



<b>第一章 引言</b>	.....	(1)
1.1 一点历史	.....	(1)
1.2 Java 虚拟机	.....	(1)
1.3 各章概述	.....	(2)
<b>第二章 Java 概念</b>	.....	(3)
2.1 Unicode	.....	(3)
2.2 标识符	.....	(3)
2.3 文字	.....	(4)
2.4 类型和值	.....	(4)
2.4.1 基本类型和值	.....	(4)
2.4.2 整型值上的操作符	.....	(5)
2.4.3 浮点值上的操作符	.....	(5)
2.4.4 boolean 值上的操作符	.....	(5)
2.4.5 引用类型、对象和引用值	.....	(6)
2.4.6 类 Object	.....	(6)
2.4.7 类 String	.....	(6)
2.4.8 对象上的操作符	.....	(6)
2.5 变量	.....	(6)
2.5.1 变量的初始值	.....	(7)
2.5.2 变量具有类型, 对象具有类	.....	(8)
2.6 转换和提升	.....	(8)
2.6.1 等同转换	.....	(9)
2.6.2 放宽基本转换	.....	(9)
2.6.3 缩窄基本转换	.....	(9)
2.6.4 放宽引用转换	.....	(10)
2.6.5 缩窄引用转换	.....	(10)
2.6.6 赋值转换	.....	(10)
2.6.7 方法调用转换	.....	(11)
2.6.8 类型转换	.....	(11)
2.6.9 数值提升	.....	(12)
2.7 名称和包	.....	(12)
2.7.1 名称	.....	(12)
2.7.2 包	.....	(12)
2.7.3 成员	.....	(13)
2.7.4 包成员	.....	(13)
2.7.5 类类型的成员	.....	(13)

2.7.6	接口类型的成员	(13)
2.7.7	数组类型的成员	(13)
2.7.8	限定名称和访问控制	(14)
2.7.9	完整限定名称	(14)
2.8	类	(15)
2.8.1	类名称	(15)
2.8.2	类修饰符	(15)
2.8.3	超类和子类	(15)
2.8.4	类成员	(15)
2.9	域	(16)
2.9.1	域修饰符	(16)
2.9.2	域的初始化	(17)
2.10	方法	(17)
2.10.1	形式函数	(17)
2.10.2	签名	(17)
2.10.3	方法修饰符	(17)
2.11	静态初始化函数	(18)
2.12	构造函数	(18)
2.13	接口	(18)
2.13.1	接口修饰符	(19)
2.13.2	超接口	(19)
2.13.3	接口成员	(19)
2.13.4	接口(常数)域	(19)
2.13.5	接口(抽象)方法	(20)
2.13.6	接口中的覆盖、继承和重载	(20)
2.14	数组	(20)
2.14.1	数组类型	(20)
2.14.2	数组变量	(21)
2.14.3	数组创建	(21)
2.14.4	数组访问	(21)
2.15	异常	(21)
2.15.1	引起异常的原因	(22)
2.15.2	处理异常	(22)
2.15.3	异常层次	(23)
2.15.4	类 Exception 和 RuntimeException	(24)
2.16	执行	(25)
2.16.1	虚拟机启动	(25)
2.16.2	装载	(26)
2.16.3	链接:检验、准备和解析	(27)
2.16.4	初始化	(28)
2.16.5	详细的初始化过程	(29)
2.16.6	新的类实例的创建	(30)

2.16.7	类实例的终止	(31)
2.16.8	类和接口的终止和卸载	(32)
2.16.9	虚拟机退出	(32)
2.17	线程	(32)
<b>第三章</b>	<b>Java 虚拟机的结构</b>	<b>(35)</b>
3.1	数据类型	(35)
3.2	基本类型和值	(35)
3.2.1	整型和值	(36)
3.2.2	浮点型和值	(36)
3.2.3	returnAddress 类型和值	(37)
3.2.4	没有 boolean 类型	(37)
3.3	引用类型和值	(37)
3.4	字	(37)
3.5	运行期数据区	(37)
3.5.1	pc 寄存器	(37)
3.5.2	Java 栈	(38)
3.5.3	堆	(38)
3.5.4	方法区	(39)
3.5.5	常数池	(39)
3.5.6	自身方法栈	(40)
3.6	框架	(40)
3.6.1	局部变量	(41)
3.6.2	操作数栈	(41)
3.6.3	动态链接	(41)
3.6.4	正常的方法结束	(41)
3.6.5	不正常的方法结束	(42)
3.6.6	附加信息	(42)
3.7	对象的表示	(42)
3.8	特殊的初始化方法	(42)
3.9	异常	(43)
3.10	class 文件格式	(43)
3.11	指令集概述	(43)
3.11.1	类型和 Java 虚拟机	(44)
3.11.2	装载和存储指令	(46)
3.11.3	运算指令	(46)
3.11.4	类型转换指令	(47)
3.11.5	对象创建和操纵	(48)
3.11.6	操作数栈管理指令	(49)
3.11.7	控制转移指令	(49)
3.11.8	方法调用和返回指令	(49)
3.11.9	抛出和处理异常	(50)

3.11.10 实现 finally .....	(50)
3.11.11 同步 .....	(50)
3.12 公共设计,私有实现.....	(50)
<b>第四章 class 文件格式 .....</b>	<b>(51)</b>
4.1 ClassFile .....	(51)
4.2 完整限定类名称的内部形式.....	(54)
4.3 描述符.....	(54)
4.3.1 语法记号 .....	(55)
4.3.2 域描述符 .....	(55)
4.3.3 方法描述符 .....	(56)
4.4 常数池.....	(56)
4.4.1 CONSTANT_Class .....	(57)
4.4.2 CONSTANT_Fieldref, CONSTANT_Methodref 和 CONSTANT_InterfaceMethodref .....	(58)
4.4.3 CONSTANT_String .....	(59)
4.4.4 CONSTANT_Integer 和 CONSTANT_Float .....	(59)
4.4.5 CONSTANT_Long 和 CONSTANT_Double .....	(60)
4.4.6 CONSTANT_NameAndType .....	(61)
4.4.7 CONSTANT_Utf8 .....	(62)
4.5 域.....	(63)
4.6 方法.....	(64)
4.7 属性.....	(66)
4.7.1 定义和命名新属性 .....	(66)
4.7.2 SourceFile 属性 .....	(67)
4.7.3 ConstantValue 属性 .....	(67)
4.7.4 Code 属性 .....	(68)
4.7.5 Exceptions 属性 .....	(70)
4.7.6 LineNumberTable 属性 .....	(71)
4.7.7 LocalVariableTable 属性 .....	(72)
4.8 对 Java 虚拟机代码的约束 .....	(73)
4.8.1 静态约束 .....	(73)
4.8.2 结构约束 .....	(75)
4.9 class 文件的检验 .....	(77)
4.9.1 检验进程 .....	(78)
4.9.2 字节码检验器 .....	(79)
4.9.3 长整数和双精度数 .....	(81)
4.9.4 实例初始化方法和新创建的对象 .....	(81)
4.9.5 异常处理者 .....	(82)
4.9.6 异常和 finally .....	(82)
4.10 Java 虚拟机和 class 文件格式的限制 .....	(84)
<b>第五章 常数池解析 .....</b>	<b>(85)</b>

5.1	类和接口解析.....	(86)
5.1.1	不由类装载器装载的当前类或接口 .....	(86)
5.1.2	由类装载器装载的当前类或接口 .....	(88)
5.1.3	数组类 .....	(89)
5.2	域和方法解析.....	(90)
5.3	接口方法解析.....	(90)
5.4	字符串解析.....	(90)
5.5	其他常数池项的解析.....	(91)
<b>第六章</b>	<b>Java 虚拟机指令集 .....</b>	<b>(92)</b>
6.1	假定：“必须”的含义 .....	(92)
6.2	保留操作码.....	(92)
6.3	虚拟机错误.....	(92)
6.4	Java 虚拟机指令集 .....	(93)
<b>第七章</b>	<b>为 Java 虚拟机编译 .....</b>	<b>(183)</b>
7.1	范例格式 .....	(183)
7.2	常数、局部变量和控制构造的使用.....	(184)
7.3	运算 .....	(188)
7.4	访问常数池 .....	(189)
7.5	更多控制范例 .....	(190)
7.6	接收参数 .....	(193)
7.7	调用方法 .....	(194)
7.8	处理类实例 .....	(196)
7.9	数组 .....	(198)
7.10	编译开关.....	(200)
7.11	对操作数栈的操作.....	(202)
7.12	抛出和处理异常.....	(203)
7.13	编译 finally .....	(207)
7.14	同步.....	(210)
<b>第八章</b>	<b>线程和锁.....</b>	<b>(212)</b>
8.1	术语和框架 .....	(212)
8.2	执行顺序和一致性 .....	(213)
8.3	有关变量的规则 .....	(214)
8.4	Double 和 Long 变量的非原子处理 .....	(215)
8.5	有关锁的规则 .....	(215)
8.6	有关锁和变量的交互作用的规则 .....	(215)
8.7	有关易变变量的规则 .....	(216)
8.8	先见存储操作 .....	(216)
8.9	讨论 .....	(217)
8.10	范例：可能的交换 .....	(217)

8.11	范例:无序写 .....	(220)
8.12	线程.....	(222)
8.13	锁和同步.....	(222)
8.14	等待集和通知.....	(222)
<b>第九章</b>	<b>优化.....</b>	<b>(224)</b>
9.1	通过重写动态链接 .....	(224)
9.2	_quick 伪指令 .....	(224)
<b>第十章</b>	<b>操作码的操作码助记符.....</b>	<b>(247)</b>

# 第一章 引言

## 1.1 一点历史

Java 是一种一般用途的、并发的、面向对象的程序设计语言。它的语法和 C 与 C++ 相似，但是它省略了许多使 C 和 C++ 复杂的、易混淆和不安全的特性。Java 最初是为处理编制联网消费者设备软件中的问题而开发的。它被设计成支持多主机体系结构，并允许软件组件的安全发送。为了达到这些要求，编译后的 Java 代码必须可以在网络间移植，在任何客户机上操作，并对客户保证其运行是安全的。

world Wide Web 的流行使 Java 的这些属性更加令人感兴趣。互联网展示了怎样通过简单的方法来获得具有丰富媒体的内容。Web 浏览器，例如 Mosaic，使成千上万的人能够在网上漫游，并使 Web 成为大众文化的浪峰部分。最终有一种媒体，在那里你所看到的和听到的基本上是一样的，不论你使用的是 Mac、PC 还是 UNIX 机器，不论你是联接在一个高速网络上还是使用一个缓慢的调制解调器。

Web 迷们很快就发现 Web 的 HTML 文档格式支持的内容太有限了。HTML 的扩展（例如 forms），只是使这些限制更加明显，使人清楚地意识到，没有什么浏览器能够囊括用户想要的所有特点。可扩展性才是解决方案。

Sun 的 HotJava 浏览器通过允许把 Java 程序嵌入到 HTML 页中，展示了 Java 的令人感兴趣的属性。这些叫作 applet 的程序同 HTML 页一起透明地下载到 HotJava 浏览器中，并在其中显示。在被浏览器接受之前，applets 被仔细地检查以确保它们是安全的。和 HTML 页一样，编译后的 Java 程序是与网络和平台无关的。Applet 的行为相同，与它们来自何处或者与它们被装载运行在何种机器上无关。

由于 Java 是扩展语言，web 浏览器不再限于固定的能力。程序员可以对一个 applet 只写一次，它将可以在任何地方的任何机器上运行。Java 驱动的 web 页的访问者可以使用在 Web 页上找到的内容，确信这些内容不会损害他们的机器。

Java 展示了使用互联网发送软件的新方法。这个新的范例超越了浏览器，我们认为它是一个具有改变计算过程潜力的革新。

## 1.2 Java 虚拟机

Java 虚拟机是 Sun 的 Java 程序设计语言的基石。它是 Java 技术的组成部分。Java 技术负责 Java 的跨平台传输、编译后的短小的代码，保护用户不受恶意程序的侵害。

Java 虚拟机是一个抽象的计算机，与实际的计算机一样，它具有一个指令集并使用不同的存储器区域。使用一台虚拟机实现一种程序设计语言是很普通的；最著名的虚拟机可能是 UCSD Pascal 的 P-Code 机器。

Sun Microsystems 公司做成的 Java 虚拟机的第一个原型，实现在一个代表现代个人数字助理（PDA）的手持设备上，用软件仿真了 Java 虚拟机的指令集。Sun 现在的 Java 版本 —

Java 开发者工具(JDK)1.0.2 版——在 win32、MacOS 和 Solaris 平台上仿真了 Java 虚拟机。但是,Java 虚拟机并不假定任何实现技术或者主机平台。它并非得是解释型的,它也可以像传统的程序设计语言一样,通过把它的指令集编译成一个实际的 CPU 的指令集来实现。它也可以用微代码或者直接用芯片实现。

Java 虚拟机不识别 Java 程序设计语言,它只识别一种特殊的文件格式,即 class 文件格式。一个 class 文件包含 Java 虚拟机指令(或者 bytecodes)和一个符号表以及其他辅助信息。

为了安全,Java 虚拟机对 class 文件的代码强加了很强的格式和结构限制。但是,任何具有能够按照有效的 class 文件表达的功能的语言,都可以由 Java 虚拟机作主机。受到一个广泛可用的、与机器无关的平台的吸引,其他语言的实现者正在转向 Java 虚拟机作为他们的语言的发送工具。将来,我们将考虑对 Java 虚拟机作有限的扩展,以更好地支持其他语言。

### 1.3 各章概述

本书其余部分结构如下:

- 第二章给出本书其余部分所需的 Java 概念以及术语的概览。
- 第三章给出 Java 虚拟机的概览。
- 第四章定义 class 文件格式,它是编译后的 Java 代码的平台和与实现无关的文件格式。
- 第五章描述常数池的运行期管理。
- 第六章描述 Java 虚拟机的指令集。按照操作码助记符的字母顺序,展示指令。
- 第七章给出了将 Java 代码编译成 Java 虚拟机指令集的范例。
- 第八章描述 Java 虚拟机线程以及它们与存储器的交互作用。
- 第九章描述 Sun 的 Java 虚拟机实现使用的优化。虽然严格地说这不是规范的一部分,但是它本身是一个有用的技术,又是这类可用于 Java 虚拟机实现技术的一个范例。
- 第十章给出按照操作码值索引的 Java 虚拟机操作码助记符表。

## 第二章 Java 概念

Java 虚拟机被设计,以支持 Java 程序设计语言,因此需要 Java 语言中的一些概念和词汇来理解虚拟机。本章给出足够的对 Java 的概览以支持后面对 Java 虚拟机的讨论,它的材料压缩自 James Gosling、Bill Joy 以及 Guy Steele 所写的《Java 语言规范》一书中对 Java 语言的完整讨论,有关材料的细节和范例请参考该书。熟悉该书的读者可以跳过这一章。熟悉 Java 但不熟悉《Java 语言规范》的读者,应当至少跳读本章所介绍的术语。

本章不打算提供 Java 语言的介绍或者完整的处理。对于 Java 的介绍参见 Ken Arnold 和 James Gosling 所写的《Java 编程语言》一书。

### 2.1 Unicode

Java 程序用 Unicode 字符编码1.1.5版编写。Unicode 字符编码1.1.5版在 The Unicode Standard: Worldwide Character Encoding, Version 1.0, Volume 1, ISBN0-201-56788-1, 和 Volume 2, ISBN0-201-60845-6 中规范。关于 Unicode 1.1.5 的最新信息可在 <ftp://unicode.org> 处获得。最新信息中有几个小的错误,错误的更正参考《Java 语言规范》一书,那里出版的 Unicode 的更新信息将发到 URL <http://java.sun.com/Series>。

除了注释和标识符(§ 2.2)和字符与字符串文字(§ 2.3)的内容,Java 程序中的所有输入元素都只由 ASCII 字符形式。ASCII(ANSI X3.4)是美国信息交换标准代码。Unicode 字符的前128个字符编码是 ASCII 字符。

### 2.2 标识符

标识符(Identifier)是不限长度的 Unicode 字母(letters)和数字(digits)的序列,序列的第一个字符必须是字母。字母和数字可以从整个 Unicode 字符集中选择,Unicode 字符集支持当今世界上使用的绝大多数书写字母。这允许 Java 程序员在他们的程序中使用以他们的母语编写的标识符。

当被传递给一个被认为是 Java 标识符中的字母的 Unicode 字符时,Java 方法 Character.isJavaLetter 返回 true。当被传递给一个被认为是 Java 标识符中的字母或者数字的 Unicode 字符时。

两个标识符只有在它们的每个字母或者数字具有相同的 Unicode 字符时才是相同的,具有相同外形的标识符仍然可以是不同的。标识符不能与 Java 关键字或者布尔文字(true 或者 false)相同。

## 2.3 文字

文字(literal)是代表基本类型(§ 2.4.1)、String 类型(§ 2.4.7)或者 null 类型(§ 2.4)的值的源代码。字符串文字以及更一般的常数表达式的值的字符串,被用方法 String.intern“内部化”以共享唯一的实例。

null 类型有一个值,即 null 引用,用文字 null 代表。

boolean 类型有两个值,用文字 true 和 false 代表。

## 2.4 类型和值

Java 是一种强类型的(strong typed)语言,这表示每个变量和每个表达式具有一个编译期已知的类型。类型限制变量(§ 2.5)可持有的值,或者表达式可产生的值,限制这些值上支持的操作并决定这些操作的意义。强类型帮助在编译期检测错误。

Java 语言的类型分成两类:基本类型(primitive type)(§ 2.4.1)和引用类型(reference type)(§ 2.4.5)。还有一个特殊的 null 类型(null type),即表达或 null 的类型,它没有名称。null 引用是一个 null 表达式的唯一可能的值,并且总是可以转换为任何引用类型。实际上,Java 程序员可以忽略 null 类型,而把 null 假设为一个可以具有任意引用类型的特殊的文字。

与基本类型和引用类型相对应,有两类数据值。它们可以被存储在变量中,被作为参数传递,被方法返回,以及对其进行操作。它们是基本值(primitive value)(§ 2.4.1)和引用值(reference value)(§ 2.4.5)。

### 2.4.1 基本类型和值

基本类型是由 Java 语言预定义并由一个保留字命名的类型。基本值不与别的基本值共享状态。一个基本类型的变量总是持有它的类型的一个基本值。<sup>[1]</sup>

基本类型是 boolean 类型和数值类型(numeric type)。数值类型是整型(integral types)和浮点型(floating-point types)。

整型是 byte、short、int 和 long,它们的值分别是8位、16位、32位和64位有符号二进制补码整数。整型还有 char,它的值是代表 Unicode 字符的16位无符号整数。

浮点型是 float,它的值是32位 IEEE 754浮点数,和 double,它的值是64位 IEEE 754浮点数。这些浮点数在 IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985(IEEE, New York)中规范。IEEE 754标准不仅包括正的和负的有符号数量数字,还包括正和负零,正和负无穷(infinity),以及一个特殊的 Not-a-Number 值(以后缩写为 NaN)。NaN 值用于代表某些操作,例如零被零除的结果。

boolean 类型具有真值 true 和 false。

---

[1] 注意局部变量不是在创建时被初始化,并且只在被赋给一个值时才认为它持有一个值。

#### 2. 4. 2 整型值上的操作符

Java 提供了一些作用于整型值上的操作符,包括数值比较(其结果为 boolean 类型的值)、算术操作符、递增和递减、按位逻辑和移位操作符以及数值类型转换(§ 2. 6. 8)。

某些单目或者双目操作符的操作数可能会被数值提升(§ 2. 6. 9)。

内置的整数操作符不用任何方式指示溢出或者下溢,它们包装在溢出或者下溢的周围。仅有的能够抛出异常的整数操作符是整数除和整数余数操作符。如果右操作数是零,它们可能抛出一个 ArithmenticException。

任何整型值都可以与任何数值类型转换。在整型和 boolean 类型之间不能转换。

#### 2. 4. 3 浮点值上的操作符

Java 提供了一些作用于浮点值上的操作符,包括数值比较(其结果是 boolean 类型的值)、算术操作符、递增和递减、以及数值类型转换(§ 2. 6. 8)。

如果一个双目操作符的操作数至少有一个是浮点型的,则该操作是一个浮点操作,即使另一个操作数是整型的。某些单目或者双目操作符的操作数可能会被数值提升(§ 2. 6. 9)。

浮点数字上的操作符确切地按照 IEEE 754 规范进行操作。特别地,Java 要求支持 IEEE 754 非规格化(denomalized)浮点数和逐渐下溢(gradual underflow),这使得易于试验特定数值算法的适当属性。

Java 要求浮点算术的每个浮点操作符把它的浮点结果舍入到结果的精度。不精确的(inexact)结果必须舍入到最接近无穷精确结果的可表示的值。如果两个最接近的可表示值同样接近无穷精确结果,则选择带有最低位零的那个表示值。这就是称为最接近舍入(round-to-nearest)的 IEEE 754 标准的缺省舍入模式。

Java 在把一个浮点值转换成一个整数(§ 2. 6. 3)时使用向零舍入(round-to-zero)模式。向零舍入模式截断数字,抛弃尾数位。向零舍入选择最接近的、其数量不大于无穷精确结果的格式的值作为它的结果。

Java 浮点操作符不产生异常(§ 2. 15)。溢出的操作产生一个有符号的无穷;下溢的操作产生一个有符号零;没有数学定义的结果的操作产生 NaN。所有的以 NaN 作为一个操作数的数值操作(除了数值比较)产生一个 NaN 结果。

任何浮点类型的值都可以与任何数值类型转换(§ 2. 6. 8)。在浮点型与 boolean 型之间不能转换。

#### 2. 4. 4 boolean 值上的操作符

布尔操作符包括关系操作符和逻辑操作符。只有 boolean 表达式可以用于 Java 的控制流语句中,以及作为条件操作符?:的第一个操作数。一个整数值 x 可以转换成一个 boolean 类型值,按照 C 语言的约定,对于表达式  $x != 0$ ,任何非零的值都是 true。一个对象的引用 obj 可以转换成一个 boolean 类型值,按照 C 语言的约定,对于表达式  $obj != null$ ,任何除了 null 的引用都是 true。

在 boolean 类型和其他类型之间不能转换。

## 2.4.5 引用类型、对象和引用值

有三种引用类型：类类型(class types)(§ 2.8)，接口类型(interface types)(§ 2.13)和数组类型(array types)§ 2.14)。对象(object)是动态创建的类的实例或者数组。引用值(通常只是引用(reference))是这些对象的指针以及一个特殊的 null 引用，null 引用不引用任何对象。

类的实例由类实例创建表达式(class instance creation expression)，或者通过调用类 class 的方法 newInstance 显式地创建。数组由数组创建表达式(array creation expression)显式地创建。对象在 Java 的堆中创建，并且在不再引用它时作为垃圾回收。对象不由显式的 Java 语言指令回收或者释放。

同一个对象可以有多个引用。绝大多数对象具有状态，状态存储在类实例对象域中或者存储在数组对象成分变量中。如果两个变量包含同一个对象的引用，则该对象的状态可以用一个变量对该对象的引用修改，并且改变的状态可以通过另一个变量的引用观测到。

每个对象具有一个相应的锁(lock)(§ 2.17),(§ 8.13)。这个锁由 synchronized 方法和 synchronized 语句使用，以提供对多线程(§ 2.17, § 8.12)并发访问状态的控制。

引用类型形成一个层次。除了类 Object(§ 2.4.6)以外，每个类类型是另一类类型的子类。类 Object 是所有其他类类型的超类(§ 2.8.3)。所有的对象，包括数组，都支持类 Object 的方法。字符串文字(§ 2.3)是类 String(§ 2.4.7)实例的引用。

## 2.4.6 类 Object

标准类 Object 是所有其他类的超类(§ 2.8.3)。一个 Object 类型的变量可以持有任一对象的引用，不论该对象是类的实例还是数组。所有的类和数组类型都继承了类 Object 的方法。

## 2.4.7 类 String

类 String 的实例代表 Unicode 字符(§ 2.1)的序列。一个 String 对象具有一个固定不变的值。字符串文字(§ 2.3)是类 String 的实例的引用。

## 2.4.8 对象上的操作符

对象上的操作符包括域访问、方法调用、类型转换、字符串连接、相等性比较、instanceof、以及条件操作符? :。

## 2.5 变量

变量(variable)是一种存储位置。它有一个相应的类型，有时称为它的编译期类型(compile-time-type)。这种类型或者是基本类型(§ 2.4.1)，或者是引用类型(§ 2.4.5)。变量总是包含一个与它的类型赋值相容(§ 2.6.6)的值。基本类型的变量总是持有那个确定的基本类型的值。引用类型的变量可以持有一个 null 引用或者一个与该变量的类型赋值相容的(§ 2.6.6)类的任何对象的引用。

变量的类型与它的值的相容性通过 Java 语言的设计来保证，因为缺省的值(§ 2.5.1)是相容的，并且所有对变量的赋值，为了赋值相容性，都在编译期被检查。

有七种变量：

- (1) 类变量(class variable)是在类声明中用关键字 static(§ 2.9.1),或是在接口声明中用(或不用)关键字 static 声明的类类型的域。类变量在类或者接口被装载(§ 2.16.2)时创建,并且在创建时按照缺省值被初始化。在类的任何必需的终止函数(§ 2.16.8)结束后、类或者接口被卸载(§ 2.16.8)时,类变量有效地终止存在。
- (2) 实例变量(instance variable)是类声明中不使用关键字 static(§ 2.9.1)声明的域。如果类 T 具有实例变量 a,则对类 T 或者它的任何子类的每个新创建的对象,都创建并按照缺省值初始化一个新的实例变量 a 作为每个新创建的对象的一部分。当对象不再被引用,在该对象的任何必需的终止函数结束之后,该对象的域实例变量有效地终止存在。
- (3) 数组成分(array components)是在一个数组新对象被创建时创建的、并且初始化为缺省值(§ 2.5.1)的未命名变量。数组成分在数组不再被引用时有效地终止存在。
- (4) 方法参数(method parameters)指定传递给方法的参数值。对在方法声明中声明的每个参数,每次方法被调用时都创建一个新的参数变量。新的变量用来自方法调用的相应参数值初始化。在方法主体执行结束时方法参数有效地终止存在。
- (5) 构造函数参数(constructor parameters)指定传递给构造函数的参数值。对在构造函数声明中声明的每个参数,每次类实例创建表达式或者显式的构造函数调用被求值时,都 创建一个新的参数变量。新的变量用来自创建表达式或者构造函数调用的相应参数值初始化。当构造函数主体执行结束时,构造函数参数有效地终止存在。
- (6) 异常处理者参数(exception-handler parameter)变量在每次异常被 try 语句(§ 2.15.2)的 catch 子句捕捉时创建。新的变量用异常(§ 2.15.3)相应的实际的对象初始化。当 catch 子句(§ 2.15.2)相应的块执行结束时,异常处理者参数有效地终止存在。
- (7) 局部变量(local variables)由局部变量声明语句声明。每当控制流进入一个块或者 for 语句中时,就为在直接包含在该块或者 for 语句中的局部变量声明语句中声明的每个局部变量创建一个新的变量。但是,局部变量直到声明它的局部变量声明语句被执行时才初始化。当该块或者 for 语句执行结束时,局部变量有效地终止存在。

## 2.5.1 变量的初始值

Java 程序中的每个变量在使用前必须有一个值:

- 每个类变量、实例变量和数组成分在创建时用缺省值(default value)初始化:
  - 对类型 byte, 缺省值是零, 即(byte)0 的值。
  - 对类型 short, 缺省值是零, 即(short)0 的值。
  - 对类型 int, 缺省值是零, 即 0。
  - 对类型 long, 缺省值是零, 即, 0L。
  - 对类型 float, 缺省值是正零, 即, 0.0f。
  - 对类型 double, 缺省值是正零, 即 0.0d。
  - 对类型 char, 缺省值是 null 字符, 即 ‘\u0000’。
  - 对类型 boolean, 缺省值是 false。
  - 对所有的引用类型, 缺省值是 null(§ 2.3)。

- 每个方法参数(§ 2.5), 初始化为方法调用者提供的相应参数值。
- 每个构造函数参数(§ 2.5), 初始化为对象创建表达式或者显式构造函数调用提供的相应的参数值。
- 异常处理器参数(§ 2.15.2), 初始化为代表该异常的被抛出的对象。
- 局部变量在使用前必须通过初始化或者赋值显式地给一个值。

### 2.5.2 变量具有类型, 对象具有类

每个对象属于某个特定的类。这个特定的类,是产生该对象的类实例创建表达式中提及的类,或是使用其类对象调用 `newInstance` 方法以产生该对象的那个类。这个类称为该对象的类。一个对象被称为它的类以及它的类的所有超类的一个实例。有时对象的类称为该对象的“运行期类型”,但是,“类”是更准确的术语。

(有时一个变量或者表达式被称为具有一个“运行期类型”,但是那是对术语的滥用;它是指该变量或者表达式在运行期的值(假定这个值不是 `null`)所引用的对象的类。恰当地讲,类型是一个编译期概念。变量或者表达式有类型,对象或者数组没有类型,但属于一个类。)

变量的类型总是被声明的,表达式的类型可以在编译器推断。类型限制在运行期变量可以持有的、或者表达式可以产生的、可能的值。如果一个运行期值是一个非 `null` 引用,则它引用一个具有类(不是类型)的对象或者数组,并且那个类必须与编译期类型相容。

尽管变量或者表达式可以具有接口类型的编译期类型,但是没有接口(§ 2.13)的实例。接口类型的变量或者表达式可以引用任何实现该接口的类的对象。

每个数组也有一个类。数组的类具有奇怪的名称,它们不是有效的 Java 标识符,例如,具有 `int` 成分的数组的类名称为 “[I”。

## 2.6 转换和提升

转换(conversions)隐式地把一个表达式的类型(有时是值)改变成该表达式的周围上下文可接受的类型。有时这将需要运行期的一个相应的动作,以检查转换的有效性,或者把表达式的运行期值转换成适合新类型的形式。

数值提升(numeric promotions)是把数值操作的一个操作数改变到更宽的类型,或者把数值操作的两个操作数改变成相同的类型,以使操作可以进行。

在 Java 中,有六大类转换:

- 等同转换
- 放宽基本转换
- 缩窄基本转换
- 放宽引用转换
- 缩窄引用转换
- 字符串转换

有五种转换上下文(conversion context),转换表达式出现在这些转换上下文中。每种上下文允许上面讲的转换中的某一类转换,而不允许其他转换。这些转换上下文是:

- 赋值转换(§ 2.6.6):它把一个表达式的类型转换成一个指定的变量的类型。允许的赋

值转换限于不会引起异常的赋值转换。

- 方法调用转换(§ 2.6.7):它作用于方法或者构造函数调用中的每个参数。并且,除了一种情况,它进行与赋值转换相同的转换。方法调用转换不会引起异常。
- 类型转换(§ 2.6.8):它把一个表达式的类型转换成用一个类型转换操作符显式地指定的类型。它比赋值转换和方法调用转换包含得多,允许除字符串转换之外的任何特定的转换。但是某些引用类型的转换可能在运行期引起一个异常。
- 字符串转换:它允许把任何类型转换成 String 类型(§ 2.4.7)。
- 数值提升:它把数值操作符的操作数变成同一个类型以使操作可以进行。

字符串转换(string conversion)只在双目操作符+的一个参数是 String 时才作用于该操作符的操作数;不再进一步介绍这种转换。

## 2.6.1 等同转换

对任何类型都允许把该类型转换成同一类型。

## 2.6.2 放宽基本转换

以下基本类型转换称为放宽基本转换(widening primitive conversions):

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

放宽转换不丢失数值符号或者数量的数量级的信息。从一个整型放宽到另一个整型和从一个 float 放宽到一个 double 的转换不会丢失任何信息;数值被确切地保留下。从 int 或者 long 值到 float,或者 long 值到 double 的转换可能会丢失精度,即结果可能会丢失值的最低的某些位。得到的浮点值是用 IEEE 754 最接近舍入模式(§ 2.4.3)把整型值正确地舍入的结果。

按照这个规则,从有符号整数值到整型的放宽转换,只是用符号扩展该整型值的二进制补码表示以填充放宽的格式。从 char 类型的值到整型的放宽转换用零扩展该字符值的表示以填充放宽的格式。

尽管可能会丢失精度,但是基本类型之间的放宽转换不会引起运行期异常(§ 2.15)。

## 2.6.3 缩窄基本转换

以下基本类型转换称为缩窄基本转换(narrowing primitive conversions):

- byte 到 char
- short 到 byte 或 char
- char 到 byte 或 short
- int 到 byte、short 或 char
- long 到 byte、short、char 或 int
- float 到 byte、short、char、int 或 long

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

缩窄转换可能会丢失数值的符号或者数量的数量级的信息,或者两种信息都会丢失(例如,把整型值32763缩窄为 byte 类型产生值-5)。缩窄转换也可能丢失精度。

从有符号整数值到整型的缩窄转换只是抛弃除了最低 n 位的所有位,这里 n 是用于表示该类型的位的个数。这可能导致结果值与输入值的符号不同。

从字符到整型的缩窄转换类似地只是抛弃除了最低 n 位的所有位,这里 n 是用于表示该类型的位的个数。这可能导致结果值是负数,尽管字符表示16位无符号整数值。

在从浮点数到整型的缩窄转换中,如果浮点数是 NaN,则转换的结果是适当类型的0。如果浮点数太大,不能由该整型表示,或者浮点数是正无穷,则结果是该整型所能表示的最大值。如果浮点数太小,不能由该整型表示,或者浮点数是负无穷,则结果是该整型所能表示的最小值。否则,结果是用 IEEE 754向零舍入模式(§ 2.4.3)把浮点数向零舍入得到的整数值。

从 double 到 float 的缩窄转换按照 IEEE 754进行。结果用 IEEE 754的最接近舍入模式(§ 2.4.3)正确地舍入。太小而不能用 float 表示的值转换成正或者负零;太大而不能用 float 表示的值转换成正或者负无穷;double 类型的 NaN 总是转换成 float 类型的 NaN。

尽管可能发生溢出、下溢或者精度丢失,基本类型间的缩窄转换从不引起运行期异常。

#### 2.6.4 放宽引用转换

放宽引用转换(widening reference conversions)从不在运行期要求特殊的动作,因此不会在运行期抛出异常。由于它们不影响 Java 虚拟机,所以不再考虑它们。

#### 2.6.5 缩窄引用转换

以下的允许转换称为缩窄引用转换(narrowing reference conversions):

- 任何类类型 S 到任何类类型 T,只要 S 是 T 的超类。(一个重要的特例是从类类型 Object 到任何其他类类型都有一个缩窄转换。)
- 任何类类型 S 到任何接口类型 K,只要 S 不是 final 并且不实现 K。(一个重要的特例是从类类型 object 到任何接口类型都有一个缩窄转换。)
- 从类型 object 到任何数组类型。
- 从类型 object 到任何接口类型。
- 从任何接口类型 J 到任何非 final 的类类型 T。
- 从任何接口类型 J 到任何是 final 的类类型 T,只要 T 不实现 J。
- 从任何接口类型 J 到任何接口类型 K,只要 J 不是 K 的子接口,并且 J 和 K 没有用相同的签名但不同的返回类型声明方法。
- 从任何数组类型 SC[] 到任何数组类型 TC[],只要 SC 和 TC 是引用类型并且存在一个允许的从 SC 到 TC 的缩窄转换。

这些转换要求在运行期检查实际的引用值是否是新类型的合法值。如果不是,则 Java 虚拟机抛出一个 ClassCastException。

#### 2.6.6 赋值转换

赋值转换(assignment conversion)发生在把表达式的值赋给变量的时候:表达式的类型

必须转换成变量的类型。赋值上下文允许等同转换(§ 2.6.1),放宽基本转换(§ 2.6.2),或者放宽引用转换(§ 2.6.4)。另外,如果满足以下全部条件,也可以使用缩窄基本转换(§ 2.6.3):

- 表达式是 int 类型的常数表达式。
- 变量类型是 byte、short 或 char。
- 表达式的值可以用变量的类型表示。

如果一个表达式的类型可以用赋值转换转换成一个变量的类型,我们就说表达式(或者它的值)可以赋给(assignable)该变量;或者等价地,表达式的类型与变量的类型是赋值相容的(assignment compatible)。

赋值转换从不引起异常。基本类型的值不能赋给引用类型的变量,引用类型的值不能赋给基本类型的变量。boolean 类型的值只能赋给 boolean 类型的变量, null 类型的值可以赋给任何引用类型。

允许把编译期引用类型 S(源)的值赋给编译期引用类型 T(目的)的变量:

- 如果 S 是类类型:
  - 如果 T 是类类型,则 S 必须是 T 的同类或者子类。
  - 如果 T 是接口类型,则 S 必须实现接口 T。
- 如果 S 是接口类型:
  - 如果 T 是类类型,则 T 必须是 Object。
  - 如果 T 是接口类型,则 T 必须是 S 的相同接口或者超接口。
- 如果 S 是数组类型 SC[], 即一组类型为 SC 的成分:
  - 如果 T 是类类型,则 T 必须是 Object。
  - 如果 T 是接口类型,则 T 必须是 Cloneable。
  - 如果 T 是数组类型,即类型 TC[], 一组类型为 TC 的成分,则或者
    - TC 与 SC 必须是相同的基本类型,或者
    - TC 与 SC 都是引用类型并且类型 SC 可以赋给 TC。

## 2.6.7 方法调用转换

方法调用转换(method invocation conversion)作用于方法或者构造函数调用中的每个参数:参数表达式的类型必须转换成相应的参数的类型。方法调用上下文允许使用等同转换(§ 2.6.1)、放宽基本转换(§ 2.6.2)或者放宽引用转换(§ 2.6.4)。特别地,方法调用转换不包括赋值转换(§ 2.6.6)中的隐式缩窄整数常数部分(§ 2.6.6)。

## 2.6.8 类型转换

类型转换(casting conversions)比赋值转换或者方法调用转换更加强大,它作用于类型转换操作符的操作数,操作数表达式的值必须转换成由类型转换操作符显式地指定的类型。类型转换上下文允许使用等同转换(§ 2.6.1)、放宽基本转换(§ 2.6.2)、缩窄基本转换(§ 2.6.3)、放宽引用转换(§ 2.6.4)或者缩窄引用转换(§ 2.6.5)。因此,类型转换比赋值转换或者方法调用转换包括得更多:类型转换可以做除了字符串转换之外的任何允许转换。

类型转换可以把任何数值类型的值转换成任何其他数值类型。boolean 类型的值不能转换成其他类型。引用类型的值不能转换成基本类型的值。

有时类型转换可以在编译期被证明是错误的并且导致编译期错误。否则，或者类型转换可以在编译期被证明是正确的，或者需要一个运行期有效性检查（细节参见《Java语言规范》一书）。如果运行期的值是一个 null 引用，则允许类型转换。如果运行期检查失败，则 ClassCastException 被抛出。

## 2.6.9 数值提升

数值提升作用于算术操作符的操作数。数值提升上下文允许使用等同转换（§ 2.6.1）或者放宽基本转换（§ 2.6.2）。

数值提升用于把数值操作符的操作数转换成相同的类型，在这个相同的类型下操作可以进行。有两种数值提升，它们是单目数值提升（unary numeric promotion）和双目数值提升（binary numeric promotion）。C 中类似的转换叫作“通常单目转换”和“通常双目转换”。数值提升不是 Java 的一般特性，而是内置操作符的特定定义的一个属性。

对数值类型的单个操作数进行单目数值提升的操作符把类型为 byte、short 或者 char 的操作数转换成 int，否则就不对操作数操作。移位操作符的操作数独立地用单目数值提升进行提升。

当操作符对一对数值操作数进行双目数值提升时，顺序使用下述规则，在需要时用放宽转换（§ 2.6.2）转换操作数：

- 如果有一个操作数是 double 类型的，则把另一个转换成 double。
- 否则，如果有任何一个操作数是 float 类型的，则把另一个转换成 float。
- 否则，如果有任何一个操作数是 long 类型的，则把另一个转换成 long。
- 否则，两个操作数都转换成 int 类型。

## 2.7 名称和包

### 2.7.1 名称

名称（names）用于引用 Java 程序中声明的实体。声明的实体是包、类型、类型的成员（域或者方法）、参数或者局部变量。

简单（simple）名称是一种单个的标识符（§ 2.2），限定（qualified）名称提供对包和引用类型中的成员的访问。限定名称（§ 2.7.8）由一个名称，一个“.”符号和一个标识符组成。

并非 Java 程序中的所有标识符都是名称的一部分。标识符也用于声明中，在声明中标识符决定一个实体赖以被识别的名称。标识符还用于域访问表达式和方法调用表达式，以及语句行号和使用语句行号的 break 与 continue 语句中。

### 2.7.2 包

Java 程序是组织起来的包（packages）的集合。包由一些编译单元组成，并且具有一个层次性的名称。包是独立地开发的，并且每个包有它自己的名称集，这有助于防止名称冲突。

每个 Java 主机决定如何创建和存储包、编译单元和子包，并决定哪些顶层包名称在特定的编译中是可见的，以及决定哪些包是可访问的。包可以存储在本地文件系统中，分布式文件系统中，或者某种形式的数据库中。

包的名称组件或者类名可能包含某个字符,这个字符不能合法地出现在主机文件系统的普通目录或者文件名中。例如,一个 Unicode 字符在只允许文件名为 ASCII 字符的系统上。

Java 系统必须支持至少一个未命名包。它可以支持多于一个的未命名包,但并不要求这样做。在每个未命名包中有哪些编译单元由主机系统决定。在开发小的或者临时的应用程序或者刚刚开始开发时,为了方便,未命名包原则上由 Java 提供。

`import` 声明允许在另一个包中声明的类型可以以该类型的简单名称而不是完整限定名称被识别(§ 2.7.9)。导入声明只影响单个编译单元中的类型声明。编译单元自动导入在预定义的包 `java.lang` 中声明的每个 `public` 类型名称。

### 2.7.3 成员

包和引用类型具有成员(members)。包(§ 2.7.2)的成员是子包和在该包的所有编译单元中声明的所有的类(§ 2.8)与接口(§ 2.13)类型。引用类型的成员是域(§ 2.9)和方法(§ 2.10)。

### 2.7.4 包成员

包的成员是它的子包(subpackages)和在该包的编译单元中声明的类型。

总的来说,包的子包由主机系统决定。但是,标准包“`java`”总是具有子包 `lang`, `util`, `io` 和 `net`。同一个包中不能有两个不同的成员具有相同的简单名称(§ 2.7.1),但是不同的包的成员可以具有相同的简单名称。

### 2.7.5 类类型的成员

类类型(§ 2.8)的成员是域(§ 2.9)和方法(§ 2.10)。这些包括从类的直接超类(§ 2.8.3)继承来的成员(如果它有一个直接超类),和从任何直接超接口(§ 2.13.2)继承来的成员,以及在类主体内声明的任何成员。对类类型的域和方法具有相同的简单名称没有限制。

类类型可以具有两个或者更多具有相同简单名称的方法,只要这些方法的参数个数不同或者至少有一个相同位置上的参数类型不同。这样的方法成员名称称为被重载了(overloaded)。一个类类型可以包含与它的超类或者超接口中的名称和签名相同的方法的声明,这种情况下超类或者超接口中的这个方法不被继承,否则就从超类或者超接口中继承。如果不被继承的方法是 `abstract`,我们说新的声明将实现(implement)该方法;如果该方法不是 `abstract`;我们说新的声明将覆盖它。(override)。

### 3.7.6 接口类型的成员

接口类型(§ 2.13)的成员是域和方法。接口的成员是从任何直接超接口(§ 2.13.2)继承的成员和接口主体中声明的成员。

### 2.7.7 数组类型的成员

数组类型(§ 2.14)的成员是从它的超类,类 `object`(§ 2.4.6)继承的成员和域 `length`。域 `length` 是每个数组的常数(final)域。

## 2.7.8 限定名称和访问控制

限定名称(§ 2.7.1)是访问包和引用类型中的成员的方式。相关的访问方式包括域访问表达式和方法调用表达式。这三者在语法上的相似之处在于都有“.”符号，“.”前面是包类型或者具有某个类型的表达式的指示；“.”的后面是包的成员或者类型的名称的标识符。这些统称为限定访问(qualified access)的构造。

Java 提供限制限定访问的机制以防止包或者类的用户不必要地依赖于该包或者类的实现中的细节。访问控制也用于构造函数。

一个包是否可以被访问由主机系统决定。

一个类或者接口可以声明为 public，这样它就可以被能够访问声明它的包的任何 Java 代码用限定名称访问。没有声明为 public 的类和接口可以也只能从声明它的包内访问。

接口的每个域或者方法都必须是 public，public 接口的每个成员都隐式地是 public，而不论关键字 public 是否在它的声明中出现。如果接口不是 public，则它的每个域和方法都必须显式地声明为 public。由此可以推断，当且仅当接口本身是可以访问的时候，接口的成员是可以访问的。

类的域、方法，或者构造函数可以用 public、private 或者 protected 关键字中的至多一个来声明。public 成员可以被任何 Java 代码访问。private 成员只能从包含它的声明的类内访问。没有声明为 public、protected，或者 private 的成员称为具有缺省访问权(default access)，并且也可以也只能从声明它的包内访问。

对象的 protected 成员只能被负责实现该对象的代码访问。准确地说，protected 成员可以从声明它的包内访问。另外，只要服从特定的限制，它可以从包含它的声明的类类型的子类声明中访问。

## 2.7.9 完整限定名称

每个包、类、接口、数组类型和基本类型都具有一个完整限定名称。由此可以推断，除了 null 类型，每个类型都具有一个完整限定名称。

- 一个基本类型的完整限定名称是该基本类型的关键字，即 boolean、char、byte、short、int、long、float 或 double。
- 一个命名的包如果不是另一命名的包的子包，则它的完整限定名称是它的简单名称。
- 一个命名的包的命名的子包的完整限定名称由包含包的完整限定名称后面加上“.”再加上该子包的简单(成员)名称组成。
- 在未命名的包中声明的类或者接口的完整限定名称是该类或者接口的简单名称。
- 在命名的包中声明的类或者接口的完整限定名称由该包的完整限定名称后面加上“.”再加上类或者接口的简单名称组成。
- 数组类型的完整限定名称由该数组类型的成分类型的完整限定名称后面加上“[]”组成。

## 2.8 类

一个类声明(class declaration)指定一个新的引用类型并提供它的实现。每个类都作为一个已存在的单个类的继承或者子类来实现。一个类也可以实现一个或者多个接口。

类的主体声明成员(域和方法),静态初始化函数和构造函数。

### 2.8.1 类名称

如果一个类是在一个完整限定名称为 P 的命名的包中声明的,则该类的完整限定名称是 P. Identifier。如果类在一个未命名的包中声明,则类的完整限定名称是 Identifier。

如果两个类被同一个类装载器装载(§ 2.16.2)并且具有相同的完整限定名称(§ 2.7.9),则这两个类是同类(same class),从而是同类型(same type)。

### 2.8.2 类修饰符

类的声明可以包括类修饰符(class modifiers)。类可以声明为 § 2.7.8 中讨论的 public。

一个 abstract 类是一个不完整的类,或者被认为是不完整的。只有 abstract 类才可以有 abstract 方法(§ 2.10.3)。abstract 方法是已声明但尚未实现的方法。

类可以声明为 final,如果它的定义是完整的并且不需要子类。因为一个 final 类不会有任何子类,所以 final 类的方法不能在子类中被覆盖。一个类不能既是 final 又是 abstract,因为这样的类的实现永远不能完成。

类被声明为 public 以便声明它的包以外的包可以获得其类型。使用 public 类的完整限定名称,或者由 import 声明(§ 2.7.2)创建的较短的名称,可以从别的包中访问该 public 类,只要主机允许访问该 public 类的包。如果一个类没有 public 修饰符,则对该类声明的访问限于声明它的包中。

### 2.8.3 超类和子类

类声明中可选的 extends 子句指定当前类的直接超类(direct superclass)。当前类的实现从其直接超类的实现继承。一个类被称为它 extends 的那个类的直接子类(direct subclass)。只有类 object(§ 2.4.6)没有直接超类。如果在类声明中省略 extends 子句,则新类的超类是 object。

子类(subclass)关系是直接子类关系的一个真包含。如果类 A 是类 C 的直接子类,或者 C 有一个直接子类 B 并且 A 是 B 的直接子类,则 A 是 C 的子类。当类 C 是类 A 的子类时称 A 是 C 的超类(superclass)。

### 2.8.4 类成员

类类型包括以下的成员:

- 从它的直接超类(§ 2.8.3)继承的成员。没有直接超类的类 Object 除外。
- 从任何直接超接口(§ 2.13.2)继承的成员。
- 在类的主体中声明的成员。

超类中声明为 `private` 的成员不被该类的子类继承。类中没有用 `private`、`protected`，或者 `public` 声明的成员不被在声明该类的包之外的包中声明的子类继承。构造函数（§ 2.12）和静态初始化函数（§ 2.11）不是成员，因此不被继承。

## 2.9 域

类类型中的变量是它的域(fields)。类(static)变量对每个类存在一次，实例变量对类的每个实例存在一次。域可以包括初始化器，并且可以用不同的修饰符关键字修饰。

如果类用一个特定的名称声明一个域，则称这个域的声明隐藏(hide)了该类的超类和超接口中所有可以访问的、具有相同名称的域的声明。一个类继承它的直接超类和直接超接口中的、该类的代码可以访问的、并且没有被该类的声明所隐藏的、直接超类和直接超接口的所有域。被隐藏的域可以通过使用限定名称(如果它是 static)或者通过使用一个包含转换成超类的类型转换的、或者关键字 `super` 的、访问表达式来访问。

### 2.9.1 域修饰符

如 § 2.7.8 中所讨论的，域可以声明为 `public`、`protected` 或者 `private`。

如果一个域被声明为 `static`，则不论最终创建了该类的多少个实例(可能是零个)，都确切地存在该域的一个实例。`static` 域，有时称为类变量(class variables)，在初始化类时创建(§ 2.16.4)。

没被声明为 `static` 的域称为实例变量(instance variables)。当类的一个新的实例被创建时，对在该类或者其任何超类中声明的每个实例变量，都伴随该实例创建一个新的变量。

域可以声明为 `final`，这种情况下它的声明必须包括一个变量初始化器(§ 2.9.2)。类和实例变量(static 和非 static 域)都可以声明为 `final`。一旦一个 `final` 域被初始化，它将一直包含同一个值。如果一个 `final` 域持有对一个对象的引用，则该对象的状态可以通过对象上的操作改变，但是这个域将总是引用同一个对象。

变量可以被标记为 `transient` 以指示它们不是对象的永久状态。`transient` 属性可用于 Java 实现以支持特殊的系统服务。《Java 语言规范》一书中尚未规范这些服务的细节。

Java 语言允许访问共享变量的线程保留该变量的私有工作拷贝；这允许了对多线程(§ 2.17)的更有效的实现。这些工作拷贝只在规定的同步点上需要与共享的主存储器中的主拷贝一致，例如当对象被锁定或者解锁(§ 2.17)时。作为一个规则，为了保证共享的变量被一致地、可靠地更新，线程应当通过获得一个锁来保证它对这些变量的访问是排他性的。这个锁保守地增强了这些共享的变量相互间的排他性。

Java 提供了使某些用途更加方便的第二个机制：域可以声明为 `volatile`。在这种情况下线程每次访问该变量时都必须使它对该域的工作拷贝与主拷贝一致。而且，代表一个线程的一个或者多个易变变量的主拷贝上的操作由主存储器确切地按照线程要求的顺序进行。一个 `final` 域不能再被声明为 `volatile`。

## 2.9.2 域的初始化

如果一个域的声明包含一个变量初始化器,则该声明具有一个对声明的变量赋值的符号,并且:

- 如果是对类变量(即 static 域)的声明,则该变量的初始化器在类初始化(§ 2.16.4)时被求值,并且确切地进行一次赋值。
- 如果是对实例变量(即,非 static 域)的声明,则在每次创建该类的实例时变量初始化器被求值并且进行赋值操作。

## 2.10 方法

方法(method)声明可被调用的可执行代码,传递固定个数的值作为参数。每个方法声明都属于某个类。类从它的直接超类(§ 2.8.3)和任何直接超接口(§ 2.13.2)继承超类和超接口的所有可访问的方法。只有一个例外:如果在新类中一个名称被声明为一个方法,则具有相同签名(2.10.2)的方法不被继承;相反,新声明的方法被称为覆盖 override)任何这样的方法的声明。覆盖方法必须不与它所覆盖的定义冲突。例如,具有一个不同的返回类型。超类中被覆盖的方法可以用包含 super 关键字的方法调用表达式访问。

### 2.10.1 形式参数

方法的形式参数(formal parameters)(如果有)由一列逗号隔开的参数区分符指定。每个参数区分符由一个类型和一个指定该参数名称的标识符组成。当方法被调用时,在执行函数主体之前,用实际参数表达式的值初始化每个声明的类型新创建的参数变量(§ 2.5)。

### 2.10.2 签名

方法的签名(signature)由方法的名称和方法的形式参数(§ 2.10.1)的个数与类型组成。一个类不能声明两个相同签名的方法。

### 2.10.3 方法修饰符

访问修饰符 public、protected 和 private 在(§ 2.7.8)中讨论。

一个 abstract 方法声明介绍一个成员方法,提供它的签名(§ 2.10.2),返回类型以及 throws 子句(如果有),但是不提供其实现。abstract 方法 m 的声明必须出现在一个 abstract 类(叫作 A)内。A 的每个非 abstract 子类必须提供 m 的一个实现。一个声明为 abstract 的方法不能再声明为 private、static、final、native 或者 synchronized。

声明为 static 的方法称为类方法(class method)。类方法被调用时总是不引用某个特定的对象。类方法只能通过简单名称引用该类的类方法和类(static)变量。

没被声明为 static 的方法是实例方法(instance method)。实例方法被调用时总是关于一个对象,这个对象成为当前对象,当前对象在执行该方法的主体时由关键字 this 和 super 引用。

方法可以声明为 final 以防止子类覆盖或者隐藏它。private 方法和所有在 final 类(§ 2.8.

2)中声明的方法都隐式地是 final,因为不可能覆盖它们。如果一个方法是 final 或者隐式地是 final,则编译器或者运行期代码生成器可以安全地“直接插入”final 方法主体,用方法主体内的代码代替对方法的调用。

一个 synchronized 方法在它执行前得到一个监视器锁(§ 2.17)。对于类(static)方法,使用与该方法的类的类对象相关联的锁。对于实例方法,使用与 this(为之调用该方法的对象)相关联的锁。synchronized 语句对各对象使用同样的锁。

方法可以声明为 native 以指示它是用依赖于平台的代码实现的,通常用另一种程序设计语言例如 C、C++ 或者汇编语言编写。

## 2.11 静态初始化函数

类中声明的任何静态初始化函数(static initializer)在类被初始化时执行(§ 2.16.4)。而且一起执行类变量的任何域初始化器(§ 2.9.2),用于初始化该类的类变量(§ 2.16.4)。

静态初始化函数与类变量初始化器按照文本顺序执行。它们不能引用那些其声明文本出现在使用之后的、在类中声明的类变量,即使这些类变量是可见的。设计这个限制是为了在编译期捕捉绝大多数循环的或者其他损坏的初始化。

## 2.12 构造函数

构造函数(constructor)用于类实例对象的创建。构造函数的声明看上去像是一个没有结果类型的方法声明。构造函数由以下部分调用:类实例创建表达式(§ 2.16.6)、由字符串连接操作符+引起的转换与连接以及从别的构造函数的显式构造函数调用。它们从不被方法调用表达式调用。

对构造函数的访问和继承由访问修饰符 public、protected 和 private(§ 2.7.8)控制。构造函数声明不是成员,它们从不被继承,因此不会被隐藏或者覆盖。

如果一个构造函数主体不以一个显式的构造函数调用开始,并且声明的构造函数不是最初的类 Object 的一部分,则该构造函数的主体被编译器隐式地假定以超类构造函数调用“super();”开始。“super();”是对该类的直接超类的不带参数的构造函数的调用。

如果一个类没有声明构造函数,则自动提供一个不带参数的缺省构造函数(default constructor)。如果声明的类是 Object,则缺省构造函数的主体是空的。否则,缺省构造函数不带参数,并且只是调用超类的不带参数的构造函数。如果类声明为 public,则隐式地给予缺省构造函数访问修饰符 public。否则,缺省构造函数具有无访问修饰符(§ 2.7.8)的隐含的缺省访问权。

通过声明至少一个构造函数以防止创建缺省构造函数,并且把所有的构造函数声明为 private,可以把类设计成防止该类的声明之外的代码创建该类的实例。

## 2.13 接口

接口(interface)是一个其成员为常数和 abstract 方法的引用类型。这种类型没有实现,但

是其他的无关类可以通过提供它的 abstract 方法的实现来实现它。Java 程序可以使用接口以使得相关类不必共享同一个 abstract 超类，或者向 Object 增加方法。

一个接口可以声明为一个或者多个其他接口的直接继承(direct extension)。这表示除了它可能隐藏的任何常数和增加它自己的新声明的成员，它隐式地指定它所继承的接口的全部 abstract 方法和常数。

一个类可以声明为直接实现(directly implement)一个或者多个接口。这表示该类的任何实例实现该接口指定的全部 abstract 方法。一个类必然地实现它的直接超类和直接超接口实现的全部接口。这种(多重)接口继承允许对象不需共享任何实现即可支持(多重)共同行为。

一个其声明类型为接口类型的变量可以以一个对象的引用作为它的值，这个对象是为实现这个特定的接口而声明的任何一个类的一个实例。一个类碰巧实现了接口的全部 abstract 方法还不够；类或者它的一个超类必须实际地声明为实现该接口，否则该类不被认为实现了该接口。

### 2.13.1 接口修饰符

接口声明的前面可以是接口修饰符 public 和 abstract。访问修饰符 public 在(§ 2.7.8)中讨论。每个接口都隐式地是 abstract。接口的所有成员都隐式地是 public。

接口不能是 final，因为这样的类的实现永远不能完成。

### 2.13.2 超接口

如果提供一个 extends 子句，则声明的接口从每个别的命名的接口继承，从而继承每个别的命名的接口的方法和常数。任何 implements 声明的接口的类也被认为实现该接口继承的，这个类可以访问的全部接口。

类声明中的 implements 子句列出被声明的类的直接超接口(direct super interfaces)的名称。当前包中的全部接口都是可访问的。可以访问别的包中的接口，如果主机允许访问这个包，并且接口声明为 public。

如果 K 是 C 的直接超接口，或者 C 有一个以 K 为超接口的直接超接口 J，或者 K 是 C 的超类的一个超接口，则接口类型 K 是类类型 C 的超接口(superinterface)。我们说类实现(implement)它的全部超接口。

没有与类 Object 相似的接口，即，虽然每个类都是类 Object 的一个继承，但是没有一个所有接口都是其继承的单个接口。

### 2.13.3 接口成员

接口的成员是从直接超接口继承的成员和在接口中声明的成员。接口从它所继承的接口继承那些接口的除去与它声明的域同名的域以外的全部成员。

接口成员或者是域，或者是方法。

### 2.13.4 接口(常数)域

在接口主体中声明的每个域都隐式地是 static 和 final。接口没有实例变量。接口中每个域声明都隐式地是 public。接口中的常数声明不能包括修饰符 transient 或者 volatile。

接口主体中的每个域都必须有一个初始化表达式,该表达式不必是常数表达式。在接口初始化(§ 2.16.4)时,确切地进行一次对变量初始化器的求值和赋值。

### 2.13.5 接口(抽象)方法

接口主体中的每个方法声明都隐式地是 abstract。

接口主体中的每个方法声明都隐式地是 public。

接口中声明的方法不能声明为 static,因为在 Java 中 static 方法不能是 abstract。接口主体中声明的方法不能声明为 native 或者 synchronized。因为这些关键字描述实现属性而不是接口属性。但是,接口中声明的方法可以由实现该接口的类中声明为 native 或者 synchronized 的方法实现。接口主体中声明的方法不能声明为 final,但是,它可以由实现该接口的类中声明为 final 的方法实现。

### 2.13.6 接口中的覆盖、继承和重载

如果接口声明一个方法,则该方法声明被称为覆盖(override)该接口的超接口中所有具有相同签名的方法。否则,超接口中的方法可以被该接口中的代码访问。

一个接口从其直接超接口继承超接口中所有没有被该接口的声明覆盖的方法。

如果接口的两个方法(不论两个方法是在同一个接口中声明的,还是都是从一个接口继承的,还是一个是在接口中声明,另一个是继承的)具有相同的名称和不同的签名,则称该方法的名称被重载了(overloaded)。

## 2.14 数组

Java 数组(array)是对象,被动态地创建,并且可以赋给 Object 类型的变量(§ 2.4.6)。类 Object 的所有方法都可以在数组上调用。

一个数组对象包含一些变量。变量的个数可以是零,这种情况下称数组是空的(empty)。数组中包含的变量没有名称,通过使用非负整数索引值的数组访问表达式引用它们。这些变量称为数组的成分(components)。如果一个数组有 n 个成分,则称 n 是该数组的长度(length)。

有零个成分的数组与 null 引用(§ 2.4)不同。

### 2.14.1 数组类型

一个数组的所有成分具有相同的类型,称为该数组的成分类型(component type)。如果一个数组的成分类型是 T,则数组自身的类型写作 T[]。

数组成分的类型自身可以是数组类型。这样的数组的成分可以包含子数组的引用。如果从任一个数组类型开始,考虑它的成分类型,然后(如果它也是数组类型)考虑这个类型的成分的类型,等等,最终一定会得到一个非数组类型的成分类型。这个类型称为原始数组的元素类型(element type)。该数据结构在这个层次上的成分,称为原始数组的元素(elements)。

在一种情况下数组的元素可能是数组:如果元素的类型是 Object(§ 2.4.6),则一些或者全部元素可以是数组,因为每个数组对象都可以赋给 Object 类型的变量。

在 Java 中,与 C 不同,char 数组不是 String(§ 2.4.6),并且 String 与 char 数组都不以'\

u0000' (NUL 字符) 结束。Java 的 String 对象是不可变的 (它的值从不改变), 而 char 数组具有可变的元素。

数组的元素类型可以是基本的或者引用的任何类型。特别地, 支持数组以接口类型作为成分类型; 这样的数组的元素可以以 null 引用或者实现该接口的任何类类型的实例作为它们的值。支持以 abstract 类类型作为其成分类型的数组。这样的数组的元素可以以 null 引用或者该 abstract 类的任何非 abstract 子类作为它们的值。

## 2.14.2 数组变量

数组类型的变量持有一个对象的引用。声明一个数组类型的变量并不创建一个数组对象或者为数组成分分配任何空间。它只创建变量自身, 这个变量可以包含一个数组的引用。

因为数组的长度不是它的类型的一部分, 所以数组类型的单个变量可以包含不同长度的数组的引用。一旦创建一个数组对象, 它的长度就不再改变。为了使一个数组变量引用一个不同长度的数组, 必须把不同的数组的引用赋给这个变量。

如果一个数组变量 v 具有类型 A[], 这里 A 是一个引用类型, 则 v 可以持有任何数组类型 B[] 的一个引用, 只要 B 可以赋给 A (§ 2.6.6)。

## 2.14.3 数组创建

数组通过数组创建表达式 (array creation expression) 或者数组初始化器 (array initializer) 创建。

## 2.14.4 数组访问

使用数组访问表达式 (array access expression) 访问数组的成分。数组可以通过 int 值索引, 也可以使用 short、byte 或者 char 值, 因为它们可以被单目数值提升 (§ 2.6.9) 变成 int 值。

所有的数组都从 0 开始。长度为 n 的数组可以用整数 0 到 n - 1 索引。所有的数组访问在运行期都被检查; 试图使用小于零或者大于等于数组长度的索引将使 ArrayIndexOutOfBoundsException 被抛出。

# 2.15 异常

当一个 Java 程序违反了 Java 语言的语义约束时, Java 虚拟机把这个错误作为异常 (exception) 向程序发出信号。这种违反的一个范例是试图在一个数组的界外索引该数组。Java 规定当语义约束被违反时一个异常将被抛出, 并且引起控制从异常发生的点到程序员可以指定的点的非局部转移。我们说异常在它的发生点被抛出 (thrown), 在控制转向的点被捕捉 (caught)。由于异常引起控制转移到方法之外而造成的方法调用结束称为突然约束 (complete abruptly)。

Java 程序也可以使用 throw 语句显式地抛出异常。这提供了对处理错误情况的老式风格的一个替换。老式风格在正常情况下不会出现负值的地方通过返回一个可识别的错误值, 例如整数值 -1, 来处理错误情况。

每个异常由类 Throwable 或者它的子类的实例表示, 这样的对象可用于从异常发生

的点到捕捉异常的处理器之间传送信息。处理器通过 try 语句的 catch 子句建立。在抛出异常的过程中,Java 虚拟机一个接一个地突然结束当前线程中已经开始但尚未结束执行的任何表达式、语句、方法和构造函数调用,静态初始化函数和域初始化表达式。这个过程持续进行,直到找到通过命名抛出的异常的类或者该异常的类的超类的名称而表明它将处理该异常的处理器。如果没有找到这样的处理器,则调用当前线程的父线程 ThreadGroup 的方法 uncaughtException。

Java 的异常机制与 Java 的同步模型结合在一起,使得当 synchronized 语句和 synchronized 方法调用突然结束时锁被合适地释放。

本部分谈及的特定异常是可被 Java 虚拟机的操作直接抛出的预定义的异常的子集。类库或者用户代码可以抛出另外的异常。这里不介绍这些异常。所有预定义的异常的信息参见《Java 语言规范》一书。

## 2. 15.1 引起异常的原因

异常因三种原因之一被抛出:

- 一个不正常的执行条件被 Java 虚拟机同步地检测到。这些异常不在程序的任意点抛出,而是在一个它们被规定为表达式求值或者语句执行的可能结果的点被抛出:
  - 违反 Java 语言的正常语义的操作,例如在数组的界外索引数组。
  - 装载或者连接 Java 程序的一部分中的错误。
  - 超过了资源的某些限制,例如使用太多的存储器。
- 在 Java 代码中执行一条 throw 语句。
- 发生异步异常,由于:
  - 类 Thread 的方法 stop 被调用;或者,
  - 虚拟机发生内部错误。

异常由类 Throwable 和它的子类的实例表示。这些类统称为异常类(exception classes)。

## 2. 15.2 处理异常

当异常被抛出时,控制从引起该异常的代码转移到处理该异常的 try 语句的最近的,动态地封闭的 catch 子句。

出现在 try 语句的 try 块内的语句或者表达式被 try 语句的 catch 子句动态地封闭(dynamically enclosed),或者这些语句或表达式的调用者被 catch 子句动态地封闭。

语句或者表达式的调用者(caller)取决于它发生在何处:

- 如果在方法内,则调用者是方法调用表达式。它的执行使方法被调用。
- 如果在构造函数或者实例变量初始化器内,则调用者是类实例调用表达式或者执行它以使对象被创建的 newInstance 的方法调用。
- 如果在静态初始化函数或者 static 变量的初始化器内,则调用者是使用类或者接口以使它被初始化的表达式。

一个特定的 catch 子句是否处理(handles)一个异常通过比较被抛出的对象的类和 catch

子句中参数的声明类型来决定。如果 catch 子句的参数的类型是该异常的类或者该异常的类的超类，则它处理该异常。等价地，catch 子句将捕捉任何是声明的参数类型的实例的异常对象。

异常被抛出时发生的控制转移使直到遇到能够处理该异常的 catch 子句之前的表达式和语句突然结束，然后通过执行该 catch 子句的块继续执行。引起异常的代码不再被恢复执行。

如果不能找到处理异常的 catch 子句，则当前线程（遇到该异常的线程）被终止，但是，只在所有的 finally 子句被执行，并且当前线程的父线程 ThreadGroup 的方法 uncaughtException 被调用后才终止当前线程。

在需要保证即使代码的其他块突然结束，代码的一个块总是在另一个块之后执行的情况下，可以使用带 finally 子句的 try 语句。如果 try-finally 或者 try-catch-finally 语句中的 try 或者 catch 块突然结束，那么即使最终没有找到匹配的 catch 子句、finally 子句仍在异常的传播中被执行。如果一个 finally 子句因一个 try 块突然结束而被执行，并且 finally 子句自身突然结束，则该 try 块突然结束的原因被抛弃并且突然结束的新原因从那里传播。

Java 中绝大多数异常的发生，作为它们发生所在的线程的一个动作的结果，是同步的，并且发生在 Java 程序中指定的、可能引起这种异常的点上。作为对比，异步异常是可能潜在地发生在 Java 程序执行的任何点的异常。

Java 中异步异常很少。它们只作为下面的结果发生：

- 类 Thread 或者 ThreadGroup 的方法 stop 的调用。
- Java 虚拟机中的 InternalError。

Stop 方法可以被一个线程调用以影响一个特定的线程组中的另一线程或者所有线程。它们是异步的是因为它们可能在别的线程执行的任何点发生。

InternalError 被认为是异步的，从而可以用与处理 stop 方法相同的机制处理它，像现在描述的那样。

Java 允许在异步异常被抛出之前进行一段小的但有限数量的执行。允许这个延迟是为了优化代码以在遵守 Java 语言的语义的同时，再处理这些异常的可行的点检测并抛出它们。

一个简单的实现可以在每个控制转移指令点上查询异步异常。由于 Java 程序具有有限的尺寸，这提供了检测一个异步异常的整体延迟的一个界限。由于在控制转移之间不会发生异步异常，代码生成器在控制转移之间的计算的重新排序上具有一些灵活性以得到更好的性能。

Java 中的所有异常都是精确的（precise）；当控制转移发生时，在异常抛出点之前执行的语句和求过值的表达式的所有结果必须看上去已经发生。出现在异常抛出点后的表达式、语句或者部分看上去都尚未被求值。如果优化的代码已经预测性地执行了异常发生点之后的一些表达式或者语句，则这种代码必须准备好对 Java 程序的用户可见的状态隐藏这种预测性的执行。

### 2.15.3 异常层次

Java 程序中的可能异常被组织成类的一个层次，这个层次以 Object 的直接子类 Throwable 为根。类 Exception 和 Error 是 Throwable 的直接子类。类 RuntimeException 是 Exception 的一个直接子类。

Java 程序可以在 `throw` 语句中使用这些已存在的异常类,或者定义另外的作为 `Throwable` 或者它的任何子类的适当子类的异常类。为了利用 Java 对异常处理者的编译期检查,通常把绝大多数新的异常类定义为受检查异常类,特别地,定义为不是 `RuntimeException` 的子类的 `Exception` 的子类。

## 2.15.4 类 `Exception` 和 `RuntimeException`

类 `Exception` 是普通程序可以从中恢复的所有标准异常的超类。

类 `RuntimeException` 是类 `Exception` 的一个子类。`RuntimeException` 的子类是不受检查异常类。包 `java.lang` 定义了以下标准不受检查运行期异常:

- `ArithmaticException`:一个异常的算术情况发生。例如以零为除数的整数除法或者取余操作。
- `ArrayStoreException`:试图向一个数组成分存储一个其类与该数组的成分类型不能赋值相容的值。
- `ClassCastException`:试图把一个对象引用转换成一个不合适的类型。
- `IllegalMonitorStateException`:一个线程试图等待或者通知别的线程等待一个它未锁定的对象。
- `IndexOutOfBoundsException`:由两个值或者一个索引和一个长度指定的某种索引(例如对数组的、字符串的或者向量的)或者子范围超出了范围。
- `NegativeArraySizeException`:试图创建一个负长度的数组。
- `NullPointerException`:试图在需要对象引用的情况下使用一个 `null` 引用。
- `SecurityException`:检测到一个对安全的违犯。

类 `Error` 和它的标准子类是那些普通程序通常不能从中恢复的异常。类 `Error` 是 `Throwable` 的一个单独的子类。在类层次中与 `Exception` 区分开,以允许程序使用惯用法

```
} catch (Exception e) {
```

捕捉可从中恢复的全部异常而不捕捉通常不能从中恢复的错误。包 `java.lang` 定义了这里描述的全部错误。

Java 虚拟机在发生装载(§ 2.16.2),链接(§ 2.16.3)和初始化(§ 2.16.4)错误时抛出 `LinkageError` 的子类的一个实例对象:

- 装载进程在(§ 2.16.2)中描述。那里描述了错误 `ClassFormatError`, `ClassCircularityError` 和 `NoClassDefFound`。
- 链接进程在(§ 2.16.3)中描述。那里描述链接错误(`IncompatibleClassChangeError` 的全部子类),即 `IllegalAccessException`, `InstantiationException`, `NoSuchFieldError`, 和 `NoSuchMethodError`。
- 类检验进程在(§ 2.16.3)中描述,那里描述检验失败错误 `VerifyError`。
- 类准备进程在(§ 2.16.3)中描述。那里描述的准备错误是 `AbstractMethodError`。
- 类初始化进程在(§ 2.16.4)中描述。如果执行静态初始化函数或者 `Static` 域(§ 2.11)的初始化器引起不是 `Error` 或者其子类的异常,则虚拟机将抛出错误 `ExceptionInInitializerError`。

当内部错误或者资源限制阻止 Java 虚拟机实现 Java 语言的语义时,它抛出类 `virtualMachineError` 的一个子类的实例对象,本规范定义以下虚拟机错误:

- `InternalError`: 由于实现虚拟机的软件中的错误,或者基础主机系统软件中的错误,或者硬件中的错误使 Java 虚拟机中发生的内部错误。当检测到这个错误时,它被异步地发送,它可能发生在 Java 程序中的任何点。
- `OutOfMemoryError`: Java 虚拟机用完了虚拟的或者物理的存储器,并且自动存储器管理者不能回收足够的存储器满足对象创建要求。
- `StackOverflowError`: Java 虚拟机用完了线程的栈空间,通常是由于线程因执行程序中的错误在做无限个递归调用。
- `UnknownError`: 发生了一个异常或者错误,但是由于某种原因 Java 虚拟机不能报告实际的异常或者错误。

## 2.16 执行

本部分规定 Java 程序执行中发生的行为。它围绕 Java 虚拟机和形成 Java 程序的类、接口以及对象的生命周期组织起来。它规范了启动 Java 虚拟机(§ 2.16.1),类和接口类型的装载(§ 2.16.2)、链接(§ 2.16.3)和初始化(§ 2.16.4)中使用的详细过程。然后它规定了创建新的类实例的过程(§ 2.16.6)。它以描述类的卸载(§ 2.16.8)和虚拟机退出(§ 2.16.9)时遵循的过程结束。

### 2.16.1 虚拟机启动

虚拟机通过调用某个指定类的方法 `main` 启动,传递给 `main` 一个字符串数组参数。这使这个指定的类被装载(§ 2.16.2),与它使用的别的类型链接(§ 2.16.3),并且被初始化(§ 2.16.4)。方法 `main` 必须声明为 `public`、`static` 和 `void`。

Java 虚拟机规范的初始类的行为超出了本规范的范围。但是,在使用命令行的主机环境中,通常把类的完整限定名称指定为命令行参数,并且把其后的命令行参数作为字符串参数供给方法 `main`。例如,在 Sun 的 UNIX 上的 JDK 实现中,命令行

```
java Terminator Hasta la vista Baby!
```

将通过调用类 `Terminator`(未命名的包中的一个类)的方法 `main` 启动 Java 虚拟机,传递给 `main` 一个包含四个字符串“Hasta”、“la”、“vista”和“Baby!”的数组。

现在我们略述虚拟机执行 `Terminator` 时可能采取的步骤,作为以后部分中进一步描述的装载、链接和初始化进程的范例。

开始试图执行类 `Terminator` 的方法 `main` 时发生类 `Terminator` 尚未被装载——即,虚拟机当前不包括该类的二进制代表。然后虚拟机使用 `ClassLoader`(§ 2.16.2)试图寻找这样的二进制代表。如果这个进程失败,则抛出一个错误。装载进程在(§ 2.16.2)中进一步描述。

`Terminator` 被装载后,在可以调用 `main` 之前,必须初始化 `Terminator`,并且在初始化之前总是必须链接一个类型(类或者接口),链接包括检验,准备和(可选的)解析。链接在 § 2.16.3 中进一步描述。

检验核对 Terminator 被装载的代表是良好的,带有合适的符号表。检验也核对实现 Terminator 的代码服从 Java 虚拟机的语义要求。如果在检验中检测到问题,则抛出一个错误。检验在 § 2.16.3 中进一步描述。

准备包括虚拟机内部使用的静态存储器和任何数据结构,例如方法表的分配。如果在准备中检测到问题,则抛出一个错误,准备在 § 2.16.3 中进一步描述。

解析是检查 Terminator 对别的类和接口的符号引用的进程。通过装载 Terminator 中提及的别的类和接口并核对这些引用是正确的来进行解析。

在初始链接时解析步骤是可选的。一个实现可以很早就解析正在被链接的类或者接口的符号引用,甚至能够递归地解析被进一步引用的类和接口的所有符号引用。(这种解析可能导致进一步装载和链接步骤中的错误。)这种实现选择代表了一个极端,它与多年来 C 语言的简单实现中所做的静态链接种类相似。

一个实现也可以选择只在一个符号引用正在被活跃地使用时才解析它。对所有符号引用固定地使用这一策略代表了解析的“最迟”形式。在这种情况下,如果 Terminator 对另一个类有几个符号引用,则可能一次解析这些引用中的一个——如果在程序执行中从未使用这些引用,则可能一个也不解析。

在进行解析时的唯一要求是,在解析中检测到的任何错误,必须在程序中由程序所采取的某些可能直接或者间接地需要链接涉及该错误的类或者接口的某个动作处抛出。在前面描述的实现选择“静态”范例中,程序执行前可能发生装载和链接错误,如果它们涉及类 Terminator 中提及的类或者接口,或者涉及任何被进一步递归引用的类和接口。在实现“最迟”解析的系统中,只有当一个符号引用被活跃地使用时才会抛出这些错误。

解析进程在 § 2.16.3 中进一步描述。

在我们的运行范例中,虚拟机仍在尝试执行类 Terminator 的方法 main。这是试图活跃地使用该类,仅在类已被初始化之后才允许。

初始化由按文本顺序对类 Terminator 的任何类变量初始化器和静态初始化函数的执行组成。但是在 Terminator 能够被初始化之间,它的直接超类——以及,递归地,它的直接超类的直接超类——必须被初始化。在最简单的情况下,Terminator 以 Object 作为它的隐式直接超类;如果类 Object 尚未被初始化,则在 Terminator 初始化之前 Object 必须被初始化。

如果类 Terminator 有另一个超类 Super,则在 Terminator 之前必须初始化 Super。这需要对 Super 装载、检验准备——如果这些工作还没有做。并且,依赖于实现,也可能包括递归地解析 Super 的符号引用等等。

因此初始化可能引起装载、链接和初始化错误,包括涉及别的类型的这类错误。

初始化进程在 § 2.16.4 中进一步描述。

最后,在类 Terminator 的初始化完成后(在此过程中可能因此引起别的装载、链接和初始化的发生)。Terminator 的方法 main 被调用。

## 2.16.2 装载

装载(loading)是指寻找一个具有特定名称的类或者接口类型的二进制形式,并且用这个二进制形式构造一个代表该类或者接口的 class 对象的进程。其中特定的名称可能是通过对 fly 的计算,但更通常地是通过获取以前由编译器从源代码计算得到的,二进制表示来寻找。正

常地,类或者接口的二进制格式是 class 文件格式(参见第四章,“class 文件格式”)。

装载进程由类 ClassLoader 和它的子类实现。ClassLoader 的不同子类可以实现不同的装载策略。特别地,类装载器可以根据预测的使用预先取回类或者接口,或者一起装载一组相关的类来高速缓存它们的二进制表示。例如,如果由于一个类的旧的编译后版本被类装载器高速缓存而使类的新的编译后版本不能被找到,则类装载器的高速缓存行为对正在运行的 Java 应用程序可能是不完全透明的。但是,只在程序中不预先取回或装载一组类就可能产生装载错误的点上反映这些装载错误是类装载器的责任。

如果在类装载中发生一个错误,则抛出类 LinkageError 的下述子类之一的一个实例,它在 Java 程序中任何(直接或者间接地)使用该类型的点被抛出。

- ClassCircularityError:因一个类或者接口是其自身的超类或者超接口(§ 2.13.2)而不能被装载。
- ClassFormatError:声称其指定所要求的编译后的类或者接口的二进制数据是损坏的。
- NoClassDefNotFoundError:相关的类装载器找不到要求的类或者接口的定义。

### 2.16.3 链接:检验、准备和解析

链接(Linking)是取得类或者接口类型的二进制形式并把它链接成 Java 虚拟机的运行期状态以使它可以被执行的进程。类或者接口类型总是在链接前被装载。链接包括三种不同的行为:检验、准备和对符号引用的解析。

Java 在链接行为(以及,由于递归装载)发生时允许实现的灵活性,只要遵守语言的语义,类或者接口在被初始化前被完全地检验和准备,并且链接中检测到的错误在程序的某个点上抛出,在这个点上程序采取某些可能要求链接涉及该错误的类或者接口的动作。

例如,一个实现可以选择只在使用类或者接口中的某个符号引用时才单独地解析它(迟或者晚解析);或者,例如在类正在被检验时立即全部解析它们(静态解析)。这表示在某些实现中,在类或者接口被初始化之后可以继续进行解析进程。

检验(verification)保证类或者接口的二进制表示的结构是正确的。例如,它核对每条指令有一个有效的操作代码;每条转移指令转移到另外某条指令的开始,而不是一条指令的中间;每个方法被提供给一个结构正确的签名;每条指令都服从 Java 语言的类型规则。

如果在检验中发生错误,则在 Java 程序中引起检验该类的点抛出类 LinkageError 的下述子类之一的一个实例。

- VerifyError:类或者接口的二进制表示不能通过确认它服从 Java 语言的语义并且不侵犯 Java 虚拟机的完整性所需的一组检查。

准备(preparation)包括创建类或者接口的静态域以及把这些域初始化为标准的缺省值(§ 2.5.1)。这不要求执行任何 Java 代码。静态域的显式初始化器的执行是初始化(§ 2.16.4),而不是准备的一部分。

Java 实现必须在准备中检测下述错误:

- AbstractMethodError:没被声明为 abstract 的类具有一个 abstract 方法。例如,在编译继承了现在是 abstract 方法声明的另一个类之后,如果原始不是 abstract 的方法变成了 abstract,这个错误就可能发生。

Java 虚拟机的实现在准备期可能预先计算另外的数据结构以使类和接口上以后的操作更加有效率。一个特别有用的数据结构是“方法表”，或者允许调用任何在调用时无需搜寻其实例的类的超类的方法的别的数据结构。

一个 Java 二进制文件用别的类和接口的完整限定名称(§ 2.7.9)符号引用这些类和接口以及它们的域、方法和构造函数。对于域和方法，这些符号引用包括声明这些域或者方法的类或者接口的名称，以及这些域或者方法自身的名称和适当的类型信息。

在一个符号引用可以使用之前它必须经历解析(resolution)，在解析中符号引用被核对是正确的，并且如果该引用被重复使用，通常用一个能够被更有效地处理的直接引用替换它。

如果在解析中发生错误，则类 `IncompatibleclasschangeError` 的下述子类之一，或者某个别的子类，或者 `IncompatibleClassChangeError` 自身(类 `LinkageError` 的一个子类)的一个实例，可能在使用该类型的符号引用的 Java 程序中的任何点抛出：

- `IllegalAccessException`: 遇到一个符号引用，这个符号引用对包含该引用的代码不能访问的类的域指定使用或者赋值，或者调用方法，或者创建该类的实例。代码不能访问它们是因为域或者方法被声明为 `private`、`protected` 或者缺省访问权(不是 `public`)，或者因为类没有声明为 `public`。例如，如果一个原始地声明为 `public` 的域在另一个引用该域的类被编译后变成 `private`，则这种情况就可能发生。
- `InstantiationException`: 遇到用于类实例创建表达式的一个符号引用，但是由于该引用是引用一个接口或者一个 `abstract` 类而不能创建该实例。例如，如果一个原始地不是 `abstract` 的类在另一个引用这个有问题的类的类被编译后，变成 `abstract`，则这种情况就可能发生。
- `NoSuchFieldError`: 遇到一个引用某个特定的类或者接口的一个特定的域的符号引用，但是这个类或者接口没有声明具有那个名称的域。(特别地，符号引用只是那个类或者接口继承的域是不够的。)例如，如果在引用某个类的一个域的另一个类被编译后从声明该域的类中删除该域的声明，则这种情况就可能发生。
- `NoSuchMethodError`: 遇到一个引用一个特定的类或者接口的特定的方法的符号引用，但是这个类或者接口没有声明具有那个名称和签名的方法(特别地，符号引用只是那个类或者接口继承的方法是不够的)。例如，如果在另一个引用某个类的方法的类被编译后，从声明该方法的类中删除该方法的声明，则这种情况就可能发生。

## 2.16.4 初始化

类的初始化(initialization)由对类中声明的静态初始化函数(§ 2.11)和静态域的初始化器(§ 2.9.2)的执行组成。接口的初始化由对接口(§ 2.13.4)中声明的域的初始化器的执行组成。

在类被初始化以前，它的超类必须被初始化。但是由该类实现的接口不需要初始化。类似地，在接口被初始化以前它的超接口不需要初始化。

类或者接口类型 T 在它首次被活跃使用时(active use)被初始化(initialized)，这将发生在以下条件满足时：

- T 是类并且一个实际地在 T 中声明(而不是从超类中继承的)方法被调用。

- T 是类并且类 T 的构造函数被调用。
- T 中声明的(而不是从超类或者超接口中继承的)非常数域被使用或者被赋值。常数域是(显式地或者隐式地)既是 final 又是 static, 并且用一个编译期常数表达式的值初始化的域。Java 规定对这种域的引用必须在编译期对这个编译期常数值的拷贝解析, 因此对这种域的使用不会是活跃使用。

对类型的其他所有使用都是被动使用。

这里的意思是指具有一组将其置成为固定状态的初始化器的类型, 并且这个状态是别的类观测到的第一个状态。静态初始化函数和类变量初始化器按照文本顺序执行, 并且不可以引用该类中其声明文本出现在使用之后的类变量, 即使这些类变量是可见的。设计这个限制是为了在编译期检测绝大多数循环的或者损坏的初始化。

在类被初始之前如果其超类未曾被初始化, 则它们被初始化。

对域的引用只在实际声明该域的类或者接口中是活跃使用, 即使该域可能是通过子类、子接口或者实现接口的类的名称引用的。

对接口的初始化自身不要求对它的任何超接口初始化。

## 2.16.5 详细的初始化过程

因为 Java 是多线程的, 所以类或者接口的初始化要求仔细的同步, 因为同时某个别的线程可能正在试图初始化同一个类或者接口。也有可能作为类或者接口初始化的一部分, 要求递归地初始化该类或者接口。例如, 类 A 的一个变量初始化器可能调用无关类 B 中的一个方法, 这个方法又可能调用类 A 中的方法, Java 虚拟机的实现通过使用下述过程负责管理同步和递归初始化。它假定 class 对象已被检验和准备, 并且 class 对象包含可以指示以下四种情况之一的状态:

- class 对象被检验和准备, 但是没有被初始化。
- class 对象正在被某个特殊的线程 T 初始化。
- class 对象被完整地初始化并且可以使用。
- class 对象处于错误的状态, 可能由于检验或者准备步骤失败了, 或者由于试图初始化并且失败了。

这以后的类或者接口的初始化过程如下:

(1) 同步代表类或者接口的 class 对象的初始化。这包括一直等待到当前线程可以获得那个对象的锁(§ 8.13)。

(2) 如果某个别的线程正在初始化这个类或者接口, 则对这个 Class 对象(它临时地释放了锁)wait, 当前线程从 wait 中醒来时, 重复这一步骤。

(3) 如果当前线程正在对类或者接口初始化, 则这一定是初始化的一个递归要求。释放 Class 对象上的锁并正常地结束。

(4) 如果类或者接口已被初始化, 则不再需要进一步动作。释放 Class 对象上的锁并正常地结束。

(5) 如果 Class 对象处于错误的状态, 则不可能初始化。释放 Class 对象上的锁并抛出一

个 NoClassDefFoundError。

(6) 否则,记录 Class 对象正在被当前线程初始化的事实,并释放 Class 对象上的锁。

(7) 接着,如果 class 对象代表一个类而不是接口,并且该类的超类尚未被初始化,则对超类递归地执行这一过程,如果需要,首先检验并准备这个超类。如果因一个抛出的异常使超类初始化突然结束,则锁定这个 class 对象。把它标为错误的,通知所有正在等待的线程,释放锁,并且突然结束,抛出与初始化该超类所引起的相同的异常。

(8) 接着,除了 final static 变量和接口的其值为编译期常数的域被首先初始化外,按文本顺序执行类变量的初始化器和类的静态初始化函数,或者接口的域初始化器,就好像它们是一个单个的块。

(9) 如果初始化函数的执行正常地结束,则锁定这个 class 对象,把它标为被完整地初始化,通知所有正在等待的线程,释放锁,并正常地结束该过程。

(10) 否则,初始化函数一定是通过抛出某个异常 E 突然结束。如果 E 的类不是 Error 或者其子类之一,则用 E 作为参数创建类 ExceptionInInitializerError 的一个新实例,并在以后的步骤中用这个对象代替 E。但是如果因发生一个 OutOfMemoryError 而不能创建 ExceptionInInitializerError 的新实例,则在以后步骤中用一个 OutOfMemoryError 对象代替 E。

(11) 锁定这个 class 对象,把它标为错误的,通知所有正在等待的线程,释放锁,并且带着原因 E 或者在前面步骤中确定的 E 的替换突然结束这个过程。

在 Java 的某些早期实现中,类初始化中的异常被忽略,而不是引起一个这里描述的 exceptionInInitializerError。

## 2. 16. 6 新的类实例的创建

当以下情况之一发生时一个新的类实例被显式地创建:

- 对一个类实例创建表达式的求值创建该表达式中出现其名称的类的一个新实例。
- 类 class 的 newInstance 方法调用创建调用该方法的 class 对象代表的类的一个新实例。

在以下情况下一个新的类实例可能被隐式地创建:

- 装载一个包含 String 文字的类或者接口可能创建一个代表该文字的新的 String 对象(§ 2. 4. 7)。如果一个相同的 String 以前被内部化,则这种情况可能不会发生。
- 执行一个不是某个常数表达式的一部分的字符串连接操作符有时会创建一个代表结果的新的 String 对象。字符串连接操作符也可能为一个基本类型(§ 2. 4. 1)的值创建一个临时包装器对象。

每种情况都标识一个特殊的,带着指定参数(可能没有)的构造函数调用作为类实例创建过程的一部分。

每当一个类实例被创建时,都为该类类型中声明的所有实例变量以及该类类型的每个超类中的所有实例变量,包括可能被隐藏的所有实例变量,分配存储器空间。如果没有足够的可获得的空间为该对象分配存储器,则该类实例创建带着一个 OutOfMemoryError 突然结束。否则,这个新对象的所有的,包括在超类中声明的,实例变量被用它们的缺省值(§ 2. 5. 1)初始化。在作为结果返回这个新创建的对象的一个引用之前,使用以下过程处理被指示的构造函数

以初始化新的对象：

- (1) 把构造函数的参数赋给为该构造函数调用而新创建的参数变量。
- (2) 如果该构造函数以对同一类(用 this)中的另一构造函数的显式构造函数调用开始, 则对参数求值并且用这同样的五个步骤递归地处理那个构造函数调用。如果那个构造函数的调用突然结束, 则该过程以相同的原因突然结束。否则, 继续第(5)步。
- (3) 该构造函数不以对同一类(用 this)中的另一构造函数的显式构造函数调用开始。如果这个构造函数不是类 Object 的构造函数, 则该构造函以对超类(用 Super)的构造函数的显式或者隐式的调用开始。对参数求值并且用这同样的五个步骤递归地处理那个超类的构造函数调用。如果那个构造函数调用突然结束, 则该过程以相同的原因突然结束。否则, 继续第(4)步。
- (4) 执行该类的实例变量初始化器, 按照它们在该类的源代码中的文本出现的从左到右的顺序, 把它们的值赋给相应的实例变量。如果任何这些初始化器的执行引起异常, 则不再进一步处理任何初始化器, 并且以相同的异常突然结束该过程。否则, 继续第(5)步。(在某些早期 Java 实现中, 对于是一个常数表达式的、其值等于它的类型的缺省初始化值的域初始化器表达式, 编译器错误地省略了初始化这种域的代码, 这是一个错误。)
- (5) 执行该构造函数主体的其余部分。如果这一部分的执行突然结束, 则该过程以相同的原因突然结束。否则, 该过程正常地结束。

与 C++ 不同, Java 语言不指定新的类实例创建中方法调度的变化规则。如果调用正在被初始化的对象的, 被在子类中覆盖的方法, 即使在新的对象被完全创建之前, 也使用新方法。

## 2.16.7 类实例的终止

类 Object 具有一个称为 finalize 的 protected 方法, 它可以被别的类覆盖。可以为一个对象调用的 finalize 的特殊定义称为该对象的终止函数 finalizer 在一个对象的存储器被垃圾回收器回收之前, Java 虚拟机调用该对象的终止函数。

终止函数提供了一个释放不能被自动存储器管理者自动释放的资源(例如文件描述符或者操作系统图形上下文)的机会。在这种情况下, 只回收该对象使用的存储器不能保证它持有的资源被回收。

Java 语除了规定终止函数的调用将在对象的存储器被重新使用之前发生外, 并不指定终止函数何时被调用。而且, Java 语言不指定哪个线程将调用任何给定对象的终止函数。如果在终止中抛出一个不能被捕获的异常, 则该异常被忽略并且该对象的终止停止。

类 Object 中声明的 finalize 方法不做任何动作。但是, 类 Object 声明了一个 finalize 方法的事实意味着任何类的 finalize 方法总是可以调用其超类的 finalize 方法, 这通常是一个好的做法。(与构造函数不同, 终止函数不自动地调用超类的终止函数; 这样的调用必须显式地编码。)

为了效率, 一个实现可以跟踪不覆盖类 Object 的 finalize 方法的类, 或者普通地覆盖它, 例如:

```
protected void finalize() { Super.finalize();}
```

我们鼓励实现把这样的对象当作具有没被覆盖的终止函数来处理, 并更有效地终止它们。就像

任何别的方法一样, finalize 方法可以被显式地调用。但是,这样做对对象最终自动终止没有任何影响。

Java 虚拟机对 finalize 方法的调用顺序不加限制, 终止函数可以按任何顺序调用, 或者甚至并发地调用。

作为一个范例, 如果一组被循环地链接未终止对象变得不可达到, 则所有的对象可能变得可以一起终止。最终, 这些对象的终止函数可以按任何顺序被调用, 或者甚至用多线程并发地调用。如果自动存储器管理者以后发现这些对象不可达到, 则它们的存储器被回收。

## 2.16.8 类和接口的终止和卸载

Java 虚拟机可以提供一些机制, 类按照这些机制被终止和卸载。<sup>[2]</sup>当前版本的《Java 语言规范》一书中没有规范这些机制的细节。一般来说, 成组的相关类和接口类型将被一起卸载。例如, 这可用于卸载被一个特定的类装载器装载的一组相关类型。例如, 这样的组可能由实现一个基于 Java 的浏览器(如 HotJava)中的单个 applet 的全部类组成。

当一个类的任何实例仍可达到时这个类不能被卸载。当代表一个类或者接口的 class 对象仍是可达到的时这个类或者接口不能被卸载。

如果一个类声明了一个不带参数和返回结果的类方法 ClassFinalize

```
static void classFinalize() { ... }
```

则在该类被卸载之前这个方法被调用。像对象的 finalize 方法一样, 这个方法将只被自动地调用一次。这个方法可以被可选地声明为 private、protected 或者 public。

## 2.16.9 虚拟机退出

当两件事情之一发生时 Java 虚拟机停止它的全部动作并退出:

- 所有的非守护线程(§ 2.17)停止。
- 某个线程调用了类 Runtime 或者类 system 的 exit 方法, 并且退出操作为安全管理器所允许。

Java 程序可以规定所有具有终止函数的对象的未被自动调用的终止函数在虚拟机退出之前运行。这通过用参数 true 调用类 System 的方法 runFinalizersOnExit 来完成。<sup>[3]</sup>缺省的情况是在退出时不运行终止函数, 并且这个行为可以通过用参数 false 调用 runFinalizersOnExit 来复原。只在调用者被允许 exit 时才允许调用 runFinalizersOnExit 方法, 否则就被完全管理者拒绝。

## 2.17 线程

虽然以前绝大部分讨论只是关于 Java 代码被单个线程执行的行为, 但是每个 Java 虚拟机都能够支持多个线程的同时执行。这些线程独立地执行操作于驻留在共享的主存储器中的 Java 值和对象上的 Java 代码。多线程可以通过多个硬件处理器、时间分片单个硬件处理器或者时间分片多个硬件处理器来支持。

[2] 类的终止与卸载与 Sun 的 JDK 版本 1.0.2 实现的不同。

[3] Sun 的 JDK 版本 1.0.2 中没有实现方法 runFinalizersOnExit。

任何线程都可以标为守护线程(daemon thread)、当运行在某个线程中的代码创建一个新的 Thread 对象时,当且仅当创建线程是一个守护线程,新的线程才被初始地标为守护线程。程序可以通过调用类 Thread 中的 setDaemon 方法改变一个特定的线程是否为守护线程。Java 虚拟机初始地带着一个单个的非守护线程启动,这个线程通常调用某个类的方法 main。虚拟机也可以为内部用途创建别的守护线程。Java 虚拟机在所有的非守护线程死亡后退出(§ 2.16.9)。

Java 通过提供同步(synchronizing)线程的并发行为的机制来支持编制那种显然并发,但仍表现为确定性行为的程序。为同步线程,Java 使用监视器(monitor),它是同时只允许一个线程执行该监视器所保护的代码区的高层机制。监视器的行为用锁(lock)解释。与每个对象相关联地有一把锁。

synchronized 语句执行只与多线程操作相关的两个特殊动作:

(1) 在计算一个对象的引用之后和执行它的主体之前,它锁定该对象相关联的锁。

(2) 在主体的执行正常地或者突然地结束之后,它解锁同一个锁,为了方便,一个方法可以声明为 synchronized,这样的方法的行为就好像它的主体包含在一个 synchronized 语句中一样。

类 Object 的方法 Wait、notify 和 notifyAll 支持控制有效地从一个线程转移到另一个线程。不是简单地“旋转”(重复地锁定和解锁同一个对象以看某个内部状态是否改变了),那将消费计算负担,而是一个线程可以用 wait 挂起自己,直到另一个线程用 notify 或者 notifyAll 唤醒它。这在线程具有生产者一消费者关系(积极地为一个共同目标合作)而不是互相排斥关系(在共享一个公用资源时试图避免冲突)的情况下特别合适。

在线程执行代码时,它进行一系列的动作。线程可以使用一个变量的值或者赋给它一个新值。(别的动作包括算术操作,条件检查,和方法调用,但是这些不直接涉及变量。)如果两个或者更多的并发线程操作在一个共享变量上,则变量上的操作可能产生与时间有关的结果。这种时间依赖性是并发程序设计所固有的,并且成为了 Java 中程序的结果不仅仅由《Java 语言规范》一书中确定的很少的几个地方之一。

每个线程有一个工作存储器,它可以在其中保留与别的线程共享的主存储器中的变量的值的拷贝。为了访问一个共享的变量,线程通常首先获得一个锁并清除它的工作存储器。这保证了共享的值以后将从共享的主存储器装载到该线程的工作存储器。通过解锁一个锁,线程保证该线程在其工作存储器中持有的值被写回主存储器。

线程与主存储器间的、进而在线程之间的交互作用可以用特定的低层动作解释。有关于这些动作可以发生的顺序的规则。这些规则对 Java 的任何实现加了约束,而且 Java 程序员可以依靠这些规则预测并发 Java 程序的可能的行为。但是,这些规则确实故意地给实现者特定的自由。目的是允许可以大大提高并发代码的速度和效率的特定的标准硬件和软件技术。

简而言之,这些规则的重要性如下:

- 同步构造的适当使用将允许通过共享的变量将一个或一组值可靠地从一个线程传递到另一个线程。
- 当线程使用一个变量的值时,它包含的值实际上是由该线程或者某个别的线程存储到这个变量的值。即使程序不包含合适的同步代码也是这样。例如,如果两个线程把对不

同对象的引用存储到同一个引用值，则这个变量将包含这两个对象的引用之一，而不是某个别的对象的引用或者一个损坏的引用值。（对 long 和 double 值有一个特殊的例外，参见 § 8.4。）

- 在缺少显式同步时，Java 实现按照可能令人奇怪的顺序自由地更新主存储器。因此，希望避免奇怪的结果的程序员应当使用显式同步。

线程与主存储，从而相互之间的交互作用的细节在第八章“线程和锁”中详细讨论。

## 第三章 Java 虚拟机的结构

本书规范一台抽象的机器。它不记录包括 Sun 的 Java 虚拟机的任何特定实现。

要正确地实现 Java 虚拟机,你只需能读懂 Java class 文件格式并正确地进行 class 文件中规范的操作。不属于 Java 虚拟机规范的实现细节会不必要地约束实现者的创造力,提供它们只是为了使阐述更清楚。例如,运行期数据区的存储器布局,使用的垃圾回收算法,以及对字节码的任何优化(例如,把它们翻译成机器码)都留给实现者去决定。

### 3.1 数据类型

像 Java 语言一样,Java 虚拟机操作在两种类型之上:基本类型(primitive types)和引用类型(reference types)。相应地,有两种值,可以存储在变量中,作为参数传递,被方法返回,以及对其进行操作:基本值(primitive values)和引用值(reference values)。

Java 虚拟机希望几乎所有的类型检查在编译期进行,而不是由 Java 虚拟机自己来做,特别地,数据不需要做标记或者可被检查以确定类型。作为替换,Java 虚拟机的指令集使用操作于特定类型的值上的指令来区分它的操作数类型。例如 iadd、ladd、fadd 和 dadd 都是对两个数值相加的 Java 虚拟机指令,但是它们分别要求类型为 int、long、float 和 double 的操作数。对于 Java 虚拟机的指令集中的类型支持的概述参见 § 3.11.1。

Java 虚拟机包含对对象的显式支持。对象或者是一个动态分配的类实例,或者是一个数组。对象的引用被认为具有 Java 虚拟机类型 reference。类型 reference 的值可被看作指向对象的指针。一个对象可以有多个引用。尽管 Java 虚拟机对对象操作,但是它从不对它们直接寻址。对对象的操作、传递和检查总是通过类型 reference 的值进行。

### 3.2 基本类型和值

Java 虚拟机支持的基本数据类型是数值类型(numeric types)和 returnAddress 类型。数值类型由整型(integral types)和浮点型(float-point types)组成。

整形分为:

- byte: 其值是8位有符号二进制补码整数
- short: 其值是16位有符号二进制补码整数
- int: 其值是32位有符号二进制补码整数
- long: 其值是64位有符号二进制补码整数
- char: 其值是表示 Unicode 版本1.1.5字符(§ 2.1)的16位无符号整数

浮点型分为:

- float,其值是32位 IEEE 754浮点数

- `double`, 其值是 64 位 IEEE 754 浮点数

`returnAddress` 类型的值是 Java 虚拟机指令的操作码的指针。只有 `returnAddress` 类型不是 Java 语言类型。

### 3.2.1 整型和值

Java 虚拟机的整数类型的值和 Java 语言的整数类型的值(§ 2.4.1)相同:

- 对于 `byte`, 从  $-128$  到  $127$ ( $-2^7$  到  $2^7 - 1$ ), 包括两端。
- 对于 `short`, 从  $-32\,768$  到  $32\,767$ ( $-2^{15}$  到  $2^{15} - 1$ ), 包括两端。
- 对于 `int`, 从  $-2\,147\,483\,648$  到  $2\,147\,483\,647$ ( $-2^{31}$  到  $2^{31} - 1$ ), 包括两端。
- 对于 `long`, 从  $-9\,223\,372\,036\,854\,775\,808$  到  $9\,223\,372\,036\,854\,775\,807$ ( $-2^{63}$  到  $2^{63} - 1$ ), 包括两端。
- 对于 `char`, 从 '\u0000' 到 '\uffff'; `char` 是无符号的, 因此 '\uffff' 在用于表达式中时表示 65535, 而不是 -1。

### 3.2.2 浮点型和值

Java 虚拟机的浮点类型的值和 Java 语言的浮点类型的值相同(§ 2.4.1)。浮点类型 `float` 和 `double` 表示单精度 32 位和双精度 64 位格式 IEEE 754 值, 这些值在 IEEE standards for Binary Floating-point Arithmetic, ANSI/IEEE std. 754-1985(IEEE, New York). 中规范。

IEEE 754 标准不仅包括正和负的有符号数量数字, 也包括正和负零, 正和负无穷(infinities), 以及一个特殊的 Not-a-Number(以后缩写为 NaN)值。这个值用于表示某些操作例如零被零除的结果。对于 `float` 和 `double` 类型都存在这些值。

类型 `float` 的有限非零值具有形式  $s \cdot m \cdot z^e$ , 这里  $s$  是 +1 或者 -1,  $m$  是小于  $2^{24}$  的正整数,  $e$  是介于包括两端在内的 -149 到 104 之间的一个整数。类型 `float` 的最大正有限浮点文字是  $3.402\,823\,47e+38F$ , 类型 `float` 的最小正非零浮点文字是  $1.402\,39846e-45F$ 。

类型 `double` 的有限非零值具有形式  $s \cdot m \cdot z^e$ , 这里  $s$  是 +1 或者 -1,  $m$  是小于  $2^{53}$  的正整数,  $e$  是介于包括两端在内的 -1075 到 970 之间的一个整数。类型 `double` 的最大正有限浮点文字是  $1.797\,693\,134\,862\,315\,70e+308$ , 类型 `double` 的最小正非零浮点文字是  $4.940\,656\,458\,412\,465\,44e-324$ 。

浮点正零和浮点负零的比较是相等的, 但是有能够区分它们的其他操作。例如,  $1.0$  被  $0.0$  除产生正无穷; 但是  $1.0$  被  $-0.0$  除产生负无穷。

除了 NaN, 浮点值是有序的(ordered)。从最小到最大排列, 它们是负无穷、负有限值、负零、正零、正有限值和正无穷。

NaN 是没有顺序的(unordered), 因此, 如果数值比较的任一个或者两个操作数都是 NaN, 则 数值比较的值是 false; 数值相等性检查如果任一个操作数是 NaN, 则其值为 false; 数值不相等性的检查如果任一个操作数是 NaN, 则其值是 true。特别地, 一个值与它自身的数值相等性检查当且仅当这个值是 NaN 时, 检查的值才是 false。

IEEE 754 定义了许多不同的 NaN 值, 但是没有规范在不同的情况下将产生哪个 NaN 值。为了避免可移植性问题, Java 虚拟机把这些 NaN 值合并成一个单个的概念的 NaN 值。

### 3.2.3 returnAddress 类型和值

returnAddress 类型由 Java 虚拟机的 jsr、ret 和 Jsr-w 指令使用。returnAddress 类型的值是 Java 虚拟机指令的操作码的指针。与数值基本类型不同，returnAddress 类型不对应任何 Java 数据类型。

### 3.2.4 没有 boolean 类型

尽管 Java 定义了 boolean 类型，但是 Java 虚拟机没有用于 boolean 值上的操作的指令。作为替换，操作于 boolean 值上的 Java 表达式被编译成用 int 数据类型表示 boolean 变量。

尽管 Java 虚拟机支持创建类型 boolean 的数组（参见 newarray 指令的描述），但是它不支持对 boolean 数组元素的访问和修改。类型 boolean 的数组用 byte 数组指令访问和修改。<sup>[1]</sup>

Java 虚拟机中处理 boolean 值的更多信息参见第七章“为 Java 虚拟机编译”。

## 3.3 引用类型和值

有三种 reference 类型：类类型、接口类型和数组类型，它们的值是对动态创建的类实例、数组或者实现接口的类实例或数组的引用。reference 值也可以是特殊的 null 引用，它不引用对象，在这里用 null 表示。null 引用初始没有运行期类型，但是可以转换成任何类型（§ 2.4）。

## 3.4 字

还没有提及 Java 虚拟机不同类型的值的存储器需求，只提及过这些值可以取的范围。Java 虚拟机不指定它的数据类型的大小。作为替换，Java 虚拟机定义了一个具有平台指定的大小的抽象概念字（word）。一个字足够大以持有类型 byte、char、short、int、float、reference 或者 returnAddress 的一个值，或者持有一个自身的指针。两个字足够大以持有更大的类型——long 或 double——的值。Java 的运行期数据区全部按照这些抽象的字定义。

一个字的通常大小是主机平台的一个指针的大小。在一个 32 位平台上，字是 32 位，指针是 32 位，long 和 double 自然地占两个字。一个 Java 虚拟机的无经验的 64 位实现可能会浪费用于存储 32 位数据的字的一半，但是也可能把一个 long 或者 double 全部存储在分配给它的两个字中的一个。

尽管特定的字的大小是平台指定的，但是它的选择是在实现的层次上做出的，而不是作为 Java 虚拟机设计的一部分。它在实现之外或者对为 Java 虚拟机编译的代码是不可见的。

在本书中，所有对字数据的引用都是指字的这个抽象的概念。

## 3.5 运行期数据区

### 3.5.1 PC 寄存器

Java 虚拟机能够支持多个线程的同时执行（§ 2.17）。每个 Java 虚拟机线程有它自己的

---

[1] 在 Sun 的 JDK1.0.2 版本中，boolean 数组实际上是 byte 数组，每个 boolean 元素使用 8 位。 37

PC(程序计数器)寄存器。在任何点,每个 Java 虚拟机线程执行一个单个方法的代码,这个方法是那个线程的当前方法(§ 3.6)。如果那个方法不是 native,则 PC 寄存器包含当前正在被执行的 Java 虚拟机指令的地址。如果当前正在被线程执行的方法是 native,则 Java 虚拟机的 PC 寄存器的值没有定义。Java 虚拟机的 PC 寄存器占一个字宽,这个宽度保证可以持有一个 returnAddress 或者该特定平台的一个自身的指针。

### 3.5.2 Java 栈

每个 Java 虚拟机线程(§ 2.17)具有一个私有的,与线程同时创建的 Java 栈(Java Stack)。Java 栈存储 Java 虚拟机框架(§ 3.6)。Java 栈等价于传统语言例如 C 的栈:它持有局部变量和部分结果,并参与方法的调用和返回。由于除了压入和弹出框架外栈从不被直接操纵,所以它实际上可以作为一个堆来实现,而且 Java 框架可以是堆分配的。Java 栈的存储器不需要是连续的。

Java 虚拟机规范允许 Java 栈的大小是固定的或者是动态变化的。如果 Java 栈是固定大小的,则每个 Java 栈的大小可以在该栈被创建时独立地选择。Java 虚拟机实现可以向程序员或者用户提供对 Java 栈初始大小的控制,以及,在动态扩展或者收缩 Java 栈的情况下,对 Java 栈大小的最大值和最小值的控制。

以下异常情况与 Java 栈相关:

- 如果线程中的计算需要比所允许的大的 Java 栈,则 Java 虚拟机抛出 StackOverflowError。
- 如果 Java 栈可以动态扩展,并且试图扩展 Java 栈但是没有足够的存储器来实现扩展,或者不能得到足够的存储器为一个新线程创建初始 Java 栈,则 Java 虚拟机抛出 OutOfMemoryError。

在 Sun 的 JDK 1.0.2 的 Java 虚拟机实现中,Java 栈是不连续的,并且随计算的需求独立地扩展。Java 栈不收缩,但是当它们相应的线程停止或者被杀死后这些栈被回收。扩展受到对任何一个 Java 栈的大小的限制。Java 栈的大小的限制可以在虚拟机启动时用“-oss”标志设置。Java 栈的大小的限制可以用于限制存储器消费或者捕捉逃走的递归。

### 3.5.3 堆

Java 虚拟机有一个所有线程(§ 2.17)间共享的堆(heap)。堆是从中分配所有类实例和数组的存储器的运行期数据区。

Java 堆在虚拟机启动时创建。对象的堆存储器由自动存储器管理系统(通常是垃圾回收器(garbage collector))回收;对象从不被显式地回收。Java 虚拟机不假定特殊类型的自动存储器管理系统,存储器管理技术可以根据实现者的系统要求选择。Java 堆可以是固定大小的,或者按照计算的需求扩展,并且可以在一个大的堆变得不必要时收缩。Java 堆的存储器不需要是连续的。

Java 虚拟机实现可以向程序员或者用户提供对堆初始大小的控制,以及,如果堆可以被动态扩展或者收缩,对堆大小的最大值和最小值的控制。

以下异常情况与 Java 堆相关:

- 如果计算要求比自动存储器管理系统可获得的更多的 Java 堆，则 Java 虚拟机抛出 OutOfMemoryError。

Sun 的 JDK 1.0.2 的 Java 虚拟机实现按照计算的要求动态扩展它的 Java 堆，但是从不收缩它的堆。它的初始值和最大可以在虚拟机启动时分别用“-ms”和“-mx”标志规定。

### 3.5.4 方法区

Java 虚拟机有一个所有线程(§ 2.17)间共享的方法区(method area)。方法区类似于传统语言的编译后代码的存储区，或者 UNIX 进程中的“正文”段。它存储每个类结构例如常数池、域和方法数据，以及方法和构造函数，包括类和实例初始化与接口类型初始化中用到的特殊方法(§ 3.8)的代码。

方法区在虚拟机启动时创建。尽管方法区逻辑上是垃圾回收堆的一部分，但是简单的实现可以选择既不作为垃圾回收它，也不压缩它。Java 虚拟机规范的这个版本不指定方法区的位置或者管理编译后代码使用的策略。方法区可以固定大小，或者可以根据计算的需求扩展，并且如果大的方法区变得不必要，可以收缩它。方法区的存储器不需要是连续的。

Java 虚拟机的实现可以向程序员或者用户提供对方法区初始大小的控制，以及，在可变大小的方法区的情况下，对方法区大小的最大和最小值的控制。

以下异常情况与方法区相关：

- 如果不能得到方法区中的存储器来满足分配要求，则 Java 虚拟机抛出 OutOfMemoryError。

Sun 的 JDK 1.0.2 的 Java 虚拟机实现根据计算的需求动态扩展它的方法区，但从不收缩。没有提供用户对方法区大小的最大和最小值的控制。

### 3.5.5 常数池

常数池(constant pool)是每个类或者每个接口的 Java class 文件(§ 4.4)中的 constant\_pool 表的运行期表示。它包含几种常数，范围从编译期已知的数值文字到必须在运行期解析的方法和域引用。常数池的功能类似于传统程序设计语言的符号表，尽管它比通常的符号表包含更宽的数据范围。

每个常数池都从 Java 虚拟机的方法区(§ 3.5.4)分配。类或者接口的常数池在该类或者接口的 Java class 文件被 Java 虚拟机成功地装载(§ 2.16.2)时创建。

以下异常情况与类或者接口的常数池的创建相关：

- 在装载 class 文件时，如果常数池的创建需要比 Java 虚拟机的方法区中可获得的更多的存储器，则 Java 虚拟机抛出 OutOfMemoryError。

常数池解析，一个常数池中的项上进行的运行期操作，有它自己的相关异常集。关于常数池的运行期管理的信息参见第五章。

### 3.5.6 自身方法栈

Java 虚拟机的实现可以使用传统的栈,通俗地叫作“C 栈”,来支持 native 方法,这些方法是用不是 Java 的语言编写的。自身方法栈也可用于用一种语言例如 C 实现 Java 虚拟机指令集的一个仿真器。不支持 native 方法,并且自身不依赖于传统的栈的实现不需要提供自身方法栈。如果提供自身方法栈,则它们通常在每个线程被创建时分配在每个线程基础上。

Java 虚拟机规范允许自身方法栈固定大小或者动态可变大小。如果自身方法栈是固定大小的,则每个自身方法栈的大小可以在栈被创建时独立地选择。在任何情况下,Java 虚拟机实现可以向程序员或者用户提供对自身方法栈初始大小的控制。在可变大小的自身方法栈的情况下,它也可以获得对方法栈的大小的最大和最小值的控制。

以下异常情况与 Java 栈相关:

- 如果线程中的计算要求的所允许的大的自身方法栈,则 Java 虚拟机抛出 `stackOverflowError`。
- 如果自身方法栈可以动态扩展,并且试图扩展自身方法栈,但是不能获得足够的存储器,或者不能获得足够的存储器为一个新线程创建初始的自身方法栈,则 Java 虚拟机抛出 `OutOfMemoryError`。

Sun 的 JDK 1.0.2 的 Java 虚拟机实现分配具有单个大小的固定大小的自身方法栈。它的自身方法栈的大小可以在虚拟机启动时用“-ss”标志设置。自身方法栈的大小限制可用于限制存储器消费或者捕捉 native 方法中逃走的递归。

Sun 的实现当前不检查自身方法栈的溢出。

## 3.6 框架

Java 虚拟机框架(frame)用于存储数据和部分结果,以及进行动态链接,返回方法的值,和调度异常。

每次 Java 方法被调用时创建一个新框架。当框架的方法结束时,不论是正常的还是不正常的结束(通过抛出一个异常),框架被撤销。框架从创建该框架的线程的 Java 栈(§ 3.5.2)分配。每个框架有它自己的局部变量集(§ 3.6.1)和自己的操作数栈(§ 3.6.2)。这些结构的存储器空间可以同时分配,因为局部变量区和操作数栈的大小是编译期已知的,并且框架数据结构的大小只依赖于 Java 虚拟机的实现。

在控制的一个给定线程的任何点,只有一个框架,即正在执行的方法的框架是活跃的。这个框架称为当前框架(current frame),它的方法称为当前方法(current method),在其中定义当前方法的类是当前类(current class)。局部变量和操作数栈上的操作总是引用当前框架。

如果一个框架的方法调用另一个方法或者它的方法结束,则框架不再是当前的。当一个方法被调用,控制转移到新方法时,一个新的框架被创建并成为当前框架。方法返回时,如果方法调用有结果,则当前框架把结果传递回上一框架。然后当前框架被抛弃,上一框架成为当前框架。Java 虚拟机框架可以被自然地看作分配在栈上,对每个 Java 线程(§ 2.17)有一个栈,但是它们也可以是堆分配的。

注意一个线程创建的框架是局部于那个线程的,不能被任何别的线程直接引用。

### 3.6.1 局部变量

对每个 Java 方法调用,Java 虚拟机分配一个 Java 框架(§ 3.6),它包含一组称为它的局部变量(local variables)的字。局部变量从那个数组的根按字偏移寻址。

局部变量总是一个字宽。为每个 long 或者 double 值保留两个局部变量。这两个局部变量通过第一个变量的索引寻址。

例如,一个索引为 n 并且包含类型 double 的一个值的局部变量实际上占据局部变量索引为 n 和 n+1 的两个字。Java 虚拟机不要求 n 是偶数。(对直觉实现,64位值在局部变量数组中不需要是64位对齐的。)实现者可以自由地决定在两个局部变量间分割一个64位数据值的合适方法。

### 3.6.2 操作数栈

对每个 Java 方法调用,Java 虚拟机分配一个 Java 框架(§ 3.6),它包含一个操作数栈(operand stack)。绝大多数 Java 虚拟机指令从当前框架的操作数栈取值,对它们进行操作,并把结果返回到同一个操作数栈。操作数栈也用于向方法传递参数和接受方法的结果。

例如,iadd 指令把两个 int 值加到一起。它要求被相加的 int 值是操作数栈顶的两个字,它们由以前的指令压入到那儿。两个 int 值都被从操作数栈弹出。它们被相加,并且它们的和被压回栈中。子计算可以嵌套在操作数栈上,产生可被包围的计算使用的值。

操作数栈的每个项占一个字宽。类型 long 和 double 的值被作为两个字压入到操作数栈。Java 虚拟机不要求操作数栈上的64位值是64位对齐的。实现者可以自由地决定在两个操作数栈之间分割一个64位数据值的合适方法。

操作数栈中的值必须用与它们的值合适的方法操作。例如,压入两个 int 值然后把它们当作一个 long 处理,或者压入两个 float 值然后用一个 iadd 指令相加它们,是错误的。一小部分 Java 虚拟机指令(dup 指令和 swap)对运行期数据区作为具有给定宽度的未处理值进行操作,而不考虑类型;不能用这些指令打断或者重排64位数据的字。在 Sun 的实现中通过 class 文件检验器(§ 4.9)加强了对操作数栈操纵的这些限制。

### 3.6.3 动态链接

Java 虚拟机的框架包含当前方法的类型的常数池的一个引用以支持对方法代码的动态链接(dynamic linking)。方法的 class 文件代码引用被调用的方法和通过符号引用访问的变量。动态链接把这些符号方法引用翻译成具体的方法引用,在需要时装载类以解析尚未定义的符号,并把变量访问释译成与这些变量的运行期位置相关联的存储结构中的合适偏移。

方法和变量的这种晚联编在一个方法使用的别的类中产生的变化破坏该代码的可能性比较小。

### 3.6.4 正常的方法结束

一个方法调用正常地结束(completes normally),如果那个调用不引起异常被抛出(§ 2.15, § 3.9)。异常或者从 Java 虚拟机直接抛出,或者作为执行一个显式的 throw 语句的结果抛

出。如果当前方法的调用正常地结束，则一个值可以返回到调用当前方法的方法。这在被调用的方法执行一个返回指令(§ 3.11.8)时发生，返回指令的选择必须与被返回的值的类型(如果有)适合。

在这种情况下 Java 虚拟机框架用于恢复调用者的状态，包括它的局部变量和操作数栈，并且调用者的程序计数器恰当地增长以跳过方法调用指令。然后返回值(如果有)压入到调用方法的框架的操作数栈中，执行在这个框架中继续正常地进行。

### 3.6.5 不正常的方法结束

一个方法调用不正常地结束(completes abnormally)，如果方法中 Java 虚拟机指令的执行引起 Java 虚拟机抛出一个异常(§ 2.15, § 3.9)，并且这个异常没有在方法中处理。对一个显式 throw 语句的求值也引起一个异常被抛出，并且，如果这个异常没有被当前方法捕捉，将引起不正常的方法结束。一个不正常地结束的方法调用不会向它的调用者返回值。

### 3.6.6 附加信息

Java 虚拟机框架可以扩展，包含附加的实现指定信息，例如调试信息。

## 3.7 对象的表示

Java 虚拟机不为对象要求任何特殊的内部结构。在 Sun 的 Java 虚拟机的当前实现中，对类实例的引用是指向一个句柄(handle)的指针，这个句柄自身是一对指针：一个指向包含该对象的方法和代表该对象类型的 class 对象的指针的表，另一个指向在 Java 堆中为该对象数据分配的存储器。

别的 Java 虚拟机实现可以使用诸如直接插入高速缓存技术，而不是方法表调度技术，而且它们可以不用也可以不用句柄。

## 3.8 特殊的初始化方法

在 Java 虚拟机层次上，每个构造函数(§ 2.12)作为具有特殊名称<init>的实例初始化方法(instance initialization method)出现。这个名称由 Java 编译器提供。由于名称<init>不是一个有效的标识符，所以它不能被 Java 程序员直接使用。实例初始化方法只能在 Java 虚拟机内由 invokespecial 指令调用，并且它们只能在未初始化的类实例上调用。实例初始化方法的访问许可(§ 2.7.8)与从中获得该实例初始化方法的构造函数的访问许可相同。

在 Java 虚拟机层次上，类或者接口通过调用它的不带参数的类或者接口初始化方法(class or interface initialization method)来初始化(§ 2.16.4)。类或者接口的初始化方法具有特殊的名称<clinit>，这个名称由 Java 编译器提供。由于名称<clinit>不是一个有效的标识符，所以它不能被 Java 程序员直接使用。类和接口的初始化方法由 Java 虚拟机显式地调用；它们从不被直接地从 Java 代码或者任何 Java 虚拟机指令调用，而只是作为类初始化进程的一部分被间接地调用。

### 3.9 异常

一般来说,抛出一个异常导致控制的立即动态转移,这个转移可能会退出多个 Java 语句以及多个构造函数调用、静态初始化函数和域初始化器的求值以及方法调用,直到找到捕捉被抛出的值的 catch 子句(§ 2.15.2)。

如果在当前方法中不能找到这样的 catch 子句,则当前方法调用不正常地结束(§ 3.6.5)。它的操作数栈和局部变量被抛弃,它的框架被弹出,恢复调用方法的框架的状态。然后异常在调用者的框架的上下文中被重新抛出,并且这样继续方法调用链。如果在到达方法调用链顶之前不能找到合适的 catch 子句,则抛出异常的线程的执行被停止。

在 Java 虚拟机层次上,每个 catch 子句描述它在其中是活跃的 Java 虚拟机指令范围,描述它将处理的异常的类型,并给出处理它的代码的地址。如果引起异常的指令在适当的指令范围内,并且异常的类型是 catch 子句处理的异常的类的同类型或者子类,则异常与 catch 子句匹配。如果找到一个匹配的 catch 子句,则系统转移到指定的处理器。如果没有找到处理器,则该进程重复进行,直到找完当前方法的全部嵌套的 catch 子句。

列表中的 catch 子句的顺序很重要。Java 虚拟机的执行在第一个匹配的 catch 子句处继续。因为 Java 代码是结构化的,所以总是可能把一个方法的所有异常处理器排成一个单个的列表。对任何可能的程序记数器值,可以搜索这个列表以寻找合适的异常处理器——即既包含程序记数器值又处理被抛出的异常的、最里面的异常处理器。

如果没有匹配的 catch 子句,则称当前方法有一个未捕捉的异常。调用者,即调用该方法的方法,的执行状态被恢复。异常继续传播,就好象异常发生在调用者的调用这个实际引起异常的方法的指令处。

Java 通过它的 try-finally 和 try-catch-finally 语句支持更复杂的异常处理形式。在这种形式中,即使没有找到匹配的 catch 子句,finally 语句也被执行。Java 虚拟机支持这些形式的实现的方法在第七章“为 Java 虚拟机编译”中讨论。

### 3.10 class 文件格式

Java 虚拟机执行的编译后代码存储在一个具有平台无关格式的二进制文件中,这种格式是 class 文件格式。给定了 Java 虚拟机的目标,该文件格式的定义与文件格式的其他组件具有同样的重要性。class 文件格式精确地定义了这种文件的内容,包括诸如在平台指定的目标文件格式中可能是想当然的字节顺序的细节。

第四章“class 文件格式”覆盖了 class 文件格式的细节。

### 3.11 指令集概述

一个 Java 虚拟机指令由指定要进行的操作的一个字节的操作码(opcode)后面跟零个或多个提供该操作使用的参数或者数据的操作数(operands)组成。许多指令没有操作数,只由一个操作码组成。

忽略异常,Java 虚拟机执行的内层循环实际上是:

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

附加操作数的个数和大小由操作码决定。如果一个附加操作数的大小大于一个字节,则它被按照 big-endian 顺序存储——高粒字节在前。例如,一个对局部变量的无符号16位索引被作为两个无符号字节 byte1 和 byte2 存储,这样它的值是:

$$(\text{byte1} \ll 8) | \text{byte2}$$

字节码指令流只是单字节对齐的。两个例外是 `tableswitch` 和 `lookupswitch` 指令,它们要求对它们的一些操作数强制内部4字节边界对齐。

把 Java 虚拟机的操作码限制为一个字节的决定和在编译后代码中放弃数据对齐反映了对紧凑性的有意的偏好,在无经验的实现中可能以某些性能损失为代价。单字节的操作码排除了可能提高 Java 虚拟机仿真器的性能的某些实现技术,并限制了指令集的大小。不假定数据对齐意味着在许多机器上对大于一个字节的直接数据在运行期必须从字节构造。

### 3.11.1 类型和 Java 虚拟机

Java 虚拟机指令集中的绝大多数指令对它们进行的操作的有关类型信息编码。例如, `iload` 指令把一个必须是 `int` 的局部变量的内容装载到操作数栈。`float` 指令对一个 `float` 值做同样的事。这两个指令可以具有相同的实现,但具有不同的操作码。

对大多数有类型的指令,指令类型由操作码助记符中的一个字母显式地表示:`i` 对 `int` 操作;`l` 对 `long`;`s` 对 `short`;`b` 对 `byte`;`c` 对 `char`;`f` 对 `float`;`d` 对 `double`;`a` 对 `reference`。其类型是确定的那些指令在它们的助记符中没有类型字母。例如, `arraylength` 总是操作在一个数组对象上。某些指令,例如 `goto`,一个无条件控制转移,不能操作在有类型的操作数上。

给定了 Java 虚拟机的单字节操作码大小,将类型编码到操作码中,对它的指令集的设计施加了压力。如果每个有类型的指令支持所有的 Java 虚拟机的运行期数据类型,则指令将比一个字节能够表示的要多。作为替换,Java 虚拟机的指令集提供了对某些操作的缩减层次的类型支持。在需要时可以使用单独的指令在不支持的和支持的数据类型之间转换。

表3.1总结了 Java 虚拟机的指令集中支持的类型。只列出了多重类型的指令。一个带有类型信息的特定指令通过用类型栏中的字母替换操作码栏中的 T 来建立。如果某些指令模板和类型的类型栏是空的,则不存在支持那种类型的操作的指令。例如,对类型 `int` 有一个装载指令 `iload`,但是类型 `byte` 没有装载指令。

注意表3.1中的绝大多数指令没有整型 `byte`、`char` 和 `short` 的形式。在向它的局部变量或者操作数栈写数时,Java 虚拟机内部地符号扩展类型 `byte` 和 `short` 的值到类型 `int`,并用零扩展类型 `char` 的值到类型 `int`。这样,类型 `byte`、`char` 和 `short` 的值上的绝大多数操作通过操作在类型 `int` 的值上的指令正确地进行。如在 § 3.2.4 中所讲的,Java 虚拟机也特殊地处理 Java 类型 `boolean` 的值。

表3-1 Java 虚拟机指令集中的类型支持

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	float	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	faub	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_TempOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

Java 存储类型与 Java 虚拟机的计算类型之间的映射总结在表3. 2中：

表3-2 存储类型和计算类型

Java(Storage)Type	Size in Byte	Computational Type
byte	8	int
char	16	int
short	16	int
int	32	int
long	64	long
float	32	float
double	64	double

对这个映射的例外情况是数组。类型 boolean、byte、char 和 short 的数组可以直接由 Java

虚拟机表示。类型 byte、char 和 short 的数组用这些类型的特定指令访问。类型 boolean 的数组用 byte 数组指令访问。

本章的剩余部分概述 Java 虚拟机指令集。

### 3.11.2 装载和存储指令

装载和存储指令在 Java 虚拟机的局部变量和操作数栈之间传递值：

- 把一个局部变量装载到操作数栈: iload, iload\_<n>, lload, lload\_<n>, fload, fload\_<n>, dload, dload\_<n>, aload, aload\_<n>。
- 把一个值从操作数栈存储到局部变量: istore, istore\_<n>, lstore, lstore\_<n>, fstore, fstore\_<n>, dstore, dstore\_<n>, astore, astore\_<n>。
- 把一个常数装载到操作数栈: bipush, sipush, ldc, ldc\_w, ldc2\_w, aconst\_null, iconst\_m1, iconst\_<i>, lconst\_<l>, fconst\_<f>, dconst\_<d>。
- 同更宽的索引获得对更多局部变量的访问,或者对更大的立即操作数的访问: wide。

访问对象的域或者数组元素的指令也在操作数栈(§ 3.6.2)传递数据。

上面展示的带有尖括号中的尾字母的指令助记符(例如 iload\_<n>)表示指令族(iload\_<n>的成员为 iload-0, iload-1, iload-2 和 iload-3)。这样的指令族是一个操作数的附加一般指令(iload)的规范指令。对规范的指令,操作数是隐式的,不需要存储或获得。其他语义是相同的(iload-0 表示与 iload 操作数为0时一样)。尖括号中的字母规范该指令族的隐式操作数的类型:<n>对自然数;<i>对 int;<l>对 long;<f>对 float;<d>对 double。在许多情况下类型 int 的形式用于进行类型 byte、char 和 short(§ 3.11.1)的值上的操作。

指令族的这个记号在整个《Java 语言规范》一书中使用。

### 3.11.3 运算指令

运算指令计算一个通常是操作数栈上的两个值的函数的结果,把结果压回到操作数栈。主要有两类运算指令,操作在整数值上的和操作在浮点值上的指令。在每一种中,运算指令被规范为 Java 虚拟机数值类型。没有直接支持 byte、short 和 char 类型(§ 3.11.1)上的整数运算指令,这些操作通过操作在类型 int 上的指令处理。整数和浮点指令在溢出、下溢和被零除的行为上也不同。运算指令如下:

- 加:iadd, ladd, fadd, dadd。
- 减:isub, lsub, fsub, dsub。
- 乘:imul, lmul, fmul, dmul。
- 除:idiv, ldiv, fdiv, ddiv。
- 余数:irem, lrem, frem, drem。
- 取负: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。

- 局部变量递增:iinc。

整数和浮点值(§ 2.4.2, § 2.4.3)上的 Java 操作符的语义由 Java 虚拟机的指令集的语义直接支持。

Java 虚拟机在整数数据类型上的操作中不指示溢出或者下溢。仅有的能够抛出异常的整数操作是整数除指令(idiv 和 ldiv)和整数余数指令(irem 和 lrem),如果除数是零则它们抛出 ArithmeticException。

浮点数上的 Java 虚拟机操作确切地按照 IEEE 754 中规范的进行。特别地,Java 虚拟机要求完整地支持 IEEE 754 非规格化(denormalized)浮点数和逐渐下溢(gradual underflow),这使得易于检验特定数值算法的合适属性。

Java 虚拟机要求浮点运算的每个符点操作符把它的浮点结果舍入到结果精度。不精确的(inexact)结果必须舍入到最接近无穷精确结果的,可表示的值;如果两个最接近的可表示值同样接近,则选择带有最低位零的结果。这是 IEEE 754 标准的缺省舍入模式,称为最接近舍入(round-to-nearest)。

Java 虚拟机在把浮点值转换成整数时使用向零舍入(round-towards-zero)。这使数字被截断,表示该操作数值的分数部分有效数字的任何位被抛弃。向零舍入选择该类型的最接近,并且不大于无穷精确结果的数量的值作为它的结果。

Java 虚拟机的浮点操作符不产生异常。溢出的操作产生一个有符号无穷,下溢的操作产生一个有符号零,没有数学定义结果的操作产生 NaN。所有以 NaN 为一个操作数的数值操作产生 NaN 作为结果。

### 3.11.4 类型转换指令

类型转换指令允许 Java 虚拟机数值类型之间的转换。它们可用于在用户代码中实现显式转换,或者减轻 Java 虚拟机指令集中正交性的缺乏。

Java 虚拟机直接支持以下放宽数值转换,是 Java 的放宽基本转换(§ 2.6.2)的子集:

- int 到 long、float 或者 double;
- long 到 float 或者 double;
- float 到 double。

放宽数值转换指令是 i2l、i2f、i2d、l2f、l2d 和 f2d。这些指令的助记符被直观地给予有类型的指令的命名的转换,并且双关地使用 2 表示“到”。例如,i2d 指令把一个 int 值转换成一个 double。放宽数值转换不丢失一个数值的全部数量的有关信息。实际上,从 int 类型到 long 类型和从 float 到 double 的放宽转换不丢失任何信息,数值被确切地保留下。int 或者 long 值到 float,或者 long 值到 double 的转换可能会丢失精度(precision),即可能丢失值的某些最低位。结果浮点值是使用 IEEE 754 最接近舍入模式把整数值正确地舍入的结果。

按照这个规则,从 int 到 long 的放宽数值转换只是用符号扩展该 int 值的二进制补码表示以填充更宽的格式。从 char 到整型的放宽数值转换用零扩展该 char 值的表示以填充更宽的格式。

尽管可能发生精度丢失,放宽数值转换从不引起运行期异常。

注意,不存在从整型 byte、char 和 short 到类型 int 的放宽数值转换。如 § 3.11.1 中所述,类型 byte、char 和 short 的值被内部地放宽成类型 int,使这些转换是隐式的。

Java 虚拟机也直接支持以下缩窄数值转换——Java 的缩窄基本转换(§ 2.6.3)的一个子集:

- int 到 byte、short 或者 char;
- long 到 int;
- float 到 int 或者 long;
- double 到 int、long 或者 float。

缩窄数值转换指令是 i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l 和 d2f。缩窄数值转换可能使结果值的符号不同,或者数量的数量级不同,或者两者都不同;从而它们可能丢失精度。

int 或者 long 到一个整型 T 的缩窄数值转换只是抛弃除了 N 个最低位的所有位,这里 N 是用于表示类型 T 的位的个数。这可能使结果值与输入值不具有相同的符号。

在浮点值到整型 T 的缩窄数值转换中,这里 T 是 int 或者 long,浮点值按照下述转换类型 T:

- 如果浮点值是 NaN,则转换的结果是一个 int 或者 long 0。
- 否则,如果浮点值大于或等于类型 T 可表示的最小值并且小于或等于类型 T 可表示的最大值,则浮点值用 IEEE 754 向零舍入模式向零舍入到整数值 V。这有两种情况:
  - 如果 T 是 long 并且这个整数值可被作为一个 long 表示,则结果是 long 值 V。
  - 如果 T 是类型 int 并且这个整数值可被作为一个 int 表示,则结果是 int 值 V。
- 否则,或者:
  - 值一定是太小(一个大数量的负值或者负无穷),结果是类型 int 或者 long 可表示的最小值。或者——
  - 值一定是太大(一个大数量的正值或者正无穷),结果是类型 int 或者 long 可表示的最大值。

从 double 到 float 的缩窄数值转换按照 IEEE 754 进行。结果是用 IEEE 754 最接近舍入模式的正确舍入。太小则不能作为一个 float 表示的值而被转换成类型 float 的正或者负零;太大则不能作为一个 float 表示的值而被转换成正或者负无穷。double NaN 总是转换成 float NaN。

尽管可能发生溢出、下溢或者精度丢失,数值类型间的缩窄转换从不引起运行期异常。

### 3.11.5 对象创建和操纵

尽管类实例和数组都是对象,但是 Java 虚拟机用不同的指令集创建和操纵类实例与数组:

- 创建一个新的类实例:new。
- 创建一个新的数组:newarray, anewarray, multianewarray。
- 访问类的域(static 域,称为类变量)和类实例的域(非 static 域,称为实例变量):getfield, putfield, getstatic, putstatic。

- 把一个数组成分装载到操作数栈: `baload`, `caload`, `saload`, `iaload`, `laload`, `faload`, `daload`, `aaload`。
- 把一个值从操作数栈存储成数组成分: `bastore`, `castore`, `sastore`, `iastore`, `lastore`, `fastore`, `dastore`, `aastore`。
- 获取数组的长度: `arraylength`。
- 检查类实例或者数组的属性: `instanceof`, `checkcast`。

### 3.11.6 操作数栈管理指令

为直接操纵操作数栈提供了一些指令:

`pop`, `pop2`, `dup`, `dup2`, `dup_x1`, `dup2_x1`, `dup_x2`, `dup2_x2`, `swap`.

### 3.11.7 控制转移指令

控制转移指令有条件地或者无条件地使 Java 虚拟机从一条不是控制转移指令后面的指令继续执行。它们是:

- 有条件转移: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`, `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, `dcmpg`。
- 复合条件转移: `tableswitch`, `lookupswitch`。
- 无条件转移: `goto`, `goto_w`, `jsr`, `jsr_w`, `ret`。

Java 虚拟机对 `int`、`long`、`float`、`double` 和 `reference` 类型数据比较的有条件转移有不同的指令集。对 `byte`、`char` 和 `short` 类型数据的比较用 `int` 比较指令进行(§ 3.11.1)。由于这个对 `int` 比较的增加的强调, Java 虚拟机对类型 `int` 比对别的类型包括了更大的有条件转移指令的补充。Java 虚拟机具有检查 `null` 引用的不同的条件转移指令, 因此不需要为 `null`(§ 3.3)指定一个具体的值。

所有的 `int` 和 `long` 有条件控制转移指令进行有符号比较。浮点比较按照 IEEE 754 进行。

### 3.11.8 方法调用和返回指令

四条指令调用方法:

- 调用对象的实例方法, 调度对象的(虚)类型: `invokevirtual`。这是 Java 中的正常方法调度。
- 调用由接口实现的方法, 搜索由特定的运行期对象实现的方法以寻找合适的方法: `invokeinterface`。
- 调用需要特殊处理的实例方法, 即实例初始化方法`<init>`、`private` 方法或者超类方法: `invokespecial`。
- 调用命名的类中的类(`static`)方法: `invokestatic`。

方法返回指令, 由返回类型区分, 是 `ireturn`(用于返回类型 `byte`、`char`、`short` 或者 `int` 的

值),lreturn,freturn,dreturn 和 areturn。另外,return 指令用于从声明为 void 的方法返回。

### 3.11.9 抛出和处理异常

异常用 athrow 指令有计划地抛出。如果它们检测到一个不正常的条件,各种 Java 虚拟机指令也可能抛出异常。

### 3.11.10 实现 finally

finally 关键字的实现使用 jsr、jsr\_w 和 ret 指令。参见 4.9.6 节“异常和 finally”和 7.13 节“编译 finally”。

### 3.11.11 同步

Java 虚拟机以不同的方式用单个的机制(监视器)支持方法和块层次的同步。同步方法被作为方法调用和返回的一部分处理(参见 3.11.8 节“方法调用和返回指令”)。但是,代码块的同步在指令集中有显式的支持:monitoreenter, monitorexit。

## 3.12 公共设计,私有实现

到现在为止本书勾勒了 Java 虚拟机的公共概念:class 文件格式和指令集。这些组件对 Java 虚拟机的平台和实现无关性非常重要。实现者可以把它们看作在两个平台间安全地交流程序段的方式,而不是要确切地遵循的蓝图。

理解公共设计和私有实现的界限是很重要的。Java 虚拟机必须能够读 class 文件,并且必须确切地实现其中的 Java 虚拟机代码的语义。这样做的一个方法是把本文档作为一个规范并且实际地实现这个规范。但是实现者在本规范的约束之内修改和优化实现也是相当可行和相当好的。只要 class 文件格式可以读,并且它的代码的语义得到维护,实现者可以用任何方式实现这些语义。怎样才算“不出格”是实现者的事,只要正确的外部接口被仔细地维护。<sup>[2]</sup>

实现者可以使用这些灵活性定制高性能,低存储器占用,或者可移植性好的 Java 虚拟机实现。哪一个在给定的实现中重要取决于实现的目标。实现选择的范围包括如下:

- 在链接期(§ 2.16.3)检验 Java 虚拟机代码的属性,以在保证代码是安全的并且 Java 语言的语义被保护的同时减少运行期检查的需求(Sun 的 class 文件检验器即这样做。参见 4.9 节“class 文件的检验”。)
- 在装载期或者执行中把 Java 虚拟机的代码翻译成另一种虚拟机的指令集(第九章的主题,“优化”)。
- 在装载期或者执行中把 Java 虚拟机代码翻译成主机 CPU 的自身的指令集。(有时称为刚好及时(just-in-time)或者 JIT 代码生成)。

精确定义的虚拟机和目标文件格式的存在并不显著地限制实现者的创造力。Java 虚拟机被设计成支持多种不同的实现,在保持实现间的兼容性的同时提供新的有趣的解决方法。

[2] 有一些例外:调试器和 JIT 代码生成器可能要求访问正常地认为“不出格”的 Java 虚拟机元素。Sun 正在与别的 Java 虚拟机实现者和工具编写者合作以使这些工具所使用的 Java 虚拟机的接口标准化。

## 第四章 class 文件格式

本章描述 Java 虚拟机 class 文件格式。每个 class 文件包含一个或者是类或者是接口的 Java 类型。服从规范的 Java 虚拟机实现必须能够处理符号本书提供的规范的所有 class 文件。

class 文件由一个8位字节流组成。所有的16位、32位和64位数据分别通过读入2个、4个和8个连续的8位字节构造。多字节数据项总是按照 big-endian 顺序存放,即高位字节在前。在 Java 中,这种格式由接口 `java.io.DataInput` 与 `java.io.DataOutput` 和类,例如 `java.io.DataInputStream` 与 `java.io.DataOutputStream` 支持。

本章定义它自己的表示 Java class 文件数据的数据类型集:类型  $u_1$ 、 $u_2$  和  $u_4$ ,分别表示一个无符号的1,2或者4字节数量。在 Java 中,这些类型可以通过方法例如接口 `java.io.DataInput` 的 `readUnsignedByte`,`readUnsignedShort`,和 `readInt` 来读取。

Java class 文件格式用类似于 C 结构的记号编写的伪结构表示。为了避免与 Java 虚拟机的类和类实例的域混淆,描述 Java class 文件格式的结构的内容称为项(items)。与 C 结构的域不同,连续的项在 Java class 文件中顺序存储,不进行填充或者对齐。

可变大小的表(table),由可变大小的项组成,用于几种 class 文件结构中。尽管我们将使用类似于 C 的数组语法引用表项,但是表是可变大小的结构流的事实意味着不可能直接把表的索引翻译成表的字节偏移量。

在我们把一个数据结构称为数组的地方,它实际上就是一个数组。

### 4.1 ClassFile

一个 class 文件包含一个单个的 ClassFile 结构:

```
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];
}
```

ClassFile 结构中的项如下：

#### 【magic】

magic 项提供标识 class 文件格式的幻数,它具有值0xCAFEBABE。

#### 【minor\_version,major\_version】

minor\_version 和 major\_version 项的值是产生该 class 文件的编译器的副和主版本号。Java 虚拟机的一个实现一般支持具有一个给定的主版本号,副版本号从0到某个特定的 minor\_version 的 class 文件。

如果 Java 虚拟机的一个实现支持某个范围的副版本号,并且遇到一个具有相同主版本号但更高的副版本号的 class 文件,则该 Java 虚拟机不应试图运行新的代码。但是,除非主版本号不同,实现一个可以运行直到并且包括这个新的代码的副版本号的代码的新的 Java 虚拟机是可行的。Java 虚拟机不应试图运行具有不同主版本号的代码。主版本号的改变指示一个主要的不兼容的改变,这种改变要求一个根本不同的 Java 虚拟机。

在本书记录的 Sun 的 Java 开发者工具(JDK)1.0.2 版中,major\_version 的值是 45,minor\_version 的值是 3。只有 Sun 可以定义新的 class 文件版本号的意义。

#### 【constant\_pool\_count】

constant\_pool\_count 项的值必须大于零。它给出 class 文件中 constant\_pool 表中表项的个数,包括索引为零的 constant\_pool 表项,但是该表项不出现在 class 文件的 constant\_pool 表中。如果一个 constant\_pool 索引大于零并且小于 constant\_pool\_count,则认为它是有效的。

#### 【constant\_pool []】

constant\_pool 是一个表示在 ClassFile 结构及其子结构中引用的各种字符串常数,类名,域名,和其他常数的变长结构(§ 4.4)的表。

constant\_pool 表的第一个表项,constant\_pool[0],被保留为 Java 虚拟机实现内部使用。该表项不出现在 class 文件中。class 文件中的第一个表项是 constant\_pool[1]。

下标从1到 constant\_pool\_count - 1 的每一个 constant\_pool 表项是一个变长结构(§ 4.4),其格式由它的第一个“标记”字节指示。

#### 【access\_flags】

access\_flags 项的值是类和接口声明中使用的修饰符的掩码.access\_flags 修饰符显示在表4.1中。

接口通过它的被设置的 ACC\_INTERFACE 标志区分。如果 ACC\_INTERFACE 没有设置,则该 class 文件定义了一个类,而不是一个接口。

接口只能使用表4.1中指示的由接口使用的标志。类只能使用表4.1中指示的由类使用的标志。接口隐式地是 abstract(§ 2.13.1),它的 ACC\_ABSTRACT 标志必须被设置,接口不能是 final;否则它的实现永远不能完成(§ 2.13.1),因此它的 ACC\_FINAL 标志不能设置。

标志 ACC\_FINAL 和 ACC\_ABSTRACT 对一个类不能都设置,这样的类的实现永远不

能完成(§ 2.8.2)。

表4-1 类访问和修饰符标志

标志名称	值	意义	使用者
ACC_PUBLIC	0x0001	是 public, 可从它的包外访问。	类, 接口
ACC_FINAL	0x0010	是 Final, 不允许有子类。	类
ACC_SUPER	0x0020	用 invokespecial 特殊地处理超类方法	类, 接口
ACC_INTERFACE	0x0200	是一个接口。	接口
ACC_ABSTRACT	0x0400	是 abstract, 不能被实例化。	类, 接口

ACC\_SUPER 标志的设置指示对其 invokespecial 指令具有两种语义的 Java 虚拟机明确表示其语义, 它的存在是为了对 Sun 的旧的 Java 编译器所编译的代码向后兼容。Java 虚拟机的所有新的实现都应当实现在第六章“Java 虚拟机指令集”中记录的 invokespecial 的语义。所有新的对 Java 虚拟机指令集的编译器都应当设置 ACC\_SUPER 标志。Sun 的旧编译器产生的 ClassFile 标志没有设置 ACC\_SUPER。如果该标志被设置, 则 Sun 的旧的 Java 虚拟机实现忽略它。

#### 【this\_class】

this\_class 项的值必须是对 constant\_pool 表的一个有效的索引。该索引处的 constant\_pool 项必须是一个 CONSTANT\_Class\_info(§ 4.4.1) 结构, 它表示该 Class 文件定义的类或者接口。

#### 【super\_class】

对于一个类, super\_class 项的值必须为零, 或者是对 constant\_pool 表的一个有效索引。如果 super\_class 项的值不是零, 则该索引处的 constant\_pool 表项必须是表示该 class 文件定义的类的超类的 CONSTANT\_Class\_info(§ 4.4.1) 结构。这个超类和它的任何超类都不能是一个 final 类。

如果 super\_class 的值是零, 则该 class 文件必须表示类 java.lang.Object, 它是唯一的没有超类的类或者接口。

对于一个接口, super\_class 的值必须总是对 constant\_pool 表的一个有效的索引。该索引处的 constant\_pool 表项必须是表示类 java.lang.Object 的一个 CONSTANT\_Class\_info 结构。

#### 【interfaces\_count】

interfaces\_count 项的值给出该类或者接口类型的直接超接口的个数。

#### 【interfaces []】

interfaces 数组中的每个值必须是一个对 constant\_pool 表的有效索引。interfaces[i] (这里  $0 \leq i < \text{interfaces\_count}$ ) 的每个值处的 constant\_pool 表项必须是表示该类或者接口类型的一个直接超接口的 CONSTANT\_Class\_info(§ 4.4.1) 结构, 按照该类型的源代码中给定的从左到右的顺序。

#### 【fields\_count】

fields\_count 项的值给出 fields 表中的 field\_info 结构的个数。field\_info(§ 4.5) 结构表

示该类或者接口类型声明的类变量和实例变量的全部域。

#### 【fields []】

fields 表中的每个值必须是一个给出该类或者接口类型中的一个域的完整描述的变长 field\_info(§ 4.6)结构。fields 表只包括该类或者接口声明的那些域。它不包括表示从超类或者超接口中继承的域的项。

#### 【methods\_count】

methods\_count 项的值给出 methods 表中 method\_info 结构的个数。

#### 【methods []】

methods 表中的每个值必须是给出该类或者接口中的一个方法的 Java 虚拟机代码的完整描述的一个变长 method\_info(§ 4.6)结构。

method\_info 结构表示该类或者接口类型声明的实例方法和类(static)方法的全部方法。methods 表只包括该类显式地声明的那些方法。接口只有单个的方法<clinit>, 即接口初始化方法(§ 3.8)。methods 表不包括表示从超类或者超接口继承的方法的项。

#### 【attributes\_count】

attributes\_count 项的值给出该类的 attributes 表中(§ 4.7)的个数。

#### 【attributes []】

attributes 表的每个值必须是一个变长属性结构。一个 ClassFile 结构可以有与它相关联的任意个数的属性(§ 4.7)。

本规范为 ClassFile 结构的 attributes 表定义的唯一属性是 SourceFile 属性(§ 4.7.2)。

要求 Java 虚拟机实现暗中忽略 ClassFile 结构的 attributes 表中的它不识别的任何或者全部属性。不允许本规范中未定义的属性影响 class 文件的语义, 它们只能提供附加的描述性信息(§ 4.7.1)。

## 4.2 完整限定类名称的内部形式

出现在 class 文件结构中的类名总是用完整限定形式(§ 2.7.9)表示。这些类名总是作为 CONSTANT\_Utf8\_info(§ 4.4.7)结构表示, 并且从那些以类名作为它们的描述符的一部分的 CONSTANT\_NameAndType\_info(§ 4.4.6)结构引用, 以及从所有的 CONSTANT\_Class\_info(§ 4.4.1)结构引用。

由于历史原因, 出现在 class 文件结构中的完整限定类名的确切语法与§ 2.7.9中记录的熟悉的 Java 完整限定类名不同。在内部形式中, 通常分开组成完整限定名称的标识符(§ 2.2)的 ASCII 句点(‘.’)由 ASCII 向前斜杠(‘/’))代替。例如, 类 Thread 的正常的完整限定名称是 java.lang.Thread。在 class 文件中的描述符中使用的形式中, 对类 Thread 的名称的引用用一个表示字符串“java/lang/Thread”的 CONSTANT\_Utf8\_info 结构实现。

## 4.3 描述符

描述符是一个表示域或者方法的类型的字符串。

#### 4.3.1 语法规记号

描述符用一种语法规规范。这种语法是一组描述字符序列如何形成语法上正确的，各种类型的描述符的产物。这种语法的终止符号用黑体定宽字体(**bold fixed-width**)显示，非终止符号用斜体(*italic*)类型显示。非终止的定义由被定义的非终止的名称后跟随一个分号引入。非终止的一个或者多个可替换的右手边跟随在连续的行后面。一个产物的右手边上的，后面跟随一个星号(\*)的非终止符号表示从那个非终止产生的零，或者更可能地，不同的值，不间断地附加在后面。

#### 4.3.2 域描述符

域描述符(field descriptor)表示一个类或者实例变量的类型。它是由语法产生的一列字符：

- 域描述符：

域类型

- 成分类型：

域类型

- 域类型：

根类型

对象类型

数组类型

根类型：

B

C

D

F

I

J

S

Z

- 对象类型：

L <classname>;

- 数组类型：

[ *ComponentType*

根类型的字符、对象类型的 L 与；以及数组类型的[都是 ASCII 字符。*<classname>* 表示一个完整限定类名，例如 `java.lang.Thread`。由于历史原因，它在 class 文件中用一种修改的内部形式(§ 4.2)存储。

域类型的意义如下：

B	byte	有符号字节
C	char	字符

D	double	双精度 IEEE 754 浮点数
F	float	单精度 IEEE 754 浮点数
I	int	整数
J	long	长整数
L<classname>;	...	类实例
S	short	有符号 short
Z	boolean	true 或者 false
[	...	一个数组维

例如,一个 int 实例变量的描述符只是 I,一个类型为 Object 的实例变量的描述符是 Ljava/lang/Object。注意使用了类 Object 的完整限定类名的内部形式。多维 double 数组

double d[][][];

的实例变量的描述符是

[ [ [ D

#### 4.3.3 方法描述符

参数描述符(parameter descriptor)表示传递给方法的一个参数:

·参数描述符:域类型

方法描述符(method descriptor)表示方法的参数和它的返回值:

·方法描述符:(参数描述符 \*)返回描述符

返回描述符(return descriptor)表示从方法返回的值。它是由语法产生的一列字符:

·返回描述符:

域类型

V

字符 V 指示该方法不返回值(它的返回类型是 void)。否则,描述符指示返回值的类型。

一个有效的 Java 方法描述符必须表示255个或者更少的字的方法参数,这里这个限制包括实例方法调用的情况下 this 的字。这个限制作用在方法参数的字的个数上,而不是参数自身的个数上。类型 long 和 double 的参数各占两个字。

例如,方法

Object mymethod(int i, double d, Thread t) 的方法描述符是

(IDLjava/lang/Thread;)Ljava/lang/Object;

注意在方法描述符中使用 Thread 和 Object 的完整限定类名的内部形式。

mymethod 的方法描述符不论 mymethod 是 static 还是一个实例方法,都是相同的。尽管实例方法除了它自己的参数,还传递给它 this,即对当前类实例的一个引用,但是这个事实不反映在方法描述符中(对 this 的引用不传递给一个 static 方法)。对 this 的引用由用于调用实例的方法的 Java 虚拟机的方法调用指令隐式地传递。

#### 4.4 常数池

所有的 constant\_pool 表项具有下述一般格式:

```

cp_info {
    u1 tag;
    u1 info[];
}

```

constant\_pool 表中的每一项必须以指示 cp\_info 表项的种类的单字节标记开始。info 数组的内容随 tag 的值变化。有效的标记和它们的值列在表4-2中。每个标记字节后面必须跟随两个或者更多的给出该特定常数有关信息的字节。附加信息的格式随标记值变化。

表4-2 常数池标记

常数类型	值
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

#### 4.4.1 CONSTANT\_Class

CONSTANT\_Class\_info 结构用于表示类或者接口：

```

CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}

```

CONSTANT\_Class\_info 结构的项如下：

- tag  
tag 项具有值 CONSTANT\_Class(7)。
- name\_index

name\_index 项的值必须是对 constant\_pool 表的一个有效索引。该索引处的 Constant\_pool 表项必须是表示已被转换成 class 文件内部形式(§ 4.2)的有效的完整限定 Java 类名(§ 2.8.1)的一个 CONSTANT\_Utf8\_info(§ 4.4.7)结构。

因为数组是对象,所以操作码 anewarray 和 multianewarray 能够通过 constant\_pool 表中的 CONSTANT\_Class\_info(§ 4.4.1)结构引用数组“类”。这种情况下,类的名称是数组类型的描述符。例如,表示一个二维 int 数组类型

int [] []

的类名是

[ [ I

表示类 Thread 类组类型

Thread []

的类名是

[ Ljava.lang.Thread;

一个有效的 Java 数组类型描述符必须有 255 个或者更少的数组维数。

#### 4.4.2 CONSTANT\_Fieldref, CONSTANT\_Methodref, 和 CONSTANT\_InterfaceMethodref

域、方法和接口方法由相似的结构表示：

```
CONSTANT-Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT-Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT-InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

这些结构的项如下：

- tag

CONSTANT\_Fieldref\_info 结构的 tag 项具有值 CONSTANT\_Fieldref(9)。

CONSTANT\_Methodref\_info 结构的 tag 项具有值 CONSTANT\_Methodref(10)。

CONSTANT\_InterfaceMethodref\_info 结构的 tag 项具有值 CONSTANT\_InterfaceMethodref(11)。

- class\_index

class\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是表示包含该域或者方法声明的类或者接口类型的 CONSTANT\_Class\_info(§ 4.4.1) 结构。

CONSTANT\_Fieldref\_info 或者 CONSTANT\_Methodref\_info 结构的 class\_index 项必须是类类型，不能是接口类型。CONSTANT\_InterfaceMethodref\_info 结构的 class\_

index 项必须是声明给定方法的接口类型。

- name\_and\_type\_index

name\_and\_type\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是一个 CONSTANT\_NameAndType(§ 4.4.6) 结构。这个 constant\_pool 表项指示该域或者方法的名称和描述符。

如果 CONSTANT\_Methodref\_info 或者 CONSTANT\_InterfaceMethodref\_info 的方法的名称以'('(''\u003C')开始，则这个名称一定是特殊的内部方法(§ 3.8)之一，或者是<init>，或者是<clinit>。在这种情况下，该方法不能返回值。

#### 4.4.3 CONSTANT\_String

CONSTANT\_String\_info 结构用于表示类型

java.lang.String 的常数对象：

```
CONSTANT_String_info {  
    u1 tag;  
    u2 string_index;  
}
```

CONSTANT\_String\_info 结构的项如下：

- tag

CONSTANT\_String\_info 结构的 tag 项具有值 CONSTANT\_String(8)。

- string\_index

string\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是表示 java.lang.String 对象被初始化成的字符序列的 CONSTANT\_Utf8\_info(§ 4.4.3) 结构。

#### 4.4.4 CONSTANT\_Integer 和 CONSTANT\_Float

CONSTANT\_Integer\_info 和 CONSTANT\_Float\_info 结构表示 4 字节数值(int 和 float)常数：

```
CONSTANT_Integer_info {  
    u1 tag;  
    u4 bytes;  
}  
CONSTANT_Float_info {  
    u1 tag;  
    u4 bytes;  
}
```

这些结构的项如下：

- tag

CONSTANT\_Integer\_info 结构的 tag 项具有值 CONSTANT\_Integer(3)。

CONSTANT\_Float\_info 结构的 tag 项具有值 CONSTANT\_Float(4)。

- bytes

CONSTANT\_Integer\_info 结构的 bytes 项包含该 int 常数的值。值的字节按照 big-endian 顺序存储(高位字节在前)。

CONSTANT\_Float\_info 结构的 bytes 项包含 IEEE 754 浮点“单格式”位布局中的该 float 常数的值。值的字节按照 big-endian 顺序存储(高位字节在前),并首先转换成一个 int 参数:

- 如果参数是 0x7f80 0000, 则 float 值是正无穷。
- 如果参数是 0xff80 0000, 则 float 值是负无穷。
- 如果参数值在范围 0x7f8 00001 到 0x7fff ffff 或者 0xff800001 到 0xffffffff 中, 则浮点值是 NaN。
- 在所有其他情况下, 设 s、e 和 m 是可以由以下计算的三个值;

```
int s = ((bytes >> 31) == 0)? 1:-1;  
int e = ((bytes >> 23) & 0xff);  
int m = (e == 0)?  
    (bytes & 0x7fffff) <<1:  
    (bytes & 0x7fffff) | 0x800000;
```

则 float 值等于数字表达式  $s \cdot m \cdot 2^{e-150}$  的结果。

#### 4.4.5 CONSTANT\_Long 和 CONSTANT\_Double

CONSTANT\_Long\_info 和 CONSTANT\_Double\_info 表示8字节数值(long 和 double)常数:

```
CONSTNAT_Long_info {  
    u1 tag;  
    u4 high_bytes;  
    u4 low_bytes;  
}
```

```
CONSTANT_Double_info {  
    u1 tag;  
    u4 high_bytes;  
    u4 low_bytes;  
}
```

所有的8字节常数在 class 文件的 constant\_pool 表中,以及在 class 文件被读入时构造的常数池的存储器内版本中,占两个表项。如果一个 CONSTANT\_long\_info 或者 CONSTANT\_Double\_info 结构是 constant\_pool 表中索引为 n 的项,则池中下一个有效项的位置在索引 n+2,constant\_pool 索引 n+1 必须被认为是无效的并且不能使用<sup>[1]</sup>

这些结构的项如下:

[1] 回顾一下,使8字节常数占两个常数池表项是一个糟糕的选择。

- **tag**  
CONSTANT\_Long\_info 结构的 tag 项具有值 CONSTANT\_Long(5)。  
CONSTANT\_Double\_info 结构的 tag 项具有值 CONSTANT\_Double(6)
- **high\_bytes,low\_bytes**  
CONSTANT\_Long\_info 结构的无符号的 high\_bytes 和 low\_bytes 项一起包含该 long 常数的值((long)high\_bytes << 32)+low\_bytes, 这里每个 high\_bytes 和 low\_bytes 的字节按照 big-endian 顺序存储(高位字节在前)。CONSTANT\_Double\_info 结构的 high\_bytes 和 low\_bytes 项包含 IEEE 754 浮点“双格式”位布局中的该 double 值。high\_bytes 和 low\_bytes 项被首先转换成一个 long 参数:
  - 如果参数是 0x7f80 0000 0000 0000L, 则 double 值是正无穷。
  - 如果参数是 0xff80 0000 0000 0000L, 则 double 值是负无穷。
  - 如果参数在范围 0x7ff0 0000 0000 0001L 到 0x7fff ffff ffff ffffL 或者 0xffff0 0000 0000 0001L 到 0xffff ffff ffff ffffL 中, 则 double 值是 NaN。
  - 在所有其他情况下, 设 s、e 和 m 是可从参数计算的三个值:
 

```
int s=((bits >> 63)==0)? 1:-1;
int e=(int)((bits >> 52) & 0x7ffL);
long m=(e==0)?
    (bits & 0xffffffffffffL) <<1;
    (bits & 0xfffffffL) | 0x1000000000000000L;
```

 则该浮点值等于数学表达式  $s \cdot m \cdot z^{e-1075}$  的 double 值。

#### 4.4.6 CONSTANT\_NameAndType

CONSTANT\_NameAndType\_info 结构用于表示一个域或者方法, 不指示它属于哪个类或者接口类型:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

CONSTANT\_NameAndType\_Info 结构的项如下:

- **tag**  
CONSTANT\_NameAndType\_info 结构的 tag 项具有值 CONSTANT\_NameAndType(12)。
- **name\_index**  
name\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是表示一个有效的 Java 域名或者方法名(§ 2.7)的 CONSTANT\_Utf8\_info(§ 4.4.7)结构, 其中域名或者方法名以简单(不是完整限定)名称(§ 2.7.1)——即 Java 标识符——存储。
- **descriptor\_index**

`descriptor_index` 项的值必须是一个对 `constant_pool` 表的有效索引。该索引处的 `constant_pool` 表项必须是表示一个有效的 Java 域描述符(§ 4.3.2)或者方法描述符(§ 4.3.3)的 `CONSTANT_Utf8_info`(§ 4.4.7)结构。

#### 4.4.7 CONSTANT\_Utf8

`CONSTANT_Utf8_info` 结构用于表示常数字符串值。

UTF-8字符串被编码使得只包含非 nullASCII 字符的字符序列能够对每个字符只用一个字节表示,但是能够表示直到16位的字符。在范围'\u0001'到'\u007F'中的所有字符用一个单个字节表示:

0	bits 0-7
---	----------

字节中七位数据给出表示的字符的值。null 字符 ('\u0000') 和范围'\u0080'到'\u07ff'中的字符用一对字节 `x` 和 `y` 表示:

x: [1 1 0] bits 6-10	y: [1 0] bits 0-5
----------------------	-------------------

这些字节用值(`(x & 0x1f) << 6`) + (`y & 0x3f`) 表示字符。

范围'\u0800'到'\uFFFF'中的字符用三个字节 `x`、`y` 和 `z` 表示:

x: [1 1 1 0] bits 12-15	y: [1 0] bits 6-11	z: [1 0] bits 0-5
-------------------------	--------------------	-------------------

具有值(`(x & 0xf) << 12`) + (`(y & 0x3f) << 6`) + (`z & 0x3f`) 的字符由这些字节表示。

多字节字符的字节在 class 文件中按照 big-endian 顺序(高位字节在前)存储。

这种格式与“标准”的 UTF-8 格式有两点差异。第一, null 字节(byte)0 用双字节格式而不是单字节格式编码,因此 Java 虚拟机 UTF-8 字符串从不会有嵌入的 null。第二,只使用单字节、双字节和三字节格式。Java 虚拟机不识别更长的 UTF-8 格式。

关于 UTF-8 格式的更多信息,参见 File system Safe UCS Transformation Format(FSS-UTF), X/Open Preliminary Specification, X/Open Company Ltd. , Document Number: P316.

这些信息也出现在 ISO/IEC 10646, AnnexP 中。

`CONSTANT_Utf8.info` 结构是:

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes [length];
}
```

`CONSTANT_Utf8_info` 结构的项如下:

- `tag`

`CONSTANT_Utf8_info` 结构的 `tag` 项具有值 `CONSTANT_Utf8(1)`。

- **length**

**length** 项的值给出该 bytes 数组中字节的个数(不是结果字符串的长度)。CONSTANT\_Utf8\_info 结构中的字符串不是以 null 结束的。

- **bytes[]**

bytes 数组包含字符串的字节。没有字节可以具有值(byte)0 或者 (byte)0xf0-(byte)0xff.

## 4.5 域

每个域由一个变长 field\_info 结构描述。该结构的格式是：

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes [attributes_count];
}
```

field\_info 结构的项如下：

- **access\_flags**

**access\_flags** 项的值是用于描述域的访问权和属性的修饰符的掩码。**access\_flags** 修饰符显示在表4-3中。

表4-3 域访问权和修饰符标志

标志名	值	意义	使用者
ACC_PUBLIC	0x0001	是 public, 可从其包外访问	任何域
ACC_PRIVATE	0x0002	是 private, 只在定义它的类内可用。	类域
ACC_PROTECTED	0x0004	是 protected, 可在子类中访问。	类域
ACC_STATIC	0x0008	是 static	任何域
ACC_FINAL	0x0010	是 final, 初始化后不能再覆盖或赋值。	任何域
ACC_VOLATILE	0x0040	是 volatile, 不能被高速缓存。	类域
ACC_TRANSIENT	0x0080	是 transient, 不被固定对象方式读写。	类域

接口的域只能使用表4.3中指示的可以由任何域使用的标志,类的域可以使用表4-3中的任何标志

**access\_flags** 项的所有未使用的位,包括那些没有在表4-3中分配的位为将来的使用保留。它们应当在生成的 class 文件中设置为零并被 Java 虚拟机的实现忽略。

类域至多可以被设置标志 ACC\_PUBLIC,ACC\_PROTECTED,ACC\_PRIVATE 中的一个(§ 2.7.8)。一个类域不能同时设置 ACC\_FINAL 和 ACC\_VOLATILE(§ 2.9.1)。

每个接口域隐式地是 static 和 final(§ 2.13.4)并且必须对它的 ACC\_STATIC 和

ACC\_FINAL 标志两个都设置。每个接口域隐式地是 public(§ 2.13.4)并且必须使它的 ACC\_PUBLI 标志被设置。

- **name\_index**

name\_index 项的值必须是对 constant\_pool 表的一个有效索引。该索引处的 constant\_pool 表项必须是一个 CONSTANT\_Utf8\_info(§ 4.4.1) 结构，该结构必须表示一个作为简单(不是完整限定)名称(§ 2.7.1)，即，作为 Java 标识符，存储的一个有效的 Java 域名(§ 2.7)。

- **descriptor\_index**

descriptor\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是一个 CONSTANT\_Utf8\_info(§ 4.4.7) 结构，该结构必须表示一个有效的 Java 域描述符(§ 4.3.2)。

- **attributes\_count**

attributes\_count 项的值指示该域的附加属性(§ 4.7)的个数。

- **attributes [ ]**

attributes 表的每个值必须是一个变长属性结构。一个域可以具有任意个数的与它相关联的属性(§ 4.7)。本规范为 field\_info 结构的 attributes 表定义的唯一属性是 ConstantValue 属性(§ 4.7.3)，Java 虚拟机实现必须识别 field\_info 结构的 attributes 表中的 ConstantValue 属性。要求 Java 虚拟机实现暗中忽略 attributes 表中的它不识别的任何或者所有其他属性。不允许本规范中未定义的属性影响 class 文件的语义，它们只能提供附加的描述性信息(§ 4.7.1)

## 4.6 方法

每个方法，以及每个实例初始化方法<init>由一个变长 method\_info 结构描述。该结构具有下述格式：

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

method\_info 结构的项如下：

- **access\_flags**

access\_flags 项的值是用于描述方法或者实例初始化方法(§ 3.8)的访问权和属性的修饰符的掩码。access\_flags 修饰符显示在表4-4中。接口中的方法只能使用表4-4中指示为由任何方法使用的标志。类和实例方法(§ 2.10.3)可以使用表4-4中的任何标志。实例初始化方法(§ 3.8)只能使用 ACC\_PUBLIC、ACC\_PROTECTED 和 ACC\_PRIVATE，access\_flags 项的所有未使用的位，包括那些在表4-4中没有分配的位为将来的

使用保留。它们应当在生成的 class 文件中设置为零并被 Java 虚拟机实现忽略。

表4-4 方法访问权和修饰符标志

标志名	值	意义	使用者
ACC_PUBLIC	0x0001	是 public, 可从其包外访问。	任何方法
ACC_PRIVATE	0x0002	是 private, 只在定义它的类内可用。	类/实例方法
ACC_PROTECTED	0x0004	是 protected, 可在子类中访问。	类/实例方法
ACC_STATIC	0x0008	是 static。	类/实例方法
ACC_FINAL	0x0010	是 final, 不允许覆盖。	类/实例方法
ACC_SYNCHRONIZED	0x0020	是 synchronized, 用监视器锁包装使用。	类/实例方法
ACC_NATIVE	0x0100	是 native, 用非 Java 语言实现。	类/实例方法
ACC_ABSTRACT	0x0400	是 abstract, 未提供实现。	任何方法

对任何方法至多可以设置标志 ACC\_PUBLIC, ACC\_PROTECTED, 和 ACC\_PRIVATE 中的一个。类和实例方法不可以与 ACC\_FINAL, ACC\_NATIVE, 或者 ACC\_SYNCHRONIZED 一起使用 ACC\_ABSTRACT(即 native 和 synchronized 方法需要一个实现)。类或者实例方法不能一起使用 ACC\_PRIVATE 和 ACC\_ABSTRACT(即, private 方法不能被覆盖, 所以这样的方法不能被实现或者使用)。类或者实例方法不能一起使用 ACC\_STATIC 和 ACC\_ABSTRACT(即, static 方法隐式地是 final, 从而不能被覆盖, 因此这样的方法不能被实现或者使用)。

类和接口初始化方法(§ 3.8), 即名称为<clinit>的方法, 被 Java 虚拟机隐式地调用; 它们的 access\_flags 项的值被忽略。

每个接口方法隐式地是 abstract, 因此它的 ACC\_ABSTRACT 标志必须被设置。每个接口方法隐式地是 public (§ 2.13.5)因此它的 ACC\_PUBLIC 标志必须被设置。

- name\_index

name\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是一个 CONSTANT\_Utf8\_info(§ 4.4.7)结构, 该结构表示特殊的内部方法名称(§ 3.8), <init>或者<clinit>之一, 或者一个作为简单(不是完整限定)名称(§ 2.7.1)存储的有效的 Java 方法名称(§ 2.7)。

- descriptor\_index

descriptor\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是一个表示一个有效的 Java 方法描述符(§ 4.3.3)的 CONSTANT\_Utf8\_info(§ 4.4.7)结构。

- attributes\_count

attributes\_count 项的值指示该方法的附加属性(§ 4.7)的个数。

- attributes []

attributes 表的每个值必须是一个变长属性结构。一个方法可以有任意个数的, 可选的, 与它相关联的属性(§ 4.7)。

本规范为 method\_info 结构的 attributes 表定义的仅有的属性是 Code(§ 4.7.4)和 Exceptions(§ 4.7.5)属性。

Java 虚拟机实现必须识别 Code(§ 4.7.4)和 Exceptions(§ 4.7.5)属性。要求 Java 虚拟

机实现暗中忽略 method\_info 结构的 attributes 表中的它不识别的任何或者全部属性。不允许本规范中未定义的属性影响 class 文件的语义,它们只能提供附加的描述性信息(§ 4.7.1)。

## 4.7 属性

属性用于 class 文件格式的 ClassFile(§ 4.1),field\_info(§ 4.5),method\_info(§ 4.6),和 Code\_attribute(§ 4.7.4)结构中。所有的属性具有下述一般格式:

```
attribute_info {  
    u2 attribute_name_index;  
    u4 attribute_length;  
    u1 info[attribute_length];  
}
```

对所有的属性,attribute\_name\_index 必须是一个对该类的常数池的有效无符号16位索引。attribute\_name\_index 处的 constant\_pool 表项必须是表示该属性的名称的一个 CONSTANT\_Utf8(§ 4.4.7)字符串。attribute\_length 项的值按字节指示后面的信息的长度。长度不包括包含 attribute\_name\_index 和 attribute\_length 项的初始的6个字节。

某些属性被预定义作为 class 文件规范的一部分。这些预定义的属性是 SourceFile(§ 4.7.2)、ConstantValue(§ 4.7.3)、Code(§ 4.7.4)、Exceptions(§ 4.7.5)、LineNumberTable(§ 4.7.6)和 LocalVariableTable(§ 4.7.7)属性。在本规范中它们的使用的上下文中,即它们出现的 class 文件结构的 attributes 表中,这些预定义的属性的名称被保留。

在这些预定义的属性中,为了由 Java 虚拟机正确地解释 class 文件、Code、ConstantValue,和 Exceptions 属性必须被 class 文件读入器识别并正确地读入。对剩余的预定义的属性的使用是可选的。class 文件读入器可以使用它们包含的信息,否则必须暗中忽略这些属性。

### 4.7.1 定义和命名新的属性

允许 Java 源代码的编译器定义和发出在 class 文件结构的 attributes 表中包含新的属性的 class 文件。允许 Java 虚拟机的实现识别和使用在 class 文件结构的 attributes 表中发现的新属性。但是,不是作为本 Java 虚拟机规范的一部分定义的所有属性不能影响类或者接口类型的语义。要求 Java 虚拟机实现暗中忽略它们不识别的属性。

例如,允许定义新的属性以支持销售商特定的调试。因为要求 Java 虚拟机实现忽略它们不识别的属性,所以为那种特定的 Java 虚拟机实现使用的 class 文件可以被别的实现使用,即使那些实现不能使用该 class 文件包含的附加调试信息。

特别地,禁止 Java 虚拟机实现仅仅因为一个 class 文件中有一些新的属性而抛出异常或者拒绝使用该类文件。当然,如果给定的 class 文件不包含工具要求的全部属性,则操作在 class 文件上的工具可能不能正确地运行。

两个应该不同的属性,但是碰巧使用相同的属性名称并具有相同的长度,将在识别任一属性的实现上冲突。不是由 Sun 定义的属性必须具有按照由《Java 语言规范》一书中定义的包名称约定选择的名称。例如,由网景定义的一个新的属性可能具有名称“COM. NetScape. new-

attribute”。

Sun 可能在本 class 文件规范的将来版本中定义附加的属性。

#### 4.7.2 SourceFile 属性

SourceFile 属性是 ClassFile(§ 4.1)结构的 attributes 表中的一个可选的定长属性。在一个给定的 classFile 结构的 attributes 表中不能有多于一个的 SourceFile 属性。

SourceFile 属性具有格式

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

SourceFile\_attribute 结构的项如下：

- attribute\_name\_index

attribute\_name\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是表示字符串“SourceFile”的一个 CONSTANT\_Utf8\_info (§ 4.4.7) 结构。

- attribute\_length

SourceFile\_attribute 结构的 attribute\_length 项的值必须是2。

- sourcefile\_index

sourcefile\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的常数池表项必须是表示给出从其编译该 class 文件的源文件的名称的字符串的 CONSTANT\_Utf8\_info (§ 4.4.7) 结构。SourceFile 属性只给出源文件的名称，它从不表示包含该文件的目录的名称，或者该文件的绝对路径名称。例如，SourceFile 属性可能包含文件名称 foo.java，但是不会包含 UNIX 路径名称/home/lindholm/foo.java。

#### 4.7.3 ConstantValue 属性

ConstantValue 属性是一个用于 field\_info(§ 4.5) 结构的 attributes 表中的定长属性。ConstantValue 属性表示一个必须是(显式地或隐式地) static 的常数域的值；即，field\_info 结构的 flags 项中的 ACC\_STATIC 位(表4.3)必须是设置的。该域不需要是 final。在一个给定的 field\_info 结构的 attributes 表中不可以有多于一个的 ConstantValue 属性。由该 field\_info 结构表示的常数域被赋给它的 ConstantValue 属性引用的值作为它的初始化(§ 2.16.4)的一部分。

每个 Java 虚拟机实现必须识别 ConstantValue 属性。

ConstantValue 属性具有格式：

```
ConstantValue_attribute {
```

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

}

ConstantValue\_attribute 的结构的项如下：

- attribute\_name\_index

attribute\_name\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是表示字符串“ConstantValue”的一个 CONSTANT\_Utf8\_info(§ 4.4.7)结构。

- attribute\_length

constantValue\_attribute 结构的 attribute\_length 项的值必须是2。

- constantvalue\_index

constantvalue\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须给出该属性表示的常数值。

该 constant\_pool 表项必须具有与该域适合的类型，显示在表4-5中：

表4-5 常数值属性类型

域类型	表项类型
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer
java.lang.String	CONSTANT_String

#### 4.7.4 Code 属性

Code 属性是用于 method\_info 结构的 attributes 表中的一个变长属性。一个 Code 属性包含一个单个的 Java 方法，实例初始化方法(§ 3.8)，或者类或接口初始化方法(§ 3.8)的 Java 虚拟机指令和辅助信息。每个 Java 虚拟机实现必须识别 code 属性，在每个 method\_info 结构中必须确切地有一个 Code 属性。

Code 属性具有格式

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
```

```

    u2 handler_pc;
    u2 catch_type;
} exception_table[exception_table_length];
u2 attributes_count;
attribute_info attributes[attributes_count];
}

```

Code\_attribute 结构的项如下：

- attribute\_name\_index

attribute\_name\_index 项的值必须是一个对 constant\_pool 表的有效索引。该索引处的 constant\_pool 表项必须是一个表示字符串“Code”的 CONSTANT\_Utf8\_info(§ 4.4.7)结构。

- attribute\_length

attribute\_length 项的值指示该属性的长度,不包括初始的6个字节。

- max\_stack

max\_stack 项的值给出该方法执行中任何点操作数栈上字的最大个数。

- max\_locals

max\_locals 项的值给出该方法使用的局部变量的个数,包括调用时传递给该方法的参数。第一个局部变量的索引是0,单字值的最大局部变量索引是 max\_locals-1,双字值的最大局部变量索引是 max\_locals-2。

- code\_length

code\_length 项的值给出该方法的 code 数组中字节的个数。code\_length 的值必须大于零;code 数组不能是空的。

- code[]

code 数组给出实现该方法的 Java 虚拟机代码的实际字节。

在 code 数组被读入一个可按字节寻址的机器上的存储器中时,如果数组的第一个字节是4字节边界对齐的,则 tableswith 和 lookupswitch 32位偏移量将是4字节对齐的;关于 code 数组对齐的结果的更多信息参考那些指令的描述。

code 数组的内容上的详细约束是扩展的并在一个单独的节(§ 4.8)中给出。

- exception\_table\_length

exception\_table\_length 项的值给出 exception\_table 表中表项的个数。

- exception\_table[]

exception\_table 数组中的每个表项描述 code 数组中的一个异常处理器。每个 exception\_table 表项包含以下项:

- start\_pc,end\_pc

start\_pc 和 end\_pc 这两项的值指示在其中异常处理器是活跃的 code 数组中的范围。start\_pc 的值必须是一个对指令的操作码在 code 数组中的有效索引.end\_pc 的值或者必须是一个对指令的操作码在 code 数组中的有效索引,或者必须等于 code\_length,即该 code 数组的长度.start\_pc 的值必须比 end\_pc 的值小。

`start_pc` 是包含的, `end_pc` 是不包含的; 即异常处理器必须在程序计数器在区间 `[start_pc, end_pc]`<sup>[1]</sup> 中时是活跃的。

- `handler_pc`

`handler_pc` 项的值指示异常处理器的开始。该项的值必须是一个对 `code` 数组的有效索引, 必须是一条指令的操作码的索引, 并且必须小于 `Code_length` 项的值。

- `catch_type`

如果 `catch_type` 项的值不是零, 则它必须是一个对 `constant_pool` 表的有效索引。该索引处的 `constant_pool` 表项必须是一个表示该异常处理器设计捕捉的异常的类的 `CONSTANT_Class_info` (§ 4.4.1) 结构。

这个类必须是类 `Throwable` 或者其子类之一。异常处理器只在被抛出的异常是给定的类或者其子类之一的一个实例时才被调用。如果 `catch_type` 项的值是零, 则对所有的异常调用该异常处理器。这用于实现 `finally` (参见 7.13 节, “编译 `finally`”)。

- `attributes_count`

`attributes_count` 项的值指示 `Code` 属性的属性的个数。

- `attributes []`

`attributes` 表的每个值必须是一个变长属性结构。`Code` 属性可以具有与它相关联的任意个数的可选的属性。

当前, `LineNumberTable` (§ 4.7.6) 和 `Local Variable Table` (§ 4.7.7) 属性, 两者都包含调试信息, 由 `Code` 属性定义和使用。允许 Java 虚拟机实现暗中忽略 `Code` 属性的 `attributes` 表中的任何或全部属性。不允许本规范中未定义的属性影响 `class` 文件的语义, 它们只能提供附加的描述性信息 (§ 4.7.1)。

#### 4.7.5 Exceptions 属性

`Exceptions` 属性是用于 `method_info` (§ 4.8) 结构的 `attributes` 表中的变长属性。`Exceptions` 属性指示一个方法可以抛出哪些受检查异常。在每个 `method_info` 结构中必须确切地有一个 `Exception` 属性。

`Exceptions` 属性具有格式

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

`Exceptions_attribute` 结构的项如下:

- `attribute_name_index`

`attribute_name_index` 项的值必须是一个对 `Constant_pool` 表的一个有效索引。该索引处的 `constant_pool` 表项必须是表示字符串“`Exceptions`”的 `CONSTANT_Utf8_info`

[1] `end_pc` 是不包含的事实是 Java 虚拟机中的一个历史错误: 如果一个方法的 Java 虚拟机代码确切地是 65535 字节长并且以一个单字节长度的指令结束, 则该指令不能被异常处理器保护。

(§ 4.4.7)结构。

- **attribute\_length**

**attribute\_length** 项的值指示属性长度,不包括初始的6个字节。

- **number\_of\_exceptions**

**number\_of\_exceptions** 项的值指示 **exception\_index\_table** 中表项的个数。

- **exception\_index\_table** []

**exception\_index\_table** 数组中的每个非零值必须是一个对 **constant\_pool** 表的有效索引。对每个表项,如果 **exception\_index\_table[i] != 0**,这里  $0 \leq i < \text{number\_of\_exceptions}$ ,则在索引 **exception\_index\_table[i]** 处的 **constant\_pool** 表项必须是一个表示该方法声明抛出的类类型的 **CONSTANT\_Class\_info**(§ 4.4.1)结构。

只有下述三个标准中至少一个被满足,方法才应当抛出一个异常:

- 异常是 **RuntimeException** 或者其子类之一的一个实例。

- 异常是 **Error** 或者其子类之一的一个实例。

- 异常是上面的 **exception\_index\_table** 中指定的异常类,或者它们的子类之一的一个实例。

当前 Java 虚拟机没有加强上述要求,它们只在编译期被加强。Java 语言的将来版本可能在类被检验时要求对 **throws** 子句的更严格的检查。

#### 4.7.6 LineNumberTable 属性

**LineNumberTable** 属性是 **Code**(§ 4.7.4)属性的 **attributes** 表中的一个可选的变长属性。它可由调试器使用以确定 Java 虚拟机 **code** 数组的哪部分对应原始的 Java 源文件中的给定行号。如果 **LineNumberTable** 属性出现在给定的 **Code** 属性的 **attributes** 表中,则它们可以按任何顺序出现。而且,多个 **LineNumberTable** 属性可以一起表示一个 Java 源文件中的一个给定的行;即, **LineNumberTable** 属性不必与源文件的行一一对应<sup>[1]</sup>

**LineNumberTable** 属性具有格式:

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

**LineNumberTable\_attribute** 结构的项如下:

- **attribute\_name\_index**

**attribute\_name\_index** 项的值必须是一个对 **Constant\_pool** 表的有效索引。该索引处的

[1] Sun 的 JDK1.0.2 版中的 java c 编译器能够实际地生成不按行号顺序并且不与源文件行一一对应的 **LineNumberTable** 属性。很不幸,因为我们希望指定与源文件一一对应的,顺序映射的 **LineNumberTable** 属性,但是必须服从向后兼容。

`constant_pool` 表项必须是表示字符串“`LineNumberTable`”的一个 `CONSTANT_Utf8_info`(§ 4.4.7)结构。

- `attribute_length`

`attribute_length` 项的值指示属性的长度,不包括初始的6个字节。

- `line_number_table_length`

`line_number_table_length` 项的值指示  
`line_number_table` 数组中表项的个数。

- `line_number_table[]`

`line_number_table` 数组中的每个表项指示原始的 Java 源文件中的行号在 `code` 数组中  
的一个给定点改变。每个表项必须包含以下项:

- `start_pc`

`start_pc` 项的值必须指示 `code` 数组的索引,该索引处开始原始的 Java 源文件中的新  
行。`start_pc` 的值必须小于 `Code` 属性的 `code_length` 项的值,其中 `LineNumberTable`  
是 `Code` 属性的一个属性。

- `line_number`

`line_number` 项的值必须给出原始的 Java 源文件中的对应行号。

#### 4.7.7 LocalVariableTable 属性

`LocalVariableTable` 属性是 `Code`(§ 4.7.4)属性的一个可选的变长属性。它可由调试器使  
用以在方法的执行中确定一个给定的局部变量的值。如果 `LocalVariableTable` 属性出现在一  
个给定的 `Code` 属性的 `attributes` 表中,则它们可以按任何顺序出现。在 `Code` 属性中每个局部  
变量不能多于一个的 `LocalVariableTable` 属性。

`LocalVariableTable` 属性具有格式

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

`LocalVariableTable_attribute` 结构的项如下:

- `attribute_name_index`

`attribute_name_index` 项的值必须是一个对 `constant_pool` 表的有效索引。该索引处的

`constant_pool` 表项必须是表示字符串“`LocalVariableTable`”的 `CONSTANT_Utf8_info`(§ 4.4.7)结构。

- `attribute_length`

`attribute_length` 项的值指示属性的长度,不包括初始的6个字节。

- `local_variable_table_length`

`local_variable_table_length` 项的值指示。

`local_variable_table` 数组中表项的个数。

- `local_variable_table[]`

`local_variable_table` 数组中的每个表项指示 `code` 数组偏移量的一个范围,在这个范围内局部变量具有一个值。它也指示当前框架的局部变量的索引,在该索引处可以找到那个局部变量。每个表项必须包含以下项:

- `start_pc,length`

给定的局部变量在 `code` 数组的区间 $[start\_pc, start\_pc + length]$ ,即从 `start_pc` 到 `start_pc + length` 的闭区间中的索引处必须具有一个值。

`start_pc` 的值必须是对一条指令的操作码在这个 `Code` 属性的 `code` 数组中的一个有效索引。`start_pc + length` 的值必须或者是对一条指令的操作码在这个 `Code` 属性的 `code` 数组中的一个有效索引,或者是那个 `code` 数组尾之后的第一个索引。

- `name_index,descriptor_index`

`name_index` 项的值必须是 `constant_pool` 表的一个有效索引。该索引处的 `constant_pool` 表项必须包含表示一个以简单名称(§ 2.7.1)存储的有效的 Java 局部变量名称的 `CONSTANT_Utf8_info`(§ 4.4.7)结构。

`descriptor_index` 项的值必须是一个对 `constant_pool` 表的有效索引。该索引处的 `constant_pool` 表项必须包含表示一个 Java 局部变量的有效描述符的 `CONSTANT_Utf8_info`(§ 4.4.7)结构。Java 局部变量描述符与域描述符(§ 4.3.2)具有相同的形式。

- `index`

给定的局部变量在其方法的局部变量中的索引必须是 `index`。如果 `index` 处的局部变量是双字类型(`double` 或者 `long`),则它占据 `index` 和 `index+1`。

## 4.8 对 Java 虚拟机代码的约束

方法、实例初始化方法(§ 3.8)以及类或接口初始化方法(§ 3.8)的 Java 虚拟机代码存储在 class 文件的 `method_info` 结构 `Code` 属性的 `code` 数组中。本节描述与 `Code_attribute` 结构的内容相关的约束。

### 4.8.1 静态约束

class 文件上的静态约束(static constraints)是那些定义文件的良好性的约束。除了 class 文件的 Java 虚拟机代码上的静态约束,这些约束已在前面的节中给出。class 文件中的 Java 虚

拟机代码上的静态约束指定 Java 虚拟机指令必须如何在 code 数组中布局以及单独的指令的操作数必须是什么。

code 数组中的指令上的静态约束如下：

- code 数组不能是空的,因此 code\_length 属性的值不能是0。
- code 数组中的第一条指令的操作码从索引0开始。
- 只有(§ 6.4)中记录的指令的实例可以出现在 code 数组中。使用保留的操作码(§ 6.2)、第九章“一种优化”中记录的\_quick 操作码以及本规范中没有记录的任何操作码的指令的实例不能出现在 code 数组中。
- 对 code 数组中除了最后一条的每条指令,下一条指令的操作码的索引等于当前指令的操作码的索引加上那条指令——包括其所有操作数——的长度。wide 指令类似于为这些目的的任何其他指令处理;指定由一个 wide 指令修饰的操作的操作码被作为那条 wide 指令的一个操作数处理。那个操作码不能被计算直接达到。
- code 数组中的最后一条指令的最后一个字节必须是索引 code\_length-1 处的字节。

code 数组中的指令的操作数上的静态约束如下：

- 每条跳转或转移指令(jsr, jsr\_w, goto, goto\_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonnull, if\_icmpeq, if\_icmpne, if\_icmplt, if\_icmpge, if\_icmpgt, if\_icmple, if\_acmpeq, if\_acmpne)的目标必须是该方法中的一条指令的操作码。跳转或转移指令的目标不能是用于指定被一条 wide 指令修饰的操作的操作码;跳转或转移的目标可以是 wide 指令自身。
- 每条 tableswitch 指令的每个目标,包括缺省目标,必须是本方法中一条指令的操作码。每条 tableswitch 指令在它的跳转表中必须有一些与它的 low 和 high 跳转表操作数一致的表项,并且它的 low 值必须小于或等于它的 high 值。tableswitch 指令的目标不能是用于指定被一条 wide 指令修饰的操作的操作码;tableswitch 的目标可以是 wide 指令自身。
- 每条 lookupswitch 指令的每个目标,包括缺省目标,必须是本方法中一条指令的操作码。每条 lookupswitch 指令必须有一些与它的 npairs 操作数一致的 match\_offsets 对。match\_offsets 对必须按照有符号的 match 值的递增数值顺序存储。lookupswitch 指令的目标不能是用于指定被一条 wide 指令修饰的操作的操作码;lookupswitch 的目标可以是 wide 指令自身。
- 每条 ldc 和 ldc\_w 指令的操作数必须是对 constant\_pool 表的一个有效索引。该索引引用的常数池表项的类型必须是 CONSTANT\_Integer、CONSTANT\_Float 或者 CONSTANT\_String。
- 每条 ldc2\_w 指令的操作数必须是对 constant\_pool 表的一个有效索引。该索引引用的常数池表项的类型必须是 CONSTANT\_Long 或者 CONSTANT\_Double,另外,后面的常数池索引不能是对常数池的一个有效索引,并且那个索引处的常数池表项不能使用。
- 每条 getfield, putfield, getstatic 和 putstatic 指令的操作数必须是对 constant\_pool 表的有效索引。该索引引用的常数池表项的类型必须是 CONSTANT\_Fieldref。
- 每条 invoke virtual、invokespecial 和 invokesstatic 指令的索引操作数必须是对 con-

stant\_pool 表的有效索引。该索引引用的常数池表项必须具有类型 CONSTANT\_Methodref。

- 只允许 invokespecial 指令调用方法<init>, 即实例初始化方法(§ 3.8)。方法调用指令不能调用其他的名称以字符'<'('\'u003c')开始的方法。特别地, 类初始化方法<clinit>从不被 Java 虚拟机指令显式地调用, 而只是被 Java 虚拟机自己隐式地调用。
- 每条 invokeinterface 指令的索引操作数必须是对 constant\_pool 表的有效索引。该索引引用的常数池表项类型必须是 CONSTANT\_InterfaceMethodref。每条 invokeinterface 指令的 nargs 操作数的值必须与 CONSTANT\_InterfaceMethodref 常数池表项引用的 CONSTANT\_NameAndType\_info 结构的描述符所暗示的参数字的个数相同。每条 invokeinterface 指令的第4个操作数字节的值必须是零。
- 每条 instanceof、checkcast、new、anewarray 和 multianewarray 指令的索引操作数必须是对 constant\_pool 表的有效索引。该索引引用的常数池表项的类型必须是 CONSTANT\_class。
- anewarray 指令不能用于创建超过255维的数组。
- new 指令不能引用表示一个数组类的 CONSTANT\_Class constant\_pool 表项。new 指令不能用于创建一个数组。new 指令也不能用于创建接口或者 abstract 类实例, 这些检查在链接期进行。
- multianewarray 指令只能用于创建至少与它的 dimensions 操作数的值一样多的维数的数组类型。即, 不要求 multianewarray 指令创建由它的 CONSTANT\_Class 操作数引用的数组类型的所有维数, 它不能试图创建比这个数组类型更多的维数。每条 multianewarray 指令的 dimensions 操作数不能是零。
- 每条 newarray 指令的 atype 操作数必须从下面取一个值:  
T\_BOOLEAN(4)、T\_CHAR(5)、T\_FLOAT(6)、T\_DOUBLE(7)、T\_BYTE(8)、T\_SHORT(9)、T\_INT(10)或 T\_LONG(11)。
- 每条 iload、fload、aload、istore、fstore、astore、wide、iinc 和 ret 指令的索引操作数必须是一个不大于 max\_locals-1 的自然数。
- 每条 iload-<n>、fload-<n>、aload-<n>、istore-<n>、fstore-<n>和 astore-<n>指令的隐式索引必须不大于 max\_locals-1 的值。
- 每条 lload、dload、lstore 和 dstore 指令的索引操作数必须不大于 max\_locals-2 的值。
- 每条 lload-<n>、dload-<n>、lstore-<n>和 dstore-<n>指令的隐式索引必须不大于 max\_locals-2 的值。

#### 4.8.2 结构约束

code 数组上的结构约束指定 Java 虚拟机指令间的关系的约束。结束约束如下:

- 每条指令必须只与操作数栈和局部变量中适当类型和个数的参数一起执行, 而与引导到它的执行的执行路径无关。操作在类型 int 的值上的指令也允许操作在类型 byte、char 和 short 的值上。(如 § 3.11.1 中所述, Java 虚拟机内部地把类型 byte、char 和 short 的值转换成类型 int)。
- 在一条指令可以沿着几个不同的执行路径执行的地方, 在指令执行以前, 不论采取哪条

路径,操作数栈必须具有相同的大小。

- 在执行中的任何点一个双字类型(long 或者 double)的字的顺序都不能翻转或者被分割。在任何点双字类型的字都不能被单独地操作。
- 局部变量(或者,在双字类型的情况下,局部变量对)在被赋给一个值之前不能被访问。
- 在执行中的任何点操作数栈不能增长到包含多于 max\_stack 个字。
- 在执行中的任何点不能从操作数栈弹出多于它所包含的字。
- 每条 invokespecial 指令必须只命名一个实例初始化方法<init>,this 中的一个方法,一个 private 方法,或者 this 的超类中的一个方法。
- 当实例初始化方法<init>被调用时,一个未初始化的类实例必须位于操作数栈的适当位置上。<init>方法不能在一个已初始化的类实例上调用。
- 当任何实例方法被调用,或者任何实例变量被访问时,包含该实例方法或者实例变量的类实例必须已经被初始化。
- 在进行向后转移时,操作数栈或者局部变量中不能有未初始化的类实例。在被一个异常处理器或者 finally 子句保护的代码中的局部变量中不能有未初始化的类实例。但是,由一个异常处理器或者 finally 子句保护的代码的操作数栈上可以有未初始化的类实例。当异常被抛出时,操作数栈的内容被抛弃。
- 除了从类 Object 的构造函数得到的,每个实例初始化方法(§ 3.8)在它的实例成员被访问之前必须或者调用 this 的另一个实例初始化方法,或者调用它的直接超类 super 的一个实例初始化方法。但是,在没有超类的类 Object(§ 2.4.6)的情况下,这并不必要。
- 每个方法调用的参数必须与方法描述符(§ 4.3.3)是方法调用相容的(§ 2.6.7)。
- 不能调用 abstract 方法。
- 每条返回指令必须与它的方法的返回类型匹配。如果方法返回 byte、char、short 或者 int,则只能使用 ireturn 指令。如果方法返回 float、long 或者 double,则分别只能使用 freturn、lreturn 或者 dreturn 指令。如果方法返回一个 reference 类型,则它必须使用一条 areturn 指令这样做,并且返回的值与方法的返回描述符(§ 4.3.3)必须是赋值相容的(§ 2.6.6)。所有的实例初始化方法、静态初始化方法以及声明为返回 void 的方法只能使用 return 指令。
- 如果用 getfield 或者 putfield 访问一个超类的 protected 域,则被访问的类实例的类型必须是当前类的同类或者子类。如果用 invokevirtual 访问一个超类的 protected 方法,则被访问的类实例的类型必须是当前类的同类或者子类。
- 由 getfield 或者 putfield 指令装载或者存储的每个类实例的类型必须是该类或者该类的子类类型的一个实例。
- 由 putfield 或者 putstatic 指令存储的每个值的类型必须与被存储到的类实例或者类的域描述符(§ 4.3.2)相容。如果描述符类型是 byte、char、short 或者 int,则值必须是 int。如果描述符类型是 float、long 或者 double,则值必须分别是 float、long 或者 double。如果描述符类型是一个 reference 类型,则值必须是与描述符类型赋值相容(§ 2.6.6)的类型
- 由 astore 指令存储到类型 reference 的数组中的每个值的类型必须与该数组的成分类型

型赋值相容(§ 2.6.6)。

- 每条 `athrow` 指令只能抛出是类 `Throwable` 或者 `Throwable` 的子类的实例的值。
- 执行从不降落到 `code` 数组的底之下。
- 不能从一个局部变量装载返回地址(类型 `returnAddress` 的一个值)。
- 只能用一个单个的 `ret` 指令返回到跟随在每条 `jsr` 和 `jsr_w` 指令后面的指令。
- 如果一个子程序已经在子程序调用链中,则不能用 `jsr` 或者 `jsr_w` 指令递归地调用这个子程序(从一个 `finally` 子句中使用 `try_finally` 构造时子程序可以被嵌套。关于 Java 虚拟机子程序的更多信息,参见 § 4.9.6)。
- 对类型 `returnAddress` 的每个实例只能返回到它至多一次。如果一条 `ret` 指令返回到子程序调用链中高于对应类型 `returnAddress` 的给定实例的 `ret` 指令的点,则那个实例不能用作一个返回地址。

## 4.9 class 文件的检验

尽管 Sun 的 Java 编译器试图只生成满足前面节中的全部静态约束的类文件,但是 Java 虚拟机不能保证它要装载的任何文件是由那个编译器生成的或者是形式合适的。应用程序如 Sun 的 HotJava World Wide Web 浏览器不下载它们要编译的源文件;这些应用程序下载已经编译过的 `class` 文件。HotJava 浏览器不需要确定 `class` 文件是由一个值得依赖的 Java 编译器生成的,还是由一个试图利用解释器的敌人生成的。

编译期检查的另一个问题是版本错位。用户可能成功地编译了一个类,例如 `purchaseStockOptions`,它是 `TradingClass` 的一个子类。但是 `TradingClass` 的定义在 `PurchaseStockOptions` 编译后可能变得与预存在的二进制不相容了。方法可能被删除,或者它们的返回类型或者修饰符被改变。域可能改变类型或者由实例变量变成了类变量。方法或者变量的访问修饰符可能由 `public` 变成 `private`。对这些事情的讨论参见《Java 语言规范》一书中第十三章“二进制相容性”。

由于这些潜在的问题,Java 虚拟机需要为自己检验它试图并入的 `class` 文件保持要求的约束。一个编写良好的 Java 虚拟机仿真器在 `class` 文件被装载时能够拒绝形式糟糕的指令。其他约束可以在运行期检查。例如,Java 虚拟机实现能够标记运行期数据并且使每条指令检查它的操作数具有正确的类型。

作为替换,Sun 的 Java 虚拟机实现在链接(§ 2.16.3)期检验它认为不可信任的每个 `class` 文件满足必要的约束。Java 虚拟机代码上的结构约束用一个简单定理证明器检查。

链接期检验提高了解释器的性能。可以去掉本来必须在运行期为每条被解释的指令进行的检验约束的昂贵的检查。Java 虚拟机可以假定这些检查已被进行。例如,Java 虚拟机已经知道以下事情:

- 没有操作数栈溢出或者下溢。
- 所有局部变量的使用和存储都是有效的。
- 所有的 Java 虚拟机指令的参数具有有效的类型。

Sun 的 `class` 文件检验器独立于任何 Java 编译器。它将确证由 Sun 的当前的 Java 编译器

生成的所有代码,它也将确保其他编译器能够生成的代码,以及当前编译器不能生成的代码。满足结构标准和静态约束的任何 class 文件将得到检验器的认证。

class 文件检验器也是独立于 Java 语言的。其他语言可以被编译成 class 格式,但是只有它们满足与从 Java 源文件编译的 class 文件满足的相同的约束才能通过检验。

#### 4.9.1 检验进程

class 文件检验器操作分四段:

第一段:当一个预期的 class 文件被 Java 虚拟机装载时(§ 2.16.2),Java 虚拟机首先确认这个文件具有 Java class 文件的基本格式。头4个字节必须包含正确的幻数。所有被识别的属性必须具有适当的长度。class 文件不能被裁断或者在结尾有多余的字节。常数池不能包含任何表面上不可识别的信息。

因为 class 文件检验适当地发生在类链接时(§ 2.16.3),这个基本的 class 文件完整性检查对 class 文件内容的任何解释是必要的,并且可以认为是检验进程逻辑上的一部分。

第二段:当 class 文件被链接时,检验器进行不需查看 Code 属性(§ 4.7.4)的 code 数组即可进行的所有附加检验。这一段进行的检查包括如下:

- 确认 final 类不被子类化,final 方法不被覆盖。
- 确认每个类(除了 Object)有一个超类。
- 确认常数池满足记录的静态约束。例如,常数池中的类引用必须包含指向常数池中的一个 CONSTANT\_Utf8 字符串引用的一个域。
- 确认常数池中的所有域引用和方法引用具有有效的名称、有效的类和有效的类型描述符。

注意当它查看域和方法引用时,本段不检查以确认给定的类中实际地存在给定的域或者方法;它也不检查给定的类型描述符引用真正的类。它只检查这些项是形式上良好的。更详细的检查延迟到第三段和第四段。

第三段:仍是在链接中,检验器通过进行每个方法上的数据流分析检查 class 文件的每个方法的 Code 属性的 code 数组。检验器确认在程序中的任何给定点,不论采取哪条代码路径到达这一点:

- 操作数栈总是相同的大小并且包含相同类型的对象。
- 局部变量不被访问,直到已知它包含一个适当类型的值。
- 用合适的参数调用方法。
- 只用适当类型的值给域赋值。
- 所有的操作码在操作数栈和局部变量中有适当类型的参数。

关于本段的进一步信息,参见4.9.2节“字节码检验器”。

第四段:为了效率的原因,某些原则上能够在第三段中进行的检查被延迟到直到该方法的

代码被首次实际地调用时。这样做，检验器的第三段除非必须，避免了装载 class 文件。

例如，如果一个方法调用另一个返回类 A 的一个实例的方法，并且那个实例只被赋给同类型的一个域，则检验器不去检查类 A 是否实际存在。但是，如果它被赋给类型 B 的一个域，则 A 和 B 的定义必须都被装载以确认 A 是 B 的一个子类。

第四段是一个虚拟段，它的检查由适当的 Java 虚拟机指令来做。在引用一个类型的指令首次抛行时，执行的指令做以下事情：

- 如果被引用的类型的定义尚未装载则装载它。
- 检查当前执行的类型允许引用该类型。
- 如果类还没有被初始化则初始化它。

调用一个方法或者访问或修改一个域的指令首次执行时，执行的指令做以下事情：

- 确认在给定的类中存在被引用的方法或者域。
- 检查被引用的方法或者域具有指示的描述符。
- 检查当前执行的方法具有对被引用的方法或者域的访问权。

Java 虚拟机不用检查操作数栈上的对象的类型。这个检查已经由第三段做了。第四段中检测到的错误使 LinkageError 的子类的实例被抛出。

允许 Java 虚拟机进行第四段的任何或者所有步骤，除了作为第三段的一部分的类或者实例初始化；范例和更多的讨论参见 2.16.1“虚拟机启动”。

在 Sun 的 Java 虚拟机实现中，进行检验之后，Java 虚拟机代码中的指令被指令的一种替换形式代替（参见第九章“一种优化”）。例如，操作码 new 被 new\_quick 替换。这个替换的指令指示该指令需要的检验已经发生并且不需要再进行一遍。以后对该方法的调用将因此而更快。这些替换指令形式出现在 class 文件中是非法的，并且不应被检验器遇到。

#### 4.9.2 字节码检验器

如在前面所指示的，检验进程的第三段是 class 文件检验的四段中最复杂的一段。本节详细查看 Java 虚拟机代码的检验。

每个方法的代码被独立地检验。首先，组成代码的字节被分开成指令序列，并且每条指令的开始在 code 数组中的索引被放进一个数组中。然后检验器第二次遍览代码并分析指令。在这一段中一个数据结构被建立以持有该方法中每条 Java 虚拟机指令的有关信息。每条指令的操作数（如果有）被检查以确认它们是有效的。例如：

- 转移必须在该方法的 code 数组的边界中。
- 所有控制流指令的每个目标是一条指令的开始。在 wide 指令的情况下，wide 操作码被认为是指令的开始，给出被 wide 指令修饰的操作的操作码不被认为是一条指令的开始。不允许转移到一条指令的中间。
- 没有能够访问或者修改大于它的方法指示它使用的局部变量的个数的索引处的局部变量的指令。
- 对常数池的所有引用必须是对适当类型的表项的引用。例如，指令 ldc 只能用于类型为 int、float 的数据或者类 String 的实例；指令 getfield 必须引用一个域。
- 代码不在一条指令的中间结束。
- 执行不能降落到代码的结尾之后。

- 对每个异常处理器，该处理器保护的代码的开始与结尾点必须在一条指令的开始。开始点必须在结尾点之前。异常处理器代码必须在一条有效的指令处开始，它不能在一个被 `wide` 指令修饰的操作码处开始。

对方法的每条指令，指令执行之前检验器记录操作数栈和局部变量的内容。对于操作数栈，它需要知道栈的高度和它上面每个值的类型。对每个局部变量，它需要知道这个局部变量的内容的类型，或者局部变量包含一个不可用的或者未知的值（它可能未被初始化）。字节码检验器在确定操作数栈上的值的类型时不必区分整数类型（例如，`byte`、`short` 或 `char`）。

接着，一个数据流分析器被初始化。对方法的第一条指令，表示参数的局部变量初始地包含类型为该方法的类型描述符所指示的值；操作数栈是空的。所有的其他局部变量包含一个非法的值。对于其他指令，它们尚未被检查，不能获得关于操作数栈或者局部变量的信息。

最后，数据流分析器被运行。对于每条指令，一个“改变”位指示该指令是否需要被查看。初始地，“改变”位只对第一条指令设置。数据流分析器执行以下循环：

- 选择一条其“改变”位被设置的虚拟机指令。如果没有剩下其“改变”位被设置的指令，则方法已被成功地检验。否则，关掉选中指令的“改变”位。
- 为指令对操作数栈和局部变量的影响建模：
  - 如果指令使用来自操作数栈的值，确认栈中有足够个数的值并且栈顶的值具有适当类型。否则检验失败。
  - 如果指令使用一个局部变量，确认指定的局部变量包含适当类型的值。否则检验失败。
  - 如果指令把一个值压入到操作数栈，确认操作数栈上有为新值的足够的空间。把指示的类型加到模型操作数栈顶。
  - 如果指令修改一个局部变量，记录局部变量现在包含的新类型。
- 确定可以跟随在当前指令之后的指令，后续指令可以是下述之一：
  - 下一条指令，如果当前指令不是一条无条件控制转移指令（例如，`goto`、`return` 或者 `athrow`）。如果它可能“降落”到该方法的最后一条指令之下，则检验失败。
  - 一个有条件或者无条件转移，或者转移的目标。
  - 该指令的任何异常处理器。
- 在当前指令的执行结尾把操作数栈和局部变量的状态分别合并到后续指令的操作数栈和局部变量。在控制转移到一个异常处理器的特例中，操作数栈被设置成包含一个单个的对象，该对象的类型是由异常处理器信息指示的异常类型。
  - 如果后续指令是首次被访问，记录第2和第3步中计算的操作数栈和局部变量值是执行该后续指令前操作数栈和局部变量的状态。设置后续指令的“改变”位
  - 如果后续指令以前被访问过，把第2和第3步中计算的操作数栈和局部变量的值合并到已经在那里的值。如果有对值的任何修改，设置“改变”位
- 继续第1步。

为了合并两个操作数栈，每个栈上值的个数必须相等。栈上值的类型也必须相同，除了不同类型的 `reference` 值可以出现在两个栈上的对应位置处。在这种情况下，合并的操作数栈包含这两种类型的第一个公共超类或者公共超接口的实例的 `reference`。这样的引用类型总是存

在的,因为类型 Object 是所有类和接口类型的超类型。如果操作数栈不能被合并,则该方法的检验失败。

为了合并两个局部变量状态,局部变量的对应对被比较。如果两个类型不同,则除非都包含 reference 值,否则检验器记录局部变量包含一个不可用的值。如果局部变量对都包含 reference 值,则合并的状态包含对这两种类型的第一个公共超类的实例的 reference。

如果运行在一个方法上的数据流分析器没有报告检验失败,则该方法被 class 文件检验器的第三段成功地检验。

某些指令和数据类型使数据流分析器复杂化。我们现在更详细地检查它们。

#### 4.9.3 长整数和双精度数

long 和 double 类型的每个值在操作数栈和局部变量中占两个连续的字。

当一个 long 或者 double 被移进一个局部变量时,后面的那个局部变量被标为包含 long 或者 double 的第二半。这个特殊的值指示所有对 long 或者 double 的引用必须通过索引数较小的那个局部变量的索引。

当任何值被移进一个局部变量时,前一个局部变量被检查以看它是否包含一个 long 或者 doible 的第一个字。如果是,则前一个变量被改变以指示它现在包含一个不可用的值。由于这个 long 或者 double 的一半已被覆盖,另一半必须不再被使用。

处理操作数栈上的 64 位数量比较简单,检验器把它们作为栈上的单个单位处理。例如,dadd 操作码(如两个 double 值)的检验代码检查栈顶的两项都是类型 double 的。在计算操作数栈长度时,类型 long 和 double 的值长度为二。

无类型的操纵操作数栈的指令必须把类型 double 和 long 的值作为原子处理。例如,如果栈顶的值是一个 double 并且它遇到一条指令例如 pop 或者 dup,则检验器报告失败。必须用指令 pop2 或者 dup2 替代。

#### 4.9.4 实例初始化方法和新创建的对象

创建一个新的类实例是一个多步骤进程,Java 语句

```
...
new myClass (i;j,k);
...
```

可以由以下语句实现:

```
...
new #1           // Allocate uninitialized space for myClass
dup             // Duplicate object on the operand stack
iload_1         // Push i
iload_2         // Push j
iload_3         // push k
invokespecial myClass.<init>      // Initialize object
```

这个指令序列把新创建并初始化的对象放在操作数栈项。(把 Java 代码编译到 Java 虚拟机指令集的更多范例在第七章“为 Java 虚拟机编译”中给出。)

类 myClass 的实例初始化方法<init>把新的未初始化的对象看作它的在局部变量0中的 this 参数。在它被允许对 this 做任何事情之前它必须对 this 对象或者调用类 myClass 的一个替换的实例初始化方法,或者调用一个超类的初始化方法。

在对实例方法做数据流分析时,检验器把局部变量0初始化为包含当前类的一个对象,或者,对于实例初始化方法,局部变量0包含指示一个未初始化的对象的特殊类型。在适当的初始化方法被对这个对象(从当前类或者当前超类)调用之后,检验器的操作数栈模型和局部变量中出现的所有这个特殊的类型被当前类类型替换。检验器拒绝在一个新的对象被初始化之前使用它或者初始化一个对象两次的代码。另外,它确认每个正常的方法返回调用了或者该方法的这个类,或者直接超类,中的一个初始化方法。

类似地,一个特殊的类型被创建并且压入检验器的操作数栈模型上作为 Java 虚拟机指令 new 的结果。这个特殊的类型指示创建该类实例的指令和被创建的未初始化的类实例的类型。当一个初始化方法被对这个类实例调用时,这个特殊类型全部被想要的这个类实例的类型替换。随着数据流分析的进行,这种类型改变可能传播到以后的指令。

指令数需要作为这个特殊类型的一部分存储,因为操作数栈上一次可能存在一个类的多个尚未被初始化的实例。例如,实现

```
new InputStream(new Foo(),new InputStream("foo"))
```

的 Java 虚拟机指令序列可能在操作数栈上同时有 InputStream 的两个未初始化的实例。当一个初始化方法被对一个类实例调用时,只有与该类实例是同对象(same object)的,操作数栈或者寄存器中出现的特殊类型被替换。

一个有效的指令序列在向后转移时的操作数栈中,或者在一个异常处理器或 finally 子句保护的代码中的局部变量中,不能有未被初始化的对象。否则,一个迂回的代码可能会欺骗检验器,使检验器在该代码实际上初始化在前一段到这个循环中创建的一个类实例时认为它已经初始化了一个类实例。

#### 4.9.5 异常处理器

从 Sun 的 Java 编译器产生的 Java 虚拟机代码总是生成满足以下的异常处理器:

- 由两个不同的异常处理器保护的指令范围总是或者完全不相连,或者一个是另一个的子范围。从不会有范围的部分叠加。
- 一个异常的处理器从不在被保护的代码中。
- 对异常处理器的唯一入口是通过一个异常。不可能降落进或者“转移”到异常处理器。

class 文件检验器不加强这些限制,因为它们不会威胁到 Java 虚拟机的完整性。只要到异常处理器的每条非异常路径使操作数栈上有一个单个的对象,并且满足检验器的所有其他标准,检验器将通过代码。

#### 4.9.6 异常和 finally

给定 Java 代码段

...

```
try {
```

```
    startFaucet();
    waterLawn();
} finally {
    stopFaucet();
}
...

```

不论我们结束给草坪浇水,还是在打开水龙头,或者给草坪浇水中发生异常,Java 语言都保证 stopFaucet 被调用(水龙头被关上)即不论它的 try 子句是正常结束还是通过抛出一个异常突然结束,finally 子句都保证被执行。

为了实现 try-finally 构造,Java 编译器与两条特殊的指令 jsr(“跳转到子程序”)和 ret(“从子程序返回”)一起使用异常处理设施。很类似异常处理者的代码,finally 子句被编译成它的方法的 Java 虚拟机代码中的子程序。当调用子程序的 jsr 指令被执行时,它把它的返回地址,即正在被执行的 jsr 后面的那条指令的地址,作为类型 returnAddress 的一个值压入操作数栈。子程序的代码把返回地址存储在一个局部变量中。在子程序的结尾,一条 ret 指令从局部变量中取回返回地址并把控制转移到返回地址处的指令。

控制可以以几种不同的方式被转移到 finally 子句(finally 子程序可能被调用)。如果 try 子句正常地结束,则在对下一个 Java 表达式求值之前通过一条 jsr 指令调用 finally 子程序。把控制转移出 try 子句之外的 try 子句中 break 或者 continue 首先执行到 finally 子句代码的 ret。如果 try 子句执行一个 return,则编译后代码做以下事情:

1. 在局部变量中保存返回值(如果有)。
2. 执行到 finally 子句的代码的 jsr。
3. 从 finally 子句返回前,返回保存在局部变量中的值。

编译器建立一个特殊的异常处理器,它捕捉 try 子句抛出的任何异常。如果 try 子句抛出一个异常,则这个异常处理器做以下事情:

1. 在局部变量中保存异常。
2. 执行到 finally 子句的 jsr。
3. 从 finally 子句返回之前,重新抛出这个异常。

关于 Java 的 try-finally 构造的实现的更多信息,参见 7.13 节“编译,finally”。

finally 子句的代码表示检验器的一个特殊问题。通常,如果一条特定的指令可以通过多条路径达到并且一个特定的局部变量通过这多条路径包含不相容的值,则这个局部变量变得不可使用。但是,一个 finally 子句可以从几个不同的地方调用,产生几个不同的情况:

- 从异常处理器的调用可能有一个包含异常的特定局部变量。
- 实现 return 的调用可能有包含返回值的某个局部变量。
- 从 try 子句的底的调用可能在同一个局部变量中有一个不确定的值。

finally 子句自身的代码可能通过检验,但是在更新了 ret 指令的所有后续指令之后,检验器将注意到,异常处理器期望它持有一个异常,或者返回代码期望它持有一个返回值的那个局部变量现在包含一个不确定的值。

包含 finally 子句的检验代码是复杂的,基本的概念如下:

- 每条指令跟踪需要用它们达到该指令的一列 jsr 目标。对绝大多数代码,这个列表是空的,对 finally 子句的代码内的指令,列表的长度是1。对多层嵌套的 finally 代码(极少!)它可能长于1。
- 对每条指令和需要用它达到该指令的每条 jsr,维护一个从 jsr 指令执行后被访问或修改的所有局部的变量的位向量。
- 在执行一个实现从子程序返回的 ret 指令时,只能有一个可能的,该指令正在从中返回的子程序。两个不同的子程序不能把它们的执行“合并”到一个单个的 ret 指令。
- 为了对一条 ret 指令进行数据流分析,使用一个特殊的过程。由于检验器已知该指令正在从中返回的子程序,所以它能够找到调用该子程序的所有 jsr 指令,并且在 ret 指令执行时把操作数栈和局部变量的状态合并到 jsr 后面的那条指令的操作数栈和局部变量。对局部变量使用一组特殊的值进行合并:
- 对任何在上面为其构造的位向量指示子程序曾访问或者修改过的局部变量,使用 ret 执行时的局部变量的类型。
- 对其他局部变量,使用 jsr 指令之前的局部变量的类型。

#### 4.10 Java 虚拟机和 class 文件格式的限制

Java 虚拟机中的下述限制由本版的 Java 虚拟机规范所加:

- 每个类的常数池被 ClassFile 结构(§ 4.1)的 constant\_pool\_count 域限制到(少于)65 535个表项。
- 这作为对单个类的整体复杂性的内部限制。
- 每个方法的代码总数被 LineNumberTable 属性(§ 4.7.6)和 LocalVariableTable 属性(§ 4.7.7)中的 code 属性(§ 4.7.4)的 exception\_table 中的索引的大小限制到65 535个字节
- 一个方法中局部变量的个数被许多 Java 虚拟机指令的双字节索引操作数和 ClassFile 结构(§ 4.1)的 max\_locals 项的大小限制到65 535个。(回忆类型 long 和 double 的值被认为占据两个局部变量)。
- 一个类的域的个数被 ClassFile 结构(§ 4.1)的 fields\_count 项的大小限制到65 535个。
- 一个类的方法的个数被 ClassFile 结构(§ 4.1)的 methods\_count 项的大小限制到65 535个。
- 操作数栈的大小被 Code\_attribute 结构(§ 4.7.4)的 max\_stack 域限制到65535个字。
- 一个数组的维数被限制到255,这是由 multianewarray 指令的 dimensions 操作数的大小和 § 4.8.2对 multianewarray anewarray 和 newarray 指令所加的约束限制的。
- 一个有效的 Java 方法描述符(§ 4.3.3)必须要求255个或者更少的字的方法参数,对实例方法调用的情况这个限制包括 this 的字。注意这个限制是对方法参数的字的个数的,而不是对参数自身的个数的限制类型 long 和 double 的参数是两个字长,其他所有类型的参数是一个字长。

## 第五章 常数池解析

Java 类和接口被动态的装载(§ 2.16.2)、链接(§ 2.16.3)和初始化(§ 2.16.4)。装载是寻找特定名称的类或者接口类型的二进制形式并用这个二进制形式构造一个表示该类或者接口的 Class 对象的进程。链接是获取一个类或者接口类型的二进制形式并把它组合成 Java 虚拟机的运行期状态以使它能够被执行的进程。一个类的初始化由执行该类中声明的它的静态初始化函数和静态域的初始化器组成。

Java 虚拟机通过一个常数池上(§ 4.4)的操作进行这些过程的大部分。常数池是一个每种类型的运行期数据结构,它与传统语言中的符号表的许多用途相同。例如,Java 虚拟机指令也可以设计成取立即数或者字符串操作数以代替从常数池获取它们的操作数。类、方法和域,不论是从 Java 虚拟机的指令引用还是从其他常数池表项引用,都被称为使用常数池。

Java 编译器不假定已知 Java 虚拟机布局类、接口、类实例或者数组的方式。常数池中的引用总是初始地是符号的。在运行期,常数池中的引用的符号表示被用于计算被引用的实体的实际位置。从常数池中的符号引用动态地确定具体的值的进程被称为常数池解析(*constant pool resolution*)。常数池解析可能包括装载一个或多个类或者接口、链接几种类型以及初始化类型。有几种常数池表项,解析的细节随被解析的表项的不同而不同。

引用常数池中实体的单独的 Java 虚拟机指令负责对它们引用的实体解析。从其他常数池表项引用的常数池表项在引用表项被解析时解析。

一个给定的常数池表项可能被任意个数的 Java 虚拟机指令或者别的常数池表项引用;因此可能对一个已被解析过的常数池表项试图进行常数池解析。对一个已被成功地解析的常数池表项进行解析的尝试总是普遍地成功,并且总是产生与该表项初始的解析所产生的相同的实体。

常数池解析通常以引用该常数池的 Java 虚拟机指令的执行开始。不是在 Java 虚拟机指令的单独描述中给出它们进行的解析进程的完整描述,而是用本章概述常数池解析进程。我们将规范在解析每种常数池表项中必须检测的错误。这些错误必须被响应的顺序,以及相应地抛出的错误。

当从某些 Java 虚拟机指令的上下文引用时,对链接操作加了附加的约束。例如,getField 指令不仅要求它所引用的域的常数池表项能够被成功地解析,而且要求被解析的域不是一个类(*static*)域。如果它是一个类域,则一个异常必须被抛出。特定于一条特定 Java 虚拟机指令的执行的链接异常在那条指令的描述中给出,不在常数池解析的这个一般讨论中给出。注意,这样的异常,仅作为 Java 虚拟机指令执行的一部分,而不是作为常数解析的一部分描述,仍然被恰当地认为是 Java 虚拟机执行的链接期的失败。

Java 虚拟机规范记录并排序可能作为常数池解析的结果产生的全部异常。它不指定这些异常应当怎样被检测,只规定它们必须被检测。另外,如 § 6.3 中所述,作为 *VirtualMachineError* 的子类列出的任何虚拟机错误可能在常数池解析中的任何时间抛出。

## 5.1 类和接口解析

标记为 CONSTANT\_Class(§ 4.4.1)的常数池表项表示一个类或者接口。各种 Java 虚拟机指令在它们的执行(§ 3.6)中引用当前类的常数池中的 CONSTANT\_Class 表项。几种其他种类的常数池表项(§ 4.4.2)引用 CONSTANT\_Class 表项并且使那些类或者接口引用在引用表项被解析时解析。例如,在一个方法引用(一个 CONSTANT\_Methodref 常数池表项)能够被解析之前,它对该方法的类的引用(通过该常数池表项的 class\_index 项)必须首先被解析。

如果一个类或者接口尚未被解析,则解析进程的细节依赖于被解析的 CONSTANT\_Class 表项表示的实体的种类。数组类的处理与非数组类和接口不同。解析进程的细节也依赖于显示该类或接口的解析的引用是来自用一个类装载器(§ 2.16.2)装载的类还是一个接口。

CONSTANT\_Class 常数池表项的 name\_index 项是对一个 UTF\_8 字符串的 CONSTANT\_Utf8 常数池表项(§ 4.4.7)的引用,该 UTF\_8 字符串表示将被解析的类或者接口的完整限定名称(§ 2.7.9)。CONSTANT\_Class 常数池表项表示的实体的种类,以及如何解析这个表项,由以下确定:

- 如果将被解析的常数池表项的完整限定名称的第一个字符不是左括号(“[”),则该表项是对一个非数组类或者接口的引用。
- 如果当前类(§ 3.6)未被一个类装载器装载,则使用“正常的”类解析(§ 5.1.1)。
- 如果当前类已被一个类装载器装载,则使用应用程序定义的代码(§ 5.1.2)解析该类。
- 如果将被解析的常数池表项的完整限定名称的第一个字符是一个左括号(“[”),则该表项引用一个数组类。数组类被特殊地解析(§ 5.1.3)。

### 5.1.1 不由类装载器装载的当前类或接口

如果一个不是用类装载器装载的类或者接口引用一个非数组类或者接口 C,则进行以下步骤解析对 C 的引用:

1. 类或者接口 C 和它的超类首先被装载(§ 2.16.2)。

如果类或者接口 C 尚未被装载,则 Java 虚拟机将搜寻文件 C.class 并试图从那个文件装载类或者接口 C。注意没有保证文件 C.class 实际地包含类或者接口 C,或者甚至 C.class 是一个有效的 class 文件。也有可能类或者接口已被装载,但是尚未被初始化。装载的这一时期必须检测以下错误:

- 如果不能找到或者读入具有恰当名称的文件,则类或者接口解析抛出 NoClassDefFoundError。
- 否则,如果它确定选中的文件不是一个良好的 class 文件(§ 4.9.1的第一段),或者不是一个支持的主或者副版本的(§ 4.1)class 文件,则类或者接口解析抛出一个 NoClassDefFoundError。
- 否则,如果选中的 class 文件实际上不包含所要的类或者接口,则类或者接口解析抛出一个 NoClassDefFoundError。
- 否则,如果选中的 class 文件没有指定一个超类并且不是类 Object 的 class 文件,则类或

者接口解析抛出一个 classFormatError。

如果被装载的类的超类尚未被装载,则循环地使用这个步骤1装载它。装载一个超类必须检测步骤1a 中的任何错误,这里这个超类被认为是正在被装载的类。注意所有的接口必须以 java.lang.Object 作为它们的超类,它必须已被装载。

2. 如果类 C 和它的超类装载成功,则类 C 的超类 C 以及它的超类,如果有通过循环地进行步骤2\_4被链接和初始化。

3. 类 C 被链接(2.16.3),即,它被检验(§ 4.9)和准备。

首先,类或者接口 C 被检验以确认它的二进制表示结构上是有效的(§ 4.9.1的第二和第三段)<sup>[1]</sup>检验自身可能使类或者接口用步骤1中的过程被装载,但不被初始化(以避免循环)。

- 如果 class 文件 C.class 中包含的类或者接口 C 不满足4.8节“Java 虚拟机代码上的约束”中列出的有效的 class 文件上的静态的或者结构约束,则类或者接口解析抛出一个 VerifyError。
- 如果类或者接口 C 的 class 文件被成功地检验则类或者接口被准备。准备包括创建类或者接口的静态域并把这些域初始化为它们的标准缺省值(§ 2.5.1)。不应当把准备与静态初始化函数(§ 2.1.1)的执行相混淆;与静态初始化函数的执行不同,准备不要求任何 Java 代码的执行。在准备中:
- 如果一个未被声明为 abstract 的类有一个 abstract 方法,则类的解析抛出 AbstractMethodError。

特定于单独的 Java 虚拟机指令,但是逻辑上与常数池解析的这一段时期相关的某些检查在那些指令的文档中描述。例如,getField 指令解析它的域引用,并且只在以后检查那个域是否是一个实例域(不是 static)。这种异常仍被认为,并记录为链接异常,而不是运行期异常。

4. 下面,类被初始化,初始化过程的细节在 § 2.16.5 和《Java 语言规范》一书中给出。

- 如果一个初始化方法通过抛出某个异常 E 而突然结束,并且 E 的类不是 Error 或者其子类之一,则以 E 作为参数创建类 ExceptionInInitializerError 的一个新实例并用它替换 E。
- 如果 Java 虚拟机试图创建类 ExceptionInInitializerError 的一个新实例,但是由于发生 OutOfMemoryError 而不能创建,则作为替换,抛出 OutOfMemoryError 对象。

5. 最后,对被解析的类的访问许可被检查:

- 如果当前类或者接口没有对被解析的类或者接口的访问许可,则类或者接口解析抛出 IllegalAccessException。例如,如果一个原始地声明为 public 的类在另一个引用该类的类被编译后变成 private,则这种情况可能发生。

如果没有检测到前述错误,则该类或者接口引用的常数也解析一定已经成功地结束。但是,如果检测到一个错误,则以下之一必须为真:

- 如果在步骤1\_4中某个异常被抛出,则被解析的类一定已经被标为不可用或者被抛弃。

---

[1] Sun 的 JDK 版本1.0.2只检验 class 文件有关装载器。它假定在本地装载的 class 文件是可信的,不需要检验。

- 如果在第5步中一个异常被抛出，则被解析的类仍然有效并且可用。

在每种情况下，解析都失败，并且禁止试图进行解析的类或者接口访问被引用的类或者接口。

### 5.1.2 由类装载器装载的当前类或接口

如果一个用类装载器装载的类或者接口引用一个非数组类或者接口 C，则用同一个类装载器装载 C。那个类装载器的 loadClass 方法被对被解析的类的完整限定路径名称（§ 2.7.9）调用。loadClass 方法返回的值是被解析后的类。本节的剩余部分详细地描述这个进程。

每个类装载器是抽象类 ClassLoader 的一个子类的实例。应用程序实现 ClassLoader 的子类以扩展 Java 虚拟机动态装载类的方式。类装载器能够用于创建除了源自文件的类，还能用于创建源自源代码的类。例如，一个类可能被从网络中下载，可能在 fly 上生成，或者从一个加密的文件中解出来。

Java 虚拟机调用类装载器的 loadClass 方法以使它装载（以及可选地链接和初始化）一个类。loadClass 的第一个参数是被装载的类的完整限定名称。第二个参数是一个布尔值。值 false 指示这个指定的类必须被装载，但不被链接或者初始化；值 true 指示该类必须被装载、链接，并初始化。

类装载器的实现被要求跟踪它们已经装载、链接，和初始化的类<sup>[1]</sup>。

- 如果一个类装载器被要求装载（但不链接或者初始化）一个已被它装载（并且可能已被链接和初始化）的类或者接口，则它应当只是返回这个类或者接口。
- 如果一个类装载器被要求装载、链接和初始化一个已被它装载，但尚未链接和初始化的类或者接口，则它不应当重新装载这个类或者接口，而只应当链接和初始化它。
- 如果一个类装载器被要求装载、链接和初始化一个它已经装载、链接和初始化的类或者接口，则类装载器应当只是返回这个类或者接口。

当类装载器的 loadClass 方法被用一个它尚未装载的类或者接口的名称调用时，该类装载器必须进行下述两种操作之一以装载这个类或者接口：

- 类装载器可能创建一个表示 class 文件格式的文件的字节的字节数组；然后它必须对这些字节调用类 ClassLoader 的方法 defineClass 以把它们转换成把该类装载器作为新定义的类的类装载器的类或者接口。调用 defineClass 使 Java 虚拟机进行 § 5.1.1 的步骤 1a。

调用 defineClass 然后使该类装载器的 loadClass 方法被循环地调用以装载新定义的类或者接口的超类。超类的完整限定路径名称从该 class 文件格式中的 super\_class 项获取。当超类被装载时，loadClass 的第二个参数是 false，指示这个超类不被立即地链接和初始化。

---

[1] 未来的实现可能改变 Java 虚拟机与类 ClassLoader 之间的 API。特别地，跟踪被一个特定的类装载器装载的类和接口的将是 Java 虚拟机而不是类装载器。一个可能是 loadClass 方法将被用一个指示被装载的类或者接口的单个参数调用。虚拟机将处理链接和初始化的细节并确认类装载器不被对同一个类或者接口名称多次调用。

- 类装载器也可能用被装载的类或者接口的完整限定名称调用类 ClassLoader 中的静态方法 findSystemClass。调用这个方法使 Java 虚拟机进行 § 5.1.1 的步骤 1。结果 class 文件不被标为被一个类装载器装载。

在类或者接口以及它的超类被成功地装载之后,如果 loadClass 的第二个参数是 true 则该类或者接口被链接和初始化。如果类装载器被调用以解析一个类或者接口的常数池中的一个表项,则第二个参数总是 true。类装载器通过调用类 ClassLoader 中的方法 resolveClass 链接和初始化一个类或者接口。链接和初始化一个由类装载器创建的类或者接口非常类似于不用类装载器链接和初始化一个类或者接口(§ 5.1.1 的步骤 2~4):

首先,通过用该类或者接口的超类的完整限定名称作为第一个参数和 true 作为第二个参数调用该类装载器的 loadClass 方法链接和初始化这个超类。链接和初始化可能使这个超类自己的超类被链接和初始化。超类的链接和初始化必须检测 § 5.1.1 的第 3 步的任何错误。

其次,对被链接和初始化的类或者接口运行字节码检验器。检验器自身可能需要类或者接口被装载。如果是这样,它通过以第二个参数为 false 调用同一个类装载器的 loadClass 方法来装载它们。由于检验自身可能使类或者接口被装载(但不被链接或者初始化以避免循环),它必须对它试图装载的任何类或者接口检测 § 5.1.1 的步骤 1 的错误。运行检验器也可能引起 § 5.1.1 的步骤 3 的错误。

如果类文件被成功地检验,则类或者接口被准备(§ 5.1.1 的步骤 3b)和初始化(§ 5.1.1 的步骤 4)。

最后,检查该类或者接口的访问许可(§ 5.1.1 的步骤 5)。如果当前类或者接口没有对被解析的类的访问许可,则类的解析抛出一个 `IllegalAccessException` 异常。

如果没有检测到上述错误,则该类或者接口的装载,链接,和初始化一定是成功地结束。

### 5.1.3 数组类

如果一个标记为 CONSTANT\_Class(§ 4.4.1)的常数池表项的 name\_index 项引用的 UTF\_8 字符串(§ 4.4.7)的第一个字符是一个左括号("[") ,则该常数池表项表示一个数组类。名称中开始的连续的在括号的个数表示该数组类的维数。一个或者多个开始的连续的左括号的后面是一个表示或者一个基本类型,或者一个非数组引用类型的域描述符(§ 4.3.2)。这个域描述符表示该数组类的根类型(base type)。

进行以下步骤以解析从一个类或者接口的常数池引用的数组类。

1. 确定该数组类的维数和表示该数组类的根类型的域描述符。

2. 确定该数组类的根类型:

- 如果域描述符表示一个基本类型(它的第一个字符不是“L”),则这个基本类型就是该数组类的根类型。
- 如果域描述符表示一个非数组的引用类型(它的第一个字符是“L”),则这个引用类型就是该数组类的根类型。这个引用类型自身用上面 § 5.1.1 或者 § 5.1.2 中指示的步骤解析。

3. 如果一个表示相同根类型和相同维数的数组类已被创建,则解析的结果就是那个数组

类。否则创建一个表示指示的根类型和维数的新的数组类。

## 5.2 域和方法解析

标记为 CONSTANT\_Fieldref(§ 4.4.2)的常数池表项表示一个类或者实例变量(§ 2.9),或者一个接口的(常数)域(§ 2.13.4)。注意接口没有实例变量,标记为 CONSTANT\_Methodref(§ 4.4.2)的常数池表项表示一个类的方法(static 方法)或者一个类实例的方法(实例方法)。用 CONSTANT\_Interface Methodref 常数池表项引用接口方法;这种表项的解析在§ 5.3中描述。

为了解析一个域引用或者一个方法引用,表示该域或者方法是其成员的类的 CONSTANT\_Class 表项(§ 4.4.1)必须首先被成功地解析(§ 5.1)。因此,在解析一个 CONSTANT\_Class 常数池表项时可能抛出的任何异常也可能作为解析一个 CONSTANT\_Fieldref 或者 CONSTANT\_Methodref 表项的结果被抛出。即使表示类或者接口的 CONSTANT\_Class 表项能够被成功地解析,与方法或者域自身的链接相关的异常也可能被抛出。在解析一个域引用时:

- 如果在指定的类或者接口中不存在被引用的域,则域解析抛出一个 NoSuchFieldError。
- 否则,如果当前类没有对被引用的域的访问权,则域解析抛出一个 IllegalAccessException 异常。

如果解析一个方法:

- 如果在指定的类或者接口中不存在被引用的方法,则方法解析抛出一个 NoSuchMethodError。
- 否则如果,当前类没有对被解析的方法的访问权,则方法解析抛出一个 IllegalAccessException 异常。

## 5.3 接口方法解析

标记为 CONSTANT\_InterfaceMethodref(§ 4.4.2)的常数池表项表示对一个接口声明的实例方法的调用。这种常数池表项通过把它转换成一种依赖于机器的内部格式来解析。除了在§ 6.3中记录的之外,不可能有错误或者异常。

## 5.4 字符串解析

标记为 CONSTANT\_String(§ 4.4.3)的常数池表项表示一个字符串文字(§ 2.3)的实例,字符串文字是内建的类型 java.lang.String 的文字。CONSTANT\_String 表项表示的字符串文字的 Unicode 字符(§ 2.1)在 CONSTANT\_String 表项引用的 CONSTANT\_Utf8(§ 4.4.7)常数池表项中寻找。

Java 语言要求相同的字符串文字(即包含相同序列的 Unicode 字符的文字)必须引用类 String 的同一个实例。另外,如果对任何字符串调用方法 intern,则结果是如果那个字符串作

为一个文字出现而返回的相同的类实例。因此

`(“a”+“b”+“c”).intern() == “abc”`

必须具有值 `true`。<sup>[1]</sup>

为了解析标记为 `CONSTANT_String` 的常数池表项, Java 虚拟机检查该 `CONSTANT_String` 表项引用的 `UTF_8` 字符串所表示的 Unicode 字符系列。

- 如果标记为 `CONSTANT_String` 并且表示相同序列的 Unicode 字符的另一个常数池表项已被解析, 则解析的结果是对为前面那个常数池表项创建的类 `String` 的实例的引用。
- 否则, 如果以前对包含该常数池表项所表示的相同的 Unicode 字符序列的类 `String` 的实例调用过方法 `intern`, 则解析的结果是对类 `String` 的同一个实例的引用。
- 否则, 创建一个包含该 `CONSTANT_String` 表项所表示的 Unicode 字符序列的类 `String` 的新实例; 那个类实例是解析的结果。

除了 § 6.3 中记录的, 字符串解析中的错误和异常之外, 不可能有别的错误或者异常。

## 5.5 其他常数池项的解析

标记为 `CONSTANT_Integer` 或者 `CONSTANT_Float`(§ 4.4.4), `CONSTANT_Long` 或者 `CONSTANT_Double`(§ 4.4.5) 的常数池表项都具有直接在常数池中表示的值。它们的解析不可能抛出除 § 4.3 中记录的之外异常。

标记为 `CONSTANT_NameAndType`(§ 4.4.6) 和 `CONSTANT_Utf8`(§ 4.4.7) 的常数池表项从不被直接解析。它们只被其他常数池表项直接或者间接地引用。

---

[1] 字符串文字解析没有在 Sun 的 JDK 版本 1.0.2 中正确的实现。在 Java 虚拟机的那个实现中, 解析常数池中的一个 `CONSTANT_String` 总是分配一个新的字符串。即使两个不同的类包含相同序列的字符, 它们的两个字符串文字也不会彼此相等。一个字符串文字永远不会等于 `intern` 方法的结果。

## 第六章 Java 虚拟机指令集

Java 虚拟机指令由指定要执行的操作的操作码,后跟零或多个体现要操作的值的操作数组成。本章给出有关每条 Java 虚拟机指令格式及其执行操作的细节。

### 6.1 假定:“必须”的含义

每条指令的描述总是在满足第四章“class 文件格式”的静态的和结构的约束之 Java 虚拟机代码上下文中给出。在单个的 Java 虚拟机指令的描述中,我们经常将一些情况“必须”或“不能”陈述为“value2 必须是 int 类型。”第四章的约束保证所有这样的异常将事实上被遇到。如果指令描述中的一些约束(“必须”或“不能”)在运行期不满足,Java 虚拟机的行为是未定义的。

Java 虚拟机的 Sun 实现在装载期间使用 class 文件检验器(见 4.9 节,“class 文件的检验”),检查所有的未受信任的 Java 虚拟机代码是否满足静态和结构的约束。这样,Sun 的 Java 虚拟机将仅可见到有效的 class 文件。在 class 文件装载期间执行大多数检验是有吸引力的,因为这些检查只执行一次,显著地减少了必须在运行期所做的工作。其他实现策略是可能的,如果它们遵守第十二章“Java 语言规范”。

或者,一种无经验的 Java 虚拟机实现可以在执行期检查静态和结构的约束。然而,这第二种方法可能有重要的性能含义。

### 6.2 保留操作码

在本章后面指定的用在 Java class 文件(见第四章“class 文件格式”)中的指令的操作码外,三个操作码被保留,以供 Java 虚拟机实现内部使用。如果将来 Sun 扩展 Java 虚拟机的指令集,这些保留字保证不会被使用。

保留操作码中的两个,数 254(0xfe) 和 255(0xff),分别具有助记符 impdep1 和 impdep2。这些指令旨在分别为在软件和硬件中实现的特定实现功能提供“后门”或陷阱。第三个保留操作码,数 202(0xca),具有助记符 breakpoint,并且旨在由调试程序使用来实现断点。

虽然这些操作码已被保留,但它们仅只在 Java 虚拟机实现的内部使用。它们不能出现在有效的 class 文件中。诸如调试程序或 JIT 代码生成器(§ 3.12)这样的工具,可以直接同已装载并执行的 Java 虚拟机代码交互作用,将遇到这些操作码。这样的工具应尝试作出适当的表现,如果它们遇到这些保留指令中的任何一个。

### 6.3 虚拟机错误

当内部错误或资源限制阻止实现 Java 语言的语义时,Java 虚拟机抛出一个对象,该对象是 VirtualMachineError 类的子类的实例。Java 虚拟机规范不能预言在何处可能遇到资源限

制或内部错误，并且不精确地指定它们何时可被报告。这样，任何作为在 § 2.15.4 中 VirtualMachineError 的子类列出的虚拟机错误，都可以在 Java 虚拟机操作中的任何时候被抛出。

## 6.4 Java 虚拟机指令集

Java 虚拟机指令在本章中由图 6.1 所示表格的表项表示，按字母顺序排列。

助记符					
操作	对指令的简短描述				
格式	<table border="1"><tr><td>mnemonic</td></tr><tr><td>operand1</td></tr><tr><td>operand2</td></tr><tr><td>...</td></tr></table>	mnemonic	operand1	operand2	...
mnemonic					
operand1					
operand2					
...					
操作					
形式	助记符 = 操作码				
栈	..., value1, value2 ⇒ ..., value3				
描述	对操作数栈的常数或常数池表项的约束以及对执行的操作和结果的类型等等细节的较长的描述。				
链接异常	如果该指令的执行可能抛出任何链接异常，它们各从一行开始，按照它们必须被抛出的顺序。				
运行期异常	如果指令的执行可能抛出任何运行期异常，它们各从一行开始，按照它们必须被抛出的顺序。 除链接和运行期异常外，如果有，为指令列出，该指令不能抛出任何运行期异常，除非是 VirtualMachineError 或其子类的实例。				
注解	不是严格意义上的指令规范的一部分的注释被作为注解放在描述的末端。				

图 6.1 指令页范例

指令格式图表的每个单元代表一个 8 位字节。指令的助记符是其名称。其操作码是它的数值表示，并同时给出其十进制和十六进制形式。只有数值表示实际出现于 class 文件的 Java 虚拟机代码中。

记住在编译时产生并嵌入 Java 虚拟机指令的“操作数”，以及在运行期计算并提供给操作数栈的“操作数”。虽然它们由几个不同区域提供，但所有这些操作数表示同样的事：正在执行

的 Java 虚拟机指令所要操作的值。通过隐式地从其操作数栈取它的一些操作数,而不是在其编译代码中作为附加的操作数字节、寄存器数等显式地表示它们,Java 虚拟机代码保持紧凑。

一些指令被作为相关指令族的成员表示,指令族共享一种描述、格式和操作数栈图表。这样,指令族包含几个操作码和操作码助记符;只有该族的助记符出现在指令格式图表中,而分隔的形式行列出所有成员的助记符和操作码。例如,指令族 lconst\_{l} 的形式行,给出该族的两条指令(lconst\_0 和 lconst\_1)的助记符和操作码信息为:

格式      lconst\_0=9(0x9),  
              lconst\_1=10(0xa)

在 Java 虚拟机指令的描述中,指令执行对当前框架(§ 3.6)操作数栈的影响被表示为正文,栈从左到右增长,并且每个字(§ 3.4)分别表示。这样,

栈      ... , value1, value2 =>  
              ... , result

展示了一个操作由处于操作数栈顶的一个字 value2 和它下面的一个字 value1 开始。作为指令执行的结果,value1 和 value2 从操作数栈弹出,并被一个字的 result 替换,result 已通过该指令计算出。操作数栈的其余部分,以省略号(...) 表示,不受指令执行的影响。

long 和 double 类型从操作数栈取两个字。在操作数栈表示中,每个字用圆点记号分别表示:

栈      ... , value1.word1, value1.word2, value2.word1, value2.word2 =>  
              ... , result.word1, result.word2

Java 虚拟机规范并不强制两个字如何用于表示 64 位的 long 或 double 值,它仅要求特定的实现内部一致。

---

## aaload

---

操作	从数组装载 reference。	
格式	<table border="1"><tr><td>aaload</td></tr></table>	aaload
aaload		
形式	aaload = 50(0x32)	
栈	..., arrayref, index⇒ ..., value	
描述	arrayref 必须是 reference 类型并且必须引用成分为 reference 类型的数组。index 必须为 int 类型。arrayref 和 index 都从操作数栈弹出。数组 index 处成分的 reference 值被获取并压入操作数栈顶。	
运行期异常	如果 arrayref 为 null, aaload 抛出 NullPointerException 否则, 如果 index 不在由 arrayref 引用的数组的边界内, aaload 指令抛出 ArrayIndexOutOfBoundsException。	

---

## aastore

---

操作	存储到 reference 数组中。	
格式	<table border="1"><tr><td>aastore</td></tr></table>	aastore
aastore		
形式	aastore = 83(0x53)	
栈	..., arrayref, index, value⇒ ...	
描述	arrayref 必须是 reference 类型并且必须引用成分为 reference 类型的数组。index 必须是 int 类型, 同时 value 必须是 reference 类型。arrayref、index 和 value 从操作数栈弹出。reference 类型的 value 作为单元存储在数组的 index 位置。 value 的类型必须同 arrayref 所引用的数组的分类型赋值相容(§ 2.6.6)。将引用类型 S(源)的值赋予引用类型 T(目标)的变量仅当类型 S 支持所有定义在类型 T 上的操作时才被允许。详细规则如下： <ul style="list-style-type: none"><li>· 如果 S 是类类型, 则：<ul style="list-style-type: none"><li>· 如果 T 是类类型, 则 S 必须是与 T 相同的类(§ 2.8.1), 或者 S 必须是 T 的子类。</li></ul></li></ul>	

- 如果 T 是接口类型,S 必须实现(§ 2.13)接口 T。
- 如果 S 是数组类型,即类型 SC[],也就是成分为 SC 类型的数组,则:
  - 如果 T 是类类型,T 必须是 Object(§ 2.4.6),或者:
  - 如果 T 是数组类型,即类型 TC[],成分为 TC 类型的数组,则或者 TC 及 SC 必须是相同的基本类型,或者
  - TC 及 SC 必须同为经上述规则,类型 SC 可赋值给 TC 的引用类型。S 不能是接口类型,因为不存在接口的实例,只有类和数组的实例。

<b>运行期异常</b>	<p>如果 arrayref 为 null,aastore 抛出 NullPointerException。</p> <p>否则,如果 index 不在 arrayref 所引用的数组的边界内,aastore 指令抛出 ArrayIndexOutOfBoundsException。</p> <p>否则,如果 arrayref 不为 null,并且 value 的实际类型同数组成分的实际类型不赋值相容,aastore 抛出 ArrayStoreException。</p>
--------------	---

### aconst\_null

<b>操作</b>	压入 null。
<b>格式</b>	aconst_null
<b>形式</b>	aconst_null=1(0x1)
<b>栈</b>	$\cdots \Rightarrow$ $\cdots, \text{null}$
<b>描述</b>	将 null 对象 reference 压入操作数栈顶。
<b>注解</b>	Java 虚拟机不指定 null 的具体值。

### aload

<b>操作</b>	从局部变量装载 reference。
<b>格式</b>	aload index
<b>形式</b>	aload=25(0x19)

栈                    ...⇒

..., objectref

#### 描述

index 是一个无符号字节, 必须是当前框架(§ 3.6)的局部变量的有效索引。index 处的局部变量必须包含 reference。index 处的局部变量的 objectref 被压入操作数栈。

#### 注解

aload 指令不能用于从局部变量装载 returnAddress 类型的值到操作数栈。这种同 astore 指令的非对称性是有意的。aload 操作码可以同 wide 指令一起使用两字节无符号索引访问局部变量。

---

### aload\_<n>

---

#### 操作

从局部变量装载 reference。

#### 格式

aload_<n>
-----------

#### 形式

aload\_0=42(0x2a)  
aload\_1=43(0x2b)  
aload\_2=44(0x2c)  
aload\_3=45(0x2d)

#### 栈

...⇒

..., objectref

#### 描述

<n>必须是对当前框架(§ 3.6)局部变量的有效索引。<n>处的局部变量必须包含 reference。index 处的局部变量的 objectref 被压入操作数栈。

#### 注解

aload\_<n>指令不能用于从局部变量装载 returnAddress 类型的值到操作数栈。这种同相应的 astore\_<n>指令的非对称性是有意的。每个aload\_<n>指令与带有 index<n>的aload 相同, 只不过操作数<n>是隐式的。

---

### anewarray

---

#### 操作

创建新的 reference 数组。

#### 格式

anewarray
indexbyte1
indexbyte2

<b>形式</b>	<code>anewarray=189(0xbd)</code>
<b>栈</b>	$\dots, \text{count} \Rightarrow$ $\dots, \text{arrayref}$
<b>描述</b>	<p><code>count</code> 必须是 <code>int</code> 类型。它从操作数栈弹出。<code>count</code> 代表要创建的数组的成分的数目。无符号的 <code>indexbyte1</code> 和 <code>indexbyte2</code> 用于构造当前类 (§ 3.6) 常数池的索引，其中索引的值为 <math>(\text{indexbyte1} \ll 8)   \text{indexbyte2}</math>。常数池该索引处的项目必须标记为 <code>CONSTANT_Class</code> (§ 4.4.1)，对类、数组或接口类型的符号引用。该符号引用被解析 (§ 5.1)，新的数组具有该类型的成分及长度 <code>count</code>，被从垃圾回收堆中分配，同时对这个新数组对象的 <code>reference</code> 类型的 <code>arrayref</code> 被压入操作数栈。新数组的所有成分初始化为 <code>null</code>，<code>reference</code> 类型的缺省值 (§ 2.5.1)</p>
<b>链接异常</b>	在解析 <code>CONSTANT_Class</code> 常数池项目时，任何在 § 5.1 中的异常都可以被抛出。
<b>运行期异常</b>	否则，如果 <code>count</code> 小于零， <code>anewarray</code> 指令抛出 <code>NegativeArraySizeException</code> 。
<b>注解</b>	<code>anewarray</code> 指令用于创建一维的对象引用数组。它还可用于创建多维数组的一部分。

---

### areturn

---

<b>操作</b>	从方法返回 <code>reference</code> 。
<b>格式</b>	<code>areturn</code>
<b>形式</b>	<code>areturn=176(0xb0)</code>
<b>栈</b>	$\dots, \text{objectref} \Rightarrow$ $[\text{empty}]$
<b>描述</b>	<p><code>objectref</code> 必须是 <code>reference</code> 类型并且必须引用一个对象，该对象的类型同由返回方法的返回描述符 (§ 4.3.3) 所代表的类型赋值相容 (§ 2.6.6)。<code>objectref</code> 从当前框架 (§ 3.6) 的操作数栈弹出并被压入列调用者框架的操作数栈。当前方法的操作数栈上的其它任何值被抛弃。如果返回方法为 syn-</p>

chronized 方法, 调用该方法时获得的或重入的监视器被释放或退出(分别的)好像执行了 monitorexit 指令。  
解释器恢复调用者框架并将控制返回给调用者。

---

### arraylength

---

操作	获取数组长度。	
格式	<table border="1"><tr><td>arraylength</td></tr></table>	arraylength
arraylength		
形式	arraylength=190(0xbe)	
栈	..., arrayref⇒ ..., length	
描述	arrayref 必须是 reference 类型并且必须引用一个数组。它从操作数栈弹出。它引用的数组的长度是确定的。该长度作为 int 压入操作数栈。	
运行期异常	如果 arrayref 为 null, arraylength 指令抛出 NullPointerException。	

---

### astore

---

操作	将 reference 存储到局部变量中。		
格式	<table border="1"><tr><td>astore</td></tr><tr><td>index</td></tr></table>	astore	index
astore			
index			
形式	astore=58(0x3a)		
栈	..., objectref⇒ ...		
描述	index 是无符号字节, 必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 objectref 必须是 returnAddress 类型或 reference 类型。它从操作数栈弹出, index 处的局部变量的值被设置为 objectref。		
注解	当实现 Java 的 finally 关键字(见 7.13 节, “编译 finally”)时, astore 指令带 returnAddress 类型的 objectref 使用。aload 指令不能用于从局部变量装载 returnAddress 类型的值到操作数栈。这种同 astore 指令的非对称性是有		

意的。

`astore` 操作码可以同 `wide` 指令一同使用来用两字节无符号索引访问局部变量。

---

### `astore_<n>`

---

**操作** 将 `reference` 存储到局部变量中。

**格式** `astore_<n>`

**形式** `astore_0=75(0x4b)`  
`astore_1=76(0x4c)`  
`astore_2=77(0x4d)`  
`astore_3=78(0x4e)`

**栈** `..., objectref⇒`  
`...`

**描述** `<n>` 必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 `objectref` 必须是 `returnAddress` 类型或 `reference` 类型。它从操作数栈弹出, 同时 `<n>` 处的局部变量的值被设置为 `objectref`

**注解** 当实现 Java 的 `finally` 关键字(见 7.13 节, “编译 `finally`”)时, `astore_<n>` 指令带 `return Address` 类型的 `objectref` 使用, `aload_<n>` 指令不能用于从局部变量装载 `returnAddress` 类型的值到操作数栈。这种同相应的 `astore_<n>` 的非对称性是有意的。  
每个 `astore_<n>` 指令同带索引 `<n>` 的 `astore` 相同, 只不过操作数 `<n>` 是隐式的。

---

### `athrow`

---

**操作** 抛出异常或错误。

**格式** `athrow`

**形式** `athrow=191(0xbf)`

**栈** `..., objectref⇒`  
`objectref`

<b>描述</b>	<p>objectref 必须是 reference 类型并且必须引用一个对象,该对象是 Throw-able 类或其子类的实例。它从操作数栈弹出。通过在当前框架(§ 3.6)中查找捕捉 objectref 类或其超类的最近的 catch 子句来抛出 objectref。</p> <p>如果找到了 catch 子句,它包含用于处理该异常的代码的位置。PC 寄存器被复位到该位置,当前框架的操作数栈被清除,objectref 被压回到操作数栈,并继续执行。如果在当前框架中未找到适当的子句,该框架被弹出,恢复其调用者的框架,同时再次抛出 objectref。</p> <p>如果未找到处理该异常的 catch 子句,则退出当前线程。</p>
<b>运行期异常</b>	如果 objectref 为 null,athrow 抛出 NullPointerException 而不是 objectref。
<b>注解</b>	<p>athrow 指令的操作数栈图表可能造成误导:如果在当前方法中找到了该异常的处理者,athrow 指令抛弃操作数栈上的所有字,然后将抛出的对象压入该栈。然而,如果在当前方法中未找到处理者,并且抛出的异常在方法调用者链的更上层,则处理该异常的方法(如果有)的操作数栈被清除,并将 objectref 压入到这个空的操作数栈。所有插入栈框架,从抛出异常的方法到(但不包括)处理该异常的方法都被抛弃。</p>

---

## baload

---

<b>操作</b>	从数组装载 byte 或 boolean。	
<b>格式</b>	<table border="1"><tr><td>baload</td></tr></table>	baload
baload		
<b>形式</b>	baload=51(0x33)	
<b>栈</b>	$\dots, \text{arrayref}, \text{index} \Rightarrow$ $\dots, \text{value}$	
<b>描述</b>	<p>arrayref 必须是 reference 类型并且必须引用一个数组,该数组成分是 byte 类型或 boolean 类型。index 必须是 int 类型。arrayref 和 index 都从操作数栈弹出。数组中 index 处的成分的 byte 值被获取,有符号扩展为 int 值,并压入操作数栈顶。</p>	
<b>运行期异常</b>	如果 arrayref 为 null,baload 抛出 NullPointerException。否则,如果 index 不在 arrayref 所引用的数组的边界内,则 baload 指令抛出 ArrayIndexOutOfBoundsException。	

**注解** `baload` 指令用于从 byte 和 boolean 数组装载值。在 Sun 的 Java 虚拟机实现中, boolean 数组(T\_BOOLEAN 类型的数组;见 § 3.1 及 `newarray` 指令的描述)被作为8位值的数组实现。其他实现可能实现压缩的 boolean 数组;这种实现的 `baload` 指令必定用于访问那些数组。

---

### **bastore**

---

**操作** 存储到 byte 或 boolean 数组。

**格式**

<code>bastore</code>
----------------------

**形式** `bastore=84(0x54)`

**栈** `..., arrayref, index, value⇒`  
...

**描述** `arrayref` 必须是 reference 类型并且必须引用一个数组,该数组成分是 byte 类型或 boolean 类型。`index` 和 `value` 必须都是 int 类型。`arrayref, index` 和 `value` 从操作数栈弹出。`int` 值被截断为 byte,并作为数组的 `index` 处的成分保存。

**运行期异常** 如果 `arrayref` 为 null,`bastore` 抛出 `NullPointerException`。否则,如果 `index` 不在 `arrayref` 所引用的数组的边界内,`bastore` 指令抛出 `ArrayIndexOutOfBoundsException`。

**注解** `bastore` 指令用于将值存储到 byte 和 boolean 数组中。在 Sun 的 Java 虚拟机实现中, boolean 数组(T\_BOOLEAN 类型的数组;见 § 3.1 及 `newarray` 指令的描述)作为8位值的数组实现。其他实现可能实现压缩的 boolean 数组;这种实现的 `bastore` 指令必定用于存储到那些数组中。

---

### **bipush**

---

**操作** 压入 byte。

**格式**

<code>bipush</code>
<code>byte</code>

**形式** `bipush=16(0x10)`

<b>栈</b>	...⇒ ..., value	
<b>描述</b>	紧接着的 byte 被有符号扩展为 int, 同时结果值被压入操作数栈。	
<hr/>		
<b>caload</b>		
<b>操作</b>	从数组装载 char。	
<b>格式</b>	<table border="1"><tr><td>caload</td></tr></table>	caload
caload		
<b>形式</b>	caload=52(0x34)	
<b>栈</b>	..., arrayref, index⇒ ..., value	
<b>描述</b>	arrayref 必须是 reference 类型并且必须引用一个数组, 该数组的成分是 char 类型。index 必须是 int 类型。arrayref 和 index 都从操作数栈弹出。数组的 index 处成分的 char 值被获取, 零扩展为 int 值, 并压入操作数栈顶。	
<b>运行期异常</b>	如果 arrayref 为 null, caload 抛出 NullPointerException。否则, 如果 index 不在 arrayref 所引用的数组的边界内, caload 指令抛出 ArrayIndexOutOfBoundsException。	
<hr/>		
<b>castore</b>		
<b>操作</b>	存储到 char 数组。	
<b>格式</b>	<table border="1"><tr><td>castore</td></tr></table>	castore
castore		
<b>形式</b>	castore=85(0x55)	
<b>栈</b>	..., arrayref, index, value⇒ ...	
<b>描述</b>	arrayref 必须是 reference 类型并且必须引用一个数组, 该数组的成分是 char 类型。index 和 value 必须都是 int 类型。arrayref、index 和 value 从操作数栈弹出。int 值被截断为 char, 并作为数组的 index 处的成分保存。	

**运行期异常** 如果 arrayref 为 null, astore 指令抛出 NullPointerException。否则, 如果 index 不在 arrayref 所引用的数组的边界内, astore 指令抛出 ArrayIndexOutOfBoundsException。

---

### checkcast

---

**操作** 检查对象是否为给定类型。

**格式**

checkcast
indexbyte1
indexbyte2

**形式** checkcast=192(0xc0)

**栈** ..., objectref⇒  
..., objectref

**描述** objectref 必须是 reference 类型。无符号 indexbyte1 和 indexbyte2 用于构造当前类(§ 3.6)的常数池的索引, 其中索引值(indexbyte1<sub>8</sub>) | indexbyte2。常数池中 index 处项目必须是 CONSTANT\_CLASS(§ 4.4.1), 对类、数组或接口类型的符号引用。该符号引用被解析(§ 5.1)。

如果 objectref 为 null 或者可以被转换为解析的类、数组或接口类型, 操作数栈不变; 否则, checkcast 指令抛出 ClassCastException。下述规则用于确定不是 null 的 objectref 是否可以转换为解析的类型: 如果 S 是 objectref 引用的对象的类, 而 T 是解析的类、数组或接口类型, checkcast 确定 objectref 是否可转换为 T 类型, 如下:

- 如果 S 是普通的(非数组)类, 则:
  - 如果 T 是类类型, 则 S 必须是与 T 相同的类(§ 2.8.1)或 T 的子类。
  - 如果 T 是接口类型, 则 S 必须实现(§ 2.13)T 接口。
- 如果 S 是代表数组类型 SC[] 的类, 即成分为 SC 类型的数组, 则:
  - 如果 T 是类类型, 则 T 必须是 Object(§ 2.4.6)。
  - 如果 T 是数组类型 TC[], 即成分为 TC 类型的数组, 则下述之一必定为真:
    - TC 和 SC 为相同的基本类型(§ 2.4.1)
    - TC 和 SC 为引用类型(§ 2.4.5), 并且类型 SC 可通过这些运行时规则转换为 TC。

S 不能是接口类型, 因为不存在接口的实例, 只有类和数组的实例。

**链接异常** 在解析 CONSTANT\_CLASS 常数池项目时,任何 § 5.1 中的异常都可被抛出。

**运行期异常** 否则,如果 objectref 不能被转换为解析的类,数组或接口类型,checkcast 指令抛出 ClassCastException。

**注解** checkcast 指令同 instanceof 指令非常相似。不同之处在于对 null 的处理,当检测失败时的行为(checkcast 抛出异常,instanceof 压入结果代码)以及对操作数栈的影响。

---

### d2f

---

**操作** 将 double 转换为 float。

**格式**

d2f
-----

**形式** d2f=144(0x90)

**栈** ..., value.word1, value.word2⇒  
..., result

**描述** 操作数栈顶的值必须是 double 类型。它从操作数栈弹出并使用 IEEE 754 最接近舍入模式转换为 float 结果。结果被压入操作数栈。  
有限 value 太小以致不能由 float 代表的,被转换为具有同样符号的零;有限 value 太大以致不能由 float 代表的,被转换为具有同样符号的无穷数。  
double 类型的 NaN 转换为 float 类型的 NaN。

**注解** d2f 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息,并且可能丢失精度。

---

### d2i

---

**操作** 将 double 转换为 int。

**格式**

d2i
-----

**形式** d2i=142(0x83)

**栈** ..., value.word1, value.word2⇒

…, result

**描述** 操作数栈顶的 value 必须是 double 类型。它从操作数栈弹出并转换为 int。result 被压入操作数栈：

- 如果 value 为 NaN, 转换的 result 为 int 0。
- 否则, 如果 value 不是无穷, 它被舍入为整数值 V, 使用 IEEE 754 向零舍入模式进行向零舍入。如果该整数值 V 可以表示为 int, 则 result 为 int 值 V。
- 否则, 或者 value 必定太小(大数量的负值或负无穷)并且 result 是 int 类型的最小的可表示值, 或者 value 必定太大(大数量的正值或正无穷)并且 result 是 int 类型的最大可表示值。

**注解** d2i 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息, 并且可能丢失精度。

---

## d2l

---

**操作** 将 double 转换为 long。

**格式**

d2l

**形式** d2l=143(0x8f)

**栈**

…, value.word1, value.word2⇒  
…, result.word1, result.word2

**描述** 操作数栈顶的 value 必须是 double 类型。它从操作数栈弹出并转换为 long。result 被压入操作数栈：

- 如果 value 是 NaN, 转换的 result 为 long 0。
- 否则, 如果 value 不是无穷, 它被舍入为整数值 V, 使用 IEEE 754 向零舍入模式进行向零舍入。如果整数值 V 可以表示为 long, 则 result 为 long 值 V。
- 否则, 或者 value 必定太小(大数量的负值或者负无穷)并且 result 是 long 类型的最小可表示值, 或者 value 必定太大(大数量的正值或正无穷)并且 result 是 long 类型的最大可表示值。

**注解** d2l 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息, 并且可能丢失精度。

---

**dadd**

---

<b>操作</b>	double 相加。	
<b>格式</b>	<table border="1"><tr><td>dadd</td></tr></table>	dadd
dadd		
<b>形式</b>	dadd=99(0x63)	
<b>栈</b>	$\dots, \text{value1.word1}, \text{value1.word2}, \text{value2.word1}, \text{value2.word2} \Rightarrow \dots, \text{result.word1}, \text{result.word2}$	
<b>描述</b>	<p><code>value1</code> 和 <code>value2</code> 必须都是 double 类型。这些值从操作数栈弹出。double 类型的 <code>result</code> 为 <code>value1 + value2</code>。<code>result</code> 被压入操作数栈。<code>dadd</code> 指令的结果由 IEEE 算术规则控制：</p> <ul style="list-style-type: none"><li>· 如果任一值为 NaN，结果为 NaN。</li><li>· 两个符号相反的无穷之和为 NaN。</li><li>· 两个符号相同的无穷之和为该符号的无穷大。</li><li>· 无穷和任何有限值之和等于无穷。</li><li>· 两个符号相反的零之和为正零。</li><li>· 两个同号的零之和为该符号的零。</li><li>· 零和非零有限值之和等于非零值。</li><li>· 两个数量相同符号相反的非零有限值之和为正零。</li><li>· 在其他情况下，不涉及无穷、零或 NaN，并且两个值有相同的符号或有不同的数量，其和被计算并使用 IEEE 754 最接近舍入模式舍入到最接近的可表示值。如果数量太大以致不能表示为 double，我们说操作溢出；结果为适当符号的无穷。如果数量太小以致不能表示为 double，我们说操作下溢；则结果为适当符号的零。</li></ul> <p>Java 虚拟机需要 IEEE 754 定义的逐渐下溢的支持。尽管可能出现溢出，下溢或丢失精度，<code>dadd</code> 指令的执行从不抛出运行期异常。</p>	

---

**daload**

---

<b>操作</b>	从数组装载 double。	
<b>格式</b>	<table border="1"><tr><td>daload</td></tr></table>	daload
daload		
<b>形式</b>	daload=49(0x31)	
<b>栈</b>	$\dots, \text{arrayref}, \text{index} \Rightarrow$	

..., value. word1, value. word2

**描述** arrayref 必须是 reference 类型并且必须引用一个数组, 该数组成分是 double 类型。index 必须是 int 类型。arrayref 和 index 都从操作数栈弹出。数组中 index 处的成分的 double 类型的 value 被获取并压入操作数栈顶。

**运行期异常** 如果 arrayref 为 null, daload 抛出 NullPointerException。否则, 如果 index 不在 arrayref 所引用的数组的边界内, daload 指令抛出 ArrayIndexOutOfBoundsException。

---

### dastore

---

**操作** 存储到 double 数组。

**格式** dastore

**形式** dastore=82(0x52)

**栈** ..., arrayref, index, value. word1, value. word2⇒  
...

**描述** arrayref 必须是 reference 类型并且必须引用一个数组, 该数组的成分是 double 类型。index 必须是 int 类型。并且 value 必须是 double 类型。arrayref、index 和 value 从操作数栈弹出。double 类型的 value 作为数组中 index 处的成分存储。

**运行期异常** 如果 arrayref 为 null, dastore 抛出 NullPointerException。否则, 如果 index 不在 arrayref 引用的数组的边界内, dastore 抛出 ArrayIndexOutOfBoundsException。

---

### dcmp<op>

---

**操作** 比较 double。

**格式** dcmp<op>

**形式** dcmpg=152(0x98)  
dcmpl=151(0x97)

**栈**       $\dots, \text{value1.word1}, \text{value1.word2}, \text{value2.word1}, \text{value2.word1} \Rightarrow$   
 $\dots, \text{result}$

**描述**       $\text{value1}$  和  $\text{value2}$  必须都是 double 类型。这些值从操作数栈弹出，并执行浮点数比较。如果  $\text{value1}$  大于  $\text{value2}$ , 将 int 值 1 压入操作数栈。如果  $\text{value1}$  等于  $\text{value2}$ , 将 int 值 0 压入操作数栈。如果  $\text{value1}$  小于  $\text{value2}$ , 将 int 值 -1 压入操作数栈。如果  $\text{value1}$  或  $\text{value2}$  是 NaN, `dcmpg` 指令将 int 值 1 压入操作数栈，并且 `dcmpl` 指令将 int 值 -1 压入操作数栈。  
浮点数比较按照 IEEE 754 执行。除 NaN 外的所有值被排序，负无穷小于所有有限值，而正无穷大于所有有限值。正零和负零被认为相等。

**注解**      `dcmpg` 和 `dcmpl` 指令仅在涉及 NaN 的比较的处理上不同。NaN 不排序，所以如果操作数之一或全部为 NaN，则所有 double 比较都将失败。`dcmpg` 和 `dcmpl` 均可用，任何 double 比较可以编译为向操作数栈压入同样的 `result`，无论是比较在非 NaN 值上失败或是因为遇到 NaN 而失败。更多信息，见 7.5 节，“更多控制范例。”

---

### **dconst\_<d>**

---

**操作**      压入 double。

**格式**

<code>dconst_&lt;d&gt;</code>
-------------------------------

**形式**      `dconst_0=14(0xe)`  
`dconst_1=15(0xf)`

**栈**       $\dots \Rightarrow$   
 $\dots, \langle d \rangle.\text{word1}, \langle d \rangle.\text{word2}$

**描述**      将 double 常数  $\langle d \rangle$  (0.0 或 1.0) 压入操作数栈。

---

### **ddiv**

---

**操作**      double 相除。

**格式**

<code>ddiv</code>
-------------------

**形式**      `ddiv=111(0x6f)`

**栈**

..., value1.word1, value1.word2, value2.word1, value2.word2⇒  
..., result.word1, result.word2

**描述**

value1 和 value2 必须都是 double 类型。这些值从操作数栈弹出。double 类型的 result 为 value1/value2。result 被压入操作数栈。ddiv 指令的结果由 IEEE 算术规则控制：

- 如果任一值为 NaN，结果为 NaN。
- 如果值均不是 NaN，如果两个值同号则结果的符号为正，如果两个值异号则为负。
- 无穷被无穷除导致 NaN。
- 无穷被有限值除导致有符号无穷，遵循刚给出的符号生成规则。
- 有限值被无穷除导致有符号的零，遵循刚给出的符号生成规则。
- 零被零除导致 NaN；零被其他任何有限值除导致有符号的零，遵循刚给出的符号生成规则。
- 非零有限值被零除导致有符号的无穷，遵循刚给出的符号生成规则。
- 在其他情况下，不涉及无穷、零或 NaN，商被计算并使用 IEEE 754 最接近舍入模式舍入为最接近的 double。如果数量太大以致不能表示为 double，我们说操作溢出；结果为适当符号的无穷。如果数量太小以致不能表示为 double，我们说操作下溢；结果为适当符号的零。

Java 虚拟机需要由 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出、下溢、被零除或丢失精度，ddiv 指令的执行从不抛出运行期异常。

---

**dload**

---

**操作**

从局部变量装载 double。

**格式**

dload
index

**形式**

dload = 24(0x18)

**栈**

...⇒  
..., value.word1, value.word2

**描述**

index 是无符号字节。index 和 index+1 必须都是对当前框架（§ 3.6）的局部变量的有效索引。index 和 index+1 处的局部变量合在一起必须包含一个 double。index 和 index+1 处的局部变量被压入操作数栈。

**注解**

dload 操作码可以同 wide 指令一同使用来用两字节无符号索引访问局部变

量。

---

### dload\_<n>

---

**操作** 从局部变量装载 double。

**格式** dload\_<n>

**形式**  
dload\_0=38(0x26)  
dload\_1=39(0x27)  
dload\_2=40(0x28)  
dload\_3=41(0x29)

**栈**  $\dots \Rightarrow \dots, \text{value.word1}, \text{value.word2}$

**描述** <n>和<n>+1必须都是对当前框架(§ 3.6)的局部变量的有效索引。在<n>和<n>+1处的局部变量合在一起必须包含一个 double。<n>和<n>+1处的局部变量被压入操作数栈。

**注解** 每个 dload\_<n>指令同带 index<n>的 dload 相同,只不过操作数<n>是隐式的。

---

### dmul

---

**操作** double 相乘。

**格式** dmul

**形式** dmul=107(0x6b)

**栈**  $\dots, \text{value1.word1}, \text{value1.word2}, \text{value2.word1}, \text{value2.word2} \Rightarrow \dots, \text{result.word1}, \text{result.word2}$

**描述** value1 和 value2 必须都是 double 类型。这些值从操作数栈弹出。double 类型的 result 为 value1 \* value2。结果被压入操作数栈。dmul 指令的结果被 IEEE 算术规则控制:  
· 如果任一值为 NaN, 结果为 NaN。

- 如果值均不是 NaN，则如果两个值同号则结果的符号为正，如果两个值异号则为负。
- 无穷乘以零导致 NaN。
- 无穷乘以有限值导致有符号无穷，遵循刚给出的符号生成规则。
- 在其他情况下，不涉及无穷或 NaN，乘积被计算并使用 IEEE 754 最接近舍入模式舍入为最接近的可表示值。如果数量太大以致不能表示为 double，我们说操作溢出；则结果为适当符号的无穷。如果数量太小以致不能表示为 double，我们说操作下溢；结果为适当符号的零。

Java 虚拟机需要由 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出，下溢或丢失精度，dmul 指令的执行从不抛出运行期异常。

## dneg

<b>操作</b>	对 double 求反。	
<b>格式</b>	<table border="1"><tr><td>dneg</td></tr></table>	dneg
dneg		
<b>形式</b>	dneg=119(0x77)	
<b>栈</b>	..., value.word1, value.word2⇒ ..., result.word1, result.word2	
<b>描述</b>	value 必须是 double 类型。它从操作数栈弹出。double 类型的 result 是 value 的算术相反数，即 -value。result 被压入操作数栈。对 double 值，相反数同从零减去不同。如果 x 是 +0.0，则 $0.0 - x$ 等于 +0.0，但 $-x$ 等于 -0.0。一目减仅取反 double 的符号。特别关心的情形： <ul style="list-style-type: none"> <li>· 如果操作数为 NaN，结果为 NaN（记得 NaN 无符号）。</li> <li>· 如果操作数为无穷，结果为相反符号的无穷。</li> <li>· 如果操作数为零，结果为相反符号的零。</li> </ul>	

## drem

<b>操作</b>	求 double 余数。	
<b>格式</b>	<table border="1"><tr><td>drem</td></tr></table>	drem
drem		
<b>形式</b>	drem=115(0x73)	
<b>栈</b>	..., value1.word1, value1.word2, value2.word1, value2.word2⇒	

…, rssult. word1, result. word2

**描述** value1 和 value2 必须都是 double 类型。这些值从操作数栈弹出。result 被计算并作为 double 压入操作数栈。

drem 指令的结果同 IEEE 754 定义的所谓余数操作不同。IEEE 754“余数”操作从舍入除法计算余数，而不是从截断除法计算，所以其行为同通常的整数余数操作符不相似。相反，Java 虚拟机将 drem 行为定义为同 Java 虚拟机整数余数指令 (irem 和 lrem) 的方式相似；这可以同 C 的库函数 fmod 相比较。

drem 指令的结果由这些规则控制：

- 如果任一值为 NaN，结果为 NaN。
- 如果值均不是 NaN，结果的符号等于被除数的符号。
- 如果被除数是无穷，或者除数为零，或两者兼有，则结果为 NaN。
- 如果被除数是有限的而除数是无穷，结果等于被除数。
- 如果被除数为零并且除数是有限的，结果等于被除数。
- 在其他情况下，不涉及无穷，零或 NaN，被除数 value1 和除数 value2 的浮点余数 result 由数学关系  $result = value1 - (value2 \cdot q)$  定义，其中  $q$  是整数，仅当  $value1/value2$  为负时为负，仅当  $value1/value2$  为正时为正，其数量在不超出  $value1$  和  $value2$  的真正的数学商的数量的情况下尽可能大。尽管可能出现被零除，drem 指令的求值从不抛出运行期异常。不会出现溢出，下溢和丢失精度。

**注解** IEEE 754 余数操作可以通过 Java 库例程 Math. IEEEremainder 计算。

---

### dreturn

---

**操作** 从方法返回 double。

**格式** dreturn

**形式** dreturn=175(0xaf)

**栈** …, value. word1, value. word2 ⇒  
[empty]

**描述** 返回方法必须具有 double 返回类型。value 必须是 double 类型。value 从当前框架（§ 3.6）的操作数栈弹出，并压入调用者框架的操作数栈。当前方法的操作数栈上的任何其他值都被抛弃。如果返回方法是 synchronized 方法，调用该方法时获得的或重入的监视器被释放或退出（分别地），好像执行了

`monitorexit` 指令。

然后解释器将控制返回给该方法的调用者,恢复调用者的框架。

---

## **dstore**

---

**操作** 将 double 存储到局部变量。

**格式**

dstore
index

**形式** `dstore=57(0x39)`

**栈** `..., value.word1, value.word2⇒`  
...

**描述** `index` 是无符号字节。`index` 和 `index+1` 必须都是当前框架(§ 3.6)的局部变量的有效索引。操作数栈上的 `value` 必须是 double 类型。它从操作数栈弹出,并且 `index` 和 `index+1` 处的局部变量被设置为 `value`。

**注解** `dstore` 操作码可以同 `wide` 指令一同使用,来用两字节无符号索引访问局部变量。

---

## **dstore\_<n>**

---

**操作** 将 double 存储到局部变量。

**格式**

<code>dstore_&lt;n&gt;</code>
-------------------------------

**形式** `dstore_0=71(0x47)`  
`dstore_1=72(0x48)`  
`dstore_2=73(0x49)`  
`dstore_3=74(0x4a)`

**栈** `..., value.word1, value.word2⇒`  
...

**描述** `<n>` 和 `<n>+1` 必须都是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈上的 `value` 必须是 double 类型。它从操作数栈弹出,并且 `<n>` 和 `<n>+1` 处

的局部变量被设置为 value。

**注解** 每个 `dstore_{n}` 指令同带 `index{n}` 的 `dstore` 相同, 只不过操作数 `{n}` 是隐式的。

---

## dsub

---

**操作** double 相减。

**格式**

dsub
------

**形式** `dsub=103(0x67)`

**栈**  $\dots, \text{value1.word1}, \text{value1.word2}, \text{value2.word1}, \text{value2.word2} \Rightarrow \dots, \text{result.word1}, \text{result.word2}$

**描述** `value1` 和 `value2` 必须都是 double 类型。这些值从操作数栈弹出。`double` 类型的 `result` 为 `value1 - value2`, `result` 被压入操作数栈。对于 `double` 减法, 总有  $a - b$  同  $a + (-b)$  的结果相同。然而, 对于 `dsub` 指令, 从零减去并不与相反数相同, 因为如果 `x` 为 `0.0`, 则 `0.0 - x` 等于 `+0.0`, 而 `-x` 等于 `-0.0`。Java 虚拟机需要 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出, 下溢, 或者丢失精度, `dsub` 指令的执行从不抛出运行期异常。

---

## dup

---

**操作** 复制操作数栈顶的字。

**格式**

dup
-----

**形式** `dup=89(0x59)`

**栈**  $\dots, \text{word} \Rightarrow \dots, \text{word}, \text{word}$

**描述** 操作数栈顶的字被复制并压入操作数栈。  
除非 `word` 包含 32 位的数据类型, 不要使用 `dup` 指令。

**注解** 除保持 64 位数据类型完整性的限制外, `dup` 指令对无类型字操作, 忽略它所包含的数据的类型。

---

## **dup\_x1**

---

<b>操作</b>	复制操作数栈顶字并放到向下两个字处。	
<b>格式</b>	<table border="1"><tr><td>dup_x1</td></tr></table>	dup_x1
dup_x1		
<b>形式</b>	dup_x1=90(0x5a)	
<b>栈</b>	..., word2, word1⇒ ..., word1, word2, word1	
<b>描述</b>	操作数栈顶的字被复制,并且该拷贝被插入到操作数栈自下两个字的位置上。 除非 word1和 word2的每一个都是包含32位数据类型的字,不要使用 dup_x1指令。	
<b>注解</b>	除保持64位数据类型的完整性的限制外,dup_x1指令对无类型字操作,忽略它们所包含的数据的类型。	

---

## **dup\_x2**

---

<b>操作</b>	复制操作数栈顶的字并放在向下三个字处。	
<b>格式</b>	<table border="1"><tr><td>dup_x2</td></tr></table>	dup_x2
dup_x2		
<b>形式</b>	dup_x2=91(0x5b)	
<b>栈</b>	..., word3, word2, word1⇒ ..., word1, word3, word2, word1	
<b>描述</b>	操作数栈顶的字被复制,并且该拷贝被插入到操作数栈向下三个字的位置上。 除非 word2和 word3的每一个都是包含32位数据类型的字或者合在一起是64位数据的两个字,或除非 word1包含一个32位数据类型,不要使用 dup_x2指令。	
<b>注解</b>	除保持64位数据类型完整性的限制外,dup_x2指令对无类型字操作,忽略它们所包含的数据的类型。	

---

## dup2

---

操作	复制操作数栈顶的两个字。	
格式	<table border="1"><tr><td>dup2</td></tr></table>	dup2
dup2		
形式	dup2=92(0x5c)	
栈	…, word2, word1⇒ …, word2, word1, word2, word1	
描述	操作数栈顶的两个字被复制并按原顺序压入操作数栈。 除非 word1和 word2都是包含32位数据类型的字或者合在一起是一个64位数据的两个字,不要使用 dup2指令。	
注解	除保持64位数据类型完整性的限制外,dup2指令对无类型字操作,忽略它们所包含的数据的类型。	

---

## dup2\_x1

---

操作	复制操作数栈顶的两个字并放在向下三个字处。	
格式	<table border="1"><tr><td>dup2_x1</td></tr></table>	dup2_x1
dup2_x1		
形式	dup2_x1=93(0x5d)	
栈	…, word3, word2, word1⇒ …, word2, word1, word3, word2, word1	
描述	操作数栈顶的两个字被复制并且其拷贝以原顺序插入到操作数栈向下三个字的位置上。 除非 word1和 word2都是包含32位数据类型的字或者合在一起是包含一个64位数据的两个字,或者除非 word3是包含32位数据类型的字,不要使用 dup2_x1指令。	
注解	除保持64位数据类型完整性的限制外,dup2_x1指令对无类型字操作,忽略它们所包含的数据的类型。	

---

## **dup2\_x2**

---

**操作**            复制操作数栈顶的两个字并放在向下四个字处。

**格式**

dup2_x2
---------

**形式**            dup2\_x2=94(0x5e)

**栈**             $\dots, \text{word}_4, \text{word}_3, \text{word}_2, \text{word}_1 \Rightarrow$   
 $\dots, \text{word}_2, \text{word}_1, \text{word}_4, \text{word}_3, \text{word}_2, \text{word}_1$

**描述**            操作数栈顶的两个字被复制并按原顺序插入操作数栈向下四个字的位置上。除非 word1 和 word2 都是 32 位数据类型或者合在一起是一个 64 位数据的两个字，或者除非 word3 和 word4 都是包含 32 位数据类型的字或者合在一起是一个 64 位数据的两个字，不要使用 dup2\_x2 指令。

**注解**            除保持 64 位数据类型完整性的限制外，dup2\_x2 指令对无类型字操作，忽略它们所包含的数据的类型。

---

## **f2d**

---

**操作**            将 float 转换为 double。

**格式**

f2d
-----

**形式**            f2d=141(0x8d)

**栈**             $\dots, \text{value} \Rightarrow$   
 $\dots, \text{result}. \text{word}_1, \text{result}. \text{word}_2$

**描述**            操作数栈顶的 value 必须是 float 类型。它从操作数栈弹出并转换为 double。result 被压入操作数栈。

**注解**            f2d 指令执行放宽的基本转换（§ 2.6.2）。因为 float 类型的所有值可由 double 类型精确地表示，所以转换是精确的。

---

**f2i**

---

操作	将 float 转换为 int。	
格式	<table border="1"><tr><td>f2i</td></tr></table>	f2i
f2i		
形式	f2i=139(0x8b)	
栈	..., value⇒ ..., result	
描述	操作数栈顶的 value 必须是 float 类型。它从操作数栈弹出并转换为 int。result 被压入操作数栈： <ul style="list-style-type: none"><li>如果 value 为 NaN, 转换的 result 为 int 0。</li><li>否则, 如果 value 不是无穷, 它被舍入为整数值 V, 使用 IEEE 754 向零舍入模式进行向零舍入。如果整数值 V 可以表示为 int, 则 result 为 int 值 V。</li><li>否则, 或者 value 必定太小(大数量的负值或负无穷)并且 result 是 int 类型的最小可表示值; 或者 value 必定太大(大数量的正值或正无穷)并且 result 是 int 类型的最大可表示值。</li></ul>	
注解	f2i 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息, 还可能丢失精度。	

---

**f2l**

---

操作	将 float 转换为 long。	
格式	<table border="1"><tr><td>f2l</td></tr></table>	f2l
f2l		
形式	f2l=140(0x8c)	
栈	..., value⇒ ..., result.word1, result.word2	
描述	操作数栈顶的 value 必须是 float 类型。它从操作数栈弹出并转换为 long。result 被压入操作数栈： <ul style="list-style-type: none"><li>如果 Nalue 为 NaN, 转换的 result 为 long 0。</li><li>否则, 如果 Nalue 不是无穷, 它被舍入为整数值 V, 使用 IEEE 754 向零舍入模式进行向零舍入。如果整数值 V 可以表示为 long, 则 result 为 long 值 V。</li></ul>	

入模式进行向零舍入。如果整数值 V 可以表示为 long，则 result 是 long 值 V。

- 否则，或者 value 必定太小(大数量的负值或负无穷)并且 result 是 long 类型的最小可表示值，或者 value 必定太大(大数量的正值或正无穷)并且 result 是 long 类型的最大可表示值。

**注解** f2l 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息，还可能丢失精度。

---

**fadd**

---

**操作** float 相加。

**格式**

fadd
------

**形式** fadd=98(0x62)

**栈** ..., value1, value2⇒  
..., result

**描述** value1 和 value2 必须都是 float 类型。这些值从操作数栈弹出。float 类型的 result 为 value1+value2。result 被压入操作数栈。fadd 指令的结果由 IEEE 算术规则控制：

- 如果任一值为 NaN，结果为 NaN。
- 两个反号无穷的和为 NaN。
- 两个同号无穷的和为该符号的无穷。
- 无穷和任何有限值的和等于该无穷。
- 两个反号的零的和为正零。
- 两个同号的零的和为该符号的零。
- 零和非零有限值的和等于该非零值。
- 两个同数量并且符号相反的非零有限值的和为正零。
- 在其它情况下，不涉及无穷，零或 NaN，并且两个值有相同符号或不同数量，则和被计算并使用 IEEE 754 最接近舍入模式舍入为最接近的可表示值。如果数量太大以致不能表示为 float，我们说操作溢出；则结果为适当符号的无穷。如果数量太小以致不能表示为 float，我们说操作下溢；则结果为适当符号的零。

Java 虚拟机需要 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出、下溢或丢失精度，fadd 指令的执行从不抛出运行期异常。

---

## faload

---

操作	从数组装载 float。	
格式	<table border="1"><tr><td>faload</td></tr></table>	faload
faload		
形式	faload = 48(0x30)	
栈	..., arrayref, index⇒ ..., value	
描述	arrayref 必须是 reference 类型并且必须引用一个数组, 该数组成分为 float 类型。index 必须是 int 类型。arrayref 和 index 都从操作数栈弹出。数组中 index 处的成分的 float 值被获取并压入操作数栈顶。	
运行期异常	如果 arrayref 为 null, faload 抛出 NullPointerException。否则, 如果 index 不在 arrayref 所引用的数组的边界内, faload 指令抛出 ArrayIndexOutOfBoundsException。	

---

## fastore

---

操作	存储到 float 数组。	
格式	<table border="1"><tr><td>fastore</td></tr></table>	fastore
fastore		
形式	fastore = 81(0x51)	
栈	..., arrayref, index, value⇒ ...	
描述	arrayref 必须是 reference 类型并且必须引用一个数组, 该数组成分是 float 类型。index 必须是 int 类型并且 value 必须是 float 类型。arrayref、index 和 value 从操作数栈弹出。float 类型的 value 作为数组的 index 处的成分存储。	
运行期异常	如果 arrayref 为 null, fastore 抛出 NullPointerException。否则, 如果 index 不在 arrayref 所引用的数组的边界内, fastore 指令抛出 ArrayIndexOutOfBoundsException。	

---

## **fcmp<op>**

---

<b>操作</b>	比较 float。	
<b>格式</b>	<table border="1"><tr><td><b>fcmp&lt;op&gt;</b></td></tr></table>	<b>fcmp&lt;op&gt;</b>
<b>fcmp&lt;op&gt;</b>		
<b>形式</b>	<b>fcmpg</b> =150(0x96) <b>fcmpl</b> =149(0x95)	
<b>栈</b>	..., value1, value2⇒ ..., result	
<b>描述</b>	value1和 value2必须都是 float 类型。这些值从操作数栈弹出，并进行浮点数比较。如果 value1大于 value2,int 值1被压入操作数栈。如果 value1等于 value2,int 值0被压入操作数栈。如果 value1小于 value2,int 值−1被压入操作数栈。如果 value1或 value2为 NaN,fcmpg 指令将 int 值1压入操作数栈，而 fcmpl 指令将 int 值−1压入操作数栈。 浮点数比较按 IEEE 754执行。除 NaN 外的所有值被排序，负无穷小于所有有限值，正无穷大于所有有限值。正零和负零被认为相等。	
<b>注解</b>	fcmpg 和 fcmpl 指令仅在对涉及 NaN 的比较的处理上不同。NaN 不排序，所以如果操作数之一或全部是 NaN 则 float 比较失败。在 fcmpg 和 fcmpl 均可用时，任何 float 比较可以编译为向操作数栈压入相同的 result，无论比较在非 NaN 值上失败还是因为遇到 NaN 而失败。更多信息见7.5节“更多控制范例。”	

---

## **fconst\_<f>**

---

<b>操作</b>	压入 float。	
<b>格式</b>	<table border="1"><tr><td><b>fconst_&lt;f&gt;</b></td></tr></table>	<b>fconst_&lt;f&gt;</b>
<b>fconst_&lt;f&gt;</b>		
<b>形式</b>	<b>fconst_0</b> =11(0xb) <b>fconst_1</b> =12(0xc) <b>fconst_2</b> =13(0xd)	
<b>栈</b>	...⇒ ..., <f>	

**描述** 将 float 常数 $\langle f \rangle$ (0.0、1.0 或 2.0)压入操作数栈。

---

### fdiv

---

**操作** float 相除。

**格式**

fdiv
------

**形式**  $\text{fdiv} = 110(0x6e)$

**栈**  
..., value1, value2  $\Rightarrow$   
..., result

**描述** value1 和 value2 必须都是 float 类型。这些值从操作数栈弹出。float 类型的 result 为 value1/value2。result 被压入操作数栈。fdiv 指令的结果由 IEEE 算术规则控制：

- 如果任一值为 NaN, result 为 NaN。
- 如果值均不是 NaN, 如果两值同号则结果的符号为正, 如果两值反号则为负。
- 无穷被无穷除导致 NaN。
- 无穷被有限值除导致有符号无穷, 遵循刚给出的符号生成规则。
- 有限值被无穷除导致有符号的零, 遵循刚给出的符号生成规则。
- 零被零除导致 NaN; 零被其他任何有限值除导致有符号的零, 遵循刚给出的符号生成规则。
- 非零有限值被零除导致有符号无穷, 遵循刚给出的符号生成规则。
- 在其他情况下, 不涉及无穷, 零或 NaN, 商被计算并使用 IEEE 754 最接近舍入模式舍入为最接近的 float。如果数量太大以致不能表示为 float, 我们说操作溢出; 则结果为适当符号的无穷。如果数量太小以致不能表示为 float, 我们说操作下溢; 则结果为适当符号的零。

Java 虚拟机需要 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出, 下溢, 被零除或丢失精度, fdiv 指令的执行从不抛出运行期异常。

---

### fload

---

**操作** 从局部变量装载 float。

**格式**

fload
index

<b>形式</b>	<code>fload=23(0x17)</code>
<b>栈</b>	$\dots \Rightarrow$ $\dots, \text{value}$
<b>描述</b>	<code>index</code> 是一个无符号字节, 必须是对当前框架(§ 3.6)的局部变量的有效索引。 <code>index</code> 处的局部变量必须包含 <code>float</code> 。 <code>index</code> 处局部变量的 <code>value</code> 被压入操作数栈。
<b>注解</b>	<code>fload</code> 操作码可同 <code>wide</code> 指令一同使用, 来用两字节无符号索引访问局部变量。

---

### **fload\_<n>**

---

<b>操作</b>	从局部变量部装载 <code>float</code> 。
<b>格式</b>	<code>fload_&lt;n&gt;</code>
<b>形式</b>	<code>fload_0=34(0x22)</code> <code>fload_1=35(0x23)</code> <code>fload_2=36(0x24)</code> <code>fload_3=37(0x25)</code>
<b>栈</b>	$\dots \Rightarrow$ $\dots, \text{value}$
<b>描述</b>	<code>&lt;n&gt;</code> 必须是对当前框架(§ 3.6)的局部变量的有效索引。 <code>&lt;n&gt;</code> 处的局部变量必须包含一个 <code>float</code> 。 <code>&lt;n&gt;</code> 处局部变量的 <code>value</code> 被压入操作数栈。
<b>注解</b>	每个 <code>fload_&lt;n&gt;</code> 指令和带 <code>index &lt;n&gt;</code> 的 <code>fload</code> 相同, 只不过操作数 <code>&lt;n&gt;</code> 是隐式的。

---

### **fmul**

---

<b>操作</b>	<code>float</code> 相乘。
<b>格式</b>	<code>fmul</code>
<b>形式</b>	<code>fmul=106(0x6a)</code>

<b>栈</b>	$\dots, \text{value1}, \text{value2} \Rightarrow$ $\dots, \text{result}$
<b>描述</b>	<p><code>value1</code>和<code>value2</code>必须都是<code>float</code>类型。这些值从操作数栈弹出。<code>float</code>类型的<code>result</code>为<math>\text{value1} * \text{value2}</math>。<code>result</code>被压入操作数栈。<code>fmul</code>指令的结果由 IEEE 算术规则控制：</p> <ul style="list-style-type: none"> <li>如果任一值为<code>NaN</code>,结果为<code>NaN</code>。</li> <li>如果值均不为<code>NaN</code>,如果两值同号则结果的符号为正,如果两值反号则为负。</li> <li>无穷与零相乘导致<code>NaN</code>。</li> <li>无穷与有限值相乘导致有符号的无穷,遵循刚给出的符号生成规则。</li> <li>在其他情况下,不涉及无穷或<code>NaN</code>,乘积被计算并使用 IEEE 754 最接近舍入模式舍入为最接近的可表示值。如果数量太大以致不能表示为<code>float</code>,我们说操作溢出;则结果为适当符号的无穷。如果数量太小以致不能表示为<code>float</code>,我们说操作下溢;则结果为适当符号的零。</li> </ul> <p>Java 虚拟机需要 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出、下溢或丢失精度,<code>fmul</code>指令的执行从不抛出运行期异常。</p>

---

### **fneg**

---

<b>操作</b>	对 <code>float</code> 求反。	
<b>格式</b>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="padding: 2px 10px;">fneg</td></tr></table>	fneg
fneg		
<b>形式</b>	<code>fneg = 118(0x76)</code>	
<b>栈</b>	$\dots, \text{value} \Rightarrow$ $\dots, \text{result}$	
<b>描述</b>	<p><code>value</code>必须是<code>float</code>类型。它从操作数栈弹出。<code>float</code>类型的<code>result</code>是<code>value</code>的算术相反数,<math>-value</math>。<code>result</code>被压入操作数栈。对<code>float</code>值,相反数同从零减去不同。如果<code>x</code>为<code>0.0</code>,则<code>0.0 - x</code>等于<code>+0.0</code>,但<code>-x</code>等于<code>-0.0</code>。单目减仅取反<code>float</code>的符号。特别关心的情形：</p> <ul style="list-style-type: none"> <li>如果操作数为<code>NaN</code>,结果为<code>NaN</code>(记起<code>NaN</code>无符号)。</li> <li>如果操作数为无穷,结果为相反符号的无穷。</li> <li>如果操作数为零,结果为相反符号的零。</li> </ul>	

---

## frem

---

操作	求 float 余数。
格式	<div style="border: 1px solid black; padding: 2px; display: inline-block;">frem</div>
形式	frem = 114(0x72)
栈	..., value1, value2 => ..., result
描述	<p>value1 和 value2 必须都是 float 类型。这些值从操作数栈弹出。result 被计算并作为 float 压入操作数栈。</p> <p>frem 指令的 result 同 IEEE 754 定义的所谓余数操作不同。IEEE 754 “余数”操作从舍入除法而不是截断除法计算余数，所以其行为同通常的整数余数操作符不相似。相反，Java 虚拟机将 frem 行为定义为同 Java 虚拟机整数余数指令 (irem 和 lrem) 的方式相似；这可以同 C 的库函数 fmod 相比较。</p> <p>frem 指令的结果由这些规则控制：</p> <ul style="list-style-type: none"><li>如果任一值为 NaN，结果为 NaN。</li><li>如果值均不是 NaN，结果的符号等于被除数的符号。</li><li>如果被除数是无穷，或者除数为零，或两者兼有，结果为 NaN。</li><li>如果被除数为有限而除数为无穷，结果等于被除数。</li><li>如果被除数为零而除数为有限，结果等于被除数。</li><li>在其他情况下，不涉及无穷，零或 NaN，被除数 value1 和除数 value2 的浮点余数 result 由数学关系式 <math>result = value1 - (value2 \cdot q)</math> 定义，其中 q 是整数，仅当 <math>value1/value2</math> 为负时为负，仅当 <math>value1/value2</math> 为正时为正，其数量在不超出 value1 和 value2 的真正的数学商的数量的情况下尽可能大。</li></ul> <p>尽管可能出现被零除，frem 指令的求值从不抛出运行期异常。不会出现溢出，下溢或丢失精度。</p>

注解 IEEE 754 余数操作可以通过 Java 库例程 Math.IEEEremainder 计算。

---

## freturn

---

操作	从方法返回 float。
格式	<div style="border: 1px solid black; padding: 2px; display: inline-block;">freturn</div>

---

<b>形式</b>	<code>freturn=174(0xae)</code>
<b>栈</b>	<code>..., value⇒</code> <code>[empty]</code>
<b>描述</b>	返回方法必须具有 float 返回类型。 <code>value</code> 必须是 float 类型。 <code>value</code> 从当前框架(§ 3.6)的操作数栈弹出并压入到调用者框架的操作数栈。当前方法的操作数栈上的其他任何值都被抛弃。如果返回方法是 synchronized 方法，则调用方法时获得的或重入的监视器被释放或退出(分别地)，好像执行了 <code>monitorexit</code> 指令。然后解释器将控制返回给方法的调用者，恢复调用者的框架。

---

### **fstore**

---

<b>操作</b>	将 float 存储到局部变量。		
<b>格式</b>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">fstore</td> </tr> <tr> <td style="text-align: center;">index</td> </tr> </table>	fstore	index
fstore			
index			
<b>形式</b>	<code>fstore=56(0x38)</code>		
<b>栈</b>	<code>...,value⇒</code> ...		
<b>描述</b>	<code>index</code> 是一个无符号字节，必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 <code>value</code> 必须是 float 类型。它从操作数栈弹出，同时 <code>index</code> 处的局部变量的值被设置为 <code>value</code> 。		
<b>注解</b>	<code>fstore</code> 操作码可以同 <code>wide</code> 指令一同使用，来用两字节无符号索引访问局部变量。		

---

### **fstore\_{n}**

---

<b>操作</b>	将 float 存储到局部变量。	
<b>格式</b>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;"><code>fstore_{n}</code></td> </tr> </table>	<code>fstore_{n}</code>
<code>fstore_{n}</code>		
<b>形式</b>	<code>fstore_0=67(0x43)</code> <code>fstore_1=68(0x44)</code>	

fstore\_2=69(0x45)  
fstore\_3=70(0x46)

**栈**       $\dots, \text{value} \Rightarrow$   
               $\dots$

**描述**       $\langle n \rangle$  必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 value 必须是 float 类型。它从操作数栈弹出, 同时 $\langle n \rangle$ 处局部变量的值被设置为 value。

**注解**      每个 fstore\_ $\langle n \rangle$  同带 index $\langle n \rangle$  的 fstore 相同, 只不过操作数 $\langle n \rangle$  是隐式的。

---

## fsub

---

**操作**      float 相减。

**格式**

fsub
------

**形式**      fsub=102(0x66)

**栈**       $\dots, \text{value1}, \text{value2} \Rightarrow$   
               $\dots, \text{result}$

**描述**      value1 和 value2 必须都是 float 类型。这些值从操作数栈弹出。float 类型的 result 为 value1 - value2, result 被压入操作数栈。对 float 减法,  $a - b$  总是产生与  $a + (-b)$  相同的结果。然而, 对于 fsub 指令, 从零减去和取反不同, 因为如果  $x$  是  $+0.0$ , 则  $0.0 - x$  等于  $+0.0$ , 但  $-x$  等于  $-0.0$ 。  
Java 虚拟机需要 IEEE 754 所定义的逐渐下溢的支持。尽管可能出现溢出, 下溢或丢失精度, fsub 指令的执行从不抛出运行期异常。

---

## getfield

---

**操作**      从对象中获取域。

**格式**

getfield
indexbyte1
indexbyte2

<b>形式</b>	<code>getfield=180(0xb4)</code>
<b>栈</b>	<p>..., objectref⇒      ..., value      OR</p>
<b>栈</b>	<p>..., objectref⇒      ..., value.word1, value.word2</p>
<b>描述</b>	<p>objectref 必须是 reference 类型, 它从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引为(indexbyte1&lt;&lt;8) indexbyte2。索引处的常数池项必须是 CONSTANT_Fieldref(§ 4.4.2), 对类名和域名的引用。如果该域为 protected(§ 4.6), 则它必须是当前类的成员或者是当前类的超类的成员, 而 Objectref 的类必须是当前类或当前类的子类。</p> <p>该项被解析(§ 5.2), 确定域宽度和域偏移量。objectref 所引用的类实例的该偏移量处的 value 被获取并压入操作数栈。</p>
<b>链接异常</b>	<p>在解析 CONSTANT_Fieldref 常数池项时, § 5.2 中的任何错误都可能被抛出。</p> <p>否则, 如果指定的域存在但为 static 域, getField 抛出 IncompatibleClassChangeError。</p>
<b>运行期异常</b>	否则, 如果 objectref 为 null, getField 指令抛出 NullPointerException。
<b>注解</b>	getField 指令对一个或两个字宽的域都可进行操作。

---

### **getstatic**

---

**操作** 从类中获取 static 域。

**格式**

<code>getstatic</code>
<code>indexbyte1</code>
<code>indexbyte2</code>

**形式** `getstatic=178(0xb2)`

**栈**  $\dots \Rightarrow$   
 $\dots, \text{value}$

或  
栈  
 $\dots, \Rightarrow$   
 $\dots, \text{value.word1}, \text{value.word2}$

**描述** 无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引为  $(\text{indexbyte1} \ll 8) | \text{indexbyte2}$ 。索引处的常数池项必须是 CONSTANT\_Fieldref(§ 4.4.2), 对类名和域名的引用。如果该域为 protected(§ 4.6), 则它必须是当前类的成员或者是当前类的超类的成员。该项被解析(§ 5.2), 确定类域及其宽度。类域的 value 被获取并压入操作数栈。

**链接异常** 在解析 CONSTANT\_Fieldref 常数池项时, § 5.2 中的任何异常都可能被抛出。  
否则, 如果指定的域存在但不是 static(类)域, getstatic 抛出 Incompatible-ClassChangeError。

**注解** getstatic 指令对一个和两个字宽的域都可进行操作。

---

## goto

---

**操作** 恒转移。

**格式**

goto
branchbyte1
branchbyte2

**形式** goto=167(0xa7)

**栈** 不变。

**描述** 无符号字节 branchbyte1 和 branchbyte2 用于构造有符号的 16 位 branchoffset, 其中 branchoffset 为  $(\text{branchbyte1} \ll 8) | \text{branchbyte2}$ 。从该 goto 指令的操作码地址起的偏移量处开始执行。目标地址必须是包含此 goto 指令的方法内的一条指令的操作码的地址。

---

## **goto\_w**

---

**操作** 恒转移(宽索引)。

**格式**

goto_w
branchbyte1
branchbyte2
branchbyte3
branchbyte4

**形式**  $\text{goto\_w} = 200(0xc8)$

**栈** 无变化。

**描述** 无符号字节 branchbyte1、branchbyte2、branchbyte3 和 branchbyte4 用于构造有符号的 32 位 branchoffset，其中 branchoffset 为  $(\text{branchbyte1} \ll 24) | (\text{branchbyte2} \ll 16) | (\text{branchbyte3} \ll 8) | \text{branchbyte4}$ 。从该 goto\_w 指令的操作码地址起的偏移量处开始执行。目标地址必须是包含此 goto\_w 指令的方法内的一条指令的操作码。

**注解** 虽然 goto\_w 指令有 4 字节转移偏移量，但其他因素将 Java 方法的长度限制为 65 535 字节（§ 4.10）。这种限制可能在 Java 虚拟机的将来版本中被解除。

---

## **i2b**

---

**操作** 将 int 转换为 byte。

**格式**

i2b

**形式**  $\text{i2b} = 145(0x91)$

**栈**  
..., value  $\Rightarrow$   
..., result

**描述** 操作数栈顶的 value 必须是 int 类型。它从操作数栈弹出，截断为 byte，然后扩展为 int 类型的 result。result 被压入操作数栈。

**注解** i2b 指令执行缩窄的基本转换（§ 2.6.3）。它可能丢失 value 的总的数量信息。

息。`result` 还可能同 `value` 有不同的符号。

---

### i2c

---

操作	将 int 转换为 char。	
格式	<table border="1"><tr><td>i2c</td></tr></table>	i2c
i2c		
形式	$i2c = 146(0x92)$	
栈	..., value⇒ ..., result	
描述	操作数栈顶的 <code>value</code> 必须是 int 类型。它从操作数栈弹出，截断为 char，然后零扩展为 int 类型的 <code>result</code> 。 <code>result</code> 被压入操作数栈。	
注解	<code>i2c</code> 指令执行缩窄的基本转换（§ 2.6.3）。它可能丢失 <code>value</code> 的总的数量信息。 <code>result</code> （总为正）还可能同 <code>value</code> 有不同的符号。	

---

### i2d

---

操作	将 int 转换为 double。	
格式	<table border="1"><tr><td>i2d</td></tr></table>	i2d
i2d		
形式	$i2d = 135(0x87)$	
栈	..., value⇒ ..., <code>result.word1</code> , <code>result.word2</code>	
描述	操作数栈顶的 <code>value</code> 必须是 int 类型。它从操作数栈弹出并转换为 double 类型的 <code>result</code> 。 <code>result</code> 被压入操作数栈。	
注解	<code>i2d</code> 指令执行放宽的基本转换（§ 2.6.2）。因为 int 类型的所有值可由 double 类型精确表示，转换是精确的。	

---

**i2f**

---

<b>操作</b>	将 int 转换为 float。	
<b>格式</b>	<table border="1"><tr><td>i2f</td></tr></table>	i2f
i2f		
<b>形式</b>	i2f=134(0x86)	
<b>栈</b>	..., value⇒ ..., result	
<b>描述</b>	操作数栈顶的 value 必须是 int 类型。它从操作数栈弹出并使用 IEEE 754 最接近舍入模式转换为 float 类型的 result。result 被压入操作数栈。	
<b>注解</b>	i2f 执行放宽的基本转换(§ 2.6.2)，但可能导致丢失精度，因为 float 类型只有24个尾数位。	

---

**i2l**

---

<b>操作</b>	将 int 转换为 long。	
<b>格式</b>	<table border="1"><tr><td>i2l</td></tr></table>	i2l
i2l		
<b>形式</b>	i2l=133(0x85)	
<b>栈</b>	..., value⇒ ..., result.word1, result.word2	
<b>描述</b>	操作数栈项的 value 必须是 int 类型。它从操作数栈弹出并有符号扩展为 long 类型的 result。result 被压入操作数栈。	
<b>注解</b>	i2l 指令执行放宽的基本转换(§ 2.6.2)。因为 int 类型的所有值可由 long 类型精确表示，转换是精确的。	

---

**i2s**

---

<b>操作</b>	将 int 转换为 short。	
<b>格式</b>	<table border="1"><tr><td>i2s</td></tr></table>	i2s
i2s		

<b>形式</b>	i2s = 147(0x93)
<b>栈</b>	..., value⇒ ..., result
<b>描述</b>	操作数栈顶的 value 必须是 int 类型。它从操作数栈弹出，截断为 short，然后有符号扩展为 int 类型的 result。result 被压入操作数栈。
<b>注解</b>	i2s 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信息。result 还可能同 value 有不同的符号。

---

### iadd

---

<b>操作</b>	int 相加。
<b>格式</b>	iadd
<b>形式</b>	iadd = 96(0x60)
<b>栈</b>	..., value1, value2⇒ ..., result
<b>描述</b>	value1 和 value2 必须都是 int 类型。这些值从操作数栈弹出。int 类型的 result 为 value1 + value2。result 被压入操作数栈。如果 iadd 溢出，则 result 为真正的数学结果的低序位，采用二进制补码格式。如果出现溢出，则结果的符号将与两值的数学和的符号不同。

---

### iaload

---

<b>操作</b>	从数组装载 int。
<b>格式</b>	iaload
<b>形式</b>	iaload = 46(0x2e)
<b>栈</b>	..., arrayref, index⇒ ..., value
<b>描述</b>	arrayref 必须是 reference 类型，并且必须引用一个数组，该数组成分为 int

类型。`index` 必须是 `int` 类型。`arrayref` 和 `index` 都从操作数栈弹出。数组中 `index` 处的成分的 `int` 类型的 `value` 被获取并压入操作数栈顶。

<b>运行期异常</b>	如果 <code>arrayref</code> 为 <code>null</code> , <code>iaload</code> 抛出 <code>NullPointerException</code> 。否则, 如果 <code>index</code> 不在 <code>arrayref</code> 所引用的数组的边界内, <code>iaload</code> 指令抛出 <code>ArrayIndexOutOfBoundsException</code> 。
--------------	--

---

### iand

---

<b>操作</b>	<code>int</code> 布尔“与”。	
<b>格式</b>	<table border="1"><tr><td><code>iand</code></td></tr></table>	<code>iand</code>
<code>iand</code>		
<b>形式</b>	<code>iand = 126(0x7e)</code>	
<b>栈</b>	<code>..., value1, value2 =&gt;</code> <code>..., result</code>	
<b>描述</b>	<code>value1</code> 和 <code>value2</code> 必须都是 <code>int</code> 类型。它们从操作数栈弹出。通过对 <code>value1</code> 和 <code>value2</code> 按位“与”来计算出 <code>int</code> 类型的 <code>result</code> 。 <code>result</code> 被压入操作数栈。	

---

### iastore

---

<b>操作</b>	存储到 <code>int</code> 数组。	
<b>格式</b>	<table border="1"><tr><td><code>iastore</code></td></tr></table>	<code>iastore</code>
<code>iastore</code>		
<b>形式</b>	<code>iastore = 79(0x4f)</code>	
<b>栈</b>	<code>..., arrayref, index, value =&gt;</code> <code>...</code>	
<b>描述</b>	<code>arrayref</code> 必须是 <code>reference</code> 类型并且必须引用一个数组, 该数组成分是 <code>int</code> 类型。 <code>index</code> 和 <code>value</code> 都必须是 <code>int</code> 类型。 <code>arrayref</code> 、 <code>index</code> 和 <code>value</code> 从操作数栈弹出。 <code>int</code> 类型的 <code>value</code> 作为数组的 <code>index</code> 处的成分存储。	
<b>运行期异常</b>	如果 <code>arrayref</code> 为 <code>null</code> , <code>iastore</code> 抛出 <code>NullPointerException</code> 。否则, 如果 <code>index</code> 不在 <code>arrayref</code> 所引用的数组的边界内, <code>iastore</code> 指令抛出 <code>ArrayIndexOutOfBoundsException</code> 。	

---

## iconst\_{i}

---

**操作** 压入 int 常数。

**格式** iconst\_{i}

**形式** `iconst_m1=2(0x2)`  
`iconst_0=3(0x3)`  
`iconst_1=4(0x4)`  
`iconst_2=5(0x5)`  
`iconst_3=6(0x6)`  
`iconst_4=7(0x7)`  
`iconst_5=8(0x8)`

**栈**  $\dots \Rightarrow \dots, \langle i \rangle$

**描述** 将 int 常数  $\langle i \rangle$  (-1、0、1、2、3、4或5) 压入操作数栈。

**注解** 这个指令族的每一个,对于各自的值  $\langle i \rangle$ ,都等价于 bipusb  $\langle i \rangle$ ,只不过操作数  $\langle i \rangle$  是隐式的。

---

## idiv

---

**操作** int 相除。

**格式** idiv

**形式** `idiv=108(0x6c)`

**栈**  $\dots, \text{value1}, \text{value2} \Rightarrow \dots, \text{result}$

**描述**  $\text{value1}$ 和 $\text{value2}$ 必须都是 int 类型。这些值从操作数栈弹出。int 类型的  $\text{value}$  是 Java 表达式  $\text{value1}/\text{value2}$  的值。 $\text{result}$  被压入操作数栈。  
int 除法向零舍入;即, $n/d$  中均为 int 值,其商为 int 值  $q$ , $q$  的数量在满足  $|d \cdot q| \leq |n|$  的情况下尽可能大。此外,当  $|n| \geq |d|$  并且  $n$  和  $d$  同号时  $q$  为正,但当  $|n| \geq |d|$  并且  $n$  和  $d$  反号时  $q$  为负。

有一种特殊情况不满足此规则：如果被除数是 int 类型的具有最大可能数量的负整数，并且除数为 -1，则出现溢出，并且结果等于被除数。尽管有溢出，在此情况下无异常抛出。

**运行期异常** 如果在一个 int 除法中的除数值为 0，idiv 抛出 ArithmeticException。

---

### if\_acmp<cond>

---

**操作** 如果 reference 比较成功则转移。

**格式**

if_acmp<cond>
branchbyte1
branchbyte2

**形式** if\_acmpeq=165(0xa5)  
if\_acmpne=166(0xa6)

**栈** ..., value1, value2⇒  
...

**描述** value1 和 value2 必须都是 reference 类型。它们都从操作数栈弹出并被比较。比较结果如下：

- eq 当且仅当 value1=value2 时成功。
- ne 当且仅当 value1≠value2 时成功。

如果比较成功，无符号的 branchbyte1 和 branchbyte2 用于构造有符号的 16 位偏移量，其中偏移量被计算为 (branchbyte1<<8) | branchbyte2。从该 if\_acmp<cond> 指令操作码的地址起的偏移量处开始执行。目标地址必须是包含此 if\_acmp<cond> 指令的方法内的一条指令的操作码的地址。

否则，如果比较失败，从紧接着此 if\_acmp<cond> 指令的指令的地址处开始执行。

---

### if\_icmp<cond>

---

**操作** 如果 int 比较成功则转移。

**格式**

if_icmp<cond>
branchbyte1
branchbyte2

<b>形式</b>	if_icmp eq = 159(0x9f) if_icmp ne = 160(0xa0) if_icmp lt = 161(0xa1) if_icmp ge = 162(0xa2) if_icmp gt = 163(0xa3) if_icmp le = 164(0xa4)
<b>栈</b>	..., value1, value2 ⇒ ...
<b>描述</b>	<p>value1 和 value2 必须都是 int 类型。它们都从操作数栈弹出并被比较。所有的比较都是有符号的。比较结果如下：</p> <ul style="list-style-type: none"> <li>• eq 当且仅当 value1 = value2 时成功。</li> <li>• ne 当且仅当 value1 ≠ value2 时成功。</li> <li>• lt 当且仅当 value1 &lt; value2 时成功。</li> <li>• le 当且仅当 value1 ≤ value2 时成功。</li> <li>• gt 当且仅当 value1 &gt; value2 时成功。</li> <li>• ge 当且仅当 value1 ≥ value2 时成功。</li> </ul> <p>如果比较成功，无符号的 branchbyte1 和 branchbyte2 用于构造有符号的 16 位偏移量，其中偏移量被计算为 (branchbyte1 ≪ 8)   branchbyte2。从此 if_icmp&lt;cond&gt; 指令操作码地址起的偏移量处开始执行。目标地址必须是包含此 if_icmp&lt;cond&gt; 指令的方法内的一条指令的操作码的地址。</p> <p>否则，从紧接着此 if_icmp&lt;cond&gt; 指令的指令的地址处开始执行。</p>

---

### if<cond>

---

**操作** 如果 int 同零比较成功则转移。

<b>格式</b>	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>if&lt;cond&gt;</td></tr> <tr><td>branchbyte1</td></tr> <tr><td>branchbyte2</td></tr> </table>	if<cond>	branchbyte1	branchbyte2
if<cond>				
branchbyte1				
branchbyte2				

<b>形式</b>	ifeq = 153(0x99) ifne = 154(0x9a) iflt = 155(0x9b) ifge = 156(0x9c) ifgt = 157(0x9d)
-----------	--

ifle = 158(0x9e)

### 栈

..., value⇒

...

### 描述

value 必须是 int 类型。它从操作数栈弹出并同零比较。所有比较都是有符号的。比较结果如下：

- eq 当且仅当 value=0时成功。
- ne 当且仅当 value≠0时成功。
- lt 当且仅当 value<0时成功。
- le 当且仅当 value≤0时成功。
- gt 当且仅当 value>0时成功。
- ge 当且仅当 value≥0时成功。

如果比较成功,无符号的 branchbyte1 和 branchbyted 用于构造有符号的16位偏移量,其中偏移量被计算为(branchbyte1<<8)|branchbyte2。从该 if<cond> 指令操作码地址起的偏移量处开始执行。目标地址必须是一条指令的操作码的地址,该指令在包含此 if<cond> 指令的方法内。

否则,从紧接着该 if<cond> 指令的指令的地址处开始执行。

---

## ifnonnull

---

### 操作

如果 reference 不为 null 则转移。

### 格式

ifnonnull
branchbyte1
branchbyte2

### 形式

ifnonnull = 199(0xc7)

### 栈

..., value⇒

...

### 描述

value 必须是 reference 类型。它从操作数栈弹出。如果 value 不为 null,无符号的 branchbyte1 和 branchbyte2 用于构造有符号的16位偏移量,其中偏移量被计算为(branchbyte1<<8)|branchbyte2。从该 ifnonnull 指令操作码地址起的偏移量处开始执行。目标地址必须是一条指令的操作码的地址,该指令在包含此 ifnonnull 指令的方法内。

否则,从紧接着此 ifnonnull 指令的指令的地址处开始执行。

---

## **ifnull**

---

**操作** 如果 reference 为 null 则转移。

**格式**

ifnull
branchbyte1
branchbyte2

**形式** ifnull=198(0xc6)

**栈** ..., value⇒  
...

**描述** value 必须是 reference 类型。它从操作数栈弹出。如果 value 为 null, 无符号的 branchbyte1 和 branchbyte2 用于构造有符号的 16 位偏移量, 其中偏移量被计算为  $(\text{branchbyte1} \ll 8) | \text{branchbyte2}$ 。从该 ifnull 指令操作码地址起的偏移量处开始执行。目标地址必须是一条指令的操作码的地址, 该指令在包含此 ifnull 指令的方法内。  
否则, 从紧接着该 ifnull 指令的指令的地址处开始执行。

---

## **iinc**

---

**操作** 将局部变量增加常数值。

**格式**

iinc
index
const

**形式** iinc=132(0x84)

**栈** 无变化。

**描述** index 是无符号字节, 必须是对当前框架(§ 3.6)的局部变量的有效索引。const 是立即有符号字节。index 处的局部变量必须包含 int。const 值首先有符号扩展为 int, 然后 index 处的局部变量增加该数量。

**注解** iinc 操作码可同 wide 指令一同使用, 来用两字节无符号索引访问局部变量, 并增加两字节立即值。

---

## iload

---

**操作** 从局部变量装载 int。

**格式**

iload
index

**形式** iload = 21(0x15)

**栈**

...⇒  
..., value

**描述** index 是无符号字节, 必须是对当前框架(§ 3.6)局部变量的有效索引。index 处的局部变量必须包含 int。index 处局部变量的值被压入操作数栈。

**注解** iload 操作码可以用 wide 指令一同使用, 来用两字节无符号索引访问局部变量。

---

## iload\_<n>

---

**操作** 从局部变量装载 int。

**格式**

iload_<n>
-----------

**形式** iload\_0 = 26(0xla)

iload\_1 = 27(0xeb)

iload\_2 = 28(0xfc)

iload\_3 = 29(0xd)

**栈**

...⇒  
..., value

**描述** <n> 必须是对当前框架(§ 3.6)的局部变量的有效索引。<n> 处的局部变量必须包含 int。<n> 处局部变量的 value 被压入操作数栈。

**注解** 每个 iload\_<n> 指令同带 index<n> 的 iload 相同, 只不过操作数<n>是隐式的。

---

## imul

---

**操作** int 相乘。

**格式**

imul
------

**形式** imul = 104(0x68)

**栈** ..., value1, value2 =>  
..., result

**描述** value1 和 value2 必须都是 int 类型, 这些值从操作数栈弹出。int 类型的 value 为 value1 \* value2。result 被压入操作数栈。如果 int 乘法溢出, 则结果是作为 int 的数学乘积的低序位。如果出现溢出, 则结果的符号可能同两值的数学乘积的符号不同。

---

## ineg

---

**操作** 对 int 求反。

**格式**

ineg
------

**形式** ineg = 116(0x74)

**栈** ..., value =>  
..., result

**描述** value 必须是 int 类型。它从操作数栈弹出。int 类型的 result 是 value 的算术相反数, -value。result 被压入操作数栈。对 int 值, 取反数与从零减去相同。因为 Java 虚拟机对整数使用二进制补码表示, 并且二进制补码值的范围不是对称的, 最大负 int 的相反数为相同的最大负数。尽管已出现溢出, 并不抛出异常。

对所有的 int 值  $x, -x$  等于  $(\sim x) + 1$ 。

---

## instanceof

---

**操作** 确定对象是否为给定类型。

<b>格式</b>	<table border="1"> <tr><td>instanceof</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	instanceof	indexbyte1	indexbyte2
instanceof				
indexbyte1				
indexbyte2				
<b>形式</b>	instanceof = 193(0xc1)			
<b>栈</b>	<p>..., objectref <math>\Rightarrow</math></p> <p>..., result</p>			
<b>描述</b>	<p>objectref 必须是 reference 类型, 它从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引的值为 <math>(indexbyte1 \ll 8   indexbyte2)</math>。常数池索引处的项必须是 CONSTANT_Class(§ 4.1), 对类、数组或接口的符号引用。符号引用被解析(§ 5.1)。</p> <p>如果 objectref 不为 null, 并且是解析的类、数组的实例或是接口, 则 instanceof 指令将 int 类型的 result 作为整数压入操作数栈。否则, 它压入 int 类型的 result 0。</p> <p>以下规则用于确定不为 null 的 objectref 是否是解析的类型的实例。如果 S 是 objectref 引用的对象的类, 并且 T 是解析的类、数组或接口类型, instanceof 确定 objectref 是否为 T 的实例如下:</p> <ul style="list-style-type: none"> <li>· 如果 S 是普通(非数组)类, 则: <ul style="list-style-type: none"> <li>· 如果 T 是类类型, 则 S 必须是与 T 相同的类或 T 的子类。</li> <li>· 如果 T 是接口类型, 则 S 必须实现(§ 2.13)接口 T。</li> </ul> </li> <li>· 如果 S 是代表数组类型 SC[] 的类, 即成分为 SC 类型的数组, 则: <ul style="list-style-type: none"> <li>· 如果 T 为类类型, 则 T 必须是 Object(§ 2.4.6)。</li> <li>· 如果 T 是 TC[] 数组类型, 即成分为 TC 类型的数组, 则如下之一必为真: <ul style="list-style-type: none"> <li>· TC 和 SC 为相同的基本类型(§ 2.4.1)。</li> <li>· TC 和 SC 为引用类型(§ 2.4.5), 并且 SC 类型可由这些运行期规则转换为 TC。</li> </ul> </li> </ul> </li> </ul> <p>S 不能是接口类型, 因为不存在接口的实例, 仅有类和数组的实例。</p> <p><b>链接异常</b> 在解析 CONSTANT_Class 常数池项时, § 5.1 中的任何异常都可能被抛出。</p> <p><b>注解</b> instanceof 指令基本上同 checkcast 指令非常相似。不同在于它对 null 的处理上, 当其测试失败时的行为(checkcast 抛出异常, instanceof 压入结果代码), 以及它对操作数栈的影响。</p>			

---

## invokeinterface

---

**操作** 调用接口方法。

**格式**

invokeinterface
indexbyte1
indexbyte2
nargs
0

**形式** invokeinterface=185(0xb9)

**栈** ..., objectref,[arg1,[arg2...]]⇒  
...

**描述** 无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引的值为(indexbyte1<sub>1</sub>8) indexbyte2。常数池索引处的项必须有 CONSTANT\_InterfaceMethodref(§ 4.4.2) 标记,对接口名,方法名和方法的描述符(§ 4.3.3)的引用。常数池项被解析(§ 5.3)。接口方法不能是<init>、实例初始化方法(§ 3.8)或者<clinit>,类或接口初始化方法(§ 3.8)。

nargs 操作数是无符号字节,不能为零。objectref 必须是 reference 类型,并且必须跟随操作数栈上的 nargs - 1 个参数字。参数字的数目和类型,以及它们代表的值的顺序必须同选择的接口方法的描述符一致。

objectref 类型的类的方法表被确定。如果 object 是数组类型,则使用 Object 类的方法表。在方法表中查找名称和描述符同解析的常数池表表项的名称及描述符相同的方法。

查找结果是方法表表项,其中包括对接口方法代码和方法的修饰符信息的引用(见表4-4“方法访问和修饰符标志”)。方法表表项必须是 public 方法。如果 method 为 synchronized,将获得与 objectref 相关联的监视器。

如果方法不是 native,nargs - 1 个参数字和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架,而 objectref 及参数字成为其第一个 nargs 局部变量的值,objectref 在局部变量 0,arg1 在局部变量 1 等等。新框架成为当前的,并且 Java 虚拟机 pc 设置为要调用的方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法是 native 并且它实现的依赖于平台的代码尚未装载并链接到 Java 虚拟机,则这些将被完成。nargs - 1 个参数字及 objectref 从操作数栈弹

出;实现该方法的代码以依赖于平台的方式调用。

#### 链接异常

在解析 CONSTANT\_InterfaceMethodref 常数池项时,§ 5.3 中的任何异常都可能被抛出。

否则,如果在 objectref 类中找不到同解析的名称和描述符匹配的方法,invokeinterface 抛出 IncompatibleClassChangeError。

否则,如果选择的方法是类(static)方法,invokeinterface 指令抛出 IncompatibleClassChangeError。否则,如果选择的方法不是 public,invokeinterface 抛出 IllegalAccessException。

否则,如果选择的方法是 abstract, invokeinterface 抛出 AbstractMethodError。

否则,如果选择的方法是 native,并且实现方法的代码不能被装载或链接,invokeinterface 抛出 UnsatisfiedLinkError。

#### 运行期异常

否则,如果 objectref 为 null, invokeinterface 指令抛出 NullPointerException。

#### 注解

不像 invokevirtual、invokestatic 和 invokespecial,方法调用的参数字(nargs)的数目可作为 invokeinterface 指令的操作数获取。同其他指令一样,还可从选择的方法的描述符得到该值。得到的值一定同 nargs 操作数相同。这种冗余是历史性的,但 nargs 操作数还可以为由 invokeinterface\_quick 伪指令(可以在运行期替换 invokeinterface)使用的操作数在指令中保留空间。见第九章,“优化”中有关 invokeinterface\_quick 的信息。

---

### invokespecial

---

#### 操作

调用实例方法;对超类、私有和实例初始化方法调用的特殊处理。

#### 格式

invokespecial
indexbyte1
indexbyte2

#### 形式

invokespecial=183(0xb7)

#### 栈

..., objectref, [arg1,[arg2...]]⇒  
...

#### 描述

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)常数池的索引。

引,其中索引的值为(indexbyte1 $\ll$ 8) | indexbyte2。常数池索引处的项必须具有 CONSTANT\_Methodref(§ 4.4.2)。标记,对类名,方法名和方法的描述符(§ 4.3.3)的引用。命名的方法被解析(§ 5.2)。被解析方法的描述符必须同解析的类中一个方法的描述符相同。

下一步,Java 虚拟机确定是否下述所有条件都为真:

- 方法名不是<init>,实例初始化方法(§ 3.8)。
- 方法不是 private 方法。
- 方法的类是当前方法的类的超类。
- 对当前类设置了 ACC\_SUPER 标志(见表4-1“类访问和修饰符标志”)。如果这样,则 Java 虚拟机在最接近的超类中选择有相同描述符的方法,有可能选择刚解析的方法。

结果方法不能是<clinit>,类或接口初始化方法(§ 3.8)。

如果方法是<init>,实例初始化方法(§ 3.8),则该方法只能在未初始化的对象上调用一次,并且要在分配该对象的 new 指令执行后的第一次向后转移之前调用。

最后,如果方法为 protected(§ 4.6),则它必须是当前类的成员或当前类的超类的成员,并且 objectref 的类必须是当前类或当前类的子类。

代表解析的方法的常数池表项包括对方法代码的引用,不能为零的无符号字节 nargs 和方法的修饰符信息(见表4-4“方法访问和修饰标志”)。objectref 必须是 reference 类型,并且必须跟随操作数栈上 nargs-1 个参数字,其中参数字的数目和类型,以及它们表示的值的顺序必须同选择的实例方法的描述符一样。

如果方法为 synchronized,则获得与 objectref 相关联的监视器。

如果方法不是 native,nargs-1 个参数字及 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架,而 objectref 和参数字成为它的第一个 nargs 局部变量的值,objectref 在局部变量 0,arg1 在局部变量 1 等等。新的栈框架成为当前的,并且 Java 虚拟机 pc 设置为要调用的方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法是 native 并且它实现的依赖于平台的代码尚未装载并链接到 Java 虚拟机,则这些将被完成。nargs-1 个参数字及 objectref 从操作数栈弹出;实现该方法的代码以依赖于平台的方式调用。

## 链接异常

在解析 CONSTANT\_Methodref 常数池项时,§ 5.2 中的任何异常都可能被抛出。

否则,如果指定的方法存在但为类(static)方法,invokespecial 指令抛出 IncompatibleClassChangeError。

否则,如果指定的方法是 abstract, invokespecial 抛出 AbstractMethodError。

否则,如果指定的方法是 native,并且实现该方法的代码不能被装载或链

接,invokespecial 抛出 UnsatisfiedLinkError。

**运行期异常** 否则,如果 object 为 null,invokespecial 指令抛出 NullPointerException。

**注解** invokespecial 和 invokevirtual 指令的不同在于,invokevirtual 基于对象的类调用方法。invokespecial 指令用于调用实例初始化方法(<init>),private 方法,以及当前类的超类的方法。invokespecial 指令在 Sun 的 JDk 1.0.2 版本之前被命名为 invokenonvirtual。

---

### invokestatic

---

**操作** 调用类(static)方法。

**格式**

invokestatic
indexbyte1
indexbyte2

**形式** invokestatic=184(0xb8)

**栈** ..., [arg1,[arg2...]]⇒  
...

**描述** 无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引的值为(indexbyte1<<8)|indexbyte2。常数池索引处的项必须有 CONSTANT\_Methodref(§ 4.4.2) 标记,对类名,方法名和方法的描述符(§ 4.3.3)的引用。命名的方法被解析(§ 5.2)。被解析方法的描述符必须同解析的类中一个方法的描述符相同。方法不能是<init>,实例初始化方法(§ 3.8),或<clinit>,类或接口初始化方法(§ 3.8)。它必须是 static,不能是 abstract。最后,如果方法是 protected(§ 4.6),则它必须是当前类的成员或当前类的超类的成员。

代表被解析方法的常数池表项包括对方法代码的直接引用,可能为零的无符号字节 nargs,以及方法的修饰符信息(见表4.4“方法访问和修饰符标志”)。操作数栈必须包括 nargs 个参数字,其中参数字的数目,类型以及它们所表示的值的顺序必须同被解析方法的描述符一致。

如果方法是 synchronized,则获得同当前方法关联的监视器。

如果方法不是 native,nargs 个参数字从操作数栈弹出。为正调用的方法创建新的栈框架,参数字成为它的第一个 args 局部变量的值,arg1 在局部变量 0,arg2 在局部变量 1 等等。新的栈框架成为当前的,并且 Java 虚拟机 pc 设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执

行。

如果方法是 native, nargs 个参数字从操作数栈弹出; 以依赖于平台的方法调用实现该方法的代码。

## 链接异常

在解析 CONSTANT\_Methodref 常数池项时, § 5.2 中的任何异常都可能被抛出。

否则, 如果指定的方法存在, 但不是实例方法, invokestatic 指令抛出 IncompatibleClassChangeError。否则, 如果指定的方法是 native, 并且实现该方法的代码不能被装载或链接, invokestatic 抛出 UnsatisfiedLinkError。

---

## invokevirtual

---

### 操作

调用实例方法; 基于类分派。

### 格式

invokevirtual
indexbyte1
indexbyte2

### 形式

invokevirtual = 182(0xb6)

### 栈

..., objectref,[arg1,[arg2...]]⇒  
...

### 描述

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。常数池索引处的项必须具有 CONSTANT\_Methodref(§ 4.4.2)标记, 对类名, 方法名和方法的描述符(§ 4.3.3)的引用。命名的方法被解析(§ 5.2)。被解析方法的描述符必须同被解析类中一个方法的描述符相同。该方法不能是实例初始化方法(§ 3.8)(init), 或类及接口初始化方法(§ 3.8)(clinit)。最后, 如果方法是 protected(§ 4.6), 则它必须是当前类的成员或当前类的超类的成员, objectref 的类必须是当前类或当前类的子类。代表被解析方法的常数池表项包括对被解析类的方法表的无符号索引, 以及不能为零的无符号字节 nargs。

objectref 必须是 reference 类型。index 作为对具有 objectref 类型的类的方法表的索引使用。如果 objectref 为数组类型, 则使用 Object 类的方法表。index 处的表项包括对方法代码的直接引用和其修饰符信息(见表4-4“方法访问和修饰符标志”)。

objectref 必须跟随操作数栈上的 nargs - 1 个参数字, 其中参数字的数目,

类型以及它们所表示的值的顺序必须同选择的实例方法的描述符一致。如果方法是 `synchronized`, 则获得同 `objectref` 关联的监视器。如果方法不是 `native`, `nargs` - 1 个参数字及 `objectref` 从操作数栈弹出。为正调用的方法创建新的栈框架。`objectref` 和参数字成为它的第一个 `nargs` 局部变量的值, `objectref` 在局部变量 0, `arg1` 在局部变量 1 等等。新的栈框架成为当前的, 并且 Java 虚拟机 `pc` 设置为要调用的方法的第一条指令的操作码。从该方法的第一条指令继续执行。如果方法是 `native` 并且实现它的依赖于平台的代码尚未装载并链接到 Java 虚拟机, 这些将被完成。`nargs` - 1 个参数字和 `objectref` 从操作数栈弹出; 以依赖于平台的方式调用实现该方法的代码。

<b>链接异常</b>	在解析 <code>CONSTANT_Methodref</code> 常数池项时, § 5.2 中的任何异常都可能被抛出。 否则, 如果指定的方法存在, 但为类( <code>static</code> )方法, <code>invokevirtual</code> 指令抛出 <code>IncompatibleClassChangeError</code> 。 否则, 如果指定的方法为 <code>abstract</code> , <code>invokevirtual</code> 抛出 <code>AbstractMethodError</code> 。 否则, 如果指定的方法为 <code>native</code> , 并且实现方法的代码不能被装载或链接, <code>invokevirtual</code> 抛出 <code>UnsatisfiedLinkError</code> 。
<b>运行期异常</b>	否则, 如果 <code>objectref</code> 为 <code>null</code> , <code>invokevirtual</code> 指令抛出 <code>NullPointerException</code> 。

---

## ior

---

<b>操作</b>	int 布尔“或”。
<b>格式</b>	<input type="text" value="ior"/>
<b>形式</b>	<code>ior=128(0x80)</code>
<b>栈</b>	<code>..., value1, value2 =&gt;</code> <code>..., result</code>
<b>描述</b>	<code>value1</code> 和 <code>value2</code> 必须都是 int 类型。它们从操作数栈弹出。 <code>int</code> 类型的 <code>result</code> 通过对 <code>value1</code> 和 <code>value2</code> 按位“或”来计算。 <code>result</code> 被压入操作数栈。

---

## irem

---

操作	求 int 余数。
格式	irem
形式	irem = 112(0x70)
栈	..., value1, value2 $\Rightarrow$ ..., result
描述	value1 和 value2 必须都是 int 类型。这些值从操作数栈弹出。int 类型的 result 为 $value1 - (value1 / value2) * value2$ 。result 被压入操作数栈。 irem 指令的结果 $(a/b) * b + (a \% b)$ 等于 a。这种相等性即使在特殊情况下，即被除数是该类型的最大可能数量的负 int 而除数为 -1 (余数为 0)，也可以保持。服从该规则，求余数操作的结果仅当被除数为负时可为负，并且仅当被除数为正时可为正。此外，result 的数量总小于除量的数量。
运行期异常	如果 int 求余数操作的除数值为 0，irem 抛出 ArithmeticException。

---

## ireturn

---

操作	从方法返回 int。
格式	ireturn
形式	ireturn = 172(0xac)
栈	..., value $\Rightarrow$ [空]
描述	返回方法必须有 byte、short、char 或 int 返回类型。value 必须是 int 类型。 value 从当前框架 (§ 3.6) 的操作数栈弹出并压入调用者框架的操作数栈。 当前方法的操作数栈上的其他任何值被抛弃。如果返回方法是 synchronized 方法，则调用方法时获得的或重入的监视器被释放或退出 (分别地)， 好像执行了 monitorexit 指令。 然后解释器将控制返回给方法的调用者，恢复调用者的框架。

---

## ishl

---

操作	int 左移。
格式	ishl
形式	ishl=120(0x78)
栈	..., value1, value2⇒ ..., result
描述	value1和 value2必须都是 int 类型。这些值从操作数栈弹出。int 类型的 result 通过将 value1左移 s 位来计算,其中 s 是 value2的低5位的值。result 被压入操作数栈。
注解	这等同于(即使出现溢出)乘以2的 s 次幂。实际使用的移动距离总在0到31范围内(闭区间),好像 value2与屏蔽值0x1f 进行按位逻辑“与”。

---

## ishr

---

操作	int 算术右移。
格式	ishr
形式	ishr=122(0x7a)
栈	..., value1, value2⇒ ..., result
描述	value1和 value2必须都是 int 类型。这些值从操作数栈弹出。int 类型的 result 通过将 value1右移 s 位来计算,有符号扩展,其中 s 是 value2的低5位的值。result 被压入操作数栈。
注解	结果值为 $\lfloor (value1)/2^s \rfloor$ ,其中 s 为 value2&.0x1f。对于非负 value1,这等同于截断被2的 s 次幂除的 int。实际使用的移动距离总在闭区间0到31范围内,好像 value2与屏蔽值0x1f 进行按位逻辑“与”。

---

**istore**

---

**操作** 将 int 存储到局部变量。

**格式**

istore
index

**形式**  $istore = 54(0x36)$

**栈**  $\dots, value \Rightarrow$

$\dots$

**描述** index 是无符号字节, 必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 value 必须是 int 类型。它从操作数栈弹出, 并且 index 处的局部变量的值被设置为 value。

**注解** istore 操作码可以同 wide 指令一同使用两字节无符号索引访问局部变量。

---

**istore\_{n}**

---

**操作** 将 int 存储到局部变量。

**格式**

istore\_{n}

**形式**  $istore\_0 = 59(0x3b)$   
 $istore\_1 = 60(0x3c)$   
 $istore\_2 = 61(0x3d)$   
 $istore\_3 = 62(0x3e)$

**栈**  $\dots, value \Rightarrow$

$\dots$

**描述** {n} 必须是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 value 必须是 int 类型。它从操作数栈弹出, {n} 处局部变量的值被设置为 value。

**注解** 每条 istore\_{n} 指令同带 index{n} 的 istore 相同, 只不过操作数{n} 是隐式的。

---

## **isub**

---

<b>操作</b>	int 相减。	
<b>格式</b>	<table border="1"><tr><td>isub</td></tr></table>	isub
isub		
<b>形式</b>	isub=100(0x64)	
<b>栈</b>	...,value1,value2⇒ ...,result	
<b>描述</b>	<p>value1和 value2必须都是 int 类型。这些值从操作数栈弹出。int 类型的 value 为 value1—value2。result 被压入操作数栈。</p> <p>对 int 减法, <math>a - b</math> 产生与 <math>a + (-b)</math> 同样的结果。对 int 值, 从零减去与相反数相同。</p> <p>尽管可能出现溢出或下溢, 在此情况下 result 可能具有与数学结果不同的符号, 但 isub 指令的执行从不抛出运行期异常。</p>	

---

## **iushr**

---

<b>操作</b>	int 逻辑右移。	
<b>格式</b>	<table border="1"><tr><td>iushr</td></tr></table>	iushr
iushr		
<b>形式</b>	iushr=124(0x7c)	
<b>栈</b>	...,value1,value2⇒ ...,result	
<b>描述</b>	value1和 value2必须都是 int 类型。这些值从操作数栈弹出。int 类型的 result 通过将 value 右移 s 位来计算。result 被压入操作数栈。	
<b>注解</b>	如果 value1 为正而 s 为 value2&0x1f, 则结果与 $value1 \gg s$ 相同; 如果 value1 为负, 则结果与表达式 $(value1 \gg s) + (2 \ll \sim s)$ 的值相同。附加的 $(2 \ll \sim s)$ 项抵消了传播的符号位。实际使用的移动距离总在闭区间 0 到 31 范围内。	

---

## ixor

---

**操作** int 布尔“异或”。

**格式**

ixor
------

**形式** ixor=130(0x82)

**栈** ... , value1 , value2 =>  
... , result

**描述** value1 和 value2 必须都是 int 类型。它们从操作数栈弹出。int 类型的 result 通过对 value1 和 value2 按位“异或”来计算。result 被压入操作数栈。

---

## jsr

---

**操作** 跳转到子例程。

**格式**

jsr
branchbyte1
branchbyte2

**形式** jsr=168(0xa8)

**栈** ... =>  
... , address

**描述** 紧跟着此 jsr 指令的指令操作码的 address 被作为 returnAddress 类型的值压入操作数栈。无符号的 branchbyte1 和 branchbyte2 用来构造有符号的 16 位偏移量，其偏移量为 (branchbyte1 << 8) | branchbyte2。从此 jsr 指令地址起的偏移量处开始执行。目标地址必须是一条指令的操作码的地址，该指令在包含此 jsr 指令的方法内。

**注解** 在实现 Java 语言的 finally 子句时，jsr 指令同 ret 指令一同使用（见 7.13 节“编译 finally”）。注意到 jsr 将地址压入栈，而 ret 将其从局部变量中取出。这种非对称性是有意的。

---

**jsr\_w**

---

**操作** 跳转到子例程(宽索引)。

**格式**

jsr_w
branchbyte1
branchbyte2
branchbyte3
branchbyte4

**形式** jsr\_w=201(0xc9)

**栈**

…⇒  
…, address

**描述** 紧跟着此 jsr\_w 指令的指令操作码的 address 被作为 returnAddress 类型的值压入操作数栈。无符号的 branchbyte1、branchbyte2、branchbyte3 和 branchbyte4 用于构造有符号的 32 位偏移量，其偏移量为 (branchbyte1<24) | (branchbyte2<16) | (branchbyte3<8) | branchbyte4。从此 jsr\_w 指令地址起的偏移量处开始执行。目标地址是一条指令的操作码的地址，该指令在包含此 jsr\_w 指令的方法内。

**注解** 在实现 Java 语言的 finally 子句时，jsr\_w 指令同 ret 指令一同使用（见 7.13 节“编译 finally”）。注意到 jsr\_w 将地址压入栈，而 ret 将其从局部变量中取出。这种非对称性是有意的。  
虽然 jsr\_w 指令有 4 字节的转移偏移量，但其他因素将 Java 方法的长度限制为 65 535 字节（§ 4.10）。这种限制可能在 Java 虚拟机的将来版本中被解除。

---

**l2d**

---

**操作** 将 long 转换为 double。

**格式**

l2d

**形式** l2d=138(0x8a)

**栈** …, value.word1, value.word2⇒  
…, result.word1, result.word2

**描述** 操作数栈顶的 value 必须是 long 类型。它从操作数栈弹出并使用 IEEE 754 最接近舍入模式转换为 double 类型的 result。result 被压入操作数栈。

**注解** l2d 指令执行放宽的基本转换(§ 2.6.2)，可能丢失精度，因为 double 类型只有53个尾数位。

---

## l2f

---

**操作** 将 long 转换为 float。

**格式**

l2f
-----

**形式** l2f=137(0x89)

**栈** ..., value.word1, value.word2 =>  
..., result

**描述** 操作数栈顶的 value 必须是 long 类型。它从操作数栈弹出并使用 IEEE 754 最接近舍入模式转换为 float 类型的 result。result 被压入操作数栈。

**注解** l2f 指令执行放宽的基本转换(§ 2.6.2)，可能丢失精度，因为 float 类型只有24个尾数位。

---

## l2i

---

**操作** 将 long 转换为 int。

**格式**

l2i
-----

**形式** l2i=136(0x88)

**栈** ..., value.word1, value.word2 =>  
..., result

**描述** 操作数栈顶的 value 必须是 long 类型。它从操作数栈弹出并通过取 long 值的低序32位并抛弃高序32位来转换为 int。result 被压入操作数栈。

**注解** l2i 指令执行缩窄的基本转换(§ 2.6.3)。它可能丢失 value 的总的数量信

息。`result` 还可能和 `value` 有不同的符号。

---

## **ladd**

---

<b>操作</b>	long 相加。	
<b>格式</b>	<table border="1"><tr><td>ladd</td></tr></table>	ladd
ladd		
<b>形式</b>	<code>ladd=97(0x61)</code>	
<b>栈</b>	$\dots, \text{value1. word1}, \text{value1. word2}, \text{value2. word1}, \text{value2. word2} \Rightarrow \dots, \text{result. word1}, \text{result. word2}$	
<b>描述</b>	<code>value1</code> 和 <code>value2</code> 必须都是 long 类型。这些值从操作数栈弹出。long 类型的 <code>result</code> 为 <code>value1 + value2</code> 。 <code>result</code> 被压入操作数栈。如果 long 加法溢出，则结果为由 long 表示的数学和的低序位。如果出现溢出，则结果的符号将同两值的数学和的符号不同。	

---

## **ladd**

---

<b>操作</b>	从数组装载 long。	
<b>格式</b>	<table border="1"><tr><td>laload</td></tr></table>	laload
laload		
<b>形式</b>	<code>laload=47(0x2f)</code>	
<b>栈</b>	$\dots, \text{arrayref}, \text{index} \Rightarrow \dots, \text{value. word1}, \text{value. word2}$	
<b>描述</b>	<code>arrayref</code> 必须是 reference 类型并且必须引用一个数组，该数组成分为 long 类型。 <code>index</code> 必须是 int 类型。 <code>arrayref</code> 和 <code>index</code> 都从操作数栈弹出。数组中 <code>index</code> 处成分的 long 值被获取并压入操作数栈顶。	
<b>运行期异常</b>	如果 <code>arrayref</code> 为 null， <code>ladd</code> 抛出 <code>NullPointerException</code> 。否则，如果 <code>index</code> 不在 <code>arrayref</code> 所引用的数组的边界内，则 <code>ladd</code> 指令抛出 <code>ArrayIndexOutOfBoundsException</code> 。	

---

## land

---

**操作** long 布尔“与”。

**格式**

land
------

**形式** land=127(0x7f)

**栈**  $\dots, value1.\text{word1}, value1.\text{word2}, value2.\text{word1}, value2.\text{word2} \Rightarrow \dots, result.\text{word1}, result.\text{word2}$

**描述** value1 和 value2 必须都是 long 类型。它们从操作数栈弹出。long 类型的 result 通过对 value1 和 value2 进行按位“与”来计算。result 被压入操作数栈。

---

## lastorelastore

---

**操作** 存储到 long 数组。

**格式**

lastore
---------

**形式** lastore=80(0x50)

**栈**  $\dots, arrayref, index, value.\text{word1}, value.\text{word2} \Rightarrow \dots$

**描述** arrayref 必须是 reference 类型并且必须引用一个数组，该数组成分是 long 类型。index 必须是 int 类型，而 value 必须是 long 类型。arrayref, index 和 value 从操作数栈弹出。long 值 value 作为数组中 index 处的成分存储。

**运行期异常** 如果 arrayref 为 null，lastore 抛出 NullPointerException。否则，如果 index 不在 arrayref 所引用的数组的边界内，则 lastore 指令抛出 ArrayIndexOutOfBoundsException。

---

## lcmp

---

**操作** 比较 long。

**格式**

lcmp
------

<b>形式</b>	<code>lcmp = 148(0x94)</code>
<b>栈</b>	$\dots, \text{value1}. \text{word1}, \text{value1}. \text{word2}, \text{value2}. \text{word1}, \text{value2}. \text{word1} \Rightarrow \dots, \text{result}$
<b>描述</b>	<code>value1</code> 和 <code>value2</code> 必须都是 long 类型。它们都从操作数栈弹出，并执行有符号整数的比较。如果 <code>value1</code> 大于 <code>value2</code> ，则将 int 值 1 压入操作数栈。如果 <code>value1</code> 等于 <code>value2</code> ，则将 int 值 0 压入操作数栈。如果 <code>value1</code> 小于 <code>value2</code> ，则将 int 值 -1 压入操作数栈。

### **lconst\_{l}**

<b>操作</b>	压入 long 常数。	
<b>格式</b>	<table border="1"><tr><td><code>lconst_{l}</code></td></tr></table>	<code>lconst_{l}</code>
<code>lconst_{l}</code>		
<b>形式</b>	<code>lconst_0 = 9(0x9)</code> <code>lconst_1 = 10(0xa)</code>	
<b>栈</b>	$\dots \Rightarrow \dots, \langle l \rangle. \text{word1}, \langle l \rangle. \text{word2}$	
<b>描述</b>	将 long 常数 $\langle l \rangle$ (0 或 1) 压入操作数栈。	

### **ldc**

<b>操作</b>	压入常数池中的项		
<b>格式</b>	<table border="1"><tr><td><code>ldc</code></td></tr><tr><td><code>index</code></td></tr></table>	<code>ldc</code>	<code>index</code>
<code>ldc</code>			
<code>index</code>			
<b>形式</b>	<code>ldc = 18(0x12)</code>		
<b>栈</b>	$\dots \Rightarrow \dots, \text{item}$		
<b>描述</b>	<code>index</code> 是无符号字节，必须是对当前类（§ 3.6）的常数池的有效索引。 <code>index</code> 处的常数池表项必须是 CONSTANT_Integer（§ 4.4.4）、CONSTANT_		

Float(§ 4.4.4)或 CONSTANT\_String(§ 4.4.3)。常数池表项被解析(§ 5.4, § 5.5)。如果表项是 CONSTANT\_Integer 或 CONSTANT\_Float, 它必须包含数值 item, item 分别作为 int 或 float 压入操作数栈。如果 index 处的表项为 CONSTANT\_String, 它必须包含 CONSTANT\_Utf8(§ 4.4.7)字符串。类 String 的一个实例被创建并初始化为 CONSTANT\_Utf8 字符串.item, 即对实例的 reference, 被压入操作数栈。

**链接异常** 在解析 CONSTANT\_String 常数池项时, § 5.4 中的任何异常都可能被抛出。

---

### ldc\_w

---

**操作** 压入常数池中的项(宽索引)。

**格式**

ldc_w
indexbyte1
indexbyte2

**形式** ldc\_w=19(0x13)

**栈**

...⇒  
...,item

**描述** 无符号的 indexbyte1 和 indexbyte2 组成对当前类(§ 3.6)的常数池的无符号的 16 位索引, 其索引值被计算为(indexbyte1<<8)|indexbyte2。该索引必须是对当前类的常数池的有效索引。index 处的常数池表项必须是 CONSTANT\_Integer(§ 4.4.4)、CONSTANT\_Float(§ 4.4.4) 或 CONSTANT\_String(§ 4.4.3)。常数池表项被解析(§ 5.4, § 5.5)。如果表项是 CONSTANT\_Integer 或 CONSTANT\_Float, 它必须包含数值 item, item 分别作为 int 和 float 压入操作数栈。

如果常数池索引处的表项为 CONSTANT\_String, 它必须包含 CONSTANT\_Utf8(§ 4.4.7)字符串。类 String 的实例被创建并初始化为 CONSTANT\_Utf8 字符串.item, 即对实例的 reference, 被压入操作数栈。

**链接异常** 在解析 CONSTANT\_String 常数池项时, § 5.4 中的任何异常都可能被抛出。

**注解** 除 ldc\_w 有更宽的常数池索引外, ldc\_w 指令与 ldc 指令相同。

---

## ldc2\_w

---

**操作** 压入常数池中的 long 或 double(宽索引)。

**格式**

ldc2_w
indexbyte1
indexbyte2

**形式** ldc2\_w=20(0x14)

**栈**

...⇒

..., item.word1, item.word2

**描述** 无符号的 indexbyte1 和 indexbyte2 组成对当前类(§ 3.6)的常数池的无符号的 16 位索引, 其索引值被计算为(indexbyte1<=81 indexbyte2)。该索引必须是对当前类的常数池的有效索引。索引处的常数池表项必须是 CONSTANT\_Long(§ 4.4.5) 或 CONSTANT\_Double(§ 4.4.5)。常数池表项被解析(§ 5.5)。该表项必须包含数值 item, item 分别作为 long 或 double 压入操作数栈。

**注解** 只存在宽索引版本的 ldc2\_w 指令; 不存在用一个字节的索引压入 long 或 double 的 ldc2 指令。

---

## ldiv

---

**操作** long 相除。

**格式**

ldiv
------

**形式** ldiv=109(0x6d)

**栈**

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒  
..., result.word1, result.word2

**描述** value1 和 value2 必须都是 long 类型。这些值从操作数栈弹出。long 类型的 result 是 Java 表达式 value1/value2 的值。result 被压入操作数栈。  
long 除法向零舍入; 即, n/d 中 long 值产生的商是 long 值 q, 其数量在满足 |d·q| ≤ |n| 时尽可能大。此外, 当 |n| ≥ |d| 并且 n 和 d 同号时 q 是正的, 但

当  $|n| \geq |d|$  并且 n 和 d 反号时 q 是负的。

有一种特殊情况不满足此规则：如果被除数是 long 类型的最大可能数量的负整数，而除数为 -1，则出现溢出并且结果等于被除数；尽管有溢出，在此情况下，无异常被抛出。

**运行期异常** 如果 long 除法中的除数的值为 0，ldiv 抛出 ArithmeticException。

---

## lload

---

**操作** 从局部变量装载 long。

**格式**

lload
index

**形式** lload = 22(0x16)

**栈** ...  $\Rightarrow$   
..., value.word1, value.word2

**描述** index 是无符号字节。index 和 index + 1 必须都是对当前框架（§ 3.6）的局部变量的有限索引。index 和 index + 1 处的局部变量合在一起必须包含 long。index 和 index + 1 处局部变量的值被压入操作数栈。

**注解** lload 操作码可以同 wide 指令一同使用，来用两字节无符号索引访问局部变量。

---

## lload\_{n}

---

**操作** 从局部变量装载 long。

**格式** lload\_{n}

**形式** lload\_0 = 30(0x1e)  
lload\_1 = 31(0x1f)  
lload\_2 = 32(0x20)  
lload\_3 = 33(0x21)

**栈** ...  $\Rightarrow$

..., value. word1, value. word2

**描述**  $\langle n \rangle$  和  $\langle n \rangle + 1$  必须都是对当前框架(§ 3.6)局部变量的有效索引。 $\langle n \rangle$  和  $\langle n \rangle + 1$  处的局部变量合在一起必须包含 long。 $\langle n \rangle$  和  $\langle n \rangle + 1$  处局部变量的值被压入操作数栈。

**注解** 每个 lload\_ $\langle n \rangle$  指令同带 index $\langle n \rangle$  的 lload 相同,只不过操作数 $\langle n \rangle$ 是隐式的。

---

### lmul

---

**操作** long 相乘。

**格式**

lmul
------

**形式** lmul = 105(0x69)

**栈** ..., value1. word1, value1. word2, value2. word1, value2. word2  $\Rightarrow$   
..., result. word1, result. word2

**描述** value1 和 value2 必须都是 long 类型。这些值从操作数栈弹出。long 类型的 result 为 value1 \* value2, result 被压入操作数栈。如果 long 乘法溢出,则结果是由 long 表示的数学乘积的低序位。如果出现溢出,则结果的符号可能与两值的数学乘积的符号不同。

---

### lneg

---

**操作** 对 long 求反。

**格式**

lneg
------

**形式** lneg = 117(0x75)

**栈** ..., value. word1, value. word2  $\Rightarrow$   
..., result. word1, result. word2

**描述** value 必须是 long 类型。它从操作数栈弹出。long 类型的 result 是 value 的算术相反数, -value。result 被压入操作数栈。  
对 long 值,相反数与从零减去相同。因为 Java 虚拟机对整数使用二进制补

码表示,而且二进制补码值的区间是不对称的,最大负 long 的相反数为相同的最大负数。尽管已出现溢出,无异常被抛出。

对所有的 long 值  $x$ , $-x$  等于  $(\sim x) + 1$ 。

---

## lookupswitch

---

**操作** 通过关键字匹配访问跳转表并跳转。

**格式**

lookupswitch
⟨0–3 byte pad⟩
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
npairs1
npairs2
npairs3
npairs4
match-offset pairs...

**形式**  $\text{lookupswitch} = 171(0xab)$

**栈** ..., key $\Rightarrow$

...

**描述**

lookupswitch 是变长指令。紧跟在 lookupswitch 操作码后面,插入0到3个 null 字节(为零的字节,不是 null 对象)作为填充。选择 null 字节的数目,以使 defaultbyte1 开始于距当前方法开始处(它的第一条指令的操作码)为4字节倍数的地址。紧跟在填充体后面,是一系列有符号的32位值:有符号32位值的 default, npairs 及其后的 npairs 对。npairs 必须大于等于0。每个 npairs 由 int 类型的 match 和有符号的32位 offset 组成。这些有符号的32位值的每一个都是从4个无符号字节  $(\text{byte1} \ll 24) | (\text{byte2} \ll 16) | (\text{byte3} \ll 8) | \text{byte4}$  构造出来的。

lookupswitch 指令的表的 match-offset 对必须按 match 的数值增长的顺序排列。

key 必须是 int 类型,并从操作数栈弹出。key 同 match 值比较。如果同其中一个相等,则目标地址通过将相应的 offset 加到该 lookupswitch 指令操作码的地址上来计算。如果 key 不同 match 值的任何一个匹配,目标地址通

过将 default 加到该 lookupswitch 指令操作码的地址上来计算。然后从目标地址继续执行。

可从每个 match-offset 对的偏移量计算出的目标地址，与从 default 计算出的目标地址，都必须是一条指令的操作码的地址，该指令在包含此 lookupswitch 指令的方法内。

<b>注解</b>	lookupswitch 指令 4 字节操作数对齐的要求保证那些操作数当且仅当包含 lookupswitch 指令的方法定位在 4 字节边界时 4 字节对齐。match-offset 对被排序以支持比线性查找更快的查询例程。
-----------	--

---

## lor

---

<b>操作</b>	long 布尔“或”。	
<b>格式</b>	<table border="1"><tr><td>lor</td></tr></table>	lor
lor		
<b>形式</b>	lor = 129(0x81)	
<b>栈</b>	..., value1.word1, value1.word2, value2.word1, value2.word2 => ..., result.word1, result.word2	
<b>描述</b>	value1 和 value2 必须都是 long 类型。它们从操作数栈弹出。long 类型的 result 通过对 value1 和 value2 按位逻辑“或”来计算。result 被压入操作数栈。	

---

## lrem

---

<b>操作</b>	求 long 余数。	
<b>格式</b>	<table border="1"><tr><td>lrem</td></tr></table>	lrem
lrem		
<b>形式</b>	lrem = 113(0x71)	
<b>栈</b>	..., value1.word1, value1.word2, value2.word1, value2.word2 => ..., result.word1, result.word2	
<b>描述</b>	value1 和 value2 必须都是 long 类型。这些值从操作数栈弹出。long 类型的 result 为 $value1 - (value1 / value2) * value2$ 。result 被压入操作数栈。 lrem 指令的结果 $(a/b) * b + (a \% b)$ 等于 a。这种相等性即使在特殊情况下，即被除数是该类型的最大可能数量的负 long，而除数为 -1 (余数为 0)，也	

将保持。服从此规则,余数操作的结果仅当被除数为负时可为负,并且仅当被除数为正时可为正;此外,结果的数量总小于除数的数量。

**运行期异常** 如果 long 余数操作符的除数的值为0,lrem 抛出 ArithmeticException。

---

### lreturn

---

**操作** 从方法返回 long。

**格式** lreturn

**形式** lreturn=173(0xad)

**栈** ..., value. word1, value. word2 =>  
[空]

**描述** 返回方法必须有 long 返回类型。value 必须为 long 类型。value 从当前框架(§ 3.6)的操作数栈弹出,并压入调用者框架的操作数栈。当前方法的操作数栈上其他任何值都被抛弃。如果返回方法为 synchronized 方法,则调用方法时获得或重入的监视器被释放或退出(分别地),好像执行了 monitorexit 指令。

然后解释器将控制返回给方法的调用者,恢复调用者的框架。

---

### lshl

---

**操作** long 左移。

**格式** lshl

**形式** lshl=121(0x79)

**栈** ..., value1. word1, value1. word2, value2 =>  
..., result. word1, result. word2

**描述** value1 必须是 long 类型,而 value2 必须是 int 类型。这些值从操作数栈弹出。long 类型的 result 通过将 value1 左移 s 位来计算,其中 s 是 value2 的低 6 位。result 被压入操作数栈。

**注解** 这等同于(即使出现溢出)乘以2的 s 次幂。实际使用的移动距离总在闭区间

0到63范围内,好像 value2与屏蔽值0x3f按位进行逻辑“与”。

---

## lshr

---

<b>操作</b>	long 算术右移。	
<b>格式</b>	<table border="1"><tr><td>lshr</td></tr></table>	lshr
lshr		
<b>形式</b>	lshr=123(0x7b)	
<b>栈</b>	...,value1.word1,value1.word2,value2⇒ ..., result.word1, result.word2	
<b>描述</b>	value1必须是 long 类型,而 value2必须是 int 类型。这些值从操作数栈弹出。long 类型的 result 通过将 value1右移 s 位来计算,有符号扩展,其中 s 是 value2的低6位的值。result 被压入操作数栈。	
<b>注解</b>	结果值为 $[(value1)/2^s]$ ,其中 s 为 value2&0x3f。对非数 value1,这等价于截断被2的 s 次幂除的 long。实际使用的移动距离总在闭区间0到63范围内,好像 value2与屏蔽值0x3f 进行按位逻辑“与”。	

---

## lstore

---

<b>操作</b>	将 long 存储到局部变量。		
<b>格式</b>	<table border="1"><tr><td>lstore</td></tr><tr><td>index</td></tr></table>	lstore	index
lstore			
index			
<b>格式</b>	lstore=55(0x37)		
<b>栈</b>	...,value.word1,value.word2⇒ ...		
<b>描述</b>	index 是无符号字节。index 和 index+1 必须都是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 value 必须是 long 类型。它从操作数栈弹出,同时 index 和 index+1 处的局部变量被设置为 value。		
<b>注解</b>	lstore 操作码可由 wide 指令一同使用,来用两字节无符号索引访问局部变		

量。

---

### **lstore\_<n>**

---

**操作** 将 long 存储到局部变量。

**格式**

lstore_<n>
------------

**形式** lstore\_0=63(0x3f)  
lstore\_1=64(0x40)  
lstore\_2=65(0x41)  
lstore\_3=66(0x42)

**栈** ..., value. word1, value. word2 =>  
...

**描述** <n> 和 <n>+1 必须都是对当前框架(§ 3.6)的局部变量的有效索引。操作数栈顶的 value 必须是 long 类型。它从操作数栈弹出，同时 <n> 和 <n>+1 处的局部变量被设置为 value。

**注解** 每个 lstore\_<n> 指令与带 index<n> 的 lstore 相同，只不过操作数 <n> 是隐式的。

---

### **lsub**

---

**操作** long 相减。

**格式**

lsub
------

**形式** lsub=101(0x65)

**栈** ..., value1. word1, value1. word2, value2. word1, value2. word2 =>  
..., result. word1, result. word2

**描述** value1 和 value2 必须都是 long 类型。这些值从操作数栈弹出。long 类型的 result 为 value1 - value2, result 被压入操作数栈。对 long 减法, a - b 产生与 a + (-b) 同样的结果。对 long 值, 从零减去和相反数相同。尽管可能出现溢出或下溢, 在此情况下 result 可能与真正的数学结果有不同的符号, 但 lsub 指令的执行从不抛出运行期异常。

---

## lushr

---

操作	long 逻辑右移。	
格式	<table border="1"><tr><td>lushr</td></tr></table>	lushr
lushr		
形式	lushr=125(0x7d)	
栈	...,value1.word1,value1.word2,value2⇒ ...,result.word1,result.word2	
描述	value1必须是 long 类型,而 value2必须是 int 类型。这些值从操作数栈弹出。long 类型的 result 通过将 value1逻辑右移由 value2的低6位所指示的数量来计算。result 被压入操作数栈。	
注解	如果 value1为正而 s 是 value2&0x3f,则结果与 value1 $\gg$ s 相同;如果 value1 为负,结果等于表达式(value1 $\gg$ s)+(2L $\ll$ ~s)的值。附加的(2L $\ll$ ~s)项抵消了传播的符号位。实际使用的移动距离总在闭区间0到63范围内。	

---

## lxor

---

操作	long 布尔“异或”。	
格式	<table border="1"><tr><td>lxor</td></tr></table>	lxor
lxor		
形式	lxor=131(0x83)	
栈	...,value1.word1,value1.word2,value2.word1,value2.word2⇒ ...,result.word1,result.word2	
描述	value1和 value2必须都是 long 类型。它们从操作数栈弹出。long 类型的 result 通过对 value1和 value2进行按位“异或”来计算。result 被压入操作数栈。	

---

## monitorenter

---

操作	进入对象的监视器。
----	-----------

<b>格式</b>	<b>monitorenter</b>
<b>形式</b>	<code>monitorenter = 194(0xc2)</code>
<b>栈</b>	<code>..., objectref =&gt;</code> <code>...</code>
<b>描述</b>	<p><code>objectref</code> 必须是 <code>reference</code> 类型。</p> <p>每个对象都有一个与其关联的监视器。执行 <code>monitorenter</code> 的线程获得同 <code>objectref</code> 相关联的监视器的所有权。如果另一个线程已经拥有同 <code>objectref</code> 相关联的监视器，则当前线程将等待直到对象被解锁，然后再次尝试获得所有权。如果当前线程已拥有同 <code>objectref</code> 相关联的监视器，它增加监视器中的计数器来指示该线程已进入监视器的次数。如果同 <code>objectref</code> 相关联的监视器不被任何线程拥有，则当前线程成为该监视器的所有者，设置该监视器的计数表项为1。</p>
<b>运行期异常</b>	如果 <code>objectref</code> 为 <code>null</code> , <code>monitorenter</code> 抛出 <code>NullPointerException</code> 。
<b>注解</b>	<p>有关 Java 虚拟机中线程和监视器的详细信息，见第8章，“线程和锁”。<code>monitorenter</code> 指令可同 <code>monitorexit</code> 指令一同使用来实现 Java 的 <code>synchronized</code> 块。<code>monitorenter</code> 指令不在 <code>synchronized</code> 方法的实现中使用，虽然它提供等价的语义；调用 <code>synchronized</code> 方法时的监视器进入由 Java 虚拟机的方法调用指令隐式地处理。见 § 7.14“同步”，以获取关于使用 <code>monitorenter</code> 和 <code>monitorexit</code> 指令的更多信息。</p> <p>监视器同对象的关联可以用不同的方法管理，这超出了此规范的范围。例如，监视器可以像对象一样同时分配和解除分配。或者，它可在当线程试图获得对对象的互斥访问时动态分配，并在以后对象的监视器中无线程时被释放。Java 语言的同步化构造，除进入和退出外，还需要对监视操作的其他支持，包括等待监视器(<code>Object.wait</code>)并通知其他等待监视器的线程(<code>Object.notify</code> and <code>Object.notifyAll</code>)。Java 虚拟机提供的标准包 <code>java.lang</code> 支持这些操作。Java 虚拟机指令集中不显式地出现对这些操作的支持。</p>

---

<b>monitorexit</b>	
<b>操作</b>	退出对象的监视器。
<b>格式</b>	<b>monitorexit</b>
<b>形式</b>	<code>monitorexit = 195(0xc3)</code>

<b>栈</b>	<code>...,objectref⇒ ...</code>
<b>描述</b>	objectref 必须是 reference 类型。 当前线程必须是同 objectref 所引用的实例相关联的监视器的所有者。该线程增加计数器,以指示它已进入该监视器的次数。如果计数器的值变为零,当前线程释放监视器。如果同 objectref 关联的监视器成为自由的,其他等待获得该监视器的线程可以试图获取它。
<b>运行期异常</b>	如果 objectref 为 null,monitorexit 抛出 NullPointerException。否则,如果当前线程不是监视器的所有者,monitorexit 抛出 IllegalMonitorStateException。
<b>注解</b>	有关 Java 虚拟机中线程和监视器的详细信息,见第8章,“线程和锁”。 monitorenter 和 monitorexit 指令可以用于实现 Java 的 synchronized 块。 monitorexit 指令不用于 synchronized 方法的实现,虽然它提供等价的语言,监视器退出正常或非正常 synchronized 方法完成是由 Java 虚拟机的方法调用指令隐式处理的。当错误被抛出时,Java 虚拟机还隐式地从 synchronized 块内处理监视器的退出。有关使用 monitorenter 和 monitorexit 指令的更多信息,见 § 7.14“同步”。

---

## multianewarray

---

<b>操作</b>	创建新的多维数组。				
<b>格式</b>	<table border="1"> <tr><td>multianewarray</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> <tr><td>dimensions</td></tr> </table>	multianewarray	indexbyte1	indexbyte2	dimensions
multianewarray					
indexbyte1					
indexbyte2					
dimensions					
<b>形式</b>	<code>multianewarray=197(0xc5)</code>				
<b>栈</b>	<code>...,count1,[count2,...]⇒ ...,arrayref</code>				
<b>描述</b>	dimensions 是无符号字节,必须大于等于1。它表示要创建的数组的维数的值。操作数栈必须包含 dimensions 字,它必须是 int 类型并非负,每个代表要创建数组的一个维数中成分的数目。count1是在第一维中想要的长度,				

count2是第二维的长度等等。

所有的 count 值从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。常数池索引处的项必须是 CONSTANT\_Class(§ 4.4.1)。符号引用被解析(§ 5.1.3)。结果表项必须是维数大于等于 dimensions 的数组类类型。

新的数组类型的多维数组从垃圾回收堆中分配。第一维中数组的成分被初始化为第二维类型的子数组。数组第一维的成分初始化为该成分类型(§ 2.5.1)的缺省值。对新数组的 reference 类型的 arrayref 被压入操作数栈。

**链接异常** 在解析 CONSTANT\_Class 常数池项时,§ 5.1 中的任何异常都可能被抛出。

否则,如果不允许当前类访问被解析数组类的基类,multianewarray 抛出 IllegalAccessError。

**运行期异常** 否则,如果操作数栈上的任何 dimensions 值小于零,multianewarray 指令抛出 NegativeArraySizeException。

**注解** 当创建一维数组时,使用 newarray 或 anewarray 可能更有效率。

常数池指令引用的数组类可能比 multianewarray 指令的 dimensions 操作数有更多维数。在此情况下,仅创建数组维数的第一个 dimensions。

---

## new

---

**操作** 创建新对象。

**格式**

new
indexbyte1
indexbyte2

**形式** new=187(0xbb)

**栈**

..., $\Rightarrow$   
...,objectref

**描述** 无符号的 indexbyte1 和 indexbyte 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。常数池索引处的项必须是

CONSTANT\_Class(§ 4.4.1)。符号引用被解析(§ 5.1)并导致类类型(它不能导致数组或接口类型)。该类的新实例的存储器从垃圾回收堆中分配,新对象的实例变量被初始化为它们的缺省初始值(§ 2.5.1)。objectref,即对实例的reference,被压入操作数栈。

#### 链接异常

在解析 CONSTANT\_Class 常数池项时,§ 5.1 中的任何异常都可能被抛出。

否则,如果 CONSTANT\_Class 常数池项解析为接口或 abstract 类,new 抛出 InstantiationException。

否则,如果不允许当前类访问被解析的类(§ 2.7.8),new 抛出 IllegalAccessError。

#### 注解

new 指令不彻底创建新实例;新实例直到在未初始化实例上调用了实例初始化方法时,才彻底创建。

---

#### newarray

---

#### 操作

创建新数组。

#### 格式

newarray
atype

#### 形式

newarray=188(0xbc)

#### 栈

...,count⇒  
...,arrayref

#### 描述

count 必须是 int 类型。它从操作数栈弹出。count 代表要创建的数组中的元素数目。

atype 是指示要创建数组类型的代码。它必须取下述值之一:

数组类型	atype
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9

T_INT	10
T_LONG	11

成分为 *atype* 类型, 长度为 *count* 的新数组由垃圾回收堆分配。对该新数组对象的 reference 类型的 arrayref 被压入操作数栈。新数组的所有元素被初始化为该类型(§ 2.5.1)的缺省初始值。

**运行期异常** 如果 *count* 小于零, newarray 抛出 NegativeArraySizeException。

**注解** 在 sun 的 Java 虚拟机实现中, boolean 类型的数组(*atype* 为 T\_BOOLEAN)被作为8位值的数组存储, 并使用 baload 和 bastore 指令操纵, 这些指令也访问 byte 类型的数组。其他实现可以实现压缩的 boolean 数组; baload 和 bastore 指令必须仍用于访问那些数组。

### nop

**操作** 不作任何事。

**格式**

nop

**形式** nop=0(0x0)

**栈** 无变化。

**描述** 不作任何事。

### pop

**操作** 弹出操作数栈顶的字。

**格式**

pop

**形式** pop=87(0x57)

**栈** ..., word⇒  
...

**描述** 从操作数栈弹出其顶部的字。  
pop 指令不能被使用,除非 word 是包含32位数据类型的字。

**注解** 除保持64位数据类型完整性的限制外, pop 指令对无类型字操作, 忽略它所包含的数据类型。

---

### pop2

---

**操作** 弹出操作数栈顶的两个字。

**格式**

pop2
------

**形式** pop2=88(0x58)

**栈** ..., word2, word1⇒  
...

**描述** 从操作数栈弹出其顶部的两个字。pop2指令不能被使用, 除非字 word1和 word2的每一个都是包含32位数据类型的字, 或者合在一起是一个64位数据的两个字。

**注解** 除保持64位数据类型完整性的限制外, pop2指令对原始的字操作, 忽略它所包含的数据类型。

---

### putfield

---

**操作** 设置对象中的域。

**格式**

putfield
indexbyte1
indexbyte2

**形式** putfield=181(0xb5)

**栈** ..., objectref, value⇒  
...  
或  
**栈** ..., objectref, value. word1, value. word2⇒  
...

**描述** 无符号的 indexbyte1和 indexbyte2用于构造对当前类(§ 3.6)的常数池的

索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。索引处的常数池项必须是CONSTANT\_Fieldref(§ 4.4.2),即对类名和域名的引用。如果域为protected(§ 4.6),则它必须是当前类的成员或当前类的超类的成员,而objectref的类必须是当前类或当前类的子类。

常数池项被解析(§ 5.2),确定域宽度和域偏移量。由putfield指令存储的value的类型必须同正存储到的类实例的域的描述符(§ 4.3.2)相容。如果域描述符为byte char, short或int类型,则value必须是int。如果域描述符为float、long或double类型,则value必须分别是float、long或double。如果域描述符为引用类型,则value必须是同域描述符赋值相容的类型。

value和必须是reference类型的objectref,从操作数栈弹出。从objectref引用的对象开始的偏移量处的域被设置为value。

**链接异常** 在解析CONSTANT\_Fieldref常数池项时,§ 5.2中的任何异常都可能被抛出。

否则,如果指定的域存在,但为static域,putfield抛出IncompatibleClassChangeError。

**运行期异常** 否则,如果objectref为null,putfield指令抛出NullPointerException。

**注解** putfield指令对一个和两个字节宽的域都可进行操作。

---

### putstatic

---

**操作** 设置类中的static域。

**格式**

putstatic
indexbyte1
indexbyte2

**形式** putstatic=179(0xb3)

**栈** ..., value $\Rightarrow$

...

或

**栈** ..., value.word1, value.word2 $\Rightarrow$

...

**描述** 无符号的indexbyte1和indexbyte2用于构造对当前类(§ 3.6)的常数池的

索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。索引处的常数池项必须是CONSTANT\_Fieldref(§ 4.4.2),即对类名和域名的引用。如果域为protected(§ 4.6),则它必须是当前类的成员或当前类的子类的成员。

常数池项被解析(§ 5.2),确定域和其宽度。由putstatic指令存储的value的类型必须同正存储到的类实例的域描述符(§ 4.3.2)相容。如果域描述符为byte、char、short或int类型,则value必须是int。如果域描述符为float、long或double类型,则value必须分别是float、long或double。如果域描述符类型为引用类型,则value必须是同域描述符赋值相容的类型。value从操作数栈弹出,同时类域被设置为value。

#### 链接异常

在解析CONSTANT\_Fieldref常数池项时,§ 5.2中的任何异常都可能被抛出。

否则,如果指定的域存在,但不是static域(类变量),putstatic抛出IncompatibleClassChangeError。

#### 注解

putstatic指令对一个和两个字宽的域都可进行操作。

---

#### ret

---

#### 操作

从子例程返回。

#### 格式

ret
index

#### 形式

ret=169(0xa9)

#### 栈

无变化。

#### 描述

index是闭区间0到255间的无符号字节。当前框架(§ 3.6)index处的局部变量必须包含returnAddress类型。局部变量的内容被写入Java虚拟机的pc寄存器,并从那里继续执行。

#### 注解

在Java语言的finally关键的实现中(见7.13节“编译finally”),ret指令同jsr或jsr\_w指令一同使用。注意到jsr将地址压入栈,而ret从局部变量中将其取出。这种非对称性是有意的。

ret指令不应同return指令混淆。return指令将控制从一个Java方法返回到其调用者,而不向调用者传回任何值。

ret指令可用wide指令一同使用两字节无符号索引访问局部变量。

---

**return**

---

**操作** 从方法返回 void。

**格式**

return
--------

**形式** return=177(0xb1)

**栈** ...⇒  
[空]

**描述** 返回方法必须有 void 返回类型。当前框架(§ 3.6)操作数栈上的任何值都被抛弃。如果返回方法是 synchronized 方法，则调用方法时获得或重入的监视器被释放或退出(分别地)，好像执行了 monitorexit 指令。  
解释器将控制返回给方法的调用者，恢复调用者的框架。

---

**saload**

---

**操作** 从数组装载 short。

**格式**

saload
--------

**形式** saload=53(0x35)

**栈** ..., arrayref, index⇒  
..., value

**描述** arrayref 必须是 reference 类型并且必须引用一个数组，该数组成分为 short 类型。index 必须是 int 类型。arrayref 和 index 都从操作数栈弹出。数组中 index 处成分的 short 类型的 value 被获取，有符号扩展为 int 类型的 value，并压入操作数栈顶。

**运行期异常** 如果 arrayref 为 null，saload 抛出 NullPointerException。否则，如果 index 不在 arrayref 引用的数组的边界内，saload 指令抛出 ArrayIndexOutOfBoundsException。

---

## sastore

---

**操作** 存储到 short 数组。

**格式**

sastore
---------

**形式** sastore=86(0x56)

**栈** ... , array , index , value ⇒  
...

**描述** arrayref 必须是 reference 类型，并且必须引用一个数组，该数组成分是 short 类型。index 和 value 必须都是 int 类型。arrayref、index 和 value 从操作数栈弹出。int 类型的 value 被截断为 short，并作为数组中 index 处的成分存储。

**运行期异常** 如果 arrayref 为 null，sastore 抛出 NullPointerException。否则，如果 index 不在 arrayref 引用的数组的边界内，sastore 指令抛出 ArrayIndexOutOfBoundsException。

---

## sipush

---

**操作** 压入 short。

**格式**

sipush
byte1
byte2

**形式** sipush=17(0x11)

**栈** ... ⇒  
..., value

**描述** 立即的无符号的 byte1 和 byte2 值组成立即 short，其值为  $(\text{byte1} \ll 8 | \text{byte2})$ 。该立即值被有符号扩展为 int，而结果 value 被压入操作数栈。

---

## suap

---

**操作** 交换操作数栈顶的两个字。

<b>格式</b>	<code>swap</code>
<b>形式</b>	<code>swap = 95(0x5f)</code>
<b>栈</b>	$\dots, \text{word2}, \text{word1} \Rightarrow$ $\dots, \text{word1}, \text{word2}$
<b>描述</b>	<p>操作数栈顶的两个字被交换。</p> <p><code>swap</code> 指令不能被使用,除非 <code>word2</code> 和 <code>word1</code> 的每一个都是包含32位数据类型的字。</p>
<b>注解</b>	除保持64位数据类型完整性的限制外, <code>swap</code> 指令对无类型字操作,忽略它们所包含的数据的类型。

## **tableswitch**

<b>操作</b>	通过索引访问跳转表并跳转。															
<b>格式</b>	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px;">tableswitch</td></tr> <tr><td style="padding: 2px;">⟨0—3 byte pad⟩</td></tr> <tr><td style="padding: 2px;">defaultbyte1</td></tr> <tr><td style="padding: 2px;">defaultbyte2</td></tr> <tr><td style="padding: 2px;">defaultbyte3</td></tr> <tr><td style="padding: 2px;">defaultbyte4</td></tr> <tr><td style="padding: 2px;">lowbyte1</td></tr> <tr><td style="padding: 2px;">lowbyte2</td></tr> <tr><td style="padding: 2px;">lowbyte3</td></tr> <tr><td style="padding: 2px;">lowbyte4</td></tr> <tr><td style="padding: 2px;">highbyte1</td></tr> <tr><td style="padding: 2px;">highbyte2</td></tr> <tr><td style="padding: 2px;">highbyte3</td></tr> <tr><td style="padding: 2px;">highbyte4</td></tr> <tr><td style="padding: 2px;">jump offsets...</td></tr> </table>	tableswitch	⟨0—3 byte pad⟩	defaultbyte1	defaultbyte2	defaultbyte3	defaultbyte4	lowbyte1	lowbyte2	lowbyte3	lowbyte4	highbyte1	highbyte2	highbyte3	highbyte4	jump offsets...
tableswitch																
⟨0—3 byte pad⟩																
defaultbyte1																
defaultbyte2																
defaultbyte3																
defaultbyte4																
lowbyte1																
lowbyte2																
lowbyte3																
lowbyte4																
highbyte1																
highbyte2																
highbyte3																
highbyte4																
jump offsets...																
<b>形式</b>	<code>tableswitch = 170(0xaa)</code>															
<b>栈</b>	$\dots, \text{index} \Rightarrow$ $\dots$															
<b>描述</b>	<code>tableswitch</code> 是变长指令。紧接在 <code>tableswitch</code> 操作码后面,插到0到3个 null 字节															

(为零的字节,不是 null 对象)作为填充。选择 null 字节的数目,以使随后的字节开始于距当前方法开始处(其第一条指令的操作码)为4字节的倍数的地址。紧跟在填空体后面的是系列有符号的32位值:default、low、high 和 high-low+1 个有符号32位偏移量。low 的值必须小于等于 high。high-low+1 个有符号32位偏移量被作为基于0的跳转表对待。这些有符号的32位值的每一个由 (byte1<24 | (byte2<16) | (byte3<8) | byte4 构成。

index 必须是 int 类型并从操作数栈弹出。如果 index 小于 low 或 index 大于 high,则目标地址通过将 default 加到该 tableswitch 指令的操作码的地址来计算。否则,提取跳转表 index-low 处的偏移量。目标地址通过将该偏移量加到 tableswitch 指令的操作码的地址来计算。从目标地址处继续执行。

可从每个跳转表偏移量或从 default 计算出的目标地址,必须是一条指令的操作码的地址,该指令在包含此 tableswitch 指令的方法内。

**注解** tableswitch 指令的4字节操作数对齐的要求,保证那些操作数当且仅当包含 tableswitch 的方法开始于4字节边界时4字节对齐。

---

### wide

---

**操作** 通过附加字节扩展局部变量索引。

**格式1**

wide
⟨opcode⟩
indexbyte1
indexbyte2

其中 ⟨opcode⟩ 为 iload、fload、aload、lload、dload、istore、fstore、astore、lstore、dstore 或 ret 中的一个。

**格式2**

wide
iinc
indexbyte1
indexbyte2
constbyte1
constbyte2

**形式** wide=196(0xc4)

**栈** 与更改的指令相同。

**描述** wide 指令更改另一条指令的行为。它有两种格式,依赖于被更改的指令。wide 指令的第一种形式更改 iload、fload、aload、lload、dload、istore、fstore、astore、lstore、dstore 或 ret 中的一个。第二种形式只适用于 iinc 指令。在两

种情况下，在编译的代码中，wide 操作码本身后面跟随着 wide 所更改的指令的操作码。两种形式下，两个无符号字节 indexbyte1 和 indexbyte2 跟随更改的操作码并被组成对当前框架（§ 3.6）中局部变量的 16 位无符号索引，其索引值为  $(indexbyte1 \ll 8) | indexbyte2$ 。计算出的索引必须是对当前框架局部变量的有效索引。wide 指令更改 lload、dload、lstore 或 dstore 指令，跟随在计算出的索引后的索引（索引 + 1）必须也是对局部变量的有效索引。在第二种形式下，两个立即的无符号字节 constbyte1 和 constbyte2 在代码流中跟随在 indexbyte1 和 indexbyte2 后面。这些字节也可以装配成有符号的 16 位常数，该常数为  $(constbyte1 \ll 8) | constbyte2$  放宽的字节码像正常时一样操作，只不过使用更宽的索引，并且在第二种形式中有更大的增加范围。

#### 注解

虽然我们说 wide “更改另一条指令的行为”，wide 指令有效地将更改的指令作为 wide 操作数对待，使进程中嵌入的指令非自然化。在更改 iinc 指令的情况下，iinc 的一个逻辑操作数甚至不在距操作码的正常偏移量处。嵌入的指令不能直接执行；其操作码不能是任何控制转移指令的目标。

## 第七章 为 Java 虚拟机编译

Java 虚拟机的设计支持 Java 程序设计语言。Sun 的 Java 程序设计语言的 JDK 1.0.2 发布版包含从 Java 源代码到 Java 虚拟机指令集的编译器(java c)，以及实现 Java 虚拟机本身的运行期系统(java)。理解一个 Java 编译器如何使用 Java 虚拟机，对未来的 Java 编译器编写员，以及试图理解 Java 虚拟机操作的人是有用的。

虽然本章集中在编译 Java 代码，Java 虚拟机并不假定它执行的指令从 Java 源代码产生。虽然已有一些努力旨在将其他语言编译到 Java 虚拟机，Java 虚拟机的 1.0.2 版的设计不支持广泛的语言。一些语言可以相当直接地由 Java 虚拟机作主机。其他可能支持仅能非高效实现的构造。

我们正在考虑 Java 虚拟机未来版本的限制扩展。来更直接地支持广泛的语言。如果你对这种努力感兴趣，请同我们联系 [jvm@javasoft](mailto:jvm@javasoft)。

注意到术语“编译器”有时在涉及从 Java 虚拟机指令集到特定 CPU 的指令集的翻译器时使用。这种翻译器的一个例子是“刚好及时”(Just In Time, JIT) 代码生成器，它仅在 Java 虚拟机代码已被装载到 Java 虚拟机后产生特定平台的指令。本章不提及同代码生成相关的问题，而仅涉及那些同从 Java 源代码编译到 Java 虚拟机指令有关的问题。

### 7.1 范例格式

本章主要由 Java 源代码和 Sun 的 JDK 1.0.2 发布版中 java c 编译器为范例产生的 Java 虚拟机代码的注释列表所组成。Java 虚拟机以非正式的“虚拟机汇编语言”书写，它是由 Sun 的 javap 实用程序输出的，同时随 JDK 发送。你可以使用 javap 生成编译的 Java 方法的附加范例。

范例格式将为任何已阅读过汇编代码的人所熟悉。每条指令采取形式：

```
<index><opcode>[<operand1>[<operand2>...]][<comment>]
```

<index>是指令操作码的索引，该指令在包含此方法的 Java 虚拟机代码字节的数组中。或者，<index>可理解为从方法开始处算起的字节偏移量。<opcode>是指令操作码的助记符，零或更多<operandN>是指令的操作数。可选的<comment>在 Java 风格的行结尾注释语法中给出：

```
8 bipush 100          // Push constant 100
```

注释中的一些材料由 javap 产生；其余由作者提供。每条指令前的<index>可用作控制转移指令的目标。例如，“goto 8”指令将控制转移到索引 8 处的指令。注意到 Java 虚拟机控制转移指令的实际操作数是从这些指令操作码的地址算起的偏移量；这些操作数由 javap 显示，并被展

示在本章中,作为它们的方法的更易读取的偏移量。

我们以散列符号开始一个代表常数池索引的操作数,并以识别引用的常数池项的注释跟在指令后面,如同在:

```
10 ldc #1           // Float 100.00000  
或  
9 invokevirtual #4 // Method Example.addTwo(II) I
```

出于本章的目的,我们不担心诸如指定操作数尺寸这样的细节。

## 7.2 常数、局部变量和控制构造的使用

Java 虚拟机代码展示由 Java 虚拟机的设计所强加的一般特征集以及类型的使用。在第一个范例中,我们遇到许多这种情况,我们在某些细节上考虑它们。

spin 方法简单地围绕空的 for 循环 100 次:

```
void spin() {  
    int i;  
    for (i=0;i<100;i++) {  
        ;      //Loop body is empty  
    }  
}
```

Java 编译器将 spin 编译为:

```
方法 void spin()  
0 iconst_0          // Push int constant0  
1 istore_1          // Store into local 1 (i=0)  
2 goto 8            //First time through don't increment  
5 iinc 1 1          // Increment local 1 by 1(i++)  
8 iload_1           //Push local 1(i)  
9 bipush 100         // push int constant(100)  
11 if_icmplt 5       //Compare,loop if<(i<100)  
14 return           //Return void when done
```

Java 虚拟机是面向栈的,大多数操作从 Java 虚拟机的当前框架的操作数栈中取一个或多个操作数,或者将结果压回到操作数栈。新框架在每次调用 Java 方法时创建,同时创建该方法使用的新的操作数栈和局部变量集(见 3.6 节“框架”)。在计算的任一点,可能因而每个控制线程有许多框架以及同样多的操作数栈,对应于许多嵌套的方法调用。仅当前框架中的操作数栈是活跃的。

Java 虚拟机指令集通过对在其各种数据类型上的操作使用不同字节码,来区分操作数类型。方法 spin 仅在 int 类型的值上操作。其编译代码中的指令被选择以在有类型的数据(icontst\_0,istore\_1,iinc,iload\_1,if\_icmplt)上操作,这些指令都专用于 int 类型。

spin 中的两个常数,0 和 100,被用两个不同的指令压入操作数栈。0 使用 iconst\_0 指令压入,即 iconst\_{i} 指令族中的一个。100 使用 bipush 指令压入,它将其压入的值作为立即操作数取得。

Java 虚拟机经常通过使操作数(在 iconst\_{i} 指令情况下是 int 常数 -1、0、1、2、3、4 和 5)在操作码中为隐式,来利用某种操作数的相似性。因为 iconst\_0 指令知道它要压入 int\_0,iconst\_0 不需要存储操作数来告诉它要压入什么值,也不需要获取或译码操作数。将压入 0 编译为 bipush\_0 会是正确的,但会使 spin 的编译代码增加一个字节长度。简单的虚拟机也会在每次循环时花费附加时间来获取和译码显式的操作数。使用隐式操作数使编译代码要紧凑也更有效率。

spin 中的 int 作为 Java 虚拟机局部变量 1 存储。因为大多数的 Java 虚拟机指令对从操作数栈弹出的值,而不是对局部变量进行操作,在局部变量和操作数栈间转移值的指令在为 Java 虚拟机所编译的代码中是常见的。这些操作还具有指令集中特殊的 support。在 spin 中,使用 istore\_1 和 iload\_1 指令在值和局部变量间转移,每条指令隐式地对局部变量 1 操作。istore\_1 指令从操作数栈弹出 int,并将其存储到局部变量 1 中。iload\_1 指令将局部变量 1 中的值压入操作数栈。

局部变量的使用(和重用)是编译器编写者的责任。特定的装载和存储指令应当鼓励编译器编写者尽可能地重用局部变量。结果代码更快,更紧凑,并且使用 Java 框架中更少的空间。

某些对局部变量的频繁操作特别为 Java 虚拟机所迎合。iinc 指令将局部变量的内容增加一个字节的有符号值。spin 中的 iinc 指令将第一个局部变量(它的第一个操作数)增加 1(它的第二个操作数)。在实现循环构造时,iinc 指令非常方便。

spin 的 for 循环主要由这些指令伴随:

```
5 iinc 11          //Increment local 1 by 1 (i++)
8 iload_1         //Push local 1(i)
9 bipush100        //Push int constant(100)
11 if_icmplt 5    //Compare,loop if<(i<100)
```

bipush 指令将值 100 作为 int 压入操作数栈,然后 if\_icmplt 指令将该值从栈中弹出并将其同 i 比较。如果比较成功(Java 变量 i 小于 100),控制被转移到索引 5 并且开始 for 循环的下一次重复。否则,控制传递到跟在 if\_icmplt 后的指令。

如果 spin 范例对计数器使用了不是 int 的数据类型,编译代码将必需反映不同的数据类型。例如,如果在 spin 范例中不是使用 int 而是用 double:

```
void dspin() {
    double i;
    for (i=0.0;i<100.0;i++) {
```

```

        ;      // Loop body is empty
    }
}

```

编译代码为：

```

方法 void dspin()
0 dconst_0           //push double constant 0.0
1 dstore_1           //Store into locals 1 and 2 (i=0.0)
2 goto 9             //First time through don't increment
5 dload_1            //Push double onto operand stack
6 dconst_1            //Push double constant 1 onto stack
7 dadd                //Add; there is no dinc instruction
8 dstore_1           //Store result in locals 1 and 2
9 dload_1            //Push local
10 ldc2_w #4          //Double 100.000000
13 dcmplg             //There is no if_dcmplt instruction
14 iflt 5              //Compare, loop if < (i < 100.000000)
17 return             //Return void when done

```

操作有类型数据的指令现在专用于 double 类型。(ldc2\_w 指令将在本章后面讨论。)

注意到 dspin, double 值使用两个字的存储器,无论在操作数栈上或是局部变量中。这对 long 类型的值也适用。作为另一个范例：

```

double doubleLocals(double d1,double d2) {
    return d1+d2;
}

```

成为

```

方法 double doubleLocals(double,double)
0 dload_1           //First argument in locals 1 and 2
1 dload_3           //Second argument in locals 3 and 4
2 dadd                //Each also uses two words on stack
3 dreturn

```

总是必须成对地并且以其原来的顺序访问两个字类型的字。例如, doublelocals 中 double 值的字不能单个操纵。

Java 虚拟机的一个字节的操作码尺寸导致其编译代码非常紧凑。然而,一个字节的操作

码还意味着 Java 虚拟机指令集必须保留短小。析中的作法是 Java 虚拟机不对所有数据类型提供同等的支持:并不完全正交(见表3-1“Java 虚拟机指令集中的类型支持”)。对于 dspin,注意到 Java 虚拟机指令集中设有 if\_icmplt 指令。相反,比较必须用后跟 iflt 的 dcmplt 来执行,它需要比 spin 的 int 版本多用一个 Java 虚拟机指令。

Java 虚拟机提供对 int 类型数据的最直接的支持。这部分地因为 Java 虚拟机的操作数栈和局部变量是一个字宽,而一个字被保证持有所有直到整数类型的值,并包括 int 值。它也是由在通常的 Java 程序中频繁使用 int 数据所引起的。

更小的整数类型具有较少的直接支持。没有存储,装载或相应指令的 byte、char 或 short 版本,例如,这里是使用 short 编写的 spin 范例:

```
void sspin() {  
    short i;  
    for(i=0;i<100;i++) {  
        ;      // Loop body is empty  
    }  
}
```

必须使用操作另一种类型的指令,最可能是 int,在 short 和 int 值之间转换,如果需要,来确保对 short 数据操作的结果保持在适当范围内。

```
方法 void sspin ()  
0  iconst_0  
1  istore_1  
2  goto 10  
5  iload_1           //The short is stored in an int  
6  iconst_1  
7  iadd  
8  i2s              //Truncate int to short  
9  istore_1  
10 iload_1  
11 bipush 100  
13 if_icmplt 5  
16 return
```

Java 虚拟机中缺少对 byte、char 和 short 类型的直接支持并不特别可惜,因为这些类型的值被内部提升为 int(byte 和 short 是有符号扩展为 int,char 是零扩展)。这样对 byte、char 和 short 数据的操作可以使用 int 指令完成。仅有的附加代价是将 int 操作的值截断为有效的范围。

long 和浮点类型在 Java 虚拟机中拥有中间级别的支持,只是缺少条件控制转移指令的完

整补充。

### 7.3 运算

Java 虚拟机一般在其操作数栈上进行运算(例外是 iinc 指令,它直接增加局部变量的值)。例如,align2grain 方法将 int 值对齐为 2 grain 大小的给定次幂。

```
int align2grain(int i,int grain) {  
    return ((i+grain-1)&~(grain-1));  
}
```

运算操作的操作数从操作数栈弹出,并且操作结果被压回操作数栈。运算的子计算的结果可作为它们嵌套计算的操作数获得。例如,计算 $\sim(grain-1)$ 由这些指令处理:

```
5 iload_2          //Load grain onto operand stack  
6 iconst_1         //Load constant 1 onto operand stack  
7 isub             //Subtract;push result onto stack  
8 iconst_m1        //Load constant -1 onto operand stack  
9 ixor             //Do XOR;push result onto stack
```

首先使用局部变量 2 和立即 int 值 1 的内容计算 grain-1。这些操作数从操作数栈弹出,它们的不同值被压回到操作数栈,在那里可作为 ixor 指令的一个操作数立即获取以使用。相似地,ixor 指令的结果成为随后 iand 指令的操作数。

整个方法的代码如下:

```
方法 int align2grain(int ,int )  
0 iload_1  
1 iload_2  
2 iadd  
3 iconst_1  
4 isub  
5 iload_2  
6 iconst_1  
7 isub  
8 iconst_m1  
9 ixor  
10 iand  
11 ireturn
```

## 7.4 访问常数池

许多数值常数、对象、域和方法经当前类的常数池访问。对象访问以后再考虑(§ 7.8)。`int`、`long`、`float` 和 `double` 类型的 Java 数据,以及对 `String` 实例的引用(常数池项标记为 `CONSTANT_String`),使用 `ldc`、`ldc_w` 和 `ldc2_w` 指令管理。

`ldc` 和 `ldc_w` 指令用于访问常数池中一个字节的值(包括 `String` 类的实例),而 `ldc2_w` 用于访问两个字节的值。仅当需要大量的常数池项和较大的索引来访问一个项时代替 `ldc` 使用 `ldc_w`。`ldc2_w` 指令用于访问所有两个字的项,没有非宽变体。

`byte`、`char` 或 `short` 类型的整数常数以及小 `int` 值,可以用 `bipush`、`sipush` 或 `iconst_{i}` 指令编译,如同前面见到的(§ 7.2)。某种小数点常数可使用 `fconst_{f}` 和 `dconst_{d}` 指令编译。

在所有这些情况下编译是简明的。例如,对常数:

```
void useManyNumeric() {  
    int i=100;  
    int j=1000000;  
    long l1=1;  
    long l2=0xffffffff;  
    double d=2.2;  
    ... do some calculations...  
}
```

设置如下:

```
方法 void useManyNumeric()  
0 bipush 100          //Push a small int with bipush  
2 istore_1  
3 ldc #1              //Integer 1000000;a larger int  
                      // value uses ldc  
5 istore_2  
6 lconst_1            //A tiny long value uses short, fast lconst_1  
7 lstore_3  
8 ldc2_w #6           //A long 0xffffffff(that is, an int_1);any  
                      //long constant value can be pushed by ldc2_w  
11 lstore 5  
13 ldc2_w #8           //Double 2.200000;so do  
                      //uncommon double values  
16 dstore 7  
进行那些计算
```

## 7.5 更多控制范例

Java 的 for 语句的编译如同在前面的节(§ 7.2)所示。大多数 Java 的其他内部方法控制转移构造(if-then-else、do、while、break 和 continue)也以显见的方式编译。Java 的 switch 语句的编译在隔开的节(7.10节“编译开关”)中处理,就像异常的编译(7.12节“抛出和处理异常”)以及 Java 的 finally 语句(7.13节“编译 finally”)。

作为深入范例,while 循环以显见的方式编译,显然特定的控制转移指令由 Java 虚拟机获得,因数据类型而不同。像通常一样,对 int 类型的数据有更多的支持:

```
void whileInt() {  
    int i=0;  
    while(i<100) {  
        i++;  
    }  
}
```

被编译为:

```
方法 void whileInt()  
0  iconst_0  
1  istore_1  
2  goto 8  
5  iinc 11  
8  iload_1  
9  bipush 100  
11 if_icmplt5  
14 return
```

注意到 while 语句的检测(使用 if\_icmplt 指令实现)在 loop 的 Java 虚拟机代码的底端(在前面的 spin 范例中也是如此)。loop 底端的检测强制 goto 指令的使用,在循环的第一次重复之前进行检测。如果检测失败,而且从不进入循环体,则此额外指令被浪费。然而,while 循环通常在其循环体要运行时使用,常常进行多次循环。对随后的重复,在循环底端放置检测,每次循环时节省了 Java 虚拟机指令:如果检测在循环顶端,循环体将需要跟踪 goto 指令来返回顶端。

涉及其他数据类型的控制构造以类似方式编译,但必须使用对那些数据类型可用的指令。这导致稍微低效的代码,因为需要更多的 Java 虚拟机指令:

```
void whileDouble() {
    double i=0.0;
    while (i<100.1) {
        i++;
    }
}
```

被编译为：

```
方法 void whileDouble()
0 dconst_0
1 dstore_1
2 goto 9
5 dload_1
6 dconst_1
7 dadd
8 dstore_1
9 dload_1
10 ldc2_w #4          //Double 100.100000
13 dcmpg              //To test we have to use
14 iflt 5             //two instructions...
17 return
```

每个浮点类型具有两个比较指令：对 float 类型为 fcmpl 和 fcmpg，对 double 类型为 dcmpl 和 dcmpg。这些变体的区别仅在于它们对 NaN 的处理。NaN 是无序的，所以如果任一操作数为 NaN，则浮点比较失败。编译器对适当的类型选择比较指令的变体，无论比较在非 NaN 值上失败或遇到 NaN 都产生相同的结果。例如：

```
int lessThan100(double d) {
    if (d<100.0) {
        return 1;
    } else {
        return -1;
    }
}
```

编译为：

```
方法 int lessThan100(double)
```

```

0 dload_1
1 ldc2_w #4           //Double 100.00000
4 dcmpg               //Push 1 if d is NaN or d>100.00000;
                      //push 0 if d==100.00000
5 ifge 10              //Branch on 0 or 1
8 iconst_1
9 ireturn
10 iconst_m1
11 ireturn

```

如果 d 不是 NaN 而且小于100.0,dcmpg 指令将 int-1压入操作数栈,而 ifge 指令不转移。无论 d 大于100.0或为 NaN,dcmpg 指令将 int 1压入操作数栈,而 ifge 转移。如果 d 等于100.0,dcmpg 指令将 int 0压入操作数栈,而 ifge 转移。

dcmpl 指令达到相同的效果,如果比较被反转:

```

int greaterThan100(double d) {
    if(d>100.0) {
        return 1;
    } else {
        return -1;
    }
}

```

成为:

```

方法 int greaterThan100(double)
0 dload_1
1 ldc2_w #4           //Double 100.00000
4 dcmpl               //Push -1 if d is Nan or d<100.00000;
                      //push 0 if d==100.00000
5 ifle 10              //Branch on 0 or -1
8 iconst_1
9 ireturn
10 iconst_m1
11 ireturn

```

再一次,无论比较在非 NaN 值上失败或者因为向它传递了 NaN,dcmpl 指令将 int 值压入操作数栈,导致 ifle 转移。如果 dcmp 指令都不存在,范例方法中的一个将不得不进行更多工作来检测 NaN。

## 7.6 接收参数

如果向 Java 实例方法传递了 n 个参数,它们被接收,按约定,框架的编号1到 n 的局部变量为新的方法调用创建。参数以它们传递的顺序接收。例如:

```
int addTwo(int i,int j) {  
    return i+j;  
}
```

编译为:

```
方法 int addTwo(int,int)  
0 iload_1          //Push value of local 1 (i)  
1 iload_2          //Push value of local 2 (j)  
2 iadd             //Add;leave int result on val stack  
3 ireturn          //Return int result
```

按约定,实例方法被传递一个对其实例的 reference 到局部变量零。在 Java 中实例可由 this 关键字访问。将 this 压入局部变量零的代码必须在实例方法的调用者(见 7.7 节,“调用方法”)中出现。

类(static)方法不具有实例,所以对它们来说局部变量零的这种用法是不必要的。类方法开始时使用索引零处的局部变量。如果 addTwo 方法是类方法,其参数将以类似于其最初版本的方式传递:

```
static int addTwoStatic (int i, int j) {  
    return i+j;  
}
```

编译为:

```
方法 int addTwoStatic (int,int)  
0 iload_0  
1 iload_1  
2 iadd  
3 ireturn
```

唯一的不同在于方法参数从局部变量0而不是1处出现。

## 7.7 调用方法

对 Java 实例方法的通常的方法调用在对象的运行期类型上调用(它们是虚拟的,按 C++ 术语)。这样的调用使用 invokevirtual 指令实现,它将其参数取作对常数池表项的索引,给出对象的类类型的完整限符名称,再调用方法的名称,和方法的描述符(§ 4.3.3)。要调用 addTwo 方法,先前作为实例方法定义,我们可以编写:

```
int add12and13() {  
    return addTwo(12,13);  
}
```

它编译为:

```
方法 int add12and13()  
0 aload_0          //Push this local 0(this) onto stack  
1 bipush 12        //Push int constant 12 onto stack  
3 bipush 13        //Push int constant 13 onto stack  
5 invokevirtual #4 //Method Example.addtwo(II)I  
8 ireturn          //Return int on top of stack; it is  
                   //the int result of addTwo()
```

该调用通过首先将对当前实例的引用 this,压到操作数栈来建立。然后该方法调用的参数 int 值 12 和 13 被压入。当创建了 addTwo 方法的框架时,传递到方法的参数成为新框架的局部变量的初始值。即,对 this 的 reference 和两个参数,被调用者压入操作数栈,将成为被调用方法的局部变量 0、1 和 2 的初始值。

最后,addTwo 被调用。当它返回时,其 int 返回值被压入调用者框架的操作数栈,此调用者为 add12and13 方法。这样其返回值被放在适当的位置以便立即返回经 add12and13 的调用者。

从 add12and13 的返回由 add12and13 的 ireturn 指令处理。ireturn 指令取出由 addTwo 返回的,当前框架的操作数栈上的 int 值,并将其压入调用者框架的操作数栈。然后它将控制返回给调用者,使调用者的框架成为当前的。Java 虚拟机对它的许多数值和 reference 数据类型提供不同的返回指令,和对不带返回值方法的 return 指令。同样的返回指令被用于各种方法调用。

invokevirtual 指令的操作数(在范例中,为常数池索引 #4)不是类实例中方法的偏移量。Java 编译器不知道类实例的内部布局。相反,它产生对实例的方法的符号引用,它被存储在常数池中。那些常数池项在运行期解析,来确定实际的方法位置。这对其他所有访问类实例的 Java 虚拟机指令都成立。

调用 addTwoStatic,addTwo 的类(static)变体,类似于:

```
int add12and13() {
    return addTwoStatic(12,13);
}
```

虽然使用了不同的 Java 虚拟机方法调用指令：

方法 int add12and13()  
0 bipush 12  
2 bipush 13  
4 invokestatic #3 // Method Example.addTwoStatic(II)I  
7 ireturn

编译类(static)方法的调用同编译实例方法的调用非常相似,只不过 this 并不传递给调用者。这样方法参数将从局部变量0开始被接收(见7.6节“接收参数”)。invokestatic 指令总用于调用类方法。

invokespecial 指令必须用于调用实例初始化(<init>)方法(见7.8节“处理类实例”)。它还在当调用超类(super)中的方法以及当调用 private 方法时使用。例如,假定 Near 和 Far 类声明为:

```
class Near {
    int it;
    public int getItNear() {
        return getIt();
    }
    private int getIt() {
        return it;
    }
}

class Far extends Near {
    int getItFar() {
        return super.getItNear();
    }
}
```

Near.getItNear 方法(它调用 private 方法)成为:

方法 int getItNear()

```
0 aload_0
1 invokespecial #5           //Method Near. getIt()I
4 ireturn
```

Far. getItFar 方法(它调用超类方法)成为:

```
方法 int getItFar()
0 aload_0
1 invokespecial #4           //Method Near. getItNear()I
4 ireturn
```

注意到使用 invokespecial 指令调用的方法总是向被调用方法传递 this, 作为其第一个参数。像通常一样, 它被接收到局部变量0中。

## 7.8 处理类实例

Java 虚拟机类实例使用 Java 虚拟机的 new 指令创建。一旦类实例已创建, 并且其实例变量, 包括该类的和其超类的, 已被初始化为缺省值, 则新类实例的实例初始化方法(<init>)被调用。(记得在 Java 虚拟机层次, 作为带有特殊的编译器提供的名为<init>的方法出现。这种特殊方法被认为是实例初始化方法(§ 3.8)。多实例初始化方法, 对应于多构造函数, 可以对给定的类存在。)例如:

```
Object create() {
    return new Object();
}
```

编译为:

```
方法 java.lang.Object create()
0 new #1                  //Class java.lang.Object
3 dup
4 invokespecial #4         //Method java.lang.Object.<init>()V
7 areturn
```

类实例被传递和返回(作为 reference 类型), 非常像数值, 虽然 reference 类型有它自己的指令补充:

```
int i;                      // An instance variable
MyObj example() {
```

```

    MyObj o=new MyObj();
    return silly(o);
}

MyObj silly(MyObj o) {
    if (o !=null) {
        return o;
    } else {
        return o;
    }
}

```

成为：

```

方法 MyObj example()
0 new #2                      //Class MyObj
3 dup
4 invokespecial #5           //Method Myobj.<init>() V
7 astore_1
8 aload_0
9 aload_1
10 invokevirtual #4
    //Method Example.silly(LMyObj;)LMyObj;
13 areturn

```

```

方法 MyObj silly(MyObj)
0 aload_1
1 ifnull 6
4 aload_1
5 areturn
6 aload_1
7 areturn

```

类实例的域(实例变量)使用 getfield 和 putfield 指令来访问。如果 i 是 int 类型的实例变量, setIt 和 getIt 方法定义为：

```

void setIt(int value) {
    i=value;
}
int getIt() {

```

```
    return i;  
}
```

成为：

```
方法 void setIt(int)  
0 aload_0  
1 iload_1  
2 putfield #4 //Field Example.i I  
5 return  
  
Method int getIt()  
0 aload_0  
1 getfield #4 //Field Example.i I  
4 ireturn
```

如同对方法调用指令的操作数，putfield 和 getfield 指令的操作数（常数池索引#4）不是类实例中域的偏移量。Java 虚拟机产生对实例的域的符号引用，它被存储在常数池中。那些常数池项在运行期被解析，以确定实际的域偏移量。

## 7.9 数组

Java 虚拟机数组也是对象。数组使用不同的指令集创建和操纵。newarray 指令用于创建数值类型的数组。代码

```
void createBuffer() {  
    int buffer[];  
    int bufsz=100;  
    int value=12;  
    buffer=new int [bufsz];  
    buffer[10]=value;  
    value=buffer[11];  
}
```

必须编译为

```
方法 void createBuffer()  
0 bipush 100      //Push bufsz  
2 istore_2        //Store bufsz in local2  
3 bipush 12       //Push value
```

```

5 istore_3          //Store value in local 3
6 iload_2           //Push bufsz...
7 newarray int      //...and create new array of int
9 astore_1           //Store new array in buffer
10 aload_1           //Push buffer
11 bipush 10         //Push constant 10
13 iload_3           //Push value
14 iastore          //Store value at buffer[10]
15 aload_1           //Push buffer
16 bipush 11         //Push constant 11
18 iaload            //Push value at buffer [11]
19 istore_3          //...and store it in value
20 return

```

anewarray 指令用于创建对象引用的一维数组：

```

void createThreadArray() {
    Thread threads[];
    int count=10;
    threads=new Thread[count];
    thread[0]=new Thread();
}

```

成为

```

方法 void CreateThreadArray()
0 bipush 10          //Push 10...
2 istore_2           //...and initialize count to that
3 iload_2            //Push count,used by anewarray
4 anewarray class #1 //Create new array of class Thread
7 astore_1            //Store new array in threads
8 aload_1             //Load value of threads on stack
9 iconst_0            //Load 0 into stack
10 new #1             //Create instance of class Thread
13 dup                //Make duplicate reference...
14 invokespecial #5  //...to pass to initialization method
                     //Method java.lang.Thread.<init>()V
17 aastore            //Store new Thread in array at 0
18 return

```

`anewarray` 指令还可用于创建多维数组的第一维。或者 `multianewarray` 指令可用于一次创建多个维。例如，如下三维数组：

```
int [] [] [] create3DArray() {  
    int grid[] [] [];  
    grid=new int [10][5][];  
    return grid;  
}
```

由如下代码创建：

```
方法 int create3DArray()[][]
0 bipush 10           //Push 10(dimension one)
2 iconst_5           //Push 5(dimension two)
3 multianewarray #1 dim #2 //Class[[[I,a three
                           // dimensional int array;
                           // only create first two
                           // dimensions
7 astore_1            //Store new array...
8 aload_1             //... then prepare to return it
9 areturn
```

`multianewarray` 指令的第一个操作数对要创建的数组类类型的常数池索引。第二个是要实际创建的数组类型的维数。`multianewarray` 指令可用于创建该类型的所有维，如 `create3DArray` 代码所示。注意到多维数组只是一个对象，并因而分别由 `aload-1` 和 `areturn` 指令装载和返回。有关数组类名称的信息见 § 4.4.1。

所有数组都具有相关的长度，它由 `arraylength` 指令访问。

## 7.10 编译开关

Java 的 `switch` 语句使用 `tableswitch` 和 `lookupswitch` 指令编译。`tableswitch` 指令在当 `switch` 可有效地由对目标偏移量表的索引表示时使用。如果 `switch` 的表达式值落在有效索引的范围外，则使用 `switch` 的 `default` 目标。

例如：

```
int chooseNear(int i) {  
    switch (i) {  
        case 0: return 0;
```

```

        case 1: return 1;
        case 2: return 2;
        default:return -1;
    }
}

```

编译为：

方法 int chooseNear(int)	
0 iload_1	// Load local 1 (argument i)
1 tableswitch 0 to 2:	//Valid indices are 0 through 2
0:28	//If i is 0, continue at 28
1:30	//If i is 1,continue at 30
2:32	//If i is 2,continue at 32
default:34	//Otherwise,continue at 34
28  iconst_0	// i was 0;push int 0...
29 ireturn	///...and return it
30  iconst_1	//i was 1; push int 1...
31 ireturn	///...and return it
32  iconst_2	//i was 2; push int 2...
33 ireturn	///...and return it
34  iconst_m1	//otherwise push int -1...
35 ireturn	///...and return it

Java 虚拟机的 tableswitch 和 lookupswitch 指令仅对 int 数据操作。因为对 byte、char 或 short 值的操作被内部提升为 int，表达式被求值为那些类型中的一个的 switch 被编译，好像它被求值为 int 类型。如果 chooseNear 方法正使用 shork 类型编写，将会产生与使用 int 类型时相同的 Java 虚拟机指令。其他数值类型必须缩窄为 int 类型，以便在 switch 中使用。

对于 switch 稀疏的情况，tableswitch 指令的表表示对于空间变得非高效了。可用 lookupswitch 指令作替代。lookupswitch 指令将 int 关键字(case 标签的值)同表中的目标偏移量配对。当执行 lookupswitch 指令时，switch 表达式的值被同表中的关键字比较。如果关键字中的一个同表达式的值匹配，在相关的目标偏移量处继续执行。如果没有关键字匹配，在 default 目标处继续执行。例如，编译代码：

```

int chooseFar(int i) {
    switch (i) {
        case -100:      return -1;
        case 0:         return 0;
        case 100:        return 1;
    }
}

```

```
    default:           return -1;
}
}
```

看上去就像 chooseNear 的代码,除了使用了 lookupswitch 指令:

```
方法 int chooseFar(int)
0 iload_1
1 lookupswitch 3:
    -100:36
    0:38
    100:40
    default:42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  const_m1
43  ireturn
```

Java 虚拟机指定 lookupswitch 指令的表必须按关键字排序,以便实现可使线性扫描更高效地查找。即使这样,lookupswitch 指令必须查找其关键字是否匹配,而不是简单地执行边界检查并像 tableswitch 一样索引到一个表。这样,在空间考虑允许作出一个选择时,tableswitch 指令可能地 lookupswitch 更高效。

## 7.11 对操作数栈的操作

Java 虚拟机有大量的指令补充,将操作数栈的内容作为无类型字或成对的无类型字操纵。因为 Java 虚拟机的对灵巧地操纵其操作数栈的依赖,所以这些是有用的。例如:

```
public long nextIndex() {
    return index++;
}
private long index=0;
```

编译为:

```

方法 long nextIndex()
0 aload_0      //Write this onto operand stack
1 dup        //Make a copy of it
2 getfield #4 //One of the copies of this is consumed
              //loading long field index onto stack,
              //above the original this
5 dup2_x1    //The long on top of the stack is
              //inserted into the stack below the
              //original this
6 lconst_1    //A long 1 is loaded onto the stack
7 ladd        //The index value is incremented
8 putfield #4 //and the result stored back in the field
11 lreturn    //The original value of index is left on
              //top of the stack, ready to be returned

```

注意到 Java 虚拟机并不允许其操作数栈操纵指令单独修改或移动其两个字的数据类型的字。

## 7.12 抛出和处理异常

异常用 throw 关键字从 Java 程序中抛出。其编译是简单的：

```

void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
}

```

成为：

```

方法 void cantBeZero(int)
0 iload_1          //Load argument 1(i) onto stack
1 ifne 12          //If i==0, allocate instance and throw
4 new #1           //Create instance of TestExc
7 dup              //One reference goes to the constructor
8 invokespecial #7 //Method TestExc.<init>()V
11 athrow           //Second reference is thrown
12 return           //Never get here if we threw TestExc

```

Java 的 try\_catch 的编译是简明的。例如：

```
void catchOne() {  
    try {  
        tryItOut();  
    } catch(TestExc e) {  
        handleExc(e);  
    }  
}
```

编译为：

方法 void catchOne()  
0 aload\_0 //Beginning of try block  
1 invokevirtual #6 //Method Example.tryItOut()V  
4 return //End of try block;normal return  
5 astore\_1 //Store thrown value in local variable 1  
6 aload\_0 //Load this onto stack  
7 aload\_1 //Load thrown value onto stack  
8 invokevirtual #5 //Invoke handler method:  
 // Example.handleExc(LTestExc;)V  
11 return //Return after handling TestExc  
  
异常表：  
从 到 目标 类型  
0 4 5 类 TestExc

更仔细地看，try 块就像 try 不存在一样被编译：

方法 void catchOne()  
0 aload\_0 //Beginning of try block  
1 invokevirtual #4 //Method Example.TryItOut()V  
4 return //End of try block;normal return

如果在 try 块执行中无异常抛出，它表现得好像 try 不存在：tryItOut 被调用而 catchOne 返回。

跟在 try 块后的是实现一个 catch 子句的 Java 虚拟机代码：

```
5 astore_0          //Store thrown value in local variable1  
6 aload_0          //Load this onto stack
```

```

7 aload_1          //Load thrown value onto stack
8 invokevirtual #5 //Invoke havdler method:
                  //Example. handleExc(LTestExc;) V
11 return         //Return after handling TestExc
异常表:
从 到 目标    类型
0   4   5       类 TestExc

```

handleExc 的调用,catch 子句的内容,也像普通方法调用一样编译。然而,catch 子句的存在导致编译器产生异常表表项。catchOne 方法的异常表具有对应于一个 catchOne 的 catch 字句可处理的参数(TextExc 类的实例)的表项。如果一些 TextExc 实例的值在指令执行当中抛出,这些指令在 catchOne 中索引0到索引4(包括)之间,则控制被转移到索引5处的 Java 虚拟机代码处,它实现 catch 子句的块。如果被抛出的值不是 TextExc 的实例,catchOne 的 catch 子句不能处理它。相反,该值被重抛回给 catchOne 的调用者。

try 可以有多个 catch 子句:

```

void catchTwo() {
    try {
        tryItOut();
    } catch (TestExc1 e) {
        handleExc(e);
    } catch (TestExc2 e) {
        handleExc(e);
    }
}

```

给定的 try 语句的多个 catch 子句通过一个接一个地简单地为每个 catch 子句添加 Java 虚拟机代码,并向异常表添加表项来编译:

```

方法 void catchTwo()
0  aload_0          //Begin try block
1  invokevirtual #5 //Method Example. tryItOut()V
4  return           //End of try block;normal return
5  astore_1          //Beginning of handler for TestExc1;
                     //Store thrown value in local variable 1
6  aload_0          //Load this onto stack
7  aload_1          //Load thrown value onto stack
8  invokevirtual #7 //Invoke handler method:
                     //Example. handleExc(LTestExc1;)V

```

```

11 return          //Return after handling TestExc1
12 astore_1        //Beginning of handler for TestExc2;
                  Store thrown value in local variable 1
13 aload_0         //Load this onto stack
14 aload_1         //Load thrown value onto stack
15 invokevirtual #7 //Invoke handler method:
                  //Example. handleExc(LTestExc2;)V
18 return          //Return after handling TestExc2

```

异常表:

从	到	目标	类型
0	4	5	类 TestExc1
0	4	12	类 TestExc2

如果在 try 子句执行中(在索引0和4间)一个值被抛出,它匹配一个或更多 block 块参数(该值是一个或更多参数的实例),第一个(最左边)这样的 catch 子句被选择。控制被转移到该 catch 子句块的 Java 虚拟机代码。如果抛出的值不同。catchTwo 的任何 catch 子句的参数匹配,Java 虚拟机重抛出该值,而无需调用 catchTwo 的任何 catch 子句中的代码。

嵌套的 try\_catch 语句的编译非常象有多个 catch 子句的 try 语句:

```

void nestedCatch() {
    try {
        try {
            tryItOut();
        } catch (TestExc1 e) {
            handleExc1(e);
        }
    } catch (TestExc2 e) {
        handleExc2(e);
    }
}

```

成为:

```

方法 void nestedCatch()
0 aload_0          //Begin try block
1 invokevirtual #8 //Method Example.tryItOut()V
4 return           //End of try block;normal return
5 astore_1          //Beginning of handler for TestExc1;

```

```

        //Store thrown value in local variable 1
6  aload_0          //Load this onto stack
7  aload_1          //Load thrown value onto stack
8  invokevirtual #7 //Invoke handler method:
                    //Example. handleExc1(LTestExc1;)V
11 return           //Return after handling TestExc1
12 astore_1         //Beginning of handler for TestExc2;
                    //Store thrown value in local variable 1
13 aload_0          //Load this onto stack
14 aload_1          //Load thrown value onto stack
15 invokevirtual #6 //Invoke handler method:
                    // Example. handleExc2(LtestExc2;)V
18 return           // Return after handling TestExc2

```

异常表:

从	到	目标	类型
0	4	5	TestExc1
0	12	12	TestExc2

catch 子句的嵌套仅在异常表中表示。当一个异常被抛出时,包含异常场点并带有匹配参数的最内层 catch 子句被选择来处理异常。例如,如果 tryItOut(索引1处)的调用抛出 TestExc1 实例,它将会由调用 handleExc1 的 catch 子句处理。即使异常出现在外部 catch 子句(捕捉 TestExc2)的边界内也是如此,而且即使外部 catch 子句也能处理抛出值时也是如此。

作为一点细微之处,注意到 catch 子句的范围在“从”端是包括的,而在“到”端是不包括的(见 § 4.7.4)。这样,捕捉 TestExc2 的 catch 子句的异常表项覆盖偏移量11处的 return 指令。嵌套的 catch 子句内的返回指令包括在由嵌套的 catch 子句覆盖的指令范围内。

## 7.13 编译 finally

try-finally 语句的编译同 try-catch 相似。在将控制转移到 try 语句外部前,无论该转移是正常的还是突然的,因为异常已被抛出,finally 子句必须首先被执行。简单范例:

```

void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}

```

编译代码为：

```
方法 void tryFinally()
0 aload_0          //Beginning of try block
1 invokevirtual #6 //Method Example.tryItOut()V
4 jsr 14           //Call finally block
7 return            //End of try block
8 astore_1          //Beginning of handler for any throw
9 jsr 14           //Call finally block
12 aload_1          //Push thrown value,
13 athrow            //and rethrow the value to the invoker
14 astore_2          //Beginning of finally block
15 aload_0          //Push this onto stack
16 invokevirtual #5 //Method Example.wrapItUp()V
19 ret 2            //Return from finally block
```

异常表

从	到	目标	类型
0	4	8	任何

控制可有四种方式来传递到 try 语句外：在块的底部失败，返回，执行 break 或 continue 语句，或者引发异常。如果 tryItOut 返回而不引发异常，控制使用 jsr 指令转移到 finally 块。索引4处的 jst14指令对索引14处的 finally 块代码进行“子例程调用”(finally 块作为嵌入子例程编译)。当 finally 块结束时，ret2指令将控制返回到索引4处 jsr 指令后面的指令。

更详细地，子例程调用以如下方式进行：jsr 指令在跳栈前将随后指令(在索引7处返回)的地址压入操作数栈。作为跳转目标的 astore\_2指令将操作数栈上的地址存储到局部变量2。finally 块的代码(在此为 aload\_0和 invokevirtual 指令)被运行。假定该代码的执行正常结束，ret 指令从局部变量2获取地址并从该地址处继续执行。return 指令被执行，tryFinally 正常返回。带有 finally 子句的 try 语句被编译为具有特定的异常处理器，它可以处理任何在 try 语句内抛出的异常。如果 tryItOut 抛出异常，则在 tryFinally 的异常表中查找适当的异常处理器。找到特定的处理器，导致执行在索引8处继续。索引8处的 astore 指令将抛出的值存储到局部变量1。随后的 jsr 指令对 finally 块的代码作子例程调用。假定代码正常返回，索引12处的 aload-1指令将抛出值压回操作数栈，随后的 athrow 指令重抛出该值。

编译同时有 catch 子句和 finally 子句的 try 语句更为复杂：

```
void tryCatchFinally() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
```

```

    } finally {
        wrapItUp();
    }
}

```

成为：

```

方法 void tryCatchFinally()
0 aload_0          //Beginning of try block
1 invokevirtual #4 //Method Example.tryItOut()V
4 goto 16          //Jump to finally block
7 astore_3          //Beginning of handler for TestExc;
                   //Store thrown value in local variable 3
8 aload_0          //Push this onto stack
9 aload_3          //Push thrown value onto stack
10 invokevirtual #6 //Invoke handler method:
                   //Example.handleExc(LTestExc;)V
13 goto 16          //Huh???[1]
16 jsr 26          //Call finally block
19 return          //Return after handling TestExc
20 astore_1          // Beginning of handler for exceptions
                   //other than TestExc, or exceptions
                   //thrown while handling TestExc
21 jsy 26          //Call finally block
24 aload_1          //Push thrown value,
25 athrow          //and rethrow the value to the invoker
26 astore_2          //Beginning of finally block
27 aload_0          //Push this onto stack
28 invokevirtual #5 //Method Example.wrapItUp()V
31 ret 2           //Return from finally block

```

异常表

从	到	目标	类型
0	4	7	类 TestExc
0	16	20	任何

如果 try 语句正常结束,索引4处的 goto 指令跳转到对索引16处 finally 块的子例程调用。执行索引26处的 finally 块,控制返回到索引19处的 return 指令,而 tryCatchFinally 正常返回。

如果 tryItOut 抛出 TestExc 的实例,异常表中第一个适用的异常处理器被选择来处理该

---

[1] 此 goto 指令严格地讲是不必要的,但它由 Sun 的 JDK 1.0.2 版的 java c 编译器产生。

异常。该异常处理者的代码，从索引7处开始，将抛出值传递给 handleExc，并且在它返回时象在通常情况下一样，对索引26处的 finally 块作同样的子例程调用。如果异常不是 handleExc 抛出的，tryCatchFinally 正常返回。

如果 tryItOut 抛出的值不是 TestExc 的实例，或者如果 handleExc 本身抛出异常，该情况由异常表中第二个表项处理，它处理任何索引0和16间的任何抛出值。该异常处理者将控制转移到索引20，抛出值首先被存储到局部变量1。索引26处的代码称为子例程。如果它返回，抛出值从局部变量1获取，并使用 athrow 指令重新抛出。如果在 finally 子句执行中抛出了新的值，则 finally 子句异常结束，tryCatchFinally 正常返回，向其调用者抛出新值。

## 7.14 同步

Java 虚拟机通过其 monitoreenter 和 monitorexit 指令对同步提供显式支持。然而对于 Java，可能最普通的同步形式是 synchronized 方法。

synchronized 方法通常不是使用 moniterenter 和 moniterexit 实现的。相反，它可通过由方法调用指令检查的 ACC\_SYNCHRONIZED 标志，来简单地在常数池中区分。当调用 ACC\_SYNCHRONIZED 所设置的方法时，当前线程获取一个监视器，调用方法本身，并且无论方法调用正常结束还是突然结束，都释放该监视器，在执行线程拥有监视器时，其他线程不可能获得它。如果在 synchronized 方法调用中抛出了异常，而且 synchronized 方法不处理该异常，该方法的监视器在该异常被重抛出到 synchronized 方法外部之前自动释放。

monitoreenter 和 monitorexit 指令为支持 Java 的 synchronized 语句而存在。synchronized 语句获得代表执行线程的监视器，执行语句体，然后释放监视器。

```
void onlyMe(Foo f) {  
    synchronized(f) {  
        doSomething();  
    }  
}
```

同步语句的编译是简明的：

```
方法 void onlyMe(Foo)  
0 aload_1          // Load f onto operand stack  
1 astore_2          // Store it in local variable2  
2 aload_2          // Load local variable 2 (f) onto stack  
3 monitoreenter    // Enter the monitor associated with f  
4 aload_0          // Holding the monitor, pass this and  
5 invokevirtual #5 // call Example. doSomething()V  
8 aload_2          // Load local variable 2 (f) onto stack  
9 monitorexit      // Exit the monitor associated with f
```

```
10 return          //Return normally  
11 aload_2         //In case of any throw, end up here  
12 monitorexit     //Be sure to exit monitor,  
13 athrow          //then rethrow the value to the invoker
```

### 异常表

从	到	目标	类型
4	8	11	任何

## 第八章 线程和锁

本章详述低层动作,它可用于解释有共享主存储器的 Java 虚拟机线程的交互作用,它已被重印,和 James Gosling、Bill Joy 和 Guy steale 的《Java 语言规范》一书有微小的差异。

### 8.1 术语和框架

变量(variable)是 Java 程序中任何可被存储到的位置。还不仅包括类变量和类实例,而且包括数组成分。变量保存在由所有线程共享的主存储器(main memory)中。因为一个线程不可能访问另一个线程的参数或局部变量,参数和局部变量是被认为驻留在共享主存储器中、还是在拥有它们的线程的工作存储器并不重要。

每个线程都有一个工作存储器(working memory),它在其中保存它必须使用的或赋值的变量的工作拷贝(working copy)。在线程执行 Java 程序时,它对这些工作拷贝操作。主存储器包含每个变量的主拷贝(master copy)。有一些规则,关于何时线程被允许或需要将变量工作拷贝的内容传送到主拷贝或反之。

主存储器也包含锁(locks):每个对象有一个与其关联的锁。线程可以竞争以获得一个锁。

出于本章的目的,动词使用(use)、赋值(assign)、装载(load)、存储(store)、锁定(lock)和解锁(unlock)命名线程可以执行的动作。动词读(read)、写(write)、锁定(lock)和解锁(unlock)命名主存储器子系统可以执行的动作。这些操作的每一个都是原子的(不可分的)。

使用(use)和赋值(assign)操作是线程的执行引擎和线程的工作存储器间紧密耦合的交互作用。锁定(lock)和解锁(unlock)操作是线程的执行引擎和主存储器间的紧密耦合的交互作用。但在主存储器和线程的工作存储器间的数据传送是松散耦合的。当数据从主存储器复制到工作存储器时,必须出现两种动作:由主存储器执行的读(read)操作,一段时间后是由工作存储器执行的相应的 load 操作。当数据从工作存储器复制到主存储器时,必须出现两种动作:由工作存储器执行的存储(store)操作,一段时间后是由主存储器执行的相应的写(write)操作。在主存储器和工作存储器间可能有一段转接时间,并且对每种交互作用转接时间可能不同;这样,由线程初始化的对不同变量的操作,可能被另一个线程看作以不同的顺序发生。然而对每个变量,在代表任何一个线程的主存储器中的操作,按照同该线程相应操作相同的顺序执行(这在后面更详细地解释)。

一个 Java 线程发送使用(use)、赋值(assign)、锁定(lock)和解锁(unlock)操作流,按照它正执行的 Java 程序的语义所提出的。基础 Java 实现被另外要求执行适当的装载(load)、存储(store)、读(read)和写(write)操作,以便遵守某个约束集,这将在后面解释。如果 Java 实现正确地遵守这些规则并且 Java 应用程序的程序员遵守程序设计的某些其他规则,则数据可以通过共享变量在线程间可靠地传送。这些规则被设计得足够“紧”以实现这种可能,但足够“松”以允许硬件和软件设计者相当自由地通过像寄存器、队列和高速缓存这样的机制来提高速度和吞吐量。

这里是每种操作的详细定义：

- 使用(use)动作(由线程)将一个变量的线程工作拷贝传送到线程的执行引擎。无论何时线程执行了使用变量值的虚拟机指令,该动作都将执行。
- 赋值(assign)动作(由线程)将一个值从线程的执行引擎传送到变量的线程工作拷贝。无论何时线程执行了对变量赋值的虚拟机指令,该动作都将执行。
- 读(read)动作(由主存储器)将变量主拷贝的内容传送到线程的工作存储器,供后面的装载(load)操作使用。
- 装载(load)动作(由线程)将由读(read)动作从主存储器中传送的值放入一个变量的线程工作拷贝中。
- 存储(store)动作(由线程)将一个变量的线程工作拷贝的内容传送到主存储器,供后面的写(write)操作使用。
- 写(write)动作(由主存储器)将由存储(store)动作从线程的工作拷贝中传送的值放入主存储器中一个变量的主拷贝。
- 锁定(lock)动作(由同主存储器紧密同步的线程)导致线程获得特定锁上的一个声明。
- 解锁(unlock)动作(由同主存储器紧密同步的线程)导致线程释放特定锁上的一个声明。

这样,线程同变量的交互作用过程由使用(use)、赋值(assign)、装载(load)和存储(store)操作的序列组成。主存储器对每次装载(load)执行读(read)操作,而对每次存储(store)执行写(write)操作。线程同锁的交互作用过程由锁定(lock)的解锁(unlock)操作序列组成。线程的所有全局可见行为包含线程的所有对变量和锁的操作。

## 8.2 执行顺序和一致性

执行顺序规则约束了某些事件可能发生的顺序。有四种关于动作间关系的一般约束：

- 由任何一个线程执行的动作全被排序,即对任何由线程执行的动作,一个动作优先于另一个。
- 对任何一个变量的由主存储器执行的动作全被排序,即对于由同一变量的由主存储器执行的任何两个动作,一个动作先于另一个。
- 对任何锁的由主存储器执行的动作全被排序,即对于同一个锁的由主存储器执行的任何两个动作,一个动作优先于另一个。
- 不允许一个动作跟在它自身后面。

最后一条规则可能看上去无关紧要,但必须分开来显式地声明,以保持完整性。没有它,可能会由两个或更多线程提交一组动作,动作间的优先关系可能满足其他所有规则,但会要求一个动作跟在它自身后面。

线程不直接交互作用,它们仅通过主存储器通信。线程的动作和主存储器的动作间的关系在三方面受到约束:

- 每个锁定(lock)或解锁(unlock)动作由一些线程和主存储器共同执行。

- 每个由线程执行的装载(load)动作,唯一地同由主存储器执行的写(write)动作配对,这样写(write)动作跟在存储(store)动作后。

后面各节中的大多数规则进一步约束某些动作发生的顺序。一条规则可以声明一个动作必须先于或晚于其他一些动作。注意到这种关系是暂时的:如果动作 A 必须先于动作 B,而 B 必须先于 C,则 A 必须先于 C。程序员必须记住这些规则是对于动作顺序的仅有的约束;如果没有规则或规则的组合指示动作 A 必须先于动作 B,则 Java 实现可自由地在动作 A 之前执行动作 B,或者并发地执行动作 A 和动作 B。这种自由性可能是良好性能的关键。相反,不要求一个实现利用给予它的所有自由性。

### 8.3 有关变量的规则

令 T 为一个线程,而 V 为一个变量。对由关于 V 的由 I 执行的操作有某些约束:

- 按标准的 Java 执行模型,仅当由 Java 程序的 T 执行提出时,才允许由 T 执行的对 V 的使用或赋值。例如,作为 + 操作符的操作数出现的 V,要求在 V 上发生一个使用操作;作为赋值操作符 = 的左侧操作符出现的 V,需要发生一个赋值操作。所有由给定线程执行的使用和赋值动作,必须按线程正执行的程序所指定的顺序发生。如果后面的规则禁止 T 执行所需的使用(use)作为其下一动作,T 可能需要首先执行装载以便推进。
- 由 T 执行的对 V 的存储操作必须插入到由 T 执行的对 V 的赋值和后继的由 T 执行的对 V 的装载之间(较不正式地:不允许线程丢掉最近的赋值)。
- 由 T 执行的对 V 的赋值操作必须插入到由 T 执行的对 V 的装载或存储和后继的由 T 执行的对 V 的存储之间(较不正式地:不允许线程无故将数据从其工作存储器写回主存储器)。
- 在一个线程被创建后,它在对变量执行使用或存储操作之前,必须执行赋值和装载操作(较不正式地:新线程开始时带有空的工作存储器)。
- 在一个变量被创建后,每个线程必须在对该变量执行使用或存储操作之前,对该变量执行赋值或装载(较不正式地:新变量仅在主存储器中创建,并且初始时不在任何线程的工作存储器中)。

假定 § 8.3、§ 8.6 和 § 8.7 中的所有约束被遵守,则对任何变量的存储操作可在实现过程中由任何线程在任何时间发出。

对于由主存储器执行的读和写操作也有某些约束:

- 对于每个由线程 T 执行的对变量 V 的工作拷贝的装载操作,必须有由主存储器执行的对 V 的主拷贝的相应优先的读操作,而且装载操作必须将由相应读操作传送的数据放入工作拷贝。
- 对于每个由线程 T 执行的对变量 V 的工作拷贝的存储操作,必须有由主存储器,执行的对 V 的主拷贝的相应后继的写操作,而且写操作必须将由相应存储操作传送的数据放入主拷贝。
- 令动作 A 是由线程 T 执行的对变量 V 的装载或存储动作,并令动作 P 为由主存储器执

行的、对变量 V 相应的读或写动作。相似地，令动作 B 为由线程 T 执行的对同一变量 V 的其他一些装载或存储动作，并且令动作 Q 为由主存储器执行的，对变量 V 的相应的读或写动作。如果 A 先于 B，则 P 必须优于 Q（较不正式地：对主拷贝的操作由主存储器严格地按线程请求的顺序执行，这些主拷贝是任何给定的代表线程的变量的主拷贝）。

注意到这最后一条规则仅适用于由线程执行的对同一变量的动作。然而，对 volatile 变量（§ 8.7）有更严格的规则。

## 8.4 Double 和 Long 变量的非原子处理

如果 double 或 long 变量未声明为 volatile，则出于装载、存储、读和写操作的目的，按照好像它是两个各有32位的变量来处理：无论何时规则需要这些操作中的一种，将执行两个这样的操作，每个对32位部分进行。Double 或 long 变量的64位编码为32位量的方式以及对变量的操作的顺序未由《Java 语言规范》一书定义。

这只是因为 double 或 long 变量的读或写，可以由实际的主存储器处理，如同可在时间上分开进行的32位读或写操作，其他操作可在其中间出现。因而，如果两个线程并发地将不同的值赋予同一个共享的非 volatile double 或 long 变量，则该变量随后的使用可能获得不同于任一所赋予值的值，但一些实现依赖于两个值的混合。

一种实现可自由地将对 double 和 long 值的装载、存储、读和写操作，作为原子的64位操作实现。事实上，这是大受鼓励的。那种将它们分为32位的两部分的模型，是因为几种当前流行的微处理器不能在64位数量上提供高效的原子存储器处理。如果 Java 将对一个变量的所有存储器处理定义为原子的将更为简单，这种更复杂的定义是对当前硬件条件的实际让步。将来这种让步将被消除。同时，程序员总被警告要显式地同步对共享 double 和 long 变量的访问。

## 8.5 有关锁的规则

令 T 为线程，而 L 为锁。对于由 T 执行的关于 L 的操作有某些约束。

- 由 T 执行的对 L 的锁定操作可能仅当如下情况时发生，即对每个不是 T 的线程 S，先前由 S 执行的对 L 的解锁操作数，等于先前由 S 执行的对 L 的锁定操作数。（较不正式地：一次只允许一个线程提出锁定要求；此外，线程可以多次获得同一个锁，而不用放弃对它的所有权，直到已执行了匹配的解锁操作。）
- 由线程 T 执行的对锁 L 的解锁操作，仅当如下情况时发生，即先前由 T 执行的对 L 的解锁操作，严格小于先前由 T 执行的对 L 的锁定操作数（较不正式地：不允许线程解锁它不拥有的锁）。

## 8.6 有关锁和变量的交互作用的规则

令 T 为任意线程，令 V 为任意变量，而令 L 为任意锁。对于由 T 执行的关于 V 和 L 的操作有某些约束。

- 在由 T 执行的对 V 的赋值操作和随后由 T 执行的对 L 的解锁操作之间, 必须插入由 T 执行的对 V 的存储操作; 此外, 对应于该存储的写操作必须先于解锁操作, 如同为主存储器所见的一样(较不正式地: 如果线程要对任意锁执行解锁操作, 它必须首先将其工作存储器中的所有赋予值复制回主存储器)。
- 在由 T 执行的对 L 的锁定操作, 和随后由 T 执行的对变量 V 的使用和存储操作之间, 必须插入赋值或装载操作; 此外, 如果是装载操作, 则对该装载操作的读操作必须跟在锁定操作后, 如同为主存储器所见的一样(较不正式地: 锁定操作表现得好像从线程的工作存储器中清除所有变量, 在其操作后必须自己为它们赋值或从主存储器再度装载其拷贝)。

## 8.7 有关易变变量的规则

如果变量声明为易变类型, 则附加约束适用于每个线程的操作。令 T 为线程, 令 V 和 W 为易变变量。

- 由 T 执行的对 V 的使用操作, 仅当先前由 T 执行的对 V 的操作为装载时才被允许。由 T 执行的对 V 的装载操作仅当下一个操作是由 T 执行的对 V 的使用时才被允许。使用操作被认为同对应于装载的读操作“相关”。
- 由 T 执行的对 V 的存储操作, 仅当先前由 T 执行的对 V 的操作为赋值时才被允许。由 T 执行的对 V 的赋值操作, 仅当下一操作是由 T 执行的对 V 的存储时才被允许。赋值操作被认为与对应于存储的写操作“相关”。
- 令动作 A 为由 T 执行的对 V 的使用或赋值, 令动作 F 为同 A 相关的装载或存储, 并令动作 P 为对应于 F 的对 V 的读或写。类似地, 令动作 B 为由 T 执行的对变量 W 的使用或赋值, 令动作 G 为与 B 相关的装载或存储, 并令动作 Q 为对应于 G 的对 V 的读或写。如果 A 先于 B, 则 P 必须先于 Q。(较不正式地: 对代表线程的易变变量主拷贝的操作, 由主存储器严格地按线程请求的顺序执行)。

## 8.8 先见存储操作

如果变量未声明为 volatile, 则前面各节中的规则可略微极松, 以便让存储操作比原来允许的更早。这种放松的目的在于允许优化 Java 编译器, 以执行某种代码重排列, 维护适当的同步程序的语义, 但可能在由未适当同步的程序执行的无序的存储器操作中被捕捉。

假定由 T 执行的对 V 的存储将跟在由 T 执行的对 V 的特定赋值操作后面, 按照先前各节中的规则, 而不插入由 T 执行的对 V 的装载或赋值操作。这样存储操作将把赋值操作放入线程 T 的工作存储器中的值, 传送到主存储器。相反, 这种特殊规则允许存储操作事实上出现在赋值操作之前, 如果下述限制被遵守:

- 如果发生存储操作, 必定发生赋值。(记住, 这些是对实际发生的操作的限制, 而不是对线程要做的操作的限制。在赋值发生前不执行存储并抛出异常。)
- 重定位的存储和赋值间未插入锁定操作。
- 重定位的存储和赋值间未插入对 V 的装载。

- 重定位的存储和赋值间未插入其他对 V 的存储。
- 存储操作将赋值操作要放到线程 T 的 Z 作存储器中的值传送到主存储器。

这最后一个属性启发我们有先见之明地调用这样的提早存储操作：它要以某种方法预先知道，要由原该它跟随的赋值操作存储什么值。实际上，优化的编译代码将提早计算这种值（例如，计算没有副作用并且不抛出异常，则它被允许），提早存储它们（例如，在进入循环之前），并将它们保存在工作寄存器中，以便以后在循环中使用。

## 8.9 讨论

锁和变量间的任何关联都纯粹是约定。锁定任何锁概念上从线程的工作存储器中清除所有变量，而解锁任何锁强制将所有线程赋予的变量写到主存储器中。在一些应用程序中，在访问对象的任何实例变量前，总锁定该对象是适当的。例如，`synchronized` 方法是遵守这种约定的方便的途径。在其他应用程序中，可能对于同步访问大的对象集合使用一个锁就够了。

如果只在锁定特定的锁之后和在同一个锁被相应解锁前，使用一个特定的共享变量，则该线程将在锁定操作后，从主存储器读取该变量的共享值，如果需要，并在解锁操作前将最近赋予该变量的值复制回主存储器。这和有关锁的相互排斥规则一起，足够保证值通过共享变量，正确地从一个线程传送到另一个。

有关易变变量的规则实际上要求，对于每个由线程执行的对易变变量的使用或赋值，主存储器恰被涉及一次，还要求刚好按线程执行语义提出的顺序涉及主存储器。然而，这样的存储器操作，对于对非易变变量操作的读和写并不是有序的。

## 8.10 范例：可能的交换

考虑具有类变量 a 和 b，以及方法 `hither` 和 `yon` 的类：

```
class Sample {
    int a=1,b=2;
    void hither() {
        a=b;
    }
    void yon()
        b=a;
}
```

现在假定创建了两个线程，一个线程调用 `hither`，而另一个线程调用 `yon`。所需的动作集是什么，有序的约束是什么？

让我们考虑调用 `hither` 的线程。按照规则，此线程必须执行 b 的使用，其后是 a 的赋值。这

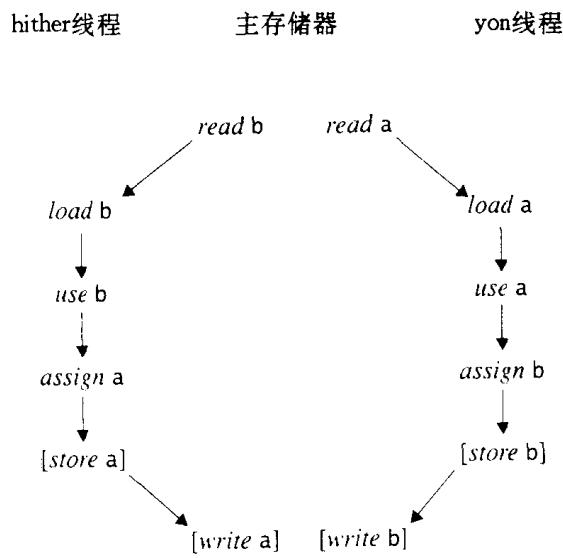
是执行对方法 hither 的调用的仅有的最小要求。

现在,由线程执行的对变量 b 的第一个操作不能是使用。但它可以是赋值或装载。不能发生对 b 的赋值,因为程序正文不调用这样的赋值操作,所以需要 b 的装载。这个由线程执行的装载操作需要先前有由主存储器执行的对 b 的读操作。

线程可以在赋值发生后有选择地存储 a 的值。如果它保存了,则存储操作需要后面是由主存储器执行的对 a 的写操作。

调用 yon 的线程的情况类似,但 a 和 b 的角色互换了。

全部操作集可以描绘如下:



这里从动作 A 到动作 B 的箭头指示 A 必须先于 B。

主存储器可能以何种顺序执行操作?仅有的约束是不可能同时出现 a 的写先于 a 的读,而 b 的写先于 b 的读,因为图表中不经心的箭头将形成循环,这样一个动作将不得不先于它自身,这是不允许的。假定发生了可选的存储和写操作,主存储器可以有三种可能的顺序来合法地执行其操作。令 ha 和 hb 为 hither 线程的 a 和 b 的工作拷贝,令 ya 和 yb 为 yon 线程的工作拷贝,并且令 ma 和 mb 为主存储器中的主拷贝。初始时 ma=1 而 mb=2。操作的三种可能顺序和结果状态如下:

- write a → read a. read b → write b (then ha=2, hb=2, ma=2, mb=2, ya=2, yb=2)
- read a → write a. write b → read b (then ha=1, hb=1, ma=1, mb=1, ya=1, yb=1)
- read a → write a. read b → write b (then ha=2, hb=2, ma=2, mb=1, ya=1, yb=1)

这样,最终结果可能是,在主存储器中,b 被复制到 a,a 复制到 b,或者 a 和 b 的值交换;此外,变量的工作拷贝可能一致或不一致。当然,假定这些结果中的任何一个都比其他的更可能将是不正确的。这里 Java 程序的行为必须依赖于时间。

当然,一种实现也可以选择不执行存储和写操作,或者只执行两者之一,导致其他可能的结果。

现在假定我们修改范例来使用 synchronized 方法:

```
class SynchSample {
```

```

int a=1,b=2;
synchronized void hither() {
    a=b;
}
synchronized void yon()
    b=a;
}

```

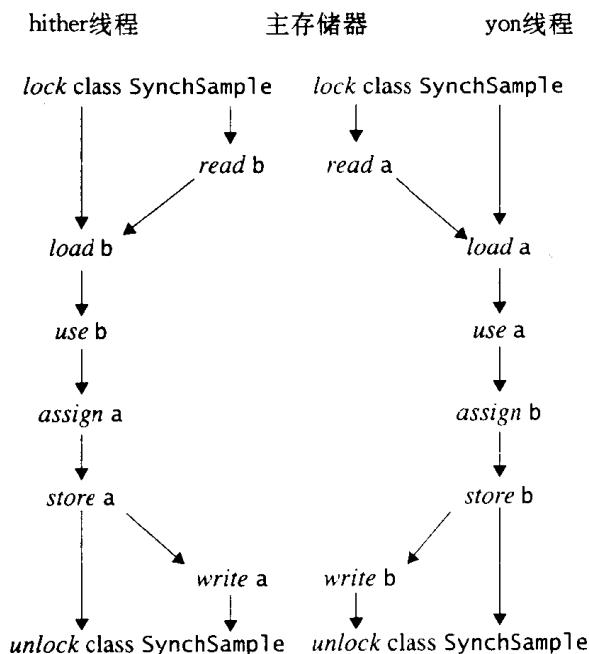
让我们再次考虑调用 hither 的那个线程。按照规则,该线程必须在 hither 方法的主体执行前,执行锁定操作(对 SynchSample 类的 class 对象)。其后是 b 的使用和 a 的赋值。最后,对 class 对象的解锁操作必须在 hither 方法的主体结束后执行。这是执行对 hither 方法的调用的仅有的最小要求。

像以前一样,需要 b 的装载,它需要前面有由主存储器进行的对 b 的读操作。因为装载在锁定操作后面,相应的读也必须在锁定操作后面。

因为解锁操作在 a 的赋值后面,对 a 的存储操作是强制的,它需要后面是由主存储器执行的对 a 的写操作。写必须先于解锁操作。

调用 yon 的线程的情况类似,但 a 和 b 的角色互换了。

全部操作集可以描绘如下:



锁定和解锁操作对由主存储器执行的操作的顺序提出进一步的约束;由一个线程执行的锁定不能发出在另一个线程的锁定和解锁操作间。此外,解锁操作要求发出存储和写操作。其结果是只有两种序列是可能的:

- write a → read a. read b → write b (then ha=2, hb=2, ma=2, mb=2, ya=2, yb=2).
- read a → write a. write b → read b (then ha=1, hb=1, ma=1, mb=1, ya=1, yb=1)

虽然结果状态是依赖于时间的,但可以看出,两个线程在 a 和 b 的值上必然一致。

## 8.11 范例:无序写

此范例同前面小节中的类似,只不过一个方法对两个变量赋值,而另一个方法读取两个变量。考虑具有类变量 a 和 b,和方法 to 和 fro 的类:

```
class Simple {
    int a=1,b=2;
    void to() {
        a=3;
        b=4;
    }
    void fro()
        System.out.println("a="+a+",b="+b);
    }
}
```

现在假定创建了两个线程,一个线程调用 to,而另一个线程调用 fro。需要什么动作集,有序的约束是什么?

让我们考虑调用 to 的线程。按照规则,该线程必须执行 a 的赋值,然后是 b 的赋值。这是执行对方法 to 的调用的仅有的最小要求。因为没有同步,由实现选择是否将赋值的值存回主存储器!因而,调用 fro 的线程可获取 a 的值1或3,并且独立地,可获取 b 的值2或4。

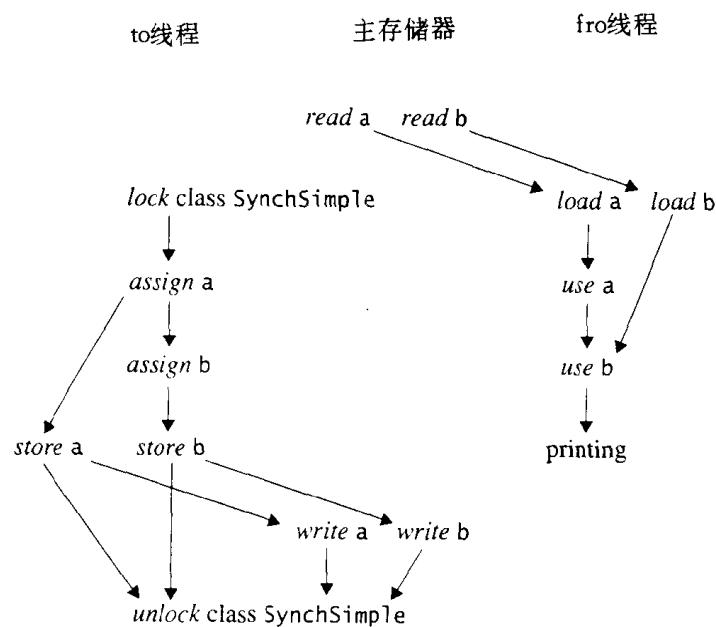
现在假定 to 为 Synchronized,但 fro 不是:

```
class SynchSimple {
    int a=1,b=2;
    synchronized void to() {
        a=3;
        b=4;
    }
    void fro()
        System.out.println("a="+a+",b="+b);
    }
}
```

在这种情况下,将强制方法 to 在方法结尾处的解锁操作执行前,将赋值的值存回主存储。

方法 fro 当然必须使用 a 和 b(以该顺序), 并且必须从主存储装载 a 和 b 的值。

全部操作集可以描绘如下:



这里从动作 A 到动作 B 的箭头指示 A 必须先于 B。

由主存储器执行的操作可能以什么顺序发生? 注意到规则不要求写 a 发生在写 b 前; 它们也不要求读 a 发生在读 b 前。同时, 即使方法 to 是同步的, 方法 fro 也不同步, 所以无法阻止读操作发生在锁定和解锁操作间。(要点是仅仅声明一个方法为 Synchronized 方法, 并不使该方法表现得好象它是原子的)。

结果, 方法 fro 可能仍然得到 a 的值 1 或 3, 或可独立地获得 b 的值 2 或 4。特别地, fro 可以观察 a 的值 1 和 b 的值 4。这样, 即使 to 对 a 赋值, 然后对 b 赋值, 对主存储器的写操作在另一个线程看来好象以相反的顺序发生。

最后, 假定 to 和 fro 都是同步的:

```
class SynchSynchSimple {  
    int a=1,b=2;  
    synchronized void to() {  
        a=3;  
        b=4;  
    }  
    synchronized void fro()  
        System.out.println("a="+a+",b="+b);  
    }  
}
```

在此情况下,方法 fro 的动作不能同方法 to 的动作交错,因此 fro 将打印 "a=1,b=2" 或 "a =3,b=4"。

## 8.12 线程

线程由 Thread 和 ThreadGroup 类创建并管理。创建 Thread 对象就创建一个线程,而且它是创建线程的唯一方式。当线程被创建时,它还不是活跃的;当其 start 方法被调用时,它开始运行。

## 8.13 锁和同步

每个对象都有同其关联的一个锁。Java 语言不提供执行分开的锁定和解锁操作的方法;相反,它们由高层构造隐式地执行,这些构造总是进行排列以正确地将这样的操作配对。(然而 Java 虚拟机提出分开的 monitorenter 和 monitorexit 指令,实现锁定和解锁操作。)

Synchronized 方法计算对一个对象的引用;然后它试图对该对象执行锁定操作,并且直到锁定操作成功结束才进一步执行。(锁定操作可能被延迟,因为关于锁的规则可能阻止主存储器的加入,直到其他一些线程准备执行一个或更多解锁操作。)在锁定操作被执行后,执行 synchronized 语句的主体。如果主体的执行结束,正常地或突然地,自动对同一个锁执行解锁操作。

当 synchronized 方法被调用时,它自动执行锁定操作;其主体直到锁定操作已成功结束后才执行。如果该方法是实例方法,它锁定同实例相关联的锁,该实例是该方法的实例(即,在方法主体执行中被称为 this 的对象)。如果方法是 static,它锁定同 Class 对象相关联的锁,该对象代表定义方法的类。如果方法主体的执行结束,正常地或突然地,自动对同一个锁执行解锁操作。

最好的措施是,如果变量由一个线程赋值并由另一个线程使用或赋值,则对该变量的所有访问,应封闭在 synchronized 方法或 synchronized 语句中。

## 8.14 等待集和通知

每个对象,除具有关联的锁外,还有相关联的等待集,即一组线程。当对象首次创建时,其等待集为空。

等待集由类 Object 的 wait,notify,和 notifyAll 方法使用。这些方法也同线程的调度机制交互作用。

仅当当前线程(称它为 T)已锁定了对象的锁时,才应为该对象调用 wait 方法。假定线程 T 事实上已执行了尚未同解锁操作匹配的 N 次锁定操作。于是 wait 方法将当前线程加到对象的等待集中,出于线程调度的目的禁止当前线程,并执行 N 次解锁操作以释放该锁。线程 T 将处于休眠,直到三种情况之一发生:

- 其他一些线程对该对象调用了 notify 方法,而且线程 T 碰巧为作为要通知的,任意选取的线程。

- 其他一些线程对该对象调用了 notifyall 方法。
- 如果线程 T 对 wait 方法的调用指定了超时间隔,而实际时间的指定数量已过。

则线程 T 从等待集中删除,并对线程调度重新生效。然后它再次锁定对象(可能涉及同其他线程的以通常方式进行的竞争);一旦它获得了对锁的控制,它执行 N-1 次附加的锁定操作,然后从 wait 方法的调用返回。这样,从 wait 方法返回时,对象的锁的状态同调用 wait 方法时完全相同。

仅在当前线程已锁定了对象的锁,或者将要抛出 IllegalMonitorStateException 时,才应为对象调用 notify 方法。如果对象的等待集不为空,则从等待集中删除一些任意选择的线程,并对线程调度重新生效。(当然,直到当前线程释放了对象的锁,该线程才能进行。)

仅在当前线程已锁定了对象的锁,或者将要抛出 IllegalMonitorStateException 时,才应为对象调用 notifyall 方法。该对象等待集中的每个线程被从等待集中删除,并对线程调度重新生效。(当然,直到当前线程释放了对象的锁,那些线程才能进行。)

## 第九章 优 化

本章描述在 Java 虚拟机的 Sun 版本中实现的优化。在此优化中，编译的 Java 虚拟机代码在运行期修改以获得更好的性能。

优化采用伪指令集的形式。这些是通常 Java 虚拟机指令的变体，利用在运行期获知的信息来做比源指令更少的工作。伪指令通过其助记符中的后缀来区分。

理解这些伪指令不是 Java 虚拟机规范或指令集的一部分是很重要的。它们在 Java 虚拟机实现外不可见。然而，在 Java 虚拟机实现内部，它们被证明是有效的优化。

本章论述的技术由美国专利 5 367 685 覆盖。

### 9.1 通过重写动态链接

以 Java 虚拟机为目标的编译器必须只发出第六章“Java 虚拟机指令集”中所记述的指令集的指令。

新指令利用相关联的正常指令第一次执行时完成的装载和链接工作。

对重写的指令，指令的每个实例在其第一次执行时被替换为<sub>\_quick</sub> 伪指令。该指令实例随后执行的总是<sub>\_quick</sub> 变体。大多数有<sub>\_quick</sub> 变体的指令只具有一个替换版本，虽然一些指令有几个替换版本。

在任何情况下，带<sub>\_quick</sub> 变体的指令都引用常数池，这是有相当代价的操作。<sub>\_quick</sub> 伪指令通过开发这样一种事实来节省时间，指令第一次引用常数池时，必须动态地解析常数池表项，该指令的随后的调用必须引用同一对象，而且不需要再次解析该表项。重写过程如下：

1. 解析常数池中指定项。
2. 如果常数池中的项不能被解析则抛出异常。
3. 用<sub>\_quick</sub> 伪指令和所需的任何新操作数重写指令。每条 putstatic, getstatic, putfield, 和 getfield 指令都有两个<sub>\_quick</sub> 版本，其选择依赖于正被操作的域的类型。
4. 执行新的<sub>\_quick</sub> 的指令。

这同源指令的定义相同，除了指令用其<sub>\_quick</sub> 变体重写它自身的附加步骤。<sub>\_quick</sub> 伪指令的操作数必须适合于为源指令操作数分配的空间中。

指令的<sub>\_quick</sub> 变体可以假定常数池中的项已被解析，而且这种解析不产生任何错误。它简单地在解析的项上执行计划的操作。这样为随后的伪指令调用节省了大量的时间。

### 9.2 <sub>\_quick</sub> 伪指令

本章其余部分指定由 Sun 的 Java 虚拟机实现所使用的<sub>\_quick</sub> 伪指令。虽然以同通常的 Java 虚拟机指令相同的格式记载，但<sub>\_quick</sub> 伪指令不是 Java 虚拟机规范的一部分，不出现在 Class 类中。它们通常是不可见的实现细节，因而像操作码选择这样的决定由实现者作出。

然而,该规则有例外。某些像调试程序和刚好及时(JIT)代码生成器这样的工具,可能需要知道有关`_quick`伪指令的细节,这样它们能对已执行的代码操作。Java虚拟机的实现可以使用Sun的`_quick`伪指令相似但不同的技术,或者可以使用不同于Sun的实现的操作码数目。各工具假定Sun的`_quick`伪指令的细节可能对这些不同的实现不起作用。

正在为调试程序和JIT代码生成器开发API。这些API可以提供隐藏内部伪指令细节的方法,以便可以编写不依赖于内部实现细节的工具。然而,到本书编写时为止,这些API尚未建立,所以我们同时记述操作码值,以及Sun的`_quick`指令的其他细节。工具可以假定从Sun的派生出来的Java虚拟机的实现,或者被编写得同Sun的实现兼容的虚拟机实现,将遵守下面给出的规范。

有关调试程序和JIT代码生成器API的更多信息请同`Jvm@java.sun.com`联系。

---

## **anewarray\_quick**

---

**操作** 创建新的 reference 数组。

**格式**

anewarray_quick
indexbyte1
indexbyte2

**形式** anewarray\_quick=222(0xde)

**栈**  
...,count⇒  
...,arrayref

**描述** count 必须是 int 类型。它从操作数栈弹出。count 代表要创建的数组的成分数目。无符号的 indexbyte1 和 indexbyte2 用于构造到当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1<=8)|indexbyte2。常数池索引处的项必须已被成功解析,并且必须是类或接口类型。该类型的长度为 count 的数组,从垃圾回收堆中分配,对此新数组对象的 reference 类型的 arrayref 被压入操作数栈。新数组的所有成分被初始化为 null,即引用类型的缺省值(§ 2.5.1)。

**运行期异常** 如果 count 小于零,anewarray\_quick 指令抛出 NegativeArraySizeException。

**注解** 该指令的操作码原为 anewarray.anewarray 指令的操作数未被修改。anewarray\_quick 指令用于创建一组的对象引用数组。它还可以用于创建多维数组的第一维。

---

## **checkcast\_quick**

---

**操作** 检查对象是否为给定类型。

**格式**

checkcast_quick
indexbyte1
indexbyte2

**形式** checkcast\_quick=224(0xe0)

**栈** ...,objectref⇒

..., objectref

#### 描述

objectref 必须是 reference 类型。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1<<8)|indexbyte2。常数池索引处的对象必须已被成功解析,并且必须是类或接口类型。

如果 objectref 为 null,或者可被转换为解析的类、数组、或接口类型,则操作数栈无变化;否则,checkcast\_quick 指令抛出 ClassCastException。

下面的规则用于确定不为 null 的 objectref 是否可以转换为解析的类型:如果 S 是由 objectref 引用的对象的类,T 是解析的类、数组或接口类型,则 checkcast\_quick 确定 objectref 是否可以被转换为 T 类型,如下:

- 如果 S 为普通(非数组)类,则:
  - 如果 T 为类类型,则 S 必须是与 T 相同的类(§ 2.8.1),或者 S 必须是 T 的子类;
  - 如果 T 为接口类型,则 S 必须实现(§ 2.13)接口 T。
- 如果 S 为代表数组类型 SC[] 的类,即成分为 SC 类型的数组,则:
  - 如果 T 为类类型,则 T 必须是 Object(§ 2.4.6)。
  - 如果 T 为数组类型 TC[],即成分为 TC 的数组,则下述之一必为真:
    - TC 和 SC 为相同的基本类型(§ 2.4.1)。
    - TC 和 SC 是引用类型(§ 2.4.5),而且 SC 类型可按这些运行期规则转换为 TC。

S 不能是接口类型,因为不存在接口的实例,只有类和数组的实例。

#### 运行期异常

如果 objectref 不能转换为解析的类的类型,checkcast\_quick 指令抛出 ClassCastException。

#### 注解

该指令的操作码原为 checkcast. checkcast 指令的操作数未被修改。

checkcast\_quick 指令同 instanceof\_quick 指令非常相似。其不同在于对 null 的处理,当其检测失败时的行为(checkcast\_quick 抛出异常,instanceof\_quick 压入结果代码),以及对操作数栈的影响。

---

### getfield\_quick

---

#### 操作

从对象获取域。

#### 格式

getfield_quick
offset
⟨unused⟩

#### 形式

getfield\_quick=206(0xce)

<b>栈</b>	<code>... ,objectref⇒ ... ,value</code>
<b>描述</b>	objectref 必须是 reference 类型, 它从操作数栈弹出。一个字的域的值被获取并压入操作数栈。该域在由 objectref 引用的类实例中 offset 处。
<b>运行期异常</b>	如果 objectref 为 null, 则 getfield_quick 指令抛出NullPointerException。
<b>注解</b>	<p>该指令的操作码原为 getfield, 动态确定对域的操作, 以具有对 255 个字或更少的类实例数据的偏移量, 并且有一个字的宽度。</p> <p>当由 getfield 指令引用的常数池表项被解析时, 产生它引用的域的偏移量。该偏移量替换源 getfield 指令的第一个操作数字节。getfield 的第二个操作数字节未被 getfield_quick 使用。</p>

---

### getfield\_quick\_w

---

<b>操作</b>	从对象获取域。			
<b>格式</b>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>getfield_quick_w</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	getfield_quick_w	indexbyte1	indexbyte2
getfield_quick_w				
indexbyte1				
indexbyte2				
<b>形式</b>	<code>getfield_quick_W=227(Oxe3)</code>			
<b>栈</b>	<code>... ,objectref⇒ ... ,value</code> 或 <code>... ,objectref⇒ ... ,value.word1,value.word2</code>			
<b>描述</b>	objectref 必须是 reference 类型, 它从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引为 $(\text{indexbyte1} \ll 8)   \text{indexbyte2}$ 。索引处的常数池项必须为 CONSTANT_Fieldref(§ 4.4.2), 它必须已被解析并且不能是类(static)域。域偏移量必须已被存储在常数池中。由 objectref 引用的类实例中该偏移量处的值被获取并压入操作数栈。			

**运行期异常** 如果 objectref 为 null, getfield\_quick\_w 指令抛出 NullPointerException。

**注解** 该指令的操作码原为 getfield, 动态确定对域的操作, 以具有对多于 255 个字的类实例数据的偏移量。

getfield 指令的操作数未被修改。因为 getfield\_quick\_w 指令对一个和两个字宽的域都可进行操作, 它需要知道域偏移量和域类型。因为源 getfield 指令需要 16 位索引, 域偏移量可能为 16 位宽。因为指令中无足够空间来同时存储 16 位偏移量和域类型, getfield\_quick\_w 保留其源操作数并使用它们索引到常数池, 其中偏移量和域类型在解析的表项中得到。

---

### **getfield2\_quick**

---

**操作** 从对象获取 long 或 double 域。

**格式**

getfield2_quick
offset
⟨unused⟩

**形式** getfield2\_quick = 208(Oxd0)

**栈**

..., objectref ⇒  
..., value.word1, value.word2

**描述** objectref 必须是 reference 类型, 它从操作数栈弹出。两个字的域的值被获取并压入操作数栈, 该域在由 objectref 所引用的类实例的 offset 处。

**运行期异常** 如果 objectref 为 null, getfield2\_quick 指令抛出 NullPointerException。

**注解** 该指令的操作码原为 getfield, 动态确定对域的操作, 以具有对于有 255 个字或更少的类实例数据的偏移量, 并具有两个字的宽度。

当由 getfield 指令引用的常数池表项被解析时, 产生它引用域的偏移量。该偏移量替换源 getfield 指令的第一个操作数。getfield 的第二个操作数未被 getfield2\_quick 使用。

---

### **getstatic quick**

---

**操作** 从类取得静态域。

<b>格式</b>	<table border="1"> <tr><td>getstatic_quick</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	getstatic_quick	indexbyte1	indexbyte2
getstatic_quick				
indexbyte1				
indexbyte2				
<b>形式</b>	<code>getstatic_quick = 210(Oxd2)</code>			
<b>栈</b>	$\dots, \Rightarrow$ $\dots, \text{value}$			
<b>描述</b>	无符号的 indexbyte1 和 indexbyte2 用于构造对当前类常数池的索引 (§ 3.6)，其中索引值为 $(\text{indexbyte1} \ll 8)   \text{indexbyte2}$ 。索引处的常数池项必须为 CONSTANT_Fieldref (§ 4.4.2)，它必须已被解析，并且必须是一个字宽的类(static)域。该类域的值被获取并压入操作数栈。			
<b>注解</b>	该指令的操作码原为 getstatic，动态确定对 static 域的操作，以具有一个字宽。getstatic 指令的操作数未被修改。没有同 getfield_quick 指令等价之处，对一个字的 static 域，作为指令操作数存储类偏移量。			

---

### **getstatic2\_quick**

---

<b>操作</b>	从类中取得静态域。			
<b>格式</b>	<table border="1"> <tr><td>getstatic2_quick</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	getstatic2_quick	indexbyte1	indexbyte2
getstatic2_quick				
indexbyte1				
indexbyte2				
<b>形式</b>	<code>getstatic2_quick = 212(Oxd4)</code>			
<b>栈</b>	$\dots, \Rightarrow$ $\dots, \text{value. word1}, \text{value. word2}$			
<b>描述</b>	无符号的 indexbyte1 和 indexbyte2 用于构造对当前类 (§ 3.6) 的常数池的索引，其中索引值为 $(\text{indexbyte1} \ll 8)   \text{indexbyte2}$ 。索引处的常数池项必须是 CONSTANT_Fieldref (§ 4.4.2)，它必须已被解析并且必须是两个字宽的类(static)域。该类域的值被获取并压入操作数栈。 常数池项是对类的静态域的域引用。域类型必须为 long 或 double。域的			

值被压入操作数栈。

## 注解

该指令的操作码原为 getstatic, 对类域的操作动态确定为两个字宽。getstatic 指令的操作数未被修改。没有同 getfield2\_quick 指令等价之处, 对两个字的 static 域, 作为指令操作数存储类偏移量。

---

### instanceof\_quick

---

#### 操作

确定对象是否为给定类型。

#### 格式

instanceof_quick
indexbyte1
indexbyte2

#### 形式

instanceof\_quick = 225(0xe1)

#### 栈

... , objectref  $\Rightarrow$   
... , result

#### 描述

objectref 必须是 reference 类型, 它从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引值为(indexbyte1  $\ll 8$ ) | indexbyte2。常数池索引处的类必须已被成功解析, 并且可能是类, 数组或接口。

如果 objectref 为 null, 而且是解析的类, 数组, 或接口的实例, instanceof\_quick 指令将 int 类型的 result 1 作为整数压入操作数栈。否则, 它压入 int 类型的 result 0。下面的规则用于确定不为 null 的 objectref 是否为解析类型的实例: 如果 S 为由 objectref 引用的对象的类, 而 T 是解析的类、类组或实例类型, instanceof\_quick 确定 objectref 是否为 T 的实例如下:

- 如果 S 为普通(非数组)类, 则:
- 如果 T 为类类型, 则 S 必须为与 T 相同的类(§ 2.8.1), 或 T 的子类。
- 如果 T 为接口类型, 则 S 必须实现(§ 2.13)接口 T。
- 如果 S 是代表数组类型 SC[] 的类, 即成分为 SC 类型的数组, 则:
- 如果 T 为类类型, 则 T 必须是 Object(§ 2.4.6)。
- 如果 T 为数组类型 TC[], 即成分为 TC 类型的数组, 则下述之一必为真:
  - TC 和 SC 为相同的基本类型(§ 2.4.1)。
  - TC 和 SC 为引用类型(§ 2.4.5), 而 SC 类型可按这些运行期规则转

换为 TC。

S 不能是接口类型,因为不存在接口的实例,只有类和数组的实例。

## 注解

该指令的操作码原为 instanceof。instanceof 指令的操作数未被修改。

### invokeinterface\_quick

#### 操作

调用接口方法。

#### 格式

invokeinterface_quick
idbyte1
idbyte2
nargs
guess

#### 形式

invokeinterface-quick=218(Oxda)

#### 栈

... ,objectref ,[arg1 ,[arg2... ]] ⇒  
...

#### 描述

无符号的 idbyte1 和 idbyte2 用于构造所需方法的名称和描述符(§ 4.3.3)的标识符,其中标识符的值为  $(\text{idbyte1} \ll 8) | \text{idbyte2}$ 。

nargs 操作数为无符号字节,不能为零。objectref 必须为 reference 类型,并且必须在操作数栈上由 nargs-1 个字的参数跟随。

objectref 类型的类的方法表被确定。如果 objectref 为数组类型,则使用类 Object 的方法表。

无符号的 guess 用于索引到方法表。如果索引 guess 处有一个方法,并且如果其标识符同构造的标识符相同,则该方法被选择。否则,在方法表中查找一个方法,其标识符同构造的标识符相同。如果找到一个,则 guess 的当前值用该索引重写。

查找的结果是方法表表项,它包括对接口方法代码和方法的修饰符信息的直接引用(见表 4.4“方法访问和修饰符标志”)。方法表表项必须是 public 方法的表项。

如果方法为 synchronized,则获得同 objectref 相关联的监视器。

如果方法不是 native、nargs-1 个参数字和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架,而 objectref 和参数字成为其第一个 nargs 局部变量的值,其中 objectref 在局部变量 0,arg1 在局部变量 1,等等。新框架成为当前的,Java 虚拟机 pc 被设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法为 native,而且实现它的依赖于平台的代码尚未装载并链接到 Java 虚拟机,它将被完成。nargs—1个参数和 objectref 从操作数栈弹出;实现该方法的代码被以依赖于实现的方式调用。

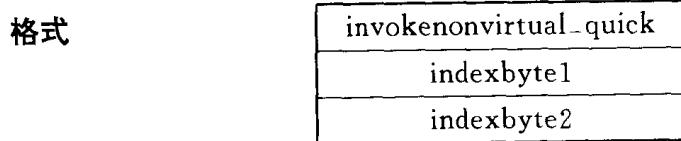
链接	如果在 objectref 类中不能找到同解析的名称和描述符匹配的方法,invokeinterface_quick 抛出 IncompatibleClassChangeError。 否则,如果选取的方法是类(static)方法,invokeinterface_quick 指令抛出 IncompatibleClassChangeError。否则,如果选取的方法不为 public,invokeinterface_quick 指令抛出 IllegalAccessException。否则,如果选取的方法是 abstract,invokeinterface_quick 抛出 AbstractMethodError。 否则,如果选取的方法为 native,而且实现该方法的代码不能被装载或链接,invokeinterface_quick 抛出 UnsatisfiedLinkError。
运行期异常	否则,如果 objectref 为 null,invokeinterface_quick 指令抛出 NullPointerException。
注解	该指令的操作码原为 invokeinterface.guess 的初始值为 0,操作数栈由 invokeinterface 提供。在 invokeinterface_quick 指令中被比较并存储的标识符,将方法名称和描述符编码为可以快速比较的 16 位数量。编码的细节是特定于实现的,正被调用方法的标识符字节, indexbyte1 和 indexbyte2, 替换原常数池索引字节。标识符可在每个方法都被装载或在运行时计算。nargs 操作数的值未被修改。

---

#### **invokenonvirtual\_quick**

---

**操作** 调用实例初始化方法或私有方法,基于编译期类型进行调度。



**形式** invokenonvirtual\_quick=215(0xd7)

**栈** ... , objectref, [arg1, [arg2...]] ⇒  
...

**描述** 无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为 (indexbyte1 << 8) | indexbyte2。索引处的常数池项

必须为 CONSTANT\_Methodref(§ 4.4.2), 它必须已被成功解析。代表解析的方法的常数池表项包括对该方法代码, 必须大于零的无符号字节 args, 和方法的修饰符信息的直接引用(见表4.4, “方法访问和修饰符标志”)。

如果方法为 synchronized, 则获得同 objectref 相关联的监视器。

如果方法不是 native, nargs-1个参数字节和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架, 而 objectref 和参数字节成为其第一个 nargs 局部变量的值, 其中 objectref 在局部变量0, arg1在局部变量1, 等等。然后新的栈框架成为当前的, 而且 Java 虚拟机 pc 被设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法为 native, nargs-1个参数字节和 objectref 从操作数栈弹出; 实现该方法的代码以依赖于平台的方式调用。

#### 运行期异常

如果 objectref 为 null, invokeonvirtual\_quick 指令抛出 NullPointerException。

#### 注解

该指令的操作码原为 invokespecial, 它调用的方法被动态确定为实例初始化方法<init>或 private 方法。invokeespecial 指令的操作数未被修改。

invokeonvirtual\_quick 和 invokevirtual\_quick\_w 指令间的不同在于 invokevirtual\_quick\_w 基于对象的实际(运行期)类型调用方法。invokeonvirtual\_quick 指令基于对象的编译期类型调用实例初始化方法或 private 方法。

---

### invokesuper\_quick

---

#### 操作

调用超类方法, 基于编译期类型进行调度。

#### 格式

invokesuper_quick
indexbyte1
indexbyte2

#### 形式

invokesuper\_quick=216(0xd8)

#### 栈

..., objectref, [arg1, [arg2...]] ⇒  
...

#### 描述

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的超类的方法表的索引, 其中索引值为(indexbyte1<<8)|indexbyte2。特定的方法

表项包括对方法代码,必须大于零的无符号字节 nargs,和方法的修饰符信息的直接引用(见表4.4“方法访问和修饰符标志”)。

如果方法为 synchronized,则获得同 objectref 关联的监视器。

如果方法不是 native,则 nargs-1个参数字节和 objectref 从操作数栈弹出。为正调用的访问创建新的栈框架,objectref 和参数字节成为其第一个 nargs 局部变量的值,其中 objectref 在局部变量0,arg1在局部变量1,等等。然后新的栈框架成为当前的,Java 虚拟机 pc 设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法为 native,则 nargs-1个参数字节和 objectref 从操作数栈弹出;实现该方法的代码被以依赖于实现的方式调用。

#### 运行期异常

如果 objectref 为 null,invokesuper\_quick 指令抛出NullPointerException。

#### 注解

该指令的操作码原来为 invokespecial,它调用的方法被动态确定为当前对象的超类中的方法。invokespecial 指令的操作数未被修改。

invokesuper\_quick 和 invokevirtual\_quick\_w 指令的区别在于 invokevirtual\_quick\_w 基于对象的类调用方法。invokesuper\_quick 指令用于调用当前类的超类中的方法。

invokesuper\_quick 指令在 Sun 的 JDK1.0.2 版中引入,以弥补 Java 虚拟机早期版本中的一个错误。在此版本之前,invokespecial 指令(命名为 invokenonvirtual)总是被转换为 invokenonvirtual\_quick 指令。

---

### invokestatic\_quick

---

#### 操作

调用类(静态)方法。

#### 格式

invokestatic_quick
indexbyte1
indexbyte2

#### 形式

invokestatic\_quick=217(Oxd9)

#### 栈

..., [arg1, [arg2...]] ⇒  
...

#### 描述

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1<<8)|indexbyte2。索引处的常数池项

必须为 CONSTANT\_Methodref(§ 4.4.2), 它必须已被成功解析。

代表解析的方法的常数池表项包括对方法代码的直接引用, 可以为零的无符号字节 nargs 和方法的修饰符信息(见表4.4“方法访问和修饰符标志”)。

如果方法为 synchronized, 则获得同当前类相关联的监视器。

如果方法不为 native, nargs 个参数字从操作数栈弹出。为正调用的方法创建新的栈框架, 参数字成为其第一个 nargs 局部变量的值, 其中 arg1 在局部变量 0, arg2 在局部变量 1, 等等。然后新的栈框架成为当前的, 并且 Java 虚拟机 pc 被设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法为 native, nargs 个参数字从操作数栈弹出; 实现该方法的代码被以依赖于实现的方式调用。

## 注解

该指令的操作码原来为 invokestatic, invokestatic 指令的操作数未被修改。

### **invokevirtual\_quick**

#### 操作

调用实例方法; 基于类进行调度。

#### 格式

invokevirtual_quick
index
nargs

#### 形式

invokevirtual\_quick = 214(Oxd6)

#### 栈

..., objectref, [atg1, [arg2...]] ⇒  
...

#### 描述

objectref 必须为 reference 类型, 并且必须引用类实例。index 操作数是无符号字节, 而 nargs 操作数是不能为零的无符号字节。index 是对 objectref 类型的类的方法表的索引。索引处的表表项包括方法的代码和其修饰符信息(见表4.4, “方法访问和修饰符标志”)。

如果方法为 synchronized, 则获得同 objectref 相关联的监视器。

如果方法不为 native, nargs-1 个参数字和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架, 并且 objectref 和参数字成为其第一个 nargs 局部变量的值, 其中 objectref 在局部变量 0, arg1 在局部变量 1, 等等。然后新的栈框架成为当前的, Java 虚拟机 pc 被设置为要调用方法的

第一条指令的操作码。从该方法的第一条指令继续执行。  
如果方法为 native，并且实现它的依赖于平台的代码尚未装载并链接到 Java 虚拟器，它被完成。nargs – 1 个参数字和 objectref 从操作数域弹出；实现方法的代码被以依赖于实现的方式调用。

#### 链接异常

如果指定的方法为 native，并且实现该方法的代码不能被装载并链接，invokevirtual\_quick 指令抛出 UnsatisfiedLinkError。

#### 运行期异常

否则，如果 objectref 为 null，invokevirtual\_quick 指令抛出 NullPointerException。

#### 注解

该指令的操作码原为 invokevirtual，具有不引用 java.lang.Object 的实例的 objectref，以及动态确定的操作数。操作数代表有 255 个或更少的方法表索引的方法。当由 invokevirtual 指令引用的常数池表项被解析时，产生时它所引用的方法的一个字节的索引。该索引替换源 invokevirtual 指令的第一个操作数字节。invokevirtual 指令的第二个操作数字节由 nargs，即方法所期望的参数字数目所替换。

相反，若 invokevirtual 指令引用 java.lang.Object 的实例，并具有代表 255 个或更少的常数池索引的操作数，则它将被转换为 invokevirtual-object\_quick 指令。带有代表大于 255 的常数池索引的操作数的 invokevirtual 指令，将被转换为 invokevirtual\_quick\_w 指令。

---

### invokevirtual\_quick\_w

---

#### 操作

调用实例方法，基于类(宽索引)调度。

#### 格式

invokevirtual_quick_w
indexbyte1
indexbyte2

#### 形式

invokevirtual\_quick\_w = 226(0xe2)

#### 栈

... , objectref , [arg1 , [arg2...]] =>  
...

#### 描述

无符号的 indexbyte1 和 indexbyte2 用于创建对当前类(§ 3.6)的常数池的索引，其中索引为(indexbyte1<<8)|indexbyte2。索引处的常数池项必须为 CONSTANT\_Methodref(§ 4.4.2)，它必须已被成功解析。代表解

析的方法的常数池表项包括对被解析类的方法表的无符号索引,以及不能为零的无符号字节 nargs。

objectref 必须为 reference 类型。index 作为对 objectref 类型的类的方法表的索引使用。如果 objectref 为数组类型,则使用类 Object 的方法表。索引处的表项包括方法的代码和其修饰符信息(见表4-4“方法访问和修饰符标志”)。

如果方法为 synchronized,则获得同 objectref 相关联的监视器。

如果方法不是 native,nargs-1个参数字节和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架,并且 objectref 和参数字成为其第一个 nargs 局部变量的值,其中 objectref 在局部变量0,arg1在局部变量1,等等。然后新的栈框架成为当前的,并且 Java 虚拟机 pc 被设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。

如果方法是 native,而且实现它的依赖于平台的代码尚未装载并链接到 Java 虚拟机,它被完成,nargs-1个参数字节和 objectref 从操作数栈弹出;实现该方法的代码被以依赖于实现的方式调用。

#### 链接异常

如果指定的方法为 native,并且实现该方法的代码不能被装载或链接,invokevirtual\_quick\_w 指令抛出 UnsatisfiedLinkError。

#### 运行期异常

否则,如果 objectref 为 null,invokevirtual\_quick\_w 指令抛出 NullPointerException。

#### 注解

该指令的操作码原为 invokevirtual,具有动态确定的操作数,操作数代表具有大于 255 的方法表索引的方法。invokevirtual 指令的操作数未被修改。

invokevirtual\_quick 和 invokevirtualobject\_quick 指令仅支持对 objectref 的方法表的一个字节的偏移量。invokevirtual\_quick\_w 可用于不能用 invokevirtual\_quick 表示的方法调用。

---

### invokevirtualobject\_quick

---

#### 操作

调用 java.lang.Object 类的实例方法。

#### 格式

invokevirtualobject_quick
index
nargs

#### 形式

invokevirtualobject\_quick=219(Oxdb)

栈	<pre>...,objectref,[arg1,arg2...]]⇒ ... </pre>
描述	<p>objectref 必须为 reference 类型。index 操作数为无符号字节,而 nargs 操作数为不能是零的无符号字节。index 是对 objectref 类型的类的方法表的索引。如果 objectref 为数组类型,则使用类 Object 的方法表。该索引处的表项包括方法的代码及其修饰符信息(见表4-4“方法访问和修饰符标志”)。</p> <p>如果方法是 synchronized,则获得同 objectref 相关联的监视器。</p> <p>如果方法不是 native,则 nargs-1个参数字和 objectref 从操作数栈弹出。为正调用的方法创建新的栈框架,objectref 和参数字成为其第一个 nargs 局部变量的值,其中 objectref 在局部变量0,arg1在局部变量1,等等,然后新的栈框架成为当前的,并且 Java 虚拟机 pc 被设置为要调用方法的第一条指令的操作码。从该方法的第一条指令继续执行。</p> <p>如果方法是 native,并且实现它的依赖于平台的代码尚未装载并链接到 Java 虚拟机,它被完成。margs-1个参数字和 objectref 从操作数栈弹出;实现该方法的代码被以依赖于实现的方式调用。</p>
链接异常	<p>如果指定的方法为 native,并且实现该方法的代码不能装载或链接,invokevirtual_quick 抛出 UnsatisfiedLinkError。</p>
运行期异常	<p>否则,如果 objectref 为 null,invokevirtualobject_quick 指令抛出 NullPointerException。</p>
注解	<p>该指令的操作码原为 invokevirtual,它引用 java.lang.Object 类的方法,该方法是动态确定的具有255或更小的方法表索引。</p> <p>invokevirtualobject_quick 指令特别适于数组。</p> <p>当由 invokevirtual 指令引用的常数池表现被解析时,产生它所引用的方法的一个字节的索引。该索引替换源 invokevirtual 指令的第一个操作数字节。invokevirtual 指令的第二个操作数字节由 nargs,即方法所期望的参数字的数目所替换。</p> <p>invokevirtualobject_quick 仅只持对 objectref 的方法表的一个字节的索引。具有大量方法的对象,可能不能使其所有的方法由_quick 变体引用。如果效率较低,则拒绝将 invokevirtual 指令的实例转换为 invokevirtualobject_quick,这总是对的。</p> <p>相反,若 invokevirtual 指令不引用 java.lang.Object 的实例,并具有代表 255或更小的常数池索引的操作数,则它将被转换为 invokevirtual-quick 指令。任何具有代表大于255的常数池索引的操作数的 invokevirtual 指</p>

令,将被转换为 invokevirtual\_quick\_w 指令。

---

### ldc\_quick

---

**操作** 压入常数池中的项。

**格式**

ldc_quick
index

**形式** ldc\_quick=203(0xcb)

**栈**  
... ⇒  
...,item

**描述** index 是无符号字节,必须是对当前类(§ 3.6)的常数池的有效索引。index 处的常数池 item 必须已被解析,并且必须为一个字宽。item 从常数池获取,并压入操作数栈。

**注解** 该指令的操作码原为 ldc。ldc 指令的操作数未被修改。

---

### ldc\_w\_quick

---

**操作** 压入常数池中的项(宽索引)。

**格式**

ldc_w_quick
indexbyte1
indexbyte2

**形式** ldc\_w\_quick=204(0xcc)

**栈**  
... ⇒  
...,item

**描述** 无符号的 indexbyte1 和 indexbyte2 被装配成对当前类(§ 3.6)的常数池的无符号16位索引,其中索引为(indexbyte1<<8)|indexbyte2。该索引必须是对当前类的常数池的有效索引。索引处的常数池 item 必须已被解析,并且必须为一个字宽。item 从常数池获取并压入操作数栈。

**注解**

该指令的操作码原为 ldc\_w。ldc\_w 指令的操作数未被修改。

ldc\_w\_quick 指令同 ldc\_quick 指令相同，除了它的更宽的常数池索引。

**ldc2\_w\_quick****操作**

压入常数池中的 long 或 double(宽索引)。

**格式**

ldc2_w_quick
indexbyte1
indexbyte2

**形式**

ldc2\_w\_quick = 205 (0xcd)

**栈**

... ⇒  
..., item.word1, item.word2

**描述**

无符号的 indexbyte1 和 indexbyte2 被装配为对当前类(§ 3.6)的常数池的无符号的 16 位索引，其中索引值为(indexbyte1<8) | indexbyte2。该索引必须是对当前类的常数池的有效索引。索引处的(64 位)常数池 constant 必须已被解析并且必须为两个字宽。

**注解**

该指令的操作码原为 ldc2\_w。源 ldc2\_w 指令的操作数未被修改。

仅存在该指令的宽索引版本；不存在以一个单字节的索引压入两个字的常数的 ldc2\_quick。

**multianewarray\_quick****操作**

创建新的多维数组。

**格式**

multianewarray_quick
indexbyte1
indexbyte2
dimensions

**形式**

multianewarray\_quick = 223 (0xdf)

**栈**

..., count1, [count2, ...] ⇒  
..., arrayref

**描述**

dimensions 是必须大于等于1的无符号字节。它代表要创建的数组的维数。操作数栈必须包含 dimensions 字，它必须为 int 类型并且非负，每个代表要创建的数组中一个维的成分数目。count1是第一维中想要的长度，count2是第二维中的，等等。

所有的 count 值从操作数栈弹出。无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引，其中索引值为(indexbyte1<sub>8</sub>) | indexbyte2。结果表项必须已被解析到维数大于等于 dimensions 的数组类类型。

该数组类型的新多维数组从垃圾回收堆中分配。该数组的第一维的成分被初始化为第二维类型的子数组，等等。

最后一维中数组的成分被初始化为其类型(§ 2.5.1)的缺省初始值。对新数组的 reference 类型的 arrayref 被压入操作数栈。

**运行期异常**

如果操作数栈上的任意 dimensions 值小于零，则 multianewarray\_quick 抛出 NegativeArraySizeException。

**注解**

该指令的操作码原为 multianewarray。multianewarray 指令的操作数未被修改。

---

## new\_quick

---

**操作**

创建新对象。

**格式**

new_quick
indexbyte1
indexbyte2

**形式**

new\_quick = 221(Oxdd)

**栈**

... ⇒  
..., objectref

**描述**

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引，其中索引值为(indexbyte1<sub>8</sub>) | indexbyte2。该索引处的项必须

已被解析到类类型。该类的新实例被创建,该新对象的实例变量被初始化为其缺省初始值(§ 2.5.1)。objectref,即对实例的 reference,被压入操作数栈。

#### 注解

该指令的操作码原为 new。源 new 指令的操作数未被修改。

---

### **putfield\_quick**

---

#### 操作

设置对象中的域。

#### 格式

putfield_quick
offset
unused

#### 形式

putfield\_quick=207(0xcf)

#### 栈

...,objectref,value⇒

...

#### 描述

objectref 必须是 reference 类型,value 必须是适于该指定域的类型的值,它们从操作数栈弹出.value 被写入由 objectref 引用的类实例的 offset 处。

#### 运行期异常

如果 objectref 为 null,putfield\_quick 指令抛出NullPointerException。

#### 注解

该指令的操作码原为 putfield,它对域操作。该域被动态确定,具有对255个字或更少的类实例数据的偏移量,并具有一个字的宽度。

当由 putfield 指令引用的常数池表项被解析时,产生它所引用的域的偏移量。该偏移量替换源 putfield 指令的第一个操作数字节.putfield 的第二个操作数字节在 putfield\_quick 中未被使用。

---

### **putfield\_quick\_w**

---

#### 操作

设置对象中的域(宽索引)。

<b>格式</b>	<table border="1"> <tr><td>putfield_quick_w</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	putfield_quick_w	indexbyte1	indexbyte2
putfield_quick_w				
indexbyte1				
indexbyte2				
<b>形式</b>	<code>putfield_quick_w = 228(0xe4)</code>			
<b>栈</b>	$\dots, \text{objectref}, \text{value} \Rightarrow$ $\dots$ 或 $\dots, \text{objectref}, \text{value. word1}, \text{value. word2} \Rightarrow$ $\dots$			
<b>描述</b>	<p><code>objectref</code> 必须是 <code>reference</code> 类型, <code>value</code> 必须是适于指定域的类型的值, 它们从操作数栈弹出。无符号的 <code>indexbyte1</code> 和 <code>indexbyte2</code> 用于构造对当前类(§ 3.6)的常数池的索引, 其中索引为 <math>(\text{indexbyte1} \ll 8)   \text{indexbyte2}</math>。索引处的常数池项必须为 <code>CONSTANT_Fieldref</code>(§ 4.4.2), 它必须已被解析, 并且不能是类(<code>static</code>)域。<code>value</code> 被写入由 <code>objectref</code> 引用的类实例的 <code>offset</code> 处。</p>			
<b>运行期异常</b>	如果 <code>objectref</code> 为 <code>null</code> , <code>putfield_quick_w</code> 指令抛出 <code>NullPointerException</code> 。			
<b>注解</b>	<p>该指令的操作码原为 <code>putfield</code>, 它对域操作, 该域被动态确定, 具有对于多于 255 个字的类实例数据的偏移量。</p> <p><code>putfield</code> 指令的操作数未被修改。因为 <code>putfield_quick_w</code> 对一个字和两个字宽的域都可进行操作, 它需要知道域偏移量和该域的类型。因为源 <code>putfield</code> 指令需要 16 位索引, 域偏移量可为 16 位宽。因为指令中无足够空间同时保存 16 位偏移量和域类型, <code>putfield_quick_w</code> 保留其源操作数并使用它们索引到常数池, 其中偏移量和域类型可在解析的表项中获得。</p>			

---

## **putfield2\_quick**

---

<b>操作</b>	设置对象中的 <code>long</code> 或 <code>double</code> 。
-----------	--

<b>格式</b>	<table border="1"> <tr><td>putfield2_quick</td></tr> <tr><td>offset</td></tr> <tr><td>unused</td></tr> </table>	putfield2_quick	offset	unused
putfield2_quick				
offset				
unused				
<b>形式</b>	putfield2_quick = 209(Oxd1)			
<b>栈</b>	... , objectref, value. word1, value. word2 => ...			
<b>描述</b>	objectref 必须为 reference 类型, value 必须为适于指定的域的类型的值, 它们从操作数栈弹出。value 被写入由 objectref 引用的类实例的 offset 处。			
<b>运行期异常</b>	如果 objectref 为 null, putfield2_quick 指令抛出 NullPointerException。			
<b>注解</b>	<p>该指令的操作码原为 putfield, 它对域操作。该域被动态确定, 具有对于 255 个字或更少的类实例数据的偏移量, 并具有两个字的宽度。</p> <p>当由 objectref 指令引用的常数池表项被解析时, 产生它引用的域的偏移量。该偏移量替换源 putfield 指令的第一个操作数。第二个操作数在 putfield2_quick 中未被使用。</p>			

---

### **putstatic\_quick**

---

<b>操作</b>	设置类中的 static 域。			
<b>格式</b>	<table border="1"> <tr><td>putstatic_quick</td></tr> <tr><td>indexbyte1</td></tr> <tr><td>indexbyte2</td></tr> </table>	putstatic_quick	indexbyte1	indexbyte2
putstatic_quick				
indexbyte1				
indexbyte2				
<b>形式</b>	putstatic_quick = 211(Oxd3)			
<b>栈</b>	... , value => ...			
<b>描述</b>	无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池			

的索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。常数池项必须是对类(static)域的域引用,该引用必须已被成功解析到一个字宽的类型。value 必须为适于该类域的类型。value 从操作数栈弹出,并且该类域被设置为 value。

#### 注解

该指令操作码原为 putstatic,它对 static 域操作,该域被动态确定为一个字宽。putstatic 指令的操作数未被修改。同 putfield\_quick 指令没有相同之处,对一个字的 static 域,作为指令操作数存储类偏移量。

---

### **putstatic2\_quick**

---

#### 操作

设置类中的 static 域。

#### 格式

putstatic2_quick
indexbyte1
indexbyte2

#### 形式

putstatic2\_quick = 213(Oxd5)

#### 栈

..., value.word1, value.word2  $\Rightarrow$   
...

#### 描述

无符号的 indexbyte1 和 indexbyte2 用于构造对当前类(§ 3.6)的常数池的索引,其中索引值为(indexbyte1 $\ll$ 8) | indexbyte2。该常数池项必须是对类(static)域的域引用,该引用必须已被成功解析为两个字宽的类型。value 必须为适于该类域的类型。value 从操作数栈弹出,并且该类域被设置为 value。

#### 注解

该指令的操作码原为 putstatic,它对 static 域操作,该域被动态确定为两个字宽。putstatic 指令的操作数未被修改。同 putfield2\_quick 没有相同之处,对两个字的 static 域,作为指令操作数存储类偏移量。

## 第十章 操作码的操作码助记符

0 (0x00) .....	<i>nop</i>
1 (0x01) .....	<i>aconst_null</i>
2 (0x02) .....	<i>iconst_m1</i>
3 (0x03) .....	<i>iconst_0</i>
4 (0x04) .....	<i>iconst_1</i>
5 (0x05) .....	<i>iconst_2</i>
6 (0x06) .....	<i>iconst_3</i>
7 (0x07) .....	<i>iconst_4</i>
8 (0x08) .....	<i>iconst_5</i>
9 (0x09) .....	<i>lconst_0</i>
10 (0x0a) .....	<i>lconst_1</i>
11 (0x0b) .....	<i>fconst_0</i>
12 (0x0c) .....	<i>fconst_1</i>
13 (0x0d) .....	<i>fconst_2</i>
14 (0x0e) .....	<i>dconst_0</i>
15 (0x0f) .....	<i>dconst_1</i>
16 (0x10) .....	<i>bipush</i>
17 (0x11) .....	<i>sipush</i>
18 (0x12) .....	<i>ldc</i>
19 (0x13) .....	<i>ldc_w</i>
20 (0x14) .....	<i>ldc2_w</i>
21 (0x15) .....	<i>iload</i>
22 (0x16) .....	<i>lload</i>
23 (0x17) .....	<i>fload</i>
24 (0x18) .....	<i>dload</i>
25 (0x19) .....	<i>aload</i>
26 (0x1a) .....	<i>iload_0</i>
27 (0x1b) .....	<i>iload_1</i>
28 (0x1c) .....	<i>iload_2</i>
29 (0x1d) .....	<i>iload_3</i>
30 (0x1e) .....	<i>lload_0</i>
31 (0x1f) .....	<i>lload_1</i>
32 (0x20) .....	<i>lload_2</i>
33 (0x21) .....	<i>lload_3</i>
34 (0x22) .....	<i>fload_0</i>
35 (0x23) .....	<i>fload_1</i>
36 (0x24) .....	<i>fload_2</i>
37 (0x25) .....	<i>fload_3</i>

38 (0x26)	.....	<i>dload</i> .0
39 (0x27)	.....	<i>dload</i> .1
40 (0x28)	.....	<i>dload</i> .2
41 (0x29)	.....	<i>dload</i> .3
42 (0x2a)	.....	<i>aload</i> .0
43 (0x2b)	.....	<i>aload</i> .1
44 (0x2c)	.....	<i>aload</i> .2
45 (0x2d)	.....	<i>aload</i> .3
46 (0x2e)	.....	<i>iaload</i>
47 (0x2f)	.....	<i>laload</i>
48 (0x30)	.....	<i>faload</i>
49 (0x31)	.....	<i>daload</i>
50 (0x32)	.....	<i>aaload</i>
51 (0x33)	.....	<i>baload</i>
52 (0x34)	.....	<i>caload</i>
53 (0x35)	.....	<i>saload</i>
54 (0x36)	.....	<i>istore</i>
55 (0x37)	.....	<i>lstore</i>
56 (0x38)	.....	<i>fstore</i>
57 (0x39)	.....	<i>dstore</i>
58 (0x3a)	.....	<i>astore</i>
59 (0x3b)	.....	<i>istore</i> .0
60 (0x3c)	.....	<i>istore</i> .1
61 (0x3d)	.....	<i>istore</i> .2
62 (0x3e)	.....	<i>istore</i> .3
63 (0x3f)	.....	<i>lstore</i> .0
64 (0x40)	.....	<i>lstore</i> .1
65 (0x41)	.....	<i>lstore</i> .2
66 (0x42)	.....	<i>lstore</i> .3
67 (0x43)	.....	<i>fstore</i> .0
68 (0x44)	.....	<i>fstore</i> .1
69 (0x45)	.....	<i>fstore</i> .2
70 (0x46)	.....	<i>fstore</i> .3
71 (0x47)	.....	<i>dstore</i> .0
72 (0x48)	.....	<i>dstore</i> .1
73 (0x49)	.....	<i>dstore</i> .2
74 (0x4a)	.....	<i>dstore</i> .3
75 (0x4b)	.....	<i>astore</i> .0
76 (0x4c)	.....	<i>astore</i> .1
77 (0x4d)	.....	<i>astore</i> .2
78 (0x4e)	.....	<i>astore</i> .3
79 (0x4f)	.....	<i>iastore</i>
80 (0x50)	.....	<i>lastore</i>
81 (0x51)	.....	<i>fastore</i>

82 (0x52)	.....	<i>dastore</i>
83 (0x53)	.....	<i>aastore</i>
84 (0x54)	.....	<i>bastore</i>
85 (0x55)	.....	<i>castore</i>
86 (0x56)	.....	<i>sastore</i>
87 (0x57)	.....	<i>pop</i>
88 (0x58)	.....	<i>pop2</i>
89 (0x59)	.....	<i>dup</i>
90 (0x5a)	.....	<i>dup.x1</i>
91 (0x5b)	.....	<i>dup.x2</i>
92 (0x5c)	.....	<i>dup2</i>
93 (0x5d)	.....	<i>dup2.x1</i>
94 (0x5e)	.....	<i>dup2.x2</i>
95 (0x5f)	.....	<i>swap</i>
96 (0x60)	.....	<i>iadd</i>
97 (0x61)	.....	<i>ladd</i>
98 (0x62)	.....	<i>fadd</i>
99 (0x63)	.....	<i>dadd</i>
100 (0x64)	.....	<i>isub</i>
101 (0x65)	.....	<i>lsub</i>
102 (0x66)	.....	<i>fsub</i>
103 (0x67)	.....	<i>dsub</i>
104 (0x68)	.....	<i>imul</i>
105 (0x69)	.....	<i>lmul</i>
106 (0x6a)	.....	<i>fmul</i>
107 (0x6b)	.....	<i>dmul</i>
108 (0x6c)	.....	<i>idiv</i>
109 (0x6d)	.....	<i>ldiv</i>
100 (0x6e)	.....	<i>fdiv</i>
111 (0x6f)	.....	<i>ddiv</i>
112 (0x70)	.....	<i>irem</i>
113 (0x71)	.....	<i>lrem</i>
114 (0x72)	.....	<i>frem</i>
115 (0x73)	.....	<i>drem</i>
116 (0x74)	.....	<i>ineg</i>
117 (0x75)	.....	<i>lneg</i>
118 (0x76)	.....	<i>fneg</i>
119 (0x77)	.....	<i>dneg</i>
120 (0x78)	.....	<i>ishl</i>
121 (0x79)	.....	<i>lshl</i>
122 (0x7a)	.....	<i>ishr</i>
123 (0x7b)	.....	<i>lshr</i>
124 (0x7c)	.....	<i>iushr</i>
125 (0x7d)	.....	<i>lushr</i>

126 (0x7e)	.....	<i>iand</i>
127 (0x7f)	.....	<i>land</i>
128 (0x80)	.....	<i>ior</i>
129 (0x81)	.....	<i>lor</i>
130 (0x82)	.....	<i>ixor</i>
131 (0x83)	.....	<i>lxor</i>
132 (0x84)	.....	<i>iinc</i>
133 (0x85)	.....	<i>i2l</i>
134 (0x86)	.....	<i>i2f</i>
135 (0x87)	.....	<i>i2d</i>
136 (0x88)	.....	<i>l2i</i>
137 (0x89)	.....	<i>l2f</i>
138 (0x8a)	.....	<i>l2d</i>
139 (0x8b)	.....	<i>f2i</i>
140 (0x8c)	.....	<i>f2l</i>
141 (0x8d)	.....	<i>f2d</i>
142 (0x8e)	.....	<i>d2i</i>
143 (0x8f)	.....	<i>d2l</i>
144 (0x90)	.....	<i>d2f</i>
145 (0x91)	.....	<i>i2b</i>
146 (0x92)	.....	<i>i2c</i>
147 (0x93)	.....	<i>i2s</i>
148 (0x94)	.....	<i>lcmp</i>
149 (0x95)	.....	<i>fcmpl</i>
150 (0x96)	.....	<i>fcmpg</i>
151 (0x97)	.....	<i>dcmpl</i>
152 (0x98)	.....	<i>dcmpg</i>
153 (0x99)	.....	<i>ifeq</i>
154 (0x9a)	.....	<i>ifne</i>
155 (0x9b)	.....	<i>iflt</i>
156 (0x9c)	.....	<i>ifge</i>
157 (0x9d)	.....	<i>ifgt</i>
158 (0x9e)	.....	<i>ifle</i>
159 (0x9f)	.....	<i>if_icmpeq</i>
160 (0xa0)	.....	<i>if_icmpne</i>
161 (0xa1)	.....	<i>if_icmplt</i>
162 (0xa2)	.....	<i>if_icmpge</i>
163 (0xa3)	.....	<i>if_icmpgt</i>
164 (0xa4)	.....	<i>if_icmple</i>
165 (0xa5)	.....	<i>if_acmpeq</i>
166 (0xa6)	.....	<i>if_acmpne</i>
167 (0xa7)	.....	<i>goto</i>
168 (0xa8)	.....	<i>jsr</i>
169 (0xa9)	.....	<i>ret</i>

170 (0xaa)	tableswitch
171 (0xab)	lookupswitch
172 (0xac)	ireturn
173 (0xad)	lreturn
174 (0xae)	freturn
175 (0xaf)	dreturn
176 (0xb0)	areturn
177 (0xb1)	return
178 (0xb2)	getstatic
179 (0xb3)	putstatic
180 (0xb4)	getfield
181 (0xb5)	putfield
182 (0xb6)	invokevirtual
183 (0xb7)	invokespecial
184 (0xb8)	invokestatic
185 (0xb9)	invokeinterface
186 (0xba)	xxxunusedxxx
187 (0xbb)	new
188 (0xbc)	newarray
189 (0xbd)	anewarray
190 (0xbe)	arraylength
191 (0xbf)	athrow
192 (0xc0)	checkcast
193 (0xc1)	instanceof
194 (0xc2)	monitorenter
195 (0xc3)	monitorexit
196 (0xc4)	wide
197 (0xc5)	multianewarray
198 (0xc6)	ifnull
199 (0xc7)	ifnonnull
200 (0xc8)	goto.w
201 (0xc9)	jsr.w
<b>_quick 操作码</b>	
203 (0xcb)	ldc quick
204 (0xcc)	ldc.w quick
205 (0xcd)	ldc2.w quick
206 (0xce)	getfield.quick
207 (0xcf)	putfield.quick
208 (0xd0)	getfield2.quick
209 (0xd1)	putfield2.quick
210 (0xd2)	getstatic.quick
211 (0xd3)	putstatic.quick
212 (0xd4)	getstatic2.quick
213 (0xd5)	putstatic2.quick

214 (0xd6) .....	<i>invokevirtual.quick</i>
215 (0xd7) .....	<i>invokenonvirtual.quick</i>
216 (0xd8) .....	<i>invokesuper.quick</i>
217 (0xd9) .....	<i>invokestatic.quick</i>
218 (0xda) .....	<i>invokeinterface.quick</i>
219 (0xdb). invokevirtualobject.quick	
221 (0xdd) .....	<i>new.quick</i>
222 (0xde) .....	<i>anewarray.quick</i>
223 (0xdf) .....	<i>multianewarray.quick</i>
224 (0xe0) .....	<i>checkcast.quick</i>
225 (0xe1) .....	<i>instanceof.quick</i>
226 (0xe2) .....	<i>invokevirtual.quick.w</i>
227 (0xe3) .....	<i>getfield.quick.w</i>
228 (0xe4) .....	<i>putfield.quick.w</i>
保留操作码	
202 (0xca) .....	<i>breakpoint</i>
254 (0xfe) .....	<i>impdep1</i>
255 (0xff) .....	<i>impdep2</i>

## 无忧书库书籍版权申明

无忧书库提供的所有书籍、资料版权归原作者和出版商所有。

本站提供下载的书籍仅供个人学习、参考之用，任何集体、个人不得用于商业用途。

请您在下载书籍 24 小时之后删除书籍，购买正版书籍！

本站书籍如有侵犯您的版权，请与我们联系，我们将尽快删除。联系信箱：  
[pcbook@51soft.com](mailto:pcbook@51soft.com)。

无忧书库  
<http://pcbook.51soft.com>