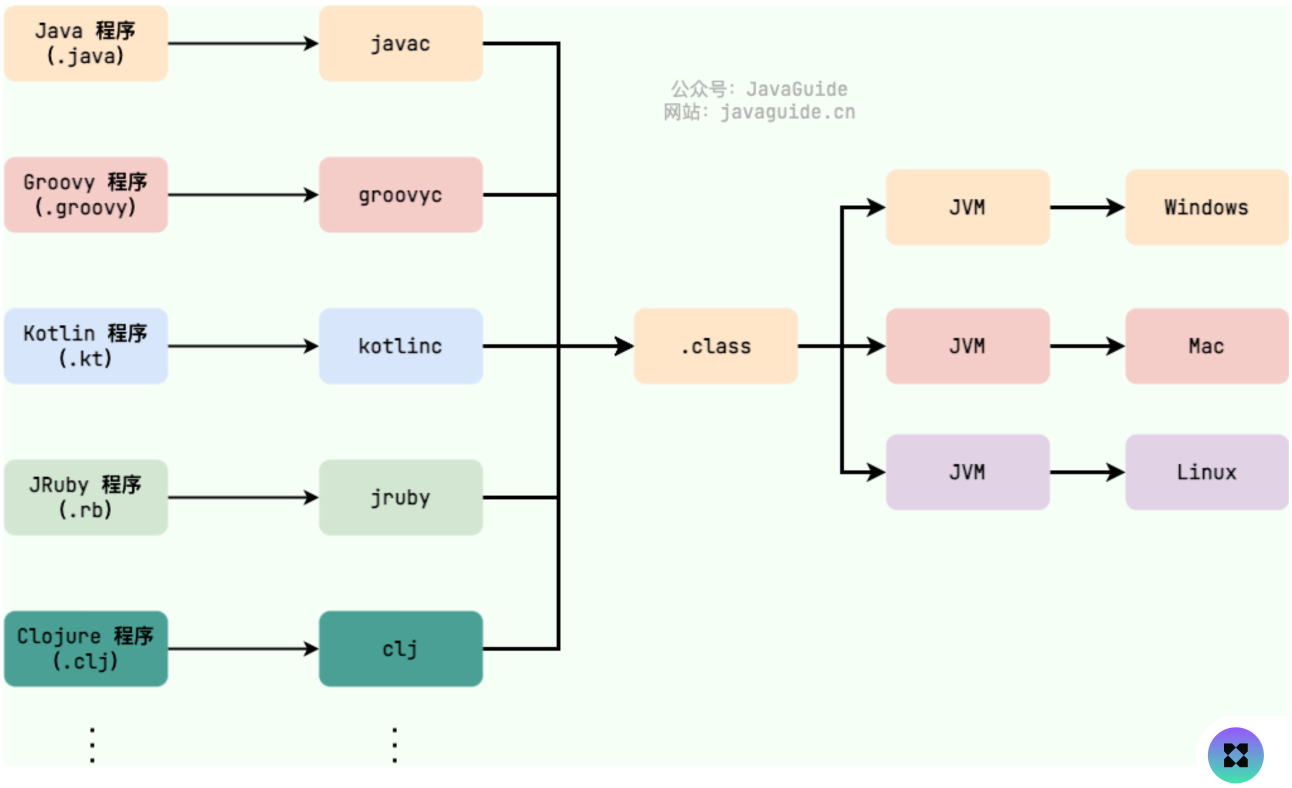
Detailed explanation of class file structure

Guide Java JVM About 2662 words About 9 minutes

Review of bytecode

In Java, the code that the JVM understands is called .h 字节码 (`.class` files with the extension). It's not processor-specific, but rather targeted only at the virtual machine. The Java language, through its bytecode format, addresses the inefficiency of traditional interpreted languages while retaining the portability of interpreted languages. This results in highly efficient runtime performance for Java programs. Furthermore, because bytecode isn't machine-specific, Java programs can run on computers with a variety of operating systems without recompilation.

Languages such as Clojure (a dialect of Lisp), Groovy, Scala, JRuby, and Kotlin all run on the Java Virtual Machine. The following diagram shows how different languages are compiled into `.class` files by different compilers and ultimately run on the Java Virtual Machine. `.class` The binary format of the files can be viewed using WinHex .



It can be said that `.class` files are an important bridge between different languages in the Java virtual machine, and it is also an important reason for supporting Java cross-platform.

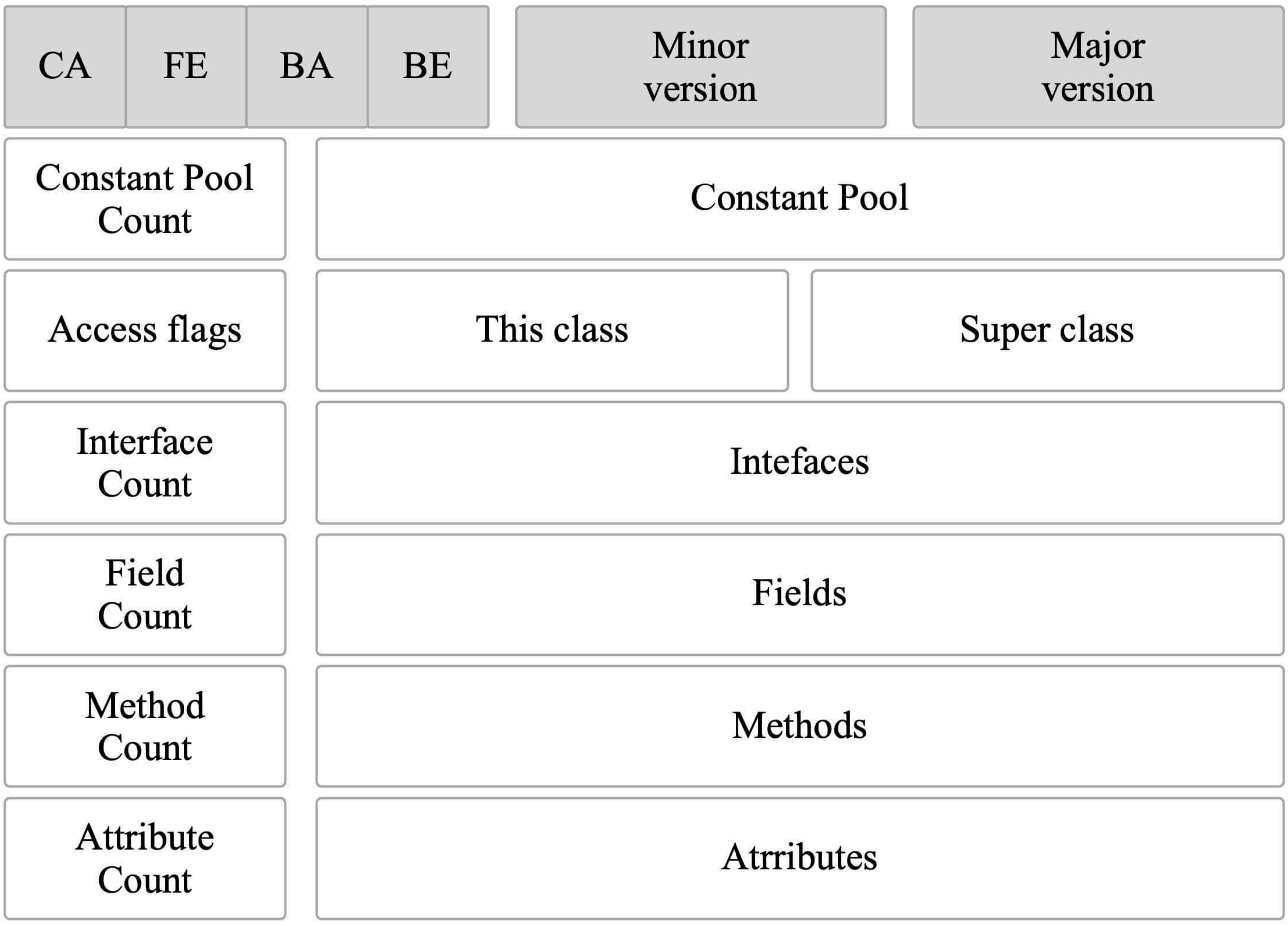
Class file structure summary

According to the Java virtual machine specification, Class files are `ClassFile` defined by , which is somewhat similar to the structure of C language.

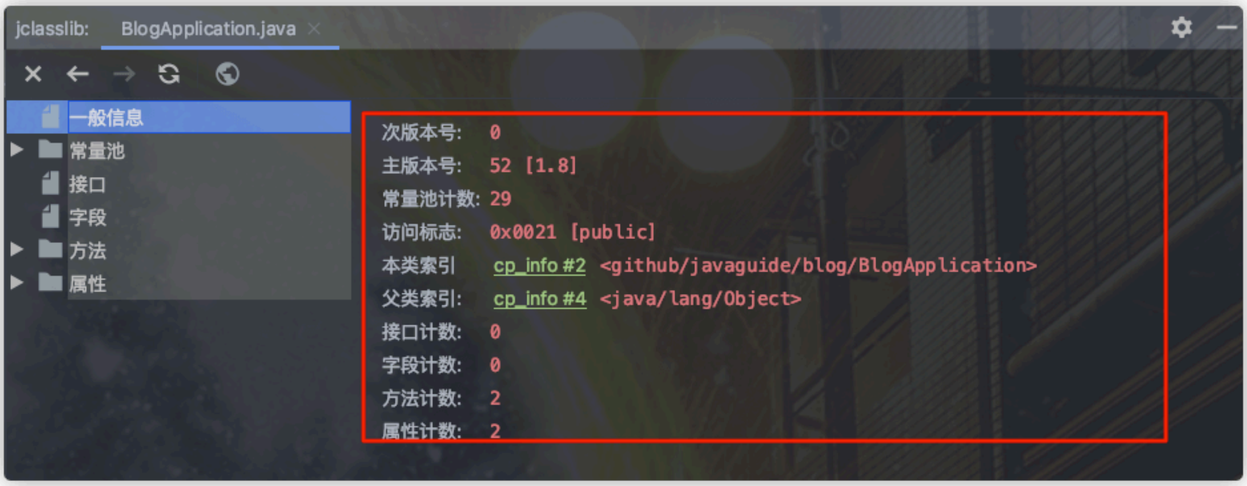
`ClassFile` The structure is as follows:

```java
ClassFile {
    u4             magic; //Class 文件的标志
    u2             minor_version;//Class 的小版本号
    u2             major_version;//Class 的大版本号
    u2             constant_pool_count;//常量池的数量
    cp_info        constant_pool[constant_pool_count-1];//常量池
    u2             access_flags;//Class 的访问标记
    u2             this_class;//当前类
    u2             super_class;//父类
    u2             interfaces_count;//接口数量
    u2             interfaces[interfaces_count];//一个类可以实现多个接
口
    u2             fields_count;//字段数量
    field_info     fields[fields_count];//一个类可以有多个字段
    u2             methods_count;//方法数量
    method_info    methods[methods_count];//一个类可以有个多个方法
    u2             attributes_count;//此类的属性表中的属性数
    attribute_info attributes[attributes_count];//属性表集合
}
```

By analyzing `ClassFile` the content, we can know the composition of the class file.



The following picture is `jclasslib` viewed through the IDEA plug-in, where you can see the Class file structure more intuitively.



```
次版本号:    0
主版本号:    52 [1.8]
常量池计数: 29
访问标志:    0x0021 [public]
本类索引    cp_info #2 <github/javaguide/blog/BlogApplication>
父类索引:    cp_info #4 <java/lang/Object>
接口计数:    0
字段计数:    0
方法计数:    2
属性计数:    2
```

Using `jclasslib` can not only visually view the bytecode file corresponding to a class, but also view the basic information, constant pool, interface, properties, functions and other information of the class.

The following is a detailed introduction to some components involved in the Class file structure.

Magic Number

```java
    u4             magic; //Class 文件的标志
```

The first four bytes of each class file are called the magic number. Its sole function is **to determine whether the file is a class file acceptable to the Java virtual machine** . The Java specification specifies a fixed value for the magic number: 0xCAFEBABE. If a file does not begin with this magic number, the Java virtual machine will refuse to load it.

Class file version number (Minor & Major Version)

```java
    u2             minor_version;//Class 的小版本号
    u2             major_version;//Class 的大版本号
```

The four bytes following the magic number store the version number of the Class file: the 5th and 6th bytes are **the minor version number** , and the 7th and 8th bytes are **the major version number** .

Whenever a major version of Java is released (such as Java 8, Java 9), the major version number will increase by 1. You can use `javap -v` the command to quickly view the version number information of the Class file.

A higher-level Java virtual machine can execute class files generated by a lower-level compiler, but a lower-level Java virtual machine cannot execute class files generated by a higher-level compiler. Therefore, during actual development, we must ensure that the JDK version used for development is consistent with the JDK version used in the production environment.

Constant Pool

```java
    u2             constant_pool_count;//常量池的数量
    cp_info        constant_pool[constant_pool_count-1];//常量池
```

Following the major and minor version numbers is the constant pool, which has a num of `constant_pool_count-1` (**the constant pool counter starts at 1, leaving the 0th constant empty for special consideration, and an index value of 0 means "do**

not reference any constant pool item").

The constant pool primarily stores two types of constants: literals and symbolic references. Literals are closer to the Java language's concept of constants, such as text strings and constant values declared final. Symbolic references, on the other hand, are a concept from the perspective of compiler theory. They include the following three types of constants:

- Fully qualified names of classes and interfaces
- Field names and descriptors
- Method name and descriptor

Each constant in the constant pool is a table. These 14 tables have one common feature: **the first bit is a u1-type flag bit - tag to identify the type of the constant, indicating which constant type the current constant belongs to.**

type	tag	describe
CONSTANT_utf8_info	1	UTF-8 encoded string
CONSTANT_Integer_info	3	Integer literals
CONSTANT_Float_info	4	Floating-point literals
CONSTANT_Long_info	5	Long integer literals
CONSTANT_Double_info	6	Double-precision floating-point literals
CONSTANT_Class_info	7	A symbolic reference to a class or interface
CONSTANT_String_info	8	String literals
CONSTANT_FieldRef_info	9	Symbolic references to fields
CONSTANT_MethodRef_info	10	Symbolic references to methods in a class
CONSTANT_InterfaceMethodRef_info	11	Symbolic reference to a method in an interface
CONSTANT_NameAndType_info	12	A symbolic reference to a fiel method

type	tag	describe
CONSTANT_MethodType_info	16	Mark method type
CONSTANT_MethodHandle_info	15	Represents a method handle
CONSTANT_InvokeDynamic_info	18	Represents a dynamic method call site

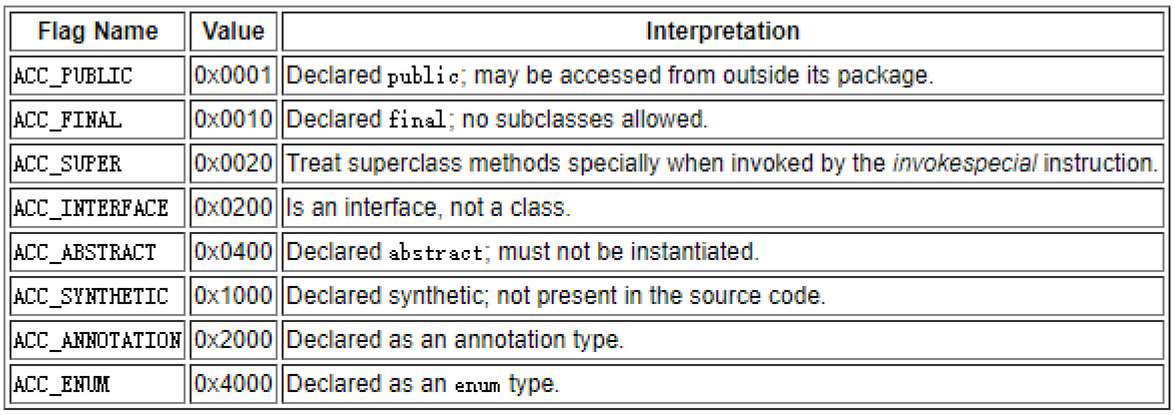
`.class` You can use `javap -v class类名` the command to view the information in the constant pool of the file (`javap -v class类名-> temp.txt` : output the result to the temp.txt file).

Access Flags

```java
    u2             access_flags;//Class 的访问标记
```

After the constant pool ends, the next two bytes represent the access flag, which is used to identify some class or interface-level access information, including: whether this Class is a class or an interface, whether it is of type `public` or `abstract` type, and if it is a class, whether it is declared as `final` and so on.

Class access and property modifiers:

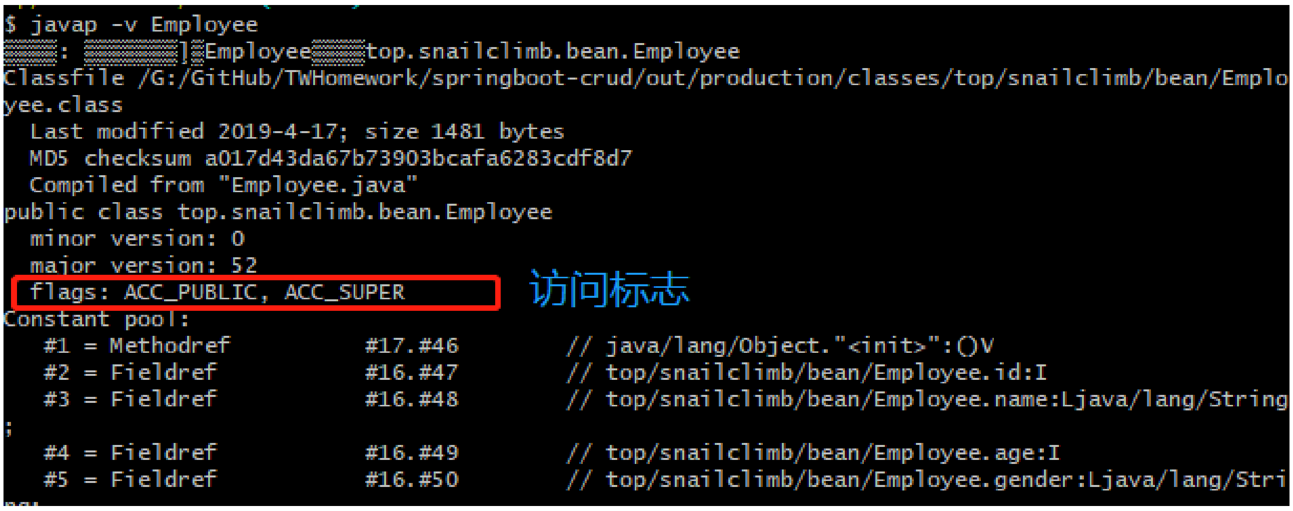


Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared `final`; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared `abstract`; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an `enum` type.

We define a `Employee` class

```java
package top.snailclimb.bean;
public class Employee {
   ...
}
```

`javap -v class类名` Take a look at the class's access flags using the directive.



```
$ javap -v Employee
▒▒▒▒: ▒▒▒▒▒▒▒]▒Employee▒▒▒▒top.snailclimb.bean.Employee
Classfile /G:/GitHub/TWHomework/springboot-crud/out/production/classes/top/snailclimb/bean/Emplo
yee.class
  Last modified 2019-4-17; size 1481 bytes
  MD5 checksum a017d43da67b73903bcafa6283cdf8d7
  Compiled from "Employee.java"
public class top.snailclimb.bean.Employee
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #17.#46        // java/lang/Object."<init>":()V
   #2 = Fieldref           #16.#47        // top/snailclimb/bean/Employee.id:I
   #3 = Fieldref           #16.#48        // top/snailclimb/bean/Employee.name:Ljava/lang/String
;
   #4 = Fieldref           #16.#49        // top/snailclimb/bean/Employee.age:I
   #5 = Fieldref           #16.#50        // top/snailclimb/bean/Employee.gender:Ljava/lang/Stri
```

This Class, Super Class, and Interfaces index collections

```java
    u2             this_class;//当前类
    u2             super_class;//父类
    u2             interfaces_count;//接口数量
    u2             interfaces[interfaces_count];//一个类可以实现多个接
```

The inheritance relationship of a Java class is determined by the class index, parent class index, and interface index set. The class index, parent class index, and interface index set are arranged in order after the access flag.

The class index is used to determine the fully qualified name of this class, and the parent class index is used to determine the fully qualified name of the parent class of this class. Due to the single inheritance of the Java language, there is only one parent class index. Except `java.lang.Object` for , all Java classes have a parent class, so except `java.lang.Object` for , the parent class index of all Java classes is not 0.

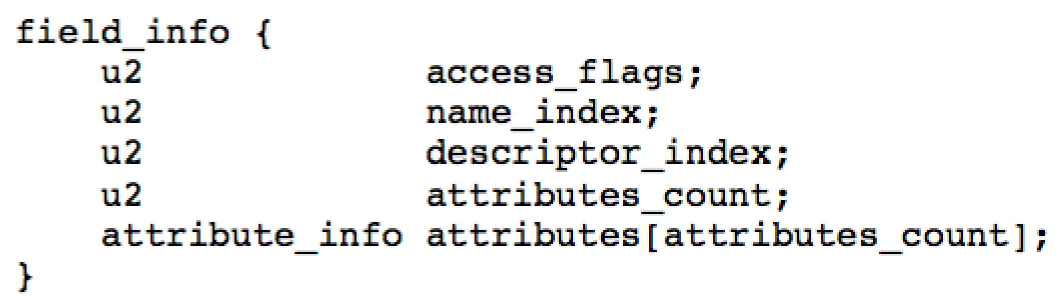
The interface index set is used to describe which interfaces this class implements. These implemented interfaces will be arranged from left to right in the interface index set in the order of the interfaces `implements` (if the class itself is an interface). `extends`

Fields

```java
    u2             fields_count;//字段数量
    field_info     fields[fields_count];//一个类会可以有个字段
```

The field info table describes the variables declared in an interface or class. Fields include class-level variables and instance variables, but do not include local variables declared within methods.

The structure of field info (field table):

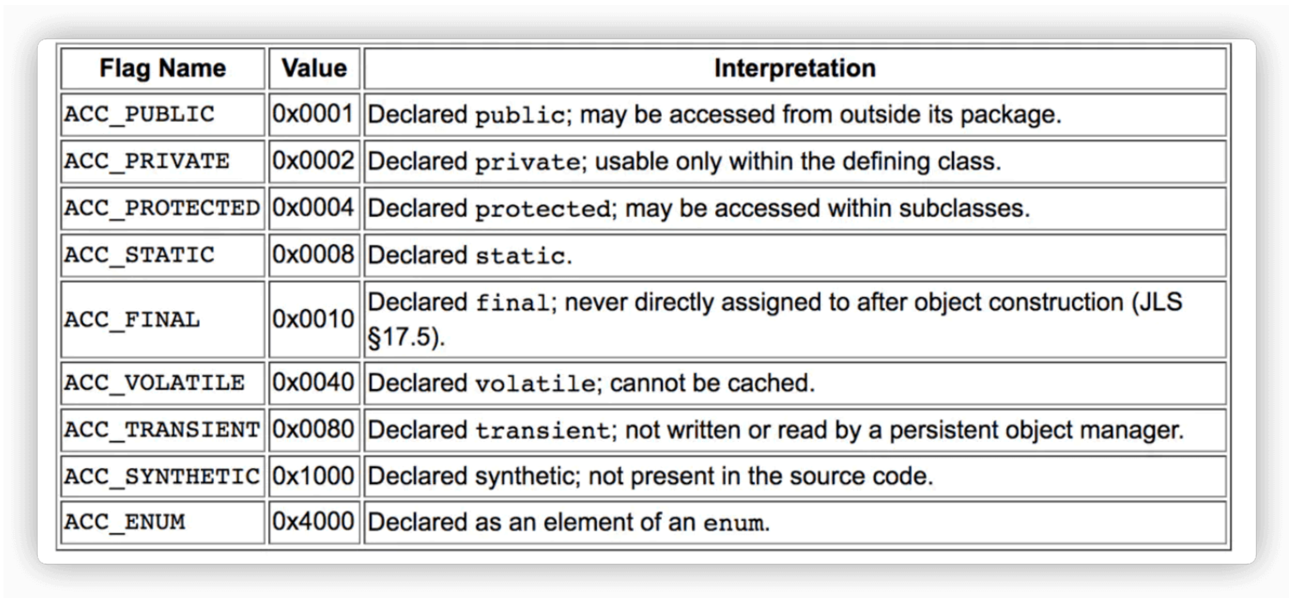


```
field_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

- **access_flags:** The scope of the field (,, `public` modifiers), whether it is an instance variable or a class variable (modifiers), whether it can be serialized (transient modifier), mutability (final), visibility (volatile modifier, whether to force reading and writing from main memory). `private` `protected` `static`
- **name_index:** a reference to the constant pool, representing the name of the field;
- **descriptor_index:** a reference to the constant pool, representing the descriptor of the field and method;
- **attributes_count:** A field may also have some additional attributes, attributes_count stores the number of attributes;
- **attributes[attributes_count]:** stores the specific content of specific attributes.

In the above information, each modifier is a Boolean value; it either has a modifier or not, making it well-suited for use as a flag. However, the field name and data type cannot be fixed and can only be described by referencing constants in the constant pool.

The value of the access_flag field:



Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared volatile; cannot be cached.
ACC_TRANSIENT	0x0080	Declared transient; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an enum.

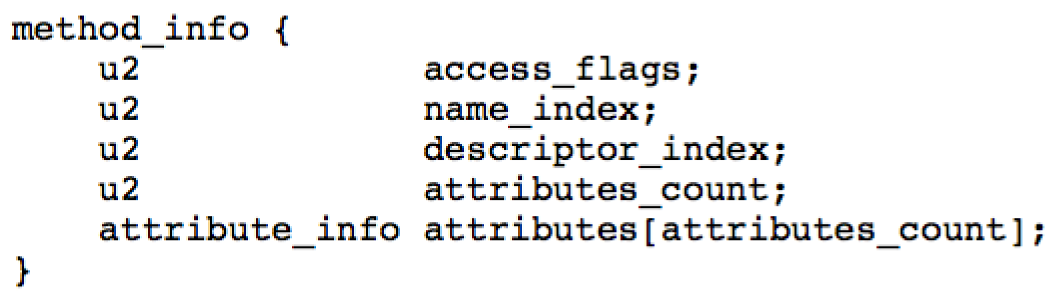
Methods collection

```java
    u2             methods_count;//方法数量
    method_info    methods[methods_count];//一个类可以有个多个方法
```

methods_count indicates the number of methods, and method_info indicates the method table.

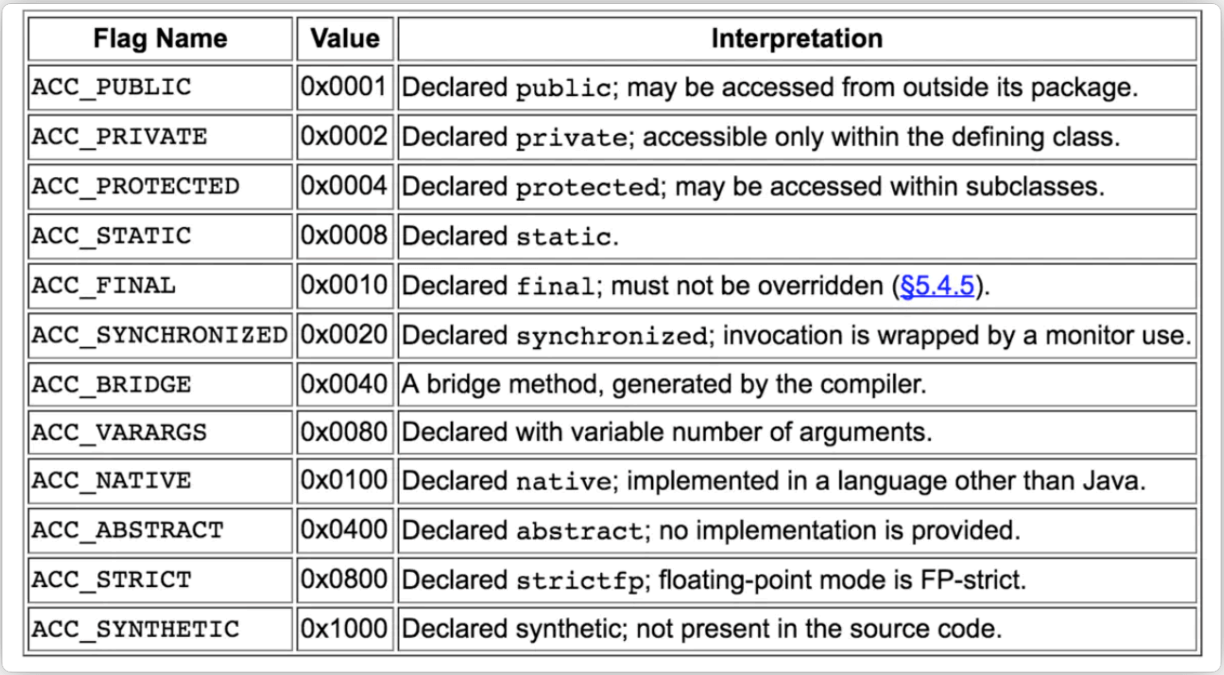
The class file storage format uses almost identical descriptions for methods and fields. The structure of a method table is similar to that of a field table, consisting of an access flag, a name index, a descriptor index, and a property table set.

method_info (method table) structure:



```
method_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

The access_flag value of the method table:



Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; must not be overridden (§5.4.5).
ACC_SYNCHRONIZED	0x0020	Declared synchronized; invocation is wrapped by a monitor use.
ACC_BRIDGE	0x0040	A bridge method, generated by the compiler.
ACC_VARARGS	0x0080	Declared with variable number of arguments.
ACC_NATIVE	0x0100	Declared native; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared abstract; no implementation is provided.
ACC_STRICT	0x0800	Declared strictfp; floating-point mode is FP-strict.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.

Note: Because `volatile` modifiers and `transient` modifiers cannot modify methods, there are no corresponding flags in the access flags of the method table. However, keywords such as `synchronized` , `native` , `abstract` and so on are added to modify methods, so there are more flags corresponding to these keywords.

Attributes collection

```java
    u2             attributes_count;//此类的属性表中的属性数
    attribute_info attributes[attributes_count];//属性表集合
```

Class files, field tables, and method tables can all carry their own set of attribute tables to describe scenario-specific information. Unlike other data items in class files, which have strict ordering, length, and content requirements, attribute tables have slightly looser restrictions. There's no strict ordering requirement for attribute tables. Furthermore, any compiler implementation can write custom attribute information to an attribute table, as long as the name doesn't overlap with existing ones. The Java Virtual Machine will ignore any attributes it doesn't recognize at runtime.

refer to

- Practical Java Virtual Machine

- Chapter 4. The class File Format - Java Virtual Machine Specification: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html
- Example analysis of the file structure of JAVA CLASS: https://coolshell.cn/articles/9229.html
- Java Virtual Machine Principles Diagram 1.2.2, Detailed Explanation of the Constant Pool in the Class File (Part 1): https://blog.csdn.net/luanlouis/article/details/39960815

Recently Updated2025/6/30 15:20

Contributors: SnailClimb , tianyu94 , Shuang Kou , shuang.kou , what , guide , hailong.sha , Gale , Guide , Erzbir , Mr.Hope , wym199807 , fu , Kisa-Dong