参考：<http://blog.csdn.net/u010425776/article/details/51170118>

# 深入理解JVM(一)——JVM内存模型

# JVM内存模型

Java虚拟机(Java Virtual Machine=JVM)的内存空间分为五个部分，分别是：   
1. 程序计数器   
2. Java虚拟机栈   
3. 本地方法栈   
4. 堆   
5. 方法区。

6. 运行时常量池

7. 直接内存

下面对这五个区域展开深入的介绍。 

## 1. 程序计数器

### 1.1. 什么是程序计数器？

程序计数器是一块较小的内存空间，可以把它看作当前线程正在执行的字节码的行号指示器。也就是说，程序计数器里面记录的是当前线程正在执行的那一条字节码指令的地址。   
**注：**但是，如果当前线程正在执行的是一个本地方法，那么此时程序计数器为空。 

### 1.2. 程序计数器的作用

程序计数器有两个作用：

1. 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。
2. 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。

### 1.3. 程序计数器的特点

1. 是一块较小的存储空间
2. 线程私有。每条线程都有一个程序计数器。
3. 是唯一一个不会出现OutOfMemoryError的内存区域。
4. 生命周期随着线程的创建而创建，随着线程的结束而死亡。

## 2. Java虚拟机栈(JVM Stack)

### 2.1. 什么是Java虚拟机栈？

Java虚拟机栈是描述Java方法运行过程的内存模型。   
Java虚拟机栈会为每一个即将运行的Java方法创建一块叫做“栈帧”的区域，这块区域用于存储该方法在运行过程中所需要的一些信息，这些信息包括：

1. 局部变量表   
   存放基本数据类型变量、引用类型的变量、returnAddress类型的变量。
2. 操作数栈
3. 动态链接
4. 方法出口信息
5. 等

当一个方法即将被运行时，Java虚拟机栈首先会在Java虚拟机栈中为该方法创建一块“栈帧”，栈帧中包含局部变量表、操作数栈、动态链接、方法出口信息等。当方法在运行过程中需要创建局部变量时，就将局部变量的值存入栈帧的局部变量表中。   
当这个方法执行完毕后，这个方法所对应的栈帧将会出栈，并释放内存空间。

**注意：**人们常说，Java的内存空间分为“栈”和“堆”，栈中存放局部变量，堆中存放对象。   
这句话不完全正确！这里的“堆”可以这么理解，但这里的“栈”只代表了Java虚拟机栈中的局部变量表部分。真正的Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。 

### 2.2. Java虚拟机栈的特点

1. 局部变量表的创建是在方法被执行的时候，随着栈帧的创建而创建。而且，局部变量表的大小在编译时期就确定下来了，在创建的时候只需分配事先规定好的大小即可。此外，在方法运行的过程中局部变量表的大小是不会发生改变的。
2. Java虚拟机栈会出现两种异常：StackOverFlowError和OutOfMemoryError。   
   a) StackOverFlowError：   
   若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。   
   b) OutOfMemoryError：   
   若Java虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。
3. Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

**注：StackOverFlowError和OutOfMemoryError的异同？**   
StackOverFlowError表示当前线程申请的栈超过了事先定好的栈的最大深度，但内存空间可能还有很多。   
而OutOfMemoryError是指当线程申请栈时发现栈已经满了，而且内存也全都用光了。 

## 3. 本地方法栈

### 3.1. 什么是本地方法栈？

本地方法栈和Java虚拟机栈实现的功能类似，只不过本地方法区是本地方法运行的内存模型。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间。

也会抛出StackOverFlowError和OutOfMemoryError异常。

## 4. 堆

### 4.1. 什么是堆？

堆是用来存放对象的内存空间。   
**几乎所有**的对象都存储在堆中。 

### 4.2. 堆的特点

1. 线程共享   
   整个Java虚拟机只有一个堆，所有的线程都访问同一个堆。而程序计数器、Java虚拟机栈、本地方法栈都是一个线程对应一个的。
2. 在虚拟机启动时创建
3. 垃圾回收的主要场所。
4. 可以进一步细分为：新生代、老年代。   
   新生代又可被分为：Eden、From Survior、To Survior。   
   不同的区域存放具有不同生命周期的对象。这样可以根据不同的区域使用不同的垃圾回收算法，从而更具有针对性，从而更高效。
5. 堆的大小既可以固定也可以扩展，但主流的虚拟机堆的大小是可扩展的，因此当线程请求分配内存，但堆已满，且内存已满无法再扩展时，就抛出OutOfMemoryError。

## 5. 方法区

### 5.1. 什么是方法区？

Java虚拟机规范中定义方法区是堆的一个逻辑部分。   
方法区中存放已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等。 

### 5.2. 方法区的特点

1. 线程共享   
   方法区是堆的一个逻辑部分，因此和堆一样，都是线程共享的。整个虚拟机中只有一个方法区。
2. 永久代   
   方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，我们把方法区称为老年代。
3. 内存回收效率低   
   方法区中的信息一般需要长期存在，回收一遍内存之后可能只有少量信息无效。   
   对方法区的内存回收的主要目标是：对常量池的回收 和 对类型的卸载。
4. Java虚拟机规范对方法区的要求比较宽松。   
   和堆一样，允许固定大小，也允许可扩展的大小，还允许不实现垃圾回收。

### 5.3. 什么是运行时常量池？

方法区中存放三种数据：类信息、常量、静态变量、即时编译器编译后的代码。其中常量存储在运行时常量池中。

我们一般在一个类中通过public static final来声明一个常量。这个类被编译后便生成Class文件，这个类的所有信息都存储在这个class文件中。

当这个类被Java虚拟机加载后，class文件中的常量就存放在方法区的运行时常量池中。而且在运行期间，可以向常量池中添加新的常量。如：String类的intern()方法就能在运行期间向常量池中添加字符串常量。

当运行时常量池中的某些常量没有被对象引用，同时也没有被变量引用，那么就需要垃圾收集器回收。 

## 6. 直接内存

直接内存是除Java虚拟机之外的内存，但也有可能被Java使用。

在NIO中引入了一种基于通道和缓冲的IO方式。它可以通过调用本地方法直接分配Java虚拟机之外的内存，然后通过一个存储在Java堆中的DirectByteBuffer对象直接操作该内存，而无需先将外面内存中的数据复制到堆中再操作，从而提升了数据操作的效率。

直接内存的大小不受Java虚拟机控制，但既然是内存，当内存不足时就会抛出OOM异常。 

## 综上所述

1. Java虚拟机的内存模型中一共有两个“栈”，分别是：Java虚拟机栈和本地方法栈。   
   两个“栈”的功能类似，都是方法运行过程的内存模型。并且两个“栈”内部构造相同，都是线程私有。   
   只不过Java虚拟机栈描述的是Java方法运行过程的内存模型，而本地方法栈是描述Java本地方法运行过程的内存模型。
2. Java虚拟机的内存模型中一共有两个“堆”，一个是原本的堆，一个是方法区。方法区本质上是属于堆的一个逻辑部分。堆中存放对象，方法区中存放类信息、常量、静态变量、即时编译器编译的代码。
3. 堆是Java虚拟机中最大的一块内存区域，也是垃圾收集器主要的工作区域。
4. 程序计数器、Java虚拟机栈、本地方法栈是线程私有的，即每个线程都拥有各自的程序计数器、Java虚拟机栈、本地方法区。并且他们的生命周期和所属的线程一样。   
   而堆、方法区是线程共享的，在Java虚拟机中只有一个堆、一个方法栈。并在JVM启动的时候就创建，JVM停止才销毁。

# 深入理解JVM(二)——揭开HotSpot对象创建的奥秘

原创 2016年04月19日 15:19:20

* 标签：
* [jvm](http://so.csdn.net/so/search/s.do?q=jvm&t=blog) /
* [对象](http://so.csdn.net/so/search/s.do?q=%E5%AF%B9%E8%B1%A1&t=blog) /
* [虚拟机](http://so.csdn.net/so/search/s.do?q=%E8%99%9A%E6%8B%9F%E6%9C%BA&t=blog) /
* [class](http://so.csdn.net/so/search/s.do?q=class&t=blog)
* 1544

## 对象的创建过程

当虚拟机遇到一条含有new的指令时，会进行一系列对象创建的操作：

1. 检查常量池中是否有即将要创建的这个对象所属的类的符号引用；
   * 若常量池中没有这个类的符号引用，说明这个类还没有被定义！抛出ClassNotFoundException；
   * 若常量池中有这个类的符号引用，则进行下一步工作；
2. 进而检查这个符号引用所代表的类是否已经被JVM加载；
   * 若该类还没有被加载，就找该类的class文件，并加载进方法区；
   * 若该类已经被JVM加载，则准备为对象分配内存；
3. 根据方法区中该类的信息确定该类所需的内存大小；   
   一个对象所需的内存大小是在这个对象所属类被定义完就能确定的！且一个类所生产的所有对象的内存大小是一样的！JVM在一个类被加载进方法区的时候就知道该类生产的每一个对象所需要的内存大小。
4. 从堆中划分一块对应大小的内存空间给新的对象；   
   分配堆中内存有两种方式：
   * 指针碰撞   
     如果JVM的垃圾收集器采用复制算法或标记-整理算法，那么堆中空闲内存是完整的区域，并且空闲内存和已使用内存之间由一个指针标记。那么当为一个对象分配内存时，只需移动指针即可。因此，这种在完整空闲区域上通过移动指针来分配内存的方式就叫做“指针碰撞”。
   * 空闲列表   
     如果JVM的垃圾收集器采用标记-清除算法，那么堆中空闲区域和已使用区域交错，因此需要用一张“空闲列表”来记录堆中哪些区域是空闲区域，从而在创建对象的时候根据这张“空闲列表”找到空闲区域，并分配内存。   
     综上所述：JVM究竟采用哪种内存分配方法，取决于它使用了何种垃圾收集器。
5. 为对象中的成员变量赋上初始值(默认初始化)；
6. 设置对象头中的信息；
7. 调用对象的构造函数进行初始化   
   此时，整个对象的创建过程就完成了。

## 对象的内存模型

一个对象从逻辑角度看，它由成员变量和成员函数构成，从物理角度来看，对象是存储在堆中的一串二进制数，这串二进制数的组织结构如下。

对象在内存中分为三个部分：

1. 对象头
2. 实例数据
3. 对齐补充

### 1. 对象头

对象头中记录了对象在运行过程中所需要使用的一些数据：哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。

此外，对象头中可能还包含类型指针。通过该指针能确定这个对象所属哪个类。

此外，如果对象是一个数组，那么对象头中还要包含数组长度。

### 2. 实例数据

实力数据部分就是成员变量的值，其中包含父类的成员变量和本类的成员变量。

### 3. 对齐补充

用于确保对象的总长度为8字节的整数倍。   
HotSpot要求对象的总长度必须是8字节的整数倍。由于对象头一定是8字节的整数倍，但实例数据部分的长度是任意的，因此需要对齐补充字段确保整个对象的总长度为8的整数倍。

## 访问对象的过程

我们知道，引用类型的变量中存放的是一个地址，那么根据地址类型的不同，对象有不同的访问方式：

1. 句柄访问方式   
   堆中需要有一块叫做“句柄池”的内存空间，用于存放所有对象的地址和所有对象所属类的类信息。   
   引用类型的变量存放的是该对象在句柄池中的地址。访问对象时，首先需要通过引用类型的变量找到该对象的句柄，然后根据句柄中对象的地址再访问对象。
2. 直接指针访问方式   
   引用类型的变量直接存放对象的地址，从而不需要句柄池，通过引用能够直接访问对象。   
   但对象所在的内存空间中需要额外的策略存储对象所属的类信息的地址。

### 比较

HotSpot采用直接指针方式访问对象，因为它只需一次寻址操作，从而性能比句柄访问方式快一倍。但它需要额外的策略存储对象在方法区中类信息的地址。