图文 060、案例实战：一次线上系统每天数十次Full GC导致频繁卡死的优化实战！

详情 评论

案例实战：
一次线上系统每天数十次Full GC导致频繁卡死的优化实战！

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（**ps**：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解）

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

1、案例开始前的说明

今天的这个案例也是我们之前线上系统经历过的一个真实的生产JVM优化案例，这个优化的过程比较复杂，经过了多次优化，当然核心原理和知识其实还是之前给大家讲解过的那些东西。

只不过这个真实生产系统优化的过程大家如果能理解透彻，那么对于大家利用学过的知识和掌握的工具自己去进行JVM优化的时候，肯定是大有好处的。

这个线上系统是一个团队开发的，那个团队开发完一个新系统上线之后发现一天的Full GC次数多大数十次，甚至有的时候会上百次，大家可想而知这是什么概念！

通常来说，我们建议的一个比较良好的JVM性能，应该是Full GC在几天才发生一次，或者是最多一天发生几次而已。

所以当时这个新系统在线上的表现非常不好，明显是有经常性的卡顿的，因此针对这个系统，我们进行了一连串的排查、定位、分析和优化

下面就给大家分析一下整个优化的过程。

2、未优化前的JVM性能分析

大家还记得之前带着大家动手实操过的jstat工具吧？那个工具是非常好用，非常实用，也是非常常用的一个工具。

因为很多中小型公司都没上那种可视化的监控平台，没法直接可视化的看到JVM各个区域的内存变化，GC次数和GC耗时。

当然，如果有办法的话，我建议大家可以给自己所在公司推荐一下类似Zabbix、Ganglia、Open-Falcon、Prometheus之类的可视化监控平台，其实接入都非常简单，如果把线上系统接入了这些平台，可以直接图形化看到JVM的表现。

但是哪怕你有了可视化监控平台，有时候直接对线上系统进行分析的时候，还是jstat更加好用和直接。

所以当时我们通过监控平台+jstat工具分析，直接得出当时没优化过的系统的JVM性能表现大致如下：

机器配置：2核4G
JVM堆内存大小：2G
系统运行时间：6天
系统运行6天内发生的Full GC次数和耗时：250次，70多秒
系统运行6天内发生的Young GC次数和耗时：2.6万次，1400秒

综合分析一下，就可以知道，大致来说每天会发生40多次Full GC，平均每小时2次，每次Full GC在300毫秒左右；

每天会发生4000多次Young GC，每分钟会发生3次，每次Young GC在50毫秒左右。

上述数据对任何一个线上系统，用jstat可以轻松看出来，因为jstat显示出来的Full GC和Young GC的次数都是系统启动以来的总次数，耗时都是所有GC加起来的总耗时，所以直接可以拿到上述数据，略微分析一下就知道具体情况了。

基本看起来，这个系统的性能是相当差了，每分钟3次Young GC，每小时2次Full GC，这种系统是必须得进行优化的。

3、未优化前的线上JVM参数

下面是未优化前的线上JVM参数，大致如下：

```
-Xms1536M -Xmx1536M -Xmn512M -Xss256K -XX:SurvivorRatio=5 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=68 -XX:+CMSParallelRemarkEnabled -XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC
```

其实基本上跟我们之前看到的参数没多大的不同，一个4G的机器上，给JVM的堆内存是设置了1.5G的大小，其中新生代是给了512M，老年代是1G。

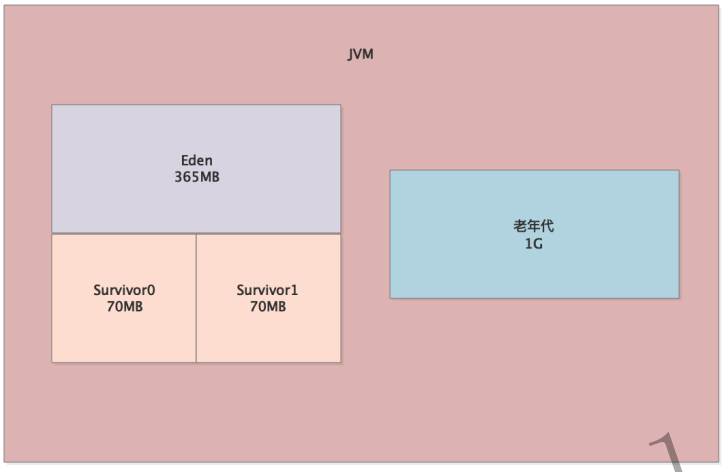
比较关键的是“-XX:SurvivorRatio”设置为了5，也就是说，Eden:Survivor1:Survivor2的比例是5:1:1

所以此时Eden区域大致为365M，每个Survivor区域大致为70MB。

而且这里有一个非常关键的参数，那就是“-XX:CMSInitiatingOccupancyFraction”参数设置为了68

所以一旦老年代内存占用达到68%，也就是大概有680MB左右的对象时，就会触发一次Full GC。

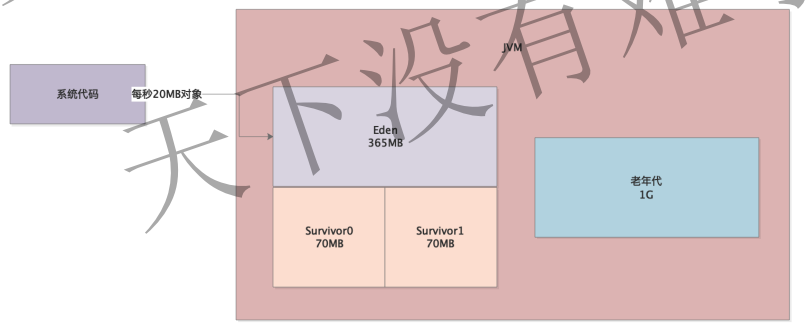
此时整个系统的内存模型图如下所示：



4、根据线上系统的GC情况倒推运行内存模型

接着我们可以根据系统的内存模型以及GC情况，直接根据学习过的知识推导出系统运行时的内存模型了。

首先我们知道每分钟会发生3次Young GC，说明系统运行20秒就会让Eden区满，也就是产生300多MB的对象，平均下来系统每秒钟会产生15~20MB的对象，如下图所示。

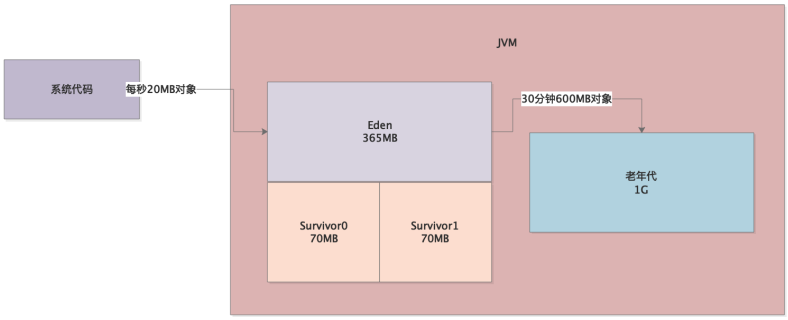


所以20秒左右就会导致Eden区满，然后触发一次Young GC。

接着我们根据每小时2次Full GC推断出，30分钟会触发一次Full GC

根据“-XX:CMSInitiatingOccupancyFraction=68”参数的设置，应该是在老年代有600多MB左右的对象时大概就会触发一次Full GC，因为1GB的老年代有68%空间占满了就会触发CMS的GC了。

所以系统运行30分钟就会导致老年代里有600多MB的对象，进而触发CMS垃圾回收器对老年代进行GC，如下图。



所以基本上我们就能根据推导出的运行内存模型得出一个结论：

每隔20秒会让300多MB的Eden区满触发一次Young GC，一次Young GC耗时50毫秒左右。

每隔30分钟会让老年代里600多MB空间占满，进而触发一次CMS的GC，一次Full GC耗时300毫秒左右。

但是到这里大家先暂停一下，有的朋友可能立马会推断了，他会说，是不是因为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30分钟后触发Full GC?

实际上仅仅只是分析到这里，绝对不能草率下这个判断的。

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

也有可能是有很多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68%的占比触发GC。

所以仅仅分析到这里，绝对不能轻易下结论。

5、老年代里到底为什么会有那么多的对象?

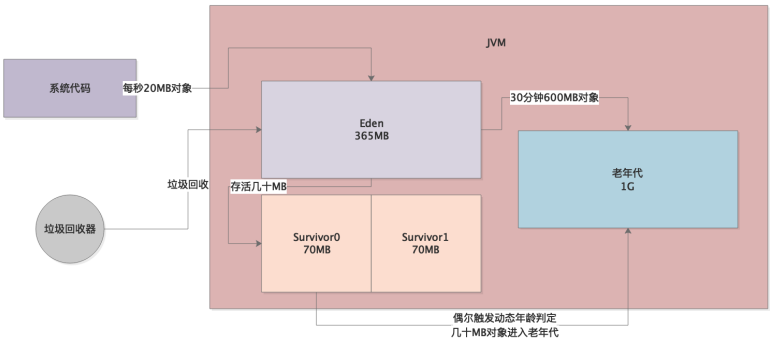
分析到这里，说句实话，仅仅根据可视化监控和推论是绝对没法往下分析了，因为我们并不知道老年代里到底为什么会有那么多的对象

此时就完全可以用jstat在高峰期观察一下JVM实际运行的情况。

通过jstat的观察，我们当时可以明确看到，每次Young GC过后升入老年代里的对象很少

一般来说，每次Young GC过后大概就存活几十MB而已，那么Survivor区域因为就70MB，所以经常会触发动态年龄判断规则，导致偶尔一次Young GC过后有几十MB对象进入老年代。

我们看下图的图示。



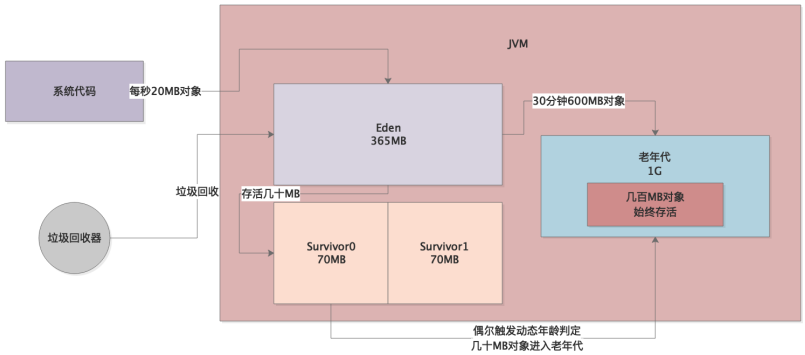
因此分析到这里很奇怪，因为通过jstat追踪观察，并不是每次Young GC后都有几十MB对象进入老年代的，而是偶尔一次Young GC才会有几十MB对象进入老年代，记住，是偶尔一次!

所以正常来说，应该不至于30分钟就导致老年代占用空间达到68%。

那么老年代里到底为什么有那么多对象呢?

这个时候我们通过jstat运行的时候就观察到一个现象，就是老年代里的内存占用在系统运行的时候，不知道为什么系统运行着运行着，就会突然有几百MB的对象占据在里面，大概有五六百MB的对象，一直占据在老年代中

大家看下图。



正是因为系统运行的时候，不知道为什么突然有有几百MB对象进入老年代中，所以才导致Young GC偶尔一次让几十MB对象升入老年代，平均30分钟左右就会触发一次Full GC！！！

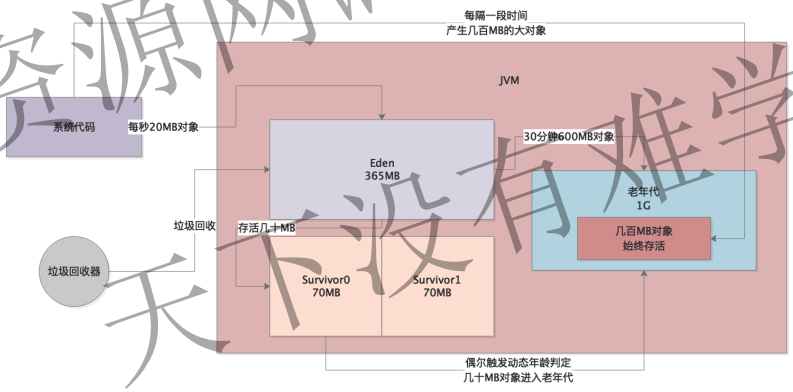
那么我们就很奇怪了，为什么系统运行着会突然有几百MB的对象进入老年代?

答案已经呼之欲出了，**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，直接进入老年代，不会走年轻代的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，所以才会30分钟触发一次Full GC!

大家看如下图所示。



6、定位系统的大对象

分析到这里，就很简单了，我们只需要采用之前给大家介绍的jmap工具，通过后台jstat工具观察系统，什么时候发现老年代里突然进入了几百MB的大对象，就立马用jmap工具导出一份dump内存快照。

接着可以采用之前说过的jhat，或者是Visual VM之类的可视化工具来分析dump内存快照

关于Visual VM之类的工具，大家自行百度即可，非常简单易用，其实本质就是让你分析导出的内存快照。

通过内存快照的分析，直接定位出来那个几百MB的大对象，就是几个Map之类的数据结构，这是什么东西？直接让负责写那个系统代码的几个同学分析了一下，明显是从数据库里查出来的!

因为那个系统仅仅就是操作数据库而已，不存在别的什么特殊操作。

然后这个时候也没太好的办法了，直接笨办法，几个人地毯式排查这个系统的所有SQL语句，结果还真的有一个人发现，自己的一个SQL居然在某种特殊的场景下，会类似如下所示：

select * from tbl。

这是啥意思？就是没有where条件！

没有where条件，就代表这个SQL可能会把表中几十万条数据直接全部查出来！

正是因为这个代码层面的bug，导致了每隔一段时间系统会搞出几个上百MB的大对象，这些对象是会全部直接进入老年代的！

然后过一会儿随着偶尔几次Young GC有几十MB对象进入老年代，所以平均几十分钟就会触发一次Full GC！！！

7、针对本案例的JVM和代码优化

其实分析到这里，这个案例如何优化已经呼之欲出了！

非常简单，分成两步走

第一步，让开发同学解决代码中的bug，避免一些极端情况下SQL语句里不拼接where条件，务必要拼接上where条件，不允许查询表中全部数据。彻底解决那个时不时有几百MB对象进入老年代的问题。

第二步，年轻代明显过小，Survivor区域空间不够，因为每次Young GC后存活对象在几十MB左右，如果Survivor就70MB很容易触发动态年龄判定，让对象进入老年代中。所以直接调整JVM参数如下：

-Xms1536M -Xmx1536M -Xmn1024M -Xss256K -XX:SurvivorRatio=5 -XX:PermSize=256M -XX:MaxPermSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=92 -XX:+CMSParallelRemarkEnabled -XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC

直接把年轻代空间调整为700MB左右，每个Surivor是150MB左右，此时Young GC过后就几十MG存活对象，一般不会进入老年代。

反之老年代就留500MB左右就足够了，因为一般不会有对象进入老年代。

而且调整了参数“-XX:CMSInitiatingOccupancyFraction=92”，避免老年代仅仅占用68%就触发GC，现在必须要占用到92%才会触发GC。

最后，就是主动设置了永久代大小为256MB，因为如果不主动设置会导致默认永久代就在几十MB的样子，很容易导致万一系统运行时候采用了反射之类的机制，可能一旦动态加载的类过多，就会频繁触发Full GC。

这几个步骤优化完毕之后，线上系统基本上表现就非常好了，基本上每分钟大概发生一次Young GC，一次在几十毫秒；

Full GC几乎就很少，大概可能要运行至少10天才会发生一次，一次就耗时几百毫秒而已，频率很低。

8、今日小思考题

今天这个案例的特点，是搭配上了大对象的问题排查，当你发现Young GC过后并不是每次都有很多存活对象进入老年代的时候，就得从别的角度考虑一下到底为什么会有那么多的对象进入老年代了。

希望大家可以自己结合这个案例的思路，动手画画图，对这个案例进行一定的分析和推导，自己动手做一下分析，一定可以让你i积累出来JVM优化的思路和经验的！

同样老规矩，对文章有什么疑问，或者自己对思考题的答案，都可以在评论区留言，我会逐一回复大家。

End

如何加群?