

# JVM基础知识及性能调优

Jason

# JVM基础知识及性能调优



**JVM**基本结构

JVM的重要概念及相关参数

JVM工具

垃圾回收算法、垃圾收集器

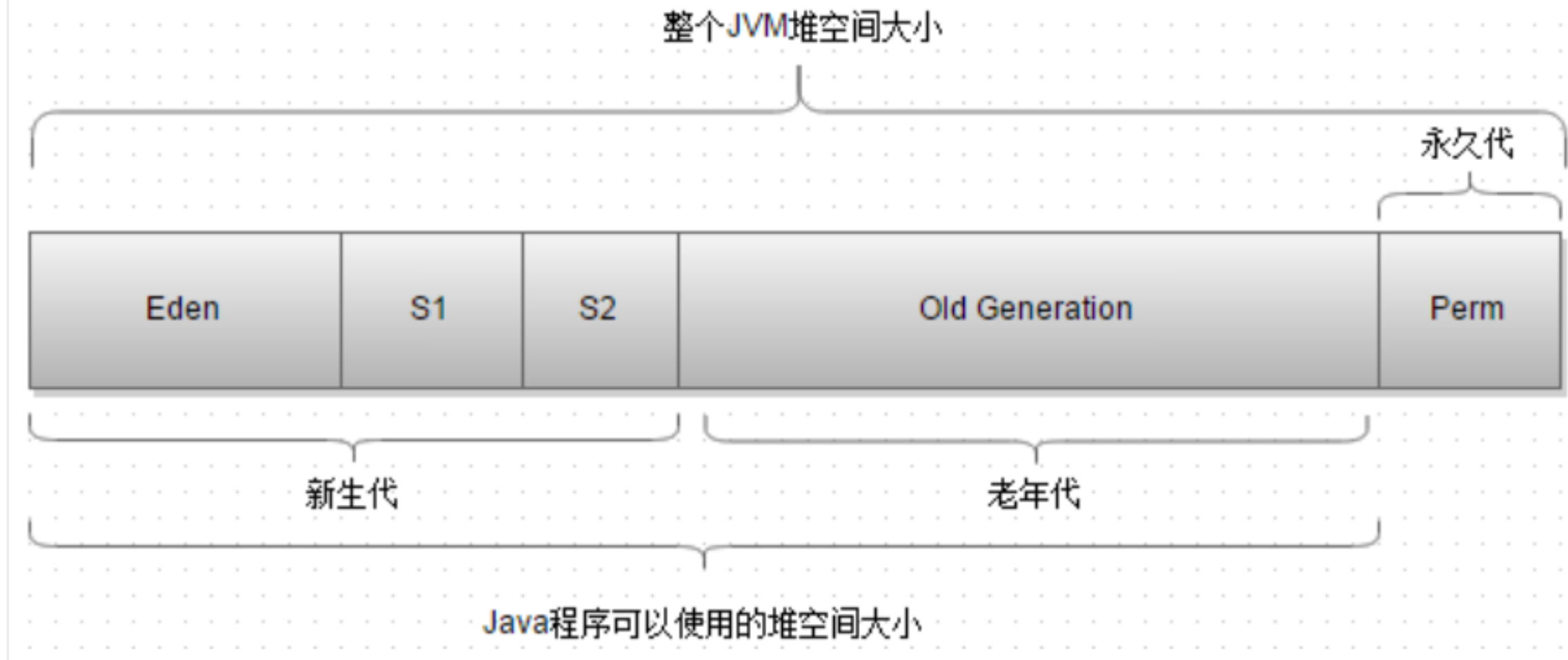
**JVM**常见错误及调优示例



# Java虚拟机的基本结构



# JVM堆空间布局



# JVM的分代

年轻代

老年代

永久代



# 为什么分代

---

- 不同类型对象的生命周期是不一样的;
- 不同年代的对象采取不同的收集方式, 以便提高回收效率;



# 为什么分代

---

- 如果不分代，每次垃圾回收都需要遍历内存空间，花费时间较长，效率低；
- 如果不分代，多次垃圾回收后，生命周期长的对象仍然存在，效率低；



# 年轻代

---

- 新生成的对象首先都是分配在年轻代的;
- 年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象;





# JVM年轻代的结构

- Eden: Eden用来存放JVM刚分配的对象;
- Survivor1 和Survivor2: 两个Survivor空间一样大, 当Eden中的对象经过垃圾回收没有被回收掉时, 会在两个Survivor之间来回 Copy, 当满足某个条件, 比如Copy次数, 就会被Copy到Tenured;



# 年轻代内存的设置

## ➤两个重要参数:

- $\text{SurvivorRatio} = \text{eden} / \text{from} = \text{eden} / \text{to}$  伊甸园空间和幸存者空间的比值;
- $\text{NewRatio} = \text{tenured} / \text{young}$  老年代内存和年轻代内存比值;

## ➤设置策略:

- 尽可能将对象预留在新生代;
- 减少老年代GC次数;
- 演示NewSizeDemo1;



# 老年代

---

- 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到老年代；
- 老年代中存放的都是一些生命周期较长的对象，例如： Session对象、Socket连接等；

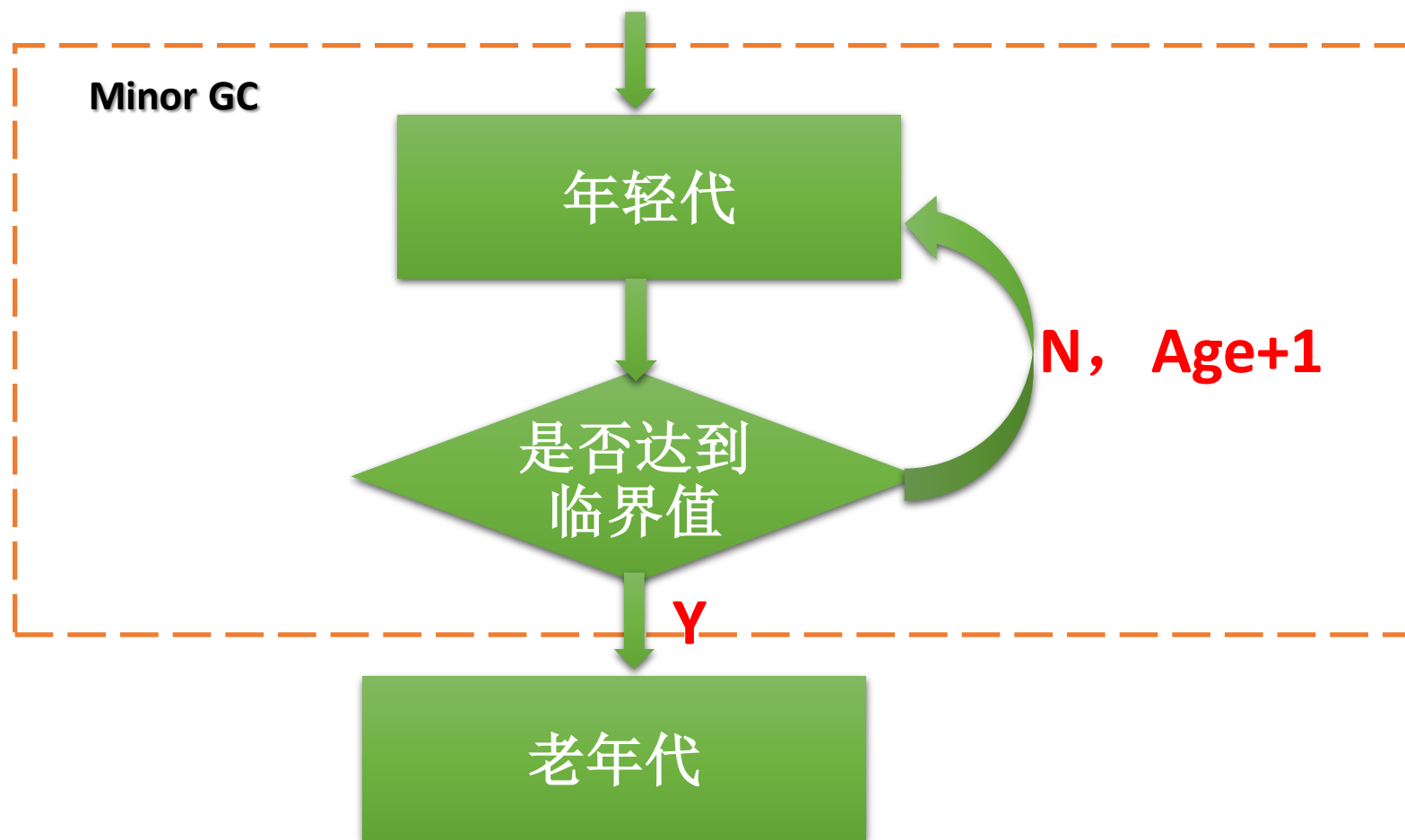


# MaxTenuringThreshold

- MaxTenuringThreshold是年轻代和老年代的临界值；
- MaxTenuringThreshold用于控制对象能经历多少次Minor GC才晋升到旧生代；
- 每发生一次Minor GC，年龄就增加1岁，当它的年龄增加到临界值（默认为15岁），就将会被晋升到老年代中；



# 年轻代向老年代的转变



# 永久代（持久代）内存

- 永久代内存用于保存类信息。方法区的大小决定了系统可以保存多少个类；
- 如果定义了太多的类，会导致永久代内存溢出，参考代码示例 PermTest ；
- 在JDK1.8中，废弃永久代，取而代之的是元数据区(Metaspace) ；



# 直接内存

- 直接内存常用于使用NIO的场景，例如：mina，netty框架;参考DirectBufferOOM、ByteBuffer；
- 直接内存跳过了java堆，使java程序可以直接访问原生内存空间，因此，直接内存访问速度会快于堆内存；
- 直接内存适合申请次数较少，访问较频繁的场所。如果内存空间本身需要平衡申请，则不适合使用直接内存，参考代码示例： AccessDirectBuffer和AccessDirectBuffer2；



# JVM基础知识及性能调优

**JVM**基本结构

**JVM**的重要概念、相关参数、日志

**JVM**工具

垃圾回收算法、垃圾收集器

**JVM**常见错误及调优示例





# Minor GC、Full GC

- 新生代GC( Minor GC ) : 指发生在新生代的GC, Minor GC非常频繁, 一般回收速度也比较快;
- 老年代GC ( Full GC / Major GC ):指发生在老年代的GC, Major GC 的速度一般要比Minor GC 慢10倍以上;
- 一般而言, 新生代回收的频率高, 但是每次回收的耗时都很短, 而老年代回收的频率低, 但是会消耗更多的时间(演示NewSizeDemo2及visualVM);



# JVM基本参数

参数名称	含义	默认值
-Xms	初始堆大小	物理内存的1/64(<1GB)
-Xmx	最大堆大小	物理内存的1/4(<1GB)
-Xmn	年轻代大小(1.4or 1.6or)	
-XX:NewSize	设置年轻代大小(for 1.3/1.4)	
-XX:MaxNewSize	年轻代最大值(for 1.3/1.4)	
-XX:PermSize	设置永久代(perm gen)初始值	物理内存的1/64
-XX:MaxPermSize	设置永久代最大值	物理内存的1/4
-Xss	每个线程的堆栈大小	
-XX:NewRatio	年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)	
-XX:SurvivorRatio	Eden区与Survivor区的大小比值	
-XX:MaxDirectMemorySize	设置最大可用直接内存大小	



# 垃圾收集器参数

参数名称	含义
-XX:+UseSerialGC	串行垃圾回收器
-XX:+UseParallelGC	并行垃圾回收器
-XX:ParallelGCThreads	并行收集器的线程数
-XX:+UseParallelOldGC	年老代垃圾收集方式为并行收集
-XX:+UseConcMarkSweepGC	设置年老代为并发收集
-XX:+UseParNewGC	设置年轻代为并行收集
-XX:+UseG1GC	使用 <b>G1</b> 垃圾回收器



# JVM日志参数

参数名称	含义
-XX:+PrintGC	输出GC日志
-XX:+PrintGCDetails	输出GC的详细日志
-XX:+PrintGCTimeStamps	输出GC的时间戳（以基准时间的形式）
-XX:+PrintGCDateStamps	输出GC的时间戳（以日期的形式输出
-XX:+PrintHeapAtGC	在进行GC的前后打印出堆的信息



# GC日志

- 典型的GC日志:
- 33.125: [GC [DefNew: 3324K->152K(3712K), 0.0025925 secs] 3324K->152K(11904K), 0.0031680 secs]
- 100.667: [Full GC [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K), [Perm : 2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
- 最前面的数字“33.125:”和“100.667:”代表了GC发生的时间, 这个数字的含义是从Java虚拟机启动以来经过的秒数。

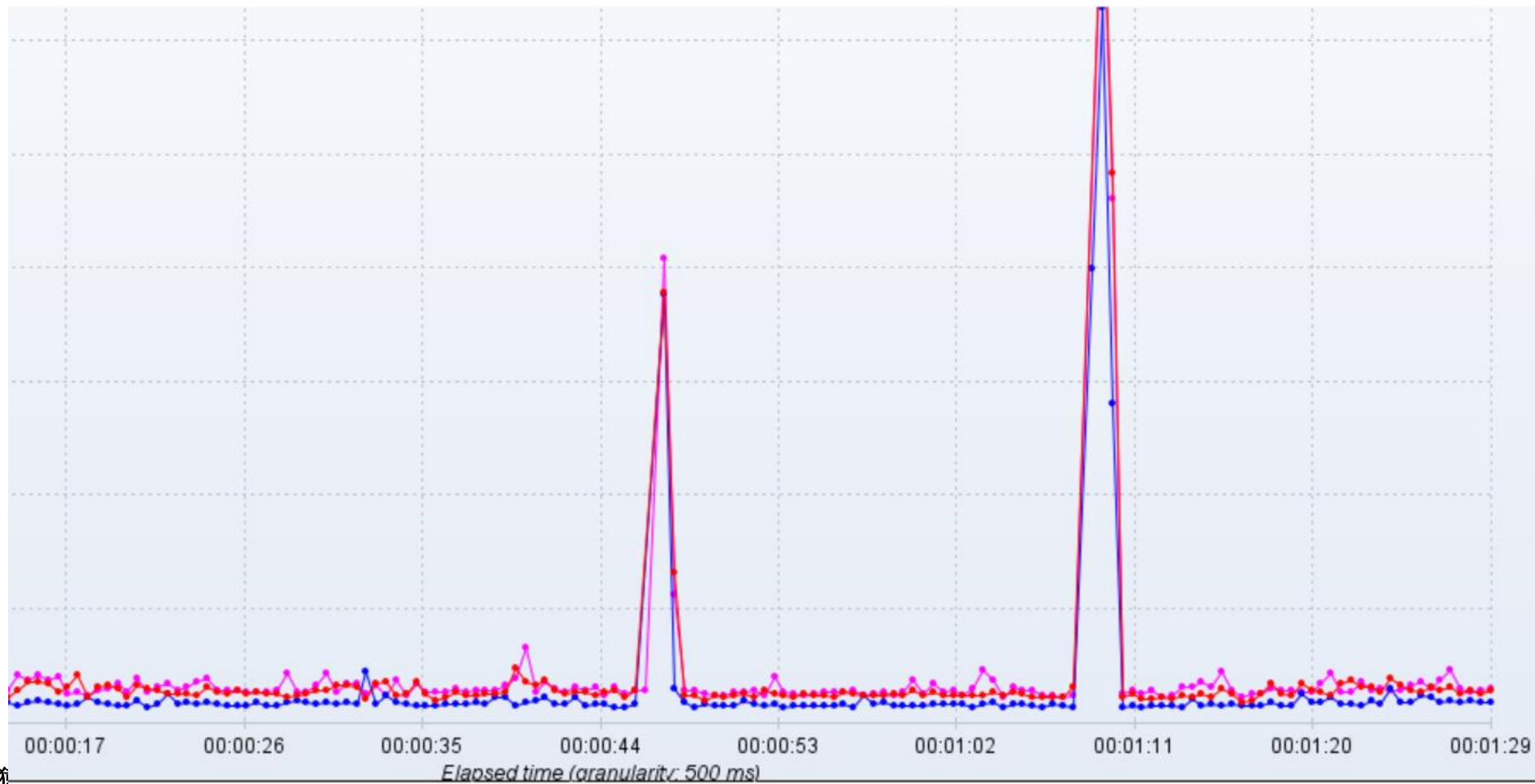


# GC日志

- “ [DefNew”、 “ [Tenured”、 “ [Perm”表示GC发生的区域，这里显示的区域名称与使用的GC收集器是密切相关的，例如上面样例所使用的Serial收集器中的新生代名为“Default New Generation”，所以显示的是“ [DefNew”。如果是ParNew收集器，新生代名称就会变为“ [ParNew”，意为“Parallel New Generation”。如果采用Parallel Scavenge收集器，那它配套的新生代称为“PSYoungGen”，老年代和永久代同理，名称也是由收集器决定的。



# 接口响应时间过长



# 日志分析-Full GC

```
PS Old Generation
capacity = 1610612736 (1536.0MB)
used      = 1610612464 (1535.999740600586MB)
free      = 272 (2.593994140625E-4MB)
99.99998311201732% used
```

```
PS Perm Generation
capacity = 115408896 (110.0625MB)
used      = 115119128 (109.7861557006836MB)
free      = 289768 (0.27634429931640625MB)
99.74892056848027% used
```

```
[root@SHA0ITTES002 nmon]# jstat -gc 12538 5000
12538 not found
```

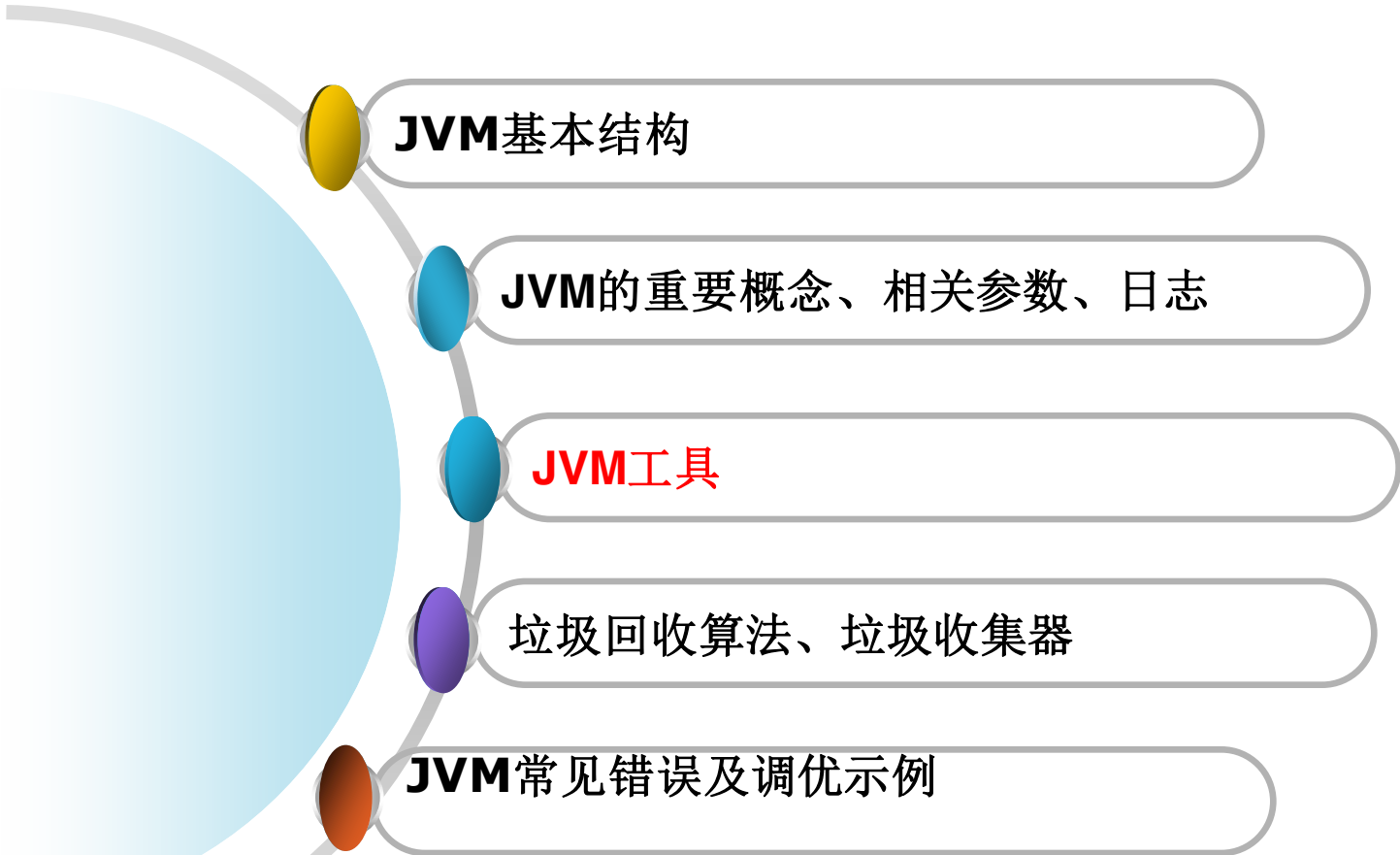
```
[root@SHA0ITTES002 nmon]# jstat -gc 15721 5000
```

S0C	S1C	S0U	S1U	EC	EU	OC	OU	PC	PU	YGC	YGCT	FGC	FGCT	GCT
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572863.8	112768.0	112395.8	5697	75.352	105	554.680	630.032
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572863.8	112768.0	112414.4	5697	75.352	106	560.832	636.185
40640.0	41344.0	0.0	0.0	440256.0	184532.8	1572864.0	1572863.8	112768.0	112420.8	5697	75.352	106	567.115	642.467
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572863.8	112768.0	112433.1	5697	75.352	107	567.115	642.467
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572863.8	112768.0	112496.8	5697	75.352	108	573.312	648.664
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572863.8	112768.0	112379.4	5697	75.352	108	573.312	648.664
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112394.0	5697	75.352	109	581.441	656.793
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112420.0	5697	75.352	110	587.658	663.010
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112420.0	5697	75.352	110	587.658	663.010
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112436.2	5697	75.352	111	593.865	669.217
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112458.7	5697	75.352	112	600.158	675.510
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112768.0	112458.7	5697	75.352	112	600.158	675.510
40640.0	41344.0	0.0	0.0	440256.0	440256.0	1572864.0	1572864.0	112704.0	112404.4	5697	75.352	113	608.221	683.574





# JVM基础知识及性能调优



**JVM**基本结构

JVM的重要概念、相关参数、日志

**JVM**工具

垃圾回收算法、垃圾收集器

**JVM**常见错误及调优示例



# JVM性能相关分析工具

---

➤ 可视化性能监控工具 Visual VM

➤ 内存分析工具 MAT

➤ 图形化虚拟机监控工具 Jconsole



# VisualVM介绍

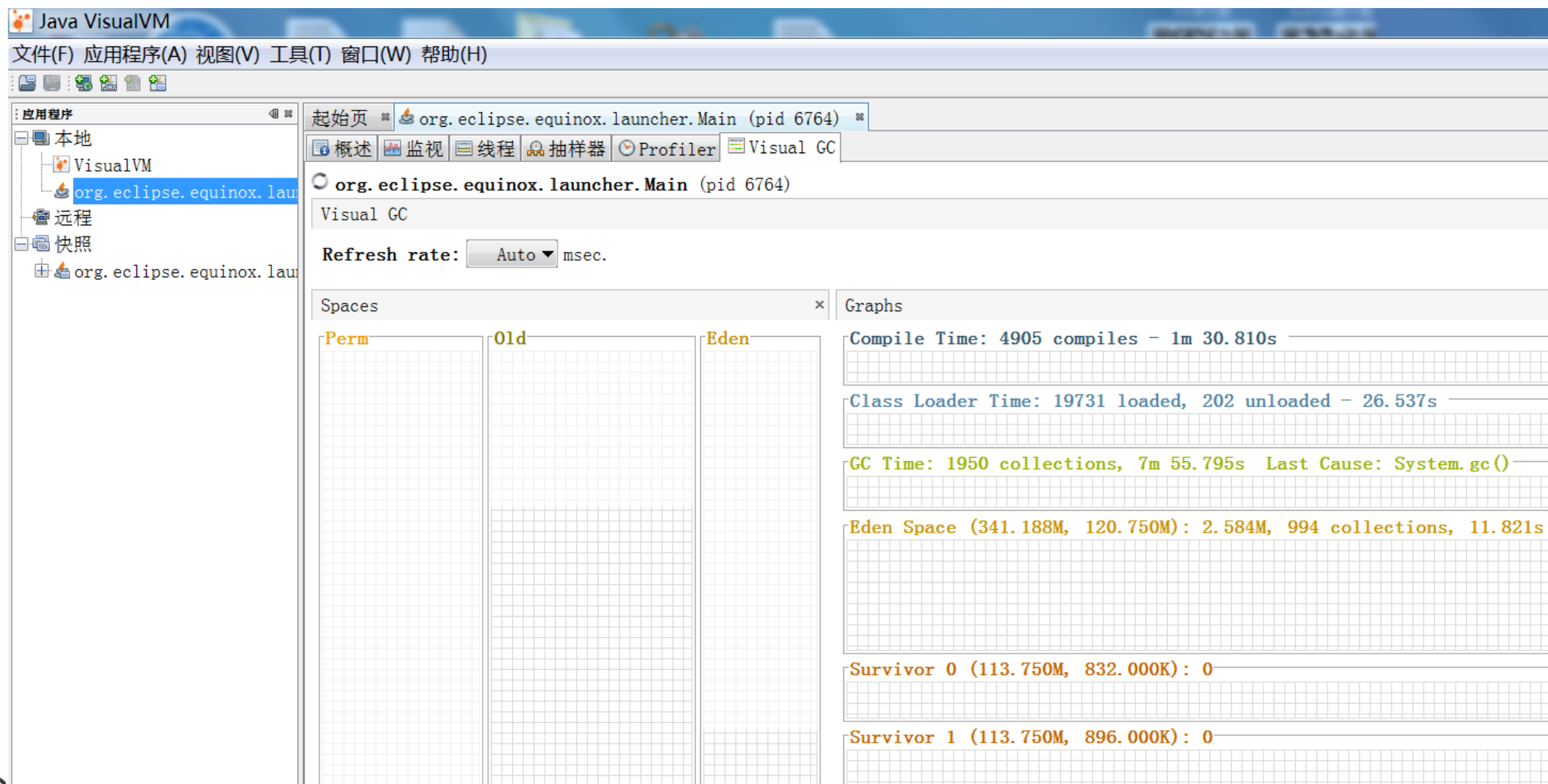
---

➤ VisualVM官网 <http://visualvm.java.net>

➤ Java VisualVM默认没有安装Visual GC插件，需要手动安装，JDK的安装目录的bin目录下双击jvisualvm.exe



# VisualVM介绍



# 运用VisualVM调优

- 运用VisualVM的快照查看耗时方法的具体时间及调用次数、CPU、内存占用情况，分析问题，解决问题；
- 参考示例：CpuTest和历史邮件；

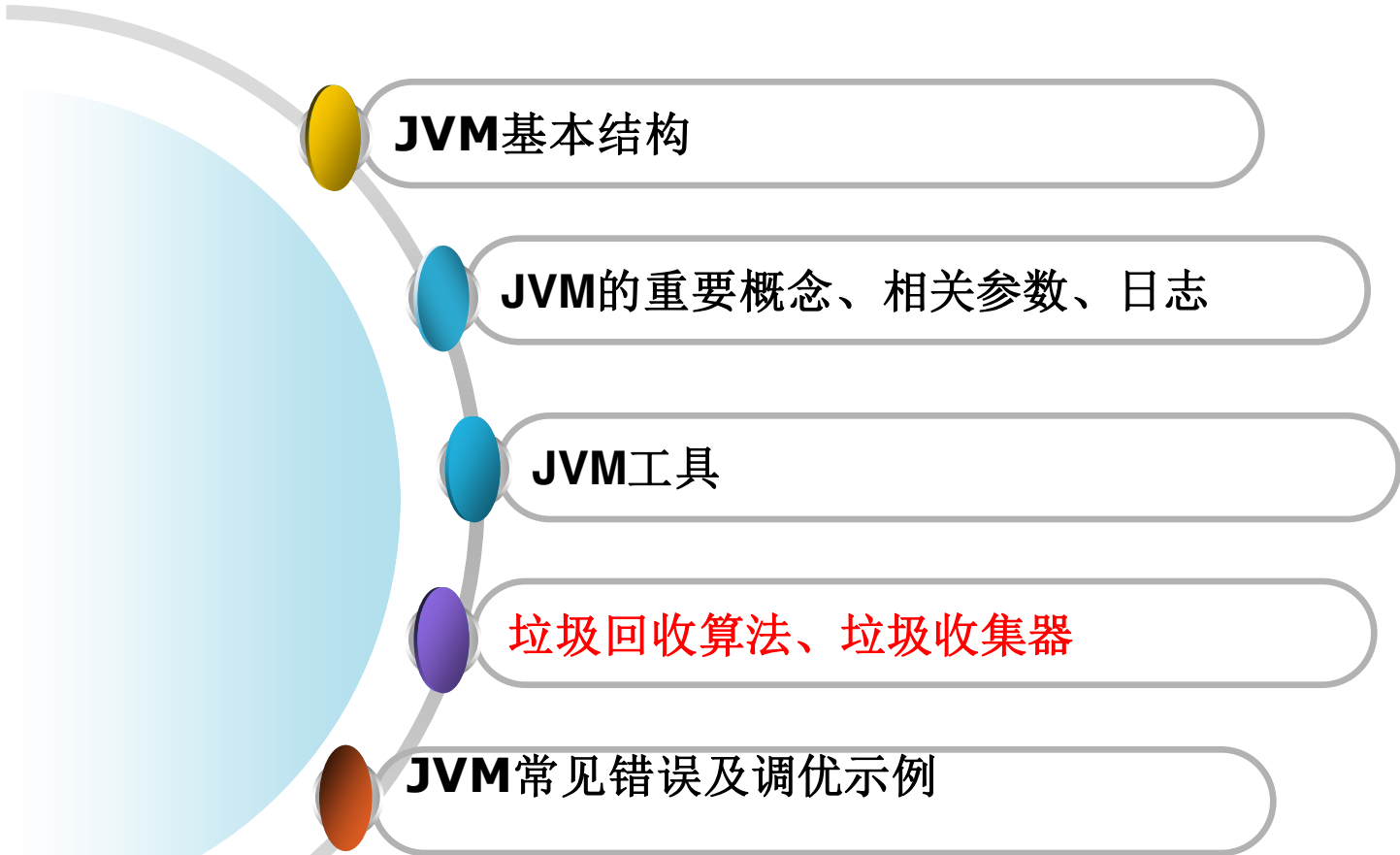
Profiler 快照

视图: 方法

调用树 - 方法	时间 [%] ▼	时间	时间 (CPU)	调用
main		73186 ms (100%)	73186 ms	1
com.yhd.jvm.CpuTest.main ()		73186 ms (100%)	73186 ms	2
com.yhd.jvm.CpuTest.endlessLoop ()		73186 ms (100%)	73186 ms	2
java.io.PrintStream.println ()		72999 ms (99.7%)	72999 ms	4
自用时间		187 ms (0.3%)	187 ms	2
自用时间		0.000 ms (0%)	0.000 ms	2



# JVM基础知识及性能调优



**JVM**基本结构

**JVM**的重要概念、相关参数、日志

**JVM**工具

垃圾回收算法、垃圾收集器

**JVM**常见错误及调优示例



# 垃圾回收算法

---

- 复制算法（ Copying ）
- 标记清除法（ Mark-Sweep ）
- 标记压缩法（ Mark-Compat ）
- 分代算法（ Generational Collection ）



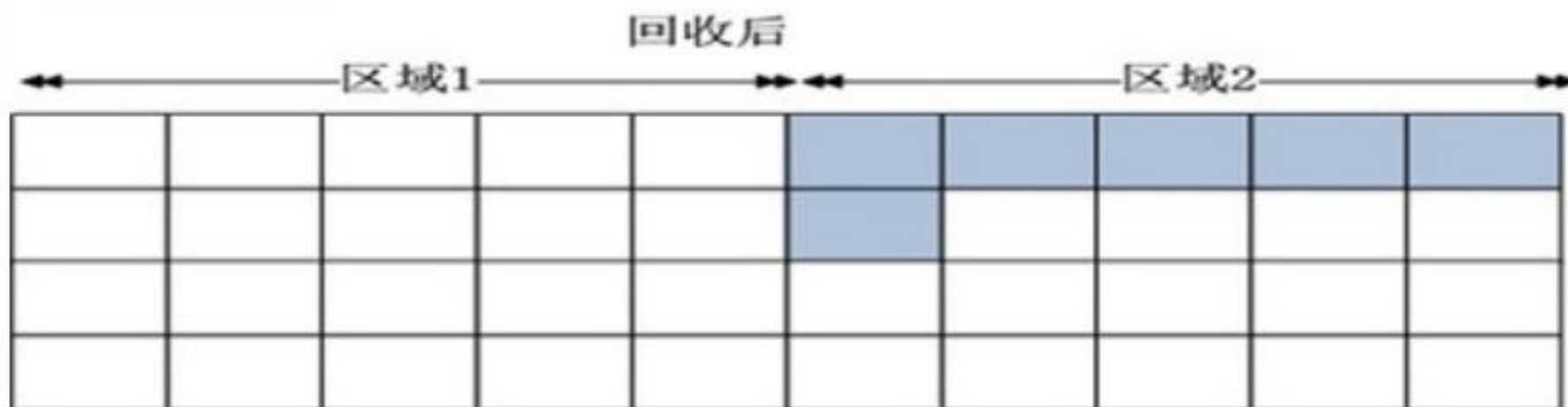
# 复制算法（Copying）

- 复制算法把内存分配为两个空间，一个空间（A）用来负责装载正常的对象信息，另外一个内存空间（B）是垃圾回收用的；
- 每次把空间A中存活的对象全部复制到空间B里面，在一次性的把空间A删除；
- 复制算法对内存要求比较大，内存的利用率比较低。适用于短生存期的对象，持续复制长生存期的对象则导致效率降低；





# 复制算法 (Copying)



可回收对象

可用内存

存活对象



# 标记清除法 ( Mark-Sweep )

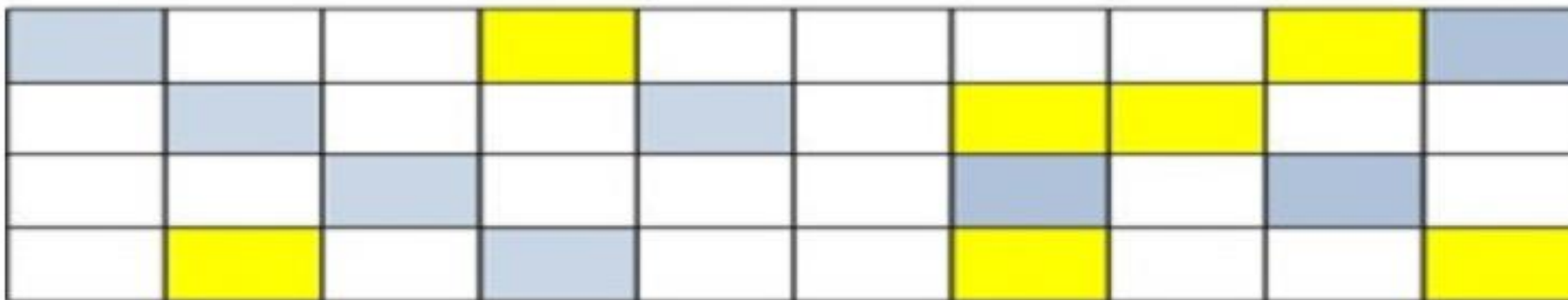
---

- 标记—清除算法包括两个阶段：“标记”和“清除”。在标记阶段，确定所有要回收的对象，并做标记。清除阶段紧随标记阶段，将标记阶段确定不可用的对象清除；
- 标记—清除算法是基础的收集算法，标记和清除阶段的效率不高，而且清除后会产生大量的不连续空间，这样当程序需要分配大内存对象时，可能无法找到足够的连续空间；

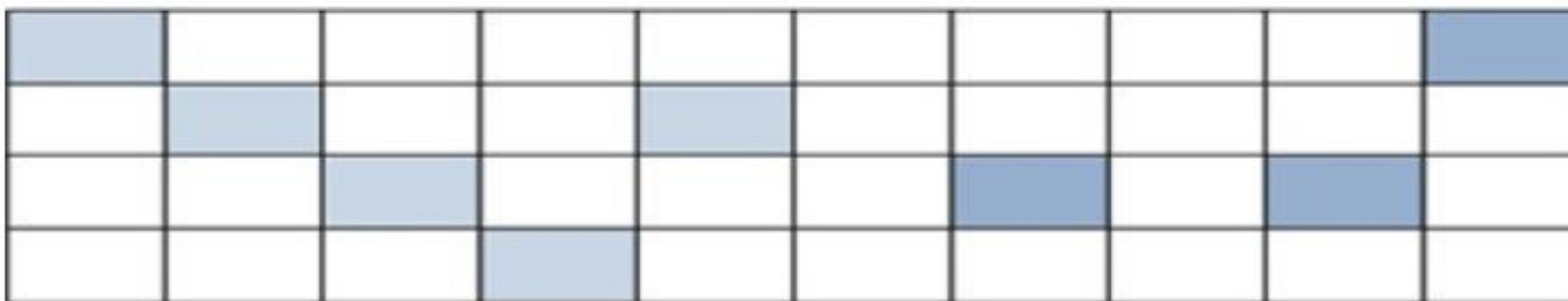


# 标记清除法 ( Mark-Sweep )

回收前



回收后



可回收对象

可用内存

存活对象



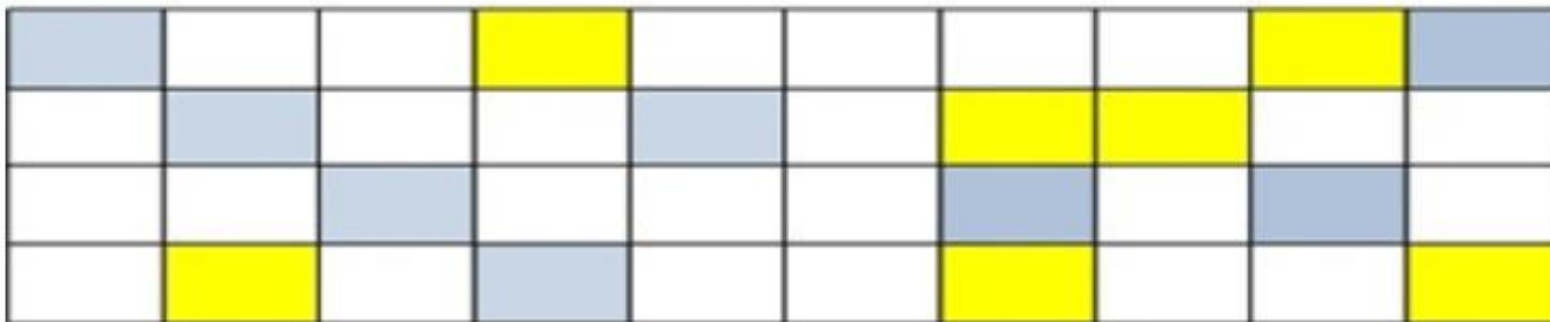
# 标记压缩法（Mark-Compat）

- 标记—整理算法是在标记-清除的算法之上进行一下压缩空间，重新移动对象的过程。但是标记—整理算法不是把存活对象复制到另一块内存，而是把存活对象往内存的一端移动，然后直接回收边界以外的内存；
- 标记—整理算法提高了内存的利用率，并且它适合在收集对象存活时间较长的老年代；

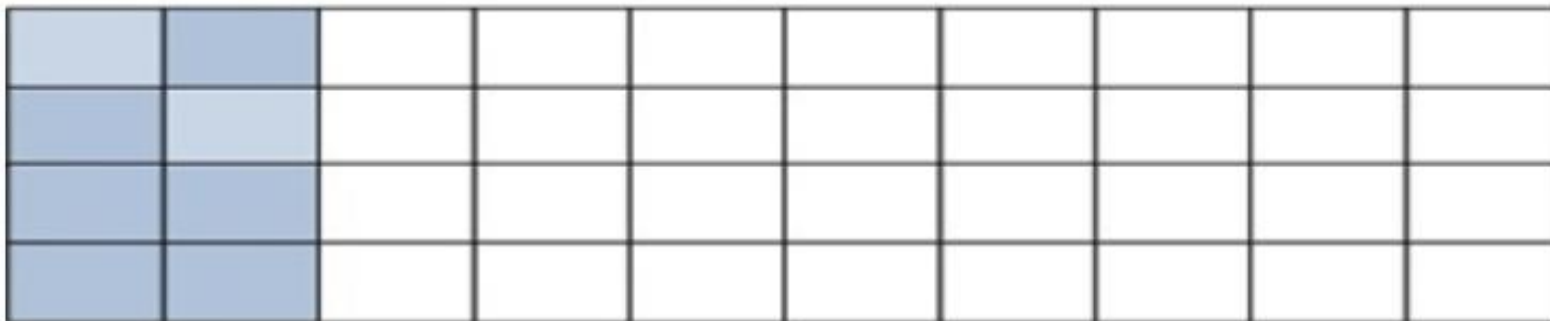


# 标记压缩法 (Mark-Compat)

回收前



回收后



可回收对象

可用内存

存活对象



# 分代算法

## 内存分代

新生代（复制算法）

老年代（标记清除算法、标记压缩算法）



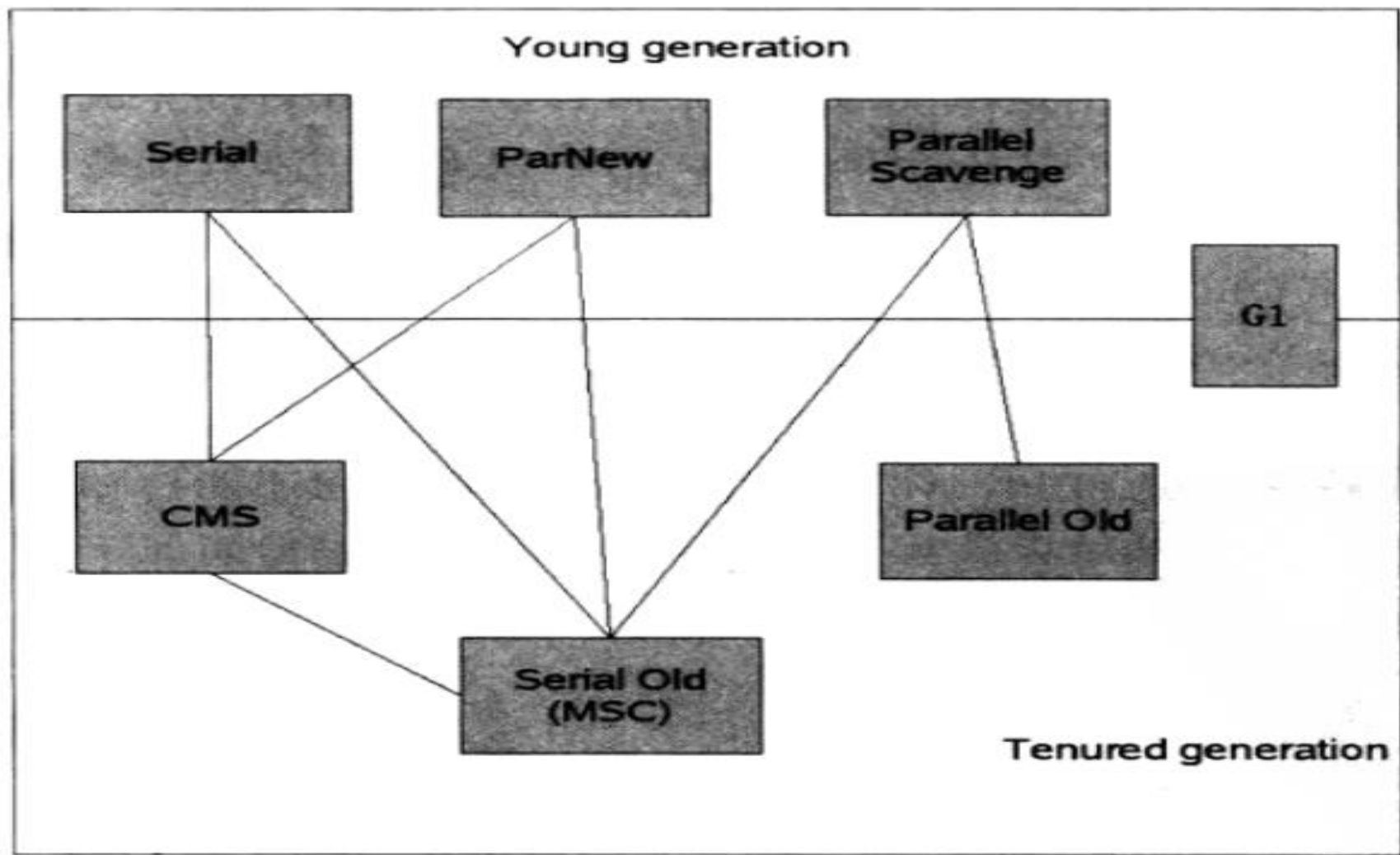
# 垃圾回收算法 VS 垃圾回收器

内存回收算法理论  
垃圾回收算法

内存回收具体实现  
垃圾回收器

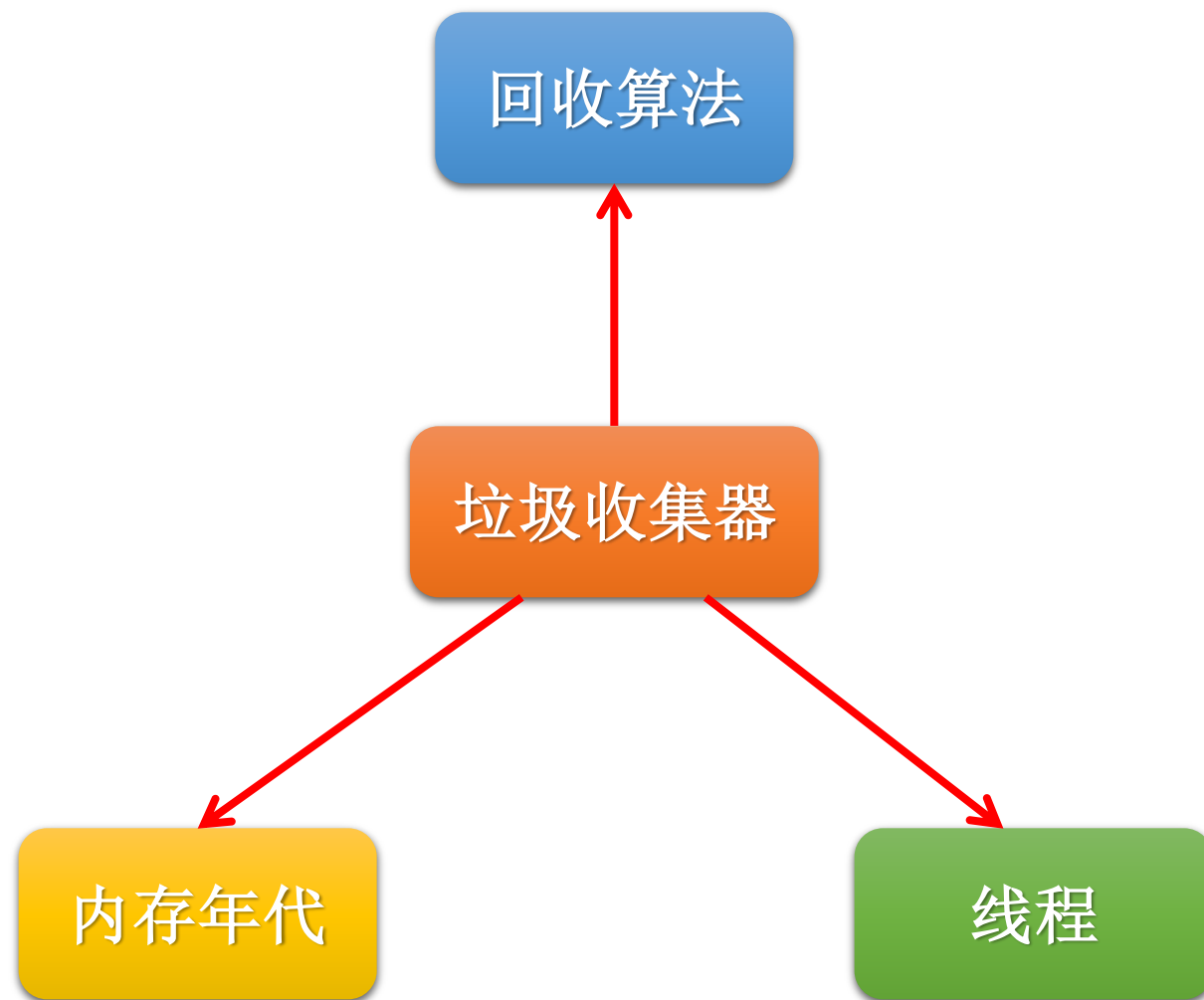


# 垃圾回收器





# 从三个维度理解垃圾回收器



# Serial收集器

- 单线程的收集器；
- 对于单个CPU环境，Serial收集器由于没有线程交互的开销可以获得最高的单线程收集效率；
- 它是虚拟机运行在Client模式下的默认新生代收集器；
- 缺点是因为单线程GC，会造成中断的时间（Stop-the-world）比较大；



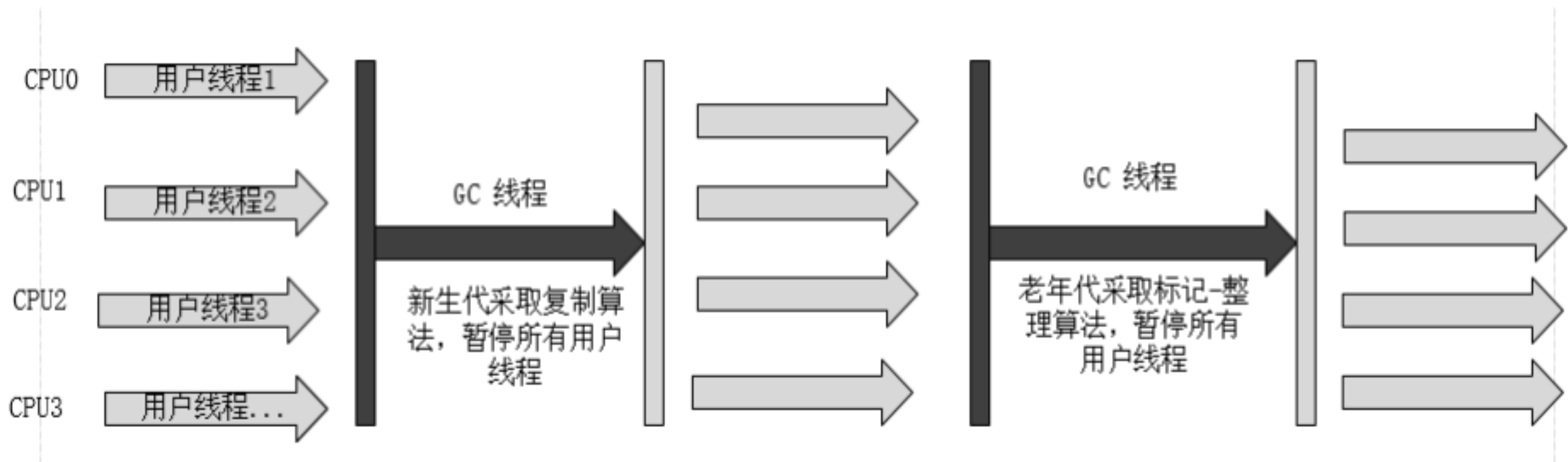
# Serial Old 收集器

---

- Serial Old是Serial 收集器的老年代版本；
- Serial Old 同样是单线程收集器，使用“标记压缩”算法；
- 它是虚拟机运行在Client模式下的默认新生代收集器；



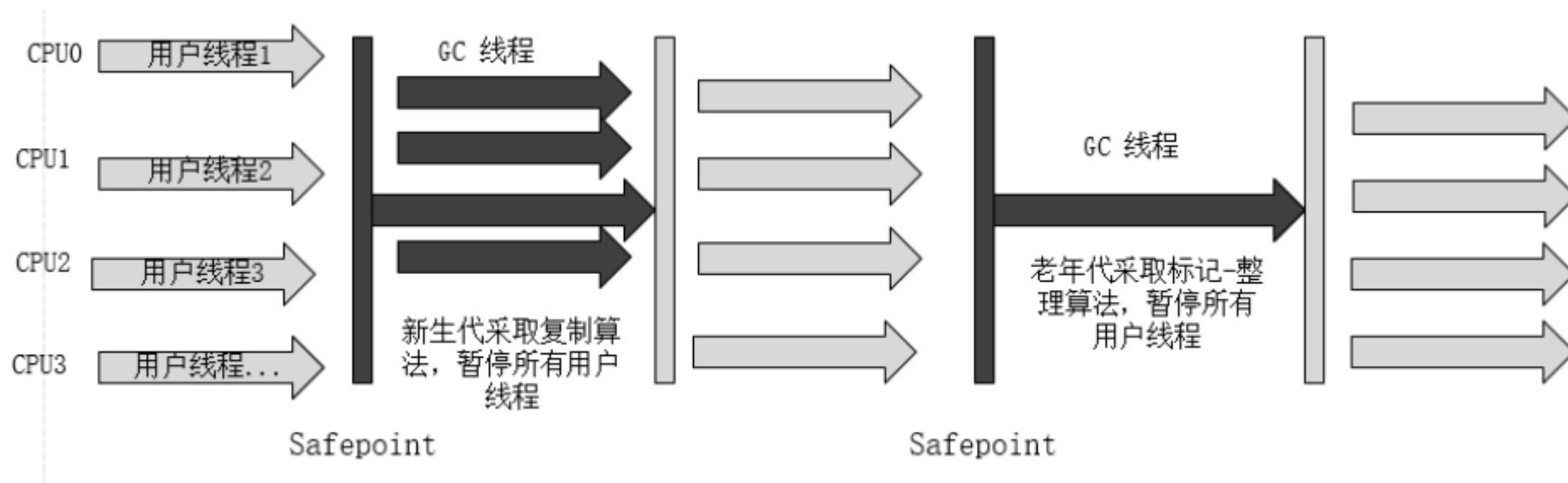
# Serial/ Serial Old收集器运行示意图



# ParNew 收集器

➤ ParNew收集器是Serial收集器的多线程版本；

➤ ParNew收集器运行示意图：

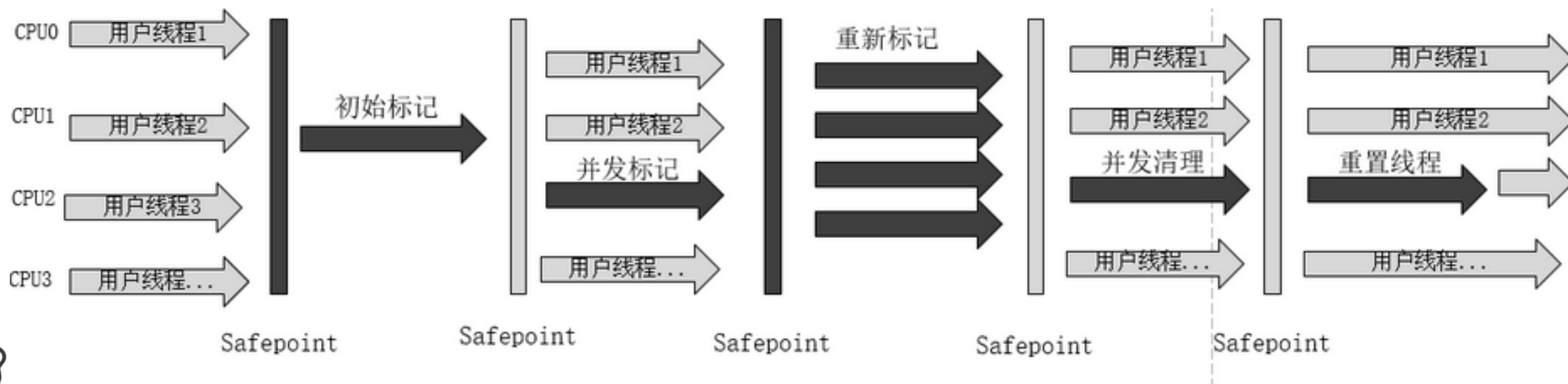


# CMS收集器

➤CMS( Concurrent Mark Sweep ) 收集器是以获得最短响应时间为目标的收集器;

➤CMS过程: 初始标记、并发标记、重新标记、并发清除;

➤CMS收集器运行示意图:

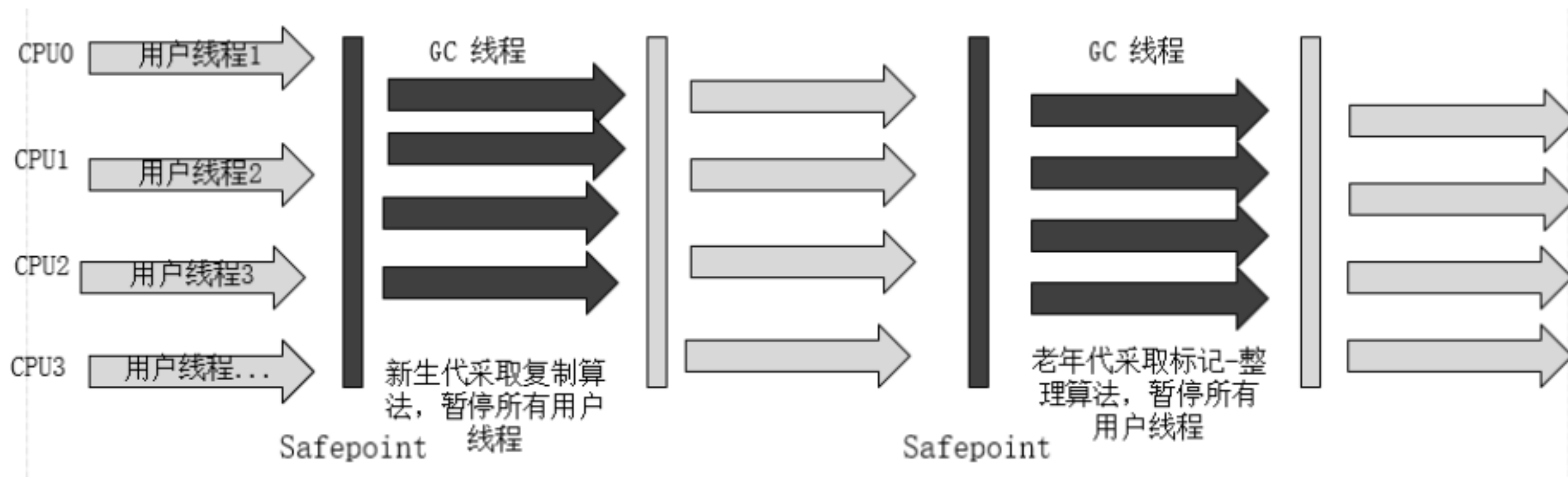


# Parallel Scavenge/Old 收集器

- Parallel Scavenge用于新生代回收， Parallel Old用于老年代回收，复制算法，并行收集；
- 与ParNew的不同是它的关注点不同，它可以精确的控制吞吐量，例如：JVM共运行了100min。其中垃圾收集花掉1min，那吞吐量就是99%；



# Parallel Scavenge/Old收集器



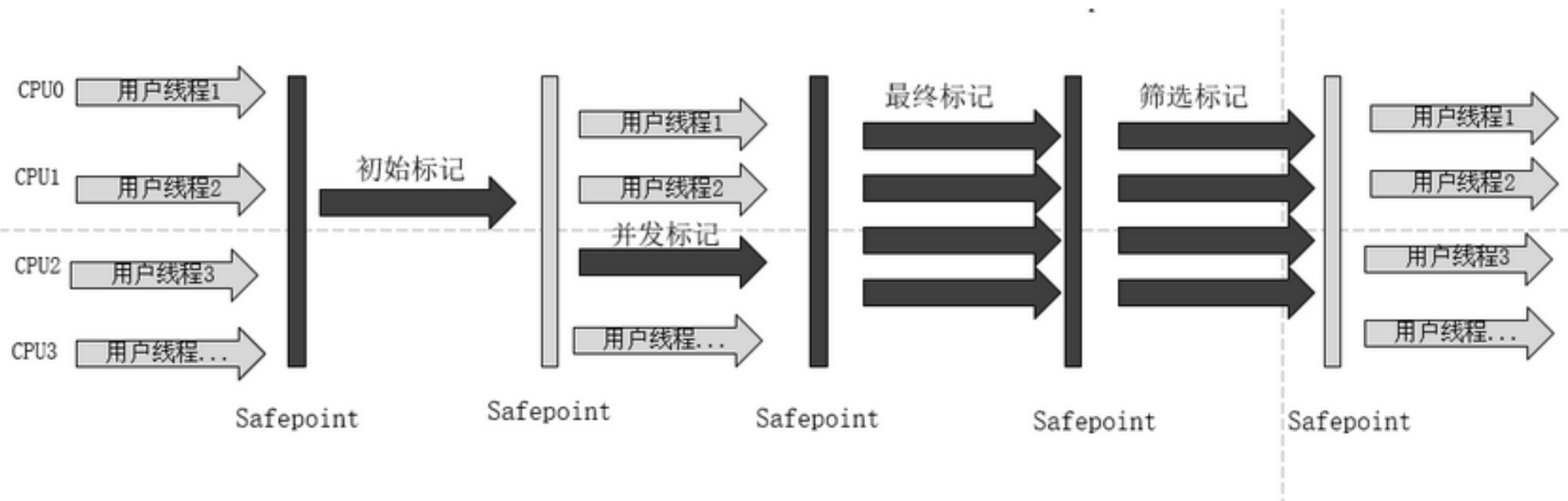


# G1收集器

- G1收集器是一款面向服务端应用的垃圾收集器;
- 并行与并发: G1能充分利用多CPU、多核环境下的硬件优势, 使用多个CPU来缩短Stop-The-World停顿时间;
- 分代收集: G1收集器可收集新生代与老年代两种, 不需要其他收集器配合就可以独立管理整个GC堆;
- 空间整合: G1采用“标记-整理”算法实现收集器, 意味着G1运作期间不会产生内存空间碎片, 收集后可提供规整的可用内存;
- 可预测的停顿: 建立可预测的停顿时间模型, 能让使用者明确指定在一个长度为M毫秒的时间片段内, 消耗在垃圾收集器上的时间不得超过N毫秒;



# G1收集器



# JVM参数配置示例（tomcat）

---

```
JAVA_OPTS='-server -Xms6144m -Xmx6144m -Xmn2048m -XX:PermSize=256m -  
XX:MaxPermSize=256m -XX:MaxTenuringThreshold=15 -XX:GCTimeRatio=19 -  
XX:+DisableExplicitGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -  
XX:+CMSPermGenSweepingEnabled -XX:+UseCMSCompactAtFullCollection -  
XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:-  
CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=70 -  
XX:SoftRefLRUPolicyMSPerMB=0 -Dglobal.config.path=/var/www/webapps/config/'
```



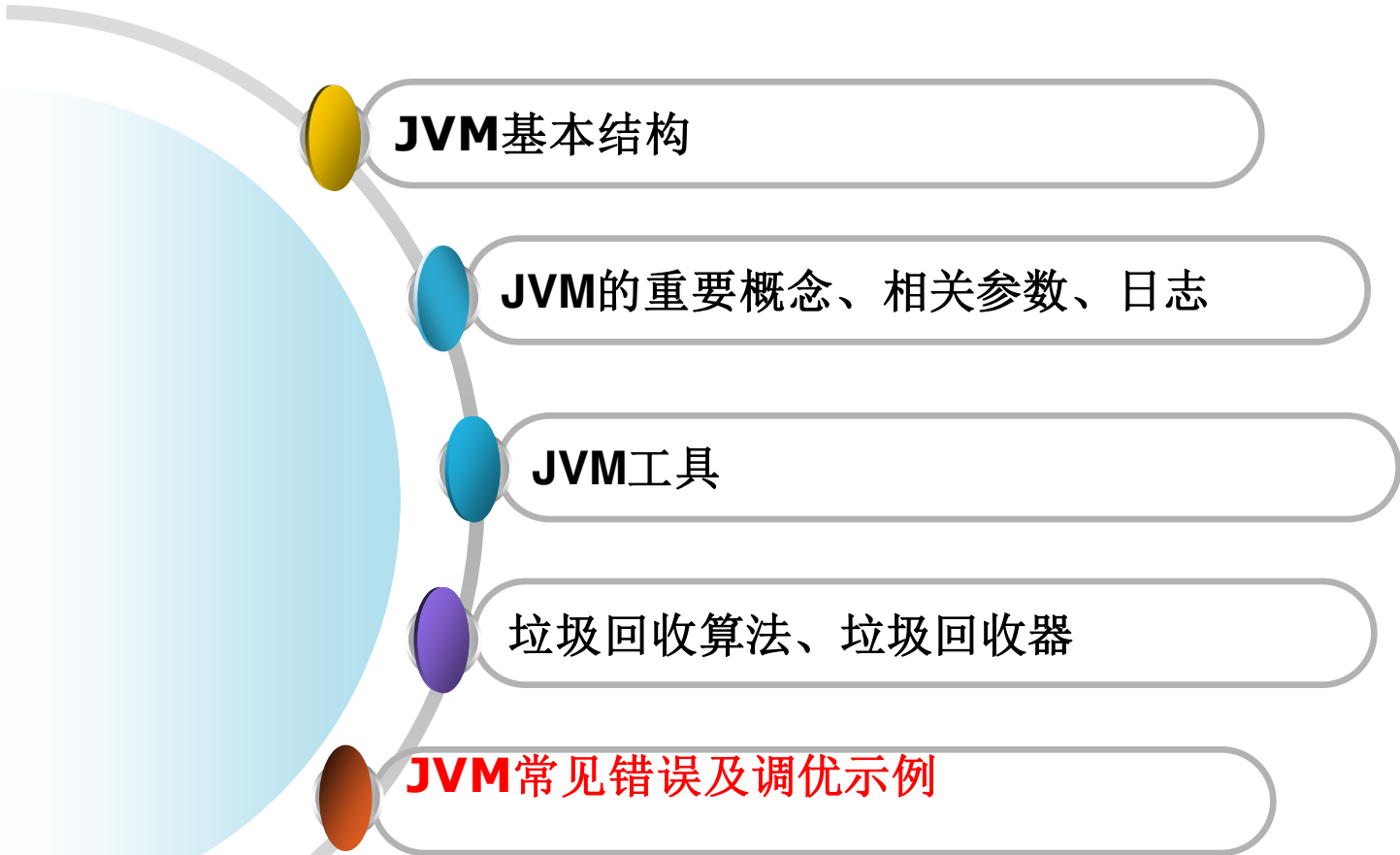
# JVM参数配置示例（tomcat）

---

```
JAVA_OPTS='-server -Xms4g -Xmx4g -XX:+UseConcMarkSweepGC -  
XX:+CMSIncrementalMode -XX:NewSize=512m -XX:MaxPermSize=256m -  
Dglobal.config.path=/var/www/webapps/config/ -  
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/var/www/webapps/'
```



# JVM性能调优



**JVM**基本结构

JVM的重要概念、相关参数、日志

JVM工具

垃圾回收算法、垃圾回收器

**JVM**常见错误及调优示例



## 永久代内存溢出(OutOfMemoryError:PermGen space)

```
2016-07-18T14:29:33.108+0800: [Full GC [PSYoungGen: 64K->0K(478208K)] [PSOldGen: 182K->182K(1048576K)
2016-07-18T14:29:33.108+0800: [GC [PSYoungGen: 0K->0K(472704K)] 182K->182K(1521280K), 0.0001252 secs]
2016-07-18T14:29:33.108+0800: [Full GC [PSYoungGen: 0K->0K(472704K)] [PSOldGen: 182K->182K(1048576K)]
Exception in thread "Reference Handler" java.lang.OutOfMemoryError: PermGen space
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:123)
2016-07-18T14:29:33.124+0800: [GC [PSYoungGen: 18905K->64K(516288K)] 19088K->246K(1564864K), 0.000746
2016-07-18T14:29:33.124+0800: [Full GC [PSYoungGen: 64K->0K(516288K)] [PSOldGen: 182K->189K(1048576K)
2016-07-18T14:29:33.124+0800: [GC [PSYoungGen: 0K->0K(524032K)] 189K->189K(1572608K), 0.0001734 secs]
2016-07-18T14:29:33.124+0800: [Full GC [PSYoungGen: 0K->0K(524032K)] [PSOldGen: 189K->188K(1048576K)]
```



# 永久代内存溢出解决方法

- 代码中存在不断创建类（**注意是类不是对象实例**）的死循环；
- jar包过多或者jar包重复，加载了大量class文件；
- 永久代内存大小偏小，增加JVM的PermSize和MaxPermSize参数大小；
- Eclipse和tomcat容器的启动时经常会出现类似的错误；
- **参考PermTest、PermTest2、PermTest3、eclipse.ini文件，邮件调优示例；**



# 堆内存溢出(OutOfMemoryError: Java heap space)

```
Java HotSpot(TM) 64-Bit Server VM warning: MaxNewSize (1048576k) is equal to  
java.lang.OutOfMemoryError: Java heap space  
Dumping heap to java_pid7104.hprof ...  
Heap dump file created [23172140 bytes in 0.182 secs]  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
    at java.util.Arrays.copyOf(Arrays.java:2760)  
    at java.util.Arrays.copyOf(Arrays.java:2734)  
    at java.util.ArrayList.ensureCapacity(ArrayList.java:167)  
    at java.util.ArrayList.add(ArrayList.java:351)  
    at com.yhd.jvm.HeapOOM.main(HeapOOM.java:31)
```





# 堆内存溢出解决方法

- java虚拟机创建的对象太多，在进行垃圾回收之间，虚拟机分配的到堆内存空间已经用满了，与Heap space有关；
- 增加JVM的Xms（初始堆大小）和Xmx（最大堆大小）参数的大小；
- 检查是否有直接内存的使用、直接内存大小是否合适；
- 是否有死循环或不必要地重复创建大量对象。找到原因后，修改程序和算法；参考代码示例：HeapOOMTest；



# 直接内存溢出

```
[GC [PSYoungGen: 5244K->256K(305856K)] 5244K->256K(1004928K), 0.0004701 secs]
[Full GC (System) [PSYoungGen: 256K->0K(305856K)] [PSOldGen: 0K->163K(699072K)]
Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory
    at java.nio.Bits.reserveMemory(Bits.java:632)
    at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:97)
    at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:288)
    at com.yhd.jvm.DirectBufferOOM.main(DirectBufferOOM.java:22)
```



# 直接内存溢出解决方法

---

- 分析堆大小，直接内存大小是否合理；
- 设置合理的-XX:MaxDirectMemorySize也可以避免意外的内存溢出发生；
- 参考示例： DirectBufferOOM；



# 过多线程导致内存溢出

附上报错日志:

```
Exception in thread "analysisSkuThreadPool-3" java.lang.OutOfMemoryError: unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:640)
    at sun.net.www.http.KeepAliveCache$1.run(KeepAliveCache.java:89)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.net.www.http.KeepAliveCache.put(KeepAliveCache.java:75)
    at sun.net.www.http.HttpClient.putInKeepAliveCache(HttpClient.java:364)
    at sun.net.www.http.HttpClient.finished(HttpClient.java:352)
    at sun.net.www.http.KeepAliveStream.close(KeepAliveStream.java:87)
    at sun.net.www.MeteredStream.justRead(MeteredStream.java:75)
    at sun.net.www.MeteredStream.read(MeteredStream.java:117)
    at java.io.FilterInputStream.read(FilterInputStream.java:116)
    at sun.net.www.protocol.http.HttpURLConnection$HttpInputStream.read(HttpURLConnection.java:2672)
    at sun.net.www.protocol.http.HttpURLConnection$HttpInputStream.read(HttpURLConnection.java:2667)
    at sun.net.www.protocol.http.HttpURLConnection$HttpInputStream.read(HttpURLConnection.java:2656)
    at com.yihaodian.architecture.hedwig.common.hessian.HedwigHessianInput.read(HedwigHessianInput.java:1553)
    at com.yihaodian.architecture.hedwig.common.hessian.HedwigHessianInput.completeReply(HedwigHessianInput.java:375)
    at com.yihaodian.architecture.hedwig.common.hessian.HedwigHessianProxy.invoke(HedwigHessianProxy.java:163)
    at $Proxy319.queryBaseSupplierByBaseCondition(Unknown Source)
    at sun.reflect.GeneratedMethodAccessor113.invoke(Unknown Source)
```



# OutOfMemoryError: unable to create new native thread

➤本质原因是我们创建了太多的线程，而能创建的线程数是有限制的，导致了异常的发生；

➤能创建的线程数的具体计算公式如下：

$$(\text{MaxProcessMemory} - \text{JVMMemory} - \text{ReservedOsMemory}) / (\text{ThreadStackSize}) = \text{Number of threads}$$

MaxProcessMemory 指的是一个进程的最大内存

JVMMemory JVM内存

ReservedOsMemory 保留的操作系统内存

ThreadStackSize 线程栈的大小



# 多线程导致OOM的解决方法

---

- 合理减少线程数：检查应用中的线程池，多线程使用场景是否合理，切忌随意设置较大的线程数；
- 分析最大堆空间、直接内存大小是否合理；
- 参考邮件调优示例；



# CPU使用率高

```
top - 17:57:40 up 350 days, 2:53, 2 users, load average: 2.08, 1.96, 1.55
Tasks: 142 total, 1 running, 137 sleeping, 4 stopped, 0 zombie
Cpu(s): 13.8%us, 10.8%sy, 0.0%ni, 74.4%id, 0.4%wa, 0.0%hi, 0.6%si, 0.0%st
Mem: 8149296k total, 7585800k used, 563496k free, 440632k buffers
Swap: 20482864k total, 3112k used, 20479752k free, 3095020k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
28555	root	18	0	2356m	796m	9200	S	200.0	10.0	108:46.96	java
19148	root	20	0	1306m	404m	7932	S	0.3	5.1	18:24.97	java
30096	root	15	0	1342m	743m	7952	S	0.3	9.3	48:18.35	java
1	root	15	0	10348	632	540	S	0.0	0.0	0:39.88	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:07.31	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:20.17	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	RT	-5	0	0	0	S	0.0	0.0	0:03.51	migration/1



# CPU使用率过高解决方法

---

- 生成dump文件；
- 使用top命令找到耗cpu的线程（top -H -p PID）；
- 将线程的pid 转成16进制（printf “%x\n” tid ,tid是dump文件中的线程id）；
- 查看dump文件分析，定位代码，解决问题；
- 参考文章：<http://blog.csdn.net/blade2001/article/details/9065985>





# 内存使用率高

```
top - 20:34:08 up 15 days, 8:54, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 61 total, 1 running, 60 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.2%us, 0.2%sy, 0.0%ni, 99.3%id, 0.3%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 2058776k total, 1944636k used, 114140k free, 293544k buffers
Swap: 2048276k total, 0k used, 2048276k free, 849912k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
9004	root	18	0	1727m	495m	10m	S	0.7	24.7	1:31.28	java
1	root	15	0	10372	696	584	S	0.0	0.0	0:01.06	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:05.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:03.40	migration/1
5	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/1
6	root	10	-5	0	0	0	S	0.0	0.0	0:01.09	events/0



# 内存使用率过高解决方法

- 生成dump文件;
- 用工具分析dump文件;
- 定位代码，解决问题;

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.concurrent.ThreadPoolExecutor @ 0x7705b9f18	104	378,005,752
java.util.concurrent.LinkedBlockingQueue @ 0x7727e7d40	80	378,004,984
java.util.concurrent.LinkedBlockingQueue\$Node @ 0x7b115e550	32	378,002,792
java.util.concurrent.LinkedBlockingQueue\$Node @ 0x7b115e580	32	378,002,760
java.util.concurrent.LinkedBlockingQueue\$Node @ 0x7b115e5b0	32	378,001,008
com.taobao.inventory.core.event.control.EventSendMulticaster\$1 @ 0x7b115e598	32	1,720
java.util.ArrayList @ 0x7b12fd030	40	1,688
java.lang.Object[10] @ 0x7b12bd0f8	104	1,648
com.taobao.inventory.core.event.type.SPItemInvalidateEvent @ 0x7b12d4ba0	56	1,256
com.taobao.inventory.core.event.type.InventoryNotifyEvent @ 0x7b12d4bc0	56	200
java.lang.String @ 0x7b12d4be0 test-test-#	40	88



