[一． Jvm结构图](#_Toc5949)

[1. JVM的解释](#_Toc80)

[2. 结构图介绍](#_Toc20164)

[二． 加载器](#_Toc9616)

[1.加载器结构图](#_Toc32042)

[2.加载过程及原理](#_Toc15121)

[1. 加载过程](#_Toc26925)

[2. 原理](#_Toc17812)

[三．虚拟机栈](#_Toc24991)

[四．虚拟机栈帧](#_Toc2414)

[五．堆结构](#_Toc12066)

[1. 结构说明](#_Toc29647)

[六．方法区结构](#_Toc7383)

[1.方法区的介绍](#_Toc13126)

[2. 永久带和方法区关系](#_Toc5975)

[七．Class文件常量池](#_Toc8508)

[1.常量池构造](#_Toc7349)

[2. 常量池所存信息](#_Toc27779)

[八．运行时常量池](#_Toc19805)

[1.运行时常量池与class常量池](#_Toc13616)

[九．程序计数器](#_Toc11525)

[1.什么是程序计数器](#_Toc3149)

[2.程序计数器有两个作用](#_Toc23565)

[十．Jvm常见内存溢出错误](#_Toc7730)

[1. 栈溢出(StackOverflowError)](#_Toc5381)

[2. 堆溢出(OutOfMemoryError:java heap space)](#_Toc17534)

[3. 持久带溢出(OutOfMemoryError: PermGen space)](#_Toc22314)

[4.OutOfMemoryError:unable to create native thread](#_Toc25868)

[二．不同方式配置不同作用域的jvm的堆**栈内存！**](#_Toc11811)

[1. 设置eclipse中jvm堆栈内存](#_Toc27990)

[2. Tomcat内存设置](#_Toc18498)

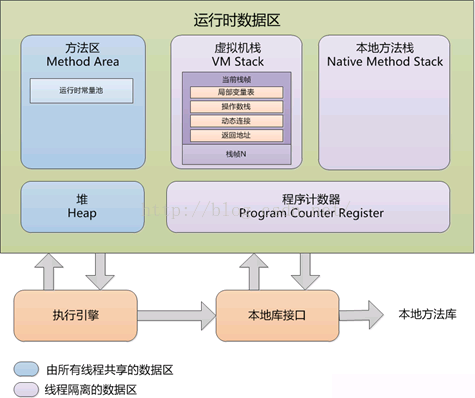
[三． 程序启动jvm内存分配过程案例](#_Toc30242)

[十一．基础准备](#_Toc22537)

# http://www.itnose.net/st/6707336.html

# Jvm结构图

http://blog.csdn.net/chjttony/article/category/1239947



## JVM的解释

JVM是Java Virtual Machine（Java[虚拟机](http://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)）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](http://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80)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](http://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E4%BB%A3%E7%A0%81" \t "http://baike.baidu.com/_blank)。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](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E7%A8%8B%E5%BA%8F" \t "http://baike.baidu.com/_blank)只需生成在Java虚拟机上运行的目标代码（[字节码](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "http://baike.baidu.com/_blank)），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](http://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4" \t "http://baike.baidu.com/_blank)执行。这就是Java的能够“一次编译，到处运行”的原因。

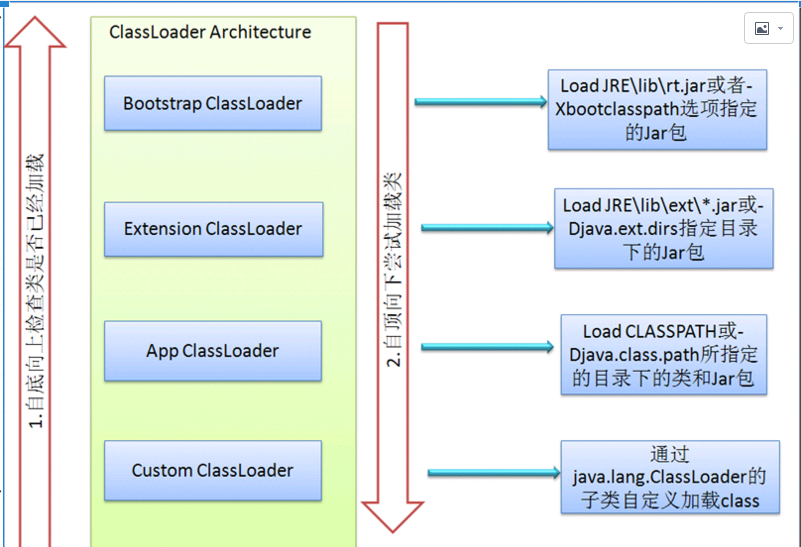
## 结构图介绍

1. method area :
   1. 方法区是所有线程共享的，它是在JVM启动的时候创建的。它保存所有被JVM加载的类和接口的运行时常量池，成员变量以及方法的信息，静态变量以及方法的字节码。JVM的提供者可以通过不同的方式来实现方法区。在Oracle 的HotSpot JVM里，方法区被称为永久区或者永久代（PermGen）。是否对方法区进行垃圾回收对JVM的实现是可选的。
2. VM stack :
   1. 每个线程启动的时候，都会创建一个JVM线程栈。它是用来保存栈帧的。JVM只会在JVM线程栈上对栈帧进行push和pop的操作。如果出现了异常，线程栈跟踪信息的每一行都代表一个栈帧立的信息，这些信息是通过类似于printStackTrace()这样的方法来展示的。
3. Native method stack：
   1. 供用非Java语言实现的本地方法的堆栈。换句话说，它是用来调用通过JNI(Java Native Interface Java本地接口）调用的C/C++代码。根据具体的语言，一个C堆栈或者C++堆栈会被创建。
4. heap :
   1. 用来保存实例或者对象的空间，而且它是垃圾回收的主要目标。当讨论类似于JVM性能之类的问题时，它经常会被提及。JVM提供者可以决定怎么来配置堆空间，
5. program counter register :

a) 每个线程启动的时候，都会创建一个PC（Program Counter，程序计数器）寄存 器。PC寄存器里保存有当前正在执行的JVM指令的地址。

# 加载器

## 1.加载器结构图

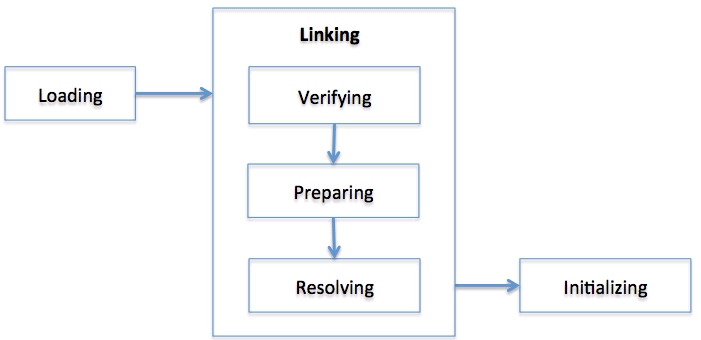


**说明：**

每个类装载器都有一个自己的命名空间用来保存已装载的类。当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name)进行搜索来检测这个类是否已经被加载了。如果两个类的全局限定名是一样的，但是如果命名空间不一样的话，那么它们还是不同的类。不同的命名空间表示class被不同的类装载器装载。

## 2.加载过程及原理

## 加载过程



Loading：类的信息从文件中获取并且载入到JVM的内存里。

Verifying：检查读入的结构是否符合Java语言规范以及JVM规范的描述。这是类装载中最 复杂的过程，并且花费的时间也是最长的。并且JVM TCK工具的大部分场景的 用例也用来测试在装载错误的类的时候是否会出现错误。

Preparing:分配一个结构用来存储类信息，这个结构中包含了类中定义的成员变量，方法和 接口的信息。

Resolving：把这个类的常量池中的所有的符号引用改变成直接引用。

Initializing：把类中的变量初始化成合适的值。执行静态初始化程序，把静态变量初始化成 指定的值。

**符号引用直接引用说明：**

char b='a';

这就是在内存中的常量池中一个字符，它的引用名为b,转换为的Ascll码为:97

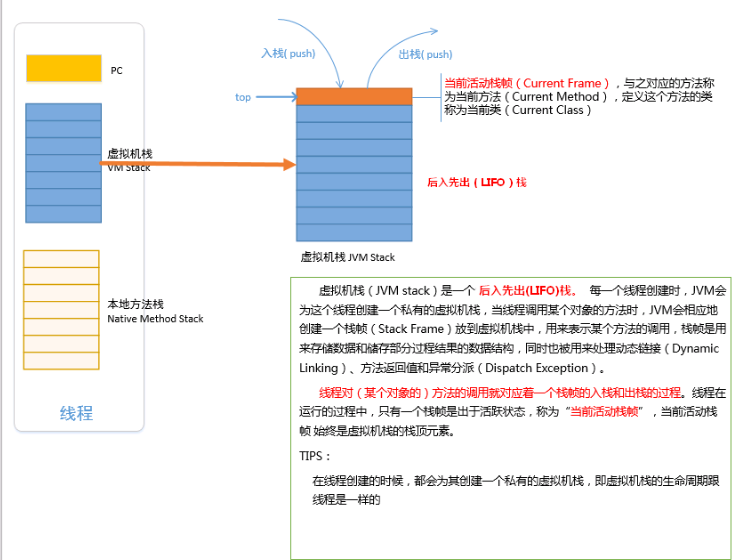
这里说的97就是字符a在常量池中的直接引用。

## 原理

* 1. **双亲委托原理**

ClassLoader使用的是双亲委托模型来搜索类的，每个ClassLoader实例都有一个父类加载器的引用（不是继承的关系，而是用组合的方式来复用父类的代码。），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但可以用作其它ClassLoader实例的的父类加载器。当一个ClassLoader实例需要加载某个类时，它会试图亲自搜索某个类之前，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器Bootstrap ClassLoader试图加载，如果没加载到，则把任务转交给Extension ClassLoader试图加载，如果也没加载到，则转交给App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等URL中加载该类。如果它们都没有加载到这个类时，则抛出ClassNotFoundException异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的Class实例对象。

# 三．虚拟机栈



该区域也是线程私有的，它的生命周期也与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧，栈它是用于支持续虚拟机进行方法调用和方法执行的**[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/ns_code/article/details/_blank)**。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入了方法表的Code属性之中。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

线程栈的大小可以通过设置JVM的-xss参数进行配置，32位系统下，一般默认的大小是512K

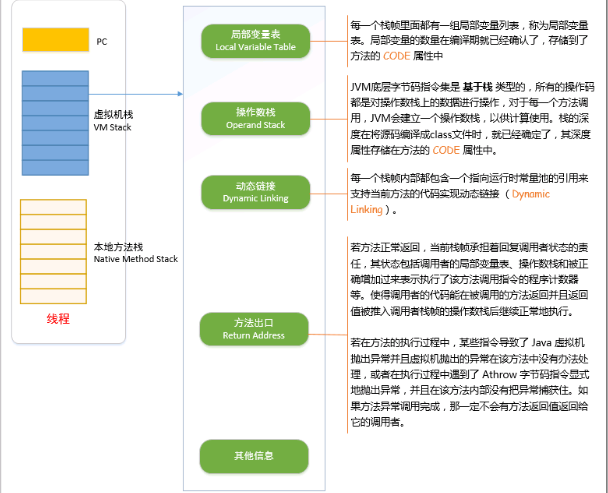
在Java虚拟机规范中，对这个区域规定了两种异常情况：

    1、如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。

   2、如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError异常。

 这两种情况存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。在单线程的操作中，无论是由于栈帧太大，还是虚拟机栈空间太小，当栈空间无法分配时，虚拟机抛出的都是StackOverflowError异常，而不会得到OutOfMemoryError异常。而在多线程环境下，则会抛出OutOfMemoryError异常。

# 四．虚拟机栈帧



1. **定义**
2. 栈帧（Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态 链接（Dynamic Linking）、方法返回值和异常分派（Dispatch Exception）。

B)栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完 成（抛出了在方法内未被捕获的异常）都算作方法结束。栈帧的存储空间分配在Java 虚拟机栈之中，每一个栈帧都有自己的局部变量表、操作数栈和指向当前方法所属的 类的运行时常量池的引用。

1. **补充**
2. 在一条线程之中，只有目前正在执行的那个方法的栈帧是活动的。这个栈帧就被称为 是当前栈帧（Current Frame），这个栈帧对应的方法就被称为是当前方法 （Current Method），定义这个方法的类就称作当前类（Current Class）。对局部变 量表和操作数栈的各种操作，通常都指的是对当前栈帧的对局部变量表和操作数栈进 行的操作。

B)如果当前方法调用了其他方法，或者当前方法执行结束，那这个方法的栈帧就不再是 当前栈帧了。新的方法被调用，一个新的栈帧也会随之而创建，并且随着程序

控制权移交到新的方法而成为新的当前栈帧。当方法返回的之际，当前栈帧会传回此方 法的执行结果给前一个栈帧，在方法返回之后，当前栈帧就随之被丢弃，前一个栈帧就 重新成为当前栈帧了。

**3.各部分信息的作用和数据结构。**

1. 局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量，其中存放的数据的类型是编译期可知的各种基本数据类型、对象引用（reference）和returnAddress类型（它指向了一条字节码指令的地址）。局部变量表所需的内存空间在编译期间完成分配，即在Java程序被编译成Class文件时，就确定了所需分配的最大局部变量表的容量。当进入一个方法时，这个方法需要在栈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。

局部变量表的容量以变量槽（Slot）为最小单位。在虚拟机规范中并没有明确指明一个 Slot应占用的内存空间大小（允许其随着处理器、操作系统或虚拟机的不同而发生变 化），一个Slot可以存放一个32位以内的数据类型：boolean、byte、char、short、int、float,reference和returnAddresss。reference是对象的引用类型，returnAddress是为字 节指令服务的，它执行了一条字节码指令的地址。对于64位的数据类型（long和 double），虚拟机会以高位在前的方式为其分配两个连续的Slot空间。

  虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从0开始到局 部变量表最大的Slot数量，对于32位数据类型的变量，索引n代表第n个 Slot，对于64位的，索引n代表第n和第n+1两个Slot。

     在方法执行时，虚拟机是使用局部变量表来完成参数值到参数变量列表的传 递过程的，如果是实例方法（非static），则局部变量表中的第0位索引的 Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字 “this”来访问这个隐含的参数。其余参数则按照参数表的顺序来排列，占 用从1开始的局部变量Slot，参数表分配完毕后，再根据方法体内部定义的 变量顺序和作用域分配其余的Slot。

     局部变量表中的Slot是可重用的，方法体中定义的变量，作用域并不一 定会覆盖整个方法体，如果当前字节码PC计数器的值已经超过了某个变量 的作用域，那么这个变量对应的Slot就可以交给其他变量使用。这样的设计 不仅仅是为了节省空间，在某些情况下Slot的复用会直接影响到系统的而垃 圾收集行为。

1. 操作数栈

  操作数栈又常被称为操作栈，操作数栈的最大深度也是在编译的时候就确定 了。32位数据类型所占的栈容量为1,64位数据类型所占的栈容量为2。当一 个方法开始执行时，它的操作栈是空的，在方法的执行过程中，会有各种字 节码指令（比如：加操作、赋值元算等）向操作栈中写入和提取内容，也就 是入栈和出栈操作。

     Java虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈” 就是操作数栈。因此我们也称Java虚拟机是基于栈的，这点不同于**[Android](http://lib.csdn.net/base/android" \o "Android知识库" \t "http://blog.csdn.net/ns_code/article/details/_blank)** 虚拟机，**[android](http://lib.csdn.net/base/android" \o "Android知识库" \t "http://blog.csdn.net/ns_code/article/details/_blank)**虚拟机是基于寄存器的。

    基于栈的指令集最主要的优点是可移植性强，主要的缺点是执行速度相对会 慢些；而由于寄存器由硬件直接提供，所以基于寄存器指令集最主要的优点 是执行速度快，主要的缺点是可移植性差。

1. 动态连接

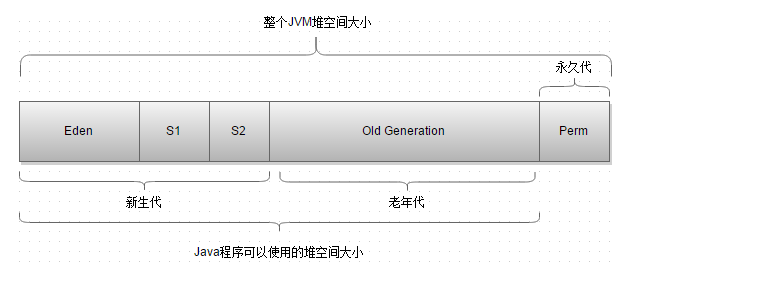
每个栈帧都包含一个指向运行时常量池（在方法区中，后面介绍）中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。Class文件的常量池中存在有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用，一部分会在类加载阶段或第一次使用的时候转化为直接引用（如final、static域等），称为静态解析，另一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。

1. 方法返回地址

当一个方法被执行后，有两种方式退出该方法：执行引擎遇到了任意一个方法返回的字节码指令或遇到了异常，并且该异常没有在方法体内得到处理。无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行。方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的PC计数器的值就可以作为返回地址，栈帧中很可能保存了这个计数器值，而方法异常退出时，返回地址是要通过异常处理器来确定的，栈帧中一般不会保存这部分信息。

 方法退出的过程实际上等同于把当前栈帧出站，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，如果有返回值，则把它压入调用者栈帧的操作数栈中，调整PC计数器的值以指向方法调用指令后面的一条指令。

# 五．堆结构



## 结构说明

A)上中，刻画了Java程序运行时的堆空间,可以简述成如下2条

1.JVM中堆空间可以分成三个大区，新生代、老年代、永久代

2.新生代可以划分为三个区，Eden区，两个幸存区

B)主要存储new关键字创建的对象和常量

**更多关于堆的知识将在GC垃圾回收中讲到**

# 六．方法区结构



## 1.方法区的介绍

方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。[Java虚拟机规范Java SE 7 Edition](http://www.amazon.co.uk/Virtual-Machine-Specification-Edition-Series/dp/0133260445)清楚地表示：“虽然方法区域在逻辑上是堆的一部分，但简单的实现可能选择不要垃圾收集或压缩它。

在Class文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池，用来存储编译期间生成的字面量和符号引用。

在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法。

## 永久带和方法区关系

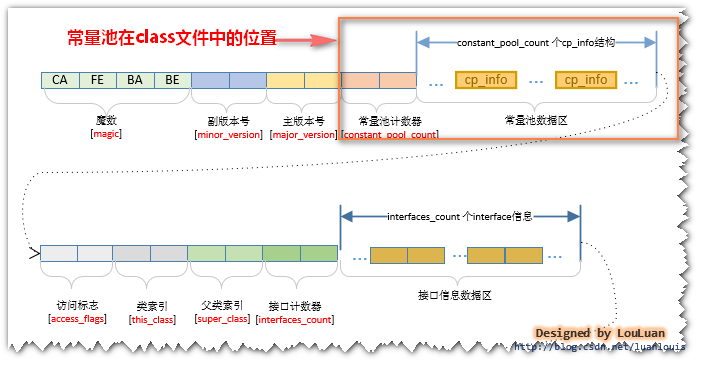
《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。同时，大多数用的JVM都是Sun公司的HotSpot。在HotSpot上把GC分代收集扩展至方法区，或者说使用永久代来实现方法区。

虽然可以牵强的解释这种将方法区和永久带等同对待观点。但最终方法区和永久带还是不同的。一个是标准一个是实现。这就相当于你将java中的接口和接口的实现等同对待了一样。同时，这种牵强的解释也仅仅是在HotSpot虚拟机上才能勉强成立。其他的虚拟机实现并没有永久带这一说法。   
有人说，HotSpot之所以用永久带来实现方法区是因为这样可以不必专门为方法区编写一套内存管理的代码。

.

# 七．Class文件常量池

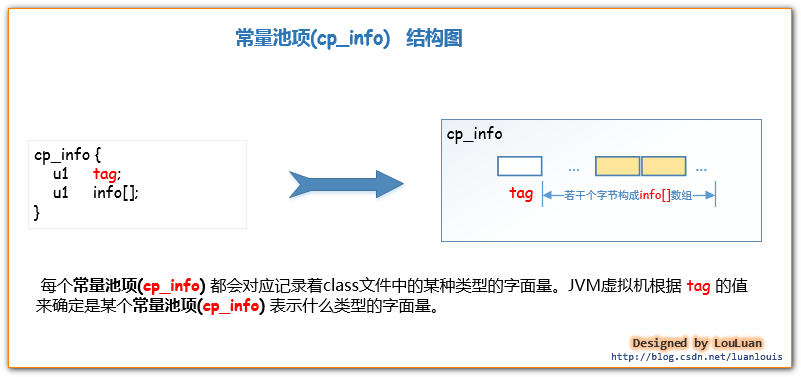
## 1.常量池构造



在Class文件结构中，最头的4个字节用于存储魔数Magic Number，用于确定一个文件是否能被JVM接受，再接着4个字节用于存储版本号，前2个字节存储次版本号，后2个存储主版本号，再接着是用于存放常量的常量池，由于常量的数量是不固定的，所以常量池的入口放置一个U2类型的数据(constant\_pool\_count)存储常量池容量计数值，如下图



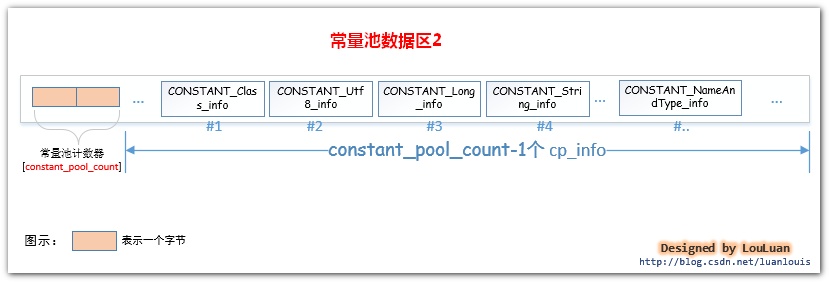
看看常量池项的具体结构



 JVM虚拟机规定了不同的tag值和不同类型的字面量对应关系如下：



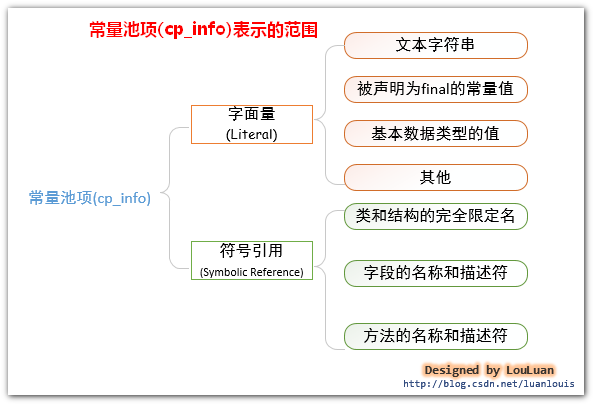
定义类型后的常量池如下图



以上是常量池的结构以及基本信息，下面看看常量池存的什么东西

## 常量池所存信息

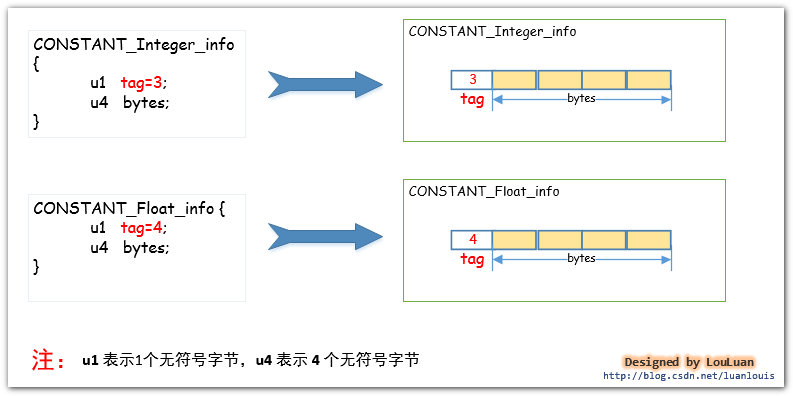
下面试常量池存储信息分类



接下来看看各种封装类型在常量池如何存储

**A) int和float数据类型的常量在常量池中是怎样表示和存储的？**

Java语言规范规定了 int类型和Float 类型的数据类型占用 4 个字节的空间。那么存在于class字节码文件中的该类型的常量是如何存储的呢？相应地，在常量池中，将 int和Float类型的常量分别使用CONSTANT\_Integer\_info和 Constant\_float\_info表示，他们的结构如下所示：



举例：建下面的类 IntAndFloatTest.java，在这个类中，我们声明了五个变量，但是取值就两种int类型的10 和Float类型的11f。

**package com.louis.jvm;**

**public class IntAndFloatTest {**

**private final int a = 10;**

**private final int b = 10;**

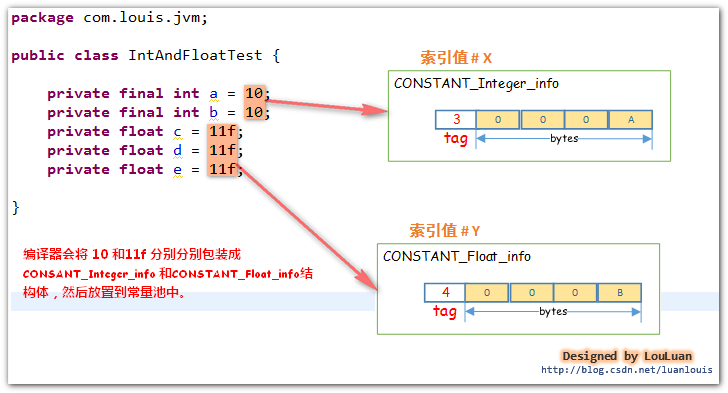
**private float c = 11f;**

**private float d = 11f;**

**private float e = 11f;**

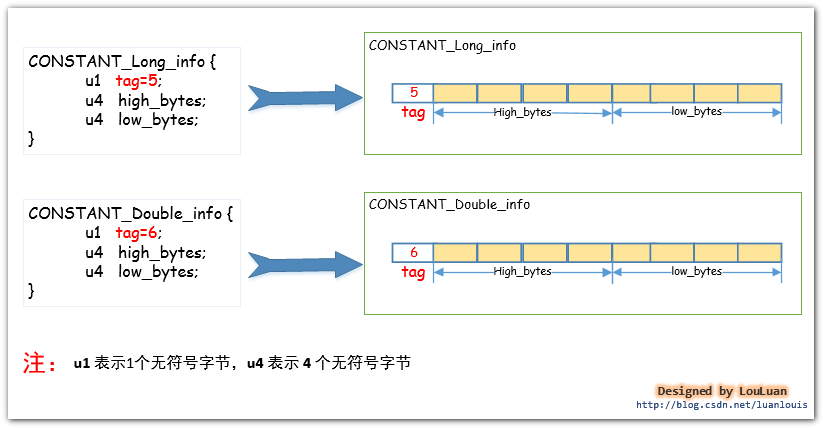
**}**

查看class文件后知道虽然我们在代码中写了两次10 和三次11f，但是常量池中，就只有一个常量10 和一个常量11f，实际上他们是不同的引用指向同一个地址



**B)long和 double数据类型的常量在常量池中是怎样表示和存储的**

Java语言规范规定了 long 类型和 double类型的数据类型占用8 个字节的空间。那么存在于class 字节码文件中的该类型的常量是如何存储的呢？相应地，在常量池中，将long和double类型的常量分别使用CONSTANT\_Long\_info和Constant\_Double\_info表示，他们的结构如下所示：



 举例：建下面的类 LongAndDoubleTest.java，在这个类中，我们声明了六个变量，但是取值就两种Long 类型的-6076574518398440533L 和Double 类型的10.1234567890D。

**package com.louis.jvm;**

**public class LongAndDoubleTest {**

**private long a = -6076574518398440533L;**

**private long b = -6076574518398440533L;**

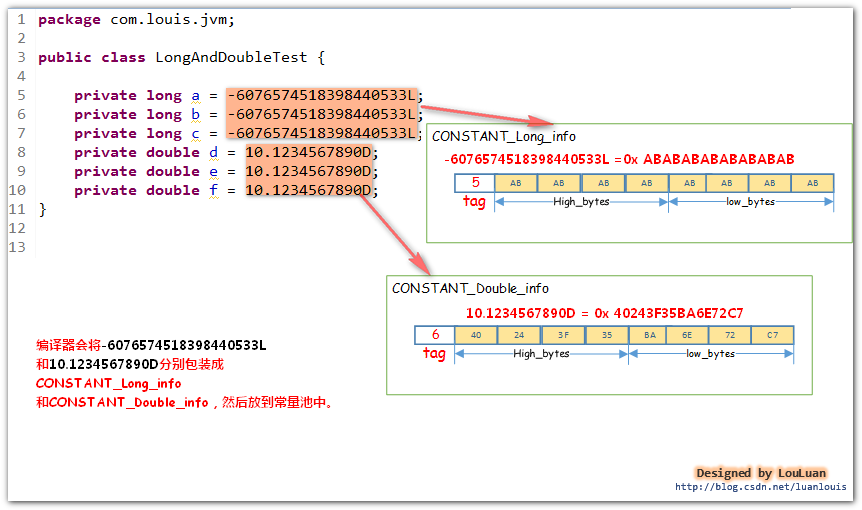
**private long c = -6076574518398440533L;**

**private double d = 10.1234567890D;**

**private double e = 10.1234567890D;**

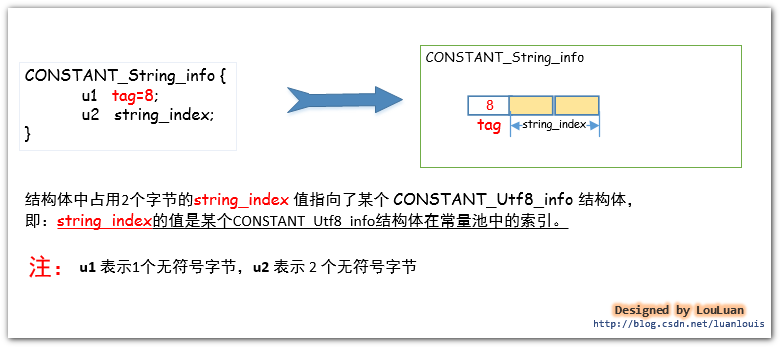
**private double f = 10.1234567890D;**

**}**

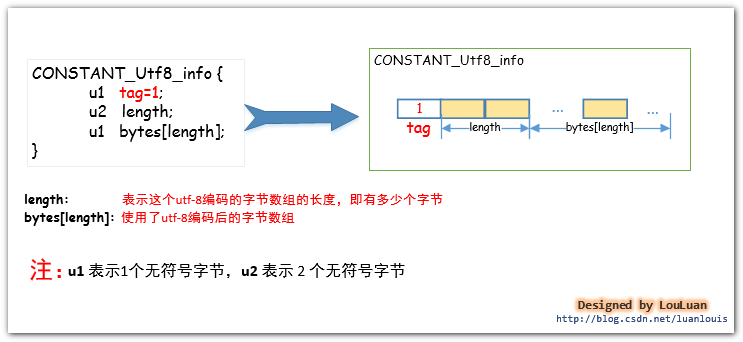


**String类型的字符串常量在常量池中是怎样表示和存储的**

在编译器编译的时候，都会将这些字符串转换成CONSTANT\_String\_info结构体，然后放置于常量池中。其结构如下所示：



如上图所示的结构体，CONSTANT\_String\_info结构体中的string\_index的值指向了CONSTANT\_Utf8\_info结构体，而字符串的utf-8编码数据就在这个结构体之中。如下图所示：



测试案例

**public class StringTest {**

**private String s1 = "JVM原理";**

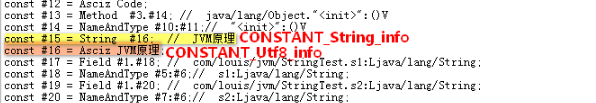
**private String s2 = "JVM原理";**

**private String s3 = "JVM原理";**

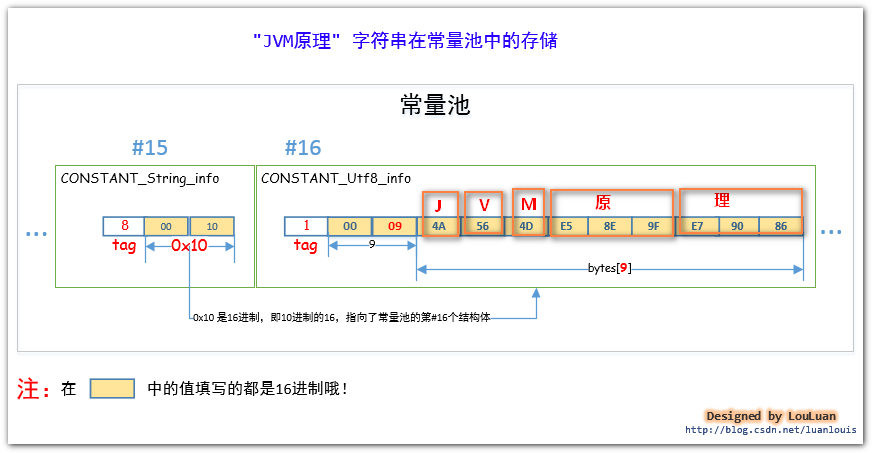
**private String s4 = "JVM原理";**

**}**

**常量池中信息如下图**

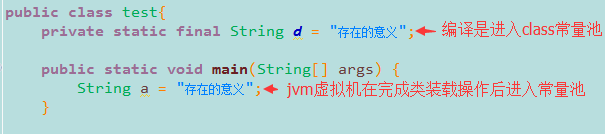


 在面的图中，我们可以看到CONSTANT\_String\_info结构体位于常量池的第#15个索引位置。而存放"Java虚拟机原理" 字符串的 UTF-8编码格式的字节数组被放到CONSTANT\_Utf8\_info结构体中，该结构体位于常量池的第#16个索引位置。上面的图只是看了个轮廓，让我们再深入地看一下它们的组织吧。请看下图：



# ****八．运行时常量池****

## 1.运行时常量池与class常量池



jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在**方法区**中，我们常说的常量池，就是指方法区中的运行时常量池。

# 九．程序计数器

## 1.什么是程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），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。

**注：**

    如果线程正在执行的是一个Java方法，这个计数器记录的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Natvie方法，这个计数器值则为空（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.程序计数器有两个作用

字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

程序计数器的特点

是一块较小的存储空间

线程私有。每条线程都有一个程序计数器。

是唯一一个不会出现OutOfMemoryError的内存区域。

生命周期随着线程的创建而创建，随着线程的结束而死亡。

说明:

　　由于Java虚拟机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（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了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，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

# 十．Jvm常见内存溢出错误

Java虚拟机规范规定JVM的内存分为了好几块，比如堆，栈，程序计数器，方法区等，而Hotspot jvm的实现中，将堆内存分为了三部分，新生代，老年代，持久带，其中持久带实现了规范中规定的方法区，而内存模型中不同的部分都会出现相应的OOM错误，接下来我们就分开来讨论一下。

1. 栈溢出(StackOverflowError)

栈溢出抛出java.lang.StackOverflowError错误，出现此种情况是因为方法运行的时候栈的深度超过了虚拟机容许的最大深度所致。出现这种情况，一般情况下是程序错误所致的，比如写了一个死递归，就有可能造成此种情况。 下面我们通过一段代码来模拟一下此种情况的内存溢出。

**import** java.util.\*;

**import** java.lang.\*;

**public** **class** OOMTest{

**public** **void** stackOverFlowMethod(){

      stackOverFlowMethod();

  }

**public** **static** **void** main(String... args){

      OOMTest oom = **new** OOMTest();

      oom.stackOverFlowMethod();

  }

}

运行以上代码抛出stackOverflowError

## 堆溢出(OutOfMemoryError:java heap space)

堆内存溢出的时候，虚拟机会抛出java.lang.OutOfMemoryError:java heap space,出现此种情况的时候，我们需要根据内存溢出的时候产生的dump文件来具体分析（需要增加-XX:+HeapDumpOnOutOfMemoryErrorjvm启动参数）。出现此种问题的时候有可能是内存泄露，也有可能是内存溢出了。如果内存泄露，我们要找出泄露的对象是怎么被GC ROOT引用起来，然后通过引用链来具体分析泄露的原因。如果出现了内存溢出问题，这往往是程序本生需要的内存大于了我们给虚拟机配置的内存，这种情况下，我们可以采用调大-Xmx来解决这种问题。

**import** java.util.\*;

**import** java.lang.\*;

**public** **class** OOMTest{

**public** **static** **void** main(String... args){

                List<**byte**[]> buffer = **new** ArrayList<**byte**[]>();

                buffer.add(**new** **byte**[10\*1024\*1024]);

        }

}

运行以上代码抛出OutOfMemoryError:java heap space

## 持久带溢出(OutOfMemoryError: PermGen space)

 jvm通过持久带实现了Java虚拟机规范中的方法区，而运行时的常量池就是保存在方法区中的，因此持久带溢出有可能是运行时常量池溢出，也有可能是方法区中保存的class对象没有被及时回收掉或者class信息占用的内存超过了我们配置。当持久带溢出的时候抛出java.lang.OutOfMemoryError: PermGen space。  
我在工作可能在如下几种场景下出现此问题。

1.使用一些应用服务器的热部署的时候，我们就会遇到热部署几次以后发现内存溢出了，这种情况就是因为每次热部署的后，原来的class没有被卸载掉。

2.如果应用程序本身比较大，涉及的类库比较多，但是我们分配给持久带的内存（通过-XX:PermSize和-XX:MaxPermSize来设置）比较小的时候也可能出现此种问题。

1. 一些第三方框架，比如spring,hibernate都通过字节码生成技术（比如CGLib）来实现一些增强的功能，这种情况可能需要更大的方法区来存储动态生成的Class文件。

我们就可以通过String.intern方法来模拟一下运行时常量区的溢出.下面我们通过如下的代码来模拟此种情况：

**import** java.util.\*;

**import** java.lang.\*;

**public** **class** OOMTest{

**public** **static** **void** main(String... args){

                List<String> list = **new** ArrayList<String>();

**while**(**true**){

                        list.add(UUID.randomUUID().toString().intern());

                }

        }

}

运行以上代码抛出OutOfMemoryError: PermGen space

## 4.OutOfMemoryError:unable to create native thread

最后我们在来看看java.lang.OutOfMemoryError:unable to create natvie thread这种错误。 出现这种情况的时候，一般是下面两种情况导致的：

1. 程序创建的线程数超过了操作系统的限制。对于Linux系统，我们可以通过ulimit -u来查看此限制。
2. 给虚拟机分配的内存过大，导致创建线程的时候需要的native内存太少。我们都知道操作系统对每个进程的内存是有限制的，我们启动Jvm,相当于启动了一个进程，假如我们一个进程占用了4G的内存，那么通过下面的公式计算出来的剩余内存就是建立线程栈的时候可以用的内存。 线程栈总可用内存=4G-（-Xmx的值）- （-XX:MaxPermSize的值）- 程序计数器占用的内存 通过上面的公式我们可以看出，-Xmx 和 MaxPermSize的值越大，那么留给线程栈可用的空间就越小，在-Xss参数配置的栈容量不变的情况下，可以创建的线程数也就越小。因此如果是因为这种情况导致的unable to create native thread,那么要么我们增大进程所占用的总内存，或者减少-Xmx或者-Xss来达到创建更多线程的目的。

## 5．不同方式配置不同作用域的jvm的堆**栈内存！**

## 设置eclipse中jvm堆栈内存

* 1. **修改eclipse的配置文件,对所有工程都起作用**

修改eclipse根目录下的eclipse.ini文件

-vmargs  //虚拟机设置

     -Xms40m //初始内存

     -Xmx256m //最大内存

     -Xmn16m //最小内存

     -XX:PermSize=128M //持久代

     -XX:MaxPermSize=256M

* 1. **jres VM Arguments参数的设置,对所有工程都起作用**

打开eclipse

window－preferences－**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/qh_java/article/details/_blank)**－Installed JREs －Edit－Default VM Arguments

      在VM自变量中输入：-Xmx128m -Xms64m -Xmn32m -Xss16m

## Tomcat内存设置

**windows下在catalina.bat的第一行增加：**

Java代码 :set JAVA\_OPTS=-Xms64m -Xmx256m -XX:PermSize=128M -XX:MaxNewSize=256m -XX:MaxPermSize=256m

[**Linux**](http://lib.csdn.net/base/linux)**下在catalina.sh的第一行增加：**

Java代码 :JAVA\_OPTS=-Xms64m -Xmx256m -XX:PermSize=128M -XX:MaxNewSize=256m -XX:MaxPermSize=256m

设置Tomcat启动的初始内存其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。

可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置

实例，以下给出1G内存环境下**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/qh_java/article/details/_blank)**jvm 的参数设置参考：

JAVA\_OPTS="-server -Xms800m -Xmx800m -XX:PermSize=64M -XX:MaxNewSize=256m -XX:MaxPermSize=128m -Djava.awt.headless=true "

JAVA\_OPTS="-server -Xms768m -Xmx768m -XX:PermSize=128m -XX:MaxPermSize=256m -XX: NewSize=192m -XX:MaxNewSize=384m"

CATALINA\_OPTS="-server -Xms768m -Xmx768m -XX:PermSize=128m -XX:MaxPermSize=256m -XX:NewSize=192m -XX:MaxNewSize=384m"

[**linux**](http://lib.csdn.net/base/linux)：

在/usr/local/apache-tomcat-5.5.23/bin 目录下的catalina.sh添加：

JAVA\_OPTS='-Xms512m -Xmx1024m'要加“m”说明是MB，否则就是KB了，在启动tomcat时会 报内存不足。

-Xms：初始值-Xmx：最大值 -Xmn：最小值

Windows：

在catalina.bat最前面加入set JAVA\_OPTS=-Xms128m -Xmx350m

如果用startup.bat启动tomcat,OK设置生效.够成功的分配200M内存.

但是如果不是执行startup.bat启动tomcat而是利用windows的系统服务启动tomcat服务,上面的设置就不生效了,就是说set JAVA\_OPTS=-Xms128m -Xmx350m 没起作用.上面分配200M内存就OOM了..

windows服务执行的是bin\tomcat.exe.他读取注册表中的值,而不是catalina.bat的设置.

解决办法:修改注册表

HKEY\_LOCAL\_MACHINE\SOFTWARE\Apache Software Foundation\Tomcat Service Manager\Tomcat5\Parameters\JavaOptions

原值为

-Dcatalina.home="C:\ApacheGroup\Tomcat5.0"-

Djava.endorsed.dirs="C:\ApacheGroup\Tomcat 5.0\common\endorsed"-Xrs

加入 -Xms300m -Xmx350m 重起tomcat服务,设置生效

# 运行机制案例分析

## 1.案例分析

**package** ceshi;

**public** **class** test {

**private** **static** **final** String ***f*** = "hello";

**public** **static** **void** **main**(String[] args) {

String a = "jvm";

String b = "jvm";

String c = "hello";

String d = **new** String("jvm");

System.***out***.println("a.equals(b):" + a.equals(b));

System.***out***.println("a==b:" + (a == b));

System.***out***.println("--------------------------------");

System.***out***.println("a.equals(d):" + a.equals(d));

System.***out***.println("a==d:" + (a == d));

System.***out***.println("-----------------------------------");

**if**(c.equals(***f***) && c == ***f***)

{

*sayHello*(a ,c);

}

}

**public** **static** **void** **sayHello**(String a ,String c){

String d = c + a;

System.***out***.println(d);

}

**先看输出结果**a.equals(b):true

a==b:true

-----------------------------------

a.equals(d):true

a==d:false

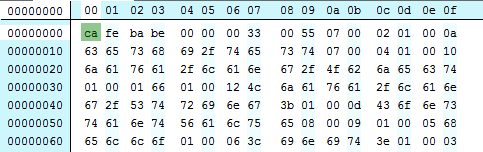
-----------------------------------

hellojvm

**思考两个问题**

## 案例分析

**1.编译**:java文件通过javac命令编译成class文件



部分class文件

//最开始的4个字节是class文件的魔数，其实很多其它格式文件也是靠魔数来识别的  
 cafe babe  
 //表明这个是JDK\_1.7 编译而成的  
 0000 0033  
 **2.常量池信息**

用命令**javap -v test** 查看常量池信息，如下图

<http://syyixin.iteye.com/blog/2183615>

https://yq.aliyun.com/articles/49201

# 十二．GC垃圾回收基础

## 对象分配

对象内存主要分配在新生代Eden区, 如果启用了本地线程分配缓冲, 则优先在TLAB上分配, 少数情况能会直接分配在老年代, 或被拆分成标量类型在栈上分配(JIT优化). 分配的规则并不是百分百固定, 细节主要取决于垃圾收集器组合, 以及VM内存相关的参数.

**TLAB :**JVM在内存新生代Eden Space中开辟了一小块线程私有的区域，称作TLAB（Thread-local allocation buffer）。

**对象分配**

**优先在Eden区分配**在[JVM内存模型](http://blog.csdn.net/zjf280441589/article/details/53437703)一文中, 我们大致了解了VM年轻代堆内存可以划分为一块Eden区和两块Survivor区. 在大多数情况下, 对象在新生代Eden区中分配, 当Eden区没有足够空间分配时, VM发起一次Minor GC, 将Eden区和其中一块Survivor区内尚存活的对象放入另一块Survivor区域, 如果在Minor GC期间发现新生代存活对象无法放入空闲的Survivor区, 则会通过空间分配担保机制使对象提前进入老年代(空间分配担保见下).

**大对象直接进入老年代**

Serial和ParNew两款收集器提供了-XX:PretenureSizeThreshold的参数, 令大于该值的大对象直接在老年代分配, 这样做的目的是避免在Eden区和Survivor区之间产生大量的内存复制(大对象一般指 需要大量连续内存的Java对象, 如很长的字符串和数组), 因此大对象容易导致还有不少空闲内存就提前触发GC以获取足够的连续空间.

### 对象晋升

**年龄阈值**

VM为每个对象定义了一个对象年龄(Age)计数器, 对象在Eden出生如果经第一次Minor GC后仍然存活, 且能被Survivor容纳的话, 将被移动到Survivor空间中, 并将年龄设为1. 以后对象在Survivor区中每熬过一次Minor GC年龄就+1. 当增加到一定程度(-XX:MaxTenuringThreshold, 默认15), 将会晋升到老年代.

**提前晋升: 动态年龄判定**

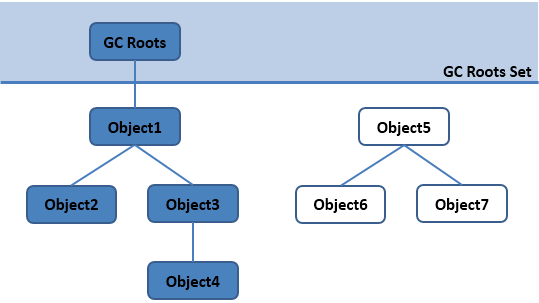
然而VM并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代: 如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半, 年龄大于或等于该年龄的对象就可以直接进入老年代, 而无须等到晋升年龄.

## 何时回收-对象生死判定

**可达性分析算法**

**GC Roots** :一组必须活跃的引用

在主流商用语言(如Java、C#)的主流实现中, 都是通过可达性分析算法来判定对象是否存活的: 通过一系列的称为 GC Roots 的对象作为起点, 然后向下搜索; 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, 即该对象不可达, 也就说明此对象是不可用的, 如下图: Object5、6、7 虽然互有关联, 但它们到GC Roots是不可达的, 因此也会被判定为可回收的对象:



在Java, 可作为GC Roots的对象包括:

方法区: 类静态属性引用的对象;

方法区: 常量引用的对象;

虚拟机栈(本地变量表)中引用的对象.

本地方法栈JNI(Native方法)中引用的对象。

# 十三．GC垃圾回收详解

## GC原理- 垃圾收集算法

**分代收集算法 VS 分区收集算法**

**分代收集算法**

当前主流VM垃圾收集都采用”分代收集”(Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为几块, 如JVM中的 新生代、老年代、永久代. 这样就可以根据各年代特点分别采用最适当的GC算法；

在新生代: 每次垃圾收集都能发现大批对象已死, 只有少量存活. 因此选用复制算法, 只需要付出少量存活对象的复制成本就可以完成收集.

在老年代: 因为对象存活率高、没有额外空间对它进行分配担保, 就必须采用“标记—清理”或“标记—整理”算法来进行回收, 不必进行内存复制, 且直接腾出空闲内存.

**分区收集算法**

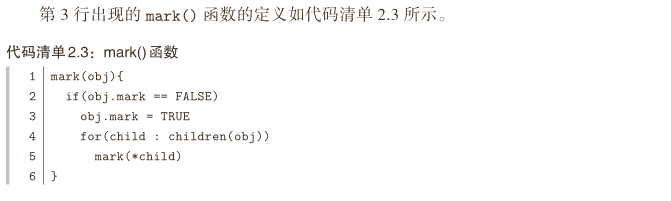
上面介绍的分代收集算法是将对象的生命周期按长短划分为两个部分, 而分区算法则将整个堆空间划分为连续的不同小区间, 每个小区间独立使用, 独立回收. 这样做的好处是可以控制一次回收多少个小区间.  
在相同条件下, 堆空间越大, 一次GC耗时就越长, 从而产生的停顿也越长. 为了更好地控制GC产生的停顿时间, 将一块大的内存区域分割为多个小块, 根据目标停顿时间, 每次合理地回收若干个小区间(而不是整个堆), 从而减少一次GC所产生的停顿.

**标记清除法**

标记清除法分为连个过程，第一是标记阶段，第二是清除阶段

标记阶段

 1.在标记阶段中，会遍历堆里面所有的对象。为此，我们首先要标记通过引用能到达的对象。首先我们标记这样的对象，然后递归地标记通过引用数组能访问到的对象。这样就能把所有活动对象都标记上了。伪代码如下



其实设置这个对象是否被标记是在对象的头部分的，obj.mark=true 为标记对象

上面伪代码只是对对象进行标记的操作，并不是遍历，那么在堆里面是如何寻找并遍历这些对象呢？**引出了两个算法深度优先搜索与广度优先搜索**

清除阶段

此阶段同样会遍历所有的对象，将没有被标记的对象回收掉，伪代码如下



**遍历对象算法**

同样使用的是深度优先算法和广度优先搜索算法

**标记法分类**

**清除法分类**

**缺点优点**