重大事故！线上系统频繁卡死，凶手竟然是 Full GC ？

转载

csdn业界要闻 于 2020-07-23 18:44:59 发布 211 收藏
文章标签： 可视化 java 编程语言 jvm 人工智能
原文链接：https://mp.weixin.qq.com/s?__biz=MzIxODM4ODg1Mg==\x26amp;mid=2247483979\x26amp;idx=1\x26amp;sn=3dd34f16a3865e9b534d4b2a7797442c\x26amp;chksm=97ea0250a

来源 | 程序员大帝

责编 | 晋兆雨

头图 | CSDN付费下载自视觉中国

案发现场

通常来说，一个系统在上线之前应该经过多轮的调试，在测试服务器上稳定的运行过一段时间。我们知道 Full GC 会导致 Stop The World 情况的出现，严重影响性能，所以一个性能良好的 JVM，应该几天才会发生一次 Full GC，或者最多一天几次而已。

但是昨天晚上突然收到短信通知，显示线上部署的四台机器全部卡死，服务全部不可用，赶紧查看问题!

涉及到类似的错误，最开始三板斧肯定是查看 JVM 的情况。很多中小型公司没有建立可视化的监控平台，比如Zabbix、Ganglia、Open-Falcon、Prometheus等等，没办法直接可视化看到JVM各个区域的内存变化，GC次数和GC耗时。

不过不用怕，咱们用 jstat 这种工具也可以。言归正传，排查了线上问题之后，发现竟然是服务器上面，JVM 这段时间疯狂 Full GC，一天搞了几十次，直接把系统卡死了！

排查现场

破案之前，我们先来要保护下案发现场并进行排查。

机器配置	2核 4G
JVM堆内存大小	2G
系统运行时间	1天
Full GC出现次数和单次耗时	48次，300ms左右
Young GC出现次数和单次耗时	4000多次，50ms

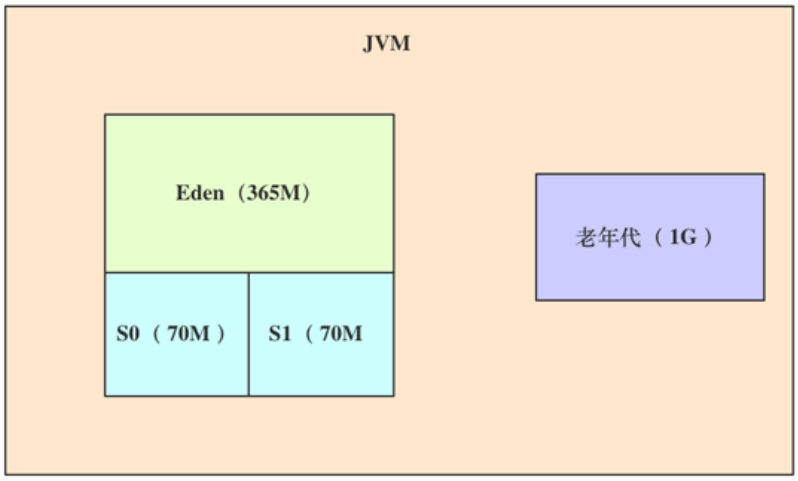
这样看起来，系统的性能是相当差了，一分钟三次 Young GC，每半小时就一次 Full GC，接下来我们再看看 JVM 的参数。可能有的同学每次见到这么多参数都会头大，但其实每一个参数的背后都会透漏着蛛丝马迹。我这里摘取关键参数如下：

CMSInitiatingOccupancyFraction	62
SurvivorRatio	5
Xmx	1536M
Xmx	1536M

简单解读一下，根据参数可以看出来，这台 4G 的机器上为 JVM 的堆内存是设置了 1.5G 左右的大小。新生代和老年代默认会按照 1：2 的比例进行划分，分别对应 512M 和 1G。

一个重要的参数 XXiSurvivorRatio 设置为5，它所代表的是新生代中Eden：Survivor1 ：Survivor2的比例是 5:1:1。所以此时Eden区域大致为365M，每个Survivor区域大致为70MB。

还有一个非常关键的参数，那就是 CMSInitiatingOccupancyFraction 设置为了62。它意味着一旦老年代内存占用达到62%，也就是存在 620MB 左右对象时，就会触发一次 Full GC。此时整个系统的内存模型图如下所示：



03

还原现场

根据对案发现场的排查，我们可以还原线上系统的 GC 运行情况了，分析一下线上的 JVM 到底出现了什么状况。

首先我们知道每分钟会发生 3 次 Young GC，说明系统运行 20 秒后就会把 Eden 区塞满，Eden 区一共有 356MB 的空间，因此平均下来系统每秒钟会产生 20MB 左右大小的对象。

接着我们根据每半小时触发一次 Full GC 的推断，以及 “-XX:CMSmitiatingOccupancyFraction=62” 参数的设置，老年代有 1G 的空间，所以应该是在老年代有 600多MB 左右的对象时就会触发一次 Full GC。

看到这里，有的同学可能立刻下结论，觉得肯定是因为 Survivor 区域太小了，导致 Young GC 后的存活对象太多放不下，就一直有对象流入老年代，进而导致后来触发的 Full GC ？

实际上分析到这里，绝对不能草率下这个判断。

因为老年代里为什么有那么多的对象？确实有可能是因为每次 Young GC后的存活对象较多，Survivor区域太小，放不下了。

但也有可能是长时间存活的对象太多了，都积累在老年代里，始终回收不掉，进而导致老年代很容易就达到 62% 的占比触发 Full GC，所以我们还要有更多的证据去验证我们的判断。

04

破案开始

老年代里到底为什么会有那么多的对象？

面对这个问题，说句实话，仅仅根据可视化监控和推论是绝对没法往下分析了，因为我们并不知道老年代里到底为什么会有那么多的对象。这个时候就有必要让线上系统重新运行，借助 jstat 工具实时去观察 JVM 实际的运行情况。这个过程非常类似警察叔叔在破案时，会假设自己是凶手，尝试再现当时的场景。

这里省略具体的 jstat 工具操作过程，如果大家没有接触过百度下即可，非常简单。通过 jstat 的观察，我们当时可以看到，每次 Young GC 过后升入老年代里的对象其实很少。

看到这个现象，我起初也很奇怪。因为通过 jstat 的追踪观察，并不是每次Young GC 后都有 几十MB 对象进入老年代的，而是偶尔一次 Young GC 才会有 几十MB 对象进入老年代！

所以正常来说，既然没有对象从新生代过渡到老年代，那么老年代就不至于快速的到达 62% 的占有率，从而导致 Full GC。那么老年代触发 Full GC 时候的几百 MB 对象到底从哪里来的？

仔细一想，其实答案已经呼之欲出了，那就是大对象！

一定是系统运行的时候，每隔一段时间就会突然产生几百 MB 的大对象，由于新生代放不下，所以会直接进入老年代，而不会走 Eden 区域。然后再配合上年轻代还偶尔会有 Young GC 后几十 MB 对象进入老年代，所以不停地触发Full GC！

抓捕真凶

分析到这里，后面的过程就很简单了，我们可以通过 jmap 工具，dump 出内存快照，然后再用 jhat 或者 Visual VM 之类的可视化工具来分析就可以了。

通过内存快照的分析，直接定位出来那个几百MB的大对象，发现竟然是个Map之类的数据结构，这是什么鬼？

返回头去开始撸代码，发现是从数据库里查出来的数据存放在了这个Map里，没有好办法，再继续地毯式排查。

最后发现竟然是有条坑爹的 SQL 语句没加 where条件！！不知道是手滑还是忘了，测试的时候这个分支也没走到（*画外音：这段代码的开发和测试都不是我*  ）

没有 where 条件，每次查询肯定会有超出预期的大量数据，导致了每隔一段时间就会搞出几个上百 MB 的大对象，这些对象全部直接进入老年代，然后触发 Full GC！

善后处理

破案定位嫌疑人最困难，在知道凶手后，靠着满大街的摄像头，抓人就是分分钟的事情。所以我们排查到这里，这个案例如何解决已经非常简单了。

（1）解决代码中的 bug，把错误的 SQL 修复，一劳永逸彻底解决这几百 MB 大对象进入老年代的问题。

（2）以防万一，对堆内存空间进行扩容，然后再把-XX:CMSInitiatingOccupancyFraction 参数从 62 调高，使得老年代的使用率更高时才会触发 Full GC。

这两个步骤优化完毕之后，线上系统基本上表现就非常好了。

总结

本文通过一个线上系统卡死的现象，详细地定位并剖析了产生问题的原因。也证明了要成为一个优秀的程序员，不光对语言本身要有所了解，还要对 JVM 调优这样偏底层的知识有所涉猎，这对排查问题会有非常大的帮助。同时完善的监控非常重要，通过提前告警，可以将问题扼杀在摇篮里！

THE END

推荐阅读

架构师们，怎么走着走着就变“烟囱”了呢？| 文末含福利

没想到 Hash 冲突还能这么玩，你的服务中招了吗？

“自由主义教皇”、Linux 之父的封神之路

微信回应“取消两分钟内删除功能”；甲骨文裁撤北京中心；Redis 6.0.6 发布| 极客头条

周志华教授力作，豆瓣10分好评，集成学习如何破解AI实践难题 | 赠书