**The Java® Virtual Machine Specification**

Java SE 8 Edition

Java虚拟机规范文档

Java8

Tim Lindholm

Frank Yellin

Gilad Bracha

Alex Buckley

整理人：Arjen10

**Version**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本号** | **修改人** | **备注** |
| 2022-06-05 | 1.0 | Arjen10 | 最初版本 |
|  |  |  |  |

# Java虚拟机结构

# Java虚拟机的编译

# .class文件的格式

JavaSE中java.io.DataInput和java.io.DataOutput这两个接口提供支持，例如可以用实现类[java.io.DataInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html)和[java.io.DataOutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/DataOutput.html)。

类文件为什么有些时候叫做字节码文件？因为在从源码编译后里面都是一个一个的字节，每项数据的不同就用不同的字节。

u4 、u2就是无符号的四个字节、两个字节。

Javac把Java源文件编译成字节码文件，JVM最终执行的就是字节码文件，JVM再把字节码翻译成机器码执行。

学习字节码建议点击下载[查看字节码文件软件](https://github.com/ClassViewer/ClassViewer/tree/v3.9)。

## .class文件的结构

一个类文件由如下结构组成：

1. ClassFile {
2. u4             magic;
3. u2             minor\_version;
4. u2             major\_version;
5. u2             constant\_pool\_count;
6. cp\_info        constant\_pool[constant\_pool\_count-1];
7. u2             access\_flags;
8. u2             this\_class;
9. u2             super\_class;
10. u2             interfaces\_count;
11. u2             interfaces[interfaces\_count];
12. u2             fields\_count;
13. field\_info     fields[fields\_count];
14. u2             methods\_count;
15. method\_info    methods[methods\_count];
16. u2             attributes\_count;
17. attribute\_info attributes[attributes\_count];
18. }

### Magic（魔术值）

在类的装载阶段用于验证这个文件是不是Java的类文件，这通常是第一关，通常值为0xCA FE BA BE（PS:咖啡宝贝，Java的logo就是一杯咖啡），等于十进制的 3405691582 。

### minor\_version（次要版本）, major\_version（主要版本）

主要是查看当前的JVM能否加载这个类文件，如果是高版本的JDK编译的类文件，那么低版本的JVM就无法加载。

且会抛出**java.lang.UnsupportedClassVersionError**。

### constant\_pool\_count（常量池计数）

这个类文件中常量池的总数。

### constant\_pool[constant\_pool\_count - 1]（常量池）

常量池，这个可以看做索引从1开始的数组，这就解释了，为什么他要比常量池计数少一个。

### access\_flags（访问标识符）

标记这个字节码文件是什么成分，是一个接口？还是一个枚举？是final还是public修饰？

**标识符对应的16进制表**

|  |  |  |
| --- | --- | --- |
| 标记名称 | 16进制值 | 说明 |
| ACC\_PUBLIC | 0x0001 | 关键字public的特性 |
| ACC\_FINAL | 0x0010 | 关键字final的特性 |
| ACC\_SUPER | 0x0020 | 当被invokespecial指令调用时，  会对父类方法作特殊处理。 |
| ACC\_INTERFACE | 0x0020 | 这是一个接口 |
| ACC\_ABSTRACT | 0x0040 | 这是一个抽象的 |
| ACC\_SYNTHETIC | 0x1000 | synthetic，由编译器产生，源码中不存在 |
| ACC\_ANNOTATION | 0x2000 | 这是一个注解 |
| ACC\_ENUM | 0x4000 | 这是一个枚举 |

### this\_class（当前字节码）

当前字节码文件，必须是constant\_pool中的有效索引，且该索引处的结构应该是一个Constant\_Class\_info结构，用来表示这个字节码文件是个类还是个接口。

### super\_class（父类）

对于任何一个类，它的super\_class只有两种情况：

1. 为0的时候

此时这个this\_class必定是java.lang.Object，这是唯一没有父类的类或者接口。

1. 不为0的时候

此时这个super\_class指向的constant\_pool中的有效索引，且该索引处的结构应该是一个Constant\_Class\_info结构，且这个super\_class的Class\_File结构中的access\_flags （访问标识符）项不能有ACC\_FINAL这个标识符，因为final修饰的类是太监类，不允许有子类。

接口interface的super\_class怎么表示呢？当然接口的super\_class必定是constant\_pool里面的有效索引，且对应的结构也必定是Object的Constant\_Class\_info结构。

**问题来了，接口的父类就是Object？**

当然不是，从学JavaSE开始接口就是没得父类的，那为啥子这里的super\_class还要指定Object？这个链接讲述的很好：[点击我](https://stackoverflow.com/questions/68874966/interfaces-implicitly-declaring-public-methods-of-object-class)。

1. **public** interface A {
2. }
3. A a = **new** A(){};
4. a.hashCode();

如上面伪代码所示，这个对象a为什么可以调用Object里面的hashCode方法？本质上是接口帮我们做了一层抽象，只是悄悄的给我们提供了这些方法，使得Object类里面的方法可以传递给我们， Java在设计的时候就是要确保所有的对象都能调用Object里面的方法。如果不这样设计，那我每个接口都要去添加那些实现重复的方法，例如hashCode、equals等等方法，这样是不利于开发人员的。所以接口的super\_class才会指定到Object帮我们完成这件事。

### interfaces\_count（接口总数）

给出了该类实现的接口数量或该接口的父接口数量。

### interfaces[]（接口表）

用于描述类实现的接口或接口的父接口。interfaces数组中的每个值都必须是在constant\_pool 表中存在有效索引。interface[i]（其中0 ≤ *i* < interfaces\_count）在constant\_pool中的每个条目必须是一个接口或者是接口的父接口Constant\_Class\_info结构，而且排列顺序按照源码里面的实现或者继承的顺序从左到右依次排列。

### fields\_count（字段总数）

fields\_count给出了在fields表中的 field\_info结构的总数，field\_info结构表示了在类或者接口中所有的字段，其中包括类变量和实例变量。

类变量（静态变量）独立于方法之外的变量，用static修饰，特点如下：

1. 随着类加载而加载。
2. 独立于对象之外，可以被所有对象共享。
3. 可以被类名加变量名调用，前提得注意类变量的权限修饰符。

实例变量独立于方法之外的变量，不过没有 static 修饰，特点如下：

1. 必须要被实例化才能调用。

**类变量和实例变量的区别**

|  |  |  |
| --- | --- | --- |
| 区别 | 类变量 | 实例变量 |
| 存储位置 | 随着类的载放在方法区中 | JVM堆内存中 |
| 生命周期 | 在类的加载时创建，程序退出时销毁 | 对象的创建而创建，对象的回收时销毁 |
| 变量的值 | 一个类无论被实例化了多少次，类只拥有类变量的一个拷贝 | 不同的对象可以有不同的值 |

### fields[]（字段表）

用来描述类或者接口的字段信息。fields表中的每个值都必须是field\_info结构，field\_info来描述这个类或接口中的字段结构。这个fields表仅仅只用来描述这个类或接口中的字段，不包含继承父类或父接口的字段。

### methods\_count（方法总数）

methods\_count给出了在methods[]中有多少个method\_info结构。

### methods[]（方法表）

用来描述类或者接口的方法信息。methods表中的每个值都必须是method\_info 结构，method\_info 来描述这个类或接口中的方法结构。如果method\_info中的access\_flags条目两个没有设置ACC\_NATIVE 和ACC\_ABSTRACT标记，那么还需要此方法的Java虚拟机指令。

method\_info描述了所有类或者接口类型中的方法，其中实例方法、类方法（静态方法）、实例初始化方法（<init>）以及所有类和接口的初始化方法。在方法表中不会包含来自父类或者父接口的方法method\_info结构。

### attributes\_count

### attributes[]（属性表）

## 名称的内部表达形式

### 类和接口名称二进制表述

出现在字节码文件中的类和接口名称始终以二进制的全限定名形式表示，结构始终为Constant\_UTF8\_INFO结构。而Constant\_NameAndType\_INFO又会指向这个二进制表述，从而得到这个字段或者方法的名称。

历史原因导致，以 （. .）作为分割符号的ASCLL符号在字节码文件中会被替换成 ASCLL的正斜杠 /。

例如：Thread类全限定名的正常表述为java.lang.Thread，但是在字节码文件中使用的是Constant\_UTF8\_INFO结构二进制编码表示的java/lang/Thread。

### 非限定名

在Java中非限定名称就是不包含包名的名称，字段名称、方法名称、局部变量名称和形参名称都是非限定名。非限定名称必须至少包含一个Unicode码点（说人话就是必须要有值），并且不得包含任何英文句号、分号、左方括号或者正斜杠的ASCLL字符。

除了特殊方法名称<init>和<clinit>以外，其他方法名称不得包含左尖括号（<）和右尖括号（>）的ASCLL字符。

注意：字段名称或接口方法名称可能是<init>或<clinit>，但是没有方法指令去调用<clinit>，只有invokespecial指令可以调用<init>。

## 描述符

描述符就是表示字段或者是方法类型的字符串。描述符采用UTF-8缩略编码的格式进行表示。

### 语法符号

### 字段描述符

字段描述符用于表示类、实例或者局部变量的类型。

字段类型由三种类型构成BaseType（基本数据类型）、ObjctType（引用数据类型）、ArrayType（数组类型）。在字节码中用于标注字段类型的代号都是ASCLL字符。

**字段描述符的解释**

|  |  |  |  |
| --- | --- | --- | --- |
| 字段类型代号 | 类型 | 说明 | 所属类型 |
| B | byte | 单字节 | **BaseType** |
| S | short | 双字节 |
| I | int | 整数，四个字节 |
| J | long | 长整数，八个字节 |
| F | float | 单精度浮点值 |
| D | double | 双精度浮点值 |
| C | char | [基本多文种平面](https://zh.wikipedia.org/wiki/Unicode%E5%AD%97%E7%AC%A6%E5%B9%B3%E9%9D%A2%E6%98%A0%E5%B0%84#%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A7%8D%E5%B9%B3%E9%9D%A2)Unicode码点，采用UTF-16编码 |
| Z | boolean | true或者false |
| L **ClassName** | reference | **ClassName**的实例化对象 | **ObjectType** |
| [ | reference | 一维数组 | **ArrayType** |

例如：

1. 字段类型为int的实例字段描述符很简单就是一个 I。
2. 字段类型为Object的实例字段描述符就是Ljava/lang/Object，但是需要注意上方的关于**类名和接口名称的二进制表述方式**，在常量池里面是个Constant\_UTF8\_INFO结构，具体的类型需要指向这个UTF8结构。
3. 字段类型为double [ ][ ][ ]的实例字段描述符就是 [ [ [ D。

个人理解：其实只为了给这个字段加个数据类型的标识符，标识这个字段类型是啥子。

### 方法描述符

一个方法描述符包含多个或者没有参数描述符，表示方法采用的参数类型，以及返回值类型，如果返回值是void那就没有返回值类型。在字节码结构methods中可以看到具体的表述方式。

MethodDescriptor**（方法描述符）**由[ParameterDescriptor](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-ParameterDescriptor)**（参数描述符）**和ReturnDescriptor**（返回值描述符）**组成，可以有也可以没有[ParameterDescriptor](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-ParameterDescriptor)因为一个方法的形参列表可有可无，但是必须要有ReturnDescriptor。

具体的表述方式:：([ParameterDescriptor](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-ParameterDescriptor)); ReturnDescriptor

[ParameterDescriptor](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-ParameterDescriptor)必定是[FieldType](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-FieldType)**（字段描述符）**，因为方法的参数绝对有类型。

ReturnDescriptor有两个，第一个是[FieldType](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-FieldType)，第二个是[VoidDescriptor](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-VoidDescriptor)**（无返回值描述符）**用字符V表示这个方法不返回任何值。

例如：

**源 码：** Object m(int i, double d, Thread t) {...}；

**字节码：** (IDLjva/lang/Thread);Ljava/lang/Object；

注意这里的Thread和Object在字节码中均是二进制表达形式。而且只是表达了方法参数和返回值，会在Method\_INFO里面看到它们。

## The Constant Pool（常量池）

所有的Constant Pool 的数据组成都应该如下组成：

cp\_info {

u1 tag; //标识符号

u1 info[]; //信息

}

**tag：**以下表为准。

**cp\_info:**都是以一字节来标明常量类型，info[] 部分随 tag 的值变动而发生变动。

**常量类型对应的16进制表**

|  |  |  |
| --- | --- | --- |
| 常量类型 | 简单说明 | 16进制值 |
| CONSTANT\_Class | 类 | 7 |
| CONSTANT\_Fieldref | 字段 | 9 |
| CONSTANT\_Methodref | 方法 | 10 |
| CONSTANT\_InterfaceMethodref | 接口方法 | 11 |
| CONSTANT\_String | 字符串 | 8 |
| CONSTANT\_Integer | 整形 | 3 |
| CONSTANT\_Float | 单精度浮点 | 4 |
| CONSTANT\_Long | 长整型 | 5 |
| CONSTANT\_Double | 双精度浮点 | 6 |
| CONSTANT\_NameAndType | 字段和方法类型 | 12 |
| CONSTANT\_Utf8 | utf8编码结构 | 1 |
| CONSTANT\_MethodHandle |  | 15 |
| CONSTANT\_MethodType |  | 16 |
| CONSTANT\_InvokeDynamic |  | 18 |

### CONSTANT\_String\_info 结构

CONSTAMT\_Strring\_info 用于表示常量对象String，具体结构如下：

CONSTANT\_String\_info {

u1 tag;

u2 string\_index;

}

**tag:**此时必定是8。

**string\_index:**这个String的值在常量池的哪里，就是constant\_pool [string\_index]。此时对应的取值结构必须是CONSTANT\_Utf8\_info。

**个人理解：**String\_info不能持有字符串的内容，而持有了值在常量池的具体位置，指向了常量池中对应的CONSTANT\_Utf8\_info。

### CONSTANT\_Utf8\_info 结构

CONSTAMT\_String\_info 用于表示常量对象String具体的值，JDK在编译源码时采用的是UTF-8缩略编码并非正常编码，具体结构如下：

CONSTANT\_Utf8\_info {

u1 tag;

u2 length;

u1 bytes [length];

}

**tag:** 此时必定是1。

**length：**表示后面占用多少个字节来表示这个字符串。

**bytes [length]：**是一个长度为**length**的16进制的数组。采用的是MUTF-8编码（UTF-8的编码再编码）

**版本差异：**在JDK1.8中String是由一个char数组，在JDK11中更新为了一个byte数组。

### CONSTANT\_Integer\_info和CONSTANT\_Float\_info结构

CONSTANT\_Integer\_info和CONSTANT\_Float\_info占用四个字节的常量，具体结构如下：

CONSTANT\_Integer\_info {

u1 tag;

u4 bytes;

}

CONSTANT\_Float\_info {

u1 tag;

u4 bytes;

}

**tag:** CONSTANT\_Integer\_info必定为3，CONSTANT\_Float\_info必定为4。

**bytes：**占用四个字节。

**CONSTANT\_Integer\_info：**高端字节开始存储，采用的是big-endian存储方式。

**例如：**int a = 2147281647，字节码第一位tag是03 加上 a 对应的16进制（7FFC EAEF）字节码文件中为descript。如果little-endian去读取的话就是EFEA FC7F = 4,025,154,687。

**CONSTANT\_Float\_info：**IEEE754 标准存储

### CONSTANT\_Long\_info和CONSTANT\_Double\_info 结构

Constant\_Long\_INFO和CONSTANT\_Double\_INFO占用8个字节。



**tag ：**Long\_INFO必定是 5。Double\_INFO必定是7。

**Long\_INFO：**存储时候会把64个字节拆分成高32位和低32位，以big-endian方式存储。

例如Long的最大取值16进制表达方式7F FF FF FF / FF FF FF FF，就会分别占用4个字节存放高32位和低32位。

**Double\_INFO：**

### CONSTANT\_NameAndType\_info结构

NameAndType用于表示一个字段或者是方法，不指明它属于哪个类或者是接口。



**tag：**此时必定为12；

**name\_index：**字段名称或者是方法名称对应的UTF8\_INFO结构常量池索引 。

**descriptor\_index：**字段描述符或者是方法描述符的UTF8\_INFO结构常量池索引。

# 加载、连接和初始化字节码

# Java虚拟机指令集

# 附录

## 字节码里面的MUTF-8编码

## IEEE754标准存储