高并发场景下JVM调优实践之路

一、背景

2022年2月，收到反馈，我司APP某核心接口高峰期RT超1400ms影响用户体验。通过**自研监控平台**发现，RT慢主要体现在P99高，怀疑与在高负载情况下与服务GC有关！

1.1 问题分析

在观察周期里，现象如下

- **平均每10分钟Young GC次数66次，峰值为470次；**
- **平均每10分钟Full GC次数0.25次，峰值5次；**

问题：Full GC非常频繁，Young GC在特定的时段也比较频繁，存在较大的优化空间。

由于对GC停顿的优化是降低接口的P99时延一个有效的手段，所以决定对该核心服务进行JVM调优。这里我们假设排除了代码层面的问题。

1.2 确定目标

核心目标：接口P99时延降低30%、减少Young GC和Full GC次数、停顿时长、单次停顿时长。

核心目标拆分为三个小目标：

目标1：高负载（单机1000 QPS以上）

- Young GC次数减少20%-30%
- Full GC次数减少50%以上
- 单次、累积Full GC耗时减少50%以上
- 服务发布不触发Full GC

目标2：中负载（单机500-600）

- Young GC次数减少20%-30%
- Young GC累积耗时减少20%
- Full GC次数不高于4次/天
- 服务发布不触发Full GC

目标3：低负载（单机200 QPS以下）

- Young GC次数减少20%-30%
- Young GC累积耗时减少20%
- Full GC次数不高于1次/天
- 服务发布不触发Full GC

问：优化的目标为什么需要根据负载分别制定?

- 由于GC的行为与并发(负载)是正相关的，不管负载去制定JVM优化目标是不合理的。
- 因为，当并发比较高时，不管如何调优，Young GC总会很频繁，总会有不该晋升的对象晋升触发Full GC

二、分析当前JVM参数存在的问题

当前服务的JVM配置参数如下：

```
-Xms4096M -Xmx4096M -Xmn1024M
-XX:PermSize=512M
-XX:MaxPermSize=512M

# 初始堆大小Xms
# 最大堆大小Xmx
# 年轻代大小Xmn
# 方法区大小PermSize
# 占整个堆内存的最大值MaxPermSize
```

单纯从参数上分析，存在以下问题：

问题1：未显示指定GC垃圾收集器

JDK 8默认搜集器为ParrallelGC即：

- Young区采用Parallel Scavenge
- 老年代采用Parallel Old进行收集

这套配置的特点是**吞吐量**优先，一般适用于后台任务型服务器。

比如批量订单处理、科学计算等对吞吐量敏感，对时延不敏感的场景，当前服务是视频与用户交互的门户，对时延非常敏感，因此不适合使用默认收集器ParrallelGC，应选择更合适的收集器。

问题2：Young区配比不合理

当前服务主要提供API，这类服务的特点是常驻对象会比较少，**绝大多数对象的生命周期都比较短**，经过一次或两次Young GC就会消亡。

再看下当前JVM配置：整个堆为4G，Young区总共1G，默认-XX:SurvivorRatio=8，即有效大小为0.9G，**老年代常驻对象大小约400M。**

不合理导致的直接结果：在服务高负载情况下。请求并发较大，Young区中Eden + S0区域会迅速填满，进而Young GC会比较频繁。**另外会引起本应被Young GC回收的对象过早晋升，这也会增加Full GC的频率**，同时单次收集的区域也会增大，由于Old区使用的是ParallelOld无法与用户线程并发执行，**导致服务长时间停顿，可用性下降， P99响应时间上升。**

那么年轻代给多少合适呢?

> 鞋子合不合脚，只有脚知道，试过才可以体会到

问题3：未设置Metaspace

未设置-XX:MetaspaceSize和-XX:MaxMetaspaceSize

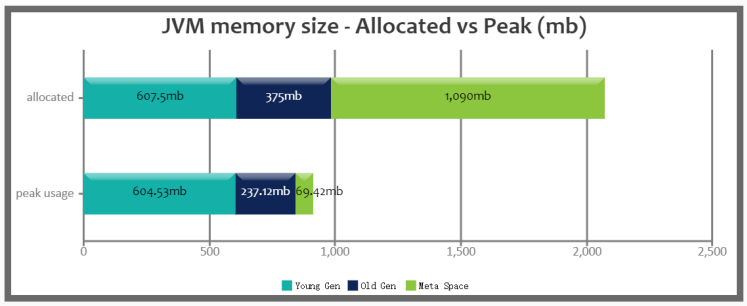
```
Perm区在jdk 1.8已经过时，被Meta区取代，
因此-XX:PermSize=512M -XX:MaxPermSize=512M配置会被忽略，

真正控制Meta区GC的参数为
-XX:MetaspaceSize:
Metaspace初始大小，64位机器默认为21M左右

-XX:MaxMetaspaceSize:
Metaspace的最大值，64位机器默认为18446744073709551615Byte，
可以理解为无上限

-XX:MaxMetaspaceExpansion:
增大触发metaspace GC阈值的最大要求

-XX:MinMetaspaceExpansion:
增大触发metaspace GC阈值的最小要求，默认为340784Byte
```

这样服务在启动和发布的过程中，元数据区域达到21M时会触发一次Full GC (Metadata GC Threshold)，随后随着元数据区域的扩张，会夹杂若干次Full GC (Metadata GC Threshold)，使服务发布稳定性和效率下降。

此外如果服务使用了大量动态类生成技术的话，也会因为这个机制产生不必要的Full GC (Metadata GC Threshold)。

JVM memory size

Generation	Allocated	Peak
Young Generation	607.5 mb	604.53 mb
Old Generation	375 mb	237.12 mb
Meta Space	1.06 gb	69.42 mb
Young + Old + Meta space	2.91 gb	897.91 mb



三、优化方案

3.1 优化方案

上面已分析出当前配置存在的较为明显的不足，下面优化方案主要先针对性解决这些问题，之后再结合效果决定是否继续深入优化。

当前主流/优秀的搜集器包含：

- **Parrallel Scavenge + Parrallel Old**：吞吐量优先，后台任务型服务适合；
- **ParNew + CMS**：经典的低停顿搜集器，绝大多数商用、延时敏感的服务在使用；
- **G1**：JDK 9默认搜集器，堆内存比较大（6G-8G以上）的时候表现出比较高吞吐量和短暂的停顿时间；
- **ZGC**：JDK 11中推出的一款低延迟垃圾回收器，目前处在实验阶段；

结合当前服务的实际情况(堆大小，可维护性)，我们选择ParNew + CMS方案是比较合适的。

参数选择的原则如下：

1）Meta区域的大小一定要指定

Meta区域的大小一定要指定，且MetaspaceSize和MaxMetaspaceSize大小应设置一致，具体多大要结合线上实例的情况，通过jstat -gc可以获取该服务线上实例的情况。

```
# jstat -gc 31247
S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT
37888.0 37888.0 0.0 32438.5 972800.0 403063.5 3145728.0 2700882.3 167320.0
152285.0 18856.0 16442.4 15189 597.209 65 70.447 667.655
```

可以看出MU在150M左右，因此-XX:MetaspaceSize=256M ，-XX:MaxMetaspaceSize=256M是比较合理的。

2）Young区不是越大越好

当堆大小一定时，Young区越大，Young GC的频率一定越小，但Old区域就会变小，如果太小，稍微晋升一些对象就会触发Full GC得不偿失。

如果Young区过小，Young GC就会比较频繁，这样Old区就会比较大，单次Full GC的停顿就会比较大。因此Young区的大小需要结合服务情况，分几种场景进行比较，最终获得最合适的配置。

基于以上原则，以下为4种参数组合：

1.ParNew +CMS，Young区扩大1倍

```
-Xms4096M -Xmx4096M -Xmn2048M
-XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+CMSScavengeBeforeRemark
```

2.ParNew +CMS，Young区扩大1倍，

去除-XX:+CMSScavengeBeforeRemark

> （使用【-XX:CMSScavengeBeforeRemark】参数可以做到在重新标记前先执行一次新生代GC）。

因为老年代和年轻代之间的对象存在**跨代引用**，因此老年代进行GC Roots追踪时，同样也会扫描年轻代，而如果能够在重新标记前先执行一次新生代GC，那么就可以少扫描一些对象，重新标记阶段的性能也能因此提升。）

```
-Xms4096M -Xmx4096M -Xmn2048M
-XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
```

3.ParNew +CMS，Young区扩大0.5倍

```
-Xms4096M -Xmx4096M -Xmn1536M
-XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+CMSScavengeBeforeRemark
```

4.ParNew +CMS，Young区不变

```
-Xms4096M -Xmx4096M -Xmn1024M
-XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+CMSScavengeBeforeRemark
```

下面，我们需要在压测环境，对不同负载下4种方案的实际表现进行比较，分析，验证。

3）使用CICD进行自动化测试，并采集测试结果

```
pipeline {
   agent  any

   options {
        //保持构建的最大个数
        buildDiscarder(logRotator(numToKeepStr: '20'))
   }

   tools {
        //需要在jenkins配置maven环境，名称为maven3.6.1
        maven 'mvn3.6.1'
   }

   // 常量参数
   environment{
        //工作空间绝对路径
        WORKSPACE_l='${WORKSPACE}'
        //手动拼接
        WORKSPACE_t='${JENKINS_HOME}/workspace/${JOB_NAME}'
        repo_code_dir='/usr/local/src/${artifactId}/${VERSION}/'
   }

   stages {
      stage('高01-启动') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73
'/usr/local/src/${artifactId}/latest/stop.sh'"
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73
'/usr/local/src/${artifactId}/latest/startup-01.sh'"
        }

      }
      stage('高01-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34
'/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/01-jvm-high-load.jmx
-l /root/01-jvm-high-load.jtl '"
        }
```

```
      }
      stage('高02-启动') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-02.sh'"
        }
      }
      stage('高02-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/01-jvm-high-load.jmx -l /root/02-jvm-high-load.jtl '"
        }
      }
      stage('高03-启动') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
            sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-03.sh'"
        }
      }
      stage('高03-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/01-jvm-high-load.jmx -l /root/03-jvm-high-load.jtl '"
        }
      }
      stage('高04-启动') {
         steps {
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-04.sh'"
         }
       }
      stage('高04-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/01-jvm-high-load.jmx -l /root/04-jvm-high-load.jtl '"
        }
      }
      stage('高05-启动') {
         steps {
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-05.sh'"
         }
       }
      stage('高05-测试') {
```

```
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/01-jvm-high-load.jmx -l /root/05-jvm-high-load.jtl '"
        }
      }

      stage('中01-启动') {
         steps {
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-06.sh'"
         }
       }
      stage('中01-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/02-jvm-med-load.jmx -l /root/06-jvm-med-load.jtl '"
        }
      }
      stage('中02-启动') {
         steps {
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-07.sh'"
         }
       }
      stage('中02-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/02-jvm-med-load.jmx -l /root/07-jvm-med-load.jtl '"

        }
      }
      stage('中03-启动') {
         steps {
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/stop.sh'"
             sh "ssh -p ${ts_port} ${ts_user}@39.103.219.73 '/usr/local/src/${artifactId}/latest/startup-08.sh'"
         }
       }
      stage('中03-测试') {
        steps {
            sh "ssh -p ${ts_port} ${ts_user}@39.103.182.34 '/usr/local/apache-jmeter-5.4.1/bin/jmeter -n -t /root/02-jvm-med-load.jmx -l /root/08-jvm-med-load.jtl '"
        }
      }
   }
}
```

```
117
```

3.1 压测环境验证

高负载场景(1100 QPS)GC表现

方案	Young GC次数峰值/均值	Young GC累积耗时/单次耗时	Full GC次数峰值/均值	Full GC累积耗时/单次耗时	接口P99	接口P95
Parrallel Scavenge + Parrallel Old（当前方案）	峰值 68次/分钟 均值 63次/分钟	累积耗时 2959ms/分钟 单次耗时 46ms/次/分钟	峰值 1次/分钟 均值 0.315次/分钟	累积耗时峰值 1566ms/分钟 累积耗时均值 429ms/分钟	1416ms	1247ms
ParNew +CMS，Young区扩大1倍	峰值 39次/分钟 均值 36次/分钟	累积耗时 2481ms/分钟 单次耗时 70ms/次/分钟	峰值 1次/分钟 均值 0.4次/分钟	累积耗时峰值 299ms/分钟 累积耗时均值 80ms/分钟	848ms	736ms
ParNew +CMS，Young区扩大1倍，去除-XX:+CMSScavengeBeforeRemark	峰值 38次/分钟 均值 36次/分钟	累积耗时 2364ms/分钟 单次耗时 65ms/次/分钟	峰值 1次/分钟 均值 0.39次/分钟	累积耗时峰值 377ms/分钟 累积耗时均值 97ms/分钟	864ms	734ms
ParNew +CMS，Young区扩大0.5倍	峰值 51次/分钟 均值 48次/分钟	累积耗时 2834ms/分钟 单次耗时 60ms/次/分钟	峰值 1次/分钟 均值 0.279次/分钟	累积耗时峰值 214ms/分钟 累积耗时均值 52ms/分钟	749ms	643ms
ParNew +CMS，Young区不变	峰值 74次/分钟 均值 71次/分钟	累积耗时 3720ms/分钟 单次耗时 53ms/次/分钟	峰值 1次/分钟 均值 0.2次/分钟46	累积耗时峰值 201ms/分钟 累积耗时均值 41ms/分钟	816ms	700ms

可以看出，在高负载场景，4种ParNew + CMS的各项指标表现均远好于Parrallel Scavenge + Parrallel Old。其中：

- 方案4表现最佳：(Young 区扩大 0.5 倍)
    - 接口P95，P99延时相对当前方案降低50%
    - Full GC累积耗时减少88%，
    - Young GC次数减少23%
    - Young GC累积耗时减少4%
    - Young区调大后，虽然次数减少了，但Young区大了，单次Young GC的耗时也大概率会上升，这是符合预期的。
- 方案2和方案3表现接近：
    - 接口P95，P99延时相对当前方案降低40%
    - Full GC累积耗时减少81%
    - Young GC次数减少43%
    - Young GC累积耗时减少17%
    - 略逊于Young区扩大0.5倍，总体表现不错，这两个方案进行合并，不再区分。

Young区不变的方案在新方案里，表现最差，淘汰。所以在中负载场景，我们只需要对比方案2和方案4。

中负载场景(600 QPS)GC表现

方案	Young GC次数峰值/均值	Young GC累积耗时/单次耗时	Full GC次数峰值/均值	Full GC累积耗时/单次耗时	接口P99	接口P95
Parrallel Scavenge + Parrallel Old（当前方案）	峰值 43次/分钟 均值 38次/分钟	累积耗时 1366ms/分钟 单次耗时 35ms/次/分钟	峰值 1次/分钟 均值 0.076次/分钟	累积耗时峰值 526ms/分钟 累积耗时均值 27ms/分钟	567ms	404ms
ParNew +CMS，Young区扩大1倍	峰值 23次/分钟 均值 22次/分钟	累积耗时 769ms/分钟 单次耗时 38ms/次/分钟	峰值 1次/分钟 均值 0.005次/分钟	累积耗时峰值 179ms/分钟 累积耗时均值 2ms/分钟	383ms	209ms
ParNew +CMS，Young区扩大0.5倍	峰值 27次/分钟 均值 26次/分钟	累积耗时 982ms/分钟 单次耗时 39ms/次/分钟	峰值 1次/分钟 均值 0.006次/分钟	累积耗时峰值 244ms/分钟 累积耗时均值 2ms/分钟	476ms	334ms

可以看出，在中负载场景，2种ParNew + CMS(方案2和方案4)的各项指标表现也均远好于Parrallel Scavenge + Parrallel Old。

- Young区扩大1倍的方案表现最佳：
    - 接口P95，P99延时相对当前方案降低32%
    - Full GC累积耗时减少93%
    - Young GC次数减少42%
    - Young GC累积耗时减少44%；
- Young区扩大0.5倍的方案稍逊一些。

综合来看，两个方案表现十分接近，原则上两种方案都可以，只是Young区扩大0.5倍的方案在业务高峰期的表现更佳，为尽量保证高峰期服务的稳定和性能，目前**更倾向于选择ParNew + CMS，Young区扩大0.5倍方案。**

3.2 灰度分析

为保证覆盖业务的高峰期，选择周五、周六、周日分别从两个机房随机选择一台线上实例，线上实例的指标符合预期后，再进行全量升级。

目标组 xx.xxx.60.6：目标方案，方案2

```
-Xms4096M -Xmx4096M -Xmn1536M -XX:MetaspaceSize=256M -
XX:MaxMetaspaceSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -
XX:+CMSScavengeBeforeRemark
```

对照组1 xx.xxx.15.215：原始方案

```
-Xms4096M -Xmx4096M -Xmn1024M -XX:PermSize=512M -XX:MaxPermSize=512M
```

对照组2 xx.xxx.40.87：候选方案，方案4

```
-Xms4096M -Xmx4096M -Xmn2048M -XX:MetaspaceSize=256M -
XX:MaxMetaspaceSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -
XX:+CMSScavengeBeforeRemark
```

灰度3台机器结论：

YoungGC

- 与原始方案相比，**目标方案(方案2)**的YGC次数减少50%，累积耗时减少47%
- 吞吐量提升的同时，服务停顿频率大大降低，而代价是单次Young GC的耗时增长3ms，收益是非常高的。
- 对照方案2即Young区2G的方案整体表现稍逊与目标方案

Full GC

- 与原始方案相比，使用目标方案时，老年代增长的速度要缓慢很多
- 在观测周期内Full GC发生的次数从155次减少至27次，减少82%
- 停顿时间均值从399ms减少至60ms，减少85%，毛刺也非常少
- 对照方案2即Young区2G的方案整体表现逊于目标方案

到这里，可以看出，目标方案从各个维度均远优于原始方案，调优目标也基本达成。

四、结果验收

灰度持续7天左右，覆盖工作日与周末，结果符合预期，因此符合在线上开启全量的条件，下面对全量后的结果进行评估。

从Young GC的指标上看表现达到预期

- 调整后Young GC次数平均减少30%
- Young GC累积耗时平均减少17%
- Young GC单次耗时平均增加约7ms

> **除了技术手段，我们也在业务上做了一些优化**，调优前实例的Young GC会出现明显的、不规律的（定时任务不一定分配到当前实例）毛刺，这里是业务上的一个定时任务，会加载大量数据，调优过程中将该任务进行分片，分摊到多个实例上，进而使Young GC更加平滑。

从Full GC的指标上看，Full GC的频率、停顿极大减少，可以说基本上没有真正意义上的Full GC了。

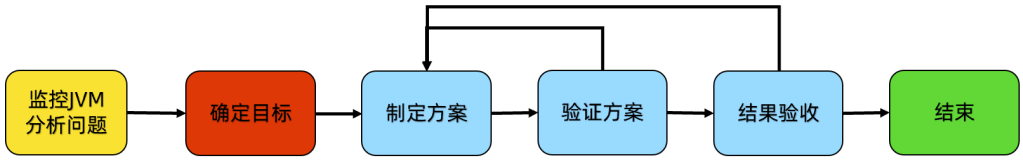
- 核心接口-A (下游依赖较多) P99响应时间，减少19%（从 3457 ms下降至 2817 ms）；
- 核心接口-B (游依赖中等) P99响应时间，减少41%（从 1647ms下降至 973ms）；
- 核心接口-C (下游依赖最少) P99响应时间，减少80%（从 628ms下降至 127ms）；

综合来看，整个结果是超出预期的。Young GC表现与设定的目标非常吻合，基本上没有真正意义上的Full GC，接口P99的优化效果取决于下游依赖的多少，依赖越少，效果越明显。

五、写在最后

由于GC算法复杂，影响GC性能的参数众多，并且具体参数的设置又取决于服务的特点，这些因素都很大程度增加了JVM调优的难度。

本文结合视频服务的调优经验，着重介绍调优的思路和落地过程，同时总结出一些通用的调优流程，希望能给大家提供一些参考。



① **监控JVM分析问题：**评估必要性，内存使用，GC频率，GC耗时

② **确定目标：**内存占用，响应延迟

③ **制定方案：**配置内存及GC相关参数

④ **验证方案：**测试环境对比方案前后差异，确定是否生效

⑤ **结果验收：**灰度测试，全量发布