链滴

Class 文件结构分析

作者：vcjmhg

原文链接：https://ld246.com/article/1618833195900

来源网站：链滴

概述

在Java语言中，Java虚拟机只能理解字节码（class文件），它不面向任何处理器，不与任何语言绑定只与Class文件这种特定的二进制文件格式所关联。

Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

另一方面由于JVM虚拟机不与任何语言、机器绑定，因而任何语言的实现着都可以将Java虚拟机作为言的运行基础，以Class文件作为他们的交付媒介。例如Clojure（Lisp 语言的一种方言）、Groovy、Ruby等语言都是运行在 Java 虚拟机之上。

下图展示了不同的语言被不同的编译器编译成.class文件最终运行在 Java 虚拟机之上的过程：

Class文件的结构

根据《Java虚拟机规范》，Class 文件通过 ClassFile 定义，而且文件结构采用一种类似c语言**结构体**的**结构体**。这种伪结构体只有两种两种数据类型：“无符号数”和“表”。

- **无符号数**：属于基本的数据结构，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节、8字节的无符号数。无符号数可以用来描述数字、索引引用、数字量值或者按照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**复合数据结构**，为了便于区分，所有表的命名都惯以 **"_info"** 结尾。

在正式开始讲，我们需要说明一点，Class文件的结构不像XML那样的结构化描述语言，它以**8个字节**基础单位，各个数据项目严格按照顺序紧凑地排列在文件中，中间**没有添加任何分隔符号**，因而在Class的数据项无论是顺序还是数量，甚至数据存储的**字节序**（大端存储，Big-Endian）这样的细节，都被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，全部都不允许改变。

ClassFile 的结构如下：

```
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
    u2            interfaces[interfaces_count];//一个类可以实现多个接口
    u2            fields_count;//Class 文件的字段属性
    field_info    fields[fields_count];//一个类会可以有多个字段
    u2            methods_count;//Class 文件的方法数量
    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
}
```

通过对ClassFile的分析，我们便可以知道class文件的组成。

上边的一些属性什么的，描述都很抽象，因而这里以一段典型的java代码产生的class文件为基础结合行讲解。

一段典型的Java程序代码如下：

```
package com.test;

//接口类
interface Car {
    void drive();
}
//实现类
public class BMWCar implements Car{

    private String name;

    public BMWCar() {
        name = "宝马";
    }

    @Override
    public void drive() {
        System.out.println("BMW car drive." + name);
    }
}
```

通过javac命令对代码进行编译，生成的class文件内容如下：

```
  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: CA FE BA BE 00 00 00 3B 00 33 07 00 02 01 00 0F    J~:>...;.3.....
```

原文链接：Class 文件结构分析

```
00000010: 63 6F 6D 2F 74 65 73 74 2F 42 4D 57 43 61 72 07    com/test/BMWCar.
00000020: 00 04 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F    .....java/lang/O
00000030: 62 6A 65 63 74 07 00 06 01 00 0C 63 6F 6D 2F 74    bject......com/t
00000040: 65 73 74 2F 43 61 72 01 00 04 6E 61 6D 65 01 00    est/Car...name..
00000050: 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69    .Ljava/lang/Stri
00000060: 6E 67 3B 01 00 06 3C 69 6E 69 74 3E 01 00 03 28    ng;...<init>...(
00000070: 29 56 01 00 04 43 6F 64 65 0A 00 03 00 0D 0C 00    )V...Code.......
00000080: 09 00 0A 08 00 0F 01 00 06 E5 AE 9D E9 A9 AC 09    .........e..i),.
00000090: 00 01 00 11 0C 00 07 00 08 01 00 0F 4C 69 6E 65    ............Line
000000a0: 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 12 4C 6F    NumberTable...Lo
000000b0: 63 61 6C 56 61 72 69 61 62 6C 65 54 61 62 6C 65    calVariableTable
000000c0: 01 00 04 74 68 69 73 01 00 11 4C 63 6F 6D 2F 74    ...this...Lcom/t
000000d0: 65 73 74 2F 42 4D 57 43 61 72 3B 01 00 05 64 72    est/BMWCar;...dr
000000e0: 69 76 65 09 00 18 00 1A 07 00 19 01 00 10 6A 61    ive...........ja
000000f0: 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 0C 00    va/lang/System..
00000100: 1B 00 1C 01 00 03 6F 75 74 01 00 15 4C 6A 61 76    ......out...Ljav
00000110: 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D    a/io/PrintStream
00000120: 3B 07 00 1E 01 00 17 6A 61 76 61 2F 6C 61 6E 67    ;.......java/lang
00000130: 2F 53 74 72 69 6E 67 42 75 69 6C 64 65 72 08 00    /StringBuilder..
00000140: 20 01 00 0E 42 4D 57 20 63 61 72 20 64 72 69 76    ....BMW.car.driv
00000150: 65 2E 0A 00 1D 00 22 0C 00 09 00 23 01 00 15 28    e.....".....#...(
00000160: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E    Ljava/lang/Strin
00000170: 67 3B 29 56 0A 00 1D 00 25 0C 00 26 00 27 01 00    g;)V....%..&.'..
00000180: 06 61 70 70 65 6E 64 01 00 2D 28 4C 6A 61 76 61    .append..-(Ljava
00000190: 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 4C 6A    /lang/String;)Lj
000001a0: 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 42    ava/lang/StringB
000001b0: 75 69 6C 64 65 72 3B 0A 00 1D 00 29 0C 00 2A 00    uilder;....)..*.
000001c0: 2B 01 00 08 74 6F 53 74 72 69 6E 67 01 00 14 28    +...toString...(
000001d0: 29 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69    )Ljava/lang/Stri
000001e0: 6E 67 3B 0A 00 2D 00 2F 07 00 2E 01 00 13 6A 61    ng;..-./......ja
000001f0: 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61    va/io/PrintStrea
00000200: 6D 0C 00 30 00 23 01 00 07 70 72 69 6E 74 6C 6E    m..0.#...println
00000210: 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 0B    ...SourceFile...
00000220: 42 4D 57 43 61 72 2E 6A 61 76 61 00 21 00 01 00    BMWCar.java.!...
00000230: 03 00 01 00 05 00 01 00 02 00 07 00 08 00 00 00    ................
00000240: 02 00 01 00 09 00 0A 00 01 00 0B 00 00 00 3D 00    ..............=.
00000250: 02 00 01 00 00 00 0B 2A B7 00 0C 2A 12 0E B5 00    .......*7..*..5.
00000260: 10 B1 00 00 00 02 00 12 00 00 00 0E 00 03 00 00    .1..............
00000270: 00 0D 00 04 00 0E 00 0A 00 0F 00 13 00 00 00 0C    ................
00000280: 00 01 00 00 00 0B 00 14 00 15 00 00 00 01 00 16    ................
00000290: 00 0A 00 01 00 0B 00 00 00 48 00 04 00 01 00 00    .........H......
000002a0: 00 1A B2 00 17 BB 00 1D 59 12 1F B7 00 21 2A B4    ..2..;..Y..7.!*4
000002b0: 00 10 B6 00 24 B6 00 28 B6 00 2C B1 00 00 00 02    ..6.$6.(6.,1....
000002c0: 00 12 00 00 00 0A 00 02 00 00 00 13 00 19 00 14    ................
000002d0: 00 13 00 00 00 0C 00 01 00 00 00 1A 00 14 00 15    ................
000002e0: 00 00 00 01 00 31 00 00 00 02 00 32                .....1.....2
```

魔数

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

每一个Class文件的**头4个字节**被称为魔数，它唯一的作用就是确定这个文件是否是一个能够被虚拟机受的Class文件。

其在class文件中的具体位置如下图所示：

Clsss文件的魔数选的很有浪漫气息，值为0xCAFFEBABY（咖啡宝贝？）

Class 文件版本号（Minor&Major Version）

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

紧跟着魔数的4个字节存储的是CLass文件的版本号：第5和第6个字节是**次版本号**（Minor Version）第7和8个字节存储的是**主版本号**（Manjor Version）。

每当 Java 发布大版本（比如 Java 8，Java9）的时候，主版本号都会加 1。你可以使用 javap -v 命来快速查看 Class 文件的版本号信息。

高版本的 Java 虚拟机可以执行低版本编译器生成的 Class 文件，但是低版本的 Java 虚拟机不能执高版本编译器生成的 Class 文件。 所以，我们在实际开发的时候要确保开发的的 JDK 版本和生产环的 JDK 版本保持一致。

class版本号具体位置如下图所示:

从图中可以看到，我们class文件的主版本号是0x003B，也就是十进制的59，这个版本说明是可以被JK15及 其以上版本的虚拟机运行。

常量池（Constant Pool）

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

紧接着主、次版本号之后的是常量池的入口，常量池的**数量**为constant_pool_count - 1（**常量池计器是从 1 开始计数的，将第 0 项常量空出来是有特殊考虑的，索引值为 0 代表“不引用任何一个常池项”**）。

常量池中主要存放两大类常量：

- **字面量**：比较**接近于Java语言层面的常量**概念，如文本字符串、被声明的final

"符号引用"：属于编译原理方面的概念，主要包括三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

常量池中，每一项常量都是一个**表**，截止到JDK13，常量表中共有**17种**不同类型的常量，它们有一个同的特点即表结构起始的第一位是一个u1类型的标志位(tag)，代表当前常量属于哪种常量。

17中常量及其所对应的标志位如下表所示：

类型	标志（tag）	描述
CONSTANT_utf8_info	1	UTF-8 编码的字符串

CONSTANT_Integer_info 3 整
字面量

CONSTANT_Float_info 4 浮点
字面量

CONSTANT_Long_info 5 长整
字面量

CONSTANT_Double_info 6
精度浮点型字面量

CONSTANT_Class_info 7 类或
口的符号引用

CONSTANT_String_info 8 字
串类型字面量

CONSTANT_Fieldref_info 9
段的符号引用

CONSTANT_Methodref_info 10
中方法的符号引用

CONSTANT_InterfaceMethodref_info 11
口中方法的符号引用

CONSTANT_NameAndType_info 12
段或方法的符号引用

CONSTANT_MothodType_info 16
志方法类型

CONSTANT_MethodHandle_info 15
示方法句柄

CONSTANT_InvokeDynamic_info 18
示一个动态方法调用点

结合前边的Class文件：

可以看到常量池中常量的数量为0x33即有**50个常量**（因为从1开始计数），通过javap -v BMWCar
令可以查看Class文件的信息如下：

```
Classfile /C:/Users/vcjmhg/Desktop/test/com/test/BMWCar.class
  Last modified 2021-4-17; size 748 bytes
  MD5 checksum e3bb3d3eaf56cc12d92423d7b99781d2
  Compiled from "BMWCar.java"
public class com.test.BMWCar implements com.test.Car
  minor version: 0
  major version: 59
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2             // com/test/BMWCar
   #2 = Utf8               com/test/BMWCar
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Class              #6             // com/test/Car
   #6 = Utf8               com/test/Car
   #7 = Utf8               name
```

```
   #8 = Utf8               Ljava/lang/String;
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Methodref          #3.#13         // java/lang/Object."<init>":()V
  #13 = NameAndType        #9:#10         // "<init>":()V
  #14 = String             #15
  #15 = Utf8
  #16 = Fieldref           #1.#17         // com/test/BMWCar.name:Ljava/lang/String;
  #17 = NameAndType        #7:#8          // name:Ljava/lang/String;
  #18 = Utf8               LineNumberTable
  #19 = Utf8               LocalVariableTable
  #20 = Utf8               this
  #21 = Utf8               Lcom/test/BMWCar;
  #22 = Utf8               drive
  #23 = Fieldref           #24.#26        // java/lang/System.out:Ljava/io/PrintStream;
  #24 = Class              #25            // java/lang/System
  #25 = Utf8               java/lang/System
  #26 = NameAndType        #27:#28        // out:Ljava/io/PrintStream;
  #27 = Utf8               out
  #28 = Utf8               Ljava/io/PrintStream;
  #29 = Class              #30            // java/lang/StringBuilder
  #30 = Utf8               java/lang/StringBuilder
  #31 = String             #32            // BMW car drive.
  #32 = Utf8               BMW car drive.
  #33 = Methodref          #29.#34        // java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
  #34 = NameAndType        #9:#35         // "<init>":(Ljava/lang/String;)V
  #35 = Utf8               (Ljava/lang/String;)V
  #36 = Methodref          #29.#37        // java/lang/StringBuilder.append:(Ljava/lang/String;)Lja
a/lang/StringBuilder;
  #37 = NameAndType        #38:#39        // append:(Ljava/lang/String;)Ljava/lang/StringBuilder

  #38 = Utf8               append
  #39 = Utf8               (Ljava/lang/String;)Ljava/lang/StringBuilder;
  #40 = Methodref          #29.#41        // java/lang/StringBuilder.toString:()Ljava/lang/String;
  #41 = NameAndType        #42:#43        // toString:()Ljava/lang/String;
  #42 = Utf8               toString
  #43 = Utf8               ()Ljava/lang/String;
  #44 = Methodref          #45.#47        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #45 = Class              #46            // java/io/PrintStream
  #46 = Utf8               java/io/PrintStream
  #47 = NameAndType        #48:#35        // println:(Ljava/lang/String;)V
  #48 = Utf8               println
  #49 = Utf8               SourceFile
  #50 = Utf8               BMWCar.java
{
  public com.test.BMWCar();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #12                 // Method java/lang/Object."<init>":()V
         4: aload_0
```

```
         5: ldc           #14                 // String
         7: putfield      #16                 // Field name:Ljava/lang/String;
        10: return
      LineNumberTable:
        line 13: 0
        line 14: 4
        line 15: 10
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      11     0  this   Lcom/test/BMWCar;

  public void drive();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=4, locals=1, args_size=1
         0: getstatic     #23                 // Field java/lang/System.out:Ljava/io/PrintStream;
         3: new           #29                 // class java/lang/StringBuilder
         6: dup
         7: ldc           #31                 // String BMW car drive.
         9: invokespecial #33                 // Method java/lang/StringBuilder."<init>":(Ljava/lang/Str
ng;)V
        12: aload_0
        13: getfield      #16                 // Field name:Ljava/lang/String;
        16: invokevirtual #36                 // Method java/lang/StringBuilder.append:(Ljava/lang/Str
ng;)Ljava/lang/StringBuilder;
        19: invokevirtual #40                 // Method java/lang/StringBuilder.toString:()Ljava/lang/St
ing;
        22: invokevirtual #44                 // Method java/io/PrintStream.println:(Ljava/lang/String;)

        25: return
      LineNumberTable:
        line 19: 0
        line 20: 25
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      26     0  this   Lcom/test/BMWCar;
}
SourceFile: "BMWCar.java"
```

首先我们尝试对**第一个常量**进行解析，首先找到它对应的标志位（表的第一个字节）为7，查询上边常量表可知，该常量为一个CONSTANT_CLASSS_info（**类或者接口的符号引用**）

查询CONSTANT_class_info的结构如下：

类型	名称	数量
u1	tag	标志位
u2	name_index	1

tag位前边我们说了，它是所有表的一个共同特征，用来指明表的类型；

name_index是常量池的索引值，指向常量池中一个CONSTANT_Utf8_info类型常量，代表这个类的限定名。

由于第一个常量的name_index = 2，也就是**指向了常量池中的第二个常量。**

首先可以看到它的tag=1，是一个CONSTANT_UTF8_info类型的常量，该类型的结构表如下图所示：

类型	名称	数量
u1	tag	1
u2	length	1
u1	bytes	length

length属性说明这个UTF-8编码的字符串的长度是多少个字节，后边紧跟着length字节的连续数据表一个使用**UTF-8缩略编码**表示的字符串。

说明：此处缩略编码与普通UTF编码的区别在于：从'\u0001'到'\u07ff'（相当于Ascii编码1到217）用**一个字节**编码，从'\u0080'到 '\u007f'之间的字符使用**两个字节编码**，剩余部分按照普通UTF-8编规则使用**三个字节**进行编码。

我们可以看到该**字符串长度为** 0x000f即有**15个字节**，然后紧接着15个字节构成了该字符串的值:com/est/BMWCar。

将前边两个常量结合在一起我们就了解到该类的全限定名为：com/test/BMWCar。

其他常量分析与之类似，我们计算出常量池在class中所占用的空间位置如下图所示：

访问标志(Access Flags)

常量池结束之后，紧接着的2个字节表示Class的访问标志（access_flags），这个标志用来识别类或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型；如果是类的话，是否定义为final等等。

具体的标志位及其含义如下表所示：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	是否为public类型
ACC_FINAL	0x0010	是否被声为final，只有类可设置
ACC_SUPER	0x0020	是否允许用invokespecial字节码指令新语义，JDK1.0.2之后都为true
ACC_INTERFACE	0x0200	标志是个接口
ACC_ABSTRACT	0x0400	是否是bstract类型，对于抽象类或者接口来说为true，其他情况为false
ACC_SYNTHETIC	0x1000	标识个类并非由用户代码产生
ACC_ANNOTATION	0x2000	识这是一个注解
ACC_ENUM	0x4000	标识这是

个枚举

ACC_MODULE	0x8000	标识这一个模块

access_flags中一共有**16个标志**位可以使用，**当前只定义了9个**，没有使用到的标志位一律为零（工上的一种冗余设计思想，值得学习[grin]
lush）。

结合我们的Class文件，可以看到该文件的访问标志为0x0021相当于0x0020 | 0x0001查询访问标志可知，该类是一个**public**类型且可以使用invokespecial指令新语义的普通类。

类索引（This Class）、父类索引（Super Class）、接口（Interfaces）索引集合

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

访问标识之后，紧接着的便是类索引、父类索引与接口索引集合。其中类索引、父类索引都是一个u2型数据，而接口索引集合是一个**u2类型的数据集合**，这三项数据构成了Class文件的继承关系。

类索引用来确定这个类的**全限定名**，父类索引用来用于确定这个类的**父类的全限定名**。由于Java不允多重继承，所以**父类索引有只有一个**(java.lang.Object类除外)。

接口索引集合就是用来描述这个类实现了哪些接口，这些接口将按照implements关键字（如果这个Class文件表示的是一个接口，则应当使用extends关键字）后的接口顺序**从左到右排列**在接口索引集合。

类索引和父类索引使用两个u2类型的索引值表示，它们各自指向一个类型为CONSTANT_Class_info类描述常量，进而找到一个定义在CONSTANT_Utf8_info类型中的索引**全限定名字符串**。

结合前边的Class文件，类索引查找全限定名的过程如下图所示：

首先根据从Class索引值为0x0001也即指向常量池中的第一个**常量**，该常量是一个CONSTANT_Class_nfo类型的数据，该数据的权限定名称指向了常量池中的第三个常量，第三个常量的常量值是com/testBMWCar，将整个过程结合在一起我们就知道该类文件的全限定名为com/test/BMWCar。

父类索引的查找过程与之类似，此处不再详述，最终可以定位到该类文件的父类为java/lang/Object。

接口索引由于是集合类型，查找过程与类查找过程可能有些许不同：

首先找到第一个u2类型接口计数器，其值为0x0001也就是说该类文件实现了一个接口，其接口索引为x0005即接口索引指向常量池中**第五个常量**。常量#5为一个CONSTAN_Class_info类型的常量，指向六个CONSTANT_Utf8_info类型的常量#6，该常量的值为com/test/Car。

整个分析下来，我们可以得到该Class文件是一个实现了一个全限定名为com/test/Car接口的类。

字段表集合（Fields）

```
u2          fields_count;//Class 文件的字段的个数
field_info  fields[fields_count];//一个类会可以有个字段
```

接口索引后边紧跟着的就是字段表信息，**字段表（field info）用于描述接口或类中声明的变量**。字包括类级变量以及实例变量，但**不包括**在方法内部声明的**局部变量**。

字段表的结构如下表所示：

类型	名称	数量	备注
u2	access_flags	1	字段作用域（public、private、protected修饰符），是实例变量还是类变量，可否被序列化（transient饰符），可变性（final），可见性（volatitle修饰符，是否强制从主内存读写）
u2	name_index	1	对常池的引用，表示字段的简单名称
u2	descriptor_index	1	常量池的引用，表示字段和方法的**描述符**
u2	attributes_count	1	个字段可能会额外拥有一些属性，attributes_count用来存放属性的数量
attribute_info	attributes	attributes_count	放属性的具体内容

字段访问access_flags的标志及其含义如下表所示：

权限名称	值	描述
ACC_PUBLIC	0x0001	public
ACC_PRIVATE	0x0002	private
ACC_PROTECTED	0x0004	prot cted
ACC_STATIC	0x0008	static，态
ACC_FINAL	0x0010	final
ACC_VOLATILE	0x0040	volatil，不可和ACC_FIANL一起使用
ACC_TRANSIENT	0x0080	在序化中被忽略的字段
ACC_SYNTHETIC	0x1000	由编器产生，不存在于源代码中
ACC_ENUM	0x4000	enum

紧随access_flags标志的是name_index和descriptor_index，他们都是对常量池的引用。name_index表着字段的简单名称，descriptor_index代表着字段的描述符。相比于全限定名和简单名称，方法和段的描述符要复杂一些。

描述符的**主要作用是**用来描述字段的数据类型、方法和参数列表（包括数量类型以及顺序）和返回值因而描述符在设计时，设计了一系列描述规则：

1. 基本数据类型（byte、char、double、float、int、long、short、boolean）以及代表无返回值的

oid类型都用一个大写字符来表示。

2. 对象类型则用 **字符L加上对象的全限定名**来表示

描述符标识字含义如下表所示：

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;
[	数组类型，多个维度则有多个[

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，参数列表按照参数的严格顺序放在一“()”之内。

例如方法int getAge()的描述符为“**()I**”，方法void print(String msg)的描述符为“**(Ljava/lang/String;)V**”，方法int indexOf(int index, char[] arr)的描述符为“**(I[C)I**”

结合我们的Class文件，可以看到该类的第一个方法是构造方法，方法名称为com.test.BMWCar，描符为()V，也即是**一个入参为空且返回值为空的函数。**

方法表集合（Methods）

```
u2             methods_count;//Class 文件的方法的数量
method_info    methods[methods_count];//一个类可以有个多个方法
```

字段表集合结束之后，紧接着就是方法表集合，与字段表的结构一样，一次包括访问标志（access_flags）、名称索引（name_index）、描述符(descriptor_index)、属性表集合（attributes)几项，具体结构下表所示：

类型	描述	备注
u2	access_flags	记录方法的访问标志
u2	name_index	常量池中的索引项，指定方法的名称
u2	descriptor_index	常量池中索引项，指定方法的描述符
u2	attributes_count	attributes包含的项目数

原文链接：Class 文件结构分析

attribute_info attributes[attributes_count]
放属性的具体内容

具体方法标志及其含义如下表所示：

权限名称	值	描述
ACC_PUBLIC	0x0001	public
ACC_PRIVATE	0x0002	private
ACC_PROTECTED	0x0004	prot cted
ACC_STATIC	0x0008	static，态
ACC_FINAL	0x0010	final
ACC_SYNCHRONIZED	0x0020	法是否为synchronized
ACC_BRIDGE	0x0040	方法是是由编译器产生的桥接方法
ACC_VARARGE	0x0080	方法否接受不定参数
ACC_NATIVE	0x0100	方法是为native
ACC_ABSTRACT	0x0400	方法否为abstract
ACC_SYNTHETIC	0x0800	方法否为strictfp
ACC_SYNTHETIC	0x1000	由编器产生，不存在于源代码中

与属性表的方法标志进行比较，我们不难发现，两者大体上是类似的，但有诸多不同之处：

因为volatile关键字和transient关键字不能够修饰方法，所以方法表的访问标志中没有了ACC_VOLATILE标志和ACC_TRANSIENT标志。与之相对应的，synchronized、native、strictfp和abstract等关键可以修改方法但不能修饰属性，因此增加了ACC_SYNCHRONIZED、ACC_NATIVE、ACC_SYNTHETC、ACC_ABSTRACT。

分析到这里，可能会有小伙伴有疑问了：前面说的好像都只是方法的定义，那方法的**主体逻辑代码**怎描述呢?

简单来说，方法体中的Java代码，经过Javac编译成字节码指令之后，存放在方法属性中的一个名为Code的属性里面了。

结合我们的Class文件可以看到，该文件中有一个drive()方法，该方法的入参和返回值都为空，访问定符为public。

当然与字段表集合相应的，如果父类方法在子类中被重写（Override），方法表集合中就不会出现来父类的方法信息。如果未被覆盖就有可能出现由编译器自动添加的方法，最常见的便是类构造器（<clnit>()）以及实例构造器<init>()方法。

结合我们的Class文件可以看到，该Class文件也是具有<init>()方法的。

属性表集合（Attributes）

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

属性表（attribute_info）前边实际上已经提到了数次，Class文件、字段表、方法表都可以携带自己属性表集合，以描述某些场景专有信息。

与其他数据项目的要求严格的顺序、长度和内容不同，属性表集合的限制稍微宽松一点，不要求各个性的严格顺序，**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信息**，Java虚拟机在运行时会**忽略掉**它所不熟悉的信息。

由于Jave当前支持的属性种类很多已经达到了29项，因而此处只列出常见的几种重要属性：

属性名称	使用位置	含义
Code 节码指令	方法表	Java代码编译成的
ConstantValue 字定义的常量值	字段表	final关
Deprecated 声明为deprecated的方法和字段	类、方法表、字段表	
Exceptions 常	方法表	方法抛出的
InnerClasses	类文件	内部类列表
LineNumberTable a源码的行号与字节码指令的对应关系	Code属性	Ja
LocalVariableTable 法的局部变量描述	Code属性	
SourceFile	类文件	源文件名称
Synthetic 识方法或字段为编译器自动生成的	类、方法表、字段表	

Code属性

Java方法里的代码被编译处理后，变为字节码指令存储在方法表的Code属性里，但并不是所有的方表里都有Code属性，例如接口或抽象类中的方法就可能没有该属性。

Code属性如下表所示：

类型	名称	含义
u2 称索引	attribute_name_index	属性
u4	attribute_length	属性长度
u2 大值	max_stack	操作数栈深度的

u2	max_locals	局部变量表所需存储空间
u4	code_length	字节码长度
u1	code[code_length]	存储字码指令的一系列字节流
u2	exception_table_length	异表长度
exception_info	exception_table	常表的值
u2	attributes_count	属性数量
attribute_info	attributes[attributes_count]	性的值

结合我们的Class文件可以看到其Code属性如下：

从图中可以看出，Code属性本身也是一个复合属性，其中包含了其他属性，比如包含了LineNumberTable和LocalVariableTable。

ConstantValue属性

只有当一个字段被声明为static final时，并且该字段是基本数据类型或String类型时，编译器才会在段的属性表集合中增加一个名为ConstantValue的属性，所以ConstantValue属性只会出现在字段表，其数据结构为：

类型	名称	含义
u2	attribute_name_index	属性称索引
u2	attribute_length	属性长度
u2	constantvalue_index	常量常量的索引

总结

Class文件是Java虚拟机执行引擎的数据入口，也是Java技术体系的基础支柱之一，因而学习Class文的结构很有意义。本文主要讲解了Class文件结构中的各个组成部分，以及每个部分的定义、数据结和使用方法。并结合一个例子（文中有代码，引用处附带有链接），讲解了Class文件是如何被存储访问的。

参考

1. Java Class文件结构解析
2. 类文件结构
3. 《深入理解JVM虚拟机》
4. 文中使用的class文件　BMWCar.class