**理解JAVA Class文件，破解class文件的第一步**

<https://blog.csdn.net/tyyj90/article/details/78472986>

1.概述

Java虚拟机中定义的Class文件格式。每一个Class文件都对应着唯一一个类或接口的定义信息，但是相对地，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。我们只是通俗地将任意一个有效的类或接口所应当满足的格式称为“Class文件格式”，即使它不一定以磁盘文件的形式存在。

每个Class文件都是由8字节为单位的字节流组成，所有的16位、32位和64位长度的数据将被构造成 2个、4个和8个8字节单位来表示。多字节数据项总是按照Big-Endian（Big-Endian顺序是指按高位字节在地址最低位，最低字节在地址最高位来存储数据，它是SPARC、PowerPC等处理器的默认多字节存储顺序，而x86等处理器则是使用了相反的Little-Endian顺序来存储数据。为了保证Class文件在不同硬件上具备同样的含义，因此在Java虚拟机规范中是有必要严格规定了数据存储顺序的。）的顺序进行存储。

便于描述u1，u2和u4，分别代表了1、2和4个字节的无符号数。Class文件格式采用类似C语言结构体的伪结构来描述Class文件格式。为了避免与类的字段、类的实例等概念产生混淆，把用于描述类结构格式的内容定义为项（Item）。在Class文件中，各项按照严格顺序连续存放的，它们之间没有任何填充或对齐作为各项间的分隔符号。

表（Table）是由任意数量的可变长度的项组成，用于表示Class文件内容的一系列复合结构。尽管我们采用类似C语言的数组语法来表示表中的项，但是我们应当清楚意识到，表是由可变长数据组成的复合结构（表中每项的长度不固定），因此无法直接将字节偏移量来作为索引对表进行访问。而我们描述一个数据结构为数组（Array）时，就意味着它含有零至多个长度固定的项组成，这个时候则可以采用数组索引的方式来访问它。

2.ClassFile结构

每一个Class文件对应于一个如下所示的ClassFile结构体。

ClassFile {

u4 magic;

u2 minor\_version;

u2 major\_version;

u2 constant\_pool\_count;

cp\_info constant\_pool[constant\_pool\_count-1];

u2 access\_flags;

u2 this\_class;

u2 super\_class;

u2 interfaces\_count;

u2 interfaces[interfaces\_count];

u2 fields\_count;

field\_info fields[fields\_count];

u2 methods\_count;

method\_info methods[methods\_count];

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

ClassFile结构体中，各项的含义描述如下：

magic

魔数，魔数的唯一作用是确定这个文件是否为一个能被虚拟机所接受的Class文件。魔数值固定为0xCAFEBABE，不会改变。

minor\_version、major\_version

副版本号和主版本号，minor\_version和major\_version的值分别表示Class文件的副、主版本。它们共同构成了Class文件的格式版本号。譬如某个Class文件的主版本号为M，副版本号为m，那么这个Class文件的格式版本号就确定为M.m。Class文件格式版本号大小的顺序为：1.5 < 2.0 < 2.1。

一个Java虚拟机实例只能支持特定范围内的主版本号（Mi至Mj）和0至特定范围内（0至m）的副版本号。假设一个Class文件的格式版本号为V，仅当Mi.0 ≤ v ≤Mj.m成立时，这个Class文件才可以被此Java虚拟机支持。不同版本的Java虚拟机实现支持的版本号也不同，高版本号的Java虚拟机实现可以支持低版本号的Class文件，反之则不成立。

constant\_pool\_count

常量池计数器，constant\_pool\_count的值等于constant\_pool表中的成员数加1。constant\_pool表的索引值只有在大于0且小于constant\_pool\_count时才会被认为是有效的（虽然值为0的constant\_pool索引是无效的，但其他用到常量池的数据结构可以使用索引0来表示“不引用任何一个常量池项”的意思），对于long和double类型有例外情况。

constant\_pool[]

常量池，constant\_pool是一种表结构，它包含Class文件结构及其子结构中引用的所有字符串常量、类或接口名、字段名和其它常量。常量池中的每一项都具备相同的格式特征——第一个字节作为类型标记用于识别该项是哪种类型的常量，称为“tag byte”。常量池的索引范围是1至constant\_pool\_count−1。

access\_flags

访问标志，access\_flags是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。access\_flags的取值范围和相应含义见下表。

标记名 值 含义

ACC\_PUBLIC 0x0001 可以被包的类外访问。

ACC\_FINAL 0x0010 不允许有子类。

ACC\_SUPER 0x0020 当用到invokespecial指令时，需要特殊处理的父类方法。

ACC\_INTERFACE 0x0200 标识定义的是接口而不是类。

ACC\_ABSTRACT 0x0400 不能被实例化。

ACC\_SYNTHETIC 0x1000 标识并非Java源码生成的代码。

ACC\_ANNOTATION 0x2000 标识注解类型

ACC\_ENUM 0x4000 标识枚举类型

1. 带有ACC\_SYNTHETIC标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。

带有ACC\_ENUM标志的类，意味着它或它的父类被声明为枚举类型。

带有ACC\_INTERFACE标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个Class文件被设置了ACC\_INTERFACE标志，那么同时也得设置ACC\_ABSTRACT标志。同时它不能再设置ACC\_FINAL、ACC\_SUPER 和 ACC\_ENUM标志。

注解类型必定带有ACC\_ANNOTATION标记，如果设置了ANNOTATION标记，ACC\_INTERFACE也必须被同时设置。如果没有同时设置ACC\_INTERFACE标记，那么这个Class文件可以具有表中的除ACC\_ANNOTATION外的所有其它标记。当然ACC\_FINAL和ACC\_ABSTRACT这类互斥的标记除外。

ACC\_SUPER标志用于确定该Class文件里面的invokespecial指令使用的是哪一种执行语义。目前Java虚拟机的编译器都应当设置这个标志。ACC\_SUPER标记是为了向后兼容旧编译器编译的Class文件而存在的，在JDK1.0.2版本以前的编译器产生的Class文件中，access\_flag里面没有ACC\_SUPER标志。同时，JDK1.0.2前的Java虚拟机遇到ACC\_SUPER标记会自动忽略它。

在表中没有使用的access\_flags标志位是为未来扩充而预留的，这些预留的标志为在编译器中会被设置为0， Java虚拟机实现也会自动忽略它们。

this\_class

类索引，this\_class的值必须是对constant\_pool表中项目的一个有效索引值。constant\_pool表在这个索引处的项必须为CONSTANT\_Class\_info类型常量，表示这个Class文件所定义的类或接口。

super\_class

父类索引，对于类来说，super\_class的值必须为0或者是对constant\_pool表中项目的一个有效索引值。如果它的值不为0，那constant\_pool表在这个索引处的项必须为CONSTANT\_Class\_info类型常量，表示这个Class文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的access\_flag中都不能带有ACC\_FINAL标记。对于接口来说，它的Class文件的super\_class项的值必须是对constant\_pool表中项目的一个有效索引值。constant\_pool表在这个索引处的项必须为代表java.lang.Object的CONSTANT\_Class\_info类型常量。如果Class文件的super\_class的值为0，那这个Class文件只可能是定义的是java.lang.Object类，只有它是唯一没有父类的类。

interfaces\_count

接口计数器，interfaces\_count的值表示当前类或接口的直接父接口数量。

interfaces[]

接口表，interfaces[]数组中的每个成员的值必须是一个对constant\_pool表中项目的一个有效索引值，它的长度为interfaces\_count。每个成员interfaces[i] 必须为CONSTANT\_Class\_info类型常量，其中0 ≤ i < interfaces\_count。在interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即interfaces[0]对应的是源代码中最左边的接口。

fields\_count

字段计数器，fields\_count的值表示当前Class文件fields[]数组的成员个数。fields[]数组中每一项都是一个field\_info结构的数据项，它用于表示该类或接口声明的类字段或者实例字段。

fields[]

字段表，fields[]数组中的每个成员都必须是一个fields\_info结构的数据项，用于表示当前类或接口中某个字段的完整描述。fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。

methods\_count

方法计数器，methods\_count的值表示当前Class文件methods[]数组的成员个数。Methods[]数组中每一项都是一个method\_info结构的数据项。

methods[]

方法表，methods[]数组中的每个成员都必须是一个method\_info结构的数据项，用于表示当前类或接口中某个方法的完整描述。如果某个method\_info结构的access\_flags项既没有设置ACC\_NATIVE标志也没有设置ACC\_ABSTRACT标志，那么它所对应的方法体就应当可以被Java虚拟机直接从当前类加载，而不需要引用其它类。method\_info结构可以表示类和接口中定义的所有方法，包括实例方法、类方法、实例初始化方法方法和类或接口初始化方法方法。methods[]数组只描述当前类或接口中声明的方法，不包括从父类或父接口继承的方法。

attributes\_count

属性计数器，attributes\_count的值表示当前Class文件attributes表的成员个数。attributes表中每一项都是一个attribute\_info结构的数据项。

attributes[]

属性表，attributes表的每个项的值必须是attribute\_info结构。在Class文件规范里，Class文件结构中的attributes表的项包括下列定义的属性：InnerClasses、EnclosingMethod、Synthetic、Signature、SourceFile，SourceDebugExtension、Deprecated、RuntimeVisibleAnnotations、RuntimeInvisibleAnnotations以及BootstrapMethods属性。对于支持Class文件格式版本号为49.0或更高的Java虚拟机实现，必须正确识别并读取attributes表中的Signature、RuntimeVisibleAnnotations和 RuntimeInvisibleAnnotations属性。对于支持Class文件格式版本号为51.0或更高的Java虚拟机实现，必须正确识别并读取attributes表中的BootstrapMethods属性。Class文件规范要求任一Java虚拟机实现可以自动忽略Class文件的attributes表中的若干（甚至全部）它不可识别的属性项。任何Class文件规范未定义的属性不能影响Class文件的语义，只能提供附加的描述信息。

3.描述符

3.1语法符号

描述符和签名都是用特定的语法符号（Grammar）来表示，这些语法是一组可表达如何根据不同的类型去产生可恰当描述它们的字符序列的标识集合。例如：

FieldType:

BaseType

ObjectType

ArrayType

1

2

3

4

上面文字表达的意思是FileType可以表示BaseType、ObjectType、ArrayType三者之一。

当一个有星号（\*）跟随的非终止符出现在一个语法标识的右侧时，说明带有这个非终止符的语法标识将产生0或多个不同值，这些值按照顺序且无间隔的排列在非终止符后面。当一个有加号（+）跟随的非终止符出现在一个结构的右侧时，说明这个非终止符将产生一个或多个不同值，这些值按照顺序且无间隔的排列在非终止符后面。例如：

MethodDescriptor:

( ParameterDescriptor\* ) ReturnDescriptor

1

2

上面文字表达的意思是MethodDescriptor的是由左括号“(”、0或若干个连续排列的ParameterDescriptor值、右括号“)”、ReturnDescriptor值构成。

3.2字段描述符

字段描述符（Field Descriptor），是一个表示类、实例或局部变量的语法符号，它是由语法产生的字符序列：

FieldDescriptor:

FieldType

ComponentType:

FieldType

FieldType:

BaseType

ObjectType

ArrayType

BaseType:

B

C

D

F

I

J

S

Z

ObjectType:

L Classname ;

ArrayType:

[ ComponentType

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

所有表示基本类型（BaseType）的字符、表示对象类型（ObjectType）中的字符”L”，表示数组类型（ArrayType）的”[“字符都是ASCII编码的字符。对象类型（ObjectType）中的Classname表示一个类或接口二进制名称的内部格式。表示数组类型的有效描述符的长度应小于等于255。所有字符类型的解释如下表所示。

字符 类型 含义

B byte 有符号字节型数

C char Unicode字符，UTF-16编码

D double 双精度浮点数

F float 单精度浮点数

I int 整型数

J long 长整数

S short 有符号短整数

Z boolean 布尔值 true/false

L Classname; reference 一个名为的实例

[ reference 一个一维数组

举个例子：描述int实例变量的描述符是“I”；java.lang.Object的实例描述符是“Ljava/lang/Object;”。注意，这里用到了类Object的二进制名的内部形式（此处是内部形式）。double的三维数组“double d[][][];”的描述符为“[[[D”。

3.3方法描述符

方法描述符（Method Descriptor）描述一个方法所需的参数和返回值信息：

MethodDescriptor:

( ParameterDescriptor\* ) ReturnDescriptor

1

2

参数描述符（ParameterDescriptor）描述需要传给这个方法的参数信息：

ParameterDescriptor:

FieldType

1

2

返回描值述符(ReturnDescriptor)从当前方法返回的值，它是由语法产生的字符序列：

ReturnDescriptor:

FieldType

VoidDescriptor

1

2

3

其中VoidDescriptor表示当前方法无返回值，即返回类型是void。符号如下（字符V即void）

VoidDescriptor:

V

1

2

如果一个方法描述符是有效的，那么它对应的方法的参数列表总长度小于等于255，对于实例方法和接口方法，需要额外考虑隐式参数this。参数列表长度的计算规则如下：每个long和double类参数长度为2，其余的都为1，方法参数列表的总长度等于所有参数的长度之和。

例如，方法：

Object mymethod(int i, double d, Thread t)

1

的描述符为：

(IDLjava/lang/Thread;)Ljava/lang/Object;

1

注意：这里使用了Object和Thread的二进制名称的内部形式。

无论mymethod()是静态方法还是实例方法，它的方法描述符都是相同的。尽管实例方法除了传递自身定义的参数，还需要额外传递参数this，但是这一点不是由法描述符来表达的。参数this的传递，是由Java虚拟机实现在调用实例方法所使用的指令中实现的隐式传递。

4.常量池

Java虚拟机指令执行时不依赖与类、接口，实例或数组的运行时布局，而是依赖常量池（constant\_pool）表中的符号信息。所有的常量池项都具有如下通用格式：

cp\_info {

u1 tag;

u1 info[];

}

1

2

3

4

常量池中，每个cp\_info项的格式必须相同，它们都以一个表示cp\_info类型的单字节“tag”项开头。后面info[]项的内容tag由的类型所决定。tag有效的类型和对应的取值在下表列出。每个tag项必须跟随2个或更多的字节，这些字节用于给定这个常量的信息，附加字节的信息格式由tag的值决定。

常量类型 值

CONSTANT\_Class 7

CONSTANT\_Fieldref 9

CONSTANT\_Methodref 10

CONSTANT\_InterfaceMethodref 11

CONSTANT\_String 8

CONSTANT\_Integer 3

CONSTANT\_Float 4

CONSTANT\_Long 5

CONSTANT\_Double 6

CONSTANT\_NameAndType 12

CONSTANT\_Utf8 1

CONSTANT\_MethodHandle 15

CONSTANT\_MethodType 16

CONSTANT\_InvokeDynamic 18

4.1 CONSTANT\_Class\_info结构

CONSTANT\_Class\_info结构用于表示类或接口，格式如下：

CONSTANT\_Class\_info {

u1 tag;

u2 name\_index;

}

1

2

3

4

CONSTANT\_Class\_info结构的项的说明：

tag CONSTANT\_Class\_info结构的tag项的值为CONSTANT\_Class（7）。

name\_index name\_index项的值，必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，代表一个有效的类或接口二进制名称的内部形式。

因为数组也是由对象表示，所以字节码指令anewarray和multianewarray可以通过常量池中的CONSTANT\_Class\_info结构来引用类数组。对于这些数组，类的名字就是数组类型的描述符，例如：

表现二维int数组类型

int[][]

1

的名字是：

[[I

1

表示一维Thread数组类型

Thread[]

1

的名字是：

[Ljava/lang/Thread;

1

一个有效的数组类型描述符中描述的数组维度必须小于等于255。

4.2 CONSTANT\_Fieldref\_info， CONSTANT\_Methodref\_info和CONSTANT\_InterfaceMethodref\_info结构

字段：

CONSTANT\_Fieldref\_info {

u1 tag;

u2 class\_index;

u2 name\_and\_type\_index;

}

1

2

3

4

5

方法：

CONSTANT\_Methodref\_info {

u1 tag;

u2 class\_index;

u2 name\_and\_type\_index;

}

1

2

3

4

5

接口方法：

CONSTANT\_InterfaceMethodref\_info {

u1 tag;

u2 class\_index;

u2 name\_and\_type\_index;

}

1

2

3

4

5

这些结构各项的说明如下：

tag

CONSTANT\_Fieldref\_info结构的tag项的值为CONSTANT\_Fieldref（9）。

CONSTANT\_Methodref\_info结构的tag项的值为CONSTANT\_Methodref（10）。

CONSTANT\_InterfaceMethodref\_info结构的tag项的值为CONSTANT\_InterfaceMethodref（11）。

class\_index

class\_index项的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\_Class\_info结构，表示一个类或接口，当前字段或方法是这个类或接口的成员。

CONSTANT\_Methodref\_info结构的class\_index项的类型必须是类（不能是接口）。CONSTANT\_InterfaceMethodref\_info结构的class\_index项的类型必须是接口（不能是类）。CONSTANT\_Fieldref\_info结构的class\_index项的类型既可以是类也可以是接口。

name\_and\_type\_index

name\_and\_type\_index项的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\_NameAndType\_info结构，它表示当前字段或方法的名字和描述符。

在一个CONSTANT\_Fieldref\_info结构中，给定的描述符必须是字段描述符。而CONSTANT\_Methodref\_info和CONSTANT\_InterfaceMethodref\_info中给定的描述符必须是方法描述符。

如果一个CONSTANT\_Methodref\_info结构的方法名以“<”（’\u003c’）开头，则说明这个方法名是特殊的，即这个方法是实例初始化方法，它的返回类型必须为空。

4.3 CONSTANT\_String\_info结构

CONSTANT\_String\_info用于表示java.lang.String类型的常量对象，格式如下：

CONSTANT\_String\_info {

u1 tag;

u2 string\_index;

}

1

2

3

4

CONSTANT\_String\_info结构各项的说明如下：

tag

CONSTANT\_String\_info结构的tag项的值为CONSTANT\_String（8）。

string\_index

string\_index项的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，表示一组Unicode码点序列，这组Unicode码点序列最终会被初始化为一个String对象。

4.4 CONSTANT\_Integer\_info和CONSTANT\_Float\_info结构

CONSTANT\_Intrger\_info和CONSTANT\_Float\_info结构表示4字节（int和float）的数值常量：

CONSTANT\_Integer\_info {

u1 tag;

u4 bytes;

}

CONSTANT\_Float\_info {

u1 tag;

u4 bytes;

}

1

2

3

4

5

6

7

8

这些结构各项的说明如下：

tag

CONSTANT\_Integer\_info结构的tag项的值是CONSTANT\_Integer（3）。

CONSTANT\_Float\_info结构的tag项的值是CONSTANT\_Float（4）。

bytes

CONSTANT\_Integer\_info结构的bytes项表示int常量的值，按照Big-Endian的顺序存储。

CONSTANT\_Float\_info结构的bytes项按照IEEE 754单精度浮点格式表示float常量的值，按照Big-Endian的顺序存储。

CONSTANT\_Float\_info结构表示的值将按照下列方式来表示，bytes项的值首先被转换成一个int常量的bits：

如果bits值为0x7f800000，表示float值为正无穷。

如果bits值为0xff800000，表示float值为负无穷。

如果bits值在范围0x7f800001到0x7fffffff或者0xff800001到0xffffffff内，表示float值为NaN。

在其它情况下，设s、e、m，它们值根据bits和如下公式计算：

int s =((bits >> 31) == 0) ? 1 : -1;

int e =((bits >> 23) & 0xff);

int m =(e == 0） ? (bits & 0x7fffff) << 1 : (bits & 0x7fffff) | 0x800000;

1

2

3

则float的浮点值为数值表达式s·m·2e–150的计算结果。

4.5 CONSTANT\_Long\_info和CONSTANT\_Double\_info结构

CONSTANT\_Long\_info和CONSTANT\_Double\_info结构表示8字节（long和double）的数值常量：

CONSTANT\_Long\_info {

u1 tag;

u4 high\_bytes;

u4 low\_bytes;

}

CONSTANT\_Double\_info {

u1 tag;

u4 high\_bytes;

u4 low\_bytes;

}

1

2

3

4

5

6

7

8

9

10

11

在Class文件的常量池中，所有的8字节的常量都占两个表成员（项）的空间。如果一个CONSTANT\_Long\_info或CONSTANT\_Double\_info结构的项在常量池中的索引为n，则常量池中下一个有效的项的索引为n+2，此时常量池中索引为n+1的项有效但必须被认为不可用。

CONSTANT\_Long\_info和CONSTANT\_Double\_info结构各项的说明如下：

tag

CONSTANT\_Long\_info结构的tag项的值是CONSTANT\_Long（5）。

CONSTANT\_Double\_info结构的tag项的值是CONSTANT\_Double（6）。

high\_bytes和low\_bytes

CONSTANT\_Long\_info结构中的无符号的high\_bytes和low\_bytes项用于共同表示long型常量，构造形式为((long) high\_bytes << 32) + low\_bytes，high\_bytes和low\_bytes都按照Big-Endian顺序存储。

CONSTANT\_Double\_info结构中的high\_bytes和low\_bytes共同按照IEEE 754双精度浮点格式表示double常量的值。high\_bytes和low\_bytes都按照Big-Endian顺序存储。

CONSTANT\_Double\_info结构表示的值将按照下列方式来表示，high\_bytes和low\_bytes首先被转换成一个long常量的bits：

如果bits值为0x7ff0000000000000L，表示double值为正无穷。

如果bits值为0xfff0000000000000L，表示double值为负无穷。

如果bits值在范围0x7ff0000000000001L到 0x7fffffffffffffffL或者0xfff0000000000001L到 0xffffffffffffffffL内，表示double值为NaN。

在其它情况下，设s、e、m，它们的值根据bits和如下公式计算：

int s =((bits >> 63) == 0) ? 1 : -1;

int e =(int)((bits >> 52) & 0x7ffL);

long m =(e == 0) ? (bits & 0xfffffffffffffL) << 1 : (bits & 0xfffffffffffffL) | 0x10000000000000L;

1

2

3

则double的浮点值为数学表达式s·m·2e – 1075的计算结果。

4.6 CONSTANT\_NameAndType\_info结构

CONSTANT\_NameAndType\_info结构用于表示字段或方法，但是和前面介绍的3个结构不同，CONSTANT\_NameAndType\_info结构没有标识出它所属的类或接口，格式如下：

CONSTANT\_NameAndType\_info {

u1 tag;

u2 name\_index;

u2 descriptor\_index;

}

1

2

3

4

5

CONSTANT\_NameAndType\_info结构各项的说明如下：

tag

CONSTANT\_NameAndType\_info结构的tag项的值为CONSTANT\_NameAndType（12）。

name\_index

name\_index项的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，这个结构要么表示特殊的方法名，要么表示一个有效的字段或方法的非限定名（Unqualified Name）。

descriptor\_index

descriptor\_index项的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，这个结构表示一个有效的字段描述符或方法描述符。

4.7 CONSTANT\_Utf8\_info结构

CONSTANT\_Utf8\_info结构用于表示字符串常量的值：

CONSTANT\_Utf8\_info {

u1 tag;

u2 length;

u1 bytes[length];

}

1

2

3

4

5

CONSTANT\_Utf8\_info结构各项的说明如下：

tag

CONSTANT\_Utf8\_info结构的tag项的值为CONSTANT\_Utf8（1）。

length

length项的值指明了bytes[]数组的长度（注意，不能等同于当前结构所表示的String对象的长度），CONSTANT\_Utf8\_info结构中的内容是以length属性确定长度而不是以null作为字符串的终结符。

bytes[]

bytes[]是表示字符串值的byte数组，bytes[]数组中每个成员的byte值都不会是0，也不在0xf0至0xff范围内。

字符串常量采用改进过的UTF-8编码表示。

4.8 CONSTANT\_MethodHandle\_info结构

CONSTANT\_MethodHandle\_info结构用于表示方法句柄，结构如下：

CONSTANT\_MethodHandle\_info {

u1 tag;

u1 reference\_kind;

u2 reference\_index;

}

1

2

3

4

5

CONSTANT\_MethodHandle\_info结构各项的说明如下：

tag

CONSTANT\_MethodHandle\_info结构的tag项的值为CONSTANT\_MethodHandle（15）。

reference\_kind

reference\_kind项的值必须在1至9之间（包括1和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（Bytecode Behavior）。

reference\_index

reference\_index项的值必须是对常量池的有效索引：

如果reference\_kind项的值为1（REF\_getField）、2（REF\_getStatic）、3（REF\_putField）或4（REF\_putStatic），那么常量池在reference\_index索引处的项必须是CONSTANT\_Fieldref\_info结构，表示由一个字段创建的方法句柄。

如果reference\_kind项的值是5（REF\_invokeVirtual）、6（REF\_invokeStatic）、7（REF\_invokeSpecial）或8（REF\_newInvokeSpecial），那么常量池在reference\_index索引处的项必须是CONSTANT\_Methodref\_info结构，表示由类的方法或构造函数创建的方法句柄。

如果reference\_kind项的值是9（REF\_invokeInterface），那么常量池在reference\_index索引处的项必须是CONSTANT\_InterfaceMethodref\_info结构，表示由接口方法创建的方法句柄。

如果reference\_kind项的值是5（REF\_invokeVirtual）、6（REF\_invokeStatic）、7（REF\_invokeSpecial）或9（REF\_invokeInterface），那么方法句柄对应的方法不能为实例初始化（）方法或类初始化方法（）。

如果reference\_kind项的值是8（REF\_newInvokeSpecial），那么方法句柄对应的方法必须为实例初始化（）方法。

4.9 CONSTANT\_MethodType\_info结构

CONSTANT\_MethodType\_info结构用于表示方法类型：

CONSTANT\_MethodType\_info {

u1 tag;

u2 descriptor\_index;

}

1

2

3

4

tag

CONSTANT\_MethodType\_info结构的tag项的值为CONSTANT\_MethodType（16）。

descriptor\_index

descriptor\_index项的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。

4.10 CONSTANT\_InvokeDynamic\_info结构

CONSTANT\_InvokeDynamic\_info用于表示invokedynamic指令所使用到的引导方法（Bootstrap Method）、引导方法使用到动态调用名称（Dynamic Invocation Name）、参数和请求返回类型、以及可以选择性的附加被称为静态参数（Static Arguments）的常量序列。

CONSTANT\_InvokeDynamic\_info {

u1 tag;

u2 bootstrap\_method\_attr\_index;

u2 name\_and\_type\_index;

}

1

2

3

4

5

CONSTANT\_InvokeDynamic\_info结构各项的说明如下：

tag

CONSTANT\_InvokeDynamic\_info结构的tag项的值为CONSTANT\_InvokeDynamic（18）。

bootstrap\_method\_attr\_index

bootstrap\_method\_attr\_index项的值必须是对当前Class文件中引导方法表的bootstrap\_methods[]数组的有效索引。

name\_and\_type\_index

name\_and\_type\_index项的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。

5.字段

每个字段（Field）都由field\_info结构所定义，在同一个Class文件中，不会有两个字段同时具有相同的字段名和描述符。

field\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

field\_info结构各项的说明如下：

access\_flags

access\_flags项的值是用于定义字段被访问权限和基础属性的掩码标志。access\_flags的取值范围和相应含义见下表所示。

标记名 值 说明

ACC\_PUBLIC 0x0001 public，表示字段可以从任何包访问。

ACC\_PRIVATE 0x0002 private，表示字段仅能该类自身调用。

ACC\_PROTECTED 0x0004 protected，表示字段可以被子类调用。

ACC\_STATIC 0x0008 static，表示静态字段。

ACC\_FINAL 0x0010 final，表示字段定义后值无法修改。

ACC\_VOLATILE 0x0040 volatile，表示字段是易变的。

ACC\_TRANSIENT 0x0080 transient，表示字段不会被序列化。

ACC\_SYNTHETIC 0x1000 表示字段由编译器自动产生。

ACC\_ENUM 0x4000 enum，表示字段为枚举类型。

字段如果带有ACC\_SYNTHETIC标志，则说明这个字段不是由源码产生的，而是由编译器自动产生的。

字段如果被标有ACC\_ENUM标志，这说明这个字段是一个枚举类型成员。

Class文件中的字段可以被设置多个表中的标记。不过有些标记是互斥的，一个字段最多只能设置ACC\_PRIVATE， ACC\_PROTECTED，和ACC\_PUBLIC三个标志中的一个，也不能同时设置标志ACC\_FINAL和ACC\_VOLATILE。

接口中的所有字段都具有ACC\_PUBLIC，ACC\_STATIC和ACC\_FINAL标记，也可能被设置ACC\_SYNTHETIC标记，但是不能含有表中的其它符号标记了。

在表中没有出现的access\_flags项的值为扩充而预留，在生成的Class文件中应被设置成0，Java虚拟机实现也应该忽略它们。

name\_index

name\_index项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，表示一个有效的字段的非全限定名。

descriptor\_index

descriptor\_index项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，表示一个有效的字段的描述符。

attributes\_count

attributes\_count的项的值表示当前字段的附加属性的数量。

attributes[]

attributes表的每一个成员的值必须是attribute结构，一个字段可以有任意个关联属性。

attributes表可出现的成员有：

ConstantValue, Synthetic, Signature, Deprecated, RuntimeVisibleAnnotations和RuntimeInvisibleAnnotations。

Java虚拟机实现必须正确的识别和读取field\_info结构的attributes表中的ConstantValue属性。如果Java虚拟机实现支持版本号为49.0或更高的Class文件，那么它必须正确的识别和读取这些Class文件中的Signature, RuntimeVisibleAnnotations和RuntimeInvisibleAnnotations结构。

所有Java虚拟机实现都必须默认忽略field\_info结构中attributes表所不可识别的成员。没有定义的属性不可影响Class文件的语义，它们只能提供附加描述信息。

6.方法

所有方法（Method），包括实例初始化方法和类初始化方法在内，都由method\_info结构所定义。在一个Class文件中，不会有两个方法同时具有相同的方法名和描述符。

method\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

method\_info结构各项的说明如下：

access\_flags

access\_flags项的值是用于定义当前方法的访问权限和基本属性的掩码标志，access\_flags的取值范围和相应含义见下表所示。

标记名 值 说明

ACC\_PUBLIC 0x0001 public，方法可以从包外访问

ACC\_PRIVATE 0x0002 private，方法只能本类中访问

ACC\_PROTECTED 0x0004 protected，方法在自身和子类可以访问

ACC\_STATIC 0x0008 static，静态方法

ACC\_FINAL 0x0010 final，方法不能被重写（覆盖）

ACC\_SYNCHRONIZED 0x0020 synchronized，方法由管程同步

ACC\_BRIDGE 0x0040 bridge，方法由编译器产生

ACC\_VARARGS 0x0080 表示方法带有变长参数

ACC\_NATIVE 0x0100 native，方法引用非java语言的本地方法

ACC\_ABSTRACT 0x0400 abstract，方法没有具体实现

ACC\_STRICT 0x0800 strictfp，方法使用FP-strict浮点格式

ACC\_SYNTHETIC 0x1000 方法在源文件中不出现，由编译器产生

ACC\_VARARGS 标志是用于说明方法在源码层的参数列表是否变长的。如果是变长的，则在编译时，方法的ACC\_VARARGS 标志设置1，其余的方法ACC\_VARARGS 标志设置为0。

ACC\_BRIDGE 标志用于说明这个方法是由编译生成的桥接方法。

如果方法设置了ACC\_SYNTHETIC[sɪn’θɛtɪk]标志，则说明这个方法是由编译器生成的并且不会在源代码中出现。

Class 文件中的方法可以设置多个表中的标志，但是有些标志是互斥的：一个方法只能设置ACC\_PRIVATE，ACC\_PROTECTED 和ACC\_PUBLI 三个标志中的一个标志；如果一个方法被设置ACC\_ABSTRACT 标志，则这个方法不能被设置ACC\_FINAL，

ACC\_NATIVE, ACC\_PRIVATE, ACC\_STATIC, ACC\_STRICT 和ACC\_SYNCHRONIZED标志。

所有的接口方法必须被设置ACC\_ABSTRACT 和ACC\_PUBLIC 标志；还可以选择设置ACC\_VARARGS，ACC\_BRIDGE 和ACC\_SYNTHETIC 标志，但是不能再设置表中其它的标志了。

实例初始化方法只能设置ACC\_PRIVATE，ACC\_PROTECTED 和ACC\_PUBLIC

中的一个标志；还可以设置ACC\_STRICT， ACC\_VARARGS 和ACC\_SYNTHETIC 标志，但是不能再设置表中的其它标志了。

类初始化方法由Java 虚拟机隐式自动调用，它的access\_flags 项的值除了ACC\_STRICT 标志，其它的标志都将被忽略。

在表中没有出现的access\_flags项值为未来扩充而预留，在生成的Class文件中应被设置成0，Java虚拟机实现应该忽略它们。

name\_index

name\_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，它要么表示初始化方法的名字（或），要么表示一个方法的有效的非全限定名。

descriptor\_index

descriptor\_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，表示一个有效的方法的描述符。注意：在未来的某个版本中可能会要求当access\_flags 项的ACC\_VARARGS 标志被设置时，方法描述符中的最后一个参数必须是数组类型。

attributes\_count

attributes\_count 的项的值表示这个方法的附加属性的数量。

attributes[]

attributes 表的每一个成员的值必须是attribute结构，一个方法可以有任意个与之相关的属性。属性表可出现的成员有：Code，Exceptions，Synthetic，Signature，Deprecated，untimeVisibleAnnotations，RuntimeInvisibleAnnotations，RuntimeVisibleParameterAnnotations，RuntimeInvisibleParameterAnnotations和AnnotationDefault结构。Java虚拟机实现必须正确识别和读取method\_info结构中的属性表的Code和Exceptions属性。如果Java虚拟机实现支持版本为49.0或更高的Class文件，那么它必须正确识别和读取这些Class文件的Signature，RuntimeVisibleAnnotations，RuntimeInvisibleAnnotations，RuntimeVisibleParameterAnnotations，RuntimeInvisibleParameterAnnotations和AnnotationDefault属性。所有Java虚拟机实现必须默认忽略method\_info结构中attributes表所不可识别的成员。没有定义的属性不可影响Class 文件的语义，它们只能提供附加描述信息。

7.属性

属性（Attributes）在Class文件格式中的ClassFile结构、field\_info结构，method\_info结构和Code\_attribute结构都有使用，所有属性的通用格式如下：

attribute\_info {

u2 attribute\_name\_index;

u4 attribute\_length;

u1 info[attribute\_length];

}

1

2

3

4

5

对于任意属性，attribute\_name\_index必须是对当前Class文件的常量池的有效16位无符号索引。常量池在该索引处的项必须是CONSTANT\_Utf8\_info结构，表示当前属性的名字。attribute\_length项的值给出了跟随其后的字节的长度，这个长度不包括attribute\_name\_index和attribute\_name\_index项的6个字节。

有些属性因Class文件格式规范所需，已被预先定义好。这些属性在表中列出，同时，被列出的信息还包括它们首次出现的Class文件版本和Java SE版本号。在当前规范定义的环境中，也就是已包含这些预定义属性的Class文件中，它们的属性名称被保留，不能再被属性表中其他的自定义属性所使用。

属性名 Java SE Class文件

ConstantValue 1.0.2 45.3

Code 1.0.2 45.3

StackMapTable 6 50.0

Exceptions 1.0.2 45.3

InnerClasses 1.1 45.3

EnclosingMethod 5.0 49.0

Synthetic 1.1 45.3

Signature 5.0 49.0

SourceFile 1.0.2 45.3

SourceDebugExtension 5.0 49.0

LineNumberTable 1.0.2 45.3

LocalVariableTable 1.0.2 45.3

LocalVariableTypeTable 5.0 49.0

Deprecated 1.1 45.3

RuntimeVisibleAnnotations 5.0 49.0

RuntimeInvisibleAnnotations 5.0 49.0

RuntimeVisibleParameterAnnotations 5.0 49.0

RuntimeInvisibleParameterAnnotations 5.0 49.0

AnnotationDefault 5.0 49.0

BootstrapMethods 7 51.0

Java虚拟机实现的Class文件加载器（Class File Reader）必须正确的识别和读取ConstantValue，Code和Exceptions属性；同样，Java虚拟机也必须能正确的解析它们的语义。

InnerClasses，EnclosingMethod和Synthetic属性必须被Class文件加载器正确的识别并读入，它们用于实现Java平台的类库。

如果Java虚拟机实现支持的Class文件的版本号为49.0或更高时，它的Class文件加载器必须能正确的识别并读取Class文件中的RuntimeVisibleAnnotations，RuntimeInvisibleAnnotations，RuntimeVisibleParameterAnnotations， RuntimeInvisibleParameterAnnotations和AnnotationDefault属性，它们用于实现Java平台类库。

如果Java虚拟机实现支持的Class文件的版本号为49.0或更高时，它的Class文件加载器必须正确的识别和读取Class文件中的Signature 属性。

如果Java虚拟机实现支持的Class文件的版本号为50.0或更高时，它的Class文件加载器必须正确的识别和读取StackMapTable属性。

如果Java虚拟机实现支持的Class文件的版本号为51.0或更高时，它的Class文件加载器必须正确的识别和读取BootstrapMethods属性。

对于剩余的预定义属性的使用不受限制；如果剩余的预定义属性包含虚拟机可识别的信息，Class文件加载器就可以选择使用这些信息，否则可以选择忽略它们。

7.1 自定义和命名新的属性

《Java虚拟机规范》允许编译器在Class文件的属性表中定义和发布新的属性。Java虚拟机实现允许识别并使用Class文件结构属性表中出现的新属性。但是，所有在《Java虚拟机规范》中没有定义的属性，不能影响类或接口类型的语义。Java虚拟机实现必须忽略它不能识别的自定义属性。

例如，编译器可以定义新的属性用于支持与特定发行者相关（Vendor-Specific）的调式，而不影响其它Java虚拟机实现。因为其他Java虚拟机实现必须忽略它不识别的属性，所以那些特定发行者相关的虚拟机实现所使用的Class文件也可以被别的Java虚拟机实现使用，即使这些Class文件包含的附加调式信息不能被它们所用。

《Java虚拟机规范》明确禁止Java虚拟机实现仅仅因为Class文件包含新属性而抛出异常或其他拒绝使用Class文件。当然，如果Class文件没有包含所需的属性，某些工具则可能无法正确操作这个Class文件。

当两个不同的属性使用了相同的属性名且长度也相同，无论虚拟机识别其中哪一个都会引起冲突。

7.2 ConstantValue属性

ConstantValue属性是定长属性，位于field\_info结构的属性表中。

ConstantValue属性表示一个常量字段的值。在一个field\_info结构的属性表中最多只能有一个ConstantValue属性。如果该字段为静态类型（即field\_info结构的access\_flags项设置了ACC\_STATIC标志），则说明这个field\_info结构表示的常量字段值将被分配为它的ConstantValue属性表示的值，这个过程也是类或接口申明的常量字段（Constant Field）初始化的一部分。这个过程发生在引用类或接口的类初始化方法执行之前。

如果field\_info结构表示的非静态字段包含了ConstantValue属性，那么这个属性必须被虚拟机所忽略。所有Java虚拟机实现必须能够识别ConstantValue属性。

ConstantValue属性的格式如下：

ConstantValue\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 constantvalue\_index;

}

1

2

3

4

5

attribute\_name\_index

attribute\_name\_index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项必须是CONSTANT\_Utf8\_info结构，表示字符串“ConstantValue”。

attribute\_length

ConstantValue\_attribute结构的attribute\_length项的值固定为2。

constantvalue\_index

constantvalue\_index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项给出该属性表示的常量值。常量池的项的类型表示的字段类型如表所示。

字段类型 项类型

long CONSTANT\_Long

float CONSTANT\_Float

double CONSTANT\_Double

int,short,char,byte,boolean CONSTANT\_Integer

String CONSTANT\_String

7.3 Code属性

Code属性是一个变长属性，位于method\_info结构的属性表。一个Code属性只为唯一一个方法、实例类初始化方法或类初始化方法保存Java虚拟机指令及相关辅助信息。所有Java虚拟机实现都必须能够识别Code属性。如果方法被声明为native或者abstract类型，那么对应的method\_info结构不能有明确的Code属性，其它情况下，method\_info有必须有明确的Code属性。

Code\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 max\_stack;

u2 max\_locals;

u4 code\_length;

u1 code[code\_length];

u2 exception\_table\_length;

{ u2 start\_pc;

u2 end\_pc;

u2 handler\_pc;

u2 catch\_type;

} exception\_table[exception\_table\_length];

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

attribute\_name\_index

attribute\_name\_index项的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，表示字符串“Code”。

attribute\_length

attribute\_length项的值表示当前属性的长度，不包括开始的6个字节。

max\_stack

max\_stack项的值给出了当前方法的操作数栈在运行执行的任何时间点的最大深度。

max\_locals

max\_locals项的值给出了分配在当前方法引用的局部变量表中的局部变量个数，包括调用此方法时用于传递参数的局部变量。long和double型的局部变量的最大索引是max\_locals-2，其它类型的局部变量的最大索引是max\_locals-1.

code\_length

code\_length项给出了当前方法的code[]数组的字节数，code\_length的值必须大于0，即code[]数组不能为空。

code[]

code[]数组给出了实现当前方法的Java虚拟机字节码。code[]数组以按字节寻址的方式读入机器内存，如果code[]数组的第一个字节是按以4字节边界对齐的话，那么tableswitch和lookupswitch指令中所有涉及到的32位偏移量也都是按4字节长度对齐的（关于code[]数组边界对齐对字节码的影响，请参考相关的指令描述）。

exception\_table\_length

exception\_table\_length项的值给出了exception\_table[]数组的成员个数量。

exception\_table[]

exception\_table[]数组的每个成员表示code[]数组中的一个异常处理器（Exception Handler）。exception\_table[]数组中，异常处理器顺序是有意义的（不能随意更改）。

exception\_table[]数组包含如下4项：

start\_pc和end\_pc

start\_pc和end\_pc两项的值表明了异常处理器在code[]数组中的有效范围。start\_pc必须是对当前code[]数组中某一指令的操作码的有效索引，end\_pc要么是对当前code[]数组中某一指令的操作码的有效索引，要么等于code\_length的值，即当前code[]数组的长度。start\_pc的值必须比end\_pc小。 当程序计数器在范围[start\_pc, end\_pc)内时，异常处理器就将生效。即设x为异常句柄的有效范围内的值，x满足：start\_pc ≤ x < end\_pc。 实际上，end\_pc值本身不属于异常处理器的有效范围这点属于Java虚拟机历史上的一个设计缺陷：如果Java虚拟机中的一个方法的code属性的长度刚好是65535个字节，并且以一个1个字节长度的指令结束，那么这条指令将不能被异常处理器所处理。不过编译器可以通过限制任何方法、实例初始化方法或类初始化方法的code[]数组最大长度为65534，这样可以间接弥补这个BUG。

handler\_pc

handler\_pc项表示一个异常处理器的起点，它的值必须同时是一个对当前code[]数组中某一指令的操作码的有效索引。

catch\_type

如果catch\_type项的值不为0，那么它必须是对常量池的一个有效索引，常量池在该索引处的项必须是CONSTANT\_Class\_info结构，表示当前异常处理器指定需要捕捉的异常类型。只有当抛出的异常是指定的类或其子类的实例时，异常处理器才会被调用。 如果catch\_type项的值如果为0，那么这个异常处理器将会在所有异常抛出时都被调用。这可以用于实现finally语句（“编译finally”）。

attributes\_count

attributes\_count项的值给出了Code属性中attributes表的成员个数。

attributes[]

属性表的每个成员的值必须是attribute结构。一个Code属性可以有任意数量的可选属性与之关联。

规范中定义的、可以出现在Code属性的属性表中的成员只能是LineNumberTable，LocalVariableTable，LocalVariableTypeTable和StackMapTable属性。

如果一个Java虚拟机实现支持的Class文件版本号为50.0或更高，那么它必须正确的识别和读取Code属性的属性表出现的StackMapTable属性。

Java虚拟机实现必须自动忽略Code属性的属性表数组中出现的所有它不能识别属性。规范中没有定义的属性不可影响Class文件的语义，只能提供附加描述信息。

7.4 StackMapTable属性

StackMapTable属性是一个变长属性，位于Code属性的属性表中。这个属性会在虚拟机类加载的类型阶段被使用。

StackMapTable属性包含0至多个栈映射帧（Stack Map Frames），每个栈映射帧都显式或隐式地指定了一个字节码偏移量，用于表示局部变量表和操作数栈的验证类型（Verification Types）。

类型检测器（Type Checker）会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。一个存储单元（Location）的含义是唯一的局部变量或操作数栈项。

我们还将用到术语“栈映射帧”（Stack Map Frame）和“类型状态”（Type State）来描述如何从方法的局部变量和操作数栈的存储单元映射到验证类型（Verification Types）。当描述Class文件侧的映射时，我们通常使用的术语是“栈映射帧”，而当描述类型检查器侧的映射关系时，我们通常使用的术语是“类型状态”。

在版本号大于或等于50.0的Class文件中，如果方法的Code属性中没有附带StackMapTable属性，那就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属性最多只能有一个StackMapTable属性，否则将抛出ClassFormatError异常。

StackMapTable属性的格式如下：

StackMapTable\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 number\_of\_entries;

stack\_map\_frame entries[number\_of\_entries];

}

1

2

3

4

5

6

StackMapTable结构项的说明如下：

attribute\_name\_index

attribute\_name\_index项的值必须是对常量池的有效索引，常量池在该索引的项处必须是CONSTANT\_Utf8\_info结构，表示“StackMapTable”字符串。

attribute\_length

attribute\_length项的值表示当前属性的长度，不包括开始的6个字节。

number\_of\_entries

number\_of\_entries项的值给出了entries表中的成员数量。Entries表的每个成员是都是一个stack\_map\_frame结构的项。

entries[]

entries表给出了当前方法所需的stack\_map\_frame结构。

每个stack\_map\_frame结构都使用一个特定的字节偏移量来表示类型状态。每个帧类型（Frame Type）都显式或隐式地标明一个offset\_delta（增量偏移量）值，用于计算每个帧在运行时的实际字节码偏移量。使用时帧的字节偏移量计算方法为：前一帧的字节码偏移量（Bytecode Offset）加上offset\_delta的值再加1，如果前一个帧是方法的初始帧（Initial Frame），那这时候字节码偏移量就是offset\_delta。

只要保证栈映射帧有正确的存储顺序，在类型检查时我们就可以使用增量偏移量而不是实际的字节码偏移量。此外，由于对每一个帧都使用了offset\_delta+1的计算方式，我们可以确保偏移量不会重复。

在Code属性的code[]数组项中，如果偏移量i的位置是某条指令的起点，同时这个Code属性包含有StackMapTable属性，它的entries项中也有一个适用于地址偏移量i的stack\_map\_frame结构，那我们就说这条指令拥有一个与之相对应的栈映射帧。

stack\_map\_frame结构的第一个字节作为类型标记（Tag），第一个字节后会跟随0或多个字节用于说明更多信息，这些信息因类型标记的不同而变化。

一个栈映射帧可以包含若干种帧类型（Frame Types）：

union stack\_map\_frame {

same\_frame;

same\_locals\_1\_stack\_item\_frame;

same\_locals\_1\_stack\_item\_frame\_extended;

chop\_frame;

same\_frame\_extended;

append\_frame; full\_frame;

}

1

2

3

4

5

6

7

8

还有一些属性就不一一列举了……

8.实战

8.1 源码

package com.lhw.test;

public class TestClassFile {

private static final int INT\_VAL = 10000;

public static void main(String[] args){

TestClassFile.test(INT\_VAL);

}

private static void test(int val){

System.out.println("Test Method val=" + val);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

这段源码非常简单，声明了一个int类型的常量INT\_VAL和一个方法test(int val)。编译成我们熟悉的class文件一探究竟。

8.2 class文件

00000000h: CA FE BA BE 00 00 00 34 00 39 0A 00 0C 00 21 07 ; 漱壕...4.9....!.

00000010h: 00 22 0A 00 02 00 23 09 00 24 00 25 07 00 26 0A ; ."....#..$.%..&.

00000020h: 00 05 00 21 08 00 27 0A 00 05 00 28 0A 00 05 00 ; ...!..'....(....

00000030h: 29 0A 00 05 00 2A 0A 00 2B 00 2C 07 00 2D 01 00 ; )....\*..+.,..-..

00000040h: 07 49 4E 54 5F 56 41 4C 01 00 01 49 01 00 0D 43 ; .INT\_VAL...I...C

00000050h: 6F 6E 73 74 61 6E 74 56 61 6C 75 65 03 00 00 27 ; onstantValue...'

00000060h: 10 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29 56 ; ....<init>...()V

00000070h: 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E 75 ; ...Code...LineNu

00000080h: 6D 62 65 72 54 61 62 6C 65 01 00 12 4C 6F 63 61 ; mberTable...Loca

00000090h: 6C 56 61 72 69 61 62 6C 65 54 61 62 6C 65 01 00 ; lVariableTable..

000000a0h: 04 74 68 69 73 01 00 1C 4C 63 6F 6D 2F 6C 68 77 ; .this...Lcom/lhw

000000b0h: 2F 74 65 73 74 2F 54 65 73 74 43 6C 61 73 73 46 ; /test/TestClassF

000000c0h: 69 6C 65 3B 01 00 04 6D 61 69 6E 01 00 16 28 5B ; ile;...main...([

000000d0h: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin

000000e0h: 67 3B 29 56 01 00 04 61 72 67 73 01 00 13 5B 4C ; g;)V...args...[L

000000f0h: 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 ; java/lang/String

00000100h: 3B 01 00 04 74 65 73 74 01 00 04 28 49 29 56 01 ; ;...test...(I)V.

00000110h: 00 03 76 61 6C 01 00 0A 53 6F 75 72 63 65 46 69 ; ..val...SourceFi

00000120h: 6C 65 01 00 12 54 65 73 74 43 6C 61 73 73 46 69 ; le...TestClassFi

00000130h: 6C 65 2E 6A 61 76 61 0C 00 11 00 12 01 00 1A 63 ; le.java........c

00000140h: 6F 6D 2F 6C 68 77 2F 74 65 73 74 2F 54 65 73 74 ; om/lhw/test/Test

00000150h: 43 6C 61 73 73 46 69 6C 65 0C 00 1C 00 1D 07 00 ; ClassFile.......

00000160h: 2E 0C 00 2F 00 30 01 00 17 6A 61 76 61 2F 6C 61 ; .../.0...java/la

00000170h: 6E 67 2F 53 74 72 69 6E 67 42 75 69 6C 64 65 72 ; ng/StringBuilder

00000180h: 01 00 10 54 65 73 74 20 4D 65 74 68 6F 64 20 76 ; ...Test Method v

00000190h: 61 6C 3D 0C 00 31 00 32 0C 00 31 00 33 0C 00 34 ; al=..1.2..1.3..4

000001a0h: 00 35 07 00 36 0C 00 37 00 38 01 00 10 6A 61 76 ; .5..6..7.8...jav

000001b0h: 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 10 ; a/lang/Object...

000001c0h: 6A 61 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D ; java/lang/System

000001d0h: 01 00 03 6F 75 74 01 00 15 4C 6A 61 76 61 2F 69 ; ...out...Ljava/i

000001e0h: 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D 3B 01 00 ; o/PrintStream;..

000001f0h: 06 61 70 70 65 6E 64 01 00 2D 28 4C 6A 61 76 61 ; .append..-(Ljava

00000200h: 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 4C 6A ; /lang/String;)Lj

00000210h: 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 42 ; ava/lang/StringB

00000220h: 75 69 6C 64 65 72 3B 01 00 1C 28 49 29 4C 6A 61 ; uilder;...(I)Lja

00000230h: 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 42 75 ; va/lang/StringBu

00000240h: 69 6C 64 65 72 3B 01 00 08 74 6F 53 74 72 69 6E ; ilder;...toStrin

00000250h: 67 01 00 14 28 29 4C 6A 61 76 61 2F 6C 61 6E 67 ; g...()Ljava/lang

00000260h: 2F 53 74 72 69 6E 67 3B 01 00 13 6A 61 76 61 2F ; /String;...java/

00000270h: 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D 01 00 ; io/PrintStream..

00000280h: 07 70 72 69 6E 74 6C 6E 01 00 15 28 4C 6A 61 76 ; .println...(Ljav

00000290h: 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56 ; a/lang/String;)V

000002a0h: 00 21 00 02 00 0C 00 00 00 01 00 1A 00 0D 00 0E ; .!..............

000002b0h: 00 01 00 0F 00 00 00 02 00 10 00 03 00 01 00 11 ; ................

000002c0h: 00 12 00 01 00 13 00 00 00 2F 00 01 00 01 00 00 ; ........./......

000002d0h: 00 05 2A B7 00 01 B1 00 00 00 02 00 14 00 00 00 ; ..\*?.?........

000002e0h: 06 00 01 00 00 00 03 00 15 00 00 00 0C 00 01 00 ; ................

000002f0h: 00 00 05 00 16 00 17 00 00 00 09 00 18 00 19 00 ; ................

00000300h: 01 00 13 00 00 00 35 00 01 00 01 00 00 00 07 11 ; ......5.........

00000310h: 27 10 B8 00 03 B1 00 00 00 02 00 14 00 00 00 0A ; '.?.?.........

00000320h: 00 02 00 00 00 08 00 06 00 09 00 15 00 00 00 0C ; ................

00000330h: 00 01 00 00 00 07 00 1A 00 1B 00 00 00 0A 00 1C ; ................

00000340h: 00 1D 00 01 00 13 00 00 00 48 00 03 00 01 00 00 ; .........H......

00000350h: 00 1A B2 00 04 BB 00 05 59 B7 00 06 12 07 B6 00 ; ..?.?.Y?...?

00000360h: 08 1A B6 00 09 B6 00 0A B6 00 0B B1 00 00 00 02 ; ..?.?.?.?...

00000370h: 00 14 00 00 00 0A 00 02 00 00 00 0C 00 19 00 0D ; ................

00000380h: 00 15 00 00 00 0C 00 01 00 00 00 1A 00 1E 00 0E ; ................

00000390h: 00 00 00 01 00 1F 00 00 00 02 00 20 ; ...........

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

8.3 class文件解析

8.3.1 魔数

从上面文件开头很容易看到CA FE BA BE（偏移量从0x00000000开始），也就是固定值固定0xCAFEBABE。

8.3.2 版本

副版本号minor version: 0 （偏移量从0x00000004开始）

主版本号major version: 52 （偏移量从0x00000006开始）

也就是说这个class文件的版本号为52.0（Java SE 8 = 52 (0x34 hex)）

8.3.3 常量池

偏移量从0x00000008开始，常量池中常量个数为57（0x0039）。然后继续往下看在偏移地址为0x0000000a处读到的字节为0x0A，也就是tag为10，说明这是一个CONSTANT\_Methodref类型常量。它后面会跟随两个字节的class\_index，然后是两个字节的name\_and\_type\_index。先来看class\_index，也就是0x000C，十进制为12，这个索引在常量池中的项必须是CONSTANT\_Class\_info结构，最后再来看name\_and\_type\_index，也就是0x0021，十进制为33，这个索引的项必须是CONSTANT\_NameAndType\_info结构。

上面索引12和33还没有从常量池中解析出具体是什么，接着向下解析。偏移地址从0x0000000f开始，读取一个字节0x07，也就是tag为7（CONSTANT\_Class），接下来读取两个字节为name\_index，即0x0022（34），name\_index在常量池中的索引是一个CONSTANT\_Utf8\_info结构。

还是无法完整解析出其常量池结构，其实还差很远，继续解析出下一个tag也是10，是一个CONSTANT\_Methodref类型常量，class\_index为2，name\_and\_type\_index为35。继续移动到偏移地址0x00000017的位置，tag为9，它是一个CONSTANT\_Fieldref结构，这个结构共包含5个字节，后面的字节为class\_index和name\_and\_type\_index，各占两个字节，它们的索引分别为36和37。下面读到tag为7，继续读取name\_index，这个索引为38。继续移动到偏移地址0x0000001f的位置，tag是10，class\_index是0x0005，也就是5，name\_and\_type\_index是0x0021，转换为十进制就是33。继续向前解析，移动到偏移地址0x00000024的位置，tag为8，它是一个CONSTANT\_String结构，后面跟了两个字节为string\_index，同样这个index指向常量池的CONSTANT\_Utf8\_info，换算0x0027即39。继续前行，又是一个tag为10的CONSTANT\_Methodref类型，其class\_index为5，name\_and\_type\_index为40。下一个tag也是10，class\_index为5，name\_and\_type\_index为41。移动到偏移地址0x00000031的位置，tag是10，class\_index为5，name\_and\_type\_index为42。偏移地址继续向下，tag是10，class\_index为43，name\_and\_type\_index为44。移动到偏移地址0x0000003b的位置，读出tag为7，继续读取name\_index，索引为45。下一个tag是1，就是一个CONSTANT\_Utf8类型，长度是0x0007也就是7，后面的七个字节是：49 4E 54 5F 56 41 4C，也就是字符串INT\_VAL。

……解析了半天才解析了上面5行，这样手工的方式去解析一个class文件效率太低了，按照java class文件格式去写一个工具？java已经迭代了很多个版本了，这类工具当然早就有了，就是javap工具，它位于JDK安装目录bin下，如果将bin已经配置到环境变量（这里指windows系统），那么cmd窗口使用javap -v xx.class命令就可以打印出它的详细信息了。记得以前看到一篇博客，上面列举了java程序员进阶需要掌握那些知识，其中之一就是写一个类似javap命令的工具，这实际上就是考察对java class文件格式的掌握情况，学习完class文件格式，完成一个类似工具自然就不难了。

javap -v C:\Users\xx\Desktop\TestClassFile.class

Classfile /C:/Users/xx/Desktop/TestClassFile.class

Last modified 2017-11-6; size 924 bytes

MD5 checksum 70ac5d771a8f2123b380f490d9fbd1f1

Compiled from "TestClassFile.java"

public class com.lhw.test.TestClassFile

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #12.#33 // java/lang/Object."<init>":()V

#2 = Class #34 // com/lhw/test/TestClassFile

#3 = Methodref #2.#35 // com/lhw/test/TestClassFile.test:(I)V

#4 = Fieldref #36.#37 // java/lang/System.out:Ljava/io/PrintStream;

#5 = Class #38 // java/lang/StringBuilder

#6 = Methodref #5.#33 // java/lang/StringBuilder."<init>":()V

#7 = String #39 // Test Method val=

#8 = Methodref #5.#40 // java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

#9 = Methodref #5.#41 // java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;

#10 = Methodref #5.#42 // java/lang/StringBuilder.toString:()Ljava/lang/String;

#11 = Methodref #43.#44 // java/io/PrintStream.println:(Ljava/lang/String;)V

#12 = Class #45 // java/lang/Object

#13 = Utf8 INT\_VAL

#14 = Utf8 I

#15 = Utf8 ConstantValue

#16 = Integer 10000

#17 = Utf8 <init>

#18 = Utf8 ()V

#19 = Utf8 Code

#20 = Utf8 LineNumberTable

#21 = Utf8 LocalVariableTable

#22 = Utf8 this

#23 = Utf8 Lcom/lhw/test/TestClassFile;

#24 = Utf8 main

#25 = Utf8 ([Ljava/lang/String;)V

#26 = Utf8 args

#27 = Utf8 [Ljava/lang/String;

#28 = Utf8 test

#29 = Utf8 (I)V

#30 = Utf8 val

#31 = Utf8 SourceFile

#32 = Utf8 TestClassFile.java

#33 = NameAndType #17:#18 // "<init>":()V

#34 = Utf8 com/lhw/test/TestClassFile

#35 = NameAndType #28:#29 // test:(I)V

#36 = Class #46 // java/lang/System

#37 = NameAndType #47:#48 // out:Ljava/io/PrintStream;

#38 = Utf8 java/lang/StringBuilder

#39 = Utf8 Test Method val=

#40 = NameAndType #49:#50 // append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

#41 = NameAndType #49:#51 // append:(I)Ljava/lang/StringBuilder;

#42 = NameAndType #52:#53 // toString:()Ljava/lang/String;

#43 = Class #54 // java/io/PrintStream

#44 = NameAndType #55:#56 // println:(Ljava/lang/String;)V

#45 = Utf8 java/lang/Object

#46 = Utf8 java/lang/System

#47 = Utf8 out

#48 = Utf8 Ljava/io/PrintStream;

#49 = Utf8 append

#50 = Utf8 (Ljava/lang/String;)Ljava/lang/StringBuilder;

#51 = Utf8 (I)Ljava/lang/StringBuilder;

#52 = Utf8 toString

#53 = Utf8 ()Ljava/lang/String;

#54 = Utf8 java/io/PrintStream

#55 = Utf8 println

#56 = Utf8 (Ljava/lang/String;)V

{

public com.lhw.test.TestClassFile();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

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

4: return

LineNumberTable:

line 3: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/lhw/test/TestClassFile;

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=1, locals=1, args\_size=1

0: sipush 10000

3: invokestatic #3 // Method test:(I)V

6: return

LineNumberTable:

line 8: 0

line 9: 6

LocalVariableTable:

Start Length Slot Name Signature

0 7 0 args [Ljava/lang/String;

}

SourceFile: "TestClassFile.java"

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

常量池中共有57 - 1 = 56项，分别有Methodref、Fieldref、String、Integer、Utf8、NameAndType和Class这几种结构的常量。Methodref类型看到了方法所属类、方法名和方法描述符，接触过jni编程，看到方法描述符是不是有种亲切的感觉。

8.3.4 访问标志

看上面javap输出的内容，紧接着class文件版本号下面的flags就是测试类的访问标志（access\_flags）了，也就是flags: ACC\_PUBLIC, ACC\_SUPER。ACC\_PUBLIC代表可以被包的类外访问，ACC\_SUPER表示当用到invokespecial指令时，需要特殊处理（“特殊处理”是相对于JDK 1.0.2之前的Class文件而言，invokespecial的语义和处理方式在JDK 1.0.2时发生了变化，为避免二义性，在JDK 1.0.2之后编译出的Class文件，都带有ACC\_SUPER标志用以区分）的父类方法。在偏移地址0x000002a0的位置，读取两个字节就代表访问标志了。即0x0021，二进制表示为0b

8.3.5 类索引

类索引，this\_class的值必须是对constant\_pool表中项目的一个有效索引值。也就是两个字节的0x0002，索引2在常量池中为：

#2 = Class #34 // com/lhw/test/TestClassFile

1

8.3.6 父类索引

这个class文件中代表父类索引的值是0x000C，也就是索引12

#12 = Class #45 // java/lang/Object

1

没有显式继承任何父类，java中所有的类都是从Object派生的，所有它的父类自然是Object了。

8.3.7 接口计数器

其值为0x0000，这个java类没有实现接口。

8.3.8 字段计数器

其值为0x0001，共有一个字段，再往后解析是field\_info结构。

8.3.9 字段

回顾一下field\_info的结构

field\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

access\_flags是0x001A，即0b

name\_index是0x000D，也就是索引13，查上面的常量表：

#13 = Utf8 INT\_VAL

1

再来看描述符索引descriptor\_index，值为0x000E（14），描述符是I，代表整型。

#14 = Utf8 I

1

接下来是attributes\_count，属性个数等于0x0001（1）。下面接着就是属性表，同样回顾一下属性表的结构：

attribute\_info {

u2 attribute\_name\_index;

u4 attribute\_length;

u1 info[attribute\_length];

}

1

2

3

4

5

首先是attribute\_name\_index，在偏移地址0x000002b2的位置往下连续读取两个字节，也就是0x000F，查看一下常量池中索引15的位置：

#15 = Utf8 ConstantValue

1

这代表当前属性的名字，ConstantValue属性表示一个常量字段的值。看看它的长度是多少，也就是0x00000002。继续向下分析前需要看看具体的ConstantValue\_attribute。

ConstantValue\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 constantvalue\_index;

}

1

2

3

4

5

前六个字节长度是固定的，我们解析出来后面的长度为2个字节，看来的确如此。constantvalue\_index的索引是0x0010，去常量池16处看看：

#16 = Integer 10000

1

是一个整型，字面值为10000，和程序中是一致的。

8.3.10 方法计数器

查到其值是0x0003，代表了包含3个方法。methods\_count的值表示当前Class文件methods[]数组的成员个数。Methods[]数组中每一项都是一个method\_info结构的数据项。

8.3.11 方法

先回顾一下方法的结构，这和字段是非常类似的。

method\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

先来解析第一个方法：

access\_flags是0x0001（ACC\_PUBLIC），到底这个方法是什么看看它的名字，name\_index为0x0011，位于常量池位置17。

#17 = Utf8 <init>

1

:在实例创建出来的时候调用，包括调用new操作符；调用Class或java.lang.reflect.Constructor对象的newInstance()方法；调用任何现有对象的clone()方法；通过java.io.ObjectInputStream类的getObject()方法反序列化。

再看它的描述符，也就是index为0x0012，常量池18的位置可查到：

#18 = Utf8 ()V

1

属性个数等于1，解析一下属性表，属性名称为Code，转到Code\_info的结构看看。

Code\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 max\_stack;

u2 max\_locals;

u4 code\_length;

u1 code[code\_length];

u2 exception\_table\_length;

{

u2 start\_pc;

u2 end\_pc;

u2 handler\_pc;

u2 catch\_type;

} exception\_table[exception\_table\_length];

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

attribute\_length等于0x0000002F，即47。下面是片段截取了47个字节长度，完成剩余的解析。

000002c0h: 00 01 00 01 00 00 ; ........./......

000002d0h: 00 05 2A B7 00 01 B1 00 00 00 02 00 14 00 00 00 ; ..\*?.?........

000002e0h: 06 00 01 00 00 00 03 00 15 00 00 00 0C 00 01 00 ; ................

000002f0h: 00 00 05 00 16 00 17 00 00 ; .........

1

2

3

4

max\_stack（当前方法的操作数栈在运行执行的任何时间点的最大深度）为1，max\_locals（当前方法引用的局部变量表中的局部变量个数）为1，code\_length是5，也就是2A B7 00 01 B1这些字节。具体代表什么，需要查阅虚拟机指令集。

字节码 助记符 指令含义

0x2a aload\_0 将第一个引用类型本地变量推送至栈顶

0xb7 invokespecial 调用超类构造方法，实例初始化方法，私有方法

0xb1 return 从当前方法返回void

需要注意一下invokespecial这条指令，具体格式如下：

invokespecial indexbyte1 indexbyte2

1

无符号数indexbyte1和indexbyte2用于构建一个当前类的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个方法的符号引用，它包含了方法的名称和描述符，以及包含该方法的接口的符号引用。最后，如果调用的方法是protected的，并且这个方法是当前类的父类成员，并且这个方法没有在同一个运行时包中定义过，那objectref所指向的对象的类型必须为当前类或者当前类的子类。

再去看看通过javap命令输出的信息，以下就是我们正在解析的。

0: aload\_0

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

4: return

1

2

3

接下来看exception\_table\_length的值0x0000，不存在异常表。直接略过。最后来看属性表的项的个数attributes\_count=2，逐个解析。

第一个为LineNumberTable。LineNumberTable属性是可选变长属性，位于Code结构的属性表。它被调试器用于确定源文件中行号表示的内容在Java虚拟机的code[]数组中对应的部分。在Code属性的属性表中，LineNumberTable属性可以按照任意顺序出现，此外，多个LineNumberTable属性可以共同表示一个行号在源文件中表示的内容，即LineNumberTable属性不需要与源文件的行一一对应。

#20 = Utf8 LineNumberTable

1

来看看LineNumberTable的结构。

LineNumberTable\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 line\_number\_table\_length;

{

u2 start\_pc;

u2 line\_number;

} line\_number\_table[line\_number\_table\_length];

}

1

2

3

4

5

6

7

8

9

解析一下，attribute\_length等于0x00000006，line\_number\_table\_length等于0x0001，代表了只有一个line\_number\_table结构，其中start\_pc=0，line\_number=3。

line\_number\_table[]数组的每个成员都表明源文件中行号的变化在code[]数组中都会有对应的标记点。line\_number\_table的每个成员都具有如下两项：

start\_pc

start\_pc项的值必须是code[]数组的一个索引，code[]数组在该索引处的字符表示源文件中新的行的起点。start\_pc项的值必须小于当前LineNumberTable属性所在的Code属性的code\_length项的值。

line\_number

line\_number项的值必须与源文件的行数相匹配。

LineNumberTable:

line 3: 0

1

2

属性表的长度是2，看看第二个属性是什么？index=0x0015。查找常量池就会发现是LocalVariableTable，也就是本地变量表。

#21 = Utf8 LocalVariableTable

1

LocalVariableTable是可选变长属性，位于Code属性的属性表中。它被调试器用于确定方法在执行过程中局部变量的信息。在Code属性的属性表中，LocalVariableTable属性可以按照任意顺序出现。Code属性中的每个局部变量最多只能有一个LocalVariableTable属性。

LocalVariableTable属性格式如下：

LocalVariableTable\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 local\_variable\_table\_length;

{ u2 start\_pc;

u2 length;

u2 name\_index;

u2 descriptor\_index;

u2 index;

} local\_variable\_table[local\_variable\_table\_length];

}

1

2

3

4

5

6

7

8

9

10

11

attribute\_length=0x00 00 00 0C，即长度是12。local\_variable\_table\_length=1，start\_pc=0，length=5，name\_index=22，descriptor\_index=23，index=0。

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/lhw/test/TestClassFile;

1

2

3

local\_variable\_table[]数组的每一个成员表示一个局部变量的值在code[]数组中的偏移量范围。它同时也是用于从当前帧的局部变量表找出所需的局部变量的索引。local\_variable\_table[]数组每个成员都有如下5个项：

start\_pc, length

所有给定的局部变量的索引都在范围[start\_pc,start\_pc+length)中，即从start\_pc（包括自身值）至start\_pc+length（不包括自身值）。

name\_index

name\_index项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是CONSTANT\_Utf8\_info结构，表示一个局部变量的有效的非全限定名。

descriptor\_index

descriptor\_index项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是CONSTANT\_Utf8\_info结构，表示源程序中局部变量类型的字段描述符。

index

index为此局部变量在当前栈帧的局部变量表中的索引。如果在index索引处的局部变量是long或double型，则占用index和index+1两个索引。

以上仅仅是解析了3个方法中的一个，还剩下两个方法。

000002f0h: 00 09 00 18 00 19 00 ; ................

00000300h: 01 00 13 00 00 00 35 00 01 00 01 00 00 00 07 11 ; ......5.........

00000310h: 27 10 B8 00 03 B1 00 00 00 02 00 14 00 00 00 0A ; '.?.?.........

00000320h: 00 02 00 00 00 08 00 06 00 09 00 15 00 00 00 0C ; ................

00000330h: 00 01 00 00 00 07 00 1A 00 1B 00 00 ; ............

1

2

3

4

5

以上的16进制片段就是第二个方法，其实就是main方法。解析以后如下：

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=1, locals=1, args\_size=1

0: sipush 10000

3: invokestatic #3 // Method test:(I)V

6: return

LineNumberTable:

line 8: 0

line 9: 6

LocalVariableTable:

Start Length Slot Name Signature

0 7 0 args [Ljava/lang/String;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

再来看最后一个方法，其实就是test方法。十六进制片段如下，就不再继续解析了。

00000330h: 00 0A 00 1C ; ................

00000340h: 00 1D 00 01 00 13 00 00 00 48 00 03 00 01 00 00 ; .........H......

00000350h: 00 1A B2 00 04 BB 00 05 59 B7 00 06 12 07 B6 00 ; ..?.?.Y?...?

00000360h: 08 1A B6 00 09 B6 00 0A B6 00 0B B1 00 00 00 02 ; ..?.?.?.?...

00000370h: 00 14 00 00 00 0A 00 02 00 00 00 0C 00 19 00 0D ; ................

00000380h: 00 15 00 00 00 0C 00 01 00 00 00 1A 00 1E 00 0E ; ................

00000390h: 00 00 ; ..

1

2

3

4

5

6

7

8.3.12 属性计数器

在偏移地址0x00000392的位置开始，读取两个字节也就是0x0001，表示当前Class文件attributes表的成员个数。Class文件结构中的attributes表的项包括下列定义的属性：InnerClasses、EnclosingMethod、Synthetic、Signature、SourceFile，SourceDebugExtension、Deprecated、RuntimeVisibleAnnotations、RuntimeInvisibleAnnotations以及BootstrapMethods属性。

8.3.13 属性表

解析出attribute\_name\_index=0x001F，attribute\_length=0x00000002，也就是长度为2。

#31 = Utf8 SourceFile

1

最后再来看一下SourceFile属性。SourceFile属性是可选定长字段，位于ClassFile结构的属性表。一个ClassFile结构中的属性表最多只能包含一个SourceFile属性。

SourceFile\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 sourcefile\_index;

}

1

2

3

4

5

文件的最后两个字节也就是sourcefile\_index了，sourcefile\_index=0x0020（32），查找常量池索引32如下：

#32 = Utf8 TestClassFile.java

1

sourcefile\_index项引用字符串表示被编译的Class文件的源文件的名字。不包括源文件所在目录的目录名，也不包括源文件的绝对路径名。

至此，整个class文件解析完毕了，并没有涉及到方方面面，感叹文件结构的复杂。想要对其深入了解建议自行解析一份十六进制class文件加深印象。

参考资料

1.《Java Virtual Machine Specification Java SE 7 中文版》

---------------------

作者：tyyj90

来源：CSDN

原文：https://blog.csdn.net/tyyj90/article/details/78472986

版权声明：本文为博主原创文章，转载请附上博文链接！