https://mp.weixin.qq.com/s/P8s3kuceBNovUP5adXpFCQ

# 线上服务的FGC问题排查

线上服务的GC问题，是Java程序非常典型的一类问题，非常考验工程师排查问题的能力。同时，几乎是面试必考题，但是能真正答好此题的人并不多，要么原理没吃透，要么缺乏实战经验。

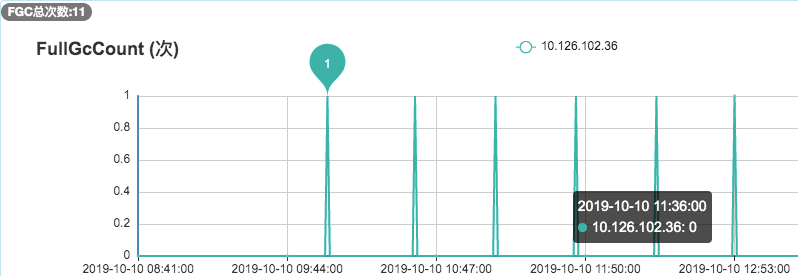
过去半年时间里，我们的广告系统出现了多次和GC相关的线上问题，有Full GC过于频繁的，有Young GC耗时过长的，这些问题带来的影响是：GC过程中的程序卡顿，进一步导致服务超时从而影响到广告收入。

这篇文章，我将以一个FGC频繁的线上案例作为引子，详细介绍下GC的排查过程，另外会结合GC的运行原理给出一份实践指南，希望对你有所帮助。内容分成以下3个部分：

* 从一次FGC频繁的线上案例说起
* GC的运行原理介绍
* 排查FGC问题的实践指南

## 01 从一次FGC频繁的线上案例说起

去年10月份，我们的广告召回系统在程序上线后收到了FGC频繁的系统告警，通过下面的监控图可以看到：平均每35分钟就进行了一次FGC。而程序上线前，我们的FGC频次大概是2天一次。下面，详细介绍下该问题的排查过程。



1. **检查JVM配置** 通过以下命令查看JVM的启动参数：ps aux | grep "applicationName=adsearch"

-Xms4g -Xmx4g -Xmn2g -Xss1024K

-XX:ParallelGCThreads=5

-XX:+UseConcMarkSweepGC

-XX:+UseParNewGC

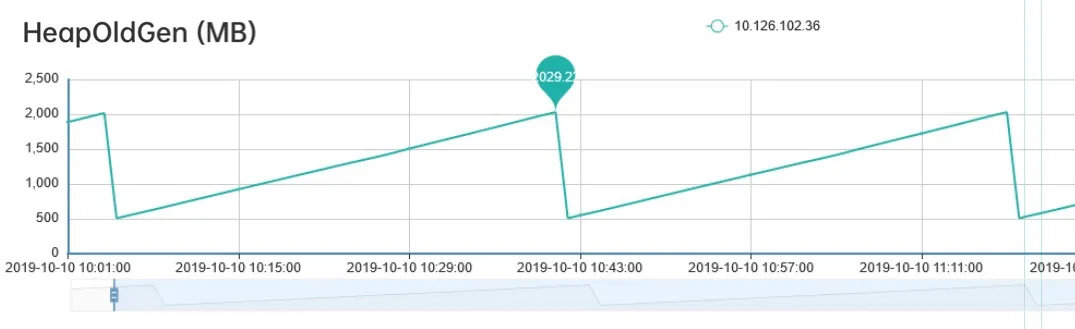
-XX:+UseCMSCompactAtFullCollection

-XX:CMSInitiatingOccupancyFraction=80

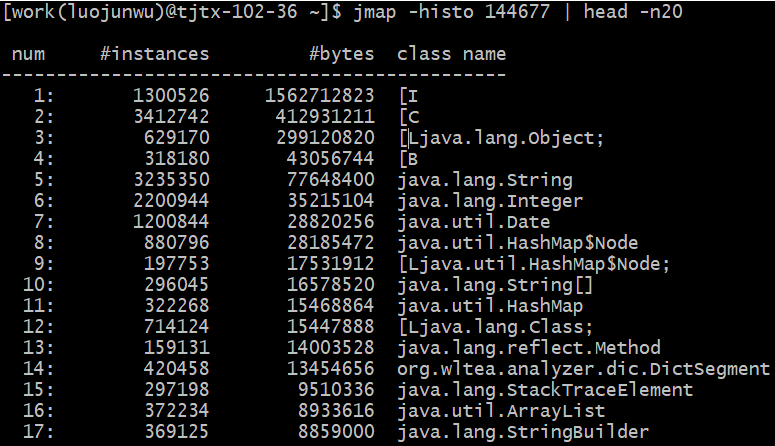
可以看到堆内存为4G，新生代为2G，老年代也为2G，新生代采用ParNew收集器，老年代采用并发标记清除的CMS收集器，当老年代的内存占用率达到80%时会进行FGC。进一步通过 jmap -heap 7276 | head -n20 可以得知新生代的Eden区为1.6G，S0和S1区均为0.2G。

**2. 观察老年代的内存变化**

通过观察老年代的使用情况，可以看到：每次FGC后，内存都能回到500M左右，因此我们排除了内存泄漏的情况。



1. **通过jmap命令查看堆内存中的对象**  
   通过命令 jmap -histo 7276 | head -n20



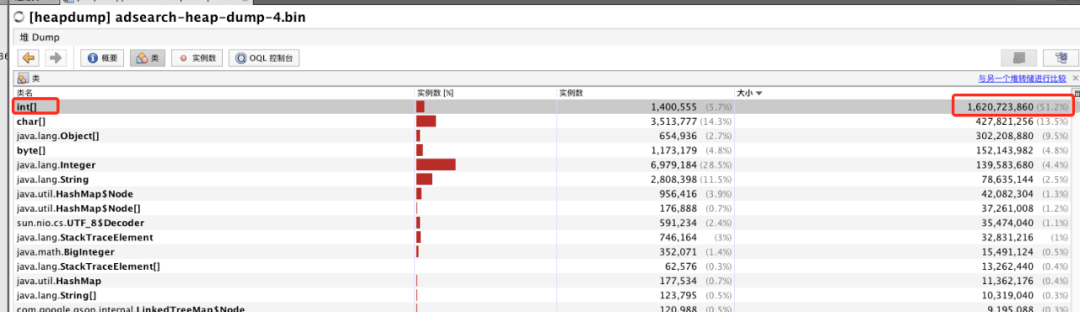
上图中，按照对象所占内存大小排序，显示了存活对象的实例数、所占内存、类名。可以看到排名第一的是：int[]，而且所占内存大小远远超过其他存活对象。至此，我们将怀疑目标锁定在了 int[] .

**4. 进一步dump堆内存文件进行分析**

锁定 int[] 后，我们打算dump堆内存文件，通过可视化工具进一步跟踪对象的来源。考虑堆转储过程中会暂停程序，因此我们先从服务管理平台摘掉了此节点，然后通过以下命令dump堆内存：

jmap -dump:format=b,file=heap 7276

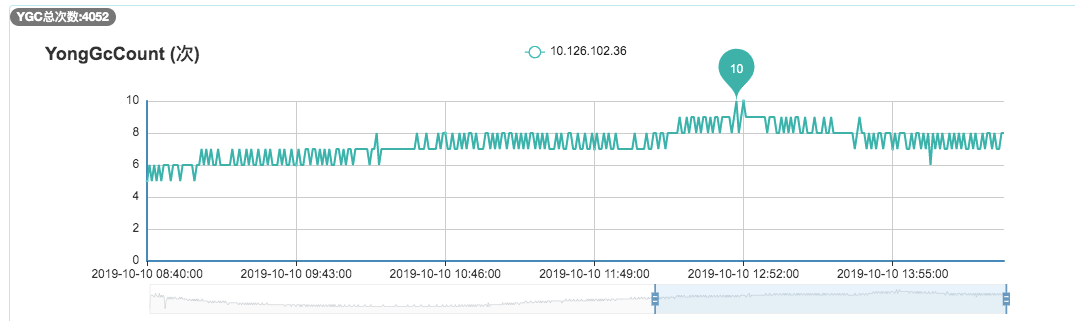
通过JVisualVM工具导入dump出来的堆内存文件，同样可以看到各个对象所占空间，其中int[]占到了50%以上的内存，进一步往下便可以找到 int[] 所属的业务对象，发现它来自于架构团队提供的codis基础组件。



**5. 通过代码分析可疑对象**

通过代码分析，codis基础组件每分钟会生成约40M大小的int数组，用于统计TP99 和 TP90，数组的生命周期是一分钟。而根据第2步观察老年代的内存变化时，发现老年代的内存基本上也是每分钟增加40多M，因此推断：这40M的int数组应该是从新生代晋升到老年代。

我们进一步查看了YGC的频次监控，通过下图可以看到大概1分钟有8次左右的YGC，这样基本验证了我们的推断：因为CMS收集器默认的分代年龄是6次，即YGC 6次后还存活的对象就会晋升到老年代，而codis组件中的大数组生命周期是1分钟，刚好满足这个要求。



至此，整个排查过程基本结束了，那为什么程序上线前没出现此问题呢？通过上图可以看到：程序上线前YGC的频次在5次左右，此次上线后YGC频次变成了8次左右，从而引发了此问题。

**6. 解决方案**

为了快速解决问题，我们将CMS收集器的分代年龄改成了15次，改完后FGC频次恢复到了2天一次，后续如果YGC的频次超过每分钟15次还会再次触发此问题。当然，我们最根本的解决方案是：优化程序以降低YGC的频率，同时缩短codis组件中int数组的生命周期，这里就不做展开了。

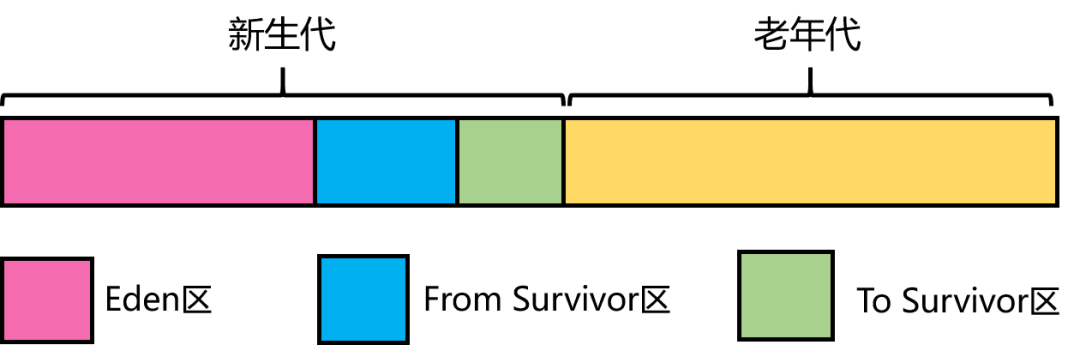
## 02 GC的运行原理介绍

上面整个案例的分析过程中，其实涉及到很多GC的原理知识，如果不懂得这些原理就着手处理，其实整个排查过程是很抓瞎的。

这里，我选择几个最核心的知识点，展开介绍下GC的运行原理，最后再给出一份实践指南。

### 1. 堆内存结构

大家都知道: GC分为YGC和FGC，它们均发生在JVM的堆内存上。先来看下JDK8的堆内存结构：



可以看到，堆内存采用了分代结构，包括新生代和老年代。新生代又分为：Eden区，From Survivor区（简称S0），To Survivor区（简称S1区），三者的默认比例为8:1:1。另外，新生代和老年代的默认比例为1:2。

堆内存之所以采用分代结构，是考虑到绝大部分对象都是短生命周期的，这样不同生命周期的对象可放在不同的区域中，然后针对新生代和老年代采用不同的垃圾回收算法，从而使得GC效率最高。

### 2. YGC是什么时候触发的

大多数情况下，对象直接在年轻代中的Eden区进行分配，**如果Eden区域没有足够的空间，那么就会触发YGC**（Minor GC），YGC处理的区域只有新生代。因为大部分对象在短时间内都是可收回掉的，因此YGC后只有极少数的对象能存活下来，而被移动到S0区（采用的是复制算法）。当触发下一次YGC时，会将Eden区和S0区的存活对象移动到S1区，同时清空Eden区和S0区。当再次触发YGC时，这时候处理的区域就变成了Eden区和S1区（即S0和S1进行角色交换）。每经过一次YGC，存活对象的年龄就会加1。

### 3. FGC又是什么时候触发的

下面4种情况，对象会进入到老年代中：

* YGC时，To Survivor区不足以存放存活的对象，对象会直接进入到老年代。
* 经过多次YGC后，如果存活对象的年龄达到了设定阈值，则会晋升到老年代中。
* 动态年龄判定规则，To Survivor区中相同年龄的对象，如果其大小之和占到了 To Survivor区一半以上的空间，那么大于此年龄的对象会直接进入老年代，而不需要达到默认的分代年龄。
* 大对象：由-XX:PretenureSizeThreshold启动参数控制，若对象大小大于此值，就会绕过新生代, 直接在老年代中分配。

当晋升到老年代的对象大于了老年代的剩余空间时，就会触发FGC（Major GC），FGC处理的区域同时包括新生代和老年代。除此之外，还有以下4种情况也会触发FGC：

* 老年代的内存使用率达到了一定阈值（可通过参数调整），直接触发FGC。
* 空间分配担保：在YGC之前，会先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间。如果小于，说明YGC是不安全的，则会查看参数 HandlePromotionFailure 是否被设置成了允许担保失败，如果不允许则直接触发Full GC；如果允许，那么会进一步检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果小于也会触发 Full GC。
* Metaspace（元空间）在空间不足时会进行扩容，当扩容到了-XX:MetaspaceSize 参数的指定值时，也会触发FGC。
* System.gc() 或者Runtime.gc() 被显式调用时，触发FGC。

### 4. 在什么情况下，GC会对程序产生影响？

不管YGC还是FGC，都会造成一定程度的程序卡顿（即Stop The World问题：GC线程开始工作，其他工作线程被挂起），即使采用ParNew、CMS或者G1这些更先进的垃圾回收算法，也只是在减少卡顿时间，而并不能完全消除卡顿。

那到底什么情况下，GC会对程序产生影响呢？根据严重程度从高到底，我认为包括以下4种情况：

* FGC过于频繁：FGC通常是比较慢的，少则几百毫秒，多则几秒，正常情况FGC每隔几个小时甚至几天才执行一次，对系统的影响还能接受。但是，一旦出现FGC频繁（比如几十分钟就会执行一次），这种肯定是存在问题的，它会导致工作线程频繁被停止，让系统看起来一直有卡顿现象，也会使得程序的整体性能变差。
* YGC耗时过长：一般来说，YGC的总耗时在几十或者上百毫秒是比较正常的，虽然会引起系统卡顿几毫秒或者几十毫秒，这种情况几乎对用户无感知，对程序的影响可以忽略不计。但是如果YGC耗时达到了1秒甚至几秒（都快赶上FGC的耗时了），那卡顿时间就会增大，加上YGC本身比较频繁，就会导致比较多的服务超时问题。
* FGC耗时过长：FGC耗时增加，卡顿时间也会随之增加，尤其对于高并发服务，可能导致FGC期间比较多的超时问题，可用性降低，这种也需要关注。
* YGC过于频繁：即使YGC不会引起服务超时，但是YGC过于频繁也会降低服务的整体性能，对于高并发服务也是需要关注的。

其中，「FGC过于频繁」和「YGC耗时过长」，这两种情况属于比较典型的GC问题，大概率会对程序的服务质量产生影响。剩余两种情况的严重程度低一些，但是对于高并发或者高可用的程序也需要关注。

## 03 排查FGC问题的实践指南

通过上面的案例分析以及理论介绍，再总结下FGC问题的排查思路，作为一份实践指南供大家参考。

### 1. 清楚从程序角度，有哪些原因导致FGC

* 大对象：系统一次性加载了过多数据到内存中（比如SQL查询未做分页），导致大对象进入了老年代。
* 内存泄漏：频繁创建了大量对象，但是无法被回收（比如IO对象使用完后未调用close方法释放资源），先引发FGC，最后导致OOM.
* 程序频繁生成一些长生命周期的对象，当这些对象的存活年龄超过分代年龄时便会进入老年代，最后引发FGC. （即本文中的案例）
* 程序BUG导致动态生成了很多新类，使得 Metaspace 不断被占用，先引发FGC，最后导致OOM.
* 代码中显式调用了gc方法，包括自己的代码甚至框架中的代码。
* JVM参数设置问题：包括总内存大小、新生代和老年代的大小、Eden区和S区的大小、元空间大小、垃圾回收算法等等。

### 2. 清楚排查问题时能使用哪些工具

* 公司的监控系统：大部分公司都会有，可全方位监控JVM的各项指标。
* JDK的自带工具，包括jmap、jstat等常用命令：

# 查看堆内存各区域的使用率以及GC情况  
jstat -gcutil -h20 pid 1000

# 查看堆内存中的存活对象，并按空间排序

jmap -histo pid | head -n20

# dump堆内存文件

jmap -dump:format=b,file=heap pid

* 可视化的堆内存分析工具：JVisualVM、MAT等

### 3. 排查指南

* 查看监控，以了解出现问题的时间点以及当前FGC的频率（可对比正常情况看频率是否正常）
* 了解该时间点之前有没有程序上线、基础组件升级等情况。
* 了解JVM的参数设置，包括：堆空间各个区域的大小设置，新生代和老年代分别采用了哪些垃圾收集器，然后分析JVM参数设置是否合理。
* 再对步骤1中列出的可能原因做排除法，其中元空间被打满、内存泄漏、代码显式调用gc方法比较容易排查。
* 针对大对象或者长生命周期对象导致的FGC，可通过 jmap -histo 命令并结合dump堆内存文件作进一步分析，需要先定位到可疑对象。
* 通过可疑对象定位到具体代码再次分析，这时候要结合GC原理和JVM参数设置，弄清楚可疑对象是否满足了进入到老年代的条件才能下结论。