一、Class 文件结构

第一章　概述

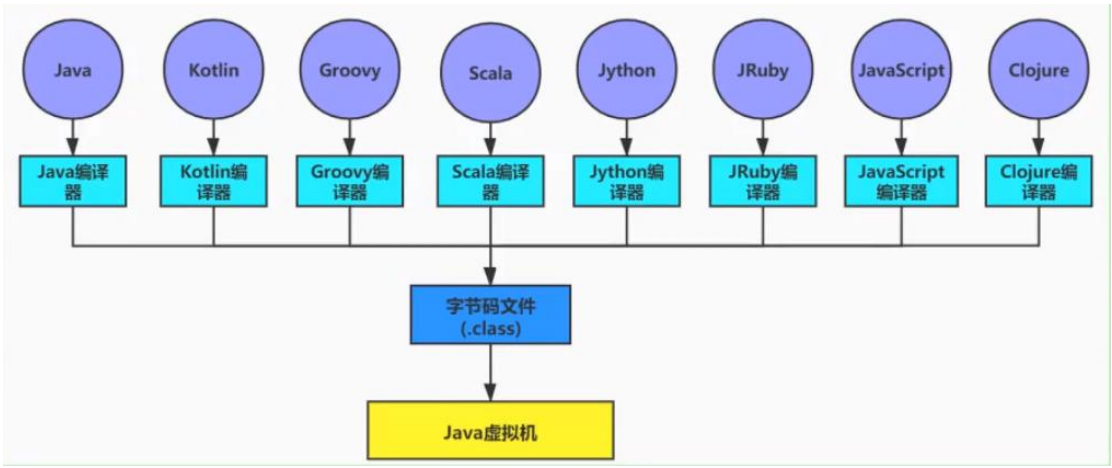
1．字节码文件的跨平台性

1. Java 语言，跨平台的(write once, run anywhere)

   - 当 Java 源代码成功编译成字节码后，如果想在不同的平台上面运行，则无须再次编译
   - 这个优势不再那么吸引人了。Python、PHP、Perl、Ruby、Lisp 等有强大的解释器
   - 跨平台似乎已经快称为一门语言必选的特性

2. Java 虚拟机：跨语言的平台

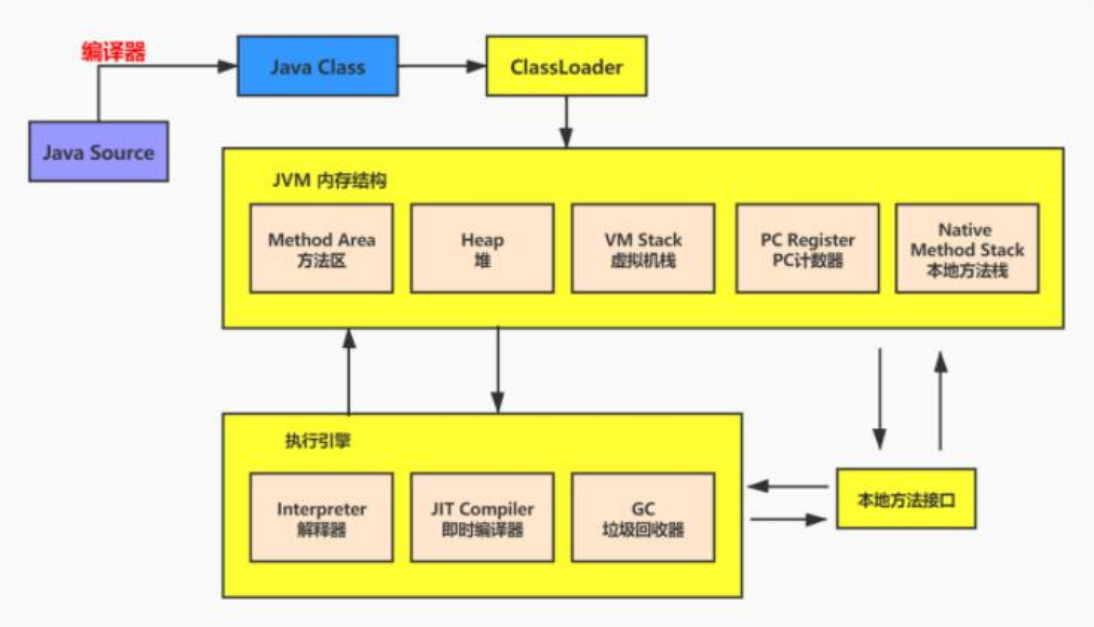
Java 虚拟机不和包括 Java 在内的任何语言绑定，它只与"Class 文件"这种特定的二进制文件格式所关联。无论使用何种语言进行软件开发， 只要能将源文件编译为正确的 Class 文件，那么这种语言就可以在 Java 虚拟机上执行，可以说，统一而强大的 Class 文件结构，就是 Java 虚拟机的基石、桥梁。



https://docs.oracle.com/javase/specs/index.html，所有的 JVM 全部遵守 Java 虚拟机规范，也就是说所有的 JVM 环境都是一样的， 这样一来字节码文件可以在各种 JVM 上进行。

3. 想要让一个 Java 程序正确地运行在 JVM 中，Java 源码就是必须要被编译为符合 JVM 规范的字节码

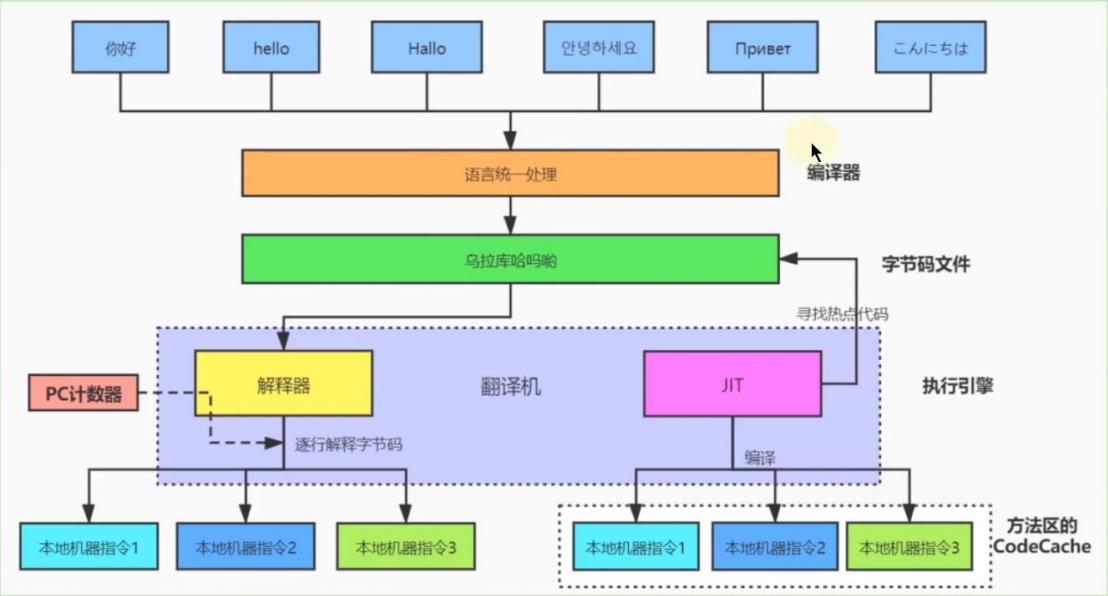
- 前端编译器的主要任务就是负责将符合 Java 语法规范的 Java 代码转换为符合 JVM 规范的字节码文件
- javac 是一种能够将 Java 源码编译为字节码的前端编译器
- javac 编译器在将 Java 源码编译为一个有效的字节码文件过程中经历了 4 个步骤，分别是词法分析、语法分析、语义分析以及生成字节码。



Oracle 的 JDK 软件包括两部分内容：

- 一部分是将 Java 源代码编译成 Java 虚拟机的指令集的编译器
- 另一部分是用于实现 Java 虚拟机的运行时环境

2．Java 的前端编译器



前端编译器 VS 后端编译器

Java 源代码的编译结果是字节码，那么肯定需要有一种编译器能够将 Java 源码编译为字节码，承担这个重要责任的就是配置在 path 环境变量中的 javac 编译器。javac 是一种能够将 Java 源码编译为字节码的前端编译器。

HotSpot VM 并没有强制要求前端编译器只能使用 javac 来编译字节码，其实只要编译结果符合 JVM 规范都可以被 JVM 所识别即可。在 Java 的前端编译器领域，除了 javac 之外，还有一种被大家经常用到的前端编译器，那就是内置在 Eclipse 中的 ECJ (Eclipse Compiler for Java)编译器。和 javac 的全量式编译不同，ECJ 是一种增量式编译器。

- 在 Eclipse 中，当开发人员编写完代码后，使用"Ctrl + S"快捷键时，ECJ 编译器所采取的编译方案是把未编译部分的源码逐行进行编译，而非每次都全量编译。因此 ECJ 的编译效率会比 javac 更加迅速和高效，当然编译质量和 javac 相比大致还是一样的

- ECJ 不仅是 Eclipse 的默认内置前端编译器，在 Tomcat 中同样也是使用 ECJ 编译器来编译 jsp 文件。由于 ECJ 编译器是采用 GPLv2 的开源协议进行源代码公开，所以，大家可以登录 Eclipse 官网下载 ECJ 编译器的源码进行二次开发
- 默认情况下，IntelliJ IDEA 使用 javac 编译器（还可以自己设置为 AspectJ 编译器 ajc）

前端编译器并不会直接涉及编译优化等方面的技术，而是将这些具体优化细节移交给 HotSpot 的 JIT 编译器负责

3．透过字节码指令看代码细节

- BAT 面试题
  - 类文件结构有几个部分？
  - 知道字节码吗？字节码都有哪些？Integer x = 5; int y = 5; 比较 x == y 都经过哪些步骤？
- 代码举例

```
public class IntegerTest {

    public static void main(String[] args) {

        Integer i1 = 10;

        Integer i2 = 10;

        System.out.println(i1 == i2);

        Integer i3 = 128;

        Integer i4 = 128;

        System.out.println(i3 == i4);

    }
```

```
}
```

```
public class StringTest {
    public static void main(String[] args) {
        String str = new String("hello") + new String("world");
        String str1 = "helloworld";
        System.out.println(str == str1);
    }
}
```

```
public class SonTest {
    public static void main(String[] args) {
        Father f = new Son();
        System.out.println(f.x);
    }
}
class Father {
    int x = 10；
    public Father() {
        this.print();
        x = 20;
    }
    public void print() {
        System.out.println("Father.x = " + x);
```

```
    }
}
class Son extends Father {
    int x = 30;
    public Son() {
        this.print();
        x = 40;
    }
    public void print() {
        System.out.println("Son.x = " + x);
    }
}
```

第二章　虚拟机的基石：Class 文件

字节码文件里是什么？

源代码经过编译器编译之后便会生成一个字节码文件，字节码是一种二进制的类文件，它的内容是 JVM 的指令，而不像 C、C++ 经由编译器直接生成机器码

什么事字节码指令(byte code)？

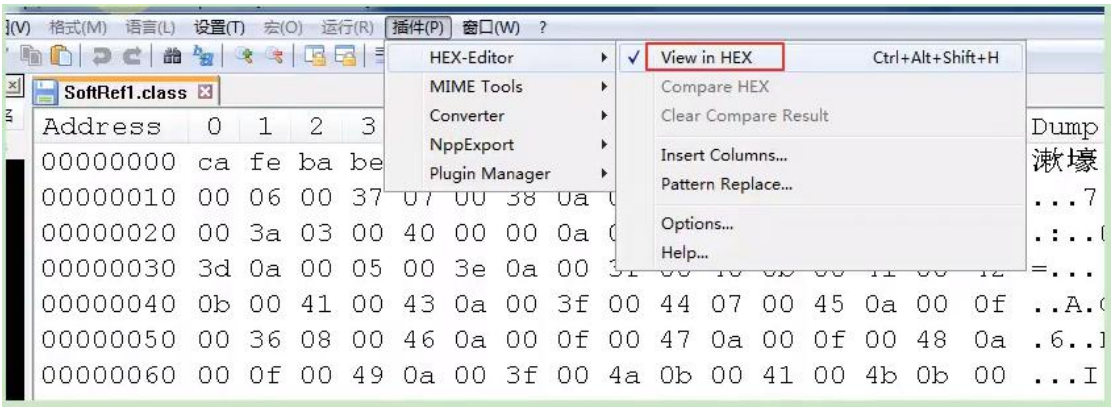
Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的**操作码(opcode)以及跟随其后的零至多个代表此操作所需参数的操作数(operand)**所构成。虚拟机中许多指令并不包含操作数，只有一个操作码

比如：

```
 0 aload_0
 1 invokespecial #1 <com/atguigu/java/Father.<init>>
 4 aload_0
 5 bipush 30
 7 putfield #2 <com/atguigu/java/Son.x>
10 aload_0
11 invokevirtual #3 <com/atguigu/java/Son.print>
14 aload_0
15 bipush 40
17 putfield #2 <com/atguigu/java/Son.x>
20 return
```

如何解读供虚拟机解释执行的二进制字节码？

方式一：一个一个二进制的看，这里用到的是 Notepad++，需要安装一个 HEX-Editor 插件，或者使用 Binary Viewer



方式二：使用 javap 指令，JDK 自带的反解析工具

方式三：使用 IDEA 插件，jclasslib 或 jclasslib bytecode viewer 客户端工具

第三章　Class 文件结构

- 官方文档位置

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

- Class 类的本质

任何一个 Class 文件都对应着唯一一个类或接口的定义信息，但反过来说，Class 文件实际上它并不一定以磁盘文件形式存在。Class 文件是一组以 8 位字节为基础单位的二进制流

- Class 文件格式

Class 的结构不像 XML 等描述语言，由于它没有任何分隔符号。所以在其中的数据项，无论是字节顺序还是数量，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变

Class 文件格式采用一种类似于 C 语言结构体的方式进行数据存储，这种结构中只有两种数据类型：无符号数和表

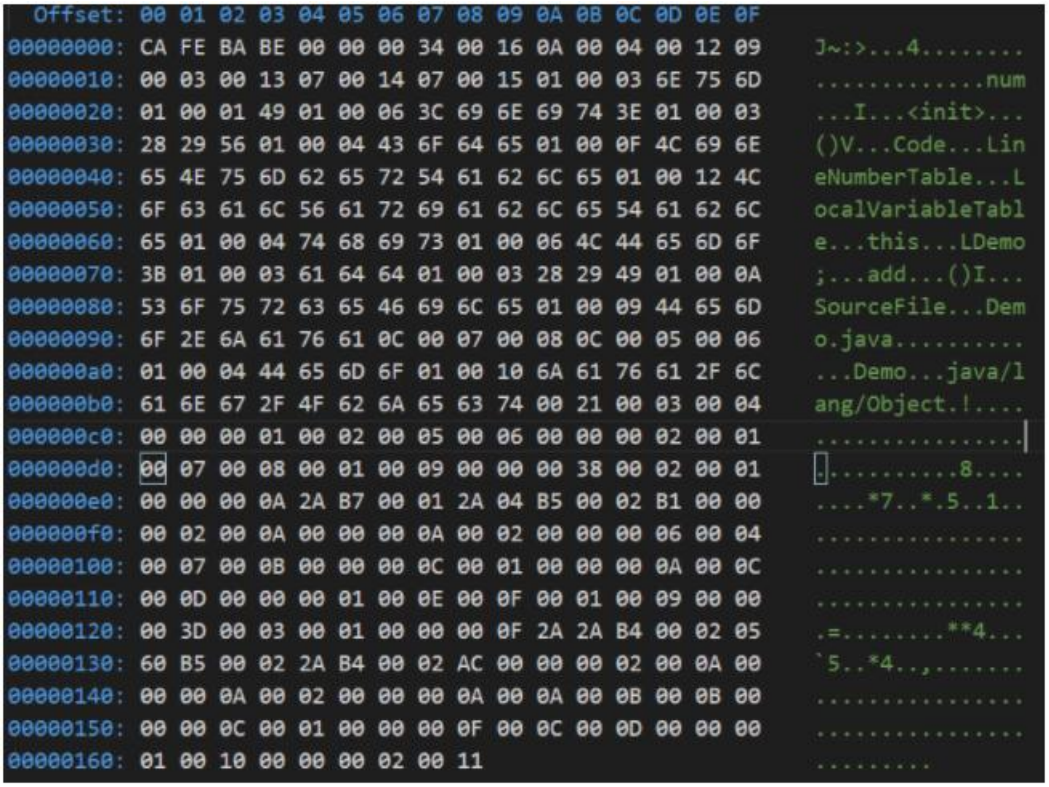
1) 无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节、8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值
2) 表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以"_info"结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表。由于表没有固定长度，所以通常会在其前面加上个数说明

代码举例

```
public class Demo {
    private int num = 1;
    public int add() {
```

```
        num = num + 2;

        return num;

    }

}
```

对应的字节码文件：



换句话说，充分理解了每一个字节码文件的细节，自己也可以反编译出 Java 源文件来

- Class 文件结构概述

Class 文件的结构并不是一成不变的，随着 Java 虚拟机的不断发展，总是不可避免地会对 Class 文件结构做出一些调整，但是其基本结构和框架是非常稳定的

Class 文件的总体结构如下：

1) 魔数

2) Class 文件版本

3) 常量池

4) 访问标志

5) 类索引、父类索引、接口索引集合

6) 字段表集合

7) 方法表集合

8) 属性表集合

类型	名称	说明	长度	数量
u4	magic	魔数,识别Class文件格式	4个字节	1
u2	minor_version	副版本号(小版本)	2个字节	1
u2	major_version	主版本号(大版本)	2个字节	1
u2	constant_pool_count	常量池计数器	2个字节	1
cp_info	constant_pool	常量池表	n个字节	constant_pool_count-1
u2	access_flags	访问标识	2个字节	1
u2	this_class	类索引	2个字节	1
u2	super_class	父类索引	2个字节	1
u2	interfaces_count	接口计数器	2个字节	1
u2	interfaces	接口索引集合	2个字节	interfaces_count
u2	fields_count	字段计数器	2个字节	1
field_info	fields	字段表	n个字节	fields_count
u2	methods_count	方法计数器	2个字节	1
method_info	methods	方法表	n个字节	methods_count
u2	attributes_count	属性计数器	2个字节	1
attribute_info	attributes	属性表	n个字节	attributes_count

1. 魔数：Class 文件的标志

Magic Number(魔数)

- 每个 Class 文件开头的 4 个字节的无符号整数称为魔数(Magic Number)
- 它的唯一作用是确定这个文件是否为一个能被虚拟机接受的有效合法的 Class 文件。即：魔数是 Class 文件的标识符
- 魔数值固定为 0xCAFEBABE。不会改变
- 如果一个 Class 文件不以 0xCAFEBABE 开头，虚拟机在进行文件校验的时候就会直接抛出以下错误：

```
Error: A JNI error has occurred, please check your installation and try again

Exception in thread "main" java.lang.ClassFormatError: Incompatible magic value 1885430635 in class file StringTest
```

2．Class 文件版本号

- 紧接着魔数的 4 个字节存储的是 Class 文件的版本号。同样也是 4 个字节。第 5 个和第 6 个字节所代表的含义就是编译的副版本号 minor_version，而第 7 个和第 8 个字节就是编译的主版本号 major_version
- 它们共同构成了 Class 文件的格式版本号。譬如某个 Class 文件的主版本号为 M，服版本号为 m，那么这个 Class 文件的格式版本号就确定为 M.m
- 版本号和 Java 编译器的对应关系如下表：

主版本（十进制）	副版本（十进制）	编译器版本
45	3	1.1
46	0	1.2
47	0	1.3
48	0	1.4
49	0	1.5
50	0	1.6
51	0	1.7
52	0	1.8
53	0	1.9
54	0	1.10
55	0	1.11

- Java 的版本号是从 45 开始的，JDK 1.1 之后的每个 JDK 大版本发布主版本号向上加 1
- 不同版本的 Java 编译器编译的 Class 文件对应的版本是不一样的。目前，高版本的 Java 虚拟机可以执行由低版本编译器生成的 Class 文件，但是低版本的 Java 虚拟机不能执行由高版本编译器生成的 Class 文件。否则 JVM 会抛出 java.lang.UnsupportedClassVersionError 异常(向下兼容)
- 在实际应用中，由于开发环境和生产环境的不同，可能会导致该问题的发生。因此，需要我们在开发时，特别注意开发编译的 JDK 版本和生产环境的 JDK 版本是否一致
  - 虚拟机 JDK 版本为 1.k (k >= 2)时，对应的 Class 文件格式版本号的范围为 45.0 - 44 + k.0(含两端)

3．常量池：存放所有常量

- 常量池是 Class 文件中内容最为丰富的区域之一。常量池对于 Class 文件中的字段和方法解析也有着至关重要的作用
- 随着 Java 虚拟机的不断发展，常量池的内容也日渐丰富，可以说，常量池

是整个 Class 文件的基石

- 在版本号之后，紧跟着的是常量池的数量，以及若干个常量池表项
- 常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 u2 类型的无符号数，代表常量池容量计数值(constant_pool_count)，与 Java 中语言习惯不一样的是，这个容量计数是从 1 而不是 0 开始的

类型	名称	数量
u2（无符号数）	constant_pool_count	1
cp_info（表）	constant_pool	constant_pool_count-1

由上表可见，Class 文件使用了一个前置的容量计数器(constant_pool_count)加若干个连续的数据项(constant_pool)的形式来描述常量池内容，我们把这一系列连续常量池数据称为常量池集合

- 常量池表项中，用于存放编译时期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放

(1) 常量池计数器

- 由于常量池的数量不固定，时长时短，所以需要放置两个字节来表示常量池容量计数值
- 常量池容量计数值(u2 类型)：从 1 开始，表示常量池中有多少项常量。即 constant_pool_count = 1 表示常量池中有 0 个常量项
- Demo 的值为：

```
Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  Dump
00000000 ca fe ba be 00 00 00 34 00 16 0a 00 04 00 12 09 漱壕..
```

其值为 0x0016，对应的十进制值为 22

需要注意的是，这实际上只有 21 项常量。索引为范围是 1-21。为什么呢？

通常我们写代码时都是从 0 开始的，但是这里的常量池却是从 1 开始，因为它把第 0 项常量空出来了。这是为了满足后面某些指向常量池的索引值的数据在特定情况下需要表达"不引用任何一个常量池项目"的含义，这种情况可用索引值 0 来表示

(2) 常量池表

- constant_pool 是一种表结构，以 1 ~ constant_pool_count - 1 为索引。表明了后面有多少个常量项
- 常量池主要存放两大类常量：字面量(Literal)和符号引用(Symbolic References)
- 它包含了 Class 文件结构及其子结构中引用的所有字符串常量、类或接口名、字段名和其他常量。常量池中的每一项都具备相同的特征。第 1 个字节作为类型标记，用于确定该项的格式，这个字节称为 tag byte(标记字节、标签字节)

类型	标志(或标识)	描述
CONSTANT_utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量

CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标志方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

① 字面量和符号引用

在对这些常量解读前，需要搞清楚几个概念

常量池主要存放两大类常量：字面量(Literal)和符号引用(Symbolic References)。如下表：

常量	具体的常量
字面量	文本字符串
	声明为final的常量值
符号引用	类和接口的全限定名
	字段的名称和描述符
	方法的名称和描述符

全限定名

com/atguigu/test/Demo 这个就是类的全限定名，仅仅是把包的"."替换成"/"，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会加入一个";"表示全限定名结束

简单名称

简单名称是指没有类型和参数修饰的方法或者字段名称，上面例子中的类的add() 方法和 num 字段的简单名称分别是 add 和 num

描述符

描述符的作用是用来描述字段的数据类型、方法的参数列表(包括数量、类型以及顺序)和返回值。根据描述符规则，基本数据类型(byte、char、double、float、int、long、short、boolean)以及代表无返回值的 void 类型都用一个大写字符来表示，而对象类型则用字符 L 加对象的全限定名表示，详见下表：

标志符	含义
B	基本数据类型byte
C	基本数据类型char
D	基本数据类型double
F	基本数据类型float
I	基本数据类型int
J	基本数据类型long
S	基本数据类型short
Z	基本数据类型boolean
V	代表void类型
L	对象类型，比如：`Ljava/lang/Object;`
[	数组类型，代表一维数组。比如：`double[][][] is [[[D`

用描述符来藐视方法时，按照先参数列表，后返回值的顺序描述，参数列表按照参数的严格顺序放在一组小括号"()"之内，如方法 java.lang.String toString()的描述符为 () Ljava/lang/String;，方法 int abc(int[] x ,int y)描述符为([II) I

虚拟机在加载 Class 文件时才会进行动态链接，也就是说，Class 文件中不会保存各个方法和字段的最终内存布局信息，因此，这些字段和方法的符号引用不经过转换是无法直接被虚拟机使用的。当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类加载过程中的解析阶段将其替换为直接引用，并翻译到具体的内存地址中。

这里说明下符号引用和直接引用的区别与关联：

- 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，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中
- 直接引用：直接引用可以是直接指**向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，**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，那说明引用的目标必定已经存在于内存之中了。

② 常量类型和结构

常量池中每一项常量都是一个表，JDK 1.7 之后共 14 种不同的表结构数据。如下表格所示：

标志	常量	描述	细节	长度	细节描述
1	CONSTANT_utf8_info	UTF-8编码的字符串	tag	u1	值为1
			length	u2	UTF-8编码的字符串占用的字符数
			bytes	u1	长度为length的UTF-8编码的字符串
3	CONSTANT_Integer_info	整型字面量	tag	u1	值为3
			bytes	u4	按照高位在前存储的int值
4	CONSTANT_Float_info	浮点型字面量	tag	u1	值为4
			bytes	u4	按照高位在前存储的float值
5	CONSTANT_Long_info	长整型字面量	tag	u1	值为5
			bytes	u8	按照高位在前存储的long值
6	CONSTANT_Double_info	双精度浮点型字面量	tag	u1	值为6
			bytes	u8	按照高位在前存储的double值
7	CONSTANT_Class_info	类或接口的符号引用	tag	u1	值为7
			index	u2	指向全限定名常量项的索引
8	CONSTANT_String_info	字符串类型字面量	tag	u1	值为8
			index	u2	指向字符串字面量的索引
9	CONSTANT_Fieldref_info	字段的符号引用	tag	u1	值为9
			index	u2	指向声明字段的类或接口描述符CONSTANT_Class_info的索引项
			index	u2	指向字段描述符CONSTANT_NameAndType的索引项
10	CONSTANT_Methodref_info	类中方法的符号引用	tag	u1	值为10
			index	u2	指向声明方法的类描述符CONSTANT_Class_Info的索引项
			index	u2	指向名称及类型描述符CONSTANT_NameAndType的索引项

11	CONSTANT_InterfaceMethodref_info	接口中方法的符号引用	tag	u1	值为11
			index	u2	指向声明方法的接口描述符CONSTANT_Class_Info的索引项
			index	u2	指向名称及类型描述符CONSTANT_NameAndType的索引项
12	CONSTANT_NameAndType_info	字段或方法的符号引用	tag	u1	值为12
			index	u2	指向该字段或方法名称常量项的索引
			index	u2	指向该字段或方法描述符常量项的索引
15	CONSTANT_MethodHandle_info	表示方法句柄	tag	u1	值为15
			reference_kind	u1	值必须在1-9之间，它决定了方法句柄的类型方法句柄类型的值表示方法句柄的字节码行为
			reference_index	u2	值必须是对常量池的有效索引
16	CONSTANT_MethodType_info	标志方法类型	tag	u1	值为16
			descriptor_index	u2	值必须是对常量池的有效索引，常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示方法的描述符
18	CONSTANT_InvokeDynamic_info	表示一个动态方法调用点	tag	u1	值为18
			bootstrap_method_attr	u2	值必须是对当前Class文件中引导方法表的bootstrap_methods[]数组的有效索引
			name_and_type_index	u2	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是CONSTANT_NameAndType_Info结构，表示方法名和方法描述符

- 根据上图每个类型的描述我们也可以知道每个类型是用来描述常量池中那些内容(主要是字面量、符号引用)的。比如： CONSTANT_Integer_info 是用来描述常量池中字面量信息的，而且只是整型字面量信息
- 标志为 15、16、18 的常量项类型是用来支持动态语言调用的(JDK 1.7 时才加入)
- 细节说明
  - CONSTANT_Class_info 结构用于表示类或接口
  - CONSTANT_Fieldref_info 、 CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 结构表示字段、方法和接口方法
  - CONSTANT_String_info 结构用于表示 String 类型的常量对象
  - CONSTANT_Integer_info 和 CONSTANT_Float_info 表示 4 字节(int 和 float)的数值常量
  - CONSTANT_Long_info 和 CONSTANT_Double_info 结构表示 8 字节(long 和 double)的数值常量
    - ✓ 在 Class 文件的常量池中，所有的 8 字节常量均占两个表成员(项)的空间，如果一个 CONSTANT_Long_info 或 CONSTANT_Double_info 结构的项在常量池表中的索引位 n，则常量池表中下一个可用项的索引位 n + 2，此时常量池表中索引为 n + 1，的项仍然有效但必须视为不可用的
  - CONSTANT_NameAndType_info 结构用于表示字段或方法，但是和之前的 3 个结构不同，CONSTANT_NameAndType_info 结构没有知名该字段或方法所属的类或接口
  - CONSTANT_Utf8_info 用于表示字符常量的值
  - CONSTANT_MethodHandle_info 结构用于表示方法句柄
  - CONSTANT_MethodType_info 结构表示方法类型
  - CONSTANT_InvokeDynamic_info 结构用于表示 invokedynamic 指令所

用到的引导方法(bootstrap method)、引导方法所用到的动态调用名称(dynamic invocation name)、参数和返回类型，并可以给引导方法传入一系列称为静态参数(static argument)的常量

- 解析方式
  - 一个字节一个字节的解析
  - 使用 javap 命令解析：javap -verbose Demo.class 或 jclasslib 工具会更方便

总结：

- 这 14 种表(或者常量项结构)的共同点是：表开始的第一位是一个 u1 类型的标志位(tag)，代表当前这个常量项使用的是哪种表结构，即哪种常量类型
- 在常量池列表中，CONSTANT_Utf8_info 常量项是一种使用改进过的 UTF-8 编码格式来存储诸如文字字符串、类或者接口的全限定名、字段或者方法的简单名称以及描述符等常量字符串信息
- 这 14 种常量项结构还有一个特点是，其中 13 个常量项占用的字节固定，只有 CONSTANT_Utf8_info 占用字节不固定，其大小由 length 决定。为什么？**因为从常量池存放的内容可知，其存放的是字面量和符号引用，最终这些内容都会是一个字符串，这些字符串的大小是在编写程序时才确定，**比如你定义一个类，类名可以取长去短，所以在没编译前，大小不固定，编译后，通过 UTF-8 编码，就可以知道其长度
- 常量池：可以理解为 Class 文件之中的资源仓库，它是 Class 文件结构中与其他项目关联最多的数据类型(后面的很多数据类型都会指向此处)，也是占用 Class 文件空间最大的数据项目之一
- 常量池中为什么包含这些内容

Java 代码在进行 javac 编译的时候，并不像 C 和 C++ 那样有"连接"这一步骤，而是在虚拟机加载 Class 文件的时候进行动态链接。也就是说，**在 Class 文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符

号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。**当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。

4. 访问标识

- 在常量池后，紧跟着访问标记。该标记使用两个字节表示，用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口；是否定义为 public 类型；是否定义为 abstract 类型；如果是类的话，是否被声明为 final 等。各种访问标记如下所示：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	标志为public类型
ACC_FINAL	0x0010	标志被声明为final，只有类可以设置
ACC_SUPER	0x0020	标志允许使用invokespecial字节码指令的新语义，JDK1.0.2之后编译出来的类的这个标志默认为真。（使用增强的方法调用父类方法）
ACC_INTERFACE	0x0200	标志这是一个接口
ACC_ABSTRACT	0x0400	是否为abstract类型，对于接口或者抽象类来说，次标志值为真，其他类型为假
ACC_SYNTHETIC	0x1000	标志此类并非由用户代码产生（即：由编译器产生的类，没有源码对应）
ACC_ANNOTATION	0x2000	标志这是一个注解
ACC_ENUM	0x4000	标志这是一个枚举

- 类的访问权限通常为 ACC_ 开头的常量
- 每一个种类型的表示都是通过设置访问标记的 32 位中的特定位来实现的。比如，若是 public final 的类，则该标记为 ACC_PUBLIC | ACC_FINAL
- 使用 ACC_SUPER 可以让类更准确地定位到父类的方法 super.method()，现代编译器都会设置并且使用这个标记

补存说明

1) 带有 ACC_INTERFACE 标志的 Class 文件表示的是接口而不是类，反之则表示的是类而不是接口

    1. 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 或 ACC_ENUM 标志

    2. 如果没有设置 ACC_INTERFACE 标志，那么这个 Class 问价可以具有上表中除 ACC_ANNOTATION 外的其他所有标志。当然，ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标志除外。这两个标志不能同时设置

2) ACC_SUPER 标志用于确定类或接口里面的 invokespecial 指令使用的是哪一种执行语义。**针对 Java 虚拟机指令集的编译器都应当设置这个标志。**对于 Java SE 8 及后续版本来说，无论 Class 文件中这个标志的实际值是什么，也不管 Class 文件的版本

3) ACC_SYNTHETIC 标志意味着该类或接口是由编译器生成的，而不是由源代码生成的

4) 注解类型必须设置 ACC_ANNOTATION 标志。如果设置了 ACC_ANNOTATION 标志，那么也必须设置 ACC_INTERFACE 标志

5) ACC_ENUM 标志标明该类或其父类为枚举类型

6) 表中没有使用的 access_flags 标志是为未来扩充而预留的，这些预留的标志在编译器中应该设置为 0，Java 虚拟机实现也应该忽略他们

5. 类索引、父类索引、接口索引集合

在访问标记后，会指定该类的类别、父类类别以及实现的接口，格式如下：

长度	含义
u2	this_class
u2	super_class
u2	interfaces_count
u2	interfaces[interfaces_count]

- 这三项数据来确定这个类的继承关系
  - 类索引用于确定这个类的全限定名
  - 父类索引用于确定这个类的父类的全限定名。由于 Java 语言不允许多重继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 Java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0
  - 接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 implements 语句(如果这个类本身是一个接口，则应当是 extends 语句)后的接口顺序从左到右排列在接口索引集合中

1. this_class(类索引)

2 字节无符号整数，指向常量池的索引。它提供了类的全限定名，如 com/atguigu/java1/Demo。this_class 的值必须是对常量池表中某项的一个有效索引值。常量池在这个索引处的成员必须为 CONSTANT_Class_info 类型结构体，该结构体表示这个 Class 文件所定义的类或接口

2. super_class(父类索引)

- 2 字节无符号整数，指向常量池的索引。它提供了当前类的父类的全限定名。如果我们没有继承任何类，其默认继承的是 java/lang/Object 类。同时，由于 Java 不支持多继承，所以其父类只有一个
- superclass 指向的父类不能是 final

3. interfaces

- 指向常量池索引集合，它提供了一个符号引用到所有已实现的接口
- 由于一个类可以实现多个接口，因此需要以数组形式保存多个接口的索引，表示接口的每个索引也是一个指向常量池的 CONSTANT_Class(当然这里就必须是接口，而不是类)

3.1 interfaces_count(接口计数器)

interfaces_count 项的值表示当前类或接口的直接超接口数量

3.2 interface[] (接口索引集合)

interfaces[] 中每个成员的值必须是对常量池表中某项的有效索引值，它的长度为 interfaces_count。每个成员 interfaces[i] 必须为 CONSTANT_Class_info 结构，其中 0 <= i < interfaces_count。在 interfaces[] 中，各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

6．字段表集合

- 用于描述接口或类中声明的变量。字段(field)包括类级变量以及实例级变量，但是不包括方法内部、代码块内部声明的局部变量
- 字段叫什么名字、字段被定义为什么数据类型，这些都是无法固定的，只能引用常量池中的常量来描述
- 它指向常量池索引集合，它描述了每个字段的完整信息。比如**字段的标识符、访问修饰符(public、private 或 protected)、是类变量还是实例变量(static 修饰符)、是否是常量(final 修饰符)**等。

注意事项：

- 字段表集合中不会列出从父类或者实现的接口中继承而来的字段，但有可能列出原 Java 代码中不存在的字段，譬如在内部类中为了保持对外部类的方文星，会自动添加指向外部类实例的字段

- 在 Java 语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来讲，如果两个字段的描述符不一致，那字段重名就是合法的

（1）字段计数器

fields_count 的值表示当前 Class 文件 fields 表的成员个数。使用两个字节来表示

fields 表中每个成员都是一个 field_info 结构，用于表示该类或接口所声明的所有类字段或者实例字段，不包括方法内部声明的变量，也不包括从父类或父接口继承的那些字段

（2）字段表

fields 表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

一个字段的信息包括如下这些信息，这些信息中，各个修饰符都是布尔值，要么有，要么没有

- 作用域(public、private、protected 修饰符)
- 是实例变量还是类变量(static 修饰符)
- 可变性(final)
- 并发可见性(volatile 修饰符，是否强制从主内存读写)
- 可否序列化(transient 修饰符)
- 字段数据类型(基本数据类型、对象、数组)
- 字段名称

字段表结构：

类型	名称	含义	数量
u2	access_flags	访问标志	1
u2	name_index	字段名索引	1
u2	descriptor_index	描述符索引	1
u2	attributes_count	属性计数器	1
attribute_info	attributes	属性集合	attributes_count

字段表访问标识

我们知道，一个字段可以被各种关键字去修饰，比如：作用域修饰符(public、private、protected)、static 修饰符、final 修饰符、volatile 修饰符等等。因此，其可像类的访问标志那样，使用一些标志来标记字段。字段的访问标志有如下这些：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为public
ACC_PRIVATE	0x0002	字段是否为private
ACC_PROTECTED	0x0004	字段是否为protected
ACC_STATIC	0x0008	字段是否为static
ACC_FINAL	0x0010	字段是否为final
ACC_VOLATILE	0x0040	字段是否为volatile
ACC_TRANSTENT	0x0080	字段是否为transient
ACC_SYNCHETIC	0x1000	字段是否为由编译器自动产生
ACC_ENUM	0x4000	字段是否为enum

字段名索引

根据字段名索引的值，查询常量池中的指定索引项即可

描述符索引

描述符的作用是用来描述字段的数据类型、方法的参数列表(包括数量、类型以及顺序)和返回值。根据描述符规则，基本数据类型(byte、char、double、float、int、long、short、boolean)及代表无返回值的 void 类型都用一个大写字符来表示，而对象则用字符 L 加对象的全限定名来表示，如下所示：

字符	类型	含义
B	byte	有符号字节型数
C	char	Unicode 字符， UTF-16 编码
D	double	双精度浮点数
F	float	单精度浮点数
I	int	整型数
J	long	长整数
S	short	有符号短整数
Z	boolean	布尔值 true/false
L Classname;	reference	一个名为Classname的实例
[	reference	一个一维数组

属性表集合

一个字段还可能拥有一些属性，用于存储更多的额外信息。比如初始化值、一些注释信息等。属性个数存放在 attribute_count 中，属性具体内容存放在 attributes 数组中

以常量属性为例，结构为：

```
ConstantValue_attribute {
```

```
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

说明：对于常量属性而言，attribute_length 值恒为 2

7．方法表集合

methods: 指向常量池索引集合，它完整描述了每个方法的签名

- 在字节码文件中，每一个 method_info 项都对应着一个类或者接口中的方法信息。比如方法的访问修饰符(public、private 或 protected)，方法的返回值类型以及方法的参数信息等
- 如果这个方法不是抽象的或者不是 native 的，那么字节码中会体现出来
- 一方面，methods 表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。另一方面，methods 表有可能会出现由编译器自动添加的方法，最典型的便是编译器产生的方法信息(比如：类(接口)初始化方法 () 和实例初始化方法 ()

  使用注意事项：

在 Java 语言中，要重载(Overload)一个方法，除了要与原方法具有相同的简单名称之外，还要求必须拥有一个与原方法不同的特征签名，特征签名就是一个方法中各个参数在常量池中的字段符号引用的集合，也就是因为返回值不会包含在特征签名之中，因此 Java 语言里无法仅仅依靠返回值的不同来对一个已有方法进行重载。但在 Class 文件格式中，特征签名的范围更大一些，只要描述符不是完全一致的两个方法就可以共存。也就是说，如果两个方法有相同的名称和特征签名，但返回值不同，那么也是可以合法共存于同一个 Class 文件中。

也就是说，尽管 Java 语法规范并不允许在一个类或者接口中声明多个方法签名相同的方法，但是和 Java 语法规范相反，字节码文件中却恰恰允许存放多个方法签名相同的方法，唯一的条件就是这些方法之间的返回值不能相同。

(1) 方法计数器

methods_count 的值表示当前 Class 文件 methods 表的成员个数，使用两个字节来表示

methods 表中每个成员都是一个 method_info 结构

(2) 方法表

- methods 表中的每个成员都必须是一个 method_info 结构，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，那么该结构中也应包含实现这个方法所有的 Java 虚拟机指令
- 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
- 方法表的结构实际跟字段表是一样的，方法表结构如下：

类型	名称	含义	数量
u2	access_flags	访问标志	1
u2	name_index	方法名索引	1
u2	descriptor_index	描述符索引	1
u2	attributes_count	属性计数器	1
attribute_info	attributes	属性集合	attributes_count

方法表访问标志

跟字段表一样，方法表也有访问标志，而且他们的标志有部分相同，部分则

标记名	值	说明
ACC_PUBLIC	0x0001	public，方法可以从包外访问
ACC_PRIVATE	0x0002	private，方法只能本类中访问
ACC_PROTECTED	0x0004	protected，方法在自身和子类可以访问
ACC_STATIC	0x0008	static，静态方法

不同，方法表的具体访问标志如下：

8．属性表结合

方法表集合之后的属性表集合，指的是 Class 文件所携带的辅助信息，比如该 Class 文件的源文件的名称。以及任何带有 RetentionPolicy.CLASS 或者 RetentionPolicy.RUNTIME 的注解。这类信息通常被用于 Java 虚拟机的验证和运行，以及 Java 程序的调试，一般无需深入了解

此外，字段表、方法表都可以有自己的属性表。用于描述某些场景专有的信息

属性表集合的限制没有那么严格，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，但 Java 虚拟机运行时会忽略掉它不认识的属性

1. 属性计数器

attributes_count 的值表示当前 Class 文件属性表的成员个数。属性表中每一项都是一个 attribute_info 结构

2. 属性表

属性表的每个项的值必须是 attribute_info 结构。属性表的结构比较灵活，各种不同的属性只要满足以下结构即可

属性的通用格式

类型	名称	数量	含义
u2	attribute_name_index	1	属性名索引
u4	attribute_length	1	属性长度
u1	info	attribute_length	属性表

即只需说明属性的名称以及占用位数的长度即可，属性表具体的结构可以去自定义

属性类型

属性表实际上可以有很多类型，上面看到的 Code 属性只是其中一种， Java 8 里面定义了 23 种属性

下面这些是虚拟机中预定义的属性：

属性名称	使用位置	含义
Code	方法表	Java代码编译成的字节码指令
ConstantValue	字段表	final关键字定义的常量池
Deprecated	类，方法，字段表	被声明为deprecated的方法和字段
Exceptions	方法表	方法抛出的异常
EnclosingMethod	类文件	仅当一个类为局部类或者匿名类是才能拥有这个属性，这个属性用于标识这个类所在的外围方法
InnerClass	类文件	内部类列表
LineNumberTable	Code属性	Java源码的行号与字节码指令的对应关系
LocalVariableTable	Code属性	方法的局部便狼描述
StackMapTable	Code属性	JDK1.6中新增的属性，供新的类型检查检验器检查和处理目标方法的局部变量和操作数有所需要的类是否匹配
Signature	类，方法表，字段表	用于支持泛型情况下的方法签名
SourceFile	类文件	记录源文件名称
SourceDebugExtension	类文件	用于存储额外的调试信息
Synthetic	类，方法表，字段表	标志方法或字段为编译器自动生成的
LocalVariableTypeTable	类	使用特征签名代替描述符，是为了引入泛型语法之后能描述泛型参数化类型而添加
RuntimeVisibleAnnotations	类，方法表，字段表	为动态注解提供支持
RuntimeInvisibleAnnotations	表，方法表，字段表	用于指明哪些注解是运行时不可见的

RuntimeVisibleParameterAnnotation	方法表	作用与RuntimeVisibleAnnotations属性类似，只不过作用对象为方法
RuntimeInvisibleParameterAnnotation	方法表	作用与RuntimeInvisibleAnnotations属性类似，作用对象哪个为方法参数
AnnotationDefault	方法表	用于记录注解类元素的默认值
BootstrapMethods	类文件	用于保存invokeddynamic指令引用的引导方式限定符

部分属性详解

1．ConstantValue 属性

ConstantValue 属性表示一个常量字段的值。位于 field_info 结构的属性表中

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index; //字段值在常量池中的索引，常量池在该索引处的项给出该属性表示的常量值。(例如，值是 long 型的，在常量池中便是CONSTANT_Long)
}
```

2．Deprecated 属性

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

3．Code 属性

Code 属性就是存放方法体里面的代码，但是，并非所有方法表都有 Code 属性，像接口或者抽象方法，他们没有具体的方法体，因此也就不会有 Code 属性了

Code 属性表的结构，如下：

类型	名称	数量	含义
u2	attribute_name_index	1	属性名索引
u4	attribute_length	1	属性长度
u2	max_stack	1	操作数栈深度的最大值
u2	max_locals	1	局部变量表所需的存储空间
u4	code_length	1	字节码指令的长度
u1	code	code_length	存储字节码指令
u2	exception_table_length	1	异常表长度
exception_info	exception_table	exception_length	异常表
u2	attributes_count	1	属性集合计数器
attribute_info	attributes	attributes_count	属性集合

可以看到：Code 属性表的前两项跟属性表是一致的，即 Code 属性表遵循属性表的结构，后面那些则是他自定义的结构

4．InnerClasses 属性

为了方便说明特别定义一个表示类或接口的 Class 格式为 C。如果 C 的常量池中包含某个 CONSTANT_Class_info 成员，且这个成员所表示的类或接口不属于任何一个包，那么 C 的 ClassFile 结构的属性表中就必须含有对应的 InnerClasses 属性。InnerClasses 属性是在 JDK 1.1 中为了支持内部类和内部接口而引入的，位于 ClassFile 结构的属性表

5．LineNumberTable 属性

LineNumberTable 属性是可选变长属性，位于 Code 结构的属性表

LineNumberTable 属性是用来描述 Java 源码行号与字节码行号之间的对应关系，这个属性可以用来在调试的时候定位代码执行的行数

start_pc，即字节码行号；line_number，即 Java 源代码行号

在 Code 属性的属性表中，LineNumberTable 属性可以按照任意顺序出现，此外，多个 LineNumberTable 属性可以共同表示一个行号在源文件中表示的内容，即 LineNumberTable 属性不需要与源文件的行一一对应

LineNumberTable 属性表结构：

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;
        u2 line_number;
    } line_number_table[line_number_table_length];
}
```

6．LocalVariableTable 属性

LocalVariableTable 是可选变长属性，位于 Code 属性的属性表中。它被调试器**用于确定方法在执行过程中局部变量的信息。**在 Code 属性的属性中，LocalVariableTable 属性可以按照任意顺序出现。Code 属性中的每个局部变量最多只能有一个 LocalVariableTable 属性。

1) start_pc + 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)
2) index 就是这个变量在局部变量表中的槽位(槽位可复用)
3) name 就是变量名称
4) Descriptor 表示局部变量类型描述

LocalVariableTable 属性表结构：

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    } local_variable_table[local_variable_table_length];
}
```

7．Signature 属性

Signature 属性是可选的定长属性，位于 ClassFile，field_info 或 method_info 结构的属性表中。在 Java 语言中，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量(Type Variables)或参数化类型(Parameterized Types)，则 Signature 属性会为它记录泛型签名信息

8．SourceFile 属性

SourceFile 属性结构

u2	attribute_name_index	1	属性名索引
u4	attribute_length	1	属性长度
u2	sourcefile_index	1	源码文件索引

可以看到，其长度总是固定的 8 个字节

9．其他属性

Java 虚拟机中预定义的属性有 20 多个，这里就不一一介绍了，通过上面几个属性的介绍，只要领会其精髓，其他属性的解读也是易如反掌

总结

通过手动去解读字节码文件，终于大概了解到其构成和原理了

实际上，我们可以使用各种工具来帮我们去解读字节码文件，而不用直接去看这些 16 进制，太繁琐了

9. 小结

本章主要介绍了 Class 文件的基本格式

随着 Java 平台的不断发展，在将来，Class 文件的内容也一定会做进一步的扩充，但是其基本的格式和结构不会做重大调整

从 Java 虚拟机的角度看，通过 Class 文件，可以让更多的计算机语言支持 Java 虚拟机平台。因此，Class 文件结构不仅仅是 Java 虚拟机的执行入口，更是 Java 生态圈的基础和核心

第四章　使用 javap 指令解析 Class 文件

自己分析类文件结构太麻烦了！Oracle 提供了 javap 工具

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，就不会再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。如果你使用的是 Eclipse，则默认情况下，Eclipse 在编译时会帮你生成局部变量表、指令和代码行盘一辆映射表等信息

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。比如我们看到的 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

(1) 解析字节码的作用

通过反编译生成的字节码文件，我们可以深入的了解 Java 代码的工作机制。

但是，自己分析类文件结构太麻烦了，除了使用第三方的 jclasslib 工具之外，Oracle 官方也提供了工具：javap

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(字节码指令)、局部变量表、异常表和代码行偏移量映射表、常量池等信息

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

(2) javac -g 操作

解析字节码文件得到的信息中，有些信息(如局部变量表、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

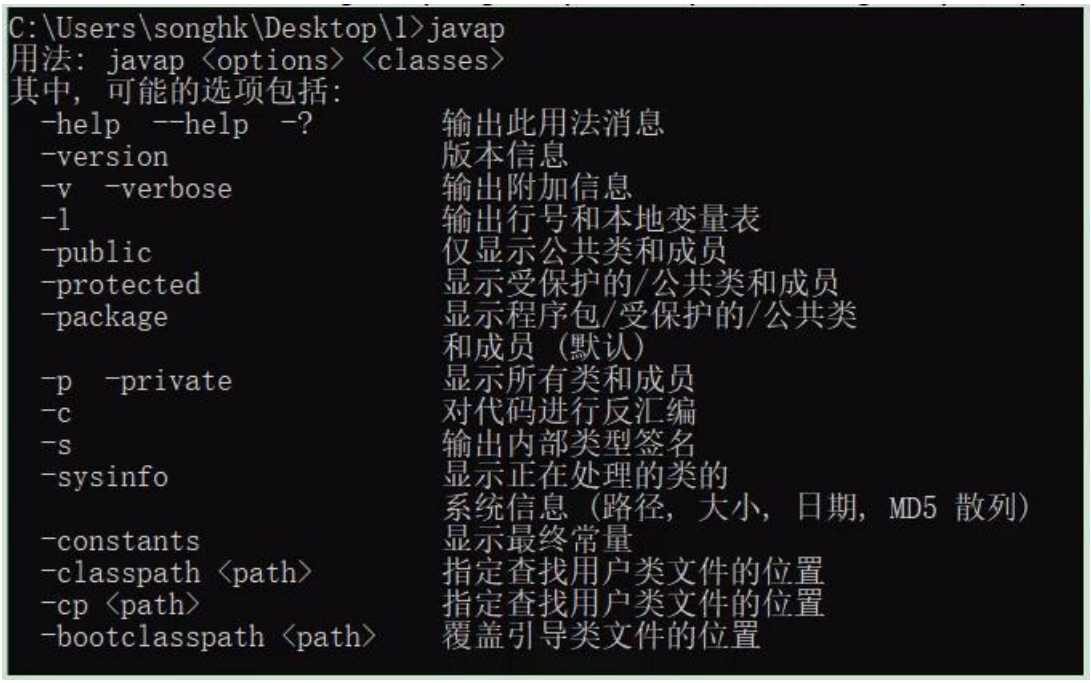
比如，你直接 javac xx.java，就不会在生成对应的局部变量表等信息，如果你使用 javac -g xx.java 就可以生成所有相关信息了。如果你使用的 Eclipse 或 IDEA，则默认情况下，Eclipse、IDEA 在编译时会帮你生成局部变量表、指令和代码行偏移量映射表等信息

(3) javap 的用法

javap 的用法格式：javap

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：



一般常用的是 -v -l -c 三个选项

- 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
- javap -v classxx 除了包含 -c 内容外，还会输出行号、局部变量表信息、常量池等信息

(4) 总结

1. 通过 javap 命令可以查看一个 Java 类反汇编得到的 Class 文件版本号、常量池、访问标识、变量表、指令代码行号表等信息。不显式类索引、父类索引、接口索引集合、()、()等结构
2. 通过对前面的例子代码反汇编文件的简单分析，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
   - Java 栈中：局部变量表、操作数栈
   - Java 堆： 通过对象的地址引用去操作

- 常量池
- 其他如帧数据区、方法区的剩余部分等情况，测试中没有显示出来，这里说明一下

3. 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含义

加载 -> 链接 -> 初始化

引导类加载
扩展类加载
系统类加载

验证
准备
解析

执行类构造器方法<clinit>

```
private static int num = 1;

static{
    num = 2;
    number = 20;
}

private static int number = 10;  //linking之prepare: number = 0 --> initial: 20 --> 10

 public static void main(String[] args) {
     System.out.println(ClassInitTest.num);//2
     System.out.println(ClassInitTest.number);//10
 }
```

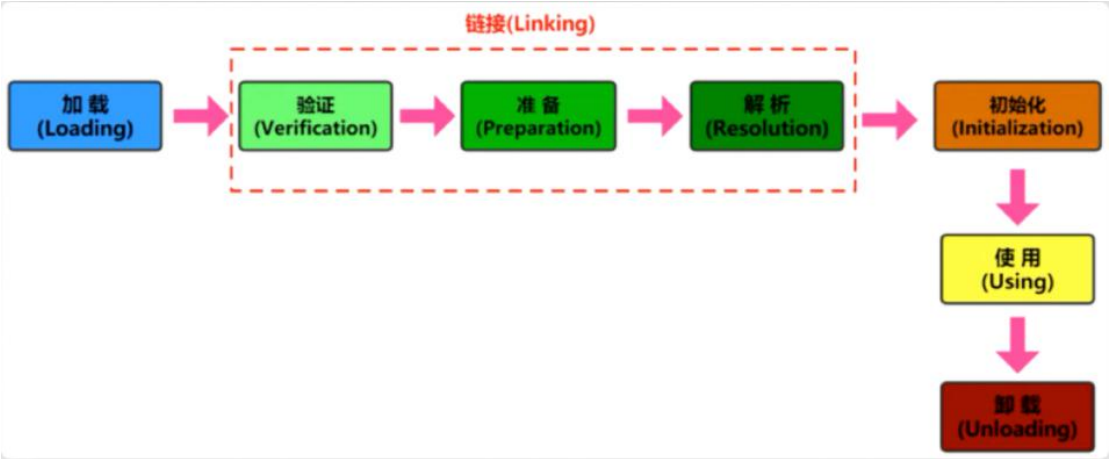
按顺序

类的加载过程详解

第一章　概述

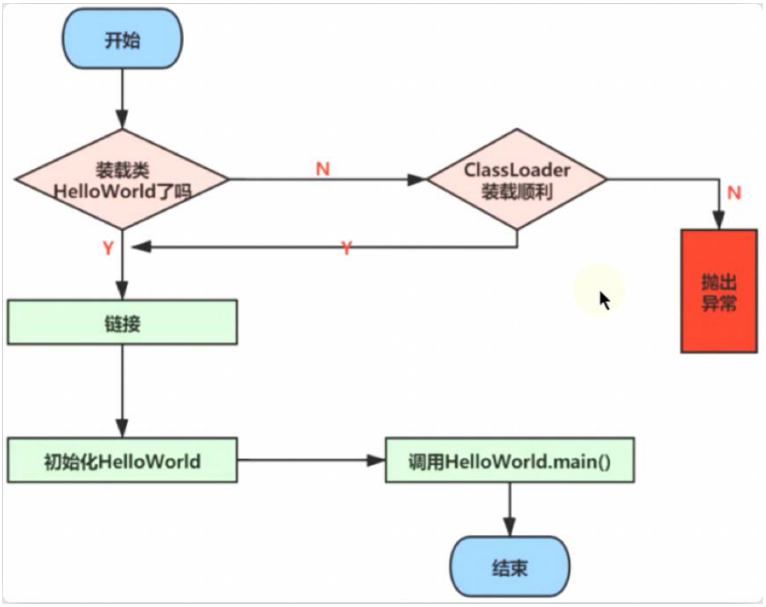
在 Java 中数据类型分为基本数据类型和引用数据类型。基本数据类型由虚拟机预先定义，引用数据类型则需要进行类的加载

按照 Java 虚拟机规范，从 Class 文件到加载到内存中的类，到类卸载出内存位置，它的整个生命周期包括如下七个阶段：



其中，验证、准备、解析 3 个部分统称为链接(Linking)

从程序中类的使用过程看：



大厂面试题

- 蚂蚁金服：

  描述一下 JVM 加载 Class 文件的原理机制？

  一面：类加载过程

- 百度：

  类加载的机制

  Java 类加载过程？

  简述 Java 类加载机制？

- 腾讯：

  JVM 中类加载机制，类加载过程？

- 滴滴：

  JVM 类加载机制

- 美团：

Java 类加载过程

描述一下 JVM 加载 Class 文件的原理机制

第二章　过程一：Loading(加载)阶段

1. 加载完成的操作

加载的理解

这个类模板对象就放在方法区中，
模板对象可视为类的骨架，当类要实例化时需要在此基础上进行扩充。
类模板对象中包含字符常量和static修饰的方法和属性。

所谓加载，简而言之就是将 Java 类的字节码文件加载到机器内存中，并在内存中构建出 Java 类的原型——类模板对象。**所谓类模板对象，其实就是 Java 类在 JVM 内存中的一个快照，JVM 将从字节码文件中解析出的常量池、类字段、类方法等信息存储到模板中，这样 JVM 在运行期便能通过类模板而获取 Java 类中的任意信息，能够对 Java 类的成员变量进行遍历，也能进行 Java 方法的调用

反射的机制即基于这一基础。如果 JVM 没有将 Java 类的声明信息存储起来，则 JVM 在运行期也无法反射

加载完成的操作

加载阶段，简言之，查找并加载类的二进制数据，生成 Class 的实例

在加载类时，Java 虚拟机必须完成以下 3 件事情：

- 通过类的全名，获取类的二进制数据流
- 解析类的二进制数据流为方法区内的数据结构(Java 类模型)
- 创建 java.lang.Class 类的实例，表示该类型。作为方法区这个类的各种数据的访问入口

2. 二进制流的获取方式

对于类的二进制数据流，虚拟机可以通过多种途径产生或获得。(只要所读取

的字节码符合 JVM 规范即可)

- 虚拟机可能通过文件系统读入一个 Class 后缀的文件(最常见)
- 读入 jar、zip 等归档数据包，提取类文件
- 事先存放在数据库中的类的二进制数据
- 使用类似于 HTTP 之类的协议通过网络进行加载
- 在运行时生成一段 Class 的二进制信息等

在获取到类的二进制信息后，Java 虚拟机就会处理这些数据，并最终转为一个 java.lang.Class 的实例

如果输入数据不是 ClassFile 的结构，则会抛出 ClassFormatError

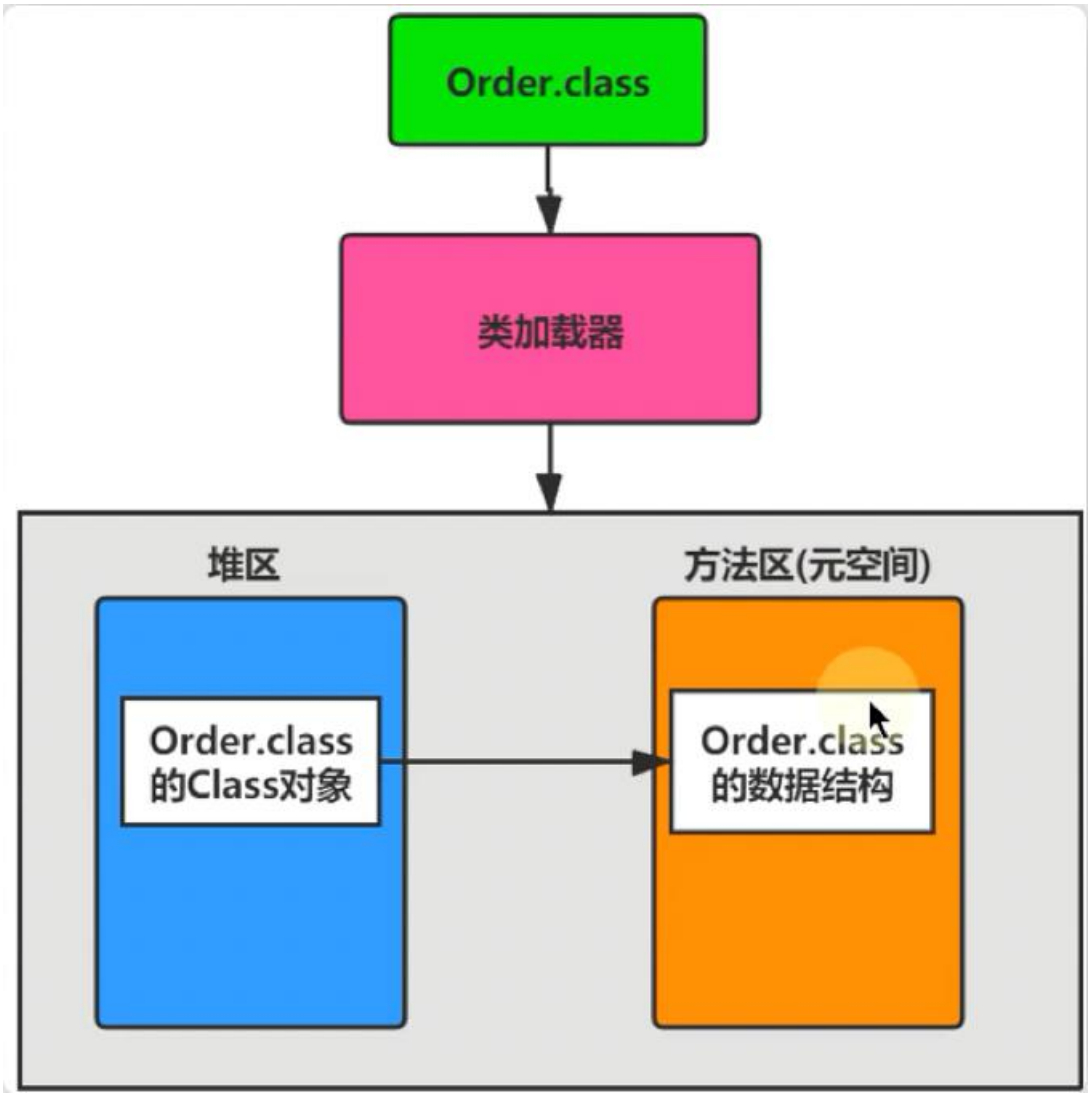
3. 类模型与 Class 实例的位置

➢ 类模型的位置

加载的类在 JVM 中创建相应的类结构，类结构会存储在方法区(JDK 1.8 之前：永久代；JDK 1.8 之后：元空间)

➢ Class 实例的位置

类将 .class 文件加载至元空间后，会在堆中创建一个 java.lang.Class 对象，用来封装类位于方法区内的数据结构，该 Class 对象是在加载类的过程中创建的，每个类都对应有一个 Class 类型的对象



➢ 图示

也就是反射

外部可以通过访问代表 Order 类的 Class 对象来获取 Order 的类数据结构

➢ 再说明

Class 类的构造方法是私有的，只有 JVM 能够创建

java.lang.Class 实例是访问类型元数据的接口，也是实现反射的关键数据、入口。通过 Class 类提供的接口，可以获得目标类所关联的 .class 文件中具体的数据结构：方法、字段等信息

4. 数组类的加载

创建数组类的情况稍微有些特殊，因为数组类本身并不是由类加载器负责创建，而是由 JVM 在运行时根据需要而直接创建的，但数组的元素类型仍然需要依靠类加载器去创建。创建数组类(下述简称 A)的过程：

1. 如果数组的元素类型是引用类型，那么就遵循定义的加载过程递归加载和创建数组 A 的元素类型
2. JVM 使用指定的元素类型和数组唯独来创建新的数组类

如果数组的元素类型是引用类型，数组类的可访问性就由元素类型的可访问性决定。否则数组类的可访问性将被缺省定义为 public

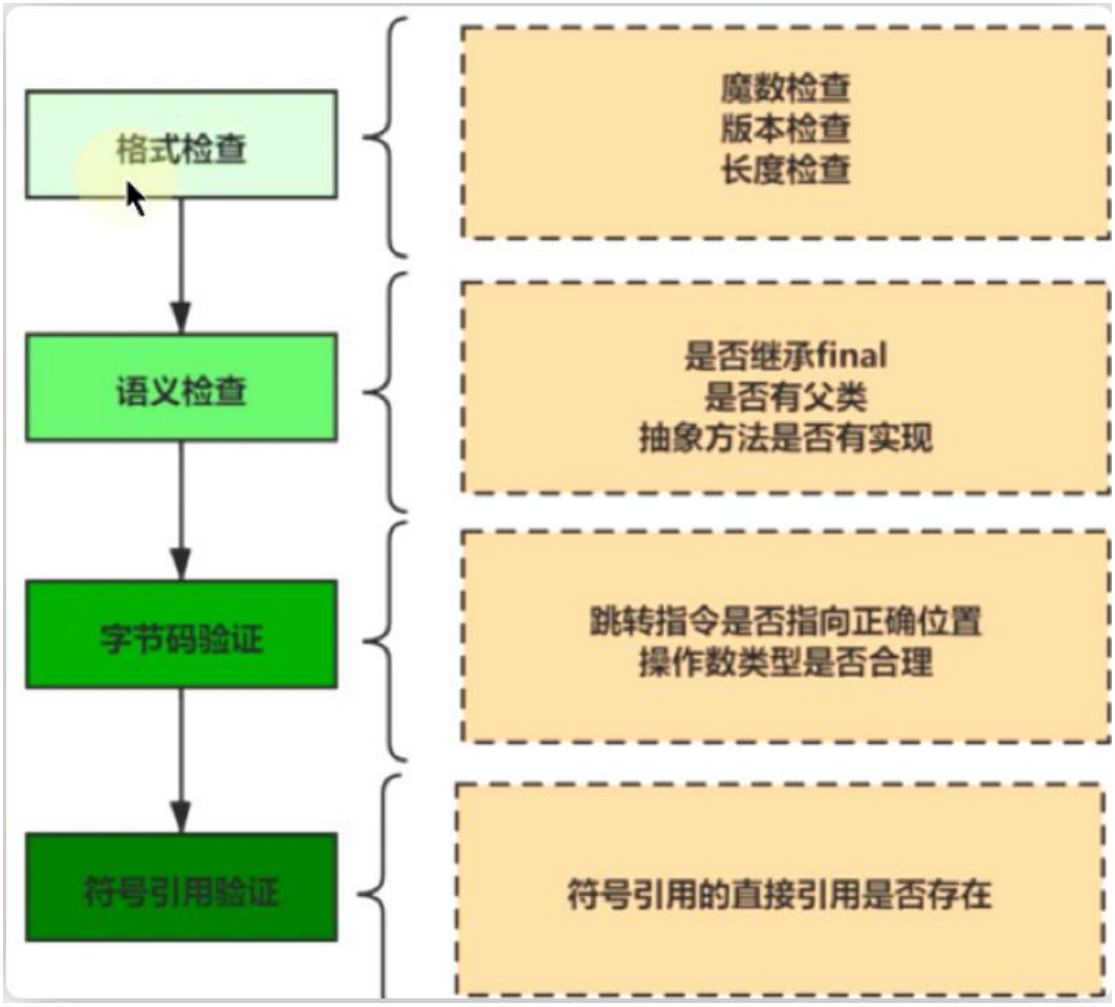
第三章 过程二：Linking(链接)阶段

1. 环节 1:链接阶段之 Verification (验证)

当类加载到系统后，就开始链接操作，验证是链接操作的第一步

它的目的是保证加载的字节码是合法、合理并符合规范的

验证的步骤比较复杂，实际要验证的项目也很繁多，大体上 Java 虚拟机需要做以下检查，如图所示



整体说明：

验证的内容则涵盖了类数据信息的格式验证、语义检查、字节码验证，以及符号引用验证等

- 其中格式验证会和加载阶段一起执行。验证通过之后，类加载器才会成功将类的二进制数据信息加载到方法区中
- 格式验证之外的验证操作将会在方法区中进行

链接阶段的验证虽然拖慢了加载速度，但是它避免了在字节码运行时还需要进行各种检查

具体说明：

1. 格式验证：是否以魔数 0xCAFEBABE 开头，主版本和副版本号是否在当前 Java 虚拟机的支持范围内，数据中每一个项是否都拥有正确的长度等
2. Java 虚拟机会进行字节码的语义检查，但凡在语义上不符合规范的，虚拟机也不会给予验证通过。比如：

- 是否所有的类都有父类的存在(在 Java 里，除了 Object 外，其他类都应该有父类)
- 是否一些被定义为 final 的方法或者类被重写或继承了
- 非抽象类是否实现了所有抽象方法或者接口方法
- 是否存在不兼容的方法(比如方法的签名除了返回值不同，其他都一样，这种方法会让虚拟机无从下手调度；absract 情况下的方法，就不能是 final 的了)

3. Java 虚拟机还会进行字节码验证，字节码验证也是验证过程中最为复杂的一个过程。它试图通过对字节码流的分析，判断字节码是否可以被正确地执行。比如：

- 在字节码的执行过程中，是否会跳转到一条不存在的指令
- 函数的调用是否传递了正确类型的参数
- 变量的赋值是不是给了正确的数据类型等

栈映射帧(StackMapTable)就是在这个阶段，用于检测在特定的字节码处，其局部变量表和操作数栈是否有着正确的数据类型。但遗憾的是，100%准确地判断一段字节码是否可以被安全执行是无法实现的，因此，该过程只是尽可能地检查出可以预知的明显的问题。如果在这个阶段无法通过检查，虚拟机也不会正确装载这个类。但是，如果通过了这个阶段的检查，也不能说明这个类是完全没有问题的

在前面 3 次检查中，已经排除了文件格式错误、语义错误以及字节码的不正确性。但是依然不能确保类是没有问题的

4. 校验器还将进行符号引用的验证。Class 文件在其常量池会通过字符串记录自己将要使用的其他类或者方法。因此，在验证阶段，虚拟机就会检查这些类或者方法确实是存在的，并且当前类有权限访问这些数据，如果一个需要使用类无法在系统中找到，则会抛出 NoClassDefFoundError，如果一个方法无法被找到，则会抛出 NoSuchMethdError 此阶段在解析环节才会执行

2. 链接阶段之 Preparation (准备)

准备阶段(Preparation)，简言之，为类的静态变量分配内存，并将其初始化为默认值

当一个类验证通过时，虚拟机就会进入准备阶段。在这个阶段，虚拟机就会为这个类分配相应的内存空间，并设置默认初始值。Java 虚拟机为各类型变量默认的初始值如表所示：

类型	默认初始值
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0
char	\u0000
boolean	false
reference	null

注意：Java 并不支持 boolean 类型，对于 boolean 类型，内部实现是 int，由于 int 的默认值是 0，故对应的，boolean 的默认值就是 false

注意：

1) 这里不包含基本数据类型的字段用 static final 修饰的情况，因为 final 在编

final在编译的时候就分配了。

比如字符串，在常量池中。

译的时候就会分配了，准备阶段会显式赋值

2) 注意这里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

3) 在这个阶段不会像初始化阶段中那样会有初始化或者代码被执行

```
/**
 * <p>
 * 基本数据类型：非 final 修饰的变量，在准备环节进行默认初始化赋值
 * final 修饰以后，在准备环节直接进行显式赋值
 * <p>
 * 拓展：如果使用字面量的方式定义一个字符串的常量的话，也是在准备环节直接进行显式赋值
 */
public class LinkingTest {
    private static long id;
    private static final int num = 1;

    public static final String constStr = "CONST";
    public static final String constStr1 = new String("CONST");
}
```

3. 环节 3:链接阶段之 Resolution (解析)

在准备阶段(Resolution)，简言之，将类、接口、字段和方法的符号引用转为直接引用

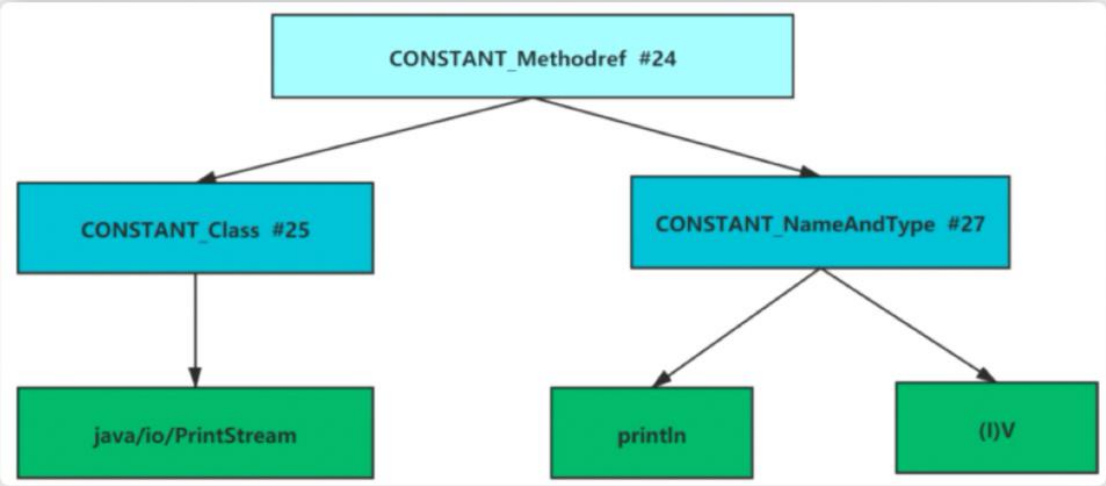
符号引用 ： 常量池，一堆数据结构。

(1) 具体描述： 直接引用：直接指向地址

符号引用就是一些字面量的引用，和虚拟机的内部数据结构和内存分布无关。比较容理解的就是在 Class 类文件中，通过常量池进行了大量的符号引用。但是在程序实际运行时，只有符号引用是不够的，比如当如下 println() 方法被调用时，系统需要明确知道该方法的位置

举例：输出操作 System.out.println() 对应的字节码：

invokevirtual #24 <java/io/PrintStream.println>



以方法为例，Java 虚拟机为每个类都准备了一张方法表，将其所有的方法都列在表中，当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法。通过解析操作，符号引用就可以转变为目标方法在类中方法表中的位置，从而使得方法被成功调用

(2) 小结

所谓解析就是将符号引用转为直接引用，也就是得到类、字段、方法在内存中的指针或者偏移量。因此，可以说，如果直接引用存在，那么可以肯定系统中存在该类、方法或者字段。但只存在符号引用，不能确定系统中一定存在该结构

不过 Java 虚拟机规范并没有明确要求解析阶段一定要按照顺序执行。在 HotSpot VM 中，加载、验证、准备和初始化会按照顺序有条不紊地执行，但链接阶段中的解析操作往往会伴随着 JVM 在执行完初始化之后再执行

(3) 字符串的复习

最后，再来看一下 CONSTANT_String 的解析。由于字符串在程序开发中有着重要的作用，因此，读者有必要了解一下 String 在 Java 虚拟机中的处理。当在 Java 代码中直接使用字符串常量时，就会在类中出现 CONSTANT_String，它表示字符串常量，并且会引用一个 CONSTANT_UTF8 的常量项。在 Java 虚拟机内部运行中的常量池，会维护一张字符串拘留表(intern)，它会保存所有出现过的字符串常量，并且没有重复项。只要以 CONSTANT_String 形式出现的字符串也都会在这张表中。使用 String.intern() 方法可以得到一个字符串在拘留表中的引用，因为该表中没有重复项，所以任何字面相同的字符串的 String.intern() 方法返回总是相等的

第四章　过程三：Initialization(初始化)阶段

初始化阶段，简言之，为类的静态变量赋予正确的初始值

1. 具体描述

类的初始化是类装载的最后一个阶段。如果前面的步骤都没有问题，那么表示类可以顺利装载到系统中。此时，类才会开始执行 Java 字节码。(即：到了初始化阶段，才真正开始执行类中定义的 Java 程序代码)

初始化阶段的重要工作是执行类的初始化方法：() 方法

- 该方法仅能由 Java 编译器生成并由 JVM 调用，程序开发者无法自定义一个同名的方法，更无法直接在 Java 程序中调用该方法，虽然该方法也是由字节码指令所组成
- 它是类静态成员的赋值语句以及 static 语句块合并产生的

2. 说明

加载 ： 将字节码文件加载到内存， 常量池 -> 方法区(Class数据结构)，栈空间(Class对象)
链接 ：
　　验证：验证字节码是否合法，符合要求。
　　准备：加载类的静态变量到内存，并设为默认值。注意static public修饰的会在此处显式赋值
　　解析：将类，接口，方法，字段的符号引用转化为直接引用
初始化：为类的静态变量加载正确的初始值，调用类的初始化方法。

1) 在加载一个类之前，虚拟机总是会试图加载该类的父类，因此父类的 总是在子类 之前被调用，也就是说，父类的 static 块优先级高于子类

2) Java 编译器并不会为所有的类都产生 () 初始化方法。哪些类在编译为字节码后，字节码文件中将不会包含 () 方法？

- 一个类中并没有声明任何的类变量，也没有静态代码块时
- 一个类中声明类变量，但是没有明确使用类变量的初始化语句以及静态代码块来执行初始化操作时
- 一个类中包含 static final 修饰的基本数据类型的字段，这些类字段初始化语句采用编译时常量表达式

1. static 与 final 的搭配问题

```
/**
 *
 * 哪些场景下，Java 编译器就不会生成<clinit>()方法
 */
public class InitializationTest1 {
   //场景 1:对应非静态的字段，不管是否进行了显式赋值，都不会生成<clinit>()方法
   public int num = 1;
   //场景 2：静态的字段，没有显式的赋值，不会生成<clinit>()方法
   public static int num1;
   //场景 3：比如对于声明为 static final 的基本数据类型的字段，不管是否进行了显式赋值，都不会生成<clinit>()方法
   public static final int num2 = 1;
```

因为编译的时候已经确定了。

```
}
```

```
/**
 *
 * 说明：使用 static + final 修饰的字段的显式赋值的操作，到底是在哪个阶段进行的赋值？
 * 情况 1：在链接阶段的准备环节赋值
 * 情况 2：在初始化阶段<clinit>()中赋值
 *
 * 结论：
 * 在链接阶段的准备环节赋值的情况：
 * 1. 对于基本数据类型的字段来说，如果使用 static final 修饰，则显式赋值(直接赋值常量，而非调用方法)通常是在链接阶段的准备环节进行
                                    private static final String = "XXX"
 * 2. 对于 String 来说，如果使用字面量的方式赋值，使用 static final 修饰的话，则显式赋值通常是在链接阶段的准备环节进行
 *
 * 在初始化阶段<clinit>()中赋值的情况
 * 排除上述的在准备环节赋值的情况之外的情况
 *
 * 最终结论：使用 static + final 修饰，且显示赋值中不涉及到方法或构造器调用的基本数据类型或 String 类型的显式赋值，是在链接阶段的准备环节进行
 */
public class InitializationTest2 {

    public static int a = 1; //在初始化阶段<clinit>()中赋值
```

```
    public static final int INT_CONSTANT = 10;    //在链接阶段的准备环节赋值


    public static final Integer INTEGER_CONSTANT1 = Integer.valueOf(100); //在初始化阶段<clinit>()中赋值

    public static Integer INTEGER_CONSTANT2 = Integer.valueOf(1000); //在初始化阶段<clinit>()中赋值


    public static final String s0 = "helloworld0"; //在链接阶段的准备环节赋值

    public static final String s1 = new String("helloworld1"); //在初始化阶段<clinit>()中赋值


}
```

2. (clinit) 的线程安全性

对于 () 方法的调用，也就是类的初始化，虚拟机会在内部确保其多线程环境中的安全性

虚拟机会保证一个类的 () 方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的 () 方法，其他线程都需要阻塞等待，直到活动线程执行 () 方法完毕

正是因为函数 () 带锁线程安全的，因此，如果一个在类的 () 方法中有耗时很长的操作，就可能造成多个线程阻塞，引发死锁。并且这种死锁是很难发现的，因为看起来它们并没有可用的锁信息

如果之前的线程成功加载了类，则等在队列中的线程就没有机会再执行 () 方法了。那么，当需要使用这个类时，虚拟机会直接返回给它已经准备好的信息

3. 类的初始化情况：主动使用 vs 被动使用

Java 程序对类的使用分为两种：主动使用 和 被动使用

一、主动使用

Class 只有在必须要首次使用的时候才会被装载，Java 虚拟机不会无条件地装载 Class 类型。Java 虚拟机规定，一个类或接口在初次使用前，必须要进行初始化。这里指的"使用"，是指主动使用，主动使用只有下列几种情况：(即：如果出现如下的情况，则会对类进行初始化操作。而初始化操作之前的加载、验证、准备已经完成)

1) 当创建一个类的实例时，比如使用 new 关键字，或者通过反射、克隆、反序列化
2) 当调用类的静态方法时，即当使用了字节码 invokestatic 指令
3) 当使用类、接口的静态字段时(final 修饰特殊考虑)，比如，使用 getstatic 或者 putsttic 指令。(对应访问变量、赋值变量操作)
4) 当使用 java.lang.reflect 包中的方法反射类的方法时。比如：Class.forname("com.atguigu.java.Test")
5) 当初始化子类时，如果发现其分类还没有进行过初始化，则需要先触发其父类的初始化
6) 如果一个接口定义了 default 方法，那么直接实现或者间接实现该接口的类的初始化，该接口要在其之前被初始化
7) 当虚拟机启动时，用户需要指定一个要执行的主类(包含 main() 方法的那个类)，虚拟机会先初始化这个主类
8) 当初次调用 MethodHandle 实例时，初始化该 MethodHandle 指向的方法所在的类。(涉及解析 REF_getStatic、REF_putStatic、REF_invokeStatic 方法

句柄对应的类)

针对 5，补充说明：

当 Java 虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则并不适用于接口

- 在初始化一个类时，并不会先初始化它所实现的接口
- 在初始化一个接口时，并不会先初始化它的父接口

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化，只有当程序首次使用特定接口的静态字段时，才会导致该接口的初始化

针对 7，说明：

JVM 启动的时候通过引导类加载器加载一个初始类。这个类在调用 public static void main(String[]) 方法之前被链接和初始化。这个方法的执行将依次导致所需的类的加载、链接和初始化

二、被动使用

除了以上的情况属于主动使用，其他的情况均属于被动使用。被动使用不会引起类的初始化

也就是说：并不是在代码中出现的类，就一定会被加载或者初始化。如果不符合主动使用的条件，类就不会初始化

1) 当访问一个静态字段时，只有真正声明这个字段的类才会被初始化

- 当通过子类引用父类的静态变量，不会导致子类初始化

2) 通过数组定义类引用，不会触发此类的初始化

3) 引用变量不会触发此类或接口的初始化。因为常量在链接阶段就已经被显式赋值了

4) 调用 ClassLoader 类的 loadClass() 方法加载一个类，并不是对类的主动使用，不会导致类的初始化

如果针对代码，设置参数 -XX:+TraceClassLoading，可以追踪类的加载信息

并打印出来

第五章　过程四：类的 Using(使用)

任何一个类型在使用之前都必须经历过完整的加载、链接和初始化 3 个类加载步骤。一旦一个类型成功经历过这 3 个步骤之后，便“万事俱备，只欠东风”，就等着开发者使用了

开发人员可以在程序中访问和调用它的静态类成员信息(比如：静态字段、静态方法)，或者使用 new 关键字为其创建对象实例

第六章　过程五：类的 Unloading(卸载)

（一）类、类的加载器、类的实例之间的引用关系

在类加载器的内部实现中，用一个 Java 集合来存放所加载类的引用。另一方面，一个 Class 对象总是会引用它的类加载器，调用 Class 对象的 getClassLoader() 方法，就能获得它的类加载器。由此可见，代表某个类的 Class 实例与其类的加载器之间为双向关联关系　　Class对象 和 ClassLoader对象 不是一个！！

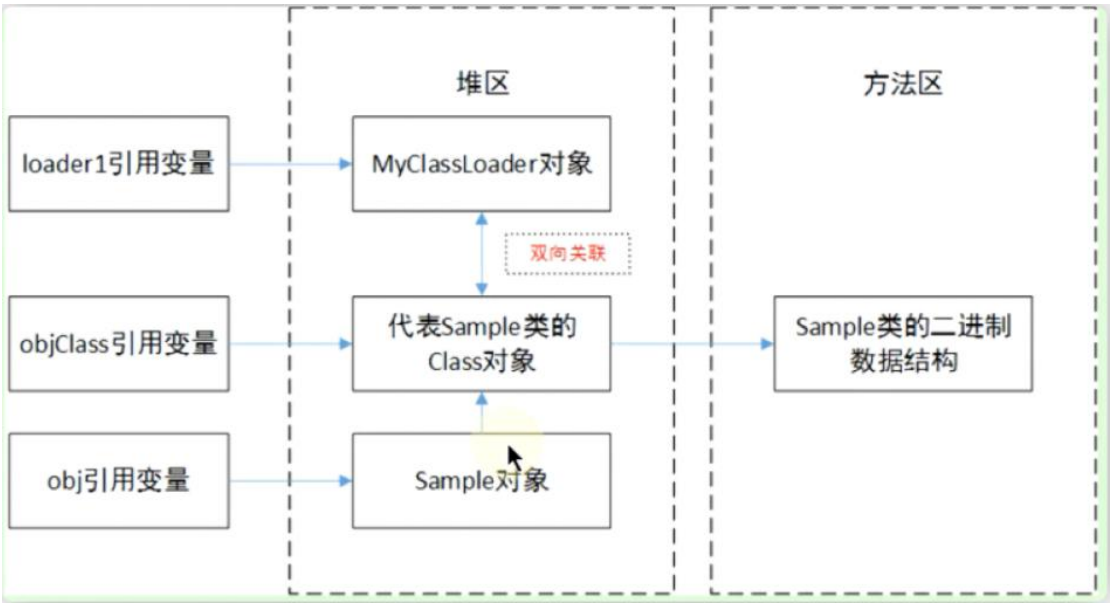
一个类的实例总是引用代表这个类的 Class 对象。在 Object 类中定义了 getClass() 方法，这个方法返回代表对象所属类的 Class 对象的引用。此外，所有的 Java 类都有一个静态属性 Class，它引用代表这个类的 Class 对象

（二）类的生命周期

当 Sample 类被加载、链接和初始化后，它的生命周期就开始了。当代表 Sample 类的 Class 对象不再被引用，即不可触及时，Class 对象就会结束生命周期，Sample 类在方法区内的数据也会被卸载，从而结束 Sample 类的生命周期

一个类何时结束生命周期，取决于代表它的 Class 对象何时结束生命周期

（三）具体例子



Loader1 变量和 obj 变量间接应用代表 Sample 类的 Class 对象，而 objClass 变量则直接引用它

如果程序运行过程中，将上图左侧三个引用变量都置为 null，此时 Sample 对象结束生命周期，MyClassLoader 对象结束生命周期，代表 Sample 类的 Class 对象也结束生命周期，Sample 类在方法区内的二进制数据被卸载

当再次有需要时，会检查 Sample 类的 Class 对象是否存在，如果存在会直接使用，不再重新加载；如果不存在 Sample 类会被重新加载，在 Java 虚拟机的堆区会生成一个新的代表 Sample 类的 Class 实例(可以通过哈希码查看是否是同一个实例)

（四）类的卸载

1) 启动类加载器加载的类型在整个运行期间是不可能被卸载的(JVM 和 JSL 规范)

2) 被系统类加载器和扩展类加载器加载的类型在运行期间不太可能被卸载，因为系统类加载器实例或者扩展类的实例基本上在整个运行期间总能直接或

者间接的访问的到，其达到 unreachable 的可能性极小

3) 被开发者自定义的类加载器实例加载的类型只有在很简单的上下文环境中才能被卸载，而且一般还要借助于强制调用虚拟机的垃圾收集功能才可以做到。可以预想，稍微复杂点的应用场景(比如：很多时候用户在开发自定义类的加载器实例的时候采用缓存的策略以提高系统性能)，被加载的类型在运行期间也是几乎不太可能被卸载的(至少卸载的时间是不确定的)

综合以上三点，一个已经加载的类型被卸载的几率很小至少被卸载的时间是不确定的。同时我们可以看的出来，开发者在开发代码时候，不应该对虚拟机的类型卸载做任何假设的前提下，来实现系统中的特定功能

回顾：方法区的垃圾回收

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型

HotSpot 虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收

判定一个常量是否"废弃"还是相对简单，而要判定一个类型是否属于"不再被使用的类"的条件就比较苛刻了。需要同时满足下面三个条件：

- 该类所有的实例都已经被回收。也就是 Java 堆中不存在该类及其任何派生子类的实例
- 加载该类的类加载器已经被回收。这个条件除非是经过精心设计的可替换类加载器的场景，如 OSGI、JSP 的重加载等，否则通常是很难达成的
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

Java 虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是"被允许"，而并不是和对象一样，没有引用了就必然会回收

再谈类的加载器

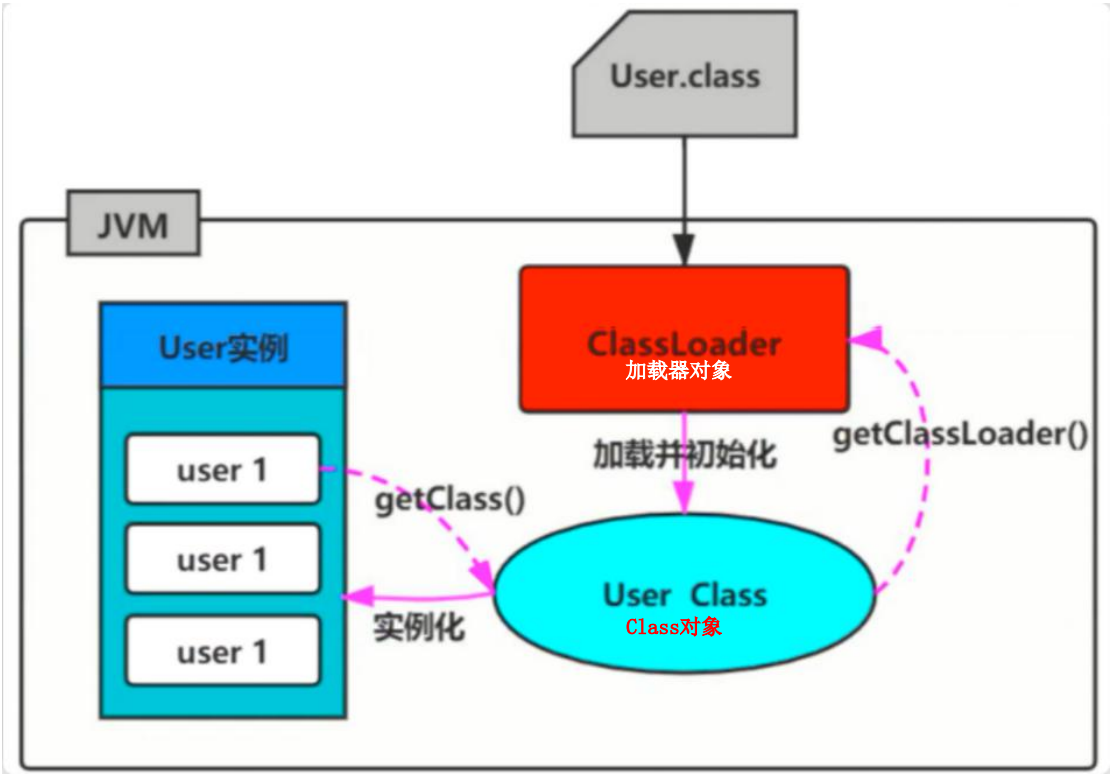
第一节　概述

类加载器是 JVM 执行类加载机制的前提

ClassLoader 的作用：

属于加载阶段的加载，

ClassLoader 是 Java 的核心组件，所有的 Class 都是由 ClassLoader 进行加载的，ClassLoader 负责通过各种方式将 Class 信息的二进制数据流读入 JVM 内部，转换为一个与目标类对应的 java.lang.Class 对象实例。然后交给 Java 虚拟机尽心链接、初始化等操作。因此，ClassLoader 在整个装载阶段，只能影响到类的加载，而无法通过 ClassLoader 去改变类的链接和初始化行为。至于它是否可以运行，则由 Execution Engine 决定



类加载器最早出现在 Java 1.0 版本中，那个时候只是单纯地为了满足 Java Applet 应用而被研发出来，但如今类加载器却在 OSGI、字节码加解密领域大放异彩。这主要归功于 Java 虚拟机的设计者们当初在设计类加载器的时候，并没有考虑将它绑定在 JVM 内部，这样做的好处就是能够更加灵活和动态地执行类加载操作

1. 大厂面试题

- **蚂蚁金服：**

  深入分析 ClassLoader，双亲委派机制

  类加载器的双亲委派模型是什么？

  一面：双亲委派机制及使用原因

- **百度：**

  都有哪些类加载器，这些类加载器都加载哪些文件？

  手写一个类加载器 Demo

  Class 的 forName("java.lang.String") 和 Class 的 getClassLoader() 的 loadClass("java.lang.String")有什么区别？

- **腾讯：**

  什么是双亲委派模型？

  类加载器有哪些？

- **小米：**

  双亲委派模型介绍一下

- **滴滴：**

  简单说说你了解的类加载器

  一面：讲一下双亲委派模型，以及其优点

- **字节跳动：**

  什么事类加载器，类加载器有哪些？

- **京东：**

  类加载器的双亲委派模型是什么？

  双亲委派机制可以打破吗？为什么？

2. 类加载的分类

类的加载分类：显式加载 vs 隐式加载

Class 文件的显式加载与隐式加载的方式是指 JVM 加载 Class 文件到内存的方式

- 显式加载指的是在代码中通过调用 ClassLoader 加载 Class 对象，如直接使用 Class.forName(name) 或 this.getClass().getClassLoader().loadClass() 加载 Class 对象
- 隐式加载则是不直接在代码中调用 ClassLoader 的方法加载 Class 对象，而是通过虚拟机自动加载到内存中，如在加载某个类的 Class 文件时，该类的 Class 文件中引用了另外一个类的对象，此时额外引用的类将通过 JVM 自动加载到内存中

在日常开发中以上两种方式一般会混合使用

3. 类加载器的必要性

一般情况下，Java 开发人员并不需要在程序中显式地使用类加载器，但是了解类加载器的加载机制却显得至关重要。从以下几个方面说：

- 避免在开发中遇到 java.lang.ClassNotFoundException 异常或 java.lang.NoClassDeFoundError 异常时手足无措。只有了解类加载器的加载

机制才能够在出现异常的时候快速地根据错误异常日志定位问题和解决问题

- 需要支持类的动态加载或需要对编译后的字节码文件进行加解密操作时，就需要与类加载器打交道了
- 开发人员可以在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一些自定义的处理逻辑

4. 命名空间

1) 何为类的唯一性？

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确认其在 Java 虚拟机中的唯一性。每一个类加载器，都拥有一个独立的类名称空间：比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义。否则，即使这两个类源自同一个 Class 文件，被同一个虚拟机加载，只要加载他们的类加载器不同，那这两个类就必定不相等

2) 命名空间

- 每个类加载器都有自己的命名空间，命名空间由该加载器所有的父加载器所加载的类组成
- 在同一命名空间中，不会出现类的完整名字(包括类的包名)相同的两个类
- 在不同的命名空间中，有可能会出现类的完整名字(包括类的包名)相同的两个类

在大型应用中，我们往往借助这一特性，来运行同一个类的不同版本

5. 类加载机制的基本特征

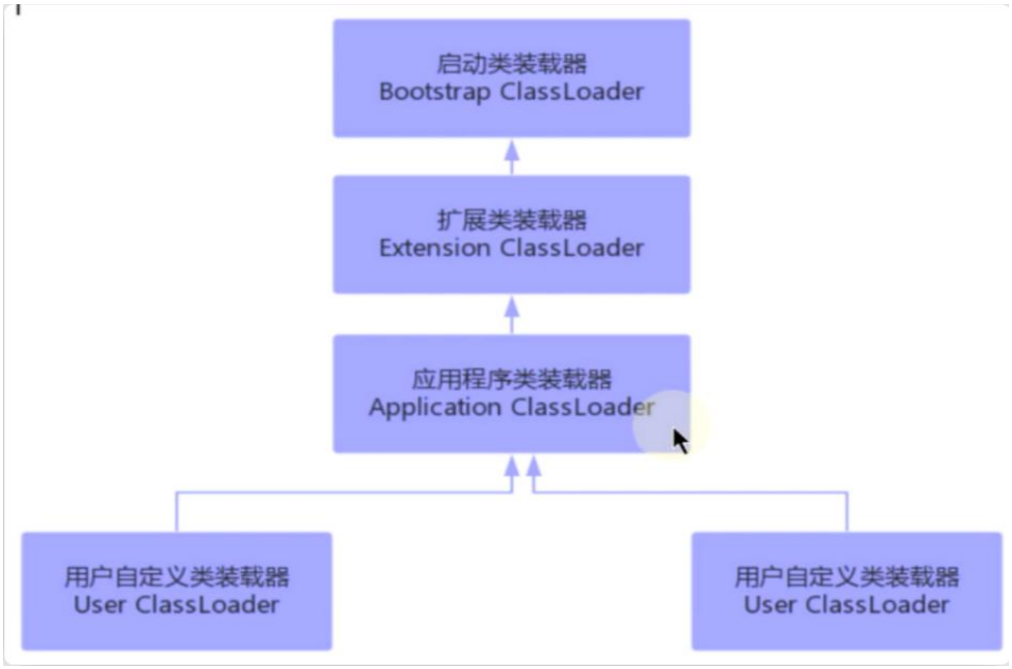
通常类加载机制有三个基本特征：

- 双亲委派模型。但不是所有类加载都遵守这个模型，有的时候，启动类加载器所加载的类型，是可能要加载用户代码的，比如 JDK 内部的 ServiceProvider/ServiceLoader 机制，用户可以在标准 API 框架上，提供自己的实现，JDK 也需要提供些默认的参考实现。例如，Java 中 JNDI、JDBC、文件系统、Cipher 等很多方面，都是利用的这种机制，这种情况就不会用双亲委派模型去加载，而是利用所谓的上下文加载器
- 可见性，子类加载器可以访问父加载器加载的类型，但是反过来是不允许的。不然，因为缺少必要的隔离，我们就没有办法利用类加载器去实现容器的逻辑
- 单一性，由于父加载器的类型对于子加载器是可见的，所以父加载器中加载过的类型，就不会在子加载器中重复加载。但是注意，类加载器"邻居"间，同一类型仍然可以被加载多次，因为相互并不可见

第二节　复习：类的加载器分类

JVM 支持两种类型的类加载器，分别为引导类加载器(Bootstrap ClassLoader)和自定义类加载器(User-Defined ClassLoader)

从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器。无论类加载器的类型如何划分，在程序中我们最常见的类加载器结构主要是如下情况：



- 除了顶层的启动类加载器外，其余的类加载器都应当有自己的"父类"加载器
- 不同类加载器看似是继承(Inheritance)关系，实际上是包含关系。在下层加载器中，包含着上层加载器的引用

```
class ClassLoader {
    ClassLoader parent; //父类加载器
    public ClassLoader(ClassLoader parent) {
        this.parent = parent;
    }
}
class ParentClassLoader extends ClassLoader {
    public ParentClassLoader(ClassLoader parent) {
        super(parent);
    }
```

```
}
class ChildClassLoader extends ClassLoader {
    public ChildClassLoader(ClassLoader parent) {
        //parent = new ParentClassLoader();
        super(parent);
    }
}
```

1. 引导类加载器

启动类加载器(引导类加载器)

- 这个类加载使用 C/C++ 语言实现的，嵌套在 JVM 内部
- 它用来加载 Java 的核心库(JAVA_HOME/jre/lib/rt.jar 或 sun.boot.class.path 路径下的内容)。用于提供 JVM 自身需要的类
- 并不继承自 java.lang.ClassLoader，没有父加载器
- 出于安全考虑，Bootstrap 启动类加载器之加载包名为 java、javax、sun 等开头的类
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器

```
[Loaded java.net.URL from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.util.jar.Manifest from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.Launcher from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.Launcher$AppClassLoader from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.Launcher$ExtClassLoader from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.CodeSource from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.lang.StackTraceElement from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
```

使用 -XX:+TraceClassLoading 参数

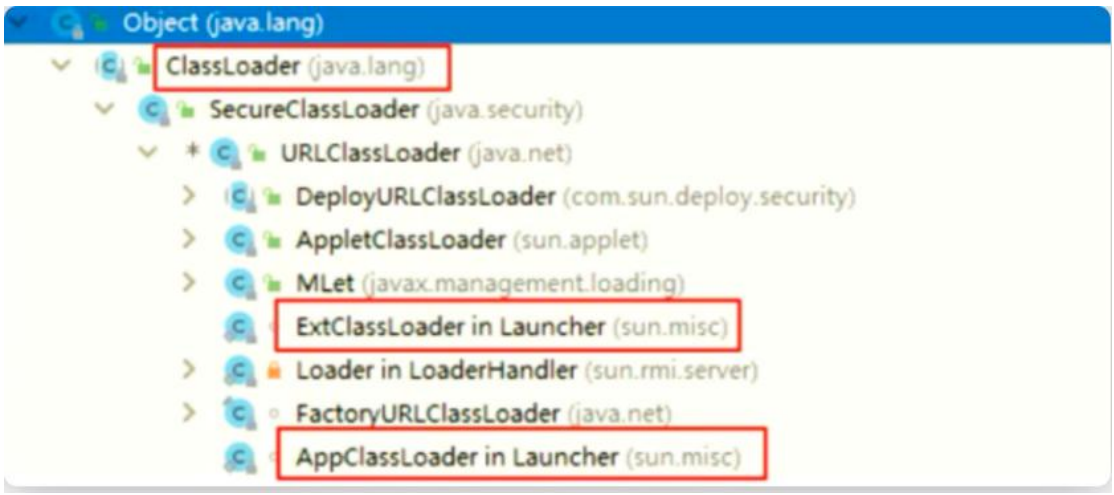
启动类加载器使用 C++ 编写的？Yes！

- C/C++：指针函数 & 函数指针、C++ 支持多继承、更加高效
- Java ：由 C++ 演变而来，(C++)-- 版，单继承

2. 扩展类加载器

扩展类加载器(Extension ClassLoader)

- Java 语言编写，由 sun.misc.Launcher$ExtClassLoader 实现
- 继承于 ClassLoader 类
- 父类加载器为启动类加载器
- 从 java.ext.dirs 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 jre/lib/ext 子目录下加载类库。如果用户创建的 JAR 放在此目录下，也会自动由扩展类加载器加载



3. 系统类加载器

应用程序类加载器(系统类加载器，AppClassLoader)

- Java 语言编写，由 sun.misc.Launcher$AppClassLoader 实现

- 继承于 ClassLoader 类
- 父类加载器为扩展类加载器
- 它负责加载环境变量 classpath 或系统属性 java.class.path 指定路径下的类库
- 应用程序中的类加载器默认是系统类加载器
- 它是用户自定义类加载器的默认父加载器
- 通过 ClassLoader 的 getSystemClassLoader() 方法可以获取到该类加载器

4. 用户自定义类加载器

- 在 Java 的日常应用程序开发中，类的加载几乎是由上述 3 种类加载器相互配合执行的。在必要时，我们还可以自定义类加载器，来定制类的加载方式
- 体现 Java 语言强大生命力和巨大魅力的关键因素之一便是，Java 开发者可以自定义类加载器来实现类库的动态加载，加载源可以是本地的 JAR 包，也可以是网络上的远程资源
- 通过类加载器可以实现非常绝妙的插件机制，这方面的实际应用案例不胜枚举。例如，著名的 OSGI 组件框架，再如 Eclipse 的插件机制。类加载器为应用程序提供了一种动态增加新功能的机制，这种机制无需重新打包发布应用程序就能实现
- 同时，自定义加载器能够实现应用隔离，例如 Tomcat、Spring 等中间件和组件框架都在内部实现了自定义的加载器，并通过自定义加载器隔离不同的组件模块。这种机制比 C/C++ 程序要好太多，想不修改 C/C++ 程序就能为其新增功能，几乎是不可能的，仅仅一个兼容性便能阻挡所有美好的设想
- 自定义类加载器通常需要继承于 ClassLoader

第三节　测试不同的类加载器

每个 Class 对象都会包含一个定义它的 ClassLoader 的一个引用

获取 ClassLoader 的途径

```
获取当前类的 ClassLoader

clazz.getClassLoader();


获得当前线程上下文的 ClassLoader

Thread.currentThread().getContextClassLoader();


获得系统的 ClassLoader

ClassLoader.getSystemClassLoader();
```

说明：

站在程序的角度看，引导类加载器与另外两种类加载器(系统类加载器和扩展类加载器)并不是同一个层次意义上的加载器，引导类加载器是使用 C++ 语言编写而成的，而另外两种类加载器则是使用 Java 语言编写的。由于引导类加载器压根儿就不是一个 Java 类，因此在 Java 程序中只能打印出空值

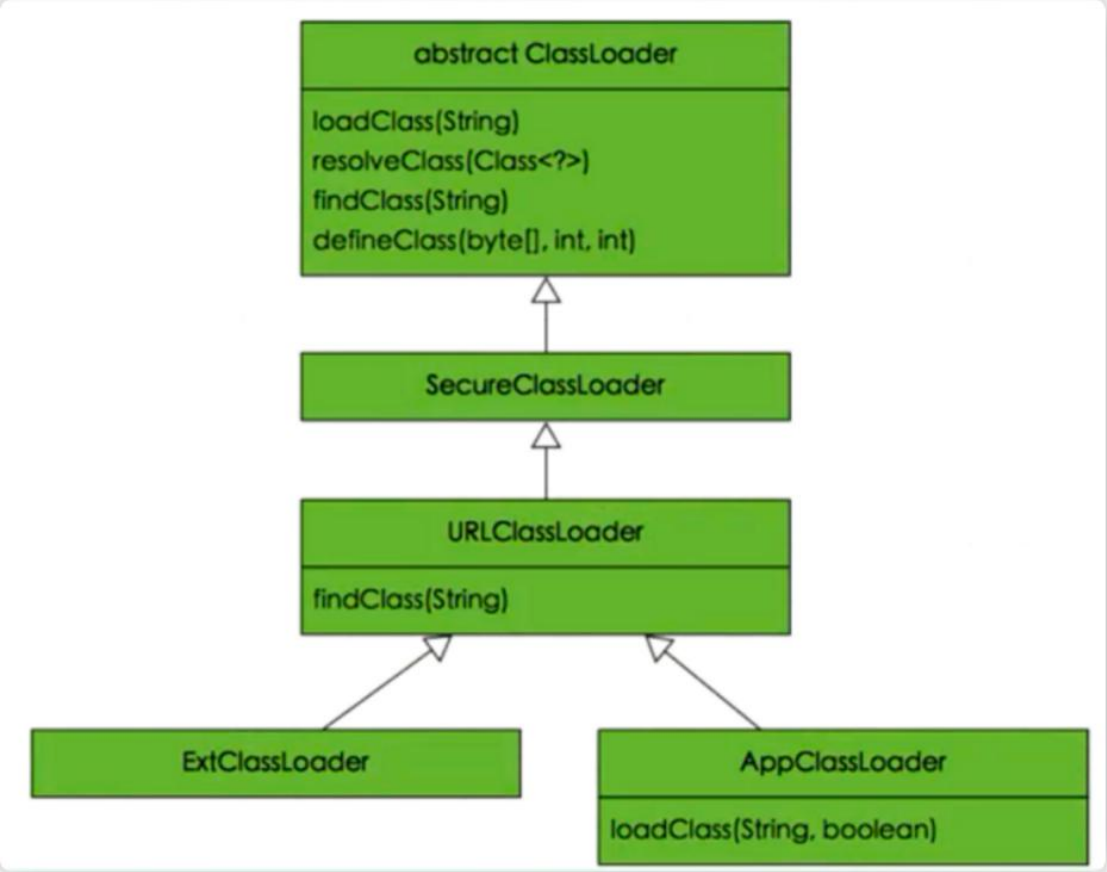
数组类的 Class 对象，不是由类加载器去创建的，而是在 Java 运行期 JVM 根据需要自动创建的。对于数组类的类加载器来说，是通过 Class.geetClassLoader() 返回的，与数组当中元素类型的类加载器是一样的：如果数组当中的元素类型是基本数据类型，数组类是没有类加载器的

```
String[] strArr = new String[6];

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

//运行结果：null
```

```
ClassLoaderTest[] test = new ClassLoaderTest[1];
System.out.println(test.getClass().getClassLoader());
//运行结果：sun.misc.Launcher$AppClassLoader@18b4aac2

int[] inst = new int[2];
System.out.println(inst.getClass().getClassLoader());
//运行结果：null
```

第四节　ClassLoader 源码解析

ClassLoader 与现有类加载的关系：



除了以上虚拟机自带的加载器外，用户还可以定制自己的类加载器。Java 提供了抽象类 java.lang.ClassLoader，所有用户自定义的类加载器都应该继承 ClassLoader 类

1. ClassLoader 的主要方法

抽象类 ClassLoader 的主要方法：(内部没有抽象方法)

- public final ClassLoader getParent()

返回该类加载器的超类加载器

- public Class<?> loadClass(String name) throws ClassNotFoundException

加载名称为 name 的类，返回结果为 java.lang.Class 类的实例。如果找不到类，则返回 ClassNotFountException 异常。该方法中的逻辑就是双亲委派模式的实现

- protected Class<?> findClass(String name) throws ClassNotFoundException

查找二进制名称为 name 的类，返回结果为 java.lang.Class 类的实例。这是一个受保护的方法，JVM 鼓励我们重写此方法，需要自定义加载器遵循双亲委派机制，该方法会在检查完父类加载器之后被 loadClass() 方法调用

在 JDK 1.2 之前，在自定义类加载时，总会去继承 ClassLoader 类并重写 loadClass 方法，从而实现自定义的类加载类。但是在 JDK 1.2 之后已不再建议用户去覆盖 loadClass() 方法，而是建议把自定义的类加载逻辑写在 find Class() 方法中，从前面的分析可知，findClass() 方法是在 loadClass() 方法中被调用的，当 loadClass() 方法中父加载器加载失败后，则会调用自己的 findClass() 方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委派机制

需要注意的是 ClassLoader 类中并没有实现 findClass() 方法的具体代码逻辑，取而代之的是抛出 ClassNotFoundException 异常，同时应该知道的是 findClass() 方法通常是和 defineClass() 方法一起使用的。一般情况下，在自定义类加载器时，会直接覆盖 ClassLoader 的 findClass() 方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用 defineClass() 方法生成类的 Class 对象

- protected final Class<?> defineClass(String name, byte[] b, int off, int len)

根据给定的字节数组 b 转换为 Class 的实例，off 和 len 参数表示实际 Class 信息在 byte 数组中的位置和长度，其中 byte 数组 b 是 ClassLoader 从外部获取的。这是受保护的方法，只有在自定义 ClassLoader 子类中可以使用

defineClass() 方法是用来将 byte 字节流解析成 JVM 能够识别的 Class 对象(ClassLoader 中已实现该方法逻辑)，通过这个方法不仅能够通过 Class 文件实例化 Class 对象，也可以通过其它方式实例化 Class 对象，如通过网络中接收一个类的字节码，然后转换为 byte 字节流创建对应的 Class 对象

defineClass() 方法通常与 findClass() 方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖 ClassLoader 的 findClass() 方法并编写加载规则，取

得要加载类的字节码后转换成流，然后调用 defineClass() 方法生成类的 Class 对象

简单举例：

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
   //获取类的字节数组
   byte[] classData = getClassData(name);
   if (classData == null) {
      throw new ClassNotFoundException();
   } else {
      //使用 defineClass 生成 Class 对象
      return defineClass(name, classData, 0, classData.length);
   }
}
```

- protected final void resolveClass(Class<?> c)

链接指定的一个 Java 类。使用该方法可以使用类的 Class 对象创建完成的同时也被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用

- protected final Class<?> findLoadedClass(String name)

查找名称为 name 的已经被加载过的类，返回结果为 java.lang.Class 类的实例。这个方法是 final 方法，无法被修改

- private final ClassLoader parent;

它也是一个 ClassLoader 的实例，这个字段所表示的 ClassLoader 也称为这个 ClassLoader 的双亲。在类加载的过程中，ClassLoader 可能会将某些请求交予自

己的双亲处理

① loadClass() 的剖析

测试代码：

```
ClassLoader.getSystemClassLoader().loadClass("com.atguigu.java.User")
```

```
/**
    * Loads the class with the specified <a href="#binary-name">binary name</a>.  The
    * default implementation of this method searches for classes in the
    * following order:
    *
    * <ol>
    *
    *   <li><p> Invoke {@link #findLoadedClass(String)} to check if the class
    *   has already been loaded.  </p></li>
    *
    *   <li><p> Invoke the {@link #loadClass(String) loadClass} method
    *   on the parent class loader.  If the parent is {@code null} the class
    *   loader built into the virtual machine is used, instead.  </p></li>
    *
    *   <li><p> Invoke the {@link #findClass(String)} method to find the
    *   class.  </p></li>
    *
```

```
    * </ol>
    *
    * <p> If the class was found using the above steps, and the
    * {@code resolve} flag is true, this method will then invoke the {@link
    * #resolveClass(Class)} method on the resulting {@code Class} object.
    *
    * <p> Subclasses of {@code ClassLoader} are encouraged to override {@link
    * #findClass(String)}, rather than this method.  </p>
    *
    * <p> Unless overridden, this method synchronizes on the result of
    * {@link #getClassLoadingLock getClassLoadingLock} method
    * during the entire class loading process.
    *
    * @param  name
    *         The <a href="#binary-name">binary name</a> of the class
    *
    * @param  resolve
    *         If {@code true} then resolve the class
    *
    * @return  The resulting {@code Class} object
    *
    * @throws  ClassNotFoundException
    *          If the class could not be found
```

```
     */
    protected Class<?> loadClass(String name, boolean resolve) //resolve:true 加载Class的同时进行解析操作
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) { //同步操作，保证只能加载一次
           // 首先，在缓存中判断是否已经加载同名类
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                 // 获取当前类加载类的父类加载器
                    if (parent != null) {
                // 如果存在父类加载器，则调用父类加载器进行类加载
                        c = parent.loadClass(name, false);
                    } else { // parent 为 null ：父类加载器是引导类加载器
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }
```

```
                if (c == null) { // 当前类的加载器的父类加载未加载其此类 or 当前类的加载器未
加载此类
                    // 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
                    PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                    PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {//是否进行解析操作
                resolveClass(c);
            }
            return c;
        }
    }
```

2. SecureClassLoader 与 URLClassLoader

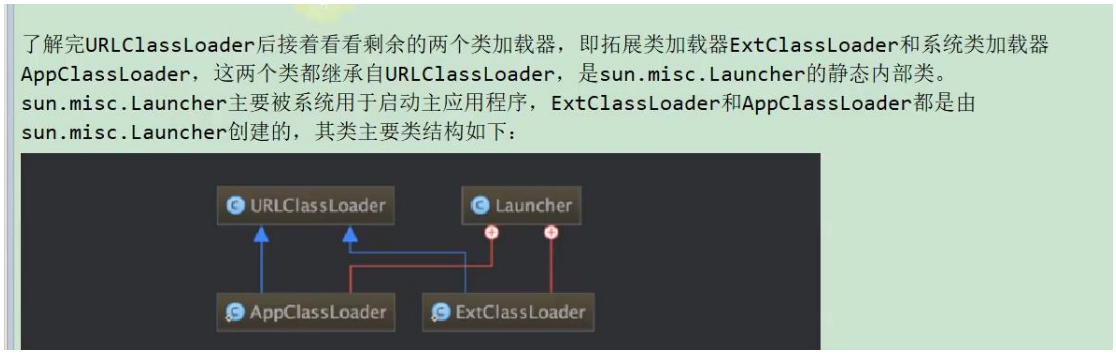
接着 SecureClassLoader 扩展了 ClassLoader，新增了几个与使用相关的代码

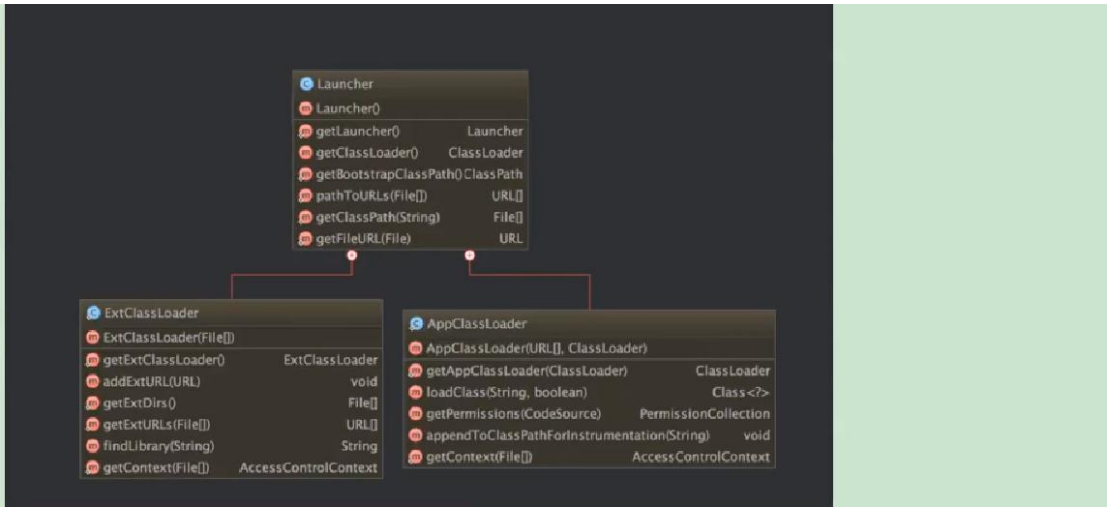
源(对代码源的位置及其证书的验证)和权限定义类验证(主要针对 Class 源码的访问权限)的方法，一般我们不会直接跟这个类打交道，更多的是与它的子类 URLClassLoader 有所关联

前面说过，ClassLoader 是一个抽象类，很多方法是空的没有实现，比如 findClass()、findResource() 等。而 URLClassLoader 这个实现类为这些方法提供了具体的实现。并新增了 URLClassPath 类协助取得 Class 字节码流等功能。在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承 URLClassLoader 类，这样就可以避免自己去编写 findClass() 方法及其获取字节码流的方式，使自定义类加载器编写更加简洁

3. ExtClassLoader 与 AppClassLoader

ExtClassLoader 并没有重写 loadClass() 方法，这足以说明其遵循双亲委派模式，而 AppClassLoader 重载了 loadClass() 方法，但最终调用的还是父类 loadClass() 方法，因此依然遵循双亲委派模式





4. Class.forName() 与 ClassLoader.loadClass()

- Class.forName()：是一个静态方法，最常用的是 Class.forName(String className)；根据传入的类的权限定名返回一个 Class 对象。**该方法在将 Class 文件加载到内存的同时，会执行类的初始化。**如：

Class.forName("com.atguigu.java.HelloWorld");

- ClassLoader.loadClass() 这是一个实例方法，需要一个 ClassLoader 对象来调用该方法。该方法将 Class 文件加载到内存时，并不会执行类的初始化，直到这个类第一次使用时才进行初始化。该方法因为需要得到一个 ClassLoader 对象，所以可以根据需要指定使用哪个类加载器，如：ClassLoader c1 = ...; c1.loadClass("com.atguigu.java.HelloWorld");

第五节　双亲委派模型

1. 定义与本质

类加载器用来把类加载到 Java 虚拟机中。从 JDK 1.2 版本开始，类的加载

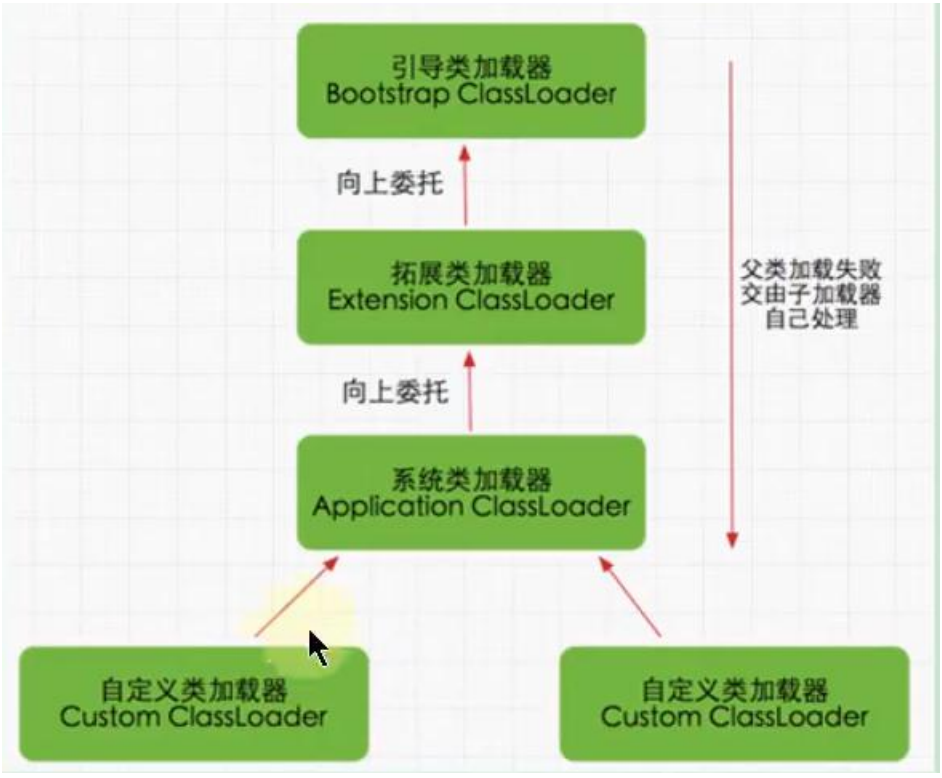
过程采用双亲委派机制，这种机制能更好地保证 Java 平台的安全

1．定义

如果一个类加载器在接到加载类的请求时，它首先不会自己尝试去加载这个类，而是把这个请求任务委托给父类加载器去完成，依次递归，如果父类加载器可以完成类加载任务，就成功返回。只有父类加载器无法完成此加载任务时，才自己去加载

2．本质

规定了类加载的顺序是：引导类加载器先加载，若加载不到，由扩展类加载器加载，若还加载不到，才会由系统类加载器或自定义的类加载器进行加载



2. 优势与劣势

1．双亲委派机制优势

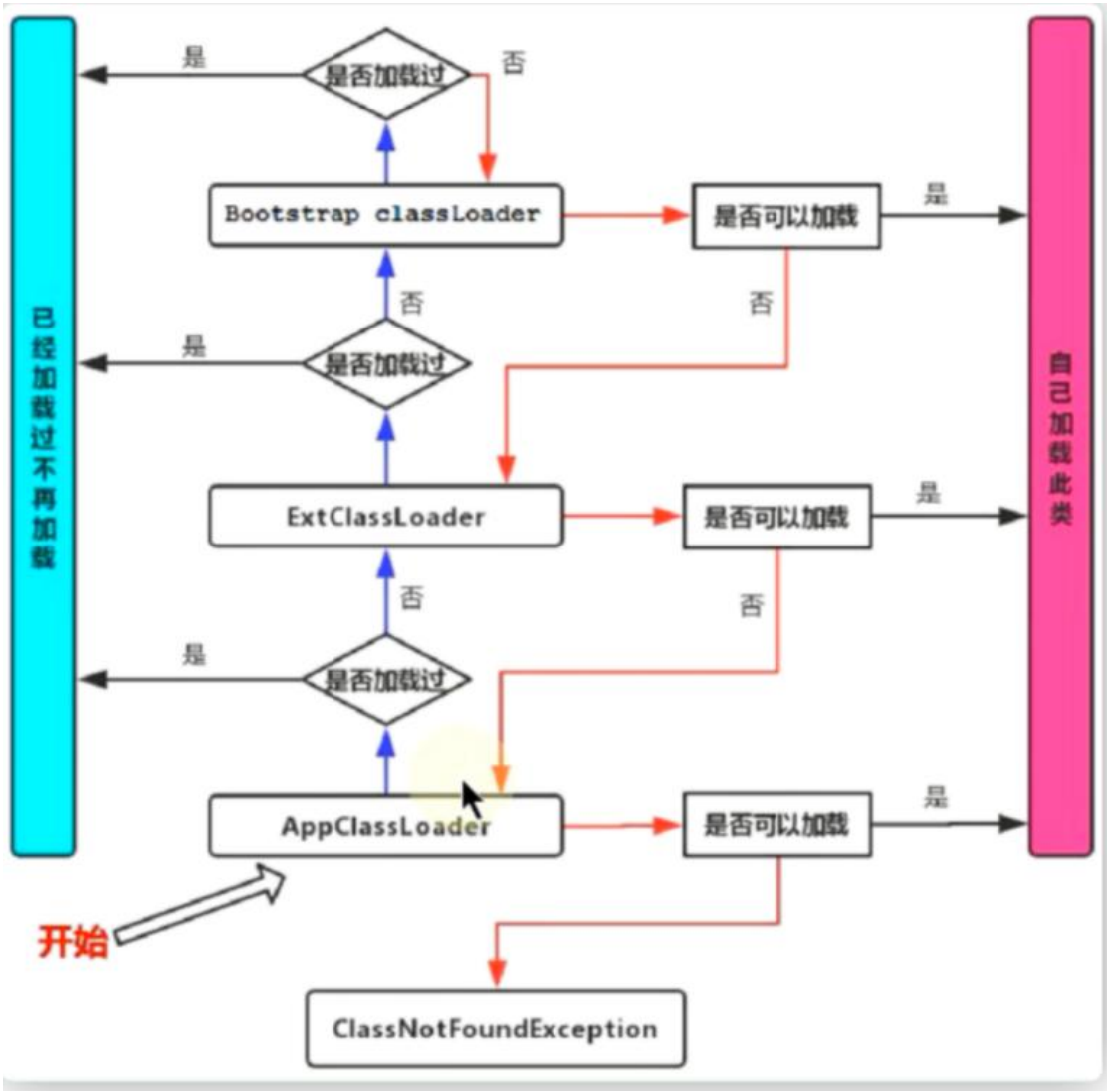
- 避免类的重复加载，确保一个类的全局唯一性

Java 类随着它的类加载器一起具备了一种带有优先级的层级关系，通过这种层级关系可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子 ClassLoader 再加载一次

- 保护程序安全，防止核心 API 被随意篡改

2．代码支持

双亲委派机制在 java.lang.ClassLoader.loadClass(String, boolean) 接口中体现。该接口的逻辑如下：



先在当前加载器的缓存中查找有无目标类，如果有，直接返回

- 判断当前加载器的父加载器是否为空，如果不为空，则调用 parent.loadClass(name, false) 接口进行加载
- 反之，如果当前加载器的父类加载器为空，则调用 findBootstrapClassOrNull(name) 接口，让引导类加载器进行加载
- 如果通过以上 3 条路径都没能成功加载，则调用 findClass(name) 接口进行加载。该接口最终会调用 java.lang.ClassLoader 接口的 defineClass 系列的 native 接口加载目标 Java 类

双亲委派的模型就隐藏在第 2 和第 3 步中

3．举例

假设当前加载的是 java.lang.Object 这个类，很显然，该类属于 JDK 中核心的不能再核心的一个类，因此一定只能由引导类加载器进行加载。当 JVM 准备加载 java.lang.Object 时，JVM 默认会使用系统类加载器去加载，按照上面 5 步加载的逻辑，在第 1 步从系统类的缓存中肯定查找不到该类，于是进入第 2 步。由于从系统类加载器的父类加载器是扩展类加载器，于是扩展类加载器继续从第 1 步开始重复。由于扩展类加载器的缓存中也一定查找不到该类，因此进入第 2 步。扩展类的父加载器是 null，因此系统调用 findClass(String)，最终通过引导类加载器进行加载

4．思考

如果在自定义的类加载器中重写 java.lang.ClassLoader.loadClass(String) 或 java.lang.ClassLoader.loadClass(String, boolean) 方法，抹去其中的双亲委派机制，仅保留上面这 4 步中的第 1 步和第 4 步，那么是不是就能够加载核心类库了呢？

这也不行！因为 JDK 还为核心类库提供了一层保护机制。不管是自定义的类加载器，还是系统类加载器抑或扩展类加载器，最终都必须调用 java.lang.ClassLoader.defineClass(String, byte[], int, int, ProtectionDomain) 方法，而该方法会执行 preDefineClass() 接口，该接口中提供了对 JDK 核心类库的保

护

5．双亲委派模式的弊端

检查类是否加载的委派过程是单向的，这个方式虽然从结构上说比较清晰，使各个 ClassLoader 的职责非常明确，但是同时会带来一个问题，即顶层的 ClassLoader 无法访问底层的 ClassLoader 所加载的类

通常情况下，启动类加载器中的类为系统核心类，包括一些重要的系统接口，而在应用类加载器中，为应用类。按照这种模式，**应用类访问系统类自然是没有问题，但是系统类访问应用类就会出现问题。**比如在系统类中提供了一个接口，该接口需要在应用类中得以实现，该接口还绑定一个工厂方法，用于创建该接口的实例，而接口和工厂方法都在启动类加载器中。这时，就会出现该工厂方法无法创建由应用类加载器加载的应用实例的问题

6．结论

由于 Java 虚拟机规范并没有明确要求类加载器的加载机制一定要使用双亲委派模型，只是建议采用这种方式而已。比如 Tomcat 中，类加载器所采用的加载机制就和传统的双亲委派模型有一定区别，当缺省的类加载器接收到一个类的加载任务时，首先会由它自行加载，当它加载失败时，才会将类的加载任务委派给它的超类加载器去执行，这同时也是 Servlet 规范推荐的一种做法

3. 破坏双亲委派机制

① 破坏双亲委派机制 1

双亲委派模型并不是一个具有强制性约束的模型，而是 Java 设计者推荐给开发者们的类加载器实现方式

在 Java 的世界中大部分的类加载器都遵循这个模型，但也有例外情况，直到 Java 模块化出现为止，双亲委派模型主要出现过 3 次较大规模"被破坏"的情况

第一次破坏双亲委派机制：

双亲委派模型的第一次"被破坏"其实发生在双亲委派模型出现之前——即 JDK 1.2 面世以前的"远古"时代

由于双亲委派模型在 JDK 1.2 之后才被引入，但是类加载器的概念和抽象类 java.lang.ClassLoader 则在 Java 的第一个版本中就已经存在，面对已经存在的用户自定义类加载器的代码，Java 设计者们引入双亲委派模型时不得不做出一些妥协，为了兼容这些已有的代码，无法再以技术手段避免 loadClass() 被子类覆盖的可能性，只能在 JDK 1.2 之后的 java.lang.ClassLoader 中添加一个新的 protected 方法 findClass()，并引导用户编写的类加载逻辑时尽可能去重写这个方法，而不是在 loadClass() 中编写代码。上节我们已经分析过 loadClass() 方法，双亲委派的具体逻辑就实现在这里面，按照 loadClass() 方法的逻辑，如果父类加载失败，会自动调用自己的 findClass() 方法来完成加载，这样既不影响用户按照自己的意愿去加载类，又可以保证新写出来的类加载器是符合双亲委派规则的

② 破坏双亲委派机制 2

第二次破坏双亲委派机制：线程像下文类加载器

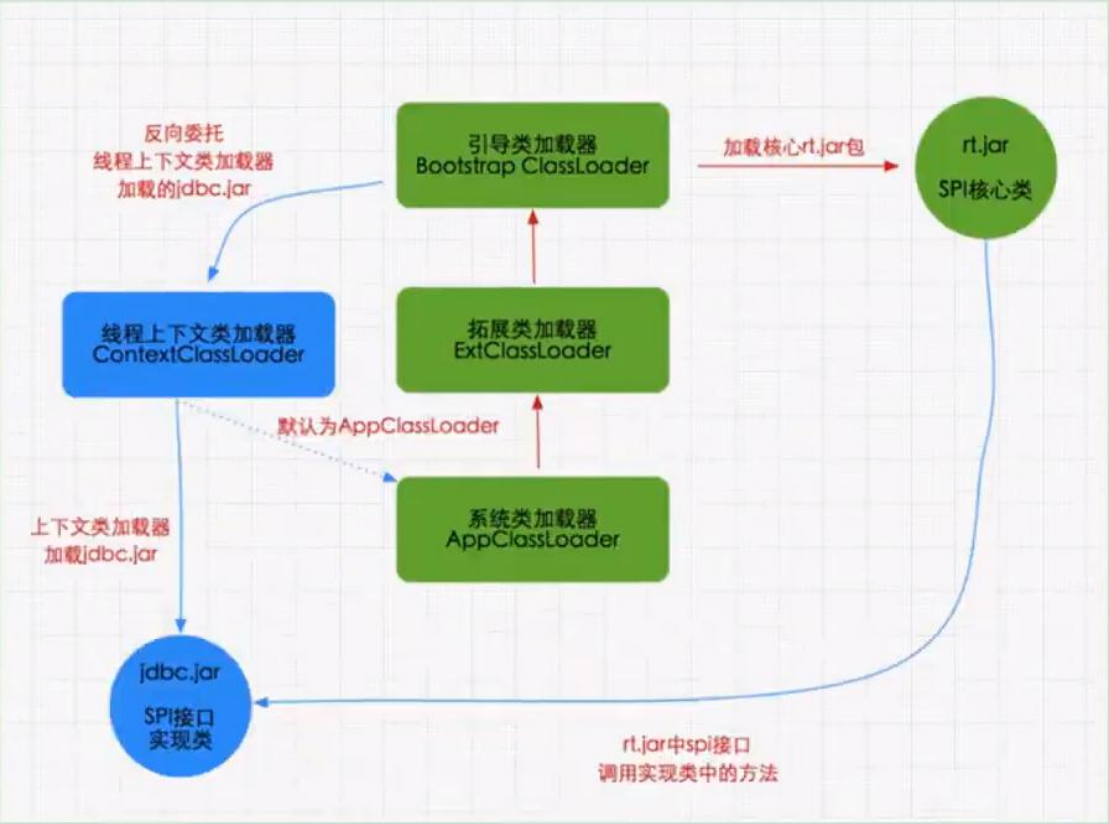
双亲委派模型的第二次"被破坏"是由这个模型自身的缺陷导致的，双亲委派很好地解决了各个类加载器协作时基础类型的一致性问题(越基础的类由越上层的加载器进行加载)，基础类型之所以被称为"基础"，是因为它们总是作为被用户代码继承、调用的 API 存在，但程序设计往往没有绝对不变的完美规则，如果有基础类型又要调用回用户代码，那该怎么办？

这并非是不可能出现的事情，一个典型的例子便是 JNDI 服务，JNDI 现在已经是 Java 的标准服务，它的代码由启动类加载器来完成加载(在 JDK 1.3 时加入到 rt.jar)，肯定属于 Java 中很基础的类型了。但 JNDI 存在的目的就是对资源进行查找和集中管理，它需要调用由其它厂商实现并部署在应用程序的 ClassPath 下的 JNDI 服务提供者接口(Service Provider Interface. SPI) 的代码，现在问题来了，启动类加载器时绝对不可能认识、加载这些代码的，那该怎么办？(SPI：在 Java 平台中，通常把核心类 rt.jar 中提供外部服务、可由应用层自行

实现的接口称为 SPI)

为了解决这个困境，Java 的设计团队只好引入了一个不太优雅的设计：**线程上下文类加载器(Thread Context ClassLoader)。**这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoader() 方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器

有了线程上下文类加载器，程序就可以做一些"舞弊"的事情了。JNDI 服务使用这个线程上下文类加载器去加载所需的 SPI 服务代码。这是一种负累加载器去请求子类加载器完成类加载的行为，这种行为实际上是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型的一般性原则，但也是无可奈何的事情。Java 中涉及 SPI 的加载基本上都采用这种方式来完成，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。不过，当 SPI 的服务提供者多于一个的时候，代码就只能根据具体提供者的类型来硬编码判断，为了消除这种极不优雅的方式，在 JDK 6 时，JDK 提供了 java.util.ServiceLoader 类，以 META-INF/Services 中的配置信息，辅以责任链模式，这才算是给 SPI 的加载提供了一种相对合理的解决方案



默认上下文加载器就是应用类加载器，这样以上下文加载器为中介，使得启动类加载器中的代码也可以访问应用类加载器中的类

③ 破坏双亲委派机制 3

第三次破坏双亲委派机制：

双亲委派模型的第三次"被破坏"是由于用户对程序动态性的追求而导致的。如：代码热替换(Hot Swap)、**模块热部署(Hot Deployment)**等

IBM 公司主导的 JSR-291(即 OSGI R4.2)实现模块化热部署的关键是它自定义的类加载器机制的实现，每个程序模块(OSGI 中称为 Bundle)都有一个自己的类加载器，当需要更换一个 Bundle 时，就把 Bundle 连同类加载器一起换掉以实现代码的热替换。在 OSGI 环境下，类加载器不再双亲委派模型推荐的树状结构，而是进一步发展为更加复杂的网状结构

当收到类加载请求时，OSGI 将按照下面的顺序进行类搜索：

1) 将以 java. 开头的类，委派给父类加载器加载*

2) 否则，将委派列表名单内的类，委派给父类加载器加载

3) 否则，将 Import 列表中的类，委派给 Export 这个类的 Bundle 的类加载器加载

4) 否则，查找当前 Bundle 的 ClassPath，使用自己的类加载器加载

5) 否则，查找类是否在自己的 Fragment Bundle 中，如果在，则委派给 Fragment Bundle 的类加载器加载

6) 否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器加载

7) 否则，类查找失败

8) 说明：只有开头两点仍然符合双亲委派模型的原则，其余的类查找都是在平级的类加载器中进行的

小结：

这里，我们使用了"被破坏"这个词来形容上述不符合双亲委派模型原则的行为，但这里"被破坏"并不一定是带有贬义的。只要有明确的目的和充分的理由，突破旧有原则无疑是一种创新

正如：OSGI 中的类加载器的设计不符合传统的双亲委派的类加载器架构，且业界对其为了实现热部署而带来的额外的高复杂度还存在不少争议，但对这方面有了解的技术人员基本还是能达成一个共识，认为 OSGI 中对类加载器的运用是值得学习的，完全弄懂了 OSGI 的实现，就算是掌握了类加载器的精髓

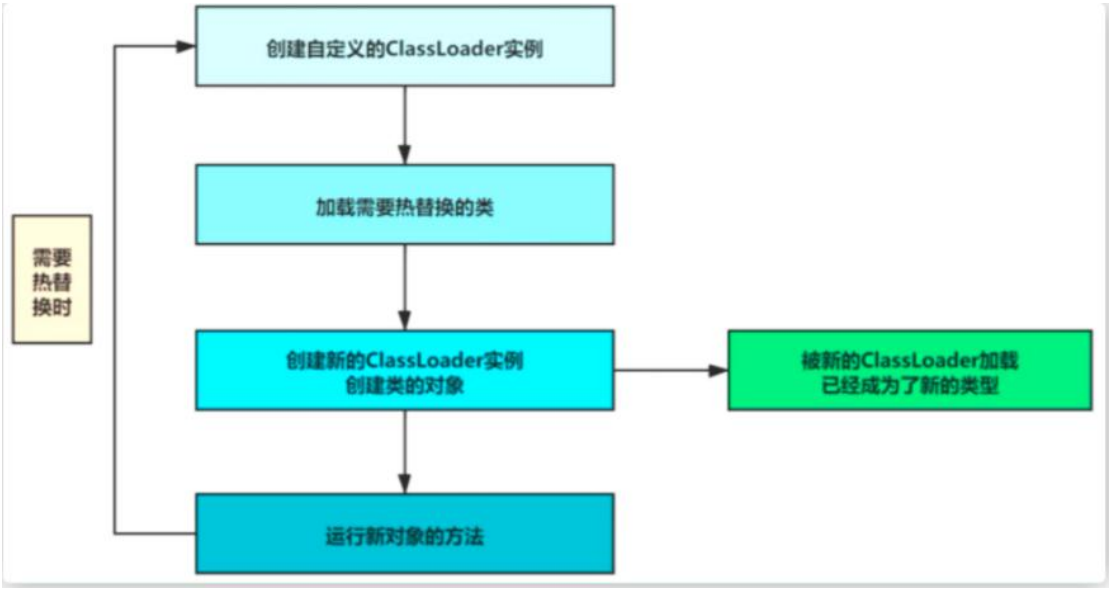
4. 热替换的实现

热替换是指在程序运行过程中，不停止服务，只通过替换程序文件来修改程序的行为。热替换的关键需求在于服务不能中断，修改必须立即表现正在运行的系统之中。基本上大部分脚本语言都是天生支持热替换的，比如：PHP，只要替换了 PHP 源文件，这种改动就会立即生效，而无需重启 Web 服务器

但对 Java 来说，热替换并非天生就支持，如果一个类已经加载到系统中，通过修改类文件，并无法让系统再来加载并重定义这个类。因此，在 Java 中实现这一功能的一个可行的方法就是灵活运用 ClassLoader

注意：由不同 ClassLoader 加载的同名类属于不同的类型，不能相互转换和兼容。即两个不同的 ClassLoader 加载同一个类，在虚拟机内部，会认为这 2 个类是完全不同的

根据这个特点，可以用来模拟热替换的实现，基本思路如下图所示：



第六节　沙箱安全机制

- 保护程序安全
- 保护 Java 原生的 JDK 代码

Java 安全模型的核心就是 Java 沙箱(Sandbox)，什么是沙箱？沙箱就是一个限制程序运行的环境

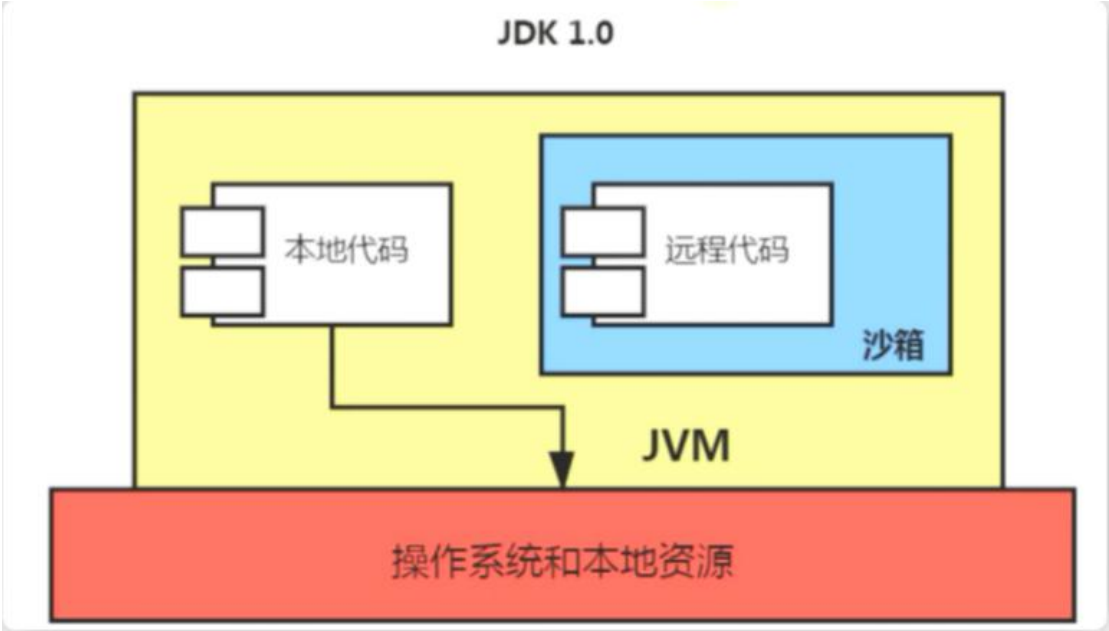
沙箱机制就是将 Java 代码**限定在虚拟机(JVM)特定的运行范围中，并且严格限制代码对本地系统资源访问。**通过这样的措施来保证对代码的有限隔离，防止对本地系统造成破坏

沙箱主要限制系统资源访问，那系统资源包括什么？CPU、内存、文件系统、网络。不同级别的沙箱对这些资源访问的限制也可以不一样

所有的 Java 程序运行都可以指定沙箱，可以定制安全策略

1. JDK 1.0 时期

在 Java 中将执行程序分成本地代码和远程代码两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。对于授信的本地代码，可以访问一切本地资源。而对于非授信的远程代码在早期的 Java 实现中，安全依赖于沙箱(Sandbox)机制。如下图所示 JDK 1.0 安全模型

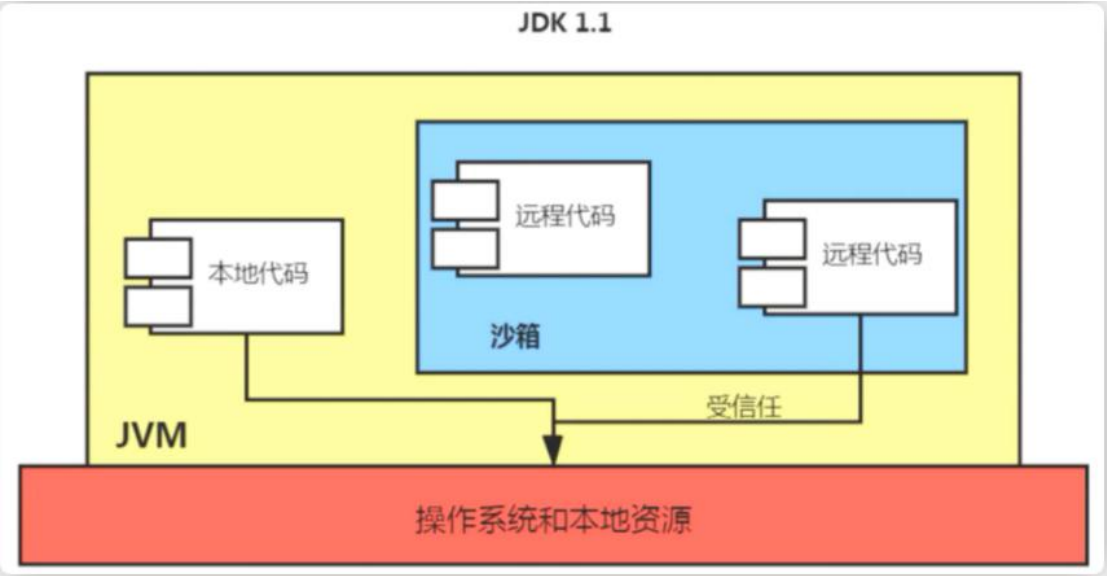


2. JDK 1.1 时期

JDK 1.0 中如此严格的安全机制也给程序的功能扩展带来障碍，比如当用户希望远程代码访问本地系统的文件时候，就无法实现

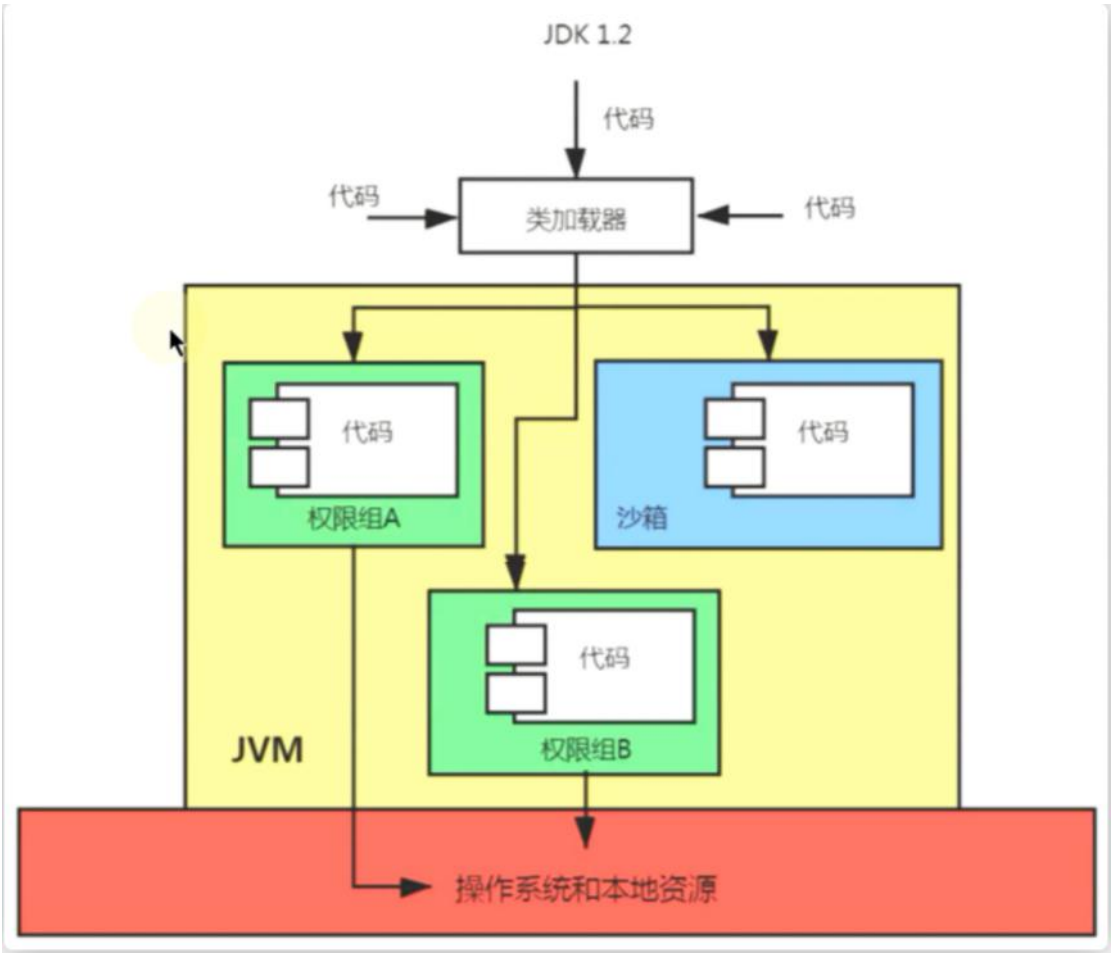
因此在后续的 JDK 1.1 版本中，针对安全机制做了改进，增加了安全策略。允许用户指定代码对本地资源的访问权限

如下图所示 JDK 1.1 安全模型



3. JDK 1.2 时期

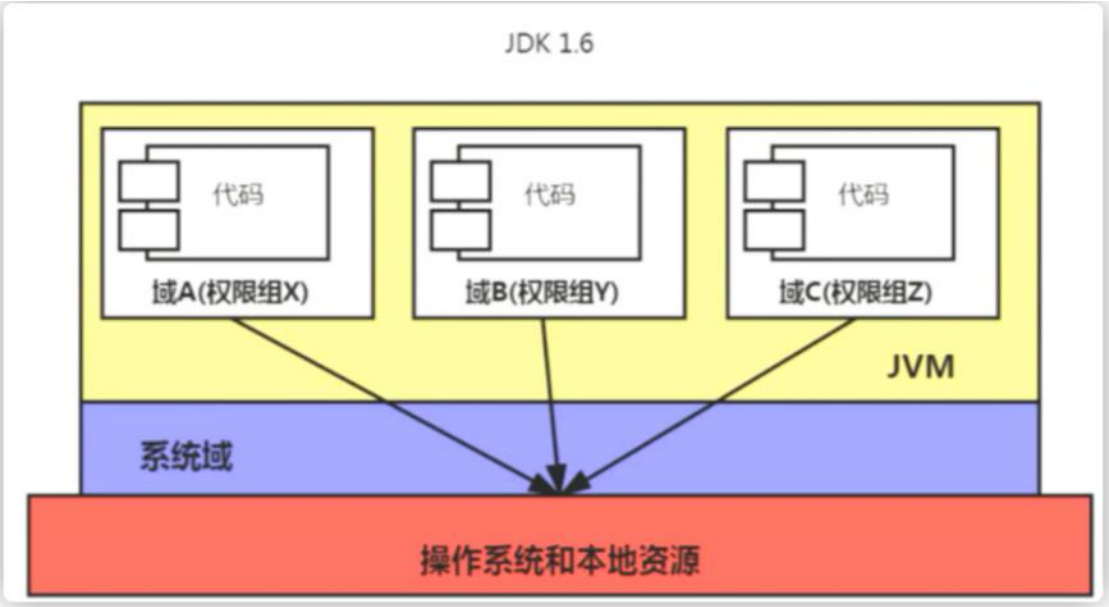
在 JDK 1.2 版本中，再次改进了安全机制，增加了代码签名。不论本地代码或是远程代码，都会按照用户的安全策略设定，由类加载器加载到虚拟机中权限不同的运行空间，来实现差异化的代码执行权限控制。如下图所示 JDK 1.2 安全模型：



4. JDK 1.6 时期

当前最新的安全机制实现，则引入了**域(Domain)**的概念

虚拟机会把所有代码加载到不同的系统域和应用域。系统域部分专门负责与关键资源进行交互，而各个应用域部分则通过系统域的部分代理来对各种需要的资源进行访问。虚拟机中不同的受保护域(Protected Domain)，对应不一样的权限(Permission)。存在于不同域中的类文件就具有了当前域的全部权限，如下图所示，最新的安全模型(JDK 1.6)



第七节　自定义类的加载器

1．为什么要自定义类加载器？

- 隔离加载类

在某些框架内进行中间件与应用的模块隔离，把类加载到不同的环境。比如：阿里内某容器框架通过自定义类加载器确保应用中依赖的 jar 包不会影响到中间件运行时使用的 jar 包。再比如：Tomcat 这类 Web 应用服务器，内部自定义了好几种类加载器，用于隔离同一个 Web 应用服务器上的不同应用程序。(类的仲裁 --> 类冲突)

- 修改类加载的方式

类的加载模型并非强制，除 Bootstrap 外，其他的加载并非一定要引入，或者根据实际情况在某个时间点按需进行动态加载

- 扩展加载源

比如从数据库、网络、甚至是电视机机顶盒进行加载

- 防止源码泄露

Java 代码容易被编译和篡改，可以进行编译加密。那么类加载也需要自定义，还原加密的字节码

2．常见的场景

实现类似进程内隔离，类加载器实际上用作不同的命名空间，以提供类似容器、模块化的效果。例如，两个模块依赖于某个类库的不同版本，如果分别被不同的容器加载，就可以互不干扰。这个方面的集大成者是 Java EE 和 OSGI、JPMS 等框架

应用需要从不同的数据源获取类定义信息，例如网络数据源，而不是本地文件系统。或者是需要自己操纵字节码，动态修改或者生成类型

3．注意

在一般情况下，使用不同的类加载器去加载不同的功能模块，会提高应用程序的安全性。但是，如果涉及 Java 类型转换，则加载器反而容易产生不美好的事情。在做 Java 类型转换时，只有两个类型都是由同一个加载器所加载，才能进行类型转换，否则转换时会发生异常

1. 实现方式

用户通过定制自己的类加载器，这样可以重新定义类的加载规则，以便实现一些自定义的处理逻辑

1. 实现方式

- Java 提供了抽象类 java.lang.ClassLoader，所有用户自定义的类加载器都应该继承 ClassLoader 类
- 在自定义 ClassLoader 的子类时候，我们常见的会有两种做法：
  - 方式一：重写 loadClass() 方法
  - 方式二：重写 findClass() 方法

2. 对比

这两种方法本质上差不多，毕竟 loadClass() 也会调用 findClass()，但是从

逻辑上讲我们最好不要直接修改 loadClass() 的内部逻辑。建议的做法是只在 findClass() 里重写自定义类的加载方法，根据参数指定类的名字，返回对应的 Class 对象的引用

- loadClass() 这个方法是实现双亲委派模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委派模型框架内进行小范围的改动，不破坏原有的稳定结构。同时，也避免了自己重写 loadClass() 方法的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择
- 当编写好自定义类加载器后，便可以在程序中调用 loadClass() 方法来实现类加载操作

3. 说明

- 其父类加载器是系统类加载器
- JVM 中的所有类加载都会使用 java.lang.ClassLoader.loadClass(String) 接口(自定义类加载器并重写 java.lang.ClassLoader.loadClass(String) 接口的除外)，连 JDK 的核心类库也不能例外

第八节 Java 9 新特性

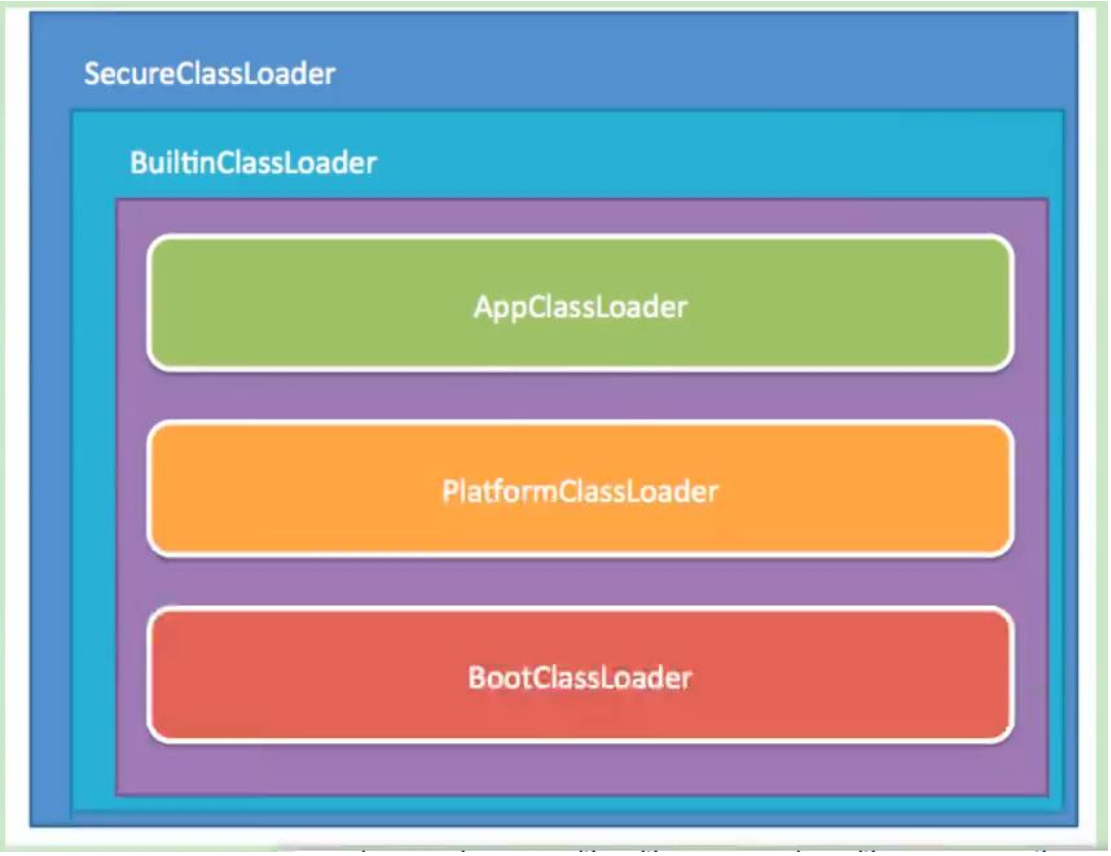
为了保证兼容性，JDK 9 没有从根本上改变三层类加载器架构和双亲委派模型，但为了模块化系统的顺利运行，仍然发生了一些值得被注意的变动

1. 扩展机制被移除，扩展类加载器由于向后兼容性的原因被保留，不过被重命名为平台类加载器(Platform Class Loader)。可以通过 ClassLoader 的新方法 getPlatformClassLoader() 来获取

JDK 9 时基于模块化进行构建(原来的 rt.jar 和 tools.jar 被拆分成数十个 JMOD 文件)，其中的 Java 类库就已天然地满足了可扩展的需求，那自然无需再保留 <JAVA_HOME>\lib\ext 目录，此前使用这个目录或者 java.ext.dirs 系统变量来扩展 JDK 功能的机制已经没有继续存在的价值了

2. 平台类加载器和应用程序类加载器都不再继承自 java.net.URLClassLoader

现在启动类加载器、平台类加载器、应用程序类加载器全都继承于 jdk.internal.loader.BuiltinClassLoader



如果有程序直接依赖了这种继承关系，或者依赖了 URLClassLoader 类的特定方法，那代码很可能会在 JDK 9 及更高版本的 JDK 中崩溃

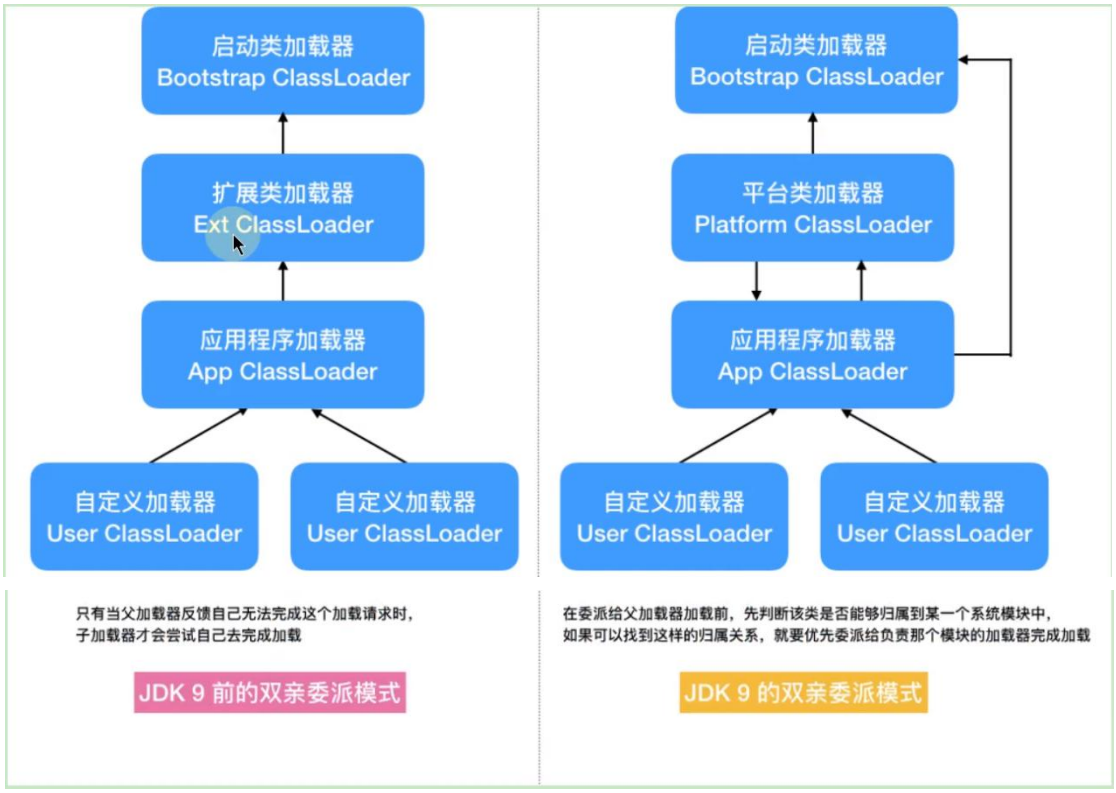
3. 在 Java 9 中，类加载器有了名称。该名称在构造方法中指定，可以通过 getName() 方法来获取。平台类加载器的名称是 Platform，应用类加载器的名称是 App。类加载器的名称在调试与类加载器相关的问题时会非常有用

4. 启动类加载器现在是在 JVM 内部和 Java 类库共同协作实现的类加载器(以前是 C++ 实现)，但为了与之前代码兼容，在获取启动类加载器的场景中仍然会返回 null，而不会得到 BootClassLoader 实例

5. 类加载的委派关系也发生了变动

当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责哪个模块的加载器完成加载

双亲委派模式示意图



附加：

在 Java 模块化系统明确规定了三个类加载器负责各自加载的模块：

- 启动类加载器负责加载的模块

java.base	java.security.sasl
java.datatransfer	java.xml
java.desktop	jdk.httpserver
java.instrument	jdk.internal.vm.ci
java.logging	jdk.management
java.management	jdk.management.agent
java.management.rmi	jdk.naming.rmi
java.naming	jdk.net
java.prefs	jdk.sctp
java.rmi	jdk.unsupported

- 平台类加载器负责加载的模块

java.activation*	jdk.accessibility
java.compiler*	jdk.charsets
java.corba*	jdk.crypto.cryptoki
java.scripting	jdk.crypto.ec
java.se	jdk.dynalink
java.se.se	jdk.incubator.httpclient
java.security.jgss	jdk.internal.vm.compiler*
java.smartcardio	jdk.jsobject
java.sql	jdk.localedata
java.sql.rowset	jdk.naming.dns
java.transaction*	jdk.scripting.nashorn

java.xml.bind*	jdk.security.auth
java.xml.crypto	jdk.security.jgss
java.xml.ws*	jdk.xml.dom
java.xml.ws.annotation*	jdk.zipfs

- 应用程序类加载器负责加载的模块

jdk.aot	jdk.jdeps
jdk.attach	jdk.jdi
jdk.compiler	jdk.jdwp.agent
jdk.editpad	jdk.jlink
jdk.hotspot.agent	jdk.jshell
jdk.internal.ed	jdk.jstatd

字节码指令集与解析

第一节　概述

- Java 字节码对于虚拟机，就好像汇编语言对于计算机，属于基本执行命令
- Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字(称为操作码，Opcode)以及跟随其后的零至多个代表此操作所需参数(称为操作数，Operands)而构成，由于 Java 虚拟机采用面向操作数栈而不是寄存器的结构，所以大多数的指令都不包含操作数，只有一个操作码
- 由于限制了 Java 虚拟机操作码的长度为一个字节(即 0~255)，这意味着指令集的操作码总数不可能超过 256 条
- 官方文档：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html
- 熟悉虚拟机的指令对于动态字节码生成、反编译 Class 文件、 Class 文件修补都有着非常重要的价值。因此，阅读字节码作为了解 Java 虚拟机的基础技能，需要熟练掌握常见指令

1. 执行模型

如果不考虑异常处理的话，那么 Java 虚拟机的解释器可以使用下面这个伪代码当做最基本的执行模型来理解

```
do {
    自动计算 PC 寄存器的值加 1;
    根据 PC 寄存器的指示位置，从字节码流中取出操作码;
```

```
    if(字节码存在操作数) 从字节码流中取出操作数;
    执行操作码所定义的操作;
}while(字节码长度>0)
```

2. 字节码与数据类型

在 Java 虚拟机的指令集中，大多数的指令都包含了其操作所对应的数据类型信息。例如，iload 指令用于从局部变量表中加载 int 类型的数据到操作数栈中，而 fload 指令加载的则是 float 类型的数据

对于大部分与数据类型相关的字节码指令，它们的操作码助记符中都有特殊的字符来表明专门为哪种数据类型服务：

- i 代表对 int 类型的数据操作
- l 代表 long
- s 代表 short
- b 代表 byte
- c 代表 char
- f 代表 float
- d 代表 double

也有一些指令的助记符中没有明确地指明操作类型的字母，如 arraylength 指令，它没有代表数据类型的特殊字符，但操作数永远只能是一个数组类型的对象

还有另一些指令，如无条件跳转指令 goto 则是与数据类型无关的

大部分的指令都没有支持整数类型 byte、char 和 short，甚至没有任何指令支持 boolean 类型。编译器会在编译器或运行期将 byte 和 short 类型的数据带符号扩展(Sign-Extend)为相应的 int 类型数据，将 boolean 和 char 类型数据零

位扩展(Zero-Extend)为相应的 int 类型数据。与之类似，在处理 boolean、byte、short 和 char 类型的数组时，也会转换为使用对应的 init 类型的字节码指令来处理。因此，大多数对于 boolean、byte、short 和 char 类型数据的操作，实际上都是使用相应的 int 类型作为运算类型

3. 指令分类

由于完全介绍和学习这些指令需要花费大量时间，为了让大家能够更快地熟悉和了解这些基本指令，这里将 JVM 中的字节码指令集按用途大致分成 9 类：

- 加载与存储指令
- 算术指令
- 类型转换指令
- 对象的创建与访问指令
- 方法调用与返回指令
- 操作数栈管理指令
- 比较控制指令
- 异常处理指令
- 同步控制指令

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、系统调用等中取得数据，这些数据(可能是值，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、系统调用等操作

第二节　加载与存储指令

- 作用

加载和存储指令用于将数据从栈帧的局部变量表和操作数栈之间来回传递

- 常用指令
  - 「局部变量压栈指令」将一个局部变量加载到操作数栈：xload、xload_<n>(其中 x 为 i、l、f、d、a，n 为 0 到 3)；xaload、xaload<n>(其 x 为 i、l、f、d、a、b、c、s，n 为 0 到 3)
  - 「常量入栈指令」将一个常量加载到操作数栈：bipush、sipush、ldc、ldc_w、ldc2_w、aconst_null、iconst_m1、iconst_<i>、iconst_<l>、fconst_<f>、dconst_<d>
  - 「出栈装入局部变量表指令」将一个数值从操作数栈存储到局部变量表：xstore、xstore_<n>(其中 x 为 i、l、f、d、a，n 为 0 到 3); xastore(其中 x 为 i、l、f、d、a、b、c、s)
  - 扩充局部变量表的访问索引的指令：wide

上面所列举的指令助记符中，有一部分是以尖括号结尾的(例如 iload_<n>)。这些指令助记符实际上代表了一组指令(例如 iload_<n>代表了 iload_0、iload_1、iload_2 和 iload_3 这几个指令)。这几组指令都是某个带有一个操作数的通用指令(例如 iload)的特殊形式，对于这若干组特殊指令来说，它们表面上没有操作数，不需要进行取操作数的动作，但操作数都隐含在指令中

除此之外，它们的语义与原生的通用指令完全一致(例如 iload_0 的语义与操作数为 0 时的 iload 指令语义完全一致)。在尖括号之间的字母指定了指令隐含操作数的数据类型，<n>代表非负的整数，<i>代表是 int 类型数据，<l>代表 long 类型，<f>代表 float 类型，<d>代表 double 类型

操作 byte、char、short 和 boolean 类型数据时，经常用 int 类型的指令来表示

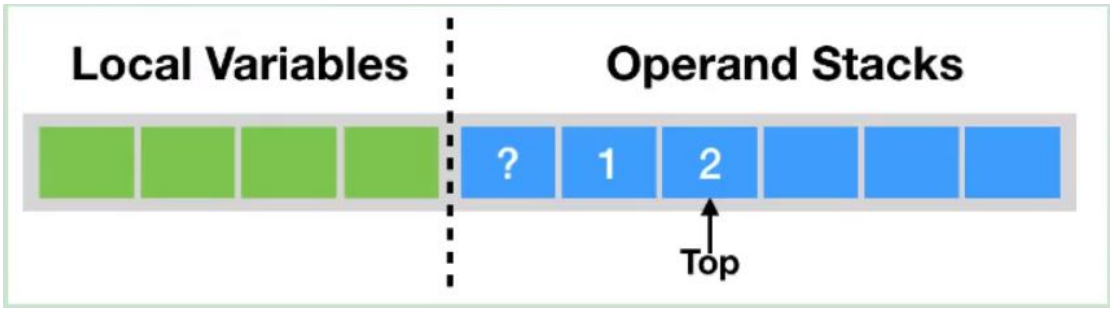
1. 复习：再谈操作数栈与局部变量表

1. 操作数栈(Operand Stacks)

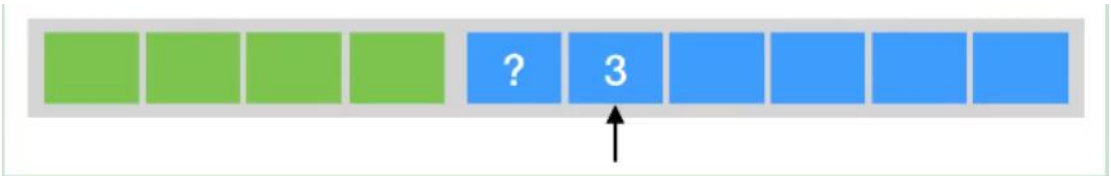
我们知道，Java 字节码是 Java 虚拟机所使用的指令集。因此，它与 Java 虚拟机基于栈的计算模型是密不可分的

在解释执行过程中，每当为 Java 方法分配栈帧时，Java 虚拟机往往需要开辟一块额外的空间作为操作数栈，来存放计算的操作数以及返回结果

具体来说便是：执行每一条指令之前，Java 虚拟机要求该指令的操作数已被压入操作数栈中。在执行指令时，Java 虚拟机会将该指令所需的操作数弹出，并且将指令的结果重新压入栈中



以加法指令 iadd 为例。假设在执行该指令之前，栈顶的两个元素分别为 int 值 1 和 int 值 2，那么 iadd 指令将弹出这两个 int，并将求得的和 int 值为 3 压入栈中



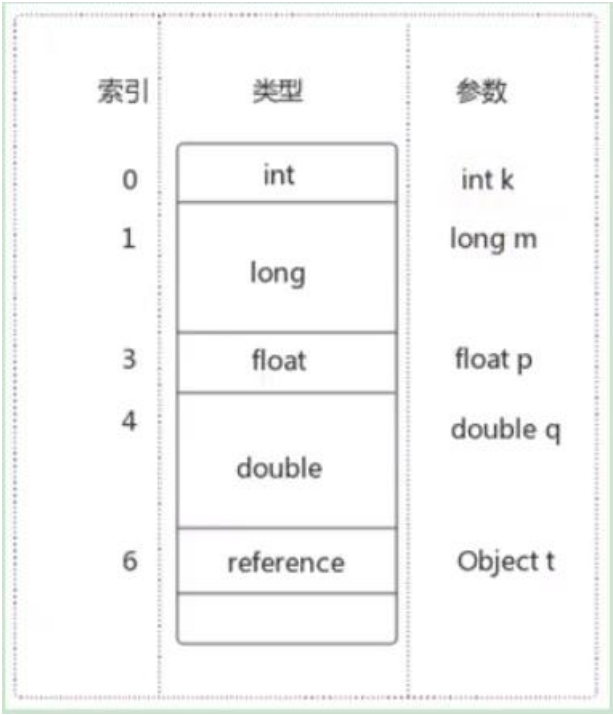
由于 iadd 指令只消耗栈顶的两个元素，因此，对于离栈顶距离为 2 的元素，即图中的问号，iadd 指令并不关心它是否存在，更加不会对其进行修改

2．局部变量表(Local Variables)

Java 方法栈帧的另外一个重要组成部分则是局部变量区，字节码程序可以将计算的结果缓存在局部变量区之中

实际上，Java 虚拟机将局部变量区当成一个数组，依次存放 this 指针(仅非静态方法)，所传入的参数，以及字节码中的局部变量。

和操作数栈一样，long 类型以及 double 类型的值将占据两个单元，其余类型仅占据一个单元



举例：

```
public void foo(long l, float f) {
  {
    int i = 0;
  }
  {
```

```
    String s = "Hello, World"
  }
}
```

在栈帧中，与性能调优关系最为密切的部分就是局部变量表。局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收

在方法执行时，虚拟机使用局部变量表完成方法的传递

2. 局部变量压栈指令

局部变量压栈指令将给定的局部变量表中的数据压入操作数栈

这类指令大体可以分为：

- xload_<n>(x 为 i、l、f、d、a，n 为 0 到 3)
- xload(x 为 i、l、f、d、a)

说明：在这里，x 的取值表示数据类型

指令 xload_n 表示将第 n 个局部变量压入操作数栈，比如 iload_1、fload_0、aload_0 等指令。其中 aload_n 表示将一个对象引用压栈

指令 xload 通过指定参数的形式，把局部变量压入操作数栈，当使用这个命令时，表示局部变量的数量可能超过了 4 个，比如指令 iload、fload 等

3. 常量入栈指令

常量入栈指令的功能是将常数压入操作数栈，根据数据类型和入栈内容的不

同，又可以分为 const 系列、push 系列和 ldc 指令

指令 const 系列：用于对特定的常量入栈，入栈的常量隐含在指令本身里。指令有：iconst_<i>(i 从-1 到 5)、lconst_<l>(l 从 0 到 1)、fconst_<f>(f 从 0 到 2)、dconst_<d>(d 从 0 到 1)、aconst_null

比如：

- iconst_m1 将-1 压入操作数栈
- iconst_x(x 为 0 到 5)将 x 压入栈
- lconst_0、lconst_1 分别将长整数 0 和 1 压入栈
- fconst_0、fconst_1、fconst_2 分别将浮点数 0、1、2 压入栈
- dconst_0 和 dconst_1 分别将 double 型 0 和 1 压入栈
- aconst_null 将 null 压入操作数栈

从指令的命名上不难找出规律，指令助记符的第一个字符总是喜欢表示数据类型，i 表示整数，l 表示长整型，f 表示浮点数，d 表示双精度浮点，习惯上用 a 表示对象引用。如果指令隐含操作的参数，会以下划线形式给出

指令 push 系列：主要包括 bipush 和 sipush，它们的区别在于接受数据类型的不同，bipush 接收 8 位整数作为参数，sipush 接收 16 位整数，它们都将参数压入栈

指令 ldc 系列：如果以上指令都不能满足需求，那么可以使用万能的 ldc 指令，它可以接收一个 8 位的参数，该参数指向常量池中的 int、float 或者 String 的索引，将指定的内容压入堆栈

类似的还有 ldc_w，它接收两个 8 位参数，能支持的索引范围大于 ldc

如果要压入的元素是 long 或者 double 类型的，则使用 ldc2_w 指令，使用方式都是类似的

总结如下：

类型	常数指令	范围
int（boolean, byte, char, short)	iconst	[-1, 5]
	bipush	[-128, 127]
	sipush	[-32768, 32767]
	ldc	any int value
long	lconst	0, 1
	ldc	any long value
float	fconst	0, 1, 2
	ldc	any float value
double	dconst	0, 1
	ldc	any double value
reference	aconst	null
	ldc	String literal, Class literal

4. 出栈装入局部变量表指令

出栈装入局部变量表指令用于将操作数栈中栈顶元素弹出后，装入局部变量表的指定位置，用于给局部变量赋值

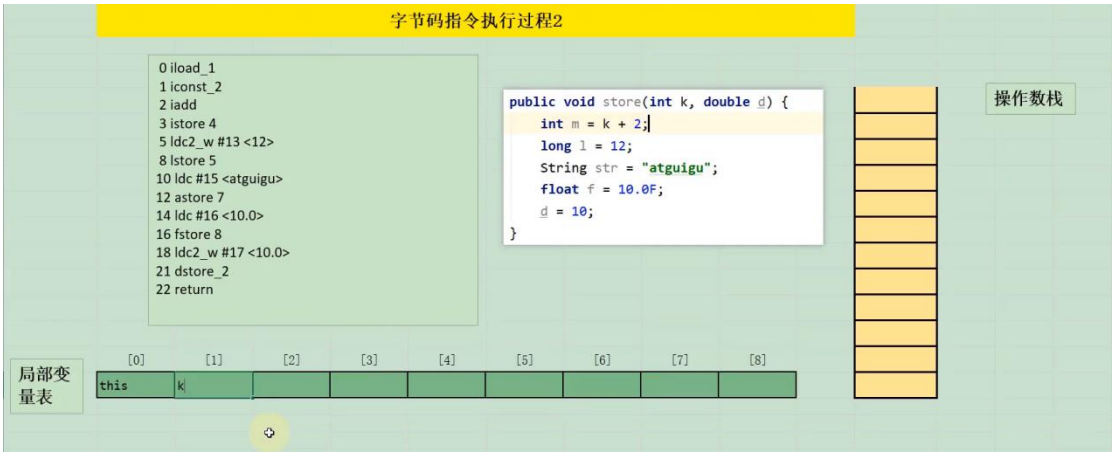
这类指令主要以 store 的形式存在，比如 xstore (x 为 i、l、f、d、a)、xstore_n(x 为 i、l、f、d、a，n 为 0 至 3)和 pasture(x 为 i、l、f、d、a、b、c、s)

- 其中，指令 istore_n 将从操作数栈中弹出一个整数，并把它赋值给局部变量 n

- 指令 xstore 由于没有隐含参数信息，故需要提供一个 byte 类型的参数类指定目标局部变量表的位置
- xastore 则专门针对数组操作，以 iastore 为例，它用于给一个 int 数组的给定索引赋值。在 iastore 执行前，操作数栈顶需要以此准备 3 个元素：值、索引、数组引用，iastore 会弹出这 3 个值，并将值赋给数组中指定索引的位置

一般说来，类似像 store 这样的命令需要带一个参数，用来指明将弹出的元素放在局部变量表的第几个位置。但是，为了尽可能压缩指令大小，使用专门的 istore_1 指令表示将弹出的元素放置在局部变量表第 1 个位置。类似的还有 istore_0、istore_2、istore_3，它们分别表示从操作数栈顶弹出一个元素，存放在局部变量表第 0、2、3 个位置

由于局部变量表前几个位置总是非常常用，因此这种做法虽然增加了指令数量，但是可以大大压缩生成的字节码的体积。如果局部变量表很大，需要存储的槽位大于 3，那么可以使用 istore 指令，外加一个参数，用来表示需要存放的槽位位置



第三节　算术指令

1. 作用

算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新压入操作数栈

2. 分类

大体上算术指令可以分为两种：对整型数据进行运算的指令与对浮点型类型数据进行运算的指令

3. byte、short、char 和 boolean 类型说明

在每一大类中，都有针对 Java 虚拟机具体数据类型的专用算术指令。但没有直接支持 byte、short、char 和 boolean 类型的算术指令，对于这些数据的运算，都使用 int 类型的指令来处理。此外，在处理 boolean、byte、short 和 char 类型的数组时，也会转换为使用对应的 int 类型的字节码指令来处理

Java 虚拟机中的实际类型与运算类型

实际类型	运算类型	分类
boolean	int	一
byte	int	一
char	int	一
short	int	一
int	int	一
float	float	一
reference	reference	一
returnAddress	returnAddress	一
long	long	二
double	double	二

4. 运算时的溢出

数据运算可能会导致溢出，例如两个很大的正整数相加，结果可能是一个负数。其实 Java 虚拟机规范并无明确规定过整型数据溢出的具体结果，仅规定了在处理整型数据时，只有除法指令以及求余指令中当出现除数为 0 时会导致虚拟机抛出异常 ArithmeticException

5. 运算模式

向最接近数舍入模式：JVM 要求在进行浮点数计算时，所有的运算结果都必须舍入到适当的精度，非精确结果必须舍入为可被表示的最接近的精确值，如果有两种可表示的形式与该值一样接近，将优先选择最低有效位为零的

向零舍入模式：将浮点数转换为整数时，采用该模式，该模式将在目标数值类型中选择一个最接近但是不大于原值的数字作为最精确的舍入结果

6. NaN 值使用

当一个操作产生溢出时，将会使用有符号的无穷大表示，如果某个操作结果没有明确的数学定义的话，将会使用 NaN 值来表示。而且所有使用 NaN 值作为操作数的算术操作，结果都会返回 NaN

1. 所有算数指令

所有算术指令包括：

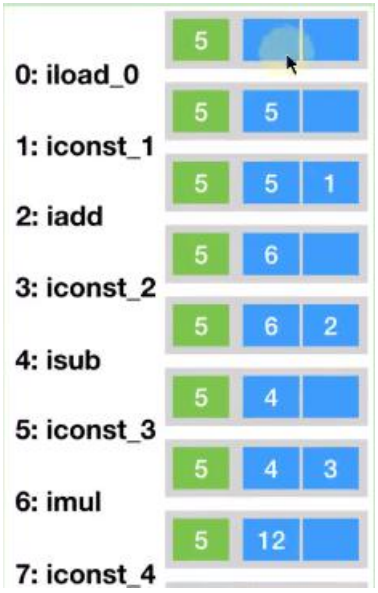
- 加法指令：iadd、ladd、fadd、dadd
- 减法指令：isub、lsub、fsub、dsub
- 乘法指令：imul、lmul、fmul、dmul
- 除法指令：idiv、ldiv、fdiv、ddiv
- 求余指令：irem、lrem、frem、drem(remainder：余数)
- 取反指令：ineg、lneg、fneg、dneg(negation：取反)
- 自增指令：iinc
- 位运算指令，又可分为：
  - 位移指令：ishl、ishr、iushr、lshl、lshr、lushr
  - 按位或指令：ior、lor
  - 按位与指令：iand、land
  - 按位异或指令：ixor、lxor

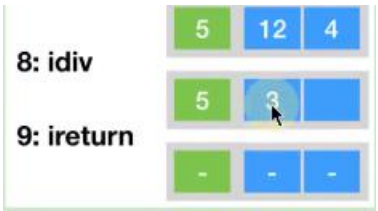
- 比较指令：dcmpg、dcmlp、fcmpg、fcmpl、lcmp

举例：

```
public static int bar(int i) {
    return ((i + 1) - 2) * 3 / 4;
}
```

字节码指令对应的图示：





2. 比较指令的说明

- 比较指令的作用是比较栈顶两个元素的大小，并将比较结果入栈
- 比较指令有：dcmpg、dcmpl、fcmpg、fcmpl、lcmp
  - 与前面讲解的指令类似，首字符 d 表示 double 类型，f 表示 float，l 表示 long
- 对于 double 和 float 类型的数字，由于 NaN 的存在，各有两个版本的比较指令，以 float 为例，有 fcmpg 和 fcmpl 两个指令，它们的区别在于在数字比较时，若遇到 NaN 值，处理结果不同
- 指令 dcmpl 和 dcmpg 也是类似的，根据其命名可以推测其含义，在此不再赘述
- 指令 lcmp 针对 long 型整数，由于 long 型整数没有 NaN 值，故无需准备两套指令

  举例：

  指令 fcmpg 和 fcmpl 都从栈中弹出两个操作数，并将它们做比较，设栈顶的元素为 v2，栈顶顺位第 2 位元素为 v1，若 v1 = v2，则压入 0；若 v1 > v2 则压入 1；若 v1 < v2 则压入-1

两个指令的不同之处在于，如果遇到 NaN 值，fcmpg 会压入 1，而 fcmpl 会压入-1

数值类型的数据才可以谈大小，boolean、引用数据类型不能比较大小

第四节　类型转换指令

类型转换指令说明：

- 类型转换指令可以将两种不同的数值类型进行相互转换
- 这些转换操作一般用于实现用户代码中的显式类型转换操作，或者用来处理

字节码指令集中数据类型相关指令无法与数据类型一一对应的问题

1. 宽化类型转换

1) 转换规则

Java 虚拟机直接支持以下数值的宽化类型转换(Widening Numeric Conversion，小范围类型向大范围类型的安全转换)。也就是说，并不需要指令执行，包括：

- 从 int 类型到 long、float 或者 double 类型，对应的指令为：i2l、i2f、i2d
- 从 long 类型到 float、double 类型。对应的指令为：l2f、l2d
- 从 flaot 类型到 double 类型。对应的指令为：f2d

简化为：int --> long --> float --> double

2) 精度损失问题

① 宽化类型转换是不会因为超过目标类型最大值而丢失信息的，例如，从 int 转换到 long，或者从 int 转换到 double，都不会丢失任何信息，转换前后的值是精确相等的

② 从 int、long 类型数值转换到 float，或者 long 类型树脂转换到 double 时，将可能发生丢失精度——可能丢失掉几个最低有效位上的值，转换后的浮点数值是根据 IEEE754 最接近舍入模式所得到的正确整数数值。尽管宽化类型转换实际上是可能发生精度丢失的，但是这种转换永远不会导致 Java 虚拟机抛出运行时异常

③ 从 byte、char 和 short 类型到 int 类型的宽化类型转换实际上是不存在的，对于 byte 类型转换为 int，虚拟机并没有做实质性的转化处理，知识简单地通过操作数栈交换了两个数据。而 byte 转为 long 时，使用的是 i2l，可以看到在内部 byte 在这里已经等同于 int 类型处理，类似的还有 short 类型，这种处理方式有两个特点：

a. 一方面可以减少实际的数据类型，如果为 short 和 byte 都准备一套指

令，那么指令的数量就会大增，而虚拟机目前的设计上，只愿意使用一个字节表示指令，因此指令总数不能超过 256 个，为了节省指令资源，将 short 和 byte 当作 int 处理也是情理之中

b. 另一方面，由于局部变量表中的槽位固定为 32 位，无论是 byte 或者 short 存入局部变量表，都会占用 32 位空间。从这个角度来说，也没有必要特意区分这几种数据类型

2. 窄化类型转换

① 转换规则

Java 虚拟机也直接支持以下窄化类型转换：

- 从 int 类型至 byte、short 或者 char 类型。对应的指令有：i2b、i2c、i2s
- 从 long 类型到 int 类型。对应的指令有：l2i
- 从 float 类型到 int 或者 long 类型。对应的指令有：f2i、f2l
- 从 double 类型到 int、long 或者 float 类型。对应的指令有：d2i、d2l、d2f

② 精度损失问题

窄化类型转换可能会导致转换结果具备不同的正负号、不同的数量级，因此，转换过程很可能会导致数值丢失精度

尽管数据类型窄化转换可能会发生上限溢出、下限溢出和精度丢失等情况，但是 Java 虚拟机规范中明确规定数值类型的窄化转换指令永远不可能导致虚拟机抛出运行时异常

③ 补充说明

A. 当一个浮点值窄化转换为整数类型 T(T 限于 int 或 long 类型之一)的时候，将遵循以下转换规则：

a. 如果浮点值是 NaN，那转换结果就是 int 或 long 类型的 0

b. 如果浮点值不是无穷大的话，浮点值使用 IEEE754 的向零舍入模式取整，获得整数值 v，如果 v 在目标类型 T(int 或 long)的表示范围之内，那转换结果就是 v。否则，将根据 v 的符号，转换为 T 所能表示的最大或者最小正数

B. 当一个 double 类型窄化转换为 float 类型时，将遵循以下转换规则：

通过向最接近数舍入模式舍入一个可以使用 float 类型表示的数字。最后结果根据下面这 3 条规则判断：

- 如果转换结果的绝对值太小而无法使用 float 来表示，将返回 float 类型的正负零
- 如果转换结果的绝对值太大而无法使用 float 来表示，将返回 float 类型的正负无穷大
- 对于 double 类型的 NaN 值将按规定转换为 float 类型的 NaN 值

第五节　对象的创建与访问指令

Java 是面向对象的程序设计语言，虚拟机平台从字节码层面就对面向对象做了深层次的支持。有一系列指令专门用于对象操作，可进一步细分为创建指令、字段访问指令、数组操作指令和类型检查指令

1. 创建指令

虽然类实例和数组都是对象，但 Java 虚拟机对类实例和数组的创建与操作使用了不同的字节码指令

1．创建类实例的指令：

创建类实例的指令：new

- 它接收一个操作数，为指向常量池的索引，表示要创建的类型，执行完成后，将对象的引用压入栈

2．创建数组的指令：

创建数组的指令：newarray、anewarray、multianewarray

- newarray：创建基本类型数组
- anewarray：创建引用类型数组
- multianewarray：创建多维数组

上述创建指令可以用于创建对象或者数组，由于对象和数组在 Java 中的广泛使用，这些指令的使用频率也很高

2. 字段访问指令

对象创建后，就可以通过对象访问指令获取对象实例或数组实例中的字段或者数组元素

- 访问类字段(static 字段，或者称为类变量)的指令：getstatic、putstatic
- 访问类实例字段(非 static 字段，或者称为实例变量)的指令：getfield、putfield

举例：

以 getstatic 指令为例，它含有一个操作数，为指向常量池的 Fieldref 索引，它的作用就是获取 Fieldref 指定的对象或者值，并将其压入操作数栈

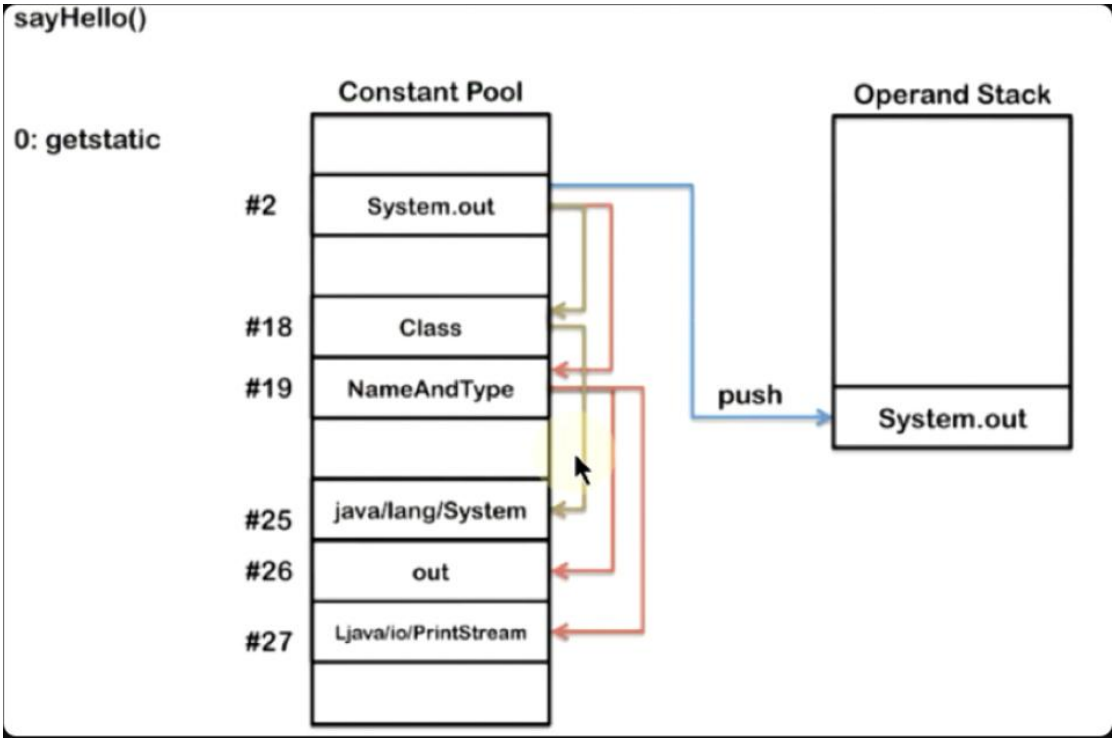
```
public void sayHello() {
   System.out.println("Hello");
}
```

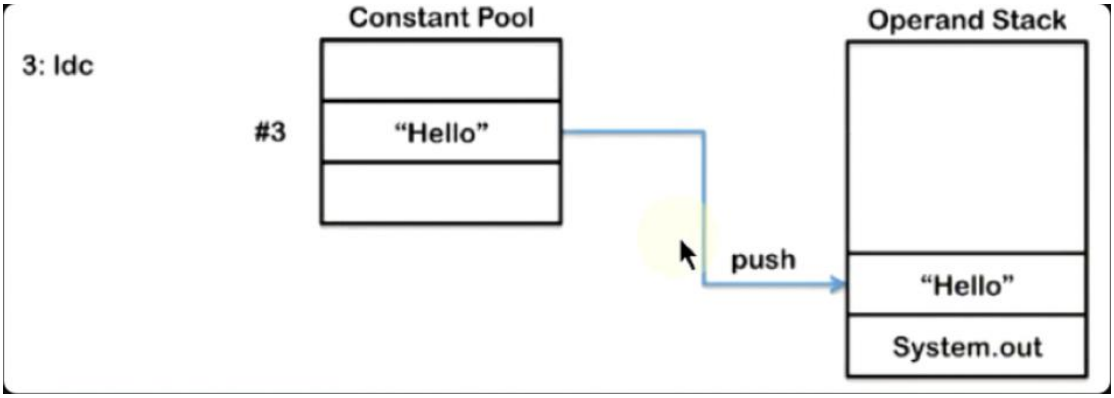
对应的字节码指令：

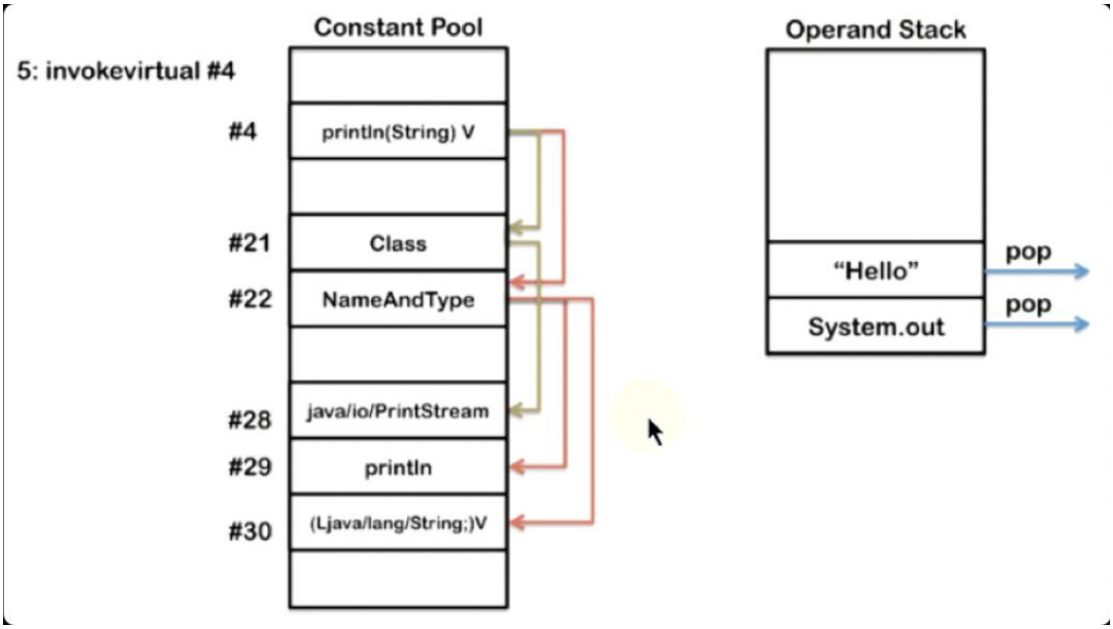
```
0 getstatic #8 <java/lang/System.out>
3 ldc #9 <Hello>
5 invokevirtual #10 <java/io/PrintStream.println>
```

8 return

图示：


sayHello()
0: getstatic
Constant Pool
Operand Stack
#2
System.out
#18
Class
#19
NameAndType
#25
java/lang/System
#26
out
#27
Ljava/io/PrintStream
push
System.out


3: ldc
Constant Pool
Operand Stack
#3
"Hello"
push
"Hello"
System.out



3. 数组操作指令

数组操作指令主要有：xastore 和 xaload 指令。具体为：

- 把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload
- 将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、lastore、fastore、dastore、aastore

即：

数组类型	加载指令	存储指令
byte (boolean)	baload	bastore
char	caload	castore
short	saload	sastore
int	iastore	iastore
long	laload	lastore
float	faload	fastore
double	daload	dastore
reference	aaload	aastore

- 取数组长度的指令：arraylength
  - 该指令弹出栈顶的数组元素，获取数组的长度，将长度压入栈

说明：

- 指令 xaload 表示将数组的元素压栈，比如 saload、caload 分别表示压入 short 数组和 char 数组。指令 xaload 在执行时，要求操作数中栈顶元素为数组索引 i，栈顶顺位第 2 个元素为数组引用 a，该指令会弹出栈顶这两个元素，并将 a[i] 重新压入堆栈
- xastore 则专门针对数组操作，以 iastore 为例，它用于给一个 int 数组的给定索引赋值。在 iastore 执行前，操作数栈顶需要以此准备 3 个元素：值、索引、数组饮用，iastore 会弹出这 3 个值，并将值赋给数组中指定索引的位置

4. 类型检查指令

检查类实例或数组类型的指令：instanceof、checkcast

- 指令 checkcast 用于检查类型强制转换是否可以进行。如果可以进行，那么 checkcast 指令不会改变操作数栈，否则它会抛出 ClassCastException 异常
- 指令 instanceof 用来判断给定对象是否是某一个类的实例，它会将判断结果压入操作数栈

第六节　方法调用与返回指令

1. 方法调用指令

方法调用指令：invokevirtual、invokeinterface、invokespecial、invokestatic、invokedynamic

一下 5 条指令用于方法调用：

- invokevirtual 指令用于调用对象的实例方法，根据对象的实际类型进行分派(虚方法分派)，支持多态。这也是 Java 语言中最常见的方法分派方式
- invokeinterface 指令用于调用接口方法，它会在运行时搜索由特定对象所实现的这个接口方法，并找出适合的方法进行调用
- invokespecial 指令用于调用一些需要特殊处理的实例方法，包括实例初始化方法(构造器)、私有方法和父类方法。这些方法都是静态类型绑定的，不会在调用时进行动态派发
- invokestatic 指令用于调用命名类中的类方法(static 方法)。这是静态绑定的
- invokedynamic 调用动态绑定的方法，这个是 JDK 1.7 后新加入的指令。用于在运行时动态解析出调用点限定符所引用的方法，并执行该方法。前面 4 条调用指令的分派逻辑都固化在 Java 虚拟机内部，而 invokedynamic 指令的分派逻辑是由用户所设定的引导方法决定的

2. 方法返回指令

方法调用结束前，需要进行返回。方法返回指令是根据返回值的类型区分的

- 包括 ireturn(当返回值是 boolean、byte、char、short 和 int 类型时使用)、lreturn、freturn、dreturn 和 areturn
- 另外还有一条 return 指令供声明为 void 的方法、实例初始化方法以及类和接口的类初始化方法使用

返回类型	返回指令
void	return
int (boolean, byte, char, short)	ireturn
long	lreturn
float	freturn
double	dreturn
reference	areturn

举例：

通过 ireturn 指令，将当前函数操作数栈的顶层元素弹出，并将这个元素压入调用者函数的操作数栈中(因为调用者非常关心函数的返回值)，所有在当前函数操作数栈中的其他元素都会被丢弃

如果当前返回的是 synchronized 方法，那么还会执行一个隐含的 monitorexit 指令，退出临界区

最后，会丢弃当前方法的整个帧，恢复调用者的帧，并将控制权转交给调用者

第七节　操作数栈管理指令

如同操作一个普通数据结构中的堆栈那样，JVM 提供的操作数栈管理指令，可以用于直接操作操作数栈的指令

这类指令包括如下内容：

- 将一个或两个元素从栈顶弹出，并且直接废弃：pop、pop2
- 复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、du p_x2、dup2_x2
- 将栈最顶端的两个 Slot 数值位置交换：swap、Java 虚拟机没有提供交换两个 64 位数据类型(long、double)数值的指令
- 指令 nop 是一个非常特殊的指令，它的字节码为 0x00。和汇编语言中的 nop 一样，它表示什么都不做，这条指令一般可用于调试、占位等

这些指令属于通用型，对栈的压入或者弹出无需知名数据类型

说明：

- 不带 _x 的指令是复制栈顶数据并压入栈顶。包括两个指令，dup 和 dup2，dup 的系数代表要复制的 Slot 个数
  - dup 开头的指令用于复制 1 个 Slot 的数据。例如 1 个 int 或 1 个 reference 类型数据
  - dup2 开头的指令用于复制 2 个 Slot 的数据。例如 1 个 long，或 2 个 int，或 1 个 int 加 1 个 float 类型数据
- 带 _x 的指令是复制栈顶数据并插入栈顶以下的某个位置。共有 4 个指令，dup_x1、dup2_x1、dup_x2、dup2_x2。对于带 _x 的复制插入指令，只要将指令的 dup 和 x 的系数相加，结果即为需要插入的位置。因此
  - dup_x1 插入位置：1+1=2，即栈顶 2 个 Slot 下面

- dup_x2 插入位置：1+2=3，即栈顶 3 个 Slot 下面
- dup2_x1 插入位置：2+1=3，即栈顶 3 个 Slot 下面
- dup2_x2 插入位置：2+2=4，即栈顶 4 个 Slot 下面

- pop：将栈顶的 1 个 Slot 数值出栈。例如 1 个 short 类型数值
- pop2：将栈顶的 2 个 Slot 数值出栈。例如 1 个 double 类型数值，或者 2 个 int 类型数值

第八节　控制转移指令

程序流程离不开条件控制，为了支持条件跳转，虚拟机提供了大量字节码指令，大体上可以分为比较指令、条件跳转指令、比较条件跳转指令、多条件分支跳转指令、无条件跳转指令等

1. 条件跳转指令

条件跳转指令通常和比较指令结合使用。在条件跳转指令执行前，一般可以先用比较指令进行栈顶元素的准备，然后进行条件跳转

条件跳转指令有：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull。这些指令都接收两个字节的操作数，用于计算跳转的位置(16 位符号整数作为当前位置的 offset)

它们的统一含义为：弹出栈顶元素，测试它是否满足某一条件，如果满足条件，则跳转到给定位置

具体说明：

ifeq	当栈顶int类型数值等于0时跳转
ifne	当栈顶int类型数值不等于0时跳转
iflt	当栈顶int类型数值小于0时跳转
ifle	当栈顶int类型数值小于等于0时跳转
ifgt	当栈顶int类型数组大于0时跳转
ifge	当栈顶int类型数值大于等于0时跳转
ifnull	为null时跳转
ifnonnull	不为null时跳转

注意：

1) 与前面运算规则一致

- 对于 boolean、byte、char、short 类型的条件分支比较操作，都是使用 int 类型的比较指令完成
- 对于 long、float、double 类型的条件分支比较操作，则会先执行相应类型的比较运算指令，运算指令会返回一个整型值到操作数栈中，随后再执行 int 类型的条件分支比较操作来完成整个分支跳转

2) 由于各类型的比较最终都会转为 int 类型的比较操作，所以 Java 虚拟机提供的 int 类型的条件分支指令是最为丰富和强大的

2. 比较条件跳转指令

比较条件跳转指令类似于比较指令和条件跳转指令的结合体，它将比较和跳转两个步骤合二为一、

这类指令有：if_icmped、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmped 和 if_acmpne

其中指令助记符加上 "if_" 后，以字符 "i" 开头的指令针对 int 型整数操作(也包括 short 和 byte 类型)，以字符 "a" 开头的指令表示对象引用的比较

具体说明：

if_icmpeq	比较栈顶两int类型数值大小，当前者等于后者时跳转
if_icmpne	比较栈顶两int类型数值大小，当前者不等于后者时跳转
if_icmplt	比较栈顶两int类型数值大小，当前者小于后者时跳转
if_icmple	比较栈顶两int类型数值大小，当前者小于等于后者时跳转
if_icmpgt	比较栈顶两int类型数值大小，当前者大于后者时跳转
if_icmpge	比较栈顶两int类型数值大小，当前者大于等于后者时跳转
if_acmpeq	比较栈顶两引用类型数值，当结果相等时跳转
if_acmpne	比较栈顶两引用类型数值，当结果不相等时跳转

这些指令都接收两个字节的操作数作为参数，用于计算跳转的位置。同时在执行指令时，栈顶需要准备两个元素进行比较。指令执行完成后，栈顶的这两个元素被清空，且没有任何数据入栈。如果预设条件成立，则执行跳转，否则，继续执行下一条语句

3. 多条件分支跳转

多条件分支跳转指令是专为 switch-case 语句设计的，主要有 tableswitch 和 lookupswitch

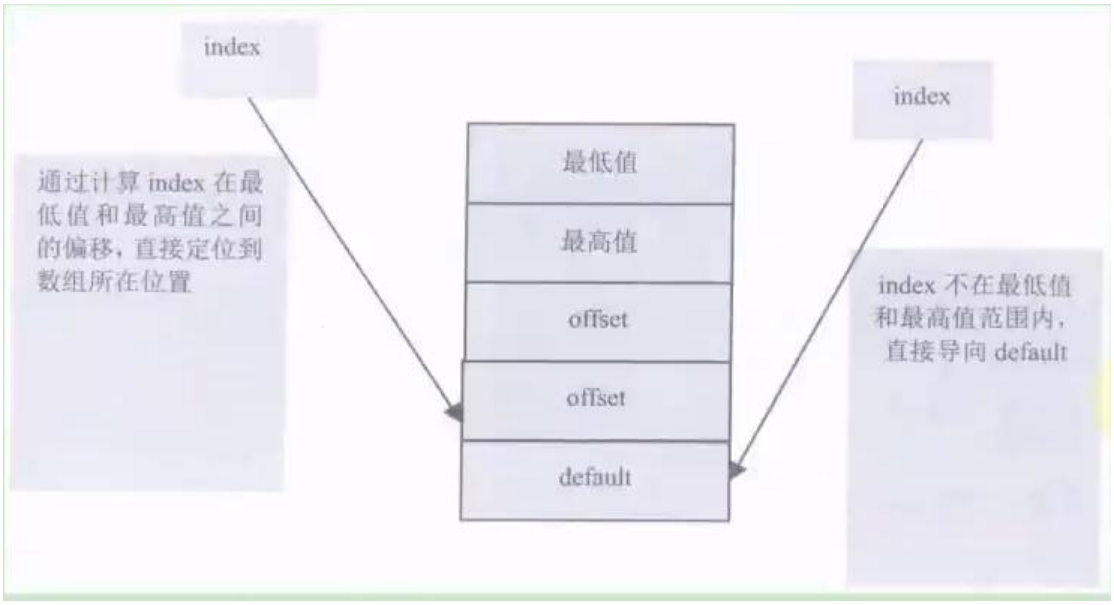
指令名称	描述
tableswitch	用于switch条件跳转，case值连续
lookupswitch	用于switch条件跳转，case值不连续

从助记符上看，两者都是 switch 语句的实现，它们的区别：

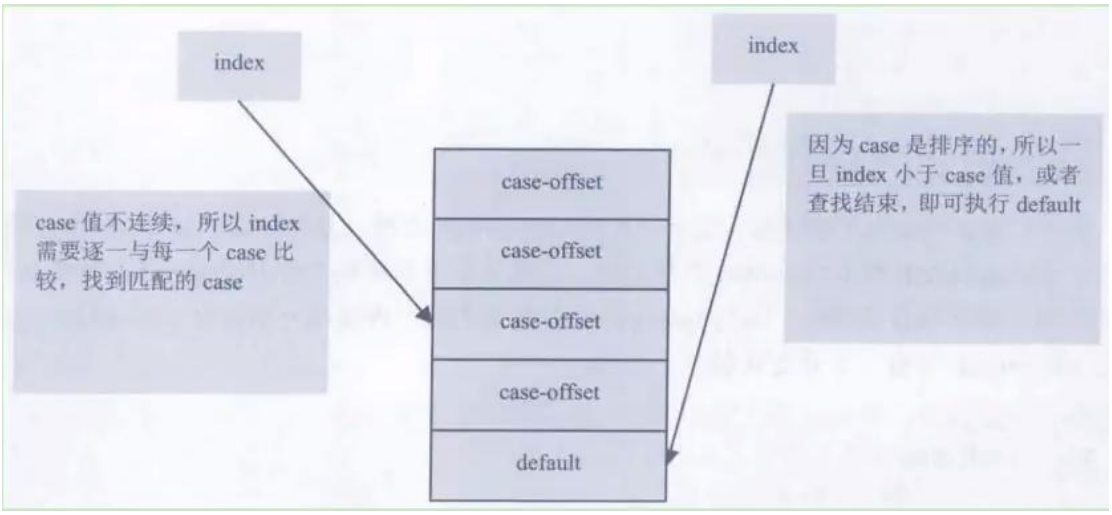
- tableswitch 要求多个条件分支值是连续的，它内部只存放起始值和终止值，以及若干个跳转偏移量，通过给定的操作数 index，可以立即定位到跳转偏移量位置，因此效率比较高
- lookupswitch 内部存放着各个离散的 case-offset 对，每次执行都要搜索全部

的 case-offset 对，找到匹配的 case 值，并根据对应的 offset 计算跳转地址，因此效率较低

指令 tableswitch 的示意图如下图所示。由于 tableswitch 的 case 值是连续的，因此只需要记录最低值和最高值，以及每一项对应的 offset 偏移量，根据给定的 index 值通过简单的计算即可直接定位到 offset



指令 lookupswitch 处理的是离散的 case 值，但是出于效率考虑，将 case-offset 对按照 case 值大小排序，给定 index 时，需要查找与 index 相等的 case，获得其 offset，如果找不到则跳转到 default。指令 lookupswitch 如下图所示



4. 无条件跳转

目前主要的无条件跳转指令为 goto，指令 goto 接收两个字节的操作数，共同组成一个带符号的整数，用于指定指令的偏移量，指令执行的目的就是跳转到偏移量给定的位置处

如果指令偏移量太大，超过双字节的带符号整数的范围，则可以使用指令 goto_w，它和 goto 有相同的作用，但是它接收 4 个字节的操作数，可以表示更大的地址范围

指令 jsr、jsr_w、ret 虽然也是无条件跳转的，但主要用于 try-finally 语句，且已经被虚拟机逐渐废弃，故不在这里介绍这两个指令

指令名称	描述
goto	无条件跳转
goto_w	无条件跳转(宽索引)
jsr	跳转至指定16位offset位置，并将jsr下一条指令地址压入栈顶
jsr_w	跳转至指定32位offer位置，并将jsr_w下一条指令地址压入栈顶
ret	返回至由指定的局部变量所给出的指令位置(一般与jsr、jsr_w联合使用)

第九节　异常处理指令

1. 抛出异常指令

1．athrow 指令

在 Java 程序中显式抛出异常的操作(throw 语句)都是由 athrow 指令来实现的

除了使用 throw 语句显式抛出异常情况之外，JVM 规范还规定了许多运行时一场会在其它 Java 虚拟机指令检测到异常状况时自动抛出。例如，在之前介绍的整数运算时，当除数为零时，虚拟机会在 idiv 或 ldiv 指令中抛出 ArithmeticException 异常

2．注意

正常情况下，操作数栈的压入弹出都是一条条指令完成的。唯一的例外情况是在抛异常时，Java 虚拟机会清除操作数栈上的所有内容，而后将异常实例压入调用者操作数栈上

异常及异常的处理：

过程一：异常对象的生成过程 ---> throw(手动/自动) ---> 指令：athrow

过程二：异常的处理：抓抛模型 try-catch-finally ---> 使用异常表

2. 异常处理与异常表

1．处理异常

在 Java 虚拟机中，处理异常(catch 语句)不是由字节码指令来实现的(早期使用 jsr、ret 指令)，而是采用异常表来完成的

2．异常表

如果一个方法定义了一个 try-catch 或者 try-finally 的异常处理，就会创建一个异常表。它包含了每个异常处理或者 finally 块的信息。异常表保存了每个异常处理信息。比如：

- 起始位置
- 结束位置
- 程序计数器记录的代码处理的偏移地址
- 被捕获的异常类在常量池中的索引

当一个异常被抛出时，JVM 会在当前的方法里寻找一个匹配的处理，如果没有找到，这个方法会强制结束并弹出当前栈帧，并且异常会重新抛给上层调用的方法(在调用方法栈帧)。如果在所有栈帧弹出前仍然没有找到合适的异常处理，这个县城将终止。如果这个异常在最后一个非守护线程里抛出，将会导致 JVM 自己终止，比如这个线程是个 main 线程

不管什么时候抛出异常，如果异常处理最终匹配了所有异常类型，代码就会继续执行。在这种情况下， 如果方法结束后没有抛出异常，仍然执行 finally 块，在 return 前，它直接跳到 finally 块来完成目标

第十节　同步控制指令

Java 虚拟机支持两种同步结构：方法级同步 和 方法内部一段指令序列的同步，这两种同步都是使用 monitor 来支持的

1. 方法级的同步

方法级的同步：是隐式的，即无需通过字节码指令来控制，它实现在方法调用和返回操作之中。虚拟机可以从方法常量池的方法表结构中的 ACC_SYNCHRONIZED 访问标志得知一个方法是否声明为同步方法

当调用方法时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否设置

- 如果设置了，执行线程将先持有同步锁，然后执行方法，最后在方法完成(无论是正常完成还是非正常完成)时释放同步锁

- 在方法执行期间，执行线程持有了同步锁，其它任何线程都无法再获得同一个锁
- 如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的锁将在异常抛到同步方法之外时自动释放

举例：

```
private int i = 0;
public synchronized void add() {
   i++;
}
```

对应字节码：

```
0 aload_0
1 dup
2 getdield #2 <com/atguigu/java1/SynchronizedTest.i>
5 iconst_1
6 iadd
7 putfield #2 <com/atguigu/java1/SynchronizedTest.i>
10 return
```

说明：

这段代码和普通的无同步操作的代码没有什么不同，没有使用 monitorenter 和 monitorexit 进行同步区控制。这是因为，对于同步方法而言，当虚拟机通过方法的访问标识符判断是一个同步方法时，会自动在方法调用前进行加锁，当同步方法执行完毕后，不管方法是正常结束还是有异常抛出，均会由虚拟机释放这个锁。因此，对于同步方法而言，monitorenter 和 monitorexit 指令是隐式存在

的，并未直接出现在字节码中

2. 方法内指定指令序列的同步

同步一段指令集序列：通常是由 Java 中的 synchronized 语句块来表示的。JVM 的指令集有 monitorenter 和 monitorexit 两条指令来支持 synchronized 关键字的语义

当一个线程进入同步代码块时，它使用 monitorenter 指令请求进入。如果当前对象的监视器计数器为 0，则它会被准许进入，若为 1，则判断持有当前监视器的线程是否为自己，如果是，则进入，否则进行等待，知道对象的监视器计数器为 0，才会被允许进入同步块

当线程退出同步块时，需要使用 monitorexit 声明退出。在 Java 虚拟机中，任何对象都有一个监视器与之相关联，用来判断对象是否被锁定，当监视器被持有后，对象处于锁定状态

指令 monitorenter 和 monitorexit 在执行时，都需要在操作数栈顶压入对象，之后 monitorenter 和 monitorexit 的锁定和释放都是针对这个对象的监视器进行的

编译器必须确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都必须执行其对应的 monitorexit 指令，而无论这个方法是正常结束还是异常结束

为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令