JVM原理与最佳实践

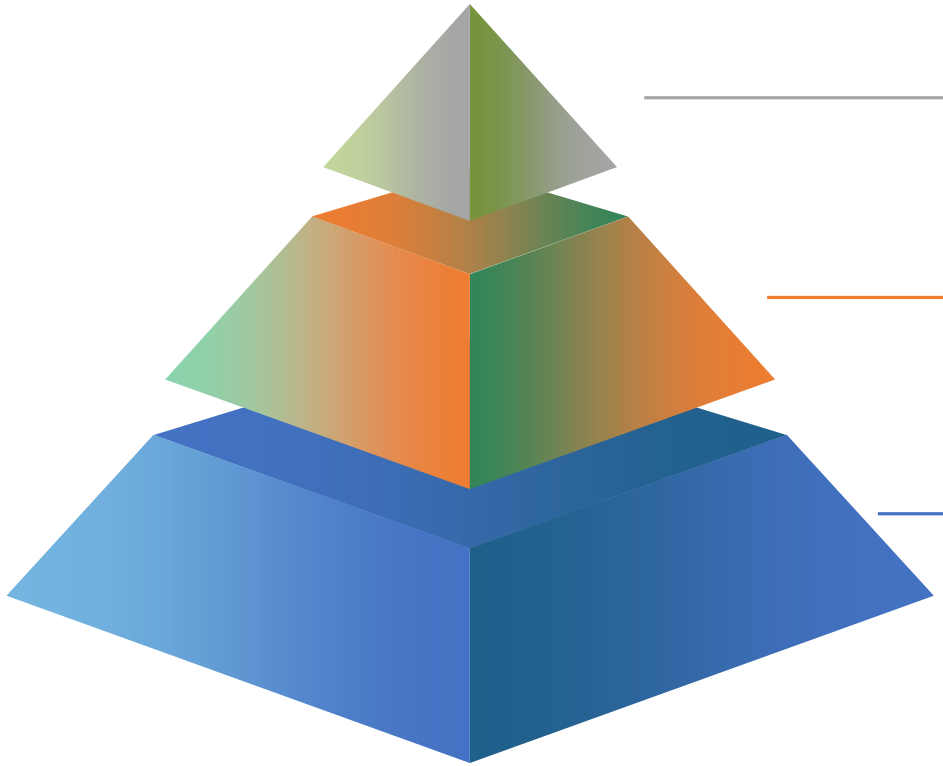
By 刘佳

1


一、JVM简介
二、内存管理
三、垃圾回收
四、运行时优化
五、类加载机制

一、JVM简介

什么是JVM?



JVM的本质

JVM本质是一个应用程序，一个进程

Sun HotSpot VM、J9 VM

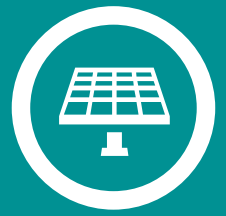
Sun JDK、OpenJDK使用

HotSpot：热点探测技术

CPU vs PSU

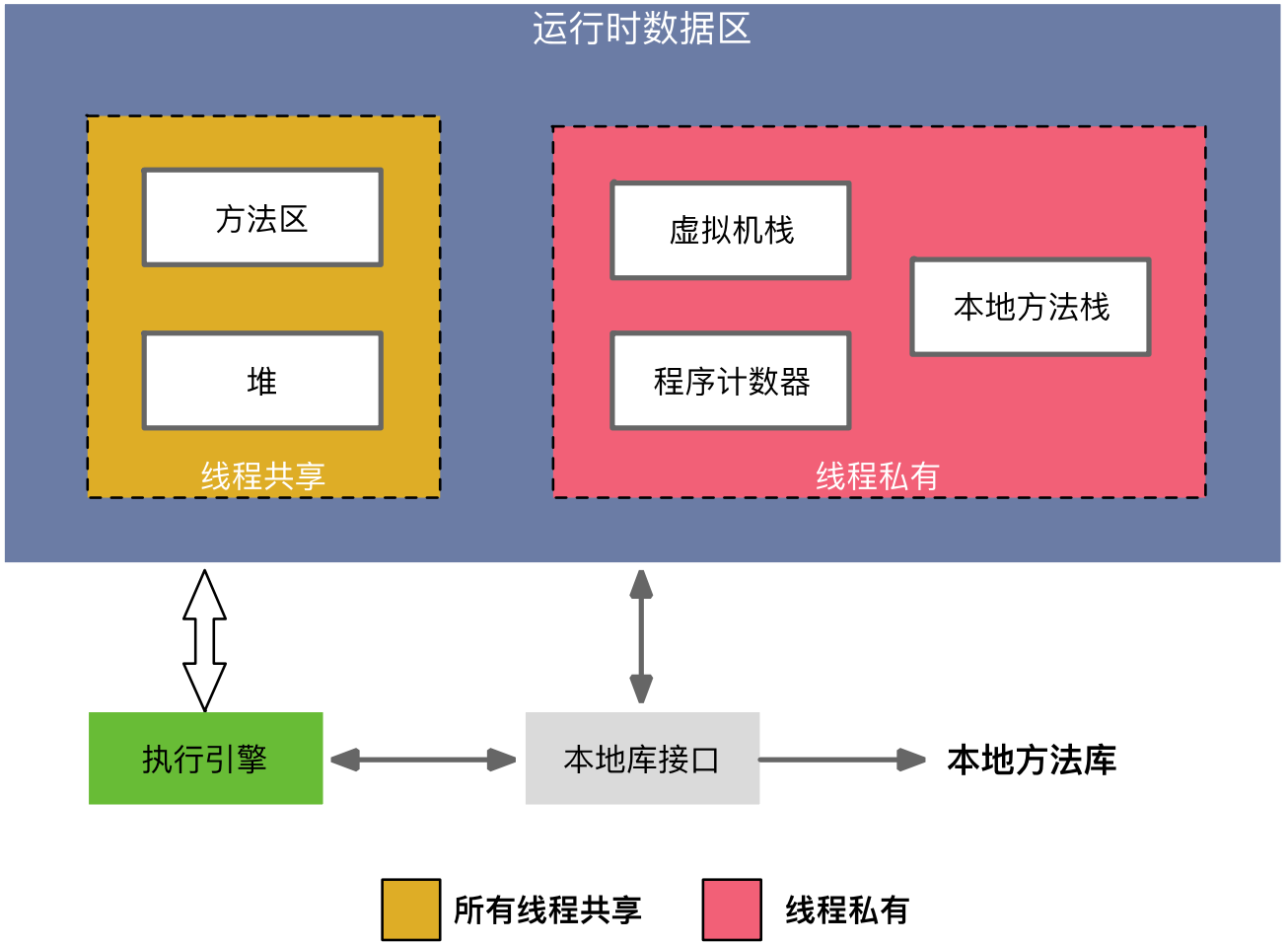
CPU： Critical Patch Updates

PSU： Patch Set Updates



二、自动内存管理

运行时数据区


运行时数据区
方法区
堆
线程共享
虚拟机栈
本地方法栈
程序计数器
线程私有
执行引擎
本地库接口
本地方法库
所有线程共享
线程私有

内存占用： init / used / committed / max

init

表示JVM在启动时从操作系统申请的初始内存量

used

表示JVM当前正在使用的内存量

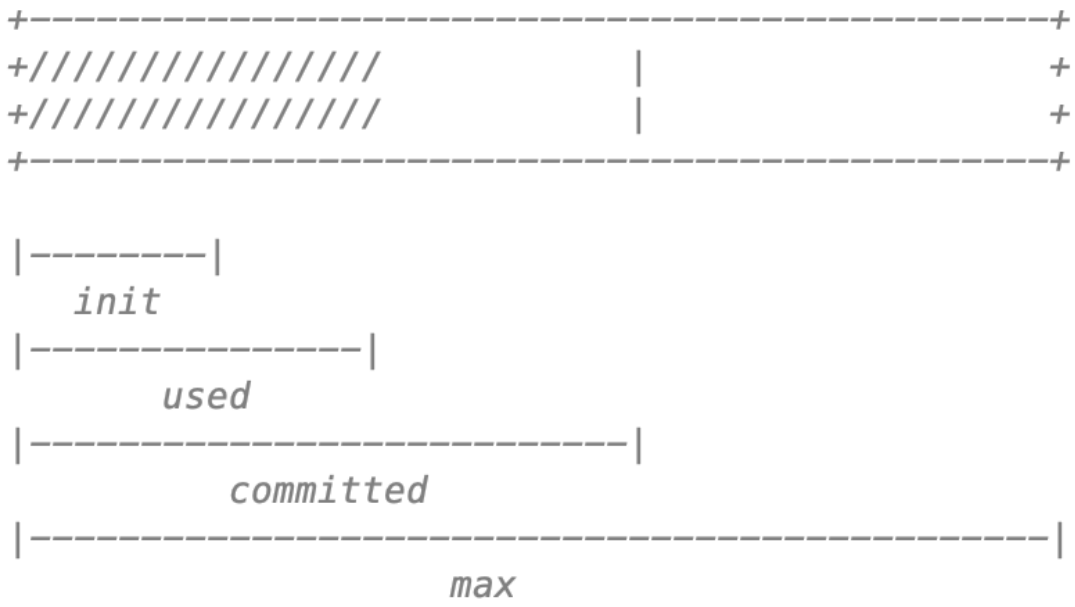
committed

表示JVM保证可以使用的内存量

Committed = used + swapped

max

表示JVM的最大内存



进程实际占用的物理内存（RES）

*127 * 8m = 0.99g*

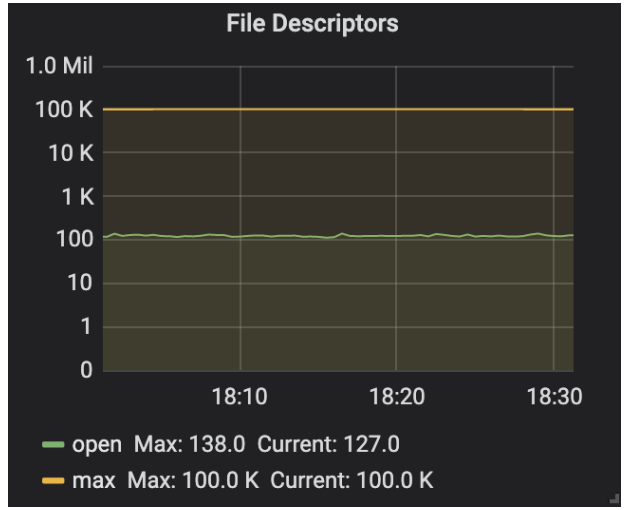
RES: Resident Memory Size, The non-swapped physical memory a task is using

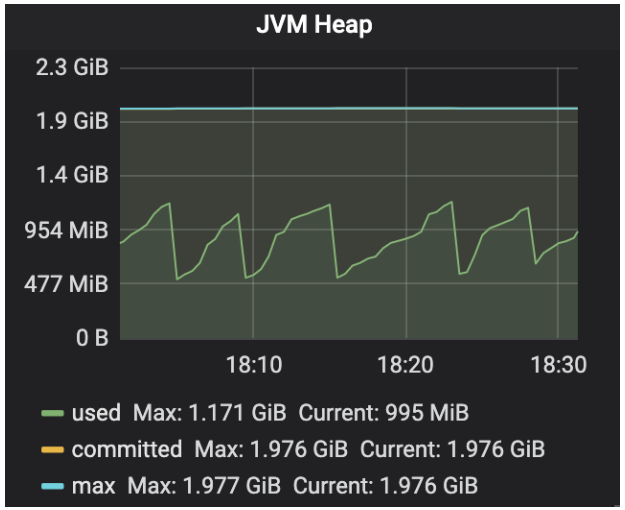
RES = 堆内存（堆used）+ 方法区内存（非堆used）+ 线程栈内存（1M*n）

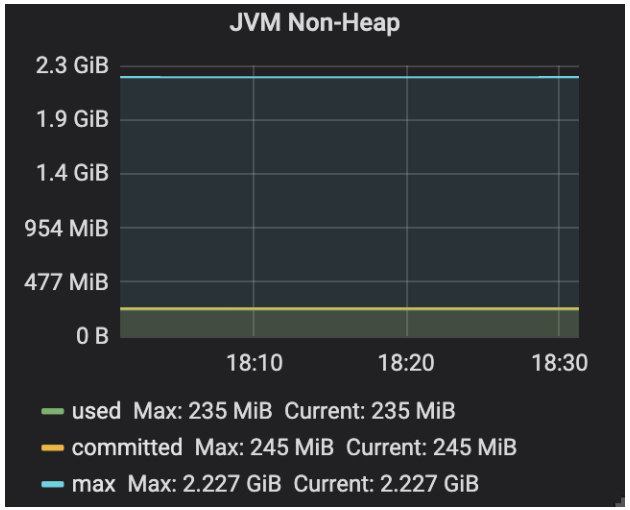
+ 直接内存（堆外used）+ socket buffer内存 + JVM进程内存 + SHR（进程间共享内存）

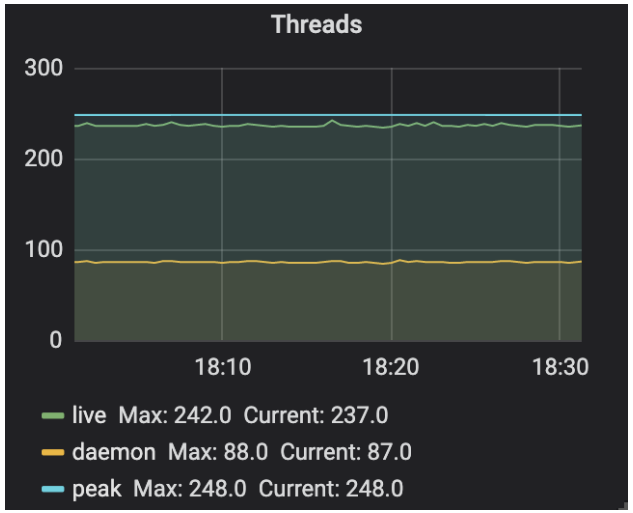
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27784	tomcat	20	0	6135428	2.7g	8204	S	2.3	35.4	128:07.64	java

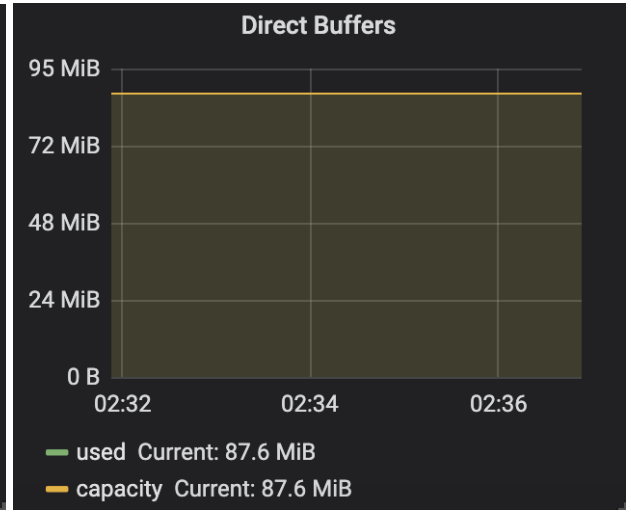
0.97g(堆) + 0.23g(非堆) + 0.32g(线程栈) + 0.09g(堆外) + 0.99g(socket buffer) = **2.6g**











995m = 0.97g

235m = 0.23g

*(237+87) * 1m = 0.32g*

87.6m = 0.09g

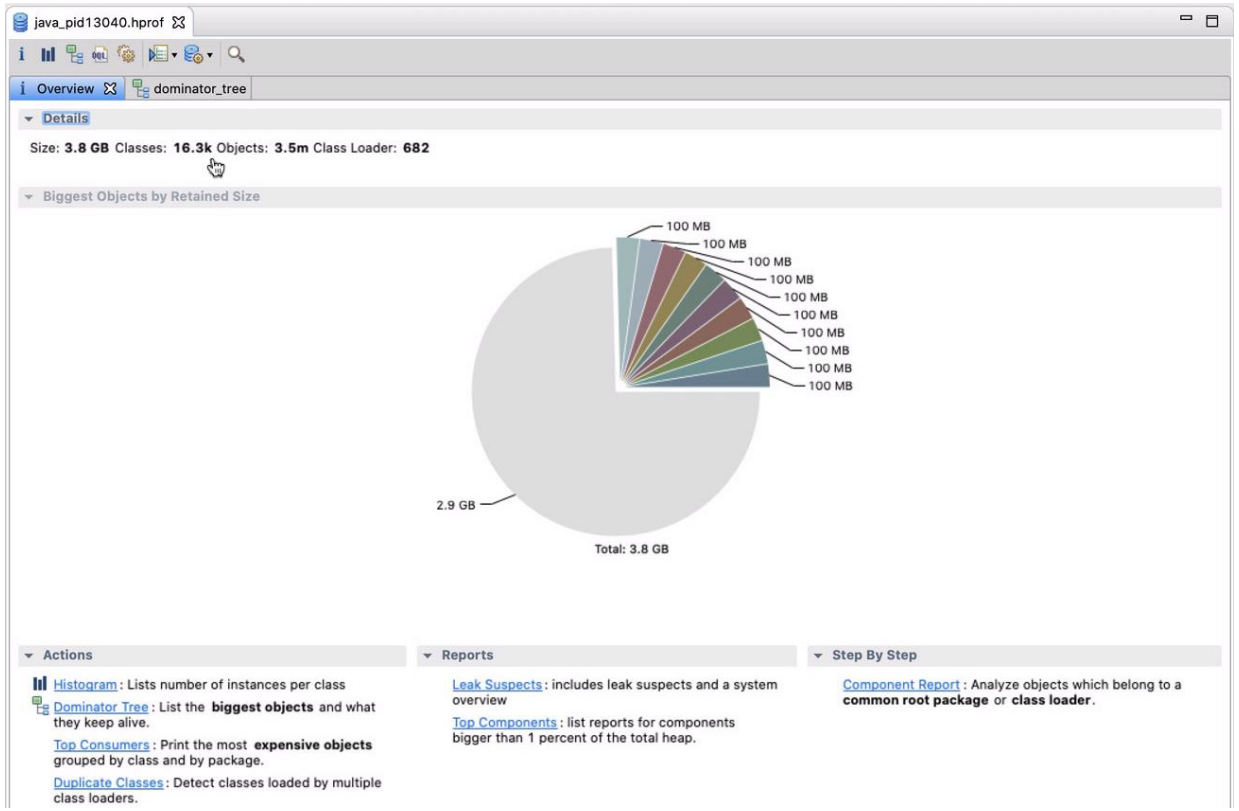
MAT（Memory Analyzer Tool）

Memory Analyzer 1.9.2

The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps you find memory leaks and reduce memory consumption. Use the Memory Analyzer... **more info**

by Eclipse.org, EPL

mat memory heap analyzer leaks ...



MAT - 浅堆和深堆

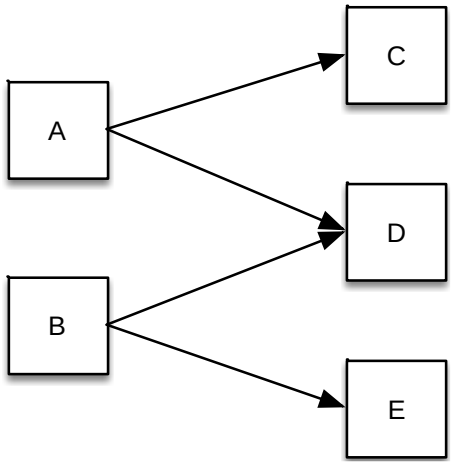
浅堆（Shallow Heap）

- 表示一个对象的数据结构所占用的内存大小
- 浅堆的大小只与对象的结构有关，与对象的实际内容无关
- 浅堆指对象本身占用的内存，不包括其内部引用对象的大小
- 浅堆大小 = 对象头（MarkWord + 类型指针 + 数组长度）

+ 实例数据（基本类型+引用类型） + 对齐填充

深堆（Retained Heap）

- 表示一个对象被GC回收后，可以真实释放的内存大小
- 深堆大小 = 只能通过该对象直接或间接访问到的所有对象的浅堆之和



A的浅堆大小：A

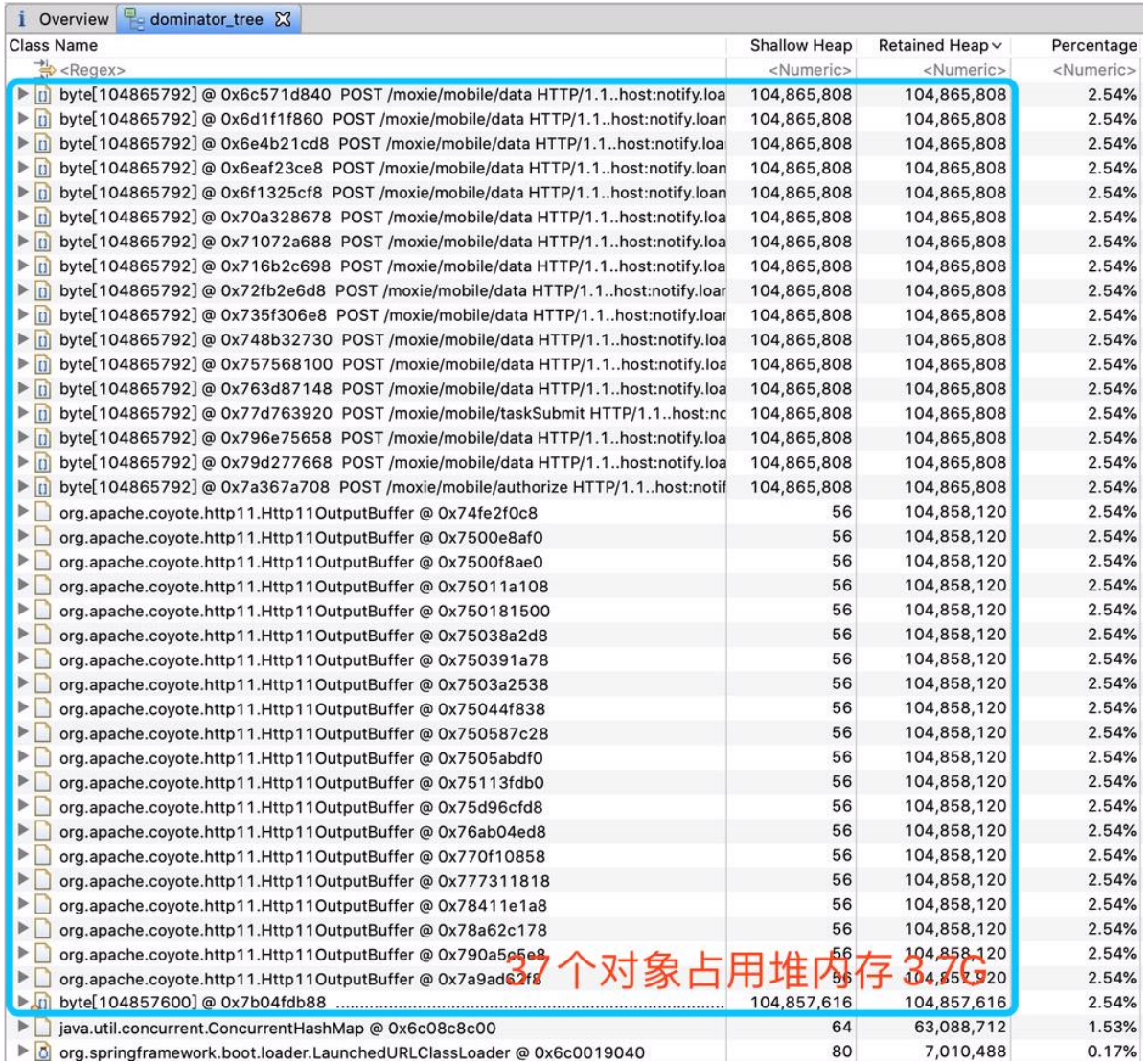
A的实际大小：A + C + D

A的深堆大小：A + C

案例：互金Gateway服务流量高峰堆内存频繁OOM

- **排查步骤**

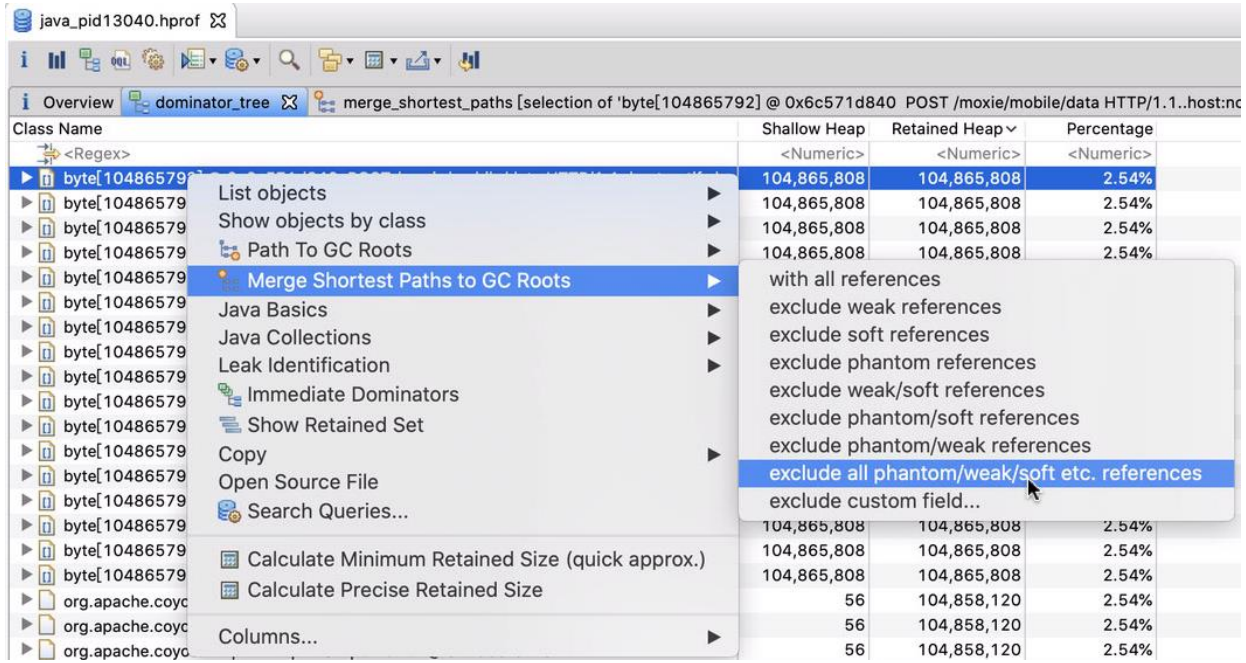
1、使用MAT查看支配树

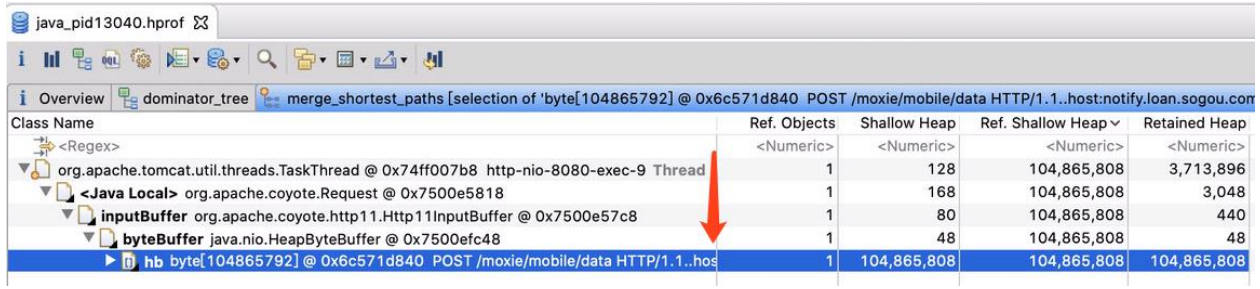


Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[104865792] @ 0x6c571d840 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x6d1f1f860 POST /moxie/mobile/data HTTP/1.1..host:notify.loan	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x6e4b21cd8 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x6eaf23ce8 POST /moxie/mobile/data HTTP/1.1..host:notify.loan	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x6f1325cf8 POST /moxie/mobile/data HTTP/1.1..host:notify.loan	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x70a328678 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x71072a688 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x716b2c698 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x72fb2e6d8 POST /moxie/mobile/data HTTP/1.1..host:notify.loan	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x735f306e8 POST /moxie/mobile/data HTTP/1.1..host:notify.loan	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x748b32730 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x757568100 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x763d87148 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x77d763920 POST /moxie/mobile/taskSubmit HTTP/1.1..host:nc	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x796e75658 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x79d277668 POST /moxie/mobile/data HTTP/1.1..host:notify.loa	104,865,808	104,865,808	2.54%
byte[104865792] @ 0x7a367a708 POST /moxie/mobile/authorize HTTP/1.1..host:notif	104,865,808	104,865,808	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x74fe2f0c8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x7500e8af0	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x7500f8ae0	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x75011a108	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x750181500	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x75038a2d8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x750391a78	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x7503a2538	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x75044f838	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x750587c28	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x7505abdf0	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x75113fdb0	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x75d96cfd8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x76ab04ed8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x770f10858	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x777311818	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x78411e1a8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x78a62c178	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x790a5c5e8	56	104,858,120	2.54%
org.apache.coyote.http11.Http11OutputBuffer @ 0x7a9ad62f8	56	104,858,120	2.54%
byte[104857600] @ 0x7b04fdb88	104,857,616	104,857,616	2.54%
java.util.concurrent.ConcurrentHashMap @ 0x6c08c8c00	64	63,088,712	1.53%
org.springframework.boot.loader.LaunchedURLClassLoader @ 0x6c0019040	80	7,010,488	0.17%

案例：互金Gateway服务流量高峰堆内存频繁OOM

2、查看对象到GC Roots的引用链





- **RootCause**

误把**server.max-http-header-size**设置为100MB，默认8k

- **解决方案**

将此参数调小为**200KB**后，重启服务后恢复正常

案例：互金Gateway服务直接内存OOM触发oom-killer

Gateway直接内存OOM排查及解决方案

- **RootCause**

- Hbase 1.x客户端存在缓慢内存泄露问题
- 系统内存不足时会触发内部的oom-killer机制，把占用内存最高的进程kill掉
- -XX:MaxDirectMemorySize未设置，最大值与堆内存最大值一样

- **解决方案**

- Gateway服务升级Hbase客户端到2.x版本
- 设置最大直接内存的JVM参数（如-XX:MaxDirectMemorySize=1G），并在触发内存使用率报警时，手动重启实例

JVM内存问题排查思路

1、查看应用服务是否有明显异常日志

2、查看JVM监控Dashboard，是否有明显异常（Grafana Dashboard）

*互金JVM监控：*http://ump.inf.sogou/d/h3Pro7nZz/jvmjian-kong?orgId=1

3、查看dump目录，是否有新的dump文件生成（通过-XX:HeapDumpPath指定）

4、查看error目录，是否有新的error文件生成（通过-XX:ErrorFile指定）

5、查看syslog，是否有明显异常（查看/var/log/messages文件或/var/log/audit/audit.log文件）

6、使用MAT分析堆内存dump文件

7、分析GC日志（通过命令或gceasy）

案例：线程堆栈分析

■ 通过命令

1、查看使用CPU最高的进程，记下进程pid

```
top
```

2、查看该进程下消耗CPU较高的线程，记录线程tid

```
top –Hp <pid>
```

3、把线程ID转换为十六进制tid_16

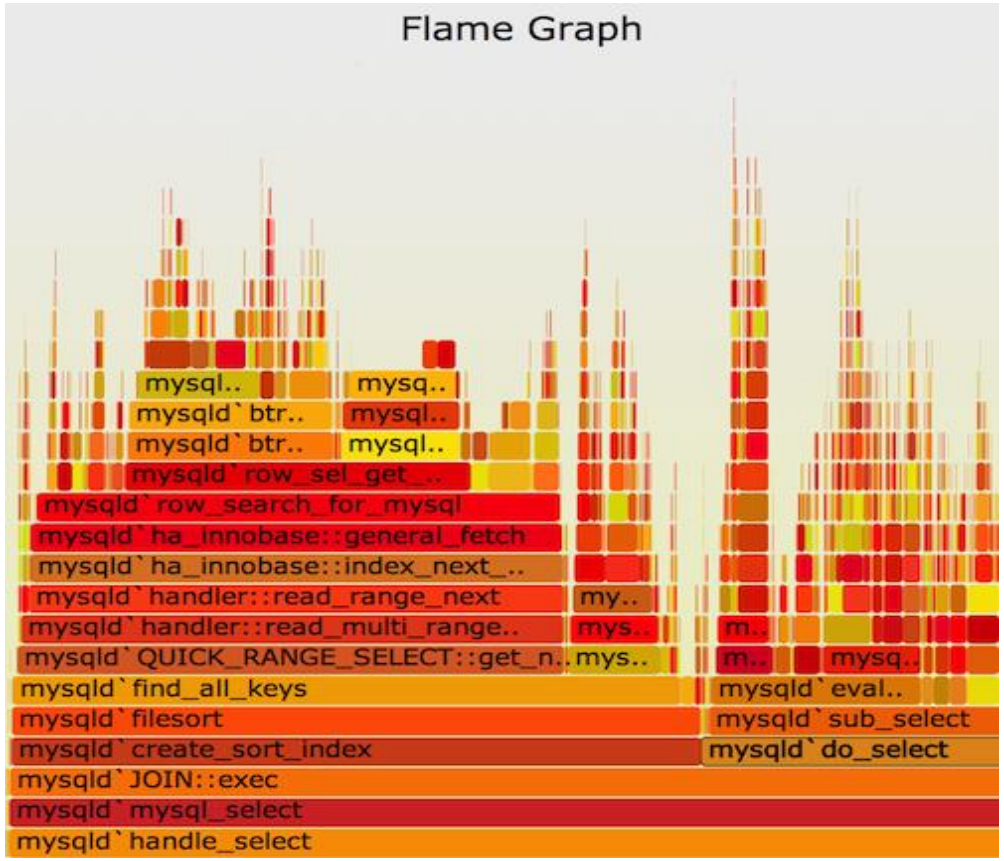
```
printf‘0x%x\n’<tid>
```

4、打印tid_16的线程堆栈

```
jstack <pid> | grep <tid_16> –A 30
```

案例：线程堆栈分析

- 通过CPU火焰图



1）y 轴：**表示调用栈**

2）x 轴：**表示抽样数**。注意，**x 轴不代表时间**

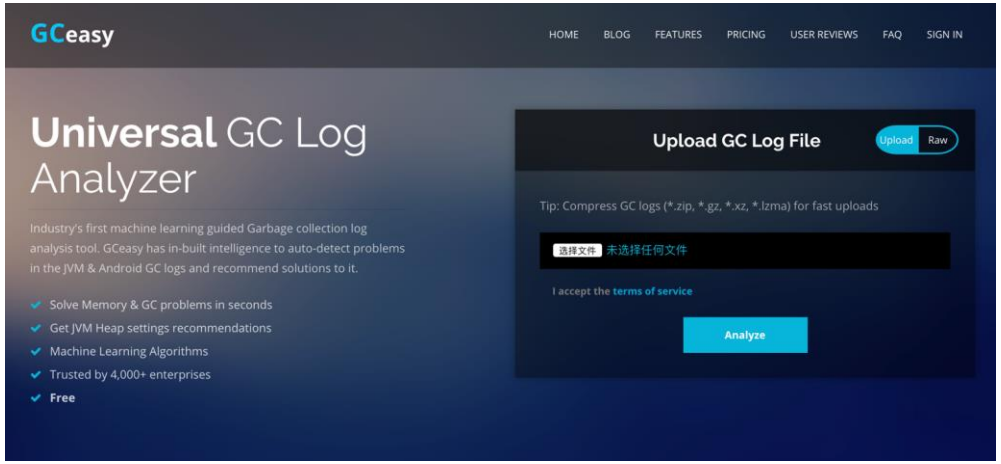
3）火焰图主要看**顶层的哪个函数占据的宽度最大**。只要有**"平顶"**（plateaus），就表示该函数可能存在性能问题。

4）颜色没有特殊含义

案例：线程堆栈分析

■ Gceasy火焰图

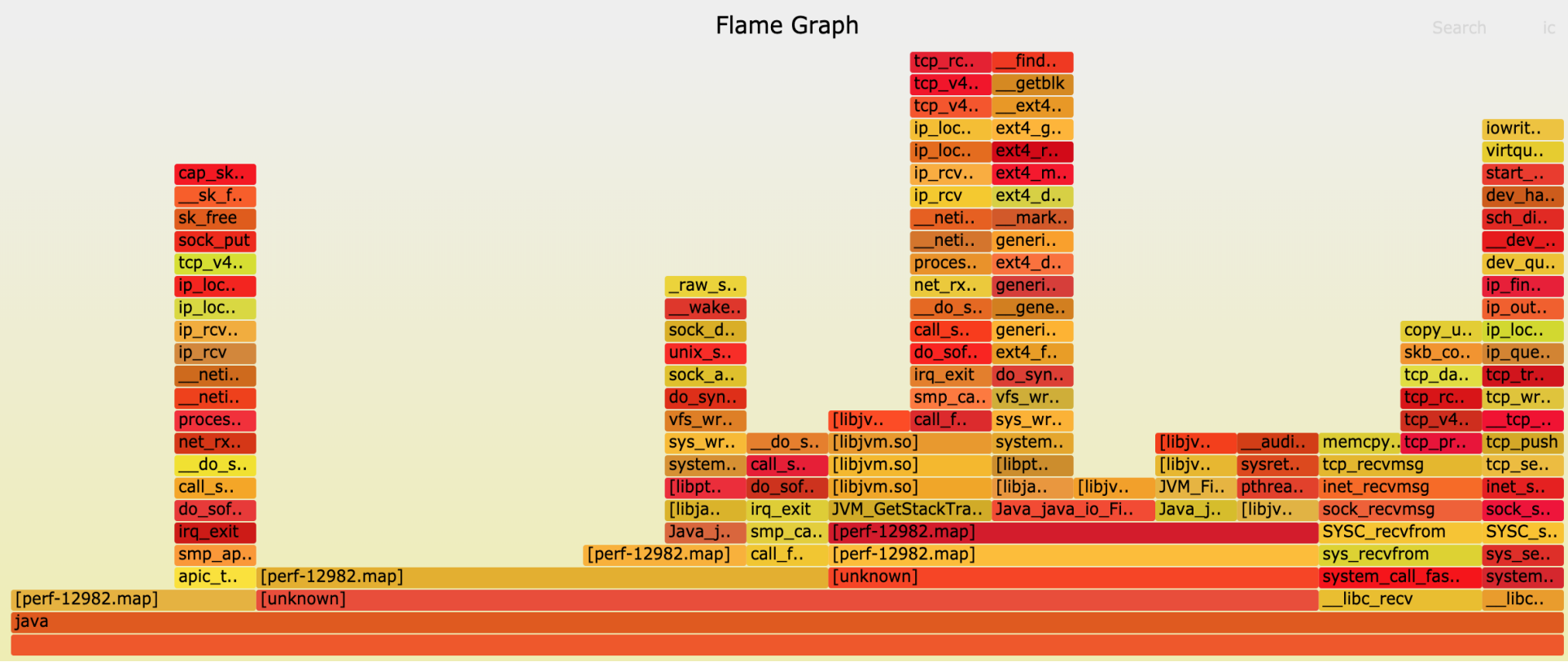
Gceasy是一个用来分析线程堆栈和GC日志的工具网站



案例：线程堆栈分析

- Perf火焰图

  Perf是从linux 2.6.31内核开始自带的一个性能分析工具，基于事件采样原理，用于性能瓶颈的查找与热点代码的定位（生成采集数据 -> 采集数据分析 -> 堆栈折叠处理 -> 生成svg火焰图 -> 浏览器打开svg火焰图）

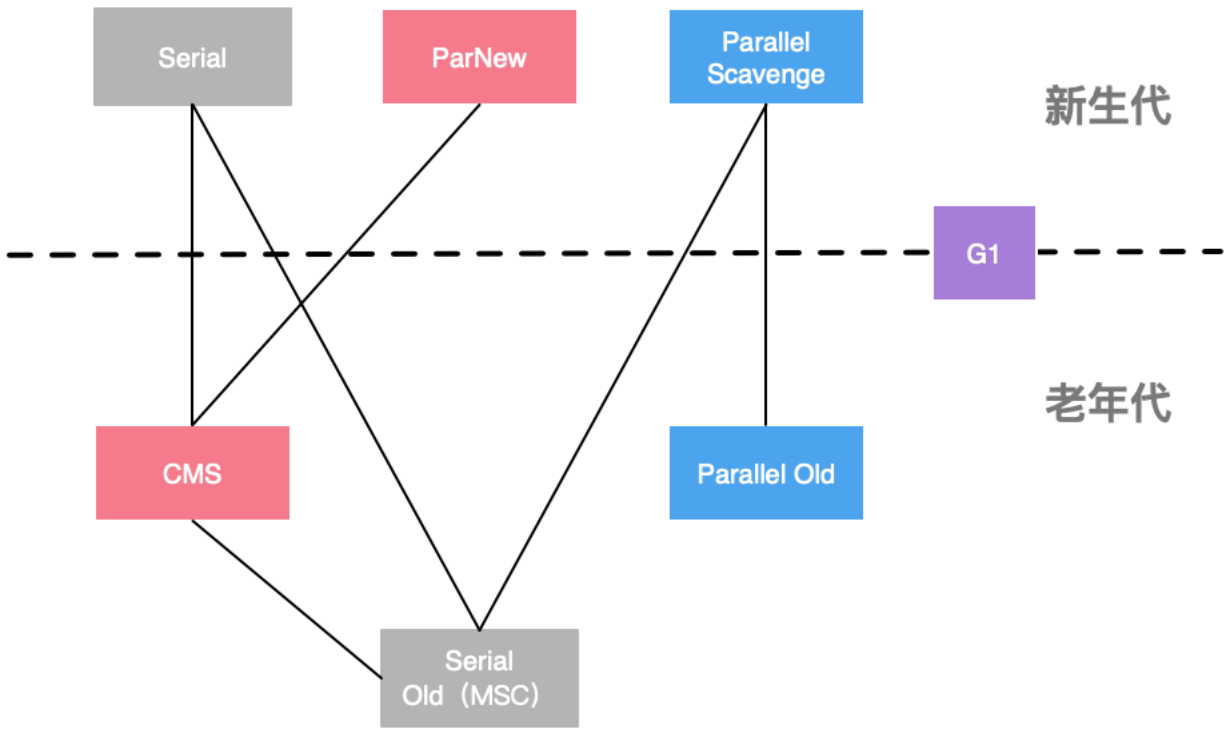


Gceasy火焰图 vs Perf火焰图

	Perf	Gceasy
互动性	支持	支持
CPU火焰图	支持	支持
高阶火焰图支持	支持红蓝差分火焰图、CPI火焰图等	不支持
精确度	高，基于一段时间的CPU高频采样	中，基于jstack时CPU瞬时采样
便捷性	较低，生成步骤较多	高，直接上传jstack线程堆栈文件即可直接生成
对应用服务器影响	无明显影响	无任何影响
功能性	对于CPU堆栈分析只支出火焰图	支持按线程状态或线程组统计，死锁统计等功能

三、垃圾回收

垃圾收集器



常用组合

- Parallel Scavenge + Parallel Old（jdk 1.8默认）
- ParNew + CMS + Serial Old
- G1

Parallel Scavenge

吞吐量优先收集器。主要适用于对客交互较少的后台运算，对响应时间敏感性不太强，以期实现可控的吞吐量的场景

吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）

- **主要参数**

- -XX:MaxGCPauseMillis：控制最大垃圾收集停顿时间，**以牺牲吞吐量和新生代空间来换取**
- -XX:GCTimeRatio：控制垃圾收集时间占总时间的比率，**默认99，即允许1%的GC停顿**
- -XX:+UseAdaptiveSizePolicy：是否开启GC自适应调节策略，**默认关闭，开启后无需指定-Xmn、-XX:SurvivorRatio等参数**

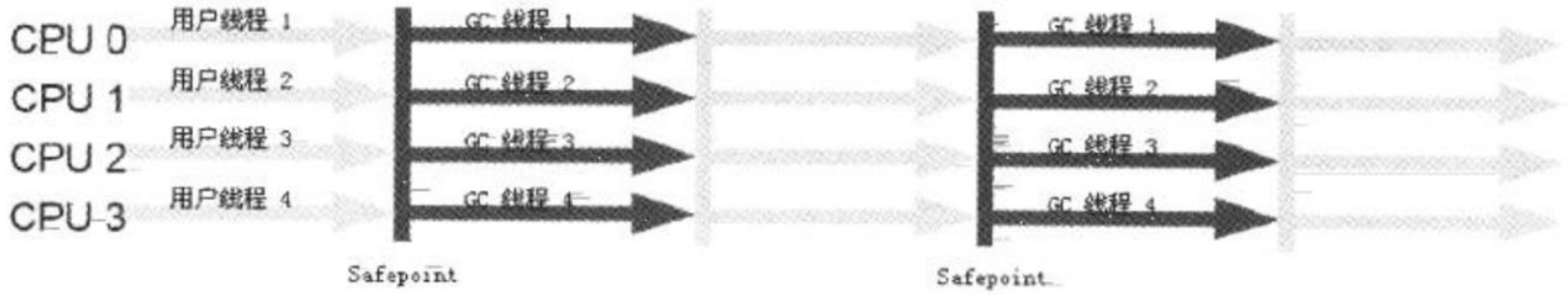


图3-9 Parallel Scavenge / Parallel Old收集器运行示意图

CMS

获得最短停顿时间， 主要适用于对响应时间敏感的对客应用，以尽可能降低对用户体验的影响

- **缺点**

1、对CPU资源非常敏感。默认回收线程数（CPU数量+3）/ 4，当CPU小于4个时，对应用服务影响可能变得很大

2、无法处理浮动垃圾。收集过程中可能出现Concurrent Mode Failure导致另一次Full GC的产生（Serial Old）

（-XX:CMSInitiatingOccupancyfraction）

3、收集结束会产生大量的内存碎片。导致分配大对象可能无法找到足够大的连续空间而触发一次FullGC

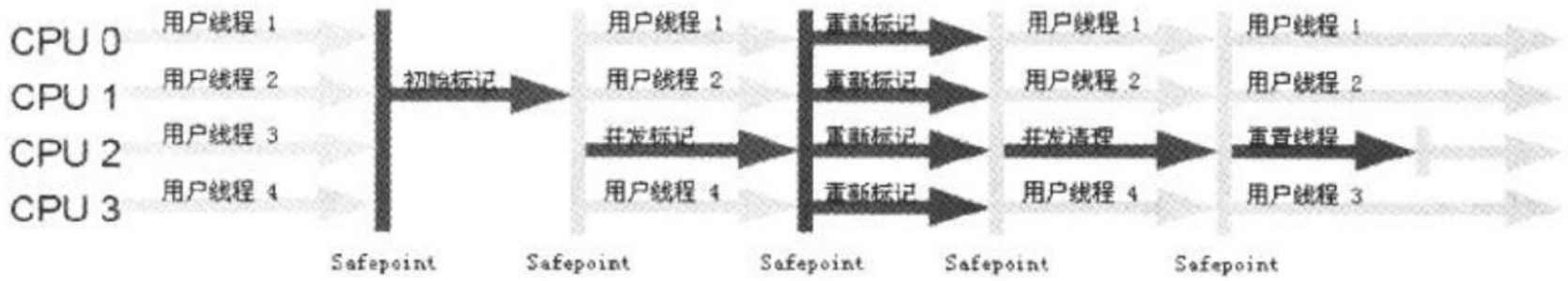
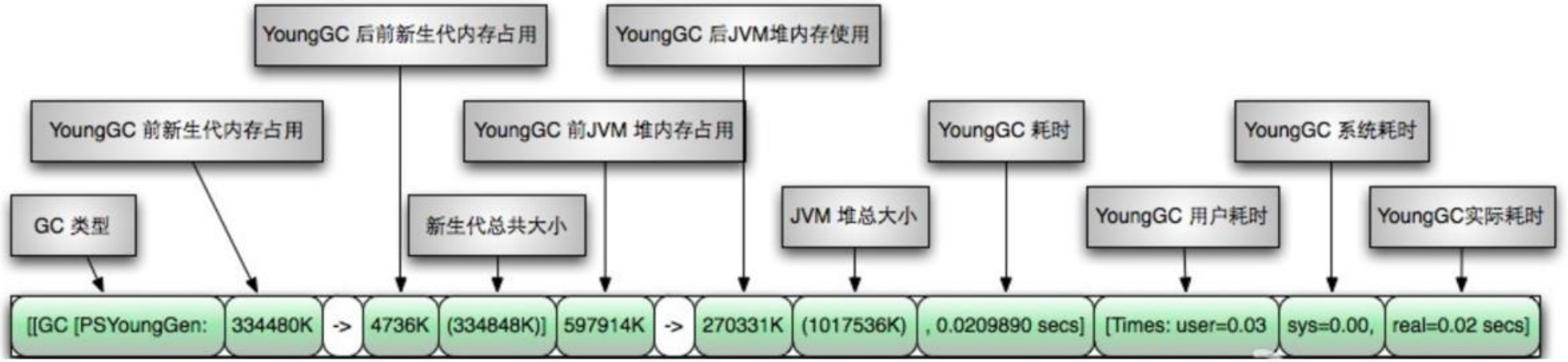


图 3-10　Concurrent Mark Sweep 收集器运行示意图

理解GC日志：Young GC日志

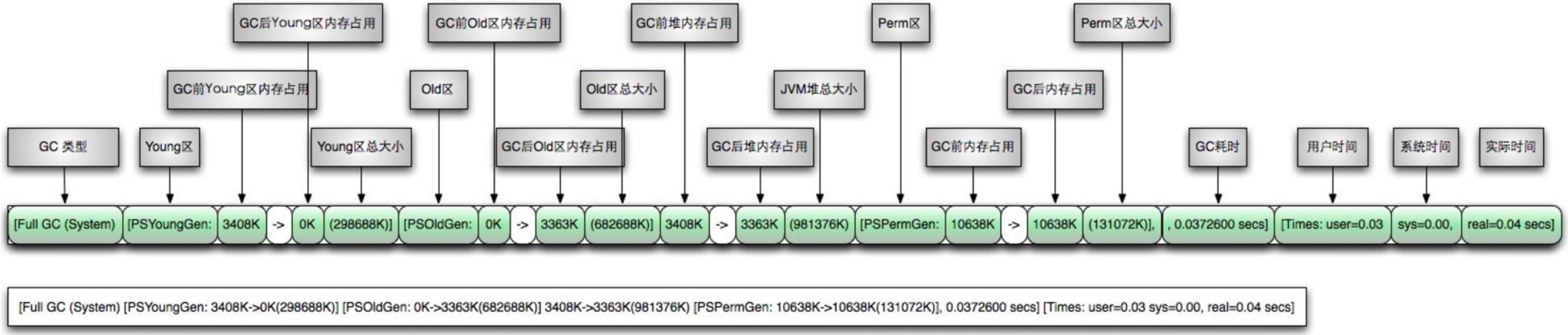
GC （minor ）日志



- user time：用户态消耗的CPU时间
- sys time：内核态消耗的CPU时间
- real time：GC从开始到结束所经过的墙钟时间，包括非运算等待耗时，如磁盘IO等待、线程阻塞等待等

理解GC日志：Full GC日志

Full GC 日志



- **Full GC类型**

  Metadata GC Threshold、Ergonomics、Heap Dump Initiated GC、System.gc()等

Full GC：Metadata GC Threshold

- **MetaspaceSize参数**

默认20.8M左右，**主要是控制Metaspace GC（Full GC）发生的初始阈值**，也是最小阈值，触发Metaspace GC的阈值是不断变化的，**建议调整该阈值到JVM稳定运行一段时间后的值**

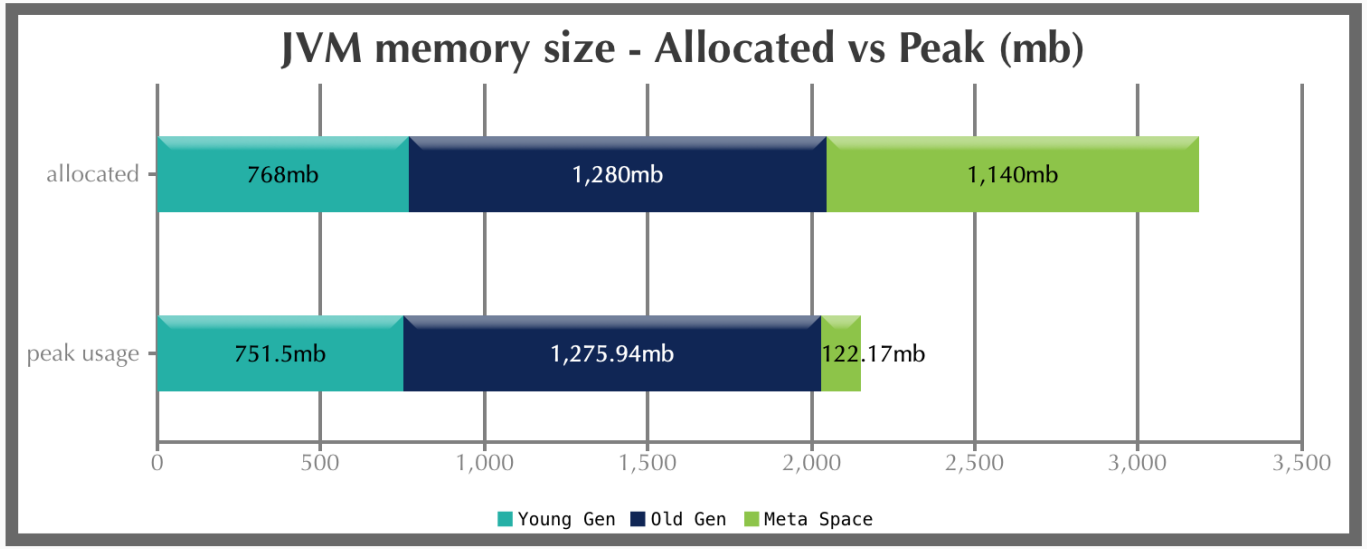
- **MaxMetaspaceSize参数**

默认基本是无穷大，建议配置。因为很可能会因为没有限制而导致Metaspace被无限制使用（一般是内存泄漏）而触发操作系统的oom-killer。这个参数会限制Metaspace被Committed的内存大小，一旦超过就会触发Full GC

案例：GC日志分析（通过gceasy.io ）

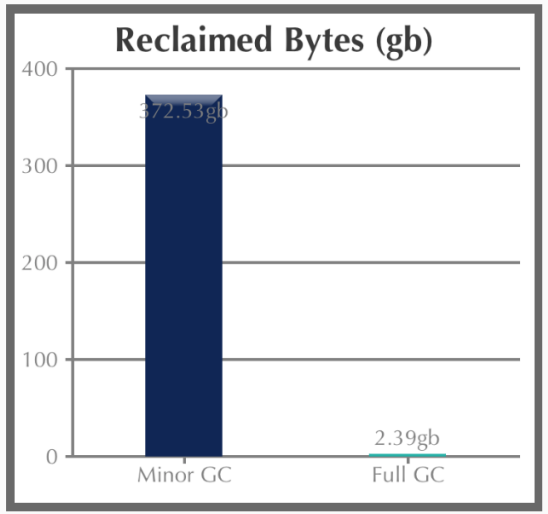
JVM memory size

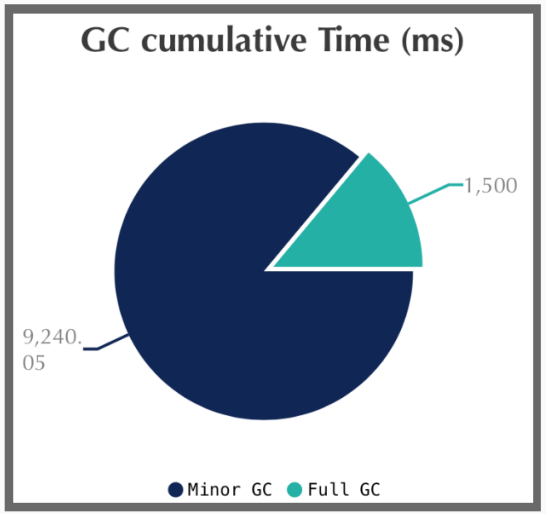
Generation	Allocated	Peak
Young Generation	768 mb	751.5 mb
Old Generation	1.25 gb	1.25 gb
Meta Space	1.11 gb	122.17 mb
Young + Old + Meta space	3.11 gb	2.08 gb

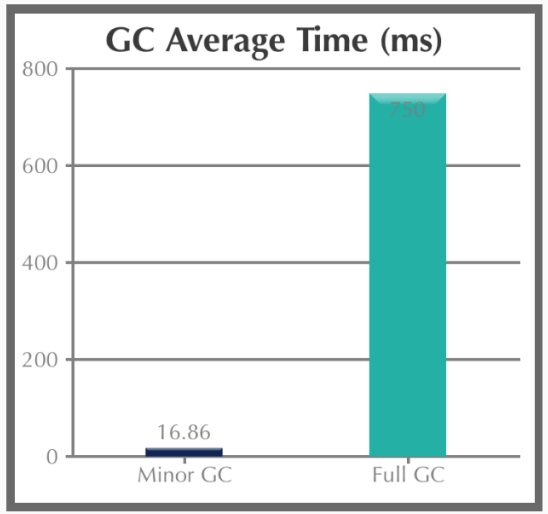


案例：GC日志分析

GC Statistics







Total GC stats

Total GC count	550
Total reclaimed bytes	374.91 gb
Total GC time	10 sec 740 ms
Avg GC time	19.5 ms
GC avg time std dev	47.7 ms
GC min/max time	0 / 940 ms
GC Interval avg time	5 min 10 sec 382 ms

Minor GC stats

Minor GC count	548
Minor GC reclaimed	372.53 gb
Minor GC total time	9 sec 240 ms
Minor GC avg time	16.9 ms
Minor GC avg time std dev	14.0 ms
Minor GC min/max time	0 / 220 ms
Minor GC Interval avg	5 min 11 sec 517 ms

Full GC stats

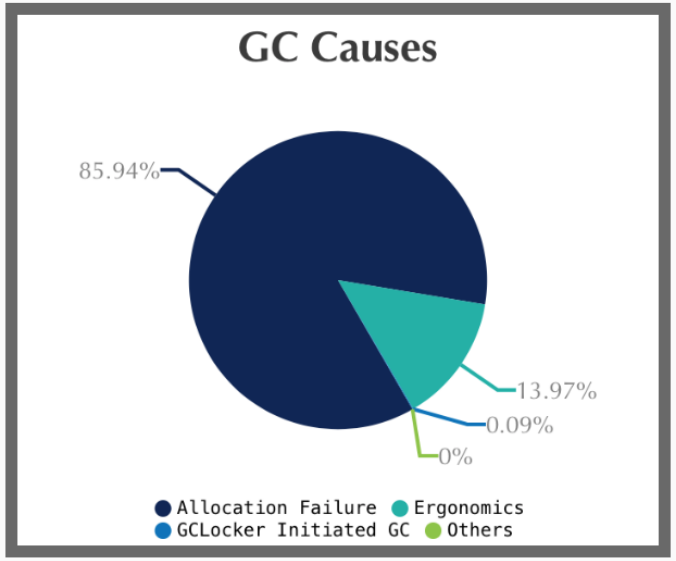
Full GC Count	2
Full GC reclaimed	2.39 gb
Full GC total time	1 sec 500 ms
Full GC avg time	750 ms
Full GC avg time std dev	190 ms
Full GC min/max time	560 ms / 940 ms
Full GC Interval avg	18 hrs 47 min 45 sec

案例：GC日志分析

GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Allocation Failure	547	16.9 ms	220 ms	9 sec 230 ms	85.94%
Ergonomics	2	750 ms	940 ms	1 sec 500 ms	13.97%
GCLocker Initiated GC	1	10.0 ms	10.0 ms	10.0 ms	0.09%
Others	n/a	n/a	n/a	0.0134 ms	0.0%
Total	550	n/a	n/a	10 sec 740 ms	100.0%



案例：GC日志分析

Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here)

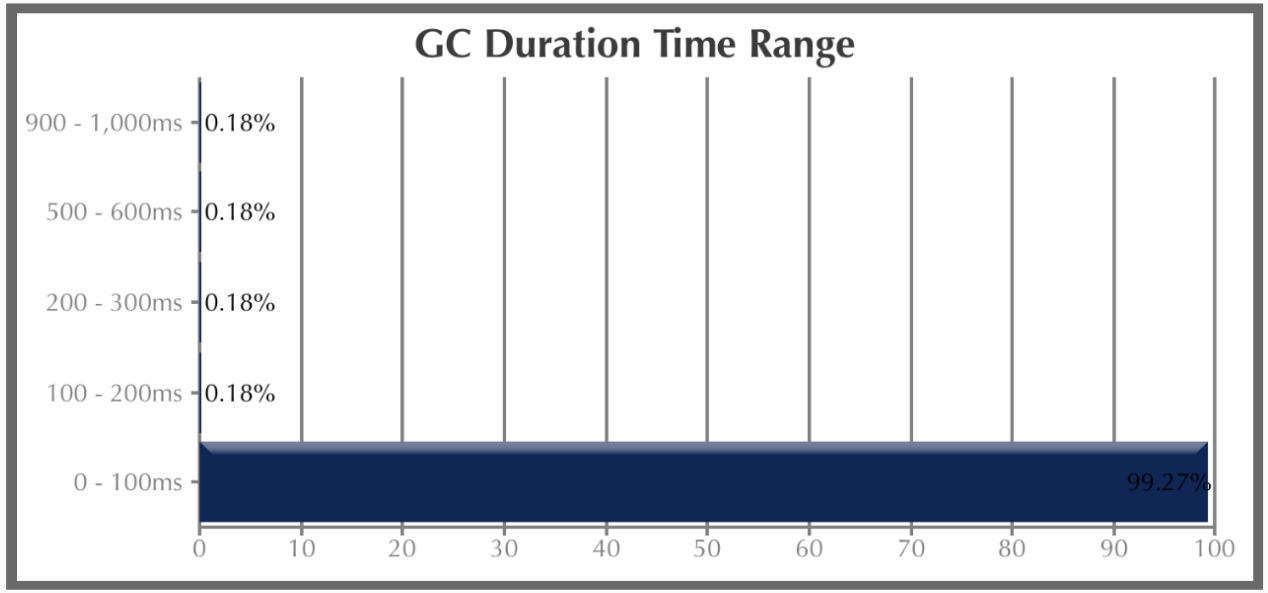
1 **Throughput** : **99.994%**

2 **Latency:**

Avg Pause GC Time	**19.5 ms**
Max Pause GC Time	**940 ms**

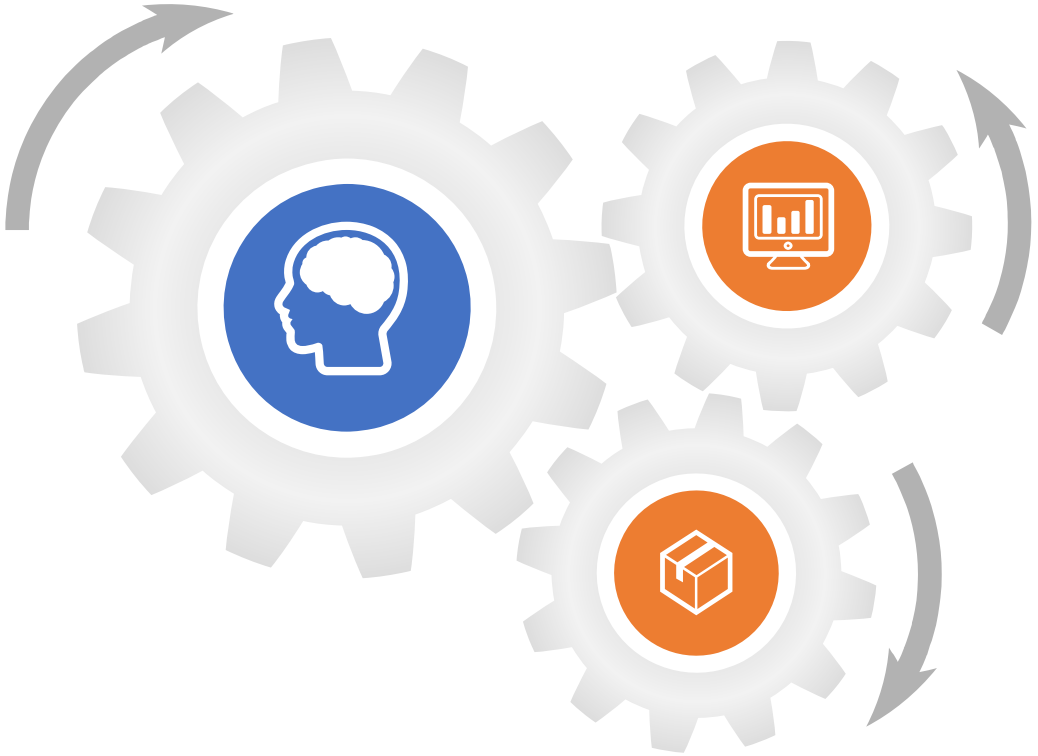
GC **Pause** Duration Time Range :

Duration (ms) 100 m Change	No. of GCs	Percentage
0 - 100	546	99.27%
100 - 200	1	0.18%
200 - 300	1	0.18%
500 - 600	1	0.18%
900 - 1,000	1	0.18%



四、运行时优化

JIT编译器（Just In Time Compiler）



主要用于将热点代码的字节码编译成机器码，获得更高的执行效率

热点代码探测

通过执行计数器找出最具有编译价值的代码，然后通知JIT编译器以方法为单位进行编译

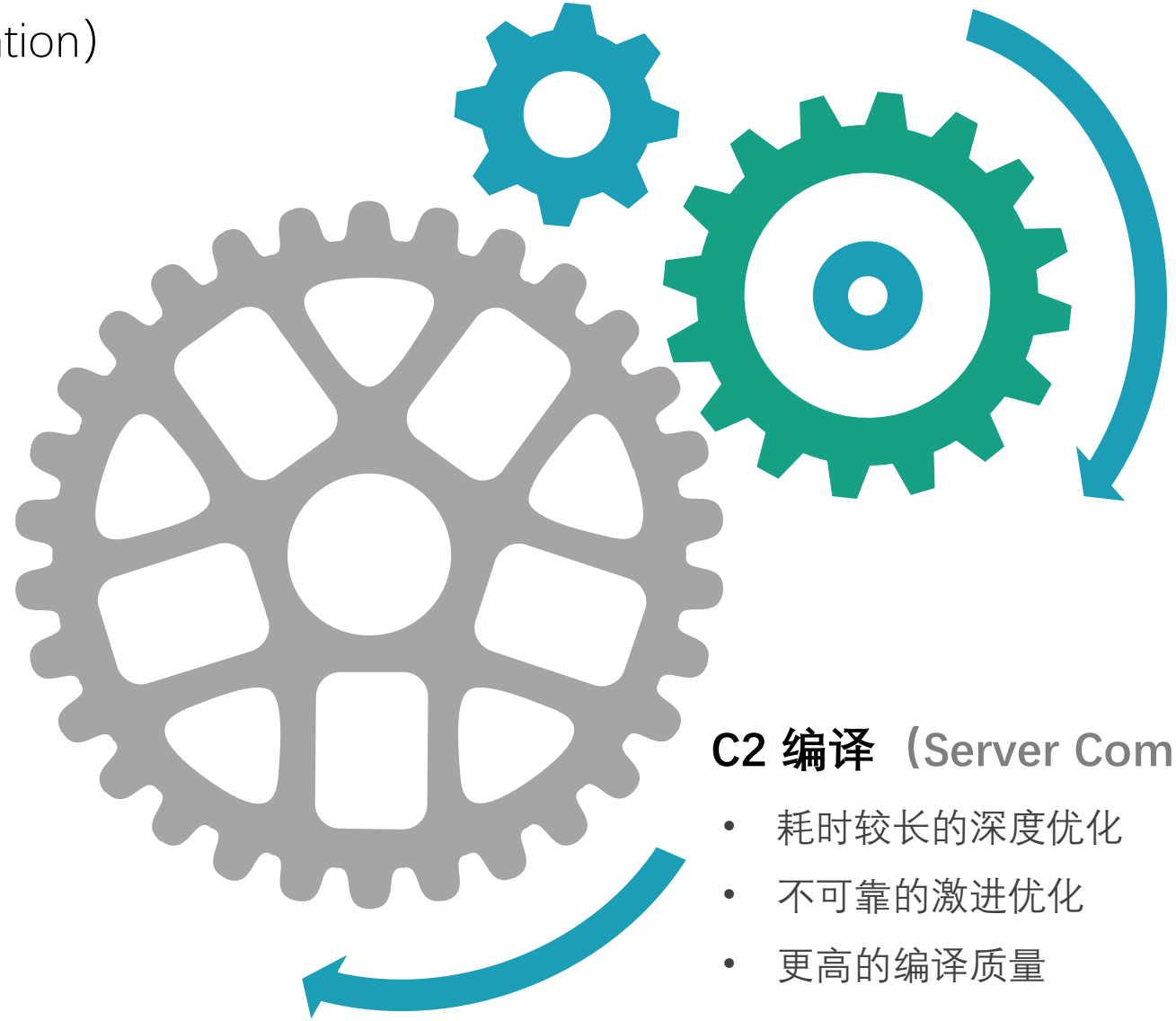
如果一个方法被频繁调用，或方法中有效循环次数很多，将会分别触发标准编译和OSR编译

JIT编译器与解释器协同工作，在最优化程序响应时间与最佳执行性能中取得平衡

分层编译

(Tiered Compilation)

解释执行



C1 编译（Client Compiler）

- 三段式编译
- 简单可靠的局部性优化
- 更快的编译速度

C2 编译（Server Compiler）

- 耗时较长的深度优化
- 不可靠的激进优化
- 更高的编译质量

混合模式（mix mode）

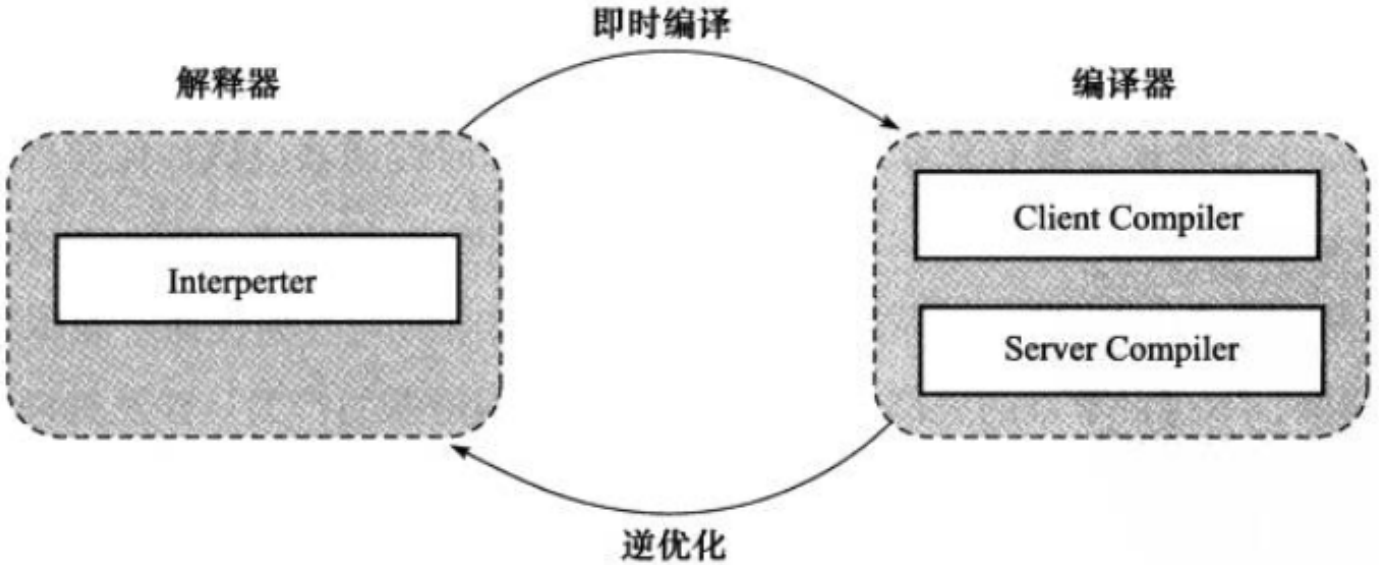


图 11-1　解释器与编译器的交互

常见的运行时优化技术


空值检查消除
自动装箱消除
逃逸分析
内联
范围检查消除
无用代码消除
分层编译
锁消除
栈上替换
消除反射
公共子表达式消除
循环展开

案例：JVM预热（Warm Up）

Warm Up的必要性和性能提升

- No Warm Up在高并发场景下可能导致系统机
- WarmUp会带来几十倍的吞吐量提升
- 灰度发布支持组件（by 架构组）

  服务治理-权重路由接入文档

  权重路由组件GitLab

7. Performance Benchmark

In the last 20 years, most contributions to Java were related to the GC (Garbage Collector) and JIT (Just In Time Compiler). Almost all of the performance benchmarks found online are done on a JVM already running for some time. However,

However, *Beihang University* has published a benchmark report taking into account JVM warm-up time. They used Hadoop and Spark based systems to process massive data:

Completion time (s)	Unmod.	HotTub	Speed-up
HDFS read 1MB	2.29	0.08	30.08x
HDFS read 10MB	2.65	0.14	18.04x
HDFS read 100MB	2.33	0.41	5.71x
HDFS read 1GB	7.08	4.26	1.66x
Spark 100GB best	65.2	36.2	1.80x
Spark 100GB median	57.8	35.2	1.64x
Spark 100GB worst	74.8	54.4	1.36x
Spark 3TB best	66.4	41.4	1.60x
Spark 3TB median	98.4	73.6	1.34x
Spark 3TB worst	381.2	330.0	1.16x
Hive 100GB best	29.0	16.2	1.79x
Hive 100GB median	38.4	25.0	1.54x
Hive 100GB worst	206.6	188.4	1.10x

Here HotTub designates the environment in which the JVM was warmed up.

As you can see, the speed-up can be significant, especially for relatively small read operations – which is why this data is interesting to consider.

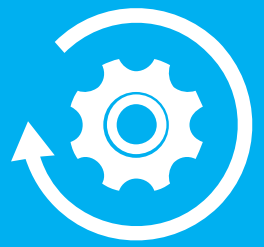
案例：C2编译器OSR内存越界Bug

Trade Crash问题排查及建议解决方案

1、JDK **8u171**版本存在C2编译器内存越界Bug，建议升级到JDK **8u211**版本

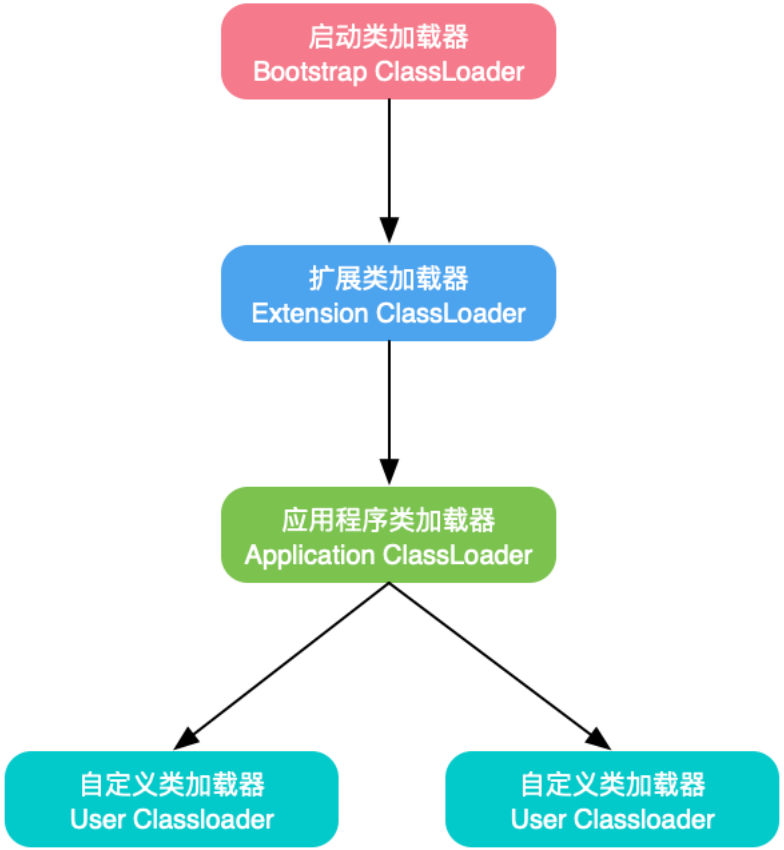
2、此Bug的临时解决方案可通过XX:CompileCommand命令解决，但此后该方法则无法进行JIT优化

命令参数：XX:CompileCommand=exclude,<class-name>,<method-name>

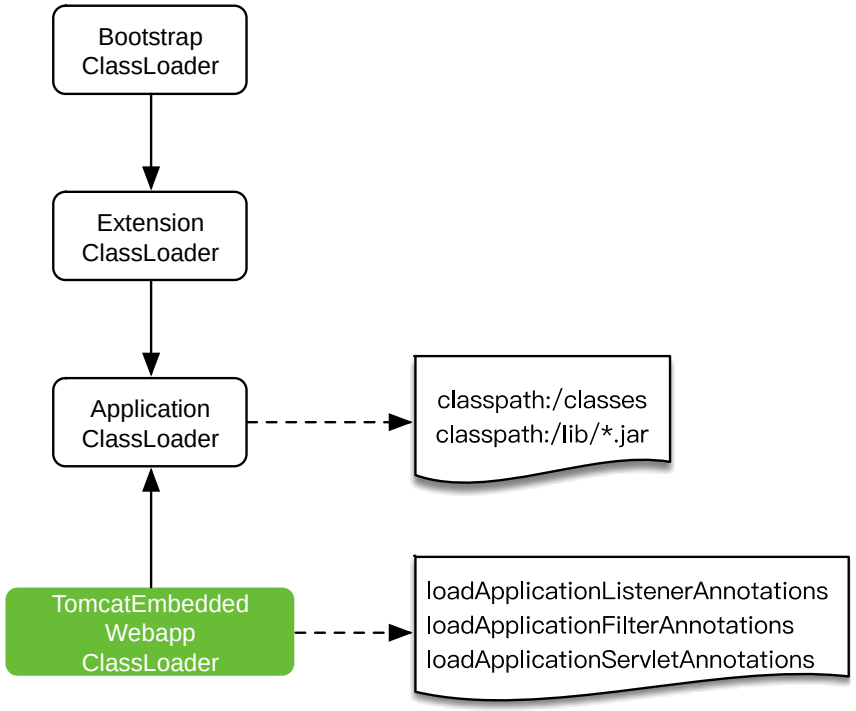


五、类加载机制

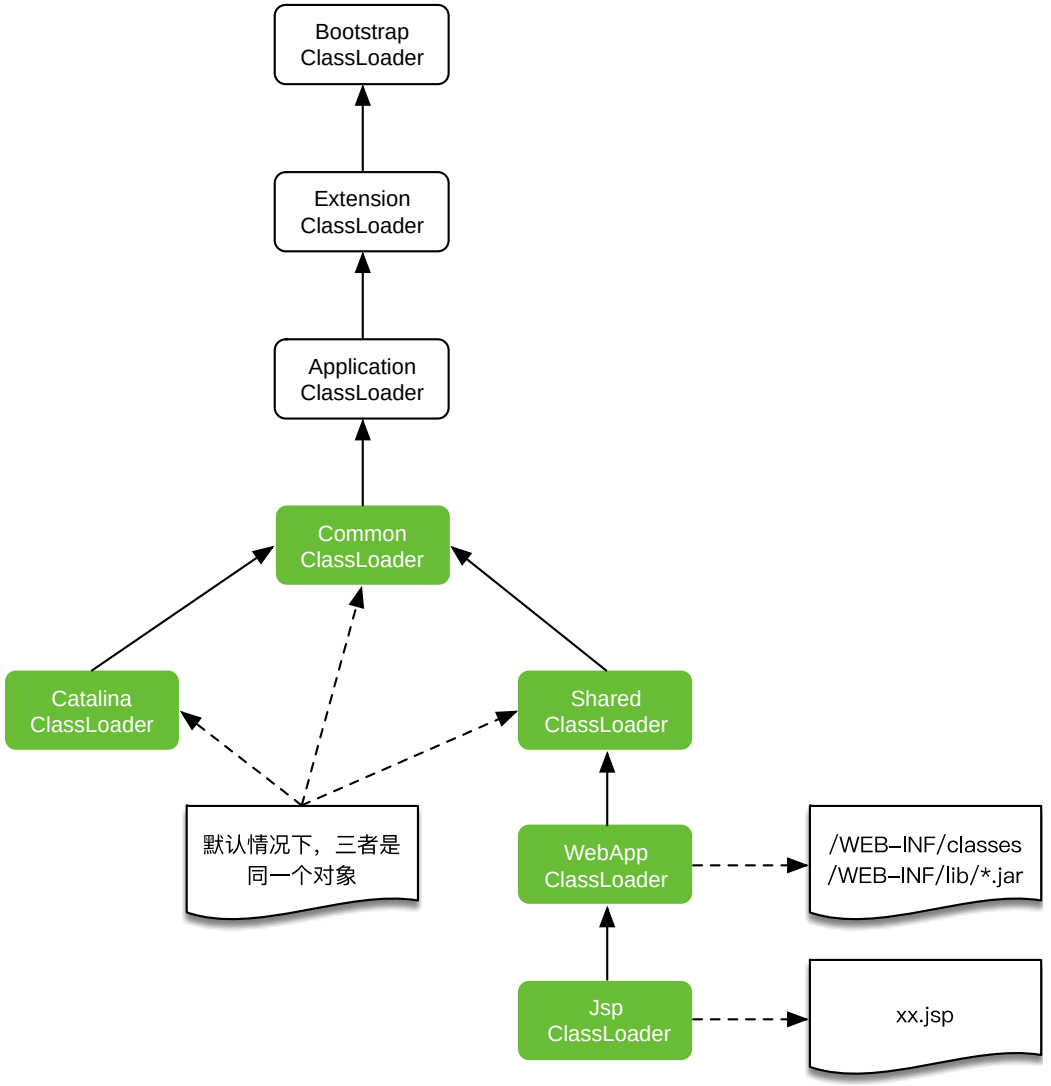
双亲委派模型


启动类加载器
Bootstrap ClassLoader
扩展类加载器
Extension ClassLoader
应用程序类加载器
Application ClassLoader
自定义类加载器
User Classloader
自定义类加载器
User Classloader

Tomcat类加载器

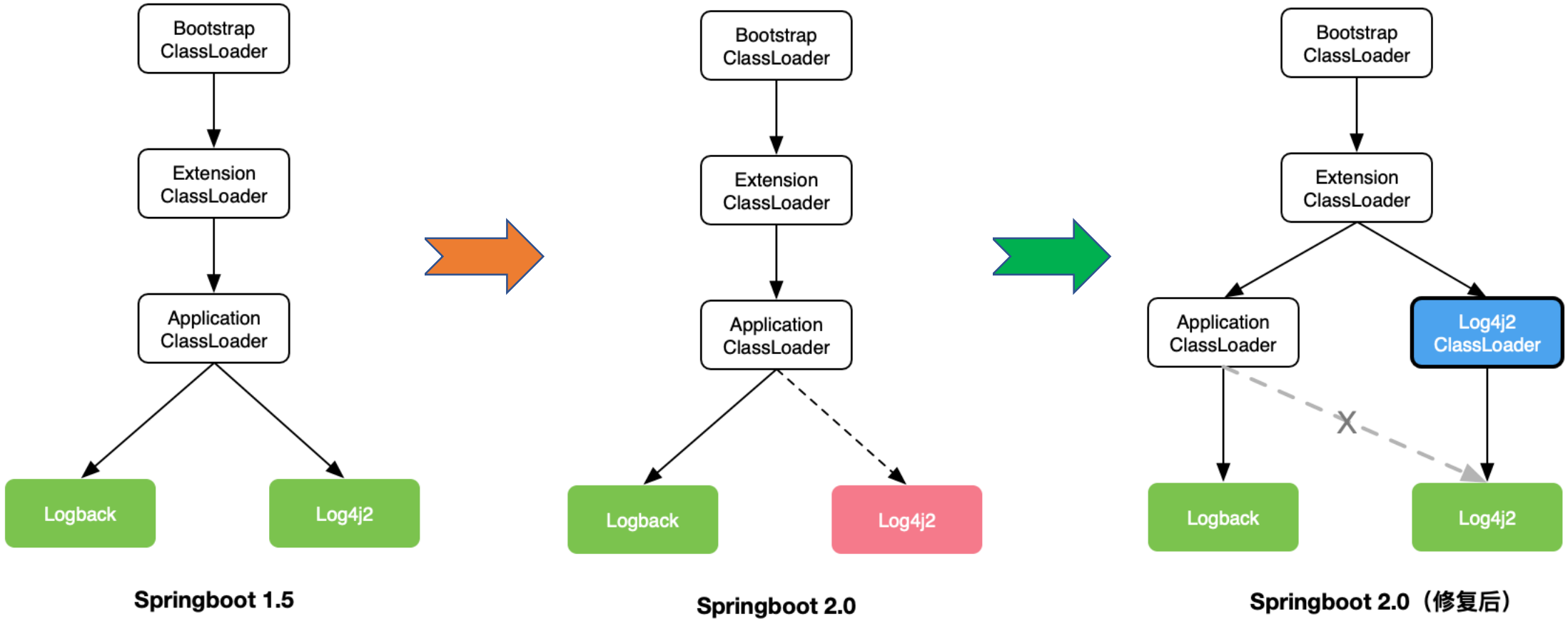


嵌入式Tomcat



非嵌入式Tomcat

案例：spring2.0同时使用logback和log4j2



End

Thanks~