

本课时我们从字节码层面分析class类文件结构。首先来看一道面试题：

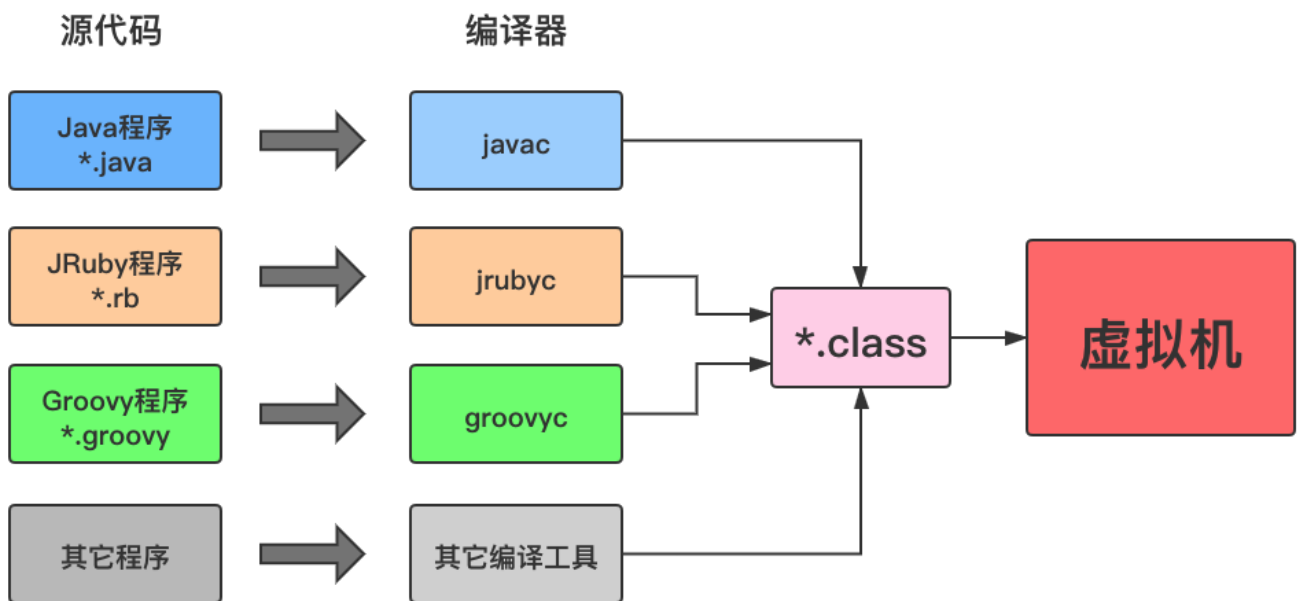
java中String字符串的长度有限制吗？

平时项目开发中，我们经常会用到String来声明字符串，比如String str = "abc"，但是你可能从来没有想过等于号之后的字符串常量到底有没有长度限制。要彻底答对这道题，就需要先学会今天所讲的内容——class 文件。

1. class 的来龙去脉

Java能够实现“一次编译，到处运行”，这其中class文件要占大部分功劳。为了让Java语言具有良好的跨平台能力，Java独具匠心的提供了一种可以在所有平台上都能使用的一种中间代码——字节码类文件（.class文件）。有了字节码，无论是哪种平台（如：Mac、Windows、Linux 等），只要安装了虚拟机都可以直接运行字节码。

并且，有了字节码，也解除了Java虚拟机和Java语言之间的耦合。这句话你可能不是很理解，这种解耦指的是什么？其实，Java虚拟机当初被设计出来的目的就不单单是只运行Java这一种语言。目前Java虚拟机已经可以支持很多除Java语言以外的其他语言了，如Groovy、JRuby、Jython、Scala等。之所以可以支持其他语言，是因为这些语言经过编译之后也可以生成能够被 JVM 解析并执行的字节码文件。而虚拟机并不关心字节码是由哪种语言编译而来的。如下图所示：

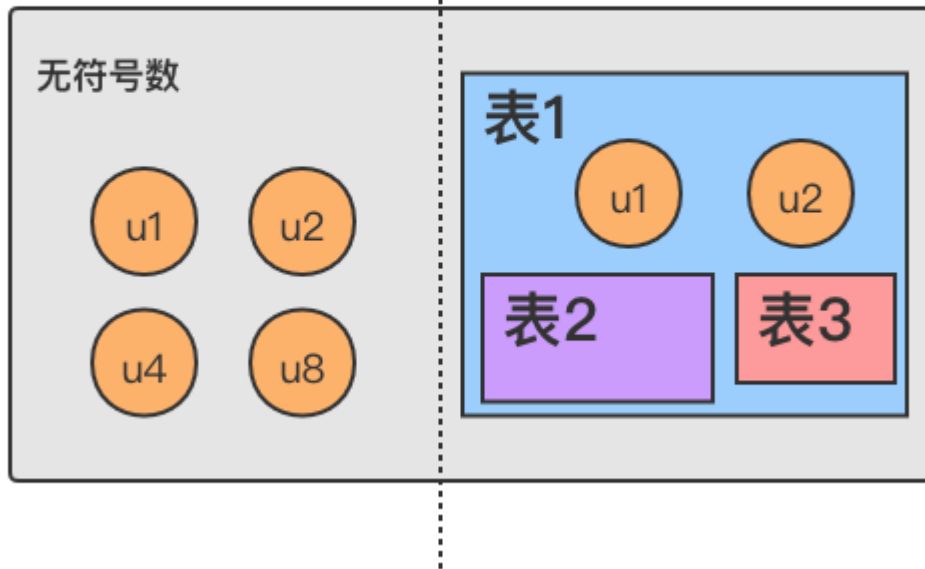


2. 上帝视角看 class 文件

如果从纵观的角度来看class文件，class文件里只有两种数据结构：无符号数 和 表。

- **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个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者字符串（UTF-8 编码）。
- **表**：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，class文件中所有的表都以“_info”结尾。其实，整个 Class 文件本质上就是一张表。这两者之间的关系可以用下面这张图来表示：

表和无符号数之间的关系



可以看出，在一张表中可以包含其他无符号数和其他表格。伪代码可以如下所示：

```
//无符号数
u1=byte[1];
u2=byte[2];
u4=byte[4];
u8=byte[8];

//表
class_table{
    //表中可以引用各种无符号数
    u1 tag;
    u2 index2;
    ...

    // 表中也可以引用其它表
    method_table mt;
    ...
}
```

3. class 文件结构

刚才我们说在class文件中只存在无符号数和表这两种数据结构。而这些无符号数和表就组成了class中的各个结构。这些结构按照 预先规定好的顺序 紧密的从前向后排列，相邻的项之间没有任何间隙。如下图：

class文件结构

魔数	版本号	常量池	访问标志	类/父类/接口	字段描述集合	字段描述集合	属性描述集合
----	-----	-----	------	---------	--------	--------	--------

当 JVM 加载某个 class 文件时，JVM 就是根据上图中的结构去解析 class 文件，加载 class 文件到内存中，并在内存中分配相应的空间。具体某一种结构需要占用大多空间，可以参考下图：

字段	名称	数据类型	数量
magic number	魔数	u4	1
major version	主版本号	u2	1
minor version	副版本号	u2	1
constant_pool_count	常量池大小	u2	1
constant_pool	常量池	cp_info	countant_pool_count - 1
access_flag	访问标志	u2	1
this_class	当前类索引	u2	1
super_class	父类索引	u2	1
interfaces_count	接口索引集合大小	u2	1
interfaces	接口索引集合	u2	interfaces_count
fields_count	字段索引集合大小	u2	1
fields	字段索引集合	field_info	fields_count
methods_count	方法索引集合大小	u2	1
methods	方法索引集合	method_info	methods_count
attributes_count	属性索引集合大小	u2	1
attributes	属性索引集合	attribute_info	attributes_count

看到这里你可能会有点概念混淆，分不清无符号数、表格以及上面的结构是什么关系。其实可以举一个简单的例子：人类的身体是由H、O、C、N等元素组成的。但是这些元素又是按照一定的规律组成了人类身体的各个器官。并且这些器官的组织顺序是有严格顺序要求的，毕竟眼睛不能长在屁股上。

4. 实例分析

理清这些概念之后，接下来通过一个Java代码实例，来看一下上面这几个结构的详细情况。首先编写一个简单的 Java 源代码 Test.java，如下所示：

```
import java.io.Serializable;
public class Test implements Serializable,Cloneable{
```

```

        private int num = 1;

        public int add(int i){
            int j = 10;
            num = num + i;
            return num;
        }
    }
}

```

通过 javac 将其编译，生成 Test.class 字节码文件。然后使用 16 进制编辑器打开 class 文件，显示内容如下所示：

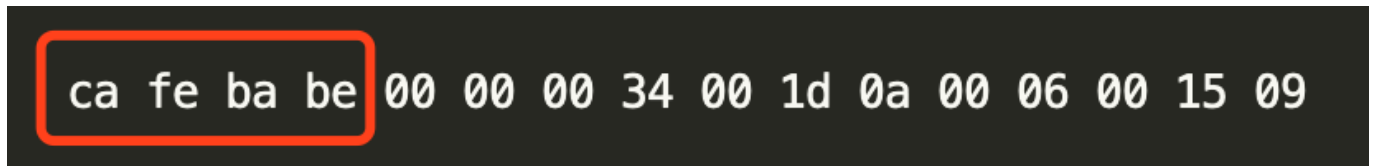
```

ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 15 09
00 05 00 16 08 00 17 09 00 05 00 18 07 00 19 07
00 1a 07 00 1b 07 00 1c 01 00 03 6e 75 6d 01 00
01 49 01 00 03 73 74 72 01 00 12 4c 6a 61 76 61
2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 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 03 61 64 64 01 00 04 28 49
29 49 01 00 0a 53 6f 75 72 63 65 46 69 6c 65 01
00 09 54 65 73 74 2e 6a 61 76 61 0c 00 0d 00 0e
0c 00 09 00 0a 01 00 03 61 62 63 0c 00 0b 00 0c
01 00 04 54 65 73 74 01 00 10 6a 61 76 61 2f 6c
61 6e 67 2f 4f 62 6a 65 63 74 01 00 14 6a 61 76
61 2f 69 6f 2f 53 65 72 69 61 6c 69 7a 61 62 6c
65 01 00 13 6a 61 76 61 2f 6c 61 6e 67 2f 43 6c
6f 6e 65 61 62 6c 65 00 21 00 05 00 06 00 02 00
07 00 08 00 02 00 02 00 09 00 0a 00 00 00 00 00
0b 00 0c 00 00 00 02 00 01 00 0d 00 0e 00 01 00
0f 00 00 00 30 00 02 00 01 00 00 00 10 2a b7 00
01 2a 04 b5 00 02 2a 12 03 b5 00 04 b1 00 00 00
01 00 10 00 00 00 0e 00 03 00 00 00 03 00 04 00
04 00 09 00 05 00 01 00 11 00 12 00 01 00 0f 00
00 00 2b 00 03 00 02 00 00 00 0f 2a 2a b4 00 02
1b 60 b5 00 02 2a b4 00 02 ac 00 00 00 01 00 10
00 00 00 0a 00 02 00 00 00 08 00 0a 00 09 00 01
00 13 00 00 00 02 00 14

```

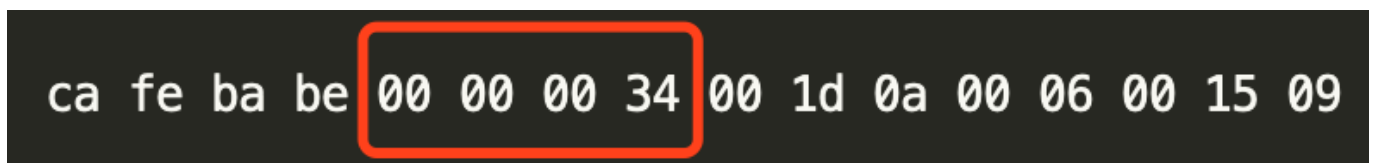
上图中都是一些 16 进制数字，每两个字符代表一个字节。乍看一下各个字符之间毫无规律，但是在 JVM 的视角里这些 16 进制字符是按照严格的规律排列的。接下来就一步一步看下 JVM 是如何解析它们的。

5. 魔数 magic number



如上图所示，在class文件开头的四个字节是class文件的魔数，它是一个固定的值--0XCAFEBABE。魔数是class文件的标志，也就是说它是判断一个文件是不是 class 格式文件的标准，如果开头四个字节不是 0XCAFEBABE，那么就说明它不是 class 文件，不能被 JVM 识别或加载。

6. 版本号



紧跟在魔数后面的两个字节代表当前class文件的版本号。前两个字节0000代表次版本号（minor_version），后两个字节0034是主版本号（major_version），对应的十进制为 52，也就是说当前 class 文件的主版本号为 52，次版本号为 0。所以综合版本号是 52.0，也就是 jdk1.8.0。

7. 常量池（重点）

紧跟在版本号之后的是一个叫作常量池的表（cp_info）。在常量池中保存了类的各种相关信息，比如类的名称、父类的名称、类中的方法名、参数名称、参数类型等，这些信息都是以各种表的形式保存在常量池中的。

常量池中的每一项都是一个表，其项目类型共有 14 种，如下表所示：

表名	标识位	描述
CONSTANT_utf8_info	1	UTF-8编码字符串表
CONSTANT_Integer_info	3	整形常量表
CONSTANT_Float_info	4	浮点型常量表
CONSTANT_Long_info	5	长整形常量表
CONSTANT_Double_info	6	双精度浮点型常量表
CONSTANT_Class_info	7	类/接口 引用表
CONSTANT_String_info	8	字符串常量表
CONSTANT_Fieldref_info	9	字段引用表
CONSTANT_Methodref_info	10	类的方法引用表
CONSTANT_InterfaceMethodref_info	11	接口的方法引用表
CONSTANT_NameAndType_info	12	字段或方法的名称和类型表
CONSTANT_MethodHandle_info	15	方法句柄表
CONSTANT_MethodType_info	16	方法类型表
CONSTANT_InvokeDynamic_info	18	动态方法调用表

可以看出，常量池中的每一项都会有一个u1大小的tag值。tag值是表的标识，JVM解析class文件时，通过这个值来判断当前数据结构是哪一种表。以上14种表都有自己的结构，这里不再一一介绍，就以CONSTANT_Class_info 和 CONSTANT_Utf8_info 这两张表举例说明，因为其他表也基本类似。

首先，CONSTANT_Class_info 表具体结构如下所示：

```
table CONSTANT_Class_info {
    u1  tag = 7;
    u2  name_index;
}
```

解释说明。

- tag： 占用一个字节大小。比如值为7，说明是CONSTANT_Class_info类型表。
- name_index： 是一个索引值，可以将它理解为一个指针，指向常量池中索引为name_index = 2，则它指向常量池中第 2 个常量。

接下来再看 CONSTANT_Utf8_info 表具体结构如下：

```
table CONSTANT_utf8_info {
    u1  tag;
    u2  length;
```

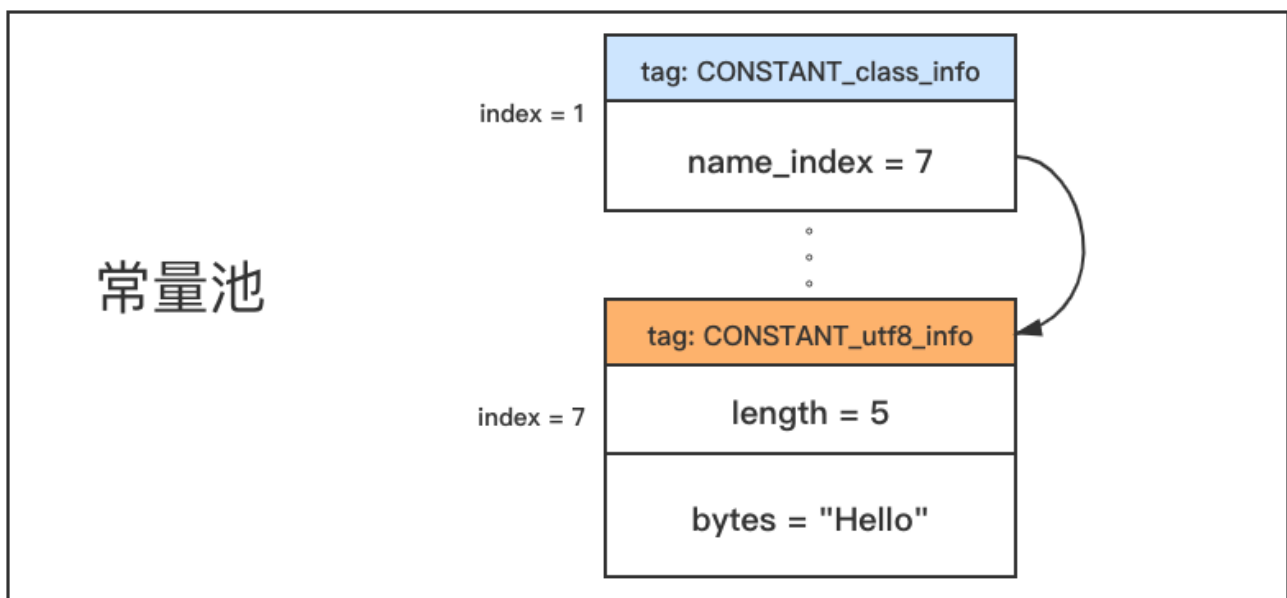
```
    u1[] bytes;  
}
```

解释说明：

- tag: 值为1, 表示是CONSTANT_utf8_info类型表。
- length: length表示u1[]的长度, 比如length=...5, 则表示接下来的数据是5个连续的 u1 类型数据。
- bytes: u1 类型数组, 长度为上面第 2 个参数 length 的值。

而我们在java代码中声明的String字符串最终在class文件中的存储格式就CONSTANT_utf8_info。因此一个字符串最大长度也就是u2所能代表的最大值65536个, 但是需要使用2个字节来保存 null 值, 因此一个字符串的最大长度为 $65536 - 2 = 65534$ 。参考 [Java String最大长度分析](#)

不难看出, 在常量池内部的表中也有相互之间的引用。用一张图来理解 CONSTANT_Class_info 和 CONSTANT_utf8_info 表格之间的关系, 如下图所示:



理解了常量池内部的数据结构之后, 接下来就看一下实例代码的解析过程。因为开发者平时定义的 Java 类各式各样, 类中的方法与参数也不尽相同。所以常量池的元素数量也就无法固定, 因此 class 文件在常量池的前面使用 2 个字节的容量计数器, 用来代表当前类中常量池的大小。如下图所示:

ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 15 09

红色框中的 001d 转化为十进制就是 29, 也就是说常量计数器的值为 29。其中下标为 0 的常量被 JVM 留作其他特殊用途, 因此 Test.class 中实际的常量池大小为这个计数器的值减 1, 也就是 28 个。

第一个常量, 如下所示:

ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 15 09

0a 转化为10进制后为10，通过查看常量池14种表格图中，可以查到tag=10的表类型为 CONSTANT_Methodref_info，因此常量池中的第一个常量类型为方法引用表。其结构如下：

```
CONSTANT_Methodref_info{
    u1 tag = 10;
    u2 class_index; 指向此方法的所属类
    u1 name_type_index; 指向此方法的名称和类型
}
```

也就是说在“0a”之后的 2 个字节指向这个方法是属于哪个类，紧接的 2 个字节指向这个方法的名称和类型。它们的值分别是：

- 0006：十进制 6，表示指向常量池中的第 6 个常量。
- 0015：十进制 21，表示指向常量池中的第 21 个常量。

至此，第 1 个常量就解读完毕了。紧接着的就是第 2 个常量，如下所示

ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 15 09
00 05 00 16 08 00 17 09 00 05 00 18 07 00 19 07

tag 09 表示是字段引用表 CONSTANT_Fieldref_info，其结构如下：

```
CONSTANT_Fieldref_info{
    u1 tag;
    u2 class_index; 指向此字段的所属类
    u2 name_type_index; 指向此字段的名称和类型
}
```

同样也是 4 个字节，前后都是两个索引。

- 0005：指向常量池中第 5 个常量。
- 0016：指向常量池中第 22 个常量。

到现在为止我们已经解析出了常量池中的两个常量。剩下的 21 个常量的解析过程也大同小异，这里就不一一解析了。实际上我们可以借助 javap 命令来帮助我们查看 class 常量池中的内容：

javap -v Test.class

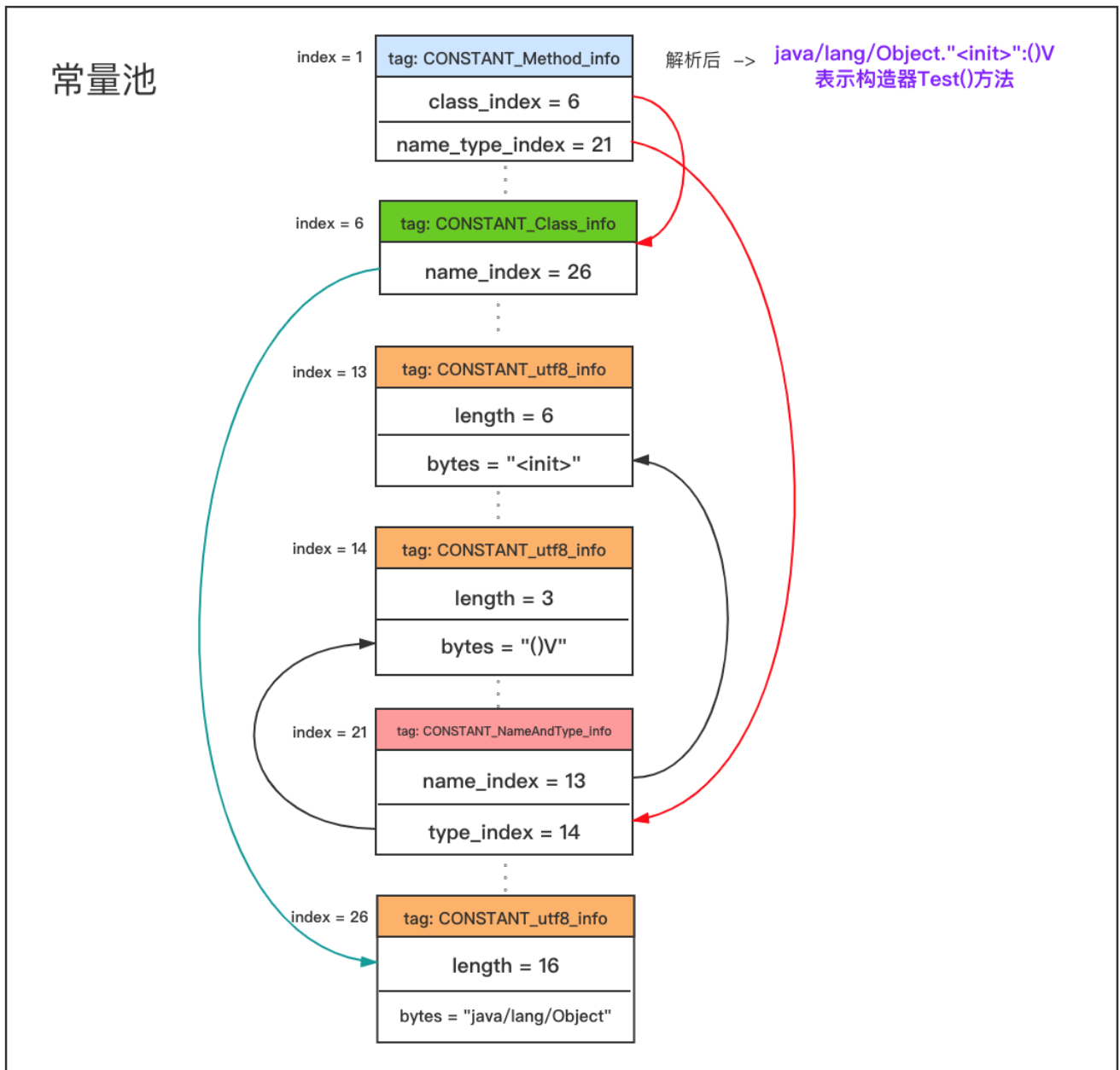
上述命令执行后，显示结果如下：

```
public class Test implements java.io.Serializable,java.lang.Cloneable
  minor version: 0
  major version: 52
  flags: ACC PUBLIC, ACC SUPER
Constant pool:
  #1 = Methodref          #6.#21      // java/lang/Object."<init>":()V
  #2 = Fieldref           #5.#22      // Test.num:I
  #3 = String             #23         // abc
  #4 = Fieldref           #5.#24      // Test.str:Ljava/lang/String;
  #5 = Class               #25         // Test
  #6 = Class               #26         // java/lang/Object
  #7 = Class               #27         // java/io/Serializable
  #8 = Class               #28         // java/lang/Cloneable
  #9 = Utf8               num
  #10 = Utf8              I
  #11 = Utf8              str
  #12 = Utf8              Ljava/lang/String;
  #13 = Utf8              <init>
  #14 = Utf8              ()V
  #15 = Utf8              Code
  #16 = Utf8              LineNumberTable
  #17 = Utf8              add
  #18 = Utf8              (I)I
  #19 = Utf8              SourceFile
  #20 = Utf8              Test.java
  #21 = NameAndType        #13:#14     // "<init>":()V
  #22 = NameAndType        #9:#10      // num:I
  #23 = Utf8              abc
  #24 = NameAndType        #11:#12     // str:Ljava/lang/String;
  #25 = Utf8              Test
  #26 = Utf8              java/lang/Object
  #27 = Utf8              java/io/Serializable
  #28 = Utf8              java/lang/Cloneable
{
  java.lang.String str;
    descriptor: Ljava/lang/String;
    flags:
```

正如我们刚才分析的一样，常量池中第一个常量是 Methodref 类型，指向下标 6 和下标 21 的常量。其中下标 21 的常量类型为 NameAndType，它对应的数据结构如下：

```
CONSTANT_NameAndType_info{
  u1 tag;
  u2 name_index;    指向某字段或方法的名称字符串
  u2 type_index;    指向某字段或方法的类型字符串
}
```

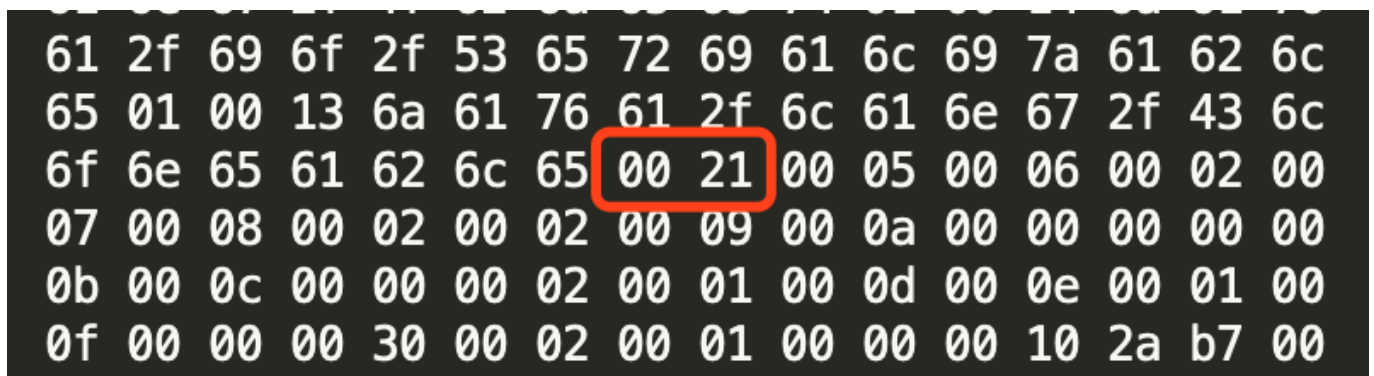
而下标在 21 的 NameAndType 的 name_index 和 type_index 分别指向了 13 和 14，也就是 "" 和 "()V"。因此最终解析下来常量池中第 1 个常量的解析过程以及最终值如下图所示：



仔细解析层层引用，最后我们可以看出，Test.class 文件中常量池的第 1 个常量保存的是 Object 中的默认构造器方法。

8. 访问标志 (access_flags)

紧跟在常量池之后的常量是访问标志，占用两个字节，如下图所示：



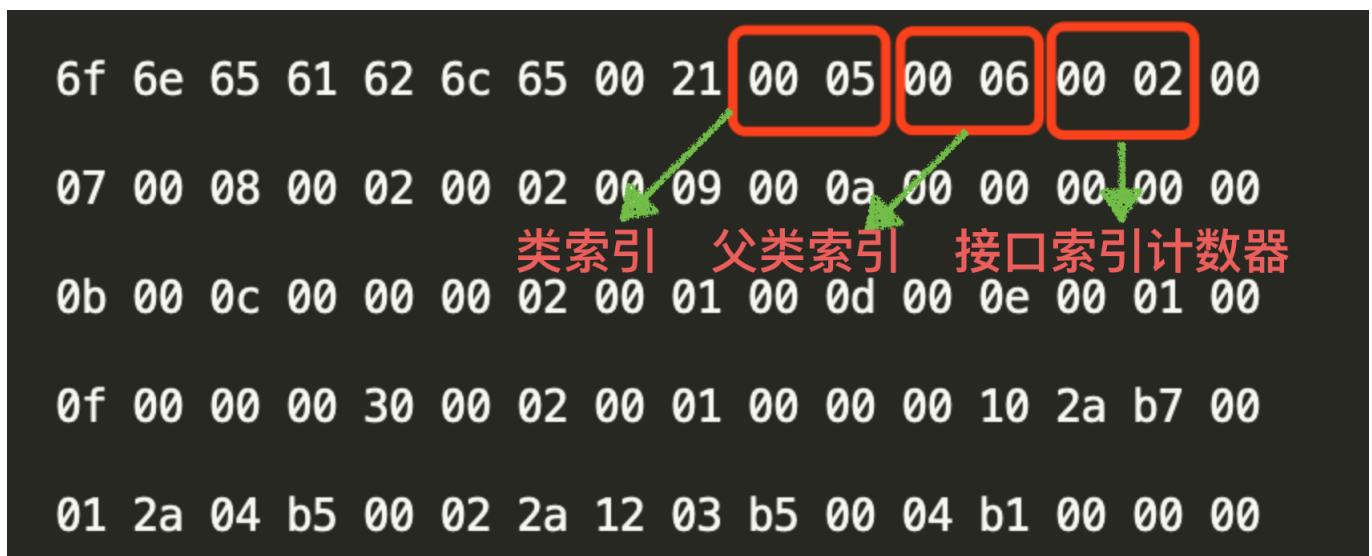
访问标志代表类或者接口的访问信息，比如：该 class 文件是类还是接口，是否被定义成 public，是否是 abstract，如果是类，是否被声明成 final 等等。各种访问标志如下所示：

访问标志	值	描述
ACC_PUBLIC	0X0001	public类型
ACC_FINAL	0X0010	被声明为final类型的类
ACC_SUPER	0X0020	是否允许使用invokespecial字节码指令的新语义，JDK1.0.2之后编译出来的类的这个标志默认为真
ACC_INTERFACE	0X0200	标志这是一个接口类型
ACC_ABSTRACT	0X0400	标志这是一个抽象类或是接口类型
ACC_ANNOTATION	0X2000	标注这是一个注解
ACC_ENUM	0X4000	标志这是一个枚举

我们定义的Test.java是一个普通Java类，不是接口、枚举或注解。并且被public修饰但没有被声明为final和abstract，因此它所对应的access_flags为 0021（0X0001 和 0X0020 相结合）。

9. 类索引、父类索引与接口索引计数器

在访问标志后的 2 个字节就是类索引，类索引后的 2 个字节就是父类索引，父类索引后的 2 个字节则是接口索引计数器。如下图所示：



可以看出类索引指向常量池中的第 5 个常量，父类索引指向常量池中的第 6 个常量，并且实现的接口个数为 2 个。再回顾下常量池中的数据：

```
Constant pool:
  #1 = Methodref          #6.#21      // java/lang/Object."<init>":()V
  #2 = Fieldref           #5.#22      // Test.num:I
  #3 = String             #23         // abc
  #4 = Fieldref           #5.#24      // Test.str:Ljava/lang/String;
  #5 = Class               #25         // Test
  #6 = Class               #26         // java/lang/Object
  #7 = Class               #27         // java/io/Serializable
  #8 = Class               #28         // java/lang/Cloneable
  #9 = Utf8               num
```

从图中可以看出，第 5 个常量和第 6 个常量均为 CONSTANT_Class_info 表类型，并且代表的类分别是“Test”和“Object”。再看接口计数器，因为接口计数器的值是 2，代表这个类实现了 2 个接口。查看在接口计数器之后的 4 个字节分别为：

- 0007：指向常量池中的第 7 个常量，从图中可以看出第 7 个常量值为“Serializable”。
- 0008：指向常量池中的第 8 个常量，从图中可以看出第 8 个常量值为“Cloneable”。

综上所述，可以得出如下结论：当前类为 **Test** 继承自 **Object** 类，并实现了“**Serializable**”和“**Cloneable**”这两个接口。

10. 字段表

紧跟在接口索引集合后面的就是字段表了，字段表的主要功能是用来描述类或者接口中声明的变量。这里的字段包含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。

同样，一个类中的变量个数是不固定的，因此在字段表集合之前还是使用一个计数器来表示变量的个数，如下所示：

```
07 00 08 00 02 00 02 00 09 00 0a 00 00 00 00 00
```

0002 表示类中声明了 2 个变量（在 class 文件中叫字段），字段计数器之后会紧跟着 2 个字段表的数据结构。

字段表的具体结构如下：

```
CONSTANT_Fieldref_info{
    u2  access_flags    字段的访问标志
    u2  name_index      字段的名称索引(也就是变量名)
    u2  descriptor_index 字段的描述索引(也就是变量的类型)
    u2  attributes_count 属性计数器
    attribute_info
}
```

继续解析 Text.class 中的字段表，其结构如下图所示：



11. 字段访问标志

对于 Java 类中的变量，也可以使用 public、private、final、static 等标识符进行标识。因此解析字段时，需要先判断它的访问标志，字段的访问标志如下所示：

字段访问标志	值	描述
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_ENUM	0X4000	字段是否为enum

字段表结构图中的访问标志的值为 0002，代表它是 private 类型。变量名索引指向常量池中的第 9 个常量，变量名类型索引指向常量池中第 10 个常量。第 9 和第 10 个常量分别为“num”和“I”，如下所示：

```
Constant pool:
  #1 = Methodref          #6.#21      // java/lang/Object."<init>":()V
  #2 = Fieldref           #5.#22      // Test.num:I
  #3 = String             #23         // abc
  #4 = Fieldref           #5.#24      // Test.str:Ljava/lang/String;
  #5 = Class               #25         // Test
  #6 = Class               #26         // java/lang/Object
  #7 = Class               #27         // java/io/Serializable
  #8 = Class               #28         // java/lang/Cloneable
  #9 = Utf8                num
 #10 = Utf8                I
 #11 = Utf8                str
 #12 = Utf8                Ljava/lang/String;
 #13 = Utf8                <init>
 #14 = Utf8                ()V
```

因此可以得知类中有一个名为 num，类型为 int 类型的变量。对于第 2 个变量的解析过程也是一样，就不再过多介绍。


注意事项：

1. 字段表集合中不会列出从父类或者父接口中继承而来的字段。
2. 内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

对于以上两种情况，建议你可以自行定义一个类查看并手动分析一下。

12. 方法表

字段表之后跟着的就是方法表常量。相信你应该也能猜到了，方法表常量应该也是以一个计数器开始的，因为一个类中的方法数量是不固定的，如图所示：



0f 00 00 00 30 00 02 00 01 00 00 00 10 2a b7 00

上图表示 Test.class 中有两个方法，但是我们只在 Test.java 中声明了一个 add 方法，这是为什么呢？这是因为默认构造器方法也被包含在方法表常量中。

方法表的结构如下所示：

```
CONSTANT_Methodref_info{
    u2  access_flags;      方法的访问标志
    u2  name_index;        指向方法名的索引
    u2  descriptor_index;  指向方法类型的索引
    u2  attributes_count;  方法属性计数器
    attribute_info attributes;
}
```

可以看到，方法也是有自己的访问标志，具体如下：

访问标志	值	描述
ACC_PUBLIC	0X0001	方法是否为public
ACC_PRIVATE	0X0002	方法是否为private
ACC_PROTECTED	0X0004	方法是否为protected
ACC_STATIC	0X0008	方法是否为static
ACC_FINAL	0X0010	方法是否为final
ACC_SYNCHRONIZED	0X0020	方法是否被synchronized修饰
ACC_VARARGS	0X0080	方法是否接收参数
ACC_NATIVE	0X0100	方法是否为native
ACC_ABSTRACT	0X0400	方法是否为abstract

我们主要来看下 add 方法，具体如下：



从图中我们可以看出 add 方法的以下字段的具体值：

1. access_flags = 0X0001 也就是访问权限为 public。
2. name_index=0X0011指向常量池中的第17个常量，也就是“add”。
3. type_index=0X0012指向常量池中的第18个常量，也即是(I)这个方法接收 int 类型参数，并返回 int 类型参数。

13. 属性表

在之前解析字段和方法的时候，在它们的具体结构中我们都能看到有一个叫作 attributes_info 的表，这就是属性表。

属性表并没有一个固定的结构，各种不同的属性只要满足以下结构即可：

```
CONSTANT_Attribute_info{
    u2 name_index;
    u2 attribute_length length;
    u1[] info;
}
```

JVM 中预定义了很多属性表，这里重点讲一下 Code 属性表。

- Code属性表

我们可以接着刚才解析方法表的思路继续往下分析：

04 00 09 00 05 00 01 00 11 00 12 00 01 00 0f 00

可以看到，在方法类型索引之后跟着的就是“add”方法的属性。0X0001是属性计数器，代表只有一个属性。0X000f是属性表类型索引，通过查看常量池可以看出它是一个Code属性表，如下所示：

```
Constant pool:
 #1 = Methodref      #6.#21      // java/lang/Object."<init>":()V
 #2 = Fieldref       #5.#22      // Test.num:I
 #3 = String         #23        // abc
 #4 = Fieldref       #5.#24      // Test.str:Ljava/lang/String;
 #5 = Class          #25        // Test
 #6 = Class          #26        // java/lang/Object
 #7 = Class          #27        // java/io/Serializable
 #8 = Class          #28        // java/lang/Cloneable
 #9 = Utf8           num
#10 = Utf8           I
#11 = Utf8           str
#12 = Utf8           Ljava/lang/String;
#13 = Utf8           <init>
#14 = Utf8           ()V
#15 = Utf8           Code
#16 = Utf8           LineNumberTable
#17 = Utf8           add
#18 = Utf8           (I)I
#19 = Utf8           SourceFile
```

Code 属性表中，最主要的就是一些列的字节码。通过 `javap -v Test.class` 之后，可以看到方法的字节码，如下图所示的是 add 方法的字节码指令：

```
public int add(int);
  descriptor: (I)I
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=2, args_size=2
     0: aload_0
     1: aload_0
     2: getfield      #2          // Field num:I
     5: iload_1
     6: iadd
     7: putfield     #2          // Field num:I
    10: aload_0
    11: getfield     #2          // Field num:I
    14: ireturn
  LineNumberTable:
    line 8: 0
    line 9: 10
```

JVM 执行 add 方法时，就通过这一系列指令来做相应的操作。

14.总结

本课时我们主要了解了一个class文件内容的数据结构到底长什么样子，并通过Test.class来模拟演示Java虚拟机解析字节码文件的过程。其中class常量池部分是重点内容，它就相当于是 class 文件中的资源仓库，其他的几种结构或多或少都会最终指向到这个资源仓库中。实际上平时我们不太会直接用一个 16 进制编辑器去打开一个 .class 文件。我们可以使用 javap 等命令或者是其他工具，来帮助我们查看 class 内部的数据结构。只不过自己亲手操作一遍是很有助于理解 JVM 的解析过程，并加深对 class 文件结构的记忆。