图文 054、案例实战：每日百亿数据量的实时分析引擎，如何定位和解决频繁Full GC问题?

详情 评论

案例实战：

每日百亿数据量的实时分析引擎，如何定位和解决频繁Full GC问题?

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、前文回顾

大家应该还记得之前我们有一篇文章，分析了一个实时计算系统因为负载过高导致了非常频繁的Full GC

这里先在下面贴出来之前的内容，大家先看一下回顾回顾，接着我们会给出示例代码，运行起来之后通过jstat来观察其运行中的问题。

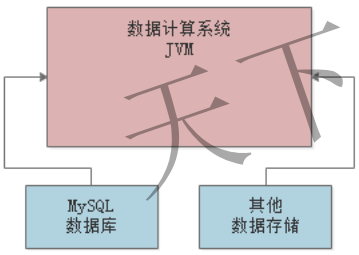
然后我还会优化一下JVM参数配置，再次运行系统，通过jstat来观察JVM优化以后的效果。

2、一个日处理上亿数据的计算系统

先给大家说一下这个系统的案例背景，当时我们团队里自己研发的一个数据计算系统，日处理数据量在上亿的规模。

为了方便大家集中注意力理解这个系统的生产环境的JVM相关的东西，所以对系统本身就简化说明了。

简单来说，这个系统就是会不停的从MySQL数据库以及其他数据源里提取大量的数据加载到自己的JVM内存里来进行计算处理，如下图所示。

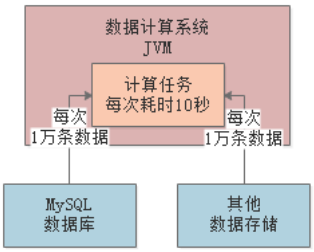


这个数据计算系统会不停的通过SQL语句和其他方式从各种数据存储中提取数据到内存中来进行计算，大致当时的生产负载是每分钟大概需要执行500次数据提取和计算的任务。

但这是一套分布式运行的系统，所以生产环境部署了多台机器，每台机器大概每分钟负责执行100次数据提取和计算的任务。

每次会提取大概1万条左右的数据到内存里来计算，平均每次计算大概需要耗费10秒左右的时间

然后每台机器是4核8G的配置，JVM内存给了4G，其中新生代和老年代分别是1.5G的内存空间，大家看下图。



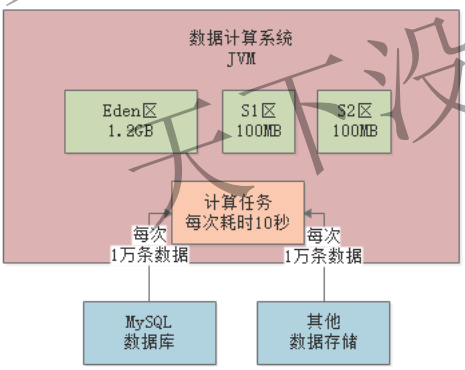
3、这个系统到底多快会塞满新生代?

现在明确了一些核心数据，接着我们来看看这个系统到底多快会塞满新生代的内存空间?

既然这个系统每台机器上部署的实例，每分钟会执行100次数据计算任务，每次是1万条数据需要计算10秒的时间，那么我们来看看每次1万条数据大概会占用多大的内存空间?

这里每条数据都是比较大的，大概每条数据包含了平均20个字段，可以认为平均每条数据在1KB左右的大小。

那么每次计算任务的1万条数据就对应了10MB的大小。所以大家此时可以思考一下，如果新生代是按照8:1:1的比例来分配Eden和两块Survivor的区域，那么大体上来说，Eden区就是1.2GB，每块Survivor区域在100MB左右，如下图。



基本上按照这个内存大小而言，大家会发现，每次执行一个计算任务，就会在Eden区里分配10MB左右的对象

一分钟大概对应100次计算任务，基本上一分钟过后，Eden区里就全是对象，基本就全满了。

所以说，回答这个小节的问题，新生代里的Eden区，基本上1分钟左右就迅速填满了。

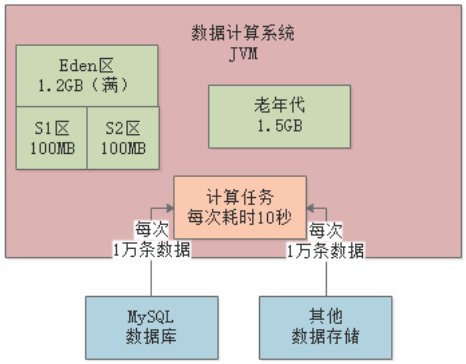
4、触发Minor GC的时候会有多少对象进入老年代?

此时假设新生代的Eden区在1分钟过后都塞满对象了，然后在接着继续执行计算任务的时候，势必会导致需要进行Minor GC回收一部分的垃圾对象。

那么上篇文章给大家讲过这里在执行Minor GC之前会先进行的检查。

首先第一步，先看看老年代的可用内存空间是否大于新生代全部对象？

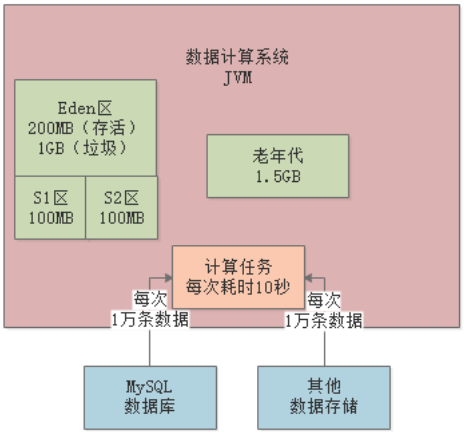
看下图，此时老年代是空的，大概有1.5G的可用内存空间，新生代的Eden区大概算他有1.2G的对象好了。



此时会发现老年代的可用内存空间有1.5GB，新生代的对象总共有1.2GB，即使一次Minor GC过后，全部对象都存活，老年代也能放的下的，那么此时就会直接执行Minor GC了。

那么此时Eden区里有多少对象还是存活的，无法被垃圾回收呢？

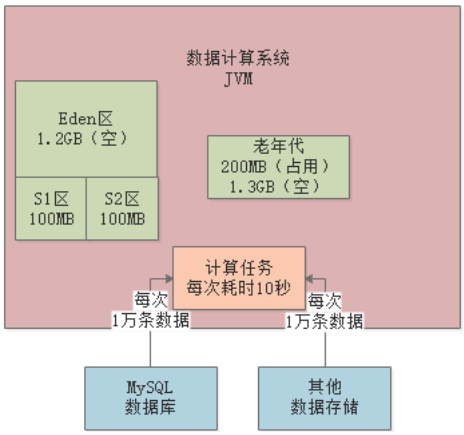
大家可以考虑一下之前说的那个点，每个计算任务1万条数据需要计算10秒钟，所以假设此时80个计算任务都执行结束了，但是还有20个计算任务共计200MB的数据，还在计算中，那么此时就是200MB的对象是存活的，不能被垃圾回收掉，然后有1GB的对象是可以垃圾回收的，大家看下图。



此时一次Minor GC就会回收掉1GB的对象，然后200MB的对象能放入Survivor区吗?

不能!

因为任何一块Survivor区实际上就100MB的空间，此时就会通过空间担保机制，让这200MB对象直接进入老年代去，占用里面200MB内存空间，然后Eden区就清空了，大家看下图。



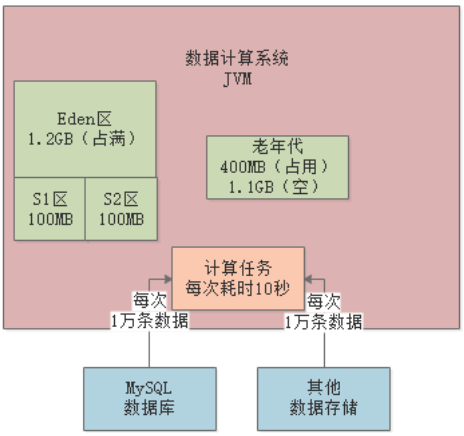
5、系统运行多久，老年代大概就会填满?

那么大家想一下，这个系统大概运行多久，老年代会填满呢?

按照上述计算，每分钟都是一个轮回，大概算下来是每分钟都会把新生代的Eden区填满，然后触发一次Minor GC，然后大概都会有200MB左右的数据进入老年代。

那么大家可以想一下，假设现在2分钟运行过去了，此时老年代已经有400MB内存被占用了，只有1.1GB的内存可用，此时如果第3分钟运行完毕，又要进行Minor GC会做什么检查呢?

如下图:



此时会先检查老年代可用空间是否大于新生代全部对象，此时老年代可用空间1.1GB，新生代对象有1.2GB，那么此时假设一次Minor GC过后新生代对象全部存活，老年代是放不下的，那么此时就得看看一个参数是否打开了 。

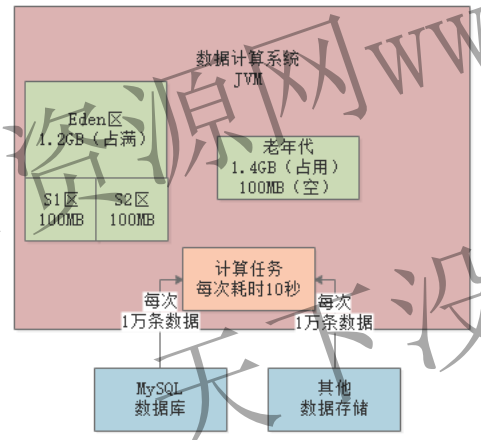
如果“-XX:-HandlePromotionFailure”参数被打开了，当然一般都会打开，此时会进入第二步检查，就是看看老年代可用空间是否大于历次Minor GC过后进入老年代的对象的平均大小。

我们已经计算过了，大概每分钟会执行一次Minor GC，每次大概200MB对象会进入老年代。

那么此时发现老年代的1.1GB空间，是大于每次Minor GC后平均200MB对象进入老年代的大小的，所以基本可以推测，本次Minor GC后大概率还是有200MB对象进入老年代，1.1G可用空间是足够的。

所以此时就会放心执行一次Minor GC，然后又是200MB对象进入老年代。

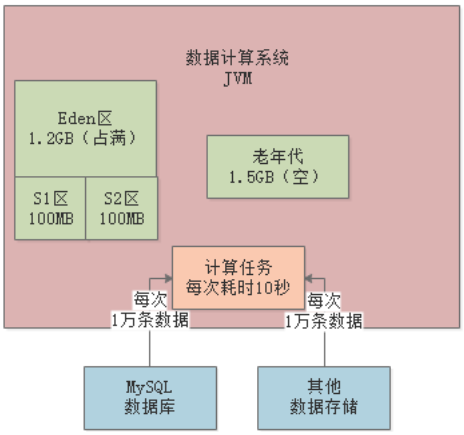
转折点大概在运行了7分钟过后，7次Minor GC执行过后，大概1.4G对象进入老年代，老年代剩余空间就不到100MB了，几乎快满了，如下图。



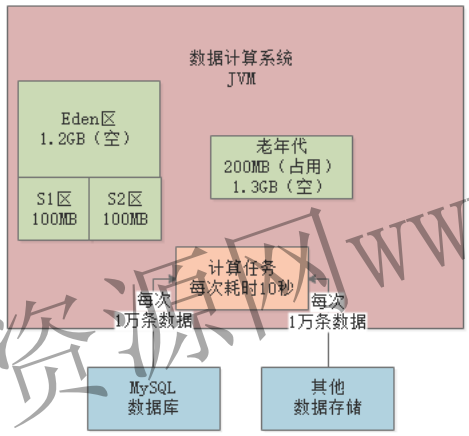
6、这个系统运行多久，老年代会触发1次Full GC?

大概在第8分钟运行结束的时候，新生代又满了，执行Minor GC之前进行检查，此时发现老年代只有100MB内存空间了，比之前每次Minor GC后进入老年代的200MB对象要小，此时就会直接触发一次Full GC。

Full GC会把老年代的垃圾对象都给回收了，假设此时老年代被占据的1.4G空间里，全部都是可以回收的对象，那么此时一次性就会把这些对象都给回收了，如下图。



然后接着就会执行Minor GC，此时Eden区情况，200MB对象再次进入老年代，之前的Full GC就是为这些新生代本次Minor GC要进入老年代的对象准备的，如下图。



按照这个运行模型，基本上平均就是七八分钟一次Full GC，这个频率就相当高了。因为每次Full GC速度都是很慢的，性能很差

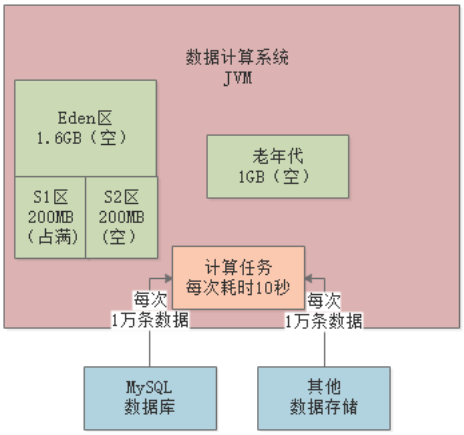
7、该案例应该如何进行JVM优化?

相信通过这个案例，大家结合图一路看下来，对新生代和老年代如何配合使用，然后什么情况下触发Minor GC和Full GC，什么情况下会导致频繁的Minor GC和Full GC，大家都有了更加深层次和透彻的理解了。

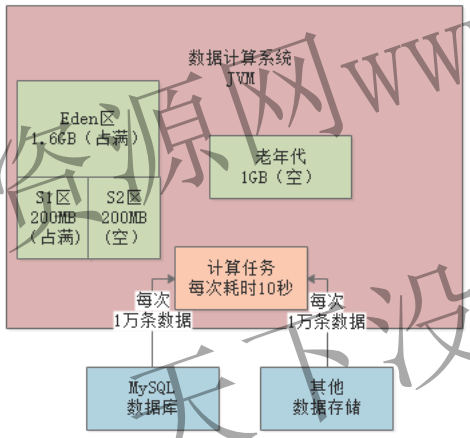
对这个系统，其实要优化也是很简单的，因为这个系统是数据计算系统，每次Minor GC的时候，必然会有一批数据没计算完毕，但是按照现有的内存模型，最大的问题，其实就是 每次Survivor区域放不下存活对象。

所以当时我们就是对生产系统进行了调整，增加了新生代的内存比例，3GB左右的堆内存，其中2GB分配给新生代，1GB留给老年代

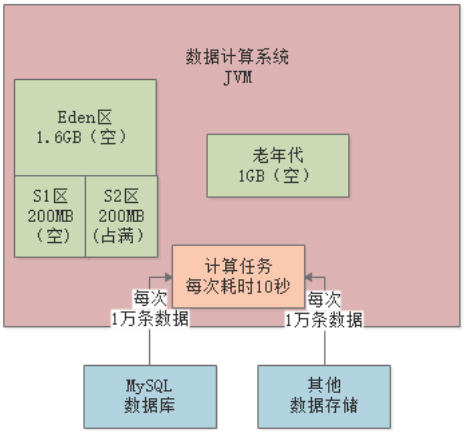
这样Survivor区大概就是200MB，每次刚好能放得下Minor GC过后存活的对象了，如下图所示。



只要每次Minor GC过后200MB存活对象可以放Survivor区域，那么等下一次Minor GC的时候，这个Survivor区的对象对应的计算任务早就结束了，都是可以回收的了，此时比如Eden区里1.6GB空间被占满了，然后Survivor1区里有200MB上一轮 Minor GC后存活的对象，如下图。



然后此时执行Minor GC，就会把Eden区里1.4GB对象回收掉，Survivor1区里的200MB对象也会回收掉，然后Eden区里剩余的200MB存活对象会放入Survivor2区里，如下图。



以此类推，基本上就很少对象会进入老年代中，老年代里的对象也不会太多的。

通过这个分析和优化，定时我们成功的把生产系统的老年代Full GC的频率从几分钟一次降低到了几个小时一次，大幅度提升了系统的性能，避免了频繁Full GC对系统运行的影响。

8、运行程序用的示例JVM参数

下面的参数唯一修改的就是“**-XX:PretenureSizeThreshold**”，把大对象阈值修改为了20MB，避免我们程序里分配的大对象直接进入老年代。

-XX:NewSize=104857600 -XX:MaxNewSize=104857600 -XX:InitialHeapSize=209715200 -XX:MaxHeapSize=209715200 -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15 -XX:PretenureSizeThreshold=20971520 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:gc.log

9、示例程序

```java
public class Demo1 {

    public static void main(String[] args) throws Exception {
        Thread.sleep(30000);
        while(true) {
            loadData();
        }
    }

    private static void loadData() throws Exception {
        byte[] data = null;
        for(int i = 0; i < 4; i++) {
            data = new byte[10 * 1024 * 1024];
        }
        data = null;

        byte[] data1 = new byte[10 * 1024 * 1024];
        byte[] data2 = new byte[10 * 1024 * 1024];

        byte[] data3 = new byte[10 * 1024 * 1024];
        data3 = new byte[10 * 1024 * 1024];

        Thread.sleep(1000);
    }

}
```

简单给大家解释一下上面的程序

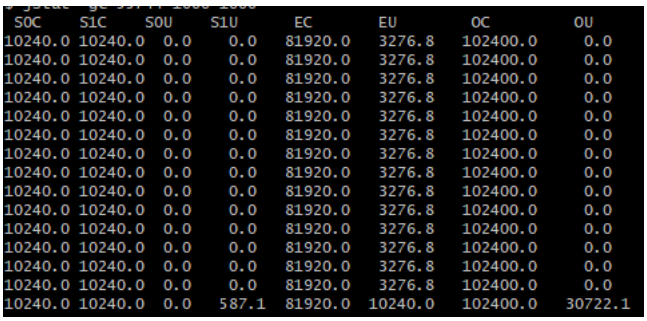
大概意思其实就是，每秒钟都会执行一次loadData()方法，他会分配4个10MB的数组，但是都立马成为垃圾，但是会有data1和data2两个10MB的数组是被变量引用必须存活的，此时Eden区已经占用了六七十MB空间了，接着是data3变量依次指向了两个10MB的数组，这是为了在1s内触发Young GC的。

10、基于jstat分析程序运行的状态

接着我们基于jstat分析程序运行的状态，启动程序后立马采用jstat监控其运行状态可以看到如下的信息：

```
 S0C     S1C     S0U     S1U      EC       EU        OC         OU
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0     0.0    81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    587.1   81920.0  10240.0   102400.0   30722.1
10240.0 10240.0 838.4    0.0    81920.0  22008.9   102400.0   51202.1
10240.0 10240.0  0.0    914.5   81920.0  32275.3   102400.0   61444.2
10240.0 10240.0 696.1    0.0    81920.0  42532.2   102400.0   30724.1
10240.0 10240.0  0.0    778.4   81920.0  52783.1   102400.0   40966.2
10240.0 10240.0 947.0    0.0    81920.0  63030.1   102400.0   51206.2
10240.0 10240.0  0.0    815.1   81920.0  73274.6   102400.0   61446.2
10240.0 10240.0  0.0     0.0    81920.0  10240.0   102400.0   31283.8
10240.0 10240.0  0.0     0.0    81920.0  22080.5   102400.0   51765.9
10240.0 10240.0  0.0     0.0    81920.0  32321.2   102400.0   62008.0
10240.0 10240.0  0.0     0.0    81920.0  42561.8   102400.0   31287.9
10240.0 10240.0  0.0     0.0    81920.0  52802.1   102400.0   41530.0
10240.0 10240.0  0.0     0.0    81920.0  63042.3   102400.0   51770.0
10240.0 10240.0  0.0     0.0    81920.0  73282.4   102400.0   62010.0
10240.0 10240.0  0.0     0.0    81920.0  10240.0   102400.0   31283.8
10240.0 10240.0  0.0     0.0    81920.0  22082.5   102400.0   51765.9
10240.0 10240.0  0.0     0.0    81920.0  32322.6   102400.0   62008.0
10240.0 10240.0  0.0     0.0    81920.0  42562.6   102400.0   31287.9
10240.0 10240.0  0.0     0.0    81920.0  52802.6   102400.0   41527.9
10240.0 10240.0  0.0     0.0    81920.0  63042.6   102400.0   51770.0
10240.0 10240.0  0.0     0.0    81920.0  73282.7   102400.0   62010.0
10240.0 10240.0  0.0     0.0    81920.0  10240.0   102400.0   31283.8
```

接着我们一点点来分析这个jvm的运行状态。首先我们先看如下一行截图：



```
 S0C    S1C    S0U    S1U      EC       EU        OC         OU
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0   587.1  81920.0  10240.0   102400.0   30722.1
```

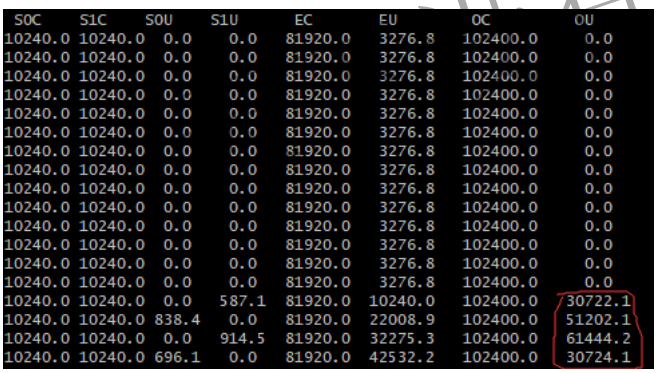
在这里最后一行，可以清晰看到，程序运行起来之后，突然在一秒内就发生了一次Young GC，这是为什么呢？

很简单，按照我们上述的代码，他一定会在一秒内触发一次Young GC的。

Young GC过后，我们发现S1U，也就是一个Survivor区中有587KB的存活对象，这应该就是那些未知对象了。

然后我们明显看到在OU中多出来了30MB左右的对象，因此可以确定，在这次Young GC的时候，有30MB的对象存活了，此时因为Survivor区域放不下，所以直接进入老年代了。

我们接着看下面的截图：



```
 S0C    S1C    S0U    S1U      EC       EU        OC         OU
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0    0.0   81920.0   3276.8   102400.0     0.0
10240.0 10240.0  0.0   587.1  81920.0  10240.0   102400.0   30722.1
10240.0 10240.0 838.4   0.0   81920.0  22008.9   102400.0   51202.1
10240.0 10240.0  0.0   914.5  81920.0  32275.3   102400.0   61444.2
10240.0 10240.0 696.1   0.0   81920.0  42532.2   102400.0   30724.1
```

大家看上图中红圈的部分，很明显每秒会发生一次Young GC，都会导致20MB~30MB左右的对象进入老年代

因为每次Young GC都会存活下来这么多对象，但是Survivor区域是放不下的，所以都会直接进入老年代。

此时看到老年代的对象占用从30KB一路到60MB左右，此时突然在60MB之后下一秒，明显发生了一次Full GC，对老年代进行了垃圾回收，因为此时老年代重新变成30MB了。

为啥会这样?

很简单，老年代总共就100MB左右，已经占用了60MB了，此时如果发生一次Young GC，有30MB存活对象要放入老年代的话，你还要放30MB对象，明显老年代就要不够了，此时必须会进行Full GC，回收掉之前那60MB对象，然后再放入进去新的30MB对象。

所以大家可以看到，按照我们的这段代码，几乎是每秒新增80MB左右，触发每秒1次Young GC，每次Young GC后存活下来20MB~30MB的对象，老年代每秒新增20MB~30MB的对象，触发老年代几乎三秒一次Full GC，是不是跟我们上面的案例中分析的场景很类似？Young GC太频繁，而且每次GC后存活对象太多，频繁进入老年代，频繁触发老年代的GC。

那么Young GC和Full GC的耗时呢？看下图:

YGC	YGCT	FGC	FGCT	GCT
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
0	0.000	0	0.000	0.000
1	0.027	0	0.000	0.027
2	0.044	1	0.000	0.045
3	0.064	1	0.000	0.064
4	0.065	2	0.002	0.067
5	0.068	2	0.002	0.070
6	0.073	2	0.002	0.075
7	0.091	3	0.003	0.094
9	0.119	4	0.011	0.131
10	0.124	5	0.011	0.135
11	0.126	5	0.011	0.137
12	0.127	6	0.015	0.142
13	0.132	6	0.015	0.146
14	0.138	6	0.015	0.153
15	0.143	7	0.015	0.158
17	0.148	8	0.023	0.171
18	0.154	9	0.023	0.176
19	0.156	9	0.023	0.179
20	0.157	10	0.025	0.183
21	0.162	10	0.025	0.188
22	0.165	10	0.025	0.190
23	0.167	11	0.025	0.193
25	0.172	12	0.033	0.205
26	0.176	13	0.033	0.209
27	0.179	13	0.033	0.211
28	0.179	14	0.034	0.213

大家看上图，有没有发现Young GC特别坑爹，28次Young GC，结果耗费了180毫秒，平均下来一次Young GC要6毫秒左右。但是14次Full GC才耗费34毫秒，平均下来一次Full GC才耗费两三毫秒。这是为什么呢?

很简单，按照上述程序，每次Full GC都是由Young GC触发的，因为Young GC过后存活对象太多要放入老年代，老年代内存不够了触发Full GC，所以必须得等Full GC执行完毕了，Young GC才能把存活对象放入老年代，才算结束。这就导致Young GC也是速度非常慢。

11、对JVM性能进行优化

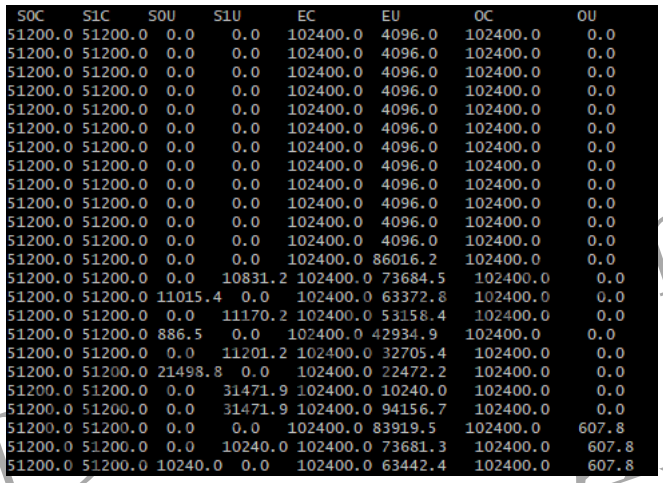
接着我们按照之前学习的思路对JVM进行优化，很简单，他最大的问题就是每次Young GC过后存活对象太多了，导致频繁进入老年代，频繁触发Full GC

我们只需要调大年轻代的内存空间，增加Survivor的内存即可，看如下JVM参数：

-XX:NewSize=209715200 -XX:MaxNewSize=209715200 -XX:InitialHeapSize=314572800 -XX:MaxHeapSize=314572800 -XX:SurvivorRatio=2 -XX:MaxTenuringThreshold=15 -XX:PretenureSizeThreshold=20971520 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:gc.log

我们把堆大小调大为了300MB，年轻代给了200MB，同时“-XX:SurvivorRatio=2”表明，Eden:Survivor:Survivor的比例为2:1:1，所以Eden区是100MB，每个Survivor区是50MB，老年代也是100MB。

接着我们用这个JVM参数运行程序，用jstat来监控其运行状态如下：

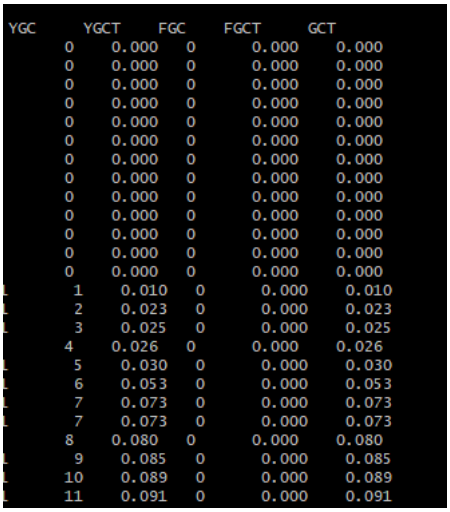


```
 S0C      S1C      S0U      S1U       EC        EU        OC         OU
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0  4096.0   102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0 86016.2   102400.0      0.0
51200.0 51200.0   0.0    10831.2 102400.0 73684.5    102400.0      0.0
51200.0 51200.0 11015.4    0.0    102400.0 63372.8    102400.0      0.0
51200.0 51200.0   0.0    11170.2 102400.0 53158.4    102400.0      0.0
51200.0 51200.0 886.5      0.0    102400.0 42934.9   102400.0      0.0
51200.0 51200.0   0.0    11201.2 102400.0 32705.4    102400.0      0.0
51200.0 51200.0 21498.8    0.0    102400.0 22472.2    102400.0      0.0
51200.0 51200.0   0.0    31471.9 102400.0 10240.0    102400.0      0.0
51200.0 51200.0   0.0    31471.9 102400.0 94156.7    102400.0      0.0
51200.0 51200.0   0.0      0.0    102400.0 83919.5   102400.0    607.8
51200.0 51200.0   0.0    10240.0 102400.0 73681.3    102400.0     607.8
51200.0 51200.0 10240.0    0.0    102400.0 63442.4    102400.0     607.8
```

在上述截图里，大家可以清晰看到，每秒的Young gC过后，都会有20MB左右的存活对象进入Survivor，但是每个Survivor区都是50MB的大小，因此可以轻松容纳，而且一般不会过50%的动态年龄判定的阈值。

我们可以清晰看到每秒触发Yuong GC过后，几乎就没有对象会进入老年代，最终就600KB的对象进入了老年代里，其他就没有对象进入老年代了。

再看下面的截图：



```
YGC     YGCT    FGC    FGCT     GCT
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
      0    0.000   0       0.000    0.000
1     1    0.010   0       0.000    0.010
1     2    0.023   0       0.000    0.023
1     3    0.025   0       0.000    0.025
      4    0.026   0       0.000    0.026
1     5    0.030   0       0.000    0.030
1     6    0.053   0       0.000    0.053
1     7    0.073   0       0.000    0.073
1     7    0.073   0       0.000    0.073
      8    0.080   0       0.000    0.080
1     9    0.085   0       0.000    0.085
1    10    0.089   0       0.000    0.089
1    11    0.091   0       0.000    0.091
```

我们可以看到，只有Young GC，没有Full GC，而且11次Young GC才不过9毫秒，平均一次GC1毫秒都不到，没有Full GC干扰之后，Young GC的性能极高。

所以，其实这个案例就优化成功了，同样的程序，仅仅是调整了内存分配比例，立马就大幅度提升了JVM的性能，几乎把Full GC给消灭掉了。

12、今日思考题

这周内容学完之后，大家不用多说，直接自己按照文章的思路，写出来模拟代码，尝试用jstat去观察一下运行状态

尤其是今天的文章，需要你去亲自动手写代码，观察优化前频繁full gc的问题，然后优化jvm参数之后，再看看优化后的效果，相信大家基本从实操层面就完全明白jvm的优化方法了。

End
