1. **参考**

[Java Virtual Machine Specification](http://docs.oracle.com/javase/specs/jvms/se8/html/index.html)

[Chapter 4. The class File Format](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html)

<http://blog.csdn.net/ochangwen/article/details/51457398>

1. **Class文件**

是一组以8字节为基础单位的二进制流，

各个数据项目严格按照顺序紧凑排列在class文件中，中间没有任何分隔符，

这使得class文件中存储的内容几乎是全部程序运行的必要数据，没有空隙存在。

Class文件采用Big endian方式存储字节序。

Big endian：最高字节在地址最低位(左)，最低字节在地址最高位，依次排列。

如果我们将u4类型的0x1234abcd写入到以0x0000开始的内存中，则结果为

          big-endian     little-endian

0x0000     0x12              0xcd

0x0001     0x34              0xab

0x0002     0xab              0x34

0x0003     0xcd              0x12

<http://blog.csdn.net/sunshine1314/article/details/2309655>

1. **数据结构**

Java虚拟机规范规定，Class文件格式采用类似[C语言](http://lib.csdn.net/base/c)结构体的伪结构来存储数据，这种结构只有两种数据类型：无符号数和表。

* 1. 无符号数

基本数据类型，使用u1、u2、u4、u8分别表示1字节、2字节、4字节和8字节的无符号数。

* 1. 表

是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有的表都习惯以“\_info”结尾。

* 1. 集合

通常用一个前置的容器计数器n 加 n个连续的数据项表示某一类型数据的集合。

如字段集合由1个u2类型的field\_count和field\_count个field\_info数据项组成。

1. **class文件格式**

可以把整个Class文件看作一张表，其格式如下：

ClassFile {

u4 magic;

u2 minor\_version;

u2 major\_version;

u2 constant\_pool\_count;

cp\_info constant\_pool[constant\_pool\_count-1];

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];

}

依次为魔数、次版本号、主版本号、常量池、类访问标志、当前类名、父类名、接口表集合、字段表集合、方法表集合、类的属性表集合。

* 1. **分析工具**
     1. 创建测试类

package com.tiro.jvm;

public class TestClass {

private int m;

public int inc() {

return m + 1;

}

}

* + 1. 使用javac编译

Javac TestClass.java

* + 1. 使用十六进制工具打开

Linux工具：

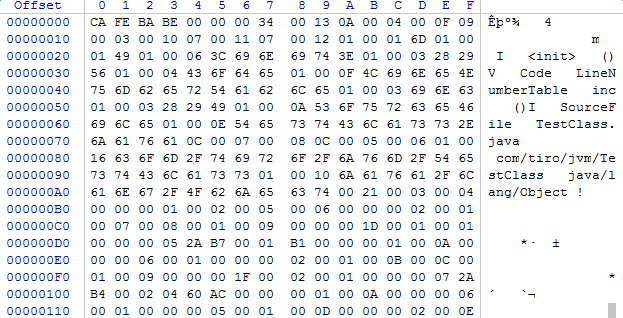
od -t x1 TestClass.class   内核命令

hexdump TestClass.class

Windows工具：

winhex  hexdump的windows GUI版

文本编辑器的hexdump插件   如notepad++，vscode



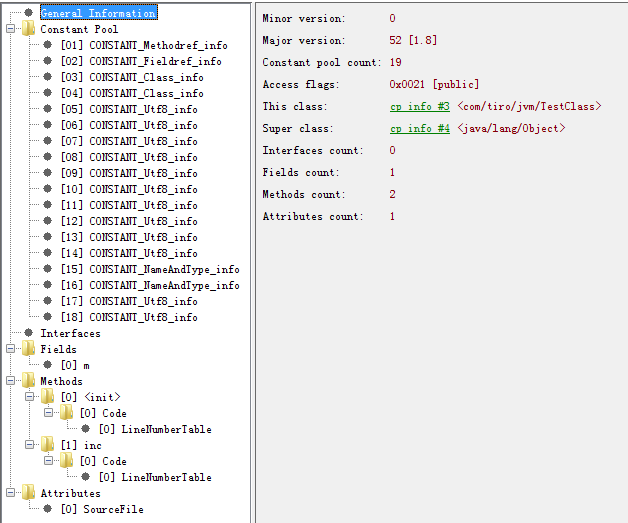
* + 1. 使用字节码反编译工具

javap：jdk自带的工具，比较方便



jclasslib bytecode viewer：字节码查看神器，有IDEA插件版

<https://github.com/ingokegel/jclasslib>



* 1. **魔数**



每个Class文件的头4个字节称为魔数（Magic Number）

唯一作用是用于确定这个文件是否为一个能被虚拟机接受的Class文件。

Class文件魔数的值为0xCAFEBABE。如果一个文件不是以0xCAFEBABE开头，那它就肯定不是[Java](http://lib.csdn.net/base/java)class文件。

* 1. **版本号**



紧接着魔数的4个字节是Class文件版本号.

版本号又分为：

**次版本号(minor\_version)**: 前2字节用于表示次版本号

**主版本号(major\_version)**: 后2字节用于表示主版本号。

Java的主版本号是从45开始的，版本号与JDK版本的对应关系如下：

0X0034（对应十进制的52）：JDK1.8

0X0033（对应十进制的51）：JDK1.7

0X0032（对应十进制的50）：JDK1.6

0X0031（对应十进制的49）：JDK1.5

0X0030（对应十进制的48）：JDK1.4

0X002F（对应十进制的47）：JDK1.3

0X002E（对应十进制的46）：JDK1.2

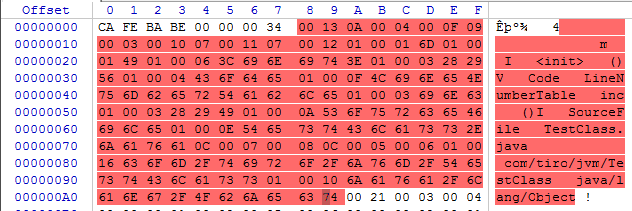
0X002D（对应十进制的45）：JDK1.1

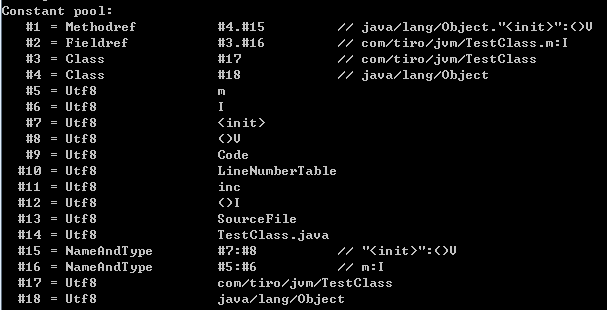
高版本的JDK能向下兼容低版本的Class文件，如果Class文件的版本号超过虚拟机版本，将被拒绝执行，并在类加载时抛出如下异常：

Exception in thread "main" java.lang.UnsupportedClassVersionError: WriteLog : Unsupported major.minor version 52.0

52.0对应jdk1.8即class文件是由jdk1.8的编译器生成，而JVM的版本低于1.8时，报不支持的Class版本错误

* 1. **常量池**





紧接着魔数与版本号之后的是常量池入口，常量池可以理解为class文件的资源仓库，可被后面的其它数据项以索引形式重用。

* + 1. 常量池数量



常量池容量计数器constant\_pool\_count显示有19个常量，但实际的常量项目只有18项。Why？

常量的索引从1开始，如果constant\_pool\_count值为21，索引从1~20；

第0项常量空出来用于某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义。

* + 1. 常量类型

在常量池中共有14种常量类型，每种常量类型都是一张表，拥有自己的表结构。表开始的u1类型为标志位tag，代表当前这个常量属于哪种常量类型。以下为常量类型与tag的对应关系：

|  |  |  |
| --- | --- | --- |
| **Constant Type** | **Value** | **Description** |
| CONSTANT\_Class | 7 | 类的符号引用 |
| CONSTANT\_Fieldref | 9 | 字段的符号引用 |
| CONSTANT\_Methodref | 10 | 方法的符号引用 |
| CONSTANT\_InterfaceMethodref | 11 | 接口方法的符号引用 |
| CONSTANT\_String | 8 | 字符串类型字面量 |
| CONSTANT\_Integer | 3 | 整型字面量 |
| CONSTANT\_Float | 4 | 单精度浮点型字面量 |
| CONSTANT\_Long | 5 | 长整型字面量 |
| CONSTANT\_Double | 6 | 双精度浮点型字面量 |
| CONSTANT\_NameAndType | 12 | 字段或方法的部分符号引用 |
| CONSTANT\_Utf8 | 1 | Utf-8编码的字符串 |
| CONSTANT\_MethodHandle | 15 | 方法语柄 |
| CONSTANT\_MethodType | 16 | 方法描述符 |
| CONSTANT\_InvokeDynamic | 18 | 动态调用 |

常量池数量之后，便是各个常量项目，以第1个常量项为例：



0x0A对应CONSTANT\_Methodref常量类型，其表结构为：

CONSTANT\_Methodref\_info {

u1 tag;

u2 class\_index;

u2 name\_and\_type\_index;

}

后面两个u2类型0x0004和0x000F表示第4和15个常量的索引。javap反编译的结果为如：



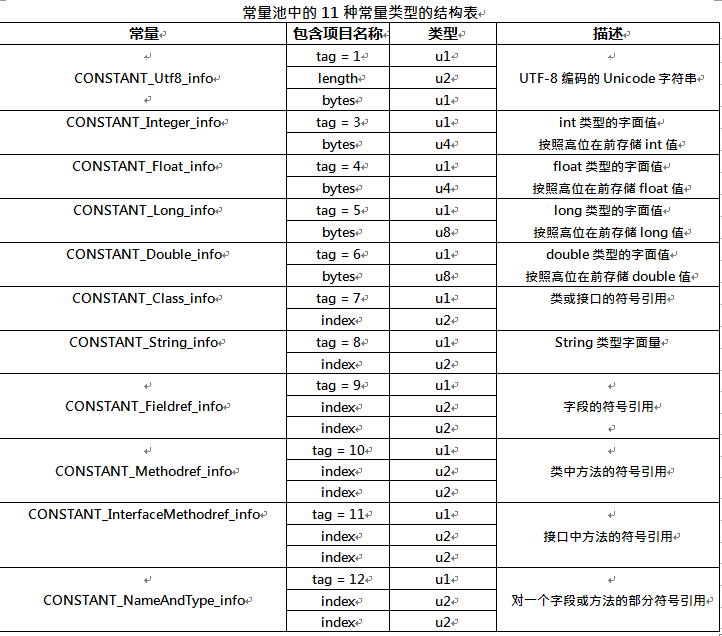
#1 表示索引为1的常量；

Methodref表示常量类型为方法符号引用；

#4.#15 表示常量的内容， 常量是可以通过索引复用其它常量的，中间的.是javap加上去的；

// 表示常量的实际内容。

其它的17个常量项都可以根据各个常量类型的结构从字节码手动计算出来。以下为11种常量类型的表结构：



* + 1. 字面量和符号引用

虽然常量池有14种常量类型，但按照其功能主要可分为字面量和符号引用两大类。

字面量

基本类型字面量：CONSTANT\_Integer、CONSTANT\_Float、CONSTANT\_Long、CONSTANT\_Double

当java基本类型被static final修饰时生成，byte、char、boolean被转换成int。

private final static int = 2;

字符串字面量：java中在任何地方出现的字符串字面量都被生成1个CONSTANT\_String\_info和1个CONSTANT\_Utf8\_info类型。

private String str = "hello";

utf-8字面量：

字节码文件中所有需要用到字符串描述的地方都会生成CONSTANT\_Utf8\_info类型。

符号引用：用来准确描述类、字段和方法的字符串。

描述类和接口：类和接口的全限定名   com/tiro/jvm/Test;

描述字段：   类全限定名称.字符名称:字段描述符  com/tiro/jvm/Test.m:I

描述方法：   类全限定名称.方法名:方法描述符  java.lang.Object."<init>":()V

* + 1. 类或接口全限定名

把类全名中的.替换为/即为类的全限定名称，以;表示全限定名称结束。

* + 1. 字段描述符

主要用来描述字段的数据类型。

根据描述符规则，基本数据类型以及无返回值的void类型都用一个大写字符来表示；引用类型则用字符L加类的全限定名来表示；数组使用前置的[来描述，如一维数组int[]描述为[I,二组数组java.lang.String[][]描述为[[Ljava.lang.String;

|  |  |  |
| --- | --- | --- |
| ***FieldType* term** | **Type** | **Interpretation** |
| B | byte | signed byte |
| C | char | Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16 |
| D | double | double-precision floating-point value |
| F | float | single-precision floating-point value |
| I | int | integer |
| J | long | long integer |
| L *ClassName* ; | reference | an instance of class *ClassName* |
| S | short | signed short |
| Z | boolean | true or false |
| [ | reference | one array dimension |

* + 1. 方法描述符

用来描述方法的参数列表（包括数量、类型以及顺序）和返回值 。

public com.tiro.jvm.TestClass();方法的描述符为()V

int indexOf(char[] source,int sourceOffset,int sourceCount,char[] target,int targeOffset,int targetCount,int fromIndex);方法的描述符为([CII[CIII)I

* 1. **类访问标志**



 常量池之后的数据结构是类访问标志(access\_flags)。

 具体的标志访问如下：

| **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. |

测试类TestClass使用了ACC\_PUBLIC和ACC\_SUPER两个标志。最终结果通过计算得出：

ACC\_PUBLIC | ACC\_SUPER = 0x0001 | 0x0020 = 0x0021

* 1. **类索引、父类索引和接口索引集合**



这三项数据主要用于确定这个类的继承关系。

0x0003 类索引，指向第3个常量；



0x0004 父类索引，指向第4个常量，只有1个，这说明Java语言不允许多重继承；



0x0000 接口索引集合，当前测试类没有实现接口，所以接口容器计数器为0，接口项也是没有的。

* 1. **字段表集合**

字段表集合由容器计数器fields\_count和fields\_count个field\_info组成。

* + 1. 字段数量fileds\_count

字段包括类变量(static修饰的类静态变量和static final修饰的常量)以及实例变量，不包含方法内部声明的局部变量。

* + 1. 字段表field\_info

field\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

access\_flags为字段的访问标志

| **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. |

name\_index为字段的简单名称的常量索引

descriptor\_index为字段描述符的常量索引

attributes\_count为字段属性数量

attribute\_info 为字段属性表

* + 1. 字节码分析



0x0001 测试类中只有1个字段

0x0002 字段访问标志为ACC\_PRIVATE

0x0005 字段名称为第5个常量m

0x0006 字段描述符为第6个常量I，即int类型

0x0000 当前字段没有属性表集合

综上可以推断出源码为：

private int m;

* 1. **方法表集合**

方法表集合由容器计数器methods\_count和methods\_count个method\_info组成。

* + 1. 方法数量methods\_count

包括实例方法、静态方法、<init>实例初始化方法、<clinit>类初始化方法以及接口方法。

* + 1. 方法表method\_info

method\_info {

u2 access\_flags;

u2 name\_index;

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

access\_flags为字段的访问标志

|  |  |  |
| --- | --- | --- |
| **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](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-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. |

name\_index为方法简单名称的常量索引

descriptor\_index为方法描述符的常量索引

attributes\_count为方法属性数量

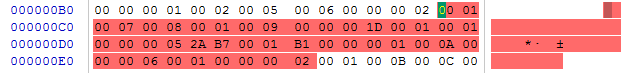
attribute\_info 为方法属性表

* + 1. 字节码分析



0x0002 测试类有2个方法。

第一个方法：



0x0001 方法访问标志为public

0x0007 方法名称引用第7个常量<init>

0x0008 方法描述符引用第8个常量()V

0x0001 有1个方法属性

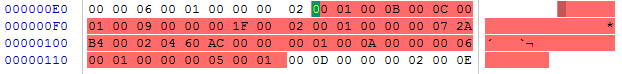
0x0009 第1个方法属性的名称引用第9个常量“Code”,之后为Code属性的内容

综上，这个方法为：



是由编译器生成的实例构造器<init>方法。

第二个方法：



0x0001 方法访问标志为public

0x0007 方法名称引用第11个常量inc

0x0008 方法描述符引用第12个常量()I

0x0001 有1个方法属性

0x0009 第1个方法属性的名称引用第9个常量“Code”,之后为Code属性的内容

综上，这个方法为：



* + 1. 方法特征签名

java方法的特征是一个方法各个参数在常量池中的字段符号引用的集合，不包含返回值，因此Java语言无法仅仅依靠返回值的不同来对一个已有方法进行重载。

但是在Class文件中，特征签名的范围更大一些，只要描述符不是完全一致的方法也可以共存，也就是说仅仅返回值不同的两个方法可以合法共存于一个Class文件中。

* 1. **属性表集合**

在Class文件、字段表、方法表都可携带自己的属性表集合。

属性表集合不要求其中包含的属性表具有严格的顺序，并且只要属性的名称不与已有的属性名称重复，任何人实现的编译器可以向属性表中写入自己定义的属性信息。虚拟机在运行时会忽略不能识别的属性。

* + 1. 预定义属性

虚拟机规范中预定义了虚拟机必须能识别的21项属性：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute** | **Location** | **Section** | **class file** | **Java SE** |
| ConstantValue | method\_info | [§4.7.2](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.2) | 45.3 | 1.0.2 |
| Code | method\_info | [§4.7.3](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.3) | 45.3 | 1.0.2 |
| StackMapTable | Code | [§4.7.4](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.4) | 50.0 | 6 |
| Exceptions | method\_info | [§4.7.5](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.5) | 45.3 | 1.0.2 |
| InnerClasses | ClassFile | [§4.7.6](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.6) | 45.3 | 1.1 |
| EnclosingMethod | ClassFile | [§4.7.7](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.7) | 49.0 | 5.0 |
| Synthetic | All | [§4.7.8](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.8) | 45.3 | 1.1 |
| Signature | All | [§4.7.9](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.9) | 49.0 | 5.0 |
| SourceFile | ClassFile | [§4.7.10](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.10) | 45.3 | 1.0.2 |
| SourceDebugExtension | ClassFile | [§4.7.11](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.11) | 49.0 | 5.0 |
| LineNumberTable | Code | [§4.7.12](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.12) | 45.3 | 1.0.2 |
| LocalVariableTable | Code | [§4.7.13](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.13) | 45.3 | 1.0.2 |
| LocalVariableTypeTable | Code | [§4.7.14](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.14) | 49.0 | 5.0 |
| Deprecated | All | [§4.7.15](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.15) | 45.3 | 1.1 |
| RuntimeVisibleAnnotations | method\_info | [§4.7.16](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.16) | 49.0 | 5.0 |
| RuntimeInvisibleAnnotations | method\_info | [§4.7.17](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.17) | 49.0 | 5.0 |
| RuntimeVisibleParameterAnnotations | method\_info | [§4.7.18](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.18) | 49.0 | 5.0 |
| RuntimeInvisibleParameterAnnotations | method\_info | [§4.7.19](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.19) | 49.0 | 5.0 |
| RuntimeVisibleTypeAnnotations | All | [§4.7.20](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.20) | 52.0 | 8 |
| RuntimeInvisibleTypeAnnotations | All | [§4.7.21](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.21) | 52.0 | 8 |
| AnnotationDefault | method\_info | [§4.7.22](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.22) | 49.0 | 5.0 |
| BootstrapMethods | ClassFile | [§4.7.23](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.23) | 51.0 | 7 |
| MethodParameters | method\_info | [§4.7.24](http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.24) | 52.0 | 8 |

* + 1. **Code属性**

java方法体中代码经过编译器处理后，最终变为字节码指令存储在Code属性中。没有方法体就没有Code属性，如接口或抽象类中抽象方法。

Code属性表结构

Code\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 max\_stack;

u2 max\_locals;

u4 code\_length;

u1 code[code\_length];

u2 exception\_table\_length;

{ u2 start\_pc;

u2 end\_pc;

u2 handler\_pc;

u2 catch\_type;

} exception\_table[exception\_table\_length];

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

attribute\_name\_index 属性名称索引，指向常量值固定为“Code”的常量

attribute\_length 代表属性值的长度，单位为字节，不包括attribute\_name\_index和attribute\_length这6个字节

max\_stack代表方法运行时，操作数栈深度的最大值

max\_locals代表方法运行时，局部变量表的最大存储空间，单位为Slot。32位虚拟机中，不超过32位的数据类型占用1个Slot，long和double占用两个Slot；64位虚拟机中，所有的数据类型占用1个Slot

code\_length和code用于存储Java源程序编译后生成的字节码指令。字节码指令由1个字节的操作码加若干个操作数组成。操作数也会占用空间，所有code\_length不是指字节码指令的个数，而是字节数，字节的索引称之为字节码行号,从0开始。

字节码分析（以<init>方法为例）



0x0009 当前为<init>方法的Code属性

0x0000001D Code属性长度为29个字节

0x0001 方法运行时，操作数栈深度最大值为1

0x0001 方法运行时，局部变量表的最大存储空间为1个Slot

0x00000005 字节码指令集合的长度为5个字节

0x2A 对应指令aload\_0，将第0个Slot的引用类型压入操作数栈，这里存的父类java.lang.Object的引用

0xB70001 对应指令invokespecial #1，调用操作数栈顶java.lang.Object对象的<init>方法

0B1 对应指令return，返回此方法，返回值为void，这条指令执行后，方法结束。

0x0000 异常表为空

0x0001 有1个附加属性

0x000A 附加属性的名称为LineNumberTable，后面是该属性的内容

javap计算如下：



等同于源码：



args\_size参数个数为什么为1？

实例方法的局部变量表会把当前对象实例或父类对象的引用当作方法参数存放到第0个slot，所有对this关键字或super关键字的访问转换成对一个普通方法参数的访问。

异常表exception\_table

异常表是Code属性的一部分，用来描述java异常及finally处理机制。

u2 exception\_table\_length;

{ u2 start\_pc;

u2 end\_pc;

u2 handler\_pc;

u2 catch\_type;

} exception\_table[exception\_table\_length];

它包含4个字段，这些字段的含义如下：

当字节码在第start\_pc行到第end\_pc行之间（不含第end\_pc行）出现了类型为catch\_type或者其子类的异常(CONSTANT\_Class\_info常量)，则转到第handler\_pc行继承处理，当catch\_type的值为0时，代表任何异常情况都需要转向到handler\_pc处进行处理。

每1个exception\_table代表着一条可能出现的异常执行路径。

测试类：

package com.tiro.jvm;

public class TestClass {

private int m;

public int inc() {

int x;

try{

x = 1;

return x;

}catch(Exception e){

x = 2;

return x;

}finally{

x = 3;

}

}

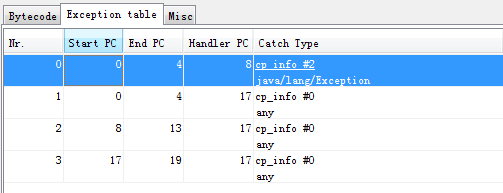
}

方法inc有7条执行路径，4条可能的异常路径：

1. try中的代码正常执行；
2. try中的代码抛出Exception及其子类异常；
3. try中的代码抛出非Exception异常；
4. catch中的代码正常执行；
5. catch中的代码抛出异常；
6. finally块中的代码正常执行；
7. finally块中的代码抛出异常；

inc方法的Code属性如下：





javap计算的异常表没有jclasslib bytecode viewer精准。

可以看出，有4条异常表信息对应4条可能的异常路径。

第1条含义，执行字条码行号0~3，如果发生Catch Type为java/lang/Exception，从字节码行号8继续执行。

0：iconst\_1 把int类型常量1入栈；

1：istore\_1 将栈顶的int类型1出栈，并存储到局部变量表的第1个slot； //try块的x=1赋值

2：iload\_1 将局部变量表的第1个slot即int类型1入栈；

3：istore\_2 将栈顶的int类型1出栈，并存储到局部变量表的第2个slot；

这时操作数栈深度为0，局部变量表的第1和2个slot存的都是int类型1，为什么要存2遍？因为第2个slot是当returnValue的，第12条指令也是如此。

8：astore\_2 将栈顶的异常类型引用（异常发生时会把catch块的参数e压入栈顶）出栈，并存储到局部变量表的第2个slot，原来的1被替换掉；

9：iconst\_2 将int类型常量2入栈；

10：istore\_1 将栈顶的int类型2出栈，并存储到局部变量表的第1个slot； //catch块的x=2赋值

。。。。。

后面的分析同理。

* + 1. **LineNumberTable属性**

Code属性的附加属性，用于描述Java源码行号与字节码行号之间的对应关系。当抛出异常时，堆栈中显示的行号就是通过该属性获取。

1：invokespecial #1 最左边的1就是字节码行号。

LineNumberTable\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 line\_number\_table\_length;

{ u2 start\_pc;

u2 line\_number;

} line\_number\_table[line\_number\_table\_length];

}

start\_pc 代表字节码行号

line\_number 代表源代码行号

* + 1. **LocalVariableTable属性**

Code属性的附加属性，用于描述栈帧中局部变量表中的变量与Java源码中定义的变量之间的关系。需要通过-g:vars才会生成这项信息。如果未生成，参数及局部变量名会被诸如arg0、arg1之类的点位符代替为原有的名称。

LocalVariableTable\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 local\_variable\_table\_length;

{ u2 start\_pc;

u2 length;

u2 name\_index;

u2 descriptor\_index;

u2 index;

} local\_variable\_table[local\_variable\_table\_length];

}

start\_pc表示局部变量作用域开始的字节码行号

length 表示局部变量作用域覆盖的长度

name\_index和descriptor\_index 表示变量名称及描述符

index 局部变量在局部变量表的第几个slot位置

public class TestClass {

private int m;

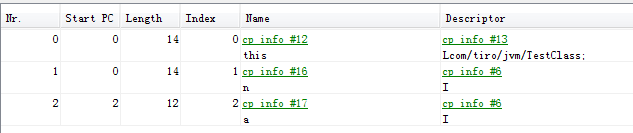
public int inc(int n) {

int a = 1;

return m + m + a;

}

}



表明有3个局部变量this、n、a。

* + 1. **StackMapTable属性**

Code属性的附加属性，这个属性会记录一系列的验证类型Verification Type，这些验证类型在虚拟机类加载的字节码验证阶段被新类型检查验证器Type Checker使用，通过检查这些验证类型代替了旧的类型推导过程，从而大幅提升了字节码验证的性能。

* + 1. **Exceptions属性**

与Code同级，用于描述throws关键字后面列举的异常。

Exceptions\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 number\_of\_exceptions;

u2 exception\_index\_table[number\_of\_exceptions];

}

* + 1. **SourceFile属性**

描述源文件。每个Class文件必有。

SourceFile\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 sourcefile\_index;

}



0x000D 属性名称引用第13个常量“SourceFile”

0x00000002 属性长度为2个字节

0x000E 属性值引用第14个常量“TestClass.java”

* + 1. **ConstantValue属性**

如果同时用final和static来一个变量(通常称之为常量)，并且这个变量的数据类型是基本类型或者java.lang.String的话，就会生成ConstantValue属性来进行初始化，如果这个变量没有被final修饰，或者并非基本类型及字符串，则将会选择在<clinit>方法中进行初始化。

package com.tiro.jvm;

public class TestClass {

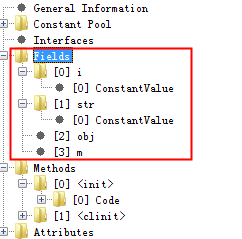
private final static int i = 99999;

private final static String str = "hello world";

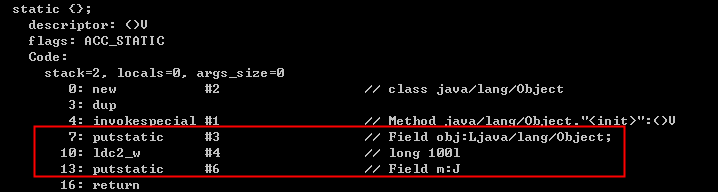
private final static Object obj = new Object();

private static long m = 100L;

}



只有变量i和str生成了ConstatntVaulue属性。



变量obj和m在<clinit>方法中进行初始化。

* + 1. **InnerClasses属性**

用于记录内部类与宿主类之间的关联。

InnerClasses\_attribute {

u2 attribute\_name\_index;

u4 attribute\_length;

u2 number\_of\_classes;

{ u2 inner\_class\_info\_index;

u2 outer\_class\_info\_index;

u2 inner\_name\_index;

u2 inner\_class\_access\_flags;

} classes[number\_of\_classes];

}

inner\_class\_info\_index表示内部类的常量索引

outer\_class\_info\_index表示外部类的常量索引

inner\_name\_index 内部类的简单名称的常量索引

inner\_class\_access\_flags 内部类的访问标志

package com.tiro.jvm;

public class TestClass {

class innerClass{

}

}

