# 内存区域

由于Java程序是交由JVM执行的，所以我们在谈Java内存区域划分的时候事实上是指JVM内存区域划分。在讨论JVM内存区域划分之前，先来看一下Java程序具体执行的过程：

|  |
| --- |
| http://images.cnitblog.com/i/288799/201405/281630330728961.jpg |

如上图所示，首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后缀)，然后由JVM中的类加载器加载各个类的 字节码文件，加载完毕之后，交由JVM执行引擎执行。在整个程序执行过程中，JVM会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一 般被称作为Runtime Data Area（运行时数据区），也就是我们常说的JVM内存。因此，在Java中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

## 运行时数据区包括哪几部分？

根据《Java虚拟机规范》的规定，运行时数据区通常包括这几个部分：程序计数器(Program Counter Register)、Java栈(VM Stack)、本地方法栈(Native Method Stack)、方法区(Method Area)、堆(Heap)。

|  |
| --- |
| http://images.cnitblog.com/i/288799/201405/281726404166686.jpg |

如上图所示，JVM中的运行时数据区应该包括这些部分。在JVM规范中虽然规定了程序在执行期间运行时数据区应该包括这几部分，但是至于具体如何实现并没有做出规定，不同的虚拟机厂商可以有不同的实现方式。

## 运行时数据区的每部分到底存储了哪些数据？

### 程序计数器

　 程序计数器（Program Counter Register），也有称作为PC寄存器。想必学过汇编语言的朋友对程序计数器这个概念并不陌生，在汇编语言中，程序计数器是指CPU中的寄存器，它保 存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所 在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所 有的指令。

虽然JVM中的程序计数器并不像汇编语言中的程序计数器一样是物理概念上的CPU寄存器，但是JVM中的程序计数器的功能跟汇编语言中的程序计数器的功能在逻辑上是等同的，也就是说是用来指示 执行哪条指令的。

由于在JVM中，多线程是通过线程轮流切换来获得CPU执行时间的，因此，在任一具体时刻，一个CPU的内核只会执行一条线程中的指令，因此，为了能够使 得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行 次序。因此，可以这么说，程序计数器是每个线程所私有的。

在JVM规范中规定，如果线程执行的是非native方法，则程序计数器中保存的是当前需要执行的指令的地址；如果线程执行的是native方法，则程序计数器中的值是undefined。

由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的。

### Java栈

Java栈也称作虚拟机栈（Java Vitual Machine Stack），也就是我们常常所说的栈，跟C语言的数据段中的栈类似。事实上，Java栈是Java方法执行的内存模型。为什么这么说呢？下面就来解释一下其中的原因。

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。讲到这里，大家就应该会明白为什么 在 使用 递归方法的时候容易导致栈内存溢出的现象了以及为什么栈区的空间不用程序员去管理了（当然在Java中，程序员基本不用关系到内存分配和释放的事情，因为 Java有自己的垃圾回收机制），这部分空间的分配和释放都是由系统自动实施的。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。下图 表示了一个Java栈的模型：

|  |
| --- |
| http://images.cnitblog.com/i/288799/201405/291429030562182.jpg |

#### 局部变量表

顾名思义，想必不用解释大家应该明白它的作用了吧。就是用来存储方法中的局部变量（包括在方法中声明的非静态变量以及函数形参）。对于基本数据类型的变量，则直接存储它的值，对于引用类型的变量，则存的是指向对象的引用。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

#### 操作数栈

想必学过数据结构中的栈的朋友想必对表达式求值问题不会陌生，栈最典型的一个应用就是用来对表达式求值。想想一个线程执行方法的过程中，实际上就是不断执行语句的过程，而归根到底就是进行计算的过程。因此可以这么说，程序中的所有计算过程都是在借助于操作数栈来完成的。

#### 指向运行时常量池的引用

因为在方法执行的过程中有可能需要用到类中的常量（这里的常量意思很广，具体参见常量池），所以必须要有一个引用指向运行时常量。

#### 方法返回地址

当一个方法执行完毕之后，要返回之前调用它的地方，因此在栈帧中必须保存一个方法返回地址。

由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰。

### 本地方法栈

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和Java栈合二为一。

### 方法区

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。

**方法区里存放的是哪些内容？**

* 类的全限定名(类的全路径名)
* 类的直接超类的全限定名(如果这个类是Object,则它没有超类)
* 所有的直接接口全限定名的有序列表(假如它实现了多个接口)
* 这个类是类型(类还是接口)
* 类的访问修饰符,如public、abstract、final等
* 字段、方法信息、
* 静态存储块/静态区
* 常量池

#### 静态存储块

类变量也叫静态变量,也就是在变量前加了static的变量; 实例变量也叫对象变量,即没加static的变量;

态区是专门用来存放静态变量以及静态块的。所有类的实例都共享方法区中的内容。访问类变量的方式可通过实例(对象)来访问，也可通过类型来直接访问，java规范推荐使用类型来直接访问。

#### 常量池（待再完善）

在符号解析的过程当中，常量池扮演着非常重要的工作。

常量池由多条“常量池项”组成，每一个常量池项又由两部分组成，这里分别称为“常量池项头”和“常量池项体”。

常量池项头表明常量池项的类型，常量池项共分为11种类型，分别为：

|  |  |  |
| --- | --- | --- |
| 常量池项类型 | 值 | 说明 |
| CONSTANT\_Utf8 | 1 | UTF-8编码的Unicode字符串 |
| CONSTANT\_Integer | 3 | int型常量 |
| CONSTANT\_Float | 4 | Float型常量 |
| CONSTANT\_Long | 5 | Long型常量 |
| CONSTANT\_Double | 6 | double型常量 |
| CONSTANT\_Class | 7 | 对一个class的符号引用 |
| CONSTANT\_String | 8 | String型常量 |
| CONSTANT\_Fieldref | 9 | 对一个字段的符号引用 |
| CONSTANT\_Methodref | 10 | 对一个类方法的符号引用 |
| CONSTANT\_InterfaceMedthodref | 11 | 对一个接口方法的符号引用 |
| CONSTANT\_NameAndType | 12 | 对名称和类型的符号引用 |

常量池项体中存放的就是对应的常量数据，比如各种数值型的值或者字符串值等等。

JVM会在常量池中定义如下信息：

* 字符型数据：utf-8，方法名称、类名称、属性名称、常量变量名称等等，这个类型一般用于定义其他类型所关联的“字串”信息。

比如：class A{…}，

* 数字型常量：long、integer、double、float，包括使用到的一些常量定义
* String常量：string，包括字串常量定义
* 类和引用信息：包括Class、MethodRef、InterfaceMethodRef、Fieldref、NameAndType信息

常量池中除了包含代码中所定义的各种基本类型（如int、long等等）和对象型（如String及数组）的常量值还，还包含一些以文本形式出现的符号引用，比如：

* + 类和接口的全限定名；
  + 字段的名称和描述符；
  + 方法和名称和描述符。

编译时，如果发现对其它类方法的调用或者对其它类字段的引用的话，记录进class文件中的，只能是一个**文本形式**的符号引用，在连接过程中，虚拟机根据这个文本信息去查找对应的方法或字段。

所以， class文件中的“常量”内容很非富，这些常量集中在class中的一个区域存放，一个紧接着一个，这里就称为“常量池”。

常量池主要涉及到常量池里的内容和常量池解析这两块:

  在方法区中，每个类型都对应一个常量池，常量池中存储了诸如文字字符串、final变量值、类名常量、方法名常量。JVM把常量池组织为入口列表的形式，可通过索引来访问常量池中的各个入口，每个常量池入口的第一个字节都是个标志，用这个标志来表示该入口中存储的常量类型，如**CONSTANT\_Long**表示里面存储的是long类型字面值，**CONSTANT\_Class\_info**表示里面存的是某个Class的类型信息(存的可能是个普通的字符串，然后经过常量池解析，则变成指向某个类的引用)。   
      除了字面常量值以外，常量池还可以容纳其它几种符号引用：类和接口的全限定名、字段名称和描述符、方法名称和描述符。

* 类和接口的全限定名指的是当前类的全限定名。
* 字段名称指的是类或接口的实例变量或类变量，字段的描述符是一个指示字段的类型的字符串。如在一个类中有以下形式的声明：A a = null， 则a为字段名，A为字段描述符。
* 方法的描述符也是个字符串，该字符串指示了方法的返回值和参数的数量、顺序和类型。

在运行时，JVM从常量池中获得符号引用，然后在运行时解析成引用项的实际地址，最后通过常量池中的全限定名、方法和字段描述符，把当前类或接口中的代码与其它类或接口中的代码联系起来。

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。

　　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。

　　在JVM规范中，没有强制要求方法区必须实现垃圾回收。很多人习惯将方法区称为“永久代”，是因为HotSpot虚拟机以永久代来实现方法区，从而JVM的垃圾收集器可以像管理堆区一样管理这部分区域，从而不需要专门为这部分设计垃圾回收机制。不过自从JDK7之后，Hotspot虚拟机便将运行时常量池从永久代移除了。

### 堆

在C语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。那么在Java中是怎么样的呢？

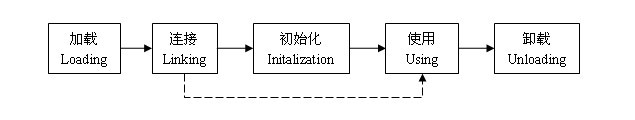
　　Java中的堆是用来存储对象本身（当然，对象引用是存放在Java栈中 的）的以及数组（当然，数组引用是存放在Java栈中 的）。只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java 垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

## 再论堆和栈

程序运行时，我们最好对数据保存到什么地方做到心中有数。特别要注意的是内存的分配。有六个地方都可以保存数据：  
(1) 寄存器。这是最快的保存区域，因为它位于和其他所有保存方式不同的地方：处理器内部。然而，寄存器的数量十分有限，所以寄存器是根据需要由编译器分配。我们对此没有直接的控制权，也不可能在自己的程序里找到寄存器存在的任何踪迹。  
(2) 堆栈。驻留于常规RAM（随机访问存储器）区域，但可通过它的“堆栈指针”获得处理的直接支持。堆栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些Java数据要保存在堆栈里——特别是对象句柄，但Java对象并不放到其中。  
(3) 堆。一种常规用途的内存池（也在RAM区域），其中保存了Java对象。和堆栈不同，“内存堆”或“堆”（Heap）最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用new命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价：在堆里分配存储空间时会花掉更长的时间！  
(4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管也在RAM里）。程序运行期间，静态存储的数据将随时等候调用。可用static关键字指出一个对象的特定元素是静态的。但Java对象本身永远都不会置入静态存储空间。  
(5) 常数存储。常数值通常直接置于程序代码内部。这样做是安全的，因为它们永远都不会改变。有的常数需要严格地保护，所以可考虑将它们置入只读存储器（ROM）。  
(6) 非RAM存储。若数据完全独立于一个程序之外，则程序不运行时仍可存在，并在程序的控制范围之外。其中两个最主要的例子便是“流式对象”和“固定对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，对象保存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普通的、基于RAM的对象。Java 1.1提供了对Lightweight persistence的支持。未来的版本甚至可能提供更完整的方案。

# 类的生命周期

当我们编写一个java的源文件后，经过编译会生成一个后缀名为class的文件，这种文件叫做字节码文件，只有这种字节码文件才能够在java虚拟机中运行，java类的生命周期就是指一个class文件从加载到卸载的全过程。



## 加载

找到需要加载的类并把类的信息加载到jvm的方法区中，然后在堆区中实例化一个java.lang.Class对象，作为方法区中这个类的信息的入口。

* class文件的存在形式：
* path中的class文件
* jar中的class文件
* 根据一定的规则实时生成，比如设计模式中的动态代理模式，就是根据相应的类自动生成它的代理类。
* 从非class文件中获取，其实这与直接从class文件中获取的方式本质上是一样的，这些非class文件在jvm中运行之前会被转换为可被jvm所识别的字节码文件。
* 从网络中获取：比如10年前十分流行的Applet。

## 连接

有一点需要注意，就是有时连接阶段并不会等加载阶段完全完成之后才开始，而是交叉进行，可能一个类只加载了一部分之后，连接阶段就已经开始了。但是这两个阶段总的开始时间和完成时间总是固定的：加载阶段总是在连接阶段之前开始，连接阶段总是在加载阶段完成之后完成。

连接阶段比较复杂，一般会跟加载阶段和初始化阶段交叉进行，这个阶段的主要任务就是做一些加载后的验证工作以及一些初始化前的准备工作，可以细分为三个步骤：验证、准备和解析。

### **验证**

当一个类被加载之后，必须要验证一下这个类是否合法，比如这个类是不是符合字节码的格式、变量与方法是不是有重复、数据类型是不是有效、继承与实现是否合乎标准等等。总之，这个阶段的目的就是保证加载的类是能够被jvm所运行。

### 准备

准备阶段的工作就是为类的静态变量分配内存并设为jvm默认的初值，对于非静态的变量，则不会为它们分配内存。有一点需要注意，这时候，静态变量的初值为jvm默认的初值，而不是我们在程序中设定的初值。jvm默认的初值是这样的：

* 基本类型（int、long、short、char、byte、boolean、float、double）的默认值为0。
* 引用类型的默认值为null。
* 常量的默认值为我们程序中设定的值，比如我们在程序中定义final static int a = 100，则准备阶段中a的初值就是100。

### **解析**

这一阶段的任务就是把常量池中的符号引用转换为直接引用。那么什么是符号引用，什么又是直接引用呢？我们来举个 例子：我们要找一个人，我们现有的信息是这个人的身份证号是1234567890。只有这个信息我们显然找不到这个人，但是通过公安局的身份系统，我们输 入1234567890这个号之后，就会得到它的全部信息：比如安徽省黄山市余暇村18号张三，通过这个信息我们就能找到这个人了。这 里，123456790就好比是一个符号引用，而安徽省黄山市余暇村18号张三就是直接引用。在内存中也是一样，比如我们要在内存中找一个类里面的一个叫 做show的方法，显然是找不到。但是在解析阶段，jvm就会把show这个名字转换为指向方法区的的一块内存地址，比如c17164，通过c17164 就可以找到show这个方法具体分配在内存的哪一个区域了。这里show就是符号引用，而c17164就是直接引用。在解析阶段，jvm会将所有的类或接 口名、字段名、方法名转换为具体的内存地址。

连接阶段完成之后会根据使用的情况（直接引用还是被动引用）来选择是否对类进行初始化。

## 初始化

初始化过程的主要操作是执行静态代码块和初始化静态域。在一个类被初始化之前，它的直接父类也需要被初始化。但是，一个接口的初始化，不会引起其父接口的初始化。在初始化的时候，会按照源代码中从上到下的顺序依次执行静态代码块和初始化静态域。

如果一个类被直接引用，就会触发类的初始化。在java中，直接引用的情况有：

* 通过new关键字实例化对象;读取或设置类的静态变量;调用类的静态方法。
* 通过反射方式执行以上三种行为。
* 初始化子类的时候，会触发父类的初始化。
* 作为程序入口直接运行时（也就是直接调用main方法）。

## 使用

当调用类方法或类变量时，没什么要说的；

但当使用实例变量或实例方法时，首先得有个实例化，那么实例化的过程是什么呢？

### 实例化

实例化过程：

* 给对象在内存（堆）中分配空间(给属性赋值)；
* 属性赋默认值(给属性赋初始值用等号‘＝’）)；
* byte,short.int,long ->0
* float  ->0.0f
* double ->0.0
* boolean  ->false
* String ->null
* char ->'\u0000'4
* 调用构造方法（方法和类名相同，没有返回类型）
* 将对象在堆区中的首地址返回给引用a

实例化方式：

* 用new语句创建对象，这是最常见的创建对象的方法。
* 通过工厂方法返回对象，如：String str = String.valueOf(23);
* 运用反射手段,调用java.lang.Class或者java.lang.reflect.Constructor类的newInstance() 实例方法。如：Object obj = Class.forName("java.lang.Object").newInstance();
* 调用对象的clone()方法。
* 通过I/O流（包括反序列化），如运用反序列化手段，调用java.io.ObjectInputStream对象的 readObject()方法。

这里对new要再唠叨一下，当我们new一个实例时，存在两种情况：

* 仅实例化，原因是该类已经加载、链接、初始化。
* 还未加载，那么就需要加载、链接、初始化、实例化一起何成，此后再实例化就仅实例化即可。

|  |
| --- |
| class Profile{  private String classname;  private String male;  private Integer age;   static {  System.*out*.println("Profile static block ");  }  Profile(){  System.*out*.println("Profile Constructor ： no parameters");  }   Profile(String classname,String male, Integer age) {  this.classname=classname;  this.male = male;  this.age = age;  System.*out*.println("Profile Constructor ： ( from "+classname+" )");  } } |

|  |
| --- |
| public class Animal {   private String name;  static {  System.*out*.println("Animal static block ");  }  private static Profile *profile*=new Profile("Animal","1",10);   public Animal(){  System.*out*.println("Animal Constructor: no parameters");  }  public Animal(String name){  this.name=name;  System.*out*.println("Animal Constructor: existing parameters");  }   public void walk(){  System.*out*.println("Animal class method :walk.");  }   public static void dosomething(){  System.*out*.println("Animal static method : dosomething.");  }    public static void main(String args[]){  Animal animal=new Animal();//<1>  Animal.dosomething();//<2>  // 单独执行<1>,console输出：  // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Animal Constructor: no parameters  // 单独执行<2>,console输出：  // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Animal static method : dosomething.  // 同时执行<1>和<2>,console输出：  // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Animal Constructor: no parameters // Animal static method : dosomething.  } } |

|  |
| --- |
| package com.study.newProduce;  */\*\*  \* Created by Edward on 2016/3/18.  \*/* public class Dog extends Animal{   private String name;  static {  System.*out*.println("Dog static block ");  }  private static Profile *profile*=new Profile("Dog","1",10);   public Dog(){  System.*out*.println("Dog Constructor: no parameters");  }   public Dog(String name){  this.name=name;  System.*out*.println("Dog Constructor: existing parameters");  }   public void walk(){  System.*out*.println("Dog class method :walk.");  }   public static void dosomething(){  System.*out*.println("Dog static method : dosomething.");  }    public static void main(String args[]){  Dog dog=new Dog();//<1>  Dog.*dosomething*();//<2>  // 单独执行<1>,console输出： // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Dog static block // Profile Constructor ： ( from Dog ) // Animal Constructor: no parameters // Dog Constructor: no parameters  // 单独执行<2>,console输出： // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Dog static block // Profile Constructor ： ( from Dog ) // Dog static method : dosomething.  // 同时执行<1>和<2>,console输出： // Animal static block // Profile static block // Profile Constructor ： ( from Animal ) // Dog static block // Profile Constructor ： ( from Dog ) // Animal Constructor: no parameters // Dog Constructor: no parameters // Dog static method : dosomething.  } } |

## 卸载

 关于类的卸载，笔者在[**单例模式讨论篇：单例模式与垃圾回收**](http://blog.csdn.net/zhengzhb/article/details/7331354)一文中有过描述，在类使用完之后，如果满足下面的情况，类就会被卸载：

* 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。
* 加载该类的ClassLoader已经被回收。
* 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

   如果以上三个条件全部满足，jvm就会在方法区垃圾回收的时候对类进行卸载，类的卸载过程其实就是在方法区中清空类信息，java类的整个生命周期就结束了。