

极客大学 Java 进阶训练营
第 2 课

JVM 核心技术--工具与 GC 策略



KimmKing

Apache Dubbo/ShardingSphere PMC

个人介绍

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

熟悉海量并发低延迟交易系统的设计实现

目 录

1. JDK 内置命令行工具*
2. JDK 内置图形化工具*
3. GC 的背景与一般原理
4. 串行 GC/并行 GC* (Serial GC/Parallel GC)
5. CMS GC/G1 GC*
6. ZGC/Shenandoah GC
7. 第 2 节课总结回顾与作业实践

第 2 课 JVM 核心技术--工具与 GC

1. JDK 内置命令行工具

JVM 命令行工具

工具	简介
java	Java 应用的启动程序
javac	JDK 内置的编译工具
javap	反编译 class 文件的工具
javadoc	根据 Java 代码和标准注释, 自动生成相关的 API 说明文档
javah	JNI 开发时, 根据 java 代码生成需要的 .h 文件
extcheck	检查某个 jar 文件和运行时扩展 jar 有没有版本冲突, 很少使用
jdb	Java Debugger ; 可以调试本地和远程程序, 属于 JPDA 中的一个 demo 实现, 供其他调试器参考。开发时很少使用
jdeps	探测 class 或 jar 包需要的依赖
jar	打包工具, 可以将文件和目录打包成为 .jar 文件; .jar 文件本质上就是 zip 文件, 只是后缀不同。使用时按顺序对应好选项和参数即可。
keytool	安全证书和密钥的管理工具; (支持生成、导入、导出等操作)
jarsigner	JAR 文件签名和验证工具
policytool	实际上这是一款图形界面工具, 管理本机的 Java 安全策略

JVM 命令行工具

工具	简介
jps/jinfo	查看 java 进程
jstat	查看 JVM 内部 gc 相关信息
jmap	查看 heap 或类占用空间统计
jstack	查看线程信息
jcmd	执行 JVM 相关分析命令 (整合命令)
jrunscript/jjs	执行 js 命令

JVM 命令行工具--jps/jinfo

D:\>jps

32432

1716 Jps

23784 QuorumPeerMain

4264 Bootstrap

2460 Launcher

D:\>jps -help

usage: jps [-help]

jps [-q] [-mlvV] [<hostid>]

Definitions:

<hostid>: <hostname>[:<port>]

```
D:\>jps -mlv
32432  exit -Xms128m -Xmx4096m -XX:ReservedCodeCacheSize=240m -XX:+UseConcMarkSweepGC -XX:SoftRefLRUPolicyMSPerMB=50 -ea -XX:CICompilerCount=2 -Dsun.io.useCanonPrefixCache=false -Djava.net.preferIPv4Stack=true -Djdk.http.auth.tunneling.disabledSchemes="" -XX:+HeapDumpOnOutOfMemoryError -XX:-OmitStackTraceInFastThrow -Djdk.attach.allowAttachSelf=true -Dkotlinx.coroutines.debug=off -Djdk.module.illegalAccess.silent=true -Djb.vmOptionsFile=C:\Users\qinjinwei5\AppData\Roaming\JetBrains\IntelliJIdea2020.2\idea64.exe.vmoptions -Djava.library.path=D:\tools\IntelliJ IDEA 2020.2.1\jbr\bin;D:\tools\IntelliJ IDEA 2020.2.1\jbr\bin\server -Didea.jreheck=true -Didea.native.launcher=true -Didea.vendor.name=JetBrains -Didea.paths.selector=IntelliJIdea2020.2 -XX:ErrorFile=C:\Users\qinjinwei5\java_error_in_idea_%p.log -XX:HeapDumpPath=C:\Users\qinjinwei5\java_error_in_idea.hprof
3648 C:\Users\QINJIN~1\AppData\Local\Temp\surefire6365885744555543168\surefirebooter3815716460569744994.jar
C:\Users\qinjinwei5\AppData\Local\Temp\surefire6365885744555543168 2020-10-13T22-16-28_460-jvmRun1 surefire7538458295501138080tmp surefire_57995250922938979483tmp -javaagent:d:\\tools\\.m2\repository\org\jacoco\org.jacoco.agent\0.8.5\org.jacoco.agent-0.8.5-runtime.jar=destfile=D:\git\shardingsphere\shardingsphere-sql-parser\shardingsphere-sql-parser-dialect\shardingsphere-sql-parser-oracle\target\jacoco.exec -Xmx1024m -XX:MaxMetaspaceSize=256m
23784 org.apache.zookeeper.server.quorum.QuorumPeerMain D:\tools\apache-zookeeper-3.6.1-bin\bin..\conf\zoo.cfg -Dzookeeper.log.dir=D:\tools\apache-zookeeper-3.6.1-bin\bin..\logs -Dzookeeper.root.logger=INFO,CONSOLE -Dzookeeper.log.file=zookeeper-qinjinwei5-server-JRA1W1PF227YSH.log -XX:+HeapDumpOnOutOfMemoryError -XX:OnOutOfMemoryError=cmd /c taskkill /pid %p /t /f
25224 sun.tools.jps.Jps -mlv -Dapplication.home=D:\tools\jdk8 -Xms8m
4264 org.apache.shardingsphere.proxy.Bootstrap -Xmx2g -Xms2g -Xmn1g -Xss256k -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 -Dfile.encoding=UTF-8
2460 org.codehaus.plexus.classworlds.launcher.Launcher clean install -Xmx1024m -XX:MaxMetaspaceSize=256m -XX:ParallelGCThreads=4 -XX:+UseConcMarkSweepGC -Dclassworlds.conf=D:\tools\apache-maven-3.6.3-aliyun\bin..\bin\m2.conf -Dmaven.home=D:\tools\apache-maven-3.6.3-aliyun\bin.. -Dlibrary.jansi.path=D:\tools\apache-maven-3.6.3-aliyun\bin..\lib\jansi-native -Dmaven.multiModuleProjectDirectory=D:\git\shardingsphere
```

JVM 命令行工具--jstat*

```
$ jstat -help
```

```
1 Usage: jstat -help|-options
2     jstat -<option> [<t> [-h<lines>] <vmid> [<interval> [<count>]]
3
4 Definitions:
5   <option> 可用的选项, 查看详情请使用 -options
6   <vmid> 虚拟机标识符. 格式: <lvmid>[@<hostname>[:<port>]]
7   <lines> 标题行间隔的频率.
8   <interval> 采样周期, <n>["ms"|"s"], 默认单位是毫秒 "ms".
9   <count> 采用总次数.
10  -J<flag> 传给jstat底层JVM的 <flag> 参数.
```

> jstat -options

-class 类加载(Class loader)信息统计

-compiler JIT 即时编译器相关的统计信息

-gc GC 相关的堆内存信息。 用法: jstat -gc -h 10 -t 864 1s 20

-gccapacity 各个内存池分代空间的容量

-gccause 看上次 GC , 本次 GC (如果正在 GC 中) 的原因 , 其他输出和 -gcutil 选项一致

-gcnew 年轻代的统计信息。 (New = Young = Eden + S0 + S1)

-gcnewcapacity 年轻代空间大小统计

-gcold 老年代和元数据区的行为统计

-gcoldcapacity old 空间大小统计

-gcmetacapacity meta 区大小统计

-gcutil GC 相关区域的使用率 (utilization) 统计

-printcompilation 打印 JVM 编译统计信息

JVM 命令行工具--jstat

Timestamp	S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
14251645.5	0.00	13.50	55.05	71.91	83.84	69.52	113767	206.036	4	0.122	206.158

-t 选项的位置是固定的，不能在前也不能在后。可以看出是用于显示时间戳，即JVM启动到现在的秒数。

简单分析一下：

- **Timestamp** 列: JVM启动了1425万秒,大约164天。
- **S0** 就是0号存活区的百分比使用率。 0%很正常, 因为 S0和S1随时有一个是空的。
- **S1** 就是1号存活区的百分比使用率。
- **E** 就是Eden区, 新生代的百分比使用率。
- **O** 就是Old区, 老年代。百分比使用率。
- **M** 就是Meta区, 元数据区百分比使用率。
- **CCS**, 压缩class空间(Compressed class space)的百分比使用率。
- **YGC** (Young GC), 年轻代GC的次数。11万多次, 不算少。
- **YGCT** 年轻代GC消耗的总时间。206秒, 占总运行时间的万分之一不到, 基本上可忽略。
- **FGC** FullGC的次数,可以看到只发生了4次, 问题应该不大。
- **FGCT** FullGC的总时间, 0.122秒, 平均每次30ms左右,大部分系统应该能承受。
- **GCT** 所有GC加起来消耗的总时间, 即 **YGCT** + **FGCT** 。

可以看到, **-gcutil** 这个选项出来的信息不太好用, 统计的结果是百分比, 不太直观。

再看看, **-gc** 选项, GC相关的堆内存信息。

演示：

```
jstat -gcutil pid 1000 1000
```

JVM 命令行工具--jstat

Timestamp	S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	YGC	YGCT	FGC	FGCT
14254245.3	1152.0	1152.0	145.6	0.0	9600.0	2312.0	11848.0	8527.3	31616.0	26528.6	11378.8	206.082	4	0.122
14254246.3	1152.0	1152.0	145.6	0.0	9600.0	2313.0	11848.1	8527.0	31616.0	26528.6	11378.8	206.082	4	0.122
14254247.3	1152.0	1152.0	145.6	0.0	9600.0	2313.0	11848.4	8527.0	31616.0	26528.6	11378.8	206.082	4	0.122

上面的结果是精简过的, 为了排版去掉了 `GCT`, `CCSC`, `CCSU` 这三列。看到这些单词可以试着猜一下意思, 详细的解读如下:

- `Timestamp` 列: JVM启动了1425万秒, 大约164天。
- `S0C` : 0号存活区的当前容量(capacity), 单位 kB.
- `S1C` : 1号存活区的当前容量, 单位 kB.
- `S0U` : 0号存活区的使用量(utilization), 单位 kB.
- `S1U` : 1号存活区的使用量, 单位 kB.
- `EC` : Eden区, 新生代的当前容量, 单位 kB.
- `EU` : Eden区, 新生代的使用量, 单位 kB.
- `OC` : Old区, 老年代的当前容量, 单位 kB.
- `OU` : Old区, 老年代的使用量, 单位 kB. (!需要关注)
- `MC` : 元数据区的容量, 单位 kB.
- `MU` : 元数据区的使用量, 单位 kB.
- `CCSC` : 压缩的class空间容量, 单位 kB.
- `CCSU` : 压缩的class空间使用量, 单位 kB.
- `YGC` : 年轻代GC的次数。
- `YGCT` : 年轻代GC消耗的总时间。 (!重点关注)
- `FGC` : Full GC 的次数
- `FGCT` : Full GC 消耗的时间. (!重点关注)
- `GCT` : 垃圾收集消耗的总时间。

演示

`jstat -gc pid 1000 1000`

JVM 命令行工具--jmap

```
Concurrent Mark-Sweep GC

Heap Configuration:
  MinHeapFreeRatio      = 40
  MaxHeapFreeRatio      = 70
  MaxHeapSize           = 1073741824 (1024.0MB)
  NewSize               = 88735744 (84.625MB)
  MaxNewSize             = 348913664 (332.75MB)
  OldSize               = 177602560 (169.375MB)
  NewRatio              = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 260046848 (248.0MB)
  MaxMetaspaceSize       = 268435456 (256.0MB)
  G1HeapRegionSize        = 0 (0.0MB)

Heap Usage:
  New Generation (Eden + 1 Survivor Space):
    capacity = 79888384 (76.1875MB)
    used      = 13292272 (12.676498413085938MB)
    free      = 66596112 (63.51100158691406MB)
    16.638554110695242% used
  Eden Space:
    capacity = 71041024 (67.75MB)
    used      = 7739720 (7.381172180175781MB)
    free      = 63301304 (60.36882781982422MB)
    10.894719085130305% used
  From Space:
    capacity = 8847360 (8.4375MB)
    used      = 5552552 (5.295326232910156MB)
    free      = 3294808 (3.1421737670898438MB)
    62.759422019675924% used
  To Space:
    capacity = 8847360 (8.4375MB)
    used      = 0 (0.0MB)
    free      = 8847360 (8.4375MB)
    0.0% used
concurrent mark-sweep generation:
  capacity = 177602560 (169.375MB)
  used      = 100680368 (96.01628112792969MB)
  free      = 76922192 (73.35871887207031MB)
  56.688579263722325% used

22863 interned Strings occupying 2224008 bytes.
```

```
D:\>jmap -histo 29684
      num      #instances      #bytes  class name
-----
      1:        581986      35263056  [C
      2:        335244      20229656  [Ljava.lang.Object;
      3:        22751      17669384  [B
      4:        267942      12861216  java.nio.HeapCharBuffer
      5:        12277      11501448  [I
      6:        261207      10448280  lombok.javac.JavacNode
      7:        280236      6725664   java.lang.String
      8:        175856      4220544   com.sun.tools.javac.util.List
      9:        260205      4163280   lombok.core.LombokImmutableList
     10:       117232      3751424   java.util.HashMap$Node
     11:       145841      3500184   java.util.ArrayList
     12:       106038      3393216   com.sun.tools.javac.code.Type$JCPrimitiveType$1
     13:        1721      3235776  [S
     14:       80034       3201360  java.util.LinkedHashMap$Entry
     15:       94201       3014432  com.sun.tools.javac.tree.JCTree$JCIdent
     16:       31551       2959216  [Ljava.util.HashMap$Node;
     17:       71599       2863960  org.codehaus.plexus.util.xml.Xpp3Dom
     18:       36241       2609352  com.sun.tools.javac.code.Symbol$MethodSymbol
     19:       31271       2001344  com.sun.tools.javac.code.Symbol$VarSymbol
     20:       62038       1985216  com.sun.tools.javac.util.SharedNameTable$NameImpl
     21:       19654       1931552  [Lcom.sun.tools.javac.code.Scope$Entry;
     22:       59185       1893920  com.sun.tools.javac.code.Type$MethodType
     23:       58398       1868736  com.sun.tools.javac.code.Scope$Entry
     24:       34887       1674576  com.sun.tools.javac.file.ZipFileIndex$Entry
     25:       50730       1623360  com.sun.tools.javac.tree.JCTree$JCLiteral
     26:       39100       1564000  com.sun.tools.javac.tree.JCTree$JCBinary
     27:       45814       1466048  com.sun.tools.javac.code.Types$ImplementationCache$Entry
     28:       15903       1399464  java.lang.reflect.Method
     29:       19061       1372392  java.lang.reflect.Field
     30:        10       1310880  [Lcom.sun.tools.javac.util.SharedNameTable$NameImpl;
     31:       11240       1258736  java.lang.Class
     32:        7509       1191536  [Lorg.jacoco.core.internal.analysis.LineImpl;
     33:       10822       1038912  org.apache.maven.model.Dependency
     34:       17997       1007832  java.util.LinkedHashMap
```

常用选项就 3 个：

-heap 打印堆内存（/内存池）的配置和使用信息。

-histo 看哪些类占用的空间最多, 直方图。

-dump:format=b,file=xxxx.hprof

Dump 堆内存。

演示:

jmap -heap pid

jmap -histo pid

jmap -dump:format=b,file=3826.hprof
3826

JVM 命令行工具--jstack

```
D:\>jstack -l 29684
2020-10-13 23:07:38
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.231-b11 mixed mode):

"Thread-692" #1306 prio=5 os_prio=0 tid=0x000000001b64c800 nid=0x325c runnable [0x000000002cae000]
  java.lang.Thread.State: RUNNABLE
    at java.io.FileInputStream.readBytes(Native Method)
    at java.io.FileInputStream.read(FileInputStream.java:255)
    at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
    at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
    at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    - locked <0x00000000c3da83e0> (a java.io.InputStreamReader)
    at java.io.InputStreamReader.read(InputStreamReader.java:184)
    at java.io.BufferedReader.fill(BufferedReader.java:161)
    at java.io.BufferedReader.readLine(BufferedReader.java:324)
    - locked <0x00000000c3da83e0> (a java.io.InputStreamReader)
    at java.io.BufferedReader.readLine(BufferedReader.java:389)
    at org.codehaus.plexus.util.cli.StreamPumper.run(StreamPumper.java:129)

Locked ownable synchronizers:
  - None

"Thread-691" #1305 prio=5 os_prio=0 tid=0x000000001b64e000 nid=0x4224 runnable [0x0000000020dfe000]
  java.lang.Thread.State: RUNNABLE
    at java.io.FileInputStream.readBytes(Native Method)
    at java.io.FileInputStream.read(FileInputStream.java:255)
    at java.io.BufferedInputStream.read1(BufferedInputStream.java:284)
    at java.io.BufferedInputStream.read(BufferedInputStream.java:345)
    - locked <0x00000000c3da3310> (a java.io.BufferedInputStream)
    at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
    at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
    at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    - locked <0x00000000c3da5798> (a java.io.InputStreamReader)
    at java.io.InputStreamReader.read(InputStreamReader.java:184)
    at java.io.BufferedReader.fill(BufferedReader.java:161)
    at java.io.BufferedReader.readLine(BufferedReader.java:324)
    - locked <0x00000000c3da5798> (a java.io.InputStreamReader)
    at java.io.BufferedReader.readLine(BufferedReader.java:389)
    at org.codehaus.plexus.util.cli.StreamPumper.run(StreamPumper.java:129)
```

-F 强制执行 thread dump , 可在 Java 进程卡死 (hung 住) 时使用 , 此选项可能需要系统权限。

-m 混合模式 (mixed mode) , 将 Java 帧和 native 帧一起输出 , 此选项可能需要系统权限。

-l 长列表模式 , 将线程相关的 locks 信息一起输出 , 比如持有的锁 , 等待的锁。

演示 :

jstack -l pid

JVM 命令行工具--jcmand*

```
D:\>jcmand 29684 help
29684:
The following commands are available:
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.native_memory
VM.check_commercial_features
VM.unlock_commercial_features
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
VM.classloader_stats
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.finalizer_info
GC.heap_info
GC.run_finalization
GC.run
VM.uptime
VM.dynlibs
VM.flags
VM.system_properties
VM.command_line
VM.version
help
```

Jcmand 综合了前面的几个命令

示例：

jcmand pid VM.version

jcmand pid VM.flags

jcmand pid VM.command_line

jcmand pid VM.system_properties

jcmand pid Thread.print

jcmand pid GC.class_histogram

jcmand pid GC.heap_info

JVM 命令行工具--jrunscript/jjs

```
D:\>jjs
jjs> a=1
1
jjs> b=a+1
2
jjs> function c(x,y){return x*y;}
function c(x,y){return x*y;}
jjs>
jjs> c(a+10,b)
22
jjs> |
```

```
D:\>jrunscript -e "cat('http://www.baidu.com')"
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html; charset=utf-8><meta http-equiv=X-UA-Compatible content=IE=Edge><meta content=always name=referrer><link rel=stylesheet type=text/css href=http://s1.bdstatic.com/r/www/cache/bdorz/baidu.min.css><title>網惧害涓?涓嬪紝浣犲氨鑱?亾</title></head> <body link=#0000cc> <div id=wrapper> <div id=head> <div class=head_wrapper> <div class=s_form> <div class=s_form_wrapper> <div id=lg> <img hidefocus=true src=/www.baidu.com/img/bd_logo1.png width=270 height=129> </div> <form id=form name=f action=/www.baidu.com/s class=fm> <input type=hidden name=bdorz_come value=1> <input type=hidden name=ie value=utf-8> <input type=hidden name=f value=8> <input type=hidden name=rsv_bp value=1> <input type=hidden name=rsv_idx value=1> <input type=hidden name=tn value=baidu><span class="bg s_ipt_wr"><input id=kw name=wd class=s_ipt value maxlength=255 autocomplete=off autofocus></span><span class="bg s_btn_wr"><input type=submit id=su value=網惧害涓?涓? class="bg s_btn"></span> </form> </div> </div> <div id=u1> <a href=http://news.baidu.com name=tj_trnews class=mnav>鍚伴櫔</a> <a href=http://www.hao123.com name=tj_trhao123 class=mnav>hao123</a> <a href=http://map.baidu.com name=tj_trmap class=mnav>鏟板澠</a> <a href=http://v.baidu.com name=tj_trvideo class=mnav>瑙嚟?</a> <a href=http://tieba.baidu.com name=tj_trtieba class=mnav>瑙村愓</a> <noscript> <a href=http://www.baidu.com/bdorz/login.gif?login&tpl=mn&u=http%3A%2F%2Fwww.baidu.com%2F%3fbdorz_come%3d1 name=tj_login class=lb>網海綽</a> </noscript> <script>document.write('<a href="http://www.baidu.com/bdorz/Login.gif?login&tpl=mn&u='+ encodeURIComponent(window.location.href+ (window.location.search == "" ? "?" : "&")+ "bdorz_come=1")+ '' name="tj_login" class="lb">網海綽</a>');</script> <a href=/www.baidu.com/more/ name=tj_bricon class=bri style="display: block; ">瑙村?浜y擴</a> </div> </div> <div id=ftCon> <div id=ftConw> <p id=lh> <a href=http://home.baidu.com>鍚充箇網惧害</a> <a href=http://ir.baidu.com>About Baidu</a> </p> <p id=cp>&copy;2017&nbsp;Baidu&nbsp;<a href=http://www.baidu.com/duty/>浣跨粃網惧害鍚?</a>&nbsp; <a href=http://jianyi.baidu.com/ class=cp-feedback>鑑他?鑿確?</a>&nbsp;浜璇CP璇?030173鑿?&nbsp; <img src=/www.baidu.com/img/gs.gif> </p> </div> </div> </body> </html>
```

当 curl 命令用：

```
jrunscript -e "cat('http://www.baidu.com')"
```

执行 js 脚本片段

```
jrunscript -e "print('hello,kk.jvm'+1)"
```

执行 js 文件

```
jrunscript -l js -f /XXX/XXX/test.js
```

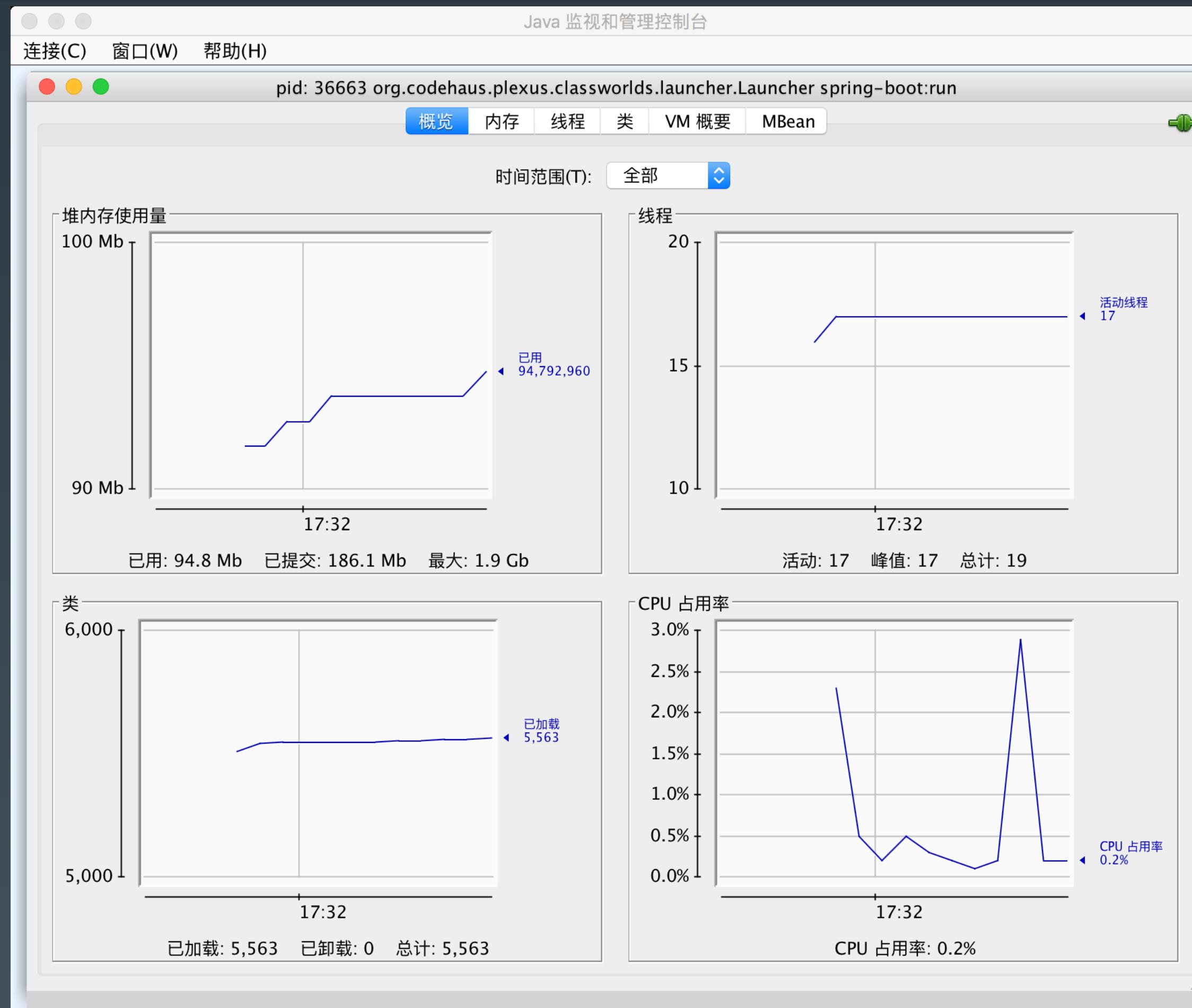
2. JDK 内置图形化工具

JVM 图形化工具--jconsole



在命令行输入 jconsole 即可打开
本地 JVM 可以直接选择
远程 JVM 可以通过 JMX 方式连接

JVM 图形化工具--jconsole



共有 6 个面板

第一个为概览，四项指标具体为：

堆内存使用量：此处展示的就是前面 Java 内存模型课程中提到的堆内存使用情况，从图上可以看到，堆内存使用了 94MB 左右，并且一直在增长。

线程：展示了 JVM 中活动线程的数量，当前时刻共有 17 个活动线程。

类：JVM 一共加载了 5563 个类，没有卸载类。

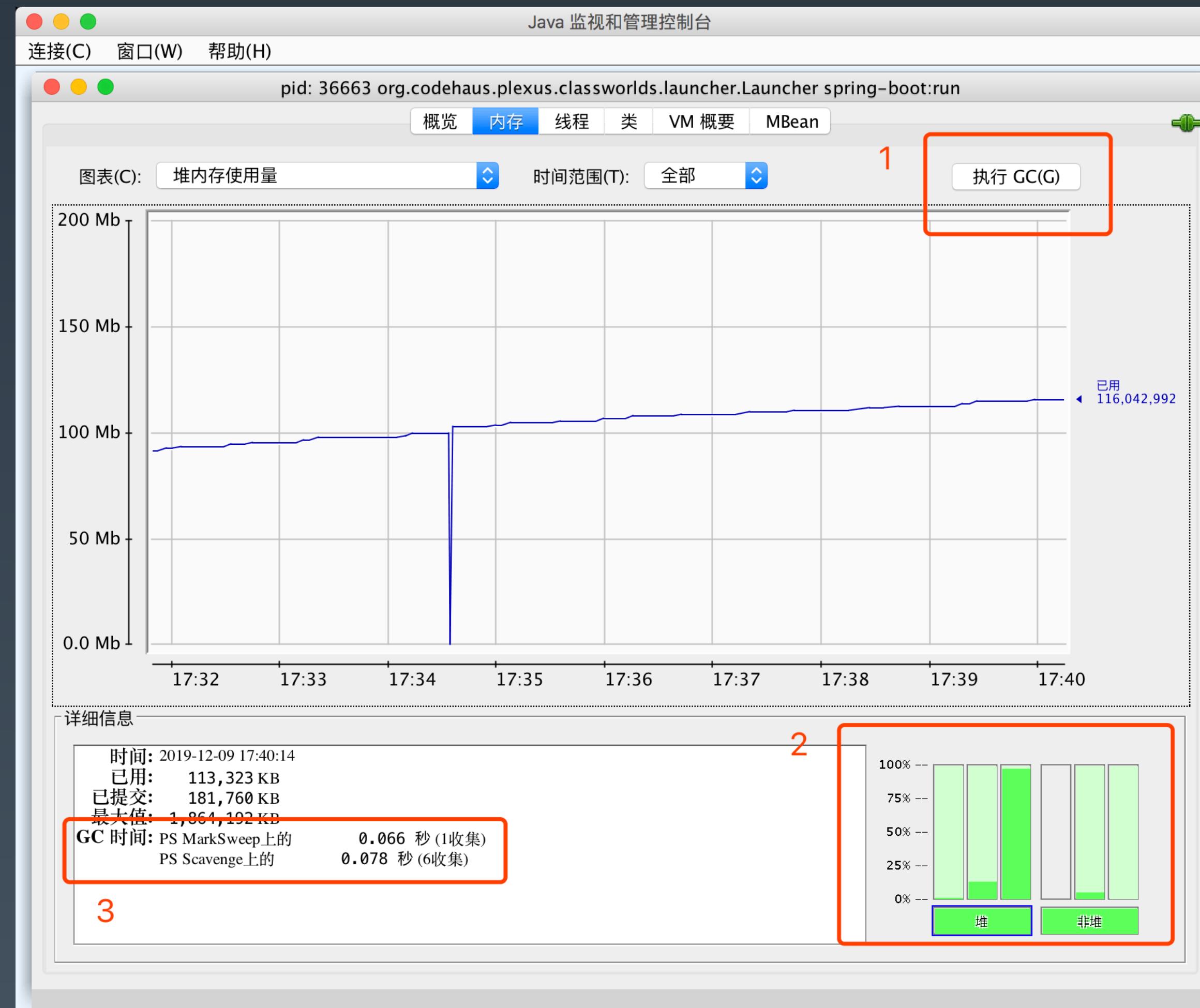
CPU 占用率：目前 CPU 使用率为 0.2%，这个数值非常低，且最高的时候也不到 3%，初步判断系统当前并没有什么负载和压力。

有如下几个时间维度可供选择：

1分钟、5分钟、10分钟、30分钟、1小时、2小时、3小时、6小时、12小时、1天、7天、1个月、3个月、6个月、1年、全部，一共是16档。

当我们想关注最近1小时或者1分钟的数据，就可以选择对应的档。旁边的3个标签页(内存、线程、类)，也都支持选择时间范围。

JVM 图形化工具--jconsole



内存图表包括:

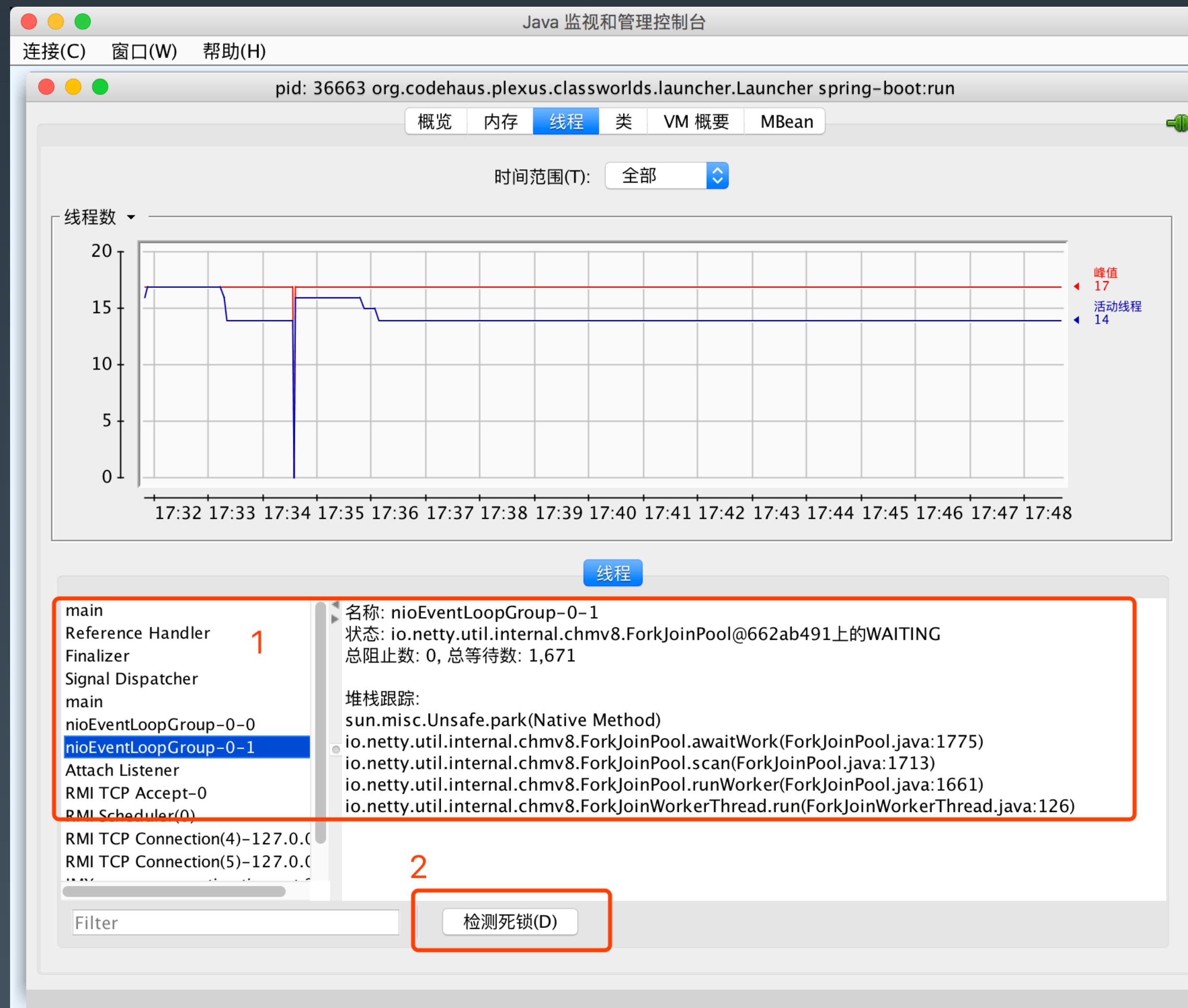
- 堆内存使用量，主要包括老年代（内存池“PS Old Gen”）、新生代（“PS Eden Space”）、存活区（“PS Survivor Space”）；
- 非堆内存使用量，主要包括内存池“Metaspace”、“Code Cache”、“Compressed Class Space”等。

可以分别选择对应的 6 个内存池。

通过内存面板，我们可以看到各个区域的内存使用和变化情况，并且可以：

1. 手动执行 gc，见图上的标号1，点击按钮即可执行 JDK 中的 System.gc()
2. 通过图中右下角标号 2 的界面，可以看到各个内存池的百分比使用率，以及堆/非堆空间的汇总使用情况。
3. 从左下角标号 3 的界面，可以看到 JVM 使用的垃圾收集器，以及执行垃圾收集的次数，以及相应的时间消耗。

JVM 图形化工具--jconsole

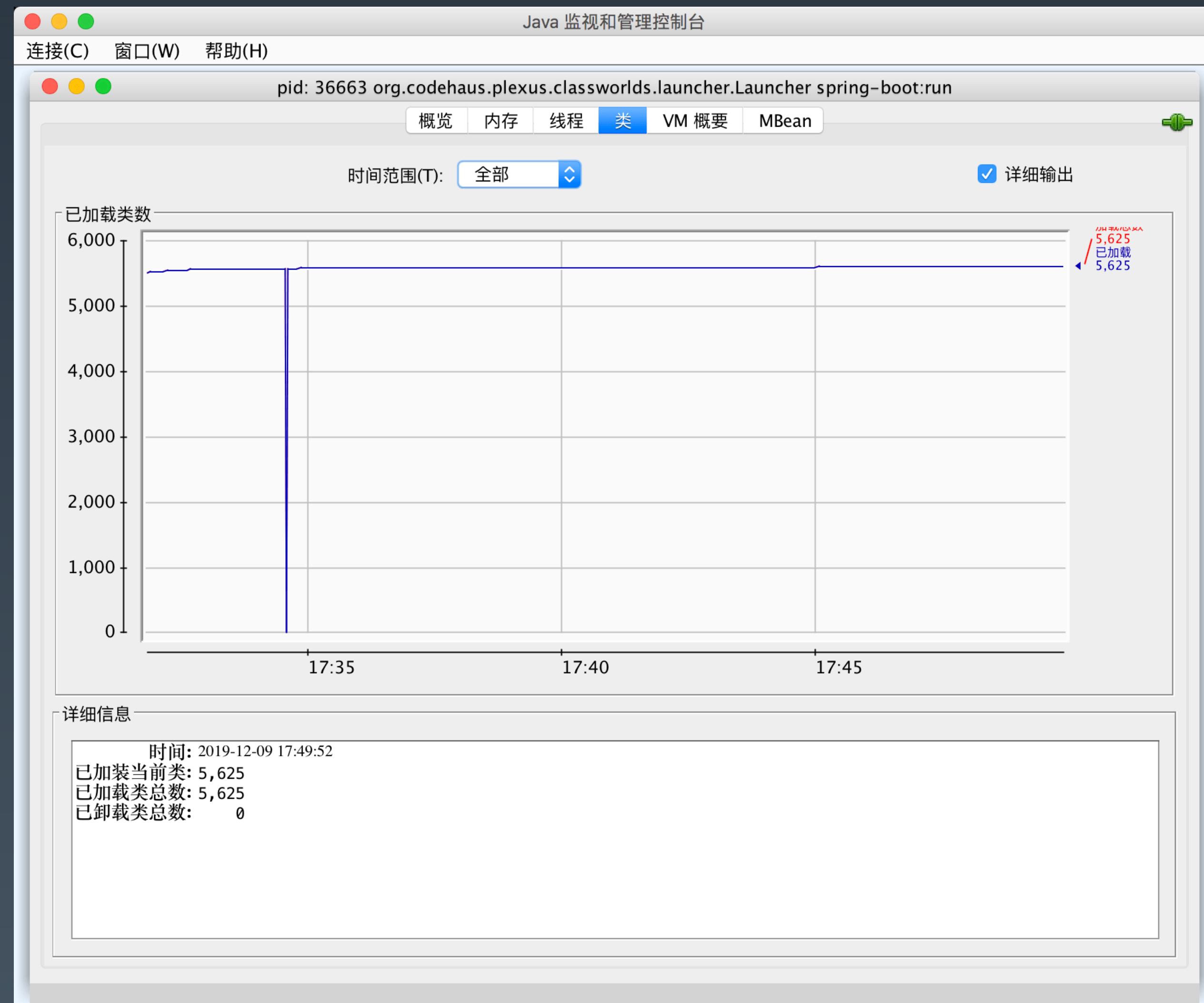


线程面板展示了线程数变化信息，以及监测到的线程列表。

我们可以常根据名称直接查看线程的状态（运行还是等待中）和调用栈（正在执行什么操作）。

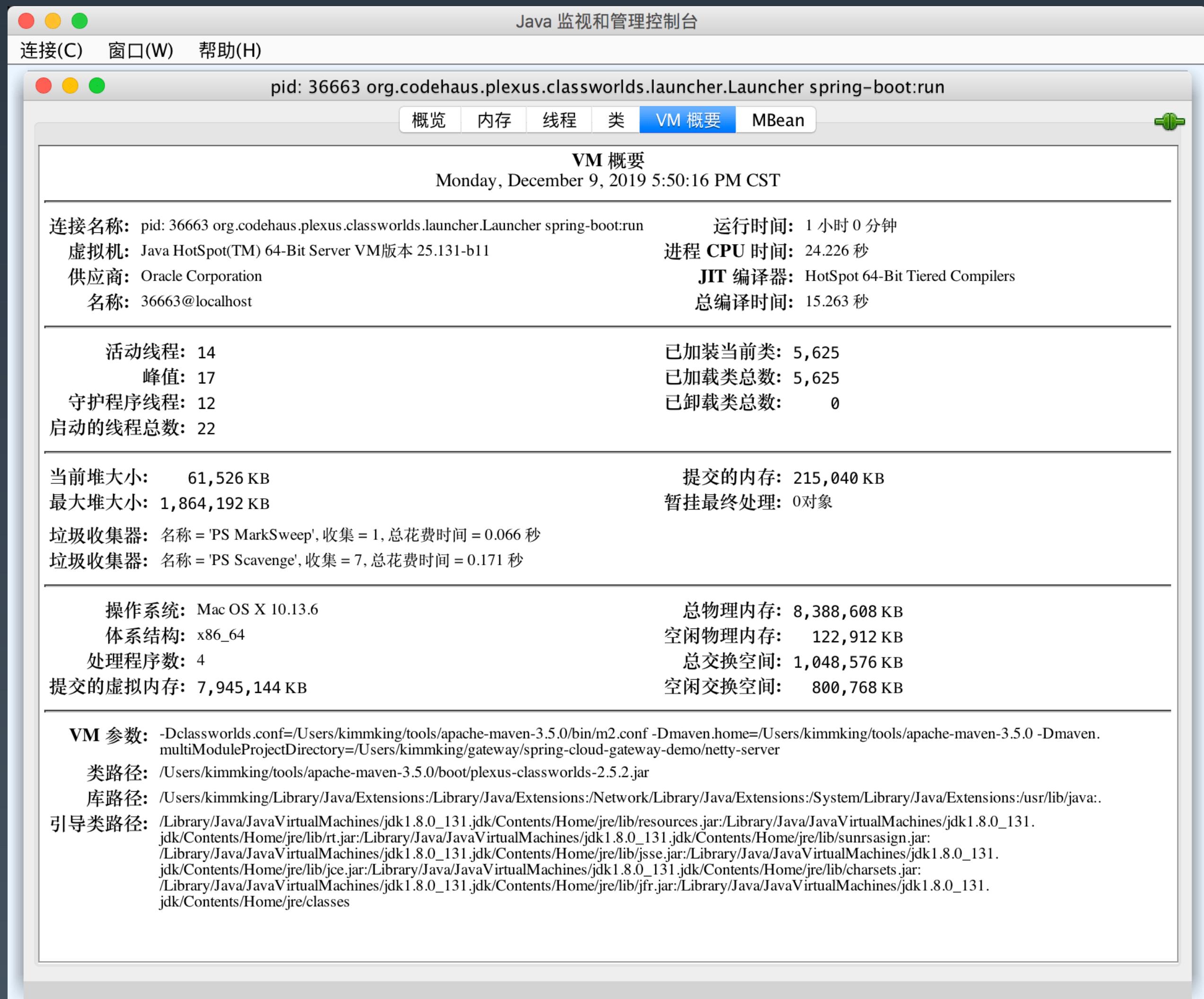
特别地，我们还可以直接点击“检测死锁”按钮来检测死锁，如果没有死锁则会提示“未检测到死锁”。

JVM 图形化工具--jconsole



类监控面板，可以直接看到 JVM 加载和卸载的类数量汇总信息，以及随着时间的动态变化。

JVM 图形化工具--jconsole



VM 概要的数据有五个部分:

第一部分是虚拟机的信息；

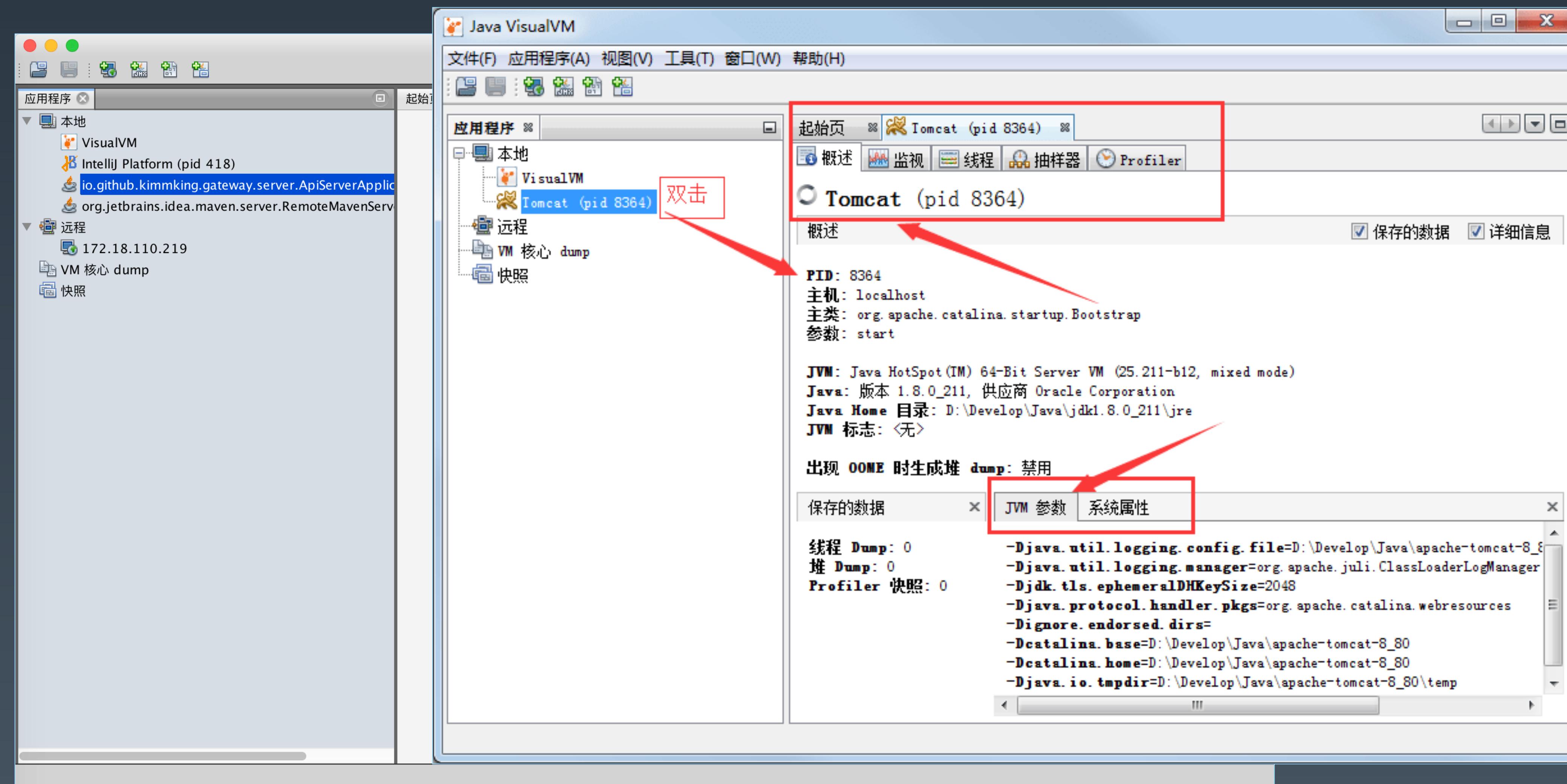
第二部分是线程数量，以及类加载的汇总信息；

第三部分是堆内存和 GC 统计；

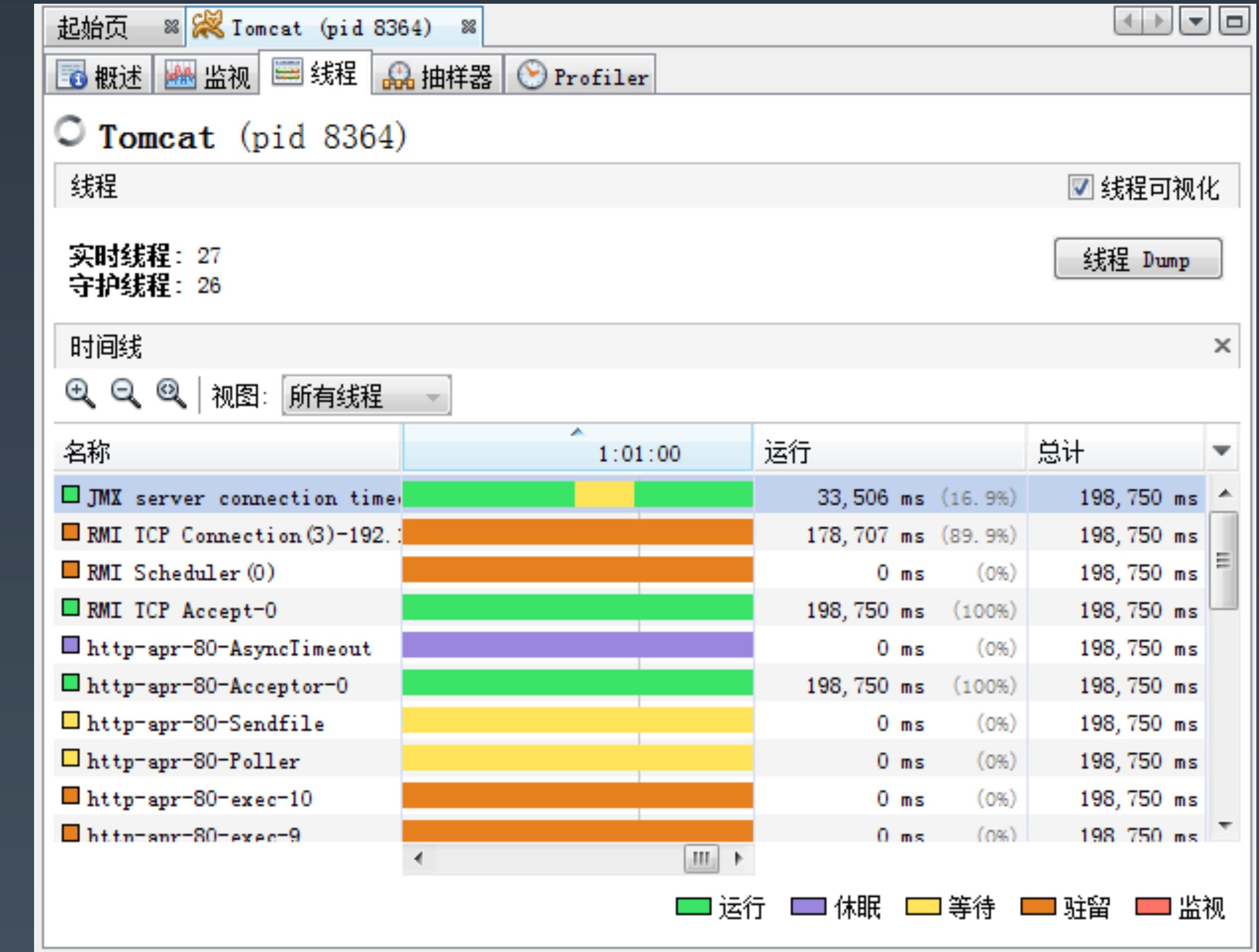
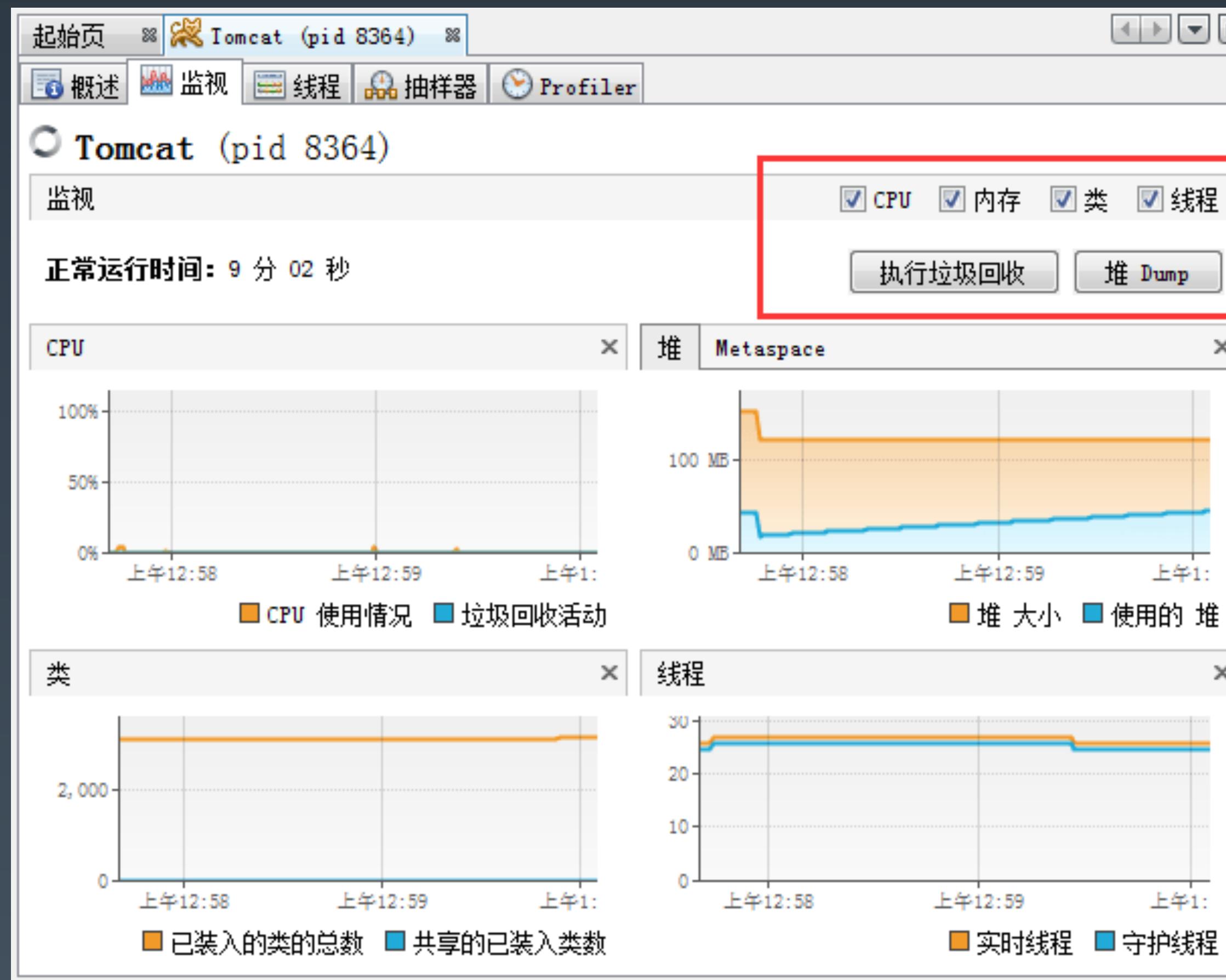
第四部分是操作系统和宿主机的设备信息，比如 CPU 数量、物理内存、虚拟内存等等；

第五部分是 JVM 启动参数和几个关键路径，这些信息其实跟 jinfo 命令看到的差不多。

JVM 图形化工具--jvisualvm



JVM 图形化工具--jvisualvm



JVM 图形化工具--jvisualvm

The image shows two side-by-side jVisualVM windows for the application `io.github.kimmking.gateway.server.ApiServerApplication` (pid 63434).

Left Window (CPU Sampling):

- Sampling:** CPU
- Status:** 正在进行 CPU 抽样 (Sampling CPU)
- Threads:** 43, 总的 CPU 时间 [毫秒]: 168,512
- Table:** Shows thread names and their CPU usage. The top thread is `RMI TCP Connection(idle)` at 6,821.751 ms (0.0%).

线程名称	线程 CPU 时间 [%]	线程 CPU 时间 [毫秒]	线程 CPU 时间 [毫秒]/秒
RMI TCP Connection(idle)	6,821.751 (0.0%)	6,821.751	0
http-nio-8088-exec-21	6,380.229 (0.0%)	6,380.229	0
http-nio-8088-exec-20	6,181.009 (0.0%)	6,181.009	0
http-nio-8088-exec-9	6,117.711 (0.0%)	6,117.711	0
http-nio-8088-exec-10	6,071.691 (0.0%)	6,071.691	0
http-nio-8088-exec-5	5,959.476 (0.0%)	5,959.476	0
http-nio-8088-exec-12	5,935.091 (0.0%)	5,935.091	0
http-nio-8088-exec-6	5,808.818 (0.0%)	5,808.818	0
http-nio-8088-exec-4	5,782.109 (0.0%)	5,782.109	0
http-nio-8088-exec-3	5,778.329 (0.0%)	5,778.329	0
http-nio-8088-exec-22	5,776.163 (0.0%)	5,776.163	0
http-nio-8088-exec-8	5,754.528 (0.0%)	5,754.528	0
http-nio-8088-exec-13	5,749.318 (0.0%)	5,749.318	0
http-nio-8088-exec-11	5,714.689 (0.0%)	5,714.689	0
http-nio-8088-exec-7	5,679.325 (0.0%)	5,679.325	0
http-nio-8088-exec-18	5,676.343 (0.0%)	5,676.343	0

Right Window (Memory Sampling):

- Sampling:** 内存
- Status:** 正进行内存抽样 (Sampling Memory)
- 类:** 3,078, 实例: 1,545,391, 字节: 84,220,552
- Table:** Shows the top memory-consuming classes. `byte[]` is the largest at 20,577,088 bytes (24.4%).

类名	字节 [%]	字节	实例数
<code>byte[]</code>	24.4%	20,577,088	41,145 (2.6%)
<code>char[]</code>	18.4%	15,545,864	184,875 (11.9%)
<code>int[]</code>	4.3%	3,705,016	15,968 (1.0%)
<code>java.lang.Object[]</code>	4.3%	3,679,864	104,475 (6.7%)
<code>java.lang.String</code>	3.4%	2,940,264	122,511 (7.9%)
<code>java.util.HashMap\$Node[]</code>	2.6%	2,226,648	58,191 (3.7%)
<code>java.util.HashMap</code>	2.4%	2,084,784	43,433 (2.8%)
<code>java.util.LinkedHashMap</code>	2.1%	1,814,064	32,394 (2.0%)
<code>java.util.concurrent.ConcurrentHashMap\$Node</code>	1.6%	1,422,272	44,446 (2.8%)
<code>java.lang.Class[]</code>	1.6%	1,417,704	79,219 (5.1%)
<code>java.lang.reflect.Method</code>	1.5%	1,311,992	14,909 (0.9%)
<code>java.util.LinkedHashMap\$Entry</code>	1.4%	1,260,400	31,510 (2.0%)
<code>java.util.ArrayList\$Itr</code>	1.3%	1,170,080	36,565 (2.3%)
<code>java.util.HashMap\$Node</code>	1.3%	1,150,304	35,947 (2.3%)
<code>java.util.concurrent.ConcurrentHashMap</code>	1.0%	910,656	14,229 (0.9%)
<code>java.util.ArrayList</code>	0.3%	884,976	36,874 (2.3%)

JVM 图形化工具--jvisualvm

The image shows two side-by-side JVisualVM windows, both titled "io.github.kimmking.gateway.server.ApiServerApplication (pid 63434)".

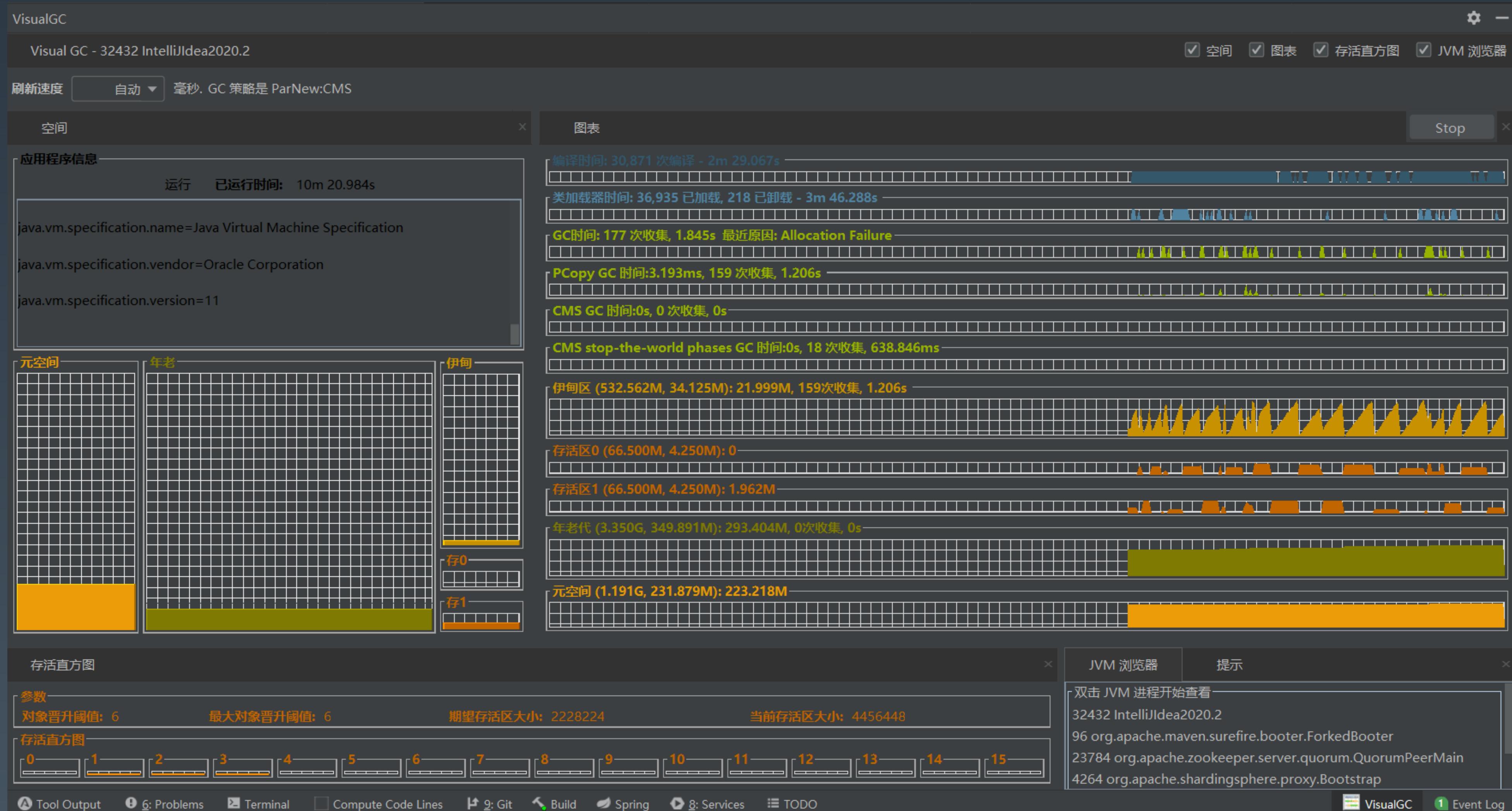
Left Window (CPU Profiler):

- 性能分析:** CPU (selected), 内存, 停止
- 状态:** 正在运行性能分析 (已分析 1,443 个方法)
- 性能分析结果:** A table showing method analysis results. The first few rows include:
 - org.apache.tomcat.util.net.SocketProces: 255,1... (46.1%)
 - org.apache.coyote.http11.Http11Process: 143,0... (25.9%)
 - org.apache.catalina.connector.CoyoteAd: 104,4... (18.9%)
 - org.apache.coyote.AbstractProtocol\$Con: 33,57... (6.1%)
 - org.apache.catalina.core.ApplicationFilte: 5,205 ... (0.9%)
- 从类启动性能分析(S):** A list of classes: io.github.kimmking.gateway.server.**
- 设置:** CPU 设置, 内存设置

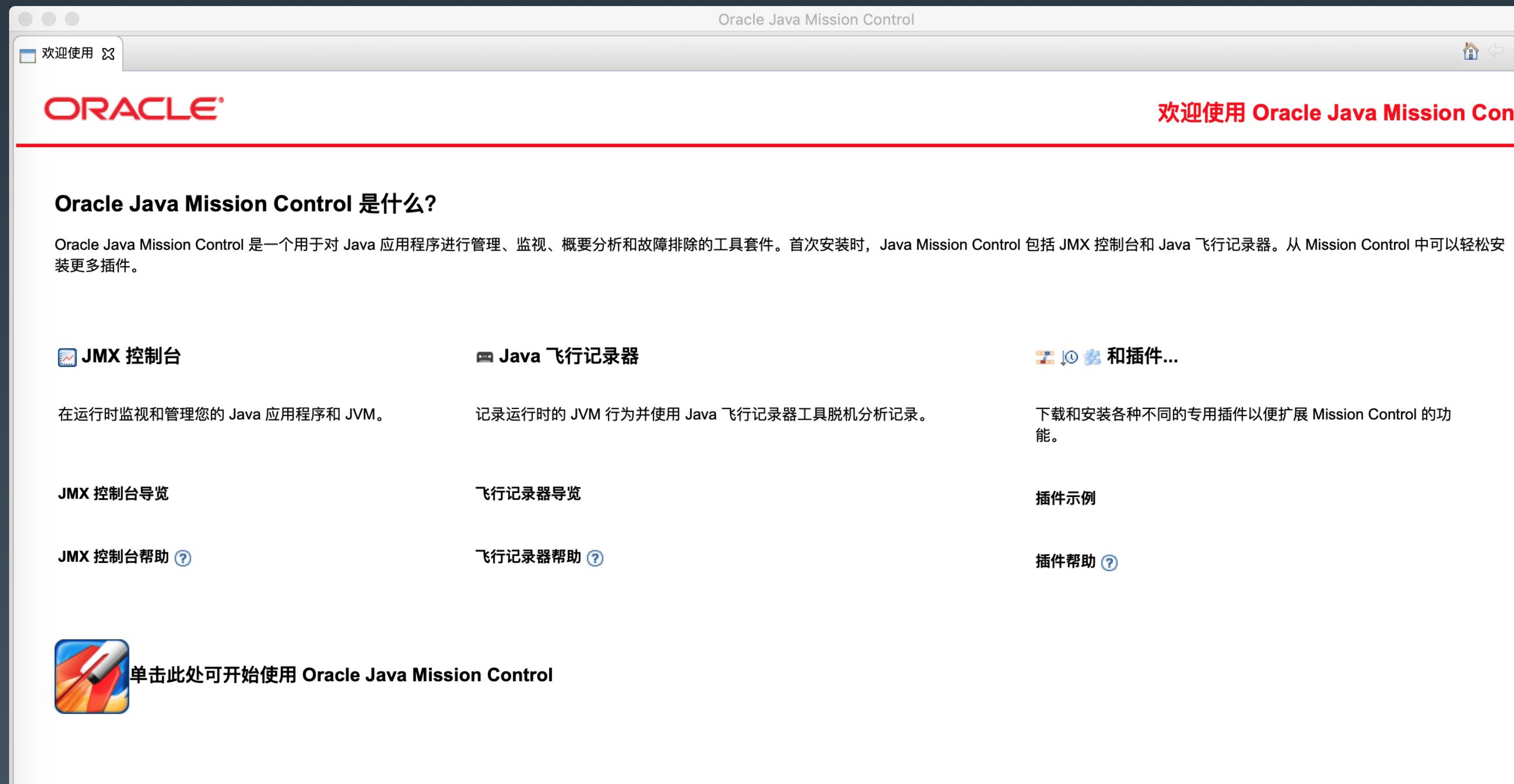
Right Window (Memory Profiler):

- 性能分析:** CPU, 内存 (selected), 停止
- 状态:** 正在运行性能分析 (已分析 6,594 个类, 正在按每 10 个对象的间隔进行跟踪)
- 性能分析结果:** A table showing class analysis results. The first few rows include:
 - byte[]: 活动字节 [%(30.2%)]
 - char[]: 活动字节 [%(14.2%)]
 - java.lang.Object[]: 活动字节 [%(4.6%)]
 - java.util.HashMap\$Node[]: 活动字节 [%(3.9%)]
 - java.lang.String: 活动字节 [%(3.7%)]
- 设置:** CPU 设置, 内存设置
- 分析对象分配:** (P) 分析对象分配 (selected), (O) 分析对象分配和 GC (未选)
- 全部跟踪(A):** 10 对象分配
- 记录分配栈跟踪(E):** (未选)

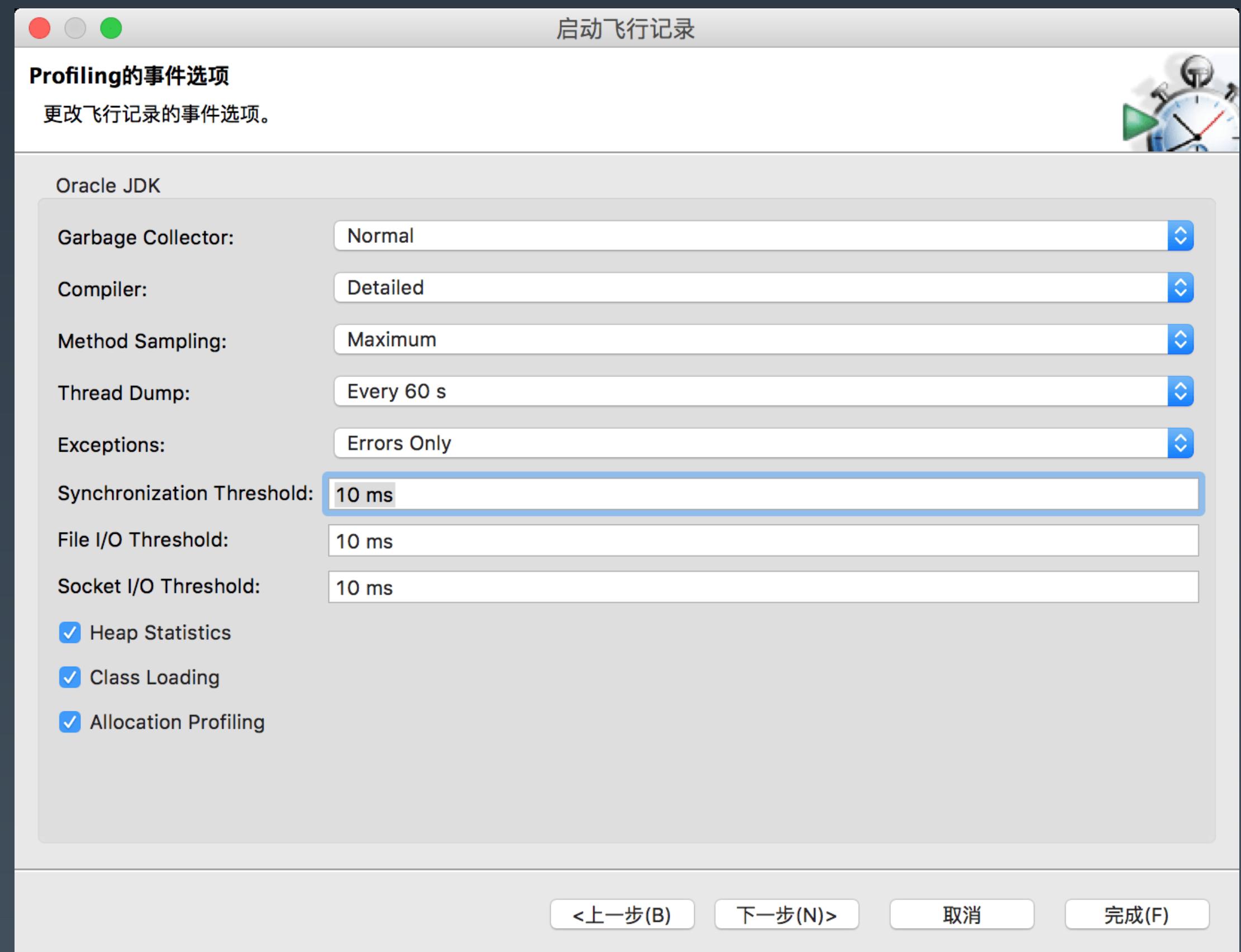
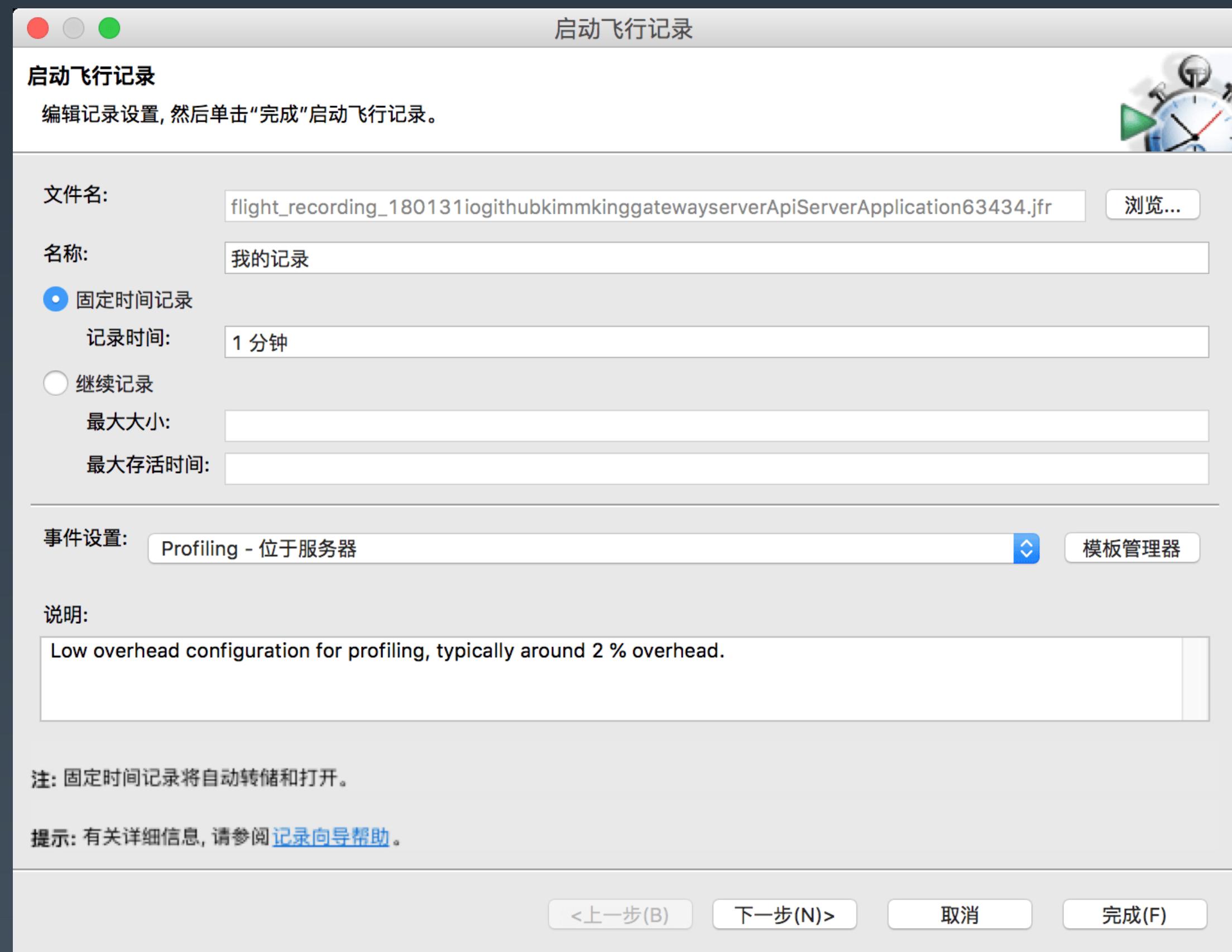
JVM 图形化工具--VisualGC



JVM 图形化工具--jmc



JVM 图形化工具--jmc



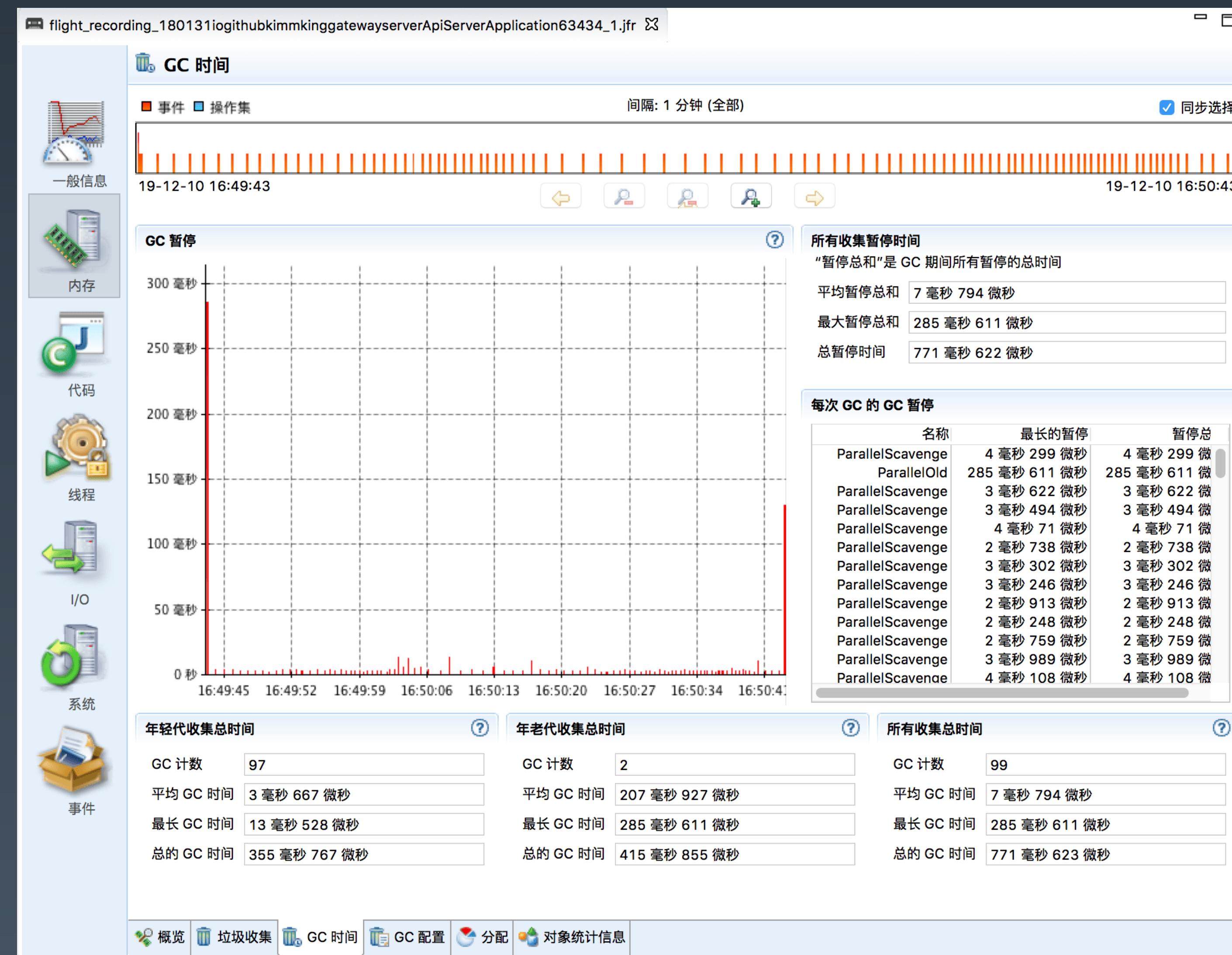
JVM 图形化工具--jmc



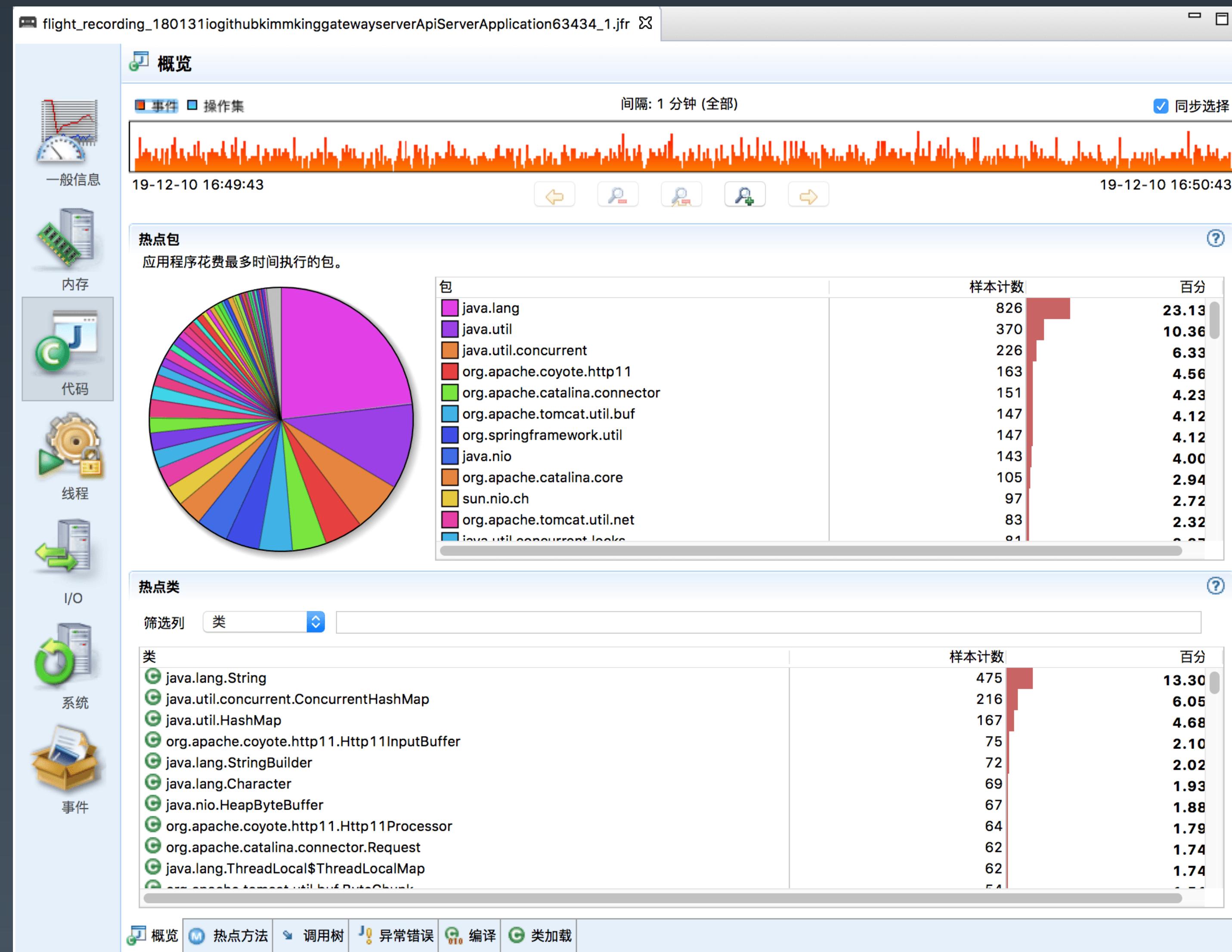
JVM 图形化工具--jmc



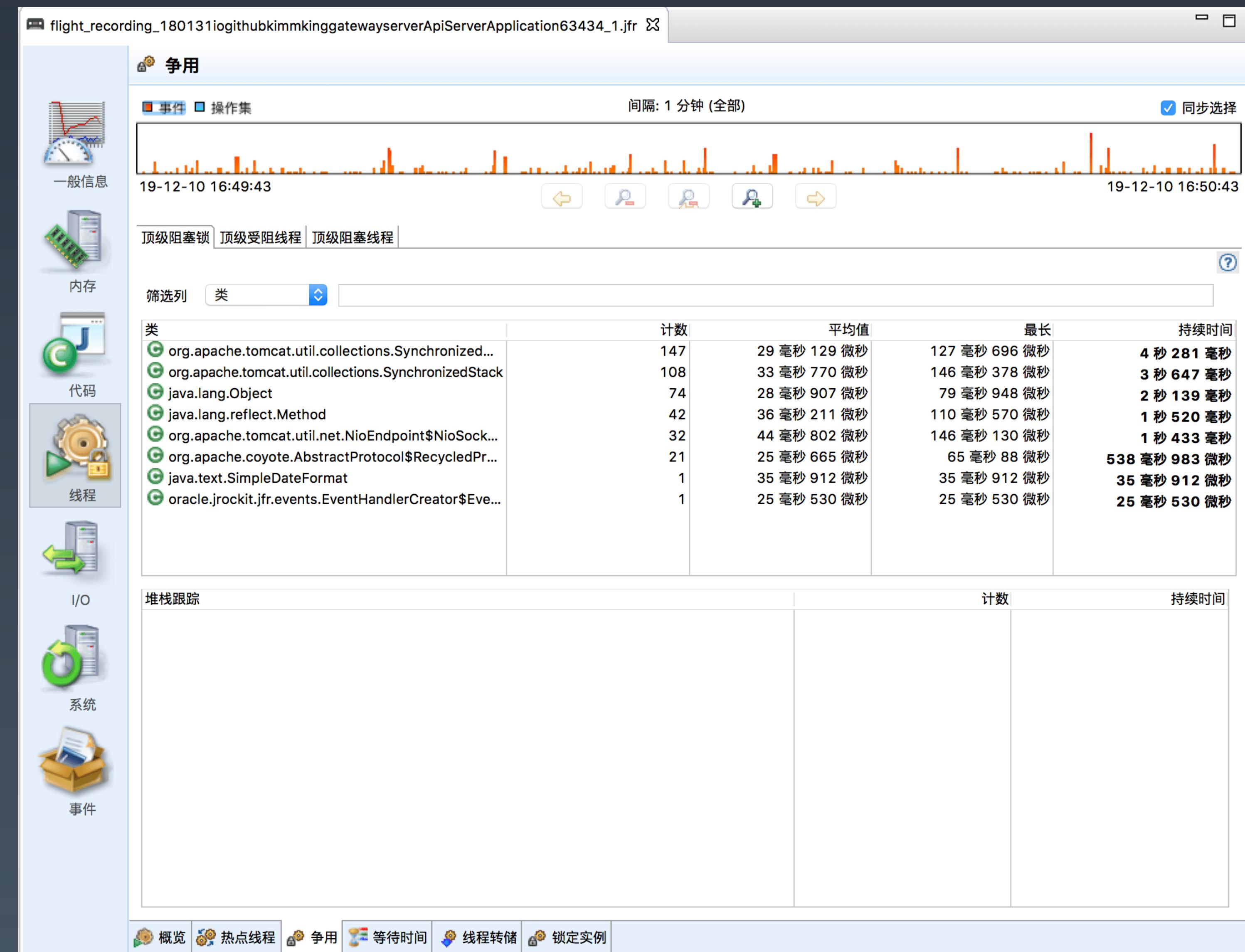
JVM 图形化工具--jmc



JVM 图形化工具--jmc



JVM 图形化工具--jmc



JVM 工具总结

jps/jinfo

jstat

jmap

jstack

jcmd

jrunscript/jjs

jsonsole

jvisualvm

visualGC --> idea

jmc

3. GC 的背景与一般原理

GC 的背景与一般原理

为什么会有 GC



本质上是内存资源的有限性

因此需要大家共享使用，手工申请，手动释放。

下面我们举个例子

GC 的背景与一般原理

引用计数



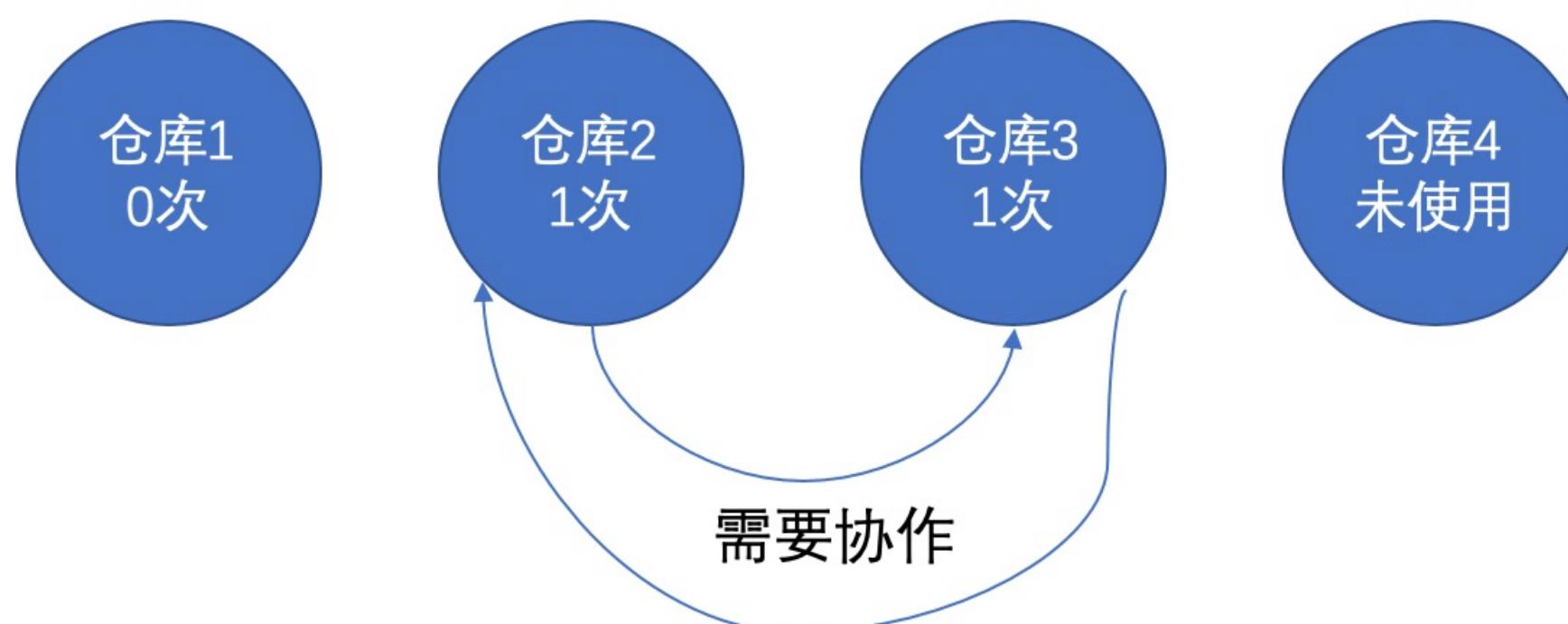
仓库与引用计数：计数为 0

简单粗暴，一般有效

实际情况复杂一点

仓库与仓库之间也有关系

引用计数-循环依赖



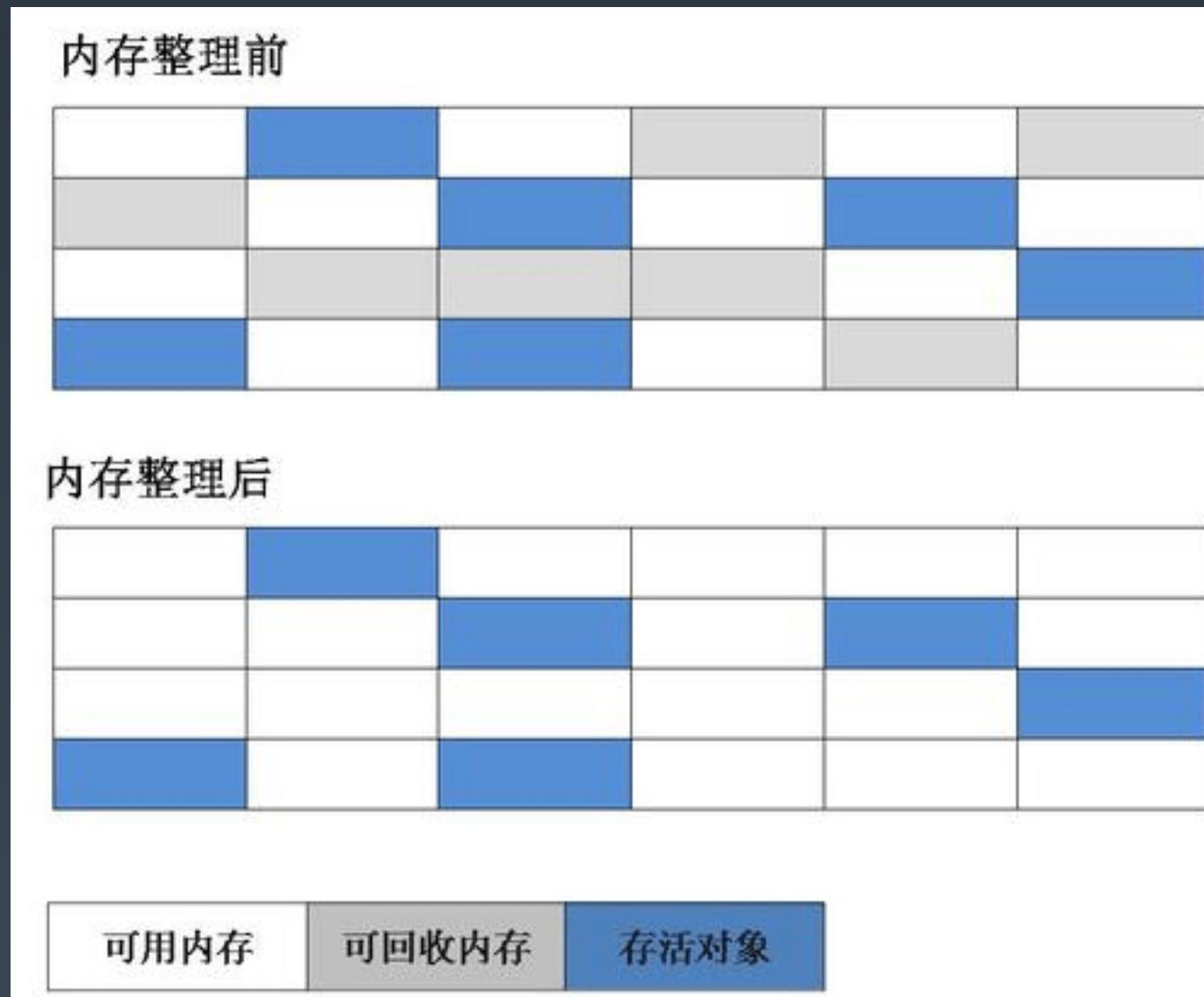
导致形成一个环，大家的计数永远不为 0

（跟线程、事务死锁一个原理）

这些仓库都没法再用：内存泄漏->内存溢出

怎么改进呢？引用计数->引用跟踪

GC 的背景与一般原理

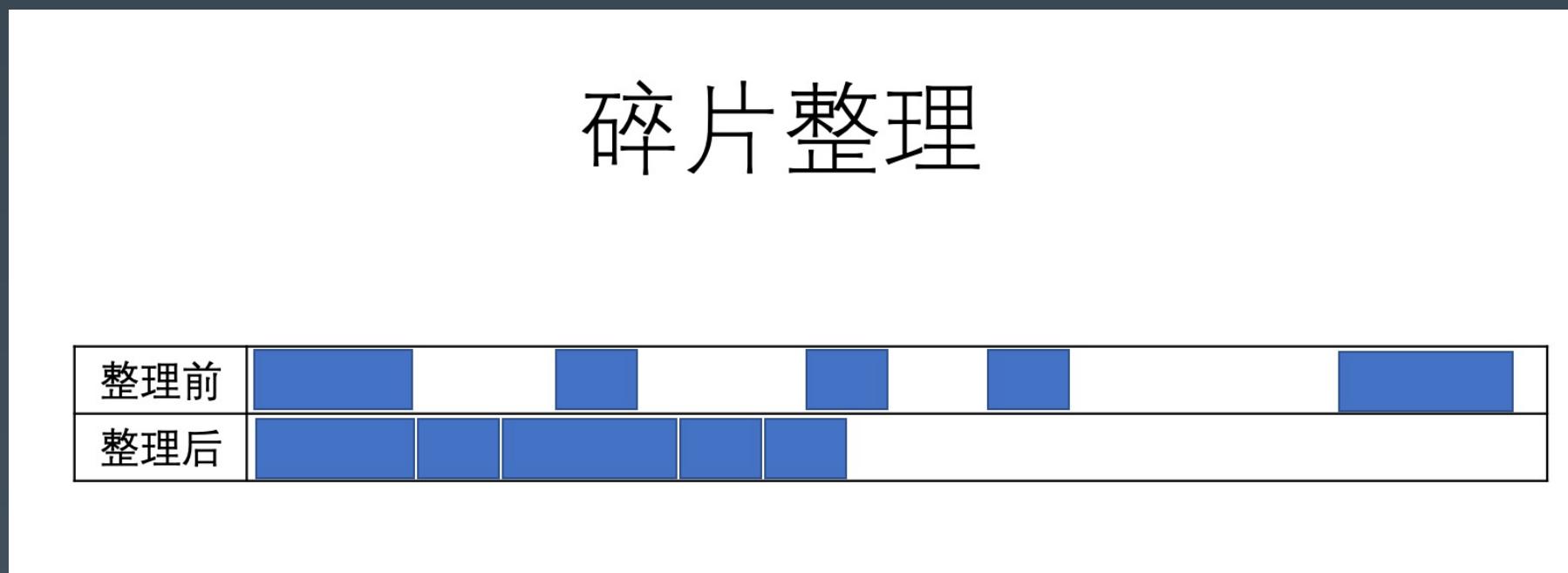


标记清除算法 (Mark and Sweep)

- Marking (标记) : 遍历所有的可达对象，并在本地内存(native)中分门别类记下。
- Sweeping (清除) : 这一步保证了，不可达对象所占用的内存，在之后进行内存分配时可以重用。

并行 GC 和 CMS 的基本原理

优势：可以处理循环依赖，只扫描部分对象

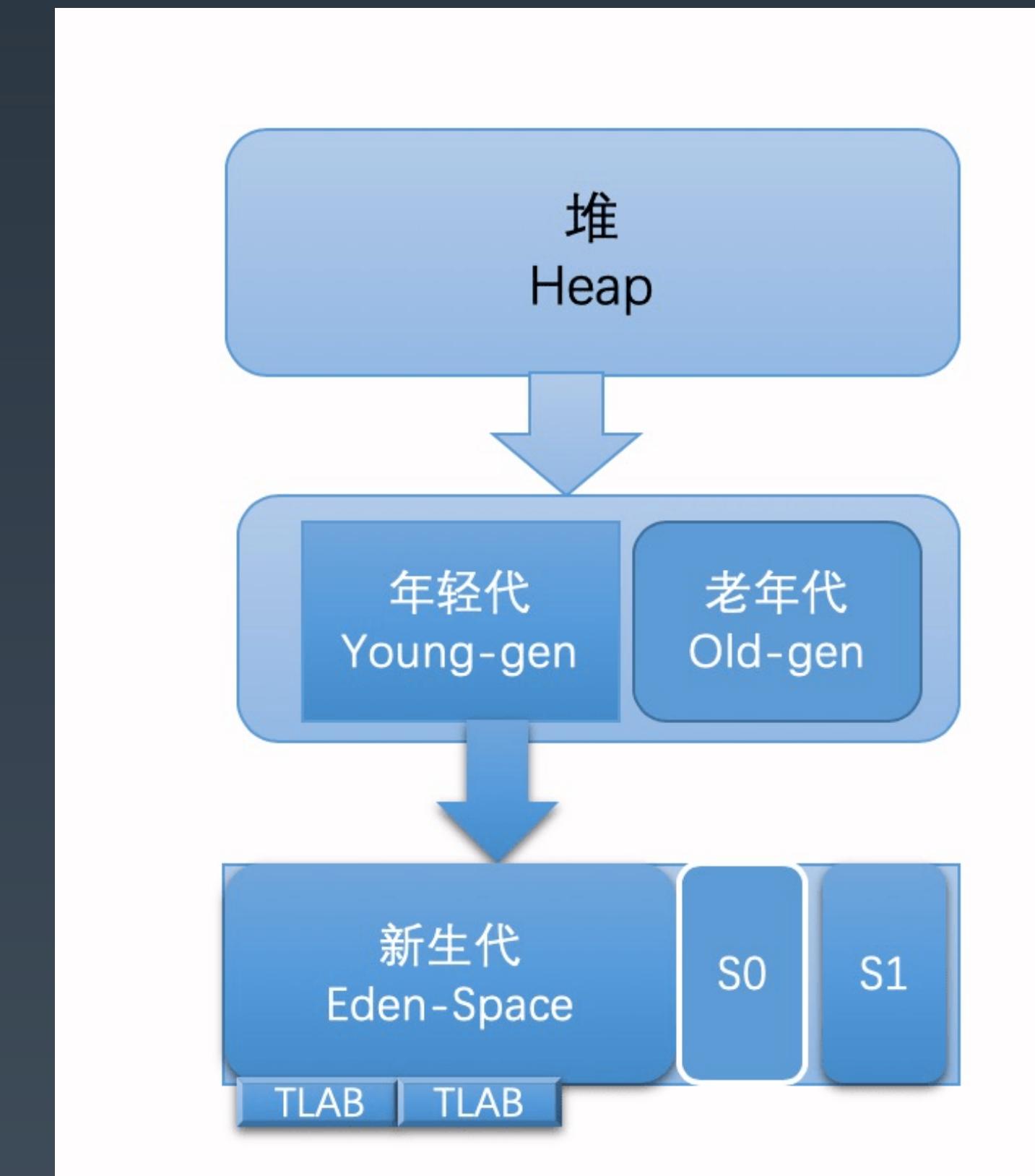
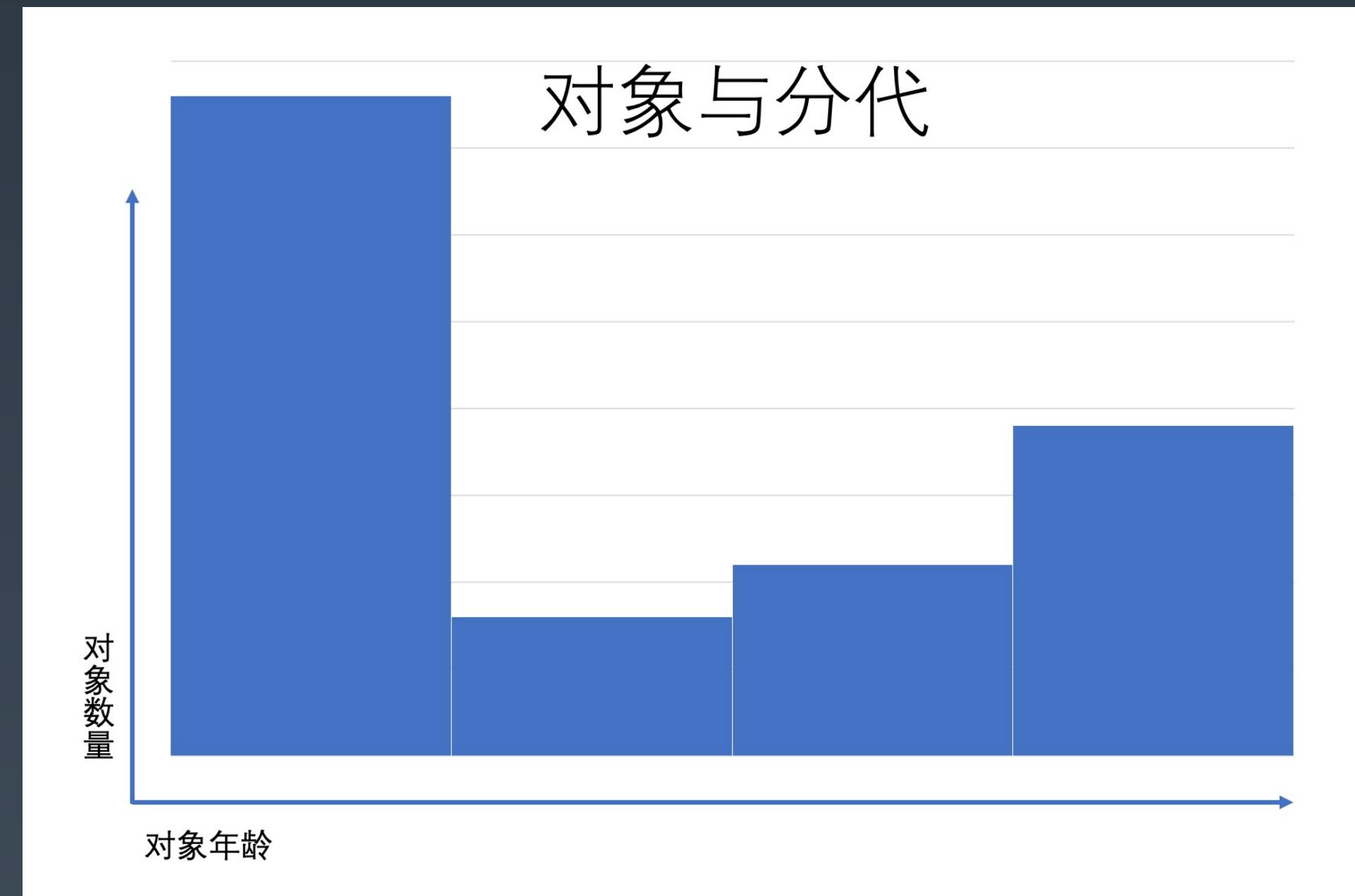


除了清除，还要做压缩。

怎么才能标记和清除清楚上百万对象呢？

答案就是 STW，让全世界停下来。

GC 的背景与一般原理

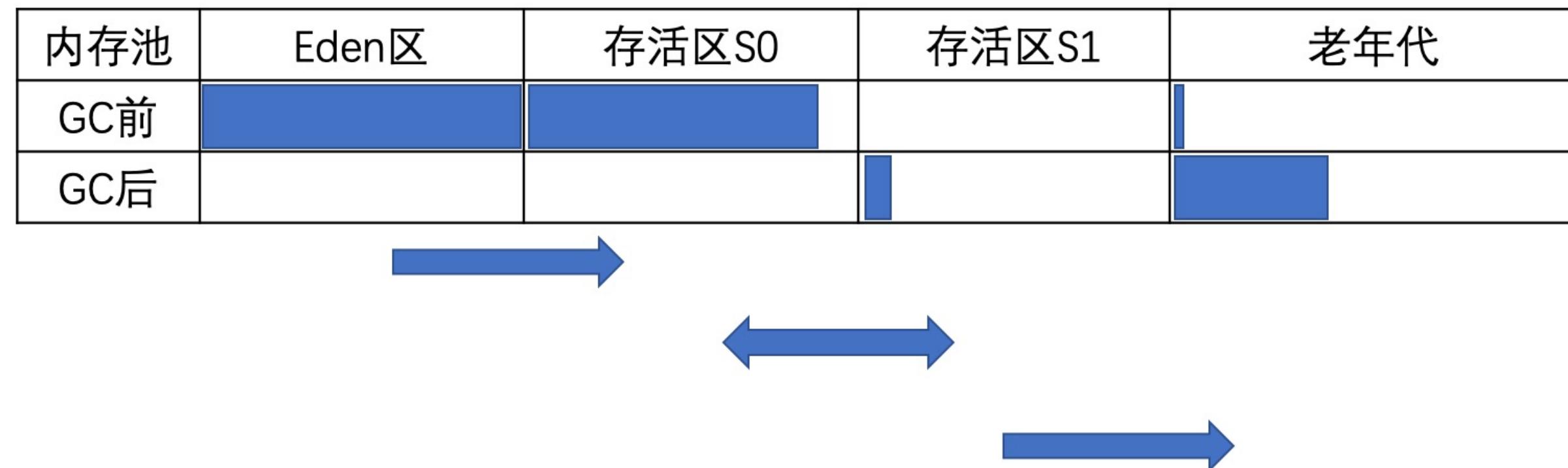


分代假设：大部分新生对象很快无用；
存活较长时间的对象，可能存活更长时间。

内存池划分
不同类型对象不同区域，不同策略处理。

GC 的背景与一般原理

GC时对象在内存池之间转移



对象分配在新生代的 Eden 区，
标记阶段 Eden 区存活的对象就会复制到存活区；
注意：为什么是复制，不是移动？？？大家想想
两个存活区 from 和 to，互换角色。对象存活到一定周期会提升到老年代。

由如下参数控制提升阈值

-XX : +MaxTenuringThreshold=15

老年代默认都是存活对象，采用移动方式：

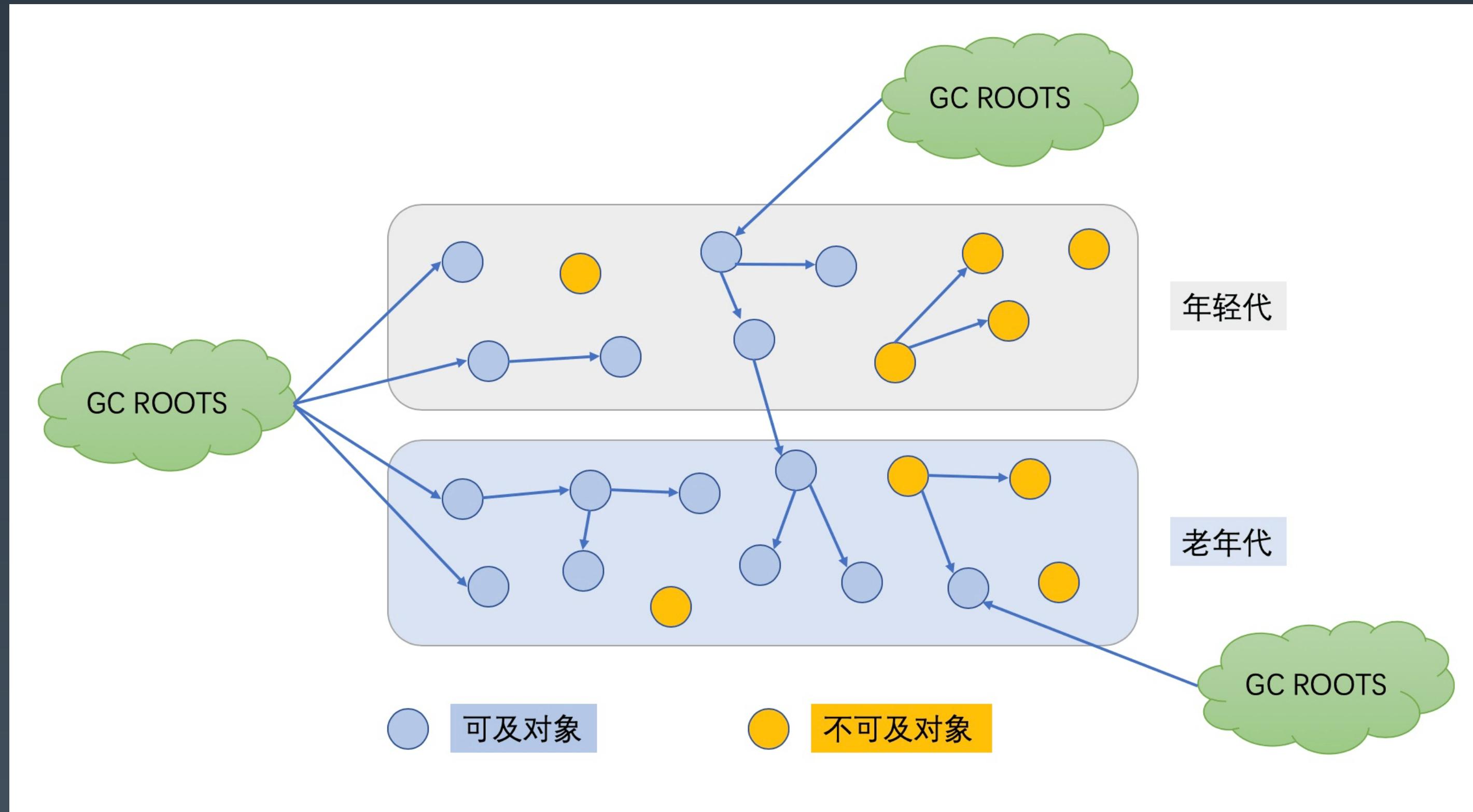
1. 标记所有通过 GC roots 可达的对象；
2. 删除所有不可达对象；
3. 整理老年代空间中的内容，方法是将所有的存活对象复制，从老年代空间开始的地方依次存放。

持久代/元数据区

1.8 之前 -XX:MaxPermSize=256m

1.8 之后 -XX:MaxMetaspaceSize=256m

GC 的背景与一般原理



可以作为 GC Roots 的对象

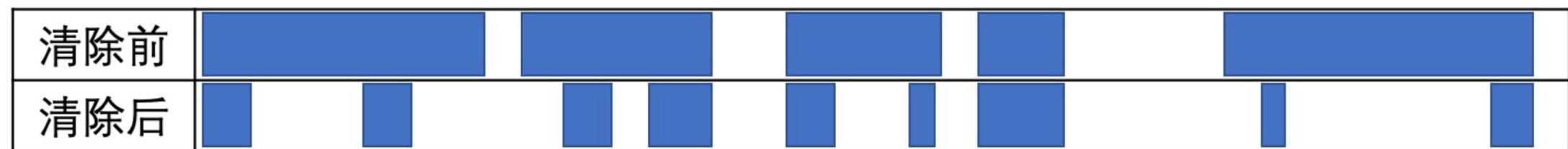
1. 当前正在执行的方法里的局部变量和输入参数
2. 活动线程 (Active threads)
3. 所有类的静态字段 (static field)
4. JNI 引用

此阶段暂停的时间，与堆内存大小，对象的总数没有直接关系，而是由存活对象 (alive objects) 的数量来决定。所以增加堆内存的大小并不会直接影响标记阶段占用的时间。

GC 的背景与一般原理

清除算法

标记-清除(Mark-Sweep)



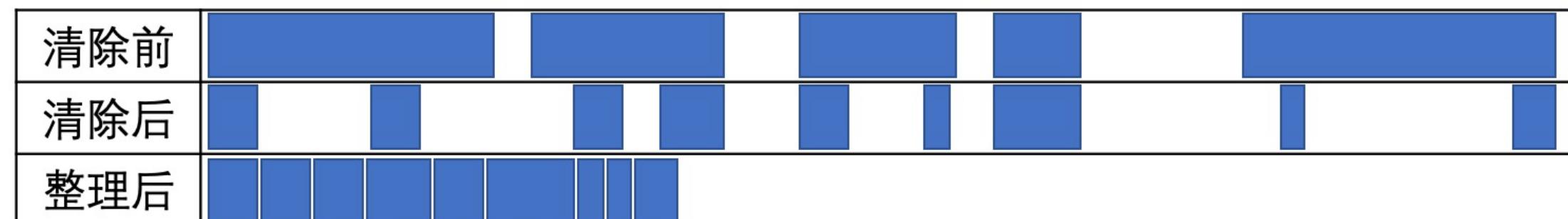
复制算法

标记-复制算法(Mark-Copy)



整理算法

标记-清除-整理算法(Mark-Sweep-Compact)



总结一下，三者有什么优缺点

串行 GC (Serial GC) /ParNewGC

-XX:+UseSerialGC 配置串行 GC

串行 GC 对年轻代使用 mark-copy (标记-复制) 算法 , 对老年代使用 mark-sweep-compact (标记-清除-整理) 算法。

两者都是单线程的垃圾收集器 , 不能进行并行处理 , 所以都会触发全线暂停 (STW) , 停止所有的应用线程。

因此这种 GC 算法不能充分利用多核 CPU 。不管有多少 CPU 内核 , JVM 在垃圾收集时都只能使用单个核心。

CPU 利用率高 , 暂停时间长。简单粗暴 , 就像老式的电脑 , 动不动就卡死。

该选项只适合几百 MB 堆内存的 JVM , 而且是单核 CPU 时比较有用。

想想 why ?

-XX:+UseParNewGC 改进版本的 Serial GC , 可以配合 CMS 使用。

并行 GC (Parallel GC)

-XX:+UseParallelGC

-XX:+UseParallelOldGC

-XX:+UseParallelGC -XX:+UseParallelOldGC

年轻代和老年代的垃圾回收都会触发 STW 事件。

在年轻代使用 标记-复制 (mark-copy) 算法，在老年代使用 标记-清除-整理 (mark-sweep-compact) 算法。

-XX : ParallelGCThreads=N 来指定 GC 线程数，其默认值为 CPU 核心数。

并行垃圾收集器适用于多核服务器，主要目标是增加吞吐量。因为对系统资源的有效使用，能达到更高的吞吐量：

- 在 GC 期间，所有 CPU 内核都在并行清理垃圾，所以总暂停时间更短；
- 在两次 GC 周期的间隔期，没有 GC 线程在运行，不会消耗任何系统资源。

演示：并行 GC, 常用参数以及其内存分配

5. CMS GC/G1 GC

CMS GC (Mostly Concurrent Mark and Sweep Garbage Collector)

`-XX:+UseConcMarkSweepGC`

其对年轻代采用并行 STW 方式的 mark-copy (标记-复制) 算法，对老年代主要使用并发 mark-sweep (标记-清除) 算法。

CMS GC 的设计目标是避免在老年代垃圾收集时出现长时间的卡顿，主要通过两种手段来达成此目标：

1. 不对老年代进行整理，而是使用空闲列表 (free-lists) 来管理内存空间的回收。
2. 在 mark-and-sweep (标记-清除) 阶段的大部分工作和应用线程一起并发执行。

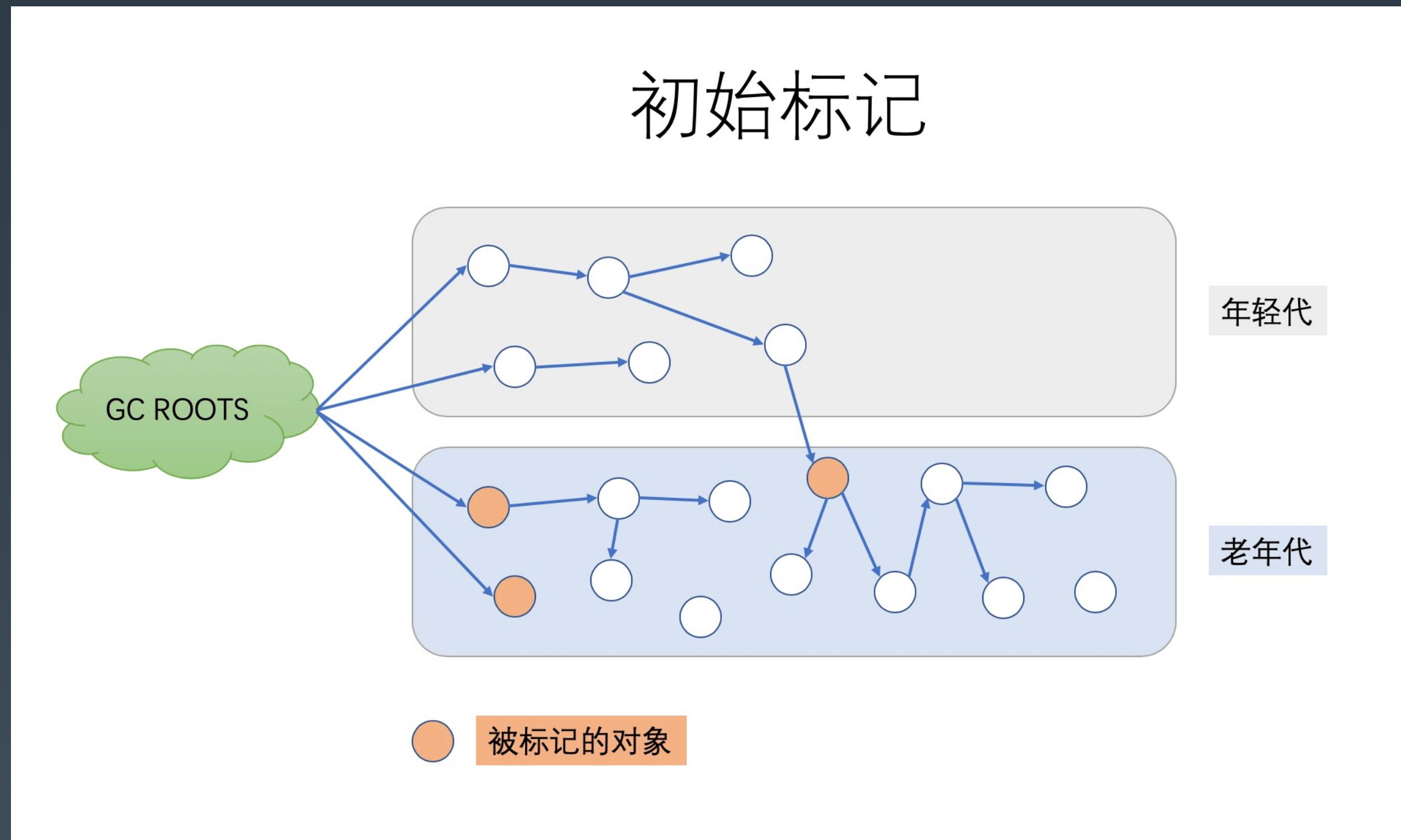
也就是说，在这些阶段并没有明显的应用线程暂停。但值得注意的是，它仍然和应用线程争抢 CPU 时。

默认情况下，CMS 使用的并发线程数等于 CPU 核心数的 1/4。

如果服务器是多核 CPU，并且主要调优目标是降低 GC 停顿导致的系统延迟，那么使用 CMS 是个很明智的选择。进行老年代的并发回收时，可能会伴随着多次年轻代的 minor GC。

思考：并行 Parallel 与并发 Concurrent 的区别？

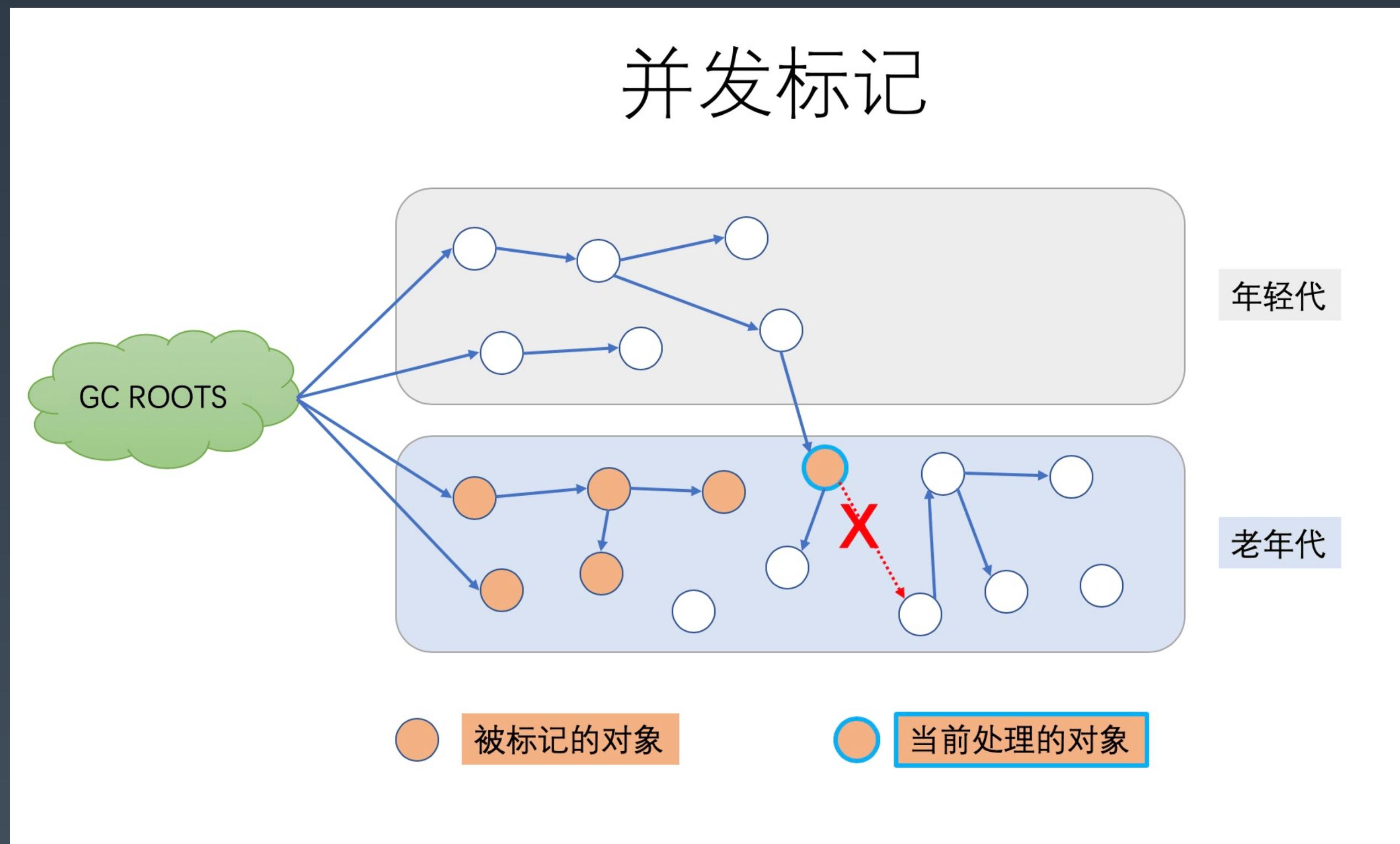
CMS GC--六个阶段 1 (STW)



- 阶段 1: Initial Mark (初始标记)
- 阶段 2: Concurrent Mark (并发标记)
- 阶段 3: Concurrent Preclean (并发预清理)
- 阶段 4: Final Remark (最终标记)
- 阶段 5: Concurrent Sweep (并发清除)
- 阶段 6: Concurrent Reset (并发重置)

这个阶段伴随着 STW 暂停。初始标记的目标是标记所有的根对象，包括根对象直接引用的对象，以及被年轻代中所有存活对象所引用的对象（老年代单独回收）。

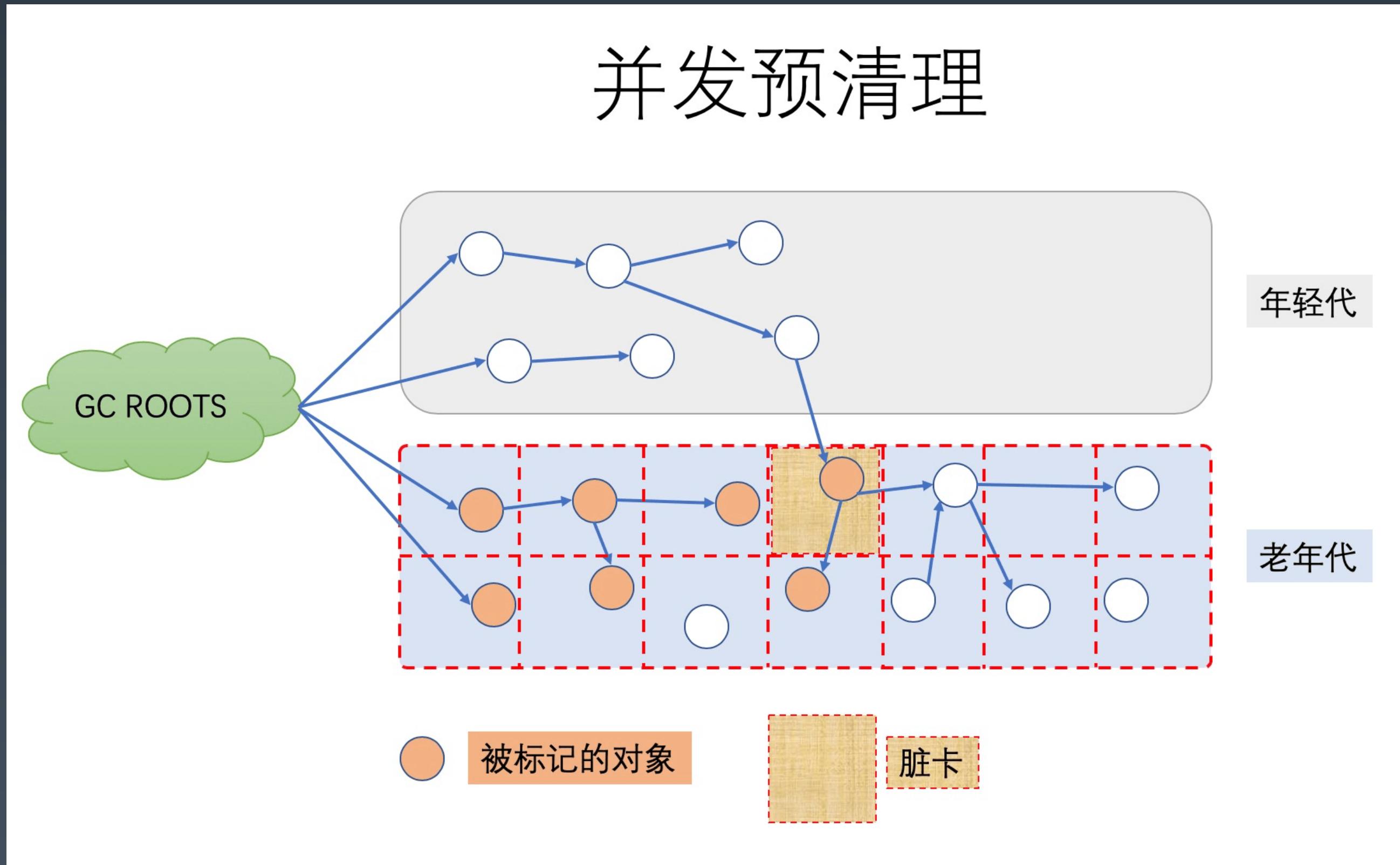
CMS GC--六个阶段 2



- 阶段 1: Initial Mark (初始标记)
- 阶段 2: Concurrent Mark (并发标记)
- 阶段 3: Concurrent Preclean (并发预清理)
- 阶段 4: Final Remark (最终标记)
- 阶段 5: Concurrent Sweep (并发清除)
- 阶段 6: Concurrent Reset (并发重置)

在此阶段，CMS GC 遍历老年代，标记所有的存活对象，从前一阶段“Initial Mark”找到的根对象开始算起。“并发标记”阶段，就是与应用程序同时运行，不用暂停的阶段。

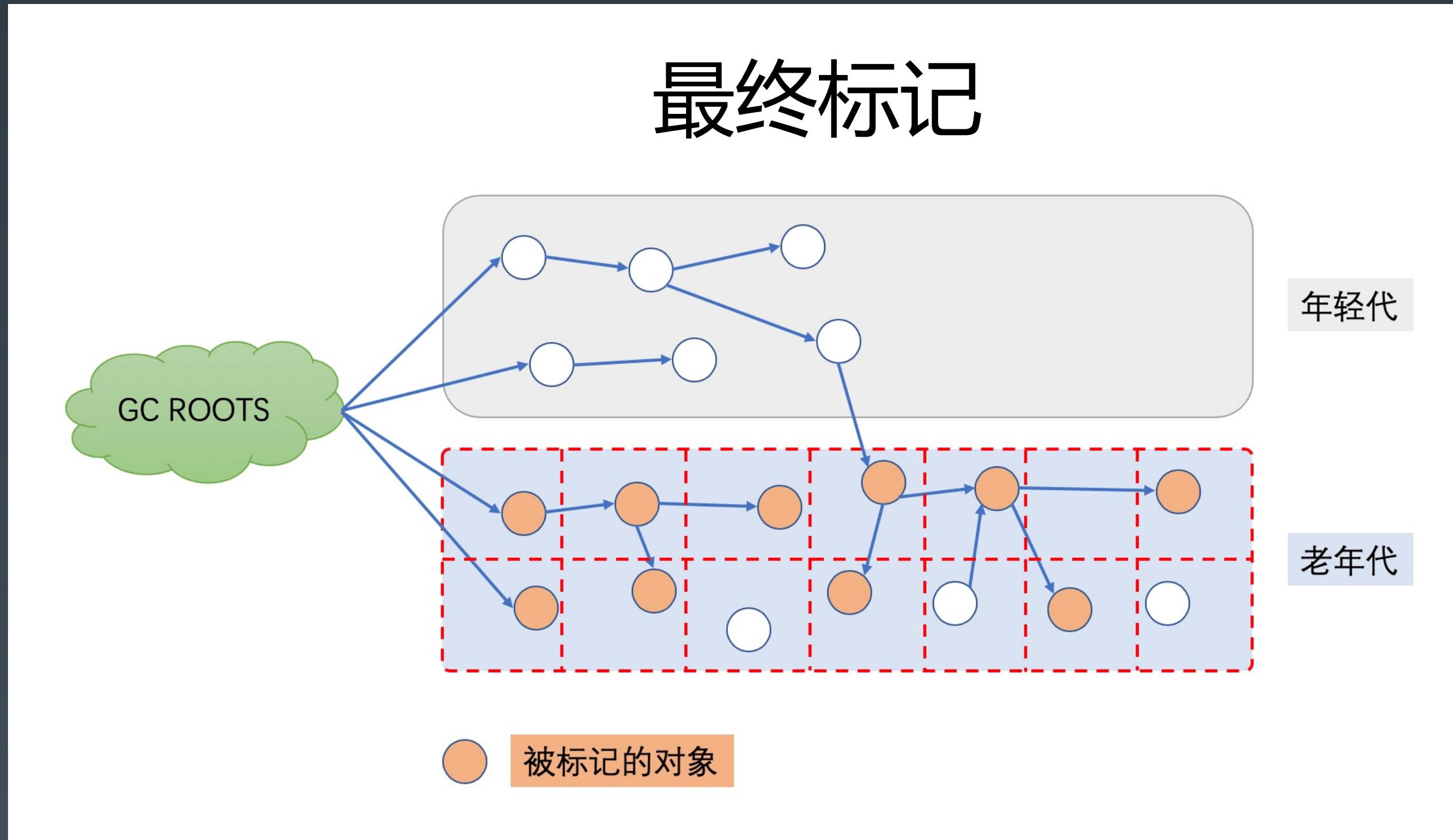
CMS GC--六个阶段 3



- 阶段 1: Initial Mark (初始标记)
- 阶段 2: Concurrent Mark (并发标记)
- 阶段 3: Concurrent Preclean (并发预清理)
- 阶段 4: Final Remark (最终标记)
- 阶段 5: Concurrent Sweep (并发清除)
- 阶段 6: Concurrent Reset (并发重置)

此阶段同样是与应用线程并发执行的，不需要停止应用线程。因为前一阶段【并发标记】与程序并发运行，可能有一些引用关系已经发生了改变。如果在并发标记过程中引用关系发生了变化，JVM 会通过“Card (卡片)”的方式将发生了改变的区域标记为“脏”区，这就是所谓的 卡片标记 (Card Marking)。

CMS GC--六个阶段 4 (STW)



阶段 1: Initial Mark (初始标记)

阶段 2: Concurrent Mark (并发标记)

阶段 3: Concurrent Preclean (并发预清理)

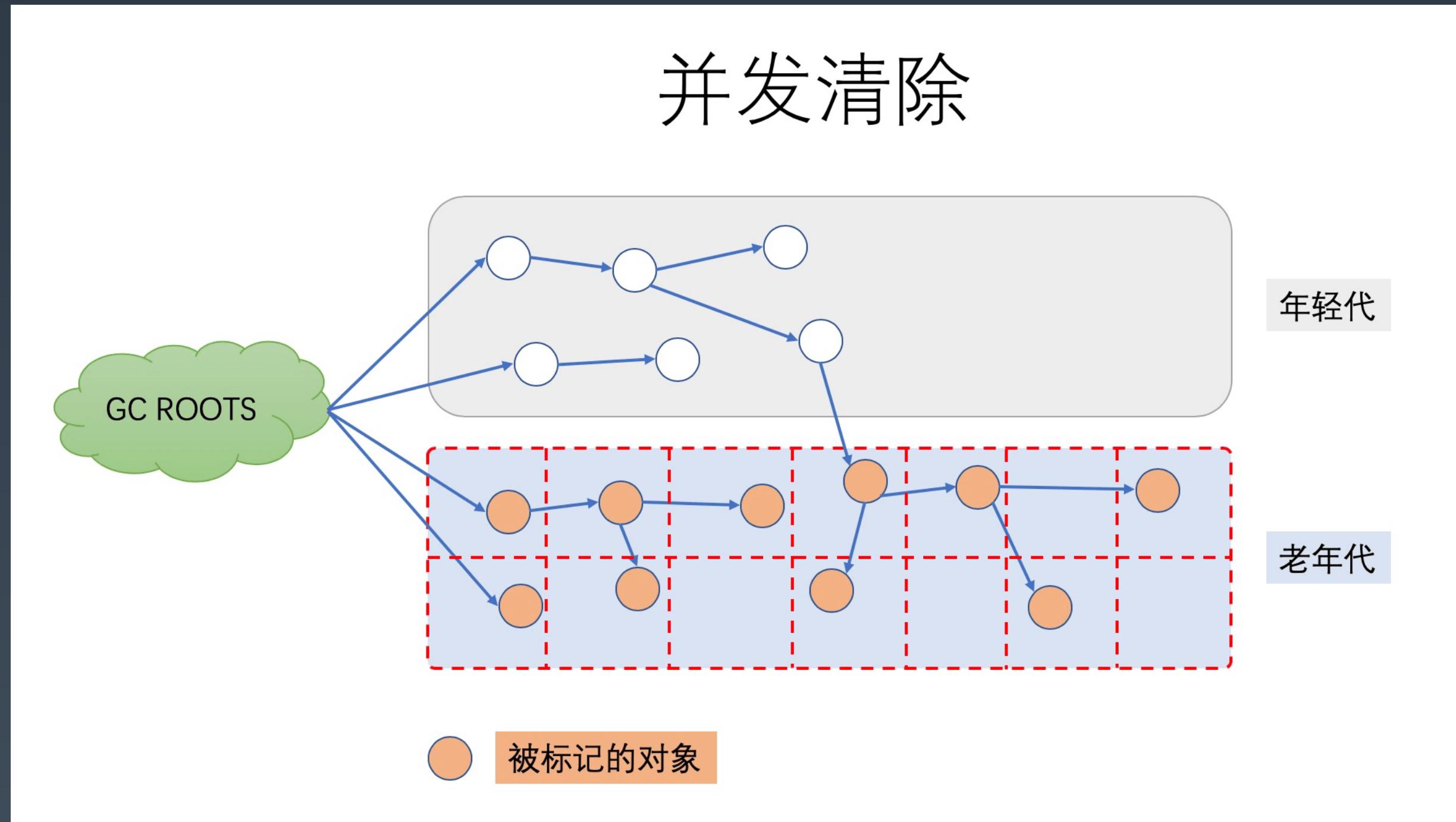
阶段 4: Final Remark (最终标记)

阶段 5: Concurrent Sweep (并发清除)

阶段 6: Concurrent Reset (并发重置)

最终标记阶段是此次 GC 事件中的第二次 (也是最后一次) STW 停顿。本阶段的目标是完成老年代中所有存活对象的标记。因为之前的预清理阶段是并发执行的，有可能 GC 线程跟不上应用程序的修改速度。所以需要一次 STW 暂停来处理各种复杂的情况。通常 CMS 会尝试在年轻代尽可能空的情况下执行 Final Remark 阶段，以免连续触发多次 STW 事件。

CMS GC--六个阶段 5



- 阶段 1: Initial Mark (初始标记)
 - 阶段 2: Concurrent Mark (并发标记)
 - 阶段 3: Concurrent Preclean (并发预清理)
 - 阶段 4: Final Remark (最终标记)
 - 阶段 5: Concurrent Sweep (并发清除)
 - 阶段 6: Concurrent Reset (并发重置)
- 此阶段与应用程序并发执行，不需要 STW 停顿。JVM 在此阶段删除不再使用的对象，并回收他们占用的内存空间。

CMS GC--六个阶段 6

阶段 1: Initial Mark (初始标记)

阶段 2: Concurrent Mark (并发标记)

阶段 3: Concurrent Preclean (并发预清理)

阶段 4: Final Remark (最终标记)

阶段 5: Concurrent Sweep (并发清除)

阶段 6: Concurrent Reset (并发重置)

此阶段与应用程序并发执行，重置 CMS 算法相关的内部

数据，为下一次 GC 循环做准备。

CMS 垃圾收集器在减少停顿时间上做了很多复杂而有用的工作，用于垃圾回收的并发线程执行的同时，并不需要暂停应用线程。当然，CMS 也有一些缺点，其中最大的问题是老年代内存碎片问题（因为不压缩），在某些情况下 GC 会造成不可预测的暂停时间，特别是堆内存较大的情况下。

演示：CMS GC，常用参数以及其内存分配
(注意跟 ParallelGC 有什么差异)

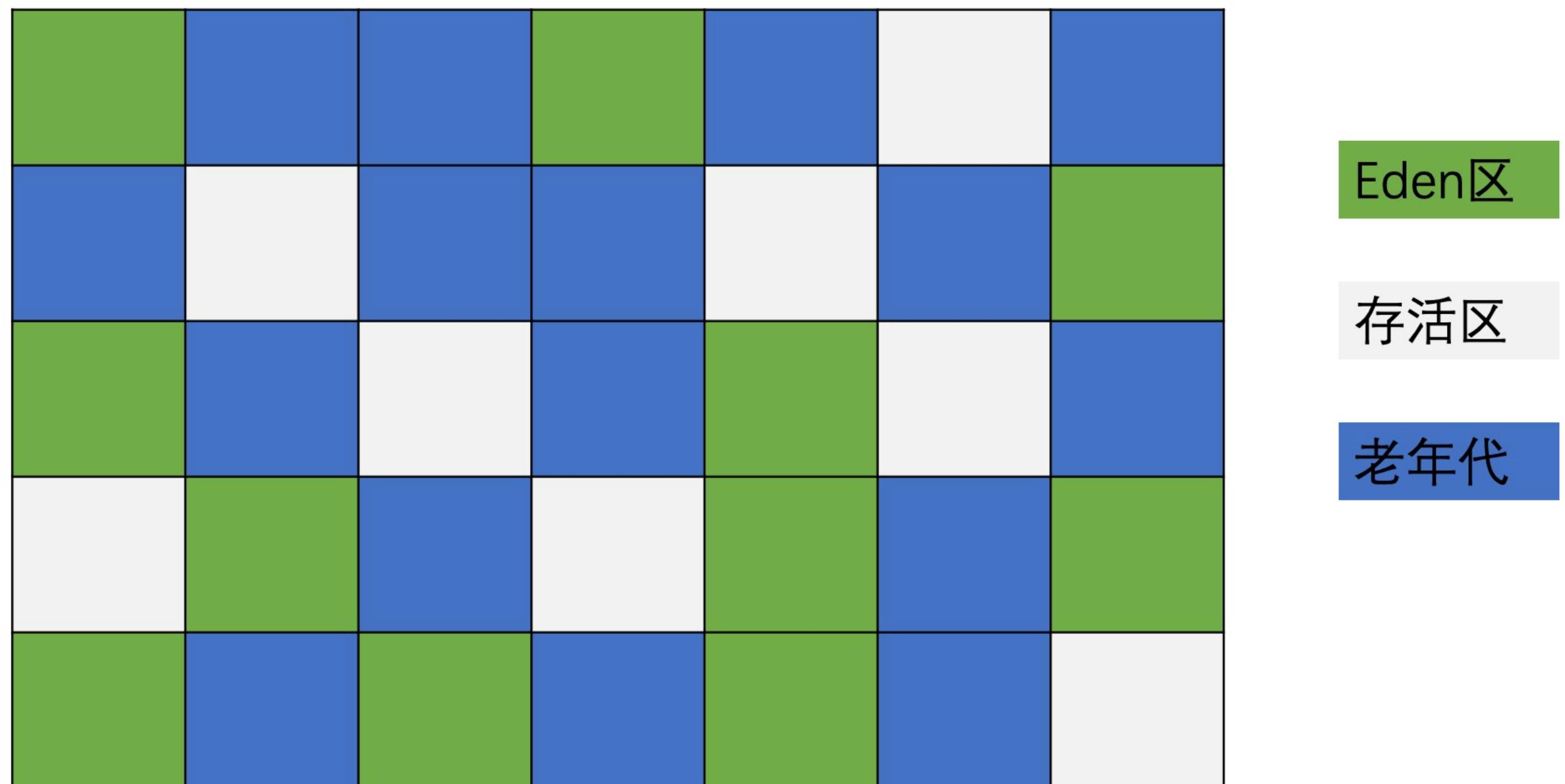
最大young区大小：

ParallelGC: $1024M / 3 = 341.3M$

CMS: $64M * \text{GC线程数} 4 * 13 / 10 = 332.8M$

G1 GC

G1内存划分



G1 的全称是 Garbage-First，意为垃圾优先，哪一块的垃圾最多就优先清理它。

G1 GC 最主要的设计目标是：将 STW 停顿的时间和分布，变成可预期且可配置的。

事实上，G1 GC 是一款软实时垃圾收集器，可以为其设置某项特定的性能指标。为了达成可预期停顿时间的指标，G1 GC 有一些独特的实现。

首先，堆不再分成年轻代和老年代，而是划分为多个（通常是 2048 个）可以存放对象的小块堆区域 (smaller heap regions)。每个小块，可能会被定义成 Eden 区，会被指定为 Survivor 区或者 Old 区。在逻辑上，所有的 Eden 区和 Survivor 区合起来就是年轻代，所有的 Old 区拼在一起那就是老年代。

-XX:+UseG1GC -XX:MaxGCPauseMillis=50

G1 GC



这样划分之后，使得 G1 不必每次都去收集整个堆空间，而是以增量的方式来进行处理：每次只处理一部分内存块，称为此次 GC 的回收集(collection set)。每次 GC 暂停都会收集所有年轻代的内存块，但一般只包含部分老年代的内存块。

G1 的另一项创新是，在并发阶段估算每个小堆块存活对象的总数。构建回收集的原则是：垃圾最多的小块会被优先收集。这也是 G1 名称的由来。

G1 GC--配置参数

- XX: +UseG1GC: 启用 G1 GC;
- XX: **G1NewSizePercent**: 初始年轻代占整个 Java Heap 的大小, 默认值为 5%;
- XX: **G1MaxNewSizePercent**: 最大年轻代占整个 Java Heap 的大小, 默认值为 60%;
- XX: **G1HeapRegionSize**: 设置每个 Region 的大小, 单位 MB, 需要为 1、2、4、8、16、32 中的某个值, 默认是堆内存的 1/2000。如果这个值设置比较大, 那么大对象就可以进入 Region 了;
- XX: **ConcGCThreads**: 与 Java 应用一起执行的 GC 线程数量, 默认是 Java 线程的 1/4, 减少这个参数的数值可能会提升并行回收的效率, 提高系统内部吞吐量。如果这个数值过低, 参与回收垃圾的线程不足, 也会导致并行回收机制耗时加长;
- XX: **+InitiatingHeapOccupancyPercent** (简称 IHOP) : G1 内部并行回收循环启动的阈值, 默认为 Java Heap 的 45%。这个可以理解为老年代使用大于等于 45% 的时候, JVM 会启动垃圾回收。这个值非常重要, 它决定了在什么时间启动老年代的并行回收;
- XX: **G1HeapWastePercent**: G1 停止回收的最小内存大小, 默认是堆大小的 5%。GC 会收集所有的 Region 中的对象, 但是如果下降到了 5%, 就会停下来不再收集了。就是说, 不必每次回收就把所有的垃圾都处理完, 可以遗留少量的下次处理, 这样也降低了单次消耗的时间;
- XX: **G1MixedGCountTarget**: 设置并行循环之后需要有多少个混合 GC 启动, 默认值是 8 个。老年代 Regions 的回收时间通常比年轻代的收集时间要长一些。所以如果混合收集器比较多, 可以允许 G1 延长老年代的收集时间。

G1 GC--配置参数

-XX: +G1PrintRegionLivenessInfo: 这个参数需要和 -XX:+UnlockDiagnosticVMOptions 配合启动，打印 JVM 的调试信息，每个 Region 里的对象存活信息。

-XX: G1ReservePercent: G1 为了保留一些空间用于年代之间的提升，默认值是堆空间的 10%。因为大量执行回收的地方在年轻代（存活时间较短），所以如果你的应用里面有比较大的堆内存空间、比较多的大对象存活，这里需要保留一些内存。

-XX: +G1SummarizeRSetStats: 这也是一个 VM 的调试信息。如果启用，会在 VM 退出的时候打印出 Rsets 的详细总结信息。

如果启用 -XX:G1SummaryRSetStatsPeriod 参数，就会阶段性地打印 Rsets 信息。

-XX: +G1TraceConcRefinement: 这个也是一个 VM 的调试信息，如果启用，并行回收阶段的日志就会被详细打印出来。

-XX: +GCTimeRatio: 这个参数就是计算花在 Java 应用线程上和花在 GC 线程上的时间比率，默认是 9，跟新生代内存的分配比例一致。这个参数主要的目的是让用户可以控制花在应用上的时间，G1 的计算公式是 $100 / (1 + GCTimeRatio)$ 。这样如果参数设置为 9，则最多 10% 的时间会花在 GC 工作上面。Parallel GC 的默认值是 99，表示 1% 的时间被用在 GC 上面，这是因为 Parallel GC 贯穿整个 GC，而 G1 则根据 Region 来进行划分，不需要全局性扫描整个内存堆。

-XX: +UseStringDeduplication: 手动开启 Java String 对象的去重工作，这个是 JDK8u20 版本之后新增的参数，主要用于相同 String 避免重复申请内存，节约 Region 的使用。

-XX: MaxGCPauseMills: 预期 G1 每次执行 GC 操作的暂停时间，单位是毫秒，默认值是 200 毫秒，G1 会尽量保证控制在这个范围内。

G1 GC 的处理步骤 1

1、年轻代模式转移暂停 (Evacuation Pause)

G1 GC 会通过前面一段时间的运行情况来不断的调整自己的回收策略和行为，以此来比较稳定地控制暂停时间。在应用程序刚启动时，G1 还没有采集到什么足够的信息，这时候就处于初始的 fully-young 模式。当年轻代空间用满后，应用线程会被暂停，年轻代内存块中的存活对象被拷贝到存活区。如果没有存活区，则任意选择一部分空闲的内存块作为存活区。

拷贝的过程称为转移 (Evacuation)，这和前面介绍的其他年轻代收集器是一样的工作原理。

G1 GC 的处理步骤 2

2、并发标记 (Concurrent Marking)

同时我们也可以看到，G1 GC 的很多概念建立在 CMS 的基础上，所以下面的内容需要对 CMS 有一定的理解。

G1 并发标记的过程与 CMS 基本上是一样的。G1 的并发标记通过 Snapshot-At-The-Beginning (起始快照) 的方式，在标记阶段开始时记下所有的存活对象。即使在标记的同时又有一些变成了垃圾。通过对象的存活信息，可以构建出每个小堆块的存活状态，以便回收集能高效地进行选择。

这些信息在接下来的阶段会用来执行老年代区域的垃圾收集。

有两种情况是可以完全并发执行的：

- 一、如果在标记阶段确定某个小堆块中没有存活对象，只包含垃圾；
- 二、在 STW 转移暂停期间，同时包含垃圾和存活对象的老年代小堆块。

当堆内存的总体使用比例达到一定数值，就会触发并发标记。这个默认比例是 45%，但也可以通过 JVM 参数 `InitiatingHeapOccupancyPercent` 来设置。和 CMS 一样，G1 的并发标记也是由多个阶段组成，其中一些阶段是完全并发的，还有一些阶段则会暂停应用线程。

G1 GC 的处理步骤 2

阶段 1: Initial Mark (初始标记)

此阶段标记所有从 GC 根对象直接可达的对象。

阶段 2: Root Region Scan (Root区扫描)

此阶段标记所有从 "根区域" 可达的存活对象。根区域包括：非空的区域，以及在标记过程中不得不收集的区域。

阶段 3: Concurrent Mark (并发标记)

此阶段和 CMS 的并发标记阶段非常类似：只遍历对象图，并在一个特殊的位图中标记能访问到的对象。

阶段 4: Remark (再次标记)

和 CMS 类似，这是一次 STW 停顿（因为不是并发的阶段），以完成标记过程。G1 收集器会短暂地停止应用线程，停止并发更新信息的写入，处理其中的少量信息，并标记所有在并发标记开始时未被标记的存活对象。

阶段 5: Cleanup (清理)

最后这个清理阶段为即将到来的转移阶段做准备，统计小堆块中所有存活的对象，并将小堆块进行排序，以提升GC 的效率，维护并发标记的内部状态。所有不包含存活对象的小堆块在此阶段都被回收了。有一部分任务是并发的：例如空堆区的回收，还有大部分的存活率计算。此阶段也需要一个短暂的 STW 暂停。

G1 GC 的处理步骤 3

3、转移暂停: 混合模式 (Evacuation Pause (mixed))

并发标记完成之后，G1将执行一次混合收集（mixed collection），就是不只清理年轻代，还将一部分老年代区域也加入到回收集中。混合模式的转移暂停不一定紧跟并发标记阶段。有很多规则和历史数据会影响混合模式的启动时机。比如，假若在老年代中可以并发地腾出很多的小堆块，就没有必要启动混合模式。

因此，在并发标记与混合转移暂停之间，很可能会存在多次 young 模式的转移暂停。

具体添加到回收集的老年代小堆块的大小及顺序，也是基于许多规则来判定的。其中包括指定的软实时性能指标，存活性，以及在并发标记期间收集的 GC 效率等数据，外加一些可配置的 JVM 选项。混合收集的过程，很大程度上和前面的 fully-young gc 是一样的。

G1 GC 的注意事项

特别需要注意的是，某些情况下 G1 触发了 Full GC，这时 G1 会退化使用 Serial 收集器来完成垃圾的清理工作，它仅仅使用单线程来完成 GC 工作，GC 暂停时间将达到秒级别的。

1. 并发模式失败

G1 启动标记周期，但在 Mix GC 之前，老年代就被填满，这时候 G1 会放弃标记周期。

解决办法：增加堆大小，或者调整周期（例如增加线程数-XX: ConcGCThreads 等）。

2. 晋升失败

没有足够的内存供存活对象或晋升对象使用，由此触发了 Full GC(to-space exhausted/to-space overflow)。

解决办法：

- 增加 -XX: G1ReservePercent 选项的值（并相应增加总的堆大小）增加预留内存量。
- 通过减少 -XX: InitiatingHeapOccupancyPercent 提前启动标记周期。
- 也可以通过增加 -XX: ConcGCThreads 选项的值来增加并行标记线程的数目。

3. 巨型对象分配失败

当巨型对象找不到合适的空间进行分配时，就会启动 Full GC，来释放空间。

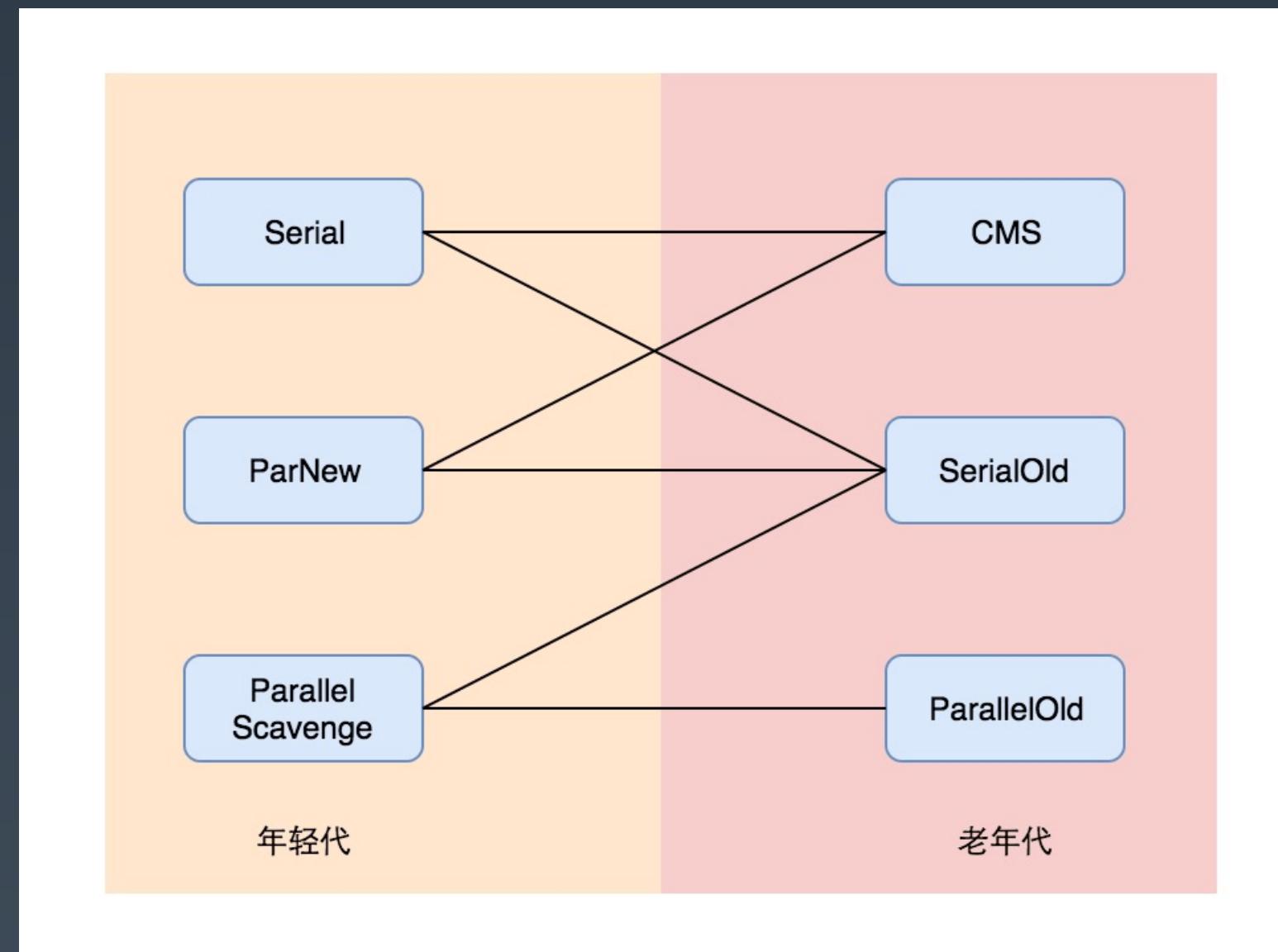
解决办法：增加内存或者增大 -XX: G1HeapRegionSize

演示：G1 GC, 常用参数以及其内存分配
(注意跟 CMS/ParallelGC 有什么差异)

各个 GC 对比

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

常用的 GC 组合（重点）



常用的组合为：

(1) **Serial+Serial Old** 实现单线程的低延迟垃圾回收机制；

(2) **ParNew+CMS**，实现多线程的低延迟垃圾回收机制；

(3) **Parallel Scavenge**和**Parallel Scavenge Old**，实现多线程的高吞吐量垃圾回收机制。

GC 如何选择

选择正确的 GC 算法，唯一可行的方式就是去尝试，一般性的指导原则：

1. 如果系统考虑吞吐优先，CPU 资源都用来最大程度处理业务，用 Parallel GC；
2. 如果系统考虑低延迟有限，每次 GC 时间尽量短，用 CMS GC；
3. 如果系统内存堆较大，同时希望整体来看平均 GC 时间可控，使用 G1 GC。

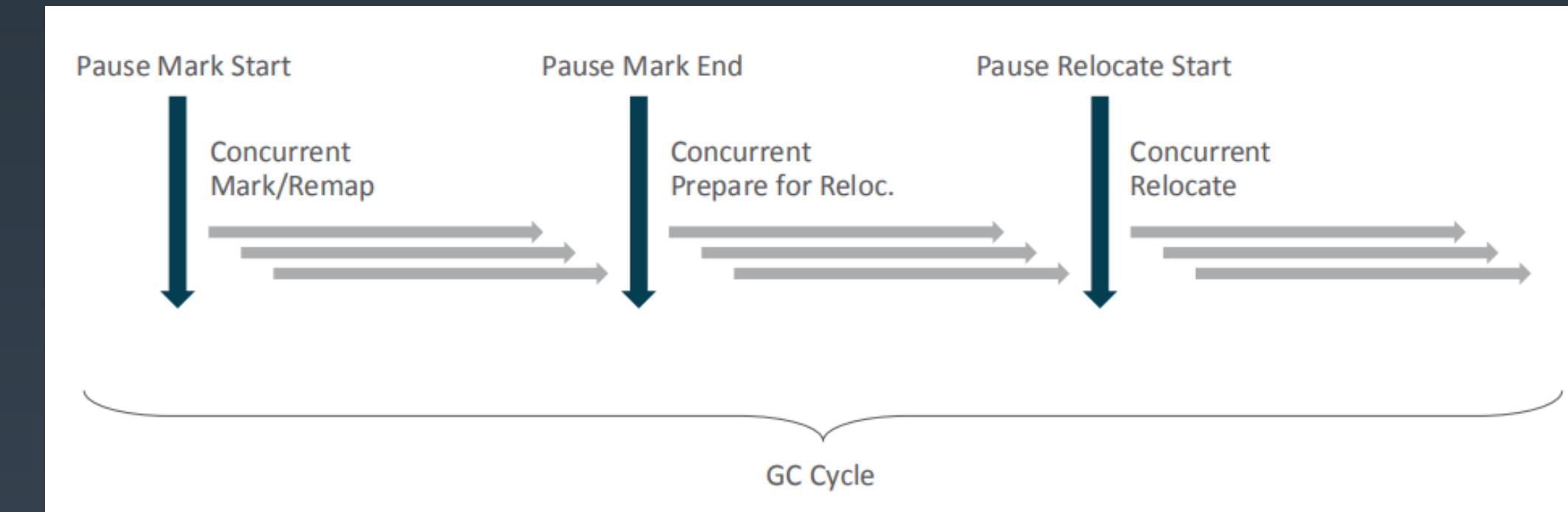
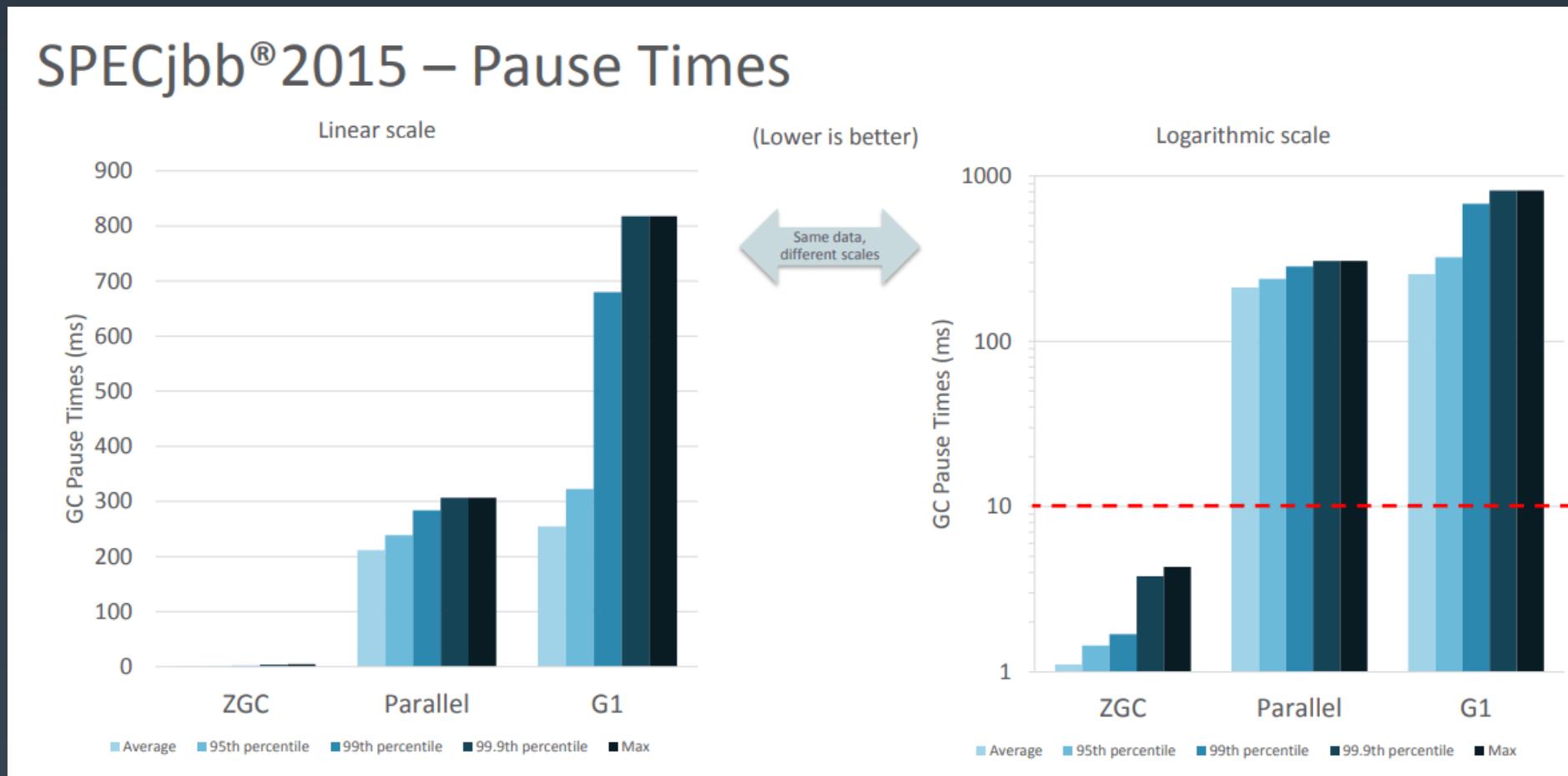
对于内存大小的考量：

1. 一般 4G 以上，算是比较大，用 G1 的性价比较高。
2. 一般超过 8G，比如 16G-64G 内存，非常推荐使用 G1 GC。

最后讨论一个很多开发者经常忽视的问题，也是面试大厂常问的问题：JDK8 的默认 GC 是什么？

JDK9, JDK10, JDK11...等等默认的 GC 是什么？

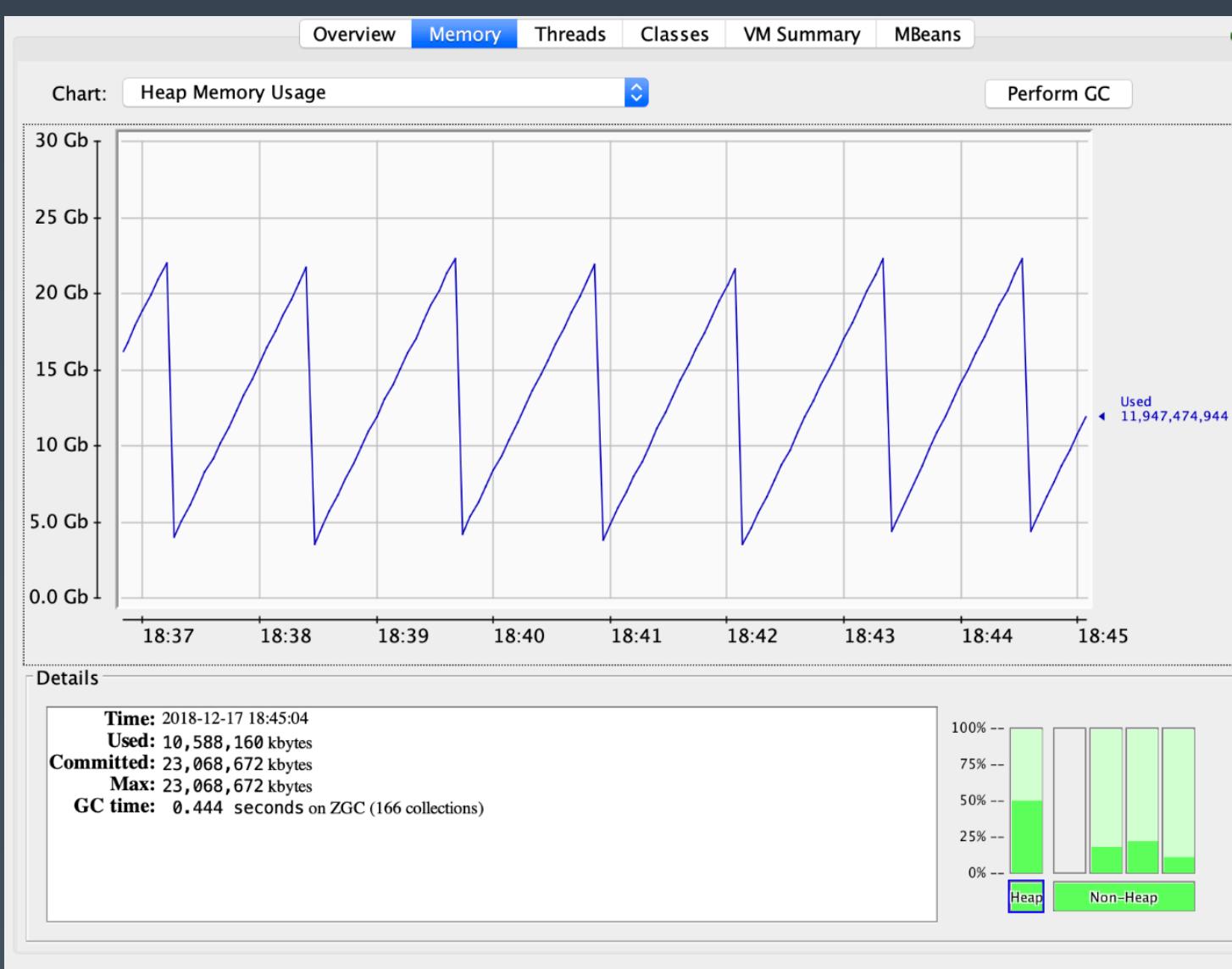
ZGC 介绍



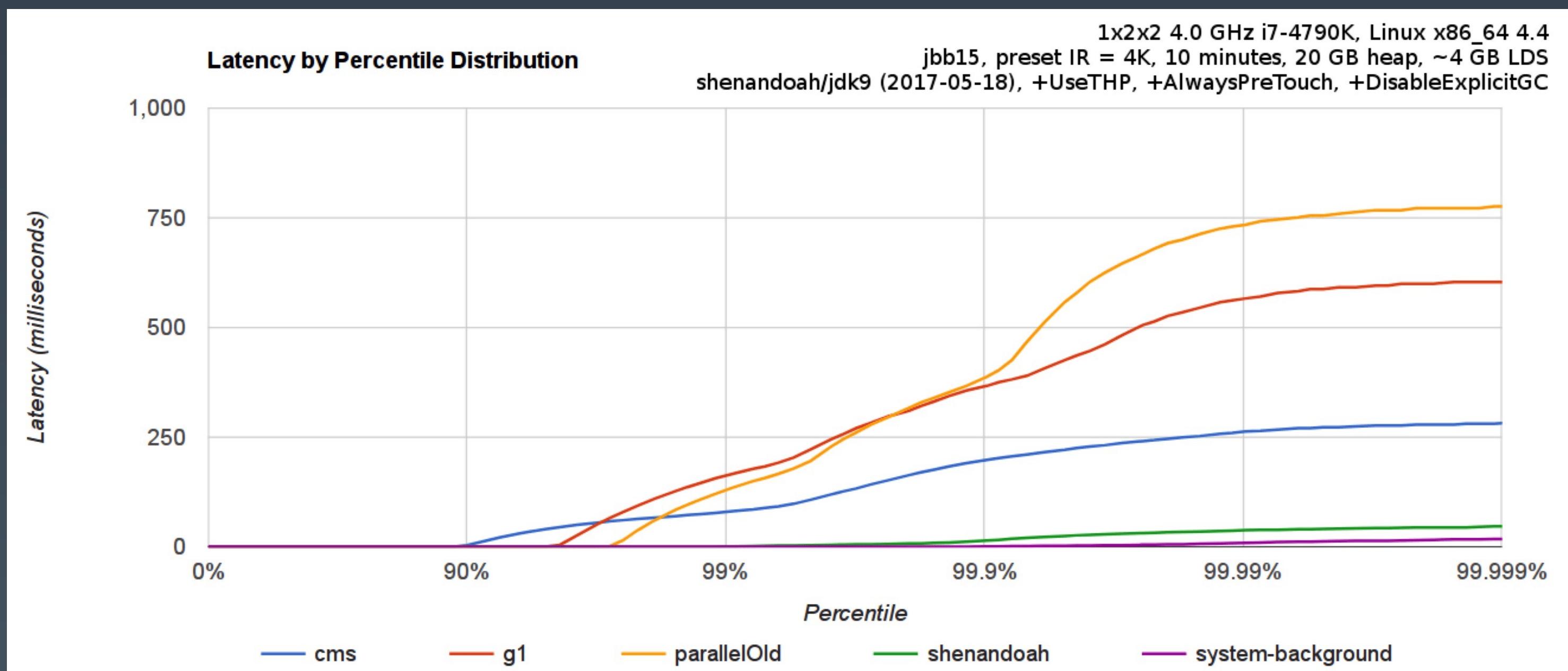
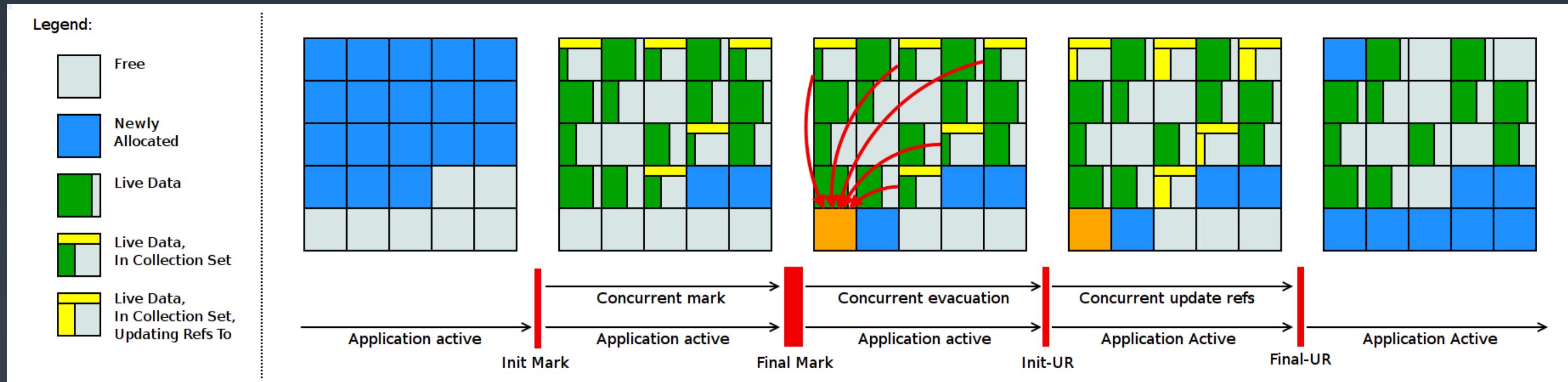
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx16g

ZGC 最主要的特点包括：

1. GC 最大停顿时间不超过 10ms
2. 堆内存支持范围广，小至几百 MB 的堆空间，大至 4TB 的超大堆内存 (JDK13 升至 16TB)
3. 与 G1 相比，应用吞吐量下降不超过 15%
4. 当前只支持 Linux/x64 位平台，JDK15 后支持 MacOS 和 Windows 系统



ShenandoahGC 介绍



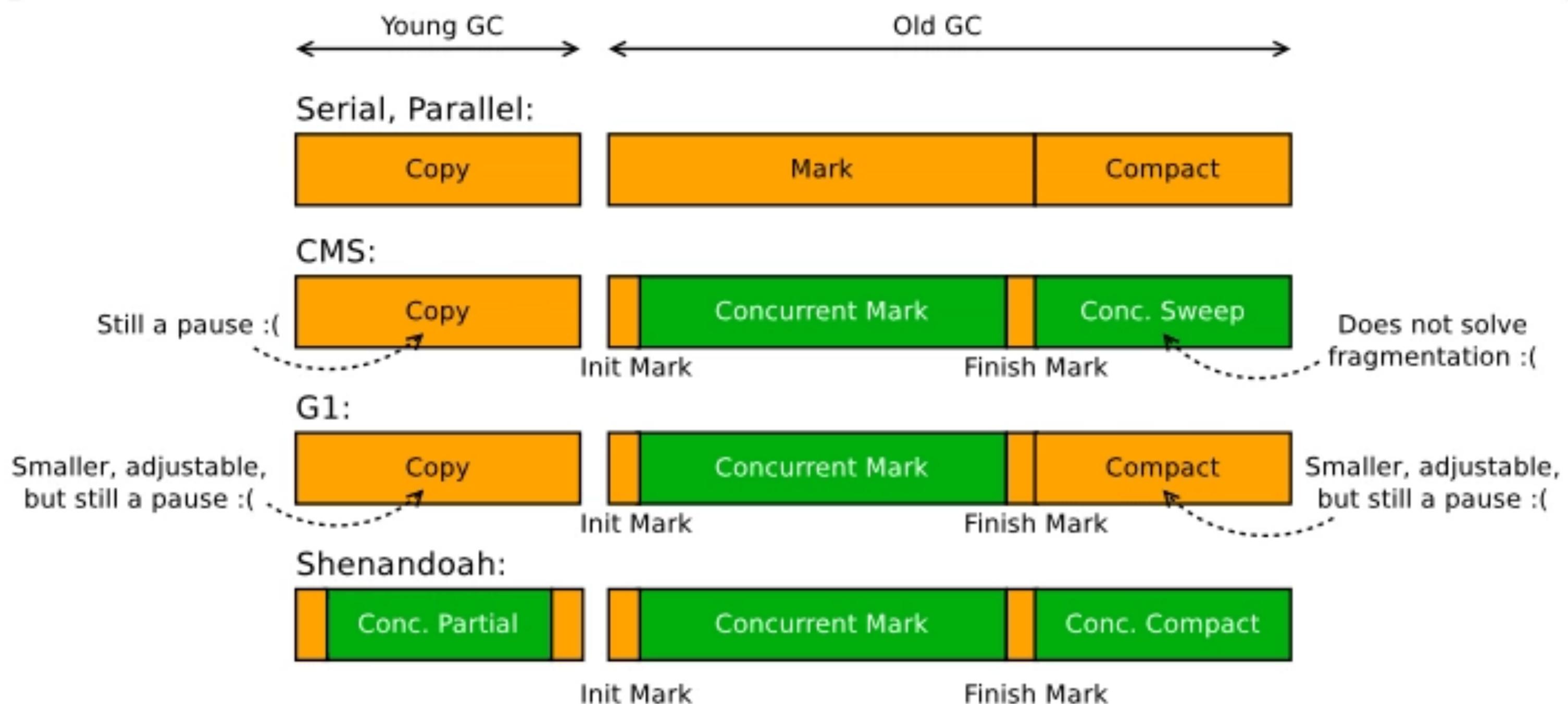
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx16g

Shenandoah GC 立项比 ZGC 更早，设计为 GC 线程与应用线程并发执行的方式，通过实现垃圾回收过程的并发处理，改善停顿时间，使得 GC 执行线程能够在业务处理线程运行过程中进行堆压缩、标记和整理，从而消除了绝大部分的暂停时间。

Shenandoah 团队对外宣称 Shenandoah GC 的暂停时间与堆大小无关，无论是 200 MB 还是 200 GB 的堆内存，都可以保障具有很低的暂停时间（注意：并不像 ZGC 那样保证暂停时间在 10ms 以内）。

ShennandoahGC 与其他 GC 的 STW 比较

Overview: Landscape



GC 总结

到目前为止，我们一共了解了 Java 目前支持的所有 GC 算法，一共有 7 类：

1. 串行 GC (Serial GC)：单线程执行，应用需要暂停；
2. 并行 GC (ParNew、Parallel Scavenge、Parallel Old)：多线程并行地执行垃圾回收，关注与高吞吐；
3. CMS (Concurrent Mark-Sweep)：多线程并发标记和清除，关注与降低延迟；
4. G1 (G First)：通过划分多个内存区域做增量整理和回收，进一步降低延迟；
5. ZGC (Z Garbage Collector)：通过着色指针和读屏障，实现几乎全部的并发执行，几毫秒级别的延迟，线性可扩展；
6. Epsilon: 实验性的 GC，供性能分析使用；
7. Shenandoah: G1 的改进版本，跟 ZGC 类似。

GC 总结

可以看出 GC 算法和实现的演进路线：

1. 串行 -> 并行：重复利用多核 CPU 的优势，大幅降低 GC 暂停时间，提升吞吐量。
2. 并行 -> 并发：不只开多个 GC 线程并行回收，还将 GC 操作拆分为多个步骤，让很多繁重的任务和应用线程一起并发执行，减少了单次 GC 暂停持续的时间，这能有效降低业务系统的延迟。
3. CMS -> G1：G1 可以说是在 CMS 基础上进行迭代和优化开发出来的，划分为多个小堆块进行增量回收，这样就更进一步地降低了单次 GC 暂停的时间
4. G1 -> ZGC：ZGC 号称无停顿垃圾收集器，这又是一次极大的改进。ZGC 和 G1 有一些相似的地方，但是底层的算法和思想又有了全新的突破。

脱离场景谈性能都是耍流氓”

目前绝大部分 Java 应用系统，堆内存并不大比如 2G-4G 以内，而且对 10ms 这种低延迟的 GC 暂停不敏感，也就是说处理一个业务步骤，大概几百毫秒都是可以接受的，GC 暂停 100ms 还是 10ms 没多大区别。另一方面，系统的吞吐量反而往往是我们追求的重点，这时候就需要考虑采用并行 GC。

如果堆内存再大一些，可以考虑 G1 GC。如果内存非常大（比如超过 16G，甚至是 64G、128G），或者是对延迟非常敏感（比如高频量化交易系统），就需要考虑使用本节提到的新 GC (ZGC/Shenandoah)。

7. 总结回顾与作业实践

第 2 课总结回顾

工具有哪些？

GC 有哪些？

都有什么特点？

第 2 节课作业实践

1、本机使用 G1 GC 启动一个程序，仿照课上案例分析一下 JVM 情况

可以使用 gateway-server-0.0.1-SNAPSHOT.jar

注意关闭自适应参数：-XX:-UseAdaptiveSizePolicy

```
> java -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+UseSerialGC -jar target/gateway-server-0.0.1-SNAPSHOT.jar
```

```
> java -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+UseParallelGC -jar target/gateway-server-0.0.1-SNAPSHOT.jar
```

```
> java -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+UseConcMarkSweepGC -jar target/gateway-server-0.0.1-SNAPSHOT.jar
```

```
> java -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+UseG1GC -XX:MaxGCPauseMillis=50 -jar target/gateway-server-0.0.1-SNAPSHOT.jar
```

使用 jmap , jstat , jstack , 以及可视化工具 , 查看 JVM 情况。

Mac 上可以用 wrk , Windows 上可以按照 superbenchmark 压测 <http://localhost:8088/api/hello> 查看 JVM。

THANKS! |  极客大学