Java 虚拟机规范

(Java SE 7 版)

原文版本．2011/07/28

译文版本．2011/11/13

作者：

Tim Lindholm、Frank Yellin

Gilad Bracha、Alex Buckley

译者：

周志明（icyfenix@gmail.com）

吴璞渊（wupuyuan@gmail.com）

冶秀刚（dennyy99@gmail.com）

译者序

从 1999 年 4 月出版的《Java 虚拟机规范（第二版）》至今，已经超过 12 年，虽然此规范在 JDK 5 发布的时候作了较大的更新，但却始终没有发布完整的规范。在今年 6 月 28 日，最新的《Java 虚拟机规范（Java SE 7 版）》终于完成并在 7 月份正式发布。对于想了解 Java 虚拟机的程序员来说，《Java 虚拟机规范》是必须阅读的，对于想深入了解 Java 语言细节的程序员，阅读《Java 虚拟机规范》也有极大好处，但是《Java 虚拟机规范》、《Java 语言规范》发布十余年，一直没有中文译本，这让中国不少对 Java 虚拟机感兴趣，但英语能力较弱的程序员都被拒之门外。

在 2011 年初，《Java 虚拟机规范（Java SE 7 版）》还是草稿状态时，我就开始关注这本书，并陆续对其中第 1、2、6、7 章进行了翻译，到 2011 年 9 月时完成了 200 余页的译稿。这时候又在国内著名 Java 社区 ItEye 中结识了另外两名译者吴璞渊和冶秀刚，我们在随后的两个多月的时间里共同完成了其余章节的翻译和校对。

《Java 虚拟机规范》并非某一款虚拟机实现的说明书，它是一份保证各个公司的 Java 虚拟机实现具备统一外部接口的契约文档，书中的概念和细节描述曾经与 Sun 的早期虚拟机的实现高度吻合，但是随着技术的发展，高性能虚拟机真正的细节实现方式已经渐渐与虚拟机规范所描述的内容产生了越来越大的差距。原作者也在书中不同地方反复强调过：虚拟机规范中所提及的“Java 虚拟机”皆为虚拟机的概念模型而非具体实现。实现只要保证与概念模型最终等效即可，而具体实现的方式无需受概念模型束缚。因此通过虚拟机规范去分析程序的执行语义问题（虚拟机会做什么）时，但分析程序的执行行为问题（虚拟机是怎样做的、性能如何）则意义不大，如需对具体虚拟机实现进行调优、性能分析等，我推荐在本书基础上继续阅读《Java Performance》和《Oracle JRockit The Definitive Guide》等书。

在翻译过程中，我们尽最大努力保证作品的准确性和可读性，力求在保证语义准确的前提下，尽可能使用通俗易懂的方式向给各位读者介绍 Java 虚拟机的约束与运作原理。为此目标，我们在专有技术名词、偏僻词中用括号保留了原文、专门在多处读者理解起来可能有困难的地方，添加了“译者注”加以解释。

囿于我们的水平和写作时间，书中难免存在不妥之处，大家如有任何意见或建议都欢迎通过以下邮件地址与我联系：icyfenix@gmail.com。本书的勘误与最新版本可以在以下网址中获取：http://www.icyfenix.com/jvms_javase7_cn/

最后，请允许我再介绍一下本书三位译者的技术背景与分工：

- **周志明**（www.icyfenix.com & weibo.com/icyfenix）：远光软件平台开发部部门经理，平台架构师，不愿意脱离编码的一线码农。著有《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》。关注各种 Java 应用，略懂 OSGi、Java 虚拟机和工作流。在本书翻译工作中负责全文统稿；前言和第 1、2、6、7 章的翻译；第 3、4、5 章的校审工作。
- **吴璞渊**（wupuyuan.iteye.com）：就职于西门子，偏向程序和工作流设计，喜好 Java 各种新技术并倒腾。在本书翻译工作中负责第 3 章以及第 4 章的 1 至 7 节。。
- **冶秀刚**（langyu.iteye.com）：思科平台工程师，从事分布式系统的研究与开发，爱好 Java 平台技术且正在努力成长中。在本书翻译工作中负责第 5 章及第 4 章的 9 至 11 节。

周志明

2011 年 11 月 2 日

版权声明

1. 本翻译工作完全基于个人兴趣爱好以及学术研究目的，不涉及出版或任何其他商业行为。本次翻译与 Oracle 或其他 Java 虚拟机厂商无关，译文是**非官方**的翻译。
2. 译者曾经尝试邮件联系过原文作者，但是一直未获得到回复。根据我国著作权法第 22 条规定，以教学、科研为目的，可以不经著作权人许可翻译其已经发表的作品，不向其支付报酬，但应当指明作者姓名、作品名称，且不得出版发行。因此本译文的传播，必须严格控制在学习与科学研究范围内，任何人未经原文作者和译者同意，不得将译文的全部或部分用于出版或其他商业行为。
3. 在符合第 2 条的前提下，任何人都可任意方式传播、使用本译文的部分或全部内容，无须预先知会译者，只要保留作、译者联系信息即可。如果需要进行商业使用，则必须到原作者和译者的授权。

附原文版权声明如下：

limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherv wise pass through to your licensees any licenses under Oracle's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.
a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be nfringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant
Implementation.
b With respect to any patent claims owned by Oracle and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect
to such claims if You initiate a claim against Oracle that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.
c Also with respect to any patent claims owned by Oracle and covered by the license granted

under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Oracle that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Oracle's source code or binary code materials nor, except with an appropriate and separate license from Oracle,
includes any of Oracle's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Oracle which corresponds to the Specification and that was available either (i) from Oracle 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Oracle if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES
THE SPECIFICATION IS PROVIDED "AS IS". ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
You will indemnify, hold harmless, and defend Oracle and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the

Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND
U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-
DoD acquisitions).

REPORT
If you provide Oracle with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS
Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply. The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee. This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

目录

前言

《Java 虚拟机规范》是一份完整的描述 Java 虚拟机是如何设计的规范文档。这份文档对于任何一个希望实现 Java 虚拟机的编译器作者，或者希望实现一个与规范相兼容的 Java 虚拟机的程序员来说都是必不可少的。

Java 虚拟机是一个抽象化的机器，整个规范中提及的 Java 虚拟机都是抽象化的概念，而不是特指 Oracle 或者其他某一间公司的 Java 虚拟机实现。这本书与一个具体的虚拟机实现之间的关系就犹如一份建筑蓝图与一间具体房屋之间的关系一样。Java 虚拟机具体实现（包括任何公司的 Java 虚拟机实现）必须包括本规范所描述的内容，但是除了少数绝对必要的地方外，本规范中的描述不应成为 Java 虚拟机具体实现的束缚。

《Java 虚拟机规范（Java SE 7 版）》将完全兼容于 Java SE 7 平台，并且完全支持《Java 语言规范（Java SE 7 版）》（Addison-Wesley 出版社，2005）中的描述的内容。

我们希望这个规范至少能作为一个“实验室”版本的虚拟机实现的完整描述。如果你正在考虑开发一个自己的 Java 虚拟机实现，可以随时联系我们以获得援助，确保你的实现具有百分百 Java 虚拟机规范兼容性。

Java 虚拟机源于由 James Gosling 在 1992 年设计，用于支持 Oak 程序语言的虚拟机。在 Java 虚拟机的发展历程中，Sun 的 Green 项目、FirstPerson 公司、LiveOak 项目、Java 产品组、JavaSoft 公司以及今天的 Oralce 公司的 Java 平台组中许多人都做出了直接或间接的贡献，作者对 Java 虚拟机的诸多贡献者和支持者表示感谢。

这本书最初是源于一份公司内部文档。Kathy Walrath 编著了本书的草稿版本，为世界提供了第一本关于 Java 语言内部是如何运作的书籍。Mary Campione 将本书转换为 HTML 版本，令大家可通过互联网站访问到本书。

《Java 虚拟机规范》的诞生，离不开 Java 产品团队的总经理 Ruth Hennigar 的大力支持，还有编辑 Lisa Friendly、Mike Hendrickson 以及他在 Addison-Wesley 出版社的团队所做出的编辑工作。读者对本书网上在线版草稿和印刷版草稿所提出诸多意见和建议，也对本书的质量提升起到重要作用，在此特别感谢 Richard Tuck 对原稿的仔细审查，还有 Bill Joy 对本书的审查、评价和指导意见，这些宝贵信息都对本书定稿有很大的帮助。

第二版说明

《Java 虚拟机规范（第二版）》将规范所描述内容的技术背景升级到了 Java 2 平台（JDK 1.2），它还包括了许多对原版的修正并且在不改变规范内容逻辑下，使描述变得更加清晰。我们也尝试调整了规范中的字体样式、勘误（希望勘误不会导致新的错误）以及对规范中模糊的部分增加额外的描述。另外，我们还修正了许多《Java 虚拟机规范（第一版）》和《Java 语言规范》之间不一致的内容。

我们很感谢所有为我们梳理过第一版规范，并指出问题的读者，特别感谢以下个人和团体，他们指出的问题甚至是还直接提供的修改意见：

Carla Schroer 与她在加州古本蒂诺、俄罗斯新西伯利亚的兼容性测试团队（尤其感谢其中的 Leonid Arbouzov 和 Alexei Kaigorodov）。他们煞费苦心地为第一版中各处可测试的场景编写了兼容性测试用例。在这个过程中，发现了许多处原版规范中不清晰和不完整的内容。

Jeroen Vermeulen、Janice Shepherd、Peter Bertelsen、Roly Perera、Joe Darcy，和 Sandra Loosemore 提交了许多有用的建议和反馈，这些建议和反馈对于第二版规范的改进工作有很大帮助。

Addison Wesley Longman 出版社的编辑 Marilyn Rash 和 Hilary Selby Polk 帮助我们在第二版中合并技术变更的同时，改进了规范的可读性和内容的布局排版，

还要特别感谢 Gilad Bracha，他对本书出版进行了严格的审查，另外他也是本书新增内容的主要贡献者，尤其是对于第 4、5 章。他对计算机理论的贡献以及他解决的《Java 虚拟机规范》和《Java 语言规范》之间的描述差异问题都极大地完善了本书。

Tim Lindholm

Frank Yellin

Java Software, Sun Microsystems

Java SE 7 版说明

Java SE 7 版的《Java 虚拟机规范》整合了自 1999 年《Java 虚拟机规范（第二版）》发布以来 Java 世界所出现的技术变化。另外，还修正了第二版中许多的错误，以及对目前主流 Java

虚拟机实现来说已经过时的内容。最后还处理率了一些 Java 虚拟机和 Java 语言概念的不清晰之处。

在 2004 年发布的 Java SE 5.0 版为 Java 语言带来了翻天覆地的变化，但是对 Java 虚拟机设计上的影响力则相对较小。在这个版本中，我们扩充了 Class 文件格式以便支持新的 Java 语言特性，譬如泛型和变长参数方法等。

在 2006 年发布的 Java SE 6.0 版看起来并没有为 Java 语言带来什么新的变化，但是对 Java 虚拟机的影响就比较大。如新的字节码验证方式，它源于 Eva Rose 的一篇硕士论文，文中以 Java Card 平台为背景，展示了 JVM 字节码验证的另一种全新的实现思路。这导致了 Java ME CLDC 第一版实现的诞生，并最终成为 Java SE 平台 Class 验证过程的理论基础。关于这部分内容将会在本书的第四章中介绍[①]。

Sheng Liang 实现了 Java ME CLDC 的验证器，Gilad Bracha 负责对该验证器作出详细说明，Antero Taivalsaari 则是整个 Java ME CLDC 规范的负责人。Alessandro Coglio 对字节码验证的分析和验证工作，对本规范作出了很大的贡献。Wei Tao、Frank Yellin、Tim Lindholm 和 Gilad Bracha 一起实现的 Prolog 验证器是 Java ME 和 Java SE 平台规范的共同基础。Wei Tao 后续继续实现了实际运用于 Java 虚拟机的验证器。Mingyao Yang 改进了规范和设计，形成了 Java SE 6 中最终实现版本。

该规范成文得益于以下 JSR 202 专家组成员：Peter Burka、Alessandro Coglio、Sanghoon Jin、Christian Kemper、Larry Rau、Eva Rose 以及 Mark Stolz。

在 2011 年发布的 Java SE 7 终于兑现了在 1997 年《Java 虚拟机规范（第一版）》中就已作出的承诺：“在未来，我们会对 Java 虚拟机进行适当的扩展，以便更好的支持其他语言运行于 JVM 之上”。Gilad Bracha 的工作是 Java 虚拟机中的热替换（Hotswapping）功能和以及在 Java 虚拟机静态类型系统上去支持动态类型语言实现。invokedynamic 指令以及支持这条指令的基础架构由 John Rose 以及 JSR 292 专家组成员：Ola Bini、Rémi Forax、Dan Heidinga、Fredrik Öhrström、Jochen Theodorou 进行开发。还有 Charlie Nutter 和 Christian Thalinger 作出了其他贡献。

还有许多人的名字应当出现在这里，他们在不同时间里面对 Java 虚拟机的设计和实现做出过贡献。我们今天所见的 Java 虚拟机拥有卓越的执行性能，这离不开 David Ungar 和他的同事们

[①] 译者注：由于原文涉及到大量方法原型和细节，译者认为翻译成中文意义不大。所以介绍此内容的 4.10.1 小节“类型检查验证”是本中文译本中唯一没有翻译的内容。读者可参考英文版 Java 虚拟机规范。

在 Sun 实验室 Self 项目中所积累的技术基础。这些技术最初用于 Self 语言，后来形成了 Animorphic Smalltalk 虚拟机，经过长期而曲折的发展，最终成为了今天 Oracle HotSpot 虚拟机的技术基础。Lars Bak 和 Urs Hoelzle 经历了所有上述的技术发展阶段，对于今天的 Java 虚拟机能够拥有大家认为理所当然的高效执行性能，他们实在是居功至伟。

在 Java 虚拟机运行越来越高效的同时，它也在变得越来越轻巧，KVM 就是一种由 Antero Taivalsaari 所设计，面向移动设备领域的 Java 虚拟机。在本规范中将会介绍到的新字节码验证技术最初就来实现在 KVM 之中。

本规范中很多意义深远的改进来自于 Martin Buchholz、Brian Goetz、Paul Hohensee、David Holmes、Karen Kinnear、Keith McGuigan、Jeff Nisewanger、 Mark Reinhold、Naoto Sato、Bill Pugh、Uday Dhanikonda、Janet Koenig、Adam Messinge、John Pampuch、Georges Saab 和 Bernard Traversat 所做出的贡献。Jon Courtney 和 Roger Riggs 帮助我们保证此规范的内容可同时适用于 Java ME 和 Java SE 平台。Leonid Arbouzov、Stanislav Avzan、Yuri Gaevsky、Ilya Mukhin、Sergey Reznick 和 Kirill Shirokov 在 Java 技术兼容包（Java Compatibility Kit，JCK）上作出了卓越贡献，以保证本规范中描述的内容是可测试并且是已测试的。

Gilad Bracha

Java SE 核心技术组，Sun Microsystems

Alex Buckley

Java 平台组，Oracle

第 1 章 引言

1.1 简史

Java 语言是一门通用的、面向对象的、支持并发的程序语言。它的语法与 C 和 C++语言非常相似，但隐藏了许多 C 和 C++中复杂、深奥及不安全的语言特性。Java 平台最初设计出来是用于解决基于网络的消费设备上的软件开发问题，它在设计上就考虑到要支持部署在不同架构的主机上，并且不同组件之间可以安全地交互。面对这些需求，编译出来的本地代码必须解决不同网络间传输问题、可以操作各式各样的客户端，并且还要代码在这些客户端上能安全正确地运行。

万维网的盛行伴随发生了一些十分有趣的事情：Web 浏览器允许数以百万计的用户共同在网上冲浪，通过很简单的方式访问丰富多样的内容。用户冲浪所使用的设备并不是其中的关键，它们仅仅是一种媒介，无论你的机器性能如何，无论你使用高速网络还是慢速的 Modem，这些外界因素本质上与你所看到、听到的内容没有任何关系。

Web 狂热者们很快就发现网络信息的载体——HTML 文档格式对信息的表达有很多限制，HTML 的一些扩展应用，譬如网页表单，让这些限制显得更加明显。显而易见的，没有任何浏览器能够承诺它可以提供给用户所需要的全部特性，扩展能力将是解决这个问题的唯一答案。

Sun 公司的 HotJava™浏览器是世界上第一款展现出 Java 语言某些有趣特性的浏览器，它允许把 Java 代码内嵌入 HTML 页面之内，在 HTML 页面呈现的时候，这些 Java 代码显式下载至浏览器中。而在浏览器获取这些代码之前，它们已被严谨地检查过以保证它们是安全的。与 HTML 语言一样，这些 Java 代码与网络和主机是完全无关的，无论代码来自哪里、在哪台机器上执行，它们执行时都能表现出一致的行为。

带有 Java 技术支持的网页浏览器将不再受限于它本身所提供的功能。浏览网页的用户可以放心地假定在他们机器上运行的动态内容不会损害他们的机器。软件开发人员编写一次代码，程序就可以运行在所有支持 Java 运行时环境的机器之上。

1.2 Java 虚拟机

Java 虚拟机是整个 Java 平台的基石，是 Java 技术用以实现硬件无关与操作系统无关的关

键部分，是 Java 语言生成出极小体积的编译代码的运行平台，是保障用户机器免于恶意代码损害的保护屏障。

Java 虚拟机可以看作是一台抽象的计算机。如同真实的计算机那样，它有自己的指令集以及各种运行时内存区域。使用虚拟机来实现一门程序设计语言有许多合理的理由，业界中流传最为久远的虚拟机可能是 UCSD Pascal 的 P-Code 虚拟机[①]。

第一个 Java 虚拟机的原型机是由 Sun Microsystems 公司实现的，它被用在一种类似 PDA（Personal Digital Assistant，俗称掌上电脑）的手持设备上仿真实现 Java 虚拟机指令集。时至今日，Oracle 已有许多 Java 虚拟机实现应用于移动设备、桌面电脑、服务器等领域。Java 虚拟机并不局限于特定的实现技术、主机硬件和操作系统，Java 虚拟机也不局限于特定的代码执行方式，它不强求使用解释器来执行程序，也可以通过把自己的指令集编译为实际 CPU 的指令来实现，它可以通过微代码（Microcode）来实现，或者甚至直接实现在 CPU 中。

Java 虚拟机与 Java 语言并没有必然的联系，它只与特定的二进制文件格式——Class 文件格式所关联，Class 文件中包含了 Java 虚拟机指令集（或者称为字节码、Bytecodes）和符号表，还有一些其他辅助信息。

基于安全方面的考虑，Java 虚拟机要求在 Class 文件中使用了许多强制性的语法和结构化约束，但任一门功能性语言都可以表示为一个能被 Java 虚拟机接收的有效的 Class 文件。作为一个通用的、机器无关的执行平台，任何其他语言的实现者都可以将 Java 虚拟机作为他们语言的产品交付媒介。

1.3 各章节提要

本书中其余章节的结构与大意如下：

- 第 2 章：概览了 Java 虚拟机整体架构。
- 第 3 章：介绍如何将 Java 语言编写的程序转换为 Java 虚拟机指令集描述。
- 第 4 章：定义 Class 文件格式，它是一种与硬件及操作系统无关的二进制格式，被用来表示编译后的类和接口。

[①] 译者注：由加州大学圣地亚哥分校（University of California, San Diego，UCSD）于 1978 年发布的高度可移植、机器无关的、运行 Pascal 语言的虚拟机。

- 第 5 章：定义 Java 虚拟机启动以及类和接口的加载、链接和初始化过程。
- 第 6 章：定义 Java 虚拟机指令集，按这些指令的指令助记符的照字母顺序来表示。
- 第 7 章：提供了一张以操作码值为索引的 Java 虚拟机操作码助记符表。

在《Java 虚拟机规范（第二版）》中，第 2 章是 Java 语言概览，这可以使读者更好的理解 Java 虚拟机规范，但它本身并不属于规范的一部分。

在本规范里没有再包含此章节的内容，读者可以参考《Java 语言规范（Java SE 7 版）》来获取这部分信息，如在本文中有需要引用这些信息的地方，将使用类似于“（JLS §x.y）”的形式来表示。

在《Java 虚拟机规范（第二版）》中，第 8 章用于描述 Java 虚拟机线程和共享内存的底层操作，它对应于《Java 语言规范（第一版）》的第 17 章。而《Java 语言规范（第二版）》出版时，第 17 章则对应于 JSR-133 专家组所发布的《Java 内存模型和线程规范》[①]，本规范中不再包含这部分内容，读者参考上述规范来获取关于线程与锁的信息。

1.4 说明

在本书中，我们所使用到的类和接口来自于 Java SE 平台的 API，无论何时，我们使用某个字母（譬如 N）来表示某个类或接口时，默认都是指 java.lang.N 所代表的那个类或接口，如果要描述其他包中的类或接口，我们将会使用全限定名。

无论何时，当我们提及某个类或接口的包是 java 或者它的子包（java.*），那就意味着这个类或接口是由引导类加载器进行加载的（§5.3.1）。无论何时，我们提及某个包是 java 包的子包时，就意味着这个包是由引导类加载器所定义的。

在本书中，下面两种字体含义为[②]：

```
这种字体用于代码样例
```

这些代码样例可能包括 Java 语言、Java 虚拟机的数据类型、异常和错误等等。

*斜体*用于描述 Java 虚拟机中的“汇编语言”，即操作码和操作数，也包括一些 Java 虚拟机运行时区域中的项目，有时也被用来说明一些新的条目和需要强调的内容。

[①] 译者注：《Java Memory Model and Thread Specification》：http://www.jcp.org/en/jsr/summary?id=133

[②] 译者注：由于中文和英文语言的差异，字体使用上译本无法完全跟随原文的使用方式。

第 2 章 Java 虚拟机结构

本规范描述的是一种抽象化的虚拟机的行为，而不是任何一种（译者注：包括 Oracle 公司自己的 HotSpot 和 JRockit 虚拟机）被广泛使用的虚拟机实现。

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

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

2.1 Class 文件格式

编译后被 Java 虚拟机所执行的代码使用了一种平台中立（不依赖于特定硬件及操作系统的）的二进制格式来表示，并且经常（但并非绝对）以文件的形式存储，因此这种格式被称为 Class 文件格式。Class 文件格式中精确地定义了类与接口的表示形式，包括在平台相关的目标文件格式中一些细节上的惯例[①]，例如字节序（Byte Ordering）等。

关于对 Class 文件格式细节的定义，请参见第 4 章“Class 文件格式”相关内容。

[①] 译者注：读者请勿误认为此处“平台相关的目标文件格式”是指在特定平台编译出的 Class 文件无法在其他平台中使用。相反，正是因为强制、明确地定义了本来会跟平台相关的细节，所以才达到了平台无关的效果。例如在 SPARC 平台上数字以 Big-Endian（高位的 Byte 放在内存中的低地址处）形式存储，在 x86 平台上数字则是以 Little-Endian（低位的 Byte 放在内存中的高地址处）形式存储的，如果不强制统一字节序的话，同一个 Class 文件的二进制形式放在不同平台上就可能被不同的方式解读。

2.2 数据类型

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

Java 虚拟机希望尽可能多的类型检查能在程序运行之前完成，换句话说，编译器应当在编译期间尽最大努力完成可能的类型检查，使得虚拟机在运行期间无需进行这些操作。原始类型的值不需要通过特殊标记或别的额外识别手段来在运行期确定它们的实际数据类型，也无需刻意将它们与引用类型的值区分开来，虚拟机的字节码指令本身就可以确定它的指令操作数的类型是什么，所以可以利用这种特性即可直接确定操作数的数值类型。举个例子，iadd、ladd、fadd 和 dadd 这几条指令的操作含义都是将两个数值相加，并返个相加的结果，但是每一条指令都有自己的专属操作数类型，此处按顺序分别为：int、long、float 和 double。关于虚拟机字节码指令的介绍，读者可以参见本章的“§2.11 指令集简介”部分。

Java 虚拟机是直接支持对象的，这里的对象可以是指动态分配的某个类的实例，也可以指某个数组的实例。虚拟机中使用 reference 类型[①]来表示对某个对象的引用，reference 类型的值读者可以想象成类似于一个指向对象的指针。每一个对象都可能存在多个指向它的引用，对象的操作、传递和检查都通过引用它的 reference 类型的数据进行操作。

2.3 原始类型与值

Java 虚拟机所支持的原始数据类型包括了数值类型（Numeric Types）、布尔类型（Boolean Type §2.3.4）和 returnAddress 类型（§2.3.3）三类。其中数值类型又分为整型类型（Integral Types，§2.3.1）和浮点类型（Floating-Point Types，§2.3.2）两种，其中：

[①] 译者注：这里的 reference 类型与 int、long、double 等类型是同一个层次的概念，reference 是前文提到过的引用类型（Reference Types）的一种，而 int、long、double 等则是前面提到的原始类型（Primitive Types）的一种。前者是具体的数据类型，后者是某种数据类型的统称，原文使用不同的英文字体标识，译者根据通常使用习惯，在译文中把具体类型使用小写英文表示，而类型统称则翻译为中文。

整数类型包括：

- byte 类型：值为 8 位有符号二进制补码整数，默认值为零。
- short 类型：值为 16 位有符号二进制补码整数，默认值为零。
- int 类型：值为 32 位有符号二进制补码整数，默认值为零。
- long 类型：值为 64 位有符号二进制补码整数，默认值为零。
- char 类型：值为使用 16 位无符号整数表示的、指向基本多文本平面（Basic Multilingual Plane，BMP[①]）的 Unicode 值，以 UTF-16 编码，默认值为 Unicode 的 null 值（'\u0000'）。

浮点类型包括：

- float 类型：值为单精度浮点数集合[②]中的元素，或者（如果虚拟机支持的话）是单精度扩展指数（Float-Extended-Exponent）集合中的元素。默认值为正数零。
- double 类型：取值范围是双精度浮点数集合中的元素，或者（如果虚拟机支持的话）是双精度扩展指数（Double-Extended-Exponent）集合中的元素。默认值为正数零。

布尔类型：

- boolean 类型：取值范围为布尔值 true 和 false，默认值为 false。

returnAddress 类型：

- returnAddress 类型：表示一条字节码指令的操作码（Opcode）。在所有的虚拟机支持的原始类型之中，只有 returnAddress 类型是不能直接 Java 语言的数据类型对应起来的。

2.3.1 整型类型与整型值

Java 虚拟机中的整型类型的取值范围如下：

- 对于 byte 类型，取值范围是从-128 至 127（-2^7 至 2^7-1），包括-128 和 127。
- 对于 short 类型，取值范围是从–32768 至 32767（-2^{15} 至 $2^{15}-1$），包括–32768 和 32767。

① 译者注：基本多文本平面相关解释可参见：“http://zh.wikipedia.org/zh/基本多文種平面”

② 译者注：单精度浮点数集合、双精度浮点数集合、单精度扩展指数集合和双精度扩展指数集合将会在稍后的§2.3.2 中详细介绍。

- 对于 int 类型，取值范围是从-2147483648 至 2147483647（-2^{31} 至 $2^{31}-1$），包括-2147483648 和 2147483647。
- 对于 long 类型，取值范围是从-9223372036854775808 至 9223372036854775807（-2^{63} 至 $2^{63}-1$），包括-9223372036854775808 和 9223372036854775807。
- 对于 char 类型，取值范围是从 0 至 65535，包括 0 和 65535。

2.3.2 浮点类型、取值集合及浮点值

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

IEEE 754 标准的内容不仅包括了正负带符号可数的数值（Sign-Magnitude Numbers），还包括了正负零、正负无穷大和一个特殊的“非数字”标识（Not-a-Number，下文用 NaN 表示）。NaN 值用于表示某些无效的运算操作，例如除数为零等情况。

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

对于一个非零的、可数的任意浮点值，都可以表示为 $s\times m\times 2^{(e-N+1)}$ 的形式，其中 s 可以是+1 或者-1，m 是一个小于 2^N 的正整数，e 是一个介于 $E_{min}=-(2^{K-1}-2)$ 和 $E_{max}=2^{K-1}-1$ 之间的整数（包括 E_{min} 和 E_{max}）。这里的 N 和 K 两个参数的取值范围决定于当前采用的浮点数值集合。部分浮点数使用这种规则得到的表示形式可能不是唯一的，例如在指定的数值集合内，可以存在一个数字 v，它能找到特定的 s、m 和 e 值来表示，使得其中 m 是偶数，并且 e 小于 2^{K-1}，这样我们就能够通过把 m 的值减半再将 e 的值增加 1 来的方式得到 v 的另外一种不同的表示形式。在这些表示形式中，如果其中某种表示形式中 m 的值满足条件 $m\geq 2^{N-1}$ 的话，那就称这种表示为标准表示（Normalized Representation），不满足这个条件的其他表示形式就称为非标准表示（Denormalized Representation）。如果某个数值不存在任何满足 $m\geq 2^{N-1}$ 的表示形式，即不存在任何标准表示，那就称这个数字为非标准值（Denormalized Value）。

在两个必须支持的浮点数值集合和两个可选的浮点数值集合内，对参数 N 和 K（也包括衍生参

数 E_{min} 和 E_{max}）的约束如表 2-1 所示。

表 2-1 浮点数值集合的参数

参数	单精度 浮点数集合	单精度扩展 指数集合	双精度 浮点数集合	双精度扩展 指数集合
N	24	24	53	53
K	8	≥11	11	≥15
E_{min}	+127	≥+1023	+1023	≥+16383
E_{max}	-126	≤-1022	-1022	≤-16382

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

上述四种数值集合都不仅包含可数的非零值，还包括五个特殊的数值：正数零、负数零、正无穷大、负无穷大和 NaN。

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

每一个单精度浮点数集合中的元素，都可以精确地使用 IEEE 754 标准中定义的单精度浮点格式表示出来，只有 NaN 一个例外。类似的，每一个双精度浮点数集合中的元素，都可以精确地使用 IEEE 754 标准中定义的双精度浮点格式表示出来，也只有 NaN 一个例外。不过请读者注意，在这里定义的单精度扩展指数集合和双精度扩展指数集合中的元素和 IEEE 754 标准里面单精度扩展和双精度扩展格式的表示并不完全一致。不过除了 Class 文件格式中必要的浮点数表示描述（§4.4.4，§4.4.5）以外，本规范并不特别要求表示浮点数值表示形式。

上面提到的单精度浮点数集合、单精度扩展指数集合、双精度浮点数集合和双精度扩展指数集合都并不是具体的数据类型。虚拟机实现使用一个单精度浮点数集合的元素来表示一个 float 类

型的数值在所有场景中都是可行的，但是在某些特定的上下文环境中，也允许虚拟机实现使用单精度扩展指数集合的元素来代替。类似的，虚拟机实现使用一个双精度浮点数集合的元素来表示一个 double 类型的数值在所有场景中都是可行的，但是在某些特定的上下文环境中，也允许虚拟机实现使用双精度扩展指数集合的元素来代替。

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

浮点数中，正数零和负数零是相等的，但是它们有一些操作会有区别。例如 1.0 除以 0.0 会产生正无穷大的结果，而 1.0 除以-0.0 则会产生负无穷大的结果。

NaN 是无序的，对它进行任何的数值比较和等值测试都会返回 false 的比较结果。值得一提的是，有且只有 NaN 一个数与自身比较是否数值上相等时会得到 false 的比较结果，任何数字与 NaN 进行非等值比较都会返回 true。

2.3.3 returnAddress 类型和值

returnAddress 类型会被 Java 虚拟机的 jsr、ret 和 jsr_w 指令[①]所使用。returnAddress 类型的值指向一条虚拟机指令的操作码。与前面介绍的那些数值类的原始类型不同，returnAddress 类型在 Java 语言之中并不存在相应的类型，也无法在程序运行期间更改 returnAddress 类型的值。

2.3.4 boolean 类型

虽然 Java 虚拟机定义了 boolean 这种数据类型，但是只对它提供了非常有限的支持。在 Java 虚拟机中没有任何供 boolean 值专用的字节码指令，在 Java 语言之中涉及到 boolean 类型值的运算，在编译之后都使用 Java 虚拟机中的 int 数据类型来代替。

Java 虚拟机直接支持 boolean 类型的数组，虚拟机的 newarray 指令可以创建这种数组。

[①] 译者注：这几条指令以前主要被使用来实现 finally 语句块，后来改为冗余 finally 块代码的方式来实现，甚至到了 JDK7 时，虚拟机已不允许 Class 文件内出现这几条指令。那相应地，returnAddress 类型就处于名存实亡的状态。

boolean 的数组类型的访问与修改共用 byte 类型数组的 baload 和 bastore 指令[①]。

2.4 引用类型与值

Java 虚拟机中有三种引用类型：类类型（Class Types）、数组类型（Array Types）和接口类型（Interface Types）。这些引用类型的值分别由类实例、数组实例和实现了某个接口的类实例或数组实例动态创建。

其中，数组类型还包含一个单一维度（即长度不由其类型决定）的组件类型（Component Type），一个数组的组件类型也可以是数组。但从任意一个数组开始，如果发现其组件类型也是数组类型的话，继续重复取这个数组的组件类型，这样操作不断执行，最终一定可以遇到组件类型不是数组的情况，这时就把这种类型成为数组类型的元素类型（Element Type）。数组的元素类型必须是原始类型、类类型或者接口类型之中的一种。

在引用类型的值中还有一个特殊的值：null，当一个引用不指向任何对象的时候，它的值就用 null 来表示。一个为 null 的引用，在没有上下文的情况下不具备任何实际的类型，但是有具体上下文时它可转型为任意的引用类型。引用类型的默认值就是 null。

Java 虚拟机规范并没有规定 null 在虚拟机实现中应当怎样编码表示。

2.5 运行时数据区

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

[①] 在 Oracle 公司的虚拟机实现里，Java 语言里面的 boolean 数组将会被编码成 Java 虚拟机的 byte 数组，每个 boolean 元素占 8 位长度。

2.5.1 PC 寄存器

Java 虚拟机可以支持多条线程同时执行（可参考《Java 语言规范》第 17 章），每一条 Java 虚拟机线程都有自己的 PC（Program Counter）寄存器。在任意时刻，一条 Java 虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法（Current Method，§2.6）。如果这个方法不是 native 的，那 PC 寄存器就保存 Java 虚拟机正在执行的字节码指令的地址，如果该方法是 native 的，那 PC 寄存器的值是 undefined。PC 寄存器的容量至少应当能保存一个 returnAddress 类型的数据或者一个与平台相关的本地指针的值。

2.5.2 Java 虚拟机栈

每一条 Java 虚拟机线程都有自己私有的 Java 虚拟机栈（Java Virtual Machine Stack）[①]，这个栈与线程同时创建，用于存储栈帧（Frames，§2.6）。Java 虚拟机栈的作用与传统语言（例如 C 语言）中的栈非常类似，就是用于存储局部变量与一些过程结果的地方。另外，它在方法调用和返回中也扮演了很重要的角色。因为除了栈帧的出栈和入栈之外，Java 虚拟机栈不会再受其他因素的影响，所以栈帧可以在堆中分配[②]，Java 虚拟机栈所使用的内存不需要保证是连续的。

Java 虚拟机规范允许 Java 虚拟机栈被实现成固定大小的或者是根据计算动态扩展和收缩的。如果采用固定大小的 Java 虚拟机栈设计，那每一条线程的 Java 虚拟机栈容量应当在线程创建的时候独立地选定。Java 虚拟机实现应当提供给程序员或者最终用户调节虚拟机栈初始容量的手段，对于可以动态扩展和收缩 Java 虚拟机栈来说，则应当提供调节其最大、最小容量的手段。

Java 虚拟机栈可能发生如下异常情况：

- 如果线程请求分配的栈容量超过 Java 虚拟机栈允许的最大容量时，Java 虚拟机将会抛出一个 StackOverflowError 异常。
- 如果 Java 虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是目前无法申请到足够的

[①] 在 Java 虚拟机规范第一版之中，Java 虚拟机栈也被称为“Java 栈”。

[②] 译者注：请读者注意避免混淆 Stack、Heap 和 Java （VM）Stack、Java Heap 的概念，Java 虚拟机的实现本身是由其他语言编写的应用程序，在 Java 语言程序的角度上看分配在 Java Stack 中的数据，而在实现虚拟机的程序角度上看则可以是分配在 Heap 之中。

内存去完成扩展，或者在建立新的线程时没有足够的内存去创建对应的虚拟机栈，那 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

2.5.3 Java 堆

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

Java 堆在虚拟机启动的时候就被创建，它存储了被自动内存管理系统（Automatic Storage Management System，也即是常说的“Garbage Collector（垃圾收集器）”）所管理的各种对象，这些受管理的对象无需，也无法显式地被销毁。本规范中所描述的 Java 虚拟机并未假设采用什么具体的技术去实现自动内存管理系统。虚拟机实现者可以根据系统的实际需要来选择自动内存管理技术。Java 堆的容量可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。Java 堆所使用的内存不需要保证是连续的。

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

Java 堆可能发生如下异常情况：

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

2.5.4 方法区

在 Java 虚拟机中，方法区（Method Area）是可供各条线程共享的运行时内存区域。方法区与传统语言中的编译代码储存区（Storage Area Of Compiled Code）或者操作系统进程的正文段（Text Segment）的作用非常类似，它存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法（§2.9）。

方法区在虚拟机启动的时候被创建，虽然方法区是堆的逻辑组成部分，但是简单的虚拟机实现可以选择在这个区域不实现垃圾收集。这个版本的 Java 虚拟机规范也不限定实现方法区的内存位

置和编译代码的管理策略。方法区的容量可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。方法区在实际内存空间中可以是不连续的。

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

方法区可能发生如下异常情况：

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

2.5.5 运行时常量池

运行时常量池（Runtime Constant Pool）是每一个类或接口的常量池（Constant_Pool，§4.4）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。运行时常量池扮演了类似传统语言中符号表（Symbol Table）的角色，不过它存储数据范围比通常意义上的符号表要更为广泛。

每一个运行时常量池都分配在 Java 虚拟机的方法区之中（§2.5.4），在类和接口被加载到虚拟机后，对应的运行时常量池就被创建出来。

在创建类和接口的运行时常量池时，可能会发生如下异常情况：

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

关于构造运行时常量池的详细信息，可以参考“第 5 章 加载、链接和初始化”的内容。

2.5.6 本地方法栈

Java 虚拟机实现可能会使用到传统的栈（通常称之为“C Stacks”）来支持 native 方法（指使用 Java 以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（Native Method Stack）。当 Java 虚拟机使用其他语言（例如 C 语言）来实现指令集解释器时，也会使用到本地方法栈。如果 Java 虚拟机不支持 natvie 方法，并且自己也不依赖传统栈的话，可以无需支持本地方法栈，如果支持本地方法栈，那这个栈一般会在线程创建的时候按线程分配。

Java 虚拟机规范允许本地方法栈被实现成固定大小的或者是根据计算动态扩展和收缩的。如果采用固定大小的本地方法栈，那每一条线程的本地方法栈容量应当在栈创建的时候独立地选定。一般情况下，Java 虚拟机实现应当提供给程序员或者最终用户调节虚拟机栈初始容量的手段，对于长度可动态变化的本地方法栈来说，则应当提供调节其最大、最小容量的手段。

本地方法栈可能发生如下异常情况：

- 如果线程请求分配的栈容量超过本地方法栈允许的最大容量时，Java 虚拟机将会抛出一个 StackOverflowError 异常。
- 如果本地方法栈可以动态扩展，并且扩展的动作已经尝试过，但是目前无法申请到足够的内存去完成扩展，或者在建立新的线程时没有足够的内存去创建对应的本地方法栈，那 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

2.6 栈帧

栈帧（Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接（Dynamic Linking）、方法返回值和异常分派（Dispatch Exception）。

栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。栈帧的存储空间分配在 Java 虚拟机栈（§2.5.5）之中，每一个栈帧都有自己的局部变量表（Local Variables，§2.6.1）、操作数栈（Operand Stack，§2.6.2）和指向当前方法所属的类的运行时常量池（§2.5.5）的引用。

局部变量表和操作数栈的容量是在编译期确定，并通过方法的 Code 属性（§4.7.3）保存及提供给栈帧使用。因此，栈帧容量的大小仅仅取决于 Java 虚拟机的实现和方法调用时可被分配的内存。

在一条线程之中，只有目前正在执行的那个方法的栈帧是活动的。这个栈帧就被称为是当前栈帧（Current Frame），这个栈帧对应的方法就被称为是当前方法（Current Method），定义这个方法的类就称作当前类（Current Class）。对局部变量表和操作数栈的各种操作，通常都指的是对当前栈帧的对局部变量表和操作数栈进行的操作。

如果当前方法调用了其他方法，或者当前方法执行结束，那这个方法的栈帧就不再是当前栈帧了。当一个新的方法被调用，一个新的栈帧也会随之而创建，并且随着程序控制权移交到新的方法而成为新的当前栈帧。当方法返回的之际，当前栈帧会传回此方法的执行结果给前一个栈帧，在方

法返回之后，当前栈帧就随之被丢弃，前一个栈帧就重新成为当前栈帧了。

请读者特别注意，栈帧是线程本地私有的数据，不可能在一个栈帧之中引用另外一条线程的栈帧。

2.6.1 局部变量表

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

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

局部变量使用索引来进行定位访问，第一个局部变量的索引值为零，局部变量的索引值是从零至小于局部变量表最大容量的所有整数。

long 和 double 类型的数据占用两个连续的局部变量，这两种类型的数据值采用两个局部变量之中较小的索引值来定位。例如我们讲一个 double 类型的值存储在索引值为 n 的局部变量中，实际上的意思是索引值为 n 和 n+1 的两个局部变量都用来存储这个值。索引值为 n+1 的局部变量是无法直接读取的，但是可能会被写入，不过如果进行了这种操作，就将会导致局部变量 n 的内容失效掉。

上文中提及的局部变量 n 的 n 值并不要求一定是偶数，Java 虚拟机也不要求 double 和 long 类型数据采用 64 位对其的方式存放在连续的局部变量中。虚拟机实现者可以自由地选择适当的方式，通过两个局部变量来存储一个 double 或 long 类型的值。

Java 虚拟机使用局部变量表来完成方法调用时的参数传递，当一个方法被调用的时候，它的参数将会传递至从 0 开始的连续的局部变量表位置上。特别地，当一个实例方法被调用的时候，第 0 个局部变量一定是用来存储被调用的实例方法所在的对象的引用（即 Java 语言中的“this”关键字）。后续的其他参数将会传递至从 1 开始的连续的局部变量表位置上。

2.6.2 操作数栈

每一个栈帧（§2.6）内部都包含一个称为操作数栈（Operand Stack）的后进先出（Last-In-First-Out，LIFO）栈。栈帧中操作数栈的长度由编译期决定，并且存储于类和接口的二进制表示之中，既通过方法的 Code 属性（§4.7.3）保存及提供给栈帧使用。

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

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

举个例子，iadd 字节码指令的作用是将两个 int 类型的数值相加，它要求在执行的之前操作数栈的栈顶已经存在两个由前面其他指令放入的 int 型数值。在 iadd 指令执行时，2 个 int 值从操作栈中出栈，相加求和，然后将求和结果重新入栈。在操作数栈中，一项运算常由多个子运算（Subcomputations）嵌套进行，一个子运算过程的结果可以被其他外围运算所使用。

每一个操作数栈的成员（Entry）可以保存一个 Java 虚拟机中定义的任意数据类型的值，包括 long 和 double 类型。

在操作数栈中的数据必须被正确地操作，这里正确操作是指对操作数栈的操作必须与操作数栈栈顶的数据类型相匹配，例如不可以入栈两个 int 类型的数据，然后当作 long 类型去操作他们，或者入栈两个 float 类型的数据，然后使用 iadd 指令去对它们进行求和。有一小部分 Java 虚拟机指令（例如 dup 和 swap 指令）可以不关注操作数的具体数据类型，把所有在运行时数据区中的数据当作裸类型（Raw Type）数据来操作，这些指令不可以用来修改数据，也不可以拆散那些原本不可拆分的数据，这些操作的正确性将会通过 Class 文件的校验过程（§4.10）来强制保障。

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

2.6.3 动态链接

每一个栈帧（§2.6）内部都包含一个指向运行时常量池（§2.5.5）的引用来支持当前方法的代码实现动态链接（Dynamic Linking）。在 Class 文件里面，描述一个方法调用了其他方法，或者访问其成员变量是通过符号引用（Symbolic Reference）来表示的，动态链接的作用就是将这些符号引用所表示的方法转换为实际方法的直接引用。类加载的过程中将要解析掉尚未被解析的符号引用，并且将变量访问转化为访问这些变量的存储结构所在的运行时内存位置的正确偏移量。

由于动态链接的存在，通过晚期绑定（Late Binding）使用的其他类的方法和变量在发生变化时，将不会对调用它们的方法构成影响。

2.6.4 方法正常调用完成

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

在这种场景下，当前栈帧（§2.6）承担着回复调用者状态的责任，其状态包括调用者的局部变量表、操作数栈和被正确增加过来表示执行了该方法调用指令的程序计数器等。使得调用者的代码能在被调用的方法返回并且返回值被推入调用者栈帧的操作数栈后继续正常地执行。

2.6.5 方法异常调用完成

方法异常调用完成是指在方法的执行过程中，某些指令导致了 Java 虚拟机抛出异常（§2.10），并且虚拟机抛出的异常在该方法中没有办法处理，或者在执行过程中遇到了 athrow 字节码指令显式地抛出异常，并且在该方法内部没有把异常捕获住。如果方法异常调用完成，那一定不会有方法返回值返回给它的调用者。

2.7 对象的表示

Java 虚拟机规范不强制规定对象的内部结构应当如何表示[①]。

2.8 浮点算法

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

2.8.1 Java 虚拟机和 IEEE 754 中的浮点算法

Java 虚拟机中支持的浮点算法和 IEEE 754 标准中的主要差别有：

- 在 Java 虚拟机中的浮点操作在遇到非法操作，如被零除（Divison By Zero）、上限溢出（Overflow）、下限溢出（Underflow）和非精确（Inexact）时，不会抛出 exception、trap 或者其他 IEEE 754 异常情况中定义的信号。
- 在 Java 虚拟机中不支持 IEEE 754 中的信号浮点比较（Signaling Floating-Point Comparisons）。
- 在 Java 虚拟机中，舍入操作永远使用 IEEE 754 规范中定义的向最接近数舍入模式（Round To Nearest Mode），无法精确表示的结果将会舍入为最接近的可表示值来保证此值的最低有效位为零（A Zero Least-Significant Bit），这种模式也是 IEEE 754 中的默认模式。不过在 Java 虚拟机里面，将浮点数转化为整型数是使用向零舍入

[①] 在一些 Oracle 的 Java 虚拟机实现中，指向对象实例的引用是一个指向句柄的指针，这个句柄包含两部分信心，一部分是指向这个对象所包括的方法表以及指向这个对象所属类相关的信息；另一部分是指向在堆中分配的对象实例数据。（译者注：这条注释在 10 多年前出版的 Java 虚拟机规范第二版中就已经存在，第三版中仅仅是将 Sun 修改为 Oracle 而已，所表达的实际信息已比较陈旧。在 HotSpot 虚拟机中，指向对象的引用并不通过句柄，而是直接指向堆中对象的实例数据，因此 HotSpot 虚拟机并不包括在上面所描述的“一些 Oracle 的 Java 虚拟机实现”范围之内）

（Round Toward Zero），这点属于特别定义，并不意味着 Java 虚拟机要改变浮点运算的舍入模式[①]。

- 在 Java 虚拟机中不支持 IEEE 754 的单精度扩展和双精度扩展格式（Single Extended Or Double Extended Format），但是在双精度浮点数集合和双精度扩展指数集合（Double And Double Extended-Exponent Value Sets，§2.3.2）范围与单精度扩展指数格式的表示方位会有重叠。虚拟机实现可以选择是否支持的单精度扩展指数和双精度扩展指数集合并不等同于 IEEE 754 中的单精度和双精度扩展格式：IEEE 754 中的扩展格式规定了扩展精度与扩展指数的范围。

2.8.2 浮点模式

每一个方法都有一项属性称为浮点模式（Floating-Point Mode），取值有两种，要么是 FP-strict 模式要么是非 FP-strict 模式。方法的浮点模式决定于 Class 文件中代表该方法的 method_info 结构（§4.6）的访问标志（access_flags）中的 ACC_STRICT 标志位的取值。如果此标志位为真，则该方法的浮点模式就是 FP-strict，否则就是非 FP-strict 模式。

需要注意，编译器编译出来的方法的 ACC_STRICT 标志位对于 JDK 1.1 或者更早的 JDK 版本没有效果。

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

如果虚拟机实现支持单精度指数扩展集合（§2.3.2），那在非 FP-strict 模式的操作数栈上，除非数值集合转换（§2.8.3）所禁止的，否则一个 float 类型的值将可能会超过单精度浮点数集合的范围。同样的，如果虚拟机实现支持双精度指数扩展集合（§2.3.2），那在非 FP-strict 模式的操作数栈上，除非数值集合转换（§2.8.3）所禁止的，否则一个 double 类型的值将可能会超过双精度浮点数集合的范围。

在其他的上下文中，无论是操作数栈或者别的地方都不再特别去关注浮点模式，float 和

[①] 译者注：在 IEEE 754 中定义了 4 种舍入模式，除了上面提到的向最接近数舍入和向零舍入以外，还有向正无穷舍入和向负无穷舍入两种模式。向最接近数舍入模式既我们平常所说的“四舍五入”法，而向零舍入既平常所说的“去尾”法舍入。

double 两种浮点值都分别限于单精度和双精度浮点数集合之中。类和实例的字段、数组元素、局部变量和方法参数的取值范围都限于标准的数值集合之中。

2.8.3 数值集合转换

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

在数值集合转换发生的位置，虚拟机实现允许对数值执行下面操作之一：

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

此外，在数值集合转换发生的位置，下面操作是必须的：

- 假设正在执行的 Java 虚拟机字节码指令是非 FP-strict 模式的，但这条指令导致了一个 float 类型的值推入到一个 FP-strict 模式的操作数栈中，例如作为方法参数进行传递或者存储进局部变量、字段或者数组元素之中。如果这个数值不是单精度浮点数集合中的元素，需要将其映射到单精度浮点数集合中数值最接近的元素。
- 假设正在执行的 Java 虚拟机字节码指令是非 FP-strict 模式的，但这条指令导致了一个 double 类型的值推入到一个 FP-strict 模式的操作数栈中，例如作为方法参数进行传递或者存储进局部变量、字段或者数组元素之中。如果这个数值不是双精度浮点数集合中的元素，需要将其映射到双精度浮点数集合中数值最接近的元素。

在方法调用中的参数传递（包括 native 方法的调用）、一个非 FP-strict 模式的方法返回浮点型的结果到 FP-strict 模式的调用者栈帧中或者在非 FP-strict 模式的方法中存储浮点型数值到局部变量、字段或者数组元素之中都可能会导致上述的数值集合转换发生。

并非所有扩展指数集合中的数值都可以精确映射到标准浮点数值集合的元素之中。如果进行映射的数值过大（扩展指数集合的指数可能比标准数值集合的允许最大值要大），无法在标准数值集合之中精确表示的话，这个数字将会被转化称对应类型的（正或负）无穷大。如果进行映射的数值过大（扩展指数集合的指数可能比标准数值集合的允许最小值要小），无法在标准数值集合之中精

确表示的话，这个数字将会被转化成最接近的可以表示非正规值（Denormalized Value，§2.3.2）或者相同正负符号零。

2.9 初始化方法的特殊命名

在 Java 虚拟机层面上，Java 语言中的构造函数在《Java 语言规范 （第三版）》（下文简称 JLS3，§8.8）是以一个名为<init>的特殊实例初始化方法的形式出现的，<init>这个方法名称是由编译器命名的，因为它并非一个合法的 Java 方法名字，不可能通过程序编码的方式实现。实例初始化方法只能在实例的初始化期间，通过 Java 虚拟机的 invokespecial 指令来调用，只有在实例正在构造的时候，实例初始化方法才可以被调用访问（JLS3，§6.6）。

一个类或者接口最多可以包含不超过一个类或接口的初始化方法，类或者接口就是通过这个方法完成初始化的（§5.5）。这个方法是一个不包含参数的静态方法，名为<clinit>[①]。这个名字也是由编译器命名的，因为它并非一个合法的 Java 方法名字，不可能通过程序编码的方式实现。类或接口的初始化方法由 Java 虚拟机自身隐式调用，没有任何虚拟机字节码指令可以调用这个方法，只有在类的初始化阶段中会被虚拟机自身调用。

2.10 异常

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

绝大多数的异常的产生都是由于当前线程执行的某个操作所导致的，这种可以称为是同步的异常。与之相对的，异步异常是指在程序的其他任意地方进行的动作而导致的异常。Java 虚拟机中异常的出现总是由下面三种原因之一导致的：

1. 虚拟机同步检测到程序发生了非正常的执行情况，这时异常将会紧接着在发生非正常执行情况的字节码指令之后抛出。例如：

[①] 在 Class 文件中把其他方法命名为<clinit>是没有意义的，这些方法并不是类或接口的初始化方法，它们既不能被字节码指令调用，也不会被虚拟机自己调用。

- 字节码指令所蕴含的操作违反了 Java 语言的语义，如访问一个超出数组边界范围的元素。
- 类在加载或者链接时出现错误。
- 使用某些资源的时候产生资源限制，例如使用了太多的内存。

2. athrow 字节码指令被执行。
3. 由于以下原因，导致了异步异常的出现：
   - 调用了 Thread 或者 ThreadGroup 的 stop 方法。
   - Java 虚拟机实现的内部程序错误。

   当某条线程调用了 stop 方法时，将会影响到其他的线程，或者在线程组中的所有线程。这时候其他线程中出现的异常就是异步异常，因为这些异常可能出现在程序执行过程的任何位置。虚拟机的内部异常也被认为是一种异步异常（§6.3）

《Java 虚拟机规范》允许在异步异常被抛出时额外执行一小段有限的代码，允许代码优化器在不违反 Java 语言语义的前提下检测并把这些异常在可处理它们的地方抛出[①]。

抛出异常的动作在 Java 虚拟机之中是一种被精确定义的程序控制权转移过程，当异常抛出、程序控制权发生转移的那一刻，所有在异常抛出的位置之前的字节码指令所产生的影响[②]都应当是可以被观察到的，而在异常抛出的位置之后的字节码指令，则应当是没有被执行过的。如果虚拟机执行的代码是被优化后的代码[③]，有一些在异常出现位置之后的代码可能已经被执行了，那这些优化过的代码必须保证被它们提前执行所产生的影响对用户程序来说都是不可见的。

由 Java 虚拟机执行的每一个方法都会配有零至多个异常处理器（Exception Handlers），异常处理器描述了其在方法代码中的有效作用范围（通过字节码偏移量范围来描述）、能处理的异常类型以及处理异常的代码所在的位置。要判断某个异常处理器是否可以处理某个具体的异常，需要同时检查异常出现的位置是否在异常处理的有效作用范围内并且出现的异常是否异常处理器声

[①] 对于简单的 Java 虚拟机实现，可以把异步异常简单地放在程序控制权转移指令上处理。因为程序终归是有限的，总会遇到控制权转移的指令，所以异步异常抛出的延迟时间也是有限的。如果能保证在控制权转移指令之间的代码没有异步异常抛出，那代码生成器就可以有相当的灵活性进行指令重排序优化来获取更好的性能。相关的资料推荐进一步阅读论文：《Polling Efficiently on Stock Hardware》，Marc Feeley，Proc.1993，《Conference on Functional Programming and Computer Architecture》，Copenhagen，Den-mark，第 179-187 页。

[②] 译者注：这里的“影响”包括了异常出现之前的字节码指令执行后对局部变量表、操作数栈、其他运行时数据区域以及虚拟机外部资源产生的影响。

[③] 译者注：这里的“优化后的代码”主要是指进行了指令重排序优化的代码。

明可以处理的异常类型或其子类型两个条件。当有异常被抛出时，Java 虚拟机搜索当前方法的包含的各个异常处理器，如果能找到可以处理该异常的异常处理器，则将代码控制权转向到异常处理器中描述的处理异常的分支之中。

搜索异常处理器时的搜索顺序是很关键的，在 Class 文件里面，每个方法的异常处理器都存储在一个表中（§4.7.3）。在运行时，当有异常出现之后，Java 虚拟机就按照 Class 文件中的异常处理器表描述异常处理器的先后顺序，从前至后进行搜索。

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

2.11 字节码指令集简介

Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的操作码（Opcode）以及跟随其后的零至多个代表此操作所需参数的操作数（Operands）所构成。虚拟机中许多指令并不包含操作数，只有一个操作码。

如果忽略异常处理，那 Java 虚拟机的解释器使用下面这个伪代码的循环即可有效地工作：

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

操作数的数量以及长度取决于操作码，如果一个操作数的长度超过了一个字节，那它将会以 Big-Endian 顺序存储——即高位在前的字节序。举个例子，如果要将一个 16 位长度的无符号整数使用两个无符号字节存储起来（将它们命名为 byte1 和 byte2），那它们的值应该是这样的：

```
(byte1 << 8) | byte2
```

字节码指令流应当都是单字节对齐的，只有“tableswitch”和“lookupswitch”两条指令例外，由于它们的操作数比较特殊，都是以 4 字节为界划分开的，所以这两条指令那个也需要预留出相应的空位来实现对齐。

限制 Java 虚拟机操作码的长度为一个字节，并且放弃了编译后代码的参数长度对齐，是为了

尽可能地获得短小精干的编译代码，即使这可能会让 Java 虚拟机的具体实现付出一定的性能成本为代价。由于每个操作码只能有一个字节长度，所以直接限制了整个指令集的数量[①]，又由于没有假设数据是对齐好的，这就意味着虚拟机处理那些超过一个字节的数据的时候，不得不在运行时从字节中重建出具体数据的结构，这在某种程度上会损失一些性能。

2.11.1 数据类型与 Java 虚拟机

在 Java 虚拟机的指令集中，大多数的指令都包含了其操作所对应的数据类型信息。举个例子，iload 指令用于从局部变量表中加载 int 型的数据到操作数栈中，而 fload 指令加载的则是 float 类型的数据。这两条指令的操作可能会是由同一段代码来实现的，但它们必须拥有各自独立的操作符。

对于大部分与数据类型相关的字节码指令，他们的操作码助记符中都有特殊的字符来表明专门为哪种数据类型服务：i 代表对 int 类型的数据操作，l 代表 long，s 代表 short，b 代表 byte，c 代表 char，f 代表 float，d 代表 double，a 代表 reference。也有一些指令的助记符中没有明确的指明操作类型的字母，例如 arraylength 指令，它没有代表数据类型的特殊字符，但操作数永远只能是一个数组类型的对象。还有另外一些指令，例如无条件跳转指令 goto 则是与数据类型无关的。

由于 Java 虚拟机的操作码长度只有一个字节，所以包含了数据类型的操作码对指令集的设计带来了很大的压力：如果每一种与数据类型相关的指令都支持 Java 虚拟机所有运行时数据类型的话，那恐怕就会超出一个字节所能表示的数量范围了。因此，Java 虚拟机的指令集对于特定的操作只提供了有限的类型相关指令去支持它，换句话说，指令集将会故意被设计成非完全独立的（Not Orthogonal，即并非每种数据类型和每一种操作都有对应的指令）。有一些单独的指令可以在必要的时候用来将一些不支持的类型转换为可被支持的类型。

表 2.2 列举了 Java 虚拟机所支持的字节码指令集，通过使用数据类型列所代表的特殊字符替换 opcode 列的指令模板中的 T，就可以得到一个具体的字节码指令。如果在表中指令模板与数据类型两列共同确定的格为空，则说明虚拟机不支持对这种数据类型执行这项操作。例如 load 指令有操作 int 类型的 iload，但是没有操作 byte 类型的同类指令。

[①] 译者注：字节码无法超过 256 条的限制就来源于此。

请注意，从表 2.2 中看来，大部分的指令都没有支持整数类型 byte、char 和 short，甚至没有任何指令支持 boolean 类型。编译器会在编译期或运行期会将 byte 和 short 类型的数据带符号扩展（Sign-Extend）为相应的 int 类型数据，将 boolean 和 char 类型数据零位扩展（Zero-Extend）为相应的 int 类型数据。与之类似的，在处理 boolean、byte、short 和 char 类型的数组时，也会转换为使用对应的 int 类型的字节码指令来处理。因此，大多数对于 boolean、byte、short 和 char 类型数据的操作，实际上都是使用相应的对 int 类型作为运算类型（Computational Type）。

表 2.2 Java 虚拟机指令集所支持的数据类型

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				

Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_Tcmp OP			if_icmp OP					if_acmp OP
Treturn			ireturn	lretu rn	fretu rn	dretu rn		areturn

在 Java 虚拟机中，实际类型与运算类型之间的映射关系，如表 2.3 所示。

表 2.3 Java 虚拟机指令集所支持的数据类型

实际类型	运算类型	分类
boolean	int	分类一
byte	int	分类一
char	int	分类一
short	int	分类一
int	int	分类一
float	float	分类一
reference	reference	分类一
returnAddress	returnAddress	分类一
long	long	分类二

double	double	分类二

有部分对操作栈进行操作的 Java 虚拟机指令（例如 pop 和 swap 指令）是与具体类型无关的，不过这些指令也必须受到运算类型分类的限制，这些分类也在表 2.3 中列出了。

2.11.2 加载和存储指令

加载和存储指令用于将数据从栈帧（§2.6）的局部变量表（§2.6.1）和操作数栈之间来回传输（§2.6.2）：

- 将一个局部变量加载到操作栈的指令包括有：iload、iload_<n>、lload、lload_<n>、fload、fload_<n>、dload、dload_<n>、aload、aload_<n>
- 将一个数值从操作数栈存储到局部变量表的指令包括有：istore、istore_<n>、lstore、lstore_<n>、fstore、fstore_<n>、dstore、dstore_<n>、astore、astore_<n>
- 将一个常量加载到操作数栈的指令包括有：bipush、sipush、ldc、ldc_w、ldc2_w、aconst_null、iconst_m1、iconst_<i>、lconst_<l>、fconst_<f>、dconst_<d>
- 扩充局部变量表的访问索引的指令：wide

访问对象的字段或数组元素（§2.11.5）的指令也同样会与操作数栈传输数据。

上面所列举的指令助记符中，有一部分是以尖括号结尾的（例如 iload_<n>），这些指令助记符实际上是代表了一组指令（例如 iload_<n>，它代表了 iload_0、iload_1、iload_2 和 iload_3 这几条指令）。这几组指令都是某个带有一个操作数的通用指令（例如 iload）的特殊形式，对于这若干组特殊指令来说，它们表面上没有操作数，不需要进行取操作数的动作，但操作数都是在指令中隐含的。除此之外，他们的语义与原生的通用指令完全一致（例如 iload_0 的语义与操作数为 0 时的 iload 指令语义完全一致）。在尖括号之间的字母制定了指令隐含操作数的数据类型，<i>代表是 int 形数据，<l>代表 long 型，<f>代表 float 型，<d>代表 double 型。在操作 byte、char 和 short 类型数据时，也用 int 类型表示（§2.11.1）。

这种指令表示方法，在整个《Java 虚拟机规范》之中都是通用的。

2.11.3 运算指令

算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。大体上运算指令可以分为两种：对整型数据进行运算的指令与对浮点型数据进行运算的指令，无论是那种算术指令，都是使用 Java 虚拟机的数字类型的。数据没有直接支持 byte、short、char 和 boolean 类型（§2.11.1）的算术指令，对于这些数据的运算，都是使用操作 int 类型的指令。整数与浮点数的算术指令在溢出和被零除的时候也有各自不同的行为，所有的算术指令包括：

- 加法指令：iadd、ladd、fadd、dadd
- 减法指令：isub、lsub、fsub、dsub
- 乘法指令：imul、lmul、fmul、dmul
- 除法指令：idiv、ldiv、fdiv、ddiv
- 求余指令：irem、lrem、frem、drem
- 取反指令：ineg、lneg、fneg、dneg
- 位移指令：ishl、ishr、iushr、lshl、lshr、lushr
- 按位或指令：ior、lor
- 按位与指令：iand、land
- 按位异或指令：ixor、lxor
- 局部变量自增指令：iinc
- 比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp

Java 虚拟机的指令集直接支持了在《Java 语言规范》中描述的各种对整数及浮点数操作（JSL3 §4.2.2，JSL3 §4.2.4）的语义。

Java 虚拟机没有明确规定整型数据溢出的情况，但是规定了在处理整型数据时，只有除法指令（idiv 和 ldiv）以及求余指令（irem 和 lrem）出现除数为零时会导致虚拟机抛出异常，如果发生了这种情况，虚拟机将会抛出 ArithmeitcException 异常。

Java 虚拟机在处理浮点数时，必须遵循 IEEE 754 规范中所规定行为限制。也就是说 Java 虚拟机要求完全支持 IEEE 754 中定义的非正规浮点数值（Denormalized Floating-Point Numbers，§2.3.2）和逐级下溢（Gradual Underflow）。这些特征将会使得某些数值算法处理起来变得更容易一些。

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

结果必须舍入为可被表示的最接近的精确值，如果有两种可表示的形式与该值一样接近，那将优先选择最低有效位为零的。这种舍入模式也是 IEEE 754 规范中的默认舍入模式，称为向最接近数舍入模式（§2.8.1）。

在把浮点数转换为整数时，Java 虚拟机使用 IEEE 754 标准中的向零舍入模式（§2.8.1），这种模式的舍入结果会导致数字被截断，所有小数部分的有效字节都会被丢弃掉。向零舍入模式将在目标数值类型中选择一个最接近，但是不大于原值的数字来作为最精确的舍入结果。

Java 虚拟机在处理浮点数运算时，不会抛出任何运行时异常（这里所讲的是 Java 的异常，请勿与 IEEE 754 规范中的浮点异常互相混淆），当一个操作产生溢出时，将会使用有符号的无穷大来表示，如果某个操作结果没有明确的数学定义的话，将会时候 NaN 值来表示。所有使用 NaN 值作为操作数的算术操作，结果都会返回 NaN。

在对 long 类型数值进行比较时，虚拟机采用带符号的比较方式，而对浮点数值进行比较时（dcmpg、dcmpl、fcmpg、fcmpl），虚拟机采用 IEEE 754 规范说定义的无信号比较（Nonsignaling Comparisons）方式。

2.11.4 类型转换指令

类型转换指令可以将两种 Java 虚拟机数值类型进行相互转换，这些转换操作一般用于实现用户代码的显式类型转换操作，或者用来处理 Java 虚拟机字节码指令集中指令非完全独立独立的问题（§2.11.1）。

Java 虚拟机直接支持（译者注：“直接支持”意味着转换时无需显式的转换指令）以下数值的宽化类型转换（Widening Numeric Conversions，小范围类型向大范围类型的安全转换）：

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

窄化类型转换（Narrowing Numeric Conversions）指令包括有：i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l 和 d2f。窄化类型转换可能会导致转换结果产生不同的正负号、不同的数量级，转换过程很可能会导致数值丢失精度。

在将 int 或 long 类型窄化转换为整数类型 T 的时候，转换过程仅仅是简单的丢弃除最低位 N 个字节以外的内容，N 是类型 T 的数据类型长度，这将可能导致转换结果与输入值有不同的正负

号（译者注：在高位字节符号位被丢弃了）。

在将一个浮点值转窄化转换为整数类型 T（T 限于 int 或 long 类型之一）的时候，将遵循以下转换规则：

- 如果浮点值是 NaN，那转换结果就是 int 或 long 类型的 0
- 否则，如果浮点值不是无穷大的话，浮点值使用 IEEE 754 的向零舍入模式（§2.8.1）取整，获得整数值 v，这时候可能有两种情况：
  - 如果 T 是 long 类型，并且转换结果在 long 类型的表示范围之内，那就转换为 long 类型数值 v
  - 如果 T 是 int 类型，并且转换结果在 int 类型的表示范围之内，那就转换为 int 类型数值 v
- 否则：
  - 如果转换结果 v 的值太小（包括足够小的负数以及负无穷大的情况），无法使用 T 类型表示的话，那转换结果取 int 或 long 类型所能表示的最小数字。
  - 如果转换结果 v 的值太大（包括足够大的正数以及正无穷大的情况），无法使用 T 类型表示的话，那转换结果取 int 或 long 类型所能表示的最大数字。

从 double 类型到 float 类型做窄化转换的过程与 IEEE 754 中定义的一致，通过 IEEE 754 向最接近数舍入模式（§2.8.1）舍入得到一个可以使用 float 类型表示的数字。如果转换结果的绝对值太小无法使用 float 来表示的话，将返回 float 类型的正负零。如果转换结果的绝对值太大无法使用 float 来表示的话，将返回 float 类型的正负无穷大，对于 double 类型的 NaN 值将就规定转换为 float 类型的 NaN 值。

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

2.11.5 对象创建与操作

虽然类实例和数组都是对象，但 Java 虚拟机对类实例和数组的创建与操作使用了不同的字节码指令：

- 创建类实例的指令：new

- 创建数组的指令：newarray，anewarray，multianewarray
- 访问类字段（static 字段，或者称为类变量）和实例字段（非 static 字段，或者成为实例变量）的指令：getfield、putfield、getstatic、putstatic
- 把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload
- 将一个操作数栈的值储存到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore
- 取数组长度的指令：arraylength
- 检查类实例类型的指令：instanceof、checkcast

2.11.6 操作数栈管理指令

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

2.11.7 控制转移指令

控制转移指令可以让 Java 虚拟机有条件或无条件地从指定指令而不是控制转移指令的下一条指令继续执行程序。控制转移指令包括有：

- 条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpeq、if_icmpne、if_icmplt, if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne。
- 复合条件分支：tableswitch、lookupswitch
- 无条件分支：goto、goto_w、jsr、jsr_w、ret

在 Java 虚拟机中有专门的指令集用来处理 int 和 reference 类型的条件分支比较操作，为了可以无需明显标识一个实体值是否 null，也有专门的指令用来检测 null 值（§2.4）。

boolean 类型、byte 类型、char 类型和 short 类型的条件分支比较操作，都使用 int 类型的比较指令来完成，而对于 long 类型、float 类型和 double 类型的条件分支比较操作，则

会先执行相应类型的比较运算指令（§2.11.3），运算指令会返回一个整形值到操作数栈中，随后再执行 int 类型的条件分支比较操作来完成整个分支跳转。由于各种类型的比较最终都会转化为 int 类型的比较操作，基于 int 类型比较的这种重要性，Java 虚拟机提供了非常丰富的 int 类型的条件分支指令。

所有 int 类型的条件分支转移指令进行的都是有符号的比较操作。

2.11.8 方法调用和返回指令

以下四条指令用于方法调用：

invokevirtual 指令用于调用对象的实例方法，根据对象的实际类型进行分派（虚方法分派），这也是 Java 语言中最常见的方法分派方式。

invokeinterface 指令用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。

invokespecial 指令用于调用一些需要特殊处理的实例方法，包括实例初始化方法（§2.9）、私有方法和父类方法。

invokestatic 指令用于调用类方法（static 方法）。

而方法返回指令则是根据返回值的类型区分的，包括有 ireturn（当返回值是 boolean、byte、char、short 和 int 类型时使用）、lreturn、freturn、dreturn 和 areturn，另外还有一条 return 指令供声明为 void 的方法、实例初始化方法、类和接口的类初始化方法使用。

2.11.9 抛出异常

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

2.11.10 同步

Java 虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构都是使

用管程（Monitor）来支持的。

方法级的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作（§2.11.8）之中。虚拟机可以从方法常量池中的方法表结构（method_info Structure，§4.6）中的 ACC_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有管程，然后再执行方法，最后再方法完成（无论是正常完成还是非正常完成）时释放管程。在方法执行期间，执行线程持有了管程，其他任何线程都无法再获得同一个管程。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放。

同步一段指令集序列通常是由 Java 语言中的 synchronized 块来表示的，Java 虚拟机的指令集中有 monitorenter 和 monitorexit 两条指令来支持 synchronized 关键字的语义，正确实现 synchronized 关键字需要编译器与 Java 虚拟机两者协作支持（读者可参见§3.14 中关于同步的描述）。

结构化锁定（Structured Locking）是指在方法调用期间每一个管程退出都与前面的管程进入相匹配的情形。因为无法保证所有提交给 Java 虚拟机执行的代码都满足结构化锁定，所以 Java 虚拟机允许（但不强制要求）通过以下两条规则来保证结构化锁定成立。假设 T 代表一条线程，M 代表一个管程的话：

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

请注意，在同步方法调用时自动持有和释放管程的过程也被认为是在方法调用期间发生。

2.12 类库

Java 虚拟机必须对不同平台下 Java 类库的实现提供充分的支持，因为其中有一些类库如果没有 Java 虚拟机的支持的话是根本无法实现的。

可能需要 Java 虚拟机特殊支持的类库包括有：

- 反射，譬如在 java.lang.reflect 包中的各个类和 java.lang.Class 类

- 类和接口的加载和创建，最显而易见的例子就是 java.lang.ClassLoader 类
- 类和接口的链接和初始化，上一点的例子也适用于这点
- 安全，譬如在 java.security 包中的各个类和 java.lang.SecurityManager 等其他类
- 多线程，譬如 java.lang.Thread 类
- 弱引用，譬如在 java.lang.ref 包中的各个类

上面列举的几点意在简单说明一下而不是详细介绍这些类库，介绍这些类库和功能的详细信息已经超出了本书的范围，如果读者想了解这些类库，阅读 Java 平台类库的说明书可以获得想要的信息。

2.13 公有设计，私有实现

到此为止，本书简单描绘了 Java 虚拟机应有的共同外观：Class 文件格式以及字节码指令集等。这些内容与硬件、操作系统和 Java 虚拟机的独立实现都是密切相关的，虚拟机实现者可能更愿意把它们看做是程序在各种 Java 平台实现之间互相安全地交互的手段，而多于一张需要精确跟随的计划蓝图。

理解公有设计与私有实现之间的分界线是非常有必要的，Java 虚拟机实现必须能够读取 Class 文件并精确实现包含在其中的 Java 虚拟机代码的语义。拿着《Java 虚拟机规范》一成不变地逐字实现其中要求的内容当然是一种可行的途径，但实现者在本规范约束下对具体实现做出修改和优化也是完全可行，并且也推荐这样做的。只要优化后 Class 文件依然可以被正确读取，并且包含在其中的语义能得到保持，那实现者就可以选择任何方式去实现这些语义，虚拟机后台如何处理 Class 文件完全是实现者自己的事情，只要它在外部接口上看起来与规范描述的一致即可[①]。

实现者可以使用这种伸缩性来让 Java 虚拟机获得更高的性能、更低的内存消耗或者更好的可移植性，选择哪种特性取决于 Java 虚拟机实现的目标和关注点是什么，虚拟机实现的方式主要有以下两种：

[①] 这里多少存在一些例外：譬如调试器（Debuggers）、性能监视器（Profilers）和即时代码生成器（Just-In-Time Code Generator）等都可能需要访问一些通常被认为是“虚拟机后台”的元素。Oracle 与其他 Java 虚拟机实现者以及工具提供商一起开发这类 Java 虚拟机工具的通用接口，令这些接口可以在整个行业中通用。

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

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

第 3 章 为 JAVA 虚拟机编译

Java 虚拟机是为了支持 Java 语言而的设计的。Oracle 的 JDK 包括两部分内容：一部分是将 Java 源代码编译成 Java 虚拟机的指令集的编译器，另一部分是用于 Java 虚拟机的运行时环境。理解编译器是如何与 Java 虚拟机协同工作的，对编译器开发人员来说很有好处，同样也有助于理解 Java 虚拟机本身。

请注意：术语“编译器”在某些上下文场景中专指把 Java 虚拟机的指令集转换为特定 CPU 指令集的翻译器。譬如即时代码生成器（Just-In-Time/JIT Code Generator）就是一种在 Class 文件中的代码被 Java 虚拟机代码加载后，生成与平台相关的特定指令的编译器。但是在本章中讨论的编译器将不考虑这类代码生成的问题，只会涉及到从使用 Java 语言编写的源代码编译为 Java 虚拟机指令集的编译器。

3.1 示例的格式说明

本章节的示例主要包括有源文件和 Java 虚拟机代码注解列表（Annotated Listings），其中，Java 虚拟机的代码注解列表是由 Oracle 的 1.0.2 版本的 JDK 的 javac 编译器生成。Java 虚拟机代码将使用 Oracle 的 javap 工具所生成的非正式的“虚拟机汇编语言（Virtual Machine Assembly Language）”格式来描述。读者可以自行使用 javap 命令去查看更多已编译方法的例子。

如果读者阅读过汇编代码，应该很熟悉示例中的格式。所有指令的格式如下：

```
<index> <opcode> [<operand1> [<operand2>...]] [<comment>]
```

<index>是 code[]数组中的指令的操作码的索引，此处的 code[]数组就是存储当前方法的 Java 虚拟机字节码的 Code 属性中的 code[]数组（§4.7.3）。也可以认为<index>是相对于方法起始处的字节偏移量。<opcode>为指令的操作码的助记符号，<operandN>是指令的操作数，一条指令可以有 0 至多个操作数。<comment>为行尾的语法注释，譬如：

```
8 bipush 100 // Push int constant 100
```

注释中的某些部分由 javap 自动加入，其余部分由作者手工添加。每条指令之前的<index>可以作为控制转移指令（Control Transfer Instruction）的跳转目标。譬如：goto 8 指

令表示跳转到索引为 8 的指令上继续执行。需要注意的是，Java 虚拟机的控制转移指令的实际操作数是在当前指令的操作码集合中的地址偏移量，但这些操作数会被 javap 工具（以及在本章的内容中）按照更容易被人所阅读的方式来显示。

在每一行中，在表示运行时常量池索引的操作数前，会井号（'#'）开头，在指令后的注释中，会带有这个操作数的描述，譬如：

```
10 ldc #1 // Push float constant 100.0
```

和

```
9 invokevirtual #4 // Method Example.addTwo(II)I
```

本章节主要目的是描述虚拟机的编译过程，我们将忽略一些诸如操作数容量等的细节问题。

3.2 常量、局部变量的使用和控制结构[①]

Java 虚拟机代码中展示了 Java 虚拟机设计和使用所遵循的一些通用特性。从第一个例子我们就可以感受到许多这类特性，示例如下：

spin()是一个很简单的方法，它进行了 100 次空循环：

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ; // Loop body is empty
    }
}
```

编译后代码如下：

```
Method void spin()
 0 iconst_0 // Push int constant 0
 1 istore_1 // Store into local variable 1 (i=0)
 2 goto 8 // First time through don't increment
 5 iinc 1 1 // Increment local variable 1 by 1 (i++)
 8 iload_1 // Push local variable 1 (i)
 9 bipush 100 // Push int constant 100
 11 if_icmplt 5 // Compare and loop if less than (i < 100)
```

[①] 译者注：控制结构（Control Constructs）是指控制程序执行路径的语句体。譬如 for、while 等循环、条件分支等。

```
14 return // Return void when done
```

Java 虚拟机是基于栈架构设计的，它的大多数操作是从当前栈帧的操作数栈取出 1 个或多个操作数，或将结果压入操作数栈中。每调用一个方法，都会创建一个新的栈帧，并创建对应方法所需的操作数栈和局部变量表（参见§2.6 “栈帧”）。每条线程在运行时的任意时刻，都会包含若干个由不同方法嵌套调用而产生的栈帧，当然也包括了若干个栈帧内部的操作数栈，但是只有当前栈帧中的操作数栈才是活动的。

Java 虚拟机指令集合使用不同的字节码来区分不同的操作数类型，用于操作各种类型的变量。在 spin()方法中，只有对 int 类型的运算。因此在编译代码里面，所采用的对类型数据进行操作的指令（iconst_0、istore_1、iinc、iload_1、if_icmplt）都是针对 int 型的。

spin()方法中，0 和 100 两个常量分别使用了两条不同的指令压入操作数栈。对于 0 采用了 iconst_0 指令，它属于 iconst_<i>指令族。而对于 100 采用 bipush 指令，这条指令会获取它的立即操作数（Immediate Operand）[①]压入到操作数栈中。

Java 虚拟机经常利用操作码隐式包含某些操作数，譬如指令 iconst_<i> 中的 i 表示的 int 常量 -1、0、1、2、3、4、5。iconst_0 表示把 int 型的 0 值压入操作数栈，这样，iconst_0 不需要专门为入栈操作保存一个立即操作数的值，这样也避免了操作数的读取和解析（Decode）的步骤。在本例中，把 0 压入操作数栈这个操作的指令由 iconst_0 改为 bipush 0 也能获取正确的结果，但是 spin()的编译代码会因此额外增加 1 个字节的长度。简单实现的虚拟机可能在每次循环时消耗更多的时间用于获取和解析这个操作数。因此使用隐式操作数可让编译后的代码更简洁，更高效。

spin()方法中，int 型的 i 被保存在编号为 1 的局部变量中[②]。因为大部分 Java 虚拟机指令操作的值是来源于操作数栈中出栈的值，而不是操作局部变量本身，所以在 Java 虚拟机的已编译代码中，在局部变量表和操作数栈之间传输值的指令很常见的，在指令集里，这类操作也有特殊地支持。spin()方法的第一个局部变量的传输过程由 istore_1 和 iload_1 指令完成，这两条指令都隐式指明了是对于第一个局部变量进行操作。istore_1 指令作用是从操作数栈中弹出一个 int 型的值，并保存在第一个局部变量中。iload_1 指令作用是将第一个局部变量的值压入操作数栈。

[①] 译者注：在指令流中直接跟随在指令后面，而不是在操作数栈中的操作数称为立即操作数（也有直译为立即数或直接操作数）。

[②] 译者注：读者需要注意到，局部变量的编号从 0 开始，。

如何使用（以及重用）局部变量由编译器的开发者所决定。尤其是对于 load 和 store 指令，编译器的开发者应尽可能多的重用局部变量，这样会使得代码更高效，更简洁，占用的内存（当前栈帧的空间）更少。

某些对局部变量频繁进行的操作在 Java 虚拟机中也有特别地支持。iinc 指令的作用是对局部变量加上一个长度为 1 字节有符号的递增量。譬如 spin()方法中 iinc 指令，它的作用是对第一个局部变量（第一个操作数）的值增加 1（第二个操作数）。iinc 指令很适合实现循环结构。

spin()方法的循环部分由这些指令完成：

```
 5 iinc 1 1 // Increment local 1 by 1 (i++)
 8 iload_1 // Push local variable 1 (i)
 9 bipush 100 // Push int constant 100
 11 if_icmplt 5 // Compare and loop if less than (i < 100)
```

bipush 指令将 int 型的 100 压入操作数栈，然后 if_icmplt 指令将 100 从操作数栈中弹出值并与 i 进行比较，如果满足条件（即 i 的值小于 100），将转移到索引为 5 的指令继续执行，开始下一轮循环的迭代。否则，程序将执行 if_icmplt 的下一条指令，即 return 指令。

如果在 spin()例子的中循环的计数器使用了非 int 类型，那么编译代码也要有调整。譬如在 spin 例子中使用 double 型取代 int，则：

```
void dspin() {
     double i;
     for (i = 0.0; i < 100.0; i++) {
            ; // Loop body is empty
     }
}
```

编译后代码如下：

```
Method void dspin()
  0 dconst_0 // Push double constant 0.0
  1 dstore_1 // Store into local variables 1 and 2
  2 goto 9 // First time through don't increment
  5 dload_1 // Push local variables 1 and 2
  6 dconst_1 // Push double constant 1.0
  7 dadd // Add; there is no dinc instruction
  8 dstore_1 // Store result in local variables 1 and 2
  9 dload_1 // Push local variables 1 and 2
  10 ldc2_w #4 // Push double constant 100.0
  13 dcmpg // There is no if_dcmplt instruction
  14 iflt 5 // Compare and loop if less than (i < 100.0)
```

```
 17 return // Return void when done
```

操作数据类型的指令已变成针对 double 类型值的指令了。（ldc2_w 指令将在本章节晚些时候讨论）。

前文提到过 double 类型的值占用两个局部变量的空间，但是只能通过两个局部变量空间中索引较小的一个进行访问（这种情况对于 long 类型也一样）。譬如下面例子展示了 double 类型值的访问：

```
double doubleLocals(double d1, double d2) {
    return d1 + d2;
}
```

编译后代码如下：

```
Method double doubleLocals(double,double)
 0 dload_1 // First argument in local variables 1 and 2
 1 dload_3 // Second argument in local variables 3 and 4
 2 dadd
 3 dreturn
```

注意到局部变量表中使用了一对局部变量来存储 doubleLocals()方法中的 double 值，这对局部变量不能被分开来进行单个操作。

Java 虚拟机中，操作码长度为 1 个字节，这使得编译后的代码显得很紧凑。但是同样意味着 Java 虚拟机指令集必须保持一个较小的数量（小于 256 条，1 字节所能表示的范围）。作为妥协，Java 虚拟机无法对全部操作都一视同仁地提供所有数据类型的支持。换句话说，并非每条指令都是完全独立（Not Completely Orthogonal）支持一种类型的（参见表 2.2 “Java 虚拟机指令集所支持的数据类型”）。

举例来说，在 spin()方法的 for 循环语句中，对于 int 型值的判断可以统一用 if_icmplt 指令实现；但是，在 Java 虚拟机指令集合中，对于 double 类型的值的没有这样的指令。所以，在 dspin()方法中，对于 double 类型的值的操作就必须在 dcmpg 指令后面再联合 iflt 指令来实现。

Java 虚拟机支持下，对 int 类型的数据的大部分操作可以直接进行。这在一定程度上是考虑到了 Java 虚拟机操作数栈和局部变量表的实现效率。当然也有考虑到了大多数程序都会对 int 型数据进行频繁操作的原因。Java 虚拟机对其余的数据类型的直接操作的支持较少，在 Java 虚拟机指令集中，没有对 byte，char 和 short 类型变量的 store、load 和 add 等指令。譬如，用 short 类型来实现 spin()中的循环时：

```
void sspin() {
     short i;
     for (i = 0; i < 100; i++) {
            ; // Loop body is empty
     }
}
```

Java 虚拟机编译时要将其他可安全转化为 int 类型的数据转换为 int 类型进行操作。在将 short 类型值转换为 int 类型值时，可以保证 short 类型值操作后结果一定在 int 类型的精度范围之内，因此 sspin()的编译后代码如下：

```
Method void sspin()
  0 iconst_0
  1 istore_1
  2 goto 10
  5 iload_1 // The short is treated as though 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
```

在 Java 虚拟机中，缺乏对 byte、char 和 short 类型数据直接操作的支持所带来的问题并不大，因为这些类型的值都在编译过程中就自动被转换为 int 类型（byte 和 short 带符号扩展为 int 类型，char 零位扩展为 int 类型）。因此对于 byte、char 和 short 类型的数据均可以用 int 的指令操作。唯一额外的代价是将它们长度扩展至 4 字节。

Java 虚拟机对于 long 和浮点类型（float 和 double）提供了中等程度的支持，比起 int 类型数据所支持的操作，它们仅缺少了条件转移指令部分，其他操作都与 int 类型具有相同程度的支持。

3.3 算术运算

Java 虚拟机通常基于操作数栈来进行算术运算（只有 iinc 指令例外，它直接对局部变量进行自增操作），譬如下面的 align2grain()方法，它的作用是将 int 值对齐到 2 的指定幂次：

```
int align2grain(int i, int grain) {
    return ((i + grain-1) & ~(grain-1));
}
```

算术运算使用到的操作数都是从操作数栈中弹出的，运算结果被压回操作数栈中。在内部运算时，中间运算（Arithmetic Subcomputations）的结果可以被当作操作数使用。譬如~(grain-1)的值就是被这样使用的：

```
5 iload_2 // Push grain
6 iconst_1 // Push int constant 1
7 isub // Subtract; push result
8 iconst_m1 // Push int constant -1
9 ixor // Do XOR; push result
```

首先，grain-1 的结果由第 2 个局部变量和立即操作数 int 数值 1 的计算得出。参与运算的操作数从操作数栈中弹出，然后它们将被改变，最后再入栈到操作数栈之中。这里被改变是单个操作数的算术指令 ixor 运算的结果（因为~x == -1^x）。类似地，ixor 指令的结果是接下来将作为 iand 指令的操作数被使用。

整个方法的编译代码如下：

```
Method 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
```

3.4 访问运行时常量池

很多数值常量，以及对象、字段和方法，都是通过当前类的运行时常量池进行访问。对象的访问将在稍后的§3.8 中讨论。类型为 int、long、float 和 double 的数据，以及表示 String

实例的引用类型数据的访问将由 ldc、ldc_w 和 ldc2_w 指令实现。

ldc 和 ldc_w 指令用于访问运行时常量池中的对象，包括 String 实例，但不包括 double 和 long 类型的值。当使用的运行时常量池的项的数目过多时（多于 256 个，1 个字节能表示的范围），需要使用 ldc_w 指令取代 ldc 指令来访问常量池。ldc2_w 指令用于访问类型为 double 和 long 的运行时常量池项，这条指令没有非宽索引的版本（即没有 ldc2 指令）。

对于整型常量，包括 byte、char、short 和 int，如前面 § 3.2 节所描述，将编译到代码之中，使用 bipush、sipush 和 iconst_<i>指令进行访问。某些浮点常量也可以编译进代码，使用 fconst_<f>和 dconst_<d>指令进行访问。

通过以上描述，编译的规则已经基本解释清楚了。下面这个例子将这些规则汇总了起来：

```
void useManyNumeric() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
    double d = 2.2;
    ...do some calculations...
}
```

编译后代码如下：

```
Method void useManyNumeric()
0 bipush 100 // Push a small int with bipush
2 istore_1
3 ldc #1 // Push int constant 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 // Push long 0xffffffff (that is, an int -1); any
// long constant value can be pushed using ldc2_w
11 lstore 5
13 ldc2_w #8 // Push double constant 2.200000; uncommon
// double values are also pushed using ldc2_w
16 dstore 7
...do those calculations...
```

3.5 更多的控制结构示例

在§3.2 小节中展示了一些控制结构是如何编译的。在 Java 语言中还有很多其他的控制结构（if-then-else、do、while、break 以及 continue）也有特定的编译规则。本规范将在§3.10“编译 switch 语句块”、§3.12“抛出异常和处理异常”和§3.13“编译 finally 语句块”中分别讨论关于 switch 语句块，异常和 finally 语句块的编译规则。

下面这个例子是关于 while 的循环的编译规则，Java 虚拟机会根据数据的变化而生成不同的条件跳转语句，通常情况下，Java 虚拟机对 int 类型数据提供的支持最为完善。

```
void whileInt() {
    int i = 0;
    while (i < 100) {
            i++;
    }
}
```

编译后代码如下：

```
Method void whileInt()
0 iconst_0
1 istore_1
2 goto 8
5 iinc 1 1
8 iload_1
9 bipush 100
11 if_icmplt 5
14 return
```

注意到 while 语句的条件判断（由 if_icmplt 指令实现）在 Java 虚拟机编译代码中的循环的最底部，这和§3.2 节中 spin()方法的条件判断位置一致。原本处于循环最底部、在迭代结束时执行的条件测试指令被一条 goto 指令强制跳转到在循环的第一次迭代之前执行。这样如果条件判断失败，则不会在进入循环体，其他额外的指令就不会执行了。不过 while 循环通常都使用在期望循环体会被执行的场景之中（而不是用来当作 if 语句使用）。在接下来的循环迭代中，由于条件判断的操作被置于循环体的底部，这样相当于在执行循环体时节省了一条 Java 虚拟机指令。如果将条件判断的操作放在循环体的顶部，那循环体就必须在尾部额外增加一条 goto 指令用于循环体结束时跳转回顶部。

虚拟机对各种数据类型的控制结构采用了相似的方式编译，只是根据不同数据类型使用不同的

指令来访问。这么做多少会使编译代码效率降低，因为这样可能需要更多的 Java 虚拟机指令来实现，譬如：

```
void whileDouble() {
    double i = 0.0;
    while (i < 100.1) {
            i++;
    }
}
```

编译后代码如下：

```
Method 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 // Push double constant 100.1
13 dcmpg // To do the compare and branch we have to use...
14 iflt 5 // ...two instructions
17 return
```

每个浮点型数据都有两条比较指令：对于 float 类型是 fcmpl 和 fcmpg 指令，对于 double 是 dcmpl 和 dcmpg 指令。这些指令语义相似，仅仅在对待 NaN 变量时有所区别。NaN 是无序的，所以如果有其中一个操作数为 NaN，则所有浮点型的比较指令都失败[①]。无论是比较操作是否会因为遇到 NaN 值而失败，编译器都会根据不同的操作数类型来选择不同的比较指令，譬如：

```
int lessThan100(double d) {
    if (d < 100.0) {
            return 1;
    } else {
            return -1;
    }
}
```

编译后代码如下：

[①] 译者注：请注意，“比较失败（Comparisons Fail）”的意思是比较指令返回“fail（对于 fcmpl 为-1，而 fcmpg 为 1）”的结果到操作数栈，而不是抛出异常。在 Java 虚拟机指令集中，所有的算术比较指令都不会抛出异常。

```
Method int lessThan100(double)
0 dload_1
1 ldc2_w #4 // Push double constant 100.0
4 dcmpg // Push 1 if d is NaN or d > 100.0;
// push 0 if d == 100.0
5 ifge 10 // Branch on 0 or 1
8 iconst_1
9 ireturn
10 iconst_m1
11 ireturn
```

如果 d 不是 NaN 并且小于 100.0，dcmpg 指令将 int 值-1 压入操作数栈，ifge 指令就不会被执行。如果 d 大于 100.0 或者是 NaN，dcmpg 指令将 int 值 1 压入操作数栈，ifge 指令将执行。如果 d 等于 100.0，dcmpg 指令将 int 值 0 压入操作数栈，ifge 指令将执行。

如果比较逻辑相反，dcmpl 指令可以达到相同的效果，譬如：

```
int greaterThan100(double d) {
    if (d > 100.0) {
            return 1;
    } else {
            return -1;
    }
}
```

编译后代码如下：

```
Method int greaterThan100(double)
0 dload_1
1 ldc2_w #4 // Push double constant 100.0
4 dcmpl // Push -1 if d is Nan or d < 100.0;
// push 0 if d == 100.0
5 ifle 10 // Branch on 0 or -1
8 iconst_1
9 ireturn
10 iconst_m1
11 ireturn
```

再来分析一次，无论是否因参数 d 传入了 NaN 值而导致的比较失败，dcmpl 指令都会向操作数栈压入一个 int 型值，使程序进入 ifle 指令的分支。如果没有 dcmpl 和 dcmpg 指令，那么对于实例中的方法，编译器不得不因 NaN 值而做更多处理。

3.6 接收参数

如果传递了 n 个参数给某个实例方法，则当前栈帧会按照约定的顺序接收这些参数，将它们保存为方法的第 1 个至第 n 个局部变量之中。譬如：

```
int addTwo(int i, int j) {
    return i + j;
}
```

编译后代码如下：

```
Method int addTwo(int,int)
0 iload_1 // Push value of local variable 1 (i)
1 iload_2 // Push value of local variable 2 (j)
2 iadd // Add; leave int result on operand stack
3 ireturn // Return int result
```

按照约定，实例方法需要传递一个自身实例的引用作为第 0 个局部变量。在 Java 语言中自身实例可以通过 this 关键字来访问。

类（static）方法不需要传递实例引用，所以它们不需要使用第 0 个局部变量来保存 this 关键字。如果 addTwo()是类方法，那么接收的参数和之前示例相比略有不同：

```
static int addTwoStatic(int i, int j) {
    return i + j;
}
```

编译后代码如下：

```
Method int addTwoStatic(int,int)
0 iload_0
1 iload_1
2 iadd
3 ireturn
```

两段代码唯一的区别是，后者方法保存参数到局部变量表时，是从编号为 0 的局部变量开始而不是 1。

3.7 方法调用

对普通实例方法调用是在运行时根据对象类型进行分派的(相当于在 C++中所说的“虚方法”)，

这类方法通过调用 invokevirtual 指令实现，每条 invokevirtual 指令都会带有一个表示索引的参数，运行时常量池在该索引处的项为某个方法的符号引用，这个符号引用可以提供方法所在对象的类型的内部二进制名称、方法名称和方法描述符（§4.3.3）。下面这个例子中定义了一个实例方法 add12and13()来调用前面的 addTwo()方法，如下：

```
    int add12and13() {
        return addTwo(12, 13);
    }
```

编译后代码如下：

```
    Method int add12and13()
    0 aload_0 // Push local variable 0 (this)
    1 bipush 12 // Push int constant 12
    3 bipush 13 // Push int constant 13
    5 invokevirtual #4 // Method Example.addtwo(II)I
    8 ireturn // Return int on top of operand stack; it is  the int
result of addTwo()
```

方法调用过程的第一步是将当前实例的自身引用“this”压入到操作数栈中。传递给方法的参数，int 值 12 和 13 随后入栈。当调用 addTwo()方法时，Java 虚拟机会创建一个新的栈帧，传递给 addTwo()方法的参数作为新的帧的对应局部变量的初始值。即 this 和两个传递给的 addTwo()方法的参数 12 和 13 被作为 addTwo()方法栈帧的第 0、1、2 个局部变量。

最后，当 addTwo()方法执行结束、方法返回时，int 型的返回值被压入方法调用者的栈帧的操作数栈，即 add12and13()方法的操作数栈中。这样 addTwo()方法的返回值就被放置在调用者 add12and13()方法的代码可以立刻使用到的地方。

add12and13()方法的返回过程由 add12and13()方法中的 ireturn 指令实现。由于调用的 addTwo()方法返回的 int 值被压入当前操作数栈的栈顶，ireturn 指令将会把当前操作数栈的栈顶值（此处就是 addTwo()的返回值）压入调用 add12and13()方法的操作数栈。然后跳转至调用者调用方法的下一条指令继续执行，并将调用者的栈帧重新设为当前栈帧。Java 虚拟机对不同数据类型（包括声明为 void，没有返回值的方法）的返回值提供了不同的方法返回指令，各种不同返回值类型的方法都使用这一组返回指令来返回。

invokevirtual 指令操作数（在上面示例中的运行时常量池索引#4）不是 Class 实例中方法指令的偏移量。编译器并不需要了解 Class 实例的内部布局，它只需要产生方法的符号引用并保存于运行时常量池即可，这些运行时常量池项将会在执行时转换成调用方法的实际地址。Java 虚拟机指令集中其他指令在访问 Class 实例时也采用相同的方式。

如果上个例子中，调用的实例方法 addTwo()变成类（static）方法的话，编译代码会有略微变化，如下：

```
int add12and13() {
    return addTwoStatic(12, 13);
}
```

编译代码中使用了另一个 Java 虚拟机调用指令 invokestatic：

```
Method int add12and13()
0 bipush 12
2 bipush 13
4 invokestatic #3 // Method Example.addTwoStatic(II)I
7 ireturn
```

类方法和实例方法的调用的编译代码很类似，两者的区别仅仅是实例方法需要调用者传递 this 参数而类方法则不用。所以在两种方法的局部变量表中，序号为 0（第一个）的局部变量会有所区别（参见§3.6 “接收参数”）。invokestatic 指令用于调用类方法。

invokespecial 指令用于调用实例初始化方法（参见§3.8 “使用类实例”），它也可以用来调用父类方法和私有方法。譬如下面例子中的 Near 和 Far 两个类：

```
class Near {
    int it;
    public int getItNear() {
            return getIt();
    }
    private int getIt() {
            return it;
    }
}

class Far extends Near {
    int getItFar() {
            return super.getItNear();
    }
}
```

调用类 Near 的方法 getItNear()（调用私有方法）被编译为：

```
Method int getItNear()
0 aload_0
1 invokespecial #5 // Method Near.getIt()I
4 ireturn
```

调用类 Far 的方法 getItFar ()（调用父类方法）被编译为：

```
Method int getItFar()
0 aload_0
1 invokespecial #4 // Method Near.getItNear()I
4 ireturn
```

请注意，所有使用 invokespecial 指令调用的方法都需要 this 作为第一个参数，保存在第一个局部变量之中。

如果编译器要调用某个方法，必须先产生这个方法描述符，描述符中包含了方法实际参数和返回类型。编译器在方法调用时不会处理参数的类型转换问题，只是简单地将参数的压入操作数栈，且不改变其类型。通常，编译器会先把一个方法所在的对象的引用压入操作数栈，方法参数则按顺序跟随这个对象之后入栈。编译器在生成 invokevirtual 指令时，也会生成这条指令所引用的描述符，这个描述符提供了方法参数和返回值的信息。作为方法解析（§5.4.3.3）时的一个特殊处理过程，一个用于调用 java.lang.invoke.MethodHandle 的 invoke()或者 invokeExact()方法的 invokevirtual 指令会提供一个方法描述符，这个方法描述符符合语法规则，并且在描述符中确定的类型信息将会被解析。

3.8 使用类实例

Java 虚拟机类实例通过 Java 虚拟机 new 指令创建。之前提到过，在 Java 虚拟机层面，构造函数将会以一个编译器提供的以<init>命名的方法出现。这个特殊的名字的方法也被称作实例初始化方法（§2.9）。一个类可以有多个构造函数，对应地也就会有多个实例初始化方法。一旦类实例被创建，那么这个实例包含的所有实例变量，除了在本身定义的以及父类中所定义的，都将被赋予默认初始值，接着，新对象的实例初始化方法将会被调用。譬如下面示例：

```
Object create() {
    return new Object();
}
```

编译后代码如下：

```
Method java.lang.Object create()
  0 new #1 // Class java.lang.Object
  3 dup
  4 invokespecial #4 // Method java.lang.Object.<init>()V
```

```
  7 areturn
```

在参数传递和方法返回时，类实例（作为 reference 类型）与普通的数值类型没有太大区别，reference 类型也有它自己类型专有的 Java 虚拟机指令，譬如：

```
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;
    }
}
```

编译后代码如下：

```
Method 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

Method MyObj silly(MyObj)
0 aload_1
1 ifnull 6
4 aload_1
5 areturn
6 aload_1
7 areturn
```

类实例的字段（实例变量）将使用 getfield 和 putfield 指令进行访问，假设 i 是一个 int 型的实例变量，且方法 getIt() 和 setIt() 的定义如下：

```
void setIt(int value) {
    i = value;
```

```
}
int getIt() {
    return i;
}
```

编译后代码如下：

```
Method 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
```

无论方法调用指令的操作数，还是 putfield、getfield 指令的操作数（上面例子中运行时常量池索引#4）都并非类实例中的地址偏移量。编译器会为这些字段生成符号引用，保存在运行时常量池之中。这些运行时常量池项会在解析阶段转换为引用对象中真实的字段位置。

3.9 数组

在 Java 虚拟机中，数组也使用对象来表示。数组由专门的指令集来创建和操作。newarray 指令用于创建元素类型为数值类型的数组。譬如：

```
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

编译后代码如下：

```
Method void createBuffer()
0 bipush 100 // Push int constant 100 (bufsz)
2 istore_2 // Store bufsz in local variable 2
3 bipush 12 // Push int constant 12 (value)
5 istore_3 // Store value in local variable 3
```

```
6 iload_2 // Push bufsz...
7 newarray int // ...and create new array of int of that length
9 astore_1 // Store new array in buffer
10 aload_1 // Push buffer
11 bipush 10 // Push int constant 10
13 iload_3 // Push value
14 iastore // Store value at buffer[10]
15 aload_1 // Push buffer
16 bipush 11 // Push int constant 11
18 iaload // Push value at buffer[11]...
19 istore_3 // ...and store it in value
20 return
```

anewarray 指令用于创建元素为引用类型的一维数组。譬如：

```
void createThreadArray() {
    Thread threads[];
    int count = 10;
    threads = new Thread[count];
    threads[0] = new Thread();
}
```

编译后代码如下：

```
Method void createThreadArray()
0 bipush 10 // Push int constant 10
2 istore_2 // 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 // Push value of threads
9 iconst_0 // Push int constant 0
10 new #1 // Create instance of class Thread
13 dup // Make duplicate reference...
14 invokespecial #5 // ...to pass to instance initialization method
// Method java.lang.Thread.<init>()V
17 aastore // Store new Thread in array at 0
18 return
```

anewarray 指令也可以用于创建多维数组的第一维。不过我们也可以选择采用 multianewarray 指令一次性创建多维数组。譬如三维数组：

```
int[][][] create3DArray() {
    int grid[][][];
    grid = new int[10][5][];
```

```
        return grid;
    }
```

编译后代码如下：

```
    Method int create3DArray()[][][]
    0 bipush 10               // Push int 10 (dimension one)
    2 iconst_5                // Push int 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是运行时常量池索引，它表示将要被创建的数组的成员类型。第二个操作数是需要创建的数组的实际维数。multianewarray 指令可以用于创建所有类型的多维数组，譬如 create3DArray 中展示的。注意，多维数组也只是一个对象，所以使用 aload_1 指令加载，使用 areturn 指令返回，更多关于数组类的命名信息在 § 2.9 章节中讨论。

所有的数组都有一个与之关联的长度属性，通过 arraylength 指令访问。

3.10 编译 switch 语句

编译器会使用 tableswitch 和 lookupswitch 指令来生成 switch 语句的编译代码。tableswitch 指令用于表示 switch 结构中的 case 语句块，可以高效地从索引表中确定 case 语句块的分支偏移量。当 switch 语句中提供的条件值不能从索引表中确定任何一个 case 语句块的分支偏移量时， default 分支将起作用。譬如：

```
    int chooseNear(int i) {
        switch (i) {
                case 0: return 0;
                case 1: return 1;
                case 2: return 2;
                default: return -1;
        }
    }
```

编译后代码如下：

```
Method int chooseNear(int)
0 iload_1 // Push local variable 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 constant 0...
29 ireturn // ...and return it
30 iconst_1 // i was 1; push int constant 1...
31 ireturn // ...and return it
32 iconst_2 // i was 2; push int constant 2...
33 ireturn // ...and return it
34 iconst_m1 // otherwise push int constant -1...
35 ireturn // ...and return it
```

Java 虚拟机的 tableswitch 和 lookupswitch 指令都只能支持 int 类型的条件值。选择支持 int 类型是因为 byte、char 和 short 类型的值都会被隐式展为 int 型。如果 chooseNear() 方法中使用 short 类型作为条件值，那编译出来的代码中与使用 int 类型时是完全相同的。如果使用其他数值类型的条件值，那就必须窄化转换成 int 类型。

当 switch 语句中的 case 分支的条件值比较稀疏时，tableswitch 指令的空间使用率偏低。这种情况下将使用 lookupswitch 指令来替代。lookupswitch 指令的索引表由 int 型的键值（来源于 case 语句块后面的数值）与对应的目标语句偏移量所构成。当 lookupswitch 指令执行时，switch 语句的条件值将和索引表中的 key 进行比较，如果某个 key 和条件值相符，那么将转移到这个 key 对应的分支偏移量继续继续执行，如果没有 key 值符合，执行将在 default 分支执行。譬如：

```
int chooseFar(int i) {
    switch (i) {
            case -100: return -1;
            case 0: return 0;
            case 100: return 1;
    default: return -1;
    }
}
```

编译后的代码如下，相比 chooseNear() 方法的编译代码，仅仅把 tableswitch 指令换成了 lookupswitch 指令：

```
Method 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 iconst_m1
43 ireturn
```

Java 虚拟机规定的 lookupswitch 指令的索引表必须根据 key 值排序，这样使用（如采用二分搜索）将会比直接使用线性扫描搜索来得更有效率。在从索引表确定分支偏移量的过程中，lookupswitch 指令是把条件值与不同的 key 的进行比较，而 tableswitch 指令则只需要索引值进行一次范围检查。因此，在如果不需要考虑空间效率时，tableswitch 指令相比 lookupswitch 指令有更高的执行效率。

3.11 使用操作数栈

Java 虚拟机为方便使用操作数栈，提供了的大量的不区分操作数栈数据类型的指令。这些指令都很常用，因为 Java 虚拟机是基于栈的虚拟机，大量操作是建立在操作数栈的基础之上的。譬如：

```
public long nextIndex() {
return index++;
}
private long index = 0;
```

编译后代码如下：

```
Method long nextIndex()
0 aload_0 // Push this
1 dup // Make a copy of it
2 getfield #4 // One of the copies of this is consumed
// pushing long field index,
```

```
// above the original this
5 dup2_x1  // The long on top of the operand stack is
// inserted into the operand stack below the
// original this
6 lconst_1 // Push long constant 1
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 operand stack, ready to be returned
```

注意，Java 虚拟机不允许作用于操作数栈的指令修改或者拆分那些不可拆分操作数（如存放 long 或 double 的操作数）。

3.12 抛出异常和处理异常

程序中使用 throw 关键字来抛出异常，它的编译过程很简单，譬如：

```
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
            throw new TestExc();
    }
}
```

编译后代码如下：

```
Method void cantBeZero(int)
0 iload_1 // Push argument 1 (i)
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
```

try-catch 结构的编译也同样简单，譬如：

```
void catchOne() {
    try {
            tryItOut();
    } catch (TestExc e) {
            handleExc(e);
    }
```

```
}
```

编译后代码如下：

```
Method 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 // Push this
7 aload_1 // Push thrown value
8 invokevirtual #5 // Invoke handler method:
// Example.handleExc(LTestExc;)V
11 return // Return after handling TestExc
Exception table:
From To Target Type
0    4   5     Class TestExc
```

仔细看可发现，try 语句块被编译后似乎没有生成任何指令，就像它没有出现过一样。

```
Method 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 语句块执行过程中没有异常抛出，程序那么就犹如没有使用 try 结构一样：在 tryItOut()调用 catchOne()方法后就返回了。

在 try 语句块之后，Java 虚拟机代码实现的一个 catch 语句如下：

```
6 aload_0 // Push this
7 aload_1 // Push thrown value
8 invokevirtual #5 // Invoke handler method:
// Example.handleExc(LTestExc;)V
11 return // Return after handling TestExc
Exception table:
From To Target Type
0    4   5    Class TestExc
```

在 catch 语句块里，调用 handleExc()方法的指令和正常的方法调用完全一样。不过，每个 catch 语句块的会使编译器在异常表中增加一个成员（即一个异常处理器，§2.10，§4.7.3）。catchOne()方法的异常表中有一个成员，这个成员对应 catchOne()方法 catch 语句块的一个可捕获的异常参数（本例中为 TestExc 的实例）。在 catchOne()的执行过程中，如果在它的编译代码第 0 至 4 句之间有 TestExc 异常实例被抛出，那么操作将转移至第 5 句继续执

行，即进入 catch 语句块的实现步骤。如果抛出的异常不是 TestExc 实例，那么 catchOne() 的 catch 语句块则不能捕获它，这个异常将被抛出给 catchOne()方法的调用者。

一个 try 结构中可包含多个 catch 语句块，譬如：

```
void catchTwo() {
    try {
            tryItOut();
    } catch (TestExc1 e) {
            handleExc(e);
    } catch (TestExc2 e) {
            handleExc(e);
    }
}
```

如果 try 语句包含有多个 catch 语句块，那么在编译代码中，多个 catch 语句块的内容将连续排列，在异常表中也会有对应的连续排列的成员，它们的排列的顺序和源码中的 catch 语句块出现的顺序一致。

```
Method 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 // Push this
7 aload_1 // Push thrown value
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 // Push this
14 aload_1 // Push thrown value
15 invokevirtual #7 // Invoke handler method:
// Example.handleExc(LTestExc2;)V
18 return // Return after handling TestExc2
Exception table:
From To Target Type
0   4   5     Class TestExc1
0   4   12    Class TestExc2
```

catchTwo()在执行时，如果 try 语句块中（编译代码的第 1 至 4 句）抛出了一个异常，这个异常可以被多个 catch 语句块捕获（即这个异常的实例是一个或多个 catch 语句块的参数），

则 Java 虚拟机将选择第一个（最上层）catch 语句块来处理这个异常。程序将转移至这个 catch 语句块对应的 Java 虚拟机代码块中继续执行。如果抛出的异常不是任何 catch 语句块的参数（即不能被捕获），那么 Java 虚拟机将把这个异常抛出给 catchTwo()方法的调用者，catchTwo()方法本身的所有 catch 语句块的编译代码都不会被执行。

在 try-catch 语句可以嵌套使用，编译后产生的编译代码和一个 try 语句对应多个 catch 语句的结构很相似，譬如：

```
void nestedCatch() {
    try {
            try {
                    tryItOut();
            } catch (TestExc1 e) {
                    handleExc1(e);
            }
    } catch (TestExc2 e) {
            handleExc2(e);
    }
}
```

编译后代码如下：

```
Method 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 // Push this
7 aload_1 // Push thrown value
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 // Push this
14 aload_1 // Push thrown value
15 invokevirtual #6 // Invoke handler method:
// Example.handleExc2(LTestExc2;)V
18 return // Return after handling TestExc2
Exception table:
From To Target Type
0   4   5    Class TestExc1
```

```
0    12   12    Class TestExc2
```

try-catch 语句的嵌套关系只体现在异常表之中，Java 虚拟机本身并不要求异常表中成员（§2.10）的顺序，但是编译器需要保证 try-catch 语句是有结构顺序的，编译器会根据 catch 语句在代码中的顺序对异常处理表进行排序，以保证在代码任何位置抛出的任何异常，都会被最接近异常抛出位置的、可处理该异常的 catch 语句块所处理。

例如，如果在编译代码第 1 句，即 tryItOut()方法执行过程中抛出一个 TestExc1 异常的实例，这个异常实例将被调用 handleExc1()方法的 catch 语句块处理。即使这个异常是发生在外层 catch 语句（捕获 TestExc2 异常的 catch 语句）的处理范围之内，并且外层 catch 语句也同样声明了能处理这类异常，但依然不会分配给外部的 catch 语句来处理，因为它在异常处理表中顺序在内部异常之后。

还有一个微妙之处需要注意，catch 语句块的处理范围包括 from 但不包括 to 所表示的偏移量本身（§4.7.3）。即 catch 语句块中，TestExc1 在异常表所对应的异常处理器并没有覆盖到字节偏移量为 4 处的返回指令。不过，TestExc2 在异常表中对应的异常处理器却覆盖了偏移量为 11 处的返回指令。因此在此场景下，如果在嵌套 catch 语句块中如果返回指令抛出了异常，将由外层的异常处理器进行处理。

3.13 编译 finally 语句块

前面提到过，本章所使用的编译代码由 Oracle 的 1.0.2 版本 JDK 的 javac 编译器生成，因此这里的 Class 文件版本必定低于 50.0，即将会使用 jsr 指令来编译 finally 语句块，更多内容请参见 4.10.2.5 节的“异常与 finally”[①]。

编译 try-finally 语句和编译 try-catch 语句基本相同。在代码执行完 try 语句之前（无论有没有抛出异常），finally 语句块中的内容都会被执行，譬如：

```
void tryFinally() {
    try {
            tryItOut();
    } finally {
```

[①] 译者注：很早之前（JDK 1.4.2 之前）的 Sun Javac 已经不再为 finally 语句生成 jsr 和 ret 指令了，而是改为在每个分支之后冗余代码的形式来实现 finally 语句，所以在这节开头作者需要特别说明。在版本号为 51.0（JDK 7 的 Class 文件）的 Class 文件中，甚至还明确禁止了指令流中出现 jsr、jsr_w 指令。

```
            wrapItUp();
        }
    }
```

编译后代码如下：

```
Method 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
16 invokevirtual #5 // Method Example.wrapItUp()V
19 ret 2 // Return from finally block
Exception table:
From To Target Type
0    4  8     any
```

有四种方式可以让程序退出 try 语句：1.语句块所有正常执行结束；2.通过 return 语句退出方法；3.通过 break 或 continue 语句退出循环；4.抛出异常。如果 tryItOut()正常结束（没有抛出异常）并返回，后面的 jsr 指令会使程序跳转到 finally 语句块继续执行。编译代码中的第 4 句的“jsr 14”表示的意思是“调用程序子片段（Subroutine Call)”，这条指令使程序跳转至第 14 句的 finally 语句块（finally 语句块的内容被编译为一段程序子片段）的实现代码之中。当 finally 语句块运行结束，使用“ret 2”指令将程序返回至 jsr 指令（即第 4 句）的下一句继续执行。

调用程序子片段的更多细节如下：在本例中，jsr 指令将其下一条指令的地址（即第 7 句的 return 指令）在程序跳转前压入操作数栈。程序跳转后使用 astore_2 指令将栈顶的元素（即 return 指令的地址）保存在第 2 个局部变量中。然后，执行 finally 语句块（在这个例子中，finally 语句块的内容包括 aload_0 和 invokevirtual 两条指令)。当 finally 语句块的代码正常地执行结束后，ret 指令使程序跳转至第 2 个局部变量所保存的地址（即 return 指令的地址）继续执行，至此 tryFinally()运行结束。

一个带有 finally 语句块的 try 语句，在编译时会生成一个特殊的异常处理器，这个异常处

理器可以捕获 try 语句中抛出的所有异常。当 tryItOut()抛出异常时，Java 虚拟机会在 tryFinally()方法则在异常处理器表中寻找一个合适的异常处理器。这个处理器被找到后，会转到异常处理器实现代码处（这个例子中为编译代码的第 8 句）继续执行。编译代码第 8 句的 astore_1 指令用于将抛出的异常保存在第 1 个局部变量中。接下来的 jsr 指令调用 finally 语句块的程序子片段。如果正常返回（finally 语句块正常运行结束），位于编译代码第 12 句的 aload_1 指令将抛出的异常压入操作数栈顶，再接下来的 athrow 指令将异常抛出给 tryFinally()方法的调用者。

一个 try 语句中同时带有 catch 和 finally 语句块的编译示例如下：

```
void tryCatchFinally() {
    try {
            tryItOut();
    } catch (TestExc e) {
            handleExc(e);
    } finally {
            wrapItUp();
    }
}
```

编译后代码如下：

```
Method 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
9 aload_3 // Push thrown value
10 invokevirtual #6 // Invoke handler method:
                  // Example.handleExc(LTestExc;)V
13 goto 16 // Huh???
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 jsr 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
28 invokevirtual #5 // Method Example.wrapItUp()V
```

```
31 ret 2 // Return from finally block
Exception table:
From To Target Type
0    4  7     Class TestExc
0   16  20    any
```

严格来说，上面代码中的 goto 指令是没有必要，但是 Oracle 的 1.0.2 版本的 JDK 的 javac 编译器会产生这条指令。

如果 try 语句块中所有指令都正常执行结束，第 4 句的 goto 指令将程序跳转至第 16 句的 finally 语句块之中。在第 26 句，finally 语句块执行结束，程序将跳转回第 19 句的 return 指令，至此 tryCatchFinally()方法运行结束。

如果 tryItOut()方法中抛出了并非 TestExc 异常的异常实例，又或者 catch 语句块中的 handleExc()抛出异常，则这时候异常表中对应为第二个异常处理器就会生效。此时程序将跳转至第 20 句，进入这个异常处理器的代码，将前面抛出的异常保存为第 1 个局部变量中，第 26 句的 finally 语句块一样会被调用。在 tryCatchFinally()方法结束时，这个异常会从第 1 个局部变量中取出，并通过 athrow 指令抛给方法调用者。如果在 finally 子句中有任何的异常抛出，则 finally 语句块停止运行，tryCatchFinally()方法异常退出，并抛出新出现的异常给 tryCatchFinally()方法的调用者。

3.14 同步

Java 虚拟机中的同步（Synchronization）基于进入和退出管程（Monitor）对象实现。无论是显式同步（有明确的 monitorenter 和 monitorexit 指令）还是隐式同步（依赖方法调用和返回指令实现的）都是如此。

在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC_SYNCHRONIZED 标志来隐式实现的（参见 § 2.11.10 “同步”）。

monitorenter 和 monitorexit 指令用于实现同步语句块，譬如：

```
void onlyMe(Foo f) {
  synchronized(f) {
```

```
    doSomething();
  }
}
```

编译后代码如下：

```
Method void onlyMe(Foo)
0 aload_1 // Push f
1 dup // Duplicate it on the stack
2 astore_2 // Store duplicate in local variable 2
3 monitorenter // 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 // Push local variable 2 (f)
9 monitorexit // Exit the monitor associated with f
10 goto 18 // Complete the method normally
13 astore_3 // In case of any throw, end up here
14 aload_2 // Push local variable 2 (f)
15 monitorexit // Be sure to exit the monitor!
16 aload_3 // Push thrown exception...
17 athrow // ...then rethrow the value to the invoker
18 return // Return in the normal case
Exception table:
FromTo Target Type
   4    10    13 any
  13    16    13 any
```

编译器必须确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都必须有执行其对应 monitorexit 指令，而无论这个方法是正常结束（§2.6.4）还是异常结束（§2.6.5）。为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器（§2.10），这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令。

3.15 注解

注解（Annotation）在 Class 文件中如何表示将在§4.7.16 和§4.7.17 中详细描述，这两节明确描述了在 Class 文件格式中，如何描述修饰类型、字段和方法的注解。这里只描述关于包注解的一些额外规则。

如果编译器遇到一个被注解的、声明为在运行时可见的包，那编译器将会生成一个表示接口的 Class 文件，内部形式为（§4.2.1）“package-name.package-info”。这个接口有默认的访问权限（“package-private”），并且没有父接口。它的 ClassFile 结构（§4.1）中的 ACC_INTERFACE 和 ACC_ABSTRACT 的标志（表 4.1）会被自动设置。如果 Class 文件的版本号小于 50.0，则不设置 ACC_SYNTHETIC 标志；如果 Class 文件的版本号为 50.0 或更高，则必须设置 ACC_SYNTHETIC 标志。接口中的全部成员都在《Java 语言规范（Java SE 7 版）》（JLS §9.2）中被明确定义。

package 层次的注解中保存了 Class 文件结构（§4.1）中的 RuntimeVisibleAnnotations（§4.7.16）和 RuntimeInvisibleAnnotations（§4.7.17）属性。

第 4 章 Class 文件格式

本章将描述 Java 虚拟机中定义的 Class 文件格式。每一个 Class 文件都对应着唯一一个类或接口的定义信息，但是相对地，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。本章中，我们只是通俗地将任意一个有效的类或接口所应当满足的格式称为“Class 文件格式”，即使它不一定以磁盘文件的形式存在。

每个 Class 文件都是由 8 字节为单位的字节流组成，所有的 16 位、32 位和 64 位长度的数据将被构造成 2 个、4 个和 8 个 8 字节单位来表示。多字节数据项总是按照 Big-Endian[①]的顺序进行存储。在 Java SDK 中，访问这种格式的数据可以使用 java.io.DataInput、java.io.DataOutput 等接口和 java.io.DataInputStream 和 java.io.DataOutputStream 等类来实现。

本章还定义了一组私有数据类型来表示 Class 文件的内容，它们包括 u1，u2 和 u4，分别代表了 1、2 和 4 个字节的无符号数。在 Java SDK 中这些类型的数据可以通过实现接口 java.io.DataInput 中的 readUnsignedByte、readUnsignedShort 和 readInt 方法进行读取。

本章将采用类似 C 语言结构体的伪结构来描述 Class 文件格式。为了避免与类的字段、类的实例等概念产生混淆，在此把用于描述类结构格式的内容定义为项（Item）。在 Class 文件中，各项按照严格顺序连续存放的，它们之间没有任何填充或对齐作为各项间的分隔符号。

表（Table）是由任意数量的可变长度的项组成，用于表示 Class 文件内容的一系列复合结构。尽管我们采用类似 C 语言的数组语法来表示表中的项，但是读者应当清楚意识到，表是由可变长数据组成的复合结构（表中每项的长度不固定），因此无法直接将字节偏移量来作为索引对表进行访问。而我们描述一个数据结构为数组（Array）时，就意味着它含有零至多个长度固定的项组成，这个时候则可以采用数组索引的方式来访问它[②]。

[①] 译者注：Big-Endian 顺序是指按高位字节在地址最低位，最低字节在地址最高位来存储数据，它是 SPARC、PowerPC 等处理器的默认多字节存储顺序，而 x86 等处理器则是使用了相反的 Little-Endian 顺序来存储数据。为了保证 Class 文件在不同硬件上具备同样的含义，因此在 Java 虚拟机规范中是有必要严格规定了数据存储顺序的。

[②] 译者注：虽然原文中在此定义了“表”和“数组”的关系，但在后文中依然存在表和数组混用的情况。译文中作了一些修正，把各个数据项结构不一致的数据集合用“表”那表示，譬如“constant_pool 表”、“attributes 表”，而把数据项结构一致的数据集合用“数组”来表示，譬如“code[]数组“、“fields[]数组”。

4.1 ClassFile 结构

每一个 Class 文件对应于一个如下所示的 ClassFile 结构体。

```
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 结构体中，各项的含义描述如下：

- magic

  魔数，魔数的唯一作用是确定这个文件是否为一个能被虚拟机所接受的 Class 文件。魔数值固定为 0xCAFEBABE，不会改变。

- minor_version、major_version

  副版本号和主版本号，minor_version 和 major_version 的值分别表示 Class 文件的副、主版本。它们共同构成了 Class 文件的格式版本号。譬如某个 Class 文件的主版本号为 M，副版本号为 m，那么这个 Class 文件的格式版本号就确定为 M.m。Class 文件格式版本号大小的顺序为：1.5 < 2.0 < 2.1。

  一个 Java 虚拟机实例只能支持特定范围内的主版本号（Mi 至 Mj）和 0 至特定范围内（0 至 m）的副版本号。假设一个 Class 文件的格式版本号为 V，仅当 Mi.0 ≤ v ≤ Mj.m 成立时，这个 Class 文件才可以被此 Java 虚拟机支持。不同版本的 Java 虚拟机实现支持的版本号也不同，高版本号的 Java 虚拟机实现可以支持低版本号的 Class 文件，

反之则不成立[①]。

- constant_pool_count

  常量池计数器，constant_pool_count 的值等于 constant_pool 表中的成员数加 1。constant_pool 表的索引值只有在大于 0 且小于 constant_pool_count 时才会被认为是有效的[②]，对于 long 和 double 类型有例外情况，可参见§4.4.5。

- constant_pool[]

  常量池，constant_pool 是一种表结构（§4.4），它包含 Class 文件结构及其子结构中引用的所有字符串常量、类或接口名、字段名和其它常量。常量池中的每一项都具备相同的格式特征——第一个字节作为类型标记用于识别该项是哪种类型的常量，称为“tag byte”。常量池的索引范围是 1 至 constant_pool_count−1。

- access_flags

  访问标志，access_flags 是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。access_flags 的取值范围和相应含义见表 4.1 所示。

表 4.1 访问和修饰符标志

标记名	值	含义
ACC_PUBLIC	0x0001	可以被包的类外访问。
ACC_FINAL	0x0010	不允许有子类。
ACC_SUPER	0x0020	当用到 invokespecial 指令时，需要特殊处理[③]的父类方法。
ACC_INTERFACE	0x0200	标识定义的是接口而不是类。
ACC_ABSTRACT	0x0400	不能被实例化。
ACC_SYNTHETIC	0x1000	标识并非 Java 源码生成的代码。

[①] Oracle 的 JDK 在 1.0.2 版本时，支持的 Class 格式版本号范围是 45.0 至 45.3；JDK 版本在 1.1.x 时，支持的 Class 格式版本号范围扩展至 45.0 至 45.65535；JDK 版本为 1. k 时（k ≥2）时，对应的 Class 文件格式版本号的范围是 45.0 至 44+k.0

[②] 译者注：虽然值为 0 的 constant_pool 索引是无效的，但其他用到常量池的数据结构可以使用索引 0 来表示“不引用任何一个常量池项”的意思。

[③] 译者注：此处“特殊处理”是相对于 JDK 1.0.2 之前的 Class 文件而言，invokespecial 的语义和处理方式在 JDK 1.0.2 时发生了变化，为避免二义性，在 JDK 1.0.2 之后编译出的 Class 文件，都带有 ACC_SUPER 标志用以区分。

ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

- 带有 ACC_SYNTHETIC 标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。
- 带有 ACC_ENUM 标志的类，意味着它或它的父类被声明为枚举类型。
- 带有 ACC_INTERFACE 标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个 Class 文件被设置了 ACC_INTERFACE 标志，那么同时也得设置 ACC_ABSTRACT 标志（JLS §9.1.1.1）。同时它不能再设置 ACC_FINAL、ACC_SUPER 和 ACC_ENUM 标志。
- 注解类型必定带有 ACC_ANNOTATION 标记，如果设置了 ANNOTATION 标记，ACC_INTERFACE 也必须被同时设置。如果没有同时设置 ACC_INTERFACE 标记，那么这个 Class 文件可以具有表 4.1 中的除 ACC_ANNOTATION 外的所有其它标记。当然 ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标记除外（JLS §8.1.1.2）。
- ACC_SUPER 标志用于确定该 Class 文件里面的 invokespecial 指令使用的是哪一种执行语义。目前 Java 虚拟机的编译器都应当设置这个标志。ACC_SUPER 标记是为了向后兼容旧编译器编译的 Class 文件而存在的，在 JDK1.0.2 版本以前的编译器产生的 Class 文件中，access_flag 里面没有 ACC_SUPER 标志。同时，JDK1.0.2 前的 Java 虚拟机遇到 ACC_SUPER 标记会自动忽略它。
- 在表 4.1 中没有使用的 access_flags 标志位是为未来扩充而预留的，这些预留的标志为在编译器中会被设置为 0， Java 虚拟机实现也会自动忽略它们。

- this_class

  类索引，this_class 的值必须是对 constant_pool 表中项目的一个有效索引值。constant_pool 表在这个索引处的项必须为 CONSTANT_Class_info 类型常量（§4.4.1），表示这个 Class 文件所定义的类或接口。

- super_class

  父类索引，对于类来说，super_class 的值必须为 0 或者是对 constant_pool 表中项目的一个有效索引值。如果它的值不为 0，那 constant_pool 表在这个索引处的项必须为 CONSTANT_Class_info 类型常量（§4.4.1），表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的 access_flag 中都不能带

有 ACC_FINAL 标记。对于接口来说，它的 Class 文件的 super_class 项的值必须是对 constant_pool 表中项目的一个有效索引值。constant_pool 表在这个索引处的项必须为代表 java.lang.Object 的 CONSTANT_Class_info 类型常量（§4.4.1）。如果 Class 文件的 super_class 的值为 0，那这个 Class 文件只可能是定义的是 java.lang.Object 类，只有它是唯一没有父类的类。

- interfaces_count

  接口计数器，interfaces_count 的值表示当前类或接口的直接父接口数量。

- interfaces[]

  接口表，interfaces[]数组中的每个成员的值必须是一个对 constant_pool 表中项目的一个有效索引值，它的长度为 interfaces_count。每个成员 interfaces[i] 必须为 CONSTANT_Class_info 类型常量（§4.4.1），其中 0 ≤ i < interfaces_count。在 interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即 interfaces[0]对应的是源代码中最左边的接口。

- fields_count

  字段计数器，fields_count 的值表示当前 Class 文件 fields[]数组的成员个数。fields[]数组中每一项都是一个 field_info 结构（§4.5）的数据项，它用于表示该类或接口声明的类字段或者实例字段[①]。

- fields[]

  字段表，fields[]数组中的每个成员都必须是一个 fields_info 结构（§4.5）的数据项，用于表示当前类或接口中某个字段的完整描述。fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。

- methods_count

  方法计数器，methods_count 的值表示当前 Class 文件 methods[]数组的成员个数。Methods[]数组中每一项都是一个 method_info 结构（§4.5）的数据项。

- methods[]

[①] 译者注：类字段即被声明为 static 的字段，也称为类变量或者类属性，同样，实例字段是指未被声明为 static 的字段。由于《Java 虚拟机规范》中，“Variable”和“Attribute”出现频率很高且在大多数场景中具备其他含义，所以译文中统一把“Field”翻译为“字段”，即“类字段”、“实例字段”。

方法表，methods[]数组中的每个成员都必须是一个 method_info 结构（§4.6）的数据项，用于表示当前类或接口中某个方法的完整描述。如果某个 method_info 结构的 access_flags 项既没有设置 ACC_NATIVE 标志也没有设置 ACC_ABSTRACT 标志，那么它所对应的方法体就应当可以被 Java 虚拟机直接从当前类加载，而不需要引用其它类。method_info 结构可以表示类和接口中定义的所有方法，包括实例方法、类方法、实例初始化方法方法（§2.9）和类或接口初始化方法方法（§2.9）。methods[]数组只描述当前类或接口中声明的方法，不包括从父类或父接口继承的方法。

- attributes_count

  属性计数器，attributes_count 的值表示当前 Class 文件 attributes 表的成员个数。attributes 表中每一项都是一个 attribute_info 结构（§4.7）的数据项。

- attributes[]

  属性表，attributes 表的每个项的值必须是 attribute_info 结构（§4.7）。在本规范里，Class 文件结构中的 attributes 表的项包括下列定义的属性：InnerClasses（§4.7.6）、EnclosingMethod（§4.7.7）、Synthetic（§4.7.8）、Signature（§4.7.9）、SourceFile（§4.7.10），SourceDebugExtension（§4.7.11）、Deprecated（§4.7.15）、RuntimeVisibleAnnotations（§4.7.16）、RuntimeInvisibleAnnotations（§4.7.17）以及 BootstrapMethods（§4.7.21）属性。对于支持 Class 文件格式版本号为 49.0 或更高的 Java 虚拟机实现，必须正确识别并读取 attributes 表中的 Signature（§4.7.9）、RuntimeVisibleAnnotations（§4.7.16）和 RuntimeInvisibleAnnotations（§4.7.17）属性。对于支持 Class 文件格式版本号为 51.0 或更高的 Java 虚拟机实现，必须正确识别并读取 attributes 表中的 BootstrapMethods（§4.7.21）属性。本规范要求任一 Java 虚拟机实现可以自动忽略 Class 文件的 attributes 表中的若干（甚至全部）它不可识别的属性项。任何本规范未定义的属性不能影响 Class 文件的语义，只能提供附加的描述信息（§4.7.1）。

4.2 各种内部表示名称

4.2.1 类和接口的二进制名称

在 Class 文件结构中出现的类或接口的名称，都通过全限定形式（Fully Qualified Form）来表示，这被称作它们的“二进制名称”（JLS §13.1）。这个名称使用 CONSTANT_Utf8_info（§4.4.7)结构来表示，因此如果忽略其他一些约束限制的话，这个名称可能来自整个 Unitcode 字符空间的任意字符组成。类和接口的二进制名称还会被 CONSTANT_NameAndType_info（§4.4.6）结构所引用，用于构成它们的描述符（§4.3），引用这些名称通过引用它们的 CONSTANT_Class_info（§4.4.1）结构来实现。

由于历史原因，出现在 Class 文件结构中的二进制名称的语法与《Java 语言规范》中 §13.1 规定的语法二进制名格式有差别。在本规范规定的内部形式中，用来分隔各个标识符的符号不在是 ASCII 字符点号（'.'），而是被 ASCII 字符斜杠（'/'）所代替，每个标识符都是一个非全限定名（Unqualified Names，§4.2.2）[①]。

譬如，类 Thread 的正常的二进制名是 java.lang.Thread。在 Class 文件的内部表示形式里面，对类 java.lang.Thrad 的引用是通过来一个代表字符串“java/lang/Thread”的 CONSTANT_Utf8_info 结构来实现的。

4.2.2 非全限定名

方法名，字段名和局部变量名都被使用非全限定名（Unqualified Names）进行存储。非全限定名中不能包含 ASCII 字符"."、";"、"["和"/"（也不能包含他们的 Unitcode 表示形式，既类似"\u2E"这种形式）。

方法的非全限定名还有一些额外的限制，除了实例初始化方法“<init>”和类初始化方法“<clinit>”以外，其他方法非全限定名中不能包含 ASCII 字符"<"和">"（也不能包含他们的 Unicode 表示形式）[②]。

---

① 译者注：解释一下全限定名和非全限定名，全限定名是在整个 JVM 中的绝对名称，譬如“java.lang.Object”，而非全限定名是指当前环境（譬如当前类）中的相对名称，譬如“Object”。

② 请注意：虽然字段名和接口方法名可以使用<init>和<clinit>（译者注：这里是在 Class 文件角度上描述，Class 文件格式中可以存在这 2 个方法名。在 Java 程序编码时还要遵循 JLS 的约束，即 Java 源码中不能存在这 2 个方法名），但是没有任何调用指令可以调用到<clinit>，也仅有 invokespecial 可以调用<init>。

4.3 描述符和签名

描述符（Descriptor）是一个描述字段或方法的类型的字符串。在 Class 文件格式中，描述符使用改良的 UTF-8 字符串（§4.4.7）来表示。如果不考虑其他的约束，它可以使用 Unicode 字符空间中的任意字符。

签名（Signature）是用于描述字段、方法和类型定义中的泛型信息的字符串。

4.3.1 语法符号

描述符和签名都是用特定的语法符号（Grammar）来表示，这些语法是一组可表达如何根据不同的类型去产生可恰当描述它们的字符序列的标识集合。在本规范中，语法的终止符号用定长的黑体字表示。非终止符号用斜体字表示，非终止符的定义由被定义的非终止名后跟随一个冒号表示。冒号右侧的一个或多个可交换的非终止符连续排列，每个非终止符占一行[①]。例如：

```
FieldType:
     BaseType
     ObjectType
     ArrayType
```

上面文字表达的意思是 FileType 可以表示 BaseType、ObjectType、ArrayType 三者之一。

当一个有星号（*）跟随的非终止符出现在一个语法标识的右侧时，说明带有这个非终止符的语法标识将产生 0 或多个不同值，这些值按照顺序且无间隔的排列在非终止符后面。当一个有加号（+）跟随的非终止符出现在一个结构的右侧时，说明这个非终止符将产生一个或多个不同值，这些值按照顺序且无间隔的排列在非终止符后面。例如：

```
MethodDescriptor:
     ( ParameterDescriptor* ) ReturnDescriptor
```

上面文字表达的意思是 MethodDescriptor 的是由左括号“(”、0 或若干个连续排列的 ParameterDescriptor 值、右括号“)”、ReturnDescriptor 值构成。

[①] 译者注：由于中文、英文之间排版差异，译文中某些地方没有完全遵循作者说描述的字体样式。

4.3.2 字段描述符

字段描述符（Field Descriptor），是一个表示类、实例或局部变量的语法符号，它是由语法产生的字符序列：

```
FieldDescriptor:
    FieldType
ComponentType:
    FieldType
FieldType:
    BaseType
    ObjectType
    ArrayType
BaseType:
    B
    C
    D
    F
    I
    J
    S
    Z
ObjectType:
    L Classname ;
ArrayType:
    [ ComponentType
```

所有表示基本类型（BaseType）的字符、表示对象类型（ObjectType）中的字符"L"，表示数组类型（ArrayType）的"["字符都是 ASCII 编码的字符。对象类型（ObjectType）中的 Classname 表示一个类或接口二进制名称的内部格式（§4.2.1）。表示数组类型的有效描述符的长度应小于等于 255。所有字符类型的解释如表 4.2 所示。

表 4.2 基本类型字符解释表

字符	类型	含义
B	byte	有符号字节型数
C	char	Unicode 字符，UTF-16 编码
D	double	双精度浮点数

F	float	单精度浮点数
I	int	整型数
J	long	长整数
S	short	有符号短整数
Z	boolean	布尔值 true/false
L Classname;	reference	一个名为<Classname>的实例
[	reference	一个一维数组

举个例子：描述 int 实例变量的描述符是**“I”**；java.lang.Object 的实例描述符是**“Ljava/lang/Object;”**。注意，这里用到了类 Object 的二进制名的内部形式（此处是内部形式）。double 的三维数组“double d[][][];”的描述符为“**[[[D**”。

4.3.3 方法描述符

方法描述符（Method Descriptor）描述一个方法所需的参数和返回值信息：

```
MethodDescriptor:
    ( ParameterDescriptor* ) ReturnDescriptor
```

参数描述符（ParameterDescriptor）描述需要传给这个方法的参数信息：

```
ParameterDescriptor:
FieldType
```

返回描值述符(ReturnDescriptor)从当前方法返回的值，它是由语法产生的字符序列：

```
ReturnDescriptor:
    FieldType
    VoidDescriptor
```

其中 VoidDescriptor 表示当前方法无返回值，即返回类型是 void。符号如下（字符 V 即 void）：

```
VoidDescriptor:
  V
```

如果一个方法描述符是有效的，那么它对应的方法的参数列表总长度小于等于 255，对于实例方法和接口方法，需要额外考虑隐式参数 this。参数列表长度的计算规则如下：每个 long 和

double 类参数长度为 2，其余的都为 1，方法参数列表的总长度等于所有参数的长度之和。

例如，方法：

```
Object mymethod(int i, double d, Thread t)
```

的描述符为：

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

注意：这里使用了 Object 和 Thread 的二进制名称的内部形式。

无论 mymethod()是静态方法还是实例方法，它的方法描述符都是相同的。尽管实例方法除了传递自身定义的参数，还需要额外传递参数 this，但是这一点不是由法描述符来表达的。参数 this 的传递，是由 Java 虚拟机实现在调用实例方法所使用的指令中实现的隐式传递。

4.3.4 签名

签名（Signatures）是用于给 Java 语言使用的描述信息编码，不在 Java 虚拟机系统使用的类型中。泛型类型、方法描述和参数化类型描述等都属于签名。关于签名的详细说明请参考《Java 语言规范（Java SE 7 版）》。

Java 编译器需要这类信息来实现（或辅助实现）反射（reflection）和跟踪调试功能。

终止符用于表示 Java 编译器产生的各种描述符的名字。Java 编译器产生的类型有：类型（Type）、字段（Field）、局部变量（Local Variable）、参数（Parameter）、方法（Method）、变量类型（type variable）等。这些类型的名字中，不能包含字符"."、";"、"["、"/"、"<"、">"和":"，但是可以包含其他 Java 语言没有规定的标识符。

类签名（Class Signature）由 ClassSignature 定义，作用是把 Class 申明的类型信息编译成对应的签名信息。它描述当前类可能包含的所有的（泛型类型的）形式类型参数，如果 Class 有直接父类和父接口，类签名会列出它们（或将它们参数化）。

```
ClassSignature:
    FormalTypeParametersopt SuperclassSignature
SuperinterfaceSignature*
```

一个正式的形式类型参数必须有自己的名字和对这个参数的类或接口的限制（bounds），限制跟在类型名之后。如果形式类型参数的类限定没有给定一个具体定义，那么这个参数类型被默认定义为 java.lang.Object。

```
FormalTypeParameters:
    < FormalTypeParameter+ >
FormalTypeParameter:
    Identifier ClassBound InterfaceBound*
ClassBound:
    : FieldTypeSignatureopt
InterfaceBound:
    : FieldTypeSignature
SuperclassSignature:
    ClassTypeSignature
SuperinterfaceSignature:
    ClassTypeSignature
```

字段类型签名（Field Type Signature）由 FieldTypeSignature 定义，作用是将字段、参数或局部变量的类型编译成对应的签名信息。

```
FieldTypeSignature:
    ClassTypeSignature
    ArrayTypeSignature
    TypeVariableSignature
```

类的类型签名（Class Type Signature）给出了当前类或接口的完整类型信息。类的类型签名必被须明确表达，否则不能准确无误的映射至二进制名，在类的类型签名表示方式中，将内部形式的名称中的符号“.”替换成“$”。

```
ClassTypeSignature:
    L PackageSpecifieropt SimpleClassTypeSignature
ClassTypeSignatureSuffix* ;
PackageSpecifier:
    Identifier / PackageSpecifier*
SimpleClassTypeSignature:
    Identifier TypeArgumentsopt
ClassTypeSignatureSuffix:
    . SimpleClassTypeSignature
TypeVariableSignature:
    T Identifier ;
TypeArguments:
    < TypeArgument+ >
TypeArgument:
    WildcardIndicatoropt FieldTypeSignature
    *
WildcardIndicator:
    +
```

```
        -
    ArrayTypeSignature:
        [ TypeSignature
    TypeSignature:
        FieldTypeSignature
        BaseType
```

方法签名（Method Signature）由 MethodTypeSignature 定义，作用是将方法中所有的形式参数的类型编译成相应的签名信息（或将它们参数化）。形式参数不包括在 throw 语句中申明的参数类型，返回类型和方法申明的所有的形式类型参数。

```
    MethodTypeSignature:
        FormalTypeParametersopt (TypeSignature*) ReturnType
ThrowsSignature*
    ReturnType:
        TypeSignature
        VoidDescriptor
    ThrowsSignature:
        ^ ClassTypeSignature
        ^ TypeVariableSignature
```

如果方法或构造函数不需要抛出异常时，那么 ThowsSignature 就会从 MethodTypeSignature 中移除。

如果泛型类，接口，构造函数或成员的泛型签名在 Java 编程语言中包含对类型变量或参数化类型的引用，则 Java 编译器必须这些签名的信息。

对于同一个方法或构造函数，描述符和签名（§4.3.3）并不一定完全匹配，这取决于使用的编译器。尤其是 MethodTypeSignature 编译形式参数的 TypeSignatures 数量会少于 MethodDescriptor 中 ParameterDescriptors 的数量。

Oracle 发布的 Java 虚拟机实现在加载和链接时，并不校验 Class 文件的签名内容。直到被反射方法调用时才会校验，JDK 的 API 中的 java.lang.reflect 的部分也说明了这点。在未来版本中，Java 虚拟机可能会在启动和链接时对签名加入部分或全部校验。

4.4 常量池

Java 虚拟机指令执行时不依赖与类、接口，实例或数组的运行时布局，而是依赖常量池（constant_pool）表中的符号信息。

所有的常量池项都具有如下通用格式：

```
cp_info {
    u1 tag;
    u1 info[];
}
```

常量池中，每个 cp_info 项的格式必须相同，它们都以一个表示 cp_info 类型的单字节“tag”项开头。后面 info[]项的内容 tag 由的类型所决定。tag 有效的类型和对应的取值在表 4.3 列出。每个 tag 项必须跟随 2 个或更多的字节，这些字节用于给定这个常量的信息，附加字节的信息格式由 tag 的值决定。

表 4.3 常量池的 tag 项说明

常量类型	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
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

4.4.1 CONSTANT_Class_info 结构

CONSTANT_Class_info 结构用于表示类或接口，格式如下：

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

CONSTANT_Class_info 结构的项的说明：

- tag

  CONSTANT_Class_info 结构的 tag 项的值为 CONSTANT_Class（7）。

- name_index

  name_index 项的值，必须是对常量池的一个有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，代表一个有效的类或接口二进制名称的内部形式。

因为数组也是由对象表示，所以字节码指令 anewarray 和 multianewarray 可以通过常量池中的 CONSTANT_Class_info（§4.4.1）结构来引用类数组。对于这些数组，类的名字就是数组类型的描述符，例如：

表现二维 int 数组类型

```
int[][]
```

的名字是：

```
[[I
```

表示一维 Thread 数组类型

```
Thread[]
```

的名字是：

```
[Ljava/lang/Thread;
```

一个有效的数组类型描述符中描述的数组维度必须小于等于 255。

4.4.2 CONSTANT_Fieldref_info， CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 结构

字段，方法和接口方法由类似的结构表示：

字段：

```
CONSTANT_Fieldref_info {
     u1 tag;
     u2 class_index;
     u2 name_and_type_index;
}
```

方法：

```
CONSTANT_Methodref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
}
```

接口方法：

```
CONSTANT_InterfaceMethodref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
}
```

这些结构各项的说明如下：

- tag

  CONSTANT_Fieldref_info 结构的 tag 项的值为 CONSTANT_Fieldref（9）。

  CONSTANT_Methodref_info 结构的 tag 项的值为 CONSTANT_Methodref（10）。

  CONSTANT_InterfaceMethodref_info 结构的 tag 项的值为

  CONSTANT_InterfaceMethodref（11）。

- class_index

  class_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Class_info（§4.4.1）结构，表示一个类或接口，当前字段或方法是这个类或接口的成员。

  CONSTANT_Methodref_info 结构的 class_index 项的类型必须是类(不能是接口)。

  CONSTANT_InterfaceMethodref_info 结构的 class_index 项的类型必须是接口（不能是类）。CONSTANT_Fieldref_info 结构的 class_index 项的类型既可以是类也可以是接口。

- name_and_type_index

  name_and_type_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必

须是 CONSTANT_NameAndType_info（§4.4.6）结构，它表示当前字段或方法的名字和描述符。

在一个 CONSTANT_Fieldref_info 结构中，给定的描述符必须是字段描述符（§4.3.2）。而 CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 中给定的描述符必须是方法描述符（§4.3.3）。

如果一个 CONSTANT_Methodref_info 结构的方法名以“<”（'\u003c'）开头，则说明这个方法名是特殊的<init>，即这个方法是实例初始化方法（§2.9），它的返回类型必须为空。

4.4.3 CONSTANT_String_info 结构

CONSTANT_String_info 用于表示 java.lang.String 类型的常量对象，格式如下：

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

CONSTANT_String_info 结构各项的说明如下：

- tag

  CONSTANT_String_info 结构的 tag 项的值为 CONSTANT_String（8）。

- string_index

  string_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一组 Unicode 码点序列，这组 Unicode 码点序列最终会被初始化为一个 String 对象。

4.4.4 CONSTANT_Integer_info 和 CONSTANT_Float_info 结构

CONSTANT_Intrger_info 和 CONSTANT_Float_info 结构表示 4 字节（int 和 float）的数值常量：

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

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

这些结构各项的说明如下：

- tag

  CONSTANT_Integer_info 结构的 tag 项的值是 CONSTANT_Integer（3）。

  CONSTANT_Float_info 结构的 tag 项的值是 CONSTANT_Float（4）。

- bytes

  CONSTANT_Integer_info 结构的 bytes 项表示 int 常量的值，按照 Big-Endian 的顺序存储。

  CONSTANT_Float_info 结构的 bytes 项按照 IEEE 754 单精度浮点格式(§2.3.2)．表示 float 常量的值，按照 Big-Endian 的顺序存储。

  CONSTANT_Float_info 结构表示的值将按照下列方式来表示，bytes 项的值首先被转换成一个 int 常量的 bits：

  - 如果 bits 值为 0x7f800000，表示 float 值为正无穷。
  - 如果 bits 值为 0xff800000，表示 float 值为负无穷。
  - 如果 bits 值在范围 0x7f800001 到 0x7fffffff 或者 0xff800001 到 0xffffffff 内，表示 float 值为 NaN。
  - 在其它情况下，设 s、e、m，它们值根据 bits 和如下公式计算：

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

    则 float 的浮点值为数值表达式 $s \cdot m \cdot 2^{e-150}$ 的计算结果。

4.4.5 CONSTANT_Long_info 和 CONSTANT_Double_info 结构

CONSTANT_Long_info 和 CONSTANT_Double_info 结构表示 8 字节（long 和 double）的数值常量：

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

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

在 Class 文件的常量池中，所有的 8 字节的常量都占两个表成员（项）的空间。如果一个 CONSTANT_Long_info 或 CONSTANT_Double_info 结构的项在常量池中的索引为 n，则常量池中下一个有效的项的索引为 n+2，此时常量池中索引为 n+1 的项有效但必须被认为不可用[①]。

CONSTANT_Long_info 和 CONSTANT_Double_info 结构各项的说明如下：

- tag

  CONSTANT_Long_info 结构的 tag 项的值是 CONSTANT_Long（5）。

  CONSTANT_Double_info 结构的 tag 项的值是 CONSTANT_Double（6）。

- high_bytes 和 low_bytes

  CONSTANT_Long_info 结构中的无符号的 high_bytes 和 low_bytes 项用于共同表示 long 型常量，构造形式为((long) high_bytes << 32) + low_bytes，high_bytes 和 low_bytes 都按照 Big-Endian 顺序存储。

  CONSTANT_Double_info 结构中的 high_bytes 和 low_bytes 共同按照 IEEE 754 双精度浮点格式（§2.3.2）表示 double 常量的值。high_bytes 和 low_bytes 都按照 Big-Endian 顺序存储。

  CONSTANT_Double_info 结构表示的值将按照下列方式来表示，high_bytes 和

[①] 由于历史原因（译者注：是指 JVM 开发时是处于 32 位机为主流的时代），让 8 字节常量占用 2 个表元素的空间是一个无奈的选择。

low_bytes 首先被转换成一个 long 常量的 bits：

- 如果 bits 值为 0x7ff0000000000000L，表示 double 值为正无穷。
- 如果 bits 值为 0xfff0000000000000L，表示 double 值为负无穷。
- 如果 bits 值在范围 0x7ff0000000000001L 到 0x7fffffffffffffffL 或者 0xfff0000000000001L 到 0xffffffffffffffffL 内，表示 double 值为 NaN。
- 在其它情况下，设 s、e、m，它们的值根据 bits 和如下公式计算：

```
int s =((bits >> 63) == 0) ? 1 : -1;
int e =(int)((bits >> 52) & 0x7ffL);
long m =(e == 0) ?
        (bits & 0xfffffffffffffL) << 1 :
        (bits & 0xfffffffffffffL) | 0x10000000000000L;
```

则 double 的浮点值为数学表达式 $s \cdot m \cdot 2^{e-1075}$ 的计算结果。

4.4.6 CONSTANT_NameAndType_info 结构

CONSTANT_NameAndType_info 结构用于表示字段或方法，但是和 4.4.2 章节中介绍的 3 个结构不同，CONSTANT_NameAndType_info 结构没有标识出它所属的类或接口，格式如下：

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

CONSTANT_NameAndType_info 结构各项的说明如下：

- tag

  CONSTANT_NameAndType_info 结构的 tag 项的值为 CONSTANT_NameAndType（12）。

- name_index

  name_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，这个结构要么表示特殊的方法名<init>（§2.9），要么表示一个有效的字段或方法的非限定名（Unqualified Name）。

- descriptor_index

descriptor_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，这个结构表示一个有效的字段描述符（§4.3.2）或方法描述符（§4.3.3）。

4.4.7 CONSTANT_Utf8_info 结构

CONSTANT_Utf8_info 结构用于表示字符串常量的值：

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

CONSTANT_Utf8_info 结构各项的说明如下：

- tag

  CONSTANT_Utf8_info 结构的 tag 项的值为 CONSTANT_Utf8（1）。

- length

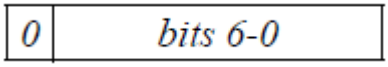
  length 项的值指明了 bytes[]数组的长度（注意，不能等同于当前结构所表示的 String 对象的长度），CONSTANT_Utf8_info 结构中的内容是以 length 属性确定长度而不是以 null 作为字符串的终结符。

- bytes[]

  bytes[]是表示字符串值的byte数组，bytes[]数组中每个成员的byte值都不会是0，也不在 0xf0 至 0xff 范围内。

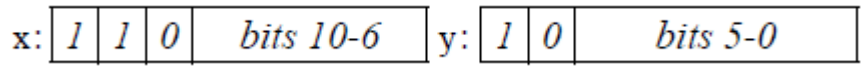
字符串常量采用改进过的 UTF-8 编码表示。这种以改进过的 UTF-8 编码中，用于表示的字符串的码点字符序列可以包含 ASCII 中的所有非空(Non-Null)字符和所有 Unicode 编码的字符，一个字符占一个 byte。

码点在范围'\u0001'至'\u007F'内的字符用一个单字节表示：



byte 的后 7 位数据表示一个码点值。

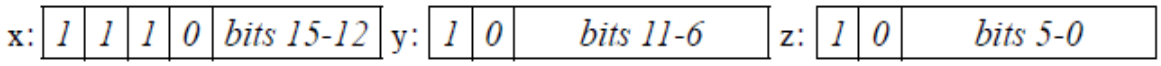
字符为'\u0000'（表示字符'null'），或者在范围'\u0080'至'\u07FF'的字符用一对字节 x 和 y 表示：



x 和 y 计算字符值的公式为：

```
((x & 0x1f) << 6) +(y & 0x3f)
```

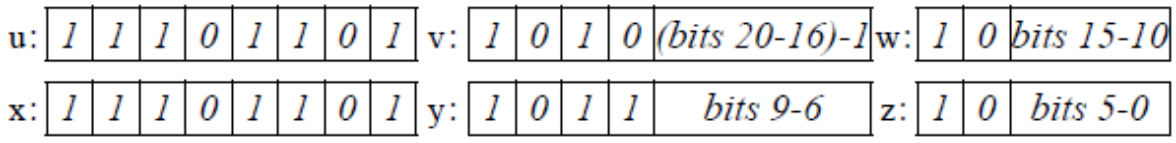
在范围'\u0800'至'\uFFFF'中的字符用 3 个字节 x，y 和 z 表示：



x，y 和 z 计算字符值的公式为：

```
((x & 0xf) << 12) + ((y & 0x3f) << 6) + (z & 0x3f)
```

超过 U+FFFF 范围的字符（称为补充字符，Supplementary Characters），在 UTF-16 编码中也需要 2 个 UTF-16 字符单元来表示，而 UTF-16 中的每个字符单元占 3 个字节，这就意味着在我们的编码方式中，补充字符需要 6 个字节来表示，u，v，w，x，y 和 z：



这 6 个字符计算字符值的公式为：

```
0x10000+((v&0x0f)<<16)+((w&0x3f)<<10)+(y&0x0f)<<6)+(z&0x3f)
```

在 Class 文件中，多字节字符按照 Big-Endian 顺序存储。

和“标准”版 UTF-8 格式相比，Java 虚拟机采用的改进版 UTF-8 格式有两点不同。第一，“null”字符（(char) 0）用双字节格式编码表示而不是单字节，所以，改进版 UTF-8 格式不会直接出现 null 值。第二，改进版的 UTF-8 只使用标准版 UTF-8 中的单字节、双字节和三字节格式。Java 虚拟机不能识别标准版 UTF-8 格式定义 4 字节格式，而是使用自定义的二次三字节（Two-Times-Three-Byte）格式来代替。

更多关于标准版 UTF-8 格式的内容可以参考《The Unicode Standard》（版本 6.0.0）的第 3.9 章节“Unicode Encoding Forms”。

4.4.8 CONSTANT_MethodHandle_info 结构

CONSTANT_MethodHandle_info 结构用于表示方法句柄，结构如下：

```
CONSTANT_MethodHandle_info {
```

```
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

CONSTANT_MethodHandle_info 结构各项的说明如下：

- tag

  CONSTANT_MethodHandle_info 结构的 tag 项的值为 CONSTANT_MethodHandle（15）。

- reference_kind

  reference_kind 项的值必须在 1 至 9 之间（包括 1 和 9），它决定了方法句柄的类型。方法句柄类型的值表示方法句柄的字节码行为（Bytecode Behavior §5.4.3.5）。

- reference_index

  reference_index 项的值必须是对常量池的有效索引：

  - 如果 reference_kind 项的值为 1（REF_getField）、2（REF_getStatic）、3（REF_putField）或 4（REF_putStatic），那么常量池在 reference_index 索引处的项必须是 CONSTANT_Fieldref_info（§4.4.2）结构，表示由一个字段创建的方法句柄。
  - 如果 reference_kind 项的值是 5（REF_invokeVirtual）、6（REF_invokeStatic）、7（REF_invokeSpecial）或 8（REF_newInvokeSpecial），那么常量池在 reference_index 索引处的项必须是 CONSTANT_Methodref_info（§4.4.2）结构，表示由类的方法或构造函数创建的方法句柄。
  - 如果 reference_kind 项的值是 9（REF_invokeInterface），那么常量池在 reference_index 索引处的项必须是 CONSTANT_InterfaceMethodref_info（§4.4.2）结构，表示由接口方法创建的方法句柄。
  - 如果 reference_kind 项的值是 5（REF_invokeVirtual）、6（REF_invokeStatic）、7（REF_invokeSpecial）或 9（REF_invokeInterface），那么方法句柄对应的方法不能为实例初始化（<init>）方法或类初始化方法（<clinit>）。
  - 如果 reference_kind 项的值是 8（REF_newInvokeSpecial），那么方法句柄

对应的方法必须为实例初始化（<init>）方法。

4.4.9 CONSTANT_MethodType_info 结构

CONSTANT_MethodType_info 结构用于表示方法类型：

```
CONSTANT_MethodType_info {
    u1 tag;
    u2 descriptor_index;
}
```

- tag

  CONSTANT_MethodType_info 结构的 tag 项的值为 CONSTANT_MethodType（16）。

- descriptor_index

  descriptor_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示方法的描述符（§4.3.3）。

4.4.10 CONSTANT_InvokeDynamic_info 结构

CONSTANT_InvokeDynamic_info 用于表示 invokedynamic 指令所使用到的引导方法（Bootstrap Method）、引导方法使用到动态调用名称（Dynamic Invocation Name）、参数和请求返回类型、以及可以选择性的附加被称为静态参数（Static Arguments）的常量序列。

```
CONSTANT_InvokeDynamic_info {
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}
```

CONSTANT_InvokeDynamic_info 结构各项的说明如下：

- tag

  CONSTANT_InvokeDynamic_info 结构的 tag 项的值为 CONSTANT_InvokeDynamic（18）。

- bootstrap_method_attr_index

bootstrap_method_attr_index 项的值必须是对当前 Class 文件中引导方法表（§4.7.21）的 bootstrap_methods[]数组的有效索引。

- name_and_type_index

  name_and_type_index 项的值必须是对当前常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_NameAndType_info（§4.4.6）结构，表示方法名和方法描述符（§4.3.3）。

4.5 字段

每个字段（Field）都由 field_info 结构所定义，在同一个 Class 文件中，不会有两个字段同时具有相同的字段名和描述符（§4.3.2）。

field_info 结构格式如下：

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

field_info 结构各项的说明如下：

- access_flags

  access_flags 项的值是用于定义字段被访问权限和基础属性的掩码标志。

  access_flags 的取值范围和相应含义见表 4.4 所示。

表 4.4 字段 access_flags 标记列表

标记名	值	说明
ACC_PUBLIC	0x0001	public，表示字段可以从任何包访问。
ACC_PRIVATE	0x0002	private，表示字段仅能该类自身调用。
ACC_PROTECTED	0x0004	protected，表示字段可以被子类调用。
ACC_STATIC	0x0008	static，表示静态字段。
ACC_FINAL	0x0010	final，表示字段定义后值无法修改（JLS §17.5）。

ACC_VOLATILE	0x0040	volatile，表示字段是易变的。
ACC_TRANSIENT	0x0080	transient，表示字段不会被序列化。
ACC_SYNTHETIC	0x1000	表示字段由编译器自动产生。
ACC_ENUM	0x4000	enum，表示字段为枚举类型。

字段如果带有 ACC_SYNTHETIC 标志，则说明这个字段不是由源码产生的，而是由编译器自动产生的。

字段如果被标有 ACC_ENUM 标志，这说明这个字段是一个枚举类型成员。

Class 文件中的字段可以被设置多个表 4.4 的标记。不过有些标记是互斥的，一个字段最多只能设置 ACC_PRIVATE， ACC_PROTECTED，和 ACC_PUBLIC（JLS §8.3.1）三个标志中的一个，也不能同时设置标志 ACC_FINAL 和 ACC_VOLATILE（JLS §8.3.1.4）。

接口中的所有字段都具有 ACC_PUBLIC，ACC_STATIC 和 ACC_FINAL 标记，也可能被设置 ACC_SYNTHETIC 标记，但是不能含有表 4.4 中的其它符号标记了（JLS §9.3）。在表 4.4 中没有出现的 access_flags 项的值为扩充而预留，在生成的 Class 文件中应被设置成 0，Java 虚拟机实现也应该忽略它们。

- name_index

  name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的字段的非全限定名（§4.2.2）。

- descriptor_index

  descriptor_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的字段的描述符（§4.3.2）。

- attributes_count

  attributes_count 的项的值表示当前字段的附加属性（§4.7）的数量。

- attributes[]

  attributes 表的每一个成员的值必须是 attribute（§4.7）结构，一个字段可以有任意个关联属性。

  本规范所定义的 field_info 结构中，attributes 表可出现的成员有：

ConstantValue（§4.7.2），Synthetic（§4.7.8），Signature（§4.7.9），Deprecated（§4.7.15），RuntimeVisibleAnnotations（§4.7.16）和 RuntimeInvisibleAnnotations（§4.7.17）。

Java 虚拟机实现必须正确的识别和读取 field_info 结构的 attributes 表中的 ConstantValue（§4.7.2）属性。如果 Java 虚拟机实现支持版本号为 49.0 或更高的 Class 文件，那么它必须正确的识别和读取这些 Class 文件中的 Signature（§4.7.9），RuntimeVisibleAnnotations（§4.7.16）和 RuntimeInvisibleAnnotations（§4.7.17）结构。

所有 Java 虚拟机实现都必须默认忽略 field_info 结构中 attributes 表所不可识别的成员。本规范中没有定义的属性不可影响 Class 文件的语义，它们只能提供附加描述信息（§4.7.1）。

4.6 方法

所有方法(Method)，包括实例初始化方法和类初始化方法(§2.9)在内，都由 method_info 结构所定义。在一个 Class 文件中，不会有两个方法同时具有相同的方法名和描述符(§ 4.3.3)。

method_info 结构格式如下：

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

method_info 结构各项的说明如下：

- access_flags

  access_flags 项的值是用于定义当前方法的访问权限和基本属性的掩码标志，access_flags 的取值范围和相应含义见表 4.5 所示。

表 4.5 方法 access_flags 标记列表

标记名	值	说明

ACC_PUBLIC	0x0001	public，方法可以从包外访问
ACC_PRIVATE	0x0002	private，方法只能本类中访问
ACC_PROTECTED	0x0004	protected，方法在自身和子类可以访问
ACC_STATIC	0x0008	static，静态方法
ACC_FINAL	0x0010	final，方法不能被重写（覆盖）
ACC_SYNCHRONIZED	0x0020	synchronized，方法由管程同步
ACC_BRIDGE	0x0040	bridge，方法由编译器产生
ACC_VARARGS	0x0080	表示方法带有变长参数
ACC_NATIVE	0x0100	native，方法引用非 java 语言的本地方法
ACC_ABSTRACT	0x0400	abstract，方法没有具体实现
ACC_STRICT	0x0800	strictfp，方法使用 FP-strict 浮点格式
ACC_SYNTHETIC	0x1000	方法在源文件中不出现，由编译器产生

ACC_VARARGS 标志是用于说明方法在源码层的参数列表是否变长的。如果是变长的，则在编译时，方法的 ACC_VARARGS 标志设置 1，其余的方法 ACC_VARARGS 标志设置为 0。ACC_BRIDGE 标志用于说明这个方法是由编译生成的桥接方法[①]。

如果方法设置了 ACC_SYNTHETIC 标志，怎说明这个方法是由编译器生成的并且不会在源代码中出现，少量的例外情况将在 4.7.8 节“Synthetic 属性”中提到。

Class 文件中的方法可以设置多个表 4.5 中的标志，但是有些标志是互斥的：一个方法只能设置 ACC_PRIVATE，ACC_PROTECTED 和 ACC_PUBLI 三个标志中的一个标志；如果一个方法被设置 ACC_ABSTRACT 标志，则这个方法不能被设置 ACC_FINAL，ACC_NATIVE，ACC_PRIVATE，ACC_STATIC，ACC_STRICT 和 ACC_SYNCHRONIZED 标志（JLS §8.4.3.1，JLS §8.4.3.3，JLS §8.4.3.4）。

所有的接口方法必须被设置 ACC_ABSTRACT 和 ACC_PUBLIC 标志；还可以选择设置 ACC_VARARGS，ACC_BRIDGE 和 ACC_SYNTHETIC 标志，但是不能再设置表 4.5 中其它的标志了（JLS §9.4）。

实例初始化方法（§2.9）只能被 ACC_PRIVATE，ACC_PROTECTED 和 ACC_PUBLIC 中的一个标志；还可以设置 ACC_STRICT， ACC_VARARGS 和 ACC_SYNTHETIC 标志，

[①] 译者注：桥接方法是 JDK 1.5 引入泛型后，为了使 Java 的范型方法生成的字节码和 1.5 版本前的字节码相兼容，由编译器自动生成的方法。

但是不能再设置表 4.5 中的其它标志了。

类初始化方法（§2.9）由 Java 虚拟机隐式自动调用，它的 access_flags 项的值除了 ACC_STRICT 标志，其它的标志都将被忽略。

在表 4.5 中没有出现的 access_flags 项值为未来扩充而预留，在生成的 Class 文件中应被设置成 0，Java 虚拟机实现应该忽略它们。

- name_index

  name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，它要么表示初始化方法的名字（<init>或<clinit>），要么表示一个方法的有效的非全限定名（§4.2.2）

- descriptor_index

  descriptor_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的方法的描述符（§4.3.3）

  注意：本规范在未来的某个版本中可能会要求当 access_flags 项的 ACC_VARARGS 标志被设置时，方法描述符中的最后一个参数必须是数组类型。

- attributes_count

  attributes_count 的项的值表示这个方法的附加属性（§4.7）的数量。

- attributes[]

  attributes 表的每一个成员的值必须是 attribute（§4.7）结构，一个方法可以有任意个与之相关的属性。

  本规范所定义的 method_info 结构中，属性表可出现的成员有：Code（§4.7.3），Exceptions（§4.7.5），Synthetic（§4.7.8），Signature（§4.7.9），Deprecated（§4.7.15），untimeVisibleAnnotations（§4.7.16），RuntimeInvisibleAnnotations（§4.7.17），RuntimeVisibleParameterAnnotations（§4.7.18），RuntimeInvisibleParameterAnnotations（§4.7.19）和 AnnotationDefault（§4.7.20）结构。

  Java 虚拟机实现必须正确识别和读取 method_info 结构中的属性表的 Code（§4.7.3）和 Exceptions（§4.7.5）属性。如果 Java 虚拟机实现支持版本为 49.0 或更高的 Class 文件，那么它必须正确识别和读取这些 Class 文件的 Signature（§

4.7.9），RuntimeVisibleAnnotations（§4.7.16），RuntimeInvisibleAnnotations（§4.7.17），RuntimeVisibleParameterAnnotations（§4.7.18），RuntimeInvisibleParameterAnnotations（§4.7.19）和 AnnotationDefault（§4.7.20）属性。

所有 Java 虚拟机实现必须默认忽略 method_info 结构中 attributes 表所不可识别的成员。本规范中没有定义的属性不可影响 Class 文件的语义，它们只能提供附加描述信息（§4.7.1）。

4.7 属性

属性(Attributes)在 Class 文件格式中的 ClassFile（§4.1)结构、field_info （§4.5）结构，method_info（§4.6）结构和 Code_attribute（§4.7.3）结构都有使用，所有属性的通用格式如下：

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

对于任意属性，attribute_name_index 必须是对当前 Class 文件的常量池的有效 16 位无符号索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示当前属性的名字。attribute_length 项的值给出了跟随其后的字节的长度，这个长度不包括 attribute_name_index 和 attribute_name_index 项的 6 个字节。

有些属性因 Class 文件格式规范所需，已被预先定义好。这些属性在表 4.6 中列出，同时，被列出的信息还包括它们首次出现的 Class 文件版本和 Java SE 版本号。在本规范定义的环境中，也就是已包含这些预定义属性的 Class 文件中，它们的属性名称被保留，不能再被属性表中其他的自定义属性所使用。

表 4.6 被预定义的 Class 文件属性

属性名	Java SE	Class 文件
ConstantValue（§4.7.2）	1.0.2	45.3

Code（§4.7.3）	1.0.2	45.3
StackMapTable（§4.7.4）	6	50.0
Exceptions（§4.7.5）	1.0.2	45.3
InnerClasses（§4.7.6）	1.1	45.3
EnclosingMethod（§4.7.7）	5.0	49.0
Synthetic（§4.7.8）	1.1	45.3
Signature（§4.7.9）	5.0	49.0
SourceFile（§4.7.10）	1.0.2	45.3
SourceDebugExtension（§4.7.11）	5.0	49.0
LineNumberTable（§4.7.12）	1.0.2	45.3
LocalVariableTable（§4.7.13）	1.0.2	45.3
LocalVariableTypeTable（§4.7.14）	5.0	49.0
Deprecated（§4.7.15）	1.1	45.3
RuntimeVisibleAnnotations（§4.7.16）	5.0	49.0
RuntimeInvisibleAnnotations（§4.7.17）	5.0	49.0
RuntimeVisibleParameterAnnotations（§4.7.18）	5.0	49.0
RuntimeInvisibleParameterAnnotations（§4.7.19）	5.0	49.0
AnnotationDefault（§4.7.20）	5.0	49.0
BootstrapMethods（§4.7.21）	7	51.0

- Java 虚拟机实现的 Class 文件加载器（Class File Reader）必须正确的识别和读取 ConstantValue，Code 和 Exceptions 属性；同样，Java 虚拟机也必须能正确的解析它们的语义。
- InnerClasses，EnclosingMethod 和 Synthetic 属性必须被 Class 文件加载器正确的识别并读入，它们用于实现 Java 平台的类库（§2.12）。
- 如果 Java 虚拟机实现支持的 Class 文件的版本号为 49.0 或更高时，它的 Class 文件加载器必须能正确的识别并读取 Class 文件中的 RuntimeVisibleAnnotations，RuntimeInvisibleAnnotations，RuntimeVisibleParameterAnnotations，RuntimeInvisibleParameterAnnotations 和 AnnotationDefault 属性，它们用于实现 Java 平台类库（§2.12）。

- 如果 Java 虚拟机实现支持的 Class 文件的版本号为 49.0 或更高时，它的 Class 文件加载器必须正确的识别和读取 Class 文件中的 Signature 属性。
- 如果 Java 虚拟机实现支持的 Class 文件的版本号为 50.0 或更高时，它的 Class 文件加载器必须正确的识别和读取 StackMapTable 属性。
- 如果 Java 虚拟机实现支持的 Class 文件的版本号为 51.0 或更高时，它的 Class 文件加载器必须正确的识别和读取 BootstrapMethods 属性。
- 对于剩余的预定义属性的使用不受限制；如果剩余的预定义属性包含虚拟机可识别的信息，Class 文件加载器就可以选择使用这些信息，否则可以选择忽略它们。

4.7.1 自定义和命名新的属性

《Java 虚拟机规范》允许编译器在 Class 文件的属性表中定义和发布新的属性。Java 虚拟机实现允许识别并使用 Class 文件结构属性表中出现的新属性。但是，所有在《Java 虚拟机规范》中没有定义的属性，不能影响类或接口类型的语义。Java 虚拟机实现必须忽略它不能识别的自定义属性。

例如，编译器可以定义新的属性用于支持与特定发行者相关（Vendor-Specific）的调式，而不影响其它 Java 虚拟机实现。因为其他 Java 虚拟机实现必须忽略它不识别的属性，所以那些特定发行者相关的虚拟机实现所使用的 Class 文件也可以被别的 Java 虚拟机实现使用，即使这些 Class 文件包含的附加调式信息不能被它们所用。

《Java 虚拟机规范》明确禁止 Java 虚拟机实现仅仅因为 Class 文件包含新属性而抛出异常或其他拒绝使用 Class 文件。当然，如果 Class 文件没有包含所需的属性，某些工具则可能无法正确操作这个 Class 文件。

当两个不同的属性使用了相同的属性名且长度也相同，无论虚拟机识别其中哪一个都会引起冲突。本规范定义之外的自定义属性，必须按照《Java 语言规范（Java SE7 版）》（§7.7）中所规定的包命名方式来命名。

本规范在未来的版本中可能会定义增加预定义的属性。

4.7.2 ConstantValue 属性

ConstantValue 属性是定长属性，位于 field_info（§4.5）结构的属性表中。ConstantValue 属性表示一个常量字段的值。在一个 field_info 结构的属性表中最多只能有一个 ConstantValue 属性。如果该字段为静态类型（即 field_info 结构的 access_flags 项（表 4.4）设置了 ACC_STATIC 标志），则说明这个 field_info 结构表示的常量字段值将被分配为它的 ConstantValue 属性表示的值，这个过程也是类或接口申明的常量字段（Constant Field（§5.5））初始化的一部分。这个过程发生在引用类或接口的类初始化方法（§2.9）执行之前。

如果 field_info 结构表示的非静态字段包含了 ConstantValue 属性，那么这个属性必须被虚拟机所忽略。所有 Java 虚拟机实现必须能够识别 ConstantValue 属性。

ConstantValue 属性的格式如下：

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

ConstantValue 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值，必须是一个对常量池的有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“ConstantValue”。

- attribute_length

  ConstantValue_attribute 结构的 attribute_length 项的值固定为 2。

- constantvalue_index

  constantvalue_index 项的值，必须是一个对常量池的有效索引。常量池在该索引处的项给出该属性表示的常量值。常量池的项的类型表示的字段类型如表 4.7 所示。

表 4.7 ConstantValue 属性的类型

字段类型	项类型
long	CONSTANT_Long

float	CONSTANT_Float
double	CONSTANT_Double
int,short,char,byte,boolean	CONSTANT_Integer
String	CONSTANT_String

4.7.3 Code 属性

Code 属性是一个变长属性，位于 method_info（§4.6）结构的属性表。一个 Code 属性只为唯一一个方法、实例类初始化方法或类初始化方法（§2.9）保存 Java 虚拟机指令及相关辅助信息。 所有 Java 虚拟机实现都必须能够识别 Code 属性。如果方法被声明为 native 或者 abstract 类型，那么对应的 method_info 结构不能有明确的 Code 属性，其它情况下，method_info 有必须有明确的 Code 属性。

Code 属性的格式如下：

```
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 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“Code”。

- attribute_length

  attribute_length 项的值表示当前属性的长度，不包括开始的 6 个字节。

- max_stack

  max_stack 项的值给出了当前方法的操作数栈在运行执行的任何时间点的最大深度（§2.6.2）。

- max_locals

  max_locals 项的值给出了分配在当前方法引用的局部变量表中的局部变量个数，包括调用此方法时用于传递参数的局部变量。

  long 和 double 型的局部变量的最大索引是 max_locals-2，其它类型的局部变量的最大索引是 max_locals-1.

- code_length

  code_length 项给出了当前方法的 code[]数组的字节数[①]，code_length 的值必须大于 0，即 code[]数组不能为空。

- code[]

  code[]数组给出了实现当前方法的 Java 虚拟机字节码。

  code[]数组以按字节寻址的方式读入机器内存，如果 code[]数组的第一个字节是按以 4 字节边界对齐的话，那么 tableswitch 和 lookupswitch 指令中所有涉及到的 32 位偏移量也都是按 4 字节长度对齐的（关于 code[]数组边界对齐对字节码的影响，请参考相关的指令描述）。

  本规范对关于 code[]数组内容的详细约束有很多，将在后面单独章节（§4.9）中列出。

- exception_table_length

  exception_table_length 项的值给出了 exception_table[]数组的成员个数量。

- exception_table[]

  exception_table[]数组的每个成员表示 code[]数组中的一个异常处理器（Exception Handler）。exception_table[]数组中，异常处理器顺序是有意义的（不能随意更改），详细内容见 2.10 节。

  exception_table[]数组包含如下 4 项：

[①] 译者注：请注意，由于部分指令在 code[]数组中存有直接操作数，换句话说，有一些字节码指令的实际长度是超过一个字节的，因此此处字节数长度 code_length 并不等同于 code[]数组的成员个数。

- start_pc 和 end_pc

  start_pc 和 end_pc 两项的值表明了异常处理器在 code[]数组中的有效范围。start_pc 必须是对当前 code[]数组中某一指令的操作码的有效索引，end_pc 要么是对当前 code[]数组中某一指令的操作码的有效索引，要么等于 code_length 的值，即当前 code[]数组的长度。start_pc 的值必须比 end_pc 小。

  当程序计数器在范围[start_pc, end_pc)内时，异常处理器就将生效。即设 x 为异常句柄的有效范围内的值，x 满足：start_pc ≤ x < end_pc。

  实际上，end_pc 值本身不属于异常处理器的有效范围这点属于 Java 虚拟机历史上的一个设计缺陷：如果 Java 虚拟机中的一个方法的 code 属性的长度刚好是 65535 个字节，并且以一个 1 个字节长度的指令结束，那么这条指令将不能被异常处理器所处理。不过编译器可以通过限制任何方法、实例初始化方法或类初始化方法的 code[]数组最大长度为 65534，这样可以间接弥补这个 BUG。

- handler_pc

  handler_pc 项表示一个异常处理器的起点，它的值必须同时是一个对当前 code[]数组中某一指令的操作码的有效索引。

- catch_type

  如果 catch_type 项的值不为 0，那么它必须是对常量池的一个有效索引，常量池在该索引处的项必须是 CONSTANT_Class_info（§4.4.1）结构，表示当前异常处理器指定需要捕捉的异常类型。只有当抛出的异常是指定的类或其子类的实例时，异常处理器才会被调用。

  如果 catch_type 项的值如果为 0，那么这个异常处理器将会在所有异常抛出时都被调用。这可以用于实现 finally 语句（§3.13，“编译 finally”）。

- attributes_count

  attributes_count 项的值给出了 Code 属性中 attributes 表的成员个数。

- attributes[]

  属性表的每个成员的值必须是 attribute 结构（§4.7）。一个 Code 属性可以有任意数量的可选属性与之关联。

  本规范中定义的、可以出现在 Code 属性的属性表中的成员只能是 LineNumberTable（§4.7.12），LocalVariableTable（§4.7.13），LocalVariableTypeTable

（§4.7.14）和 StackMapTable（§4.7.4）属性。

如果一个 Java 虚拟机实现支持的 Class 文件版本号为 50.0 或更高，那么它必须正确的识别和读取 Code 属性的属性表出现的 StackMapTable（§4.7.4）属性。

Java 虚拟机实现必须自动忽略 Code 属性的属性表数组中出现的所有它不能识别属性。本规范中没有定义的属性不可影响 Class 文件的语义，只能提供附加描述信息（§4.7.1）。

4.7.4 StackMapTable 属性

StackMapTable 属性是一个变长属性，位于 Code（§4.7.3）属性的属性表中。这个属性会在虚拟机类加载的类型阶段（§4.10.1）被使用。

StackMapTable 属性包含 0 至多个栈映射帧（Stack Map Frames），每个栈映射帧都显式或隐式地指定了一个字节码偏移量，用于表示局部变量表和操作数栈的验证类型（Verification Types §4.10.1）。

类型检测器（Type Checker）会检查和处理目标方法的局部变量和操作数栈所需要的类型。本章节中，一个存储单元（Location）的含义是唯一的局部变量或操作数栈项。

我们还将用到术语“栈映射帧”（Stack Map Frame）和“类型状态”（Type State）来描述如何从方法的局部变量和操作数栈的存储单元映射到验证类型（Verification Types）。当描述 Class 文件侧的映射时，我们通常使用的术语是“栈映射帧”，而当描述类型检查器侧的映射关系时，我们通常使用的术语是“类型状态”。

在版本号大于或等于 50.0 的 Class 文件中，如果方法的 Code 属性中没有附带 StackMapTable 属性，那就意味着它带有一个隐式的 StackMap 属性。这个 StackMap 属性的作用等同于 number_of_entries 值为 0 的 StackMapTable 属性。一个方法的 Code 属性最多只能有一个 StackMapTable 属性，否则将抛出 ClassFormatError 异常。

StackMapTable 属性的格式如下：

```
StackMapTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_entries;
    stack_map_frame entries[number_of_entries];
```

```
}
```

StackMapTable 结构项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的有效索引，常量池在该索引的项处必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示“StackMapTable”字符串。

- attribute_length

  attribute_length 项的值表示当前属性的长度，不包括开始的 6 个字节。

- number_of_entries

  number_of_entries 项的值给出了 entries 表中的成员数量。Entries 表的每个成员是都是一个 stack_map_frame 结构的项。

- entries[]

  entries 表给出了当前方法所需的 stack_map_frame 结构。

每个 stack_map_frame 结构都使用一个特定的字节偏移量来表示类型状态。每个帧类型（Frame Type）都显式或隐式地标明一个 offset_delta（增量偏移量）值，用于计算每个帧在运行时的实际字节码偏移量。使用时帧的字节偏移量计算方法为：前一帧的字节码偏移量（Bytecode Offset）加上 offset_delta 的值再加 1，如果前一个帧是方法的初始帧（Initial Frame），那这时候字节码偏移量就是 offset_delta。

只要保证栈映射帧有正确的存储顺序，在类型检查时我们就可以使用增量偏移量而不是实际的字节码偏移量。此外，由于堆每一个帧都使用了 offset_delta+1 的计算方式，我们可以确保偏移量不会重复。

在 Code 属性的 code[]数组项中，如果偏移量 i 的位置是某条指令的起点，同时这个 Code 属性包含有 StackMapTable 属性，它的 entries 项中也有一个适用于地址偏移量 i 的 stack_map_frame 结构，那我们就说这条指令拥有一个与之相对应的栈映射帧。

stack_map_frame 结构的第一个字节作为类型标记（Tag），第一个字节后会跟随 0 或多个字节用于说明更多信息，这些信息因类型标记的不同而变化。

一个栈映射帧可以包含若干种帧类型（Frame Types）：

```
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
```

```
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
}
```

所有的帧类型，包括 full_frame，它们的部分语义会依赖于前置帧，这点使得确定基准帧（Very First Frame）变得尤为重要，方法的初始帧是隐式的，它通过方法描述符计算得出，详细信息请参考 methodInitialStackFrame（§4.10.1.3.3）。

帧类型 same_frame 的类型标记（frame_type）的取值范围是 0 至 63，如果类型标记所确定的帧类型是 same_frame 类型，则明当前帧拥有和前一个栈映射帧完全相同的 locals[]数组，并且对应的 stack 项的成员个数为 0。当前帧的 offset_delta 值就使用 frame_type 项的值来表示[①]。

```
same_frame {
    u1 frame_type = SAME; /* 0-63 */
}
```

帧类型 same_locals_1_stack_item_frame 的类型标记的取值范围是 64 至 127。如果类型标记所确定的帧类型是 same_locals_1_stack_item_frame 类型，则说明当前帧拥有和前一个栈映射帧完全相同的 locals[]数组，同时对应的 stack[]数组的成员个数为 1。当前帧的 offset_delta 值为 frame_type-64。并且有一个 verification_type_info 项跟随在此帧类型之后，用于表示那一个 stack 项的成员。

```
same_locals_1_stack_item_frame {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM;/* 64-127 */
    verification_type_info stack[1];
}
```

范围在 128 至 246 的类型标记值是为未来使用而预留的。

帧类型 same_locals_1_stack_item_frame_extended 由值为 247 的类型标记表示，它表明当前帧拥有和前一个栈映射帧完全相同的 locals[]数组，同时对应的 stack[]数组的成员个数为 1。当前帧的 offset_delta 的值需要由 offset_delta 项明确指定。有一个 stack[]

[①] 译者注：此处描述的“stack”、“locals”是 StackMapTable 属性中的项，它们与运行时栈帧中的操作数栈、局部变量表有映射关系，但并非同一样东西。原文中的对它们的描述为“stack”和“operand stack”、“locals”和“local variables”，译文中，指代属性项时使用 locals[]数组、stack 表来表示，而提到运行时栈帧时，则会明确翻译为操作数栈、局部变量表，也请读者注意根据上下文注意区分。

数组的成员跟随在 offset_delta 项之后。

```
same_locals_1_stack_item_frame_extended {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED;/* 247 */
    u2 offset_delta;
    verification_type_info stack[1];
}
```

帧类型 chop_frame 的类型标记的取值范围是 248 至 250。如果类型标记所确定的帧类型是为 chop_frame，则说明对应的操作数栈为空，并且拥有和前一个栈映射帧相同的 locals[]数组，不过其中的第 k 个之后的 locals 项是不存在的。k 的值由 251-frame_type 确定。

```
chop_frame {
    u1 frame_type = CHOP; /* 248-250 */
    u2 offset_delta;
}
```

帧类型 same_frame_extended 由值为 251 的类型标记表示。如果类型标记所确定的帧类型是 same_frame_extended 类型，则说明当前帧有拥有和前一个栈映射帧的完全相同的 locals[]数组，同时对应的 stack[]数组的成员数量为 0。

```
same_frame_extended {
    u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
    u2 offset_delta;
}
```

帧类型 append_frame 的类型标记的取值范围是 252 至 254。如果类型标记所确定的帧类型为 append_frame，则说明对应操作数栈为空，并且包含和前一个栈映射帧相同的 locals[]数组，不过还额外附加 k 个的 locals 项。k 值为 frame_type-251。

```
append_frame {
    u1 frame_type = APPEND; /* 252-254 */
    u2 offset_delta;
    verification_type_info locals[frame_type - 251];
}
```

在 locals[]数组中，索引为 0 的（第一个）成员表示第一个添加的局部变量。如果要从条件“locals[M]表示第 N 个局部变量”中推导出结论“locals[M+1]就表示第 N+1 个局部变量”的话，那就意味着 locals[M]一定是下列结构之一：

- Top_variable_info
- Integer_variable_info

- Float_variable_info
- Null_variable_info
- UninitializedThis_variable_info
- Object_variable_info
- Uninitialized_variable_info

否则，locals[M+1]就将表示第 N+2 个局部变量。对于任意的索引 i，locals[i]所表示的局部变量的索引都不能大于此方法的局部变量表的最大索引值。

在 stack[]数组中，索引为 0 的（第一个）成员表示操作数栈的最底部的元素，之后的成员依次靠近操作数栈的顶部。操作数栈栈底的元素对应的索引为 0，我们称之为元素 0，之后元素依次是元素 1、元素 2 等。如果要从条件“stack[M]表示第 N 个元素”中推导出结论“stack[M+1]表示第 N+1 个元素”的话，那就意味着 stack[M]一定是下列结构之一：

- Top_variable_info
- Integer_variable_info
- Float_variable_info
- Null_variable_info
- UninitializedThis_variable_info
- Object_variable_info
- Uninitialized_variable_info

否则，stack[M+1]将表示第 N+2 个元素，对于任意的索引 i，stack[i]所表示的栈元素索引都不能大于此方法的操作数的最大深度。

verification_type_info 结构的第一个字节 tag 作为类型标记，之后跟随 0 至多个字节表示由 tag 类型所决定的信息。每个 verification_type_info 结构可以描述 1 个至 2 个存储单元的验证类型信息。

```
union verification_type_info {
    Top_variable_info;
    Integer_variable_info;
    Float_variable_info;
    Long_variable_info;
    Double_variable_info;
    Null_variable_info;
    UninitializedThis_variable_info;
```

```
    Object_variable_info;
    Uninitialized_variable_info;
}
```

Top_variable_info 类型说明这个局部变量拥有验证类型 top(T)。

```
Top_variable_info {
    u1 tag = ITEM_Top; /* 0 */
}
```

Integer_variable_info 类型说明这个局部变量包含验证类型 int

```
Integer_variable_info {
    u1 tag = ITEM_Integer; /* 1 */
}
```

Float_variable_info 类型说明局部变量包含验证类型 float

```
Float_variable_info {
    u1 tag = ITEM_Float; /* 2 */
}
```

Long_variable_info 类型说明存储单元包含验证类型 long，如果存储单元是局部变量，则要求：

- 不能是最大索引值的局部变量。
- 按顺序计数的下一个局部变量包含验证类型 T

如果单元存储是操作数栈成员，则要求：

- 当前的存储单元不能在栈顶。
- 靠近栈顶方向的下一个存储单元包含验证类型 T。

Long_variable_info 结构在局部变量表或操作数栈中占用 2 个存储单元。

```
Long_variable_info {
    u1 tag = ITEM_Long; /* 4 */
}
```

Double_variable_info 类型说明存储单元包含验证类型 double。如果存储单元是局部变量，则要求：

- 不能是最大索引值的局部变量。
- 按顺序计数的下一个局部变量包含验证类型 T

如果单元存储是操作数栈成员，则要求：

- 当前的存储单元不能在栈顶。

- 靠近栈顶方向的下一个存储单元包含验证类型 T。

Double_variable_info 结构在局部变量表或操作数栈中占用 2 个存储单元。

```
Double_variable_info {
    u1 tag = ITEM_Double; /* 3 */
}
```

Null_variable_info 类型说明存储单元包含验证类型 null。

```
Null_variable_info {
    u1 tag = ITEM_Null; /* 5 */
}
```

UninitializedThis_variable_info 类型说明存储单元包含验证类型 uninitializedThis。

```
UninitializedThis_variable_info {
    u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

Object_variable_info 类型说明存储单元包含某个 Class 的实例。由常量池在 cpool_index 给出的索引处的 CONSTANT_CLASS_Info（§4.4.1）结构表示。

```
Object_variable_info {
    u1 tag = ITEM_Object; /* 7 */
    u2 cpool_index;
}
```

Uninitialized_variable_info 说明存储单元包含验证类型 uninitialized(offset)。offset 项给出了一个偏移量，表示在包含此 StackMapTable 属性的 Code 属性中，new 指令创建的对象所存储的位置。

```
Uninitialized_variable_info {
    u1 tag = ITEM_Uninitialized /* 8 */
    u2 offset;
}
```

4.7.5 Exceptions 属性

Exceptions 属性是一个变长属性，它位于 method_info（§4.6）结构的属性表中。Exceptions 属性指出了一个方法需要检查的可能抛出的异常。一个 method_info 结构中最多

只能有一个 Exceptions 属性。

Exceptions 属性格式如下：

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

Exceptions_attribute 格式的项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串"Exceptions"。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。

- number_of_exceptions

  number_of_exceptions 项的值给出了 exception_index_table[]数组中成员的数量。

- exception_index_table[]

  exception_index_table[]数组的每个成员的值都必须是对常量池的有效索引。常量池在这些索引处的成员必须都是 CONSTANT_Class_info（§4.4.1）结构，表示这个方法声明要抛出的异常的类的类型。

一个方法如果要抛出异常，必须至少满足下列三个条件中的一个：

- 要抛出的是 RuntimeException 或其子类的实例。
- 要抛出的是 Error 或其子类的实例。
- 要抛出的是在 exception_index_table[]数组中申明的异常类或其子类的实例。

这些要求没有在 Java 虚拟机中进行强制检查，它们只在编译时进行强制检查。

4.7.6 InnerClasses 属性

InnerClasses 属性是一个变长属性，位于 ClassFile（§4.1）结构的属性表。本小结为

了方便说明特别定义一个表示类或接口的 Class 格式为 C。如果 C 的常量池中包含某个 CONSTANT_Class_info 成员，且这个成员所表示的类或接口不属于任何一个包，那么 C 的 ClassFile 结构的属性表中就必须含有对应的 InnerClasses 属性。

InnerClasses 属性是在 JDK 1.1 中为了支持内部类和内部接口而引入的。

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    {       u2 inner_class_info_index;
            u2 outer_class_info_index;
            u2 inner_name_index;
            u2 inner_class_access_flags;
    } classes[number_of_classes];
}
```

- attribute_name_index

  attribute_name_index 项的值必须是一个对常量池的有效引用。常量池在该索引处的成员必须是 CONSTANT_Utf8_info(§4.4.7)结构,表示字符串"InnerClasses"。

- attribute_length

  attribute_length 项的值给出了这个属性的长度，不包括前 6 个字节。

- number_of_classes

  number_of_classes 项的值表示 classes[]数组的成员数量。

- classes[]

  常量池中的每个 CONSTANT_Class_info 结构如果表示的类或接口并非某个包的成员，则每个类或接口在 classes[]数组中都有一个成员与之对应。

如果 Class 中包含某些类或者接口，那么它的常量池（以及对应的 InnerClasses 属性）必须包含这些成员，即使某些类或者接口没有被这个 Class 使用过。

这条规则暗示着内部类或内部接口成员在其被定义的外部类（Enclosing Class）中都会包含有它们的 InnerClasses 信息。

classes[]数组中每个成员包含以下 4 个项：

- inner_class_info_index

  inner_class_info_index 项的值必须是一个对常量池的有效索引。常量池在该索引处的项必须是 CONSTANT_Class_info（§4.4.1）结构，表示接口 C。当前元素的另

外 3 项都用于描述 C 的信息。

- outer_class_info_index

  如果 C 不是类或接口的成员（也就是 C 为顶层类或接口（JLS §7.6）、局部类（JLS §14.3）或匿名类（JLS §15.9.5）），那么 outer_class_info_index 项的值为 0，否则这个项的值必须是对常量池的一个有效索引，常量池在该索引处的项必须是 CONSTANT_Class_info（§4.4.1）结构，代表一个类或接口，C 为这个类或接口的成员。

- inner_name_index

  如果 C 是匿名类（JLS §15.9.5），inner_name_index 项的值则必须为 0。否则这个项的值必须是对常量池的一个有效索引，常量池在该索引处的项必须 CONSTANT_Utf8_info（§4.4.7）结构，表示 C 的 Class 文件在对应的源文件中定义的 C 的原始简单名称（Original Simple Name）

- inner_class_access_flags

  inner_class_access_flags 项的值是一个掩码标志，用于定义 Class 文件对应的源文件中 C 的访问权和基本属性。用于编译器在无法访问源文件时可以恢复 C 的原始信息。inner_class_access_flags 项的取值范围和说明见表 4.8。

表 4.8 内部类访问全和基础属性标志

标记名	值	含义
ACC_PUBLIC	0x0001	源文件定义 public
ACC_PRIVATE	0x0002	源文件定义 private
ACC_PROTECTED	0x0004	源文件定义 protected
ACC_STATIC	0x0008	源文件定义 static
ACC_FINAL	0x0010	源文件定义 final
ACC_INTERFACE	0x0200	源文件定义 interface
ACC_ABSTRACT	0x0400	源文件定义 abstract
ACC_SYNTHETIC	0x1000	声明 synthetic，非源文件定义
ACC_ANNOTATION	0x2000	声明 annotation
ACC_ENUM	0x4000	声明 enum

所有表 4.8 中没有定义的 inner_class_access_flags 项，都是为未来使用而预留

的。这些字节在通常的 Class 文件中应被设置为 0，Java 虚拟机实现应忽略它们。

如果 Class 文件的版本号为 51.0 或更高，属性表中有 InnerClasses 属性，并且 InnerClasses 属性的 classes[]数组中的 inner_name_index 项的值为 0，则它对应的 outer_class_info_index 项的值也必须为 0。

Oracle 的 Java 虚拟机实现不会检查 InnerClasses 属性和某个属性引用的类或接口的 Class 文件的一致性。

4.7.7 EnclosingMethod 属性

EnclosingMethod 属性是可选的定长属性，位于 ClassFile（§4.1）结构的属性表。当且仅当 Class 为局部类或者匿名类时，才能具有 EnclosingMethod 属性。一个类最多只能有一个 EnclosingMethod 属性。

EnclosingMethod 属性格式如下：

```
EnclosingMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 class_index
    u2 method_index;
}
```

EnclosingMethod_attribute 结构的项说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是一个对常量池的有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串 "EnclosingMethod"。

- attribute_length

  attribute_length 项的值固定为 4。

- class_index

  class_index 项的值必须是一个对常量池的有效索引。常量池在该索引出的项必须是 CONSTANT_Class_info（§4.4.1）结构，表示包含当前类声明的最内层类。

- method_index

如果当前类不是在某个方法或初始化方法中直接包含（Enclosed），那么 method_index 值为 0，否则 method_index 项的值必须是对常量池的一个有效索引，常量池在该索引处的成员必须是 CONSTANT_NameAndType_info（§4.4.6）结构，表示由 class_index 属性引用的类的对应方法的方法名和方法类型。Java 编译器有责任在语法上保证通过 method_index 确定的方法是语法上最接近那个包含 EnclosingMethod 属性的类的方法（Closest Lexically Enclosing Method）。

4.7.8 Synthetic 属性

Synthetic 属性是定长属性，位于 ClassFile（§4.1）中的属性表。如果一个类成员没有在源文件中出现，则必须标记带有 Synthetic 属性，或者设置 ACC_SYNTHETIC 标志。唯一的例外是某些与人工实现无关的、由编译器自动产生的方法，也就是说，Java 编程语言的默认的实例初始化方法（无参数的实例初始化方法）、类初始化方法，以及 Enum.values()和 Enum.valueOf()等方法是不用使用 Synthetic 属性或 ACC_SYNTHETIC 标记的。

Synthetic 属性是在 JDK 1.1 中为了支持内部类或接口而引入的。

Synthetic 属性的格式如下：

```
Synthetic_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

Synthetic 结构的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引，常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“Synthetic”。

- attribute_length

  attribute_length 项的值固定为 0。

4.7.9 Signature 属性

Signature 属性是可选的定长属性，位于 ClassFile（§4.1），field_info（§4.5）或 method_info（§4.6）结构的属性表中。在 Java 语言中，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则 Signature 属性会为它记录泛型签名信息。

Signature 属性格式如下：

```
Signature_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 signature_index;
}
```

Signature_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是一个对常量池的有效索引。常量池在索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“Signature”。

- attribute_length

  signature_attribute 结构的 attribute_length 项的值必须为 2。

- signature_index

  signature_index 项的值必须是一个对常量池的有效索引。常量池在该索引处的项必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示类签名或方法类型签名或字段类型签名：如果当前的 Signature 属性是 ClassFile 结构的属性，则这个结构表示类签名，如果当前的 Signature 属性是 method_info 结构的属性，则这个结构表示方法类型签名，如果当前 Signature 属性是 field_info 结构的属性，则这个结构表示字段类型签名。

4.7.10 SourceFile 属性

SourceFile 属性是可选定长字段，位于 ClassFile（§4.1）结构的属性表。一个 ClassFile 结构中的属性表最多只能包含一个 SourceFile 属性。

SourceFile 属性格式如下：

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

SourceFile_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是一个对常量池的有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“SourceFile”。

- sourcefile_index

  sourcefile_index 项的值必须是一个对常量池的有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个字符串。

  sourcefile_index 项引用字符串表示被编译的 Class 文件的源文件的名字。不包括源文件所在目录的目录名，也不包括源文件的绝对路径名。平台相关（绝对路径名等）的附加信息必须是运行时解释器（Runtime Interpreter）或开发工具在文件名实际使用时提供。

4.7.11 SourceDebugExtension 属性

SourceDebugExtension 属性是可选属性，位于 ClassFile（§4.1）结构属性表。ClassFile 结构的属性表最多只能包含一个 SourceDebugExtension 属性。

SourceDebugExtension 属性的格式如下：

```
SourceDebugExtension_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 debug_extension[attribute_length];
}
```

SourceDebugExtension_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处

的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“SourceDebugExtension”。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。即 attribute_length 项的值是字节数组 debug_extension[]数组的长度

- debug_extension[]

  debug_extension[]数组用于保存扩展调试信息，扩展调试信息对于 Java 虚拟机来说没有实际的语义。这个信息用改进版的 UTF-8 编码的字符串（§4.4.7）表示，这个字符串不包含 byte 值为 0 的终止符。需要注意的是，debug_extension[]数组表示的字符串可以比 Class 实例对应的字符串更长。

4.7.12 LineNumberTable 属性

LineNumberTable 属性是可选变长属性，位于 Code（§4.7.3）结构的属性表。它被调试器用于确定源文件中行号表示的内容在 Java 虚拟机的 code[]数组中对应的部分。在 Code 属性的属性表中，LineNumberTable 属性可以按照任意顺序出现，此外，多个 LineNumberTable 属性可以共同表示一个行号在源文件中表示的内容，即 LineNumberTable 属性不需要与源文件的行一一对应。

LineNumberTable 属性格式如下：

```
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 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串

“LineNumberTable”。

- attribute_length

  attribute_length 给出了当前属性的长度，不包括开始的 6 个字节。

- line_number_table_length

  line_number_table_length 项的值给出了 line_number_table[]数组的成员个数。

- line_number_table[]

  line_number_table[]数组的每个成员都表明源文件中行号的变化在 code[]数组中都会有对应的标记点。line_number_table 的每个成员都具有如下两项：

  - start_pc

    start_pc 项的值必须是 code[]数组的一个索引，code[]数组在该索引处的字符表示源文件中新的行的起点。start_pc 项的值必须小于当前 LineNumberTable 属性所在的 Code 属性的 code_length 项的值。

  - line_number

    line_number 项的值必须与源文件的行数相匹配。

4.7.13 LocalVariableTable 属性

LocalVariableTable 是可选变长属性，位于 Code（§4.7.3）属性的属性表中。它被调试器用于确定方法在执行过程中局部变量的信息。在 Code 属性的属性表中，LocalVariableTable 属性可以按照任意顺序出现。Code 属性中的每个局部变量最多只能有一个 LocalVariableTable 属性。

LocalVariableTable 属性格式如下：

```
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 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“LocalVariableTable”。

- attribute_length

  attribute_length 项的值给出当前属性的长度，不包括开始的 6 个字节。

- local_variable_table_length

  local_variable_table_length 项的值给出了 local_variable_table[]数组的成员的数量。

- local_variable_table[]

  local_variable_table[]数组的每一个成员表示一个局部变量的值在 code[]数组中的偏移量范围。它同时也是用于从当前帧的局部变量表找出所需的局部变量的索引。local_variable_table[]数组每个成员都有如下 5 个项：

  - start_pc, length

    所有给定的局部变量的索引都在范围[start_pc, start_pc+length)中，即从 start_pc（包括自身值）至 start_pc+length（不包括自身值）。start_pc 的值必须是一个对当前 Code 属性的 code[]数组的有效索引，code[]数组在这个索引处必须是一条指令操作码。start_pc+length 要么是当前 Code 属性的 code[]数组的有效索引，code[]数组在该索引处必须是一条指令的操作码，要么是刚超过 code[]数组长度的最小索引值。

  - name_index

    name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个局部变量的有效的非全限定名（§4.2.2）。

  - descriptor_index

    descriptor_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的

成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示源程序中局部变量类型的字段描述符（§4.3.2）。

- index

  index 为此局部变量在当前栈帧的局部变量表中的索引。如果在 index 索引处的局部变量是 long 或 double 型，则占用 index 和 index+1 两个索引。

4.7.14 LocalVariableTypeTable 属性

LocalVariableTypeTable 属性是可选变长属性，位于 Code（§4.7.3）的属性表。它被用于给调试器确定方法在执行中局部变量的信息，在 Code 属性的属性表中，LocalVariableTable 属性可以按照任意顺序出现。Code 属性中的每个局部变量最多只能有一个 LocalVariableTable 属性。

LocalVariableTypeTable 属性和 LocalVariableTable 属性并不相同，LocalVariableTypeTable 提供签名信息而不是描述符信息。这仅仅对泛型类型有意义。泛型类型的属性会同时出现在 LocalVariableTable 属性和 LocalVariableTypeTable 属性中，其他的属性仅出现在 LocalVariableTable 属性表中。

LocalVariableTypeTable 属性格式如下：

```
   LocalVariableTypeTable_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 local_variable_type_table_length;
        {       u2 start_pc;
                u2 length;
                u2 name_index;
                u2 signature_index;
                u2 index;
        }local_variable_type_table[local_variable_type_table_length
];
   }
```

LocalVariableTypeTable_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处

的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“LocalVariableTypeTable”。

- attribute_length

  attribute_length 项的值给出当前属性的长度，不包括开始的 6 个字节。

- local_variable_type_table_length

  local_variable_type_table_length 项的值给出了 local_variable_type_table[]数组的成员的数量。

- local_variable_type_table[]

  local_variable_type_table[]数组的每一个成员表示一个局部变量的值在 code[]数组中的偏移量范围。它同时也是用于从当前帧的局部变量表找出所需的局部变量的索引。local_variable_type_table[]数组每个成员都有如下 5 个项：

  - start_pc, length

    所有给定的局部变量的索引都在范围[start_pc, start_pc+length)中，即从 start_pc（包括自身）至 start_pc+length（不包括自身）。start_pc 的值必须是一个对当前 Code 属性的 code[]数组的有效索引，code[]数组在这个索引处必须是一条指令操作码。start_pc+length 要么是当前 Code 属性的 code[]数组的有效索引，code[]数组在该索引处必须是一条指令的操作码，要么是刚超过 code[]数组长度的最小索引值。

  - name_index

    name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个局部变量的有效的非全限定名（§4.2.2）。

  - signature_index

    signature_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示给源程序中局部变量类型的字段签名（§4.3.4）。

  - index

    index 为此局部变量在当前栈帧的局部变量表中的索引。如果在 index 索引处的局部变量是 long 或 double 型，则占用 index 和 index+1 两个索引。

4.7.15 Deprecated 属性

Deprecated 属性是可选定长属性，位于 ClassFile（§4.1），field_info（§4.5）或 method_info（§4.6）结构的属性表中。类、接口、方法或字段都可以带有为 Deprecated 属性，如果类、接口、方法或字段标记了此属性，则说明它将会在后续某个版本中被取代。在运行时解释器或工具（譬如编译器）读取 Class 文件格式时，可以用 Deprecated 属性来告诉使用者避免使用这些类、接口、方法或字段，选择其他更好的方式。Deprecated 属性的出现不会修改类或接口的语义。

Deprecated 属性是在 JDK 1.1 为了支持注释中的关键词@deprecated 而引入的。

Deprecated 属性格式如下：

```
Deprecated_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
}
```

Deprecated_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“Deprecated”。

- attribute_length

  attribute_length 项的值固定为 0。

4.7.16 RuntimeVisibleAnnotations 属性

RuntimeVisibleAnnotations 属性是可变长属性，位于 ClassFile（§4.1），field_info（§4.5）或 method_info（§4.6）结构的属性表中。RuntimeVisibleAnnotations 用于保存 Java 语言中的类、字段或方法的运行时的可见注解（Annotations）。每个 ClassFile，field_info 和 method_info 结构最多只能含有一个 RuntimeVisibleAnnotations 属性为当前的程序元素保存所有的运行时可见的 Java 语言注

解。Java 虚拟机必须支持这些注解可被反射的 API 使用它们。

RuntimeVisibleAnnotations 属性格式如下：

```
RuntimeVisibleAnnotations_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 num_annotations;
     annotation annotations[num_annotations];
}
```

RuntimeVisibleAnnotations_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“RuntimeVisibleAnnotations”。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。

  attribute_length 项的值由当前结构的运行时可见注解的数量和值决定。

- num_annotations

  num_annotations 项的值给出了当前结构表示的运行时可见注解的数量。每个程序元素最多可能会被附加 65535 个运行时可见的 java 语言注解。

- annotations[]

  annotations[]数组的每个成员的值表示一个程序元素的唯一的运行时可见注解。

  annotation 结构的格式如下：

```
annotation {
     u2 type_index;
     u2 num_element_value_pairs;
     {       u2 element_name_index;
             element_value value;
     } element_value_pairs[num_element_value_pairs]
}
```

  annotation 结构各项的说明如下：

  - type_index

    type_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个字段描述符，

这个字段描述符表示一个注解类型，和当前 annotation 结构表示的注解一致。

- num_element_value_pairs

  num_element_value_pairs 项的值给出了当前 annotation 结构表示的注解的键值对（键值对的格式为：元素-值）的数量，即 element_value_pairs[] 数组成员数量。需要注意的是，在单独一个注解中可能含有数量最多为 65535 个键值对。

- element_value_pairs[]

  element_value_pairs[]数组的每一个成员的值对应当前 annotation 结构表示的注解中的一个唯一的键值对。element_value_pairs 的成员包含如下两个项。

  - element_name_index

    element_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的字段描述符（§4.3.2），这个字段描述符用于定义当前 element_value_pairs 的成员表示的注解的注解名。

  - value

    value 的项的值给出了 element_value_pairs 中的成员的键值对中的值。

4.7.16.1 element_value 结构

element_value 结构是一个可辨识联合体（Discriminated Union）[①]，用于表示“元素-值”的键值对中的值。它被用来描述所有注解（包括 RuntimeVisibleAnnotations、RuntimeInvisibleAnnotations、RuntimeVisibleParameterAnnotations 和 untimeInvisibleParameterAnnotations）中涉及到的元素值。

element_value 的结构格式如下：

```
element_value {
    u1 tag;
```

[①] 译者注：“Discriminated Union”是一种数据结构，用于表示若干种具有独立特征的同类项集合。

```
        union {
                u2 const_value_index;
                {       u2 type_name_index;
                        u2 const_name_index;
                } enum_const_value;
                u2 class_info_index;
                annotation annotation_value;
                {       u2 num_values;
                        element_value values[num_values];
                } array_value;
        } value;
}
```

element_value 结构各项的说明如下。

- tag

  tag 项表明了当前注解的元素-值对的类型。tag 值为字母'B'、'C'、'D'、'F'、'I'、'J'、'S'和'Z'时表示的含义和章节 4.3.2 中表 4.2 所定义的一样。其余 tag 的预定义值和对应解释在表 4.9 列出。

  表 4.9 附加的 tag 值解释

tag 值	元素类型
s	String
e	enum constant
c	class
@	annotation type
[	array

- value

  value 项的表示当前注解元素的值。此项是一个联合体结构，当前项的 tag 项决定了这个联合体结构的哪一项会被使用：

  - type_name_index

    type_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的字段描述符（§4.3.2），这个字段描述符表示当前 element_value 结构所表示的枚举常量类型的内部形式的二进制名称（§ 4.2.1）。

- const_name_index

  const_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示一个有效的字段描述符（§4.3.2），这个字段描述符表示当前 element_value 结构所表示的枚举常量类型的简单名称。

- class_info_index

  当 tag 项为'c'时，class_info_index 项才会被使用。class_info_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示返回描述符（§4.3.3）的类型，这个类型由当前 element_value 结构所表示的类型决定（譬如：'V'表示 Void，'Ljava/lang/Object;'表示类 java.lang.Object 等）。

- annotation_value

  当 tag 项为'@'时，annotation_value 项才会被使用。这时 element_value 结构表示一个内部的注解（Nested Annotation）。

- array_value

  当 tag 项为'['时，array_value 项才会被使用。array_value 项包含如下两项：

  - num_values

    num_values 项的值给定了当前 element_value 结构表示的数组类型的成员的数量。注意，允许数组类型元素中最多有 65535 个元素。

  - values

    values 的每个成员的值都给指明了当前 element_value 结构所表示的数组类型的一个元素值。

4.7.17 RuntimeInvisibleAnnotations 属性

RuntimeInvisibleAnnotations 属性和 RuntimeVisibleAnnotations 属性相似，不同的是 RuntimeVisibleAnnotations 表示的注解不能被反射 API 访问，除非 Java 虚拟机通过特殊的实现相关的方式（譬如特定的命令行参数）收到才会（为反射的 API）使用这些注解。否则，Java 虚拟机将忽略 RuntimeVisibleAnnotations 属性。

RuntimeInvisibleAnnotations 属性是一个变长属性，位于 ClassFile（§4.1），field_info（§4.5）或 method_info（§4.6）结构的属性表中。用于保存 Java 语言中的类、字段或方法的运行时的非可见注解。每个 ClassFile，field_info 和 method_info 结构最多只能含有一个 RuntimeInvisibleAnnotations 属性，它为当前的程序元素保存所有的运行时非可见的 Java 语言注解。

RuntimeInvisibleAnnotations 属性格式如下：

```
RuntimeInvisibleAnnotations_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 num_annotations;
     annotation annotations[num_annotations];
}
```

RuntimeInvisibleAnnotations_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“RuntimeInvisibleAnnotations”。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。

  attribute_length 项的值由当前结构的运行时非可见注解的数量和值决定。

- num_annotations

  num_annotations 项的值给出了当前结构表示的运行时可见注解的数量。每个程序元素最多可能会被附加 65535 个运行时非可见的 java 语言注解。

- annotations[]

  annotations[]数组的每个成员的值表示一个程序元素的唯一的运行时可见注解。

4.7.18 RuntimeVisibleParameterAnnotations 属性

RuntimeVisibleParameterAnnotations 属性是一个变长属性，位于 method_info（§4.6）结构的属性表中。用于保存对应方法的参数的所有运行时可见 Java 语言注解。每个

method_info 结构最多只能包含一个 RuntimeVisibleParameterAnnotations 属性，用于保存当前方法的参数的所有可见的 Java 语言注解。Java 虚拟机必须保证这些注解可被反射的 API 使用。

RuntimeVisibleParameterAnnotations 格式如下：

```
RuntimeVisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {       u2 num_annotations;
            annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

RuntimeVisibleParameterAnnotations_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“RuntimeVisibleParameterAnnotations”。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。attribute_length 项的值由对应方法的参数数量，参数的运行时可见注解和它们的值所决定。

- num_parameters

  num_parameters 项的值给出了注解中出现的 method_info（§4.6）结构表示的方法参数的数量（这些信息可以从方法描述符中获得）。

- parameter_annotations

  parameter_annotations[]数组中每个成员的值表示一个的参数的所有的运行时可见注解。它们的顺序和方法描述符表示的参数的顺序一致。每个 parameter_annotations 成员都包含如下两项：

  - num_annotations

    num_annotations 项的值表示 parameter_annotations[]数组在当前所索引处的元素的可见注解的数量。

- annotations[]

  annotations[]数组中的每个成员的值表示 parameter_annotations[]数组在当前索引处的元素的一个唯一的可见注解。

4.7.19 RuntimeInvisibleParameterAnnotations 属性

RuntimeInvisibleParameterAnnotations 属性和 RuntimeVisibleParameterAnnotations 属性类似，区别是 RuntimeInvisibleParameterAnnotations 属性表示的注解不能被反射的 API 访问，除非 Java 虚拟机通过特殊的实现相关的方式（譬如特定的命令行参数）收到才会（为反射的 API）使用这些注解。否则，Java 虚拟机将忽略 RuntimeInvisibleParameterAnnotations 属性。

RuntimeInvisibleParameterAnnotations 属性是一个变长属性，位于 method_info（§4.6）结构的属性表中。用于保存对应方法的参数的所有运行时非可见的 Java 语言注解。每个 method_info 结构最多只能含有一个 RuntimeInvisibleParameterAnnotations 属性用于保存当前 Java 语言中的程序元素的所有运行时非可见注解。

RuntimeInvisibleParameterAnnotations 属性的格式如下：

RuntimeInvisibleParameterAnnotations 格式如下：

```
RuntimeInvisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {       u2 num_annotations;
            annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

RuntimeInvisibleParameterAnnotations_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“RuntimeInvisibleParameterAnnotations”。

- attribute_length

attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。

attribute_length 项的值由对应方法的参数数量，参数的运行时非可见注解和它们的值所决定。

- num_parameters

  num_parameters 项的值给出了注解中出现的 method_info（§4.6）结构表示的方法参数的数量（这些信息可以从方法描述符中获得）。

- parameter_annotations

  parameter_annotations[]数组中每个成员的值表示一个的参数的所有的运行时非可见注解。它们的顺序和方法描述符表示的参数的顺序一致。每个 parameter_annotations 成员都包含如下两项：

  - num_annotations

    num_annotations 项的值表示 parameter_annotations[]在当前所索引处的元素的非可见注解的数量。

  - annotations[]

    annotations[]数组中的每个成员的值表示 parameter_annotations[]数组在当前索引处的元素的一个唯一的非可见注解。

4.7.20 AnnotationDefault 属性

AnnotationDefault 属性是一个变长属性，位于某些特殊的 method_info（§4.6）结构的属性表中，这些结构表示注解类型的元素。AnnotationDefault 属性用于保存 method_info 结构表示的元素的默认值。每个 method_info 结构表示的一个元素的注解最多只能有一个 AnnotationDefault 属性。Java 虚拟机必须保证这些默认值可见，可供反射 API 使用。

AnnotationDefault 属性格式如下：

```
AnnotationDefault_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     element_value default_value;
}
```

AnnotationDefault_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“AnnotationDefault”。

- attribute_length

  attribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。attribute_length 项的值由默认值决定。

- default_value

  default_value 项表示 AnnotationDefault 属性所对应注解类型元素的默认值。

4.7.21 BootstrapMethods 属性

BootstrapMethods 属性是一个变长属性，位于 ClassFile（§4.1）结构的属性表中。用于保存 invokedynamic 指令引用的引导方法限定符。

如果某个 ClassFile 结构的常量池中有至少一个 CONSTANT_InvokeDynamic_info（§4.4.10）项，那么这个 ClassFile 结构的属性表中必须有一个明确的 BootstrapMethods 属性。ClassFile 结构的属性表中最多只能有一个 BootstrapMethods 属性。

BootstrapMethods 属性格式如下：

```
BootstrapMethods_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_bootstrap_methods;
    {       u2 bootstrap_method_ref;
            u2 num_bootstrap_arguments;
            u2 bootstrap_arguments[num_bootstrap_arguments];
    } bootstrap_methods[num_bootstrap_methods];
}
```

BootstrapMethods_attribute 结构各项的说明如下：

- attribute_name_index

  attribute_name_index 项的值必须是对常量池的一个有效索引。常量池在该索引处

的成员必须是 CONSTANT_Utf8_info（§4.4.7）结构，表示字符串“BootstrapMethods”。

- attribute_length

  ttribute_length 项的值给出了当前属性的长度，不包括开始的 6 个字节。attribute_length 项的值由默认值决定[①]。

- num_bootstrap_methods

  num_bootstrap_methods 项的值给出了 bootstrap_methods[]数组中的引导方法限定符的数量。

- bootstrap_methods[]

  bootstrap_methods[]数组的每个成员包含一个指向 CONSTANT_MethodHandle 结构的索引值，它代表了一个引导方法。还包含了这个引导方法静态参数的序列（可能为空）。bootstrap_methods 每项必须包含以下 3 项：

  - bootstrap_method_ref

    bootstrap_method_ref 项的值必须是一个对常量池的有效索引。常量池在该索引处的值必须是一个 CONSTANT_MethodHandle_info 结构。

    注意：此 CONSTANT_MethodHandle_info 结构的 reference_kind 项应为值 6（REF_invokeStatic）或 8（REF_newInvokeSpecial）（§5.4.3.5），否则在 invokedynamic 指令解析调用点限定符时，引导方法会执行失败。

  - num_bootstrap_arguments

    num_bootstrap_arguments 项的值给出了 bootstrap_arguments[]数组成员的数量。

  - bootstrap_arguments[]

    bootstrap_arguments[]数组的每个成员必须是一个对常量池的有效索引。常量池在该索引出必须是下列结构之一：

    CONSTANT_String_info,CONSTANT_Class_info、

    CONSTANT_Integer_info,CONSTANT_Long_info、

[①] 译者注：这里原文为“attribute_length item is thus dependent on the default value”，但从上下文来说词不达意，笔者怀疑是原作者偷懒，误将前一节“AnnotationDefault 属性”中的 attribute_length 项的描述内容复制到这里。

CONSTANT_Float_info,CONSTANT_Double_info、

CONSTANT_MethodHandle_info 或 CONSTANT_MethodType_info。

4.8 格式检查

如果 Java 虚拟机准备加载（§5.3）某个预期为 Class 文件格式的文件，它首先应保证这个文件符合 Class 文件格式（§4.1）的基本格式标准，这个过程就被称为格式检查。格式检查需要确认 Class 文件的开头 4 个字节必须为正确的魔数，所有预定义的属性必须符合它们应有的长度，文件的尾部不能缺少或者多出额外的字节，常量池必须不含任何被规范未预定义的信息。

这个基本的 Class 文件完整性检查对于 Class 文件的内容的后续解析工作是非常有必要的。格式检查本应独立于字节码验证，它们两者都是都是 Class 文件验证过程的一部分。但由于历史原因，格式检查与字节码验证曾经混淆在一起，因为它们两者都是完整性检查的一种形式。

4.9 Java 虚拟机代码约束

Java 虚拟机将普通方法、实例初始化方法（§2.9）或类和接口初始化方法（§2.9）的代码存储在 Class 文件 method_info 结构里的 Code 属性的 code[]数组中。这节将会描述与 Code_attribute 结构的内容相关的约束情况。

4.9.1 静态约束

Class 文件的静态约束（Static Constraints）是一系列对文件是否编排良好的定义。在之前章节描述到 Class 文件中的 Java 虚拟机代码静态约束而导致的异常时，就已经提及过这些约束。Java 虚拟机代码对 Class 文件中的静态约束确定了 Java 虚拟机指令在 code[]数组中是如何排列的，某些特殊的指令必须带有哪些操作数等等。

code[]数组中的指令的约束情况如下：

1. code[]数组不能为空，因此 code_length 项的值也一定大于 0。
2. code_length 项的值必须小于 65536。

3. code[]数组中第一条指令的操作码是从数组中索引为 0 处开始。
4. code[]数组中仅仅只能出现 6.4 章节中所列的指令。那些被保留使用的或没有列在本规范的操作码的指令一定不允许出现在 code[]数组中。
5. 如果 Class 文件的版本号大于或等于 51.0，jsr 和 jsr_w 这两个操作码也不能出现在 code[]数组中。
6. 对于 code[]数组中除最后一条指令外的其它指令来说，后一条指令的操作码的索引等于当前指令操作码的索引加上当前指令的长度（包含指令带有的操作数）。wide 指令在这种情况下将与其它的指令使用相同的规则进行处理：跟随在 wide 指令之后的操作码将会被视为 wide 指令的操作数。程序跳转时不能被直接跳转到该操作码。
7. code[]数组中最后一条指令的最后一个字节的索引必须等于 code_length 值减 1。

code[]数组中的指令的操作数的约束情况如下：

- 所有跳转和分支指令（jsr、jsr_w、goto、goto_w、ifeq、ifne、ifle、iflt、ifgt、ifnull、ifnonnull、if_icmpeq、if_icmpne、if_icmple、if_icmplt、if_icmpge、if_icmpgt、if_acmpeq、if_acmpne）的跳转目标必须是本方法内某条指令的操作码。但跳转与分支指令的目标一定不能是某条被 wide 指令修饰的指令的操作码；跳转与分支指令的跳转目标可以是 wide 指令本身。
- 每个 tableswitch 指令的跳转目标（包含 default 目标）都必须是当前方法内的某条指令的操作码。每个 tableswitch 指令必须在它的跳转表中包含与自己 low 和 high 跳转表操作数的值相等的条目，并且 low 值必须小于等于 high 值。tableswitch 指令的跳转目标也不能是某条被 wide 指令修饰的指令的操作码；tableswitch 的跳转目标可能是 wide 指令本身。
- 每个 lookupswitch 指令的跳转目标（包含 default 目标）都必须是当前方法内的某条指令的操作码。每个 lookupswitch 指令都必须包含与自己 npairs 操作数的值相等数量的 match-offset 值对。match-offset 值对必须是以 match 值递增的顺序排列的。lookupswitch 指令的跳转目标也不能是某条被 wide 指令修饰的指令的操作码；lookupswitch 指令的跳转目标可能是 wide 指令本身。
- 每个 ldc 和 ldc_w 指令的操作数都必须是 constant_pool 表中的一个有效索引。操作数必须表示到 constant_pool 表的有效索引。被此索引所引用的常量池项必须符合下列规则：

- 如果 Class 文件版本号小于 49.0，那就可能是 CONSTANT_Integer，CONSTANT_Float 或 CONSTANT_String。
- 如果 Class 文件版本号是 49.0 或 50.0，那就可能是 CONSTANT_Integer，CONSTANT_Float，CONSTANT_String 或 CONSTANT_Class。
- 如果 Class 文件版本号是 51.0，那就可能是 CONSTANT_Integer，CONSTANT_Float，CONSTANT_String，CONSTANT_Class，CONSTANT_MethodType 或 CONSTANT_MethodHandle。

- 每个 ldc2_w 指令的操作数都必须是 constant_pool 表内的一个有效索引。被此索引所引用的常量池项必须是 CONSTANT_Long 或 CONSTANT_Double 之一。并且在紧随该常量池索引之后那个常量池项一定不能被独立使用。
- 每个 getfield、putfield、getstatic 和 putstatic 指令的操作数都必须是 constant_pool 表内的一个有效索引。由这些索引所引用的常量池项必须是 CONSTANT_Fieldref 类型。
- 每个 invokevirtual、invokespecial 和 invokestatic 指令的索引位操作数（Indexbyte Operand）都必须是 constant_pool 表内的有效索引。由这些索引所引用的常量池项必须是 CONSTANT_Methodref 类型。
- 每个 invokedynamic 指令的索引位操作数必须表示对 constant_pool 表内的有效索引。由此索引所引用的常量池项必须是 CONSTANT_InvokeDynamic 类型。所有 invokedynamic 指令的第三和第四个操作数字节必须是 0。
- 只有 invokespecial 指令可以调用实例初始化方法（§2.9）。其他所有以'<'（'\u003c'）字符开头的方法都不能再被方法调用指令所调用。需要特别说明的是，被命名为<clinit>的类和接口的初始化方法也不会被 Java 虚拟机指令所调用，而是由 Java 虚拟机自己来隐式调用。
- 每个 invokeinterface 指令的索引位操作数都必须表示对 constant_pool 表的有效索引。被此索引所引用的常量池项必须是 CONSTANT_InterfaceMethodref 类型。每个 invokeinterface 指令的 count 操作数的值必须真实反映出可传入到接口方法的参数所占用的局部变量个数，它由 CONSTANT_InteraceMethodref 项所引用的 CONSTANT_NameAndType_info 结构的描述符所暗示。所有 invokeinterface 指令的第四个操作数字节必须是 0。

- instanceof，checkcast，new 和 anewarray 指令的操作数，和 multianewarray 指令的索引位操作数都必须表示对 constant_pool 表的有效索引。由这些索引所引用的常量池项必须是 CONSTANT_Class 类型。
- anewarray 指令不能用于创建维度超过 255 维的数组。
- new 指令引用的 contant_pool 表中 CONSTANT_Class 项来表示创建的对象类型，这个类型不能是数组类型。也就是说 new 指令不能用于创建数组。
- multianewarray 指令只能创建一个最多不超过它 dimensions 操作数值所代表的维度的数组。那就是说，multianewarray 指令创建的数组的维度可以比操作数所示的小（注：多维数组中某一维度的数组长度为 0 的情况），但是它所创建的数组维度绝对不能比操作数中所示的维度值要多。multianewarray 指令的 dimensions 操作数必须不能为 0。
- 每个 newarray 指令的 atype 操作数必须为下列类型的值：T_BOOLEAN（4），T_CHAR（5），T_FLOAT（6），T_DOUBLE（7），T_BYTE（8），T_SHORT（9），T_INT（10）和 T_LONG（11）。
- 所有 iload，fload，aload，istore，fstore，astore，iinc 和 ret 指令的索引位操作数必须是非负整数且不能大于值 max_locals-1。
- 所有 iload_<n>，fload_<n>，aload_<n>，istore_<n>，fstore_<n>和 astore_<n>指令的显式索引必须小于等于值 max_locals-1。
- 所有 lload，dload，lstore 和 dstore 指令的索引位操作数必须小于等于值 max_locals-2。
- 所有 lload_<n>，dload_<n>，lstore_<n>和 dstore_<n>指令的显式索引必须小于或等于值 max_locals-2。
- 修饰 iload，fload，aload，istore，fstore，astore，ret 或 innc 指令的 wide 指令的索引位操作数必须表示非负整数且小于等于值 max_locals-1。修改 lload，dload，lstore 和 dstore 指令的 wide 指令的索引位操作数必须表示非负整数且小于或等于值 max_locals-2。

4.9.2 结构化约束

在 code[]数组上的结构化约束目的是具体限定 Java 虚拟机指令之间的关系。如下所示：

- 所有指令都必须可以在操作数栈和局部变量表中，基于合适的类型及参数下正常执行，且不用关心调用它的执行路径。指令如果可以操作 int 类型的值，那它同样也有可以去操作 boolean，byte，char 及 short 类型的值。（在§2.3.4 和§2.11.1 节曾经提到，Java 虚拟机内部会将 boolean，byte，char 及 short 转成 int 类型。）
- 如果某条指令可以通过几个不同的执行路径执行，在执行指令之前，不管最终采用哪条执行路径，操作数栈必须有相同的深度（§2.3.4）。
- 在执行过程中，表示 long 或 double 类型的值的局部变量表值对的存储顺序绝不可以倒置或割裂。对于这样的局部变量值对也绝不可以分开单独使用。
- 在正式地赋值之前，局部变量（或局部变量值对，表示 long 或 double 类型）不可以被访问。
- 在执行过程中，操作数栈不允许增长到超过 max_stack 项的值的深度（§2.6.2）。
- 在执行过程中，不允许从操作数栈中取出比它包含的全部数据还多的数据。
- 每个 invokespecial 指令应该指明一个实例初始化方法（§2.9），这个实例初始化方法或是当前类的方法，或是当前类某个父类的方法。

  如果 invokespecial 所指明的实例初始化方法的类不是当前类或父类的方法，并且操作数栈上的目标引用是一个先前使用 new 指令创建的实例，那么 invokespecial 必须从这个实例所属的类中确定一个实例初始化方法。
- 在调用实例初始化方法（§2.9）时，那个还未被初始化的实例必须存放在操作数栈上适当的位置。如果一个实例已经被初始化过，那就不允许再调用它的实例初始化方法。
- 在调用任意实例的方法或访问任意实例变量之前，那个包含此实例方法或实例变量的实例对象必须是已经被初始化过的。
- 在操作数栈或局部变量表上不允许有未被初始化的类变量作为回向分支（Backwards Branch）[①]的目标，除非在分支指令上有特殊的未初始化实例类型已经在分支的目标上与自身归并（§4.10.2.4）。
- 被异常处理器（§4.10.2.4）所保护的代码中，局部变量表内不允许出现未被初始化的

[①] 译者注：如果一个分支的结果指向指令流中一个在此指令之前的目标，则称为回向分支。

实例。

- 当 jsr 或 jsr_w 指令执行时，在操作数栈或局部变量表中不允许出现未被初始化的实例。
- 除了 Object 类以外，所有实例初始化方法，在访问实例成员之前都要么调用 this() 来引用该类中的其它初始化方法，要么通过 super() 来调用它的直接父类的实例初始化方法来初始化实例。然而，在调用任何实例初始化方法之前，声明在当前类中的实例字段应当被赋予默认值。
- 所有方法调用的参数都必须与方法描述符（§4.3.3）相兼容（JLS §5.3）。
- 方法调用指令的目标实例的类型必须与指令所指定的类或接口的类型相兼容（JLS §5.2）。除非某个实例初始化方法正在被调用，否则每个 invokespecial 指令的目标类型必须与当前类相兼容。
- 所有返回指令都必须与方法的返回类型匹配。如果方法的返回类型是 boolean，byte，char，short 或 int 就只能使用 ireturn 指令。如果方法返回 float，long 或 doule 类型，就分别使用 freturn，lreturn 或 dreturn 指令。如果方法返回 reference 类型，就必须使用 areturn 指令，并且返回值的类型必须与方法的返回描述符（§4.3.3）的类型相兼容（JLS §5.2）。所有的实例初始化方法，类或接口初始化方法和声明返回 void 的方法都必须使用 return 指令返回。
- 如果通过 getfield 或 putfield 访问与当前类不在同一运行时包的父类中的 protected 字段，那么正在被访问的实例必须与当前类或当前类的子类是相同的。如果通过 invokevirtual 或 invokespecial 访问与当前类不在同一运行时包的父类中的 protected 方法，那么正在被访问的实例必须与当前类或当前类的子类是相同的。
- 通过 getfield 访问指令或 putfield 修改指令来操作的实例类型必须与指令所指定的类型相兼容（JLS §5.2）。
- 通过 putfield 或 putstatic 来保存的值的类型必须与正在操作的实例或类的字段描述符相兼容（§4.3.2）。如果描述符类型是 boolean，byte，char，short 或 int，那么值就一定是 int 类型。如果描述符类型是 float，long 或 double，那么值就分别是 float，long 或 double。如果描述符是 reference 类型，那么值的类型就必须与描述符类型相兼容（JLS §5.2）。
- 由 aastore 指令存储到数组中每个值都必须是引用类型。正在被 aastore 指令操作的数组的组件类型（Component Type，§2.4）也必须是引用类型。

- 每个 athrow 指令只能抛出 Throwable 类或它的子类的实例。所有出现在方法异常表中 catch_item 项的类都必须是 Throwable 或它的子类。
- 程序执行时，不允许执行超过 code[]数组末端。
- 返回地址（returnAddress 类型值）不可以从局部变量表中加载。
- 在多个 jsr 或 jsr_w 之后的指令可能会由单个 ret 指令来返回。
- 当某程序子片段（Subroutine）已经出现在其他程序子片段调用链上时，跳转入该程序子片段的 jsr 或 jsr_w 指令就不能作为另一个程序子片段的返回地址，因为这样会产生无限递归的情况。（当使用 try-finally 结构时，程序子片段可以在 finally 语句块中嵌套，但这并不会无限递归）。
- 所有 returnAddress 类型的实例至多会返回一次。如果 ret 指令在某码子片段调用链中返回且超过规定实例的 returnAddress 类型的相应地址，之后那个实例就不能再将它作为返回地址。

4.10 Class 文件校验

在前面章节中提到了许多规则和约束，Java 语言编译器需要遵循这些规则来生成代码，以保证所生成的 Class 文件符合静态和结构化约束。但是 Java 虚拟机无法保证所有它将要加载的 Class 文件都来自于正确实现的编译器或者有正确的格式。某些应用程序，譬如网络浏览器可能会下载程序的源码，然后将它们编译为 Class 文件，也有某些应用程序可能会直接下载已被编译过的 Class 文件。这些应用程序需要确定 Class 文件是否是来自于一个错误实现的编译器，甚至是否是来自于想恶意破坏虚拟机的人。

如果仅做编译时检查的话还存在另外一个问题：版本偏差（Version Skew）。假设有一个用户成功编译了一个名为 PurchaseStockOptions 的类，它是 TradingClass 的子类。但是 TradingClass 的内容很可能在上次编译之后又发生了变化，从而导致无法与以前存在的二进制内容向兼容。如原来存在的方法可能被删除、返回值类型可能被修改、字段的类型被改变、字段从实例变量修改为类变量、方法的访问修饰符可能从 public 修改为 private 等等。对于两个类型是否能兼容，在《Java 语言规范》第 13 章“二进制兼容性”之中作了详细讨论。

鉴于上述潜在问题，Java 虚拟机需要自己能够独立校验它准备载入的 Class 文件是否能满足规定的约束条件。Java 虚拟机实现会在文件的链接阶段（§5.4）对 Class 文件进行必要的校

验。

链接期校验还有助于增强解释器的执行性能，因为解释器在运行期无需再对每个执行指令进行检查。Java 虚拟机在运行期可以假设所有必要的校验都已经执行过。例如，Java 虚拟机可以确保以下内容：

- 操作数栈不会发生上限或下线溢出。
- 所有局部变量的使用和存储都是有效的。
- 所有 Java 虚拟机指令都拥有正确的参数类型。

校验器也会对那些不是用 Code 属性（§4.7.3）中 code[]数组的内容进行校验，这些检验包括如下内容：

- 确保 final 类不会有子类，以及 final 方法不会被覆盖（§5.4.5）。
- 确保除了 Object 之外的每一个类都有直接父类。
- 确保常量池满足文档静态约束：例如常量池中所有 CONSTANT_Class_info 结构所包含的 name_index 项都是一个指向 CONSTANT_Utf8_info 项的有效常量池索引。
- 确保常量池之中所有字段引用和方法引用都有有效的名称、类型和方法描述符。

请注意，上述检查不能保证给定的字段或者方法在给定的类中实际存在，也不能保证类型描述符中引用的是一定是一个真实存在的类，而只能保证这些项在形式上是正确的。更多相关的检查会在字节码本身已被验证过之后的解析阶段进行。

Java 虚拟机可以使用两种不同的检查策略：类型检查验证和类型推导验证。对于版本号大于或等于 50.0 的 Class 文件，规定必须使用类型检查验证。

由于验证版本号小于 50.0 的 Class 文件的需要，类型推导验证必须被除使用于 Java ME CLDC 和 Java Card 平台的虚拟机外的所有虚拟机实现所支持。使用于 Java ME CLDC 和 Java Card 平台的虚拟机要遵循它们自身的规范。

4.10.1 类型检查验证

译者注：在原文的 4.10.1 中，列举了大量如何使用 StackMapTable 属性进行类型检查的规则和方法原型。由于大量方法原型的存在，这部分内容翻译出来意义不大，所以本小节是全规范之中唯一未进行翻译的部分，请有需要的读者阅读规范原文。

4.10.2 类型推导验证

对于不包含 StackMapTable 属性的 Class 文件（这样的 Class 文件其版本号必须小于或等于 49.0）需要使用类型推断的方式来验证。

4.10.2.1 类型推断的验证过程

在链接过程中，验证器通过数据流分析的方式检查每个方法 Code 属性中的 code[]数组。验证器必须确保程序中无论在任何执行时间、无论是选择哪条执行路径，都必须遵循以下规则：

- 操作数栈的深度及所包含的值的类型总是相同。
- 在明确确定要局部变量的数据类型之前，不能访问该局部变量。
- 方法调用必须携带适当的参数。
- 对字段所赋的值一定会与对象的类型相符。
- 所有的操作码在操作数栈和局部变量表中都有适当类型的参数。
- 对于由异常处理器所保护的代码，局部变量中不允许出现未初始化的类实例。然而，未初始化的类实例可以出现在由异常处理器保护的代码的操作数栈中，因为当异常被抛出时，操作数栈的内容就会被丢弃。

更多数据流分析的细节，请参考 4.10.2.2 节“字节码验证器”。

有时候考虑到效率，验证器中一些关键测试会被延迟至方法的代码第一次被真正调用时才执行。正因如此，验证器除非确实有必要的情况，否则都会尽量避免触发其他 Class 文件的加载。

例如，某个方法调用另一个方法，且被调用的方法返回了一个类 A 的实例，这个实例会赋值给与它相同类型的一个字段，这时验证器不会耗费时间去检查 A 类是否真实存在。然而，如果这个实例被赋值给 B 类型的字段，那么验证器必须确保 A 和 B 类都已经加载过且 A 是 B 的某个子类。

4.10.2.2 字节码验证器

本节将介绍更多如何使用类型推导方式来对 Java 虚拟机代码进行校验的细节。

Class 文件中每个方法的代码都要被单独地验证。首先，组成代码的字节序列被会分隔成一系列指令，每条指令在 code[]数组中的起始位置索引将被记录在另外的数组中。然后，验证器再

次遍历代码并分析每条指令。经过这次遍历之后，会生成一个数组结构用于存储方法中每个Java虚拟机指令的相关信息。如果有必要的话，验证器会检查每条指令的操作数，以确保它们是合法的。例如将会检查以下内容：

- 方法分支跳转一定不能超过code[]数组的范围。
- 所有控制流指令的目标都应该是某条指令的起始处。以wide指令为例，wide操作码可以看作是指令的起始处，但被wide指令所修饰的操作码不能再被看作是指令的起始处。方法中如果某个分支指向了一条指令中间，那这种行为是非法的。
- 方法会显式声明自己局部变量表的大小，指令所访问或修改的局部变量索引绝不能大于这个限制值。
- 对常量池项的引用必须要求该项符合预期的类型。譬如，getfield指令只能引用字段项。
- 代码执行不能终止于指令的中部。
- 代码执行不能超出code[]数组的尾部。
- 对于所有的异常处理器，由处理器所保护的代码的起点应该是指令的起始处，终点应在代码的尾部。起点必须在终点之前。异常处理器保护的代码必须起始于一个有效的指令，且不能是被wide指令修饰的操作码。

对于方法中的每条指令来说，在指令执行之前，验证器会记录下此时操作数栈和局部变量表中的内容。对于操作数栈，验证器需要知道栈的深度及里面每个值的类型。对于每个局部变量，它需要知道当前局部变量的值的类型，如果当前值还没有被初始化，那么它需要知道这是一个未使用或未知的值。在确定操作数栈上值的类型时，字节码验证器不需要区分到底是哪种整型（例如byte，short和char）。

接下来真正的数据流分析器（Data-Flow Analyzer）被初始化了。在方法的第一条指令执行之前，局部变量表中存放方法参数的局部变量就已根据方法描述符中描述的参数数据类型初始化好，此时操作数栈为空，其它的局部变量包含非法（不可使用）的值。对于那些还没有被检查的指令，分析器不保存与它们有关的操作数栈或局部变量表信息。

接着，数据流分析器可以开始运作了。它为每条指令都设置一个“变更位”（Changed Bit），用来表示指令是否需要被检测。最开始时只有方法的第一条指令设置了变更位。数据流分析器执行流程如下面的循环：

1．选取一个变更位被设置过的指令。如果不能选取到变更位被设置过的指令，那么表示方法

被成功地验证过。否则，关闭选取的指令的变更位。

2．通过下述的方式来模拟操作数栈和局部变量表中指令的行为：

- 如果指令使用操作数栈中的值，就得确保操作数栈中有足量的数据且栈顶值的类型是合适的。否则验证失败。
- 如果指令使用局部变量中的值，就得确保那个特定变量的值符合预期的类型。否则验证失败。
- 如果指令需要往操作数栈存储数据，就得确保操作数栈中有充足的空间来容纳新值，并在模拟的操作数栈的栈顶增加新值的类型。
- 如果指令试图修改局部变量中的值，记录下当前局部变量包含的新值的类型。

3．检查当前指令的后续指令。后续指令可以是下述的某一种：

- 如果当前指令不是非条件的控制转移指令（如 goto，return 或 athrow），那么后续指令就是下一条指令。如果此时超出方法的最后一条指令，那么验证失败。
- 条件或非条件的分支或转换指令的目标指令。
- 当前指令的任何异常处理器。

4．在继续执行下一条指令之前，需要将当前指令执行结束后操作数栈和局部变量表的状态合并起来。在处理控制转移到异常处理器的情况时，操作数栈上只包含异常处理器的那个异常类型对象。为此操作数栈上必须有充足的空间来容纳这个值，就如同有指令将值压到栈中一样。

- 如果这是后续指令是第一次被访问到，在指令执行之前，将采用第 2 和第 3 步中描述的规则，把操作数栈和局部变量表的操作记录为分析器模拟的操作数栈和局部变量表的初始值。为后续指令设置变更位。
- 如果后续指令之前执行过，只要将操作数栈和局部变量表中按照第 2 和 3 步的规则计算出值合并入模拟的操作数栈和局部变量表中即可。如果之前对这些值有更改过，那么也得设置变更位。

5．继续第 1 步。

合并两个操作数栈之前，每个栈上的值的数量必须是相同的。栈上每个元素的类型也必须是相同的，除非两个栈上相应位置有不同类型的 reference 值。在这种情况下，合并后的操作数栈也包含一个 reference 指向两种类型的“最小公共父类”[①]。这种最小公共父类应该总是有的，因

[①] 译者注：最小公共父类是指类继承体系中，两个类型父类中最底层的那个父类

为 Object 类型是所有类或接口的父类。如果两个操作数栈不能合并，那么验证失败。

合并两个局部变量表状态之前，局部变量对的数量应该是相同的。如果两个类型不相同，除非它们两个都包含着 reference 值，不然验证器就记录下当前局部变量包含着一个未使用的值。如果两个局部变量对都包含 reference 值，最终合并后的状态是 reference 且指向两种类型的最小公共父类的实例。

如果数据流分析器在检测某个方法时没有发现错误，那就表示此方法被 Class 文件验证器成功地验证了。

某些指令和数据类型会使数据流分析器的行为变得更为复杂，接下来我们详细介绍这方面的内容。

4.10.2.3 long 和 double 类型的值

long 和 double 类型的数值在验证过程中要被特殊处理。

当一个 long 或 double 型的数值被存放到局部变量表的索引 n 处时，索引 n+1 也会被显式声明由索引 n 持有，且不能再被用作其它局部变量的索引。先前索引 n+1 处的值也会变为不可用。

当一个值想要存放到局部变量表的索引 n 处时，就必须检查索引 n-1 处是否为 long 和 double 型数值的索引。如果是的话，在索引 n-1 处的局部变量就应该显式声明：它包含一个未使用的值。如果在索引 n 处的局部变量已经被 long 和 double 型覆盖，在索引 n-1 处的局部变量就不能再表示一个 long 和 double 型的数值了。

在操作数栈上处理 long 和 double 型的数值很简单：验证器把它们当作栈上的单个数值。例如，验证 dadd 操作码（对两个 double 值加和）的代码时就需要检查栈顶的两个元素是否为 double 类型。在计算操作数栈的深度时，long 和 double 型的数值都占有两个位置。

类型无关的指令[①]在使用操作数栈时必须将 long 和 double 型数值整体对待。例如，如果使用 pop 或 dup 指令且栈顶的元素是 double 类型时，验证器就会提示错误。此时必须使用 pop2 或 dup2 指令来处理。

[①] 译者注：Java 虚拟机指令集中类型无关的指令一般都是操作数栈指令，譬如 pop、dup 等。

4.10.2.4 实例初始化方法与创建对象

创建一个新的类实例需要多个步骤的处理过程。例如下面语句：

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

它的实现可能像这样：

```
... ...
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 #5 // Invoke myClass.<init>
... ...
```

上述指令序列在操作数栈栈顶上保留着最新创建和初始化过的对象引用(代码编译成 Java 虚拟机指令集的其它例子请参考第 3 章 “为 Java 虚拟机编译”)。

类 myClass 的实例初始化方法（§2.9）可以看到新建但未初始化的对象，并且这个对象以 this 参数的方式存放在局部变量索引 0 处。在初始化方法通过 this 调用 myClass 或其父类的其它初始化方法之前，这个初始化方法唯一能做的事情就是在 this 对象上为 myClass 类声明的字段赋值。

在为实例方法做数据流分析时，验证器初始化局部变量索引 0 处为当前类的一个对象。在分析实例初始化方法时，局部变量 0 处包含一个特殊类型用来显示此未初始化对象。在调用这个对象的恰当的实例初始化方法（由当前的类或它的父类）后，由当前的类类型在验证器的操作数栈和局部变量表模型中都放置此特殊类型。验证器会拒绝那些对象初始化之前使用对象及初始化多个对象的代码。除此之外，它还要确保在调用正常的方法之前都得事先调用方法所在类或它的直接父类的实例初始化方法。

与此类似的是，一个特殊类型会被创建并入栈到验证器的操作数栈模型中用来表示 Java 虚拟机 new 指令的结果。这个特殊的类型将被用来区分一个已被完整创建的实例和一个被 new 指令创建，但是还未执行过初始化方法的实例。当一个未初始化类实例的实例初始化方法被调用之后，所有使用这个特殊类型的地方都被替换成为真实的类实例的类型。这种类型的改变可能影响到数据流分析器对后续指令的分析过程。

在存储这个特殊类型的时候，指令序号必须一起存储起来。因为操作数栈中有可能会同时出现多于 1 个实例已被创建，但未被初始化的情况，譬如下面这个例子：

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

如果使用 Java 虚拟机指令集序列来实现这个例子的代码，那在执行时，操作数栈中就会同时存在 2 个未初始化的 InputStream 实例。当某个类实例的实例初始化方法被调用，那在操作数栈或局部变量表中的代表那个被初始化类实例的对象将会被替换掉。

一个有效地指令集序列里，在回向分支（Backwards Branch）的操作数栈或局部变量表中，以及被 finally 语句块或异常处理器所保护的代码的局部变量表中都不允许出现未被初始化的对象。否则，验证器将可能会被一些有歧义的代码片段所干扰，误以为某些类实例在它需要被初始化的时候已经被初始化过，而实际上，这个被初始化的实例已经通过前面某个循环被创建了。

4.10.2.5 异常和 finally

为了实现 try-finally 结构，在版本号小于或等于 50.0 的 Java 语言编译器中，可以[①]使用将两种特殊指令：jsr（“跳转到程序子片段”）和 ret（“程序子片段返回”）组合的方式来生成 try-finally 结构的 Java 虚拟机代码。这样的 finally 语句以程序子片段的方式嵌入到 Java 虚拟机方法代码中，有点像异常处理器的代码那样。当使用 jsr 指令来调用程序子片段时，该指令会把程序子片段结束后应返回的地址值压入操作数栈中，以便在 jsr 之后的指令能被正确执行。这个地址值会作为 returnAddress 类型数据存放于操作数栈上。程序子片段的代码中会把返回地址存放在局部变量中，在程序子片段执行结束时，ret 指令从局部变量中取回返回地址并将执行的控制权交给返回地址处的指令。

程序有多种不同的执行路径会执行到 finally 语句（即代表 finally 语句的程序子片段被调用）。如果 try 中全部语句正常地完成的话，那在执行下一条指令之前，会通过 jsr 指令来调用 finally 的程序子片段。如果在 try 语句中遇到 break 或 continue 关键字把程序执行权转移到 try 语句之外的话，也会保证在跳转出 try 之前使用 jsr 指令来调用 finally 的程序子片段。如果 try 语句中遇到了 return，代码的行为如下：

[①] 译者注：这里写“可以使用 jsr 和 ret 指令”，但在 Oracle JDK 的编译器里，很久之前（JDK 1.4.2）就已经不再使用这两条指令来实现 try-finally 语法结构了。

1．如果有返回值的话，将返回值保存在局部变量中。

2．执行 jsr 指令将控制权转到给 finally 语句中。

3．在 finally 执行完成后，返回事先保存在局部变量中的值。

编译器会构造特殊的异常处理器，来保证当 try 语句中有异常发生时，它会拦截住任何类型的异常。如果在 try 语句中有异常抛出，异常处理器的行为是：

1． 将异常保存在局部变量中。

2． 执行 jsr 指令将控制权转到给 finally 语句中。

3． 在执行完 finally 语句后，重新抛出这个事先保存好的异常。

如果想了解更多关于 try-finally 语法结构的实现，请参考 3.13 节 “编译 finally 语句块”的内容。

finally 语句中的代码也给验证器带来了一些特殊的问题。通常情况下，如果可以通过多条路径抵达一个特殊的指令或是由这些路径找到的某个特殊包含一些不兼容的值的局部变量，那么这个局部变量是不可用的。然而，由于 finally 语句可以在不同的地方被调用，也会导致一些不同的情况：

- 如果从异常处理器处调用，就会带着一个包含异常实例的局部变量。
- 如果从 return 处调用，那么某个局部变量中应该包含着方法返回值。
- 如果从 try 语句的结尾处调用，那么某些局部变量的值可能是不明确的。

验证 finally 语句，不仅要保证 finally 语句本身的代码通过验证，而且在更新完所有 ret 指令的后续指令状态后，验证器还得注意到：异常处理器的局部变量中应该有一个异常实例，return 代码期望的局部变量是返回值而不是未确定的值。

验证 finally 语句中的代码是很复杂的，但几个基本的思路如下：

- 每个保持追踪 jsr 目标的指令都需要能到达那个目标指令。对于大部分代码来说，这个列表是空的。对于 finally 语句中的代码来说，列表的长度应该是 1。对于多级嵌入 finally 代码来说，列表的长度应该大于 1。
- 对于每条指令及每条 jsr 指令将要转向到那条指令，在 jsr 指令执行后，就有一个位向量（Bit Vector）记录着所有对局部变量的访问及修改。
- 执行 ret 指令就意味着从程序子片段中返回，这应该是唯一的一条从程序子片段中返回的路径。两个不同的程序子片段是不同将 ret 指令的执行结果归并到一起。
- 为了对 ret 指令实施数据流分析，需要进行一些特殊处理。因为验证器知道程序子片段

中将从哪些指令中返回，所以它可以找出调用程序子片段的所有的 jsr 指令，并将它们对应的 ret 指令的操作数栈和局部变量表状态合并。对于合并局部变量表时使用的特殊值定义如下：

- 如果位向量（前面定义过）表明局部变量在程序子片段中被访问或修改过，那么就使用执行 ret 时局部变量的值的类型。
- 对于其它局部变量，使用执行 jsr 指令之前的局部变量的类型。

4.11 Java 虚拟机限制

下面为隐含在 Class 文件格式中的 Java 虚拟机限制：

1. 每个类或接口的常量池项最多为 65535 个，它是由 ClassFile 结构（§4.1）中的 16 位 constant_pool_count 字段的值决定。这限制了单个类或接口的复杂度。
2. 方法调用时创建的栈帧的局部变量表中的最大局部变量个数 65535 个，它是由方法代码所处的 Code 属性（§4.7.3）中的 max_locals 项的值和 Java 虚拟机指令集的 16 位局部变量索引所决定。注意，每个 long 和 double 类型都被认为会使用两个局部变量位置并占据 max_locals 中的两个单元，所以使用这些类型时，局部变量的限制的最大值就会相应地减少。
3. 类或接口中可以声明的字段数最多为 65535 个，它是由 ClassFile 结构（§4.1）中 fields_count 项的值所决定。注意，ClassFile 结构中的 fields_count 项的值不包含从父类或父接口中继承下来的字段。
4. 类或接口中可以声明的方法数最多为 65535 个，它是由 ClassFile 结构（§4.1）中的 methods_count 项的值所决定。注意，ClassFile 结构中的 methods_count 项的值不包含从父类或父接口中继承下来的方法。
5. 类或接口的直接父接口最多为 65535 个，它是由 ClassFile 结构（§4.1）中的 interfaces_count 项的值所决定。
6. 方法栈帧（§2.6）中的操作数栈的大小深度为 65535， 它是由 Code 属性（§4.7.3）的 max_stack 字段值来决定。需要注意的是每个 long 和 double 类型的数据被认为占用 max_locals 中的两个单元，所以使用这些类型时，操作数栈的限制的最大值就会相应地减少。

7. 数组的维度最大为 255 维，这由 multianewarray 指令的 dimensions 操作码及§4.9.1 和§4.9.2 节 multianewarray，anewarray 和 newarray 指令的约束来决定。
8. 方法的参数最多有 255 个，它是由方法描述符（§4.3.3）的定义所限制，如果方法调用是针对实例或接口方法，那么这个限制也包含着占有一个单元的 this。注意对于定义在方法描述符中的参数长度来说，每个 long 和 double 参数都会占用两个长度单位，所以如果有这些类型的话，最终的限制的最大值将会变小。
9. 字段和方法名称、字段和方法描述符以及其它常量字符串值（由 ConstantValue 属性（§4.7.2）引用的值）最大长度为 65535 个字符，它是由 CONSTANT_Utf8_info 结构（§4.4.7）的 16 位无符号 length 项决定。需要注意的是，这里的限制是已编码字符串的字节数量而不是被编码的字符数量。UTF-8 一般用两个或三个字节来编码字符，因此，当字符串中包含多字节字符时，会受到更大的约束。

第 5 章 加载、链接与初始化

Java 虚拟机动态地加载、链接与初始化类和接口。加载是根据特定名称查找类或接口类型的二进制表示（Binary Representation），并由此二进制表示创建类或接口的过程。链接是为了让类或接口可以被 Java 虚拟机执行，而将类或接口并入虚拟机运行时状态的过程。类或接口的初始化是指执行类或接口的初始化方法<clinit>（§2.9）。

在这章里，5.1 节描述 Java 虚拟机如何从类或接口的二进制表示中得到符号引用。5.2 节解释 Java 虚拟机启动时会有怎样的加载、链接和初始化过程。5.3 节详述了类和接口的二进制表示是如何通过类加载器加载并由此创建类和接口。5.4 描述链接过程。5.5 节详述类和接口是如何被初始化的。5.6 节介绍绑定本地方法的概念。最后 5.7 节会说到 Java 虚拟机的退出时机。

5.1 运行时常量池

Java 虚拟机为每个类型都维护一个常量池（§2.5.5）。它是 Java 虚拟机中的运行时数据结构，像传统编程语言实现中的符号表一样有很多用途。

当类或接口创建时（§5.3），它的二进制表示中的 constant_pool 表（§4.4）被用来构造运行时常量池。运行时常量池中的所有引用最初都是符号引用。这些符号引用来自于类或接口的二进制表示的如下结构中：

- 某个类或接口的符号引用来自于类或接口二进制表示中的 CONSTANT_Class_info 结构（§4.4）。这种引用提供的类或接口的名称如同 Class.getName()方法返回值的格式一样，也就是说：
  - 对于非数组的类或接口，那就是类或接口的二进制名称（§4.2.1）。
  - 对于一个 M 维的数组类，名称会以 M 个 ASCII 字符"["开头，随后是数组元素类型的表示：
    - 如果数组的元素类型是 Java 原始类型之一，它就以相应的字段描述符（§4.3.2）来表示。
    - 否则，如果数组元素类型是某种引用类型，它就以 ASCII 字符"L"加上二进制名

称，并以 ASCII 字符";"结尾的字符串表示。

- 在本章中，无论何时提到类或接口的名称，读者都可以按 Class.getName 方法的返回值的格式来理解。
- 类或接口的某个字段的符号引用来自于类或接口二进制表示中的 CONSTANT_Fieldref_info 结构（§4.4.2）。这种引用包含了字段的名称和描述符，及指向字段所属类或接口的符号引用。
- 类中某个方法的符号引用来自于类或接口二进制表示中的 CONSTANT_Methodref_info 结构（§4.4.2）。这种引用提供方法的名称和描述符，及指向方法所属类的符号引用。
- 接口的某个方法的符号引用来自于类或接口二进制表示中的 CONSTANT_InterfaceMethodref_info 结构（§4.4.2）。这种引用提供接口方法的名称和描述符，及指向方法所属接口的符号引用。
- 方法句柄（Method Handle）的符号引用来自于类或接口二进制表示中的 CONSTANT_MethodHandle_info 结构（§4.4.8）。
- 方法类型（Method Type）的符号引用来自于类或接口二进制表示中的 CONSTANT_MethodType_info 结构（§4.4.9）。
- 调用点限定符（Call Site Specifier）的符号引用来自于类或接口二进制表示的 CONSTANT_InvokeDynamic_info 结构（§4.4.10）。这种引用包含了：
  - 方法句柄的符号引用，为 invokedynamic 指令的引导方法（Bootstrap Method）来提供服务。
  - 一系列符号引用（到类、方法类型和方法句柄）、字符常量和运行时常量（如 Java 原生数值类型的值），它们将作为静态参数（Static Arguments）提供给引导方法。
  - 调用方法的名称与描述符

另外，一些运行时数值它不是由符号引用，而是由 constant_pool 表中某些项的值来决定：

- 字符常量表示 String 类实例的一个引用，它来自于类或接口二进制表示的 CONSTANT_String_info 结构（§4.4.3）。CONSTANT_String_info 结构包含了由

Unicode 码点（Code points）[①]序列来组成字符常量。

Java 语言需要全局统一的字符常量（这就意味着如果不同字面量（Literal）包含着相同的码点序列，就必须引用着相同的 String 类的实例（JLS §3.10.5））。此外，在任意字符串上调用 String.intern 方法，如果那个字符串是字面量的话，方法的结果应当是对相同字面量的 String 实例的引用。因此，

```
("a" + "b" + "c").intern()  ==  "abc"
```

必须返回 true。

为了得到字符常量，Java 虚拟机需要检查 CONSTANT_String_info 结构中的码点序列。

- 如果 String.intern 方法以前曾经被某个与 CONSTANT_String_info 结构中的码点序列一样的 String 实例调用过，那么此次字符常量获取的结果将是一个指向相同 String 实例的引用。
- 否则，一个新的 String 实例会被创建，它会包含着指定 CONSTANT_String_info 结构中 Unicode 码点序列；字符常量获取的结果是指向那个新 String 实例的引用。最后，新 String 实例的 intern 方法被 Java 虚拟机自动调用。

- 其它运行时常量值来自于类或接口二进制表示的 CONSTANT_Interger_info、CONSTANT_Float_info、CONSTANT_Long_info 或是 CONSTANT_Double_info 结构（§4.4.4，§4.4.5）。请注意，这里 CONSTANT_Float_info 结构的值以 IEEE 754 单精度浮点格式表示，CONSTANT_Double_info 结构的值以 IEEE 754 双精度浮点格式表示（§4.4.4，§4.4.5）。来自这些结构的运行时常量值必须可以用 IEEE 754 单或双精度浮点格式分别表示。

在类或接口的二进制表示中，constant_pool 表中剩下的结构还有 CONSTANT_NameAndType_info（§4.4.6）和 CONSTANT_Utf8_info（§4.4.7），它们被间接用以获得对类、接口、方法、字段、方法类型和方法句柄的符号引用，或是在需要得到字符常量和调用点限定符时被引用。

[①] 译者注：码点是指组成字符集代码空间的数值表示，譬如 ASCII 有 0x0 至 0x7F 共 128 个码点，扩展 ASCII 有 0x0 至 0xFF 共 256 个码点，而 Unicode 则有 0x0 至 0x10FFFF 共 1114112 个码点。

5.2 虚拟机启动

Java 虚拟机的启动是通过引导类加载器（Bootstrap Class Loader §5.3.1）创建一个初始类（Initial Class）来完成，这个类是由虚拟机的具体实现指定。紧接着，Java 虚拟机链接这个初始类，初始化并调用它的 public void main(String[])方法。之后的整个执行过程都是由对此方法的调用开始。执行 main 方法中的 Java 虚拟机指令可能会导致 Java 虚拟机链接另外的一些类或接口，也可能会调用另外的方法。

可能在某种 Java 虚拟机的实现上，初始类会作为命令行参数被提供给虚拟机。当然，虚拟机实现也可以利用一个初始类让类加载器依次加载整个应用。初始类当然也可以选择组合上述的方式来工作。

5.3 创建和加载

如果要创建标记为 N 的类或接口 C，就需要先在 Java 虚拟机方法区（§2.5.4）上为 C 创建与虚拟机实现规定相匹配的内部表示。C 的创建是由另外一个类或接口 D 所触发的，它通过自己的运行时常量池引用了 C。当然，C 的创建也可能是由 D 调用 Java 核心类库（§2.12）中的某些方法而触发，譬如使用反射等。

如果 C 不是数组类型，那么它就可以通过类加载器加载 C 的二进制表示来创建（参见第 4 章，“Class 文件格式”）。数组类型没有外部的二进制表示；它们都是由 Java 虚拟机创建，而不是通过类加载器加载的。

Java 虚拟机支持两种类加载器：Java 虚拟机提供的引导类加载器（Bootstrap Class Loader）和用户自定义类加载器（User-Defined Class Loader）。每个用户自定义的类加载器应该是抽象类 ClassLoader 的某个子类的实例。应用程序使用用户自定义类加载器是为了便于扩展 Java 虚拟机的功能，支持动态加载并创建类。当然，它也可以从用户自定义的数据来源来获取类的二进制表示并创建类。例如，用户自定义类加载器可以通过网络下载、动态产生或是从一个加密文件中提取类的信息。

类加载器 L 可能会通过直接创建或是委托其它类加载器的方式来创建 C。如果 L 直接创建 C，我们就可以说 L 定义了（Define）C，或者，L 是 C 的定义加载器（Defining Loader）。

当一个类加载器把加载请求委托给其它的类加载器后，发出这个加载请求的加载器与最终完成

加载并定义类的类加载器不需要是同一个加载器。如果 L 创建了 C，它可能直接创建了 C 或者是委托了加载请求，我们可以说 L 导致（Initiate）了 C 的加载，或者说，L 是 C 的初始加载器（Initiating Loader）。

在 Java 虚拟机运行时，类或接口不仅仅是由它的名称来确定，而是由一个值对：二进制名称（§4.2.1）和它的定义类加载器共同确定。每个这样的类或接口都归属于独立的运行时包结构（Runtime Package）。类或接口的运行时包结构由包名及类或接口的定义类加载器来决定。

Java 虚拟机通过下面三个过程中之一来创建标记为 N 的类或接口 C：

- 如果 N 表示一个非数组的类或接口，可以用下面的两个方法之一来加载并创建 C：
  - 如果 D 是由引导类加载器所定义，那么引导类加载器初始加载 C（§5.3.1）。
  - 如果 D 是由用户自定义类加载器所定义，那么此用户自定义类加载器也用来初始加载 C（§5.3.2）。
- 如果 N 表示一个数组类。数组类是由 Java 虚拟机而不是类加载器创建。然而，在创建数组类 C 的过程中，D 的定义类加载器也要被用到。

如果在类加载过程中有错误产生，某个 LinkageError 的子类的实例将被抛出。抛出位置应当是当前正在（直接或间接）加载类或接口的那段程序中。

如果 Java 虚拟机曾经试图在 D 的验证（§5.4.1）或解析（§5.4.3）阶段、但又还没有进行初始化（§5.5）时加载 C 类，当用于加载 C 的初始类加载器抛出 ClassNotFoundException 实例时，Java 虚拟机在 D 中必须抛出 NoClassDefFoundError 异常，它的 cause 字段中就保存了那个 ClassNotFoundException 异常实例。

（这里有个需要注意的地方，作为解析（§5.3.5）过程的一部分，类加载器会递归加载它的父类。如果类加载器在加载父类时因失败而产生 ClassNotFoundException 异常，就应该被包装成 NoClassDefFoundError 异常。）

请注意：一个功能良好的类加载器应当保证下面三个属性：

- 给定相同的名称，类加载器应当总是返回相同的 Class 对象。
- 如果类加载器 L1 将加载类 C 的请求委托给另外的类加载器 L2，那么对于下列的一种类型 T：它可以是 C 的直接父类或直接接口、或是 C 中的字段类型、或是 C 中方法或构造函数的通用参数、或是 C 中方法的返回值，L1 和 L2 都应当返回相同的 Class 对象。
- 如果某个用户自定义的类加载器预先加载了某个类或接口的二进制表示，或是加载一组相关的类型，并在加载时出现异常，那它必须在程序的某个点反映出加载时的错误。

我们通常使用标识<N，Ld>来表示一个类或接口，这里的 N 表示类或接口的名称，Ld 表示类或接口的定义类加载器。我们也可以使用标识 N^{Li} 来表示一个类或接口，这里的 N 表示类或接口的名称，Li 表示类或接口的初始类加载器。

5.3.1 使用引导类加载器来加载类型

下列步骤描述使用引导类加载器加载并创建标记为 N 的非数组类型的类或接口 C。

首先，Java 虚拟机检查引导类加载器是否是已加载过的标记为 N 的类或接口的初始加载器。如果是的话，这个类或接口就是 C，并且不再创建其它类型。

否则，Java 虚拟机将参数 N 传递给引导类加载器的特定方法，以平台相关的方式搜索 C 的描述。典型的情况是，类或文件会被表示为树型文件系统中的某个文件，类或接口的名称就是此文件的路径名。

此处需要注意，搜索过程没有任何保证一定可以找到 C 的有效描述。所以加载的过程必须检查到这些错误：

- 如果没有找到与 C 相关的描述，加载过程要抛出 ClassNotFoundException 异常。

之后，Java 虚拟机根据 5.3.5 节的算法描述，尝试通过引导类加载器加载标识为 N 的描述，加载完成的类就是 C。

5.3.2 使用用户自定义类加载器来加载类型

下列步骤描述使用用户自定义类加载器 L 来加载标记为 N 的类或接口，然后创建这个非数组类型的类或接口 C。

首先，Java 虚拟机检查 L 是否为已经加载过的标识为 N 的类或接口的初始加载器。如果是的话，那个类或接口就是 C，不用再创建其它类了。

否则 Java 虚拟机会调用 L 的 loadClass(N)[①]方法。 这次调用的返回值就是创建好的类或

[①] 自从 JDK 版本 1.1 开始，Oracle 的 Java 虚拟机实现是通过调用类加载器的 loadClass 方法来加载类或接口。方法 loadClass 的参数就是类或接口的名称。同时也存在着另外一个有两个参数的 loadClass 方法，

接口 C。Java 虚拟机会记录下 L 是 C 的初始加载器（§5.3.4）。这节其余的部分会更详细地描述这个过程。

当通过类或接口 C 的名称 N 为参数去调用类加载器 L 的 loadClass 方法，L 必须执行下面两种操作之一来加载 C：

1. 类加载器 L 可以创建一个如 ClassFile 结构（§4.1）的字节数组用来表示 C；然后必须调用 ClassLoader 的 defineClass 方法。调用 defineClass 方法会让 Java 虚拟机使用 5.3.5 节所描述的算法通过 L 由字节数组得到标记为 N 的类或接口。

2. 类加载器 L 可能将对 C 的加载委托给其它的类加载器 L'。这是通过直接或间接的方式传递参数 N 来调用 L'的方法（也就是 loadClass 方法）。这次调用会产生 C。

不管第 1 步或第 2 步中，如果类加载器 L 因为任何原因不能加载标识为 N 的类或接口，它必须抛出 ClassNotFoundException 异常。

5.3.3 创建数组类

下列步骤描述使用类加载器 L 来创建标记为 N 的数组类 C 的过程。类加载器 L 既可以是引导类加载器，也可以是用户自定义的类加载器。

如果 L 已经被记录成某个与 N 相同的组件类型（Component Type，§2.4）的数组类的初始加载器，那么类就是 C，不再创建新的数组类了。否则的话，创建 C 的过程就遵循下面的步骤：

1. 如果组件类型是引用类型，那就遵循这节（§5.3）的算法使用 L 递归加载和创建 C 的组件类型。

2. Java 虚拟机使用显式的组件类型和数组维度来创建新的数组类。如果组件类型是引用类型，C 就被标记为它已经被该组件类型的定义类加载器定义过。否则，C 就被标记为它被引导类加载器定义过。不管哪种情况，Java 虚拟机都会把 L 记录为 C 的初始加载器（§5.3.4）。如果数组的组件类型是引用类型，数组类的可见性就由组件类型的可见性决定，否则，数组类的可见性将被默认为 public。

第二个参数是 boolean 显示类或接口是否被链接过。只有 JDK 1.0.2 版本支持两个参数的方法，Oracle 的 Java 虚拟机实现也依赖它来链接已经加载过的类或接口。自 JDK 1.1 之后，Oracle 的 Java 虚拟机直接链接类或接口，而不再依赖于类加载器。

5.3.4 加载限制

类加载器需要特别考虑到类型的安全链接问题。一种可能出现的情况是，当两个不同的类加载器初始加载标记为 N 的类或接口时，在每个加载器里 N 表示着不同的类或接口。

当类或接口 C=<N1，L1>包含另外一个类或接口 D=<N2，L2>的字段或方法的符号引用时，这个符号引用会包含字段的特定描述符，或方法的参数和返回值类型。很重要的一点是，不管是 L1 还是 L2 加载，任何字段或方法描述符类型 N 都应该表示相同的类或接口。

为了确保这个原则，Java 虚拟机在准备（§5.4.2）和解析（§5.4.3）阶段强制“$N^{L1}=N^{L2}$”形式的加载约束（Loading Constraints）。为了强制实施这个约束，Java 虚拟机会在类型加载的某些关键点（§5.3.1、§5.3.2、§5.3.3、§5.3.4、§5.3.5）记录下每个特定类的初始加载器。在记录一个加载器是某个类的初始加载器后，Java 虚拟机会立即检查是否有加载过程违反了这一约束。如果有违约情况发生，此次记录过程将被撤销，Java 虚拟机抛出 LinkageError，引起记录产生的那次加载操作也同样会失败。

与之相似的是，在强制执行了加载约束（参见§5.4.2、§5.4.3.2、§5.4.3.3、§5.4.3.4）之后，虚拟机必须立即去检查是否有违约情况发生。如果有，最新的那个加载约束就会被撤销，Java 虚拟机抛出 LinkageError 异常，引入约束（解析或准备，视情况而定）的那些操作也会失败。

下列步骤描述 Java 虚拟机检查加载约束被违约的发生的条件。加载约束被违反，当且仅当下面的四个条件同时符合：

- 类加载器 L 被 Java 虚拟机记录为标记为 N 的类 C 的初始加载器。
- 类加载器 L′被 Java 虚拟机记录为标记为 N 的类 C′的初始加载器。
- 强加的限制定义了等价关系 $N^{L}=N^{L'}$
- C ≠ C′

对于类加载器和类型安全的进一步讨论已经超出了我们这篇规范的范畴。如果读者想了解更多的细节，请参阅由 Sheng Liang 和 Gilad Bracha 所著的《Dynamic Class Loading in the Java Virtual Machine》（1998 年 ACM SIGPLAN 关于面向对象编程系统、语言及应用的会议记要）。

5.3.5 从 Class 文件中获取类

下列步骤描述如何使用类加载器 L 从 Class 文件格式的描述中得到标记为 N 的非数组类或接口的 Class 对象。

1．首先，Java 虚拟机检查 L 是否被记录为标记为 N 的类或接口的初始加载器。如果是，这次创建的尝试动作是无效的，且加载动作抛出 LinkageError 异常。

2．否则，Java 虚拟机尝试解析二进制表示。但是，这个二进制表示可能不是 C 的有效描述。这个阶段的加载动作必须能够检测出下列错误：

- 如果发现这个可能的 C 的描述不符合 ClassFile 结构（§4.1、§4.8），加载过程将抛出 ClassFormatError 异常。
- 否则，如果二进制表示的版本号不在虚拟机所支持的最低版本与最高版本之间（§4.1），加载动作会抛出 UnspportedClassVersionError[①]异常。
- 如果描述不能真正表示名称为 N 的类，那么加载过程就会抛出 NoClassDefError 异常或是它的子类异常。

3．如果 C 有一个直接父类，由 C 到它的直接父类的符号引用就需要使用 5.4.3.1 节描述的算法来解析。需要注意，如果 C 是一个接口，它必须以 Object 作为它的直接父类，并且 Object 必须是已经被加载过的。只有 Object 类没有自己的直接父类。

类或接口解析过程中的异常可以被当作是加载阶段的异常抛出。除此之外，加载阶段还必须可以检查出下列错误：

- 如果类或接口 C 的直接父类事实上是一个接口，那么加载过程就必须抛出 IncompatibleClassChangeError 异常。
- 否则，如果 C 的父类是 C 自己，加载过程就必须抛出 ClassCircularityError 异常。

4．如果 C 有一些直接父接口，由 C 到它的直接父接口的符号引用就需要使用 5.4.3.1 节描述的算法来解析。

在类或接口解析过程中产生的异常可以当作是加载阶段的异常抛出。除此之外，加载过程还必

[①] UnsupportedClassVersionError，它是 ClassFormatError 的子类，它可以很容易地从 ClassFormatError 异常中辨别出具体异常原因是由于 Java 虚拟机尝试加载的二进制表述使用了不支持的 Class 版本号。在 JDK 1.1 版本之前，如果发生不支持的版本问题，就会有 NoClassDefFoundError 或 ClassFormatError 被抛出，且取决于类是由引导类或用户自定义类加载器所加载。

须可以检查到下列错误：

- 如果类或接口 C 的直接父接口实际上不是一个接口的话，那么加载过程就必须抛出 IncompatibleClassChangeError 异常。
- 否则，如果 C 的某个父接口是 C 自己，加载过程必须抛出 ClassCircularityError 异常。

5．Java 虚拟机标记 C 的定义类加载器是 L，并且记录下 L 是 C 的初始加载器（§5.3.4）。

5.4 链接

链接类或接口包括验证和准备类或接口、它的直接父类、它的直接父接口、它的元素类型（如果是一个数组类型）及其它必要的动作。而解析这个类或接口中的符号引用是链接过程中可选的部分。

《Java 虚拟机规范》允许灵活地选择链接（并且会有递归加载）发生的时机，但必须保证下列几点成立：

- 在类或接口被链接之前，它必须是被成功地加载过。
- 在类或接口初始化之前，它必须是被成功地验证及准备过．
- 程序的直接或间接行为可能会导致链接发生，链接过程中检查到的错误应该在请求链接的程序处被抛出。

例如，Java 虚拟机实现可以选择只有在使用类或接口中符号引用时才去逐一解析它（延迟解析），或是当类在验证时就解析每个引用（预先解析）。这意味着在一些虚拟机实现中，在类或接口被初始化动作开始后，解析动作可能还正在进行。不管使用哪种策略，解析过程中的任何错误都必须被抛出，抛出的位置是在通过直接或间接调用而导致解析过程发生的程序处。

由于链接过程会涉及到新数据结构的内存分配，它也可能因为发生 OutOfMemoryError 异常而导致失败。

5.4.1 验证

验证（Verification，§4.10）阶段用于确保类或接口的二进制表示结构上是正确的

（§4.9）。验证过程可能会导致某些额外的类和接口被加载进来（§5.3），但不应该会导致它们也需要验证或准备。

如果类或接口的二进制表示不能满足 4.9 节（Java 虚拟机代码限制）中描述的静态或结构上的约束，那就必须在导致验证发生的程序调用处被抛出 VerityError 异常。

如果 Java 虚拟机尝试验证类或接口，却因为 LinkageError 或其子类的实例而导致验证失败，那么随后对于此类或接口的验证尝试总是会因为第一次尝试失败的同样原因而失败。

5.4.2 准备

准备（Preparation）阶段的任务是为类或接口的静态字段分配空间，并用默认值初始化这些字段（§2.3，§2.4）。这个阶段不会执行任何的虚拟机字节码指令。在初始化阶段（§5.5）会有显式的初始化器来初始化这些静态字段，所以准备阶段不做这些事情。

在某个类或接口 C 的准备阶段，Java 虚拟机也会强制进行加载约束（§5.3.4）。假定 L1 是 C 的定义加载器。对于每个声明在 C 中的方法 m，它重写（§5.4.5）声明在 C 的父类或父接口<D，L2>中的某个方法，Java 虚拟机强制执行下面的加载约束：

- 给定 m 的返回值类型是 Tr，
- 并且给定 m 的形参类型从 Tf*1*，…，Tf*n*，
- 如果 Tr 不是数组类型，那么 T0 代替 Tr；不然的话，T0 就可以表示 Tr 的元素类型（§2.4）。
- 由 i 表示 1 到 n，如果 Tf*i* 不是数组类型，Ti 就表示 Tf*i*；不然，Ti 就是 Tfi 的元素类型（§2.4）。
- 那么对于 i = 0…n，$Ti^{L1} = Ti^{L2}$ 就应该成立（§5.3.4）。

此外，如果 C 实现了它的父接口<I，L3>中的方法 m，但 C 自己是没有这个方法 m 声明的，如果 C 有父类<D，L2>，它声明实现了由 C 继承的那个方法 m。Java 虚拟机强制执行下面的约束：

给定 m 的返回值类型是 Tr，并且给定 m 的形参类型从 Tf*1*，…，Tf*n*，如果 Tr 不是数组类型，那么 T0 代替 Tr；不然的话，T0 就可以表示 Tr 的元素类型（§2.4）。由 i 表示 1 到 n，如果 Tf*i* 不是数组类型，Ti 就表示 Tf*i*；不然，Ti 就是 Tfi 的元素类型（§2.4）。那么对于 i = 0…n，$Ti^{L2} = Ti^{L3}$ 就应该成立（§5.3.4）。

在类的对象创建之后的任何时间，都可以进行准备阶段，但它得确保一定要在初始化阶段开始前完成。

5.4.3 解析

Java 虚拟机指令 anewarray、checkcast、getfield、getstatic、instanceof、nvokedynamic、invokeinterface、invokespecial、invokestatic、invokevirtual、ldc、ldc_w、multianewarray、new、putfield 和 putstatic 将符号引用指向运行时常量池。执行上述任何一条指令都需要对它的符号引用的进行解析。

解析（Resolution）是根据运行时常量池的符号引用来动态决定具体的值的过程。

当碰到已经因某个 invokedynamic 指令而解析过的符号引用时，并不意味着对于其它 invokedynamic 指令，相同的符号引用也被解析过。

但是对于上述的其它指令，当碰到某个因被其他指令引用而解析的符号引用时，就表示对于其它所有非 invokedynamic 的指令来说，相同的符号引用已经被解析过了。

（上面的内容暗示着，对于特定的某一条 invokedynamic 指令，它的解析过程会返回一个特定的值，这个值是一个与此 invokedynamic 指令相关联的调用点对象。）

虚拟机在解析过程中也可以尝试去重新解析之前已经成功解析过的符号引用。如果有这样的尝试动作，那么它总是会像之前一样解析成功，且总是返回此与引用初次解析时的结果相同的实体。

如果在某个符号引用解析过程中有错误发生，那么就应该在使用（直接或间接）这个符号引用的代码处抛出 IncompatibleClassChangeError 或它的子类异常。

如果在虚拟机解析符号引用时，因为 LinkageError 或它的子类实例而导致失败，那么随后的每次试图对此引用的解析也总会抛出与第一次解析时相同的错误。

通过 invokedynamic 指令指定的调用点限定符的符号引用，在执行这条 invokedynamic 指令被实际执行之前不能被提早解析。

如果解析某个 invokedynamic 指令的时候出错，引导方法在随后的尝试解析时就不会再被重新执行。

上述的某些指令在解析符号引用时，需要有额外的链接检查。譬如，getfield 指令为了成功解析所指的字段的符号引用，不仅得完成 5.4.3.2 节描述的字段解析步骤，还得检查这个字符是不是 static 的。如果这个字段是 static 的，就必须抛出链接时异常。

为了让 invokedynamic 指令成功解析一个指向调用点限定符的符号引用，指定的引导方法就必须正常执行完成并返回一个合适的调用点对象。如果引导方法执行被中断或返回一个不合法的调用点对象，那也必须抛出链接时异常。

链接时异常的产生是由于 Java 虚拟机指令被检查出未按照指令定义中所描述的语义来执行，或没有通过指令的解析过程。对于这些异常，尽管有可能是因为 Java 虚拟机指令执行的问题而导致的，但这类问题依然会被当作是解析失败的问题。

后续各章节将描述对于类或接口 D 的运行时常量池（§5.1）中的符号引用进行解析的过程。解析细节会因为符号引用类型的不同而有所区别。

5.4.3.1 类与接口解析

Java 虚拟机为了解析 D 中对标记为 N 的类或接口 C 的未解析符号引用，就会执行下列步骤：

- D 的定义类加载器被用来创建标记为 N 的类或接口。这个类或接口就是 C。在创建类或接口的过程中，如果有任何异常发生，可以被认为是类和接口解析失败而抛出。过程的细节在 5.3 节进行描述。
- 如果 C 是数组类并且它的元素类型是引用类型，那么表示元素类型的类或接口的符号引用会按照 5.4.3.1 节的算法通过递归调用来解析。
- 最后，检查 C 的访问权限：
- 如果 C 对 D 是不可见的（§5.4.4），类或接口解析抛出 IllegalAccessError 异常。

第 4 步这种情况有可能发生，譬如，C 是一个原来被声明为 public 的类，但它在 D 编译后被改成非 public。

如果第 1 步和第 2 步成功但是第 3 步失败，C 仍然是有效、可用的。当然，整个解析过程将被认为是失败的，并且 C 将会拒绝 D 访问请求。

5.4.3.2 字段解析

为了解析 D 中一个未被解析的类或接口 C 的字段的符号引用，字段描述符中提供的对 C 的符号引用应该首先被解析（§5.4.3.1）。因此，在解析类或接口引用时发生的任何异常都可以当作

是解析字段引用的异常一样被抛出。如果到 C 的引用可以被成功地解析，解析字段引用时发生的异常就可以被抛出。

当解析一个字段引用时，字段解析会在 C 和它的父类中先尝试查找这个字段：

1．如果 C 中声明的字段与字段引用有相同的名称及描述符，那么此次查找成功。字段查找的结果是 C 中那个声明的字段。

2．不然的话，字段查找就会被递归应用到类或接口 C 的直接父接口上。

3．否则，如果 C 有一个父类 S，字段查找也会递归应用到 S 上。

4．如果还不行，那么字段查找失败。

如果字段查找失败，字段解析会抛出 NoSuchFieldError 异常。如果字段查找成功，但是引用的那个字段对 D 是不可见的（§5.4.4），字段解析会抛出 IllgalAccessError 异常。

另外，假设<E，L1>是真正声明所引用字段的类或接口，L2 是 D 的定义加载器。给定引用字段的类型是 Tf，设定 Tf 是非数组类型时 T=Tf，或 Tf 是数组时，T 是它的元素类型（§2.4）。Java 虚拟机必须强制执行这个的加载约束：$T^{L1} = T^{L2}$（§5.3.4）。

5.4.3.3 普通方法解析

为了解析 D 中一个对类或接口 C 中某个方法的未解析符号引用，方法引用中包含的对 C 的符号引用应该首先被解析（§5.4.3.1）。因此，在解析类的符号引用时出现的任何异常都可以看作解析方法引用的异常而被抛出。如果对 C 的引用成功地被解析，方法引用解析相关的异常就可以被抛出。

当解析一个方法引用时：

1．首先检查方法引用中的 C 是否类或接口。

- 如果 C 是接口，那么方法引用就会抛出 IncompatibleClassChangeError 异常。

2．方法引用解析过程会检查 C 和它的父类中是否包含此方法：

- 如果 C 中确有一个方法与方法引用的指定名称相同，并且声明是签名多态方法（Signature Polymorphic Method，§2.9），那么方法的查找过程就被认为是成功的。所有方法描述符中所提到的类也需要解析（§5.4.3.1）。

  这次解析的方法是签名多态方法。对于 C 来说，没有必要使用方法引用指定的描述符来

声明方法。

- 否则，如果 C 声明的方法与方法引用拥有同样的名称与描述符，那么方法查找也是成功。
- 如果 C 有父类的话，那么如第 2 步所述的查找方式递归查找 C 的直接父类。

3．另外，方法查找过程也会试图从 C 的父接口中去定位所引用的方法。

- 如果 C 的某个父接口中确实声明了与方法引用相同的名称与描述符的方法，那么方法查找成功。
- 否则，方法查找失败。

如果方法查找失败，方法的解析过程就会抛出 NoSuchMethodError 异常。如果方法查找成功且方法是 abstract 的，但 C 类不是 abstract 的，那么此次方法解析就会抛出 AbstractMethodError 异常。另外，如果引用的方法对 D 是不可见的（§5.4.4），方法解析就会抛出 IllegalAccessError 异常。

如果<E，L1>是所引用方法 m 所在的类或接口，L2 是 D 的定义加载器。假定 m 的返回值类型是 Tr，并且假定 m 的形参类型从 Tf*1*，…，Tf*n*。如果 Tr 不是数组类型，那么 T0 代替 Tr；不然的话，T0 就可以表示 Tr 的元素类型（§2.4）。由 i 表示 1 到 n 的整数，如果 Tfi 不是数组类型，Ti 就表示 Tfi；否则，Ti 就是 Tfi 的元素类型（§2.4）。Java 虚拟机必须保证对于 i = 0…n，加载约束$Ti^{L1} = Ti^{L2}$能够成立（§5.3.4）。

5.4.3.4 接口方法解析

为了解析 D 一个对 C 中接口方法的未解析符号引用，接口方法引用中到接口 C 的符号引用应该最先被解析（§5.4.3.1）。因此，在解析接口引用时出现的任何异常都可以当作接口方法引用解析的异常而被抛出。如果对接口引用被成功地解析，接口方法引用解析相关的异常就可以被抛出。

当解析接口方法引用时：

- 如果 C 不是一个接口，那么接口方法解析就会抛出 IncompatibleClassChangeError 异常。
- 否则，如果在 C 与它的父接口（或 Object 类）中也不存在与接口方法描述符相同的方法的话，接口方法解析会抛出 NoSuchMethodError 异常。

如果<E，L1>是所引用的接口方法所在的类或接口，且 L2 是 D 的定义加载器。假定 m 的返回

值类型是 Tr，并且假定 m 的形参类型从 Tf1，…，Tfn。如果 Tr 不是数组类型，那么 T0 代替 Tr；不然的话，T0 就可以表示 Tr 的元素类型（§2.4）。由 i 表示 1 到 n，如果 Tfi 不是数组类型，Ti 就表示 Tfi；不然，Ti 就是 Tfi 的元素类型（§2.4）。那么对于 i = 0 …n，Java 虚拟机就要强制执行这样的约束：$Ti^{L1} = Ti^{L2}$（§5.3.4）。

5.4.3.5 方法类型与方法句柄解析

为了解析方法类型（Method Type）的未解析符号引用，此方法类型所封装的方法描述符中所有的类型符号引用（Symbolic References To Classes）都必须先被解析（§5.4.3.1）。因此，在解析这些类型符号引用的过程中如果有任何异常发生，也会当作解析方法类型的异常而被抛出。

解析方法类型的结果是得到一个对 java.lang.invoke.MethodType 实例的引用，它可用来表示一个方法的描述符。

解析方法句柄的符号引用的过程会更为复杂。每个被 Java 虚拟机解析的方法句柄都有一个被称为字节码行为（Bytecode Behavior）的等效指令序列（Equivalent Instruction Sequence），它由方法句柄的类型（Kind）值来标识。九种方法句柄的类型值和描述见下表所示。

指令序列到字段或方法的符号引用被标记为：C.x:T。这里的 x 和 T 分别表示字段和方法的名称和描述符（§4.3.2、§4.3.3），C 表示字段或方法所属的类或接口。

类型	描述	字节码行为
1	REF_getField	getfield C.f:T
2	REF_getStatic	getstatic C.f:T
3	REF_putField	putfield C.f:T
4	REF_putStatic	putstatic C.f:T
5	REF_invokeVirtual	invokevirtual C.m: (A*) T
6	REF_invokeStatic	invokestatic C.m: (A*) T
7	REF_invokeSpecial	invokespeical C.m: (A*) T
8	REF_newInvokeSpecial	new C; dup; invokespecial C.<init>:(A*)void

9	REF_invokeInterface	invokeinterface C.m:（A*）T

假设 MH 表示正在被解析的方法句柄（§5.1）的符号引用，那么：

- 设 R 是 MH 中字段或方法的符号引用。

（MH 是从 CONSTANT_MethodHandle 结构中得来，它的 reference_index 项的索引所指向的 CONSTANT_Fieldref，CONSTANT_Methodref 或 CONSTANT_InterfaceMethodref 结构可以得到 R。）

- 设 C 是 R 所引用的类型的符号引用。

（C 是由表示 R 的 CONSTANT_Fieldref，CONSTANT_Methodref 和 CONSTANT_InterfaceMethodref 处 class_index 项引用的 CONSTANT_Class 结构所得到。）

- 设 f 或 m 是 R 所引用的字段或方法的名称。

（f 或 m 是由 R 的 CONSTANT_Fieldref，CONSTANT_Methodref 和 CONSTANT_InterfaceMethodref 结构处 name_and_type_index 项所引用的 CONSTANT_NameAndType 结构得到。）

- 设 T 和（如果是方法的话）A*是 R 所引用的字段或方法的返回值和参数类型序列

（T 和 A*是由得到 R 的 CONSTANT_Fieldref，CONSTANT_Methodref 和 CONSTANT_InterfaceMethodref 结构处 name_and_type_index 项所引用的 CONSTANT_NameAndType 结构得到。）

为了解析 MH，所有 MH 的字节码行为中对类、字段或方法的符号引用都必须进行解析（§5.4.3.1、§5.4.3.2、§5.4.3.3、§5.4.3.4）。那就是说，C、f、m、T 和 A*都需要解析。在解析这些与类、字段或方法相关的符号引用过程中所抛出的异常，都可以看作是解析方法句柄的异常而抛出。

（通常来说，成功解析一个方法句柄需要 Java 虚拟机事先成功解析字节码行为中的所有符号引用。特别的是，到 private 和 protected 成员的方法句柄可以被正常地创建使得相应地正常方法请求成为合法的请求）

如果所有这些符号引用被解析后就会获得一个对 java.lang.invoke.MethodType 实例的引用，就像下表中所述的给定 MH 的类型而去解析方法描述符的行为一样

类型	描述	方法描述符
1	REF_getField	（C）T

2	REF_getStatic	() T
3	REF_putField	(C, T) V
4	REF_putStatic	(T) V
5	REF_invokeVirtual	(C, A*) T
6	REF_invokeStatic	(A*) T
7	REF_invokeSpecial	(C, A*) T
8	REF_newInvokeSpecial	(A*) C
9	REF_invokeInterface	(C, A*) T

方法句柄解析的结果是一个指向 java.lang.invoke.MethodHandle 实例的引用 o，它表示着方法句柄 MH。如果方法 m 有 ACC_VARARGS 标志（§4.6），那么 o 是一个可变元方法句柄；否则，o 是固定元方法句柄。

（可变元方法句柄在调用 invoke 方法时它的参数列表就会有装箱动作（JLS §15.12.4.2），考虑 invokeExact 的调用行为就像 ACC_VARAGS 标志没有被设置一样。）

当方法 m 有 ACC_VARARGS 标志且要么 m 的参数类型序列为空，要么 m 参数类型的最后一个参数不是数组类型，那么方法句柄解析就会抛出 IncompatibleClassChangeError 异常（这表示创建可变元方法句柄失败）。

o 所引用的 java.lang.invoke.MethodHandle 实例的类型描述符是一个 java.lang.invoke.MethodType 的实例，它是之前由方法类型解析时产生的。

（在方法句柄的类型描述符上调用 java.lang.invoke.MethodHandle.invokeExact 方法会与字符码行一样，对栈造成相同的影响。当方法句柄带有有效的参数集合时，它会与相应的字节码行为有相同的影响及返回值）

（Java 虚拟机的实现完全可以不需要内部的方法类型与方法句柄。那就是说，纵使拥有同样结构的对方法类型或方法句柄的两个不同符号引用都可能会获得不同的 java.lang.invoke.MethodType 或 java.lang.invoide.MethodHandle 实例。）

（在 Java SE 平台 API 中，允许在没有字节码行为的时候通过 java.lang.invoke.MethodHandles 类创建方法句柄。具体的行为是由创建 java.lang.invoke.MethodHandles 的方法来决定。例如，当调用一个方法句柄时，就对传入它的参数，并通过传入的值去调用另外一个方法句柄，接着将调用的返回值传回来，并返回传回的值作为它的最终结果。）

5.4.3.6 调用点限定符解析

解析一个未被解析过的调用点限定符（Call Site Specifier）需要下列三个步骤：

- 调用点限定符提供对方法句柄的符号引用，它作为引导方法（Bootstrap Method）向动态调用点提供服务。解析这个方法句柄（§5.4.3.5）是为了获取一个对 java.lang.invoke.MethodHandle 实例的引用。
- 调用点限定符提供了一个方法描述符，记作 TD。它是一个 java.lang.invoke.MethodType 实例的引用。可以通过解析与 TD 有相同的参数及返回值的方法类型（§5.4.3.5）的符号引用而获得。
- 调用点限定符提供零至多个静态参数（Static Arguments），用于传递与特定应用相关的元数据给引导方法。静态参数只要是对类、方法句柄或方法类型的符号引用，都需要被解析，例如调用 ldc 指令，可以分别获取到对 Class 对象、java.lang.invoke.MethodHandle 对象和 java.lang.invoke.MethodType 对象等。如果静态参数都是字面常量，它就是为了获取对 String 对象的引用。

  调用点限定符的解析结果是一个数组，它包含：

  - 到 java.lang.invoke.MethodHandle 实例的引用，
  - 到 java.lang..invoke.MethodType 实例的引用，
  - 到 Class，java.lang.invoke.MethodHandle，java.lang.invoke.MethodType 和 String 实例的引用。

在解析调用点限定符的方法句柄符号引用，或解析调用点限定符中方法类型的描述符的符号引用时，或是解析任何的静态参数的符号引用时，任何与方法类型或方法句柄解析（§5.4.3.5）有关的异常都可以被抛出。

5.4.3 访问控制

一个类或接口 C 对另外一个类或接口 D 是可见的（Accessible），当且仅当下面的条件之一成立：

- C 是 public 的。
- C 和 D 处于同一个运行包（Runtime Package，§5.3）下面。

一个字段或方法 R 对另外一个类或接口 D 可见的，当且仅当下面的条件之一成立：

- R 是 public 的。
- R 在 C 中是 protected，那么 D 要么与 C 相同，要么就是 C 的子类。如果 R 不是 static 的，那么到 R 的符号引用就必须包含一个到 T 类的符号引用，这里的 T 要么与 D 相同，要么就是 D 的子类或父类。
- R 要么是 protected，要么是默认访问权限（既不是 public，也不是 protected，更不是 private，并且它所属的那个类应该与 D 处于同一运行包下）。
- R 是 private 的，它声明在 D 类中。

上面的这些对权限控制的讨论忽略了一些限制，就是对于 protected 字段访问或方法调用的目标（调用的目标必须是 D 或它的子类型）。这需要在验证部分（§5.4.1）做相应地检查。它不是链接过程中的访问控制所需要涉及的。

5.4.5 方法覆盖

要声明在 C 类中的实例方法 m1 覆盖（Overriding）另外一个声明在 A 类里的实例方法 m2，当且仅当下面的条件都成立才合法：

- C 是 A 的子类。
- m2 与 m1 拥有相同的名称及方法描述符。
- 下面其中一条成立：
  - m2 的权限标志是 ACC_PUBLIC 或 ACC_PROTECTED 或默认权限（既不是 ACC_PUBLIC、也不是 ACC_PROTECTED 更不是 ACC_PRIVATE，且和 C 类处于同一运行包下面）。
  - m1 覆盖方法 m3，m3 与 m1 不同，m3 也与 m2 不同，并且 m3 覆盖了 m2。

5.5 初始化

初始化（Initialization）对于类或接口来说，就是执行它的初始化方法（§2.9）。在发生下列行为时，类或接口将会被初始化：

- 在执行下列需要引用类或接口的 Java 虚拟机指令时：new，getstatic，putstatic 或 invokestatic。这些指令通过字段或方法引用来直接或间接地引用其它类。执行上面所述的 new 指令，在类或接口没有被初始化过时就初始化它。执行上面的 getstatic，putstatic 或 invokestatic 指令时，那些解析好的字段或方法中的类或接口如果还没有被初始化那就初始化它。
- 在初次调用 java.lang.invoke.MethodHandle 实例时，它的执行结果为通过 Java 虚拟机解析出类型是 2（REF_getStatic）、4（REF_putStatic）或者 6（REF_invokeStatic）的方法句柄（§5.4.3.5）。
- 在调用 JDK 核心类库中的反射方法时，例如，Class 类或 java.lang.reflect 包。
- 在对于类的某个子类的初始化时。
- 在它被选定为 Java 虚拟机启动时的初始类（§5.2）时。

在类或接口被初始化之前，它必须被链接过，也就是经过验证、准备阶段，且有可能已经被解析完成了。

因为 Java 虚拟机是支持多线程的，所以在初始化类或接口的时候要特别注意线程同步问题，可能其它一些线程也想要初始化相同名称的类或接口。也有可能在初始化一些类或接口时，初始的请求被递归要求初始化它自己。Java 虚拟机实现需要负责处理好线程同步和递归初始化，具体可以使用下面的步骤来处理。这些处理步骤假定 Class 对象已经被验证和准备过，并且处于下面所述的四种状态之一：

- Class 对象已经被验证和准备过，但还没有被初始化。
- Class 对象正在被其它特定线程初始化。
- Class 对象已经成功被初始化且可以使用。
- Class 对象处于错误的状态，可能因为尝试初始化时失败过

每个类或接口 C，都有一个唯一的初始化锁 LC。如何实现从 C 到 LC 的映射可由 Java 虚拟机实现自行决定。例如，LC 可以是 C 的 Class 对象，或者是与 Class 对象相关的管程（Monitor）。初始化 C 的过程如下：

- 同步 C 的初始化锁 LC。这个操作会导致当前线程一直等待直到可以获得 LC 锁。
- 如果 C 的 Class 对象显示当前 C 的初始化是由其它线程正在进行，那么当前线程释放 LC 并进入阻塞状态，直到它知道初始化工作已经由其它线程完成，那么当前线程在此重试此步骤。
- 如果 C 的 Class 对象显示 C 的初始化正由当前线程在做，这就是对初始化的递归请求。释放 LC 并正常返回。
- 如果 C 的 Class 对象显示 Class 已经被初始化完成，那么什么也不做。释放 LC 并正常返回。
- 如果 C 的 Class 对象显示它处于一个错误的状态，就不再初始化了。释放 LC 并抛出 NoClassDefFoundError 异常。
- 否则，记录下当前线程正在初始化 C 的 Class 对象，随后释放 LC。根据属性出现在 ClassFile 的顺序，利用常量池中的 ConstantValue 属性（§4.7.2）来初始化 C 中的各个 final static 字段。
- 接下来，如果 C 是类而不是接口，而且它的父类 SC 还没有被初始化过，那就对于 SC 也进行完整的初始化过程。当然如果必要的话，需要先验证和准备 SC。如果在初始化 SC 的时候因为抛出异常而中断，那么就获取 LC 后将 C 的 Class 对象标识为错误状态，并通知所有正在等待的线程，最后释放 LC 并异常退出，抛出与 SC 初始化遇到的异常相同的异常。
- 之后，通过查询 C 的定义加载器来决定是否为 C 开启断言机制。
- 执行 C 的类或接口初始化方法。
- 如果正常地执行了类或接口的初始化方法，之后就请求获取 LC，标记 C 的 Class 对象已经被完全初始化，通知所有正在等待的线程，接着释放 LC，正常地退出整个过程。
- 否则，类或接口的初始化方法就必须抛出一个异常 E 并中断退出。如果 E 不是 Error 或它的某个子类，那就创建一个新的 ExceptionInInitializerError 实例，然后将此实例作为 E 的参数，之后的步骤就使用 E 这个对象。如果因为 OutOfMemoryError 问题而不能创建 ExceptionInInitializerError 实例，那在之后就使用 OutOfMemoryError 异常对象作为 E 的参数。
- 获取 LC，标记下 C 的 Class 对象有错误发生，通知所有的等待线程，释放 LC，将 E 或上述的具体错误对象作为此次意外中断的原因。

虚拟机在具体实现时可以通过省略第 1 步在检查类初始化是否完成时的锁获取过程（在第 4、5 步时释放）而获得更好的性能。允许这样做，是因为在 Java 内存模型中的 happens-before

规则（JLS §17.4.5）在锁释放后依然存在，因此可以进行相应的优化。

5.6 绑定本地方法实现

绑定（Binding）是指将使用 Java 之外的语言编写的函数集成到 Java 虚拟机中的过程。此函数需要实现在代码中定义好的 native 方法，之后才可以在 Java 虚拟机中运行。这个过程在传统编译原理的表述中被称“链接”，所以规范里使用“绑定”这个词就，就是为了避免与 Java 虚拟机中链接类或接口的语义发生冲突。

5.7 Java 虚拟机退出

Java 虚拟机的退出条件一般是：某些线程调用 Runtime 类或 System 类的 exit 方法，或是 Runtime 类的 halt 方法，并且 Java 安全管理器也允许这些 exit 或 halt 操作。

除此之外，在 JNI（Java Native Interface）规范中还描述了当使用 JNI API 来加载和卸载（Load & Unload）Java 虚拟机时，Java 虚拟机的退出过程。

第 6 章 Java 虚拟机指令集

一条 Java 虚拟机指令由一个特定操作的操作码和零至多个操作所使用到的操作数所构成。在本章中将会描述每条 Java 虚拟机指令的格式和所代表的操作含义。

6.1 设定：“必须”的含义

我们无法完全孤立地介绍字节码指令作用，对指令的描述应在给定的符合“第 4 章 Class 文件格式”中相关静态和结构化约束的 Java 虚拟机代码上下文中进行。在对某些指令进行讲解时，我们常常会提到“必须”或者“不允许”等词汇，例如“value2 必须是一个 int 类型的数据”。在第 4 章的约束下，所有被“必须”或“不允许”修饰的要求都需要得到满足。如果在某些运行时场景中，这些约束没有得到满足，那 Java 虚拟机在这时的行为是不可预知的，《Java 虚拟机规范》不会去定义违反约束前提下的虚拟机行为。

Java 虚拟机会在链接阶段通过 Class 文件校验器（参见 § 4.10 “Class 文件验证过程”）来检查 Java 虚拟机代码是否满足上述静态的和结构化的约束。因此，Java 虚拟机只会尝试执行一个被检查器确认过的、有效的 Class 文件中的代码。基于减少运行时的实际工作量考虑，检查过程只会执行一次。当然，其他实现策略也是可以的，只要保证这些实现遵循《Java 语言规范》和《Java 虚拟机规范》即可。

6.2 保留操作码

在 Class 文件中使用（参见 § 4.10 Class 文件验证过程）的、本章稍后会逐一讲解的指令操作码中，有三个是保留操作码，它们是被 Java 虚拟机内部使用的。如果 Java 虚拟机指令集在将来被扩充的话，这 3 个保留操作码需要保证不被占用。

其中，操作码值分别为 254（0xfe）和 255（0xff），助记符分别为 impdep1 和 impdep2 的两个操作码是作为“后门”和“陷阱”出现，目的是在某些硬件和软件中提供一些与实现相关的功能。第三个操作码值分别为 202（0xca）、助记符为 breakpoint 的操作码是用于调试器实现断点功能。

这个三个操作码是被保留的，只能用于 Java 虚拟机实现内部，而不能真的出现在一个有效的 Class 文件之中。调试器或者即时代码生成器（§2.13）可以直接与已经加载的或者正在执行中的 Java 虚拟机代码交互，如果遇到这些保留操作码，那调试器或代码生成器应当能对其语义做出正确的处理。

6.3 虚拟机错误

当 Java 虚拟机出现了内部错误，或者由于资源限制导致虚拟机无法实现 Java 语言中的语义时，Java 虚拟机将会抛出一个属于 VirtualMachineError 的子类的异常对象实例。本规范无法预测虚拟机会遇到哪些内部错误或者资源受限的情况，也不可能精确地指出虚拟机何时会发生这些异常情况。因此，下面定义的这些 VirtualMachineError 的子类异常可能会出现在 Java 虚拟机运作过程中的任意时刻：

- InternalError：Java 虚拟机实现的软件或硬件错误都会导致 InternalError 异常的出现，InternalError 是一个典型的异步异常（§2.10），它可能出现在程序中的任何位置。
- OutOfMemoryError：当 Java 虚拟机实现耗尽了所有虚拟和物理内存，并且内存自动管理子系统无法回收到足够共新对象分配所需的内存空间时，虚拟机将抛出 OutOfMemoryError 异常。
- StackOverflowError：当 Java 虚拟机实现耗尽了线程全部的栈空间，这种情况经常是由于程序执行时无限制的递归调用而导致的，虚拟机将会抛出 StackOverflowError 异常。
- UnknownError：当某种异常或错误出现，但虚拟机实现无法确定具体实际是哪种异常或错误的时候，将会抛出 UnknownError 异常。

6.4 指令描述格式

在本章中介绍的 Java 虚拟机指令将会按照字母顺序排序，使用如表 6.1 所示的指令格式表来表示。每一条指令都将采用一页独立页面进行描述。

在表中的“格式”行里，每一个格代表了 8bit 长度，表格名字就是这条指令的助记符，操作码通过数字表示，将同时给出十进制和十六进制的数字表示形式。实际上在 Class 文件里面的 Java 虚拟机代码只会出现数字形式的操作码，不会出现助记符。

请记住，操作数也有可能会在编译期产生，并且嵌入到 Java 虚拟机的字节码指令当中，这些操作数会在运行期参与运算并加载到操作数栈当中。虽然操作数可能会有不同的来源，但是它们都是为了实现同一种目的：作为 Java 虚拟机指令执行时所使用的参数值。操作数隐式地从操作数栈中获取会比显式地生成到编译后的代码中更利于保持 Java 虚拟机字节码的紧凑性。

有部分指令会以一系列描述、格式、操作数栈图等都一致的关联指令族的形式出现，这种指令族会包含若干个操作码和操作码助记符，但只有指令族的助记符会出现在指令格式表中，并且在“结构”行中会列出指令族所包括的所有助记符和操作码。例如 lconst_<l>指令族的“结构”行就会给出这条指令族中 lconst_0 和 lconst_1 两条指令的助记符和操作码信息：

```
结构 lconst_0 = 9 (0x9)
     lconst_1 = 10 (0xa)
```

在 Java 虚拟机指令描述中，指令执行之后对当前栈帧（§2.6）的操作数栈（§2.6.2）产生的影响将会使用文本的方式从左至右分别显示操作数栈中每一个数值的变化，因此，下面“操作数栈”行显示了这条指令执行时会使用到操作数栈栈顶的 value2 以及随后的 value1，执行结果是 value1 和 value2 从操作数栈出栈，并且指令的计算结果值 result 入栈到操作数栈中：

```
操作数栈      …, value1, value2 ➞
              …, result
```

操作数栈中的其余部分使用一组省略号（…）来表示，代表指令执行不会影响到操作数栈的这部分的内容。

long 和 double 类型的数值也只使用一个操作数栈元素来表示[①]。

[①] 在本规范的第一版中，操作数栈内的 long 核 double 数据使用了两个栈元素来表示。

表 6.1 指令格式表

助记符	
操作	该指令功能的简要描述
格式	**助记符** 操作数 1 操作数 2 ……
结构	助记符 = 操作码
操作数栈	…, value1, value2 → …, result
描述	关于操作数栈内容、常量池项、指令操作和结果类型等信息的详细描述。
链接时异常	如果执行该指令可能抛出任何链接时异常，那么每一个可能抛出的异常都需要在此进行描述。
运行时异常	如果执行该指令可能抛出任何运行时异常，那么每一个可能抛出的异常都需要在此进行描述。 除了在此已列出的链接时、运行时异常以及 VirtualMachineError 或其子类之外，指令不得再抛出其他任何异常。
注意	某些并非本规范对该指令强制约束的注释，将会在这里描述。

6.5 指令集描述

aaload	
操作	从数组中加载一个 reference 类型数据到操作数栈
格式	**aaload**
结构	aaload = 50（0x32）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 reference 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 reference 类型值将压入到操作数栈中。
运行时异常	如果 arrayref 为 null，aaload 指令将抛出 NullPointerException 异常。 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，aaload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

aastore	
操作	从操作数栈读取一个 reference 类型数据存入到数组中
格式	**aastore**
结构	aastore = 83（0x53）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 reference 的数组，index 必须为 int 类型，value 必须为 reference 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 存储到 index 作为索引定位到数组元素中。 在运行时，value 的实际类型必须与 arrayref 所代表的数组的组件类型相匹配。具体地说，reference 类型值 value（记作 S）能匹配组件类型为 reference（记作 T）的数组的前提是： ❑ 如果 S 是类类型（Class Type），那么： ■ 如果 T 也是类类型，那 S 必须与 T 是同一个类类型，或者 S 是 T 所代表的类型的子类。 ■ 如果 T 是接口类型，那 S 必须实现了 T 的接口。 ❑ 如果 S 是接口类型（Interface Type），那么： ■ 如果 T 是类类型，那么 T 只能是 Object。 ■ 如果 T 是接口类型，那么 T 与 S 应当是相同的接口，或者 T 是 S 的父接口。 ❑ 如果 S 是数组类型（Array Type），假设为 SC[]的形式，这个数组的组件类型为 SC，那么： ■ 如果 T 是类类型，那么 T 只能是 Object。 ■ 如果 T 是数组类型，假设为 TC[]的形式，这个数组的组件类型为 TC，

	那么下面两条规则之一必须成立： ◆ TC 和 SC 是同一个原始类型。 ◆ TC 和 SC 都是 reference 类型，并且 SC 能与 TC 类型相匹配（以此处描述的规则来判断是否互相匹配）。 ❑ 如果 T 是接口类型，那 T 必须是数组类型所实现的接口之一（JLS3 §4.10.3）。
运行时异常	如果 arrayref 为 null，aastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，aastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。 另外，如果 arrayref 不为 null，并且 value 的实际类型与数组组件类型不能互相匹配（JLS3 §5.2），aastore 指令将抛出 ArrayStoreException 异常。

aconst_null	
操作	将一个 null 值压入到操作数栈中
格式	**aconst_null**
结构	aconst_null = 1（0x1）
操作数栈	… → …, null
描述	将一个 null 值压入到操作数栈中。
注意	《Java 虚拟机规范》并没有强制规定 null 值的在虚拟机的内存中是如何实际表示的。

aload	
操作	从局部变量表加载一个 reference 类型值到操作数栈中
格式	**aload** index
结构	aload = 25（0x19）
操作数栈	… → …, objectref
描述	index 是一个代表当前栈帧（§2.6）中局部变量表的索引的无符号 byte 类型整数，index 作为索引定位的局部变量必须为 reference 类型，称为 objectref。指令执行后，objectref 将会压入到操作数栈栈顶。
注意	aload 指令无法被用于加载类型为 returnAddress 类型的数据到操作数栈中，这点是特意设计成与 astore 指令不相对称的（astore 指令可以操作 returnAddress 类型的数据）。 aload 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表。

aload_<n>	
操作	从局部变量表加载一个 reference 类型值到操作数栈中
格式	**aload_<n>**
结构	aload_0 = 42（0x2a） aload_1 = 43（0x2b） aload_2 = 44（0x2c） aload_3 = 45（0x2d）
操作数栈	… → …, objectref
描述	<n>代表当前栈帧（§2.6）中局部变量表的索引值，<n>作为索引定位的局部变量必须为 reference 类型，称为 objectref。指令执行后，objectref 将会压入到操作数栈栈顶
注意	aload_<n>指令无法被用于加载类型为 returnAddress 类型的数据到操作数栈中，这点是特意设计成与 astore_<n>指令不相对称的（astore_<n>指令可以操作 returnAddress 类型的数据）。 aload_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 aload 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

anewarray	
操作	创建一个组件类型为 reference 类型的数组
格式	**anewarray** indexbyte1 indexbyte2
结构	anewarray = 189（0xbd）
操作数栈	…, count → …, arrayref
描述	count 应为 int 类型的数据，指令执行时它将从操作数栈中出栈，它代表了要创建多大的数组。indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个类、接口或者数组类型的符号引用，这个类、接口或者数组类型应当是已被解析（§5.4.3.1）的。一个以此类型为组件类型、以 count 值为长度的数组将会被分配在 GC 堆中，并且一个代表该数组的 reference 类型数据 arrayref 压入到操作数栈中。这个新数组的所有元素值都被初始化为 null，也即是 reference 类型的默认值。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。
运行时异常	另外，如果 count 值小于 0 的话，anewarray 指令将会抛出一个 NegativeArraySizeException 异常。
注意	anewarray 指令可用于创建一个单维度的数组，或者用于创建一个多维度数组的一部分。

areturn	
操作	结束方法，并返回一个 reference 类型数据
格式	**areturn**
结构	areturn = 176（0xb0）
操作数栈	…, objectref → [empty]
描述	objectref 必须是一个 reference 类型的数据，并且必须指向一个类型与当前方法的方法描述符（§4.3.3）中的返回值相匹配（JLS §5.2）的对象。 如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，那 objectref 将从当前栈帧（§2.6）中出栈，并压入到调用者栈帧的操作数栈中，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在§2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 areturn 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，areturn 指令也会抛出 IllegalMonitorStateException 异常。

arraylength	
操作	取数组长度
格式	**arraylength**
结构	arraylength = 190（0xbe）
操作数栈	…, arrayref → …, length
描述	arrayref 必须是指向数组的 reference 类型的数据，指令执行时，arrayref 从操作数栈中出栈，数组的长度 length 将被计算出来并作为一个 int 类型数据压入到操作数栈中。
运行时异常	如果 arrayref 是 null，arraylength 将会抛出 NullPointerException 异常。

astore	
操作	将一个 reference 类型数据保存到局部变量表中
格式	**astore** index
结构	astore = 58（0x3a）
操作数栈	…，objectref → …
描述	index 是一个无符号 byte 型整数，它必须是一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 objectref 必须是 returnAddress 或者 reference 类型的数据，这个数据将从操作数栈出栈，然后保存到 index 所指向的局部变量表位置中。
注意	astore 指令可以与 returnAddress 类型的数据配合来实现 Java 语言中的 finally 子句（参见§3.13，“编译 fianlly”）。但是 aload 指令**不可以**用来从局部变量表加载 returnAddress 类型的数据到操作数栈，这种 astore 指令的不对称性是有意设计的。 astore 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

astore_<n>	
操作	将一个 reference 类型数据保存到局部变量表中
格式	**astore_<n>**
结构	astore_0 = 75（0x4b） astore_1 = 76（0x4c） astore_2 = 77（0x4d） astore_3 = 78（0x4e）
操作数栈	…, objectref → …
描述	<n>必须是一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 objectref 必须是 returnAddress 或者 reference 类型的数据，这个数据将从操作数栈出栈，然后保存到<n>所指向的局部变量表位置中。
注意	astore_<n>指令可以与 returnAddress 类型的数据配合来实现 Java 语言中的 finally 子句（参见§3.13，“编译 fianlly”）。但是 aload_<n>指令**不可以**用来从局部变量表加载 returnAddress 类型的数据到操作数栈，这种 astore_<n>指令的不对称性是有意设计的。 astore_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 astore 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

athrow	
操作	抛出一个异常实例（exception 或者 error）
格式	**athrow**
结构	athrow = 191（0xbf）
操作数栈	…，objectref → objectref
描述	objectref 必须为一个 reference 类型的数据，它指向一个 Throwable 或其子类的对象实例。在指令执行时，objectref 首先从操作数栈中出栈，然后通过 §2.10 中描述的算法搜索当前方法（§2.6）中与 objectref 的类型相匹配的第一个异常处理器。 如果找到了适合 objectref 的异常处理器，这个异常处理器将包含一个用于处理此异常的代码位置。PC 寄存器的值就会被重设为异常处理器指定的那个位置上，整个当前栈帧的操作数栈都会被清空，objectref 重新压入到操作数栈中，然后程序继续执行。 如果在当前栈帧中没有找到适合的异常处理器，那么栈帧就要从操作数栈中出栈，如果当前栈帧对应的方法是一个同步方法，那在方法调用时持有或重入的管程就应当释放（对于重入来说是计数减 1），就像执行了 monitorexit 一样。最后，这个栈帧的调用者被恢复。如果此栈帧仍然没有找到合适的异常处理器，那它也会继续退出，objectref 也会不断重新抛出，假设已经没有任何的栈帧可以退出，那当前线程将被结束掉。
运行时异常	如果 objectref 为 null，athrow 指令将会抛出 NullPointerException 来代替 objectref 所代表的异常。 另外，如果虚拟机实现没有严格执行在 §2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有或重入

	相应的管程，那 athrow 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，athrow 指令也会抛出 IllegalMonitorStateException 异常。
注意	athrow 指令的操作数栈图（本表中“操作数栈”行的图）可能会产生一些误解：如果当前方法中某个异常处理器被匹配到，athrow 指令将抛弃掉操作数栈上所有的值，然后重新将被抛出的异常对象入栈，但是如果在当前方法中没有找到适合的异常处理器，即异常被抛到方法调用链其他地方时，被清空的和 objectref 入栈的操作数栈是真正处理异常的那个方法的操作数栈，而从最初抛出异常的那个方法一直到最终处理异常的那个方法（不含）之间的栈帧全部都会被丢弃掉。

baload	
操作	从数组中读取 byte 或者 boolean 类型的数据
格式	**baload**
结构	baload = 51（0x33）
操作数栈	…，arrayref，index → …，value
描述	arrayref 是一个 reference 类型的数据，它指向一个以 byte 或者 boolean 为组件类型的数组对象，index 是一个 int 型的数据。在指令执行时，arrayref 和 index 都从操作数栈中出栈，在数组中使用 index 为索引定位到的 byte 类型数据被带符号扩展（Sign-Extended）为一个 int 型数据并压入到操作数栈中。
运行时异常	如果 arrayref 为 null，baload 指令将抛出 NullPointerException 异常。 另外，如果 index 不在数组的上下界范围之内，baload 指令将抛出 ArrayIndexOutOfBoundsException 异常。
注意	baload 指令可以用来从数组中读取 byte 或者 boolean 的数据，在 Oracle 的虚拟机实现中，布尔类型的数组（T_BOOLEAN 类型的数组可参见 § 2.2 和本章中对 newarray 指令的介绍）被实现为 8 位宽度的数值，而其他的虚拟机实现很可能使用其他方式实现 boolean 数组，那其他虚拟机实现的 baload 就必须能正确访问相应实现的数组。

bastore	
操作	从操作数栈读取一个 byte 或 boolean 类型数据存入到数组中
格式	**bastore**
结构	bastore = 84（0x54）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 byte 或 boolean 的数组，index 和 value 都必须为 int 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 将被转换为 byte 类型，然后存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，bastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，bastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。
注意	bastore 指令可以用来保存 byte 或者 boolean 的数据到数组之中，在 Oracle 的虚拟机实现中，布尔类型的数组（T_BOOLEAN 类型的数组可参见 §2.2 和本章中对 newarray 指令的介绍）被实现为 8 位宽度的数值，而其他的虚拟机实现很可能使用其他方式实现 boolean 数组，那其他虚拟机实现的 bastore 就必须能正确访问相应实现的数组。

bipush	
操作	将一个 byte 类型数据入栈
格式	**bipush** **byte**
结构	bipush = 16（0x10）
操作数栈	… → …, value
描述	将 byte 带符号扩展为一个 int 类型的值 value，然后将 value 压入到操作数栈中。

caload	
操作	从数组中加载一个 char 类型数据到操作数栈
格式	**caload**
结构	caload = 52（0x34）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 char 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 char 类型值先被零位扩展（Zero-Extended）为一个 int 类型数据 value，然后再将 value 压入到操作数栈中。
运行时异常	如果 arrayref 为 null，caload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，caload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

castore	
操作	从操作数栈读取一个 char 类型数据存入到数组中
格式	**castore**
结构	castore = 85（0x55）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 char 的数组，index 和 value 都必须为 int 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 将被转换为 char 类型，然后存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，castore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，castore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

checkcast

操作	检查对象是否符合给定的类型
格式	**checkcast** **indexbyte1** **Indexbyte2**
结构	checkcast = 192（0xc0）
操作数栈	…，objectref → …，objectref
描述	objectref 必须为 reference 类型的数据，indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个类、接口或者数组类型的符号引用。 如果 objectref 为 null 的话，那操作数栈不会有任何变化。 否则，参数指定的类、接口或者数组类型会被虚拟机解析（§5.4.3.1）。如果 objectref 可以转换为这个类、接口或者数组类型，那操作数栈就保持不变，否则 checkcast 指令将抛出一个 ClassCastException 异常。 以下规则可以用来确定一个非空的 objectref 是否可以转换为指定的已解析类型：假设 S 是 objectref 所指向的对象的类型，T 是进行比较的已解析的类、接口或者数组类型，checkcast 指令根据这些规则来判断转换是否成立： ❑ 如果 S 是类类型（Class Type），那么： ■ 如果 T 也是类类型，那 S 必须与 T 是同一个类类型，或者 S 是 T 所代表的类型的子类。 ■ 如果 T 是接口类型，那 S 必须实现了 T 的接口。 ❑ 如果 S 是接口类型（Interface Type），那么： ■ 如果 T 是类类型，那么 T 只能是 Object。

	■ 如果 T 是接口类型，那么 T 与 S 应当是相同的接口，或者 T 是 S 的父接口。 ❑ 如果 S 是数组类型（Array Type），假设为 SC[]的形式，这个数组的组件类型为 SC，那么： ■ 如果 T 是类类型，那么 T 只能是 Object。 ■ 如果 T 是数组类型，假设为 TC[]的形式，这个数组的组件类型为 TC，那么下面两条规则之一必须成立： ◆ TC 和 SC 是同一个原始类型。 ◆ TC 和 SC 都是 reference 类型，并且 SC 能与 TC 类型相匹配（以此处描述的规则来判断是否互相匹配）。 如果 T 是接口类型，那 T 必须是数组所实现的接口之一（JLS3 §4.10.3）。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。
运行时异常	如果 objectref 不能转换成参数指定的类、接口或者数组类型，checkcast 指令将抛出 ClassCastException 异常
注意	checkcast 指令与 instanceof 指令非常类似，它们之间的区别是如何处理 null 值的情况、测试类型转换的结果反馈方式（checkcast 是抛异常，而 instanceof 是返回一个比较结果）以及指令执行后对操作数栈的影响。

d2f	
操作	将 double 类型数据转换为 float 类型
格式	**d2f**
结构	d2f = 144（0x90）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 double 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再通过 IEEE 754 的向最接近数舍入模式（§2.8.1）转换为 float 类型值 result。然后 result 被压入到操作数栈中。 如果 d2f 指令运行在 FP-strict（§2.8.2）模式下，那转换的结果永远是转换为单精度浮点值集合中与原值最接近的可表示值。 如果 d2f 指令运行在非 FP-strict 模式下，那转换结果可能会从单精度扩展指数集合（§2.3.2）中选取，也就是说并非一定会转换为单精度浮点值集合中与原值最接近的可表示值的。 当有限值 value’太小以至于无法使用 float 类型数据来表示时，将会被转换为与原值符号相同的零值。同样，当有限值 value’太大以至于无法使用 float 类型数据来表示时，将会被转换为与原值符号相同的无穷大。double 类型的 NaN 值永远转换为 float 类型的 NaN 值。
注意	d2f 指令执行了窄化类型转换(Narrowing Primitive Conversion，JLS3 §5.1.3)，它可能会导致 value’的数值大小和精度发生丢失。

d2i	
操作	将 double 类型数据转换为 int 类型
格式	**d2i**
结构	d2i = 142（0x8e）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 double 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再转换为 int 类型值 result。然后 result 被压入到操作数栈中。 ❑ 如果 value’是 NaN 值，那 result 的转换结果为 int 类型的零值。 ❑ 另外，如果 value’不是无穷大，那将会使用 IEEE 754 标准中的向零舍入模式（§2.8.1）转换成整数值 V，如果这个整数 V 在 int 类型的可表示范围之内，那么 result 的转换结果就是这个整数 V。 ❑ 另外，如果 value’太小（绝对值很大的负数或者负无穷大）以至于超过了 int 类型可表示的下限，那将转换为 int 类型中最小的可表示数。同样地，如果 value’太大（很大的正数或者无穷大）以至于超过了 int 类型可表示的上限，那将转换为 int 类型中最大的可表示数。
注意	d2i 指令执行了窄化类型转换(Narrowing Primitive Conversion，JLS3 §5.1.3)，它可能会导致 value’的数值大小和精度发生丢失。

d2l	
操作	将 double 类型数据转换为 long 类型
格式	**d2l**
结构	d2l = 143（0x8f）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 double 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再转换为 long 类型值 result。然后 result 被压入到操作数栈中。 ❑ 如果 value’是 NaN 值，那 result 的转换结果为 long 类型的零值。 ❑ 另外，如果 value’不是无穷大，那将会使用 IEEE 754 标准中的向零舍入模式（§2.8.1）转换成整数值 V，如果这个整数 V 在 long 类型的可表示范围之内，那么 result 的转换结果就是这个整数 V。 ❑ 另外，如果 value’太小（绝对值很大的负数或者负无穷大）以至于超过了 long 类型可表示的下限，那将转换为 long 类型中最小的可表示数。同样地，如果 value’太大（很大的正数或者无穷大）以至于超过了 long 类型可表示的上限，那将转换为 long 类型中最大的可表示数。
注意	d2l 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value’的数值大小和精度发生丢失。

dadd	
操作	double 类型数据相加
格式	**dadd**
结构	dadd = 99（0x63）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相加，结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 dadd 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 两个不同符号的无穷大相加，结果为 NaN。 ❑ 两个相同符号的无穷大相加，结果仍然为相同符号的无穷大。 ❑ 一个无穷大的数与一个有限的数相加，结果为无穷大。 ❑ 两个不同符号的零值相加，结果为正零。 ❑ 两个相同符号的零值相加，结果仍然为相同符号的零值。 ❑ 零值与一个非零有限值相加，结果等于那个非零有限值。 ❑ 两个绝对值相等、符号相反的非零有限值相加，结果为正零。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零、NaN 以及两个值具有相同符号或者不同的绝对值，就按算术求和，并以最接近数舍入模式得到运算结果。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢(Gradual Underflow)，尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 dadd 指令永远不会抛出任何运行时异常。

daload	
操作	从数组中加载一个 double 类型数据到操作数栈
格式	**daload**
结构	daload = 49（0x31）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 double 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 double 类型值将压入到操作数栈中。
运行时异常	如果 arrayref 为 null，daload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，daload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

dastore	
操作	从操作数栈读取一个 double 类型数据存入到数组中
格式	**dastore**
结构	dastore = 82（0x52）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 double 的数组，index 必须为 int 类型，value 必须为 doube 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 并且经过数值集合转换（§2.8.3）后得到值 value’，然后存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，dastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，dastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

dcmp<op>	
操作	比较 2 个 double 类型数据的大小
格式	**dcmp<op>**
结构	dcmpg = 152（0x98） dcmpl = 151（0x97）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着对这 2 个值进行浮点比较操作： ❑ 如果 value1’大于 value2’的话，int 值 1 将压入到操作数栈中。 ❑ 另外，如果 value1’与 value2’相等的话，int 值 0 将压入到操作数栈中。 ❑ 另外，如果 value1’小于 value2’相等的话，int 值－1 将压入到操作数栈中。 ❑ 另外，如果 value1’和 value2’之中最少有一个为 NaN，那 dcmpg 指令将 int 值 1 压入到操作数栈中，而 dcmpl 指令则把 int 值－1 压入到操作数栈中。 浮点比较操作将根据 IEEE 754 规范定义进行，除了 NaN 之外的所有数值都是有序的，正无穷大大于所有有限值，正数零和负数零则被看作是相等的。
注意	dcmpg 和 dcmpl 指令之间的差别仅仅是当比价参数中出现 NaN 值时的处理方式而已。NaN 值是没有顺序的，因此只要参数中出现一个或者两个都为 NaN 值时，比较操作就会失败。但对于 dcmpg 和 dcmpl 指令来说，无论何种情况导致比较失败，都会有一个确定的返回值压入到操作数栈中。读者可以参见 §

	3.5“更多控制例子”获取更多的信息。

dconst_<d>	
操作	将 double 类型数据压入到操作数栈中
格式	**dconst_<d>**
结构	dconst_0 = 14（0xe） dconst_1 = 15（0xf）
操作数栈	… → …, <d>
描述	将 double 类型的常量<d>（0.0 或 1.0）压入到操作数栈中。

ddiv	
操作	double 类型数据除法
格式	**ddiv**
结构	ddiv = 111（0x6f）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相除（value1’÷value2’），结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 ddiv 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那当两者符号相同时，运算结果为正，反着，当两者符号不同时，运算结果为负。 ❑ 两个无穷大相除，运算结果为 NaN。 ❑ 一个无穷大的数与一个有限的数相除，结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 一个有限的数与一个无穷大的数相除，结果为零，零值的符号由第 2 点规则确定。 ❑ 零除以零结果为 NaN，零除以任意其他非零有限值结果为零，零值的符号由第 2 点规则确定。 ❑ 任意非零有限值除以零结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零以及 NaN，就按算术求商，并以 IEEE 754 规范的最接近数舍入模式得到运算结果，如果运算结果的绝对值太大以至于无法使用 double 类型来表示，换句话

	说就是出现了上限溢出，那结果将会使用具有适当符号的无穷大来代替。 如果运算结果的绝对值太小以至于无法使用 double 类型来表示，换句话说就是出现了下限溢出，那结果将会使用具有适当符号的零值来代替。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢(Gradual Underflow)，尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 ddiv 指令永远不会抛出任何运行时异常。

dload	
操作	从局部变量表加载一个 double 类型值到操作数栈中
格式	**dload** index
结构	dload = 24（0x18）
操作数栈	… → …, value
描述	index 是一个代表当前栈帧（§2.6）中局部变量表的索引的无符号 byte 类型整数，index 作为索引定位的局部变量必须为 double 类型（占用 index 和 index＋1 两个位置），记为 value。指令执行后，value 将会压入到操作数栈栈顶
注意	dload 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表。

dload_<n>	
操作	从局部变量表加载一个 double 类型值到操作数栈中
格式	**dload_<n>**
结构	dload_0 = 38（0x26） dload_1 = 39（0x27） dload_2 = 40（0x28） dload_3 = 41（0x29）
操作数栈	… → …, value
描述	<n>和<n>＋1 共同代表一个当前栈帧（§2.6）中局部变量表的索引值，<n>作为索引定位的局部变量必须为 double 类型，记作 value。指令执行后，value 将会压入到操作数栈栈顶
注意	dload_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 dload 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

dmul	
操作	double 类型数据乘法
格式	**dmul**
结构	dmul = 107（0x6b）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相乘（value1’×value2’），结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 dmul 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那当两者符号相同时，运算结果为正，反着，当两者符号不同时，运算结果为负。 ❑ 无穷大与零值相乘，运算结果为 NaN。 ❑ 一个无穷大的数与一个有限的数相乘，结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大或者 NaN，就按算术求积，并以 IEEE 754 规范的最接近数舍入模式得到运算结果，如果运算结果的绝对值太大以至于无法使用 double 类型来表示，换句话说就是出现了上限溢出，那结果将会使用具有适当符号的无穷大来代替。如果运算结果的绝对值太小以至于无法使用 double 类型来表示，换句话说就是出现了下限溢出，那结果将会使用具有适当符号的零值来代替。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢（Gradual Underflow），尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 dmul

	指令永远不会抛出任何运行时异常。

dneg	
操作	double 类型数据取负运算
格式	**dneg**
结构	dneg = 119（0x77）
操作数栈	…, value → …, result
描述	value 必须为 double 类型数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value'，接着对这个数进行算术取负运算，结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 对于 double 类型数据，取负运算并不等同于与零做减法运算。如果 x 是＋0.0，那么 0.0－x 等于＋0.0，但是－x 则等于－0.0，后面这种一元减法运算仅仅把数值的符号反转。 下面是一些值得注意的场景： ❑ 如果操作数为 NaN，那运算结果也为 NaN（NaN 值是没有符号的）。 ❑ 如果操作数是无穷大，那运算结果是与其符号相反的无穷大。 ❑ 如果操作数是零，那运算结果是与其符号相反的零值。

drem

操作	double 类型数据求余
格式	**drem**
结构	drem = 115（0x73）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值求余，结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 drem 指令的运算结果与 IEEE 754 中定义的 remainder 操作并不相同，IEEE 754 中的 remainder 操作使用舍入除法（Rounding Division）而不是去尾触发（Truncating Division）来获得求余结果，因此这种运算与通常对整数的求余方式并不一致。Java 虚拟机中定义的 drem 则是与虚拟机中整数求余指令（irem 和 lrem）保持了一致的行为，这可以与 C 语言中的 fmod 函数互相比较。 drem 指令的运算结果通过以下规则获得： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那运算结果的符号被除数的符号一致。 ❑ 如果被除数是无穷大，或者除数为零，那运算结果为 NaN。 ❑ 如果被除数是有限值，而除数是无穷大，那运算结果等于被除数。 ❑ 如果被除数为零，而除数是有限值，那运算结果等于被除数。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零以及 NaN，就以 value1’为被除数、value2’为除数使用浮点算术规则求余：

	result = value1’ − (value2’ * q)，这里的 q 是一个整数，其符号与 value1’ ÷value2’的符号相同，大小与他们的商相同。 尽管除数为零的情况可能发生，但是 drem 指令永远不会抛出任何运行时异常，上限溢出、下限溢出和进度丢失的情况也不会出现。 IEEE 754 规范中定义的 remainder 操作可以使用 Math.IEEEremainder 来完成。

dreturn	
操作	结束方法，并返回一个 double 类型数据
格式	**dreturn**
结构	dreturn = 175（0xaf）
操作数栈	…, value → [empty]
描述	当前方法的返回值必须为 double 类型，value 也必须是一个 double 类型的数据。如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，那 value 将从当前栈帧（§2.6）中出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后压入到调用者栈帧的操作数栈中，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在§2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 dreturn 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，dreturn 指令也会抛出 IllegalMonitorStateException 异常。

dstore	
操作	将一个 double 类型数据保存到局部变量表中
格式	**dstore** index
结构	dstore = 57（0x39）
操作数栈	…，value → …
描述	index 是一个无符号 byte 型整数，它和 index＋1 共同一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 double 类型的数据，这个数据将从操作数栈出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后保存到 index 和 index＋1 所指向的局部变量表位置中。
注意	dstore 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

dstore_<n>	
操作	将一个 double 类型数据保存到局部变量表中
格式	**dstore_<n>**
结构	dstore_0 = 71（0x47） dstore_1 = 72（0x48） dstore_2 = 73（0x49） dstore_3 = 74（0x4a）
操作数栈	…, value → …
描述	<n>和<n>＋1 必须是一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 double 类型的数据，这个数据将从操作数栈出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后保存到<n>和<n>＋1 所指向的局部变量表位置中。
注意	dstore_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 dstore 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

dsub	
操作	double 类型数据相减
格式	**dsub**
结构	dsub = 103（0x67）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 double 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相减（result = value1’－value2’），结果转换为 double 类型值 result，最后 result 被压入到操作数栈中。 对于一般 double 类型数据的减法来说，a－b 与 a＋（－b）的结果永远是一致的，但是对于 dsub 指令来说，与零相减的符号则会相反，因为如果 x 是＋0.0 的话，那 0.0－x 等于＋0.0，但－x 等于－0.0。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢（Gradual Underflow），尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 dsub 指令永远不会抛出任何运行时异常。

dup	
操作	复制操作数栈栈顶的值，并插入到栈顶
格式	**dup**
结构	dup = 89（0x59）
操作数栈	…, value → …, value, value
描述	复制操作数栈栈顶的值，并将此值压入到操作数栈顶。 如果 value 不是§2.11.1 的表 2.3 中列出的分类一中的数据类型，就不能使用 dup 指令来复制栈顶值。

dup_x1	
操作	复制操作数栈栈顶的值，并插入到栈顶以下 2 个值之后
格式	**dup_x1**
结构	dup_x1 = 90（0x5a）
操作数栈	…，value2，value1 → …，value1，value2，value1
描述	复制操作数栈栈顶的值，并将此值压入到操作数栈顶以下 2 个值之后。 如果 value1 和 value2 不是§2.11.1 的表 2.3 中列出的分类一中的数据类型，就不能使用 dup_x1 指令来复制栈顶值。

dup_x2	
操作	复制操作数栈栈顶的值，并插入到栈顶以下 2 个或 3 个值之后
格式	**dup_x2**
结构	dup_x2 = 91（0x5b）
操作数栈	结构 1： …, value3, value2, value1 → …, value1, value3, value2, value1 当 value1、value2 和 value3 都是 § 2.11.1 的表 2.3 中列出的分类一中的数据类型时满足结构 1。 结构 2： …, value2, value1 → …, value1, value2, value1 当 value1 是 § 2.11.1 的表 2.3 中列出的分类一中的数据类型，而 value2 是分类二的数据类型时满足结构 2。
描述	复制操作数栈栈顶的值，并将此值压入到操作数栈顶以下 2 个或 3 个值之后。

dup2	
操作	复制操作数栈栈顶 1 个或 2 个值，并插入到栈顶
格式	**dup2**
结构	dup2 = 92（0x5c）
操作数栈	结构 1： …，value2，value1 → …，value2，value1，value2，value1 当 value1 和 value2 都是 § 2.11.1 的表 2.3 中列出的分类一中的数据类型时满足结构 1。 结构 2： …，value → …，value，value 当 value 是 § 2.11.1 的表 2.3 中列出的分类二中的数据类型时满足结构 2。
描述	复制操作数栈栈顶 1 个或 2 个值，并将这些值按照原来的顺序压入到操作数栈顶。

dup2_x1	
操作	复制操作数栈栈顶 1 个或 2 个值，并插入到栈顶以下 2 个或 3 个值之后
格式	**dup2_x1**
结构	dup2_x1 = 93（0x5d）
操作数栈	结构 1： …，value3，value2，value1 → …，value2，value1，value3，value2，value1 当 value1、value2 和 value3 都是§2.11.1 的表 2.3 中列出的分类一中的数据类型时满足结构 1。 结构 2： …，value2，value1 → …，value1，value2，value1 当 value1 是§2.11.1 的表 2.3 中列出的分类二中的数据类型，而 value2 是分类一的数据类型时满足结构 2。
描述	复制操作数栈栈顶 1 个或 2 个值，并按照原有的顺序插入到栈顶以下 2 个或 3 个值之后。

dup2_x2	
操作	复制操作数栈栈顶 1 个或 2 个值，并插入到栈顶以下 2 个、3 个或者 3 个值之后
格式	**dup2_x2**
结构	dup2_x2 = 94（0x5e）
操作数栈	结构 1： …，value4，value3，value2，value1 → …，value2，value1，value4，value3，value2，value1 当 value1、value2 、value3 和 value4 全部都是§2.11.1 的表 2.3 中列出的分类一中的数据类型时满足结构 1。 结构 2： …，value3，value2，value1 → …，value1，value3，value2，value1 当 value1 是§2.11.1 的表 2.3 中列出的分类二中的数据类型，而 value2 和 value3 是分类一的数据类型时满足结构 2。 结构 3： …，value3，value2，value1 → …，value2，value1，value3，value2，value1 当 value1 和 value2 是§2.11.1 的表 2.3 中列出的分类一中的数据类型，而 value3 是分类二的数据类型时满足结构 3。 结构 4： …，value2，value1 → …，value1，value2，value1 当 value1 和 value2 是§2.11.1 的表 2.3 中列出的分类二中的数据类型时满足结构 4。

描述	复制操作数栈栈顶 1 个或 2 个值，并按照原来的顺序插入到栈顶以下 2 个、3 个或者 3 个值之后。

f2d	
操作	将 float 类型数据转换为 double 类型
格式	**f2d**
结构	f2d = 141（0x8d）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 float 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再通过 IEEE 754 的向最接近数舍入模式（§2.8.1）转换为 double 类型值 result。然后 result 被压入到操作数栈中。 如果 d2f 指令运行在 FP-strict（§2.8.2）模式下，那指令执行过程就是一种宽化类型转换（Widening Primitive Conversion，JLS3 §5.1.2）。因为所有单精度浮点数集合（§2.3.2）都可以在双精度浮点数集合（§2.3.2）中找到精确对应的数值，因此这种转换是精确的。 如果 d2f 指令运行在非 FP-strict 模式下，那转换结果就可能会从双精度扩展指数集合（§2.3.2）中选取，并不需要舍入为双精度浮点数集合中最接近的可表示值。不过，如果操作数 value 是单精度扩展指数集合中的数值，那把结果舍入为双精度浮点数集合则是必须的。

f2i	
操作	将 float 类型数据转换为 int 类型
格式	**f2i**
结构	f2i = 139（0x8b）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 float 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再转换为 int 类型值 result。然后 result 被压入到操作数栈中。 ❑ 如果 value’是 NaN 值，那 result 的转换结果为 int 类型的零值。 ❑ 另外，如果 value’不是无穷大，那将会使用 IEEE 754 标准中的向零舍入模式（§2.8.1）转换成整数值 V，如果这个整数 V 在 int 类型的可表示范围之内，那么 result 的转换结果就是这个整数 V。 ❑ 另外，如果 value’太小（绝对值很大的负数或者负无穷大）以至于超过了 int 类型可表示的下限，那将转换为 int 类型中最小的可表示数。同样地，如果 value’太大（很大的正数或者无穷大）以至于超过了 int 类型可表示的上限，那将转换为 int 类型中最大的可表示数。
注意	f2i 指令执行了窄化类型转换(Narrowing Primitive Conversion，JLS3 §5.1.3)，它可能会导致 value’的数值大小和精度发生丢失。

f2l	
操作	将 float 类型数据转换为 long 类型
格式	**f2l**
结构	f2l = 140（0x8c）
操作数栈	…, value → …, result
描述	在操作数栈栈顶的值 value 必须为 float 类型的数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value’，value’再转换为 long 类型值 result。然后 result 被压入到操作数栈中。 ❑ 如果 value’是 NaN 值，那 result 的转换结果为 long 类型的零值。 ❑ 另外，如果 value’不是无穷大，那将会使用 IEEE 754 标准中的向零舍入模式（§2.8.1）转换成整数值 V，如果这个整数 V 在 long 类型的可表示范围之内，那么 result 的转换结果就是这个整数 V。 ❑ 另外，如果 value’太小（绝对值很大的负数或者负无穷大）以至于超过了 long 类型可表示的下限，那将转换为 long 类型中最小的可表示数。同样地，如果 value’太大（很大的正数或者无穷大）以至于超过了 long 类型可表示的上限，那将转换为 long 类型中最大的可表示数。
注意	f2l 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value’的数值大小和精度发生丢失。

fadd

操作	float 类型数据相加
格式	**fadd**
结构	dadd = 99（0x63）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相加，结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 fadd 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 两个不同符号的无穷大相加，结果为 NaN。 ❑ 两个相同符号的无穷大相加，结果仍然为相同符号的无穷大。 ❑ 一个无穷大的数与一个有限的数相加，结果为无穷大。 ❑ 两个不同符号的零值相加，结果为正零。 ❑ 两个相同符号的零值相加，结果仍然为相同符号的零值。 ❑ 零值与一个非零有限值相加，结果等于那个非零有限值。 ❑ 两个绝对值相等、符号相反的非零有限值相加，结果为正零。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零、NaN 以及两个值具有相同符号或者不同的绝对值，就按算术求和，并以最接近数舍入模式得到运算结果。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢(Gradual Underflow)，尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 fadd 指令永远不会抛出任何运行时异常。

faload	
操作	从数组中加载一个 float 类型数据到操作数栈
格式	**faload**
结构	faload = 48（0x30）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 float 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 float 类型值将压入到操作数栈中。
运行时异常	如果 arrayref 为 null，faload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，faload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

fastore	
操作	从操作数栈读取一个 float 类型数据存入到数组中
格式	**fastore**
结构	fastore = 81（0x51）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 float 的数组，index 必须为 int 类型，value 必须为 float 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 并且经过数值集合转换（§2.8.3）后得到值 value’，然后存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，fastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，fastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

fcmp<op>	
操作	比较 2 个 float 类型数据的大小
格式	**fcmp<op>**
结构	fcmpg = 150（0x96） fcmpl = 149（0x95）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着对这 2 个值进行浮点比较操作： ❑ 如果 value1’大于 value2’的话，int 值 1 将压入到操作数栈中。 ❑ 另外，如果 value1’与 value2’相等的话，int 值 0 将压入到操作数栈中。 ❑ 另外，如果 value1’小于 value2’相等的话，int 值－1 将压入到操作数栈中。 ❑ 另外，如果 value1’和 value2’之中最少有一个为 NaN，那 fcmpg 指令将 int 值 1 压入到操作数栈中，而 fcmpl 指令则把 int 值－1 压入到操作数栈中。 浮点比较操作将根据 IEEE 754 规范定义进行，除了 NaN 之外的所有数值都是有序的，正无穷大大于所有有限值，正数零和负数零则被看作是相等的。
注意	fcmpg 和 fcmpl 指令之间的差别仅仅是当比价参数中出现 NaN 值时的处理方式而已。NaN 值是没有顺序的，因此只要参数中出现一个或者两个都为 NaN 值时，比较操作就会失败。但对于 fcmpg 和 fcmpl 指令来说，无论何种情况导致比较失败，都会有一个确定的返回值压入到操作数栈中。读者可以参见 §

	3.5“更多控制例子”获取更多的信息。

fconst_<f>	
操作	将 float 类型数据压入到操作数栈中
格式	**fconst_<f>**
结构	fconst_0 = 11（0xb） fconst_1 = 12（0xc） fconst_2 = 13（0xd）
操作数栈	… → …, <f>
描述	将 float 类型的常量<f>（0.0、1.0 或者 2.0）压入到操作数栈中。

fdiv	
操作	float 类型数据除法
格式	**fdiv**
结构	fdiv = 110（0x6e）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相除（value1’÷value2’），结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 fdiv 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那当两者符号相同时，运算结果为正，反着，当两者符号不同时，运算结果为负。 ❑ 两个无穷大相除，运算结果为 NaN。 ❑ 一个无穷大的数与一个有限的数相除，结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 一个有限的数与一个无穷大的数相除，结果为零，零值的符号由第 2 点规则确定。 ❑ 零除以零结果为 NaN，零除以任意其他非零有限值结果为零，零值的符号由第 2 点规则确定。 ❑ 任意非零有限值除以零结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零以及 NaN，就按算术求商，并以 IEEE 754 规范的最接近数舍入模式得到运算结果，如果运算结果的绝对值太大以至于无法使用 float 类型来表示，换句话

	说就是出现了上限溢出，那结果将会使用具有适当符号的无穷大来代替。 如果运算结果的绝对值太小以至于无法使用 float 类型来表示，换句话说就是出现了下限溢出，那结果将会使用具有适当符号的零值来代替。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢(Gradual Underflow)，尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 fdiv 指令永远不会抛出任何运行时异常。

fload	
操作	从局部变量表加载一个 float 类型值到操作数栈中
格式	**fload** index
结构	fload = 23（0x17）
操作数栈	… → …, value
描述	index 是一个代表当前栈帧（§2.6）中局部变量表的索引的无符号 byte 类型整数，index 作为索引定位的局部变量必须为 float 类型，记为 value。指令执行后，value 将会压入到操作数栈栈顶
注意	fload 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表。

fload_<n>	
操作	从局部变量表加载一个 float 类型值到操作数栈中
格式	**fload_<n>**
结构	fload_0 = 34（0x22） fload_1 = 35（0x23） fload_2 = 36（0x24） fload_3 = 37（0x25）
操作数栈	… → …, value
描述	<n>代表一个当前栈帧（§2.6）中局部变量表的索引值，<n>作为索引定位的局部变量必须为 float 类型，记作 value。指令执行后，value 将会压入到操作数栈栈顶
注意	fload_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 fload 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

fmul	
操作	float 类型数据乘法
格式	**fmul**
结构	fmul = 106（0x6a）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相乘（value1’×value2’），结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 fmul 指令的运算结果取决于 IEEE 规范中规定的运算规则： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那当两者符号相同时，运算结果为正，反着，当两者符号不同时，运算结果为负。 ❑ 无穷大与零值相乘，运算结果为 NaN。 ❑ 一个无穷大的数与一个有限的数相乘，结果为无穷大，无穷大的符号由第 2 点规则确定。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大或者 NaN，就按算术求积，并以 IEEE 754 规范的最接近数舍入模式得到运算结果，如果运算结果的绝对值太大以至于无法使用 float 类型来表示，换句话说就是出现了上限溢出，那结果将会使用具有适当符号的无穷大来代替。如果运算结果的绝对值太小以至于无法使用 float 类型来表示，换句话说就是出现了下限溢出，那结果将会使用具有适当符号的零值来代替。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢（Gradual Underflow），尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 fmul

	指令永远不会抛出任何运行时异常。

fneg	
操作	float 类型数据取负运算
格式	**fneg**
结构	fneg = 118（0x76）
操作数栈	…, value → …, result
描述	value 必须为 float 类型数据，指令执行时，value 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value'，接着对这个数进行算术取负运算，结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 对于 float 类型数据，取负运算并不等同于与零做减法运算。如果 x 是＋0.0，那么 0.0－x 等于＋0.0，但是－x 则等于－0.0，后面这种一元减法运算仅仅把数值的符号反转。 下面是一些值得注意的场景： ❑ 如果操作数为 NaN，那运算结果也为 NaN（NaN 值是没有符号的）。 ❑ 如果操作数是无穷大，那运算结果是与其符号相反的无穷大。 ❑ 如果操作数是零，那运算结果是与其符号相反的零值。

frem	
操作	float 类型数据求余
格式	**frem**
结构	frem = 114（0x72）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值求余，结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 frem 指令的运算结果与 IEEE 754 中定义的 remainder 操作并不相同，IEEE 754 中的 remainder 操作使用舍入除法（Rounding Division）而不是去尾触发（Truncating Division）来获得求余结果，因此这种运算与通常对整数的求余方式并不一致。Java 虚拟机中定义的 drem 则是与虚拟机中整数求余指令（irem 和 lrem）保持了一致的行为，这可以与 C 语言中的 fmod 函数互相比较。 drem 指令的运算结果通过以下规则获得： ❑ 如果 value1’和 value2’中有任意一个值为 NaN，那运算结果即为 NaN。 ❑ 如果 value1’和 value2’两者都不为 NaN，那运算结果的符号被除数的符号一致。 ❑ 如果被除数是无穷大，或者除数为零，那运算结果为 NaN。 ❑ 如果被除数是有限值，而除数是无穷大，那运算结果等于被除数。 ❑ 如果被除数为零，而除数是有限值，那运算结果等于被除数。 ❑ 对于上述情况之外的场景，即任意一个操作数都不是无穷大、零以及 NaN，就以 value1’为被除数、value2’为除数使用浮点算术规则求余：

	result = value1’ − (value2’ * q)，这里的 q 是一个整数，其符号与 value1’ ÷value2’的符号相同，大小与他们的商相同。 尽管除数为零的情况可能发生，但是 frem 指令永远不会抛出任何运行时异常，上限溢出、下限溢出和进度丢失的情况也不会出现。 IEEE 754 规范中定义的 remainder 操作可以使用 Math.IEEEremainder 来完成。

freturn	
操作	结束方法，并返回一个 float 类型数据
格式	**freturn**
结构	freturn = 174（0xae）
操作数栈	…, value → [empty]
描述	当前方法的返回值必须为 float 类型，value 也必须是一个 float 类型的数据。如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，那 value 将从当前栈帧（§2.6）中出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后压入到调用者栈帧的操作数栈中，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在§2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 freturn 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，freturn 指令也会抛出 IllegalMonitorStateException 异常。

fstore	
操作	将一个 float 类型数据保存到局部变量表中
格式	**fstore** index
结构	fstore = 56（0x38）
操作数栈	…, value → …
描述	index 是一个无符号 byte 型整数，它指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 float 类型的数据，这个数据将从操作数栈出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后保存到 index 所指向的局部变量表位置中。
注意	fstore 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

fstore_<n>	
操作	将一个 float 类型数据保存到局部变量表中
格式	**fstore_<n>**
结构	fstore_0 = 67（0x43） fstore_1 = 68（0x44） fstore_2 = 69（0x45） fstore_3 = 70（0x46）
操作数栈	…, value → …
描述	<n>必须是一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 float 类型的数据，这个数据将从操作数栈出栈，并且经过数值集合转换（§2.8.3）后得到值 value′，然后保存到<n>所指向的局部变量表位置中。
注意	fstore_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 fstore 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

fsub	
操作	float 类型数据相减
格式	**fsub**
结构	fsub = 102（0x66）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 float 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且经过数值集合转换（§2.8.3）后得到值 value1’和 value2’，接着将这两个数值相减（result = value1’－value2’），结果转换为 float 类型值 result，最后 result 被压入到操作数栈中。 对于一般 float 类型数据的减法来说，a－b 与 a＋（－b）的结果永远是一致的，但是对于 fsub 指令来说，与零相减的符号则会相反，因为如果 x 是＋0.0 的话，那 0.0－x 等于＋0.0，但－x 等于－0.0。 Java 虚拟机必须支持 IEEE 754 中定义的逐级下溢（Gradual Underflow），尽管指令执行期间，上溢、下溢以及精度丢失等情况都有可能发生，但 fsub 指令永远不会抛出任何运行时异常。

getfield	
操作	获取对象的字段值
格式	**getfield** **indexbyte1** **indexbyte2**
结构	getfield = 180（0xb4）
操作数栈	…, objectref → …, value
描述	objectref 必须是一个 reference 类型的数据，在指令执行时，objectref 将从操作数栈中出栈。无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8） | indexbyte2，该索引所指向的运行时常量池项应当是一个字段（§5.1）的符号引用，它包含了字段的名称和描述符，以及包含该字段的类的符号引用。这个字段的符号引用是已被解析过的（§5.4.3.2）。指令执行后，被 objectref 所引用的对象中该字段的值将会被取出，并推入到操作数栈顶。objectref 所引用的对象不能是数组类型，如果取值的字段是 protected 的（§4.6），并且这个字段是当前类的父类成员，并且这个字段没有在同一个运行时包（§5.3）中定义过，那 objectref 所指向的对象的类型必须为当前类或者当前类的子类。
链接时异常	在字段的符号引用解析过程中，任何在§5.4.3.2 中描述过的异常都可能会被抛出。 另外，如果已解析的字段是一个静态（static）字段，getfield 指令将会抛出一个 IncompatibleClassChangeError 异常
运行时异常	如果 objectref 为 null，getfield 指令将抛出一个

	NullPointerException.异常。
注意	不可以使用 getfield 指令来访问数组对象的 length 字段,如果要访问这个字段，应当使用 arraylength 指令。

getstatic	
操作	获取对象的静态字段值
格式	**getstatic** **indexbyte1** **indexbyte2**
结构	getstatic = 178（0xb2）
操作数栈	… → …, value
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个字段（§5.1）的符号引用，它包含了字段的名称和描述符，以及包含该字段的类或接口的符号引用。这个字段的符号引用是已被解析过的（§5.4.3.2）。 在字段被成功解析之后，如果字段所在的类或者接口没有被初始化过（§5.5），那指令执行时将会触发其初始化过程。 参数所指定的类或接口的该字段的值将会被取出，并推入到操作数栈顶。
链接时异常	在字段的符号引用解析过程中，任何在§5.4.3.2 中描述过的异常都可能会被抛出。 另外，如果已解析的字段是一个非静态（not static）字段，getstatic 指令将会抛出一个 IncompatibleClassChangeError 异常
运行时异常	如果 getstatic 指令触发了所涉及的类或接口的初始化，那 getstatic 指令就可能抛出在§5.5 中描述到的任何异常。

goto	
操作	分支跳转
格式	**goto** **branchbyte1** **branchbyte2**
结构	goto = 167（0xa7）
操作数栈	无改变
描述	无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 goto 指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 goto 指令所在的方法之中。

goto_w	
操作	分支跳转（宽范围）
格式	**goto_w** **branchbyte1** **branchbyte2** **branchbyte3** **branchbyte4**
结构	goto_w = 200（0xc8）
操作数栈	无改变
描述	无符号 byte 型数据 branchbyte1、branchbyte2、branchbyte3 和 branchbyte4 用于构建一个 32 位有符号的分支偏移量，构建方式为（branchbyte1 << 24）|（branchbyte1 << 16）|（branchbyte1 << 8）|branchbyte2。指令执行后，程序将会转到这个 goto_w 指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 goto_w 指令所在的方法之中。
注意	尽管 goto_w 指令拥有 4 字节宽度的分支偏移量，但是还受到方法最大字节码长度为 65535 字节（§4.11）的限制，这个限制值可能会在未来的 Java 虚拟机版本中增大。

i2b	
操作	将 int 类型数据转换为 byte 类型
格式	**i2b**
结构	i2b = 145（0x91）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，转换成 byte 类型数据，然后带符号扩展（Sign-Extended）回一个 int 的结果压入到操作数栈之中。
注意	i2b 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value 的数值大小发生改变，甚至导致转换结果与原值有不同的正负号。

i2c	
操作	将 int 类型数据转换为 char 类型
格式	**i2c**
结构	i2c = 146（0x92）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，转换成 byte 类型数据，然后零位扩展（Zero-Extended）回一个 int 的结果压入到操作数栈之中。
注意	i2c 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value 的数值大小发生改变，甚至导致转换结果（结果永远为正数）与原值有不同的正负号。

i2d	
操作	将 int 类型数据转换为 double 类型
格式	**i2d**
结构	i2d = 135（0x87）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，转换成 double 类型数据，然后压入到操作数栈之中。
注意	i2d 指令执行了宽化类型转换（Widening Primitive Conversion，JLS3 §5.1.2),因为所有 int 类型的数据都可以精确表示为 double 类型的数据，所以转换是精确的。

i2f	
操作	将 int 类型数据转换为 float 类型
格式	**i2f**
结构	i2f = 134（0x86）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，使用 IEEE 754 规范的向最接近数舍入模式转换成 float 类型数据，然后压入到操作数栈之中。
注意	i2f 指令执行了宽化类型转换（Widening Primitive Conversion，JLS3 §5.1.2），但是转换结果可能会有精度丢失，因为 float 类型只有 24 位有效数值位。

i2l	
操作	将 int 类型数据转换为 long 类型
格式	**i2l**
结构	i2l = 133（0x85）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，并带符号扩展（Sign-Extended）成 long 类型数据，然后压入到操作数栈之中。
注意	i2l 指令执行了宽化类型转换（Widening Primitive Conversion，JLS3 §5.1.2），因为所有 int 类型的数据都可以精确表示为 long 类型的数据，所以转换是精确的。

i2s	
操作	将 int 类型数据转换为 short 类型
格式	**i2s**
结构	i2s = 147（0x93）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 int 类型数据，指令执行时，它将从操作数栈中出栈，转换成 short 类型数据，然后带符号扩展（Sign-Extended）回一个 int 的结果压入到操作数栈之中。
注意	i2s 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value 的数值大小发生改变，甚至导致转换结果与原值有不同的正负号。

iadd	
操作	int 类型数据相加
格式	**iadd**
结构	iadd = 96（0x60）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，将这两个数值相加得到 int 类型数据 result（result=value1＋value2），最后 result 被压入到操作数栈中。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 32 位空间中，其数据类型为 int。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 iadd 指令的执行过程中不会抛出任何运行时异常。

iaload	
操作	从数组中加载一个 int 类型数据到操作数栈
格式	**iaload**
结构	iaload = 46（0x2e）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 int 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 int 类型值将压入到操作数栈中。
运行时异常	如果 arrayref 为 null，iaload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，iaload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

iand	
操作	对 int 类型数据进行按位与运算
格式	**iand**
结构	iand = 126（0x7e）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，对这两个数进行按位与（Bitwis And）操作得到 int 类型数据 result，最后 result 被压入到操作数栈中。

iastore	
操作	从操作数栈读取一个 int 类型数据存入到数组中
格式	**iastore**
结构	iastore = 79（0x4f）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 int 的数组，index 和 value 都必须为 int 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，然后 value 存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，iastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，iastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

iconst_<i>	
操作	将 int 类型数据压入到操作数栈中
格式	**iconst_<i>**
结构	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）
操作数栈	… → …, <i>
描述	将 int 类型的常量<i>（－1，0，1，2，3，4 或者 5）压入到操作数栈中。
注意	iconst_<i>指令族中的每一条指令都与使用<i>作为参数的 bipush 指令作的作用一致，仅仅除了操作数<i>是隐式包含在指令中这点不同而已。

idiv	
操作	int 类型数据除法
格式	**idiv**
结构	idiv = 108（0x6c）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 int 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且将这两个数值相除（value1÷value2），结果转换为 int 类型值 result，最后 result 被压入到操作数栈中。 int 类型的除法结果都是向零舍入的，这意味着 n÷d 的商 q 会在满足 |d|×|q|≤|n|的前提下取尽可能大的整数值。另外，当|n|≥|d|并且 n 和 d 符号相同时，q 的符号为正。而当|n|≥|d|并且 n 和 d 的符号相反时，q 的符号为负。 有一种特殊情况不适合上面的规则：如果被除数是 int 类型中绝对值最大的负数，除数为－1。那运算时将会发生溢出，运算结果就等于被除数本身。尽管这里发生了溢出，但是依然不会有异常抛出。
运行时异常	如果除数为零，idiv 指令将抛出 ArithmeticException 异常。

if_acmp<cond>

操作	reference 数据的据条件分支判断
格式	**if_acmp<cond>** **branchbyte1** **branchbyte2**
结构	if_acmpeq = 165（0xa5） if_acmpne = 166（0xa6）
操作数栈	…, value1, value2 → …
描述	value1 和 value2 都必须为 reference 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后进行比较运算，比较的规则如下： ❑ eq 当且仅当 value1=value2 比较的结果为真。 ❑ ne 当且仅当 value1≠value2 比较的结果为真。 如果比较结果为真，那无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 if_acmp<cond>指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 if_acmp<cond>指令所在的方法之中。 另外，如果比较结果为假，那程序将继续执行 if_acmp<cond>指令后面的其他直接码指令。

if_icmp<cond>

操作	int 数值的条件分支判断
格式	**if_icmp<cond>** **branchbyte1** **branchbyte2**
结构	if_icmpeq = 159（0x9f） if_icmpne = 160（0xa0） if_icmplt = 161（0xa1） if_icmpge = 162（0xa2） if_icmpgt = 163（0xa3） if_icmple = 164（0xa4）
操作数栈	…, value1, value2 → …
描述	value1 和 value2 都必须为 int 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后进行比较运算（所有比较都是带符号的），比较的规则如下： ❑ eq 当且仅当 value1=value2 比较的结果为真。 ❑ ne 当且仅当 value1≠value2 比较的结果为真。 ❑ lt 当且仅当 value1<value2 比较的结果为真。 ❑ le 当且仅当 value1≤value2 比较的结果为真。 ❑ ge 当且仅当 value1>value2 比较的结果为真。 ❑ ge 当且仅当 value1≥value2 比较的结果为真。 如果比较结果为真，那无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 if_icmp<cond>指令之

	后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 if_icmp<cond>指令所在的方法之中。 另外，如果比较结果为假，那程序将继续执行 if_acmp<cond>指令后面的其他直接码指令。

if<cond>

操作	整数与零比较的条件分支判断
格式	**if<cond>** **branchbyte1** **branchbyte2**
结构	ifeq = 153（0x99） ifne = 154（0x9a） iflt = 155（0x9b） ifge = 156（0x9c） ifgt = 157（0x9d） ifle = 158（0x9e）
操作数栈	…, value → …
描述	value 必须为 int 类型数据，指令执行时，value 从操作数栈中出栈，然后与零值进行比较（所有比较都是带符号的），比较的规则如下： ❑ eq 当且仅当 value=0 比较的结果为真。 ❑ ne 当且仅当 value≠0 比较的结果为真。 ❑ lt 当且仅当 value<0 比较的结果为真。 ❑ le 当且仅当 value≤0 比较的结果为真。 ❑ ge 当且仅当 value>0 比较的结果为真。 ❑ ge 当且仅当 value≥0 比较的结果为真。 如果比较结果为真，那无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 if<cond>指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 if<cond>

	指令所在的方法之中。 另外，如果比较结果为假，那程序将继续执行 if_acmp<cond>指令后面的其他直接码指令。

ifnonnull	
操作	引用不为空的条件分支判断
格式	**ifnonnull** **branchbyte1** **branchbyte2**
结构	ifnonnull = 199（0xc7）
操作数栈	…, value → …
描述	value 必须为 reference 类型数据，指令执行时，value 从操作数栈中出栈，然后判断是否为 null，如果 value 不为 null，那无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 ifnonnull 指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 ifnonnull 指令所在的方法之中。 另外，如果比较结果为假，那程序将继续执行 ifnonnull 指令后面的其他直接码指令。

ifnull	
操作	引用为空的条件分支判断
格式	**ifnull** **branchbyte1** **branchbyte2**
结构	fnull = 198（0xc6）
操作数栈	…, value → …
描述	value 必须为 reference 类型数据，指令执行时，value 从操作数栈中出栈，然后判断是否为 null，如果 value 为 null，那无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行后，程序将会转到这个 ifnull 指令之后的，由上述偏移量确定的目标地址上继续执行。这个目标地址必须处于 ifnull 指令所在的方法之中。 另外，如果比较结果为假，那程序将继续执行 ifnull 指令后面的其他直接码指令。

Iinc	
操作	局部变量自增
格式	**iinc** **index** **const**
结构	iinc = 132（0x84）
操作数栈	无改变
描述	index 是一个代表当前栈帧（§2.6）中局部变量表的索引的无符号 byte 类型整数，const 是一个有符号的 byte 类型数值。由 index 定位到的局部变量必须是 int 类型，const 首先带符号扩展成一个 int 类型数值，然后加到由 index 定位到的局部变量中。
注意	iinc 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表以及令局部变量增加 2 个字节长度的有符号数值。

iload	
操作	从局部变量表加载一个 int 类型值到操作数栈中
格式	**iload** index
结构	iload = 21（0x15）
操作数栈	… → …, value
描述	index 是一个代表当前栈帧（§2.6）中局部变量表的索引的无符号 byte 类型整数，index 作为索引定位的局部变量必须为 int 类型，记为 value。指令执行后，value 将会压入到操作数栈栈顶
注意	iload 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表。

iload_<n>	
操作	从局部变量表加载一个 int 类型值到操作数栈中
格式	**iload_<n>**
结构	iload_0 = 26（0x1a） iload_1 = 27（0x1b） iload_2 = 28（0x1c） iload_3 = 29（0x1d）
操作数栈	… → …, value
描述	<n>代表一个当前栈帧（§2.6）中局部变量表的索引值，<n>作为索引定位的局部变量必须为 int 类型，记作 value。指令执行后，value 将会压入到操作数栈栈顶
注意	iload_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 iload 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

imul	
操作	int 类型数据乘法
格式	**imul**
结构	imul = 104（0x68）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，接着将这两个数值相乘（value1×value2），结果压入到操作数栈中。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 32 位空间中，其数据类型为 int。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 imul 指令的执行过程中不会抛出任何运行时异常。

ineg	
操作	int 类型数据取负运算
格式	**ineg**
结构	ineg = 116（0x74）
操作数栈	…, value → …, result
描述	value 必须为 int 类型数据，指令执行时，value 从操作数栈中出栈，接着对这个数进行算术取负运算，运算结果－value 被压入到操作数栈中。 对于 int 类型数据，取负运算等同于与零做减法运算。因为 Java 虚拟机使用二进制补码来表示整数，而且二进制补码值的范围并不是完全对称的，int 类型中绝对值最大的负数取反的结果也依然是它本身。尽管指令执行过程中可能发生上限溢出，但是不会抛出任何异常。 对于所有的 int 类型值 x 来说，－x 等于（~x）＋1

instanceof

操作	判断对象是否指定的类型
格式	**instanceof** **indexbyte1** **indexbyte2**
结构	instanceof = 193（0xc1）
操作数栈	…, value → …, result
描述	objectref 必须是一个 reference 类型的数据，在指令执行时，objectref 将从操作数栈中出栈。无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个类、接口或者数组类型的符号引用。 如果 objectref 为 null 的话，那 instanceof 指令将会把 int 值 0 推入到操作数栈栈顶。 另外，如果参数指定的类、接口或者数组类型会被虚拟机解析（§5.4.3.1）。如果 objectref 可以转换为这个类、接口或者数组类型，那 instanceof 指令将会把 int 值 1 推入到操作数栈栈顶；否则，推入栈顶的就是 int 值 0。 以下规则可以用来确定一个非空的 objectref 是否可以转换为指定的已解析类型：假设 S 是 objectref 所指向的对象的类型，T 是进行比较的已解析的类、接口或者数组类型，checkcast 指令根据这些规则来判断转换是否成立： ❑ 如果 S 是类类型（Class Type），那么： ■ 如果 T 也是类类型，那 S 必须与 T 是同一个类类型，或者 S 是 T 所代表的类型的子类。 ■ 如果 T 是接口类型，那 S 必须实现了 T 的接口。

	❑ 如果 S 是接口类型（Interface Type），那么： ■ 如果 T 是类类型，那么 T 只能是 Object。 ■ 如果 T 是接口类型，那么 T 与 S 应当是相同的接口，或者 T 是 S 的父接口。 ❑ 如果 S 是数组类型（Array Type），假设为 SC[]的形式，这个数组的组件类型为 SC，那么： ■ 如果 T 是类类型，那么 T 只能是 Object。 ■ 如果 T 是数组类型，假设为 TC[]的形式，这个数组的组件类型为 TC，那么下面两条规则之一必须成立： ◆ TC 和 SC 是同一个原始类型。 ◆ TC 和 SC 都是 reference 类型，并且 SC 能与 TC 类型相匹配（以此处描述的规则来判断是否互相匹配）。 如果 T 是接口类型，那 T 必须是数组所实现的接口之一（JLS3 §4.10.3）。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。
运行时异常	如果 objectref 不能转换成参数指定的类、接口或者数组类型，checkcast 指令将抛出 ClassCastException 异常
注意	instanceof 指令与 checkcast 指令非常类似，它们之间的区别是如何处理 null 值的情况、测试类型转换的结果反馈方式（checkcast 是抛异常，而 instanceof 是返回一个比较结果）以及指令执行后对操作数栈的影响。

inovkedynamic

操作	调用动态方法
格式	**inovkedynamic** **indexbyte1** **indexbyte2** **0** **0**
结构	invokedynamic = 186 (0xba)
操作数栈	…, [arg1, [arg2 …]]→ …
描述	代码中每条 invokedynamic 指令出现的位置都被称为一个动态调用点（Dynamic Call Site）。 首先。无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个调用点限定符（§5.1）的符号引用。指令第 3、4 个操作数固定为 0。 调用点限定符会被解析（§5.4.3.6）为一个动态调用点，从中可以获取到 java.lang.invoke.MethodHandle 实例的引用、java.lang.invoke.MethodType 实例的引用和所涉及的静态参数的引用。接着，作为调用点限定符解析过程的一部分，引导方法将会被执行。如同使用 invokevirtual 指令调用普通方法那样，会包含一个运行时常量池的索引指向一个带有如下属性的方法（§5.1）： ❑ 此方法名为 invoke。 ❑ 此方法描述符中的返回值是 java.lang.invoke.CallSite。

- 此方法描述符中的参数来源自操作数栈中的元素，包括如下顺序排列的 4 个参数：
  - java.lang.invoke.MethodHandle
  - java.lang.invoke.MethodHandles.Lookup
  - String
  - java.lang.invoke.MethodType
- 如果调用点限定符有静态参数，那么这些参数的参数类型应附加在方法描述符的参数类型中，以便在调用时按顺序入栈至操作数栈。静态参数的参数类型可以包括：
  - Class
  - java.lang.invoke.MethodHandle
  - java.lang.invoke.MethodType
  - String
  - int
  - long
  - float
  - double
- 在 java.lang.invoke.MethodHandle 之中可以提供关于在哪个类中能找到方法的符号引用所对应的实际方法的信息。

引导方法执行前，下面各项内容将会按顺序压入到操作数栈中：

- 用于代表引导方法的 java.lang.invoke.MethodHandle 对象的引用。
- 用于确定动态调用点发生位置的 java.lang.invoke.MethodHandles.Lookup 对象的引用。
- 用于确定调用点限定符中方法名的 String 对象引用。
- 在方法限定符中出现的各种静态参数，包括：类、方法类型、方法句柄、字符串以及各种数值类型（§2.3.1, §2.3.2）都必须按照他们在方法限定符中出现的顺序依次入栈（此处基本类型不会发生自动装箱）。

只要引导方法能够被正确调用，它的描述符可以是不精确的。举个例子，引导方法的第一个参数应该是 java.lang.invoke.MethodHandles.Lookup，但是在可以使用 Object 来代替，返回值应该是 java.lang.invoke.CallSite，也可以使用 Object 来代替。

如果引导方法是一个变长参数方法（Variable Arity Method），那某些（甚至是全部）上面描述中要压入到操作数栈的参数会被包含在一个数组参数之中。

引导方法的调用发生在试图解析动态方法调用点的调用点限定符的那条线程上，如果同时有多条线程进行此操作，那引导方法将会被并发调用。因此，如果引导方法中有访问公有数据的话，需要注意多线程竞争问题，对公有数据访问施行适当的保护措施。

引导方法执行后的返回值是一个 java.lang.invoke.CallSite 或其子类的实例，这个对象被称为调用点对象（Call Site Object），此对象的引用将会从操作数栈中出栈，就像 invokevirtual 指令执行过程一样。

如果多条线程同时执行了一个动态调用点的引导方法，那 Java 虚拟机必须选择其中一个引导方法的返回值作为调用点对象，并将其发布到所有线程之中。此动态调用点中其余的引导方法会完成整个执行过程，但是它们的返回结果将被忽略掉，转为使用哪个被 Java 虚拟机选中的调用点对象来继续执行。

调用点对象拥有一个类型描述符（一个 java.lang.invoke.MethodType 的实例），它必须语义上等同于调用点限定符中方法描述符内所包含的 java.lang.invoke.MethodType 对象。

调用点限定符解析的结果是一个调用点对象，此对象将会与它的动态调用点永久绑定起来。

绑定于动态调用点的调用点对象所表示的方法句柄将会被调用，这次调用就和执行 invokevirtual 指令一样，会带有一个指向运行时常量池的索引，它指向的常量池项是一个方法的符号引用，此方法具备如下属性：

- 方法名为 invokeExact。
- 方法描述符为调用点限定符中包含的描述符。

	❑ 由 java.lang.invoke.MethodHandle 来确定在哪个类中查找方法的符号引用所对应的方法。 指令执行时，操作数栈中的内容会被虚拟机解释为包含一个调用点对象的引用以及跟随 nargs 个参数值，这些参数的数量、类型和顺序都必须与调用点限定符中的方法描述符保持一致。
链接时异常	如果调用点限定符的符号引用解析过程中抛出了异常 E，那 invokedynamic 指令必须抛出包装着异常 E 的 BootstrapMethodError 异常。 另外，在调用点限定符的后续解析过程中，如果引导方法执行过程因异常 E 而异常退出（§2.6.5），那 invokedynamic 指令必须抛出包装着异常 E 的 BootstrapMethodError 异常。 （这可能是由于引导方式有错误的参数长度、参数类型或者返回值而导致 java.lang.invoke.MethodHandle.invoke 方法抛出了 java.lang.invoke.WrongMethodTypeException 异常。） 另外，在调用点限定符的后续解析过程中，如果引导方法的返回值不是一个 java.lang.invoke.CallSite 的实例，那 invokedynamic 指令必须抛出 BootstrapMethodError 异常。 另外，在调用点限定符的后续解析过程中，如果调用点对象的目标的类型描述符与方法限定符中所包括的方法描述符不一致，那 invokedynamic 指令必须抛出 BootstrapMethodError 异常。
运行时异常	如果动态调用点的调用点限定符解析过程成功完成，那就意味着将有一个非空的 java.lang.invoke.CallSite 的实例绑定到该动态调用点之上。因此，操作数栈中表示调用点目标的对象不会为空，这也意味着，调用点限定符中的方法描述符与等效于被 invokevirtual 指令所调用方法句柄那个方法句柄的类型描述符语义上是一致的。 上面描述的意思是已经绑定了调用点对象的 invokedynamic 指令，永远不可能抛出 NullPointerException 异常或者 java.lang.invoke.WrongMethodTypeException 异常。

invokeinterface

操作	调用接口方法
格式	**invokeinterface** **indexbyte1** **indexbyte2** **count** **0**
结构	invokeinterface = 185（0xb9）
操作数栈	…, objectref, [arg1, [arg2 …]] → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个接口方法（§5.1）的符号引用，它包含了方法的名称和描述符，以及包含该方法的接口的符号引用。这个方法的符号引用是已被解析过的（§5.4.3.2），而且这个方法不能是实例初始化方法（§2.9）和类或接口的初始化方法（§2.9）。 操作数 count 是一个无符号 byte 型数据，而且不能为零。objectref 必须是一个 reference 类型的数据。在操作数栈中，objectref 之后还跟随着连续 n 个参数值，这些参数的数值、数据类型和顺序都必须遵循接口方法的描述符中的描述。invokeinterface 指令的第四个参数规定永远为 byte 类型的 0。 假设 C 是 objectref 所对应的类，虚拟机将按下面规则查找实际执行的方法： ❑ 如果 C 中包含了名称和描述符都和要调用的接口方法一致的方法，那这个方法就会被调用，查找过程终止。

	❑ 另外，如果 C 有父类，查找过程将按顺序递归搜索 C 的直接父类，如果超类中能搜索到名称和描述符都和要调用的接口方法一致的方法，那这个方法就会被调用。 ❑ 另外，如果抛出 AbstractMethodError 异常。 如果要调用的是同步方法，那与 objectref 相关的管程（monitor）将会进入或者重入，就如当前线程中同执行了 monitorenter 指令一般。 如果要调用的不是本地方法，n 个 args 参数和 objectref 将从操作数栈中出栈。方法调用的时候，一个新的栈帧将在 Java 虚拟机栈中被创建出来，objectref 和连续的 n 个参数将存放到新栈帧的局部变量表中，objectref 存为局部变量 0，arg1 存为局部变量 1（如果 arg1 是 long 或 double 类型，那将占用局部变量 1 和 2 两个位置），依此类推。参数中的浮点类型数据在存入局部变量之前会先进行数值集合转换（§2.8.3）。新栈帧创建后就成为当前栈帧，Java 虚拟机的 PC 寄存器被设置成指向调用方法的首条指令，程序就从这里开始继续执行。 如果要调用的是本地方法，要是这些平台相关的代码尚未绑定（§5.6）到虚拟机中的话，绑定动作先要完成。指令执行时，n 个 args 参数和 objectref 将从操作数栈中出栈并作为参数传递给实现此方法的代码。参数中的浮点类型数据在传递给调用方法之前会先进行数值集合转换（§2.8.3）。参数传递和代码执行都会以具体虚拟机实现相关的方式进行。当这些平台相关的代码返回时： ❑ 如果这个本地方法是同步方法，那与 objectref 相关的管程状态将会被更新，也可能退出了，就如当前线程中同执行了 monitorexit 指令一般。 ❑ 如果这个本地方法有返回值，那平台相关的代码返回的数据必须通过某种实现相关的方式转换成本地方法所定义的 Java 类型，并压入到操作数栈中。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.4 章节中描述的异常都可能被抛出。

运行时异常	如果 objectref 为 null，invokeinterface 指令将抛出 NullPointerException 异常。 另外，如果 objectref 所对应的对象并未实现接口方法中所需的接口，那 invokeinterface 指令将抛出 IncompatibleClassChangeError 异常。 另外，如果没有找到任何名称和描述符都与要调用的接口方法一致的方法，那 invokeinterface 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法不是 public 的话，那 invokeinterface 指令将抛出 IllegalAccessError 异常。 另外，如果搜索到的方法是 abstract 的话，那 invokeinterface 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法是 native 的话，当实现代码实现代码无法绑定到虚拟机中,那 invokeinterface 指令将抛出 UnsatisfiedLinkError 异常。
注意	invokeinterface 指令的 count 操作数用于确定参数的数量，long 和 double 类型的参数占用 2 个数量单位,而其他类型的参数占用 1 个数量单位。其实这些信息完全可以从方法的描述符中获取到，有这个参数完全是历史原因。 invokeinterface 指令的第四个操作数是为了给 Oracle 实现的虚拟机的额外操作数而预留的空间，invokeinterface 指令会在运行时被替换为特殊的其他指令，这必须要保持向后兼容性。 objectref 和随后的 n 个参数并不一定与局部变量表的数量一一对应，因为参数中的 long 和 double 类型参数需要使用 2 个连续的局部变量来存储，因此在参数传递的时候，可能需要比参数个数更多的局部变量。

invokespecial

操作	调用实例方法；专门用来处理调用超类方法、使用方法和实例初始化方法方法。
格式	**invokespecial** **indexbyte1** **indexbyte2**
结构	invokespecial = 183（0xb7）
操作数栈	…，objectref，[arg1，[arg2 …]] → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个方法（§5.1）的符号引用，它包含了方法的名称和描述符，以及包含该方法的接口的符号引用。最后，如果调用的方法是 protected 的（§4.6），并且这个方法是当前类的父类成员，并且这个方法没有在同一个运行时包（§5.3）中定义过，那 objectref 所指向的对象的类型必须为当前类或者当前类的子类。 只有下面所有的条件都成立的前提下，才会进行调用方法的搜索： ❑ 当前类的 ACC_SUPER 标志为真（参见表 4-1，“类访问和属性修改”）。 ❑ 调用方法所在的类似当前类的超类。 ❑ 调用方法不是实例初始化方法（§2.9）。 如果以上条件都成立，虚拟机将按下面规则查找实际执行的方法，假设 C 是当前类的直接父类： ❑ 如果 C 中包含了名称和描述符都和要调用的实例方法一致的方法，那这个方法就会被调用，查找过程终止。 ❑ 另外，如果 C 有父类，查找过程将按顺序递归搜索 C 的直接父类，如果超类中能搜索到名称和描述符都和要调用的实例方法一致的方法，那这个方

	法就会被调用。 ❑ 否则，抛出 AbstractMethodError 异常。 objectref 必须是一个 reference 类型的数据，在操作数栈中，objectref 之后还跟随着连续 n 个参数值，这些参数的数值、数据类型和顺序都必须遵循实例方法的描述符中的描述。 如果要调用的是同步方法，那与 objectref 相关的管程（monitor）将会进入或者重入，就如当前线程中同执行了 monitorenter 指令一般。 如果要调用的不是本地方法，n 个 args 参数和 objectref 将从操作数栈中出栈。方法调用的时候，一个新的栈帧将在 Java 虚拟机栈中被创建出来，objectref 和连续的 n 个参数将存放到新栈帧的局部变量表中，objectref 存为局部变量 0，arg1 存为局部变量 1（如果 arg1 是 long 或 double 类型，那将占用局部变量 1 和 2 两个位置），依此类推。参数中的浮点类型数据在存入局部变量之前会先进行数值集合转换（§2.8.3）。新栈帧创建后就成为当前栈帧，Java 虚拟机的 PC 寄存器被设置成指向调用方法的首条指令，程序就从这里开始继续执行。 如果要调用的是本地方法，要是这些平台相关的代码尚未绑定（§5.6）到虚拟机中的话，绑定动作先要完成。指令执行时，n 个 args 参数和 objectref 将从操作数栈中出栈并作为参数传递给实现此方法的代码。参数中的浮点类型数据在传递给调用方法之前会先进行数值集合转换（§2.8.3）。参数传递和代码执行都会以具体虚拟机实现相关的方式进行。当这些平台相关的代码返回时： ❑ 如果这个本地方法是同步方法，那与 objectref 相关的管程状态将会被更新，也可能退出了，就如当前线程中同执行了 monitorexit 指令一般。 ❑ 如果这个本地方法有返回值，那平台相关的代码返回的数据必须通过某种实现相关的方式转换成本地方法所定义的 Java 类型，并压入到操作数栈中。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.3 章节中描述的异常都可能被抛出。

	另外，如果调用方法是实例初始化方法，但是定义这个方法的类与指令参数中符号引用所代表的类并不是同一个，那 invokespecial 指令将抛出 NoSuchMethodError 异常。 另外，如果调用方法是一个类（静态，即 static）方法，那 invokespecial 指令将抛出 IncompatibleClassChangeError 异常。
运行时异常	如果 objectref 为 null，invokespecial 指令将抛出 NullPointerException 异常。 另外，如果没有找到任何名称和描述符都与要调用的接口方法一致的方法，那 invokespecial 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法是 abstract 方法的话，那 invokespecial 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法是 native 方法的话，当实现代码实现代码无法绑定到虚拟机中，那 invokespecial 指令将抛出 UnsatisfiedLinkError 异常。
注意	invokespecial 和 invokevirtual 指令之间的差异是：invokevirtual 指令用于调用象所属的类中定义的方法，而 invokespecial 指令用于调用实例初始化方法（§2.9）、private 方法和当前类的超类中的方法。 在 JDK 1.0.2 之前，invokespecial 指令曾被命名为 invokenonvirtual。 objectref 和随后的 n 个参数并不一定与局部变量表的数量一一对应，因为参数中的 long 和 double 类型参数需要使用 2 个连续的局部变量来存储，因此在参数传递的时候，可能需要比参数个数更多的局部变量。

invokestatic

操作	调用静态方法
格式	**invokestatic** **indexbyte1** **indexbyte2**
结构	invokestatic = 184（0xb8）
操作数栈	…, [arg1, [arg2 …]] → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个方法（§5.1）的符号引用，它包含了方法的名称和描述符（§4.3.3），以及包含该方法的接口的符号引用。 此方法应是已被解析过（§5.4.3.3）的，而且不能是实例的初始化方法和类或接口的初始化方法。这个方法必须被声明为 static，因此它也不能是 abstract 方法。 在方法被成功解析之后，如果方法所在的类没有被初始化过（§5.5），那指令执行时将会触发其初始化过程。 在操作数栈中必须包含连续 n 个参数值，这些参数的数值、数据类型和顺序都必须遵循实例方法的描述符中的描述。 如果要调用的是同步方法，那与这个类的 Class 对象相关的管程（monitor）将会进入或者重入，就如当前线程中同执行了 monitorenter 指令一般。 如果要调用的不是本地方法，n 个 args 参数将从操作数栈中出栈。方法调用的时候，一个新的栈帧将在 Java 虚拟机栈中被创建出来，连续的 n 个参数将存放到新栈帧的局部变量表中，arg1 存为局部变量 1（如果 arg1 是 long 或 double 类型，那将占用局部变量 1 和 2 两个位置），依此类推。参数中的浮

	点类型数据在存入局部变量之前会先进行数值集合转换（§2.8.3）。新栈帧创建后就成为当前栈帧，Java 虚拟机的 PC 寄存器被设置成指向调用方法的首条指令，程序就从这里开始继续执行。 如果要调用的是本地方法，要是这些平台相关的代码尚未绑定（§5.6）到虚拟机中的话，绑定动作先要完成。指令执行时，n 个 args 参数将从操作数栈中出栈并作为参数传递给实现此方法的代码。参数中的浮点类型数据在传递给调用方法之前会先进行数值集合转换（§2.8.3）。参数传递和代码执行都会以具体虚拟机实现相关的方式进行。当这些平台相关的代码返回时： ❑ 如果这个本地方法是同步方法，那与它所属类的 Class 对象相关的管程状态将会被更新，也可能退出了，就如当前线程中同执行了 monitorexit 指令一般。 ❑ 如果这个本地方法有返回值，那平台相关的代码返回的数据必须通过某种实现相关的方式转换成本地方法所定义的 Java 类型，并压入到操作数栈中。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.3 章节中描述的异常都可能被抛出。 另外，如果调用方法是实例方法，那 invokestatic 指令将抛出 IncompatibleClassChangeError 异常。
运行时异常	如果 invokestatic 指令执行时触发了类的初始化过程，那 invokestatic 指令有可能所有在§5.5 中描述过的异常。 另外，如果执行的方法是 native 方法的话，当实现代码实现代码无法绑定到虚拟机中，那 invokestatic 指令将抛出 UnsatisfiedLinkError 异常。
注意	方法调用使用到的 n 个参数的总个数并非与局部变量表的个数是一一对应的，因为参数中的 long 和 double 类型参数需要使用 2 个连续的局部变量来存储，因此在参数传递的时候，可能需要比参数个数更多的局部变量。

invokevirtual

操作	调用实例方法，依据实例的类型进行分派
格式	**invokevirtual** **indexbyte1** **indexbyte2**
结构	invokevirtual = 182（0xb6）
操作数栈	…，objectref，[arg1，[arg2 …]] → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个方法（§5.1）的符号引用，它包含了方法的名称和描述符，以及包含该方法的接口的符号引用。这个方法的符号引用是已被解析过的（§5.4.3.3），而且这个方法不能是实例初始化方法（§2.9）和类或接口的初始化方法（§2.9）。最后，如果调用的方法是 protected 的（§4.6），并且这个方法是当前类的父类成员，并且这个方法没有在同一个运行时包（§5.3）中定义过，那 objectref 所指向的对象的类型必须为当前类或者当前类的子类。 假设 C 是 objectref 所对应的类，虚拟机将按下面规则查找实际执行的方法： ❑ 如果 C 中定义了一个实例方法 M，它了重写（override，§5.4.3.5）了符号引用中表示的方法，那方法 M 就会被调用，查找过程终止。 ❑ 另外，如果 C 有父类，查找过程将按第一点的方式顺序递归搜索 C 的直接父类，如果超类中能搜索到符合的方法，那这个方法就会被调用。 ❑ 否则，抛出 AbstractMethodError 异常。 在操作数栈中，objectref 之后必须跟随 N 个参数，它们的数量、类型和顺序都必须与方法描述符所描述的保持一致。

	如果要调用的是同步方法，那与 objectref 相关的管程（monitor）将会进入或者重入，就如当前线程中同执行了 monitorenter 指令一般。 如果要调用的不是本地方法，n 个 args 参数和 objectref 将从操作数栈中出栈。方法调用的时候，一个新的栈帧将在 Java 虚拟机栈中被创建出来，objectref 和连续的 n 个参数将存放到新栈帧的局部变量表中，objectref 存为局部变量 0，arg1 存为局部变量 1（如果 arg1 是 long 或 double 类型，那将占用局部变量 1 和 2 两个位置），依此类推。参数中的浮点类型数据在存入局部变量之前会先进行数值集合转换（§2.8.3）。新栈帧创建后就成为当前栈帧，Java 虚拟机的 PC 寄存器被设置成指向调用方法的首条指令，程序就从这里开始继续执行。 如果要调用的是本地方法，要是这些平台相关的代码尚未绑定（§5.6）到虚拟机中的话，绑定动作先要完成。指令执行时，n 个 args 参数和 objectref 将从操作数栈中出栈并作为参数传递给实现此方法的代码。参数中的浮点类型数据在传递给调用方法之前会先进行数值集合转换（§2.8.3）。参数传递和代码执行都会以具体虚拟机实现相关的方式进行。当这些平台相关的代码返回时： ❑ 如果这个本地方法是同步方法，那与 objectref 相关的管程状态将会被更新，也可能退出了，就如当前线程中同执行了 monitorexit 指令一般。 ❑ 如果这个本地方法有返回值，那平台相关的代码返回的数据必须通过某种实现相关的方式转换成本地方法所定义的 Java 类型，并压入到操作数栈中。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.4 章节中描述的异常都可能被抛出。 另外，如果被调用的方法是一个 static 方法，那 invokevirtual 指令将会抛出一个 IncompatibleClassChangeError 异常。
运行时异常	如果 objectref 为 null，invokevirtual 指令将抛出 NullPointerException 异常。

	另外，如果没有找到任何名称和描述符都与要调用的接口方法一致的方法，那 invokevirtual 指令将抛出 AbstractMethodError 异常。 另外，如果被调用方法是 abstract 的，那 invokevirtual 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法是 abstract 的话，那 invokevirtual 指令将抛出 AbstractMethodError 异常。 另外，如果搜索到的方法是 native 的话，当实现代码实现代码无法绑定到虚拟机中，那 invokevirtual 指令将抛出 UnsatisfiedLinkError 异常。
注意	objectref 和随后的 n 个参数并不一定与局部变量表的数量一一对应，因为参数中的 long 和 double 类型参数需要使用 2 个连续的局部变量来存储，因此在参数传递的时候，可能需要比参数个数更多的局部变量。

ior	
操作	int 类型数值的布尔或运算
格式	**ior**
结构	ior = 128（0x80）
操作数栈	…, value1, value2 → …, result
描述	value1、value2 必须为 int 类型数据，指令执行时，它们从操作数栈中出栈，接着对这 2 个数进行按位或（Bitwise Inclusive OR）运算，运算结果 result 被压入到操作数栈中。

irem	
操作	int 类型数据求余
格式	**irem**
结构	irem = 112（0x70）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，根据 value1－（value1÷value2）×value2 计算出结果，然后把运算结果入栈回操作数栈中。 irem 指令的运算结果就是保证（a÷b）×b ＋（a%b）=a 能够成立，唯一的特殊情况是当被除数是 int 类型绝对值最大的负数，并且除数为－1 的时候（这时候余数值为 0）。irem 运算指令执行时会遵循当被除数为负数时余数才能是负数，当被除数为正数时余数才能是正数的规则。另外，irem 运算结果的绝对值永远小于除数的绝对值。
运行时异常	如果除数为 0，irem 指令将会抛出一个 ArithmeticException 异常。

ireturn	
操作	结束方法，并返回一个 int 类型数据
格式	**ireturn**
结构	ireturn = 172（0xac）
操作数栈	…, value → [empty]
描述	当前方法的返回值必须为 boolean、short、char 或者 int 类型，value 必须是一个 int 类型的数据。如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，那 value 将从当前栈帧（§2.6）中出栈，然后压入到调用者栈帧的操作数栈中，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在§2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 ireturn 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，ireturn 指令也会抛出 IllegalMonitorStateException 异常。

ishl	
操作	int 数值左移运算
格式	**ishl**
结构	ishl = 120（0x78）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，然后将 value1 左移 s 位，s 是 value2 低 5 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	这个操作（即使是出现了溢出的情况下）等同于把 value1 乘以 2 的 s 次方，位移的距离实际上被限制在 0 到 31 之间，相当于指令执行时会把 value2 与 0x1f 做一遍算术与操作。

ishr	
操作	int 数值右移运算
格式	**ishr**
结构	ishr = 122（0x7a）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，然后将 value1 右移 s 位，s 是 value2 低 5 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	这个操作的结果等于[value1÷2^s]，这里的 s 是 value2 与 0x1f 算术与运算后的结果。对于 value1 为非负数的情况，这个操作等同于把 value1 除以 2 的 s 次方。位移的距离实际上被限制在 0 到 31 之间，相当于指令执行时会把 value2 与 0x1f 做一遍算术与操作。

istore	
操作	将一个 int 类型数据保存到局部变量表中
格式	**istore** index
结构	istore = 54（0x36）
操作数栈	…, value → …
描述	index 是一个无符号 byte 型整数，它指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 int 类型的数据，这个数据将从操作数栈出栈，然后保存到 index 所指向的局部变量表位置中。
注意	istore 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

istore_<n>	
操作	将一个 int 类型数据保存到局部变量表中
格式	**istore_<n>**
结构	istore_0 = 59（0x3b） istore_1 = 60（0x3c） istore_2 = 61（0x3d） istore_3 = 62（0x3e）
操作数栈	…, value → …
描述	<n>必须是一个指向当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 int 类型的数据，这个数据将从操作数栈出栈，然后保存到<n>所指向的局部变量表位置中。
注意	istore_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 istore 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

isub	
操作	int 类型数据相减
格式	**isub**
结构	isub = 100（0x64）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，将这两个数值相减（result = value1′− value2′），结果转换为 int 类型值 result，最后 result 被压入到操作数栈中。 对于 int 类型数据的减法来说，a−b 与 a+（−b）的结果永远是一致的，0 减去某个 int 值相当于对这个 int 值进行取负运算。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 32 位空间中，其数据类型为 int。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 isub 指令的执行过程中不会抛出任何运行时异常。

iushr	
操作	int 数值逻辑右移运算
格式	**iushr**
结构	iushr = 124（0x7c）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据,指令执行时,value1 和 value2 从操作数栈中出栈，然后将 value1 右移 s 位，s 是 value2 低 5 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	假设value1是正数并且s为value2与0x1f算术与运算后的结果,那iushr指令的运算结果与 value1 >> s 的结果是一致的；假设 value1 是负数，那 iushr 指令的运算结果与表达式（value1 >> s）＋（2 << ~s）一致。附加的（2 << ~s）操作用于取消符号位的移动。位移的距离实际上被限制在 0 到 31 之间，

ixor	
操作	int 数值异或运算
格式	**ixor**
结构	ixor = 130（0x82）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 int 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后将 value1 和 value2 进行按位异或运算，并把运算结果入栈回操作数栈中。

jsr	
操作	程序段落跳转
格式	**jsr** **branchbyte1** **branchbyte2**
结构	jsr = 168（0xa8）
操作数栈	…，→ …，address
描述	address 是一个 returnAddress 类型的数据，它由 jsr 指令推入操作数栈中。无符号 byte 型数据 branchbyte1 和 branchbyte2 用于构建一个 16 位有符号的分支偏移量，构建方式为（branchbyte1 << 8）| branchbyte2。指令执行时，将产生一个当前位置的偏移坐标，并压入到操作数栈中。跳转目标地址必须在 jsr 指令所在的方法之内。
注意	请注意，jsr 指令将 address 压入到操作数栈，ret 指令从局部变量表中把它取出，这种不对称的操作是故意设计的。 在 Oracle 的实现 Java 语言编译器中，Java SE 6 前的版本是使用 jsr 和 ret 指令配合来实现 finally 语句块的。详细信息读者可以参见 § 3.13 “编译 finally 语句块”和 § 4.10.2.5 “异常与 finally”。

jsr_w	
操作	程序段落跳转
格式	**jsr_w** **branchbyte1** **branchbyte2** **branchbyte3** **branchbyte4**
结构	jsr_w = 201（0xc9）
操作数栈	…，→ …，address
描述	address 是一个 returnAddress 类型的数据，它由 jsr_w 指令推入操作数栈中。无符号 byte 型数据 branchbyte1、branchbyte2、branchbyte3 和 branchbyte4 用于构建一个 32 位有符号的分支偏移量，构建方式为（branchbyte1 << 24）|（branchbyte1 << 16）|（branchbyte1 << 8）|branchbyte2。指令执行时，将产生一个当前位置的偏移坐标，并压入到操作数栈中。跳转目标地址必须在 jsr_w 指令所在的方法之内。
注意	jsr_w 指令被用来与 ret 指令一同实现 Java 语言中的 finally 语句块（参见§3.13“编译 fianlly 语句块”）。请注意，jsr_w 指令推送 address 到操作数栈，ret 指令从局部变量表中把它取出，这种不对称的操作是故意设计的。 虽然 jsr_w 指令拥有 4 个字节的分支偏移量，但是其他因素限定了一个方法的最大长度不能超过 65535 个字节（§4.11）。这个上限值可能会在将来发布的 Java 虚拟机中被提升。

l2d	
操作	将 long 类型数据转换为 double 类型
格式	**l2d**
结构	l2d = 138（0x8a）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 long 类型数据，指令执行时，它将从操作数栈中出栈，使用 IEEE 754 规范的向最接近数舍入模式转换成 double 类型数据，然后压入到操作数栈之中。
注意	i2d 指令执行了宽化类型转换（Widening Primitive Conversion，JLS3 §5.1.2），由于 double 类型只有 53 位有效位数，所以转换可能会产生精度丢失。

l2f	
操作	将 long 类型数据转换为 float 类型
格式	**l2f**
结构	l2f = 137（0x89）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 long 类型数据，指令执行时，它将从操作数栈中出栈，使用 IEEE 754 规范的向最接近数舍入模式转换成 float 类型数据，然后压入到操作数栈之中。
注意	i2d 指令执行了宽化类型转换（Widening Primitive Conversion，JLS3§5.1.2），由于 double 类型只有 24 位有效位数，所以转换可能会产生精度丢失。

l2i	
操作	将 long 类型数据转换为 int 类型
格式	**l2i**
结构	l2i = 136（0x88）
操作数栈	…, value → …, result
描述	value 必须是在操作数栈栈顶的 long 类型数据，指令执行时，它将从操作数栈中出栈，使用保留低 32 位、丢弃高 32 位的方式转换为 int 类型数据，然后压入到操作数栈之中。
注意	i2d 指令执行了窄化类型转换（Narrowing Primitive Conversion，JLS3 §5.1.3），它可能会导致 value 的数值大小发生改变，甚至导致转换结果与原值有不同的正负号。

ladd	
操作	long 类型数据相加
格式	**ladd**
结构	ladd = 97（0x61）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，将这两个数值相加得到 long 类型数据 result（result=value1＋value2），最后 result 被压入到操作数栈中。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 64 位空间中，其数据类型为 long。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 ladd 指令的执行过程中不会抛出任何运行时异常。

laload	
操作	从数组中加载一个 long 类型数据到操作数栈
格式	**laload**
结构	laload = 47（0x2f）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 int 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 long 类型值将压入到操作数栈中。
运行时异常	如果 arrayref 为 null，laload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，laload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

land	
操作	对 long 类型数据进行按位与运算
格式	**land**
结构	land = 127（0x7f）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，对这两个数进行按位与（Bitwis And）操作得到 long 类型数据 result，最后 result 被压入到操作数栈中。

lastore	
操作	从操作数栈读取一个 long 类型数据存入到数组中
格式	**lastore**
结构	lastore = 80（0x50）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 long 的数组，index 必须为 int 类型，而 value 必须为 long 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，然后 value 存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，iastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，lastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

lcmp	
操作	比较 2 个 long 类型数据的大小
格式	**lcmp**
结构	lcmp = 148（0x94）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，使用一个 int 数值作为比较结果：如果 value1 大于 value2，结果为 1；如果 value1 等于 value2，结果为 0；如果 value1 小于 value2，结果为－1，最后比较结果被压入到操作数栈中。

lconst_<l>	
操作	将 long 类型数据压入到操作数栈中
格式	**lconst_<l>**
结构	lconst_0 = 9（0x9） lconst_1 = 10（0xa）
操作数栈	… → …, <l>
描述	将 int 类型的常量<l>（0 或者 1）压入到操作数栈中。

ldc	
操作	从运行时常量池中提取数据推入操作数栈
格式	**ldc** **index**
结构	ldc = 18（0x12）
操作数栈	… → …, value
描述	index 是一个无符号 byte 型数据，它作为当前类（§2.6）的运行时常量池的索引使用。index 指向的运行时常量池项必须是一个 int 或者 float 类型的运行时常量，或者是一个类的符号引用（§5.4.3.1）或者字符串字面量（§5.1）。 如果运行时常量池项必须是一个 int 或者 float 类型的运行时常量，那数值这个常量所对应的数值 value 将被压入到操作数栈之中。 另外，如果运行时常量池项必须是一个代表字符串字面量（§5.1）的 String 类的引用，那这个实例的引用所对应的 reference 类型数据 value 将被压入到操作数栈之中。 另外，如果运行时常量池项必须是一个类的符号引用（§4.4.1），这个符号引用是已被解析（§5.4.3.1）过的，那这个类的 Class 对象所对应的 reference 类型数据 value 将被压入到操作数栈之中。
链接时异常	在类的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。
注意	ldc 指令只能用来处理单精度浮点集合（§2.3.2）中的 float 类型数据，因为常量池（§4.4.4）中 float 类型的常量必须从单精度浮点集合中选取。

ldc_w

操作	从运行时常量池中提取数据推入操作数栈（宽索引）
格式	**ldc_w** **indexbyte1** **indexbyte2**
结构	ldc_w = 19（0x13）
操作数栈	… → …, value
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个 int 或者 float 类型的运行时常量，或者是一个类的符号引用（§5.4.3.1）或者字符串字面量（§5.1）。 如果运行时常量池项必须是一个 int 或者 float 类型的运行时常量，那数值这个常量所对应的数值 value 将被压入到操作数栈之中。 另外，如果运行时常量池项必须是一个代表字符串字面量（§5.1）的 String 类的引用，那这个实例的引用所对应的 reference 类型数据 value 将被压入到操作数栈之中。 另外，如果运行时常量池项必须是一个类的符号引用（§4.4.1），这个符号引用是已被解析（§5.4.3.1）过的，那这个类的 Class 对象所对应的 reference 类型数据 value 将被压入到操作数栈之中。
链接时异常	在类的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。
注意	ldc_w 指令与 ldc 指令的差别在于它使用了更宽的运行时常量池索引。 ldc_w 指令只能用来处理单精度浮点集合（§2.3.2）中的 float 类型数据，因为常量池（§4.4.4）中 float 类型的常量必须从单精度浮点集合中选取。

ldc2_w	
操作	从运行时常量池中提取 long 或 double 数据推入操作数栈（宽索引）
格式	**ldc2_w** **indexbyte1** **indexbyte2**
结构	ldc2_w = 20（0x14）
操作数栈	… → …, value
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个 long 或者 double 类型的运行时常量（§5.1）。数值这个常量所对应的数值将被压入到操作数栈之中。
注意	只有宽索引版本的 ldc2_w 指令，不存在单 byte 类型索引的 ldc2 指令用于从运行时常量池中提取 long 或 double 数据推入操作数栈。 ldc2_w 指令只能用来处理双精度浮点集合（§2.3.2）中的 double 类型数据，因为常量池（§4.4.5）中 double 类型的常量必须从双精度浮点集合中选取。

ldiv	
操作	long 类型数据除法
格式	**ldiv**
结构	ldiv = 109（0x6d）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，并且将这两个数值相除（value1÷value2），结果转换为 long 类型值 result，最后 result 被压入到操作数栈中。 int 类型的除法结果都是向零舍入的，这意味着 n÷d 的商 q 会在满足 |d|×|q|≤|n|的前提下取尽可能大的整数值。另外，当|n|≥|d|并且 n 和 d 符号相同时，q 的符号为正。而当|n|≥|d|并且 n 和 d 的符号相反时，q 的符号为负。 有一种特殊情况不适合上面的规则：如果被除数是 long 类型中绝对值最大的负数，除数为－1。那运算时将会发生溢出，运算结果就等于被除数本身。尽管这里发生了溢出，但是依然不会有异常抛出。
运行时异常	如果除数为零，ldiv 指令将抛出 ArithmeticException 异常。

lload	
操作	从局部变量表加载一个 long 类型值到操作数栈中
格式	**lload** index
结构	iload = 22（0x16）
操作数栈	… → …, value
描述	index 是一个无符号 byte 类型整数，它与 index＋1 共同构成一个当前栈帧（§2.6）中局部变量表的索引的，index 作为索引定位的局部变量必须为 long 类型，记为 value。指令执行后，value 将会压入到操作数栈栈顶
注意	lload 操作码可以与 wide 指令联合一起实现使用 2 个字节长度的无符号 byte 型数值作为索引来访问局部变量表。

lload_<n>	
操作	从局部变量表加载一个 long 类型值到操作数栈中
格式	**lload_<n>**
结构	lload_0 = 30（0x1e） lload_1 = 31（0x1f） lload_2 = 32（0x20） lload_3 = 33（0x21）
操作数栈	… → …, value
描述	<n>与<n>＋1 共同构成一个当前栈帧（§2.6）中局部变量表的索引的，<n>作为索引定位的局部变量必须为 long 类型，记为 value。指令执行后，value 将会压入到操作数栈栈顶
注意	lload_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 lload 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

lmul	
操作	long 类型数据乘法
格式	**lmul**
结构	lmul = 105（0x69）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，接着将这两个数值相乘（value1×value2），结果压入到操作数栈中。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 64 位空间中，其数据类型为 long。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 lmul 指令的执行过程中不会抛出任何运行时异常。

lneg	
操作	long 类型数据取负运算
格式	**lneg**
结构	lneg = 117（0x75）
操作数栈	…, value → …, result
描述	value 必须为 long 类型数据，指令执行时，value 从操作数栈中出栈，接着对这个数进行算术取负运算，运算结果－value 被压入到操作数栈中。 对于 long 类型数据，取负运算等同于与零做减法运算。因为 Java 虚拟机使用二进制补码来表示整数，而且二进制补码值的范围并不是完全对称的，long 类型中绝对值最大的负数取反的结果也依然是它本身。尽管指令执行过程中可能发生上限溢出，但是不会抛出任何异常。 对于所有的 long 类型值 x 来说，－x 等于（~x）＋1

lookupswitch

操作	根据键值在跳转表中寻找配对的分支并进行跳转
格式	**lookupswitch** **<0-3 byte pad>** **defaultbyte1** **defaultbyte2** **defaultbyte3** **defaultbyte4** **npairs1** **npairs2** **npairs3** **npairs4** **match-offset pairs…**
结构	lookupswitch = 171（0xab）
操作数栈	…, key → …
描述	lookupswitch 是一条变长指令。紧跟 lookupswitch 之后的 0 至 3 个字节作为空白填充，而后面 defaultbyte1 至 defaultbyte4 等代表了一个个由 4 个字节组成的、从当前方法开始（第一条操作码指令）计算的地址，即紧跟随空白填充的是一系列 32 位有符号整数值：包括默认跳转地址 default、匹配坐标的数量 npairs 以及 npairs 组匹配坐标。其中，npairs 的值应当大于或等于 0，每一组匹配坐标都包含了一个整数值 match 以及一个有符号 32 位偏移量 offset。上述所有的 32 位有符号数值都由以下形式构成：（byte1 << 24）|（byte2 << 16）|（byte3 << 8）| byte4。

	lookupswitch 指令之后所有的匹配坐标必须以其中的 match 值排序，按照升序储存。 指令执行时，int 型的 key 从操作数栈中出栈，与每一个 match 值相互比较。如果能找到一个与之相等的 match 值，那就就以这个 match 所配对的偏移量 offset 作为目标地址进行跳转。如果没有配对到任何一个 match 值，那就是用 default 作为目标地址进行跳转。程序从目标地址开始继续执行。 目标地址既可能从 npairs 组匹配坐标中得出，也可能从 default 中得出，但无论如何，最终的目标地址必须在包含 lookupswitch 指令的那个方法之内。
注意	当且仅当包含 lookupswitch 指令的方法刚好位于 4 字节边界上，lookupswitch 指令才能确保它的所有操作数都是 4 直接对齐的。 所有的匹配坐标以有序方式存储是为了查找时的效率考虑。

lor	
操作	long 类型数值的布尔或运算
格式	**lor**
结构	lor = 129（0x81）
操作数栈	…，value1，value2 → …，result
描述	value1、value2 必须为 long 类型数据，指令执行时，它们从操作数栈中出栈，接着对这 2 个数进行按位或（Bitwise Inclusive OR）运算，long 类型的运算结果 result 被压入到操作数栈中。

lrem	
操作	long 类型数据求余
格式	**lrem**
结构	lrem = 113（0x71）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，根据 value1－（value1÷value2）×value2 计算出结果，然后把运算结果入栈回操作数栈中。 lrem 指令的运算结果就是保证（a÷b）×b ＋（a%b）=a 能够成立，唯一的特殊情况是当被除数是 long 类型绝对值最大的负数，并且除数为－1 的时候（这时候余数值为 0）。lrem 运算指令执行时会遵循当被除数为负数时余数才能是负数，当被除数为正数时余数才能是正数的规则。另外，lrem 运算结果的绝对值永远小于除数的绝对值。
运行时异常	如果除数为 0，lrem 指令将会抛出一个 ArithmeticException 异常。

lreturn	
操作	结束方法，并返回一个 long 类型数据
格式	**lreturn**
结构	lreturn = 173（0xad）
操作数栈	…, value → [empty]
描述	当前方法的返回值必须为 long 类型，value 必须是一个 long 类型的数据。 如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，那 value 将从当前栈帧（§2.6）中出栈，然后压入到调用者栈帧的操作数栈中，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在§2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 lreturn 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了§2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，lreturn 指令也会抛出 IllegalMonitorStateException 异常。

lshl	
操作	long 数值左移运算
格式	**lshl**
结构	lshl = 121（0x79）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后将 value1 左移 s 位，s 是 value2 低 6 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	这个操作（即使是出现了溢出的情况下）等同于把 value1 乘以 2 的 s 次方，位移的距离实际上被限制在 0 到 63 之间，相当于指令执行时会把 value2 与 0x3f 做一遍算术与操作。

lshr	
操作	long 数值右移运算
格式	**lshr**
结构	lshr = 123（0x7b）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后将 value1 右移 s 位，s 是 value2 低 6 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	这个操作的结果等于[value1÷2^s]，这里的 s 是 value2 与 0x3f 算术与运算后的结果。对于 value1 为非负数的情况，这个操作等同于把 value1 除以 2 的 s 次方。位移的距离实际上被限制在 0 到 63 之间，相当于指令执行时会把 value2 与 0x1f 做一遍算术与操作。

lstore	
操作	将一个 long 类型数据保存到局部变量表中
格式	**lstore** index
结构	lstore = 55（0x37）
操作数栈	…, value → …
描述	index 是一个无符号 byte 型整数，它与 index 共同表示一个当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 long 类型的数据，这个数据将从操作数栈出栈，然后保存到 index 及 index＋1 所指向的局部变量表位置中。
注意	lstore 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

lstore_<n>	
操作	将一个 long 类型数据保存到局部变量表中
格式	**lstore_<n>**
结构	lstore_0 = 63（0x3f） lstore_1 = 64（0x40） lstore_2 = 65（0x41） lstore_3 = 66（0x42）
操作数栈	…, value → …
描述	<n>与<n>＋1 共同表示一个当前栈帧（§2.6）局部变量表的索引值，而在操作数栈栈顶的 value 必须是 long 类型的数据，这个数据将从操作数栈出栈，然后保存到<n>及<n>＋1 所指向的局部变量表位置中。
注意	lstore_<n>指令族中的每一条指令都与使用<n>作为 index 参数的 lstore 指令作的作用一致，仅仅除了操作数<n>是隐式包含在指令中这点不同而已。

lsub	
操作	long 类型数据相减
格式	**lsub**
结构	lsub = 101（0x65）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，将这两个数值相减（result = value1′− value2′），结果转换为 long 类型值 result，最后 result 被压入到操作数栈中。 对于 long 类型数据的减法来说，a−b 与 a+（−b）的结果永远是一致的，0 减去某个 long 值相当于对这个 long 值进行取负运算。 运算的结果使用低位在高地址（Low-Order Bites）的顺序、按照二进制补码形式存储在 64 位空间中，其数据类型为 long。如果发生了上限溢出，那结果的符号可能与真正数学运算结果的符号相反。 尽管可能发生上限溢出，但是 lsub 指令的执行过程中不会抛出任何运行时异常。

lushr	
操作	long 数值逻辑右移运算
格式	**lushr**
结构	lushr = 125（0x7d）
操作数栈	…，value1，value2 → …，result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后将 value1 右移 s 位，s 是 value2 低 6 位所表示的值，计算后把运算结果入栈回操作数栈中。
注意	假设 value1 是正数并且 s 为 value2 与 0x3f 算术与运算后的结果，那 lushr 指令的运算结果与 value1 >> s 的结果是一致的；假设 value1 是负数，那 lushr 指令的运算结果与表达式（value1 >> s）＋（2L << ~s）一致。附加的（2L << ~s）操作用于取消符号位的移动。位移的距离实际上被限制在 0 到 63 之间，

lxor	
操作	long 数值异或运算
格式	**lxor**
结构	lxor = 131（0x83）
操作数栈	…, value1, value2 → …, result
描述	value1 和 value2 都必须为 long 类型数据，指令执行时，value1 和 value2 从操作数栈中出栈，然后将 value1 和 value2 进行按位异或运算，并把运算结果入栈回操作数栈中。

monitorenter	
操作	进入一个对象的 monitor
格式	**monitorenter**
结构	monitorenter = 194（0xc2）
操作数栈	…，objectref → …
描述	objectref 必须为 reference 类型数据。 任何对象都有一个 monitor 与之关联。当且仅当一个 monitor 被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取 objectref 所对应的 monitor 的所有权，那么： 如果 objectref 的 monitor 的进入计数器为 0，那线程可以成功进入 monitor，以及将计数器值设置为 1。当前线程就是 monitor 的所有者。 如果当前线程已经拥有 objectref 的 monitor 的所有权，那它可以重入这个 monitor，重入时需将进入计数器的值加 1。 如果其他线程已经拥有 objectref 的 monitor 的所有权，那当前线程将被阻塞，直到 monitor 的进入计数器值变为 0 时，重新尝试获取 monitor 的所有权。
运行时异常	当 objectref 为 null 时，monitorenter 指令将抛出 NullPointerException 异常。
注意	一个 monitorenter 指令可能会与一个或多个 monitorexit 指令配合实现 Java 语言中 synchronized 同步语句块的语义（§3.14）。但 monitorenter 和 monitorexit 指令不会用来实现 synchronized 方法的语义，尽管它们确实可以实现类似的语义。当一个 synchronized 方法被调用时，自动进入对应的 monitor，当方法返回时，自动退出 monitor，这些

	动作是 Java 虚拟机在调用和返回指令中隐式处理的。 对象与它的 monitor 之间的关联关系有很多种实现方式，这些内容已超出本规范的范围之外，但可以稍作介绍。monitor 即可以实现为与对象一同分配和销毁，也可以在某条线程尝试获取对象所有权时动态生成，在没有任何线程持有对象所有权时自动释放。 在 Java 语言里面，同步的概念除了包括 monitor 的进入和退出操作以外，还包括有等待（Object.wait）和唤醒（Object.notifyAll 和 Object.notify）。这些操作包含在 Java 虚拟机提供的标准包 java.lang 之中，而不是通过 Java 虚拟机的指令集来显式支持。

monitorexit	
操作	退出一个对象的 monitor
格式	**monitorexit**
结构	monitorexit = 195（0xc3）
操作数栈	…，objectref → …
描述	objectref 必须为 reference 类型数据。 执行 monitorexit 指令的线程必须是 objectref 对应的 monitor 的所有者。 指令执行时，线程把 monitor 的进入计数器值减 1，如果减 1 后计数器值为 0，那线程退出 monitor，不再是这个 monitor 的拥有者。其他被这个 monitor 阻塞的线程可以尝试去获取这个 monitor 的所有权。
运行时异常	当 objectref 为 null 时，monitorexit 指令将抛出 NullPointerException 异常。 另外，如果执行 monitorexit 的线程原本并没有这个 monitor 的所有权，那 monitorexit 指令将抛出 IllegalMonitorStateException.异常。 另外，如果 Java 虚拟机执行 monitorexit 时发现违反了§2.11.10 中第二条规则，那 monitorexit 指令将抛出 IllegalMonitorStateException.异常。
注意	一个 monitorenter 指令可能会与一个或多个 monitorexit 指令配合实现 Java 语言中 synchronized 同步语句块的语义（§3.14）。但 monitorenter 和 monitorexit 指令不会用来实现 synchronized 方法的语义，尽管它们确实可以实现类似的语义。 Java 虚拟机对在 synchronized 方法和 synchronized 同步语句块中抛出

	的异常有不同的处理方式： 在 synchronized 方法正常完成时，monitor 通过 Java 虚拟机的返回指令退出。在 synchronized 方法非正常完成时，monitor 通过 Java 虚拟机的 athrow 指令退出。 当有异常从 synchronized 同步语句块抛出，将由 Java 虚拟机异常处理机制（§3.14）来保证退出了之前在 synchronized 同步语句块开始时进入的 monitor。

multianewarray

操作	创建一个新的多维数组
格式	**multianewarray** indexbyte1 indexbyte2 dimensions
结构	multianewarray = 197（0xc5）
操作数栈	…, count1, [count2, …] → …, arrayref
描述	dimensions 操作数是一个无符号 byte 类型数据，它必须大于或等于 1，代表创建数组的维度值。相应地，操作数栈中必须包含 dimensions 个数值，数组每一个值代表每个维度中需要创建的元素数量。这些值必须为非负数 int 类型数据。count1 描述第一个维度的长度，count2 描述第二个维度的长度，依此类推。 指令执行时，所有 count 都将从操作数栈中出栈，无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个类、接口或者数组类型的符号引用，这个类、接口或者数组类型应当是已被解析（§5.4.3.1）的。指令执行产生的结果将会是一个维度不小于 dimensions 的数组。 一个新的多维数组将会被分配在 GC 堆中，如果任何一个 count 值为 0，那就不会分配维度。数组第一维的元素被初始化为第二维的子数组，后面每一维都依此类推。数组的最后一个维度的元素将会被分配为数组元素类型的初始值（§2.3，§2.4）。并且一个代表该数组的 reference 类型数据 arrayref 压入到操作数栈中。

链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。 另外，如果当前类没有权限访问数组的元素类型，multianewarray 指令将会抛出 IllegalAccessError 异常。
运行时异常	另外，如果 dimensions 值小于 0 的话，multianewarray 指令将会抛出一个 NegativeArraySizeException 异常。
注意	对于一维数组来说，使用 newarray 或者 anewarray 指令创建会更加高效。在运行时常量池中确定的数组类型维度可能比操作数栈中 dimensions 所代表的维度更高，在这种情况下，multianewarray 指令只会创建数组的第一个维度。

new	
操作	创建一个对象
格式	**new** indexbyte1 indexbyte2
结构	new = 187（0xbb）
操作数栈	… → …, objectref
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个类或接口的符号引用，这个类或接口类型应当是已被解析（§5.4.3.1）并且最终解析结果为某个具体的类型。一个以此为类型为对象将会被分配在 GC 堆中，并且它所有的实例变量都会进行初始化为相应类型的初始值（§2.3，§2.4）。一个代表该对象实例的 reference 类型数据 objectref 将压入到操作数栈中。 对于一个已成功解析但是未初始化（§5.5）的类型，在这时将会进行初始化。
链接时异常	在类、接口或者数组的符号解析阶段，任何在§5.4.3.1 章节中描述的异常都可能被抛出。 另外，如果在类、接口或者数组的符号引用最终被解析为一个接口或抽象类，new 指令将抛出 InstantiationError 异常。
运行时异常	另外，如果 new 指令触发了类的初始化，new 指令可能抛出任意在《Java 语言规范》§15.9.4 中所描述的异常。
注意	new 指令执行后并没有完成一个对象实例创建的全部过程，只有实例初始化方

	法被执行并完成后，实例才算完全创建。

newarray

操作

创建一个数组

格式

newarray
atype

结构

newarray = 188（0xbc）

操作数栈

…, count →

…, arrayref

描述

count 为 int 类型的数据，指令执行时它将从操作数栈中出栈，它代表了要创建多大的数组。

atype 为要创建数组的元素类型，它将为以下值之一：

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

一个以 atype 为组件类型、以 count 值为长度的数组将会被分配在 GC 堆中，并且一个代表该数组的 reference 类型数据 arrayref 压入到操作数栈中。这个新数组的所有元素将会被分配为相应类型的初始值（§2.3，§2.4）。

运行时异常

如果 count 值小于 0 的话，newarray 指令将会抛出一个 NegativeArraySizeException 异常。

注意	在 Oracle 实现的 Java 虚拟机中，布尔类型（atype 值为 T_BOOLEAN）是以 8 位储存，并使用 baload 和 bastore 指令操作，这些指令也可以操作 byte 类型的数组。其他 Java 虚拟机可能有自己的 boolean 型数组实现方式，但必须保证 baload 和 bastore 指令依然使用于它们的 boolean 类型数组。

nop	
操作	什么事情都不做
格式	**nop**
结构	nop = 0（0x0）
操作数栈	无变化
描述	什么事情都不做

pop	
操作	将操作数栈的栈顶元素出栈
格式	**pop**
结构	pop = 87（0x57）
操作数栈	…, value → …
描述	将操作数栈的栈顶元素出栈。 pop 指令只能用来操作（§2.11.1）中定义的分类一的运算类型。

pop2	
操作	将操作数栈的栈顶一个或两个元素出栈
格式	**pop2**
结构	pop2 = 88（0x58）
操作数栈	结构 1： …, value2, value1 → … 这时候 value1 和 value2 都必须为（§2.11.1）中定义的分类一的运算类型。 结构 2： …, value → … 这时候 value 必须为（§2.11.1）中定义的分类二的运算类型。
描述	将操作数栈的栈顶一个或两个元素出栈。

putfield

操作	设置对象字段
格式	**putfield** **indexbyte1** **indexbyte2**
结构	putfield = 181（0xb5）
操作数栈	…，objectref，value → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个字段（§5.1）的符号引用，它包含了字段的名称和描述符，以及包含该字段的类的符号引用。objectref 所引用的对象不能是数组类型，如果取值的字段是 protected 的（§4.6），并且这个字段是当前类的父类成员，并且这个字段没有在同一个运行时包（§5.3）中定义过，那 objectref 所指向的对象的类型必须为当前类或者当前类的子类。 这个字段的符号引用是已被解析过的（§5.4.3.2）。被 putfield 指令存储到字段中的 value 值的类型必须与字段的描述符相匹配（§4.3.2）。如果字段描述符的类型是 boolean、byte、char、short 或者 int，那么 value 必须为 int 类型。如果字段描述符的类型是 float、long 或者 double，那 value 的类型必须相应为 float、long 或者 double。如果字段描述符的类型是 reference 类型，那 value 必须为一个可与之匹配（JLS §5.2）的类型。如果字段被声明为 final 的，那就只有在当前类的实例初始化方法（<init>）中设置当前类的 final 字段才是合法的。 指令执行时，value 和 objectref 从操作数栈中出栈，objectref 必须为

	reference 类型数据，value 将根据（§2.8.3）中定义的转换规则转换为 value′，objectref 的指定字段的值将被设置为 value′。
链接时异常	在字段的符号引用解析过程中，任何在§5.4.3.2 中描述过的异常都可能会被抛出。 另外，如果已解析的字段是一个静态（static）字段，getfield 指令将会抛出一个 IncompatibleClassChangeError 异常。 另外,如果字段声明为 final,那就只有在当前类的实例初始化方法(<init>)中设置当前类的 final 字段才是合法的，否则将会抛出 IllegalAccessError 异常。
运行时异常	另外，如果 objectref 为 null，putfield 指令将抛出一个 NullPointerException.异常。

putstatic	
操作	设置对象的静态字段值
格式	**putstatic** **indexbyte1** **indexbyte2**
结构	putstatic = 179（0xb3）
操作数栈	…, value → …
描述	无符号数 indexbyte1 和 indexbyte2 用于构建一个当前类（§2.6）的运行时常量池的索引值，构建方式为（indexbyte1 << 8）| indexbyte2，该索引所指向的运行时常量池项应当是一个字段（§5.1）的符号引用，它包含了字段的名称和描述符，以及包含该字段的类或接口的符号引用。这个字段的符号引用是已被解析过的（§5.4.3.2）。 在字段被成功解析之后，如果字段所在的类或者接口没有被初始化过（§5.5），那指令执行时将会触发其初始化过程。 被putstatic指令存储到字段中的value值的类型必须与字段的描述符相匹配（§4.3.2）。如果字段描述符的类型是 boolean、byte、char、short 或者 int，那么 value 必须为 int 类型。如果字段描述符的类型是 float、long 或者 double，那 value 的类型必须相应为 float、long 或者 double。如果字段描述符的类型是 reference 类型，那 value 必须为一个可与之匹配（JLS §5.2）的类型。如果字段被声明为 final 的，那就只有在当前类的类初始化方法（<clinit>）中设置当前类的 final 字段才是合法的。 指令执行时，value 从操作数栈中出栈，根据（§2.8.3）中定义的转换规则转换为 value′，类的指定字段的值将被设置为 value′。

链接时异常	在字段的符号引用解析过程中，任何在§5.4.3.2 中描述过的异常都可能会被抛出。 另外，如果已解析的字段是一个非静态（not static）字段，putstatic 指令将会抛出一个 IncompatibleClassChangeError 异常。 另外，如果字段声明为 final，那就只有在当前类的实例初始化方法（<clinit>）中设置当前类的 final 字段才是合法的，否则将会抛出 IllegalAccessError 异常。
运行时异常	另外，如果 putstatic 指令触发了所涉及的类或接口的初始化，那 putstatic 指令就可能抛出在§5.5 中描述到的任何异常。
注意	putstatic 指令只有在接口初始化时才能用来设置接口字段的值，接口字段只会在接口初始化的时候初始化赋值一次（§5.5，JLS §9.3.1）。

ret	
操作	代码片段中返回
格式	**ret** **index**
结构	ret = 169（0xa9）
操作数栈	无变化
描述	index 是一个 0 至 255 之间的无符号数，它代表一个当前栈帧（§2.6）的局部变量表的索引值，在该索引位置应为一个 returnAddress 类型的局部变量，指令执行后，将该局部变量的值更新到 Java 虚拟机的 PC 寄存器中，令程序从修改后的位置继续执行。
注意	ret 指令被用来与 jsr、jsr_w 指令一同实现 Java 语言中的 finally 语句块（参见§3.13“编译 finally 语句块”）。请注意，jsr_w 指令推送 address 到操作数栈，ret 指令从局部变量表中把它取出，这种不对称的操作是故意设计的。 ret 指令不应与 return 指令混为一谈，return 是在没有返回值的方法返回时使用。 ret 指令可以与 wide 指令联合使用，以实现使用 2 字节宽度的无符号整数作为索引来访问局部变量表。

Ireturn	
操作	无返回值的方法返回
格式	**Return**
结构	return = 177（0xb1）
操作数栈	… → [empty]
描述	当前方法的返回值必须被声明为 void。如果当前方法是一个同步（声明为 synchronized）方法，那在方法调用时进入或者重入的管程应当被正确更新状态或退出，就像当前线程执行了 monitorexit 指令一样。如果执行过程当中没有异常被抛出的话，在当前栈帧操作数栈中所有其他的值都将会被丢弃掉。 指令执行后，解释器会恢复调用者的栈帧，并且把程序控制权交回到调用者。
运行时异常	如果虚拟机实现没有严格执行在 § 2.11.10 中规定的结构化锁定规则，导致当前方法是一个同步方法，但当前线程在调用方法时没有成功持有（Enter）或重入（Reentered）相应的管程，那 return 指令将会抛出 IllegalMonitorStateException 异常。这是可能出现的，譬如一个同步方法只包含了对方法同步对象的 monitorexit 指令，但是未包含配对的 monitorenter 指令。 另外，如果虚拟机实现严格执行了 § 2.11.10 中规定的结构化锁定规则，但当前方法调用时，其中的第一条规则被违反的话，return 指令也会抛出 IllegalMonitorStateException 异常。

saload	
操作	从数组中加载一个 short 类型数据到操作数栈
格式	**saload**
结构	saload = 53（0x35）
操作数栈	…，arrayref，index → …，value
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 int 的数组，index 必须为 int 类型。指令执行后，arrayref 和 index 同时从操作数栈出栈，index 作为索引定位到数组中的 short 类型值先被零位扩展（Zero-Extended）为一个 int 类型数据 value，然后再将 value 压入到操作数栈中。
运行时异常	如果 arrayref 为 null，saload 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，saload 指令将抛出 ArrayIndexOutOfBoundsException 异常。

sastore	
操作	从操作数栈读取一个 short 类型数据存入到数组中
格式	**sastore**
结构	sastore = 86（0x56）
操作数栈	…，arrayref，index，value → …
描述	arrayref 必须是一个 reference 类型的数据，它指向一个组件类型为 short 的数组，index 和 value 都必须为 int 类型。指令执行后，arrayref、index 和 value 同时从操作数栈出栈，value 将被转换为 short 类型，然后存储到 index 作为索引定位到数组元素中。
运行时异常	如果 arrayref 为 null，sastore 指令将抛出 NullPointerException 异常 另外，如果 index 不在 arrayref 所代表的数组上下界范围中，sastore 指令将抛出 ArrayIndexOutOfBoundsException 异常。

sipush	
操作	将一个 short 类型数据入栈
格式	**sipush** **byte1** **byte2**
结构	sipush = 17（0x11）
操作数栈	… → …, value
描述	无符号数 byte1 和 byte2 通过（byte1 << 8）| byte2 方式构造成一个 short 类型数值，然后此数值带符号扩展为一个 int 类型的值 value，然后将 value 压入到操作数栈中。

swap	
操作	交换操作数栈顶的两个值
格式	**swap**
结构	swap = 95（0x5f）
操作数栈	…, value2, value1 → …, value1, value2
描述	交换操作数栈顶的两个值。 swap 指令只有在 value1 和 value2 都是（§2.11.1）中定义的分类一的运算类型才能使用。 Java 虚拟机为提供交换操作数栈中两个分类二数值的指令。

tableswitch	
操作	根据索引值在跳转表中寻找配对的分支并进行跳转
格式	**tableswitch** **<0-3 byte pad>** **defaultbyte1** **defaultbyte2** **defaultbyte3** **defaultbyte4** **lowbyte1** **lowbyte2** **lowbyte3** **lowbyte4** **highbyte1** **highbyte2** **highbyte3** **highbyte4** **jump offsets…**
结构	tableswitch = 170（0xaa）
操作数栈	…, index → …
描述	tableswitch 是一条变长指令。紧跟 tableswitch 之后的 0 至 3 个字节作为空白填充，而后面 defaultbyte1 至 defaultbyte4 等代表了一个个由 4 个字节组成的、从当前方法开始（第一条操作码指令）计算的地址，即紧跟随空白填充的是一系列 32 位有符号整数值：包括默认跳转地址 default、高位

	值 high 以及低位值 low。在此之后，是 high－low＋1 个有符号 32 位偏移量 offset，其中要求 low 小于或等于 high。这 high－low＋1 个 32 位有符号数值形成一张零基址跳转表（0-Based Jump Table），所有上述的 32 位有符号数都以（byte1 << 24）|（byte2 << 16）|（byte3 << 8）| byte4 方式构成。 指令执行时，int 型的 index 从操作数栈中出栈，如果 index 比 low 值小或者比 high 值大，那就是用 default 作为目标地址进行跳转。否则，在跳转表中第 index-low 个地址值将作为目标地址进行跳转，程序从目标地址开始继续执行。 目标地址既可能从跳转表匹配坐标中得出，也可能从 default 中得出，但无论如何，最终的目标地址必须在包含 tableswitch 指令的那个方法之内。
注意	当且仅当包含 tableswitch 指令的方法刚好位于 4 字节边界上，lookupswitch 指令才能确保它的所有操作数都是 4 直接对齐的。

wide	
操作	扩展局部变量表索引
格式 1	**wide** **<opcode>** **indexbyte1** **indexbyte2** 当<opcode>为 iload，fload，aload，lload，dload，istore，fstore，astore，lstore，dstore 以及 ret 指令之一时。
格式 2	**wide** **iinc** **indexbyte1** **indexbyte2** **constbyte1** **constbyte2**
结构	wide = 196（0xc4）
操作数栈	与被扩展的指令一致
描述	wide 指令用于扩展其他指令的行为，取决于被不同扩展的指令，它可以有两种形式。第一种形式是当被扩展指令为 iload，fload，aload，lload，dload，istore，fstore，astore，lstore，dstore 以及 ret 指令之一时使用，第二种形式仅当被扩展指令为 iinc 时使用。 无论哪种形式，wide 指令后面都跟随者被扩展指令的操作码，之后是两个无符号 byte 型数值 indexbyte1 和 indexbyte2,它们通过(indexbyte1 << 8）| indexbyte2 的形式构成一个指向当前栈帧（§2.6）的局部变量表的 16 位无符号索引。随后，wide 使用这个被新计算出来的索引值替换掉被扩展

	指令中的索引参数，如果被扩展指令为 lload，dload，lstore 以及 dstore 指令，那 index 和 index＋1 都必须为合法的局部变量索引值。对于 wide 指令的第二种形式，在 indexbyte1 和 indexbyte2 后面还有另外两个无符号 byte 型数值 constbyte1 和 constbyte2，它们将以（constbyte1 << 8）| constbyte2 的形式构成一个有符号的 16 位常量。 被 wide 指令扩展的那些指令，行为上与原有指令的语义没有任何区别，仅仅是索引参数被替换了而已。对于 wide 指令的第二种形式，则是有了更大的增加范围。
注意	虽然我们所 wide 指令扩展了其他指令。实际上更准确的 wide 指令修改了这些指令的操作数，而不是改变这些指令的本来语义。对于 iinc 指令来说，则是被修改了全部两个操作数。被 wide 指令扩展的那些指令不应当脱离 wide 指令直接执行，即不能有任何跳转指令的目标是这些跟随在 wde 指令之后的字节码指令。

第 7 章 操作码助记符

本章中提供了一张以操作码值为索引的 Java 虚拟机指令操作码的映射表[①]，它包含了所有保留操作码（§6.2）到指令助记符的对应。

字节码	助记符	指令含义
0x00	nop	什么都不做。
0x01	aconst_null	将 null 推送至栈顶。
0x02	iconst_m1	将 int 型-1 推送至栈顶。
0x03	iconst_0	将 int 型 0 推送至栈顶。
0x04	iconst_1	将 int 型 1 推送至栈顶。
0x05	iconst_2	将 int 型 2 推送至栈顶。
0x06	iconst_3	将 int 型 3 推送至栈顶。
0x07	iconst_4	将 int 型 4 推送至栈顶。
0x08	iconst_5	将 int 型 5 推送至栈顶。
0x09	lconst_0	将 long 型 0 推送至栈顶。
0x0a	lconst_1	将 long 型 1 推送至栈顶。
0x0b	fconst_0	将 float 型 0 推送至栈顶。
0x0c	fconst_1	将 float 型 1 推送至栈顶。
0x0d	fconst_2	将 float 型 2 推送至栈顶。
0x0e	dconst_0	将 double 型 0 推送至栈顶。
0x0f	dconst_1	将 double 型 1 推送至栈顶。
0x10	bipush	将单字节的常量值（-128~127）推送至栈顶。
0x11	sipush	将一个短整型常量值（-32768~32767）推送至栈顶。
0x12	ldc	将 int，float 或 String 型常量值从常量池中推送至栈顶。
0x13	ldc_w	将 int，float 或 String 型常量值从常量池中推送至栈顶（宽索引）。

[①] 译者注：“指令含义”一列在规范原文中并不存在，是为了方便读者使用，译者自己加入的。

0x14	ldc2_w	将 long 或 double 型常量值从常量池中推送至栈顶（宽索引）。
0x15	iload	将指定的 int 型局部变量推送至栈顶。
0x16	lload	将指定的 long 型局部变量推送至栈顶。
0x17	fload	将指定的 float 型局部变量推送至栈顶。
0x18	dload	将指定的 double 型局部变量推送至栈顶。
0x19	aload	将指定的引用类型局部变量推送至栈顶。
0x1a	iload_0	将第一个 int 型局部变量推送至栈顶。
0x1b	iload_1	将第二个 int 型局部变量推送至栈顶。
0x1c	iload_2	将第三个 int 型局部变量推送至栈顶。
0x1d	iload_3	将第四个 int 型局部变量推送至栈顶。
0x1e	lload_0	将第一个 long 型局部变量推送至栈顶。
0x1f	lload_1	将第二个 long 型局部变量推送至栈顶。
0x20	lload_2	将第三个 long 型局部变量推送至栈顶。
0x21	lload_3	将第四个 long 型局部变量推送至栈顶。
0x22	fload_0	将第一个 float 型局部变量推送至栈顶。
0x23	fload_1	将第二个 float 型局部变量推送至栈顶。
0x24	fload_2	将第三个 float 型局部变量推送至栈顶
0x25	fload_3	将第四个 float 型局部变量推送至栈顶。
0x26	dload_0	将第一个 double 型局部变量推送至栈顶。
0x27	dload_1	将第二个 double 型局部变量推送至栈顶。
0x28	dload_2	将第三个 double 型局部变量推送至栈顶。
0x29	dload_3	将第四个 double 型局部变量推送至栈顶。
0x2a	aload_0	将第一个引用类型局部变量推送至栈顶。
0x2b	aload_1	将第二个引用类型局部变量推送至栈顶。
0x2c	aload_2	将第三个引用类型局部变量推送至栈顶。
0x2d	aload_3	将第四个引用类型局部变量推送至栈顶。
0x2e	iaload	将 int 型数组指定索引的值推送至栈顶。
0x2f	laload	将 long 型数组指定索引的值推送至栈顶。

0x30	faload	将 float 型数组指定索引的值推送至栈顶。
0x31	daload	将 double 型数组指定索引的值推送至栈顶。
0x32	aaload	将引用型数组指定索引的值推送至栈顶。
0x33	baload	将 boolean 或 byte 型数组指定索引的值推送至栈顶。
0x34	caload	将 char 型数组指定索引的值推送至栈顶。
0x35	saload	将 short 型数组指定索引的值推送至栈顶。
0x36	istore	将栈顶 int 型数值存入指定局部变量。
0x37	lstore	将栈顶 long 型数值存入指定局部变量。
0x38	fstore	将栈顶 float 型数值存入指定局部变量。
0x39	dstore	将栈顶 double 型数值存入指定局部变量。
0x3a	astore	将栈顶引用型数值存入指定局部变量。
0x3b	istore_0	将栈顶 int 型数值存入第一个局部变量。
0x3c	istore_1	将栈顶 int 型数值存入第二个局部变量。
0x3d	istore_2	将栈顶 int 型数值存入第三个局部变量。
0x3e	istore_3	将栈顶 int 型数值存入第四个局部变量。
0x3f	lstore_0	将栈顶 long 型数值存入第一个局部变量。
0x40	lstore_1	将栈顶 long 型数值存入第二个局部变量。
0x41	lstore_2	将栈顶 long 型数值存入第三个局部变量。
0x42	lstore_3	将栈顶 long 型数值存入第四个局部变量。
0x43	fstore_0	将栈顶 float 型数值存入第一个局部变量。
0x44	fstore_1	将栈顶 float 型数值存入第二个局部变量。
0x45	fstore_2	将栈顶 float 型数值存入第三个局部变量。
0x46	fstore_3	将栈顶 float 型数值存入第四个局部变量。
0x47	dstore_0	将栈顶 double 型数值存入第一个局部变量。
0x48	dstore_1	将栈顶 double 型数值存入第二个局部变量。
0x49	dstore_2	将栈顶 double 型数值存入第三个局部变量。
0x4a	dstore_3	将栈顶 double 型数值存入第四个局部变量。
0x4b	astore_0	将栈顶引用型数值存入第一个局部变量。

0x4c	astore_1	将栈顶引用型数值存入第二个局部变量。
0x4d	astore_2	将栈顶引用型数值存入第三个局部变量
0x4e	astore_3	将栈顶引用型数值存入第四个局部变量。
0x4f	iastore	将栈顶 int 型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶 long 型数值存入指定数组的指定索引位置。
0x51	fastore	将栈顶 float 型数值存入指定数组的指定索引位置。
0x52	dastore	将栈顶 double 型数值存入指定数组的指定索引位置。
0x53	aastore	将栈顶引用型数值存入指定数组的指定索引位置。
0x54	bastore	将栈顶 boolean 或 byte 型数值存入指定数组的指定索引位置。
0x55	castore	将栈顶 char 型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶 short 型数值存入指定数组的指定索引位置。
0x57	pop	将栈顶数值弹出（数值不能是 long 或 double 类型的）。
0x58	pop2	将栈顶的一个（long 或 double 类型的）或两个数值弹出（其它）。
0x59	dup	复制栈顶数值并将复制值压入栈顶。
0x5a	dup_x1	复制栈顶数值并将两个复制值压入栈顶。
0x5b	dup_x2	复制栈顶数值并将三个（或两个）复制值压入栈顶。
0x5c	dup2	复制栈顶一个（long 或 double 类型的)或两个（其它）数值并将复制值压入栈顶。
0x5d	dup2_x1	dup_x1 指令的双倍版本。
0x5e	dup2_x2	dup_x2 指令的双倍版本。
0x5f	swap	将栈最顶端的两个数值互换（数值不能是 long 或 double 类型的）。
0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶。
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶。
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶。
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶。
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶。

0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶。
0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶。
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶。
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶。。
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶。
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶。
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶。
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶。
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶。
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶。
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶。
0x70	irem	将栈顶两 int 型数值作取模运算并将结果压入栈顶。
0x71	lrem	将栈顶两 long 型数值作取模运算并将结果压入栈顶。
0x72	frem	将栈顶两 float 型数值作取模运算并将结果压入栈顶。
0x73	drem	将栈顶两 double 型数值作取模运算并将结果压入栈顶。
0x74	ineg	将栈顶 int 型数值取负并将结果压入栈顶。
0x75	lneg	将栈顶 long 型数值取负并将结果压入栈顶。
0x76	fneg	将栈顶 float 型数值取负并将结果压入栈顶。
0x77	dneg	将栈顶 double 型数值取负并将结果压入栈顶。
0x78	ishl	将 int 型数值左移位指定位数并将结果压入栈顶。
0x79	lshl	将 long 型数值左移位指定位数并将结果压入栈顶。
0x7a	ishr	将 int 型数值右（有符号）移位指定位数并将结果压入栈顶。
0x7b	lshr	将 long 型数值右（有符号）移位指定位数并将结果压入栈顶。
0x7c	iushr	将 int 型数值右（无符号）移位指定位数并将结果压入栈顶。
0x7d	lushr	将 long 型数值右（无符号）移位指定位数并将结果压入栈顶。
0x7e	iand	将栈顶两 int 型数值作“按位与”并将结果压入栈顶。
0x7f	land	将栈顶两 long 型数值作“按位与”并将结果压入栈顶。
0x80	ior	将栈顶两 int 型数值作“按位或”并将结果压入栈顶。

0x81	lor	将栈顶两 long 型数值作“按位或”并将结果压入栈顶。
0x82	ixor	将栈顶两 int 型数值作“按位异或”并将结果压入栈顶。
0x83	lxor	将栈顶两 long 型数值作“按位异或”并将结果压入栈顶。
0x84	iinc	将指定 int 型变量增加指定值。
0x85	i2l	将栈顶 int 型数值强制转换成 long 型数值并将结果压入栈顶。
0x86	i2f	将栈顶 int 型数值强制转换成 float 型数值并将结果压入栈顶。
0x87	i2d	将栈顶 int 型数值强制转换成 double 型数值并将结果压入栈顶。
0x88	l2i	将栈顶 long 型数值强制转换成 int 型数值并将结果压入栈顶。
0x89	l2f	将栈顶 long 型数值强制转换成 float 型数值并将结果压入栈顶。
0x8a	l2d	将栈顶 long 型数值强制转换成 double 型数值并将结果压入栈顶。
0x8b	f2i	将栈顶 float 型数值强制转换成 int 型数值并将结果压入栈顶。
0x8c	f2l	将栈顶 float 型数值强制转换成 long 型数值并将结果压入栈顶。
0x8d	f2d	将栈顶 float 型数值强制转换成 double 型数值并将结果压入栈顶。
0x8e	d2i	将栈顶 double 型数值强制转换成 int 型数值并将结果压入栈顶。
0x8f	d2l	将栈顶 double 型数值强制转换成 long 型数值并将结果压入栈顶。
0x90	d2f	将栈顶 double 型数值强制转换成 float 型数值并将结果压入栈顶。
0x91	i2b	将栈顶 int 型数值强制转换成 byte 型数值并将结果压入栈顶。
0x92	i2c	将栈顶 int 型数值强制转换成 char 型数值并将结果压入栈顶。
0x93	i2s	将栈顶 int 型数值强制转换成 short 型数值并将结果压入栈顶。
0x94	lcmp	比较栈顶两 long 型数值大小，并将结果（1，0，-1）压入栈顶。

0x95	fcmpl	比较栈顶两 float 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为“NaN”时，将-1 压入栈顶。
0x96	fcmpg	比较栈顶两 float 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为“NaN”时，将 1 压入栈顶。
0x97	dcmpl	比较栈顶两 double 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为“NaN”时，将-1 压入栈顶。
0x98	dcmpg	比较栈顶两 double 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为“NaN”时，将 1 压入栈顶。
0x99	ifeq	当栈顶 int 型数值等于 0 时跳转。
0x9a	ifne	当栈顶 int 型数值不等于 0 时跳转。
0x9b	iflt	当栈顶 int 型数值小于 0 时跳转。
0x9c	ifge	当栈顶 int 型数值大于等于 0 时跳转。
0x9d	ifgt	当栈顶 int 型数值大于 0 时跳转。
0x9e	ifle	当栈顶 int 型数值小于等于 0 时跳转。
0x9f	if_icmpeq	比较栈顶两 int 型数值大小，当结果等于 0 时跳转。
0xa0	if_icmpne	比较栈顶两 int 型数值大小，当结果不等于 0 时跳转。
0xa1	if_icmplt	比较栈顶两 int 型数值大小，当结果小于 0 时跳转。
0xa2	if_icmpge	比较栈顶两 int 型数值大小，当结果大于等于 0 时跳转。
0xa3	if_icmpgt	比较栈顶两 int 型数值大小，当结果大于 0 时跳转
0xa4	if_icmple	比较栈顶两 int 型数值大小，当结果小于等于 0 时跳转。
0xa5	if_acmpeq	比较栈顶两引用型数值，当结果相等时跳转。
0xa6	if_acmpne	比较栈顶两引用型数值，当结果不相等时跳转。
0xa7	goto	无条件跳转。
0xa8	jsr	跳转至指定 16 位 offset 位置，并将 jsr 下一条指令地址压入栈顶。
0xa9	ret	返回至局部变量指定的 index 的指令位置（一般与 jsr，jsr_w 联合使用）。
0xaa	tableswitch	用于 switch 条件跳转，case 值连续（可变长度指令）。

0xab	lookupswitch	用于 switch 条件跳转，case 值不连续（可变长度指令）。
0xac	ireturn	从当前方法返回 int。
0xad	lreturn	从当前方法返回 long。
0xae	freturn	从当前方法返回 float。
0xaf	dreturn	从当前方法返回 double。
0xb0	areturn	从当前方法返回对象引用。
0xb1	return	从当前方法返回 void。
0xb2	getstatic	获取指定类的静态域，并将其值压入栈顶。
0xb3	putstatic	为指定的类的静态域赋值。
0xb4	getfield	获取指定类的实例域，并将其值压入栈顶。
0xb5	putfield	为指定的类的实例域赋值。
0xb6	invokevirtual	调用实例方法。
0xb7	invokespecial	调用超类构造方法，实例初始化方法，私有方法。
0xb8	invokestatic	调用静态方法。
0xb9	invokeinterface	调用接口方法。
0xba	invokedynamic	调用动态链接方法[①]。
0xbb	new	创建一个对象，并将其引用值压入栈顶。
0xbc	newarray	创建一个指定原始类型（如 int、float、char……）的数组，并将其引用值压入栈顶。
0xbd	anewarray	创建一个引用型（如类，接口，数组）的数组，并将其引用值压入栈顶。
0xbe	arraylength	获得数组的长度值并压入栈顶。
0xbf	athrow	将栈顶的异常抛出。
0xc0	checkcast	检验类型转换，检验未通过将抛出 ClassCastException。
0xc1	instanceof	检验对象是否是指定的类的实例，如果是将 1 压入栈顶，否则将 0 压入栈顶。

[①] 操作码为 186（0xba）的 invokedynamic 指令是 Java SE 7 中新加入的。

0xc2	monitorenter	获得对象的 monitor，用于同步方法或同步块。
0xc3	monitorexit	释放对象的 monitor，用于同步方法或同步块。
0xc4	wide	扩展访问局部变量表的索引宽度。
0xc5	multianewarray	创建指定类型和指定维度的多维数组（执行该指令时，操作栈中必须包含各维度的长度值），并将其引用值压入栈顶。
0xc6	ifnull	为 null 时跳转。
0xc7	ifnonnull	不为 null 时跳转。
0xc8	goto_w	无条件跳转（宽索引）。
0xc9	jsr_w	跳转至指定 32 位地址偏移量位置，并将 jsr_w 下一条指令地址压入栈顶。
保留指令		
0xca	breakpoint	调试时的断点标志。
0xfe	impdep1	用于在特定硬件中使用的语言后门。
0xff	impdep1	用于在特定硬件中使用的语言后门。