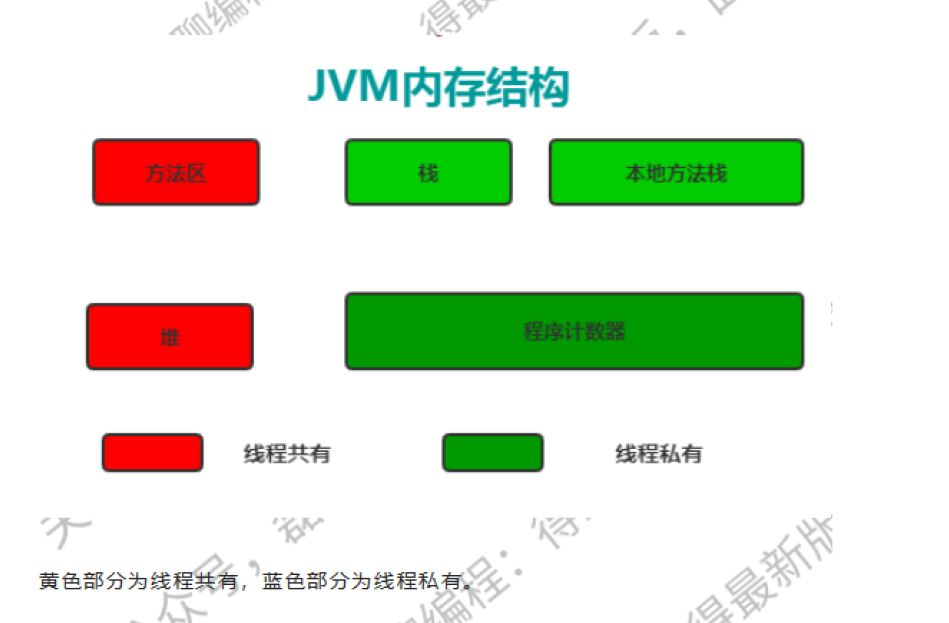
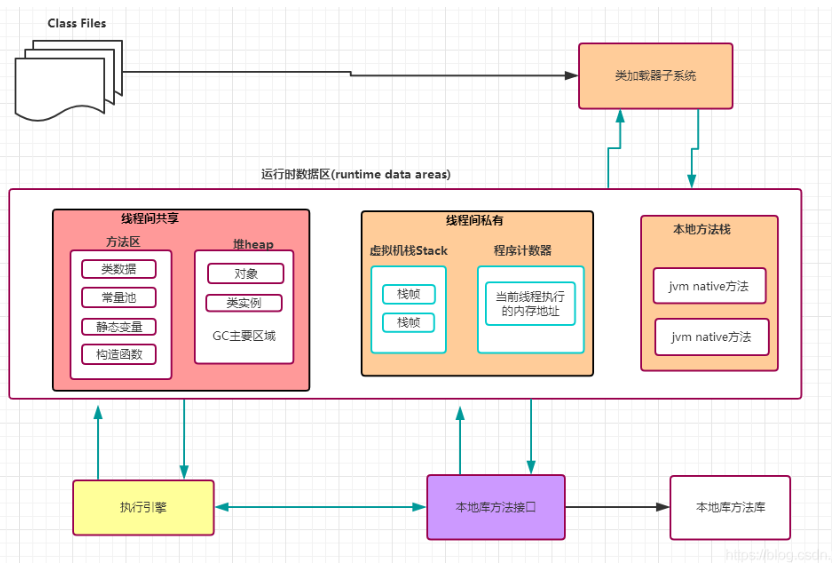
# JVM内存模型

根据 JVM 规范，JVM 内存共分为虚拟机栈、本地方法栈、堆、方法区、程序计数器五个部分。





## 方法区

用于存储虚拟机加载的类信息，常量，静态变量等数据。

### 运行时常量池

运行时常量池是方法区的一部分，用于存放编译器生成：

（1）字面量

（2）符号引用

这部分内容将在类加载后，进入方法区的运行时常量池存放。除了保存Class文件中描述的**符号引用**外，还会把翻译出来的**直接引用**也存储在运行时常量池中。运行期间也可能将新的常量放入池中，比如String类的intern( )方法。

## 堆

存放对象实例，所有的对象和数组都要在堆上分配，是 JVM 所管理的内存中最大的一块区域。

## 栈（虚拟机栈）

Java 方法执行的内存模型: 存储局部变量表，操作数栈，动态链接，方法出口等信息，生命周期与线程相同。

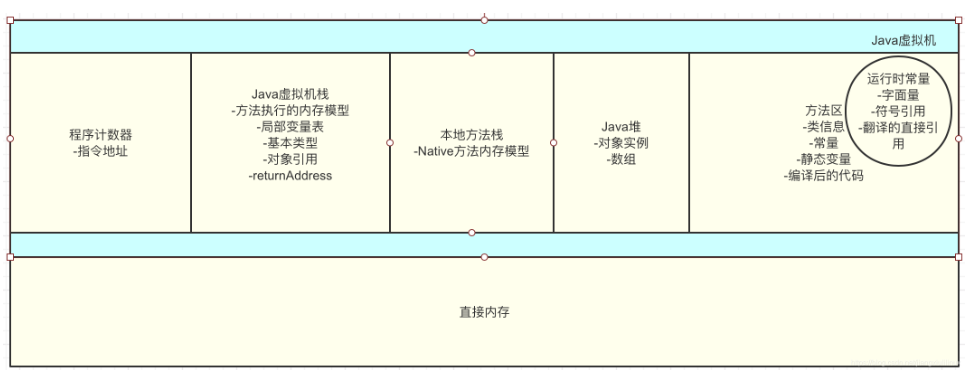
## 本地方法栈

作用与虚拟机栈类似，不同点是本地方法栈为native 方法执行服务的，虚拟机栈为虚拟机执行的 Java 方法服务。

## 程序计数器

当前线程所执行的行号指示器，是 JVM 内存区域最小的一块区域。执行字节码工作时就是利用程序计数器来选取下一条需要执行的字节码指令。

## 总结



# 各内存区域可能发生的异常情况



# Java内存分配

## 1、寄存器

我们无法控制

## 2、静态域

static定义的静态成员。

## 3、常量池

文件中的 (final) 常量值和一些文本修饰常量池: 编译时被确定并保存在 .class.的符号引用(类和接口的全限定名，字段的名称和描述符，方法的名称和描述符)。

## 4、非RAM存储

硬盘等永久存储空间。

## 5、堆内存

new 创建的对象和数组，由 Java 虚拟机自动垃圾回收器管理，存取速度慢。

## 6、栈内存

基本类型的变量和对象的引用变量（堆内存空间的访问地址)，速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

栈由系统自动分配，速度较快，但程序员是无法控制的。堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

# 永久代和元空间

永久代和元空间都是方法区的落地实现，jdk1.8将永久代（PermGen）替换为元空间（Metaspace）。

（1）原本永久代存储的数据: 符号引用(Symbols)转移到了native heap; 字面量(interned strings)转移到了java heap; 类的静态变量(class statics)转移到了java heap。

（2）Metaspace (元空间) 存储的是类的元数据信息 (metadata)

（3）元空间的本质和永久代类似，都是对JVM 规范中方法区的实现。不过元空间与永久代之间最大的区别在于: 元空间并不在虚拟机中，而是使用本地内存。

（4）替换的好处:

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、永久代会为GC带来不必要的复杂度，并且回收效率偏低。

# Java堆

## 基本概念

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM启动的时候被创建。对象所占的堆内存是由自动内存管理系统，也就是垃圾收集器回收。堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。

## 特点

1、堆内存的存储特点：先进先出，后进后出。

2、堆是JVM占用区域最大的一块，并且在运行时动态地分配内存大小。

3、线程共享，整个 Java 虚拟机运行过程中只会有一个堆，所有的线程都访问同一个堆。而JVM中的程序计数器、Java 虚拟机栈、本地方法栈都是一个线程对应一个。

4、虚拟机启动的时候创建堆。

5、堆是JVM中涉及垃圾回收的主要场所。

## 组成结构

在 JVM中，堆被划分成两个不同的区域：新生代 ( Young )、老年代 ( Old )。新生代 ( Young ) 又被划分为三个区域：Eden区、From Survivor区、To Survivor区。这样划分的目的是为了使JVM能够更好的管理堆内存中的对象，包括内存的分配以及回收。



# Java内存泄漏

所谓内存泄露是指一个不再被程序使用的对象或变量一直被占据在内存中。Java 中有垃圾回收机制，它可以保证一个对象不再被引用的时候，即对象变成了孤儿的时候，对象将自动被垃圾回收器从内存中清除掉。由于 Java 使用有向图的方式进行垃圾回收管理，可以消除引用循环的问题，例如有两个对象，相互引用只要它们和根进程不可达的，那么 GC 也是可以回收它们的。

Java 中的内存泄露的情况: 长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 Java 中内存泄露的发生场景，通俗地说就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收，这就是 java 中可能出现内存泄露的情况，例如缓存系统，我们加载了一个对象放在缓存中 (例如放在一个全局 map 对象中)，然后一直不再使用它，这个对象一直被缓存引用但却不再被使用。

检查 Java 中的内存泄露，一定要让程序将各种分支情况都完整执行到程序结束，然后看某个对象是否被使用过，如果没有，则才能判定这个对象属于内存泄露。如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

# Java内存溢出

指程序在申请内存时，没有足够的内存空间供其使用，出现OOM异常。比如我们的JVM内存还剩10MB，但是创建了一个20MB的对象，那么此时JVM已经没有足够的空间可以盛放这个对象，此时就会发生内存溢出的问题。

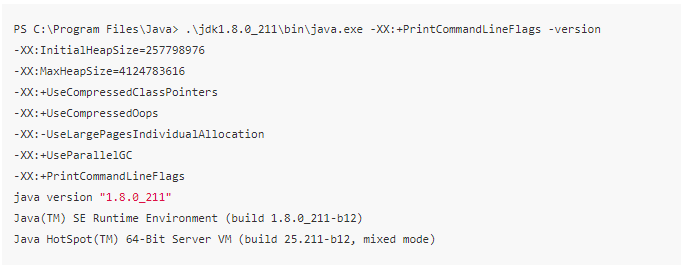
# GC是什么

GC 是垃圾收集的意思 (GabageCollection)，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

# Java各版本垃圾收集器

## JDK8

ParallelGC



## JDK9-JDK17

默认都是G1

# JVM参数

## X参数

这是非标准化参数。表示在将来的JVM版本中可能会发生改变，但是这类以 -X开始的参数变化的比较小。我们可以通过 Java -X 命令来检索所有-X 参数。



## XX参数

这是我们日常开发中接触到最多的参数类型。这也是非标准化参数，相对来说不稳定，随着JVM版本的变化可能会发生变化，主要用于**JVM调优**和debug。

**注意**：这种参数是我们后续介绍JVM调优讲解最多的参数。该参数的书写形式又分为两大类：

### 1、Boolean类型

格式：-XX:[+-]<name> 表示启用或者禁用name属性。

例子：-XX:+UseG1GC（表示启用G1垃圾收集器）

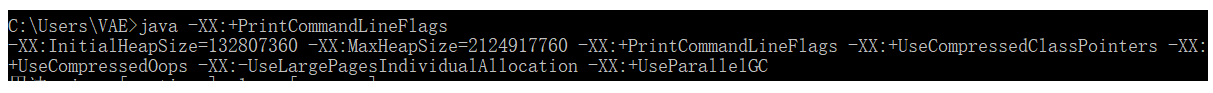
### 2、Key-Value类型

格式：-XX:<name>=<value> 表示name的属性值为value。

例子：-XX:MaxGCPauseMillis=500（表示设置GC的最大停顿时间是500ms）

## ****打印已经被用户或者当前虚拟机设置过的参数****

-XX:+PrintCommandLineFlags



## ****最大堆和最小堆内存设置****

-Xms512M：设置堆内存初始值为512M

-Xmx1024M：设置堆内存最大值为1024M

这里的ms是memory start的简称，mx是memory max的简称，分别代表最小堆容量和最大堆容量。但是别看这里是-X参数，其实这是-XX参数，等价于：

-XX:InitialHeapSize

-XX:MaxHeapSize

在通常情况下，服务器项目在运行过程中，堆空间会不断的收缩与扩张，势必会造成不必要的系统压力。所以在生产环境中，**JVM的Xms和Xmx要设置成一样的**，能够避免GC在调整堆大小带来的不必要的压力。

## ****Dump异常快照以及以文件形式导出****

-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath

堆内存出现OOM的概率是所有内存耗尽异常中最高的，出错时的堆内信息对解决问题非常有帮助，所以给JVM设置这个参数(**-XX:+HeapDumpOnOutOfMemoryError**)，让JVM遇到OOM异常时能输出堆内信息，并通过（**-XX:+HeapDumpPath**）参数设置堆内存溢出快照输出的文件地址，这对于特别是对相隔数月才出现的OOM异常尤为重要。

这两个参数通常配套使用：

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=./

## ****发送OOM后，执行一个脚本****

-XX:OnOutOfMemoryError

比如这样设置：

-XX:OnOutOfMemoryError="C:\Program Files\Java\jdk1.8.0\_152\bin\jconsole.exe"

表示发生OOM后，运行jconsole.exe程序。这里可以不用加“”，因为jconsole.exe路径Program Files含有空格。利用这个参数，我们可以在系统OOM后，自定义一个脚本，可以用来发送邮件告警信息，可以用来重启系统等等。

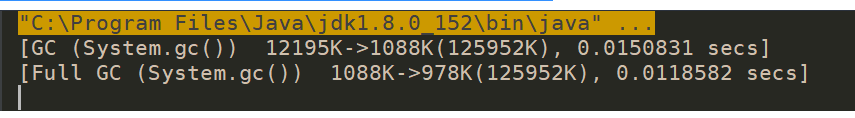
## ****打印gc信息****

### 1、**打印GC简单信息**

-verbose:gc

-XX:+PrintGC

一个是标准参数，一个是-XX参数，都是打印详细的gc信息。通常会打印如下信息：

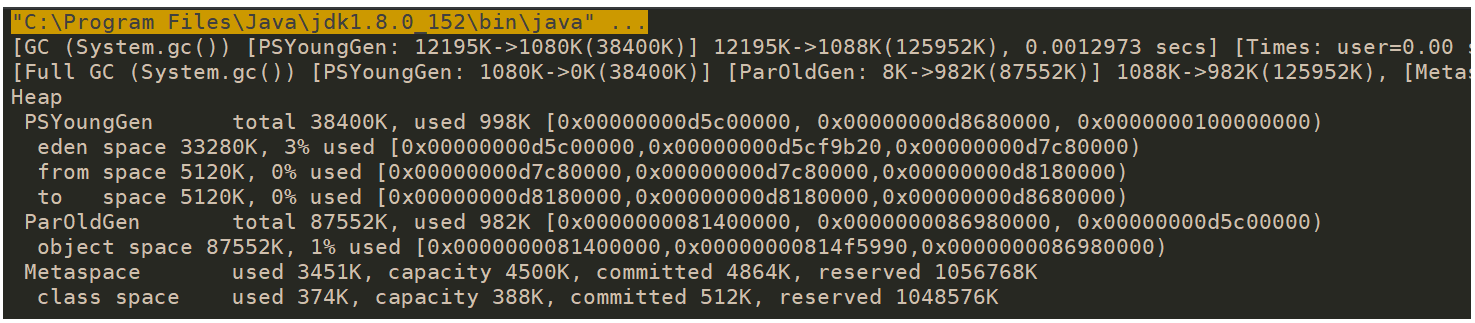


比如第一行，表示GC回收之前有12195K的内存，回收之后剩余1088K，总共内存为125951K

### 2、**打印详细GC信息**

**-XX:+PrintGCDetails**

**-XX:+PrintGCTimeStamps**



## **指定GC日志以文件输出**

**-Xloggc:./gc.log**

这个在参数用于将gc日志以文件的形式输出，更方便我们去查看日志，定位问题。

## ****设置永久代大小（jdk1.7之前）****

**-XX:MaxPermSize=1280m**

　　在JDK1.7以及以前的版本中，只有Hotspot 才有Perm区，称为永久代，它在启动时固定大小，很难进行调优。

　　在某些情况下，如果动态加载类过多，容易产生Perm区的 OOM。比如某个实际 Web 工程中，因为功能点较多，在运行过程中，要不断动态加载很多类，就会出现类似错误：

　　"Exception in thread 'dubbo client x.x.connect' java.lang.OutOfMemoryError:PermGenspace"

　　为了解决这个问题，就需要在项目启动时，设定运行参数-XX:MaxPermSize。

**注意：在JDK1.8以后面的版本，使用元空间来代替永久代。在 JDK1.8以及后面的版本中，如果设定参数-XX:MaxPermSize，启动JVM不会报错，但是会提示：**

**Java Hotspot 64Bit Server VM warning：ignoring option MaxPermSize=1280m:support was removed in 8.0**

## 常用的基本配置参数

### -Xms

初始堆分配内存，默认为物理内存的 1/64，等价于 -XX:InitialHeapSize。

### -Xmx

最大堆分配内存，默认为物理内存的 1/4，等价于 -XX:MaxHeapSize。

### -Xss

等价于-XX:ThreadStackSize，单个线程栈空间大小，默认一般为512k-1024k，通过 jinfo 查看为 0时，表示使用默认值。

### -Xmn

设置年轻代大小

### -XX:MetaspaceSize

设置元空间大小(默认 21M 左右，可以配置大一些)元空间的本质与永久代类似，都是对JM 规范中方法区的实现，不过元空间与永久代的最大区别在于: 元空间不在虚拟机中，而是使用本地内存，因此默认情况下元空间大小仅受本地内存大小限制。

典型设置案例:

-Xms128m -Xmx4096m -Xss1024k -XX:MetaspaceSize=512m

-XX:+PrintCommandLineFlags-XX:+PrintGCDetails -XX:+UseSerialGC

### -XX:+PrintGCDetails

打印垃圾回收细节，打印 GC。

### -XX:SurvivorRatio

调整 Eden 中 survivor 区比例,默认-XX:SurvivorRatio=8(8:1:1) ，调整为-XX:SurvivorRatio=4 (4:1:1),一股使用默认值

### -XX:NewRatio

调整新生代与老年代的比例，默认为 2(新生代 1，老年代年轻代占整个堆的 1/3) ,调整为-XX:NewRatio=4 表示(新生代 1，老年代 4,年轻代占堆的 1/5)，一般使用默认值。

### -XX:MaxTenuringThreshold

设置垃圾的最大年 (经历多少次垃圾回收进入老年代)，默认 15(15 次垃圾回收后依旧存活的对象进入老年代)，JDK1.8设置必须 0<-XX:MaxTenuringThreshold<15

# Java垃圾回收机制

在 Java 中，程序员是不需要显示的去释放一个对象的内存，而是由虚拟机自行执行。在 JVM 中有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时才会触发执行，扫描那些没有被任何引用的对象，并将它们添加到要回收的集合中进行回收。

# 垃圾回收的优点和原理

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 C++ 程序员最头疼的内存管理的问题迎刃而解，它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有“作用域”的概念，只有对象的引用才有"作用域”。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低级别的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

垃圾回收器可以马上回收内存吗?有什么办法主动通知虚拟机进行垃圾回收?

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常GC 采用有向图的方式记录和管理堆 (heap) 中的所有对象。通过这种方式确定哪些对象是”可达的”，哪些对象是”不可达的”。当 GC 确定一些对象为“不可达”时，GC 就有责任回收这些内存空间。程序员可以手动执行 System.gc()通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

# 如何判断一个对象是否存活

## 1、引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收。次引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象B，对象 B 又引用者对象 A，那么此时 A、B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

## 2、可达性算法 (引用链法)

该算法的思想是: 从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。

在 Java 中可以作为 GC Roots 的对象有以下几种:

1、虚拟机栈中引用的对象

2、方法区类静态属性引用的对象

3、 方法区常量池引用的对象

4、本地方法栈JNI引用的对象

# 回收机制

回收机制有分代复制垃圾回收、标记垃圾回收，增量垃圾回收。

# 在 Java 中，对象什么时候可以被垃圾回收

当对象在当前应用程序中变得不可触及的时候，这个对象就可以被回收了。

# JVM 中垃圾收集器

## 新生代垃圾收集器

### Serial收集器

特点:Serial 收集器只能使用一条线程进行垃圾收集工作，并且在进行垃圾收集的时候，所有的工作线程都需要停止工作，等待垃圾收集线程完成以后，其他线程才可以继续工作。

使用算法: 复制算法

### ParNew 收集器

特点: ParNew 垃圾收集器是 Serial 收集器的多线程版本。为了利用 CPU 多核多线程的优势，ParNew 收集器可以运行多个收集线程来进行垃圾收集工作，这样可以提高垃圾收集过程的效率。

使用算法:复制算法

### Parallel Scavenge 收集器

特点：Parallel Scavenge 收集器是一款多线程的垃圾收集器，但是它又和ParNew 有很大的不同点。Parallel Scavenge 收集器和其他收集器的关注点不同。其他收集器，比如ParNew 和 CMS 这些收集器，它们主要关注的是如何缩短垃圾收集的时间。而Parallel Scavenge 收集器关注的是如何控制系统运行的吞吐量。这里说的吞吐量指的是 CPU 用于运行应用程序的时间和 CPU 总时间的占比，吞吐量 = 代码运行时间 / (代码运行时间 + 垃圾收集时间)。如果虚拟机运行的总的 CPU 时间是 100 分钟，而用于执行垃圾收集的时间为 1 分钟，那么吞吐量就是 99%。

使用算法:复制算法

## 老年代垃圾收集器

### Serial Old 收集器

Serial Old 收集器是 Serial 收集器的老年代版本。特点:。这款收集器主要用于客户端应用程序中作为老年代的垃圾收集器，也可以作为服务端应用程序的垃圾收集器。

使用算法:标记-整理

### Parallel Old 收集器

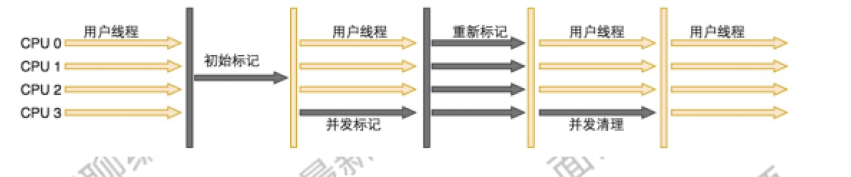
特点: Parallel Old 收集器是 Parallel Scavenge 收集器的老年代版本这个收集器是在 JDK1.6 版本中出现的，所以在 JDK1.6 之前，新生代的 ParallelScavenge 只能和 Serial Old 这款单线程的老年代收集器配合使用。ParallelOld 垃圾收集器和 Parallel Scavenge 收集器一样，也是一款关注吞吐量的垃圾收集器，和 Parallel Scavenge 收集器一起配合，可以实现对 Java 堆内存的吞吐量优先的垃圾收集策略。

使用算法: 标记-整理

### CMS 收集器

特点:CMS 收集器是目前老年代收集器中比较优秀的垃圾收集器。CMS 是Concurrent Mark Sweep，从名字可以看出，这是一款使用”标记-清除”算法的并发收集器。

CMS 垃圾收集器是一款以获取最短停顿时间为目标的收集器。如下图所示:



从图中可以看出，CMS 收集器的工作过程可以分为 4 个阶段：

1、初始标记(CMS initial mark) 阶段

2、并发标记 (CMS concurrent mark) 阶段

3、重新标记 (CMS remark) 阶段

4、 并发清除( (CMS concurrent sweep) 阶段

使用算法: 复制+标记清除

## 其他

### G1垃圾收集器

G1 收集器，G1(Garbage-First)是一款面向服务器的垃圾收集器，主要针对配备多颗处理器及大容量内存的机器、以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征。

特点：主要步骤: 初始标记，并发标记，重新标记，复制清除。

使用算法:复制 + 标记整理

# 什么情况下会出现内存溢出、内存泄漏

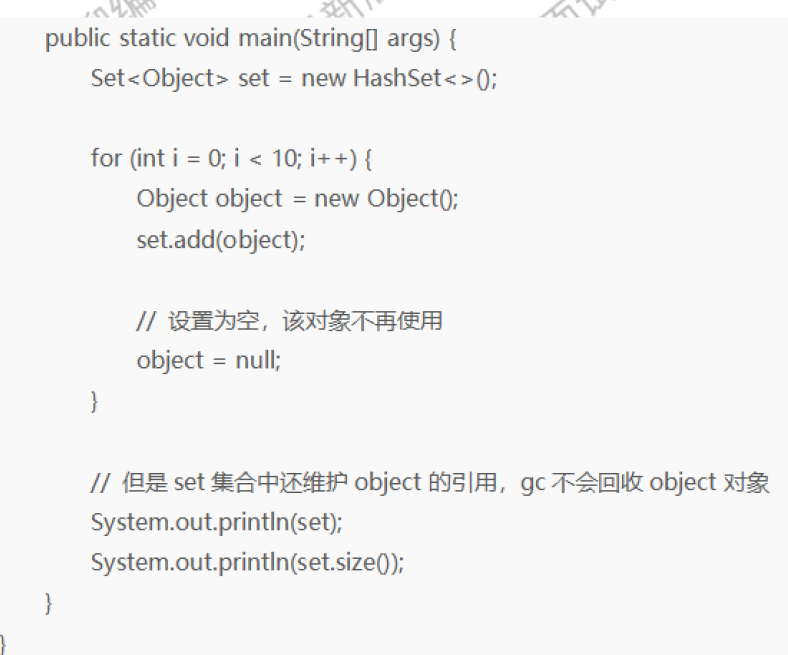
## 基本说明

内存泄漏的原因很简单：

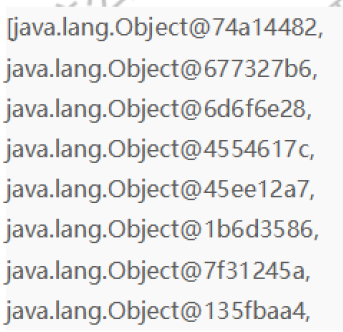
1、对象是可达的，一直被引用。

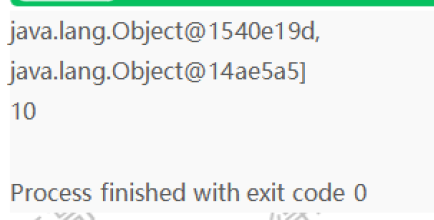
2、但是对象不会被使用。

## 代码示例



输出结果：





## 解决方法

将 set 设置为 null，那就可以避免上述内存泄漏问题了。

# 内存溢出的原因

## 基本说明

1、内存泄露导致堆栈内存不断增大，从而引发内存溢出。

2、大量的jar，class 文件加载，装载类的空间不够，溢出。

3、操作大量的对象导致堆内存空间已经用满了，溢出。

4、nio 直接操作内存，内存过大导致溢出

## 解决方法

1、查看程序是否存在内存泄漏的问题

2、设置参数加大空间。

3、代码中是否存在死循环或循环产生过多重复的对象实体.

4、查看是否使用了 nio 直接操作内存

# 线程栈

这里的线程栈指的是虚拟机栈。

1、JVM 规范让每个 Java 线程拥有自己的独立的 JVM ，也就是 Java 方法的调用栈。

2、当方法调用的时候，会生成一个栈帧。栈帧是保存在虚拟机栈中的，栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。

3、线程运行过程中，只有一个栈是处于活跃状态，称为“当前活跃栈帧”，当前活动栈帧始终是虚拟机栈的栈顶元素。

4、通过 jstack 工具查看线程状态。

# Full GC

清理整个堆内存空间—包括年轻代和永久代。

# Minor GC

年轻代是所有新对象产生的地方，当年轻代内存空间被用完时，就会触发垃圾回收，这个垃圾回收叫做Minor GC。

# Major GC

老年代的垃圾收集叫做Major GC，Major GC通常是跟full GC是等价的，收集整个GC堆。

# JVM出现 full GC 很频繁，怎么去线上排查问题

1、如果有 perm gen 的话(jdk1.8 就没了)，要给 perm gen 分配空间，但没有足够的空间时，会触发 full gc，所以看看是不是perm gen区的值设置的太小了。

2、System.gc()方法的调用。

3、当统计得到的 Minor GC 晋升到旧生代的平均大小大于老年代的剩余空间，则会触发 full gc(这就可以从多个角度上看了)

4、是不是频繁创建了大对象，也有可能 eden 区设置过小，大对象直接分配在老年代中，导致老年代空间不足--->从而频繁 gc。

5、是不是老年代的空间设置过小了，Minor GC 几个对象就大于老年代的剩余空间了。

# 请解释 StackOverflowError 和OutofMemeryError 的区别

通过之前的分析可以发现，实际上每一块内存中都会存在有一部分的可变伸缩区其基本流程为: 如果空间内存不足，在可变范围之内扩大内存空间，当一段时间之后发现内存充足，会缩小内存空间。

永久代 (JDK 1.8 后消失了)

虽然java 的版本是JDK1.8，但是 java EE 的版本还是 jdk1.7，永久代存在于堆内存之中。

元空间

元空间在 Jdk1.8 之后才有的，器功能实际上和永久代没区别，唯一的区别在于永久代使用的是JVM 的堆内存空间，元空间使用的是物理内存，所以元空间的大小受本地内存影响，一般默认在 2M 左右。

示例：设置一些参数，让元空间出错

Java -XX:MetaspaceSize=1m

# GC回收流程

1、对于整个的 GC 流程里面，那么最需要处理的就是新生代和老年代的内存清理操作，而元空间 (永久代) 都不在 GC 范围内。

2、当现在有一个新的对象产生，那么对象一定需要内存空间，平均每个栈内存存 4k，每个堆内存存 8k，那么对象一定需要进行堆空间的申请。

3、首先会判断 Eden 区是否有内存空间，如果此时有内存空间，则直接将新对象保存在伊甸园区。

4、但是如果此时在伊甸园区内存不足，那么会自动执行一个 Minor GC 操作,将伊甸园区的无用内存空间进行清理，Minor GC 的清理范围只在 Eden 园区，清理之后会继续判断 Eden 园区的内存空间是否充足? 如果内存空间充足，则将新对象直接在 Eden 园区进行空间分配。

5、如果执行 Minor GC 之后发现伊甸园区的内存空间依然不足，那么这个时候会执行存活区的判断，如果存活区有剩余空间，则将 Eden 园区部分活跃对象保存在存活区，那么随后继续判断 Eden 园区的内存空间是否充足，如果充足怎则将新对象直接在 Eden 园区进行空间分配。

6、此时如果存活区没有内存空间，则继续判断老年区。则将部分存活对象保存在老年代，而后存活区将有空余空间。

7、如果这个时候老年代也满了，那么这个时候将产生 Major GC (Full GC),那

么这个时候将进行老年代的清理。

8、如果老年代执行 Full GC 之后，无法进行对象的保存，则会产生 0OM 异常、OutOfMemoryError 异常。