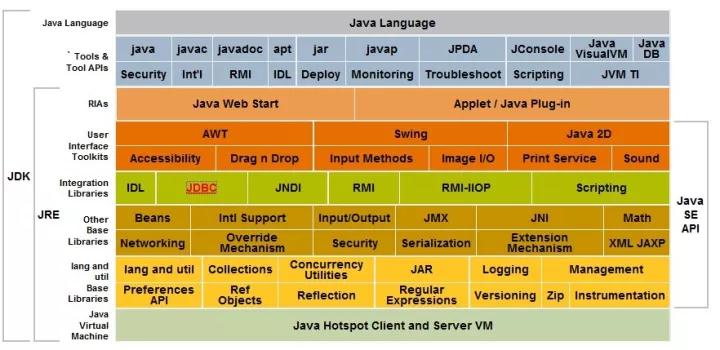
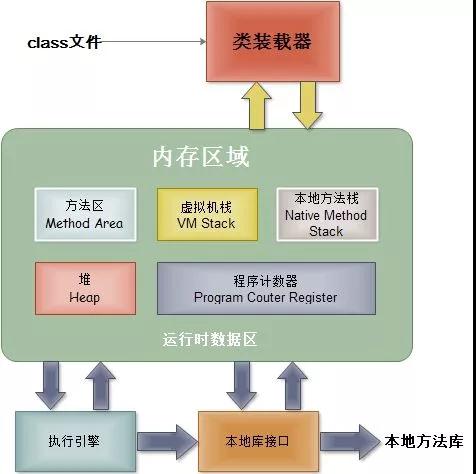
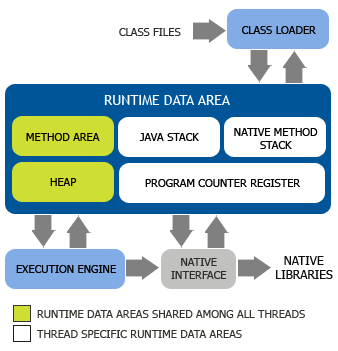
## 1: [内存模型](http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479241&idx=2&sn=3b117753bcc9d710b131cac86738e0cd&chksm=bd2530b68a52b9a026ace7b20fbdb1d19f9e058f85064b896427df080f568dcc2c9abe64a223&scene=21#wechat_redirect)

**Java技术体系模块图**

****

**JVM内存区域模型**

****



### **1.1 方法区(PermGen Space)**

也称”永久代”，用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。

默认最小值为16MB，最大值为64MB，可通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

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，用于存放编译器生成的各种符号引用，这部分内容将在类加载后放到方法区的运行时常量池中。

### **1.2 虚拟机栈(Java Virtual Machine Stacks)**

栈帧：一个栈帧随着一个方法的调用开始而创建，这个方法调用完成而销毁。栈帧内存放者方法中的局部变量，操作数栈等数据。

描述的是Java 方法执行的内存模型： Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池（运行时常量池的概念在方法区部分会谈到）的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。对于所有的程序设计语言来说，栈这部分空间对程序员来说是不透明的。

局部变量表存放了编译器可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用(引用指针，并非对象本身)，其中64位长度的long和double类型的数据会占用2个局部变量的空间，其余数据类型只占1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量是完全确定的，在运行期间栈帧不会改变局部变量表的大小空间。

栈内存的大小可以有两种设置，固定值和根据线程需要动态增长。

1. StackOverflowError 出现在栈内存设置成固定值的时候，当程序执行需要的栈内存超过设定的固定值会抛出这个错误。   
2. OutOfMemoryError 出现在栈内存设置成动态增长的时候，当JVM尝试申请的内存大小超过了其可用内存时会抛出这个错误。

### **1.3 本地方法栈**

与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法（非Java语言的方法：methods written in a language other than the Java programming language）服务。

#### **1.4 堆(Heap Memory)**

也叫做java 堆、GC堆，是java虚拟机所管理的内存中最大的一块内存区域，也是被各个线程共享的内存区域，用来存放对象和数组（特殊的对象）。在JVM启动时创建。该内存区域存放了对象实例及数组(所有new的对象)。

其大小通过-Xms(最小值)和-Xmx(最大值)参数设置，-Xms为JVM启动时申请的最小内存，默认为操作系统物理内存的1/64但小于1G，-Xmx为JVM可申请的最大内存，默认为物理内存的1/4但小于1G，默认当空余堆内存小于40%时，JVM会增大Heap到-Xmx指定的大小，可通过

-XX:MinHeapFreeRation=来指定这个比列；当空余堆内存大于70%时，JVM会减小heap的大小到-Xms指定的大小，可通过XX:MaxHeapFreeRation=来指定这个比列，对于运行系统，为避免在运行时频繁调整Heap的大小，通常-Xms与-Xmx的值设成一样。

由于现在收集器都是采用分代收集算法，堆被划分为新生代和老年代。新生代主要存储新创建的对象和尚未进入老年代的对象。老年代存储经过多次新生代GC(Minor GC)任然存活的对象。

**新生代：**

程序新创建的对象都是从新生代分配内存，新生代由Eden Space和两块相同大小的Survivor Space(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRation来调整Eden Space及Survivor Space的大小。

**老年代：**

用于存放经过多次新生代GC任然存活的对象，例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：①.大对象，可通过启动参数设置-XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。②.大的数组对象，切数组中无引用外部对象。

老年代所占的内存大小为-Xmx对应的值减去-Xmn对应的值。

### **1.5 程序计数器**

是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，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、异常处理、线程恢复等基础功能都需要依赖计数器完成。

**直接内存**

直接内存并不是虚拟机内存的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

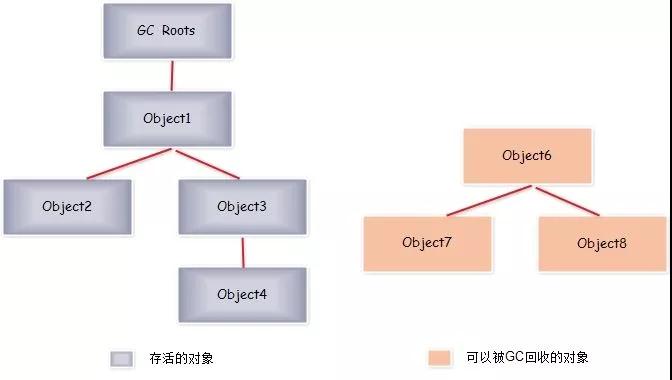
## 2: [垃圾收集算法](http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479245&idx=2&sn=7ffbc7bbad8f9063ecae70a573329801&chksm=bd2530b28a52b9a4100d8397c16916087e566f042271a7be40441e01b52fd2d3ab947f2fff47&scene=21#wechat_redirect)

**跟踪收集器**

在主流的商用程序语言中(Java和C#)，都是使用根搜索算法(GC Roots Tracing)判断对象是否存活的。这个算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，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。

跟踪收集器采用集中式的管理方式，全局记录对象之间的引用状态，执行时从一些列GC  Roots的对象为起点，从这些节点向下开始进行搜索所有引用链，当一个对象到GC  Roots 没有任何引用链时，则证明此对象是不可用。

下图中，对象Object6、Object7、Object8虽然互相引用，但他们的GC Roots是不可到达的，所以它们将会被判定为是可回收的对象。



可作为GC Roots 的对象包括：

* 虚拟机栈(栈帧中的本地变量表)中的引用对象
* 方法区中的类静态属性引用的对象
* 方法区中的常量引用的对象
* 本地方法栈中JNI的引用对象。

GC管理的主要区域是Java堆，一般情况下只针对堆进行垃圾回收。方法区、栈和本地方法区不被GC所管理,因而选择这些区域内的对象作为GC roots,被GC roots引用的对象不被GC回收。

主要有复制、标记清除、标记压缩三种实现算法。

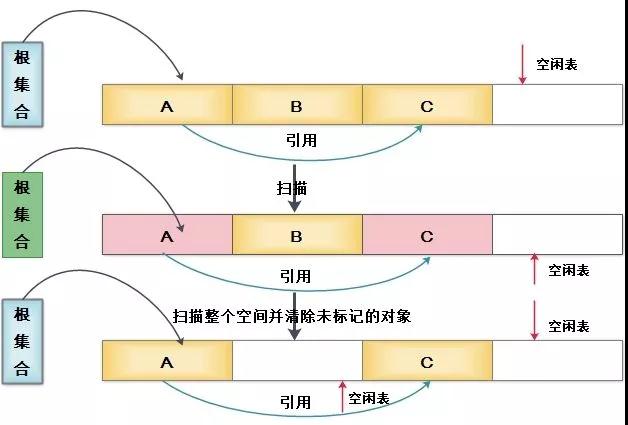
**1. 标记 – 清除算法**

标记清除算法是最基础的收集算法，其他收集算法都是基于这种思想。标记清除算法分为“标记”和“清除”两个阶段：首先标记出需要回收的对象，标记完成之后统一清除对象。

它的主要缺点：

①.标记和清除过程效率不高

②.标记清除之后会产生大量不连续的内存碎片。

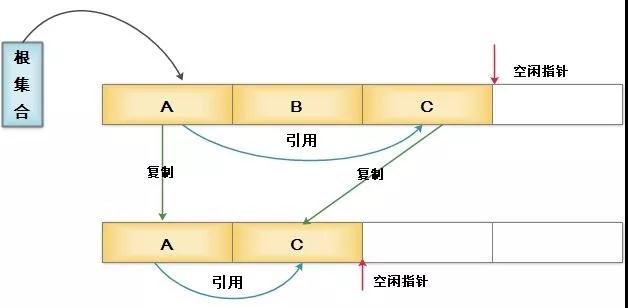




**2. 复制算法**

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，就将还存活的对象复制到另外一块上面，然后在把已使用过的内存空间一次清理掉。这样使得每次都是对其中的一块进行内存回收，不会产生碎片等情况，只要移动堆订的指针，按顺序分配内存即可，实现简单，运行高效。

主要缺点：内存缩小为原来的一半。

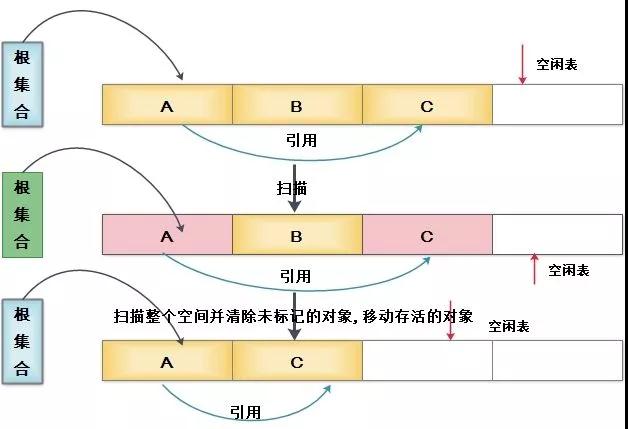


**3. 标记  - 整理算法**

标记操作和“标记-清除”算法一致，后续操作不只是直接清理对象，而是在清理无用对象完成后让所有存活的对象都向一端移动，并更新引用其对象的指针。

主要缺点：

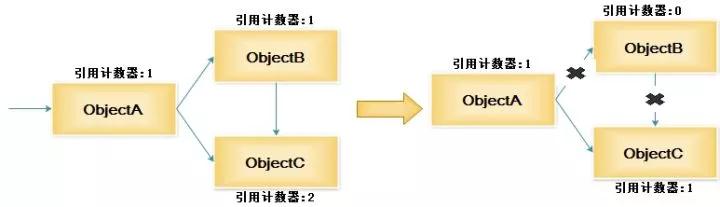
在标记-清除的基础上还需进行对象的移动，成本相对较高，好处则是不会产生内存碎片。



**引用计数收集器**

引用计数收集器采用的是分散式管理方式，通过计数器记录对象是否被引用。当计数器为0时说明此对象不在被使用，可以被回收。

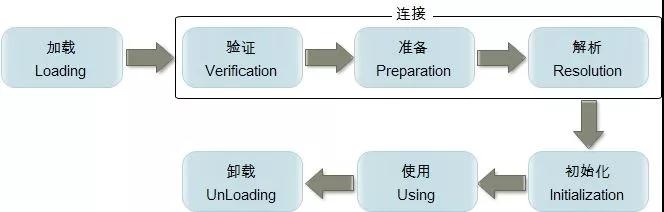
主要缺点：循环引用的场景下无法实现回收，例如下面的图中，ObjectC和ObjectB相互引用，那么ObjectA即便释放了对ObjectC、ObjectB的引用，也无法回收。sunJDK在实现GC时未采用这种方式。



## **3: 类加载机制**

JVM把class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成JVM可以直接使用的Java类型的过程就是加载机制。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的生命周期包括了：加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)、卸载(Unloading)七个阶段，其中验证、准备、解析三个部分统称链接。



加载(装载)、验证、准备、初始化和卸载这五个阶段顺序是固定的，类的加载过程必须按照这种顺序开始，而解析阶段不一定；它在某些情况下可以在初始化之后再开始，这是为了运行时动态绑定特性。值得注意的是：这些阶段通常都是互相交叉的混合式进行的，通常会在一个阶段执行的过程中调用或激活另外一个阶段。

**加载：**

加载阶段是“类加载机制”中的一个阶段，也被称作“装载”，主要完成：

1.通过“类全名”来获取定义此类的二进制字节流；

2.将字节流所代表的静态存储结构转换为方法区的运行时数据结构；

3.在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口；

虚拟机规范对于“通过“类全名”来获取定义此类的二进制字节流”并没有指明二进制流必须要从一个本地class文件中获取，准确地说是根本没有指明要从哪里获取及怎样获取。例如：

* 从Zip包中读取，这很常见，最终成为日后JAR、EAR、WAR格式的基础。
* 从网络获取，常见应用Applet。
* 运行时计算生成，这种场景使用的最多的就是动态代理技术，在java.lang.reflect.Proxy中，就是用ProxyGenerator.generateProxyClass来为特定接口生成$Prxoy的代理类的二进制字节流。
* 由其他格式文件生成，典型场景：JSP应用
* 从数据库中读取，这种场景相对少见，有些中间件服务器(如SAP Netweaver)可以选择把程序安装到数据库中来完成程序代码在集群间的分发。

相对于类加载过程的其他阶段，加载阶段(准确地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，也可以由用户自定义的类加载器完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机并未规定此区域的具体数据结构。然后在java堆中实例化一个java.lang.Class类的对象，这个对象作为程序访问方法区中的这些类型数据的外部接口。加载阶段与链接阶段的部分内容(如一部分字节码文件格式验证动作)是交叉进行的，加载阶段尚未完成，链接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于链接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

**验证：**

验证是链接阶段的第一步，这一步主要的目的是确保class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。

验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

**1.文件格式验证**

验证class文件格式规范，例如： class文件是否已魔术0xCAFEBABE开头 ， 主、次版本号是否在当前虚拟机处理范围之内等

**2.元数据验证**

这个阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范要求。验证点可能包括：这个类是否有父类(除了java.lang.Object之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被final修饰的)、如果这个类的父类是抽象类，是否实现了其父类或接口中要求实现的所有方法。

**3.字节码验证**

进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

**4.符号引用验证**

符号引用中通过字符串描述的全限定名是否能找到对应的类、符号引用类中的类，字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

**准备：**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(static 修饰的变量),而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在java堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value  = 12;

那么变量value在准备阶段过后的初始值为0而不是12，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为12的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在ConstantValue属性（ConstantValue属性的作用是通知虚拟机自动为静态变量赋值，只有被static修饰的变量才可以使用这项属性。非static类型的变量的赋值是在实例构造器方法中进行的；在实际的程序中，只有同时被final和static修饰的字段才有ConstantValue属性，且限于基本类型和String），那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，建设上面类变量value定义为：

public static final int value = 123;

编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value设置为123。

**解析：**

解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。

符号引用：符号引用是一组符号来描述所引用的目标对象，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范并没有规定解析阶段发生的具体时间，只要求了在执行anewarry、checkcast、getfield、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield和putstatic这13个用于操作符号引用的字节码指令之前，先对它们使用的符号引用进行解析，所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析的动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。分别对应编译后常量池内的CONSTANT\_Class\_Info、CONSTANT\_Fieldref\_Info、CONSTANT\_Methodef\_Info、CONSTANT\_InterfaceMethoder\_Info四种常量类型。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

**初始化：**

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。在以下四种情况下初始化过程会被触发执行：

1.遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需先触发其初始化。生成这4条指令的最常见的java代码场景是：使用new关键字实例化对象、读取或设置一个类的静态字段(被final修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用类的静态方法的时候。

2.使用java.lang.reflect包的方法对类进行反射调用的时候

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先出发其父类的初始化

4.jvm启动时，用户指定一个执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value  = 12;  在准备阶段完成后 value的值为0，而在初始化阶调用了类构造器<clinit>()方法，这个阶段完成后value的值为12。

\*类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句快可以赋值，但是不能访问。

\*类构造器<clinit>()方法与类的构造函数(实例构造函数<init>()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

\*由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句快要优先于子类的变量赋值操作。

\*<clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，那么编译器可以不为这个类生成<clinit>()方法。

\*接口中不能使用静态语句块，但接口与类不太能够的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

\*虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

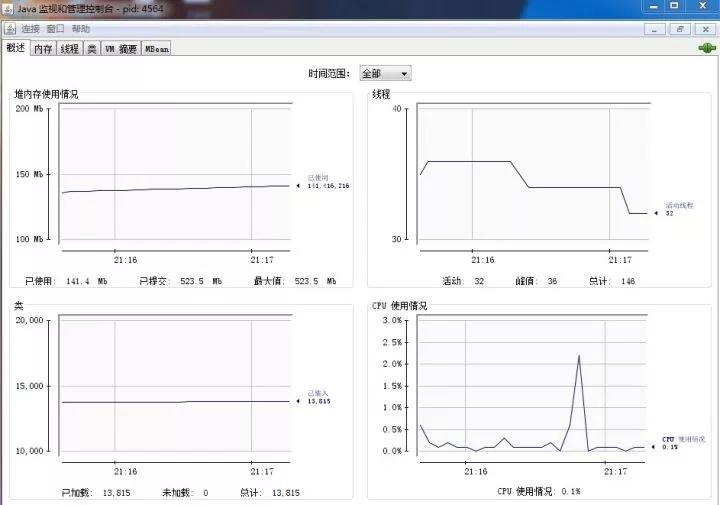
## 4: JDK 可视化监控工具

**1.JConsole**

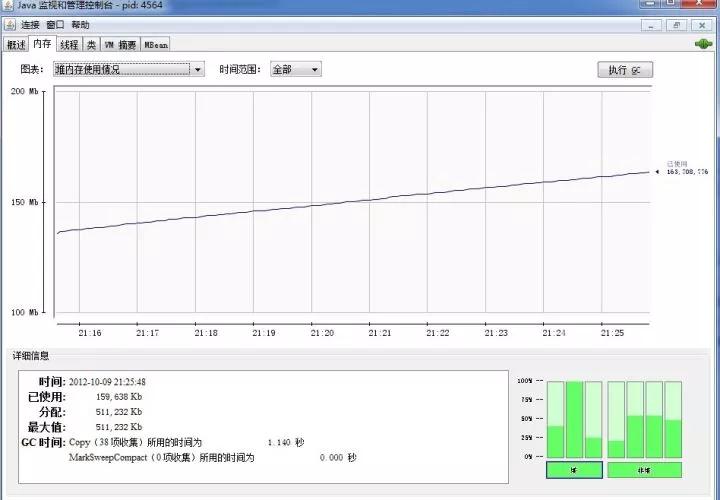
JConsole工具在JDK/bin目录下，启动JConsole后，将自动搜索本机运行的jvm进程，不需要jps命令来查询指定。双击其中一个jvm进程即可开始监控，也可使用“远程进程”来连接远程服务器。



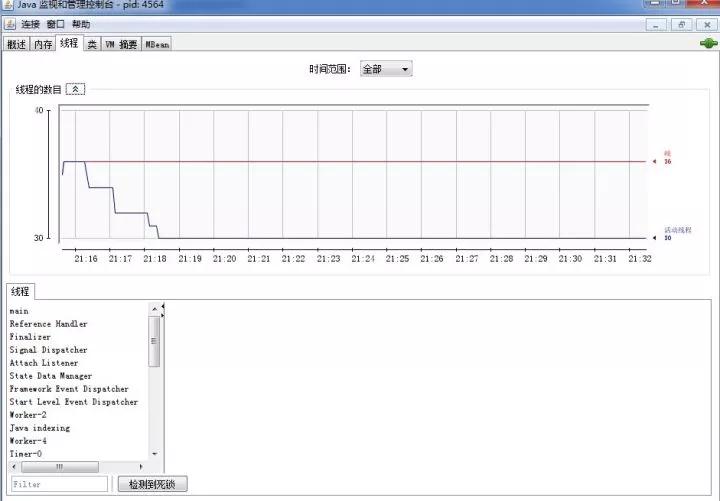
进入JConsole主界面，有“概述”、“内存”、“线程”、“类”、“VM摘要”和”Mbean”六个页签：



内存页签相当于jstat命令，用于监视收集器管理的虚拟机内存(Java堆和永久代)变化趋势，还可在详细信息栏观察全部GC执行的时间及次数。



线程页签



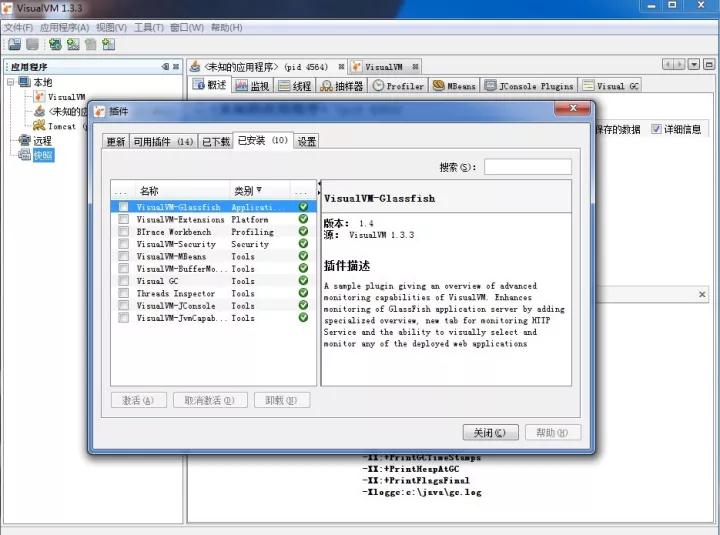
最后一个常用页签，VM页签，可清楚的了解显示指定的JVM参数及堆信息。



**2.VisualVM**

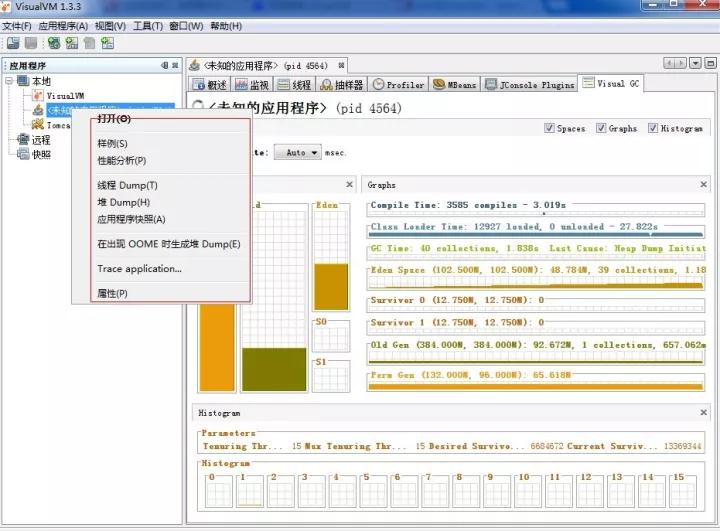
VisualVM是一个集成多个JDK命令行工具的可视化工具。VisualVM基于NetBeans平台开发，它具备了插件扩展功能的特性，通过插件的扩展，可用于显示虚拟机进程及进程的配置和环境信息(jps，jinfo)，监视应用程序的CPU、GC、堆、方法区及线程的信息(jstat、jstack)等。VisualVM在JDK/bin目录下。

安装插件： 工具- 插件





在VisualVM中生成dump文件：



**3.jprofiler**

<http://blog.csdn.net/java2000_wl/article/details/7012998>

## 5: 内存调优

JVM调优主要是针对内存管理方面的调优，包括控制各个代的大小，GC策略。由于GC开始垃圾回收时会挂起应用线程，严重影响了性能，调优的目是为了尽量降低GC所导致的应用线程暂停时间、 减少Full GC次数。

**代大小调优**

最关键参数：-Xms、 -Xmx 、-Xmn 、-XX:SurvivorRatio、-XX:MaxTenuringThreshold、-XX:PermSize、-XX:MaxPermSize

-Xms、 -Xmx 通常设置为相同的值，避免运行时要不断扩展JVM内存，这个值决定了JVM heap所能使用的最大内存。

-Xmn 决定了新生代空间的大小，新生代Eden、S0、S1三个区域的比率可以通过-XX:SurvivorRatio来控制(假如值为 4  表示：Eden:S0:S1 = 4:3:3 )

-XX:MaxTenuringThreshold 控制对象在经过多少次minor GC之后进入老年代，此参数只有在Serial 串行GC时有效。

-XX:PermSize、-XX:MaxPermSize 用来控制方法区的大小，通常设置为相同的值。

**1.避免新生代大小设置过小**

当新生代设置过小时，会产生两种比较明显的现象，一是minor GC次数频繁，二是可能导致 minor GC对象直接进入老年代。当老年代内存不足时，会触发Full GC。

**2.避免新生代设置过大**

新生代设置过大，会带来两个问题：一是老年大变小，可能导致Full  GC频繁执行；二是 minor GC 执行回收的时间大幅度增加。

**3.避免Survivor区过大或过小**

-XX:SurvivorRatio参数的值越大，就意味着Eden区域变大，minor GC次数会降低，但两块Survivor区域变小，如果超过Survivor区域内存大小的对象在minor GC后仍没被回收，则会直接进入老年代，

-XX:SurvivorRatio参数值设置过小，就意味着Eden区域变小，minor GC触发次数会增加，Survivor区域变大，意味着可以存储更多在minor GC后任存活的对象，避免其进入老年代。

**4.合理设置对象在新生代存活的周期**

新生代存活周期的值决定了新生代对象在经过多少次Minor GC后进入老年代。因此这个值要根据自己的应用来调优，Jvm参数上这个值对应的为-XX:MaxTenuringThreshold，默认值为15次。

**GC调优**

**1.GC搭配**

参见另一遍垃圾收集器

垃圾收集器

http://blog.csdn.net/java2000\_wl/article/details/8030172

**2.查看JVM参数**

命令行工具

http://blog.csdn.net/java2000\_wl/article/details/8042010

可视化工具

http://www.importnew.com/?p=22235

**3.GC日志参数**

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:c:\Java\gc.log

## 6: 对象访问

对象访问会涉及到Java栈、Java堆、方法区这三个内存区域。

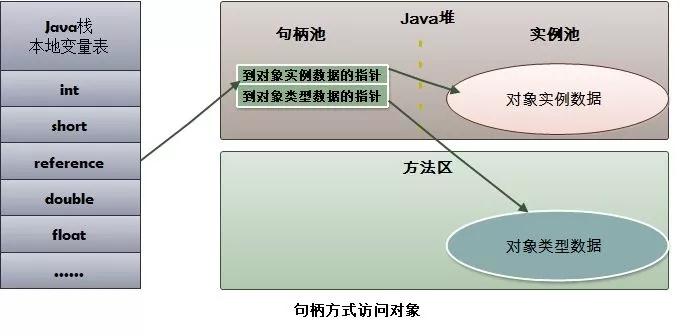
如下面这句代码：

Object objectRef = new Object();

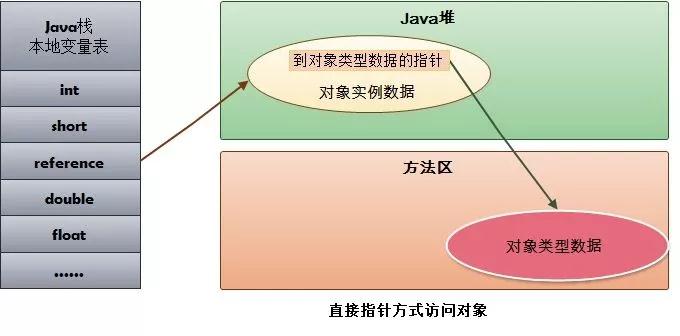
假设这句代码出现在方法体中，”Object objectRef” 这部分将会反映到Java栈的本地变量中，作为一个reference类型数据出现。而“new Object()”这部分将会反映到Java堆中，形成一块存储Object类型所有实例数据值的结构化内存，根据具体类型以及虚拟机实现的对象内存布局的不同，这块内存的长度是不固定。另外，在java堆中还必须包括能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些数据类型存储在方法区中。

reference类型在java虚拟机规范里面只规定了一个指向对象的引用地址，并没有定义这个引用应该通过那种方式去定位，访问到java堆中的对象位置，因此不同的虚拟机实现的访问方式可能不同,主流的方式有两种：使用句柄和直接指针。

**句柄访问方式**：java堆中将划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。



**指针访问方式**：reference变量中直接存储的就是对象的地址，而java堆对象一部分存储了对象实例数据，另外一部分存储了对象类型数据。

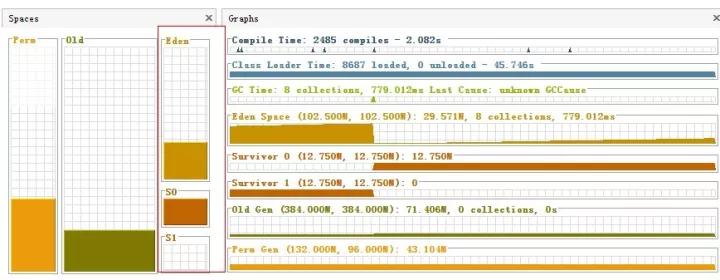


这两种访问对象的方式各有优势，使用句柄访问方式最大好处就是reference中存储的是稳定的句柄地址，在对象移动时只需要改变句柄中的实例数据指针，而reference不需要改变。使用指针访问方式最大好处就是速度快，它节省了一次指针定位的时间开销，就虚拟机而言，它使用的是第二种方式(直接指针访问)。

## 7: 对象内存分配与回收

**对象优先在Eden上分配**

大多数情况下，对象优先在新生代Eden区域中分配。当Eden内存区域没有足够的空间进行分配时，虚拟机将触发一次 Minor GC(新生代GC)。Minor GC期间虚拟机将Eden区域的对象移动到其中一块Survivor区域。



**大对象直接进入老年代**

所谓大对象是指需要大量连续空间的对象。虚拟机提供了一个XX:PretenureSizeThreshold参数，令大于这个值的对象直接在老年代中分配。

**长期存活的对象将进入老年代**

虚拟机采用分代收集的思想管理内存，那内存回收时就必须能识别那些对象该放到新生代，那些该到老年代中。为了做到这点，虚拟机为每个对象定义了一个对象年龄Age，每经过一次新生代GC后任然存活，将对象的年龄Age增加1岁，当年龄到一定程度（默认为15）时，将会被晋升到老年代中，对象晋升老年代的年龄限定值，可通过-XX:MaxTenuringThreshold来设置。

**Minor GC 和Full GC区别**

新生代GC(Minor GC)：指发生在新生代的垃圾收集动作，因为对象大多都具备朝生夕灭特性，所以Minor GC非常频繁，回收速度也比较快。

老年代GC(Major GC / Full GC)：指发生在老年代中的GC，出现Major GC后，经常会伴随至少一次的 Minor GC。Major GC的速度一般会比Minor GC慢10倍以上。

## 8: 查看 JVM 参数及值的命令行工具

查看JVM各个参数值方式

**1. HotSpot vm中的各个globals.hpp文件  查看jvm初始的默认值及参数**

globals.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/runtime/globals.hpp

globals\_extension.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/runtime/globals\_extension.hpp

c1\_globals.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/c1/c1\_globals.hpp

c1\_globals\_linux.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/linux/vm/c1\_globals\_linux.hpp

c1\_globals\_solaris.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/solaris/vm/c1\_globals\_solaris.hpp

c1\_globals\_sparc.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/sparc/vm/c1\_globals\_sparc.hpp

c1\_globals\_windows.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/windows/vm/c1\_globals\_windows.hpp

c1\_globals\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/x86/vm/c1\_globals\_x86.hpp

c2\_globals.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/opto/c2\_globals.hpp

c2\_globals\_linux.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/linux/vm/globals\_linux.hpp

c2\_globals\_solaris.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/solaris/vm/c2\_globals\_solaris.hpp

c2\_globals\_sparc.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/sparc/vm/c2\_globals\_sparc.hpp

c2\_globals\_windows.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/windows/vm/c2\_globals\_windows.hpp

c2\_globals\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/x86/vm/c2\_globals\_x86.hpp

g1\_globals.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/gc\_implementation/g1/g1\_globals.hpp

globals\_linux.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/linux/vm/globals\_linux.hpp

globals\_linux\_sparc.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/linux\_sparc/vm/globals\_linux\_sparc.hpp

globals\_linux\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/linux\_x86/vm/globals\_linux\_x86.hpp

globals\_linux\_zero.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/linux\_zero/vm/globals\_linux\_zero.hpp

globals\_solaris.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/solaris\_sparc/vm/globals\_solaris\_sparc.hpp

globals\_solaris\_sparc.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/solaris\_sparc/vm/globals\_solaris\_sparc.hpp

globals\_solaris\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/solaris\_x86/vm/globals\_solaris\_x86.hpp

globals\_sparc.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/sparc/vm/globals\_sparc.hpp

globals\_windows.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os/windows/vm/globals\_windows.hpp

globals\_windows\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/os\_cpu/windows\_x86/vm/globals\_windows\_x86.hpp

globals\_x86.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/x86/vm/globals\_x86.hpp

globals\_zero.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/zero/vm/globals\_zero.hpp

shark\_globals.hpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/shark/shark\_globals.hpp

shark\_globals\_zero.hpp

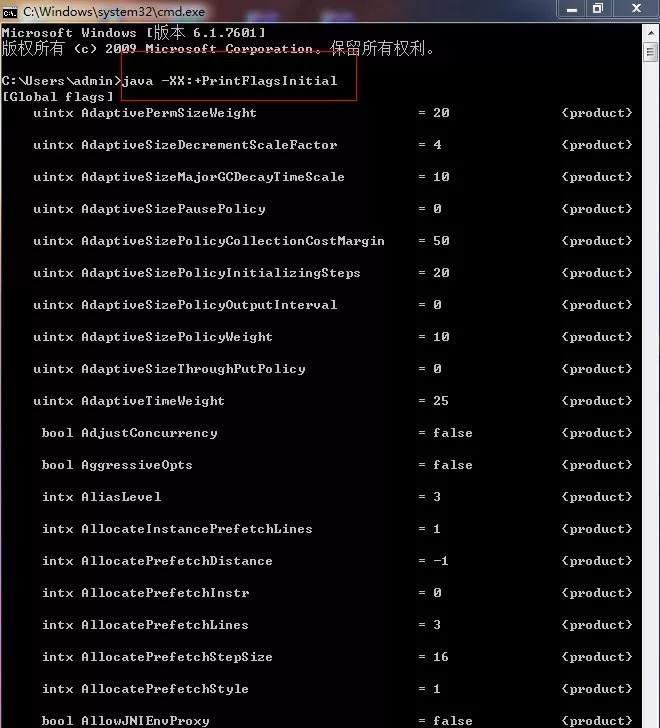
http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/cpu/zero/vm/shark\_globals\_zero.hpp

arguments.cpp

http://hg.openjdk.java.net/hsx/hsx20/master/file/f0f676c5a2c6/src/share/vm/runtime/arguments.cpp

**2.-XX:+PrintFlagsInitial参数**

显示所有可设置参数及默认值，可结合-XX:+PrintFlagsInitial与-XX:+PrintFlagsFinal对比设置前、设置后的差异，方便知道对那些参数做了调整。



**3.-XX:+PrintFlagsFinal参数**

可以获取到所有可设置参数及值(手动设置之后的值)，这个参数只能使用在Jdk6 update 21以上版本(包括该版本)。-XX:+PrintFlagsFinal参数的使用 与上面-XX:+PrintFlagsInitial 参数使用相同  Java -XX:+PrintFlagsFinal

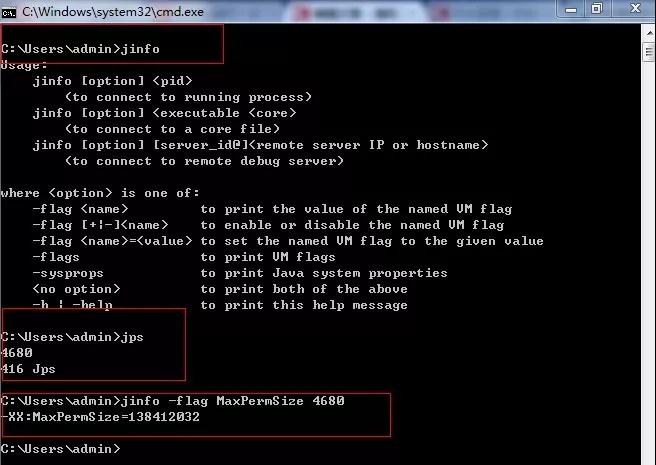
**4.使用 jinfo 命令 查看或设置某个参数的值,**

jinfo命令格式：

jinfo [option] <pid>

pid虚拟机进程id  可以通过  jps命令查看

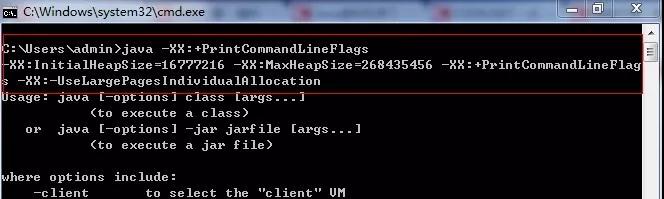
例子：查询MaxPermSize 参数的值



或直接使用 jinfo -flags pid 查看vm的所有设置参数

**5. -XX:+PrintCommandLineFlags参数**

显示出JVM初始化完毕后所有跟最初的默认值不同的参数及它们的值。



## 9: 对象引用强度

无论是通过计数算法判断对象的引用数量，还是通过根搜索算法判断对象引用链是否可达，判定对象是否存活都与“引用”相关。

引用主要分为 ：强引用(Strong Reference)、软引用(Soft Reference)、弱引用(Weak Reference)、虚引用(PhantomReference) 四种，引用的强度依次骤减。

**强引用：**

就是指在代码之中普遍存在的，类似：“Object objectRef = new Obejct”，这种引用，只要强引用还存在，永远不会被GC清理。

**软引用：**

用来描述一些还有用，但并非必须存在的对象，当Jvm内存不足时（内存溢出之前）会被回收，如果执行GC后，还是没有足够的空间，才会抛出内存溢出异常。

通过SoftReference类来实现软引用，SoftReference很适合用于实现缓存。另，当GC认为扫描的SoftReference不经常使用时，可会进行回收。

使用方法：

User user = new User();

SoftReference<Object> softReference  = new SoftReference<Object>(user);

softReference.get();

**弱引用**

弱引用也是用来描述一些还有用，但并非必须存在的对象，它的强度会被软引用弱些，被弱引用关联的对象，只能生存到下一次GC前，当GC工作时，无论内存是否足够，都会回收掉弱引用关联的对象。JDK通过WeakReference类来实现。

当获取时，可通过weakReference.get方法获取，可能返回null

可传入一个ReferenceQueue对象到WeakReference构造，当引用对象被表示为可回收时，isEnqueued返回true

User user = new User();

WeakReference<User> weakReference = new WeakReference<User>(user);

weakReference.get();

ReferenceQueue<User> referenceQueue = new ReferenceQueue<User>();

WeakReference<User> weakReference2 = new WeakReference<User>(user, referenceQueue);

//当引用对象被标识为可回收时  返回true,  即当user对象标识为可回收时，返回true

weakReference.isEnqueued();

**虚引用**

虚引用称为“幻影引用”，它是最弱的一种引用关系，一个对象是否有虚引用的存在，完全不会对生存时间构成影响。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被GC回收时收到一个系统通知。通过PhantomReference类实现。

值得注意的是：phantomReference.get方法永远返回null, 当user从内存中删除时，调用isEnqueued会返回true

User user = new User();

ReferenceQueue<User> referenceQueue = new ReferenceQueue<User>();

PhantomReference<User>  phantomReference = new PhantomReference<User>(user, referenceQueue);

//即当user对象标识为可回收时，返回true

System.out.println(phantomReference.isEnqueued());

//永远返回null

System.out.println(phantomReference.get());

**其他相关类：**

WeakCache weakCache  = new WeakCache();

SoftCache softCache = new SoftCache();

WeakHashMap weakHashMap  = new WeakHashMap();

当垃圾回收机制运行，扫描引用关系，GC会对这三种类型的引用进行不同的处理，简单来说，GC首先会判断所扫描到的引用是否为Reference类型，如果为Reference类型，且其所引用的对象无强引用，则认为该对象为相应的Reference类型，之后GC在垃圾回收时这些对象则根据Reference类型的不同进行相应处理。