课程主题

class类文件和类加载器详解

课程目标

课程内容

一、JVM通识

1.为什么要学习JVM

> - 面试重灾区，我们必须搞懂它。
> - 生产过程中，肯定会面临JVM调优相关问题，需要也必须搞懂它。
> - 写出好代码，搞懂了JVM的一些优化手段，更加知道什么样的代码会被优化

2.什么是JVM

> JVM就是Java虚拟机，它是Java程序运行的载体。

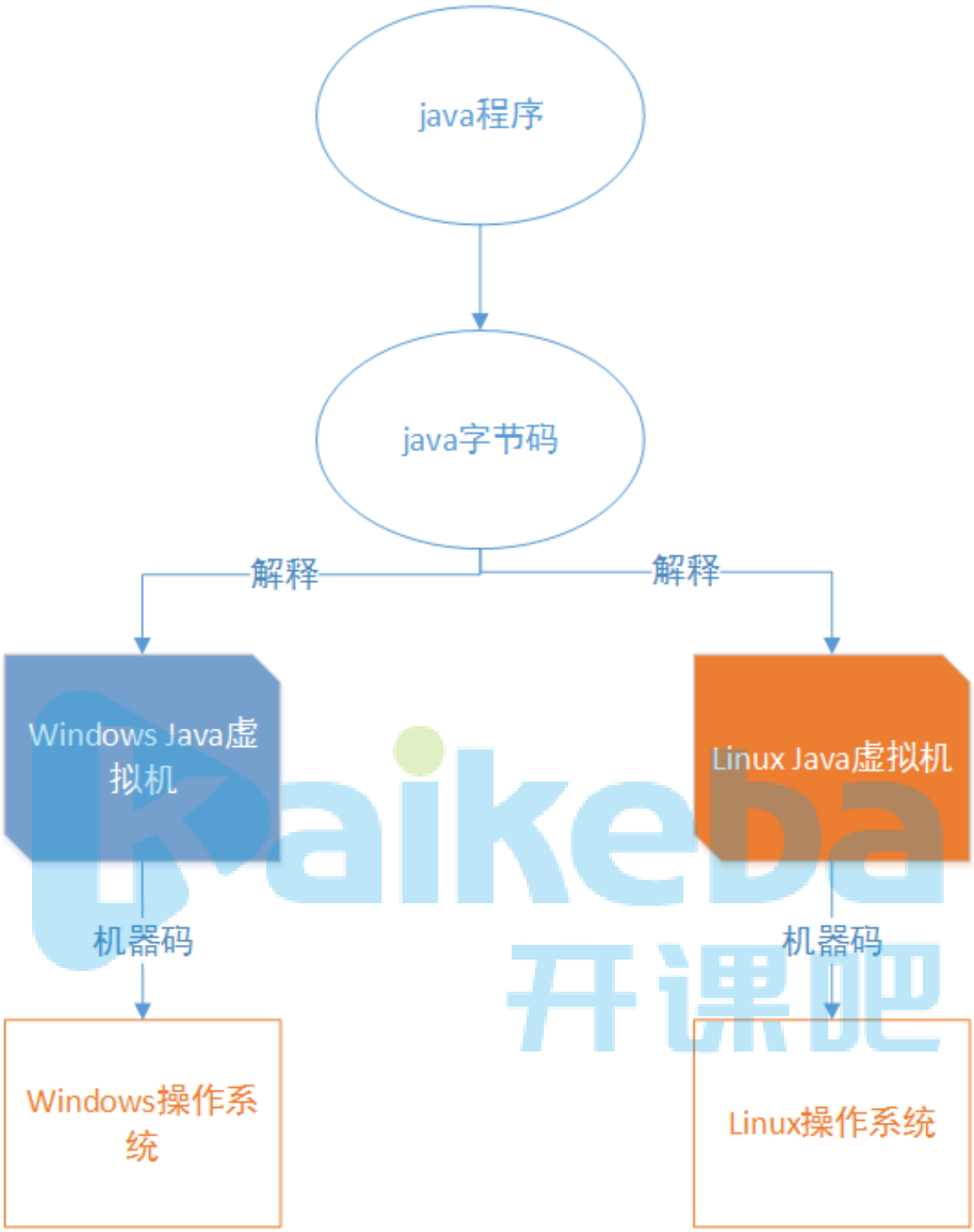
Java和JVM

计算机只识别0和1。

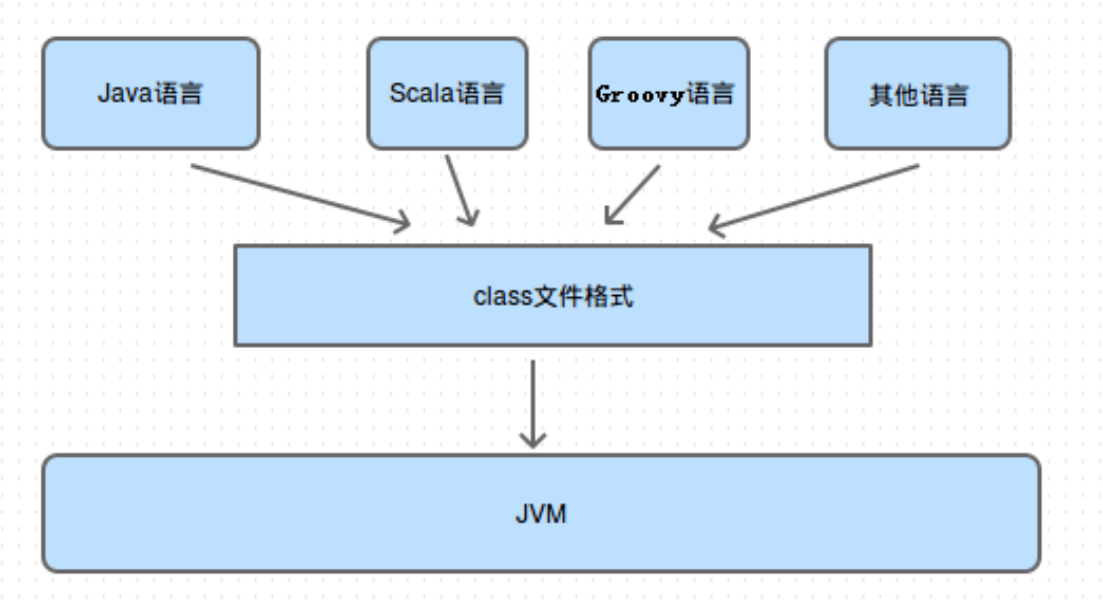
Java是高级语言。高级语言编写的程序要想被计算机执行，需要变成二进制形式的本地机器码。能直接变成机器码的语义是C++，它的缺点是不同操作系统，需要准备多份。Java需要先变成Java字节码（class文件）。然后再变成机器码。

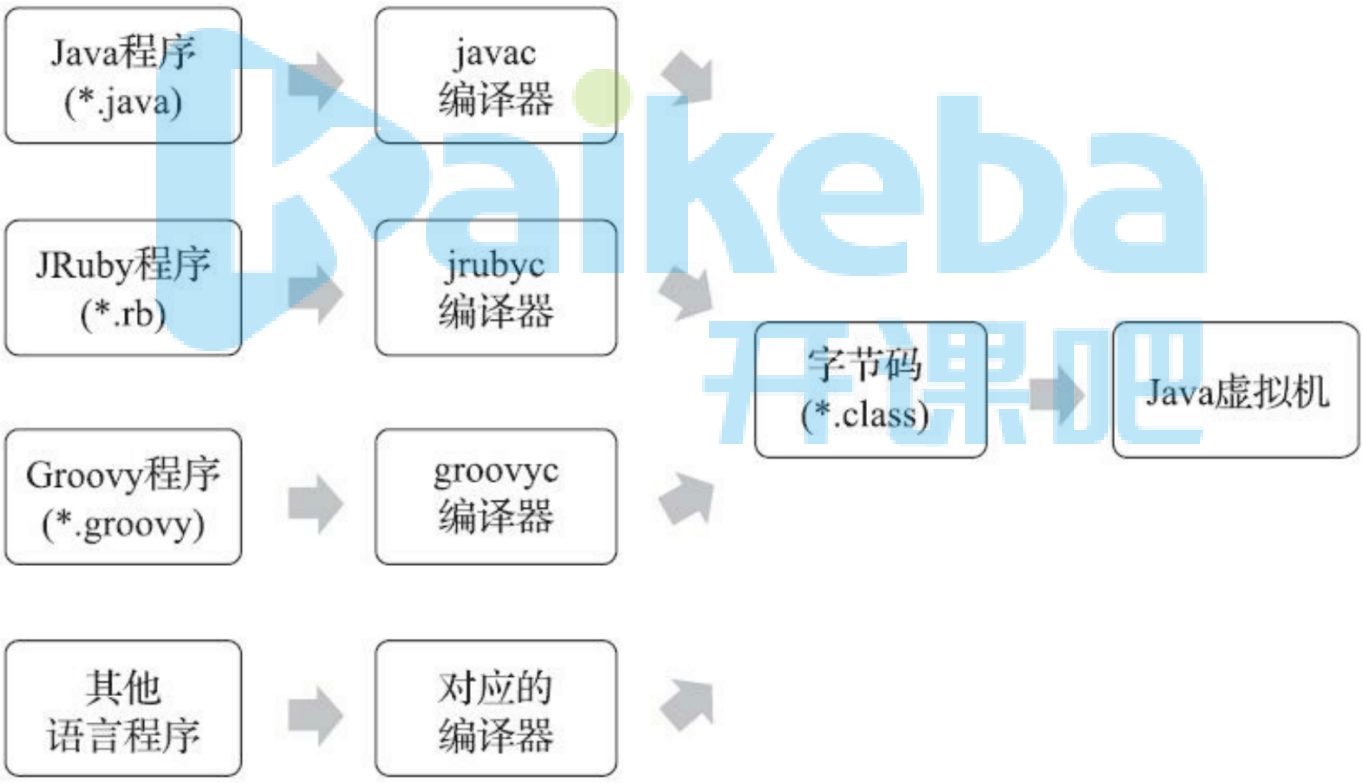
JVM可以实现Java的一次编译，到处运行。

这个就是区别于类似于C语言的方式。

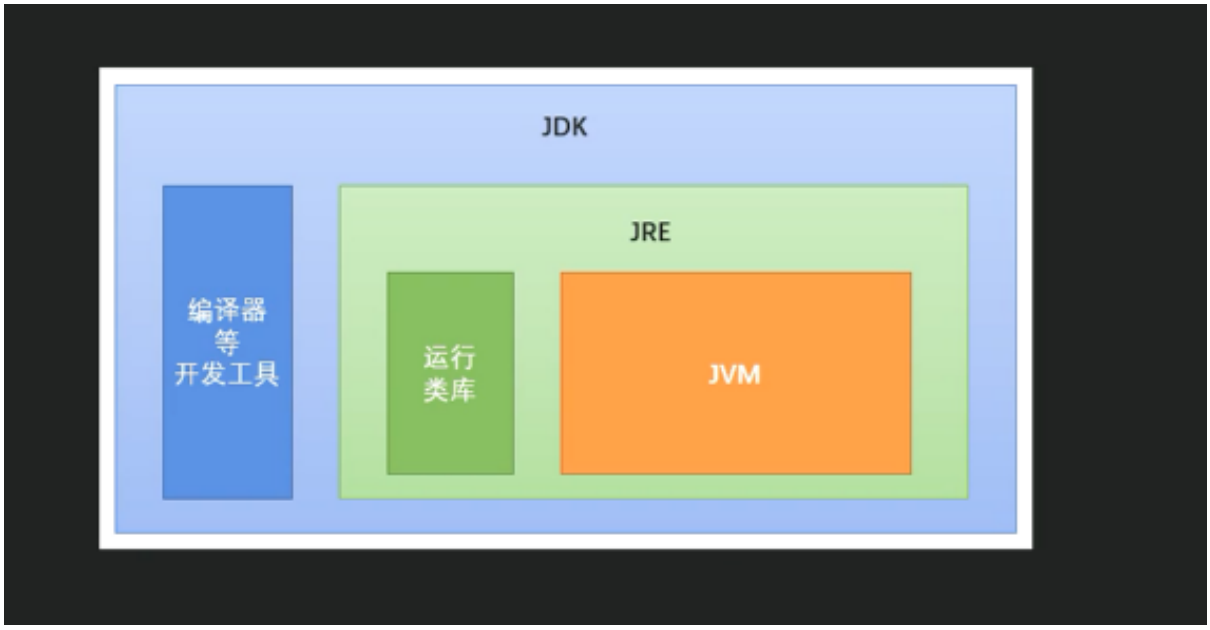


> 机器码是电脑CPU直接读取运行的机器指令，运行速度最快，但是非常晦涩难懂，也比较难编写，一般从业人员接触不到。
>
> 字节码是一种中间状态（中间码）的二进制代码（文件）。需要直译器转译后才能成为机器码。


Java语言
Scala语言
Groovy语言
其他语言
class文件格式
JVM


Java程序
(*.java)
javac
编译器
JRuby程序
(*.rb)
jrubyc
编译器
Groovy程序
(*.groovy)
groovyc
编译器
其他
语言程序
对应的
编译器
字节码
(*.class)
Java虚拟机

JDK、JRE和JVM



OracleJDK和OpenJDK

查看JDK的版本

```
java -version
```

(1) 如果是SUN/OracleJDK， 显示信息为:

```
[root@localhost ~]# java -version
java version "1.8.0_162"
Java(TM) SE Runtime Environment (build 1.8.0_162-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12,  mixed
mode)
```

说明:

> `Java HotSpot(TM) 64-Bit Server VM` 表明， 此JDK的JVM是Oracle的64位HotSpot虚拟机， 运行在Server模式下(虚拟机有Server和Client两种运行模式).

> `Java(TM) SE Runtime Environment (build 1.8.0_162-b12)` 是Java运行时环境(即JRE)的版本信息.

⑵ 如果OpenJDK， 显示信息为:

```
[root@localhost ~]# java -version
openjdk version "1.8.0_144"
OpenJDK Runtime Environment (build 1.8.0_144-b01)
OpenJDK 64-Bit Server VM (build 25.144-b01,  mixed mode)
```

OpenJDK 的来历

Java由SUN公司(Sun Microsystems， 发起于美国斯坦福大学， SUN是Stanford University Network的缩写)发明， 2006年SUN公司将Java开源， 此时的JDK即为OpenJDK.

也就是说， OpenJDK是Java SE的开源实现， 它由SUN和Java社区提供支持， 2009年Oracle收购了Sun公司， 自此Java的维护方之一的SUN也变成了Oracle .

大多数JDK都是在OpenJDK的基础上编写实现的， 比如IBM J9， Azul Zulu， Azul Zing和Oracle JDK. 几乎现有的所有JDK都派生自OpenJDK， 它们之间不同的是许可证:

> OpenJDK根据许可证GPL v2发布;
>
> Oracle JDK根据Oracle二进制代码许可协议获得许可。

Oracle JDK的来历

> Oracle JDK之前被称为SUN JDK， 这是在2009年Oracle收购SUN公司之前， 收购后被命名为Oracle JDK。

实际上， Oracle JDK是基于OpenJDK源代码构建的， 因此Oracle JDK和OpenJDK之间没有重大的技术差异。

Oracle的项目发布经理Joe Darcy在OSCON 2011 上对两者关系的介绍也证实了OpenJDK 7和Oracle JDK 7在程序上是非常接近的， 两者共用了大量相同的代码(如下图)

> 注意: 图中提示了两者共同代码的占比要远高于图形上看到的比例， 所以我们编译的OpenJDK基本上可以认为性能、功能和执行逻辑上都和官方的Oracle JDK是一致的.

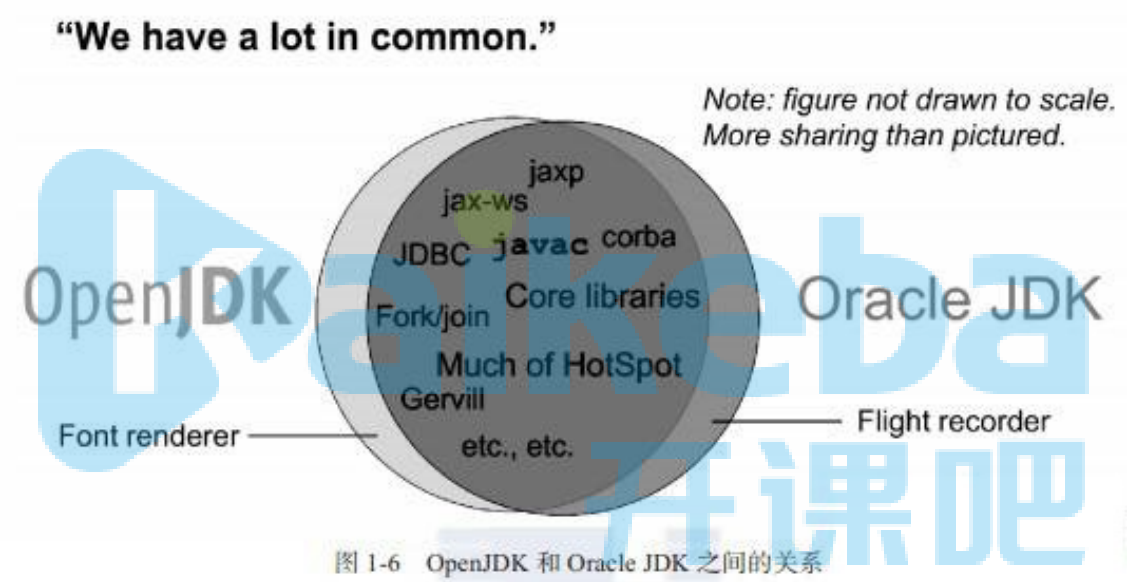


图 1-6 OpenJDK 和 Oracle JDK 之间的关系

Oracle JDK与OpenJDK的区别

> OpenJDK使用的是开源免费的FreeType， 可以按照GPL v2许可证使用.GPL V2允许在商业上使用;
>
> Oracle JDK则采用JRL(Java Research License，Java研究授权协议) 放出.JRL只允许个人研究使用，要获得Oracle JDK的商业许可证， 需要联系Oracle的销售人员进行购买。

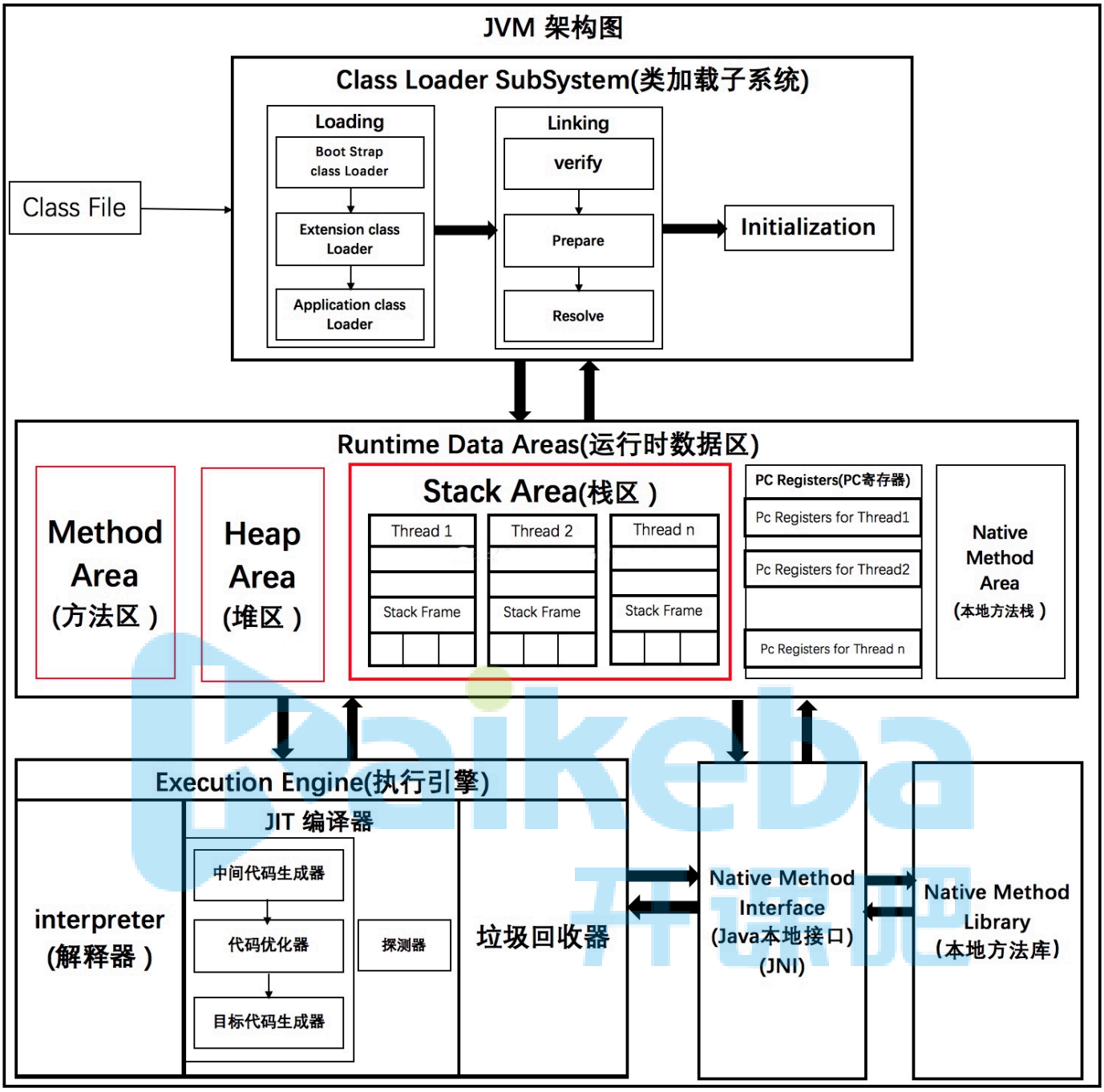
JVM和Hotspot

JVM是《JVM虚拟机规范》中提出来的规范。

Hotspot是使用JVM规范的商用产品，除此之外还有Oracle JRockit、IBM的J9也是JVM产品

> JRockit是Oracle的JVM， 从Java SE 7开始， HotSpot和JRockit合并为一个JVM。

3.如何学习JVM

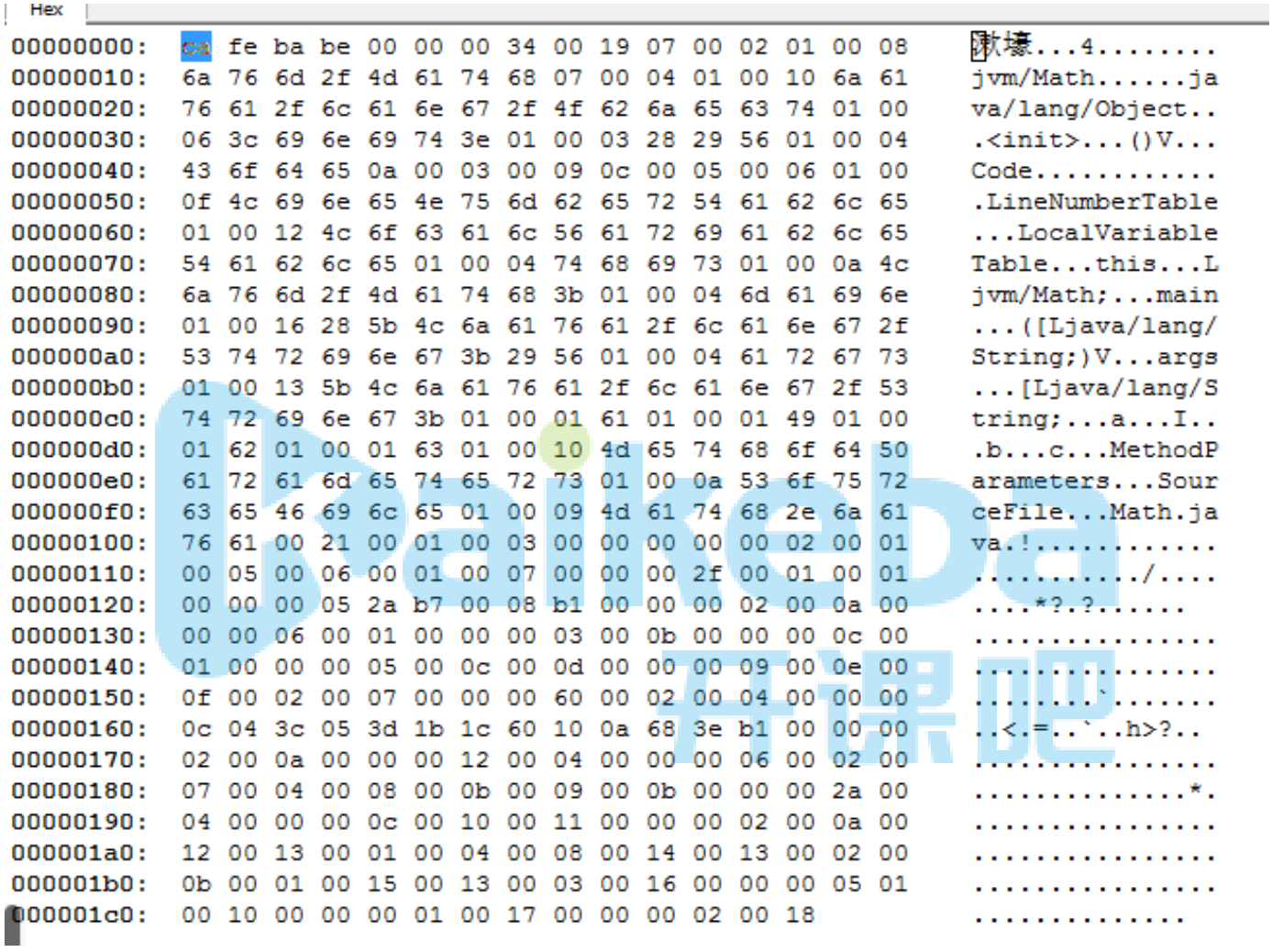


> 程序执行方式主要有三种：静态编译执行、动态编译执行和动态解释执行。
>
> 注意：此处所说的编译指的是编译成可让操作系统直接执行的机器码。

二、class文件介绍

1.class里都存了什么数据

查看字节码文件

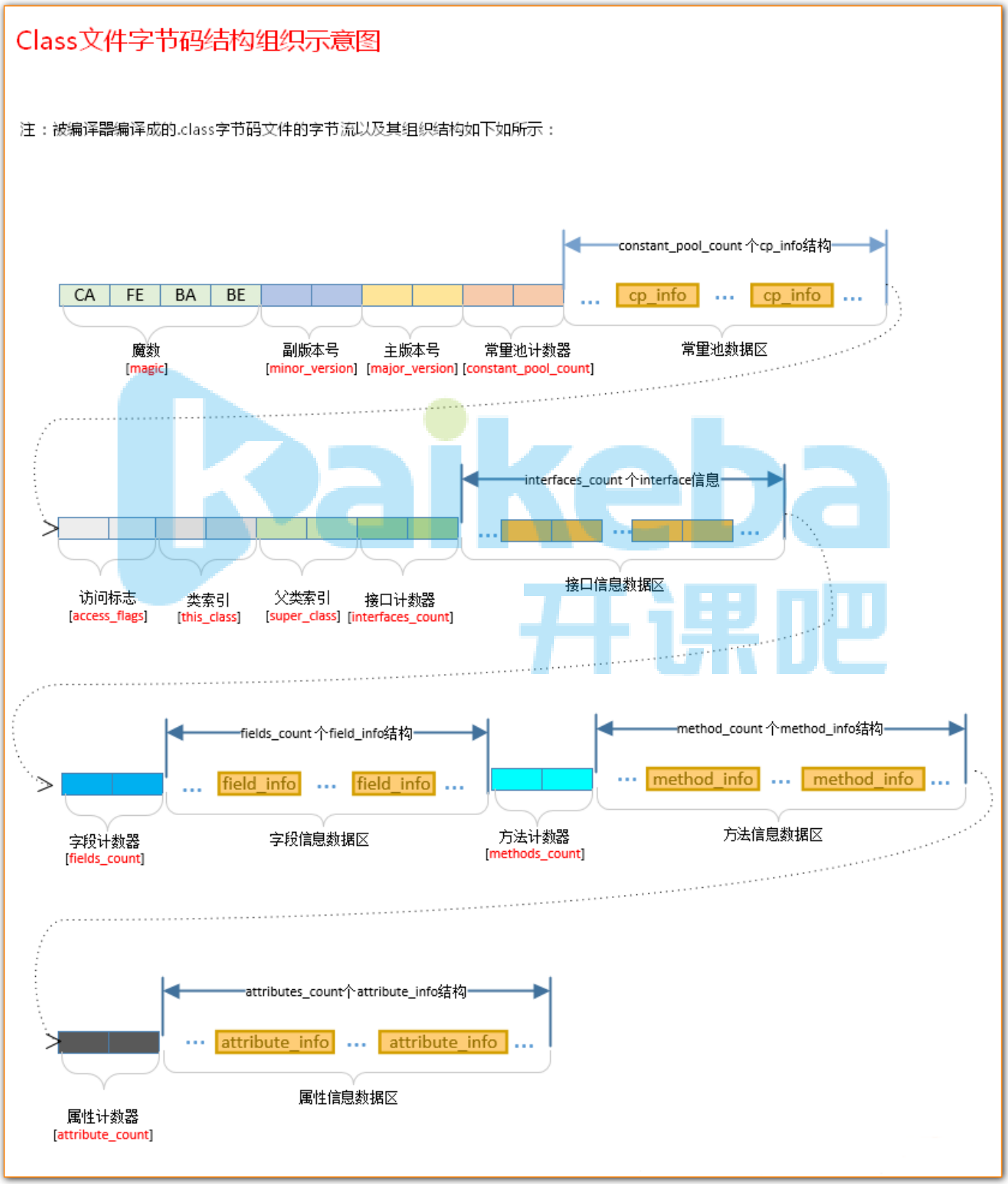


在线HEX Editor：https://www.onlinehexeditor.com/

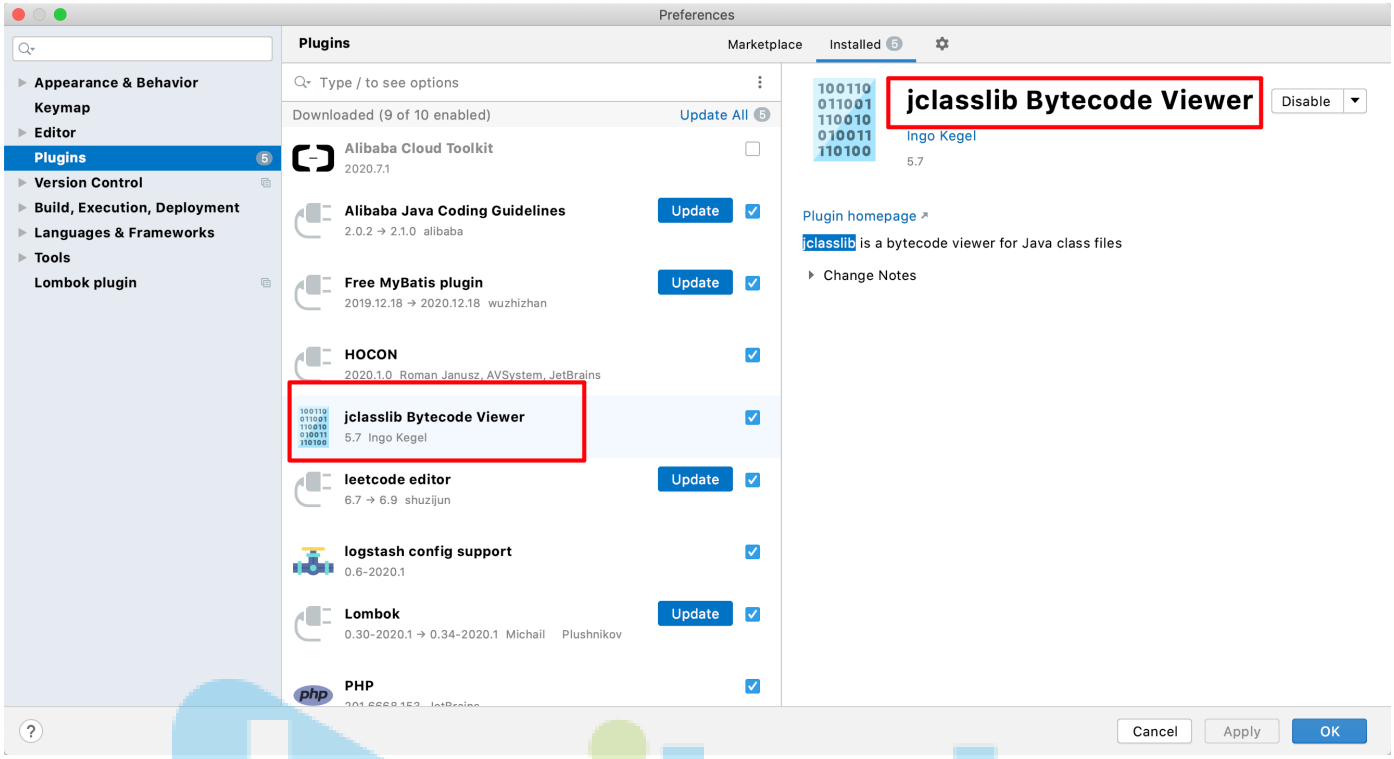
- 十六进制转字符串：http://www.bejson.com/convert/ox2str/
- 十六进制转十进制：http://tool.oschina.net/hexconvert/

字节码文件结构示意图

参考下图去阅读上面的十六进制文档：



IDEA jclasslib插件



2.class常量池如何存储数据

1. 常量池的常量有哪些

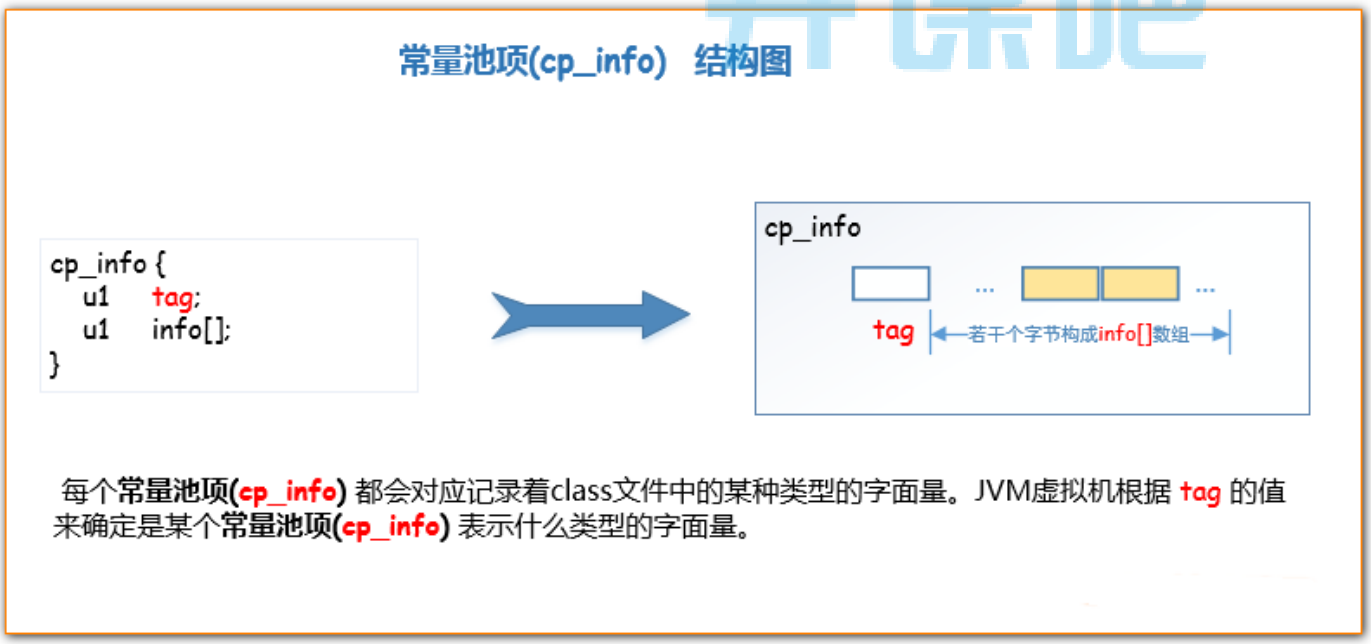
分为字面量和符号引用

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

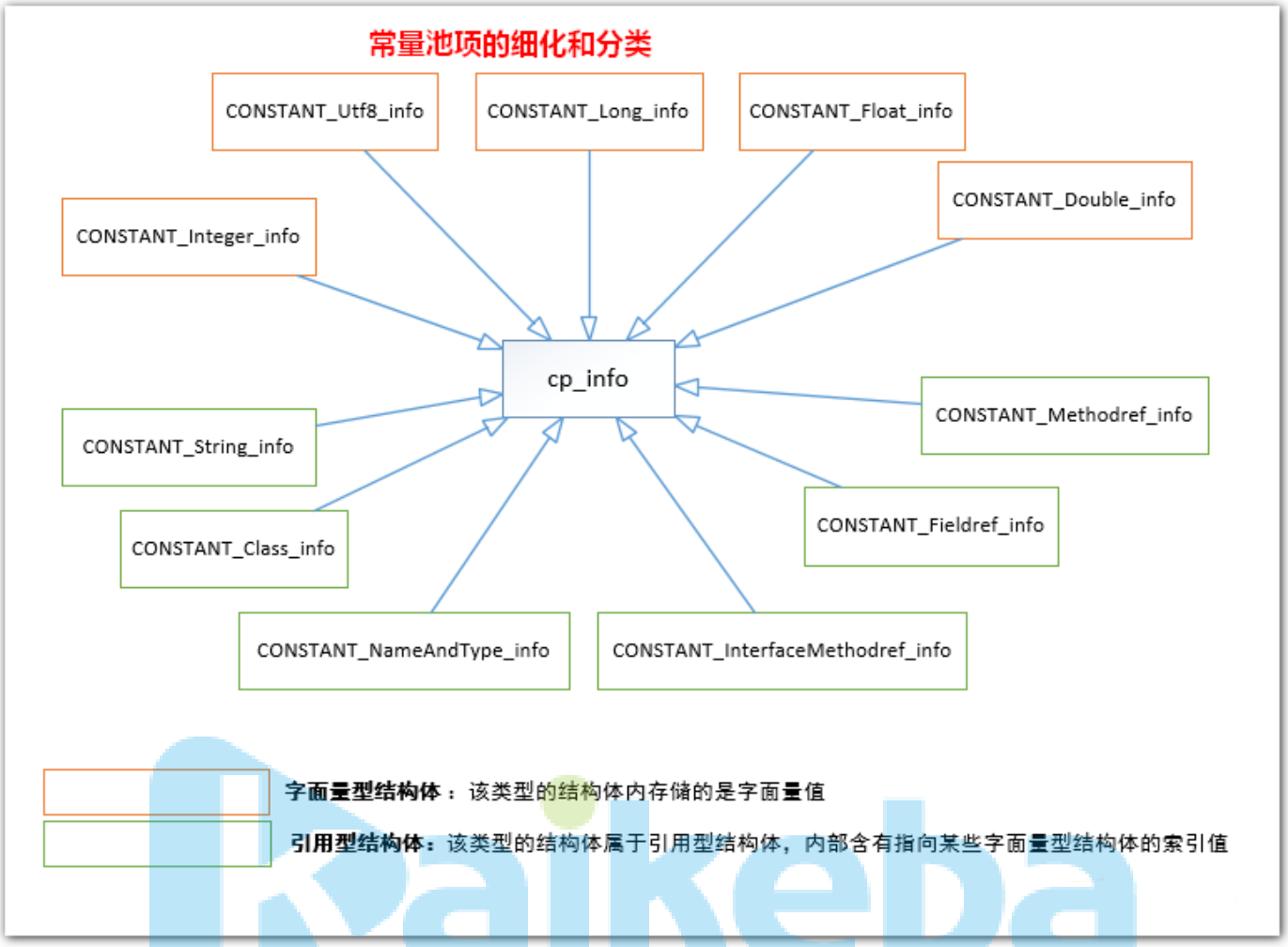
3. 常量池项 (cp_info) 的结构是什么?



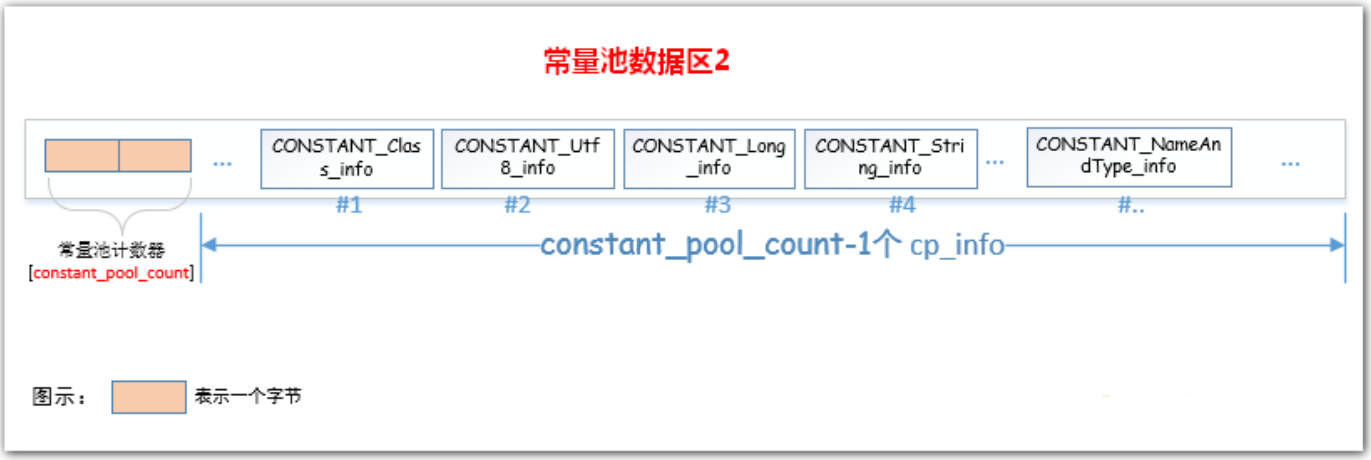
JVM虚拟机规定了不同的tag值和不同类型的字面量对应关系如下：

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

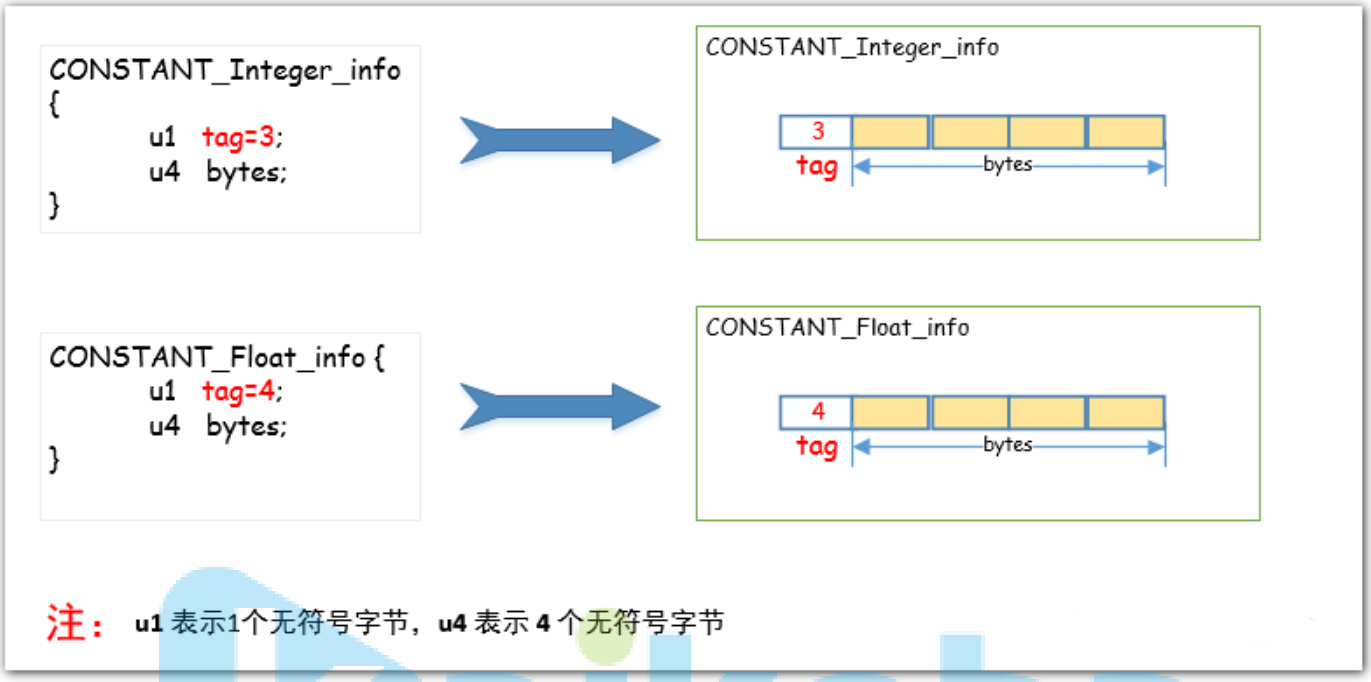
所以根据cp_info中的tag 不同的值，可以将cp_info 更细化为以下结构体：



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. 如何存储int和float数据类型的常量?



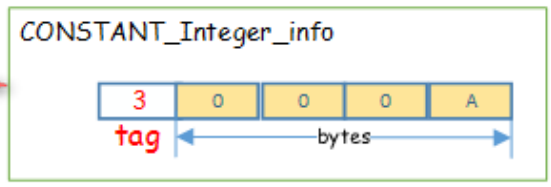

```
package com.kkb.jvm;
public class IntAndFloatTest {

    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;

}
```

```
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;
```

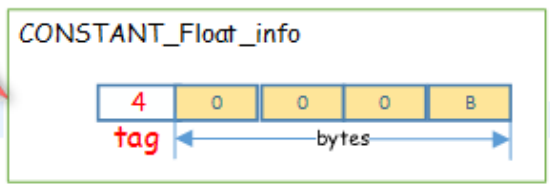


```
package com.kkb.jvm;

public class IntAndFloatTest {

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

}
```

索引值#X

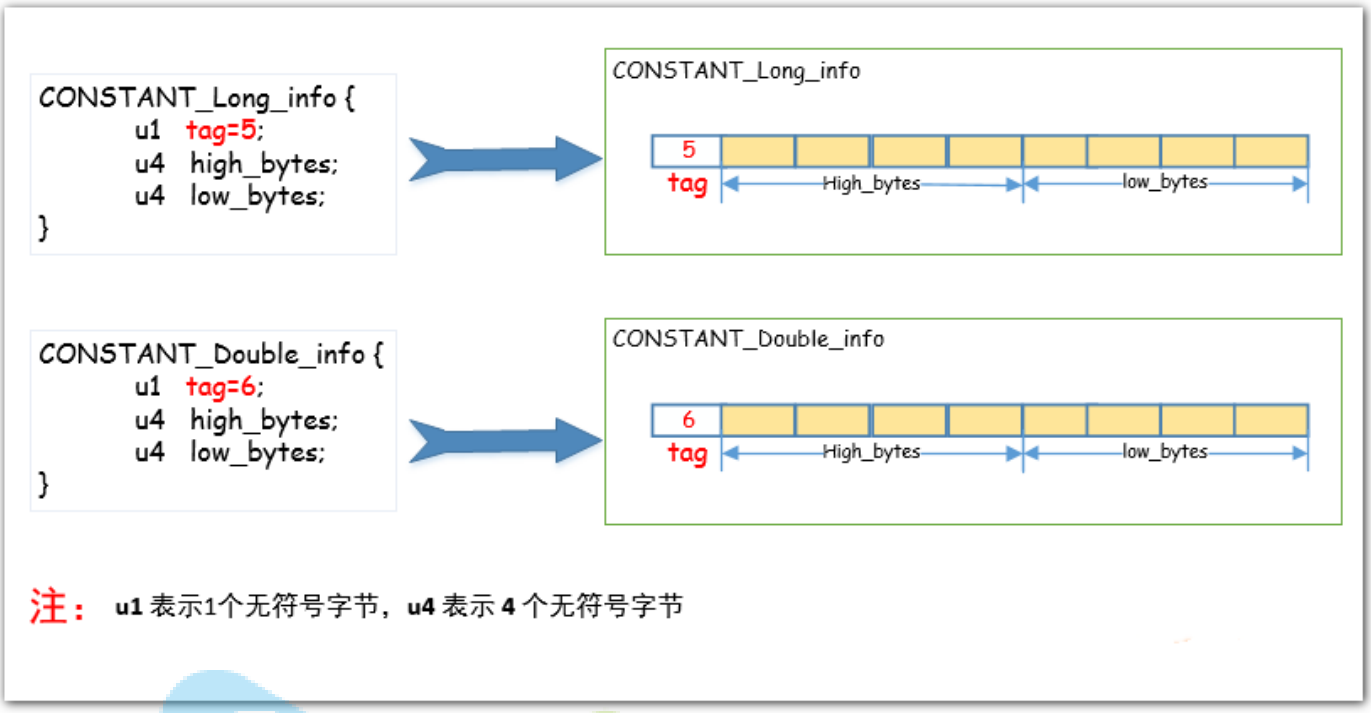


索引值#Y



编译器会将 10 和11f 分别分别包装成 CONSANT_Integer_info 和CONSTANT_Float_info结构体，然后放置到常量池中。

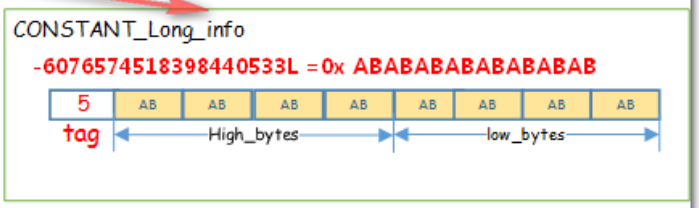
5. 如何存储long和 double数据类型的常量?

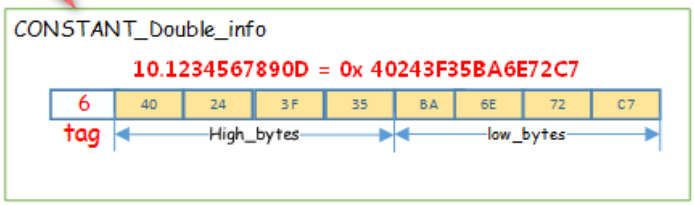



```
package com.kkb.jvm;
  public class LongAndDoubleTest {

  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;
 }
```

```
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;
```

```java
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;
}
```

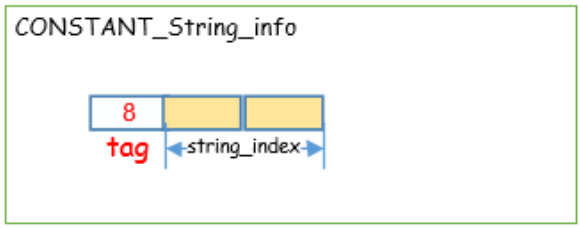




编译器会将-6076574518398440533L
和10.1234567890D分别包装成
CONSTANT_Long_info
和CONSTANT_Double_info，然后放到常量池中。

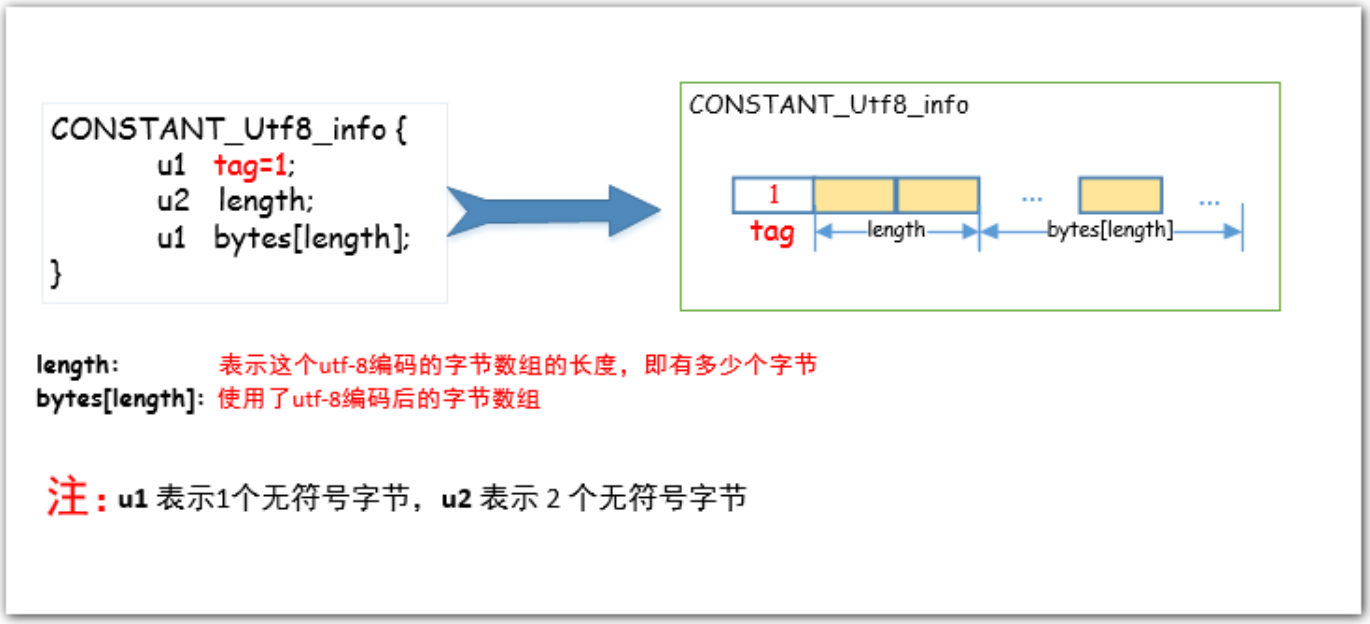
6. 如何存储String类型的字符串常量?

```
CONSTANT_String_info {
        u1  tag=8;
        u2  string_index;
}
```



结构体中占用2个字节的string_index 值指向了某个 CONSTANT_Utf8_info 结构体，
即：string_index的值是某个CONSTANT_Utf8_info结构体在常量池中的索引。

注： **u1** 表示1个无符号字节，**u2** 表示 2 个无符号字节

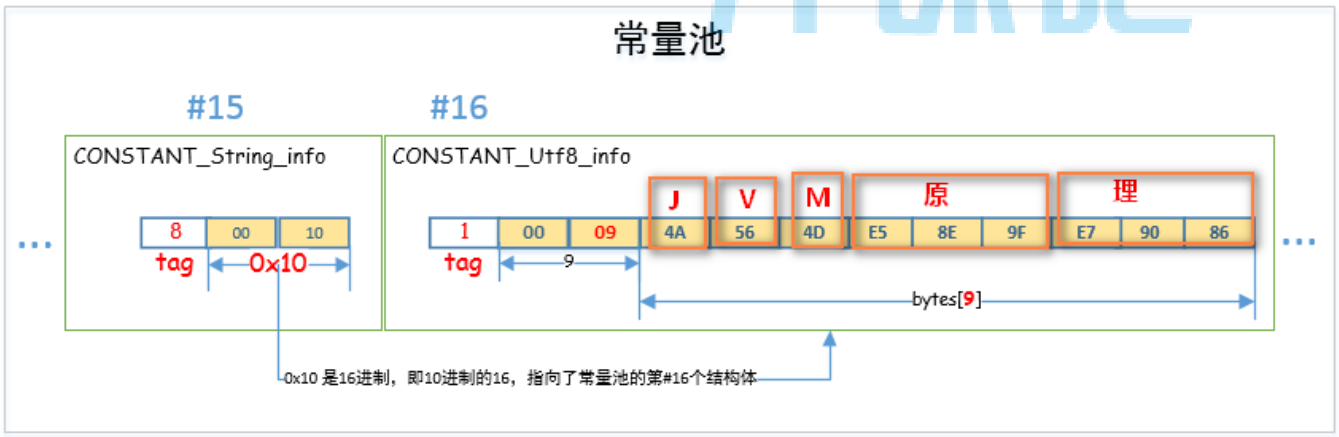



```java
package com.kkb.jvm;

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";
}
```

```
javap -v StringTest
```

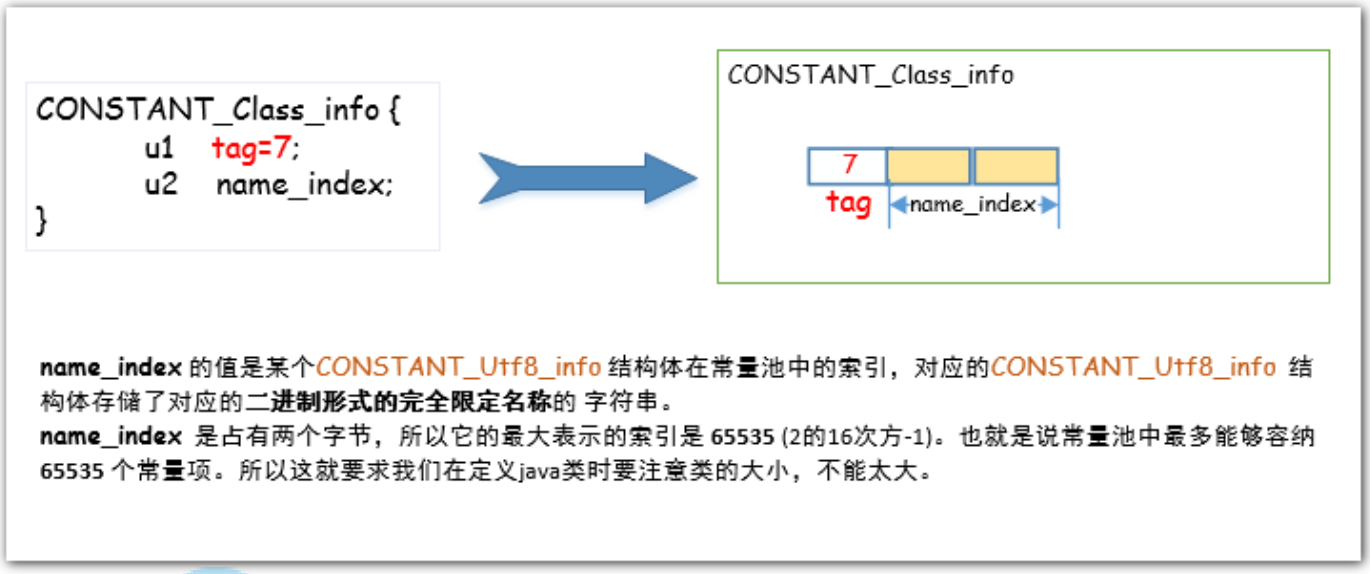
```
const #1 = class  #2; //  com/kkb/jvm/StringTest
const #2 = Asciz  com/kkb/jvm/StringTest;
const #3 = class  #4; //  java/lang/Object
const #4 = Asciz  java/lang/Object;
const #5 = Asciz  s1;
const #6 = Asciz  Ljava/lang/String;;
const #7 = Asciz  s2;
const #8 = Asciz  s3;
const #9 = Asciz  s4;
const #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;
```


"JVM原理" 字符串在常量池中的存储



注：在 中的值填写的都是16进制哦！

7. 如何存储Class引用类型?

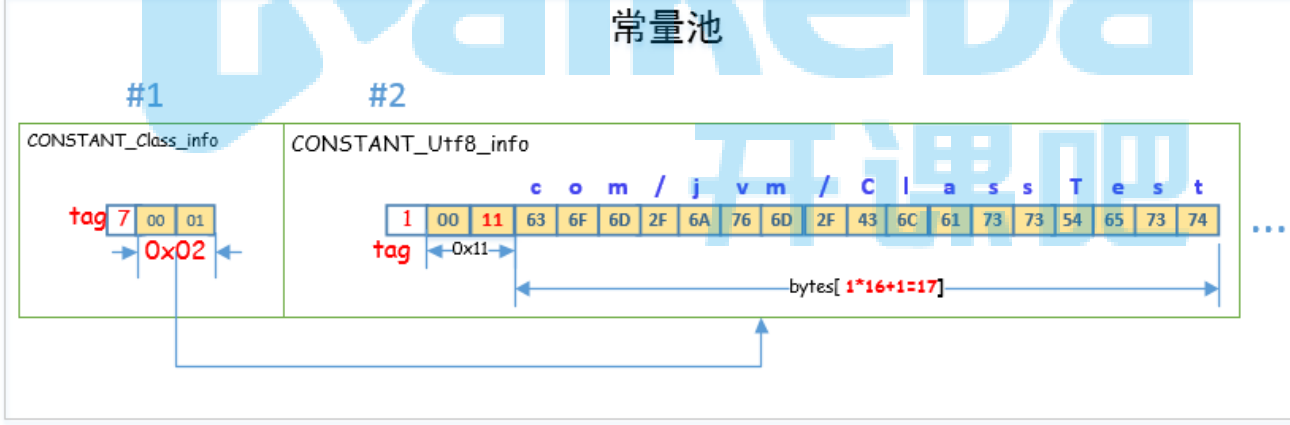



```
package com.jvm;
import  java.util.Date;
public class ClassTest {
 private Date date =new Date();
}
```

```
javap -v ClassTest
```

```
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。



注：在 [ ] 中的值填写的都是16进制哦！

CONSTANT_Class_info 结构体中的name_index值是 2，则它指向了常量池的第二个常量池项，第二个常量池项必须是Constant_Utf8_info,它存储了以utf-8编码编码的“com/jvm/ClassTest”字符串。

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

```
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
{
```

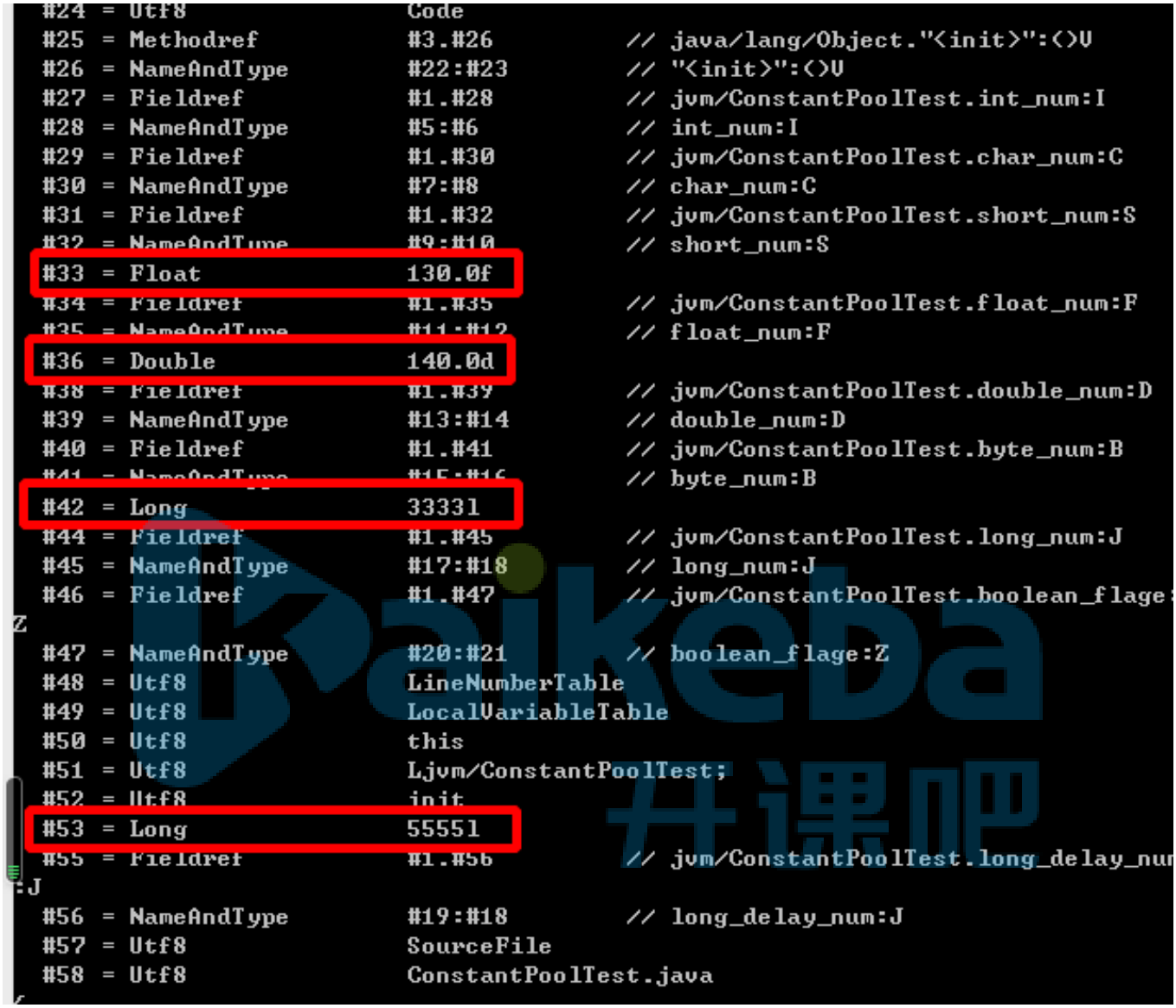
8. 哪些字面量会进入常量池中?

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：



结论：

1. 【final修饰】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】）。

3.符号引用和直接引用

符号引用(class文件中)

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。

直接引用（运行时内存中）

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。

引用替换的时机

1、类加载的解析阶段（需要将一部分符号引用转换为直接引用）

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。

2、运行期间（动态分派）

4.class中的特殊字符串

类的全限定名

比如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`。

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

- 8种基本数据类型：除 long 和 boolean，其他都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

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 表示
- 静态初始化方法的方法名使用字符串 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。

5.javap命令

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。

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。

三、类加载详解

1.类加载的时机

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。

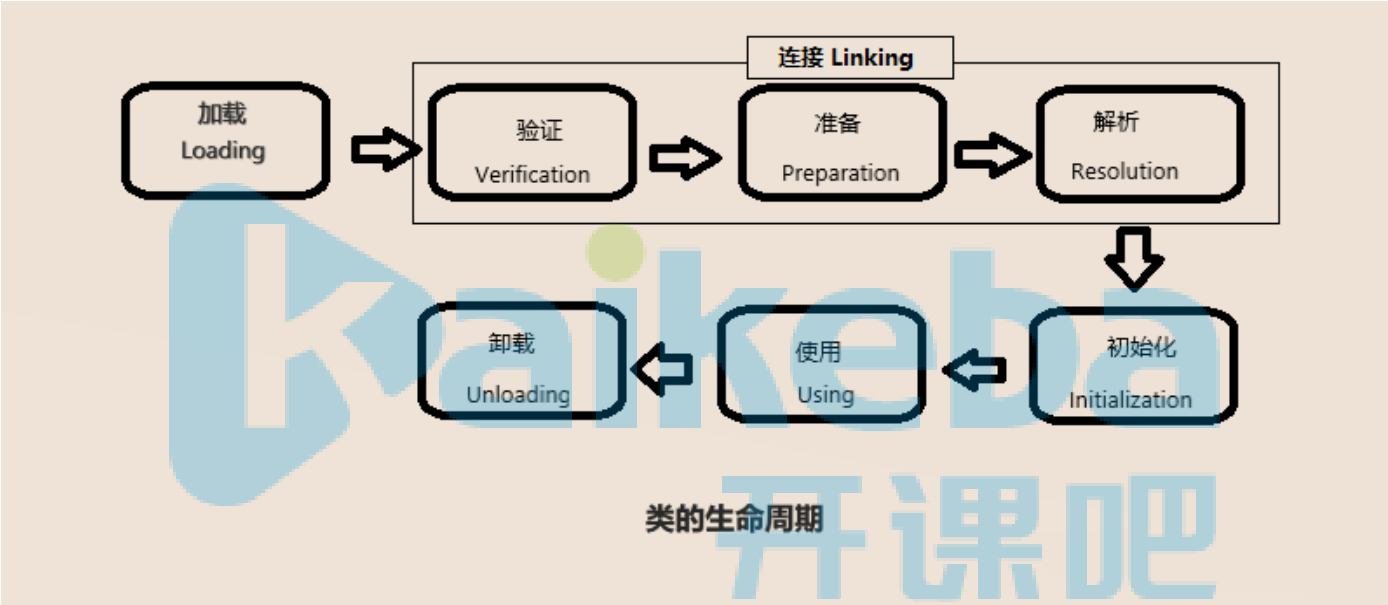
```
public class Student{
  private static int age ;
  public static void method(){
  }
}

//Student.age
//Student.method();
//new Student();
```

2. 使用`java.lang.reflect`包方法时对类进行反射调用的时候。

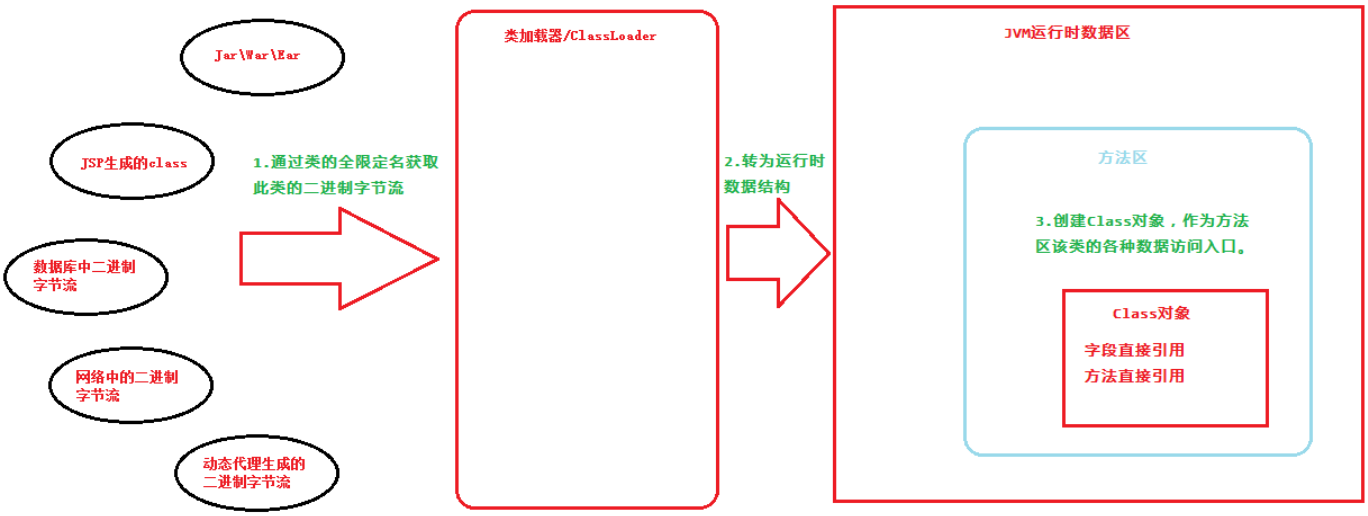
```
Class c = Class.forname("com.kkb.Student");
```

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（main），虚拟机会先执行这个主类的初始化。

2.类加载的过程



[加载]--class文件-->Class对象



加载的过程

在加载的过程中，JVM主要做3件事情：

- 全限定名--->二进制字节流(class文件)
- 字节流的静态存储结构--->方法区的运行时数据结构
- 创建java.lang.Class对象

加载源

- 本地class文件
- **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

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。

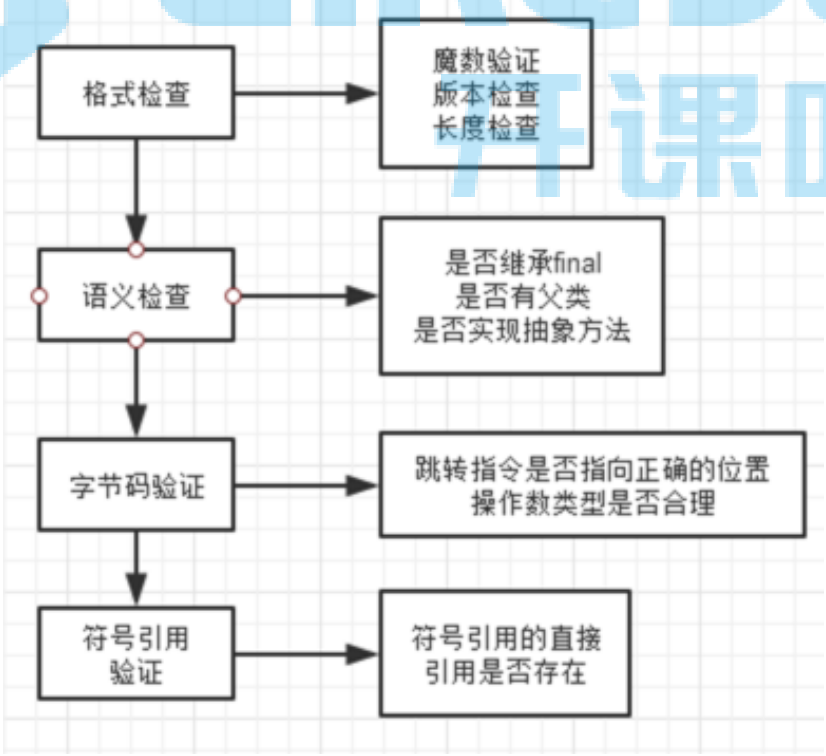
[验证]--各种检查

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]--为静态成员变量分配内存并初始化0值

> 1.7之前方法区
>
> 1.7之后堆

准备阶段主要完成两件事情：

- 为已在内存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

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（静态变量）]()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。

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]---将符号引用替换为直接引用

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：

   1. 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

2. 如果有，则查找结束；
3. 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，
4. 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，只是搜索的是接口不会有父类，因此，只递归向上搜索父接口就行了。

[初始化]----调用<clinit>方法

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.

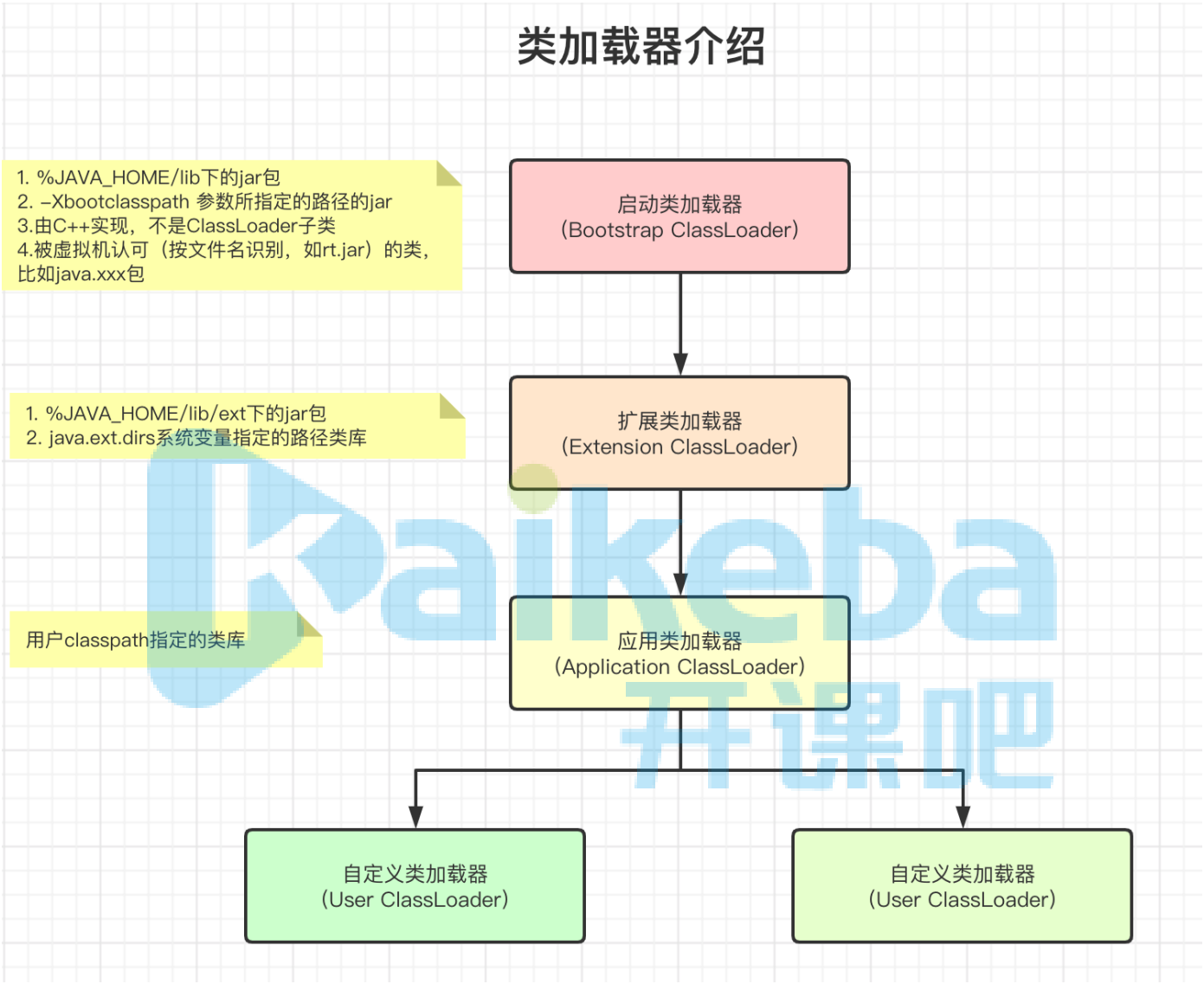
- 虚拟机会保证在多线程环境中一个类的方法被正确地加锁，同步.当多条线程同时去初始化一个类时，只会有一个线程去执行该类的方法，其它线程都被阻塞等待，直到活动线程执行方法完毕.其他线程虽会被阻塞，只要有一个方法执行完，其它线程唤醒后不会再进入方法.同一个类加载器下，一个类型只会初始化一次.

  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;
  }

}
```

3.类加载器介绍

processon地址：https://processon.com/diagraming/60b828685653bb4826da2786



自定义类加载器

为什么要定义自己的类加载器呢?

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，就只能自定义一个ClassLoader类了。

> 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

如何定义类加载器?

```
继承ClassLoader类，重写findClass()方法，loadClass()方法
```

自定义类加载器注意事项:

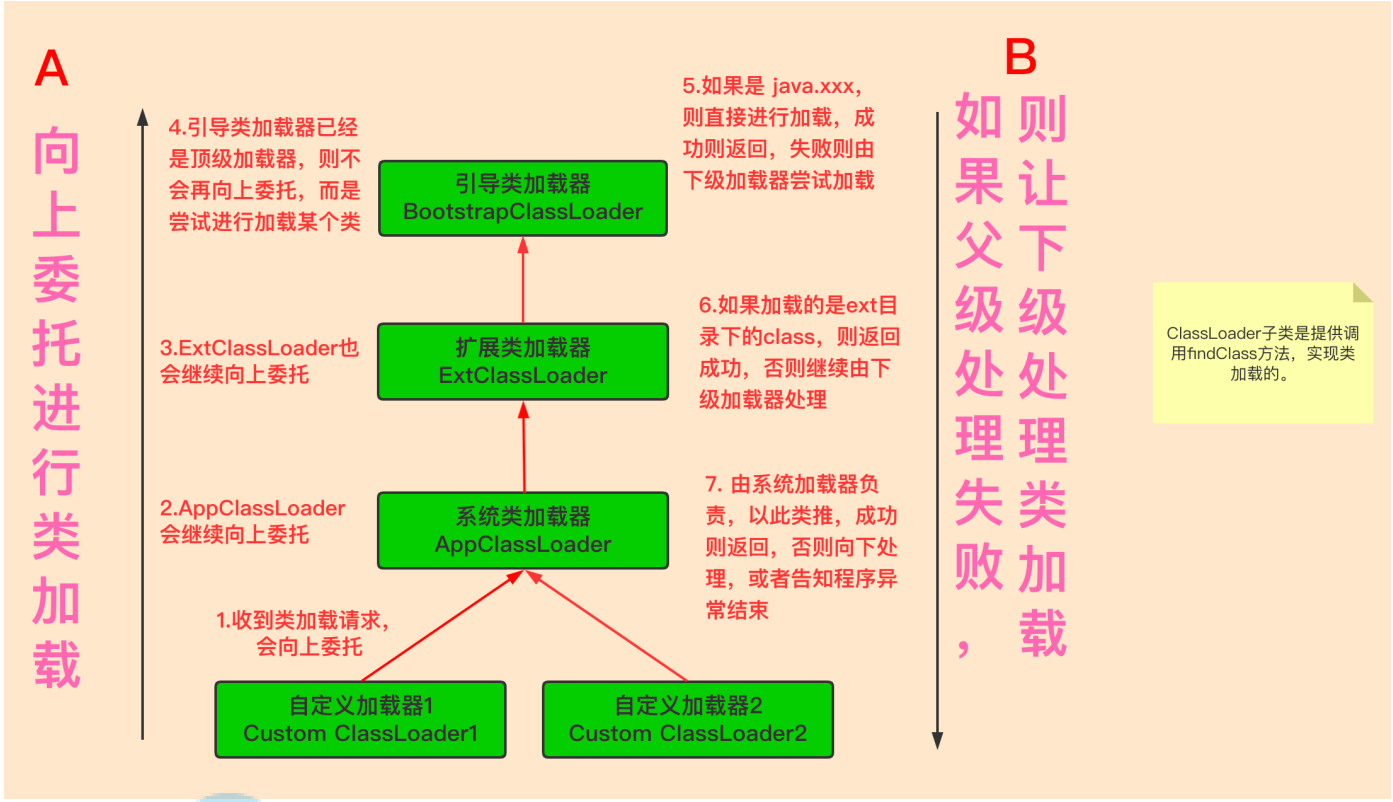
> 自定义类加载器需要去继承ClassLoader类。
>
> JDK1.2之前是重写ClassLoader类中的loadClass方法
>
> JDK1.2以后是重写ClassLoader类中的findClass方法

自定义类加载器的实现，不要去覆盖ClassLoader类的loadClass方法，去实现findClass方法，为什么呢?

4.双亲委派机制（1.2）

什么是双亲委派机制呢?

processon地址：https://www.processon.com/diagraming/60b78eace401fd3caea5b424



为什么要使用双亲委托这种模型呢？

考虑到安全因素，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。

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。

另外我们还可以试想一下，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。

如何判定两个class是相同?

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。

Proxy.newProxyInstance(ClassLoader)

双亲委派机制源码

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been
loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c =
findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if
class not found
                    // from the non-null parent class
loader
                }
```

```
                if (c == null) {
                    // If still not found, then invoke
findClass in order
                    // to find the class.
                    long t1 = System.nanoTime();



                    c = findClass(name);


                    // this is the defining class loader;
record the stats

 sun.misc.PerfCounter.getParentDelegationTime().addTime(t1
- t0);

 sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFro
m(t1);

 sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
```

5.破坏双亲委派模型

双亲委派机制的破坏，在JDK发展的过程中，是发生了多次：

第一次破坏

在 jdk 1.2 之前，那时候还没有双亲委派模型，不过已经有了 ClassLoader 这个抽象类，所以已经有人继承这个抽象类，重写 loadClass 方法来实现用户自定义类加载器。

而在 1.2 的时候要引入双亲委派模型，为了向前兼容， loadClass 这个方法还得保留着使之得以重写，新搞了个 findClass 方法让用户去重写，并呼吁大家不要重写 loadClass 只要重写 findClass。

这就是第一次对双亲委派模型的破坏，因为双亲委派的逻辑在 loadClass 上，但是又允许重写 loadClass，重写了之后就可以破坏委派逻辑了。

第二次破坏

双亲委派机制，是一种自上而下的加载需求，越往上类越基础。在实际的应用中双亲委派解决了java 基础类统一加载的问题，但是却着实存在着一定的缺陷。jdk中的基础类作为典型的api被用户调用，但是也存在被api调用用户代码的情况，典型的如SPI代码。

SPI机制简介

SPI的全名为Service Provider Interface，主要是应用于厂商自定义组件或插件中。在java.util.ServiceLoader的文档里有比较详细的介绍。简单的总结下java SPI机制的思想：

> 我们系统里抽象的各个模块，往往有很多不同的实现方案，比如日志模块、xml解析模块、jdbc模块等方案。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。Java SPI就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。

DriverManager源码

```
      static {
        loadInitialDrivers();
        println("JDBC DriverManager initialized");
    }

    private static void loadInitialDrivers() {
        String drivers;
        try {
            drivers = AccessController.doPrivileged(new
PrivilegedAction<String>() {
                public String run() {
                    return
System.getProperty("jdbc.drivers");
                }
            });
        } catch (Exception ex) {
            drivers = null;
        }
```

```
        AccessController.doPrivileged(new
PrivilegedAction<Void>() {
            public Void run() {

                ServiceLoader<Driver> loadedDrivers =
ServiceLoader.load(Driver.class);
                Iterator<Driver> driversIterator =
loadedDrivers.iterator();
                try{
                    while(driversIterator.hasNext()) {
                        driversIterator.next();
                    }
                } catch(Throwable t) {
                // Do nothing
                }
                return null;
            }
        });

        println("DriverManager.initialize: jdbc.drivers =
" + drivers);

        if (drivers == null || drivers.equals("")) {
            return;
        }
        String[] driversList = drivers.split(":");
        println("number of Drivers:" +
driversList.length);
        for (String aDriver : driversList) {
            try {
                println("DriverManager.Initialize: loading
" + aDriver);
                Class.forName(aDriver, true,
```

```
 ClassLoader.getSystemClassLoader());
            } catch (Exception ex) {
                println("DriverManager.Initialize: load
failed: " + ex);
            }
        }
    }
```

如果出现SPI相关代码时，我们应该如何解决基础类去加载用户代码类呢？这个时候，JVM不得不妥协，推出线程上下文类加载器的概念，去解决该问题。

线程上下文类加载器

（Thread Context ClassLoader）

processon地址：https://www.processon.com/diagraming/60b887d25653bb353c98f225

设置线程上下文源码

```
public Launcher() {
        Launcher.ExtClassLoader var1;
        // 扩展类加载器
        try {
            var1 =
Launcher.ExtClassLoader.getExtClassLoader();
        } catch (IOException var10) {
```

```
            throw new InternalError("Could not create
extension class loader", var10);
        }

        // 应用类加载器/系统类加载器
        try {
            this.loader =
Launcher.AppClassLoader.getAppClassLoader(var1);
        } catch (IOException var9) {
            throw new InternalError("Could not create
application class loader", var9);
        }

        // 线程上下文类加载器

 Thread.currentThread().setContextClassLoader(this.loader)
;
        String var2 =
System.getProperty("java.security.manager");
        if (var2 != null) {
            SecurityManager var3 = null;
            if (!"".equals(var2) &&
!"default".equals(var2)) {
                try {
                    var3 =
(SecurityManager)this.loader.loadClass(var2).newInstance()
;
                } catch (IllegalAccessException var5) {
                } catch (InstantiationException var6) {
                } catch (ClassNotFoundException var7) {
                } catch (ClassCastException var8) {
                }
            } else {
```

```
                var3 = new SecurityManager();
            }

            if (var3 == null) {
                throw new InternalError("Could not create
SecurityManager: " + var2);
            }

            System.setSecurityManager(var3);
        }

    }
```

获取线程上下文源码

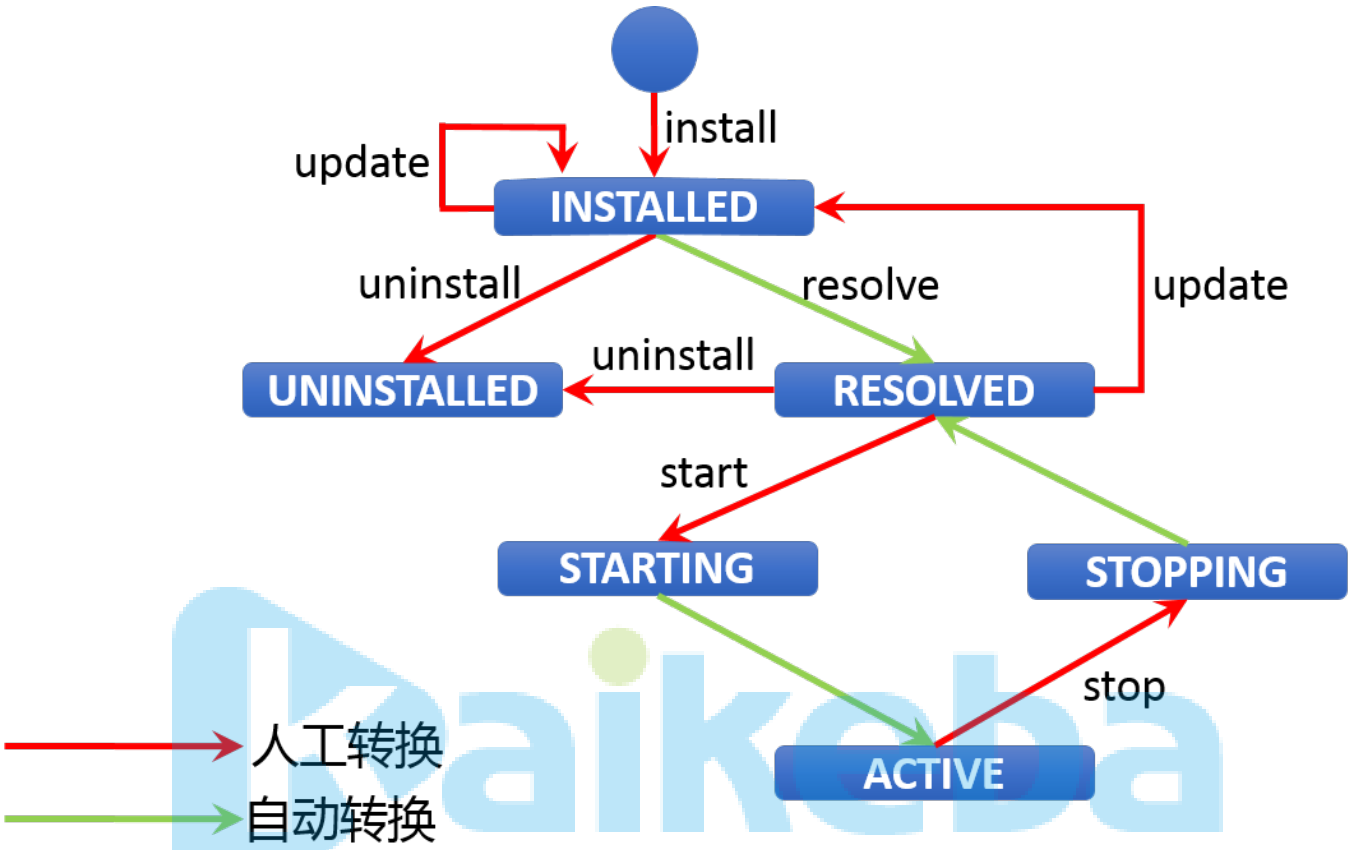
```
    public static <S> ServiceLoader<S> load(Class<S>
service) {
        ClassLoader cl =
Thread.currentThread().getContextClassLoader();
        return ServiceLoader.load(service, cl);
    }
```

第三次破坏

这次破坏是为了满足热部署的需求，不停机更新这对企业来说至关重要，毕竟停机是大事。

OSGI 就是利用自定义的类加载器机制来完成模块化热部署，而它实现的类加载机制就没有完全遵循自下而上的委托，有很多平级之间的类加载器查找，具体就不展开了，有兴趣可以自行研究一下。

OSGi：Bundle生命周期



在上图中，人工转换箭头线上的install、update、resolve、start、stop是通过在控制台输入对应命令来触发。Bundle生命周期过程之中的6种状分别为：

1. UNINSTALLED（未安装）：状态值为整数1。此时Bundle中的资源是不可用的。
2. INSTALLED（已安装）：状态值为整数2。此时Bundle已经通过了OSGi框架的有效性校验并分配了Bundle ID，本地资源已加载，但尚未对其依赖关系进行解析处理。
3. RESOLVED（已解析）：状态值为整数4。此时Bundle已经完成了依赖关系解析并已经找到所有依赖包，而且自身导出的Package已可以被其它Bundle导入使用。在此种状态的Bundle要么是已经准备好运行，要么就是被停止了。
4. STARTING（启动中）：状态值为整数8。此时Bundle的BundleActivator

的start()方法已经被调用但是尚未返回。如果start()方法正常执行结束，Bundle将自动转换到ACTIVE状态； 否则如果start()方法抛出了异常，Bundle将退回到RESOLVED状态。

5. STOPPING（停止中）：状态值为整数16。此时Bundle的BundleActivator的stop()方法已经被调用但是尚未返回。无论stop()是正常结束还是抛出了异常，在这个方法退出之后，Bundle的状态都将转为RESOLVED。
6. ACTIVE（已激活）：状态值为整数32。Bundle处于激活状态，说明BundleActivator的start()方法已经执行完毕，如果没有其他动作，Bundle将继续维持ACTIVE状态。

OSGi：Bundle解析

Class Loader在Bundle被正确解析（状态变成Resolved）之后创建，而只有Bundle 中的所有包约束条件都满足后，对应的Bundle才能被正确解析完毕。

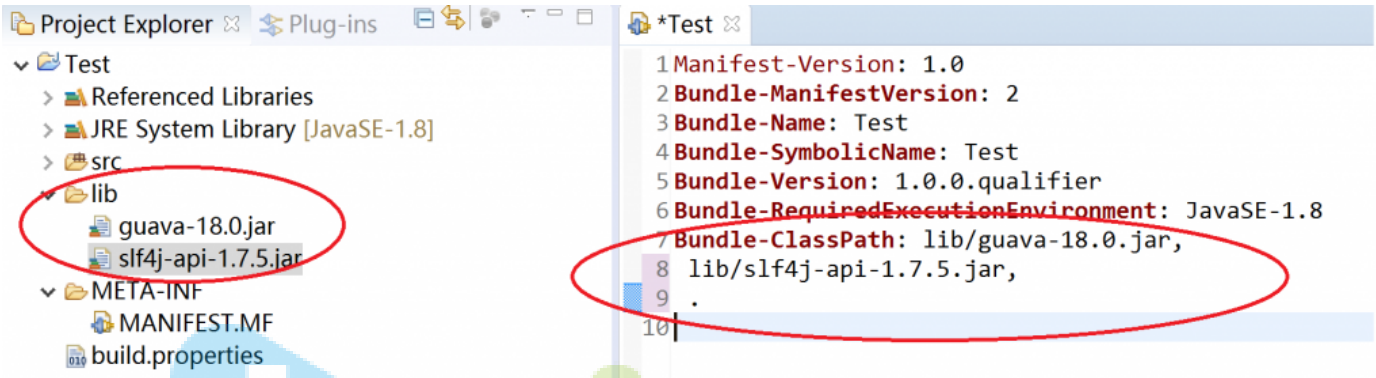
Bundle的解析由定义在MANIFEST.MF文件中的四个配置项定义：

1. **Import-Package**：定义当前Bundle需要导入的其它包（Package），一般位于其它Bundle之中并且被定义为Export-Package。
2. **Export-Package**：定义当前Bundle要导出的包。被导出的包中的类资源可以被其它Bundle导入，可以被定义到Import-Package中。
3. **Require-Bundle**：定义当前Bundle需要依赖的Bundle。
4. **DynamicImport-Package**：定义需要动态导入的包。这里定义的包在Bundle解析过程中不会被使用到，而是会在运行时被动态解析并加载。

OSGi：Bundle-ClassPath

在MANIFEST.MF文件中定义的Bundle-Classpath会描述Classpath范围，告诉Classloader去哪里查找类。

下图是一个简单的样例：



OSGi：类加载机制

Bundle的加载策略、如何导入和导出代码是在OSGi规范的模块（Modules）层实现的。

OSGi框架对于每个Bundle（非Fragment Bundle）都会创建一个单独的类加载器（Class Loader），不过对于Class Loader的创建可能会延迟到真正需要它时才会发生。

processon地址：https://www.processon.com/diagraming/60b845757d9c0819e4a0fee8

Bundle的Class Loader搜索类资源的规则简要介绍如下：

1. 如类资源属于java.* 包，则将加载请求委托给父 Class Loader；
2. 如类资源定义在 OSGi框架的启动委托列表（osgi.framework.bootdelegation）中，则将加载请求委托给父Class Loader；例如：

```
osgi.framework.bootdelegation=*
osgi.framework.bootdelegation=sun.*,com.sun.*
```

3. 如类资源属于在 Import-Package 中定义的包，则框架会将加载请求委托给导出此包的Bundle的Class Loader；
4. 如类资源属于在Require-Bundle 中定义的 Bundle，则框架会将加载请求委托给此Bundle的Class Loader；
5. Bundle搜索自己的Bundle-Classpath中定义的类资源；
6. Bundle搜索属于该Bundle的Fragment的类资源；
7. 判断是否找到导出（Export-Package）了对应资源，如果仍未找到进入动态导入查找；
8. 若类在DynamicImport-Package中定义，则开始尝试在运行环境中寻找符合条件的 Bundle，框架会将加载请求委托给导出此包的Bundle的Class Loader；

> 注意DynamicImport-Package: *使用了星号来对所有资源进行通配，这也意味着对应的Bundle可以“看到”任何可能访问到的资源，这个选项应该尽量少地使用，除非别无它法。

9. 如果在经过上面一系列步骤后，仍然没有正确地加载到类资源，则OSGi框架会向外抛出类未发现异常。

总结

JVM的类加载机制，是自上而下的加载过程。

OSGi的类加载机制，既能在Bundle内部按照JVM的类机制机制去加载，本身也会按照Bundle之间横向委托的方式进行类加载，所以它是一种网状结构的类加载方式。