**第6章 类文件结构**

Java代码编译的结果：从本地机器码转变为字节码

各种运行在不同平台（不同操作系统和硬件体系结构）的虚拟机都可以载入和执行同一种与平台无关的字节码，从而实现了程序的

“一次编写，到处运行”—— Write Once, Run Anywhere.

各种不同平台的虚拟机与所有平台都统一使用的程序存储格式——**字节码（ByteCode）**是构成平台无关性、语言无关性的基石。

java程序 🡺 javac编译器 🡺 字节码 🡺 java虚拟机执行

\*.java \*.class

一、Class类文件的结构

任何一个Class文件都对应着唯一一个类或接口的定义信息；

但相反，类或接口并不一定得定义在文件里（比如类或接口也可以通过类加载器直接生成，即不一定以磁盘文件的形式存在：\*.class）

Class文件是一组以**8位字节为基础单位**的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件之中，中间没有添加任何分隔符。因此Class文件存储内容几乎全部是程序运行的必要数据，没有空隙存在。当遇到需要占用**8位字节以上空间的数据项**时，会按照**高位在前**（Big-Endian）的方式**分割成若干个8位字节进行存储**。

注：Big-Endian：用于SPARC、PowerPC等处理器，具体指最高位字节在地址最低位、最低位字节在地址最高位来存储数据。

Little-Endian：用于x86等处理器

Class文件格式采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：**无符号数**和**表**，后面的解析都要以这两种数据类型为基础。

无符号数：基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成字符串值。

表：由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以\_info结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上就是一张表。

|  |  |  |
| --- | --- | --- |
| Class 文件格式 | | |
| 类型 | 名称 | 数量 |
| u4 | magic | 1 |
| u2 | minor\_version | 1 |
| u2 | major\_version | 1 |
| u2 | constant\_pool\_count | 1 |
| cp\_info | constant\_pool | constant\_pool\_count-1 |
| u2 | access\_flags | 1 |
| u2 | this\_class | 1 |
| u2 | super\_class | 1 |
| u2 | interfaces\_count | 1 |
| u2 | interfaces | interfaces\_count |
| u2 | fields\_count | 1 |
| field\_info | fields | fields\_count |
| u2 | methods\_count | 1 |
| method\_info | methods | methods\_count |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |

无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会使用：**一个前置的容量计数器** + **若干个连续的数据项形式**，这时称这一系列连续的某一类型的数据为**某一类型的集合**。

TestClass.class文件

addr = 0x00000000 … const #1 const #2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **CA** | **FE** | **BA** | **BE** | **00** | **00** | **00** | **32** | **00** | **16** | **07** | **00** | **02** | **01** | **00** | **1D** |
| Magic Number 魔数  值 = 0xCAFEBABE | | | | Minor  Version  次版本号  0x0000 | | Major  Version  主版本号  0x0032 =  50,JDK1.6 | | constant\_  pool\_count  常量池容量计数值,计数  从1开始。  0x0016=22D  即有21项  索引值取值范围[1,21] | | tag=  0x07  即表示该常量为类或接口的符号引用 | name index  索引值，指向常量池中一个CONSTANT  \_Utf8\_info类型常量，此常量代表类/接口的全限定名  0x0002指向常量池第2项常量 | | tag=  0x01即表示该常量为UTF-  8编码的字符串 | length=  0x001D=29D  表示该utf-  8缩略编码  的字符串长  度大小是29个字节 | |

addr = 0x00000010 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | | **C** | **D** | **E** | **F** |
| **6F** | **72** | **67** | **2F** | **66** | **65** | **6E** | **69** | **78** | **73** | **6F** | **66** | | **74** | **2F** | **63** | **6C** |
| ‘o’ | ‘r’ | ‘g’ | ‘/’ | ‘f’ | ‘e’ | ‘n’ | ‘I’ | ‘x’ | ‘s’ | ‘o’ | ‘f’ | ‘t’ | | ‘/’ | ‘c’ | ‘l’ |

addr = 0x00000020 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | | **C** | **D** | **E** | **F** |
| **61** | **7A** | **7A** | **2F** | **54** | **65** | **73** | **74** | **43** | **6C** | **61** | **73** | | **74** | **07** | **00** | **04** |
| ‘a’ | ‘z’ | ‘z’ | ‘/’ | ‘T’ | ‘e’ | ‘s’ | ‘t’ | ‘C’ | ‘l’ | ‘a’ | ‘s’ | ‘s’ | |  |  |  |

addr = 0x00000030 …

…

**魔数：**

每个Class文件的头4个字节，用于确定该文件是否为一个能被虚拟机接受的Class文件。gif、jpeg文件头都存有魔数。使用魔数而不用扩展名识别主要基于安全方面的考虑，因为文件扩展名可以被随意改动。Class文件的魔数值为：0xCAFEBABE

**主版本号、次版本号：**

java版本号从45开始，JDK1.1之后，每个JDK大版本发布，主版本号向上 + 1。高版本的JDK能向下兼容以前版本的Class文件，但不能运行以后版本的Class文件，即使文件格式并未发生任何变化，虚拟机也必须拒绝执行超过其版本号的Class文件。0x0032 = 50D -> JDK1.6

eg：JDK1.1 能支持 45.0 ~ 45.65535 1111111111111111 = 2^17-1

JDK1.2 能支持 45.0 ~ 46.65535

JDK1.7 能支持 45.0 ~ 51.65535

JDK1.8 能支持 45.0 ~ 52.65535

**常量池**：

Class文件之中的资源仓库。是Class文件结构中与其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一，同时它还是在Class文件中第一个出现的表类型数据项目。常量池入口需放置一项u2类型的数据，代表常量池容量计数值(constant\_pool\_count)。注意这个容量计数是从1开始的。0x0016(22D)表示常量池中有21项常量，索引值范围为1~21。将第0项常量空出来，目的在于满足后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义。这种情况就可以把索引值置为0来表示。

注：**Class文件结构中只有常量池的容量计数是从1开始**，对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从0开始的。

常量池中主要存放两大类常量：**字面量**(Literal)和**符号引用**(Symbolic References)。

字面量：接近于Java语言层面的常量概念，如：

（1）文本字符串，

（2）声明为final的常量值等。

符号引用：属于编译原理方面的概念，包括以下三类常量：

（1）类和接口的全限定名

（2）字段的名称和描述符

（3）方法的名称和描述符

注：在Class文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。当虚拟机运行时，需要**从常量池获得对应的符号引用**，再在**类创建时或运行时解析、翻译到具体的内存地址**之中。

常量池中**每一项常量都是一个表**，JDK1.7共有14种结构各不相同的表结构数据。

|  |  |  |
| --- | --- | --- |
| 常量池的项目类型 | | |
| 类型 | 标志(tag) | 描述 |
| CONSTANT\_Utf8\_info | 1 | UTF-8编码的字符串 |
| CONSTANT\_Integer\_info | 3 | 整型字面量 |
| CONSTANT\_Float\_info | 4 | 浮点型字面量 |
| CONSTANT\_Long\_info | 5 | 长整型字面量 |
| CONSTANT\_Double\_info | 6 | 双精度浮点型字面量 |
| CONSTANT\_Class\_info | 7 | 类或接口的符号引用 |
| CONSTANT\_String\_info | 8 | 字符串类型字面量 |
| CONSTANT\_Fieldref\_info | 9 | 字段的符号引用 |
| CONSTANT\_Methodref\_info | 10 | 类中方法的符号引用 |
| CONSTANT\_InterfaceMethodref\_info | 11 | 接口中方法的符号引用 |
| CONSTANT\_NameAndType\_info | 12 | 字段或方法的部分符号引用 |
| CONSTANT\_MethodHandle\_info | 15 | 表示方法句柄 |
| CONSTANT\_MethodType\_info | 16 | 标识方法类型 |
| CONSTANT\_InvokeDynamic\_info | 18 | 表示一个动态方法调用点 |

表开始第一个字节是一个u1类型的标志位(tag)，代表当前这个常量属于哪种常量类型。

UTF-8缩略编码与普通UTF-8编码的区别是：

（1）从 ‘\u0001’ 到 ‘\u007f’ 之间的字符（相当于1~127的ASCII码）的缩略编码使用一个字节表示；

（2）从 ‘\u0080’ 到 ‘\u07ff’ 之间的所有字符的缩略编码用两个字符表示；

（3）从 ‘\u0800’ 到 ‘\uffff’ 之间的所有字符的缩略编码就按照普通UTF-8编码规则使用三个字节表示

|  |  |  |  |
| --- | --- | --- | --- |
| 常量池中的14种常量项的结构总表 | | | |
| 常量 | 常量表项 | 类型 | 取值及描述 |
| CONSTANT\_Utf8\_info | tag | u1 | 值为1,表示该常量为utf-8编码的字符串 |
| length | u2 | 字符串占用的字节数 |
| bytes | u1 | 长度为length的utf-8编码的字符串 |
| CONSTANT\_Integer\_info | tag | u1 | 值为3,表示该常量为整型字面量 |
| bytes | u4 | 按照高位在前存储的int值 |
| CONSTANT\_Float\_info | tag | u1 | 值为4,表示该常量为浮点型字面量 |
| bytes | u4 | 按照高位在前存储的float值 |
| CONSTANT\_Long\_info | tag | u1 | 值为5,表示该常量为长整型字面量 |
| bytes | u4 | 按照高位在前存储的long值 |
| CONSTANT\_Double\_info | tag | u1 | 值为6,表示该常量为双精度浮点型字面量 |
| bytes | u8 | 按照高位在前存储的Double值 |
| CONSTANT\_Class\_info | tag | u1 | 值为7, 表示该常量为类或接口的符号引用 |
| name\_index | u2 | 是一个指向全限定名常量项CONSTANT\_Utf8\_  info的索引 |
| CONSTANT\_String\_info | tag | u1 | 值为8, 表示该常量为字符串类型字面量 |
| index | u2 | 是一个指向字符串字面量的索引 |
| CONSTANT\_Fieldref\_info | tag | u1 | 值为9, 表示该常量为字段的符号引用 |
| index | u2 | 是一个指向声明字段的类或者接口描述符CONSTANT\_Class\_info的索引项 |
| index | u2 | 是一个指向字段描述符CONSTANT\_NameAnd  Type的索引项 |
| CONSTANT\_Methodref\_info | tag | u1 | 值为10, 表示该常量为类中方法的符号引用 |
| index | u2 | 是一个指向声明方法的类描述符常量项CONST  ANT\_Class\_info的索引 |
| index | u2 | 是一个指向名称及类型描述符常量项CONSTANT\_NameAndType的索引 |
| CONSTANT\_InterfaceMetho  dref\_info | tag | u1 | 值为11, 表示该常量为接口中方法的符号引用 |
| index | u2 | 是一个指向声明方法的接口描述符常量项CONSTANT\_Class\_info的索引 |
| index | u2 | 是一个指向名称及类型描述符常量项CONSTANT\_NameAndType的索引 |
| CONSTANT\_NameAndType  \_info | tag | u1 | 值为12, 表示该常量为字段或方法的部分符号引用 |
| index | u2 | 是一个指向该字段或方法名称常量项的索引 |
| index | u2 | 是一个指向该字段或方法描述符常量项的索引 |
| **CONSTANT\_MethodHandle**  **\_info** | tag | u1 | 值为15, 表示该常量为方法句柄 |
| reference\_  kind | u1 | 值的取值必须在[1,9], 它决定了方法句柄的类型。方法句柄类型的值表示方法句柄的字节码行为 |
| reference\_  index | u2 | 值必须是对常量池的有效索引 |
| **CONSTANT\_MethodType\_**  **info** | tag | u1 | 值为16, 表示该常量为标识方法类型 |
| descriptor\_  index | u2 | 值必须是对常量池的有效索引, 常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构, 表示方法的描述符 |
| **CONSTANT\_InvokeDynamic\_**  **info** | tag | u1 | 值为18, 表示该常量为一个动态方法调用点 |
| bootstrap\_  method\_attr  \_index | u2 | 值必须是对当前Class文件中引导方法表的  bootstrap\_methods[] 数组的有效索引。 |
| name\_and\_typ  e\_index | u2 | 值必须是对当前常量池的有效索引, 常量池在该索引处的项必须是CONSTANT\_NameAndType  \_info结构, 表示方法名和方法描述符 |

蓝色部分为JDK1.7新增

**访问标志**

在**常量池结束之后，紧接着两个字节代表访问标志（access\_flags）**，这个标志用于标识一些**类或者接口层次**的访问信息，包括：这个Class是类还是接口；是否定义为public类型；是否定义为abstract类型；如果是类的话，是否被声明为final等。

|  |  |  |
| --- | --- | --- |
| 访问标志 | | |
| 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字节码指令的新语意。标志必须为真 |
| ACC\_INTERFACE | 0x0200 | 标识这是一个接口 |
| ACC\_ABSTRACT | 0x0400 | 是否为abstract类型，对于接口或者抽象类来说，此标志值为真，其他类值为假 |
| ACC\_SYNTHETIC | 0x1000 | 标识这个类并非由用户代码产生的 |
| ACC\_ANNOTATION | 0x2000 | 标识这是一个注解 |
| ACC\_ENUM | 0x4000 | 标识这是一个枚举 |

eg: public class TestClass {}

ACC\_PUBLIC、ACC\_SUPER 为真

ACC\_FINAL、ACC\_INTERFACE、ACC\_ABSTRACT、ACC\_SYNTHETIC、ACC\_ANNOTATION、ACC\_ENUM 为假

∴ access\_flags = 0x0001|0x0020 = 0x0021

**类索引、父类索引与接口索引集合**

类索引、父类索引、接口索引集合按顺序排列在访问标志之后；

类索引（this\_class）和父类索引（super\_class）都是一个u2类型的数据，它们各自指向一个类型为CONSTANT\_Class\_info的类描述符常量，通过CONSTANT\_Class\_info类型的常量中的索引值（name\_index），可以查找到定义在CONSTANT\_Utf8\_info类型的常量中的全限定名字符串。

类索引查找全限定名过程

**this\_class** **CONSTANT\_Class\_info**  **CONSTANT\_Utf8\_info**

{u2 **index=0x0001** { u1 tag=0x07 { u1 tag=0x01

} u2 **name\_index=0x0002** u2 length=0x001D

} u1 bytes=“org/fenixsoft/clazz/

TestClass”

}

**常量池中**

而接口索引集合（interfaces）是一组u2类型的数据的集合，入口的第一项为u2类型的接口计数器（interfaces\_count），表示索引表的容量。如果该类没有实现任何接口，则该计数器值为0，后面接口的索引表不再占用任何字节。

Class文件中由这三项数据来确定这个类的继承关系：

类索引用于确定这个类的全限定名；

父类索引用于确定这个类的父类的全限定名。由于Java只允许单继承，因此父类索引只有一个。除java.lang.Object外，所有Java类的父类索引都不为0；

接口索引集合用于描述这个类实现了哪些接口。这些被实现的接口将按implements语句（如果这个类本身是一个接口，则应当是extends语句，因为接口可以继承多个接口）后的接口顺序从左到右排列在接口索引集合中。

addr = 0x00000000 … const #1 const #2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **CA** | **FE** | **BA** | **BE** | **00** | **00** | **00** | **32** | **00** | **16** | **07** | **00** | **02** | **01** | **00** | **1D** |
| Magic Number | | | | Minor  Version | | Major  Version | | constant\_  pool\_count | | tag=  0x07 | name\_index  =0x0002 | | tag=  0x01 | length=  0x001D=29D | |

addr = 0x000000**1**0 …

addr = 0x000000**2**0 …

… **常量池{}**

addr = 0x000000**E**0 … const #21

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | | **C** | **D** | **E** | **F** |
| **0E** | **54** | **65** | **73** | **74** | **43** | **6C** | **61** | **73** | **73** | **2E** | **6A** | | **61** | **76** | **61** | **00** |
|  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  | 访问标志 |

addr = 0x000000**F**0 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | 7 | 8 | 9 | A | B | | C | D | E | F |
| **21** | **00** | **01** | **00** | **03** | **00** | **00** | 00 | 01 | 00 | 02 | 00 | | 05 | 00 | 06 | 00 |
| 0x0  021 | 类索引=0x0001 | | 父类索引=0x0003 | | 接口计数器=0x0000 | |  |  |  |  |  |  | |  |  |  |

**字段表集合**

字段表（field\_info）用于描述接口或者类中声明的变量；

字段（field）包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。

字段信息：

（1）字段的作用域：public、private、protected修饰符

（2）实例变量还是类变量：是否有static修饰符

（3）可变性：final修饰符

（4）并发可见性：volatile修饰符，是否强制从主内存读写

（5）可否被序列化：transient修饰符

（6）字段数据类型：基本类型、对象、数组

（7）字段名称

上述信息中，修饰符类的字段信息都是布尔值，即要么有该修饰符，要么没有。因此适合用标志位表示。

而字段的名字，字段的数据类型都是无法固定的，必须引用常量池中的常量来描述。

|  |  |  |  |
| --- | --- | --- | --- |
| 字段表结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | access\_flags | 1 | 字段标志，访问修饰符 |
| u2 | name\_index | 1 | 对常量池的引用，字段的简单名称索引 |
| u2 | descriptor\_index | 1 | 对常量池的引用，字段的描述符索引 |
| u2 | attributes\_count | 1 |  |
| attribute\_info | attributes | attributes\_count | 属性表集合  {attributes\_name\_index; …; …} |

|  |  |  |
| --- | --- | --- |
| 字段访问标志  （access\_flags，字段访问修饰符） | | |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 字段是否被public修饰 |
| ACC\_PRIVATE | 0x0002 | 字段是否被private修饰 |
| ACC\_PROTECTED | 0x0004 | 字段是否被protected修饰 |
| ACC\_STATIC | 0x0008 | 字段是否被static修饰 |
| ACC\_FINAL | 0x0010 | 字段是否被final修饰 |
| ACC\_VOLATILE | 0x0040 | 字段是否被volatile修饰 |
| ACC\_TRANSIENT | 0x0080 | 字段是否被transient修饰 |
| ACC\_SYNTHETIC | 0x1000 | 字段是否由编译器自动产生 |
| ACC\_ENUM | 0x4000 | 字段是否被enum修饰 |

注：根据java语言规则：

（1）ACC\_PUBLIC、ACC\_PRIVATE、ACC\_PROTECTED 最多选其一

（2）ACC\_FINAL、ACC\_VOLATILE 不能同时选择

（3）接口（interface）之中的字段必须有ACC\_PUBLIC、ACC\_STATIC、ACC\_FINAL标志（如果我们没

有在编程中指定，编译时会默认产生并加入到字段表集合中）

public interface test {

int a = 10; //编译后：public static final int a = 10;

}

**“全限定名”、“简单名称”、“描述符”的概念和区别**

全限定名：”org/fenixsoft/clazz/TestClass;”

简单名称：没有类型和参数修饰的方法或者字段名称，如类中的inc()方法和m字段的简单名称为 ”inc” 和 ”m”

描述符：用来描述字段的数据类型，或方法的参数列表（包括数量、类型、顺序）和返回值。

根据描述符规则，基本数据类型（byte、char、double、float、int、long、short、boolean）以及代表无返回值的void类型都用一个大写字母表示，而对象类型则用字母L加对象的类全限定名来表示。

|  |  |
| --- | --- |
| 描述符标识字符含义 | |
| 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 | 基本类型void |
| L | 对象类型，如Ljava/lang/Object |

对于数组类型，每一维度使用一个前置的 “[” 字符来描述。例如：

（1）一个定义为 “java.lang.String[][]” 类型的二维数组，将被记录为“[[Ljava/lang/String”;

（2）一个整型数组“int[]”将被记录为 “[I”;

对于方法，按照先参数列表，后返回值类型的顺序描述，参数列表按照参数的严格顺序放在一组小括号“()”之内。例如：

（1）方法 void inc() “()V”

（2）方法 java.lang.String.toString() “()Ljava/lang/String;”

（3）方法 int indexOf(char[] source, int sourceOffset, int sourceCount, char[] target, int targetOffset, int targetCount, int fromIndex) “([CII[CIII)I”

“org/fenixsoft/clazz/TestClass”

addr = 0x00000000 … const #1 const #2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **CA** | **FE** | **BA** | **BE** | **00** | **00** | **00** | **32** | **00** | **16** | **07** | **00** | **02** | **01** | **00** | **1D** |
| Magic Number | | | | Minor  Version | | Major  Version | | constant\_  pool\_count | | tag=  0x07 | name\_index  =0x0002 | | tag=  0x01 | length=  0x001D=29D | |

addr = 0x00000010 …

… const #3->const #4 “java/lang/Object” **常量池{}**

addr = 0x000000**E**0 … const #21

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | | **C** | **D** | **E** | **F** |
| **0E** | **54** | **65** | **73** | **74** | **43** | **6C** | **61** | **73** | **73** | **2E** | **6A** | | **61** | **76** | **61** | **00** |
|  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  | 访问标志 |

**字段表集合（private int m）**

const #5 “m” const #6 “I”

addr = 0x000000F0 …I

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **21** | **00** | **01** | **00** | **03** | **00** | **00** | **00** | **01** | **00** | **02** | **00** | **05** | **00** | **06** | **00** |
| 0x0  021 | 类索引  this\_class  =**0x0001**  指向第1项常量 | | 父类索引  super\_class  =**0x0003**  指向第3项常量 | | 接口计数器  interfaces\_c  ount  =0x0000 | | 容量计数器fields\_count  = 0x0001,表明该类只有  1个字段表 | | 访问标志  access\_flags  = 0x0002,仅  ACC\_PRIVATE为真，余为假  private | | 字段简单名称索引，对常量池的引用  name\_index  =**0x0005**, 即指向第5项常量  “m” | | 字段描述符索引，对常  量池的引用descriptor  \_index =**0x0006,**  即指向第6项常量项  “I” | | 属性表计数器  attri  bute  s\_co  unt =0x0  000 |

**方法表集合（public void <init>()、public void inc()）**

const #7 “<init>” const #8 “()V” const #9 “Code”

addr = 0x00000100 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **00** | **00** | **02** | **00** | **01** | **00** | **07** | **00** | **08** | **00** | **01** | **00** | **09** | **00** | **00** | **00** |
| 即没有额外描述的信息 | 容量计数器  method\_cou  nt  = 0x0002  即有2个方  法 | | access\_flags  =0x0001,  public | | name\_index  =0x0007  “<init>” | | descriptor  \_index  =0x0008  “()V” | | attributes\_co  unt = 0x0001  即该方法的属性表集合有一项属性 | | attributes\_na  me\_index =  0x0009  “code”说明此属性是方法的字节码描述 | |  |  |  |

**方法表集合**

对方法的描述与对字段的描述在结构上几乎完全一致。

|  |  |  |  |
| --- | --- | --- | --- |
| 方法表结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | access\_flags | 1 | 方法访问标志，修饰符 |
| u2 | name\_index | 1 | 对常量池的引用，方法的简单名称索引 |
| u2 | descriptor\_index | 1 | 对常量池的引用，方法的描述符索引 |
| u2 | attributes\_count | 1 |  |
| attribute\_info | attributes | attributes\_count | 属性表集合 |

|  |  |  |
| --- | --- | --- |
| 方法访问标志 | | |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 方法是否被public修饰 |
| ACC\_PRIVATE | 0x0002 | 方法是否被private修饰 |
| ACC\_PROTECTED | 0x0004 | 方法是否被protected修饰 |
| ACC\_STATIC | 0x0008 | 方法是否被static修饰 |
| ACC\_FINAL | 0x0010 | 方法是否被final修饰 |
| **ACC\_SYNCHRONIZED** | 0x0020 | 方法是否被synchronized修饰 |
| 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 | 方法是否被native修饰 |
| **ACC\_ABSTRACT** | 0x0400 | 方法是否被abstract修饰 |
| **ACC\_STRICTFP** | 0x0800 | 方法是否被strictfp修饰 |
| ACC\_SYNTHETIC | 0x1000 | 方法是否是由编译器自动产生的 |

方法里的Java代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为“Code”的属性里面。

与字段表集合相对应，如果父类方法在子类中没有被重写，方法表集合中就不会出现来自父类的方法信息。但同样的，有可能会出现由编译器自动添加的方法，最典型的便是类构造器“<clinit>”方法和实例构造器“<init>”方法。

在Java语言中，要**重载**（Overload）一个方法，除了要与原方法具有相同的简单名称之外，还要求必须拥有一个与原方法不同的特征签名，**特征签名**就是一个方法中各个参数在常量池中的字段符号引用的集合，即方法返回值是不包括在特征签名之内的。因此Java语言层面无法仅仅依靠返回值的不同来对一个已有方法进行重载。而在字节码层面（class文件）中，如果两个方法有相同的名称和特征签名，但返回值不同，那么也是可以合法共存在同一个Class文件中的。

**属性表集合**

在每个字段表、每个方法表都可以携带自己的属性表集合，以用于描述某些场景专有的信息。

属性表集合不要求各个属性表和其他数据项目一样具有严格的顺序、长度和内容。只要不和已有属性名重复，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，Java虚拟机运行时会忽略掉它不认识的属性。为了能正确解析Class文件。《Java虚拟机规范（Java SE 7）》版中，预定义属性有21项，具体如下：

|  |  |  |
| --- | --- | --- |
| 虚拟机规范预定义的所有属性 | | |
| 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、方法表、字段表 | 被声明为deprecated的方法和字段 |
| **Exceptions** | 方法表 | 方法抛出的异常 |
| EnclosingMethod | 类文件 | 仅当一个类为局部类或者匿名类时才能拥有这个属性，这个属性用于标识这个类所在的外围方法 |
| **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 |
| **StackMapTable** | Code属性 | 供类型检查验证器（Type Checker）检查和处理目标方法的局部变量和操作数栈所需要的类型是否匹配 |
| **Signature** | 类、方法表、字段表 | 用于支持泛型情况下的方法签名，在Java语言中，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则Signature属性会为它记录泛型签名信息。由于Java的泛型采用擦除法实现，在为了避免类型信息被擦除后导致签名混乱，需要这个属性记录泛型中的相关信息 |
| **SourceFile** | 类文件 | 记录源文件名称 |
| SourceDebugExtension | 类文件 | 用于存储额外的调试信息 |
| **Synthetic** | 类、方法表、字段表 | 标识方法或字段为编译器自动生成的 |
| LocalVariableTypeTable | 类 | 使用特征签名代替描述符，是为了引入泛型语法之后能描述泛型参数化类型而添加 |
| RuntimeVisibleAnnotations | 类、方法表、字段表 | 为动态注解提供支持。用于指明哪些注解是运行时（实际上运行时就是进行反射调用）可见的。 |
| RuntimeInvisibleAnnotations | 类、方法表、字段表 | 指明哪些注解是运行时不可见的。 |
| RuntimeVisibleParameterAnnotations | 方法表 | 指明哪些方法参数注解是运行时可见的。 |
| RuntimeInvisibleParameterAnnotations | 方法表 | 指明哪些方法参数注解是运行时不可见的。 |
| AnnotationDefault | 方法表 | 用于记录注解类元素的默认值 |
| **BootstrapMethods** | 类文件 | 用于保存invokedynamic指令引用的引导方法限定符 |

**对每个属性，它的名称都需要从常量池中引用一个CONSTANT\_Utf8\_info类型的常量来表示**，而属性值的结构则是完全自定义的，只需要通过一个u4的长度属性去说明属性值所占用的位数即可。

一个符合规则的属性表应该满足如下定义的结构：

|  |  |  |
| --- | --- | --- |
| 属性表结构 | | |
| 类型 | 名称 | 数量 |
| u2 | attribute\_name\_index | 1 |
| u4 | attribute\_length | 1 |
| u1 | info | attribute\_length |

**Code属性的属性表**（Class文件中最重要的属性）

|  |  |  |  |
| --- | --- | --- | --- |
| Code属性表的结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | 是一个指向CONSTANT  \_Utf8\_info型常量的索引，常量值固定为“Code” |
| u4 | attribute\_length | 1 | 属性值长度，固定为：  属性表长度 – 6 bytes |
| u2 | max\_stack | 1 | 操作数栈深度的最大值 |
| u2 | max\_locals | 1 | 表示局部变量表所需的存储空间。单位为Slot。 |
| u4 | code\_length | 1 | java源程序编译后的字节码长度 |
| u1 | code | code\_length | 字节码指令的一系列字节流 |
| u2  **非**  **必**  **须** | exception\_table\_length | 1 |  |
| exception\_info | exception\_table | exception\_table\_length |  |
| u2 | attributes\_count | 1 |  |
| attribute\_info | attributes | attributes\_count |  |

注：（1）max\_stack，即操作数栈深度的最大值。在方法执行的任意时刻，操作数栈都不会超过这个深度。虚拟机运行时根据这个值来分配**栈帧**（Stack Frame）**中的操作栈**深度。

（2）max\_locals，表示局部变量表所需的存储空间。单位为Slot。**Slot是虚拟机为局部变量分配内存所使用的最小单位**，对于byte、char、float、int、short、boolean和returnAddress等长度不超过32位的数据类型，每个局部变量占用1个Slot，而double和long这两种64位的数据类型则需要2个Slot来存放。方法参数（包括实例方法中的**隐藏参数（“this”）**、显式异常处理器的参数（Exception Handler Parameter，就是try-catch语句中**catch块所定义的异常**）、方法体中定义的**局部变量**都需要使用局部变量表来存放。另外局部变量表的Slot可以重用，当代码执行超出一个局部变量的作用域时，这个局部变量所占的Slot可以被其它局部变量所使用。javac编译器会根据变量的作用域来分配Slot给各个变量使用，然后计算出max\_locals的大小。

（3）code，表示字节码指令的一系列字节流。称为字节码指令，就是指每条指令就是一个u1类型的单字节。虚拟机毎读取code中的一个字节码，就可以对应找出这个字节码代表的是什么指令，并且可以知道这条指令后面是否需要跟随参数，以及参数应当如何理解。u1数据类型的取值范围为[0x00，0xFF]，即[0,255]，所以可以表示256条指令。当前JVM规范定义了约200条编码值对应的指令含义。值得注意的是u4类型的code\_length实际只能使用u2的长度，因为JVM规范明确限制一个方法不允许超过65535条字节码指令。如果超过这个限制，javac编译器也会拒绝编译。

Code属性是Class文件中最重要的一个属性，如果把一个Java程序中的信息分为代码（Code，方法体里面的Java代码）和元数据（Metadata，包括类、字段、方法定义及其他信息）两部分，那么在整个Class文件中，Code属性用于描述代码，所有的其他数据项目用于描述元数据。

**方法表集合**

**<init>()方法表** **init()方法表的属性表集合**

const #7 “<init>” const #8 “()V” const #9 “**Code**”

addr = 0x00000100 … **Code属性表**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **00** | **00** | **02** | **00** | **01** | **00** | **07** | **00** | **08** | **00** | **01** | **00** | **09** | **00** | **00** | **00** |
|  | 容量计数器  method\_cou  nt  = 0x0002  即有2个方  法 | | access\_flags  =0x0001,  public | | name\_index  =0x0007  “<init>” | | descriptor  \_index  =0x0008  “()V” | | attributes\_co  unt = 0x0001  即该方法的属性表集合有一项属性 | | attributes\_na  me\_index =  0x0009  “**Code**”说明此属性是方法的字节码描述 | | attribute\_length = 0x0000002F  = 47 bytes  属性值的长度为47 bytes | | |

**字节码指令**

addr = 0x00000110 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **2F** | **00** | **01** | **00** | **01** | **00** | **00** | **00** | **05** | **2A** | **B7** | **00** | **0A** | **B1** | **00** | **00** |
|  | max\_stack  = 0x0001 | | max\_locals  = 0x0001 | | code\_length = 0x00000005  字节码区域长度为5，虚拟机按照顺序依次读入5个字节 | | | | code  =2A B7 00 0A B1 | | | | |  |  |

0x2A: aload\_0 指令 0xB7: invokespecial指令 u2参数：0x000A: 0xB1:return void

const #10: CONSTANT\_Methodref\_info常量

“<init>”

addr = 0x00000120 …

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **00** | **02** | **00** | **0C** | **00** | **00** | **00** | **0E** | **00** | **01** | **00** | **00** | **0C** | **03** | **00** | **0D** |
|  |  | |  | |  | | | |  | | | | |  |  |

可以发现字节码执行过程中的数据交换、方法调用等操作都是基于栈（操作栈）的。所以，Java虚拟机执行字节码是基于栈的体系结构

检验：

javac:编译源程序生成字节码

javap:反编译工具，根据class字节码文件，反解析出当前类对前类对应的常量池、code区属性表（汇编指令）、本地变量表、异常表和代码行偏移量映射表等等。

源代码: javac编译下述代码：

package org.fenixsoft.clazz;

public class TestClass {

private int m;

public int inc(){

return m + 1;

}

}

命令:javap –verbose TestClass

//… 在这里省略常量池的输出

{

public org.fenixsoft.clazz.TestClass(); this关键字

Code:

Stack=1, Locals=1, Args\_size=1

0: aload\_0

1: invokespecial #10; //Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 3: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lorg/fenixsoft/clazz/TestClass;

**表示int()方法参数个数。无static修饰方法，则有this参数传入，**

**值至少为1；如有用static修饰方法，则无Args\_size值至少为0**

public int inc();

Code:

Stack=2, **Locals**=**1**, **Args\_size**=**1**

0: aload\_0

1: getfield #18; //Field m:I

4: iconst\_1

5: iadd

6: ireturn

**表示int()方法内部的局部变量个数。由于javac编译时会把this关键字的访问转变为对一个普通方法参数的访问，然后在虚拟机调用实例方法时自动传入此参数。因此在实例方法的局部变量表中至少会存在一个指向当前对象实例的局部变量，局部变量表也会预留出第一个Slot位来存放对象实例的引用，方法参数值从1开始计算。**

LineNumberTable:

line 8: 0

LocalVariableTable:

Start Length Slot Name Signature

0 7 0 this Lorg/fenixsoft/clazz/TestClass;

}

在字节码指令之后的是这个方法的显式异常处理表（下文简称异常表）集合。异常表对于Code属性来说并不是必须存在的。

属性表格式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性表结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | start\_pc | 1 | 当字节码在第start\_pc行到第end\_pc之间（不含第end\_pc行）出现了类型为catch\_type（catch\_type为指向一个CONSTANT\_Class\_info型常量的索引）或者其子类的异常，则转到第handler\_pc行继续处理。当catch\_type的值为0时，代表任意异常情况都需要转向到handler\_pc处进行处理。 |
| u2 | end\_pc | 1 |
| u2 | handler\_pc | 1 |
| u2 | catch\_type | 1 |

虽然异常也是java代码的一部分，但是如今javac编译器使用异常表而不是简单的跳转命令来实现Java异常及finally处理机制。

检验：

package org.fenixsoft.clazz;

public class TestException {

public int inc() {

int x;

try {

x = 1;

return x;

} catch (Exception e) {

x = 2;

return x;

} finally {

x = 3;

}

}

}

注：局部变量表存放如下：

数组元素[0]对应 ：变量实例对象this

数组元素[1]对应 ：变量x

数组元素[2]对应 ：returnValue、e

数组元素[3]对应 ：returnValue2

数组元素[4]对应 ：不属于java.lang.Excption及其子类的异常的引用

//编译后的code字节码以及异常表

public int inc();

Code:

Stack=1, Locals=5, Args\_size=1

0: iconst\_1 //将int型常量1压入操作数栈（try块中的 x=1）

1: istore\_1 //将操作数栈顶int型元素出栈，并保存到局部变量表的第2个位置

2: iload\_1 //将局部变量表第2个位置的int型元素入栈

3: istore\_2 //将操作数栈顶int型元素出栈，并保存到局部变量表第3个位置 returnValue

**出现其他**

**异常情况**

**任一位置出现指定异常**

**没有异常**

4: iconst\_3 //将int型常量3压入操作数栈（finally块中的 x = 3）

**执行finally并return**

5: istore\_1 //将操作数栈顶int型元素出栈，并保存到局部变量表的第2个位置

6: iload\_2 //将局部变量表第3个位置的int型元素入栈（将returnValue中的值放到栈顶，准备

给ireturn返回）

注：这就是为什么try和catch语句块中的返回值会被保留下来，即使finally语句块修改了返回值的大小。

7: ireturn //返回栈顶int元素（从当前方法返回int，执行完finally后才真正返回）

8: astore\_2 //将栈顶引用型数值出栈，并保存到局部变量表的第3个位置（给catch中定义的Exception e赋值，存储在Slot 2 中）

9: iconst\_2 //将int型常量2压入操作数栈（catch块中的 x = 2）

10: istore\_1 //将操作数栈顶int型元素出栈，并保存到局部变量表的第2个位置

11: iload\_1 //将局部变量表第2个位置的int型元素入栈

12: istore\_3 //将操作数栈顶int型元素出栈，并保存到局部变量表的第4个位置 returnValue2

13: iconst\_3 //将int型常量3压入操作数栈（finally块中的 x = 3）

**执行finally并return**

14: istore\_1 //将操作数栈顶int型元素出栈，并保存到局部变量表的第2个位置

15: iload\_3 //将局部变量表第4个位置的int型元素入栈（returnValue中的值放到栈顶，准备给

ireturn返回）

16: ireturn //返回栈顶int元素（从当前方法返回int，执行完finally后才真正返回）

**出现其他**

**异常情况**

17: astore 4 //将栈顶引用型数值出栈，并保存到局部变量表的第5个位置（如果出现了不属于java.lang.Exception及其子类的异常才会走到这里）

19: iconst\_3 //将int型常量3压入操作数栈（finally块中的 x = 3）

**执行finally并抛出异常**

20: istore\_1 //将操作数栈顶int型元素出栈，并保存到局部变量表的第2个位置

21: aload 4 //将局部变量表第5个位置的引用类型（异常）入栈

23: athrow //抛出异常

Exception table: //异常表记录

from to target type

0 4 8 Class java/lang/Exception

//执行路径1：如果try语句块中出现Exception或其子类的异常，则转到catch语句块处理

0 4 17 any

//执行路径2：如果try语句块中出现不属于Exception或其子类的异常，则转到finally语句块处理

8 13 17 any

//执行路径3：如果catch语句块中出现任何异常，则转到finally语句块处理

Exceptions属性

这里的Exceptions属性是在方法表中与Code属性平级的一项属性。Exceptions属性的作用是列举出方法中可能抛出的受查异常（Checked Exceptions），也就是方法描述时**在throws关键字后面列举的异常**。

Exceptions属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性表结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | 表示方法可能抛出number\_of\_exceptions种受查异常，每一种受查异常使用一个exception\_index\_table项表示。  是一个指向常量池中CONSTANT\_Class\_info型常量的索引，代表了该受查异常的类型。 |
| u4 | attribute\_length | 1 |
| u2 | number\_of\_exceptions | 1 |
| u2 | exception\_index\_table | number\_of\_exceptions |

LineNumberTable属性

LineNumberTable属性用于描述Java源码行号与字节码行号（字节码的偏移量）之间的对应关系。非必须，可在javac编译时使用-g:none或-g:lines选项来取消或要求生成这项信息。如果取消生成，当程序抛出异常时，堆栈不会出现行号，并且在调试程序时，也无法按照源码行来设置断点。

LineNumberTable属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| LineNumberTable属性结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | line\_number\_table是一个数量为line\_number\_table\_length、类型为line\_number\_info的集合，line\_number\_info表包括了start\_pc和line\_number两个u2类型的数据项，前者是字节码行号，后者是Java源码行号。 |
| u4 | attribute\_length | 1 |
| u2 | line\_number\_table\_length | 1 |
| line\_number\_info | line\_number\_table | line\_number\_table\_length |

LocalVariable Table属性

LocalVariable Table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，非必须属性。可在javac中分别使用-g:none或-g:vars选项来取消或要求生成这项信息。如果没有生成这项属性，最大的影响就是当其他人引用这个方法是，所有的参数名称都将会丢失，IDE将会使用诸如arg0，arg1之类的占位符代替原有的参数名。另外在调试期间无法根据参数名称从上下文中获得参数值。

LocalVariable Table属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| LocalVariableTable属性结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | Local\_variable\_info项目代表了一个栈帧与源码中的局部变量的关联 |
| u4 | attribute\_length | 1 |
| u2 | local\_variable\_table\_length | 1 |
| local\_variable\_info | local\_variable\_table | local\_variable\_table\_length |

local\_variable\_info项目结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| local\_variable\_info项目结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | start\_pc | 1 | start\_pc和length属性表示该局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。 |
| u2 | length | 1 |
| u2 | name\_index | 1 | name\_index和descriptor\_index都是指向常量池CONSTANT\_Utf8\_info型常量的索引，分别代表该局部变量的名称以及这个局部变量的描述符 |
| u2 | descriptor\_index | 1 |
| u2 | index | 1 | index是这个局部变量在栈帧局部变量表中Slot的位置。当这个变量数据类型为64位类型（double/long），它占用的Slot为index和index + 1 |

注：JDK1.5引入泛型之后，LocalVariable属性增加了一个“姐妹属性”：LocalVariableTypeTable，这个新增属性结构与LocalVariableTable非常类似，仅仅将记录的字段描述符descriptor\_index替换成了字段的特征签名（Signature），对于非泛型类型，描述符和特征签名能描述的信息是基本一致的，但是泛型引入之后，由于描述符中泛型的参数化类型被擦除掉，描述符就不能准确地描述泛型类型，因此引入LocalVariableTypeTable。

SourceFile属性

SourceFile属性用于记录生成这个Class文件的源码文件名称。该属性非必须，是一个定长类型。可以使用javac的-g:none或-g:source选项来关闭或要求生成这项信息。在Java中对大多数类来说，类名和文件名是一致的，但也有特殊情况（如内部类）。如果不记录这项属性，当抛出异常时，堆栈将不会显示出错代码所属的文件名。

SourceFile属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| SourceFile属性结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | sourcefile\_index数据项指向常量池中CONSTANT\_Utf8\_info型常量的索引，常量值是源码文件的文件名 |
| u4 | attribute\_length | 1 |
| u2 | sourcefile\_index | 1 |

ConstantValue属性

ConstantValue属性的作用是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。

例如实例变量“int x = 123”和类变量“static int x = 123”，虚拟机对这两种变量赋值的方式和时刻都是不同的。对于实例变量，赋值是在实例构造器<init>方法中进行的；

而对于类变量，则有两种方式可以选择：

① 在类构造器<clinit>方法中

② 使用ConstantValue属性

目前Sun javac编译器的做法是：

如果同时使用final和static来修饰变量（常量），并且这个变量的数据类型是基本数据类型或者java.lang.String的话，就生成ConstantValue属性来进行初始化；如果这个变量没有被final修饰，或者并非为基本数据类型及字符串，则会选择在<clinit>方法中进行初始化。

至于为什么ConstantValue属性必须限制基本类型和String才能使用，这是因为ConstantValue属性的属性值只是一个常量池的索引号，而常量池中的常量类型只有与基本属性和字符串相对应的字面量，所以ConstantValue属性也无法支持别的类型（比如对象类型）

ConstantValue属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| ConstantValue属性结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 |  |
| u4 | attribute\_length | 1 | 值固定为2 |
| u2 | constantvalue\_index | 1 | 代表常量池中一个字面量常量的引用，可以是CONSTANT\_Long\_info、CONSTANT\_Float\_info、CONSTANT\_Double\_info、  CONSTANT\_Integer\_info、CONSTANT\_String\_info常量中的一种 |

InnerClasses属性

InnerClasses属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那编译器将会为它以及它所包含的内部类生成InnerClasses属性。

InnerClasses属性表结构如下：

|  |  |  |
| --- | --- | --- |
| InnerClasses属性结构 | | |
| 类型 | 名称 | 数量 |
| u2 | attribute\_name\_index | 1 |
| u4 | attribute\_length | 1 |
| u2 | number\_of\_classes | 代表需要记录多少个内部类信息，每一个内部类的信息都由一个inner\_classes\_info表进行描述 |
| inner\_classes\_info | inner\_classes | number\_of\_classes |

inner\_classes\_info表结构如下

|  |  |  |  |
| --- | --- | --- | --- |
| inner\_classes\_info表结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | inner\_class\_info\_index | 1 | 指向常量池中CONSTANT\_Class\_info型常量的索引，代表内部类的符号引用 |
| u2 | outer\_class\_info\_index | 1 | 指向常量池中CONSTANT\_Class\_info型常量的索引，代表宿主类的符号引用 |
| u2 | inner\_name\_index | 1 | 指向常量池中CONSTANT\_Utf8\_info型常量的索引，代表这个内部类的名称，如果是匿名类，那么这项值为0 |
| u2 | inner\_class\_access\_flags | 1 | 内部类的访问标志，类似于类的access\_flag  s，它的取值范围 |

|  |  |  |
| --- | --- | --- |
| inner\_class\_access\_flags标志 | | |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 内部类是否被public |
| ACC\_PRIVATE | 0x0002 | 内部类是否被private |
| ACC\_PROTECTED | 0x0004 | 内部类是否被protected |
| ACC\_STATIC | 0x0008 | 内部类是否被static |
| ACC\_FINAL | 0x0010 | 内部类是否被final |
| ACC\_INTERFACE | 0x0020 | 内部类是否为synchronized |
| ACC\_ABSTRACT | 0x0400 | 内部类是否为abstract |
| ACC\_SYNTHETIC | 0x1000 | 内部类是否并非由用户代码产生的 |
| ACC\_ANNOTATION | 0x2000 | 内部类是否是一个注解 |
| ACC\_ENUM | 0x4000 | 内部类是否是一个枚举 |

Deprecated及Synthetic属性

Deprecated和Synthetic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属性用于表示某个类、字段或者方法，已经被程序作者定为不再推荐使用，它可以通过在代码中使用@deprecated注释进行设置。

Synthetic属性代表此字段或者方法并不是由Java源码产生的，而是由编译器自行添加的。在JDK1.5之后，标识一个类、字段或者方法是由编译器自动产生的，也可以设置它们访问标志中的ACC\_SYNTHETIC标志位，其中最典型的例子就是Bridge Method。所有由非用户代码产生的类、方法及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器“<init>”方法和类构造器“<clinit>”方法。

Deprecated及Synthetic属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Deprecated及Synthetic属性的结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 |  |
| u4 | attribute\_length | 1 | 0x00000000，因为没有任何属性值需要设置 |

StackMapTable属性

StackMapTable属性在JDK1.6发布后增加到了Class文件规范中，它是一个复杂的变长属性，位于Code属性的属性表中。这个属性会在虚拟机类加载的字节码验证阶段被**新类型检查验证器（Type Checker）**使用。该验证器在保证Class文件合法性的前提下，省略了在运行期通过数据流分析去确认字节码的行为逻辑合法性的步骤，而在编译阶段将一系列的验证类型（Verification Types）直接记录在Class文件之中，通过检查这些验证类型代替了之前传统的类型推导过程，从而大幅度提升了字节码验证的性能。

StackMapTable属性中包含零至多个**栈映射帧（Stack Map Frames）**，每个栈映射帧都显式或隐式地**代表了一个字节码偏移量**，用于**表示执行到该字节码时局部变量表和操作数栈的验证类型**。类型检查验证器会通过检查目标方法的局部变量和操作数栈所需要的类型来确定一段字节码指令是否符合逻辑约束。

StackMapTable属性表结构如下：

|  |  |  |
| --- | --- | --- |
| StackMapTable属性的结构 | | |
| 类型 | 名称 | 数量 |
| u2 | attribute\_name\_index | 1 |
| u4 | attribute\_length | 1 |
| u2 | number\_of\_entries | 1 |
| stack\_map\_frame | stack\_map\_frame\_entries | number\_of\_entries |

在版本号大于或等于50.0的Class文件中，如果文件的Code属性中没有附带StackMapTable属性，则意味着它带有一个隐式的StackMap属性。这个StackMap属性的作用等同于number\_of\_entries值为0的StackMapTable属性。一个方法的Code属性最多只能带有一个StackMapTable，否则将抛出ClassFormatError异常。

Signature属性

Signature属性在JDK1.5发布后增加到了Class文件规范中，它是一个可选的定长属性，可以出现于类、属性表和方法表结构的属性表中。在JDK1.5中大幅增强了Java语言的语法，在此之后，任何类、接口、初始化方法或者成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则Signature属性会为它记录泛型签名信息。之所以要专门使用这样的一个属性去记录泛型类型，是因为Java语言的泛型采用的是擦除法实现的伪类型，在字节码（Code属性）中，泛型信息编译（类型变量、参数化类型）之后都通通被擦除掉。使用擦触法的好处是实现简单 （主要修改Javac编译器，虚拟机内部只做了很少的改动）、非常容易实现Backport，运行期也能够节省一些类型所占的内存空间。但坏处是运行期就无法将泛型类型与用户定义的普通类型同等对待，例如运行期做反射时无法获得到泛型信息。因此需要增设Signature属性解决这个缺陷。现在Java反射API能够获取泛型类型，最终的数据来源也是因为这个属性。

Signature属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Signature属性的结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 | signature\_index是对常量池的有效索引，指向常量类型CONSTANT\_Utf  8\_info，表示类签名、方法类型签名或字段类型签名。 |
| u4 | attribute\_length | 1 |
| u2 | signature\_index | 1 |

如果当前Signature属性是类文件的属性，则这个结构表示类签名，如果当前的Signature属性是方法表的属性，则这个结构表示方法类型签名；如果当前Signature属性是字段表的属性，则这个结构表示字段类型签名。

BootstrapMethods属性

BootstrapMethods属性在JDK1.7发布后增加到了Class文件规范中，它是一个可选的变长属性，位于类的属性表中。这个属性用于保存invokedynamic指令引用的引导方法限定符。《Java虚拟机规范（Java SE 7 版）》规定，如果某个类文件结构的常量池中曾经出现过CONSTANT\_InvokeDynamic\_info类型的常量，那么这个类文件的属性表中必须存在一个明确的BootstrapMethods属性，并且最多也只能够有一个BootstrapMethods属性。该属性与InvokeDynamic指令和java.lang.Invoke包关系非常密切。

BootstrapMethods属性表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| BootstrapMethods属性的结构 | | | |
| 类型 | 名称 | 数量 | 作用 |
| u2 | attribute\_name\_index | 1 |  |
| u4 | attribute\_length | 1 |  |
| u2 | num\_bootstrap\_methods | 1 | 给出bootstrap\_methods[]数组中的引导方法限定符的数量 |
| bootstrap\_method | bootstrap\_methods | num\_bootstrap\_methods | 指向常量池CONSTANT\_  MethodHandle\_info结构的索引值，代表一个引导方法，还包含了这个引导方法静态参数的序列（可能为空）。 |

bootstrap\_methods[]数组中的每个成员必须包含以下3项内容

（1）bootstrap\_method\_ref

bootstrap\_method\_ref项的值必须是一个对常量池的有效索引，常量池在该索引处的值必须是一个CONSTANT\_MethodHandle\_info结构。

（2）num\_bootstrap\_arguments

num\_bootstrap\_arguments项的值给出了bootstrap\_arguments[]数组成员的数量。

（3）bootstrap\_arguments[]

bootstrap\_arguments[]数组额的每个成员必须是一个对常量池的有效索引。常量池在该索引处必须是下列结构之一：

①CONSTANT\_String\_info ②CONSTANT\_Class\_info

③CONSTANT\_Integer\_info ④CONSTANT\_Long\_info

⑤CONSTANT\_Float\_info ⑥CONSTANT\_Double\_info

⑦CONSTANT\_MethodHandle\_info/CONSTANT\_MethodType\_info

补充：关于字节码指令

① Java虚拟机采用面向操作数栈而不是寄存器的架构，所以JVM的大多数指令都不包含操作数，只有一个操作码（Opcode）

② 限制JVM操作码长度为单字节（0~255），所以指令集的操作码总数不超过256条

③ Class文件格式放弃了编译后代码的操作数长度对齐，因此JVM处理超过一个字节的操作数时，就必须在运行时从字节中重建出具体数据的结构。比如要将一个16位长度的无符号整数使用两个无符号字节存储起来（命名为byte1和byte2），则重构这个操作数的值则必须进行如右计算：(byte << 8) | byte2。这会导致解释执行字节码有一定性能损失，但是放弃操作数长度对齐，也意味着可以省略很多填充和间隔符号。

④ JVM的解释器若不考虑异常处理，可以用如下伪代码作为基本执行模型理解：

do {

自动计算PC寄存器的值+1;

根据PC寄存器的指示位置，从字节码流中取出操作码;

if (字节码存在操作数)

从字节码流中取出操作数;

执行操作码所定义的操作

} while (字节码流长度 > 0);

⑤ JVM可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构是使用管程（Monitor）来支持的。

方法级的同步是隐式的，即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之中。虚拟机可以从方法表结构中的ACC\_SYNCHRONIZED访问标志得知一个方法是否声明为同步方法。当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED访问标志是否被设置，如果设置了，执行线程就要求先成功持有管程，然后才能执行方法，最后当方法完成（无论正常完成还是非正常完成）时释放管程。在方法执行期间，执行线程持有了管程，其他任何线程都无法再获取到同一个管程。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放。

同步一段指令集序列通常由Java语言中synchronized语句块来表示，Java虚拟机指令集中有monitorenter和monitorexit两条指令来支持synchronized关键字的语义，正确实现synchronized关键字需要Javac编译器与Java虚拟机两者共同协作支持。

测试代码：

package org.fenixsoft.clazz;

public class testMonitor {

public void doSomething(){};

void onlyMe (testMonitor f){

synchronized (f) {

doSomething();

}

}

}

onlyMe()方法编译后的字节码序列如下：

void onlyMe(org.fenixsoft.clazz.testMonitor);

Code:

Stack=2, Locals=4, Args\_size=2

0: aload\_1 //对象引用f入栈

1: dup //复制栈顶元素（f的引用）并压栈

2: astore\_2 //将栈顶元素（f的复制）出栈并存储到局部变量表Slot2

3: monitorenter //将栈顶元素f作为锁，开始同步

4: aload\_0 //将局部变量Slot0（this指针）元素入栈

5: invokevirtual #2; //Method doSomething:()V //调用doSomething()方法

8: aload\_2 //将局部变量Slow2元素入栈

9: monitorexit //退出同步

10: goto 18 //方法正常结束，跳转到18返回

13: astore\_3 //从这里开始是异常路径，见异常表的Type 13

14: aload\_2 //将局部变量Slow2的元素（即f）入栈

15: monitorexit //退出同步

16: aload\_3 //将异常对象Slow3元素入栈

17: athrow //把异常对象重新抛出给onlyMe()方法的调用者

18: return //方法正常返回

Exception table:

from to target type

4 10 13 any

13 16 13 any