**[深入理解JVM(一)——JVM内存模型](http://blog.csdn.net/u010425776/article/details/51170118)**

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

# 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。

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。

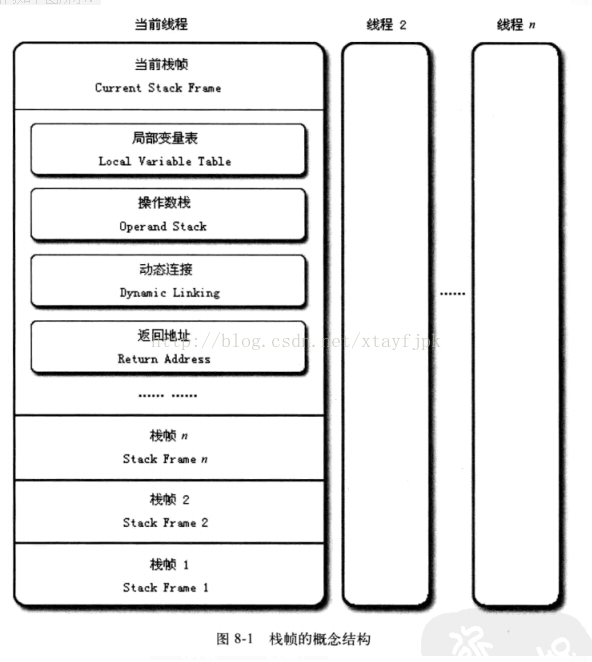
**注意：**人们常说，Java的内存空间分为“栈”和“堆”，栈中存放局部变量，堆中存放对象。   
这句话不完全正确！这里的“堆”可以这么理解，但这里的“栈”只代表了Java虚拟机栈中的局部变量表部分。真正的Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。

**补充：运行时栈帧结构**

栈帧(Stack Frame)是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区的虚拟机栈(Virtual Machine Stack)的栈元素。栈帧存储了方法的局部变量表，操作数栈，动态连接和方法返回地址等信息。第一个方法从调用开始到执行完成，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

每一个栈帧都包括了局部变量表，操作数栈，动态连接，方法返回地址和一些额外的附加信息。在编译代码的时候，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，并且写入到了方法表的Code属性中，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，而仅仅取决于具体虚拟机的实现。

一个线程中的方法调用链可能会很长，很多方法都同时处理执行状态。对于执行引擎来讲，活动线程中，只有虚拟机栈顶的栈帧才是有效的，称为**当前栈帧(Current Stack Frame)**，这个栈帧所关联的方法称为当前方法(Current Method)。执行引用所运行的所有字节码指令都只针对当前栈帧进行操作。栈帧的概念结构如下图所示：



1.局部变量表**max\_locals**

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**。在Java程序编译为Class文件时，就在方法表的Code属性的**max\_locals**数据项中确定了该方法需要分配的最大局部变量表的容量。

在方法执行时，虚拟机是使用局部变量表完成参数变量列表的传递过程，如果是**实例方法**，那么局部变量表中的**每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列表的顺序来排列，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来分配其余的Slot。**局部变量表中的Slot是可重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法，如果当前字节码PC计算器的值已经超出了某个变量的作用域，那么这个变量对应的Slot就可以交给其它变量使用。**

局部变量不像前面介绍的类变量那样存在“准备阶段”。**类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义的值**。因此即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值。但局部变量就不一样了，如果一个局部变量定义了但没有赋初始值是不能使用的。

2.操作数栈**max\_stacks**

操作数栈也常被称为**操作栈**，它是一个后入先出栈。同局部变量表一样，操作数栈的最大深度也是编译的时候被写入到方法表的Code属性的**max\_stacks**数据项中。操作数栈的每一个元素可以是任意Java数据类型，包括long和double。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。栈容量的单位为“字宽”，对于32位虚拟机来说，一个”字宽“占4个字节，对于64位虚拟机来说，**一个”字宽“占8个字节。**

当一个方法刚刚执行的时候，这个方法的操作数栈是空的，在方法执行的过程中，会有**各种字节码指向操作数栈中写入和提取值，也就是入栈与出栈操作。**例如，在做算术运算的时候就是通过操作数栈来进行的，又或者调用其它方法的时候是通过操作数栈来行参数传递的。

和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作——压栈和出栈—来访问的。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的：如int、long、float、double、reference和returnType的存储。对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int。

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个int类型的局部变量相加，再把结果保存到第三个局部变量的：

begin

iload\_0 // push the int in local variable 0 ontothe stack

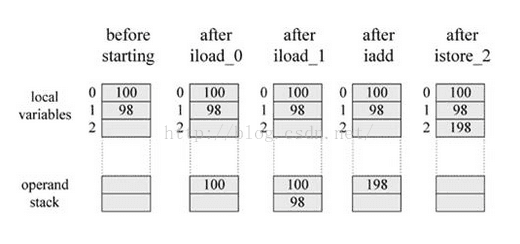
iload\_1 //push the int in local variable 1 onto the stack

iadd // pop two ints, add them, push result

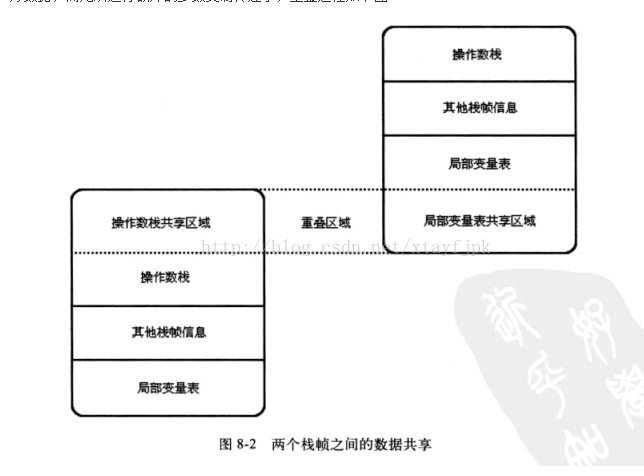
istore\_2 // pop int, store into local variable 2

end

在这个字节码序列里，前两个指令iload\_0和iload\_1将存储在局部变量中索引为0和1的整数压入操作数栈中，其后iadd指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令istore\_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置。下图详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数栈区域以空白表示。

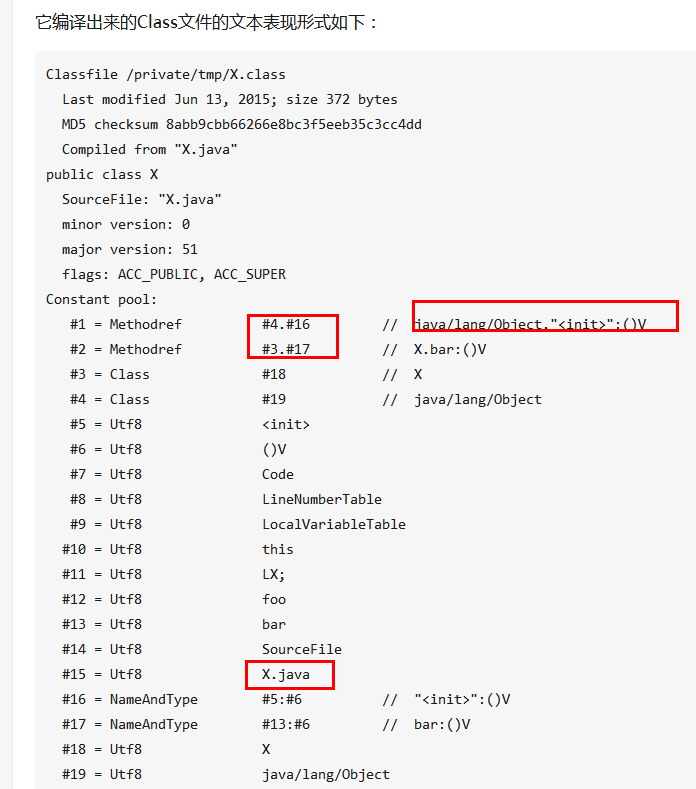


另外，在概念模型中，两个栈帧作为虚拟机栈的元素，相互之间是完全独立的，但是大多数虚拟机的实现里都会作一些优化处理，令两个栈帧出现一部分重叠。让下栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用返回时就可以共用一部分数据，而无须进行额外的参数复制传递了，重叠过程如下图：



3.动态连接：

每个栈帧都包含一个指向**运行时常量池**中该栈帧所属性方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。在Class文件的常量池中存有大量的符号引用，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，这种转化称为静态解析。**另外一部分将在每一次的运行期期间转化为直接引用，这部分称为动态连接**。



细节看知乎链接：https://www.zhihu.com/question/30300585

4.方法返回地址

当一个方法被执行后，有两种方式退出这个方法。第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者(调用当前方法的的方法称为调用者)，是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法方式称为**正常完成出口**(Normal Method Invocation Completion)。

另外一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方式称为**异常完成出口**(Abrupt Method Invocation Completion)。一个方法使用异常完成出口的方式退出，**是不会给它的调用都产生任何返回值的。**

无论采用何种方式退出，在方法退出之前，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PC计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。**

**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栈，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，把返回值(如果有的话)压入调用都栈帧的操作数栈中，调用PC计数器的值以指向方法调用指令后面的一条指令等。

5. 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与高度相关的信息，这部分信息完全取决于具体的虚拟机实现。**在实际开发中，一般会把动态连接，方法返回地址与其它附加信息全部归为一类，称为栈帧信息。**

### 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？（类信息、常量、静态变量、即时编译器编译后的代码）

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（基于通道和缓冲的IO方式。）

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。