第二章

Java 虚拟机的结构

此文档指定了一个抽象机器。它没有描述 Java 虚拟机的任何特定实现。

要正确实现 Java 虚拟机，您只需要能够读取 class 文件格式并正确执行其中指定的操作。不属于 Java 虚拟机规范的实现细节将不必要地限制实现者的创造力。例如，运行时数据区域的内存布局、使用的垃圾收集算法以及 Java 虚拟机指令的任何内部优化（例如，将其转换为机器代码）都由实现者自行决定。例如，运行时数据区域的内存布局、使用的垃圾收集算法以及 Java 虚拟机指令的任何内部优化（例如，将其转换为机器代码）都由实现者自行决定。

本规范中对 Unicode 的所有引用均与 Unicode 标准 13.0 版相关，可在 https://www.unicode.org/获得。

2.1 class 文件格式

由 Java 虚拟机执行的编译代码使用独立于硬件和操作系统的二进制格式表示，通常(但不一定)存储在一个文件中，称为 class 文件格式。class 文件格式精确地定义了类或接口的表示，包括字节排序等细节，这些细节在特定于平台的对象文件格式中可能被认为是理所当然的。

第四章, "class 文件格式", 详细介绍了 class 文件格式。

2.2 数据类型

和 Java 编程语言一样，Java 虚拟机操作两种类型：原生类型和引用类型。相应地，有两种值可以存储在变量中，作为参数传递，由方法返回，并进行操作:原生值和引用值。

Java 虚拟机期望几乎所有类型检查都在运行时之前完成，通常由编译器完成，而不必由 Java 虚拟机本身完成。原生类型的值不需要标记，也不需要在运行时进行检查以确定其类型，或者与引用类型的值区分开来。相反，Java 虚拟机的指令集使用旨在对特定类型的值进行操作的指令来区分其操作数类型。例如, iadd, ladd, fadd, 和 dadd 都是 Java 虚拟机指令，它们将两个数值相加产生一个数值结果, 但每个操作数都专门用于其操作数类型: int, long, float, 和 double, 分别地。有关 Java 虚拟机指令集中类型支持的概述，请参见§2.11.1。

Java 虚拟机包含对对象的显式支持。对象是动态分配的类实例或数组。对对象的引用被视为具有 Java 虚拟机类型 reference。类型 reference 的值可以被视为指向对象的指针。 可能存在对一个对象的多个引用。对象始终通过类型 reference 的值进行操作、传递和测试。

2.3 原生类型和值

Java 虚拟机支持的基本数据类型是数字类型, boolean 类型(§2.3.4), 和 returnAddress 类型(§2.3.3)。

数字类型包括整数类型（§2.3.1）和浮点类型（§3.2）。

整数类型是:

- byte, 其值为 8 位有符号二进制补码整数，其缺省值为零
- short, 其值为 16 位有符号二进制补码整数，其缺省值为零
- int, 其值为 32 位有符号二进制补码整数，其缺省值为零
- long, 其值为 64 位有符号二进制补码整数，其缺省值为零
- char, 其值为 16 位无符号整数，表示基本多语言平面中的 Unicode 代码点，使用 UTF-16 编码，其默认值为 null 代码点('\u0000')

浮点类型包括:

- float, 其值与可用 32 位 IEEE 754 binary32 格式表示的值完全对应，并且其缺省值为正零
- double, 其值与 64 位 IEEE 754 binary64 格式的值完全对应，并且其缺省值为正零

布尔类型的值编码了真值 true 和 false，默认值为 false。

> Java 虚拟机规范的第一版没有将布尔值视为 Java 虚拟机类型。但是，布尔值在 Java 虚拟机中支持有限。Java®虚拟机规范的第二版通过将 boolean 作为一种类型来澄清这个问题。

returnAddress 类型的值是指向 Java 虚拟机指令操作码的指针。在原生类型中，只有 returnAddress 类型与 Java 编程语言类型没有直接关联。

2.3.1 整型和值

Java 虚拟机的整型值为:

- 对于 byte, 从 -128 到 127 (-2^7 到 $2^7 - 1$), 包含的
- 对于 short, 从 -32768 到 32767 (-2^{15} 到 $2^{15} - 1$), 包含的
- 对于 int, 从 -2147483648 到 2147483647 (-2^{31} 到 $2^{31} - 1$), 包含的
- 对于 long, 从 -9223372036854775808 到 9223372036854775807 (-2^{63} 到 $2^{63} - 1$), 包含的
- 对于 char, 从 0 到 65535 包含的

2.3.2 浮点类型和值

浮点类型是 float 和 double，它们在概念上与 IEEE 754 值和操作的 32 位 binary32 和 64 位 binary64 浮点格式相关联，如 IEEE 754 标准(JLS§1.7)所规定的。

在 Java SE 15 及以后版本中，Java 虚拟机使用 IEEE 754 标准的 2019 版。在 Java SE 15 之前，Java 虚拟机使用的是 1985 年版本的 IEEE 754 标准，其中 binary32 格式被称为单精度格式，binary64 格式被称为双精度格式。

IEEE 754 不仅包括由符号和大小组成的正负数字，还包括正零和负零、正无穷和负无穷，以及特殊的非数字值(以下简称 NaN)。NaN 值用于表示某些无效操作的结果，例如 0 除以 0。float 和 double 类型的 NaN 常量都被预定义为 Float.NaN 和 Double.NaN。

浮点类型的有限非零值都可以用形式 $s \cdot m \cdot 2^{(e - N + 1)}$表示, 其中:

- *s* 为 +1 或 -1,
- m 是一个小于 2^N的整数,
- e 是在 $E_{min} = -(2^{K-1}-2)$ 和 $E_{max} = 2^{K-1}-1$ 之间的整数（闭区间），并且
- N 和 K 是依赖于类型的参数。

在这种形式中，有些值可以用多种方式表示。例如，假设一个浮点类型的值 v 可以用 s、m 和 e 的特定值表示成这种形式，那么如果 m 是偶数，e 小于 2^{K-1}，人们可以将 m 减半，并将 e 增加 1 来产生相同值 v 的第二种表示。

如果 $m \geqslant 2^{N-1}$，这种形式的表示称为归一化;否则，这种表现就被称为低于正常的。如果浮点类型的值不能以 $m \geqslant 2^{N-1}$的方式表示，则该值被称为低于正常的值，因为它的大小低于最小归一化值的大小。

表 2.3.2-A 总结了 float 和 double 的参数 N 和 K(以及派生参数 E_{min} 和 E_{max})的约束。

表 2.3.2-A. 浮点参数

参数	`float`	`double`
N	24	53
K	8	11
Emax	+127	+1023
Emin	-126	-1022

除 NaN 外，浮点值是有序的。当从最小到最大排列时，它们是负无穷、负有限非零值、正和负零、正有限非零值和正无穷。

IEEE 754 允许对其每一种 binary32 和 binary64 浮点格式使用多个不同的 NaN 值。然而，Java SE 平台通常将给定浮点类型的 NaN 值视为成单个规范值，因此该规范通常将任意 NaN 视为规范值。

在 IEEE 754 下，带有非 NaN 参数的浮点操作可能会生成 NaN 结果。IEEE 754 指定了一组 NAN 位模式，但没有规定使用哪个特定的 NAN 位模式来表示 NAN 结果；这留给了硬件体系结构。程序员可以创建具有不同位模式的 NAN 来编码，例如，回溯诊断信息。这些 NaN 值可以分别用 float 的 Float.intBitsToFloat 或 double 的 Double.longBitsToDouble 来创建。相反，要检查 NAN 值的位模式，可以分别使用 float 的 Float.floatToRawIntBits 方法或 double 的 Double.doubleToRawLongBits 方法。

正零和负零比较相等，但有其他操作可以区分它们; 例如，1.0 除以 0.0 会产生正无穷大，但 1.0 除以-0.0 会产生负无穷大。

NaN 是无序的，因此如果它们的一个或两个操作数都是 NaN，则数值比较和数值相等测试的值为 false。具体地说，当且仅当值为 NaN 时，值相对于其自身的数值相等的测试才具有值 false。如果任何一个操作数为 NaN，则数值不等性测试的值为 true。

2.3.3 returnAddress 类型和值

returnAddress 类型被 Java 虚拟机的 jsr、ret 和 jsr_w 指令(§jsr、§ret、§jsr_w)使用。returnAddress 类型的值是指向 Java 虚拟机指令操作码的指针。与数字原生类型不同，returnAddress 类型不对应于任何 Java 编程语言类型，不能由正在运行的程序修改。

2. 3. 4 boolean 类型

尽管 Java 虚拟机定义了布尔类型，但它只提供了非常有限的支持。没有专门用于操作布尔值的 Java 虚拟机指令。相反，Java 编程语言中对布尔值进行操作的表达式被编译为使用 Java 虚拟机 int 数据类型的值。

Java 虚拟机直接支持布尔数组。它的 newarray 指令(§newarray)允许创建布尔数组。使用字节数组指令 baload 和 bastore(§baload，§bastore)访问和修改布尔型数组。

> 在 Oracle 的 Java 虚拟机实现中，Java 编程语言中的布尔数组被编码为 Java 虚拟机字节数组，每个布尔元素使用 8 位。

Java 虚拟机对布尔数组组件进行编码，使用 1 表示 true，0 表示 false。当 Java 编程语言的布尔值被编译器映射到 Java 虚拟机类型 int 的值时，编译器必须使用相同的编码。

2.4 引用类型和值

有三种 reference 类型:类类型、数组类型和接口类型。它们的值分别引用动态创建的类实例、数组或实现接口的类实例或数组。

数组类型由具有单一维度的组件类型组成(其长度没有由类型给出)。数组类型的组件类型本身可以是数组类型。如果从任何数组类型开始，考虑它的组件类型，然后(如果也是数组类型)考虑该类型的组件类型，依此类推，最终必须得到一个不是数组类型的组件类型;这称为数组类型的元素类型。数组类型的元素类型必须是原生类型、类类型或接口类型。

引用值也可以是特殊的空引用，即对无对象的引用，这里将用空来表示。空引用最初没有运行时类型，但可以强制转换为任何类型。引用类型的缺省值为空。

此规范不要求将具体值编码为 null。

2.5 运行时数据区

Java 虚拟机定义了在程序执行期间使用的各种运行时数据区。其中一些数据区是在 Java 虚

拟机启动时创建的，只有在 Java 虚拟机退出时才会被销毁。其他数据区域以线程为单位。以线程为单位的数据区域在创建线程时创建，在线程退出时销毁。

2.5.1 pc 寄存器

Java 虚拟机可以同时支持多个执行线程（JLS§17）。每个 Java 虚拟机线程都有自己的 pc（程序计数器）寄存器。在任何时候，每个 Java 虚拟机线程都在执行单个方法的代码，即该线程的当前方法（§2.6）。如果该方法不是 native，则 pc 寄存器包含当前正在执行的 Java 虚拟机指令的地址。如果线程当前正在执行的方法是 native，则 Java 虚拟机的 pc 寄存器的值是未定义的。Java 虚拟机的 pc 寄存器足够宽，可以在特定平台上保存 returnAddress 或本地指针。

2.5.2 Java 虚拟机栈

每个 Java 虚拟机线程都有一个与线程同时创建的私有 Java 虚拟机堆栈。Java 虚拟机栈存储帧（§2.6）。Java 虚拟机栈类似于 C 等传统语言的栈：它保存局部变量和部分结果，并在方法调用和返回中发挥作用。因为除了 push 和 pop 帧外，Java 虚拟机栈永远不会被直接操作，所以帧可以被堆分配。Java 虚拟机栈的内存不需要是连续的。

> 在 Java 虚拟机规范的第一版中，Java 虚拟机栈被称为 Java 栈。

该规范允许 Java 虚拟机栈具有固定的大小，或者根据计算的需要动态地扩展和收缩。如果 Java 虚拟机栈的大小是固定的，那么在创建每个 Java 虚拟机栈时，可以独立选择该栈的大小。

> Java 虚拟机实现可以让程序员或用户控制 Java 虚拟机栈的初始大小，在动态扩展或收缩 Java 虚拟机栈的情况下，还可以控制最大和最小值。

以下异常情况与 Java 虚拟机栈相关:

• 如果线程中的计算需要比允许的更大的 Java 虚拟机栈，Java 虚拟机将抛出 StackOverflowError。

• 如果可以动态扩展 Java 虚拟机栈，并且尝试了扩展，但没有足够的内存来实现扩展，或者如果没有足够的可用内存来为新线程创建初始 Java 虚拟机栈，则 Java 虚拟机将抛出 OutOfMemoryError。

2.5.3 堆

Java 虚拟机有一个在所有 Java 虚拟机线程之间共享的堆。堆是运行时数据区，从中为所有类实例和数组分配内存。

堆是在虚拟机启动时创建的。对象的堆存储由自动存储管理系统(称为垃圾收集器)回收;对象永远不会显式地释放。Java 虚拟机没有采用特定类型的自动存储管理系统，可以根据实现者的系统需求选择存储管理技术。堆的大小可以是固定的，或者可以根据计算的需要进行扩展，并且如果不需要更大的堆，则可以收缩。堆的内存不需要是连续的。

> Java 虚拟机实现可以向程序员或用户提供对堆的初始大小的控制，以及如果堆可以动态扩展或收缩，则

可以控制最大和最小堆的大小。

下面是与堆相关的异常条件:

• 如果计算需要的堆超过自动存储管理系统的可用堆，Java 虚拟机将抛出 OutOfMemoryError。

2.5.4 方法区

Java 虚拟机具有在所有 Java 虚拟机线程之间共享的方法区。方法区类似于常规语言编译代码的存储区域，或类似于操作系统进程中的“文本”段。它存储每个类的结构，如运行时常量池、字段和方法数据，以及方法和构造函数的代码，包括类和接口初始化以及实例初始化中使用的特殊方法（§2.9）。

方法区在虚拟机启动时创建。虽然方法区在逻辑上是堆的一部分，但简单的实现可能选择不进行垃圾收集或压缩。本规范不规定方法区的位置或用于管理编译代码的策略。方法区域可以具有固定大小，或者可以根据计算的需要进行扩展，并且如果不需要更大的方法区域，则可以缩小。方法区的内存不需要是连续的。

Java 虚拟机实现可以向程序员或用户提供对方法区的初始大小的控制，以及在大小变化的方法区的情况下，对最大和最小方法区大小的控制。

以下异常情况与方法区相关:

• 如果方法区中的内存无法满足分配请求，Java 虚拟机将抛出 OutOfMemoryError。

2.5.5 运行时常量池

运行时常量池是 class 文件中 constant_pool 表的每个类或每个接口运行时表示（§4.4）。它包含几种常量，从编译时已知的数字字面量到必须在运行时解析的方法和字段引用。运行时常数池的功能类似于传统编程语言的符号表，尽管它包含的数据范围比典型的符号表更广泛。

每个运行时常量池都是从 Java 虚拟机的方法区分配的(§2.5.4)。类或接口的运行时常量池是在类或接口由 Java 虚拟机创建(§5.3)时构造的。

以下异常情况与类或接口的运行时常量池的构造相关:

• 在创建类或接口时，如果构建运行时常量池所需的内存超过 Java 虚拟机的方法区所能提供的内存，则 Java 虚拟机将抛出 OutOfMemoryError。

关于运行时常量池的构造，请参见§5(加载、链接和初始化)。

2.5.6 本地方法栈

Java 虚拟机的实现可以使用传统的栈，通俗地称为“C 栈”，以支持本地方法(用 Java 编程语言以外的语言编写的方法)。Java 虚拟机指令集解释器的实现(在 C 语言中)也可以使用本地方法栈。Java 虚拟机实现如果不能加载本地方法，并且本身不依赖于传统栈，则不需要提供本地方法栈。如果提供本地方法栈，则通常在创建每个线程时为每个线程分配本地方法

栈。

该规范允许本地方法栈具有固定的大小，或者根据计算的需要动态地扩展和收缩。如果本地方法栈的大小是固定的，则每个本地方法栈的大小可以在创建该栈时独立选择。

> Java 虚拟机实现可以向程序员或用户提供对本机方法栈的初始大小的控制，以及在大小变化的本地方法栈的情况下，对最大和最小方法栈大小的控制。

以下异常条件与本地方法栈相关:

• 如果线程中的计算需要比允许的更大的本地方法栈，Java 虚拟机将抛出 StackOverflowError。

• 如果本地方法栈可以动态扩展，并且尝试了本地方法栈扩展，但没有足够的内存可用，或者没有足够的内存可用来为新线程创建初始本地方法栈，则 Java 虚拟机将抛出 OutOfMemoyError。

2.6 帧

帧用于存储数据和部分结果，以及执行动态链接、方法返回值和调度异常。

每次调用方法时都会创建一个新帧。当帧的方法调用完成时，无论该完成是正常完成还是突然完成(它抛出一个未捕获的异常)，帧都会被销毁。帧是从创建帧的线程的 Java 虚拟机栈(§2.5.2)中分配的。每个帧都有自己的局部变量数组(§2.6.1)、自己的操作数栈(§2.6.2)以及对当前方法类的运行时常量池(§2.5.5)的引用。

> 可以使用附加的特定于实现的信息来扩展帧，例如调试信息。

局部变量数组和操作数栈的大小在编译时确定，并与与帧相关的方法的代码一起提供(§4.7.3)。因此，帧数据结构的大小仅取决于 Java 虚拟机的实现，并且这些结构的内存可以在方法调用时同时分配。

在给定的控制线程中的任何点上，只有一个帧是活动的，即执行方法的帧。此帧称为当前帧，其方法称为当前方法。在其中定义当前方法的类是当前类。局部变量和操作数栈上的操作通常引用当前帧。

如果帧的方法调用另一个方法或其方法完成，则它不再是当前帧。当调用方法时，将创建一个新帧，并在控制转移到新方法时成为当前帧。在方法返回时，当前帧将其方法调用的结果(如果有的话)返回给前一帧。当前一帧变成当前帧时，当前帧就会被丢弃。

请注意，由线程创建的帧是该线程局部的，不能被任何其他线程引用。

2.6.1 局部变量

每一帧(§2.6)包含一个变量数组，称为局部变量。帧的局部变量数组的长度在编译时确定，并在类或接口的二进制表示形式中提供，同时提供与该帧相关的方法代码(§4.7.3)。

单个局部变量可以保存类型为 boolean，byte，char，short，int，float，reference，或

returnAddress 的值。一对局部变量可以保存类型为 long 或 double 的值。

局部变量通过索引来处理。第一个局部变量的索引为零。当且仅当整数大于零小于局部变量数组的大小时，该整数被认为是局部变量数组的索引。

long 类型或 double 类型的值占用两个连续的局部变量。这样的值只能使用较小的索引来寻址。例如，存储在索引 n 处的局部变量数组中的 double 类型的值实际上占用了索引为 n 和 n+1 的局部变量；但是，不能从索引 n+1 处加载局部变量。可以将其存储到中。但是，这样做会使局部变量 n 的内容无效。

Java 虚拟机不要求 n 为偶数。直观地说，long 和 double 类型的值不需要在局部变量数组中按 64 位对齐。实现者可以自由决定使用为该值保留的两个局部变量来表示这些值的适当方式。

Java 虚拟机使用局部变量在方法调用时传递参数。在类方法调用时，所有参数都以从局部变量 0 开始的连续局部变量传递。在实例方法调用时，局部变量 0 始终用于传递对其上调用实例方法的对象的引用(在 Java 编程语言中的 this)。所有参数随后都会在从局部变量 1 开始的连续局部变量中传递。

2.6.2 操作数栈

每个帧(§2.6)包含一个后进先出(LIFO)的栈，称为其操作数栈。帧的操作数堆栈的最大深度在编译时确定，并随与帧相关的方法的代码一起提供(§4.7.3)。

在上下文很清楚的情况下，我们有时会将当前帧的操作数栈简称为操作数栈。

创建包含操作数栈的帧时，操作数栈为空。Java 虚拟机提供将常量或值从局部变量或字段加载到操作数栈的指令。其他 Java 虚拟机指令从操作数栈中获取操作数，对其进行操作，并将结果推回到操作数栈。操作数栈还用于准备要传递给方法的参数和接收方法结果。

例如，iadd 指令（§iadd）将两个 int 值相加。它要求要添加的 int 值是操作数栈的前两个值，由前面的指令推送到那里。这两个 int 值都从操作数栈中弹出。它们被相加，它们的和被推回操作数栈。子计算可以嵌套在操作数栈上，从而产生可由包围计算使用的值。

操作数栈上的每个条目都可以保存任何 Java 虚拟机类型的值，包括 long 或 double 类型的值。

操作数栈中的值必须以适合其类型的方式进行操作。例如，不可能推送两个 int 值并随后将其视为 long，也不可能推送到两个 float 值并随后使用 iadd 指令将它们相加。少量 Java 虚拟机指令（dup 指令（§dup）和 swap（§swap））作为原始值在运行时数据区上运行，而不考虑其特定类型；这些指令的定义方式是这样的，它们不能用来修改或拆分单独的值。这些对操作数栈操作的限制是通过 class 文件验证来实施的(§4.10)。

在任何时间点，操作数栈都具有关联的深度，其中 long 或 double 类型的值为深度贡献两个单位，而任何其他类型的值贡献一个单位。

2.6.3 动态链接

每个帧（§2.6）包含对当前方法类型的运行时常量池（§2.5.5）的引用，以支持方法代码的动态链接。方法的 class 文件代码引用要调用的方法和要通过符号引用访问的变量。动态链接将这些符号方法引用转换为具体的方法引用，根据需要加载类以解析尚未定义的符号，并将变量访问转换为与这些变量的运行时位置相关联的存储结构中的适当偏移。

方法和变量的这种后期绑定使得方法使用的其他类中的更改不太可能破坏代码。

2.6.4 正常完成方法调用

如果方法调用没有引发异常（§2.10），则方法调用正常完成，无论是直接从 Java 虚拟机还是执行显式抛出语句。如果当前方法的调用正常完成，则可能会向调用方法返回一个值。当被调用的方法执行其中一条返回指令（§2.11.8）时，会发生这种情况，其中的选择必须适合返回的值的类型（如果有）。

在这种情况下，当前帧（§2.6）用于恢复调用者的状态，包括其局部变量和操作数栈，调用者的程序计数器适当递增，以跳过方法调用指令。然后在调用方法的帧中继续正常执行，返回值（如果有）被推送到该帧的操作数栈上。

2.6.5 突然完成方法调用

如果在方法中执行 Java 虚拟机指令导致 Java 虚拟机抛出异常（§2.10），并且该异常未在方法中处理，则方法调用会突然完成。执行 athrow 指令（§athrow）也会导致显式抛出异常，如果当前方法未捕获异常，则会导致方法调用突然完成。突然完成的方法调用永远不会向其调用方返回值。

2.7 对象的表示

Java 虚拟机不要求对象具有任何特定的内部结构。

> 在 Java 虚拟机的 Oracle 的一些实现中，对类实例的引用是指向句柄的指针，句柄本身就是一对指针：一个指向包含对象方法的表，和一个指向表示对象类型的 Class 对象的指针，另一个指向从堆中为对象数据分配的内存。

2.8 浮点运算

Java 虚拟机包含 IEEE 754 标准（JLS§1.7）中规定的浮点运算的子集。

> 在 Java SE 15 及以后版本中，Java 虚拟机使用 IEEE 754 标准的 2019 版。在 Java SE 15 之前，Java 虚拟机使用的是 1985 年版本的 IEEE 754 标准，其中 binary32 格式被称为单精度格式，binary64 格式被称为双精度格式。

许多用于算术(§2.11.3)和类型转换(§2.11.4)的 Java 虚拟机指令都使用浮点数。这些指令通常对应于 IEEE 754 操作(表 2.8-A)，除了下面描述的某些指令。

表 2.8-A. 与 IEEE 754 操作的通信

指令	IEEE 754 操作
dcmp<op> (§*dcmp<op>*), *fcmp<op>* (§*fcmp<op>*)	compareQuietLess, compareQuietLessEqual, compareQuietGreater, compareQuietGreaterEqual, compareQuietEqual, compareQuietNotEqual
dadd (§*dadd*), *fadd* (§*fadd*)	addition
dsub (§*dsub*), *fsub* (§*fsub*)	subtraction
dmul (§*dmul*), *fmul* (§*fmul*)	multiplication
ddiv (§*ddiv*), *fdiv* (§*fdiv*)	division
dneg (§*dneg*), *fneg* (§*fneg*)	negate
i2d (§*i2d*), *i2f* (§*i2f*), *l2d* (§*l2d*), *l2f* (§*l2f*)	convertFromInt
d2i (§*d2i*), *d2l* (§*d2l*), *f2i* (§*f2i*), *f2l* (§*f2l*)	convertToIntegerTowardZero
d2f (§*d2f*), *f2d* (§*f2d*)	convertFormat

Java 虚拟机支持的浮点运算与 IEEE 754 标准之间的关键区别是:

• 浮点余数指令 drem(§drem)和 frem(§frem)并不对应于 IEEE 754 的余数操作。说明基于使用舍入到零舍入策略的隐含除法；IEEE 754 余数是基于使用四舍五入舍入策略的隐含除法。(舍入策略将在下面讨论。)

• 浮点取反指令 dneg(§dneg)和 fneg(§fneg)并不完全对应于 IEEE 754 的取反操作。特别是，指令不要求 NaN 操作数的符号位反转。

• Java 虚拟机的浮点指令不会抛出异常、陷阱或以其他方式通知 IEEE 754 无效操作、除零、上溢、下溢或不准确的异常情况。

• Java 虚拟机不支持 IEEE 754 信令浮点比较，并且没有信令 NaN 值。

• IEEE 754 包含的舍入方向属性与 Java 虚拟机中的舍入策略不对应。Java 虚拟机不提供任何方法来更改给定浮点指令所使用的舍入策略。

• Java 虚拟机不支持 IEEE 754 定义的 binary32 扩展和 binary64 扩展浮点格式。操作或存储浮点值时，不得使用超出 float 和 double 类型指定的扩展范围或扩展精度。

> Java 虚拟机中没有相应指令的一些 IEEE 754 操作通过 Math 和 StrictMath 类中的方法提供，包括 IEEE 754 squareRoot 操作的 sqrt 方法、IEEE 754 fusedMultiplyAdd 操作的 fma 方法和 IEEE 754 remainder 操作的 IEEEremainder 方法。

Java 虚拟机需要 IEEE 754 次标准浮点数和逐渐下溢的支持，这使得证明特定数值算法的理想属性更容易。

浮点运算是实数运算的近似。虽然实数有无限个，但特定的浮点格式只有有限个值。在 Java 虚拟机中，舍入策略是一个函数，用于将给定格式的实数映射到浮点值。对于浮点格式的可表示范围内的实数，实数线的连续段被映射为单个浮点值。在数值上等于浮点值的实数被映射到该浮点值; 例如，实数 1.5 以给定的格式映射到浮点值 1.5。Java 虚拟机定义

了两个舍入策略，如下所示:

• 四舍五入策略适用于所有浮点指令，但(i)转换为整数值和(ii)取余数除外。在四舍五入策略下，不精确的结果必须四舍五入到最接近无限精确结果的可表示值; 如果两个最近的可表示值接近相等，则选择最低有效位为零的值。

四舍五入舍入策略对应于 IEEE 754 中二进制算术的默认舍入方向属性 roundTiesToEven。

roundTiesToEven 舍入方向属性在 1985 年版本的 IEEE 754 标准中被称为“四舍五入”舍入模式。Java 虚拟机中的舍入策略以这种舍入模式命名。

• 向零舍入策略适用于(i)通过 d2i、d2l、f2i 和 f2l 指令(§d2i、§d2l、§f2i、§f2l)和(ii)浮点余数指令 drem 和 frem(§drem、§frem)将浮点值转换为整数值。在向零舍入策略下，不精确的结果被舍入到最接近的可表示值，该值的大小不大于无限精确的结果。对于转换为整数，向零舍入策略相当于截断，其中小数有效位被丢弃。

向零舍入策略对应于 IEEE 754 中二进制算法的 roundTowardZero 舍入方向属性。

roundTowardZero 舍入方向属性在 1985 年版本的 IEEE 754 标准中被称为“向零舍入”舍入模式。Java 虚拟机中的舍入策略就是以这种舍入模式命名的。

Java 虚拟机要求每个浮点指令将其浮点结果舍入到结果精度。如上所述，每个指令使用的舍入策略要么是四舍五入，要么是向零舍入。

Java 1.0 和 1.1 要求对浮点表达式进行严格的求值。严格求值意味着每个 float 操作数对应一个 IEEE 754 binary32 格式表示的值，每个 double 操作数对应一个 IEEE 754 binary64 格式表示的值，每个具有相应 IEEE 754 操作的浮点操作符匹配相同操作数的 IEEE 754 结果。

严格的计算提供了可预测的结果，但在 Java 1.0/1.1 时代常见的一些处理器家族的 Java 虚拟机实现中造成了性能问题。因此，在 Java 1.2 到 Java SE 16 中，Java SE 平台允许 Java 虚拟机实现具有一个或两个与每个浮点类型相关联的值集。float 类型与 float 值集和 float 扩展指数值集相关联，而 double 类型与 double 值集和 double 扩展指数值集相关联。float 值集对应于 IEEE 754 binary32 格式所表示的值;float 扩展指数值集具有相同数量的精度位数，但指数范围更大。同样，double 值集对应于 IEEE 754 binary64 格式所表示的值;double 扩展指数值集具有相同的精度位数，但指数范围更大。默认情况下，允许使用扩展指数值集可以改善某些处理器家族的性能问题。

为了兼容，Java 1.2 允许 class 文件禁止实现使用扩展指数值集。class 文件通过在方法的声明上设置 ACC_STRICT 标志来表示这一点。ACC_STRICT 限制了方法指令的浮点语义，对 float 操作数使用 float 值集，对 double 操作数使用 double 值集，确保此类指令的结果是完全可预测的。因此标记为 ACC_STRICT 的方法具有与 Java 1.0 和 1.1 中指定的相同的浮点语义。

在 Java SE 17 及以后的版本中，Java SE 平台总是要求对浮点表达式进行严格的求值。在执行严格的计算时存在性能问题的处理器家族的新成员不再有这种困难。该规范不再将 float 和 double 与上面描述的四个值集关联起来，ACC_STRICT 标志也不再影响浮点操作的计算。为了兼容性，在主要版本号为 46-60 的 class 文件中分配用于表示 ACC_STRICT 的位模式在主要版本编号大于 60 的 class 中未分配（即，不表示任何标志）（§4.6）。Java 虚拟机的未来版本可能会给未来 class 文件中的位模式赋予不同的含义。

2.9 特殊方法

2.9.1 实例初始化方法

类具有零个或多个实例初始化方法，每个方法通常对应于用 Java 编程语言编写的构造函数。

如果满足以下所有条件，则方法是实例初始化方法：

- 它是在类（而不是接口）中定义的。
- 它有一个特殊的名称 < init > 。
- 它是 void 的(§4.3.3)。

在类中，任何名为<init>的非 void 方法都不是实例初始化方法。在接口中，任何名为<init>的方法都不是实例初始化方法。这些方法不能被任何 Java 虚拟机指令(§4.4.2，§4.9.2)调用，并且会被格式检查(§4.6，§4.8)拒绝。

实例初始化方法的声明和使用受到 Java 虚拟机的约束。对于声明，方法的 access_flags 项和代码数组是受限的(§4.6，§4.9.2)。对于使用，实例初始化方法只能通过未初始化的类实例的 invokespecial 指令调用(§4.10.1.9)。

> 因为名称<init>在 Java 编程语言中不是有效的标识符，所以不能在用 Java 编程语言编写的程序中直接使用它。

2.9.2 类初始化方法

一个类或接口最多有一个类或接口的初始化方法，由 Java 虚拟机调用该方法进行初始化(§5.5)。

如果下列条件都为真，方法就是类或接口的初始化方法：

- 它有一个特殊的名字<clinit>。
- 它是 void 的(§4.3.3)。
- 在版本号为 51.0 或更高的 class 文件中，该方法设置了 ACC_STATIC 标志，并且不带参数(§4.6)。

> ACC_STATIC 的要求是在 Java SE 7 中引入的，在 Java SE 9 中没有参数。在版本号为 50.0 或更低的 class 文件中，名为<clinit>且为 void 的方法被视为类或接口的初始化方法，而不管其 ACC_STATIC 标志的设置或是否接受参数。

class 文件中名为<clinit>的其他方法不是类或接口初始化方法。它们永远不会被 Java 虚拟机本身调用，不能被任何 Java 虚拟机指令调用(§4.9.1)，并且被格式检查拒绝(§4.6、§4.8)。

> 因为名称<clinit>在 Java 编程语言中不是有效的标识符，所以不能在用 Java 编程语言编写的程序中直接使用它。

2.9.3 签名多态方法

如果以下所有条件都为真，则该方法为签名多态方法:

- 它声明在 java.lang.invoke.MethodHandle 类或 java.lang.invoke.VarHandle 类中。
- 它只有一个 Object[]类型的形参。
- 它设置了 ACC_VARARGS 和 ACC_NATIVE 标志。

Java 虚拟机在 invokevirtual 指令(§invokevirtual)中对签名多态方法进行特殊处理，以影响方法句柄的调用，或影响 java.lang.invoke.VarHandle 实例引用的变量的访问。

方法句柄是对底层方法、构造函数、字段或类似低级操作(§5.4.3.5)的动态强类型、可直接执行的引用，具有可选的参数转换或返回值转换。java.lang.invoke.VarHandle 的实例是对变量或变量族的动态强类型引用，包括静态字段、非静态字段、数组元素或堆外数据结构的组件。有关更多信息，请参阅 JavaSE 平台 API 中的 java.lang.invoke 包。

2.10 异常

Java 虚拟机中的异常由 Throwable 类或其子类之一的实例表示。引发异常会导致从引发异常的点立即进行非局部控制转移。

大多数异常是由于发生它们的线程的操作而同步发生的。相反，异步异常可能发生在程序执行的任何点。Java 虚拟机抛出异常有三个原因之一:

- 执行了一条 athrow 指令（§athrow）。
- Java 虚拟机同步检测到异常执行条件。这些异常不会在程序中的任意点抛出，而只会在执行以下指令后同步抛出:
  - 将异常指定为可能的结果，例如:
    > 当指令包含违反 Java 编程语言语义的操作时，例如在数组边界外进行索引。
    > 在加载或链接程序的一部分时发生错误。
  - 导致超出资源的某些限制，例如，当使用了太多内存时。
- 发生异步异常的原因是:
  - 调用类 Thread 或 ThreadGroup 的 stop 方法，或者
  - Java 虚拟机实现出现内部错误。

  一个线程可以调用 stop 方法来影响另一个线程或指定线程组中的所有线程。它们是异步的，因为它们可能发生在其他一个或多个线程的执行过程中的任何点。内部错误被认为是异步的（§6.3）。

Java 虚拟机可能允许在抛出异步异常之前发生少量有限数量的执行。这种延迟允许优化的

代码检测并抛出这些异常，在符合 Java 编程语言语义的情况下处理它们是可行的。

> 一个简单的实现可能会在每个控制传输指令点轮询异步异常。由于程序的大小是有限的，这就为检测异步异常的总延迟提供了一个界限。由于在控制传输之间不会发生异步异常，代码生成器可以灵活地在控制传输之间重新排序计算，以获得更好的性能。Marc Feeley 在 Proc. 1993 函数式编程和计算机架构会议上发表的关于库存硬件的高效轮训的论文，哥本哈根，丹麦，第 179- 187 页，推荐作为进一步阅读。

Java 虚拟机抛出的异常是精确的:当发生控制转移时，在抛出异常的点之前执行的指令的所有效果必须看起来像是已经发生了。在抛出异常的点之后发生的任何指令看起来都没有被求值。如果优化后的代码在异常发生点之后投机性地执行了一些指令，那么这些代码必须准备好对用户可见的程序状态隐藏这种投机性执行。

异常处理程序指定 Java 虚拟机代码中的偏移量范围，实现异常处理程序激活的方法，描述异常处理程序能够处理的异常类型，并指定要处理该异常的代码的位置。如果导致异常的指令的偏移量在异常处理程序的偏移量范围内，并且异常类型与异常处理程序处理的异常类相同或是其子类，则异常匹配异常处理程序。当抛出异常时，Java 虚拟机在当前方法中搜索匹配的异常处理程序。如果找到匹配的异常处理程序，系统将分支到匹配处理程序指定的异常处理代码。

如果在当前方法中未找到此类异常处理程序，则当前方法调用将突然完成（§2.6.5）。突然完成时，当前方法调用的操作数栈和局部变量将被丢弃，并弹出其帧，恢复调用方法的帧。然后在调用方帧的上下文中重新抛出异常，依此类推，继续方法调用链。如果在到达方法调用链顶部之前未找到合适的异常处理程序，则抛出异常的线程的执行将终止。

方法的异常处理程序搜索匹配的顺序很重要。在 class 文件中，每个方法的异常处理程序存储在表中（§4.7.3）。在运行时，当抛出异常时，Java 虚拟机将按照当前方法的异常处理程序出现在 class 文件中的相应异常处理程序表中的顺序，从该表的开头开始搜索当前方法的异常处理程序。

请注意，Java 虚拟机不强制方法的异常表项的嵌套或任何排序。Java 编程语言的异常处理语义只能通过与编译器的合作来实现(§3.12)。当通过其他方法生成 class 文件时，定义的搜索过程确保所有 Java 虚拟机实现的行为保持一致。

2.11 指令集摘要

Java 虚拟机指令由一个字节的操作码组成，该操作码指定要执行的操作，后面跟着零个或多个操作数，提供操作所使用的参数或数据。许多指令没有操作数，只由操作码组成。

忽略异常，Java 虚拟机解释器的内部循环是有效的

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

操作数的数量和大小由操作码决定。如果操作数的大小超过一个字节，则它首先以大端顺序-高位字节存储。例如，本地变量的一个 16 位无符号索引存储为两个无符号字节 byte1 和 byte2，这样它的值为(byte1 << 8) | byte2。

字节码指令流仅为单字节对齐。两个例外是 lookupswitch 和 tableswitch 指令(§lookupswitch, §tableswitch)，它们被填充以强制它们的一些操作数在 4 字节边界上进行内部对齐。

> 将 Java 虚拟机操作码限制在一个字节内，并放弃编译代码中的数据对齐，这反映了一种有意识的倾向于紧凑，可能会以在原始实现中牺牲一些性能为代价。一个字节的操作码也限制指令集的大小。不假设数据对齐意味着在许多机器上，必须在运行时从字节构造大于一个字节的即时数据。

2.11.1 类型和 Java 虚拟机

Java 虚拟机指令集中的大多数指令都对它们执行的操作的类型信息进行编码。例如，iload 指令（§iload）将局部变量（必须是整数）的内容加载到操作数栈中。fload 指令（§fload）对 float 值执行相同的操作。这两条指令可以具有相同的实现，但具有不同的操作码。

对于大多数类型化指令，指令类型在操作码助记符中由字母显式表示：i 表示 int 运算，l 表示 long，s 表示 short，b 表示 byte，c 表示 char，f 表示 float，d 表示 double，a 表示 reference。有些类型明确的指令在它们的助记符中没有一个类型字母。例如，arraylength 总是作用于数组对象。有些指令(如 goto，一种无条件的控制传输)不操作类型化的操作数。

考虑到 Java 虚拟机的一个字节的操作码大小，将类型编码为操作码会给其指令集的设计带来压力。如果每个类型的指令都支持 Java 虚拟机的所有运行时数据类型，那么将会有比一个字节所能表示的更多的指令。相反，Java 虚拟机的指令集为某些操作提供了较低级别的类型支持。换句话说，指令集故意不是正交的。根据需要，可以使用单独的指令在不支持和支持的数据类型之间进行转换。

表 2.11.1-A 总结了 Java 虚拟机指令集中的类型支持。通过将操作码列中的指令模板中的 T 替换为类型列中的字母来构建具有类型信息的特定指令。如果某些指令模板和类型的类型列为空，则不存在支持该操作类型的指令。例如，有一个用于 int 类型的加载指令 iLoad，但没有用于类型 byte 的加载指令。

请注意，表 2.11.1-A 中的大多数指令没有 byte、char 和 short 整数类型的形式。没有 boolean 类型的形式。编译器使用 Java 虚拟机指令对 byte 和 short 类型的字面量值进行编码，这些指令在编译时或运行时将这些值符号扩展为 int 类型的值。boolean 和 char 类型的字面量值的加载使用在编译时或运行时将字面量零扩展为 int 类型的值的指令进行编码。同样，使用 Java 虚拟机指令对 boolean、byte、short 和 char 类型的值数组的加载进行编码，这些指令将值符号扩展或零扩展为 int 类型的值。因此，对实际类型 boolean、byte、char 和 short 的值的大多数操作都是由对计算类型 int 的值进行操作的指令正确执行的。

表 2.11.1-A. Java 虚拟机指令集中的类型支持

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

表 2.11.1-B 总结了 Java 虚拟机实际类型和 Java 虚拟机计算类型之间的映射。

某些 Java 虚拟机指令（如 pop 和 swap）在操作数栈上操作，而不考虑类型；然而，此类指令仅限于使用表 2.11.1-B 中给出的某些计算类型种类的值。

表 2.11.1-B. Java 虚拟机中的实际类型和计算类型

实际类型	计算类型	种类
`boolean`	`int`	1

byte	int	1
char	int	1
short	int	1
int	int	1
float	float	1
reference	reference	1
returnAddress	returnAddress	1
long	long	2
double	double	2

2.11.2 加载和存储指令

加载和存储指令在 Java 虚拟机帧(§2.6)的局部变量(§2.6.1)和操作数栈(§2.6.2)之间传递值:

- 将局部变量加载到操作数栈: iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>.
- 将操作数栈中的值存储到局部变量中: 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: bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i>, lconst_<l>, fconst_<f>, dconst_<d>.
- 使用更宽的索引访问更多局部变量，或访问更大的立即操作数:wide。

访问对象字段和数组元素的指令(§2.11.5)也在操作数栈之间来回传输数据。

上面的指令助记符后面的字母放在尖括号中(例如，iload_<n>)表示指令族(iload_<n>的情况下，成员为 iload_0, iload_1, iload_2 和 iload_3)。这种指令族是采用一个操作数的附加通用指令（iload）的特化。对于专用指令，操作数是隐式的，不需要存储或获取。语义在其他方面是相同的（iload_0 意味着与操作数为 0 的 iload 相同）。尖括号之间的字母指定该指令族的隐式操作数类型：对于 < n > 而言，是非负整数；对于 < i > ，一个整数；对于 < l > ，一个 long；对于<f>，一个 float；对于<d>，一个 double。 在许多情况下，int 类型的形式用于对 byte、char 和 short 类型的值执行操作（§2.11.1）。

指令族的这种表示法贯穿本规范。

2.11.3 算术指令

算术指令计算的结果通常是操作数栈上两个值的函数，将结果推回到操作数栈。有两种主要的算术指令：对整数值操作的指令和对浮点值操作的指令。在每种类型中，算术指令都专门用于 Java 虚拟机数字类型。对于 byte、short 和 char 类型（§2.11.1）的值或 boolean 类型的值，不直接支持整数运算；这些操作由在 int 类型上操作的指令处理。整数和浮点

指令在溢出和除零时的行为也不同。算术指令如下:

- 加: iadd, ladd, fadd, dadd。
- 减: isub, lsub, fsub, dsub。
- 乘: imul, lmul, fmul, dmul。
- 除: idiv, ldiv, fdiv, ddiv。
- 取余: irem, lrem, frem, drem。
- 取反: ineg, lneg, fneg, dneg。
- 移位: ishl, ishr, iushr, lshl, lshr, lushr。
- 按位或: ior, lor。
- 按位与: iand, land。
- 按位异或: ixor, lxor。
- 局部变量增量: iinc。
- 比较: dcmpg, dcmpl, fcmpg, fcmpl, lcmp。

Java 编程语言整数和浮点值操作符(JLS§4.2.2,JLS§4.2.4)的语义直接由 Java 虚拟机指令集的语义支持。

Java 虚拟机不会在操作整数数据类型时提示溢出。唯一能引发异常的整数操作是整数除法指令(idiv 和 ldiv)和整数取余指令(irem 和 lrem),如果除数为零,会引发一个 ArithmeticException 异常。

在操作浮点数据类型的期间,Java 虚拟机不会指示上溢或下溢。也就是说,浮点指令永远不会导致 Java 虚拟机抛出运行时异常(不要与 IEEE 754 浮点异常混淆)。上溢的操作产生有符号的无穷;下溢操作产生一个低于正常值或有符号的零; 没有唯一数学定义结果的操作产生 NaN。所有使用 NaN 作为操作数的数值运算结果都是 NaN。

对 long 类型(lcmp)的值进行比较将执行带符号比较。

使用 IEEE 754 非信令比较对浮点类型(dcmpg, dcmpl, fcmpg, fcmpl)的值进行比较。

2.11.4 类型转换指令

类型转换指令允许在 Java 虚拟机数值类型之间进行转换。它们可以用于在用户代码中实现显式转换,或者缓解 Java 虚拟机指令集中缺乏正交性的问题。

Java 虚拟机直接支持以下扩展数字转换:

- int 到 long, float, 或 double
- long 到 float 或 double

- float 到 double

拓宽数字转换指令是 i2l、i2f、i2d、l2f、l2d 和 f2d。这些操作码的助记符很简单，因为类型化指令的命名约定以及 2 的双关用法表示“to”。 例如，i2d 指令将 int 值转换为 double。

大多数拓宽数值转换不会丢失有关总体数值大小的信息。事实上，从 int 到 long 和从 int 到 double 的拓宽转换不会丢失任何信息；数值被精确地保留。 从 float 扩展到 double 的拓宽转换也会精确保留数值。

从 int 到 float、从 long 到 float 或从 long 到 double 的转换可能会丢失精度，也就是说，可能会丢失值的一些最低有效位；结果浮点值是整数值的正确舍入版本，使用四舍五入舍入策略（§2.8）。

尽管可能会出现精度损失，但拓宽数值转换从不会导致 Java 虚拟机抛出运行时异常（不要与 IEEE 754 浮点异常混淆）。

从 int 到 long 的拓宽数字转换符号扩展了 int 值的二进制补码表示，以填充成更宽的格式。从 char 到整数类型的拓宽数字转换零扩展了 char 值的表示，以填充成更宽的格式。

请注意，不存在从整数类型 byte、char 和 short 到 int 类型的拓宽数字转换。如§2.11.1 所述，byte、char 和 short 类型的值在内部扩展为 int 类型，使得这些转换是隐式的。

Java 虚拟机还直接支持以下收窄数值转换：

- int 到 byte, short, 或 char
- long 到 int
- float 到 int 或 long
- double 到 int, long, 或 float

收窄数字转换指令是 i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l 和 d2f。缩小数值转换可以产生不同符号的值、不同数量级的值，或两者兼有;因此，它可能会失去精度。

从 int 或 long 到整型 T 的收治安数值转换只会丢弃除 n 个最低阶位以外的所有位，其中 n 是用于表示类型 T 的位数。这可能导致结果值与输入值的符号不同。

在浮点值到整数类型 T 的收窄数值转换中，其中 T 是 int 或 long，浮点值转换如下：

- 如果浮点值为 NaN，则转换结果为整 int 或 long 0。
- 否则，如果浮点值不是无穷大，则使用向零舍入策略将浮点值舍入为整数值 V（§2.8）。有两种情况：
  - 如果 T 是 long，并且该整数值可以表示为 long，则结果是 long 值 V。
  - 如果 T 的类型为 int，并且该整数值可以表示为 int，则结果为 int 值 V。
- 否则:

- 要么该值必须太小（大幅度的负值或负无穷大），结果是 int 或 long 类型的最小可表示值。
- 或者该值必须太大（大幅度的正值或正无穷大），结果是 int 或 long 类型的最大可表示值。

从 double 到 float 的收窄数值转换符合 IEEE 754。使用四舍五入舍入策略（第 2.8 节）对结果进行正确舍入。太小而不能表示为 float 的值被转换为 float 类型的正零或负零；太大而无法表示为 float 的值将转换为正无穷大或负无穷大。double NaN 始终转换为 float NaN。

尽管可能发生上溢、下溢或精度损失，但数值类型之间的收窄转换不会导致 Java 虚拟机抛出运行时异常（不要与 IEEE 754 浮点异常混淆）。

2.11.5 对象创建和操作

尽管类实例和数组都是对象，但 Java 虚拟机使用不同的指令集创建和操作类实例和数组：

- 创建一个新类实例: new。
- 创建一个新数组: newarray, anewarray, multianewarray。
- 类的访问字段（静态字段，称为类变量）和类实例的字段（非静态字段，也称为实例变量）: getstatic, putstatic, getfield, putfield。
- 将数组组件加载到操作数栈: baload, caload, saload, iaload, laload, faload, daload, aaload。
- 将操作数栈中的值存储为数组组件: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore。
- 获得数组的长度: arraylength。
- 检查类实例或数组的属性: instanceof, checkcast。

2.11.6 操作数栈管理指令

为直接操作操作数栈提供了许多指令: pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap.

2.11.7 控制转移指令

控制转移指令有条件地或无条件地使 Java 虚拟机继续执行以下控制转移指令以外的指令。它们是：

- 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。
- 复合条件分支: tableswitch, lookupswitch。
- 无条件分支: goto, goto_w, jsr, jsr_w, ret。

Java 虚拟机有不同的指令集，它们有条件地根据 int 和 reference 类型数据的比较进行分支。

它也有不同的条件分支指令来测试空值引用，因此不需要为空值指定具体的值(§2.4)。

boolean、byte、char 和 short 类型数据比较的条件分支使用 int 比较指令执行(§2.11.1)。long、float 或 double 类型的数据之间比较的条件分支是使用比较数据并产生比较的 int 结果的指令(§2.11.3)启动的。随后的 int 比较指令测试这个结果并影响条件转移。由于强调 int 类型比较，Java 虚拟机为 int 类型提供了丰富的条件分支指令。

所有 int 条件控制传输指令都执行有符号比较。

2.11.8 方法调用和返回指令

以下 5 条指令调用方法:

• invokvirtual 调用对象的实例方法，分派对象的(虚拟)类型。 这是 Java 编程语言中常见的方法分派。

• invokeinterface 调用一个接口方法，搜索由特定运行时对象实现的方法以找到适当的方法。

• invokspecial 调用需要特殊处理的实例方法，可以是实例初始化方法(§2.9.1)，也可以是当前类或其超类型的方法。

• invokestatic 调用命名类中的类(静态)方法。

• invokedynamic 调用方法，该方法是绑定到 invokedynamic 指令的调用站点对象的目标。调用站点对象被 Java 虚拟机绑定到 invokedynamic 指令的特定词法出现，这是在第一次执行指令之前运行引导方法的结果。因此，与调用方法的其他指令不同，invokedynamic 指令的每次出现都有一个惟一的链接状态。

方法返回指令根据返回类型区分为 ireturn(用于返回 boolean、byte、char、short 或 int 类型的值)、lreturn、freturn、return 和 areturn。此外，返回指令用于从声明为 void 的方法、实例初始化方法以及类或接口初始化方法返回。

2.11.9 抛出异常

使用 athrow 指令以编程方式引发异常。如果检测到异常情况，各种 Java 虚拟机指令也会引发异常。

2.11.10 同步

Java 虚拟机支持通过一个同步构造（监视器）同步方法和方法中的指令序列。

作为方法调用和返回的一部分，隐式执行方法级同步（§2.11.8）。在运行时常量池的 method_info 结构（§4.6）中，通过 ACC_SYNCHRONIZED 标志来区分同步方法，该标志由方法调用指令检查。当调用为其设置了 ACC_SYNCHRONIZED 的方法时，执行线程进入监视器，调用方法本身，并退出监视器，无论方法调用是正常完成还是突然完成。在执行线程拥有监视器期间，任何其他线程都不能进入监视器。如果在调用同步方法期间引发异常，并且同步方法不处理该异常，则在从同步方法中重新引发异常之前，将自动退出该方法的监视

器。

指令序列的同步通常用于对 Java 编程语言的同步块进行编码。Java 虚拟机提供了 monitorenter 和 monitorexit 指令来支持这种语言结构。同步块的正确实现需要面向 Java 虚拟机的编译器的合作(§3.14)。

结构化锁定是这样一种情况:在方法调用期间，给定监视器上的每个退出都与该监视器上的前一个进入匹配。由于不能保证提交给 Java 虚拟机的所有代码都将执行结构化锁定，所以 Java 虚拟机的实现允许但不是必须执行以下两条保证结构化锁定的规则。设 T 为线程，M 为监视器。则:

1. 在方法调用期间，T 对 M 执行的监视器进入数量必须等于方法调用期间 T 对 M 执行的监视器退出数量，无论方法调用是正常完成还是突然完成。

2. 在方法调用期间，T 在 M 上执行的监控器退出的数量不能超过自方法调用以来 T 在 M 上执行的监控器进入的数量。

注意，Java 虚拟机在调用同步方法时自动执行的监视器进入和退出被认为是在调用方法的过程中发生的。

2.12 类库

Java 虚拟机必须为 Java SE 平台类库的实现提供足够的支持。如果没有 Java 虚拟机的合作，这些库中的一些类就无法实现。

- 可能需要 Java 虚拟机特殊支持的类包括:
- 反射，例如 java.lang.reflect 包中的类和类 Class。
- 加载和创建类或接口。最明显的例子是 ClassLoader 类。
- 链接和初始化类或接口。上面引用的示例类也属于这一类。
- 安全, 例如包 java.security 中的类和其他类比如 SecurityManager。
- 多线程, 比如类 Thread。
- 若引用, 例如包 java.lang.ref 中的类。

上面的列表是为了说明，而不是全面的。这些类或它们提供的功能的详尽列表超出了本规范的范围。有关详细信息，请参阅 Java SE 平台类库的规范。

2.13 公共设计，私有实现

到目前为止，本规范已经勾勒出了 Java 虚拟机的公共视图：class 文件格式和指令集。这些组件对于 Java 虚拟机的硬件、操作系统和实现独立性至关重要。实现者可能更愿意将它们视为在每个实现 JavaSE 平台的主机之间安全地通信程序片段的一种手段，而不是将其视

为要严格遵循的蓝图。

理解公共设计和私有实现之间的界线是很重要的。Java 虚拟机实现必须能够读取 class 文件，并且必须准确地实现其中的 Java 虚拟机代码的语义。 这样做的一种方法是将本文档作为一个规范，并从字面上实现该规范。但是，对于实现者来说，在本规范的约束下修改或优化实现也是完全可行和理想的。只要可以读取class文件格式，并维护其代码的语义，实现者就可以以任何方式实现这些语义。只要仔细维护正确的外部接口，“内部的”就是实现者的事情。

> 有一些例外:调试器、分析器和即时代码生成器都需要访问 Java 虚拟机的元素，这些元素通常被认为是“内部的”。 在适当的情况下，Oracle 与其他 Java 虚拟机实现者和工具供应商合作，为这些工具开发 Java 虚拟机的通用接口，并在整个行业推广这些接口。

实现者可以使用这种灵活性来定制 Java 虚拟机实现，以实现高性能、低内存使用或可移植性。在一个给定的实现中什么是有意义的取决于该实现的目标。实施选项的范围包括以下内容:

- 在加载时或执行期间将 Java 虚拟机代码转换为另一台虚拟机的指令集。
- 在加载时或执行期间将 Java 虚拟机代码转换为主机 CPU 的本机指令集(有时称为即时代码生成或 JIT 代码生成)。

精确定义的虚拟机和对象文件格式的存在不需要显著限制实现者的创造性。Java 虚拟机被设计为支持许多不同的实现，提供了新的、有趣的解决方案，同时保持了实现之间的兼容性。