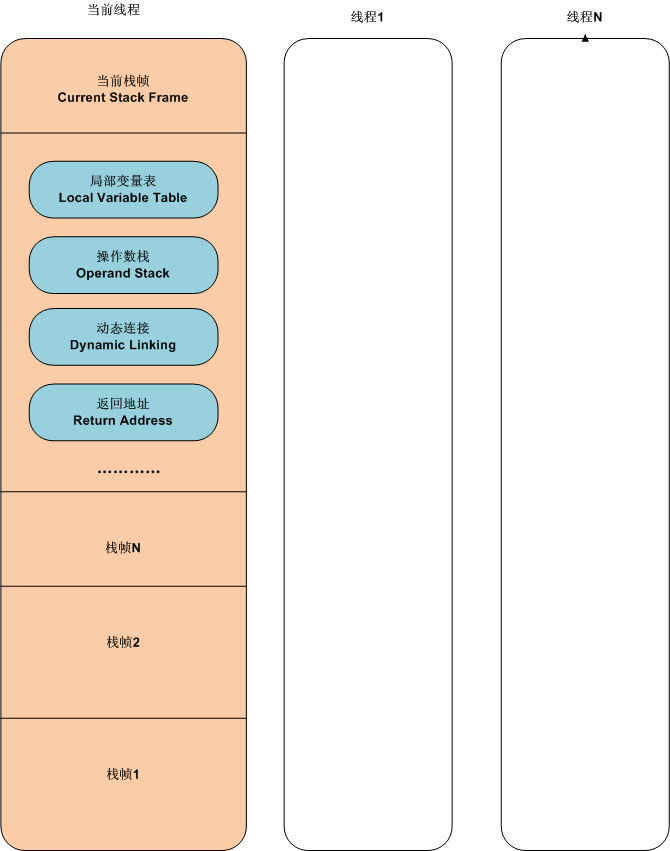
**栈帧、局部变量表、操作数栈**

<https://blog.csdn.net/a616413086/article/details/51272309>

**1.定义**     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中的虚拟机栈的栈元素。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。  
     每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。  
     对于执行引擎来说，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。

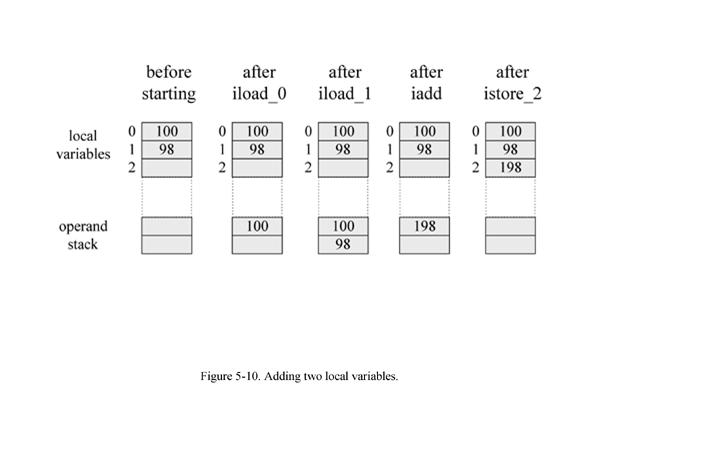


**2.组成** （1）局部变量表  
      局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序被编译成Class文件时，就在方法的Code属性的max\_locals数据项中确定了该方法所需要分配的  
      最大局部变量表的容量。  
      局部变量表的容量以**变量槽（Slot）**为最小单位，32位虚拟机中一个Slot可以存放一个32位以内的数据类型（boolean、byte、char、short、int、float、reference和returnAddress八种）。  
      reference类型虚拟机规范没有明确说明它的长度，但一般来说，虚拟机实现至少都应当能从此引用中直接或者间接地查找到对象在Java堆中的起始地址索引和方法区中的对象类型数据。  
      returnAddress类型是为字节码指令jsr、jsr\_w和ret服务的，它指向了一条字节码指令的地址。  
      虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法（非static），那么局部变量表的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中通过this访问。  
      Slot是可以重用的，当Slot中的变量超出了作用域，那么下一次分配Slot的时候，将会覆盖原来的数据。Slot对对象的引用会影响GC（要是被引用，将不会被回收）。  
  系统不会为局部变量赋予初始值（实例变量和类变量都会被赋予初始值）。也就是说不存在类变量那样的准备阶段。  
 （2）操作数栈  
      Java虚拟机的解释执行引擎被称为"基于栈的执行引擎"，其中所指的栈就是指－操作数栈。  
      操作数栈也常被称为操作栈。  
     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：

**Java代码  收藏代码**

1. begin
2. iload\_0    // push the int in local variable 0 onto the stack
3. iload\_1    // push the int in local variable 1 onto the stack
4. iadd       // pop two ints, add them, push result
5. istore\_2   // pop int, store into local variable 2
6. 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。



例一：

先看4个题目：

①int i = 0;

    i = i++;

②int i = 0;

    i = ++i;

③int i = 0;

    int j = 0;

    j = i++ + i++;

④ int i = 0;

     int j = 0;

     j = i++ + i++ + i++;

每道题里的i和j都是多少？

用myeclipse测试，结果分别是①i = 0，②i = 1，③i = 2，j = 1，④i = 3，j = 3。

i++和++i的问题，困扰很多人。网上、书上的解释各种各样。现在通过分析字节码，来确定这两条语句究竟是怎样执行的。

先给出今天要用到的字节码的含义

|  |  |  |
| --- | --- | --- |
| Bytecode | Stack  before->after | Description |
| iconst\_0 | ->0 | Loads the int value 0 onto the stack |
| istore\_1 | value-> | Store int value into variable 1 |
| istore\_2 | value-> | Store int value into variable 2 |
| iinc | No change | Increment local variable #index by signed byte const |
| iload\_1 | ->value | Loads an int value from variable 1 |
| iadd | value 1,value 2->result | Adds 2 ints together |

说明两点需要注意的地方：

①iinc操作是有参数的，但是在此忽略，简写为iinc，此操作对应于自加操作，并且该操作不对stack有任何改变；

②iadd操作过后只在stack中保留结果result。

接下来是四段程序主要的字节码：

①iconst\_0        ②iconst\_0        ③iconst\_0       ④iconst\_0

    istore\_1            istore\_1            istore\_1           istore\_1

    iload\_1             iinc 1,1              iconst\_0          iconst\_0

    iinc 1,1             iload\_1              istore\_2           istore\_2

    istore\_1            istore\_1            iload\_1             iload\_1

                                                      iinc 1,1             iinc 1,1

                                                      iload\_1             iload\_1

                                                      iinc 1,1             iinc 1,1

                                                      iadd                  iadd

                                                      istore\_2            iload\_1

                                                                               iinc 1,1

                                                                               iadd

                                                                               istore\_2

现在解释①。第一步在stack中存入一个int常量0；第二步把它赋值给第一个变量，即我们的i；第三步把第一个变量i的值存入到stack中；第四步在i自身的空间进行自加，而第三步存入到stack中的值没有变；第五步把第三步存入到stack中的值再赋值给第一个变量i。也就是说，i真的是进行自加了，但是被自己原来的值覆盖掉了。从这里我们可以看出，自加操作比赋值操作的优先级高。

再看②。和①比较起来，区别就在于，++i是先进行自加，然后把自加后的值存入到stack中，所以最后赋值给i的是1，不是0。

③呢？首先，③比前两个多了一个变量，但这不是重点。往下看，把第一个变量i的值（0）存入到stack中，i自加，再第一个变量i的值（1）存入到stack中，再自加。此时stack中有两个值了，0和1，进行加操作后的结果是1，然后赋值给第二个变量j。这里颠覆了我们以前被告知的，加的运算优先级比自加高。这就是困扰我们的，其实，只要把值读入到stack中，接下来就进行自加，只有得到了两个参数，才进行加操作。

来看④。前面都和③一样，只到进行了第一次加操作，得到的结果（1）保存在stack中，没有赋值给j，然后再次读入i的值（2）到stack，i自加，得到加操作的两个参数，1和2，进行第二次加操作的结果是3，赋值给j。

到此，问题都解决了。

例2：

# 从字节码角度分析 i++ 和 ++i 实现

# 内容导读

话虽这么说，那怎么来证明上面两点呢？万物皆有始有终，对于代码，只能从字节码的角度看看是不是能挖掘有价值的信息。每个栈帧对应一个方法，当线程执行方法时，就是栈帧出栈入栈的过程。每个栈帧包含三部分数据：本地变量（参数+方法内的变量）、操作数栈和其它数据，本文主要涉及本地变量和操作数栈。可以发现变量a在执行iinc 1,1的时候已经变成1了，但是istore\_1又把变量a所在位置覆盖成了0，所以执行完i = i++，i还是原来那个值。和后置++不同的地方在于，在变量进入操作数栈之前，就先执行了iinc指令，所以进入操作数的值是加1后的值，最后写回的值也是最新值。

在知识星球中，有小伙伴提问，最近看到个面试题：

int j=0;for(int i=0;i<100;i++) j = j++;System.out.println(j);

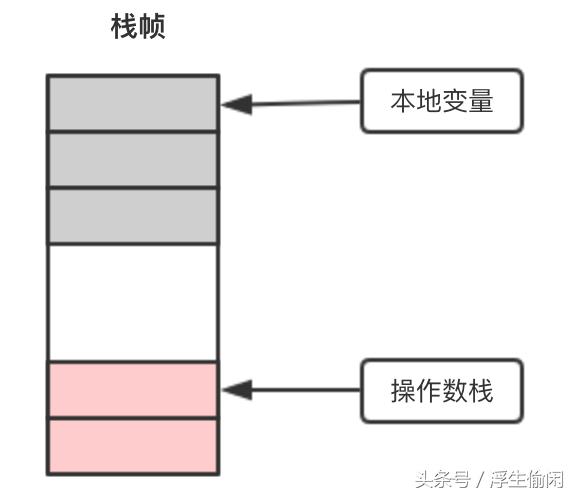
输出结果是0，如果换成j++，那么输出100，这是为什么？

对于这种问题，其实有点经验的程序员都知道，前置++和后置++的运算区别：  
1、前置++是将自身加1的值赋值给新变量，同时自身也加1  
2、后置++是将自身的值赋给新变量，然后才自身加1

话虽这么说，那怎么来证明上面两点呢？  
万物皆有始有终，对于代码，只能从字节码的角度看看是不是能挖掘有价值的信息。

在讲字节码之前，先简单的了解下Java栈，在JVM中有这么一个数据结构叫Java栈，当线程启动的时候，会分配一块内存当做该线程的栈，每个栈由一系列的栈帧组成。每个栈帧对应一个方法，当线程执行方法时，就是栈帧出栈入栈的过程。

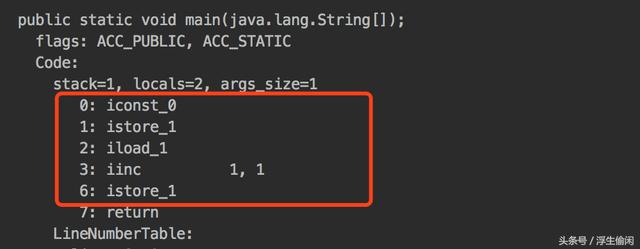
每个栈帧包含三部分数据：本地变量（参数+方法内的变量）、操作数栈和其它数据，本文主要涉及本地变量和操作数栈。



先看看后置++的实现：

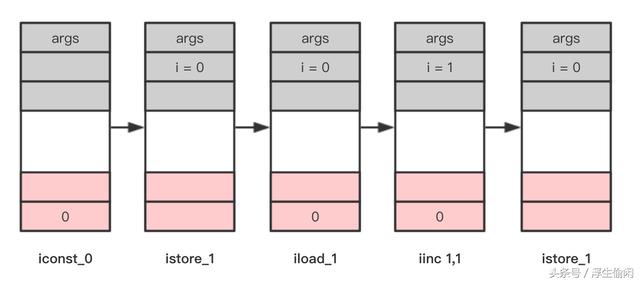
public static void main(String[] args){ int i= 0; i = i++;}

编译之后的字节码如下：



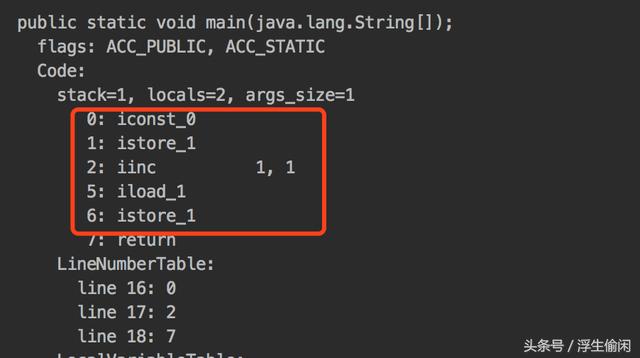
iconst\_0 //把数值0 push到操作数栈istore\_1 // 把操作数栈写回到本地变量第2个位置iload\_1 // 把本地变量第2个位置的值push到操作数栈iinc 1,1 // 把本地变量表第2个位置加1 istore\_1 // 把操作数据栈写回本地变量第2个位置

整个过程这样的：

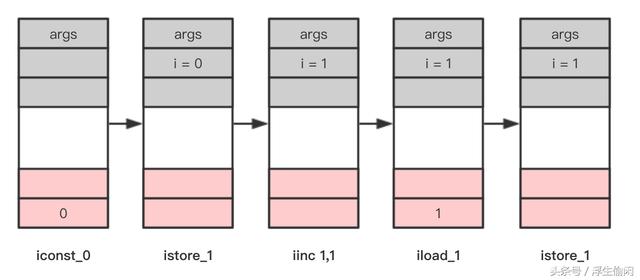


可以发现变量a在执行iinc 1,1的时候已经变成1了，但是istore\_1又把变量a所在位置覆盖成了0，所以执行完i = i++，i还是原来那个值。

另外，来看下前置++的实现：



字节码就不再解释了，整个过程实现如下：



和后置++不同的地方在于，在变量进入操作数栈之前，就先执行了iinc指令，所以进入操作数的值是加1后的值，最后写回的值也是最新值。