课程主题

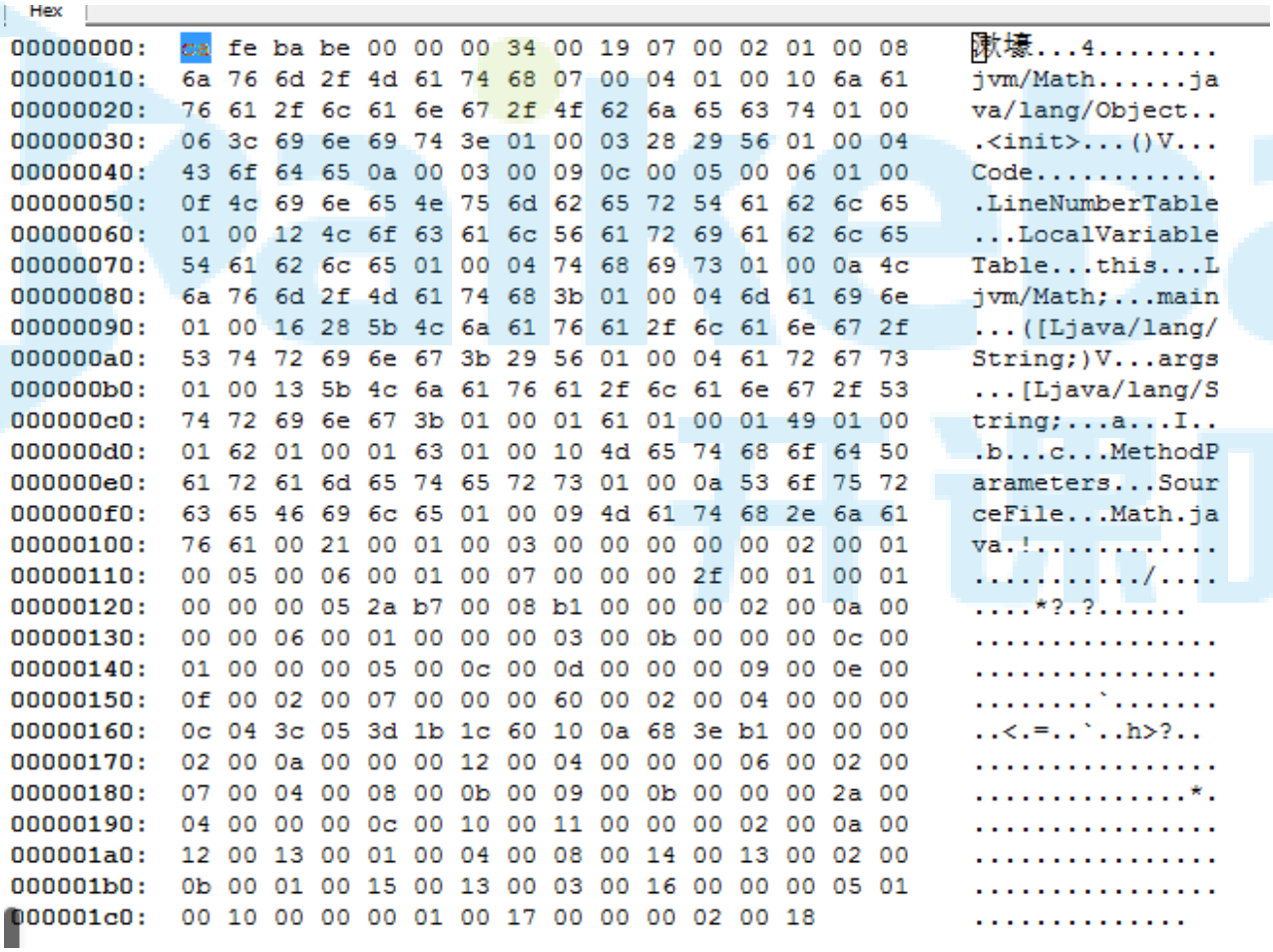
课程目标

课程内容

带你认识一下class文件

class文件概述

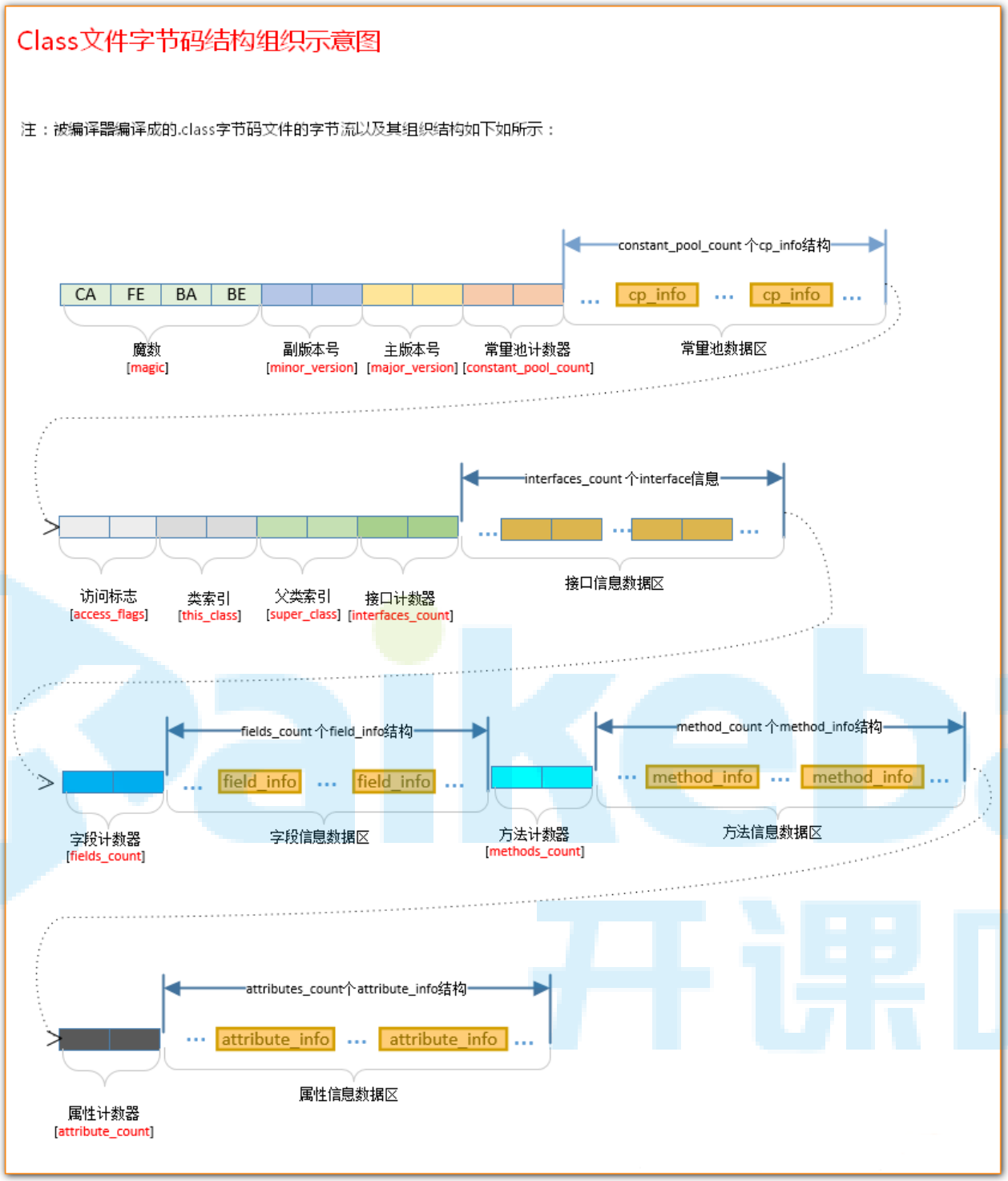
我们可任意打开一个Class文件（**使用Hex Editor等工具打开**），内容如下（**内容是16进制**）：



十六进制转字符串：http://www.bejson.com/convert/ox2str/

进制转换网址（**十六进制转十进制**）：http://tool.oschina.net/hexconvert/

参考下图去阅读上面的十六进制文档：



据上述的叙述，我们可以将class的文件组织结构概括成以下面这个表格（**其中u表示u4表示4个无符号字节，u2表示2个无符号字节**）：

类型	名称	数量
u4	magic(魔数)	1
u2	minor_version(JDK次版本号)	1
u2	major_version(JDK主版本号)	1
u2	constant_pool_count(常量池数量)	1
cp_info	constan_pool(常量表)	constant_pool_count-1
u2	access_flags(访问标志)	1
u2	this_class(类引用)	1
u2	super_class（父类引用）	1
u2	interfaces_count(接口数量)	1
u2	interfaces(接口数组)	interfaces_count
u2	fields_count(字段数量)	1
field_info	fields(字段表)	fields_count
u2	methods_count(方法数量)	1
method_info	methods(方法表)	methods_count
u2	attributes_count(属性数量)	1
attribute_info	attributes(属性表)	attributes_count

魔数

所有的由Java编译器编译而成的class文件的前4个字节都是“**0xCAFEBABE**”。

它的作用在于：当JVM在尝试加载某个文件到内存中来的时候，会首先判断此class文件有没有JVM认为可以接受的“签名”，即JVM会首先读取文件的前4个字节，判断该4个字节是否是“0xCAFEBABE”，如果是，则JVM会认为可以将此文件当作class文件来加载并使用。

版本号

随着Java本身的发展，Java语言特性和JVM虚拟机也会有相应的更新和增强。目前我们能够用到的JDK版本如：1.5，1.6，1.7，还有现如今的1.8及更高的版本。发布新版本的目的在于：在原有的版本上增加新特性和相应的JVM虚拟机的优化。而随着主版本发布的次版本，则是修改相应主版本上出现的bug。我们平时只需要关注主版本就可以了。

主版本号和次版本号在class文件中各占两个字节，**副版本号占用第5、6两个字节，而主版本号则占用第7，8两个字节。**JDK1.0的主版本号为45，以后的每个新主版本都会在原先版本的基础上加1。若现在使用的是JDK1.7编译出来的class文件，则相应的主版本号应该是51，对应的7，8个字节的十六进制的值应该是 0x33。

一个 JVM实例只能支持特定范围内的主版本号 （Mi 至Mj） 和 0 至特定范围内 （0 至 m） 的副版本号。假设一个 Class 文件的格式版本号为 V， 仅当**Mi.0 ≤ v ≤ Mj.m**成立时，这个 Class 文件才可以被此 Java 虚拟机支持。不同版本的 Java 虚拟机实现支持的版本号也不同，高版本号的 Java 虚拟机实现可以支持低版本号的 Class 文件，反之则不成立。

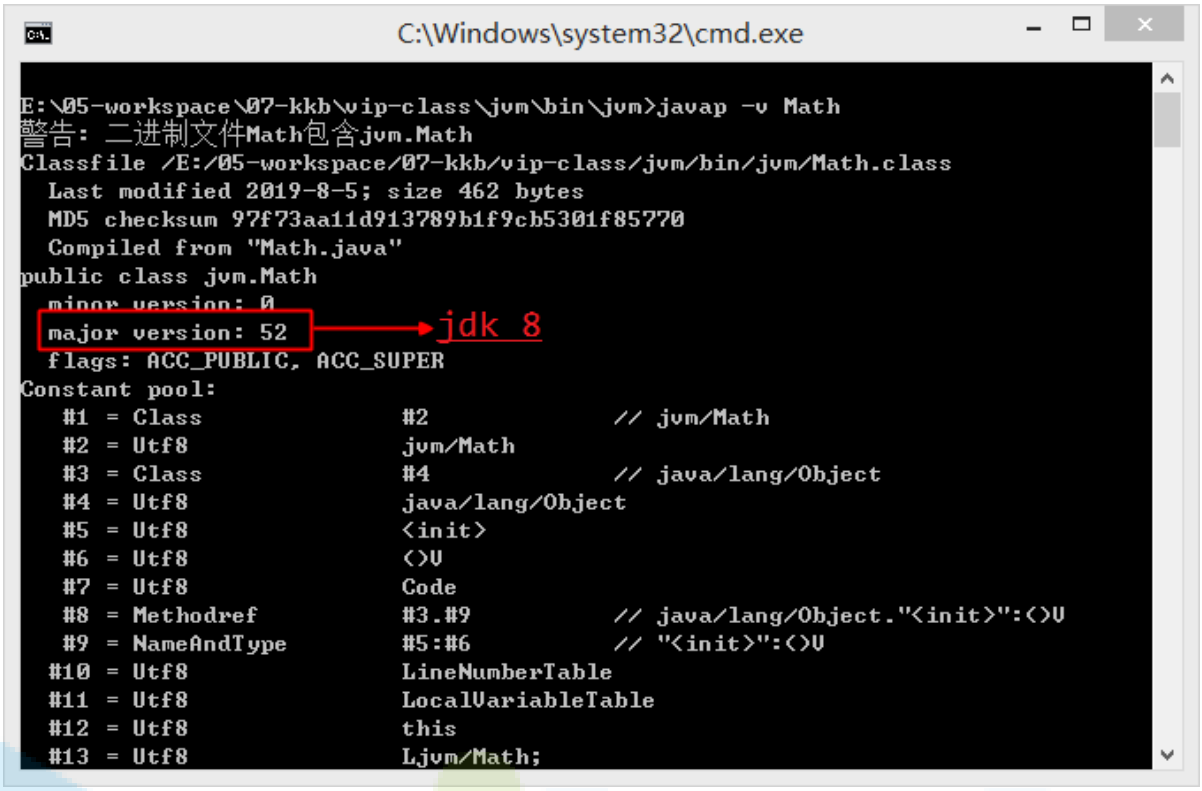
JVM在加载class文件的时候，会读取出主版本号，然后比较这个class文件的主版本号和JVM本身的版本号，如果JVM本身的版本号 < class文件的版本号，JVM会认为加载不了这个class文件，会抛出我们经常见到的"`java.lang.UnsupportedClassVersionError: Bad version number in .class file`" **Error 错误**；反之，JVM会认为可以加载此class文件，继续加载此class文件。

JDK版本号信息对照表：

JDK版本	16进制版本号	十进制版本号
JDK8	00 00 00 34	52
JDK7	00 00 00 33	51
JDK6	00 00 00 32	50
JDK5	00 00 00 31	49
JDK1.4	00 00 00 30	48
JDK1.3	00 00 00 2F	47
JDK1.2	00 00 00 2E	46
JDK1.1	00 00 00 2D	45

小贴士：

> 1. 有时候我们在运行程序时会抛出这个**Error 错误**："*java.lang.UnsupportedClassVersionError: Bad version number in .class file*"。上面已经揭示了出现这个问题的原因，就是在于当前尝试加载class文件的JVM虚拟机的版本 低于class文件的版本。
>
>   解决方法：
>
>   a). 重新使用当前jvm编译源代码，然后再运行代码；
>
>   b). 将当前JVM虚拟机更新到class文件的版本。
> 2. 怎样查看class文件的版本号？可以借助于文本编辑工具，直接查看该文件的7，8个字节的值，确定class文件是什么版本的。
>
> 当然快捷的方式使用JDK自带的**javap**工具，如当前有Math.class 文件，进入此文件所在的目录，然后执行 "**javap -v Math**"，结果会类似如下所示：



常量池计数器

常量池是class文件中非常重要的结构，它描述着整个class文件的字面量信息。常量池是由一组constant_pool结构体数组组成的，而数组的大小则由常量池计数器指定。常量池计数器constant_pool_count 的值 =constant_pool表中的成员数+ 1。constant_pool表的索引值只有在大于 0且小于constant_pool_count时才会被认为是有效的。

注意事项：

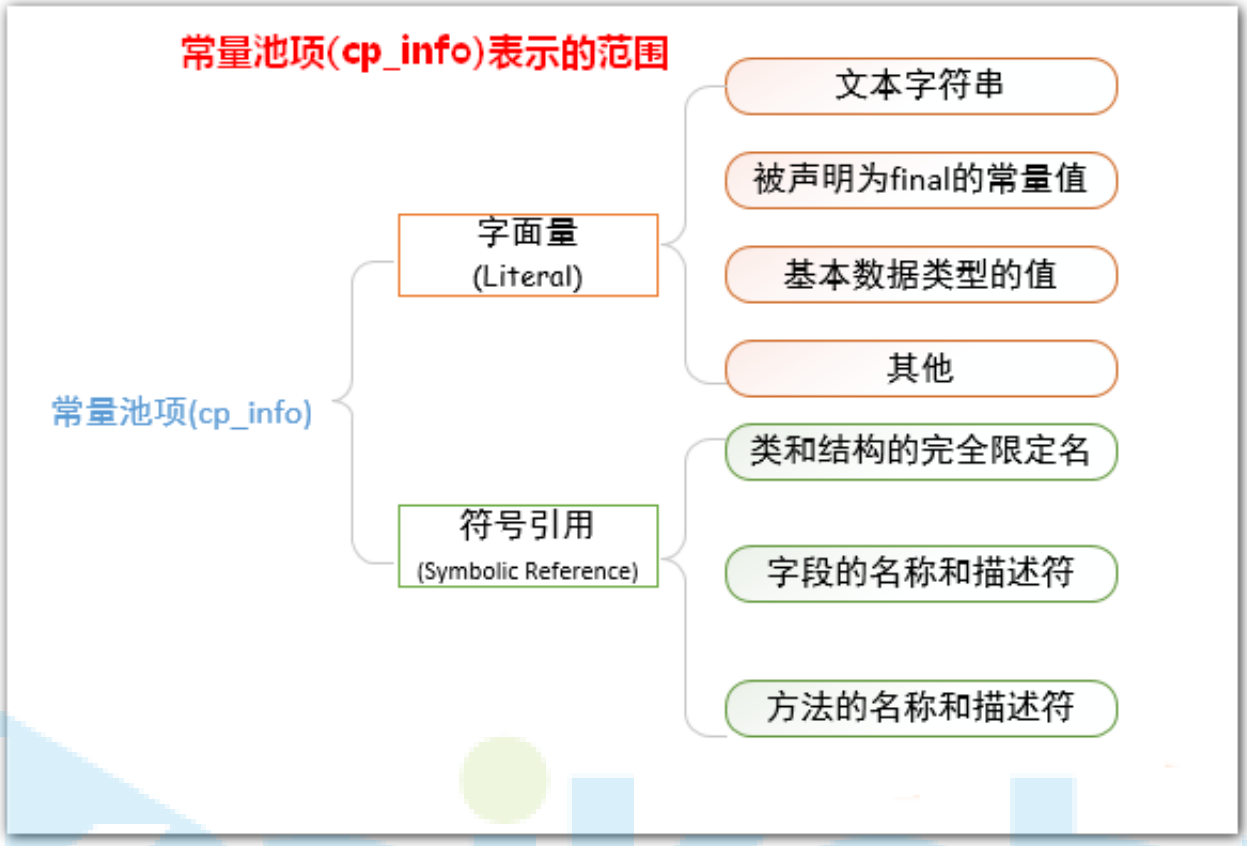
常量池计数器默认从1开始而不是从0开始：

当constant_pool_count = 1时，常量池中的cp_info个数为0；当constant_pool_count为n时，常量池中的cp_info个数为n-1。

原因：

在指定class文件规范的时候，将索引#0项常量空出来是有特殊考虑的，这样当：某些数据在特定的情况下想表达“不引用任何一个常量池项”的意思时，就可以将其引用的常量的索引值设置为#0来表示。

常量池数据区



访问标志

访问标志，access_flags 是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。

标记名	值	含义
ACC_PUBLIC	0x0001	可以被包的类外访问。
ACC_FINAL	0x0010	不允许有子类。
ACC_SUPER	0x0020	当用到 invokespecial 指令时，需要特殊处理的父类方法。
ACC_INTERFACE	0x0200	标识定义的是接口而不是类。
ACC_ABSTRACT	0x0400	不能被实例化。
ACC_SYNTHETIC	0x1000	标识并非 Java 源码生成的代码。
ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

注：

- 带有 ACC_SYNTHETIC 标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。
- 带有 ACC_ENUM 标志的类，意味着它或它的父类被声明为枚举类型。
- 带有 ACC_INTERFACE 标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个 Class 文件被设置了 ACC_INTERFACE 标志，那么同时也得设置 ACC_ABSTRACT 标志。同时它不能再设置 ACC_FINAL、ACC_SUPER 和 ACC_ENUM 标志。
- 注解类型必定带有 ACC_ANNOTATION 标记，如果设置了 ANNOTATION 标记，ACC_INTERFACE 也必须被同时设置。如果没有同时设置 ACC_INTERFACE 标记，那么这个 Class 文件可以具有表中的除 ACC_ANNOTATION 外的所有其它标记。当然 ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标记除外。
- ACC_SUPER 标志用于确定该 Class 文件里面的 invokespecial 指令使用的是哪一种执行语义。目前 Java 虚拟机的编译器都应当设置这个标志。ACC_SUPER 标记是为了向后兼容旧编译器编译的 Class 文件而存在的，在 JDK1.0.2 版本以前的编译器产生的 Class 文件中，access_flag 里面没有 ACC_SUPER 标志。同时，JDK1.0.2 前的 Java 虚拟机遇到 ACC_SUPER 标记会自动忽略它。
- 在表中没有使用的 access_flags 标志位是为未来扩充而预留的，这些预留的标志为在编译器中会被设置为 0，Java 虚拟机实现也会自动忽略它们.

类索引

类索引，this_class的值必须是对constant_pool表中项目的一个有效索引值。constant_pool表在这个索引处的项必须为CONSTANT_Class_info 类型常量，表示这个 Class 文件所定义的类或接口。

父类索引

父类索引，对于类来说，super_class 的值必须为 0 或者是对constant_pool 表中项目的一个有效索引值。

如果它的值不为 0，那 constant_pool 表在这个索引处的项必须为CONSTANT_Class_info 类型常量，表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的access_flag 中都不能带有ACC_FINAL 标记。对于接口来说，它的Class文件的super_class项的值必须是对constant_pool表中项目的一个有效索引值。constant_pool表在这个索引处的项必须为代表java.lang.Object 的 CONSTANT_Class_info 类型常量 。

如果 Class 文件的 super_class的值为 0，那这个Class文件只可能是定义的是java.lang.Object类，只有它是唯一没有父类的类。

接口计数器

接口计数器，interfaces_count的值表示当前类或接口的【直接父接口数量】。

接口信息数据区

接口表，interfaces[]数组中的每个成员的值必须是一个对constant_pool表中项目的一个有效索引值， 它的长度为 interfaces_count。每个成员interfaces[i] 必须为 CONSTANT_Class_info 类型常量，其中 【0 ≤ i <interfaces_count】。在interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即interfaces[0]对应的是源代码中最左边的接口。

字段计数器

字段计数器，fields_count的值表示当前 Class 文件 fields[]数组的成员个数。 fields[]数组中每一项都是一个field_info结构的数据项，它用于表示该类或接口声明的【类字段】或者【实例字段】。

字段信息数据区

字段表，fields[]数组中的每个成员都必须是一个fields_info结构的数据项，用于表示当前类或接口中某个字段的完整描述。 fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。

方法计数器

方法计数器， methods_count的值表示当前Class 文件 methods[]数组的成员个数。Methods[]数组中每一项都是一个 method_info 结构的数据项。

方法信息数据区

方法表，methods[] 数组中的每个成员都必须是一个 method_info 结构的数据项，用于表示当前类或接口中某个方法的完整描述。

如果某个method_info 结构的access_flags 项既没有设置 ACC_NATIVE 标志也没有设置 ACC_ABSTRACT 标志，那么它所对应的方法体就应当可以被 Java 虚拟机直接从当前类加载，而不需要引用其它类。

method_info结构可以表示类和接口中定义的所有方法，包括【实例方法】、【类方法】、【实例初始化方法】和【类或接口初始化方法】。

methods[]数组只描述【当前类或接口中声明的方法】，【不包括从父类或父接口继承的方法】。

```
Father{
  method(){

  }
}

Son extends Father{
  method(){

  }
}
Test1{
  main(){
    Father f = new Son();
    f.method(); // 方法调用（如何去选择方法表）
  }
}
```

属性计数器

属性计数器，attributes_count的值表示当前 Class 文件attributes表的成员个数。attributes表中每一项都是一个attribute_info 结构的数据项。

属性信息数据区

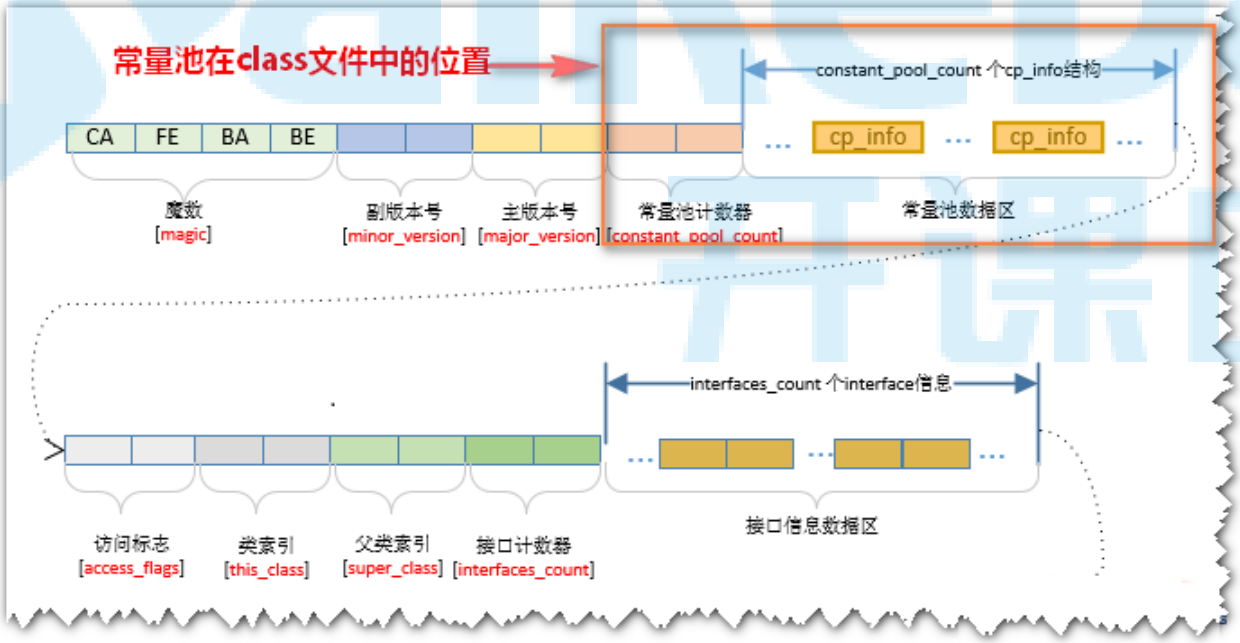
属性表，attributes 表的每个项的值必须是attribute_info结构。

在Java 7 规范里，Class文件结构中的attributes表的项包括下列定义的属性： InnerClasses 、EnclosingMethod 、 Synthetic 、Signature、SourceFile，SourceDebugExtension 、Deprecated、RuntimeVisibleAnnotations 、RuntimeInvisibleAnnotations以及BootstrapMethods属性。

对于支持 Class 文件格式版本号为 49.0 或更高的 Java 虚拟机实现，必须正确识别并读取attributes表中的Signature、RuntimeVisibleAnnotations和RuntimeInvisibleAnnotations属性。对于支持Class文件格式版本号为 51.0 或更高的 Java 虚拟机实现，必须正确识别并读取attributes表中的BootstrapMethods属性。Java 7 规范 要求任一 Java 虚拟机实现可以自动忽略 Class 文件的 attributes表中的若干 （甚至全部） 它不可识别的属性项。任何本规范未定义的属性不能影响Class文件的语义，只能提供附加的描述信息 。

class常量池理解

1.常量池在class文件的什么位置?

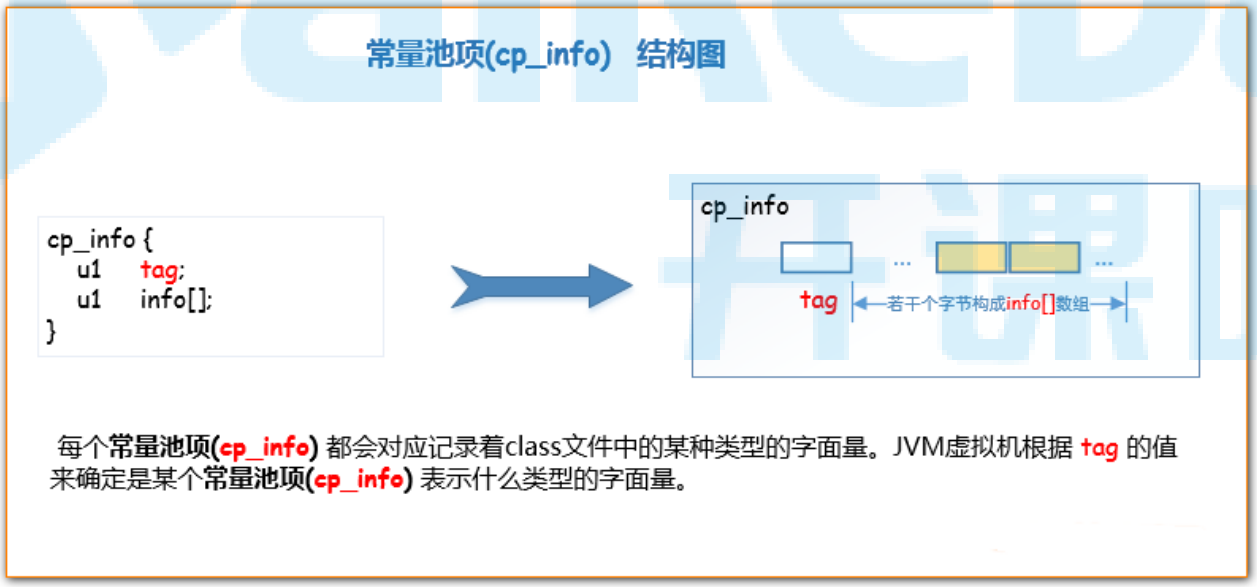


2.常量池的里面是怎么组织的?

cp_info：常量池项

constant_pool_count：常量池计算器

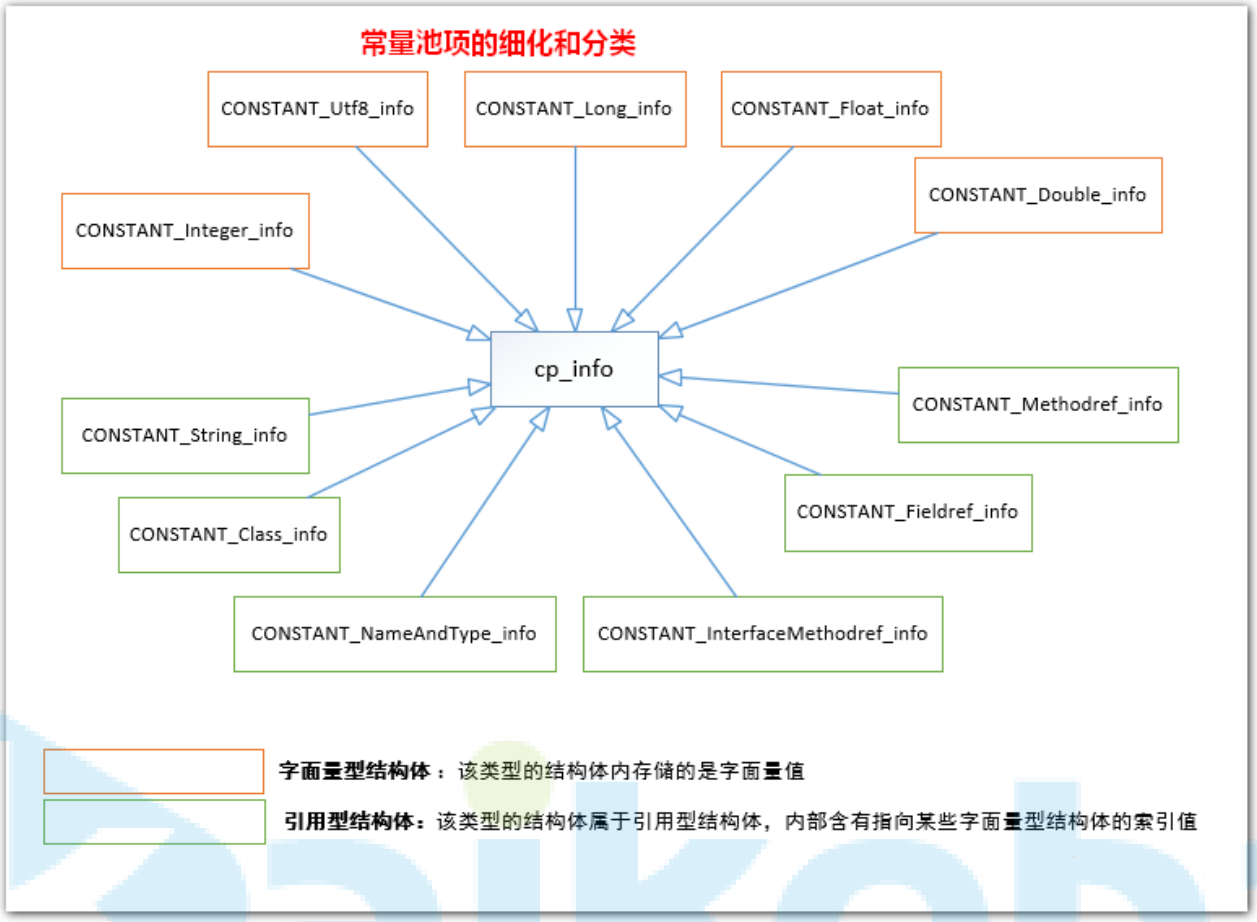
3.常量池项 (cp_info) 的结构是什么?



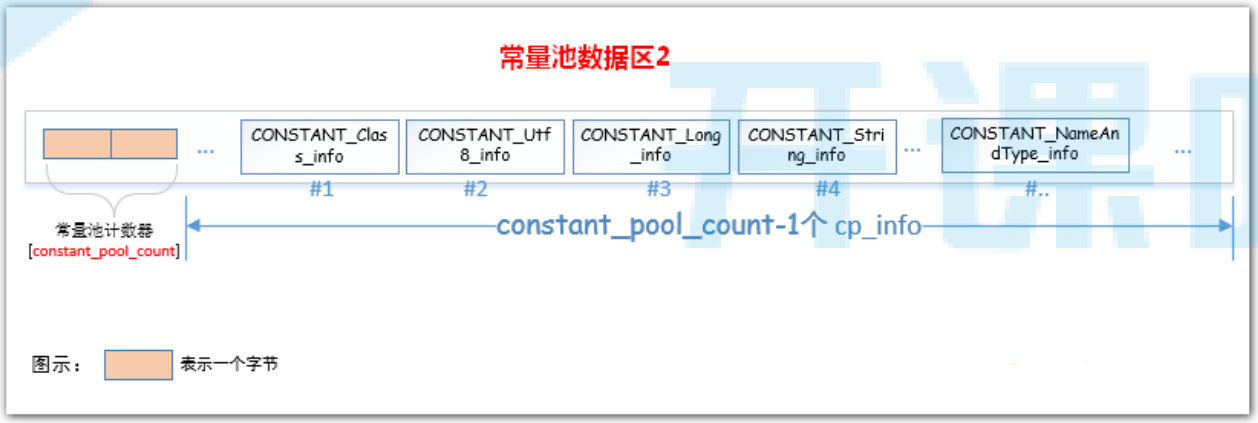
JVM虚拟机规定了不同的tag值和不同类型的字面量对应关系如下：

Tag值	表示的字面量	更细化的结构
1	用于表示字符串常量的值	CONSTANT_Utf8_info
3	表示 4 字节（int）的数值常量	CONSTANT_Integer_info
4	表示 4 字节（Float）的数值常量	CONSTANT_Float_info
5	表示 8 字节（Long）的数值常量	CONSTANT_Long_info
6	表示 8字节（double）的数值常量	CONSTANT_Double_info
7	表示类或接口的完全限定名	CONSTANT_Class_info
8	用于表示 java.lang.String 类型的常量对象	CONSTANT_String_info
9	表示类中的字段	CONSTANT_Fieldref_info
10	表示类中的方法	CONSTANT_Methodref_info
11	表示类所实现的接口的方法	CONSTANT_InterfaceMethodref_info
12	表示字段或方法的名称和类型	CONSTANT_NameAndType_info
15	表示方法句柄	CONSTANT_MethodHandle_info
16	表示方法类型	CONSTANT_MethodType_info
18	用于表示 invokedynamic 指令所使用到的引导方法（Bootstrap Method）、引导方法使用到动态调用名称（Dynamic Invocation Name）、参数和请求返回类型、以及可以选择性的附加被称为静态参数（Static Arguments）的常量序列	CONSTANT_InvokeDynamic_info

所以根据cp_info中的tag 不同的值，可以将cp_info 更细化为以下结构体：

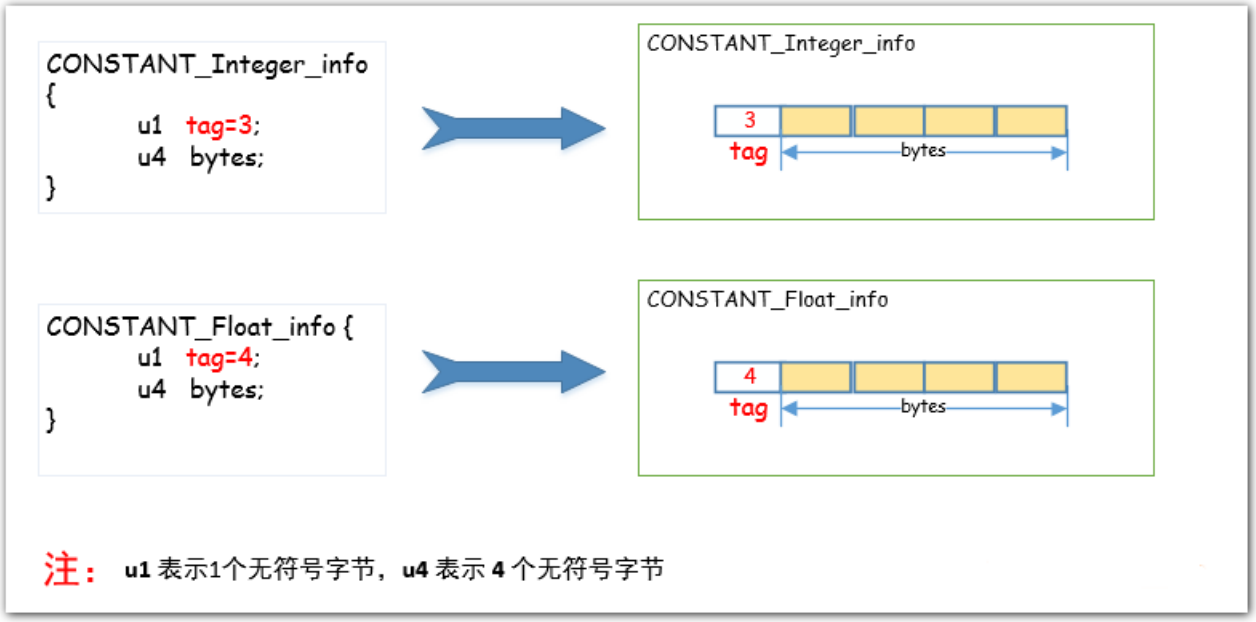


现在让我们看一下细化了的常量池的结构会是类似下图所示的样子：



4. int和float数据类型的常量在常量池中是怎样表示和存储的?

Java语言规范规定了 int类型和Float 类型的数据类型占用 **4** 个字节的空间。那么存在于class字节码文件中的该类型的常量是如何存储的呢?



举例：建下面的类 IntAndFloatTest.java，在这个类中，我们声明了五个变量，但是取值就两种int类型的**10** 和Float类型的**11f**。

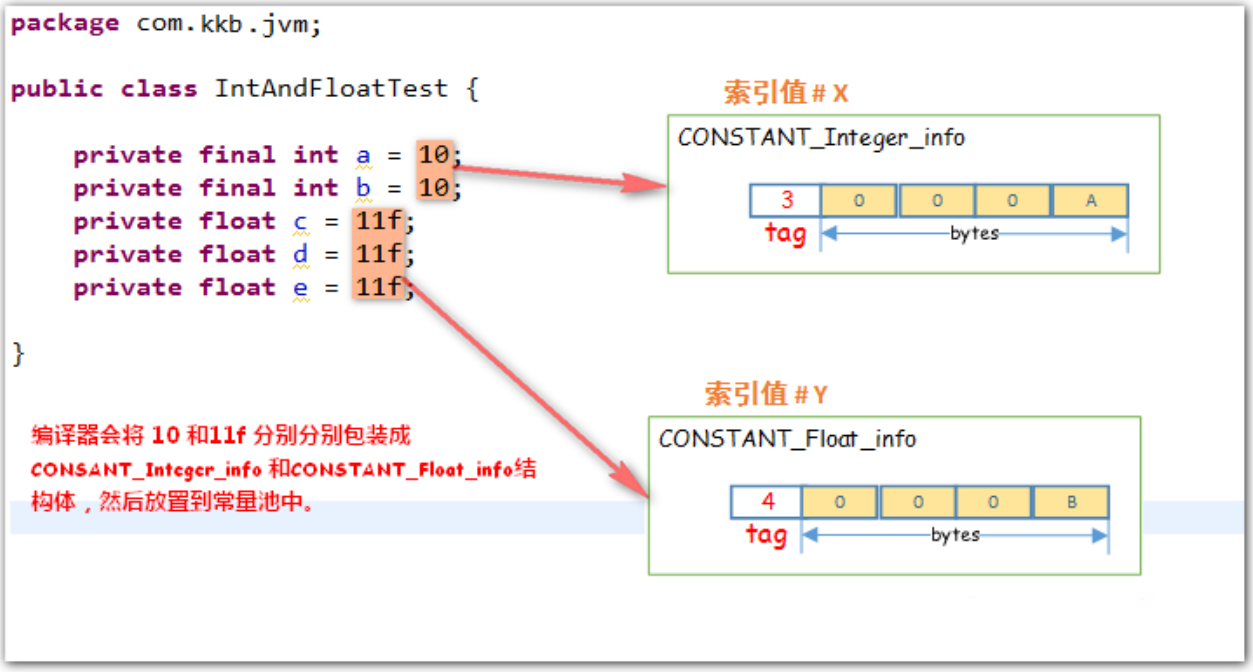
```
package com.kkb.jvm;
public class IntAndFloatTest {

    private final int a = 10;
    private final int b = 10;
    //private int c = 20;
    private float c = 11f;
    private float d = 11f;
    private float e = 11f;

}
```

然后用编译器编译成IntAndFloatTest.class字节码文件，我们通过**javap -v IntAndFloatTest** 指令来看一下其常量池中的信息，可以看到虽然我们在代码中写了两次**10** 和三次**11f**，但是常量池中，就只有一个常量**10** 和一个常量**11f**，如下图所示:

```
Constant pool:
   const #1 = class  #2; //    com/kkb/jvm/IntAndFloatTest
   const #2 = Asciz  com/kkb/jvm/IntAndFloatTest;
   const #3 = class  #4; //  java/lang/Object
   const #4 = Asciz  java/lang/Object;
   const #5 = Asciz  a;
   const #6 = Asciz  I;
   const #7 = Asciz  ConstantValue;
   const #8 = int  10;
   const #9 = Asciz  b;
   const #10 = Asciz c;
   const #11 = Asciz F;
   const #12 = Asciz d;
   const #13 = Asciz e;
   const #14 = Asciz <init>;
   const #15 = Asciz ()V;
   const #16 = Asciz Code;
   const #17 = Method  #3.#18; //  java/lang/Object."<init>":()V
   const #18 = NameAndType #14:#15;//  "<init>":()V
   const #19 = Field #1.#20; //    com/kkb/jvm/IntAndFloatTest.a:I
   const #20 = NameAndType #5:#6;//  a:I
   const #21 = Field #1.#22; //    com/kkb/jvm/IntAndFloatTest.b:I
   const #22 = NameAndType #9:#6;//  b:I
   const #23 = float 11.0f;
   const #24 = Field #1.#25; //    com/kkb/vm/IntAndFloatTest.c:F
   const #25 = NameAndType #10:#11;//  c:F
   const #26 = Field #1.#27; //    com/kkb/jvm/IntAndFloatTest.d:F
   const #27 = NameAndType #12:#11;//  d:F
   const #28 = Field #1.#29; //    com/kkb/jvm/IntAndFloatTest.e:F
   const #29 = NameAndType #13:#11;//  e:F
   const #30 = Asciz LineNumberTable;
   const #31 = Asciz LocalVariableTable;
   const #32 = Asciz this;
   const #33 = Asciz Lcom/kkb/jvm/IntAndFloat
   const #34 = Asciz SourceFile;
   const #35 = Asciz IntAndFloatTest.java;
```

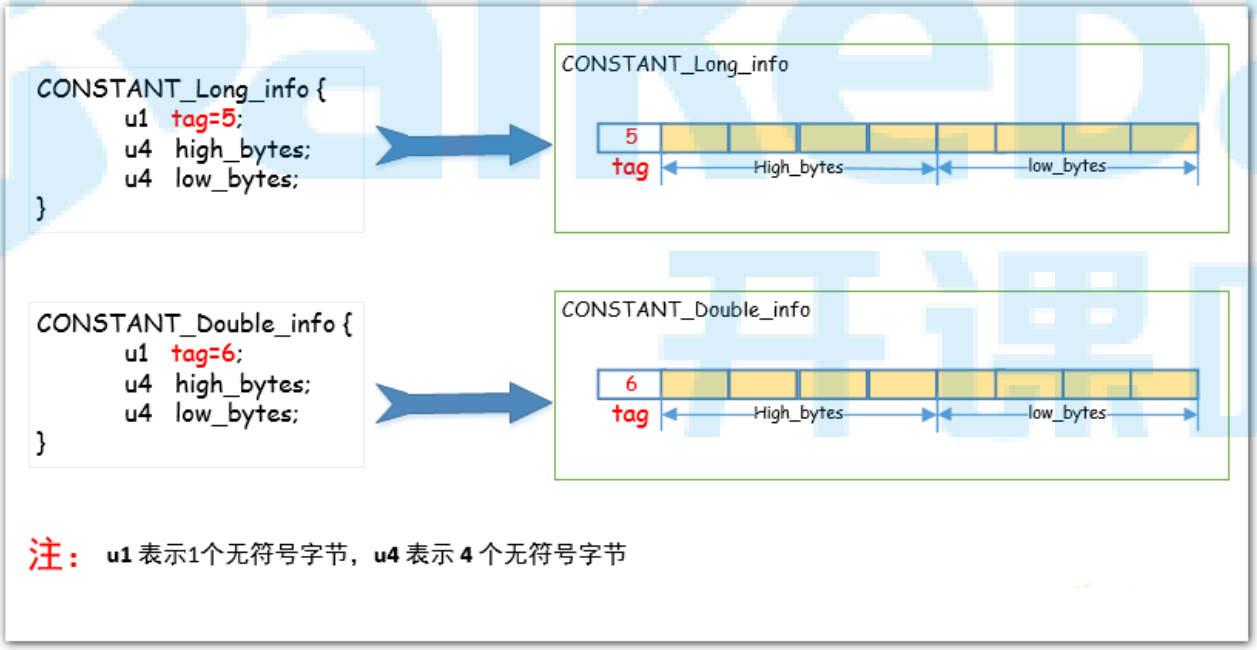
从结果上可以看到常量池第**#8** 个**常量池项(cp_info)** 就是CONSTANT_Integer_info，值为**10**；第**#23**个**常量池项(cp_info)** 就是CONSTANT_Float_info，值为**11f**。(*常量池中其他的东西先别纠结啦，我们后面会一一讲解的哦*)。

代码中所有用到 int 类型 **10** 的地方，会使用指向常量池的指针值**#8** 定位到第**#8** 个**常量池项(cp_info)**，即值为 **10**的结构体CONSTANT_Integer_info，而用到float类型的**11f**时，也会指向常量池的指针值#23来定位到第#23个**常量池项(cp_info)** 即值为**11f**的结构体CONSTANT_Float_info。如下图所示：



5. long和 double数据类型的常量在常量池中是怎样表示和存储的?

Java语言规范规定了 **long** 类型和 **double**类型的数据类型占用**8** 个字节的空间。那么存在于class 字节码文件中的该类型的常量是如何存储的呢?



举例：建下面的类 LongAndDoubleTest.java，在这个类中，我们声明了六个变量，但是取值就两种 **Long** 类型的**-6076574518398440533L** 和**Double** 类型的**10.1234567890D**。

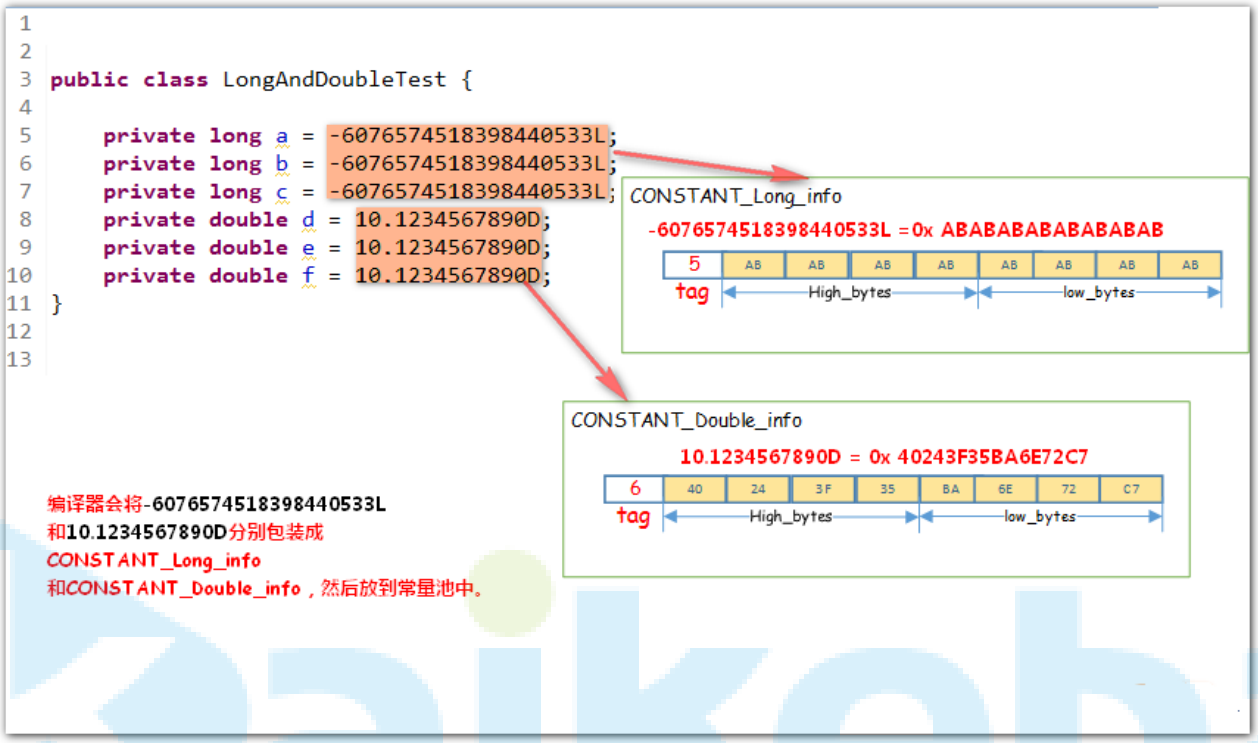
```java
package com.kkb.jvm;
  public class LongAndDoubleTest {

  private long a = -6076574518398440533L;
  private long b = -6076574518398440533L;
  private long c = -6076574518398440533L;
  private double d = 10.1234567890D;
  private double e = 10.1234567890D;
  private double f = 10.1234567890D;
 }
```

然后用编译器编译成 LongAndDoubleTest.class 字节码文件，我们通过**javap -v LongAndDoubleTest** 指令来看一下其常量池中的信息，可以看到虽然我们在代码中写了三次**-6076574518398440533L** 和三次**10.1234567890D**，但是常量池中，就只有一个常量**-6076574518398440533L** 和一个常量**10.1234567890D**，如下图所示:

```
Constant pool:
   const #1 = class  #2; //  com/kkb/jvm/LongAndDoubleTest
   const #2 = Asciz  com/kkb/jvm/LongAndDoubleTest;
   const #3 = class  #4; //  java/lang/Object
   const #4 = Asciz  java/lang/Object;
   const #5 = Asciz  a;
   const #6 = Asciz  J;
   const #7 = Asciz  b;
   const #8 = Asciz  c;
   const #9 = Asciz  d;
   const #10 = Asciz D;
   const #11 = Asciz e;
   const #12 = Asciz f;
   const #13 = Asciz <init>;
   const #14 = Asciz ()V;
   const #15 = Asciz Code;
   const #16 = Method  #3.#17; //  java/lang/Object."<init>":()V
   const #17 = NameAndType #13:#14;//  "<init>":()V
   const #18 = long  -6076574518398440533l;
   const #20 = Field #1.#21; //  com/kkb/jvm/LongAndDoubleTest.a:J
   const #21 = NameAndType #5:#6;//  a:J
   const #22 = Field #1.#23; //  com/kkb/jvm/LongAndDoubleTest.b:J
   const #23 = NameAndType #7:#6;//  b:J
   const #24 = Field #1.#25; //  com/kkb/jvm/LongAndDoubleTest.c:J
   const #25 = NameAndType #8:#6;//  c:J
   const #26 = double  10.123456789d;
   const #28 = Field #1.#29; //  com/kkb/jvm/LongAndDoubleTest.d:D
   const #29 = NameAndType #9:#10;//  d:D
   const #30 = Field #1.#31; //  com/kkb/jvm/LongAndDoubleTest.e:D
   const #31 = NameAndType #11:#10;//  e:D
   const #32 = Field #1.#33; //  com/kkb/jvm/LongAndDoubleTest.f:D
   const #33 = NameAndType #12:#10;//  f:D
   const #34 = Asciz LineNumberTable;
   const #35 = Asciz LocalVariableTable;
   const #36 = Asciz this;
   const #37 = Asciz Lcom/kkb/jvm/LongAndDoubleTest;;
   const #38 = Asciz SourceFile;
   const #39 = Asciz LongAndDoubleTest.java;
```

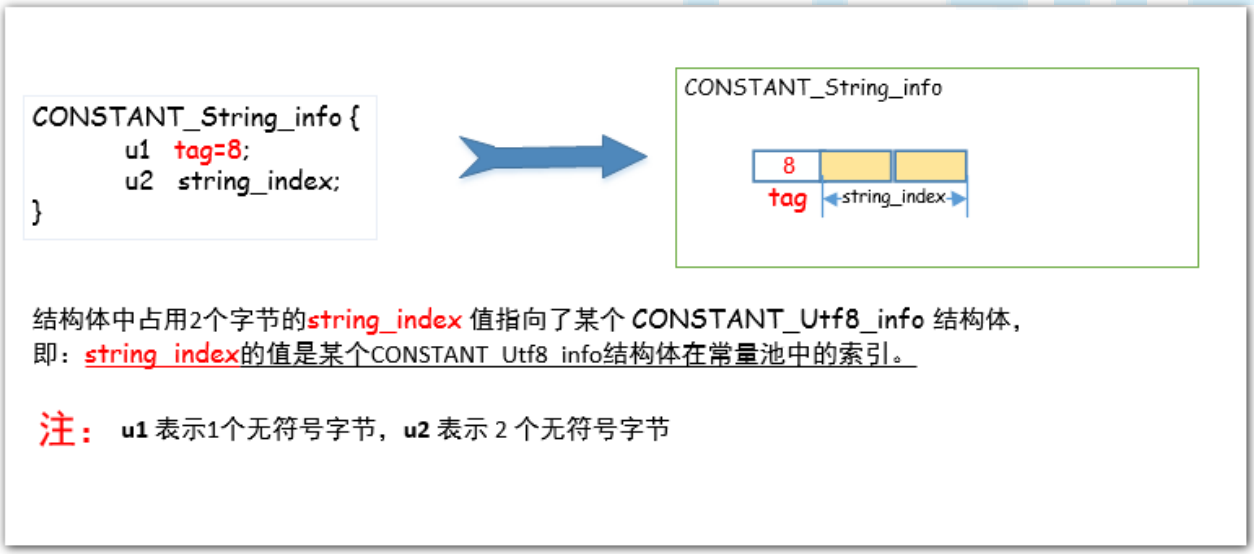
从结果上可以看到常量池第 **#18** 个**常量池项(cp_info)** 就是**CONSTANT_Long_info**，值为**-6076574518398440533L** ； 第 **#26**个**常量池项(cp_info)** 就是**CONSTANT_Double_info**，值为**10.1234567890D**。(常量池中其他的东西先别纠结啦，我们会面会一一讲解的哦)。

代码中所有用到 long 类型**-6076574518398440533L** 的地方，会使用指向常量池的指针值**#18** 定位到第**#18** 个**常量池项(cp_info)**，即值为**-6076574518398440533L** 的结构体**CONSTANT_Long_info**，而用到double类型的**10.1234567890D**时，也会指向常量池的指针值**#26**来定位到第 **#26** 个**常量池项(cp_info)** 即值为**10.1234567890D**的结构体**CONSTANT_Double_info**。如下图所示：

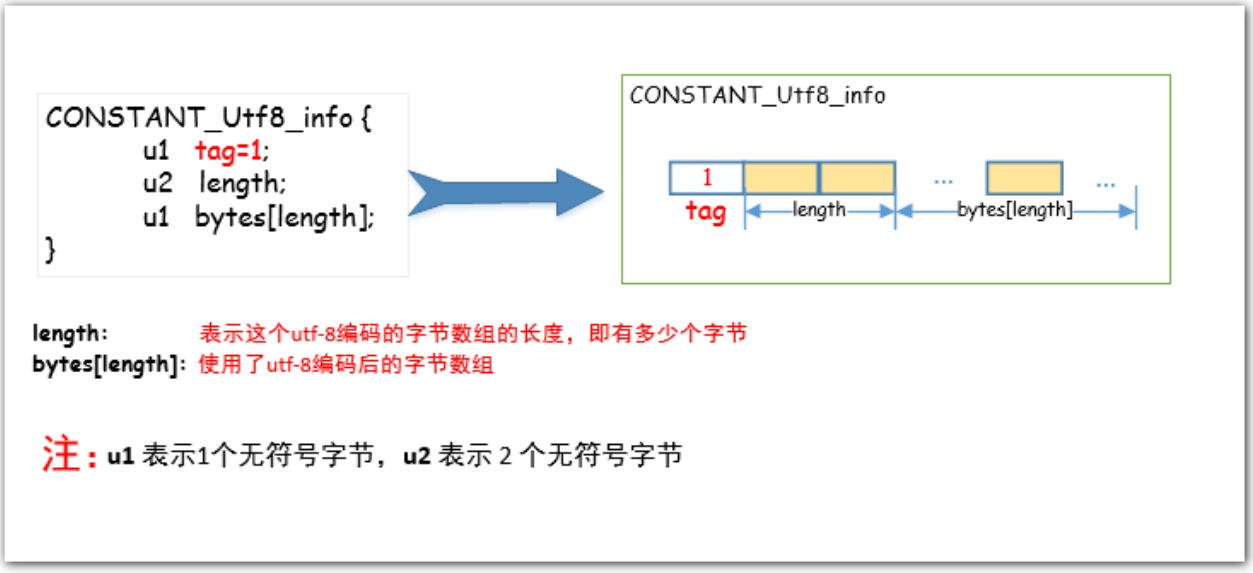


6. String类型的字符串常量在常量池中是怎样表示和存储的?

对于字符串而言，JVM会将字符串类型的字面量以UTF-8 编码格式存储到在class字节码文件中。这么说可能有点摸不着北，我们先从直观的Java源码中中出现的用双引号"" 括起来的字符串来看，在编译器编译的时候，都会将这些字符串转换成**CONSTANT_String_info**结构体，然后放置于常量池中。其结构如下所示：



如上图所示的结构体，**CONSTANT_String_info**结构体中的string_index的值指向了CONSTANT_Utf8_info结构体，而字符串的utf-8编码数据就在这个结构体之中。如下图所示：



请看一例，定义一个简单的StringTest.java类，然后在这个类里加一个"JVM原理" 字符串，然后，我们来看看它在class文件中是怎样组织的。

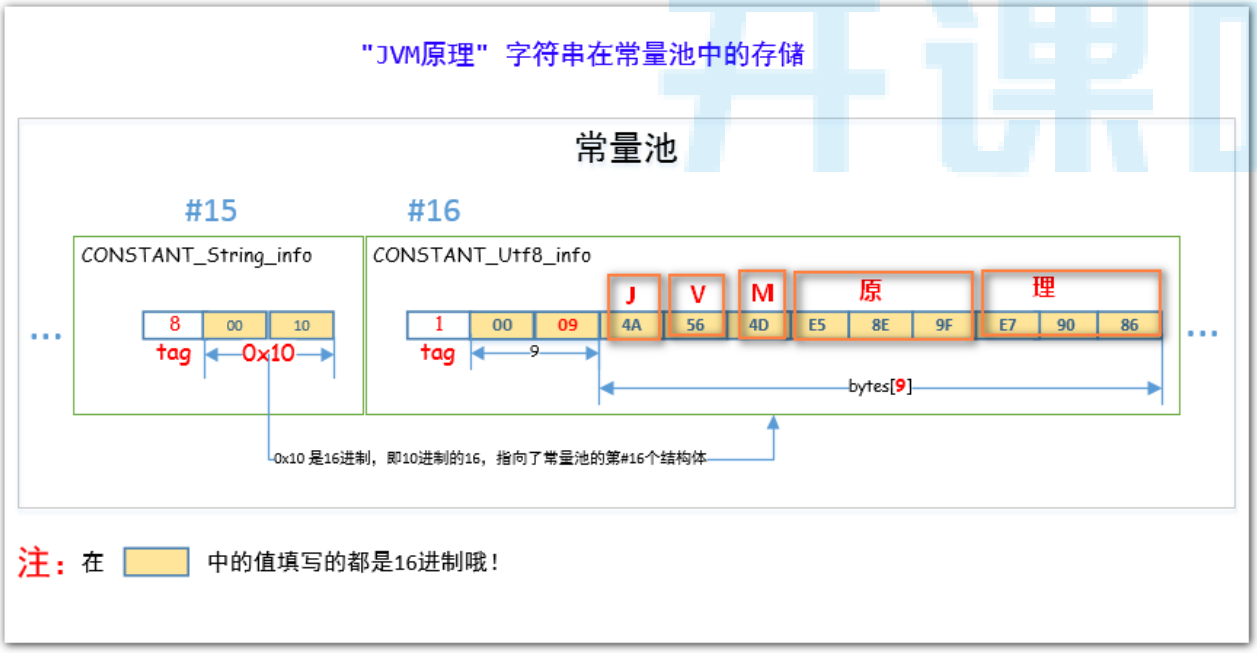
```
package com.kkb.jvm;

public class StringTest {
    private String s1 = "JVM原理";
    private String s2 = "JVM原理";
    private String s3 = "JVM原理";
    private String s4 = "JVM原理";
}
```

将Java源码编译成StringTest.class文件后，在此文件的目录下执行 javap -v StringTest 命令，会看到如下的常量池信息的轮廓：

```
const #1 = class  #2; //  com/kkb/jvm/StringTest
const #2 = Asciz  com/kkb/jvm/StringTest;
const #3 = class  #4; //  java/lang/Object
const #4 = Asciz  java/lang/Object;
const #5 = Asciz  s1;
const #6 = Asciz  Ljava/lang/String;;
const #7 = Asciz  s2;
const #8 = Asciz  s3;
const #9 = Asciz  s4;
const #10 = Asciz <init>;
const #11 = Asciz ()V;
const #12 = Asciz Code;
const #13 = Method  #3.#14; //  java/lang/Object."<init>":()V
const #14 = NameAndType #10:#11;//  "<init>":()V
const #15 = String  #16;  //  JVM原理 CONSTANT_String_info
const #16 = Asciz JVM原理;CONSTANT_Utf8_info
const #17 = Field #1.#18; //  com/kkb/jvm/StringTest.s1:Ljava/lang/String;
const #18 = NameAndType #5:#6;//  s1:Ljava/lang/String;
const #19 = Field #1.#20; //  com/kkb/jvm/StringTest.s2:Ljava/lang/String;
const #20 = NameAndType #7:#6;//  s2:Ljava/lang/String;
const #21 = Field #1.#22; //  com/kkb/jvm/StringTest.s3:Ljava/lang/String;
const #22 = NameAndType #8:#6;//  s3:Ljava/lang/String;
const #23 = Field #1.#24; //  com/kkb/jvm/StringTest.s4:Ljava/lang/String;
const #24 = NameAndType #9:#6;//  s4:Ljava/lang/String;
const #25 = Asciz LineNumberTable;
const #26 = Asciz LocalVariableTable;
const #27 = Asciz this;
const #28 = Asciz Lcom/kkb/jvm/StringTest;;
const #29 = Asciz SourceFile;
const #30 = Asciz StringTest.java;
```

(PS :使用javap -v 指令能看到易于我们阅读的信息，查看真正的字节码文件可以使用HEXWin、NOTEPAD++、UtraEdit 等工具。)

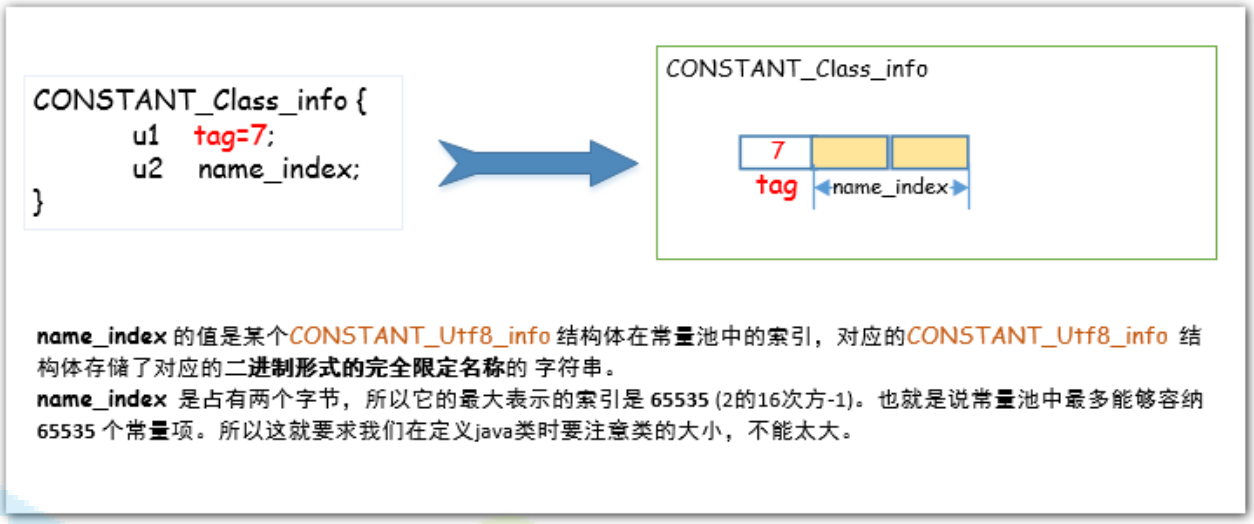
在面的图中，我们可以看到**CONSTANT_String_info**结构体位于常量池的第**#15**个索引位置。而存放"Java虚拟机原理" 字符串的 UTF-8编码格式的字节数组被放到**CONSTANT_Utf8_info**结构体中，该结构体位于常量池的第**#16**个索引位置。上面的图只是看了个轮廓，让我们再深入地看一下它们的组织吧。请看下图：



由上图可见："**JVM原理**"的UTF-8编码的数组是：4A564D E5 8E 9FE7 90 86，并且存入了**CONSTANT_Utf8_info**结构体中。

7. 类文件中定义的类名和类中使用到的类在常量池中是怎样被组织和存储的?

JVM会将某个Java 类中所有使用到了的**类的完全限定名** 以**二进制形式的完全限定名** 封装成**CONSTANT_Class_info**结构体中，然后将其放置到常量池里。**CONSTANT_Class_info** 的tag值为 **7** 。其结构如下：



> Tips：**类的完全限定名和二进制形式的完全限定名**
>
> 在某个Java源码中，我们会使用很多个类，比如我们定义了一个 **ClassTest**的类，并把它放到**com.kkb.jvm** 包下，则 **ClassTest**类的完全限定名为**com.kkb.jvm.ClassTest**，将JVM编译器将类编译成class文件后，此完全限定名在class文件中，是以二进制形式的完全限定名存储的，即它会把完全限定符的"."换成**"/"** ，即在class文件中存储的 **ClassTest**类的完全限定名称是"**com/kkb/jvm/ClassTest**"。因为这种形式的完全限定名是放在了class二进制形式的字节码文件中，所以就称之为 **二进制形式的完全限定名。**

举例，我们定义一个很简单的**ClassTest**类，来看一下常量池是怎么对类的完全限定名进行存储的。

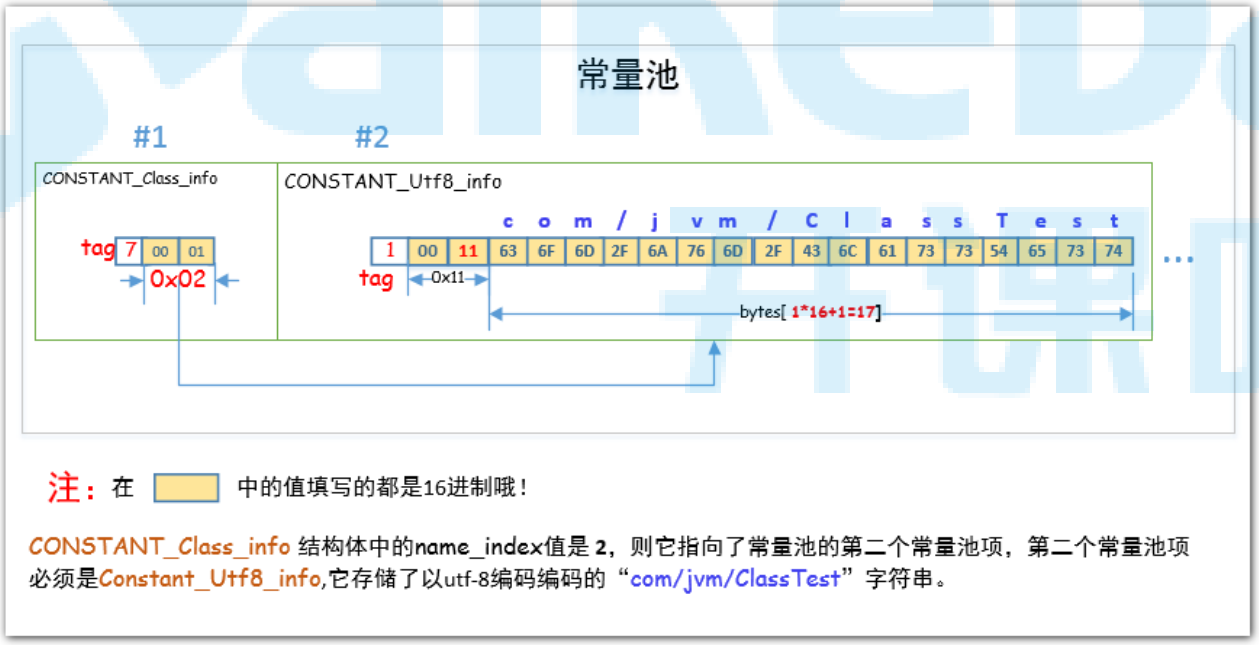
```
package com.jvm;
import  java.util.Date;
public class ClassTest {
 private Date date =new Date();
}
```

将Java源码编译成**ClassTest.class**文件后，在此文件的目录下执行 javap -v ClassTest 命令，会看到如下的常量池信息的轮廓：

```
Constant pool:
const #1 = class    #2; //  com/jvm/ClassTest
const #2 = Asciz    com/jvm/ClassTest;
const #3 = class    #4; //  java/lang/Object
const #4 = Asciz    java/lang/Object;
const #5 = Asciz    date;
const #6 = Asciz    Ljava/util/Date;;
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 = class   #13;    //  java/util/Date
const #13 = Asciz   java/util/Date;
const #14 = Method  #12.#11;    //  java/util/Date."<init>":()V
const #15 = Field   #1.#16; //  com/jvm/ClassTest.date:Ljava/util/Date;
const #16 = NameAndType #5:#6;//  date:Ljava/util/Date;
const #17 = Asciz   LineNumberTable;
const #18 = Asciz   LocalVariableTable;
const #19 = Asciz   this;
const #20 = Asciz   Lcom/jvm/ClassTest;;
const #21 = Asciz   SourceFile;
const #22 = Asciz   ClassTest.java;
```

Java规定所有的类都要继承自java.lang.Object类，即所有类都是java.lang.Object的子类。JVM在编译类的时候，即使我们没有显式地写上 *extends java.lang.Object*，JVM编译器在编译的时候也会自动帮我们加上。

如上图所示，在**ClassTest.class**文件的常量池中，共有 3 个**CONSTANT_Class_info**结构体，分别表示**ClassTest** 中用到的Class信息。 我们就看其中一个表示**com/jvm.ClassTest**的**CONSTANT_Class_info** 结构体。它在常量池中的位置是**#1**，它的name_index值为**#2**，它指向了常量池的第***2*** 个常量池项，如下所示:



注意：

> 对于某个类而言，其class文件中至少要有两个CONSTANT_Class_info常量池项，用来表示自己的类信息和其父类信息。(除了java.lang.Object类除外，其他的任何类都会默认继承自java.lang.Object)
> 如果类声明实现了某些接口，那么接口的信息也会生成对应的CONSTANT_Class_info常量池项。

除此之外，如果在类中使用到了其他的类，只有真正使用到了相应的类，JDK编译器才会将类的信息组成CONSTANT_Class_info常量池项放置到常量池中。如下图：

```
package com.kkb.jvm;
import java.util.Date;
public  class Other{
  private Date date;
  public Other()  {
    Date da;
  }
}
```

上述的Other的类，在JDK将其编译成class文件时，常量池中并没有java.util.Date对应的CONSTANT_Class_info常量池项，为什么呢?

在Other类中虽然定义了Date类型的两个变量date、da，但是JDK编译的时候，认为你只是声明了“Ljava/util/Date”类型的变量，并没有实际使用到Ljava/util/Date类。将类信息放置到常量池中的目的，是为了在后续的代码中有可能会反复用到它。很显然，JDK在编译Other类的时候，会解析到Date类有没有用到，发现该类在代码中就没有用到过，所以就认为没有必要将它的信息放置到常量池中了。

将上述的Other类改写一下，仅使用new Date()，如下图所示：

```
package com.kkb.jvm;
import java.util.Date;
public  class Other{
   public Other()
   {
       new Date();
   }
 }
```

这时候使用javap -v Other ，可以查看到常量池中有表示java/util/Date的常量池项：

```
Constant pool:
   #1 = Class              #2             //  com/jvm/Other
   #2 = Utf8               com/jvm/Other
   #3 = Class              #4             //  java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Code
   #8 = Methodref          #3.#9          //  java/lang/Object."<init>":()V
   #9 = NameAndType        #5:#6          //  "<init>":()V
  #10 = Class              #11            //  java/util/Date
  #11 = Utf8               java/util/Date
  #12 = Methodref          #10.#9         //  java/util/Date."<init>":()V
  #13 = Utf8               LineNumberTable
  #14 = Utf8               LocalVariableTable
  #15 = Utf8               this
  #16 = Utf8               Lcom/jvm/Other;
  #17 = Utf8               SourceFile
  #18 = Utf8               Other.java
{
```

总结：

1. 对于某个类或接口而言，其自身、父类和继承或实现的接口的信息会被直接组装成CONSTANT_Class_info常量池项放置到常量池中；
2. 类中或接口中使用到了其他的类，只有在类中实际使用到了该类时，该类的信息才会在常量池中有对应的CONSTANT_Class_info常量池项；
3. 类中或接口中仅仅定义某种类型的变量，JDK只会将变量的类型描述信息以UTF-8字符串组成CONSTANT_Utf8_info常量池项放置到常量池中，上面在类中的private Date date;JDK编译器只会将表示date的数据类型的“Ljava/util/Date”字符串放置到常量池中。

8.哪些字面量会进入常量池中?

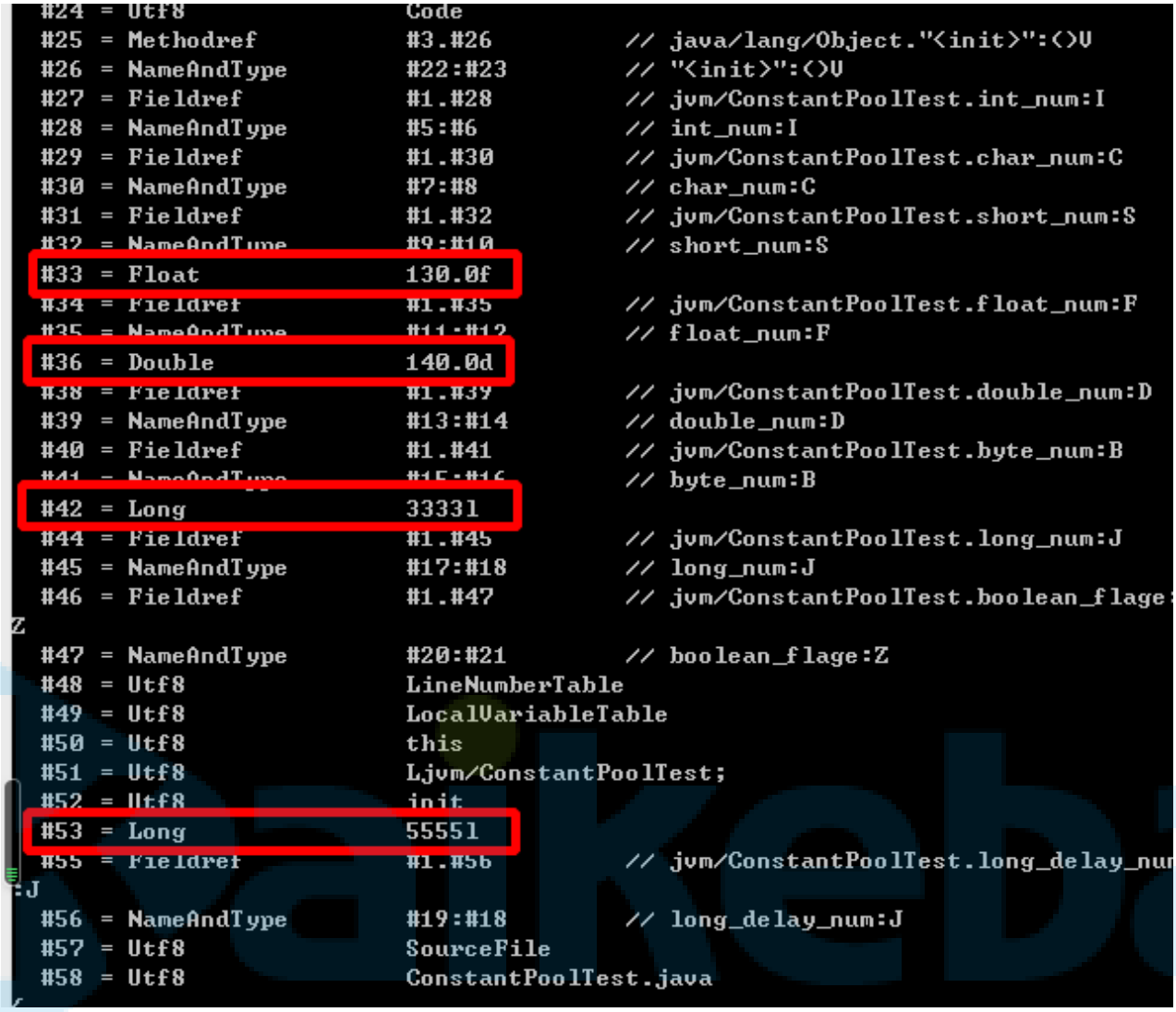
结论：

```
1. final类型的8种基本类型的值会进入常量池。
2. 非final类型（包括static的）的8种基本类型的值，只有double、float、long的值会进入常量池。
3. 常量池中包含的字符串类型字面量（双引号引起来的字符串值）。
```

测试代码：

```
public class Test{
    private int int_num = 110;
  private char char_num = 'a';
  private short short_num = 120;
  private float float_num = 130.0f;
  private double double_num = 140.0;
  private byte byte_num = 111;
  private long long_num = 3333L;
  private long long_delay_num;
  private boolean boolean_flage = true;

  public void init() {
    this.long_delay_num = 5555L;
  }
}
```

使用javap命令打印的结果如下：



class文件中的引用和特殊字符串

符号引用

符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。

例如，在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现。

符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。

在Java中，一个java类将会编译成一个class文件。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。

比如`org.simple.People类`引用了`org.simple.Language类`，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号`org.simple.Language`（假设是这个，当然实际中是由类似于CONSTANT_Class_info的常量来表示的）来表示Language类的地址。

各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用

直接引用可以是：

1. 直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）
2. 相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）
3. 一个能间接定位到目标的句柄

> 直接引用是和虚拟机的布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经被加载入内存中了。

引用替换的时机

符号引用替换为直接引用的操作发生在类加载过程(加载 -> 连接(验证、准备、解析) -> 初始化)中的解析阶段，会将符号引用转换(替换)为对应的直接引用，放入运行时常量池中。

后面我们会通过一些问题，来更深入的了解class常量池相关的知识点！！！

特殊字符串字面量

特殊字符串包括三种： 类的全限定名， 字段和方法的描述符， 特殊方法的方法名。 下面我们就分别介绍这三种特殊字符串。

类的全限定名

Object类，在源文件中的全限定名是`java.lang.Object`。

而class文件中的全限定名是将点号替换成“/” 。 也就是`java/lang/Object` 。

源文件中一个类的名字， 在class文件中是用全限定名表述的。

描述符

各类型的描述符

对于字段的数据类型，其描述符主要有以下几种

- **基本数据类型**（byte、char、double、float、int、long、short、boolean）：除 long 和 boolean，其他基本数据类型的描述符用对应单词的大写首字母表示。**long 用 J 表示，boolean 用 Z 表示**。
- **void**：描述符是 V。
- **对象类型**：描述符用字符`L`加上对象的全限定名表示，如 `String` 类型的描述符为`Ljava/lang/String`。
- **数组类型**：每增加一个维度则在对应的字段描述符前增加一个 `[` ，如一维数组 `int[]` 的描述符为`[I`，二维数组 `String[][]` 的描述符为 `[[Ljava/lang/String` 。

数据类型	描述符
byte	B
char	C
double	D
float	F
int	I
long	J
short	S
boolean	Z
特殊类型void	V
对象类型	“L” + 类型的全限定名 + “;”。如 `Ljava/lang/String;` 表示 `String` 类型
数组类型	若干个 `“[”` + 数组中元素类型的对应字符串，如一维数组 `int[]` 的描述符为 `[I`，二维数组 `String[][]` 的描述符为 `[[java/lang/String;`

字段描述符

字段的描述符就是字段的类型所对应的字符或字符串。

如：

```
int i 中， 字段i的描述符就是 I
Object o中， 字段o的描述符就是 Ljava/lang/Object;
double[][] d中， 字段d的描述符就是 [[D
```

方法描述符

方法的描述符比较复杂， 包括所有参数的类型列表和方法返回值。 它的格式是这样的：

```
(参数1类型 参数2类型 参数3类型 ...)返回值类型
```

注意事项：

> 不管是参数的类型还是返回值类型， 都是使用对应字符和对应字符串来表示的， 并且参数列表使用小括号括起来， 并且**各个参数类型之间没有空格， 参数列表和返回值类型之间也没有空格。**

方法描述符举例说明如下：

方法描述符	方法声明
()I	int getSize()
()Ljava/lang/String;	String toString()
([Ljava/lang/String;)V	void main(String[] args)
()V	void wait()
(JI)V	void wait(long timeout, int nanos)
(ZILjava/lang/String;II)Z	boolean regionMatches(boolean ignoreCase, int toOffset, String other, int ooffset, int len)
([BII)I	int read(byte[] b, int off, int len)
()[[Ljava/lang/Object;	Object[][] getObjectArray()

特殊方法的方法名

首先要明确一下， 这里的特殊方法是指的类的构造方法和类型初始化方法。

构造方法就不用多说了， 至于**类型的初始化方法， 对应到源码中就是静态初始化块。** 也就是说， 静态初始化块， 在class文件中是以一个方法表述的， 这个方法同样有方法描述符和方法名，具体如下：

- 类的构造方法的方法名使用字符串 <init> 表示
- 静态初始化方法的方法名使用字符串 <clinit> 表示。
- 除了这两种特殊的方法外， 其他普通方法的方法名， 和源文件中的方法名相同。

总结

1. **方法和字段的描述符中， 不包括字段名和方法名**， 字段描述符中只包括字段类型， 方法描述符中只包括参数列表和返回值类型。
2. **无论method()是静态方法还是实例方法，它的方法描述符都是相同的。**尽管实例方法除了传递自身定义的参数，还需要额外传递参数this，但是这一点不是由方法描述符来表达的。参数this的传递，是由Java虚拟机实现在调用实例方法所使用的指令中实现的隐式传递。

通过javap命令分析java指令

javap命令简述

javap是jdk自带的反解析工具。它的作用就是根据class字节码文件，反解析出当前类对应的code区（汇编指令）、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。
当然这些信息中，有些信息（如本地变量表、指令和代码行偏移量映射表、常量池中方法的参数名称等等）需要在使用javac编译成class文件时，指定参数才能输出，比如，你直接javac xx.java，就不会在生成对应的局部变量表等信息，如果你使用javac -g xx.java就可以生成所有相关信息了。如果你使用的eclipse，则默认情况下，eclipse在编译时会帮你生成局部变量表、指令和代码行偏移量映射表等信息的。
通过反编译生成的汇编代码，我们可以深入的了解java代码的工作机制。比如我们可以查看i++；这行代码实际运行时是先获取变量i的值，然后将这个值加1，最后再将加1后的值赋值给变量i。
通过局部变量表，我们可以查看局部变量的作用域范围、所在槽位等信息，甚至可以看到槽位复用等信息。

javap的用法格式：

```
javap <options> <classes>
```

其中classes就是你要反编译的class文件。
在命令行中直接输入javap或javap -help可以看到javap的options有如下选项：

```
 -help  --help  -?        输出此用法消息
 -version                 版本信息，其实是当前javap所在jdk的版本信息，不是class在哪个
jdk下生成的。
 -v  -verbose             输出附加信息（包括行号、本地变量表，反汇编等详细信息）
 -l                         输出行号和本地变量表
 -public                    仅显示公共类和成员
 -protected               显示受保护的/公共类和成员
 -package                 显示程序包/受保护的/公共类 和成员 (默认)
 -p  -private             显示所有类和成员
 -c                       对代码进行反汇编
 -s                       输出内部类型签名
 -sysinfo                 显示正在处理的类的系统信息 (路径， 大小， 日期， MD5 散列)
 -constants               显示静态最终常量
 -classpath <path>        指定查找用户类文件的位置
 -bootclasspath <path>    覆盖引导类文件的位置
```

一般常用的是-v -l -c三个选项。

- **javap -v classxx**，不仅会输出行号、本地变量表信息、反编译汇编代码，还会输出当前类用到的常量池等信息。
- **javap -l** 会输出行号和本地变量表信息。
- **javap -c** 会对当前class字节码进行反编译生成汇编代码。

查看汇编代码时，需要知道里面的jvm指令，可以参考官方文档：
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html
另外通过jclasslib工具也可以看到上面这些信息，而且是可视化的，效果更好一些。

javap测试及内容详解

前面已经介绍过javap输出的内容有哪些，东西比较多，这里主要介绍其中code区(汇编指令)、局部变量表和代码行偏移映射三个部分。
如果需要分析更多的信息，可以使用**javap -v**进行查看。
另外，为了更方便理解，所有汇编指令不单拎出来讲解，而是在反汇编代码中以注释的方式讲解。

下面写段代码测试一下：
例子1：分析一下下面的代码反汇编之后结果：

```
public class TestDate {

    private int count = 0;

    public static void main(String[] args) {
        TestDate testDate = new TestDate();
        testDate.test1();
    }

    public void test1(){
        Date date = new Date();
        String name1 = "wangerbei";
        test2(date, name1);
        System.out.println(date+name1);
    }

    public void test2(Date dateP, String name2){
        dateP = null;
        name2 = "zhangsan";
    }

    public void test3(){
        count++;
    }

    public void  test4(){
        int a = 0;
        {
            int b = 0;
            b = a+1;
        }
        int c = a+1;
    }
}
```

上面代码通过JAVAC -g 生成class文件，然后通过javap命令对字节码进行反汇编：

```
$ javap -c -l TestDate
```

得到下面内容(指令等部分是我参照着官方文档总结的)：

```
Warning: Binary file TestDate contains com.justest.test.TestDate
Compiled from "TestDate.java"
public class com.justest.test.TestDate {
  //默认的构造方法，在构造方法执行时主要完成一些初始化操作，包括一些成员变量的初始化赋值等操作
  public com.justest.test.TestDate();
    Code:
       0: aload_0 //从本地变量表中加载索引为0的变量的值，也即this的引用，压入栈
       1: invokespecial #10  //出栈，调用java/lang/Object."<init>":()V 初始化对象，就是this指定的对象的<init>方法完成初始化
       4: aload_0  // 4到6表示，调用this.count = 0，也即为count复制为0。这里this引用入栈
       5: iconst_0 //将常量0，压入到操作数栈
       6: putfield //出栈前面压入的两个值（this引用，常量值0），  将0取出，并赋值给count
       9: return
//指令与代码行数的偏移对应关系，每一行第一个数字对应代码行数，第二个数字对应前面code中指令前面的数字
    LineNumberTable:
      line 5: 0
      line 7: 4
      line 5: 9
    //局部变量表，start+length表示这个变量在字节码中的生命周期起始和结束的偏移位置（this生命周期从头0到结尾10），slot就是这个变量在局部变量表中的槽位（槽位可复用），name就是变量名称，Signatur局部变量类型描述
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   Lcom/justest/test/TestDate;

  public static void main(java.lang.String[]);
    Code:
// new指令，创建一个class com/justest/test/TestDate对象，new指令并不能完全创建一个对象，对象只有在初，只有在调用初始化方法完成后（也就是调用了invokespecial指令之后），对象才创建成功，
       0: new  //创建对象，并将对象引用压入栈
       3: dup //将操作数栈定的数据复制一份，并压入栈，此时栈中有两个引用值
       4: invokespecial #20  //pop出栈引用值，调用其构造函数，完成对象的初始化
       7: astore_1 //pop出栈引用值，将其（引用）赋值给局部变量表中的变量testDate
       8: aload_1  //将testDate的引用值压入栈，因为testDate.test1();调用了testDate，这里使用aload_1从局部变量表中获得对应的变量testDate的值并压入操作数栈
       9: invokevirtual #21 // Method test1:()V  引用出栈，调用testDate的test1()方法
      12: return //整个main方法结束返回
    LineNumberTable:
```

```
      line 10: 0
      line 11: 8
      line 12: 12
    //局部变量表，testDate只有在创建完成并赋值后，才开始声明周期
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
         0      13     0  args   [Ljava/lang/String;
         8       5     1 testDate   Lcom/justest/test/TestDate;

  public void test1();
    Code:
       0: new           #27                 // 0到7创建Date对象，并赋值给date变量
       3: dup
       4: invokespecial #29                 // Method java/util/Date."<init>":
()V
       7: astore_1
       8: ldc           #30     // String wangerbei，将常量“wangerbei”压入栈
      10: astore_2  //将栈中的“wangerbei”pop出，赋值给name1
      11: aload_0 //11到14，对应test2(date，name1);默认前面加this.
      12: aload_1 //从局部变量表中取出date变量
      13: aload_2 //取出name1变量
      14: invokevirtual #32                 // Method test2:
(Ljava/util/Date;Ljava/lang/String;)V  调用test2方法
  // 17到38对应System.out.println(date+name1);
      17: getstatic     #36                 // Field
java/lang/System.out:Ljava/io/PrintStream;
  //20到35是jvm中的优化手段，多个字符串变量相加，不会两两创建一个字符串对象，而使用
StringBuilder来创建一个对象
      20: new           #42                 // class java/lang/StringBuilder
      23: dup
      24: invokespecial #44                 // Method java/lang/StringBuilder."
<init>":()V
      27: aload_1
      28: invokevirtual #45                 // Method
java/lang/StringBuilder.append:(Ljava/lang/Object;)Ljava/lang/StringBuilder;
      31: aload_2
      32: invokevirtual #49                 // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      35: invokevirtual #52                 // Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
      38: invokevirtual #56                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V  invokevirtual指令表示基于类调用
方法
      41: return
    LineNumberTable:
      line 15: 0
      line 16: 8
      line 17: 11
      line 18: 17
```

```
      line 19: 41
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      42     0  this   Lcom/justest/test/TestDate;
          8      34     1  date   Ljava/util/Date;
         11      31     2 name1   Ljava/lang/String;

  public void test2(java.util.Date,  java.lang.String);
    Code:
       0: aconst_null //将一个null值压入栈
       1: astore_1 //将null赋值给dateP
       2: ldc           #66       // String zhangsan 从常量池中取出字符
串“zhangsan”压入栈中
       4: astore_2 //将字符串赋值给name2
       5: return
    LineNumberTable:
      line 22: 0
      line 23: 2
      line 24: 5
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lcom/justest/test/TestDate;
          0       6     1 dateP   Ljava/util/Date;
          0       6     2 name2   Ljava/lang/String;

  public void test3();
    Code:
       0: aload_0 //取出this，压入栈
       1: dup   //复制操作数栈栈顶的值，并压入栈，此时有两个this对象引用值在操作数组栈
       2: getfield #12// Field count:I this出栈，并获取其count字段，然后压入栈，此时
栈中有一个this和一个count的值
       5: iconst_1 //取出一个int常量1，压入操作数栈
       6: iadd  // 从栈中取出count和1，将count值和1相加，结果入栈
       7: putfield      #12 // Field count:I  一次弹出两个，第一个弹出的是上一步计算
值，第二个弹出的this，将值赋值给this的count字段
      10: return
    LineNumberTable:
      line 27: 0
      line 28: 10
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/justest/test/TestDate;
 public void test4();
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_1
```

```
        5: iconst_1
        6: iadd
        7: istore_2
        8: iload_1
        9: iconst_1
       10: iadd
       11: istore_2
       12: return
    LineNumberTable:
      line 33: 0
      line 35: 2
      line 36: 4
      line 38: 8
      line 39: 12
    //看下面，b和c的槽位slot一样，这是因为b的作用域就在方法块中，方法块结束，局部变量表中的槽位就被释放，后面的变量就可以复用这个槽位
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      13     0  this   Lcom/justest/test/TestDate;
          2      11     1     a   I
          4       4     2     b   I
         12       1     2     c   I
}
```

例子2：下面一个例子
先有一个User类：

```
public class User {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

然后写一个操作User对象的测试类：

```
public class TestUser {

    private int count;

    public void test(int a){
        count = count + a;
    }

    public User initUser(int age, String name){
        User user = new User();
        user.setAge(age);
        user.setName(name);
        return user;
    }

    public void changeUser(User user, String newName){
        user.setName(newName);
    }
}
```

先**javac -g** 编译成class文件。
然后对TestUser类进行反汇编：
`$ javap -c -l TestUser`
得到反汇编结果如下：

```
Warning: Binary file TestUser contains com.justest.test.TestUser
Compiled from "TestUser.java"

public class com.justest.test.TestUser {

//默认的构造函数
  public com.justest.test.TestUser();

    Code:
       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/justest/test/TestUser;
```

```
  public void test(int);

    Code:
       0: aload_0 //取this对应的对应引用值，压入操作数栈
       1: dup //复制栈顶的数据，压入栈，此时栈中有两个值，都是this对象引用
       2: getfield      #18 // 引用出栈，通过引用获得对应count的值，并压入栈
       5: iload_1 //从局部变量表中取得a的值，压入栈中
       6: iadd //弹出栈中的count值和a的值，进行加操作，并将结果压入栈
       7: putfield      #18 // 经过上一步操作后，栈中有两个值，栈顶为上一步操作结果，栈顶下面是this引用，这一步putfield指令，用于将栈顶的值赋值给引用对象的count字段
      10: return //return void

    LineNumberTable:
      line 8: 0
      line 9: 10

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/justest/test/TestUser;
          0      11     1     a   I

  public com.justest.test.User initUser(int,  java.lang.String);

    Code:
       0: new           #23   // class com/justest/test/User 创建User对象，并将引用压入栈
       3: dup //复制栈顶值，再次压入栈，栈中有两个User对象的地址引用
       4: invokespecial #25   // Method com/justest/test/User."<init>":()V 调用user对象初始化
       7: astore_3 //从栈中pop出User对象的引用值，并赋值给局部变量表中user变量
       8: aload_3 //从局部变量表中获得user的值，也就是User对象的地址引用，压入栈中
       9: iload_1 //从局部变量表中获得a的值，并压入栈中，注意aload和iload的区别，一个取值是对象引用，一个是取int类型数据
      10: invokevirtual #26  // Method com/justest/test/User.setAge:(I)V 操作数栈pop出两个值，一个是User对象引用，一个是a的值，调用setAge方法，并将a的值传给这个方法，setAge操作的就是堆中对象的字段了
      13: aload_3 //同7，压入栈
      14: aload_2 //从局部变量表取出name，压入栈
      15: invokevirtual #29  // MethodUser.setName:(Ljava/lang/String;)V 操作数栈pop出两个值，一个是User对象引用，一个是name的值，调用setName方法，并将a的值传给这个方法，setName操作的就是堆中对象的字段了
      18: aload_3 //从局部变量取出User引用，压入栈
      19: areturn //areturn指令用于返回一个对象的引用，也就是上一步中User的引用，这个返回值将会被压入调用当前方法的那个方法的栈中objectref is popped from the operand stack of the current frame ([§2.6](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.6)) and pushed onto the operand stack of the frame of the invoker

    LineNumberTable:
```

```
      line 12: 0
      line 13: 8
      line 14: 13
      line 15: 18

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      20     0  this   Lcom/justest/test/TestUser;
          0      20     1   age   I
          0      20     2  name   Ljava/lang/String;
          8      12     3  user   Lcom/justest/test/User;

  public void changeUser(com.justest.test.User,  java.lang.String);

    Code:
       0: aload_1 //局部变量表中取出this，也即TestUser对象引用，压入栈
       1: aload_2 //局部变量表中取出newName，压入栈
       2: invokevirtual #29 // Method User.setName:(Ljava/lang/String;)V pop出栈newName值和TestUser引用，调用其setName方法，并将newName的值传给这个方法
       5: return

    LineNumberTable:
      line 19: 0
      line 20: 5

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lcom/justest/test/TestUser;
          0       6     1  user   Lcom/justest/test/User;
          0       6     2 newName   Ljava/lang/String;

public static void main(java.lang.String[]);

    Code:
       0: new      #1 // class com/justest/test/TestUser 创建TestUser对象，将引用压入栈
       3: dup //复制引用，压入栈
       4: invokespecial #43   // Method "<init>":()V 引用值出栈，调用构造方法，对象初始化
       7: astore_1 //引用值出栈，赋值给局部变量表中变量tu
       8: aload_1 //取出tu值，压入栈
       9: bipush    10 //将int值10压入栈
      11: ldc           #44   // String wangerbei 从常量池中取出“wangerbei” 压入栈
      13: invokevirtual #46    // Method initUser(ILjava/lang/String;)Lcom/justest/test/User; 调用tu的initUser方法，并返回User对象 ，出栈三个值：tu引用，10和“wangerbei”，并且initUser方法的返回值，即User的引用，也会被压入栈中，参考前面initUser中的areturn指令
      16: astore_2 //User引用出栈，赋值给user变量
      17: aload_1 //取出tu值，压入栈
```

```
      18: aload_2 //取出user值，压入栈
      19: ldc           #48      // String lisi 从常量池中取出“lisi”压入栈
      21: invokevirtual #50      // Method changeUser:
(Lcom/justest/test/User;Ljava/lang/String;)V 调用tu的changeUser方法，并将user引用和
lisi传给这个方法
      24: return //return void

 LineNumberTable:
      line 23: 0
      line 24: 8
      line 25: 17
      line 26: 24

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
             0      25     0  args   [Ljava/lang/String;
             8      17     1    tu   Lcom/justest/test/TestUser;
            17       8     2  user   Lcom/justest/test/User;

}
```

总结

1、通过javap命令可以查看一个java类反汇编、常量池、变量表、指令代码行号表等等信息。

2、平常，我们比较关注的是java类中每个方法的反汇编中的指令操作过程，这些指令都是顺序执行的，可以参考官方文档查看每个指令的含义，很简单：

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.areturn

3、通过对前面两个例子代码反汇编中各个指令操作的分析，可以发现，一个方法的执行通常会涉及下面几块内存的操作：

（1）**java栈**：局部变量表、操作数栈。这些操作基本上都值操作。
（2）**java堆**：通过对象的地址引用去操作。
（3）**常量池**。
（4）其他如**帧数据区、方法区**（jdk1.8之前，常量池也在方法区）等部分，测试中没有显示出来，这里说明一下。

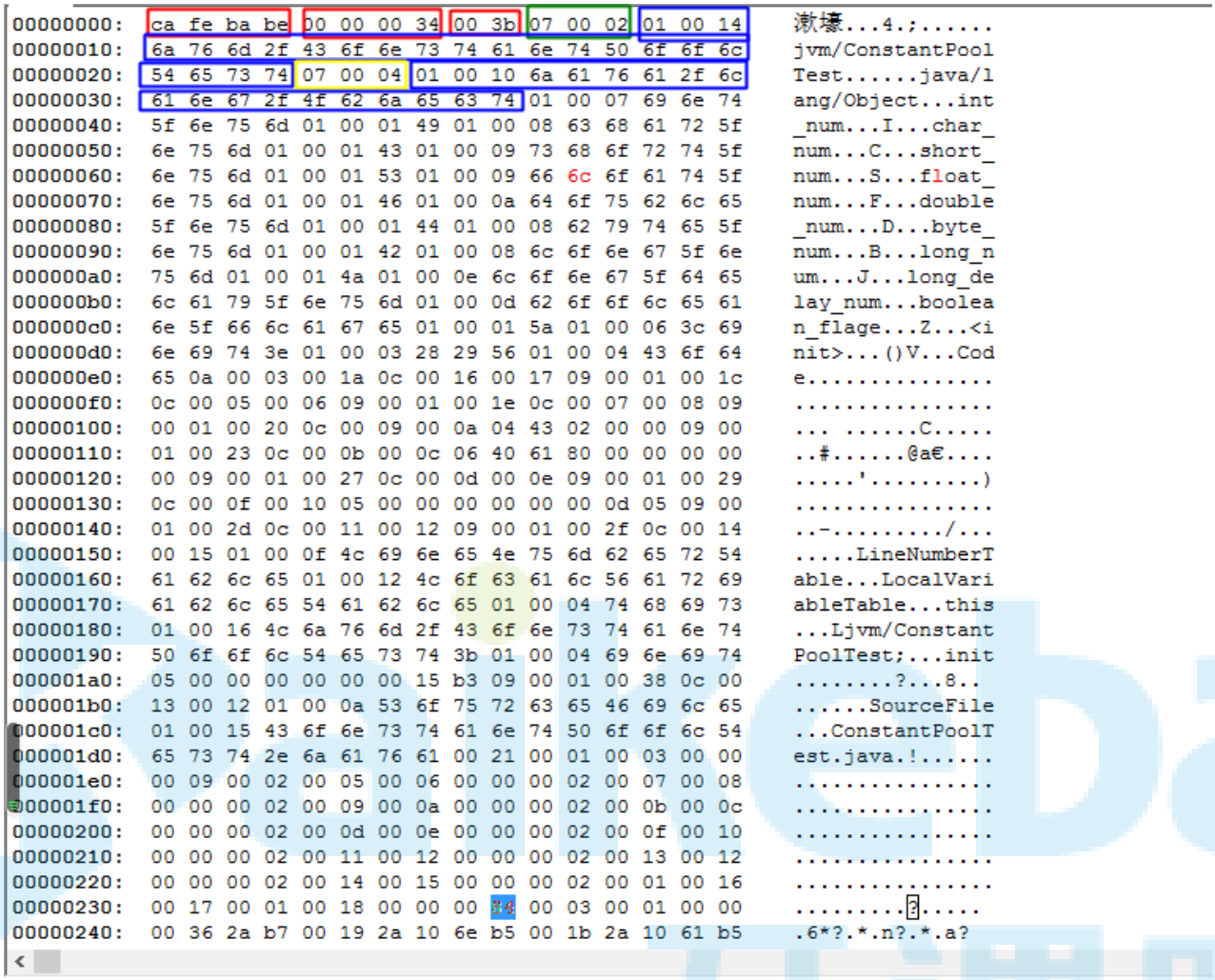
在做值相关操作时：

一个指令，可以从局部变量表、常量池、堆中对象、方法调用、系统调用中等取得数据，这些数据（可能是指，可能是对象的引用）被压入操作数栈。

一个指令，也可以从操作数数栈中取出一到多个值（pop多次），完成赋值、加减乘除、方法传参、系统调用等等操作。

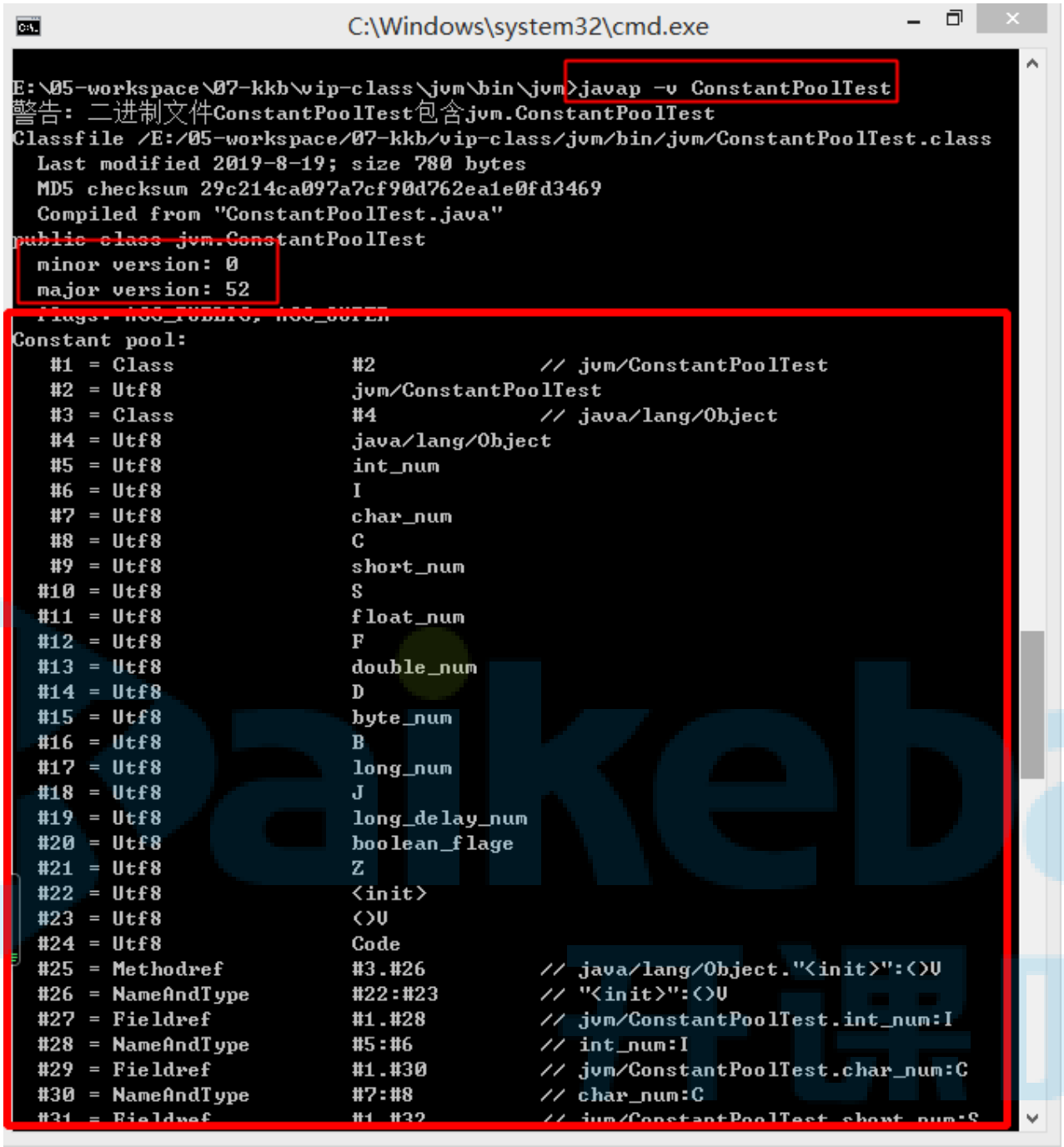
案例分析

class文件解读



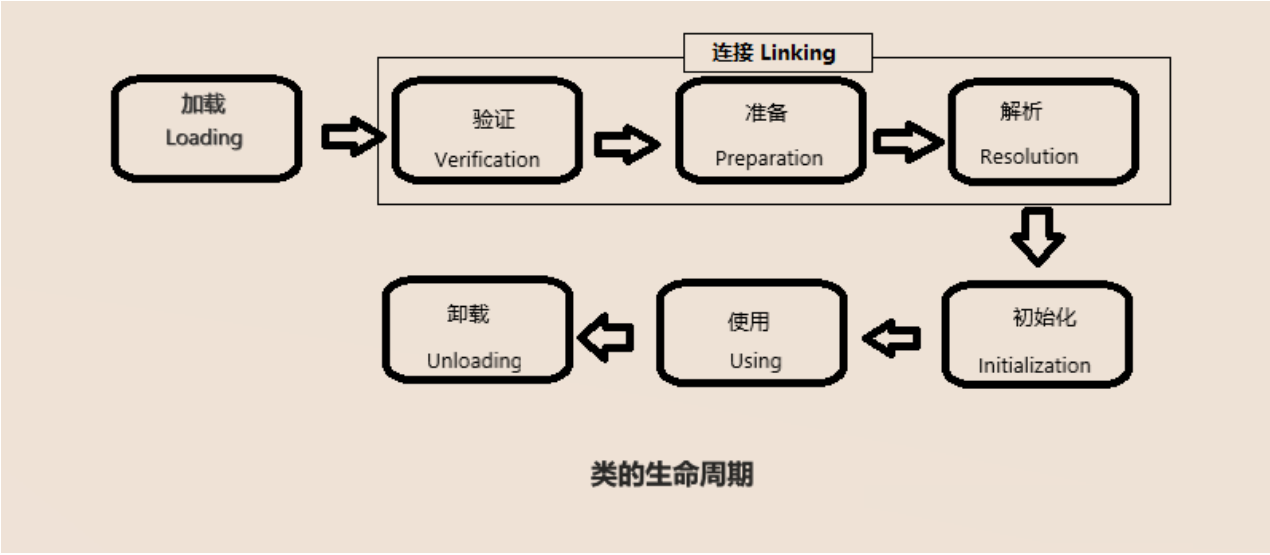
```
00000000:  ca fe ba be 00 00 00 34 00 3b 07 00 02 01 00 14    滾壞...4.;......
00000010:  6a 76 6d 2f 43 6f 6e 73 74 61 6e 74 50 6f 6f 6c    jvm/ConstantPool
00000020:  54 65 73 74 07 00 04 01 00 10 6a 61 76 61 2f 6c    Test......java/l
00000030:  61 6e 67 2f 4f 62 6a 65 63 74 01 00 07 69 6e 74    ang/Object...int
00000040:  5f 6e 75 6d 01 00 01 49 01 00 08 63 68 61 72 5f    _num...I...char_
00000050:  6e 75 6d 01 00 01 43 01 00 09 73 68 6f 72 74 5f    num...C...short_
00000060:  6e 75 6d 01 00 01 53 01 00 09 66 6c 6f 61 74 5f    num...S...float_
00000070:  6e 75 6d 01 00 01 46 01 00 0a 64 6f 75 62 6c 65    num...F...double
00000080:  5f 6e 75 6d 01 00 01 44 01 00 08 62 79 74 65 5f    _num...D...byte_
00000090:  6e 75 6d 01 00 01 42 01 00 08 6c 6f 6e 67 5f 6e    num...B...long_n
000000a0:  75 6d 01 00 01 4a 01 00 0e 6c 6f 6e 67 5f 64 65    um...J...long_de
000000b0:  6c 61 79 5f 6e 75 6d 01 00 0d 62 6f 6f 6c 65 61    lay_num...boolea
000000c0:  6e 5f 66 6c 61 67 65 01 00 01 5a 01 00 06 3c 69    n_flage...Z...<i
000000d0:  6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64    nit>...()V...Cod
000000e0:  65 0a 00 03 00 1a 0c 00 16 00 17 09 00 01 00 1c    e...............
000000f0:  0c 00 05 00 06 09 00 01 00 1e 0c 00 07 00 08 09    ................
00000100:  00 01 00 20 0c 00 09 00 0a 04 43 02 00 00 09 00    ... ......C.....
00000110:  01 00 23 0c 00 0b 00 0c 06 40 61 80 00 00 00 00    ..#......@a€....
00000120:  00 09 00 01 00 27 0c 00 0d 00 0e 09 00 01 00 29    .....'.........)
00000130:  0c 00 0f 00 10 05 00 00 00 00 00 00 0d 05 09 00    ................
00000140:  01 00 2d 0c 00 11 00 12 09 00 01 00 2f 0c 00 14    ..-........./...
00000150:  00 15 01 00 0f 4c 69 6e 65 4e 75 6d 62 65 72 54    .....LineNumberT
00000160:  61 62 6c 65 01 00 12 4c 6f 63 61 6c 56 61 72 69    able...LocalVari
00000170:  61 62 6c 65 54 61 62 6c 65 01 00 04 74 68 69 73    ableTable...this
00000180:  01 00 16 4c 6a 76 6d 2f 43 6f 6e 73 74 61 6e 74    ...Ljvm/Constant
00000190:  50 6f 6f 6c 54 65 73 74 3b 01 00 04 69 6e 69 74    PoolTest;...init
000001a0:  05 00 00 00 00 00 00 15 b3 09 00 01 00 38 0c 00    ........?...8..
000001b0:  13 00 12 01 00 0a 53 6f 75 72 63 65 46 69 6c 65    ......SourceFile
000001c0:  01 00 15 43 6f 6e 73 74 61 6e 74 50 6f 6f 6c 54    ...ConstantPoolT
000001d0:  65 73 74 2e 6a 61 76 61 00 21 00 01 00 03 00 00    est.java.!......
000001e0:  00 09 00 02 00 05 00 06 00 00 00 02 00 07 00 08    ................
000001f0:  00 00 00 02 00 09 00 0a 00 00 00 02 00 0b 00 0c    ................
00000200:  00 00 00 02 00 0d 00 0e 00 00 00 02 00 0f 00 10    ................
00000210:  00 00 00 02 00 11 00 12 00 00 00 02 00 13 00 12    ................
00000220:  00 00 00 02 00 14 00 15 00 00 00 02 00 01 00 16    ................
00000230:  00 17 00 01 00 18 00 00 00 2f 00 03 00 01 00 00    ........./......
00000240:  00 36 2a b7 00 19 2a 10 6e b5 00 1b 2a 10 61 b5    .6*?.*.n?.*.a?
```

javap显示结果解读



看清楚类加载子系统

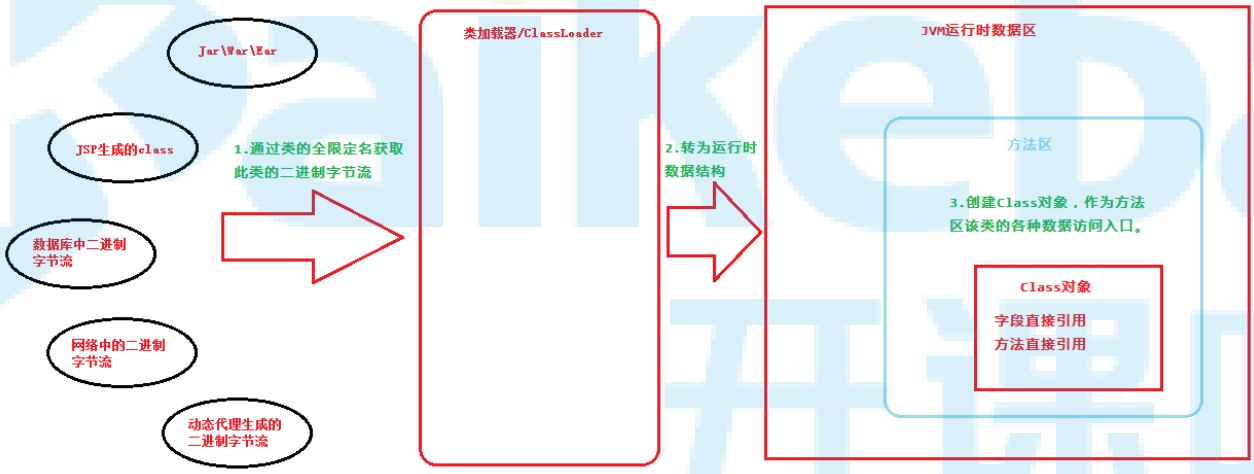
类加载的过程



类的生命周期

加载

“加载”是“类加载”(Class Loading)过程的第一步。这个加载过程主要就是靠**类加载器**实现的，包括用户自定义类加载器。



加载的过程

在加载的过程中，JVM主要做3件事情

- **通过一个类的全限定名来获取定义此类的二进制字节流(class文件)**
  在程序运行过程中，当要访问一个类时，若发现这个类尚未被加载，并满足类初始化的条件时，就根据要被初始化的这个类的全限定名找到该类的二进制字节流，开始加载过程
- **将这个字节流的静态存储结构转化为方法区的运行时数据结构**
- **在内存中创建一个该类的java.lang.Class对象**，作为方法区该类的各种数据的访问入口

> 程序在运行中所有对该类的访问都通过这个类对象，也就是这个Class对象是提供给外界访问该类的接口。

加载源

JVM规范对于加载过程给予了较大的宽松度.一般二进制字节流都从已经编译好的本地class文件中读取，此外还可以从以下地方读取。

- **zip包**
  Jar、War、Ear等
- **其它文件生成**
  由JSP文件中生成对应的Class类.
- **数据库中**
  将二进制字节流存储至数据库中，然后在加载时从数据库中读取.有些中间件会这么做，用来实现代码在集群间分发
- **网络**
  从网络中获取二进制字节流.典型就是Applet.
- **运行时计算生成**
  动态代理技术，用ProxyGenerator.generateProxyClass为特定接口生成形式为"*$Proxy"的代理类的二进制字节流.

类和数组加载的区别

数组也有类型，称为“数组类型”.如:

```
String[] str = new String[10];
```

这个数组的数组类型是`[Ljava.lang.String`，而String只是这个数组的元素类型。

数组类和非数组类的类加载是不同的，具体情况如下：

- **非数组类：**是由类加载器来完成。
- **数组类：**数组类本身不通过类加载器创建，它是由*java*虚拟机直接创建，但数组类与类加载器有很密切的关系，因为数组类的元素类型最终要靠类加载器创建。

加载过程的注意点

- **JVM规范并未给出类在方法区中存放的数据结构**

  类完成加载后，二进制字节流就以特定的数据结构存储在方法区中，但存储的数据结构是由虚拟机自己定义的，虚拟机规范并没有指定。

- **JVM规范并没有指定Class对象存放的位置**

  在二进制字节流以特定格式存储在方法区后，JVM会创建一个java.lang.Class类的对象，作为本类的外部访问接口。

  既然是对象就应该存放在Java堆中，不过JVM规范并没有给出限制，不同的虚拟机根据自己的需求存放这个对象。

  HotSpot将Class对象存放在方法区。

- **加载阶段和链接阶段是交叉的**

类加载的过程中每个步骤的开始顺序都有严格限制，但每个步骤的结束顺序没有限制。也就是说，**类加载过程中，必须按照如下顺序开始:**

```
加载 -> 链接 -> 初始化
```

但结束顺序无所谓，因此由于每个步骤处理时间的长短不一就会导致有些步骤会出现交叉。

验证

验证阶段比较耗时，它非常重要但不一定必要(因为对程序运行期没有影响)，如果所运行的代码已经被反复使用和验证过，那么可以使用`-Xverify:none`参数关闭，以缩短类加载时间。

验证的目的

保证二进制字节流中的信息符合虚拟机规范，并没有安全问题。

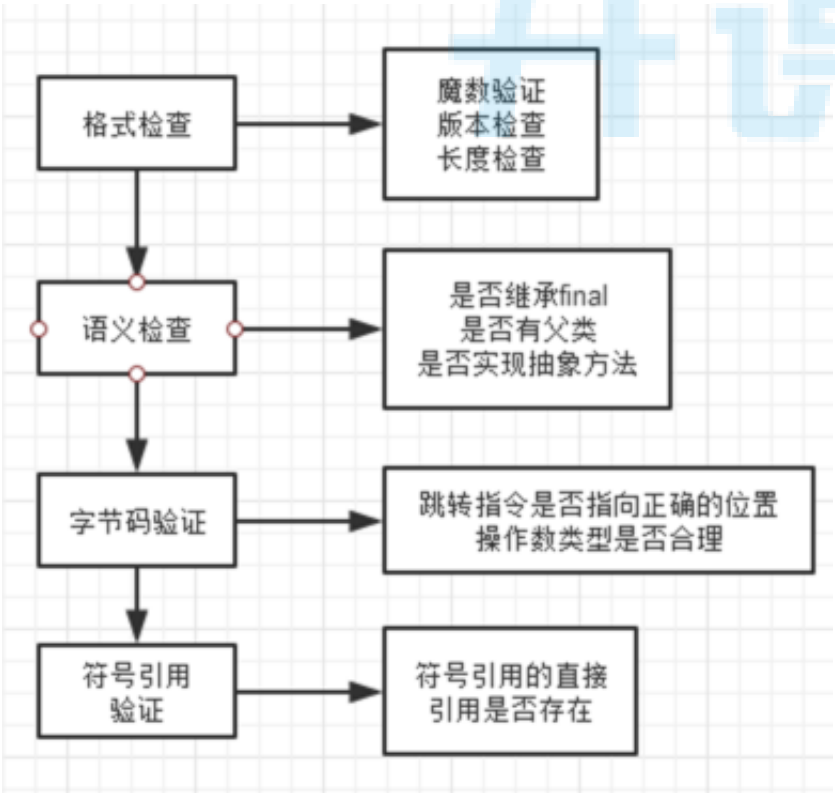
验证的必要性

虽然Java语言是一门安全的语言，它能确保程序猿无法访问数组边界以外的内存、避免让一个对象转换成任意类型、避免跳转到不存在的代码行.也就是说，Java语言的安全性是通过编译器来保证的.

但是我们知道，编译器和虚拟机是两个独立的东西，虚拟机只认二进制字节流，它不会管所获得的二进制字节流是哪来的，当然，如果是编译器给它的，那么就相对安全，但如果是从其它途径获得的，那么无法确保该二进制字节流是安全的。

通过上文可知，虚拟机规范中没有限制二进制字节流的来源，在字节码层面上，上述Java代码无法做到的都是可以实现的，至少语义上是可以表达出来的，为了防止字节流中有安全问题，需要验证!

验证的过程



- **文件格式验证**
  验证字节流是否符合Class文件格式的规范，并且能被当前的虚拟机处理.

  本验证阶段是**基于二进制字节流**进行的，只有通过本阶段验证，才被允许存到方法区

  后面的三个验证阶段都是基于方法区的存储结构进行，不会再直接操作字节流。

  印证【加载和验证】是交叉进行的：

  ```
  1.加载开始前，二进制字节流还没进方法区，而加载完成后，二进制字节流已经存入方法区

  2.而在文件格式验证前，二进制字节流尚未进入方法区，文件格式验证通过之后才进入方法区

  也就是说，加载开始后，立即启动了文件格式验证，本阶段验证通过后，二进制字节流被转换成特定数据结构存储至方法区中，继而开始下阶段的验证和创建Class对象等操作
  ```

- **元数据验证**

  对字节码描述信息进行语义分析，确保符合Java语法规范.

- **字节码验证**

  本阶段是验证过程的最复杂的一个阶段。

  本阶段对方法体进行语义分析，保证方法在运行时不会出现危害虚拟机的事件。

  字节码验证将对类的方法进行校验分析，保证被校验的方法在运行时不会做出危害虚拟机的事，一个类方法体的字节码没有通过字节码验证，那一定有问题，但若一个方法通过了验证，也不能说明它一定安全。

- **符号引用验证**

  发生在JVM将符号引用转化为直接引用的时候，这个转化动作发生在解析阶段，对类自身以外的信息进行匹配校验，确保解析能正常执行。

准备

仅仅为类变量（即static修饰的字段变量）分配内存并且设置该类变量的初始值即零值，这里不包含用final修饰的static，因为final在编译的时候就会分配了（编译器的优化），同时这里也不会为实例变量分配初始化。**类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。**

准备阶段主要完成两件事情：

- **为已在方法区中的类的静态成员变量分配内存**
- **为静态成员变量设置初始值，初始值为0、false、null等**

数据类型	默认零值
int	0
long	0L
short	(short)0
char	'\u0000'
byte	(byte)0
boolean	false
float	0.0f
double	0.0d
reference	null

比如：

```
public static int x = 1000;
```

注意：

```
实际上变量x在准备阶段过后的初始值为0，而不是1000
将x赋值为1000的putstatic指令是程序被编译后，存放于类构造器<clinit>方法之中
```

但是如果声明为：

```
public static final int x = 1000;
```

```
在编译阶段会为x生成ConstantValue属性，在准备阶段虚拟机会根据ConstantValue属性将x赋值为1000。
```

解析

解析是虚拟机将常量池的**符号引用替换为直接引用的过程。**

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的`CONSTANT_Class_info`、`CONSTANT_Fieldref_info`、`CONSTANT_Methodref_info`、`CONSTANT_InterfaceMethodref_info`四种常量类型。

1. **类或接口的解析：**

   判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。

2. **字段解析：**

   对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束(优先从接口来，然后是继承的父类.理论上是按照上述顺序进行搜索解析，但在实际应用中，虚拟机的编译器实现可能要比上述规范要求的更严格一些。如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译）.

3. **类方法解析：**

   对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。

4. **接口方法解析：**

   与类方法解析步骤类似，只是接口不会有父类，因此，只递归向上搜索父接口就行了。

初始化

初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码(初始化成为代码设定的默认值)。在准备阶段，类变量已经被赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序指定的主观计划去初始化类变量和其他资源，或者可以从另一个角度来表达：初始化阶段是执行类构造器()方法的过程。

其实初始化过程就是调用类初始化方法的过程，完成对static修饰的类变量的手动赋值还有主动调用静态代码块。

初始化过程的注意点

- 方法是编译器自动收集**类中所有类变量的赋值动作和静态语句块中的语句合并产生**的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的.
- 静态代码块只能访问到出现在静态代码块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问.

```
public class Test {
    static {
        i=0;
        System.out.println(i);//编译失败:"非法向前引用"
    }
    static int i = 1;
}
```

- 实例构造器需要显式调用父类构造函数，而类的不需要调用父类的类构造函数，虚拟机会确保子类的方法执行前已经执行完毕父类的方法.因此在JVM中第一个被执行的方法的类肯定是java.lang.Object.
- 如果一个类/接口中没有静态代码块，也没有静态成员变量的赋值操作，那么编译器就不会为此类生成方法.
- 接口也需要通过方法为接口中定义的静态成员变量显示初始化。

  接口中不能使用静态代码块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成方法.不同的是，执行接口的方法不需要先执行父接口的方法.只有当父接口中的静态成员变量被使用到时才会执行父接口的方法.
- 虚拟机会保证在多线程环境中一个类的方法别正确地加锁，同步.当多条线程同时去初始化一个类时，只会有一个线程去执行该类的方法，其它线程都被阻塞等待，直到活动线程执行方法完毕.其他线程虽会被阻塞，只要有一个方法执行完，其它线程唤醒后不会再进入方法.同一个类加载器下，一个类型只会初始化一次.

  使用静态内部类的单例实现：

```
public class Student {
  private Student() {}
  /*
   * 此处使用一个内部类来维护单例 JVM在类加载的时候，是互斥的，所以可以由此保证线程安全问题
   */
  private static class SingletonFactory {
    private static Student student = new Student();
  }

  /* 获取实例 */
  public static Student getSingletonInstance() {
    return SingletonFactory.student;
  }
}
```

类加载的时机

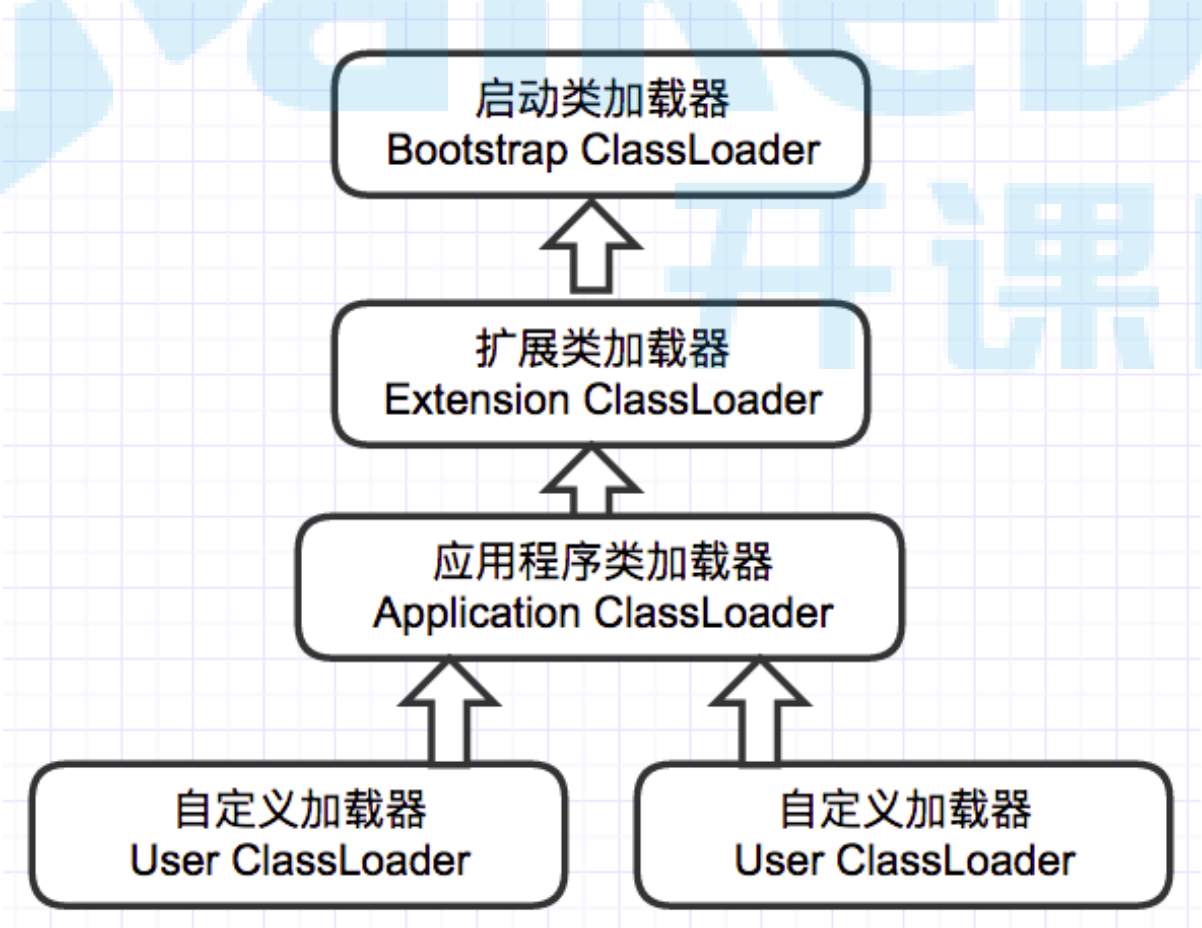
什么时候开始加载，虚拟机规范并没有强制性的约束，对于其它大部分阶段究竟何时开始虚拟机规范也都没有进行规范，这些都是交由虚拟机的具体实现来把握。所以不同的虚拟机它们开始的时机可能是不同的。但是对于初始化却严格的规定了有且只有四种情况必须先对类进行“初始化”(加载，验证，准备自然需要在初始化之前完成)：

1. 遇到`new`、`getstatic`、`putstatic`和`invokestatic`这四条指令时，如果对应的类没有初始化，则要对对应的类先进行初始化。

   这四个指令对应到我们java代码中的场景分别是：

   - new关键字实例化对象的时候；
   - 读取或设置一个类的静态字段（读取被final修饰，已在编译器把结果放入常量池的静态字段除外）；
   - 调用类的静态方法时。
2. 使用`java.lang.reflect`包方法时对类进行反射调用的时候。
3. 初始化一个类的时候发现其父类还没初始化，要先初始化其父类。
4. 当虚拟机开始启动时，用户需要指定一个主类，虚拟机会先执行这个主类的初始化。

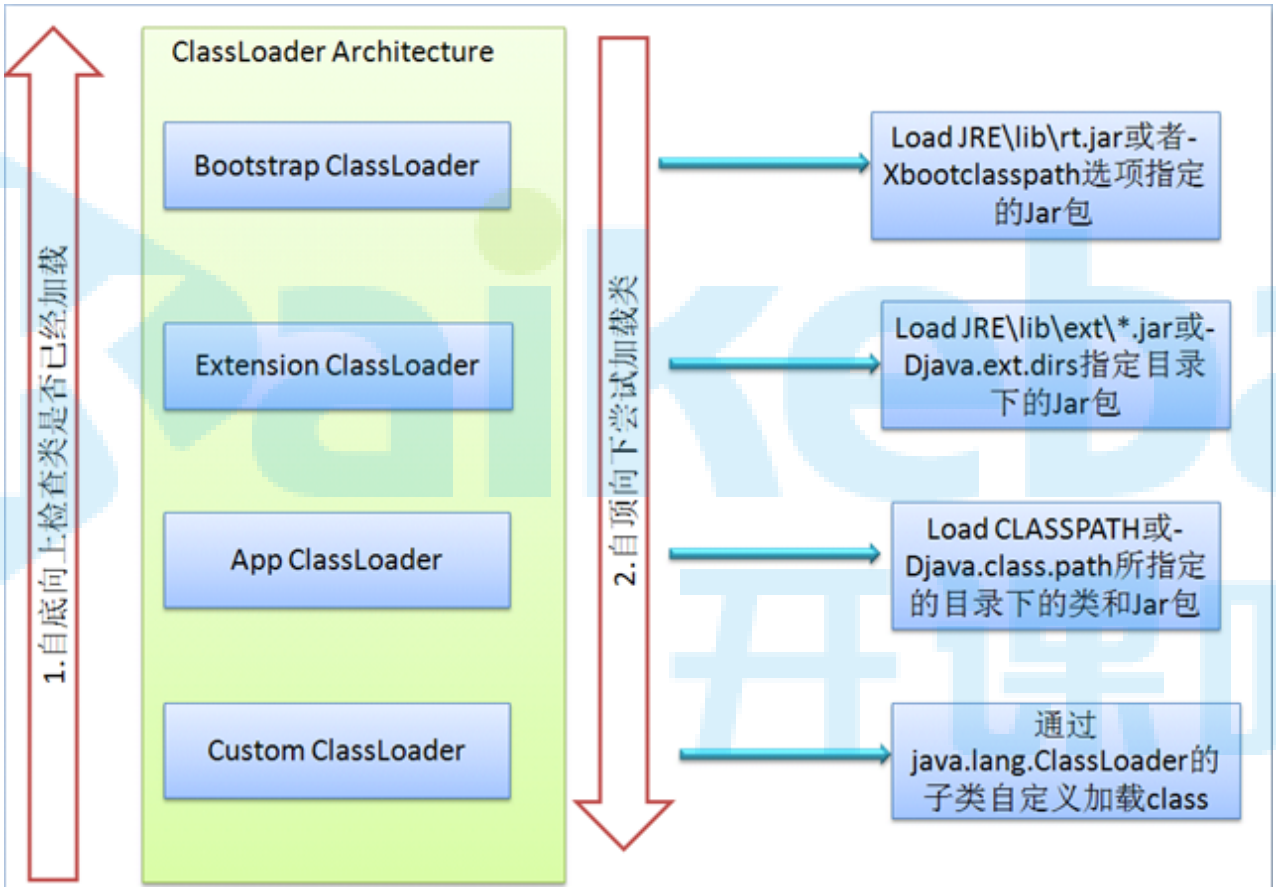
类加载器



- **启动类加载器(Bootstrap ClassLoader)**：
  - 负责加载 JAVA_HOME\lib 目录中的，

    - 或通过-Xbootclasspath参数指定路径中的，
    - 且被虚拟机认可（按文件名识别，如rt.jar）的类。
    - 由C++实现，不是ClassLoader子类
    - java、 javax
- **扩展类加载器(Extension ClassLoader)**：
    - 负责加载 JAVA_HOME\lib\ext 目录中的，
    - 或通过java.ext.dirs系统变量指定路径中的类库。
- **应用程序类加载器(Application ClassLoader)**：
    - 负责加载用户路径（classpath）上的类库。

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

自定义类加载器

自定义类加载器步骤

（1）继承ClassLoader

（2）重写findClass（）方法

（3）调用defineClass（）方法

实践

下面写一个自定义类加载器：指定类加载路径在D盘下的lib文件夹下。

（1）在本地磁盘新建一个`Test.java`类，代码如下：

```
package jvm.classloader;
public class Test {
    public void say(){
        System.out.println("Hello MyClassLoader");
    }
}
```

（2）使用`javac -d . Test.java`命令，将生成的`Test.class`文件放到`D:/lib/jvm/classloader`文件夹下。

（3）在Eclipse中自定义类加载器，代码如下：

```
package jvm.classloader;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class MyClassLoader extends ClassLoader{

    private String classpath;

    public MyClassLoader(String classpath) {

        this.classpath = classpath;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        try {
```

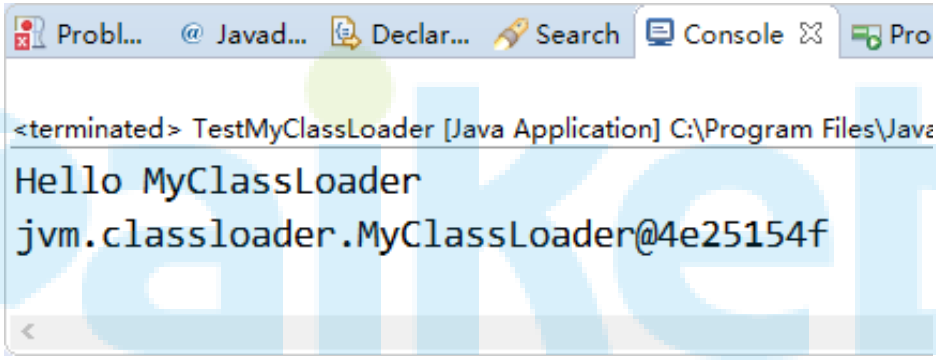
```
            byte [] classDate=getData(name);
            if(classDate==null){}
            else{
                //defineClass方法将字节码转化为类
                return defineClass(name, classDate, 0, classDate.length);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return super.findClass(name);
    }
    //返回类的字节码
    private byte[] getData(String className) throws IOException{
        InputStream in = null;
        ByteArrayOutputStream out = null;
        String path=classpath + File.separatorChar +
                    className.replace('.', File.separatorChar)+".class";
        try {
            in=new FileInputStream(path);
            out=new ByteArrayOutputStream();
            byte[] buffer=new byte[2048];
            int len=0;
            while((len=in.read(buffer))!=-1){
                out.write(buffer, 0, len);
            }
            return out.toByteArray();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        finally{
            in.close();
            out.close();
        }
        return null;
    }
}
```

测试代码如下(在Eclipse中):

```
package jvm.classloader;

import java.lang.reflect.Method;

public class TestMyClassLoader {

    public static void main(String []args) throws Exception{
```

```
        //自定义类加载器的加载路径
        MyClassLoader myClassLoader=new MyClassLoader("D:\\lib");
        //包名+类名
        Class c=myClassLoader.loadClass("jvm.classloader.Test");

        if(c!=null){
            Object obj=c.newInstance();
            Method method=c.getMethod("say",  null);
            method.invoke(obj,  null);
            System.out.println(c.getClassLoader().toString());
        }
    }
}
```

输出结果如下：



自定义类加载器的作用：

JVM自带的三个加载器只能加载指定路径下的类字节码。

如果某个情况下，我们需要加载应用程序之外的类文件呢？比如本地D盘下的，或者去加载网络上的某个类文件，这种情况就可以使用自定义加载器了

双亲委派模型

JVM通过**双亲委派模型**进行类的加载，当然我们也可以通过继承java.lang.ClassLoader实现自定义的类加载器。

- 当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，
- 只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。

采用双亲委派的一个好处是：

- 比如加载位于rt.jar包中的类java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个Object对象。

为什么要使用双亲委托这种模型呢?

因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要子ClassLoader再加载一次。

考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的String来动态替代java核心api中定义的类型，这样会存在非常大的安全隐患，而双亲委托的方式，就可以避免这种情况，**因为String已经在启动时就被引导类加载器（Bootstrcp ClassLoader）加载，所以用户自定义的ClassLoader永远也无法加载一个自己写的String，除非你改变JDK中ClassLoader搜索类的默认算法。**

但是JVM在搜索类的时候，又是如何判定两个class是相同的呢?

JVM在判定两个class是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实例加载的。

只有两者同时满足的情况下，JVM才认为这两个class是相同的。就算两个class是同一份class字节码，如果被两个不同的ClassLoader实例所加载，JVM也会认为它们是两个不同class。

```
for{
  ClassLoader cl = new ClassLoader();
  cl.load("com.kaikeba.James");
}
```

既然JVM已经提供了默认的类加载器，为什么还要定义自已的类加载器呢?

因为Java中提供的默认ClassLoader，只加载指定目录下的jar和class，如果我们想加载其它位置的类或jar时。

比如：我要加载网络上的一个class文件，通过动态加载到内存之后，要调用这个类中的方法实现我的业务逻辑。在这样的情况下，默认的ClassLoader就不能满足我们的需求了，所以需要定义自己的ClassLoader

破坏双亲委派模型

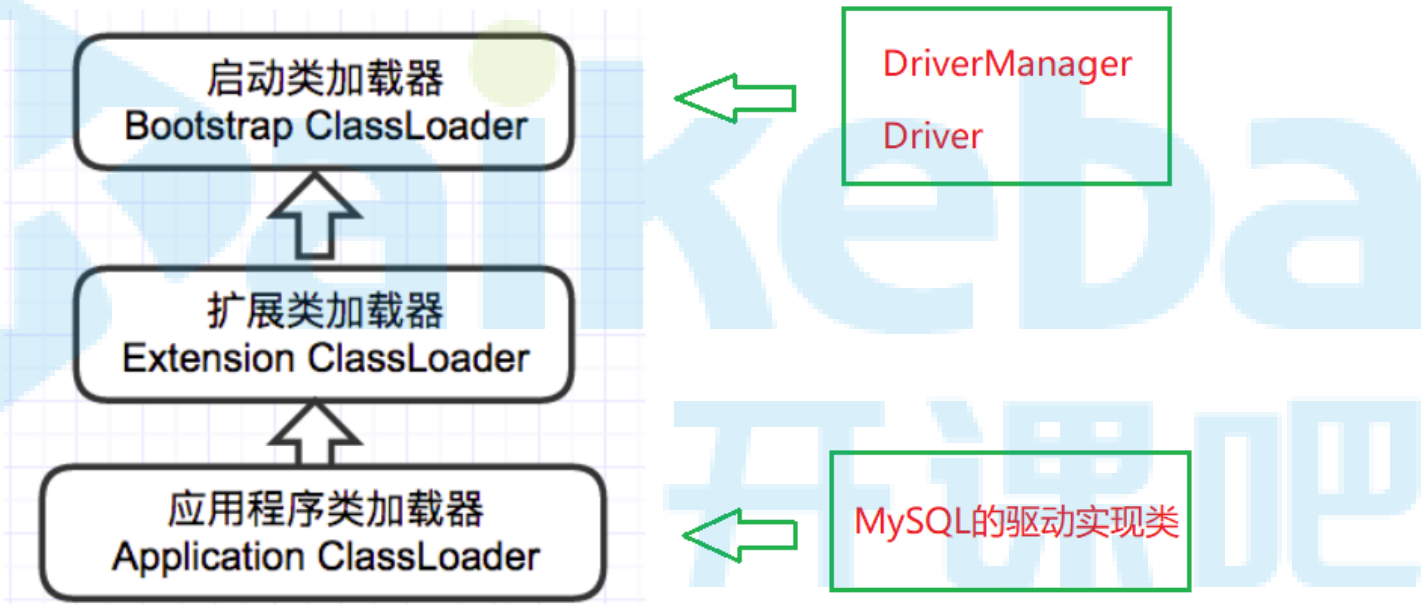
为什么需要破坏双亲委派?

因为在某些情况下父类加载器需要加载的class文件由于受到加载范围的限制，父类加载器无法加载到需要的文件，这个时候就需要委托子类加载器进行加载。

而按照双亲委派模式的话，是子类委托父类加载器去加载class文件。这个时候需要破坏双亲委派模式才能加载成功父类加载器需要的类。也就是说父类会委托子类去加载它需要的class文件。

以**Driver接口**为例，由于Driver接口定义在jdk当中的，而其实现由各个数据库的服务商来提供，比如mysql的就写了`MySQL Connector`，这些实现类都是以jar包的形式放到classpath目录下。

那么问题就来了，DriverManager（也由jdk提供）要加载各个实现了Driver接口的实现类（classpath下），然后进行管理，但是DriverManager由启动类加载器加载，只能加载JAVA_HOME的lib下文件，而其实现是由服务商提供的，由系统类加载器加载，这个时候就需要启动类加载器来委托子类来加载Driver实现，从而破坏了双亲委派，这里仅仅是举了破坏双亲委派的其中一个情况。



JDBC代码：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.junit.Test;

public class TestJdbc {
  @Test
  public void testJdbc() {
```

```
    Connection connection = null;
    PreparedStatement preparedStatement = null;
    ResultSet rs = null;

    try {
      // 加载数据库驱动
      Class.forName("com.mysql.jdbc.Driver");

      // 通过驱动管理类获取数据库链接connection = DriverManager
      connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/ssm?
characterEncoding=utf-8",  "root",
          "root");

      // 定义sql语句 ?表示占位符
      String sql = "select * from user where id = ?";

      // 获取预处理 statement
      preparedStatement = connection.prepareStatement(sql);

      // 设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数为设置的
      preparedStatement.setInt(1,  1);

      // 向数据库发出 sql 执行查询，查询出结果集
      rs = preparedStatement.executeQuery();

      // 遍历查询结果集
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      // 释放资源
      if (rs != null) {
        try {
          rs.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
      if (preparedStatement != null) {
        try {
          preparedStatement.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
      if (connection != null) {
        try {
          connection.close();
        } catch (SQLException e) {
```

```
        }
      }
    }
  }
}
```