# 1、Java与JVM介绍

JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

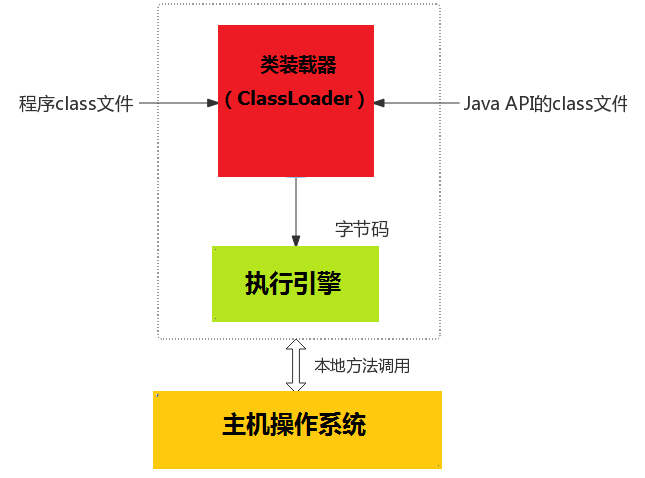
Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

## 1.1、Java体系结构包括四个独立的方面：

* Java程序设计语言
* Java class文件格式
* Java应用编程接口（Java API）
* Java虚拟机

Java面向网络的核心就是Java虚拟机，它支持Java面向网络体系结构三大支柱的所有方面：平台无关性，完全性和网络移动性。

## 1.2、Java程序的执行过程



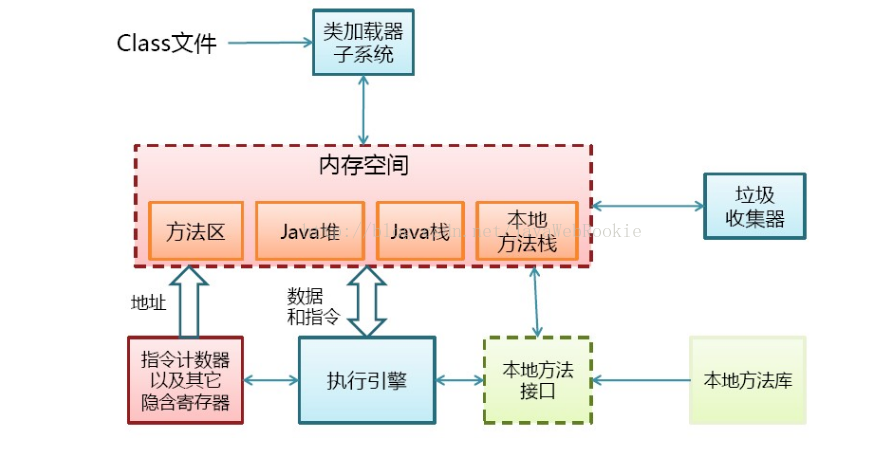
Java技术的核心就是Java虚拟机，因为所有的Java程序都在虚拟机上运行。Java程序的运行需要Java虚拟机、Java API和Java Class文件的配合。Java虚拟机实例负责运行一个Java程序。当启动一个Java程序时，一个虚拟机实例就诞生了。当程序结束，这个虚拟机实例也就消亡。

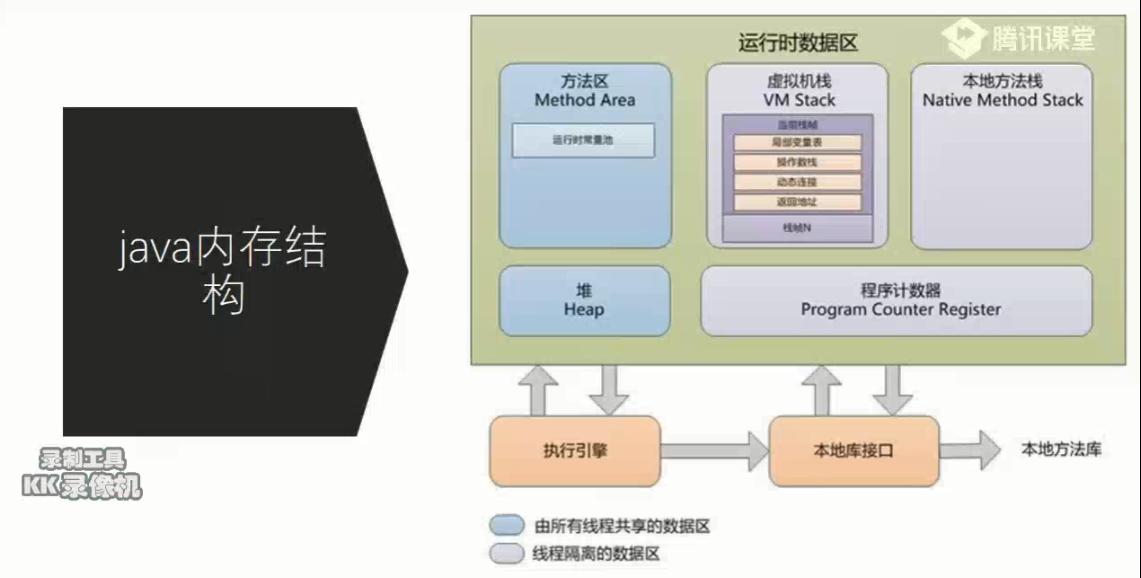
## 1.3、加载.class文件的方式

1. 从本地系统中直接加载
2. 通过网络下载.class文件（URLClassLoader）
3. 从zip， jar等归档文件中加载.class文件
4. 从专有数据库中提取.class文件
5. 将Java源文件动态编译为.class文件

**博客地址：**<http://blog.csdn.net/javawebrookie/article/details/51438059>

## 1.4、JVM内部结构





在java的内存分配中，java将内存分为：方法区、运行时常量池、堆、虚拟机栈、本地方法栈、程序计数器。其中，方法区（运行时常量池）和堆被所有线程共享，另外虚拟机栈、本地方法栈和程序计数器对于线程来说是隔离的，每个线程各有一份。方法区、堆、虚拟机栈、本地方法栈、程序计数器。每个内存区域都有各自的创建和销毁时间。

### 1.4.1、JVM与程序的生命周期

**• 在如下几种情况下， Java虚拟机将结束生命周期**

1. 执行了System.exit()方法

|  |
| --- |
| [java.lang.Object](mk:@MSITStore:E:\ImportantData\Learning\J_API\j2se6.chm::/j2se6/api/java/lang/Object.html)  **java.lang.System**  **public static void exit(int status)**  Terminates the currently running Java Virtual Machine. The argument serves as a status code; by convention, **a nonzero status code indicates abnormal termination.**  This method calls the exit method in class Runtime. This method never returns normally. |

1. 程序正常执行结束
2. 程序在执行过程中遇到了异常或错误而异常终止
3. 由于操作系统出现错误而导致Java虚拟机进程终止

### 1.4.1、类装载器（ClassLoader）

#### 1.4.2、类装载器的装载、链接与初始化

类装载器子系统除了要定位和导入二进制class文件外，还必须负责验证被导入类的正确性，为类变量分配并初始化内存，以及帮助解析符号引用。这些动作必须严格按以下顺序进行：  
　　（1）装载——查找并装载类型的二进制数据  
　　（2）连接——指向验证、准备、以及解析（可选）

　　　　●　验证：确保被导入类型的正确性（java可以自定义安全策略等）  
　　　　●　准备：为类变量分配内存，并将其初始化为默认值  
　　　　●　解析：把类型中的符号引用转换为直接引用  
　　（3）初始化——把类变量初始化为正确初始值

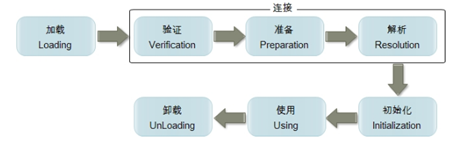


图1.3.2

#### 1.4.3、有两种类型的类加载器（父委托机制）

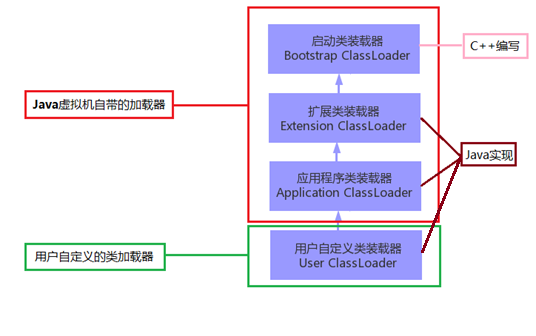
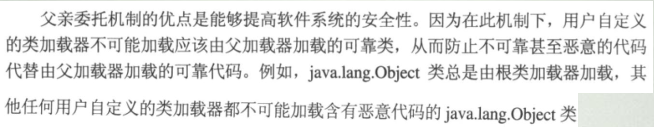


图1.3.3

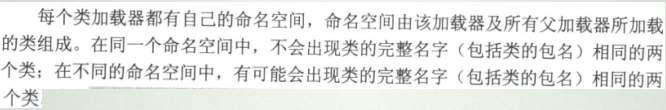
1. **Java**虚拟机自带的加载器  
   • 根类加载器（ Bootstrap【C++编写，无法在Java代码获取】）  
   • 扩展类加载器（ Extension【Java代码实现】）  
   • 系统类加载器（ System【AppClassLoader】）
2. 用户自定义的类加载器

• **java.lang.ClassLoader**的子类  
• 用户可以定制类的加载方式

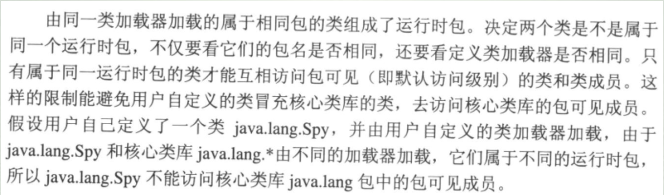
**类的父委托机制**



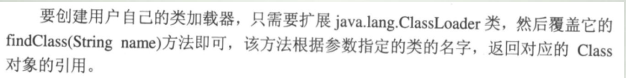
**命名空间**



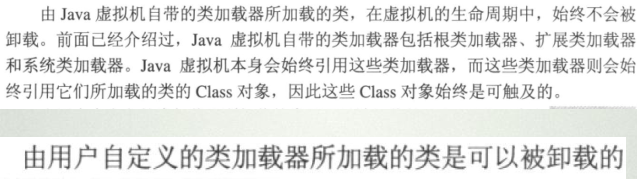
**运行时包**

****

**创建用户自定义加载器（N）**

****

**类的卸载**

****

#### 1.4.4、类的加载器与类

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

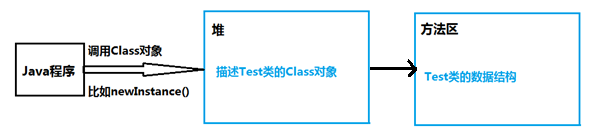


图1.3.4

类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟中的唯一性。说通俗一些，比较两个类是否“相等”，只有在两个类是由同一个类加载器的前提之下才有意义，否则，即使这两个类来源于同一个class文件，只要加载它的类加载器不同，那这两个类必定不相等。这里所指的“相等”包括代表类的Class对象的equal方法、isAssignableFrom()、isInstance()方法及instance关键字返回的结果。

1. 类加载器并不需要等到某个类被“首次主动使用”时再加载它。意思就是在使用之前类加载器已经加载了，只是等到我们进行main函数时才提示我们（比如类没有找到什么的）。
2. JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（ **LinkageError**错误）。

|  |
| --- |
| java.lang  Class LinkageError [java.lang.Object](mk:@MSITStore:E:\ImportantData\Learning\J_API\j2se6.chm::/j2se6/api/java/lang/Object.html)  [java.lang.Throwable](mk:@MSITStore:E:\ImportantData\Learning\J_API\j2se6.chm::/j2se6/api/java/lang/Throwable.html)  [java.lang.Error](mk:@MSITStore:E:\ImportantData\Learning\J_API\j2se6.chm::/j2se6/api/java/lang/Error.html)  **java.lang.LinkageError** |

1. 如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。
2. 类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

#### 1.4.5、类的验证（class文件检查器）

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

1. **类文件的结构检查（保证正确的内部结构）：**是通过**class文件检查器**保证装载的class文件内容有正确的内部结构，并且这些class文件互相间协调一致。Class文件检查器实现的安全目标之一就是程序的健壮性。如果某个有漏洞的编译器，产生了一个class文件，而这个class文件中包含了一个方法，这个方法的字节码中含有一条跳转到方法之外的指令，那么，一旦这个方法被调用，它将导致虚拟机的崩溃，所以，处于对健壮性的考虑，由虚拟机检验它装载的字节码的完整性非常重要。

第一趟扫描的目的就是保证这个字节序列正确的定义了一个class。它必须遵从Java的class文件的固定格式，这样它才能被编译成在方法区中的(基于实现的)内部数据结构。

1. **语义检查（比如字符串必须符合特定的语法规范）：**检查器对每个组成都分进行检查的目的之一，是为了确认每个方法描述符合特定的语法规则，格式正确的字符串。
2. **字节码验证（确保字节码流可以被Java虚拟机安全地执行）**：字节码流代表Java方法（包括静态方法和实例方法），它是由操作码单字节指令的组成序列，每一个操作数后都跟着一个或者多个操作数。
3. **符号引用的验证（必须检查被检测的class文件以外的其他类）：**可能引入装载的类，大多数Java虚拟机的实现采用延迟加载类的策略，就是用到了才加载。将符号引用替换为直接引用，就是一个类指向类，字段或者方法的指针或者偏移量。
4. **二进制兼容性的验证（确保相互引用之间协调一致）：**比如引用的jar包中的类之间方法引用，在Java虚拟机在验证类中的方法，检查在方法区内是否存在这个方法，如果不存在就抛出NoSuchMethodError错误。或者版本不兼容，比如有时低版本的引用到高版本可以会出错，但是Java的版本都是向上兼容的，在高版本引用低版本不会出现这种错误。

#### 1.4.6、类的准备

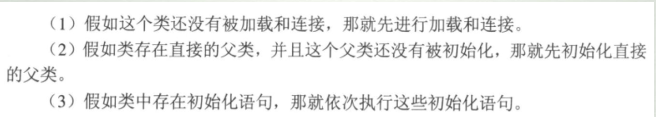
Java虚拟机为类的静态变量分配内存，并设计默认的初始值。比如int类型的类型静态变量a分配4个字节的内存空间，并赋予默认值为0（都知道int类型的默认值吧）。

#### 1.4.7、类的解析

Java虚拟机会把类的二进制数据中的符号引用替换为直接引用，比如在一个类A中引用到类B的方法，在类A的二进制数据中，包含了这个方法的符号引用，由方法的全名和相关的描述组成。在此阶段会把这个符号引用替换成一个指针，该指针指向类B中的方法在方法区的内存位置。

#### 1.4.8、类的初始化

Java虚拟机执行初始化语句是有先后顺序依次执行，而且是在加载和链接完成后（如果没有加载链接那就进行加载链接），再为类的静态变量赋值赋予初始值。在静态变量的声明中出进行初始化（就是int a=1;初始化值就是1），在静态代码块中进行初始化（int a;而赋值是在static方法里面进行初始化），如果没初始化的变量将保持默认值。



#### 1.5、类的使用方式可分为两种

– 主动使用

– 被动使用

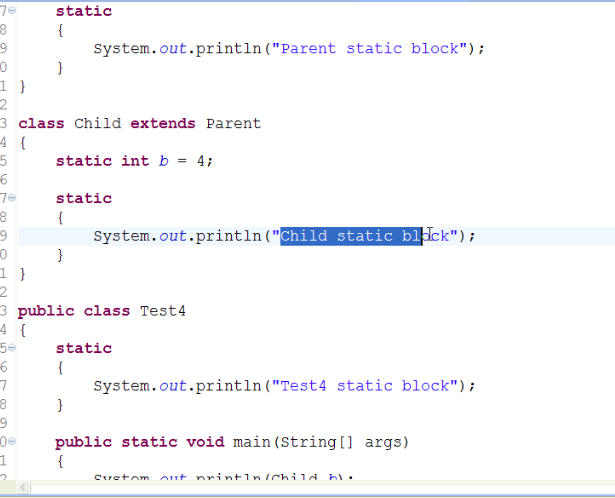
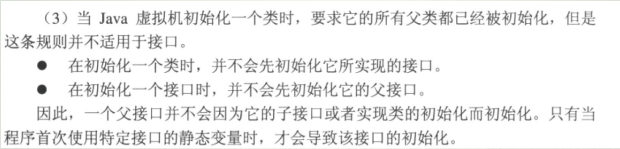
• 所有的Java虚拟机实现必须在每个类或接口被Java程序“首次主动使用”时才初始化它们。

**主动使用（ 六种）**

1. 创建类的实例（如new 一个对象）
2. 访问某个类或接口的静态变量，或者对该静态变量赋值（类.变量）
3. 调用类的静态方法（类.静态方法）
4. 反射（如Class.forName(“com.shengsiyuan.Test”)）
5. 初始化一个类的子类
6. Java虚拟机启动时被标明为启动类的类（JavaTest）

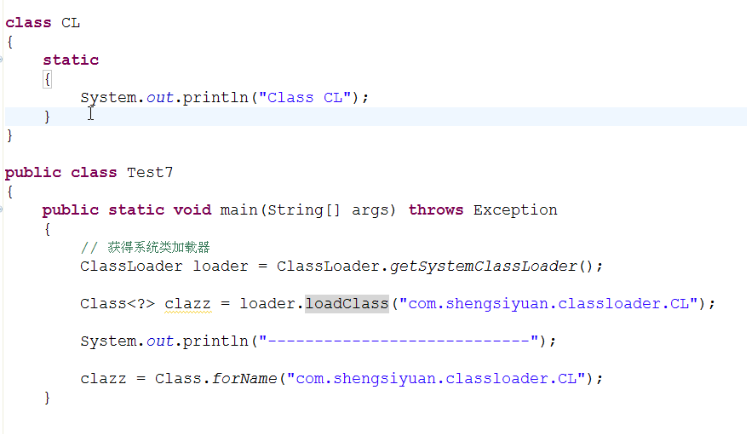
• 除了以上六种情况，其他使用Java类的方式都被看作是对类的被看作是对类的被动使用，都不会导致类的初始化。

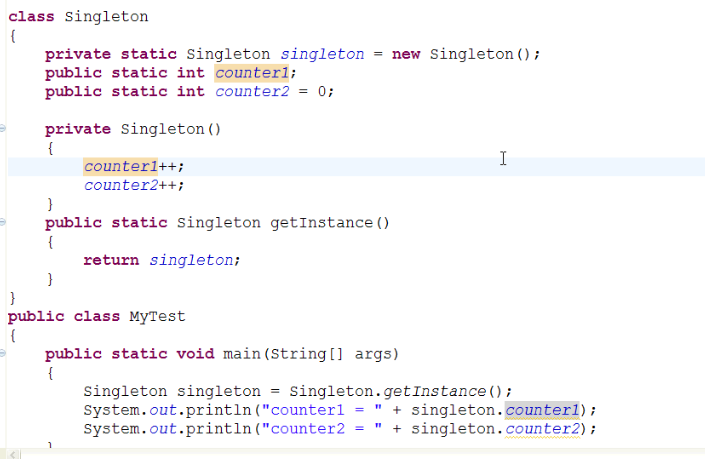
##### 1.5.8.1、类的初始化时机

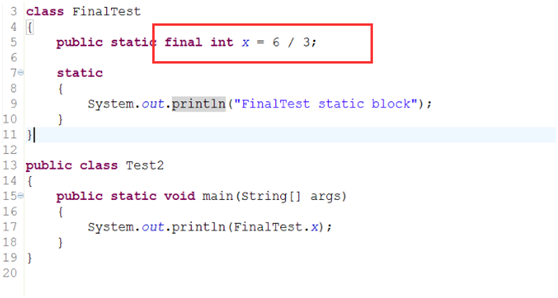


只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用。

调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。



****





### 1.4.5、方法区

方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，可见方法区域的重要性。同样，方法区域也是全局共享的，它在虚拟机启动时产生，在一定的条件下它也会被GC回收，当方法区域需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

在Sun JDK中这块区域对应的为PermanetGeneration，又称为持久代，默认为64M，可通过-XX:PermSize以及-XX:MaxPermSize来指定其大小。

方法区在一个jvm实例的内部，类型信息被存储在一个称为方法区的内存逻辑区中。类型信息是由类加载器在类加载时从类文件中提取出来的。类(静态)变量也存储在方法区中。

– 保存装载的类信息

* 类型的常量池 。
* 字段，方法信息 。
* 方法字节码 。

–通常和永久区(Perm)关联在一起

–通常和永久区(Perm)关联在一起

–JDK6时，String等常量信息置于方法 、JDK7时，已经移动到了堆

### 1.4.6、Java堆

Heap是JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收，Heap在32位的操作系统上最大为2G，在64位的操作系统上则没有限制，其大小通过-Xms和-Xmx来控制，-Xms为JVM启动时申请的最小Heap内存，默认为物理内存的1/64但小于1G，-Xmx为JVM可申请的最大Heap内存，默认为物理内存的1/4，默认当空余堆内存小于40%时，JVM会增大Heap的大小到-Xmx指定的大小，可通过-XX:MinHeapFreeRatio=来指定这个比例，当空余堆内存大于70%时，JVM会将Heap的大小往-Xms指定的大小调整，可通过-XX:MaxHeapFreeRatio=来指定这个比例，但对于运行系统而言，为了避免频繁的Heap Size的大小，通常都会将-Xms和-Xmx的值设成一样，因此这两个用于调整比例的参数通常是没用的。其实jvm中对于堆内存的分配、使用、管理、收集等有更为精巧的设计。 当堆中需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

堆是Java 虚拟机所管理的内存中最大的一块。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。此内存区域的唯一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。

–和程序开发密切相关。

–应用系统对象都保存在Java堆中 （new对象）。

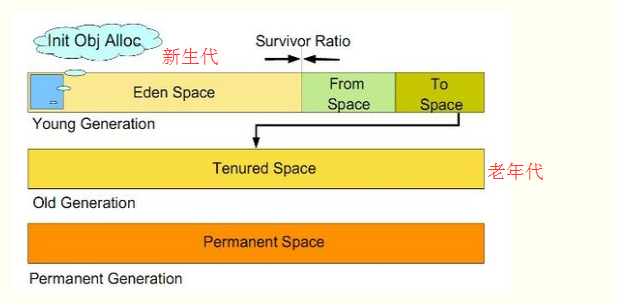
–所有线程共享Java堆 。

–对分代GC来说，堆也是分代的 。

–GC的主要工作区间 。

#### [Jvm 堆内存内部结构](http://www.cnblogs.com/zxf330301/articles/5252728.html)

http://www.cnblogs.com/zxf330301/articles/5252728.html



### 1.4.7、Java栈

JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈，JVM栈中存放的为当前线程中局部基本类型的变量（java中定义的八种基本类型：boolean、char、byte、short、int、long、float、double）、部分的返回结果以及Stack Frame，非基本类型的对象在JVM栈上仅存放一个指向堆上的地址，因此Java中基本类型的变量是值传递，而非基本类型的变量是引用传递，Sun JDK的实现中JVM栈的空间是在物理内存上分配的，而不是从堆上分配。   
由于JVM栈是线程私有的，因此其在内存分配上非常高效，并且当线程运行完毕后，这些内存也就被自动回收。当JVM栈的空间不足时，会抛出StackOverflowError的错误，在Sun JDK中可以通过-Xss来指定栈的大小

虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。

* + 线程私有
  + 栈由一系列帧组成（因此Java栈也叫做帧栈）
  + 帧保存一个方法的局部变量、操作数栈、常量池指针
  + 每一次方法调用创建一个帧，并压栈

### 1.4.8、本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法(该方法的实现由非java语言实现)服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与[数据结构](http://lib.csdn.net/base/datastructure)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（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。

与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

### 1.4.9、运行时常量池

存放的为类中的固定的常量信息、方法和Field的引用信息等，其空间从方法区域中分配。类或接口的常量池在该类的class文件被java虚拟机成功装载时分配。

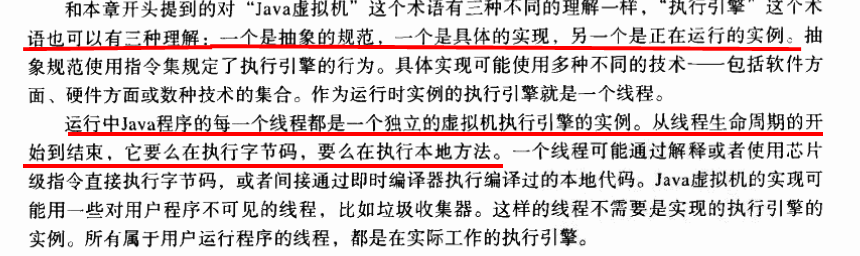
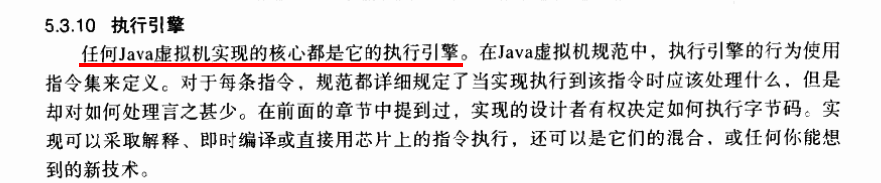
### 1.4.10、PC寄存器

程序计数器是一块很小的内存区域，主要作用是记录当前线程所执行的字节码的行号。字节码解释器工作时就是通过改变当前线程的程序计数器选取下一条字节码指令来工作的。任何分支，循环，方法调用，判断，异常处理，线程等待以及恢复线程，递归等等都是通过这个计数器来完成的。由于java多线程是通过交替线程轮流切换并分配处理器时间的方式来实现的，在任何一个确定的时间里，在处理器的一个内核只会执行一条线程中的指令。因此为了线程等待结束需要恢复到正确的位置执行，每条线程都会有一个独立的程序计数器来记录当前指令的行号。计数器之间相互独立互不影响，我们称这块内存为“线程私有”的内存。

如果所调用的方法为native的，则PC寄存器中不存储任何信息。

* 每个线程拥有一个PC寄存器。
* 在线程创建时创建 。
* 指向下一条指令的地址。
* 执行本地方法时，PC的值为undefined 。

### 1.4.11、执行引擎

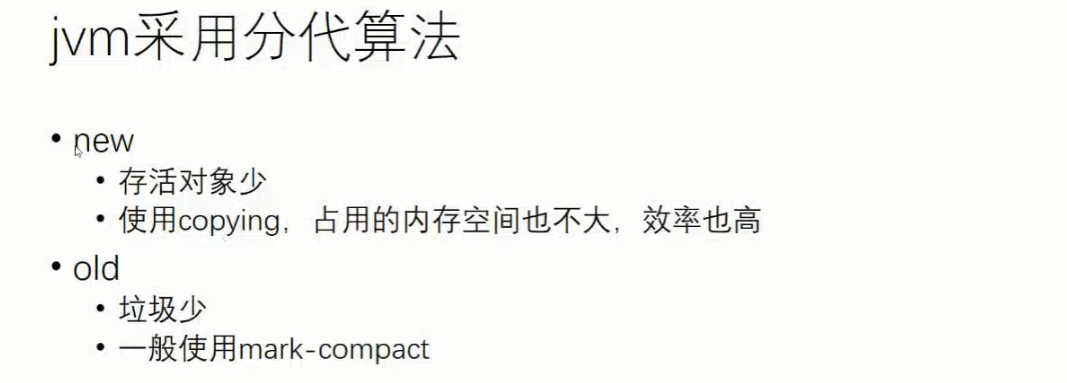


## 1.5、垃圾回收机制GC

 java  语言中一个显著的特点就是引入了java回收机制，是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在考虑内存管理。由于有个垃圾回收机制，java中的额对象不在有“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，有效的使用空闲的内存；

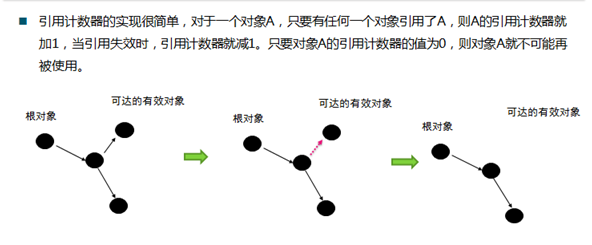
内存泄露：指该内存空间使用完毕后未回收，在不涉及复杂数据结构的一般情况下，java的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有是也将其称为“对象游离”；





### 1.5.1、老牌垃圾回收算法 （引用计算 ）

  引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。



### 1.5.2、标记-清除算法mark-sweep

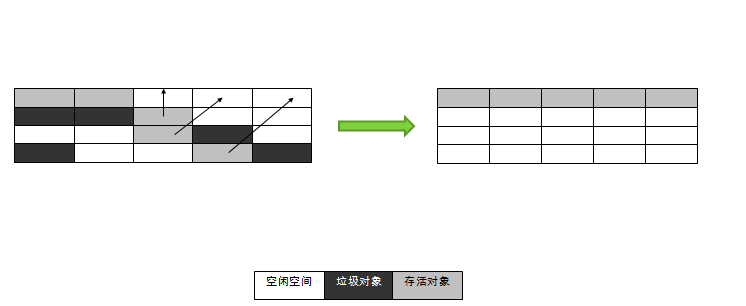
是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。



缺点：内存的不连续

### 1.5.3、标记-压缩算法mark-compact

适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。

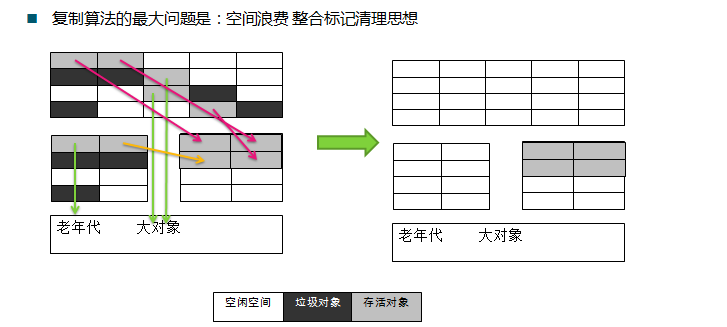
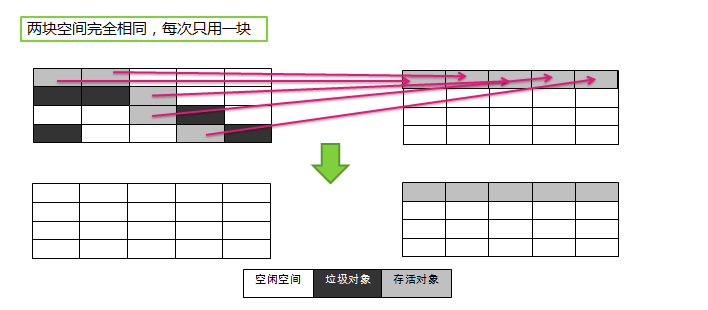


Copying效率低点

### 1.5.4、复制算法Copying

与标记-清除算法相比，复制算法是一种相对高效的回收方法 、不适用于存活对象较多的场合 如老年代 。

将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收 。



缺点：内存浪费

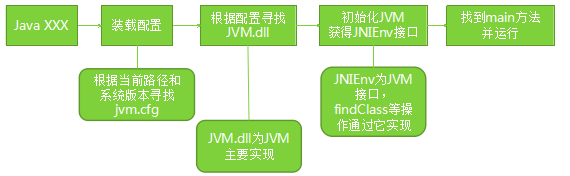
### 1.5.5、分代思想

* 依据对象的存活周期进行分类，短命对象归为新生代，长命对象归为老年代。
* 根据不同代的特点，选取合适的收集算法
  + 少量对象存活，适合复制算法
  + 大量对象存活，适合标记清理或者标记压缩

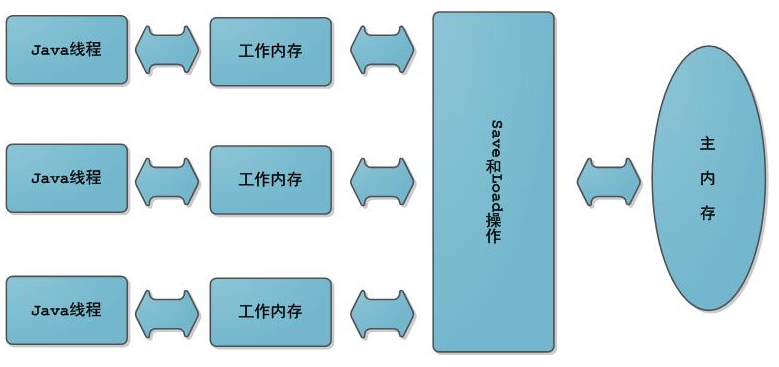
**可触及性**

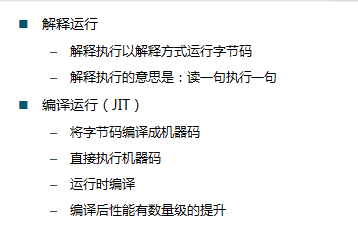
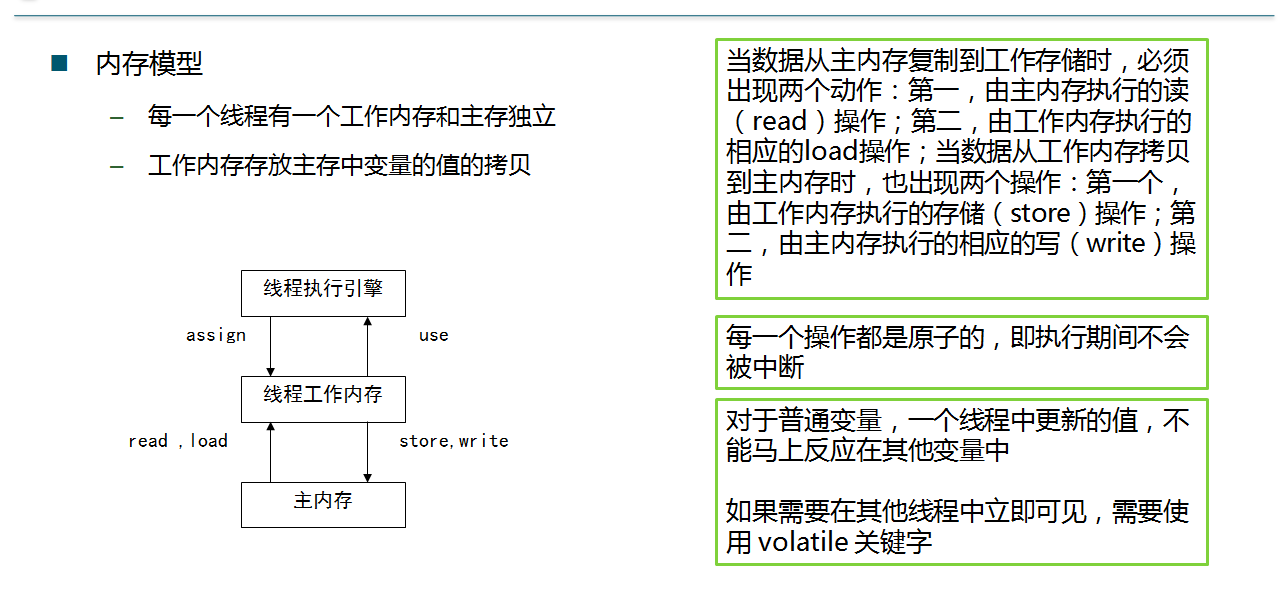
* 可触及的
  + 从根节点可以触及到这个对象
* 可复活的
  + 一旦所有引用被释放，就是可复活状态
  + 因为在finalize()中可能复活该对象
* 不可触及的
  + 在finalize()后，可能会进入不可触及状态
  + 不可触及的对象不可能复活
  + 可以回收
* 根
  + 栈中引用的对象
  + 方法区中静态成员或者常量引用的对象（全局对象）
  + JNI方法栈中引用对象

## 1.6、JVM启动流程



## 1.7、内存模型

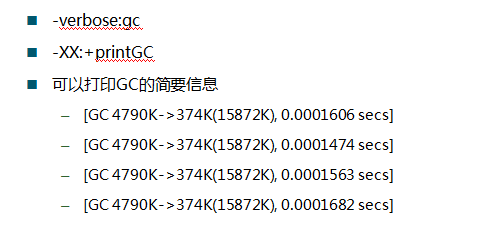




## 1.7、常用JVM配置参数

* Trace跟踪参数
* 堆的分配参数
* 栈的分配参数

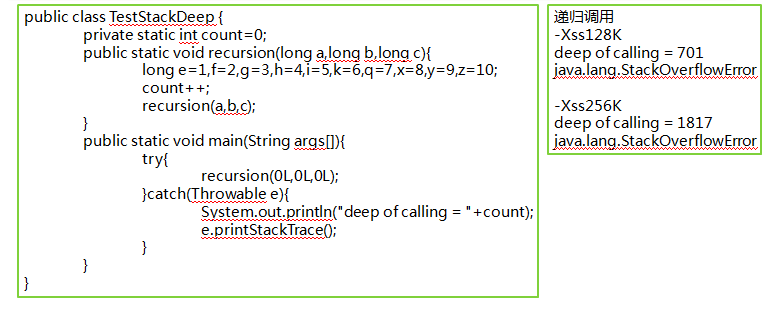
### Trace跟踪参数



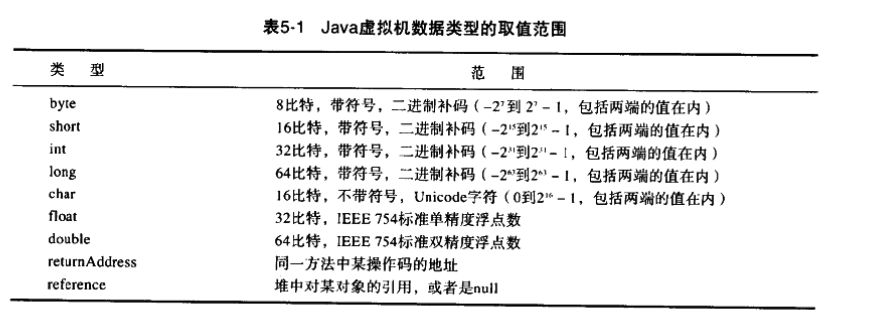
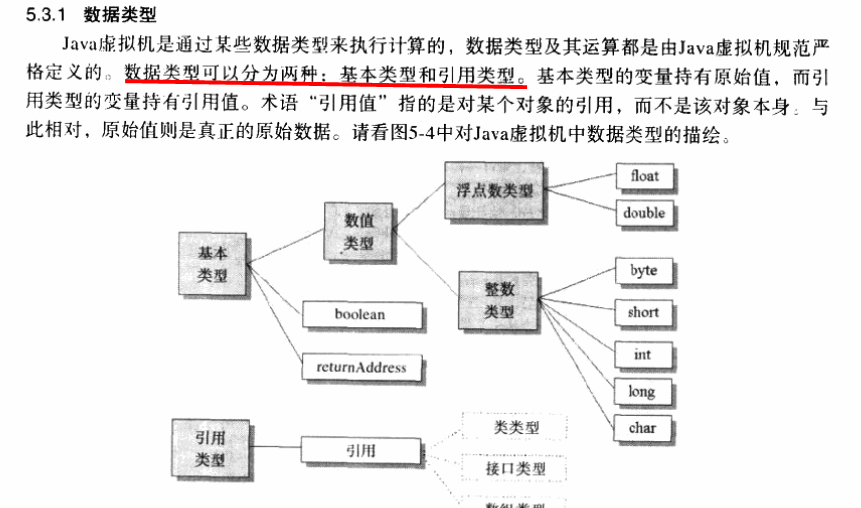
### 堆的分配参数



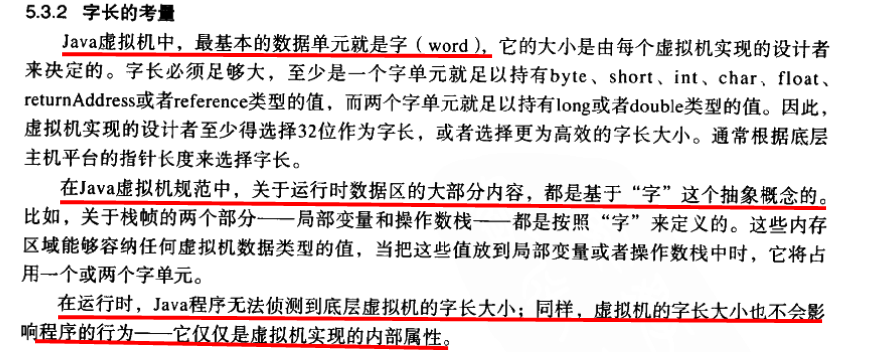
### 栈大小分配



## 1.8、数据类型

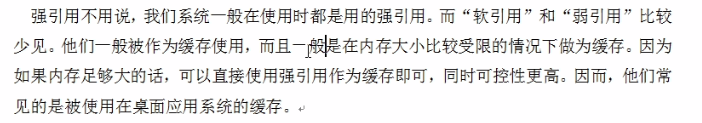
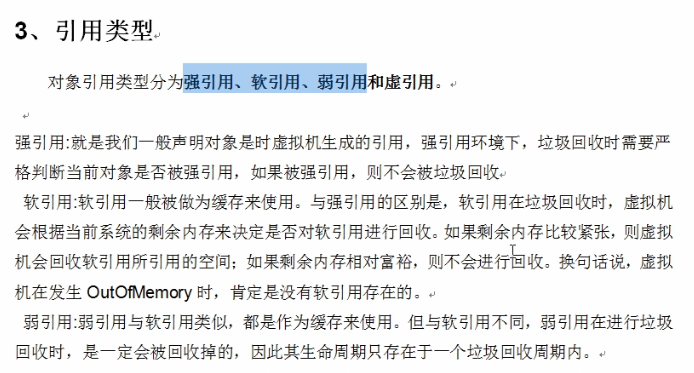
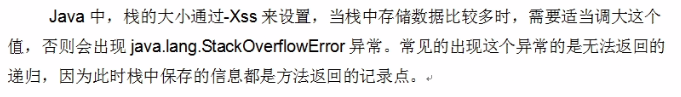
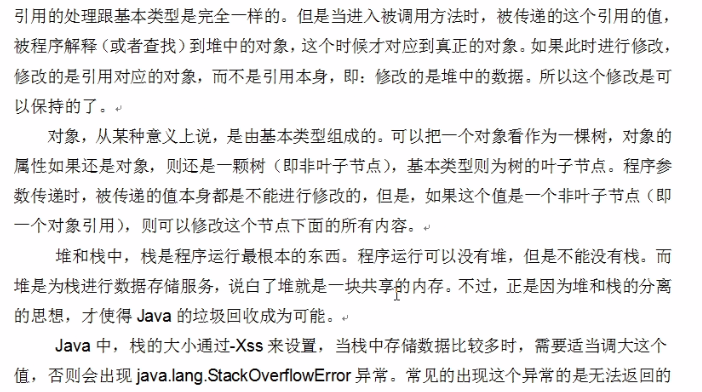
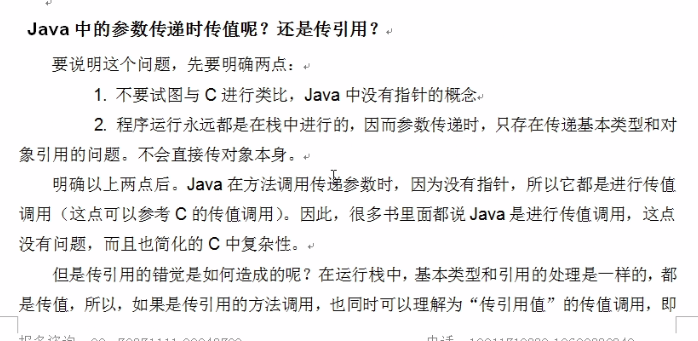
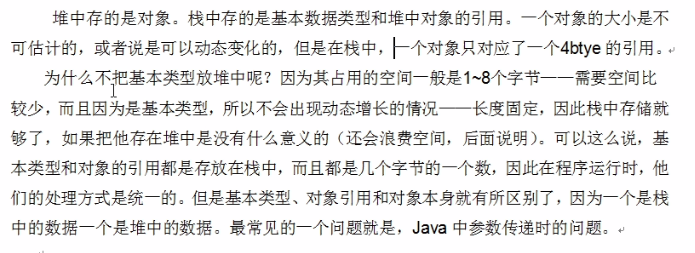
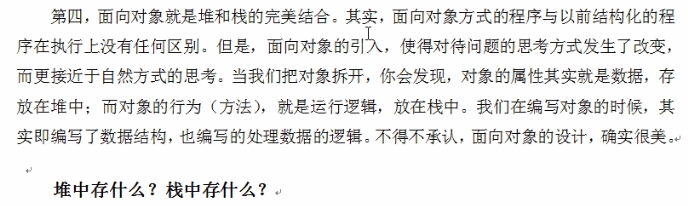
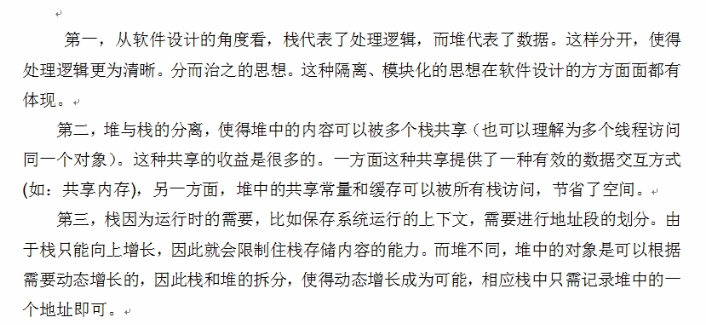
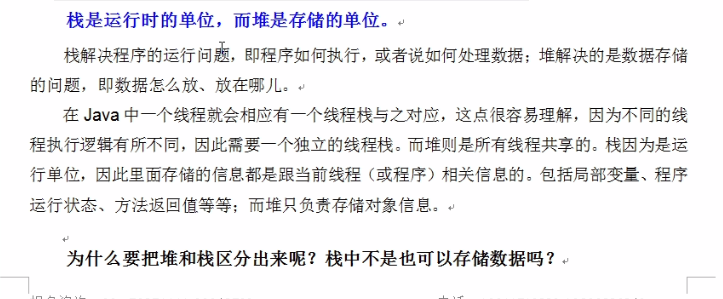


## 1.9、字长考量



# JVM调优





## JVM的垃圾收集器

http://www.cnblogs.com/lushilin/p/6140507.html



**Serial(年轻代）**

**ParNew(年轻代）**

**Paralle Scavenge(年轻代）**

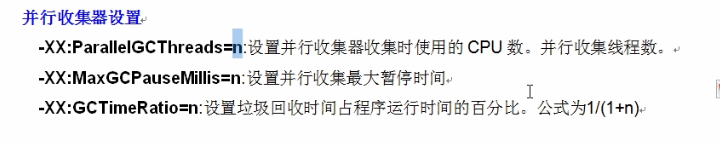
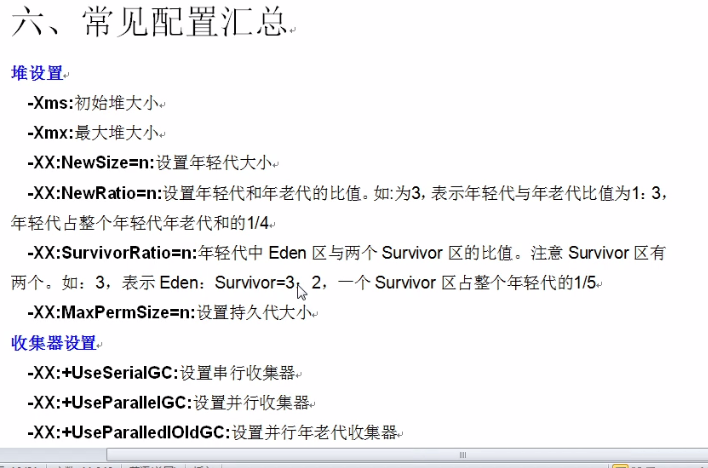
**Serial Old(年老代）**

**Parallel Old(年老代）**

**CMS(Concurrent Mark Sweep年老代）**

**G1**

## 参数配置



# 感谢信与参考文献

**主要参考文献：**深入Java虚拟机(原书第二版清晰版)

特别感谢网络老师、和网上提供资料的博主，以后有学到新的知识我都会更新博客，算是一个成长吧。如果博文、文档有错误请理解，因为整理文档的时候可能疏漏。整理这个文档是为了容易理解吧，也为了加强我的理解吧。

**博客地址：**<http://blog.csdn.net/javawebrookie/article/details/51438059>