**1.总体格式**

|  |  |  |
| --- | --- | --- |
| **Class File format** | | |
| **type** | **descriptor** | **remark** |
| u4 | magic | 0xCAFEBABE |
| u2 | minor\_version |  |
| u2 | major\_version |  |
| u2 | constant\_pool\_count |  |
| cp\_info | constant\_pool[cosntant\_pool\_count – 1] | index 0 is invalid |
| 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] |  |

**2.**     **格式详解**

**2.1  magic**

magic被称为“魔数”，用来标识.class文件的开头。所有合法的.class字节码都应该是该数开头，占4个字节。

**2.2  major\_version.minor\_version**

major\_version.minor\_version合在一起形成当前.class文件的版本号，该版本号一般由编译器产生，并且由sun定义。如59.0。它们一起占4个字节。

**2.3  constant\_pool**

在Java字节码中，有一个常量池，**用来存放不同类型的常量**。由于Java设计的目的之一就是字节码需要经网络传输的，因而字节码需要比较紧凑，以减少网络传输的流量和时间。常量池的存在则可以让一些相同类型的值通过索引的方式从常量池中找到，而不是在不同地方有不同拷贝，缩减了字节码的大小。

每个常量池中的项是通过**cp\_info**的类型来表示的，它的格式如下：

|  |  |  |
| --- | --- | --- |
| **cp\_info format** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag |  |
| u1 | info[] |  |

这里tag用来表示当前常量池不同类型的项。info中存放常量池项中存放的数据。

tag中表示的数据类型：

CONSTANT\_Class\_info                                  （7）、

CONSTANT\_Integer\_info                              （3）、

CONSTANT\_Long\_info                                   （5）、

CONSTANT\_Float\_info                                  （4）、

CONSTANT\_Double\_info                              （6）、

CONSTANT\_String\_info                                 （8）、

CONSTANT\_Fieldref\_info                              （9）、

CONSTANT\_Methodref\_info                       （10）、

CONSTANT\_InterfaceMethodref\_info      （11）、

CONSTANT\_NameAndType\_info                （12）、

CONSTANT\_Utf8\_info                                   （1）、

注：在Java字节码中，所有boolean、byte、char、short类型都是用int类型存放，因而在常量池中没有和它们对应的项。

**2.3.1**    **CONSTANT\_Class\_info**

用于记录类或接口名（used to represent a class or an interface）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Class\_info format** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Class (7) |
| u2 | name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。表示类或接口名。 |

注：在Java字节码中，类和接口名不同于源码中的名字，详见**附件A.**

**2.3.2**    **CONSTANT\_Integer\_info**

用于记录int类型的常量值（represent 4-byte numeric (int) constants:）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Integer\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Integer (3) |
| u4 | bytes | 整型常量值 |

**2.3.3**    **CONSTANT\_Long\_info**

用于记录long类型的常量值（represent 8-byte numeric (long) constants:）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Long\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Long (5) |
| u4 | high\_bytes | 长整型的高四位值 |
| u4 | low\_bytes | 长整型的低四位值 |

**2.3.4**    **CONSTANT\_Float\_info**

用于记录float类型的常量值（represent 4-byte numeric (float) constants:）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Float\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Float(4) |
| u4 | bytes | 单精度浮点型常量值 |

几个特殊值：0x7f800000 => Float.POSITIVE\_INFINITY、0xff800000 => Float.NEGATIVE\_INFINITY、

                0x7f800001 to 0x7fffffff => Float.NaN、0xff800001 to 0xffffffff => Float.NaN

**2.3.5**    **CONSTANT\_Double\_info**

用于记录double类型的常量值（represent 8-byte numeric (double) constants:）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Double\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Double(6) |
| u4 | high\_bytes | 双精度浮点的高四位值 |
| u4 | low\_bytes | 双精度浮点的低四位值 |

几个特殊值：0x7ff0000000000000L => Double.POSITIVE\_INFINITY、

0xfff0000000000000L => Double.NEGATIVE\_INFINITY

0x7ff0000000000001L to 0x7fffffffffffffffL => Double.NaN 、

0xfff0000000000001L to 0xffffffffffffffffL => Double.NaN

**2.3.6**    **CONSTANT\_String\_info**

用于记录常量字符串的值（represent constant objects of the type String:）

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_String\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_String(8) |
| u2 | string\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。表示String类型值。 |

**2.3.7**    **CONSTANT\_Fieldref\_info**

用于记录字段信息（包括类或接口中定义的字段以及代码中使用到的字段）。

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Fieldref\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Fieldref(9) |
| u2 | class\_index | constant\_pool中的索引，CONSTANT\_Class\_info类型。记录定义该字段的类或接口。 |
| u2 | name\_and\_type\_index | constant\_pool中的索引，CONSTANT\_NameAndType\_info类型。指定类或接口中的字段名（name）和字段描述符（descriptor）。 |

**2.3.8**    **CONSTANT\_Methodref\_info**

用于记录方法信息（包括类中定义的方法以及代码中使用到的方法）。

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Methodref\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Methodref(10) |
| u2 | class\_index | constant\_pool中的索引，CONSTANT\_Class\_info类型。记录定义该方法的类。 |
| u2 | name\_and\_type\_index | constant\_pool中的索引，CONSTANT\_NameAndType\_info类型。指定类中扽方法名（name）和方法描述符（descriptor）。 |

**2.3.9**    **CONSTANT\_InterfaceMethodref\_info**

用于记录接口中的方法信息（包括接口中定义的方法以及代码中使用到的方法）。

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_InterfaceMethodref\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_InterfaceMethodref(11) |
| u2 | class\_index | constant\_pool中的索引，CONSTANT\_Class\_info类型。记录定义该方法的接口。 |
| u2 | name\_and\_type\_index | constant\_pool中的索引，CONSTANT\_NameAndType\_info类型。指定接口中的方法名（name）和方法描述符（descriptor）。 |

**2.3.10**   **CONSTANT\_NameAndType\_info**

记录方法或字段的名称(name)和描述符(descriptor)（represent a field or method, without indicating which class or interface type it belongs to:）。

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_NameAndType\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_NameAndType (12) |
| u2 | name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定字段或方法的名称。 |
| u2 | descriptor\_index | constant\_pool中的索引，CONSTANT\_utf8\_info类型。指定字段或方法的描述符（**见附录C**） |

**2.3.11**   **CONSTANT\_Utf8\_info**

记录字符串的值（represent constant string values. String content is encoded in *modified UTF-8*.）

modifie

d UTF-8 refer to :

[http://download.ora](http://download.oracle.com/javase/1.4.2/docs/api/java/io/DataInputStream.html)

[cle.com/javase/1.4.2/docs/api/java/io/DataInputStream.html](http://download.oracle.com/javase/1.4.2/docs/api/java/io/DataInputStream.html)

|  |  |  |
| --- | --- | --- |
| **CONSTANT\_Utf8\_info** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | CONSTANT\_Utf8 (1) |
| u2 | length | bytes所代表  的字符串的长度 |
| u1 | bytes[length] | 字符串的byte数据，可以通过DataInputStream中的readUtf()方法（实例方法或静态方法读取该二进制的字符串的值。） |

**2.4  access\_flags**

指定类或接口的访问权限。

|  |  |  |
| --- | --- | --- |
| **类或接口的访问权限** | | |
| **Flag Name** | **Value** | **Remarks** |
| ACC\_PUBLIC | 0x0001 | pubilc，包外可访问。 |
| ACC\_FINAL | 0x0010 | final，不能有子类。 |
| ACC\_SUPER | 0x0020 | 用于兼容早期编译器，新编译器都设置该标记，以在使用 *invokespecial*指令时对子类方法做特定处理。 |
| ACC\_INTERFACE | 0x0200 | 接口，同时需要设置：ACC\_ABSTRACT。不可同时设置：ACC\_FINAL、ACC\_SUPER、ACC\_ENUM |
| ACC\_ABSTRACT | 0x0400 | 抽象类，无法实例化。不可和ACC\_FINAL同时设置。 |
| ACC\_SYNTHETIC | 0x1000 | synthetic，由编译器产生，不存在于源代码中。 |
| ACC\_ANNOTATION | 0x2000 | 注解类型（annotation），需同时设置：ACC\_INTERFACE、ACC\_ABSTRACT |
| ACC\_ENUM | 0x4000 | 枚举类型 |

**2.5  this\_class**

this\_class是指向constant pool的索引值，该值必须是CONSTANT\_Class\_info类型，指定当前字节码定义的类或接口。

**2.6  super\_class**

super\_class是指向constant pool的索引值，该值必须是CONSTANT\_Class\_info类型，指定当前字节码定义的类或接口的直接父类。只有Object类才没有直接父类，此时该索引值为0。并且父类不能是final类型。接口的父类都是Object类。

**2.7  interfaces**

interfaces数组记录所有当前类或接口直接实现的接口。interfaces数组中的每项值都是一个指向constant pool的索引值，这些值必须是CONSTANT\_Class\_info类型。数组中接口的顺序和源代码中接口定义的顺序相同。

**2.8  fields**

fields数组记录了类或接口中的所有字段，包括实例字段和静态字段，但不包含父类或父接口中定义的字段。fields数组中每项都是field\_info类型值，它描述了字段的详细信息，如名称、描述符、字段中的attribute等。

|  |  |  |
| --- | --- | --- |
| **field\_info** | | |
| **type** | **descriptor** | **remark** |
| u2 | access\_flags | 记录字段的访问权限。见2.8.1 |
| u2 | name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定字段的名称。 |
| u2 | descriptor\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型，指定字段的描述符（见附录C）。 |
| u2 | attributes\_count | attributes包含的项目数。 |
| attribute\_info | attributes[attributes\_count] | 字段中包含的Attribute集合。见2.8.2-2.8.7 |

**注：**fields中的项目和CONSTANT\_Fieldref\_info中的项目部分信息是相同的，他们主要的区别是CONSTANT\_Fieldref\_info中的项目不仅包含了类或接口中定义的字段，还包括在字节码中使用到的字段信息。不过这里很奇怪，为什么field\_info结构中不把name\_index和descriptor\_index合并成fieldref\_index，这样的class文件不是更加紧凑吗？？不知道这是sun因为某些原因故意这样设计还是这是他们的失误？？

**2.8.1**    **字段访问权限**

|  |  |  |
| --- | --- | --- |
| **字段的访问权限** | | |
| **Flag Name** | **Value** | **Remarks** |
| ACC\_PUBLIC | 0x0001 | pubilc，包外可访问。 |
| ACC\_PRIVATE | 0x0002 | private，只可在类内访问。 |
| ACC\_PROTECTED | 0x0004 | protected，类内和子类中可访问。 |
| ACC\_STATIC | 0x0008 | static，静态。 |
| ACC\_FINAL | 0x0010 | final，常量。 |
| ACC\_VOILATIE | 0x0040 | volatile，直接读写内存，不可被缓存。不可和ACC\_FINAL一起使用。 |
| ACC\_TRANSIENT | 0x0080 | transient，在序列化中被忽略的字段。 |
| ACC\_SYNTHETIC | 0x1000 | synthetic，由编译器产生，不存在于源代码中。 |
| ACC\_ENUM | 0x4000 | enum，枚举类型字段 |

**注：**接口中的字段必须同时设置：ACC\_PUBLIC、ACC\_STATIC、ACC\_FINAL

**2.8.2**    **ConstantValue Attribute （JVM识别）**

|  |  |  |
| --- | --- | --- |
| **ConstantValue Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“ConstantValue”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度（固定值：2） |
| u2 | constant\_value\_index | constant\_pool中的索引，  CONSTANT\_Integer\_info（int，boolean，char、short、byte）、  CONSTANT\_Float\_info（float）、  Constant\_Double\_info（double）、  CONSTANT\_Long\_info（long）  CONSTANT\_String\_info（String）类型 |

每个常量字段（final，静态常量或实例常量）都包含有且仅有一个ConstantValue Attribute。ConstantValue Attribute结构用于存储一个字段的常量值。

对一个**静态常量字段**，该常量值会在类或接口被初始化之前，由JVM负责赋给他们，即它在任何静态字段之前被赋值。

对一个**非静态常量字段**，该值会被虚拟机忽略，它的赋值由生成的实例初始化函数（<init>）实现。如类：

**class** A {

**public** **static** **final** **int** *fa* = 10;

**public** **final** **int** fa2 = 30;

**private** **static** **int** *sa* = 20;

**static** {

*sa* = 30;

    }

}

生成的字节码如下：

// Compiled from Test.java (version 1.6 : 50.0, super bit)

class org.levin.insidejvm.miscs.staticinit.A {

***public static final int fa = 10;***

**public final int fa2 = 30;**

 private static int sa;

**static {};**

     0 bipush 20

     2 putstatic org.levin.insidejvm.miscs.staticinit.A.sa : int [16]

     5 bipush 30

     7 putstatic org.levin.insidejvm.miscs.staticinit.A.sa : int [16]

    10 return

**public A();**

     0 aload\_0 [this]

     1 invokespecial java.lang.Object() [21]

     4 aload\_0 [this]

**5 bipush 30**

**7 putfield org.levin.insidejvm.miscs.staticinit.A.fa2 : int [23]**

10 return

**2.9**  **methods**

methods数组记录了类或接口中的所有方法，包括实例方法、静态方法、实例初始化方法和类初始化方法，但不包括父类或父接口中定义的方法。methods数组中每项都是method\_info类型值，它描述了方法的详细信息，如名称、描述符、方法中的attribute（如Code Attribute记录了方法的字节码）等。

|  |  |  |
| --- | --- | --- |
| **method\_info** | | |
| **type** | **descriptor** | **remark** |
| u2 | access\_flags | 记录方法的访问权限。见2.9.1 |
| u2 | name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定方法名称。 |
| u2 | descriptor\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型，指定方法的描述符（见附录C）。 |
| u2 | attributes\_count | attributes包含的项目数。 |
| attribute\_info | attributes[attributes\_count] | 字段中包含的Attribute集合。见2.9.2-2.9.11 |

**注**：methods数组同样有和fields数组一样的问题，包括fields中项和CONSTANT\_Methodref\_info以及CONSTANT\_InterfaceMethodref\_info中的区别。以及设计上的问题。详见field\_info中的注。

**2.9.1**    **方法访问权限**

|  |  |  |
| --- | --- | --- |
| **方法的访问权限** | | |
| **Flag Name** | **Value** | **Remarks** |
| ACC\_PUBLIC | 0x0001 | pubilc，包外可访问。 |
| ACC\_PRIVATE | 0x0002 | private，只可在类内访问。 |
| ACC\_PROTECTED | 0x0004 | protected，类内和子类中可访问。 |
| ACC\_STATIC | 0x0008 | static，静态。 |
| ACC\_FINAL | 0x0010 | final，不可被重写。 |
| ACC\_SYNCHRONIZED | 0x0020 | synchronized，同步方法。 |
| ACC\_BRIDGE | 0x0040 | bridge方法，由编译器生成。**（什么是bridge方法？）** |
| ACC\_VARARGS | 0x0080 | 包含不定参数个数的方法。 |
| ACC\_NATIVE | 0x0100 | native，非Java语言实现的方法。**（如何实现Native方法？）** |
| ACC\_ABSTRACT | 0x0400 | abstract，抽象方法。 |
| ACC\_STRICT | 0x0800 | strictfp，设置floating-point模式为FP-strict。**（什么是FP-strict模式？）** |
| ACC\_SYNTHETIC | 0x1000 | synthetic，由编译器产生，不存在于源代码中。 |

**注：**接口中的方法必须同时设置：ACC\_PUBLIC、ACC\_ABSTRACT。设置了ACC\_ABSTRACT后，不可以再设置ACC\_FINAL、ACC\_STATIC、ACC\_PRIVATE、ACC\_NATIVE、ACC\_SYNCHRONIZED 、ACC\_STRICT。

**2.9.2**    **Code Attribute （JVM识别）**

每个非abstract、非native方法的attributes集合都包含有且仅有一项Code Attribute。它包含了一个方法的栈、局部变量、字节码以及和代码相关的Attribute信息。

|  |  |  |
| --- | --- | --- |
| **Code Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“Code”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | max\_stack | 该方法操作栈的最大深度。 |
| u2 | max\_locals | 该方法调用时需要分配的局部变量的最大个数，包括该方法的参数。 |
| u4 | code\_length | 该方法字节码长度（以字节为单位） |
| u1 | code[code\_length] | 存放字节码数组**（字节码如何解析？）**。 |
| u2 | exception\_table\_length | 异常表的长度。 |
| |  |  |  | | --- | --- | --- | | exception\_table\_info | | | | 每个表项记录一段异常处理代码信息和范围。 | | | | u2 | start\_pc | 记录应用该项异常处理的起始字节码。在字节码数组中的起始索引号**[start\_pc, end\_pc)。**索引号必须是opcode（一条指令的开始位置）对应的位置。 | | u2 | end\_pc | | u2 | handler\_pc | 记录该项异常处理代码的开始地址。在字节码数组中的开始索引号。索引号必须是opcode对应的位置。 | | u2 | catch\_type | constant\_pool中的索引，CONSTANT\_Class\_info类型。指定该项能捕获的异常类（或其子类）。或0用于实现finally语法（即不管什么类型都会捕获。） | | exception\_table[exception\_table\_length] | | | | | |
| u2 | attributes\_count | attributes包含的项目数。 |
| attribute\_info | attributes[attributes\_count] | 字段中包含的Attribute集合。见2.9.2.1-2.9.2.4 |

**2.9.2.1 StackMapTable Attribute （JVM识别）**

StackMapTable Attribute在J2SE 6中引入，记录了类型检查时需要用到的信息，如字节码的偏移量、局部变量的验证类型、操作栈中的验证类型，用于类型检查过程。在Code Attribute只能包含一项StackMapTable Attribute，记录所有当前Code Attribute中的验证信息。

一项StackMapTable Attribute中包含多项stack\_map\_frame。每项stack\_map\_frame显式或隐式得记录了字节码的偏移量、局部变量验证类型和操作栈的验证类型。验证器就是通过获取局部变量类型和操作栈类型进行验证的**（具体如何验证呢？）**。

在stack\_map\_frame中，并不是直接记录了字节码的索引值，而是记录了offset\_delta的值。stack\_map\_frame中的每一项都通过**前一项的值+1+offset\_delta**计算出当前项对应的真正的字节码的位置，只有当当前stack\_map\_frame的前一项是当前方法的初始帧（initial frame of the method，**什么是方法的初始帧？**）的时候，offset\_delta的值才直接表示字节码位置。

（为什么不直接记录字节码的索引值？为了保证stack\_map\_frame被正确的排序了。为什么要加1再加offset\_delta？为了避免重复出现stack\_map\_frame项。）

|  |  |  |
| --- | --- | --- |
| **StackMapTable Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“StackMapTable”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | number\_of\_entries | stack\_map\_frame项的数目。 |
| |  |  | | --- | --- | | stack\_map\_frame（union，联合体类型） | | | 每一项显式或隐式得记录了字节码的偏移量、局部变量验证类型和操作栈的验证类型。每一项第一个字节指定当前stack\_map\_frame的类型（tag） | | | same\_frame | same\_frame {       u1 frame\_type = SAME; /\* 0-63 \*/  }  tag的值为[0-63]。offset\_delta = frame\_type。表示当前帧和前一帧有相同的局部变量，并且当前操作数栈为空。 | | same\_locals\_1\_stack  \_item\_frame | same\_locals\_1\_stack\_item\_frame {  u1 frame\_type = SAME\_LOCALS\_1\_STACK\_ITEM; /\* 64-127 \*/       verification\_type\_info stack[1]  }  tag值为[64-127]。offset\_delta = frame\_type – 64。表示当前帧和前一帧有相同的局部变量，并且操作栈内的操作数条目数为1，因而它为该操作数栈内的操作数保存了一项verification\_type\_info（详见下表）。（[128-246]的tag值保留） | | same\_locals\_1\_stack  \_item\_frame\_extended | same\_locals\_1\_stack\_item\_frame\_extended {       u1 frame\_type = SAM\_LOCALS\_1\_STACK\_ITEM\_EXTENEDED; /\* 247 \*/       u2 offset\_delta;       verification\_type\_info stack[1];  }  tag值为247。offset\_delta = offset\_delta。表示当前值帧和前一帧有相同的局部变量，并且操作栈内的操作数条目数为1，因而它为该操作数栈内的操作数保存了一项verification\_type\_info（详见下表）。和上一类型的区别是这里的offset\_delta是直接给出的。 | | chop\_frame | chop\_frame {       u1 frame\_type = CHOP; /\* 248 – 250 \*/       u2 offset\_delta;  }  tag值为[248, 250]。offset\_delta = offset\_delta。表示当前操作栈为空，而当前局部变量比前一帧的局部变量少后面的251 – frame\_type个局部变量。 | | same\_frame\_extended | same\_frame\_extended {       u1 frame\_type = SAME\_FRAME\_EXTENDED; /\* 251 \*/       u2 offset\_delta;  }  tag值为251。offset\_delta = offset\_delta。表示当前帧和前一帧有相同的局部变量，并且操作数栈为空。和samep\_frame的区别是same\_frame\_extended中的offset\_delta值直接给出。 | | append\_frame | append\_frame {       u1 frame\_type = APPEND; /\* 252 – 254 \*/       u2 offset\_delta;       verification\_type\_info locals[frame\_type – 251];  }  tag值为[252-254]。offset\_delta = offset\_delta。表示操作数栈为空，而当前帧的局部变量比前一帧的局部变量多frame\_type – 251个。因而它也定义了frame\_type – 251项的verification\_type\_info类型。详见下表。 | | full\_frame | full\_frame {       u1 frame\_type = FULL\_FRAME; /\* 255 \*/       u2 offset\_delta;       u2 number\_of\_locals;       verification\_type\_info locals[number\_of\_locals];       u2 number\_of\_stack\_items;       verification\_type\_info stack[number\_of\_stack\_items];  }  tag值255。offset\_delta = offset\_delta。full\_frame则定义了所有的信息，包括offset\_delta的值，以及当前帧和前一帧不同的所有局部变量和操作数。locals[0]表示0号局部变量；stack[0]表示栈底操作数。 | | entries[number\_of\_entries] | | | | |

|  |  |
| --- | --- |
| verification\_type\_info（union，联合体类型） | |
| 记录字节码的验证类型。每一项第一个字节指定验证类型（tag）。 | |
| Top\_variable\_info | Top\_variable\_info {      u1 tag = ITEM\_Top; /\* 0 \*/  }  指定局部变量的验证类型为Top。 |
| Integer\_variable\_info | Integer\_variable\_info {      u1 tag = ITEM\_Integer; /\* 1 \*/  }  指定验证类型为int。 |
| Float\_variable\_info | Float\_variable\_info {      u1 tag = ITEM\_Float; /\* 2 \*/  }  指定验证类型为float。 |
| Long\_variable\_info | Long\_variable\_info {      u1 tag = ITEM\_Long; /\* 4 \*/  }  指定验证类型为long。 |
| Double\_variable\_info | Double\_variable\_info {      u1 tag = ITEM\_Double; /\* 3 \*/  }  指定验证类型为double。 |
| Null\_variable\_info | Null\_variable\_info {      u1 tag = ITEM\_Null; /\* 5 \*/  }  指定验证类型为null。 |
| UninitializedThis\_variable\_info | UninitializedThis\_variable\_info {      u1 tag = ITEM\_UninitializedThis; /\* 6 \*/  }  指定验证类型为uninitializedThis（什么是uninitializedThis？）。 |
| Object\_variable\_info | Object\_variable\_info {  u1 tag = ITEM\_Object; /\* 7 \*/  u2 cpool\_index; //constant\_pool索引，CONSTANT\_Class\_info类型。  }  指定验证类型为cpool\_index中指定的类型实例。 |
| Uninitialized\_variable\_info | Uninitiated\_variable\_info {  u1 tag = ITEM\_Uninitialized; /\* 8 \*/  u2 offset;  }  指定验证类型为uninitialized（这种类型是指什么？）。offset记录了用于创建实例的new指令的偏移量。（The offset item indicates the offset of the new instruction that created the object being stored in the location.这段话是什么意思？） |

**注：**对验证过程不太了解，因而StackMapTable Attribute的一些描述也没能理解。

**2.9.2.2 LineNumberTable Attribute （调试信息）**

LineNumberTable Attribute用于调试器，以获取某条指令对应的源代码中的行号。多条指令可以对应相同的行号。

|  |  |  |
| --- | --- | --- |
| **LineNumberTable Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“LineNumberTable”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | line\_number\_table\_length | 异常表的长度。 |
| |  |  |  | | --- | --- | --- | | line\_number\_table | | | | 一条指令和源代码行号的映射关系表。 | | | | u2 | start\_pc | 一条指令的开始索引（字节码数组中的索引号） | | u2 | line\_number | 源代码中的行号。 | | line\_number\_table[line\_number\_table\_length] | | | | | |

**2.9.2.3 LocalVariableTable Attribute （调试信息）**

LocalVariableTable Attribute用于调试器，以获取在方法运行时局部变量的信息。在一个Code Attribute中只包含1或0项LocalVariableTable Attribute。

|  |  |  |
| --- | --- | --- |
| **LocalVariableTable Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“LocalVariableTable”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | local\_variable\_table\_length | 局部变量表的长度。 |
| |  |  |  | | --- | --- | --- | | local\_variable\_table | | | | 每项记录了一个局部变量有值的范围和该局部变量在局部变量数组中的索引值。 | | | | u2 | start\_pc | 记录该局部变量的有值范围，字节码数组中的索引范围**[**start\_pc, start\_pc+length**)**。 | | u2 | length | | u2 | name\_index | constant\_pool中索引，CONSTANT\_Utf8\_info类型，记录该项代表的局部变量名。 | | u2 | descriptor\_index | constant\_pool中索引，CONSTANT\_Utf8\_info类型，记录该项代表的局部变量的字段描述符（见附录C）。 | | u2 | index | 记录该项代表的局部变量在方法的局部变量数组中的索引。 | | local\_variable\_table[local\_variable\_table\_length] | | | | | |

**2.9.2.4 LocalVariableTypeTable Attribute （调试信息）**

LocalVariableTypeTable Attribute用于调试器，以获取在方法运行时泛型局部变量的信息。在一个Code Attribute中只包含1或0项LocalVariableTypeTable Attribute。

**LocalVariableTable Attribute和LocalVariableTypeTable Attribute表达的信息是类似的，他们的区别是对泛型类型的局部变量**，需要用Signature的形式表达，而不能仅仅用Descriptor的形式表达，因而对泛型类型的局部变量，需要在LocalVariableTable Attribute和LocalVariableTypeTable Attribute中同时存在一项；而对非泛型类型的局部变量来说，只要在LocalVariableTable Attribute存在表项就可以了。

从这里我们也可以看出泛型是后期才被字节码所支持的痕迹。我感觉很奇怪的是Java在设计的时候，泛型应该已经开始流行了，为什么它在设计之初没有把它考虑进去，而要到后期加入，然后让这种修补的设计做的那么糟糕呢，Java的设计者如果能在设计的时候把它作为扩展考虑，然后再后期去实现，不是更好吗？

|  |  |  |
| --- | --- | --- |
| **LocalVariableTypeTable Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“LocalVariableTypeTable”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | local\_variable\_type\_table\_length | 泛型局部变量表的长度。 |
| |  |  |  | | --- | --- | --- | | local\_variable\_type\_table | | | | 每项记录了一个泛型局部变量有值的范围和该局部变量在局部变量数组中的索引值。 | | | | u2 | start\_pc | 记录该局部变量的有值范围，字节码数组中的索引范围**[**start\_pc, start\_pc+length**)**。 | | u2 | length | | u2 | name\_index | constant\_pool中索引，CONSTANT\_Utf8\_info类型，记录该项代表的局部变量名。 | | u2 | signature\_index | constant\_pool中索引，CONSTANT\_Utf8\_info类型，记录该项代表的局部变量的字段签名（见附录D）。 | | u2 | index | 记录该项代表的局部变量在方法的局部变量数组中的索引。 | | local\_variable\_type\_table[local\_variable\_table\_length] | | | | | |

**2.9.3**    **Exceptions Attribute （JVM识别）**

Exceptions Attribute记录了一个方法需要检验的异常类型。一个method\_info的attributes中只能包含一项Exceptions Attribute。即记录一个方法可以抛出的异常类型。

一个方法可以抛出的异常类型遵循三点：

1. 抛出的异常是RuntimeException类型或其子类。

2. 抛出的异常是Error类型或其子类。

3. 抛出的异常是Exceptions Attribute中记录的类型或它们的子类。

|  |  |  |
| --- | --- | --- |
| **Exceptions Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“Exceptions”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | number\_of\_exceptions | exception\_index\_table表项长度 |
| u2 | exception\_index\_table[number\_of\_exceptions] | 每项为constant\_pool中的索引，CONSTANT\_Class\_info类型。记录该方法可抛出的异常类型。 |

**2.9.4**    **RuntimeVisibleParameterAnnotations Attribute**

RuntimeVisibleParameterAnnotations Attribute记录该方法在运行时可见的修饰该方法参数的Annotation，从而Java程序可以通过反射机制获取这些Annotation中的值。一个method\_info中的attributes中只能包含一项RuntimeVisibleAnnotations Attribute。

|  |  |  |
| --- | --- | --- |
| **RuntimeVisibleAnnotations Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“RuntimeVisibleAnnotations”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | num\_parameters | 记录该方法中参数个数 |
| |  |  |  | | --- | --- | --- | | parameter\_annotations | | | | 每个参数和在它之上定义的annotations的映射表。顺序和源码定义顺序一致。 | | | | u2 | num\_annotations | 记录该参数定义的annotation的个数 | | annotation | annotations[num\_annotations] | 记录该项对应的参数中所有运行时可见的annotaion项，顺序和源码定义顺序一致。**（annotation类型见附件E）** | | parameter\_annotations[num\_parameters] | | | | | |

**注：**从该数据结构的定义中可以看到，Java对Parameter Annotation存放的信息是很少的，我们只能依赖于定义的顺序来获取这些Annotation，而不能通过参数名或者参数类型来获取相应的Annotation。事实上，

1.       由于参数名的信息只在调试时才有，如LocalVariableTable Attribute或LocalVariableTypeTable Attribute中；

2.       而一个方法中不同参数的类型极有可能是相同的；

因而从逻辑上来说，通过参数名或者参数类型返回相应的Annotation信息的方式也是不合理的。由于这个原因，在java.lang.reflect.Method类的方法中也只是给出了:  
**Annotation[][] getParameterAnnotations()**

的方法，获得所有参数中的Annotation，这里的二维数组一维代表参数，一维代表多个Annotation，它们的顺序和源码定义时顺序相同。

**2.9.5**    **RuntimeInvisibleParameterAnnotations Attribute**

RuntimeInvisibleParameterAnnotations Attribute记录该方法在运行时不可见的修饰该方法参数的Annotation。一个method\_info中的attributes中只能包含一项RuntimeInvisibleAnnotations Attribute。

RuntimeInvisibleParameterAnotations Attribute和RuntimeVisibleAnnotations Attribute的**区别**在于：

后者中的Annotation默认情况下，可以通过Java提供的反射函数获取相应的Annotation，而前者的Annotation在默认情况下是无法通过Java提供的反射函数被获取的，而需要通过特定的机制（如设置JVM的特定参数，该机制由不同的JVM实现来决定）才能通过Java提供的反射函数获取内部的Annotation。

然而这样就又有一个问题了，**RuntimeInvisibleParameterAnotations Attribute是如何被填入值的呢？**通过什么机制让源码中方法参数的Annotation是默认不可见的呢？我感觉这个也可能也是由不同编译器提供不同的机制来实现的，不知道sun提供的编译器有没有什么机制支持它了？？

|  |  |  |
| --- | --- | --- |
| **RuntimeInvisibleAnnotations Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“RuntimeInvisibleAnnotations”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | num\_parameters | 记录该方法中参数个数 |
| |  |  |  | | --- | --- | --- | | parameter\_annotations | | | | 每个参数和在它之上定义的annotations的映射表。顺序和源码定义顺序一致。 | | | | u2 | num\_annotations | 记录该参数定义的annotation的个数 | | annotation | annotations[num\_annotations] | 记录该项对应的参数中所有运行时不可见的annotaion项，顺序和源码定义顺序一致。**（annotation类型见附件E）** | | parameter\_annotations[num\_parameters] | | | | | |

**2.9.6**    **AnnotationDefault Attribute**

AnnotationDefault Attribute用于Annotation类型方法中，以记录该方法所代表的Annotation类型的默认值。每个Annotation类型的method\_info中的attributes中只能包含一个AnnotationDefault Attribute项。如：

@Retention(RetentionPolicy.*RUNTIME*)

@Target({ElementType.*METHOD*, ElementType.*PARAMETER*})

**public** **@interface** Test {

**public** **int** id() **default** -1;

**public** String description() **default** "no description";

}

该Annotation类产生的class二进制文件中的id方法和description方法的attributes数组中都会包含一项AnnotationDefault Attribute，它们的默认值分别为-1（CONSTANT\_Integer\_info类型）和”no description”（CONSTANT\_String\_info类型）。

|  |  |  |
| --- | --- | --- |
| **AnnotationDefault Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“AnnotationDefault”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| element\_value | default\_value | 记录该方法表示的Annotation类型的默认值。（**element\_value结构详见附录E**） |

**2.9.7**    **Synthetic Attribute**

**参见2.11.1**

**2.9.8**    **Signature Attribute**

**参见2.11.2**

**2.9.9**    **Deprecated Attribute**

**参见2.11.3**

**2.9.10**   **RuntimeVisibleAnnotations Attribute**

**参见2.11.4**

**2.9.11**   **RuntimeInvisibleAnnotations Attribute**

**参见2.11.5**

**2.10 attributes**

attributes数组记录了和类或接口相关的所有Attribute项（和字段相关的Attribute在field\_info的attributes中，和方法相关的Attribute在method\_info的attrubutes中，和字节码相关的Attribute在Code Attribute的attributes中）。attributes数组中的每项都是attribute\_info类型，它描述了Attribute的名称、详细信息等。该attributes数组描述了ClassFile的一些额外信息。JVM必须忽略它不能识别的Attribute，而且那些JVM不能识别的的Attribute也不能影响class文件的语义。

当前定义的Attribute有：Code Attribute、Constant Value Attibute、Deprecated Attribute、Enclosing Method Attribute、Exceptions Attribute、Inner Classes Attribute、Line Number Table Attribute、Local Variable Table Attribute、Local Variable Type Table Attribute、Runtime Visible Annotations Attribute、Runtime Invisible Annotation Attribute、Runtime Visible Parameter Annotation Attribute、Runtime Invisible Parameter Annotation Attribute、Signature Attribute、Source Debug Extension Attribute、Source File Attribute、Stack Map Table Attribute、Synthetic Attribute、Annotation Default Attribute等。它们有些只存在于field\_info中，有些只存在method\_info中，有些只存在ClassFile中，有些只存在于Code Attribute中，还有些可以同时存在于field\_info、method\_info、classfile中。

Attribute结构只存在与ClassFile、method\_info、field\_info、Code Attribute结构中。

|  |  |  |
| --- | --- | --- |
| **attribute\_info（Attribute的基本数据结构）** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u1 | info[attribute\_length] | 记录Attribute的内容字节数据。 |

在用户自定义的编译器或者Java虚拟机中可以扩展ClassFile中的Attribute表（即自定义新的Attribute）。但是对自定义的Attribute必须遵循一些规则：

1.       用户自定义的新的Attribute（非sun定义的Attribute）命名必须是遵循Java命名规则，即加入公司的包名信息，如：“com.levin.new\_attribute”

2.       新增的Attribute只可以作为辅助的信息，如增加和自定义调试器相关的调试信息，但是它们不可以改变ClassFile的语义。什么叫改变ClassFile的语义呢？我现在的理解，比如在自定义的Java虚拟机中，为某个方法新增一个Attribute，用以标记该方法在运行是不可以被调用。不知道这个例子合适不合适。

3.       对于自定义的Java虚拟机，禁止因为某些它不识别的Attribute存在而抛出异常或者直接报错。但是这一层限制可以加载自定义编译器中。Java虚拟机必须忽略它不识别的Attribute。

**以下是定义在ClassFile中的Attribute。**

**2.10.1**    **InnerClasses Attribute**

InnerClasses记录当前类的所有内部类。**当前类需要记录的内部类的算法如下：**

**1. 当前类中定义的内部类，包括方法中定义的类。**

**2. 如果当前类本身是内部类，则还要记录当前类的外部类，直到外部类不是一个内部类。**

如：

**class** Outer {

**public** **class** Inner {

**public** **void** getMethod() {

**class** Inner3 {

**class** Inner4 {

              }

           }

       }

**public** **class** Inner2 {

**public** **class** Inner5 {

           }

       }

    }

**public** **class** Inner\_1 {

    }

}

**Outer中的InnerClasses Attribute：**

    [inner class info: #17 org/levin/insidejvm/miscs/instructions/**Outer$Inner**, outer class info: #1 org/levin/insidejvm/miscs/instructions/Outer

     inner name: #19 Inner, accessflags: 1 public],

    [inner class info: #20 org/levin/insidejvm/miscs/instructions/**Outer$Inner\_1**, outer class info: #1 org/levin/insidejvm/miscs/instructions/Outer

     inner name: #22 Inner\_1, accessflags: 1 public]

**Inner中的InnerClasses Attribute：**

    [inner class info: #1 org/levin/insidejvm/miscs/instructions/**Outer$Inner**, outer class info: #23 org/levin/insidejvm/miscs/instructions/Outer

     inner name: #25 Inner, accessflags: 1 public],

    [inner class info: #26 org/levin/insidejvm/miscs/instructions/**Outer$Inner$1Inner3**, outer class info: #0

     inner name: #28 Inner3, accessflags: 16 final],

    [inner class info: #29 org/levin/insidejvm/miscs/instructions/**Outer$Inner$Inner2**, outer class info: #1 org/levin/insidejvm/miscs/instructions/Outer$Inner

     inner name: #31 Inner2, accessflags: 1 public]

**Inner5中的InnerClasses Attribute：**

    [inner class info: #22 org/levin/insidejvm/miscs/instructions/**Outer$Inner**, outer class info: #24 org/levin/insidejvm/miscs/instructions/Outer

     inner name: #26 Inner, accessflags: 1 public],

    [inner class info: #27 org/levin/insidejvm/miscs/instructions/**Outer$Inner$Inner2**, outer class info: #22 org/levin/insidejvm/miscs/instructions/Outer$Inner

     inner name: #29 Inner2, accessflags: 1 public],

    [inner class info: #1 org/levin/insidejvm/miscs/instructions/**Outer$Inner$Inner2$Inner5**, outer class info: #27 org/levin/insidejvm/miscs/instructions/Outer$Inner$Inner2

     inner name: #30 Inner5, accessflags: 1 public]

|  |  |  |
| --- | --- | --- |
| **InnerClasses Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“InnerClasses”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | number\_of\_classes | 记录内部类的数量。 |
| |  |  |  | | --- | --- | --- | | classes | | | | 定义每项Inner Class的信息。 | | | | u2 | inner\_class\_info\_index | constant\_pool中的索引，CONSTANT\_Class\_info类型。记录内部类名。 | | u2 | outer\_class\_info\_index | 若当前项内部类不是包含它的类的成员类，则该值为0；否则该值为constant\_pool中的索引，CONSTANT\_Class\_info类型，记录外部类名。 | | u2 | inner\_name\_index | 若当前项的内部类为匿名类，则该值为0；否则该值为constant\_pool中的索引，CONSTANT\_Utf8\_info类型，记录该内部类的简单名（没有前缀）。 | | u2 | inner\_class\_access\_flags | 定义当前项的内部类的访问属性。见下表。 | | classess[number\_of\_classes] | | | | | |

**注：**为什么需要为内部类保留那么多的信息呢？是为了在反射的时候获取必要的信息或者在反编译的时候可以更好的还原源代码的结构吗？还是有其他的作用？

|  |  |  |
| --- | --- | --- |
| **内部类的访问权限** | | |
| **Flag Name** | **Value** | **Remarks** |
| ACC\_PUBLIC | 0x0001 | pubilc，类外可访问。 |
| ACC\_PRIVATE | 0x0002 | private，类内才可访问。 |
| ACC\_PROTECTED | 0x0004 | protected，类和其子类可访问。 |
| ACC\_STATIC | 0x0008 | static，静态内部类。 |
| ACC\_FINAL | 0x0010 | final，不能有子类。 |
| ACC\_INTERFACE | 0x0200 | 接口，同时需要设置：ACC\_ABSTRACT。不可同时设置：ACC\_FINAL、ACC\_SUPER、ACC\_Enum |
| ACC\_ABSTRACT | 0x0400 | 抽象类，无法实例化。不可和ACC\_FINAL同时设置。 |
| ACC\_SYNTHETIC | 0x1000 | synthetic，由编译器产生，不存在于源代码中。 |
| ACC\_ANNOTATION | 0x2000 | 注解类型（annotation），需同时设置：ACC\_INTERFACE、ACC\_ABSTRACT |
| ACC\_ENUM | 0x4000 | 枚举类型 |

**2.10.2**    **EnclosingMethod Attribute**

当且仅当一个类是匿名类或者本地类（local class），该类才会包含一项且仅有一项EnclosingMethod Attribute。

然而什么是本地类（local class）呢？我的理解，所谓本地类就是在方法内部定义的类，如以下类的定义：

**class** A {

**public** Iterator getIterator() {

**class** LocalClass {

       }

**return** **new** Iterator() {

**public** **boolean** hasNext() { **return** **false**; }

**public** Object next() { **return** **null**; }

**public** **void** remove() { }

       };

    }

}

匿名类A$1.class中的EnclosingMethod Attribute：

 Enclosing Method: #29 #31 org/levin/insidejvm/miscs/instructions/A.getIterator()Ljava/util/Iterator;

本地类A$1LocalClass.class中的EnclosingMethod Attribute：

 Enclosing Method: #22 #24 org/levin/insidejvm/miscs/instructions/A.getIterator()Ljava/util/Iterator;

|  |  |  |
| --- | --- | --- |
| **EnclosingMethod Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“EnclosingMethod”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度(4)。 |
| u2 | class\_index | constant\_pool中的索引，CONSTANT\_Class\_info类型。记录定义当前类所在方法的宿主类。如上例中，A$1.class和A$1LocalClass.class中的class\_index都指向类A。 |
| u2 | method\_index | 若当前类没有被方法方法包含，如当前类是赋值给类成员的匿名类，则method\_index值为0，否则该method\_index的值为constant\_pool中的索引，CONSTANT\_NameAndType\_info类型。记录了class\_index指定的类中定义的包含当前类的方法名和类型信息。 |

**注：**这里同样也有一个问题，就是EnclosingMethod Attribute存在的目的问题。我现在的理解，该Attribute的存在也应该只是为了用于反射信息和反编译时可以更好的还原原来代码的结构，在虚拟机运行该程序的时候，由于所有的指令已经编译好了，虚拟机应该不需要这些信息。但是事实是这样的吗？有待考证。

**2.10.3**    **SourceFile Attribute**

SourceFile Attribute用于记录和当前字节码对应的源代码的文件（由编译器产生，该Attribute只是记录相应源代码的文件名，而不记录和路径相关的信息）。一个ClassFile中只能包含一项SourceFile Attribute。

|  |  |  |
| --- | --- | --- |
| **SourceFile Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute的名称(“SourceFile”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度(2)。 |
| u2 | sourcefile\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。记录相应的源代码文件名。 |

**2.10.4**    **SourceDebugExtension Attribute**

SourceDebugExctension Attribute是Java为调试时提供的扩展信息，主要用于自定义（扩展）的编译器和调试器。一个ClassFile中只能包含一项SourceFile Attribute。

|  |  |  |
| --- | --- | --- |
| **SourceDebugExtension Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“SourceDebugExtension”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u1 | debug\_extension[attribute\_length] | Utf-8格式的字符串记录扩展调试信息，不以0结尾。 |

**2.11 在ClassFile、method\_info、field\_info中同时存在的Attribute**

**2.11.1**    **Synthetic Attribute**

Synthetic Attribute用于指示当前类、接口、方法或字段由编译器生成，而不在源代码中存在（不包含类初始函数和实例初始函数）。相同的功能还有一种方式就是在类、接口、方法或字段的访问权限中设置ACC\_SYNTHETIC标记。

Synthetic Attribute由JDK1.1中引入，以支持内嵌类和接口（nested classes and interfaces）。但是以我现在所知，这些功能都是可以通过ACC\_SYNTHETIC标记来表达的，为什么还需要存在Synthetic Attribute呢？在什么样的情况下会生成Synthetic Attribute项呢？我还没有找到，需要继续研究。

|  |  |  |
| --- | --- | --- |
| **Synthetic Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“Synthetic”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度（0）。 |

**2.11.2**    **Signature Attribute**

|  |  |  |
| --- | --- | --- |
| **Signature Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“Signature”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度（2）。 |
| u2 | signature\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。记录当前类型的签名（类签名、字段签名、方法签名）。 |

JVM规范中没有指定什么情况下需要生成Signature Attribute。但是从Signature的目的是用于泛型类型，可以推测Signature Attribute存在于当前Signature Attribute所在类型是泛型（泛型类、泛型方法、泛型字段）的时候。它和field\_info、method\_info、this\_class一起对应于局部变量中的LocalVariableTable Attribute和LocalVariableTypeTable Attribute，他们同时都有descriptor版本和signature版本。

**2.11.3**    **Deprecated Attribute**

Deprecated Attribute指示当前类、方法、字段已经过时了，一些工具，如编译器可以根据该Attribute提示用户他们使用的类、方法、字段已经过时了，最好使用最新版本的类、方法、字段。

|  |  |  |
| --- | --- | --- |
| **Deprecated Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“Deprecated”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度（0）。 |

**2.11.4**    **RuntimeVisibleAnnotations Attribute**

RuntimeVisibleAnnotations Attribute记录了当前类、方法、字段在源代码中定义的、在运行时可见的Annotation。Java程序可以通过反射函数获取这些Annotation。一个attributes集合中只能包含一项RuntimeVisibleAnnotations Attribute，记录所有运行时可见的Annotation。

|  |  |  |
| --- | --- | --- |
| **RuntimeVisibleAnnotations Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“RuntimeVisibleAnnotations”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | num\_annotations | annotations集合长度。 |
| annotation | annotations[num\_annotations] | 记录所有运行时可见的annotation的集合。annotation类型详见**附录E**。 |

**2.11.5**    **RuntimeInvisibleParameterAnotations Attribute**

RuntimeInvisibleAnnotations Attribute记录了当前类、方法、字段在源代码中定义的、在运行时不可见的Annotation。默认情况下，这些Annotation是不可被Java提供的反射函数获取的，需要通过和实现相关的机制来获取这些Annotation。一个attributes集合中只能包含一项RuntimeInvisibleAnnotations Attribute，记录所有运行时不可见的Annotation。

|  |  |  |
| --- | --- | --- |
| **RuntimeInvisibleAnnotations Attribute** | | |
| **type** | **descriptor** | **remark** |
| u2 | attribute\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。指定Attribute名称(“RuntimeInvisibleAnnotations”)。 |
| u4 | attribute\_length | 该Attribute内容的字节长度。 |
| u2 | num\_annotations | annotations集合长度。 |
| annotation | annotations[num\_annotations] | 记录所有运行时不可见的annotation的集合。annotation类型详见**附录E**。 |

**总体格式**

|  |  |  |  |
| --- | --- | --- | --- |
| **magic(0xCAFEBABE)** | | | |
| **version(major.minor)** | | | |
| **constant pool**   |  | | --- | | CONSTANT\_Utf8\_info(1) | | CONSTANT\_Integer\_info(3) | | CONSTANT\_Float\_info(4) | | CONSTANT\_Long\_info(5) | | CONSTANT\_Double\_info(6) | | CONSTANT\_Class\_info(7) | | CONSTANT\_String\_info(8) | | CONSTANT\_Fieldref\_info(9) | | CONSTANT\_Methodref\_info(10) | | CONSTANT\_InterfaceMethodref\_info(11) | | CONSTANT\_NameAndType\_info(12) | | | | |
| **access\_flags** | **this\_class** | **super\_class** | **interfaces** |
| **fields**   |  | | --- | | access\_flags | | name | | descriptor | | **attributes**   |  | | --- | | ConstantValue Attribute | | Synthetic Attribute | | Signature Attribute | | Deprecated Attribute | | RuntimeVisibleAnnotations Attribute | | RuntimeInvisibleAnnotations Attribute | | | | | |
| **methods**   |  | | --- | | access\_flags | | name | | descriptor | | attributes   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Code Attribute   |  | | --- | | StackMapTable Attribute | | LineNumberTable Attribute | | LocalVariableTable Attribute | | LocalVariableTypeTable Attribute | | | Exceptions Attribute | | RuntimeVisibleParameterAnnotations Attribute | | RuntimeInvisibleParameterAnnotations Attribute | | AnnotationDefault Attribute | | Synthetic Attribute | | Signature Attribute | | Deprecated Attribute | | RuntimeVisibleAnnotations Attribute | | RuntimeInvisibleAnnotations Attribute | | | | | |
| **attributes**   |  | | --- | | InnerClasses Attribute | | EnclosingMethod Attribute | | SourceFile Attribute | | SourceDebugExtension Attribute | | Synthetic Attribute | | Signature Attribute | | Deprecated Attribute | | RuntimeVisibleAnnotations Attribute | | RuntimeInvisibleAnnotations Attribute | | | | |

**附件A ：Java字节码中的类和接口名**

在Java字节码中类和接口名主要表现以下几点：

1.       类和接口名都是以**全限定名**的方式存放（包名加类或接口名）。

2.       在源代码中的点分隔符**（”.”）**在字节码中以斜杠**（”/”）**代替。如：“java.lang.Object”-> “**java/lang/Object**”

3.       数组类型名做了特殊处理。如：“int[][]”-> “[[I”、“Thread[]”->“[Ljava/lang/Thread”。详见**附录B：Java字节码中的数组类型名**

**附件B ： Java字节码中的数组类型名**

在Java中，**数组被认为是类**，因而它也有对应的类名表示，而Java字节码为数组名指定了特定的格式：

1． 所有数组名都**以“[”开头**，n维数组有n个“[”。

2． 对引用类型的数组，**在“[”后加“L”**后加引用类型的全限定名。

3． 对基本类型，**在“[”后加基本类型的对应字符**。

|  |  |
| --- | --- |
| **基本类型对应字符表** | |
| **基本类型** | **对应字符** |
| byte | B |
| char | C |
| double | D |
| float | F |
| int | I |
| long | J |
| short | S |
| boolean | Z |

**附件C ： 描述符（Descriptor）**

**描述符（Descriptor）定义了字段或方法的类型（**A *descriptor*is a string representing the type of a field or method.这段描述感觉不怎么精确**）。它存放在constant pool中的CONSTANT\_Utf8\_info类型项中。**

**1.**       **字段描述符（Field Descriptor）**

字段描述符是定义了字段、局部变量、参数等类型的字符串。即附录A中的类或接口名。

**语法定义：**

**FieldDescrptor :**

**FieldType**

**BaseType** ： B、C、D、F、I、J、S、Z（参考附录B中的基本类型对应字符表）

**ObjectType** ： LfullClassName;

**ArrayType** ： [+BaseType | [+ObjectType

**FieldType** ： BaseType | ObjectType | ArrayType

如：[[D -> double[][]、[Ljava/lang/Thread; -> Thread[]、I->int、**L**java/lang/Object; -> Object

**2.**       **方法描述符（Method Descriptor）**

方法描述符是定义了方法参数、方法返回等信息的字符串。

**语法定义：**

**MethodDescriptor:**

**（ParameterDescriptor\*）ReturnDescriptor**

**ParameterDescriptor**： FieldType

**ReturnDescriptor**： FieldType | VoidDescriptor

**VoidDescriptor**： V

如：void method（int i, Object obj）-> (ILjava/lang/Object;)V

Object getValue（）-> ( )Ljava/lang/Object;

Object mymethod(int i, double d, Object o) -> (IDLjava/lang/Object;)Ljava/lang/Object;

**附件D ： 签名（Signature）**

**签名（Signature）定义了类、字段或方法的泛型类型信息（**A *signature*is a string representing the generic type of a field or method, or generic type information for a class declaration. 这段描述感觉不怎么精确**）。它也存放在constant pool中的CONSTANT\_Utf8\_info类型项中。**

**它存在于Signature Attribute中，只有包含泛型的类、字段、方法才会产生Signature Attribute。**

**签名信息并不是给JVM用的，而是用于编译、调试、反射。**

**1.**       **类签名**

**语法定义：**

*ClassSignature:*

*FormalTypeParametersopt SuperclassSignature SuperinterfaceSignature\**

*FormalTypeParameters:*

**<***FormalTypeParameter+***>**

*FormalTypeParameter:*

**Identifier***ClassBound InterfaceBound\**

*ClassBound:*

**:***FieldTypeSignatureopt*

*InterfaceBound:*

**:***FieldTypeSignature*

*SuperclassSignature:*

*ClassTypeSignature*

*SuperinterfaceSignature:*

*ClassTypeSignature*

*FieldTypeSignature:*

*ClassTypeSignature*

*ArrayTypeSignature*

*TypeVariableSignature*

*ClassTypeSignature:*

**L***PackageSpecifier\* SimpleClassTypeSignature*

*ClassTypeSignatureSuffix\****;**

*PackageSpecifier:*

**Identifier /***PackageSpecifier\**

*SimpleClassTypeSignature:*

**Identifier***TypeArgumentsopt*

*ClassTypeSignatureSuffix:*

**.***SimpleClassTypeSignature*

*TypeVariableSignature:*

**T Identifier ;**

*TypeArguments:*

**<***TypeArgument+***>**

*TypeArgument:*

*WildcardIndicatoropt FieldTypeSignature*

**\***

*WildcardIndicator:*

**+**

**-**

*ArrayTypeSignature:*

**[***TypeSignature*

TypeSignature:

*FieldTypeSignature*

*BaseType*

**以上定义没有看懂？？例子如：**

对class MyClass<T> { } 定义的类，产生如下的签名：

<T:Ljava/lang/Object;>Ljava/lang/Object;

而对以下类定义：

**class** MyClass<T1, T2> **extends** ClassFileParser **implements** IndexParser {

}

则产生如下签名：

<T1:Ljava/lang/Object;T2:Ljava/lang/Object;>Lorg/levin/classfilereader/ClassFileParser;Lorg/levin/classfilereader/IndexParser;

**2.**       **字段签名**

**语法定义如上，没能看懂。从Tomcat代码中的Digester.class文件中可以解析得到如下的例子：**

Ljava/util/HashMap<Ljava/lang/String;Ljava/util/Stack<Ljava/lang/String;>;>;（对应的descriptor：“Ljava/util/HashMap;”）

Ljava/util/Stack<Ljava/lang/Object;>;（对应的descriptor：“Ljava/util/Stack;”）

**3.**       **方法签名**

**语法定义：**

*MethodTypeSignature:*

*FormalTypeParametersopt***(***TypeSignature\****)***ReturnType*

*ThrowsSignature\**

*ReturnType:*

*TypeSignature*

*VoidDescriptor*

*ThrowsSignature:*

**^***ClassTypeSignature*

**^***TypeVariableSignature*

**也没能看懂。同样从Tomcat代码中的Digester.class文件中可以解析得到如下例子：**

(Ljava/lang/String;Ljava/lang/Class<\*>;Ljava/lang/String;)V（对应descriptor：“(Ljava/lang/String;Ljava/lang/Class;Ljava/lang/String;)V”）

(Ljava/lang/String;Ljava/lang/Class<\*>;Ljava/lang/String;Z)V（对应descriptor：“(Ljava/lang/String;Ljava/lang/Class;Ljava/lang/String;Z)V”）

()Ljava/util/Map<Ljava/lang/String;Ljava/net/URL;>;（对应descriptor：“()Ljava/util/Map;”）

**附录E：annotation结构和element\_value结构**

**1.**      **annotation结构**

每一项annotation结构记录一项用户定义的annotation的值。如：

    @Test(id = 4, description = "description", useCase = @UseCase())

    @UseCase()

**void** testExecute(**int** a) {

    }

编译器会为该方法生成两项annotation。每项annotation指定了annotation的类型和键值对。

|  |  |  |
| --- | --- | --- |
| **annotation结构** | | |
| **type** | **descriptor** | **remark** |
| u2 | type\_index | constant\_pool中的索引。CONSTANT\_Utf8\_info类型。以字段描述符（field descriptor）方式记录当前结构表示的annotation类型。 |
| u2 | num\_element\_value\_pairs | 记录当前annotation中的键值对数。 |
| |  |  |  | | --- | --- | --- | | element\_value\_pair | | | | 记录每项annotation中的键值对表。 | | | | u2 | element\_name\_index | constant\_pool中的索引。CONSTANT\_Utf8\_info类型。记录当前annotation中当前键值对的键名。如上例的“id”、“description”等。 | | element\_value | value | 当前annotation中当前键值对的值。详见element\_value结构一节。 | | element\_value\_pairs[num\_element\_value\_pairs] | | | | | |

**2.**      **element\_value结构**

element\_value结构记录了所有annotation类型的键值对中的值。它是一个联合类型，可以表示多种类型的值。

|  |  |  |
| --- | --- | --- |
| **element\_value结构** | | |
| **type** | **descriptor** | **remark** |
| u1 | tag | tag记录了当前annotation键值对中值的类型，’B’、’C’、’D’、’F’、’I’、’J’、’S’、’Z’表示基本类型（见附录B中的基本类型对应表）；其他的合法值有：  ’s’ -> String  ‘e’ -> enum constant  ‘c’ -> class  ‘@’ -> annotation type  ‘[‘ -> array |
| |  |  |  | | --- | --- | --- | | value 联合体类型（union） | | | | union类型，记录当前annotaion键值对中的值。 | | | | u2 | constant\_value\_index | constant\_pool中的索引，索引项必须是常量类型。当tag中的值为’B’ ‘C’ ‘D’ ‘F’ ‘I’ ‘J’ ‘S’ ‘Z’ ‘s’时该项有效。 | | enum\_const\_value | | | | 当tag值为’e’时，该项有效。记录枚举类型值。 | | | | u2 | type\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。记录当前枚举类型二进制名（binary name，好像就是类型名，以descriptor的形式表示）。 | | u2 | const\_name\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。记录当前枚举类型的值（枚举类型内部成员字符串）。 | | enum\_const\_value | | | | u2 | class\_info\_index | constant\_pool中的索引，CONSTANT\_Utf8\_info类型。以descriptor记录当前值所表达的Class类型。当tag值为’c’时，该项有效。 | | annotation | annotation\_value | 当tag值为’@’时，该项有效。记录当前annotation键值对中的值为内嵌的annotation。 | | array\_value | | | | 当tag值为’[‘时，该项有效。记录当前annotation键值对中的值为数组类型。 | | | | u2 | num\_values | 数组的长度。 | | element\_value | values[num\_values] | 每一项记录数组中的值。 | | array\_value | | | | value | | | | | |

注：从这个结构中，我们也可以得出annotation中可以设置的值类型：

1.       基本类型值（byte、char、double、float、int、long、short、boolean）

2.       字符串（String）

3.       枚举（enum）

4.       类实例（Class）

5.       嵌套注解类型（annotation）

6.       以上所有以上类型的一维数组。