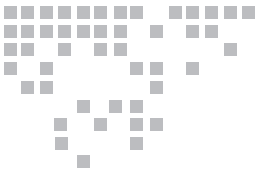

Chapter 2

第 2 章 Java 虚拟机结构

本规范描述的是一种抽象化的虚拟机的行为，而不是任何一种[㊀]广泛使用的虚拟机实现。

要去“正确地”实现一台 Java 虚拟机，其实并不像大多数人所想的那样高深和困难——只需要正确读取 class 文件中每一条字节码指令，并且能正确执行这些指令所蕴含的操作即可。所有在虚拟机规范之中没有明确描述的实现细节，都不应成为虚拟机设计者发挥创造性的牵绊，设计者可以完全自主决定所有规范中不曾描述的虚拟机内部细节，例如，运行时数据区的内存如何布局，选用哪种垃圾收集算法，是否要对虚拟机字节码指令进行一些内部优化操作（如使用即时编译器把字节码编译为机器码）。

在本规范之中所有关于 Unicode 的描述，都是基于 Unicode 6.0.0 标准，读者可以在 Unicode 的网站（http://www.unicode.org）中查找到相关资料。

2.1 class 文件格式

编译后被 Java 虚拟机所执行的代码使用了一种平台中立（不依赖于特定硬件及操作系统）的二进制格式来表示，并且经常（但并非绝对）以文件的形式存储，因此这种格式称为 class 文件格式。class 文件格式中精确地定义了类与接口的表示形式，包括在平台相关的目标文件格式中一些细节上的惯例[㊁]，例如字节序（byte ordering）等。

㊀ 包括 Oracle 公司自己的 HotSpot 和 JRockit 虚拟机。——译者注

㊁ 请勿误认为此处“平台相关的目标文件格式”是指在特定平台编译出的 class 文件无法在其他平台中使用。相反，正是因为强制、明确地定义了本来会跟平台相关的细节，所以才达到了平台无关的效果。例如在 SPARC 平台上数字以 Big-Endian（高位的字节存储在内存中的低地址处）形式存储，在 x86 平台上数字则是以 Little-Endian（高位的字节存储在内存中的高地址处）形式存储的，如果不强制统一字节序的话，同一个 class 文件的二进制形式放在不同平台上就可能以不同的方式解读。——译者注

关于 class 文件格式细节的定义，请参见第 4 章的相关内容。

2.2　数据类型

与 Java 程序语言中的数据类型相似，Java 虚拟机可以操作的数据类型可分为两类：**原始类型**（primitive type，也经常翻译为原生类型或者基本类型）和**引用类型**（reference type）。与之对应，也存在**原始值**（primitive value）和**引用值**（reference value）两种类型的数值，它们可用于变量赋值、参数传递、方法返回和运算操作。

Java 虚拟机希望尽可能多的类型检查能在程序运行之前完成，换句话说，编译器应当在编译期间尽最大努力完成可能的类型检查，使得虚拟机在运行期间无需进行这些操作。原始类型的值不需要通过特殊标记或别的额外识别手段来在运行期确定它们的实际数据类型，也无需刻意将它们与引用类型的值区分开。虚拟机的字节码指令本身就可以确定它的指令操作数的类型是什么，所以可以利用这种特性直接确定操作数的数值类型。例如，*iadd*、*ladd*、*fadd* 和 *dadd* 这几个指令的操作含义都是将两个数值相加，并返回相加的结果，但是每条指令都有自己的专属操作数类型，此处按顺序分别为：int、long、float 和 double。关于虚拟机字节码指令的介绍，读者可以参见 2.11.1 小节。

Java 虚拟机是直接支持对象的。这里的对象可以是指动态分配的某个类的实例，也可以指某个数组。虚拟机中使用 reference 类型[㊀]来表示对某个对象的引用。关于 reference 类型的值，你可以想象成指向对象的指针。每一个对象都可能存在多个指向它的引用，对象的操作、传递和检查都通过引用它的 reference 类型的数据来进行。

2.3　原始类型与值

Java 虚拟机所支持的原始数据类型包括**数值类型**（numeric type）、boolean 类型（见 2.3.4 小节）和 returnAddress 类型（见 2.3.3 小节）三类。

数值类型又分为**整数类型**（integral type，见 2.3.1 小节）和**浮点类型**（floating-point type，见 2.3.2 小节）两种。

整数类型包括：

- byte 类型：值为 8 位有符号二进制补码整数，默认值为零。
- short 类型：值为 16 位有符号二进制补码整数，默认值为零。
- int 类型：值为 32 位有符号二进制补码整数，默认值为零。
- long 类型：值为 64 位有符号二进制补码整数，默认值为零。

㊀ 这里的 reference 类型与 int、long、double 等类型是同一个层次的概念，reference 是前面提到过的引用类型（reference type）的一种，而 int、long、double 等则是前面提到的原始类型（primitive type）的一种。前者是具体的数据类型，后者是某种数据类型的统称，原书中使用不同的英文字体标识，译者根据通常使用习惯，在本书中把具体类型使用小写英文表示，而类型则统一翻译为中文形式。

- char 类型：值为使用 16 位无符号整数表示的、指向基本多文种平面（Basic Multilingual Plane，BMP）的 Unicode 码点，以 UTF-16 编码，默认值为 Unicode 的 null 码点（'\u0000'）。

浮点类型包括：

- float 类型：值为单精度浮点数集合[㊀]中的元素，或者（如果虚拟机支持的话）是单精度扩展指数（float-extended-exponent）集合中的元素，默认值为正数 0。
- double 类型：值为双精度浮点数集合中的元素，或者（如果虚拟机支持的话）是双精度扩展指数（double-extended-exponent）集合中的元素，默认值为正数 0。

boolean 类型的值为布尔值 true 和 false，默认值为 false。

在《Java 虚拟机规范（第 1 版）》中，boolean 类型并没有作为虚拟机的原始类型进行定义，当时的 Java 虚拟机只对 boolean 类型和值进行非常有限的支持，这导致 Java 虚拟机的后续发展出现了许多不必要的问题和麻烦。直到《Java 虚拟机规范（第 2 版）》时，boolean 类型才以虚拟机原始类型的形式定义。

returnAddress 类型是指向某个操作码（opcode）的指针，此操作码与 Java 虚拟机指令相对应。在虚拟机支持的所有原始类型中，只有 returnAddress 类型是不能直接与 Java 语言的数据类型相对应的。

2.3.1 整数类型与整型值

Java 虚拟机中的整数类型的取值范围如下：

- 对于 byte 类型，取值范围是 –128 ~ 127（$-2^7 \sim 2^7-1$），包括 –128 和 127。
- 对于 short 类型，取值范围是 –32 768 ~ 32 767（$-2^{15} \sim 2^{15}-1$），包括 –32 768 和 32 767。
- 对于 int 类型，取值范围是 –2 147 483 648 ~ 2 147 483 647（$-2^{31} \sim 2^{31}-1$），包括 –2 147 483 648 和 2 147 483 647。
- 对于 long 类型，取值范围是 –9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807（$-2^{63} \sim 2^{63}-1$），包括 –9 223 372 036 854 775 808 和 9 223 372 036 854 775 807。
- 对于 char 类型，取值范围是 0 ~ 65 535，包括 0 和 65 535。

2.3.2 浮点类型、取值集合及浮点值

浮点类型包含 float 类型和 double 类型两种，它们在概念上与《IEEE Standard for Binary Floating-Point Arithmetic》（ANSI/IEEE Std.754-1985，New York）标准中定义的 32 位单精度和 64 位双精度 IEEE 754 格式的取值与操作是一致的。

IEEE 754 标准的内容不仅包括了正负的带符号量（sign-magnitude number），而且包括

㊀ 单精度浮点数集合、双精度浮点数集合、单精度扩展指数集合和双精度扩展指数集合将会在稍后的 2.3.2 小节中详细介绍。——译者注

了正负零、正负**无穷大**和一个特殊的“非数字”标识（Not-a-Number，下文用 NaN 表示）。NaN 值用于表示某些无效的运算操作，例如 0 除以 0 等情况。

所有 Java 虚拟机的实现都必须支持两种标准的浮点值集合：**单精度浮点数集合**和**双精度浮点数集合**。另外，Java 虚拟机实现可以自由选择是否要支持**单精度扩展指数集合**和**双精度扩展指数集合**中的一种或全部。这些扩展指数集合可能在某些特定情况下代替标准浮点数集合来表示 `float` 和 `double` 类型的数值。

任意一个非零的、可数的任意浮点值都可以表示为 $s \times m \times 2^{(e-N+1)}$ 的形式，其中 s 可以是 +1 或者 −1，m 是一个小于 2^N 的正整数，e 是一个介于 $E_{min}=-(2^{K-1}-2)$ 和 $E_{max}=2^{K-1}-1$ 之间的整数（包括 E_{min} 和 E_{max}）。这里的 N 和 K 两个参数的取值范围决定于当前采用的浮点数值集合。部分浮点数使用这种规则得到的表示形式可能不是唯一的，例如，在指定的数值集合内，可以存在一个数字 v，它能找到特定的 s、m 和 e 值来表示，使得其中 m 是偶数，并且 e 小于 2^{K-1}，这样我们就能够通过把 m 的值减半再将 e 的值增加 1 的方式来得到 v 的另外一种不同的表示形式。在这些表示形式中，如果其中某种表示形式中 m 的值满足条件 $m \geqslant 2^{N-1}$，那就称这种表示为标准表示（mormalized representation），不满足这个条件的其他表示形式就称为非标准表示（denormalized representation）。如果某个数值不存在任何满足 $m \geqslant 2^{N-1}$ 的表示形式，即不存在任何标准表示，那就称这个数字为非标准值（denormalized value）。

对于两个必须支持的浮点数值集合和两个可选的浮点数值集合来说，参数 N 和 K（也包括衍生参数 E_{min} 和 E_{max}）的约束如表 2-1 所示。

表 2-1　浮点数集合的参数

参数	单精度浮点数集合	单精度扩展指数集合	双精度浮点数集合	双精度扩展指数集合
N	24	24	53	53
K	8	≥ 11	11	≥ 15
E_{max}	+127	≥ +1 023	+1 023	≥ +16 383
E_{min}	−126	≤ −1 022	−1 022	≤ −16 382

如果虚拟机实现支持了（无论是支持一种还是支持全部）扩展指数集合，那每一种支持的扩展指数集合都有一个由具体虚拟机实现决定的参数 K，表 2-1 给出了这个参数的约束范围（≥ 11 和 ≥ 15），这个参数也决定了 E_{min} 和 E_{max} 两个衍生参数的取值范围。

上述四种数值集合都不仅包含可数的非零值，而且包括 5 个特殊的数值：正数零、负数零、正无穷大、负无穷大和 NaN。

有一点需要注意的是，表 2-1 中的约束经过精心设计，可以保证每一个单精度浮点数集合中的元素都一定是单精度扩展指数集合、双精度浮点数集合和双精度扩展指数集合中的元素。与此类似，每一个双精度浮点数集合中的元素都一定是双精度扩展指数集合的元素。换句话说，每一种扩展指数集合都有比相应的标准浮点数集合更大的指数取值范围，但是不会有更高的精度。

每一个单精度浮点数集合中的元素都可以精确地使用 IEEE 754 标准中定义的单精度浮点格式表示出来，但 NaN 例外，取值集合中只有 1 个值用来表示 NaN。（而 IEEE 754 却规定了 $2^{24}-2$ 种不同的值，都可用来表示 NaN）。与此类似，每一个双精度浮点数集合中的元素都可以精确地使用 IEEE 754 标准中定义的双精度浮点格式表示出来，但 NaN 例外，取值集合中也只有 1 个值用来表示 NaN。（而 IEEE 754 却规定了 $2^{53}-2$ 种不同的值，都可用来表示 NaN）。不过请注意，在这里定义的单精度扩展指数集合和双精度扩展指数集合中的元素和 IEEE 754 标准里面单精度扩展与双精度扩展格式的表示**并不完全一致**。除了 `class` 文件格式中明确限定浮点值表示方式（参见 4.4.4 及 4.4.5 小节）的场合之外，本规范并不强求采用何种形式来表示浮点数。

上面提到的单精度浮点数集合、单精度扩展指数集合、双精度浮点数集合和双精度扩展指数集合都并不是具体的数据类型。虚拟机实现可以通过单精度浮点数集合的元素来表示一个 `float` 类型的数值，但是在某些特定的环境中，可以使用单精度扩展指数集合的元素来代替。相类似，虚拟机实现可以使用双精度浮点数集合的元素来表示 `double` 类型的数值，但是在某些特定的环境中，也可以使用双精度扩展指数集合的元素来代替。

除了 NaN 以外，浮点数集合中的所有元素都是**有序的**。如果把它们从小到大按顺序排列好，那顺序将会是：负无穷、可数负数、正负零、可数正数、正无穷。

在浮点数中，正数零和负数零是相等的，但是它们在某些操作上会有区别。例如，1.0 除以 0.0 会产生正无穷大的结果，而 1.0 除以 –0.0 则会产生负无穷大的结果。

NaN 是**无序的**，只要有操作数是 NaN，那么对它进行任何数值比较和等值测试都会返回 `false`。值得一提的是，有且只有 NaN 这一个数在与自身比较是否等值时会得到 `false`。任何数字与 NaN 进行不等值比较都会返回 `true`。

2.3.3 `returnAddress` 类型和值

`returnAddress` 类型会被 Java 虚拟机的 *jsr*、*ret* 和 *jsr_w* 指令[⊖]所使用参见第 6 章的 *jsr*、*ret* 和 *jsr_w* 小节。`returnAddress` 类型的值指向一条虚拟机指令的操作码。与前面介绍的那些数值类的原生类型不同，`returnAddress` 类型在 Java 语言之中并不存在相应的类型，而且也无法在程序运行期间更改。

2.3.4 `boolean` 类型

虽然 Java 虚拟机定义了 `boolean` 这种数据类型，但是只对它提供了非常有限的支持。在 Java 虚拟机中没有任何供 `boolean` 值专用的字节码指令，Java 语言表达式所操作的 `boolean` 值，在编译之后都使用 Java 虚拟机中的 `int` 数据类型来代替。

Java 虚拟机直接支持 `boolean` 类型的数组，虚拟机的 *newarray* 指令参见第 6 章的

⊖ 这几个指令以前主要用来实现 `finally` 语句块，后来改为冗余 `finally` 块代码的方式来实现，甚至到了 JDK 7 时，虚拟机已不允许 `class` 文件内出现这几个指令。那相应地，`returnAddress` 类型就处于名存实亡的状态。——译者注

newarray 小节可以创建这种数组。boolean 类型数组的访问与修改共用 byte 类型数组的 *baload* 和 *bastore* 指令。参见第 6 章的 baload 及 bastore 小节。

> 在 Oracle 公司的虚拟机实现里，Java 语言中的 boolean 数组将会被编码成 Java 虚拟机的 byte 数组，每个 boolean 元素占 8 位。

Java 虚拟机会把 boolean 数组元素中的 true 值采用 1 来表示，false 值采用 0 来表示，当 Java 编译器把 Java 语言中的 boolean 类型值映射为 Java 虚拟机的 int 类型值时，也必须采用上述表示方式。

2.4　引用类型与值

Java 虚拟机中有三种引用类型：类类型（class type）、数组类型（array type）和接口类型（interface type）。这些引用类型的值分别指向动态创建的类实例、数组实例和实现了某个接口的类实例或数组实例。

数组类型最外面那一维元素的类型（此维度的长度不由数组类型来决定），叫做该数组类型的**组件类型**（component type）。[⊖]一个数组的组件类型也可以是数组。从任意一个数组开始，如果发现其组件类型也是数组类型，那就继续取这个小数组的组件类型，不断执行这样的操作，最终一定可以遇到组件类型不是数组的情况，这时就把这种类型称为本数组类型的**元素类型**（element type）。数组的元素类型必须是原生类型、类类型或者接口类型之一。

在引用类型的值中还有一个特殊的值：null，当一个引用不指向任何对象的时候，它的值就用 null 来表示。一个为 null 的引用，起初并不具备任何实际的运行期类型，但是它可转型为任意的引用类型。引用类型的默认值就是 null。

Java 虚拟机规范并没有规定 null 在虚拟机实现中应当怎样用编码来表示。

2.5　运行时数据区

Java 虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

2.5.1　pc 寄存器

Java 虚拟机可以支持多条线程同时执行（见 JLS §17），每一条 Java 虚拟机线程都有自己的 pc（program counter）寄存器。在任意时刻，一条 Java 虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法（current method，见 2.6 节）。如果

⊖ 例如对于 int[][][] 这种数组类型来说，其组件类型可以理解为 int[][]。——译者注

这个方法不是 native 的，那 pc 寄存器就保存 Java 虚拟机正在执行的字节码指令的地址，如果该方法是 native 的，那 pc 寄存器的值是 undefined。pc 寄存器的容量至少应当能保存一个 returnAddress 类型的数据或者一个与平台相关的本地指针的值。

2.5.2 Java 虚拟机栈

每一条 Java 虚拟机线程都有自己私有的 **Java 虚拟机栈**（Java Virtual Machine stack），这个栈与线程同时创建，用于存储栈帧（Frame，见 2.6 节）。Java 虚拟机栈的作用与传统语言（例如 C 语言）中的栈非常类似，用于存储局部变量与一些尚未算好的结果。另外，它在方法调用和返回中也扮演了很重要的角色。因为除了栈帧的出栈和入栈之外，Java 虚拟机栈不会再受其他因素的影响，所以栈帧可以在堆中分配[⊖]，Java 虚拟机栈所使用的内存不需要保证是连续的。

在《Java 虚拟机规范》第 1 版中，Java 虚拟机栈也称为“Java 栈”。

Java 虚拟机规范既允许 Java 虚拟机栈被实现成固定大小，也允许根据计算动态来扩展和收缩。如果采用固定大小的 Java 虚拟机栈，那每一个线程的 Java 虚拟机栈容量可以在线程创建的时候独立选定。

Java 虚拟机实现应当提供给程序员或者最终用户调节虚拟机栈初始容量的手段，对于可以动态扩展和收缩 Java 虚拟机栈来说，则应当提供调节其最大、最小容量的手段。

Java 虚拟机栈可能发生如下异常情况：

- 如果线程请求分配的栈容量超过 Java 虚拟机栈允许的最大容量，Java 虚拟机将会抛出一个 StackOverflowError 异常。
- 如果 Java 虚拟机栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

2.5.3 Java 堆

在 Java 虚拟机中，**堆**（heap）是可供各个线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区域。

Java 堆在虚拟机启动的时候就被创建，它存储了被自动内存管理系统（automatic storage management system，也就是常说的 garbage collector（垃圾收集器））所管理的各种对象，这些受管理的对象无需也无法显式地销毁。本规范中所描述的 Java 虚拟机并未假设采用何种具体技术去实现自动内存管理系统。虚拟机实现者可以根据系统的实际需要来选择自动内存管

⊖ 请注意避免混淆 Stack、Heap 和 Java（VM）Stack、Java Heap 的概念，Java 虚拟机的实现本质上是由其他语言所编写的应用程序，Java 语言程序里分配在 Java Stack 中的数据，从实现虚拟机的程序角度上看则可能分配在 Heap 之中。——译者注

理技术。Java 堆的容量可以是固定的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。Java 堆所使用的内存不需要保证是连续的。

Java 虚拟机实现应当提供给程序员或者最终用户调节 Java 堆初始容量的手段，对于可以动态扩展和收缩 Java 堆来说，则应当提供调节其最大、最小容量的手段。

Java 堆可能发生如下异常情况：

- 如果实际所需的堆超过了自动内存管理系统能提供的最大容量，那 Java 虚拟机将会抛出一个 `OutOfMemoryError` 异常。

2.5.4 方法区

在 Java 虚拟机中，方法区（method area）是可供各个线程共享的运行时内存区域。方法区与传统语言中的编译代码存储区（storage area for compiled code）或者操作系统进程的正文段（text segment）的作用非常类似，它存储了每一个类的结构信息，例如，运行时常量池（runtime constant pool）、字段和方法数据、构造函数和普通方法的字节码内容，还包括一些在类、实例、接口初始化时用到的特殊方法（见 2.9 节）。

方法区在虚拟机启动的时候创建，虽然方法区是堆的逻辑组成部分，但是简单的虚拟机实现可以选择在这个区域不实现垃圾收集与压缩。这个版本的 Java 虚拟机规范也不限定实现方法区的内存位置和编译代码的管理策略。方法区的容量可以是固定的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。方法区在实际内存空间中可以是不连续的。

Java 虚拟机实现应当提供给程序员或者最终用户调节方法区初始容量的手段，对于可以动态扩展和收缩方法区来说，则应当提供调节其最大、最小容量的手段。

方法区可能发生如下异常情况：

- 如果方法区的内存空间不能满足内存分配请求，那么 Java 虚拟机将抛出一个 `OutOfMemoryError` 异常。

2.5.5 运行时常量池

运行时常量池（runtime constant pool）是 `class` 文件中每一个类或接口的常量池表（constant_pool table，见 4.4 节）的运行时表示形式，它包括了若干种不同的常量，从编译期可知的数值字面量到必须在运行期解析后才能获得的方法或字段引用。运行时常量池类似于传统语言中的符号表（symbol table），不过它存储数据的范围比通常意义上的符号表要更为广泛。

每一个运行时常量池都在 Java 虚拟机的方法区中分配（见 2.5.4 小节），在加载类和接口到虚拟机后，就创建对应的运行时常量池（见 5.3 节）。

在创建类和接口的运行时常量池时，可能会发生如下异常情况：

- 当创建类或接口时，如果构造运行时常量池所需要的内存空间超过了方法区所能提供的最大值，那么 Java 虚拟机将会抛出一个 `OutOfMemoryError` 异常。

关于构造运行时常量池的详细信息，可以参考第 5 章的内容。

2.5.6 本地方法栈

Java 虚拟机实现可能会使用到传统的栈（通常称为 C stack）来支持 native 方法（指使用 Java 以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（native method stack）。当 Java 虚拟机使用其他语言（例如 C 语言）来实现指令集解释器时，也可以使用本地方法栈。如果 Java 虚拟机不支持 native 方法，或是本身不依赖传统栈，那么可以不提供本地方法栈，如果支持本地方法栈，那这个栈一般会在线程创建的时候按线程分配。

Java 虚拟机规范允许本地方法栈实现成固定大小或者根据计算来动态扩展和收缩。如果采用固定大小的本地方法栈，那么每一个线程的本地方法栈容量可以在创建栈的时候独立选定。

Java 虚拟机实现应当提供给程序员或者最终用户调节本地方法栈初始容量的手段，对于长度可动态变化的本地方法栈来说，则应当提供调节其最大、最小容量的手段。

本地方法栈可能发生如下异常情况：

- 如果线程请求分配的栈容量超过本地方法栈允许的最大容量，Java 虚拟机将会抛出一个 StackOverflowError 异常。
- 如果本地方法栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的本地方法栈，那么 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

2.6 栈帧

栈帧（frame）是用来存储数据和部分过程结果的数据结构，同时也用来处理动态链接（dynamic linking）、方法返回值和异常分派（dispatch exception）。

栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。栈帧的存储空间由创建它的线程分配在 Java 虚拟机栈（见 2.5.5 小节）之中，每一个栈帧都有自己的本地变量表（local variable，见 2.6.1 小节）、操作数栈（operand stack，见 2.6.2 小节）和指向当前方法所属的类的运行时常量池（见 2.5.5 小节）的引用。

栈帧中还允许携带与 Java 虚拟机实现相关的一些附加信息，例如，对程序调试提供支持的信息。

本地变量表和操作数栈的容量在编译期确定，并通过相关方法的 code 属性（见 4.7.3 小节）保存及提供给栈帧使用。因此，栈帧数据结构的大小仅仅取决于 Java 虚拟机的实现。实现者可以在调用方法时给它们分配内存。

在某条线程执行过程中的某个时间点上，只有目前正在执行的那个方法的栈帧是活动的。这个栈帧称为**当前栈帧**（current frame），这个栈帧对应的方法称为**当前方法**（current method），定义这个方法的类称作**当前类**（current class）。对局部变量表和操作数栈的各种操作，通常都指的是对当前栈帧的局部变量表和操作数栈所进行的操作。

如果当前方法调用了其他方法，或者当前方法执行结束，那这个方法的栈帧就不再是当前栈帧了。调用新的方法时，新的栈帧也会随之而创建，并且会随着程序控制权移交到新方法而成为新的当前栈帧。方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，然后，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。

请特别注意，栈帧是线程本地私有的数据，不可能在一个栈帧之中引用另外一个线程的栈帧。

2.6.1　局部变量表

每个栈帧（见 2.6 节）内部都包含一组称为局部变量表的变量列表。栈帧中局部变量表的长度由编译期决定，并且存储于类或接口的二进制表示之中，即通过方法的 code 属性（见 4.7.3 小节）保存及提供给栈帧使用。

一个局部变量可以保存一个类型为 `boolean`、`byte`、`char`、`short`、`int`、`float`、`reference` 或 `returnAddress` 的数据。两个局部变量可以保存一个类型为 `long` 或 `double` 的数据。

局部变量使用索引来进行定位访问。首个局部变量的索引值为 0。局部变量的索引值是个整数，它大于等于 0，且小于局部变量表的长度。

`long` 和 `double` 类型的数据占用两个连续的局部变量，这两种类型的数据值采用两个局部变量中较小的索引值来定位。例如，将一个 `double` 类型的值存储在索引值为 *n* 的局部变量中，实际上的意思是索引值为 *n* 和 *n*+1 的两个局部变量都用来存储这个值。然而，索引值为 *n*+1 的局部变量是无法直接读取的，但是可能会被写入。不过，如果进行了这种操作，那将会导致局部变量 *n* 的内容失效。

前面提及的局部变量索引值 *n* 并不要求一定是偶数，Java 虚拟机也不要求 `double` 和 `long` 类型数据采用 64 位对齐的方式连续地存储在局部变量表中[㊀]。虚拟机实现者可以自由地选择适当的方式，通过两个局部变量来存储一个 `double` 或 `long` 类型的值。

Java 虚拟机使用局部变量表来完成方法调用时的参数传递。当调用类方法时，它的参数将会依次传递到局部变量表中从 *0* 开始的连续位置上。当调用实例方法时，第 *0* 个局部变量一定用来存储该实例方法所在对象的引用（即 Java 语言中的 `this` 关键字）。后续的其他参数将会传递至局部变量表中从 *1* 开始的连续位置上。

㊀ 所谓 64 位对齐（64-bit aligned），大概意思是：数据首个二进制位与局部变量表首个二进制位之间的偏移量，是 64 的整数倍。本书中类似的说法还有 4 字节对齐（4-byte aligned），意思就是：数据首个字节的位置，是 4 的整数倍。——译者注

2.6.2 操作数栈

每个栈帧（见 2.6 节）内部都包含一个称为**操作数栈**的后进先出（Last-In-First-Out，LIFO）栈。栈帧中操作数栈的最大深度由编译期决定，并且通过方法的 code 属性（见 4.7.3 小节）保存及提供给栈帧使用。

在上下文明确不会产生误解的前提下，我们经常把“当前栈帧的操作数栈”直接简称为“操作数栈”。

栈帧在刚刚创建时，操作数栈是空的。Java 虚拟机提供一些字节码指令来从局部变量表或者对象实例的字段中复制常量或变量值到操作数栈中，也提供了一些指令用于从操作数栈取走数据、操作数据以及把操作结果重新入栈。在调用方法时，操作数栈也用来准备调用方法的参数以及接收方法返回结果。

例如，*iadd* 字节码指令（参见第 6 章的 iadd 小节）的作用是将两个 `int` 类型的数值相加，它要求在执行之前操作数栈的栈顶已经存在两个由前面的其他指令所放入的 `int` 类型数值。在执行 *iadd* 指令时，两个 `int` 类型数值从操作栈中出栈，相加求和，然后将求和结果重新入栈。在操作数栈中，一项运算常由多个子运算（subcomputation）嵌套进行，一个子运算过程的结果可以被其他外围运算所使用。

操作数栈的每个位置上可以保存一个 Java 虚拟机中定义的任意数据类型的值，包括 `long` 和 `double` 类型。

在操作数栈中的数据必须正确地操作。例如，不可以入栈两个 `int` 类型的数据，然后当做 `long` 类型去操作，或者入栈两个 `float` 类型的数据，然后使用 *iadd* 指令对它们求和。有一小部分 Java 虚拟机指令（例如 *dup* 和 *swap* 指令，分别参见第 6 章的 dup 和 swap 小节）可以不关注操作数的具体数据类型，把所有在运行时数据区中的数据当做裸类型（raw type）数据来操作，这些指令不可以用来修改数据，也不可以拆散那些原本不可拆分的数据，这些操作的正确性将会通过 `class` 文件的校验过程（见 4.10 节）来强制保障。

在任意时刻，操作数栈都会有一个确定的栈深度，一个 `long` 或者 `double` 类型的数据会占用两个单位的栈深度，其他数据类型则会占用一个单位的栈深度。

2.6.3 动态链接

每个栈帧（见 2.6 节）内部都包含一个指向当前方法所在类型的运行时常量池（见 2.5.5 小节）的引用，以便对当前方法的代码实现**动态链接**。在 `class` 文件里面，一个方法若要调用其他方法，或者访问成员变量，则需要通过符号引用（symbolic reference）来表示，动态链接的作用就是将这些以符号引用所表示的方法转换为对实际方法的直接引用。类加载的过程中将要解析尚未被解析的符号引用，并且将对变量的访问转化为变量在程度运行时，位于存储结构中的正确偏移量。

由于对其他类中的方法和变量进行了晚期绑定（late binding），所以即便那些类发生变化，也不会影响调用它们的方法。

2.6.4 方法调用正常完成

方法调用正常完成是指在方法的执行过程中，没有抛出任何异常（见 2.10 节）——包括直接从 Java 虚拟机中抛出的异常以及在执行时通过 throw 语句显式抛出的异常。如果当前方法调用正常完成，它很可能会返回一个值给调用它的方法。方法正常完成发生在一个方法执行过程中遇到了方法返回的字节码指令（见 2.11.8 小节）时，使用哪种返回指令取决于方法返回值的数据类型（如果有返回值）。

在这种场景下，当前栈帧（见 2.6 节）承担着恢复调用者状态的责任，包括恢复调用者的局部变量表和操作数栈，以及正确递增程序计数器，以跳过刚才执行的方法调用指令等。调用者的代码在被调用方法的返回值压入调用者栈帧的操作数栈后，会继续正常执行。

2.6.5 方法调用异常完成

方法调用异常完成是指在方法的执行过程中，某些指令导致了 Java 虚拟机抛出异常（见 2.10 节），并且虚拟机抛出的异常在该方法中没有办法处理，或者在执行过程中遇到 athrow 字节码指令（参见第 6 章的 athrow 小节）并显式地抛出异常，同时在该方法内部没有捕获异常。如果方法异常调用完成，那一定不会有方法返回值返回给其调用者。

2.7 对象的表示

Java 虚拟机规范不强制规定对象的内部结构应当如何表示。

在 Oracle 的某些 Java 虚拟机实现中，指向对象实例的引用是一个指向**句柄**的指针，这个句柄又包含了两个指针，其中一个指针指向一张表格，此表格包含该对象的各个方法，还包含指向 Class 对象的指针，那个 Class 对象用来表示该对象的类型。句柄的另外一个指针指向分配在堆中的对象实例数据。㊀

2.8 浮点算法

Java 虚拟机采纳了《IEEE Standard for Binary Floating-Point Arithmetic》（ANSI/IEEE Std.754-1985，New York）浮点算法规范中的一个子集。

2.8.1 Java 虚拟机和 IEEE 754 中的浮点算法

Java 虚拟机中支持的浮点算法和 IEEE 754 标准中的主要差别有：

㊀ 这条注释在 10 多年前出版的《Java 虚拟机规范（第 2 版）》中就已经存在，第 3 版中仅仅是将 Sun 修改为 Oracle 而已，所表达的实际信息已比较陈旧。在 HotSpot 虚拟机中，指向对象的引用并不通过句柄，而是直接指向堆中对象的实例数据，因此 HotSpot 虚拟机并不包括在上面所描述的“Oracle 的某些 Java 虚拟机实现”范围之内。——译者注

- Java 虚拟机中的浮点操作在遇到非法操作，如被零除（division by zero）、上限溢出（overflow）、下限溢出（underflow）和非精确（inexact）时，不会抛出 exception、trap 或者 IEEE 754 异常情况中定义的其他信号。Java 虚拟机也没有信号 NaN 值（signaling NaN value）。
- Java 虚拟机不支持 IEEE 754 中的信号浮点比较（signaling floating-point comparison）。
- 在 Java 虚拟机中，舍入操作永远使用 IEEE 754 标准中定义的向最接近数舍入模式（round to nearest mode），无法精确表示的结果将会舍入为最接近的可表示值，如果最接近的值有两个，那就舍入到最低有效位为 0（a zero least-significant bit）的那个值。这种模式也是 IEEE 754 中的默认模式。不过在 Java 虚拟机里面，将浮点数值转化为整型数值使用向零舍入（round toward zero）。Java 虚拟机并不给出改变浮点运算舍入模式的手段[㊀]。
- Java 虚拟机不支持 IEEE 754 的单精度扩展和双精度扩展格式，但是在双精度浮点数集合和双精度扩展指数集合（见 2.3.2 小节）的范围与单精度扩展格式的表示会有重叠。虚拟机实现可以选择是否支持单精度扩展指数和双精度扩展指数集合，但它们并不等同于 IEEE 754 中的单精度和双精度扩展格式：IEEE 754 中的扩展格式不仅扩展了指数的范围，而且还扩展了精度。

2.8.2 浮点模式

每个方法都有一项属性称为**浮点模式**（floating-point mode），取值有两种，要么是 **FP-strict 模式**要么是**非 FP-strict 模式**。方法的浮点模式决定于 `class` 文件中代表该方法的 `method_info` 结构（见 4.6 节）的访问标志（`access_flags`）中的 `ACC_STRICT` 标志位。如果此标志位为真，则该方法的浮点模式就是 FP-strict，否则就是非 FP-strict 模式。

> 上述 `ACC_STRICT` 标志位与浮点数模式之间的对应关系意味着：如果方法所在的类是用 JDK 1.1 或早前版本的编译器来编译的，那么该方法的浮点数模式实际上就是非 FP-strict 模式。

我们说一个操作数栈具有某种给定浮点模式，所指的就是包含操作数栈的栈帧所对应的方法具备的浮点模式，相类似，我们说一条 Java 虚拟机字节码指令具备某种浮点模式，所指的也是包含这条指令的方法具备的浮点模式。

如果虚拟机实现支持单精度指数扩展集合（见 2.3.2 小节），那么在非 FP-strict 模式的操作数栈上，除非数值集合转换（见 2.8.3 小节）明确禁止，否则 `float` 类型的值可能会超过单精度浮点数集合的取值范围。同样，如果虚拟机实现支持双精度指数扩展集合（见 2.3.2 小

㊀ IEEE 754 中定义了 4 种舍入模式，除了上面提到的向最接近数舍入和向零舍入以外，还有向正无穷舍入和向负无穷舍入两种模式。向最接近数舍入模式即我们平常所说的“四舍五入”法，而向零舍入即平常所说的“去尾”法。——译者注

节），那么在非 FP-strict 模式的操作数栈上，除非数值集合转换（见 2.8.3 小节）明确禁止，否则 double 类型的值可能会超过双精度浮点数集合的取值范围。

在其他的上下文中，无论操作数栈或者别的地方都不再特别关注浮点模式，float 和 double 两种浮点类型数值都分别限于单精度与双精度浮点数集合之中。尤其是，类和实例的字段、数组元素、本地变量和方法参数的取值范围都限于标准的数值集合之中。

2.8.3　数值集合转换

在一些特定场景下，支持扩展指数集合的 Java 虚拟机实现数值在标准浮点数集合与扩展指数集合之间的映射关系是允许或必要的，这种映射操作就称为**数值集合转换**。数值集合转换并非数据类型转换，而是在同一种数据类型的不同数值集合之间进行映射。

在数值集合转换发生的位置，虚拟机实现允许对数值执行下面操作之一。

- 如果一个数值是 float 类型，并且不是单精度浮点数集合中的元素，允许将其映射到单精度浮点数集合中数值最接近的元素。
- 如果一个数值是 double 类型，并且不是双精度浮点数集合中的元素，允许将其映射到双精度浮点数集合中数值最接近的元素。

此外，在数值集合转换发生的位置，下面的操作是必需的。

- 假设正在执行的 Java 虚拟机字节码指令是非 FP-strict 模式的，但这个指令导致一个 float 类型的值压入一个 FP-strict 模式的操作数栈中，或作为方法参数进行传递，或者存储进局部变量、字段或者数组元素之中。如果这个数值不是单精度浮点数集合中的元素，则必须将其映射到单精度浮点数集合中数值最接近的元素。
- 假设正在执行的 Java 虚拟机字节码指令是非 FP-strict 模式的，但这个指令导致一个 double 类型的值压入一个 FP-strict 模式的操作数栈中，或作为方法参数进行传递，或者存储进局部变量、字段或者数组元素之中。如果这个数值不是双精度浮点数集合中的元素，则必须将其映射到双精度浮点数集合中数值最接近的元素。

在方法调用中传递参数（包括 native 方法的调用），在非 FP-strict 模式的方法里返回浮点类型的结果到 FP-strict 模式的方法，或者在非 FP-strict 模式的方法中存储浮点类型数值到局部变量、字段或者数组元素之中时，都必须执行上述数值集合转换。

并非所有扩展指数集合中的数值都可以精确映射到标准浮点数值集合中的元素。如果进行映射的数值过大（扩展指数集合的指数可能比标准数值集合的允许最大值要大），无法在标准数值集合之中精确表示的话，这个数字将会被转化成对应类型的（正或负）无穷大。如果进行映射的数值过小（扩展指数集合的指数可能比标准数值集合的允许最小值要小），无法在标准数值集合之中精确表示，这个数字将会被转化成最接近的可以表示的非标准值（见 2.3.2 小节）或者相同正负符号的零。

数值集合转换不改变正负无穷和 NaN，而且也不能改变待转换数值的符号，对于一个非浮点类型的数值，数值集合转换是无效的。

2.9 特殊方法

在 Java 虚拟机层面上，Java 编程语言中的构造器（JLS §8.8）是以一个名为 <init> 的特殊**实例初始化方法**的形式出现的。<init> 这个方法名称是由编译器命名的，因为它并非一个合法的 Java 方法名字，不可能通过程序编码的方式实现。实例初始化方法只能在实例的初始化期间，通过 Java 虚拟机的 *invokespecial* 指令来调用，而且只能在尚未初始化的实例上调用该指令。构造器的访问权限（参见 JLS §6.6），也会约束由该构造器所衍生出来的实例初始化方法。

一个类或者接口最多可以包含不超过一个**类或接口的初始化方法**，类或者接口就是通过这个方法完成初始化的（见 5.5 节）。这个方法是一个不包含参数的、返回类型为 void 的方法，名为 <clinit>（见 4.3.3 小节）。

> 在 class 文件中把其他方法命名为 <clinit> 是没有意义的，这些方法并不是类或接口的初始化方法，它们既不能被字节码指令调用，也不会被虚拟机自己调用。

当 class 文件的版本号不小于 51.0 时，<clinit> 方法要想成为类或接口的初始化方法，必须设置 ACC_STATIC 标志。

> 这个规定是在 Java SE 7 中新增的。在 class 文件版本号不大于 50.0 时，<clinit> 方法只要求保证不包含参数，并且返回类型为 void 即可，不强制要求检查是否设置了 ACC_STATIC 标志。

<clinit> 这个名字也是由编译器命名的，因为它并非一个合法的 Java 方法名字，不可能通过 Java 程序编码的方式直接实现。类或接口的初始化方法由 Java 虚拟机自身隐式调用，没有任何虚拟机字节码指令可以调用这个方法，它只会在类的初始化阶段中由虚拟机自身调用。

当一个方法具有**签名多态性**（signature polymorphic），则意味着这个方法满足以下全部条件：

- 通过 java.lang.invoke.MethodHandle 类进行声明。
- 只有一个类型为 Object[] 的形参。
- 返回值为 Object。
- ACC_VARARGS 和 ACC_NATIVE 标志被设置。

在 Java SE 8 中，只有 java.lang.invoke.MethodHandle 的 invoke 和 invokeExact 是签名多态性方法。

在 Java SE 8 中，*invokevirtual* 指令（参见第 6 章的 invokevirtual 小节）将对具有签名多态性的方法进行特殊处理，以保证方法句柄能够正常调用。方法句柄是一种可以直接运行的强类型引用，它可以指向相关的方法、构造器、字段或者其他低级操作（见 5.4.3.5 小节），并具有参数或返回值转换能力。这里所说的转换能力（transformation）是相当广

泛的，它可以对原方法执行转化（conversion）、插入（insertion）、删除（deletion）及替换（substitution）等形式的变换，具体可参见 Java SE 平台 API 文档中 `java.lang.invoke` 包的相关信息。

2.10　异常

Java 虚拟机里面的异常使用 `Throwable` 或其子类的实例来表示，抛异常的本质实际上是程序控制权的一种即时的、非局部（nonlocal）的转换——从异常抛出的地方转换至处理异常的地方。

绝大多数异常的产生都是由于当前线程执行的某个操作所导致的，这种可以称为同步异常。与之相对，异步异常可以在程序执行过程中随时发生。Java 虚拟机中异常的出现总是由下面三种原因之一导致的。

- *athrow* 字节码指令被执行。
- 虚拟机同步检测到程序发生了非正常的执行情况，这时异常必将紧接着在发生非正常执行情况的字节码指令之后抛出，而不会在执行程序的过程中随时抛出。例如：
  - 程序所执行的操作可能会引发异常，例如：
    - 当字节码指令所蕴含的操作违反了 Java 语言的语义，如访问一个超出数组边界范围的元素。
    - 当程序在加载或者连接时出现错误。
  - 使用某些资源的时候产生资源限制，例如使用了太多的内存。
- 由于以下原因，导致了异步异常的出现：
  - 调用了 `Thread` 或者 `ThreadGroup` 的 `stop` 方法。
  - Java 虚拟机实现发生了内部错误。

  当某个线程调用了 `stop` 方法时，将会影响到其他的线程，或者在特定线程组中的所有线程。这时候其他线程中出现的异常就是异步异常，因为这些异常可能出现在线程执行过程的任何位置。虚拟机的内部错误也被认为是一种异步异常（见 6.3 节）。

Java 虚拟机规范允许在异步异常抛出之前额外执行一小段有限的代码，使得代码优化器能够在不违反 Java 语言语义的前提下检测并把这些异常在可处理它们的地方抛出。

> 简单的 Java 虚拟机实现，可以在程序执行控制权转移指令时，处理异步异常。因为程序终归是有限的，总会遇到控制权转移的指令，所以异步异常抛出的延迟时间也是有限的。如果能保证在控制权转移指令之间的代码没有异步异常抛出，那么代码生成器就可以相当灵活地进行指令重排序优化来获取更好的性能。相关的资料推荐进一步阅读论文：“Polling Efficiently on Stock Hardware”，Marc Feeley，Proc.1993，《Conference on Functional Programming and Computer Architecture》，Copenhagen，Denmark，第 179 ~ 187 页。

抛出异常的动作在 Java 虚拟机之中是有精确的定义，当异常抛出、程序控制权发生转

移的那一刻，所有在异常抛出的位置之前的字节码指令所产生的影响[㊀]都应当是可以观察到的， 而在异常抛出的位置之后的字节码指令，则不应当产生执行效果。如果虚拟机执行的代码是优化后的代码[㊁]，有一些在异常出现位置之后的代码可能已经执行了，那这些优化过的代码必须保证被它们提前执行所产生的影响对用户程序来说都是不可见的。

由 Java 虚拟机执行的每个方法都会配有零至多个**异常处理器**（exception handler）。异常处理器描述了其在方法代码中的有效作用范围（通过字节码偏移量范围来描述）、能处理的异常类型以及处理异常的代码所在的位置。要判断某个异常处理器是否可以处理某个具体的异常，需要同时检查异常出现的位置是否在异常处理的有效作用范围内，以及出现的异常是否是异常处理器声明可以处理的异常类型或其子类型。当抛出异常时，Java 虚拟机搜索当前方法包含的各个异常处理器，如果能找到可以处理该异常的异常处理器，则将代码控制权转向异常处理器中描述的处理异常的分支之中。

如果当前方法中没有找到任何异常处理器，并且当前方法调用期间确实发生了异常（见 2.6.5 小节），也即方法异常完成的情况，那当前方法的操作数栈和局部变量表都将被丢弃，随后它对应的栈帧出栈，并恢复到该方法调用者的栈帧中。未被处理的异常将在方法调用者的栈帧中重新被抛出，并在整个方法调用链里不断重复进行前面描述的处理过程。如果已经到达方法调用链的顶端，却还没有找到合适的异常处理器去处理这个异常，那整个执行线程都将被终止。

搜索异常处理器时的搜索顺序是很关键的，在 `class` 文件里面，每个方法的异常处理器都存储在一个表中（见 4.7.3 小节）。在运行时，当有异常抛出之后，Java 虚拟机就按照 class 文件中的异常处理器表所描述的异常处理器的先后顺序，从前至后进行搜索。

需要注意，Java 虚拟机本身不会对方法的异常处理器表进行排序或者其他方式的强制处理，所以 Java 语言中对异常处理的语义，实际上是通过编译器适当安排异常处理器在表中的顺序来协助完成的（参见 3.12 节）。只有在 `class` 文件中定义了明确的异常处理器查找顺序，才能保证无论 class 文件是通过何种途径产生的，Java 虚拟机执行时都能有一致的行为表现。

2.11 字节码指令集简介

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。

如果忽略异常处理，那么 Java 虚拟机的解释器通过下面这个伪代码的循环即可有效工作：

㊀ 这里的“影响”包括了异常出现之前的字节码指令执行后对局部变量表、操作数栈、其他运行时数据区域以及虚拟机外部资源产生的影响。——译者注

㊁ 这里的“优化后的代码”主要是指进行了指令重排序优化的代码。——译者注

```
do {
    自动计算 pc 寄存器以及从 pc 寄存器的位置取出操作码；
    if （存在操作数） 取出操作数；
    执行操作码所定义的操作；
} while （处理下一次循环）;
```

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 *big-endian* 顺序存储，即高位在前的字节序。例如，如果要将一个 16 位长度的无符号整数使用两个无符号字节存储起来（将它们命名为 *byte1* 和 *byte2*），那这个 16 位无符号整数的值就是：(*byte1* << 8) | *byte2*。

字节码指令流应当都是单字节对齐的，只有 *tableswitch* 和 *lookupswitch* 两个指令例外（参见第 6 章 tableswitch、lookupswitch 小节），由于它们的操作数比较特殊，都是以 4 字节为界划分的，所以当这两个指令的参数位置不是 4 字节的倍数时，需要预留出相应的空位补全到 4 字节的倍数以实现对齐。

> 限制 Java 虚拟机操作码的长度为一个字节，并且放弃了编译后代码的参数长度对齐，是为了尽可能地获得短小精悍的编译代码，但这样做可能会使某些简单的虚拟机实现损失一些性能。由于每个操作码只能有一个字节长度，所以直接限制了整个指令集的最大数量[㊀]，又由于没有假设数据是经过对齐的，所以意味着虚拟机处理那些超过一个字节的数据时，不得不在运行时从字节流中重建出具体数据的结构，这在某种程度上会损失一些性能。

2.11.1　数据类型与 Java 虚拟机

在 Java 虚拟机的指令集中，大多数的指令都包含了其所操作的数据类型信息。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。这两个指令的操作可能会是由同一段代码来实现的，但它们必须拥有各自独立的操作码。

对于大部分与数据类型相关的字节码指令来说，它们的操作码助记符中都有特殊的字符来表明该指令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`，*a* 代表 `reference`。也有一些指令的助记符没有明确用字母指明数据类型，例如 *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。还有另外一些指令，例如，无条件跳转指令 *goto* 则是与数据类型无关的。

因为 Java 虚拟机的操作码长度只有一个字节，所以包含了数据类型的操作码给指令集的设计带来了很大的压力。如果每一种与数据类型相关的指令都支持 Java 虚拟机的所有运行时数据类型，那恐怕就会超出一个字节所能表示的数量范围了。因此，Java 虚拟机的指令集对于特定的操作只提供了有限的类型相关指令，换句话说，指令集将会故意设计成非完全独立

㊀ 字节码无法超过 256 种的限制就来源于此。——译者注

的（not orthogonal，即并非每种数据类型和每一种操作都有对应的指令）。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的类型。

表 2-2 列举了 Java 虚拟机所支持的字节码指令集。用数据类型列所代表的特殊字符替换 opcode 列的指令模板中的 *T*，就可以得到一个具体的字节码指令。如果在表中指令模板与数据类型两列共同确定的单元格为空，则说明虚拟机不支持对这种数据类型执行这项操作。例如，load 指令有操作 `int` 类型的 *iload*，但是没有操作 `byte` 类型的同类指令。

请注意，从表 2-2 中可以看出，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` 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 `int` 类型的字节码指令来处理。因此，操作数的实际类型为 `boolean`、`byte`、`char` 及 `short` 的大多数操作，都可以用操作数的运算类型（computational type）为 `int` 的指令来完成。。

表 2-2　Java 虚拟机指令集所支持的数据类型

操作码	byte	short	int	long	float	double	char	refernce
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			

（续）

操作码	byte	short	int	long	float	double	char	refernce
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

在 Java 虚拟机中，实际类型与运算类型之间的映射关系如表 2-3 所示。

表 2-3　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

某些对操作数栈进行操作的 Java 虚拟机指令（例如 *pop* 和 *swap* 指令）是与具体类型无关的，不过，这些指令必须遵守运算类型分类的限制，这些分类也在表 2-3 中列出了。

2.11.2　加载和存储指令

加载和存储指令用于将数据从栈帧（见 2.6 节）的本地变量表（见 2.6.1 小节）和操作数栈之间来回传递（见 2.6.2 小节）：

- 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>*。
- 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>*。
- 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>*。
- 用于扩充局部变量表的访问索引或立即数的指令：wide。

访问对象的字段或数组元素（见 2.11.5 小节）的指令同样也会与操作数栈传递数据。

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` 和 `short` 类型数据时，经常用 `int` 类型的指令来表示（见 2.11.1 小节）。

这种指令表示方法在整个 Java 虚拟机规范之中都是通用的。

2.11.3 算术指令

算术指令用于对两个操作数栈上的值进行某种特定运算，并把结构重新压入操作数栈。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类型数据进行运算的指令。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（见 2.11.1 小节）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。整型与浮点类型的算术指令在溢出和被零除的时候也有各自不同的行为。所有的算术指令包括：

- 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*
- 求负值指令：*ineg*、*lneg*、*fneg*、*dneg*
- 移位指令：*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*
- 局部变量自增指令：*iinc*
- 比较指令：*dcmpg*、*dcmpl*、*fcmpg*、*fcmpl*、*lcmp*

Java 虚拟机的指令集直接支持了在 Java 语言规范中描述的各种对整型及浮点类型数进行操作（JSL §4.2.2，JSL §4.2.4）的语义。

Java 虚拟机没有明确规定整型数据溢出的情况，只有整数除法指令（*idiv* 和 *ldiv*）及整数求余指令（*irem* 和 *lrem*）在除数为零时会导致虚拟机抛出异常。如果发生了这种情况，虚拟机将会抛出 `ArithmeticException` 异常。

Java 虚拟机在处理浮点数时，必须遵循 IEEE 754 标准中所规定的行为限制。也就是说，Java 虚拟机要求完全支持 IEEE 754 中定义的**非标准**浮点数值（见 2.3.2 小节）和**逐级下溢**（gradual underflow）。这使得开发者更容易判断出某些数值算法是否满足预期的特征。

Java 虚拟机要求在进行浮点数运算时，所有的运算结果都必须舍入到适当的精度，**非精**

确的结果必须舍入为可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，那将优先选择最低有效位为 0 的。这种舍入模式也是 IEEE 754 标准中的默认舍入模式，称为**向最接近数舍入模式**（见 2.8.1 小节）。

在把浮点类型数转换为整型数时，Java 虚拟机使用 IEEE 754 标准中的**向零舍入模式**（见 2.8.1 小节），这种模式的舍入结果会导致数字被截断，所有表示小数部分的有效位都会被丢弃。向零舍入模式将在目标数值类型中选择一个值最接近，但是在绝对值上不大于原值的数字来作为舍入结果。

Java 虚拟机在处理浮点类型数运算时，不会抛出任何运行时异常（这里所讲的是 Java 的异常，请勿与 IEEE 754 标准中的浮点异常互相混淆），当一个操作向上溢出时，将会使用有符号的无穷大来表示，当一个操作向下溢出时，会产生非标准值，或带符号的 0 值。如果某个操作结果没有明确的数学定义，将会使用 NaN 值来表示。所有使用 NaN 值作为操作数的算术操作，结果都会返回 NaN。

在对 `long` 类型数进行比较时，虚拟机采用带符号的比较方式，而对浮点类型数进行比较时（*dcmpg*、*dcmpl*、*fcmpg*、*fcmpl*），虚拟机采用 IEEE 754 标准所定义的无信号比较（nonsignaling comparison）方式。

2.11.4　类型转换指令

类型转换指令可以在两种 Java 虚拟机数值类型之间相互转换。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，或者用来解决 Java 虚拟机字节码指令的不完备问题（见 2.11.1 小节）。

Java 虚拟机直接支持[㊀]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）：

- 从 `int` 类型到 `long`、`float` 或者 `double` 类型
- 从 `long` 类型到 `float`、`double` 类型
- 从 `float` 类型到 `double` 类型

宽化类型转换指令包括：*i2l*、*i2f*、*i2d*、*l2f*、*l2d* 和 *f2d*。从这些操作码的助记符中可以很容易知道转换的源和目标类型的名字，两个类型名中间的“2”（two）表示“to”的意思。例如，*i2d* 指令就代表从 `int` 转换到 `double`。

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。在 FP-strict（见 2.8.2 小节）模式下，从 `float` 转换到 `double` 也是可以保证转换前后精确相等，但是在非 FP-strict 模式下，则不能保证这一点。

从 `int` 或者 `long` 类型数值转换到 `float`，或者 `long` 类型数值转换到 `double` 时，将可能发生精度丢失——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 IEEE

㊀ “直接支持”意味着只需一条转换指令。——译者注

754 最接近舍入模式所得到的正确整数值。

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（注意，这里的异常不要与 IEEE 754 中的浮点异常信号混淆了）。

从 int 到 long 的宽化类型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。从 char 到一个整数类型的宽化类型转换是零位扩展，即直接给 char 的二进制形式添上若干个 0，以填充成更宽的格式。

需要注意，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，其中原因在 2.11.1 小节提到过：byte、char 和 short 类型值在虚拟机内部本来就是按更宽的 int 类型来存储的，所以这些类型的转换自然就完成了。

Java 虚拟机也直接支持以下窄化类型转换：

- 从 int 类型到 byte、short 或者 char 类型
- 从 long 类型到 int 类型
- 从 float 类型到 int 或者 long 类型
- 从 double 类型到 int、long 或者 float 类型

窄化类型转换（narrowing numeric conversion）指令包括：*i2b*、*i2c*、*i2s*、*l2i*、*f2i*、*f2l*、*d2i*、*d2l* 和 *d2f*。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。

在将 int 或 long 类型窄化转换为整数类型 T 时，转换过程仅仅是简单丢弃除最低 N 个二进制位以外的内容，其中 N 是表示类型 T 所需的二进制位个数。这将可能导致转换结果与输入值有不同的正负号[⊖]。

在将一个浮点类型数值窄化转换为整数类型 T（其中 T 限于 int 或 long 类型）时，将遵循以下转换规则：

- 如果浮点类型数值是 NaN，那转换结果就是 int 或 long 类型的 0。
- 否则，如果浮点类型数值不是无穷，那么浮点类型数值就依照 IEEE 754 标准的向零舍入模式（见 2.8.1 小节）取整，获得整型数值 V，这时可能有两种情况：
  - 如果 T 是 long 类型，并且转换结果在 long 类型的表示范围之内，那就转换为 long 类型数值 V。
  - 如果 T 是 int 类型，并且转换结果在 int 类型的表示范围之内，那就转换为 int 类型数值 V。
- 否则：
  - 如果转换结果 V 的值太小（包括绝对值很大的负数以及负无穷大的情况），无法使用 T 类型表示，那转换结果取 int 或 long 类型所能表示的最小数值。
  - 如果转换结果 V 的值太大（包括很大的正数以及正无穷大的情况），无法使用 T 类型表示，那转换结果取 int 或 long 类型所能表示的最大数值。

⊖ 在高位字节的符号位被丢弃了。——译者注

从 double 类型到 float 类型做窄化转换的过程与 IEEE 754 中定义的一致，通过 IEEE 754 向最接近数舍入模式（见 2.8.1 小节）舍入得到一个可以使用 float 类型表示的数值。如果转换结果的绝对值太小无法使用 float 来表示，将返回 float 类型的正负 0。如果转换结果的绝对值太大无法使用 float 来表示，将返回 float 类型的正负无穷大，double 类型的 NaN 值将转换为 float 类型的 NaN 值。

尽管可能发生上限溢出、下限溢出和精度丢失等情况，但是 Java 虚拟机中数值类型的窄化转换永远不可能导致虚拟机抛出运行时异常（此处的异常是指 Java 虚拟机规范中定义的异常，请不要与 IEEE 754 中定义的浮点异常信号混淆）。

2.11.5　对象的创建与操作

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：

- 创建类实例的指令：*new*。
- 创建数组的指令：*newarray*、*anewarray*、*multianewarray*。
- 访问类字段（static 字段，或者称为类变量）和类实例字段（非 static 字段，或者称为实例变量）的指令：*getfield*、*putfield*、*getstatic*、*putstatic*。
- 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*。
- 取数组长度的指令：*arraylength*。
- 检查类实例或数组类型的指令：*instanceof*、*checkcast*。

2.11.6　操作数栈管理指令

Java 虚拟机提供了一些用于直接控制操作数栈的指令，包括：*pop*、*pop2*、*dup*、*dup2*、*dup_x1*、*dup2_x1*、*dup_x2*、*dup2_x2* 和 *swap*。

2.11.7　控制转移指令

控制转移指令可以让 Java 虚拟机有条件或无条件地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。控制转移指令包括：

- 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* 和 *if_acmpne*。
- 复合条件分支：*tableswitch*、*lookupswitch*。
- 无条件分支：*goto*、*goto_w*、*jsr*、*jsr_w*、*ret*。

Java 虚拟机中有专门的条件分支指令集用来处理 int 和 reference 类型的比较操作，而且也有专门的指令用来检测 null 值，所以无需用某个具体的值来表示 null（见 2.4 节）。

boolean、byte、char 和 short 类型的条件分支比较操作，都使用 int 类型的比较指令来完成，而对于 long、float 和 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（见 2.11.3 小节），运算指令会返回一个整型数值到操作数栈中，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。由于各种类型的比较最终都会转化为 int 类型的比较操作，所以基于 int 类型比较的重要性，Java 虚拟机提供了非常丰富的 int 类型的条件分支指令。

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。

2.11.8 方法调用和返回指令

以下 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）。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（见 2.9 节）、私有方法和父类方法。
- *invokestatic* 指令用于调用命名类中的类方法（static 方法）。
- *invokedynamic* 指令用于调用以绑定了 *invokedynamic* 指令的调用点对象（call site object）作为目标的方法。调用点对象是一个特殊的语法结构，当一条 *invokedynamic* 指令首次被 Java 虚拟机执行前，Java 虚拟机将会执行一个引导方法（bootstrap method）并以这个方法的运行结果作为调用点对象。因此，每条 *invokedynamic* 指令都有独一无二的链接状态，这是它与其他方法调用指令的一个差异。

方法返回指令根据返回值的类型进行区分，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、类和接口的类初始化方法使用。

2.11.9 抛出异常

在程序中显式抛出异常的操作由 *athrow* 指令实现，除了这种情况，还有别的异常会在其他 Java 虚拟机指令检测到异常状况时由虚拟机自动抛出。

2.11.10 同步

Java 虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构都是使用同步锁（monitor）来支持的。

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（见 2.11.8 小节）之中。虚拟机可以从方法常量池中的方法表结构（method_info structure，见

4.6 节）中的 ACC_SYNCHRONIZED 访问标志区分一个方法是否是同步方法。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，其他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。

指令序列的同步通常用来表示 Java 语言中的 synchronized 块，Java 虚拟机的指令集中有 *monitorenter* 和 *monitorexit* 两个指令来支持这种 synchronized 关键字的语义。正确实现 synchronized 关键字需要编译器与 Java 虚拟机两者协作支持（见 3.14 节）。

结构化锁定（structured locking）是指在方法调用期间每一个同步锁退出都与前面的同步锁进入相匹配的情形。因为无法保证所有提交给 Java 虚拟机执行的代码都满足结构化锁定，所以 Java 虚拟机允许（但不强制要求）通过以下两条规则来保证结构化锁定成立。假设 T 代表一个线程，M 代表一个同步锁，那么：

1. T 在方法执行时持有同步锁 M 的次数必须与 T 在方法执行（包括正常和非正常完成）时释放同步锁 M 的次数相等。
2. 在方法调用过程中，任何时刻都不会出现线程 T 释放同步锁 M 的次数比 T 持有同步锁 M 次数多的情况。

请注意，在调用同步方法时也认为自动持有和释放同步锁的过程是在方法调用期间发生。

2.12 类库

Java 虚拟机必须对 Java SE 平台下的类库实现提供充分的支持，因为其中有一些类库如果没有 Java 虚拟机的支持是根本无法实现的。

可能需要 Java 虚拟机特殊支持的类包括：

- 反射，例如在 java.lang.reflect 包中的各个类和 Class 类。
- 加载和创建类或接口的类，最显而易见的例子就是 ClassLoader 类。
- 连接和初始化类或接口的类，刚才说的 ClassLoader 也属于这样的类。
- 安全，例如在 java.security 包中的各个类和 SecurityManager 等其他类。
- 多线程，譬如 Thread 类。
- 弱引用，譬如在 java.lang.ref 包中的各个类。

上面列举的几点旨在简单说明而不是详细介绍这些类库，详细列举这些类及其功能已经超出了本书的范围。如果读者想了解这些类库，请阅读 Java 平台的类库说明书。

2.13 公有设计、私有实现

到目前为止，本书简单描绘了 Java 虚拟机应有的共同外观：class 文件格式以及字节码指令集等。这些内容与 Java 虚拟机的硬件独立性、操作系统独立性以及实现独立性都是密切相关的。虚拟机实现者可能更愿意把它们看做程序在各种 Java 平台实现之间安全交互的手段，而不是一张需要精确遵从的计划蓝图。

理解公有设计与私有实现之间的分界线是非常有必要的[㊀]，Java 虚拟机实现必须能够读取 class 文件并精确实现包含在其中的 Java 虚拟机代码的语义。根据本规范一成不变地逐字实现其中要求的内容当然是一种可行的途径，但实现者在本规范约束下对具体实现做出修改和优化也是完全可行的，并且也推荐这样做。只要优化后 class 文件依然可以正确读取，并且包含在其中的语义能得到保持，实现者就可以选择任何方式去实现这些语义，虚拟机内部如何处理 class 文件完全是实现者自己的事情，只要它在外部接口上看起来与规范描述的一致即可。

> 这里多少存在一些例外，例如，调试器（debugger）、性能监视器（profiler）和即时代码生成器（just-in-time code generator）等都可能需要访问一些通常被认为是虚拟机“内部”的元素。在适当的情况下，Oracle 会与其他 Java 虚拟机实现者以及工具提供商一起开发这类 Java 虚拟机工具的通用接口，并推广这些接口，令其可以在整个行业中通用。

实现者可以使用这种伸缩性来让 Java 虚拟机获得更高的性能、更低的内存消耗或者更好的可移植性，选择哪种改装方式取决于 Java 虚拟机实现的目标。虚拟机实现可以考虑的方式主要有以下两种：

- 将输入的 Java 虚拟机代码在加载时或执行时翻译成另外一种虚拟机的指令集。
- 将输入的 Java 虚拟机代码在加载时或执行时翻译成宿主机 CPU 的本地指令集（有时候称 Just-In-Time 代码生成或 JIT 代码生成）。

精确定义的虚拟机和目标文件格式不应当对虚拟机实现者的创造性产生太多的限制，Java 虚拟机支持众多不同的实现，并且各种实现可以在保持兼容性的同时提供不同的新的、有趣的解决方案。

㊀ 公有设计（public design）、私有实现（private implementation）可以理解为：统一设计、各自实现。——译者注