第6章 类文件结构

6.2 无关性基石

Sun公司以及其他虚拟机提供商发布了许多可以运行在各种不同平台上的虚拟机，这些虚拟机都可以载入和执行同一种平台无关的字节码，从而实现“一次编写，到处运行”。

实现语言无关性的基础仍然是虚拟机和字节码存储格式。Java虚拟机不和包括Java在内的任何语言绑定，它只与“Class文件”这种特定的二进制文件格式所关联，Class文件中包含了Java虚拟机指令集和符号表自己若干辅助信息。

6.3 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，但反过来说，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)。

Class文件是一组以8位字节为基础单位的二进制流，中间没有添加任何分隔符。当遇到需要占用8位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储。

Class文件格式采用一种类似于C语言的结构体的伪结构来存储数据：无符号数和表。

无符号数属于基本数据类型，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。

表示有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。

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	field_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加若干个连续的数据项的格式，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。

6.3.1 魔数与Class文件的版本

每个Class文件的头四个字节称为魔数，它的唯一作用是确定这个文件是否为一个能被虚拟机接收的Class文件。Class文件的魔数为：0xCAFEBABE。

第5个和第6个字节是次版本号，第7和第8个字节是主版本号。Java的版本号是从45开始的，JDK1.1后的每个JDK大版本发布主版本号向上加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，及时文件格式未发生任何变化，虚拟机也必须拒绝执行超过其版本号的Class文件。

6.3.2 常量池

紧接着主次版本号之后的是常量池的入口，常量池可以理解为Class文件之中的资源仓库，它是Class文件结构中与其他项目关联最多的数据结构，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。

由于常量池中的常量的数量是不固定的，所以在常量池的入口需要放置一项u2类型的数据，代表常量池容量计数器。该常量计数器从1开始而不是从0开始。设计者将第0项常量空出来是有特殊考虑的，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。

常量池中主要存放两大类常量：字面量和符号引用。符号引用包含下面3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

常量池中的每一项常量都是一个表。这14种表都有一个共同的特点就是表的开始的第一位是一个u1类型的标志位，代表这个常量属于哪种常量类型。

常量池项目类型：

类型	标志	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	字符串类型的字面量
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

CONSTANT_Class_info型常量的结构：

类型	名称	数量
u1	tag	1
u2	name_index	1

其余的详见P172

6.3.3 访问标志

在常量池结束之后，紧接着的两个字节代表访问标志，这个标志用于识别一些类或接口层次的访问信息。

具体标志位以及标志的含义如下表：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	是否为public类型
ACC_FINAL	0x0010	被声明为final,只有类可设置
ACC_SUPER	0x0020	是否允许使用使用invokespecial字节码指令的新语意，invokespecial指令语义在JDK 1.0.2发生过改变，为了区别这条指令使用哪种语义，JDK 1.0.2之后编译出来的类的这个标志都必须为真。
ACC_INTERFACE	0x0200	标识这是一个接口
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

access_flags中一共有16个标志位，当前只定义了8个，没有使用的均为0。

6.3.4 类索引、父类索引和接口索引集合

类索引和父类索引都是一个u2类型的数据，而接口索引集合是一组u2类型的数据的集合，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。

父类索引只有一个，除了java.lang.Object外，所有Java类的父类索引不为0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口按implements语句后的接口顺序从左到右排列在接口索引的集合中。

类索引、父类索引和接口索引集合都按照顺序排列在访问标志之后，类索引和父类索引用两个u2类型的索引值表示，它指向一个类型为CONSTANT_Class_info的类描述符常量。

对于接口索引集合，入口的第一项——u2类型的数据为接口计数器。如果没有实现任何接口，则该计数器的值为0，后面接口的索引表不再占用任何字节。

6.3.5 字段表集合

字段表用于描述接口或者类中声明的变量。字段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。Java描述字段包括的信息：字段的作用域（public、private、protected修饰符）、是实例变量还是类变量（static修饰符）、可变性（final）、并发可见性（volatile修饰符，是否强制从主内存读写）、可否被序列化（transient修饰符）、字段数据类型（基本类型、对象、数组）、字段名称。上述的各个修饰符都是布尔值。字段叫什么名字、字段被定义为什么数据类型，只能引用常量池中的常量来描述。

字段表结构：

||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	

字段修饰符放在access_flags项目中，它与类中的access_flags项目是非常类似的，都是放在一个u2的数据类型，其中可以设置的标志位和含义见下表：

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private
ACC_PROTECTED	0x0004	字段是否protected
ACC_STATIC	0x0008	字段是否static
ACC_FINAL	0x0010	字段是否为final
ACC_VOLATILE	0x0040	字段是否volatile
ACC_TRANSIENT	0x0080	字段是否transient
ACC_SYNTHETIC	0x1000	字段是否由编译器自动产生的
ACC_ENUM	0x4000	字段是否enum

全限定名：把类全名中的“.”换成“/”，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会假如一个“;”。

简单名称：没有类型和参数修饰的方法或者字段名称。

描述符的作用是用来描述字段的数据类型、方法的参数列表（包括数量、类型以及顺序）和返回值。基本数据类型以及代表无返回的void类型都用一个大写字符来表示，而对象类型则用字符L加对象的全限定名来表示。

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	特殊类型void
L	对象类型，如Ljava/lang/Object

对于数组类型，每一维度将使用一个前置的“[”字符来描述，如一个定义为“java.lang.String[][]”的二维数组，其描述符为“[[Ljava/lang/String”。

用描述符来描述方法时，按照先参数列表，后返回值的顺序描述，参数列表按照严格顺序放在一组小括号"()"之内，例如方法int indexOf(char[] source,int sourceOffset,int sourceCount,char[] traget,int targetOffset,int targetCount,int fromIndex),其描述符为"([CI[CIII)I"。

字段表集合中不会列出从超类或者符接口中继承而来的字段，但有可能列出原本Java代码之间不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。在Java语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来说，如果两个字段的描述符不一致，那字段重名就是合法的。

6.3.6 方法表集合

方法表结构：

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

对于方法表，所有标志位以及取值参见下表：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	方法是否为public
ACC_PRIVATE	0x0002	方法是否为private
ACC_PROTECTED	0x0004	方法是否为protected
ACC_STATIC	0x0008	方法是否为static
ACC_FINAL	0x0010	方法是否为final
ACC_SYNCHRONIZED	0x0020	方法是否为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	方法是否为native
ACC_ABSTRACT	0x0400	方法是否为abstract
ACC_STRICTFP	0x0800	方法是否为strictfp
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，存在方法属性表集合中一个名为“Code”的属性里面，属性表作为Class文件格式中最具扩展性的一种数据项目。

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

在Class文件格式中，特征签名的范围更大些，只要描述符不是完全一致的两个方法也能共存。

6.3.7 属性表集合

并且只要不与已有属性名重复，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。

1.Code属性

Java程序方法体的代码经过编译后最终变为字节码指令存储在Code属性内。但并非所有的方法表都必须存在这个属性，譬如接口或者抽象类中的方法就不存在Code属性。

Code属性表的结构：

	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

attribute_name_index是一项指向CONSTANT_Utf8_info型常量的索引，常量值固定为“code”。

attribute_length指示了属性值的长度，由于属性名称索引与属性长度一共为6字节，所以属性值的长度固定为整个属性表减去6字节。

max_stack代表了操作数栈深度的最大值。

max_locals代表了局部变量表所需的存储空间。max_locals的单位是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、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两个slot来存放。方法参数（包括this）、显示异常处理器的参数、方法体中定义的局部变量都使用 局部变量表存放。

并不是在方法中用到了多少个局部变量，就把这些局部变量所占的Slot之和作为max_locals，原因是局部变量中的Slot可以重用，当代码执行超过一个局部变量的作用域时，这个局部变量所占的Slot可以被其他局部变量所使用，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。

code_length和code用来存储Java源程序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。每个指令就是一个u1类型的单字节。虽然code_length是一个u4类型的长度值，但是虚拟机规范明确限制了一个方法不允许超过65535条字节码指令，即只是用了u2的长度，如果超过这个限制，javac编译器会拒绝编译。

在字节码指令后的是这个方法的显示异常处理表集合，异常表对于Code属性来说并不是必须存在的。

异常表结构：

类型	名称	数量
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_type	1

如果字节码在第start_pc行到第end_pc行之间(不含end_pc行)出现了类型为catch_type或者其子类的异常，则转到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。

2.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的作用是列举出方法中可能抛出的受检查异常，也就是在方法描述时在throws关键字后面列举的异常。结构如下：

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_exception

Exceptions属性中的number_of_exceptions项表示方法可能抛出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，exception_index_table是一个纸箱常量池中CONSTANT_Class_info型常量的索引，代表了该受查异常的类型。

3.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。可以在javac中分别使用-g:none或-g:lines选项来取消或者要求生成这项信息。如果选择不生成LineNumberTable属性，对程序运行产生的最主要影响就是当抛出异常时，堆栈中将不会显示出错的行号。

LineNumberTable属性结构：

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_info	line_number_table_length

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。

4.LocalVariable Table属性

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，可以在Javac中分别使用-g:none或者-g:vars选项来取消或者要求生成这项信息，当没有这项属性时，其他人引用这个方法时，所有参数名称都会消失，IDE将会使用诸如arg0、arg1之类的占位符代替原有的参数名。

LocalVariableTable属性结构：

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

其中，local_variable_info项目代表了一个栈帧与源码中的局部变量的关联。

local_variable_info项目结构：

类型	名称	数量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

start_pc和length属相分别代表了这个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。

name_index和descriptor_index都是指向常量池中CONSTANT_Utf8_info型常量的索引，分别代表了局部变量的名称以及这个局部变量的描述符。

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是64位类型时（double和long），它占用的Slot为index和index+1两个。

在JDK1.5引入泛型之后，LocalVariableTable属性增加了一个“姐妹属性”：LocalVariableTypeTable，仅仅把记录字段描述符的descriptor_index替换成了字段特征签名（Signature）。

5.SourceFile属性

SourceFile属性用于记录整个生成这个Class文件的源码文件名称。可以分别使用Javac的-g:none或者-g:source选项来关闭或要求生成这项信息，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代码所述的文件名，这个属性是一个定长属性。结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

sourcefile_index数据项是指向常量池中CONSTANT_Utf8_info型常量的索引，常量值是源码文件的文件名。

6.ConstantValue属性

ConstantValue属性的作用是通知虚拟机自动为静态变量赋值。对于非static类型的变量的赋值是在实例构造器方法中进行的；而对于类变量，则有两种方式可以选择：在类构造器方法在或者使用ConstantValue属性。如果同时使用final和static来修饰一个变量，并且这个变量的数据类型是基本类型或者java.lang.String的话，就生成ConstantValue属性来初始化，如果这个变量没有被final修饰，或者并非基本变量类型及字符串，则会选择在方法中进行初始化。

虽然有final关键字才更符合“ConstantValue”的语义，但虚拟机规范中并没有前置要求字段必须设置了ACC_FINAL标志，值要求了有ConstantValue属性的字段必须设置ACC_STATIC标志而已。ConstantValue属性结构：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantValue_index	1

7.InnerClasses属性

InnerClasses属性用于记录内部类和宿主类之间的关联。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。结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	inner_class	number_of_classes

number_of_classes代表需要记录多少个内部类信息，每一个内部类信息都由一个inner_classses_info表进行描述。

inner_classes_info表结构如下：

类型	名称	数量
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_class_access_flags	1

inner_class_info_index和outter_class_info_index都是指向常量池中CONSTANT_Class_info型常量的索引，分别代表了内部类和宿主类的符号引用。

inner_name_index是指向常量池中CONSTANT_Utf8_info型常量的索引，代表这个内部类的名称，如果是匿名内部类，那么这项值为0。

inner_class_access_flags是内部类的访问标志，类似于类的access_flags，它的取值范围如下表：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	方法是否为public
ACC_PRIVATE	0x0002	方法是否为private
ACC_PROTECTED	0x0004	方法是否为protected
ACC_STATIC	0x0008	方法是否为static
ACC_FINAL	0x0010	方法是否为final
ACC_INTERFACE	0x0020	内部类是否为接口
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
ACC_ANNOTATION	0x2000	内部类是否为一个注解
ACC_ENUM	0x4000	内部类是否为一个枚举

8.Deprecated及Synthetic属性

Deprecated和Synthetic两个属性都属于标志类型的布尔属性，只存在有或没有的区别，没有属性值的概念。

Deprecated属性用于表示某各类、字段或者方法，已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注解进行设置。

Synthetic属性代表此字段或者方法并不是由Java源码直接产生的，而是由编译器自行添加的，在JDK1.5之后，标识一个类、字段或方法是编译器自动产生的，也可以设置它的ACC_SYNTHETIC标志位。所有由非用户代码产生的类、方法以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“”方法和类构造器“”方法。

结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1

其中attribute_length的数据项的值必须为0x00000000，因为没有任何属性值需要设置。

9.StackMapTable属性

StackMapTable属性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使用，目的在于代替以前比较消耗性能的数据流分析的类型推导验证器。

新的验证器在同样能保证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，而是在便一阶段将一系列的验证类型记录在Class文件中，通过检查这些验证类型代替了类型推导过程。

StackMapTable属性中包含零至多个栈映射帧，每个栈映射帧都显示或隐式地代表了一个字节的偏移量，用于表示该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属性的结构如下：

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

在版本号大于等于50.0的Class文件中吗、，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Code属性最多只能有一个StackMapTable属性，否则将抛出ClassFormatError异常。

10.Signature属性

Signature属性在JDK1.5之后加入到Class文件规范之中，是一个可选的定长属性，可以出现于类、字段表和方法表结构的属性表中。JDK1.5之后任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量或参数化类型，则Signatture属性会为它记录泛型签名信息。Java采用的泛型是伪泛型，在字节码中，泛型信息编译之后会被擦除掉。使用擦除法的好处是实现简单（主要修改Javac编译器，虚拟机只做了少量改动）、非常容易实现Backport，运行期也能节省一些类型所占的内存空间。但是无法像真正的泛型那样，将泛型类型与用户定义的普通类型同等对待。

Signature属性的结构：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	signature_index	1

其中signature_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

类签名、方法类型签名或字段类型签名。

11.BootstrapMethods属性

BootstrapMethods属性在JDK1.7发布后增加到了Class文件规范中，是一个变长属性，位于类文件的属性表中。这个属性用于保存invokedynamic指令引用的引导方法限定符。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，另外及时CONSTANT_InvokeDynamic_info类型的常量在常量池出现多次，类文件的属性表中最多也只能有一个BootstrapMethods属性。

BootstrapMethods属性的结构：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	num_bootstrap_methods	1
bootstrap_method	bootstrap_methods	num_bootstrap_methods

其中bootstrap_method结构如下：

类型	名称	数量
u2	bootstrap_method_ref	1
u2	num_bootstrap_arguments	1
u2	bootstrap_arguments	num_bootstrap_arguments

BootstrapMethods属性中，num_bootstrap_methods项的值给出了bootstrap_methods[]数组中的引导方法限定符的数量。而bootstrap_methods[]数组的每个成员包含了一个纸箱常量池CONSTANT_MethodHandle结构的索引值，它代表了一个引导方法，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如下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[]数组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：CONSTANT_String_info、CONSTANT_Integer_info、CONSTANT_Long_info、CONSTANT_Float_info、CONSTANT_Double_info、CONSTANT_MethodHandle_info、CONSTANT_MethodType_info。

6.4 字节码指令简介

Java虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字（又称操作码），以及跟随其后的零至多个代表此操作所需参数而构成。由于Java虚拟机采用面向操作数栈而不是寄存器的架构，所以大多数的指令都不包含操作数，只有一个操作码。

6.4.1 字节码与数据类型

对于大部分与数据类型相关的字节码指令，它们的操作码助记符中都有特殊的字符来表明专门为那种数据类型服务：i表示对int类型的操作，l代表long，s代表short，b代表byte，c代表char，f代表float，d代表double，a代表reference。还有部分指令只能是特定类型才能执行，因此不具有上述助记符。另外存在一些指令，与数据类型无关。

Java虚拟机的指令集对于特定的操作值提供了有限的类型相关指令去支持它。编译器会在编译器或运行期将byte和short类型的数据带符号扩展为相应的int类型数据，将boolean和char类型数据零位扩展为相应的int类型数据。与之类似，在处理boolean、byte、short、char类型的数组时，也会转换为使用对应的int类型的字节码指令来处理。因此，大多数对于boolean、byte、short、char类型数据的操作，实际上都是使用相应的int类型作为运算类型。

6.4.2 加载和存储指令

加载和存储指令用于将数据在栈帧中的局部变量表和操作数栈之间来回传输，这些指令包含如下内容：

- 将一个局部变量表加载到操作数栈：iload、iload_< n >、lload、lload_< n >、fload、fload_< n >、dload、dload_< n >、aload、aload_< n >。
- 将一个数值从操作数栈存储到局部变量表：istore、istore_< n >、lstore、lstore_< n >、fstore、fstore_< n >、dstore、dstore_< n >、astore、astore_< n >。
- 将一个常量加载到操作数栈：bipush、sipush、ldc、ldc_w、ldc2_w、aconst_null、iconst_m1、iconst_< i >、lconst_< l >、fconst_< f >、dconst_< d >。
- 扩充局部变量表的访问索引的指令：wide。

存储数据的操作数栈和局部变量表主要就是由加载和存储指令进行操作。

6.4.3 运算指令

运算或算数指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。大体上算数运算可以分为两种：对整型数据进行运算的指令和对浮点型数据进行运算的指令。整型数据和浮点型数据的算术指令在溢出和被0除时也有不同的表现。算数指令如下：

- 加法指令：iadd、ladd、fadd、dadd
- 减法指令：isub、lsub、fsub、dsub
- 乘法指令：imul、lmul、fmul、dmul
- 除法指令：idiv、ldiv、fdiv、ddiv
- 求余指令：irem、lrem、frem、drem
- 求反指令：ineg、lneg、fneg、dneg
- 位移指令：ishl、ishr、iushr、lshl、lshr、lushr
- 按位或指令：ior、lor
- 按位与指令：iand、land
- 按位异或指令：ixor、lxor
- 局部变量自增指令：iinc
- 比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp

6.4.4 类型转化指令