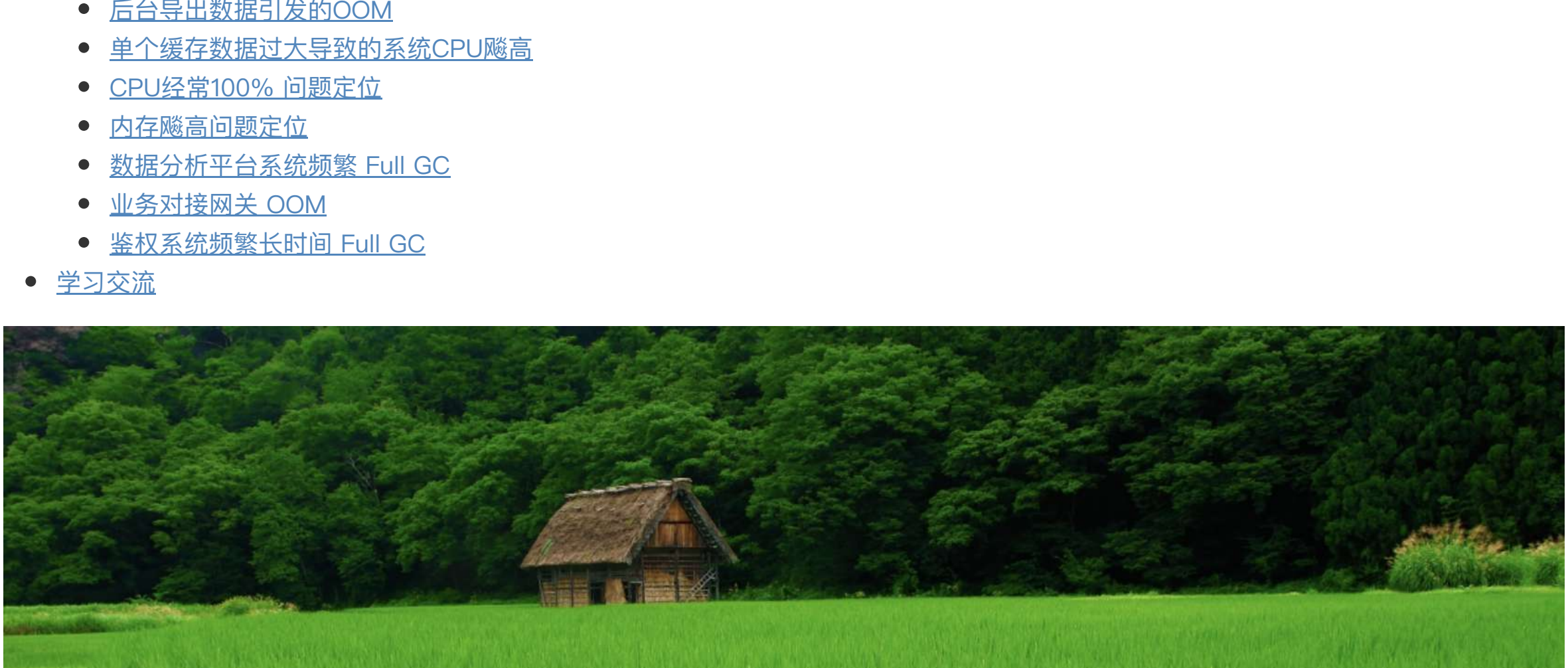

> 转载文章，讲解JVM的常用调优策略，以及一些工作中遇到的JVM调优实例。

这篇文章并非原创，感觉原文实战操作比较强，特此转载，留以后用，原文地址：https://juejin.cn/post/6949806402743304206#heading-21

对于需要进行JVM调优，或者遇到JVM相关问题，且不知如何去解决的同学，这篇文章真的非常值得阅读。

前言

JVM调优听起来很高大上，但是要认识到，JVM调优应该是Java性能优化的最后一颗子弹。

比较认可廖雪峰老师的观点，要认识到JVM调优不是常规手段，性能问题一般第一选择是优化程序，最后的选择才是进行JVM调优。

常用调优策略

这里还是要提一下，及时确定要进行JVM调优，也不要陷入"知见障"，进行分析之后，发现可以通过优化程序提升性能，仍然首选优化程序。

选择合适的垃圾回收器

- CPU单核，那么毫无疑问Serial 垃圾收集器是你唯一的选择。
- CPU多核，关注吞吐量 ，那么选择PS+PO组合。
- CPU多核，关注用户停顿时间，JDK版本1.6或者1.7，那么选择CMS。
- CPU多核，关注用户停顿时间，JDK1.8及以上，JVM可用内存6G以上，那么选择G1。

参数配置：

```
//设置Serial垃圾收集器（新生代）
开启：-XX:+UseSerialGC

//设置PS+PO,新生代使用功能Parallel Scavenge 老年代将会使用Parallel Old收集器
开启 -XX:+UseParallelOldGC

//CMS垃圾收集器（老年代）
开启 -XX:+UseConcMarkSweepGC

//设置G1垃圾收集器
开启 -XX:+UseG1GC
```

调整内存大小

现象：垃圾收集频率非常频繁。

原因：如果内存太小，就会导致频繁的需要进行垃圾收集才能释放出足够的空间来创建新的对象，所以增加堆内存大小的效果是非常显而易见的。

注意：如果垃圾收集次数非常频繁，但是每次能回收的对象非常少，那么这个时候并非内存太小，而可能是内存泄露导致对象无法回收，从而造成频繁GC。

参数配置：

```
//设置堆初始值
指令1：-Xms2g
指令2：-XX:InitialHeapSize=2048m

//设置堆区最大值
指令1：`-Xmx2g`
指令2： -XX:MaxHeapSize=2048m

//新生代内存配置
指令1：-Xmn512m
指令2：-XX:MaxNewSize=512m
```

设置符合预期的停顿时间

现象：程序间接性的卡顿

原因：如果没有确切的停顿时间设定，垃圾收集器以吞吐量为主，那么垃圾收集时间就会不稳定。

注意：不要设置不切实际的停顿时间，单次时间越短也意味着需要更多的GC次数才能回收完原有数量的垃圾.

参数配置：

```
//GC停顿时间，垃圾收集器会尝试用各种手段达到这个时间
-XX:MaxGCPauseMillis
```

调整内存区域大小比率

现象：某一个区域的GC频繁，其他都正常。

原因：如果对应区域空间不足，导致需要频繁GC来释放空间，在JVM堆内存无法增加的情况下，可以调整对应区域的大小比率。

注意：也许并非空间不足，而是因为内存泄造成内存无法回收。从而导致GC频繁。

参数配置：

```
//survivor区和Eden区大小比率
指令：-XX:SurvivorRatio=6  //S区和Eden区占新生代比率为1:6,两个S区2:6

//新生代和老年代的占比
-XX:NewRatio=4  //表示新生代:老年代 = 1:4 即老年代占整个堆的4/5；默认值=2
```

调整对象升老年代的年龄

现象：老年代频繁GC，每次回收的对象很多。

原因：如果升代年龄小，新生代的对象很快就进入老年代了，导致老年代对象变多，而这些对象其实在随后的很短时间内就可以回收，这时候可以调整对象的升级代年龄，让对象不那么容易进入老年代解决老年代空间不足频繁GC问题。

注意：增加了年龄之后，这些对象在新生代的时间会变长可能导致新生代的GC频率增加，并且频繁复制这些对象新生的GC时间也可能变长。

配置参数：

```
//进入老年代最小的GC年龄,年轻代对象转换为老年代对象最小年龄值，默认值7
 -XX:InitialTenuringThreshol=7
```

调整大对象的标准

现象：老年代频繁GC，每次回收的对象很多,而且单个对象的体积都比较大。

原因：如果大量的大对象直接分配到老年代，导致老年代容易被填满而造成频繁GC，可设置对象直接进入老年代的标准。

注意：这些大对象进入新生代后可能会使新生代的GC频率和时间增加。

配置参数：

```
//新生代可容纳的最大对象,大于则直接会分配到老年代，0代表没有限制。
 -XX:PretenureSizeThreshold=1000000
```

调整GC的触发时机

现象：CMS，G1 经常 Full GC，程序卡顿严重。

原因：G1和CMS 部分GC阶段是并发进行的，业务线程和垃圾收集线程一起工作，也就说明垃圾收集的过程中业务线程会生成新的对象，所以在GC的时候需要预留一部分内存空间来容纳新产生的对象，如果这个时候内存空间不足以容纳新产生的对象，那么JVM就会停止并发收集暂停所有业务线程（STW）来保证垃圾收集的正常运行。这个时候可以调整GC触发的时机（比如在老年代占用60%就触发GC），这样就可以预留足够的空间来让业务线程创建的对象有足够的空间分配。

注意：提早触发GC会增加老年代GC的频率。

配置参数：

```
//使用多少比例的老年代后开始CMS收集，默认是68%，如果频繁发生SerialOld卡顿，应该调小
-XX:CMSInitiatingOccupancyFraction

//G1混合垃圾回收周期中要包括的旧区域设置占用率阈值。默认占用率为 65%
-XX:G1MixedGCLiveThresholdPercent=65
```

调整 JVM本地内存大小

现象：GC的次数、时间和回收的对象都正常，堆内存空间充足，但是报OOM

原因： JVM除了堆内存之外还有一块堆外内存，这片内存也叫本地内存，可是这块内存区域不足了并不会主动触发GC，只有在堆内存区域触发的时候顺带会把本地内存回收了，而一旦本地内存分配不足就会直接报OOM异常。

注意： 本地内存异常的时候除了上面的现象之外，异常信息可能是OutOfMemoryError：Direct buffer memory。 解决方式除了调整本地内存大小之外，也可以在出现此异常时进行捕获，手动触发GC（System.gc()）。

配置参数：

```
XX:MaxDirectMemorySize
```

JVM调优实例

以下是整理自网络的一些JVM调优实例：

网站流量浏览量暴增后，网站反应页面响很慢

1、问题推测：在测试环境测速度比较快，但是一到生产就变慢，所以推测可能是因为垃圾收集导致的业务线程停顿。

2、定位：为了确认推测的正确性，在线上通过jstat -gc 指令 看到JVM进行GC 次数频率非常高，GC所占用的时间非常长，所以基本推断就是因为GC频率非常高，所以导致业务线程经常停顿，从而造成网页反应很慢。

3、解决方案：因为网页访问量很高，所以对象创建速度非常快，导致堆内存容易填满从而频繁GC，所以这里问题在于新生代内存太小，所以这里可以增加JVM内存就行了，所以初步从原来的2G内存增加到16G内存。

4、第二个问题：增加内存后的确平常的请求比较快了，但是又出现了另外一个问题，就是不定期的会间断性的卡顿，而且单次卡顿的时间要比之前要长很多。

5、问题推测：练习到是之前的优化加大了内存，所以推测可能是因为内存加大了，从而导致单次GC的时间变长从而导致间接性的卡顿。

6、定位：还是通过jstat -gc 指令 查看到 的确FGC次数并不是很高，但是花费在FGC上的时间是非常高的,根据GC日志 查看到单次FGC的时间有达到几十秒的。

7、解决方案： 因为JVM默认使用的是PS+PO的组合，PS+PO垃圾标记和收集阶段都是STW，所以内存加大了之后，需要进行垃圾回收的时间就变长了，所以这里要想避免单次GC时间过长，所以需要更换并发类的收集器，因为当前的JDK版本为1.7，所以最后选择CMS垃圾收集器，根据之前垃圾收集情况设置了一个预期的停顿的时间，上线后网站再也没有了卡顿问题。

后台导出数据引发的OOM

问题描述：公司的后台系统，偶发性的引发OOM异常，堆内存溢出。

1、因为是偶发性的，所以第一次简单的认为就是堆内存不足导致，所以单方面的加大了堆内存从4G调整到8G。

2、但是问题依然没有解决，只能从堆内存信息下手，通过开启了-XX:+HeapDumpOnOutOfMemoryError参数 获得堆内存的dump文件。

3、VisualVM 对 堆dump文件进行分析，通过VisualVM查看到占用内存最大的对象是String对象，本来想跟踪着String对象找到其引用的地方，但dump文件太大，跟踪进去的时候总是卡死，而String对象占用比较多也比较正常，最开始也没有认定就是这里的问题，于是就从线程信息里面找突破点。

4、通过线程进行分析，先找到了几个正在运行的业务线程，然后逐一跟进业务线程看了下代码，发现有个引起我注意的方法，导出订单信息。

5、因为订单信息导出这个方法可能会有几万的数据量，首先要从数据库里面查询出来订单信息，然后把订单信息生成excel，这个过程会产生大量的String对象。

6、为了验证自己的猜想，于是准备登录后台去测试下，结果在测试的过程中发现到处订单的按钮前端居然没有做点击后按钮置灰交互事件，结果按钮可以一直点，因为导出订单数据本来就非常慢，使用的人员可能发现点击后很久后页面都没反应，结果就一直点，结果就大量的请求进入到后台，堆内存产生了大量的订单对象和EXCEL对象，而且方法执行非常慢，导致这一段时间内这些对象都无法被回收，所以最终导致内存溢出。

7、知道了问题就容易解决了，最终没有调整任何JVM参数，只是在前端的导出订单按钮上加上了置灰状态，等后端响应之后按钮才可以进行点击，然后减少了查询订单信息的非必要字段来减少生成对象的体积，然后问题就解决了。

单个缓存数据过大导致的系统CPU飚高

1、系统发布后发现CPU一直飚高到600%，发现这个问题后首先要做的是定位到是哪个应用占用CPU高，通过top 找到了对应的一个java应用占用CPU资源600%。

2、如果是应用的CPU飚高，那么基本上可以定位可能是锁资源竞争，或者是频繁GC造成的。

3、所以准备首先从GC的情况排查，如果GC正常的话再从线程的角度排查，首先使用jstat -gc PID 指令打印出GC的信息，结果得到得到的GC 统计信息有明显的异常，应用在运行了才几分钟的情况下GC的时间就占用了482秒，那么问这很明显就是频繁GC导致的CPU飚高。

4、定位到了是GC的问题，那么下一步就是找到频繁GC的原因了，所以可以从两方面定位了，可能是哪个地方频繁创建对象，或者就是有内存泄露导致内存回收不掉。

5、根据这个思路决定把堆内存信息dump下来看一下，使用jmap -dump 指令把堆内存信息dump下来（堆内存空间大的慎用这个指令否则容易导致会影响应用，因为我们的堆内存空间才2G所以也就没考虑这个问题了）。

6、把堆内存信息dump下来后，就使用visualVM进行离线分析了，首先从占用内存最多的对象中查找，结果排名第三看到一个业务VO占用堆内存约10%的空间，很明显这个对象是有问题的。

7、通过业务对象找到了对应的业务代码，通过代码的分析找到了一个可疑之处，这个业务对象是查看新闻资讯信息生成的对象，由于想提升查询的效率，所以把新闻资讯保存到了redis缓存里面，每次调用资讯接口都是从缓存里面获取。

8、把新闻保存到redis缓存里面这个方式是没有问题的，有问题的是新闻的50000多条数据都是保存在一个key里面，这样就导致每次调用查询新闻接口都会从redis里面把50000多条数据都拿出来，再做筛选分页拿出10条返回给前端。50000多条数据也就意味着会产生50000多个对象，每个对象280个字节左右，50000个对象就有13.3M，这就意味着只要查看一次新闻信息就会产生至少13.3M的对象，那么并发请求量只要到10，那么每秒钟都会产生133M的对象，而这种大对象会被直接分配到老年代，这样的话一个2G大小的老年代内存，只需要几秒就会塞满，从而触发GC。

9、知道了问题所在后那么就容易解决了，问题是因为单个缓存过大造成的，那么只需要把缓存减小就行了，这里只需要把缓存以页的粒度进行缓存就行了，每个key缓存10条作为返回给前端1页的数据，这样的话每次查询新闻信息只会从缓存拿出10条数据，就避免了此问题的 产生。

CPU经常100% 问题定位

问题分析：CPU高一定是某个程序长期占用了CPU资源。

1、所以先需要找出那个进行占用CPU高。

```
top  列出系统各个进程的资源占用情况。
```

2、然后根据找到对应进行里哪个线程占用CPU高。

```
top -Hp 进程ID   列出对应进程里面的线程占用资源情况
```

3、找到对应线程ID后，再打印出对应线程的堆栈信息

```
printf "%x\n"  PID    把线程ID转换为16进制。
jstack PID 打印出进程的所有线程信息，从打印出来的线程信息中找到上一步转换为16进制的线程ID对应的线程信息。
```

4、最后根据线程的堆栈信息定位到具体业务方法,从代码逻辑中找到问题所在。

```
查看是否有线程长时间的watting 或blocked
如果线程长期处于watting状态下， 关注watting on xxxxxx，说明线程在等待这把锁，然后根据锁的地址找到持有锁的线程。
```

内存飚高问题定位

分析： 内存飚高如果是发生在java进程上，一般是因为创建了大量对象所导致，持续飚高说明垃圾回收跟不上对象创建的速度，或者内存泄露导致对象无法回收。

1、先观察垃圾回收的情况

```
jstat -gc PID 1000 查看GC次数，时间等信息，每隔一秒打印一次。
jmap -histo PID | head -20   查看堆内存占用空间最大的前20个对象类型,可初步查看是哪个对象占用了内存。
```

如果每次GC次数频繁，而且每次回收的内存空间也正常，那说明是因为对象创建速度快导致内存一直占用很高；如果每次回收的内存非常少，那么很可能是因为内存泄露导致内存一直无法被回收。

2、导出堆内存文件快照

```
jmap -dump:live,format=b,file=/home/myheapdump.hprof PID  dump堆内存信息到文件。
```

3、使用visualVM对dump文件进行离线分析,找到占用内存高的对象，再找到创建该对象的业务代码位置，从代码和业务场景中定位具体问题。

数据分析平台系统频繁 Full GC

平台主要对用户在 App 中行为进行定时分析统计，并支持报表导出，使用 CMS GC 算法。

数据分析师在使用中发现系统页面打开经常卡顿，通过 jstat 命令发现系统每次 Young GC 后大约有 10% 的存活对象进入老年代。

原来是因为 Survivor 区空间设置过小，每次 Young GC 后存活对象在 Survivor 区域放不下，提前进入老年代。

通过调大 Survivor 区，使得 Survivor 区可以容纳 Young GC 后存活对象，对象在 Survivor 区经历多次 Young GC 达到年龄阈值才进入老年代。

调整之后每次 Young GC 后进入老年代的存活对象稳定运行时仅几百 Kb，Full GC 频率大大降低。

业务对接网关 OOM

网关主要消费 Kafka 数据，进行数据处理计算然后转发到另外的 Kafka 队列，系统运行几个小时候出现 OOM，重启系统几个小时之后又 OOM。

通过 jmap 导出堆内存，在 eclipse MAT 工具分析才找出原因：代码中将某个业务 Kafka 的 topic 数据进行日志异步打印，该业务数据量较大，大量对象堆积在内存中等待被打印，导致 OOM。

鉴权系统频繁长时间 Full GC

系统对外提供各种账号鉴权服务，使用时发现系统经常服务不可用，通过 Zabbix 的监控平台监控发现系统频繁发生长时间 Full GC，且触发时老年代的堆内存通常并没有占满，发现原来是业务代码中调用了 System.gc()。

学习交流

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。