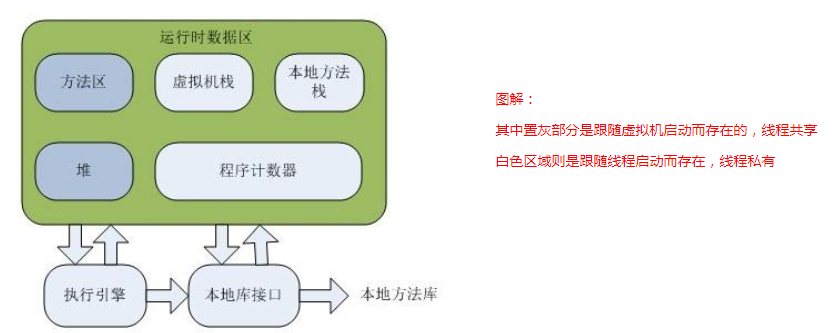
JVM

JVM是运行在操作系统之上的，与硬件没有直接的交互。

JVM体系结构：

1. Class files
2. 类装载器
3. 运行时数据区
4. 执行引擎
5. 本地方法接口

## JVM运行时数据区



### 1.程序计数器

　　占据一块较小的内存空间，可以看做当前线程所执行的字节码的行号指示器。在虚拟机概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖这个计数器来完成。

　　由于jvm的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此未来线程切换后能恢复到正确的执行位置，**每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们成这类内存区域为“线程私有”的内存。**

　　如果线程正在执行的是一个Java方法，这个计数器记录的则是正在执行的虚拟机字节码指令的地址；

　　如果正在执行的是Native方法，这个计数器则为空（undefined）。

**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。**

### 2.Java虚拟机栈

**线程私有**，生命周期和线程相同，虚拟机栈描述的是Java方法执行的内存模型，**每个方法在执行的同时都会创建一个栈帧  用于存储局部变量表，操作数栈，动态链接，方法出口等信息。每一个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。**

　　局部变量表存放了编译期可知的各种**基本类型数据**（boolean、byte、char、short、int、float、long、double）、**对象引用**、returnAddress类型（指向了一条字节码指令的地址）。

　　其中64位长度的long和double类型的数据会占用2个局部变量表空间（slot），其余的数据类型只占用1个。局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法所需要在栈帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对此区域规定了两种异常状况：**如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出Stack OverflowError异常；如果虚拟机栈可以动态扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。**

### 3.本地方法栈

　　本地方法栈与虚拟机栈所发挥的作用非常相似，他们之间的区别不过是**虚拟机栈为虚拟机执行Java方法**（字节码）服务，**而本地方法栈则为虚拟机中使用到的native方法服务**。在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机直接把本地方法栈和虚拟机栈合二为一，与虚拟机栈一样也会抛出Stack OverflowError异常和OutOfMemoryError异常。

### 4.Java堆

 　　对于大多数应用来说，**堆空间是jvm内存中最大的一块。**Java堆是被**所有线程共享**，虚拟机启动时创建，此内存区域唯一的目的就是**存放对象实例**，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展和逃逸分析技术逐渐成熟，栈上分配，标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也就变得不那么绝对了。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。从内存回收角度看，由于现在收集器基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区。不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好的回收内存，或者更快的分配内存。（**如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常**。）

### 5.方法区（也有人叫永久代）

和堆一样所有线程共享，主要用于存储已被jvm加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

　　（在JDK1.7发布的HotSpot中，已经把字符串常量池移除方法区了。）

### 6.常量池

　　运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

　　Java虚拟机对class文件每一部分的格式都有严格规定，每一个字节用于存储哪种数据都必须符合规范才会被jvm认可。但对于运行时常量池，Java虚拟机规范没做任何细节要求。

　　运行时常量池有个重要特性是动态性，Java语言不要求常量一定只在编译期才能产生，也就是并非预置入class文件中常量池的内容才能进入方法区的运行时常量池，运行期间也有可能将新的常量放入池中，这种特性使用最多的是String类的intern()方法。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制。当常量池无法再申请到内存时会抛出outOfMemeryError异常。

## 对象的创建

### 2.1 当虚拟机遇到一条New指令时：会进行如下步骤

1、检查指令的参数（即工作中我们New的对象），能否在常量池中找到它的符号引用。

2、如果存在，检查符号引用代表的类是否被加载、解析、初始化过。（如果没有则执行类的加载-----相关加载过程参考《[Jvm类的加载机制](https://www.cnblogs.com/chenpt/p/9777367.html" \t "https://www.cnblogs.com/chenpt/p/_blank)》）。

3、加载通过后，虚拟机将为新生对象分配内存。（所需内存大小在类加载完成后便可确定）

### 2.2 两种内存分配的方式：

**指针碰撞**：假设Java堆中的内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边。中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针往空闲空间那边挪动一段与对象大小相等的距离。这种方式则属于指针碰撞。

**空闲列表**：如果堆中的内存并不是规整的，已使用的内存和空闲内存相互交错，显然无法使用指针碰撞。虚拟机就必须维护一个列表，记录哪些内存是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新记录表上的数据。这种方式属于空闲列表。

具体选择哪种分配方式由Java堆决定，而Java堆是否规整，则有GC收集器决定。因此使用Serial、ParNew等带Compact过程的收集器时，系统采用的分配算法是指针碰撞。而使用CMS这种基于Mark-Sweep算法的收集器时，通常采用的空闲列表。

### 2.3如何保证分配内存时线程的安全性？

1、对分配内存的动作进行同步处理（实际上虚拟机采用CAS配上失败重试的机制保证了更新操作的原子性）

2、把分配内存的动作按照线程划分在不同的空间之中进行（即每个线程在Java堆中预先分配一小块内存（称为本地线程分配缓冲））。

**类加载器**

类装载器：负责加载class文件。

第一类是虚拟机自带的类加载器，有三种：

1：启动类加载器（Bootstrap）C++

2：扩展类加载器（Extension）Java

3：应用程序类加载器（App）Java ，也叫系统类加载器，加载当前应用的classpath的所有类

第二类是用户自定义加载器，Java.lang.ClassLoader的子类，用户可以定制类的加载方式

**类加载器的双亲委派机制**

**沙箱机制（防止恶意代码对java的破坏）**

比如自己写了一个java.lang.String

主要是为了保证java的沙箱机制

双亲委派:某个特定的类加载器在接到加载类请求的时候，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成加载任务，就返回成功，只有父类加载器无法完成加载任务的时候，才自己去加载。

**执行引擎**

负责解释命令，提交操作系统执行。

**Native Interface本地方法接口**

Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI接口调用其他语言来实现对操作系统的访问。本地接口的作用是融合不同编程语言为Java所用，它的初衷是融合C/C++程序，Java在内存中专门开辟了一块区域处理标记为Native的代码，它的具体做法是Native Method Stack中等级Native方法，在Execution Engine执行的时候加载Native libraries（JNI）

目前该方法使用的越来越少，除非是与硬件有关的应用，比如通过Java程序驱动打印机或者Java系统管理生产设备，在企业级应用中已经比较少见。业务现在的异构领域间通信技术很发达，比如可以使用Socker通信，也可以使用WebService等。

**Native Method Stack 本地方法栈**

他的具体做法是在Native Method Stack 中等级Native方法，在Execution Engine执行的时候加载本地方法库。

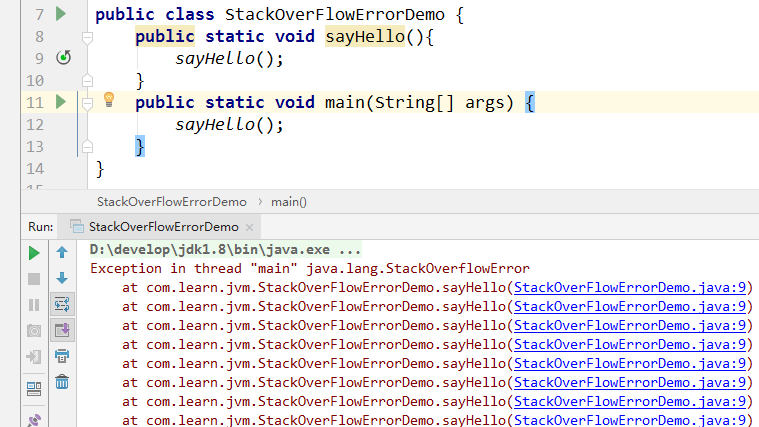
**PC寄存器**

每一个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码（用来存储指向下一条指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计。

**栈**

也叫栈内存，主管Java程序的运行，在创建线程的时候创建，它的生命周期是跟随着线程的生命周期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收的问题，只要线程一结束，该栈就Over，生命周期和线程一致，是线程私有的。基本类型中的变量、实例方法、引用类型变量都是在函数的栈内存中分配。

**StackOverFlowError**



**方法区**

1、方法区是线程共享的，通常用来保存装载的类的元结构信息

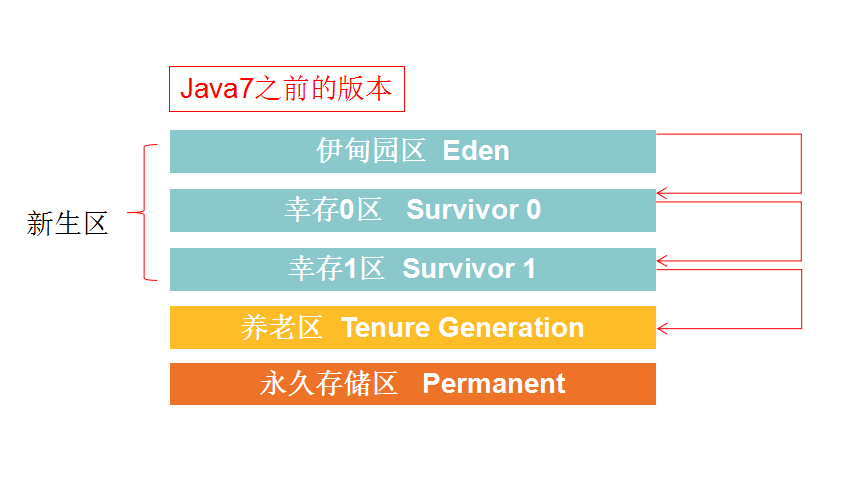
比如：运行时常量池+静态变量+常量+字段+方法字节码+在类/实例/接口 初始化用到的特殊方法。

1. 通常和永久区管理在一起（Java7之前），但是具体的跟JVM的实现和版本有关。

**Heap堆**

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件之后，需要把类、方法、常变量放到堆内存中，保存所有类型的真实信息，以方便执行器执行。

堆内存在逻辑上分为三种：新生区，养老区，永久区



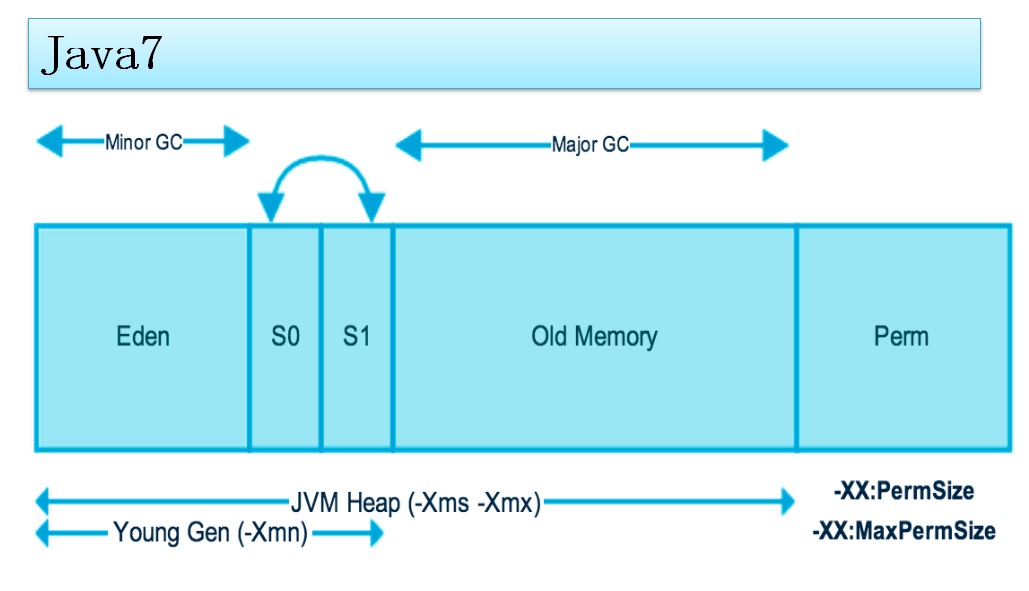
**新生区**

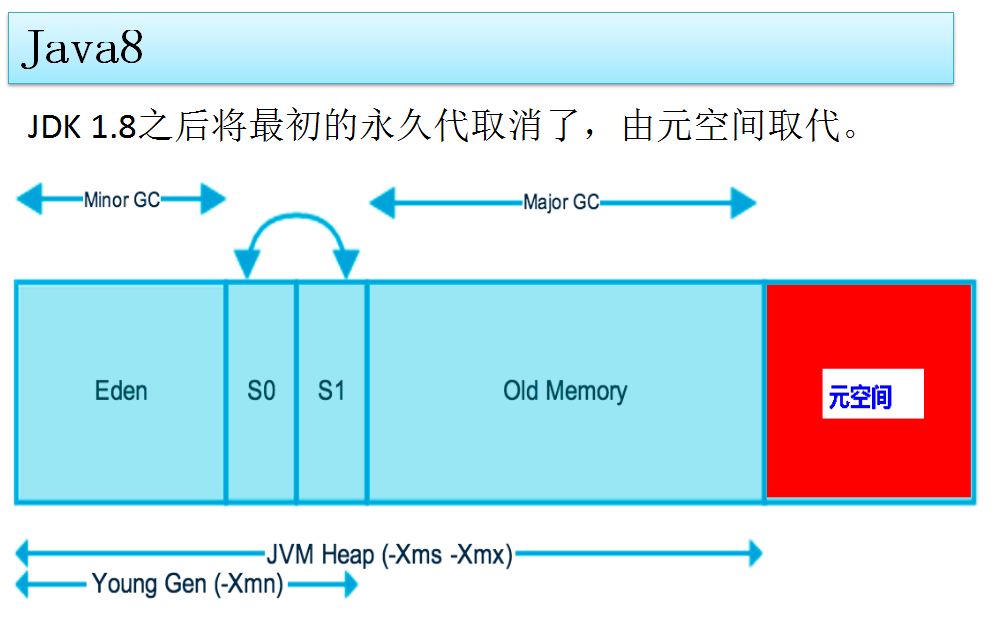
新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分： 伊甸区（Eden space）和幸存者区（Survivor pace） ，所有的类都是在伊甸区被new出来的。幸存区有两个： 0区（Survivor 0 space）和1区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC)，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存0区.若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区。那如果1区也满了呢？再移动到养老区。若养老区也满了，那么这个时候将产生MajorGC（FullGC），进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

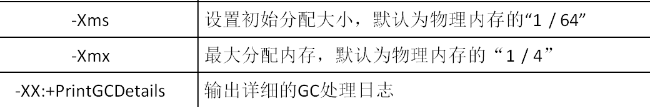
如果出现java.lang.OutOfMemoryError: Java heap space异常，说明Java虚拟机的堆内存不够。原因有：

（1）Java虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx来调整。

（2）代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）







## 常用的垃圾回收算法

一、引用计数。有对象引用加一，没有对象引用减一，到0的时候回收

缺点：

每次对对象赋值时均要维护引用计数器，并且计数器本身也有一定的消耗

比较难处理循环引用

JVM的实现一般不采用这种方式。

二、复制算法。赋值算法是在年轻代引用。复制之后有交换，谁空谁是to

1、eden,SurvivorFrom复制到SurvivorTo ，年龄+1

首先，当Eden区满的时候会触发第一次GC,把活着的对象拷贝到SurvivorFrom区域，当Eden区再次触发GC的时候会扫描Eden区和From区域，对这两个区域进行垃圾回收，经过这次回收之后还存活的对象，则直接复制到To区域（如果有对象的年龄已经达到了老年的标准，则复制到老年区域），同时把这些对象的年龄+1

2、清空Eden，SurvivorFrom

然后清空Eden和SurvivorFrom区域中的对象，也即复制之后有交换，谁空谁是To区域

1. SurvivorTo区域和SurvivorFrom区域互换

最后，SurvivorTo和SurvivorFrom互换，原来SurvivorTo成为下一次GC时的SurvivorFron区域。部分对象会在From和To区域中复制来复制去，如此交换15次（由参数MaxTenuringThreshold决定，这个参数的默认值是15），如果最终还是存活，就存入到老年代中。

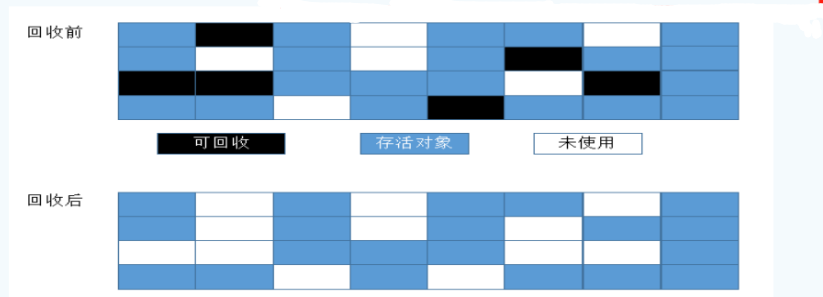
优缺点：优点，没有产出内存碎片。缺点，空间耗费。大对象复制起来耗费时间。

1. 标记清除

先标记，后清除。

优点：对象的标记，没有大面积复制，但是会产生内存碎片。

分为标记和清除两个阶段，先标记出需要回收的对象，然后统一回收这些对象，形如：

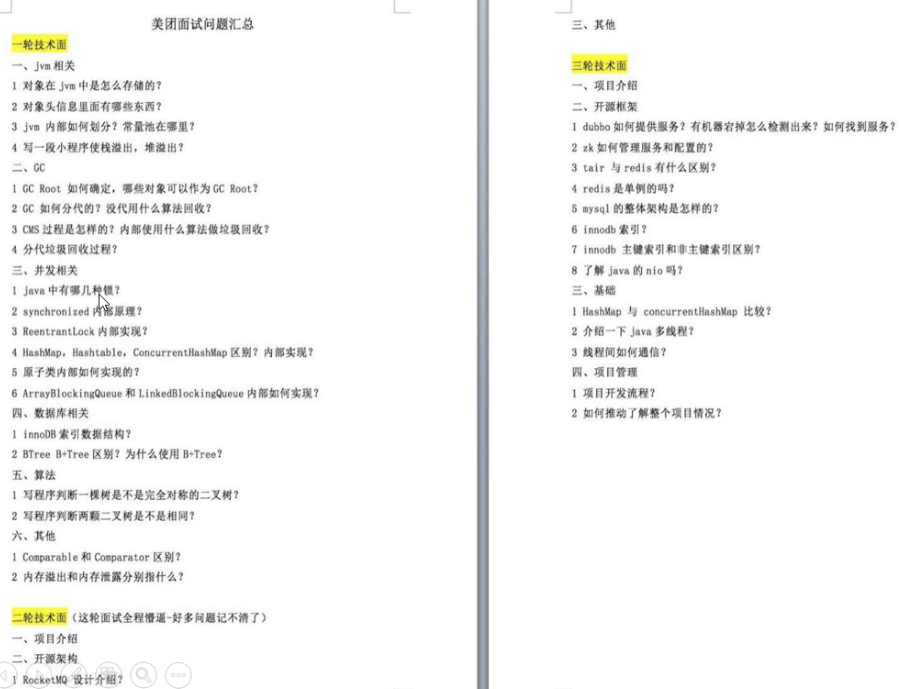


四、标记清除整理

原理：1、标记，与标记清除一样。2、压缩，再次扫描，并且往一端滑动存活对象。

缺点：移动对象需要成本。优点：没有内存碎片，可以利用bump

复制算法用于新生代，标记清除和标记清除整理用于养老区。



## 大厂面试

### 一、JVM垃圾回收的时候如何确定垃圾？是否知道什么是GC Roots?

什么是垃圾？ 内存中已经不再被使用到的空间就是垃圾。

要进行垃圾回收，如何判断一个对象是否可以被回收？

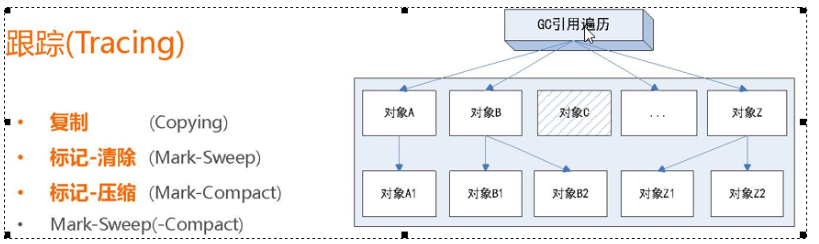
1、引用计数法：

Java中，引用和对象是有关联的。如果要操作对象，则必须用引用进行。

一个简单的办法就是通过引用计数来判断一个对象是否可以被回收，简单来说，给对象中添加一个引用计数器，每当有一个地方引用它，计数器值加1，没当有一个引用失效的时，计数器值减1。任何时刻计数器值为0的对象就是不可能再被使用，那么这个对象就是可回收对象。该算法存在但是目前无人使用，原因是无法解决循环引用的问题。

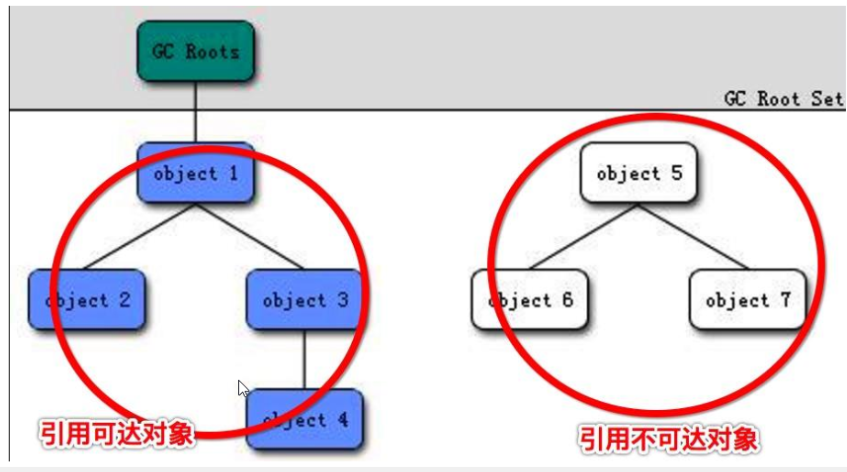
1. 枚举根节点做可达性分析（根搜索路径）

为了解决引用计数法的循环引用问题，Java使用了可达性分析办法。



所谓“GC Roots”或者说tracing GC的“跟集合” 就是**一组必须活跃的引用**。

基本思路就是通过一系列名为 “GC Roots”的对象作为起始点，从这个被称为DC 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接时，则说明此对象不可用。也即给定一个集合的引用作为跟出发，通过引用关系遍历对象图，能被遍历到的（可达到）对象就判定为存活，没有遍历到的就判定为死亡。



对象5、6、7没有从GC Root开始，所以他们会判断为引用不可达对象？

那些对象可以作为GC Root对象？总共是4种。

1、虚拟机栈（栈帧中局部变量区，也叫局部变量表）中引用的对象。

2、方法区中的类静态属性引用的对象。

3、方法区中常量引用的对象。

4、本地方法栈中JNI(Native方法)引用的对象。

Person p = new Person();

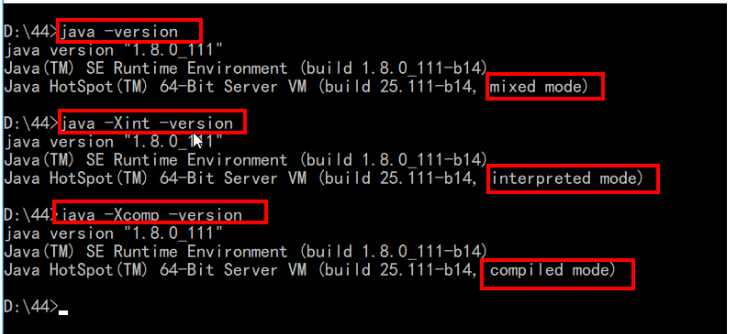
左边叫一个引用，右边称为一个实例对象。

### 二、JVM参数调优和参数配置，JVM系统默认值。

JVM的参数类型：标配参数，X参数，XX参数

标配参数：-version -help java -showversion

X参数：-Xint 解释执行 -Xcomp第一次使用就编译成本地代码 -Xmixed混合模式



XX参数：

**1、Boolean类型**

公式： -XX +或者- 某个属性值 +表示开启 -表示关闭

**如何查看一个正在运行中的Java程序，它的某个JVM参数是否开启？具体的值是多少？**



-XX: + PrintGCDeatils

-XX: +UseSerialGC

举例：是否打印GC的收集细节 jps -l 查看进程编号 jinfo -flag PrintGCDetails 1111进程

是否使用串行垃圾回收器

**2、KV设值类型**

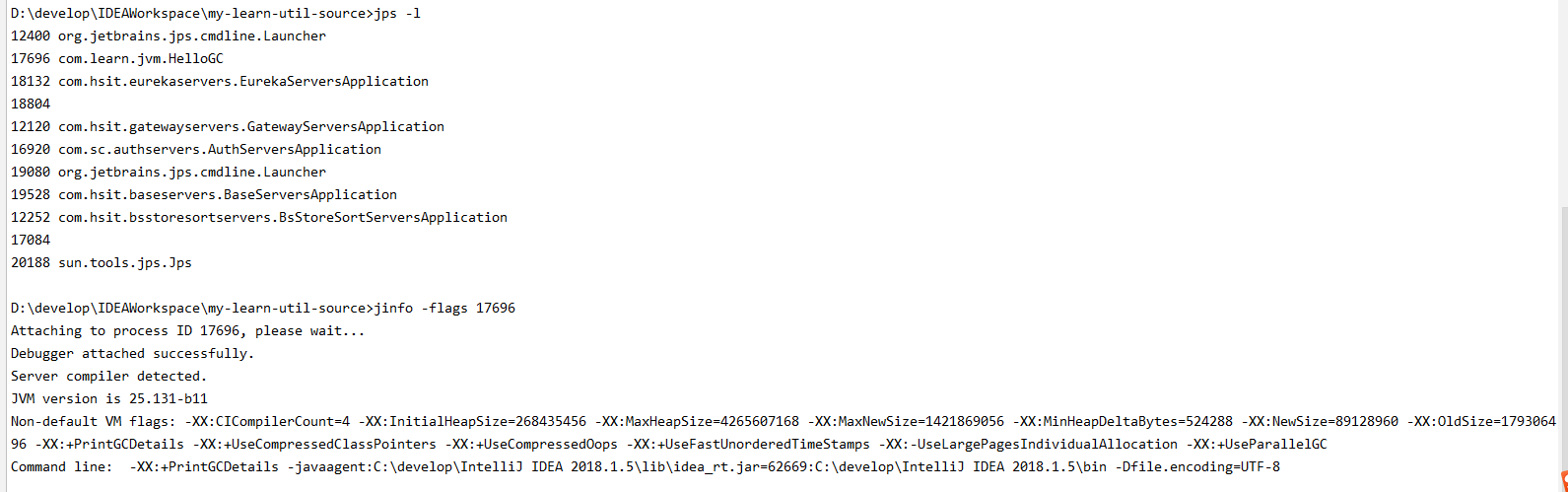
公式： -XX:属性值key = 属性值value

举例：-XX:MetaspaceSize=128m

-XX:MaxTenuringThreshould=15

**jinfo查看某个java进程所有jvm根据机器性能自带的参数**

**常见的jvm参数：<https://cloud.tencent.com/developer/article/1525732>**



Non-default VM flags :初始化加载，系统做的

Command line：人为加的

**另外两个参数 -Xms -Xmx**

-Xms等价于： -XX:InitialHeapSize

-Xmx等价于： -XX:MaxHeapSize

-Xms 默认值是系统内存的64分之1 -Xmx:默认是系统的4分之1

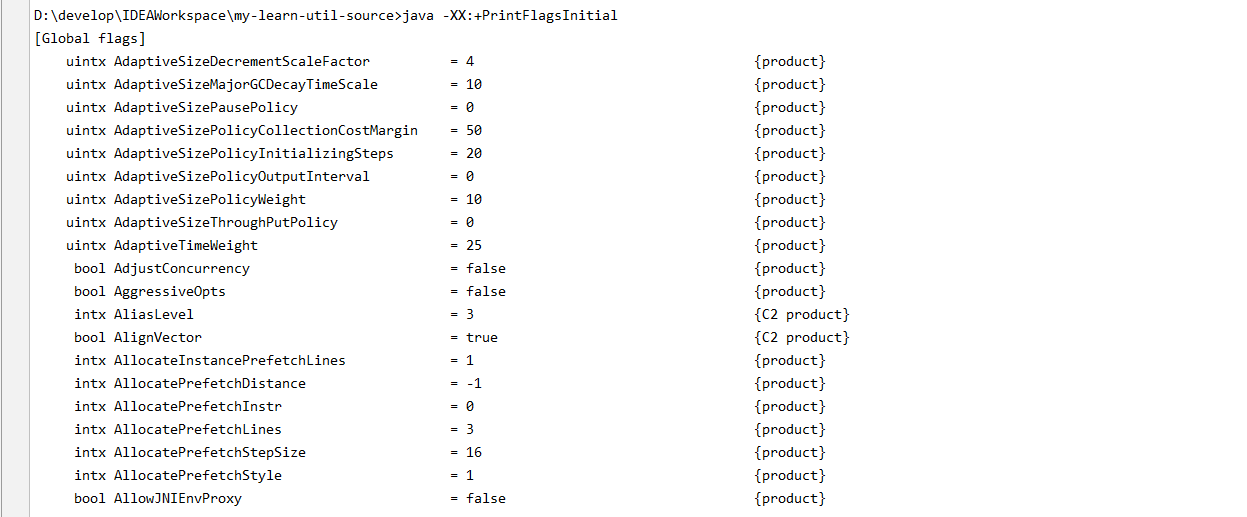
**盘点家底查看JVM默认值**

**-XX:PrintFlagsInitial** 主要是查看初始默认值，程序不需要启动

java -XX:+PrintFlagsInitial -version

java -XX:+PrintFlagsInitial

特别多



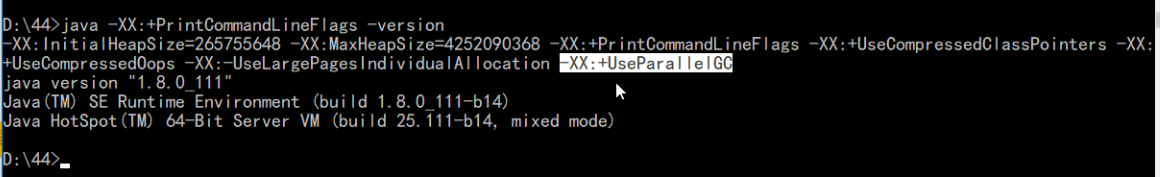
-XX:+PrintFlagsFinal主要查看修改更新

=表示没改过

:= 人为改过或者Jvm自己去修改过的

-XX:+PrintCommandLineFlags

最后一个是查看默认的垃圾回收器。

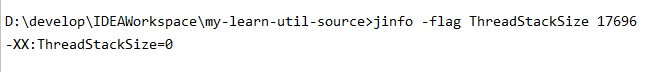


**三、JVM常用参数清单和配置列表**

**-Xms**  初始内存大小，默认是1/64 等价于 -XX:InitialHeapSize

**-Xmx** 最大内存分配，默认是1/4 等价于 -XX:MaxHeapSize

**-Xss** 设置单个线程栈的大小，一般默认是512k - 1024k 等价于 -XX:ThreadStackSize



如果这个值是0，则代表使用的是系统默认的栈空间大小。

栈管运行，堆管存储。

**-Xmn** 设置新生区，年轻代的大小，一般使用默认值即可

**-XX:MetaspaceSize**  设置元空间的大小

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，因此，默认情况下，元空间的大小仅仅受到本地内存的限制。

-Xms10m -Xmx10m -XX:MetaspaceSize=1024m -XX:PrintFlagsFinal

**-XX:+PrintGCDetails**

**-XX:SurvivorRatio**

**-XX:NewRatio**

**-XX:MaxTenuringThreshould**