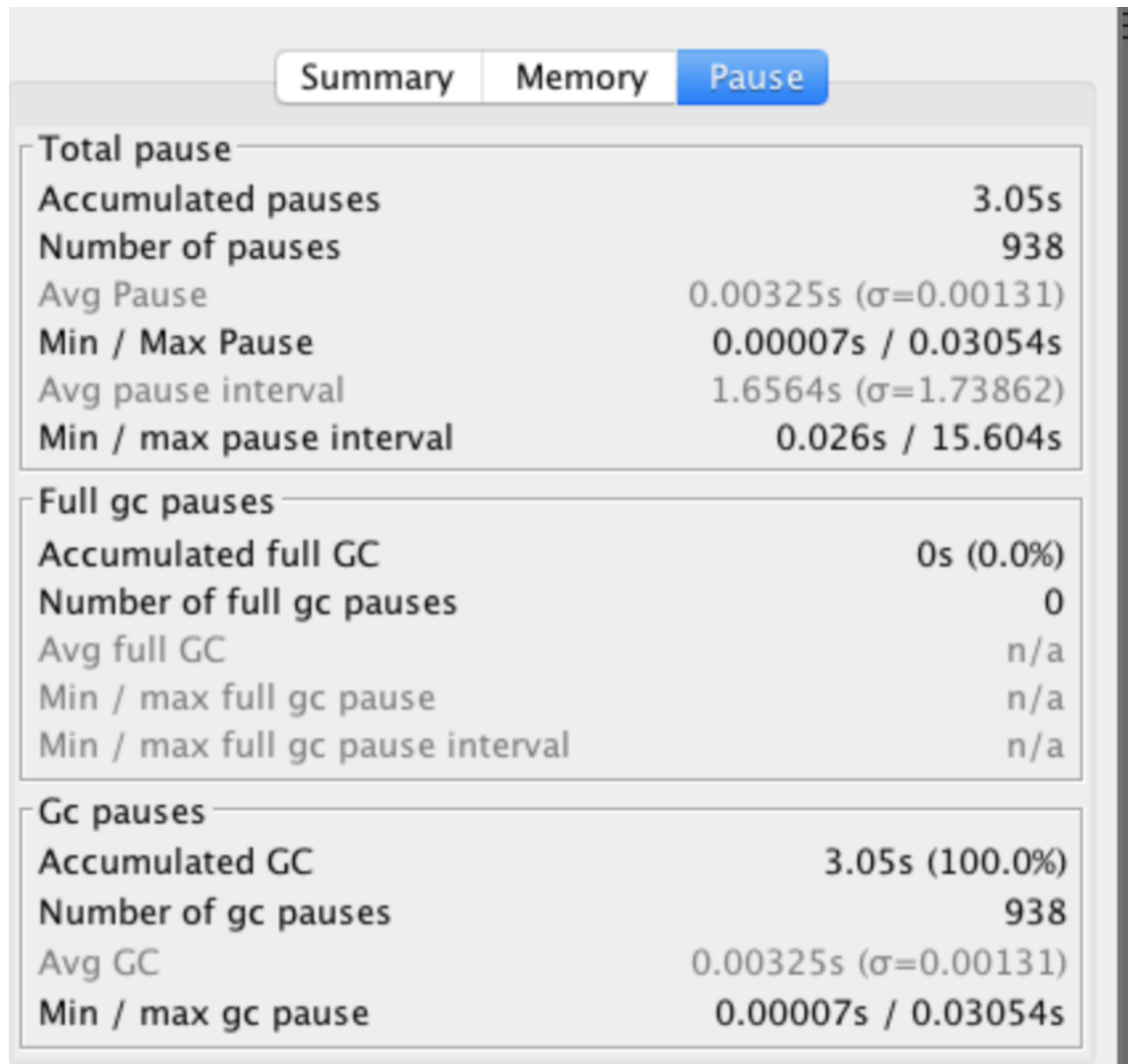


image.png

总结

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。

发布于: 2020 年 08 月 22 日 | 阅读数: 224

JVM

**李博@Alex**[+ 关注](#)

我们始于迷惘，终于更高的迷惘。 2020.03.25 加入
一个酷爱计算机技术、健身运动、悬疑推理的极客狂人

[👍 点赞](#) | [★ 收藏](#) | [💬 微信](#) | [🐦 微博](#) | [🏠 部落](#) | [🚩 举报](#)

评论

快抢沙发！虚位以待

发布

暂无评论



促进软件开发及相关领域知识与创新的传播

商务专区

AWS 技术专区 Intel 精彩芯体验 Google Cloud
华为云5G+X联创营 华为云 DevRun 专区 Intel AI
腾讯Techo开发者大会 百度技术沙龙
T11数据智能峰会 迅雷链技术专区
OPPO技术开放日 云+社区开发者大会
百度AI 开放平台 华为云 MeetUp
华为昇腾技术沙龙 DevRun鲲鹏开发者沙龙2020

InfoQ

[关于我们](#)
[我要投稿](#)
[合作伙伴](#)
[加入我们](#)
[关注我们](#)

联系我们

内容投稿: editors@geekbang.com
业务合作: hezuo@geekbang.com
反馈投诉: feedback@geekbang.com
加入我们: zhaopin@geekbang.com
联系电话: 010-64738142
地址: 北京市朝阳区叶青大厦北园

InfoQ 近期会议

[深圳](#) 全球架构师峰会 09月11-12日
[上海](#) 全球人工智能与机器学习技术大会 09月24-25日
[北京](#) 全球软件开发大会 10月15-17日
[北京](#) 全球大前端技术大会 11月24-25日