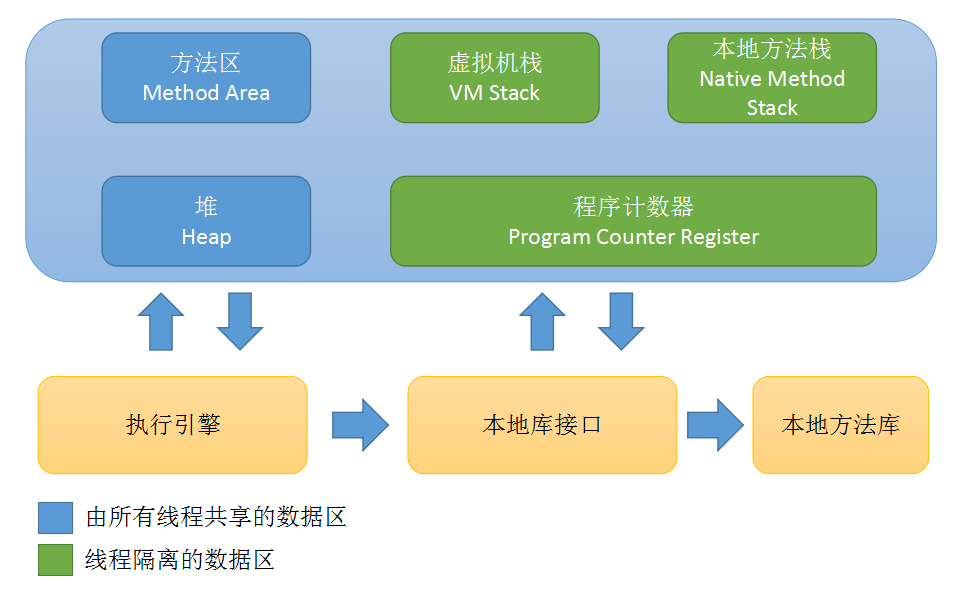
**Java虚拟机**

# 1 Java内存区域与内存溢出异常

Java与C++之间有一堵由内存动态分配和垃圾收集技术所围成的“高墙”，墙外面的人想进去，墙里面的人却想出来。

## 运行时数据

Java虚拟机在执行的过程中会把它所管理的内存划分为若干个不同的数据区域。各个区域的都有各自的用途，以及创建销毁的时间，有的内存随着虚拟机启动而存在，有些区域依赖用户线程的启动和结束而建立和销毁。



## 程序计数器

程序计数器是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器。

### 1. 作用

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还行的字节码指令，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。

### 2. 特性

**1 .** 因为处理器在某一个特定时刻只会执行一个线程的指令，线程切换后，是通过计数器来记录执行痕迹的，因而可以看出，程序计数器是每个线程私有的。

**2.** 如果执行的是Java方法,那么记录的是正在执行的虚拟机字节码指令的地址，如果是native方法，计数器的值为空。

**3.** 这个内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError异常的区域。

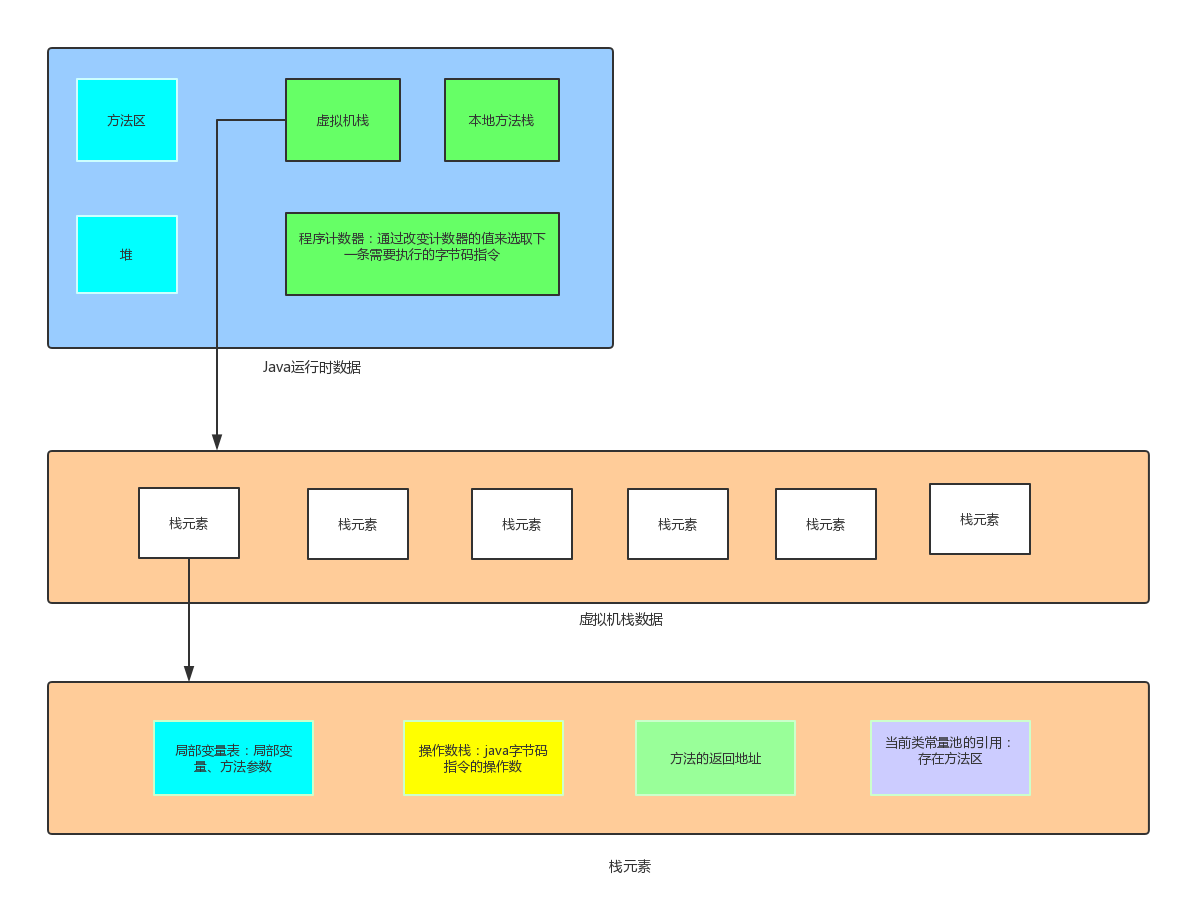
## 3. Java虚拟机栈

Java虚拟机栈是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型。

### 1. 什么是虚拟机栈

虚拟机栈中存的栈元素是栈帧。当有一个方法被调用时，代表这个方法的栈帧入栈；当方法返回时出栈。因此虚拟栈中的入栈顺序就是方法的调用顺序

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。

方法中定义的局部变量和方法中的参数就存在局部变量表中。而Java的字节码指令的操作数存放在操作数栈中,当执行某条带n个操作数的指令时，就从栈顶取n个操作数，然后把指令的计算结果（如果有的话）入栈。因此当我们说JVM的执行引擎是基于‘栈’的时候，其中的栈就是指的操作数栈。

### 2. JVM指令

#### 1. 示例1

**代码 ：**

**int a = 1 + 2 ;**

**JVM指令：**

**iconst\_1 // 把整数1压入操作数栈**

**iconst\_2 // 把整数2压入操作数栈**

**iadd // 栈顶的两个数相加后出栈,结果入栈**

#### 2. 示例2

**代码 ：**

**int a = 1 + 2;**

**int b = a + 3;**

**JVM指令：**

**iconst\_1 // 把整数1压入操作数栈**

**iconst\_2 // 把整数2压入操作数栈**

**iadd // 栈顶的两个数相加后出栈,结果入栈;实际上前三步会被编译器优 化为：inconst\_3**

**iload\_1 // 将局部变量表索引1的slot中存放的变量值(3)加载至操作数栈**

**iconst\_3**

**iadd // 栈顶的两个数相加后出栈，结果入栈**

**istore\_2 //栈顶的内容放入局部变量表中索引为2的slot中，也就是b对应的空间中**

**return**

##### 注意

局部变量表中以及操作数栈的容量的最大值在编译时已经确定，运行时不会改变。并且局部变量表的空间是可以复用的，例如，当指令的位置超出局部变量表中的莫个变量a的作用域时，如果有新的局部变量b要被定义，b就会覆盖a在局部变量表的空间。

##### 什么是slot

slot是局部变量表中的空间单位，虚拟机规范中有规定，对于32位之内的数据，用一个slot来存放，如int，short，float等；对于64位的数据用连续的两个slot来存放，如long,double等。引用类型的变量JVM并没有规定其长度，它可能是32位 ，也可能是64位，所以既有可能占一个slot，也有可能占两个slot。

#### JVM字节码指令

##### 指令格式

Java 的指令以字节为单位，也就是一个字节代表一条指令。比如iconst\_1就是一条指令，它占一个字节，那么自然Java指令不会超过256条。实际上Java指令目前定义了200多条。指令虽然是一个字节，但是它可以带自己的操作数。JVM中有这样一条指令putstatic，其作用是给特定的静态字段赋值。但是给哪个字段赋值呢？仅仅通过这条指令并不能说明，那么只有通过操作数来指定了。紧跟在putstatic后面的两个字节就是它的操作数，这个操作数是一个索引值，指向运行时常量池中该静态字段的符号引用。由于符号引用包含了该字段的基本信息，如所属类、简单名称以及描述符，因此putstatic指令就知道是给哪个类的哪个字段赋值了。

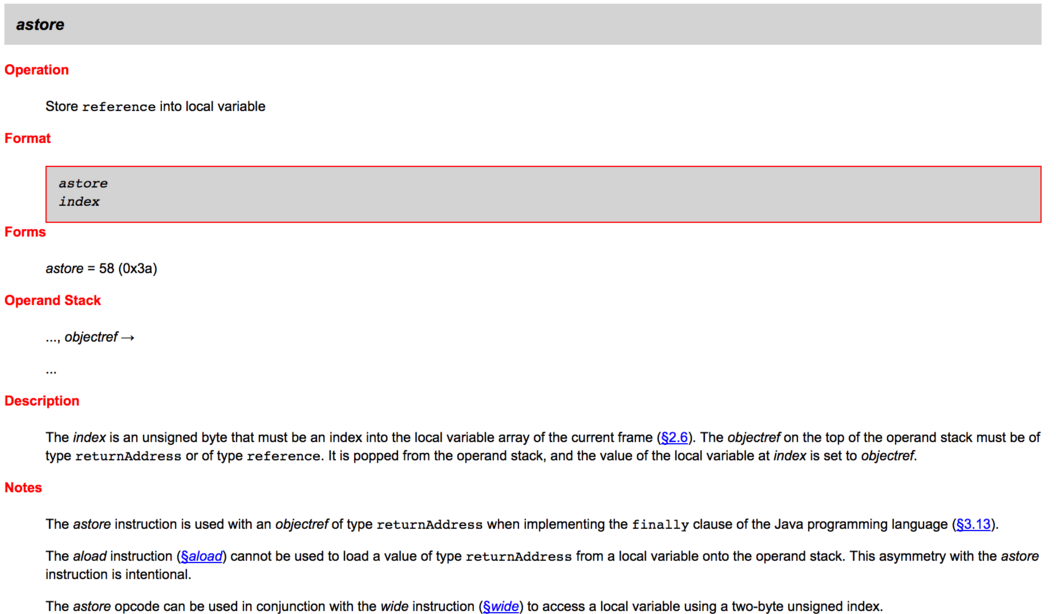
指令的操作数分两种：一种是嵌入在指令中的，通常是指令字节后面的若干个字节；另一种是存放在操作数栈中的。为了区别，我们把前者叫做嵌入式操作数，把后者叫做栈类操作数。这两者的区别是：嵌入式操作数是在编译时就已经确定，运行时不会改变，它和指令一样存放于类文件方法表的Code属性中；而操作数是运行时确定的，即程序在执行过程中动态生成的。拿putstatic指令来说，它有一个嵌入式操作数，该操作数是一个索引值，它由两个字节组成，紧跟在pustatic对应的字节之后；同时它还有一个栈内操作数，位于操作数的栈顶，这个操作数就是要赋给静态字段的值，其对应的字节数根据静态字段的类型决定。如果静态字段的类型是short、int、boolean、double或者long类型，那么操作数就是相应的类型，即由栈顶的4个、8个或者8个字节组成；如果静态字段是引用类型，那么这个操作数的类型也必须是引用类型，即由栈顶的8个字节组成。

iconst\_<i>代表了一个指令族，它的意思是把整个i放入操作栈中，i的范围是(m1,0,1,2,3,4,5)，其中m1代表的是-1。注意，这里的i并不是指令的操作数(即非嵌入式操作数，也非栈内操作数)如iconst\_1、iconst\_2和iconst\_3都是一个由一个字节码指令。我们可以把i可以看作是指令的“隐藏操作数”，即指令本身就蕴含了操作数。如果整数i超过[-1,5] 这个范围，就不能用iconst\_<i>表示了，因为仅一个字节的指令不可能蕴含所有的整数。此时就需要bipush这个条指令了，这条指令有一个嵌入式操作数，由一个字节组成，用来表示要放入栈顶的那个整数，该整数放入栈顶时通过扩展符号位变为32位的整型。但是一个字节也表示不了所有的整数，如果整数值超过一个字节所能表示的范围，那就只能通过Idc这条指令了，这条指令带有一个字节的嵌入式操作数，它代表的是一个指向运行时常量池中Constant\_Integer\_info类型常量的索引，通过索引的方式引用运行时常量池中的整数，再大的整数也不怕了。

##### 怎样阅读JVM指令

文档的地址是：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html>

示例：astore命令



说明:

* 第一行的粗体字是指令的名称
* Operation是指令的功能：把引用存入到本地变量中
* Format是指令的格式：它的第一个字节是指令，名称为astore，第二个字节是指令的嵌入式操作数，名称为index；Forms指的是指令的十进制(十六进制)码，astore的十进制(十六进制)码是58 (0x3a)
* Operation stack是指令执行前后的操作数栈的状态：第一行代表的是指令执行前操作数的状态，第二行是指令执行后操作数的状态，箭头是栈顶方向。Astore命令执行前栈顶是对象引用objectRef,它是astore的栈内操作数,执行后objectRef被弹出并存入局部变量表中
* Description是对这条指令的描述：index是无符号字节，这个index必须指向

局部变量表的某个位置。操作数栈的栈顶的那个引用值必须是returnAddress(返回地址)或者是reference(对象引用)。这个引用会被弹出，其值会被存入局部变量表中索引为index的slot中；

* Notes是注意事项：实现Java中的finally子句时，astore指令使用的操作数类型是一个returnAddress,与astore对应的aload指令(将局部变量表的引用值压栈)不能将类型为returnAddress类型的值加载到操作数栈，而只能是reference类型。aload和astore这种不对称的设计是有意而为之的。astroe指令可以和wide指令配合使用以无符号双字节类型的索引来获取局部变量表中的变量。

##### 局部变量表的第一变量

从Java语言层面上讲，静态方法和实例方法的本质区别在于是否是对象所共享的。而从JVM的角度来看，不管是静态方法还是实例方法其实都是对象共享的，实例变量才是对象所私有的。对JVM而言，静态方法和实例方法的本质区别在于是否需要和具体对象关联：静态方法可通过类名来调用,它不需要和具体的对象关联；而实例方法必须通过对象来调用，它需要和具体对象关联。那么实例方法如何与具体对象产生关联的呢？其实很简单，编译器在编译时会将方法接受者作为一个隐含的参数传入该实例方法，这个参数在方法中有一个很熟悉的名字，叫做”this”。之所以实例方法可以访问该类的的实例变量和其他实例方法，正因为它有“this”这个隐含参数。举个例子，类A中的某个方法b需要访问实例变量x，由于实例变量是对象私有的，如果b是静态方法，由于它没有具体对象引用，它并不知道该访问哪个对象的实例变量x;如果b是实例方法，通过隐藏参数this就能确定要访问的实例变量是this.x。那么，为什么静态方法也不能调用该类的实例方法？本质原因也是没有this引用。因为调用实例方法的前提是要传入一个隐藏参数，实例方法本来就有这个引用，所以能够把它作为隐藏参数传给另一个实例方法；静态方法没有this引用，无法给实例方法提供指向方法接受者的隐含参数，因此不能调用实例方法。

因为我们定义的方法时void foo(), 它是一个实例方法，因此会有一个指向具体对象的的隐含参数this，this就存在局部变量表的第一个位置，即存放在索引为0的slot中，又由于它的作用域从方法一直到结束，因此它在局部变量表中的位置不会被其他变量覆盖，从而使得我们在方法中定义的变量只能放到局部变量表后面的位置。需要注意的是，如果方法参数（非隐含参数），那么参数会按顺序紧接着this存放在局部变量表中，由于参数作用域也是整个方法体，所以方法中定义的局部变量就只能放到参数后面了。总的来说局部变量表中变量的存放顺序是：this(如果是实例方法) => 参数(如果有的话) => 定义的局部变量(如果有的话)

## 本地方法栈

对于一个运行中Java程序而言，它还可能会调用一些跟本地方法相关的数据区。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。本地方法可以通过本地方法接口来访问虚拟机运行时数据区，但不止如此，他还可以做它想做的事。

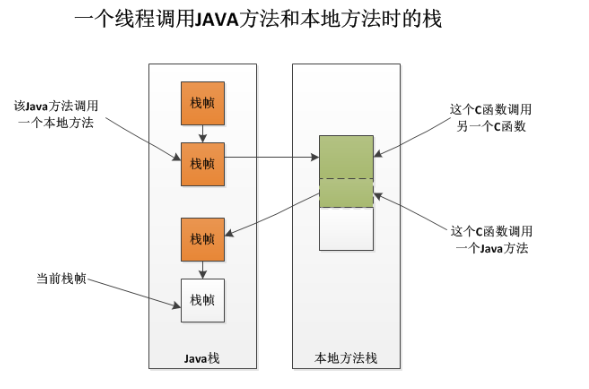
任何本地方法接口都会使用某种本地方法栈。当线程调用java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。

如果某个虚拟机实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。当C程序调用一个C函数时，其栈操作都是确定的。传递给该函数的参数以某个确定的顺序压入栈，它的返回值也以确定的方式传回调用者。同样，这就是虚拟机实现中本地方法栈的行为。

很可能本地方法接口需要回调Java虚拟机中的Java方法，在这种情况下，该线程会保存本地方法栈的状态并进入到另一个Java栈。

下图描绘了这样一个情景，就是当一个线程调用一个本地方法时，本地方法又回调虚拟机中的另一个Java方法。

　　这幅图展示了JAVA虚拟机内部线程运行的全景图。一个线程可能在整个生命周期中都执行Java方法，操作它的Java栈；或者它可能毫无障碍地在Java栈和本地方法栈之间跳转。



## Java堆

1. 对于大多数应用来说，Java堆是Java虚拟机所管理的内存中最大的一块。
2. Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。
3. Java堆内存是存放对象实例的区域。
4. Java堆是垃圾收集器管理的主要区域，因此也被称作为“GC堆”。
5. Java堆可以处于物理上不连续的两个内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间。如果堆中没有内存完成例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。
6. 堆内存中所有的实体都有内存地址值。堆内存中的实体是用来封装数据的，这些数据都有默认初始化值。

## 方法区

1. 线程共享的内存区域
2. 用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

**类型信息**

**对于每个加载的类型，jvm必须在方法区中存储以下的类型信息：**

* + 1. 这个类型的完整有效名
    2. 这个类型直接父类的完整有效名(除非这个类型是interface或是java.lang.Object，两种情况下都没有父类。)
    3. 这个类型的修饰符(public，abstract，final)
    4. 这个类型直接接口的一个有序列表

**常量池**

Jvm虚拟机为每个已加载的类型都维护一个常量池。常量池就是这个类型用到的常量的一个有序集合，包括实际的常量(string,integer,和floating point常量)和对类型，域和方法的符号引用。池中的数据像数组一样，通过索引来访问。

因为常量池存储了一个类型所使用到的所有类型,域和方法的引用，所以它在java程序的动态链接中起了核心作用。

**字段信息**

1. 字段修饰符(public, private, protected,static,final   volatile, transient的某个子集)
2. 字段的名称
3. 字段的类型

**方法信息**

* + 1. 方法名
    2. 方法的返回类型
    3. 方法参数的数量和类型(有序的)
    4. 方法的修饰符(public, private, protected, static, final, synchronized, native, abstract的一个子集)除了abstract和native方法外，其他方法还有保存方法的字节码(bytecodes)操作数栈和方法栈帧的局部变量区的大小
    5. 方法字节码
    6. 操作数栈和该方法在栈帧中的局部变量区大小
    7. 异常表

**类变量（静态变量）**

指该类所有对象共享的变量，即使没有任何实例对象时，也可以访问的类变量。它们与类进行绑定。

**指向类加载器的引用**

每一个被JVM加载的类型，都保存这个类加载器的引用，类加载器动态链接时会用到。

**指向Class实例的引用**

类加载的过程中，虚拟机会创建该类型的Class实例，方法区中必须保存对该对象的引用。通过Class.forName(String className)来查找获得该实例的引用，然后创建该类的对象。

**方法表**

为了提高访问效率，JVM可能会对每个装载的非抽象类，都创建一个数组，数组的每个元素是实例可能调用方法的直接引用，包括父类中继承过来的方法。这个表在抽象类或接口中是没有的，类似于C++虚函数表vtbl。

**运行时常量池**

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信息、字段、方法、接口描述信息外，还有一项信息时常量池，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。

# 垃圾收集器与内存分配策略

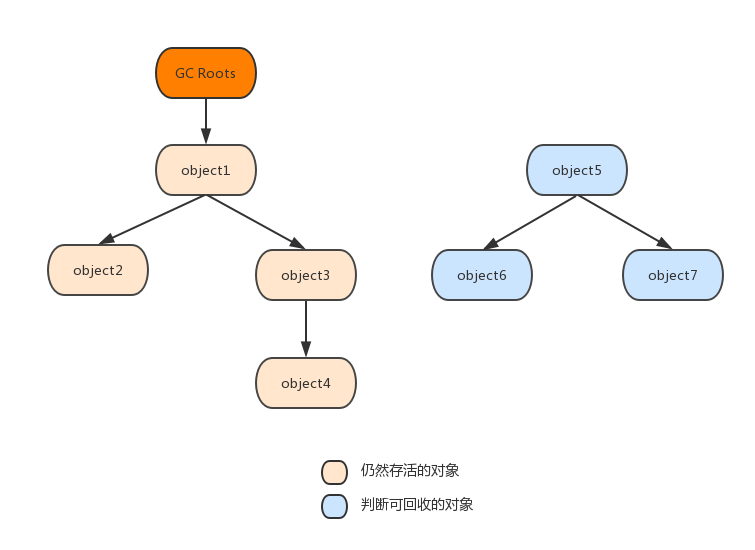
## 怎么判断对象是否存活

### 引用计数算法

给对象添中添加一个引用计数器，每当一个地方引用它，计算器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。 引用计数算法实现简单且性能高效，是一个不错的算法，但是不足是无法解决对象之间循环引用的问题，所以Java并没有使用该算法。

### 可达性分析算法

很多主流的商用程序语言包括Java的主流实现中，都是通过可达性分析来判定对象是否存活。这个算法的基本思路就是通过一系列的称为”GC Roots”的对象作为起始点，从这个节点往下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时,则证明此对象是不可用。

在Java语言中，可作为GC Roots的对象包括下面几种：

1. 虚拟机栈(栈帧中的本地变量表)中引用的对象。
2. 方法区中类静态属性引用的对象。
3. 方法区中常量的引用对象。
4. 本地方法栈中的JNI(即Native方法)引用的对象。

## 引用

### JDK1.2之前定义

如果reference类型的数据存储的数值代表的是另外一块内存地址，就称这块内存代表着一个引用。

### JDK1.2之后的定义

JDK1.2之前定义的引用仅存在引用和没有引用两种状态，无法描述其他一些状态对象的引用。

JDK1.2之后，Java对引用概念进行了扩充，将引用划分为强引用、软引用、弱引用、虚引用。

强引用：指在程序代码之中普片存在的，类似“Object obj = new Object()”这类引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

软引用：描述一些还有但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次 回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK1.2之后，提供了SoftReference类来实现软引用。

弱引用：描述非必须的对象，它的强度要比软引用弱一些，被弱引用关联的对象只能生存到下一次垃圾收之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收调只被弱引用关联的对象。再JDK1.2之后，提供了WeakReference类来实现弱引用。

虚引用：是一种最弱的引用。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhanttomReference类来实现虚拟机引用。

### 生存还是死亡

不可达对象在真正“死亡”过程中，至少要经历两次标记过程：在发现对象不可达时，将会被第一次标记并且进行一次筛选，筛选条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象别判断为有必要执行finalize()方法，那么这个对象会放置在一个叫F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里的“执行”是指虚拟机会触发这个方法，但并不会承诺等待它运行结束。原因是防止finalize()执行时间过长，或者出现死循环，那么可能会导致队列中其他的对象处于永久等待。

但并不提倡使用finalize()方法，finalize()所能做的工作，try-finally或者其他的方法都可以做得更好、更及时。

### 回收方法区

永久代的垃圾回收两部分内容：废弃常量和无用的类。

废弃常量：

如果一个常量没有任何对象引用指向这个常量，如果有必要，这个常量就会被系统清理出常量池。

废弃的类：

1. 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
2. 加载该类的ClassLoader已经被回收。
3. 该类对象的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

## 垃圾收集算法

### 标记-清除算法

标记清除算法是现代垃圾回收算法的基础(以为后面的垃圾算法都是基于该算法的改进)。

标记清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。

标记阶段：HotSpot虚拟机采用可达性分析算法将有用的对象进行标记，未被标记的对象就是垃圾对象。

清除阶段：清除所有未被标记的对象。

标记清除算法的缺点：

* 该算法是对一块连续的内存空间进行回收,回收后会产生大量不连续的内存碎片。
* 不连续的内存空间会使得垃圾收集以及内存分配工作效率大大降低。

### 标记-复制算法

为了解决效率问题，一种称为“复制”算法的收集算法出现了

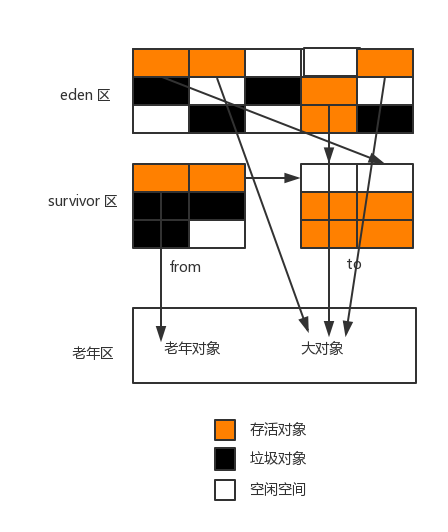
**复制算法**：

它将可用内存容量划分为大小相等的两块，每次只使用其中的一块。当这一块用完之后，就将存活的对象复制到另一块内存上，然后再把已使用过的内存空间一次清理掉。这样使得每次对整个半区进行内存回收，内存分配时也就不用考虑内存碎片的复杂情况，只要移动指针即可，实现简单、高效。

缺点：

代价是将内存缩小为了原来的一半。

在Java的新生代串行垃圾回收器中，使用了复制算法的思想。新生代分为eden空间、from和to空间。其中from和to空间可以视为大小相同、地位相等且可进行角色互换的空间块。From和to空间为被称为survivor空间。

在垃圾回收时，eden空间中的存活对象会被复制到未使用的survivor空间中(假设是to),正在使用的suvivor(假设是from)中的年轻对象也会被复制到to空间(大对象或者老年对象会直接进入到老年区)，如果to空间已满，则使用老年区空间担保，存活对象直接进入到老年代。当所有存活对象都复制完成之后，survivor(from空间)和eden空间则全部是垃圾对象，直接清空。

**注意：**

复制算法比较适用于新生代。因为在新生代中，绝大部分都是垃圾对象，这样可以保持较少的复制次数，效率比较高。

### 标记-压缩算法

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下。这种情况在新生代经常发生。但是在老年代大部分都是存活对象。

标记压缩算法首先也是先标记，但不会立马清除不可达对象，而是将可达对象压缩移动到内存的一端；

在移动对象时应考虑到对象的移动顺序，一般分为三种：

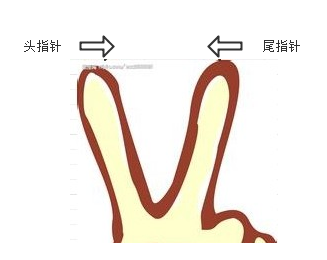
1. 任意顺序：不考虑原先对象的排列顺序，也不考虑对象间的引用关系，可随意的移动对象，这样可能导致内存访问的局部性问题。
2. 线性顺序： 在重新排列顺序时，会考虑对象的引用，尽可能将引用对象排列在一起。
3. 滑动顺序：就是在重新排列对象时，将对象按照原先堆内存中的排列顺序滑动到堆的一端。

#### Tow-Finger算法

Tow-Finger算法是任意顺序移动的一种实现。

**原理** ：

Tow-Finger算法需要经过两次遍历，第一次遍历堆将末尾的可达对象移动到堆开始的空闲内存中去，第二次遍历修改可达对象的引用。由于移动的过程中两个指针相向而行，酷似两个手指合拢，因此叫做Tow-Finger算法。



**第一次遍历**：

