

第 4 章

Java 字节码实战

本章摘要

- ◎ Java 字节码的二进制格式
- ◎ Java 字节码的魔数与版本
- ◎ Java 字节码的常量池
- ◎ Java 字节码的类继承
- ◎ Java 字节码的字段存储
- ◎ Java 字节码的方法格式

我们在前面章节与各位道友一起分享了 JVM 内部调用 Java 方法的核心技术原理，接着从宏观层面谈了 Java 在面向对象、数据结构上所做的技术选择。要想深刻理解 JVM 执行引擎的机制，就必须对 JVM 内部的数据结构有深入了解，而要了解 JVM 内部的数据结构，就必须要知道从 Java 源程序过渡到 JVM 内部数据结构的中间桥梁——Java 字节码。下面就与各位道友一起探讨 Java 字节码文件的格式组成。

4.1 字节码格式初探

相信绝大多数人初次接触字节码相关的知识时，都会觉得比较抽象。因此为了让大家能够真正熟悉并理解 Java 字节码文件的格式，下面将通过一个实际的 Java 类的字节码文件格式分析来进行讲解。如果你对这一部分内容已经非常熟悉，可以跳过本节，这完全不会影响对后续内容的阅读和理解。如果你对这一部分内容仅仅停留在“知道”的层面，那么阅读本节可以让你真正理解字节码。

4.1.1 准备测试用例

所构造的 Java 测试类如下：

清单：Test.java

作用：java 测试类

```
public class Test {  
    public int a = 3;  
    static Integer si = 6;  
    String s = "Hello World!";  
  
    public static void main(String[] args){  
        Test test = new Test();  
        test.a = 8;  
        si = 9;  
    }  
  
    private void test(){  
        this.a = a;  
    }  
}
```

这个测试类虽然简单，然而五脏俱全，包含类变量、成员变量、字符串和类成员方法，同时包含一个入口主函数 main()，并且在 main() 函数中实例化了 Test 类。为了简单起见，该类没有继承任何类，没有使用任何类库，这样有助于在分析字节码文件格式时降低难度。

4.1.2 使用 javap 命令分析字节码文件

在%JAVA_HOME%/bin 目录下包含一个 javap 命令工具，javap 命令能够分析出一个给定的 java 类中的字节码信息。这里使用 javap 命令来分析上述用于测试的 Test 类。首先编译 Test.java 类，得到 Test.class 文件，进入 Test.class 文件所在目录，输入如下命令：

```
javap -verbose Test
```

执行 javap 命令后，将输出如下信息：

清单：Test.class

作用：使用 javap 命令分析 class 字节码文件内容

```
D:\work\projects\person\bin>javap -verbose Test  
Compiled from "Test.java"  
public class Test extends java.lang.Object  
  SourceFile: "Test.java"  
  minor version: 0
```

```

major version: 50
Constant pool:
const #1 = class           #2;    . // Test
const #2 = Asciz            Test;
const #3 = class           #4;    // java/lang/Object
const #4 = Asciz            java/lang/Object;
const #5 = Asciz            a;
const #6 = Asciz            I;
const #7 = Asciz            si;
const #8 = Asciz            Ljava/lang/Integer;;
const #9 = Asciz            s;
const #10 = Asciz           Ljava/lang/String;;
const #11 = Asciz           <clinit>;
const #12 = Asciz           ()V;
const #13 = Asciz           Code;
const #14 = Method          #15.#17;      //
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
const #15 = class           #16;    // java/lang/Integer
const #16 = Asciz            java/lang/Integer;
const #17 = NameAndType     #18:#19;// valueOf:(I)Ljava/lang/Integer;
const #18 = Asciz            valueOf;
const #19 = Asciz            (I)Ljava/lang/Integer;;
const #20 = Field           #1.#21; // Test.si:Ljava/lang/Integer;
const #21 = NameAndType     #7:#8;// si:Ljava/lang/Integer;
const #22 = Asciz            LineNumberTable;
const #23 = Asciz            LocalVariableTable;
const #24 = Asciz            <init>;
const #25 = Method          #3.#26; // java/lang/Object."<init>":()V
const #26 = NameAndType     #24:#12;// "<init>":()V
const #27 = Field           #1.#28; // Test.a:I
const #28 = NameAndType     #5:#6;// a:I
const #29 = String          #30;    // Hello World!
const #30 = Asciz            Hello World!;;
const #31 = Field           #1.#32; // Test.s:Ljava/lang/String;
const #32 = NameAndType     #9:#10;// s:Ljava/lang/String;
const #33 = Asciz            this;
const #34 = Asciz            LTest;;
const #35 = Asciz            main;
const #36 = Asciz            ([Ljava/lang/String;)V;
const #37 = Method          #1.#26; // Test."<init>":()V
const #38 = Asciz            args;
const #39 = Asciz            [Ljava/lang/String;;
const #40 = Asciz            test;
const #41 = Asciz            SourceFile;
const #42 = Asciz            Test.java;

{

```

```
public int a;
static java.lang.Integer si;
java.lang.String s;

static {};
Code:
Stack=1, Locals=0, Args_size=0
0: bipush 6
2: invokestatic #14; //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5: putstatic #20; //Field si:Ljava/lang/Integer;
8: return

public Test();
Code:
Stack=2, Locals=1, Args_size=1
0: aload_0
1: invokespecial #25; //Method java/lang/Object."<init>":()V
4: aload_0
5: iconst_3
6: putfield #27; //Field a:I
9: aload_0
10: ldc #29; //String Hello World!
12: putfield #31; //Field s:Ljava/lang/String;
15: return

public static void main(java.lang.String[]);
Code:
Stack=2, Locals=2, Args_size=1
0: new #1; //class Test
3: dup
4: invokespecial #37; //Method "<init>":()V
7: astore_1
8: aload_1
9: bipush 8
11: putfield #27; //Field a:I
14: bipush 9
16: invokestatic #14; //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
19: putstatic #20; //Field si:Ljava/lang/Integer;
22: return
}
```

使用 `javap -verbose` 命令分析一个字节码文件时，将会分析字节码文件的魔数、版本号、常量池、类信息、类的构造函数、类中所包含的方法信息以及类（成员）变量信息。需要注意的是，每一次执行 `javap` 命令所输出的信息内容一定是相同的，但是信息的先后顺序则不保证完

全一致，例如，常量池中的元素编号每次都不保证相同。

`javap -verbose` 会列出 Java 类中的全部常量池，其格式如下：

清单：javap -verbose 显示常量池的格式

```
Constant pool:
const #1 = class           #2;      // Test
const #2 = Asciz            Test;
const #3 = class           #4;
.......
```

每一个 `const` 后面的#号后的数字代表了常量池项在常量池中的索引，当 JVM 在解析类的常量池信息时，常量池项的索引与此一致。

4.1.3 查看字节码二进制

下面使用十六进制工具打开 `Test.class` 文件，让大家预先感受一下字节码文件的内容到底是什么。打开文件后得到的十六进制文件内容如下（系笔者全手工录入——逐个字符录入）：

清单：project/src/Test.java

作用：Java 测试类，用于跟踪调试 JVM 的代码运行机制

00000000	CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 042·3.....
00000010	54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00	ang/Object···a··
00000030	01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F	·I···si···Ljava/
00000040	6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01	lang/Integer;···
00000050	73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53	s···Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74	tring;···<clinit
00000070	3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00	>···()V···Code··
00000080	0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01	ng/Integer.....
000000a0	00 07 76 61 6C 75 65 4F 66 01 00 16 28 49 29 4C	··valueOf···(I)L
000000b0	6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65	java/lang/Intege
000000c0	72 3B 09 00 01 00 15 0C 00 07 00 08 01 00 0F 4C	r;···.....L
000000d0	69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00	ineNumberTable··
000000e0	12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61	.LocalVariableTa
000000f0	62 6C 65 01 00 06 3C 69 6E 69 74 3E 0A 00 03 00	ble···<init>···
00000100	1A 0C 00 18 00 0C 09 00 01 00 1C 0C 00 05 00 06
00000110	08 00 1E 01 00 0C 48 65 6C 6C 6F 20 57 6F 72 6CHello Worl
00000120	64 21 09 00 01 00 20 0C 00 09 00 0A 01 00 04 74	d!.....
00000130	68 69 73 01 00 06 4C 54 65 73 74 3B 01 00 04 6D	his···LTest;···m
00000140	61 69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61	ain···([Ljava/la
00000150	6E 67 2F 53 74 72 69 6E 67 3B 29 56 0A 00 01 00	ng/String;)V···
00000160	1A 0A 00 01 00 27 0C 00 28 00 29 01 00 04 73 65'···(·)···se

00000170	74 41 01 00 16 28 4C 6A 61 76 61 2F 6C 61 6E 67 tA ··· (Ljava/lang
00000180	2F 49 6E 74 65 67 65 72 3B 29 56 01 00 04 61 72 /Integer;)V ··· ar
00000190	67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 gs ··· [Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String; ··· test.
000001b0	00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56/.0 ··· intV
000001c0	61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75 alue ··· ()I ··· Sou
000001d0	72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile ··· Test.j
000001e0	61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.!
000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 04 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D*
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04
00000270	00 00 00 01 00 04 00 02 00 09 00 06 00 0F 00 01
00000280	00 17 00 00 00 0C 00 01 00 00 10 00 21 00 22" ..
00000290	00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 00 4E# ..\$..N
000002a0	00 02 00 02 00 00 00 12 BB 00 01 59 B7 00 25 4CY ..%L
000002b0	2B 10 08 B8 00 0E B6 00 26 B1 00 00 00 02 00 16 +
000002c0	00 00 00 0E 00 03 00 00 00 09 00 08 00 0A 00 11
000002d0	00 0B 00 17 00 00 00 16 00 02 00 00 00 12 00 2A
000002e0	00 2B 00 00 00 08 00 0A 00 2C 00 22 00 01 00 01 +, ..
000002f0	00 28 00 29 00 01 00 0D 00 00 00 41 00 02 00 02 .() ..A ..
00000300	00 00 00 09 2A 2B B6 00 2D B5 00 1B B1 00 00 00 ..*+ ..
00000310	02 00 16 00 00 00 0A 00 02 00 00 00 0E 00 08 00
00000320	0F 00 17 00 00 00 16 00 02 00 00 00 09 00 21 00" ..!
00000330	22 00 00 00 00 00 09 00 05 00 08 00 01 00 01 00 "
00000340	31 00 00 00 02 00 32 1

每一行显示 16 个字节，并且使用十六进制显示，因此一行显示 32 个数字。如果直接看十六进制的一大串数字，肯定没有人能够知道这一大串数字究竟代表什么，因此需要一定的格式进行组织。下面就拿实际的例子来逐个分析 JVM 所规定的格式。

4.2 魔数与版本

本节基于上文测试用例中所使用的 Test.class 字节码文件，带领大家一起详细分析字节码文件的组成格式（一共 10 个组成部分）。

在具体分析之前，先给出一张字节码文件内容的全图，如图 4.1 所示。

```

00000000 CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 04 .....2.3.....
00000010 54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/l
00000020 61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object...a..
00000030 01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01 lang/Integer;...
00000050 73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060 74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit>
00000070 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080 0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61 .....java/la
00000090 6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer.....
000000a0 00 07 76 61 6C 75 65 4F 66 01 00 16 28 49 29 4C ..valueOf...()L
000000b0 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 java/lang/Intege
000000c0 72 3B 09 00 01 00 15 0C 00 07 00 08 01 00 0F 4C r;.....L
000000d0 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 ineNumberTable..
000000e0 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 .LocalVariableTa
000000f0 62 6C 65 01 00 06 3C 69 6E 69 74 3E 0A 00 03 00 ble...<init>...
00000100 1A 0C 00 18 00 0C 09 00 01 00 1C 0C 00 05 00 06 .....
00000110 08 00 1E 01 00 0C 48 65 6C 6C 6F 20 57 6F 72 6C .....Hello Worl
00000120 64 21 09 00 01 00 20 0C 00 09 00 0A 01 00 04 74 d!... ....t
00000130 68 69 73 01 00 06 4C 54 65 73 74 3B 01 00 04 6D his...LTest;...m
00000140 61 69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 ain...([Ljava/la
00000150 6E 67 2F 53 74 72 69 6E 67 3B 29 56 0A 00 01 00 ng/String;)V...
00000160 1A 0A 00 01 00 27 0C 00 28 00 29 01 00 04 73 65 .....'(..)se
00000170 74 41 01 00 16 28 4C 6A 61 76 61 2F 6C 61 6E 67 tA...Ljava/lang
00000180 2F 49 6E 74 65 67 65 72 3B 29 56 01 00 04 61 72 /Integer;)V...ar
00000190 67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 ga...[Ljava/lang
000001a0 2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String;...test.
000001b0 00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56 ...../.0...intV
000001c0 61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6E 75 alue...()I...Sou
000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.j
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava;.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 ..... .
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 00 01 00 ..... .
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 ..... .
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 ..... .
00000230 00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 ..... .
00000240 00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 ..... F...
00000250 00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D .....*....*...
00000260 B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 ..... .
00000270 00 00 00 01 00 04 00 02 00 09 00 04 00 0F 00 01 ..... .
00000280 00 17 00 00 00 0C 00 01 00 00 00 10 00 21 00 22 .....!."
00000290 00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 4E .....#.$.....N
000002a0 00 02 00 02 00 00 00 12 BB 00 01 59 B7 00 25 4C .....Y..$L
000002b0 2B 10 08 B8 00 0E B6 00 26 B1 00 00 00 02 00 16 +.....g.....
000002c0 00 00 00 0E 00 03 00 00 00 07 00 08 00 08 00 11 ..... .
000002d0 00 09 00 17 00 00 00 16 00 02 00 00 00 12 00 2A .....*....*
000002e0 00 2B 00 00 00 08 00 0A 00 2C 00 22 00 01 00 01 ..+....."....
000002f0 00 28 00 29 00 01 00 0D 00 00 00 41 00 02 00 02 ..(.)....A...
00000300 00 00 00 09 2A 2B B6 00 2D B5 00 1B B1 00 00 00 ...*+...-.....
00000310 02 00 16 00 00 00 0A 00 02 00 00 00 0C 00 08 00 ..... .
00000320 0D 00 17 00 00 00 16 00 02 00 00 00 09 00 21 00 .....!
00000330 22 00 00 00 00 00 09 00 05 00 08 00 01 00 01 00 ".....".
00000340 31 00 00 00 02 00 32 00 1.....2

```

图 4.1 总览 Java 字节码格式

下面会基于图 4.1 截出若干小图片，以分析字节码文件内容的格式。但是为了节省篇幅，每次仅仅截其中一部分，读者可以根据每一行左边的行号进行定位。同时，为了使不同数据结构的前后排列布局显示得更加清晰，下面的截图会将上下几行信息截下来，这样读者能够对整体字节码结构做到心中有数。

4.2.1 魔数

所有.class 字节码文件的开始 4 个字节都是魔数，并且其值一定是 0xCAFEBAE。注意，这里的 CAFEBAE 是指十六进制数值，并不是字符串“CAFEBAE”，如图 4.2 所示。如果开始 4 字节不是 0xCAFEBAE，则 JVM 将会认为该文件不是.class 字节码文件，并拒绝解析。

```

00000000 CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 04 .....|...2.3.....
00000010 54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/1
00000020 61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object...a...
00000030 01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01 lang/Integer;...
00000050 73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060 74 72 69 67 65 63 6B 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit
00000070 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080 0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61 .....java/la
00000090 6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer;....
000000a0 00 07 76 61 6C 75 65 4F 66 01 00 16 28 49 29 4C ..valueOf...(I)L
000000b0 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 java/lang/Intege

```

图 4.2 Java 字节码中的魔数

4.2.2 版本号

根据字节码文件规范，魔数之后的 4 个字节为版本信息，前两个字节表示 major version，即主版本号；后两个字节表示 minor version，即次版本号。这里版本号的值为 0x00000032，对应的十进制数是 50。

目前已发布的 version 包括：1.1(45)、1.2(46)、1.3(47)、1.4(48)、1.5(49)、1.6(50)、1.7(51)。据此可以知道，该 class 文件是 JDK 1.6 编译的，如图 4.3 所示。

```

00000000 CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 04 .....|...2.3.....
00000010 54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/1
00000020 61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object...a...
00000030 01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01 lang/Integer;...
00000050 73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060 74 72 69 67 65 63 6B 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit
00000070 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080 0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61 .....java/la
00000090 6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer;....
000000a0 00 07 76 61 6C 75 65 4F 66 01 00 16 28 49 29 4C ..valueOf...(I)L
000000b0 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 java/lang/Intege

```

图 4.3 Java 字节码文件中的版本号

4.3 常量池

常量池是.class 字节码文件中非常重要和核心的内容，一个 Java 类中绝大多数的信息都由常量池描述，尤其是 Java 类中定义的变量和方法，都由常量池保存。注意，对 JVM 有所研究的人，可能都知道 JVM 的内存模型中，有一块就是常量池，JVM 堆区的常量池就是用于保存每一个 Java 类所对应的常量池的信息的，一个 Java 应用程序中所包含的所有 Java 类的常量池，组成了 JVM 堆区中大的常量池。

4.3.1 常量池的基本结构

Java 类所对应的常量池主要由常量池数量和常量池数组两部分组成（如图 4.4 所示），常量池数量紧跟在次版本号的后面，占 2 字节。常量池数组则紧跟在常量池数量之后。

常量池数组，顾名思义，就是一个类似数组的结构。这个数组固化在字节码文件中，由多个元素组成。与一般数组概念不同的是，常量池数组中不同的元素的类型、结构都是不同的，长度也是不同的，但是每一种元素的第一个数据都是一个 u1 类型，该字节是标志位，占 1 个字节（如图 4.4 所示）。JVM 解析常量池时，根据这个 u1 类型来获取该元素的具体类型。常量池的结构组成如图 4.4 所示。

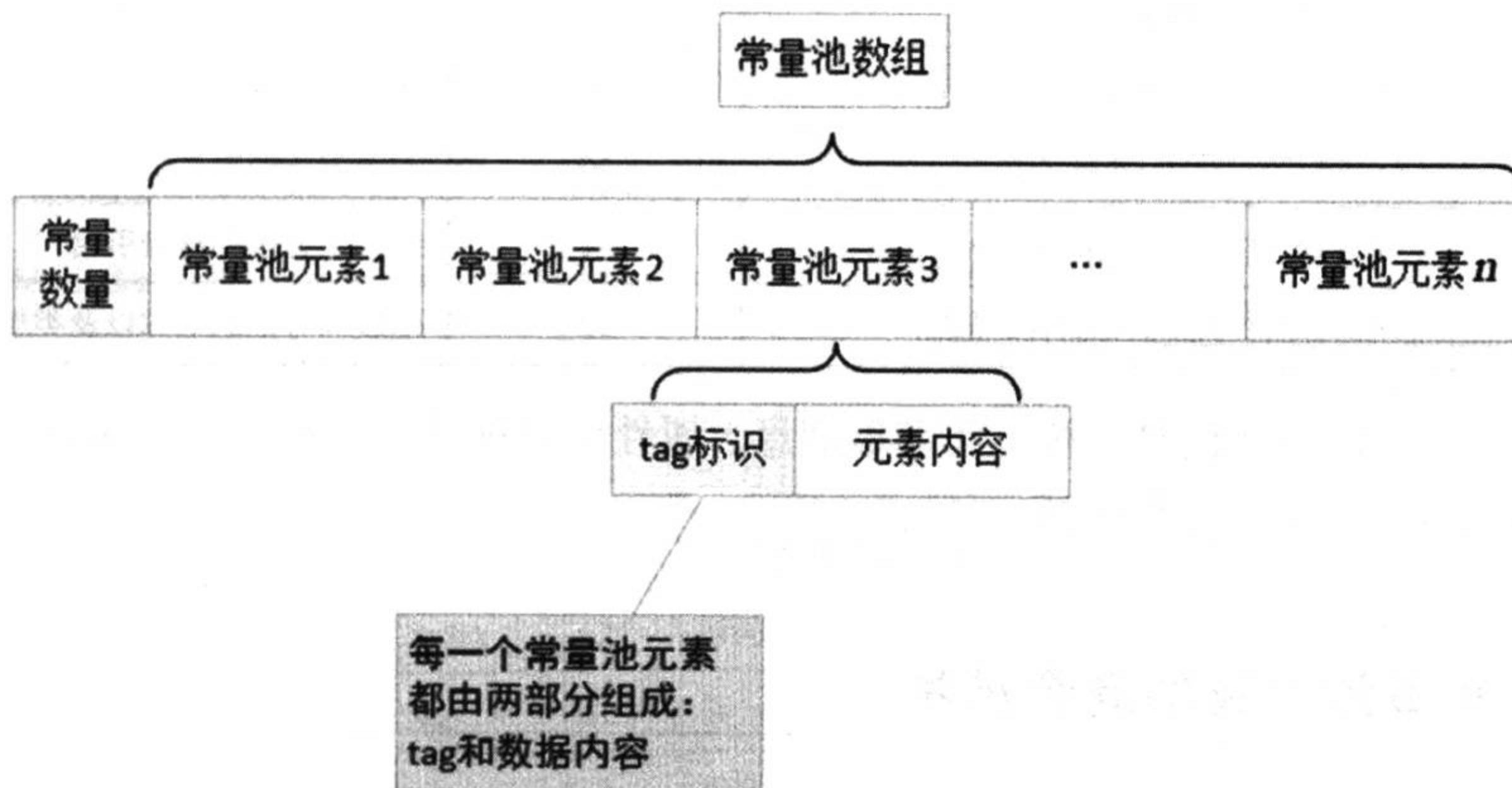


图 4.4 Java 字节码中常量池的组成结构

使用结构化的方式来描述常量池数组的编排，可以如下描述：

tag1 元素内容 1 **tag2** 元素内容 2 **tag3** 元素内容 3 ... **tagn** 元素内容 *n*

这里为了描述，在 tag 与元素内容之间添加了空格，但实际的 class 二进制文件中，这些 tag 与元素内容之间绝没有任何空格信息，也没有任何其他的多余信息，tag 后面紧跟着元素内容，第一个元素内容结束了，紧接着就是第二个元素的 tag 信息，由此可见，整个字节码文件的格式设计都是非常紧凑的。

4.3.2 JVM 所定义的 11 种常量

常量池元素中的不同元素结构与类型都是不同的，正因如此，JVM 只能定义有限的元素类型，并针对有限的类型进行专门解析。JVM 一共定义了 11 种常量，如表 4.1 所示。

表 4.1 JVM 常量池元素一览表

编 号	常量池元素名称	tag 位标识	含 义
1	CONSTANT_Utf8_info	1	UTF-8 编码的字符串
2	CONSTANT_Integer_info	3	整型字面量
3	CONSTANT_Float_info	4	浮点型字面量
4	CONSTANT_Long_info	5	长整型字面量
5	CONSTANT_Double_info	6	双精度字面量
6	CONSTANT_Class_info	7	类或接口的符号引用
7	CONSTANT_String_info	8	字符串类型的字面量
8	CONSTANT_Fieldref_info	9	字段的符号引用
9	CONSTANT_Methodref_info	10	类中方法的符号引用
10	CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
11	CONSTANT_NameAndType_info	12	字段和方法的名称以及类型的符号引用

可以看到，类的方法信息、接口和继承信息、属性信息都是定义在 NamedAndType_Info 中的，关于该结构，下文会详细讲解。

4.3.3 常量池元素的复合结构

常量池数组中的每一种元素的内容都是复合数据结构的，下面分别给出 JVM 所定义的常量池中每一种元素的具体结构（如表 4.2 所示）。

表 4.2 常量池元素结构

类型/含义	结构	类型	描述
CONSTANT_Utf8_info UTF-8 编码的字符串	tag	u1	值为 1
	length	u2	UTF-8 缩略编码字符串占用字节数
	bytes	u1	长度为 length 的 UTF-8 编码字符串
CONSTANT_Integer_info 整型字面量	tag	u1	值为 3
	bytes	u4	按照高位在前储存的 int 值
CONSTANT_Float_info 浮点型字面量	tag	u1	值为 4
	bytes	u4	按照高位在前储存的 float 值
CONSTANT_Long_info 长整型字面量	tag	u1	值为 5
	bytes	u8	按照高位在前储存的 long 值
CONSTANT_Double_info 双精度浮点型字面量	tag	u1	值为 6
	bytes	u8	按照高位在前储存的 double 值
CONSTANT_Class_info 类或接口的符号引用	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info 字符串类型字面量	tag	u1	值为 8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info 字段的符号引用	tag	u1	值为 9
	index	u2	指向声明字段的类或接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_Methodref_info 类中方法的符号引用	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_InterfaceMethodref_info 接口中方法的符号引用	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_NameAndType_info 字段或方法的部分符号引用	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引

该表中 tag 值为 1 的常量池元素 CONSTANT_Utf8_info，其组成结构分为 3 部分，分别是：tag、length 和 bytes，其中 tag 和 length 的长度分别是 u1、u2，即分别占 1 字节和 2 字节。而 bytes 则是字符串的具体内容，其长度是 length 字节。在字节码文件中，该常量池元素最终所占的字节数是：

$$1 + 2 + \text{length}$$

其他类型的常量池元素的组成结构类似，这里不一一分析，下文在实例解析常量池组成结构时会再次逐个进行详细解析。

4.3.4 常量池的结束位置

相信有不少读者读到这里，可能潜意识里会突然蹦出这么一个问题：整个字节码文件由多个部分构成，常量池数组只是其中一块，JVM 在解析字节码文件时，一定需要分别读取各个部分的字节流，其中也包括常量池数组。但是常量池中有部分元素的值是 bytes 数组，其长度是随机变化的，那么 JVM 在解析时，是如何知道整个常量池的信息解析到什么位置结束呢？其实经过分析不难发现，首先，class 文件给出了常量池的总数；其次，凡是碰到有 bytes 数组的常量池元素，class 文件在常量池的每一个元素之前都会专门划分出 2 字节用于描述该常量池元素内容所占的字节长度，这样一来，常量池中每一个元素的长度是确定的，而常量池的总数也是确定的，JVM 据此便可以从 class 文件中准确地计算出常量池结构体的末端位置（起始位置不用计算，是定死的，从第 9 字节开始，前面 8 字节分别是魔数和版本号）。

对上面这些概念有了形象上的认识后，下面让我们一起继续以 Test.class 这个字节码文件为例，详细分析其中的常量池信息。

4.3.5 常量池元素总数量

前面 8 字节用于描述魔数和版本号，从第 9 字节开始的一大段字节流都用于描述常量池数组信息。其中，第 9 和第 10 字节用于描述常量池元素的总数量。如图 4.5 所示。

第 9 和第 10 字节所保存的常量池数组大小是 0x33，换算成十进制是 51，说明该字节码文件中一共包含 51 个常量池元素。JVM 规定，不使用第 0 个元素，因此实际上一共有 50 个常量池元素（下文在解析源码时，会看到源码中的确是从第 1 个元素开始解析的，而不是从第 0 个）。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 62 61 65	61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3E 61 65	6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....
000000a0	00 07 76 61 6C 75 65 4F	66 01 00 16 28 49 29 4C	..valueOf...(I)L
000000b0	6A 61 76 61 2F 6C 61 6E	67 2F 49 6E 74 65 67 65	java/lang/Intege

图 4.5 常量池长度

4.3.6 第一个常量池元素

常量池数量之后（即从第 11 字节开始），就是常量池数组。每一个常量池元素都以 tag 位标开始，tag 位标都只占 1 字节长度。如图 4.6 所示。第 11 字节对应的值是 7，对照上文所给的常量池 11 种元素的复合结构可知，tag 位标为 7 代表的是 CONSTANT_Class_info，即类或接口的符号引用，这种类型的元素的结构组成如下：

- ◎ tag 位标 占 1 字节
- ◎ index 占 2 字节

既然 tag 位标已经占据了字节码文件的第 11 字节，则接下来的第 12 和 13 字节将合起来表示 index。如图 4.6 所示，这两个字节对应的值为 2。

从第 11 字节开始，到第 13 字节为止，一个常量池元素就被描述完成。紧接着开始描述第 2 个常量池元素。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 62 61 65	61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3E 61 65	6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....
000000a0	00 07 76 61 6C 75 65 4F	66 01 00 16 28 49 29 4C	..valueOf...(I)L
000000b0	6A 61 76 61 2F 6C 61 6E	67 2F 49 6E 74 65 67 65	java/lang/Intege
000000c0	72 3B 09 00 01 00 15 0C	00 07 00 08 01 00 0F 4C	r;.....L

图 4.6 常量池第一个元素的 tag 和 index

4.3.7 第二个常量池元素

第一个常量池元素后面紧跟着的是第二个常量池元素。其第一字节是 01，表示这是一个 UTF8 编码的字符串，其结构是：

- ◎ tag 位，占 1 字节
- ◎ length，占 2 字节
- ◎ bytes，占 length 字节

如图 4.7 所示，tag 位后面的 2 字节的值是 4，表示 bytes 占 4 字节，其值为 0x54657374，其中每两字节正好代表一个字符，对应的字符串是 Test。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.... ...
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Testjava/l...
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a...
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/...
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S...
00000060	74 72 69 6E 67 3B 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la...
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer

图 4.7 常量池第二个元素

4.3.8 父类常量

刚才的第 1 与第 2 两个常量用于描述 Java 类型信息，接下来的第 3 与第 4 两个常量则用于描述父类信息。由于 Test.java 类并没有显式继承任何类，因此编译后处理成默认继承，即父类是 Java.lang.Object。

父类常量的 tag 位也是 07（如图 4.8 所示），其类名是 java/lang/Object（如图 4.9 所示）。下面分别给出第 3 和第 4 这两个常量在文件中的内容。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Testjava/l...
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a...
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/...
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S...
00000060	74 72 69 6E 67 3B 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la...
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer

图 4.8 第 3 个常量——父类信息

00000000	CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00	ang/Object....a..
00000030	01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.9 第4个常量——父类接口名称

图 4.9 显示，字符串的 length 值为 0x10，即 16，于是其后面的 16 字节都是 bytes。由此可以进一步验证，字符串常量的结构由 tag、length 和 bytes 组成，bytes 的长度由 length 指定。

4.3.9 变量型常量池元素

常量池中前面 4 个元素主要用于描述 Java 类自身的名称和其父类名称，接下来的字节码流则开始描述类中的变量信息，这些变量既包括类的成员变量，也包括类变量（即静态变量）信息。

首先看类成员变量 a 的信息，如图 4.10 所示。

00000000	CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00	ang/Object....a..
00000030	01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.10 变量 a 所对应的常量池信息

图 4.10 中所选中的 8 字节，一共包含两个常量池元素信息，这两个常量池元素的类型都是字符串，因为其 tag 位都是 1。第一个字符串常量的 length 是 1，其值（即 bytes）是 0x61，正好对应 UTF-8 编码的字符 a。第二个字符串常量的 length 也是 1，其值是 0x49，正好对应 UTF-8 编码的字符 I。在 JVM 规范中，若变量的类型是 I，则表示该变量的实际类型是 int。这与上文对变量 a 的定义一致。

接着看类变量 si 的定义，如图 4.11 所示。

00000000	CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object....a..
00000030	01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74 65 67 65 72 3E 01 00 01 lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060	74 72 69 6E 67 3E 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit
00000070	3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080	0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer.....

图 4.11 变量 si 所对应的常量池信息

图 4.11 中所选中的 27 字节，一共描述了两个常量池元素，这两个常量池元素的类型也都是字符串。第一个字符串的 length 为 2，其值是 0x7369，对应 utf-8 编码的字符串 si。第二个字符串的 length 为 0x13，即 19，其值是 0x4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3E，这一串值是 ASCII 字符，每 2 个十六进制数对应一个 ASCII 字符，这些数字连起来就对应一个字符串，所对应的字符串是 Ljava/lang/Integer;。

这两个常量池元素合起来，描述了 Test 类中的 static Integer si 这样的类变量。

接着看类成员变量 s 的定义，如图 4.12 所示。

00000000	CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object....a..
00000030	01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74 65 67 65 72 3E 01 00 01 lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060	74 72 69 6E 67 3E 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit
00000070	3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080	0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer.....

图 4.12 变量 s 所对应的常量池信息

图 4.12 中选中的 25 字节，一共描述了两个常量池元素，这两个常量池元素的类型也都是字符串。第一个字符串的 length 为 1，其值是 0x73，对应 UTF-8 编码的字符 s。第二个字符串的 length 为 0x12，即 18，其值是 0x4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B，对应 UTF-8 编码的字符串 Ljava/lang/String;。

这两个常量池元素合起来描述了 Test 类中的 s 字符串变量。

4.4 访问标识与继承信息

上面解析完了常量池的所有元素，可能读者已经陷入常量池的分析之中，但是别忘了，一个.class字节码文件中共有10个组成部分（见上文），常量池只不过是其中的一个组成部分，因此现在需要把目光转移出来，继续回到.class字节码文件后续的组成部分的分析。

4.4.1 access_flags

在字节码文件中，常量池数组之后紧跟着的是access_flags结构，该结构类型是u2，占2字节。

access_flags代表访问标志位，该标志用于标注类或接口层次的访问信息，如当前Class是类还是接口，是否定义为public类型，是否定义为abstract类型等。

对于本文中的测试用例Test.class，其access_flags信息如图4.13所示。

The screenshot shows a hex dump of a Java class file. The access_flags field is located at offset 0x000001e0. The value 0x0021 is highlighted with a red box. The surrounding bytes are mostly zeros, with some non-zero values like 0x00, 0x01, 0x02, etc.

00000180	2F 49 6E 74 65 67 65 72 3B 29 56 01 00 04 61 72	/Integer;)V...ar
00000190	67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67	gs...[Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A	/String;...test.
000001b0	00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56/.0...intV
000001c0	61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75	alue...()I...Sou
000001d0	72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A	rceFile...Test.]
000001e0	61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00	ava!.....
000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8}.....
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06

图4.13 access属性

由图4.13可知，Test.class的access_flags的值是0x0021，这代表什么含义呢？根据JVM的规范，access_flags的可选项值如表4.3所示。

表4.3 access_flags可选项

标志名称	标志值	含义
ACC_PUBLIC	0x0001	是否为public类型
ACC_FINAL	0x0010	是否被声明为final，只有类可设置
ACC_SUPER	0x0020	是否允许使用invokespecial字节码指令，JDK1.2以后编译出来的类这个标志为真
ACC_INTERFACE	0x0200	标识这是一个接口

续表

标志名称	标志值	含义
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口和抽象类，此标志为真，其他类为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举

由于 Test.class 中的 access_flags=0x21，因此该类的访问标识既包含 ACC_PUBLIC(0x0001)，也包含 ACC_SUPER (0x0020)。其中，自 JDK1.2 以后，类被编译出来的 invokespecial 字节码指令是否允许使用的选项都是真，因此 access_flags 的值都会带有 ACC_SUPER 标识位。

4.4.2 this_class

在字节码文件中，紧跟着 access_flags 访问标识之后的是 this_class 结构，该结构类型是 u2，占 2 字节。this_class 记录当前类的全限定名（包名+类名），其值指向常量池中对应的索引值。

Test.class 中的 this_class 信息如图 4.14 所示。

图 4.14 this_class 的字节码

由图 4.14 可知，Test.class 的 this_class 的值为 1，说明该值对应 1 号常量池元素。上文使用 javap -verbose 命令将 Test.class 的所有常量池连同其编号一起打印了出来，根据打印信息可知，1 号常量池元素的信息如下：

```
const #1 = class           #2;      // Test
const #2 = Asciz          Test;
```

由此可知，this_class 的确是 Test，类的全限定名就是 Test。

4.4.3 super_class

在字节码文件中，紧跟着 this_class 访问标识之后的是 super_class 结构，该结构类型是 u2，占 2 字节。super_class 记录当前类的父类全限定名，其值指向常量池中对应的索引值。

Test.class 中的 super_class 信息如图 4.15 所示。

00000190	67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67	gs...[Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A	/String;...test.
000001b0	00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56/.0...intV
000001c0	61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75	alue...()I...Sou
000001d0	72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A	rceFile...Test.)
000001e0	61 76 61 00 21 00 01 00 03 00 00 03 00 01 00	ava.!....
000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8).....
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06

图 4.15 super_class 的字节码

由图 4.15 可知，Test.class 的 super_class 的值为 3，说明该值对应 3 号常量池元素。上文使用 javap -verbose 命令将 Test.class 的所有常量池连同其编号一起打印了出来，根据打印信息可知，3 号常量池元素的信息如下：

```
const #3 = class           #4;      // java/lang/Object
const #4 = Asciz           java/lang/Object;
```

由于 Test.class 并没有显式继承任何基类，因此编译时便让其默认继承 java.lang.Object。这与字节码中的 super_class 值是一致的。

4.4.4 interface

1. interfaces_count

在字节码文件中，紧跟着 super_class 访问标识之后的是 interfaces_count 结构，该结构类型是 u2，占 2 字节。interfaces_count 结构记录当前类所实现的接口数量。

Test.class 中的 interfaces_count 结构信息如图 4.16 所示。

00000190	67 73 01 00 13 5B 4C 6A	61 76 61 2F 6C 61 6E 67	gs...[Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B	01 00 04 74 65 73 74 0A	/String;...test.
000001b0	00 0F 00 2E 0C 00 2F 00	30 01 00 08 69 6E 74 56/.0...intV
000001c0	61 6C 75 65 01 00 03 28	29 49 01 00 0A 53 6F 75	alue...()I...Sou
000001d0	72 63 65 46 69 6C 65 01	00 09 54 65 73 74 2E 6A	rceFile...Test.j
000001e0	61 76 61 00 21 00 01 00	03 00 00 00 00 03 00 01 00	ava!.....
000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06

图 4.16 字节码中的接口信息

由图 4.16 可知，Test.class 的 interfaces_count 的值为 0，说明 Test.class 并没有实现任何接口。

2. interfaces[interfaces_count]

interfaces 表示接口索引集合，是一组 u2 类型数据的集合，该结构描述当前类实现了哪些接口，这些被实现的接口将按 implements 语句（如果该类本身为接口，则为 extends 语句）后的接口顺序从左至右排列在接口的索引集合中。

由于 Test.class 的 interfaces_count 值为 0，因此字节码文件中并没有 interfaces 信息。

4.5 字段信息

4.5.1 fields_count

在字节码文件中，接口区之后紧接着是 fields_count 结构。该结构类型是 u2，占 2 字节。该值记录当前类中所定义的变量总数量，包括类成员变量和类变量（即静态变量）。

Test.class 中的 fields_count 信息如图 4.17 所示。

00000190	67 73 01 00 13 5B 4C 6A	61 76 61 2F 6C 61 6E 67	gs...[Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B	01 00 04 74 65 73 74 0A	/String;...test.
000001b0	00 0F 00 2E 0C 00 2F 00	30 01 00 08 69 6E 74 56/.0...intV
000001c0	61 6C 75 65 01 00 03 28	29 49 01 00 0A 53 6F 75	alue...()I...Sou
000001d0	72 63 65 46 69 6C 65 01	00 09 54 65 73 74 2E 6A	rceFile...Test.j
000001e0	61 76 61 00 21 00 01 00	03 00 00 00 00 03 00 01 00	ava!.....
000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06

图 4.17 字节码文件中的字段数量

由图4.17可知，Test.class类中一共包含3个变量。从Test源文件也可以看出，该类的确包含3个变量，分别是a、si和s。

4.5.2 field_info fields[fields_count]

在字节码文件中，紧跟着fields_count之后的是fields结构，该结构长度不确定，不同的变量类型所占长度是不同的。fields记录类中所定义的各个变量的详细信息，包括变量名、变量类型、访问标识、属性等。

1. fields结构组成格式

要分析fields结构信息，首先需要清楚该结构的数据组成格式，其格式如表4.4所示。

表4.4 fields结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

表4.3中各个组成元素的说明如下：

- ◎ access_flags，标识变量的访问标识，该值是可选的，由JVM规范规定。
- ◎ name_index，表示变量的简单名称引用，占2字节，其值指向常量池的索引。
- ◎ descriptor_index，表示变量的类型信息引用，占2字节，其值指向常量池的索引。

fields结构体实际上是一个数组，数组中的每一个元素的结构都如表4.3所示，即每一个元素都包含访问标识、名称索引、描述信息索引、属性数量和属性信息。其中，如果属性数量为0，则没有属性信息。由于访问标识、名称、描述信息、属性数量的字节长度是确定的，因此JVM可以在解析过程中计算出fields结构所占的全部字节数。

变量的access_flags有如表4.5所示的可选项。

表 4.5 access_flags 的可选项

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为 public
ACC_PRIVATE	0x0002	字段是否为 private
ACC_PROTECTED	0x0004	字段是否为 protected
ACC_STATIC	0x0008	字段是否为 static
ACC_FINAL	0x0010	字段是否为 final
ACC_VOLATILE	0x0040	字段是否为 volatile
ACC_TRANSIENT	0x0080	字段是否为 transient
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生
ACC_ENUM	0x4000	字段是否为 enum

其中，ACC_PUBLIC、ACC_PRIVATE 和 ACC_PROTECTED 这 3 个标志只能选择一个，接口中的字段必须有 ACC_PUBLIC、ACC_STATIC 和 ACC_FINAL 标志，class 文件对此并无规定，这些都是 Java 语言所要求的。

2. 第 1 个变量 a

分析完理论，下面来看看 Test.class 字节码中的变量信息实际都指的是什么。第一个 field 紧跟在 fields_count 这个结构体之后，如图 4.18 所示。

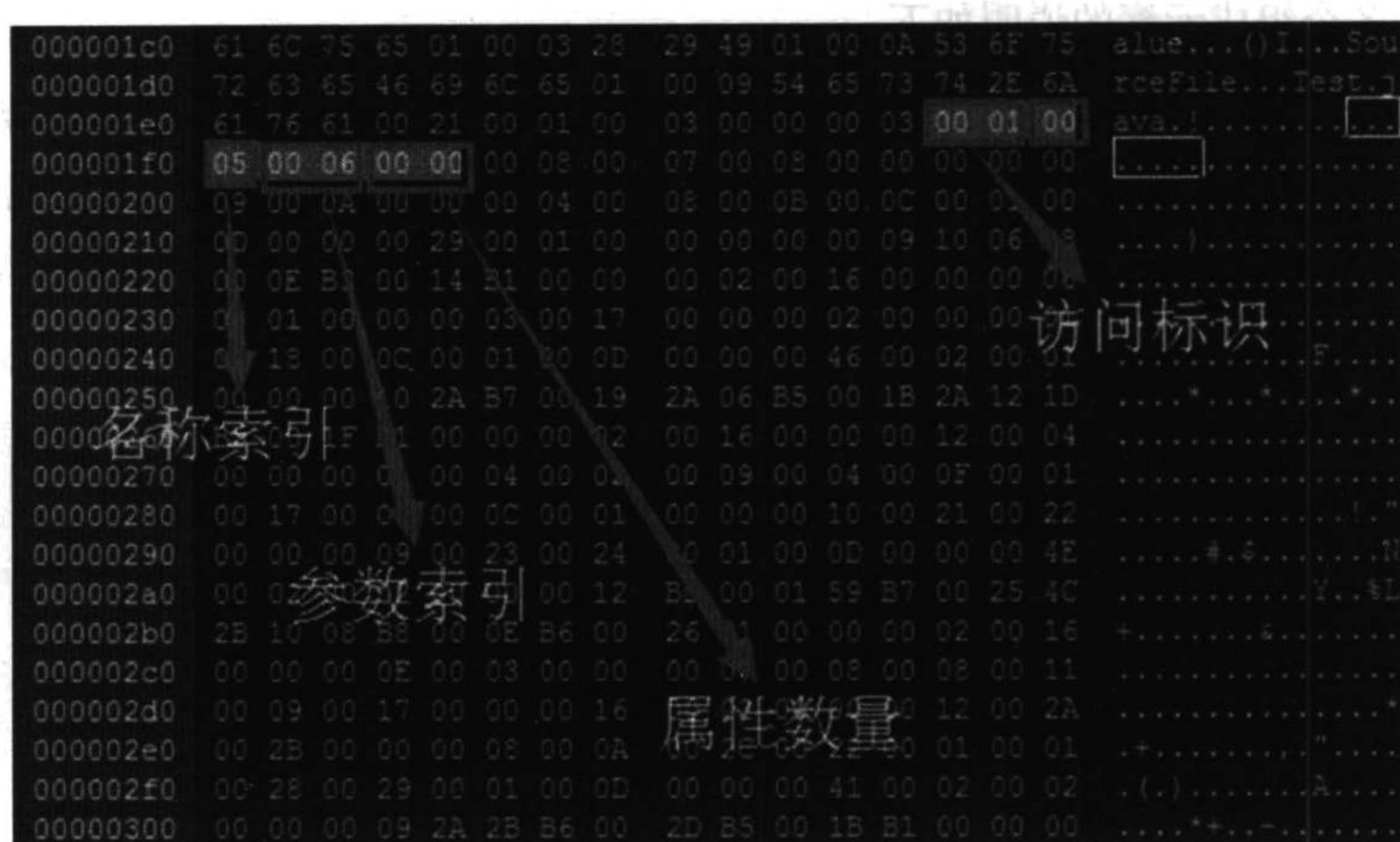


图 4.18 变量 a 在字节码中的描述信息

由图 4.18 可知，第一个变量的访问标识是 1，由参考变量访问标识选项表可知，1 表示 ACC_PUBLIC。第一个变量的名称索引是 5，类型索引是 6，按照上文使用 javap -verbose 命令所打印的字节码常量池信息可知，常量池中第 5 和第 6 个元素的信息如下：

```
const #5 = Asciz      a;
const #6 = Asciz      I;
```

由此可知，当前描述的变量是 a，其数据类型是 int。

根据图 4.18 还可知道，变量 a 的属性数量是 0，因为没有属性，所以字段描述结构中最后的元素 attributes 也就不存在。

这里需要注意描述信息索引，其指向常量池中的 6 号元素，6 号元素的值是 I，这里的 I 代表什么意思呢？在 JVM 规范中，每个变量/字段都有描述信息，描述信息主要描述字段的数据类型、方法的参数列表（包括数量、类型和顺序）和返回值。根据描述符规则，基本数据类型和代表无返回值的 void 类型都用一个大写字符表示，而对象类型则用字符 L 加对象全限定名表示。为了压缩字节码文件的体积（字节码文件最终也会占用服务器硬盘资源和内存资源），对于基本数据类型，JVM 都仅使用一个大写字母来标识。表 4.6 所示是各个基本数据类型所对应的标识符。

表 4.6 标识字符与基本数据类型对应表

标识字符	含义
B	基本类型 byte
C	基本类型 char
D	基本类型 double
F	基本类型 float
I	基本类型 int
J	基本类型 long
S	基本类型 short
Z	基本类型 boolean
V	特殊类型 void
L	对象类型，如 Ljava/lang/Object

对于数组类型，每一维将使用一个前置的 “[” 字符来描述，如 “int[]” 将被记录为 “[I” ，“String[][]” 将被记录为 “[Ljava/lang/String;]”。

用描述符描述方法时，按照先参数列表，后返回值的顺序描述，参数列表按照参数的严格顺序放在一组 “()” 之内，如方法 “String getAll(int id, String name)” 的描述符为

“(I,Ljava/lang/String;)Ljava/lang/String;”。

3. 变量 si 和 s

由于变量 a 的属性数量是 0，字节码文件中不包含 attributes 信息，因此第一个变量只占 8 字节，分别是访问标识、变量名引用、描述信息引用和属性数量。由于字节码文件标识一共包含 3 个变量，因此描述变量 a 的字节码流的后面的字节码流将继续描述另外两个变量。

Test.class 类的另外两个变量的字节码内容如图 4.19 所示。

000001d0	72 63 65 46 69 6C 65 01	00 09 54 65 73 74 2E 6A	rceFile...Test.j
000001e0	61 76 61 00 21 00 01 00	03 00 00 00 03 00 01 00	ava.!.....
000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 30 00 00
00000200	19 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 04 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01 F...
00000250	00 00 00 01 00 19 2A 06	B5 00 1B 0A 12 1D *.*...
00000260	B5 00 1F B1 00 00 00 02	00 16 00 00 00 12 00 04
00000270	00 00 00 01 00 04 00 02	00 09 00 04 00 00 00 01
00000280	00 17 00 00 00 0C 00 01	00 00 00 10 00 21 00 22
00000290	00 00 00 09 00 23 00 24	00 01 00 0D 00 00 00 4E #.S.....N

图 4.19 变量 si 和 s 在字节码中的描述信息

从图 4.19 可知，这两个变量在字节码文件中也各占 8 字节，因为其 attributes_count 都是 0。变量 si 的 access_flags 是 8，表示这是一个带有 static 修饰符的变量，而变量 s 的 access_flags 是 0，表示该变量没有任何访问修饰符，对照源程序，的确是这样。

两个变量名称分别引用常量池中的 7 号元素和 9 号元素，对照上文使用 javap -verbose 命令打印的常量池信息，可知这两个变量名分别是 si 和 s。

两个变量描述信息分别引用常量池中的 8 号和 10 号元素，对照打印出来的常量池信息可知，其变量类型分别是 Ljava/lang/Integer 和 Ljava/lang/String。由此也可知，对于引用类型的变量，字节码文件描述其变量类型的格式是“L+类全限定名”。

至此，字节码中对 fields 的描述全部结束。不过为了追求简单易懂，本示例中并没有对类变量添加任何属性，因此本示例所生成的字节码的字段信息，都没有 attribute 信息。不过各位道友可以自行做试验进行更加深入的研究。

4.6 方法信息

4.6.1 methods_count

在字节码文件中，紧跟着变量描述结构 `fields` 后面的是 `methods_count` 结构，该结构类型是 `u2`，占 2 字节。该结构描述类中一共包含多少个方法。

`Test.class` 字节码文件中该结构信息如图 4.20 所示。

```

000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.] 
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.!.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 ..... 
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....□.....
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 .....).....
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 ..... 
00000230 00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 ..... 

```

图 4.20 字节码中的方法数量

由图 4.20 可知，其值为 4，即 `Test` 类中一共有 4 个方法。可能很多人对此会有疑惑，在 `Test` 源程序中明明只定义了两个方法，为什么字节码文件中却显示有 4 个呢？这是因为在编译期间，编译器会自动为一个类增加 `void <clinit>()` 这样一个方法，其方法名就是“`<clinit>`”，返回值为 `void`。该方法的作用主要是执行类的初始化，源程序中的所有 `static` 类型的变量都会在这个方法中完成初始化，全部被 `static {}` 所包围的程序都在这个方法中执行（这个后面在讲解 JVM 源码时会详细说明）。同时，在源代码中，并没有为 `Test` 类定义构造函数，因此编译器会自动为该类添加一个默认的构造函数（这一点大家应该都知道）。因此，字节码文件会显示 `Test` 类中一共包含 4 个方法。

4.6.2 method_info methods[methods_count]

紧跟在 `methods_count` 后面的是 `methods` 结构，这是一个数组，每一个方法的全部细节都包含在里面，包括代码指令。

1. methods 结构组成格式

要分析 `methods` 结构信息，首先需要清楚该结构的数据组成格式，其格式如表 4.7（方法表结构和字段表结构一样）所示。

表 4.7 methods 结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

由表 4.7 可知，方法各个数据项的含义非常相似，仅在访问标志位和属性表集合的可选项上有略微不同。这些字段的含义与上文给出的 fields 结构的字段含义基本相同，因此这里不作具体说明。

其中，JVM 规范为 access_flags 规定了一组可选项值，如表 4.8 所示。

表 4.8 access_flags 可选项值

标 志 名 称	标 志 值	含 义
ACC_PUBLIC	0x0001	字段是否为 public
ACC_PRIVATE	0x0002	字段是否为 private
ACC_PROTECTED	0x0004	字段是否为 protected
ACC_STATIC	0x0008	字段是否为 static
ACC_FINAL	0x0010	字段是否为 final
ACC_SYNCHRONIZED	0x0020	字段是否为 synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	字段是否为 native
ACC_ABSTRACT	0x0400	字段是否为 abstract
ACC_STRICTFP	0x0800	字段是否为 strictfp
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生

由于 ACC_VOLATILE 标志和 ACC_TRANSIENT 标志不能修饰方法，所以 access_flags 中不包含这两项，同时增加 ACC_SYNCHRONIZED 标志、ACC_NATIVE 标志、ACC_STRICTFP 标志和 ACC_ABSTRACT 标志。

2. 第一个方法 void <clinit>()

上面了解了方法描述的信息结构，下面来实际看看 Test.class 字节码文件中的第一个方法究竟是如何描述的。紧跟在 methods_count 后面的就是第一个方法的信息，如图 4.21 所示。

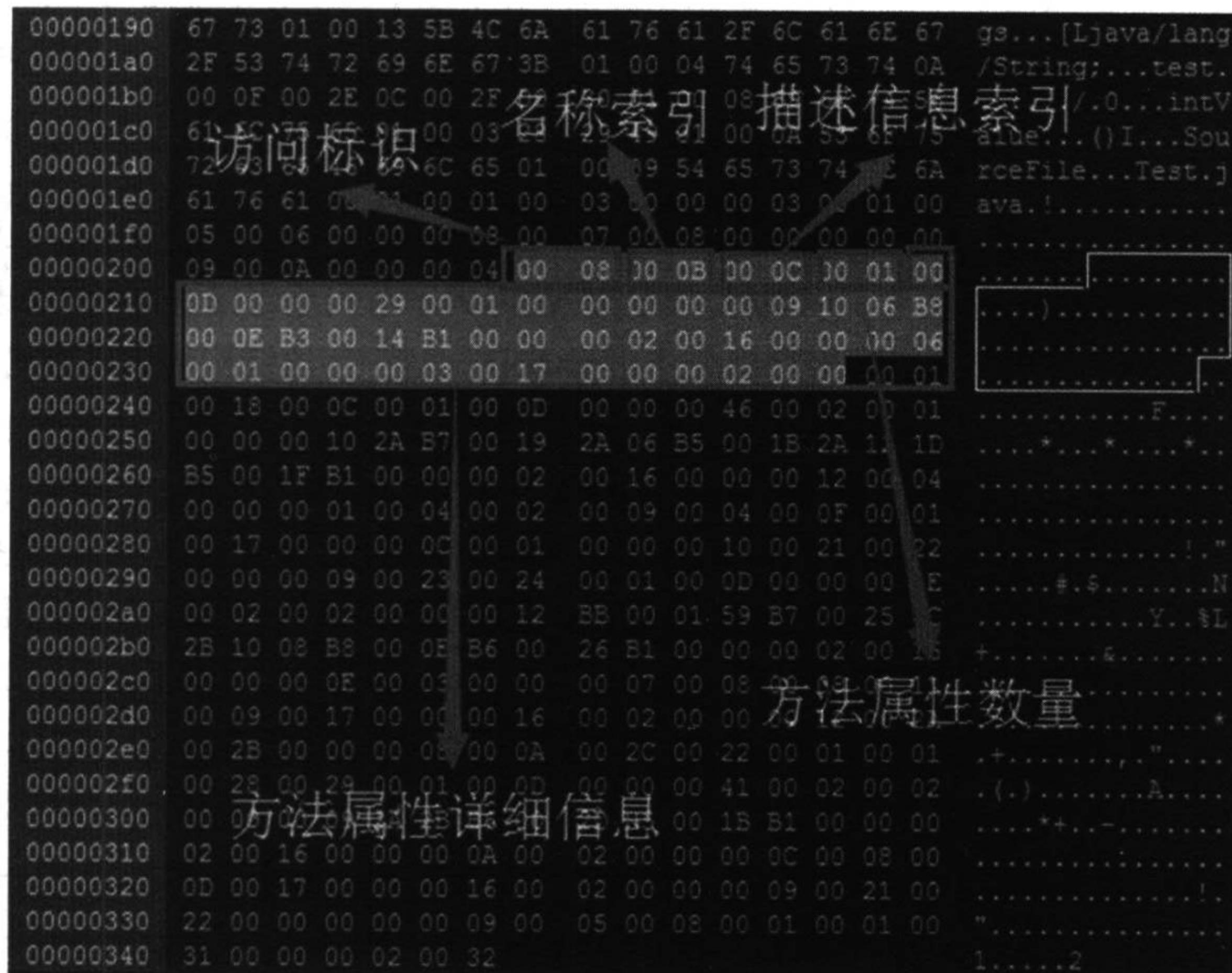


图 4.21 void <clinit>()方法对应的字节码

由图 4.21 可以看出，字节码文件对方法的描述比较复杂，不像前面对魔数、版本号、常量池等信息的描述那么简单。但无论多么复杂的描述，总是遵循其内在的结构逻辑，只要按照 JVM 的规范按图索骥，总是能够分析清楚字节码所要表达的含义。

按照 fields 的结构组成格式，前 2 字节描述 access_flags，即访问标识，由图 4.21 可知其值为 0x0008，对照上文所给出的方法访问标识可选项值的表可知，该值标识该方法的修饰符是 AC_STATIC，也即这是一个 static 类型的静态方法。

接下来的 2 字节描述 name_index，该字段描述的是方法名，其值指向常量池中对应的元素编号。由图 4.21 可知，其值是 0x000B，指向常量池中第 11 号元素，根据上文使用 javap -verbose 命令所打印出的常量池信息可知，常量池中第 11 号元素是<clinit>，即当前所描述的方法名是“<clinit>”，这在上文提到过，该方法是 Java 编译器在编译期间动态添加的类初始化的方法，而非在源程序中定义的。

接下来的 2 字节描述 `descriptor_index`，该字段描述的是方法的入参和出参信息，其值指向常量池中对应的元素编号。由图 4.21 可知，其值是 0x000C，指向常量池中第 12 号元素，根据上文打印的常量池表可知，常量池中第 12 号元素是(0)V，这表示当前方法没有入参（因为是空括号），并且方法的返回值类型是 void（V 代表 void）。这里要注意，按照 JVM 的规范，描述符对入参将严格按照源程序中所定义的参数列表顺序，从左到右依次放入“0”内，如方法“String getAll(int id, String name)” 的描述符为“(I,Ljava/lang/String;)Ljava/lang/String;”。

根据这些信息可知，字节码所描述的第一个方法是 `void <clinit>()`。

接下来的 2 字节描述方法所包含的属性的总数量 `attributes_count`，由图 4.21 可知其值为 0x0001，表示当前方法一共包含 1 个属性。该字段后面的字节码流将描述详细的属性信息。在分析字节码流中的属性信息之前，有必要先了解 `attributes` 这一字段结构的组成格式。

3. 9 大属性表集合

在 class 文件中，属性表、方法表中都可以包含自己的属性表集合，用于描述某些场景的专有信息。本部分内容与 JVM 规范官方文档保持一致，因此这里仅列出大概的概括信息，详细信息可以去阅读官方文档。

与 class 文件中其他数据项对长度、顺序、格式的严格要求不同，属性表集合不要求其中包含的属性表具有严格的顺序，并且只要属性的名称不与已有的属性名称重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息。虚拟机在运行时会忽略不能识别的属性。为了能正确解析 class 文件，虚拟机规范中预定义了虚拟机实现必须能够识别的 9 项属性，如表 4.9 所示。

表 4.9 9 大属性

属性名称	使用位置	含 义
Code	方法表	Java 代码编译成的字节码指令
ConstantValue	字段表	final 关键字定义的常量值
Deprecated	类文件、字段表、方法表	被声明为 <code>deprecated</code> 的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTable	Code 属性	Java 源码的行号与字节码指令的对应关系
LocalVariableTable	Code 属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类文件、方法表、字段表	标识方法或字段是由编译器自动生成的

这 9 种属性中的每一种属性又都是一个复合结构，均有各自的表结构。这 9 种表结构有一个共同的特点，即均由一个 u2 类型的属性名称开始，可以通过这个属性名称来判断属性的类型。该 u2 类型的属性名称指向常量池中对应的元素。

下面描述这 9 种属性具体的复合组成结构。

1) Code 属性

Java 程序方法体中的代码经过 javac 编译器处理后，最终变为字节码指令存储在 Code 属性中。当然不是所有的方法都必须有这个属性（接口中的方法或抽象方法就不存在 Code 属性），Code 属性结构如表 4.10 所示。

表 4.10 Code 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

Code 属性表中相关字段的含义如下：

- ◎ max_stack，操作数栈深度最大值，在方法执行的任何时刻，操作数栈深度都不会超过这个值。虚拟机运行时根据这个值来分配栈帧的操作数栈深度。
- ◎ max_locals，局部变量表所需存储空间，单位为 Slot。并不是所有局部变量占用的 Slot 之和，当一个局部变量的生命周期结束后，其所占用的 Slot 将分配给其他依然存活的局部变量使用，按此方式计算出方法运行时局部变量表所需的存储空间。
- ◎ code_length 和 code，用来存放 Java 源程序经编译后生成的字节码指令。code_length 代表字节码长度，code 是用于存储字节码指令的一系列字节流。

每一个指令都是一个 u1 类型的单字节，当虚拟机读到 code 中的一个字节码（一个字节能表示 256 种指令，Java 虚拟机规范定义了其中约 200 个编码对应的指令）时，就可以判断出该

字节码代表的指令，指令后面是否带有参数，参数该如何解释，虽然 code_length 占 4 字节，但是 Java 虚拟机规范限制一个方法不能超过 65 535 条字节码指令，如果超过，Javac 将拒绝编译。

2) ConstantValue 属性

ConstantValue 属性通知虚拟机自动为静态变量赋值，只有被 static 关键字修饰的变量（类变量）才可以使用这项属性。其结构如表 4.11 所示。

表 4.11 ConstantValue 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

可以看出，ConstantValue 属性是一个定长属性，其中 attribute_length 的值固定为 0x00000002，constantvalue_index 为一常量池字面量类型常量索引（class 文件格式的常量类型中只有与基本类型和字符串类型相对应的字面量常量，所以 ConstantValue 属性只支持基本类型和字符串类型）。

对非 static 类型变量（实例变量，如 int a = 123;）的赋值是在实例构造函数<init>中进行的。

对类变量（如 static int a = 123;）的赋值有两种选择，在类构造函数<clinit>方法中或使用 ConstantValue 属性。当前 javac 编译器的选择是，如果变量同时被 static 和 final 修饰（虚拟机规范只要求有 ConstantValue 属性的字段必须设置 ACC_STATIC 标志，对 final 关键字的要求是 javac 编译器自己加入的要求），并且该变量的数据类型为基本类型或字符串类型，就生成 ConstantValue 属性进行初始化；否则在类构造函数<clinit>中进行初始化。

3) Exceptions 属性

该属性列举出方法中可能抛出的受查异常（即方法描述时 throws 关键字后列出的异常），与 Code 属性平级，与 Code 属性包含的异常表不同，其结构如表 4.12 所示。

表 4.12 Exceptions 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

number_of_exceptions 表示可能抛出 number_of_exceptions 种受查异常。

`exception_index_table` 为异常索引集合，一组 `u2` 类型 `exception_index` 的集合，每一个 `exception_index` 为一个指向常量池中一个 `CONSTANT_Class_info` 型常量的索引，代表该受查异常的类型。

4) InnerClasses 属性

该属性用于记录内部类和宿主类之间的关系。如果一个类中定义了内部类，编译器将会为这个类与这个类包含的内部类生成 `InnerClasses` 属性，其结构如表 4.13 所示。

表 4.13 InnerClasses 属性结构表

类 型	名 称	数 量
<code>u2</code>	<code>attribute_name_index</code>	1
<code>u4</code>	<code>attribute_length</code>	1
<code>u2</code>	<code>number_of_classes</code>	1
<code>inner_classes_info</code>	<code>inner_classes</code>	<code>number_of_classes</code>

`inner_classes` 为内部类表集合，一组内部类表类型数据的集合，`number_of_classes` 即为集合中内部类表类型数据的个数。

每一个内部类的信息都由一个 `inner_classes_info` 表来描述，`inner_classes_info` 表结构如表 4.14 所示。

表 4.14 inner_classes_info 表结构

类 型	名 称	数 量
<code>u2</code>	<code>inner_class_info_index</code>	1
<code>u2</code>	<code>outer_class_info_index</code>	1
<code>u2</code>	<code>inner_name_index</code>	1
<code>u2</code>	<code>inner_name_access_flags</code>	1

`inner_class_info_index` 和 `outer_class_info_index` 指向常量池中 `CONSTANT_Class_info` 类型常量索引，该 `CONSTANT_Class_info` 类型常量指向常量池中 `CONSTANT_Utf8_info` 类型常量，分别为内部类的全限定名和宿主类的全限定名。

`inner_name_index` 指向常量池中 `CONSTANT_Utf8_info` 类型常量的索引，为内部类名称，如果为匿名内部类，则该值为 0。

`inner_name_access_flags` 类似于 `access_flags`，是内部类的访问标志，该标识的可选项值与前面描述类的访问属性的可选项值一致。

5) LineNumberTale 属性

用于描述 Java 源码的行号与字节码行号之间的对应关系，非运行时必需属性，会默认生成至 class 文件中，可以使用 javac 的 -g:none 或 -g:lines 命令关闭或要求生成该项属性信息，其结构如表 4.15 所示。

表 4.15 LineNumberTale 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

line_number_table 是一组 line_number_info 类型数据的集合，其所包含的 line_number_info 类型数据的数量为 line_number_table_length。line_number_info 结构如表 4.16 所示。

表 4.16 Line_number_info 属性结构表

类 型	名 称	数 量	说 明
u2	start_pc	1	字节码行号
u2	line_number	1	Java 源码行号

不生成该属性的最大影响是：

- ◎ 抛出异常时，堆栈将不会显示出错的行号。
- ◎ 调试程序时无法按照源码设置断点。

6) LocalVariableTable 属性

用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系，非运行时必需属性，默认不会生成至 class 文件中，可以使用 javac 的 -g:none 或 -g:vars 命令关闭或要求生成该项属性信息，其结构如表 4.17 所示。

表 4.17 LocalVariableTable 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

`local_variable_table` 是一组 `local_variable_info` 类型数据的集合，其所包含的 `local_variable_info` 类型数据的数量为 `local_variable_table_length`。`local_variable_info` 结构如表 4.18 所示。

表 4.18 `local_variable_info` 结构表

类 型	名 称	数 量	说 明
u2	<code>start_pc</code>	1	局部变量的生命周期开始的字节码偏移量
u2	<code>length</code>	1	局部变量作用范围覆盖的长度
u2	<code>name_index</code>	1	指向常量池中 <code>CONSTANT_Utf8_info</code> 类型常量的索引，局部变量名称
u2	<code>descriptor_index</code>	1	指向常量池中 <code>CONSTANT_Utf8_info</code> 类型常量的索引，局部变量描述符
u2	<code>index</code>	1	局部变量在栈帧局部变量表中 Slot 的位置，如果这个变量的数据类型为 64 位类型 (<code>long</code> 或 <code>double</code>)，它占用的 Slot 为 <code>index</code> 和 <code>index+1</code> 这 2 个位置

`start_pc + length` 即为该局部变量在字节码中的作用域范围。

不生成该属性的最大影响是：

- ◎ 当其他人引用这个方法时，所有的参数名称都将丢失，IDE 可能会使用诸如 `arg0`、`arg1` 之类的占位符代替原有的参数名称，对代码运行无影响，会给代码的编写带来不便。
- ◎ 调试时调试器无法根据参数名称从运行上下文中获取参数值。

7) SourceFile 属性

用于记录生成这个 class 文件的源码文件名称，为可选项，可以使用 `javac` 的 `-g:none` 或 `-g:source` 命令关闭或要求生成该项属性信息，其结构如表 4.19 所示。

表 4.19 `SourceFile` 属性结构表

类 型	名 称	数 量
u2	<code>attribute_name_index</code>	1
u4	<code>attribute_length</code>	1
u2	<code>sourcefile_index</code>	1

可以看出，`SourceFile` 属性是一个定长属性，`sourcefile_index` 是指向常量池中一个 `CONSTANT_Utf8_info` 类型常量的索引，常量的值为源码文件的文件名。

对大多数文件，类名和文件名是一致的，少数特殊类除外（如内部类），此时如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错误代码所属的文件名。

8) Deprecated 属性和 Synthetic 属性

这两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念。

`Deprecated` 属性表示某个类、字段或方法已经被程序作者定为不再推荐使用，可在代码中使用`@Deprecated`注解进行设置。

`Synthetic` 属性表示该字段或方法不是由 Java 源码直接产生的，而是由编译器自行添加的（当然也可设置访问标志 `ACC_SYNTHETIC`，所有由非用户代码产生的类、方法和字段都应当至少设置 `Synthetic` 属性和 `ACC_SYNTHETIC` 标志位中的一项，唯一的例外是实例构造函数`<init>`和类构造函数`<clinit>`）。

这两项属性的结构（当然 `attribute_length` 的值必须为 `0x00000000`）如表 4.20 所示。

表 4.20 属性结构

类 型	名 称	数 量
u2	<code>attribute_name_index</code>	1
u4	<code>attribute_length</code>	1

4. 继续第一个方法

上面对 JVM 所定义的 9 大属性表进行了简介，我们有了对属性表的基本结构认识，接下来便可以继续分析上文中第一个方法的字节码。

Test.class 字节码文件中，第 `0x020D` 和第 `0x020E` 两个位置的字节一起组成了第一个方法的属性表数量 `attributes_count`，紧跟在属性表数量后面的是 `attributes` 数组的第一个属性。上文提到，虽然 JVM 所支持的 9 大属性，其相互之间格式相差甚远，但是都会以一个 `u2` 类型的属性名称开始，JVM 根据名称便可知道当前描述的到底是这 9 大属性中的哪一个属性。Test.class 字节码文件中，紧跟在 `attributes_count` 后面的是第 `0x020F` 与第 `0x0210` 这两个字节，其值是 `0x000D`，如图 4.22 所示。



图 4.22 void <clinit>()方法的属性名称索引

属性表的名称索引指向常量池中对应的位置，根据上文所打印的 Test.class 常量池信息可知，常量池中第 `0x000D`（即 13）号元素的值是 `Code`，正是上文所介绍的 9 大属性中的 `Code` 方法表。根据上文所介绍的 `Code` 方法表属性的组成结构可知，紧跟在 `u2` 类型的属性名称后面的是 `u4`

类型的 attribute_length，对照图 4.22 可知，attribute_length 值为 0x00000029（如图 4.23 所示），对应十进制的 41。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F....

图 4.23 void <clinit>()方法的属性长度

Code 属性组成结构中紧跟在 attribute_length 后面的是 u2 类型的 max_stack 和 u2 类型的 max_locals，其值分别是 1 和 0，如图 4.24 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F....

图 4.24 void <clinit>()方法的最大栈深和局部变量表

紧跟在 max_locals 后面的是 u4 类型的 code_length，其值是 9，如图 4.25 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F....

图 4.25 void <clinit>()方法的字节码指令长度

code_length 之后的字节码流是 code 属性，code 属性开始真正描述 Java 方法所对应的字节码指令，这是 Java 的精华所在。字节码指令所占的字节码长度由 code_length 决定，由于 code_length 的值是 9，因此 code_length 后面的 9 字节都用于描述字节码指令。这 9 个字节码的值是：10 06 B8 00 0E B3 00 14 B1，如图 4.26 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F....

图 4.26 void <clinit>()方法的字节码指令

JVM 是基于栈的指令集系统，其设计的指令仅占 1 字节，由于 1 字节最多只能描述 256 种指令，所以 JVM 的指令总数只有 200 多个。同时，JVM 的指令属于一元操作数类型，其后面只有一个操作数（当然，很多指令后不跟操作数，例如 `return`）。正因如此，对于 JVM 指令需要区别对待，有些字节是代表指令，但是有些字节则代表数据（专业术语叫作“操作数”），而不是指令。

下面来分析当前方法的指令逻辑。从第一字节开始，第一字节一定是代表指令，而不能是数字（否则连 JVM 自己都不知道真正的指令到底从哪里开始）。第一字节是 0x10，查询 JVM 的指令集可知，其含义如下：

指令码	助记符	说 明
0x10	bipush	将 int、float 或 String 型常量值从常量池中推送至栈顶

该指令后面会跟 1 字节作为操作数，因此需要连着两字节一起看。其后面的 1 字节是 0x06，因此第一个指令所表述的含义是：将整型数字 6 推送至栈顶。如图 4.27 所示。



图 4.27 void <clinit>()方法的第 1 条字节码指令

第一个指令的字节码流结束后，紧跟其后的第一字节一定还是指令而非数值。跟在 0x1006 后面的第一字节是 0xB8，查询指令集可知，其含义如下：

指令码	助记符	说 明
0xB8	invokestatic	调用静态方法

该指令属于一元指令码，其后面会跟一个占 2 字节的数据来指向常量池中的方法编号，其后面的 2 字节是 0x000E，指向常量池中第 14 号元素。根据上文所打印的常量池信息可知，常量池中第 14 号元素信息如下：

```
const #14 = Method #15.#17; // java/lang/Integer.valueOf:()Ljava/lang/Integer;
const #15 = class #16; // java/lang/Integer
const #16 = Asciz java/lang/Integer;
const #17 = NameAndType #18:#19;// valueOf:()Ljava/lang/Integer;
const #18 = Asciz valueOf;
const #19 = Asciz (I)Ljava/lang/Integer;;
```

第14号常量池元素类型是Method，其引用了第15和第17号常量池元素，第17号元素又引用了第18和第19号元素，将它们最终拼凑起来，该方法是：

```
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

该方法信息分别描述了方法所在的类、方法名、入参和返回值。

第二条指令一共占3字节，其所在位置如图4.28所示。



图4.28 void <clinit>()方法的第2条字节码指令

分析完第二条指令，接下来开始分析第三条指令。同理，紧跟在第二条指令后面的第一字节一定是一个JVM指令，而非数据。该字节的值是0xB3，查询JVM指令集可知，其含义如下：

指令码	助记符	说明
0xB3	putstatic	用栈顶的值为指定的类的静态域赋值

该指令需要跟一个表明域编号的参数（占2字节），因此该指令最终占3字节，如图4.29所示。



图4.29 void <clinit>()方法的第3条字节码指令

后面2字节的值是0x0014，对应十进制数20，表明其指向常量池中第20号元素。常量池中第20号元素的信息如下：

```
const #1 = class           #2; // Test
const #7 = Asciz           si;
const #8 = Asciz           Ljava/lang/Integer;;
const #20 = Field          #1.#21; // Test.si:Ljava/lang/Integer;
const #21 = NameAndType #7:#8;// si:Ljava/lang/Integer;
```

第 20 号元素引用了常量池中其他元素，最终拼凑起来的含义是：

Test.si:Ljava/lang/Integer

这表明最终引用的是 Test 类中的 si 静态变量。于是第三条指令的含义就很清晰了，为 Test 类中的 si 静态变量赋值（值来自栈顶）。

第三条指令分析完了，接着分析后续指令。第三条指令后面紧接着的第一个字节码是 0xB1，查询 JVM 指令集可知，其代表的含义如下：

指令码	助记符	说明
0xB1	return	从当前方法返回 void

由于返回的是 void，因此该指令后面没有操作符。

前面提到，当前方法（即 void <clinit>()）的 code_length 是 9，即该方法对应的指令一共占 9 字节，而上面所分析的 4 个指令正好占了 9 字节，因此第一个方法的 Code 属性到此结束。

刚才是逐条分析各个指令的含义，但仅关注单个指令并不能清楚地知道程序想要干什么，因此需要将一个 Java 方法所对应的全部指令连起来一起看。上文使用 javap -verbose 命令分析 class 文件时，会将每一个 Java 方法所对应的 JVM 字节码打印出来，如下：

```
static {};
Code:
Stack=1, Locals=0, Args_size=0
0: bipush 6
2: invokestatic #14; //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5: putstatic      #20; //Field si:Ljava/lang/Integer;
8: return
```

这段指令的含义是：将数据 6 推送至栈顶，然后调用 java.lang.Integer.valueOf(int i)方法将其转换为 java.lang.Integer 类型，最后再将转换后的结果赋值给 Test.si 这个静态变量。由于 Test 类中只有一个静态变量 si，因此在整个类初始化阶段的确只需要将 si 进行初始化。

5. 第一个方法中的属性表

JVM 一共支持 9 大属性，Code 就是其中一个属性。然而 Code 属性中也会引用其他 8 类属性。Code 属性结构的最后 2 个组成部分分别是 attributes_count 和 attributes，分别代表所引用的属性总数和属性信息。其中，attributes_count 占 4 字节，attributes 所占字节数需要视具体情况而定。

在 Test.class 字节码文件中，描述完第一个方法（void <clinit>()）的 Code 属性后，接下来的字节流便开始描述该方法所引用的其他属性信息。开始的 4 字节表示 attributes_count，其内

容如图 4.30 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 00 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B3 00 00
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 00 06 00
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 00 00
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 00 00F....
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D 00 00*....*
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 00 00

图 4.30 void <clinit>()方法的属性长度

由图 4.30 可知, attributes_count 的值是 2, 表示 void <clinit>()方法中一共引用了两个其他的属性表。

上文提到, JVM 的 9 大属性表虽然结构各不相同,但是都以 u2 类型、占 2 字节的 tag 开头, JVM 通过 tag 来区分当前究竟是哪一种属性。由图 4.30 可知, 紧跟在 attributes_count 后面的 2 字节的值是 0x0016, 如图 4.31 所示

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 00 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B3 00 00
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 00 06 00
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 00 00
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 00 00F....
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D 00 00*....*
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 00 00

图 4.31 void <clinit>()方法属性 tag

由图 4.31 可知, 当前属性的 tag 值对应十进制的 22, 根据上文所打印的常量池信息可知, 常量池中第 22 号元素信息如下:

```
const #22 = Asciz      LineNumberTable;
```

由此可知, 当前所描述的属性是 LineNumberTable, 该属性用于记录源代码与字节码指令之间的行号对应关系。根据上文对 9 大属性的简介可知, 描述 LineNumberTable 属性的字节码流结构包括: u2 类型的 attribute_name_index、u4 类型的 attribute_length、u2 类型的 line_number_table_length 和 line_number_table 行号对应关系表。其中, u2 类型的 attribute_name_index 刚才已经分析过, 即指向常量池中第 22 号元素。紧跟在 attribute_name_index 后面的是 u4 类型的 attribute_length, 其值如图 4.32 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 F ..
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D * ... * ..
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04

图 4.32 void <clinit>()方法行号表属性元素长度

由图 4.32 可知，其值是 6，表明 LineNumberTable 这个属性接下来还占 6 字节，6 字节之后的字节码流就不属于当前属性了。

紧跟在 attribute_length 结构后面的是 u2 类型的 line_number_table_length，占 2 字节。该结构标记其后面的 line_number_table 的元素数量，其内容如图 4.33 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 F ..
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D * ... * ..
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04

图 4.33 void <clinit>()方法行号表的第一个元素的 tag

由图 4.33 可知，其值是 1，表明其后面的 line_number_table 的元素数量是 1。由于 attribute_length 的值是 6，而 line_number_table_length 的值占去了 2 字节，因此整个 LineNumberTable 属性还剩下最后 4 字节，最后 4 字节的内容如图 4.34 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 F ..
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D * ... * ..
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04

图 4.34 void <clinit>()方法行号表的字节码指令偏移与源码行号的对应关系

这最后 4 字节皆用于描述 line_number_table 行号对应关系表结构，line_number_table 是一组 line_number_info 类型数据的集合，即该表结构可以认为是一个标准的数组结构，其中所有元素类型都是 line_number_info 结构，而该结构由 2 个子结构构成：line_number_table_length

和 line_number_info，如表 4.21 所示：

表 4.21 子结构

类 型	名 称	数 量	说 明
u2	start_pc	1	字节码行号（也即字节码指令偏移位置）
u2	line_number	1	Java 源码行号

每一个子结构各占 2 字节，因此 line_number_table 结构所占的字节数一定是 4 的倍数。据此可以推测，由于留给 line_number_table 结构的只剩下 4 字节，因此这个结构数组中仅包含一个 line_number_info 元素。由图 4.34 可知，其值是 0x00000003，前两字节代表字节码指令偏移位置，值是 0，后两字节代表 Java 源码行号，值是 3。

上文使用 javap -verbose 命令所打印的常量池表中，包含 LineNumberTable 属性的描述，如图 4.35 所示。

```
static ();
Code:
Stack=1, Locals=0, Args_size=0
0: bipush 6
2: invokestatic #14; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5: putstatic     #20; //Field si:Ljava/lang/Integer;
8: return
LineNumberTable:
line 3: 0
```

图 4.35 使用 javap -verbose 命令所打印出的 void <clinit>()方法行号表

上面完成了 Test.class 字节码文件中第一个方法 void <clinit>() 中所引用的第一个属性 LineNumberTable 的分析，由于该方法一共引用了 2 个属性，因此继续分析第二个属性。同理，第二个属性以一个 u2 类型的 tag 开头，如图 4.36 所示。

000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B3)
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F...
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D*....*
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04

图 4.36 void <clinit>()方法的第二个属性 tag

其值是 0x0017，对应十进制的 23，表明该属性的名称指向常量池中第 23 号元素，常量池中该元素信息如下：

```
const #23 = Asciz      LocalVariableTable;
```

由此可知，当前属性类型是 LocalVariableTable，该属性描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系。根据上文对该属性简介可知，该属性组成结构依次是：u2 类型的 attribute_name_index、u4 类型的 attribute_length、u2 类型的 local_variable_table_length 和最后的 local_variable_table。紧跟在 attribute_name_index 后面的结构是 u4 类型的 attribute_length，其值如图 4.37 所示。

0000001f0	05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00	...
00000200	09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00	...
00000210	0D 00 00 00 29 00 01 00 00 00 00 09 10 06 B8	...
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06	...
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01	...
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F
00000250	00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D*....*
00000260	B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04

图 4.37 void <clinit>()方法的第二个属性长度

由图 4.37 可知，其值是 2，表明其后面的信息一共只占 2 字节。紧跟在 attribute_length 结构后面的结构是 u2 类型的 local_variable_table_length，由图 4.37 可知，其值是 0x0000，这表明当前方法并没有引用任何类成员变量。由于 local_variable_table_length 值是 0，因此字节码文件中就不再为其后面的结构 local_variable_table 分配空间。至此 Test.class 字节码文件对 LocalVariableTable 属性的描述结束。

现在，对 Test.class 字节码文件中第一个方法的分析全部完成。字节码文件中最复杂的部分应该就是对 Java 方法的描述，虽然 Test 类的<clinit>()方法很简单，但是相信各位道友应该有能力通过对这一简单方法的分析，窥一斑而知全豹，举一反三，独立分析其他更加复杂的 Java 方法的字节码信息。当然，在日常开发工作中，很少需要为了解决问题而去分析字节码文件，通过对字节码文件的分析，更主要的还是为了后面对 JVM 执行引擎工作原理的分析做铺垫，毕竟字节码文件是格式化的，JVM 也是基于同样的规范去解读字节码文件的，如果对字节码文件格式缺乏了解，那么在阅读 JVM 源码时将不太容易跟得上 JVM 作者们的思路和节奏，给源码阅读带来不少障碍。

6. <clinit>()方法的字节码长度

上文在分析 void <clinit>()方法时，描述该方法的 Code 属性中有一个结构专门描述该方法

所占字节数，该结构是 code_length。通过上面的分析可知，<clinit>()方法的 code_length 值是 41，这个值代表的范围是多大呢？请看图 4.38 所示。

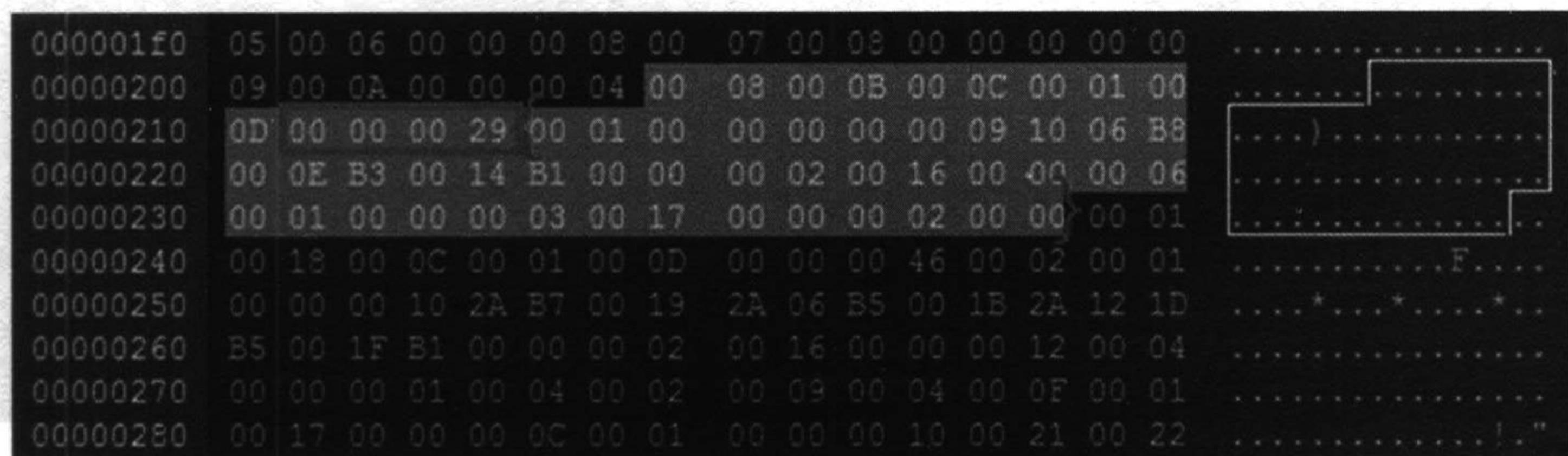


图 4.38 <clinit>()方法的 Code 属性的 code_length 结构所标识的范围

如图 4.38 所示，code_length 结构占 4 字节，从字节码文件中的第 0x0211 至第 0x02114，其后面的 41 个字节码流全部用于描述方法本身的指令、所引用的其他属性信息，正好到字节码文件中的第 0x023D 位置结束，过了这个位置，后面的字节码流信息就不再属于当前方法的范畴，由此可知，code_length 的作用域范围是当前方法的后续所有属性的长度总和。

Test.class 一共包含 4 个方法，下文不会再一个一个详细分析它们的字节码信息，而是会以 code_length 为指向标，标识出剩余 3 个方法的界限，具体的内容请读者自行完成。

7. 第二个方法

上文提到，methods 结构的组成如表 4.22 所示。

表 4.22 methods 结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

第二个方法的 access_flags、name_index、descriptor_index 和 attributes_count 的信息如图 4.39 所示。

00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8).....
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F
00000250	00 00 00 10 2A 37 00 19 2A 06 B5 00 1B 2A 12 07	...*...*....*
00000260	B5 00 1F 11 00 C0 00 02 00 16 00 00 00 12 00 04
00000270	00 00 00 00 00 00 00 00 00 00 00 00 00 0F 00 00
00000280	00 17 00 00 00 00 00 00 00 00 00 21 00 22	access_flag
00000290	00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 4E#.\$.....N
000002a0	00 00 00 02 00 00 00 00 00 00 00 00 00 25 4CY..\$L
000002b0	2B 14 08 B8 00 02 B6 00 26 B1 00 00 00 02 00 16	+.....&.....
000002c0	name_index	00 00 00 07 00 08 00 08 00 11
000002d0	00 09 00 17 00 00 00 16 00 02 00 00 00 12 00 2A

图 4.39 第二个方法的字节码信息

其值分别是 0x01、0x18、0x0C 和 0x01，对照上文所给出的相关选项值信息可知，当前方法的访问标识是 public，当前方法的名字和入参与返回值分别指向常量池中第 24 号、第 12 号元素，信息如下：

```
const #12 = Asciz      ()V;
const #24 = Asciz      <init>;
```

根据该信息可知，当前方法描述的是类的构造函数，其返回值类型是 void，无入参，因此是默认构造函数。注意，字节码文件中名字为<init>的方法表示的是类的构造函数，上文所描述的第一个方法名是<clinit>，这是类型的初始化方法。这两者的区别是：当 JVM 决定加载某个类型时，会调用<clinit>()方法，而当 JVM 决定实例化某个类型时，会调用<init>方法。一个是类型初始化，一个是类实例的初始化，两者有本质上的区别。并且<clinit>()一定先于<init>()方法被调用。

当前方法的 attribute_count 值是 1，表示当前方法引用了一个属性。

8. 第三个方法

第三个方法的 access_flags、name_index、descriptor_index 和 attributes_count 的信息如图 4.40 所示。

该方法的 access_flag 值是 9，对照上文所给出的方法的 access_flags 可选值可知，该值是由表示 ACC_PUBLIC 的 0x0001 和表示 ACC_STATIC 的 0x0008 合成的，因此当前方法的访问修饰符为 public static。



图 4.40 第三个方法的字节码信息

同时，该方法的 `name_index` 值为 0x23，找到常量池中索引号为 35 的元素，其值如下：

```
const #35 = Asciz      main;
```

由此可知，第三个方法描述的是 `main()` 主函数。关于该方法的具体信息，相信各位道友在看完上面对第一个方法的详细分析后，完全有能力自行研究，这里限于篇幅便不再逐个详细解析了。

除了本章所分析的魔数、版本号、常量池、字段、方法等属性，Java class 字节码文件中还有很多其他属性，这里也不逐个深入分析了。通过对上面这几个属性的分析，认真的道友一定能够把握住字节码格式的套路。在分析属性时，只需要按照 JVM 官方规范所指定的格式，肯定能够分析出字节码文件中所隐藏的信息。

4.7 本章回顾

本章以一个具体的 Java 程序为例，分析了其对应的字节码文件中的数据。可以发现，Java 编译器在生成 Java class 字节码文件时，全是按套路出牌的，那么我们在分析时也按照套路走，便不难理解字节码文件的内容。

字节码文件中最重要的是 Java 方法所对应的字节码指令（至少笔者这么认为），Java 源程序的逻辑都封装在字节码指令中。对字节码指令在字节码文件中的存储方式有了透彻的理解，便意味着你对 JVM 执行引擎入门了^_^。

第 5 章

常量池解析

本章摘要

- ◎ Java 字节码常量池的内存分配链路
- ◎ oop-klass 模型
- ◎ 常量池的解析原理

前文讲述了 JVM 执行引擎的核心机制，以及 Java class 字节码文件的格式。从本章开始，继续剖析 JVM 内部的源码。JVM 要完成 Java 逻辑的执行，必须能够“读懂”Java 字节码文件。而 Java 字节码文件从总体上而言，其实主要包含三部分：常量池、字段信息和方法信息。其中常量池存储了字段和方法的相关符号信息，因此对常量池的解读便成为解读 Java 字节码文件的基础。本章主要分析 JVM 解析 Java 字节码文件中常量池的逻辑。本章内容难度属于中等，只要你稍微具备一点 C/C++基础便能读懂，当然，即使完全不具备 C/C++基础，读懂本章也不是难事，大家都是写程序的，而写程序的小伙伴们，智商都是蛮高的^_^。

常量池是 Java 字节码文件的核心，因此也是 JVM 解析字节码的重头戏。要注意的是，这里所说的常量池并不等同于 JVM 内存模型中的常量区，这里的常量池仅仅是文件中的一堆字节码而已。但是字节码文件中的常量池与 JVM 内存区的常量区之间却有着千丝万缕的联系，因为 JVM 最终会将字节码文件中的常量池信息进行解析，并存储到 JVM 内存模型中的常量区。字节码文件中的常量池是 Java 编译器对 Java 源代码进行语法解析后的产物，只是这种初步解析产生的结果比较粗糙，里面包含了各种引用，信息不够直观。而 JVM 根据字节码文件中的常量池信息再进行二次解析，这种解析目标清晰，直奔终点，会还原出所有常量元素的全部信息，让内存与你所编写的 Java 源代码保持一致。