

参考书籍或资料

- * 《Java 虚拟机规范（Java SE 8版）》
- * 《深入理解 Java 虚拟机 第二版》
- * 《深入理解 Java 虚拟机 第三版》
- * 《实战 Java 虚拟机》

1.3 Java及JVM简介

TIOBE语言热度排行榜：<https://www.tiobe.com/tiobe-index/>

世界上没有最好的编程语言，如果有，我相信一定是JAVA。

1.4 Java发展的重大事件

1995年5月23日，Java语言诞生

1996年1月，第一个JDK-JDK1.0诞生

1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入JAVA技术

1996年9月，约8.3万个网页应用了JAVA技术来制作

1997年2月18日，JDK1.1发布

1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议规模之纪录

1997年9月，JavaDeveloperConnection社区成员超过十万

1998年2月，JDK1.1被下载超过2,000,000次

1998年12月8日，JAVA2企业平台J2EE发布

1999年6月，SUN公司发布Java的三个版本：标准版（J2SE）、企业版（J2EE）和微型版（J2ME）

2000年5月8日，JDK1.3发布

2000年5月29日，JDK1.4发布

2001年6月5日，NOKIA宣布，到2003年将出售1亿部支持Java的手机

2001年9月24日，J2EE1.3发布

2002年2月26日，J2SE1.4发布，自此Java的计算能力有了大幅提升

2004年9月30日18:00PM，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名，以取消其中的数字“2”：J2EE更名为Java EE，J2SE更名为Java SE，J2ME更名为Java ME

2006年12月，SUN公司发布JRE6.0

2010年9月，JDK7.0已经发布，增加了简单闭包功能。

学习JVM的动力

我们为什么要学习JVM呢？我们来看一下官方解释：

Java官网：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html#jvms-1.2>

1.2. The Java Virtual Machine

The Java Virtual Machine is the cornerstone of the Java platform. It is the component of the technology responsible for its hardware- and operating system-independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. It is reasonably common to implement a programming language using a virtual machine; the best-known virtual machine may be the P-Code machine of UCSD Pascal.

The first prototype implementation of the Java Virtual Machine, done at Sun Microsystems, Inc., emulated the Java Virtual Machine instruction set in software hosted by a handheld device that resembled a contemporary Personal Digital Assistant (PDA). Oracle's current implementations emulate the Java Virtual Machine on mobile, desktop and server devices, but the Java Virtual Machine does not assume any particular implementation technology, host hardware, or host operating system. It is inherently interpreted, but can just as well be implemented by compiling its instruction set to that of a silicon CPU. It may also be implemented in microcode or directly in silicon.

The Java Virtual Machine knows nothing of the Java programming language, only of a particular binary format, the `.class` file format. A `.class` file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

For the sake of security, the Java Virtual Machine imposes strong syntactic and structural constraints on the code in a `.class` file. However, any language with functionality that can be expressed in terms of a valid `.class` file can be hosted by the Java Virtual Machine. Attracted by a generally available, machine-independent platform, implementors of other languages can turn to the Java Virtual Machine as a delivery vehicle for their languages.

The Java Virtual Machine specified here is compatible with the Java SE 8 platform, and supports the Java programming language specified in *The Java Language Specification, Java SE 8 Edition*.

1. Java虚拟机是Java平台的基石，其负责其硬件和操作系统的独立性，其编译的代码很小以及保护用户免受恶意程序攻击的能力。

2. Java虚拟机是一种抽象计算机，像真正的计算机一样，它有一个指令集并在运行时操作各种内存区域。

3. Java虚拟机不承担任何特定的实现技术、主机硬件或主机操作系统，它本身并没有被解释。

4. Java虚拟机不知道Java编程语言，只知道特定的二进制格式，即 `class` 文件格式，`class` 文件包含Java虚拟机指令（或字节码）和符号表，以及其他辅助信息。

5. 出于安全考虑，Java虚拟机对 `class` 文件中的代码施加了强大的语法和结构约束，但是，任何具有可以用有效 `class` 文件表示的功能的语言都可以由Java虚拟机托管，由通用的、与机器无关的平台吸引，其他语言的实现者可以将Java虚拟机作为其语言的交付工具。

这是我针对上述语言精炼的一些翻译。这些理由就足以让一部分同学深入的学习JVM了。

但是上述这些理由可能都不足以让我们所有的同学信服，因为现在大部分的攻城狮以及程序猿可能更关注的是跟我们相关的一些问题，那么在这里我就换一种方式让你接受我们的JVM。

- 如果你在线上遇到了OOM，你是否会束手无策。
- 线上卡顿是否可能是因为频繁Full GC造成的。
- 新项目上线，服务器数量以及配置不足，对于性能的扩展只能靠服务器的增加，而不能通过JVM的调优达到实现服务器性能的突破。
- 面试经常会问到VM的一些问题，但是当面试官问到你实际的落地点时，你就会茫然不知所措，没有条理性，或者答非所问。

我会慢慢的在学习中理解上面的这些描述。不过在学习之前，由于可能读者的水平可能会有一些不同，所以本堂课我首先会让你们了解一些基础性的知识。

1.2 编程语言

编程语言（英语：programming language），是用来定义[计算机程序]的[形式语言]。它是一种被[标准化]的交流技巧，用来向[计算机]发出指令。一种能够让[程序员]准确地定义计算机所需要使用数据的计算机语言，并精确地定义在不同情况下所应当采取的行动。

说白了就是让人类能够和计算机沟通，所以要学习计算机能够懂的语言

1.3 计算机[硬件]能够懂的语言

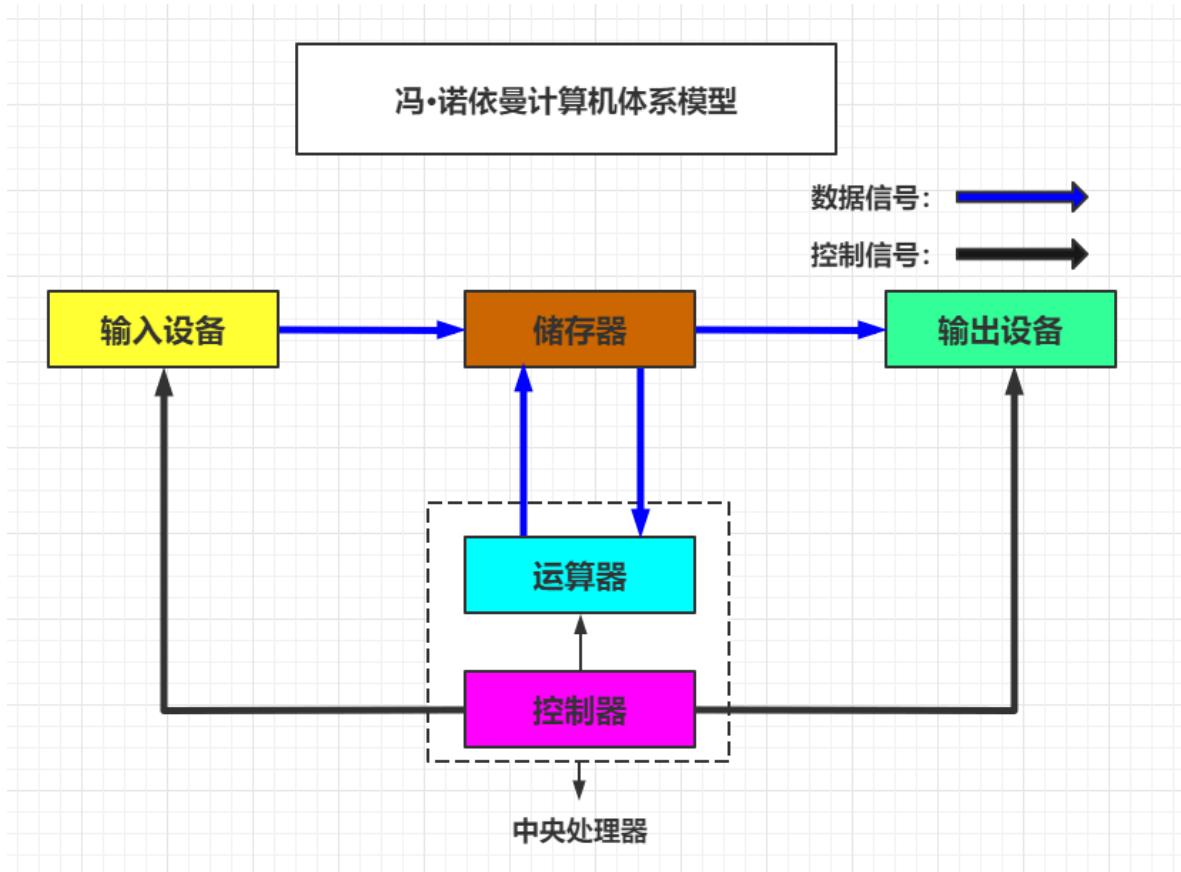
1.3.1 计算机发展史

- 1946-1958：电子管计算机

- 1958-1964: 晶体管计算机
- 1964-1970: 集成电路计算机
- 1970-至今: 大规模集成电路计算机
- 猜想: 未来以蛋白质分子作为原材料, 量子计算机[已经有了]

1.3.2 计算机体系结构

| 遵循冯诺依曼计算机结构



1.3.3 计算机处理数据过程

- (1) 提取阶段:由输入设备把原始数据或信息输入给计算机存储器存起来
- (2) 解码阶段:根据CPU的指令集架构(ISA)定义将数值解译为指令
- (3) 执行阶段:再由控制器把需要处理或计算的数据调入运算器
- (4) 最终阶段:由输出设备把最后运算结果输出

| 本质上就是CPU取数据指令然后返回

CPU=存储器+运算器+控制器

1.3.4 机器语言

我们把CPU能够直接认识的数据指令, 称为机器语言, 也就是010101001这种形式

1.3.5 不同厂商的CPU

单核、双核、多核

Intel、AMD、IBM等

| 不同CPU使用的CPU指令集是不一样的, 这就会有不兼容的问题

而且要是直接操作01这种形式的，非常麻烦并且容易出错，硬件资源管理起来也不方便

1.3.6 操作系统

- 向下对接指令系统、管理硬件资源
- 向上提供给用户简单的操作命令和界面

1.3.7 汇编语言

低级语言，通过汇编器翻译成机器语言

MOV、PUSH、ADD等

对机器友好，执行效率比较高，移植性差

但是人类操作起来还是不太方便，或者需要专业的人员

1.3.8 高级语言

C、C++、Java、Python、Golang等

最终肯定还是要转换成机器能够懂的机器语言

1.3.9 编译型和解释型

1.3.9.1 编译型

使用专门的编译器，针对**特定的平台**，将高级语言源代码一次性的编译成

可被该平台硬件执行的机器码，并包装成该平台所能识别的可执行性程序的格式。

C、C++、GoLang

编译型语言：

执行速度快、效率高；依靠编译器、跨平台性差些。

把做好的源程序全部编译成二进制代码的可运行程序。然后，可直接运行这个程序。

1.3.9.2 解释型

使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行。

是代码在执行时才被解释器一行行动态翻译和执行，而不是在执行之前就完成翻译。

Python、Javascrip

解释型语言：

执行速度慢、效率低；依靠解释器、跨平台性好。

把做好的源程序翻译一句，然后执行一句，直至结束。

1.3.9.3 Java呢？

Java属于编译型+解释型的高级语言

其实并不是因为有javac将Java源码编译成class文件，才说Java属于编译+解释语言，因为在这个编译器编译之后，生成的类文件不能直接在对应的平台上运行。

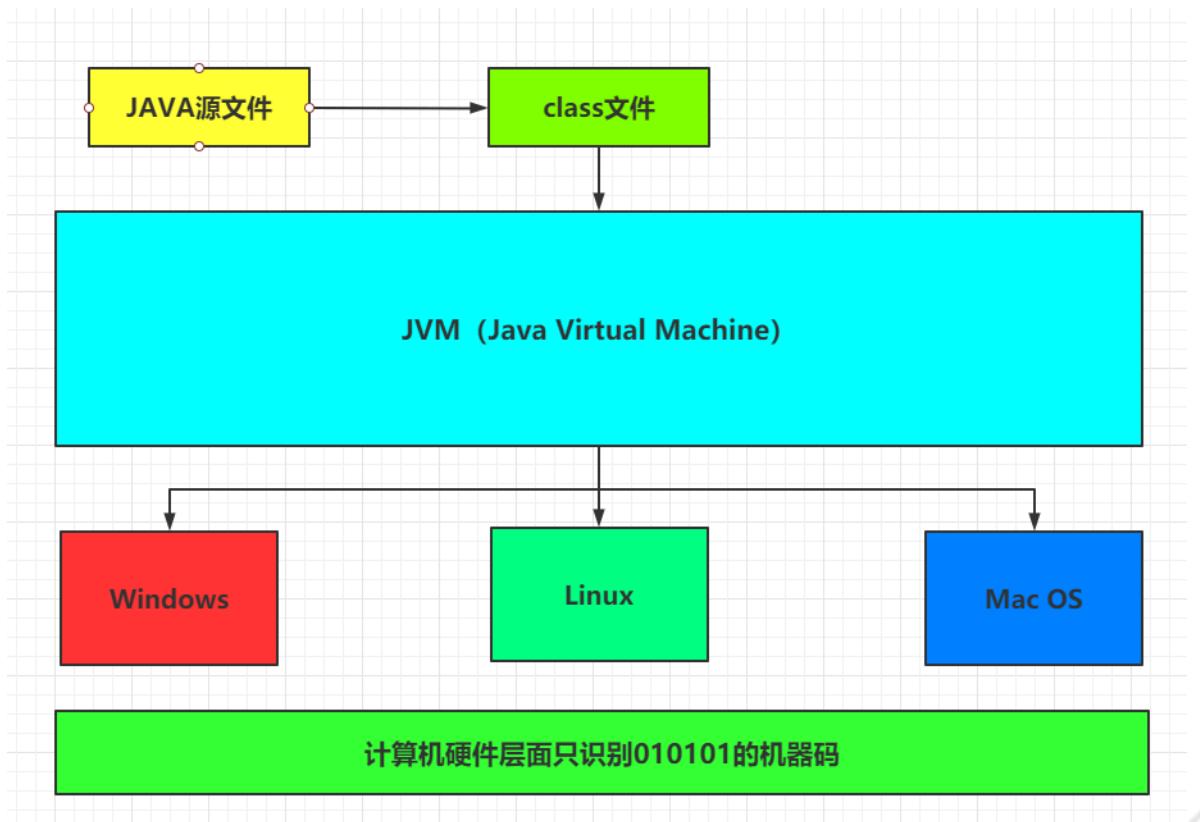
那为何又说Java是编译+解释语言呢？因为class文件最终是通过JVM来翻译才能在对应的平台上运行，而这个翻译大多数时候是解释的过程，但是也会有编译，称之为运行时编译，即JIT(Just In Time)。

综上所述，Java是一门编译型+解释型的高级语言。

1.4 So JVM是什么？

Java Virtual Machine(Java虚拟机)

Write Once Run Anywhere

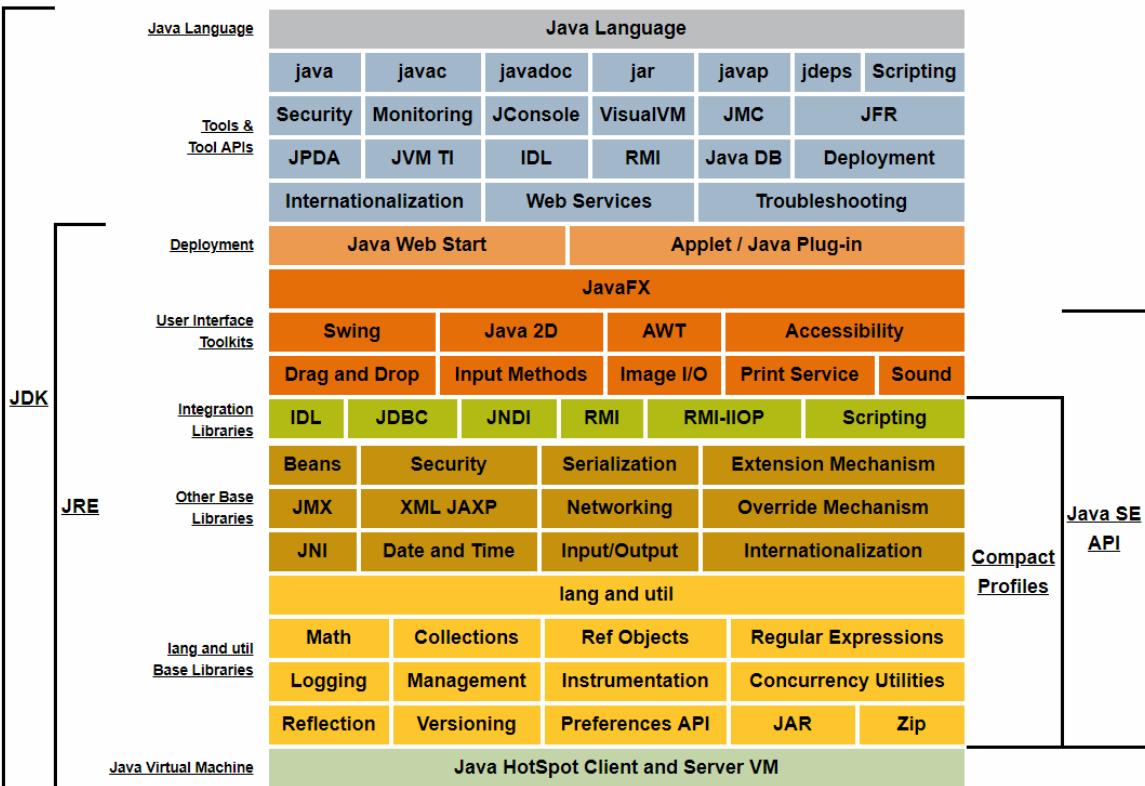


1.5 JDK JRE JVM

Java官网：<https://docs.oracle.com/javase/8/>

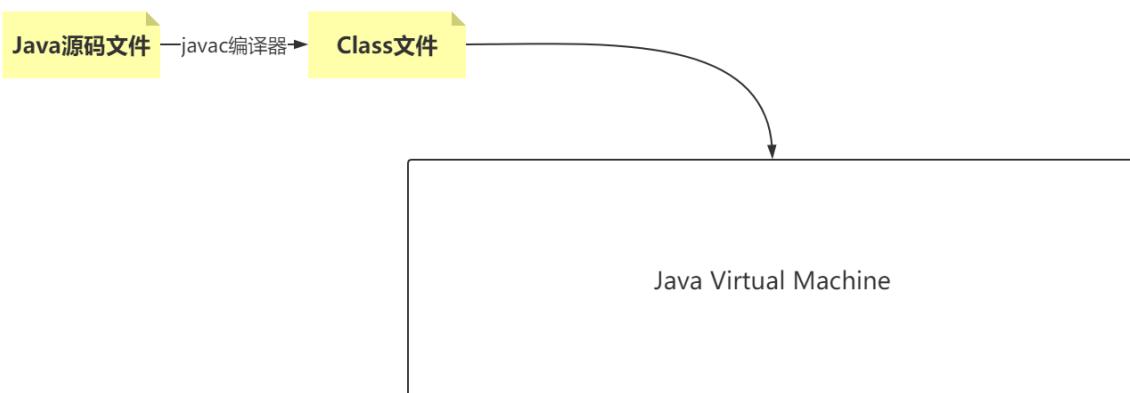
Reference -> Developer Guides -> 定位到:<https://docs.oracle.com/javase/8/docs/index.html>

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8, plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the JRE includes components not required by the Java SE specification, including both standard and non-standard Java components.



02 JVM到底该学习什么

或者换句话说，JVM到底从哪边开始学习起？



- (1) 源码到类文件
- (2) 类文件到JVM
- (3) JVM各种折腾[内部结构、执行方式、垃圾回收、本地调用等]

2.1 源码到类文件

2.1.1 源码demo

```

class Person{
    private String name="Jack";
    private int age;
    private final double salary=100;
    private static String address;
    private final static String hobby="Programming";
}
  
```

```

private static Object obj=new Object();
public void say(){
    System.out.println("person say...");
}
public static int calc(int op1,int op2){
    op1=3;
    int result=op1+op2;
    Object obj=new Object();
    return result;
}
public static void main(String[] args){
    calc(1,2);
}
}

```

编译: javac -g:vars Person.java ---> Person.class

2.4 分析编译器干了什么事

Person.java -> 词法分析器 -> tokens流 -> 语法分析器 -> 语法树/抽象语法树 -> 语义分析器 -> 注解抽象语法树 -> 字节码生成器 -> Person.class文件

由上可知，其实我们的编译器其实做的事情其实就是“对等信息转换”。JAVA文件中的信息其实跟我们Class文件中的信息，其实是一样的。

2.1.3 类文件(Class文件)

2.1.3.1 16进制

```

cafe babe 0000 0034 003f 0a00 0a00 2b08
002c 0900 0d00 2d06 4059 0000 0000 0000
0900 0d00 2e09 002f 0030 0800 310a 0032
0033 0700 340a 000d 0035 0900 0d00 3607
0037 0100 046e 616d 6501 0012 4c6a 6176
612f 6c61 6e67 2f53 7472 696e 673b 0100
0361 6765 0100 0149 0100 0673 616c 6172
7901 0001 4401 000d 436f 6e73 7461 6e74
.....

```

2.1.3.2 The ClassFile Structure

官网：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html>

```

classFile {
    u4          magic;
    u2          minor_version;
    u2          major_version;
    u2          constant_pool_count;
    cp_info    constant_pool[constant_pool_count-1];
    u2          access_flags;
    u2          this_class;
    u2          super_class;
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info fields[fields_count];
}

```

```
    u2          methods_count;
    method_info  methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

2.1.3.3 Simple analysis

u4 :cafebabe

magic:The magic item supplies the magic number identifying the class file format

u2+u2:0000+0034, 34等于10进制的52, 表示JDK8

minor_version
major_version

u2:003f=63(10进制)

constant_pool_count:
The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one.

表示常量池中的数量是62

cp_info constant_pool[constant_pool_count-1]

The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte.

The constant_pool table is indexed from 1 to constant_pool_count - 1.

常量池主要存储两方面内容：字面量(Literal)和符号引用(Symbolic References)

字面量：文本字符串，final修饰等

符号引用：类和接口的全限定名、字段名称和描述符、方法名称和描述符

2.6 反编译验证

用javap指令验证上述猜想正确性

编译指令：javap -v -p Person.class

进行反编译之后，查看字节码信息和指令等信息

是否有一种感觉？

JVM相对class文件来说可以理解为是操作系统；class文件相对JVM来说可以理解为是汇编语言或者机器语言。

```

Classfile /D:/temp/jvm/labs/Person.class
Last modified 2020-3-26; size 938 bytes
MD5 checksum 0c4ee2ae34eb7e62f4bf621dc45b3325
class Person
    minor version: 0
    major version: 52 } ← 版本号
    flags: ACC_SUPER
Constant pool:
#1 = Methodref      #10.#43          // java/lang/Object."<init>":()V
#2 = String          #44              // Jack
#3 = Fieldref        #13.#45          // Person.name:Ljava/lang/String;
#4 = Double           100.0d
#6 = Fieldref        #13.#46          // Person.salary:D
#7 = Fieldref        #47.#48          // java/lang/System.out:Ljava/io/PrintStream;
#8 = String           #49              // person say...
#9 = Methodref        #50.#51          // java/io/PrintStream.println:(Ljava/lang/String;)V
#10 = Class           #52              // java/lang/Object
#11 = Methodref        #13.#53          // Person.calc:(II)I
#12 = Fieldref        #13.#54          // Person.obj:Ljava/lang/Object;
#13 = Class           #55              // Person
#14 = Utf8             name
#15 = Utf8             Ljava/lang/String;
#16 = Utf8             age
#17 = Utf8             I
#18 = Utf8             salary
#19 = Utf8             D
#20 = Utf8             ConstantValue
#21 = Utf8             address
#22 = Utf8             hobby
#23 = String           #56              // Programming
#24 = Utf8             obj
#25 = Utf8             Ljava/lang/Object;
#26 = Utf8             <init>
#27 = Utf8             ()V
#28 = Utf8             Code
#29 = Utf8             LocalVariableTable
#30 = Utf8             this
#31 = Utf8             LPerson;
#32 = Utf8             say
#33 = Utf8             calc
#34 = Utf8             (II)I
#35 = Utf8             op1
#36 = Utf8             op2
#37 = Utf8             result
#38 = Utf8             main
#39 = Utf8             ([Ljava/lang/String;)V
#40 = Utf8             args
#41 = Utf8             [Ljava/lang/String;
#42 = Utf8             <clinit>
#43 = NameAndType     #26:#27          // "<init>":()V
#44 = Utf8             Jack
#45 = NameAndType     #14:#15          // name:Ljava/lang/String;
#46 = NameAndType     #18:#19          // salary:D
#47 = Class            #57              // java/lang/System
#48 = NameAndType     #58:#59          // out:Ljava/io/PrintStream;
#49 = Utf8             person say...
#50 = Class            #60              // java/io/PrintStream
#51 = NameAndType     #61:#62          // println:(Ljava/lang/String;)V
#52 = Utf8             java/lang/Object
#53 = NameAndType     #33:#34          // calc:(II)I
#54 = NameAndType     #24:#25          // obj:Ljava/lang/Object;
#55 = Utf8             Person
#56 = Utf8             Programming
#57 = Utf8             java/lang/System
#58 = Utf8             out
#59 = Utf8             Ljava/io/PrintStream;
#60 = Utf8             java/io/PrintStream
#61 = Utf8             println
#62 = Utf8             (Ljava/lang/String;)V
{
    private java.lang.String name;
        descriptor: Ljava/lang/String;
        flags: ACC_PRIVATE
    private int age;
        descriptor: I
        flags: ACC_PRIVATE
    private final double salary;
        descriptor: D
        flags: ACC_PRIVATE, ACC_FINAL
        ConstantValue: double 100.0d
    private static java.lang.String address;
        descriptor: Ljava/lang/String;
        flags: ACC_PRIVATE, ACC_STATIC
    private static final java.lang.String hobby;
        descriptor: Ljava/lang/String;
        flags: ACC_PRIVATE, ACC_STATIC, ACC_FINAL
        ConstantValue: String Programming
    private static java.lang.Object obj;
        descriptor: Ljava/lang/Object;
        flags: ACC_PRIVATE, ACC_STATIC
    Person();
        descriptor: ()V
        flags:
        Code:
            stack=3, locals=1, args_size=1
            0: aload_0
            1: invokespecial #1          // Method java/lang/Object."<init>":()V
            4: aload_0
            5: ldc             #2          // String Jack
            7: putfield        #3          // Field name:Ljava/lang/String;
            10: aload_0
            11: ldc2_w         #4          // double 100.0d
            14: putfield        #6          // Field salary:D
}

```

magic

版本号

常量池

字段表集合

```

    17: return
LocalVariableTable:
Start Length Slot Name   Signature
      0     18     0  this  LPerson;

public void say();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=1, args_size=1
  0: getstatic    #7           // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc         #8           // String person say...
  5: invokevirtual #9          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LocalVariableTable:
Start Length Slot Name   Signature
      0      9     0  this  LPerson;

public static int calc(int, int);
descriptor: (II)I
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=2
  0: iconst_3
  1: istore_0
  2: iload_0
  3: iload_1
  4: iadd
  5: istore_2
  6: new           #10          // class java/lang/Object
  9: dup
 10: invokespecial #1          // Method java/lang/Object."<init>":()V
 13: astore_3
 14: iload_2
 15: ireturn
LocalVariableTable:
Start Length Slot Name   Signature
      0     16     0  op1   I
      0     16     1  op2   I
      6     10     2 result  I
     14      2     3  obj   Ljava/lang/Object;

public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
  0: iconst_1
  1: iconst_2
  2: invokestatic #11          // Method calc:(II)I
  5: pop
  6: return
LocalVariableTable:
Start Length Slot Name   Signature
      0      7     0  args  [Ljava/lang/String;

static {};
descriptor: ()V
flags: ACC_STATIC
Code:
stack=2, locals=0, args_size=0
  0: new           #10          // class java/lang/Object
  3: dup
  4: invokespecial #1          // Method java/lang/Object."<init>":()V
  7: putstatic    #12          // Field obj:Ljava/lang/Object;
 10: return
}

```

方法表集合

2.1.3.5 Continous analysis

上面分析到常量池中常量的数量是58，接下来我们来具体分析一下这58个常量

cp_info constant_pool[constant_pool_count-1] 也就是这块包括的信息

cp_info其实就是一个表格的形式

All constant_pool table entries have the following general format:

```

cp_info {
  ul tag;
  ul info[];
}

```

官网：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4>

Table 4.4-A. Constant pool tags

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

(1)往下数一个u1, 即0a->10:代表的是CONSTANT_Methodref, 表示这是一个方法引用

```
CONSTANT_Fieldref_info {  
    u1 tag;  
    u2 class_index;  
    u2 name_and_type_index;  
}
```

往下数u2和u2

u2, 即00 0a->10:代表的是class_index, 表示该方法所属的类在常量池中的索引

u2, 即00 2b->43:代表的是name_and_type_index, 表示该方法的名称和类型的索引

```
#1 = Methodref      #10, #43
```

(2)往下数u1, 即08->8:表示的是CONSTANT_String, 表示字符串类型

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

往下数u2

u2, 即00 2c->44:代表的是string_index

```
#1 = Methodref      #10, #43  
#2 = String         #44
```

(3)往下数u1, 即09->9:表示的是CONSTANT_Fieldref, 表示字段类型

```
CONSTANT_Fieldref_info {  
    u1 tag;  
    u2 class_index;  
    u2 name_and_type_index;  
}
```

往下数u2和u2

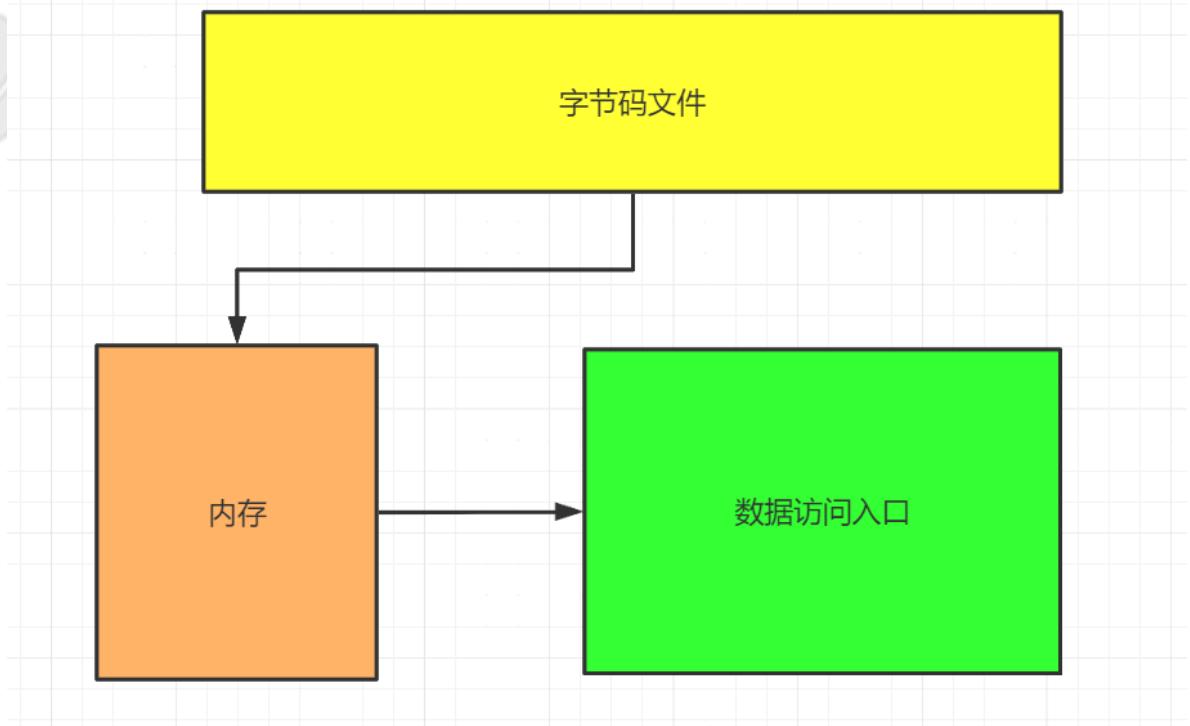
u2, 即00 0d->13:代表的是class_index

u2, 即00 2d->45:代表的是name_and_type_index

```
#1 = Methodref      #10.#43  
#2 = String         #44  
#3 = Fieldref       #13.#45
```

类加载机制

类加载机制是指我们将类的字节码文件所包含的数据读入内存，同时我们会生成数据的访问入口的一种特殊机制。那么我们可以得知，类加载的最终产品是数据访问入口。



这个时候，看到这张图，我们应该有一个问题，那就是我们的字节码加载的方式，也就是我们的字节码文件可以用什么方式进行加载呢？

加载.class文件的方式

- 从本地系统中直接加载
 - 典型场景：这个我就不废话了
- 通过网络下载.class文件
 - 典型场景：Web Applet，也就是我们的小程序应用
- 从zip, jar等归档文件中加载.class文件
 - 典型场景：后续演变为jar、war格式
- 从专有数据库中提取.class文件
 - 典型场景：JSP应用从专有数据库中提取.class文件，较为少见
- 将Java源文件动态编译为.class文件，也就是运行时计算而成
 - 典型场景：动态代理技术
- 从加密文件中获取，
 - 典型场景：典型的防Class文件被反编译的保护措施

好，聊完了这个问题之后，问题接踵而至，我们的类加载的方式已经了解了，那么加载的流程到底是怎样的呢？

2.2.1 装载(Load)

查找和导入class文件

- (1) 通过一个类的全限定名获取定义此类的二进制字节流（由上可知，我们不一定从字节码文件中获得，还有上述很多种方式）

思考：那么这个时候我们是不是需要一个工具，寻找器，来寻找获取我们的二进制字节流。

而我们的java中恰好有这么一段代码模块。可以实现通过类全名来获取此类的二进制字节流这个动作，并且将这个动作放到java虚拟机外部去实现，以便让应用程序决定如何获取所需要的类，实现这个动作的代码模块成为“类加载器”。

- (2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- (3) 在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口

获取类的二进制字节流的阶段是我们JAVA程序员最关注的阶段，也是操控性最强的一个阶段。因为这个阶段我们可以对于我们的类加载器进行操作，比如我们想自定义类加载器进行操作用以完成加载，又或者我们想通过JAVA Agent来完成我们的字节码增强操作。

在我们的装载阶段完成之后，这个时候在我们的内存当中，我们的运行时数据区的方法区以及堆就已经有数据了。

- **方法区：**类信息，静态变量，常量
- **堆：**代表被加载类的java.lang.Class对象

即时编译之后的热点代码并不在这个阶段进入方法区

2.2.2 链接(Link)

2.2.2.1 验证(Verify)

验证只要是为了确保Class文件中的字节流包含的信息完全符合当前虚拟机的要求，并且还要求我们的信息不会危害虚拟机自身的安全，导致虚拟机的崩溃。

- **文件格式验证**

验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。这阶段的验证是基于二进制字节流进行的，只有经过该阶段的验证后，**字节流才会进入内存的方法区中进行存储，后面验证都是基于方法区的存储结构进行的。**

举例：

- 是否以16进制cafebabe开头
- 版本号是否正确
- **元数据验证**

对类的元数据信息进行语义校验（其实就是对Java语法校验），保证不存在不符合Java语法规范的元数据信息。

举例：

- 是否有父类
- 是否继承了final类
 - 因为我们的final类是不能被继承的，继承了就会出现问题。
- 一个非抽象类是否实现了所有的抽象方法
 - 如果没有实现，那么这个类也是无效的。

对类的元数据信息进行语义校验（其实就是对Java语法校验），保证不存在不符合Java语法规范的元数据信息。

- **字节码验证**

进行数据流和控制流分析，确定程序语义是合法的、符合逻辑的。对类的方法体进行校验分析，保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。

举例：

字节码的验证会相对来说较为复杂。

- 运行检查
- 栈数据类型和操作码操作参数吻合（比如栈空间只有4个字节，但是我们实际需要的远远大于4个字节，那么这个时候这个字节码就是有问题的）
- 跳转指令指向合理的位置
- 符号引用验证

这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候(解析阶段)，可以看作是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。符号引用验证的目的是确保解析动作能正常执行。

举例：

- 常量池中描述类是否存在
- 访问的方法或者字段是否存在且具有足够的权限

但是，我们很多情况下可能认为我们的代码肯定是没问题的，验证的过程完全没必要，那么其实我们可以添加参数

`-xverify:none` 取消验证。

2.2.2.2 准备(Prepare)

- 为类变量（静态变量）分配内存并且设置该类变量的默认初始值。

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

- 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，准备阶段会显式初始化；
- 这里不会为实例变量（也就是没加static）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。

```
public class Demo1 {  
    private static int i;  
  
    public static void main(String[] args) {  
        // 正常打印出0，因为静态变量i在准备阶段会有默认值0  
        System.out.println(i);  
    }  
}
```

```
public class Demo2 {  
    public static void main(String[] args) {  
        // 编译通不过，因为局部变量没有赋值不能被使用  
        int i;  
        System.out.println(i);  
    }  
}
```

进行分配内存的只是包括类变量(静态变量)，而不包括实例变量，实例变量是在对象实例化时随着对象一起分配在java堆中的。通常情况下，初始值为零值，假设public static int a=1;那么a在准备阶段过后的初始值为0，不为1，这时候只是开辟了内存空间，并没有运行java代码，a赋值为1的指令是程序被编译后，存放于类构造器()方法之中，所以a被赋值为1是在初始化阶段才会执行。

对于一些特殊情况，如果类字段属性表中存在ConstantValue属性，那在准备阶段变量a就会被初始化为ContstantValue属性所指的值。对于这句话，我们又怎么理解呢？

我们可以看一看我们反编译之后的文件，我们就可以发现有这样一个属性。

思考：ConstantValue属性到底是干什么的呢？

ConstantValue属性的作用是通知虚拟机自动为静态变量赋值，只有被static修饰的变量才可以使用这项属性。非static类型的变量的赋值是在实例构造器方法中进行的；static类型变量赋值分两种，在类构造其中赋值，或使用ConstantValue属性赋值。

思考：在实际的程序中，我们什么时候才会用到ContstantValue属性呢？

在实际的程序中，只有同时被final和static修饰的字段才有ConstantValue属性，且限于基本类型和String。编译时javac将会为该常量生成ConstantValue属性，在类加载的准备阶段虚拟机便会根据ConstantValue为常量设置相应的值，**如果该变量没有被final修饰，或者并非基本类型及字符串，则选择在类构造器中进行初始化。**

思考：为什么ConstantValue的属性值只限于基本类型和string？

因为从常量池中只能引用到基本类型和String类型的字面量

那么这个时候，我们来举个例子：

假设上面的类变量a被定义为：private static final int a = 1;

编译时javac将会为a生成ConstantValue属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为1。我们可以理解为static final常量在编译期就将其结果放入了调用它的类的常量池中

2.2.2.3 解析(Resolve)

把类中的符号引用转换为直接引用

符号引用就是一组符号来描述目标，可以是任何字面量。引用的目标并不一定已经加载到了内存中。
直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

解析阶段是虚拟机将常量池内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、方法类型、方法句柄和调用限定符7类符号引用进行。

直接引用是与虚拟机内存布局实现相关，同一个符号引用在不同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，那引用的目标必定存在内存中。**

对解析结果进行缓存

同一符号引用进行多次解析请求是很常见的，除invokedynamic指令以外，虚拟机实现可以对第一次解析结果进行缓存，来避免解析动作重复进行。无论是否真正执行了多次解析动作，虚拟机需要保证的是在同一个实体中，如果一个引用符号之前已经被成功解析过，那么后续的引用解析请求就应当一直成功；同样的，如果第一次解析失败，那么其他指令对这个符号的解析请求也应该收到相同的异常。

inDy (invokedynamic) 是 java 7 引入的一条新的虚拟机指令，这是自 1.0 以来第一次引入新的虚拟机指令。到了 java 8 这条指令才第一次在 java 应用，用在 lambda 表达式中。indy 与其他 invoke 指令不同的是它允许由应用级的代码来决定方法解析。这里不演示

2.2.3 初始化(Initialize)

初始化阶段是执行类构造器()方法的过程。

或者讲得通俗易懂些

在准备阶段，类变量已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制定的主观计划去初始化类变量和其他资源，比如赋值。

在Java中对类变量进行初始值设定有两种方式：

- 声明类变量是指定初始值
- 使用静态代码块为类变量指定初始值

按照程序员的逻辑，你必须把静态变量定义在静态代码块的前面。因为两个的执行是会根据代码编写的顺序来决定的，顺序搞错了可能会影响你的业务代码。

JVM初始化步骤：

- 假如这个类还没有被加载和连接，则程序先加载并连接该类
- 假如该类的直接父类还没有被初始化，则先初始化其直接父类
- 假如类中有初始化语句，则系统依次执行这些初始化语句

使用

那么这个时候我们去思考一个问题，我们的初始化过程什么时候会被触发执行呢？或者换句话说类初始化时机是什么呢？

主动引用

只有当对类的主动**使用**的时候才会导致类的初始化，类的主动使用有六种：

- 创建类的实例，也就是new的方式
- 访问某个类或接口的静态变量，或者对该静态变量赋值
- 调用类的静态方法
- 反射 (如 `Class.forName("com.carl.Test")`)
- 初始化某个类的子类，则其父类也会被初始化
- Java虚拟机启动时被标明为启动类的类 (`JvmCaseApplication`)，直接使用 `java.exe` 命令来运行某个主类

被动引用

- 引用父类的静态字段，只会引起父类的初始化，而不会引起子类的初始化。
- 定义类数组，不会引起类的初始化。
- 引用类的static final常量，不会引起类的初始化（如果只有static修饰，还是会引起该类初始化的）。

卸载

在类使用完之后，如果满足下面的情况，类就会被卸载：

- 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例。
- 加载该类的ClassLoader已经被回收。
- 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。

如果以上三个条件全部满足，jvm就会在方法区垃圾回收的时候对类进行卸载，类的卸载过程其实就是在方法区中清空类信息，java类的整个生命周期就结束了。但是一般情况下启动类加载器加载的类不会被卸载，而我们的其他两种基础类型的类加载器只有在极少数情况下才会被卸载。

类加载器 (ClassLoader)

什么是类加载器？

- 负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例的代码模块。
- 类加载器除了用于加载类外，还可用于确定类在Java虚拟机中的唯一性。

一个类在同一个类加载器中具有唯一性(Uniqueness)，而不同类加载器中是允许同名类存在的，这里的同名是指全限定名相同。但是在整个JVM里，纵然全限定名相同，若类加载器不同，则仍然不算作是同一个类，无法通过 instanceof、equals 等方式的校验。

1) Bootstrap ClassLoader

负责加载\$JAVA_HOME中 jre/lib/rt.jar 里所有的class或Xbootclasspath选项指定的jar包。由C++实现，不是ClassLoader子类。

2) Extension ClassLoader

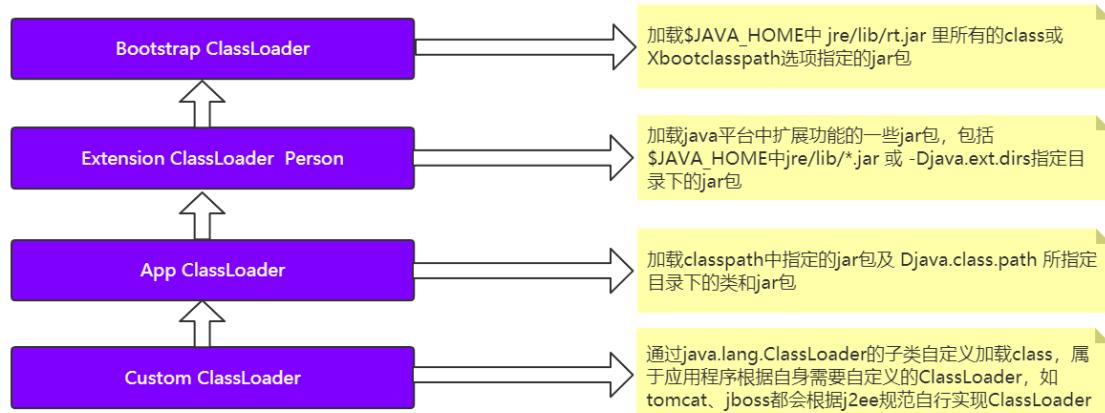
负责加载java平台中扩展功能的一些jar包，包括\$JAVA_HOME中jre/lib/*.jar 或 -Djava.ext.dirs指定目录下的jar包。

3) App ClassLoader

负责加载classpath中指定的jar包及 Djava.class.path 所指定目录下的类和jar包。

4) Custom ClassLoader

通过java.lang.ClassLoader的子类自定义加载class，属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader。



为什么我们的类加载器要分层？

1.2版本的JVM中，只有一个类加载器，就是现在的“Bootstrap”类加载器。也就是根类加载器。但是这样会出现一个问题。

假如用户调用他编写的java.lang.String类。理论上该类可以访问和改变java.lang包下其他类的默认访问修饰符的属性和方法的能力。也就是说，我们其他的类使用String时也会调用这个类，因为只有一个类加载器，我无法判定到底加载哪个。因为Java语言本身并没有阻止这种行为，所以会出现问题。

这个时候，我们就想到，可不可以使用不同级别的类加载器来对我们的信任级别做一个区分呢？

比如用三种基础的类加载器做为我们的三种不同的信任级别。最可信的级别是java核心API类。然后是安装的拓展类，最后才是在类路径中的类（属于你本机的类）。

所以，我们三种基础的类加载器由此而生。但是这是我们开发人员的视角。

```
public class Demo3 {  
    public static void main(String[] args) {  
        // App ClassLoader  
        System.out.println(new Worker().getClass().getClassLoader());  
        // Ext ClassLoader  
        System.out.println(new  
Worker().getClass().getClassLoader().getParent());  
        // Bootstrap ClassLoader  
        System.out.println(new  
Worker().getClass().getClassLoader().getParent().getParent());  
        System.out.println(new String().getClass().getClassLoader());  
    }  
}
```

```
sun.misc.Launcher$AppClassLoader@18b4aac2  
sun.misc.Launcher$ExtClassLoader@3a71f4dd  
null  
null
```

JVM类加载机制的三种方式

- **全盘负责**，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

例如，系统类加载器AppClassLoader加载入口类（含有main方法的类）时，会把main方法所依赖的类及引用的类也载入，依此类推。“全盘负责”机制也可称为当前类加载器负责机制。显然，入口类所依赖的类及引用的类的当前类加载器就是入口类的类加载器。

以上步骤只是调用了ClassLoader.loadClass(name)方法，并没有真正定义类。真正加载class字节码文件生成Class对象由“双亲委派”机制完成。

- **父类委托**，“双亲委派”是指子类加载器如果没有加载过该目标类，就先委托父类加载器加载该目标类，只有在父类加载器找不到字节码文件的情况下才从自己的类路径中查找并装载目标类。

父类委托别名就叫双亲委派机制。

“双亲委派”机制加载Class的具体过程是：

1. ClassLoader先判断该Class是否已加载，如果已加载，则返回Class对象；如果没有则委托给父类加载器。
2. 父类加载器判断是否加载过该Class，如果已加载，则返回Class对象；如果没有则委托给祖父类加载器。
3. 依此类推，直到始祖类加载器（引用类加载器）。
4. 始祖类加载器判断是否加载过该Class，如果已加载，则返回Class对象；如果没有则尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，则委托给始祖类加载器的子类加载器。
5. 始祖类加载器的子类加载器尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，则委托给始祖类加载器的孙类加载器。
6. 依此类推，直到源ClassLoader。
7. 源ClassLoader尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，源ClassLoader不会再委托其子类加载器，而是抛出异常。

“双亲委派”机制只是Java推荐的机制，并不是强制的机制。

我们可以继承java.lang.ClassLoader类，实现自己的类加载器。如果想保持双亲委派模型，就应该重写findClass(name)方法；如果想破坏双亲委派模型，可以重写loadClass(name)方法。

- **缓存机制**，缓存机制将会保证所有加载过的Class都将在内存中缓存，当程序中需要使用某个Class时，类加载器先从内存的缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效。**对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass方法不会被重复调用。**

而这里我们JDK8使用的是直接内存，所以我们会用到直接内存进行缓存。这也就是我们的类变量为什么只会被初始化一次的由来。

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, 在虚拟机内存中查找是否已经加载过此类...类缓存的主要问题所在！！！
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    //先让上一层加载器进行加载
                    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) {
                //调用此类加载器所实现的findClass方法进行加载
                c = findClass(name);
            }
        }
    }
}
```

```
    if (resolve) {
        //resolveClass方法是当字节码加载到内存后进行链接操作，对文件格式和字节码验证，并为 static 字段分配空间并初始化，符号引用转为直接引用，访问控制，方法覆盖等
        resolveClass(c);
    }
    return c;
}
```

打破双亲委派

双亲委派这个模型并不是强制模型，而且会带来一些些的问题。就比如java.sql.Driver这个东西。JDK只能提供一个规范接口，而不能提供实现。提供实现的是实际的数据库提供商。提供商的库总不能放JDK目录里吧。

所以java想到了几种办法可以用来打破我们的双亲委派。

SPI :比如java从1.6搞出了SPI就是为了优雅的解决这类问题——JDK提供接口，供应商提供服务。编程人员编码时面向接口编程，然后JDK能够自动找到合适的实现，岂不是很爽？

Java 在核心类库中定义了许多接口，并且还给出了针对这些接口的调用逻辑，然而并未给出实现。开发者要做的就是定制一个实现类，在 META-INF/services 中注册实现类信息，以供核心类库使用。比如JDBC中的DriverManager

OSGI:比如我们的JAVA程序员更加追求程序的动态性，比如代码热部署，代码热替换。也就是就是机器不用重启，只要部署上就能用。OSGI实现模块化热部署的关键则是它自定义的类加载器机制的实现。每一个程序模块都有一个自己的类加载器，当需要更换一个程序模块时，就把程序模块连同类加载器一起换掉以实现代码的热替换。

自定义类加载器

```
package com.example.jvmcase.loader;

import java.io.*;

public class MyClassLoader extends ClassLoader {

    private String root;

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classData = loadClassData(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
            return defineClass(name, classData, 0, classData.length);
        }
    }

    private byte[] loadClassData(String className) {
        String fileName = root +
            File.separatorChar + className.replace('.', File.separatorChar)
        + ".class";
        try {
            InputStream ins = new FileInputStream(fileName);
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();

    int bufferSize = 1024;

    byte[] buffer = new byte[bufferSize];

    int length = 0;
    while ((length = ins.read(buffer)) != -1) {
        baos.write(buffer, 0, length);
    }
    return baos.toByteArray();

} catch (IOException e) {
    e.printStackTrace();
}
return null;

}

public String getRoot() {
    return root;
}

public void setRoot(String root) {

    this.root = root;
}

public static void main(String[] args) {
    MyClassLoader classLoader = new MyClassLoader();
    classLoader.setRoot("E:\\temp");

    Class<?> testClass = null;
    try {
        testClass = classLoader.loadClass("com.neo.classloader.Test2");

        Object object = testClass.newInstance();

        System.out.println(object.getClass().getClassLoader());

    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

输出结果:

```
class Test
com.example.jvmcase.loader.MyClassLoader@27d6c5e0
```

自定义类加载器的核心在于对字节码文件的获取，如果是加密的字节码则需要在该类中对文件进行解密。由于这里只是演示，我并未对class文件进行加密，因此没有解密的过程。这里有几点需要注意：

- 1、这里传递的文件名需要是类的全限定性名称，即 `Test` 格式的，因为 `defineClass` 方法是按这种格式进行处理的。

如果没有全限定名，那么我们需要做的事情就是将类的全路径加载进去，而我们的 `setRoot` 就是前缀地址 `setRoot + loadClass` 的路径就是文件的绝对路径

- 2、最好不要重写 `loadClass` 方法，因为这样容易破坏双亲委托模式。
- 3、这类 `Test` 类本身可以被 `AppClassLoader` 类加载，因此我们不能把 `Test.class` 放在类路径下。否则，由于双亲委托机制的存在，会直接导致该类由 `AppClassLoader` 加载，而不会通过我们自定义类加载器来加载。

如果我们把 `Test` 放在类路径之下，那么我们将会通过 `AppClassLoader` 加载

打印结果:

```
class com.example.jvmcase.basic.Test
sun.misc.Launcher$AppClassLoader@18b4aac2
```

Tomcat现在基本8.0版本已经全面被8.5版本代替，而8.5版本源码有部分改动，不过我们还是可以看到，我们的LoadClass依然打破了双亲委派。

```
@Override
public Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        if (log.isDebugEnabled())
            log.debug("loadClass(" + name + ", " + resolve + ")");
        Class<?> clazz = null;

        // Log access to stopped class loader
        checkStateForClassLoader(name);

        // (0) Check our previously loaded local class cache
        clazz = findLoadedClass0(name);
        if (clazz != null) {
            if (log.isDebugEnabled())
                log.debug(" Returning class from cache");
            if (resolve)
                resolveClass(clazz);
            return (clazz);
        }

        // (0.1) Check our previously loaded class cache
        clazz = findLoadedClass(name);
        if (clazz != null) {
            if (log.isDebugEnabled())
                log.debug(" Returning class from cache");
            if (resolve)
```

```
        resolveClass(clazz);
        return (clazz);
    }
...中间省略一万字

    throw new ClassNotFoundException(name);
}
```