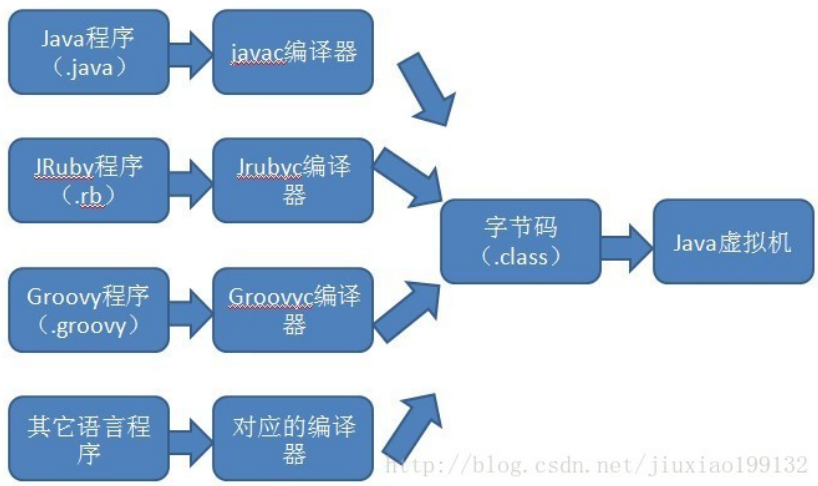
不仅跨平台，还能跨语言

Java能够跨平台在window，linux，mac等不同平台的操作系统上运行是因为Java虚拟机的存在，Javac把Java代码编译成class文件，Java虚
在多个不同操作系统平台都有实现。所有Java语音才能跨平台执行。

但时至今日，商业机构和开源机构已经在Java语言之外发展出一大批在Java虚拟机之上运行的语言，如Clojure、Groovy、JRuby、Jython、



Class文件结构

Class文件是一组以8位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地排列在Class文件中，中间没有添加任何分隔符，这使
部都是程序运行的必要数据。根据**Java虚拟机规范的规定，Class文件格式采用一种类似于C语言结构体的伪结构来存储，这种伪结构中只有**

无符号数属于基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节、8个字节的无符号数，无符号数可以用来描述
UTF-8编码构成字符串值。

表是由多个无符号数或者其它表作为数据项构成的复合数据类型，所有表都习惯性地以”_info”结尾。表用于描述有层次关系的复合结构的数
表。 它由下表所示的数据项构成。

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

由于不包含任何分隔符，故表中的数据项，无论是数量还是顺序，都是被严格限定的。哪个字节代表什么含义，长度是多少，先后顺序如何

查看Class文件的工具winhex

编写一段代码

```
1 package com.test;
2
3 public class Test {
4     private int m;
5
6     public int getM(){
7         return m + 1;
8     }
9 }
```

用winhex打开编译完成的Class文件

Test.class

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D	Êþº¾ 2
00000010	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04	com/test/Test
00000020	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	java/lang/Obj
00000030	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69	ect m I <i
00000040	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	nit> ()V Cod
00000050	65	0A	00	03	00	0B	0C	00	07	00	08	01	00	0F	4C	69	e Li
00000060	6E	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	12	neNumberTable
00000070	4C	6F	63	61	6C	56	61	72	69	61	62	6C	65	54	61	62	LocalVariableTab
00000080	6C	65	01	00	04	74	68	69	73	01	00	0F	4C	63	6F	6D	le this Lcom
00000090	2F	74	65	73	74	2F	54	65	73	74	3B	01	00	04	67	65	/test/Test; ge
000000A0	74	4D	01	00	03	28	29	49	09	00	01	00	13	0C	00	05	tM ()I
000000B0	00	06	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	SourceFile
000000C0	00	09	54	65	73	74	2E	6A	61	76	61	00	21	00	01	00	Test.java !
000000D0	03	00	00	00	01	00	02	00	05	00	06	00	00	00	02	00	
000000E0	01	00	07	00	08	00	01	00	09	00	00	00	2F	00	01	00	/
000000F0	01	00	00	00	05	2A	B7	00	0A	B1	00	00	00	02	00	0C	*· ±
00000100	00	00	00	06	00	01	00	00	00	03	00	0D	00	00	00	0C	
00000110	00	01	00	00	00	05	00	0E	00	0F	00	00	00	01	00	10	
00000120	00	11	00	01	00	09	00	00	00	31	00	02	00	01	00	00	1
00000130	00	07	2A	B4	00	12	04	60	AC	00	00	00	02	00	0C	00	*´ `¬
00000140	00	00	06	00	01	00	00	00	07	00	0D	00	00	00	0C	00	
00000150	01	00	00	00	07	00	0E	00	0F	00	00	00	01	00	14	00	
00000160	00	00	02	00	15												

魔数（Magic）

```
00000000  CA FE BA BE 00 00 00 32  00 16 07 00 02 01 00 0D  ....2........
```

每个Class文件的头4个字节称为魔数（Magic Number），它的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。很多
份识别，譬如图片格式，如gif或jpeg等在文件头中都存有魔数。Class文件魔数的值为0xCAFEBABE。如果一个文件不是以0xCAFEBABE开

Class文件的版本（minor_version 和 major_version）

```
00000000  CA FE BA BE 00 00 00 32  00 16 07 00 02 01 00 0D  ....2........
```

minor_version：占2字节，次版本号，0x0000
majro_version：占2字节，主版本号，0x0032，转化为十进制为50，是使用JDK1.6编译的，或者用的jdk1.7，1.8但是编译版本指定的1.6

表 9.1 Class文件版本号和平台的对应

大版本（十进制）	小 版 本	编译器版本
45	3	1.1
46	0	1.2
47	0	1.3
48	0	1.4
49	0	1.5
50	0	1.6
51	0	1.7
52	0	1.8

高版本的JDK可以向下兼容以前版本的Class文件，但是无法运行以后版本的Class文件，即使文件格式并未发生变化
如果使用JDK1.5运行使用JDK1.6编译的Class文件，会报

```
1 java.lang.UnsupportedClassVersionError: Bad version number in .class file
```

常量池（constant_pool_count 和 constant_pool）

跟随版本之后的就是常量池了。

constant_pool_count

由于常量池中常量的数目不是固定的，所以在常量池入口首先使用一个2字节长的无符号数constatn_pool_count来代表常量池计数值

Test.class

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D

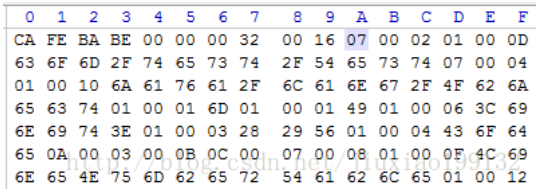
这里描述的0016转成十进制是22，代表常量池中有21个常量。（只有常量池的计数是从1开始的，其它集合类型均从0开始），索引值为1~2
些指向常量池索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义，这种情况可以将索引值置为0来表示

constant_pool

常量池中的每一个常量都是一个表，共有11种结构。这11中表都有一个共同的特点：表开始的第一位是一个u1类型的标志位（tag，取值1到11）来标识常量是哪种常量类型。11种常量类型所代表的具体含义如图：

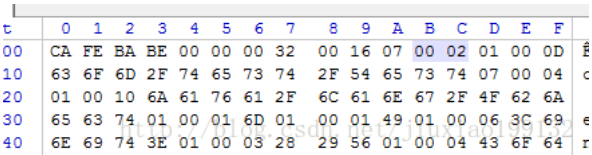
类型	简介	项目	类型	描述
CONSTANT_Utf8_info	utf-8缩略编码字符串	tag	u1	值为1
		length	u2	utf-8缩略编码字符串占用字节数
		bytes	u1	长度为length的utf-8缩略编码字符串
CONSTANT_Integer_info	整形字面量	tag	u1	值为3
		bytes	u4	按照高位在前储存的int值
CONSTANT_Float_info	浮点型字面量	tag	u1	值为4
		bytes	u4	按照高位在前储存的float值
CONSTANT_Long_info	长整型字面量	tag	u1	值为5
		bytes	u8	按照高位在前储存的long值
CONSTANT_Double_info	双精度浮点型字面量	tag	u1	值为6
		bytes	u8	按照高位在前储存的double值
CONSTANT_Class_info	类或接口的符号引用	tag	u1	值为7
		index	u2	指向全限定名常量项的索引
CONSTANT_String_info	字符串类型字面量	tag	u1	值为8
		index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	字段的符号引用	tag	u1	值为9
		index	u2	指向声明字段的类或接口描述符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	类中方法的符号引用	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	接口中方法的符号引用	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	字段或方法的部分符号引用	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

每一个常量都以一个u1类型开头，如下图这里是7对照总表发现tag = 7 的 是 CONSTANT_Class_info类型。



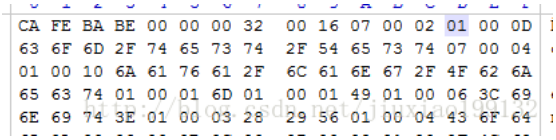
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D
	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04
	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A
	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69
	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64
	65	0A	00	03	00	0B	0C	00	07	00	08	01	00	0F	4C	69
	6E	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	12

CONSTANT_Class_info这个类型除了一个u1类型tag之外，还有一个u2类型的index，如下图：



t	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D	Ê
10	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04	c
20	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	
30	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69	e
40	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	r

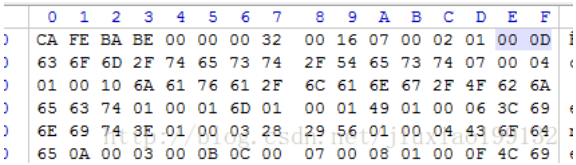
发现图中值是0002，表示指向第二个常量的索引。继续看第二个常量tag是01，如图：



	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D
	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04
	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A
	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69
	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64

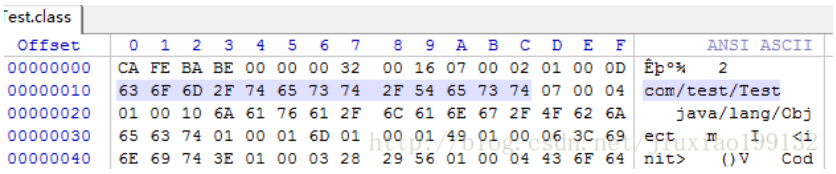
对照总表发现是CONSTANT_Utf8_info类型，这个类型由tag（u1），length（u2）和bytes（u1）组成。

那么跟着tag后面的length是两个字节，如图：



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
)	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D
)	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04
)	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A
)	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69
)	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64
)	65	0A	00	03	00	0B	0C	00	07	00	08	01	00	0F	4C	69

可以看到是0000D，转成十进制也就是13，即接下来的13个字节为一个utf-8缩略编码的字符串。如图：



test.class

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	0D	Êþº¾ 2
00000010	63	6F	6D	2F	74	65	73	74	2F	54	65	73	74	07	00	04	com/test/Test
00000020	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	java/lang/Obj
00000030	65	63	74	01	00	01	6D	01	00	01	49	01	00	06	3C	69	ect m I <i
00000040	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F	64	nit> ()V Cod

从图中右边ASCII可以看到对应的com/test/Test。
逐个分析是比较麻烦的，可以使用JDK自带的用于分析Class文件字节码的工具javap(在JDK的bin目录下)：

```
d:\>javap -verbose Test
Compiled from "Test.java"
public class com.test.Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 49
  Constant pool:
const #1 = class         #2;     //  com/test/Test
const #2 = Asciz         com/test/Test;
 const #3 = class         #4;     //  java/lang/Object
 const #4 = Asciz         java/lang/Object;
 const #5 = Asciz         m;
 const #6 = Asciz         I;
 const #7 = Asciz         <init>;
 const #8 = Asciz         ()V;
 const #9 = Asciz         Code;
 const #10 = Method       #3.#11; //  java/lang/Object."<init>":()V
 const #11 = NameAndType #7:#8;//  "<init>":()V
 const #12 = Asciz        LineNumberTable;
 const #13 = Asciz        LocalVariableTable;
 const #14 = Asciz        this;
 const #15 = Asciz        Lcom/test/Test;;
 const #16 = Asciz        getM;
 const #17 = Asciz        ()I;
 const #18 = Field        #1.#19; //  com/test/Test.m:I
 const #19 = NameAndType #5:#6;//  m:I
 const #20 = Asciz        SourceFile;
 const #21 = Asciz        Test.java;

{
public com.test.Test();
......
```

这里可以看到第一个常量是class， 后面的 #2 表示指向 第二个常量。
省略了显示结果的后半部分，这里可以看到总共有21个常量，并且可以看到常量的类型，如果常量中保存的为索引值（#），也会提示索引
当然其中也包含了很多特殊的符号（如：()V），这些将会在后面的“六，字段表集合”与“七，方法表集合”中进行说明

访问标志（access_flags）

```
000000B0  00 06 01 00 0A 53 6F 75  72 63 65 46 69 6C 65 01        SourceFile
000000C0  00 09 54 65 73 74 2E 6A  61 76 61 00 21 00 01 00    Test.java !
```

接下来2个字节代表访问标志位。这个标志用于识别类或接口层次的访问信息，如：这个Class是类还是接口，是否定义为public类型，是否

标志名称	标志值	含义
ACC_PUBLIC	0x0001	是否为public类型
ACC_FINAL	0x0010	是否被声明为final，只有类可设置
ACC_SUPER	0x0020	是否允许使用invokespecial字节码指令，JDK1.2以后编译出来的类这个标志为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为abstract类型，对于接口和抽象类，此标志为真，其它类为假
ACC_SYNTHETIC	0x1000	标识别这个类并非由用户代码产生
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举

根据上面的表格，测试类的访问标志为ACC_PUBLIC | ACC_SUPER = 0x0001 | 0x0020 =1 | 32 = [00000000][00000001] | [00000000][0
33 = 0x0021

类索引、父类索引与接口索引集合（this_class 、 super_class 、 interfaces_count 和 interfaces

```
000000c0  00 09 54 65 73 74 2E 6A  61 76 61 00 21 00 01 00    ..Test.java.!...
000000d0  03 00 00 00 01 00 02 00  05 00 06 00 00 00 02 00    ................
```

Class文件中由这3项数据来确定这个类的继承关系
this_class：类索引，用于确定这个类的全限定名，占2字节
super_class：父类索引，用于确定这个类父类的全限定名（Java语言不允许多重继承，故父类索引只有一个。除了java.lang.Object类之外
java.lang.Object类之外，所有类该字段值都不为0），占2字节
interfaces_count：接口索引计数器，占2字节。如果该类没有实现任何接口，则该计数器值为0，并且后面的接口的索引集合将不占用任何
interfaces：接口索引集合，一组u2类型数据的集合。用来描述这个类实现了哪些接口，这些被实现的接口将按implements语句（如果该类
接口顺序从左至右排列在接口的索引集合中
this_class、super_class与interfaces中保存的索引值均指向常量池中一个CONSTANT_Class_info类型的常量，通过这个常量中保存的索引值
类型的常量中的全限定名字符串
this_class的值为0x0001，即常量池中第一个常量，super_class的值为0x0003，即常量池中的第三个常量，interfaces_counts的值为0x00

```
1  const #1 = class        #2;     //  com/test/Test
2  const #2 = Asciz        com/test/Test;
3  const #3 = class        #4;     //  java/lang/Object
4  const #4 = Asciz        java/lang/Object;
```

可以看到测试类的全限定名为”com/test/Test;”，测试类的父类的全限定名为”java/lang/Object;”

字段表集合（fields_count 和 fields）

```
000000d0  03 00 00 00 01 00 02 00  05 00 06 00 00 00 02 00    ................
```

fields_count：字段表计数器，即字段表集合中的字段表数据个数。占2字节，其值为0x0001，即只有一个字段表数据，也就是测试类中只
fields：字段表集合，一组字段表类型数据的集合。字段表用于描述接口或类中声明的变量，包括类级别（static）和实例级别变量，不包括
在Java中一般通过如下几项描述一个字段：字段作用域（public、protected、private修饰符）、是类级别变量还是实例级别变量（static修
可见性（volatile修饰符）、可序列化与否（transient修饰符）、字段数据类型（基本类型、对象、数组）以及字段名称。在字段表中，变量
型和字段名称则引用常量池中常量表示，字段表格式如下表所示：

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

字段修饰符放在access_flags中，占2字节，其值为0x0002，可见这个字段由private修饰，与访问标志位十分相似

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为public
ACC_PRIVATE	0x0002	字段是否为private
ACC_PROTECTED	0x0004	字段是否为protected
ACC_STATIC	0x0008	字段是否为static
ACC_FINAL	0x0010	字段是否为final
ACC_VOLATILE	0x0040	字段是否为volatile
ACC_TRANSIENT	0x0080	字段是否为transient
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生
ACC_ENUM	0x4000	字段是否为enum

当然实际上，ACC_PUBLIC、ACC_PRIVATE和ACC_PROTECTED这3个标志只能选择一个，接口中的字段必须有ACC_PUBLIC、ACC_STATIC
无规定，这些都是java语言所要求的
name_index代表字段的简单名称（参见备注二），占2字节，是一个对常量池的引用，其值为0x0005，即常量池中第5个常量
descriptor_index代表代表参数的描述符（参见备注三），占2个字节，是一个对常量池的引用，其值为0x0006，即常量池中第6个常量

```
const #5 = Asciz        m;
const #6 = Asciz        I;
```

综上，可以推断出源代码定义的字段为（和测试类完全一样）：

```
private int m;
```

字段表包含的固定数据项到descriptor_index结束，之后跟随一个属性表集合用于存储一些附加信息：attributes_count（属性计数器，占2字
要描述的信息）和attributes（属性表集合，详细说明见后面“八，属性表集合”）
字段表集合中不会列出从父类或父接口中继承的字段，但是可能列出原本Java代码之中不存在的字段，如：内部类为了保持对外部类的访问
Java语言中字段是不能重载的，2个字段无论数据类型、修饰符是否相同，都不能使用相同的名称；但是对于字节码，只要字段描述符不同，

方法表集合（methods_count 和 methods）

```
000000d0  03 00 00 00 01 00 02 00  05 00 06 00 00 00 02 00  ................
```

methods_count：方法表计数器，即方法表集合中的方法表数据个数。占2字节，其值为0x0002，即测试类中有2个方法
methods：方法表集合，一组方法表类型数据的集合。方法表结构和字段表结构一样：

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

数据项的含义非常相似，仅在访问标志位和属性表集合中的可选项上有略微不同
由于ACC_VOLATILE标志和ACC_TRANSIENT标志不能修饰方法，所以access_flags中不包含这两项，同时增加ACC_SYNCHRONIZED标志、
志和ACC_ABSTRACT标志

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为public
ACC_PRIVATE	0x0002	字段是否为private
ACC_PROTECTED	0x0004	字段是否为protected
ACC_STATIC	0x0008	字段是否为static
ACC_FINAL	0x0010	字段是否为final
ACC_SYNCHRONIZED	0x0020	字段是否为synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	字段是否为native
ACC_ABSTRACT	0x0400	字段是否为abstract
ACC_STRICTFP	0x0800	字段是否为strictfp
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生

第一个方法（由编译器自动添加的默认构造方法）：

```
000000d0  03 00 00 00 01 00 02 00  05 00 06 00 00 00 02 00  ................
000000e0  01 00 07 00 08 00 01 00  09 00 00 00 2F 00 01 00  ............/...
```

access_flags为0x0001，即public；name_index为0x0007，即常量池中第7个常量；descriptor_index为0x0008，即常量池中第8个常量

```
1  const #7 = Asciz        <init>;
2  const #8 = Asciz        ()V;
```

```
000000e0  01 00 07 00 08 00 01 00  09 00 00 00 2F 00 01 00  ............/...
000000f0  01 00 00 00 05 2A B7 00  0A B1 00 00 00 02 00 0C  .....*..........
```

接下来2个字节为属性计数器，其值为0x0001，说明这个方法的属性表集合中有一个属性（详细说明见后面“八，属性表集合”），属性名称
9个常量：Code。接下来4位为0x0000002F，表示Code属性值的字节长度为47。接下来2位为0x0001，表示该方法的操作数栈的深度最大
示该方法的局部变量占用空间为1。接下来4位为0x0000005，则紧接着的5个字节0x2AB7000AB1为该方法编译后生成的字节码指令（各字
字节码指令表）。接下来2个字节为0x0000，说明Code属性异常表集合为空。

```
000000f0  01 00 00 00 05 2A B7 00  0A B1 00 00 00 02 00 0C  .....*..........
00000100  00 00 00 06 00 01 00 00  00 03 00 0D 00 00 00 0C  ................
```

接下来2个字节为0x0002，说明Code属性带有2个属性，那么接下来2位0x000C即为Code属性第一个属性的属性名称，指向常量池中第12
位为0x00000006，表示LineNumberTable属性值所占字节长度为6。接下来2位为0x0001，即该line_number_table中只有一个line_number
line_number为0x0003，LineNumberTable属性结束。

```
00000100  00 00 00 06 00 01 00 00  00 03 00 0D 00 00 00 0C  ................
00000110  00 01 00 00 00 05 00 0E  00 0F 00 00 00 01 00 10  ................
```

接下来2位0x000D为Code属性第二个属性的属性名，指向常量池中第13个常量：LocalVariableTable。该属性值所占的字节长度为0x0000C
local_variable_table中只有一个local_variable_info表，按照local_variable_info表结构，start_pc为0x0000，length为0x0005，name_index
量：this，descriptor_index为0x000F，指向常量池中第15个常量：Lcom/test/Test;，index为0x0000。第一个方法结束
第二个方法：

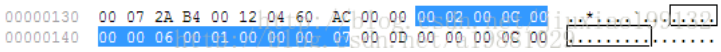
```
00000110  00 01 00 00 00 05 00 0E  00 0F 00 00 00 01 00 10  ................
00000120  00 11 00 01 00 09 00 00  00 31 00 02 00 01 00 00  .........1......
```

access_flags为0x0001，即public；name_index为0x0010，即常量池中第16个常量；descriptor_index为0x0011，即常量池中第17个常量

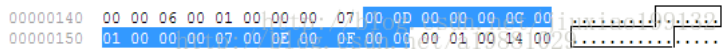
```
1  const #16 = Asciz        getM;
2  const #17 = Asciz        ()I;
```

```
00000120  00 11 00 01 00 09 00 00  00 31 00 02 00 01 00 00  .........1......
00000130  00 07 2A B4 00 12 04 60  AC 00 00 00 02 00 0C 00  ..*....`........
```

接下来2个字节为属性计数器，其值为0x0001，说明这个方法有一个方法属性，属性名称为接下来2位0x0009，指向常量池中第9个常量：
示Code属性值的字节长度为49。接下来2位为0x0002，表示该方法的操作数栈的深度最大值为2。接下来2位为0x0001，表示该方法的局部
0x0000007，则紧接着的7个字节0x2AB400120460AC为该方法编译后生成的字节码指令。接下来2个字节为0x0000，说明Code属性异常



接下来2个字节为0x0002，说明Code属性带有2个属性，那么接下来2位0x000C即为Code属性第一个属性的属性名称，指向常量池中第12
位为0x00000006，表示LineNumberTable属性值所占字节长度为6。接下来2位为0x0001，即该line_number_table中只有一个line_number
line_number为0x0007，LineNumberTable属性结束。



和第一个方法的LocalVariableTable属性基本相同，唯一的区别是局部变量this的作用范围覆盖的长度为7而不是5，第二个方法结束
如果子类没有重写父类的方法，方法表集合中就不会出现父类方法的信息；有可能会出现由编译器自动添加的方法（如：，实例类构造器）
在Java语言中，重载一个方法除了要求和原方法拥有相同的简单名称外，还要求必须拥有一个与原方法不同的特征签名（方法参数集合），
言中不能仅仅依靠返回值的不同对一个已有的方法重载；但是在Class文件格式中，特征签名即为方法描述符，只要是描述符不完全相同的2
回值不同之外完全相同的方法在Class文件中也可以合法共存
javap工具在后半部分会列出分析完成的方法（可以看到和我们的分析结果是一样的）：

```
d:\>javap -verbose Test
......
{
public com.test.Test();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #10; //Method java/lang/Object."<init>":()V
   4:   return
  LineNumberTable:
   line 3: 0

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      5      0    this       Lcom/test/Test;

public int getM();
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   aload_0
   1:   getfield        #18; //Field m:I
   4:   iconst_1
   5:   iadd
   6:   ireturn
  LineNumberTable:
   line 7: 0

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      7      0    this       Lcom/test/Test;
}
```

属性表集合（attributes_count 和 attributes）

在Class文件、属性表、方法表中都可以包含自己的属性表集合，用于描述某些场景的专有信息
与Class文件中其它数据项对长度、顺序、格式的严格要求不同，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
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，为了能正确解析Class文件，虚拟机规范中预
属性：

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

每种属性均有各自的表结构。这9种表结构有一个共同的特点，即均由一个u2类型的属性名称开始，可以通过这个属性名称来判段属性的类
Code属性：Java程序方法体中的代码经过Javac编译器处理后，最终变为字节码指令存储在Code属性中。当然不是所有的方法都必须有这
存在Code属性），Code属性表结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

max_stack：操作数栈深度最大值，在方法执行的任何时刻，操作数栈深度都不会超过这个值。虚拟机运行时根据这个值来分配栈帧的操作
max_locals：局部变量表所需存储空间，单位为Slot。并不是所有局部变量占用的Slot之和，当一个局部变量的生命周期结束后，其所占用
量使用，按此方式计算出方法运行时局部变量表所需的存储空间
code_length和code：用来存放Java源程序编译后生成的字节码指令。code_length代表字节码长度，code是用于存储字节码指令的一系列
每一个指令是一个u1类型的单字节，当虚拟机读到code中的一个字节码（一个字节能表示256种指令，Java虚拟机规范定义了其中约200个
节码代表的指令，指令后面是否带有参数，参数该如何解释，虽然code_length占4个字节，但是Java虚拟机规范中限制一个方法不能超过6
将拒绝编译
ConstantValue属性：通知虚拟机自动为静态变量赋值，只有被static关键字修饰的变量（类变量）才可以使用这项属性。其结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

可以看出ConstantValue属性是一个定长属性，其中attribute_length的值固定为0x00000002，constantvalue_index为一常量池字面量类型
中只有与基本类型和字符串类型相对应的字面量常量，所以ConstantValue属性只支持基本类型和字符串类型）
对非static类型变量（实例变量，如：int a = 123;）的赋值是在实例构造器方法中进行的
对类变量（如：static int a = 123;）的赋值有2种选择，在类构造器方法中或使用ConstantValue属性。当前Javac编译器的选择是：如果变
只要求有ConstantValue属性的字段必须设置ACC_STATIC标志，对final关键字的要求是Javac编译器自己加入的要求），并且该变量的数据
ConstantValue属性进行初始化；否则在类构造器方法中进行初始化
Exceptions属性：列举出方法中可能抛出的受查异常（即方法描述时throws关键字后列出的异常），与Code属性平级，与Code属性包含的

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

number_of_exceptions表示可能抛出number_of_exceptions种受查异常
exception_index_table为异常索引集合，一组u2类型exception_index的集合，每一个exception_index为一个指向常量池中一CONSTANT_C
常的类型
InnerClasses属性：该属性用于记录内部类和宿主类之间的关系。如果一个类中定义了内部类，编译器将会为这个类与这个类包含的内部类

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	inner_classes	number_of_classes

inner_classes为内部类表集合，一组内部类表类型数据的集合，number_of_classes即为集合中内部类表类型数据的个数
每一个内部类的信息都由一个inner_classes_info表来描述，inner_classes_info表结构如下：

类型	名称	数量
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_name_access_flags	1

inner_class_info_index和outer_class_info_index指向常量池中CONSTANT_Class_info类型常量索引，该CONSTANT_Class_info类型常量指向
量，分别为内部类的全限定名和宿主类的全限定名
inner_name_index指向常量池中CONSTANT_Utf8_info类型常量的索引，为内部类名称，如果为匿名内部类，则该值为0
inner_name_access_flags类似于access_flags，是内部类的访问标志

标志名称	标志值	含义
ACC_PUBLIC	0x0001	内部类是否为public
ACC_PRIVATE	0x0002	内部类是否为private
ACC_PROTECTED	0x0004	内部类是否为protected
ACC_STATIC	0x0008	内部类是否为static
ACC_FINAL	0x0010	内部类是否为final
ACC_INTERFACE	0x0020	内部类是否为一个接口
ACC_ABSTRACT	0x0400	内部类是否为abstract
ACC_SYNTHETIC	0x1000	内部类是否为编译器自动产生
ACC_ANNOTATION	0x4000	内部类是否是一个注解
ACC_ENUM	0x4000	内部类是否是一个枚举

LineNumberTale属性：用于描述Java源码的行号与字节码行号之间的对应关系，非运行时必需属性，会默认生成至Class文件中，可以使用
生成该项属性信息，其结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

line_number_table是一组line_number_info类型数据的集合，其所包含的line_number_info类型数据的数量为line_number_table_length，line

类型	名称	数量	说明
u2	start_pc	1	字节码行号
u2	line_number	1	Java源码行号

不生成该属性的最大影响是：1，抛出异常时，堆栈将不会显示出错的行号；2，调试程序时无法按照源码设置断点

LocalVariableTable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，非运行时必需属性，默认不会生成至g:none或-g:vars关闭或要求生成该项属性信息，其结构如下：

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

local_variable_table是一组local_variable_info类型数据的集合，其所包含的local_variable_info类型数据的数量为local_variable_table_length

类型	名称	数量	说明
u2	start_pc	1	局部变量的生命周期开始的字节码偏移量
u2	length	1	局部变量作用范围覆盖的长度
u2	name_index	1	指向常量池中CONSTANT_Utf8_info类型常量的索引，局部变量名称
u2	descriptor_index	1	指向常量池中CONSTANT_Utf8_info类型常量的索引，局部变量描述符
u2	index	1	局部变量在栈帧局部变量表中Slot的位置，如果这个变量的数据类型为64位类型（long或double），它占用的Slot为index和index+1这2个位置

start_pc + length即为该局部变量在字节码中的作用域范围

不生成该属性的最大影响是：1，当其他人引用这个方法时，所有的参数名称都将丢失，IDE可能会使用诸如arg0、arg1之类的占位符代替原给代码的编写带来不便；2，调试时调试器无法根据参数名称从运行上下文中获取参数值

SourceFile属性：用于记录生成这个Class文件的源码文件名称，为可选项，可以使用Javac的-g:none或-g:source关闭或要求生成该项属性

型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

可以看出SourceFile属性是一个定长属性，sourcefile_index是指向常量池中一CONSTANT_Utf8_info类型常量的索引，常量的值为源码文件名对大多数文件，类名和文件名是一致的，少数特殊类除外（如：内部类），此时如果不生成这项属性，当抛出异常时，堆栈中将不会显示出

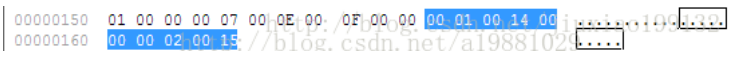
Deprecated属性和Synthetic属性：这两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念

Deprecated属性表示某个类、字段或方法已经被程序作者定为不再推荐使用，可在代码中使用@Deprecated注解进行设置

Synthetic属性表示该字段或方法不是由Java源码直接产生的，而是由编译器自行添加的（当然也可设置访问标志中的ACC_SYNTHETIC标志和字段都应当至少设置Synthetic属性和ACC_SYNTHETIC标志位中的一项，唯一的例外是实例构造器和类构造器方法）

这两项属性的结构为(当然attribute_length的值必须为0x00000000)：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1



起始2位为0x0001，说明有一个类属性。接下来2位为属性的名称，0x0014，指向常量池中第20个常量：SourceFile。接下来4位为0x0000 2个字节为0x0014，指向常量池中第21个常量：Test.java，即这个Class文件的源码文件名为Test.java