JAVA内存模型和GC优化

伍振华

提纲

- JVM内存模型
- JVM GC算法和原理
- JVM参数
- JVM GC优化
- 相关工具介绍

JVM内存模型

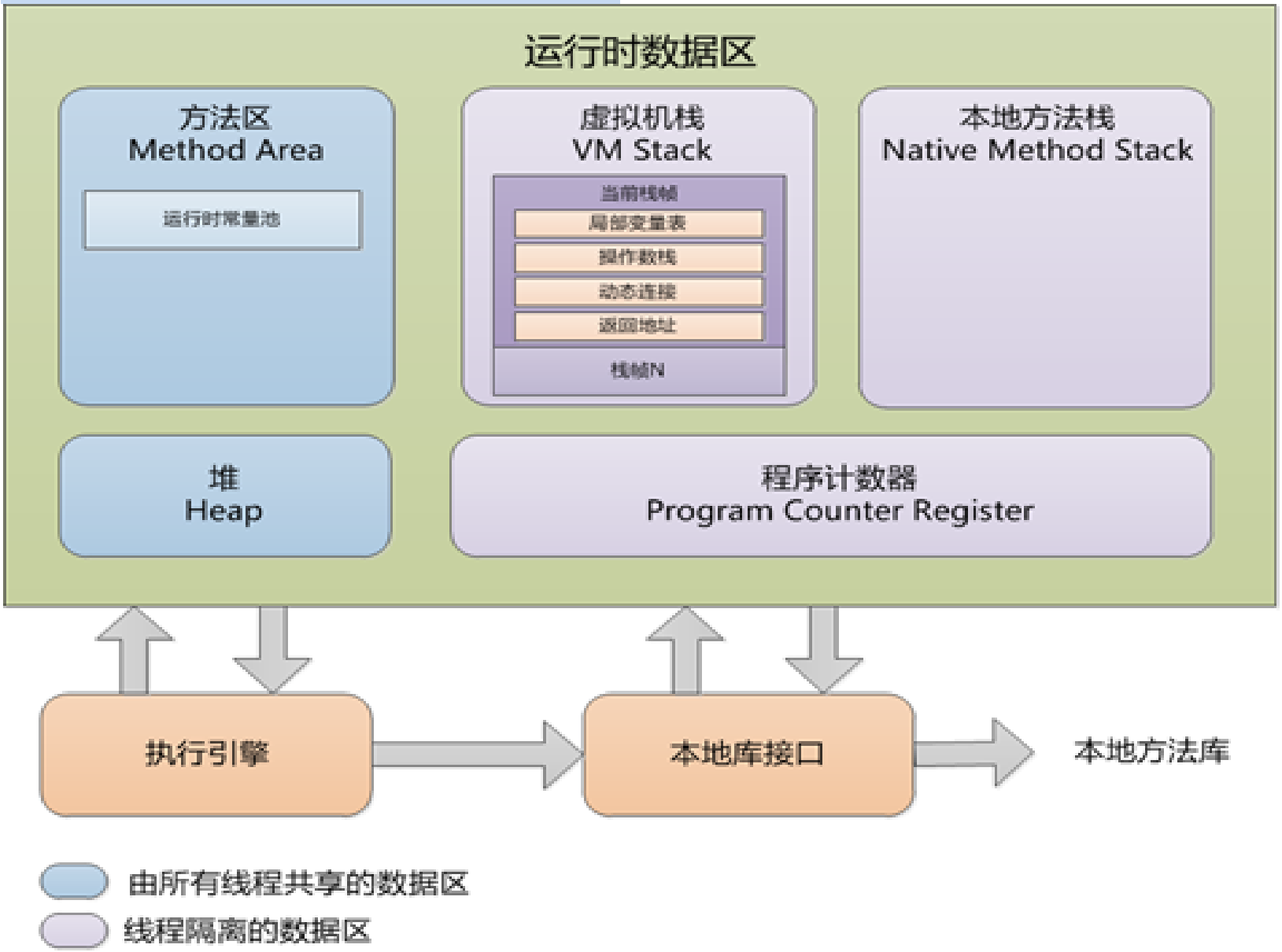
JVM GC算法和原理

JVM参数

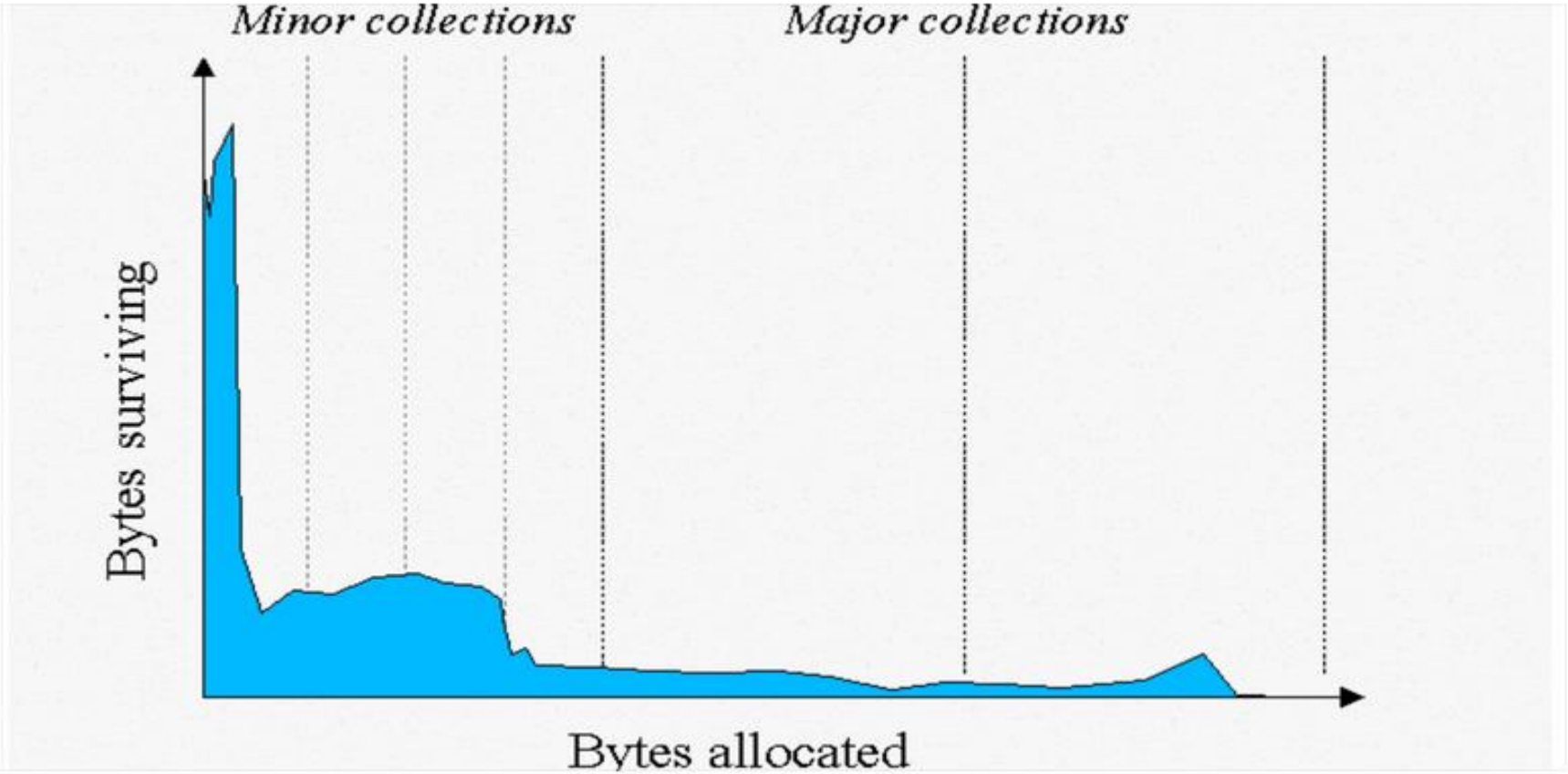
JVM GC优化

相关工具介绍

JVM内存模型


运行时数据区
方法区
Method Area
运行时常量池
虚拟机栈
VM Stack
当前栈帧
局部变量表
操作数栈
动态连接
返回地址
栈帧N
本地方法栈
Native Method Stack
堆
Heap
程序计数器
Program Counter Register
执行引擎
本地库接口
本地方法库
由所有线程共享的数据区
线程隔离的数据区

JVM堆分区的假设



IBM研究表明，绝大部分新生对象都是朝生夕死的，也就是一次GC之后就会把对应内存空间释放出来，最后存活下来的对象很少。如下图所示（显然绝大部分对象在第一次GC的时候就被回收掉了）

分代GC

- 使用分代回收技术，内存会被分为几个代（generation）。也就是说，按照对象存活的年龄，把对象放到不同的代中
- 年轻代：分配新对象，对象消亡快；GC频繁，GC速度快
- 年老代：存放长期存活的对象，一需要从年轻代熬过来，晋升到年老代；GC少，但代价大
- 超大对象也可能直接分配大年老代

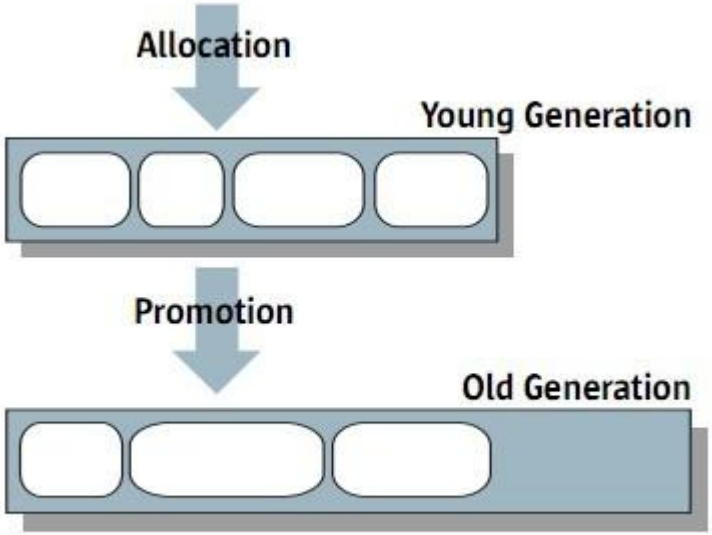
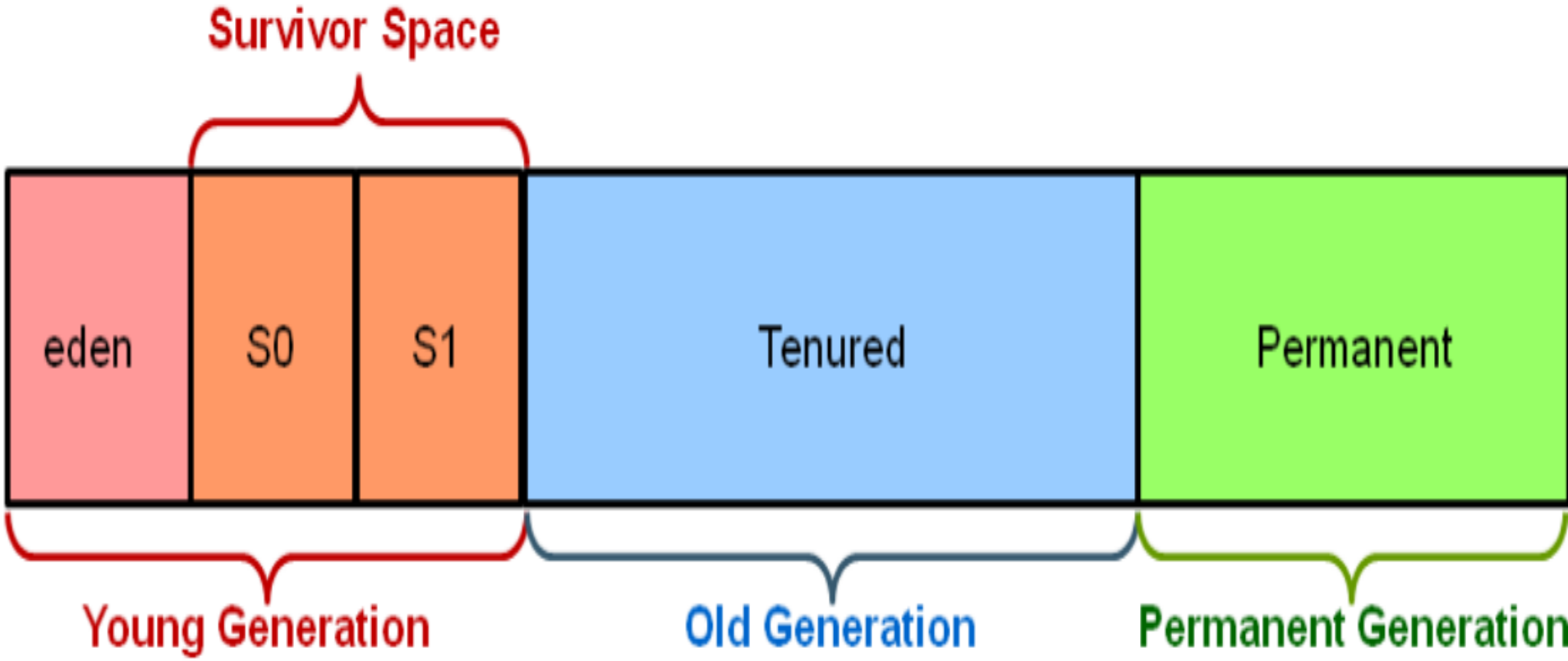


Figure 1. Generational garbage collection

JVM内存模型-Heap 分区



JVM内存模型

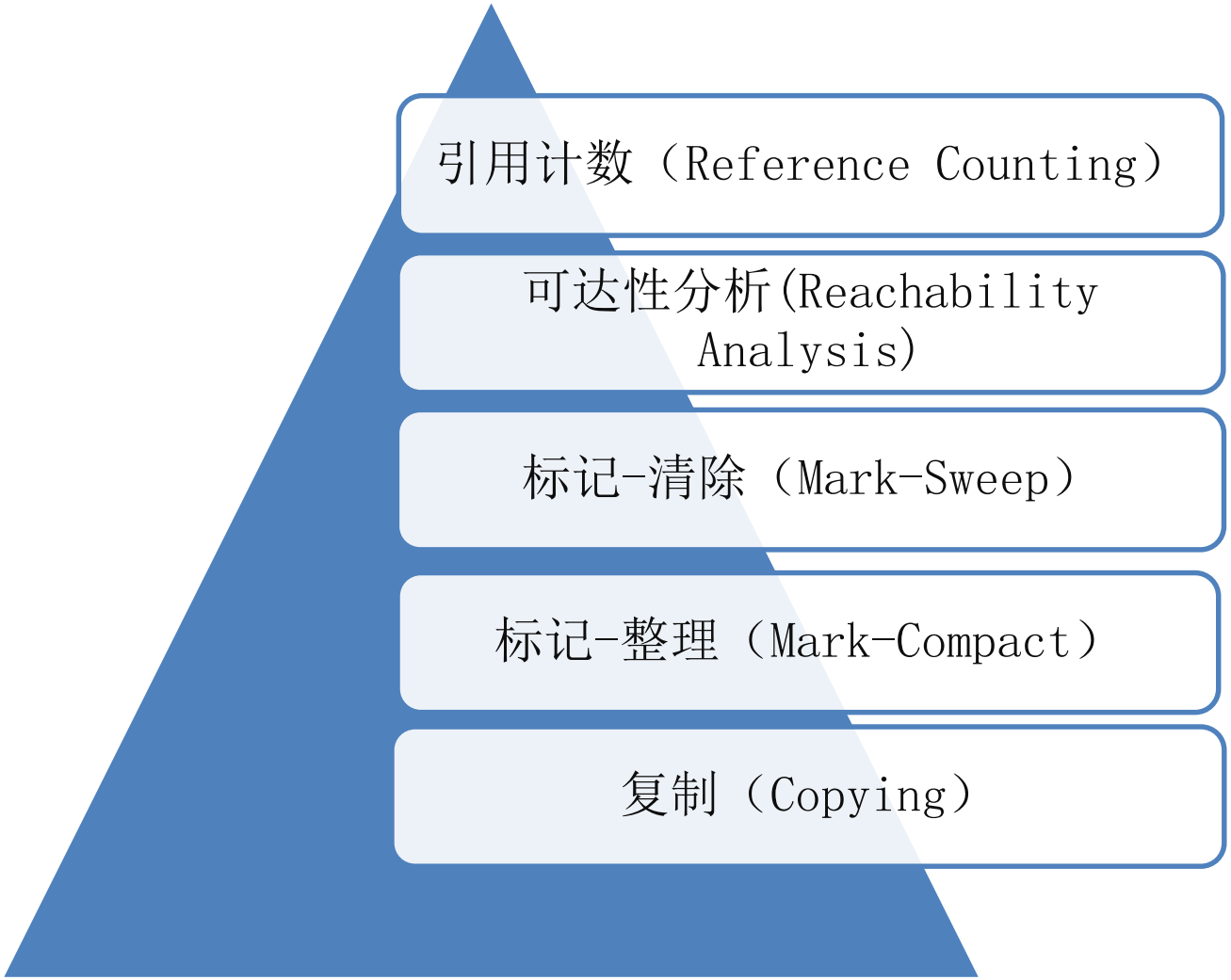
JVM GC算法和原理

JVM参数

JVM GC优化

相关工具介绍

❑基本GC算法


引用计数（Reference Counting）
可达性分析(Reachability Analysis)
标记-清除（Mark-Sweep）
标记-整理（Mark-Compact）
复制（Copying）

❑引用计数法

原理：**此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。**
垃圾回收时，只收集计数为0的对象。

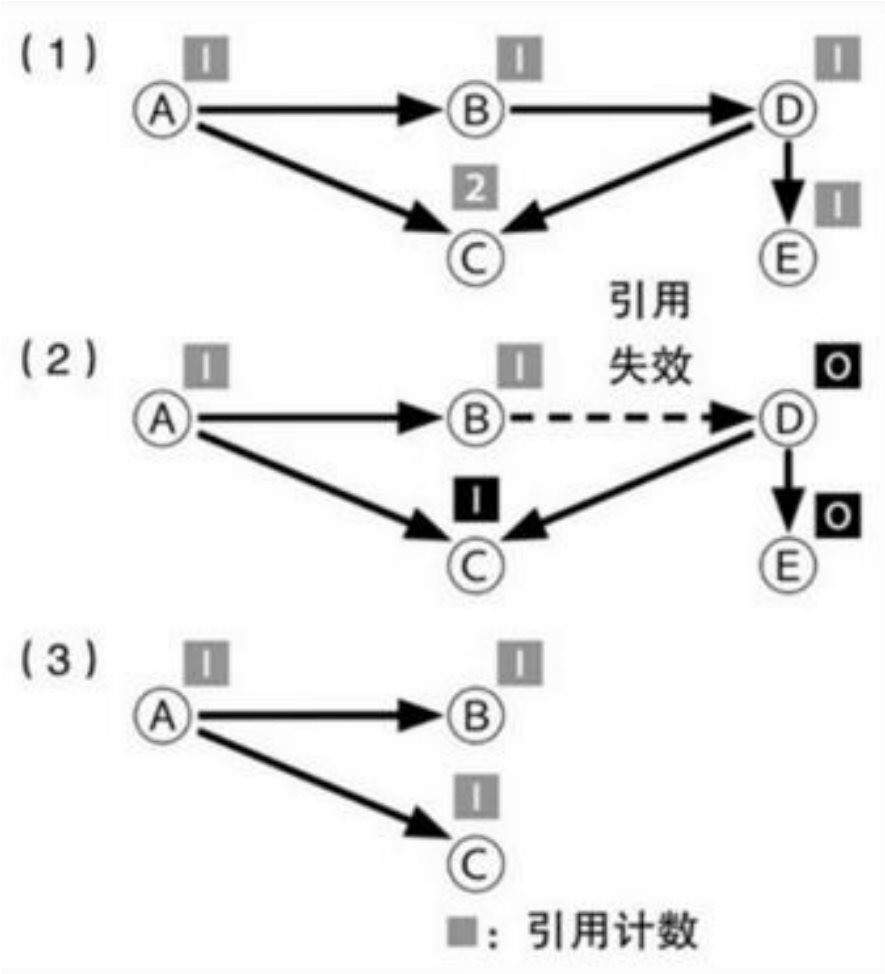
此算法最致命的是**无法处理循环引用**的问题

```
public class Main {
    public static void main(String[] args) {
        MyObject object1 = new MyObject();
        MyObject object2 = new MyObject();

        object1.object = object2;
        object2.object = object1;

        object1 = null;
        object2 = null;
    }
}
```

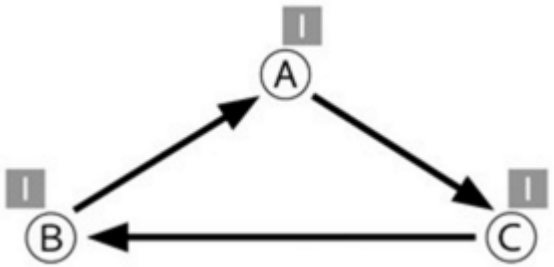
此外在并发处理中，容易出错。

❑引用计数法（续一）

引用计数法处理过程

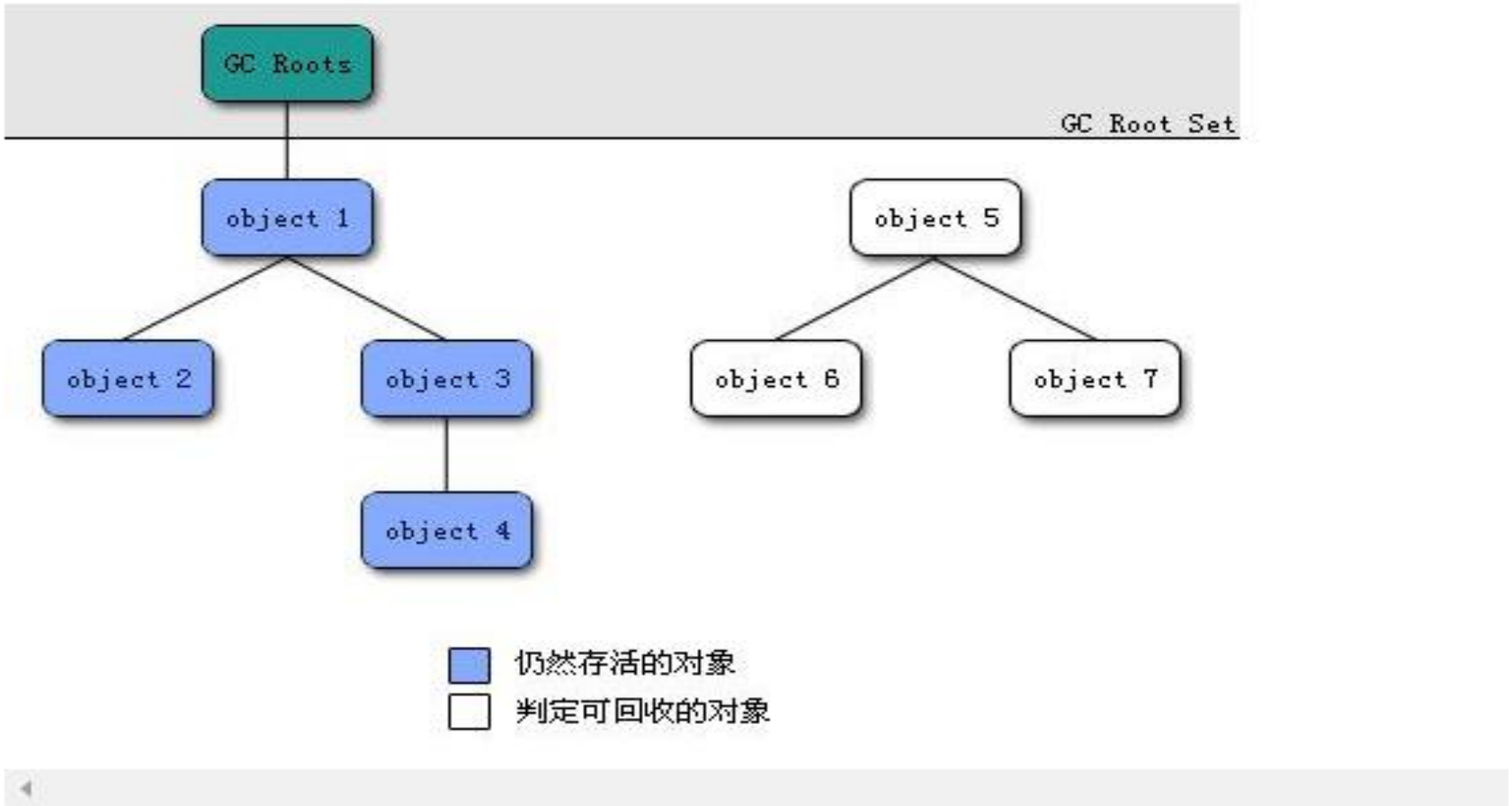


循环引用



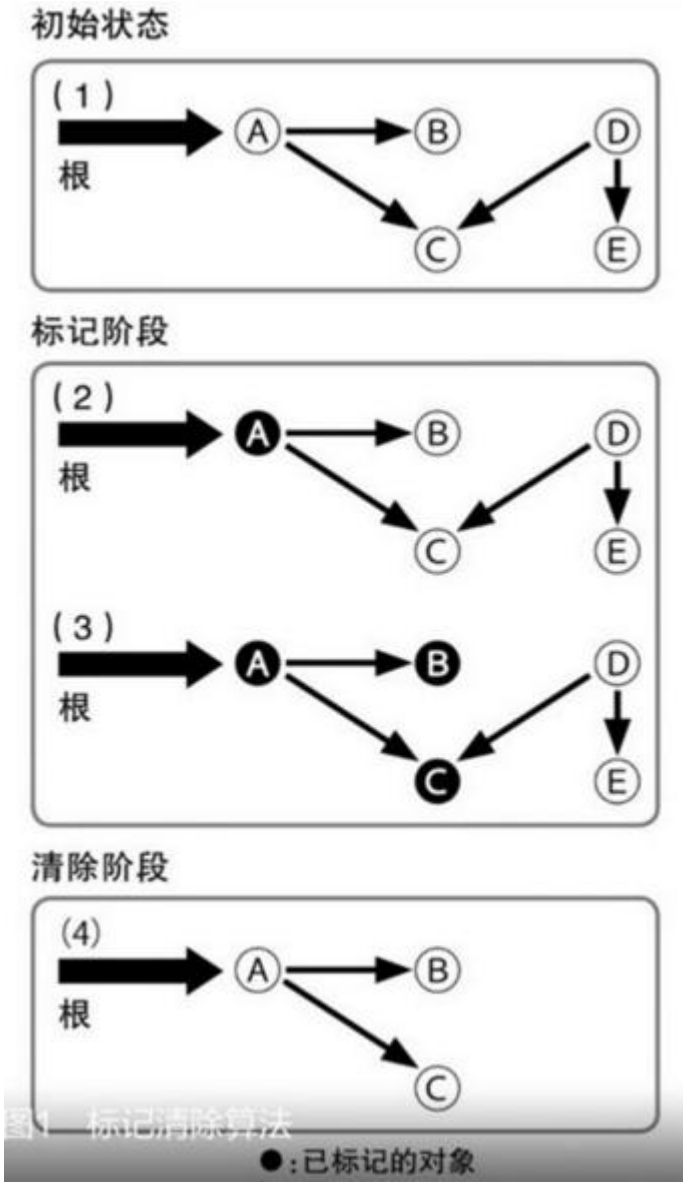
❑可达性分析

原理：基本思路就是通过一系列的称为”GC Roots”的对象作为起始点，从这些节点开始向下搜索，路径称为引用链，当一个对象到GC root没有任何引用链相连时，证明该对象不可用



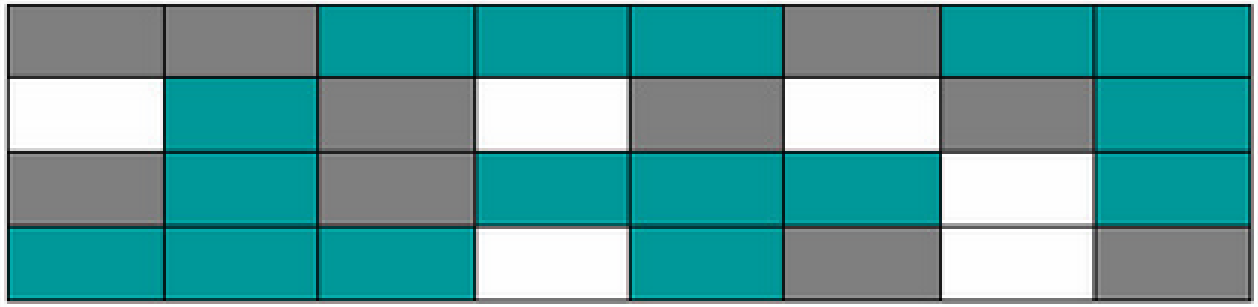
□标记清除(Mark-Sweep)

- 首先从根开始将可能被引用的对象用递归的方式进行标记，然后将没有标记到的对象作为垃圾进行回收
- 缺点一：会产生大量内存碎片。
- 缺点二：需要扫描标记、清除2遍流程，速度慢，如果存活对象比例低时，效率不好
- 改进：标记—整理（Mark-Compact），将存活对象压缩到连续空间

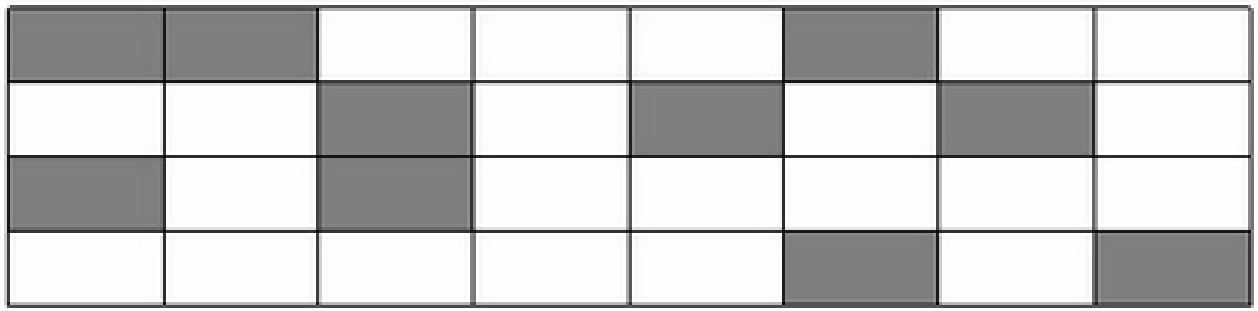


口标记清除(Mark-Sweep)

回收前状态：

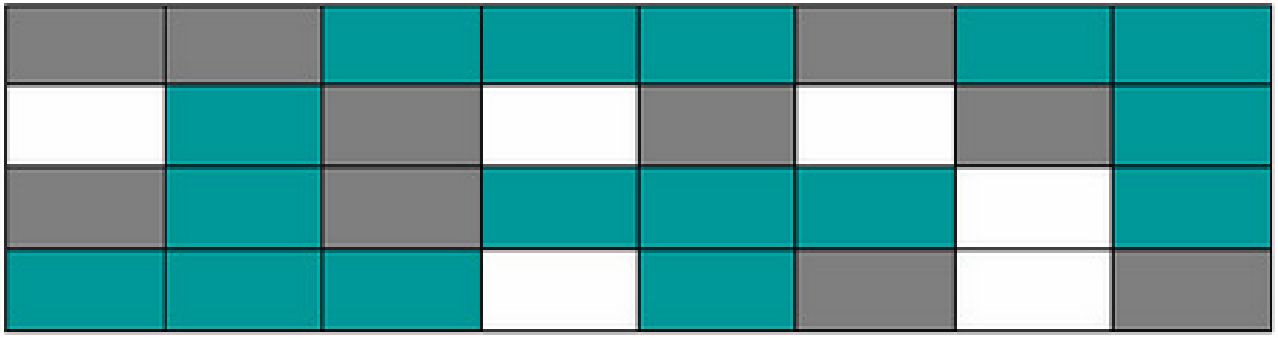


回收后状态：

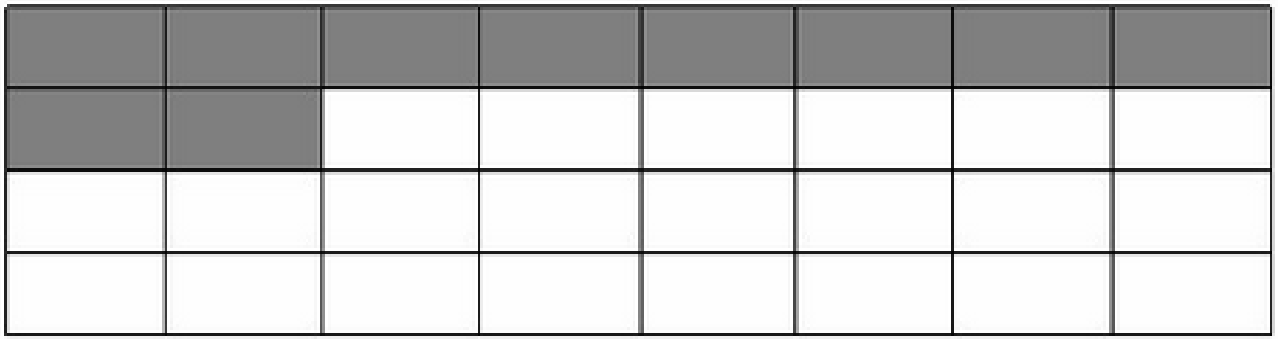


❑标记-整理（Mark-Compact）

回收前状态：

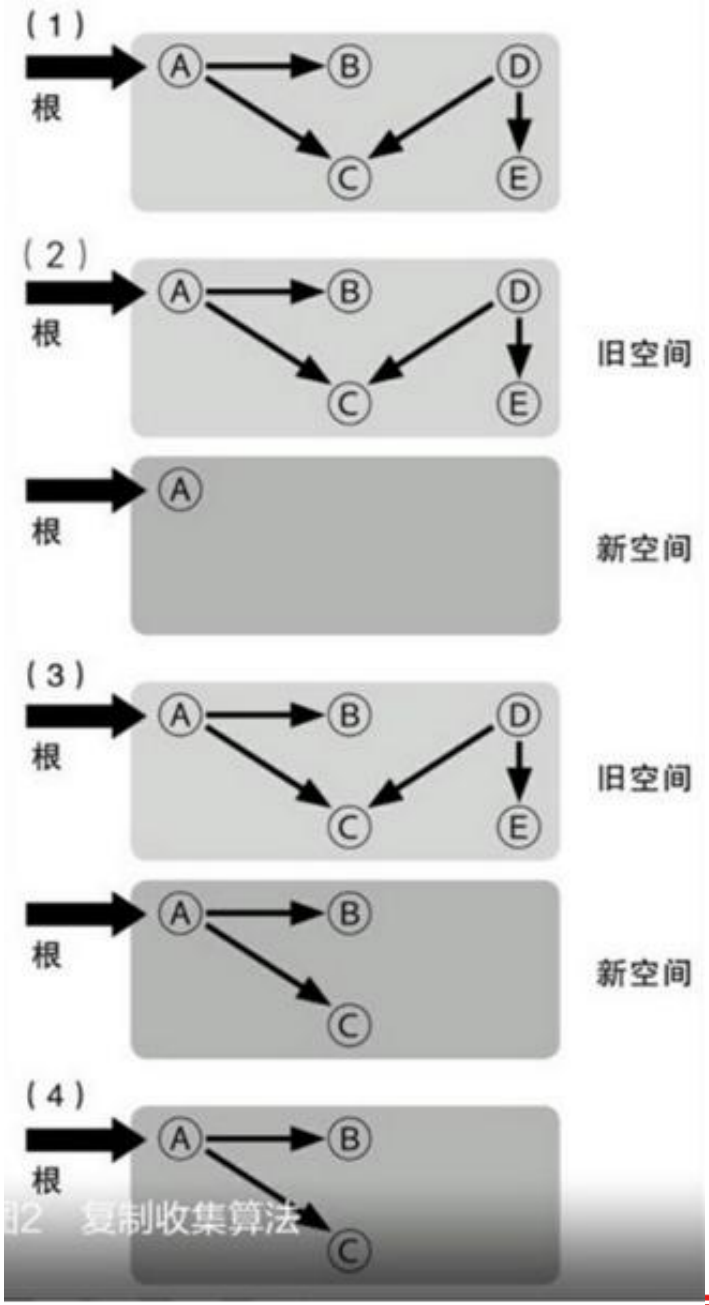


回收后状态：



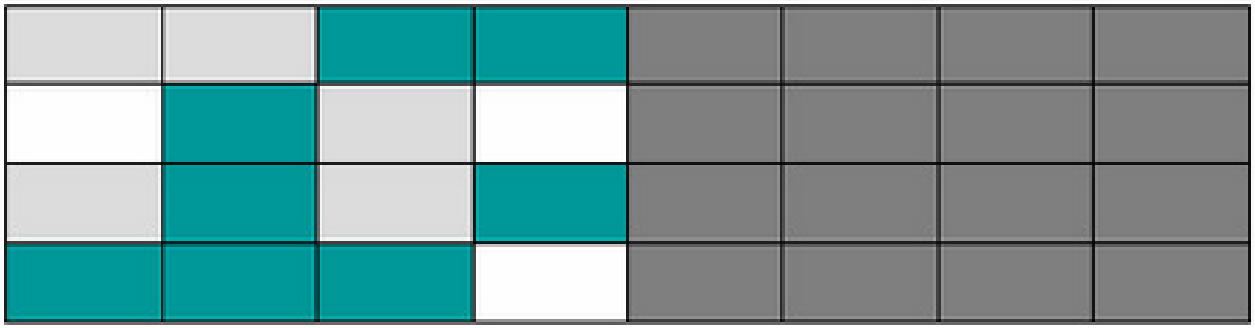
❑复制算法(copying)

- 需要额外的新空间
- 将可能从根被引用的对象复制到新空间中
- 空间换时间，只需要扫描一遍对象，速度快
- 如果存活对象比例高时，复制代价太大，可能速度比MS/MC算法更慢

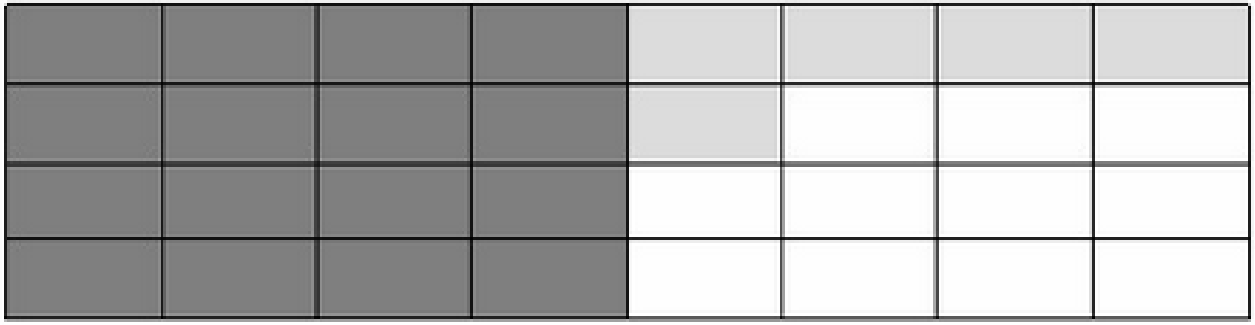


❑复制算法(copying)（续一）

回收前状态：



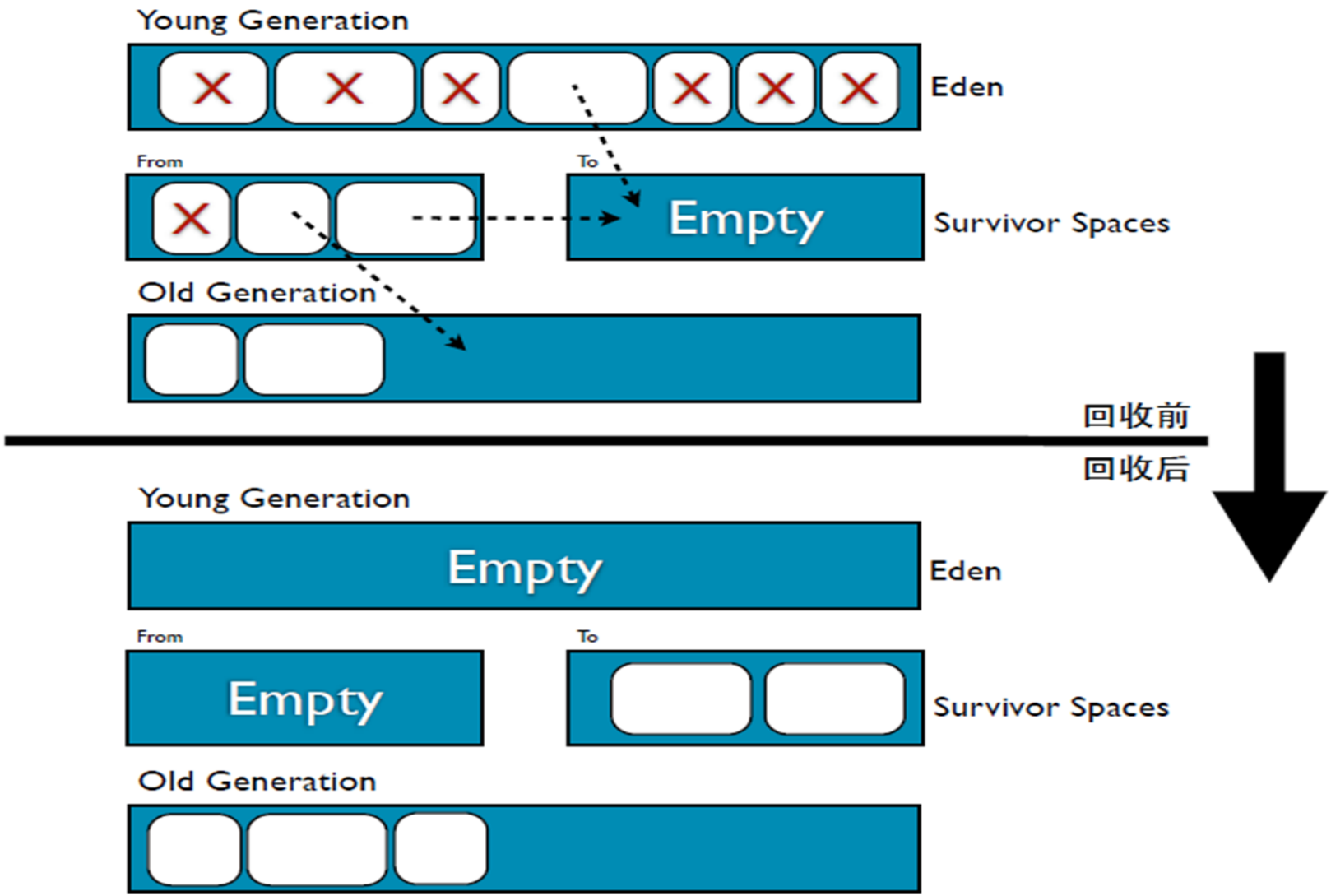
回收后状态：



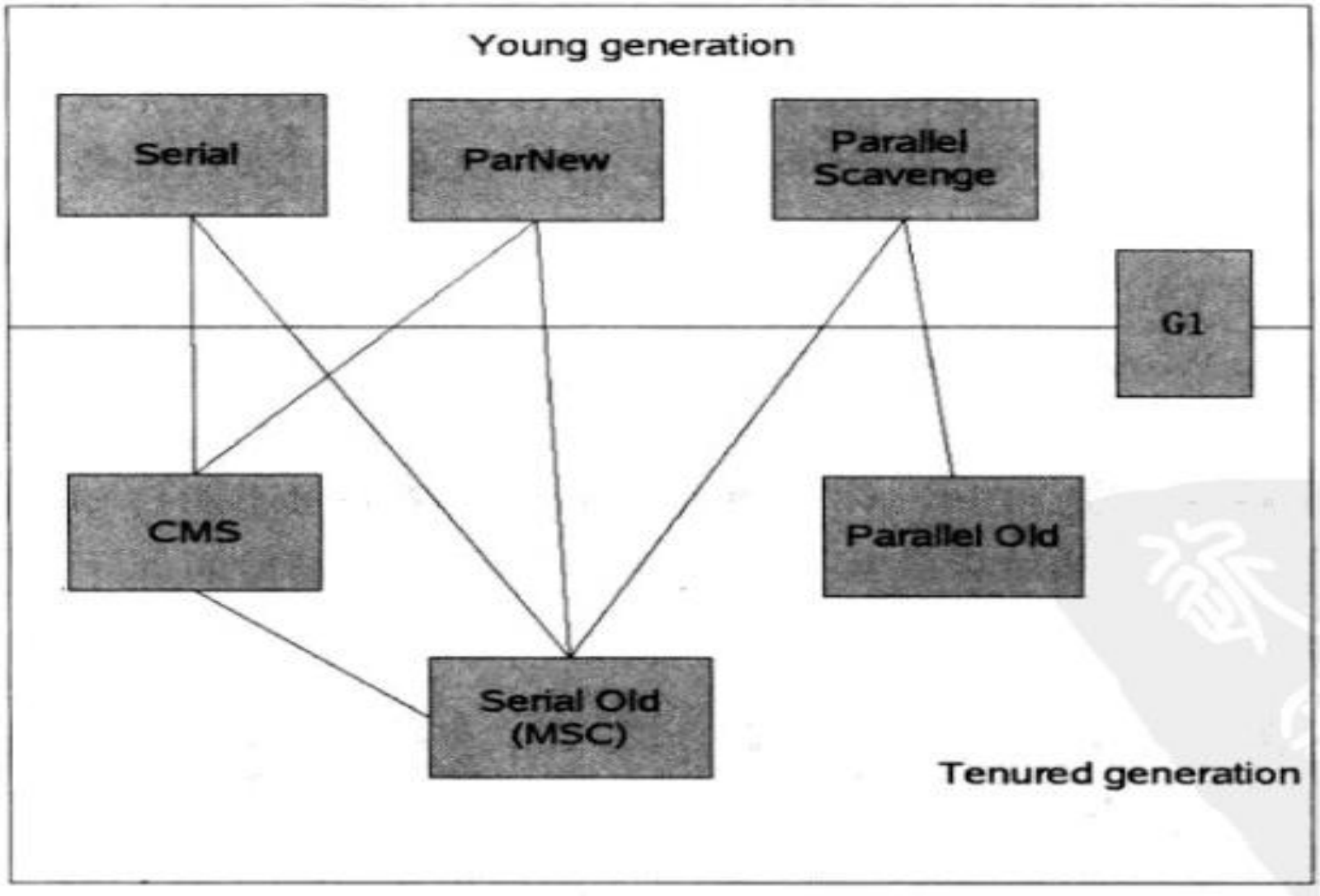
存活对象　可回收　未使用　保留区

❑复制算法(copying)-对象移动过程

垃圾回收



JAVA GC收集器


Young generation
Serial
ParNew
Parallel Scavenge
G1
CMS
Parallel Old
Serial Old (MSC)
Tenured generation

GC收集器类型

1. Serial收集器（复制算法)

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。

2. Serial Old收集器(标记-整理算法)

老年代单线程收集器，Serial收集器的老年代版本。

3. ParNew收集器(停止-复制算法)

新生代收集器，可以认为是Serial收集器的多线程版本,在多核CPU环境下有着比Serial更好的表现。

4. Parallel Scavenge收集器(停止-复制算法)

并行收集器，追求高吞吐量，高效利用CPU。吞吐量一般为99%， 吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)。适合后台应用等对交互相应要求不高的场景。

5. Parallel Old收集器(停止-复制算法)

Parallel Scavenge收集器的老年代版本，并行收集器，吞吐量优先

6. CMS(Concurrent Mark Sweep)收集器（标记-清理算法）

高并发、低停顿，追求最短GC回收停顿时间，cpu占用比较高，响应时间快，停顿时间短，多核cpu 追求高响应时间的选择

7. G1(*Garbage-First*)

全新的分块收集算法，并发处理，目标是同时实现高吞吐量和低停顿

连续 VS 并行

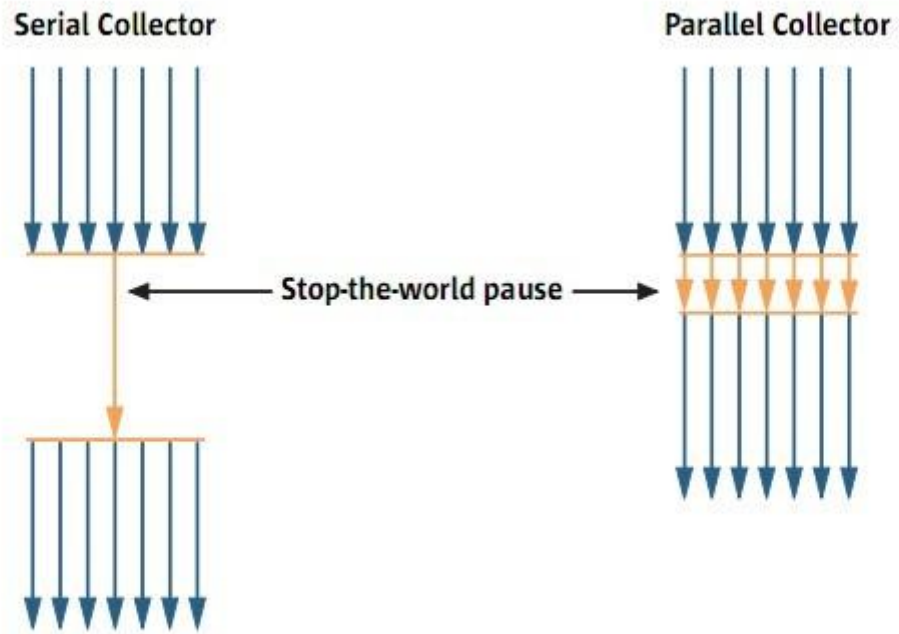


Figure 6. Comparison between serial and parallel young generation collection

CMS（Concurrent mark sweep）——并发

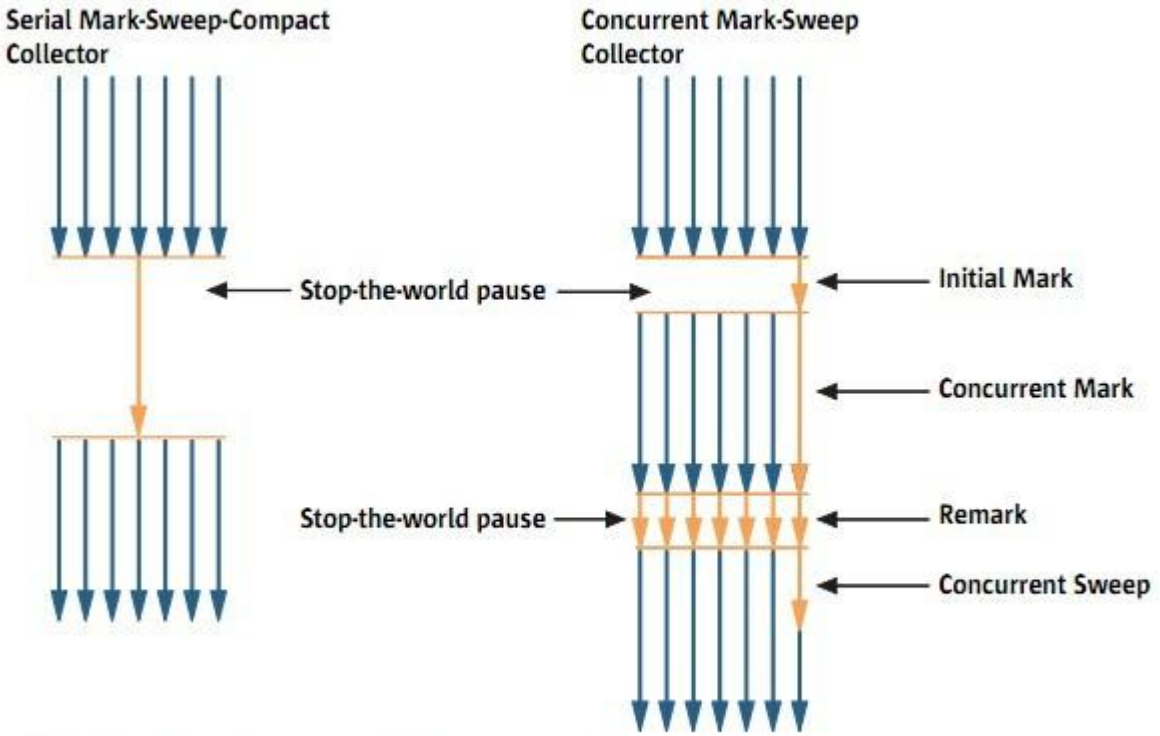
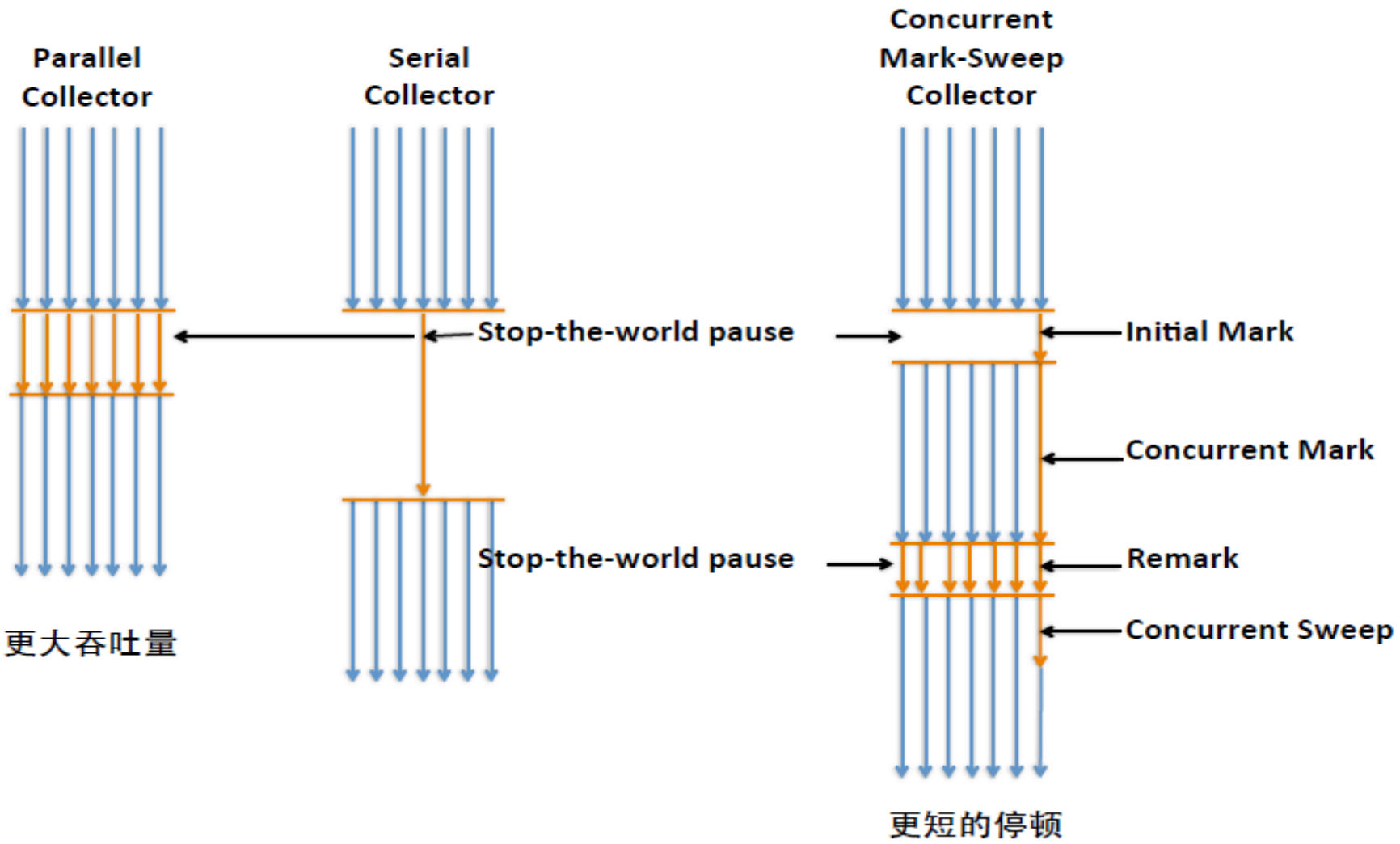


Figure 7. Comparison between serial and CMS old generation collection

CMS（Concurrent mark sweep）

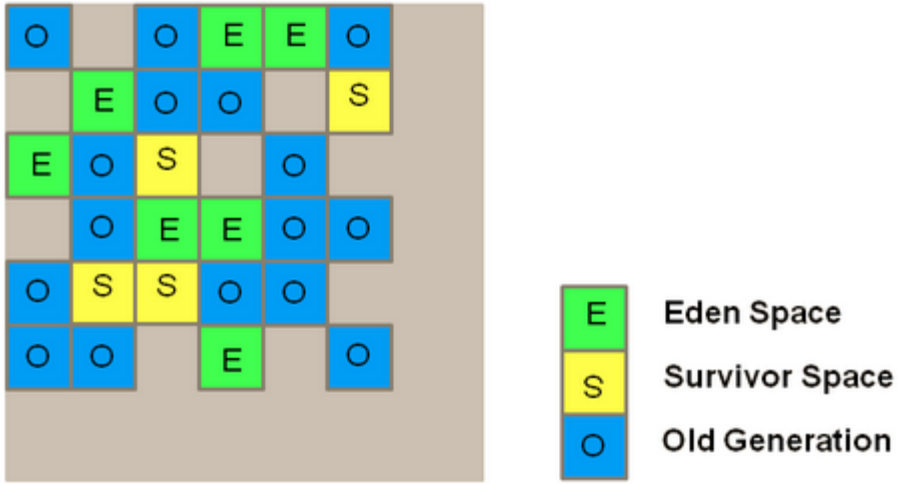
1. 过程：
    1. 初始标记：标记GC roots直接可达的对象，需要STW
    2. 并发标记：标记余下对象，和应用并发进行，一般使用1/4的CPU资源，不用STW
    3. 重标记：对第二步并发阶段时有修改的对象做重新标记，多线程处理，STW较长
    4. 并发清理：处理之前标记完成的对象，并发进行，无STW
2. 优点：STW短，适合实时性要求高的场景
3. 缺点：
    1. 内存要求大，一般在内存使用量70%左右开始GC
    2. 无压缩：需要额外的压缩过程
4. 会产生悬浮垃圾
5. 两种情况会触发full gc（普通的并行gc）
    1. Concurrent Mode Failure
    2. Promotion Failed
6. 默认GC收集器

各种GC行为比较


Parallel Collector
Serial Collector
Concurrent Mark-Sweep Collector
Stop-the-world pause
Stop-the-world pause
Initial Mark
Concurrent Mark
Remark
Concurrent Sweep
更大吞吐量
更短的停顿

G1

1. 一整块堆内存被分成多个小区域（Region）
2. 各代是由一系列不连续的区域组成的，这有助于随时改变它的大小，E、S、O区都是动态变化的
3. GC收集时会以Region为粒度进行复制
4. 有young gc、mixed gc、full gc
5. GC流程也有并发标记过程，类似CMS
6. GC时会计算区域活跃度，处理活跃度高的O区（ mixed gc ）
7. 当GC复制时没有空闲空间时（to-space exhausted），会触发整个堆的full gc，很慢，尽量避免



JVM内存模型

JVM GC算法和原理

JVM参数

JVM GC优化

相关工具介绍

通用参数

参数名称	含义	默认值和说明
-Xms	初始堆大小	物理内存的1/64
-Xmx	最大堆大小	物理内存的1/4
-XX:MinHeapFreeRatio	堆的最小空闲比例	40%
-XX:MaxHeapFreeRatio	堆的最大空闲比例	70%
-Xmn	年轻代大小	推荐整个堆的3/8
-XX:NewRatio	年轻代与年老代的比值	8
-XX:SurvivorRatio	Eden区与Survivor区的大小比值	32
-XX:+DisableExplicitGC	关闭System.gc()	
-XX:MaxTenuringThreshold	垃圾最大年龄	15
-XX:PermSize	持久代初始值（jdk1.7及以前）	物理内存的1/64
-XX:MaxPermSize	持久代最大值（jdk1.7及以前）	物理内存的1/4
-Xss	每个线程的堆栈大小	1M
-XX:TLABWasteTargetPercent	TLAB占eden区的百分比	1%

日志参数

参数名称	含义
-XX:+PrintGC	基本日志：[GC 118250K->113543K(130112K), 0.0094143 secs]
-XX:+PrintGCDetails	详细日志：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]
-XX:+PrintGCTimeStamps	输出时间：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]
-XX:+PrintGCApplicationStoppedTime	打印垃圾回收期间程序暂停的时间：Total time for which application threads were stopped: 0.0468229 seconds
-XX:+PrintGCApplicationConcurrentTime	打印每次垃圾回收前,程序未中断的执行时间：Application time: 0.5291524 seconds
-XX:+PrintHeapAtGC	打印GC前后的详细堆栈信息
-Xloggc:filename	自定义日志存储文件
-XX:+PrintReferenceGC	跟踪系统内的软引用，若引用，虚引用和Finallize队列
-XX:+PrintTLAB	查看TLAB空间的使用情况
XX:+PrintTenuringDistribution	查看每次minor GC后新的存活周期的阈值 Desired survivor size 1048576 bytes, new threshold 7 (max 15)

CMS参数

参数名称	含义
-XX:+UseConcMarkSweepGC	使用CMS内存收集
-XX:+UseCMSCompactAtFullCollection	full gc时进行压缩，默认开启
-XX:CMSFullGCsBeforeCompaction=0	多少次full gc后进行内存压缩，默认0，每次都进行
-XX:+CMSParallelRemarkEnabled	降低标记停顿，默认开启
-XX:+UseCMSInitiatingOccupancyOnly	禁止hostspot自行触发CMS GC，固定使用阈值来触发
-XX:CMSInitiatingOccupancyFraction=68	开始CMS GC的阈值，使用率大于阈值时开始CMS GC很关键
-XX:ParallelGCThreads=n	并行阶段使用的线程数，一般是CPU核数
-XX:ConcGCThreads=n	一般是(ParallelGCThreads + 3)/4
XX:+CMSScavengeBeforeRemark	CMS remark之前进行一次youngGC，这样能有效降低remark的时间

G1参数

参数名称	含义
-XX:+UseG1GC	使用G1 GC
-XX:MaxGCPauseMillis=200	最大停顿时间，这是一个目标值，JVM会尽量向此目标靠近
-XX:G1HeapRegionSize=n	G1的区域块大小，1M-32M 之间，必须是2的幂，默认值和堆大小相关
-XX:G1NewSizePercent=5	年轻代在堆中最小百分比，默认值是 5%
-XX:G1MaxNewSizePercent=60	年轻代在堆中的最大百分比
-XX:ParallelGCThreads=n	STW工作线程数的值，一般是CPU核数。
-XX:ConcGCThreads=n	并行标记的线程数，一般是ParallelGCThreads的1/4
-XX:InitiatingHeapOccupancyPercent=45	标记垃圾阀值，内存占用达到整个堆阈值时开启GC，默认45%
-XX:G1ReservePercent=10	作为空闲空间的预留内存百分比，以降低目标空间溢出的风险，默认10%

JVM参数配置示例

```
CATALINA_OPTS='-Xms60000m -Xmx60000m -Xmn15000m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
 -XX:SurvivorRatio=8 -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGC
DateStamps  -XX:+PrintHeapAtGC  -XX:+DisableExplicitGC -XX:+CMSParallelRemarkEnabled -XX
:+PrintTenuringDistribution -XX:TargetSurvivorRatio=80   -XX:MaxTenuringThreshold=8 -Dsu
n.rmi.dgc.server.gcInterval=86400000 -Dsun.rmi.dgc.client.gcInterval=86400000   -Xloggc:
/data0/log/gc.log -XX:+ExplicitGCInvokesConcurrent -XX:+CMSScavengeBeforeRemark -XX:+CMS
ClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=60 -XX:+UseCMSInitiatingOccupan
cyOnly -XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=32768 -XX:+UnlockComm
ercialFeatures -XX:+FlightRecorder -XX:+UnlockDiagnosticVMOptions -XX:FlightRecorderOpti
ons=defaultrecording=true,disk=true,maxchunksize=32m,repository=/data0/log/jfr,maxage=16
8h,maxsize=64g,dumponexit=true,dumponexitpath=/data0/log/jfr'
```

京东

JVM内存模型

JVM GC算法和原理

JVM参数

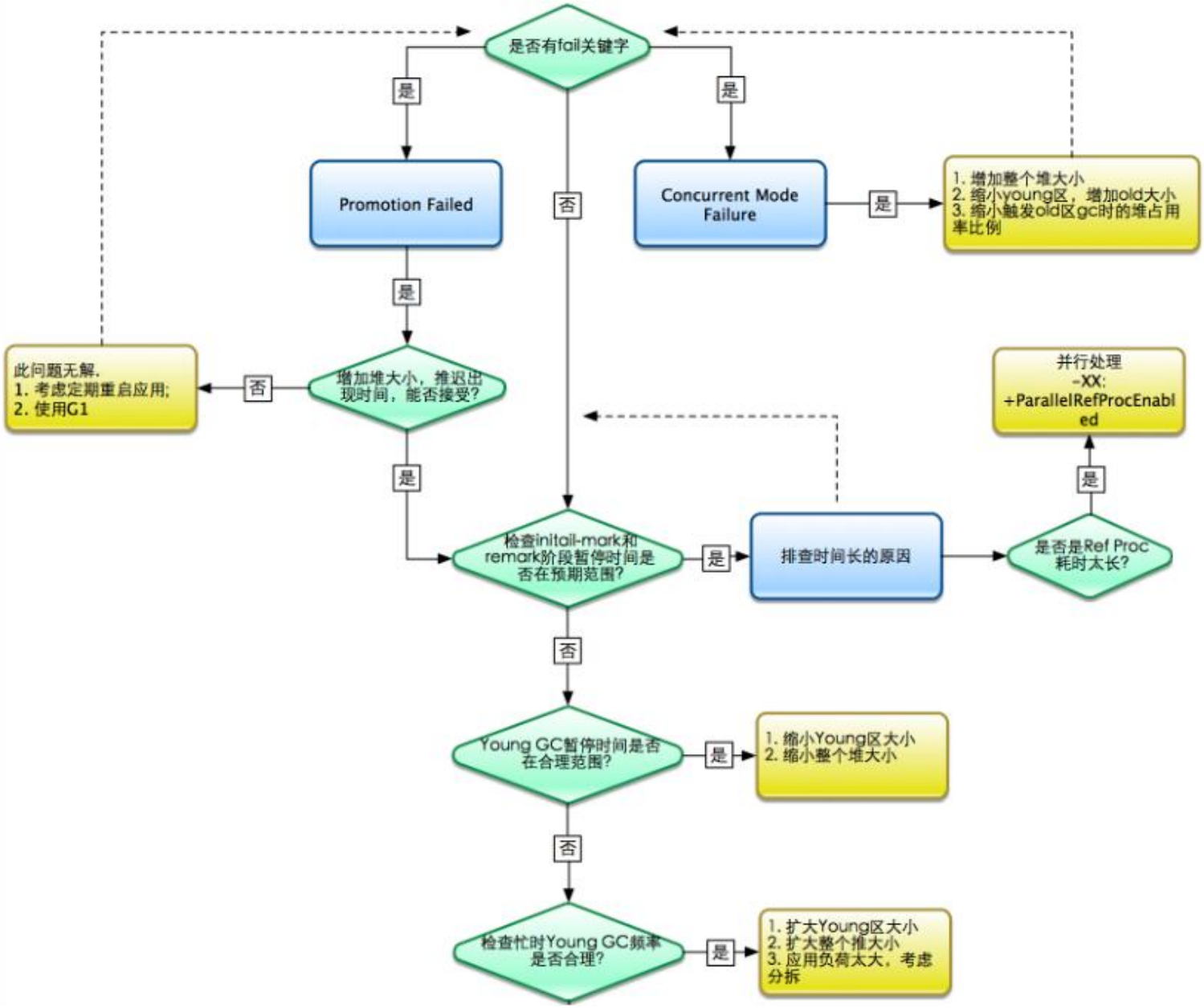
JVM GC优化

相关工具介绍

GC优化目标

- 不该回收的对象一定不能回收
- 该回收的对象一定尽量回收
- 尽可能少的暂停应用的运行
- CPU消耗尽量低
- 最终在时间，空间，回收频率这三个要素中平衡
- 衡量指标：
  - 吞吐量尽量大
  - STW尽量小

CMS优化

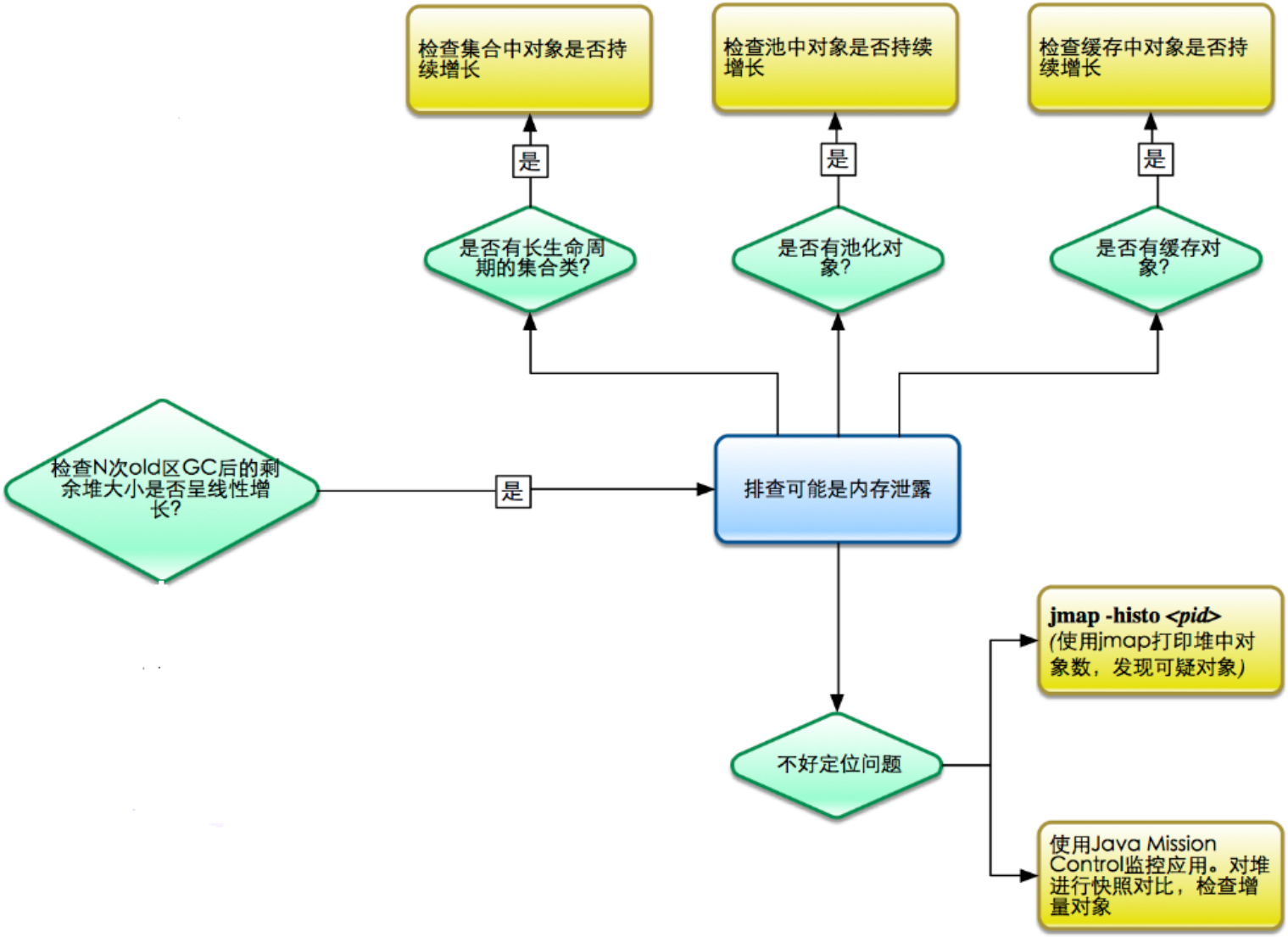

是否有fail关键字
是
是
否
Promotion Failed
Concurrent Mode Failure
是
1. 增加整个堆大小
2. 缩小young区，增加old大小
3. 缩小触发old区gc时的堆占用率比例
是
此问题无解。
1. 考虑定期重启应用;
2. 使用G1
否
增加堆大小，推迟出现时间，能否接受?
是
并行处理
-XX:
+ParallelRefProcEnabled
是
检查initail-mark和remark阶段暂停时间是否在预期范围?
是
排查时间长的原因
是否是Ref Proc耗时太长?
否
Young GC暂停时间是否在合理范围?
是
1. 缩小Young区大小
2. 缩小整个堆大小
否
检查忙时Young GC频率是否合理?
是
1. 扩大Young区大小
2. 扩大整个堆大小
3. 应用负荷太大，考虑分拆

京东

G1优化

1. 不要显式设置年轻代大小，以便G1可以更好的自动分配资源
2. 根据实际需求设置MaxGCPauseMillis,实时性要求没有太严格,可以提高该值，从而提高吞吐量
3. 避免full gc（to-space exhausted和to-space overflow日志）：
    1. -XX:G1ReservePercent 增加预存内存量
    2. -XX:InitiatingHeapOccupancyPercent 减少此值,提前启动标记周期
    3. -XX:ConcGCThreads 增加并行标记线程的数目
4. 当有较大的Update RSet时间时，增大ParallelGCThreads
5. mixed GC情况下，较长的cycle start时间，增大ConcGCThreads

内存泄露检查和解决



总结和最佳实践

- 将Xms和Xmx设为一样的值，避免动态调整的消耗
- 要吞吐量还是STW？
  - 一般扩大E区
- 内存足够且STW满足时，可以使用较大的Xmx
- CMS vs G1？
  - 一般大内存用G1，小内存用CMS
  - 分界线一般是8-10G
- MaxTenuringThreshold，根据日志中S区存活分布选择存活年龄的突变点
- 可以在高内存的单机器上布置多java实例，兼顾吞吐量和STW

GC 日志解读

```
2016-08-16T19:29:56.438+0800: 350460.166: [GC (Allocation Failure) 2016-08-16T19:29:56.438+0800
Desired survivor size 1258291200 bytes, new threshold 8 (max 8)
- age   1:   67261192 bytes,   67261192 total
- age   2:   11152968 bytes,   78414160 total
- age   3:   10181800 bytes,   88595960 total
- age   4:   76858240 bytes,  165454200 total
- age   5:   10143024 bytes,  175597224 total
- age   6:   10531848 bytes,  186129072 total
- age   7:   10644824 bytes,  196773896 total
- age   8:    9654224 bytes,  206428120 total
: 12546751K->305969K(13824000K), 0.1007610 secs] 32163905K->19933385K(59904000K), 0.1009600 secs
0 secs]
Heap after GC invocations=7433 (full 3):
 par new generation   total 13824000K, used 305969K [0x00002aaac0000000, 0x00002aae69800000, 0x0
  eden space 12288000K,   0% used [0x00002aaac0000000, 0x00002aaac0000000, 0x00002aadae000000)
  from space 1536000K,  19% used [0x00002aae0bc00000, 0x00002aae1e6cc500, 0x00002aae69800000)
  to   space 1536000K,   0% used [0x00002aadae000000, 0x00002aadae000000, 0x00002aae0bc00000)
 concurrent mark-sweep generation total 46080000K, used 19627416K [0x00002aae69800000, 0x00002al
 Metaspace       used 40525K, capacity 41056K, committed 41636K, reserved 43008K
}
{Heap before GC invocations=7433 (full 3):
```

1. **2016-08-16T19:29:56.438_0800** – GC事件(GC event)开始的时间点.
2. **350460.166** – GC时间的开始时间,相对于JVM的启动时间,单位是秒(Measured in seconds).
3. **GC** – 用来区分(distinguish)是 Minor GC 还是 Full GC 的标志(Flag). 这里的 GC 表明本次发生的是 Minor GC.
4. **Allocation Failure** – 引起垃圾回收的原因. 本次GC是因为年轻代中没有任何合适的区域能够存放需要分配的数据结构而触发的.
5. **ParNew** – 使用的垃圾收集器的名字.
6. **12546751K->305969K** – 在本次垃圾收集之前和之后的年轻代内存使用情况(Usage).
7. **(13824000K)** – 年轻代的总的大小(Total size).
8. **32163905K->19933385K** – 在本次垃圾收集之前和之后整个堆内存的使用情况(Total used heap).
9. **(59904000K)** – 总的可用的堆内存(Total available heap).
10. **0.100960 secs** – GC事件的持续时间(Duration),单位是秒.
11. **[Times: user=0.94 sys=0.00, real=0.10 secs]** – GC事件的持续时间,通过多种分类来进行衡量:

JVM内存模型

JVM GC算法和原理

JVM参数

JVM GC优化

相关工具介绍

❑GC 性能监控工具-jstat

```
C:\Users\wuzhenhua6>jstat -gcutil 27524 5000
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  74.70   8.76  89.37  77.02      2    0.052      2    0.243    0.295
  0.00   0.00  14.63   7.71  89.34  75.86      4    0.091      3    0.586    0.678
  0.00   0.00  14.10  12.34  91.50  81.47      7    0.204      4    0.843    1.047
  0.00   0.00  93.05  12.34  91.50  81.47      7    0.204      4    0.843    1.047
 74.00   0.00  52.56  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  60.50  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  60.50  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  60.50  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  60.50  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  60.50  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  61.12  12.34  91.13  81.29      8    0.227      4    0.843    1.069
 74.00   0.00  73.88  12.34  91.13  81.29      8    0.227      4    0.843    1.069
  0.00  70.95  30.92  12.34  91.05  82.05      9    0.253      4    0.843    1.096
 88.76   0.00  24.68  12.34  91.22  81.59     10    0.285      4    0.843    1.128
  0.00  99.97  17.74  12.73  91.09  81.33     11    0.342      4    0.843    1.184
  0.00  99.91  41.40  13.92  91.18  81.83     13    0.440      4    0.843    1.282
  0.00  99.95  93.12  16.07  90.77  81.41     15    0.575      4    0.843    1.418
  0.00  98.82  26.98  20.95  91.17  81.72     19    0.819      4    0.843    1.662
  0.00  43.68  38.92  24.99  91.11  81.88     21    0.907      4    0.843    1.749
  0.00  30.79  46.00  28.30  91.22  81.89     23    0.962      4    0.843    1.804
 16.09   0.00  12.22  30.21  91.25  81.97     24    0.996      4    0.843    1.839
 17.66   0.00  52.53  37.76  91.36  82.13     28    1.139      4    0.843    1.982
  0.00  24.08  65.31  41.89  91.35  82.19     31    1.234      4    0.843    2.077
 64.10   0.00  84.68  58.08  91.31  82.11     38    1.553      4    0.843    2.395
  0.00  55.99  19.02  68.09  91.35  81.99     43    1.742      4    0.843    2.585
 65.98   0.00  38.97  71.18  91.37  81.99     44    1.840      4    0.843    2.683
 65.98   0.00  39.11  71.18  91.37  81.99     44    1.840      4    0.843    2.683
```

京东

❑GC 性能监控工具-jmap

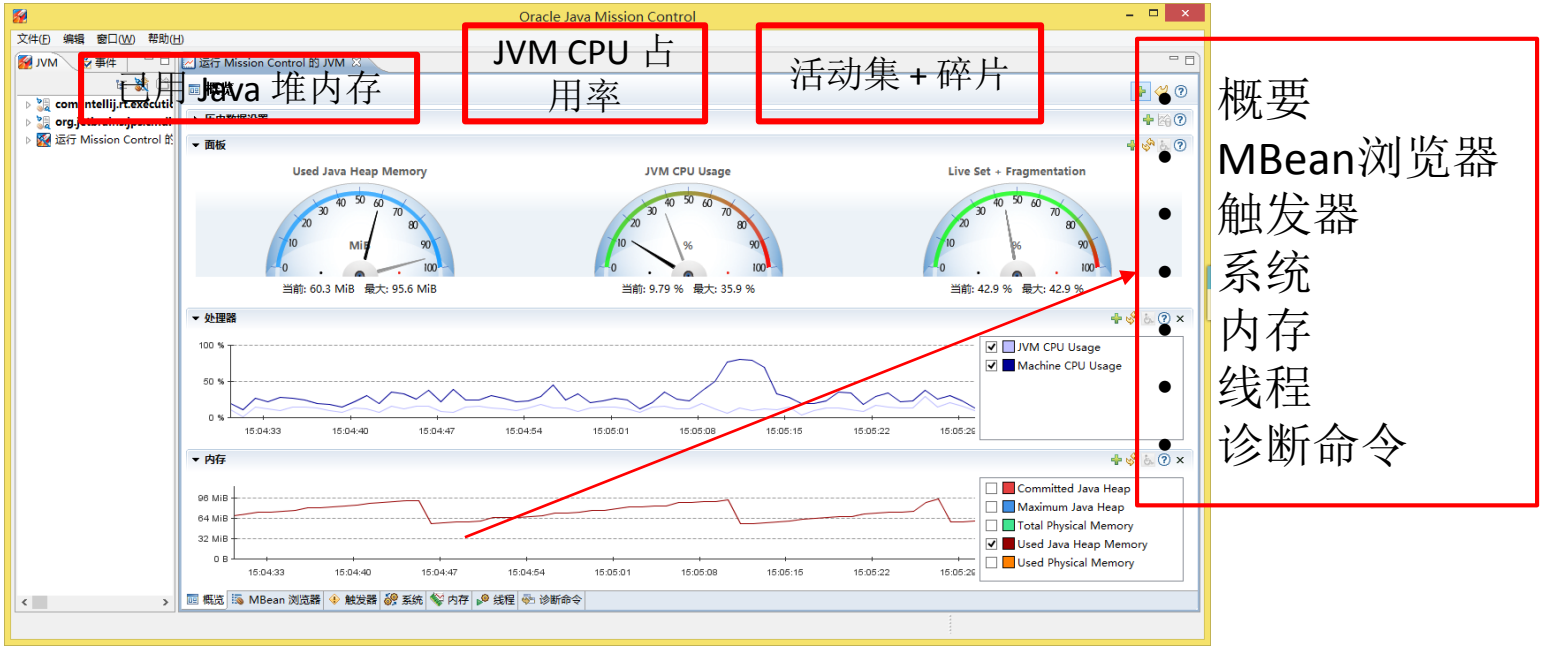
```
C:\Users\wuzhenhua6>jmap -histo 27524

 num     #instances         #bytes  class name
----------------------------------------------
   1:       2396948      150771992  [C
   2:       3004517       96144544  java.util.HashMap$Node
   3:        569056       82765360  [I
   4:       2298158       55155792  java.lang.String
   5:        482757       54334808  [Ljava.util.HashMap$Node;
   6:       1058221       42328840  java.util.LinkedHashMap$Entry
   7:        692133       35925920  [Ljava.lang.Object;
   8:        666548       21329536  org.apache.maven.model.InputLocation
   9:        463041       18521640  java.math.BigInteger
  10:         31564       18409248  [B
  11:        287042       16074352  java.util.LinkedHashMap
  12:        279494       13415712  java.util.HashMap
  13:        539316       12943584  java.util.ArrayList
  14:        159260       12740800  org.apache.maven.artifact.DefaultArtifact
  15:        223059       12491304  org.apache.maven.model.Dependency
  16:        293371        9387872  org.eclipse.equinox.internal.p2.metadata.OSGiVersion
  17:        462222        7395552  org.apache.maven.artifact.versioning.ComparableVersion$IntegerItem
  18:        159189        6367560  org.apache.maven.artifact.versioning.DefaultArtifactVersion
  19:        149031        5961240  org.eclipse.aether.artifact.DefaultArtifact
  20:         74137        5337864  org.eclipse.emf.ecore.util.EContentsEList$FeatureIteratorImpl
```

❑JMC和JFR介绍

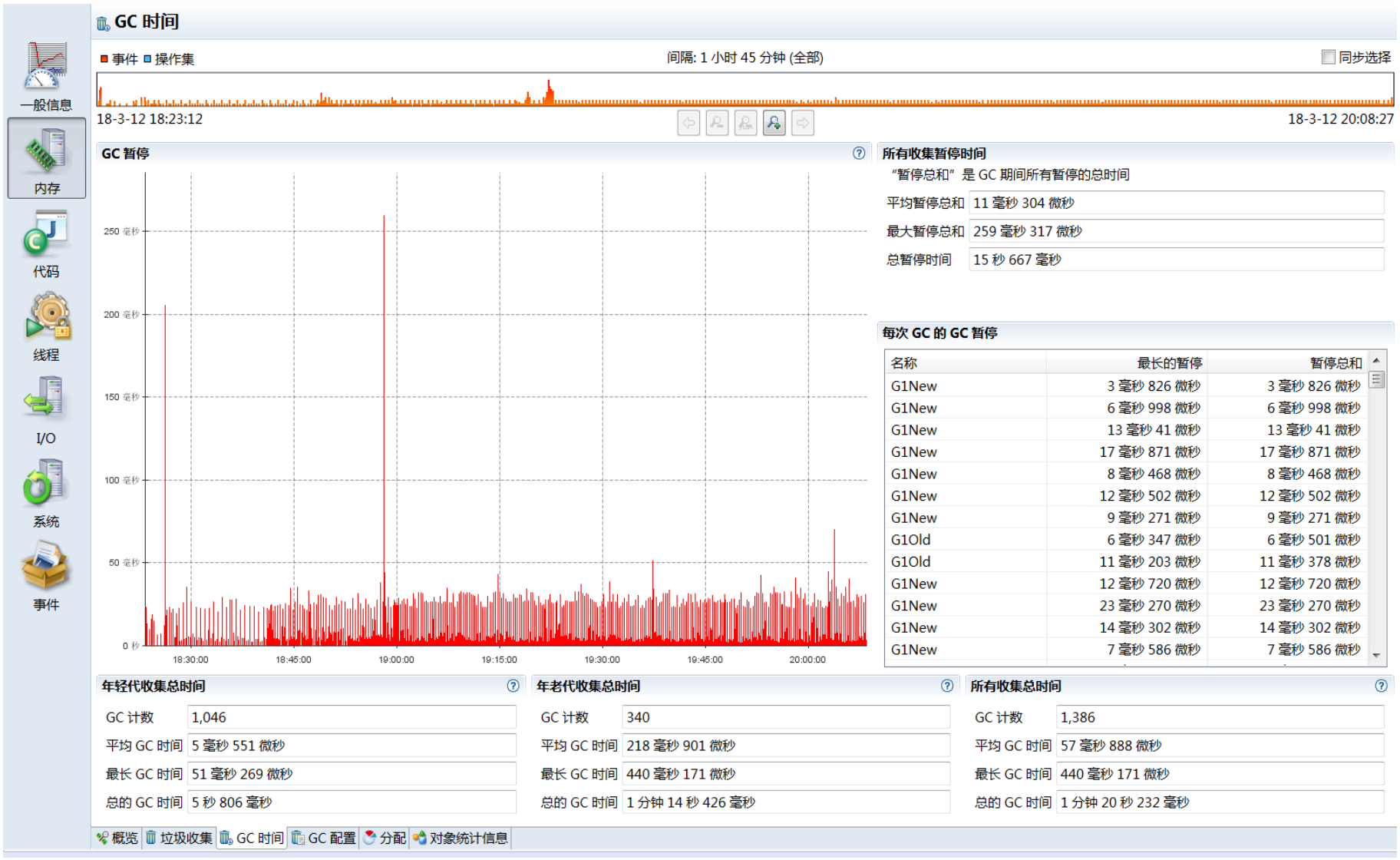
- Java Mission Control（JMC）一个工具集，用来对java应用程序进行管理、监控、概要分析和故障排除。
- 从JDK1.7_40开始，JMC和JFR工具捆绑到了HotSpot JVM中。JDK1.8版本对性能进行了更好的优化。
- 它是一个调优工具，适用于Oracle JDK 。
- Java Flight Recorder（JFR） 可以连续保存有关正在运行的系统的大量数据。此概要分析信息包括线程样本 (其中显示程序在什么地方占用了时间)、锁概要文件以及垃圾收集详细信息
- JFR收集的 (.jfr)数据可以在线下使用JMC工具进行分析。它集成到JVM中，几乎不会带来性能开销。（对于一般的应用程序不会高于1%）

❑Java Mission Control

- Jmc Console
  - 提供了一个控制台，它将连接到运行中的JVM以及之上运行的java程序，收集和展示当前的状态信息。还可以通过MBean更改一些运行时参数。



❑Java Fight Recorder



京东

❑JFR配置

- 启动商业功能和飞行记录器

-XX:+UnlockCommercialFeatures -XX:+FlightRecorder

- 飞行记录器的参数配置

-XX:FlightRecorderOptions=*parameter=value*

- **defaultrecording=*{true|false}*** — 是否在后台持续进行收集(default false)
- **disk=*{true|false}*** — JFR记录是否写磁盘(default false)
- **dumponexit=*{true|false}*** — JVM退出时是否dump记录(default false)
- **dumponexitpath=*path*** — dump记录的保存路径
- **maxage=time** — 默认记录保存的最长时间m,h,d (default 15m)
- **maxsize=size** — 默认记录写入磁的最大容量 k(K),m(M),g(G) (default not limit)
- **maxchunksize=size** — 设置数据buffer的大小 (default 12M)
- **repository=path** — JFR文件存储的路径

参考材料

- Memory Management in the Java HotSpot™ Virtual Machine（白皮书）
- Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide
- Getting Started with the G1 Garbage Collector

谢　谢！