图文 064、案例实战：一次线上大促营销活动导致的内存泄漏和Full GC优化

详情 评论

案例实战：
一次线上大促营销活动导致的内存泄漏和Full GC优化！

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、线上故障场景

先简单说一下业务背景：一次我们线上推了一个大促销活动，大致就是类似于在某个特定节日里，突然给所有用户发短信、邮件、APP Push消息，说现在有个特别优惠的活动，如果参与的话肯定可以得到很大的实惠！

这类大促活动一般都会吸引比平时多几倍的用户短时间内突然登录APP来参与，所以系统一般在这个时候压力会比平时大好几倍。

但是因为从系统的整体设计角度而言，其实给的一些数据库、缓存和机器的资源都是足够的，所以通常而言不该有什么问题。

但是那次大促活动开始之后，直接导致线上一个系统的CPU使用率飙升，而且因为CPU使用率太高，导致系统几乎陷入卡死的状态，无法处理任何请求！

在重启系统之后，会好一段时间，但是很快又立马发现机器的CPU使用率飙升，继续导致系统卡死！

这就是那次大促活动开始之后，那个系统在线上的一个真实的情况。

有人可能会问，那么CPU使用率是怎么观察到飙升的？怎么收到报警的？

其实这个之前已经说过很多次了，中大型公司都会有Zabbix、Open-Falcon、Prometheus之类的监控和告警系统，一旦机器的CPU使用率过高，会直接发送报警给你的短信、邮箱和IM工具（比如钉钉）

所以上面说的大促活动开始之后，某个线上系统的CPU使用率飙升，其实就是得到了报警才知道的，然后在监控系统上还可以去观察CPU的负载曲线，是一个折线图，可以看到CPU负载很高。

2、初步排查CPU负载过高的原因

这里给大家说一下线上系统的机器CPU负载过高的两个常见的场景。

第一个场景，是你自己在系统里创建了大量的线程，这些线程同时并发运行，而且工作负载都很重，过多的线程同时并发运行就会导致你的机器CPU负载过高。

第二个场景，就是你的机器上运行的JVM在执行频繁的Full GC，Full GC是非常耗费CPU资源的，他是一个非常重负载的过程

所以一旦你的JVM有频繁的Full GC，带来的一个明显的感觉，一个是系统可能时不时会卡死，因此Full GC会带来一定的“Stop the World”问题，一个是机器的CPU负载很高。

所以一旦知道CPU负载过高的两个原因，就很容易进行排查了。

大家完全可以使用排除法来做，首先看一下JVM Full GC的频率，通过jstat也好，或者是监控平台也好，很容易看到现在Full GC的频率。如果Full GC频率过高，那么就是Full GC引起的CPU负载过高。

那么如果JVM GC频率很正常呢？那就肯定是你的系统创建了过多线程在并发执行负载很重的任务了！

所以当时我们直接通过监控平台就可以看到，JVM的Full GC频率突然变得极为频繁，几乎是每分钟都有一次Full GC。

大家都知道，每分钟一次Full GC，一次至少耗时几百毫秒，这个系统性能绝对很糟糕，而且对机器的CPU负载也是很高的！

既然发现了频繁Full GC了，那肯定就不用去怀疑是系统自己创建过多线程了！

3、初步排查频繁Full GC的问题

大家通过之前大量的案例和文章已经初步可以得到结论，如果有频繁Full GC的问题，一般可能性有三个：

内存分配不合理，导致对象频繁进入老年代，进而引发频繁的Full GC；

存在内存泄漏等问题，就是内存里驻留了大量的对象塞满了老年代，导致稍微有一些对象进入老年代就会引发Full GC；

永久代里的类太多，触发了Full GC

当然还有之前案例说过，如果上述三个原因都不存在，但是还是有频繁Full GC，也许就是工程师错误的执行“System.gc()”导致的

但是这个一般很少见，而且之前讲过，JVM参数中可以禁止这种显式触发的GC。

所以一般排查频繁Full GC，核心的利器当然是jstat了，之前我们有大量文章带大家做过jstat分析JVM的实战，这里就不赘述了。

当时我们用jstat分析了一下线上系统的情况，发现并不存在内存分配不合理，对象频繁进入老年代的问题，而且永久代的内存使用也很正常，所以上述三个原因中的两个就被排除掉了。

那么我们来考虑最后一个原因：老年代里是不是驻留了大量的对象给塞满了？

对，当时系统就是这个问题！

我们明显发现老年代驻留了大量的对象，几乎快塞满了，所以年轻代稍微有一些对象进入老年代，很容易就会触发Full GC！而且Full GC之后还不会回收掉老年代里大量的对象，只是回收一小部分而已！

所以很明显老年代里驻留了大量的本不应该存在的对象，才导致频繁触发Full GC的。接下来就是要想办法找到这些对象了

之前我们介绍过jmap+jhat的组合来分析内存里的大对象，今天我们介绍另外一个常用的强有力的工具，**MAT**。

因为jhat适合快速的去分析一下内存快照，但是功能上不是太强大，所以一般其实常用的比较强大的**内存分析工具**，就是MAT。

4、对线上系统导出一份内存快照

既然都发现线上系统的老年代中驻留了过多的对象的问题，那么肯定要知道这些对象是谁！

所以先用jmap命令导出一份线上系统的内存快照即可：

jmap -dump:format=b,file=文件名 [服务进程ID]

拿到了内存快照之后，其实就是一份文件，接着就可以用jhat、MAT之类的工具来分析内存了。

5、MAT是如何使用的？

不少人是通过Eclipse集成的MAT插件来使用的，但是很多人其实开发是用IntelliJ IDEA的，所以这个时候可以直接下载一个MAT来使用即可

给大家官网的下载地址：

https://www.eclipse.org/mat/downloads.php

在这个地址中，就可以下载MAT的最新版本了。

大家选择自己的笔记本电脑的操作系统对应的版本就可以了，他是支持Windows、Mac、Linux三种操作系统的。

下载好MAT后，在他的安装目录里，可以看到一个文件名字叫做：MemoryAnalyzer.ini

这个文件里的内容类似如下所示：

```
-startup
../Eclipse/plugins/org.eclipse.equinox.launcher_1.5.0.v20180512-1130.jar
--launcher.library
../Eclipse/plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.700.v20180518-1200
-vmargs
-Xmx1024m
-Dorg.eclipse.swt.internal.carbon.smallFonts
-XstartOnFirstThread
```

大家务必要记得，如果dump出来的内存快照很大，比如有几个G，你务必在启动MAT之前，先在这个配置文件里给MAT本身设置一下堆内存大小，比如设置为4个G，或者8个G，他这里默认-Xmx1024m是1G。

接着大家直接启动MAT即可，启动之后看到的界面中有一个选型是：Open a Heap Dump，就是打开一个内存快照的意思，选择他，然后选择本地的一个内存快照文件即可。

6、基于MAT来进行内存泄漏分析

使用MAT打开一个内存快照之后，在MAT上有一个工具栏，里面有一个按钮，他的英文是：Leak Suspects，就是内存泄漏的分析。

接着MAT会分析你的内存快照，尝试找出来导致内存泄漏的一批对象。

这时明显可以看到他会显示给你一个大的饼图，这里就会提示你说，哪些对象占用内存过大。

这个时候直接会看到某种自己系统创建的对象占用量过大，这种对象的实例多达数十万个，占用了老年代一大半的内存空间。

接着当然是找开发工程师去排查这个系统的代码问题了，为什么会创建那么多的对象，而且始终回收不掉?

这就是典型的内存泄漏！即系统创建了大量的对象占用了内存，其实很多对象是不需要使用的，而且还无法回收掉。

后来找到了一个原因，是在系统里做了一个JVM本地的缓存，把很多数据都加载到内存里去缓存起来，然后提供查询服务的时候直接从本地内存走。

但是因为没有限制本地缓存的大小，并且没有使用LRU之类的算法定期淘汰一些缓存里的数据，导致缓存在内存里的对象越来越多，进而造成了内存泄漏。

解决问题很简单，只要使用类似EHCache之类的缓存框架就可以了，他会固定最多缓存多少个对象，定期淘汰删除掉一些不怎么访问的缓存，以便于新的数据可以进入缓存中。

7、今日文章总结

之前给大家讲过，我们的文章会不停的在后面的案例中重复一些之前案例里的内容，原因是学习周期很长，通过这种方式可以定期帮助大家复习和总结。

这篇文章就给大家分析总结了一下之前学习过的几种频繁Full GC的原因，以及分析的方法和思路，这个可以看做是一个复习性的文章。

同时给大家初步介绍了一下MAT这种内存分析的工具，其使用是非常简单的，里面有很多的功能。

而今天之所以没有给大家很多截图，一步一步教大家来用这个工具，就是希望给大家课后留一个作业。

今天的作业就是，希望大家可以自己动手玩一玩MAT，你可以自己运行一段代码，模拟生成一种对象特别多的实例，然后导出一份内存快照，基于MAT来分析一下，就可以看到他是如何清晰的告诉你系统中哪种对象实例过多了!

另外，之后我们会有更多案例，我会带着大家一步一图，使用MAT、Visual VM等工具来深度分析JVM的内存快照，找到一些内存泄漏的问题，后续都会有，咱们一步步来，敬请期待。

End
