

# 第3章

## Java 数据结构与面向对象

---

### 本章摘要

- 数据结构是什么，为什么需要数据结构
- 数据结构的发展历史以及与算法的关系
- Java 数据结构的实现机制
- Java 数据结构——面向对象之技术必然性与偶然性
- Java 数据结构的字节码格式分析
- 大端与小端

上一章讲解了 JVM 内部的 `call_stub` 函数指针及其所对应的例程逻辑，难度比较大，属于口味很重的“菜系”，所以本章就来点味道淡一点的“菜”，以免大家消化不良。荤素搭配，咸淡相宜，才能吃得有滋有味。

本章开始聊一聊关于数据结构和算法的那些事儿。我们知道，算法和数据结构是计算机的基础，而 Java 虚拟机的执行引擎框架仍然以此为基础。如同物理 CPU 能够识别字节码和存储单元，C 语言编译器能够识别使用 C 定义的结构体一样，JVM 执行引擎同样能够识别 Java 语言中的基础数据结构——类型，所以 Java 语言里的数据类型对于 JVM 执行引擎而言非常重要。我能想到的最浪漫的事儿，就是看着 JVM 执行引擎对 Java 类型和对象说：“我认得你，我们好像在哪儿见过，你记得吗？”

笔者试图通过历史上的一些事件，来推测詹爷当年将 Java 语言设计成面向对象语言的背景和原因，试图分析 Java 语言如此设计在技术上的偶然性和必然性。本章内容总体上比较“轻松”，大家可以当作小说去阅读。

## 3.1 从 Java 算法到数据结构

如果非要用一句简单的话来概括何谓“编程”，那么下面的这句话虽然不一定能够反映“编程”的全貌，但至少能够从一个侧面描述编程的本质：

“编程就是使用合适的算法处理特定的数据结构。”

同理，也可以使用下面这句话来概括什么是“程序”：

“程序就是算法与数据结构的有机结合。”

---

注：这里所说的算法是一种广义的概念，凡是能够完成有特定逻辑的一组指令的集合都可以称为算法。使用 C# 完成一个绚丽的 PC 版视窗是一种算法，使用 C++ 完成一个基于 epoll 的 IO 框架是一种算法，使用 VB 为 Excel 写一个宏脚本是一种算法，使用 Java 完成了一个网页数据加载也是一种算法。这里的算法不局限于那些特定的用于解决数学问题的技术逻辑。

---

算法是指令驱动的，一条条指令按照一定顺序执行，彼此协作，最终实现某种特定的逻辑，便完成了特定的算法。Java 程序的算法由 Java 字节码指令所驱动。而数据结构往往会被作为算法的输入、输出和中间产出，即使输出的是一个简单的整型数字，也是一种数据结构。

本来设计算法是一件挺快乐的事儿，当使用 JavaScript 完成了一个能够绘制柱状统计图的小组件的时候，内心一定是激动无比的；或者当使用汇编完成了一个哈希表设计的时候，一定会觉得自个儿特厉害。但是当算法遇上了不同的操作系统，或者不同的硬件平台，一件原本很快乐的事情便硬生生地被变成一件痛苦的事儿。例如，你想开发一款高并发、高性能的网络通信组件，或者一种分布式的调度器，如果选择的不是 Java 或者 Python，而是 C、C++ 等语言，那么多线程的处理、网络接口的调用等与平台和硬件相关的算法一定会让你如火的热情降到冰点以下。

詹爷之所以伟大就在于，他借助于 JVM 虚拟机和中间语言字节码，完成了指令的跨平台兼容，从而使得 Java 算法能够兼容大部分主流平台。这直接导致算法又变成了一项有趣的活儿，开发者终于又可以集中精力于编写算法这一件事上面，再也不用担心自己的算法是否兼容其他平台，也不用关注底层不同的实现。如同耕耘一样，没有耕坏的地，只有累死的牛，换而言之，没有兼容不了的平台，只有写不出的程序。

当年詹爷实现 Java “write once, run anywhere”的终极武器是“Java 字节码”这种中间语言（ML）技术。这条技术路线的选择有其偶然性，但是更多的却是一种必然性。

虽然可以选择的技术实现方式有很多种，但是总体思路是被限制死的。总而言之，在目前

计算机硬件执行架构的限制下，可选的“技术路径”只有那么几条，总结起来只有以下3条。

### 1) 直接编写目标硬件平台的机器码指令

这种方式最直接，也最有效，目标硬件平台所支持的指令该是什么就是什么，没有任何二义性，不会产生任何混淆，清清楚楚，明明白白。不过一般只有最底层的软件程序才需要使用这种方式，同时早期的软件程序也采用这种方式，毕竟那时候的编程语言还没有如今这么智能化。

### 2) 使用高级语言编程，通过编译器实现兼容

可以这么说，如果在马路上拿起一块小石头随便往大街上扔过去，砸中的十有八九就是这种机制的高级编程语言，例如C、C++、Delphi等都是如此。这种机制要求比较高，不仅要求在不同的目标平台上安装对应的编译器，还需要软件程序本身针对不同的目标平台调用不同的底层API。从源代码，到编译器，到打包，再到编译后的目标文件乃至可执行程序，全部都是目标平台相关的，没有一个是可以只“write once”就能实现“run anywhere”的，需要多个不同平台版本的编译器，源程序也是如此，因此开发者比较痛苦。

### 3) 通过虚拟机实现兼容性

这种方式就是以Java为代表的技术路线。使用这种机制实现兼容性的成本相当低，开发者只需要开发一次源代码，只需要在某一种硬件平台上编译通过，然后便可以在任何目标平台上运行（说任何平台确实夸张了，事实上JVM也并没有通吃全部的平台，但是基本实现了几大主流硬件平台和操作系统的兼容性），开发者所需要做的仅仅是在不同的硬件平台上部署不同的虚拟机而已。这种成本相比诸如C、C++之类的高级编程语言动不动就要针对不同目标平台而修改底层API，已经几乎可以忽略不计，如果非要为它定一个市场价的话，那一定是跳楼价了。

---

当然，除了这3种主要的方式，还有其他各种与此类似的实现方案，例如Python作为一门脚本语言，不需要编译便可以直接运行，但是其在本质上还是与JVM虚拟机类似，因为Python的编译过程由解释器代劳，解释器将python编译为中间语言（类似于Java的字节码）并基于中间语言实现跨平台，在这一点上，与JVM完全保持一致。

---

比较这3种方式，可以发现，实现兼容性的思路基本是从原始笨拙向着高度智能的方向发展，最终达到不需要为了兼容性而额外写多份源代码的目标，即“write once”。

“write once”这一目标的实现，带来的回报或者红利是十分丰厚的（现实社会也是这样，一个良好的顶层社会结构设计，往往会造成社会全方面的飞速发展，并还会额外馈赠很多东西）。“首当其冲”的红利就是开发者再也不用感知底层API了，省去了这么个麻烦并且痛苦的过程，

直接使得类似于 Java 这样非常高级的编程语言的学习门槛大幅度降低，甚至近乎于零，虽然这可能会导致开发者在底层实现原理领域存在盲区，不利于技术成长，但是对于商业项目而言，屏蔽了底层实现的高级编程语言的开发效率会成倍地加快。相比之下，在宏观层面上有这么一门近乎智能的编程语言，的确加速了信息化的进程，至于底层硬件的那些东西就交给社会大分工去调节吧，总会有一部分人会专注于那一块像原始森林一样深厚广袤的领域，比如笔者便一直专注于底层的研究。

“write once” 所馈赠的另一个回报就是——数据结构不再直接依赖于物理机器。程序是算法与数据结构的有机结合体，不管是算法还是数据结构，最终都需要被物理机器所理解。任何一门编程语言，构成算法基础的指令都会被还原成机器指令，只有物理机器才有能力执行各种各样的算法指令。虽然广义的数据结构是一种抽象的概念，甚至很多时候人们为了演算某种算法，会将抽象的数据结构具象化，使用形象化的语言符号来描述这种抽象的数据结构，但是在工程实践中，数据结构必然需要由一种具体的编程语言去实现。这种实现的背后，仍然是物理机器在支撑，仍然离不开机器指令。简单到定义一个字节变量，复杂到定义一个复合型的结构体，其实都是需要 CPU 首先能够识别对应的机器指令然后才能吭哧吭哧运行实现的。

詹爷在编程领域算得上是一个比较有品味的人，当年设计 Java 时，对这种全新的编程语言期望很高，不仅要求能够实现跨平台兼容，还要求融入当时非常时髦的设计思想——面向对象。Java 语言里面除了接口和枚举，其他的数据结构都是类型（事实上对于 JVM 而言，接口和枚举也是类型），或者反过来说，Java 中的类型都是一种专门的数据结构，整个 Java 程序都由数据结构所组成，Java 算法也由数据结构的动作所驱动，所以数据结构可以说是 Java 的核心。

詹爷为了实现算法 “write once, run anywhere” 的伟大目标，最终选择了使用字节码中间语言这条技术路线，通过 Java 字节码来统一描述算法。在 “字节码” 这条技术的康庄大道上，詹爷一溜烟跑到头，一竿子插到底，詹爷将字节码的思想贯彻得非常彻底，不仅将程序算法 “字节码” 化，连同数据结构一起被 “字节码” 化。这种 “字节码” 化的数据结构，由于以 “面向对象” 为设计宗旨，因此在面向用户（即开发者）的一端是十分人性化的，同时由于被 “字节码” 化，因此在面向机器的那端，是十分不可理喻的，因为字节码化的数据结构不认识机器，机器也不认识它，这便实现了数据结构对物理机器的去依赖。

总而言之，詹爷使出了浑身解数，不仅让算法变成一种充满趣味的活儿，解除了算法对物理机器指令的依赖，让天下没有难以兼容的平台，而且还让数据结构也脱离对底层硬件平台的依赖，使广大开发者受益匪浅，可以像定义抽象的数据结构那样，随心所欲任意组装所需要的结构体。算法与数据结构对物理硬件的双重去依赖，使得 Java 程序具有简单易学的特点，并最终保证了 Java 程序跨平台兼容的彻底性。

詹爷让数据结构脱离底层机器约束的思路其实并不复杂，例如下面这个 Java 类，是一个描

述 iPhone 6s 手机参数的数据结构：

**清单： Iphone6s.java**

**作用： 使用 Java 类描述 iPhone 6S 手机参数**

```
public class Iphone6s {
    int length = 138;           //长度(毫米)
    int width = 67;            //宽度(毫米)
    int height = 7;             //高度(毫米)
    int weight = 143;           //重量(克)
    int ram = 2;                //ram容量(G)
    int rom = 16;               //rom容量(G)
    int pixel = 1200;           //摄像头像素(万)
}
```

编译这个可以被看成是一个纯粹的“数据结构”的 Java 类，得到的字节码格式的“数据结构”信息如下：

**清单： Iphone6s.java**

**作用： 使用 Java 类描述 iPhone 6S 手机参数**

```
Compiled from "Iphone6s.java"
public class Iphone6s extends java.lang.Object
    SourceFile: "Iphone6s.java"
    minor version: 0
    major version: 50
    Constant pool:
    const #1 = class          #2;      // Iphone6s
    const #2 = Asciz           Iphone6s;
    const #3 = class          #4;      // java/lang/Object
    const #4 = Asciz           java/lang/Object;
    const #5 = Asciz           length;
    const #6 = Asciz           I;
    const #7 = Asciz           width;
    const #8 = Asciz           height;
    const #9 = Asciz           weight;
    const #10 = Asciz          ram;
    const #11 = Asciz          rom;
    const #12 = Asciz          pixel;
    const #13 = Asciz          <init>;
    const #14 = Asciz          ()V;
    const #15 = Asciz          Code;
    const #16 = Method         #3.#17; // java/lang/Object."<init>":()V
    const #17 = NameAndType   #13:#14;// "<init>":()V
    const #18 = Field          #1.#19; // Iphone6s.length:I
    const #19 = NameAndType   #5:#6;// length:I
    const #20 = Field          #1.#21; // Iphone6s.width:I
    const #21 = NameAndType   #7:#6;// width:I
```

```

const #22 = Field      #1.#23; // Iphone6s.height:I
const #23 = NameAndType #8:#6;// height:I
const #24 = Field      #1.#25; // Iphone6s.weight:I
const #25 = NameAndType #9:#6;// weight:I
const #26 = Field      #1.#27; // Iphone6s.ram:I
const #27 = NameAndType #10:#6;// ram:I
const #28 = Field      #1.#29; // Iphone6s.rom:I
const #29 = NameAndType #11:#6;// rom:I
const #30 = Field      #1.#31; // Iphone6s.pixel:I
const #31 = NameAndType #12:#6;// pixel:I
const #32 = Asciz      LineNumberTable;
const #33 = Asciz      LocalVariableTable;
const #34 = Asciz      this;
const #35 = Asciz      LIphone6s;;
const #36 = Asciz      SourceFile;
const #37 = Asciz      Iphone6s.java;

```

编译后所得到的这样一种使用字节码进行格式化的数据结构，你很难轻易地将其与具体的机器指令关联起来，而事实上这些“字节码”化的指令也的确无法被物理机器直接识别，这便实现了 Java 数据结构对物理机器的去依赖。不过这种字节码格式终归仍然需要被物理机器理解并执行，这就需要将字节码格式最终转换为最底层的机器指令，这种转换的机制将在后续章节详细分析。

詹爷使尽了浑身解数折腾出 Java 这么一款面向对象的语言，从算法与数据结构的角度看，也是具有重要意义的。其巨大作用便在于，让程序员可以专注于业务逻辑，而不需要将很多精力“浪费”在各种底层平台的接口兼容上。这对于商业项目而言，无疑具有无比巨大的实实在在的价值。

## 3.2 数据类型简史

前面讲过，虽然 Java 的类型信息通过字节码进行了格式化，但是这种编译后的格式化了的数据类型并不能直接被物理机器识别，没有哪台物理机器能够在读取了 Java 字节码文件内容之后就能直接在内存中构建出对应的结构体。Java 的这种数据结构实现机制相比于同时代的其他编程语言很独特。

数据结构与物理机器之间有着千丝万缕的联系。程序算法告诉物理机器应该怎么做，而数据结构则告诉物理机器拿什么去做。而事实上物理机器的大部分指令也的确同时包含了“怎么做”和“做什么”这两部分。例如下面这段汇编程序：

```
subl$32, %esp
movl$138, -28(%ebp)
movl$16, -8(%ebp)
```

在这段汇编程序中，出现了2个熟悉的身影——sub和mov（如果前面章节的内容你认真看过的话）这两个指令分别表示减和传送。这两个指令都是典型的带有“立即数”的机器指令，指令本身告诉物理机器要“怎么做”，而跟在指令后面的立即数则告诉物理机器“做什么”。例如对于subl\$32,%esp这条指令而言，sub是物理机器指令，告诉机器要开始做减法运算了，减多少呢？后面的立即数\$32就是答案。对谁做减法运算呢？再后面的%esp寄存器就是答案。

在这个例子里，立即数32和寄存器esp都可以认为是一种数据类型，只不过这是最简单的数据类型，简单到它就是它“自己”，而不包含任何其他元素或成员，这种情况下，数据类型已经直接退化成了数据。一般而言，所谓数据结构，至少也应该是复合结构，但是在物理机器层面，已经没有“复合”这种概念了，CPU只能操纵位或者存储单位，哪怕再稍微复杂一点点都不能支持。

在这里，有一个关键的点需要明确，那就是数据结构与数据类型的关系。简单地说，数据结构的实现需要依赖数据类型的支持，例如使用C语言定义一个单向链表，往往需要多种数据类型的参与才能实现。任何一种编程语言，简单到最原始的机器和汇编，复杂到现代的高级语言，不管哪一种，其实都能够实现任何一种复杂的数据结构，道理很简单，任何编程语言所定义的任意复杂的数据结构最终都要依靠机器指令才能实现，这本身便说明机器指令能够实现任意复杂的数据结构。区别在于实现的难易程度，编程语言所能支持的数据类型越多，则实现复杂的数据结构的成本往往越小。由于机器指令和汇编语言不支持多样化的和复杂的数据类型，因此给程序设计带来了诸多困扰，软件设计师虽然也能够使用基本数据类型通过在内存中建立映射关系从而实现结构化的内存空间布局以支撑对应的算法逻辑，但是仍然无法使用一种高级和直观的数据视图去形象化地表现复杂的对象。例如，构建一种简单的结构体来描述iPhone 6S信息，仅仅使用汇编这种并不支持任何数据类型的编程语言也能够实现，但是其在语法层面并不能够提供简单的支持，只能写成下面这种方式：

**清单：iphone6s.s**

**作用：使用汇编为iPhone 6S 定义“数据结构”**

main:

```
pushl  %ebp
movl%esp, %ebp
subl$32, %esp
movl$138, -28(%ebp)      //长度(毫米)
movl$67, -24(%ebp)       //宽度(毫米)
movl$7, -20(%ebp)        //高度(毫米)
movl$143, -16(%ebp)      //重量(克)
```

```

movl $2, -12(%ebp)          //ram 容量 (G)
movl $16, -8(%ebp)          //rom 容量 (GB)
movl $1200, -4(%ebp)         //摄像头像素 (万)

movl $0, %eax
leave

ret

```

可以看出，这种方式毫无结构感可言，如果不是编写程序的人特意说明这段程序是在定义一个关于 iPhone 6S 的数据结构，读者压根儿就看不出来。

在计算机技术问世之初，其就像太阳一样，散发出万丈光芒，继 18 世纪末 19 世纪初伟大的工业革命变革之后，再一次照亮了全世界，给人类带来了希望和信心。可是，当人们意识到无法随心所欲地定义清晰明了的数据结构的问题之后，发现整个天空都暗了。其时软件领域的天空上漂浮着一朵乌云，这朵乌云叫作“数据结构化”。如果这朵乌云始终不被驱散，那么整个软件界都会处于一片黑暗中，找不着出路。

这个问题不解决，虽然各种各样降低时间复杂度和提升性能的算法依然会陆续登上历史舞台，但是没有一种直观的数据结构的视图化编程方式，一切都按照机器的“意志”行事，那么所谓的“数据结构”便只是一种仅仅存在于观念和意识中的概念，看不见摸不着写不出读不懂，始终不能落地，就像一个如幽灵般捉摸不定的量子。因此算法的发明速度也会迟缓，而离开了算法的软件技术，也必定举步维艰得不到真正的发展。可以说，软件在短短几十年里得到迅速发展，与高级编程语言在语法层面提供的自定义数据类型的能力是息息相关的，相关的佐证俯拾皆是，并且这种相关性在 Java 语言里尤其明显，例如 Java 直接内建了 hashmap、list、set、stack 等高级数据结构类型，这些类型在大部分 Java 工程中被大量使用，而在汇编或者 C 程序中，开发者往往要自己实现这些高级的数据结构与算法，虽然可以使用第三方包，但是很多第三方包的作者并不保证程序的健壮性和兼容性。

所以现代高级编程语言的发展历程是伴随着对数据类型的日益强大的支持的过程。

回到 IT 历史发展的早期阶段，虽然伟大的先辈们从算法层面对汇编进行了抽象，逐渐发展出了高级编程语言，使开发者不用直面机器指令或者寄存器之类的硬件，但是在数据结构层面一直还没有诞生出这种意识，一切算法仍然围绕着二进制位、存储单位（字节）、字、双字等最原始的数据类型。那时候所谓的二进制位、字节甚至都不能算是数据类型，因为当时根本就没有这个概念。

不过乌云终归是浮云，在聪明的前辈们的努力之下，这朵浮云神马都不是，很快便烟消云散了。IT 科学家们拨开云雾，赫然发现“人工智能”这扇大门，并且科学家们很快便找到了打开这扇智慧大门的金钥匙，于是各种蕴含着“人工智能”的高级编程语言纷纷登场，不过其过

程却是异常艰辛。

最早的语言仅支持少量的数据结构，如 Fortran 90 之前通常用数组来模拟链表及二叉树，而所谓数组，其实汇编里面也能实现，本质上而言并不属于高级编程语言所独有，因为直接使用机器指令也能模拟出一个类似于数组的内存空间。

至于 COBOL、ADA 和 PL/I 之类的早期编程语言，也没有好到哪里去，在数据类型的概念上并没有产生质的飞跃。

及至到了 ALGOL，终于在数据结构方面迈出了历史性的一大步。ALGOL 开创性地引入了用户自定义类型的概念，注意，是自定义类型哟！虽然 ALGOL 仅提供少数的基本类型以及少量灵活的结构定义操作符，却允许开发者自主设计一种数据结构。显然，这是数据类型发展过程中最重要的进步。

到了 1967 年，Simula 67 首次提出“类型”的概念，把数据和被允许施与数据上的操作结合为一个统一体，从而成为现代“抽象数据类型”的开端及第一个“面向对象语言”。

通过以上历史进程可见，数据类型并非天生就有，而是在无数前辈们的努力下逐渐提出来的一种概念。从一开始完全没有数据类型的概念，到数据类型概念的萌芽，再到能够自定义数据类型，最终到“抽象数据类型”，其中对数据类型的每一次认识上的加深，都对软件领域历史的发展产生革命性的作用，每一次都极大地提升了编程语言对客观世界的抽象和描述能力，编程语言变得越来越易用，越来越智能，越来越“人性化”。

到 C 语言之父 Richard 发明出 C 语言的时候，人们已经可以轻轻松松地定义和实现数据类型了，已经能够描述非常复杂的客观事物了。例如，同样是为 iPhone 6S 定义一个结构，使用 C 语言的“结构体”可以这样写：

**清单：iphone6s.c**

**作用：使用 C 语言为 iPhone 6S 定义数据结构**

```
// 声明一个结构体
struct iphone6s{
    int length;           //长度（毫米）
    int width;            //宽度（毫米）
    int height;           //高度（毫米）
    int weight;           //重量（克）
    int ram;              //ram 容量（GB）
    int rom;              //rom 容量（GB）
    int pixel;             //摄像头像素（万）
};

// 结构体使用
int main() {
```

```

struct iphone6s iphone;//定义变量

// 结构体初始化
iphone.length=138;
iphone.width=67;
iphone.height=7;
iphone.weight=143;
iphone.ram=2;
iphone.rom=16;
iphone.pixel=1200;

return 0;
}

```

C 语言为了实现“数据结构”的可视化，定义了“结构体”这种类型，通过结构体，开发者可以将任意类型的基本数据组合到一起，形成复杂的数据结构。不仅如此，C 语言的结构体还能嵌套使用，结构体里面包含结构体，从而可以定义出多维度的深度树形结构。例如，将上面的结构体改造一下，变成嵌套结构体：

**清单：iphone6s.c**

**作用：**使用 C 语言为 iPhone 6S 定义数据结构

```

struct volume{
    int length;           //长度（毫米）
    int width;            //宽度（毫米）
    int height;           //高度（毫米）
};

struct basic{
    int weight;           //重量（克）
    int ram;               //ram 容量（GB）
    int rom;               //rom 容量（GB）
    int pixel;              //摄像头像素（万）
};

// 嵌套的结构体
struct iphone6s{
    struct volume volume;
    struct basic basic;
};

int main(){
    struct iphone6s iphone;

    struct volume volume;
    volume.length=138;

```

```

volume.width=67;
volume.height=7;

struct basic basic;
basic.weight=143;
basic.ram=2;
basic.rom=16;
basic.pixel=1200;

iphone.volume=volume;
iphone.basic=basic;

return 0;
}

```

在本例中，iPhone 6S 变成了一个嵌套的结构体，基于此例，可以定义任意复杂的结构体。由于 C 语言中有了“结构体”这根“如意金箍棒”，想让它变啥它就变啥，世界上几乎没有它描述不了的事物，从此数据结构有了合适的落脚点，看得见摸得着了。软件业由此大步前进。

除了 C 语言，其他算得上“高级”的编程语言，也几乎都能够从语法层面支持“数据结构”的概念，开发者按照给定的格式去定义数据结构，机器一定能够正确识别。例如在 VB 中可以使用下面这种方式去定义数据结构：

**清单： test.vb**

**作用： 使用 VB 语言定义数据结构**

```

type student
    id as string
    name as String
    age as int
end type

```

其他常见的编程语言诸如 Python、Perl、C++、PHP 等都支持自定义数据结构。

前面讲过，Java 中的“数据结构”被“字节码”格式化了，最终的结果是 Java 的数据结构解除了对物理机器的依赖。但是 C 语言中的数据结构对物理机器是有依赖的，这种依赖在 C 语言源代码被编译后便出现了。在 Linux 平台上，上面那段 iphone6s.c 的 C 语言数据结构被编译后，会直接变成下面这段汇编程序：

**清单： iphone6s.s**

**作用： C 语言数据结构被编译后的汇编程序**

```

main:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp

```

```
movl $138, -28(%ebp)
movl $67, -24(%ebp)
movl $7, -20(%ebp)
movl $143, -16(%ebp)
movl $2, -12(%ebp)
movl $16, -8(%ebp)
movl $1200, -4(%ebp)

movl $0, %eax
leave

ret
```

可以看到，C 语言中的数据结构被编译后，直接被转换成了对应平台上的机器指令，因此物理机器是可以直接识别并运行的。虽然被转换后，原本 C 语言中结构体数据的类型信息被彻底抹去，被彻底打回最原始的类型，变得不可理解，如果直接阅读编译后的汇编代码，你肯定不知道原来这段程序是一个数据结构体。

因此，C 语言的结构体类型相比汇编，就如同自动挡轿车相对于手动挡。C 程序通过编译器，将人类可理解的结构体这种具象化的高级概念，转换成了底层非结构化的低级概念，这种转换是自动的，是编译器智能分析的结果。虽然在今天看来，这种自动化并不一定算得上“智能化”，因为当今“智能化”这个词已经有了更加复杂的含义，但是在当年那种编译原理尚未完全成熟的年代，能够做到这种自动转换，已属不易！也许再过 30 年，当人工智能与神经网络的原理和技术十分成熟时，人们再回过头来看今天所谓的大数据等概念，也会觉得一点都算不上“智能”和“高级”呢。

如果说 C 语言中的数据结构依赖于物理机器，倒不如说依赖于特定平台上的特定编译器更加合适，而事实上也的确如此，不同硬件平台上的编译器各不相同，这些不同的编译器将同一个 C 语言程序编译成了各自对应的底层指令。

C 语言在编译期实现了数据结构的解释，编译器实现了对 C 程序中的数据类型的识别、解释，并最终翻译成了机器指令可以识别的数据类型。操作系统加载编译后的二进制程序执行，在内存中构建出与源程序中所定义的完全一致的数据结构，这给人一种错觉，以为物理机器能够认识 C 程序中所定义的各种复杂的结构类型。Java 语言则与此完全不同，Java 编译器虽然在编译时也能够准确分析出 Java 的类型信息，但是编译后的 Java 字节码却并不能直接由物理机器执行，因此 Java 语言的类型信息并不完全在编译期维护，而是推迟到了运行期。原理与之相似的编程语言包括同为面向对象编程语言的 SmallTalk、JavaScript、C# 等。

知道了数据类型的发展简史，各位朋友真应该为自己生在这样一个伟大的时代而暗自庆幸，因为现代的各种高级编程语言都有很丰富的数据类型可以选择，如果实在不满足要求，还可以

自定义。而在几十年前，这根本就是一件不可想象的事情。尤其对于 Java 程序员而言，Java 语言不仅类型丰富，而且面向对象，用它来编程，真的是一件简单而快乐的事情。

总而言之，高级编程语言分别从算法和数据类型这两个层面对早期编程语言进行改进，改进的最终结果是：在算法层面，早期编程语言需要使用几十甚至几百行指令才能实现的逻辑，现代高级编程语言往往只需要一两行代码便可完成。而在数据类型层面，现代高级编程语言往往直接将各种常见的结构类型集成到 SDK 中提供给开发者使用，并在此基础上允许开发者重写或自定义。而 Java 在这两方面都达到了一定的历史高度甚至巅峰。

### 3.3 Java 数据结构之偶然性

Java 丰富的数据类型和面向对象的特性，为广大软件开发者能够自由描述各种复杂的客观事物打开方便之门。不过相比于 C/C++ 等面向过程的语言，或者虽然面向对象但是不够彻底的语言，Java 是实实在在地将“面向对象”贯彻到底的语言。这种贯彻所带来的结果就是，如果不将整型、字符型、浮点型等基本数据类型划分为数据结构的一种，那么 Java 语言中的数据结构就只退化成一种——类型。不管一种被描述的事物多么简单，也不管其如何复杂，在 Java 语言中，事物的一切属性都必须被“打包”为 Java 类型，Java 类型已经成为语法层面的强制约束。而在其他语言中，可能还存在诸如结构体之类的概念，在 Java 中统统不存在（Java 中允许定义枚举类型可能已经很开恩了），尤其在 C/C++ 语言中，甚至允许专门为函数指针定义一种特别的类型。

相比于 JVM 执行引擎在技术上狭窄的选择面，Java 的这种数据结构的技术实现其实带有一定的历史偶然性。下面先回顾一下 JVM 执行引擎的技术实现为何是必然的。

#### 1. JVM 执行引擎的必然选择

在讨论 Java 的面向对象思想之前，先让我们再次回顾下 JVM 的执行引擎。

詹爷当初创立 Java 门派，便立志于改变智能家电程序的开发方式，希望能够开发出一门“write once, run anywhere”的语言。因此，Java 语言天生便口含“兼容”的金钥匙，具体如何兼容？假设用 C 语言开发一款程序，为了实现兼容性，我们需要做表 3.1 中所列这些事情

表 3.1 C 程序的兼容性处理方式

开发	不同的平台上，分别开发不同的代码。例如访问线程、读写文件、申请内存等，不同平台、不同操作系统所提供的 API 一定是不同的，因此所调用的接口也不尽相同
编译	得在不同的平台上，安装不同的编译器，然后才能执行不同的编译命令编译程序
打包	在不同的平台上，需要选择适应该平台的软件包和编译器，对程序进行打包
运行	不同的平台上，需要有不同的安装环境，例如 Windows 上需要各种 dll 动态链接库，Linux 上需要各种 so 支持

而如果选择使用 Java 开发一款软件，同样在开发、编译、打包和运行期，所需要做的兼容性措施如表 3.2 所示。

表 3.2 Java 程序的兼容性处理方式

开发	无论在何种平台上，只需开发同一套程序，无需为底层平台的不同而做特殊处理，Java 提供统一的文件访问、内存申请、线程操作、GUI 开发等接口
编译	不同的是，在不同的平台上需要安装不同的 JDK 相同的是，只需执行同一种编译指令，便能编译程序
打包	无论哪种平台，无论是通过命令行执行打包，还是通过 IDE 执行打包，命令行或 IDE 菜单完全一致
运行	只要安装了 JVM 虚拟机，便能运行，不再依赖任何其他 dll 或 so

对于软件开发，生命周期的绝大部分时间都花费在开发上，而 Java 程序只需要编写一次，就能在“任何”平台上运行。由此带来的效率提升不可谓不显著。Java 除了在开发上保持一致性，编译和打包、运行都保持了高度的兼容性和独立性，开发者所做的只是针对不同的平台，安装适当的 JVM 和 JDK。

詹爷当年为了实现这种能够带来巨大效益的编程语言，在如何执行程序的问题上花费了无数心血，以其深厚功力，以其对机器语言、汇编、操作系统的深刻理解，最终终于实现了 JVM 执行引擎。前文花了大量篇幅讲解詹爷为什么最终会选择字节码的方式作为中间语言，以及为什么最终会选择将字节码实时翻译成汇编程序这种技术路径。通过前文的分析可知，JVM 做出这种技术上的选择，是综合考虑了各种技术方案的优势和弊端、特点和特性，最后所得出的最优解。

一句话，JVM 选择这种技术路径是必然的，即使换了另一个大师级的人物来开发，最终也一定会设计成现在 JVM 的样子。只是可能 CallStub 这种分水岭不同，或者给出的诸如 entry\_point 这样的例程不同，或者采用的 jit 即时编译器的编译算法不同。但是在现有的物理硬件设备不变的前提下，基本不可能翻腾出别的浪花。这种必然性是各种技术、各种路径碰撞的必然结果，所谓“万宗归一”，不仅在武侠、宗教世界中存在这种终极的“道”，在软件编程领域，也同样存在这样无形的“天道”。

但是，天道不可畏，并非不能突破。每一次的技术革命，都是人类突破自我、超越天道的勇敢尝试。所以，研究 JVM，表面看似乎并没有什么用，但是，通过窥探 JVM 的本质，无形之中便获得了一种“道”，这种道的能力会使你如龙之遨游四海，如鹰之翱翔天空，这种道能够给你一种智慧，一种举一反三的能力，会使你研究其他领域的速度更快，领悟力更高。举个不恰当的例子，如果将来某一天，计算机硬件得到革命性的创新，那时候计算机不仅能识别 0 和 1，而且能够识别 1000、10000 了，到了那一天，如果你掌握了现在的 0 和 1 理论，那么你就可根据 0 和 1 时代的经验，在 1000、10000 的基础上构建一系列汇编语言、操作系统、高级编程语言，以及像 JVM 这种能够跨越多个底层平台和硬件的虚拟机。这就是道的力量。

## 2. Java 面向对象之技术偶然性

如果说 JVM 为了兼容各种异构硬件和平台而作出的这种技术抉择是一种必然，那么 Java 选择面向对象的编程方式和内存管理模型（数据结构总是与内存管理机制联系在一起）便具有一定的偶然性和随机性了。对于给定的执行引擎，可以使用若干种编程方式来使用这种执行引擎。所以，Java 的面向对象机制与 JVM 的执行引擎并不是强绑定的关系，这如同运载火箭一样，不同的卫星可以使用同一种运载工具发射升空，同一种卫星也可以使用不同的运载工具发射升空。Java 程序经编译后的字节码文件，既可以使用 HotSpot 执行，也可以使用 jrockit 解释，或者被 IBM JVM 运行。而随着 JVM 越来越强大、稳定和高效，JVM 本身也成为一种平台性产品，诸多编程语言可以被编译成 JVM 字节码文件，因此，对于同样一个 Java 程序编译后生成的字节码中间文件，你可以选择使用不同的执行引擎来运行，而同一款 JVM 执行引擎，可以解释 Java、PHP 等源码被编译后生成的字节码。

所以，当年詹爷选择使用编译解释的方式来执行 Java 程序，但是其实 Java 语言原本可以不是面向对象的，它可以选择面向过程，或者面向函数，或者其他。但是，在那个年代，面向对象的编程方式正如日中天，如火如荼，C++、Java、VB、C#，还有一些脚本语言，例如，JavaScript、Python、Perl、PHP、Delphi 等也是面向对象的。

退一万步讲，即使 Java 语言因为追随潮流，选择了面向对象的编程方式，也完全可以不选择现在的内存模型和垃圾回收机制。但是由于，Java 要求能够在运行时通过反射获取到类型信息，因此最终 Java “被迫” 选择了现在的这种内存模型。

C++/Delphi 等编程语言也具有面向对象的特性，但是到了运行期已经完全消除了类型概念，并且也无法在运行期“反射”到类型的成员变量、方法等信息，但是 Java 可以。但是 Java 为什么能做到在运行期动态反射到类型的成员变量和方法信息呢？说白了也很简单，在 JVM 加载 Java 类的时候，就将 Java 类的类元信息（类元信息即变量和方法）保存到内存中，这样在运行期直接读取目标内存中的数据便能获取到相应的信息。这种类元信息，其实也是一种打包好的

数据结构模板，并且在运行期可被识别。而相比于 C++/Delphi 等同样是面向对象的语言，类模板信息早在编译期便被完全擦除，而 Java 语言在编译后仍然保留了类型的内部结构信息，并且类型结构信息被带到了运行期。

不过话说回来，能够做到在运行期动态反射出类型结构信息，并不能成为编程语言选择拥有面向对象特性的必然理由，更不能因此就非要实现自动内存管理（自动垃圾回收），例如，C++这种面向对象的编程语言通过 RTTI（运行时类型识别）也能够做到在运行期识别类型，但是 C++ 的内存仍然需要开发者自己去释放。从这个角度看，Java 的自动内存管理机制就显得并不是必须的。然而 Java 选择具备运行时类型识别的特性本身便从一个十分隐晦的层面制约了 Java 必须选择成为一门面向对象的编程语言，为何？类型本身就是一种“闭包”的技术手段，只有先从语法层面实现了“闭包”，才能实现“对象”的概念，否则，何来的属性、成员变量、类方法一说？类型是实现将若干属性和动作打包成为一个整体对象进行统一识别的策略。如果 Java 像 C++那样，类型不作为属性和方法封装的唯一手段，开发者可以随心所欲地在类的外面定义变量和函数，那么对于这部分数据的“运行时识别”必然是一个难题，可能需要通过类似 namespace 或者 filename 这样的机制去实现动态反射了，但是这种反射思想都让人头大，不容易啊！换了是你，你会怎么从技术上让其落地和实现呢？

因此，从运行期动态反射的角度看，Java 语言选择成为一门彻底的面向对象语言，绝对是偶然中的必然。相对来说，Java 的自动内存管理（垃圾回收）机制就带有一种随机性。不过，这也不一定正确。当一门编程语言实现了完全的闭包语法策略（使用类型包装可以认为是闭包的一种），便自然而然具备了自动内存管理的技术基础，或者说实现自动内存管理更加容易。所以闭包便成为很多具备自动内存回收特性的编程语言的语法基础，例如 GO 语言、Phthon、JavaScript 等，虽然大家具体实现闭包的手段不同，但是殊途同归，都是为了能够让虚拟机在自动回收内存时尽量简单。

总体而言，Java 的面向对象和自动内存管理的特性实现，因为要实现运行时类型识别的目标，因此在偶然中带有必然性，而必然中又孕育出偶然性因素。

## 3.4 Java 类型识别

生活在现代世界的 Java 程序员是快乐的。Java 程序员编写的任何类，Java 虚拟机都能够在运行期识别出来，这实在是让人激动。那么 Java 虚拟机究竟是如何做到这一点的呢？一切奥秘都隐藏在 Java 源程序被编译后生成的字节码文件中。Java 类在编译期生成的字节码有其特定的组织规律，Java 虚拟机在加载类时，对编译期生成的字节码信息按照固定的格式进行解析，一

一步一步解析出字节码中所存储的类型结构信息，从而在运行期完全还原出原始的 Java 类的全部结构。

### 3.4.1 class 字节码概述

每一个 Java 类被编译后会生成一个对应的.class 字节码文件，要想研究 JVM 加载 Java 类的原理，首先必须熟练掌握 Java 类被编译成的.class 格式文件结构。下面将从几个方面来描述字节码的组成格式。

#### 1. class 文件构成基础

在 class 字节码文件中，数据都是以二进制流的形式存储。这些字节流之间都严格按照规定的顺序排列，字节之间不存在任何空隙，对于超过 8 位的数据，将按照 Big-Endian（大端）的顺序存储，即高位字节存储在低的地址上面，而低位字节存储到高地址上面。其实这也是 class 文件跨平台的关键，因为 PowerPC 架构的处理器采用 Big-Endian 的存储顺序，而 x86 系列的处理器则采用 Little-Endian（小端）的存储顺序，因此为了 class 文件在各种异构处理器架构下能够保持统一的存储顺序，虚拟机必须设置统一的存储规范。

#### 2. class 文件的 10 个组成部分

class 字节码文件是采用类似 C 语言的结构体来存储数据的，主要有两类数据项：无符号数和表。无符号数用来表述数字、索引引用以及字符串等，在.class 文件中主要使用的无符号数包括 u1、u2、u4 和 u8，分别代表 1 字节、2 字节、4 字节和 8 字节的无符号数。而表是由多个无符号数以及其他表组成的复合结构。

一个 class 字节码文件主要由以下 10 部分组成：

- MagicNumber
- Version
- Constant\_pool
- Access\_flag
- This\_class
- Super\_class
- Interfaces
- Fields
- Methods

## ◎ Attributes

这些数据的类型和长度都是不同的，用一个数据结构可以表示如下：

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

要注意的是，在 JVM 内部其实并没有定义这样的一种数据结构，这里使用类似 C 语言中的结构体方式来表示 Java 字节码文件的结构，完全是为了展示不同属性的数据类型和长度。

### 3. class 文件中的各组成字段简单说明

#### 1) MagicNumber

MagicNumber 是用来标志 class 文件的，位于每一个 Java class 文件的最前面 4 个字节，值固定为 0xCAFEBAE。注意，这个是十六进制数值，并非字符串“CAFEBAE”，其对应的二进制数是 11001010 11111110 10111010 10111110B，一共占 32 位比特即 4 字节。虚拟机加载 class 文件时会先检查这 4 字节，如果不是 0xCAFEBAE，则虚拟机拒绝加载该文件，这样就可以防止加载非 class 文件而造成虚拟机崩溃。

#### 2) Version

Version 字段由 2 个长度都为 2 字节的字段组成，分别是 Major Version 和 Minor Version，代表当前 class 文件的主版本号和次版本号。随着 Java 技术的不断发展，Java class 会增加一些新的内容来支持 Java 语言的新特性。同时，不同的虚拟机支持的 Java class 文件的版本范围是不同的，所以在加载 class 文件之前可以先看看该 class 文件是否在当前虚拟机的支持范围之内，避免加载不支持的 class 文件。高版本的 JVM 可以加载低版本的 class，但反之就不行。

目前已发布的 Version 包括：1.1(45)、1.2(46)、1.3(47)、1.4(48)、1.5(49)、1.6(50)、1.7(51)。对于 JDK 1.6 编译出来的 class file，其版本号是 0x00000032（查看 class file 的第 5~8 这 4 字节，下文会详述），转换为十进制数就是 50。

如果使用低版本的 JDK 编译 Java 程序，然后使用高版本的 JRE 执行 class file，能够顺利执行。反之，如果使用高版本的 JDK 编译 Java 程序，而使用低版本的 JRE 执行 class file，则 JVM 会抛出类似于“java.lang.UnsupportedClassVersionError: Unsupported major.minor version 50”这样的异常。当出现这种异常信息时，首先查看 JVM 的版本，查看方法很简单，进入 jvm\bin\，执行 Java -version 命令即可。其次是查看 class 的 Version（即编译该 Java 类的 JDK 版本），查看方法是，使用二进制文件编辑器打开对应的 class 文件，根据其开头第 4~8 个字节的二进制值，换算出对应的十进制数，即可得到其版本号。

### 3 ) 常量池 ( Constant\_pool )

常量池信息从 class 文件的第 9 字节开始。

首先是 2 字节（即第 9 和第 10 两个字节）的长度字段 constant\_pool\_count，表明常量池包含了多少个常量。

接下来的二进制信息描述 [constant\_pool\_count-1] 个常量，常量池里放的是字面常量和符号引用。

字面常量主要包含文本串以及被声明为 final 的常量。符号引用包含类和接口的全局限定名、字段的名称和描述符、方法的名称和描述符，因为 Java 语言在编译的时候没有连接这一步，所有的引用都是运行时动态加载的，所以就需要把这些引用的信息保存在 class 文件里。

字面常量根据具体的类型分成字符串、整型、长整型、浮点型、双精度浮点型这几种基本类型。

符号引用保存的是引用的全局限定名，所以保存的是字符串。

### 4 ) Access\_flag

主要保存当前类的访问权限。

### 5 ) This\_cass

主要保存当前类的全局限定名在常量池里的索引。

### 6 ) Super\_class

主要保存当前类的父类的全局限定名在常量池里的索引。

### 7 ) Interfaces

主要保存当前类实现的接口列表，包含两部分内容：interfaces\_count 和 interfaces[interfaces\_count]。

- ◎ interfaces\_count 指的是当前类实现的接口数目。
- ◎ interfaces[] 是包含 interfaces\_count 个接口的全局限定名的索引的数组。

#### 8 ) Fields

主要保存当前类的成员列表，包含两部分的内容：fields\_count 和 fields[fields\_count]。

- ◎ fields\_count 是类变量和实例变量的字段的数量总和。
- ◎ fields[] 是包含字段详细信息的列表。

#### 9 ) Methods

主要保存当前类的方法列表，包含两部分的内容：methods\_count 和 methods[methods\_count]。

- ◎ methods\_count 是该类或者接口显式定义的方法的数量。
- ◎ method[] 是包含方法信息的一个详细列表。

#### 10 ) Attributes

主要保存当前类 attributes 列表，包含两部分内容：attributes\_count 和 attributes[attributes\_count]。

这些属性在字节码文件中的具体存储方式，下面会通过一个示例来讲解，毕竟概念还是挺抽象的嘛^\_^。

### 3.4.2 魔数与 JVM 内部的 int 类型

由于魔数在字节码文件中占 4 字节，并且其数据值固定不变，一直都是 0xCAFEBABE，因此 JVM 内部使用 u4 这种自定义的数据类型存放魔数。u4 这种数据类型的定义如下：

清单：/src/share/vm/utilities/globalDefinitions.hpp

作用：u4 类型定义

```
typedef uint u4;
```

juint 也是自定义类型（注意，不是 junit），但是这种类型是平台相关的，在 linux 平台上，juint 定义如下：

清单：/src/share/vm/utilities/globalDefinitions\_gcc.hpp

作用：juint 类型定义

```
typedef uint32_t juint;
```

`uint32_t`仍然是自定义类型，定义如下：

清单：/src/share/vm/utilities/globalDefinitions\_gcc.hpp

作用：`juint`类型定义

```
#ifndef _UINT32_T
#define _UINT32_T
typedef unsigned int uint32_t;
#endif
```

由此可知，`uint32_t`最终所代表的类型是 `unsigned int`，在 32 位或 64 位系统平台上，`unsigned int`都占 4 字节，正好能够存放下魔数信息。所以，JVM 内部的 `u4` 数据类型能够存放 4 字节。而 JVM 内部除了 `u4`，还定义了另外 3 种常用的数据类型：

- ◎ `u1`，该数据类型占 1 字节，在 Linux 平台上所代表的 C 语言类型是 `unsigned char`。
- ◎ `u2`，该数据类型占 2 字节，在 Linux 平台上所代表的 C 语言类型是 `unsigned short`。
- ◎ `u8`，该数据类型占 8 字节，在 Linux 平台上所代表的 C 语言类型是 `unsigned long`。

这里额外对 `uint32_t` 之类的数据类型做一个补充说明，按照 POSIX 标准，一般整型对应的 `*_t` 类型为：

- ◎ 1 字节，`uint8_t`
- ◎ 2 字节，`uint16_t`
- ◎ 4 字节，`uint32_t`
- ◎ 8 字节，`uint64_t`

这 4 种基本数据类型在遵循 C99 标准的 C 语言中进行了内置定义，因此开发者可以直接使用。这些数据类型都有一个特点，那就是以 `_t` 结尾，这其实表示这些数据类型并不是什么新的类型，它们只是使用 `typedef` 为类型起的别名而已。原理虽然很简单，但是作用却很大，比如 C 中没有 `bool`，于是在一个软件中，一些程序员使用 `int`，一些程序员使用 `short`，会比较混乱，最好就是用一个 `typedef` 来定义，如：

```
typedef char bool;
```

`uint8_t` 之类的意义也正是如此，是为了让代码能够被更好地维护，并且大家都使用同一套标准。

### 3.4.3 常量池与 JVM 内部对象模型

常量池是 Java 字节码文件中比较重要的概念，是整个 Java 类的核心所在，因为常量池中记录了一个 Java 类的所有成员变量、成员方法和静态变量与静态方法、构造函数等全部信息，包

括变量名、方法名、访问标识、类型信息等。

JVM 内部定义了一个 C++ 类型 `constantPoolOop` 来记录解析后的常量池信息（本书默认以 Hotspot 1.6 版本的源码作为研究对象，下同。Hotspot 后续版本略有变化，类名有部分修改，但总体上变化不大）。`constantPoolOop` 其实是别名，其原始的类型是 `constantPoolOopDesc`。在 `/src/share/vm/oops/oopsHierarchy.hpp` 中进行了这两种类型的转换：

```
typedef class constantPoolOopDesc* constantPoolOop;
```

而事实上，JVM 内部为了在运行期描述 Java 类的类型信息和内部结构，定义了很多以 `Desc` 结尾的 `oop` 类，详细的定义见 `/src/share/vm/oops/oopsHierarchy.hpp` 文件。这个源码中所定义的类，便用于实现 Java 的面向对象特性，很重要哦！各位道友可以先进入这个源码感受一下，第一次看肯定没感觉，但是看多了总会“日久生情”，然后才能深入了解。

既然提到了 `oop` 的概念，就不得不提 JVM 内部对 Java 对象的表示模型，这个模型便是著名的“`oop-klass`”模型。

### 1. oop-klass 模型

Hotspot 虚拟机在内部使用两组类来表示 Java 的类和对象。

- ◎ `oop`(ordinary object pointer)，用来描述对象实例信息。
- ◎ `klass`，用来描述 Java 类，是虚拟机内部 Java 类型结构的对等体。

JVM 内部定义了各种 `oop-klass`，在 JVM 看来，不仅 Java 类是对象，Java 方法也是对象，字节码常量池也是对象，一切皆是对象。JVM 使用不同的 `oop-klass` 模型来表示各种不同的对象。而在技术落地时，这些不同的模型就使用不同的 `oop` 类和 `klass` 类来表示。由于 JVM 使用 C/C++ 编写，因此这些 `oop` 和 `klass` 类便是各种不同的 c++ 类。对于 Java 类型与实例对象，JVM 使用 `instanceOop` 和 `instanceKlass` 这 2 个 C++ 类来表示，这 2 个类后文会逐步分析到，此处略过不表，各位道友可以先打开 HotSpot 的源码进去看下这 2 个类，找找“感觉”。

这里贴出 HotSpot 内部所定义的 `oop` 大全：

清单：`/src/share/vm/oops/oopsHierarchy.hpp`

作用：描述 HotSpot 中的 `oop` 体系

<code>typedef class oopDesc*</code>	<code>oop;</code>
<code>typedef class instanceOopDesc*</code>	<code>instanceOop;</code>
<code>typedef class methodOopDesc*</code>	<code>methodOop;</code>
<code>typedef class constMethodOopDesc*</code>	<code>constMethodOop;</code>
<code>typedef class methodDataOopDesc*</code>	<code>methodDataOop;</code>
<code>typedef class arrayOopDesc*</code>	<code>arrayOop;</code>
<code>typedef class objArrayOopDesc*</code>	<code>objArrayOop;</code>

<b>typedef class</b>	typeArrayOopDesc*	typeArrayOop;
<b>typedef class</b>	constantPoolOopDesc*	constantPoolOop;
<b>typedef class</b>	constantPoolCacheOopDesc*	constantPoolCacheOop;
<b>typedef class</b>	klassOopDesc*	klassOop;
<b>typedef class</b>	markOopDesc*	markOop;
<b>typedef class</b>	compiledICHolderOopDesc*	compiledICHolderOop;

也许是为了简化变量名,JVM统一将最后的 Desc 去掉,全部处理成以 Oop 结尾的类型名。例如对于 Java 类中所定义的方法,JVM 使用 methodOop 去描述 Java 方法的全部信息;对于 Java 类中所定义的引用对象变量,JVM 则使用 objArrayOop 来保存这个引用变量的“全息”信息。

Hotspot 使用 klass 来描述 Java 的类型信息。Hotspot 定义了如下几种类型信息。

清单: /src/share/vm/oops/oopsHierarchy.hpp

作用: 描述 HotSpot 中的类型信息

```

class Klass;
class instanceKlass;
class instanceMirrorKlass;
class instanceRefKlass;
class methodKlass;
class constMethodKlass;
class methodDataKlass;
class klassKlass;
class instanceKlassKlass;
class arrayKlassKlass;
class objArrayKlassKlass;
class typeArrayKlassKlass;
class arrayKlass;
class objArrayKlass;
class typeArrayKlass;
class constantPoolKlass;
class constantPoolCacheKlass;
class compiledICHolderKlass;

```

纵观以上 oop 和 klass 体系的定义,可以发现,无论是 oop 还是 klass,基本都被划分为来分别描述 instance、method、constantMethod、methodData、array、objArray、typeArray、constantPool、constantPoolCache、klass、compoiledICHolder 这几种模型,这几种模型中的每一种都有一个对应的 xxxOopDesc 和对应的 xxxKlass。通俗而言,这几种模型分别用于描述 Java 类类型和类型指针、Java 方法类型和方法指针、常量池类型及指针、基本数据类型的数组类型及指针、引用类型的数组类型及指针、常量池缓存类型及指针、Java 类实例对象类型及指针。HotSpot 认为使用这几种模型,便足以勾画 Java 程序的全部:数据、方法、类型、数组和实例。

那么 oop 到底是啥,其存在的意义究竟是什么?其名称已经说得很清楚,就是普通对象指针。指针指向哪里?指向 klass 类实例。直接这么说可能比较难以理解,举个例子,若 Java 程

序中定义了一个类 ClassA，同时程序中有如下代码：

```
ClassA a = new ClassA();
```

当 Hotspot 执行到这里时，会先将 ClassA 这个类型加载到 perm 区（也叫方法区），然后在 Hotspot 堆中为其实例对象 a 开辟一块内存空间，存放实例数据。在 JVM 加载 ClassA 到 perm 区时，JVM 就会创建一个 instanceKlass，instanceKlass 中保存了 ClassA 这个 Java 类中所定义的一切信息，包括变量、方法、父类、接口、构造函数、属性等，所以 instanceKlass 就是 ClassA 这个 Java 类类型结构的对等体。而 instanceOop 这个“普通对象指针”对象中包含了一个指针，该指针就指向 klassInstance 这个实例。在 JVM 实例化 ClassA 时，JVM 又会创建一个 instanceOop，instanceOop 便是 ClassA 对象实例 a 在内存中的对等体，主要存储 ClassA 实例对象的成员变量。其中，instanceOop 中有一个指针指向 instanceKlass，通过这个指针，JVM 便可以在运行期获取这个类实例对象的类元信息。

## 2. oopDesc

既然讲到了 oop，就不得不提 JVM 中所有 oop 对象的老祖宗——oopDesc 类。上述列表里的所有 oopDesc，诸如 instanceOopDesc、constantPoolOopDesc、klassOopDesc 等，在 C++ 的继承体系中，最终全都来自顶级的父类——oopDesc（JDK8 中已经没有 oopDesc，换成了别的名字，但是换汤不换药，内部结构并没有什么太大的变化）。Java 的面向对象和运行期反射的能力，便是由 oopDesc 予以体现和支撑。看起来很神秘不是？但是看看其结构，会发现简单得似乎与其所具备的能力有点不相称：

清单：/src/share/vm/oops/oop.hpp

作用：oopDesc 定义

```
class oopDesc {
    friend class VMStructs;
private:
    volatile markOop _mark;
    union _metadata {
        wideKlassOop _klass;
        narrowOop     _compressed_klass;
    } _metadata;
    // Fast access to barrier set. Must be initialized.
    static BarrierSet* _bs;
    //...
}
```

抛开友元类 VMStructs，以及用于内存屏障的 \_bs，oopDesc 类中只剩下了 2 个成员变量（友

元类并不算成员变量): \_mark 和 \_metadata。其中 \_metadata 是联合结构体, 里面包含两个元素, 分别是 wideKlassOop 与 narrowOop, 顾名思义, 前者是宽指针, 后者是压缩指针。关于宽指针与窄指针这里先简单提一句, 主要用于 JVM 是否对 Java class 进行压缩, 如果使用了压缩技术, 自然可以节省出一定的宝贵内存空间。

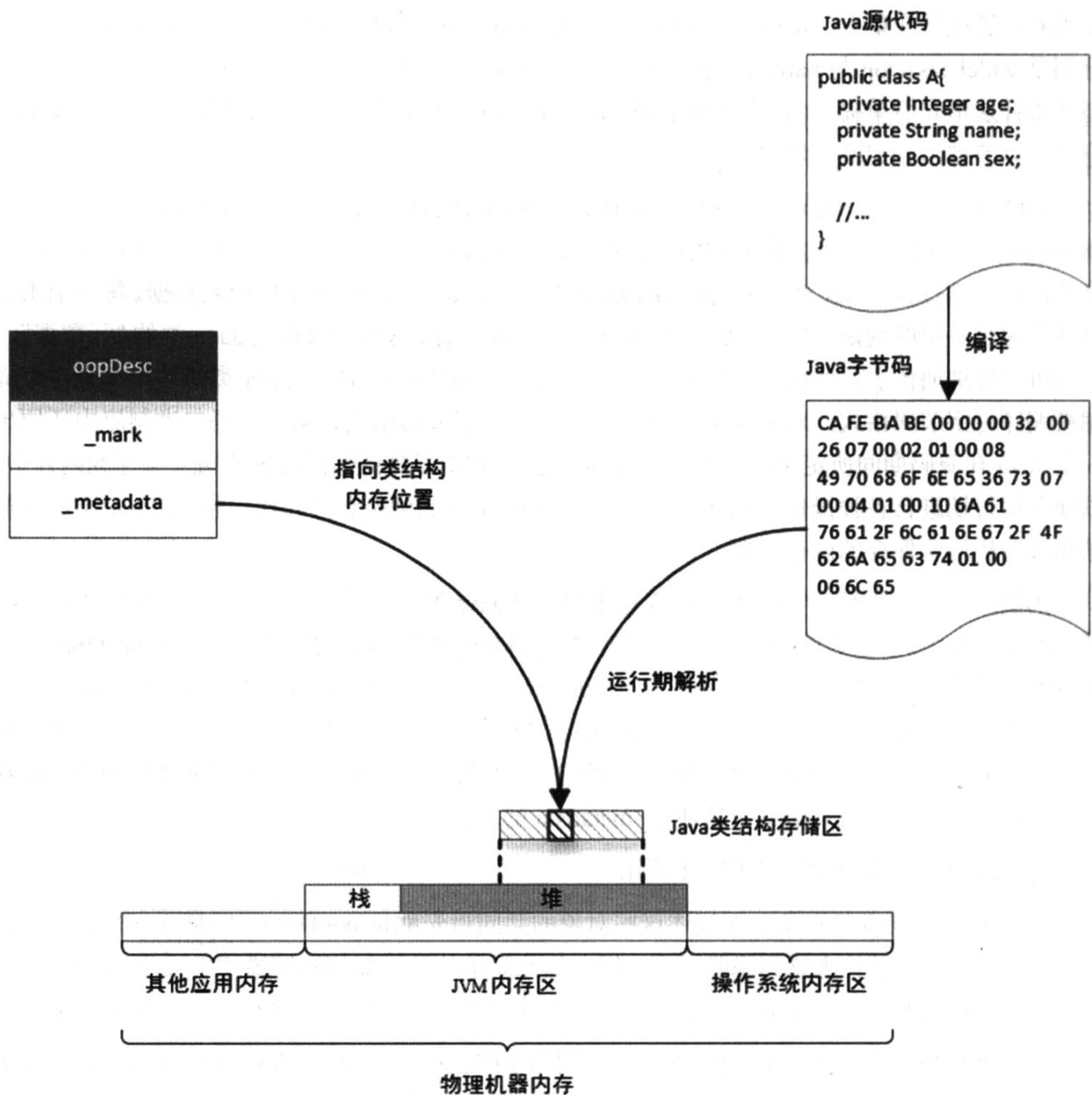
oopDesc 的这 2 个成员变量的作用很简单, \_mark 顾名思义, 似乎是一种标记, 而事实上也的确如此, Java 类在整个生命周期中, 会涉及到线程状态、并发锁、GC 分代信息等内部标识, 这些标识全都打在 \_mark 变量上。而 \_metadata 顾名思义也很简单, 用于标识元数据。每一个 Java 类都会包含一定的变量、方法、父类、所实现的接口等信息, 这些均可称为 Java 类的“元数据”, 其实可以更加通俗点, 所谓的元数据就是在前面反复讲的数据结构。Java 类的结构信息在编译期被编译为字节码格式, JVM 则在运行期进一步解析字节码格式, 从字节码二进制流中还原出一个 Java 在源码期间所定义的全部数据结构信息, JVM 需要将解析出来的结果保存到内存中, 以便在运行期进行各种操作, 例如反射, 而 \_metadata 便起到指针的作用, 指向 Java 类的数据结构被解析后所保存的内存位置。

仍然以上一节所举的实例化 ClassA 这个自定义 Java 类的例子进行说明。当 JVM 完成 ClassA 类型的实例化之后, 会为该 Java 类创建对应的 oop-klass 模型, oop 对应的类是 instanceOop, klass 对应的类是 instanceKlass。上一节讲过, instanceOop 内部会有一个指针指向 instanceKlass, 其实这个指针便是 oopDesc 中所定义的 \_metadata。klass 是 Java 类型的对等体, 而 Java 类型, 便是 Java 编程语言中用于描述客观事物的数据结构, 而数据结构包含一个客观事物的全部属性和行为, 所以叫做“类元”信息, 这便是 \_metadata 的本意。

\_metadata 的作用可以参考图 3.1 所示。

JVM 内部一切都是对象, 而描述这些对象的共同祖先就是 oopDesc, 但是 oopDesc 的结构却异常简单, 这不禁让人感到隐隐不安, 担心如此简单的结构如何能够描述 Java 类复杂的数据结构。这个问题暂且放下不表, 后面会揭示出事实真相。让我们将目光移回到常量池对象。

常量池这种对象虽然并不是由程序员在源代码中定义的, 但是在 Java 类被编译后, 其成为编译后的字节码中最核心的对象, 对于这种重量级的“人物”, JVM 当然也有对应的类型去描述它。上文讲过, JVM 使用 constantPoolOop 这种类型来保存常量池的信息, 但是 constantPoolOop 内部其实还是借助于 typeArrayOop 这种类型才可以对常量池进行描述。constantPoolOop 的结构定义在 /src/share/vm/oops/constantPoolOop.hpp 中, 本书后面章节会详细讲解, 这里就先不展开, 大家可以自己进入这个源码文件感受下。

图 3.1 oopDesc 的`_metadata` 指向

### 3. Java 结构与其他编程语言结构之比较

当 C、C++ 和 Delphi 等程序被编译成二进制程序后，原来所定义的高级数据结构都不复存在了，当 Windows/Linux 等操作系统（宿主机）加载这些二进制程序时，是不会加载这些语言中所定义的高级数据结构的，宿主机压根儿就不知道原来定义了哪些数据结构、哪些类，所有的数据结构都被转换为对特定内存段的偏移地址。例如 C 中的 Struct 结构体，被编译后不复存

在，汇编和机器语言中没有与之对应的数据结构的概念，CPU 更不知道何为结构体。C++ 和 Delphi 中的类概念被编译后也不复存在，所谓的类最终变成内存首地址。而 JVM 虚拟机在加载字节码程序时，会记录字节码中所定义的所有类型的原始信息（元数据），JVM 知道程序中包含了哪些类，以及每个类中所关联的字段、方法、父类等信息。这是 JVM 虚拟机与操作系统最大的区别所在。

正因为 JVM 需要保存字节码中的类元信息，所以 JVM 最终就自然而然地演化出了 OOP-KLASS 这种二分模型，KLASS 用于保存类元信息，而 OOP 用于表示 JVM 所创建的类实例对象。KLASS 信息被保存在 PERM 永久区，而 OOP 则被分配在 HEAP 堆区。同时 JVM 为了支持反射等技术，必须在 OOP 中保存一个指针，用于指向其所属的类型 KLASS，这样 Java 开发者便能够基于反射技术，在 Java 程序运行期获取 Java 的类型信息，例如反射 Java 类的类型、父类、字段、方法等信息，而这是其他很多语言所不具备的能力。也正因为 JVM 的这种能力，让程序员能够非常方便地开发出具有运行时动态特性的程序，例如可以根据类型来设计更为抽象和优雅的工厂模式，例如可以在运行期动态创建一个新的类型并实例化，同时执行其方法（ASM 字节码编程技术）。由于这些特性，使得 Java 语言成为了互联网时代的各种中间件、各种框架实现的首选语言，进而有力地促进了大数据时代的架构发展。虽然 Java 由于天然的缺陷，在运行性能、内存占用等方面无法与本地语言相媲美，但是在互联网工业时代，生产效率是第一生产力，对于一门编程语言，没有什么能够比易学、易用又能够迅速开发出一款不错的工业产品更具有吸引力。尤其在移动互联网时代，整个人类社会要的是能够在短时间内生产出足够多样化的，能够满足各类日常需求的软件产品，而随着硬件越来越廉价，JVM 在性能和空间两方面的缺点得以弥补。可以说，JVM 的成功不是偶然的，它是整个人类社会发展的必然结果，是 IT 行业自我革命、自我快速发展的必然结果，JVM 是人类对 IT 生产力提升的迫切需求与软件编程门槛高这一对矛盾相互促进而演化出的折中方案。虽然 JVM 不够完美，但已然足够。

## 3.5 大端与小端

Java 是跨平台的语言，并且支持丰富多样的数据类型。但是在不同的平台上，数据在寄存器、内存、磁盘上的存储格式并不相同——准确地说，是数据存储的顺序不同。这种不同的存储顺序衍生了计算机底层的 2 个概念——大端与小端。

以 JVM 解析 Java class 字节码文件中的魔数为例进行分析。魔数占 4 字节，并且位于字节码文件头部。JVM 解析魔数，只需读取出字节码流开始的 4 字节即可。JVM 读取魔数的代码如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：魔数解析

```
u4 magic = cfs->get_u4_fast();
```

cfs 的类型是 ClassFileStream，当 JVM 能够运行到 classFileParser.cpp:: parseClassFile()时，JVM 便已读入当前正在解析的 Java 字节码的文件流。

ClassFileStream 的结构如下：

清单：/src/share/vm/classfile/classFileStream.hpp

作用：classFileStream 类结构

```
class ClassFileStream: public ResourceObj {  
private:  
    u1* _buffer_start;  
    u1* _buffer_end;  
    u1* _current;  
    char* _source;  
    bool _need_verify;  
}
```

这几个字段的含义都相当清晰明了。其中，\_current 指针字段值指向 Java 字节码流中当前已经读取到的位置，当 ClassFileStream 类刚刚初始化时，\_current 指针指向 Java 字节码流的第一个字节所在的内存位置，在 JVM 解析字节码的后续流程中，随着解析的进行，该指针不断向前移动。由于魔数被第一个解析，因此解析之后，\_current 指针往前移动 4 字节步长，我们看 cfs->get\_u4\_fast()的逻辑：

清单：/src/share/vm/classfile/ classFileStream.hpp

作用：从字节码流中读取 4 字节内容

```
u4 get_u4_fast() {  
    u4 res = Bytes::get_Java_u4(_current);  
    _current += 4;  
    return res;  
}
```

---

注意，在这个方法中，由于从字节码流中读取了 4 字节的内容，因此\_current 的值被增加了 4，由于\_current 是 u1\*类型的指针，该指针所指向的数据类型是一个字节，因此将\_current 的值增加 4，其实就是增加了 4 字节的长度（相信稍微有点 C 语言指针基础的同学都能看得懂）。

在这个方法里，最关键的就是 u4 res = Bytes::get\_Java\_u4(\_current)这句代码，这句代码才真正执行了从字节码流中读取 4 字节的任务，该方法是一个 CPU 架构相关的接口，在 x86 架构上的逻辑如下：

清单: /src/cpu/x86/vm/bytes\_x86.hpp

作用: 从内存指定位置读取 4 字节并将其转换为 JVM 内部 u4 类型

```
static inline u4 get_Java_u4(address p) {
    return swap_u4(get_native_u4(p));
}
```

可以看到, 在 get\_Java\_u4()方法内部连续调用了 2 个函数, 分别是 get\_native\_u4()和 swap\_u4()。先看 get\_native\_u4()方法, 该方法定义如下:

清单: /src/cpu/x86/vm/bytes\_x86.hpp

作用: 从内存指定位置读取 4 字节并转换为 u4 类型

```
static inline u4 get_native_u2(address p) {
    return *(u4*)p;
}
```

这个方法内部直接返回\*(u4\*)p, 按照取值运算符的运算优先级, 这句表达式的运算顺序为 \*((u4\*)p), 即先将指针 p 转换为 u4\*类型的指针, 接着对转换后的指针进行取值运算, 同时将运算后的结果转换为 u4 类型。

在这里有一个很关键的问题需要注意, 由于 Java 字节码流通常比较大, 随便一个简单的 Java 类都会包含几百字节, 稍微复杂点就需要以 KB 来计了, 而当 JVM 刚开始解析魔数时, 此时 p 指针指向字节码流的最开始的内存位置, 后面还有几百字节, 因此使用一个指向首地址的指针可以循环读取后面的几百字节。而每次根据一个指针所能读取到的字节数则取决于指针的类型, 如果指针是 char\*, 则通过\*p 可以读取 1 字节的内容; 如果指针是 int \*, 则通过\*p 可以读取 4 字节的内容。在 JVM 解析魔数时, 原本人参是 cfs.\_current 变量, 其类型是 u1\*, 但是在 get\_native\_u2()方法里被强制转换为了 u4\*, 这相当于将指针的范围扩大, 这样通过\*p 能够一次读取 4 字节内容(这段内容很基础, 专业人士不要嫌弃哟, 毕竟本书是面向广大 Java 道友的, 有关 C/C++的基础知识讲解, 笔者觉得对没有 C/C++基础的道友而言可能很实惠。同样, 本书在解析其他源码时, 也会经常顺带着介绍一些 C/C++的基础知识)。

而事实上, JVM 内部自定义了好几种基本类型, 包括 u1、u2、u4、u8, 它们所代表的字节位数分别是 1、2、4、8。每次 JVM 从 Java 字节码流中读取相应位数的字节时, 只需将 cfs.\_current 指针转换为对应的 u1\*、u2\*、u4\*、u8\*类型即可。但是早期的 CPU, 并不是随随便便就能从指定的内存位置访问任意字节长度的数据, 早期的 CPU 是严格按照字节对齐的要求读取的, 只是从 x86 开始便能支持了。

从 Java 字节码文件中解析出魔数信息很简单, 但是一旦将一个简单的问题放到一个底层的、跨平台的上下文环境中时, 问题便立刻变得复杂。魔数的解析便是一个活生生的例子, 其复杂性全部体现在最关键的 swap\_u4()函数中。该函数是跨平台的, 因此需要解决兼容性, 对于 x86

平台，该函数定义如下：

清单：/src/os\_cpu/linux\_x86/vm/bytes\_linux\_x86.inline.hpp

作用：大端小端转换

```
inline u4 Bytes::swap_u4(u4 x) {
    #ifdef AMD64
        return bswap_32(x);
    #else
        u4 ret;
        __asm__ __volatile__ (
            "bswap %0"
            : "=r" (ret)      // output : register 0 => ret
            : "0"  (x)       // input  : x => register 0
            : "0"           // clobbered register
        );
        return ret;
    #endif // AMD64
}
```

这是一段内嵌了汇编的 C 函数，该函数的作用是进行大端和小端的转换。对于平常以应用开发为主的 C 和 C++ 程序员，以及以使用 Java 语言为主的开发者们，可能并不了解大端和小端的概念，而在网络协议、跨平台兼容性等开发领域，则大端和小端是必须要掌握的一项基本功。下面笔者尝试从简单的概念入手，由浅入深地介绍大端和小端的概念、由来及应用。

### 3.5.1 大端和小端的概念

关于“大端”和“小端”名词的由来，有一段有趣的故事：

故事来自于 Jonathan Swift 的《格列佛游记》。Lilliput 和 Blefuscu 这两个强国在过去的 36 个月中一直在苦战。战争的原因，大家都知道，吃鸡蛋的时候，原始的方法是打破鸡蛋较大的一端，可那时的皇帝的祖父由于小时候吃鸡蛋，按这种方法把手指弄破了，因此他的父亲，就下令，命令所有的子民吃鸡蛋的时候，必须先打破鸡蛋较小的一端，违令者重罚。然后老百姓对此法令极为反感，期间发生了多次叛乱，其中一个皇帝因此送命，另一个丢了王位，产生叛乱的原因就是另一个国家 Blefuscu 的王国大臣煽动起来的，叛乱平息后，就逃到这个帝国避难。据估计，先后几次有 11 000 余人情愿死也不肯去打破鸡蛋较小的一端吃鸡蛋。这个其实在讽刺当时英国和法国之间持续的冲突。Danny Cohen，一位网络协议的开创者，第一次使用这两个术语指代字节顺序，后来就被大家广泛接受。

大端和小端的概念之所以由一位网络协议开创者提出，是因为其实大部分人在实际的开发中都很少会直接和字节序打交道，唯有在跨平台以及网络程序中，才会涉及到一个叫做“字节序”的问题，并且这是一个必须被考虑的基础性问题。字节序，顾名思义，就是指字节的顺序，通俗而言就是数值大于一个字节类型的数据在内存中的存放顺序(一个字节的数据当然就无需谈顺序的问题了)。在各种计算机体系结构中，对于字节、字等的存储机制有所不同，通信双方交流的信息单元(比特、字节、字、双字等)的存储顺序不同，因此需要考虑双方数据的传送顺序。如果传送顺序达不成一致，通信双方将无法进行正确的编/译码从而导致通信失败。目前在各种体系的计算机中通常采用的字节存储机制主要有两种：Big-Endian 和 Little-Endian，翻译过来就是所谓的大端和小端。

标准的 Big-Endian 和 Little-Endian 的定义如下：

- ◎ Little-Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。
- ◎ Big-Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。
- ◎ 网络字节序，TCP/IP 协议中使用的字节序通常称为网络字节序，TCP/IP 各层协议将字节序定义为 Big-Endian。

如果之前没有接触过大端和小端，可以看个例子，以建立起对大端和小端概念的初步认识。例如一个值为 0x01020304 的 int 整型数据写入地址为 0x005071 的内存位置，由于该整数占 4 字节，因此需要连续使用 4 个存储单元来存储这个整数，但是存储的方式可以有两种，其中第一种策略是整数的高位字节存储在这 4 个连续存储单元的高位存储单元，整数的低位字节存储在低位存储单元。先解释下什么是整数的高低位(懂的同学可以直接略过)，例如 0x01020304，其对应的十进制数是 16909060，对于这个十进制数而言，左边的数字是高位，右边的数字是低位，同样，对于十六进制 0x01020304 而言，也是左侧的代表高位，右侧的代表低位。

按照第一种方式来存储，内存布局如下：

内存地址:	0x005071	0x0050712	0x005073	0x005074
存储的值:	0x04	0x03	0x02	0x01

可以看到，在这种存储策略下，数据增长的方向与内存地址增长的方向是相同的。这种方式不方便人类阅读，但是机器读起来却着实很“爽”，因为这样的存储顺序与机器的解读顺序是一致的。这种存储策略就叫做“小端”。

对应的，第二种存储策略与之相反，数据的高位字节存储在低位存储单元上，数据的低位字节存储在高位存储单元上。还是刚才的数据和内存位置，按照现在这种策略存储后的内存布

局如下：

内存地址:	0x005071	0x0050712	0x005073	0x005074
存储的值:	0x01	0x02	0x03	0x04

现在这种方式，数据字节增长的方向与内存位置增长的方向是相反的，这种方式有点类似于字符串的存储顺序，并且也很符合人类的阅读习惯。这种策略就叫做“大端”。

### 3.5.2 大小端产生的本质原因

有时候计算机也是挺幽默的，也会玩“时光向左，幸福向右”之类浪漫的事。但是计算机实在不是吃饱了撑的玩这种闲情逸致，内存存储顺序的向左向右，实则是由寄存器引起的。

在计算机体系结构中，内存由存储单元构成，一个存储单元的长度是一个字节，即每个存储单元都对应着一个字节，能够存储 8 比特数据。但是在 C 语言和很多其他高级语言中，除了 8 比特的 char 之外，还有 16 比特的 short 型、32 比特的 int 型与 64 比特的 long 型（int 和 long 具体所占二进制位数要看具体的编译器和 CPU 平台架构）。虽然物理内存的存储单位是 1 字节，但是现代计算机总线线宽和寄存器的宽度往往都大于 1 个字节，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，寄存器宽度都是大于一个字节的，这就造成寄存器宽度与内存存储单元宽度之间的不一致性。在软件程序的很多操作中，都会涉及数据在内存和寄存器之间的传送，例如，如果你的 C 语言程序中包含 int x=26 这样的代码，那么最终 CPU 需要先为 x 分配堆栈内存空间，然后将 26 这个立即数传入寄存器，再通过寄存器传送到 x 所在的内存位置。或者如果你的 C 语言程序中包含 int y=x 这样的代码，那么 CPU 需要先将 x 的值从内存读取到寄存器，再将数据从寄存器传送到 y 所在的内存位置。在计算机中，并不支持直接将数据在不同的内存之间传送，更不支持将数据直接从内存传送到外部设备，例如磁盘或网络端口。CPU 唯一支持不同部件之间的直接数据传送只有寄存器到寄存器了。由于在高级编程语言（相对于汇编语言而言）中并不能直接操作寄存器，所有的数据传送的指令以及针对寄存器读写的指令都被封装成面向变量的编程，而变量的存储介质是内存，因此可以这么讲，高级编程语言中的所有数据传送指令都必须经过寄存器的中转。

寄存器的宽度越大，就意味着 CPU 传送数据的能力越强，如果寄存器只能容纳 8 比特，那么 CPU 一次指令只能传输 1 字节，而如果寄存器能够容纳双字节，那么一次 CPU 指令就能传输 2 字节，这无形中提升了效率。虽然寄存器的宽度可以提高，但是内存存储单元的宽度却是一直不变的，一直都只有 1 字节，因此对于宽度达到双字节的寄存器可以一次性从内存中读取 2 个连续的存储单元的值，或者一次性向 2 个连续的内存存储单元写入数据。

对于一个占2字节的整数，例如0x0102，数据本身是区分高字节和低字节的，靠近左侧的字节为高字节，反之则为低字节。而一个双字节的寄存器也会区分高低位，对高低位的不同定位会带来数值结果的不同。假设双字节的寄存器中从左至右所存储的字节分别是0x01和0x02，如果将寄存器的左端定位为高字节端，则寄存器所存储的数值就是0x0102，对应十进制的258。而如果将寄存器的右端定位为高字节端，那么寄存器所存储的值就是0x0201，对应的十进制数是513。由此可见，以相同顺序存储的同样一段数据，如果所标定的高低位不同，则最终所代表的数值是完全不同的。由于不同厂家所生产的CPU标准不同，因此大家对于究竟将寄存器的左端还是右端标定为高字节位，并没有一个统一的标准。目前Intel的80x86系列芯片是唯一还在坚持使用小端的芯片，而MIPS和ARM等芯片要么全部采用大端的方式储存，要么提供选项支持大端——可以在大小端之间切换。

因此，大小端的问题，本质上是由寄存器引起的（当然，软件系统也能引起所谓的大小端问题，完全看人为的定义），并最终在内存的存储顺序上得以反映出来。举个例子来说明。

假设在C程序中包含int x=0x0102这样一句代码，在大端CPU架构平台上运行到这句代码时，CPU首先将立即数0x0102保存到双字节的寄存器中（假设寄存器宽度为2字节），由于以大端的方式存储，因此寄存器中从低位到高位所存储的字节分别是0x01和0x02。接着CPU将寄存器中的数据传送到x变量所在的内存位置，传送后，x变量所在的2个连续的内存存储单元中的值从低位到高位也分别是0x01和0x02。

而对于同样的程序，如果运行在小端CPU架构平台上，则寄存器从低位到高位所存储的字节分别是0x02和0x01，将其拼接起来得到的数据是0x0201，与原来的数据0x0102的字节位正好相反，于是最终保存到内存中，在内存中从低位到高位所保存的数值就是0x02和0x01。

### 3.5.3 大小端验证

x86架构的CPU是属于小端CPU，大部分Windows用户都使用这种架构的CPU，笔者的机器也是这种。可以写程序来验证，示例程序如下，使用C语言编写：

清单：/src/os\_cpu/linux\_x86/vm/bytes\_linux\_x86.inline.hpp

作用：大端小端转换

```
#include <stdio.h>
int main()
{
    short x = 1;
    char c = *(char*)&x;
    if(c == 1)
    {
```

```

        printf("little endian\n");
    }
else
{
    printf("big endian\n");
}
return 0;
}

```

这段代码测试的原理很简单，由于变量 x 的数据类型是 short，在 32 位平台上占 2 字节，其值转换为十六进制是 0x0001，如果当前 CPU 是大端类型，则最终保存到内存后，内存首地址的那个存储单元中所存储的值一定是 0x00，这是因为大端 CPU 的特点就是数据的高位字节存储在低位内存单元。反之，如果 x 的内存首地址所存储的数值是 0x1，则代表当前 CPU 是小端。而要拿到变量 x 的内存首地址所在的存储单元中所存储的值，只需像本示例中那样，通过 char c = \*(char\*)&x 来获取。这句代码的含义是，首先通过&x 获取变量 x 的内存首地址，获取的结果是一个指针类型，只是指针的类型是 short\*，这个指针指向的内存范围包含 2 个存储单元，因此需要将其转换为 char\* 这种指针类型，这样最终通过\*(char\*)得到的结果数据类型才是 char，否则就变成了 short 类型。

如果上面这段测试程序还不够直观，则下面的这段程序一定能够说明问题了：

```

#include<stdio.h>

int main()
{
    char cc[6];
    cc[0]=0x11;
    cc[1]=0x22;
    cc[2]=0x33;
    cc[3]=0x44;
    cc[4]=0x55;
    cc[5]=0x66;

    char *cp=&cc;
    short cs=*(short*)cp;
    printf("cs=%d\n", cs);

    return 0;
}

```

在本示例中，通过数组 cc 往内存中连续写入 6 字节的内容，从内存地址的低位到高位分别是 0x11、0x22、0x33、0x44、0x55、0x66。通过 char \*cp=&cc 拿到 cc 数组的内存首地址，这个首地址所存放的数值是 0x11。接着通过 short cs=\*(short\*)cp 获取从 cc 内存首地址开始的连续

2个存储单元的值，并将其转换为 short 类型。如果是在大端 CPU 平台上，则 short 的值应该是 0x1122，对应的十进制数值是 4386，但是很可惜并不是这个数值，而是 8721，8721 所对应的十六进制数正是 0x2211。这说明当前 CPU 是小端，而事实上也是的，因为是 x86 平台。之所以小端 CPU 会将 0x1122 这样的内存数据解读成 0x2211，就是因为小端 CPU 认为内存中的低位代表数据的高位字节，而内存中的高位则代表数据的低位字节，因此最终 CPU 会认为这段内存中的数值是 0x2211。

### 3.5.4 大端和小端产生的场景

虽然大端小端的问题在内存、寄存器、计算机总线甚至软件中到处都存在，但是得益于整个软硬件架构的良好设计，因此在日常编程开发中，大多数程序员都不需要去关注这个问题。在单机上，不管是往内存中写入数据还是从内存中读取数据，由于所采用的标准都是同一套，要么全部是大端模式，要么全部是小端模式，因此不会产生大小端数据转换的问题。例如对于下面这段程序：

```
int x = 0x0102;  
int y = x;
```

程序运行时，先将 0x0102 赋值给变量 x，再将 x 的内存值赋值给 y。虽然在这期间，数据会发生多次从内存到寄存器，再从寄存器到内存的传送，但是并不会产生混乱，只要所使用的大端和小端模式相同。如果使用的是小端模式，则最终变量 x 所占用的 4 个连续的内存存储单元从低位到高位所存储的数据分别是 0x01、0x00、0x00、0x00，寄存器在传送数据的过程中暂存数据时，寄存器从低位到高位所存储的数据也是 0x01、0x00、0x00、0x00，而最终寄存器往变量 y 所在内存写入的数据在内存中的存储顺序也是 0x01、0x00、0x00、0x00。当你想打印变量 x 或变量 y 的值时，虽然存储顺序与实际数据的高低位完全相反，但是 CPU 的逻辑运算器是清楚这种格式的，并会将其字节反转后拼装出正确的结果显示出来。

因此在单机上由同一种模式进行数据读写时，并不会产生对数据识别的不同，这也是为什么绝大部分程序员在日常开发中都不会接触到大端和小端问题的原因。

但是当关注网络传输和文件共享时，由于数据在网络的一端写入，而由网络的另一端读取，网络两端的 CPU 架构并不总是相同的，很可能一端使用了大端模式，而另一端使用了小端模式。在这种场景下，如果不进行大端与小端模式的转换，则数据必定会出现不一致。例如下面这段示例程序，往文件中写入 int 类型的数据：

```
#include <stdio.h>  
#include <assert.h>
```

```

void main( )
{
    short test;
    FILE* fp;

    // (0x41 的 ASCII 码对应字符 A, 0x42 的 ASCII 码对应字符 B)
    test = 0x4142;
    if ((fp = fopen ("/home/test.log", "wb")) == NULL)
    {
        assert(0);
    }
    fwrite(&test, sizeof(short), 1, fp);
    fclose(fp);
}

```

在本段代码中，往文件中写入了一个 short 类型的数据，由于一个 short 类型的数据占用 2 字节的内存空间，因此最终往文件里写入了 2 字节。而这 2 字节所对应的 ASCII 字符分别是 A 和 B，因此写入文件后，使用记事本或者在 Linux 上使用 vim 或 gedit 打开文件时，应该看到的是字符 A 和 B，而不是数值（这些文本处理器默认使用 ASCII 字符集编码）。

由于笔者的机器是 x86 架构，而 x86 属于小端模式，因此当 CPU 执行 short test=0x4142 这行代码时，最终变量 test 在内存中从低位到高位的存储内容便会与 0x4142 的字节高低位顺序相反，变成 0x4241。这导致最终写入到文件中的数据的顺序也是 0x4241，写入后在 x86 平台上使用文本编辑器打开时，会发现显示的内容是 BA，而不是 AB。如果在大端 CPU 架构上使用文本编辑器查看本文件，则结果仍为 BA，这是因为文本编辑器是按字节逐个读取内容的，而寄存器在读取一个字节时，不会发生字节之间的乱序。

现在再写一段程序来读取刚才写入的文件，示例程序如下：

```

#include <stdio.h>
#include <assert.h>

#define MAXLEN 1024

int main()
{
    FILE *fp;
    fp = fopen("/home/test.log", "rb" );

    unsigned char buf[MAXLEN];

    if( fp == NULL)
    {
        printf("%s, %s", "错误", "not exit\n");
        return 0;
    }
}

```

```

}

int rc;

while( (rc = fread(buf,sizeof(unsigned char), MAXLEN,fp)) != 0 )
{
    printf("%s\n","read start");
    int readInt = *(int*)&buf;
    printf("readInt = %d\n", readInt);
}
printf("%s\n","read over");

fclose(fp);

return 0;
}

```

这段程序从刚才所写入的文件中读出 1024 个字节到缓冲区 buf 中，并强制将指针&buf 转换成 int\*类型，最终再通过 int\*指针获取 int 值。

同样在 x86 这种小端模式的 CPU 架构平台上运行这段程序，结果打印出十进制数据 16706，其对应的十六进制正好是 4142，这与原本往文件里写入的数据是完全一致的。

在本例中，由于写入和读取文件的程序都运行在小端 CPU 架构平台上，因此虽然写入文件后的数据高低字节位置被反转了，但是在同样的 CPU 架构平台上能够识别出正确的数值。但是，如果将读取文件的这段示例程序放在大端 CPU 架构平台上运行，则会发现这段程序最终会打印出 16961，即对应十六进制 0x4241。这与将上个示例程序在小端 CPU 平台上运行后写入文件并在大端 CPU 架构平台上使用文本编辑器打开后显示的字符依然是 BA 不同，这是因为文本编辑器使用 ASCII 字符集进行编码，文本编辑器逐字节读取磁盘文件里的内容，不会将文件里的字符合并转换成一个 int 类型的数据，因此不会产生字节序反转，而本示例需要 CPU 识别多个字符合并后所代表的某种数据类型的数值，在这个过程中，由于大端 CPU 与小端 CPU 所标定的高低位相反，因此对于由 2 字节所合并出来的同一个数据的识别也必定不同。

所以在编写网络协议或者编写跨平台的程序时，大端和小端的问题就显得尤其突出，开发者必须要清楚所谓大小端问题产生的根本原因并采取适当的措施进行解决，否则必定产生乱序，导致数据不一致。

但是，并不是所有的网络通信场景都需要考虑大小端问题，例如下面的示例：

```

#include <stdio.h>
#include <assert.h>

void main()
{

```

```

// 注意：这里将 short 类型换成了 char[] 数组
char test[2];
FILE* fp;

// 0x41 的 ASCII 码对应字符 A, 0x42 的 ASCII 码对应字符 B
test[0] = 0x41;
test[1] = 0x42;
if ((fp = fopen ("/home/test.log", "wb")) == NULL)
    assert(0);
fwrite(&test, sizeof(short), 1, fp);
fclose(fp);
}

```

在本示例中，写入文件的不再是 int 整数，而是变成了 char 数组，其实就是逐字节写入。依然在 x86 平台上运行本程序，运行后使用文本编辑器打开写入的文件，会发现编辑器里显示的内容变成“AB”，而不再是“BA”。为什么这一次 CPU 没有对数据高低字节位进行反转呢？其实这就涉及一个核心的问题：究竟什么场景下才会产生大小端模式带来的字节序反转问题？

其实这一问题的答案在前文已经说得很清楚。要回答一个问题，首先就要明白问题产生的本质原因。大小端问题产生的根本原因是，当寄存器需要读取或写入超过一个字节长度的数据时，由于不同 CPU 所认定的寄存器的高低位不同（只有 2 种认定，并且认定结果完全相反），而产生了所谓大小端模式。只要不使问题产生的条件成立，那么问题自然不会产生。由于程序在处理 char 类型的数据时，寄存器只需要读写 1 字节宽度的数据，因此自然不会出现所谓的字节序反转问题。这就是本示例能够不受大小端模式影响而始终生成同样顺序的字节的原因。而在上一个示例中，我们往文件里写入的是 int\* 类型的数据，但是字节序反转并不是发生在 CPU 将变量写入文件的阶段，而是发生在 CPU 为变量写入数值的前一阶段，即 CPU 将立即数读进寄存器的阶段。因为是小端 CPU，因此寄存器读取到超过 1 个字节宽度的 int 类型数据时便发生了字节序反转。由于在寄存器中数据的字节序被反转，因此最终写入变量内存时的字节序也是反转的，由此导致最终写入文件的字节序依然是反转的。

其实，在这个过程中，有一个重要的点值得关注，那就是无论在大端还是小端，调用系统 API fwrite() 和 fread() 进行读写时，似乎并不受大小端模式的影响。理解了上面的道理之后，对于这一现象就能解释了，唯一的原因就是 fwrite 与 fread 在底层也是逐字节读取和写入的，因此并不会产生字节序反转。

### 3.5.5 如何解决字节序反转

前面讨论了大小端的概念、产生原因及现象，那么，如果真的面临大小端的现实场景，该

如何解决则是一个重中之重的问题。例如，假设有一天你需要使用 C 语言开发一个网络通信协议，该协议要求兼容主流平台，那就必须要处理大小端问题。

在 Linux 平台，可以调用 bswap 这个指令进行字节序反转。我们在前文举了一个例子，定义一个包含 6 个元素的 char 数组，然后将其前两个字节合并转换成一个 short 类型的数据，现在我们改造这个例子，同时支持字节序反转：

```
#include<stdio.h>

int swap_u4(int x);

int main(){
    char cc[6];
    cc[0]=0x11;
    cc[1]=0x22;
    cc[2]=0x33;
    cc[3]=0x44;
    cc[4]=0x55;
    cc[5]=0x66;

    char *cp=&cc;
    short cs=*(short*)cp;
    printf("cs=%d\n", cs);

    int lls=*(int*)cp;
    printf("lls=%d\n", lls);

    int lld=swap_u4(lls);//这里对字节序进行反转
    printf("lld=%d\n", lld);

    int xx=0x01020304;
    char iic=*(char*)&xx;
    printf("iic = %d\n", iic);

    return 0;
}

int swap_u4(int x) {
#ifdef AMD64
    return bswap_32(x);
#else
    int ret;
    __asm__ __volatile__ (
        "bswap %0"
        :"=r" (ret)      // output : register 0 => ret
        :"0"  (x)        // input  : x => register 0
    )
    return ret;
#endif
}
```

```

        :"0"           // clobbered register
    );
    return ret;
#endif // AMD64
}

```

在本例中，定义了一个函数 swap\_u4() 用于反转字节序。在小端 CPU x86 架构平台上运行本程序，打印结果如下：

```

lls=1144201745
lld=287454020

```

lls 对应的十六进制是 0x44332211，而 lld 对应的十进制则是 0x111223344。由此可见，lld 的值被正确地还原了出来，与原始的输入值完全相等。而 lld 之所以能够被完整还原出来，是因为调用了字节序反转函数 swap\_u4()。

### 3.5.6 大小端问题的避免

---

大小端问题的本质是由于计算机底层硬件的问题，因此显得比较复杂，但是在宏观表象上，却比较容易解释和理解，绝大多数书籍和网络博客阐述的重点往往也是表面原因和现象。

但是只要遵循下面两种方式处理数据、文件、网络传输，便可以无视大小端模式：

- ◎ 在单机上使用同一种编程语言读写变量、读写文件、进行网络通信，所读到的字节序与所写入的字节序相同，反之亦成立。
- ◎ 在分布式场景下，使用同一种编程语言，在同样大小端模式的不同机器上所读写的文件与网络信息，字节序相同。

例如，在网络环境中，在一台机器上写入文件，在另一台机器上读取文件，如果这两台机器都是大端模式或者都是小端模式，则只要读取和写入时均使用同样的编程语言便能保持所读与所写的字节序是相同的。这一点对于理解 JVM 的字节序处理很重要。

### 3.5.7 JVM 对字节码文件的大小端处理

---

在讲述关于 Java 字节码文件的大小端问题处理之前，还有一个问题需要特别说明，那就是大小端问题不仅存在于计算机硬件体系中，软件中也同样存在。当然，软件中的大小端问题多是被编译器处理了，所以在绝大多数情况下，软件开发者并不需要特别关注大小端问题。Java 编译器同样在后端默默地处理了这个问题，Java 所输出的字节信息全部是大端模式，这一点对于理解 JVM 在解析字节码信息时的处理策略至关重要。

JVM在解析魔数及Java类的其他结构信息时，均需要读取字节码信息。Java源代码一般由编译器被编译为字节码文件，而Java编译器本身一般也都是用Java语言编写而成的，因此Java编译器在对Java源程序进行语法树分析并将分析结果写入字节码文件时，字节码文件中的信息存储便是大端模式，例如对于魔数，字节码文件的写入顺序一定是如下这样：

0xCA 0xFE 0xBA 0xBE

这种写入文件的顺序不会受计算机硬件体系到底是大端模式还是小端模式的影响。这是Java与其他编程语言的一个重要区别，例如，如果用由C语言开发的编译器来编译Java源代码，那么这种编译器在小端机器上生成的字节码文件中的魔数的写入顺序基本会变成下面这样：

0xBE 0xBA 0xFE 0xCA

这是因为，魔数在C语言中可以使用一个int类型的变量表示，C语言在将这个int型变量写入字节码文件之前，首先需要将魔数信息写入该变量，而由于硬件体系属于小端模式，因此从寄存器写入内存时，魔数的字节序便已经发生反转。

既然Java编译器所生成的字节码文件并不会因为计算机架构的大小端模式而受到影响，那么JVM从字节码文件中解析魔数时，为何还要处理字节序呢？在上文讲到，在分布式环境下，要避免大小端问题，只需使网络中的各个计算机节点的大小端模式都保持一致，并且所使用的编程语言也保持一致即可。而Java字节码的读写场景对这两个条件都不符合，首先是读写问题，Java编译器一般是由Java开发的，因此字节码文件的写入可以认为是由Java语言完成的，而读取字节码文件的是JVM，JVM是由C与C++混合写成的，因此Java字节码文件的写入端与读取端属于两种不同的编程语言。接着看计算机节点的大小端模式的一致性问题。由于Java语言的跨平台性，因此对于一段已经编写好的Java源代码，既可以在Windows上编译打包，也可以在Linux或者其他平台上编译打包。同样，读取Java字节码的JVM可能运行于Windows平台，也可能运行于Linux平台，因此Java字节码文件可能在Windows平台上写入，而在Linux平台上被读取。但是Java字节码文件的字节序并不受大小端模式的影响，这是由于Java语言本身属于大端模式，因此Java语言所写入的文件的字节序全部按照大端模式进行存储。如此看来，可能引起Java字节码文件的字节序读取不一致的是读取端的编程语言的大小端模式。JVM由C和C++编写，而C和C++的大小端模式默认情况下与计算机硬件平台的大小端模式保持一致，因此最终又完全取决于读取端所在的计算机硬件平台的大小端模式。如果读机器属于大端模式，则最终所读取到内存中的魔数信息便是0xCAFEBAE；反之，如果读机器属于小端模式，则最终所读取到的魔数信息便是0xBEBAFECA。很显然，如果是后者，则JVM将会校验失败，因此如果JVM运行于小端机器上，则必须对所读取出来的魔数字节序进行反转。这便是JVM最终在bytes\_linux\_x86.inline.hpp中引入inline u4 Bytes::swap\_u4(u4 x)这类函数接口的原因。

事实上，JVM 不仅在解析魔数时需要实现大小端的正确反转，而且在后续解析所有其他信息，诸如版本号、常量池、字段等时，也都实现了大小端的兼容处理。魔数占用 4 字节，因此 JVM 定义了 swap\_u4() 这样的接口，而除魔数以外的其他字节码信息，有的占用 2 字节，有的占用 8 字节，当然，也有的仅占用 1 字节。对于 2 字节和 8 字节的读取，JVM 同样定义了相应的转换接口，如下：

**清单：/src/share/vm/classfile/classFileStream.hpp**

**作用：**大端小端转换 u2 版本

```
inline u2 Bytes::swap_u2(u2 x) {
#ifdef AMD64
    return bswap_16(x);
#else
    u2 ret;
    __asm__ __volatile__ (
        "movw %0, %%ax;"           // output : register 0 => ret
        "xchg %%al, %%ah;"         // input  : x => register 0
        "movw %%ax, %0"            // clobbered registers
        :"=r" (ret)
        :"0" (x)
        :"ax", "0"
    );
    return ret;
#endif // AMD64
}
```

**清单：/src/share/vm/classfile/classFileStream.hpp**

**作用：**大端小端转换 u8 版本

```
#ifdef AMD64
    inline u8 Bytes::swap_u8(u8 x) {
        #ifdef SPARC_WORKS
            // workaround for SunStudio12 CR6615391
            __asm__ __volatile__ (
                "bswapq %0"
                :"=r" (x)           // output : register 0 => x
                :"0" (x)           // input  : x => register 0
                :"0"                // clobbered register
            );
            return x;
        #else
            return bswap_64(x);
        #endif
    }
#else
    // Helper function for swap_u8
```

```
inline u8 Bytes::swap_u8_base(u4 x, u4 y) {
    return (((u8)swap_u4(x))<<32) | swap_u4(y);
}

inline u8 Bytes::swap_u8(u8 x) {
    return swap_u8_base(*((u4*)&x), *((((u4*)&x)+1)));
}
#endif // !AMD64
```

可以看到，JVM 并没有定义一个类似于 swap\_u1()这样的转换接口，这是因为在前文分析过，大小端问题的产生条件是读取或写入超过 1 字节长度的数据，而如果从文件中一字节一字节地读取，1 字节码的读取并不会产生乱序行为，因此 JVM 并不需要针对 u1 类型的数据的读取进行大小端的兼容性处理。

本节讲述了大端与小端的概念、产生机制以及 JVM 内部的解决之道。通过这一点更加可以看出，如果没有 Java 语言，大家要实现将一个程序部署到不同的硬件平台上，是一件多么辛苦的事情，仅仅是大端与小端，就要花费很多精力去处理。

## 3.6 本章总结

到了现在，可以回答本章开始处“摘要”中的问题：数据结构是什么，为何要数据结构？

从程序的角度看，数据结构是若干数据的有机结合，一个数组、一个链表、一个堆/栈都是数据集，这些数据在内存位置上有着紧密的联系，例如，对于数组而言，相邻的两个元素在内存位置上也是彼此相邻的；而对于链表，内存空间上未必彼此相连，但是相邻的两个元素中必定有一个元素保存着一个指针指向另一个元素的内存位置。而从人类的角度看，一个特定的数据结构可以更好地描述客观世界，例如通过一个 Java 类可以描述一只猫、一个手机，而通过类的组合则可以描述几乎任意复杂的事物。这便是数据结构的意义所在。

另一个问题是：Java 数据结构的实现机制是什么？

总体而言，Java 的数据结构的实现机制是，编译时变成字节码，运行期实现。

Java 为何编译期不支持数据结构？有两方面的因素：一是因为 Java 为了实现算法的跨平台性而选择了字节码，数据结构的实现也必须跨平台，而不能直接被编译为机器指令；二是因为 Java 要实现 RTTI，即运行期类型识别。

换言之，在编译后所生成的字节码文件中，Java 类的数据结构信息其实是被抹掉的，谁也无法一眼就能从二进制格式的字节码文件中看出一个 Java 类的明确结构。但是，字节码文件通过其本身的格式化规范，确保 JVM 能够据此还原出原始的 Java 类的结构。这便是 Java 数据结构的实现机制。

本章详细分析了 JVM 如此实现数据结构的技术必然性。