

JVM和字节码基础

刘 钦

知识点

- JVM与字节码 (P9、P11)
- Class文件结构 (大致读懂P14, P15-P16, P17-P41)
- 运行时数据区 (P52)
- 虚拟机指令 (P59-P69)
- 类加载 (P73)
- 虚拟机执行 (P81)
- Java语句与字节码 (P92-106)

Reference

- http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html
- <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>
- <http://cs.au.dk/~mis/dOvs/jvmspec/ref-Java.html>
- <http://blog.csdn.net/luanlouis/article/details/39892027>
- <http://blog.csdn.net/zq602316498/article/details/38847935>
- <https://zh.wikipedia.org/wiki/Java%E5%AD%97%E8%8A%82%E7%A0%81>

Outline

- JVM与字节码
- class文件结构
- 运行时数据区
- 字节码指令集
- 字节码的执行
- Java语句与字节码

1 JVM与字节码

Java代码

- outer:
- for (int i = 2; i < 1000; i++) {
- for (int j = 2; j < i; j++) {
- if (i % j == 0)
- continue outer;
- }
- System.out.println (i);
- }

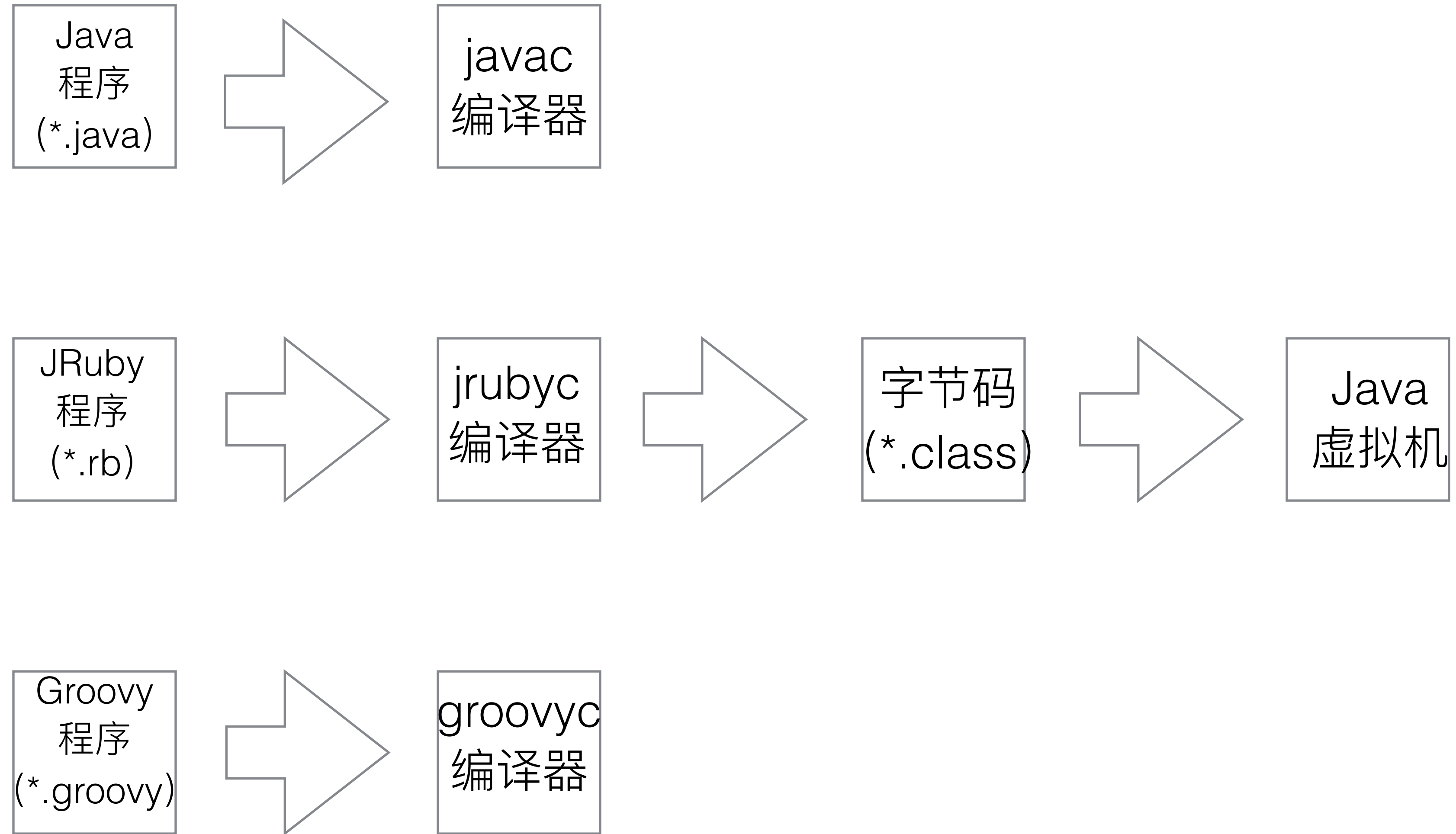
Java代码编译成字节码

字节码

- 0: iconst_2
- 1: istore_1
- 2: iload_1
- 3: sipush 1000
- 6: if_icmpge 44
- 9: iconst_2
- 10: istore_2
- 11: iload_2
- 12: iload_1
- 13: if_icmpge 31
- 16: iload_1
- 17: iload_2
- 18: irem
- 19: ifne 25
- 22: goto 38
- 25: iinc 2, 1
- 28: goto 11
- 31: getstatic #84; // Field java/lang/System.out:Ljava/io/PrintStream;
- 34: iload_1
- 35: invokevirtual #85; // Method java/io/PrintStream.println:(I)V
- 38: iinc 1, 1
- 41: goto 2
- 44: return

字节码 在虚拟机中 执行

语言无关性



Java虚拟机提供的语言无关性

2 class文件结构

实例文件

1. package org.fenixsoft.clazz;

3. public class TestClass{

4.

5. private int m;

7. public int inc(){

8. return m+1;

9. }

10.}

```
promote:~ qinliu$ javac TestClass.java
promote:~ qinliu$ javap -verbose TestClass
警告： 二进制文件TestClass包含org.fenixsoft.clazz.TestClass
Classfile /Users/qinliu/TestClass.class
  Last modified 2015-4-15; size 295 bytes
  MD5 checksum 81f2ab948a7a3068839b61a8f91f634b
  Compiled from "TestClass.java"
public class org.fenixsoft.clazz.TestClass
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #4.#15          // java/lang/
Object."<init>":()V
  #2 = Fieldref           #3.#16          // org/fenixsoft/clazz/
TestClass.m:I
  #3 = Class               #17            // org/fenixsoft/clazz/
TestClass
  #4 = Class               #18            // java/lang/Object
  #5 = Utf8                m
  #6 = Utf8                I
  #7 = Utf8                <init>
  #8 = Utf8                ()V
  #9 = Utf8                Code
#10 = Utf8                LineNumberTable
#11 = Utf8                inc
#12 = Utf8                ()I
#13 = Utf8                SourceFile
#14 = Utf8                TestClass.java
#15 = NameAndType          #7:#8          // "<init>":()V
#16 = NameAndType          #5:#6          // m:I
#17 = Utf8                org/fenixsoft/clazz/TestClass
#18 = Utf8                java/lang/Object
```

```
{
  public org.fenixsoft.clazz.TestClass();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1          // Method java/
lang/Object."<init>":()V
      4: return
    LineNumberTable:
      line 3: 0

  public int inc();
    descriptor: ()I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
      0: aload_0
      1: getfield      #2          // Field m:I
      4: iconst_1
      5: iadd
      6: ireturn
    LineNumberTable:
      line 8: 0
}
```

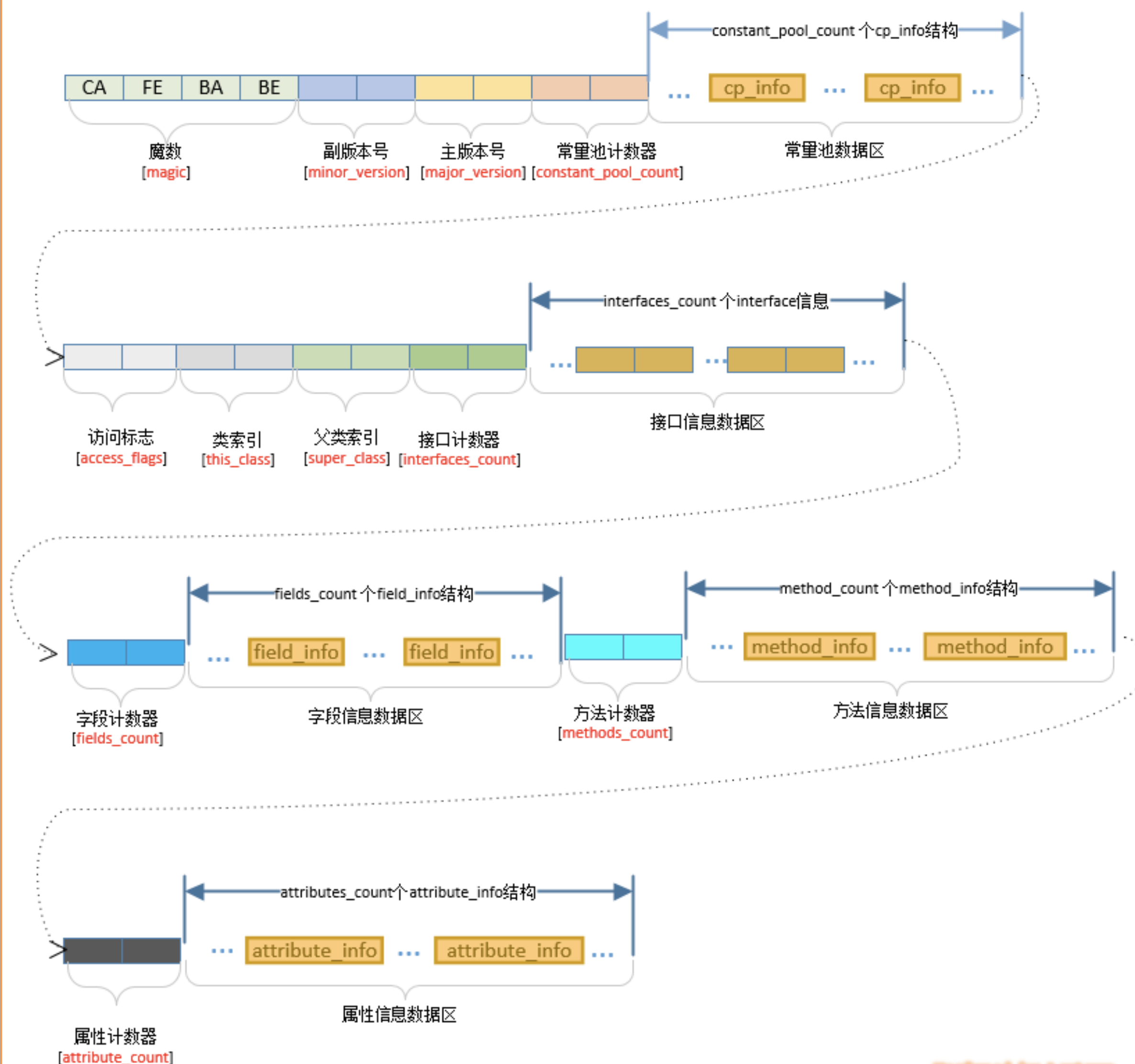
SourceFile: "TestClass.java"

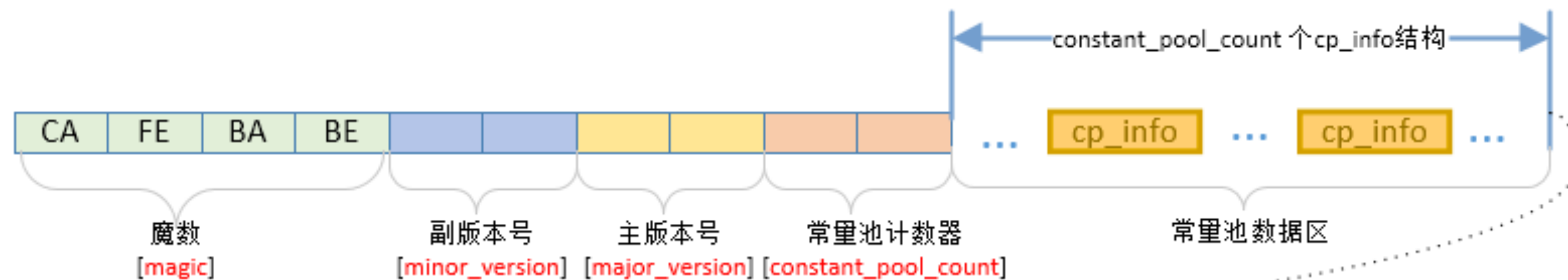
Class文件结构

- 一组以8位字节为基础单位的二进制流
- 魔数
 - 0xCAFEBAFE
- 版本号
- 常量池
 - 字面量
 - 符号引用
- 访问标志
- 类索引、父类索引与接口索引集合
- 字段表集合
- 方法表集合
- 属性表集合 （代码的实现作为一个属性值）

Class文件字节码结构组织示意图

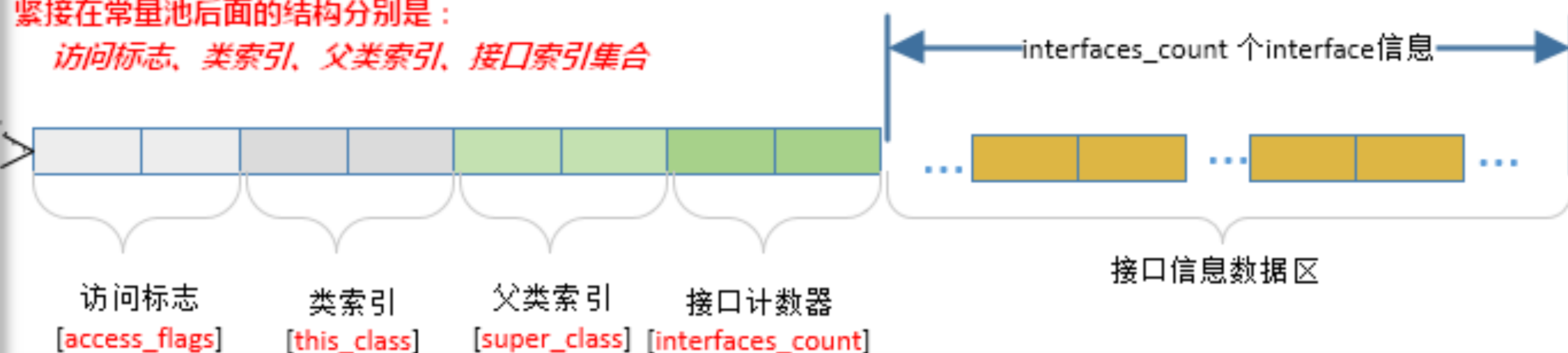
注：被编译器编译成的.class字节码文件的字节流以及其组织结构如下所示：



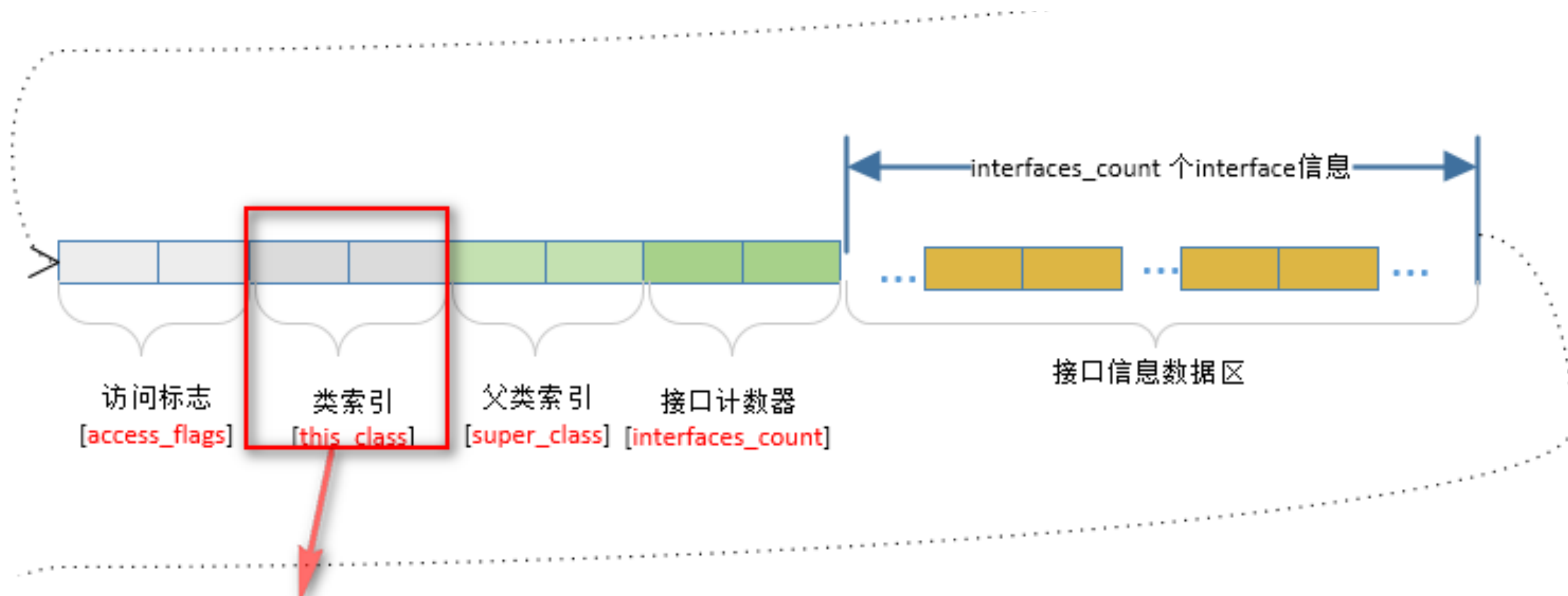


紧接在常量池后面的结构分别是：

访问标志、类索引、父类索引、接口索引集合



Designed by LouLuan



类索引有两个字节组成，两个字节中存储的值是某个常量池中的常量池项CONSTANT_Class_info的索引，而这个CONSTANT_Class_info项会进一步指向常量池项CONSTANT_Utf8_info，该项表示此类的完全限定名字字符串。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	CA	FE	BA	BE	00	00	00	32	00	10	07	00	02	01	00	14	; 漱壕...2.....
00000010h:	63	6F	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69	; com/louis/jvm/Si
00000020h:	6D	70	6C	65	07	00	04	01	00	10	6A	61	76	61	2F	6C	; mple.....java/l
00000030h:	61	6E	67	2F	4F	62	6A	65	63	74	01	00	06	3C	69	6E	; ang/Object...<in
00000040h:	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	; it>...()V...Code
00000050h:	0A	00	03	00	09	0C	00	05	00	06	01	00	0F	4C	69	6E	;Lin
00000060h:	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	12	4C	; eNumberTable...L
00000070h:	6F	63	61	6C	56	61	72	69	61	62	6C	65	54	61	62	6C	; ocalVariableTabl
00000080h:	65	01	00	04	74	68	69	73	01	00	16	4C	63	6F	6D	2F	; e...this...Lcom/
00000090h:	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69	6D	70	6C	65	; louis/jvm/Simple
000000a0h:	3B	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	; ;...SourceFile..
000000b0h:	0B	53	69	6D	70	6C	65	2E	6A	61	76	61	00	21	00	01	; .Simple.java.!..
000000c0h:	00	03	00	00	00	00	00	01	00	01	00	05	00	06	00	01	;
000000d0h:	00	07	00	00	00	2F	00	01	00	01	00	00	00	05	2A	B7	;/*?
000000e0h:	00	08	B1	00	00	00	02	00	0A	00	00	00	06	00	01	00	; ..?.....
000000f0h:	00	00	03	00	0B	00	00	00	0C	00	01	00	00	00	05	00	;
00000100h:	0C	00	0D	00	00	00	01	00	0E	00	00	00	02	00	0F		;

访问标志(access_flags)

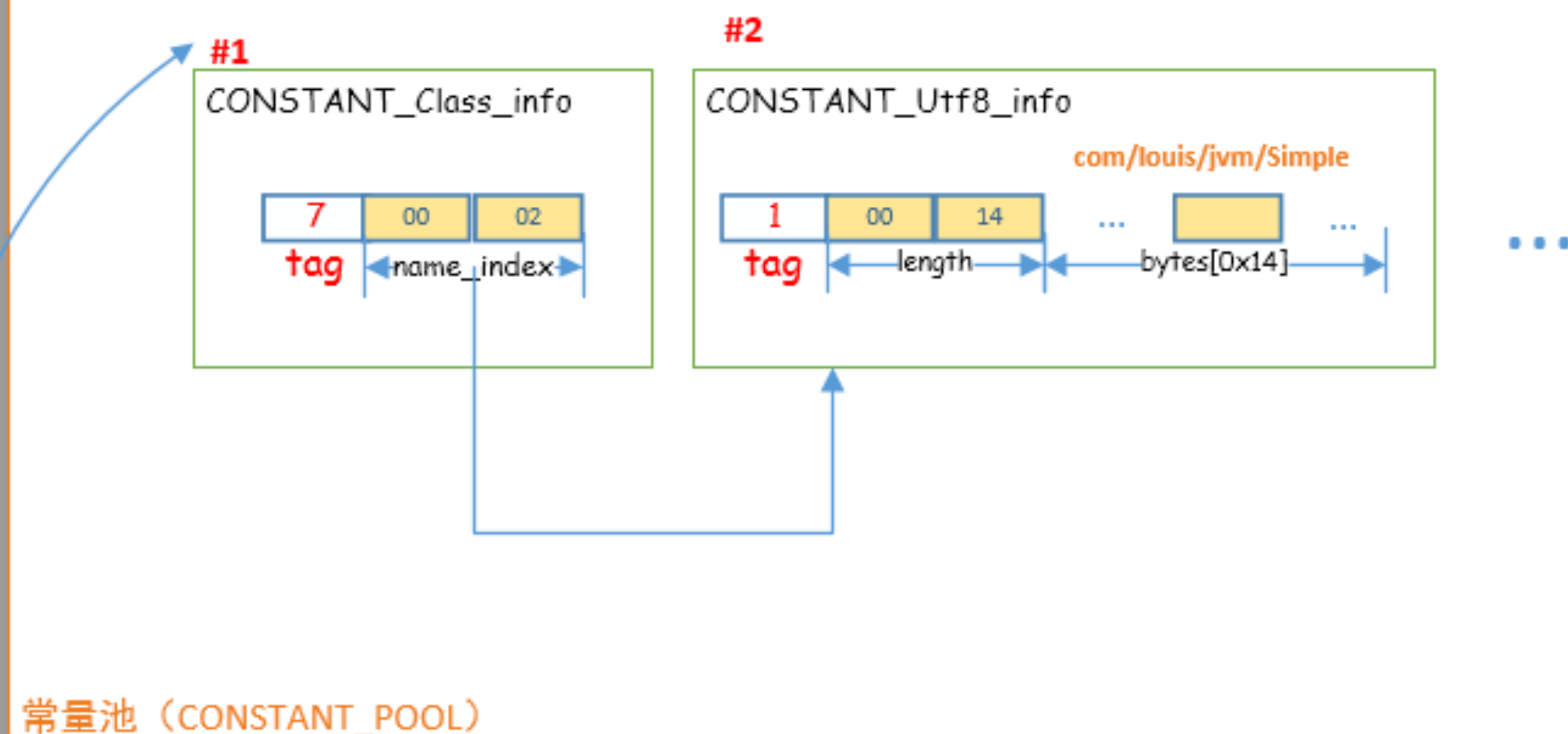
类索引(this_class), 其内的值是0x0001

Flags: ACC_PUBLIC, ACC_SUPER

Constant pool:

```
#1 = Class          #2          // com/louis/jvm/Simple
#2 = Utf8           com/louis/jvm/Simple
#3 = Class          #4          // java/lang/Object
#4 = Utf8           java/lang/Object
#5 = Utf8           <init>
#6 = Utf8           ()V
```

Designed by LouLuan
<http://blog.csdn.net/luanlouis>



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

Simple.class x

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	32	00	10	07	00	02	01	00	14
00000010h:	63	6F	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69
00000020h:	6D	70	6C	65	07	00	04	01	00	10	6A	61	76	61	2F	6C
00000030h:	61	6E	67	2F	4F	62	6A	65	63	74	01	00	06	3C	69	6E
00000040h:	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65
00000050h:	0A	00	03	00	09	0C	00	05	00	06	01	00	0F	4C	69	6E
00000060h:	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	12	4C
00000070h:	6F	63	61	6C	56	61	72	69	61	62	6C	65	54	61	62	6C
00000080h:	65	01	00	04	74	68	69	73	01	00	16	4C	63	6F	6D	2F
00000090h:	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69	6D	70	6C	65
000000a0h:	3B	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00
000000b0h:	0B	53	69	6D	70	6C	65	2E	6A	61	76	61	00	21	00	01
000000c0h:	00	03	00	00	00	00	00	01	00	01	00	05	00	06	00	01
000000d0h:	00	07	00	00	00	2F	00	01	00	01	00	00	00	05	2A	B7
000000e0h:	00	08	B1	00	00	00	02	00	0A	00	00	00	06	00	01	00
000000f0h:	00	00	03	00	0B	00	00	00	0C	00	01	00	00	00	05	00
00000100h:	0C	00	0D	00	00	00	01	00	0E	00	00	00	02	00	0F	

Constant pool:

#1 = Class

#2 = Utf8

#3 = Class

#4 = Utf8

#2

#4

// com/louis/jvm/Simple

com/louis/jvm/Simple

// java/lang/Object

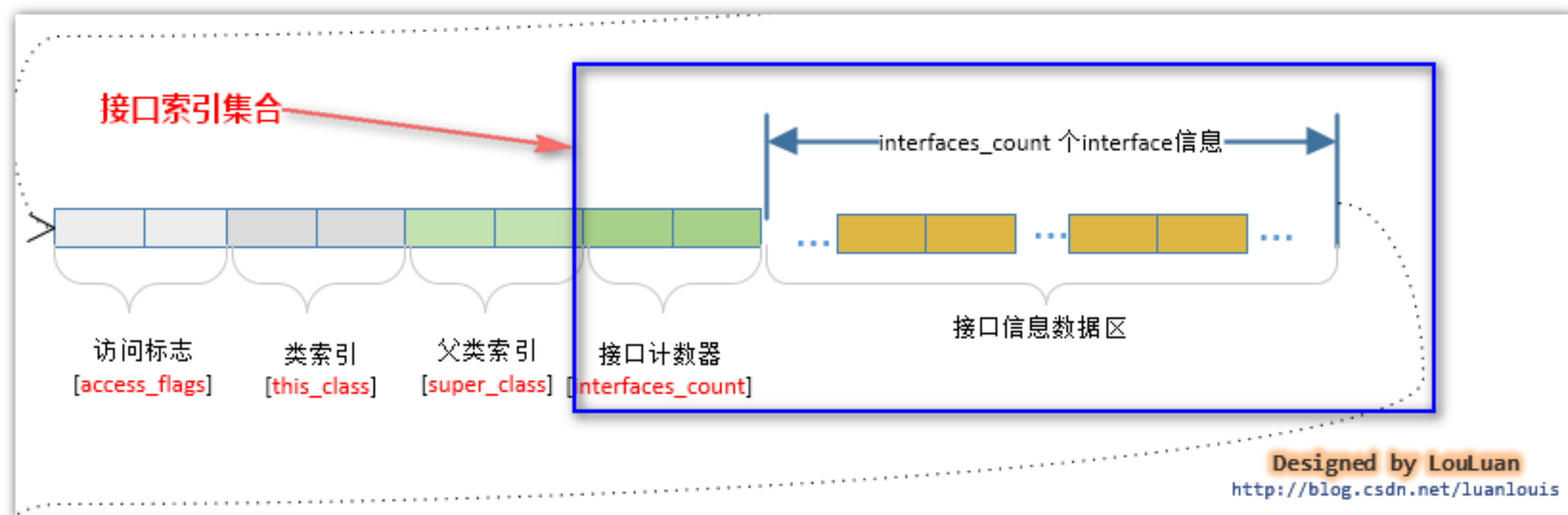
java/lang/Object

父类索引，值为0x0003

根据父类索引指向常量池中的信息可以看出：
Simple类继承自Object类

Designed by LouLuan

<http://blog.csdn.net/luanlouis>



[java] view plain copy print ? C 🔗

```
01. /**
02.  * Worker 接口类
03.  * @author luan louis
04.  */
05. public interface Worker{
06.
07.     public void work();
08.
09. }
```

[java] view plain copy print ? C 🔗

```
01. package com.louis.jvm;
02.
03. public class Programmer implements Worker {
04.
05.     @Override
06.     public void work() {
07.         System.out.println("I'm Programmer,Just coding....");
08.     }
09. }
```


Programmer.class x

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	32	00	21	07	00	02	01	00	18
00000010h:	63	6F	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	50	72
00000020h:	6F	67	72	61	6D	6D	65	72	07	00	04	01	00	10	6A	61
00000030h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	07	00
00000040h:	06	01	00	14	63	6F	6D	2F	6C	6F	75	69	73	2F	6A	76
00000050h:	6D	2F	57	6F	72	6B	65	72	01	00	06	3C	69	6E	69	74
00000060h:	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	0A	00
00000070h:	03	00	0B	0C	00	07	00	08	01	00	0F	4C	69	6E	65	4E
00000080h:	75	6D	62	65	72	54	61	62	6C	65	01	00	12	4C	6F	63
00000090h:	61	6C	56	61	72	69	61	62	6C	65	54	61	62	6C	65	01
000000a0h:	00	04	74	68	69	73	01	00	1A	4C	63	6F	6D	2F	6C	6F
000000b0h:	75	69	73	2F	6A	76	6D	2F	50	72	6F	67	72	61	6D	6D
000000c0h:	65	72	3B	01	00	04	77	6F	72	6B	09	00	12	00	14	07
000000d0h:	00	13	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	53
000000e0h:	79	73	74	65	6D	0C	00	15	00	16	01	00	03	6F	75	74
000000f0h:	01	00	15	4C	6A	61	76	61	2F	69	6F	2F	50	72	69	6E
00000100h:	74	53	74	72	65	61	6D	3B	08	00	18	01	00	1E	49	27
00000110h:	6D	20	50	72	6F	67	72	61	6D	6D	65	72	2C	4A	75	73
00000120h:	74	20	63	6F	64	69	6E	67	2E	2E	2E	2E	0A	00	1A	00
00000130h:	1C	07	00	1B	01	00	13	6A	61	76	61	2F	69	6F	2F	50
00000140h:	72	69	6E	74	53	74	72	65	61	6D	0C	00	1D	00	1E	01
00000150h:	00	07	70	72	69	6E	74	6C	6E	01	00	15	28	4C	6A	61
00000160h:	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67	3B	29
00000170h:	56	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00
00000180h:	0F	50	72	6F	67	72	61	6D	6D	65	72	2E	6A	61	76	61
00000190h:	00	21	00	01	00	03	00	01	00	05	00	00	00	02	00	01
000001a0h:	00	07	00	08	00	01	00	09	00	00	00	2F	00	01	00	01
000001b0h:	00	00	00	05	2A	B7	00	0A	B1	00	00	00	02	00	0C	00
000001c0h:	00	00	06	00	01	00	00	00	03	00	0D	00	00	00	0C	00
000001d0h:	01	00	00	00	05	00	0E	00	0F	00	00	00	01	00	10	00
000001e0h:	08	00	01	00	09	00	00	00	37	00	02	00	01	00	00	00
000001f0h:	09	B2	00	11	12	17	B6	00	19	B1	00	00	00	02	00	0C
00000200h:	00	00	00	0A	00	02	00	00	00	07	00	08	00	08	00	0D
00000210h:	00	00	00	0C	00	01	00	00	00	09	00	0E	00	0F	00	00
00000220h:	00	01	00	1F	00	00	00	02	00	20						

访问标志 类索引 父类索引 接口索引集合

常量池

接口索引集合区域，接口计数器为01，表示就实现了一个接口，该接口引用指向常量池的第5个常量池项

Constant pool:

#1 = Class

#2 = Utf8

#3 = Class

#4 = Utf8

#5 = Class

#6 = Utf8

#2

#4

#6

// com/louis/jvm/Programmer

com/louis/jvm/Programmer

// java/lang/Object

java/lang/Object

// com/louis/jvm/Worker

com/louis/jvm/Worker

Designed by LouLuan

http://blog.csdn.net/luanlouis

字段表示 = 作用域 + 静态或非静态 + 可变性 + 并发可见性 + 是否可序列化 + 数据类型 + 字段名称 + 其他属性

对应描述符

private
public
protected

static

final

volatile

transient

```
Field_info {  
    u2 access_flags;  
    u2 name_index;  
    u2 descriptor_index;  
    u2 attribute_count;  
    attribute_info attributes;  
}
```

JVM定义了field_info结构体来描述关于字段的描述。

作用域、静态或非静态、可变性、并发可见性、是否可序列化这些描述对字段来说就是有和无的关系，适合使用访问标志来表示。

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

Field_info



 表示一个字节

ConstantValue



属性名称索引的值是指向常量池中某个常量池项的索引，该常量池项表示的字符串为“ConstantValue”

属性长度中的值固定为2

常量池索引的值是指向常量池中某个常量池项的索引，表示要复制给field字段的值

Sample Code

1. `package com.louis.jvm;`

3. `public class Simple {`

5. `private transient static final String str ="This is a test";`

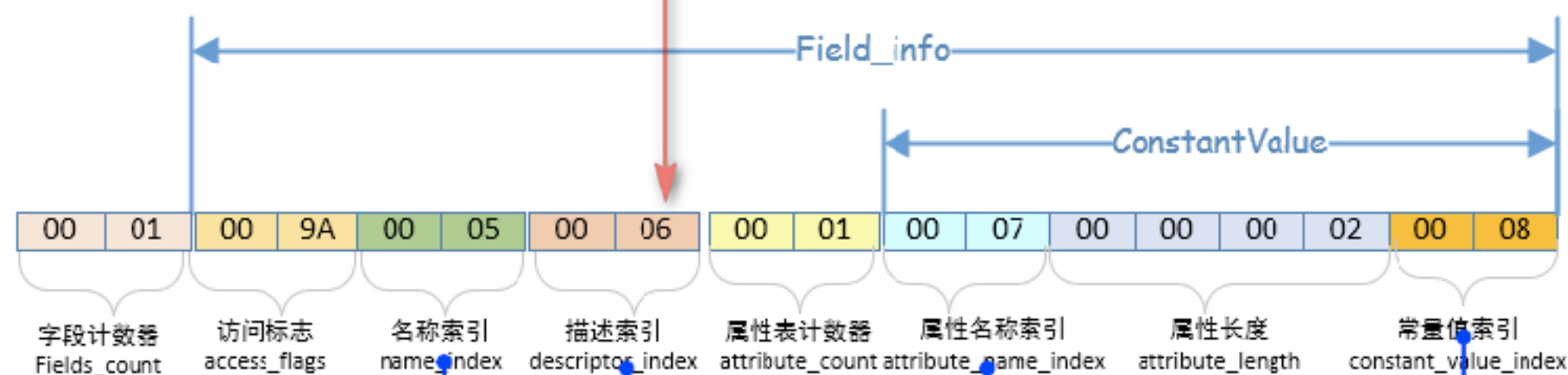
6. `}`

7.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	32	00	15	07	00	02	01	00	14
00000010h:	63	6F	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69
00000020h:	6D	70	6C	65	07	00	04	01	00	10	6A	61	76	61	2F	6C
00000030h:	61	6E	67	2F	4F	62	6A	65	63	74	01	00	03	73	74	72
00000040h:	01	00	12	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	74
00000050h:	72	69	6E	67	3B	01	00	0D	43	6F	6E	73	74	61	6E	74
00000060h:	56	61	6C	75	65	08	00	09	01	00	0E	54	68	69	73	20
00000070h:	69	73	20	61	20	74	65	73	74	01	00	06	3C	69	6E	69
00000080h:	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	0A
00000090h:	00	03	00	0E	0C	00	0A	00	0B	01	00	0F	4C	69	6E	65
000000a0h:	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	12	4C	6F
000000b0h:	63	61	6C	56	61	72	69	61	62	6C	65	54	61	62	6C	65
000000c0h:	01	00	04	74	68	69	73	01	00	16	4C	63	6F	6D	2F	6C
000000d0h:	6F	75	69	73	2F	6A	76	6D	2F	53	69	6D	70	6C	65	3B
000000e0h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0B
000000f0h:	53	69	6D	70	6C	65	2E	6A	61	76	61	00	21	00	01	00
00000100h:	03	00	00	00	01	00	9A	00	05	00	06	00	01	00	07	00
00000110h:	00	00	02	00	08	00	01	00	01	00	0A	00	0B	00	01	00
00000120h:	0C	00	00	00	2F	00	01	00	01	00	03	00	05	2A	B7	00
00000130h:	0D	B1	00	00	00	02	00	0F	00	00	03	06	00	01	00	00
00000140h:	00	03	00	10	00	00	00	0C	00	01	00	00	00	05	00	11
00000150h:	00	12	00	00	00	01	00	13	00	00	00	02	00	00	14	

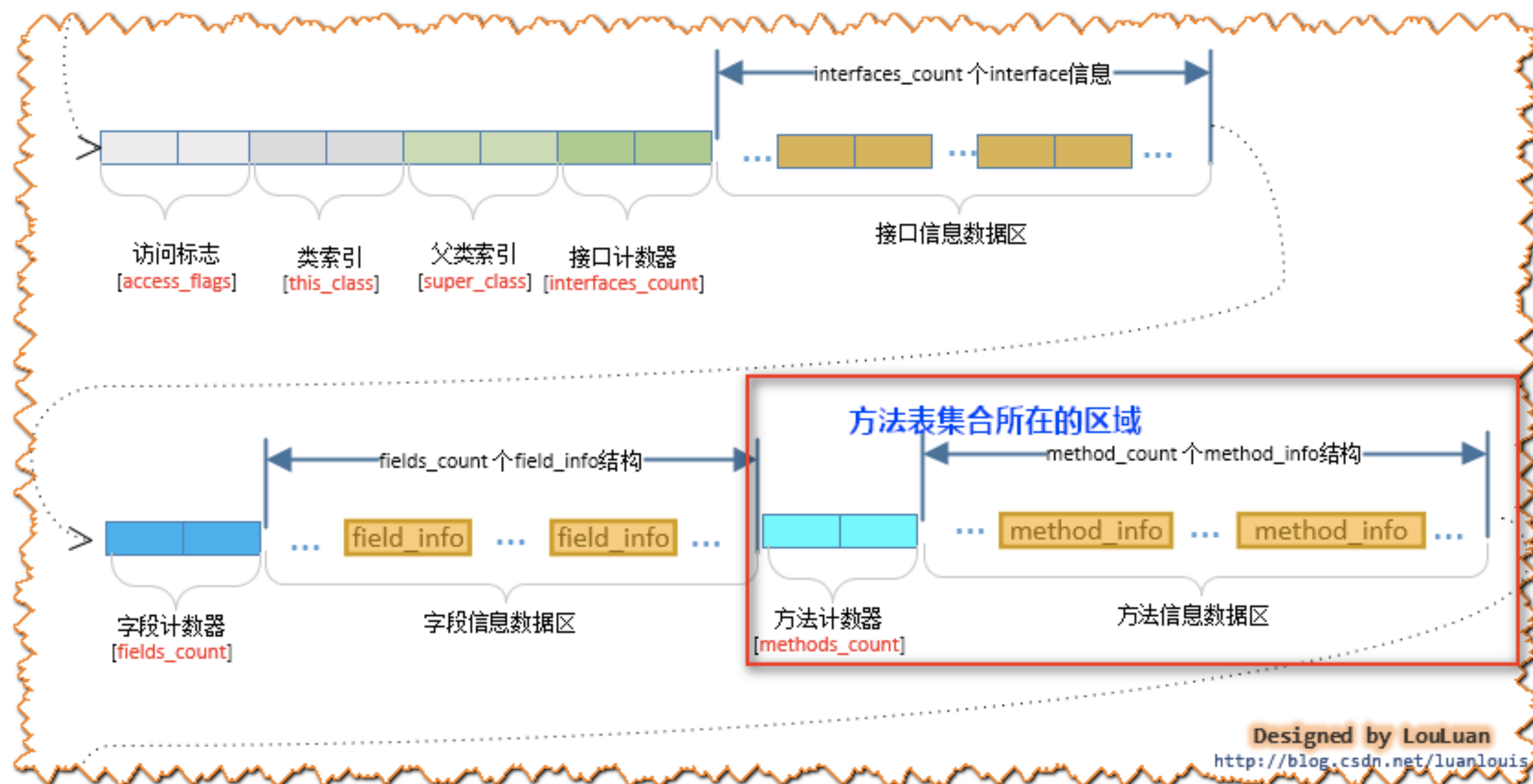
常量池

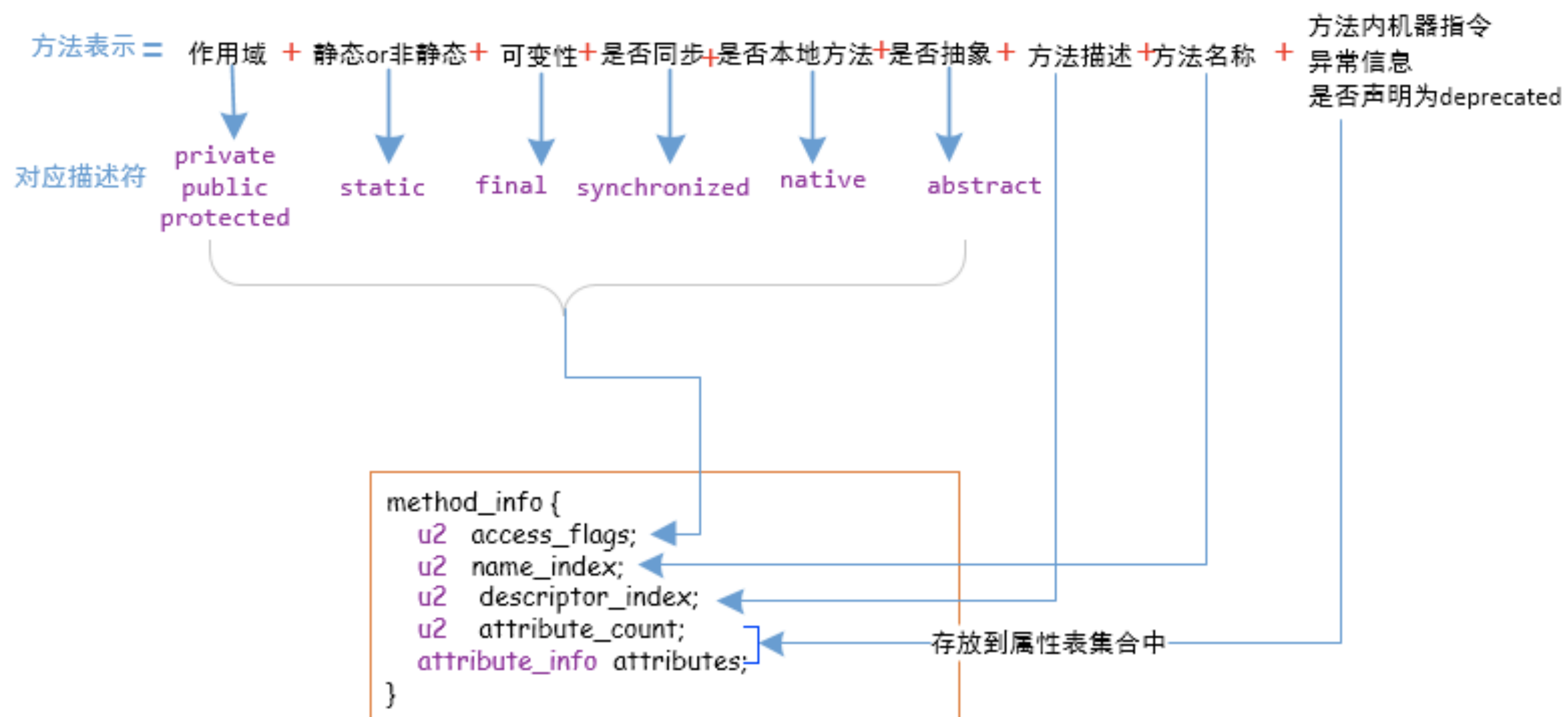
字段表集合



```

Constant pool:
#1 = Class           #2          // com/louis/jvm/Simple
#2 = Utf8            com/louis/jvm/Simple
#3 = Class           #4          // java/lang/Object
#4 = Utf8            java/lang/Object
#5 = Utf8            str
#6 = Utf8            java/lang/String;
#7 = Utf8            ConstantValue
#8 = String          #9          // This is a test
#9 = Utf8            This is a test
  
```





Method_info



 表示一个字节

Attribute_info



Attribute_info



“Code” ,表明此属性表(attribute_info)记录的是机器码

“Exceptions” ,此属性表记录的是异常信息

“Deprecated” ,此属性表被@Deprecated修饰

“Synthetic” ,此属性表表示被编译器自动生成

属性名称索引 (attribute_name_index) 占有**两个**字节，其中的值是指向了常量池中的某一项，该项表示的字符串表明该attribute_info表示的是什么类型的属性表。如：如上所示，如果指向的常量池项表示“Code”，则表明该属性表(attribute_info)表示的是代码实现，即可执行的机器码；

对于不同类型的属性表，它们的**属性长度和组织形式**会有所不同。

属性长度(attribute_length)占有两个字节，它的值表示紧跟其后的多少个字节是拿来表示这个属性信息的。即如果属性长度中的值为N，则表示它后面的N个字节是用来表示属性信息的。

属性值(attribute_value)是由若干个字节构成的，字节的个数由属性长度中的值决定。

Code属性表信息 = 机器指令 + 异常处理跳转信息 + Java源码行号和机器码对应关系 + 局部变量表描述信息

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  {
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  }
  exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

以LineNumberTable类型的属性表保存在属性表集合中

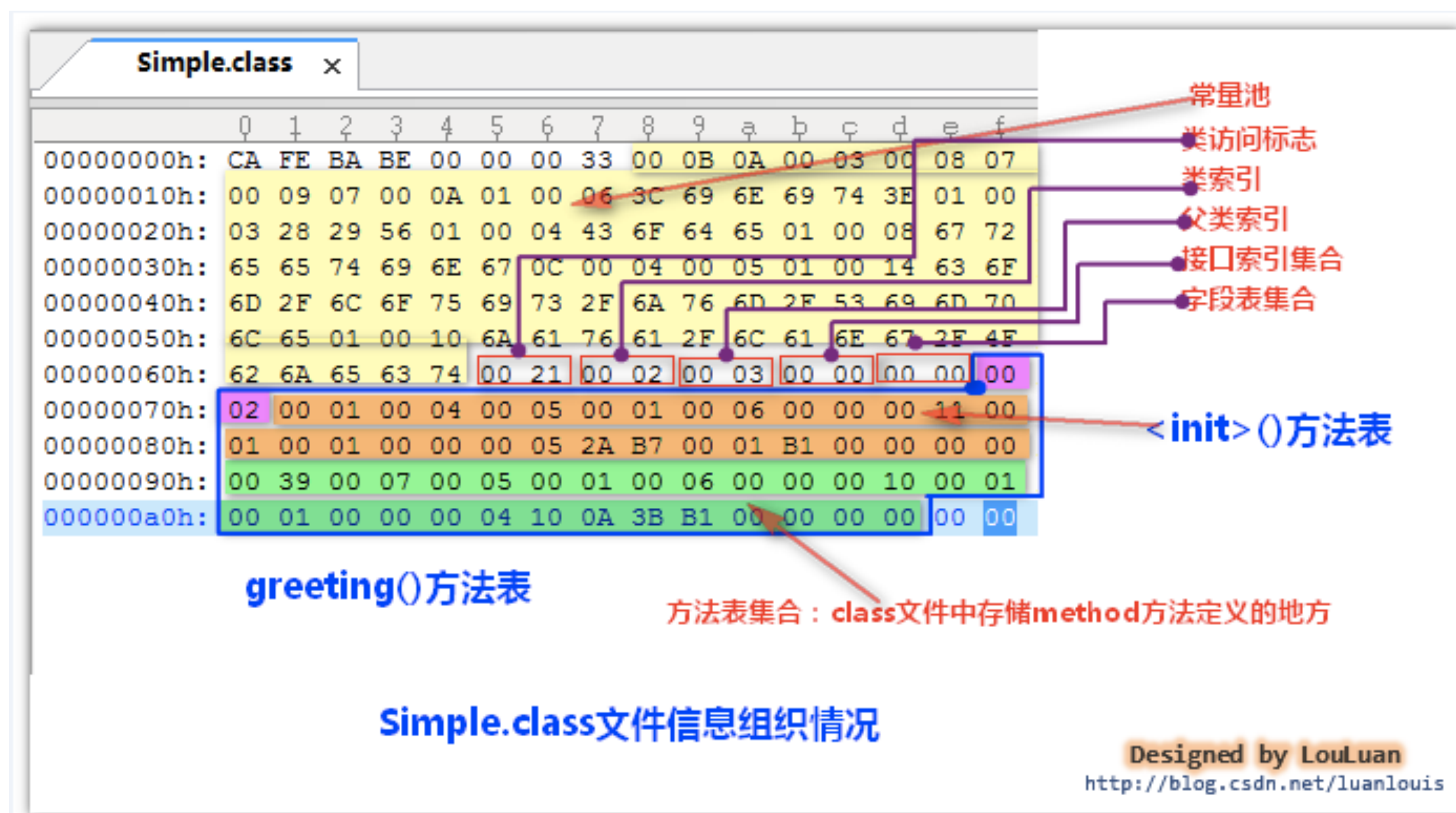
以LocalVariableTable类型的属性表保存在属性表集合中

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

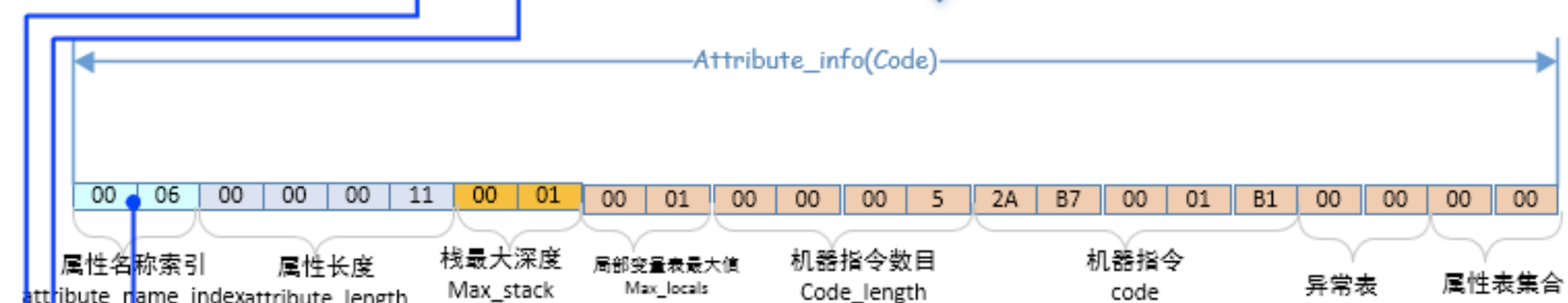
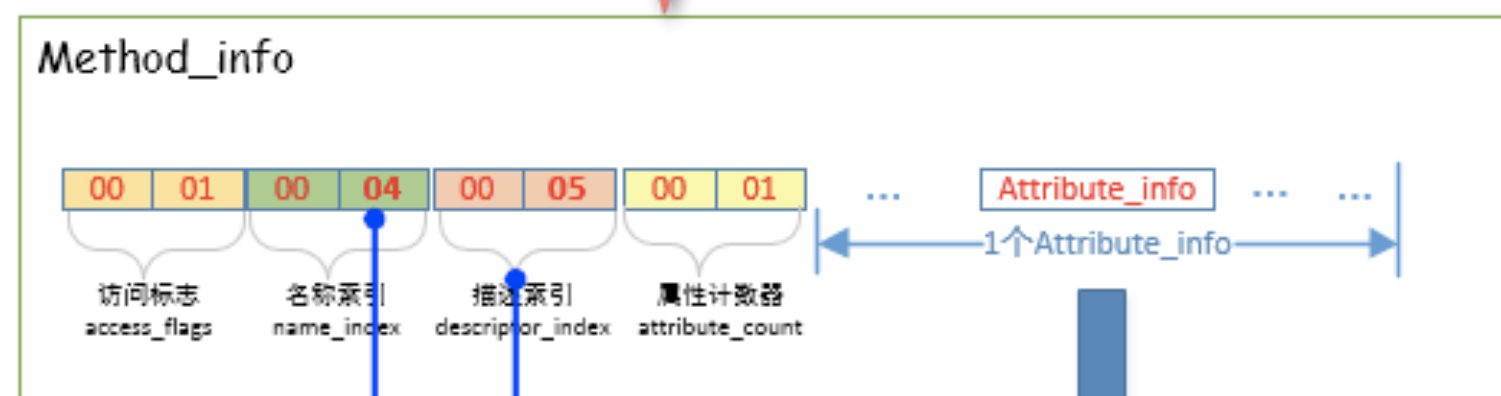
Sample Code - 方法

```
1. package com.louis.jvm;  
2.  
3. public class Simple {  
4.  
5.     public static synchronized final void greeting(){  
6.         int a = 10;  
7.     }  
8. }
```



	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	33	00	0B	0A	00	03	00	08	07
00000010h:	00	09	07	00	0A	01	00	06	3C	69	6E	69	74	3E	01	00
00000020h:	03	28	29	56	01	00	04	43	6F	64	65	01	00	08	67	72
00000030h:	65	65	74	69	6E	67	0C	00	04	00	05	01	00	14	63	6F
00000040h:	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69	6D	70
00000050h:	6C	65	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F
00000060h:	62	6A	65	63	74	00	21	00	02	00	03	00	00	00	00	00
00000070h:	02	00	01	00	04	00	05	00	01	00	06	00	00	00	11	00
00000080h:	01	00	01	00	00	00	05	2A	B7	00	01	B1	00	00	00	00
00000090h:	00	39	00	07	00	05	00	01	00	06	00	00	00	10	00	01
000000a0h:	00	01	00	00	00	04	10	0A	3B	B1	00	00	00	00	00	00

<init>()方法表信息



Constant pool:

#1 = Methodref	#3.#8	// java/lang/Object."<init>":()V
#2 = Class	#9	// com/louis/jvm/Simple
#3 = Class	#10	// java/lang/Object
#4 = Utf8	<init>	
#5 = Utf8	()V	
#6 = Utf8	Code	
#7 = Utf8	greeting	
#8 = NameAndType	#4:#5	// "<init>":()V
#9 = Utf8	com/louis/jvm/Simple	
#10 = Utf8	java/lang/Object	

```
public com.louis.jvm.Simple();  
  flags: ACC_PUBLIC
```

Code:

```
  stack=1, locals=1, args_size=1
```

```
    0: aload_0      2A
```

```
    1: invokespecial #1 B7 00 01
```

```
    4: return      B1
```

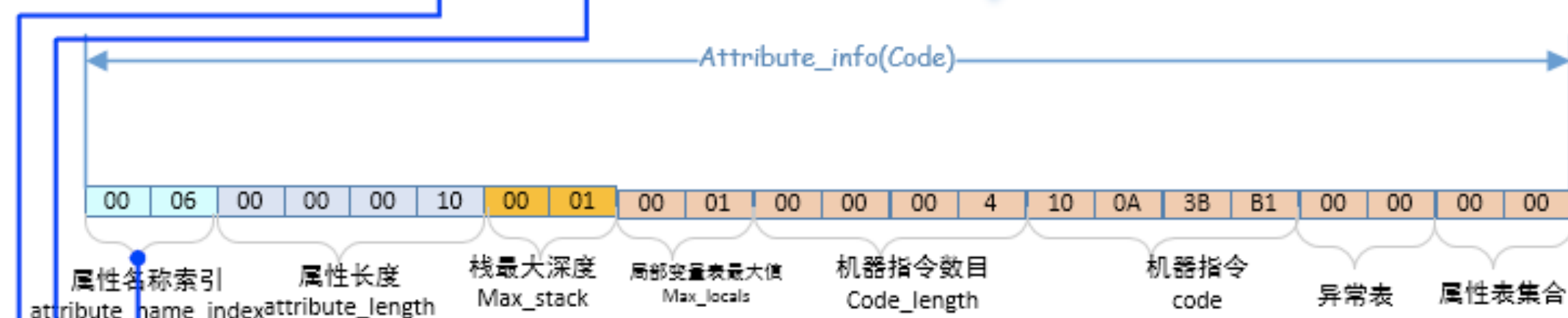
```
// Method java/lang/Object."<init>":()V
```

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	33	00	0B	0A	00	03	00	08	07
00000010h:	00	09	07	00	0A	01	00	06	3C	69	6E	69	74	3E	01	00
00000020h:	03	28	29	56	01	00	04	43	6F	64	65	01	00	08	67	72
00000030h:	65	65	74	69	6E	67	0C	00	04	00	05	01	00	14	63	6F
00000040h:	6D	2F	6C	6F	75	69	73	2F	6A	76	6D	2F	53	69	6D	70
00000050h:	6C	65	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F
00000060h:	62	6A	65	63	74	00	21	00	02	00	03	00	00	00	00	00
00000070h:	02	00	01	00	04	00	05	00	01	00	06	00	00	00	11	00
00000080h:	01	00	01	00	00	00	05	2A	B7	00	01	B1	00	00	00	00
00000090h:	00	39	00	07	00	05	00	01	00	06	00	00	00	10	00	01
000000a0h:	00	01	00	00	00	04	10	0A	3B	B1	00	00	00	00	00	00

Method_info



Constant pool:

#1 = Methodref	#3.#8	// java/lang/Object."<init>":()V
#2 = Class	#9	// com/louis/jvm/Simple
#3 = Class	#10	// java/lang/Object
#4 = Utf8	<init>	
#5 = Utf8	()V	
#6 = Utf8	Code	
#7 = Utf8	greeting	
#8 = NameAndType	#4:#5	// "<init>":()V
#9 = Utf8	com/louis/jvm/Simple	
#10 = Utf8	java/lang/Object	

```
public static final synchronized void greeting();  
    flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_SYNCHRONIZED
```

Code:

```
    stack=1, locals=1, args_size=0
```

```
    0: bipush      10    10 0A : 将单字节10 推送到栈上
```

```
    2: istore_0      3B : 取栈顶的int类型的值赋给局部变量表的第一个变量
```

```
    3: return    B1 : 无结果返回
```

```
}
```

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

实例文件

- package org.fenixsoft.clazz;
-
- public class TestClass{
-
- private int m;
-
- public int inc(){
- return m+1;
- }
- }

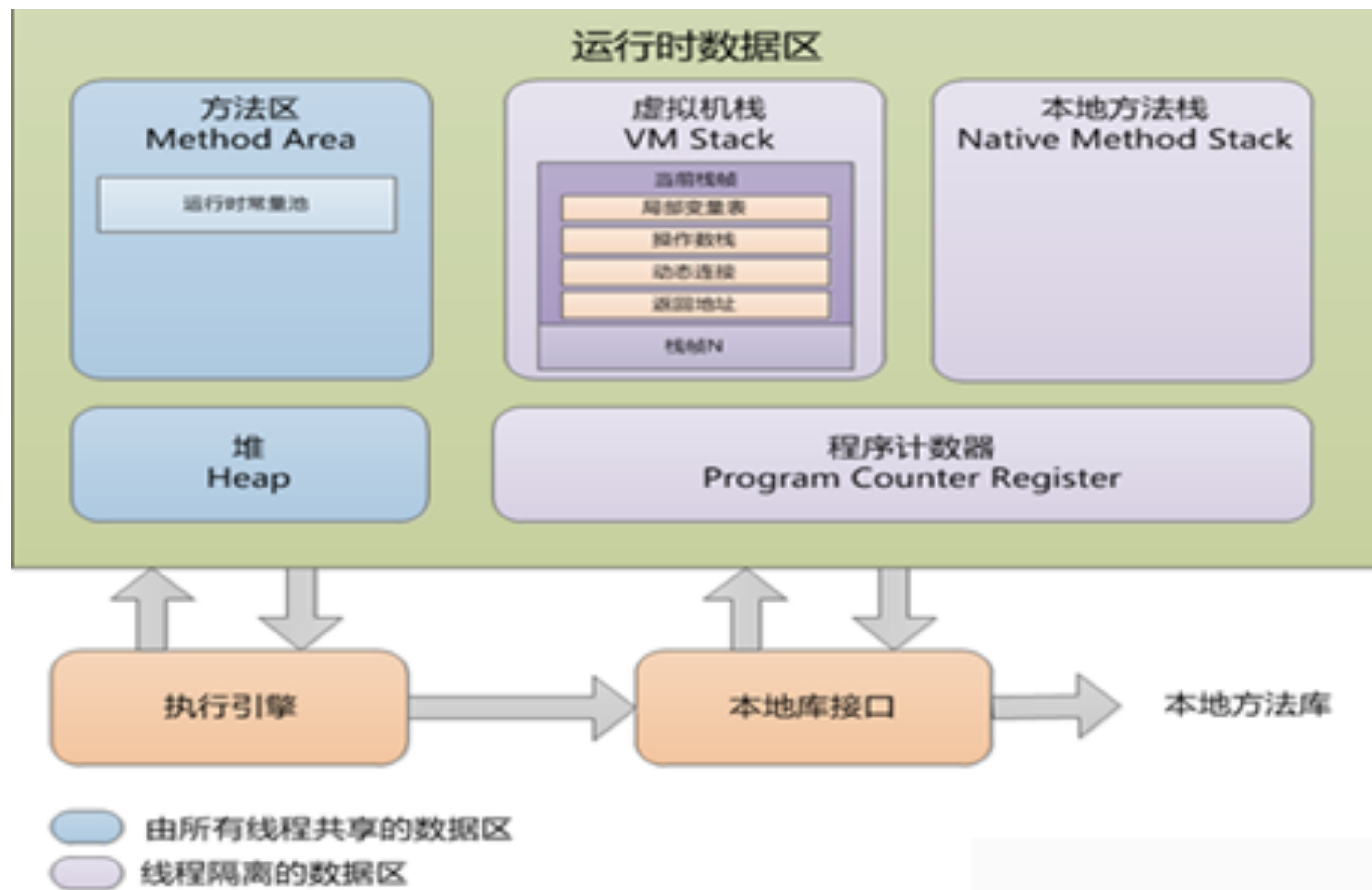

```
promote:~ qinliu$ javac TestClass.java
promote:~ qinliu$ javap -verbose TestClass
警告: 二进制文件TestClass包含org.fenixsoft.clazz.TestClass
Classfile /Users/qinliu/TestClass.class
    Last modified 2015-4-15; size 295 bytes
    MD5 checksum 81f2ab948a7a3068839b61a8f91f634b
    Compiled from "TestClass.java"
public class org.fenixsoft.clazz.TestClass
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref      #4.#15      // java/lang/Object."<init>":()V
   #2 = Fieldref       #3.#16      // org/fenixsoft/clazz/TestClass.m:I
   #3 = Class          #17         // org/fenixsoft/clazz/TestClass
   #4 = Class          #18         // java/lang/Object
   #5 = Utf8           m
   #6 = Utf8           I
   #7 = Utf8           <init>
   #8 = Utf8           ()V
   #9 = Utf8           Code
  #10 = Utf8           LineNumberTable
  #11 = Utf8           inc
  #12 = Utf8           ()I
  #13 = Utf8           SourceFile
  #14 = Utf8           TestClass.java
  #15 = NameAndType    #7:#8       // "<init>":()V
  #16 = NameAndType    #5:#6       // m:I
  #17 = Utf8           org/fenixsoft/clazz/TestClass
  #18 = Utf8           java/lang/Object

{
    public org.fenixsoft.clazz.TestClass();
        descriptor: ()V
        flags: ACC_PUBLIC
        Code:
            stack=1, locals=1, args_size=1
            0: aload_0
            1: invokespecial #1             // Method java/lang/Object.<init>:()V
            4: return
        LineNumberTable:
           line 3: 0
    public int inc();
        descriptor: ()I
        flags: ACC_PUBLIC
        Code:
            stack=2, locals=1, args_size=1
            0: aload_0
            1: getfield      #2             // Field m:I
            4: iconst_1
            5: iadd
            6: ireturn
        LineNumberTable:
           line 8: 0
}
SourceFile: "TestClass.java"
```

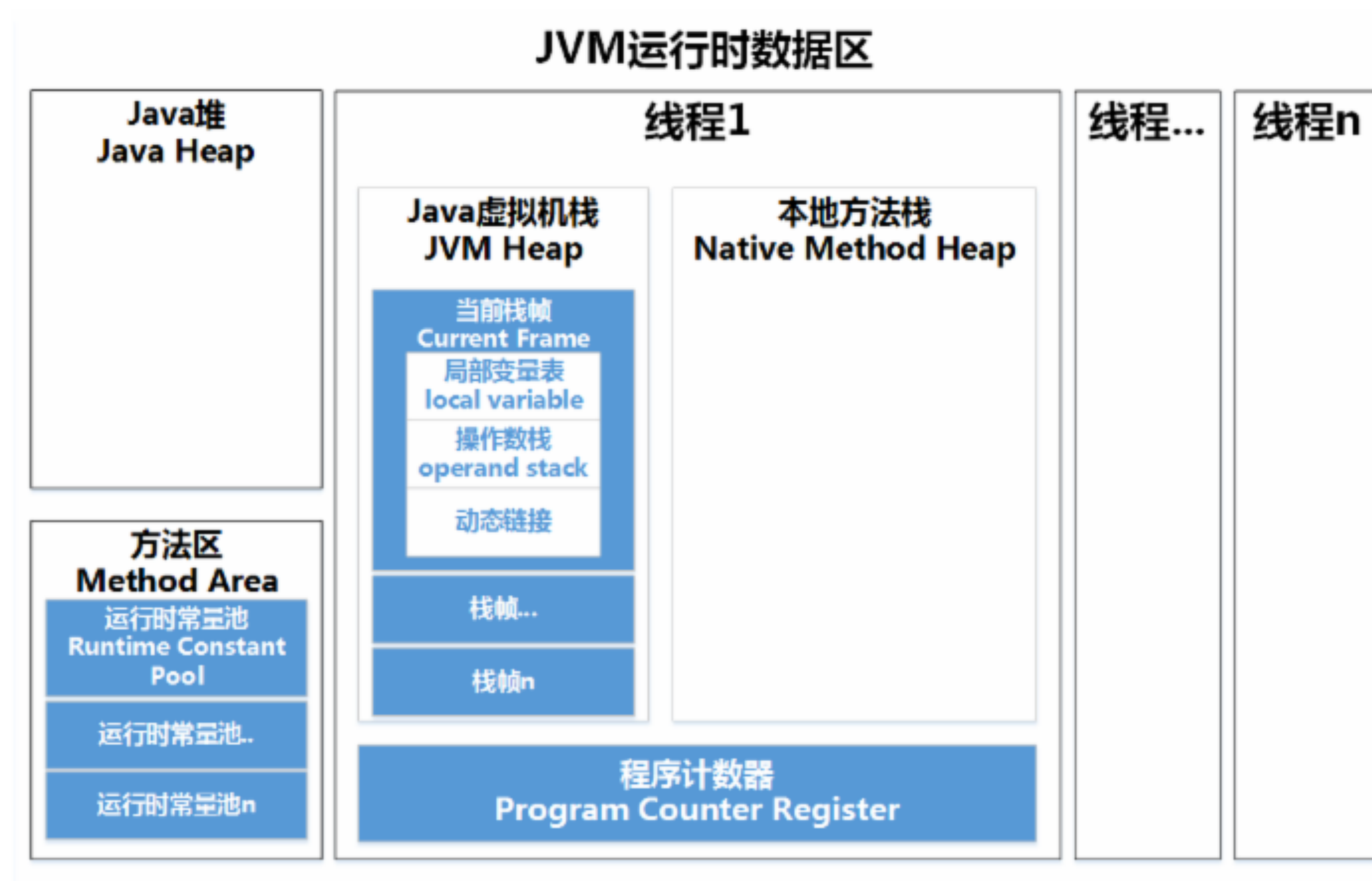
3 运行时数据区

JVM Run-Time Data Areas

- The pc Register
- Java Virtual Machine Stacks
- Heap
- Method Area
- Run-Time Constant Pool
- Native Method Stacks

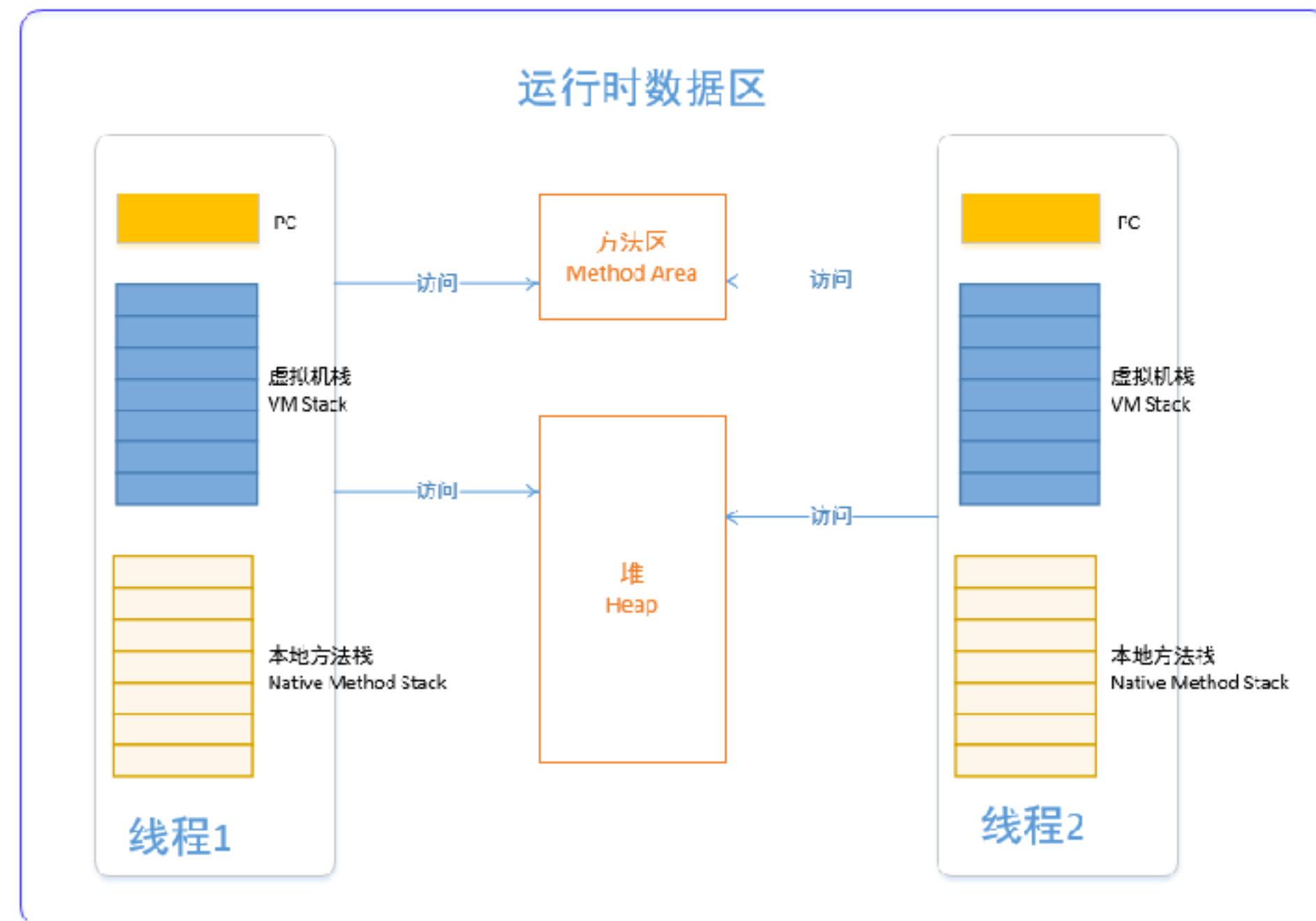


Java虚拟机运行时数据区



多线程

Topic 1.JVM运行时数据区里有什么？



PC：即程序计数器（**Program Counter Register**，一块很小的内存空间，存储了下一条需要执行的（JVM 汇编）字节码指令的地址。JVM多线程是通过线程轮换并分配执行时间的方式实现的。在任何一个确定的时间内，JVM只会执行其中一条指令。每个线程的PC记录了当前线程要执行的指令，每个线程都有自己的PC，这样他们之间不会相互影响。如果当前线程执行的是Java方法，其PC记录的是正在执行的虚拟机字节码指令的地址；如果执行的是本地方法（Native Method），则PC为空。

虚拟机栈（VM stack）：此栈中的元素叫做栈帧（**Stack Frame**），线程在调用Java方法时，会为每一个方法创建一个栈帧，来存储局部变量表、操作栈、动态链接、方法出口等信息。每个方法被调用和完成的过程，都对应一个栈帧从虚拟机栈上入栈和出栈的过程。虚拟机栈的生命周期和线程相同。

本地方法栈（Native Method Stack）：为线程私有，功能和虚拟机栈非常相似。来存储线程调用本地方法时，本地方法的局部变量表，操作栈等等信息。

堆（Heap）：Heap区被所有的线程共享，在虚拟机启动时创建。此区的功能就是存放对象实例，几乎所有的对象实例都是在这里分配内容。Heap区垃圾回收器管理的主要区域。

方法区（Method Area）：该区为各个线程共享，用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译出来的代码等数据。

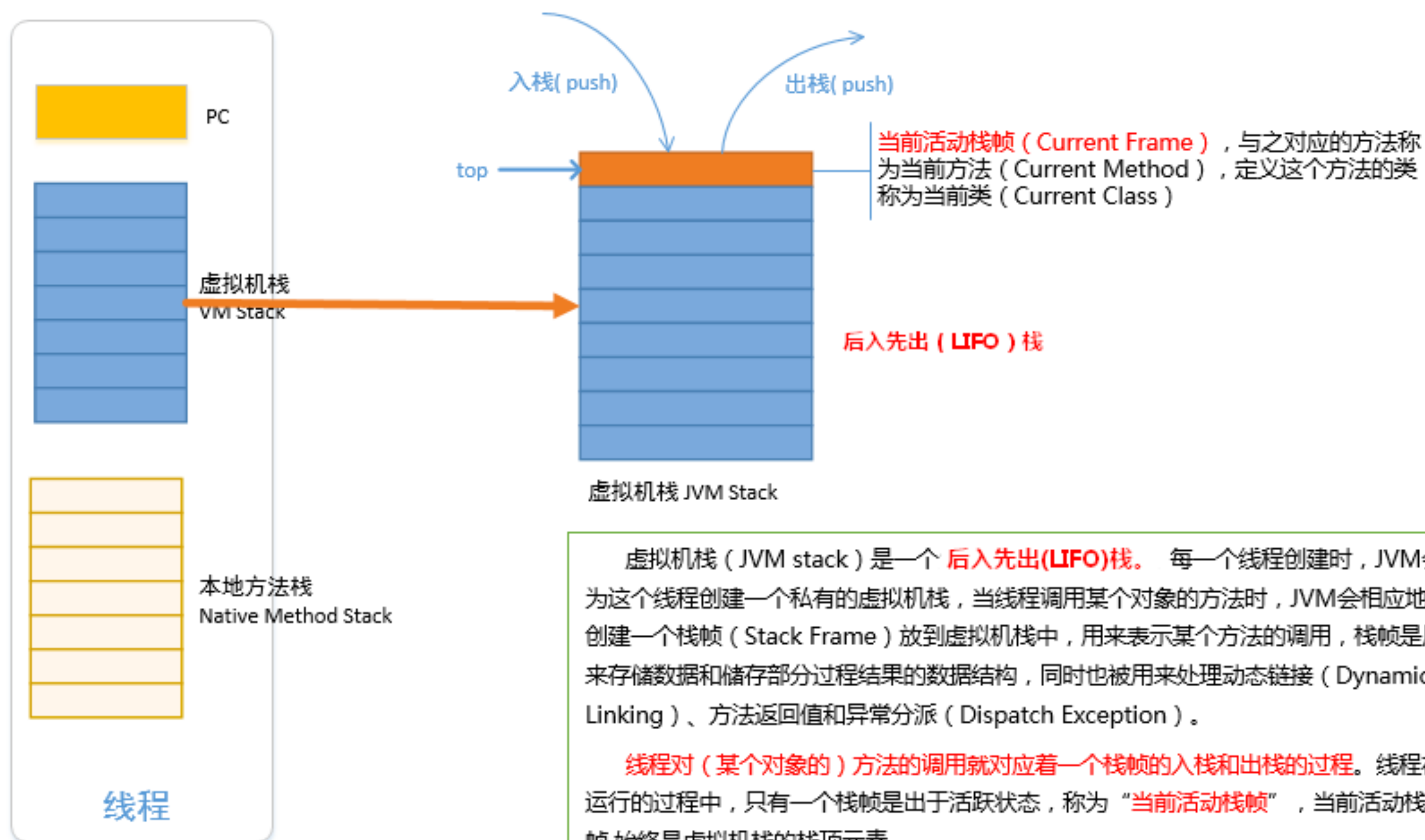
运行时常量池

- 存放编译器生成的
 - 各种字面量
 - numeric literals
 - string literals
 - 符号引用
 - class references
 - field references
 - method references

The constant pool contains the following types:

Integer	A 4 byte int constant
Long	An 8 byte long constant
Float	A 4 byte float constant
Double	A 8 byte double constant
String	A String constant that points at another Utf8 entry in the constant pool which contains the actual bytes
Utf8	A stream of bytes representing a Utf8 encoded sequence of characters
Class	A Class constant that points at another Utf8 entry in the constant pool which contains the fully qualified class name in the internal JVM format (this is used by the dynamic linking process)
NameAndType	A colon separated pair of values each pointing at other entries in the constant pool. The first value (before the colon) points at a Utf8 string entry that is the method or field name. The second value points at a Utf8 entry that represents the type, in the case of a field this is the fully qualified class name, in the case of a method this is a list of fully qualified class names one per parameter.
Fieldref, Methodref, InterfaceMethodref	A dot separated pair of values each pointing at other entries in the constant pool. The first value (before the dot) points at a Class entry. The second value points at a NameAndType entry.

Topic 2. 虚拟机栈是什么？虚拟机栈里有什么？



虚拟机栈 (JVM stack) 是一个 **后入先出(LIFO)** 栈。每一个线程创建时，JVM会为此线程创建一个私有的虚拟机栈，当线程调用某个对象的方法时，JVM会相应地创建一个栈帧 (Stack Frame) 放到虚拟机栈中，用来表示某个方法的调用，栈帧是用来存储数据和储存部分过程结果的数据结构，同时也被用来处理动态链接 (Dynamic Linking)、方法返回值和异常分派 (Dispatch Exception)。

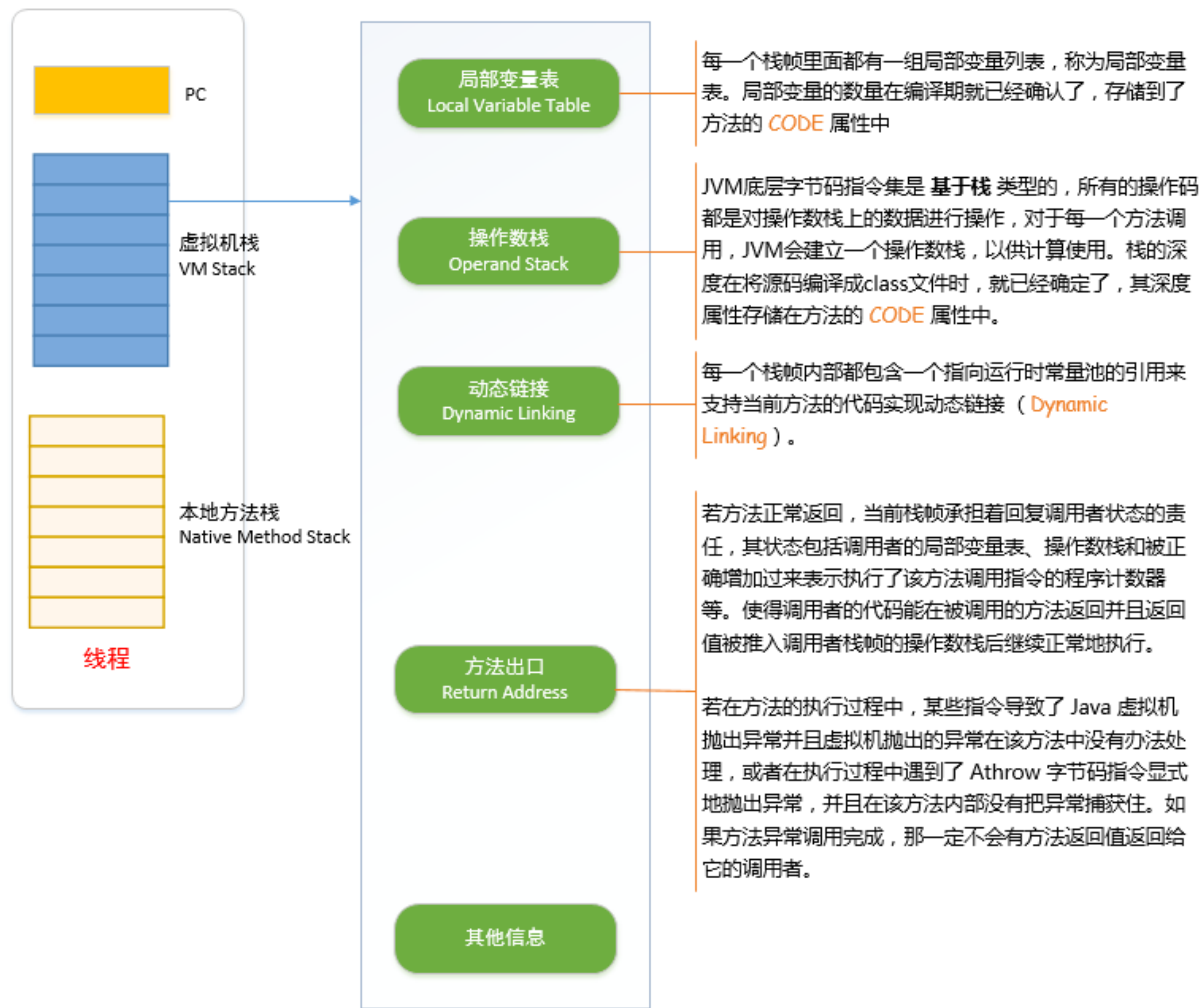
线程对 (某个对象的) 方法的调用就对应着一个栈帧的入栈和出栈的过程。线程在运行的过程中，只有一个栈帧是出于活跃状态，称为“**当前活动栈帧**”，当前活动栈帧始终是虚拟机栈的栈顶元素。

TIPS :

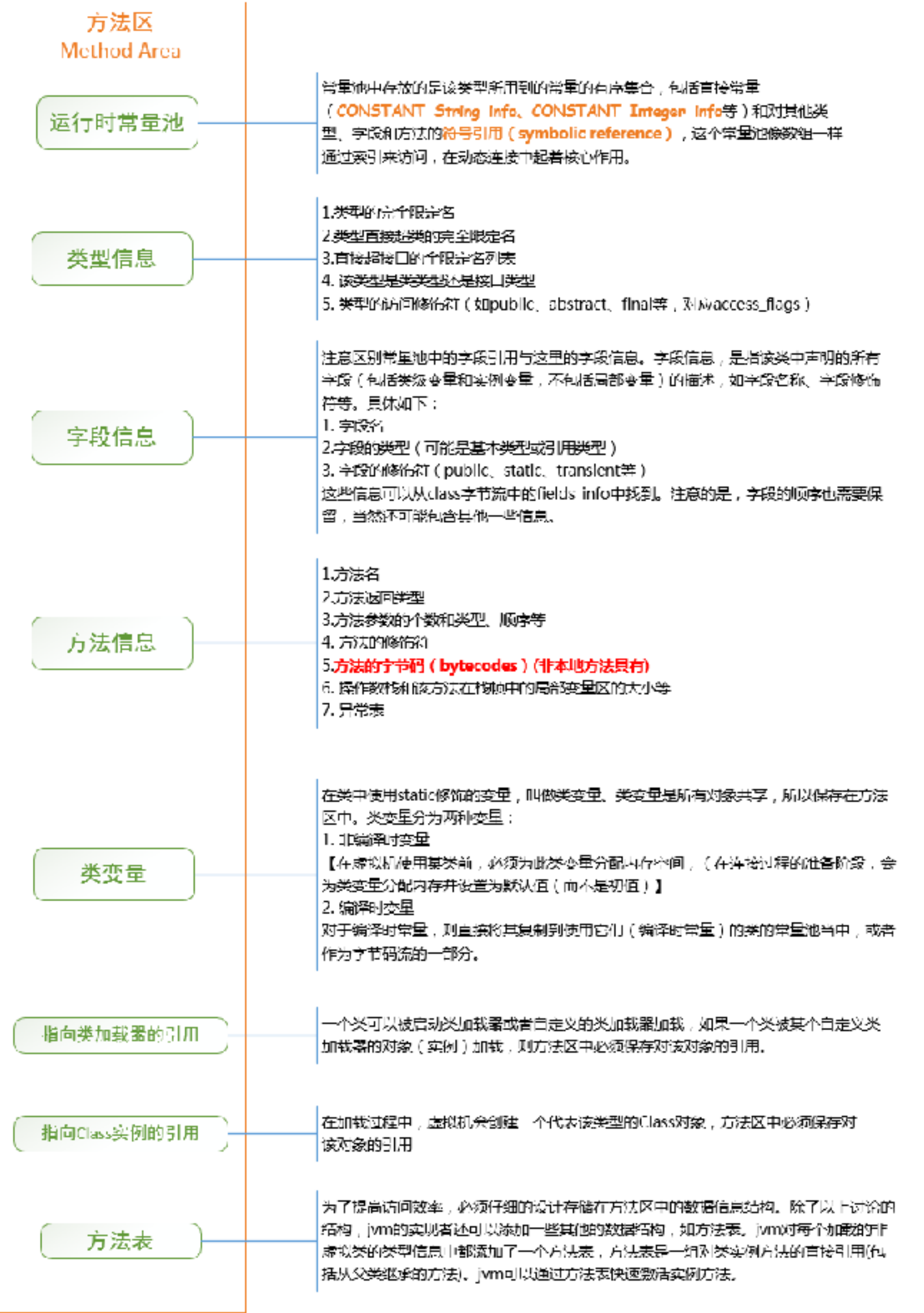
在线程创建的时候，都会为其创建一个私有的虚拟机栈，即虚拟机栈的生命周期跟线程是一样的

Topic 3. 栈帧是什么？栈帧里有什么？

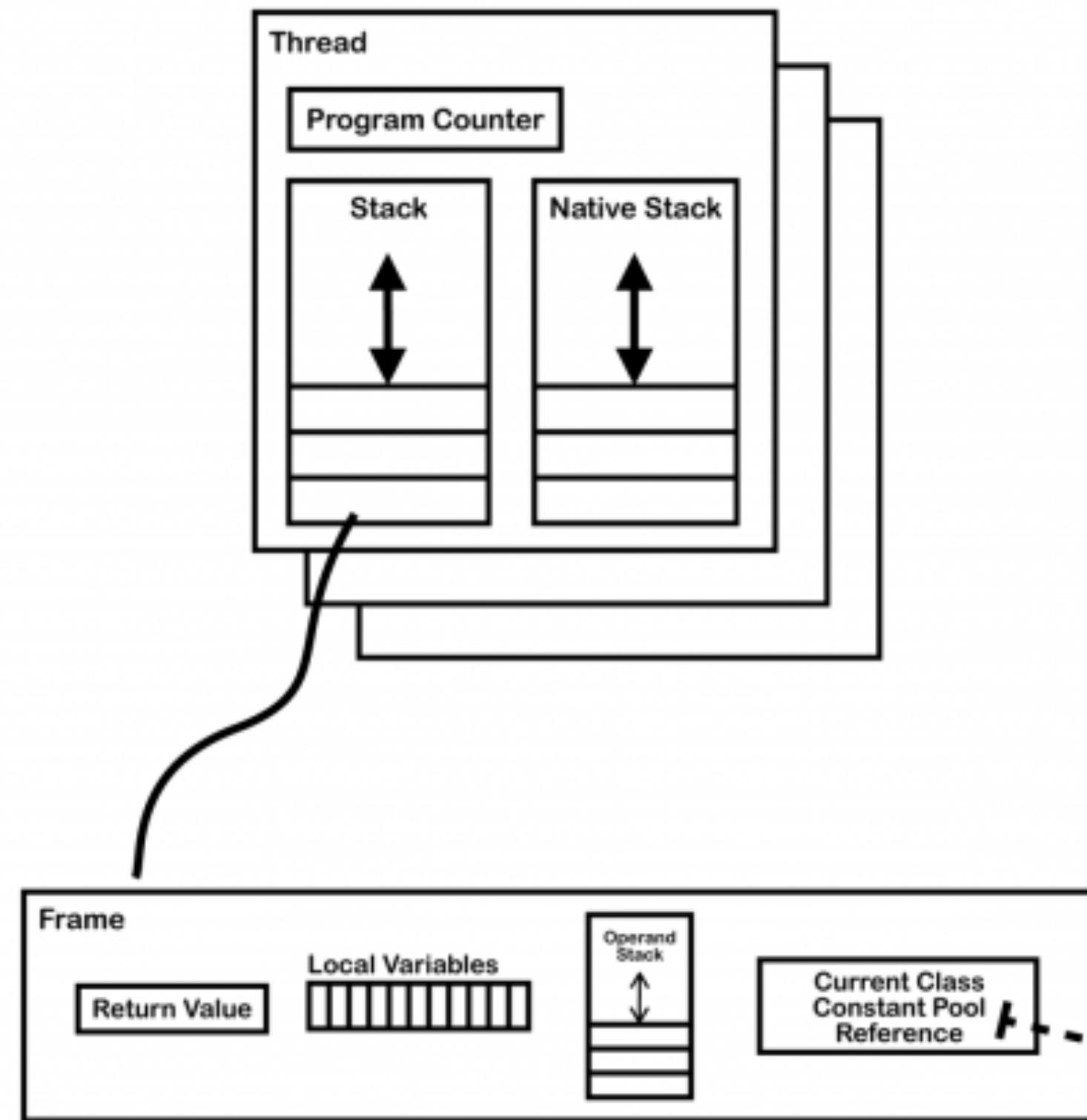
栈帧 Stack Frame



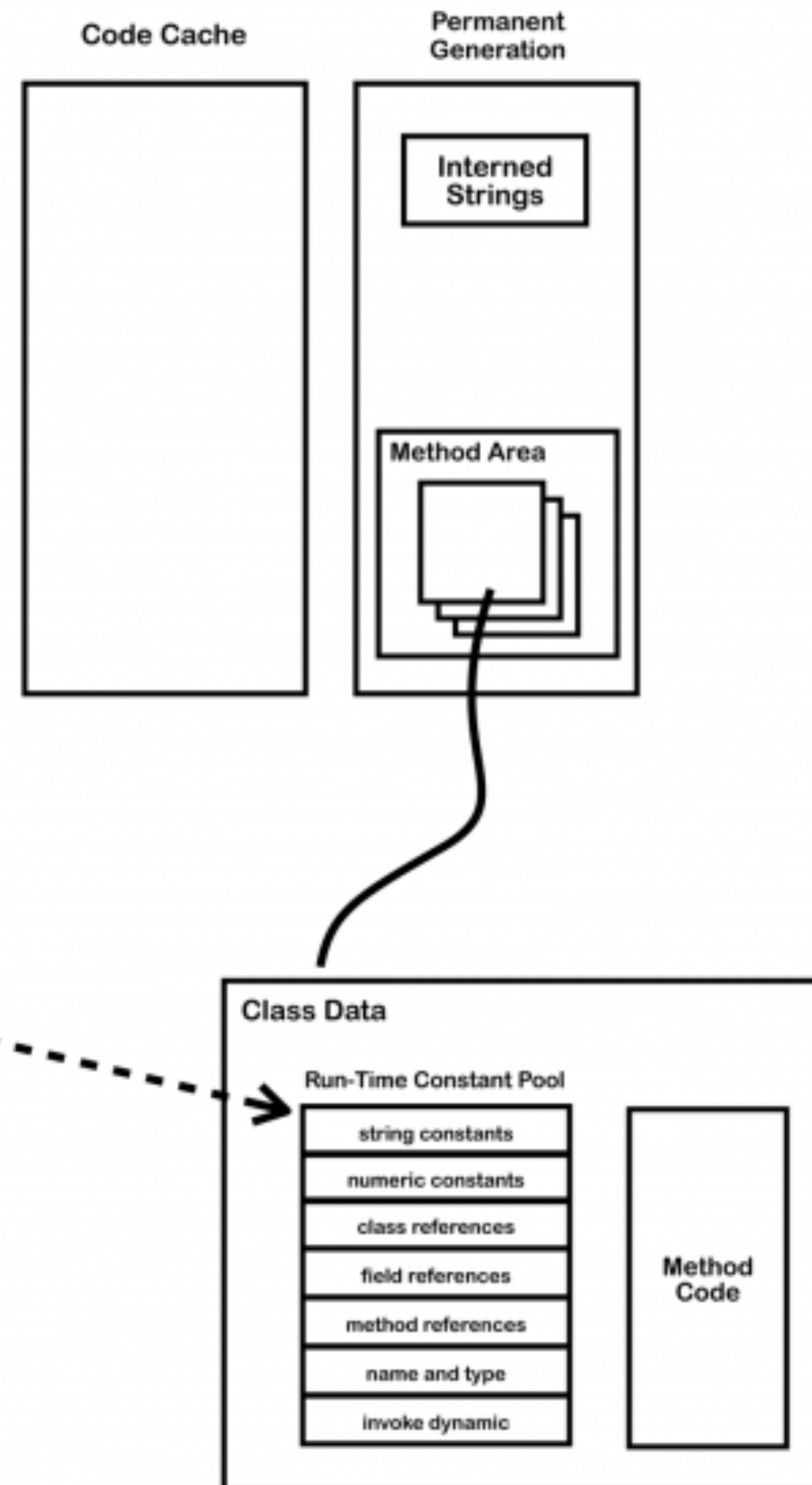
Topic 4. 方法区是什么？方法区里有什么？



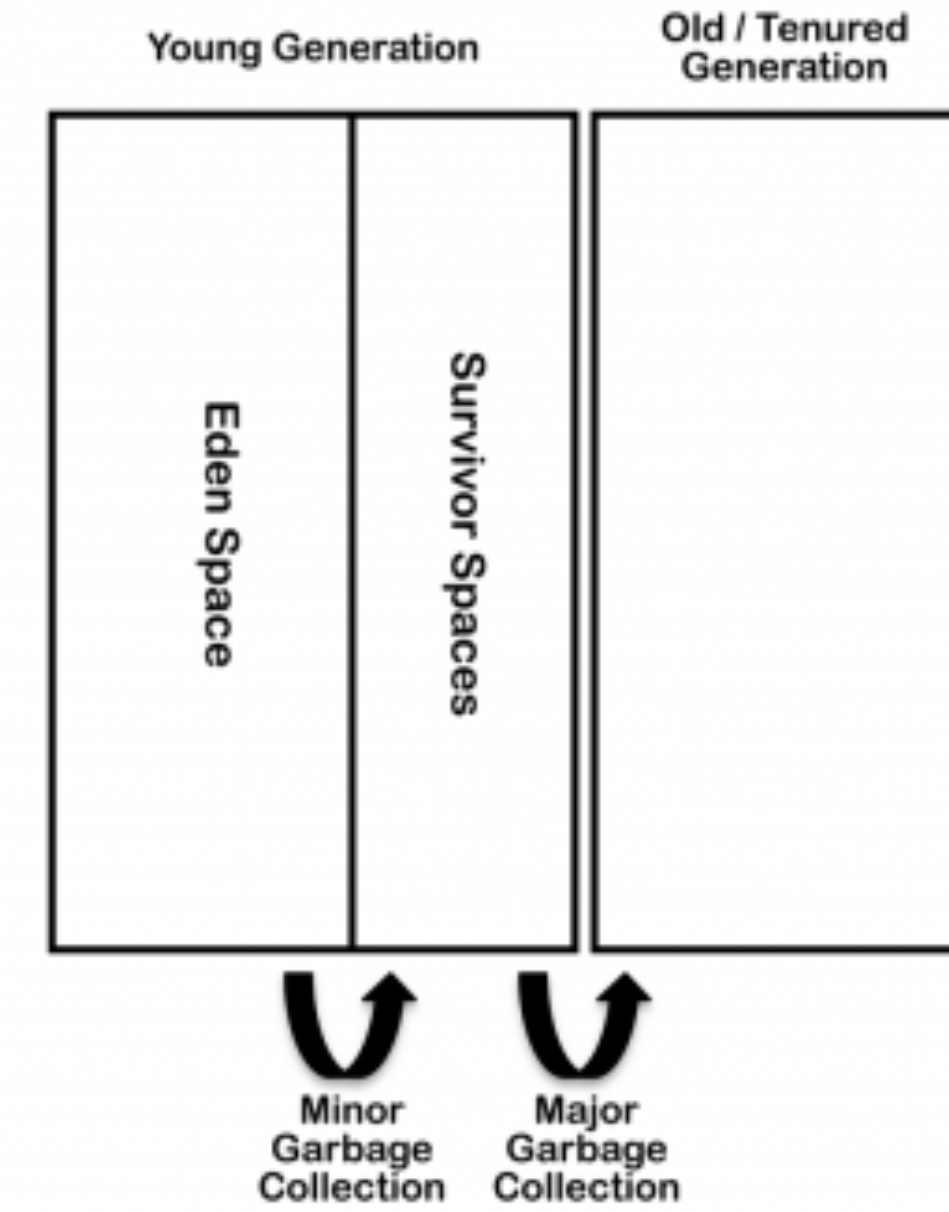
Stack



Non Heap



Heap



String.intern()

- public class RuntimeConstantPoolOOM {
- public static void main(String[] args){
- String str1 = new StringBuilder("计算机").append("软件").toString();
- System.out.println(str1.intern() == str1);
-
- String str2 = new StringBuilder("ja").append("va").toString();
- System.out.println(str2.intern() == str2);
- }
- }

解释

- jdk1.6
 - false
 - false
- idk 1.7
 - true
 - false
- <http://www.jianshu.com/p/b98851899f37>

栈帧中的局部变量表中的槽位是可以重用的

- `public void localvar1(){`
 - `int a = 0;`
 - `System.out.println(a);`
 - `int b = 0;`
- `}`
- `public void localvar1(){`
 - `{`
 - `int a = 0;`
 - `System.out.println(a);`
 - `}`
 - `int b = 0;`
- `}`

对垃圾回收的影响

- `public void localvarGc1(){`
 - `byte[] a = new byte[6*1024*1024];`
 - `System.gc();`
- `}`
- `public void localvarGc2(){`
 - `byte[] a = new byte[6*1024*1024];`
 - `a = null;`
 - `System.gc();`
- `}`
- `public void localvarGc3(){`
 - `{`
 - `byte[] a = new byte[6*1024*1024];`
 - `System.gc();`
- `}`
- `}`
- `public void localvarGc4(){`
 - `{`
 - `byte[] a = new byte[6*1024*1024];`
 - `}`
 - `int c = 10;`
 - `System.gc();`
- `}`
- `public void localvarGc5(){`
 - `localvarGc1();`
 - `System.gc();`
- `}`

解释

1. 申请了一个6M大小的空间，赋值给b引用，然后调用gc函数，因为此时这个6M的空间还被b引用着，所以不能顺利gc；
2. 申请了一个6M大小的空间，赋值给b引用，然后将b重新赋值为null，此时这个6M的空间不再被b引用，所以可以顺利gc；
3. 申请了一个6M大小的空间，赋值给b引用，过了b的作用返回之后调用gc函数，但是因为此时b并没有被销毁，还存在于栈帧中，这个空间也还被b引用，所以不能顺利gc；
4. 申请了一个6M大小的空间，赋值给b引用，过了b的作用返回之后重新创建一个变量c，此时这个新的变量会复用已经失效的b变量的槽位，所以b被迫销毁了，所以6M的空间没有被任何变量引用，于是能够顺利gc；
5. 首先调用localVarGc1()，很显然不能顺利gc，函数调用结束之后再调用gc函数，此时因为localVarGc1这个函数的栈帧已经随着函数调用的结束而被销毁，b也就被销毁了，所以6M大小的空间不被任何对象引用，于是能够顺利gc。

- 文 / winwill2012（简书作者）
- 原文链接：<http://www.jianshu.com/p/6060cc53aca7>
- 著作权归作者所有，转载请联系作者获得授权，并标注“简书作者”。

4 字节码指令集

指令分类

- 指令可以基本分为以下几类：
- 存储指令 （例如： `aload_0`, `istore`）
- 算术与逻辑指令 （例如: `ladd`, `fcmpl`）
- 类型转换指令 （例如： `i2b`, `d2i`）
- 对象创建与操作指令 （例如： `new`, `putfield`）
- 堆栈操作指令 （例如： `swap`, `dup2`）
- 控制转移指令 （例如： `ifeq`, `goto`）
- 方法调用与返回指令 （例如： `invokespecial`, `areturn`）

前后缀

- 前/后缀 操作数类型
- i 整数
- l 长整数
- s 短整数
- b 字节
- c 字符
- f 单精度浮点数
- d 双精度浮点数
- z 布尔值
- a 引用

加载和存储指令

- 加载和存储指令用于将数据从栈帧的局部变量表和操作数栈之间来回传输。
- 1)将一个局部变量加载到操作数栈的指令包括：iload,iload_<n>, lload、lload_<n>、fload、fload_<n>、dload、dload_<n>, aload、aload_<n>。
- 2)将一个数值从操作数栈存储到局部变量表的指令：
istore,istore_<n>,lstore,lstore_<n>,fstore,fstore_<n>,dstore,dstore_<n>,astore,astore_<n>
- 3)将常量加载到操作数栈的指令：
bipush,sipush,ldc,ldc_w,ldc2_w,acost_null,iconst_ml,iconst_<i>,lconst_<l>,fconst_<f>,dconst_<d>
- 4)局部变量表的访问索引指令:wide
- 一部分以尖括号结尾的指令代表了一组指令，如iload_<i>, 代表了iload_0,iload_1等，这几组指令都是带有一个操作数的通用指令。

运算指令

- 算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。
- 1)加法指令:iadd,ladd,fadd,dadd
- 2)减法指令:isub,lsub,fsub,dsub
- 3)乘法指令:imul,lmul,fmul,dmul
- 4)除法指令:idiv,ldiv,fdiv,ddiv
- 5)求余指令:irem,lrem,frem,drem
- 6)取反指令:ineg,leng,fneg,dneg
- 7)位移指令:ishl,ishr,iushr,lshl,lshr,lushr
- 8)按位或指令:ior,lor
- 9)按位与指令:iand,land
- 10)按位异或指令:ixor,lxor
- 11)局部变量自增指令:iinc
- 12)比较指令:dcmpl,dcml,fcml,fcml,lcmpl,lcml
- Java虚拟机没有明确规定整型数据溢出的情况，但规定了处理整型数据时，只有除法和求余指令出现除数为0时会导致虚拟机抛出异常。
- Java虚拟机要求在浮点数运算的时候，所有结果必须舍入到适当的精度，如果有两种可表示的形式与该值一样，会优先选择最低有效位为零的。称之为最接近数舍入模式。
- 浮点数向整数转换的时候，Java虚拟机使用IEEE 754标准中的向零舍入模式，这种模式舍入的结果会导致数字被截断，所有小数部分的有效字节会被丢掉。

类型转换指令

- 类型转换指令将两种Java虚拟机数值类型相互转换，这些操作一般用于实现用户代码的显式类型转换操作。
- JVM直接就支持宽化类型转换(小范围类型向大范围类型转换):
 - 1)int类型到long,float,double类型
 - 2)long类型到float,double类型
 - 3)float到double类型
- 但在处理窄化类型转换时，必须显式使用转换指令来完成，这些指令包括：i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l和 d2f。
- 将int 或 long 窄化为整型T的时候，仅仅简单的把除了低位的N个字节以外的内容丢弃，N是T的长度。这有可能导致转换结果与输入值有不同的正负号。
- 在将一个浮点值窄化为整数类型T（仅限于 int 和 long 类型），将遵循以下转换规则：
 - 1) 如果浮点值是NaN，转换结果就是int 或 long 类型的0
 - 2) 如果浮点值不是无穷大，浮点值使用IEEE 754 的向零舍入模式取整，获得整数v，如果v在T表示范围之内，那就过就是v
 - 3) 否则，根据v的符号，转换为T所能表示的最大或者最小正数

对象创建于访问指令

- 虽然类实例和数组都是对象，Java虚拟机对类实例和数组的创建与操作使用了不同的字节码指令。
- 1)创建实例的指令:new
- 2)创建数组的指令:newarray,anewarray,multianewarray
- 3)访问字段指令:getfield,putfield,getstatic,putstatic
- 4)把数组元素加载到操作数栈指令:baload,caload,saload,iaload,laload,faload,daload,aaload
- 5)将操作数栈的数值存储到数组元素中执行:bastore,castore,castore,sastore,iastore,fastore,dastore,aastore
- 6)取数组长度指令:arraylength JVM支持方法级同步和方法内部一段指令序列同步，这两种都是通过monitor实现的。
- 7)检查实例类型指令:instanceof,checkcast

操作数栈管理指令

- 如同操作一个普通数据结构中的堆栈那样，Java 虚拟机提供了一些用于直接操作操作数栈的指令，包括：
 - 1) 将操作数栈的栈顶一个或两个元素出栈：pop、pop2
 - 2) 复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、dup_x2、dup2_x2。
 - 3) 将栈最顶端的两个数值互换：swap

控制转移指令

- 让JVM有条件或无条件从指定指令而不是控制转移指令的下一条指令继续执行程序。控制转移指令包括：
 - 1)条件分支:ifeq,iflt,ifle,ifne,ifgt,ifge,ifnull,ifnotnull,if_cmpeq,if_icmpne,if_icmlt,if_icmpgt等
 - 2)复合条件分支:tableswitch,lookupswitch
 - 3)无条件分支:goto,goto_w,jsr,jsr_w,ret
- JVM中有专门的指令集处理int和reference类型的条件分支比较操作，为了可以无明显标示一个实体值是否是null,有专门的指令检测null 值。boolean类型和byte类型,char类型和short类型的条件分支比较操作，都使用int类型的比较指令完成，而 long,float,double条件分支比较操作，由相应类型的比较运算指令，运算指令会返回一个整型值到操作数栈中，随后再执行int类型的条件比较操作完成整个分支跳转。各种类型的比较都最终会转化为int类型的比较操作

方法调用和返回指令

- invokevirtual指令:调用对象的实例方法，根据对象的实际类型进行分派(虚拟机分派)。
- invokeinterface指令:调用接口方法，在运行时搜索一个实现这个接口方法的对象，找出合适的方法进行调用。
- invokespecial:调用需要特殊处理的实例方法，包括实例初始化方法，私有方法和父类方法
- invokestatic:调用类方法(static)
- 方法返回指令是根据返回值的类型区分的，包括ireturn(返回值是boolean,byte,char,short和 int),lreturn,freturn,dreturn和areturn，另外一个return供void方法，实例初始化方法，类和接口的类初始化i方法使用。

异常处理指令

- 在Java程序中显式抛出异常的操作（throw语句）都有athrow 指令来实现，除了用throw 语句显示抛出异常情况外，Java虚拟机规范还规定了许多运行时异常会在其他Java虚拟机指令检测到异常状况时自动抛出。
- 在Java虚拟机中，处理异常不是由字节码指令来实现的，而是采用异常表来完成的。

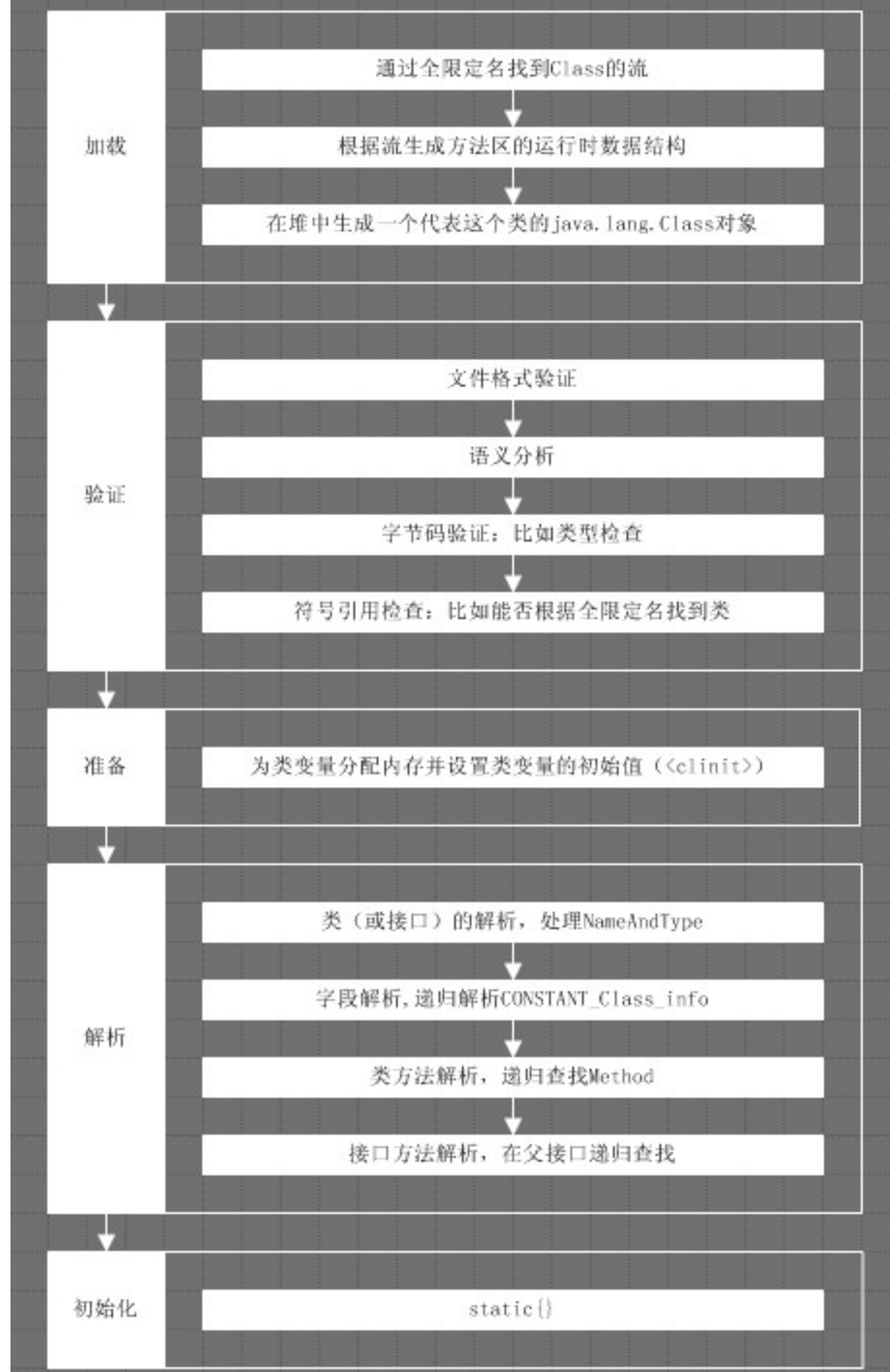
同步指令

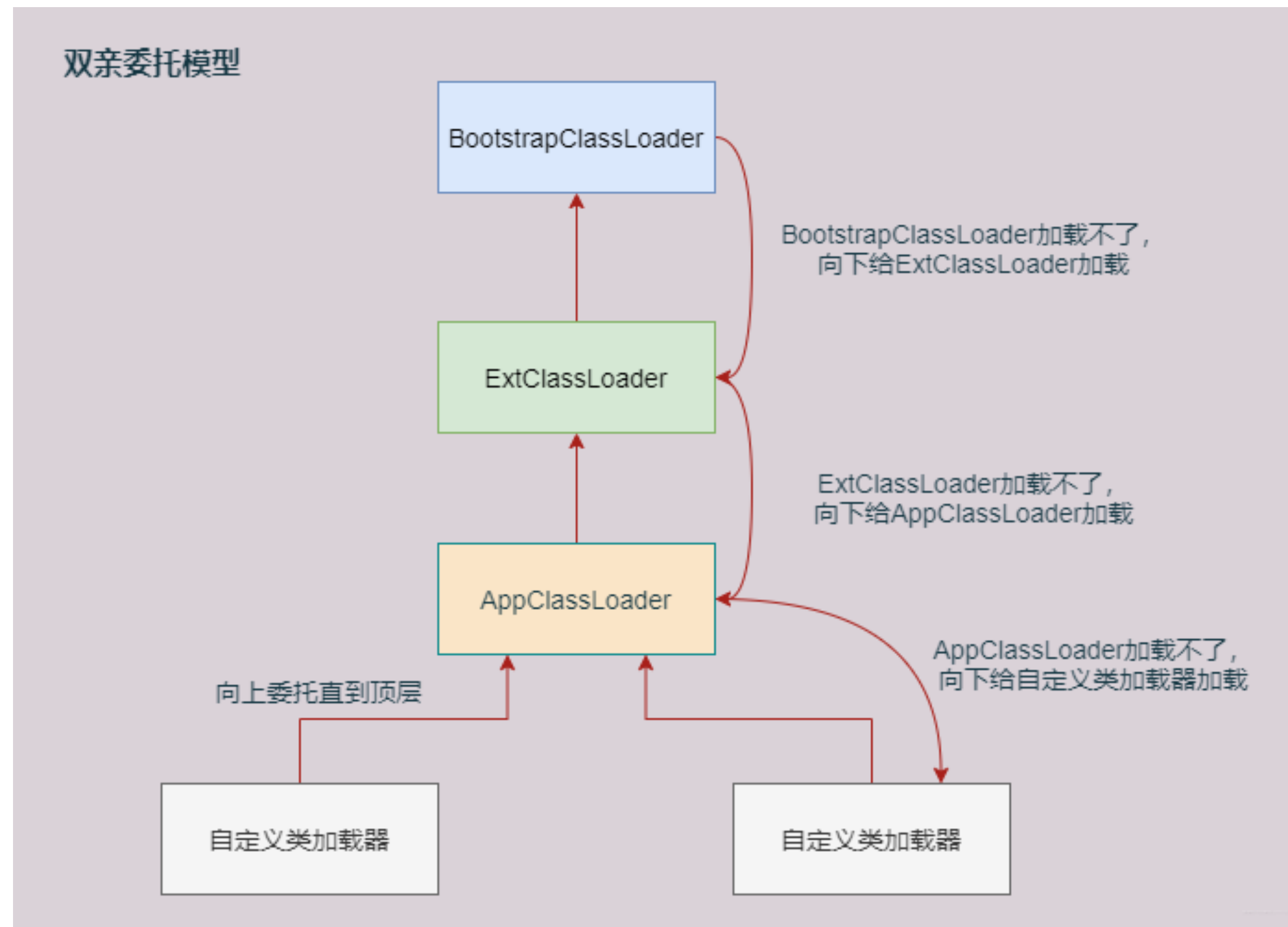
- 方法级的同步是隐式的，无需通过字节码指令来控制，它实现在方法调用和返回操作中。虚拟机从方法常量池中的方法标结构中的 ACC_SYNCHRONIZED 标志区分是否是同步方法。方法调用时，调用指令会检查该标志是否被设置，若设置，执行线程持有 monitor，然后执行方法，最后完成方法时释放 monitor。
- 同步一段指令集序列，通常由 synchronized 块标示，JVM 指令集中有 monitoreenter 和 monitorexit 来支持 synchronized 语义。
- 结构化锁定是指方法调用期间每一个 monitor 退出都与前面 monitor 进入相匹配的情形。JVM 通过以下两条规则来保证结构化锁成立 (T 代表一线程，M 代表一个 monitor)：
 - 1) T 在方法执行时持有 M 的次数必须与 T 在方法完成时释放的 M 次数相等
 - 2) 任何时刻都不会出现 T 释放 M 的次数比 T 持有 M 的次数多的情况

5 虚拟机字节码执行

类的生命周期

- 加载
- 连接
 - 验证
 - 准备
 - 解析
- 初始化
- 使用
- 卸载





- Bootstrap ClassLoader ，主要负责加载Java核心类库，`%JRE_HOME%\lib`下的`rt.jar`、`resources.jar`、`charsets.jar`和`class`等。
- Extention ClassLoader，主要负责加载目录`%JRE_HOME%\lib\ext`目录下的`jar`包和`class`文件。
- Application ClassLoader ，主要负责加载当前应用的`classpath`下的所有类
- User ClassLoader ， 用户自定义的类加载器,可加载指定路径的`class`文件

类加载

双亲委派机制，指的就是：当一个类加载器收到了类加载的请求的时候，他不会直接去加载指定的类，而是把这个请求委托给自己的父加载器去加载。只有父加载器无法加载这个类的时候，才会由当前这个加载器来负责类的加载。

```
int foo(int a, int b, int c) {  
    return a + b * c;  
}  
//...  
foo(2, 3, 4);
```

输入：语言A的源代码

(由编译器编译)

```
mov    eax, dword ptr [esp + 8]  
imul   eax, dword ptr [esp + 0C]  
add    eax, dword ptr [esp + 4]  
ret  
;...  
push   4  
push   3  
push   2  
call   foo  
add    esp, 0C
```

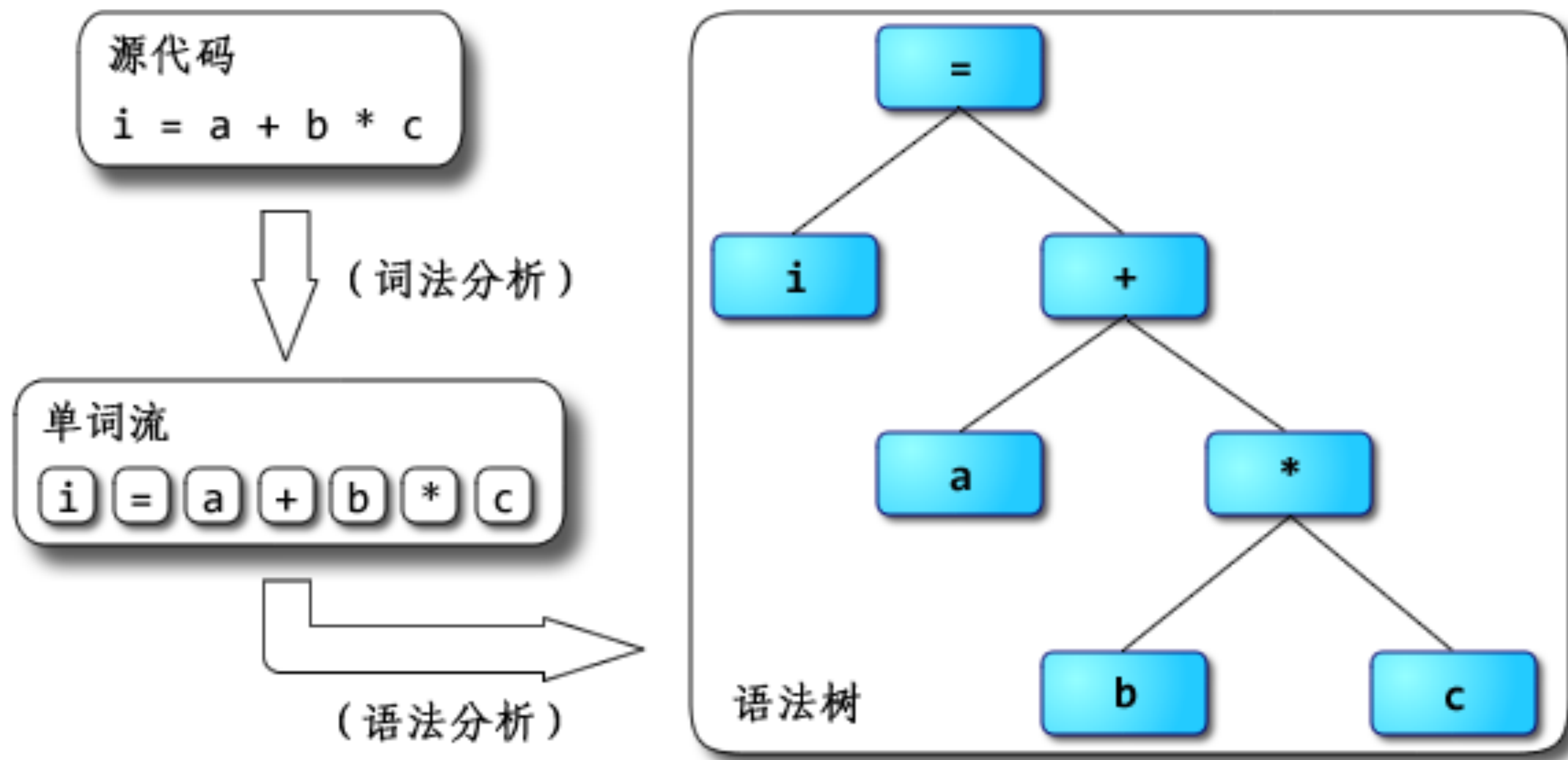
编译输出：与源程序等价的语言B源码

(由解释器解释执行)

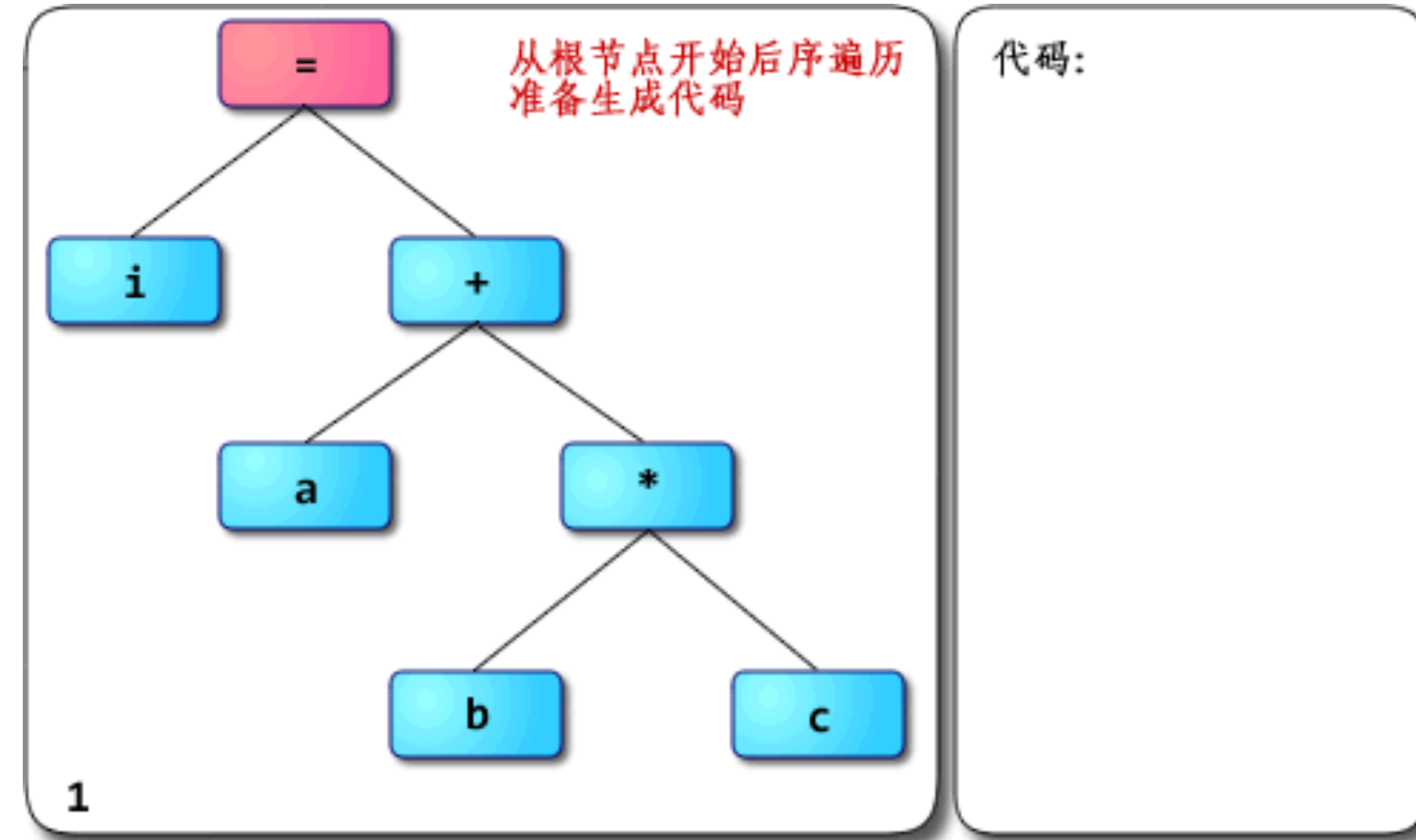
14

解释输出：程序运行结果

编译与解释



广义解析器



生成字节码

示例代码 1

- `public class EvalOrderDemo {`
- `public static void main(String[] args) {`
- `int[] arr = new int[1];`
- `int a = 1;`
- `int b = 2;`
- `arr[0] = a + b;`
- `}`
- `}`

Bytecode

- // 左子树: 数组下标
 - // a[0]
 - aload_1
 - iconst_0
-
- // 右子树: 加法
 - // a

- iload_2
- // b
- iload_3
- // +
- iadd

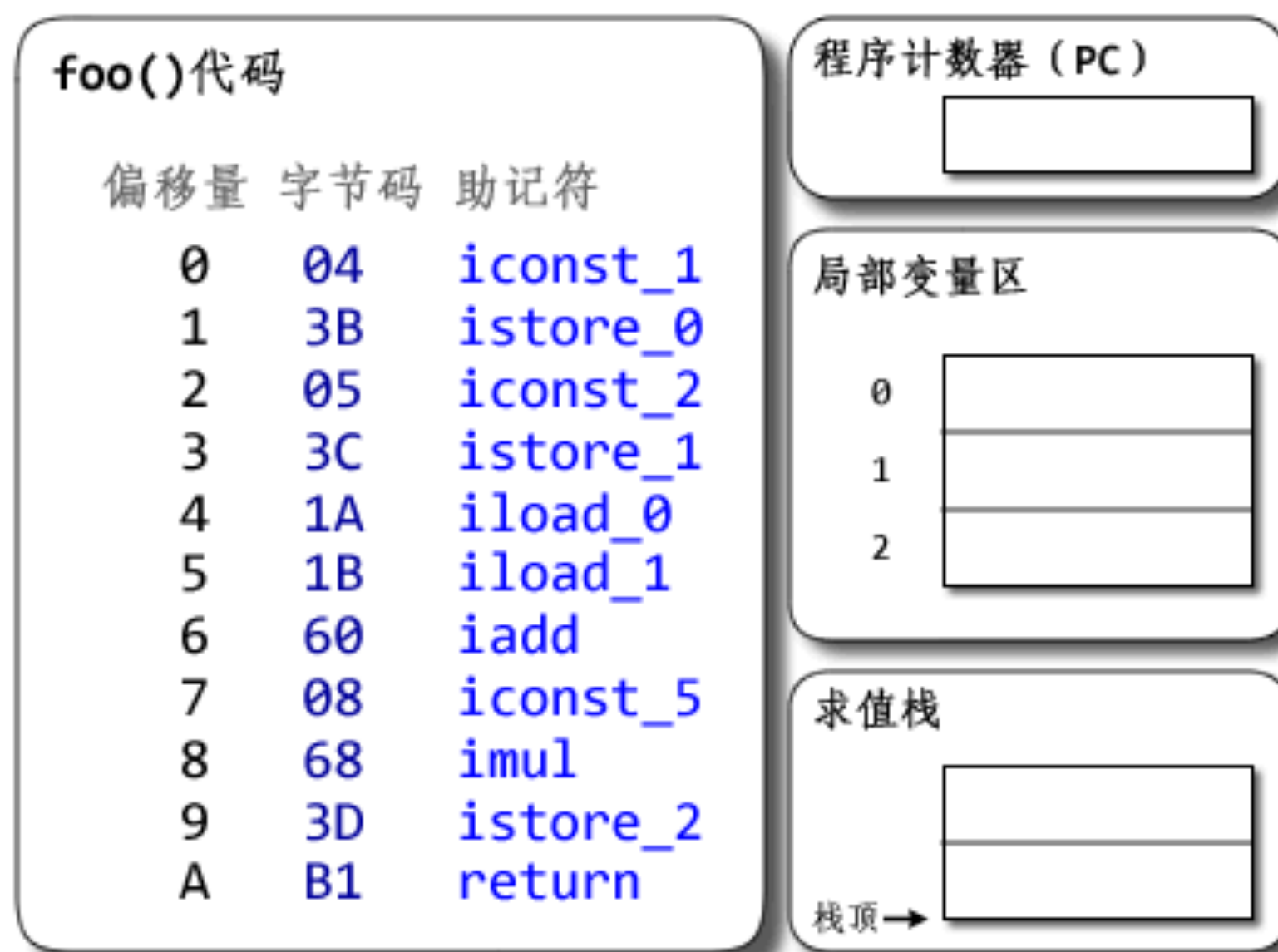
- // 根节点: 赋值
- iastore

示例代码 2

- `public class Demo {`
- `public static void foo() {`
- `int a = 1;`
- `int b = 2;`
- `int c = (a + b) * 5;`
- `}`
- `}`

Bytecode

- 0: iconst_1
- 1: istore_0
- 2: iconst_2
- 3: istore_1
- 4: iload_0
- 5: iload_1
- 6: iadd
- 7: iconst_5
- 8: imul
- 9: istore_2
- 10: return

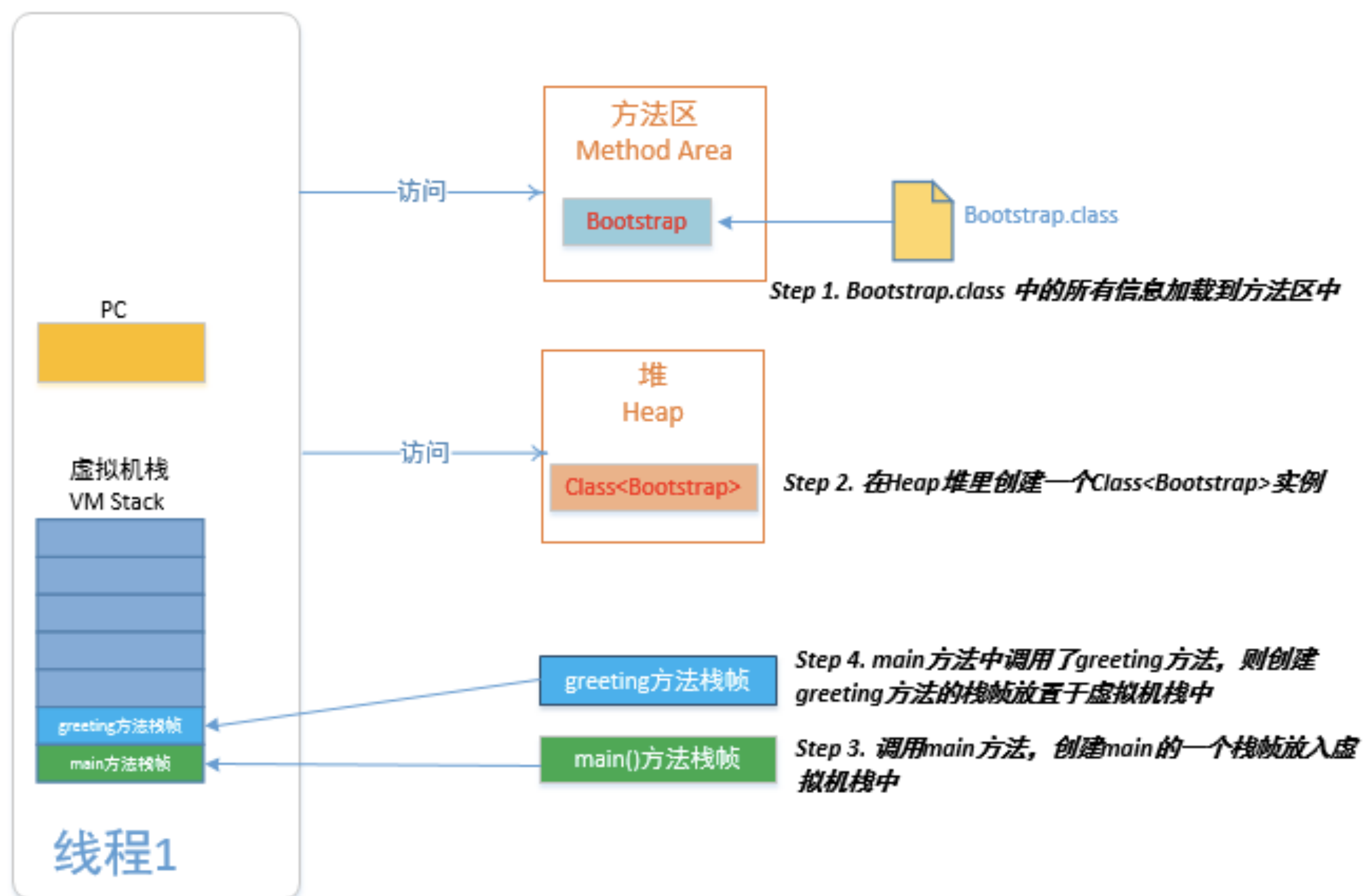


栈架构

Sample Code

```
1. package org.louis.jvm.codeset;
2. /**
3.  * JVM 原理简单用例
4.  * @author louis
5.  *
6.  */
7. public class Bootstrap {
8.
9.     public static void main(String[] args) {
10.         String name = "Louis";
11.         greeting(name);
12.     }
13.
14.     public static void greeting(String name)
15.     {
16.         System.out.println("Hello,"+name);
17.     }
18.
19. }
```

运行时数据区



Designed by LouLuan

<http://blog.csdn.net/luanlouis>

```
public static void main(String[] args) {  
    String name = "Louis";  
    greeting(name);  
}
```

编译后

入参数量 : 1
局部变量数量 : 2
操作数栈大小 : 1
机器指令 : 12 10 4c 2b b8 20 12 b1

```
public static void greeting(String name)  
{  
    System.out.println("Hello,"+name);  
}
```

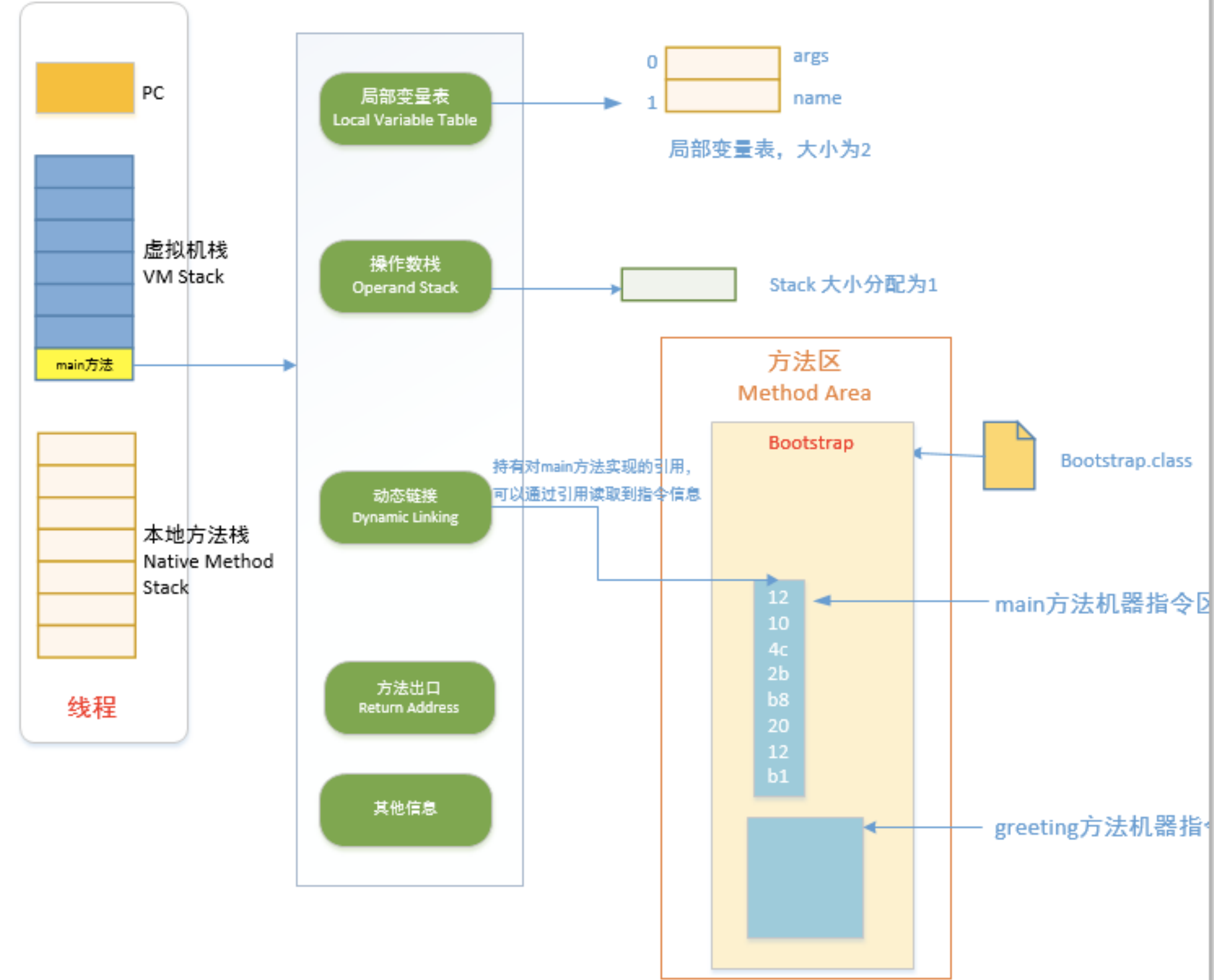
编译后

入参数量 : 1
局部变量数量 : 1
操作数栈大小 : 4
机器指令 : b2 20 1a bb 20 20 59 12 22 b7 20 24 2a b6 20 26 b6

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

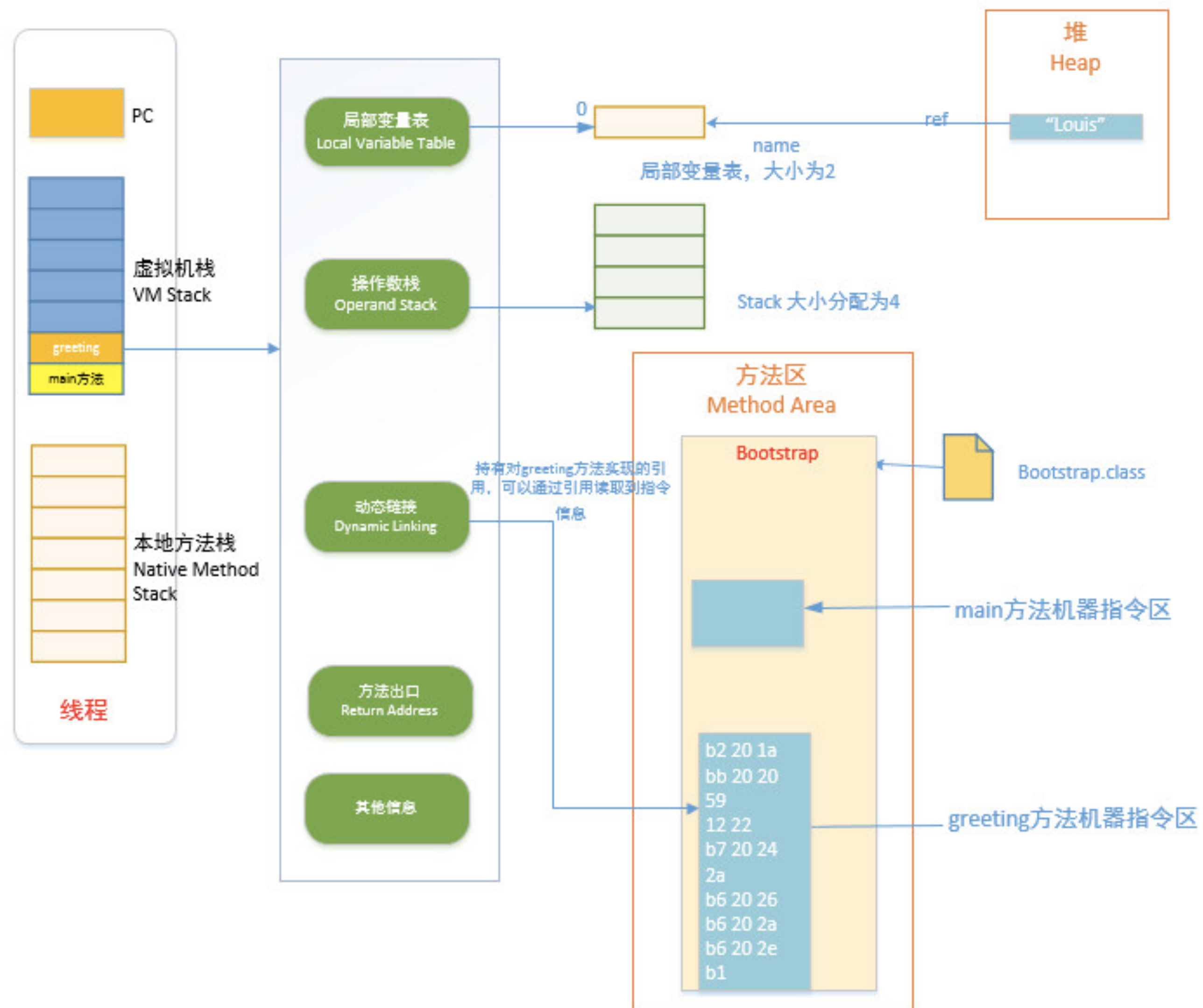
main方法的栈帧信息 Stack Frame



main()

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=2, args_size=1
       0: ldc          #2          // String Louis
       2: astore_1
       3: aload_1
       4: invokestatic #3          // Method greeting:(Ljava/lang/String;)V
       7: return
 LineNumberTable:
    line 10: 0
    line 11: 3
    line 12: 7
```

greeting方法的栈帧信息 Stack Frame



greeting()

```
public static void greeting(java.lang.String);
descriptor: (Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=1, args_size=1
    0: getstatic      #4          // Field java/lang/System.out:Ljava/io/PrintStream;
    3: new            #5          // class java/lang/StringBuilder
    6: dup
    7: invokespecial  #6          // Method java/lang/StringBuilder."<init>":()V
   10: ldc             #7          // String Hello,
   12: invokevirtual #8          // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   15: aload_0
   16: invokevirtual #8          // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   19: invokevirtual #9          // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   22: invokevirtual #10         // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   25: return
LineNumberTable:
  line 16: 0
  line 17: 25
```

机器指令是用来告诉机器做什么的指令。操作码是鉴别是什么指令，而操作数相当于执行相应的指令所需要的参数。



greeting方法机器指令案例

机器指令	操作码	操作数1	操作数2	操作数3	助记符及解释
b2 20 1a	b2	20	1a		getstatic #26
bb 20 20	bb	20	20		new #32
59	59				dup
12 22	12	22			ldc #34
b7 20 24	b7	20	24		invokespecial #36
2a	2a				aload_0
b6 20 26	b6	20	26		invokevirtual #38
b6 20 2a	b6	20	2a		invokevirtual #42
b6 20 2e	b6	20	2e		invokevirtual #46
b1	b1				return

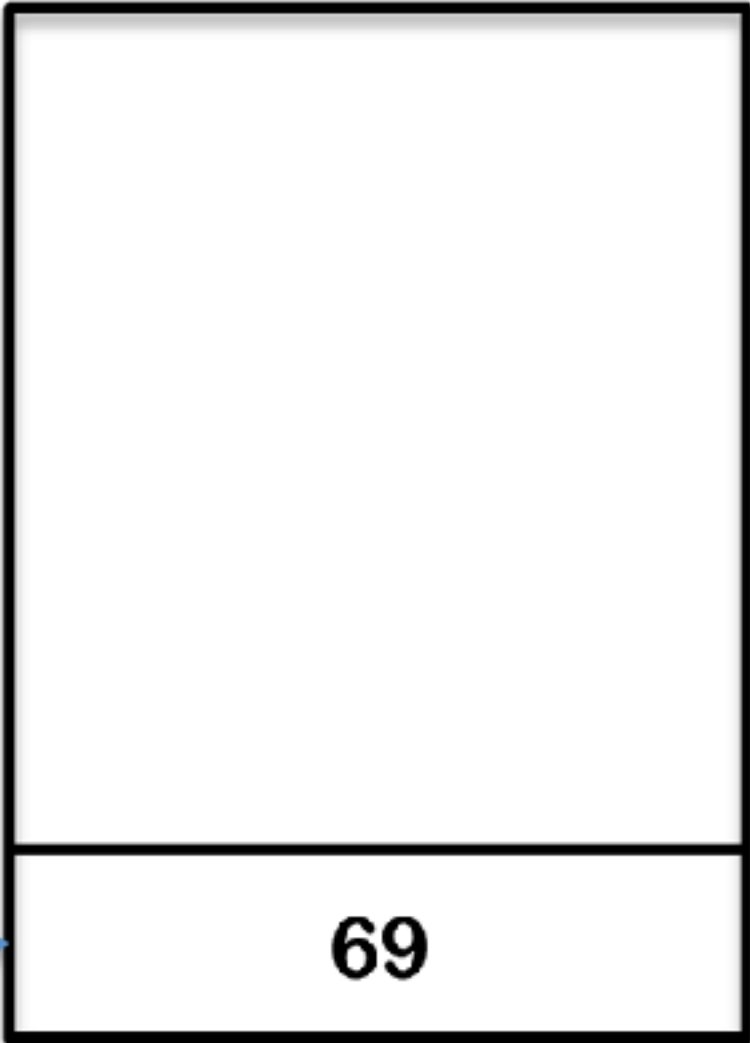
6 Java语句与字节码

变量

```
int i = 69;
```

```
0: bipush 69
```

Operand Stack

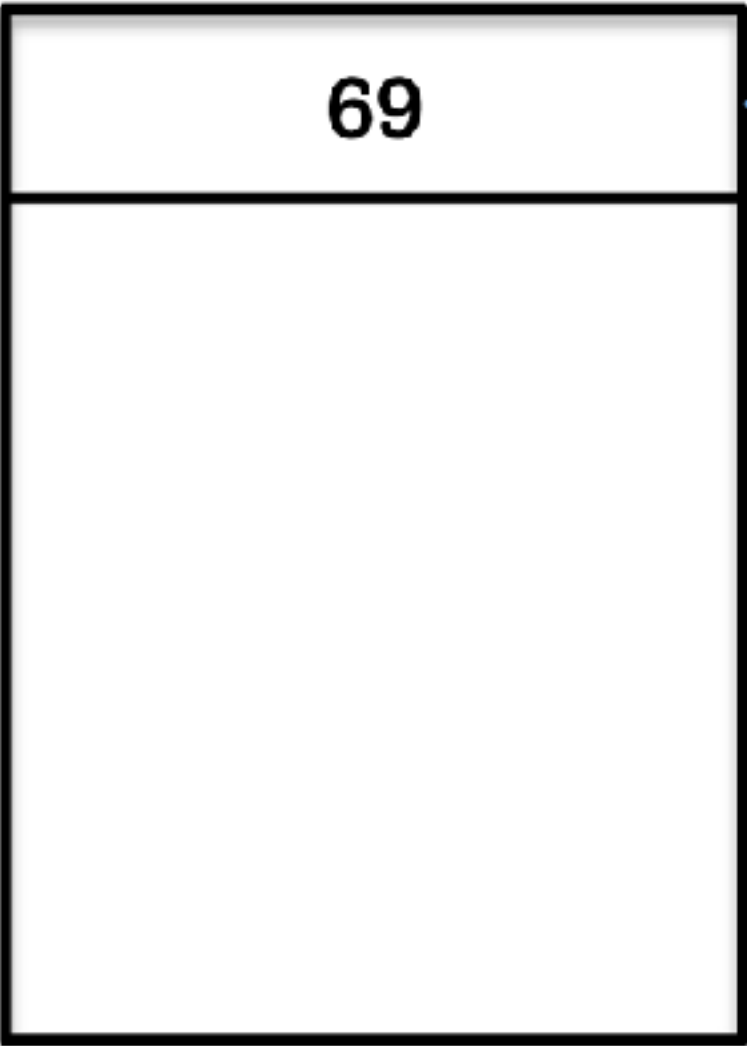


push

69

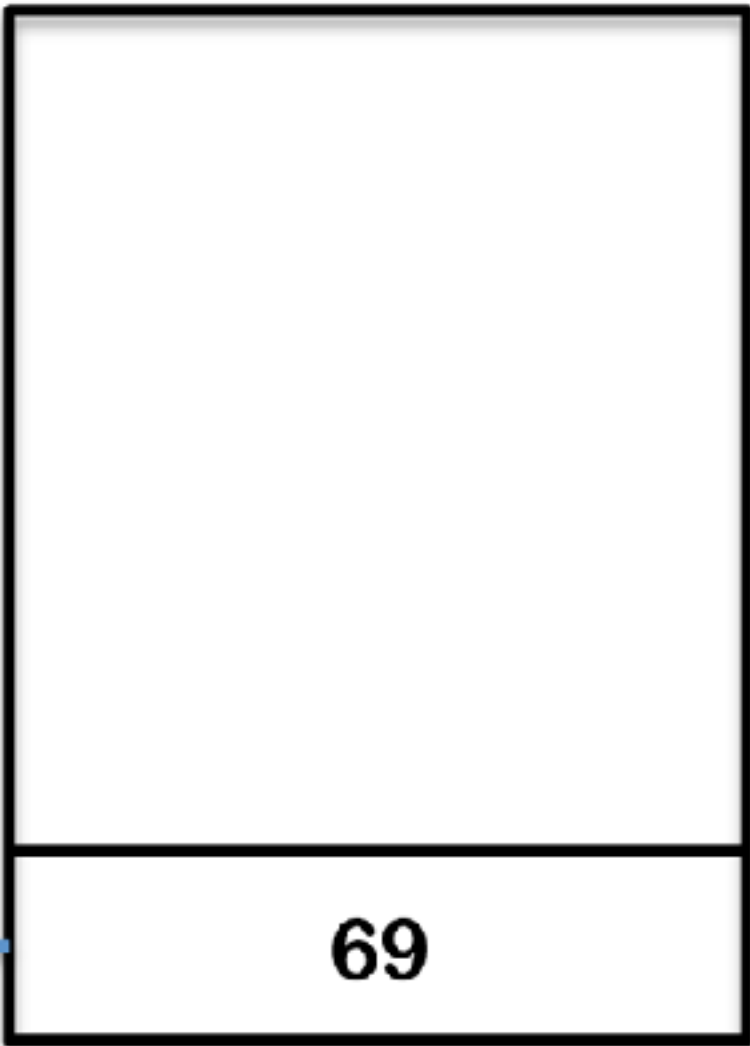
```
2: istore_0
```

Local Variables



69

Operand Stack

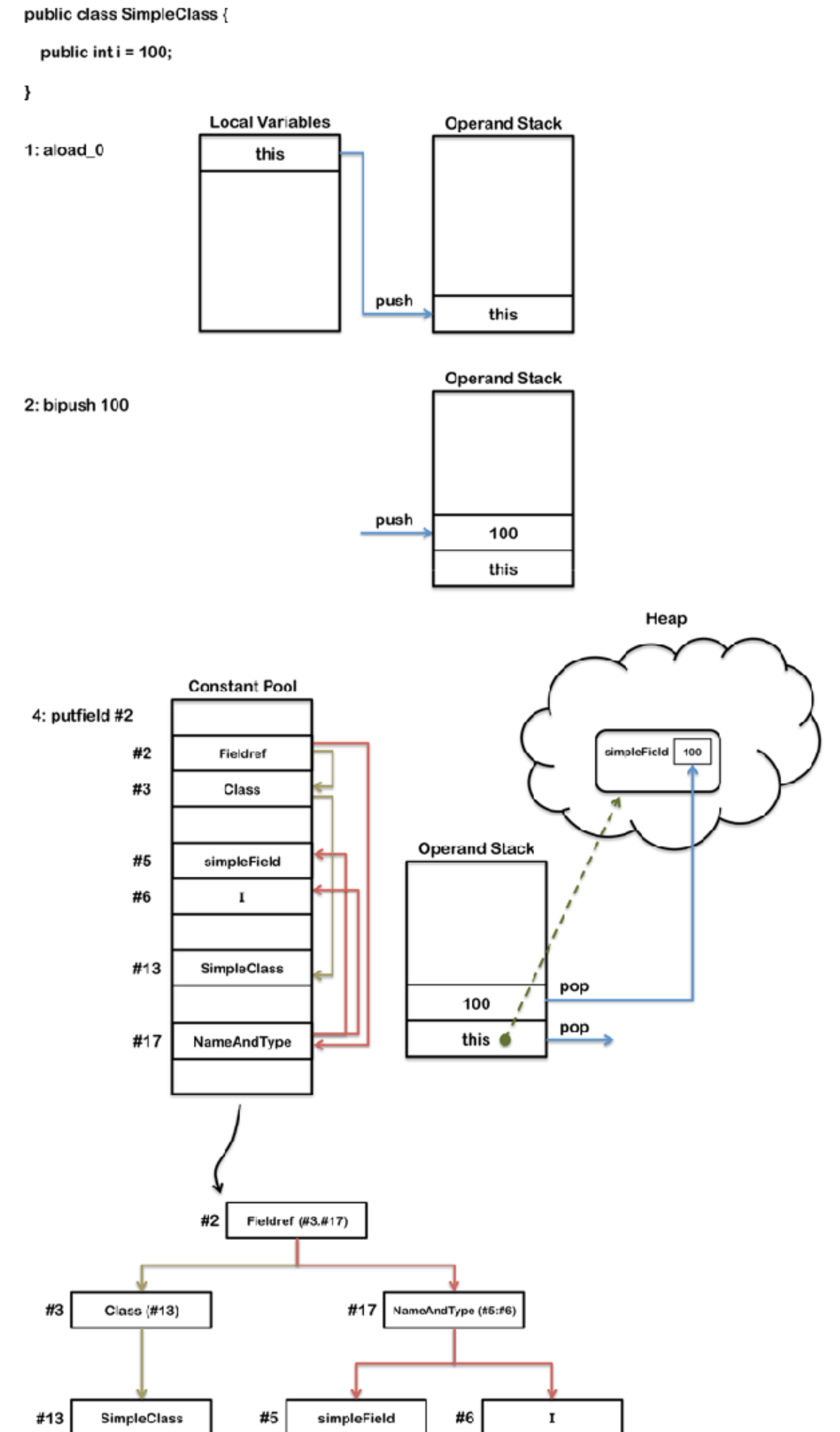


pop

69

局部变量

成员变量

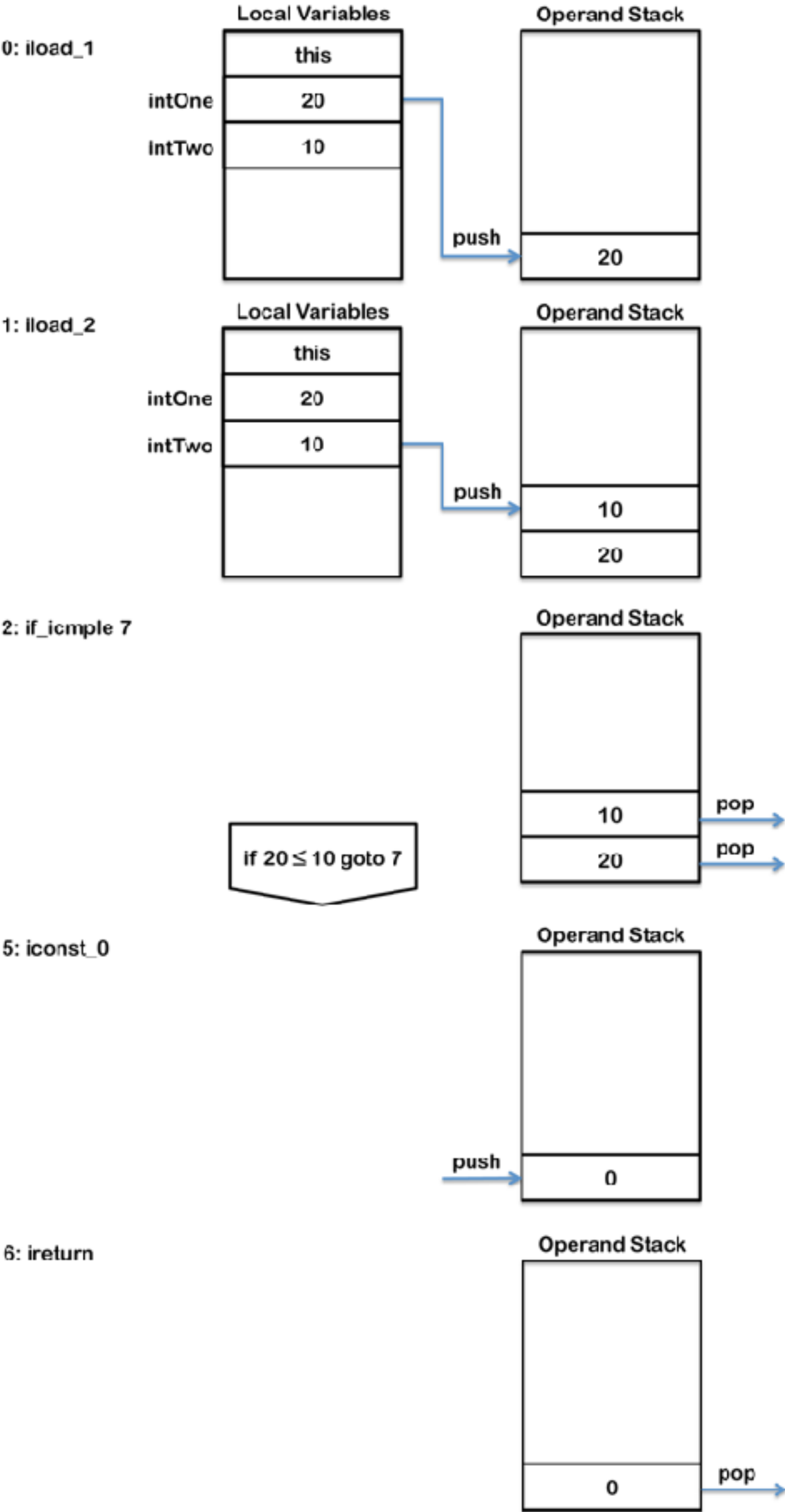


条件语句

if else

```
public int greaterThen(int intOne, int intTwo) {
    if (intOne > intTwo) {
        return 0;
    } else {
        return 1;
    }
}

greaterThen(10, 20);
```

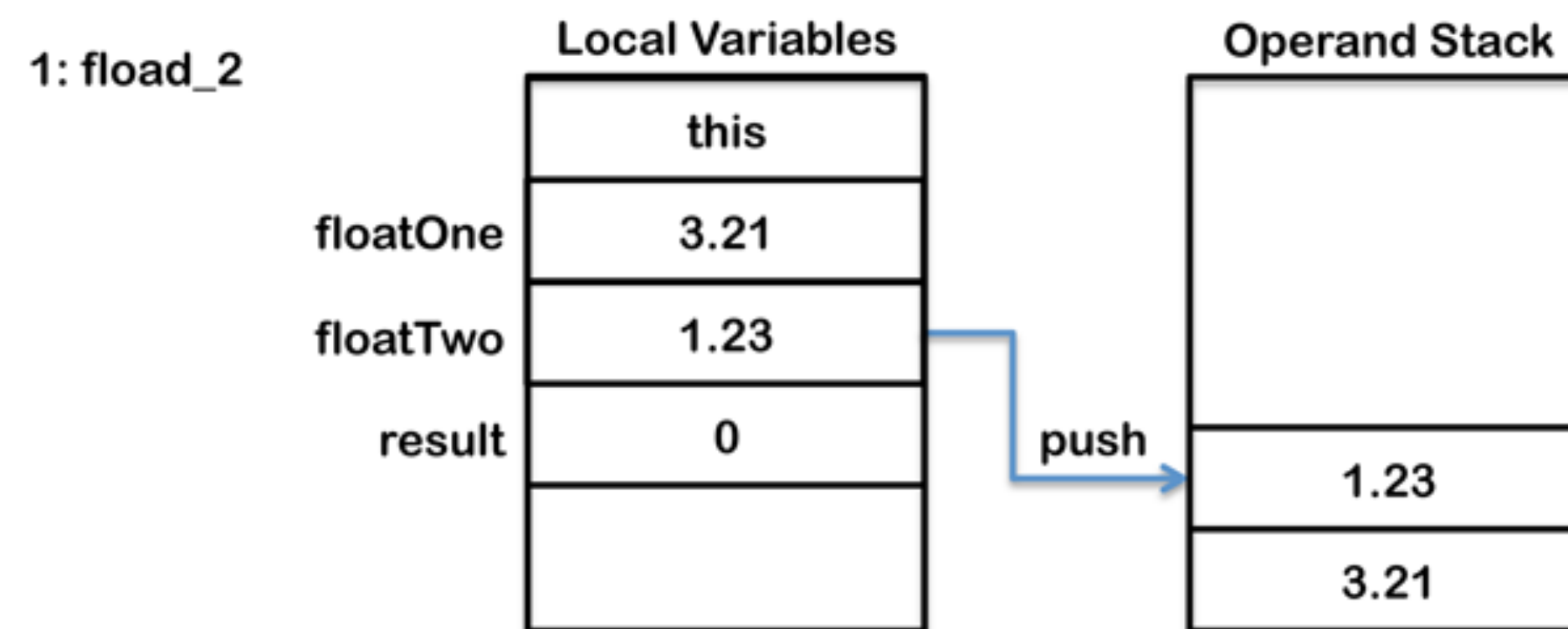
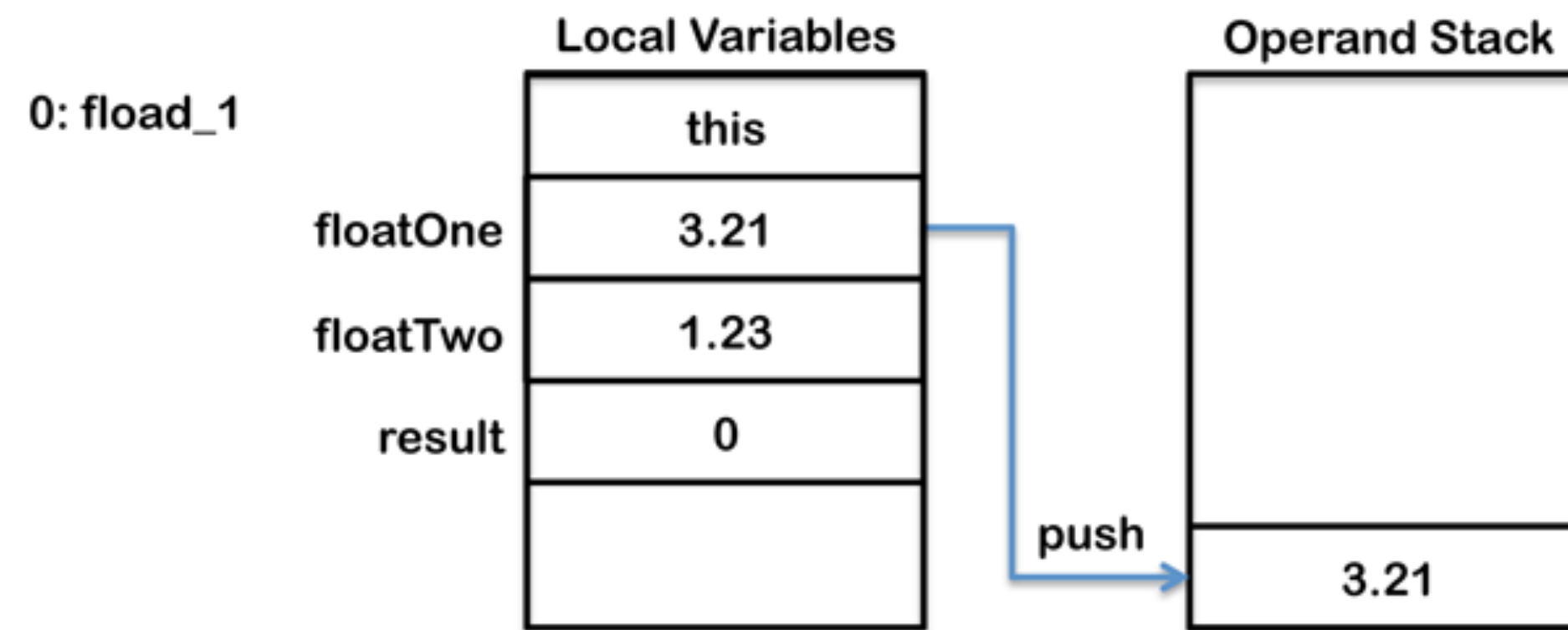


```

public int greaterThen(float floatOne, float floatTwo) {
    int result;
    if (floatOne > floatTwo) {
        result = 1;
    } else {
        result = 2;
    }
    return result;
}

```

greateThen(3.21, 1.23);

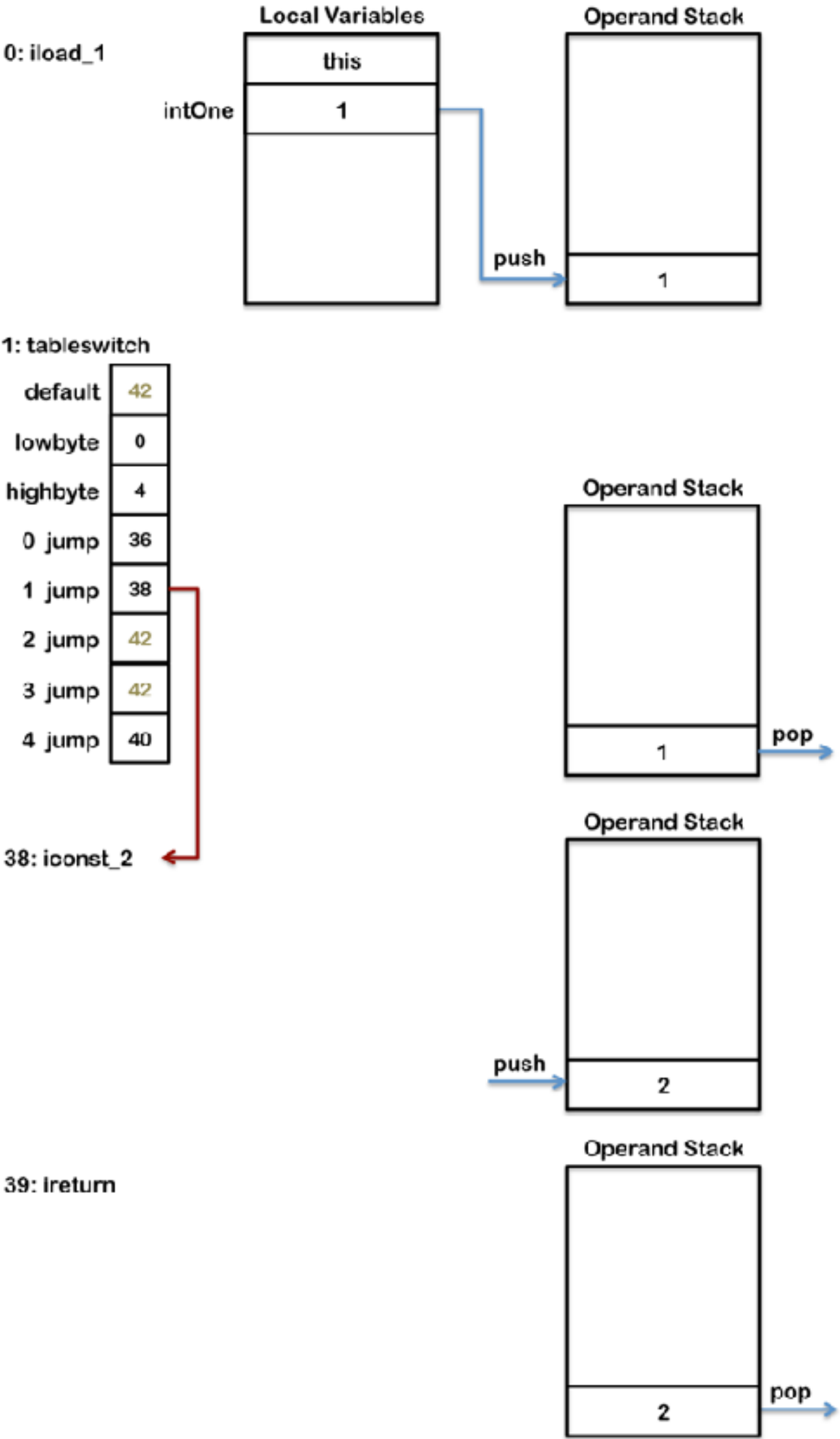


if else

Switch

```
public int simpleSwitch(int intOne) {
    switch (intOne) {
        case 0:
            return 3;
        case 1:
            return 2;
        case 4:
            return 1;
        default:
            return -1;
    }
}
```

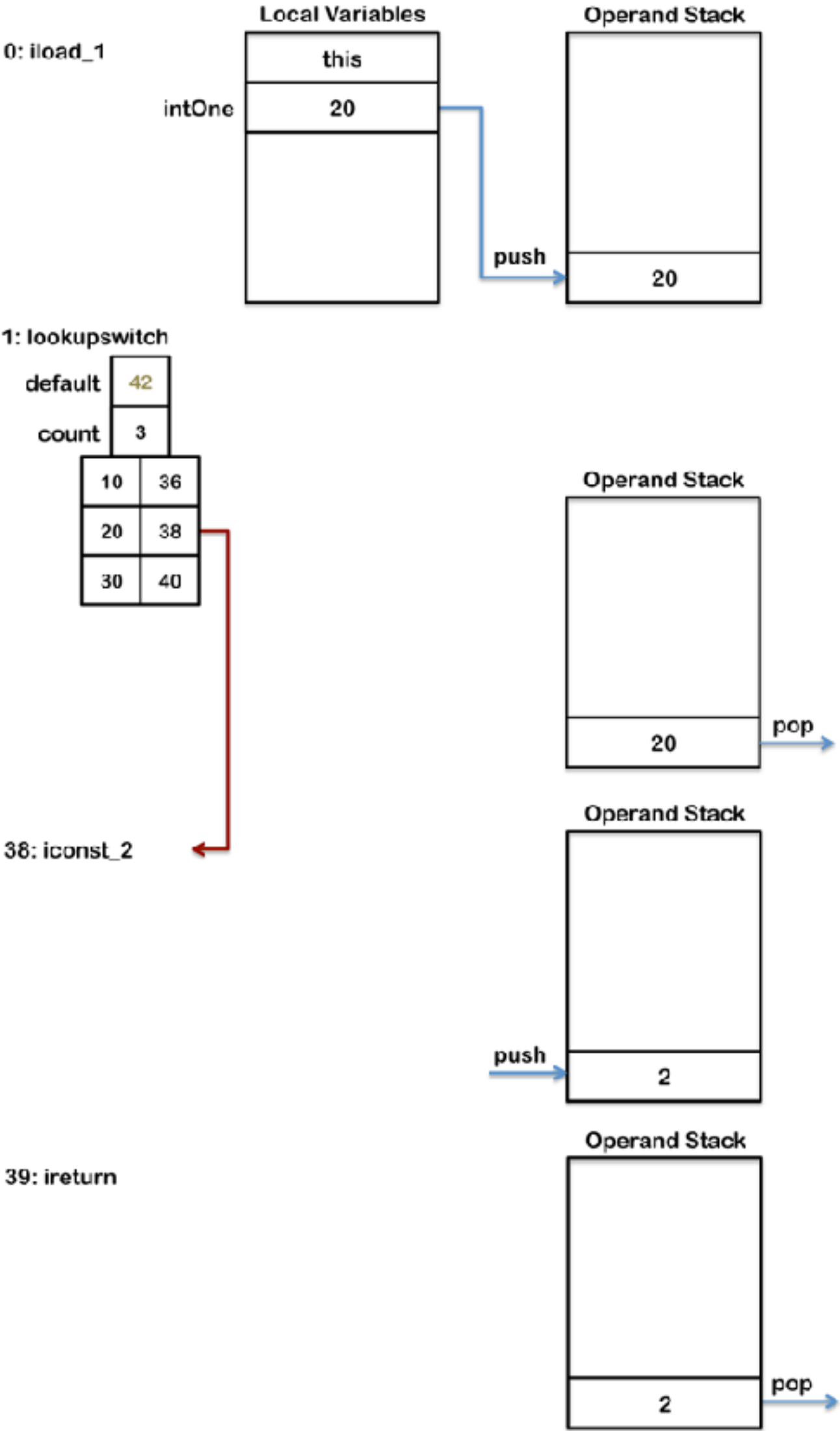
simpleSwitch(1);



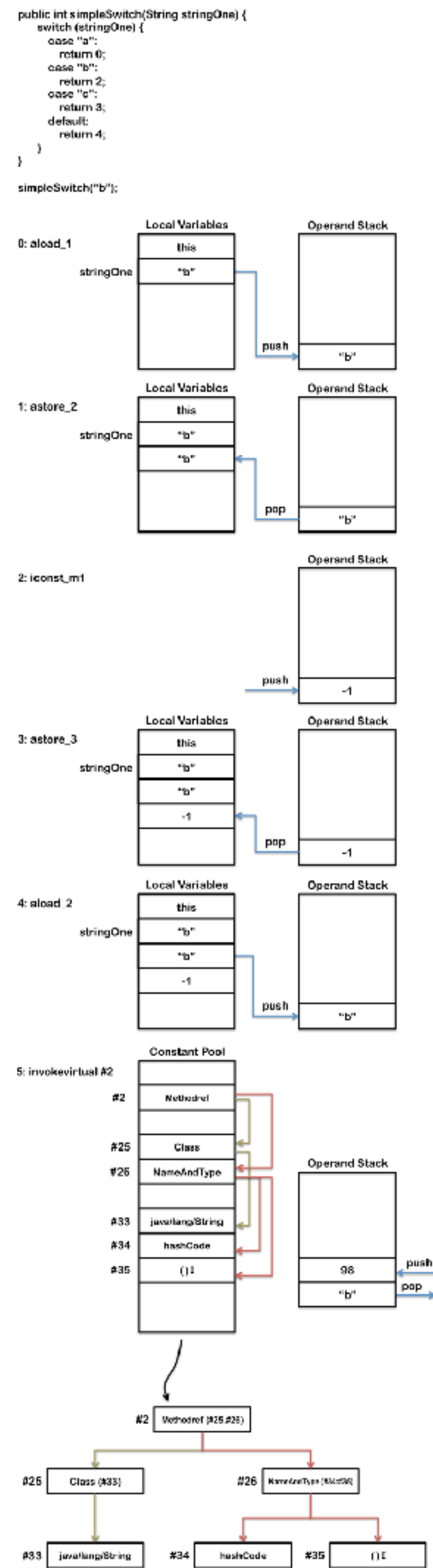
Switch

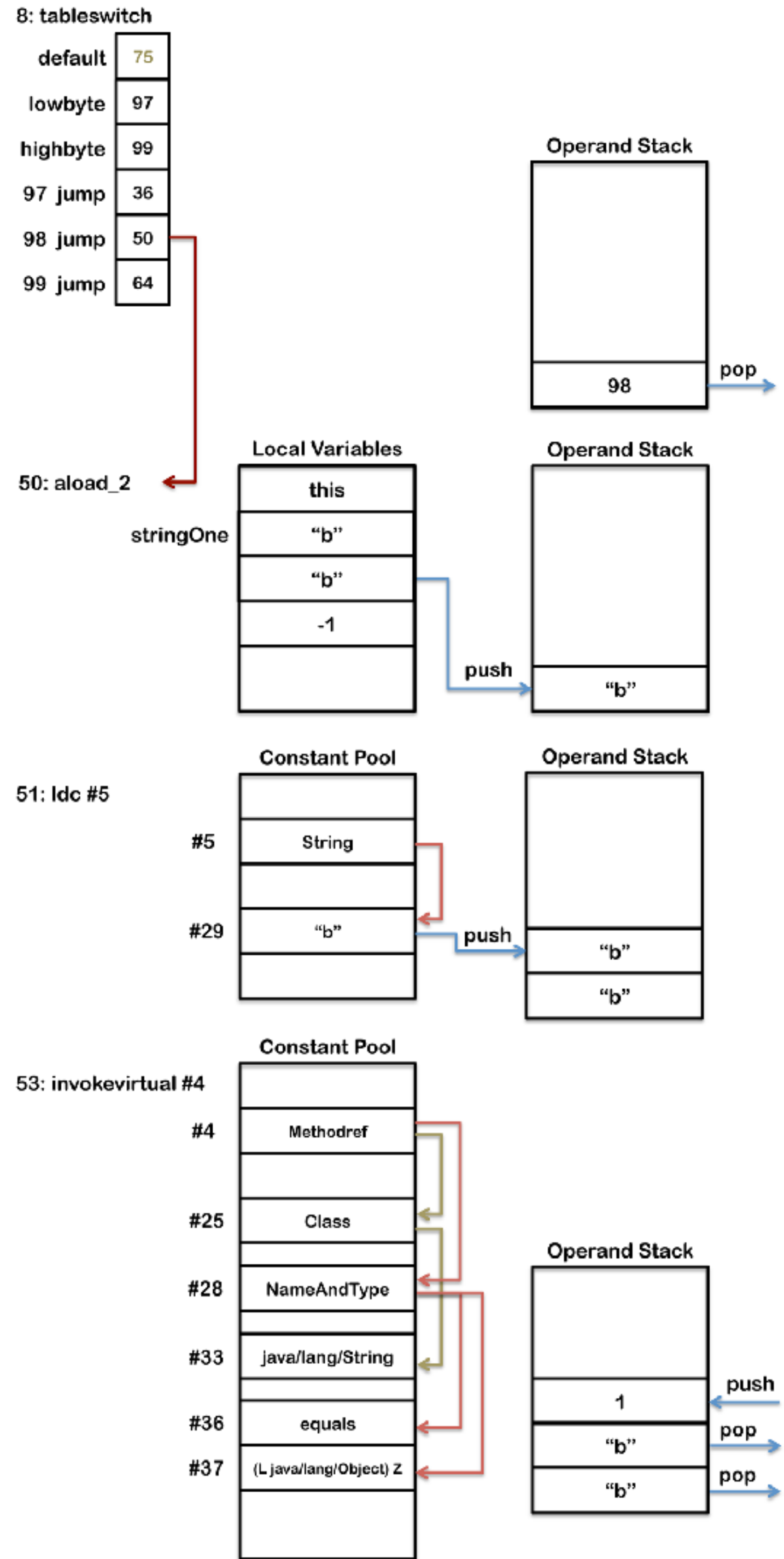
```
public int simpleSwitch(int intOne) {  
    switch (intOne) {  
        case 10:  
            return 1;  
        case 20:  
            return 2;  
        case 30:  
            return 3;  
        default:  
            return -1;  
    }  
}
```

```
simpleSwitch(20);
```



Sting switch





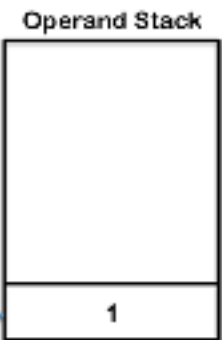
56: ifeq 75

if = 0 goto 75



pop

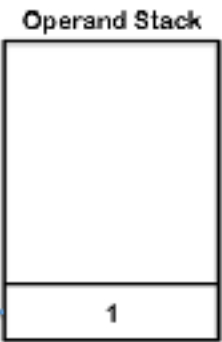
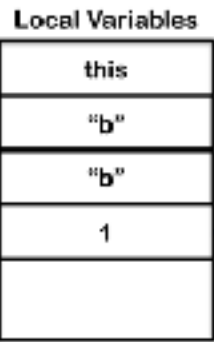
59: iconst 1



push

60: istore 3

stringOne

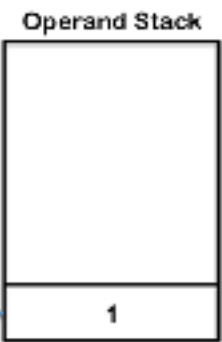
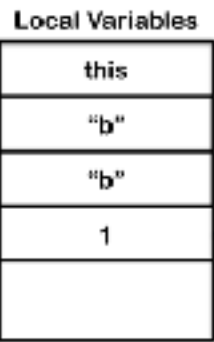


pop

61: goto 75

75: iload 3

stringOne



push

76: tableswitch

default	110
lowbyte	0
highbyte	2
0 jump	104
1 jump	106
2 jump	106



pop

106: iconst 2



push

107: ireturn

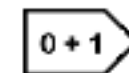


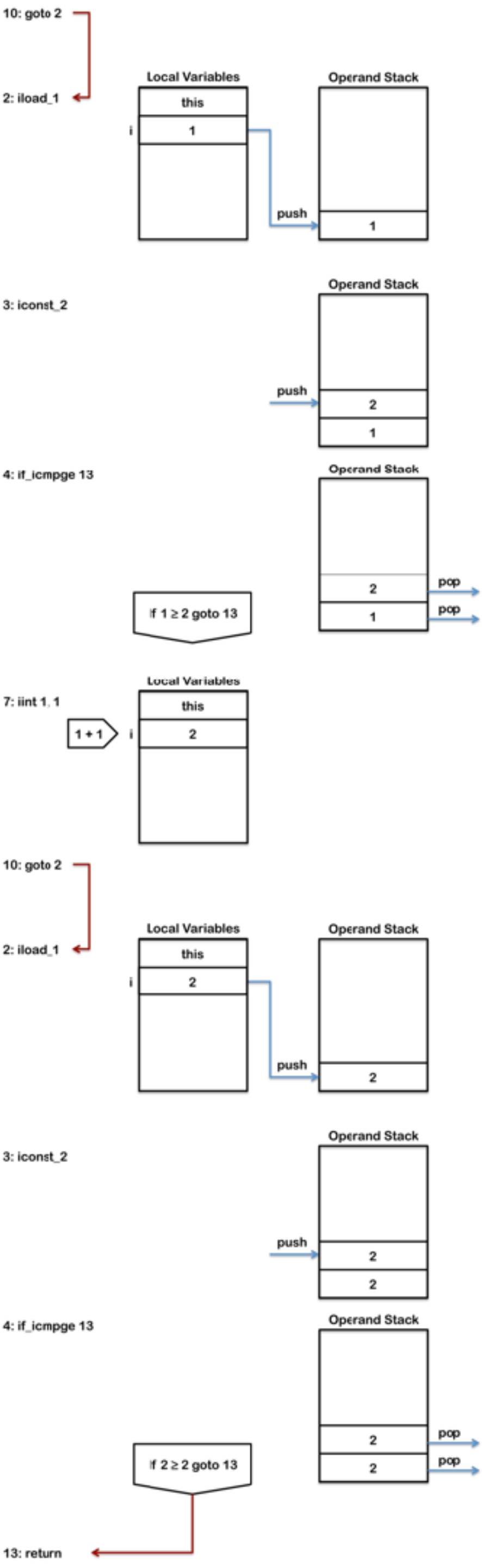
pop

循环语句

while

7: iint 1, 1





do while

