我们知道，一个java文件被编译后，会变成另一个class文件。

那你有没有观察过这个class文件里面长什么样子呢？

我们来试一下，以以下java文件为例：

|  |
| --- |
| package com.peng.jvm.c1\_bytecode;  public class T0100\_ByteCode01 {  } |

如果我们通过idea输出的class文件，会看到：

|  |
| --- |
| //  // Source code recreated from a .class file by IntelliJ IDEA  // (powered by Fernflower decompiler)  //  package com.peng.jvm.c1\_bytecode;  public class T0100\_ByteCode01 {  public T0100\_ByteCode01() {  }  } |

emmm，自动多了一个类的无参构造方法，当然这也没什么奇怪的，因为一个类在没有指定构造方法的时候，其实会自动生成一个无参构造。

但实际此时看到的class文件也还是代码形式。

但实际上，这是idea帮我们自己进行了反编译而已，真正的class文件肯定不是这样的。

**原生class文件部分解读**

我们可以通过sublime或者notepad通过16进制格式来查看class的内容，也可以通过idea的插件BinEd来查看。

这里我们使用sublime直接去打开这个class文件看一看：

|  |
| --- |
| cafe babe 0000 0034 0010 0a00 0300 0d07  000e 0700 0f01 0006 3c69 6e69 743e 0100  0328 2956 0100 0443 6f64 6501 000f 4c69  6e65 4e75 6d62 6572 5461 626c 6501 0012  4c6f 6361 6c56 6172 6961 626c 6554 6162  6c65 0100 0474 6869 7301 0030 4c63 6f6d  2f6d 6173 6869 6269 6e67 2f6a 766d 2f63  315f 6279 7465 636f 6465 2f54 3031 3030  5f42 7974 6543 6f64 6530 313b 0100 0a53  6f75 7263 6546 696c 6501 0015 5430 3130  305f 4279 7465 436f 6465 3031 2e6a 6176  610c 0004 0005 0100 2e63 6f6d 2f6d 6173  6869 6269 6e67 2f6a 766d 2f63 315f 6279  7465 636f 6465 2f54 3031 3030 5f42 7974  6543 6f64 6530 3101 0010 6a61 7661 2f6c  616e 672f 4f62 6a65 6374 0021 0002 0003  0000 0000 0001 0001 0004 0005 0001 0006  0000 002f 0001 0001 0000 0005 2ab7 0001  b100 0000 0200 0700 0000 0600 0100 0000  0300 0800 0000 0c00 0100 0000 0500 0900  0a00 0000 0100 0b00 0000 0200 0c |

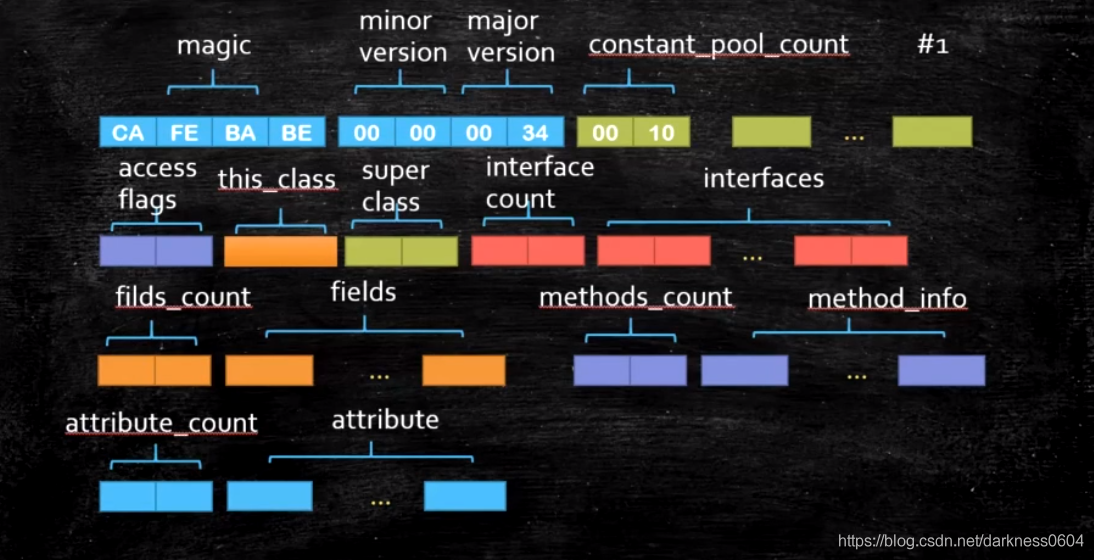
恩，有那味了。。。

映入眼帘的就是一堆16进制的数字，而这，就是我们class文件的真实模样。。

我们来解读一下这个class文件：

我们也就只能看懂开头有2个单词，分别是 cafe babe，而这其实也是一个class文件的标识。

我们知道，1个字节由8位二进制组成，而1位16进制的数代表4位2进制的数，那也就是说，这里面的2位就是一个字节。



这里选取了前面部分的位数进行说明，我们以字节为单位进行描述。

首先前四个字节 **CA FE BA BE** ，代表的是class文件中的magic number，可以理解是代表的是由java编译出的class文件的标识。

接下来的两个字节 **00 00** 代表的是minor version，指的是class文件的小版本号。

接下来的两个字节 **00 34** 代表的是major version，指的是class文件的大版本号。例如jdk1.7默认是51.0，jdk1.8是52.0

接下来的两个字节 00 10 代表的是constand\_pool\_count，常量池中的常量序号是从1开始的，实际常量池的常量个数=constand\_pool\_count-1。 为什么从1开始呢？因为预留0位置，为以后不指向任何常量池内容做的预留。 而从存储位数来看，16位最多可以表示65535个数，也就是说，一共可以有65534个常量值。

接下来，就是常量池中的分别每个常量代表的具体的内容，不得不说的是，常量池中的每个常量，在描述自己的内容之前，实际都会在最前面存放1个字节的标识，代表自己是什么类型的常量

下面是各种类型常量的定义：





结合着这个图，我们来继续向后尝试分析一个常量的内容：

首先每个常量先要有1个字节来标识自己是什么类型的常量，因此 **0a** 就是表示的这个类型，转成10进制就是代表的10，结合图中，10是一个CONSTANT\_Methodref

\_info，也就是一个方法引用类型的常量值，然后它还需要2个2字节指向不同的内容：

第1个2字节表示该方法所在的类或者接口，也就是一个CONSTANT\_Class\_info类型的常量在常量池中的位置（从这可以看出，常量池中的值是会互相指向的）。 这里对应的2个字节就是后续的 **00 03**，注意，这时候这个值可就不是从图上去找了，因为图上代表的只是类型的tag值，而非位置。 此时这个十进制的3表示的常量池中实际的常量位置。

第2个2字节表示该方法的方法描述信息，也就是一个CONSTANT\_NameAndType类型的常量在常量池中的位置（从这可以看出，常量池中的值是会互相指向的）。 这里对应的2个字节就是后续的 **00 0d**，与上述同理。

第一个常量分析到此，后面的常量以此类推。

然后后面还有：

this\_class，super\_class，

interface count，interfaces，

fields\_count，fields，

methods\_count，method\_info，

attribute\_count，attribute

我不在此一一解释了，因为我想你也发现了，这种方式分析毕竟有些费劲。。。。

**Class文件的友好些的查看**

其实关于class文件的解读，有一些比较友好的查看方式：

**javap （java自带的命令）**

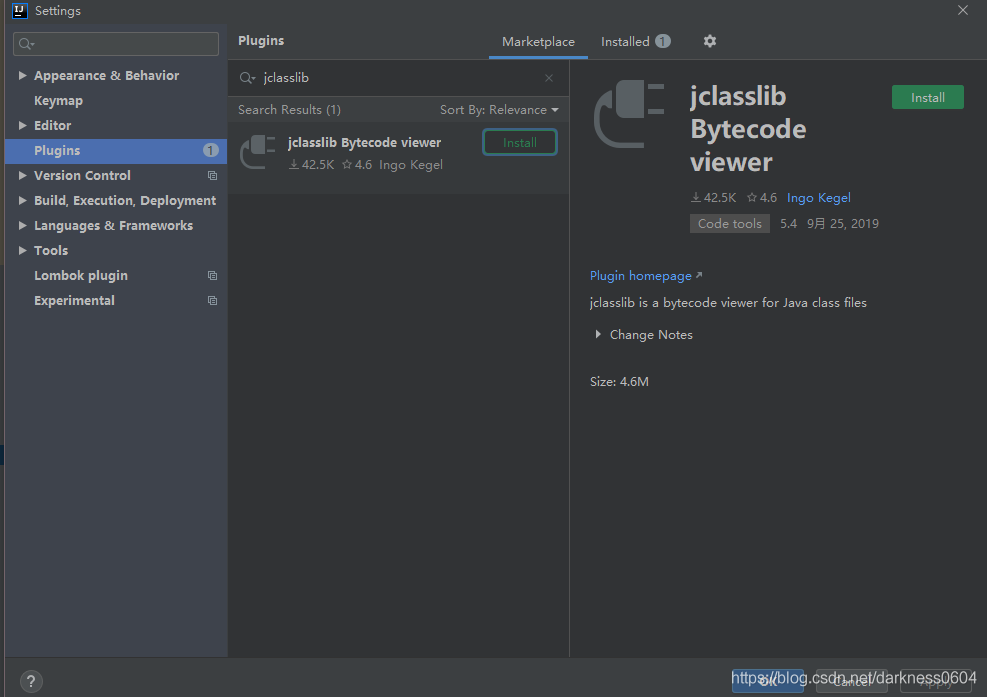
|  |
| --- |
| E:\jiagou\JVM\_GIT\out\production\JVM\com\mashibing\jvm\c1\_bytecode>javap -v T0100\_ByteCode01.class  Classfile /E:/jiagou/JVM\_GIT/out/production/JVM/com/mashibing/jvm/c1\_bytecode/T0100\_ByteCode01.class  Last modified 2019-11-17; size 333 bytes  MD5 checksum 16a7ef0d7603e3f6d34ef429be908079  Compiled from "T0100\_ByteCode01.java"  public class com.mashibing.jvm.c1\_bytecode.T0100\_ByteCode01  minor version: 0  major version: 52  flags: ACC\_PUBLIC, ACC\_SUPER  Constant pool:  #1 = Methodref #3.#13 // java/lang/Object."<init>":()V  #2 = Class #14 // com/mashibing/jvm/c1\_bytecode/T0100\_ByteCode01  #3 = Class #15 // java/lang/Object  #4 = Utf8 <init>  #5 = Utf8 ()V  #6 = Utf8 Code  #7 = Utf8 LineNumberTable  #8 = Utf8 LocalVariableTable  #9 = Utf8 this  #10 = Utf8 Lcom/mashibing/jvm/c1\_bytecode/T0100\_ByteCode01;  #11 = Utf8 SourceFile  #12 = Utf8 T0100\_ByteCode01.java  #13 = NameAndType #4:#5 // "<init>":()V  #14 = Utf8 com/mashibing/jvm/c1\_bytecode/T0100\_ByteCode01  #15 = Utf8 java/lang/Object  {  public com.mashibing.jvm.c1\_bytecode.T0100\_ByteCode01();  descriptor: ()V  flags: ACC\_PUBLIC  Code:  stack=1, locals=1, args\_size=1  0: aload\_0  1: invokespecial #1 // Method java/lang/Object."<init>":()V  4: return  LineNumberTable:  line 3: 0  LocalVariableTable:  Start Length Slot Name Signature  0 5 0 this Lcom/mashibing/jvm/c1\_bytecode/T0100\_ByteCode01;  }  SourceFile: "T0100\_ByteCode01.java" |

**JBE**

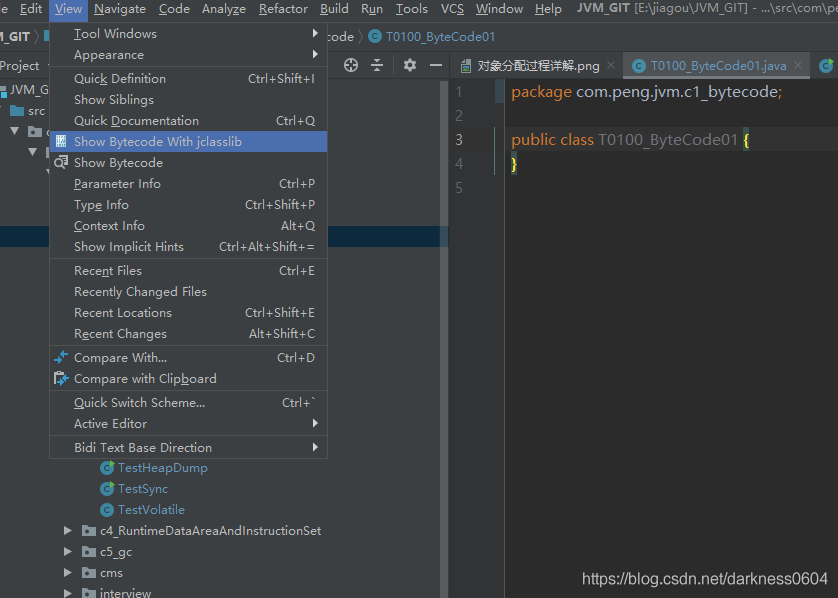
有兴趣可以了解，它 不但可以查看二进制文件，还可以直接对二进制文件进行修改

**JClassLib （IDEA的插件，比较推荐）**

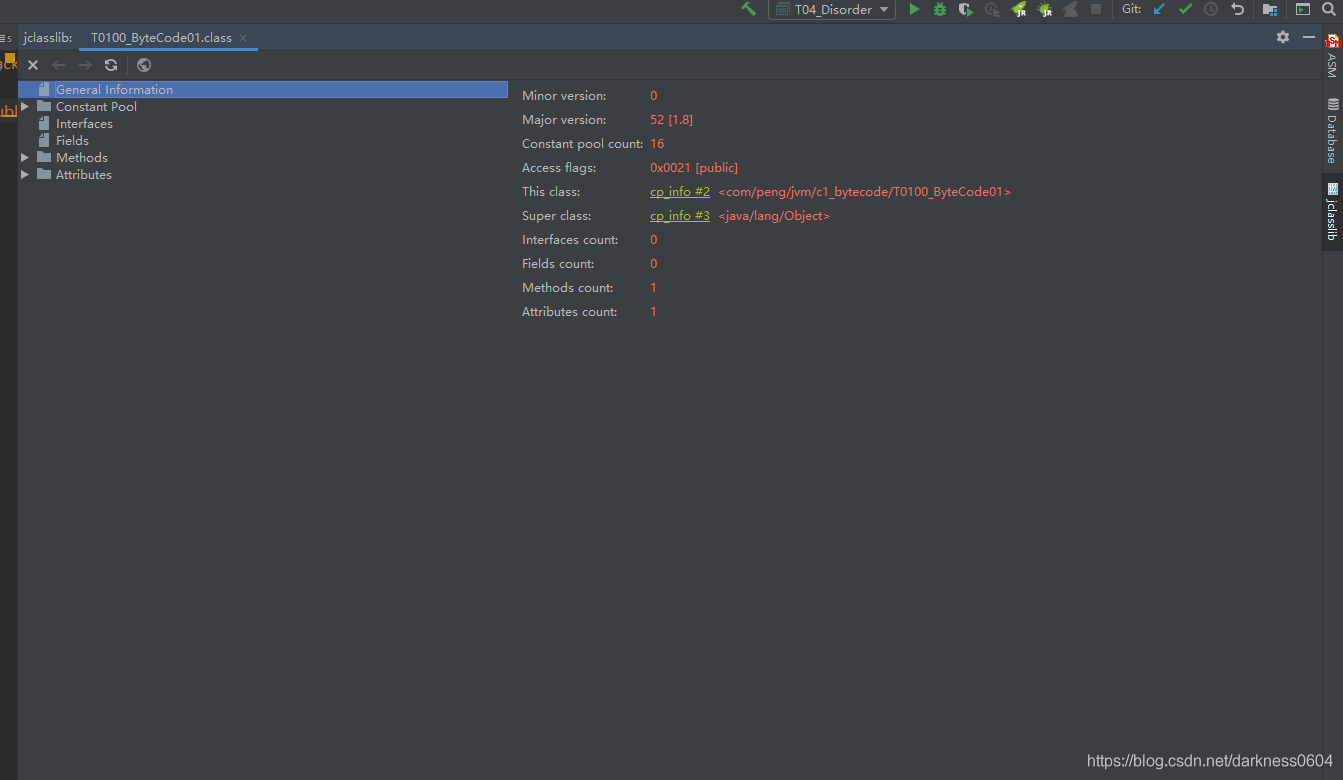
通过idea的plugins安装：



安装完毕后，需要根据提示重启idea。

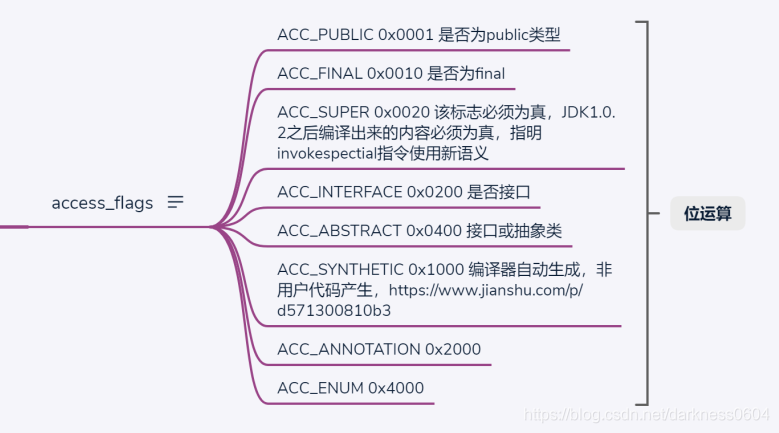


然后在我们刚刚的java文件中光标定位到类体里面。点击图上位置



**General information**

我们可以看到，在这个通用信息中，包含了我们刚刚说到的几个属性，除此之外，还有一些我没说到的。



Access flags：访问修饰符 每种修饰符都由16进制数构成，各种修饰符通过位运算后得到一个值代表它们的组合。 这里使用了 ACC\_PUBLIC 和 ACC\_SUPER 或运算 得到的0x0021

This class：这个类的名称，具体名称指向存储在常量池中的常量中。

Super class：这个类的父类名，具体名称指向存储在常量池中的常量中。

Interfaces count：这个类实现的接口个数

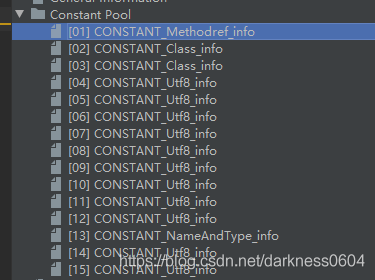
Fields count：这个类属性个数

Methods count：方法个数

Attributes count：额外的属性个数

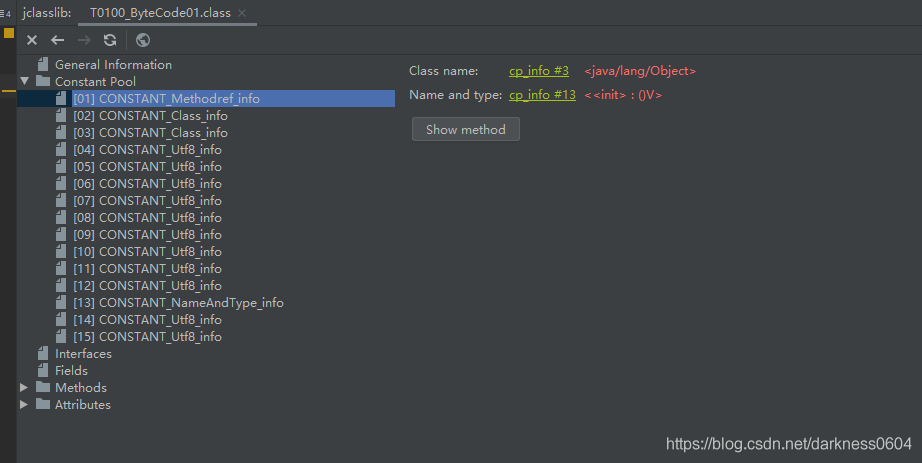
**Constant Pool**

我们在通用信息中，看到cosntant pool count是16个,但看常量池实际只有15个：



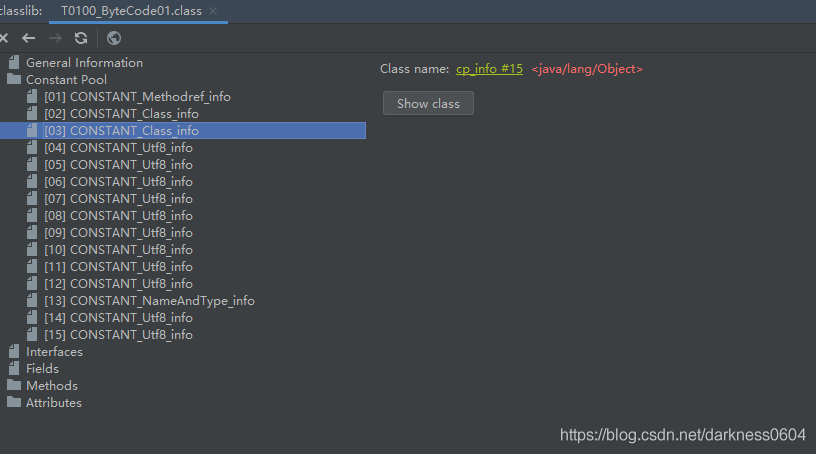
验证了我们之前说的，常量池常量数 = cosntant pool count-1

还记得我们上面分析字节码的时候分析了第一个常量的内容吗，此时看一下，第一个常量：

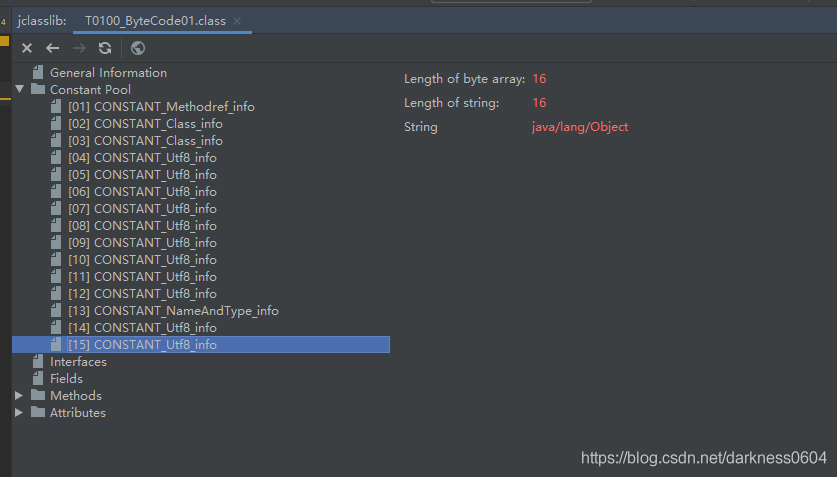


是一个方法引用常量，并且方法所在类的名称指向了常量池3号位置，方法描述指向了常量池13号位置，是不是都验证上了？

我们看下3号：

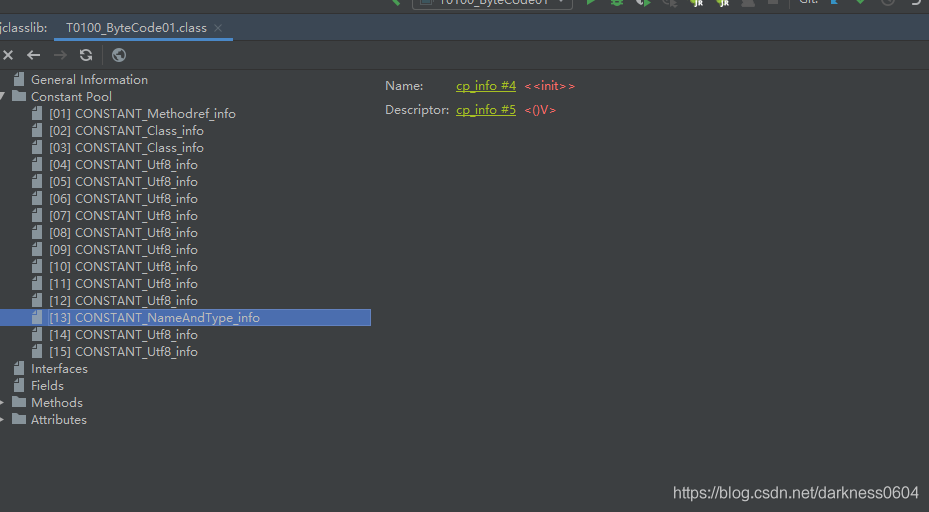


发现它的字面量内容实际又存储在了15号上，我们继续看15号：



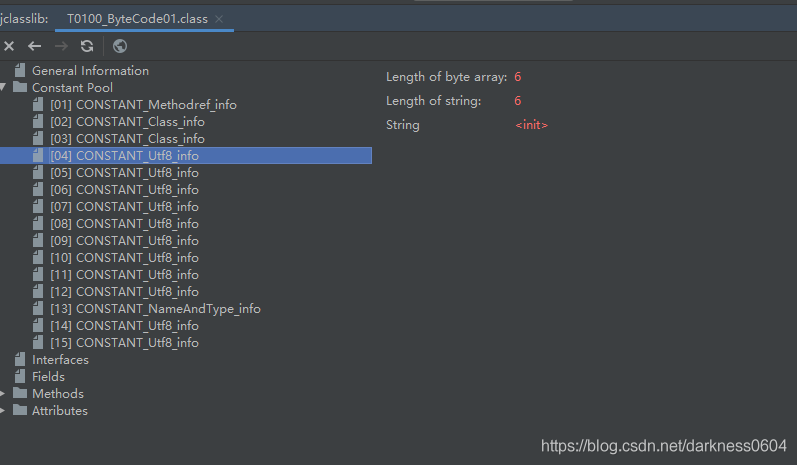
啊，就是它，最终实际的值是存在了15号常量上，代表的类名是“java/lang/object”

同理，再看下13号：

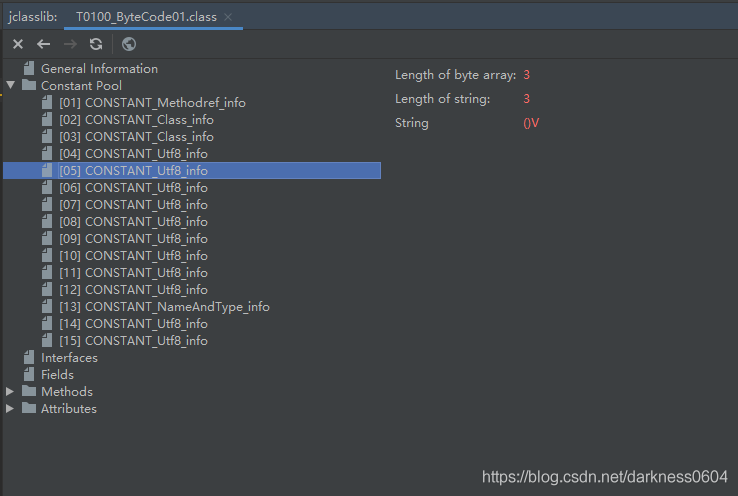


发现它内部也是要指向一个字面量的东西。。。

4号：

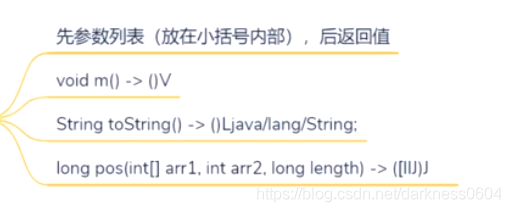


5号：



所以这也验证了我们说的，常量池里的常量是互相引用其他常量的。

**方法描述**



这里提一嘴class文件规范里边方法的描述，是先参数列表，后返回值标识的，例如有1个方法是 void m()，那它的class规范中就会表示为 ()V。V也就是void。

**类型表示**

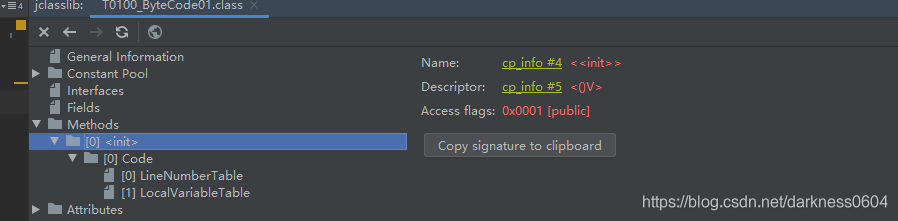
那各种类型都怎么表示呢？如图：



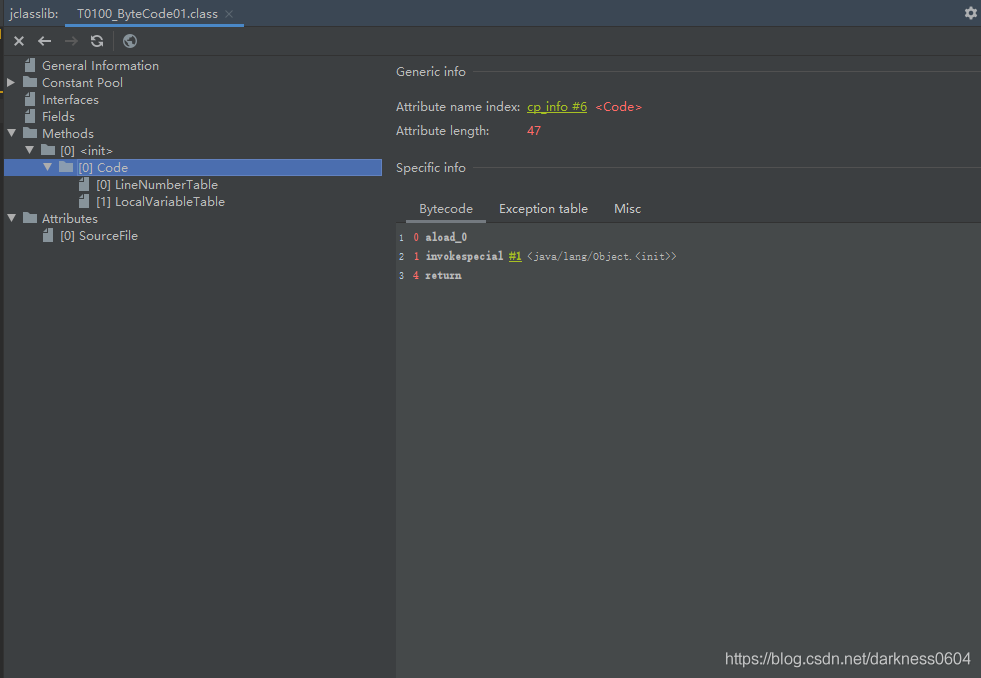
**Interface和fields**

我们这个类没有字段和接口，所以没有内容。不过这个比较简单，和常量值类似，展示接口和字段的基本信息，不进行赘述

**Methods**



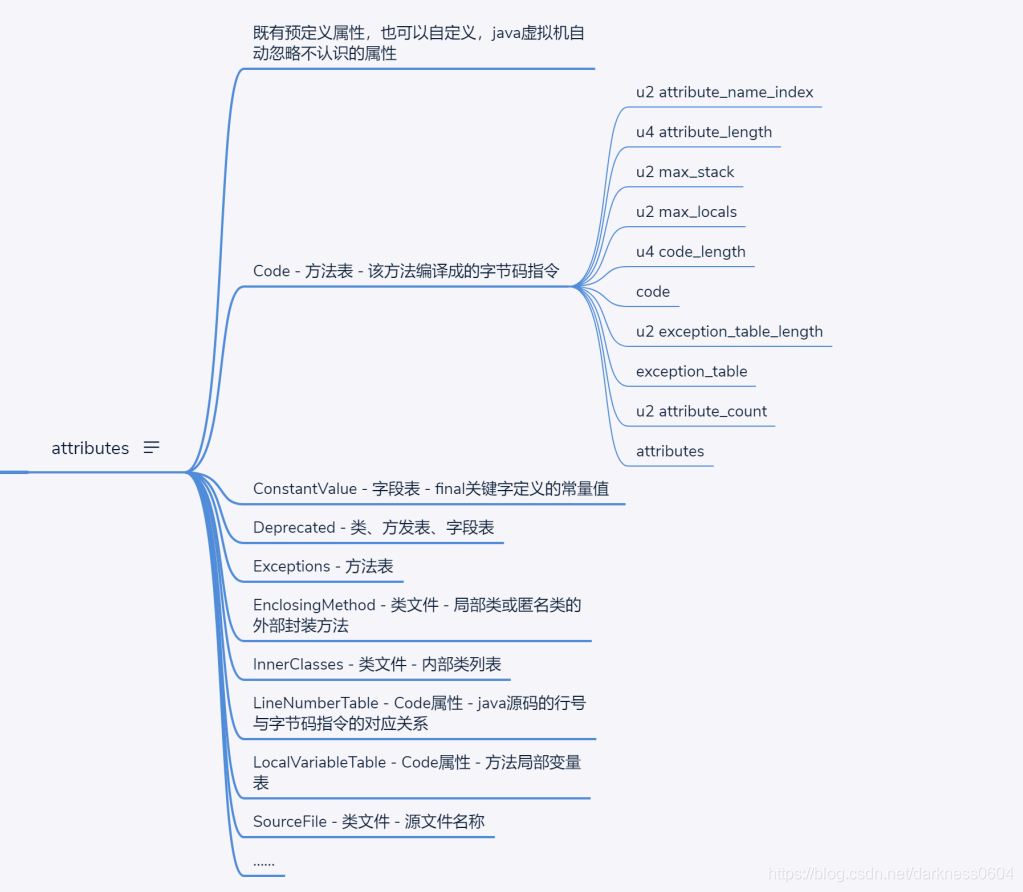
这里展示的就是这个类里各种方法的信息。其中包含了方法的描述信息，以及方法代码的具体实现，在Code中。



发现出现了一条条指令，这些指令是啥呢？ 这就牵扯到JVM的汇编指令了。也就是说，class中的这部分字节码，将根据某种规范，转换成一个个指定的操作。

通过这些操作，来完成我们代码功能的真正运作实现。

而Code，其实是属于Attributes中的一部分。



关于JVM汇编指令我们可以后面可以单独拿出来说一说，这里先有个印象就OK了。