图文 059、案例实战：新手工程师不合理设置JVM参数，是如何导致频繁Full GC的?

详情 评论

案例实战：
新手工程师不合理设置JVM参数，是如何导致频繁Full GC的?

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）

1、本文背景

本文会给大家讲解一个比较特殊的JVM优化案例，这个优化案例本身是因为新手工程师对JVM优化可能了解了一个半吊子，然后不知道从哪里找来了一个非常特殊的JVM参数错误的设置了一下，就导致线上系统频繁的出现Full GC的问题。

但是我们后续大量的优化案例其实都是各种各样奇形怪状的场景，因为正是各种奇怪场景才能让大家逐步积累出来较为丰富的JVM优化实战经验

了解的场景越多，自己未来在处理JVM性能问题的时候才能更是得心应手。

2、问题的产生

这个场景的发生大致如下过程：某天团队里一个新手工程师大概是心血来潮，觉得自己网上看到了某个JVM参数，以为学会了绝世武功秘籍，于是就在当天上线一个系统的时候，自作主张设置了一个JVM参数

这个参数是什么呢?

不用急，跟着看下面的案例分析即可，现在只要知道他设置了一个奇怪的参数，接着事故就发生了。

因为一般中大型公司都是接入类似Zabbix、OpenFalcon或者公司自研的一些监控系统的，监控系统一般都做的很好，可以让你的系统直接接入进去，然后在上面可以看到每台机器的CPU、磁盘、内存、网络的一些负载。

而且可以看到你的JVM的内存使用波动折线图，还有你的JVM GC发生的频率折线图。包括如果你自己上报某个业务指标，也可以在监控系统里看到。

而且一般都会针对线上运行的机器和系统设置一些报警，比如说，你可以设置如果10分钟内发现一个系统的JVM发生了超过3次Full GC，就必须发送报警给你，可以发送给你的短信、邮箱或者是钉钉之类的IM工具。

类似这样的监控系统不在我们的专栏范畴内，建议大家自己可以去查阅资料，其实基于我们讲解的命令行工具，比如jstat，你可以通过linux上的一些命令，让jstat自动对jvm进行监控，把监控结果可以输出到机器的某个文件里去。

然后第二天你就可以去查阅那个文件，也可以看到那台机器的jvm的一些gc统计。

所以说，没有可视化工具，用最简单的命令行工具，其实同样可以起到类似的效果。

所以那天那个工程师设置了一个JVM参数之后，直接导致线上频繁接到JVM的Full GC的报警，大家就很奇怪了，于是就开始排查那个系统了。

3、查看GC日志

之前已经给大家讲解过如何在启动系统的时候让他输出GC日志，所以一旦发现报警，直接登录到线上机器，然后就看到对应的GC日志了。

此时我们看到在GC日志中有大量的Full GC的记录。

那么是为什么导致的Full GC呢?

在日志里，看到了一个“Metadata GC Threshold”的字样，类似于如下日志：

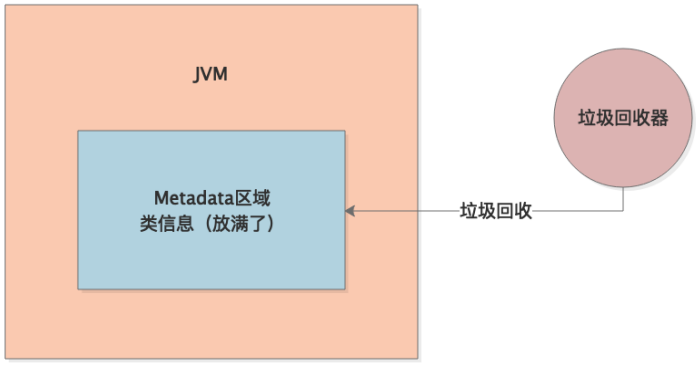
【Full GC（Metadata GC Threshold）xxxxx, xxxxx】

从这里就知道，这频繁的Full GC，实际上是JDK 1.8以后的Metadata元数据区导致的，也就是类似我们之前说的永久代。

这个Metadata区域一般是放一些加载到JVM里去的类的。

所以此时就很奇怪了，为什么会因为Metadata区域频繁的被塞满，进而触发Full GC？而且Full GC大家都知道，会带动CMS回收老年代，还会回收Metadata区域本身。

我们先看看下图：



4、查看Metaspace内存占用情况

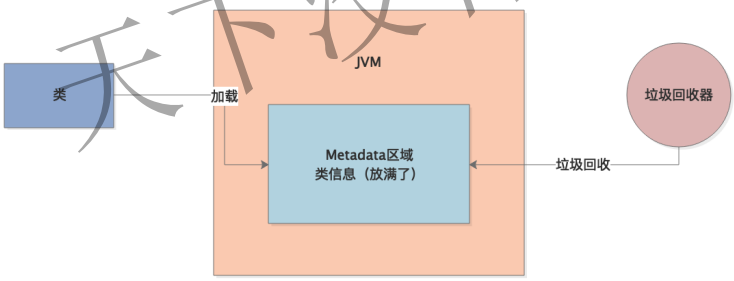
接着我们当然是想看一看Metaspace区域的内存占用情况了，简单点你可以通过jstat来观察，如果有监控系统，他会给你展示出来一个Metaspace内存区域占用的波动曲线图，类似下面这种。

看起来Metaspace区域的内存呈现一个波动的状态，他总是会先不断增加，达到一个顶点之后，就会把Metaspace区域给占满，然后自然就会触发一次Full GC，Full GC会带着Metaspace区域的垃圾回收，所以接下来Metaspace区域的内存占用又变得很小了。

5、一个综合性的分析思路

看到这里，相信大家肯定有一点感觉了，这个很明显是系统在运行过程中，不停的有新的类产生被加载到Metaspace区域里去，然后不停的把Metaspace区域占满，接着触发一次Full GC回收掉Metaspace区域中的部分类。

然后这个过程反复的不断的循环，进而造成Metaspace区域反复被占满，然后反复导致Full GC的发生，如下图所示。



6、到底是什么类不停的被加载?

接着我们就有点奇怪了，到底是什么类不停的被加载到JVM的Metaspace区域里去?

这个时候就需要在JVM启动参数中加入如下两个参数了：

“-XX:TraceClassLoading -XX:TraceClassUnloading”

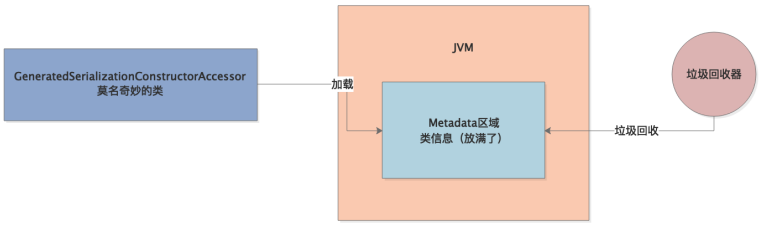
这两个参数，顾名思义，就是追踪类加载和类卸载的情况，他会通过日志打印出来JVM中加载了哪些类，卸载了哪些类。

加入这两个参数之后，我们就可以看到在Tomcat的catalina.out日志文件中，输出了一堆日志，里面显示类似如下的内容：

【Loaded sun.reflect.GeneratedSerializationConstructorAccessor from __JVM_Defined_Class】

明显可以看到，JVM在运行期间不停的加载了大量的所谓“GeneratedSerializationConstructorAccessor”类到了Metaspace区域里去

如下图所示



相信就是因为JVM运行期间不停的加载这种奇怪的类，然后不停的把Metaspace区域占满，才会引发不停的执行Full GC的。

这是一个非常实用的技巧，各位同学一定要掌握，频繁Full GC不光是老年代触发的，有时候也会因为Metaspace区域的类太多而触发。

到此为止，已经慢慢接近真相了。

7、为什么会频繁加载奇怪的类?

接着遇到类似这种问题，我们就应该找一下Google或者是百度了，当然推荐是用Google。你完全可以看看那种不停加载的类，到底是什么类，是你自己写的类？还是说JDK内置的类?

比如上面的那个类，如果你查阅一些资料，很容易就会搞明白，那个类大概是在你使用Java中的反射时加载的，所谓反射代码类似如下所示。

```
Method method = XXX.class.getDeclaredMethod(xx,xx);
method.invoke(target,params);
```

友情提示一下，反射是Java中最最基础的一个概念，不懂的朋友自己查一下资料。

简单来说，就是通过XXX.class获取到某个类，然后通过geteDeclaredMethod获取到那个类的方法。

这个方法就是一个Method对象，接着通过Method.invoke可以去调用那个类的某个对象的方法，大概就这个意思。

在执行这种反射代码时，JVM会在你反射调用一定次数之后就动态生成一些类，就是我们之前看到的那种莫名其妙的类

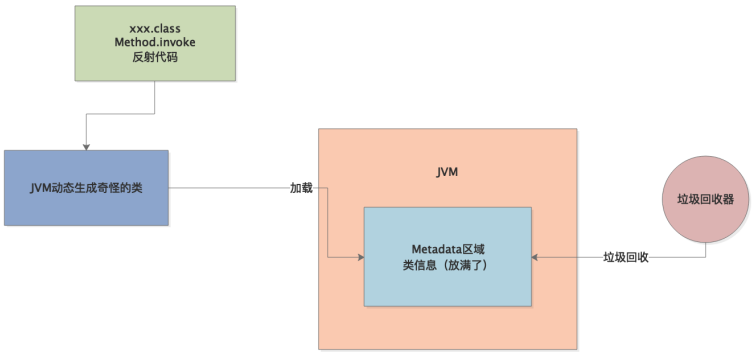
下次你再执行反射的时候，就是直接调用这些类的方法，这是JVM的一个底层优化的机制。

看到这里，有的小伙伴是不是有点蒙?

其实这倒无所谓，这段话看的蒙丝毫不影响你进行JVM优化的

你只要记住一个结论：如果你在代码里大量用了类似上面的反射的东西，那么JVM就是会动态的去生成一些类放入Metaspace区域里的。

所以上面看到的那些奇怪的类，就是由于不停的执行反射的代码才生成的，如下图所示。



8、JVM创建的奇怪类有什么玄机?

那么接下来我们就很奇怪一件事情，就是JVM为什么要不停的创建那些奇怪的类然后放入Metaspace中去?

其实这就要从一个点入手来分析一下了，因为上面说的那种JVM自己创建的奇怪的类，他们的Class对象都是SoftReference，也就是软引用的。

大家可千万别说连类的Class是什么都没听说过？简单来说，每个类其实本身自己也是一个对象，就是一个Class对象，一个Class对象就代表了一个类。同时这个Class对象代表的类，可以派生出来很多实例对象。

举例来说，Class Student，这就是一个类，他本身是由一个Class类型的对象表示的。

但是如果你走一个Student student = new Student()，这就是实例化了这个Student类的一个对象，这是一个Student类型的实例对象。

所以我们这里所说的Class对象，就是JVM在发射过程中动态生成的类的Class对象，他们都是SoftReference软引用的。

所谓的软引用，最早我们再一篇文章里说过，正常情况下不会回收，但是如果内存比较紧张的时候就会回收这些对象。

那么SoftReference对象到底在GC的时候要不要回收是通过什么公式来判断的呢?

是如下的一个公式：**clock - timestamp <= freespace * SoftRefLRUPolicyMSPerMB**。

这个公式的意思就是说，“clock - timestamp”代表了一个软引用对象他有多久没被访问过了，freespace代表JVM中的空闲内存空间，SoftRefLRUPolicyMSPerMB代表每一MB空闲内存空间可以允许SoftReference对象存活多久。

举个例子，假如说现在JVM创建了一大堆的奇怪的类出来，这些类本身的Class对象都是被SoftReference软引用的。

然后现在JVM里的空间内存空间有3000MB，SoftRefLRUPolicyMSPerMB的默认值是1000毫秒，那么就意味着，此时那些奇怪的SoftReference软引用的Class对象，可以存活3000 * 1000 = 3000秒，就是50分钟左右。

当然上面都是举例而已，大家都知道，一般来说发生GC时，其实JVM内部或多或少总有一些空间内存的，所以基本上如果不是快要发生OOM内存溢出了，一般软引用也不会被回收。

所以大家就知道了，按理说JVM应该会随着反射代码的执行，动态的创建一些奇怪的类，他们的Class对象都是软引用的，正常情况下不会被回收，但是也不应该快速增长才对。

9、为什么JVM创建的奇怪的类会不停的变多?

那么究竟为什么JVM创建的那些奇怪的类会不停的变多呢?

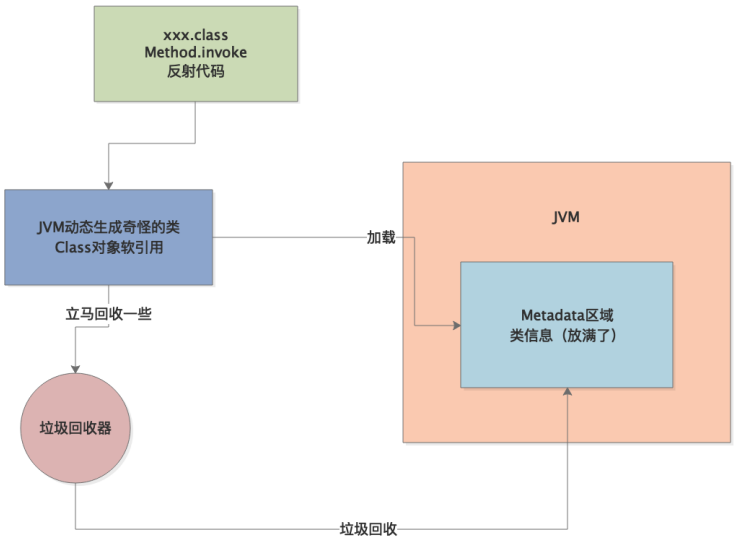
原因很简单，因为文章开头那个新手工程师不知道从哪里扒出来了SoftRefLRUPolicyMSPerMB这个JVM启动参数，他直接把这个参数设置为0了。

他想的是，一旦这个参数设置为0，任何软引用对象就可以尽快释放掉，不用留存，尽量给内存释放空间出来，这样不就可以提高内存利用效率了么?

真是想的很傻很天真。

实际上一旦这个参数设置为0之后，直接导致clock - timestamp <= freespace * SoftRefLRUPolicyMSPerMB这个公式的右半边是0，就导致所有的软引用对象，比如JVM生成的那些奇怪的Class对象，刚创建出来就可能被一次Young GC给带着立马回收掉一些。

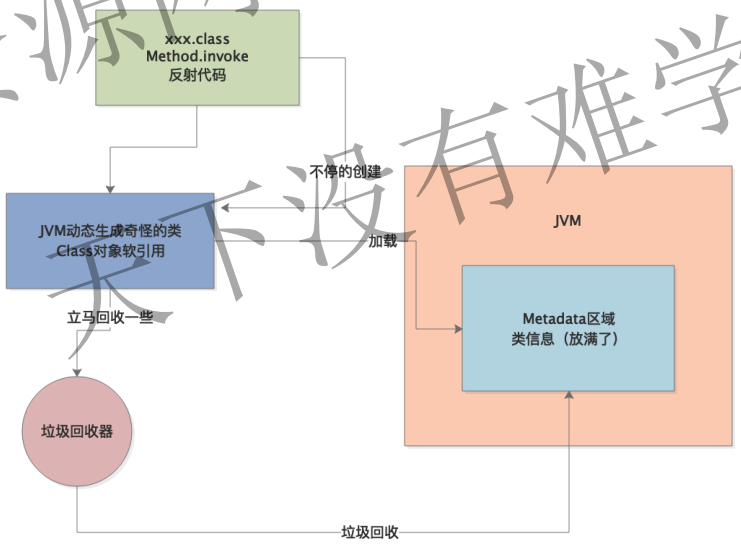
如下图所示。



比如JVM好不容易给你弄出来100个奇怪的类，结果因为你瞎设置软引用的参数，导致突然一次GC就给你回收掉几十个类

接着JVM在反射代码执行的过程中，就会继续创建这种奇怪的类，在JVM的机制之下，会导致这种奇怪类越来越多。

也许下一次gc又会回收掉一些奇怪的类，但是马上JVM还会继续生成这种类，最终就会导致Metaspace区域被放满了，一旦Metaspace区域被占满了，就会触发Full GC，然后回收掉很多类，接着再次重复上述循环，如下图所示。



其实很多人会有一个疑问，到底为什么软引用的类因为错误的参数设置被快速回收之后，就会导致JVM不停创建更多的新的类呢?

其实大家不用去扣这里的细节，这里有大量的底层JDK源码的实现，异常复杂，要真的说清楚，得好几篇文章才能讲清楚JDK底层源码的这些细节。

大家只要记住这个结论，明白这个道理就好。

10、如何解决这个问题?

虽然底层JDK的一些实现细节我们没分析，但是大致梳理出来了一个思路，大家也很清楚问题所在和原因了

解决方案很简单。在有大量反射代码的场景下，大家只要把

-XX:SoftRefLRUPolicyMSPerMB=0

这个参数设置大一些即可，千万别让一些新手同学设置为0，可以设置个1000，2000，3000，或者5000毫秒，都可以。

提高这个数值，就是让反射过程中JVM自动创建的软引用的一些类的Class对象不要被随便回收，当时我们优化这个参数之后，就可以看到系统稳定运行了。

基本上Metaspace区域的内存占用是稳定的，不会来回大幅度波动了。

11、今日思考题

结合昨天的内容，**大家思考一下这个线上事故的本质是什么?**

其实说白了不是JVM的问题，是人的问题。

大家可以考虑一下，如果你是公司的架构师，是否应该严格审核各个系统的生产环境JVM参数?

比如完全可以推行一套JVM参数模板，如果有人要做定制的JVM优化，是不是应该先在测试环境尝试一下，然后还得交给你们高级别的架构师来审核?

如果有人审核，那么就不会发生类似之类的血案了。

End

狸猫技术窝其他精品专栏推荐:

21天互联网java进阶面试训练营（分布式篇）