

Java性能优化与实践

1. 目标与思路

1.1. 运行架构背景

所谓运行架构是指一个系统运作逻辑，以及背后的原理，其关注的是系统的某个局部，或者瞬间。比如：

- 服务注册与发现具体是如何运作？
- 流量控制与服务熔断链路跟踪是如何运作的？
- 一个带索引条件的mysql具体是如何执行的？
- Java程序OOM是具体如何发生的？

面对这些问题，从系统运行的角度出发，需要一层一层的剥开、揉碎，要从算法，内存、协议，定理等低层次的原理进行解释。那么作为T4晋升到T5的架构师为什么要了解这些呢

- 首先，如果不解一些系统运行背后的底层原理，面对复杂或者一些疑难杂症我们将无从下手，失去判断能力；比如：运营商门户升级项目，宁夏云平台性能测试问题；
- 其次，作为一个技术人，面对当前浩如烟海的技术，框架，，要构建自己的技术知识体系，那具体要如何构建呢？根据我们个人的经验，一个比较有效的方法就是，对一个问题或技术要深究其原理，当我们深入研究的问题多了。



1.2. 为什么要讲JVM相关的内容？

[Java程序员技术体系](#)

我们看到市面上新技术、新概念层出不穷，每隔一段时间就会有一个新技术腾空出世替代老技术，这个是我们必须接受的可观规律，王朝更迭，江山易主，世事山河都会变迁，技术与经济、历史一样具有周期性，只是这个周期在软件行业更迭的太快，在程序员的职业生涯里面，大部分的技术可能要迭代更新很多论，“一招鲜，吃遍天”在软件行业可能不灵验，有追求的程序员需要时刻保持着一颗谦卑的心，不断的迎接变化、持续学习，从而让自己超越技术周期；

那有人可能会说，这样会很累，对于这个问题我要看**怎么定义这个累**，每个人的追求与活法都不一样，累与不累，哪些事情你觉得累，哪些事情你觉得开心、快乐，都是我们自己内心的选择。所以，我们认为累也是非常正常的，如果觉得学习很累，那我们可以选择在生活上累一点，如果我们觉得学习、生活上我都不想那么累，并且我都可以做到不累，那，我真的要恭喜你，那你真的是**跳出三界外不在五行中**，你可以洒脱、不受世俗干扰、大彻大悟了。

如果我们下定决心要学习，那具体应该如何学呢？一个最有效的方法就是：**构建自己的技术知识体系**，有一个概念叫**知识负荷**，说的就是

本次的分享主要基于一个案例来了解java，内存模型，垃圾回收，线程池等几个基础问题；

2. 环境说明

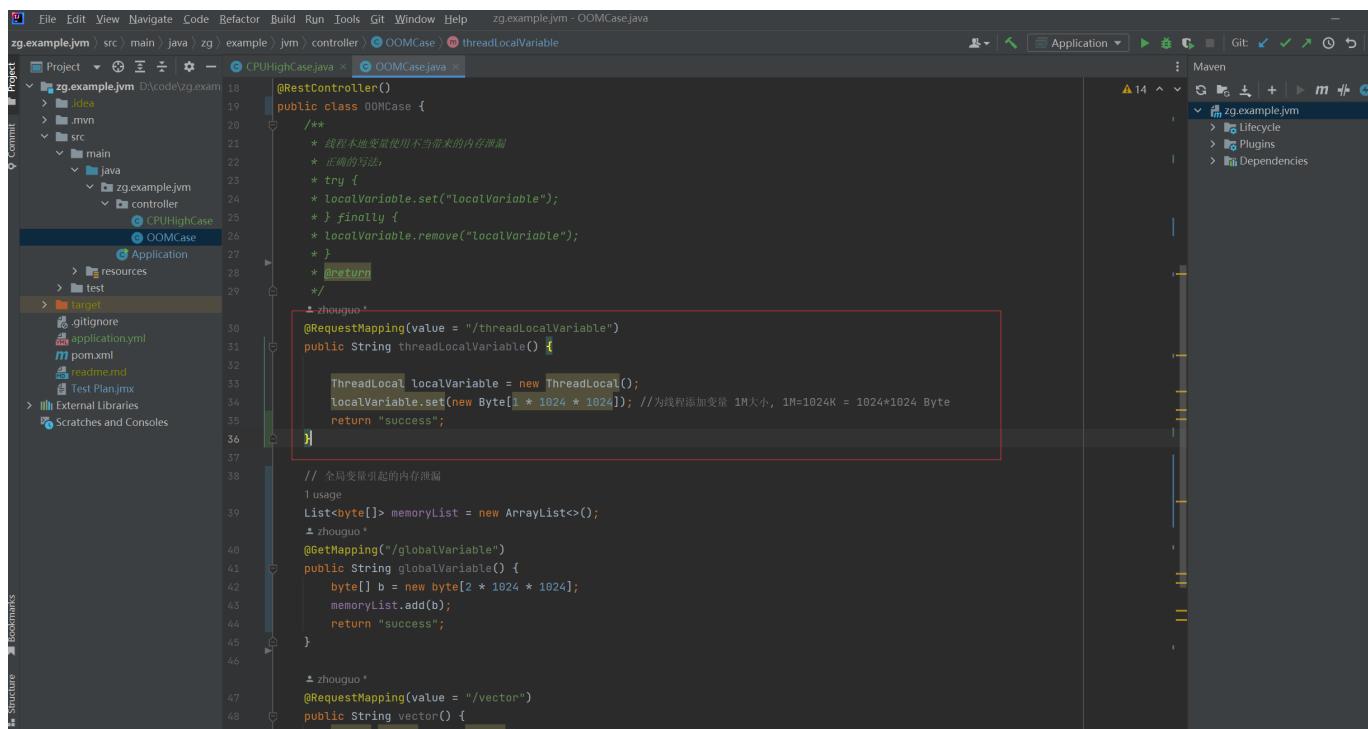
2.1. 开发环境

- IDEA 2022.1.2 (Community Edition)
- JDK-jdk1.8.0_331
- window 11 家庭版

2.2. 运行环境

- CentOS Linux 7 (Core) (虚拟机 4C8G)
- jdk1.8.0_251

2.3. 模拟代码



The screenshot shows the IntelliJ IDEA interface with the project 'zg.example.jvm' open. The code editor displays two files: 'CPUHighCase.java' and 'OOMCase.java'. A red box highlights a specific section of code in 'OOMCase.java'.

```

    public class OOMCase {
        /**
         * 线程本地变量使用不当带来的内存泄漏
         * 正确的写法：
         * try {
         * localVariable.set("localVariable");
         * } finally {
         * localVariable.remove("localVariable");
         * }
         * return;
         */
        @RequestMapping(value = "/threadLocalVariable")
        public String threadLocalVariable() {
            ThreadLocal<byte[]> localVariable = new ThreadLocal();
            localVariable.set(new byte[1 * 1024 * 1024]); //为线程添加变量 1M大小, 1M=1024K = 1024*1024 Byte
            return "success";
        }

        // 全局变量引起的内存泄漏
        1 usage
        List<byte[]> memoryList = new ArrayList<>();
        @GetMapping("/globalVariable")
        public String globalVariable() {
            byte[] b = new byte[2 * 1024 * 1024];
            memoryList.add(b);
            return "success";
        }

        @RequestMapping(value = "/vector")
        public String vector() {
            Vector<Vector> vector = new Vector();
        }
    }

```

```

@RequestMapping(value = "/threadLocalVariable")
public String threadLocalVariable() {

    ThreadLocal localVariable = new ThreadLocal();
    localVariable.set(new Byte[1 * 1024 * 1024]); //为线程添加变量 1M大小, 1M=1024K
    = 1024*1024 Byte
    return "success";
}

```

2.4. 运行参数

- 后台运行

```
nohup java -jar zg.example.jvm-0.0.1-SNAPSHOT.jar &
```

- 简易运行

```
java -jar zg.example.jvm-0.0.1-SNAPSHOT.jar
```

- 普通带参数启动

```
java -Xms1G \ -XX:+HeapDumpOnOutOfMemoryError \ -XX:HeapDumpPath=/opt/heapdump.hprof \ -
XX:+PrintGCTimeStamps \ -XX:+PrintGCDetails \ -Xloggc:/opt/heapTest.log \ -jar
zg.example.jvm-0.0.1-SNAPSHOT.jar
```

```
java -jar -Xms1G -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/heapdump.hprof -
XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:/opt/heapTest.log zg.example.jvm-
0.0.1-SNAPSHOT.jar
```

- 使用JMX远程启动 (VisualVM)

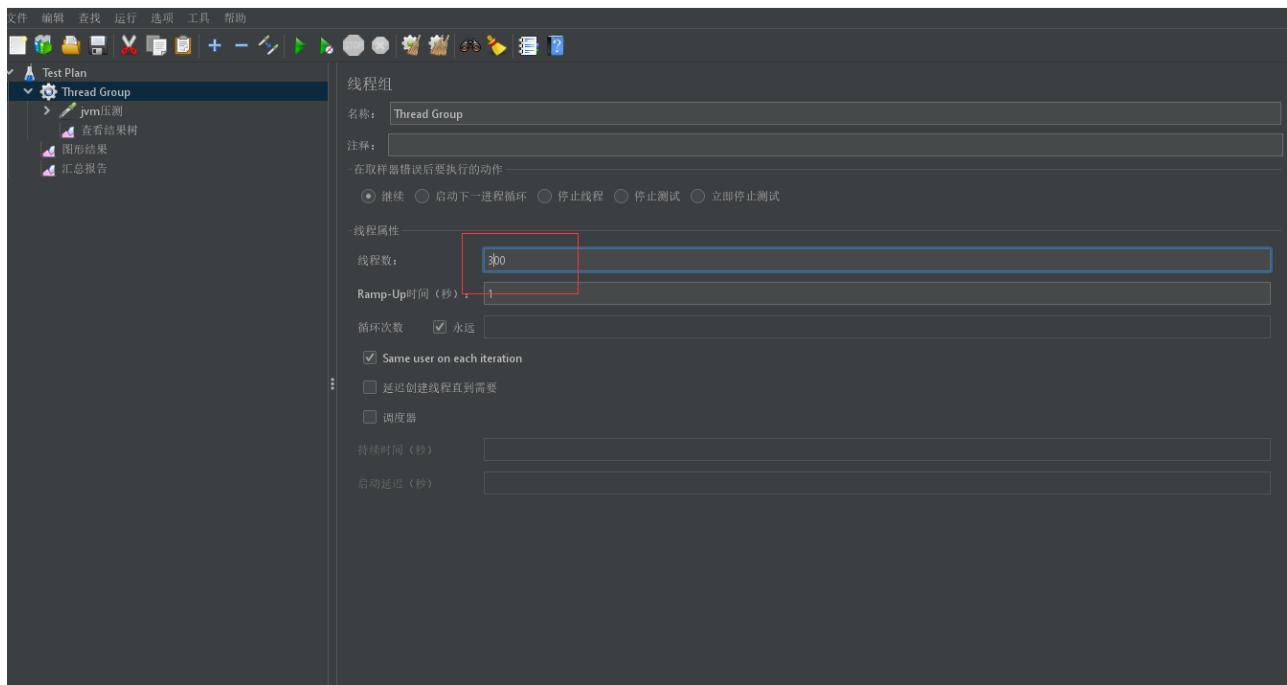
```
java -Djava.rmi.server.hostname=192.168.22.131 \ -
Dcom.sun.management.jmxremote.port=1232 \ -Dcom.sun.management.jmxremote.rmi.port=1240 \ -
-Dcom.sun.management.jmxremote.ssl=false \ -
Dcom.sun.management.jmxremote.authenticate=false \ -Xms1G \ -
XX:+HeapDumpOnOutOfMemoryError \ -XX:HeapDumpPath=/opt/heapdump.hprof \ -
XX:+PrintGCTimeStamps \ -XX:+PrintGCDetails \ -Xloggc:/opt/heapTest.log \ -jar
zg.example.jvm-0.0.1-SNAPSHOT.jar
```

参数说明:

- Xms4G -最大堆内存
- XX:+HeapDumpOnOutOfMemoryError -当内存溢出时触发java.lang.OutOfMemoryError异常
- XX:HeapDumpPath=/tmp/heapdump.hprof -内存溢出时保存快照文件 (可用MAT工具分析)
- XX:+PrintGCTimeStamp -打印GC发生的时间戳
- XX:+PrintGCDetails -Xloggc:/tmp/heapTest.log -打印GC详细日志到文件

2.5. 模拟访问

- 通过Jmeter测试工具，模拟并发访问，模拟300个并发请求；



3. 运行异常(表象)

3.1. 异常监控

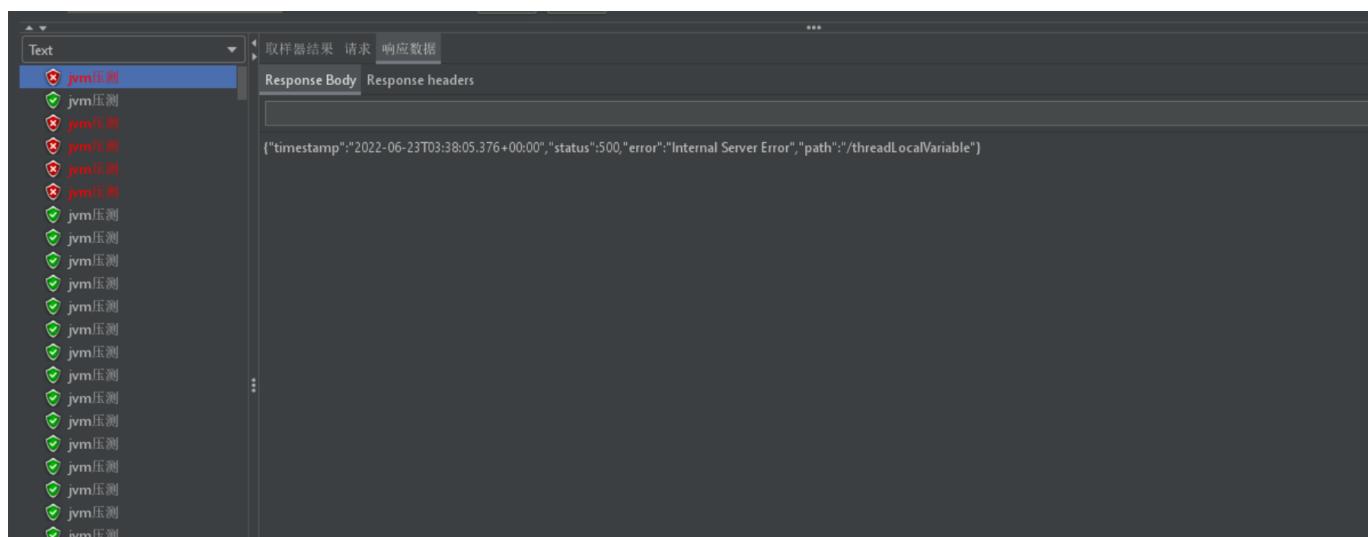
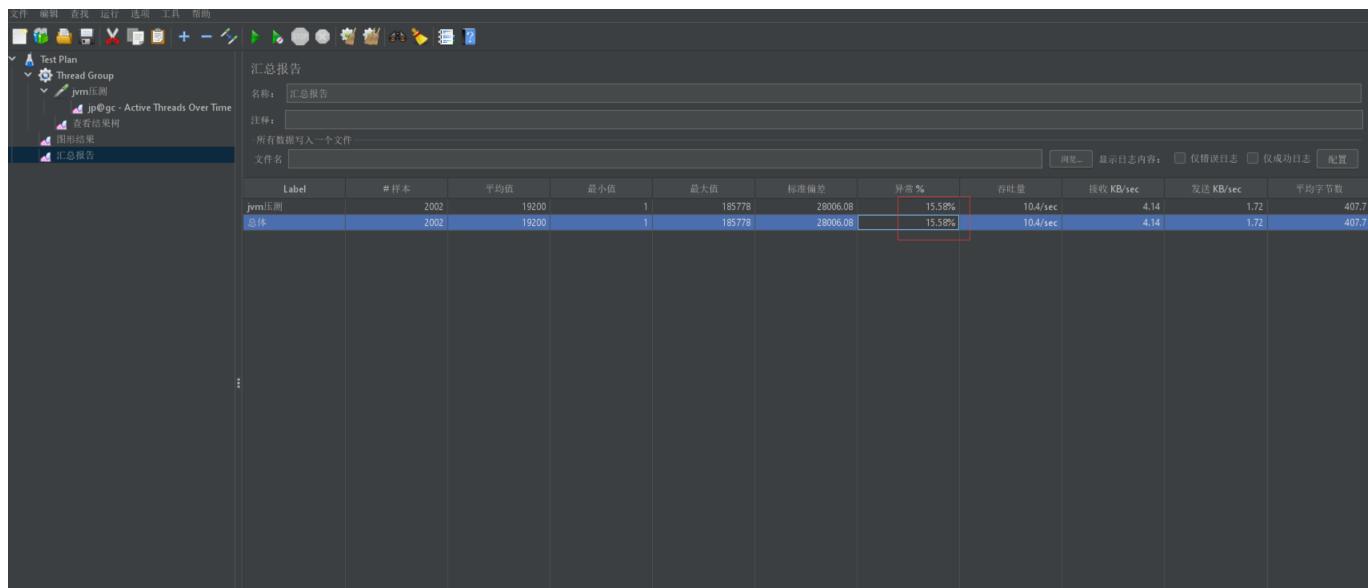
3.2. 控制台OOM异常

```

java.lang.OutOfMemoryError: Java heap space
2022-06-23 11:38:00.714 ERROR 1728 --- [o-8080-exec-176] o.a.c.c.C.[L].I.[].DispatcherServlet : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Handler dispatch failed; nested exception is java.lang.OutOfMemoryError: Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space
2022-06-23 11:37:57.809 ERROR 1728 --- [o-8080-exec-150] o.a.c.c.C.[L].I.[].DispatcherServlet : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Handler dispatch failed; nested exception is java.lang.OutOfMemoryError: Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space
2022-06-23 11:38:02.080 ERROR 1728 --- [io-8080-exec-51] o.a.c.c.C.[L].I.[].DispatcherServlet : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Handler dispatch failed; nested exception is java.lang.OutOfMemoryError: Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space
2022-06-23 11:38:02.080 ERROR 1728 --- [o-8080-exec-107] o.a.c.c.C.[L].I.[].DispatcherServlet : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Handler dispatch failed; nested exception is java.lang.OutOfMemoryError: Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space
2022-06-23 11:38:02.686 ERROR 1728 --- [io-8080-exec-42] o.a.c.c.C.[L].I.[].DispatcherServlet : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Handler dispatch failed; nested exception is java.lang.OutOfMemoryError: Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space

```

3.3. Jmeter异常返回



接口异常

4. 定位分析(里象)

4.1. JVM内存整体情况

4.1.1. jmap工具查看

- Eden Space 域 Old Generation 区内存占用率都达到了99%已无内存可再分配，JVM报OOM异常
- 工具1：jmap命令查看

```
jmap -heap <pid>
```

```
Heap Usage: PS Young Generation Eden Space: capacity = 238551040 (227.5MB) used = 235735744 (224.81512451171875MB) free = 2815296 (2.68487548828125MB) 98.81983495020604% used From Space: capacity = 215482368 (205.5MB) used = 0 (0.0MB) free = 215482368 (205.5MB) 0.0% used To Space: capacity = 238551040 (227.5MB) used = 0 (0.0MB) free = 238551040 (227.5MB) 0.0% used PS Old Generation capacity = 1431830528 (1365.5MB) used = 1430561728 (1364.2899780273438MB) free = 1268800 (1.21002197265625MB) 99.91138616091862% used
```

4.1.2. arthas-dashboard查看

查看整体监控面板

命令： dashboard

ID	NAME	GROUP	PRIORITY	STATE	%CPU	DELTA TIME	TIME	INTERRUPTED	DAEMON
-1	GC Task thread2 (ParallelGC)	-	-1	-	53.24	2.542	0:24.556	false	true
-1	GC Task thread3 (ParallelGC)	-	-1	-	52.04	2.522	0:24.550	false	true
-1	GC Task thread1 (ParallelGC)	-	-1	-	52.49	2.565	0:24.560	false	true
-1	GC Task thread0 (ParallelGC)	-	-1	-	51.01	2.455	0:24.413	false	true
-1	VM Thread	-	-1	-	37.29	1.780	0:19.210	false	true
-1	C2 Compiler Thread	-	-1	-	0.38	0.018	0:3.408	false	true
-1	CI Compiler Thread2	-	-1	-	0.16	0.007	0:1.713	false	true
45	Timer-for-arthas-dashboard-c9e4771-9c8c-41fb-ba70-0d1062dad896	System	5	RUNNABLE	0.08	0.004	0:0.477	false	true
-1	VM Periodic Task Thread	-	-1	-	0.07	0.003	0:0.800	false	true
168	http-nio-8080-exec-133	nain	5	WAITING	0.05	0.002	0:0.005	false	true
25	http-nio-8080-exec-10	nain	5	BLOCKED	0.05	0.002	0:0.017	false	true
108	http-nio-8080-exec-73	nain	5	RUNNABLE	0.05	0.002	0:0.008	false	true
85	http-nio-8080-exec-50	nain	5	BLOCKED	0.04	0.002	0:0.010	false	true
82	http-nio-8080-exec-47	nain	5	RUNNABLE	0.04	0.001	0:0.006	false	true
171	http-nio-8080-exec-136	nain	5	RUNNABLE	0.03	0.001	0:0.013	false	true
174	http-nio-8080-exec-139	nain	5	RUNNABLE	0.03	0.001	0:0.015	false	true
144	http-nio-8080-exec-109	nain	5	RUNNABLE	0.03	0.001	0:0.008	false	true
Memory									
heap	used	total	max	usage		gc			
ps_eden_space	1518M	1570M	1759M	87.51%	39.96%	gc_ps_scavenge.count	20		
ps_survivor_space	220M	223M	551M	47104K	47104K	gc_ps_scavenge.time(ms)	817		
ps_old_gen	OK	47104K	47104K	0.0%	0.002	gc_ps_marksweep.count	135		
nonheap	128M	130M	130M	99.75%	99.75%	gc_ps_marksweep.time(ms)	47110		
code_cache	63M	66M	-1	95.00%					
metaspace	13M	14M	240M	5.80%					
compressed_class_space	43M	46M	-1	94.27%					
direct	5M	5M	1024M	0.53%					
mapped	1M	1M	-	100.00%					
mapped	0M	0M	-	0.00%					
Runtime									
os.name				Linux					
os.version				3.10.0-1160.el7.x86_64					
java.version				1.8.0_251					
java.home				/opt/jdk1.8.0_251/jre					
systemLoad.average				1.63					

4.1.3. arthas-jvm查看

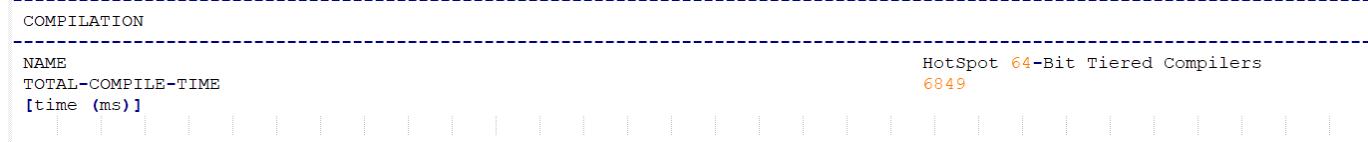
命令： jvm

jvm能查看的重点信息

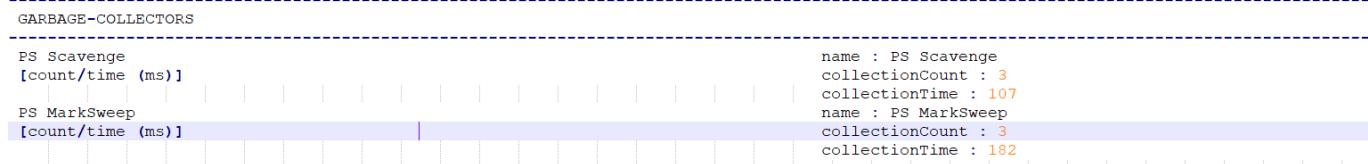
1、类加载信息



2、编译信息



3、垃圾回收情况



4、内存管理与内存

参数说明：

- HEAP-MEMORY-USAGE：堆内存初始化大小

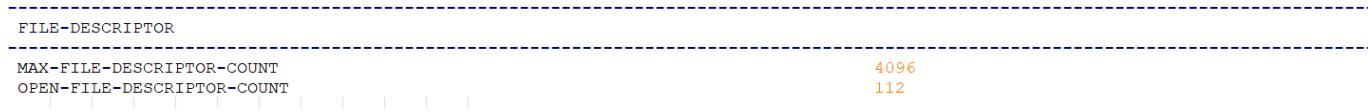
5、线性信息



参数说明：

- COUNT: JVM当前活跃的线程数
- DAEMON-COUNT: JVM当前活跃的守护线程数
- PEAK-COUNT: 从JVM启动开始曾经活着的最大线程数
- STARTED-COUNT: 从JVM启动开始总共启动过的线程次数
- DEADLOCK-COUNT: JVM当前死锁的线程数

5、文件描述符相关



参数说明：

- MAX-FILE-DESCRIPTOR-COUNT: JVM进程最大可以打开的文件描述符数
- OPEN-FILE-DESCRIPTOR-COUNT: JVM当前打开的文件描述符数

4.2. 堆内存对象

首先，可以使用jmap命令查看，堆内存的使用量；

`jmap -histo:live <pid>`

使用jmap命令，查看具体的堆内对象，live参数只显示存活对象；

num	#instances	#bytes	class name
<hr/>			
1:	101	360712896	[Ljava.lang.Byte;
2:	46350	8553520	[C
3:	6808	5807480	[B
4:	45962	1103088	java.lang.String
5:	8789	973480	java.lang.Class
6:	8093	712184	java.lang.reflect.Method
7:	7799	689136	[Ljava.lang.Object;
8:	19532	625024	java.util.concurrent.ConcurrentHashMap\$Node
9:	4924	541408	[I
10:	9943	397720	java.util.LinkedHashMap\$Entry
11:	4350	359160	[Ljava.util.HashMap\$Node;
12:	9008	288256	java.util.HashMap\$Node
13:	10018	225960	[Ljava.lang.Class;
14:	3892	217952	java.util.LinkedHashMap
15:	13594	217504	java.lang.Object
16:	143	199344	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
17:	1348	107840	java.lang.reflect.Constructor
18:	38	99136	[Ljava.nio.channels.SelectionKey;
19:	2048	98304	java.util.HashMap
20:	1276	91872	java.lang.reflect.Field

4.3. 堆文件信息分析

第一、生成堆文件，有很多种方法生成堆文件

- 1、java程序启动时指定OOM异常时输出hprof文件

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/heapdump.hprof
```

- 2、使用jconsole或VisualVM等工具在运行时获得堆转储生成hprof文件
- 3、是要jmap dump:live命令动态生成hprof文件

```
jmap -dump:live,file=/opt/a.hprof pid
```

- 4、使用arthas的heapdump命令生成hprof文件

```
heapdump /opt/dump2.hprof
```

第二、分析对文件信息，也有多种工具

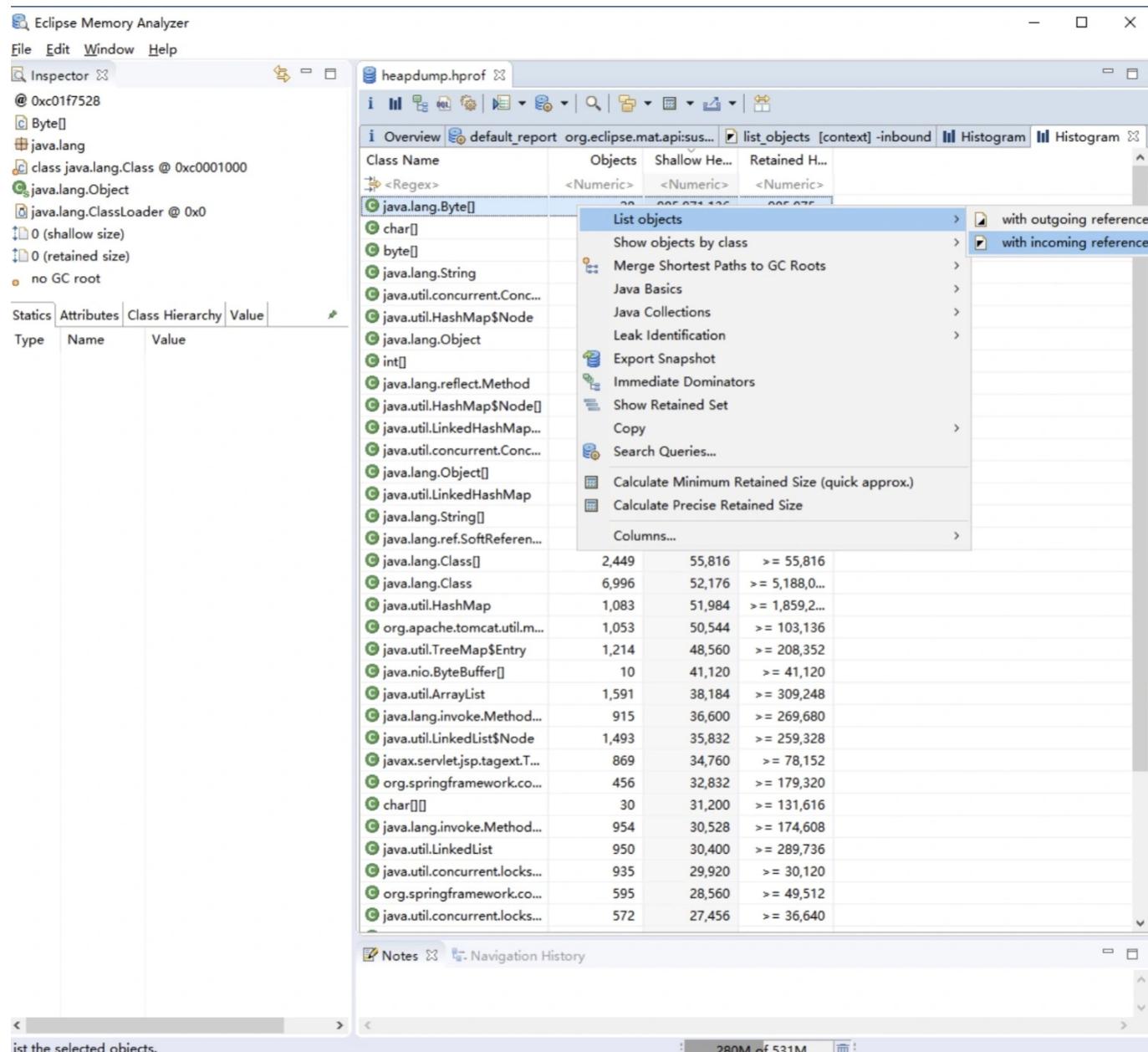
- 1、jhat工具转化为HTML文件进行查看

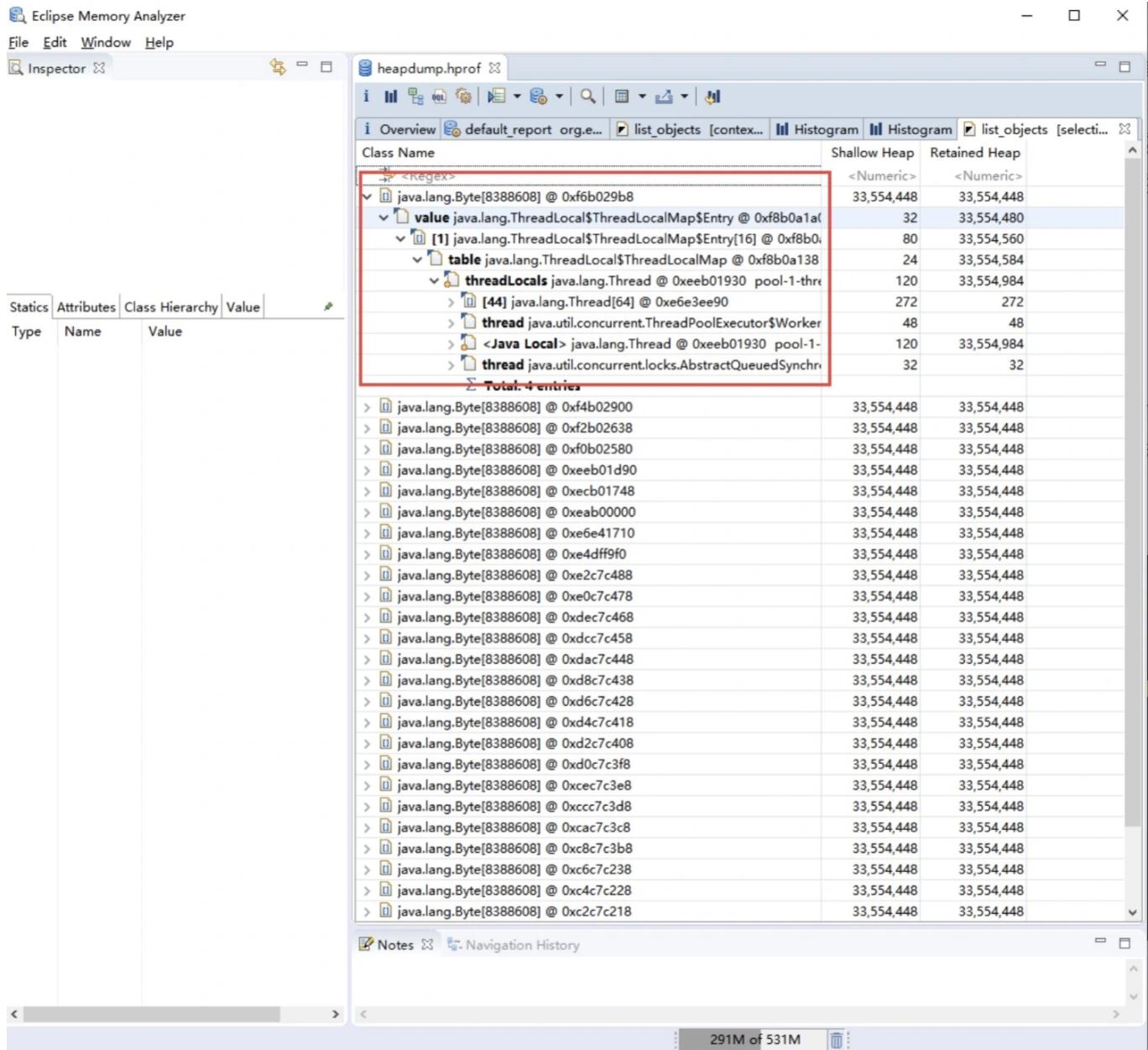
```
jhat dump.hprof
```

```
jhat -J-Xmx512m <heap dump file> --如果堆文件很大可使用参数-J-Xmx进行设置
```

```
jhat -port 8000 -J-Xmx512m dump.hprof --可指定端口号
```

- 2、使用MIT工具进行可视化查看





4.4. 垃圾回收日志

Full GC变得非常频繁，并且

```
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
47.581: [GC (Metadata GC Threshold) [PSYoungGen: 199429K->5984K(611840K)] 199429K->6008K(2010112K), 0.0687019 secs] [Times: user=0.01 sys=0.17, real=0.07 secs]
47.650: [Full GC (Metadata GC Threshold) [PSYoungGen: 5984K->0K(611840K)] [ParOldGen: 24K->5830K(1398272K)] 6008K->5830K(2010112K), [Metaspace: 20114K->20114K(1067008K)], 0.0549951 secs] [Times: user=0.02 sys=0.11, real=0.05 secs]
9788.578: [GC (Allocation Failure) [PSYoungGen: 524800K->14963K(611840K)] 530630K->20801K(2010112K), 0.0768747 secs] [Times: user=0.00 sys=0.29, real=0.08 secs]
9789.355: [GC (Allocation Failure) [PSYoungGen: 537836K->87030K(611840K)] 543675K->302014K(2010112K), 0.6647330 secs] [Times: user=0.00 sys=2.62, real=0.67 secs]

9790.028: [GC (Metadata GC Threshold) [PSYoungGen: 195994K->87014K(611840K)] 410977K->310221K(2010112K), 0.1645180 secs] [Times: user=0.00 sys=0.62, real=0.16 secs]
9790.193: [Full GC (Metadata GC Threshold) [PSYoungGen: 87014K->0K(611840K)] [ParOldGen: 223207K->306970K(1398272K)] 310221K->306970K(2010112K), [Metaspace: 32742K->32454K(1079296K), 0.2750932 secs] [Times: user=0.00 sys=0.92, real=0.28 secs]
9790.654: [GC (Allocation Failure) [PSYoungGen: 522907K->84208K(609280K)] 829878K->722956K(2007552K), 0.8502071 secs] [Times: user=0.00 sys=2.90, real=0.85 secs]

9791.598: [GC (Allocation Failure) [PSYoungGen: 609000K->88816K(368640K)] 1247756K->1165838K(1766912K), 1.0959252 secs] [Times: user=0.00 sys=3.99, real=1.09 secs]
9792.694: [Full GC (Ergonomics) [PSYoungGen: 88816K->0K(368640K)] [ParOldGen: 1077021K->882450K(1398272K)] 1165838K->882450K(1766912K), [Metaspace: 32765K->32672K(1081344K), 0.2915002 secs] [Times: user=0.70 sys=0.08, real=0.30 secs]
9793.032: [GC (Allocation Failure) [PSYoungGen: 279040K->209056K(488448K)] 1161490K->1148851K(1886720K), 0.0622915 secs] [Times: user=0.23 sys=0.00, real=0.06 secs]
9793.094: [Full GC (Ergonomics) [PSYoungGen: 209056K->0K(488448K)] [ParOldGen: 939794K->1091385K(1398272K)] 1148851K->1091385K(1886720K), [Metaspace: 32708K->32708K(1081344K), 0.3830855 secs] [Times: user=0.99 sys=0.00, real=0.39 secs]
9793.521: [GC (Allocation Failure) [PSYoungGen: 278442K->209024K(443392K)] 1369827K->1357754K(1841664K), 0.1464605 secs] [Times: user=0.50 sys=0.05, real=0.15 secs]
9793.668: [Full GC (Ergonomics) [PSYoungGen: 209024K->0K(443392K)] [ParOldGen: 1148729K->1275672K(1398272K)] 1357754K->1275672K(1841664K), [Metaspace: 32766K->32766K(1081344K), 0.5070025 secs] [Times: user=0.76 sys=0.81, real=0.50 secs]

9794.210: [Full GC (Ergonomics) [PSYoungGen: 230473K->0K(443392K)] [ParOldGen: 1275672K->1390387K(1398272K)] 1506146K->1390387K(1841664K), [Metaspace: 33048K->33048K(1081344K), 0.6101545 secs] [Times: user=1.30 sys=0.55, real=0.61 secs]
9794.852: [Full GC (Ergonomics) [PSYoungGen: 232960K->132077K(443392K)] [ParOldGen: 1390387K->1394823K(1398272K)] 1623347K->1526901K(1841664K), [Metaspace: 33101K->33101K(1081344K), 0.4821186 secs] [Times: user=1.18 sys=0.00, real=0.48 secs]
9795.355: [Full GC (Ergonomics) [PSYoungGen: 231572K->173472K(443392K)] [ParOldGen: 1394823K->1394816K(1398272K)] 1626396K->1568289K(1841664K), [Metaspace: 33209K->33209K(1081344K), 0.4125826 secs] [Times: user=1.08 sys=0.00, real=0.41 secs]
```

使用在线工具gceasy分析GC日志

5. 原理解析

5.1. JVM 内存分布

JVM内存模型

JVM专题2

使用查看初始JVM情况

```
jmap -heap 1728
```

1. Attaching to process ID 1728, please wait... Debugger attached successfully. Server compiler detected. JVM version is 25.251-b08
2. using thread-local object allocation. Parallel GC with 4 thread(s)
3. Heap Configuration: MinHeapFreeRatio = 0 MaxHeapFreeRatio = 100 MaxHeapSize = 2147483648 (2048.0MB) NewSize = 715653120 (682.5MB) MaxNewSize = 715653120 (682.5MB) OldSize = 1431830528 (1365.5MB) NewRatio = 2 SurvivorRatio = 8 MetaspaceSize = 21807104 (20.796875MB) CompressedClassSpaceSize = 1073741824 (1024.0MB) MaxMetaspaceSize = 17592186044415 MB G1HeapRegionSize = 0 (0.0MB)
4. Heap Usage: PS Young Generation Eden Space: capacity = 537395200 (512.5MB) used = 177346208 (169.13052368164062MB) free = 360048992 (343.3694763183594MB) 33.001077791539636% used From Space: capacity = 89128960 (85.0MB) used = 0 (0.0MB) free = 89128960 (85.0MB) 0.0% used To Space: capacity = 89128960 (85.0MB) used = 0 (0.0MB) free = 89128960 (85.0MB) 0.0% used PS Old Generation capacity = 1431830528 (1365.5MB) used = 5970864 (5.6942596435546875MB) free = 1425859664 (1359.8057403564453MB) 0.41700912805233886% used
5. 13484 interned Strings occupying 1182024 bytes.

参数说明

第一部分：JVM版本信息

Attaching to process ID 1728, please wait... Debugger attached successfully. Server compiler detected. JVM version is 25.251-b08

第二部分：垃圾收集器信息

using thread-local object allocation. Parallel GC with 4 thread(s)

Parallel GC 是JDK8默认 (-XX:+UseParallelGC) 的垃圾回收算法，Parallel GC属于并行垃圾回收算法.... TODO

第三部分：堆配置信息

Heap Configuration: MinHeapFreeRatio = 0 MaxHeapFreeRatio = 100 MaxHeapSize = 2147483648 (2048.0MB) NewSize = 715653120 (682.5MB) MaxNewSize = 715653120 (682.5MB) OldSize = 1431830528 (1365.5MB) NewRatio = 2 SurvivorRatio = 8 MetaspaceSize = 21807104 (20.796875MB) CompressedClassSpaceSize = 1073741824 (1024.0MB) MaxMetaspaceSize = 17592186044415 MB G1HeapRegionSize = 0 (0.0MB)

说明:

- MinHeapFreeRatio **堆空间最小百分比** 计算公式为： $\text{HeapFreeRatio} = (\text{CurrentFreeHeapSize}/\text{CurrentTotalHeapSize}) * 100.. \text{TODO}$
- MaxHeapFreeRatio **堆空间最大百分比**
- MaxHeapSize **最大堆内存**：最大堆内存默认为物理内存的1/4;
- NewSize **新生代空间默认大小**：新生代默认大小为堆内存的1/3 $=(2048*1/3=682.5\text{MB})$
- MaxNewSize **新生代空间最大大小**
- OldSize **老年代空间大小**：老年代默认大小为堆内存的2/3 $=(2048*2/3=1365.5\text{MB})$
- NewRatio **新生代中New区（伊甸园区）大小**：默认为整个新生代的80% $=(682.5*0.8=)$
- SurvivorRatio **新世代中Survivor区大小**：默认为整个新生代的80%
- MetaspaceSize **元空间初始大小**：存储jvm中的元数据，包括byte code, class等，
- CompressedClassSpaceSize ?? **TODO**
- MaxMetaspaceSize **元空间最大大小**：默认是没有限制的，会一直增长直到整个机器内存吃满，被操作系统Kill
- G1HeapRegionSize

第四部：分堆使用情况

```
Heap Usage: PS Young Generation Eden Space: capacity = 537395200 (512.5MB) used = 177346208 (169.13052368164062MB) free = 360048992 (343.3694763183594MB)
33.001077791539636% used From Space: capacity = 89128960 (85.0MB) used = 0 (0.0MB) free = 89128960 (85.0MB) 0.0% used To Space: capacity = 89128960 (85.0MB) used = 0 (0.0MB) free = 89128960 (85.0MB) 0.0% used PS Old Generation capacity = 1431830528 (1365.5MB) used = 5970864 (5.6942596435546875MB) free = 1425859664 (1359.8057403564453MB)
0.41700912805233886% used
```

说明：

该部分主要展示堆内存的实时使用情况，分别展示对堆内存4个区域中的，容量，使用量，空闲量，使用比例；

5.2. GC垃圾回收机制

- 垃圾回收算法 [图解Java垃圾回收算法及详细过程！_Java_攀岩鱼_InfoQ写作社区](#)

5.3. 线程本地变量机制

- 线程本地变量 [线程本地变量执行原型](#)

5.4. 线程池模型

[线程池运行原理-Tomact 线程池运行原理AQS-JDK](#)

6. 总结

【金山文档】JVM问题定位与分析 <https://kdocs.cn/l/clfkP4FQwvIU>

7. 参考资料

- Oracle 标准的HotSpot虚拟机垃圾回收调优指南 Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 8 (oracle.com)
- JVM参数优化 [7 JVM arguments of Highly Effective Applications – GC easy – Universal Java GC Log Analyser](#)
- 极客时间-Java性能调优课程
- 周志明《Java虚拟机第三版》
- 周志明《凤凰架构》 [凤凰架构：构筑可靠的大型分布式系统 | 凤凰架构 \(icyfenix.cn\)](#)

8. 后续改进

- 需要有一个整体的运行架构总图作为导入，告诉读者为什么要先讲JVM相关内容，后续的内容是如何规划的；
- 最好先介绍原理后介绍案例，再结合原理与案例进行系统总结；
- 所有介绍的内容的文档、代码，图形需要规范化的整理与同步输出，要给读者开箱即用的便捷性；
- 视频录制的方式最好以小微课的形式呈现，不在意多而在于精；