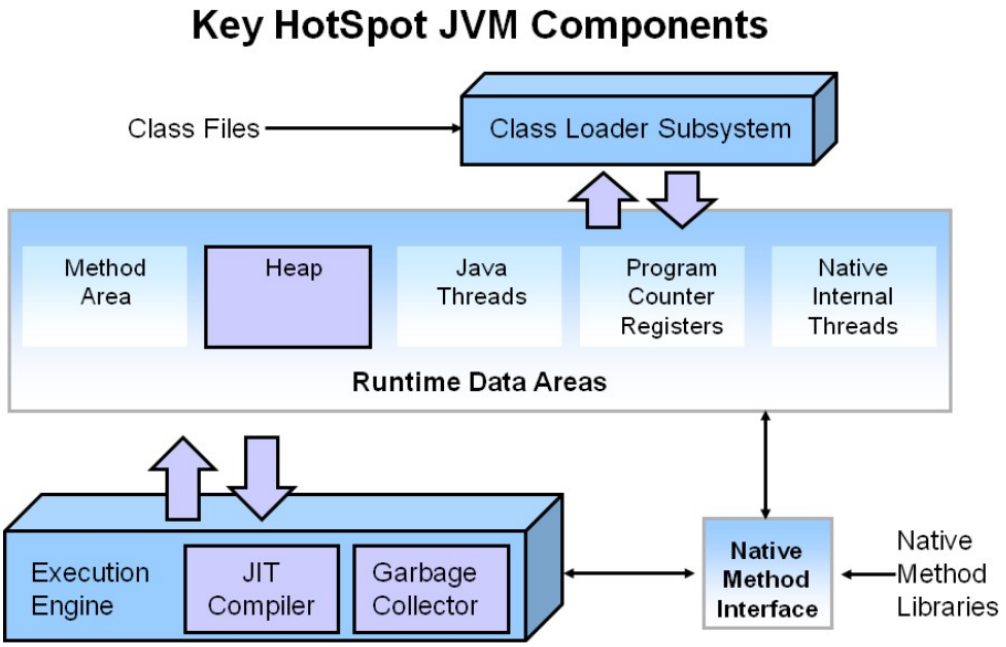
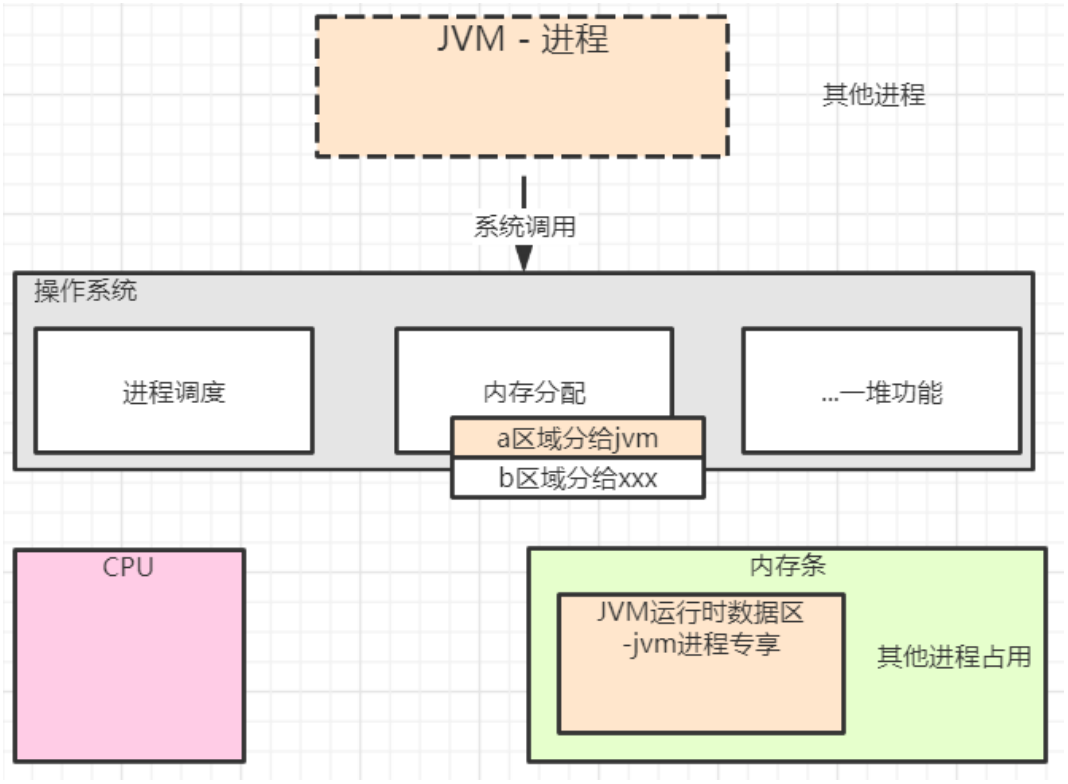
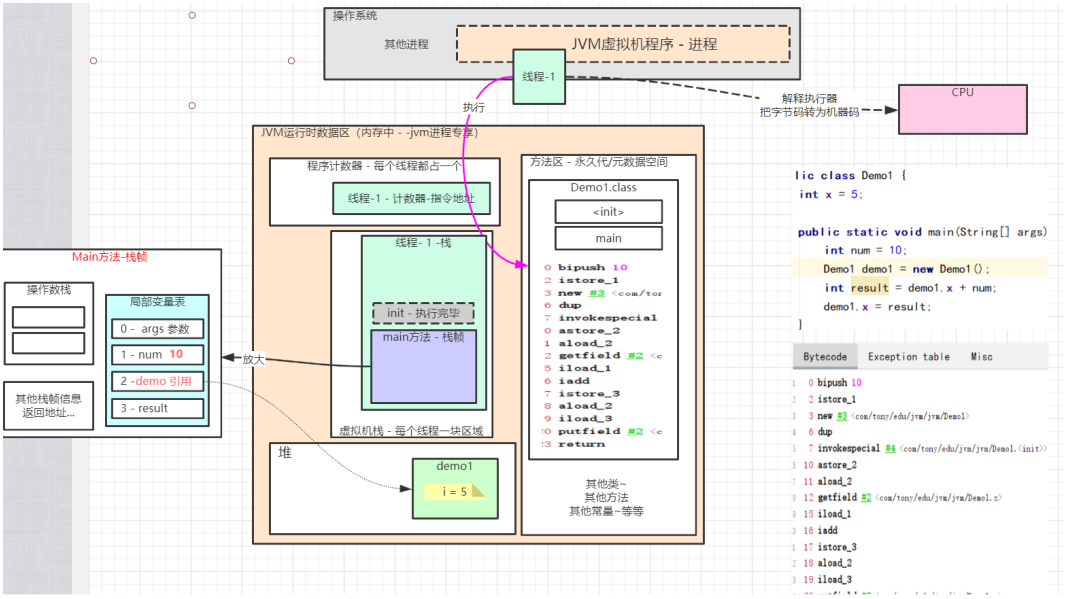
# GC相关

JAVA和JVM



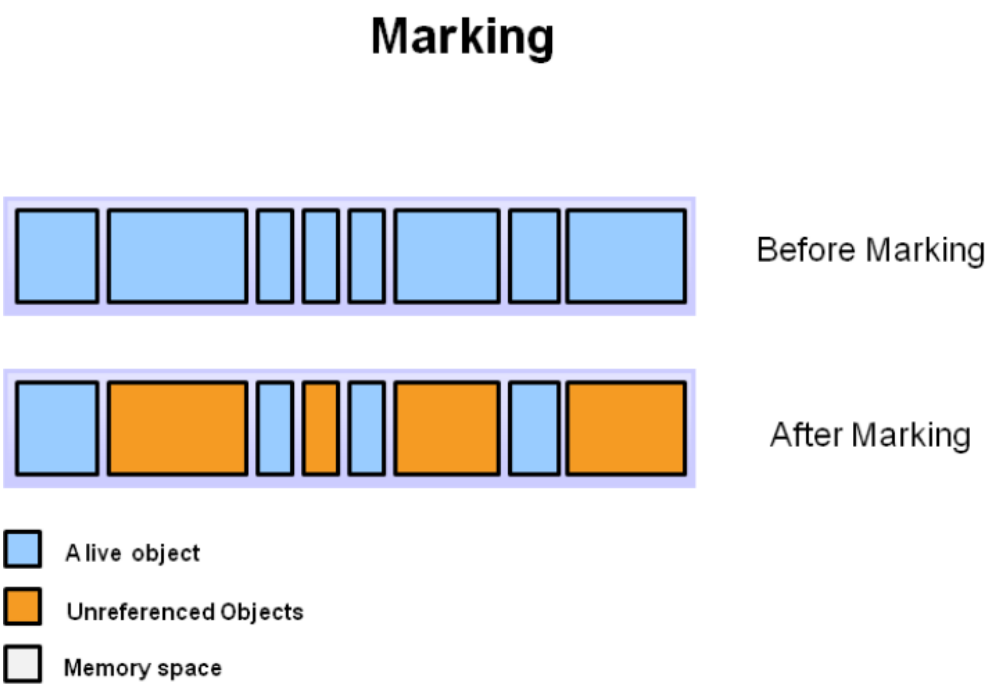






# 什么是 自动垃圾收集 GC机制

查看堆内存，识别正在使用的对象和未使用的对象以及删除未使用对象的过程主要为两大步骤：

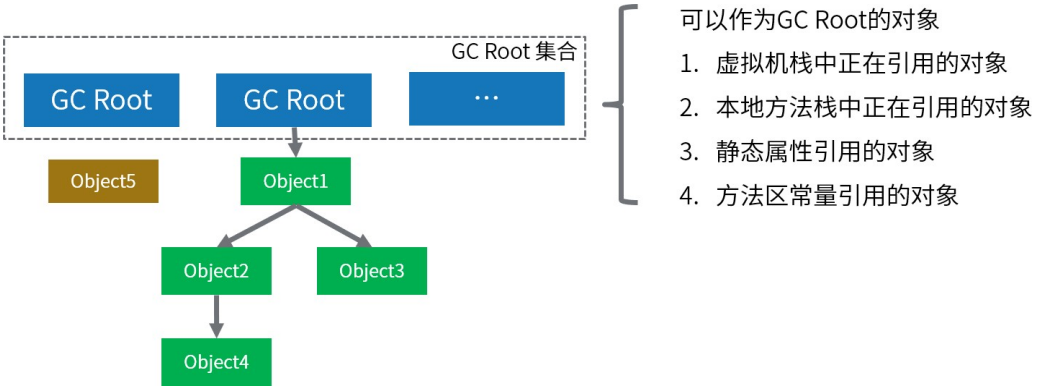
# 步骤1 - 标记




引用的对象以蓝色显示。未引用的对象以金色显示垃圾收集器找出内存中使用和未使用的对象。
如果扫描系统中所有对象，可能非常耗时。

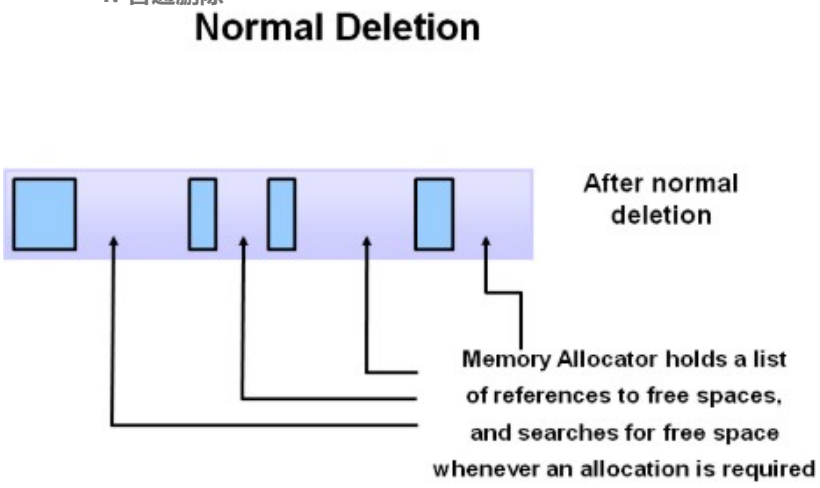
**思考**：JVM如何找出需要被回收的对象？ -- 引用计数 和 **可达性分析**
引用计数：记录着每一个对象被其它对象所持有的**引用数**。如果一个对象的引用计数为零，那么该对象 就变成了所谓的不可达对象。当一个对象被回收后，被该对象所引用的其它对象的引用计数都应该相应 减少。
存在在两个对象循环引用的问题。

目前主要使用 可达性分析算法。



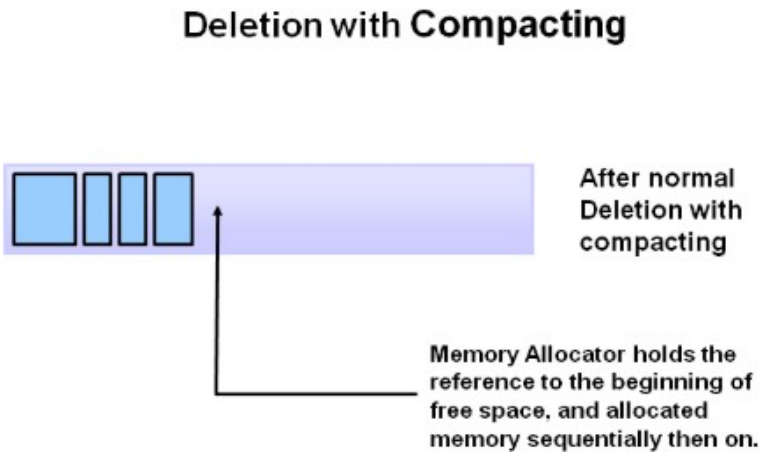

## 步骤2 - 删除

**1. 普通删除**




删除未引用的对象，内存分配器记录下可分配新对象的可用内存块。

**2. 压缩删除**




**为了提高性能**，除了删除未引用的对象之外，压缩其他的可用对象。移动引用对象，腾出连续的内存空 间。

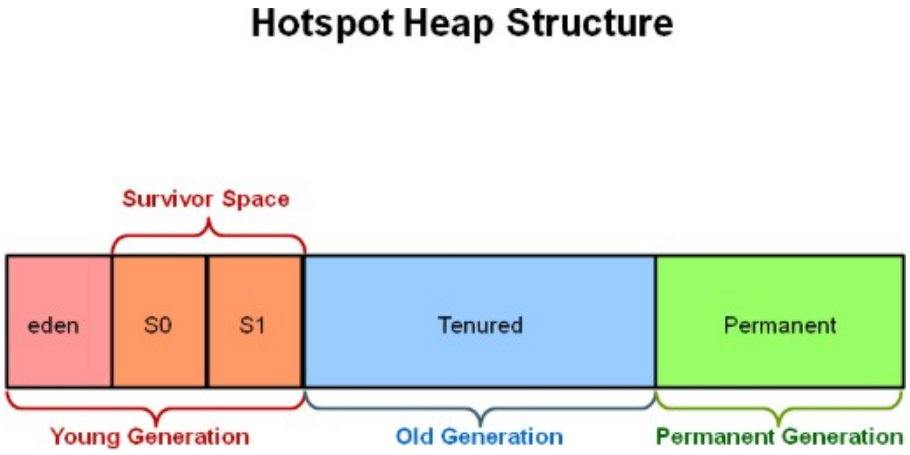
**3. 复制删除**

划分两块同等大小的区域，收集时将活着的对象复制到另一块区域。
拷贝过程中将对象顺序放置，就可以避免内存碎片化。 复制+预留内存，有一定的浪费。

## 总结

标记回收是个麻烦的过程，堆中的对象越多，则回收时间越长。

## 分代回收方案




根据实际情况分析得出，大多数对象的生命周期很短。
Hotspot把大部分的回收器，都把堆分成了3大类5个具体的区域。根据不同对象的生命周期，把对象存 在不同的区域。

## 针对不同的区域采取不同的垃圾回收方案，以此降低对象过多引起的回收时间过长 新生代

有足够空间的情况下，一般的新对象都分配在这个区域。

新生代下细分eden、s0、s1三个区域。
注1：S全称 survivor 幸存者 ， GC过程中幸存的对象从eden区挪到S区，寓意“幸存” 。

## 老年代

存放生命周期较长的对象。

a. 一般，在新生代经历了**N次**垃圾回收仍然健在的对象，会被挪到老年代。

b. 超过**阈值**的大对象分配的时候，也会被分配到老年代。

c. 新生代空间不够的时候，新对象也会被丢到老年代。（比如S区溢出）

注1： 对象在新生代经历了N次垃圾回收，对象中会有一个age年龄作为记录。具体多少次后进入老年代，是一个可配置参数，默认8次 ， 通过-XX:MaxTenuringThreshold控制
注2： 大对象的判定标准，通过-XX:+PretenureSizeThreshold控制

## 永久代【元数据空间】【特殊】

保存类方法、常量等系统运行所需的基础信息。

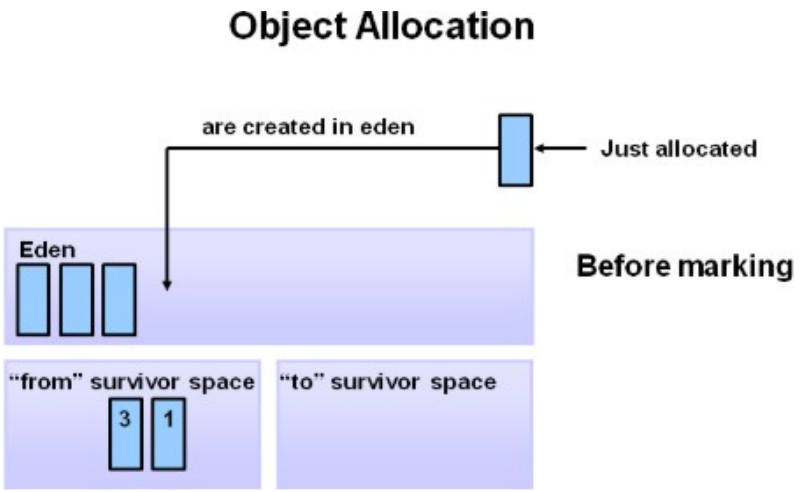
**注1**：对于不需要的类，垃圾回收过程中会卸载它。

**注1**：虚拟机规范中明确声明方法区不是堆区，但是具体的 JVM厂商实现过程中可自行发挥。

**注2**：1.8中已经剔除。新增元数据空间，在堆外内存中开辟空间存放“方法区”的数据。

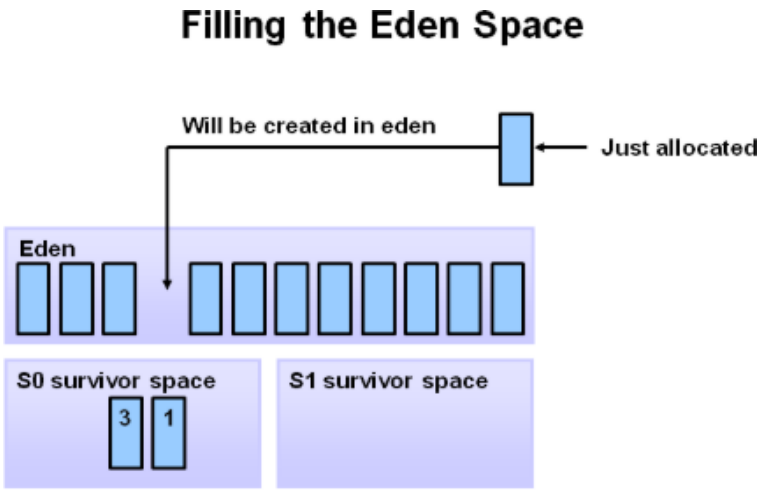
## HotSpo分代回收过程示意

## 对象分配



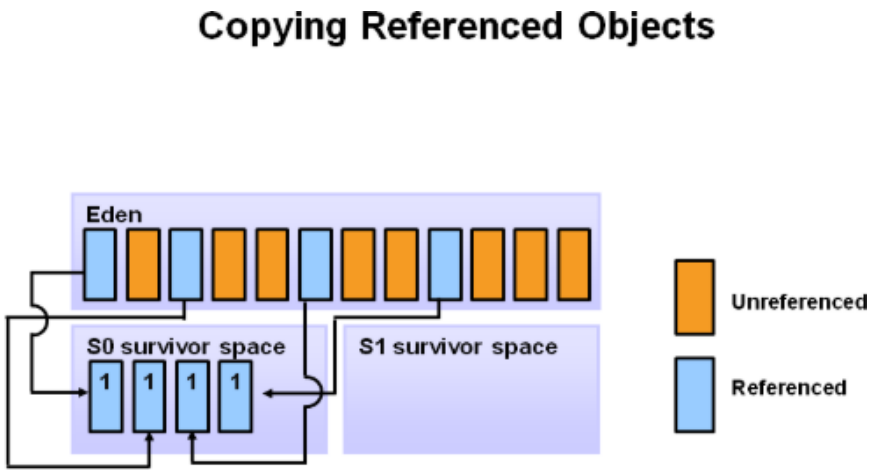

新对象分配到eden区

## eden满了




eden区满了，触发小范围垃圾回收(minor gc)。

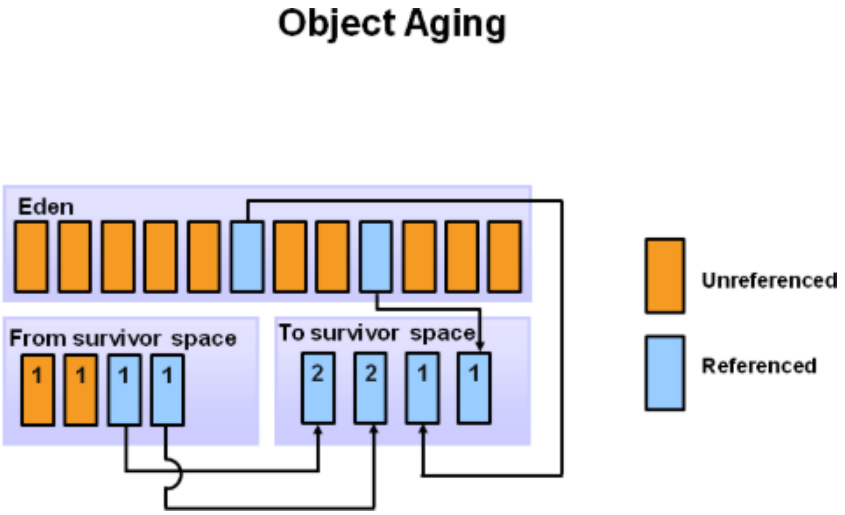
## survivor




第一次minor gc过程中删除未引用对象，将被引用的对象移动到S0区。此时S1区为空，eden区也会被清空。

一个被引用的对象挪动轨迹一般就是eden-> s0 -> s1

## 多次GC



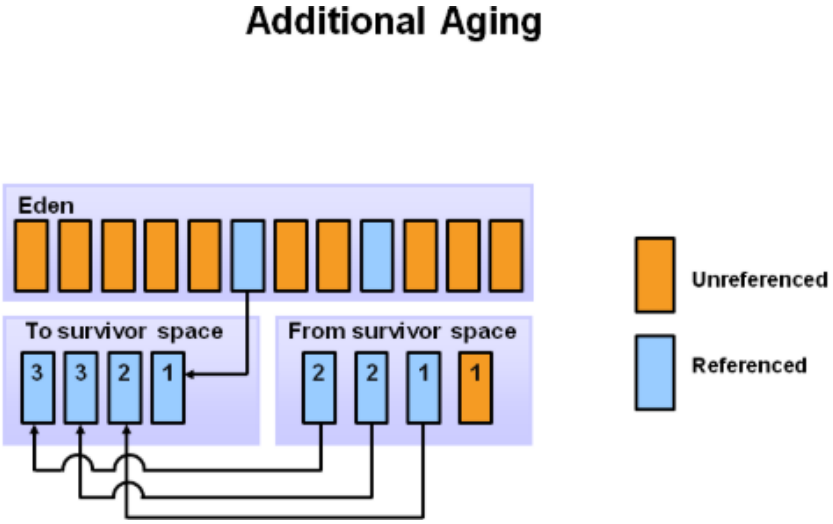

经历之前的minor gc后，eden区再次满了，则又会重复步骤3。

但是，在这种多次GC过程中，幸存的对象并非固定在某一个S区，而是在S0与S1之前互相腾挪。

（minor gc时如果某一个S区为空，则将幸存者对象移动至该区）

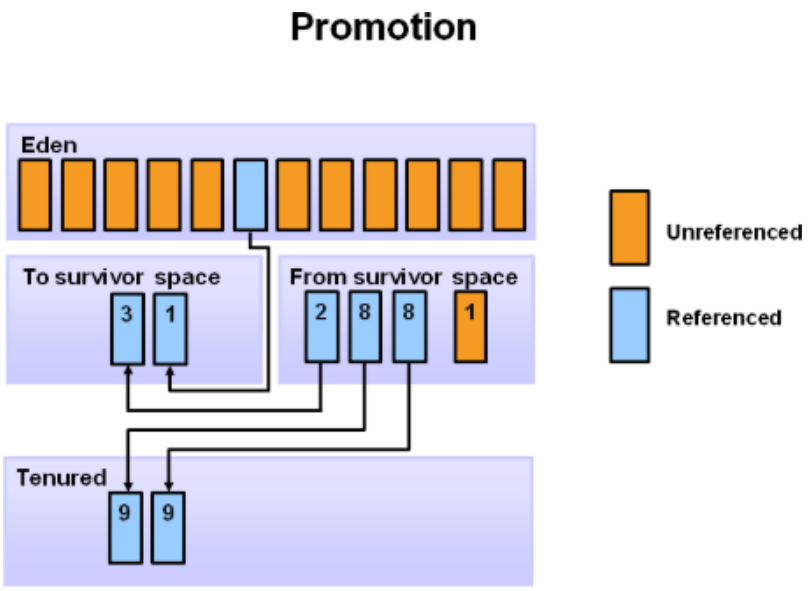
注： 多次minor gc时，对象内会记录经历GC的次数，也就是每个对象都有它自己的年龄(age)。

## 对象age



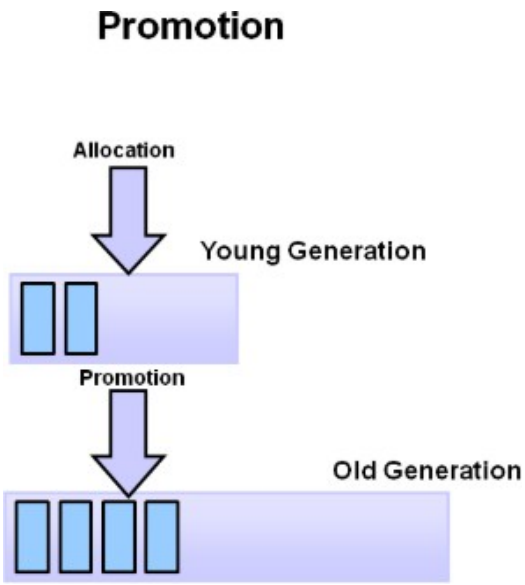

再次重复前面叙述的GC过程。S区内的对象，如果未被引用，也会被清除。

## 老年代



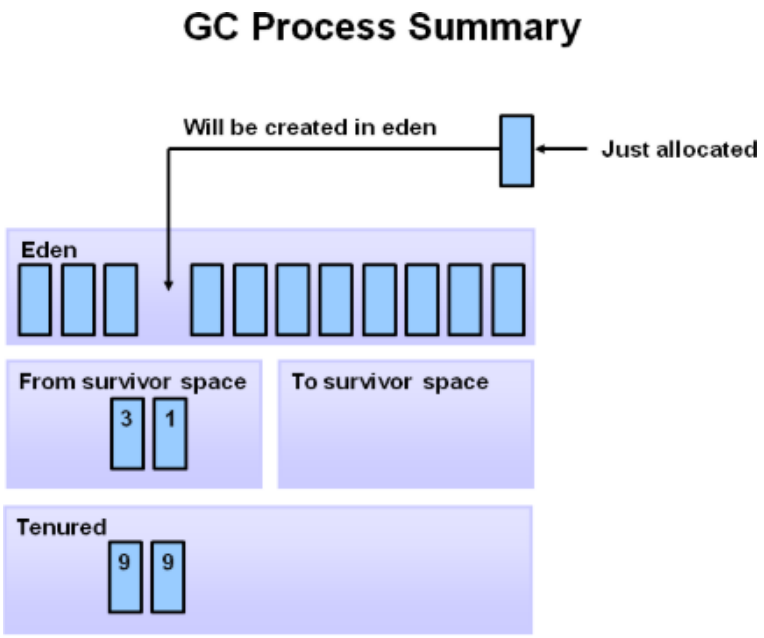

对象年龄达到一定阈值后，比如上图所示，age等于8，则对象挪动到老年代。

## 不断GC




对于新生代的回收就如前面所述，不断的会有对象产生，也会有对象消除，同时老年代也会被更多对象 充实。

## major GC




如果老年代慢慢满了，则会有针对老年代的大范围GC（major GC ）

FullGC： 包含整个堆区的清理，在hotspot中甚至包含了永久代（元数据空间）。

Major GC：针对老年代的垃圾清理。

FullGC 和 Major GC 有点让人傻傻分不清楚。在虚拟机规范和JVM厂商手册中没有明确的说明， 也没有具体的区分定义，明白区别即可。

有时候老年代满了，触发的就是full gc

## 写在最后

不论是哪一种GC，都会导致一种名为”**Stop The World(停止世界)**“的事情发生。

STW意味着GC过程中所有应用程序线程都将停止一定的时间，意味着你的业务代码执行被停顿。

所以如何让GC机制更少的执行以及减少GC过程中STW停顿时间，成为JVM厂商及JVM优化团队不断追求 的目标。

## 垃圾回收器

**垃圾回收器**：JVM实现中对垃圾回收这种机制的具体实现。有很多种不同的方法来实现GC机制。 根据垃圾回收器的工作模式，分为下面三大类：

## 串行GC

java -Xmx1024m -Xms1024m **-XX:+UseSerialGC** -jar demo.jar JVM中通过1个线程，来完成前面所述的垃圾回收机制的具体过程。

从名字和定义上即可推断出，它适用在客户端或者小机器(如物联网设备)上。

jvm以 client模式运行时，则使用SerialGC

## 并行GC

java -Xmx1024m -Xms1024m **-XX:+UseParallelGC -XX:+UseParallelOldGC** -jar demo.jar

并行GC也被成为吞吐量优先垃圾回收器。多个线程充分利用现代处理器的优势，加快GC速度。

## 并发GC（CMS）

java **-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2** -jar demo.jar CMS设计将垃圾回收分为更细致的步骤（初始标记、并发标记、重新标记、并发清除）。

在特定的步骤下，不会触发STW，以此来降低GC过程中STW停顿的时间。

目前很多互联网公司的Web程序的服务器上用的就是CMS 1.9版本中标记为废弃，不再继续发展维护，可能被新版抛

## 总结

对垃圾回收器的优化升级，一直是各个JVM团队在做的事情，不再继续描述。

G1 - Java 7提供针对大堆设计的**并发垃圾回收器**

ZGC - Java 11提供的，数据看起来非常niubility的垃圾回收器

# GC调优-理论篇

## 基础知识

小型系统基本可用忽略GC花费的时间，而在大型系统上可能会是瓶颈。垃圾回收器性能评估点

1. 吞吐量 - 花在GC上面的时间占比。
2. STW暂停时间

JVM目前**默认**启用吞吐量优先的**parallel-并行垃圾回收器**。

因为STW时间一般很短暂，对于WEB网络应用而言可以容忍，毕竟一点点的网络延迟可能就已经 超过了GC时间。调优的过程是根据业务系统的具体情况进行参数调整的，没有固定的值。需要在长时间贴近生产的压力 测试决策出最优方案。

## 【调优思路】堆内存及分代区域大小控制

影响垃圾收集性能的最重要因素是总可用内存。

由于GC是在某一区域占满发生的，因此吞吐量与可用内存量成反比。这一点我们可用看出，所谓调优就是在有限条件下追求最优解。

## 堆的总大小控制

-Xms 初始堆大小

-Xmx 最大堆大小，默认为物理内存的1/4（server模式）

-XX:MinHeapFreeRatio=【**不常用**】用意堆扩大后就不会低于这个百分比

-XX:MaxHeapFreeRatio=【**不常用**】超过70%，GC后会收缩堆的大小

大多数情况下，设置-Xms和-Xmx相同的值，避免JVM扩容和收缩堆带来的额外工作量

client模式：默认最大堆是物理内存的一半，最大为192mb，否则为物理内存的四分之一，最大 为1gb。

例如，如果您的计算机有128 MB的物理内存，那么最大的堆大小是64 MB，如果大于或等于1 GB

的物理内存，那么最大的堆大小是256 MB。在JVM初始化期间分配的堆大小要小得多

server模式：与clent模式类似，只是默认值可能会变大。在32位jvm上，如果有4 GB或更多的物理内存，则默认的最大堆大小可以达到1 GB。在64位jvm上，如果有128 GB或更多的物理内存， 则默认的最大堆大小可以达到32 GB。

默认动态计算，启动时加参数-XX:+PrintFlagsFinal，查看计算结果MaxHeapSize

## 新生代大小控制

-XX:NewRatio=2 **默认**新生代和老年代比例**1:2**

理论上新生代越大，则GC次数越少。同时也意味着老年代变小，根据对象的生命周期分布而 定。

-XX:SurvivorRatio=8 **默认**情况下Eden : from : to = **8 : 1 : 1**

S区太小，则在minor gc时可能导致对象溢出到老年代

## 垃圾收集器选型

HotSpot VM提供了多个垃圾收集器（目前常见7种），每个垃圾收集器满足满足大型和小型应用程序不同的需求。

GC实现内存管理主要的操作就是三个

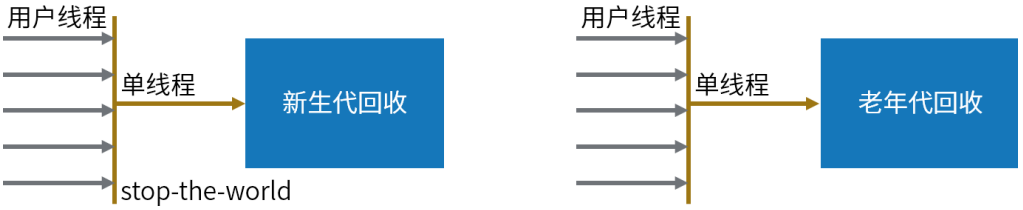
将新对象分配给新生代，生命周期长的对象丢到老年代

通过标记查找活跃对象。(堆占用率超过阈值，则开始标记)

通过复制压缩方式来恢复可用内存（例如S0,S1之间的复制方式）

花费了总时间的98%以上，而回收不到2%的堆，则抛出OutOfMemoryError:GC overhead limit exceeded异常

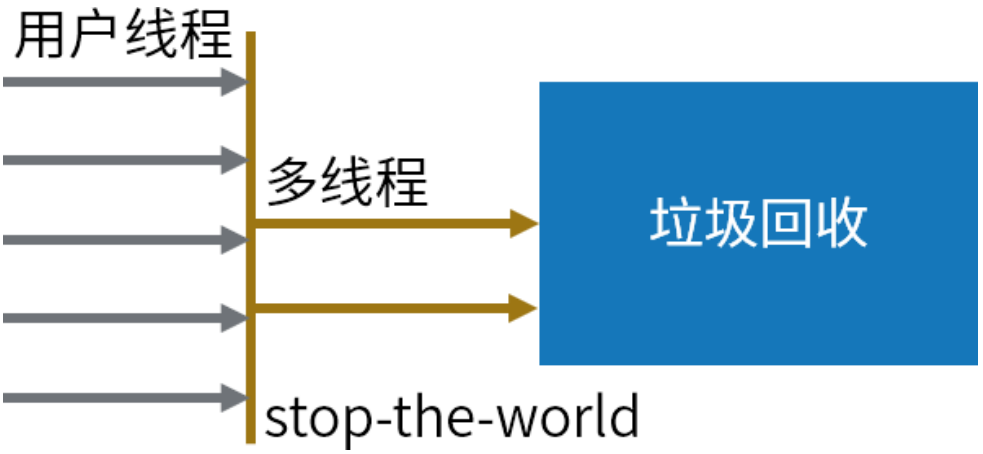
## 两种串行收集器




新生代 -XX:+UseSerialGC 复制算法

老年代 -XX:+UseSerialOldGC 标记- 整理（Mark-Compact）算法单线程工作，适用于单处理器机器，基本可用忽略

## 并行收集器




【吞吐量优先】【JVM默认】

整体和Serial比较相似，区别是并行进行； 并行线程数量控制 -XX:ParallelGCThreads=

### ParNew

Serial GC 的多线程版本

通过 -XX:+UseParNewGC 启用，配合CMS使用

### Parallel GC

新生代 -XX:+UseParallelGC

老年代默认开启-XX:+UseParallelOldGC，**关闭**的方式 **-XX:-UseParallelOldGC**

## Parallel GC自动调整机制

1. 单次最大暂停时间(STW时间)
   -XX:MaxGCPauseMillis=

推理：回收区域内存占用越大，GC时间越长，在限制了单次GC时长的情况下，则GC次数会增 加。

2. 应用程序吞吐量目标
   -XX:GCTimeRatio=

参数的含义： 花在**GC**上面的时间占比。

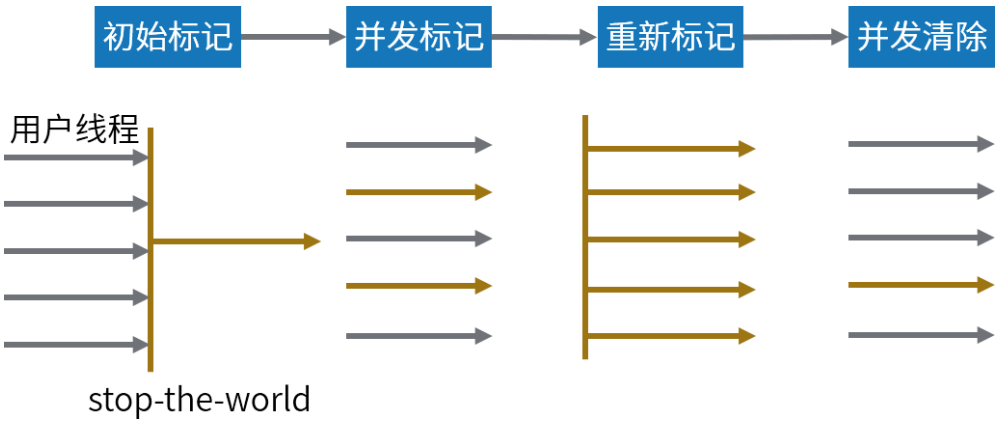
如果参数设置为19，则垃圾回收总时间小于应用程序总运行时间的5%。计算公式： GC总时长限值 = 1/(1+GCTimeRatio) * 程序总时长

这两个参数配置后，JVM会根据运行情况，**自动动态调整**堆中分代各区域的大小。 在这种动态调整的情况下，不要设定堆的最大值，最好让JVM自行选择。

同时配置，优先保证的顺序：最大暂停时间目标 > 吞吐量目标

## 并发收集器

**CMS**




响应时间优先，目标是让STW时间较短。整个过程两次STW

-XX:+UseConcMarkSweepGC 启用CMS收集器

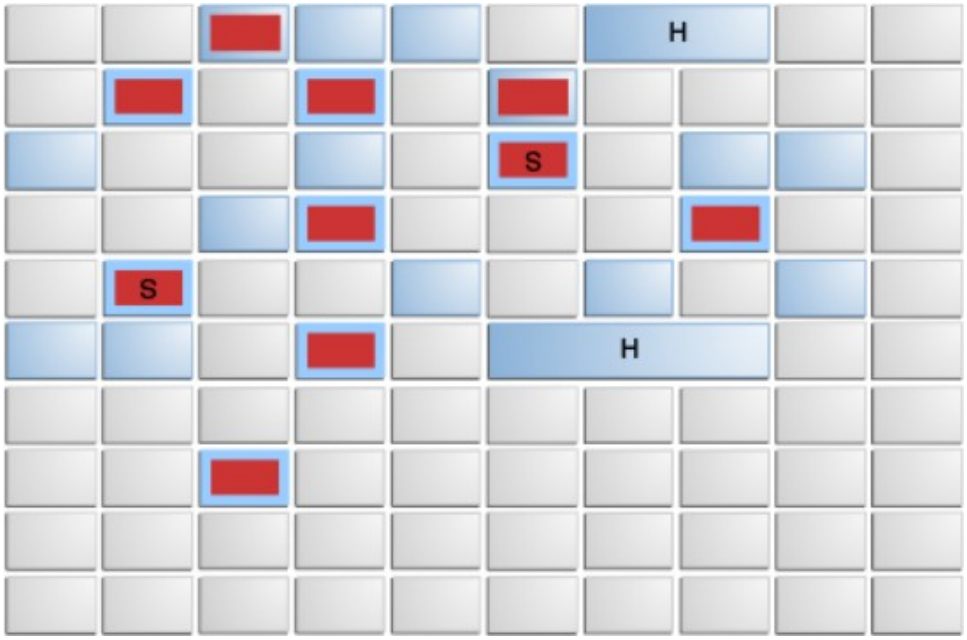
CMS并发模式故障：concurrent mode failure

粗犷理解：上一轮还没回收完毕，业务应用线程产生了新的垃圾导致老年代又不够用了，导 致回收失败的场景

**【降级】**故障发生时，切换为Serial Old收集器来进行老年代的回收。其他不足之处

实际运行中，GC和业务线程同时运行，把一部分CPU资源用于GC，同样会导致业务处理过慢，所 以CMS这个设计本身有一定问题。 这也是新版本中考虑废弃的原因

## Garbage-First（G1）




针对大堆设计的回收器。官方建议6GB以上再使用

G1计划是CMS的替代品，在工作流程上和CMS有一定相似，也是并发收集器。

G1完全改变了以往的堆分代划分方式。（整个堆划分出若干个大小相等的区域region，区域依然有老年 代 新生代 S区等属性，但是没有明确划分，也就是老年代和新生代的对象可能同时存在一个区域）

区域划分的更细，G1会记录各个小区域内存使用情况，再针对单个区域进行垃圾回收，不再是分析整个 完整的堆，意味着单个区域的回收更快。

目前仅常见于部分中间件中使用。例如：RocketMQ服务器阿里推荐使用G1。

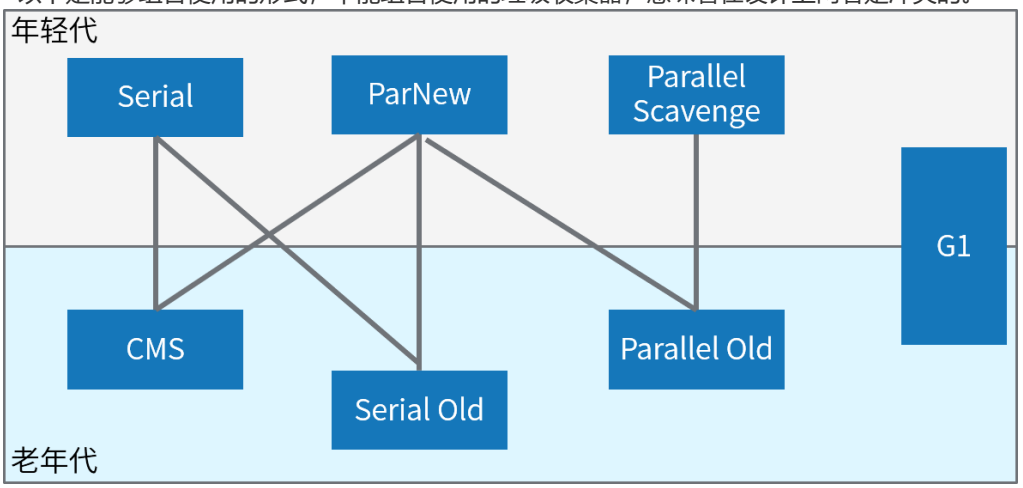
参考资料 https://www.oracle.com/technical-resources/articles/java/g1gc.html

## 一般选型准则

1. 如果是小程序100MB大小这种堆，或者运行在单处理器，则使用用串行收集器。
2. 能接受1秒甚至更长，选择并行。
3. 如果响应时间比整体吞吐量更重要，并且要求STW必须小于1秒，建议用并发。

实际应用过程中，也可以针对业务特点，7种常见的垃圾收集器组合使用。

以下是能够组合使用的形式，不能组合使用的垃圾收集器，意味着在设计上两者是冲突的。




# GC调优-实战篇

基于JAVA8编写，调优就是一个不断调整、优化、测试的反复过程。

一种优化的方式就是**提升硬件配置**，有钱着急用，肯定选升级配置或者加机器。但出于成本原 因，优化程序本身的性能必然是要做的。

系统性能优化就是压榨机器，在**资源有限**的情况下让系统性能更优。通常我们的Web系统生产环境的机器**内存范围是6G~16G，惯8G。**

## 确定优化目标

针对常见的web程序，在进行GC优化的时候，通常我们假定业务代码的执行是足够稳定的，否则 建议先优化业务流程或业务组件。

响应时间 - 例如：一个请求从接收到处理完毕的时间。（RT） 吞吐量 - 在一定时间内，能够处理完毕多少请求（QPS/TPS）。

## GC是否需要优化?

一切要结合具体的业务系统和场景进行评估

## 响应时间角度

要求：所有用户请求必须在1000ms内完成

对于响应时间的要求，根据经验来说，我们要求**GC占用的时间不超过10%。**

为了方便理解，我们假定一次请求最多经过一次GC。也就是一个请求要求1000ms完成，则GC导致

STW的时间不能超过100ms，只要在这个范围内，就是符合要求的（也就是无需优化）。

## 吞吐量角度

要求：一定时间内处理完多少次请求。例如 1分钟处理完毕6000次请求，则平均每秒处理100次。

追求吞吐量的情况下，我们就不说去细究每一次GC的时间，我们要统计的就是这段时间内GC消耗的总 时间。同样，根据经验值，GC占用的总时间不应超过10%。

通常都是两者都需要权衡考虑。

## 优化响应时间

假设一个请求要求3s内完闭，这个耗时=业务代码执行时间+GC-STW时间如果STW很耗时，那就优化GC，否则优化业务代码

## 回顾常用四种垃圾收集器的组合

| Young | Tenured | JVM options |
| --- | --- | --- |
| Serial | Serial | -XX:+UseSerialGC |
| Parallel Scavenge | Parallel Old | -XX:+UseParallelGC -XX:+UseParallelOldGC |
| Parallel New | CMS | -XX:+UseParNewGC -XX:+UseConcMarkSweepGC |
| G1 | G1 | -XX:+UseG1GC |

## 实际测验

通过日志进行分析，调整GC对应参数~

# GC番外篇 – 可达性相关

1. 强引用(StrongReference): 最常见的普通对象引用， 只要还有强引用指向 一个对象，就不会回收。
2. 软引用(SoftReference): JVM认为内存不足时，才会去试图回收软引用指向的对象。(缓存场景)
3. 弱引用(WeakReference): 虽然是引用，但随时可能被回收掉。
4. 虚引用(PhantomReference): 不能通过它访问对象。供了对象被 finalize 以后，执行指定逻辑的机制(Cleaner)

1. 强可达(Strongly Reachable): 一个对象可以有一个或多个线程可以不通过各种引 用访问到的情况。
2. 软可达(Softly Reachable): 就是当我们只能通过软引用才能访问到对象的状态。
3. 弱可达(Weakly Reachable): 只能通过弱引用访问时的状态。当弱引用被清除的时候， 就符合销毁条件。
4. 幻象可达(Phantom Reachable): 不存在其他引用，并且 finalize 过了，只有幻象引用指向这个对象。
5. 不可达(unreachable): 意味着对象可以被清除了。

# OutOfMemoryError详解

## Java heap space


java.lang.OutOfMemoryError: Java heap space

## 问题描述

代码中视图向jvm申请内存空间，但是没有足够的空间。
注意：机器物理空间足够，但是JVM的堆大小限制，也会导致出错

## 原因分析

1. 最直白的原因就是你配置的堆内存太小，或者你的机器内存不够。
2. 用户量请求量或者数据处理量上来后，程序所需的资源，远远大于平常，导致JVM堆内存不够用。
3. 内存泄露。不知情的情况下，内存被某些功能中的对象所占用了。通常是编程过程中的错误方式导 致对象使用后无非被回收。

## 实例代码

堆内存太小示例

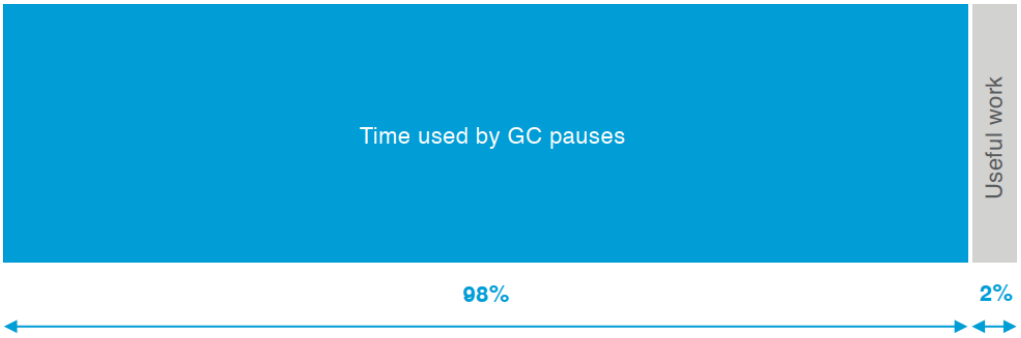
内存泄露示例

## 解决思路

第一种最简单，配置更大的堆内存： -Xmx

但是实际情况更复杂，比如**内存泄露**，你需要利用JVM调试工具，监控工具来分析堆内存中具体的情 况。例如jmap、jvisualvm、mat等等工具

# GC overhead limit exceeded




OutOfMemoryError：GC overhead limit exceeded

## 问题描述

当GC花费了程序运行总时间的98%以上，而回收不到2%的堆，则抛出该异常。

## 原因分析

程序运行时内存不够用，触发GC任务执行。
GC工作会导致一种名为" **Stop The World(停止世界)** "的事情发生。
STW意味着GC过程中所有应用程序线程都将停止一定的时间，意味着你的业务代码执行被停顿。

## 实例代码

## 解决思路

**思路和上面的问题类似**

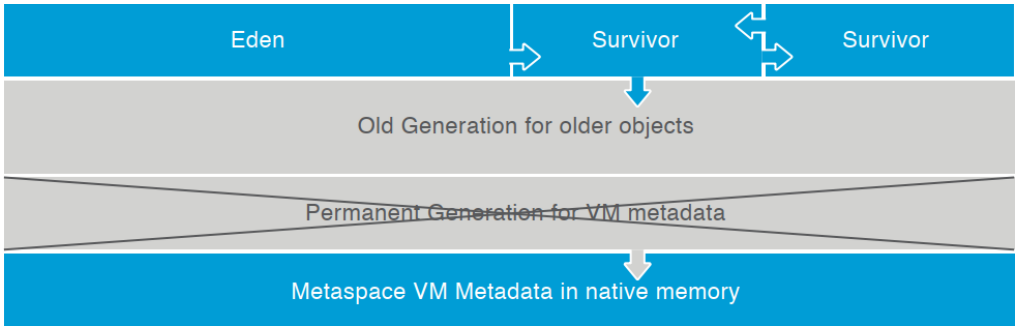
这个异常不是很容易重现。因为GC时也意味着堆内存不够，可能实际抛出的是更常见的java.lang.OutOfMemoryError: Java heap space，比如你把上面这个例子堆内存设置为16m或者更大的时候。
也不要尝试通过 -XX:-UseGCOverheadLimit 参数关闭这个功能，否则导致无法看到

java.lang.OutOfMemoryError完整的信息。

# Metaspace 或者 Permgen space

1.8以前


1.8开始




<1.8 java.lang.OutOfMemoryError: Metaspace
1.8+ java.lang.OutOfMemoryError: PermGen space
为什么会有两种，因为JAVA8开始，将以往的 PermGen 改为 Metaspace

## 问题描述

方法区内存不足
永久代/元数据空间用来存放类的名称和字段、带有方法字节码的方法、常量池信息、与类和关联的对 象数组和类型数组、JIT及时编译优化

## 原因分析

从定义可以推断，PermGen的大小需求既依赖于装入的类的数量，也依赖于此类声明的大小。 因此，我们可以说这个异常主要原因是太多类或太大类被加载到永久代。

## 实例代码

字节码动态生成类

WEB服务器运行期间多次部署项目
例如Tomcat服务器反复部署项目，如果代码编写有问题，导致类加载器及类没有卸载，就可能导致方 法区相关的溢出。

## 解决思路

启动过程，通过加大PermGen 大小

java -XX:MaxPermSize=512m com.tony.edu.Demo java -XX:MaxPermSize=512m com.tony.edu.Demo

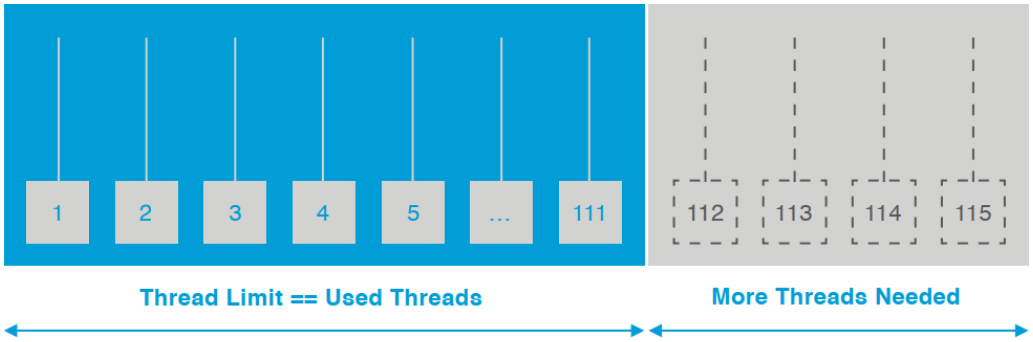
运行过程中

保存异常时的内存快照，通过**Eclipse MAT**等工具进行事后分析。

在分析器中，可以查找重复的类，特别是那些加载应用程序类的类。继而查找到找到当前活动的类加载 器。

对于非活动的类加载器，通过从非活动类加载器获取到GCroot的最短路径来确定具体是哪段代码，甚至 定位到哪一个第三方包

## Unable to create new native thread




java.lang.OutOfMemoryError: unable to create new native thread

## 问题描述

Java应用程序已经达到了它可以启动的线程数的限制

## 原因分析

JVM请求从操作系统中创建新线程时，当底层操作系统不能分配一个新的本地线程，这个OutOfMemoryError将被抛出。

1. 在JVM中运行的应用程序请求一个新的Java线程
2. JVM本机代码将请求代理为操作系统创建一个新的本机线程
3. 操作系统试图创建一个新的本机线程，它需要为线程分配内存
4. 操作系统将拒绝本机内存分配，因为32位Java进程大小耗尽了它的内存地址空间(例如进程大小限 制)，或者操作系统的虚拟内存已经完全耗尽
5. 最终抛出异常

不同的平台有不同的表现

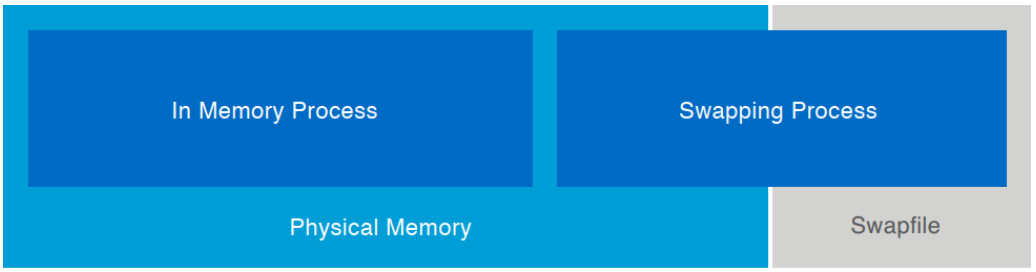
## 实例代码

## 解决思路

linux服务器上通过命令查看，可以通过修改配置加大这个阈值。

这问题，大部分情况下上不是系统配置问题，通常意味着你的程序有重大错误...

## Out of swap space?




java.lang.OutOfMemoryError: Out of swap space?

## 问题描述

JVM请求的总内存大于可用物理内存的情况下，操作系统开始将内容从内存交换到硬盘驱动器。 物理内存和交换空间的缺乏，**分配失败。**

## 原因分析

一般来说，JVM有自己的GC机制，如果出现这个异常，通常意味着是下面的原因。 操作系统配置的交换空间不足。

系统上的另一个进程正在消耗所有内存资源

当然不排除有程序代码原因

## 实例代码

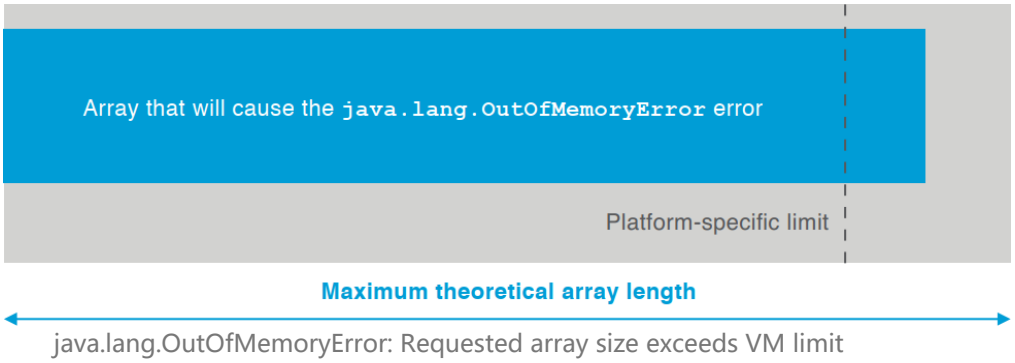
## 解决思路

最简单的解决方法是增加交换空间。
例如在Linux中，使用以下命令来实现，创建并附加一个大小为640MB的新swapfile:

### 一般来说，这个异常意味着你要升级内存

**或者意味着你的一台机器不够用了，用户请求过多或者处理的数据量很大，需要集群。**

### Requested array size exceeds VM limit




java.lang.OutOfMemoryError: Requested array size exceeds VM limit

## 问题描述

搞了一个大数组。分配一个比Java虚拟机所能支持的更大的数组。

## 原因分析

错误是由JVM中的本机代码抛出的。当JVM执行特定于平台的检查时，它发生在为数组分配内存之前:所 分配的数据结构在这个平台中是否可寻址。

## 实例代码

```
byte[] array = new byte[Integer.MAX_VALUE];
```

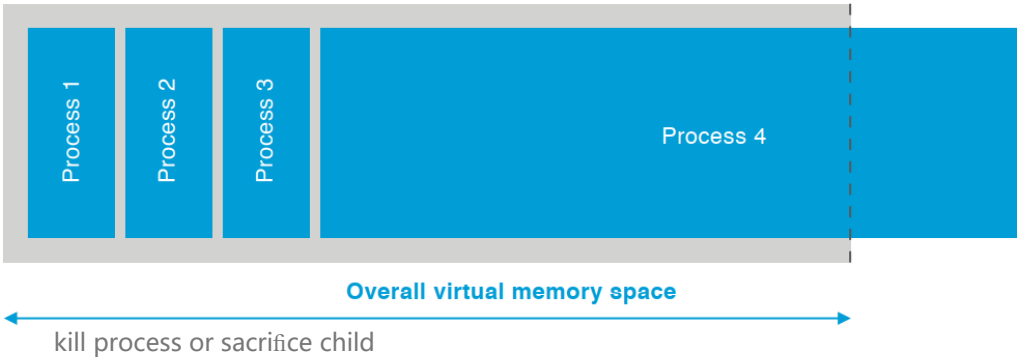
## 解决思路

代码问题，不要搞这么长的数组。

# Direct buffer memory

堆外内存溢出

# Kill process or sacrifice child




## 问题描述

操作系统的保护机制，内存不够时，干掉内存占用最大的进程看操作系统日志有提示

## 原因分析

Linux是以进程为单位管理操作系统相关资源的（如文件描述符、线程数量、内存等等）。
Linux中有一个称为“Out of memory killer”内核工作进程，一般情况下，当操作系统检测到低内存情况时，将激活Out of memory killer杀死内存占用非常大的用户进程。(**注：操作系统会根据自身的规则对进程进行评分，并选择得分最低的进程**)

## 实例代码

## 解决思路

1. 既然是JAVA程序，那么**尽量控制堆的大小**，让它不要对整个服务器造成很大的影响。所以我们通常会限制最大堆不超过整个操作系统资源的80%（当然，如果你的系统内存非常大，你可以将堆最大值控制到90%甚至更高）。
2. 系统kill你的jvm进程，一般都会留下痕迹（系统日志 /var/log/message） 关键字“kill”

# Synchronized关键字

## 用法

Java层面用法： 方法(静态/非静态)，代码块(实例对象/Class对象)

## 字节码层面原理

monitorenter、monitorexit

面试题:为什么出现两次monitorexit？ 出现异常时走的流程。

## C层原理

```
/* monitorenter and monitorexit for locking/unlocking an object */

CASE(_monitorenter): {
  oop lockee = STACK_OBJECT(-1);
  // derefing's lockee ought to provoke implicit null check
  CHECK_NULL(lockee);
  // find a free monitor or one already allocated for this object
  // if we find a matching object then we need a new monitor
  // since this is recursive enter
  BasicObjectLock* limit = istate->monitor_base();
  BasicObjectLock* most_recent = (BasicObjectLock*) istate->stack_base();
  BasicObjectLock* entry = NULL;
  while (most_recent != limit ) {
  if (entry != NULL) {
    entry->set_obj(lockee);
    int success = false;
    uintptr_t epoch_mask_in_place = (uintptr_t)markOopDesc::epoch_mask_in_p

    markOop mark = lockee->mark();
    intptr_t hash = (intptr_t) markOopDesc::no_hash;
    // implies UseBiasedLocking
    if (mark->has_bias_pattern()) {
    // traditional lightweight locking
    if (!success) {
    UPDATE_PC_AND_TOS_AND_CONTINUE(1, -1);
  } else {
    istate->set_msg(more_monitors);
    UPDATE_PC_AND_RETURN(0); // Re-execute
  }
```

偏向锁

轻量级锁

1. 查看JVM字节码解释相关的C++源码
openjdk-jdk8u-jdk8u\hotspot\src\share\vm\interpreter\bytecodeInterpreter.cpp
关键字：CASE(_monitorenter)

2. 查看运行时runtime相关锁的源码
openjdk-jdk8u-jdk8u\hotspot\src\share\vm\interpreter\interpreterRuntime.cpp

```
//------------------------------------------------------------------------------------------------------------------------
// Synchronization
//
// The interpreter's synchronization code is factored out so that it can
// be shared by method invocation and synchronized blocks.
//%note synchronization_3

//%note monitor_1
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  if (PrintBiasedLockingStatistics) {
    Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
  }
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
  }
  assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
         "must be NULL or an object");
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

关键字：InterpreterRuntime::monitorenter

3. 查看锁的fast 和 slow两类方式
openjdk-jdk8u-jdk8u\hotspot\src\share\vm\runtime\synchronizer.cpp

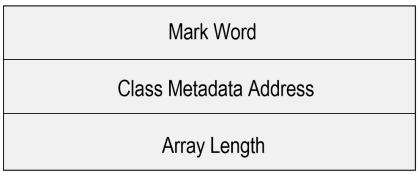
```
// -----------------------------------------------------------------------------
//  Fast Monitor Enter/Exit
// This the fast monitor enter. The interpreter and compiler use
// some assembly copies of this code. Make sure update those code
// if the following function is changed. The implementation is
// extremely sensitive to race condition. Be careful.

void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock, bool attempt_rebias, TRAPS) {

void ObjectSynchronizer::fast_exit(oop object, BasicLock* lock, TRAPS) {

// -----------------------------------------------------------------------------
// Interpreter/Compiler Slow Case
// This routine is used to handle interpreter/compiler slow case
// We don't need to use fast path here, because it must have been
// failed in the interpreter/compiler code.
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {

// This routine is used to handle interpreter/compiler slow case
// We don't need to use fast path here, because it must have
// failed in the interpreter/compiler code. Simply use the heavy
// weight monitor should be ok, unless someone find otherwise.
void ObjectSynchronizer::slow_exit(oop object, BasicLock* lock, TRAPS) {
```

关键字： fast_enter、fast_exit、slow_enter、slow_exit

# 对象结构

OOP-Klass Model。

Ordinary Object Pointer （普通对象指针）表示实例对象信息




Klass 则包含元数据和方法信息，用来描述Java类，一般jvm在加载class文件时，会在方法区创建instanceKlass，表示其元数据，包括常量池、字段、方法等。

openjdk-jdk8u-jdk8u\hotspot\src\share\vm\oops\klass.hpp

Klass是在class文件在加载过程中创建的，OOP则是在Java程序运行过程中new对象时创建的

参考sun开发人员的分享：HotspotOverview.pdf、biasedlocking-oopsla2006-preso-150106.pdf

对象整体结构

MarkWord说明(在klass也有一个_prototype_header)：

源码： openjdk-jdk8u-jdk8u\hotspot\src\share\vm\oops\markOop.hpp

| bitfields | | | tag bits |
|---|---|---|---|
| hash | age | 0 | 01 |
| ptr to lock record | | | 00 |
| ptr to heavyweight monitor | | | 10 |
| | | | 11 |
| thread id | epoch | age 1 | 01 |

| state |
|---|
| unlocked |
| lightweight locked |
| inflated |
| marked for gc |
| biasable |

32位系统

| Object Header (64 bits) | | | | | State |
|---|---|---|---|---|---|
| Mark Word (32 bits) | | | | Klass Word (32 bits) | |
| identity_hashcode:25 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Normal |
| thread:23 | epoch:2 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Biased |
| ptr_to_lock_record:30 | | | lock:2 | OOP to metadata object | Lightweight Locked |
| ptr_to_heavyweight_monitor:30 | | | lock:2 | OOP to metadata object | Heavyweight Locked |
| | | | lock:2 | OOP to metadata object | Marked for GC |

64位系统

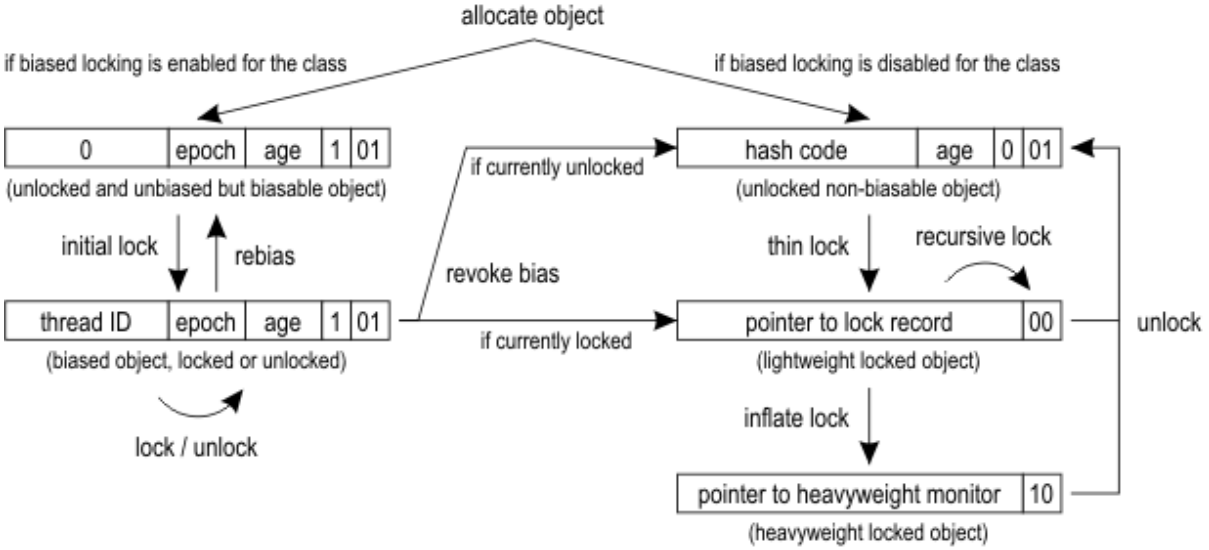
| Object Header (128 bits) | | | | | | State |
|---|---|---|---|---|---|---|
| Mark Word (64 bits) | | | | | Klass Word (64 bits) | |
| unused:25 | identity_hashcode:31 | unused:1 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Normal |
| thread:54 | epoch:2 | unused:1 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Biased |
| ptr_to_lock_record:62 | | | | | lock:2 | OOP to metadata object | Lightweight Locked |
| ptr_to_heavyweight_monitor:62 | | | | | lock:2 | OOP to metadata object | Heavyweight Locked |
| | | | | | lock:2 | OOP to metadata object | Marked for GC |

开启压缩

| Object Header (96 bits) | | | | | | | State |
|---|---|---|---|---|---|---|---|
| Mark Word (64 bits) | | | | | | Klass Word (32 bits) | |
| unused:25 | identity_hashcode:31 | cms_free:1 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Normal |
| thread:54 | epoch:2 | cms_free:1 | age:4 | biased_lock:1 | lock:2 | OOP to metadata object | Biased |
| ptr_to_lock_record | | | | | lock:2 | OOP to metadata object | Lightweight Locked |
| ptr_to_heavyweight_monitor | | | | | lock:2 | OOP to metadata object | Heavyweight Locked |
| | | | | | lock:2 | OOP to metadata object | Marked for GC |

## 锁流程说明

JVM有两种指令解析方式：字节码解释器、直接转为汇编。 逻辑大同小异，只是语法不同了。
所以很多内容是直接分析C++代码的出来的流程。
对于对象头的变化，可以通过 JOL 框架来打印对象头，实现可视化的查看。




## 偏向锁

大部分代码写完，都没啥线程竞争出现，于是偏向锁出来了。
概念不难，难点在：epoch – 重偏向和偏向取消。 （**后面这段文字，仅做思路记录~具体的看代码吧**）
判断这个对象是否获得偏向锁的条件就是：mark字段后3位是101，thread字段跟当前线程相同，epoch字段跟所属类的epoch值相同。
Epoch 即在对象头，class信息中也有一个epoch记录
jvm以类为单位，内部为类提供一个偏向锁计数器，对该类锁创建的对象进行偏向锁的撤销次数计数(撤销其实就是出现争用，变成轻量锁)。当这个撤销次数值达到一定阈值后，jvm认为这个类的偏向锁有问题，需要进行重偏向（rebias）。对该类的对象进行重偏向的操作叫批量重偏向（bulk rebias），类记录的epoch值加1，后续该类创建的对象mark字段epoch值为1，同时对当前已经获得偏向锁的对象的epoch值加1（JVM通过线程栈可以找到这些对象）。
如果epoch值不一样，即使thread字段指向当前线程，也是无效的，相当于进行过了rebias，只是没有对对象的mark字段进行更新。

如果这个类的revoke计数器继续增加到一个阈值，那个jvm就认为这个类不适合偏向锁了，就要进行bulk revoke。于是多了一个判断条件，要查看所属类的字段，看看是否允许对这个类使用偏向锁。
批量重偏向：在jvm里面记录 – 同一个类创建的对象，如果第二次有线程来抢锁，并且取消偏向超过20次，则批量重偏向—修改偏向锁的指向。
批量取消偏向：在jvm里面记录 – 如果第二次有线程来抢锁，并且取消偏向达到40次以上，则该类新创建的对象统一不再进入偏向锁。

-XX:+UnlockDiagnosticVMOptions -XX:+PrintBiasedLockingStatistics
偏向锁 – 批量重偏向 – 批量取消偏向 – （判断方式 比较线程ID）
轻量锁 – CAS争用
https://www.zhihu.com/question/311877779
http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

### 轻量级锁

对象头转到轻量级锁的结构，CAS修改lock标志
废弃参数 ---
-XX:+UseSpinning 是否开启自旋 默认开启
-XX:PreBlockSpin 自旋多少次升级为重量级

### 重量锁

复用或者创建一个新的，对象监视器（内部有锁的标志、等待抢锁的队列、wait队列..等等定义）
此处涉及JIT对于锁的两个优化概念：锁粗化、锁消除 (-XX:+EliminateLocks)

# JVM实践相关

### 启动参数

内存参数 最大值 最小值
初始化分配，预留给操作系统的空间大小
参数:-XX:+AlwaysPreTouch
描述: JAVA进程启动的时候,虽然我们可以为JVM指定合适的内存大小,但是这些内存操作系统并没有真正的分配给JVM,而是等JVM访问这些内存的时候,才真正分配；通过配置这个参数JVM就会先访问所有分配给它的内存,让操作系统把内存真正的分配给JVM.从而提高运行时的性能，后续JVM就可以更好的访问内存了；

### FullGC危害

防止System.gc()导致频繁GC
Stop The World次数变多
生产环境 web应用 用小堆小内存，响应时间优先，避免GC时间过长。

## 堆内存溢出

1. OOM问题分析
2. 拿内存快照
3. Eclipse mat 工具分析
4. 线上jvm状态统计

一定要结合应用本身的情况来选择，没有最优的参数，只有最适合的参数； 可以通过应用监控或者jstat、jmap等工具命令观察应用启动、运行时情况来配置，重点关注以下几点：
年轻代、年老代、MetaSpace占用和增长情况；
youngGC的频率和时间
fullGC的频率和时间
GC前后垃圾释放的程度

5. GC日志打印

## 堆外内存溢出

-XX:MaxDirectMemorySize=10M控制堆外内存的大小

不同JVM实现，默认大小不同。自己用一定要显式配置这个参数

## 效果演示

参考示例代码

## JVM自带分析机制

**NMT - Native Memory Tracking (NMT)** 是Hotspot VM用来分析VM内部内存使用情况的一个功能

NMT必须先通过VM启动参数中打开，不过要注意的是，打开NMT会带来性能损耗。

-XX:NativeMemoryTracking=[off | summary | detail]

# off: 默认关闭

# summary: 只统计各个分类的内存使用情况.

# detail: Collect memory usage by individual call sites.

**VM退出时打印NMT**

可以通过下面VM参数在JVM退出时打印NMT报告。

-XX:+UnlockDiagnosticVMOptions -XX:+PrintNMTStatistics

**jcmd查看NMT报告**

通过jcmd查看NMT报告以及查看对比情况。

jcmd <pid> VM.native_memory [summary | detail | baseline | summary.diff | detail.diff | shutdown] [scale= KB | MB | GB]

# summary: 分类内存使用情况.

# detail: 详细内存使用情况，除了summary信息之外还包含了虚拟内存使用情况。

# baseline: 创建内存使用快照，方便和后面做对比

# summary.diff: 和上一次baseline的summary对比

# detail.diff: 和上一次baseline的detail对比
# shutdown: 关闭NMT

Java Heap部分表示heap内存目前占用了463MB；
Class部分表示已经加载的classes个数为8801，其metadata占用了50MB；
Thread部分表示目前有225个线程，占用了27MB；
Code部分表示JIT生成的或者缓存的instructions占用了17MB；
GC部分表示目前已经占用了15MB的内存空间用于帮助GC；
Compiler部分表示compiler生成code的时候占用了26MB；
Internal部分表示命令行解析、JVM JIT等占用了5MB；
Other部分表示尚未归类的占用了2MB；
Symbol部分表示诸如string table及constant pool等symbol占用了10MB；
Native Memory Tracking部分表示该功能自身占用了5MB；
Arena Chunk部分表示arena chunk占用了63MB,一个arena表示使用malloc分配的一个memory chunk，这些chunks可以被其他subsystems做为临时内存使用，比如pre-thread的内存分配，它的内存释放是成bulk的

## 第三方分析工具

Gperftools - 谷歌的内存申请器替代操作系统的，实现统计

安装这个鬼

```
# 下载libunwind依赖
wget http://ftp.twaren.net/Unix/NonGNU/libunwind/libunwind-1.1.tar.gz
tar -xvf libunwind-1.1.tar.gz
cd libunwind-1.1
./configure --prefix=/u01/jvm-study/tools/libunwind/ CFLAGS=-U_FORTRIFY_SOURCE
make
make install
# 下载 https://github.com/gperftools/gperftools/releases
wget https://github.com/gperftools/gperftools/releases/download/gperftools-2.7/gperftools-2.7.tar.gz
tar -xvf gperftools-2.7.tar.gz
cd gperftools-2.7
./configure --prefix=/u01/jvm-study/tools/gperftools LDFLAGS=-L/u01/jvm-study/tools/libunwind/lib CPPFLAGS=-L/u01/jvm-study/tools/libunwind/include
make
make install
```

使用

**1. 临时修改内存申请器 - 设定内存申请器**

export LD_PRELOAD= /u01/jvm-study/tools/gperftools/lib/libtcmalloc.so

**2. 指定内存分析结果存放路径**

mkdir -p  /u01/jvm-study/tmp/gperftool-heap

export HEAPPROFILE= /u01/jvm-study/tmp/gperftool-heap

**3. 结果分析**

/u01/jvm-study/tools/gperftools/bin/pprof --text $JAVA_HOME/bin/java /u01/jvm-study/tmp/gperftool-heap /具体文件名 > report.txt

## 代码调用分析

# btrace分析代码调用链 – 看看哪里在调用这个鬼代码

1. 安装

```
wget https://github.com/btraceio/btrace/releases/download/v1.3.11.3/btrace-bin-1.3.11.3.tgz
mkdir -p  /u01/jvm-study/tools/btrace-bin-1.3.11.3
mv btrace-bin-1.3.11.3.tgz  /u01/jvm-study/tools/btrace-bin-1.3.11.3/btrace-bin-1.3.11.3.tgz
cd   /u01/jvm-study/tools/btrace-bin-1.3.11.3
tar -xvf btrace-bin-1.3.11.3.tgz
rm -rf btrace-bin-1.3.11.3.tgz
```

2． 使用 – 可以理解为动态的向JVM插入脚本执行

./bin/btrace -cp /u01/jvm-study/tools/btrace-bin-1.3.11.3/build/ 6358 /u01/jvm-study/tools/btrace-bin-1.3.11.3/TracingSystemGCScript.java

或者 jvisualVM插件Btrace查看效果

## 请求响应缓慢

## 死锁

## “活锁”

## CPU高占用

## 线程数量控制

## CPU缓存行