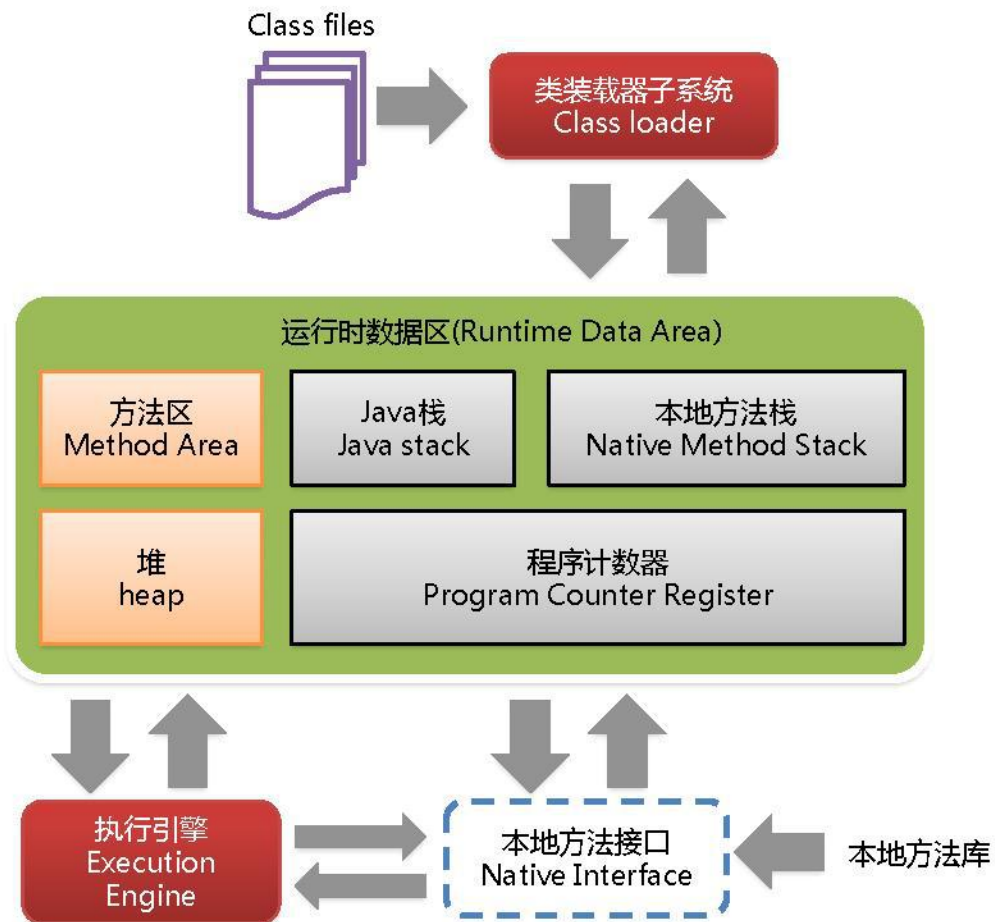


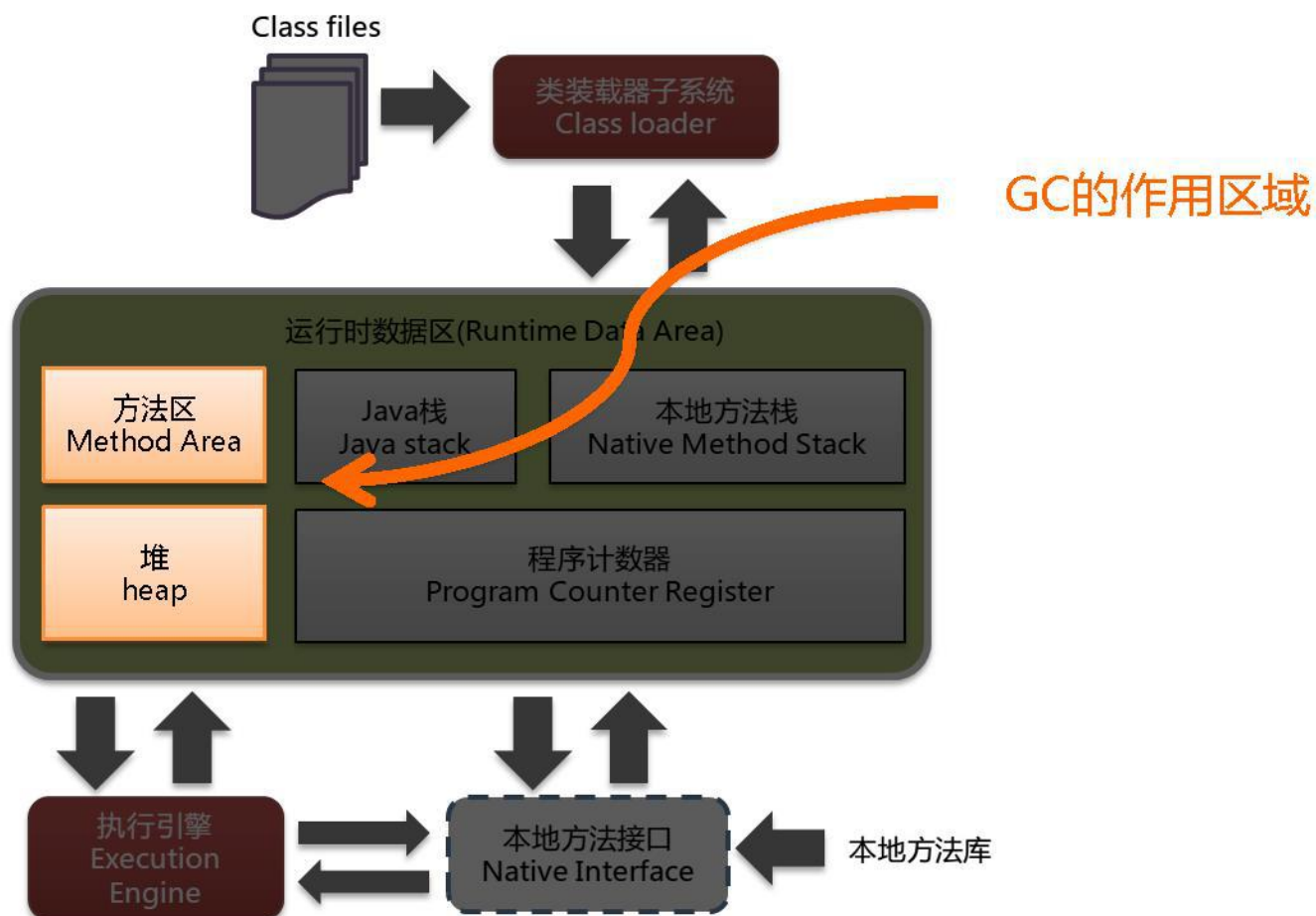
# JVM GC调优



# JVM体系结构概览



# JVM体系结构概览





# 常用GC算法



GC第一步: 找到可回收的垃圾

# 常用GC算法

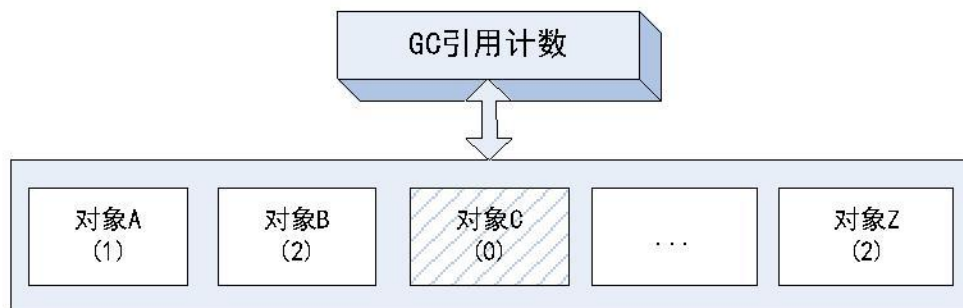
## 1. 引用计数法

(应用：微软的COM/ActionScrip3/Python...)

缺点：

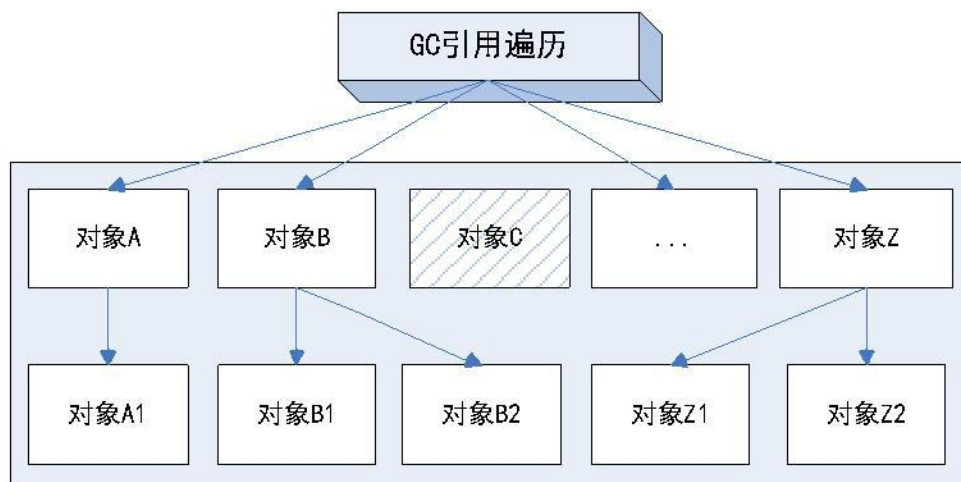
- 每次对对象赋值时均要维护引用计数器，且计数器本身也有一定的消耗；
- 较难处理循环引用

JVM的实现一般不采用这种方式



## 2. 跟踪(Tracing)

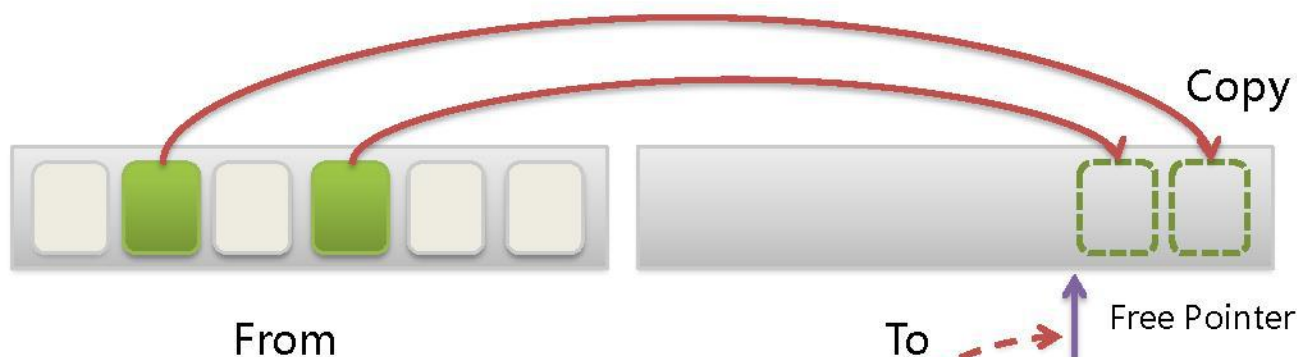
- **复制** (Copying)
- **标记-清除** (Mark-Sweep)
- **标记-压缩** (Mark-Compact)
- Mark-Sweep(-Compact)



# 复制 (Copying)

## 原理：

- 从根集合(GC Root)开始，通过Tracing从From中找到存活对象，拷贝到To中；
- From、To交换身份，下次内存分配从To开始；



没有标记和清除的过程，效率高



没有内存碎片，可以利用**bump**

**-the-pointer**实现快速内存分配

需要双倍空间



# 标记-清除 (Mark-Sweep)

原理：

## 1. 标记 (Mark)：

从根集合开始扫描，对存活的对象进行标记。



## 2. 清除 (Sweep)：

扫描整个内存空间，回收未被标记的对象，使用free-list记录空闲区域。



不需要额外空间



两次扫描，耗时严重；



会产生**内存碎片**

# 标记-压缩 (Mark-Compact)

## 原理：

### 1. 标记 (Mark)：

与 **标记-清除** 一样。



### 2. 压缩 (Compact)：

再次扫描，并往一端 **滑动** 存活对象。



没有内存碎片，可以利用bump  
-the-pointer



需要移动对象的成本

# 标记-清除-压缩 (Mark-Sweep-Compact)

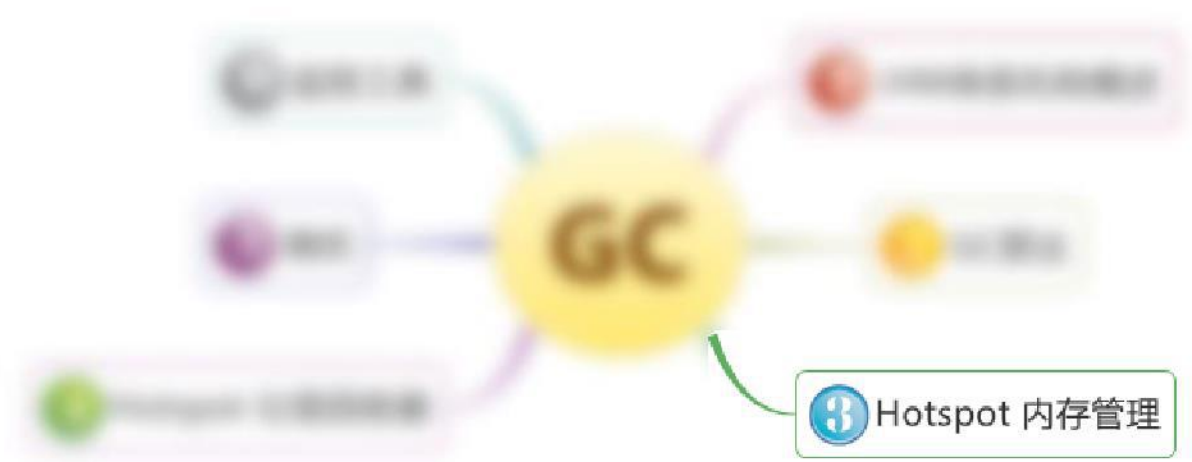
原理：

1. Mark-Sweep 和 Mark-Compact的结合。
2. 和Mark-Sweep一致，当进行多次GC后才Compact。



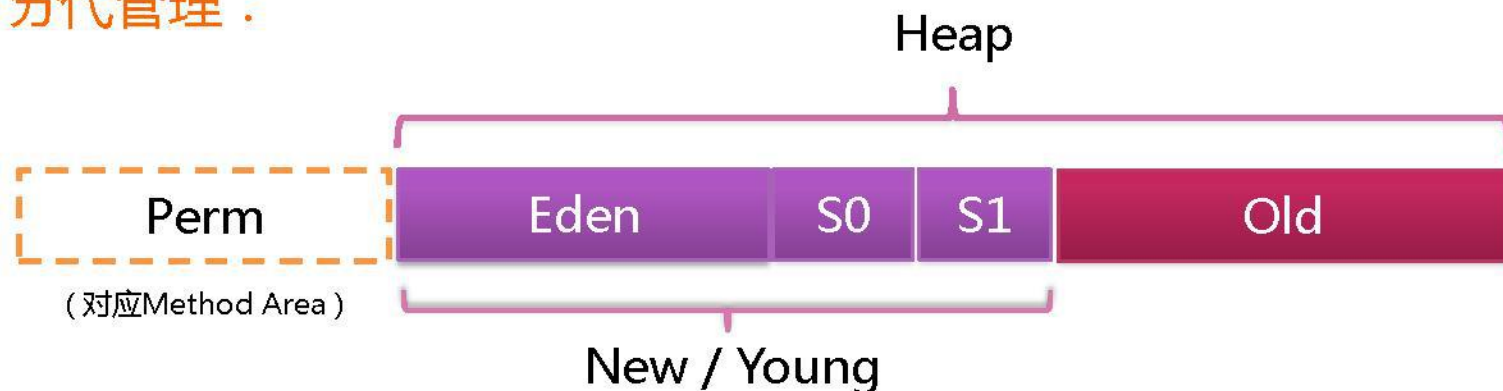
减少移动对象的成本

# 导航



# Sun HotSpot™ 内存管理

分代管理：



Why ?

- **真相**：经研究，不同对象的生命周期不同，98%的对象是临时对象。
- 根据各代特点应用不同GC算法，提高GC效率。

GC类型

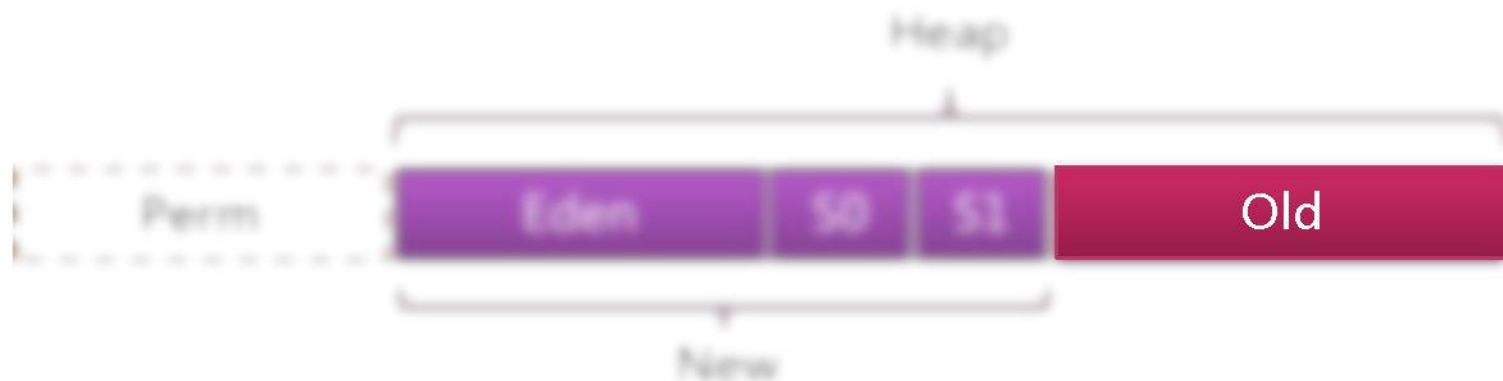
- **Minor GC** 针对新生代的GC
- **Major GC** 针对旧生代的GC
- **Full GC** 针对永久代、新生代、旧生代三者的GC

# 新生代



- 由Eden、两块相同大小的Survivor（又称为from/to,s0/s1）构成，to总为空；
- 一般在Eden分配对象，优化：Thread Local Allocation Buffer；
- 保存80%-90%生命周期较短的对象，GC频率高，采用效率较高的复制算法：

# 旧世代



- 存放新生代中经历多次GC仍然存活的对象；
- 新建的对象也有可能直接在旧世代分配，取决于具体GC的实现；
- GC频率相对降低，**标记**（mark）、**清理**（sweep）、**压缩**（compaction）算法的各种结合和优化。

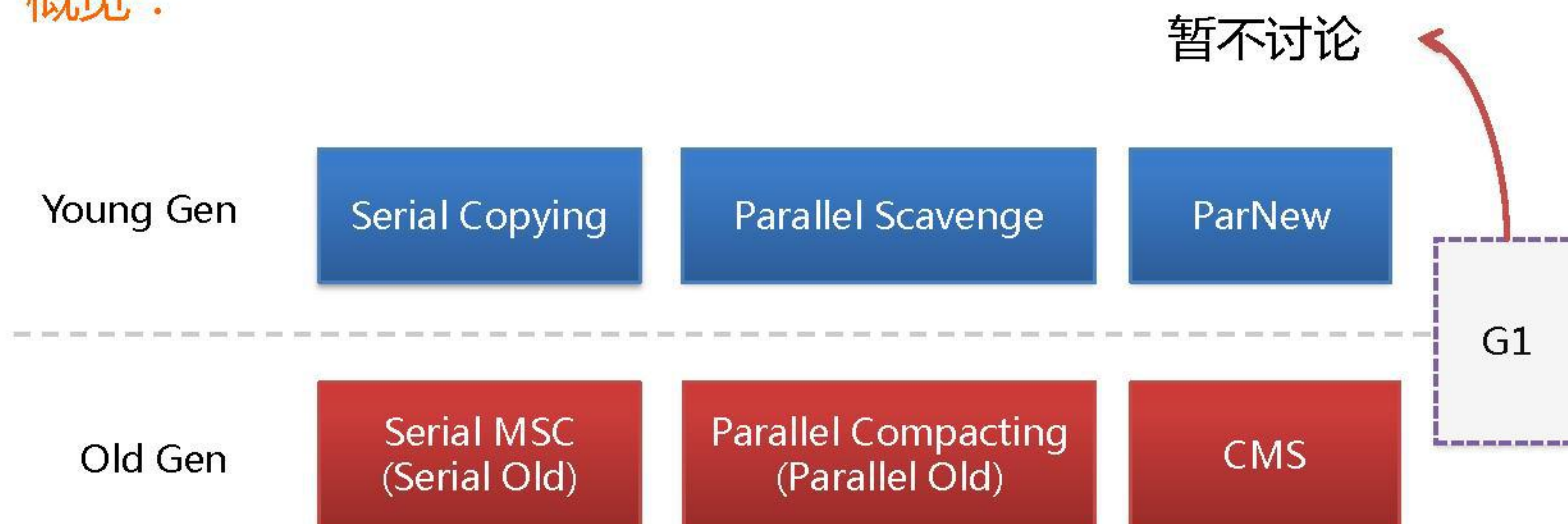
# 导航





# Sun HotSpot™ 垃圾回收器

概览：

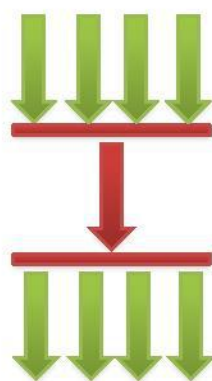


永久代呢？

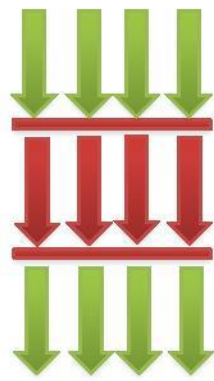
- 当 **永久代** 和 **旧世代** 触发GC时，除CMS外均会触发**Full GC**
  - 首先按照 **新生代** 配置的GC方式进行 **Minor GC**
  - 再按照 **旧世代** 配置的GC方式对 **旧世代**和**永久代** 进行GC
  - 若JVM **估计** minor GC后可能会发生晋升失败，则直接采用旧世代配置的GC方式对Young、Old、Perm进行GC。

# 一些术语I

## 串行(Serial) vs 并行(Parallel)



Serial



Parallel

- 在单核CPU下并行可能更慢；

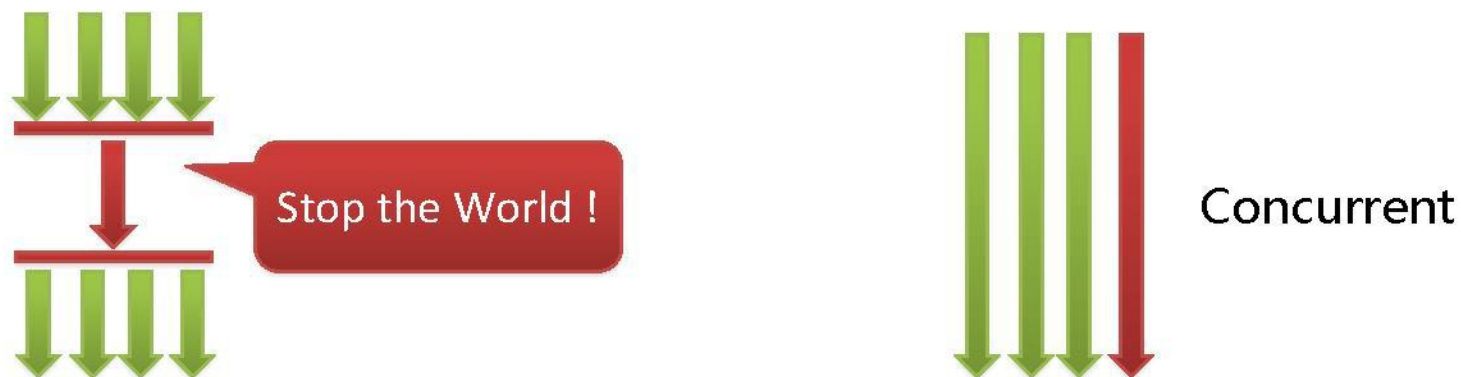


GC线程



应用线程

## STW(Stop-the-world) vs 并发(Concurrent)



- **STW**：暂停整个应用，时间可能会很长；
- **并发(Concurrent)**：更为复杂，GC可能会抢占应用的CPU；

# 新生代可用GC

Serial  
Copying

Parallel Scavenge

ParNew

- 均使用复制算法，原理上是一致的：
  1. 拷贝eden和from中的存活对象到to中；
  2. 部分对象由于某些原因晋升到old中；
  3. 清空eden、from，from和to交换身份直到下一次GC发生
- 分配对象时，Eden空间不足时触发

# Serial Copying

## 特性

- **Serial、Stop-the-world**

## 适用场景

- 单CPU、新生代小、对暂停时间要求不高的应用；
- 是client级别或32位windows上的默认选择

## 对象直接分配在Old的情况

- 对象大小超过eden space大小
- 大对象 (PretenureSizeThreshold)

## 晋升规则

- 经历多次minor gc仍存活的对象；
- To Survivor放不下的(满或剩余空间不够)对象直接晋升；

# Parallel Scavenge ( i )

## 特性

- **Parallel、Stop-the-world**
- 并行线程数默认值：
  - ✓ CPU核数  $\leq 8$  : =CPU核数
  - ✓ CPU核数  $> 8$  :  $=(3 + \text{CPU核数} * 5) / 8$亦可强制指定线程数 (-XX:ParallelGCThreads=4)
- 会根据minor GC的频率、时间等动态调整Eden/S0/S1的大小，可取消这一特性 (-XX:-UseAdaptiveSizePolicy)

## 适用场景

- 多CPU、对暂停时间要求较短的应用；
- 是server级别（2核CPU 2G内存）机器上的默认选择

# Parallel Scavenge ( ii )

## 对象直接分配在Old的情况

- 在TLAB和eden上分配失败，且对象大于eden的一半大小；
- PretenureSizeThreshold参数是无效的

## 晋升规则

- 经历多次minor gc仍存活的对象：
  - ✓ 规则和参数都比Serial Copying复杂
- To Survivor放不下的(满或剩余空间不够)对象直接晋升；

# ParNew

## 特性

- **Parallel、Stop-the-world**
- 可以认为是Serial Copying的多线程版本，各项特征与之基本一致；
- 可以搭配 CMS；
- 不可搭配 Parallel Old。



# 旧世代可用GC

Serial MSC  
(Serial Old)

Parallel Compacting  
(Parallel Old)

CMS

# Serial MSC

## 特性

- **Serial、Stop-the-world**
- 使用算法：**Mark-Sweep-Compact**
- 由于是单线程，GC造成的暂停时间可能会很长，可使用-XX:+PrintGCApplicationStoppedTime查看暂停时间

## 适用场景

- 是client级别或32位windows上的默认选择

# Parallel Compacting

## 特性

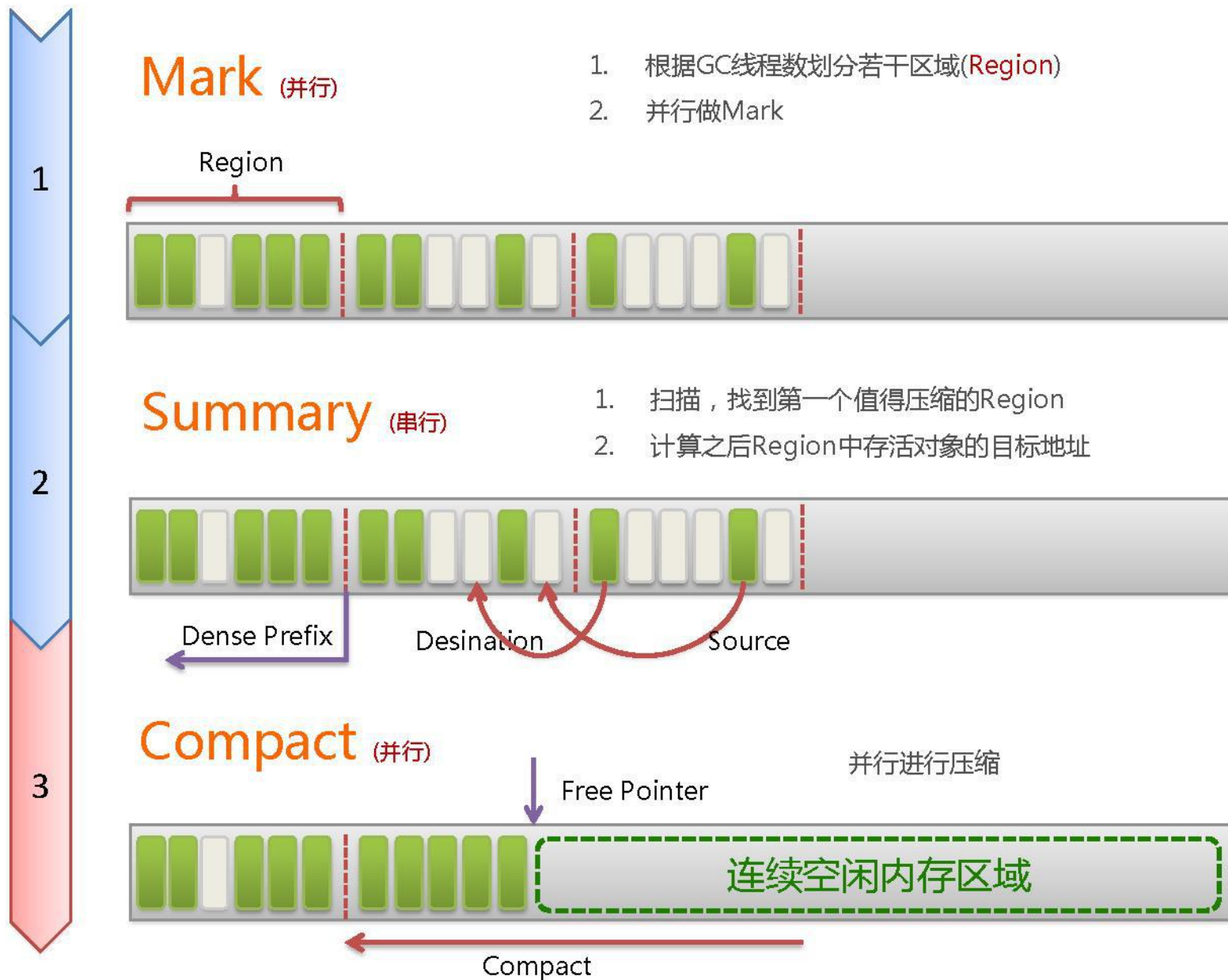
- **Parallel、Stop-the-world**
- 使用算法：**Mark-Compact**

算法较为复杂，详细描述见下一页

## 适用场景

- 多核CPU、对暂停时间较敏感的应用；
- 是server级别（2核CPU 2G内存）机器上的默认选择

# Parallel Compacting 算法详细描述



# Concurrent Mark-Sweep (CMS)

## 特性

- **Parallel、Concurrent**
- 使用算法：**Mark-Sweep**  
算法更为复杂，描述见下一页
- 缩短GC暂停时间，但相当复杂，增加了GC总时间
- 默认**并发线程数** = ( 新生代并行GC线程数 + 3 ) / 4，也可用-XX:ParallelCMSThreads=2来指定。
- 对Perm Generation也可启用CMS：-XX:+CMSPermGenSweepingEnabled，  
-XX:+CMSClassUnloadingEnabled

## 适用场景

- 暂停时间短，对追求最快响应速度的应用，尤其是互联网应用很适用

# CMS 算法简略描述



# Concurrent Mark-Sweep (CMS)

## 缺点

- 与应用抢占CPU

Solution: **i-CMS** (incremental CMS)

---- 已被废弃

- GC总耗时长

- 浮动垃圾(Floating Garbage)

- **Concurrent Sweep** 阶段有新的垃圾产生，只能下一次GC时被收集
- GC同时应用也在运行 → Old需要预留空间，达到一定比例即触发GC
  - XX:CMSInitiatingOccupancyFraction=68 指定，68%是默认值
- 预留空间不够，出现 **Concurrent Mode Failure**
  - “临时预案”：Serial MSC

- 内存碎片

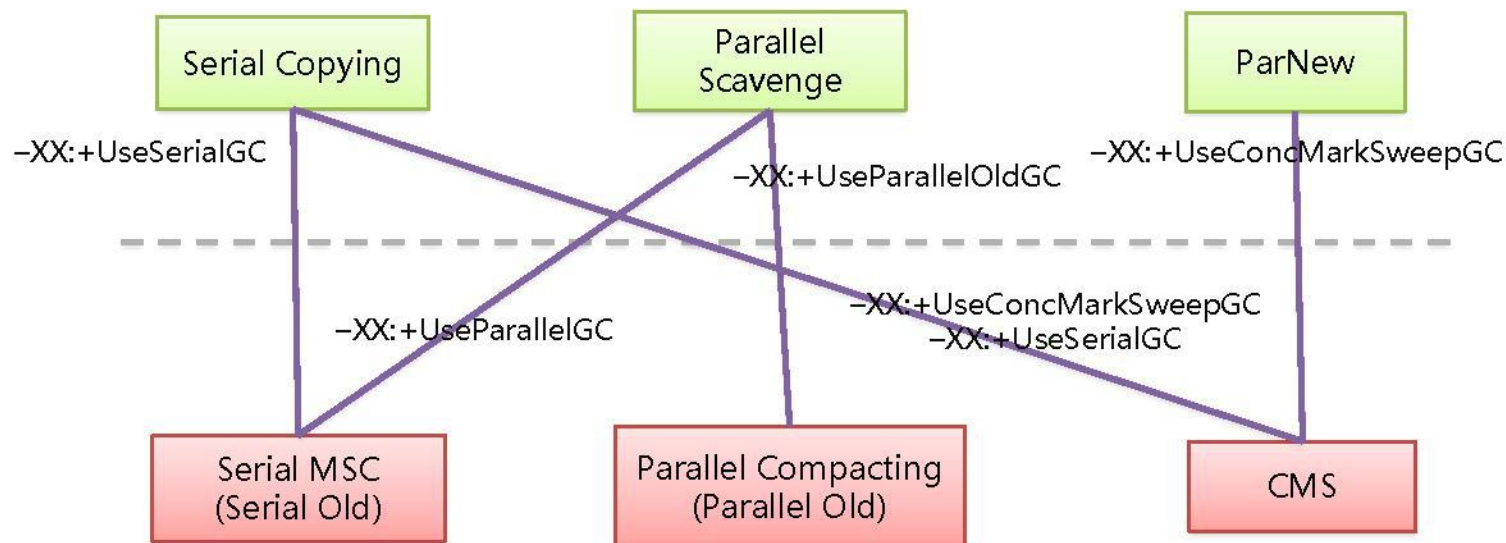
- free list中没有合适内存空间来分配对象，只能触发full GC.
  - XX:+UseCMSCompactAtFullCollection 每次full GC后进行压缩
  - XX:CMSFullGCsBeforeCompaction= 多少次full GC后进行压缩

- Minor GC耗时增长

- 每次Promotion都要搜索free list

# 组合

## 支持的组合



## 两个标准

- 吞吐量 (Throughput) = 应用运行时间 / 总时间  
关注GC总耗时
- 暂停时间 (Latency)  
关注每次GC造成的应用暂停



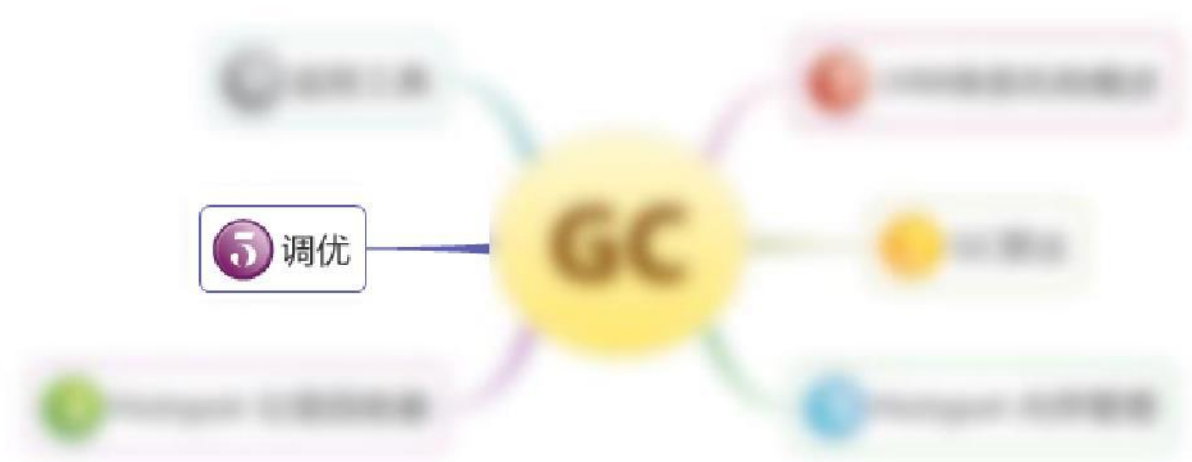
# 组合

## 组合的选择

- 单CPU或小内存，单机程序  
-XX:+UseSerialGC
- 多CPU，需要最大吞吐量，如后台计算型应用  
-XX:+UseParallelGC 或者  
-XX:+UseParallelOldGC
- 多CPU，追求低停顿时间，需快速响应如互联网应用  
-XX:+UseConcMarkSweepGC  
-XX:+ParNewGC

## 自动选择GC方式

- 吞吐量优先  
-XX:GCTimeRatio=n
- 暂停时间优先  
-XX:MaxGCPauseMillis=n
- 一般不使用



# 堆大小的调优

- 一般来说，堆越大越好

- 降低GC频率，但过大可能会增加单次GC的时间
- 对象更有可能成为垃圾

- 受硬件和操作系统限制

- 32位操作系统单个进程的最大可用内存为2G，64位无限制
- 小心平衡 New 和 Old 的比例

- 参数

- -Xms=1024MB：堆的最小值
- -Xmx=1024MB：堆的最大值
- 堆每次调整都会触发一次 **Full GC**

避免频繁调整，可以设置：

**-Xms = -Xmx**

等等，`-Xms == -Xmx` ？

**Not Always !**

- 设置-Xms为堆的预期大小
  - 堆调整的代价很大
- 如果内存允许，设置-Xmx为一个比-Xms更大的数值
  - **以防万一**
  - 也许系统负载比你想的要重
  - 随着时间的推移，数据量越来越大
  - .....
- 进行一次 Full GC & 堆调整 总比发生 OOM & 宕机 要好

# 新生代调优 – 大小

- 增大Eden的大小会.....

- 降低Minor GC的频率；
- 但不一定会增大Minor GC的时间

Minor GC的耗时和要拷贝的对象数量，即存活对象多少成正比

- 参数

- -XX:NewSize=1024MB : 新生代初始大小
- -XX:MaxNewSize=1024MB : 新生代最大值
- -XX:NewRatio=m : New和Old的比值
- -Xmn=1024MB : 新生代大小
- 出于性能考虑，一般使用-Xmn来固定新生代大小

# 新生代调优 – 晋升 (Promotion)

- 尽可能地让对象呆在Survivor中，使之在新生代被回收
  - 减少晋升到旧生代的对象
  - 降低旧生代GC频率
- 但是同时，避免长时间存活的对象在Survivor间不必要的拷贝
  - 增加Minor GC不必要的开销
- 不容易找到平衡点
  - 原则上：better copy more, than promote more
- -XX:SurvivorRatio=m：Eden和Survivor的比值

# 新生代调优 – 晋升 (Promotion)

- 对象晋升年龄的阈值：Tenuring Threshold

- 年龄标志位age，每熬过一轮GC对象年龄加1
- Serial Copying 和 ParNew 每次Minor GC后重新计算TenuringThreshold的规则：
  1. 参数-XX:TargetSurvivorRatio=n：minor GC后Survivor预期被占用的比例
  2. 计算Desired Survivor Size = Survivor大小 \* TargetSurvivorRatio；
  3. 统计存活对象的年龄，若在某个年龄上的对象总大小 > Desired Survivor Size，则TenuringThreshold = min(该年龄, MaxTenuringThreshold)；
  4. 否则TenuringThreshold = MaxTenuringThreshold；
  5. 下次Minor GC的阈值就以此为准
- 查看每次minor GC后年龄的分布和计算出来的TenuringThreshold：  
-XX:+PrintTenuringDistribution

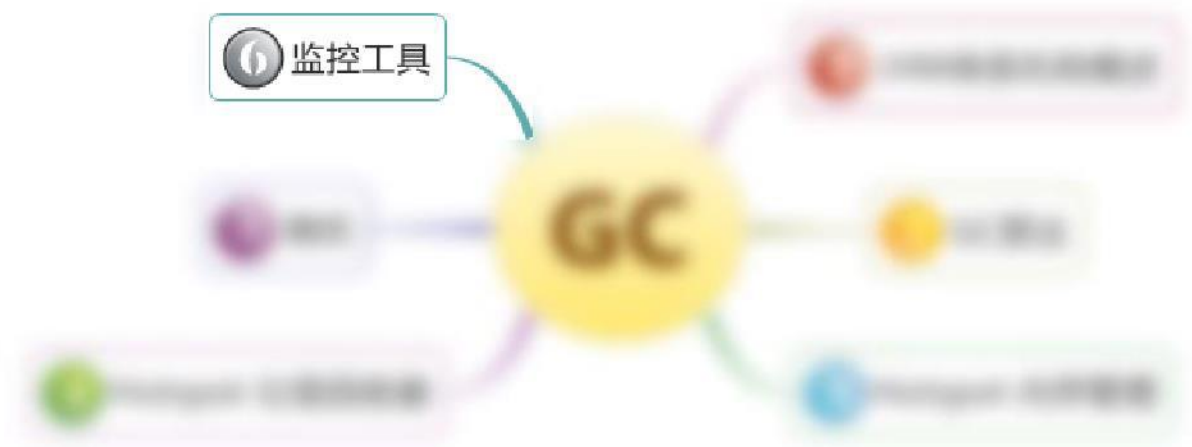


# 旧生代调优

- 尽可能地调优新生代先
  - 尤其要注意CMS中的Promotion
    - free list
    - 更容易出现内存碎片
- (对CMS) 在不紧要的时间段手动进行Full GC
  - 清理堆，压缩，减少内存碎片；
- 大小的平衡
  - 太大 --- 单次GC时间长；
  - 太小 --- GC频率高
- 硬件优化
  - 加CPU吧
- 程序优化，避免无用对象浪费Old空间
  - 去掉不必要的缓存
  - oracle 10g驱动时preparedstatement cache太大
- .....



# 导航



# 监控工具 I

- JVM参数

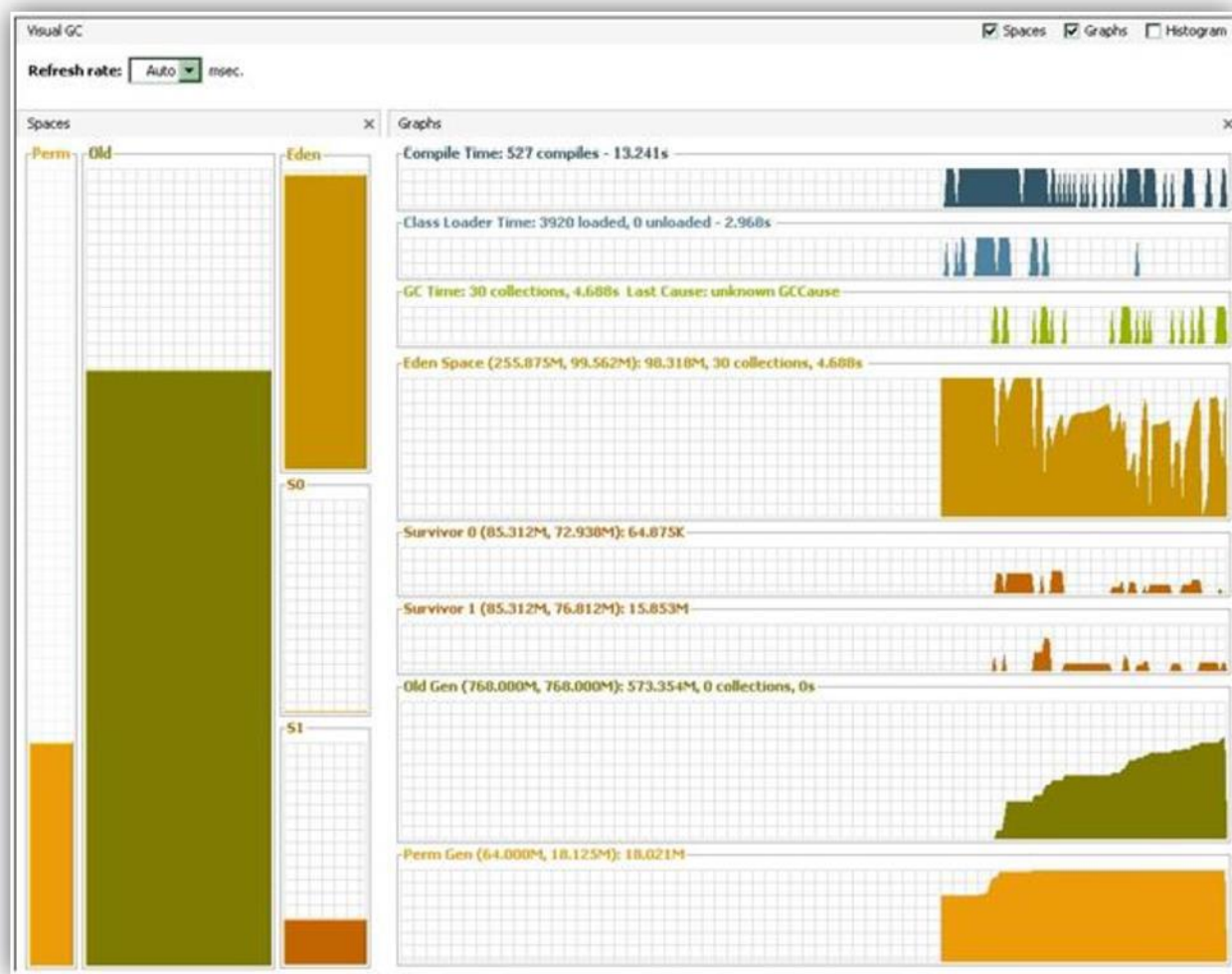
- -XX:+PrintGC 输出GC简要信息
- -XX:+PrintGCDetails 输出GC简要信息
- -XX:+PrintGCTimeStamps 输出GC时间戳
- -XX:+PrintGCApplicationStoppedTime 输出GC暂停时间
- -Xloggc c:/gc.log 输出到文件

- 命令行工具

- 查看内存使用情况、heap dump : jmap + jhat
- 查看GC情况 : jstat

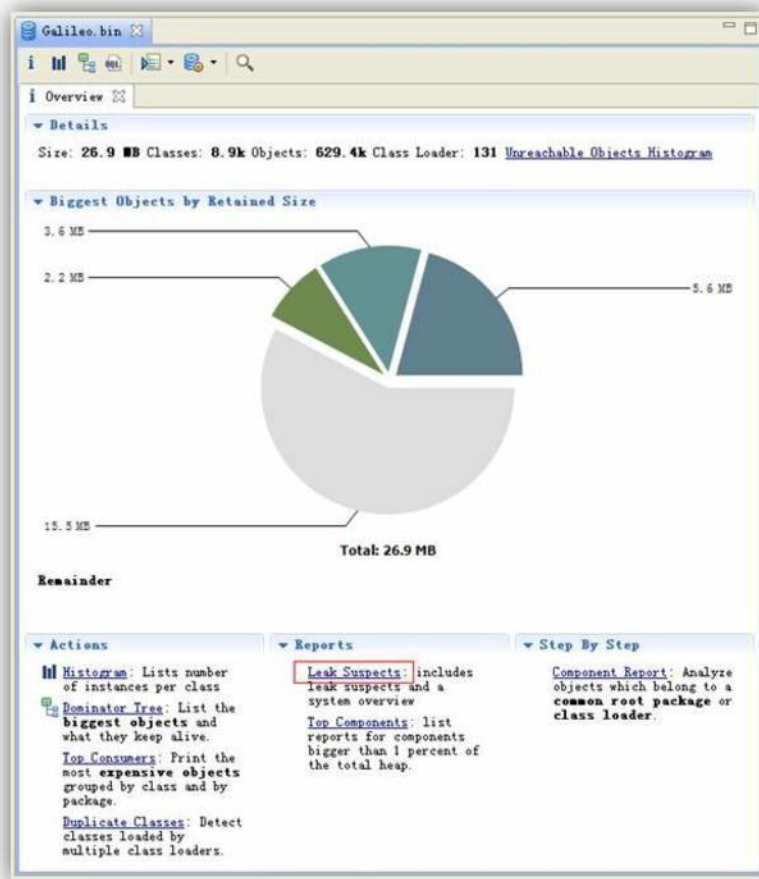
# 监控工具 II

## JVisualVM



# 监控工具 III

## MAT(Eclipse Memory Analyzer)



- ✓ 分析dump文件，快速定位内存泄露；
- ✓ 获得堆中对象的统计数据
- ✓ 获得对象相互引用的关系
- ✓ 采用树形展现对象间相互引用的情况
- ✓ 支持使用OQL语言来查询对象信息

# 总结

GC Tuning is a Dark Art

先尝试调优其他方面

实践是检验真理的唯一标准！

# End

**Thank you!**