**参考**

<http://blog.csdn.net/ochangwen/article/details/51476352>

<https://my.oschina.net/u/1156843/blog/203442>

**概述**

JVM规范制定了JVM字节码执行引擎的**概念模型**，从外观上看起来，所有的Java虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。

**虚拟栈**

线程隔离的内存区域，用于描述方法的内存模型：

方法在执行的同时会创建一个栈帧Stack Frame，入栈；

栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息；

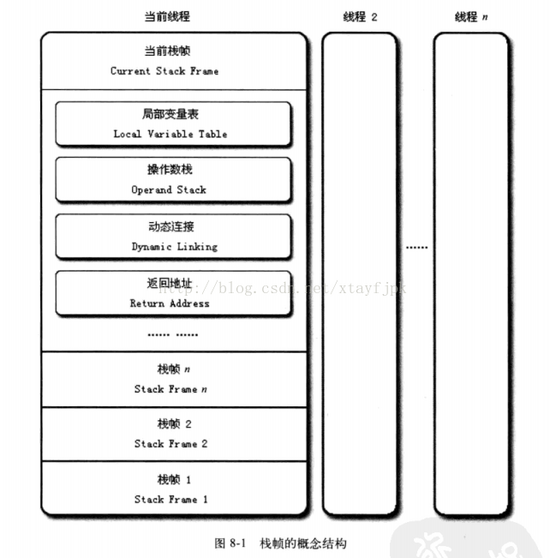
方法执行结束销毁该栈帧，出栈。

**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](http://lib.csdn.net/base/datastructure)**，**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。

    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：



**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。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。

局部变量表的最小单位为Slot。对于基本数据类型，32位数据类型占用1个Slot，64位数据类型占用2个Slot；对于引用类型，不管在32位或64位虚拟机中都占用1个Slot。如果是实例方法，第0个Slot默认存放当前对象实例this或父类对象实例super的引用。

public double inc(){

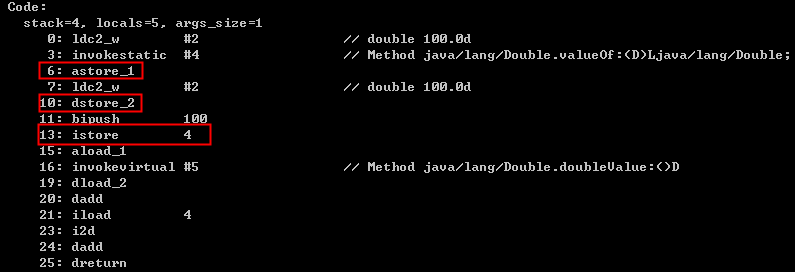
Double a = 100D;

double b = 100D;

int c = 100;

return a + b + c;

}



当前是64位HotSpot虚拟机，根据Code属性可看出：

引用类型a存储到第1个slot，double数据类型b存储到第2、3个slot，int数据类型存储到第4个slot，加上第0个slot的当前实例引用，正好locals=5个slot。

当局部变量的作用域结束，其所占用的slot可以被重用，以节省栈帧空间。

public int inc(){

{

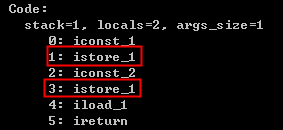
int a = 1;

}

int b = 2;

return b;

}



局部变量a的作用域只在花括号内，可以看出：

a先存储到了第1个slot，接着b也存储到了第1个slot覆盖了a的值。

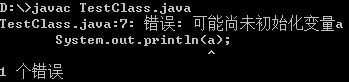
局部变量定义了但没有赋初始值是不能使用的。

public void inc(){

int a;

System.out.println(a);

}



**操作数栈**

Hotspot JVM的执行引擎是基于操作数栈的。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。

对于基本数据类型，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，对于引用类型，不管在32或64位虚拟机中占用的栈容量都为1。在数据空间占用方便与局部变量表一致。

局部变量赋值：

public void inc(){

double a = 100D;

}

stack=2, locals=3, args\_size=1

0: ldc2\_w        #3                  // double 3.0d         //将常量3.0d压入栈顶，double类型占用2个栈容量

3: dstore\_1        //将栈顶的double数据类型3.0d弹出，存储到局部变量表的从索引1开始的slot，double类型占2个slot

4:return //返回void

运算：

public int add(int a,int b) {

return a + b;

}

 stack=2, locals=3, args\_size=3

 0: iload\_1    //将局部变量表的第1个slot的int类型数据压入栈顶

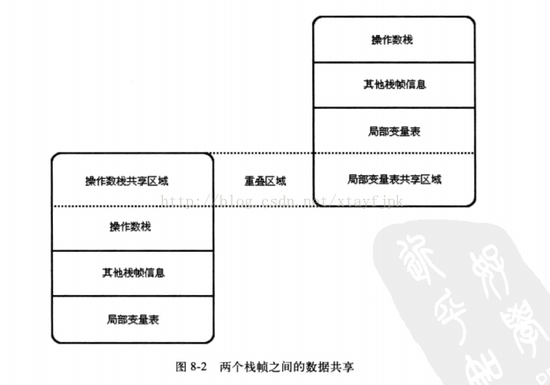
 1: iload\_2   //将局部变量表的第2个slot的int类型数据压入栈顶

  2: iadd     //将栈顶的int类型数据和相邻的int类型数据进行相加，把结果压入栈顶，弹出之前操作数

  3: ireturn   //将栈顶的int类型数据之前返回

栈帧重叠

    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：



**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**。

**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位置，因此在栈帧中必须保存这个位置，通常是调用者的PC计数器的值， 这个值就是方法返回地址。

**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。

所以可以说栈帧分为三部分：局部变量区、操作数栈和帧数据区。