JVM栈解决程序的运行问题，即程序如何执行，或者说如何处理数据;JVM堆解决的是数据存储的问题，即数据怎么放、放在哪儿，另外JVM堆中存的是对象。JVM栈中存的是基本数据类型和JVM堆中对象的引用。

**JVM基础概念：JVM堆与JVM栈**

**数据类型**

Java虚拟机中，数据类型可以分为两类：基本类型和引用类型。基本类型的变量保存原始值，即：他代表的值就是数值本身;而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

基本类型包括：byte,short,int,long,char,float,double,Boolean,returnAddress

引用类型包括：类类型，接口类型和数组。

**JVM堆与JVM栈**

JVM堆和JVM栈是程序运行的关键，很有必要把他们的关系说清楚。

JVM栈是运行时的单位，而JVM堆是存储的单位。

JVM栈解决程序的运行问题，即程序如何执行，或者说如何处理数据;JVM堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程JVM栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程JVM栈。而JVM堆则是所有线程共享的。JVM栈因为是运行单位，因此里面存储的信息都是跟当前线程(或程序)相关信息的。包括局部变量、程序运行状态、方法返回值等等;而JVM堆只负责存储对象信息。

**为什么要把JVM堆和JVM栈区分出来呢?JVM栈中不是也可以存储数据吗?**

第一，从软件设计的角度看，JVM栈代表了处理逻辑，而JVM堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，JVM堆与JVM栈的分离，使得JVM堆中的内容可以被多个JVM栈共享(也可以理解为多个线程访问同一个对象)。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，JVM堆中的共享常量和缓存可以被所有JVM栈访问，节省了空间。

第三，JVM栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于JVM栈只能向上增长，因此就会限制住JVM栈存储内容的能力。而JVM堆不同，JVM堆中的对象是可以根据需要动态增长的，因此JVM栈和JVM堆的拆分，使得动态增长成为可能，相应JVM栈中只需记录JVM堆中的一个地址即可。

第四，面向对象就是JVM堆和JVM栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在JVM堆中;而对象的行为(方法)，就是运行逻辑，放在JVM栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

**JVM堆中存什么?JVM栈中存什么?**

JVM堆中存的是对象。JVM栈中存的是基本数据类型和JVM堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在JVM栈中，一个对象只对应了一个4btye的引用(JVM堆JVM栈分离的好处：))。

为什么不把基本类型放JVM堆中呢?因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此JVM栈中存储就够了，如果把他存在JVM堆中是没有什么意义的(还会浪费空间，后面说明)。可以这么说，基本类型和对象的引用都是存放在JVM栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是JVM栈中的数据一个是JVM堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

**Java中的参数传递时传值呢?还是传引用?**

要说明这个问题，先要明确两点：

1.不要试图与C进行类比，Java中没有指针的概念

2.程序运行永远都是在JVM栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后。Java在方法调用传递参数时，因为没有指针，所以它都是进行传值调用(这点可以参考C的传值调用)。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

但是传引用的错觉是如何造成的呢?在运行JVM栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释(或者查找)到JVM堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是JVM堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树(即非叶子节点)，基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点(即一个对象引用)，则可以修改这个节点下面的所有内容。

JVM堆和JVM栈中，JVM栈是程序运行最根本的东西。程序运行可以没有JVM堆，但是不能没有JVM栈。而JVM堆是为JVM栈进行数据存储服务，说白了JVM堆就是一块共享的内存。不过，正是因为JVM堆和JVM栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，JVM栈的大小通过-Xss来设置，当JVM栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时JVM栈中保存的信息都是方法返回的记录点。

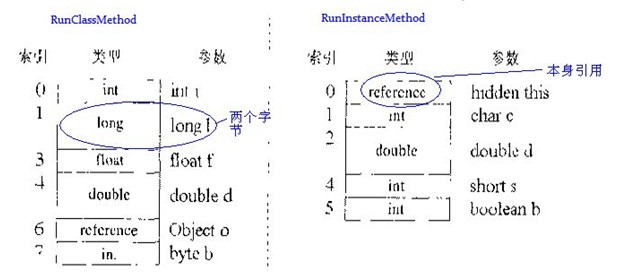
**java栈的组成元素——栈帧**

栈帧由三部分组成：**局部变量区、操作数栈、帧数据区**。局部变量区和操作数栈的大小要视对应的方法而定，他们是按字长计算的。但调用一个方法时，它从类型信息中得到此方法局部变量区和操作数栈大小，并据此分配栈内存，然后压入Java栈。

**局部变量区:**局部变量区被组织为以一个字长为单位、从0开始计数的数组，类型为short、byte和char的值在存入数组前要被转换成int值，而long和double在数组中占据连续的两项，在访问局部变量中的long或double时，只需取出连续两项的第一项的索引值即可,如某个long值在局部变量区中占据的索引时3、4项，取值时，指令只需取索引为3的long值即可。

         说再多也没用，下面就看个例子，好让大家对局部变量区有更深刻的认识。这个图来着《深入JVM》：

public static int runClassMethod(int i,long l,float f,double d,Object o,byte b) {     
        return 0;     
     }     
         
    public int runInstanceMethod(char c,double d,short s,boolean b) {     
        return 0;     
     }



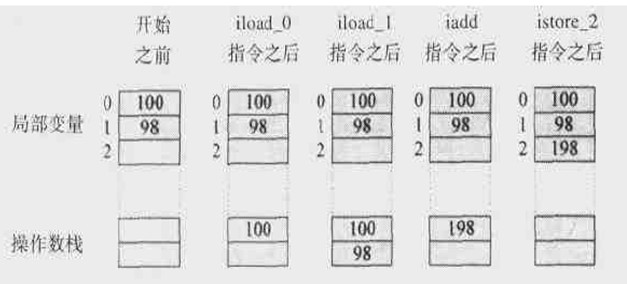
**runInstanceMethod**的局部变量区第一项是个**reference**（引用），它指定的就是对象本身的引用，也就是我们常用的**this,**但是在runClassMethod方法中，没这个引用，那是因为runClassMethod是个静态方法**。**

**操作数栈**和局部变量区一样，操作数栈也被组织成一个以字长为单位的数组。但和前者不同的是，它不是通过索引来访问的，而是通过入栈和出栈来访问的。可把操作数栈理解为存储计算时，临时数据的存储区域。下面我们通过一段简短的程序片段外加一幅图片来了解下操作数栈的作用。

**Int** a= 100;

**Int** b = 98;

**Int** c = a+b;



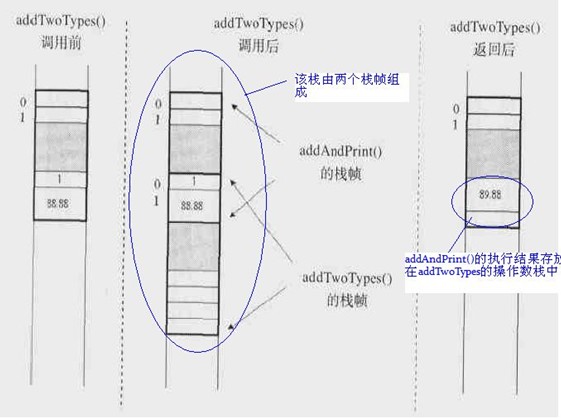
从图中可以得出：操作数栈其实就是个临时数据存储区域，它是通过入栈和出栈来进行操作的。

**帧数据区** 除了局部变量区和操作数栈外，java栈帧还需要一些数据来支持常量池解析、正常方法返回以及异常派发机制。这些数据都保存在java栈帧的帧数据区中。当JVM执行到需要常量池数据的指令时，它都会通过帧数据区中指向常量池的指针来访问它。

除了处理常量池解析外，帧里的数据还要处理java方法的正常结束和异常终止。如果是通过return正常结束，则当前栈帧从Java栈中弹出，恢复发起调用的方法的栈。如果方法又返回值，JVM会把返回值压入到发起调用方法的操作数栈。

为了处理java方法中的异常情况，帧数据区还必须保存一个对此方法异常引用表的引用。当异常抛出时，JVM给catch块中的代码。如果没发现，方法立即终止，然后JVM用帧区数据的信息恢复发起调用的方法的帧。然后再发起调用方法的上下文重新抛出同样的异常。

class Example3C{  
    public static void addAndPrint(){  
        double result = addTwoTypes(1,88.88);  
        System.out.println(result);  
    }  
    public static double addTwoTypes(int i, double d){  
    return i+d;  
    }  
  
}



1.只有在调用一个方法时，才为当前栈分配一个帧，然后将该帧压入栈

2 帧中存储了对应方法的局部数据，方法执行完，对应的帧则从栈中弹出，并把返回结果存储在**调用 方法的帧的操作数栈中**