# 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？**

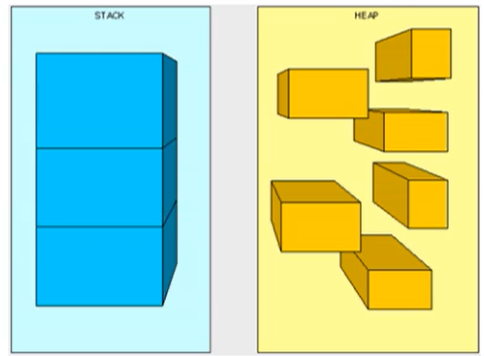
优点是跨平台，指令集小，编译器容易实现

缺点是性能下降，实现同样的功能需要更多的指令。

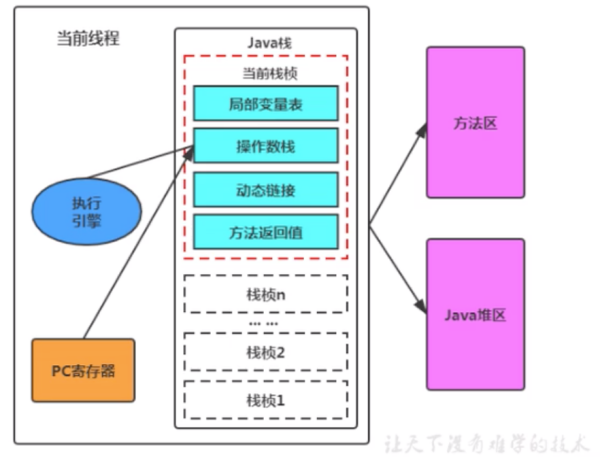
# 内存中的堆与栈

* **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。

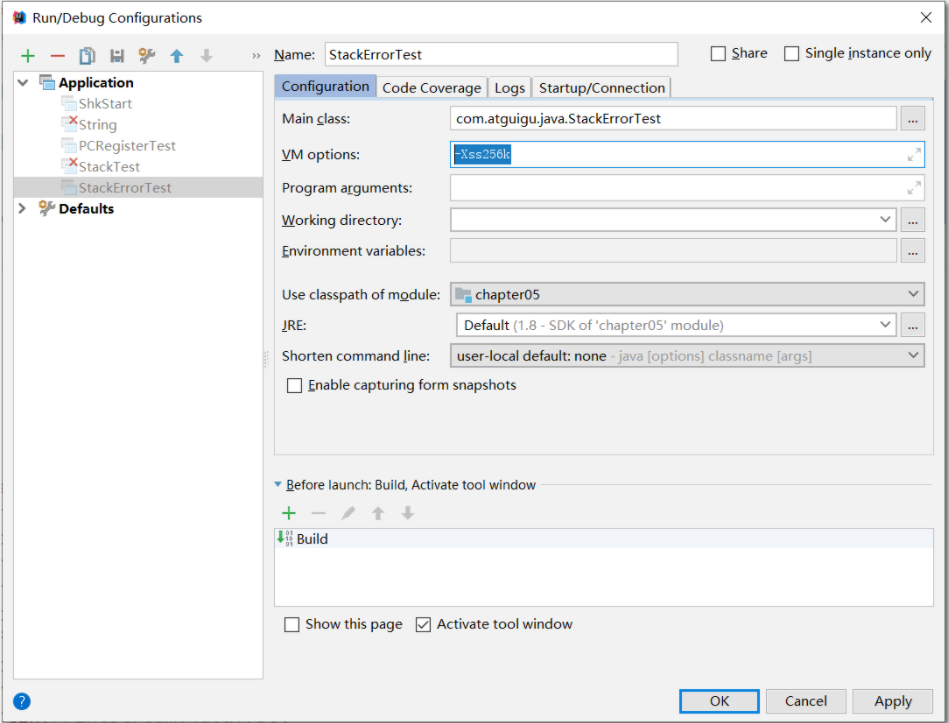
-Xss1024m // 栈内存为 1024MBS

-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：

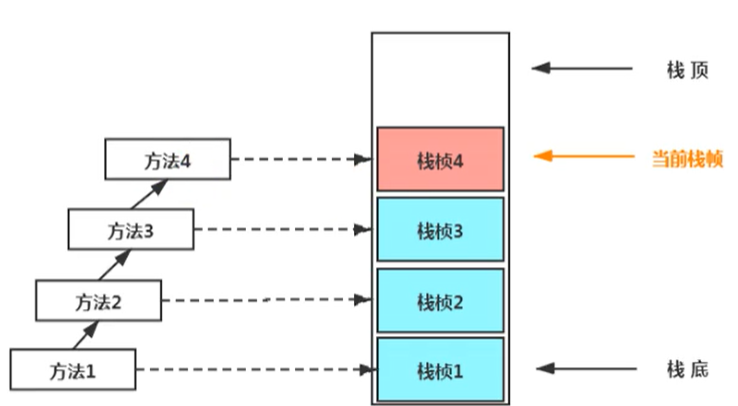
|  |
| --- |
| public class StackErrorTest {  private static int *count* = 1;  public static void main(String[] args) {  System.*out*.println(*count*);  *count*++;  *main*(args);  } }    只打印到2461就栈溢出了 |

## 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：

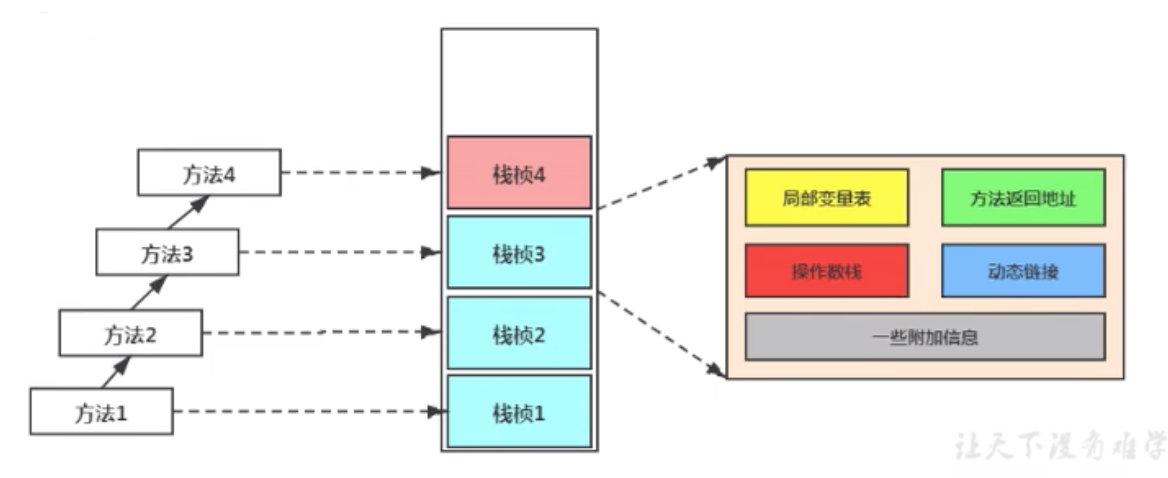
|  |
| --- |
| public class StackFrameTest {  public static void main(String[] args) {  StackFrameTest test = new StackFrameTest();  test.method1();  }   public void method1() {  System.*out*.println("method1()开始执行...");  method2();  System.*out*.println("method1()执行结束...");  }   public int method2() {  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... |

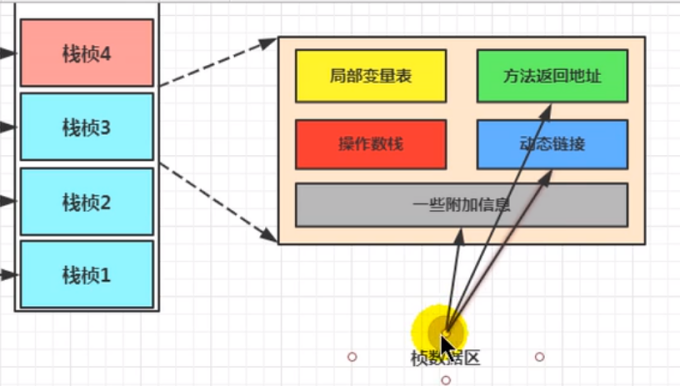
# 3.栈帧

## 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**

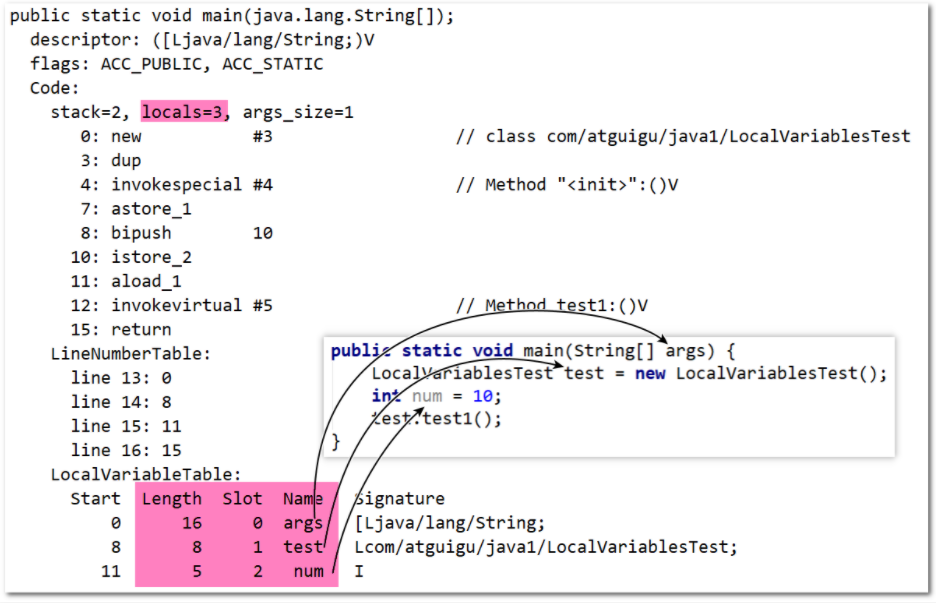
局部变量表中的变量只在当前方法调用中有效。

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。

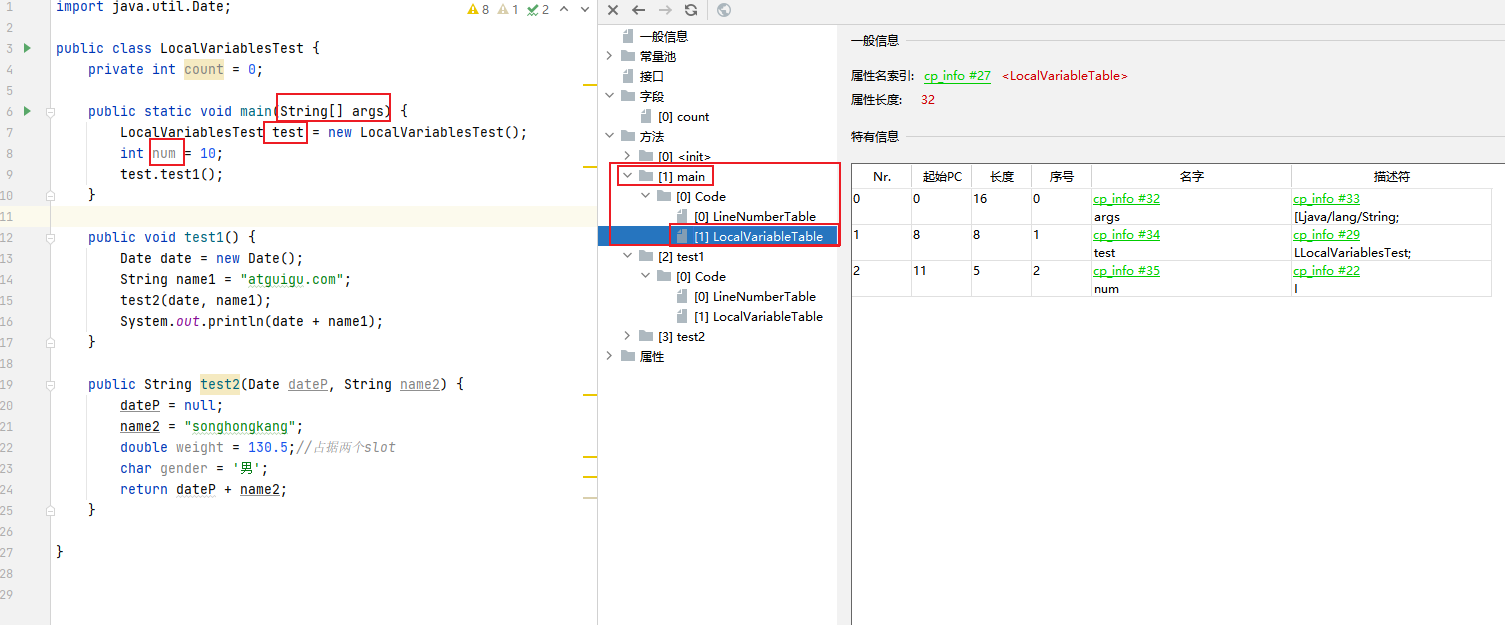
#### 实验：查看局部变量表

|  |
| --- |
| 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;  }  } |

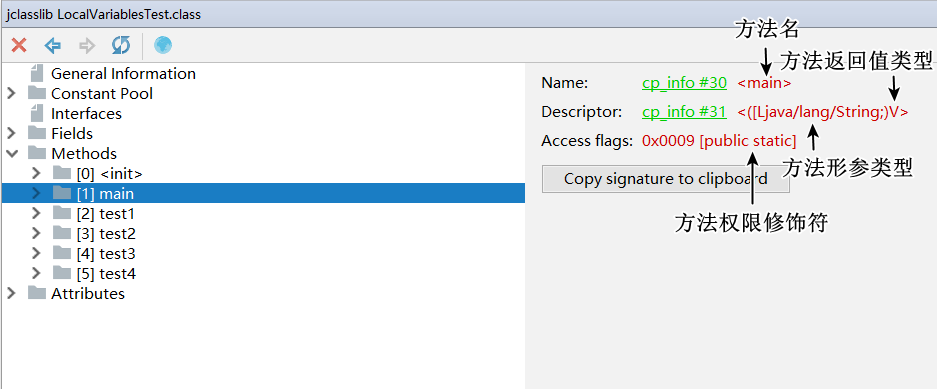
Javap查看字节码文件：



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

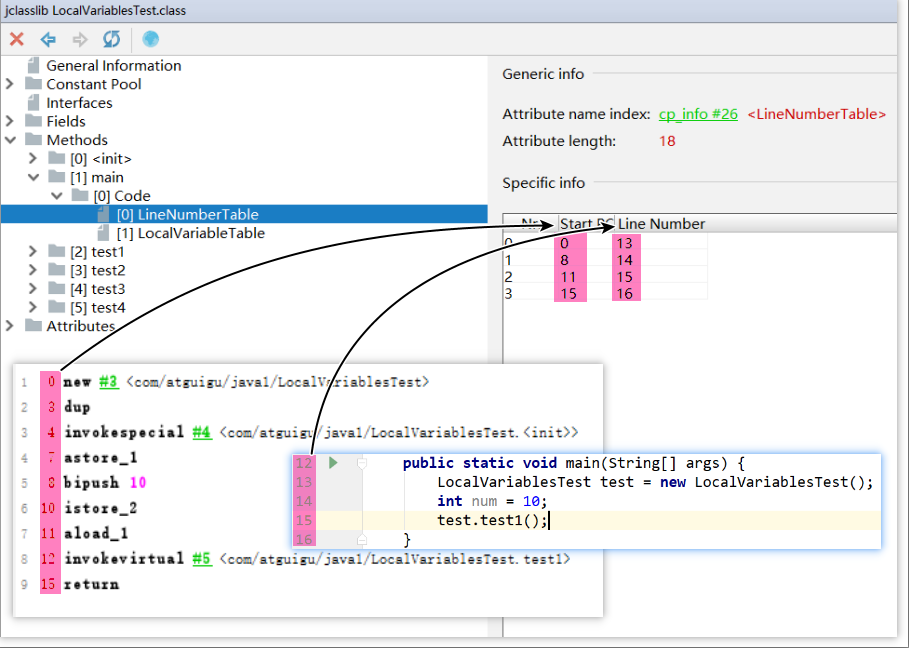
（2）查看方法的字节码长度为 16（0~15）



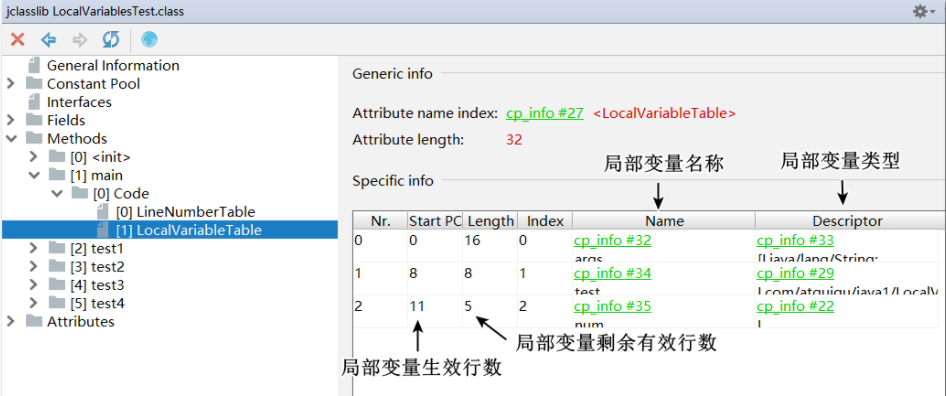
* 方法异常信息表
* 杂项（Misc）



（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

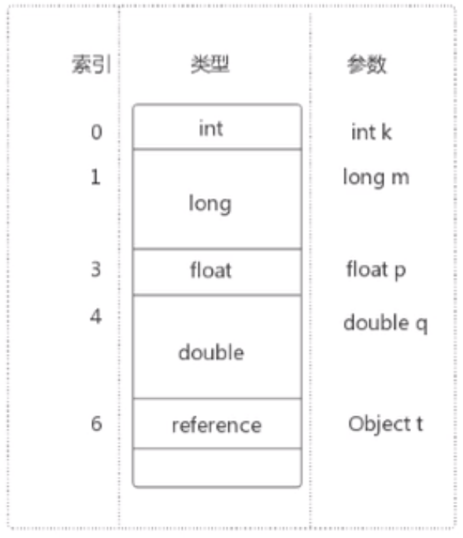
Length： 该局部变量在字节码指令中存活的长度

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

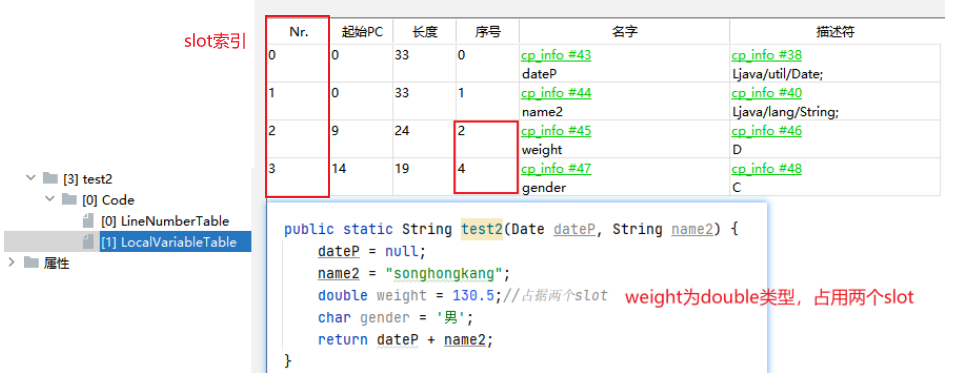
1. 局部变量表，最基本的存储单元是Slot(变量槽)
2. 参数值的存放总是在局部变量数组的index=0开始，到数组长度-1的索引结束



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

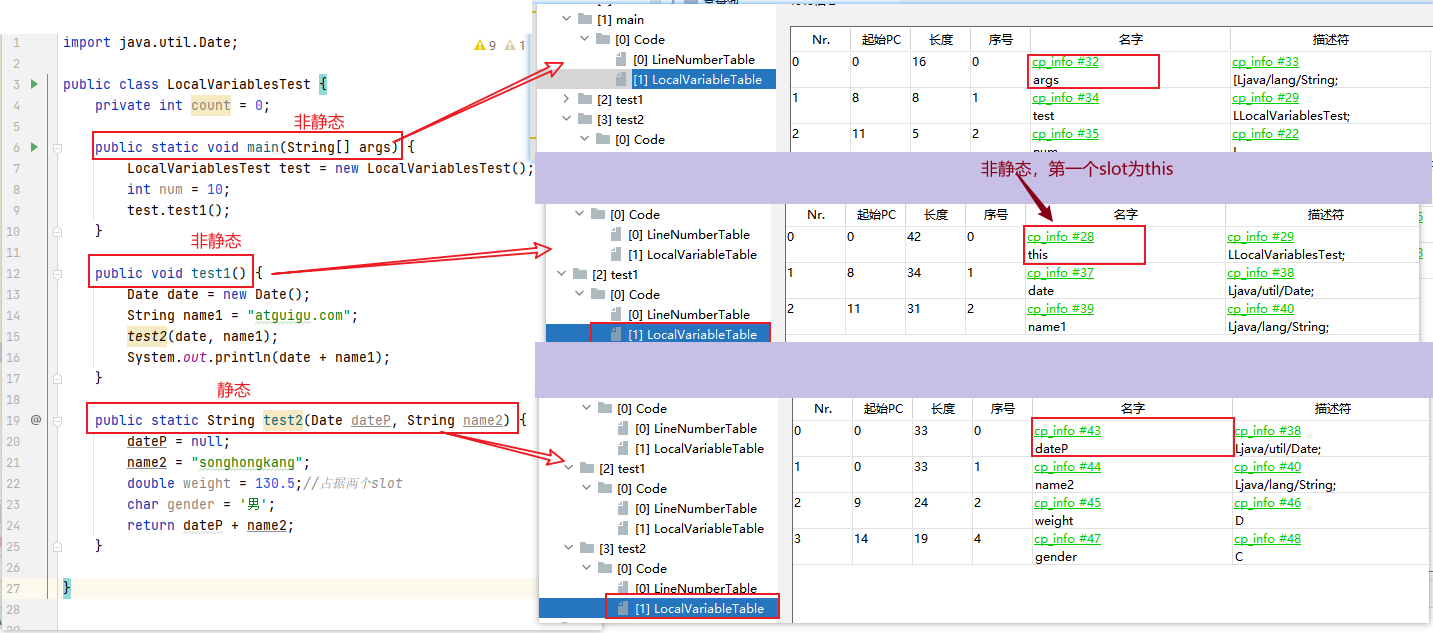


6.当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照声明顺序被复制到局部变量表中的每一个slot上

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：

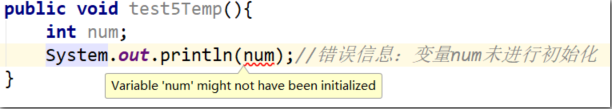
1.成员变量：在使用前，都经历过默认初始化赋值

* + - 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）

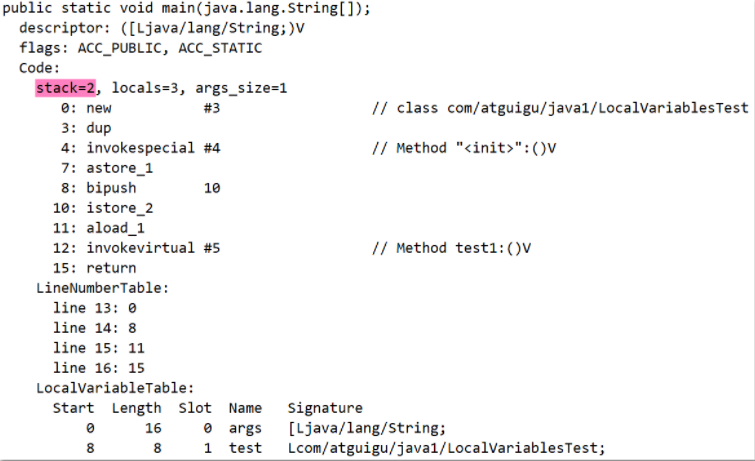
每一个独立的栈帧除了包含局部变量表以外，还包含一个后进先出（Last - In - First -Out）的 操作数栈，也可以称之为表达式栈（Expression Stack）

#### 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

1. 操作数栈，主要**用于保存计算过程的中间结果**
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





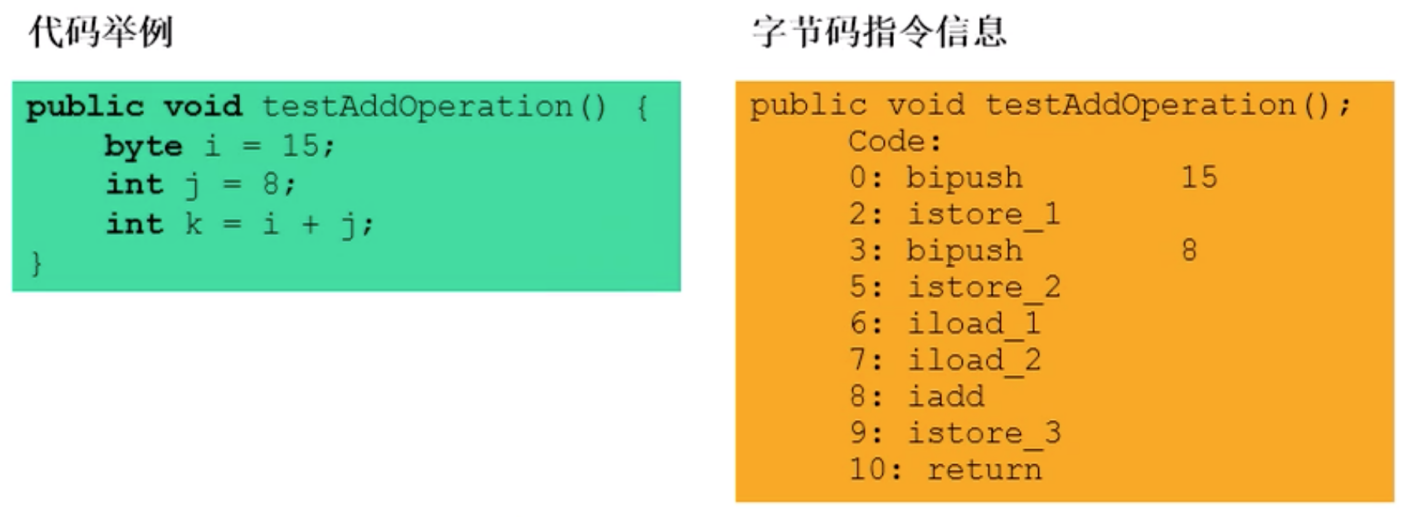
1. 栈中的任何一个元素都是可以任意的java数据类型
   1. 32bit的类型占用一个栈单位深度
   2. 64bit的类型占用两个栈深度单位
2. 操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标准的入栈push和出栈pop操作来完成一次数据访问
3. 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。
4. 操作数栈中的元素的数据类型必须与字节码指令严格匹配，这由编译器在编译期间进行验证，同时在类加载过程中的类验证阶段的数据流分析阶段要再次验证。
5. 另外，我们说Java虚拟机的**解释引擎是基于栈的执行引擎,其中的栈指的就是操作数栈。**

##### 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

|  |  |
| --- | --- |
| 代码举例：  public class testAddOperation {  public void testAddOperation() {  *//byte、short、char、boolean：都以int型来保存* byte i = 15;  int j = 8;  int k = i + j;  }   public static void main(String[] args) {  testAddOperation tao = new testAddOperation();  tao.testAddOperation();  } } | 字节码指令信息：  public void testAddOperation();  descriptor: ()V  flags: ACC\_PUBLIC  Code:  stack=2, locals=4, args\_size=1  0: bipush 15  2: istore\_1  3: bipush 8  5: istore\_2  6: iload\_1  7: iload\_2  8: iadd  9: istore\_3  10: return  LineNumberTable:  line 4: 0  line 5: 3  line 6: 6  line 7: 10  LocalVariableTable:  Start Length Slot Name Signature  0 11 0 this LtestAddOperation;  3 8 1 i B  6 5 2 j I  10 1 3 k I |

##### 操作数栈代码追踪



结合下面的图来看一下一个方法（栈帧）的执行过程：

* 首先一个方法开始执行，响应的，其对应的栈帧的局部变量表、操作数栈、pc计数器就会被初始化。然后开始按照字节码指令顺序开始进行操作；
* 字节码指令说明：

1. push：入栈
2. store：存入局部变量表
3. load: 从局部变量表取出slot中的数据放进操作数栈中

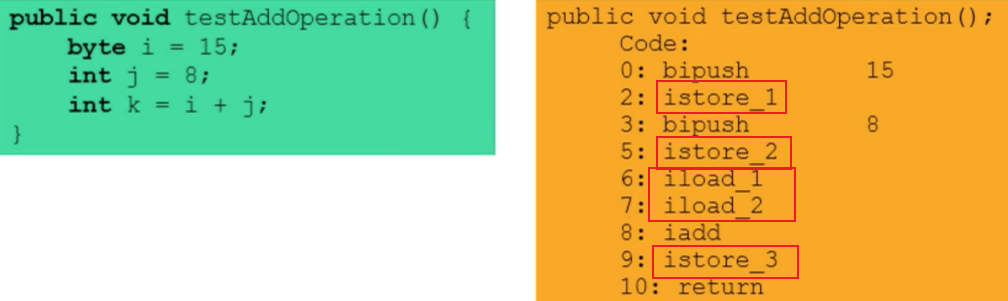
字节码指令的前缀表示操作数的数据类型：

bipush:压入操作数栈 bi表示byte类型数据

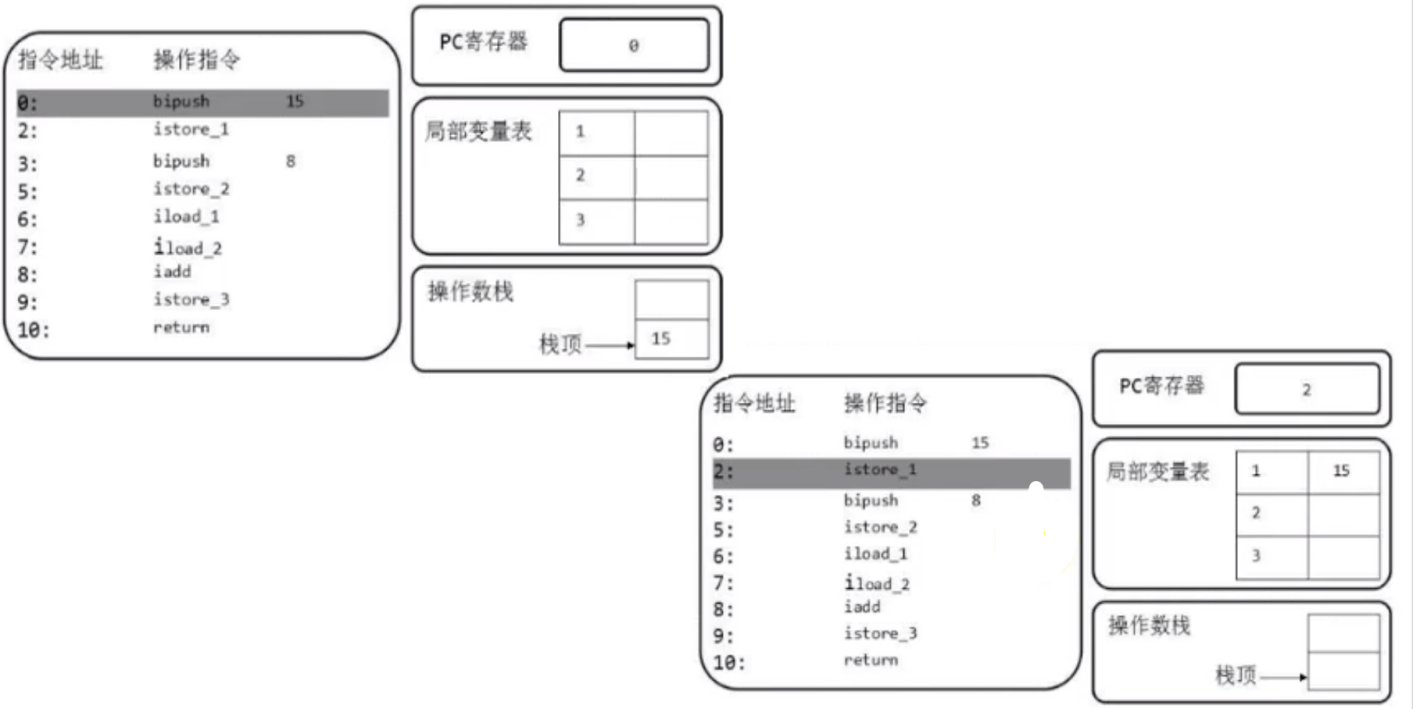
istore: 把操作数栈顶元素存进局部变量表中 i表示int类型数据

(这里验证了第7条特点：操作数栈中的元素的数据类型必须与字节码指令严格匹配)

和局部变量有关的字节码指令会在后缀中额外带有slot信息：



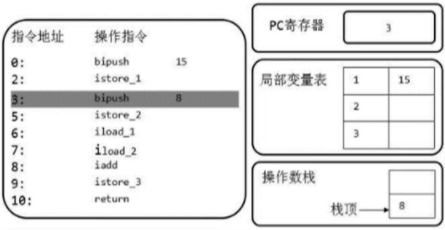
1. 首先执行第一条语句，PC寄存器指向的是0，也就是指令地址为0，然后使用bipush让操作数15入操作数栈。

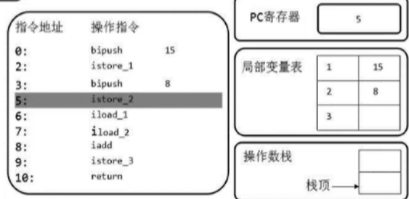


1. 执行完后，让PC + 1，指向下一行代码，下一行代码就是将操作数栈的元素存储到局部变量表index为1的位置，我们可以看到局部变量表的已经增加了一个元素

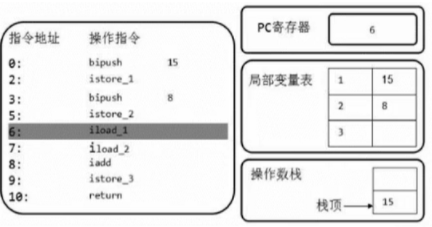
|  |
| --- |
| 解释为什么局部变量表索引从 1 开始，因为该方法为实例方法，局部变量表索引为 0 的位置存放的是 this |

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



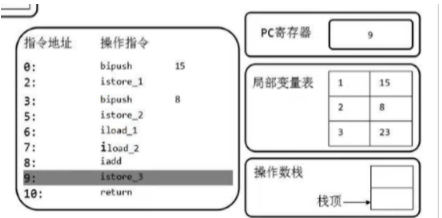
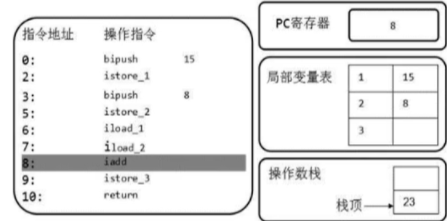


1. 然后从局部变量表中，依次将数据放在操作数栈中，等待执行 add 操作



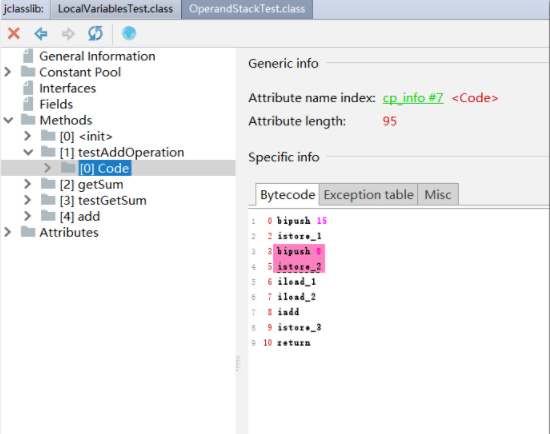


1. 然后将操作数栈中的两个元素执行相加操作，并存储在局部变量表3的位置



###### 关于 int j = 8; 的说明

我们反编译得到的字节码指令如下



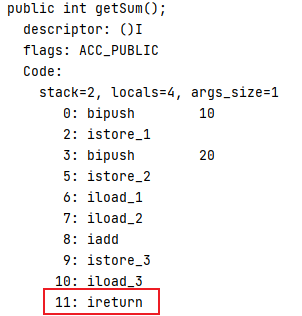
因为 8 可以存放在 byte 类型中，所以压入操作数栈的类型为 byte ，而不是 int ，所以执行的字节码指令为 bipush 8

然后执行将数值 8 存放在 int 类型的变量中：istore\_2

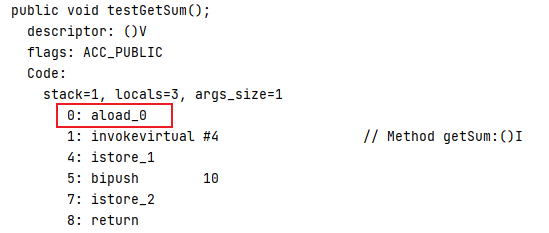
###### **关于调用方法，返回值入操作数栈的说明**

|  |
| --- |
| 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()的字节码指令：最后带着个ireturn：



* testGetSum()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++**

|  |
| --- |
| //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

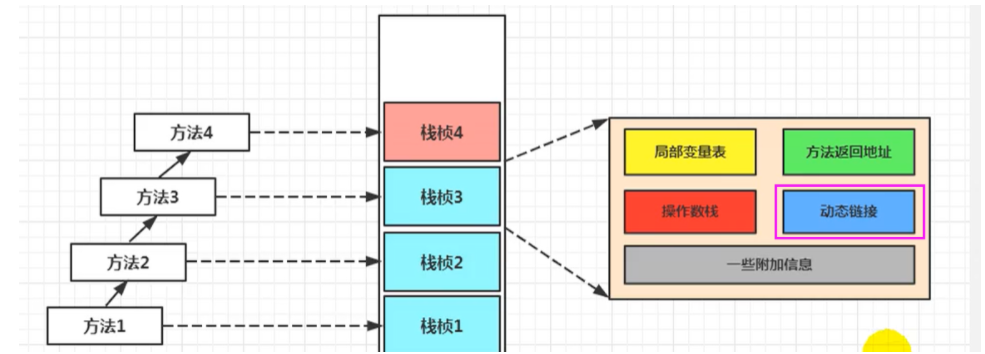
++i：先将 i 的值加 1，在将 i 的值加载到操作数栈

#### 栈顶缓存技术ToS（Top-of-Stack Cashing）

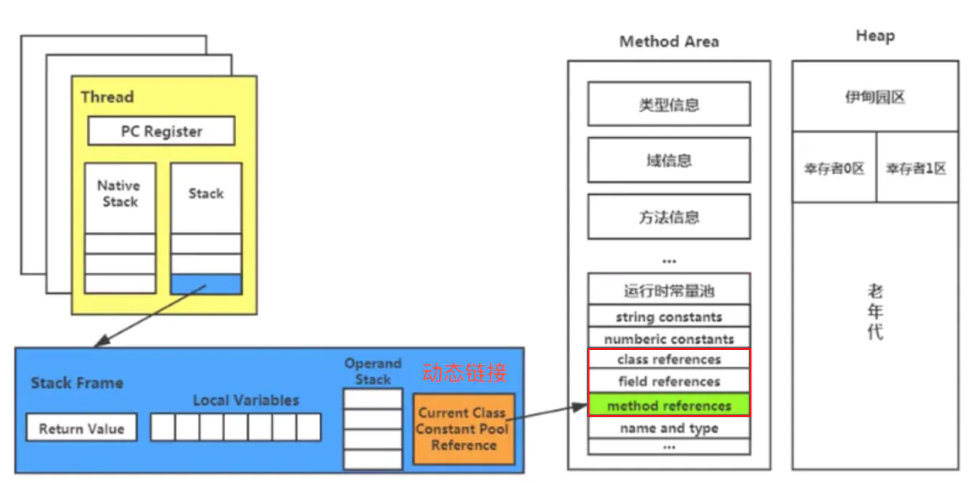
**优缺点：**基于栈式架构的虚拟机所使用的零地址指令（即不考虑地址，单纯入栈出栈）更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数

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(Dynamic Linking)



每一个栈帧内部都包含一个动态链接**[**即：指向运行时常量池中的方法引用(因此可以理解，为什么java中只有方法有多态，属性没有多态)**]**



在Java源文件被编译到字节码文件中时，所有的变量和方法引用都作为符号引用(Symbolic Reference)保存在class文件的常量池里。

从上图我们可以看到运行时常量池的引用有好几种，类引用，field引用；但是动态链接指的是method引用。

**栈帧中包含这个引用的目的就是为了支持当前的方法能够实现动态链接**(Dynamic Linking)，比如：invokedynamic指令

比如：描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用来表示的，那么**动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用**

* 代码示例

|  |
| --- |
| public class DynamicLinkingTest {  int num = 10;   public void methodA(){  System.*out*.println("methodA()....");  }   public void methodB(){  System.*out*.println("methodB()....");  methodA();  num++;  }   } |
| 在字节码指令中,methodB() 方法中通过 invokevirtual #7 指令调用了方法 A,那么 #7 是个啥呢？  找常量池：    结论：通过 #7 我们就能找到需要调用的 methodA() 方法，并进行调用 |

* 为什么需要常量池呢？

1. 因为在不同的方法，都可能调用常量或者方法，所以只需要存储一份即可，然后记录其引用即可，节省了空间
2. 常量池的作用：就是为了提供一些符号和常量，便于指令的识别

#### 动态链接的解析

##### 静态链接机制与动态链接机制

在JVM中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关，链接指的是由符号引用转变为直接引用这个过程。

1. 静态链接：

当一个字节码文件被装载进JVM内部时，如果被调用的目标方法在编译期确定，且运行期保持不变时，这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接

1. 动态链接：

如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运行期将调用的方法的符号转换为直接引用，由于这种引用转换过程具备动态性，因此也被称之为动态链接。

##### **早期绑定与晚期绑定**

静态链接和动态链接对应的方法的绑定机制为：早期绑定(Early Binding)和晚期绑定(Late Binding)

绑定是指对常量池中class reference\field reference\method reference 被替换为直接引用的过程，这仅仅发生一次。

1. 早期绑定

早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个『方法』与『该方法所属的类型』进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。

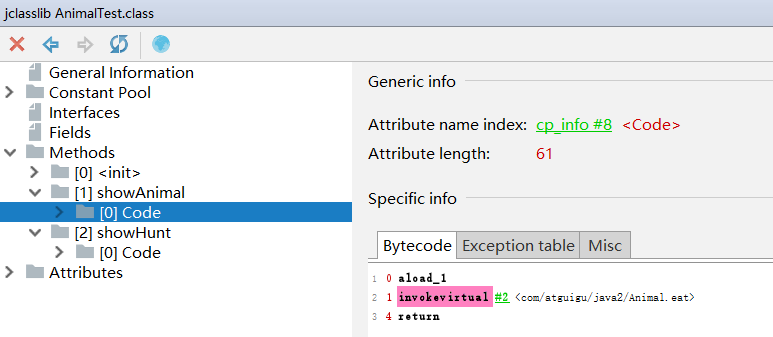
1. 晚期绑定

如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。

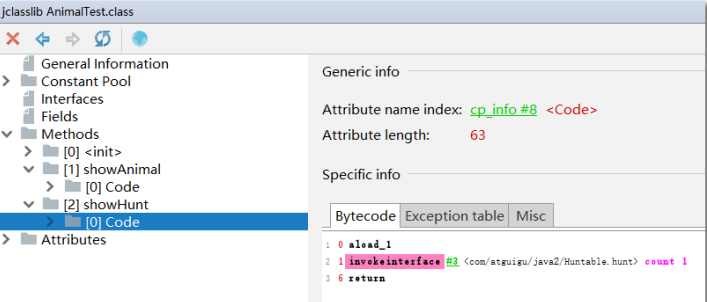
1. 代码示例

|  |
| --- |
| */\*\*  \* 说明早期绑定和晚期绑定的例子  \*  \* @author shkstart  \* @create 2020 上午 11:59  \*/* class Animal {  public void eat() {  System.*out*.println("动物进食");  } }  interface Huntable {  void hunt(); }  class Dog extends Animal implements Huntable {  @Override  public void eat() {  System.*out*.println("狗吃骨头");  }   @Override  public void hunt() {  System.*out*.println("捕食耗子，多管闲事");  } }  class Cat extends Animal implements Huntable {  public Cat() {  super();*//表现为：早期绑定* }   public Cat(String name) {  this();*//表现为：早期绑定* }   @Override  public void eat() {  super.eat();*//表现为：早期绑定* System.*out*.println("猫吃鱼");  }   @Override  public void hunt() {  System.*out*.println("捕食耗子，天经地义");  } }  public class AnimalTest {  public void showAnimal(Animal animal) {  animal.eat();*//表现为：晚期绑定* }   public void showHunt(Huntable h) {  h.hunt();*//表现为：晚期绑定* } } |

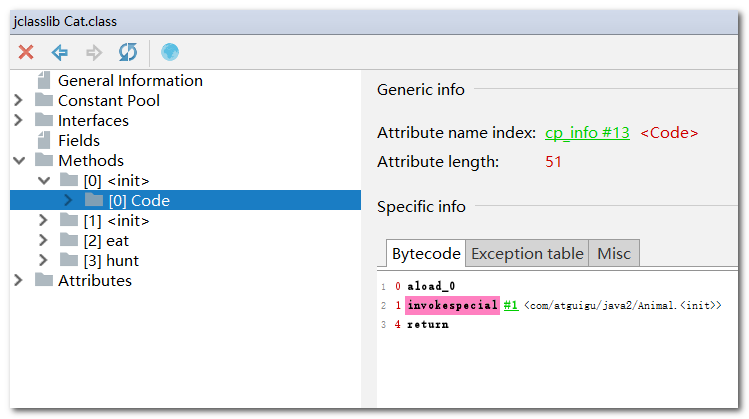
* invokevirtual 体现为晚期绑定



* invokeinterface 也体现为晚期绑定



* invokespecial 体现为早期绑定



##### **多态性与方法绑定机制**

随着高级语言的横空出世，类似于Java一样的基于面向对象的编程语言如今越来越多，尽管这类编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装、继承和多态等面向对象特性，**既然这一类的编程语言具备多态特性，那么自然也就具备早期绑定和晚期绑定两种绑定方式。**

Java中任何一个普通的方法其实都具备虚函数的特征，它们相当于C++语言中的虚函数（C++中则需要使用关键字virtual来显式定义）。**如果在Java程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字final来标记这个方法（也就是说当这个方法不能被继承的子类重写的时候，就不具备虚函数的特征了）**

##### **虚方法和非虚方法**

1. ****虚方法与非虚方法的区别****

如果方法在编译期就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非虚方法。

* 静态方法、私有方法、final方法、构造器(this(...))、父类方法(super(...))都是非虚方法。
* 其他方法称为虚方法

可以这么理解：不能被继承重写的方法都是非虚方法，因为非虚方法即无法体现多态特性。其他所有能够体现多态特性的方法称为虚方法

1. ****子类对象的多态的使用前提****

* 类的继承关系
* 方法的重写

实际开发编写代码中用的接口，实际执行是导入的三方jar包已经实现的功能

1. ****虚拟机中提供了以下几条方法调用指令****

* 4个普通调用指令：  
   invokestatic：调用静态方法，解析阶段确定唯一方法版本；  
   invokespecial:调用<init>方法、私有、父类方法；解析阶段确定唯一方法版本；  
   invokevirtual:调用所有虚方法；  
   invokeinterface：调用接口方法；
* 1个动态调用指令（Java7新增）：  
   invokedynamic：动态解析出需要调用的方法，然后执行 .
* 区别

1. 前四条指令固化在虚拟机内部，方法的调用执行不可人为干预,而invokedynamic指令则支持由用户确定方法版本
2. 其中invokestatic指令和invokespecial指令调用的方法称为非虚方法，其余的（final修饰的除外）称为虚方法。
3. final方法被JVM归属于invokevirtual指令，但是final修饰的也是非虚方法，这里要注意。

|  |
| --- |
| */\*\*  \* 解析调用中非虚方法、虚方法的测试  \*  \* invokestatic指令和invokespecial指令调用的方法称为非虚方法  \* @author shkstart  \* @create 2020 下午 12:07  \*/* class Father {  public Father() {  System.*out*.println("father的构造器");  }   public static void showStatic(String str) {  System.*out*.println("father " + str);  }   public final void showFinal() {  System.*out*.println("father show final");  }   public void showCommon() {  System.*out*.println("father 普通方法");  } }  public class Son extends Father {  public Son() {  *//invokespecial* super();  }   public Son(int age) {  *//invokespecial* this();  }   *//不是重写的父类的静态方法，因为静态方法不能被重写！* public static void showStatic(String str) {  System.*out*.println("son " + str);  }   private void showPrivate(String str) {  System.*out*.println("son private" + str);  }   public void show() {  *//invokestatic  showStatic*("atguigu.com");   *//invokestatic* super.*showStatic*("good!");   *//invokespecial* showPrivate("hello!");   *//invokevirtual  //虽然字节码指令中显示为invokevirtual，但因为此方法声明有final，不能被子类重写，所以也认为此方法是非虚方法。* showFinal();   *//invokespecial* super.showCommon();   *//invokevirtual  //有可能子类会重写父类的showCommon()方法* showCommon();  info();   MethodInterface in = null;  *//invokeinterface* in.methodA();  }   public void info() {   }   public void display(Father f) {  f.showCommon();  }   public static void main(String[] args) {  Son so = new Son();  so.show();  } }  interface MethodInterface {  void methodA(); } |
|  |

* 关于invokedynamic指令

JVM字节码指令集一直比较稳定，一直到java7才增加了一个invokedynamic指令，这是Java为了实现【动态类型语言】支持而做的一种改进

但是java7中并没有提供直接生成invokedynamic指令的方法，需要借助ASM这种底层字节码工具来产生invokedynamic指令.直到Java8的Lambda表达式的出现，invokedynamic指令的生成，在java中才有了直接生成方式

Java7中增加的动态语言类型支持的本质是对java虚拟机规范的修改，而不是对java语言规则的修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在java平台的动态语言的编译器

##### **动态类型语言和静态类型语言**

动态类型语言和静态类型语言两者的区别就在于对类型的检查是在编译期还是在运行期，满足前者就是静态类型语言，反之则是动态类型语言。

直白来说 静态语言是判断变量自身的类型信息；动态类型语言是判断变量值的类型信息，变量没有类型信息，变量值才有类型信息,这是动态语言的一个重要特征

Java是静态类型语言（尽管lambda表达式为其增加了动态特性），js，python是动态类型语言.

Java：String info = "硅谷";//静态语言

JS：var name = "硅谷“；var name = 10;//动态语言

Pythom: info = 130;//更加彻底的动态语言

#### 动态链接的分派

##### **方法重写的本质(分派)**

1. 找到操作数栈的第一个元素所执行的对象的实际类型，记作C。
2. 如果在类型C中找到与常量池中的描述符、简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回java.lang.IllegalAccessError异常。
3. 否则，按照继承关系从下往上依次对c的各个父类进行第二步的搜索和验证过程。
4. 如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError异常。

**IllegalAccessError介绍**

程序视图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会引起编译器异常。

这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。比如，你把应该有的jar包放从工程中拿走了，或者Maven中存在jar包冲突

**回看解析阶段**

1. 解析阶段就是将常量池内的符号引用转换为直接引用的过程(解析阶段=分派过程)
2. 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的CONSTANT Class info、CONSTANT Fieldref info、CONSTANT Methodref info等

##### **虚方法表**

1. **虚方法表的作用：**

在面向对象编程中，会很频繁的使用到**动态分派**，如果在每次动态分派的过程中都要重新在类的方法元数据中搜索合适的目标的话就可能影响到执行效率。 因此，为了提高性能，jvm采用在类的方法区建立一个**虚方法表(virtual method table)**（非虚方法不会出现在表中）来实现。使用索引表来代替查找。

|  |
| --- |
| 翻译翻译：  假设A方法调用了一堆方法，但是由于不知道这些方法是不是虚方法，因此就要按照**[『分派』](#_方法重写的本质)**过程去寻找方法真正所属类，由于搜索效率低下，从而引入了在当A方法所在的类的方法区中建立虚方发表，通过查表判别是否是虚方法，也就省去了在常量池中搜寻此方法属于的类的过程。 |

每个类的方法区中都有一个虚方法表，表中存放着各个方法的实际入口。

1. **虚方法表什么时候被创建？**

虚方法表会在类加载的链接阶段被创建(**链接的解析阶段**)，并开始初始化，类的变量初始值准备完成之后，jvm会把该类的虚方法表也初始化完毕。

1. **虚方法表实例**

我们定义三个类、一个Friendly接口



* Dog类的虚方法表

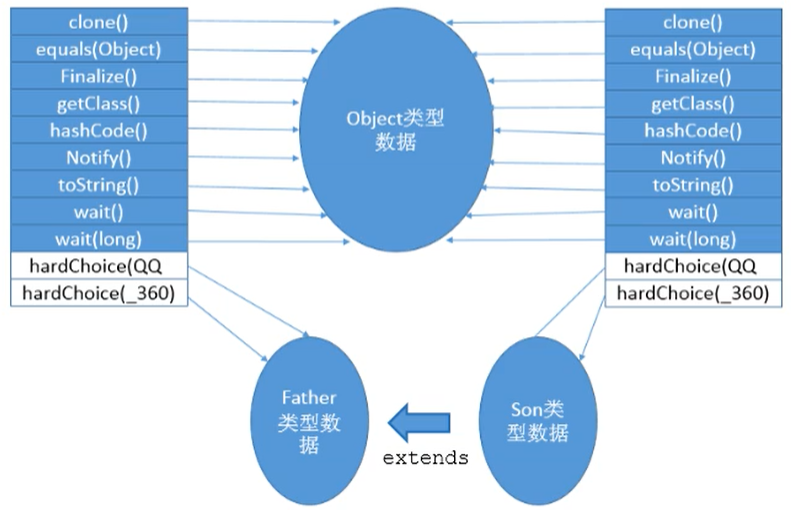
|  |
| --- |
| class Dog{  　　public void sayHello(){  　　}  　　public String toString(){  　　　　return "Dog";  　　}  }    **结论1：当前类的虚方法表中包含 当前类的虚方法 以及继承过来的虚方法** |

* 可卡犬虚方法表

|  |
| --- |
| class CockerSpaniel extends Dog implements Friendly{  　　public void sayHello(){  　　　　super.sayHello();  　　}  　　public void sayGoodbye(){  　　}  }    可卡犬若是使用toString方法无需向上找Object类，只需找到Dog类即可；这是一个效率的提升  **结论2：继承过来的虚方法指向最近父类** |

* 猫类的虚方法表：

|  |
| --- |
| interface Friendly{  　　void sayHello();  　　void sayGoodbye();  }  class Cat implements Friendly{  　　public void eat(){  　　}  　　public void sayHello(){  　　}  　　public void sayGoodbye(){  　　}  　　protected void finalize(){  　　}  　　public String toString(){  　　}  }  IMG_286  **结论3：当前类重写了父类的虚方法，那么这个虚方法指向自己** |



结论：

1. 虚方法表实在类的链接中的解析阶段就会创建并初始化
2. 这个表是该类所有虚方法的集合，放在数组中，并且标明此方法属于自己还是父类，省去了动态分配还得去查方法区常量池中，此方法所属的实际类型。
3. 如果该类重写了父类的虚方法，那么此虚方法指向自己

### 4.方法返回地址(Return Address)

#### 方法返回地址(Return Address)

1. 存放 **『调用当前方法的方法』**的pc寄存器的值(调用该方法的指令的下一条指令的地址)
2. 一个方法的结束，有两种方式：
   1. 正常执行完成
   2. 出现未处理的异常，非正常退出

无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。

方法正常退出时，将调用者的pc计数器的值作为返回地址;而通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值。

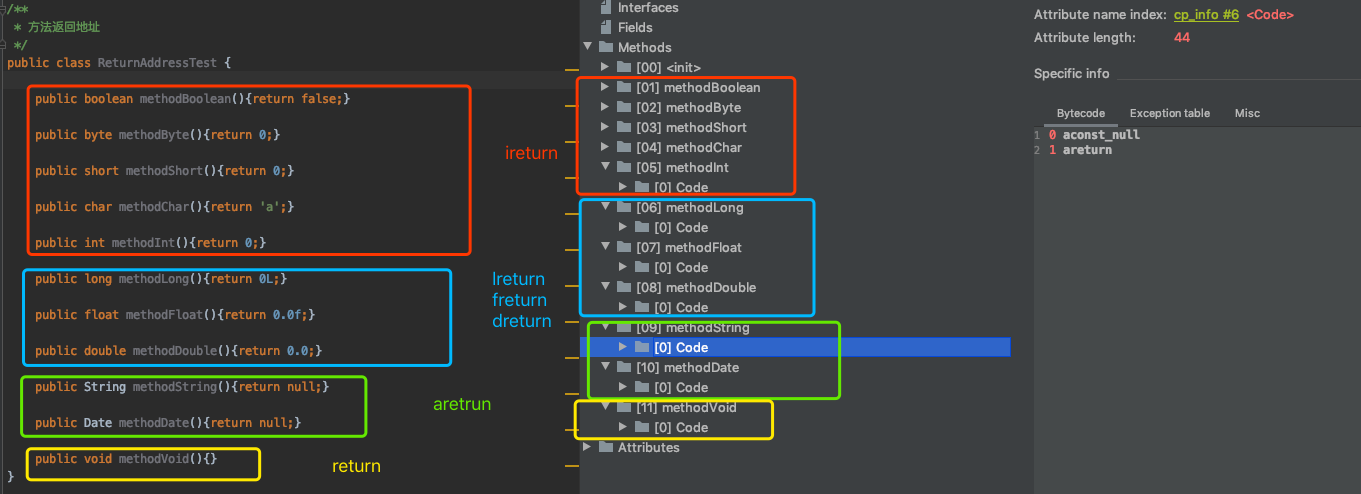
本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值压入调用者栈帧的操作数栈、设置PC寄存器值等，让调用者方法继续执行下去。

* 当一个方法开始执行后，只有两种方式可以退出这个方法：
* **正常退出：**

1. 执行引擎遇到任意一个方法返回的字节码指令(return)，会有返回值传递给上层的方法调用者，简称正常完成出口；
2. 一个方法在正常调用完成之后，究竟需要使用哪一个返回指令，还需要根据方法返回值的实际数据类型而定。

在字节码指令中，返回指令包含：

* 1. ireturn：当返回值是boolean，byte，char，short和int类型时使用
  2. lreturn：Long类型
  3. freturn：Float类型
  4. dreturn：Double类型
  5. areturn：引用类型
  6. return： 返回值类型为void的方法、实例初始化方法、类和接口的初始化方法

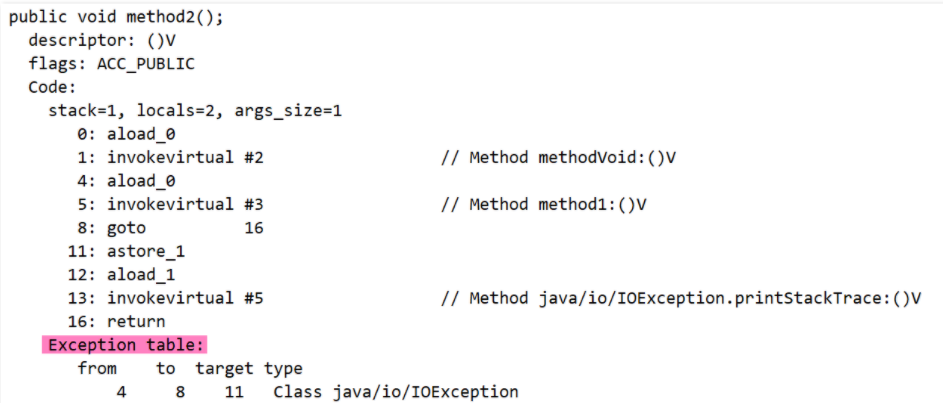


* **异常退出**

1. 在方法执行的过程中遇到了异常（Exception），并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜素到匹配的异常处理器，就会导致方法退出，简称异常完成出口
2. 方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码。

* **代码示例**

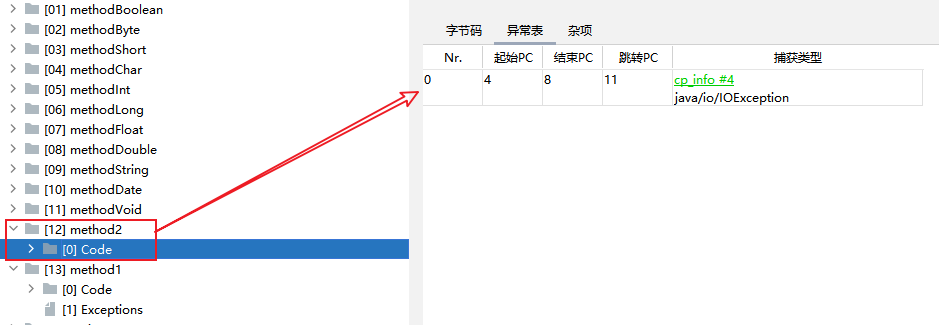
|  |
| --- |
| public class ReturnAddressTest {  //ireturn  public boolean methodBoolean() {  return false;  }  //ireturn  public byte methodByte() {  return 0;  }  //ireturn  public short methodShort() {  return 0;  }  //ireturn  public char methodChar() {  return 'a';  }  //ireturn  public int methodInt() {  return 0;  }  //lreturn  public long methodLong() {  return 0L;  }  //freturn  public float methodFloat() {  return 0.0f;  }  //dreturn  public double methodDouble() {  return 0.0;  }  //areturn  public String methodString() {  return null;  }  //areturn  public Date methodDate() {  return null;  }  //return  public void methodVoid() {   }   static {  int i = 10;  }   public void method2() {  methodVoid();  try {  method1();  } catch (IOException e) {  e.printStackTrace();  }  }   public void method1() throws IOException {  FileReader fis = new FileReader("atguigu.txt");  char[] cBuffer = new char[1024];  int len;  while ((len = fis.read(cBuffer)) != -1) {  String str = new String(cBuffer, 0, len);  System.*out*.println(str);  }  fis.close();  }   } |

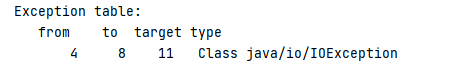


**异常处理表：**

1. 反编译字节码文件，可得到 Exception table
2. from ：字节码指令起始地址
3. to ：字节码指令结束地址
4. target ：出现异常跳转至地址为 11 的指令执行
5. type ：捕获异常的类型

* Method2()中对异常进行了try-catch处理，在Code中是有异常处理表的





* Method1()对异常做的是抛出处理，在code中是没有异常处理表的





# 4.虚拟机栈的相关面试题

## 举例栈溢出的情况？（StackOverflowError）

递归调用，通过-Xss设置栈的大小；

## 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）

## 分配的栈内存越大越好么？

不是 会挤占其他线程的空间

## 垃圾回收是否会涉及到虚拟机栈？

不会



## 方法中定义的局部变量是否线程安全？

何为线程安全？

* 如果只有一个线程才可以操作此数据，则必是线程安全的。
* 如果有多个线程操作此数据，则此数据是共享数据。如果不考虑同步机制的话，会存在线程安全问题。

具体问题具体分析：

**对于方法中的局部变量来说，本身是属于栈帧中的数据，是线程私有的，那么如果对象是在『内部产生，并在内部消亡』，没有返回到外部，那么它就是线程安全的，反之则是线程不安全的。**

|  |
| --- |
| */\*\*  \* 面试题：  \* 方法中定义的局部变量是否线程安全？具体情况具体分析  \*  \* 何为线程安全？  \* 如果只有一个线程才可以操作此数据，则必是线程安全的。  \* 如果有多个线程操作此数据，则此数据是共享数据。如果不考虑同步机制的话，会存在线程安全问题。  \* @author shkstart  \* @create 2020 下午 7:48  \*/* public class StringBuilderTest {  *//s1的声明方式是线程安全的* public static void method1(){  *//StringBuilder:线程不安全* StringBuilder s1 = new StringBuilder();  s1.append("a");  s1.append("b");  *//...* }  *//sBuilder通过参数传递方法内，存在线程不安全的问题* public static void method2(StringBuilder sBuilder){  sBuilder.append("a");  sBuilder.append("b");  *//...* }  *//操作s1之后，将s1作为返回值返回，存在线程不安全的问题* public static StringBuilder method3(){  StringBuilder s1 = new StringBuilder();  s1.append("a");  s1.append("b");  return s1;  }  *//s1的操作：是线程安全的* public static String method4(){  StringBuilder s1 = new StringBuilder();  s1.append("a");  s1.append("b");  return s1.toString();  }   public static void main(String[] args) {  StringBuilder s = new StringBuilder();  new Thread(() -> {  s.append("a");  s.append("b");  }).start();  *method2*(s);  } } |