- 使用 javap -verbose 命令分析一个字节码文件时， 将会分析该字节码文件的魔 数、版本号、常量池、类信息、类的构造方法、类中的方法信息、 类变量与成员变量等信息。
- 魔数： 所有的.class 字节码文件的前4个字节都是魔数，魔数为固定值: 0xCAFEBABE
- 版本信息，魔数之后的4个字节是版本信息，前两个字节表示 minor version (次版本号), 后2个字节表示major version (主版本号)。这里的版本号 00 00 00 34 换算成十进制表， 表示次版本号为0, 主版本号为 52. 所以该文件的版本号为 1.8.0。可以通过 java -version 来验证这一点。
- 常量池 （constant pool）: 2+N个字节 紧接着主版本号之后的就是常量池入口。一个java 类中定义的很多信息都是由常量池来描述的，可以将常量池看作是 Class 文件的资源仓库，比如说Java类中变量的方法与变量信息，都是存储在常量池中。常量池中主要存储2类常量：字面量与符号引用。
    - 字面量, 如字符串文本，java 中声明为final 的常量值等。
    - 符号引用, 如类和接口的全局限定名， 字段的名称和描述符，方法的名称和 描述符等。
- 常量池的总体结构： Java类所对应的常量池主要由常量池（常量表）的数量与常量池数组这两部分共同构成。常量池数量紧跟着在主版本号后面，占据2字节: 常量池 数组则紧跟着常量池数量之后。常量池数组与一般数组不同的是， 常量池数组中不同的元素的类型，结构都是不同的。长度当然也就不同；但是，一种元素的第一种 元素的第一个数据都是一个u1类型, 该字节是一个标识位，占据1个字节。 JVM在解析常量池时，会更具这个u1 类型来获取元素的具体类型。值得注意的是 常量池中元素的个数 = 常量池数 -1 (其中0暂时不适用), 目的是满足某些常量池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】的含义： 根本原因在于，索引为0也是一个常量（保留常量），只不过他不位于常量表中。这个常量就对应null值， 所以常量池的索引是从1开始而非0开始。

- 常量池数据结构表

Class文件结构中常量池中11种数据类型的结构总表			
常量	项目	类型	描述
CONSTANT_Utf8_info	tag	U1	值为1
	length	U2	UTF-8编码的字符串长度
	bytes	U1	长度为length的UTF-8编码的字符串
CONSTANT_Integer_info	tag	U1	值为3
	bytes	U4	按照高位在前存储的int值
CONSTANT_Float_info	tag	U1	值为4
	bytes	U4	按照高位在前存储的float值
CONSTANT_Long_info	tag	U1	值为5
	bytes	U8	按照高位在前存储的long值
CONSTANT_Double_info	tag	U1	值为6
	bytes	U8	按照高位在前存储的double值
CONSTANT_Class_info	tag	U1	值为7
	index	U2	指向全限定名常量项的索引
CONSTANT_String_info	tag	U1	值为8
	index	U2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	U1	值为9
	index	U2	指向声明字段的类或者接口描述符CONSTANT_Class_info的索引项
	index	U2	指向字段描述符CONSTANT_NameAndType_info的索引项
CONSTANT_Methodref_info	tag	U1	值为10
	index	U2	指向声明方法的类描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_InterfaceMethodref_info	tag	U1	值为11
	index	U2	指向声明方法的接口描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_NameAndType_info	tag	U1	值为12
	index	U2	指向该字段或方法名称常量项的索引
	index	U2	指向该字段或方法描述符常量项的索引

- 上面表中描述了11种数据类型的机构， 其实在jdk1.7之后又增加了3种（CONSTANT_MethodHandle_info, CONSTANT_MethodType_info以及 CONSTANT_MethodType_info 以及 CONSTANT_InvokeDynami_info）。这样一共14种。
- 在JVM规范中， 每个变量/字段都有描述信息， 描述信息主要的作用是描述字段的数据类型、方法的参数列表（包括数量、类型、顺序）与返回值。根据描述符 规则， 基本数据类型和代表无返回值的的void 类型都用一个大写字符来表示， 对象类型则使用字符L加对象的全限定名称来表示。为了压缩字节码文件的体积 对于基本数据类型，JVM都只使用一个大写字母聊表示，如下所示：B-byte, C-char, D-double, F-float, I-int, J-long, S-short, Z-boolean , V -void L -表示对象类型，如：Ljava/lang/String;
- 对于数组类型来说，每一个维度使用一个前置的 [来表示， 如int[] 被标记为 [I , String[][]被表示为 [[Ljava/lang/String;
- 用描述符描述方法时, 按照先参数列表， 后返回值的顺序来描述. 参数列表按照参数的严格顺序放在一组()内， 如方法: String

getRealnameByIdNickname(int id, String name)的描述符为: (I, Ljava/lang/String;) Ljava/lang/String

- 字节码整体结构

类型	名称	数量
u4	magic(魔术)	1
u2	minor_version(次版本号)	1
u2	major_version(主版本号)	1
u2	constant_pool_count(常量个数)	1
cp_info	constant_pool(常量池表)	constant_pool_count-1
u2	access_flags(类的访问控制权限)	1
u2	this_class(类名)	1
u2	super_class(父类名)	1
u2	interfaces_count(接口个数)	1
u2	interfaces(接口名)	interfaces_count
u2	fields_count(域个数)	1
field_info	fields(域的表)	fields_count
u2	methods_count(方法的个数)	1
method_info	methods(方法表)	methods_count
u2	attributes_count(附加属性的个数)	1
attribute_info	attributes(附加属性的表)	attributes_count

- 完整Java 字节码结构

```
ClassFile {
    u4                  magic;
    u2                  minor_version;
    u2                  major_version;
    u2                  constant_pool_count;
    cp_info             contant_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];
}
```

- Class 字节码中有两种数据类型

- 字节数据直接量： 这是基本的数据类型。共细分为u1、u2、u4、u8四种，分别代表连续的1个字节、2个字节、4个字节、8个字节组成的整体数据。
- 表（数组）：表时由多个基本数据或其他表，按照既定顺序组成的大的数据集合。表是有结构的 。它的结构体现在：组成表的成分所在的位置和顺序都是 严格定义好的。

- Java 字节码整体结构

- Access_Flag 访问标志 访问标识信息包括该Class文件时类和接口是否被定义成 了public，是否是 abstract, 如果是类，是否被申明为成final。通过扇面的源代码。
- 0x 00 21: 表示是0x0020 和0x0001的并集, 表示ACC_PUBLIC与ACC_SUPER

Flag Name	Value	Remarks
ACC_PUBLIC	0x0001	public
ACC_PRIVATE	0x0002	private
ACC_PROTECTED	0x0004	protected
ACC_STATIC	0x0008	static
ACC_FINAL	0x0010	final
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。不可与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

- 字段表用于描述类的接口汇总声明的变量。这里的字段包括了类级别变量以及实例变量, 但是不包括方法内部声明的局部变量。
- 字段表集合, fields_count: u2

字段表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

- 字段表结构

```
field_info {
    u2 access_flags; 0002
    u2 name_index; 0005 (表示字段名称在常量池中的索引位置)
```

```
    u2 descriptor_index; 0006 (描述符索引)
    u2 attributes_count; 0000
    attribute_info attributes[attributes_count]
}
```

- 方法表 methods_count: u2

方法表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

- 方法表结构 前三个字段和field_info一样

```
method_info {
    u2 access_flags; 0001
    u2 name_index; 0017
    u2 descriptor_index; 0018
    u2 attributes_count; 0001
    attributes_info attributes[attributes_count]
}
```

- 方法属性结构

```
attribute_info {
    u2 attribute_name_index; 0019
    u4 attribute_length; 000051
    u1 info[attribute_length];z
}
```

- "Code" 表示下面是执行代码
- JVM 预定义了一部分的attribute, 但是编译器自己也可以实现自己的 attribute 写入class文件中， 供运行时使用。 不同的attribute 通过 attribute_name_index 来区分。
- JVM 规范预定义的attribute

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.

- Code 结构
- Code attribute 的作用是保存该放的的结构，如所对应的字节码

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals
    u4 code_length;
    u1 code[code_lenght];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_lenght];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

- attribute_length 表示 attribute 所包含的字节数， 不包含 attribute_name_index 和 attribute_length 字段。
- max_stack 表示这个方法运行的任何时刻锁能达到的操作数栈的最大深度。
- max_locals 表示方法执行期间创建的局部变量的数目，包含用来表示传入的参数的局部变量。

- code_length 表示该方法所包含的字节码的字节数以及具体的指令码。
- 具体字节码即是该方法被调用时，虚拟机所执行的字节码。
- exception_table, 这里存放的是处理异常信息。
- 每个 exception_table, 这里存放的是处理异常的信息。
    - 每个 exception_table 表项由start_pc , end_pc, handler_pc, catch_type 组成。
    - start_pc 和 end_pc 表示在code数组中的从 start_pc 到 end_pc 处（包含start_pc, 不包含end_pc）的指令抛出的异常会由 这个表项来处理。
    - handler_pc 表示处理异常的代码的开始处。

      catch_type 表示会被处理的异常类型， 它指向常量池里的一个异常类。 当catch_type为0时, 表示处理所有的异常。
- LineNumberTable 的结构

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_lenght;
    u2 line_number_table_length;
    {
        u2 start_pc;
        u2 line_number;
    }
    line_number_table[line_number_table_length];
}
```

- LocalVariableTable的结构

LocalVariableTable属性结构：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_varible_table_length	1
local_variable_info	local_variable_table	local_varible_table_length

local_variable_info项目代表了一个栈帧与源码中的局部变量的关联，结构见下表：

类型	名称	数量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1

u2	index	1

- Synchronized关键字
  - monitorenter代表开始同步
  - monitorexit代表结束

```
 0: aload_1
 1: dup
 2: astore_2
 3: monitorenter
 4: getstatic     #10                 // Field java/lang/System.out:Ljava/io/PrintStream;
 7: ldc           #11                 // String hello world
 9: invokevirtual #12                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
12: aload_2
13: monitorexit
14: goto          22
```

- this和异常

```java
/**
 * Created By poplar on 2019/11/10
 * 对于Java类中的每一个实例方法(非static方法），其在编译后所生成的字节码当中，方法参数的数量总是会比源代码中方法数的数量多一个(this)，
 * 它位于方法的第一个参数位置处;这样,我们就可以在Java的实例方法中使用this来去访问当前对象的属性以及其他方法。
 * 这个操作是在编译期间完成的,即由javac编译器在编译的时候将对this的访问转化为对一个普通实例方法参数的访问;
 * 接下来在运行期间由JVM在调用实例方法时,自动向实例方法传入this参数.所以,在实例方法的局部变量表中,至少会有一个指向当前对象的局部变量
 */

/**
 * Java字节码对于异常的处理方式：
 * 1.统一采用异常表的方式来对异常进行处理；
 * 2.在jdk1.4.2之前的版本中，并不是使用异常表的方式对异常进行处理的，而是采用特定的指令方式；
 * 3.当异常处理存在finally语句块时，现代化的JVM采取的处理方式是将finally语句内的字节码拼接到每个catch语句块后面。
 * 也就是说，程序中存在多少个catch，就存在多少个finally块的内容。
 */
public class ByteCodeTest3 {

    public void test() throws IOException, FileNotFoundException {

        try {
            InputStream is = new FileInputStream("test.txt");

            ServerSocket serverSocket = new ServerSocket(9999);
            serverSocket.accept();
            throw new RuntimeException();

        } catch (FileNotFoundException ex) {

        } catch (IOException ex) {

        } catch (Exception ex) {

        } finally {
            System.out.println("finally");
        }
    }
```

LocalVariableTable:

Start	Length	Slot	Name	Signature
20	16	2	is	Ljava/io/InputStream;
31	5	3	ss	Ljava/net/ServerSocket;
72	4	2	e	Ljava/lang/Exception;
0	101	0	this	Lmain/java/Test1/Test;
（额外增加的一个局部变量）				
3	98	1	str	Ljava/lang/String;

- 总结
- 构造方法中会初始化成员属性的默认值，如果自己实现了默认的构造方法， 依然还是在构造方法中赋值，这就是对指令的重排序。
- 如果多个构造方法那么每个构造方法中都有初始化成员变量的属性，来保障每个构造方法初始化的时候都会执行到属性的初始化过程。

- 如果构造方法中有执行语句, 那么会先执行赋值信息, 然后在执行自定义的执行代码。
- 对于Java每一个实例方法(非静态方法), 其在编译后生成的字节码中比实际方法多一个参数, 它位于方法的第一个参数位置. 我们就可以在当前方法中 的this去访问当前对象中的this这个操作是在Javac 编译器在编译期间将this的访问转换为对普通实例方法的参数访问，接下来在运行期间， 由JVM的调用实例方法时， 自动向实例方法中传入该this参数， 所以在实例方法的局部变量表中， 至少会一个指向当前对象的局部变量。
- 字节码对于处理异常的方式：
- 统一采用异常表的方式来对异常处理。
- 在Jdk1.4.2之前的版本中， 并不是使用异常表的方式来对异常进行处理的，而是采用特定的指令方式。