认识Class文件

Class文件是一组以8字节为基础单位的二进制流，各个数据项目严格的按照顺序紧凑地排列在文件之中，中间没有添加任何分割符，这使得整个Class文件存储的内容几乎全部是程序运行的必要数据，没有空隙存在。

《Java虚拟机规范》规定了Class文件格式采用一种类似C语言结构体的伪结构来存储数据，这种伪结构只包含两种数据类型，即无符号数和表。

class文件通过固定的数据结构排列顺序并且每种数据结构指定了占用的字节长度来紧凑的在组成了完整的可读文件，jvm只需要从文件开始的地方一步一步的读取能够完全的解析出这个类文件的内容。

数据类型：u1 u2 u4 u8 和 _info(表类型)

u——unsigned 无符号 ，u 后面的数字表示字节数

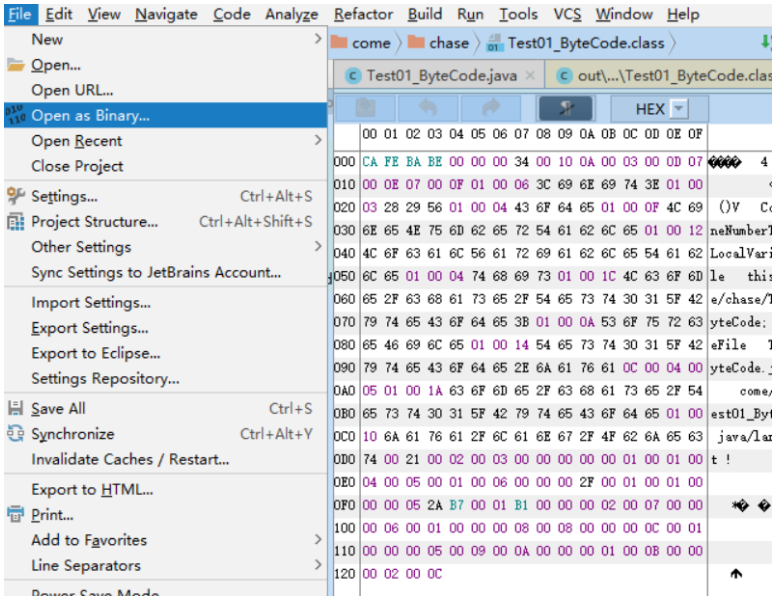
ClassFile 查看方式

可以借助很多工具或者插件查看，比如： sublime/notepad等

IDEA 插件 - BinEd

在Idea界面中，按照如下步骤：File → Open as Binary 就可以查看Class文件对应的十六进制内容

可以选择不同的进制查看方式（2/8/10/16进制）



如下为一个class文件的16进制内容

```
CA FE BA BE 00 00 00 34 00 10 0A 00 03 00 0D 07
00 0E 07 00 0F 01 00 06 3C 69 6E 69 74 3E 01 00
03 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C 69
6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 12
4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 62
6C 65 01 00 04 74 68 69 73 01 00 1C 4C 63 6F 6D
65 2F 63 68 61 73 65 2F 54 65 73 74 30 31 5F 42
79 74 65 43 6F 64 65 3B 01 00 0A 53 6F 75 72 63
65 46 69 6C 65 01 00 14 54 65 73 74 30 31 5F 42
79 74 65 43 6F 64 65 2E 6A 61 76 61 0C 00 04 00
05 01 00 1A 63 6F 6D 65 2F 63 68 61 73 65 2F 54
65 73 74 30 31 5F 42 79 74 65 43 6F 64 65 01 00
10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63
74 00 21 00 02 00 03 00 00 00 00 00 01 00 01 00
04 00 05 00 01 00 06 00 00 00 2F 00 01 00 01 00
00 00 05 2A B7 00 01 B1 00 00 00 02 00 07 00 00
00 06 00 01 00 00 00 08 00 08 00 00 00 0C 00 01
00 00 00 05 00 09 00 0A 00 00 00 01 00 0B 00 00
00 02 00 0C
```

解释：

在该文件中，每个字节都有各自的含义

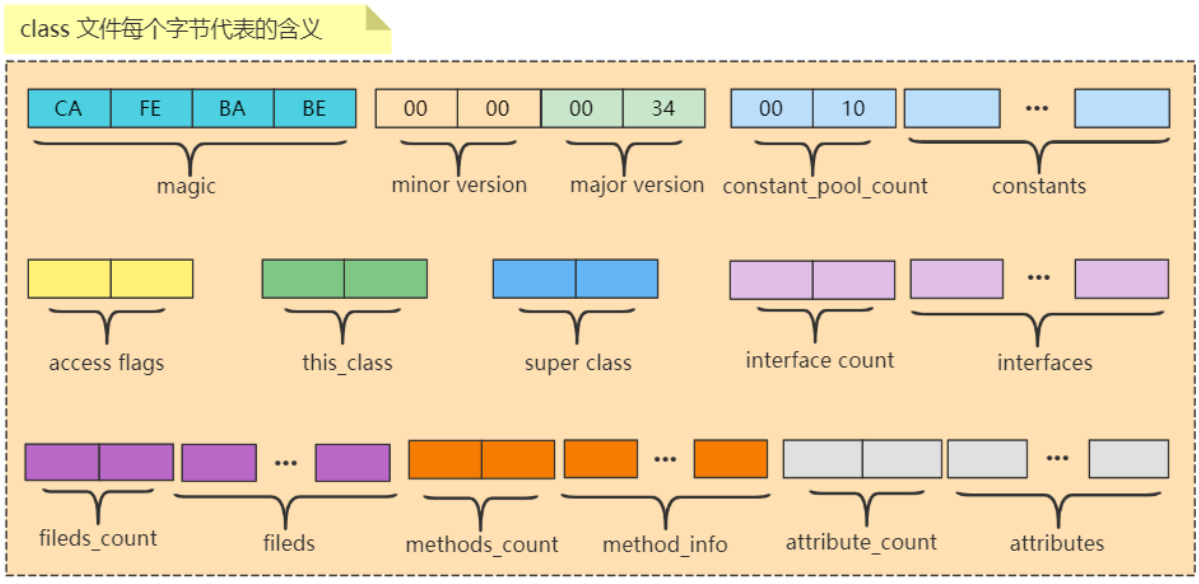
如：CA FE BA BE　文件类型(class 类型)四个字节

00 00 小版本号　00 34　大版本号

34转换成10进制是 52
jdk 1.8 的版本默认是　52.0

00 10　常量池个数

如下是一个Class文件所表示的含义



图中具体的含义会在后面的案例中解释

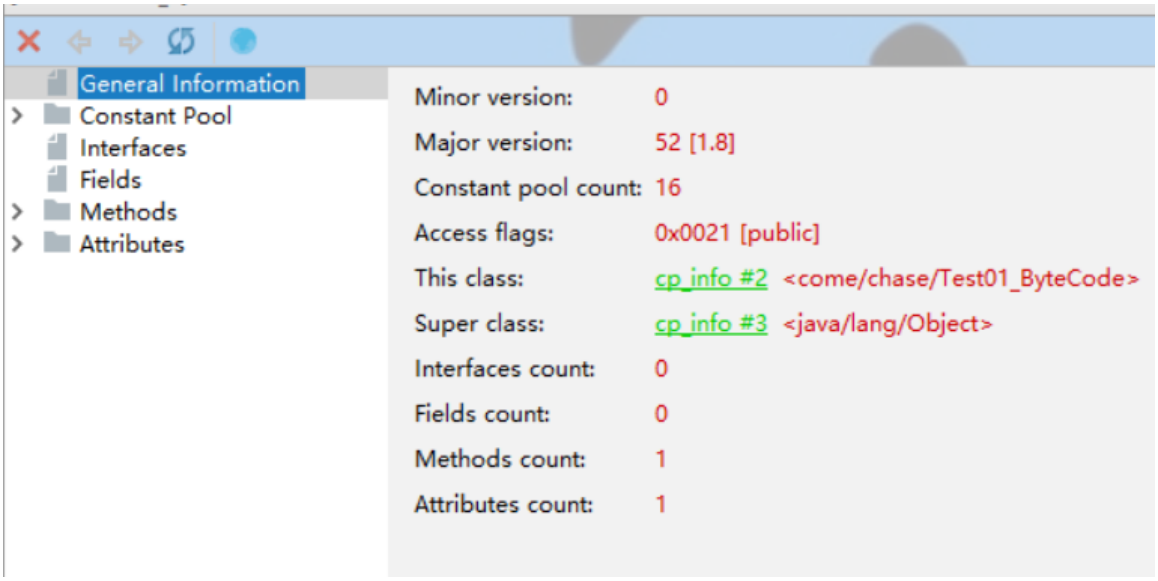
查看ByteCode的方法

javap —— jdk 自带

JClassLib - IDEA插件（看着方便）

安装插件——鼠标光标放在类体 —— View —— Show Bytecode With jclasslib 会出现如下

General Information 详细信息



Constant pool count：常量池个数 16

Access flags： 0×0021 = ACC_PUBLIC & ACC_SUPER

ACC_PUBLIC 0×0001 是否为public类型

ACC_SUPER 0×0020 该标记必须为true，jdk1.0.2之后编译出的内容必须为真

This class： 当前类的位置，在 Constant Pool（第一个编号是1） 的 2号位置

Super class： 父类的位置，在Constant Pool 的第3号位置

Interface count: 实现了多少个接口

Fields count：属性的个数

Methods count：方法个数

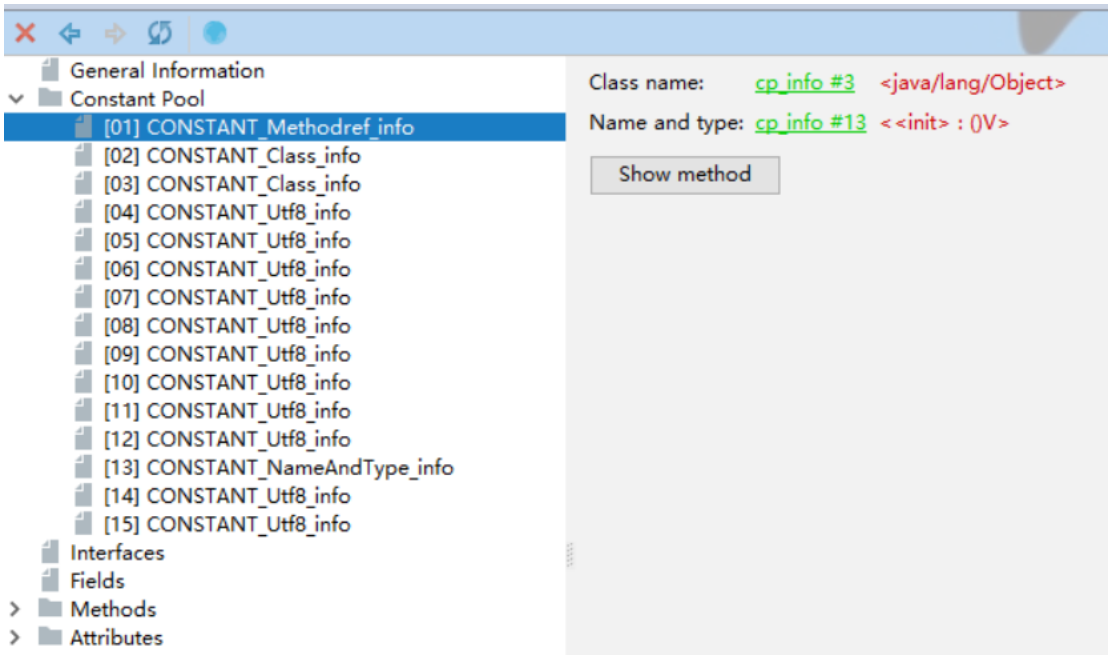
Attributes count：附加属性个数

Constant Pool 详细信息

0 号是预留的，没有任何引用指向这个位置，是从1号开始 的

个数 = constant pool count - 1 = 15 个

CONSTANT_Methodref_info



Class Name 指向 Constant pool的 3号位置

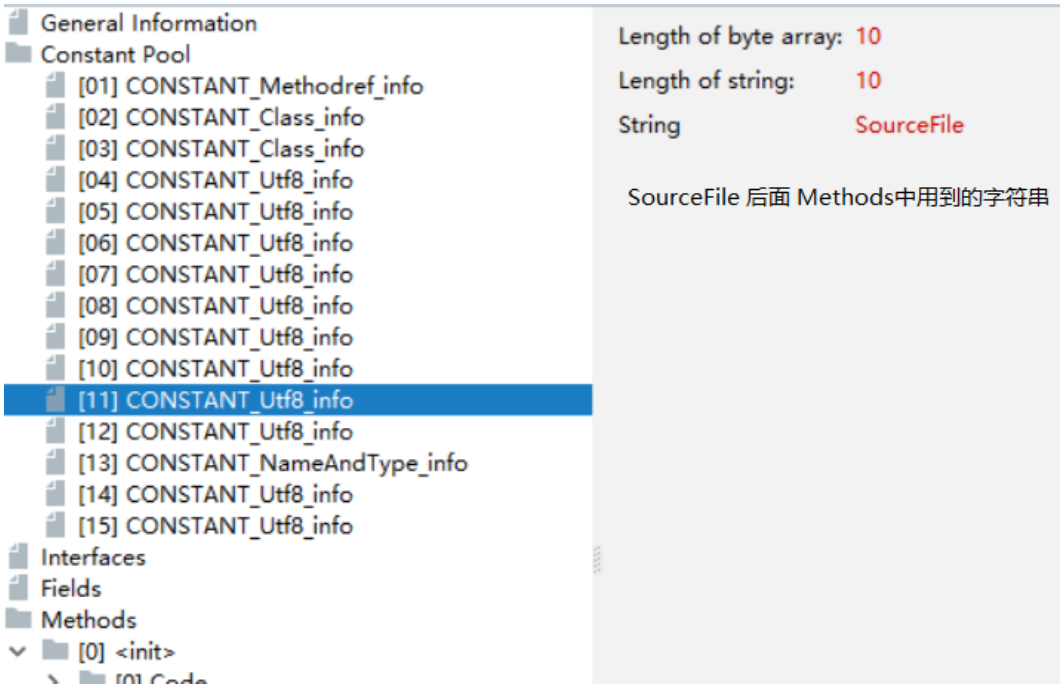
CONSTANT_NameAndType

Name and type 指向 13 位置

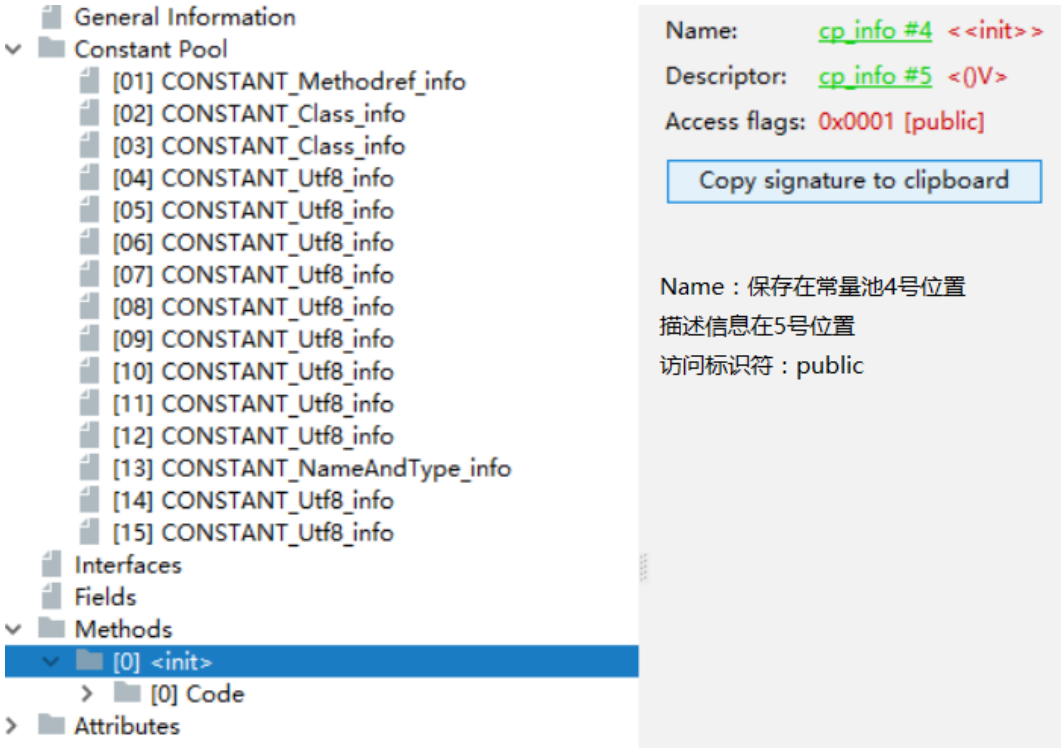
13 位置的的内容如下：

CONSTANT_Utf8_info

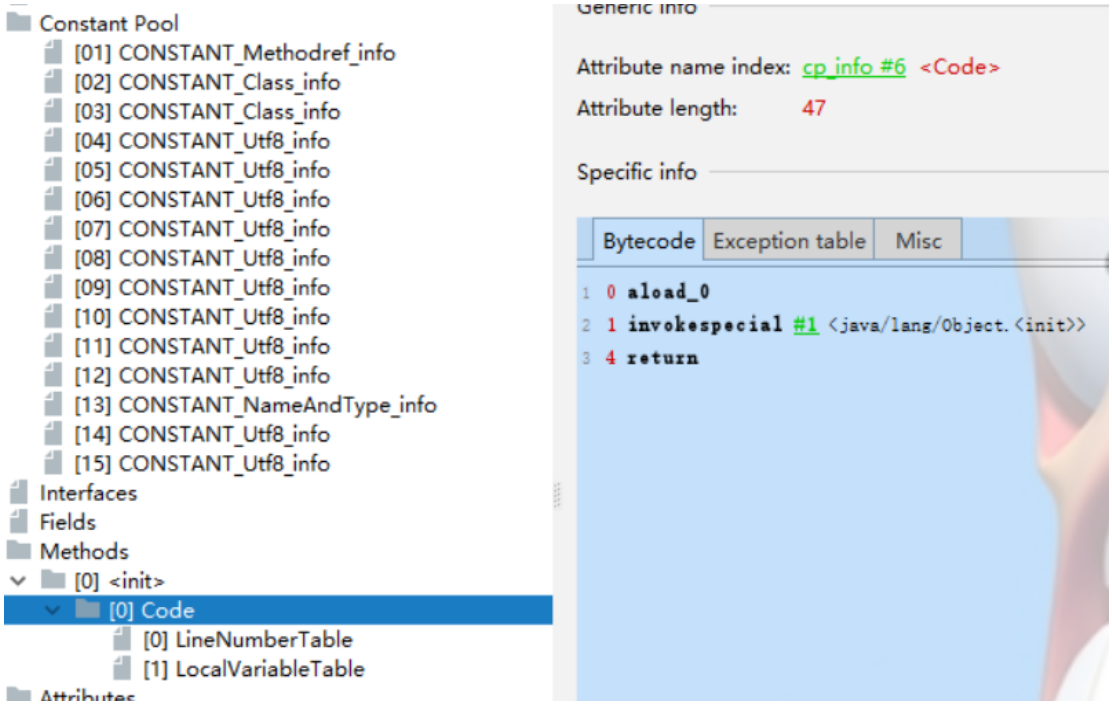
utf8 标识字符串，一般存的是别的模块用到的字符串常量



Methods



Methods - Code



Code 里面的东西标识方法的实现代码

16进制的Class文件中标识方法的 字节，每一个数字对应一条 java的汇编指令

比如 16进制中 数字 2a 对应的汇编指令时 aload_0

如下是部分 十六进制数——汇编指令的 字典

每一条汇编质量代表的意思都可以到对应文档中查找意思

文档地址：
https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf

Constants			Loads			Stores		
00	(0x00)	*nop*	21	(0x15)	*iload*	54	(0x36)	*istore*
01	(0x01)	*aconst_null*	22	(0x16)	*lload*	55	(0x37)	*lstore*
02	(0x02)	*iconst_m1*	23	(0x17)	*fload*	56	(0x38)	*fstore*
03	(0x03)	*iconst_0*	24	(0x18)	*dload*	57	(0x39)	*dstore*
04	(0x04)	*iconst_1*	25	(0x19)	*aload*	58	(0x3a)	*astore*
05	(0x05)	*iconst_2*	26	(0x1a)	*iload_0*	59	(0x3b)	*istore_0*
06	(0x06)	*iconst_3*	27	(0x1b)	*iload_1*	60	(0x3c)	*istore_1*
07	(0x07)	*iconst_4*	28	(0x1c)	*iload_2*	61	(0x3d)	*istore_2*
08	(0x08)	*iconst_5*	29	(0x1d)	*iload_3*	62	(0x3e)	*istore_3*
09	(0x09)	*lconst_0*	30	(0x1e)	*lload_0*	63	(0x3f)	*lstore_0*
10	(0x0a)	*lconst_1*	31	(0x1f)	*lload_1*	64	(0x40)	*lstore_1*
11	(0x0b)	*fconst_0*	32	(0x20)	*lload_2*	65	(0x41)	*lstore_2*
12	(0x0c)	*fconst_1*	33	(0x21)	*lload_3*	66	(0x42)	*lstore_3*
13	(0x0d)	*fconst_2*	34	(0x22)	*fload_0*	67	(0x43)	*fstore_0*
14	(0x0e)	*dconst_0*	35	(0x23)	*fload_1*	68	(0x44)	*fstore_1*
15	(0x0f)	*dconst_1*	36	(0x24)	*fload_2*	69	(0x45)	*fstore_2*
16	(0x10)	*bipush*	37	(0x25)	*fload_3*	70	(0x46)	*fstore_3*
17	(0x11)	*sipush*	38	(0x26)	*dload_0*	71	(0x47)	*dstore_0*
18	(0x12)	*ldc*	39	(0x27)	*dload_1*	72	(0x48)	*dstore_1*
19	(0x13)	*ldc_w*	40	(0x28)	*dload_2*	73	(0x49)	*dstore_2*
20	(0x14)	*ldc2_w*	41	(0x29)	*dload_3*	74	(0x4a)	*dstore_3*
			42	(0x2a)	*aload_0*	75	(0x4b)	*astore_0*
			43	(0x2b)	*aload_1*	76	(0x4c)	*astore_1*
			44	(0x2c)	*aload_2*	77	(0x4d)	*astore_2*
			45	(0x2d)	*aload_3*	78	(0x4e)	*astore_3*
			46	(0x2e)	*iaload*	79	(0x4f)	*iastore*
			47	(0x2f)	*laload*	80	(0x50)	*lastore*
			48	(0x30)	*faload*	81	(0x51)	*fastore*
			49	(0x31)	*daload*	82	(0x52)	*dastore*
			50	(0x32)	*aaload*	83	(0x53)	*aastore*
			51	(0x33)	*baload*	84	(0x54)	*bastore*
			52	(0x34)	*caload*	85	(0x55)	*castore*
			53	(0x35)	*saload*	86	(0x56)	*sastore*

Operation Load `reference` from local variable

Format *aload_<n>*

Forms
aload_0 = 42 (0x2a)
aload_1 = 43 (0x2b)
aload_2 = 44 (0x2c)
aload_3 = 45 (0x2d)

Operand Stack ... →
..., *objectref*

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain a `reference`. The *objectref* in the local variable at <*n*> is pushed onto the operand stack.

Notes An *aload_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction (§*astore_<n>*) is intentional.

Each of the *aload_<n>* instructions is the same as *aload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

总结
JVM在执行一个方法的时候，会从字节码中读取一条指令 如 2a，则再查找对应的汇编指令，即 aload_0，做相应的操作，接下来再读下一条指令 b7，查找到对应的汇编指令 invokespecial （调用构造），一直进行类似的操作到结束 到b1指令——return

2ab7 0001 b1
这5个字节可以表示一个构造方法的调用过程

2a：aload_0 表示把 this 压栈 扔进去
b7：invokespecial 需要两个参数 00 和 01
01 代表常量池中的第一项java.lang.Object 中的构造方法
即 默认构造调用的是父类Object的构造方法
b1：return