# 虚拟机栈的背景

**为什么jvm指令的执行都是基于栈结构？**

为了实现跨平台特性，基于寄存器的因为不同平台CPU架构不同从而无法实现跨平台特性，基于栈可实现跨平台。

**基于栈结构设计指令执行引擎的优缺点？**

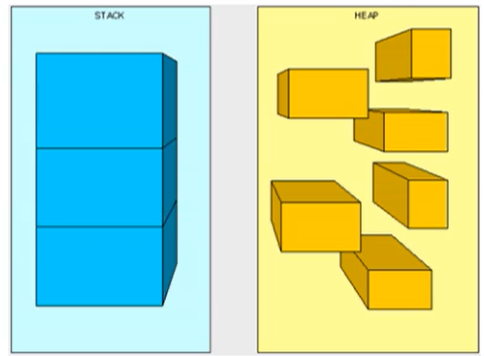
优点是跨平台，指令集小，编译器容易实现

缺点是性能下降，实现同样的功能需要更多的指令。

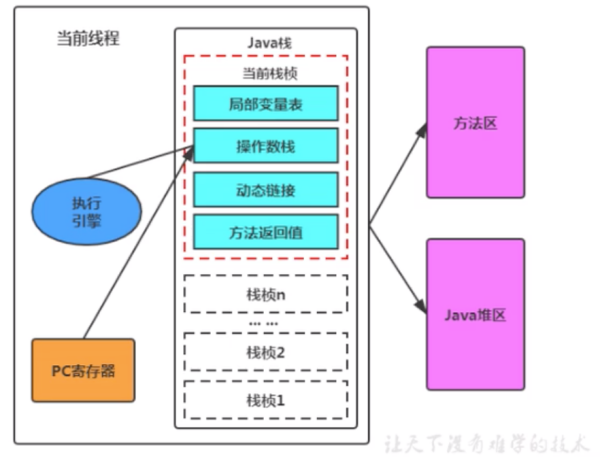
# 内存中的堆与栈

* **如何理解栈是运行时的单位，而堆是存储的单位？**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。

堆解决的是数据存储的问题，即数据怎么放、放在哪儿。



## 虚拟机栈



### (1)什么是虚拟机栈

java虚拟机栈（Java Virtual Machine Stack），也叫Java栈。 每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame）

### **(2)虚拟机栈的作用**

主管Java程序的运行.

### **Java虚拟机栈的特点**

* 生命周期和线程一致，也就是线程结束了，该虚拟机栈也销毁了
* 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器。
* JVM直接对Java栈的操作只有两个：

1. 每个方法执行，伴随着进栈（入栈、压栈）
2. 执行结束后的出栈工作

* 不存在垃圾回收问题，但是肯定存在OOM异常



### **栈中可能出现的异常**

* **前提：**java虚拟机规范允许Java栈的大小是动态的或者是固定不变的
* Case1: 如果采用固定大小的Java虚拟机栈，那每一个线程的java虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过java虚拟机栈允许的最大容量，java虚拟机将会抛出一个 StackOverFlowError异常
* Case2: 如果java虚拟机栈可以动态拓展，并且在尝试拓展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那java虚拟机将会抛出一个 OutOfMemoryError异常
* **栈异常演示**

|  |
| --- |
| public class StackErrorTest {  private static int *count* = 1;  public static void main(String[] args) {  System.*out*.println(*count*);  *count*++;  *main*(args);  } } |

### **设置栈的内存大小**

* -Xss

我们可以使用参数-Xss选项来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可达深度。

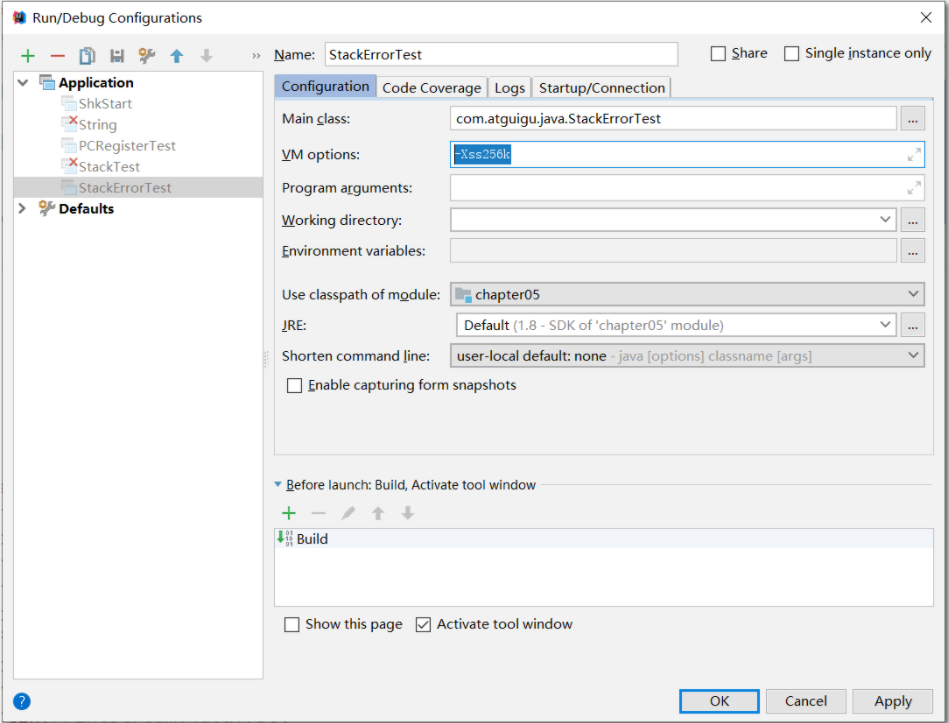
-Xss1024m // 栈内存为 1024MBS

-Xss1024k // 栈内存为 1024KB

* IDEA设置方法：

Run->EditConfigurations->VM options 填入指定栈的大小-Xss256k

* 修改栈大小测试



再次执行：

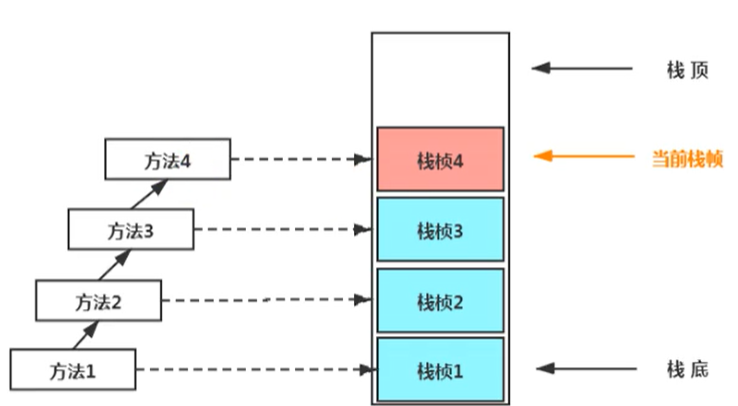
|  |
| --- |
| public class StackErrorTest {  private static int *count* = 1;  public static void main(String[] args) {  System.*out*.println(*count*);  *count*++;  *main*(args);  } }    只打印到2461就栈溢出了 |

## 虚拟机栈的存储结构和运行原理

### 栈帧

1. 栈中的数据都是以栈帧（Stack Frame）的格式存在
2. 在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）。
3. 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。

### **栈的运行原理**



1.JVM直接对Java栈的操作只有两个，就是对栈帧的压栈和出栈，遵循先进后出（后进先出）原则

2.在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的

* 这个栈帧被称为当前栈帧（Current Frame）
* 与当前栈帧相对应的方法就是当前方法（Current Method）
* 定义这个方法的类就是当前类（Current Class）

3.执行引擎运行的所有字节码指令只针对当前栈帧进行操作。

4.如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前帧。

5.不同线程中所包含的栈帧是不允许存在相互引用的，即不可能在一个栈帧之中引用另外一个线程的栈帧。

6.如果当前方法被其他方法调用了，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。

7.Java方法有两种返回函数的方式，但不管使用哪种方式，都会导致栈帧被弹出：

一种是正常的函数返回，使用return指令

另外一种是抛出异常

* 代码演示：

|  |
| --- |
| public class StackFrameTest {  public static void main(String[] args) {  StackFrameTest test = new StackFrameTest();  test.method1();  }   public void method1() {  System.*out*.println("method1()开始执行...");  method2();  System.*out*.println("method1()执行结束...");  }   public int method2() {  System.*out*.println("method2()开始执行...");  int i = 10;  int m = (int) method3();  System.*out*.println("method2()即将结束...");  return i + m;  }   public double method3() {  System.*out*.println("method3()开始执行...");  double j = 20.0;  System.*out*.println("method3()即将结束...");  return j;  } }  method1()开始执行...  method2()开始执行...  method3()开始执行...  method3()即将结束...  method2()即将结束...  method1()执行结束... |

## 虚拟机栈的相关面试题

1.举例栈溢出的情况？（StackOverflowError）

递归调用，通过-Xss设置栈的大小；

2.调整栈的大小，就能保证不出现溢出么？

不能 ,如递归无限次数肯定会溢出，调整栈大小只能保证溢出的时间晚一些，极限情况会导致OOM内存溢出（Out Of Memery Error）

3.分配的栈内存越大越好么？

不是 会挤占其他线程的空间

4.垃圾回收是否会涉及到虚拟机栈？

不会

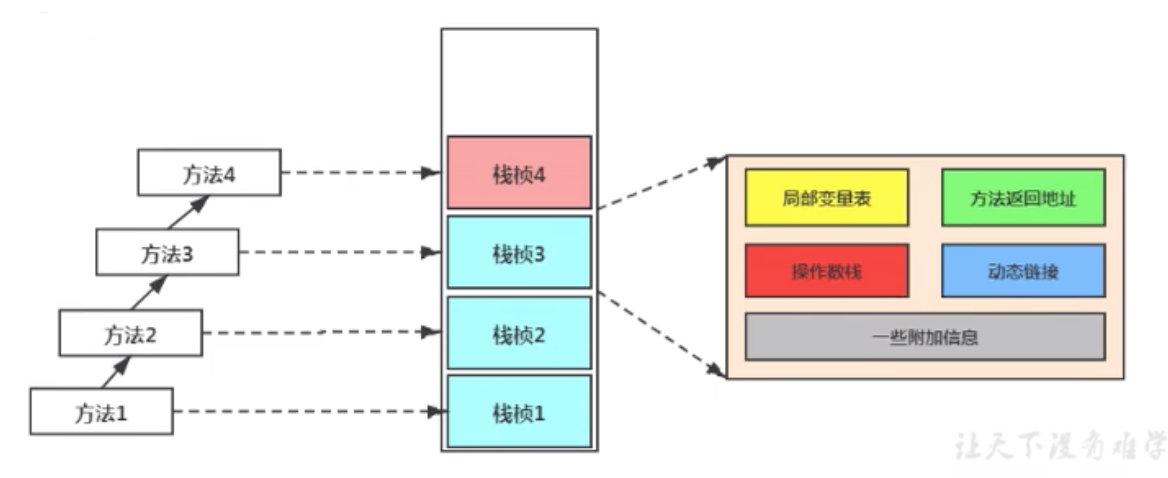


# 3.栈帧

## 栈帧内部结构

每个栈帧中存储着：

1. 局部变量表（Local Variables）
2. 操作数栈（Operand Stack）（或表达式栈）
3. 动态链接（Dynamic Linking）（或指向运行时常量池的方法引用）
4. 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
5. 一些附加信息





并行的每个线程下的栈都是私有的，因此每个线程都有自己各自的栈，并且每个栈里面都有很多栈帧，**栈帧的大小主要由局部变量表和操作数栈决定的**



### 1.局部变量表（Local Variables）

1. 局部变量表也被称之为局部变量数组或本地变量表，底层是一个数组，

**2. 存储内容**

方法参数、局部变量、返回值类型

基本数据类型、对象引用（reference）、以及returnAddress类型

局部变量就是方法内定义的变量，那局部变量表指的就是该栈帧内所有的局部变量。局部变量的类型包含：8种基本数据类型、对象的引用；如果是基本数据类型，那么存储的就是值，如果是引用类型，存储的是对象地址。

1. **线程安全**

局部变量表是建立在线程的栈上，是线程私有的数据，不存在数据安全问题

1. **大小**

局部变量表所需的容量大小是在编译期确定下来的，并**保存在Class文件方法的Code属性的maximum local variables数据项中**，在方法运行期间是不会改变局部变量表的大小的

5.方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，他的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越大，以满足方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间。

6**.局部变量表的作用范围和生命周期**

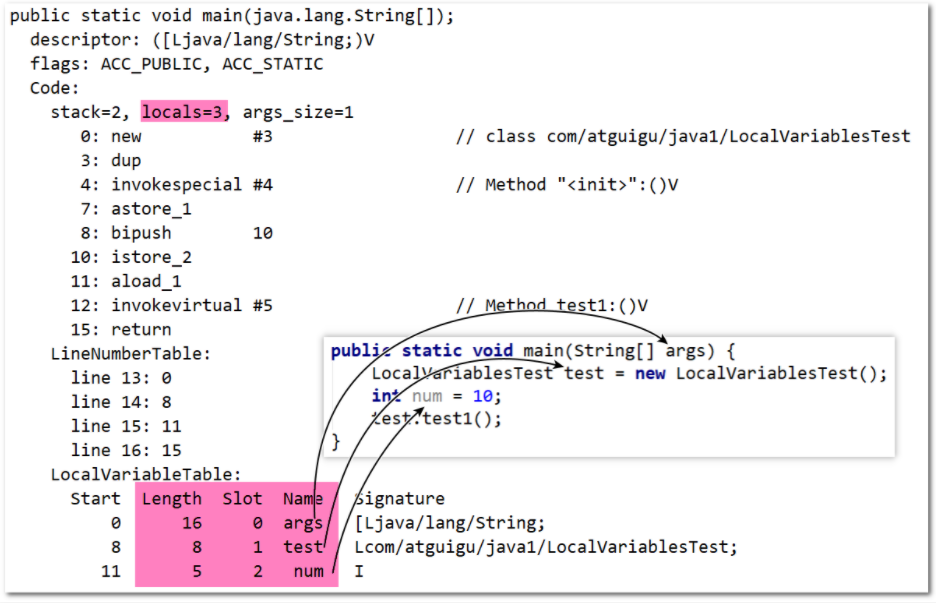
局部变量表中的变量只在当前方法调用中有效。

在方法执行时，虚拟机通过使用局部变量表完成参数值到参数变量列表的传递过程；当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。

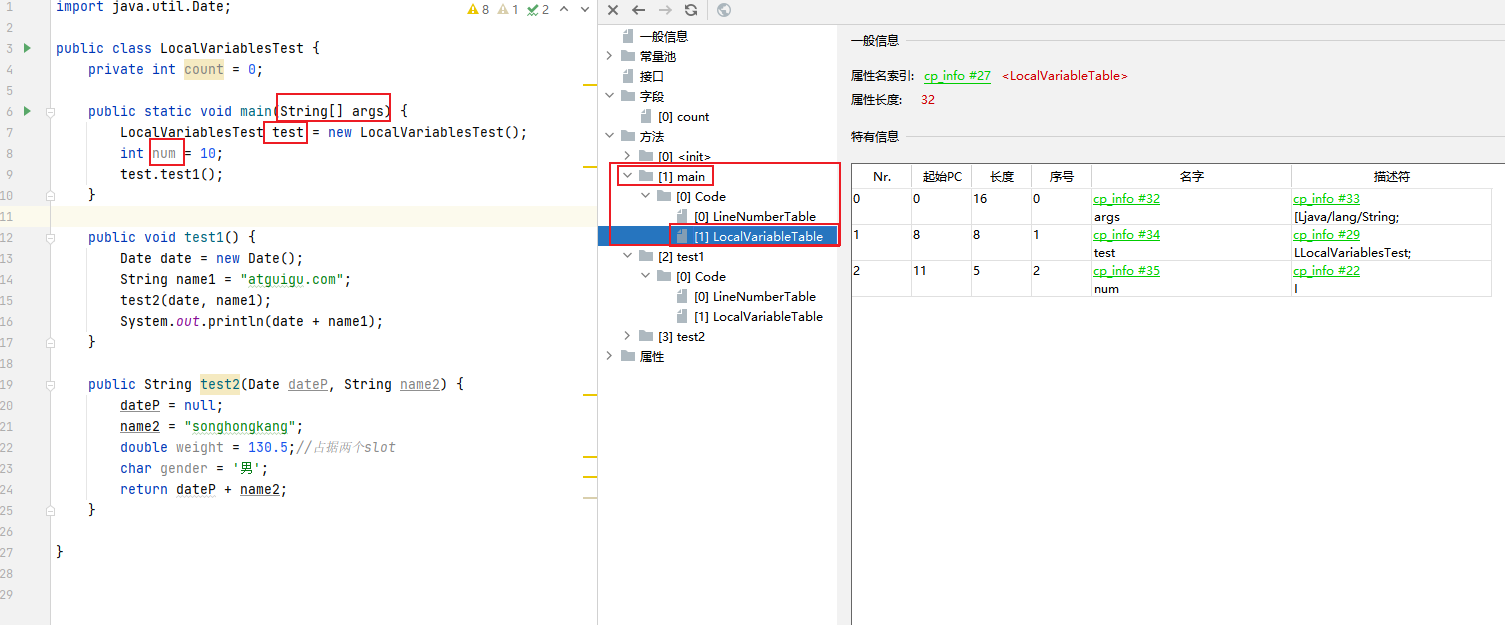
#### 实验：查看局部变量表

|  |
| --- |
| import java.util.Date;  public class LocalVariablesTest {  private int count = 0;   public static void main(String[] args) {  LocalVariablesTest test = new LocalVariablesTest();  int num = 10;  test.test1();  }   public void test1() {  Date date = new Date();  String name1 = "atguigu.com";  test2(date, name1);  System.*out*.println(date + name1);  }   public String test2(Date dateP, String name2) {  dateP = null;  name2 = "songhongkang";  double weight = 130.5;*//占据两个slot* char gender = '男';  return dateP + name2;  }  } |

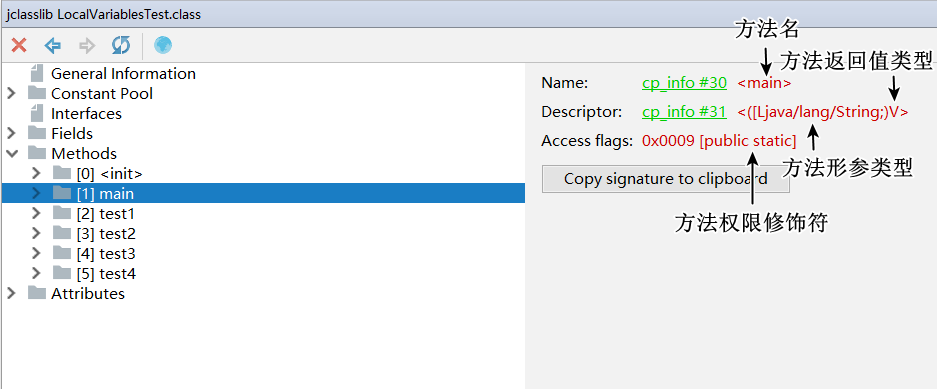
Javap查看字节码文件：



也可以在IDEA 上安装jclasslib byte viewcoder插件查看方法内部字节码信息剖析：



（1）字节码中查看方法的声明信息



[Ljava/lang/String] ：

[] 表示数组

L 表示引用类型

java/lang/String 表示 java.lang.String

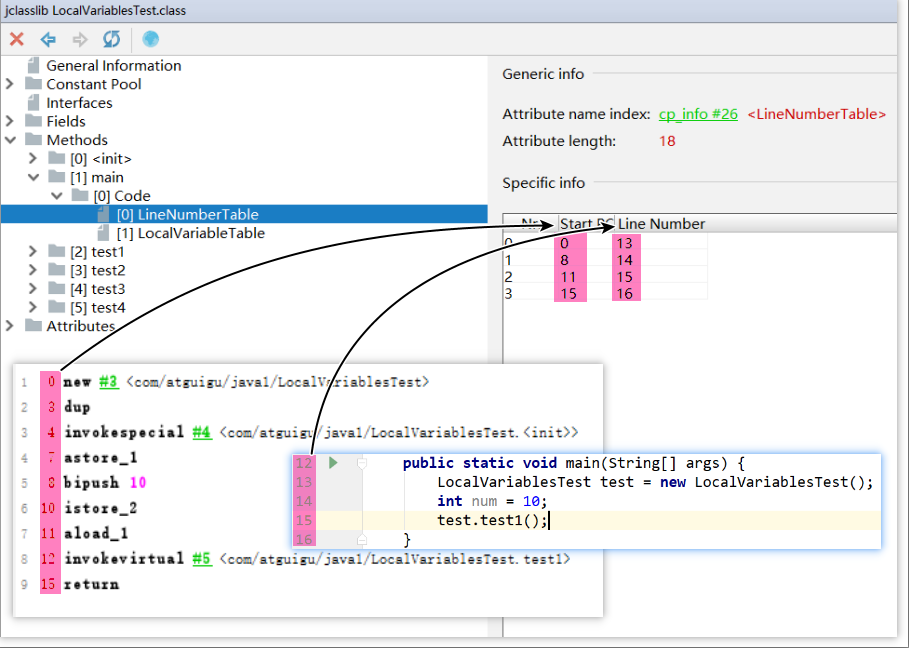
（2）查看方法的字节码长度为 16（0~15）



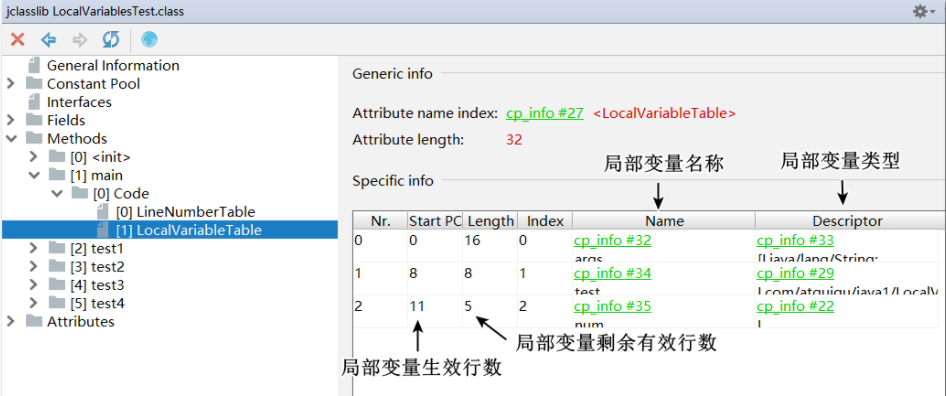
* 方法异常信息表
* 杂项（Misc）



（3）字节码指令行号和原始 java 代码行号的对应关系



（4）注意：生效行数和剩余有效行数都是针对于字节码文件的行数



LocalVariableTable中的信息有：

1. 方法的局部变量个数，名字，类型
2. 生命周期：

startPc：该局部变量从哪个字节码指令开始的

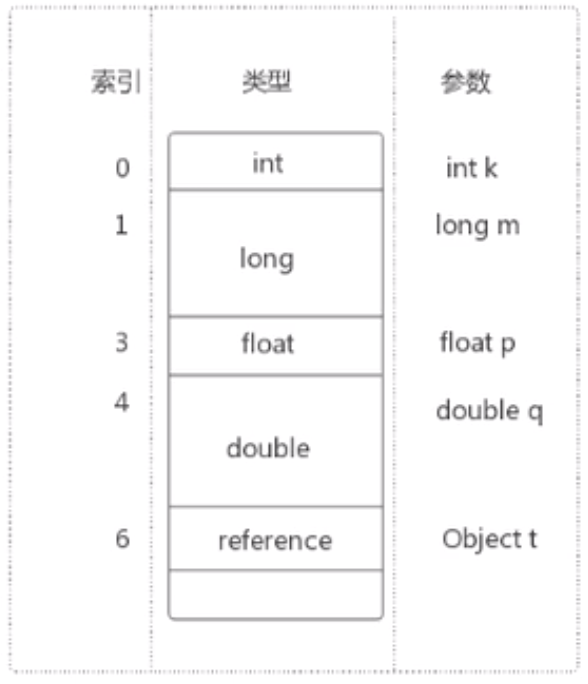
Length： 该局部变量在字节码指令中存活的长度

3.符号引用

绿的下划线的：是该局部变量对应在运行时常量池中的符号引用

#### 变量槽slot的理解与演示

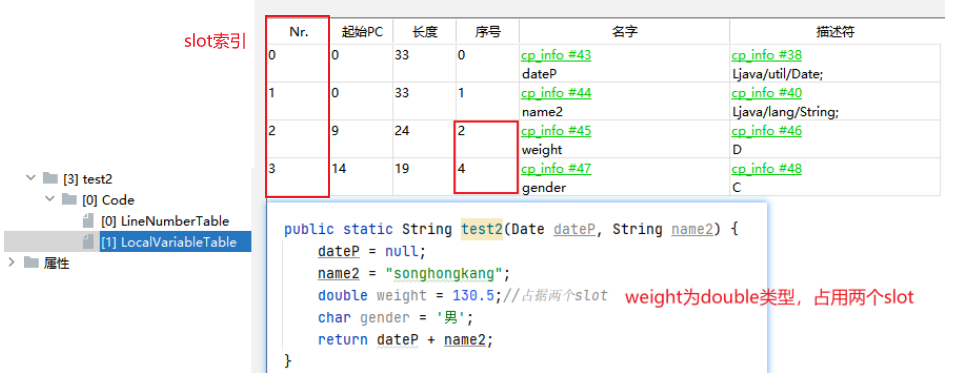
1. 局部变量表，最基本的存储单元是Slot(变量槽)
2. 参数值的存放总是在局部变量数组的index=0开始，到数组长度-1的索引结束



1. 局部变量表中存放编译期可知的各种基本数据类型（8种），引用类型（reference），returnAddress类型的变量。
2. 在局部变量表里，32位以内的类型只占用一个slot（包括引用类型和returnAddress类型），64位的类型（long和double）占用2个slot。

byte、short、char、float在存储前被转换为int，boolean也被转换为int，0表示false，非0表示true；

1. JVM会为局部变量表中的每一个slot都分配一个访问索引，通过这个索引即可成功访问到局部变量表中指定的局部变量值

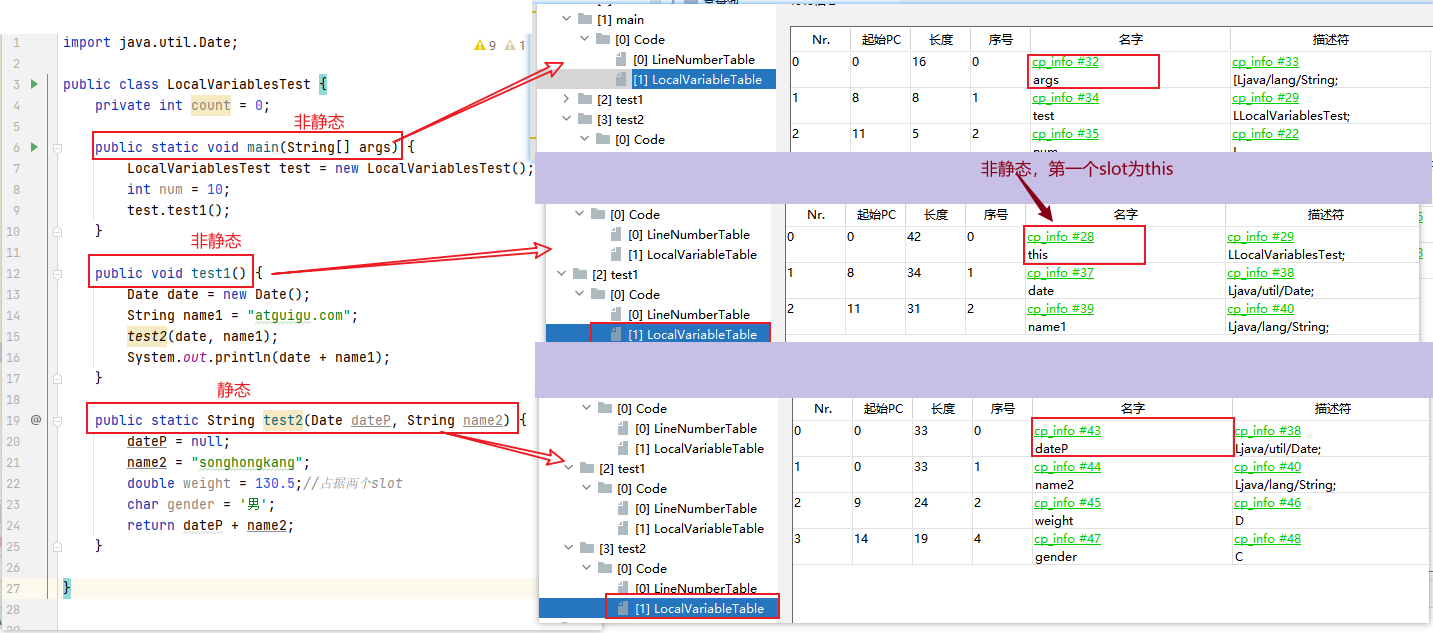


6.当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照声明顺序被复制到局部变量表中的每一个slot上

7.如果需要访问局部变量表中一个64bit的局部变量值时，只需要使用前一个索引即可。（比如：访问long或者double类型变量）

8.如果当前帧对应的方法是构造方法或者非静态方法，那么该对象引用this将会存放在index为0的slot处,其余的参数按照参数表顺序排列。

* 示例代码：



9.静态方法中不能引用this，是因为静态方法所对应的栈帧当中的局部变量表中不存在this。所以静态方法不能用this和super

|  |
| --- |
| public class LocalVariablesTest {  private int count = 1;  //静态方法不能使用this  public static void testStatic(){  //编译错误，因为this变量不存在与当前方法的局部变量表中！！！  System.out.println(this.count);  }  } |

#### slot的重复利用

栈帧中的局部变量表中的槽位是可以重复利用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。

|  |
| --- |
| private void test3() {  int a = 0;  {  int b = 0;  b = a+1;  }  //变量c使用之前以及经销毁的变量b占据的slot位置  int c = a+1;  }      只用了3个slot;  this占用0号slot,a占用1号，b占用2号，c重复利用了2号  因为b的终止pc为8，而c的起始pc为12，因此c可以重复利用2号slot |

#### 静态变量与局部变量的对比

**变量的分类：**

* 按照数据类型分：

1.基本数据类型;

2.引用数据类型；

* 按照在类中声明的位置分：

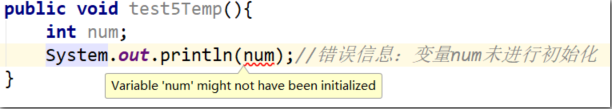
1.成员变量：在使用前，都经历过默认初始化赋值

* + - Stati修饰(类变量)：类加载阶段中第二过程链接中的准备阶段给类变量默认赋0值——>初始化阶段给类变量显式赋值即静态代码块赋值；
    - 不被static修饰(实例变量)：随着对象的创建，会在堆空间分配实例变量空间，并进行默认赋值

2.局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过

和成员变量初始化不同，局部变量表不存在系统初始化的过程，这意味着一旦定义了局部变量则必须人为的初始化，否则无法使用。

* 代码演示：



* 补充说明：

1.关于栈帧的性能调优

在栈帧中，与性能调优关系最为密切的部分就是局部变量表。在方法执行时，虚拟机使用局部变量表完成方法的传递

2.局部变量表中的变量也是GC Root

局部变量表中的变量也是重要的垃圾回收根节点（GC Roots），只要被局部变量表中直接或间接引用的对象都不会被回收。

### 2.操作数栈（Operand Stack）

每一个独立的栈帧除了包含局部变量表以外，还包含一个后进先出（Last - In - First -Out）的 操作数栈，也可以称之为表达式栈（Expression Stack）

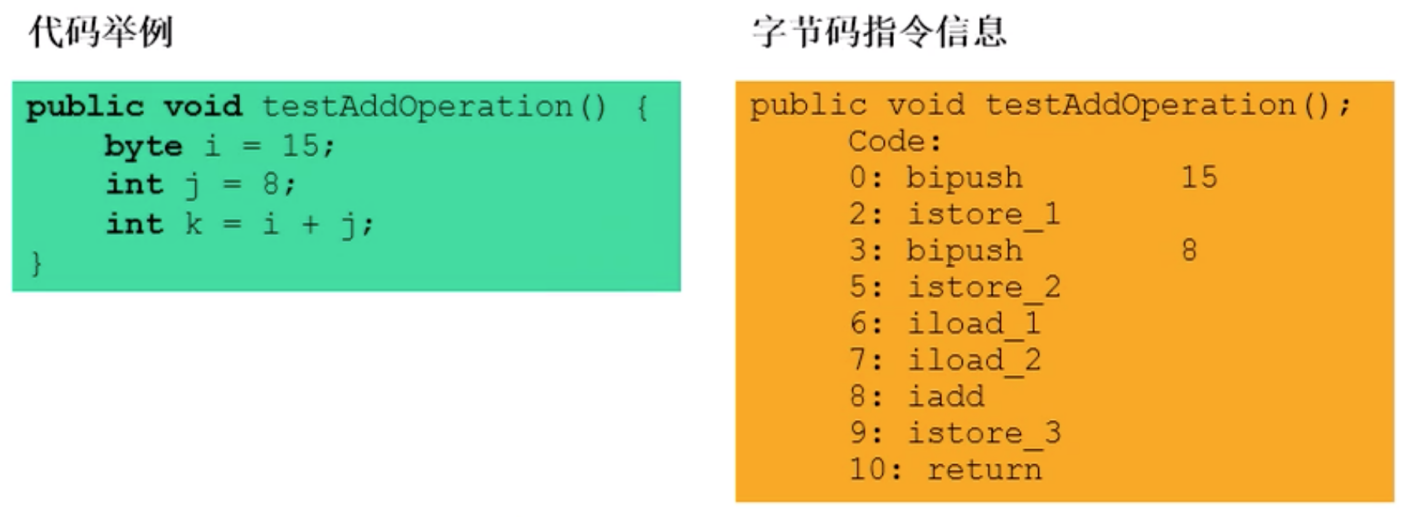
#### 操作数栈的特点

操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈（push）和 出栈（pop）

某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈，比如：执行复制、交换、求和等操作

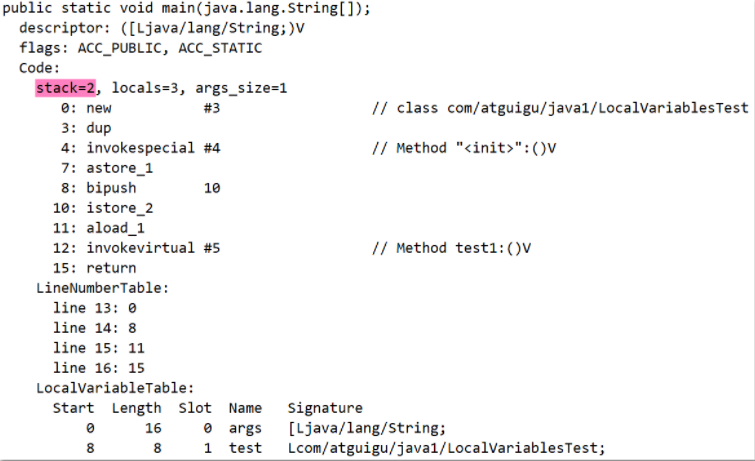
**代码举例：**

左边为 java 源代码，右边为 java 代码编译生成的字节码指令



1. 操作数栈，主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。
2. 操作数栈就是jvm执行引擎的一个工作区，当一个方法开始执行的时候，一个新的栈帧也会随之被创建出来，这个方法的操作数栈是空的
3. 每一个操作数栈都会拥有一个明确的栈深度用于存储数值，其所需的最大深度在编译器就定义好了，保存在class文件中方法的code属性中，为max\_stack的值。
4. 栈中的任何一个元素都是可以任意的java数据类型
   1. 32bit的类型占用一个栈单位深度
   2. 64bit的类型占用两个栈深度单位
5. 操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标准的入栈push和出栈pop操作来完成一次数据访问
6. 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。
7. 操作数栈中的元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译期间进行验证，同时在类加载过程中的类验证阶段的数据流分析阶段要再次验证。
8. 另外，我们说Java虚拟机的**解释引擎是基于栈的执行引擎,其中的栈指的就是操作数栈。**

**操作数栈的深度：**



#### 操作数栈代码追踪

public void testAddOperation() {

*//byte、short、char、boolean：都以int型来保存*

byte i = 15;

int j = 8;

int k = i + j;

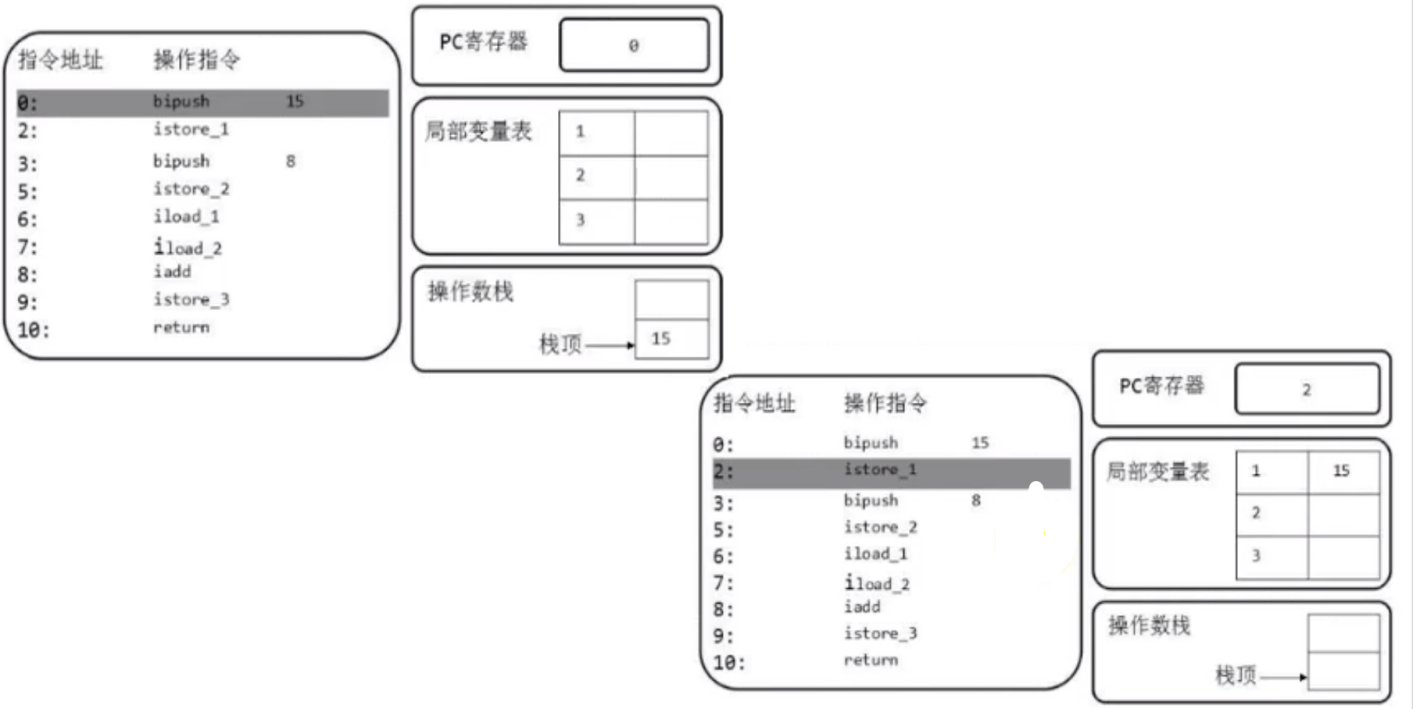
}

结合下面的图来看一下一个方法（栈帧）的执行过程：

bipush:压入操作数栈

Istore: 把操作数栈顶元素存进局部变量表中 i表示int类型，必须和被操作数保持类型一致。

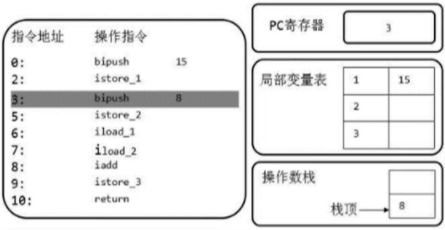
1. 首先执行第一条语句，PC寄存器指向的是0，也就是指令地址为0，然后使用bipush让操作数15入操作数栈。

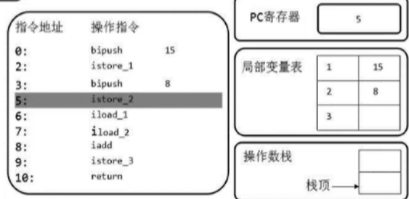


1. 执行完后，让PC + 1，指向下一行代码，下一行代码就是将操作数栈的元素存储到局部变量表1的位置，我们可以看到局部变量表的已经增加了一个元素

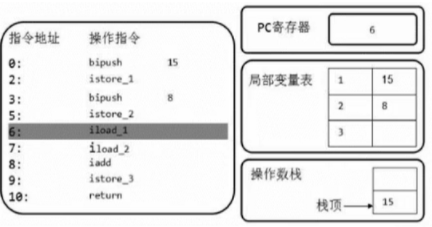
解释为什么局部变量表索引从 1 开始，因为该方法为实例方法，局部变量表索引为 0 的位置存放的是 this

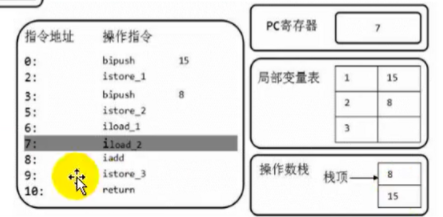
1. 然后PC+1，指向的是下一行。让操作数8也入栈，同时执行store操作，存入局部变量表中



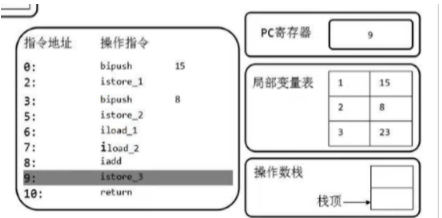
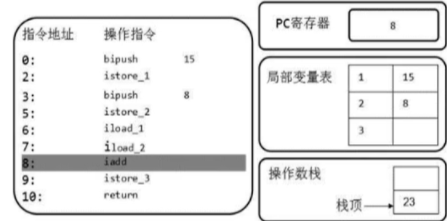


1. 然后从局部变量表中，依次将数据放在操作数栈中，等待执行 add 操作



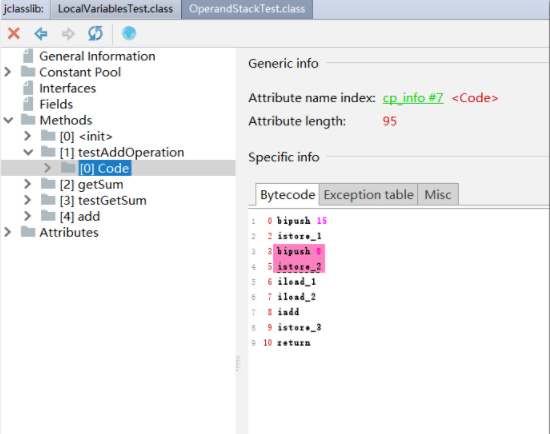


1. 然后将操作数栈中的两个元素执行相加操作，并存储在局部变量表3的位置



* **关于 int j = 8; 的说明**

我们反编译得到的字节码指令如下



因为 8 可以存放在 byte 类型中，所以压入操作数栈的类型为 byte ，而不是 int ，所以执行的字节码指令为 bipush 8

然后执行将数值 8 存放在 int 类型的变量中：istore\_2

* **关于调用方法，返回值入操作数栈的说明**

public int getSum(){

int m = 10;

int n = 20;

int k = m + n;

return k;

}

public void testGetSum(){

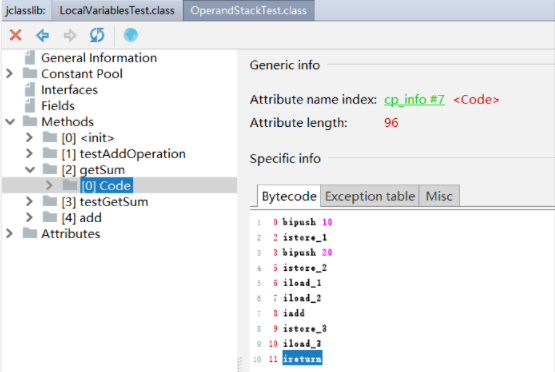
//获取上一个栈桢返回的结果，并保存在操作数栈中

int i = getSum();

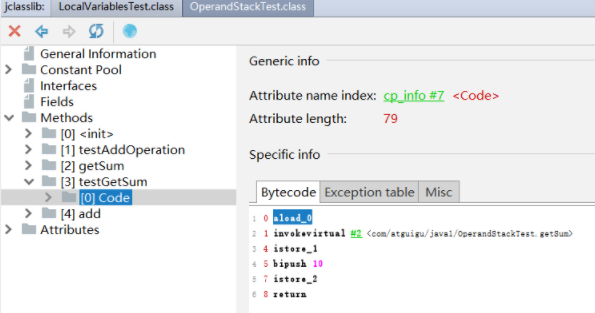
int j = 10;

}

getSum() 方法字节码指令：最后带着个 ireturn



testGetSum() 方法字节码指令：一上来就加载 getSum() 方法的返回值



#### ++i 与 i++ 的区别

public void add(){

//第1类问题：

int i1 = 10;

i1++;

int i2 = 10;

++i2;

//第2类问题：

int i3 = 10;

int i4 = i3++;

int i5 = 10;

int i6 = ++i5;

//第3类问题：

int i7 = 10;

i7 = i7++;

int i8 = 10;

i8 = ++i8;

//第4类问题：

int i9 = 10;

int i10 = i9++ + ++i9;

}

* **i++**

java 源代码

|  |
| --- |
| //第2类问题：  int i3 = 10;  int i4 = i3++; |

字节码指令：

bipush 10 ：将 10 压入操作数栈

istore\_3 ：将操作数栈中的 10 保存到变量 i3 中

iload\_3 ：将变量 i3 的值（10）加载至操作数栈中

iinc 3 by 1：变量 i3 执行 +1 操作

istore 4：将操作数栈中的值保存至变量 i4 中（10）

* **++i**

java 源代码

int i5 = 10;

int i6 = ++i5;

字节码指令

bipush 10 ：将 10 压入操作数栈

istore 5 ：将操作数栈中的 10 保存到变量 i5 中

iinc 5 by 1：变量 i5 执行 +1 操作

iload 5 ：将变量 i5 的值（11）加载至操作数栈中

istore 6：将操作数栈中的值保存至变量 i6 中（11）

总结：

i++：先将 i 的值加载到操作数栈，再将 i 的值加 1

++i：先将 i 的值加 1，在将 i 的值加载到操作数栈

#### 栈顶缓存技术ToS（Top-of-Stack Cashing）

**优缺点：**基于栈式架构的虚拟机所使用的**零地址指令**（即不考虑地址，单纯入栈出栈）**更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数**

由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题**，HotSpot JVM的设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理CPU的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率**

### 3.动态链接

动态链接（或指向运行时常量池的方法引用）