

JVM笔记无特殊说明，基于jdk8&14+HotSpot

[零、参考资料 3](#_Toc38373203)

[一、内存结构 4](#_Toc38373204)

[1、程序计数器(Program Counter Register) 4](#_Toc38373205)

[2、Java虚拟机栈(JVM Stack) 4](#_Toc38373206)

[2.2栈帧(Frame) 4](#_Toc38373207)

[2.3局部变量表(Local Variable) 5](#_Toc38373208)

[2.4操作数栈(Operand Stack) 5](#_Toc38373209)

[2.5动态链接(Dynamic Linking) 5](#_Toc38373210)

[2.6方法调用 6](#_Toc38373211)

[2.7 returnAddress数据类型 6](#_Toc38373212)

[2.8 boolean数据类型 6](#_Toc38373213)

[2.9 reference数据类型 6](#_Toc38373214)

[3、本地方法栈(Native Method Stack) 7](#_Toc38373215)

[4、堆(Heap) 8](#_Toc38373216)

[5、方法区(Method Area) 8](#_Toc38373217)

[5.1概述 8](#_Toc38373218)

[5.2运行时常量池(Run-Time Constant Pool) 8](#_Toc38373219)

[6、直接内存(Direct Memory) 8](#_Toc38373220)

[二、编译(Compiling) 9](#_Toc38373221)

[1概述 9](#_Toc38373222)

[2前端编译器 9](#_Toc38373223)

[3即时编译器 9](#_Toc38373224)

[4提前编译器 9](#_Toc38373225)

[5虚拟机指令 9](#_Toc38373226)

[5.1概述 9](#_Toc38373227)

[5.2数据类型和JVM(Types and the Java Virtual Machine) 9](#_Toc38373228)

[5.3加载和存储指令 10](#_Toc38373229)

[5.4算数指令 10](#_Toc38373230)

[5.5类型转换指令 11](#_Toc38373231)

[5.6对象创建与访问指令 12](#_Toc38373232)

[5.7操作数栈管理指令 12](#_Toc38373233)

[5.8控制转移指令 12](#_Toc38373234)

[5.9方法调用和返回指令 12](#_Toc38373235)

[5.10异常处理指令 13](#_Toc38373236)

[5.11同步指令 13](#_Toc38373237)

[5.12 invokedynamic指令 13](#_Toc38373238)

[三、类(Class) 14](#_Toc38373239)

[1概述 14](#_Toc38373240)

[2类结构 14](#_Toc38373241)

[2.1类结构元素 14](#_Toc38373242)

[2.2常量池(constant\_pool) 16](#_Toc38373243)

[3类的加载 22](#_Toc38373244)

[3.1类加载过程 22](#_Toc38373245)

[4模块化 28](#_Toc38373246)

[四、JVM字节码执行 28](#_Toc38373247)

[五、总结 28](#_Toc38373248)

# 零、参考资料

Java虚拟机规范(Java SE 8版)

Java语言规范基于Java SE 8

深入理解Java虚拟机——JVM高级特性与最佳实践

实战Java虚拟机——JVM故障诊断与性能优化

垃圾回收的算法与实现

深入理解计算机系统

Stackoverflow:

https://stackoverflow.com/questions/16549066/java-major-and-minor-garbage-collections

OpenJdk：

https://openjdk.java.net/jeps/122

http://cr.openjdk.java.net/~sundar/8022483/webrev.01/raw\_files/new/src/share/classes/com/sun/tools/hat/resources/oqlhelp.html

Oracle官方DOC：

https://docs.oracle.com/javase/8/javase-books.htm

https://docs.oracle.com/en/java/javase/13/

https://docs.oracle.com/en/java/javase/14/

https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf

https://docs.oracle.com/javase/specs/index.html

https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM\_Troubleshooting/week1/lesson1.pdf

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

资料整理说明：

1. 参考《深入理解Java虚拟机——JVM高级特性与最佳实践》第三版，中文
2. Java虚拟机规范(Java SE 8版)，中文
3. https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf，英文
4. jdk14是编写此文档时最新的jdk版本，
5. java虚拟机规范，jdk8和jdk14大致相同，但jdk8之后增加了很多新特性，所以主要参考jdk14规范
6. 因为我英文太差了，部分有歧义的，可以参考JVM规范相应章节内容

# 一、内存结构

A screenshot of a cell phone

Description automatically generated

PS：蓝色块为线程独享空间

## 1、程序计数器(Program Counter Register)

1.1概述

a当前线程正在执行Java方法，记录所执行的虚拟机指令的地址。否则为空(Undefined)。

b每个线程都有独立的PC，

c字节码解释器工作时通过改变PC的值来选取下一条需要执行的字节码指令

d任意时刻JVM线程只会执行一个方法的代码(At any point, each Java Virtual Machine thread is executing the code of a single method)。

e这块的容量至少保存一个returnAddress类型的数据或一个与平台相关的本地指针的值(wide enough to hold a returnAddress or a native pointer on the specific platform)

f唯一一个在规范中没有规定任何OOM的区域

## 2、Java虚拟机栈(JVM Stack)

2.1概述

a每线程都有独立的虚拟机栈，生命周期和线程相同，可以在堆中分配

b分配的内存地址可以不需要保证连续

c每个方法执行时，都会同步创建个栈帧(Frame)

d栈的大小，可以固定，也可以配置动态收缩。

e栈的大小为固定时，线程可在创建时自定义栈大小

new Thread(ThreadGroup group, Runnable target, String name, long stackSize)

f会抛出StackOverflowError和OutOfMemoryError

### 2.2栈帧(Frame)

2.2.1概述

a用于存储数据和部分过程结果的数据结构

b用于处理动态链接(Dynamic Linking),方法返回值(return values for methods)和异常分派(dispatch exceptions)

c生命周期同方法调用过程，且无论方法是否有异常，均视为结束，并销毁

(A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception))

d每个栈帧都有自己的一组局部变量表(Local Variable)、操作数栈(Operand Stack)、和指向当前方法所属类的运行时常量池的引用(a reference to the runtime constant pool of the class of the current method)

e栈帧数据结构大小仅取决于虚拟机的实现

d栈帧是线程私有数据

### 2.3局部变量表(Local Variable)

2.3.1概述

a长度由编译期决定，并通过方法的code属性(这里)保存以及提供给栈帧使用(supplied in the binary representation of a class or interface along with the code for the method associated with the frame)

b一个局部变量槽(Slot)可以保存一个类型为 boolean、byte、char、short、int、float、对象引用(reference)、 returnAddress的数据，long或者double需要两个连续的局部变量槽

c局部变量通过索引(indexing)来定位访问，索引从0开始

d一个局部变量槽的大小，由具体的虚拟机设计决定

e无论读写变量槽是否为原子操作，都不会引起数据竞争和线程安全问题。

f变量槽可以复用

### 2.4操作数栈(Operand Stack)

2.4.1概述

a后进先出(last-in-first-out, LIFO)

b栈深度在编译期决定，并通过方法的code属性保存以及提供给栈帧使用

c全称是：当前栈帧的操作数栈(Where it is clear by context, we will sometimes refer to the operand stack of the current frame as simply the operand stack.)

d在栈帧创建时，操作数栈是空的

e栈中元素的数据类型必须与字节码指令的序列严格匹配

f JVM的解释执行器引擎被称为“基于栈的执行引擎”，这个栈就是，操作数栈

### 2.5动态链接(Dynamic Linking)

2.5.1概述

a通过一个指向当前方法所在类型的运行时常量池的引用，对当前方法的代码实现动态链接(Each frame contains a reference to the run-time constant pool for the type of the current method to support dynamic linking of the method code)

b类文件的常量池中存有大量的符号引用，字节码中的方法调用指令就是以常量池指向方法的符号引用作为参数

b.1静态解析：指一部分符号在类加载阶段或者第一次使用的时候就被转化为直接引用的转化

b.2动态链接：指一部分符号在每一次运行期间转化为直接引用的转化

### 2.6方法调用

2.6.1概述

a正常调用完成是指在方法执行过程中没有抛出任何异常，包括但不限于通过throw语句显式抛出的异常(A method invocation completes normally if that invocation does not cause an exception to be thrown, either directly from the Java Virtual Machine or as a result of executing an explicit throw statement)

b如果有返回值的话，使用哪种返回指令取决于方法返回值的数据类型

c异常调用完成是指方法在执行过程中抛出了异常，并且没有捕获异常

d方法异常调用完成，一定不会有方法返回值返回给调用者(A method invocation that completes abruptly never returns a value to its invoker)

e方法退出后都必须返回到最初方法被调用的位置，程序才能继续执行

f方法退出过程实际上等同于吧当前栈帧出栈

### 2.7 returnAddress数据类型

a用于jsr,ret,jsr\_w 三种虚拟机指令

b值指向一个jvm操作指令，

cjava语言中没有对应的类型，且无法在运行中改变

d在旧的jvm中用于异常处理，现在用异常表代替

### 2.8 boolean数据类型

a JVM对boolean支持很有限，没有单独的操作指令

b对于java中boolean类型数据，会被编译成jvm中int数据类型(*Instead, expressions in the Java programming language that operate on boolean values are compiled to use values of the Java Virtual Machine int data type*)

c对于boolean数组,JVM复用byte数组的baload和bastore指令

d JVM用1表示true，0表示false

### 2.9 reference数据类型

a三种引用类型：类类型(class Types)、数组类型、接口类型

b默认值是null，但虚拟机没规定null值怎么表示(*The default value of a reference type is null. This specification does not mandate a concrete value encoding null*)

c没有规定reference的长度和结构，

d一般来说要做到下面两件事：

d.1可以直接或间接的查找到对象在堆中的数据存放的起始地址

d.2可以直接或间接的查找到对象所属数据类型在方法区中的存储的类型信息

e如果引用时数组类型，最外面那一维元素的类型，称为该数组的组件类型(component type)

*An array type consists of a component type with a single dimension (whose length is not given by the type)*

f元素类型(element type)定义太拗口，参考虚拟机规范jdk14的2.4章节

## 3、本地方法栈(Native Method Stack)

3.1概述

a为虚拟机用到的本地(Native)方法服务

b如果虚拟机不支持Native方法或不依赖传统栈(通常称C stack)，可以没有本地方法栈

c会抛出StackOverflowError和OutOfMemoryError

e栈的大小，可以固定，也可以配置动态收缩。

f栈的大小为固定时，线程可在创建时自定义栈大小

e虚拟机栈和本地方法栈可以被设计为合二为一，比如HotSpot虚拟机。

## 4、堆(Heap)

4.1概述

a在虚拟机启动时创建，被所有线程共享

b可以划分出线程私有的线程缓冲区(Thread Local Allocation Buffer,TLAB)

c内存地址不需要保证连续，但在逻辑上应被视为连续的

d部分虚拟机设计，会针对大对象(比如数组)存储，要求内存地址是连续的

e容量可以固定，也可以动态扩展

f存储的对象无法显式销毁(objects are never explicitly deallocated)

g会抛出OutOfMemoryError

## 5、方法区(Method Area)

### 5.1概述

a别名非堆(Non-heap),不等价于永久代(Permanent Generation),被所有线程共享

b从JDK8开始，在本地内存中实现的元空间(Metaspace)代替永久代实现方法区

c存储了每个类的结构信息(It stores per-class structures),比如，运行时常量池(the run-time constant pool)，字段和方法数据(field and method data)，构造函数和普通方法的字节码内容(the code for methods and constructors)，还包括一些在类、实例、接口初始化时用到的特殊方法(including the special methods used in class and instance initialization and interface initialization.)

d容量可以固定，也可以动态扩展，可以不实现GC

e会抛出OutOfMemoryError

f此区域的数据存储结构由JVM自定义，JVM规范不做具体规定

### 5.2运行时常量池(Run-Time Constant Pool)

2.1概述

a是class文件中每个类或接口的常量池表(constant\_pool)的运行时表现形式(a per-class or per-interface run-time representation of the constant\_pool table in a class file)

b包含编译期已知的数值字面量和运行期解析后获取的方法或字段引用(ranging from numeric literals known at compile-time to method and fi eld references that must be resolved at run-time)

ps:String.intern()

c会抛出OutOfMemoryError

d虚拟机规范没有对此作任何细节的要求

## 6、直接内存(Direct Memory)

6.1概述

a不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域

b可能会抛出OutOfMemoryError

c可以通过DirectByteBuffer对象作为这块内存的引用进行操作

# 二、编译(Compiling)

## 1概述

1.1这块虚拟机规范中只做示意，并不属于规范内容

1.2 在

## 2前端编译器

## 3即时编译器

## 4提前编译器

## 5虚拟机指令

### 5.1概述

a虚拟机的指令由一个字节长度(意味着操作码总数不能超过256条)、代表某种操作含义的数字(操作码，Opcode)以及跟随其后的0至多个代表此操作所需参数的操作数(operand)构成

b因为JVM采用面向操作数栈而不是面向寄存器的架构，所以大多数指令都不包含操作数，只有一个操作码，指令参数都存放在操作数栈中

c操作数的数量以及长度取决于操作码，如果一个操作数的长度超过了一个字节，那么它将会以高位在前(big-endian)顺序存储

d字节码指令流是单字节对齐，只有lookupswitch 和 tableswitch例

e为了尽可能获得短小精悍的编译代码，限制了JVM操作码的长度为一字节，且放弃了编译后代码的参数长度对齐，但这样会损失一些性能

f由于e的原因，JVM处理超过一个字节数据时，不得不在运行时从字节六中重建出具体数据结构，这又会浪费一些性能

g指令集被故意设计成非完全独立，有一些单独的指令可以在必要的时候用来将一些不支持的类型转换为可支持类型(In other words, the instruction set is intentionally not orthogonal. Separate instructions can be used to convert between unsupported and supported data types as necessary)

h、指令格式表模版说明

|  |  |  |
| --- | --- | --- |
| 字段名 | 字段内容 | 内容说明 |
| 助记符  (mnemonic) | 助记符 |  |
| 操作  (Operation) |  | 该指令功能的简要描述 |
| 格式  (Format) | 助记符  操作数1  操作数2  ... | 1、每行都代表一个包含8bit的字节  2、操作数有可能在编译期产生嵌入到JVM的字节码指令中  3、操作数也有可能在运行期通过计算而得出  并加载到操作数栈  4、操作数来源不同，但表示都是同一个JVM指令执行时所使用的参数值  5、操作数隐式地从操作数栈中获取，会比显式地通过额外的操作数字节及寄存器编号等形式生成的编译后的代码更利于保持JVM字节码的紧凑(*By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java Virtual Machine's code stays compact.)* |
| 结构  (Forms) | 助记符=操作码  (mnemonic = opcode) | 1、操作码通过数字表示 |
| 操作数栈  (Operand Stack) | 示例  ..., value1, value2 →  ..., value3 | 1、示例显示了这个指令执行时回时操作数栈栈顶的value2以及紧随其后的Value1出栈，并把结果值value3入栈  2、(...)省略号代表指令不会影响操作数栈的这部分内容  3、long和double类型的值只使用一个操作数栈元素表示 |
| 描述  (Description) |  | 1、详述了该指令对操作数栈的内容和常量池项所施加的限制  2、该指令所执行的操作  3、执行结果的类型 |
| 链接时异常  (Linking Exceptions) |  | 如果执行该指令时抛出任何链接时异常，每个异常按出现顺序一行一个 |
| 运行时异常  (Run-time Exception) |  | 1、如果执行该指令时抛出任何运行时异常，每个异常按出现顺序一行一个  2、除了已经列出的链接时异常、运行时异常、VirtualMachineError或其子类之外，指令不能再抛出其他任何异常(*Other than the linking and run-time exceptions, if any, listed for an instruction, that instruction must not throw any run-time exceptions except for instances of VirtualMachineError or its subclasses.*) |
| 注意(Notes) |  | 非规范强制的约束，在这里描述(*Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.)* |

### 5.2数据类型和JVM(Types and the Java Virtual Machine)

a大部分与数据类型有关的字节码指令，都有特殊字符表明该指令服务哪个数据类型: *i for an int operation, l for long, s for short, b for byte, c for char, f for float, d for double, and a for reference.*

b JVM会在编译期或运行期将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数据

c 在处理boolean、byte、short和char类型的数组时，也会转为使用对应的int类型字节码

d JVM指令所支持的数据类型表

A screenshot of a cell phone

Description automatically generated

### 5.3加载和存储指令

a将一个本地变量加载到操作数栈的指令包括：iload, iload\_<n>, lload, lload\_<n>, fload, fload\_<n>, dload, dload\_<n>, aload, aload\_<n>

b将一个数值从操作数栈存储到局部变量表的指令包括：istore, istore\_<n>, lstore, lstore\_<n>, fstore, fstore\_<n>, dstore, dstore\_<n>, astore, astore\_<n>.

c将一个常量加载到操作数栈的指令包括：bipush, sipush, ldc, ldc\_w, ldc2\_w, aconst\_null, iconst\_m1, iconst\_<i>, lconst\_<l>, fconst\_<f>, dconst\_<d>.

d将一用于扩充局部变量表的访问索引或立即数的指令：wide

*Gainaccesstomorelocalvariablesusingawiderindex,ortoalargerimmediate operand: wide.*

e补充说明，<n>非负整数，<i>int，<l>long，<f>float，<d>double

### 5.4算数指令

#### 5.4.1概述

a JVM没有明确规定整型数据一出情况，但在整数除法指令以及整数求余指令在除数为0时会抛出ArithmeticException

b JVM要求完全支持IEEE754定义的非标准浮点数值和逐级下溢(gradual underflow)

c对long进行比较，采用带符号方式比较。对浮点类型数进行比较，采用IEEE754标准所定义的无信号比较方式(nonsignaling comparisons)

d把浮点型转为整形数时，采用IEEE754标准的向零舍入模式(round towards zero mode)

e JVM在处理浮点数运算时，不会抛出任何JAVA异常。当一个操作向上溢出时，使用有符号的无穷大表示，当向下溢出时，产生非标准值或带符号的0值。如果操作结果没有明确数学定义，则用NaN值表示

f 各种类型的比较最终都会转化为int类型比较。

#### 5.4.2指令列表

a加法:iadd,ladd,fadd,dadd.

b 减法:isub,lsub,fsub,dsub.

c乘法:imul,lmul,fmul,dmul.

d除法:idiv,ldiv,fdiv,ddiv.

e求余:irem,lrem,frem,drem.

f求负值:ineg,lneg,fneg,dneg.

g移位:ishl,ishr,iushr,lshl,lshr,lushr.

h 按位或: ior, lor.

i按位与: iand, land.

j按位异或: ixor, lxor.

k局部变量自增:iinc

L 比较指令：dcmpg,dcmpl,fcmpg,fcmpl,lcmp

### 5.5类型转换指令

#### 5.5.1数值宽化转换( widening numeric conversions,小范围类型到大范围类型的安全转换)

a转换类型：

a.1 int to long, float, or double

a.2 long to float or double

a.3 float to double

b宽化转换指令：i2l, i2f, i2d, l2f, l2d, and f2d

c 从float到double转换，在FP-strict模式下可以保证精确相等，否则不能保证。int到long、int到double能保证精确相等。

d从int或long到float、long到double时，可能会发生精度丢失

e宽化转换虽然可能发生精度丢失，但不会导致JVM抛出异常

f从int到long转换，是一个简单的带符号扩展操作，即把int数值的二进制补码表示扩充至更宽的格式

g从char到一个整形转换时一个零位扩展，即直接给char的二进制形式上填若干0，以填充成更宽的格式

h JVM直接支持转换，不需要显示转换命令

#### 5.5.2数值窄化转换(narrowing numeric conversions)

a类型转换

a.1 int to byte, short, or char

a.2 long to int

a.3 float to int or long

a.4 double to int, long, or float

b窄化转换指令：i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, and d2f

c因为窄化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，因此会导致数值丢失精度

d从int或者long窄化转换为整形T，转过过程仅仅是简单丢低除最低N个二进制以外的内容，这可能导致与原始值有不同符号

e将浮点类型窄化转换为整数T(int或long)时：

e.1如果原始浮点值时NaN，则转换结果都为0

e.2如果转换结果v在T范围内，则转换结果值就为v

e.3如果转换结果v超过T的范围，则转换结果值为T最大值

e.4如果转换结果v低于T的范围，则转换结果值为T最小值

f从double转为float的窄化转换过程与IEEE754中定义一致：

f.1通过向最近数舍入模式 舍入得到一个可以使用的float类型表示值

f.2如果转换结果太小，无法用float表示，则返回float类型的正负0；

f.3如果转换结果太大，则返回float类型的无穷大

f.4 double的NaN值将转为float的NaN值

g JVM尽管可能发生上下限溢出或精度丢失等情况，但窄转化永远不会抛出运行时JAVA异常

### 5.6对象创建与访问指令

a 创建类实例: new.

b 创建数组: newarray, anewarray, multianewarray.

c 访问类字段和类实例字段: getstatic, putstatic, getfield, putfield.

d 把一个数组元素加载到操作数栈:baload,caload,saload,iaload, laload, faload, daload, aaload.

e 讲一个操作数栈的值存储到数组元素中: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore.

f 取数组长度: arraylength.

g 检查类实例或数组类型: instanceof, checkcast.

|  |  |
| --- | --- |
| 助记符 | new |
| 格式 |  |
| 操作栈 |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### 5.7操作数栈管理指令

直接控制操作数栈：pop, pop2, dup, dup2, dup\_x1, dup2\_x1, dup\_x2, dup2\_x2, swap.

### 5.8控制转移指令

a 控制转移指令可以让JVM从指定指令而不是控制转移指令的下一条指令继续执行程序(The control transfer instructions conditionally or unconditionally cause the Java Virtual Machine to continue execution with an instruction other than the one following the control transfer instruction)

b指令类型：

b.1条件分支: ifeq, ifne, iflt, ifle, ifgt, ifge, ifnull, ifnonnull, if\_icmpeq, if\_icmpne, if\_icmplt, if\_icmple, if\_icmpgt if\_icmpge, if\_acmpeq, if\_acmpne.

b.2复合条件分支: tableswitch, lookupswitch.

b.3无条件分支: goto, goto\_w, jsr, jsr\_w, ret.

c 所有int类型的条件分支转移指令进行的都是有符号的比较操作(*All int conditional control transfer instructions perform signed comparisons*)

### 5.9方法调用和返回指令

#### 5.9.1方法调用指令

a 指令invokevirtual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)

b 指令invokeinterface用于调用接口方法，会在运行时搜索指定对象所实现的这个接口方法，并找出合适的方法进行调用

c 指令invokespecial 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、私有方法、父类方法

d 指令invokestatic调用命名类中的类方法

e 指令invokedynamic用于调用以绑定了invokedynamic指令的调用点对象作为目标的方法(*invokes the method which is the target of the call site object bound to the invokedynamic instruction)*。每条invokedynamic都有独一无二的连接状态(*each occurrence of an invokedynamic instruction has a unique linkage state, unlike the other instructions which invoke methods*)

#### 5.9.2方法返回指令

a 指令ireturn 用于返回boolean, byte, char, short, int类型

b 指令lreturn, freturn, dreturn, areturn返回对应类型的值

c 指令return为声明为void的方法、实例初始化方法、类和接口的类初始化方法使用

### 5.10异常处理指令

a显式异常抛出的操作都由athrow指令实现

b JVM中处理异常，近代JVM都是用异常表实现，很久之前用jsr和ret指令实现

### 5.11同步指令

a JVM支持方法级的同步和方法内部的一条短指令序列的同步，都是用同步锁实现(monitor，管程，更常见被直接称为锁)支持。

b方法级的同步时隐式的，从方法的常量池中的方法表结构中的ACC\_SYNCHRONIZED区分是否同步方法

c同步方法执行期间抛出异常，且方法内部无法处理时，方法所持有的锁在异常抛到同步方法边界之外时自动释放

d指令monitorenter 和monitorexit 用来支持指令序列的同步(synchronized代码块)

### 5.12 invokedynamic指令

# 三、类(Class)

## 1概述

1.1类文件格式不是必须存在于文件里。

1.2多字节数据项必须严格按照大端在前(big-endian)顺序进行存储(Multibyte data items are always stored in big-endian order,where the high bytes come first)

1.3 Class文件是一组以8字节为基础单位的二进制流，各个数据项严格按照顺序紧凑排列，没有任何分隔符

1.4Class文件格式采用一种类似C语言结构体的伪结构(This chapter presents the class file format using pseudostructures written in a C-like structure notation)，只有两种数据类型:无符号数(unsigned)和表(table)

1.5无符号数是基本的数据类型，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，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

1.6表由多个无符号数或其他表作为数据项构成的复合数据类型

1.7当描述同一类型但数量不定的多个数据时，经常会使用一个前置的容量计数器加若干个连续的数据项的形式，这时候称这一系列连续的某一类型的数据为某一类型的集合

## 2类结构

### 2.1类结构元素

|  |  |  |
| --- | --- | --- |
| 类型 | 数量 | 说明 |
| u4 | 1 | 魔数(magic)，确定是否为虚拟机所接受的类文件，固定值为0xCAFEBABE |
| u2 | 1 | 副版本号(minor\_version)  1、主版本号为45-55时，副版本号可以为任何值  2、从主版本号56(jdk12)开始，副版本号必须为0或65535  3、当副版本号为65535时，为预览版 |
| u2 | 1 | 主版本号(major\_version)，  1、值从45开始  2、即使文件格式无任何变化，虚拟机也必须拒绝执行类主版本号超过虚拟机版本号的文件。 |
| u2 | 1 | constant\_pool\_count;常量池计数器 |
| cp\_info | constant\_pool[constant\_pool\_count-1] | 常量池  1、是一种表结构  2、以1～constant\_pool\_count-1为索引  3、第一个字节作为类型标记，这个字节称为标签字节(tag byte)  4、主要存放两大类常量：字面量(Literal)和符号引用(Symbolic References)  5、通用格式：  cp\_info {  u1 tag;  u1 info[];  } |
| u2 | 1 | access\_flags;访问标志，  1、一种由标志构成的掩码  The value of the access\_flags item is a mask of flags used to denote access permissions to and properties of this class or interface  2、取值以及含义参考2.3  3、没有使用到的标志位都为0 |
| u2 | 1 | this\_class;类索引  1、取值为常量池有效的索引值  2、常量池中索引处的成员必须为CONSTANT\_Class\_info类型 |
| u2 | 1 | super\_class;父类索引  1、Java不允许多重继承，所以父类索引只有一个  2、取值要么为0，要么是常量池中某个有效索引值  3、当值为0，此类只能用于表示Object类  4、当值不为0，常量池中索引处的成员必须为CONSTANT\_Class\_info类型  5、所有父类的access\_flags，均不能带有ACC\_FINAL标志  Neither the direct superclass nor any of its superclasses may have the ACC\_FINAL flag set in the access\_flags item of its ClassFile structure. |
| u2 | 1 | interfaces\_count;接口索引计数器，如果没有接口，则取值0 |
| u2 | interfaces[interfaces\_count] | 接口集合  1、每个元素值都是常量池中某个有效索引值  2、按implements关键字后面的接口，从左到右排列 |
| u2 | 1 | fields\_count;字段计数器 |
| field\_info | fields[fields\_count] | 字段表  1、每个成员都是field\_info结构，参考2.4  2、包括类变量和实例变量  3、不包括方法内声明的局部变量  4、不包括从父类或父接口继承的那些字段 |
| u2 | 1 | methods\_count;方法计数器 |
| method\_info | methods[methods\_count] | 1、每个成员都是method\_info结构，参考2.5  2、方法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  3、支持方法访问权限(access\_flags)既不是ACC\_NATIVE又不是ACC\_ABSTRACT |
| u2 | 1 | attributes\_count;属性计数器 |
| attribute\_info | attributes[attributes\_count] | 1、每个成员都是attribute\_info结构，参考2.6 |

### 2.2常量池(constant\_pool)

#### 2.2.1基本格式

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| tag | u1 |  |
| info[] | u1 |  |

#### 2.2.2常量池类型(tag)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | tag | 类文件版本 | 首次javaSE版本 | 说明 |
| CONSTANT\_Utf8 | 1 | 45.3 | 1.0.2 | UTF-8编码的字符串 |
| CONSTANT\_Integer | 3 | 45.3 | 1.0.2 | 整型字面量 |
| CONSTANT\_Float | 4 | 45.3 | 1.0.2 | 浮点型字面量 |
| CONSTANT\_Long | 5 | 45.3 | 1.0.2 | 长整型字面量 |
| CONSTANT\_Double | 6 | 45.3 | 1.0.2 | 双精度浮点型字面量 |
| CONSTANT\_Class | 7 | 45.3 | 1.0.2 | 类或接口的索引 |
| CONSTANT\_String | 8 | 45.3 | 1.0.2 | 字符串类型字面量 |
| CONSTANT\_Fieldref | 9 | 45.3 | 1.0.2 | 字段的符号引用 |
| CONSTANT\_Methodref | 10 | 45.3 | 1.0.2 | 类中方法的符号引用 |
| CONSTANT\_InterfaceMethodref | 11 | 45.3 | 1.0.2 | 接口中方法的符号引用 |
| CONSTANT\_NameAndType | 12 | 45.3 | 1.0.2 | 字段或方法的部分符号引用 |
| CONSTANT\_MethodHandle | 15 | 51 | 7 | 表示方法具柄 |
| CONSTANT\_MethodType | 16 | 51 | 7 | 方法类型 |
| CONSTANT\_InvokeDynamic | 18 | 51 | 7 | 表示一个动态方法调用点 |
| CONSTANT\_Module | 19 | 53 | 9 | 表示一个模块 |
| CONSTANT\_Package | 20 | 53 | 9 | 表示一个模块中开放或者导出的包 |
| CONSTANT\_Dynamic | 17 | 55 | 11 | 表示一个动态计算常量 |

2.2.2常量池数据类型的结构

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 字段 | 字段类型 | 说明 |
| CONSTANT\_Utf8 | tag | u1 | UTF-8编码的字符串 |
| length | u2 |
| bytes[] | u1 |
| CONSTANT\_Integer | 3 | 45.3 | 整型字面量 |
|  |  |
| CONSTANT\_Float | 4 | 45.3 | 浮点型字面量 |
| CONSTANT\_Long | 5 | 45.3 | 长整型字面量 |
| CONSTANT\_Double | 6 | 45.3 | 双精度浮点型字面量 |
| CONSTANT\_Class | 7 | 45.3 | 类或接口的索引 |
| CONSTANT\_String | 8 | 45.3 | 字符串类型字面量 |
| CONSTANT\_Fieldref | 9 | 45.3 | 字段的符号引用 |
| CONSTANT\_Methodref | 10 | 45.3 | 类中方法的符号引用 |
| CONSTANT\_InterfaceMethodref | 11 | 45.3 | 接口中方法的符号引用 |
| CONSTANT\_NameAndType | 12 | 45.3 | 字段或方法的部分符号引用 |
| CONSTANT\_MethodHandle | 15 | 51 | 表示方法具柄 |
| CONSTANT\_MethodType | 16 | 51 | 方法类型 |
| CONSTANT\_InvokeDynamic | 18 | 51 | 表示一个动态方法调用点 |
| CONSTANT\_Module | 19 | 53 | 表示一个模块 |
| CONSTANT\_Package | 20 | 53 | 表示一个模块中开放或者导出的包 |
| CONSTANT\_Dynamic | 17 | 55 | 表示一个动态计算常量 |

2.3访问标志(access\_flags)

|  |  |  |
| --- | --- | --- |
| 标志 | 值 | 说明 |
| ACC\_PUBLIC | 0x0001 | public |
| ACC\_FINAL | 0x0010 | final，不能与ACC\_ABSTRACT同时设置 |
| ACC\_SUPER | 0x0020 | 是否使用invokespecial指令的新语义，因为invokespecial指令在jdk1.0.2以后做了改变，所以在jdk1.0.2之后，这里都为真 |
| ACC\_INTERFACE | 0x0200 | 接口，设置了此值，ACC\_ABSTRACT必须同时设置  ACC\_FINAL, ACC\_SUPER, ACC\_ENUM和 ACC\_MODULE，必须不能设置 |
| ACC\_ABSTRACT | 0x0400 | 抽象类，不能实例化 |
| ACC\_SYNTHETIC | 0x1000 | 标志该类或接口是由编译器生成 |
| ACC\_ANNOTATION | 0x2000 | 声明为注解，同时必须设置ACC\_SYNTHETIC |
| ACC\_ENUM | 0x4000 | 声明为枚举 |
| ACC\_MODULE | 0x8000 | 声明为模块，如果这个值被设置，则不能有其他标志被设置  If the ACC\_MODULE flag is set in the access\_flags item, then no other flag in the access\_flags item may be set |

2.4字段表结构(field\_info)

2.4.1基础结构

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| access\_flags | u2 | 访问标志，参考2.4.2 |
| name\_index | u2 | 字段简单名称，指向常量池一个CONSTANT\_Utf8\_info值的索引 |
| descriptor\_index | u2 | 字段的描述符，指向常量池一个CONSTANT\_Utf8\_info值的索引 |
| attributes\_count | u2 | 参考2.7 |
| attributes[attributes\_count] | attribute\_info | 参考2.7 |

2.4.2字段访问标志

|  |  |  |
| --- | --- | --- |
| 标志 | 值 | 说明 |
| ACC\_PUBLIC | 0x0001 | public |
| ACC\_PRIVATE | 0x0002 | private |
| ACC\_PROTECTED | 0x0004 | protected |
| ACC\_STATIC | 0x0008 | static |
| ACC\_FINAL | 0x0010 | final |
| ACC\_VOLATILE | 0x0040 | volatile |
| ACC\_TRANSIENT | 0x0080 | transient |
| ACC\_SYNTHETIC | 0x1000 | 标志该字段是由编译器生成，如果字段存在这个标志，则不能有其他任何访问标志 |
| ACC\_ENUM | 0x0001 | 枚举 |

a ACC\_PUBLIC、ACC\_PRIVATE、ACC\_PROTECTED三选一

b ACC\_FINAL、ACC\_VOLATILE二选一

c接口中的字段必须有ACC\_PUBLIC,ACC\_STATIC,ACC\_FINAL，三个标志

2.5方法结构(method\_info)

2.5.1基础结构

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| access\_flags | u2 | 访问标志，参考2.5.2 |
| name\_index | u2 | 字段简单名称，指向常量池一个CONSTANT\_Utf8\_info值的索引，必须是有效的非限定名称，或者<init>，或者<clinit> |
| descriptor\_index | u2 | 字段的描述符，指向常量池一个CONSTANT\_Utf8\_info值的索引，  1、当是类的<init>方法，必须描述为void方法  2、当类名是<clinit>，必须描述为void方法，且jdk7以后，此方法无参数 |
| attributes\_count | u2 | 参考2.7 |
| attributes[attributes\_count] | attribute\_info | 参考2.7 |

2.5.2访问标志

|  |  |  |
| --- | --- | --- |
| 标志 | 值 | 说明 |
| ACC\_PUBLIC | 0x0001 | public |
| ACC\_PRIVATE | 0x0002 | private |
| ACC\_PROTECTED | 0x0004 | protected |
| ACC\_STATIC | 0x0008 | static，jdk7以后<clinit>方法必须有此标志 |
| ACC\_FINAL | 0x0010 | final |
| ACC\_SYNCHRONIZED | 0x0020 | synchronized |
| ACC\_BRIDGE | 0x0040 | 桥接方法，为了泛型方法兼容jdk1.5以前版本，编译器自动生成 |
| ACC\_VARARGS | 0x0080 | 有可变长参数的标志 |
| ACC\_NATIVE | 0x0100 | native |
| ACC\_ABSTRACT | 0x0400 | abstract |
| ACC\_STRICT | 0x0800 |  |
| ACC\_SYNTHETIC | 0x1000 |  |

a ACC\_PUBLIC, ACC\_PRIVATE,ACC\_PROTECTED 必须三选一

b jdk8以前，接口的方法必须有ACC\_PUBLIC 和ACC\_ABSTRACT标志

c jdk8以及以后，接口的方法必须ACC\_PUBLIC ， ACC\_PRIVATE二选一

d 方法如果标记了ACC\_ABSTRACT，则不能有ACC\_PRIVATE, ACC\_STATIC, ACC\_FINAL,ACC\_SYNCHRONIZED, ACC\_NATIVE, ACC\_STRICT标记

e实例的初始化方法只能ACC\_PUBLIC, ACC\_PRIVATE，ACC\_PROTECTED三选一，以及ACC\_VARARGS, ACC\_STRICT, ACC\_SYNTHETIC可选，不能有其他标志

f实例或接口的方法被隐式调用，只能设置ACC\_STATIC 和 ACC\_STRICT标志

2.5.3 <init>方法

2.5.4 <clinit>方法

2.6属性结构(attribute\_info)

2.6.1基本结构

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| attribute\_name\_index | u2 | 指向常量池一个CONSTANT\_Utf8\_info值的索引 |
| attribute\_length | u4 | 属性长度，长度不包括自身和attribute\_name\_index的长度 |
| info[attribute\_length] | u1 |  |

2.6.2属性类型

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 位置 | 首次版本 | 说明 |
| ConstantValue | field\_info | 45.3 |  |
| Code | method\_info | 45.3 |  |
| Exceptions | method\_info | 45.3 |  |
| SourceFile | classFile | 45.3 |  |
| LineNumberTable | code | 45.3 |  |
| LocalVariableTable | code | 45.3 |  |
| InnerClasses | classFile | 45.3 |  |
| Synthetic | classFile  field\_info  method\_info | 45.3 |  |
| Deprecated | classFile  field\_info  method\_info | 45.3 |  |
| EnclosingMethod | classFile | 49 |  |
| Signature | classFile  field\_info  method\_info | 49 |  |
| SourceDebugExtension | classFile | 49 |  |
| LocalVariableTypeTable | code | 49 |  |
| RuntimeVisibleAnnotations | classFile  field\_info  method\_info | 49 |  |
| RuntimeInvisibleAnnotations | classFile  field\_info  method\_info | 49 |  |
| RuntimeVisibleParameterAnnotations | method\_info | 49 |  |
| RuntimeInvisibleParameterAnnotations | method\_info | 49 |  |
| AnnotationDefault | method\_info | 49 |  |
| StackMapTable | code | 50 |  |
| BootstrapMethods | classFile | 51 |  |
| RuntimeVisibleTypeAnnotations | classFile  field\_info  method\_info  code | 52 |  |
| RuntimeInvisibleTypeAnnotations | classFile  field\_info  method\_info  code | 52 |  |
| MethodParameters | method\_info | 52 |  |
| Module | classFile | 53 |  |
| ModulePackages | classFile | 53 |  |
| ModuleMainClass | classFile | 53 |  |
| NestHost | classFile | 55 |  |
| NestMembers | classFile | 55 |  |

2.6.3属性类型结构

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 字段 | 字段类型 | 说明 |
| ConstantValue | field\_info | 45.3 |  |
| Code | method\_info | 45.3 |  |
| Exceptions | method\_info | 45.3 |  |
| SourceFile | classFile | 45.3 |  |
| LineNumberTable | code | 45.3 |  |
| LocalVariableTable | code | 45.3 |  |
| InnerClasses | classFile | 45.3 |  |
| Synthetic | classFile  field\_info  method\_info | 45.3 |  |
| Deprecated | classFile  field\_info  method\_info | 45.3 |  |
| EnclosingMethod | classFile | 49 |  |
| Signature | classFile  field\_info  method\_info | 49 |  |
| SourceDebugExtension | classFile | 49 |  |
| LocalVariableTypeTable | code | 49 |  |
| RuntimeVisibleAnnotations | classFile  field\_info  method\_info | 49 |  |
| RuntimeInvisibleAnnotations | classFile  field\_info  method\_info | 49 |  |
| RuntimeVisibleParameterAnnotations | method\_info | 49 |  |
| RuntimeInvisibleParameterAnnotations | method\_info | 49 |  |
| AnnotationDefault | method\_info | 49 |  |
| StackMapTable | code | 50 |  |
| BootstrapMethods | classFile | 51 |  |
| RuntimeVisibleTypeAnnotations | classFile  field\_info  method\_info  code | 52 |  |
| RuntimeInvisibleTypeAnnotations | classFile  field\_info  method\_info  code | 52 |  |
| MethodParameters | method\_info | 52 |  |
| Module | classFile | 53 |  |
| ModulePackages | classFile | 53 |  |
| ModuleMainClass | classFile | 53 |  |
| NestHost | classFile | 55 |  |
| NestMembers | classFile | 55 |  |

## 3类的加载

### 3.1类加载过程

A close up of a piece of paper

Description automatically generated

#### 3.1.1概述

a类加载必须按加载、验证、准备、初始化、卸载顺序按部就班的开始，但有可能会在一个阶段执行的过程中调用、激活另一个阶段

b解析在一些情况下可以在 初始化 阶段以后开始

c加载阶段和连接阶段部分工作交叉进行

d创建好类以后，随时可以进入准备阶段，但必须在初始化阶段开始之前完成(*Preparation may occur at any time following creation but must be completed prior to initialization.)*

e

f

#### 3.1.2加载(Loading)

3.1.2.1概述

a什么时候开始加载，JVM规范没有强制约束

b接口或类C的创建时由另外个类或接口D触发或D调用JSE类库中的某些方法触发，比如反射等

c如果C是数组，则因数组类没有外部的二进制表示，所以都是JVM创建，不通过类加载器加载(created by the Java Virtual Machine rather than by a class loader)

d无论加载器L直接创建C或授权其他加载器创建C，都称L是C的初始加载器

e JVM支持两种加载器：JVM提供的引导类加载器、用户自定义的类加载器

f在C(或它的父类)的初始加载器抛出ClassNotFoundException错误时，JVM试图验证或解析C，必须抛出NoClassDefFoundError(*If the Java Virtual Machine ever attempts to load a class C during verification or resolution (but not initialization), and the class loader that is used to initiate loading of C throws an instance of ClassNotFoundException, then the Java Virtual Machine must throw an instance of NoClassDefFoundError whose cause is the instance of ClassNotFoundException*)

g一个功能良好的加载器应当保证三个行为：

g.1如果名称相同，返回的class对象必定相同

g.2无论是L1加载，还是L1委托给L2加载，都必须返回相同Class对象

g.3用户自定义加载器在加载出错时，必须反应加载出错点

h加载数组类A创建过程遵循以下规则

i加载阶段结束后，class文件就会被以指定格式存储在方法区，并在类型数据存放好后，会在堆中生成对应的Class类对象，作为访问方法区中类型数据的外部接口

3.1.2.2双亲委派

3.1.2.3类加载器

#### 3.1.3验证(Verification)

3.2.3.1概述

a因为JVM虚拟机规范和《深入理解JAVA虚拟机规范(第三版)》对这块介绍差异较大，这里以JVM规范为主要介绍内容, 《深入理解JAVA虚拟机规范(第三版)》内容作为补充

b如果类或接口不满足静态或结构的约束，必须在失败处抛出VerifyError异常

c理解这块，最好先对JVM指令集有初步了解。

3.2.3.2格式验证

a检查项包括但不限于：

a.1是否魔数开头

a.2主、次版本是否符合当前JVM范围

a.3常量池中是否存在 不存在或类型错误的常量

a.4常量类型的数据结构是否正确

a.5所有属性的数据结构是否正确

a.6不能缺失内容，尾部不能有多余数据

a.7所有字段/方法引用，必须有有效的名称、类、描述符

b 格式检查不保证某个字段/方法/描述符是否真实存在，只保证格式正确

c

3.2.3.3JVM代码约束

a静态约束(static constraint)

1. 此约束用来定义文件是否编排良好，确定了jvm指令在code数组中是如何编排，以及某些特殊指令必须带有哪些操作数等
2. code数组中只能出现指定？的指令；使用保留的指令或其他JVM规范外的指令，不能出现在code中
3. 如果class文件大于51，则jsr和jsr\_w也不能出现在code数组中

b结构化约束

3.2.3.4 Class文件校验

a类型检查验证

b类型推导验证

a、

3.2.3.5 JVM限制

#### 3.1.4准备(Preparation)

a此阶段时正式为类中定义的静态变量在方法区中分配内存并设置变量初始值的阶段

b此阶段仅处理类变量，不包括实例变量

c当用final static 声明常量，则会在此此阶段赋具体值

d JVM在此阶段也有些强制约束，这块有点晦涩，，建议看JVM规范**英文原文**：

d.1假定L1是C的加载器，对于每个在C中的方法m，m覆盖了C的父类或父接口<D,L2>中某个方法，m返回值是Tr，形参从个Tf1....Tfn。

d.1.1如果Tr不是数组，则用T0代替Tr；否则，T0就表示Tr的元素类型

d.1.2从1到n的每个i来说，如果Tfi不是数组类型，Ti表示Tfi，否则Ti就是Tfi的元素类型

d.1.3对于0到n的每个i来说，Ti(L1)=Ti(L2)都该成立

d.2假定C实现了父接口<I,L3>中的方法m，但c自己却没有声明这个方法m，那么就把声明了由C所继承的方法m实现的那个超类标记为<D,L2>( *For each instance method m declared in a superinterface <I, L3> of C, if C does not itself declare an instance method that can override m, then a method is selected with respect to C and the method m in <I, L3>. Let <D, L2> be the class or interface that declares the selected method*)。JVM也加了同**d.1**相同的约束。

#### 3.1.5解析(Resolution)

3.1.5.1概述

a解析阶段是JVM将常量池内的符号引用替换为直接引用的过程(Resolution is the process of dynamically determining one or more concrete values from a symbolic reference in the run-time constant pool)

b**符号引用**(symbolic reference):

b.1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就行

b.2与JVM实现的内存布局无关，引用的目标不一定是已经加载到JVM内存当中的内容

b.3JVM实现的内存布局可以各不相同，但其能接受的符号引用必须是一致的

c**直接引用**(Direct reference):

c.1是可以直接指向目标的指针、相对偏移量或者是个能间接定位到目标的句柄

c.2和JVM的内存布局直接相关，同个符号在不同JVM上翻译出来的直接引用一般不相同。

c.3如果有了直接引用，那引用的目标一定已存在于JVM的内存中

d JVM规范未规定解析阶段具体发生时间，只要求了在执行anewarray, checkcast, getfield, getstatic, instanceof, invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual, ldc, ldc\_w, ldc2\_w, multianewarray, new, putfield, putstatic等指令时，需要先对符号引用进行解析(*Execution of any of these instructions requires resolution of the symbolic reference*)

e非invokedynamic指令以外的多次解析同一个符号引用，JVM需要保证在同一个实体中，解析结果必须与第一次解析结果相同，即使第一次解析失败，后续解析成功。

f对于invokedynamic指令发生的符号引用解析，其解析结果仅仅限于当前指令，当前符号引用对于其他指令有效，均为未解析状态(*The symbolic reference is still unresolved for all other instructions in the class file, of any opcode, which indicate the same entry in the run-time constant pool as the invokedynamic instruction above*)

g符号引用解析过程发生错误，抛出IncompatibleClassChangeError或它子类错误

3.1.5.2解析类和接口

a假设当前代码所处的类为D，需要将未解析的符号引用N解析为接口或类C的引用，解析过程

a.1如果C不是数组类型，JVM会把N的全限定名传给D的类加载器去加载C。在加载C过程中，出现任何失败，均认为当前解析过程失败

a.2如果C是一个数组类型，并且数组的元素类型为对象，JVM将a.1形式加载数组的元素类型，接着JVM生成一个代表数组维度和元素的对象

a.3当执行完前两点，并且无异常，那么C在JVM中已经是个有效的类或接口，但还需要进行访问权限验证。如果访问权限验证失败，则这将抛出IllegalAccessError错误

b D可访问类或接口C的权限条件,需满足以下之一：

b.1 C是Public，且与D处于同一个模块

b.2 C是Public，与D不处于同一个模块，但C所处模块允许被D所处的模块访问

b.3 C不是public，但C和D处于同一个包

3.1.5.3解析字段

a要解析一个未被解析过的字段符号引用

a.1首先将会对字段表中class\_index项中索引的CONSTANT\_Class\_info符号(字段所属的类或接口C)引用进行解析，如果解析C失败，则抛出NoSuchFieldError错误

a.2如果C解析成功，且C本身包含了匹配的字段，则返回字段的直接引用。

a.3如果C本身未包含匹配的字段，会按照继承关系，从下往上递归搜索父接口或父类，查找匹配的字段。如果找到了匹配字段，则返回字段的直接引用。否则抛出NoSuchFieldError错误。

a.4如果字段查找成功，则判断字段访问权限，如果不具备访问权限，则抛出IllegalAccessError异常

b字段或方法R可访问权限，需要满足以下条件之一

b.1 R是public

b.2 R是protected，定义在类C中，且D是C的子类或本身

b.3 R是protected或者默认权限，所在类C和类D在同一个包中

b.4 R是private，被声明在D类中

3.1.5.4解析方法

a解析方法开始的前提是，所在类C能被正常解析

b如果在类的方法表中发现class\_index中索引的C是接口的话，直接抛出IncompatibleClassChangeError错误

c解析过程：

c.1如果C中有同名的方法引用且是签名多态方法(signature polymorphic method declaration)，则查找成功。方法描述符中所提到的全部类名也被解析

c.2如果C中方法引用存在相同的名称和描述符，那么查找也成功

c.3如果C有父类，递归重复c.1-c.3步骤，在父类中查找

c.4否则C尝试从父接口去定位方法

c.5如果C中某些最具体的父接口方法(maximally-specific superinterface methods)与方法引用所指定的描述符和名称相符，且存在有且只有一个方法未设置ACC\_ABSTRACT标志，那么返回次方法，查找结束

c.6如果C的任意父接口中存在一个与方法引用相同的描述符以及名称的方法，该方法即无ACC\_STATIC标志，有没ACC\_PRIVATE标志，那么返回此方法，并成功结束

c.7上诉步骤仍找不到，宣告查找失败

d如果方法查找失败，抛出NoSuchMethodError

e如果方法查找成功，但无访问权限，则抛出IllegalAccessError

f类或接口C的最具体的父接口方法，是指满足以下所有条件的任意方法:

f.1方法是C的直接或间接父接口

f.2此方法是根据指定的名称和描述符来声明的

f.3此方法即无ACC\_STATIC标志，也没ACC\_PRIVATE标志

f.4此方法是唯一存在的

3.1.5.5解析接口方法，只列出与普通方法差异的

a.解析接口方法开始的前提是，所在接口I能被正常解析

b如果在接口的方法表中发现I是类的话，直接抛出IncompatibleClassChangeError错误

c否则，在I的父接口中递归查找，直到Object类，如果存在相同方法，这直接返回

3.1.5.6

#### 3.1.6初始化(Initialization)

a、初始化阶段，就是执行构造器<clinit>()方法的过程

b、只有发生以下行为时，类或接口才会初始化(A class or interface C may be initialized only as a result of:)

b.1执行指定指令：new、getstatic，putstatic、invokestatic

b.2在初次调用java.lang.invoke.MethodHandle实例时，方法句柄是2 (REF\_getStatic), 4 (REF\_putStatic), 6 (REF\_invokeStatic), or 8 (REF\_newInvokeSpecial)

b.3调用反射方法

b.4对某个类的子类进行初始化

b.5如果接口c声明了一个非抽象、非静态的方法，有类直接或间接引用此接口的类初始化(*If C is an interface that declares a non-abstract, non-static method, the initialization of a class that implements C directly or indirectly.)*

b.6被指定为jvm启动时初始类

c在类或接口初始化之前，当前类或接口必须已经通过验证、准备阶段。

d直到初始化阶段，JVM才真正开始执行类中编写的Java代码，将主导权交给app

e每个类或接口都有个唯一的初始化锁，锁的实现有JVM实现自定义

f JVM必须保证一个类的初始化线程安全。

g初始化过程

**注：**

1:执行初始化过程，线程的中断状态不受影响

2:抛出的异常E，不是Error或其子类，那就为E创建个ExceptionInInitializerError实例代替它，如果OOM问题，则用OOM错误代替E

3:利用ConstantValue属性初始化类中每个final static字段(*Then, initialize each final static field of C with the constant value in its ConstantValue attribute, in the order the fields appear in the ClassFile structure*.)

4:初始化方法<clinit>

A close up of a map

Description automatically generated

## 4模块化

# 四、JVM字节码执行

# 五、总结