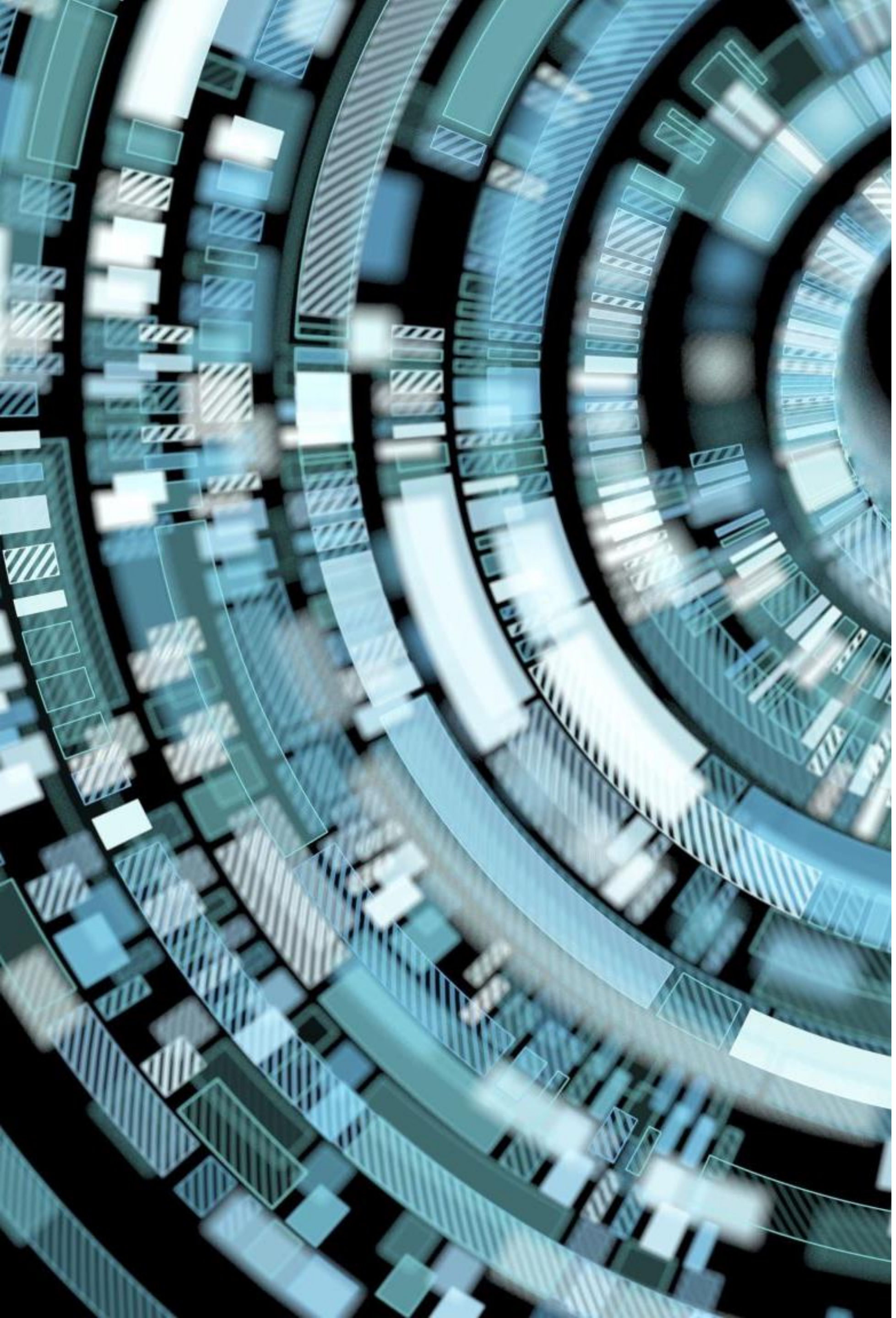
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）

taoyd@sustech.edu.cn



Lecture 14

- A Deep Dive into JVM

TAO Yida@SUSTECH

JVM Architecture

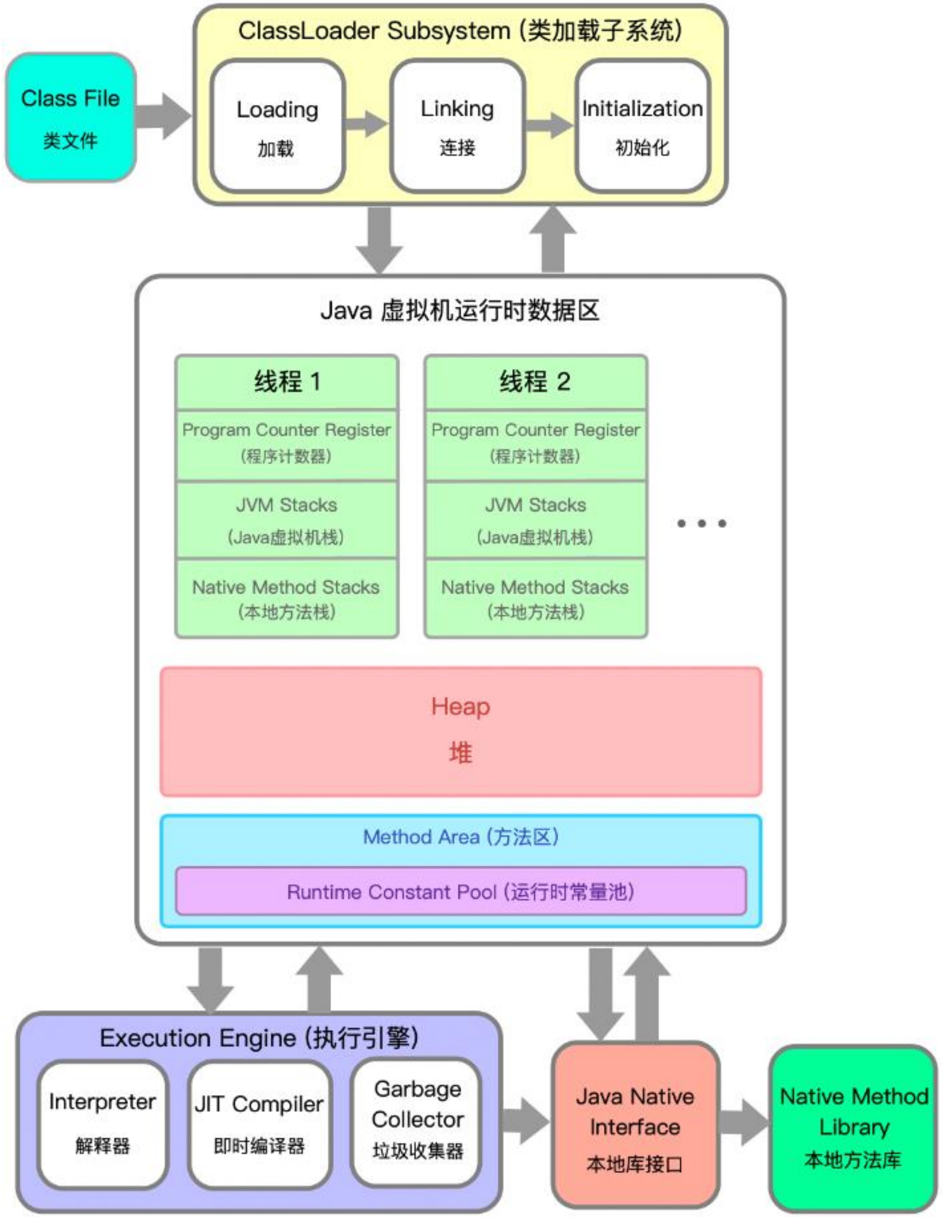
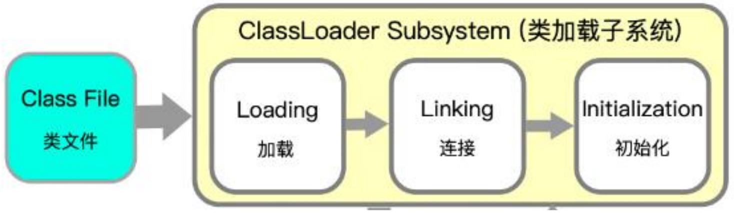
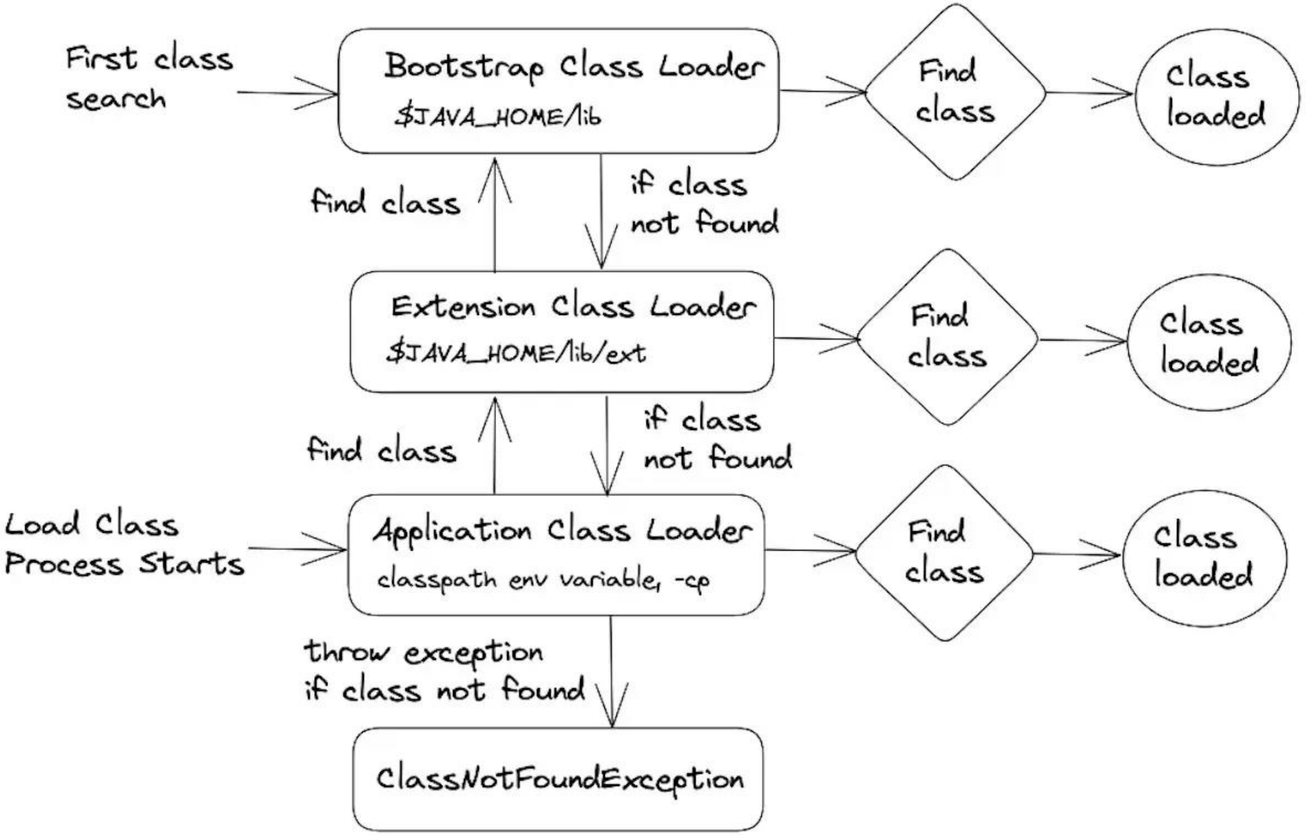


Image source: https://www.cnblogs.com/hynblogs/p/12275957.html

ClassLoader: Loading

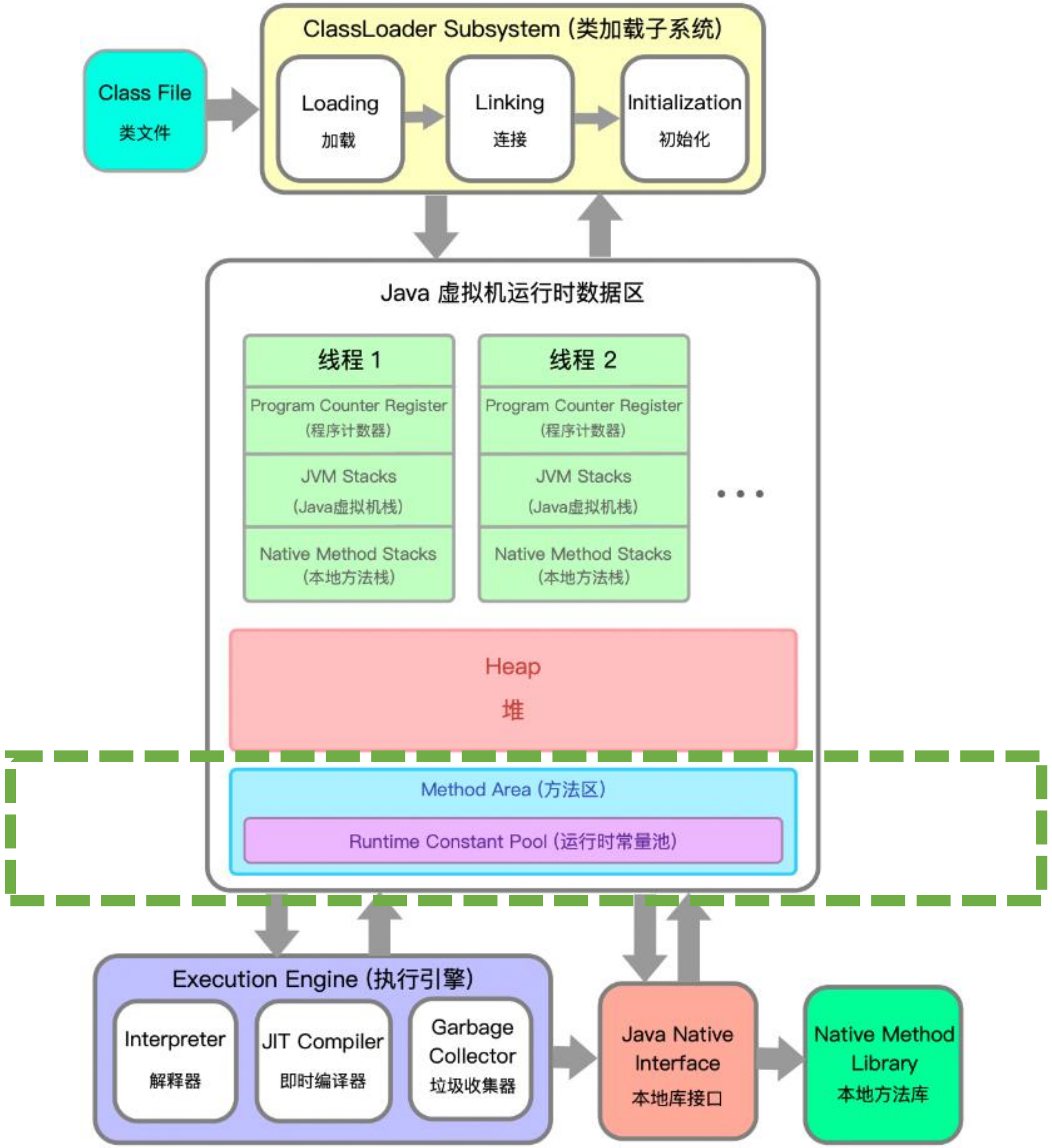
1. JVM retrieves the bytecode by its fully qualified name
2. Store the class structure info in the method area
3. JVM creates a `java.lang.Class` object as the access point for all runtime data about the class stored in the method area.





ClassLoader: Loading

1. JVM retrieves the bytecode by its fully qualified name
2. Store the class structure info in the method area
3. JVM creates a `java.lang.Class` object as the access point for all runtime data about the class stored in the method area.

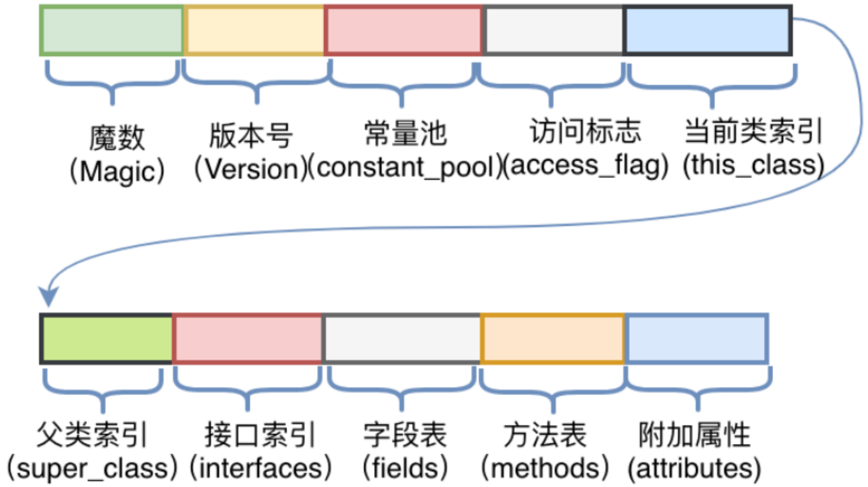


ClassLoader: Loading

1. JVM retrieves the bytecode by its fully qualified name
2. Store the class structure info in the method area
3. JVM creates a `java.lang.Class` object as the access point for all runtime data about the class stored in the method area.

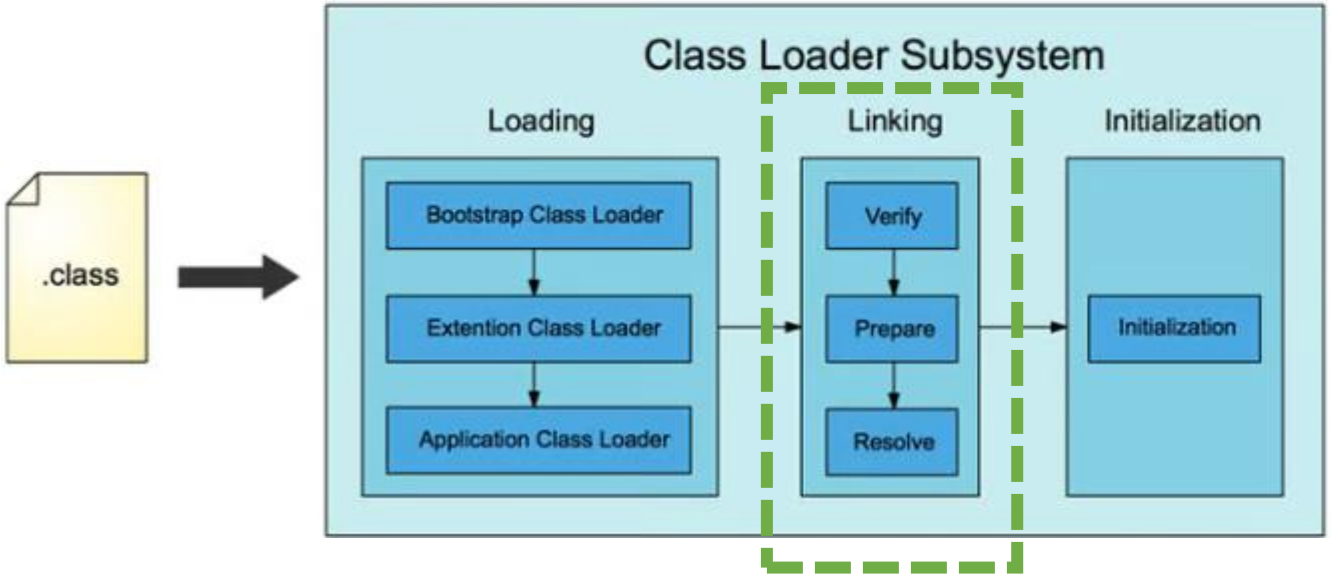
When a class is loaded, JVM stores the following in the method area:

- **Class info**: class name, modifiers, superclass, interfaces, static variables, etc.
- **Field info**: field name, types, modifiers, etc.
- **Methods**: method name, parameters, return type, modifiers, bytecode, etc.
- **Constant pool**: string literals, final constants, symbolic references like `println`.



ClassLoader: Linking

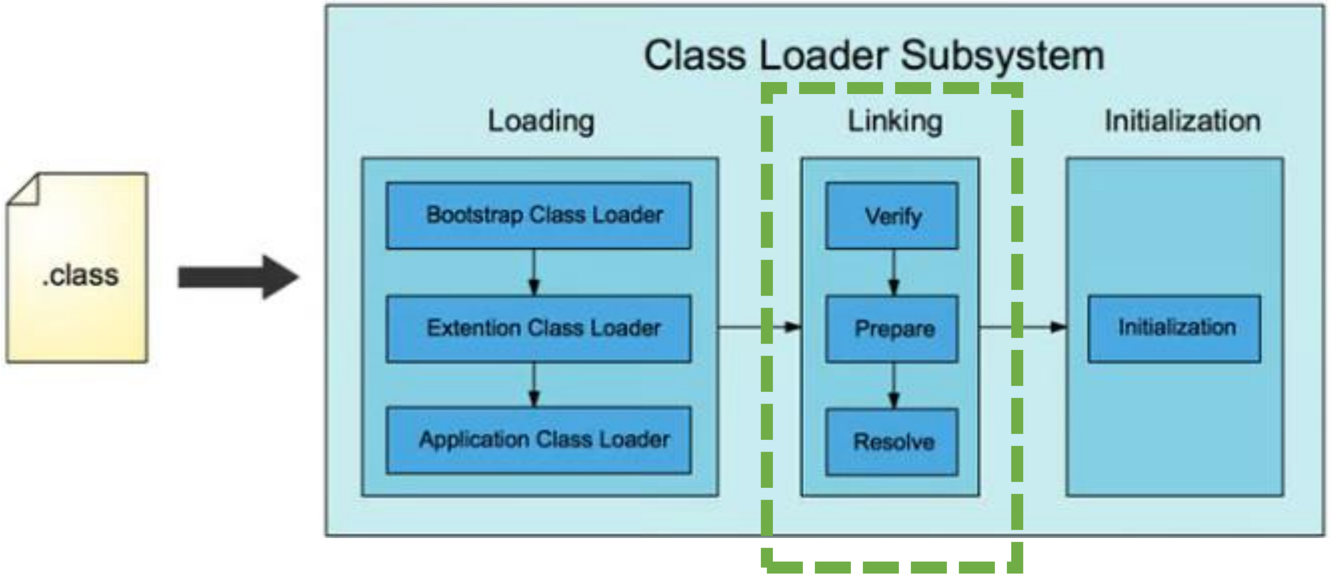
1. Verify: verify that the bytecode follows Java Specification
2. Preparation: allocate memory for static variables



ClassLoader: Linking

3. Resolve: resolve symbolic references in the constant pool to direct references

- To find the implementation of a class or method name
- Apply to private methods and static methods

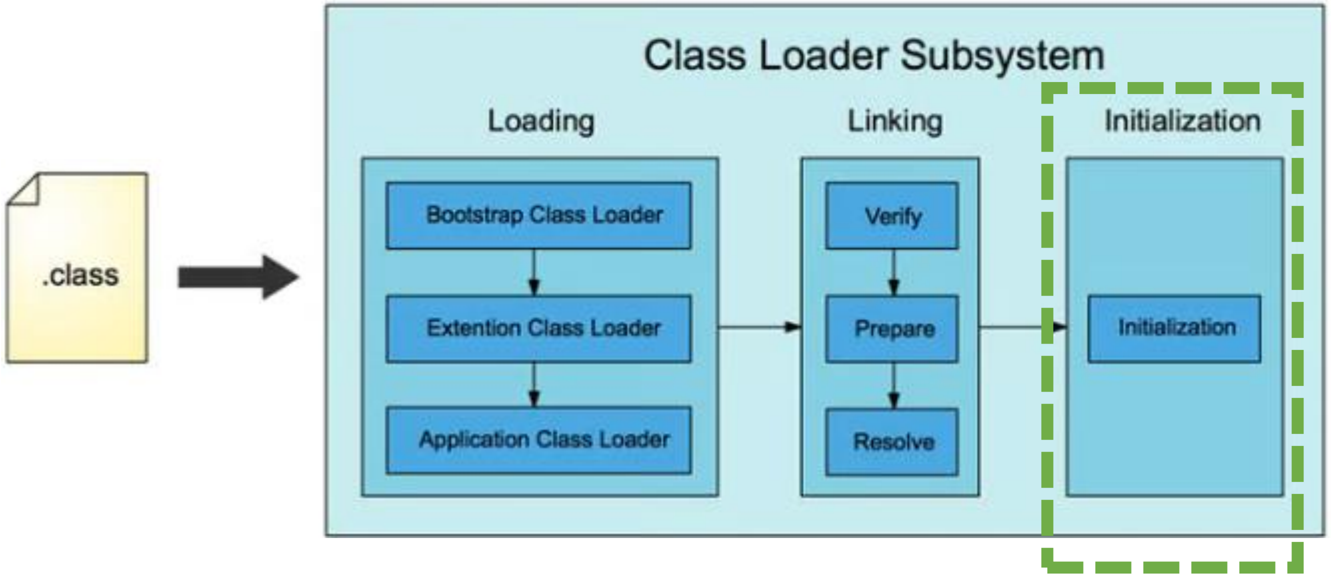


Example: For `println`, which code should be executed?

The resolve step figures out the memory address (offset) for the corresponding method in the method area

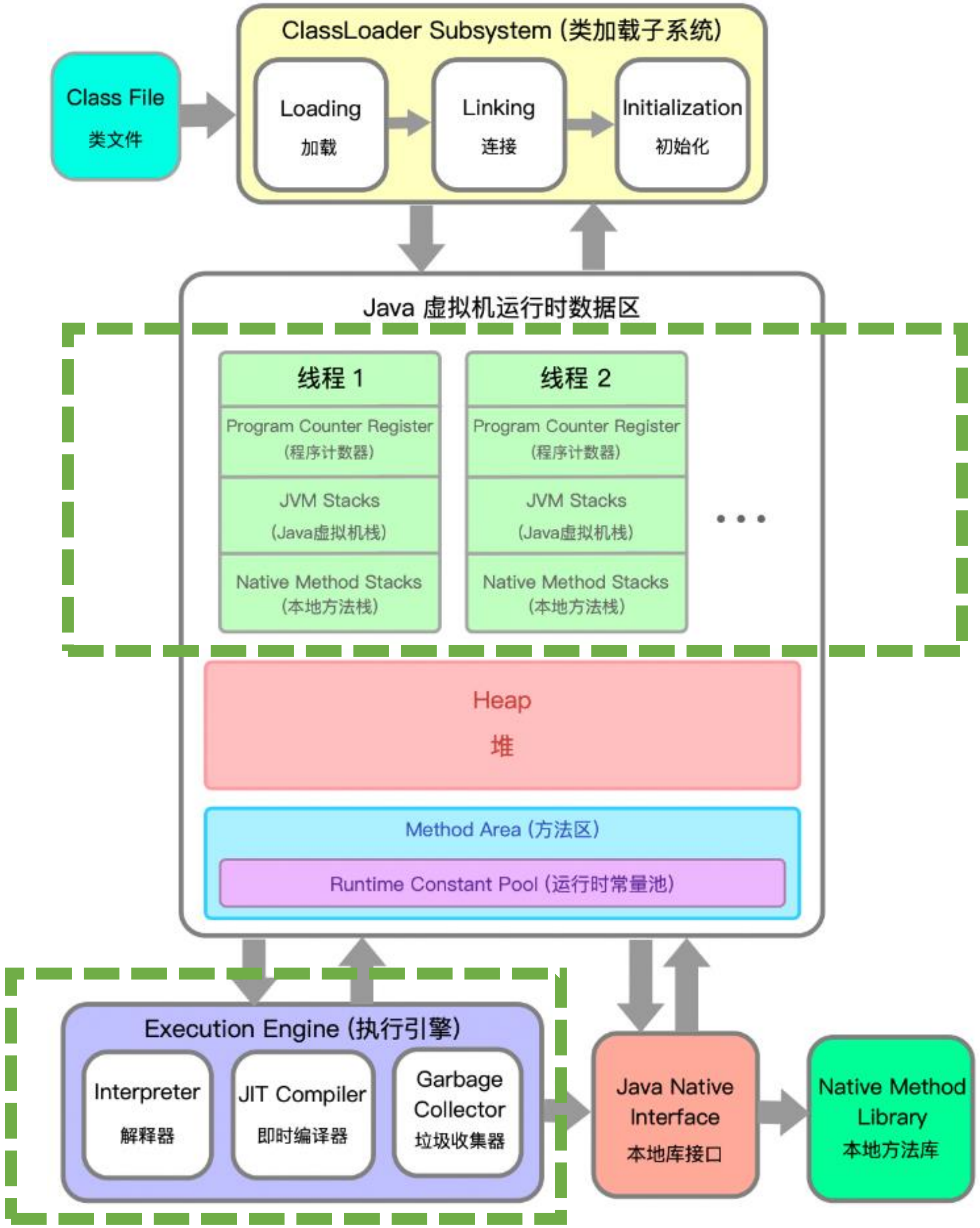
ClassLoader: Initialization

- Initialize static variables
- Execute static blocks



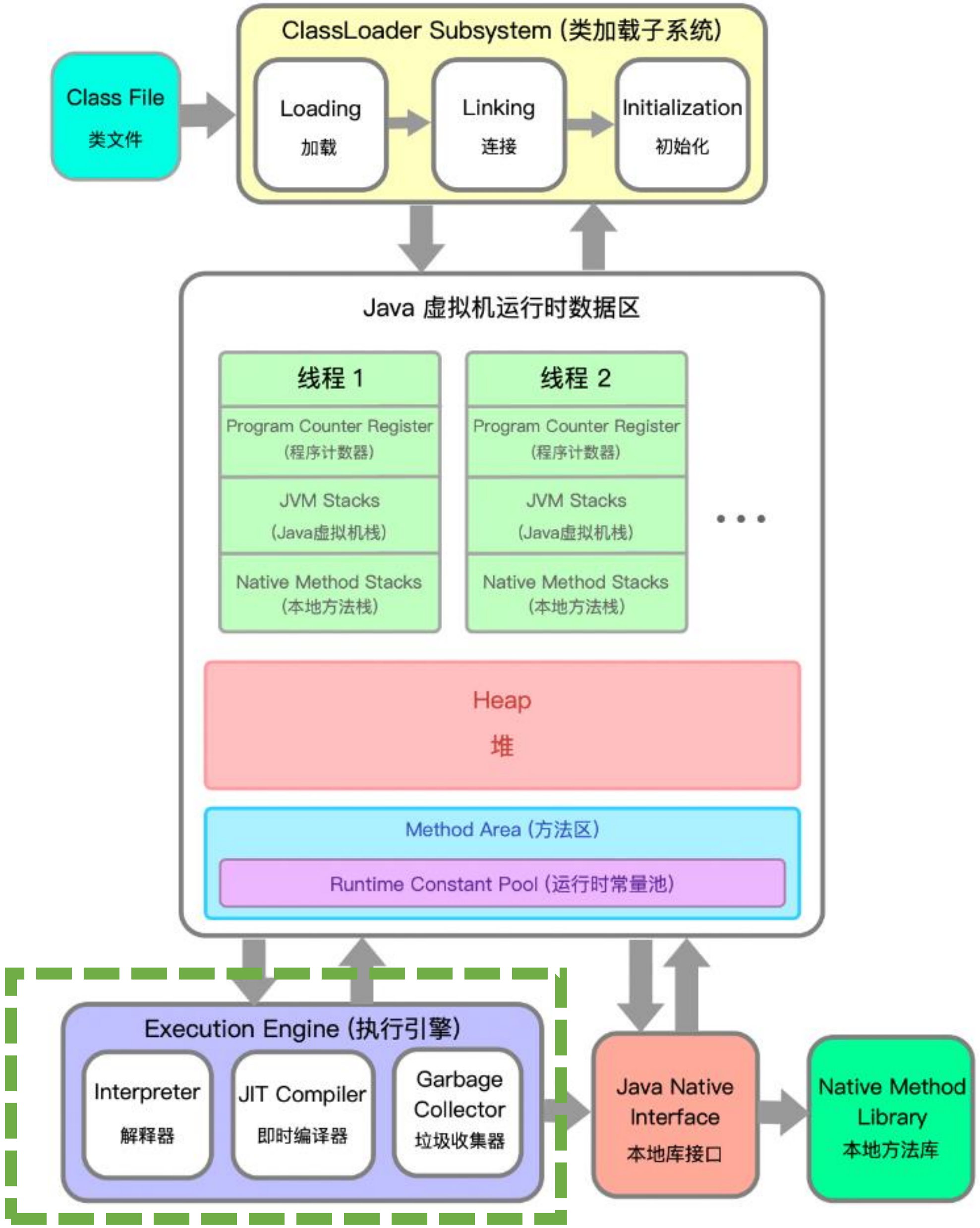
Execution Engine

- JVM uses methods as the basic execution unit.
- The process of a method from its invocation to its completion corresponds to the process of a stack frame being pushed onto and popped off the JVM stack.
- The stack frame stores information such as the local variable table, operand stack, and method return address.

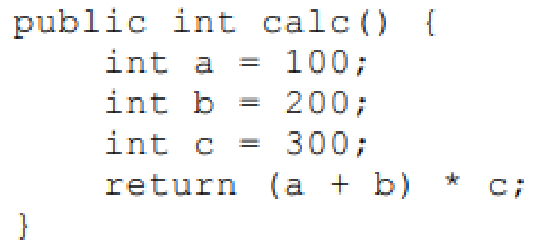


Execution Engine

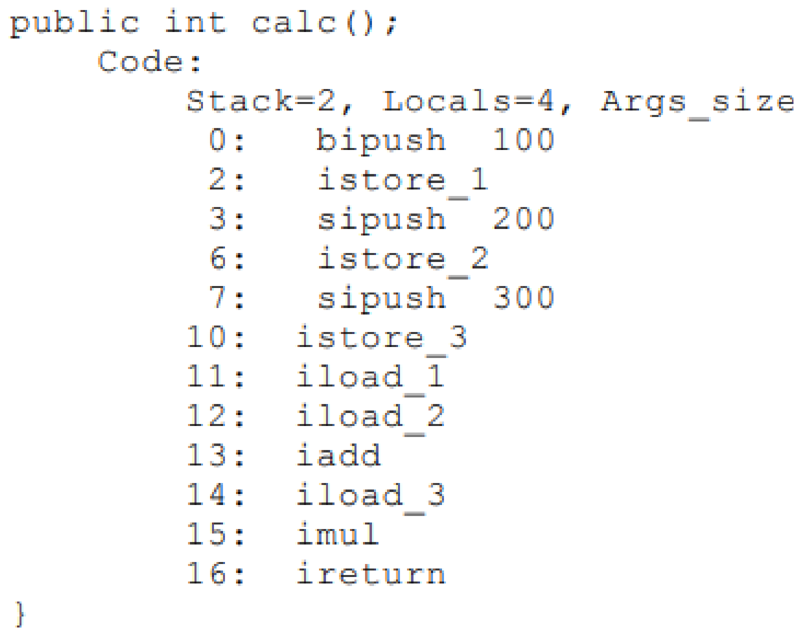
- Execution engine uses references from the local variables to locate the objects in the heap
- Execution engine also locate the class and method information in the method area



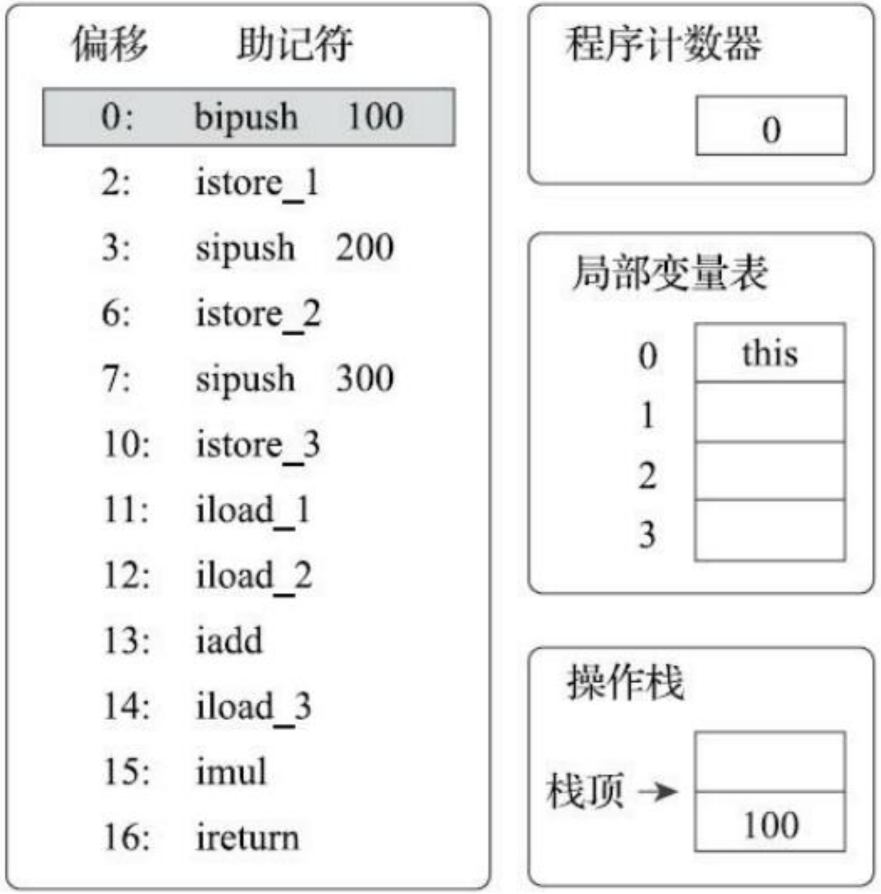
Stack-based Opcodes & Interpreter



```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```



```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:   istore_3
        11:   iload_1
        12:   iload_2
        13:   iadd
        14:   iload_3
        15:   imul
        16:   ireturn
}
```



偏移	助记符
0:	bipush 100
2:	istore_1
3:	sipush 200
6:	istore_2
7:	sipush 300
10:	istore_3
11:	iload_1
12:	iload_2
13:	iadd
14:	iload_3
15:	imul
16:	ireturn

程序计数器

0

局部变量表

0	this
1	
2	
3	

图8-5 执行偏移地址为0的指令的情况

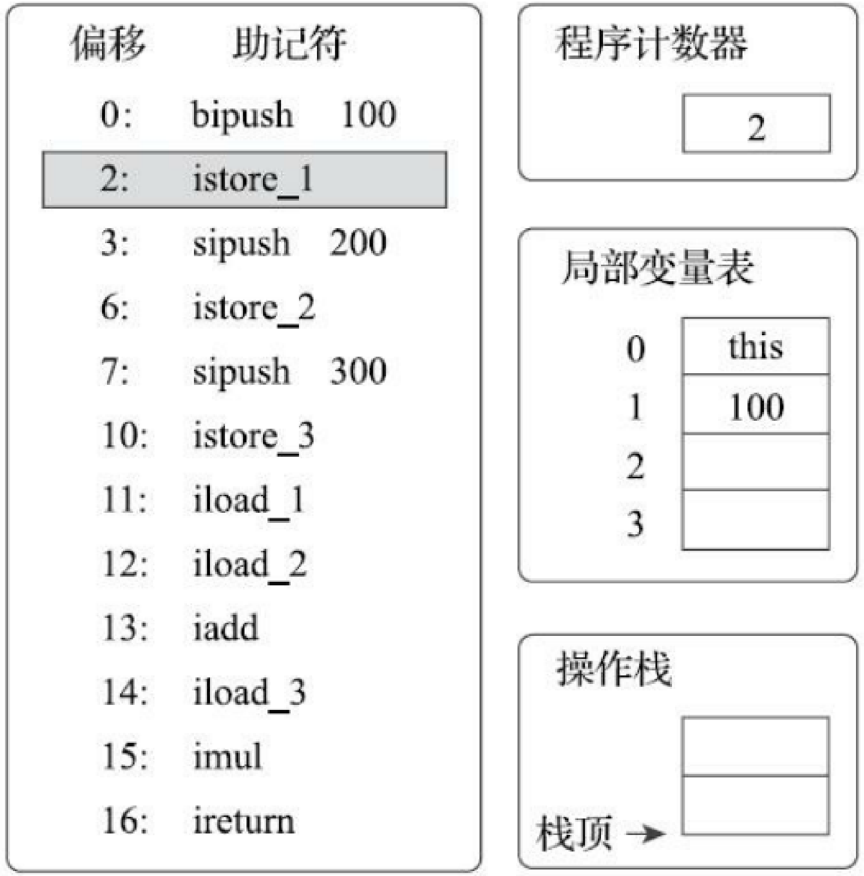
首先，执行偏移地址为0的指令，Bipush指令的作用是将单字节的整型常量值（-128～127）推入操作数栈顶，跟随有一个参数，指明推送的常量值，这里是100。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:   istore_3
        11:   iload_1
        12:   iload_2
        13:   iadd
        14:   iload_3
        15:   imul
        16:   ireturn
}
```



偏移	助记符
0:	bipush 100
2:	istore_1
3:	sipush 200
6:	istore_2
7:	sipush 300
10:	istore_3
11:	iload_1
12:	iload_2
13:	iadd
14:	iload_3
15:	imul
16:	ireturn

程序计数器

2

局部变量表

0	this
1	100
2	
3	

操作栈

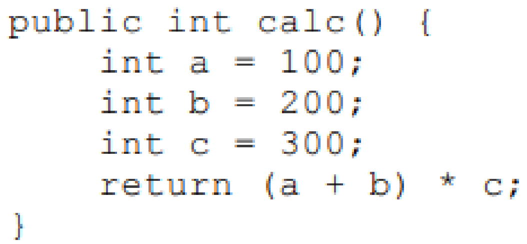
栈顶 →

图8-6　执行偏移地址为1的指令的情况

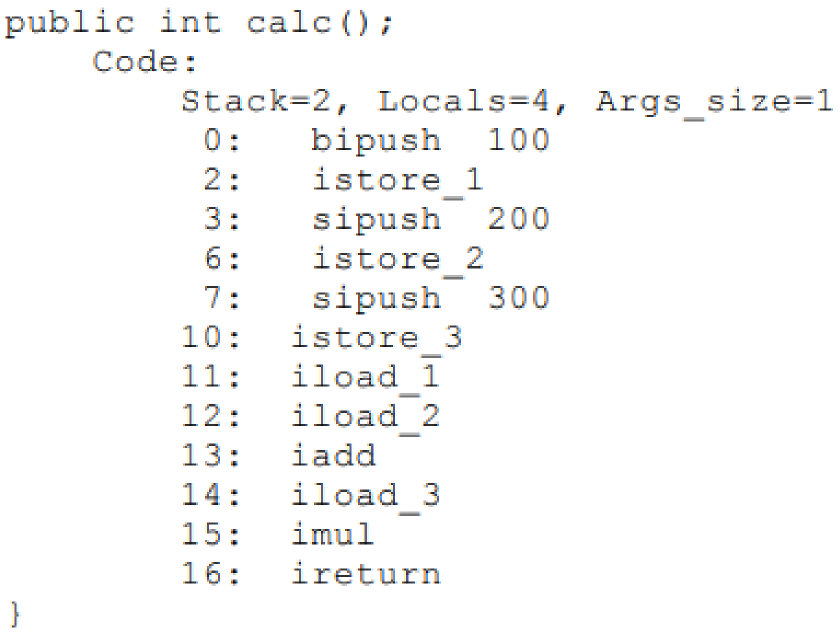
执行偏移地址为2的指令，istore_1指令的作用是将操作数栈顶的整型值出栈并存放到第1个局部变量槽中。后续4条指令（直到偏移为11的指令为止）都是做一样的事情，也就是在对应代码中把变量a、b、c赋值为100、200、300。这4条指令的图示略过。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter



```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```



```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

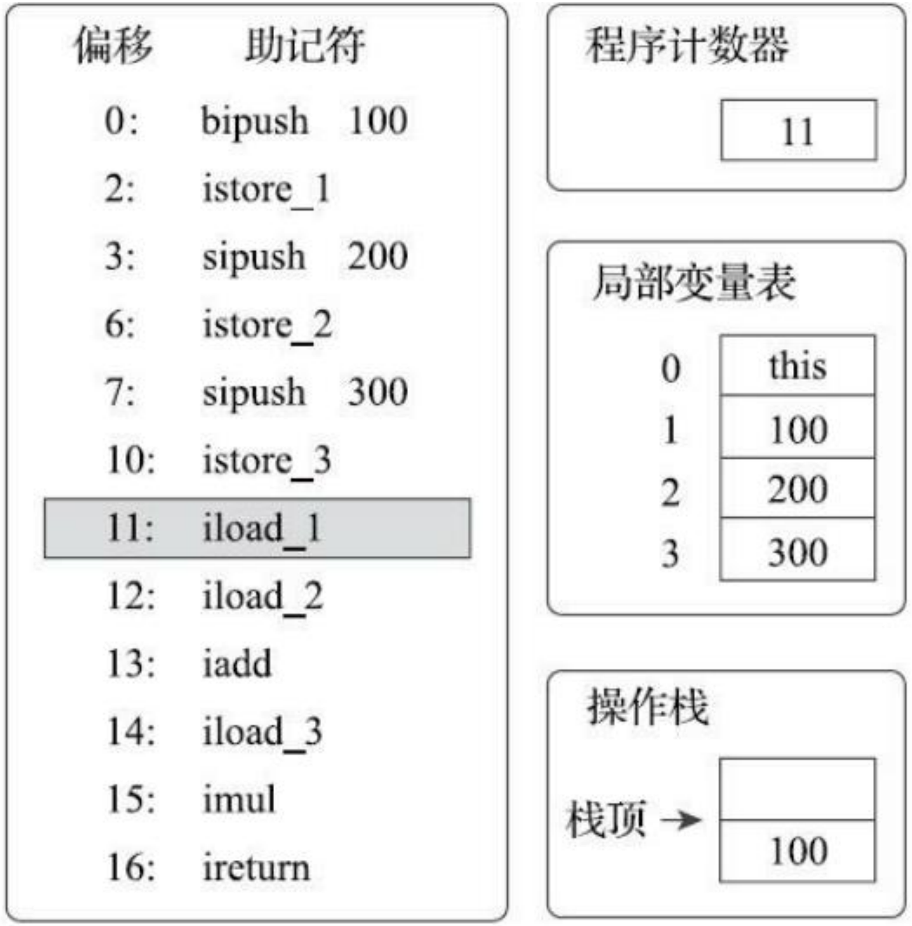


图8-7　执行偏移地址为11的指令的情况

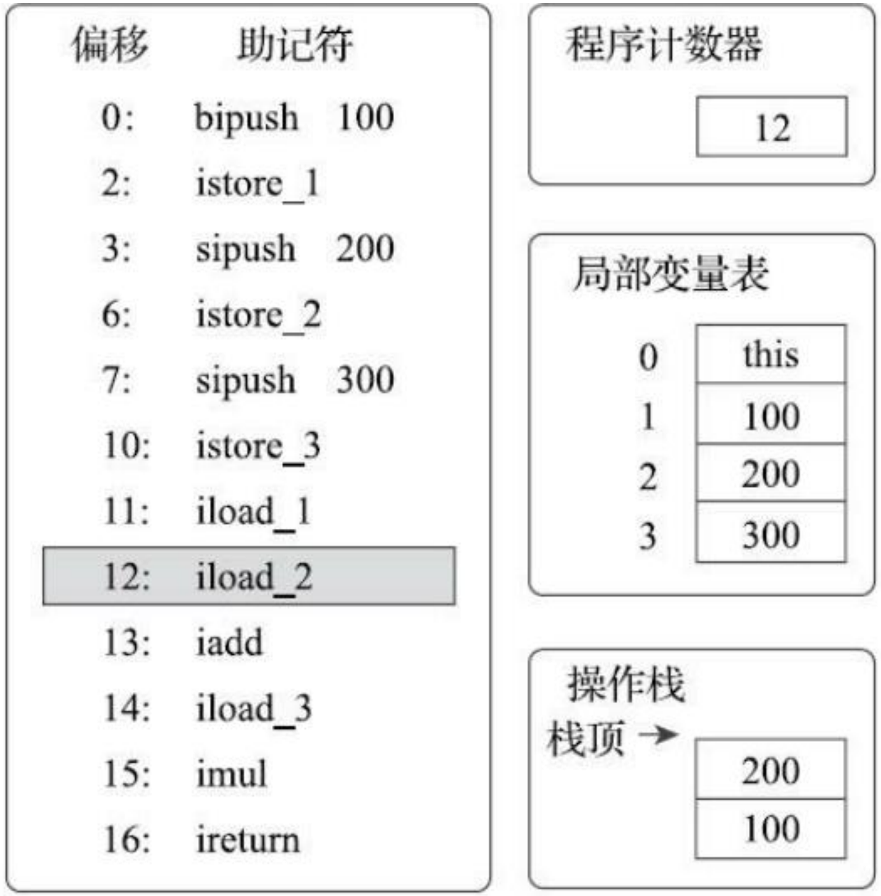
执行偏移地址为11的指令，iload_1指令的作用是将局部变量表第1个变量槽中的整型值复制到操作栈顶。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:   istore_3
        11:   iload_1
        12:   iload_2
        13:   iadd
        14:   iload_3
        15:   imul
        16:   ireturn
}
```



偏移	助记符
0:	bipush 100
2:	istore_1
3:	sipush 200
6:	istore_2
7:	sipush 300
10:	istore_3
11:	iload_1
12:	iload_2
13:	iadd
14:	iload_3
15:	imul
16:	ireturn

程序计数器

12

局部变量表

0	this
1	100
2	200
3	300

操作栈

栈顶 →

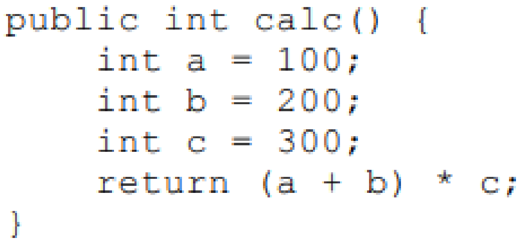
200
100

图8-8 执行偏移地址为12的指令的情况

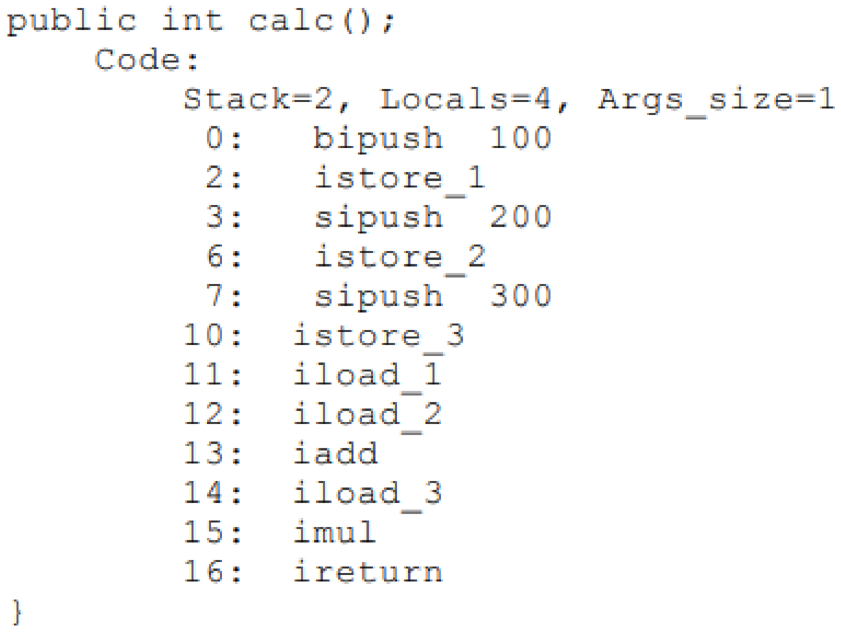
执行偏移地址为12的指令，iload_2指令的执行过程与iload_1类似，把第2个变量槽的整型值入栈。

《深入理解Java虚拟机》第三版，周志明

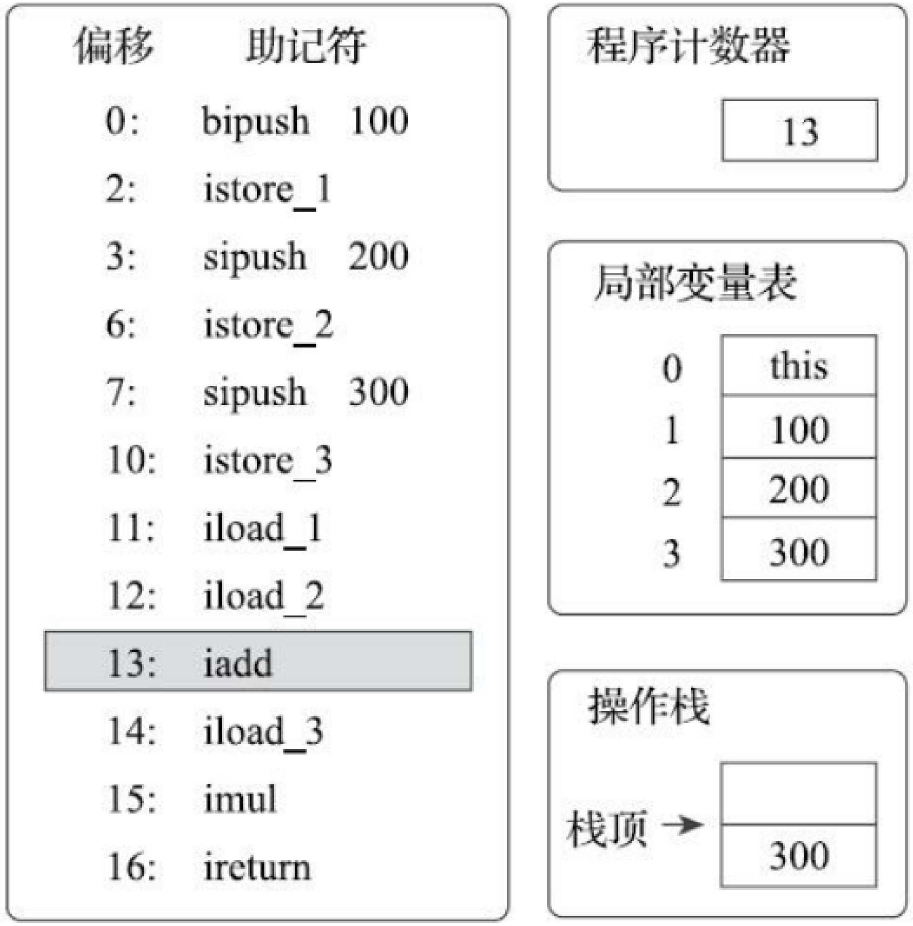
Stack-based Opcodes & Interpreter



```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```



```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:   istore_3
        11:   iload_1
        12:   iload_2
        13:   iadd
        14:   iload_3
        15:   imul
        16:   ireturn
}
```



偏移	助记符
0:	bipush 100
2:	istore_1
3:	sipush 200
6:	istore_2
7:	sipush 300
10:	istore_3
11:	iload_1
12:	iload_2
13:	iadd
14:	iload_3
15:	imul
16:	ireturn

程序计数器

13

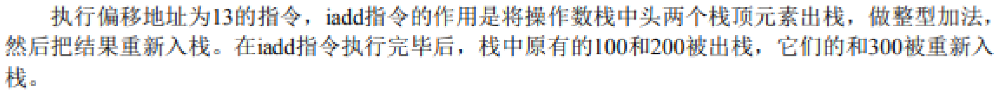
局部变量表

0	this
1	100
2	200
3	300

操作栈

300

图8-9　执行偏移地址为13的指令的情况



执行偏移地址为13的指令，iadd指令的作用是将操作数栈中头两个栈顶元素出栈，做整型加法，然后把结果重新入栈。在iadd指令执行完毕后，栈中原有的100和200被出栈，它们的和300被重新入栈。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

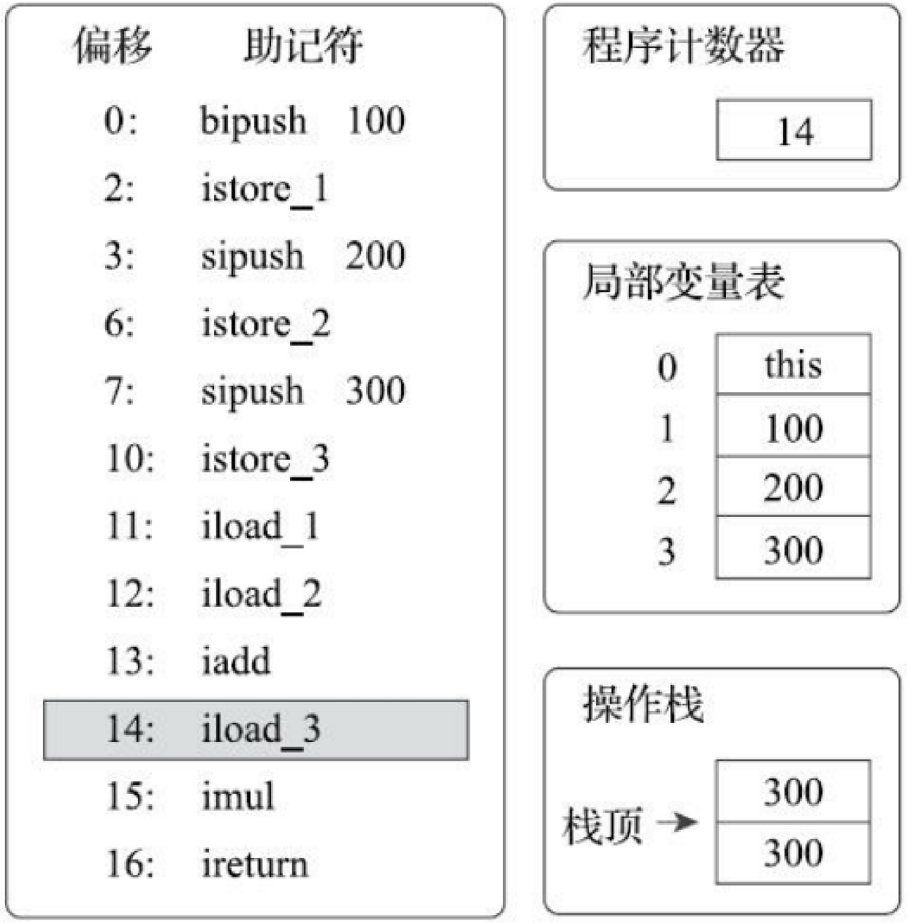
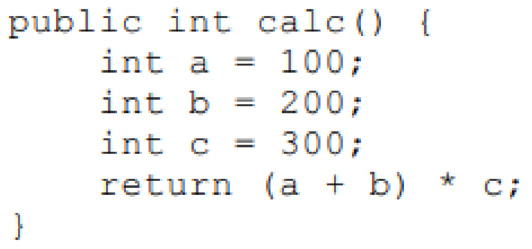


图8-10　执行偏移地址为14的指令的情况

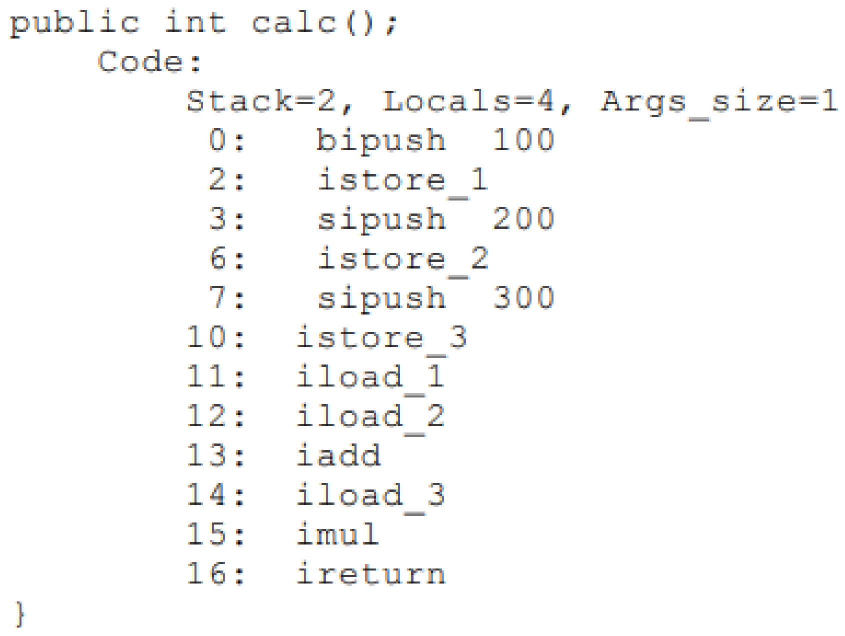
执行偏移地址为14的指令，iload_3指令把存放在第3个局部变量槽中的300入栈到操作数栈中。这时操作数栈为两个整数300。下一条指令imul是将操作数栈中头两个栈顶元素出栈，做整型乘法，然后把结果重新入栈，与iadd完全类似，所以笔者省略图示。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter



```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```



```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

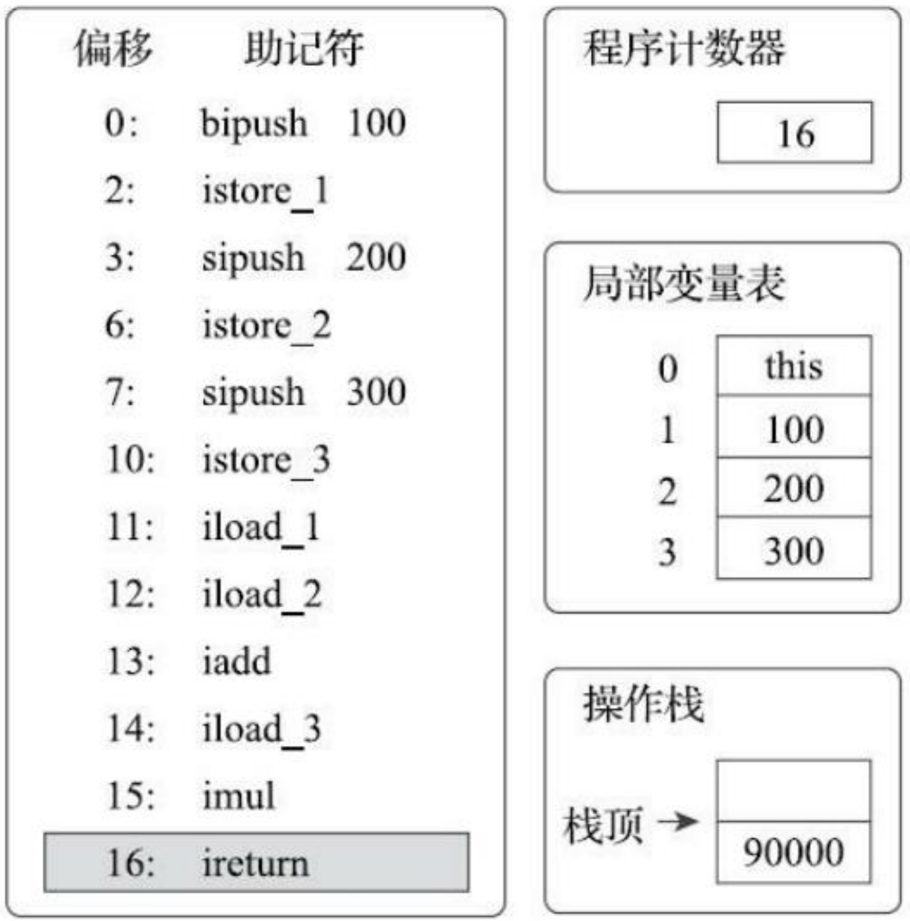
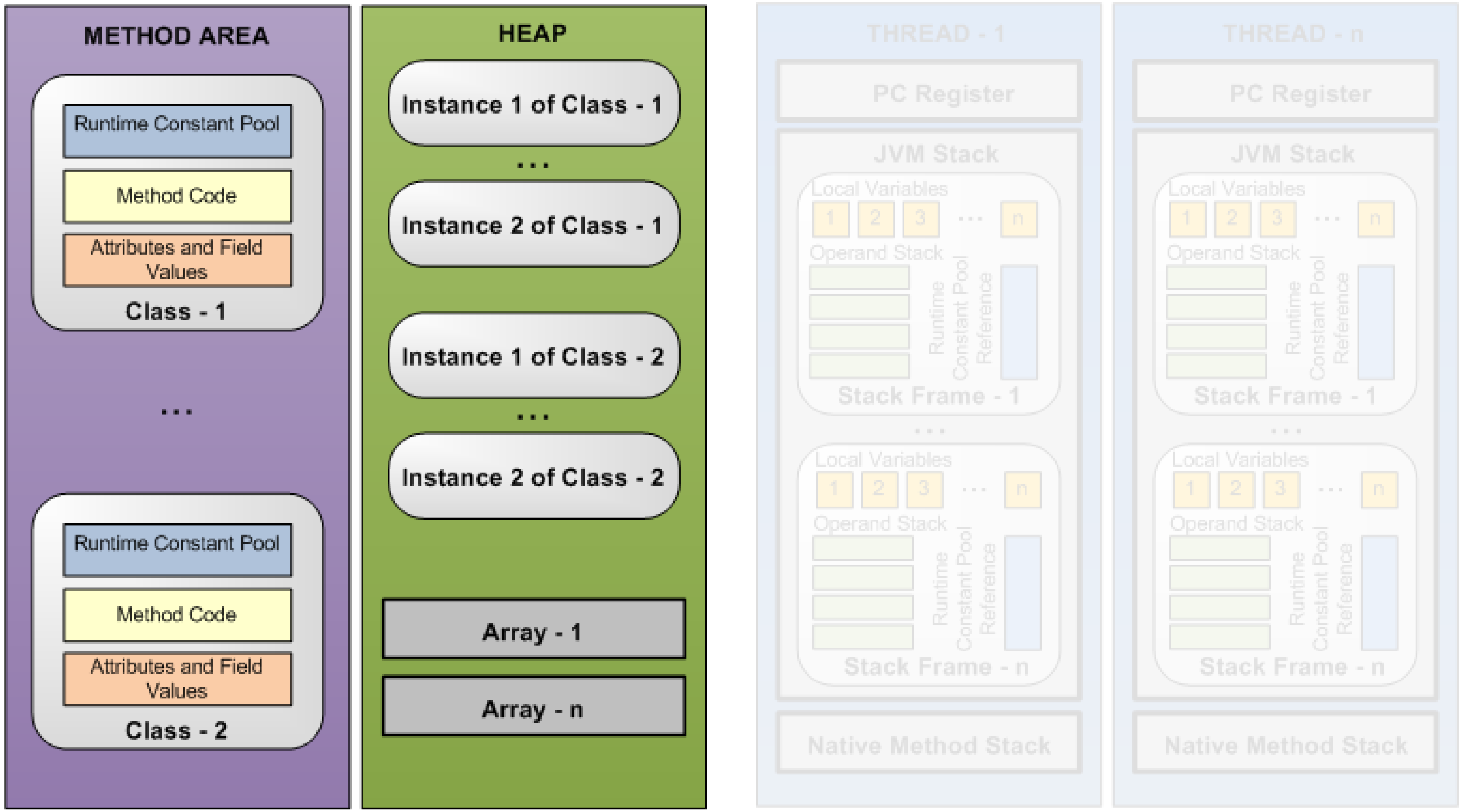


图8-11　执行偏移地址为16的指令的情况

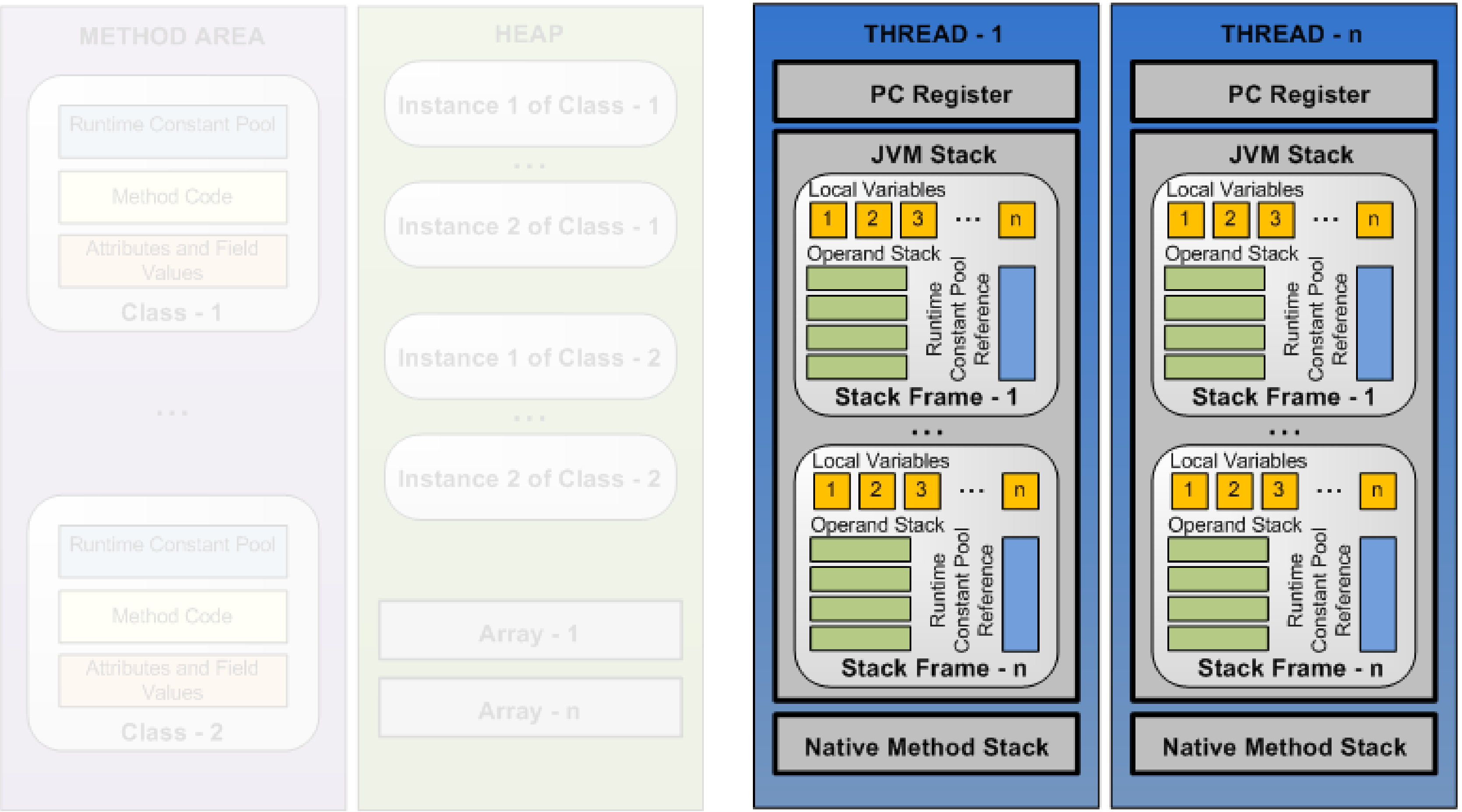
执行偏移地址为16的指令，ireturn指令是方法返回指令之一，它将结束方法执行并将操作数栈顶的整型值返回给该方法的调用者。到此为止，这段方法执行结束。

《深入理解Java虚拟机》第三版，周志明

Shared among Threads



Exclusive to Each Thread

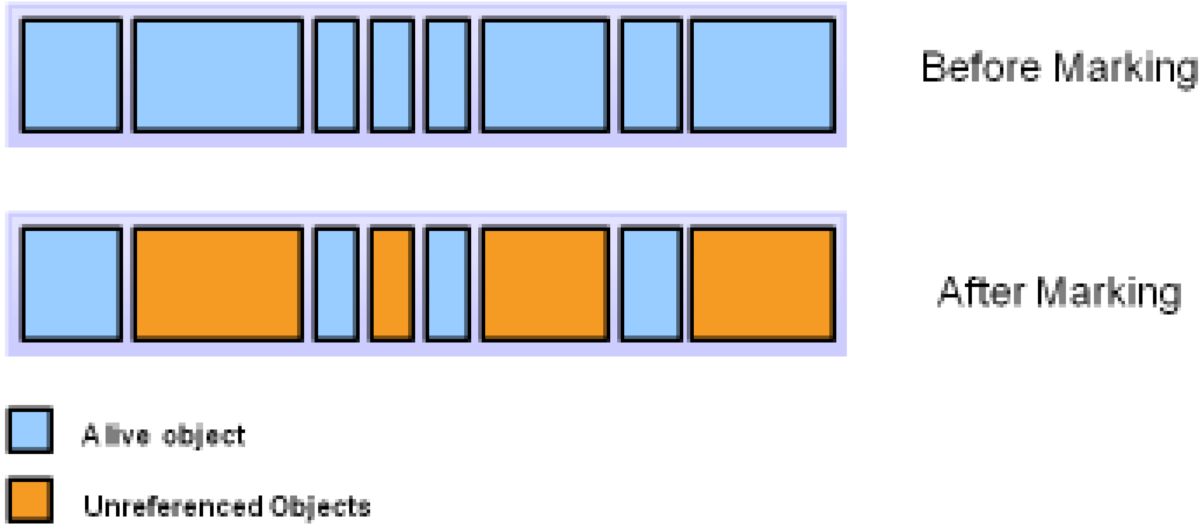


Garbage Collection Algorithm

Naïve Algorithm: scans the entire heap memory to identify and reclaim all unreachable objects.

Disadvantages

- Inefficient
- Memory fragmentation



https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Garbage Collection Algorithm

The Young Generation is where all new objects are allocated and aged.

When the young generation fills up, this causes a minor garbage collection.

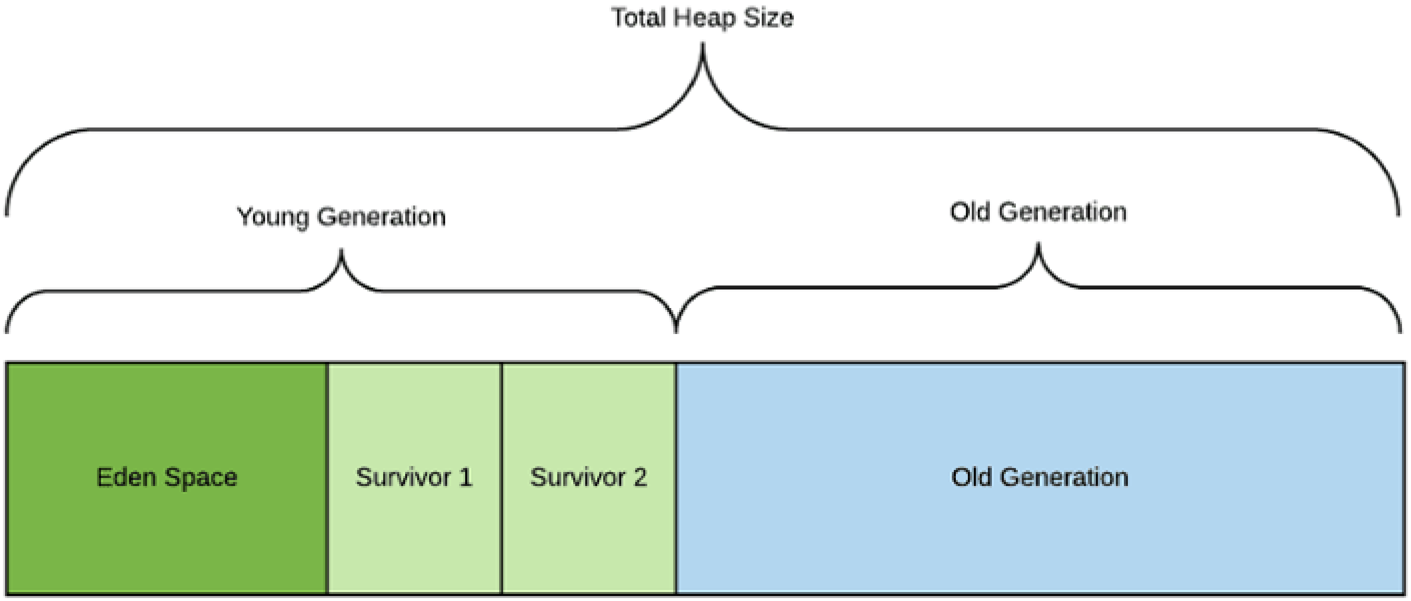


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

Garbage Collection Algorithm

The Old Generation is used to store long surviving objects.

Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation.

Eventually the old generation needs to be collected. This event is called a major garbage collection.

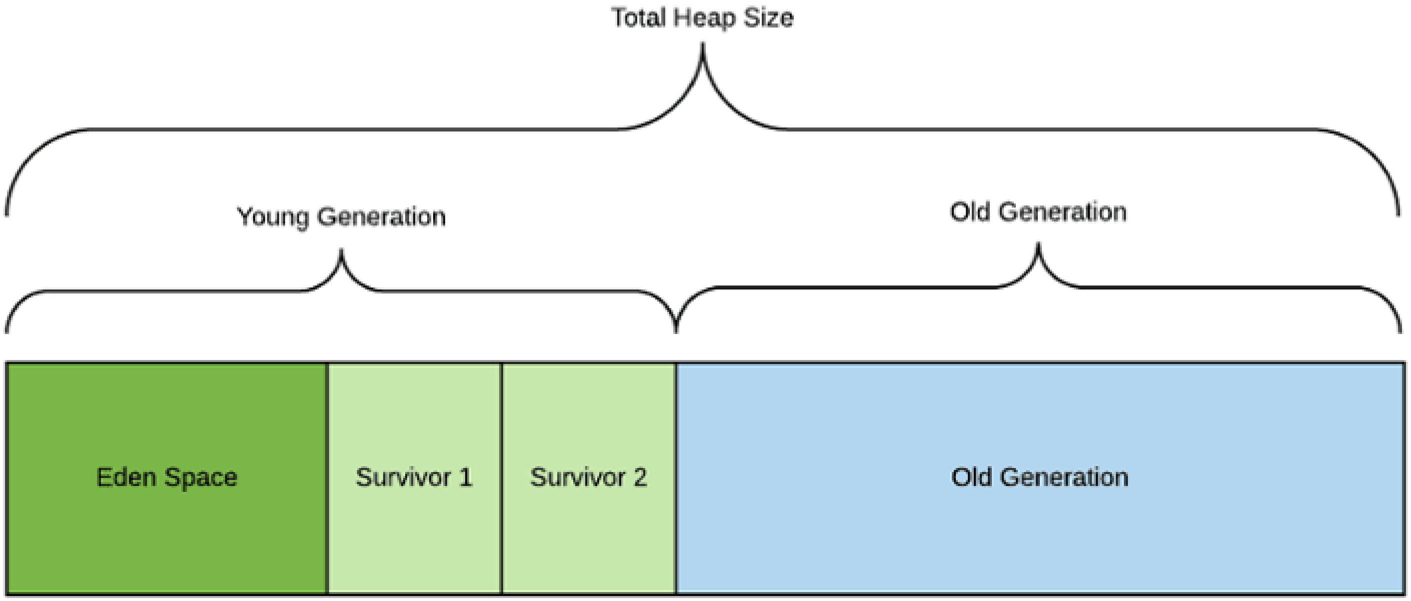


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

Animation: https://speakerdeck.com/deepu105/jvm-minor-gc

Next Lecture

- Project Presentation
- Course Review