

**基本概念：栈是运行时的单位，而堆是存储的单位。**堆存储的对象信息是所有线程共享的。栈（main函数是栈的起始点，也是程序的起始点）存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量（如基本数据类型和堆中对象的引用）、程序运行状态、方法返回值等。

**堆：**所有通过new创建的对象的内存都在堆中分配，其大小可以通过**-Xmx**和**-Xms**来控制。堆被划分为新生代和老年代，新生代又被进一步划分为Eden和Survivor区，最后Survivor由FromSpace和ToSpace组成

新生代：新建的对象都是用新生代分配内存，Eden空间不足的时候，会把存活的对象转移到Survivor中，新生代大小可以由-Xmn来控制，也可以用**-XX:SurvivorRatio**来控制Eden和Survivor的比例。

旧生代：用于存放新生代中经过多次垃圾回收仍然存活的对象。

**本地方法栈：**为线程私有，功能和虚拟机栈非常类似。线程在调用本地方法时，来存储本地方法的局部变量表，本地方法的操作数栈等信息。

**Java栈：**每个线程执行每个方法的时候都会在栈中申请一个栈帧，每个栈帧包括局部变量区、操作数栈、帧数据区，用于存放此次方法调用过程中的临时变量、参数和中间结果。

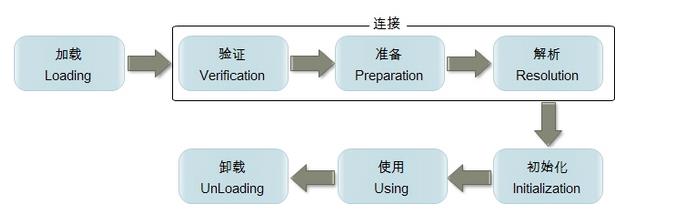
**方法区：**存放了要加载的类信息、静态变量、final类型的常量、属性和方法信息。JVM用永久代（PermanetGeneration）来存放方法区，（在JDK的HotSpot虚拟机中，可以认为方法区就是永久代，但是在其他类型的虚拟机中，没有永久代的概念，有关信息可以看周志明的书）可通过**-XX:PermSize**和**-XX:MaxPermSize**来指定最小值和最大值。

**pc寄存器：**用于存放下一条将要执行的指令的地址(字节码流)。

**JVM常用参数选项：**

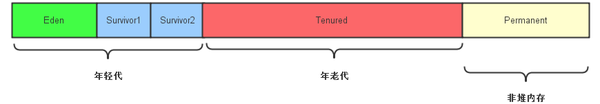
|  |  |
| --- | --- |
| -Xms | 初始堆大小。如：-Xms256m |
| -Xmx | 最大堆大小。如：-Xmx512m |
| -Xmn | 新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90% |
| -Xss | JDK1.5+ 每个线程堆栈大小为 1M，一般来说如果栈不是很深的话， 1M 是绝对够用了的。 |
| -XX:NewRatio | 新生代与老年代的比例，如 –XX:NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3 |
| -XX:SurvivorRatio | 新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10 |
| -XX:PermSize | 永久代(方法区)的初始大小 |
| -XX:MaxPermSize | 永久代(方法区)的最大值 |
| -XX:+PrintGCDetails | 打印 GC 信息 |
| -XX:+HeapDumpOnOutOfMemoryError | 让虚拟机在发生内存溢出时 Dump 出当前的内存堆转储快照，以便分 |

**类加载（classloder）机制，加载过程如下：**



1. **加载：**类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象
2. **验证：**目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。
3. **准备：**为类变量(即static修饰的字段变量)分配内存并且设置该类变量的初始值即0(如static int i=5;这里只将i初始化为0，至于5的值将在初始化时赋值)，这里不包含用final修饰的static，因为final在编译的时候就会分配了，注意这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。
4. **解析：**主要将常量池中的符号引用替换为直接引用的过程。符号引用就是一组符号来描述目标，可以是任何字面量，而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，接口方法解析(这里涉及到字节码变量的引用，如需更详细了解，可参考《深入Java虚拟机》)。
5. **初始化：**类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量(如前面只初始化了默认值的static变量将会在这个阶段赋值，成员变量也将被初始化)。

**对象的实例是存在于堆中，逻辑堆的具体结构如图：**

****

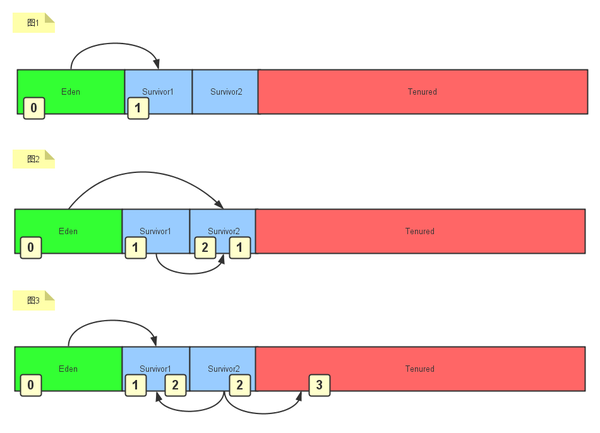
**年轻代 :**

年轻代被分为 eden survivor1 survivor2 ，新实例化的对象都是存在于年轻代中的eden区，按照GC的运行机制，会回收掉已经死掉的对象，而对象一般都是在年轻代就会死去，所以年轻代比老年代需要更频繁的GC清理。

在年轻代中jvm使用的是Mark-copy算法，第一是标记(Mark) 第二是copy（复制），Mark主要用于标记出还活着的实例，然后清除掉没有被标记的实例，释放内存，然后Copy部分则是将还活着的实例根据年龄拷贝到不同的年龄代。

对于标记与区分年龄代的技术一般用到的都是引用计数器，在每一个对象中都含有引用计数器，都有引用指向对象的时候引用计数器就会加1，不再被引用计数器减 1，对与垃圾回收的策略则是标记所有活着的实例，将没有被标记的实例全部回收释放内存，静态方法与静态变量是不会产生实例的，直接通过类的引用，使用 ClassLoader进行加载的类数据是存在于永生代里面也就是方法区里面，这个类一旦被清除，里面所有的静态变量都会被清除

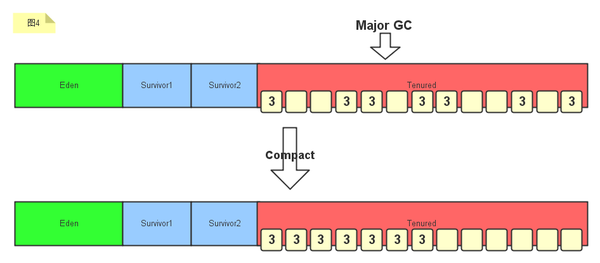
当我们在 Object obj 的时候 向逻辑堆中的 Eden区域 申请内存，当Eden区域的内存不足的时候，这个时候会触发GC这个时候称GC为小型垃圾回收，每个实例都有一个独有的年龄，每个引用被经历过一次GC后就会年龄加1，同时就会将没有被清理掉的对象全都copy到上图的survivor1区域，如图1所示：



当第二次GC执行的时候就会使用Mark算法找到存活的对象，然后将他们的年龄加1，并且将他们拷贝到survivor2区域，然后执行GC，这样就可以实现survivor1 与 survivor2 两个一样大的区域进行交替使用，当对象的年龄足够大的时候，对象就会被移动到老年代，这里移动到老年代的标准由JVM的参数所决定

**年老代 :** 当GC被触发的时候 eden的对象会转到 survivor1 然后再次就会转到 survivor2 ，当survivor1的对象太大了 survivor2的区域无法容纳得部分就会转到Tenured的区域，当Tenured的区域也容不下的时候就会自动移动到年老代，在移动年老代的时候会先触发年老代上面的GC然后在将Tenured容纳不下的对象放入年老代，对于年老代的GC算法与年轻代的Mrak-copy算法有很大不同

在这里介绍Parallel Scavenge 算法，PS算法执行的是 Mark-compact算法的过程 ，并且是用多线程进行执行这样提高了执行效率，这里的Mrak还是与之前的年轻代的Mark原理是一样的，但是Compat算法则是将年老代的对象进行碎片化的整理，并且年老代是没有像年轻代的那样有survivor1 与 survivor2来将残留的对象全部copy过去，考虑到年老代的对象比较多，所以就需要进行碎片化整理如下图：



**GC回收原理：**

可达性分析法算法的基本思想是通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，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（即GC Roots到对象不可达）时，则证明此对象是不可用的。

在Java语言中，可以作为GCRoots的对象包括下面几种：

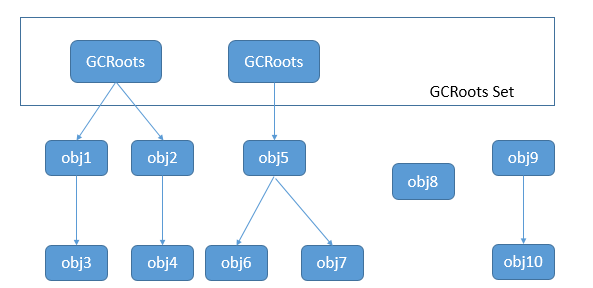
(1). 虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）中引用的对象。

(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。

GCRoots的引用链如图：



由图可知，obj8、obj9、obj10都没有到GCRoots对象的引用链，即便obj9和obj10之间有引用链，他们还是会被当成垃圾处理，可以进行回收。

对于可达性分析算法而言，未到达的对象并非是“非死不可”的，若要宣判一个对象死亡，至少需要经历两次标记阶段。

1. 如果对象在进行可达性分析后发现没有与GCRoots相连的引用链，则该对象被第一次标记并进行一次筛选，筛选条件为是否有必要执行该对象的finalize方法，若对象没有覆盖finalize方法或者该finalize方法是否已经被虚拟机执行过了，则均视作不必要执行该对象的finalize方法，即该对象将会被回收。反之，若对象覆盖了finalize方法并且该finalize方法并没有被执行过，那么，这个对象会被放置在一个叫F-Queue的队列中，之后会由虚拟机自动建立的、优先级低的Finalizer线程去执行，而虚拟机不必要等待该线程执行结束，即虚拟机只负责建立线程，其他的事情交给此线程去处理。
2. 对F-Queue中对象进行第二次标记，如果对象在finalize方法中拯救了自己，即关联上了GCRoots引用链，如把this关键字赋值给其他变量，那么在第二次标记的时候该对象将从“即将回收”的集合中移除，如果对象还是没有拯救自己，那就会被回收。如下代码演示了一个对象如何在finalize方法中拯救了自己，然而，它只能拯救自己一次，第二次就被回收了。

**方法区的垃圾回收：**

方法区的垃圾回收主要回收两部分内容：1. 废弃常量。2. 无用的类。既然进行垃圾回收，就需要判断哪些是废弃常量，哪些是无用的类。

如何判断废弃常量呢？以字面量回收为例，如果一个字符串“abc”已经进入常量池，但是当前系统没有任何一个String对象引用了叫做“abc”的字面量，那么，如果发生垃圾回收并且有必要时，“abc”就会被系统移出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

如何判断无用的类呢？需要满足以下三个条件

1. 该类的所有实例都已经被回收，即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. 该类对应的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。

满足以上三个条件的类可以进行垃圾回收，但是并不是无用就被回收，虚拟机提供了一些参数供我们配置，前文有所列举