西 南 交 通 大 学

本科毕业设计

Java 反射机制研究与应用

年　　级：2011 级

学　　号：20112799

姓　　名：林从羽

专　　业：软件工程

指导教师：胡晓鹏

二零一五年六月

西南交通大学本科毕业设计

院系 信息科学与技术学院 专 业 软件工程

年级 2011 级 姓 名 林从羽

题目 Java 反射机制研究与应用

指导教师

评 语

指导教师 (签章)

评 阅 人

评 语

评 阅 人 (签章)

成 绩

答辩委员会主任 (签章)

年 月 日

毕业设计任务书

班　　级　软件 5 班　学生姓名　林从羽　学　　号　20112799

发题日期：　2014 年 12 月 17 日　　完成日期：2015 年 6 月 18 日

题　　目　JAVA 反射机制研究与应用

1、本论文的目的、意义

反射（reflection）是 Java 被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过 Reflection APIs 取得任何一个已知名称的 class 的内部信息，包括其 modifiers（诸如 public, static 等等）、superclass（例如 Object）、实现之 interfaces（例如 Serializable），也包括 fields 和 methods 的所有信息，并可于运行时改变 fields 内容或调用 methods。因此，深入研究 Java 反射机制是专业人士 Java 进阶的重要方面。

2、学生应完成的任务

本论文要求分析 JAVA 反射机制的重要方面，并对一些特性进行验证。本论文要求学生完成的工作包括：

1）对 JAVA 反射机制进行分析整理；

2）对反射机制的重要方面进行验证；

3）与其它动态语言的反射机制进行比较。

3、论文各部分内容及时间分配：（共 16 周）

第一部分　基于元数据的软件框架理论学习　(5 周)

第二部分　基于元数据的软件框架实例学习　(4 周)

第三部分　基于元数据的软件框架改进　(4 周)

第四部分　书写论文　(2 周)

评阅及答辩　(1 周)

备　　注

指导教师：_____ 年 月 日

审 批 人：_____ 年 月 日

摘　要

反射（Reflection）是指一个运行时程序具有检查自身信息并基于这些信息改变自身行为的能力。它 1982 年被首次提出，现在已经发展起比较完备的理论体系，并在很多编程语言上有了不同程度的实现。Java 是一门静态语言，要开发具有高通用高拓展性的软件框架，就必须使用反射支持运行时类型发现和自我修改。然而，目前专门针对 Java 反射机制的体系和实现进行讲解的资料并不多，从源码角度来剖析其实现原理的资料也比较少。

本论文尝试从架构体系和源码实现两个方面研究 Java 反射机制。论文首先研究了一般性的反射理论，总结并展示了反射的定义、反射系统的三要素以及反射系统的分类。论文研究了 Java 反射的 API。然后，论文研究了 HotSpot 虚拟机的源码实现，分析了 class 文件和 oop/klass 二分模型的核心数据结构，研究了其对简单连系需求的存取器实现模式，展示了类加载子系统所应用的安全机制等。

本论文主要的工作有两方面。第一，论文研究了 Java 元对象协议中的字段和方法元数据部分，还原了该部分的概念模型，总结了该部分在 Java 元对象协议中的作用和不足。第二，论文研究了反射机制在 HotSpot 虚拟机中具体实现的源码，研究了反射机制工作过程涉及的一些方面的实现细节，包括 oop/klass 二分数据模型的创建时机及具体的数据流向、反射部分的 JNI 实现，以及虚拟机在反射过程对元数据的存取操作等。

关键词：Java 反射； 源码实现； 元对象协议体系； 结构反射； oop/klass 二分模型

Abstract

Reflection is the ability of a running program to introspect on itself and change its behaviour based on what has been found. Reflection is now being fully studied in differece aspects and is implemented in various kinds of systems with a full or partial branch of features since the first time it was brought to computer science in 1982. Reflection is of great importance for the Java programming language to support dynamic runtime-type identification and modification to the system itself in order to provide sufficient infrastructions to extensible, flexible software frameworks development. Materials brought perspectives to the architecture and internal implementation to Java reflection, however, is not as many as what we might expected.

The paper analyse both Java reflection implementation and architecture through a source-code analysing approach. First, general reflection theories and models are studied, the definition and the three basic elements of a reflective system is presented, two kinds of reflection are discussed. The static part of the Java reflection API is studied. Then, the class file and the oop/klass kernel model used to represent Java objects inside the JVM is discussed, the accessor design pattern used to implement the casual connection between the language-level and the meta-level is studied. The class loading sub-system of HotSpot used to guarantee security is discussed.

Two main parts of the work of this paper are present. One is the presentation of the static part of the Java meta-object protocol conceptual model which is reconstructed based on the study to the corresponding part of the Java reflection API. The architecture of this model is shaped. Another experimental approach is the source-code analyzing method used to illustrate the actual implementation for this conceptual, specificational model. Several aspects of the implementation are involved, such as the creation process and the data flow of the kernel oop/klass data strcuture, part of the JNI mechanism implementation that reflection features could have used, the process JVM adopt to realise the meta-level representation access, etc.

Key words: Java reflection; internal implementation; meta-object protocol architecture; structural reflection; oop/klass data model

目　　录

摘　　要 III

ABSTRACT IV

第 1 章 绪 论 1

1.1 课题背景与意义 1

1.2 国内外研究现状 1

1.3 论文主要工作 3

1.4 论文章节安排 4

第 2 章 基本反射模型 5

2.1 反射的定义 5

2.2 反射系统的基本特征 5

2.2.1 自描述 6

2.2.2 简单连系 6

2.2.3 安全性 6

2.3 反射的种类 7

2.3.1 结构反射 7

2.3.2 计算反射 8

第 3 章 Java 反射模型 10

3.1 面向对象的 Java 反射模型 10

3.1.1 类和对象 10

3.1.2 元类和元对象循环 11

3.2 元数据组织 12

3.2.1 class 文件 12

3.2.2 oop/klass 二分模型 12

3.3 简单连系 13

3.3.1 语言层的简单连系 13

3.3.2 虚拟机层的简单连系 14

3.4 Java 元对象协议 15

3.4.1 面向对象反射模型的起源 15

3.4.2 三大核心反射对象——对象、方法和构造方法 17

3.4.3 注解支持 17

3.4.4 静态信息获取 17
3.4.5 动态存取和动态调用 19
第 4 章 Java 反射机制分析 23
4.1 OpenJDK 源代码结构 23
4.2 Java 核心反射对象体系 25
4.2.1 注解标记接口 25
4.2.2 权限控制中心 25
4.2.3 成员标记接口 25
4.2.4 字段对象 25
4.2.5 方法对象 25
4.2.6 构造方法对象 26
4.3 Java 标准类库——水面上的冰山 26
4.3.1 元信息存储及其 API 26
4.3.2 数据注入 27
4.3.3 元数据操作及其 API 28
4.3.4 简单连系的实现 29
4.4 标准类库的内部实现 30
4.4.1 安全与反射工厂 30
4.4.2 存取器架构设计与实现 33
4.4.3 类文件组装模块 36
4.5 元数据仓库——class 文件 36
4.5.1 class 文件结构 37
4.5.2 常量池 39
4.5.3 字段表与方法表 42
4.6 反射部分的 JNI 机制 42
4.6.1 JNI 机制与本地方法概述 43
4.6.2 本地方法位置 43
4.6.3 JNI 中枢——JNIEnv 结构 44
4.6.4 本地方法注册 46
4.6.5 JNI 数据类型转换 47
4.6.6 JNI 相关的宏定义 48
4.7 数据结构的核心——oop/klass 二分模型 49
4.8 一个完整反射调用的实现 50
4.8.1 JVM 实现相关的宏定义 51

4.8.2 简单封装——句柄设计模式 53
4.8.3 字段迭代器和信息容器——JavaFieldStream 和 FieldInfo 54
4.9 本章小结 56
结　　论 57
致　　谢 58
参考文献 59

第1章　绪 论

1.1 课题背景与意义

Java 自 1995 年正式推出 1.0 版以来，凭借其跨平台及在 WEB 开发和并发编程方面的优势已经吸引了越来越多的开发人员，成为了使用人数最多的一门编程语言[1]。Java 标准平台提供了众多服务，如 JDBC（为多种关系数据库提供统一访问接口）、RMI（提供分布式程序开发服务）、EJB（支持企业级应用部署管理）等[2]，它们使得开发者能够快速高效地开发并发布符合客户要求的企业级应用程序。与此同时，Java 的开源社区也不断发展并涌现出了很多优秀的软件框架用以提高软件开发的生产率、减少开发过程中的重复劳动，如 Spring、Hibernate、JUnit、JFinal、Mockito 等框架。

软件框架通过抽象软件开发过程中的典型过程或切面来提供通用功能，如 Spring MVC 抽象并提供了权限验证机制和请求分发功能，使开发者能专注于业务逻辑的开发；JUnit 为开发者提供了简明的测试方法等。这对软件框架的拓展性和通用性提出了很高的要求。而 Java 是一种编译型的静态语言，Java 程序在执行前必须被编译成平台无关的字节码，并通过尽可能多的静态类型检查，这又大大限制了框架开发过程对灵活性和通用性的要求。在这种情况下，反射被大量地应用于框架开发中，使框架能够动态地根据开发者的需求进行拓展。

反射（Reflection）是指一个运行时程序具有检查自身信息并基于这些信息改变自身行为的能力[3]。这个概念在 1982 年首次被引入计算机编程领域，发展至今已经具备比较完备的理论体系，也在很多编程语言上有了不同程度的实现，如 Smalltalk[4]、C#[5]、Java[6]等。

Java 反射机制允许开发者动态加载运行时才能发现的类或接口，允许用户运行时检查并修改字段的值（甚至允许修改私有字段），允许用户在运行时动态调用方法等。Java 反射机制为开发人员提供了开发具有高拓展性和高通用性软件框架的强有力工具，同时也被用于可视化编程环境和调试追踪工具的开发中，应用场合非常丰富。但 Java 反射机制的过度使用也可能降低代码的性能和可读性，造成调试的困难等。因此，编程人员有必要深入学习 Java 反射机制。

1.2 国内外研究现状

反射这个概念是 1982 年 B.C.Smith 在其博士论文中首次提出的[7]，文献指出一个具有反射能力的系统应该具备三个基本特性：一是系统必须持有一份关于自身的信息描述；二是系统与其描述之间必须能方便地进行连接交互；三是系统必须能安全地执行反射任务。这三个特性至今仍然是所有反射系统设计的重要依据。

1984 年，Daniel P.Friedman 和 Mitchell Wand 针对 B.C.Smith 博士提到的连接和安全问题提出分层观点，并给出了在 CLOS 系统下的具体实现[8]。这个元层和基本层分离的方案，引出了在元层“开放实现”的观念，也成为很多语言和系统实现反射时采用的模型，如 SOM、SmallTalk、CLOS、Java 等。

1987 年，Pattie Maes 归纳总结了计算反射的定义，指出了其实践意义和重要性，并研究了现有的一些部分地实现了反射体系的编程语言实践[9]。P.Maes 指出，现有系统对反射体系支持并不完整。论文在面向对象的编程语言 KRS 中引入反射体系并给出了这个实现的尝试[10]。

1989 年，Jacques Ferber 给出了计算反射的两种实现模型，讨论了结构反射与计算反射的区别，展示了等价的描述反射系统的反射方程，提出元循环解释器在实现反射系统方面的优越性[11]。J.Ferber 给出并讨论了 B.C.Smith 提出的反射系统三要素在面向对象语言中的具体表述，展示了三种不同的反射系统实现模型及其优劣，是反射领域很有价值的综述资料。

1989 年，Ira R.Forman 和 Scott H.Danforth 描述了他们在 IBM 的 SOM ToolKit 3.0 中实现的一个反射模型，详细讨论了构建一个反射系统涉及的静态层面和动态层面。静态层面涉及存储数据的数据表、方法表、继承体系和元类循环等问题，动态层面涉及方法拦截、动态调用、方法解析次序、元类继承约束等问题[12]。此外，这两位作者还发表了关于这个模型的一些论文，文献[13]是一个概述。他们描述了 SOM 中的元类继承问题[14]，分析并解决了元类系统构建中的元类继承约束问题[15]，描述了元类中的 Before/After 元类的组合和应用[16]。

1991 年，George Kiczales 介绍了 CLOS 系统的元对象协议的设计和实现[17]。

1996 年，Stanley B.Lippman 出版了一本介绍 C++对象模型的书，虽然 C++仅有 RTTI（RunTime Type Identification, 运行时类型识别）的能力，严格而言并非完整的反射，但这个对象模型是研究基于类和对象的反射模型的基础[18]。特别是 HotSpot 虚拟机主要是使用 C++实现的，因此对于 C++对象模型的研究很有必要。

Ira R.Forman 的[3]是首部详细介绍 Java 反射的书籍，涉及 Java 反射的各个方面，包括使用场景、字段反射、方法反射、动态调用、异常链信息反射、类加载机制、设计模式、性能分析以及反射的发展与未来等，其附录则给出了反射的一般模型和元对象协议在 Java 反射机制中的体现。本书的其中一位作者参与过 IBM 公司的 SOM(System Object Model, 系统对象模型) Toolkit 框架的实现，对反射理论了解深刻，但书中并未过多涉及 Java 反射的内部机制。

Oracle 公司有两份研究反射必须参考的官方文档：[19]对 Java 语言的语义进行了细致的说明，但其中并不包含任何与反射实现相关的描述；[20]对 class 文件和虚拟机的架构和指令集做了规格性的说明，是 Java 反射所采用的概念数据模型及规范。[21]

是一份关于 Java 本地接口的规范和细致的使用说明。

国内方面值得注意的有[22]和[23]两篇论文的研究。前者提出可以通过在系统设计阶段区分系统的功能性需求和非功能性需求的方法来设计具有反射能力的系统，并对元层技术、元对象协议、反射理论和反射的分类等做出了归纳和总结；后者提出了一个分布式元对象协议的实现，并指出在元层开放实现与在基本层封装变化的思想一脉相承。

在 HotSpot 虚拟机方面的研究也有不少进展，周志明对虚拟机的基本架构和运行调优进行了深入研究[24]，陈涛对虚拟机内部的 oop/klass 二分模型、栈式指令集等 HotSpot 虚拟机实现的各个层面都有过深入探讨[25]，Oracle 前 JVM 编译器团队工程师莫枢也对 JVM 各个模块（包括语言处理器、即时编译器等）的实现有过分享[26]。此外，侯捷在 2004 年也发表过一篇 Java 反射机制应用的概述性文章[27]。

综上所述，在反射的理论研究及其实现模型方面，国外已经有了比较全面深入的研究。但具体到 Java 的反射机制，国内外的资料则多是从如何使用的角度进行介绍，提及其底层实现的资料比较稀少，在此基础上介绍其体系和架构的资料也不多。因此，本论文将从具体实现和体系架构两个方面对 Java 反射进行研究。

1.3 论文主要工作

本论文工作思路是通过理论和实现两方面来研究 Java 反射机制。通过研究反射理论和反射的基本模型，论文提供了研究 Java 反射的高层视角。但理论模型的实现方法可能与模型规范本身有较大的差别，仅通过理论研究不容易解答 Java 反射机制设计与实现上的一些实际问题。因此，论文还将进一步从源代码的实现层面研究 Java 反射机制。

同时，Java 语言是运行在虚拟机上面的，要了解反射机制的底层实现，就必须深入地研究虚拟机相关的实现代码。理论上，任意一台实现了 Java 虚拟机规范的虚拟机都应该能正确地解释执行 Java 程序，如 HotSpot、JRockit、J9 等虚拟机[28]。但本论文选择的研究对象是 OpenJDK 项目下的 HotSpot 虚拟机，一方面因为它是目前最主要的 Java 虚拟机实现，另一方面它是开源的，其源代码很容易获得，有利于本论文进行有针对性的研究，使得出的结论更具有说服力。

综上所述，本论文的主要工作有两个个方面：

(1) 研究基本的反射模型。论文将给出反射的定义，研究一个基本的反射模型应该具有的三个基本特征，区分两种反射模型并介绍了它们各自的优缺点。

(2) 研究 Java 反射模型。基于一般性的反射模型，论文从体系和实现两个方面对 Java 反射体系进行了研究。体系方面，论文考察了 Java 反射模型在元数据组织、元层与基本层联系等方面的处理，研究了 Java 的元对象协议覆盖的层面及其优缺点；实现

方面，论文研究了 Java 反射机制所涉及的多个方面，包括标准类库的公有实现和本地实现、类加载机制、JVM 的内部实现以及关键数据结构 oop/klass 二分模型等。

1.4 论文章节安排

论文共分 5 个部分，分别是绪论、基本反射模型、Java 反射模型、Java 反射机制实现、总结。

第 1 章介绍了本课题提出的背景和意义、国内外的研究现状、论文的主要工作和章节安排等。

第 2 章介绍了一个基本的反射模型应该具备的基本特征，区分了反射模型的两种分类及其相应的优缺点等。

第 3 章介绍了 Java 的反射模型对反射系统应该具备的三个特征的实现，从这三个特征出发分析了 Java 元对象协议。

第 4 章研究了 Java 反射机制的实现，论文以 HotSpot 虚拟机为研究对象，研究了反射实现各个层面的问题。这部分是论文的核心工作。

最后一部分是结论，总结了本文的主要工作、期望与不足。

第2章　基本反射模型

本章是对基本反射模型的概述。2.1 首先给出一个关于反射的定义；2.2 节讨论反射系统的三个基本特征，它们是任何一个反射系统设计的过程中都必须考虑的问题；2.3 节区分了反射模型的两种分类，并给出了其对应的实现模型。

2.1 反射的定义

反射是指一个运行时程序具有检查自身信息并基于这些信息改变自身行为的能力[3]。有其他学者提出过其他类似的定义，但主要都包含了以下三个重要的方面：运行时程序（Runtime program）、自我检查（Introspection）行为改变（Behaviour change）。

对于一个具备反射能力的系统来说，它首先必须能够进行自我检查。自我检查指的是程序具有获取关于自身信息的能力。比如一个 Java 类可以获知自己（甚至其他的类）拥有的方法、字段以及它们的运行时值等信息。这不是一个程序与生俱来的能力，需要额外的支持。比如一个 C++的类运行起来后就无法知道自己定义了什么字段或者方法，因为 C++并不具备完整的反射体系。

其次，程序必须可以基于这些信息改变自己的行为。就好比一个人，他可以通过照镜子发现自己打的领带不太适合今天的会议场合（获知自身的信息）从而换上一条更适合的领带（基于信息改变自身）。在程序设计中，这种行为改变可能表现为方法执行前的检查、统计、代理等。

2.2 反射系统的基本特征

B.C.Smith 博士在他的博士论文中提到，一个系统要具备反射的能力——也即想要可以进行自我检查和自我改变——至少必须具备以下三个条件[7]：

(1) 系统必须拥有一份自描述；

(2) 系统与其自描述之间必须有简单的连系；

(3) 系统必须能够安全地执行其反射任务。

Smith 博士提到的第一个需求是最基本的要求，如果一个系统没有拥有一份自描述，那“自我检查”更无从谈起。其次，系统与其自描述之间必须能够简单、有效地沟通，这意味着系统对其自描述的改变将被反映到系统上，并引起系统自身相应的改变。最后，由于系统具备了自我修改的能力，这要求系统必须能够在一个“安全点”执行反射操作，而不致引起系统的不一致甚至崩溃。这三个需求之间是层层递进的联系。

2.2.1 自描述

自描述（Self-representation）是一个反射系统必须具备的最基本组成。如果一个系统要获取关于自身的信息，它至少必须拥有这份信息。但具体来说，关于这份自描述必须存储什么信息，或必须以什么形式来组织则并没有强制约束。

为了实现一个完整的反射系统，系统的自描述应该是完整且方便获取的。完整即意味着这份自描述能够描述系统涉及的所有层面，如果一份自描述不够完整，那么它缺失的系统部分的描述就不能被外界获取到；而方便则意味着系统能够方便地从自描述中取得关于系统的信息。比如，如果用文本文件来描述一个程序就不够完整，因为它不具备完整的特征，它没有办法描述程序运行时变量的值，方法被调用的次数等信息；同时它也不够方便，因为它只是程序的一个字符串，想要从中提取有用的信息还必须经过编译器把它编译成中间文件。

关于如何更有效地组织信息来描述系统，其实是人工智能（Artificial Intelligence, AI）领域的一个核心概念——知识描述（Knowledge representation）[31]。这已经超越本文讨论的范围，本论文讨论的“知识”局限于有关计算机系统自身的信息。

2.2.2 简单连系

如果一个系统与它的自描述之间存在如下描述的联系，则称它们之间存在简单连系（Casual connection）：系统自描述的改变将引起系统相应的改变，且反之亦然，即系统的改变也将引起其自描述的改变。

简单连系在不同的反射模型中有不同的实现难度，这跟系统的自描述以什么形式存在有关系。如果一个反射系统原原本本地使用实现系统自身的数据来充当自描述，其简单连系的需求是自动被满足的，因为系统的自描述就是用来实现系统的数据本身，两者必然时刻保持一致。但在不使用系统实现数据来直接充当自描述的反射系统中，简单连系的要求就没那么容易满足了，系统同时必须维护其与自描述之间的一致性。

简单连系描述的是系统及其自描述数据之间的关系，此时一个系统的自描述本身就相当于系统的一个副本，它提供了有关系统的所有信息，同时其变化也会被反映到正在运行的系统上。这赋予了程序以修改自身的能力——正是一个反射系统所需要的能力——但同时也提出了安全问题。有些反射代码具备很高的权限，它可能被允许获取甚至修改系统关键部分的数据，这种权限如果被恶意代码持有，很可能会破坏系统的一致性，甚至带来不可恢复的伤害。因此，如何保证一个反射系统的安全，使其在执行反射操作的时候不至于破坏系统，便是 Smith 博士所提出的第三个考虑。

2.2.3 安全性

安全性（Security）指的是系统必须在一个安全的位置执行反射操作，以避免修改

系统的时候引起系统的不一致甚至导致系统崩溃，必要的时候可能还需要对反射操作的权限进行检查。

Daniel P.Friedman 和 Mitchell Wand 提出了一种分层的观点来实现反射系统的安全控制，他们主张将一个反射系统划分为基本层和元层，基本层的职责在于描述外部的问题域，而元层的职责在于维护基本层和系统的信息、执行关于基本层的计算以及维护基本层与系统自描述间的一致性等[8]。它们之间通过 reflect(反射)和 reify(具体化)两种操作来互相联系。其分层结构如图 2-1 所示。

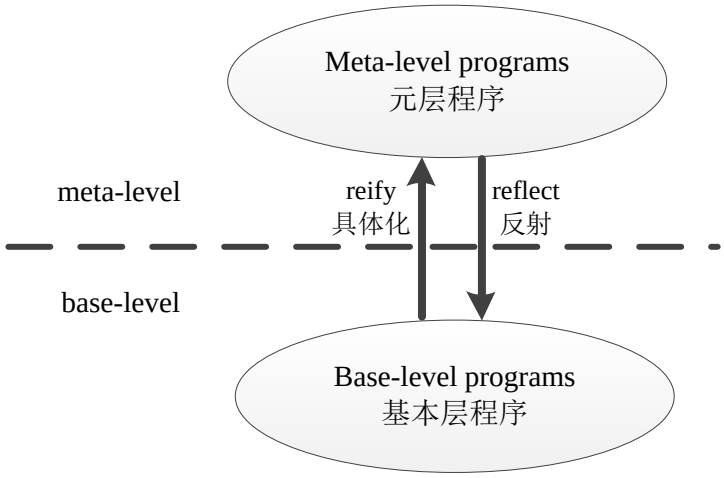

图 2-1 Friedman 和 Wand 提出的运行时反射系统分层模型

这个分层思想在很多反射系统中都得到了应用和实现，如 SmallTalk[32]、CLOS[8]、Java 等，而且元层和基本层的分离在面向对象的反射领域催生了元对象的提出和使用，进一步促成了描述系统问题域的基本层对象与描述基本层的元层对象的分离。

2.3 反射的种类

2.1 节和 2.2 节分别介绍了反射的定义及其实现要素。本节将在这个基础上根据自描述所描述的系统的不同部分，进一步区分反射的两种类型：结构反射（Structural reflection）和计算反射（Computational reflection），并讨论这两种反射可能的实现模型及优劣。结构反射指的是基于系统静态层面信息的获取，比如系统的数据类型、字段和方法信息（如字段类型、方法签名）等；计算反射指的是基于系统动态层面的行为改变，比如方法拦截、动态调用等。

2.3.1 结构反射

Jacques Ferber 指出，结构反射更多是对系统静态层面的反射，比如数据结构、数据类型等[11]。Pierre Cointe 对结构反射模型的实现具有参考意义，该模型中采用对象的元类来持有元数据[51]，可以非常完整地表示对象的结构信息。该模型的工作原理如图 2-2 所示。

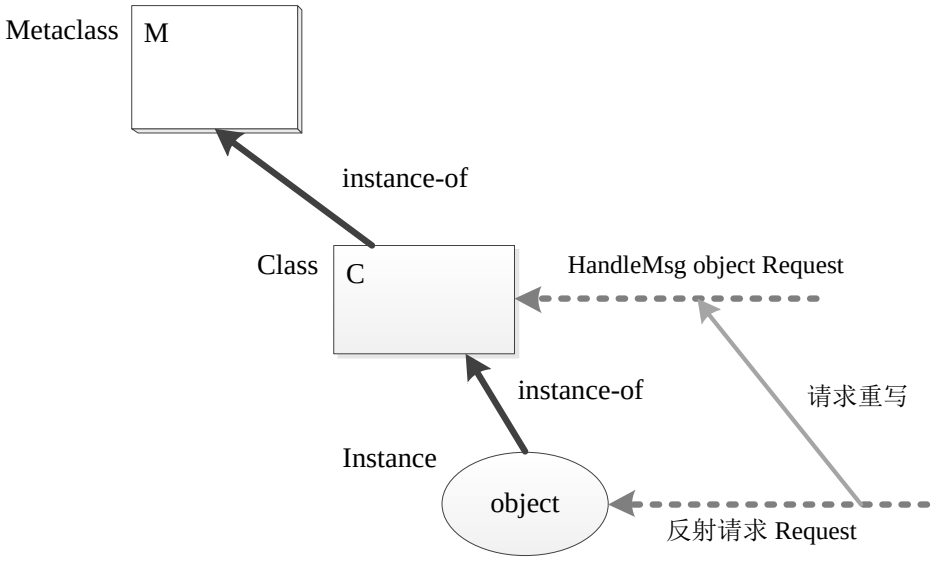

图 2-2 P.Cointe 提出的结构反射实现模型

在这个模型中，系统层面的元数据都存放在对象的类中。当对象接到一个请求（比如函数调用）时，如果需要反射计算来完成对请求的响应，那么它可以将请求发送到其声明类上去。对象的声明类同时也必须有一个方法（如图 2-2 中的 HandleMsg 方法）来处理反射计算，因此，这个方法必须在元类中定义。

上述模型可以比较完整地描述反射系统的结构信息，然而它也有一些（甚至是比较致命的）限制。其一，这个模型使得为单个对象定制解释器变得不可能，因为处理反射计算的所有方法都必须统一地定义在元类中；其二，该模型使得任何为对象保存性能统计数据或方法调用信息的想法都变得不可能，而针对特定对象保存相关统计信息又是现实的需求。

归根结底，还是这个结构反射模型职责模糊所致，它没有区分数据表示和反射调用两个方面的需求，统一使用元类来满足。因此，有另外的反射模型实现被提出，将这两个不同层面的需求分离出来，这种分离又促使了另一种反射类型——计算反射的诞生。

2.3.2 计算反射

计算反射指的是由一个反射系统完成的关于计算的计算的行为[9]。反射系统指的是一个拥有自身的自描述并存在简单连系的系统。而关于计算的计算，指的是反射系统执行的计算，它是针对自身对问题域的计算而进行的计算。对问题域的计算通常称为对象计算，后者称为反射计算。

由于结构反射无法满足更为复杂的现实需求，人们在分离数据表示和信息计算两个关注面的同时逐渐发现了对系统行为进行描述的重要性，分离出来的反射计算与系统行为结合起来就是典型的计算反射。计算反射可以有多种实现的模型。P.Maes 描述

过一个具有代表性的计算反射的实现模型，见图 2-3。

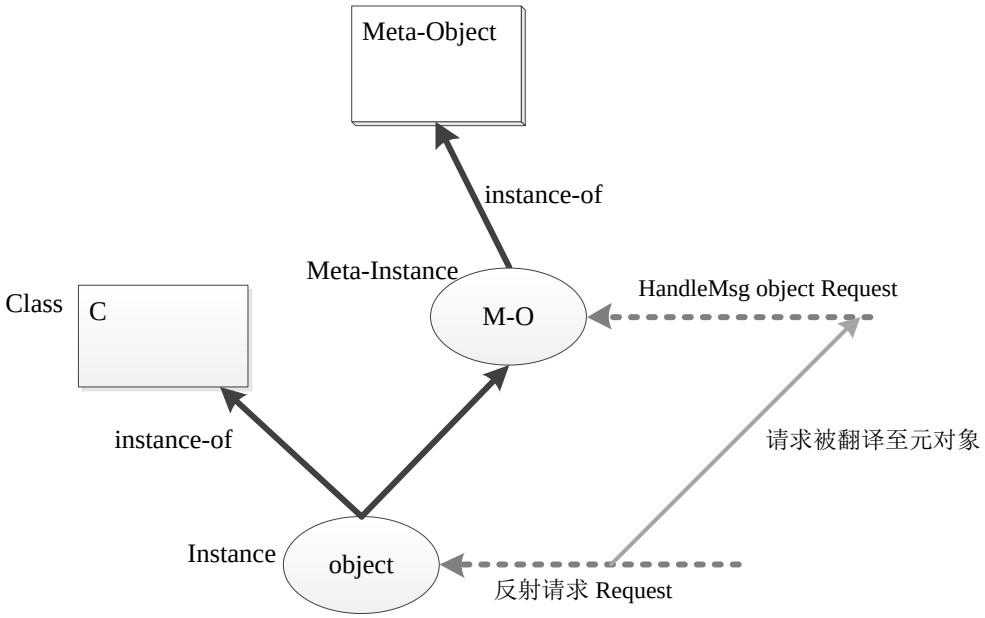

图 2-3 P.Maes 提出的计算反射实现模型

计算反射的优点在于分离了信息存储和行为。系统元数据的存储依然由对象的类来存储，但系统行为方面的信息则转移到元对象上。通常，计算反射系统可以拥有一个根 META-OBJECT 类，它是所有元对象的基类，可以派生出多个元对象，负责系统多个行为层面的表达。元对象可以临时地附着到对象上，执行一些功能，比如在方法调用前后输出调试信息、监控方法执行、统计性能数据等[9]。常用的功能还可以编写为系统常驻的元对象，提供给所有对象使用。

第3章　Java 反射模型

本章将从自我描述、简单连系和安全性三个方面入手来考察 Java 的反射模型。3.1 节是对 Java 反射模型的概述；3.2 节考察了 Java 对元数据的组织；3.3 节考察了自描述与 Java 系统的简单连系；3.4 节总结并给出了整个 Java 反射体系的架构；3.5 节总结了 3.4 节叙述的 Java 元对象协议的优点和不足。

3.1 面向对象的 Java 反射模型

Java 是一门纯面向对象的语言，系统中的一切都必须以对象的形式存在。每个对象都是一个类的实例，由此将产生一个 3.1.1 节所述的类和对象的层次图。同时，由于系统中存在的一切都是对象，因此类也是一个对象，也必须是另一个类的实例，如此继续将产生一个无限循环塔（infinite tower），这个问题将在 3.1.2 节讲解。

3.1.1 类和对象

在 Java 中，Object 类是所有对象的基类，是类体系的顶点[19]，所有未声明基类的类都默认继承于 Object 类。Object 类及其子类的实例就是实例对象。关于类和对象详细的讨论和定义，读者可参考文献[12]及[36]。Java 中类和对象的层次如图 3-1 所示。

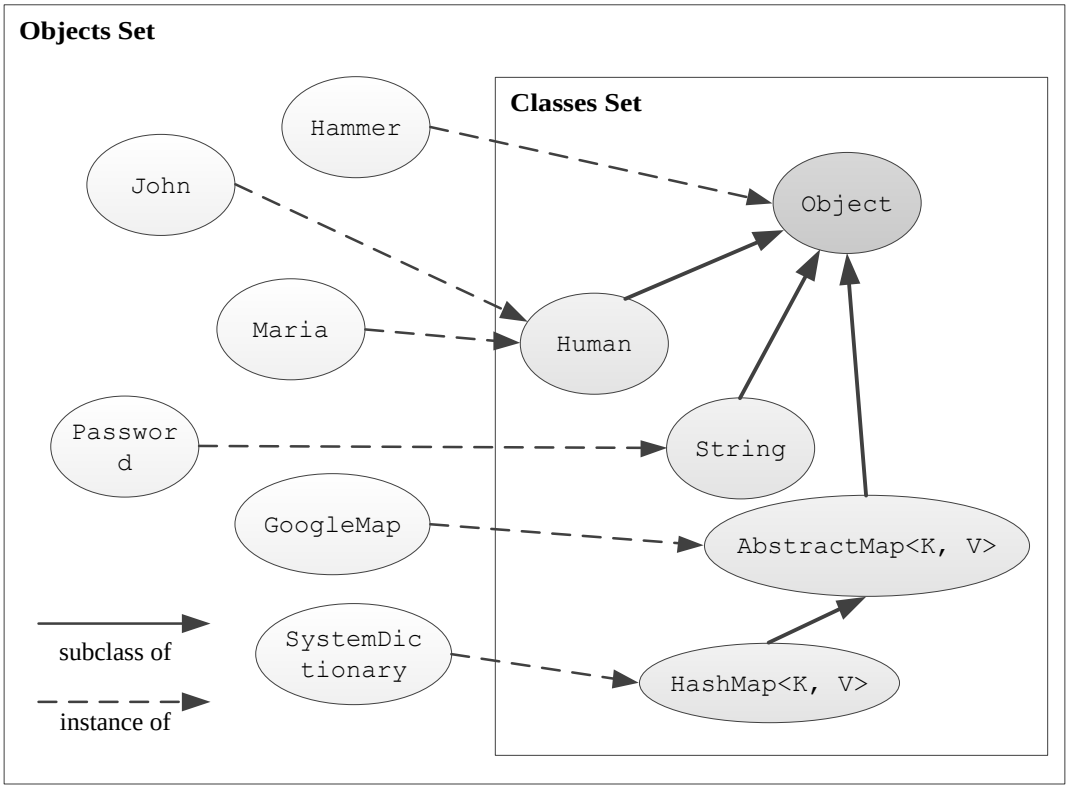

图 3-1 Java 语言的类和对象层次图

3.1.2 元类和元对象循环

根据 3.1.1 面向对象的理论可以得知，每个对象都有一个用于描述它的类，而系统中每个元素都是对象，因此，类也是对象。那么类也需要拥有一个用于描述它的类。这种用于描述类的类，称为元类（meta-class）。其实例对象即用来描述一个类，称为元对象（meta-object）。

如此会引出一个问题：元类也是对象，也需要有用来描述它的类，同理这个类也需要有一个用来描述它的类……如此即会导致无限循环的定义。怎么解决这个问题呢？总结起来，有三种比较常见的方法：

(1) 把元类和元对象移到对象体系之外，即它们不属于对象体系中一般意义上的“对象”[35][36]；

(2) 根据实际系统的需要只产生足够的元类，而非无限地生成[37]；

(3) 在类体系结构中引入循环，使有一个最终的元类，描述它的类即是它自身。

实际上，Java 的反射模型采用的正是最后一种方法。在 Java 系统中只存在一个元类，即是 Class<?>类。同时由于 Java 不支持多继承，在其元类体系中也不存在所谓的元类不兼容问题[12]。加入了元类 Class 后 Java 就拥有了一个封闭完整的类和对象层次体系，如图 3-2 所示。

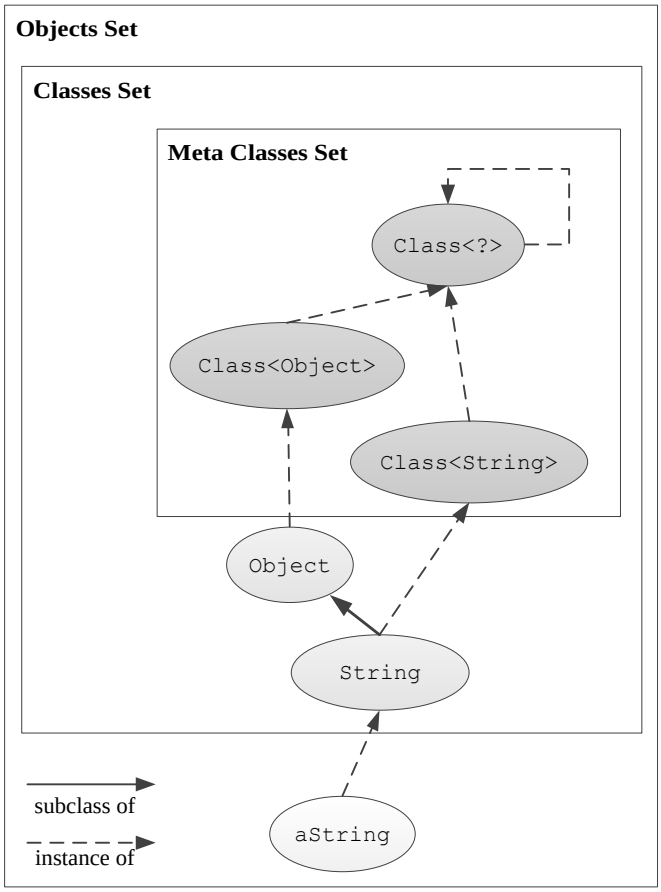

图 3-2 加入元类后 Java 完整的类-对象层次

从上图可以看出，Object 类及其子类 String 等都是类 Class 的一个实例。在对象体系的最上层，Class 对象是其自身的一个实例。同时在系统中可以存在不同的 Class 元对象，用以描述各个类，但所有元对象都是 Class 类的实例，Class 类又是其自身的实例，如此完成了一个封闭且完整的类-对象体系。

3.2 元数据组织

2.2 节提到，一个反射系统最基本的组成便是描述系统的元数据，对于 Java 语言来说也不例外。一个 Java 程序乃至系统可能涉及诸多元素，比如变量、字段、方法、类、接口、字符串、注解等。如此繁多且层次各异的元信息，Java 采用何种形式来完整地表示它们？如何平衡它们的存储效率？

实现上，HotSpot 虚拟机采用了两级存储策略来保证这些元数据能被完整地保存下来并且在运行时方便地参与系统的描述。这个策略根据程序所处时期的不同，使用了两种不同的数据结构来保存程序的元信息，分别是编译期的 class 文件，以及运行时的 oop/klass 二分模型。

3.2.1 class 文件

Java 是一门静态的编译型语言，Java 程序在执行之前都要被编译期编译成一个平台无关的 class 文件，它可以被任意一个实现了 Java 虚拟机规范的虚拟机解释执行。class 文件的结构由虚拟机规范所定义[20]。论文 4.5.1 节将简要介绍 class 文件的结构，本节仅简单介绍该 class 文件是如何完整方便地存储系统的元数据的。

为了最大限度地利用每一个数据位，class 文件被设计为一个由 8 个 bit 为单位的字节流组成的文件，字节与字节之间没有多余的数据填充，每一位数据都是有意义的。这个文件存储了程序被编译后留存下来的所有信息，包括访问标志、类（或接口）名及其基类、字段个数及类型、方法个数及类型、方法的字节码、注解、常量池等许多信息。每项信息都以特定的数据结构存储，紧密地排列在 class 文件中。

在编译过程完成以后，一个类或接口的所有元信息就保留在其对应的 class 文件中。在程序运行后并且类被初次加载时，虚拟机就会读入这个 class 文件的信息，创建相应的运行时常量池和数据结构，从而保证了元信息能够在运行时被程序所使用。

3.2.2 oop/klass 二分模型

3.2.1 节描述了编译期用来存储数据信息的 class 文件，在其被虚拟机加载链接之后，它又是如何参与描述系统结构的呢？在 HotSpot 中，其设计者设计了一套 oop/klass 二分对象系统来表示 Java 层的对象。其中 oop（Ordinary object pointer, 普通对象指针。由于它是 HotSpot 虚拟机内部使用的数据结构的名字，后文中均用小写的 oop 来表示。

klass 也同）用来表示一个 Java 层的对象，klass 部分用来描述一个 Java 层的类[33]。前者（及其子类）主要职能在于表示对象的实例数据，不持有任何虚函数。相应的，动态的函数分发功能被移到描述 Java 类的 Klass 体系中。这符合论文在第 2.3 节中关于一般反射模型的理论，即通过不同的元对象来描述对象的特征，以实现基本层和元层的隔离。

虚拟机层面的 oop/klass 二分模型完整地描述了 Java 系统的所有元数据。当用户调用一个 new 操作符来新建一个对象时，一般的流程是这样的：这个类会首先被虚拟机加载并链接（假定它还没有被加载过），它的 class 文件会被虚拟机读取，此时虚拟机会在方法区上创建一些 klass 类或它的子类来存储 class 文件中的元数据。接着使用这个类的元数据在 Java 堆上创建一些 oop 对象，这些对象用来描述一个 Java 层对象的实例数据。这样一个 Java 对象在虚拟机内部就算是创建完毕了，其所属的类及其本身的元数据已经被记录在虚拟机内部以便支持进一步的反射操作了。

在数据存储的效率上，虚拟机内部使用的多是指针来指向真实的实例数据或是使用精心安排过的对象头（如 oop 对象的_mark 字段）来最大限度地利用每一个二进制位的容量。由于一个指针采用的是平台的机器字长来存储，在 64 位的机器上要比在 32 位的机器上多花大约 1.5 倍的内存来存储 64 位的指针。为此，虚拟机提供了一个虚拟机选项 UseCompressedOops 以支持指针压缩，以便使虚拟机在 64 位机器上也拥有良好的性能[34]。

3.3 简单连系

论文对简单联系的考察将从以下几个方面展开：

(1) 是否存在基本层和元层的分离？

(2) 基本层和元层之间是否存在联系的方法？

(3) 对元层的修改是否能被反映到基本层？反之是否亦然？

特别地，在 Java 的官方实现中，它是由底层的虚拟机来支撑的，系统最终的元数据描述都存放在虚拟机层，但用户对 Java 反射的操作则都必须通过语言层的对象来完成。因此，有必要对 Java 语言层和虚拟机层的简单联系都进行研究。

3.3.1 语言层的简单连系

毫无疑问，在 Java 的语言层存在元层与基本层的分离，这在 3.1 节面向对象的反射模型中已经展示过。元层即是 Class 类及其实例对象所在的层，基本层即是所有元层以外类和对象所在的层。那么在 Java 语言层元层与基本层之间是否存在联系的方式？也即，在基本层的类和实例对象是否能得到描述其自身的 Class 类实例？答案是肯定的。在 Object 类的官方 API 文档中存在这样一个方法：public Class<?> getClass()，通

过这个方法一个对象就可以在运行时获得描述其自身的元对象了。其简单连系如图 3-3 所示。

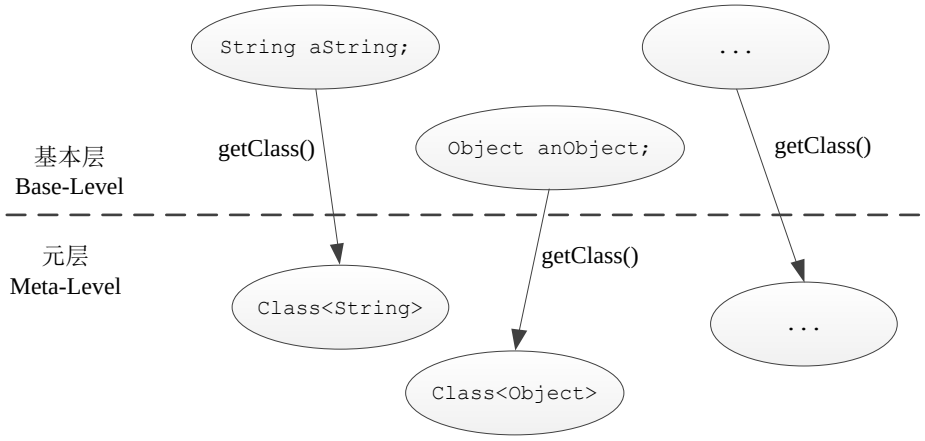

图 3-3 语言层的简单连系

3.3.2 虚拟机层的简单连系

3.2 节论文介绍了虚拟机内部用来表示 Java 层对象的 oop/klass 二分模型，其中 oop 部分被用来描述一个 Java 对象的实例数据，klass 部分存储着 Java 类的元数据。在虚拟机层，oop 体系所处的是基本层，而相应的描述元数据的 klass 体系所处的则是元层。这是虚拟机内部元层和基本层的划分，一个系统对象如果想要拿到它的自描述，需要经过两步，即 Java 对象指针->oop->klass，其概念模型如图 3-4 所示。

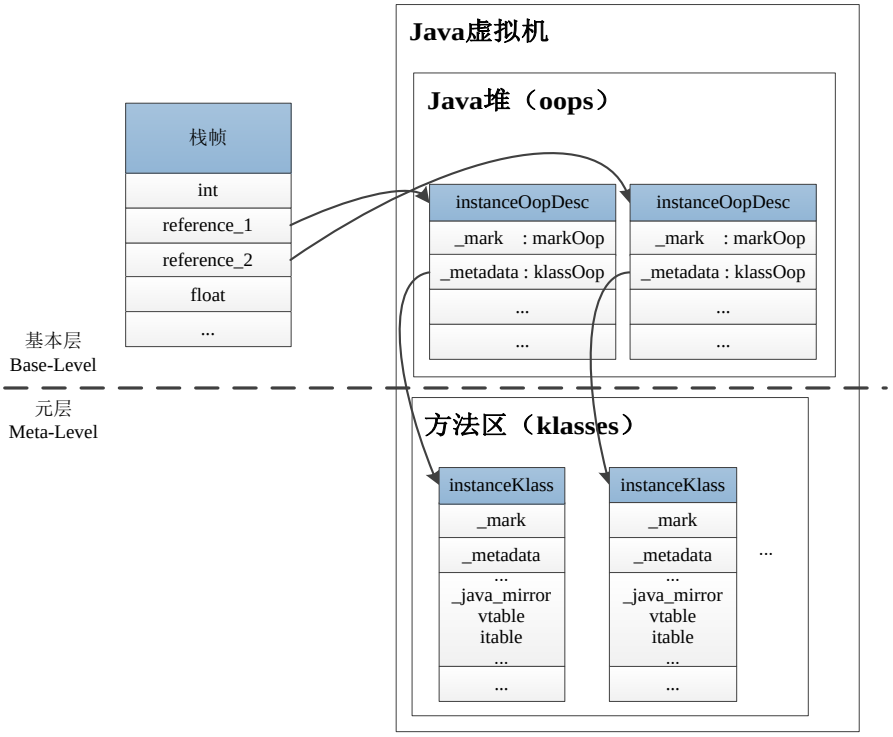

图 3-4 虚拟机层的简单连系

3.4 Java 元对象协议

由 3.1.2 节所述，元对象是一个用于描述类的实例对象。而“协议”可以理解为一个接口，即系统提供给外界使用元对象及其相关功能的接口。Java 提供的所有与元对象相关功能的总和，称为 Java 的元对象协议（Metaobject Protocol，下文简称 MOP）。

Java 用于支持反射的核心包及其相关作用在表 3-1 中给出。除此之外还有一些包下的成员也提供了支持反射所必需的功能，如 java.lang 包下一些基本数据类型的包装类（Integer、Double 等），以及 java.security 包提供的安全检查支持等。这些核心包和辅助包提供了基本的反射支持、泛型、注解、动态代理等服务，它们的总和构成了 Java 的元对象协议。

本节将从反射体系的角度研究 Java 的元对象协议，具体到从 API 和源码角度的研究将在 4.2.1 节～4.2.3 节给出。

表 3-1 支持 Java 反射的核心包及其作用

包	描述
java.lang	提供了面向对象反射系统的两个根类：Object 和 Class，以及相关的类加载功能。通过 Class 对象可以获取系统运行时几乎所有层面相关的元数据
java.lang.reflect	提供了基本反射元素的支持（如字段、方法、构造方法等元素的信息获取）、动态代理的支持等
java.lang.annotation	提供了注解支持。

3.4.1 面向对象反射模型的起源

由 3.1 节的讨论，一个基于面向对象的反射系统至少必须要有两个类：Class 和 Object。Java 是纯面向对象的语言，系统中的一切都是对象，与此同时，每个对象都拥有一个描述它的类，这个类作为模板存放着建造该对象所必须的所有信息。在 Java 中，一个对象是由一个 Object 类或其子类的实例来表示的，而类由 Class 类的实例来表示。这两个类的关系如图 3-5 所示。

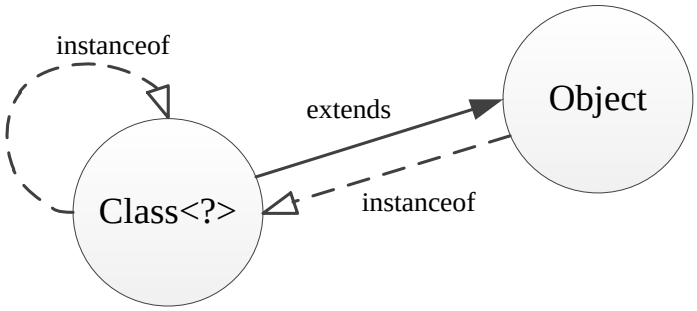

图 3-5 Class 对象和 Object 对象的关系

Object 类是所有 Java 中所有对象的基类，其官方的 API 文档中指出它有一个方法 getClass()，这个方法可以获得一个描述该对象的元对象。Class 类的实例用来描述其他的类，同时它又是一个对象。在不引起混淆的情况下，下文称一个 Class 类为元类，称一个 Class 类的实例为元对象。通过上面的 getClass()方法得到的元对象，是所有反射调用的起点。

官方 API 文档中这样描述 Class 类：Class 类的实例描述了系统运行时的类和接口。枚举类型认为是类类型，注解认为是接口类型。每个数组都可以用一个 Class 类的实例来描述，且所有基本类型相同、维数相同的数组共享一个相同的 Class 类实例。8 种基本类型（byte, short, char, int, long, float, double 和 boolean）以及关键字 void 均由 Class 元对象来描述。

Java 是个强类型语言，变量的类型限制并决定了其值的范围和解释方式。Java 语言支持两种不同的类型：基本类型和引用类型。基本类型分为 boolean 类型和数值类型，数值类型分为整数类型和浮点类型，整数类型有 byte, short, char, int, long 五种，浮点类型有 float 和 double 两种。引用类型分为类类型、接口类型和数组类型。此外还有一个特殊的 null 类型，它们的关系如图 3-6 所示。更详细的讨论请见文献[19]。

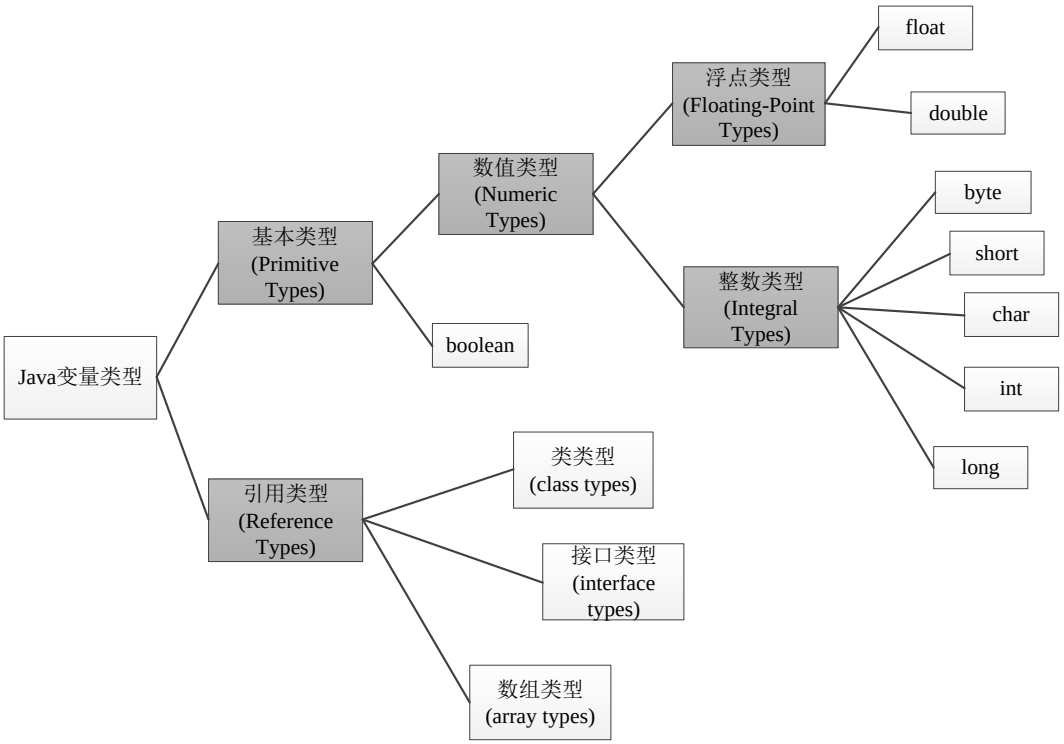

图 3-6 Java 语言的数据类型

将 API 文档的描述与 Java 语言规范中的描述相对比，不难看出 Class 类实例能描述的类型覆盖了 Java 语言规范中提及的除 null 类型外的所有类型。这是由 Java 的元对象模型所决定的，在 Java 中只存在唯一一个元类 Class，不存在多个元类来描述系统的其他元素。因此，Class 类承担了描述系统所有类型对象的职责。

3.4.2 三大核心反射对象——对象、方法和构造方法

java.lang.reflect 这个包包含了用于描述核心元信息的三个基本元素：Field（字段）、Method（方法）和 Constructor（构造方法）、对泛型及动态代理的支持。这几个核心反射元素类的主要架构如图 3-7 所示。

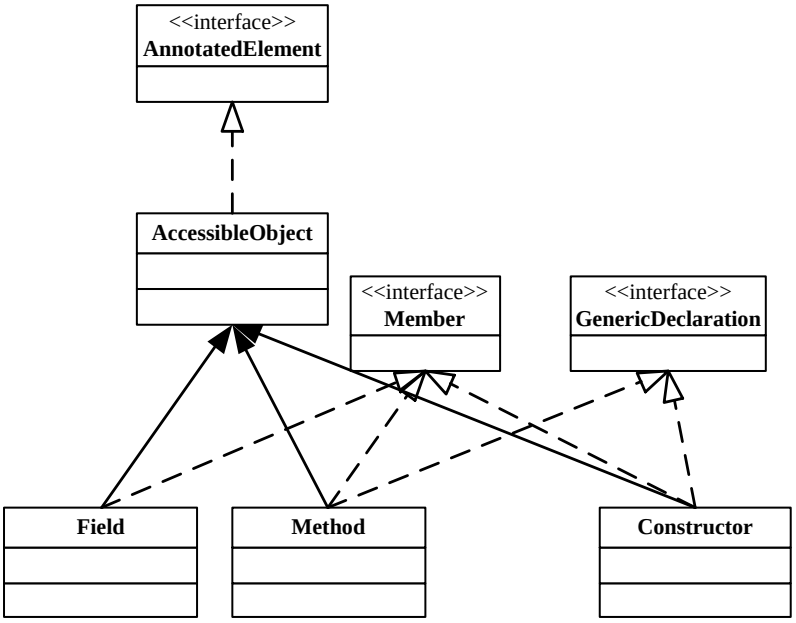

图 3-7 java.lang.reflect 包核心类的架构

Java 反射的三个基本元素 Field、Method 和 Constructor 均继承自基类 AccessibleObject，它负责反射对象存取时的权限控制。接口 AnnotatedElement 支持注解，接口 GenericDeclaration 支持泛型，Member 是一个标记接口，表示这些反射元素有归属（准确来说，归属于声明它们的 Class 类）。

3.4.3 注解支持

java.lang.annotation 这个包的结构相对简单，它定义了一个 Annotation 接口，是所有用户自定义注解必须实现的接口，同时定义了几个元注解（如 Target、Retention、Inherited 等）以及修饰元注解的枚举类（RetentionPolicy 和 ElementType）。在本论文中将不会涉及注解的使用方法和基本知识，而更多是着眼于注解实现与 Java 反射相关联的层面。

3.4.4 静态信息获取

既然字段、方法和构造方法是 Java 系统三个最核心的反射元素，那么弄清楚它们持有什么信息来描述系统对应层面的元素就尤为重要。4.3.1 节论文对源代码进行了挖掘，找出了这些信息在 Java 语言层反射对象中的定义。本节仅从反射体系的角度给出这几个类所持有的系统相应层面的元信息，如表 3-2、表 3-3、表 3-4 所示。

表 3-2 Field 类持有的字段信息

字段类型	字段名	字段描述
Class<?>	declaringClass	字段的声明类
String	simpleName	字段名
Class<?>	type	字段类型
int	modifiers	修饰符

表 3-3 Method 类持有的方法信息

字段类型	字段名	字段描述
Class<?>	declaringClass	方法的声明类
String	simpleName	方法名
Class<?>	returnType	方法返回类型
Class<?>[]	parameterTypes	方法的参数类型
Class<?>[]	exceptionTypes	方法声明的异常类型
int	modifiers	修饰符

表 3-4 Constructor 类持有的构造方法信息

字段类型	字段名	字段描述
Class<?>	declaringClass	构造方法的声明类
Class<?>[]	parameterTypes	构造方法的参数类型
Class<?>[]	exceptionTypes	方法声明的异常类型
int	modifiers	修饰符

反射对象的这些静态属性，Java 标准类库都提供了 API 来对它们进行读取。表 3-5、表 3-6 和表 3-7 总结了这部分 API。

表 3-5 Field 类读取基本信息的 API

修饰符和返回类型	方法声明
public Class<?>	getDeclaringClass()
public String	getName()
public int	getModifiers()
public Class<?>	getType()

表 3-6 Method 类读取基本信息的 API

修饰符和返回类型	方法名
public Class<?>	getDeclaringClass()
public String	getName()

续表

public int	getModifiers()
public Class<?>[]	getParameterTypes()
public Class<?>	getReturnType()
public Class<?>[]	getExceptionTypes()

表 3-7 Constructor 类读取基本信息的 API

修饰符和返回类型	方法声明
public Class<?>	getDeclaringClass()
public String	getName()
public int	getModifiers()
public Class<?>[]	getParameterTypes()

3.4.5 动态存取和动态调用

本节将讲解 Java 元对象协议的动态层面，主要有三个能力：对字段的动态存取、对方法的动态调用，以及动态地创建类实例。

3.4.5.1 字段动态存取

4.2.1 节提到，Java 语言中存在两种类型的对象：基本类型和引用类型。相应地，一个字段的类型既可能是基本类型，也可能是引用类型。在 Field 类的 API 中对这两种类型都提供了相应的支持，如表 3-8 所示。

表 3-8 Field 类的动态反射 API

修饰符和返回类型	方法声明
public Object	get(Object obj)
public short	getShort(Object obj)
public int	getInt(Object obj)
public long	getLong(Object obj)
…	…
public void	set(Object obj, Object value)
public void	setShort(Object obj, short value)
public void	setInt(Object obj, int value)
public void	setLong(Object obj, long value)
…	…

这套 API 分别具备了对基本类型字段和引用类型字段的 get/set 方法。如果采用存取引用类型字段的方法来操作基本类型的字段，那么自动装箱/拆箱将由编译器自动完

成（见文献[19]的第 5.1.7~5.1.8 节）；如果使用基本类型字段存取方法的过程中字段类型与存取类型不一致，类型的扩展/截断转换可能会发生（见文献[19]的第 5.1.2~5.1.3 节）。

以 Field 类的 get(Object obj)方法为例，其流程如图 3-8 所示。

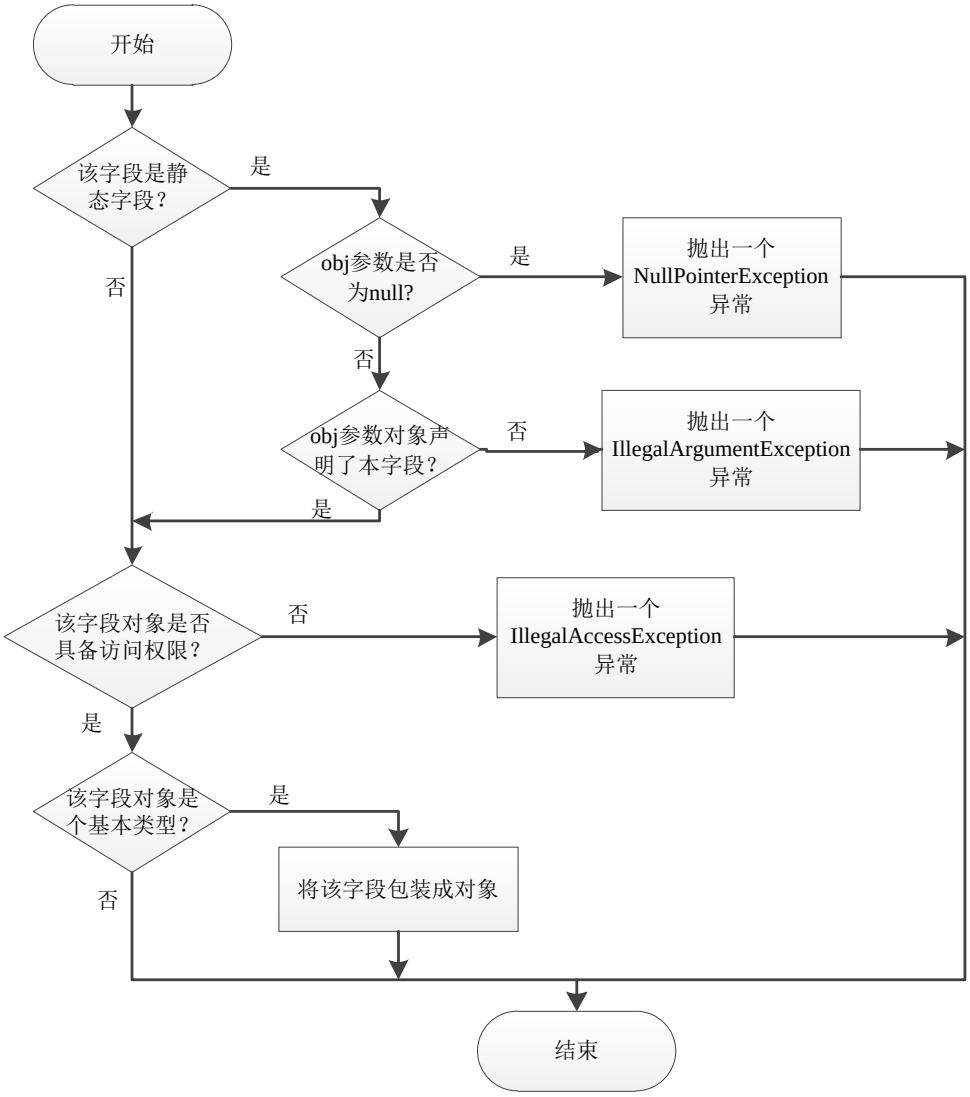

图 3-8 字段对象动态调用的 API 方法流程图

3.4.5.2 方法动态存取

Method 类提供了运行时动态调用方法的能力，它允许框架动态地调用运行时发现的用户方法，极大地扩大了 Java 程序设计的灵活性。Method 类用于支持动态调用的方法为 invoke 方法，它允许用户动态调用运行时发现的方法，是 Java 反射 API 中重要的动态方面。该方法的签名如表 3-9 所示。

该方法尝试使用 obj 对象调用本 Method 对象所描述的方法，并传入 args 数组作为参数，若方法调用成功，返回本 Method 对象描述方法的返回值。过程中可以发生类型包装和类型转换。其具体的执行流程如图 3-9 所示。

表 3-9 Method 类的动态反射 API

修饰符和返回类型	方法声明
`public Object`	`Invoke(Object obj, Object... args) throws IlleaglAccessException, IllegalArgumentException, InvocationTargetException`

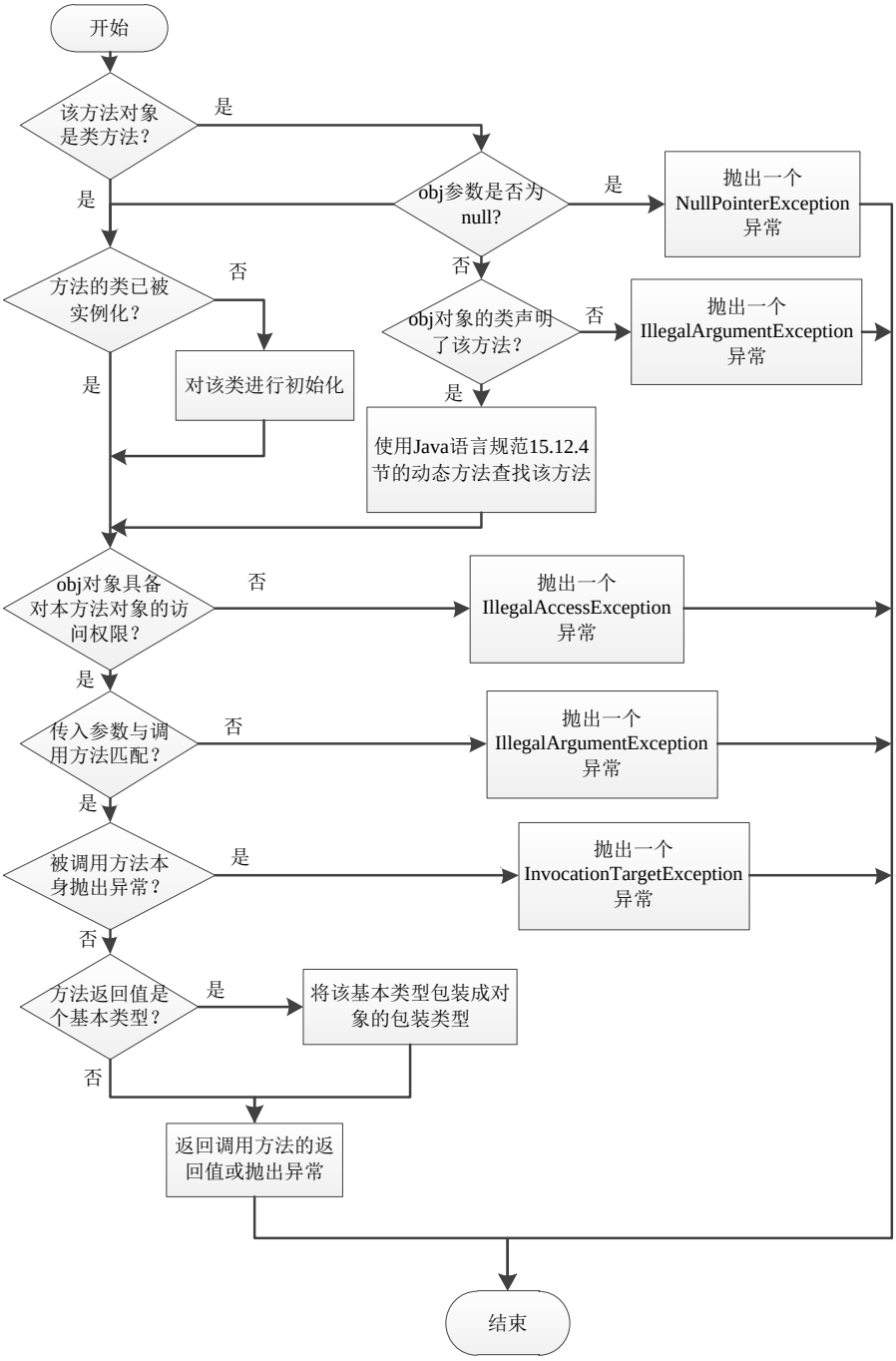

图 3-9 方法对象动态调用的 invoke 方法流程图

3.4.5.3 类实例的动态创建

Constructor 类提供了运行时动态生成一个类实例对象的能力，它允许框架动态地生成需要的对象（如 Spring 框架的 BeanFactory 等），极大地扩大了 Java 程序设计的灵活性。其用于支持类实例生成的方法为 newInstance 方法，如表 3-10 所示。

表 3-10 Constructor 类的动态反射 API

修饰符和返回类型	方法声明
public T	newInstance(Object... initargs) throws InstantiationException, IlleaglAccessException, IllegalArgumentException, InvocationTargetException

Constructor 类的 newInstance()方法用于创建一个类新的实例对象，如果该类还没有被创建或初始化，本方法可能会引发这些过程的发生。方法调用的流程如图 3-10。

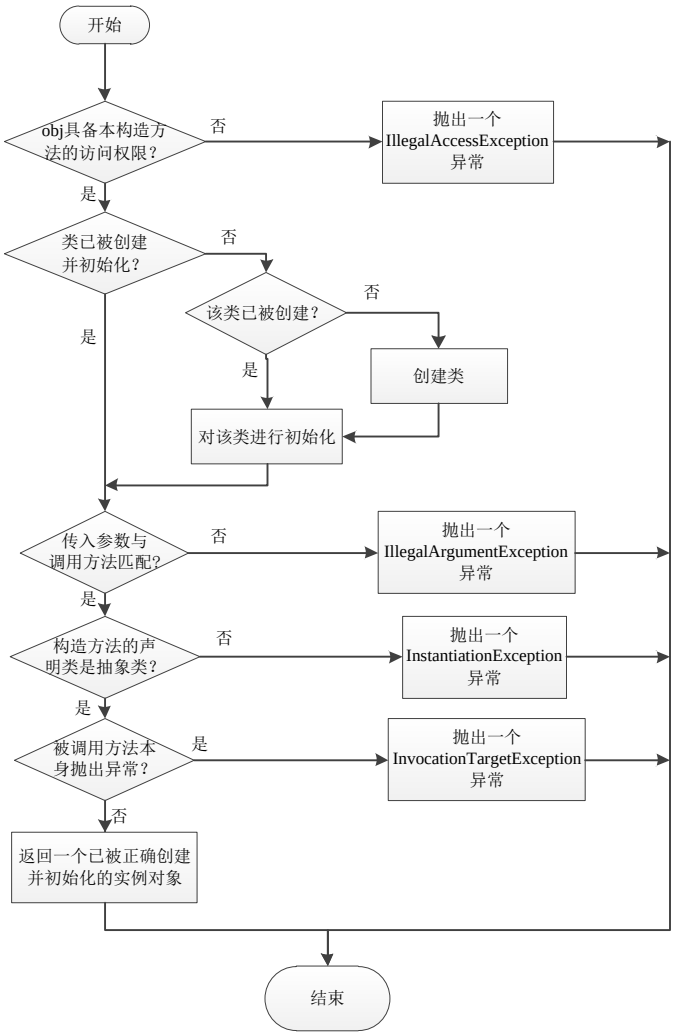

图 3-10 构造方法对象动态调用的 newInstance 方法流程图

第4章　Java 反射机制分析

本章以 HotSpot 虚拟机为样例介绍了 Java 反射机制的实现。图 4-1 是一个反射调用依次会经过的 4 个层面，以下各节分述了这 4 个层面和其中涉及的数据结构。为方便后面的讲解，4.1 节先介绍了 OpenJDK 项目的源码结构；4.2～4.3 节讲解了 Java 标准类库的公有实现；4.4 节讲解了 JDK 对标准类库的内部实现；4.5 节讲解了 class 文件的结构及其作用；4.6 节讲解了 Java 反射的本地实现；4.7 节讲解了虚拟机内部用于表示元数据的 oop/klass 核心数据模型；4.8 节讲解了一个反射调用完整实现涉及的各个方面，对虚拟机层面的讲解包含在此节中。

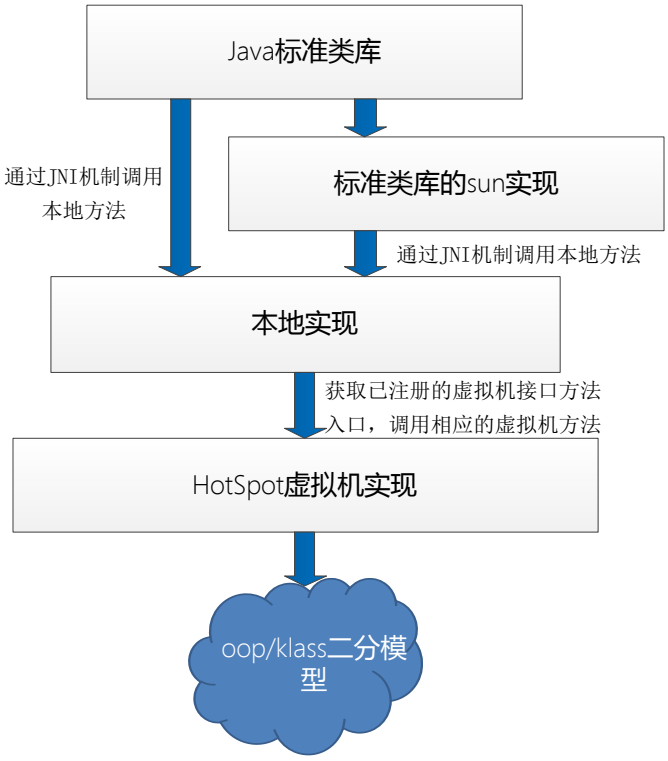

图 4-1　一个反射调用涉及的四个层面

4.1 OpenJDK 源代码结构

本论文对反射机制的研究建立在源码阅读的基础上进行，因此选择一个开源的 JDK 很有必要。论文采用的是 Oracle 公司的 OpenJDK，版本是 7u40-b43[1]，它是在维护分支 OpenJDK7 build 20 的基础上开发的版本[37]，表现和性能已经较为稳定，并且这个版本已经具备了较为完整的反射体系[38]。另外，它与我们开发使用的 Oracle JDK 除了一些闭源的商业实现（如 Flight recorder）外在很大程度上是相同的[39]，基本保留

[1] 下载地址：https://jdk7.java.net/source.html。

了原汁原味的 Oracle JDK。后续验证 Java 反射核心机制时，我们需要编译一个虚拟机来调试观察代码的运行，论文选择编译的也是这套 JDK 中的 HotSpot 虚拟机。

2014 年 3 月，Oracle 公司正式发布了 JDK8。新版本中引入了 Lambda 表达式、多重注解等新特性，并实现了增强提案 JEP 122[40]关于移除永久代的建议，改在本地内存中划出一块元空间来存储元数据[41]，如图 4-2 所示。论文采用的 OpenJDK7 在实现上依然使用永久代来存储类元数据。更多关于永久代移除过程的资料可以参考[42]。

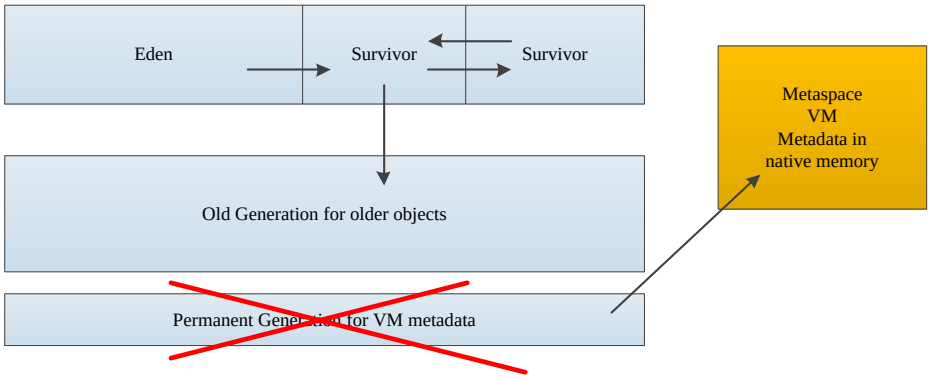

图 4-2 JDK8 永久代的移除

OpenJDK7u40-b23 源码包的结构如图 4-2 所示，其下包含了 corba 等几个子目录。

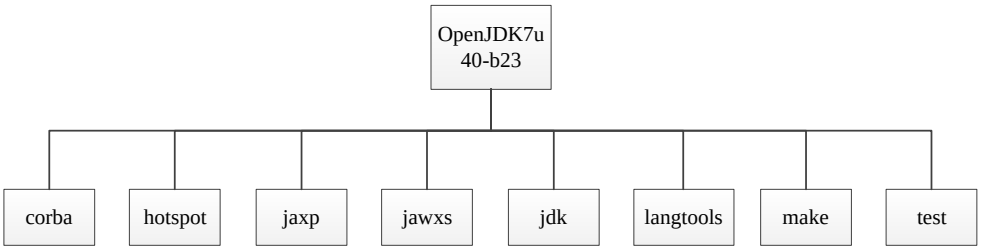

图 4-3 OpenJDK 包结构

其中，corba 目录包含了 OpenJDK corba 模块的源代码以及构建它们的 make 脚本；

hotspot 目录包含了 OpenJDK HotSpot 虚拟机的源代码和构建它们的 make 脚本；

jaxp 目录包含了 OpenJDK jaxp 模块的源代码以及构建它们的 make 脚本；

jaxws 目录包含了 OpenJDK jaxws 模块的源代码以及构建它们的 make 脚本；

jdk 目录包含了 OpenJDK 运行时类库的源代码以及用于构建它们的 make 脚本；

langtools 目录包含了 OpenJDK 的 javac 编译器和其他语言工具的源代码以及用于构建它们的 make 脚本；

顶层目录的 Makefile 用于构建整个完整的 OpenJDK[43]。

对于本论文的反射机制研究来说，最重要的两个包是 jdk 和 hotspot。jdk 包中主要包含了 JDK 标准类库和 Oracle 公司对标准类库的私有实现等，hotspot 包中则包含了 HotSpot 虚拟机的实现。关于这两个包各自包含的模块、所负责的功能以及它们具体的交互机制等，论文将在 4.3 节、4.4 节、4.7 节中选择精要的代码来讲解。

4.2 Java 核心反射对象体系

3.5 节已经对 Java 反射体系进行了架构上的总体讲解，本节将更详细地讲解 java.lang.reflect 包下几个重要反射对象的作用。

4.2.1 注解标记接口

该接口声明了一些方法来支持注解，如判断该实体是否应用了某个注解、以及一些获取注解的方法等。所有的反射对象都实现了这个接口，意味着所有的反射对象都支持注解。

4.2.2 权限控制中心

关于 AccessibleObject 这个类，它的官方 API 文档上如此描述：“AccessibleObject 提供了标记反射对象的能力，被标记的对象在使用时可以越过 Java 语言层面默认的权限控制检查。准确来说，这些权限检查发生在对一个 Field 对象进行设值/取值操作、动态调用方法或试图创建并初始化一个类的实例时。”这个类是所有反射对象的基类，关于反射对象的部分权限检查被提到这个类来完成。

4.2.3 成员标记接口

Member 接口包含了一些方法来获取一个成员拥有的一些属性，如修饰符、成员所属类、成员是否由编译器产生（是否语言层可见）等。Java 反射三个的基本元素类 Field、Method 和 Constructor 都继承了本接口的方法。

4.2.4 字段对象

Field 类的官方 API 文档描述为:“Field 类提供了对单个类字段或接口字段的信息进行检索和动态存取的能力。被反射的字段可能是一个类的（静态）字段或一个接口的字段。”一个 Field 对象不仅能对字段信息进行静态检查，还能对其进行动态存取。关于 Field 对象具体存储的字段信息，请见 4.3.1 节；关于 Field 对象提供给用户进行静态和动态信息存取的 API，请见 4.3.3 节。

4.2.5 方法对象

Method 类的官方 API 文档对其的描述为:“Method 类提供了对单个类方法或接口方法的信息进行检索及对方法进行动态存取的能力。被反射的方法可能是一个类方法，也可能是一个实例方法，甚至可能是一个抽象方法。”一个 Method 对象存储了一个方法的基本信息（方法名、参数个数及其类型、返回值类型等），并提供了对该方法进行动态调用的能力。关于 Method 对象具体存储的方法信息，请见 4.3.1 节；关于 Method

对象提供给用户进行动态方法调用的 API，请见 4.3.3 节。

4.2.6 构造方法对象

Constructor 类的官方 API 文档对其的描述为：“Constructor 类提供了对单个类的构造方法的信息进行检索及对方法进行动态存取的能力。”一个 Constructor 对象存储了一个类的构造方法的基本信息（参数个数及其类型等），并提供了对该构造方法进行动态调用的能力。关于 Constructor 对象具体存储的构造方法信息，请见 4.3.1.节；关于 Constructor 对象提供给用户进行动态方法调用的 API，请见 4.3.3 节。

4.3 Java 标准类库——水面上的冰山

4.2 节提到，java.lang.reflect 包包含了 Java 平台标准类库对反射的实现。本节将从这个包入手，挑选其中有代表性的源码来开始对 Java 标准类库中反射实现的讲解。

如 4.2.2 节所述，java 反射的三大基本的元素分别是 Field、Method 和 Constructor。我们在第 2 章中提到，一个反射系统必须具有系统的自描述，并且与其自描述必须存在安全的简单连系。在以下源码的挖掘过程中，论文以这个系统的“自描述”为核心，思考并尝试探索反射对象需要持有什么数据才能完整地描述系统的各个层面（如字段、方法等），同时也留意简单连系和安全性两个方面在 Java 标准类库反射设计中的体现。

具体来说，在本节的源码解读中，论文将尝试思考并解答以下几个问题：

(1) 这三个对象分别持有什么样的数据来描述它们各自代表的对象？

(2) 系统将这份数据结构存放于何处？它是何时被创建，又是何时被注入？

(3) 系统的自描述与系统是否存在简单连系？也即，Java 层面是否能通过简单的方法来操纵这些数据结构？若是，这个过程如何保证系统的安全和一致性？

4.3.1 元信息存储及其 API

在本小节中，论文选择采用 Field 类来进行讲解 Java 标准类库对反射体系的设计思路。原因在于，方法（包括构造方法）比字段拥有更多的属性，比如方法参数类型、返回值类型等，因此代码可能涉及更多对细节的处理。而对于论文要研究的数据存取方式、安全及权限检查等层面，其流程和设计思想是基本一致的。

3.4.4 节论文已经从 Java 反射体系的角度对 Java 核心的反射对象能提供的信息进行了整理，本节是从源代码的角度出发来研究，是对 3.4.4 节的补充加强。

以下代码节选自 Java 标准类库 java.lang.reflect 包下的 Field.java，第 59-68 行：

```
58  public final
59  class Field extends AccessibleObject implements Member {
60
```

```
61      private Class<?>            clazz;
62      private int                 slot;
    …
66      private String              name;
67      private Class<?>            type;
68      private int                 modifiers;
```

由代码可以看出，Field 类的 API 中提供支持的几个字段信息都在这里声明，包括其定义类、字段名、字段类型以及修饰符等。这四个域完整地描述了一个 Java 语言层面的字段的所有信息，而其中的 slot 域是用于对序列化的支持，并非字段具有的属性。实际上，在 Field 类的内部还定义了更多的域来支持更多丰富的反射特性，如字段签名（用于支持泛型）signature、泛型信息仓库 genericInfo、注解 annotations 等。

类似地，Method 类和 Constructor 类中持有的方法和构造方法的元信息也可以分别在 Method.java 的第 65-73 行和 Constructor.java 的第 66-70 行找到。有兴趣的读者可以自行查阅源代码。

4.3.2 数据注入

至此 4.3 节开头的第一个问题已经得到回答，值得思考的另一个问题是，这些数据被系统存储于什么地方？它在什么时候被创建和注入？关于数据的注入，其来源可能有三个不同的地方：

(1) 通过自身计算产生；

(2) 通过设值方法（setter）注入；

(3) 通过构造方法注入；

遗憾的是，对于第一种和第二种情况的可能性回答都是否定的：以 Field 类为例，类中既不存在对这几个基本信息的设值方法，也没有在计算过程中直接修改它们的值。而且这几个基本信息域的访问权限都被设置为私有（private），这意味着不可能通过任何外部途径来使用这些内部信息的值。那么，就只剩下最后一种可能的注入方式，即通过构造方法注入。

在 Field 类的第 111 到 126 行中存在这样一个构造方法：

```
111     Field(Class<?> declaringClass,
112           String name,
113           Class<?> type,
114           int modifiers,
115           int slot,
116           String signature,
```

```
117                 byte[] annotations)
118         {
119             this.clazz = declaringClass;
120             this.name = name;
121             this.type = type;
122             this.modifiers = modifiers;
123             this.slot = slot;
124             this.signature = signature;
125             this.annotations = annotations;
126         }
```

所有基本的元信息都在构造方法中被注入进来。这说明了一个事实：即反射对象只是 Java 层面用来表示系统自描述的对象，实际的元数据是从其他地方注入进来，并不存在于 Java 的语言层面。

另外，这个构造方法的写法暴露了更多值得注意的细节。首先，它是一个默认访问权限的构造方法，说明仅 java.lang.reflect 包的类和方法具有调用这个构造方法的权限。这体现了它的设计考虑：不允许外部其他任何包初始化这个 Field 类的实例对象。但同时，设计者又在 java.lang.reflect 包下提供了一个 ReflectAccess 类，为外界提供反射对象的生成服务。为何要做这样看似矛盾的设计？论文将在 4.4.1 节深入讲解。

通过本节对反射对象元数据注入的研究，针对 4.3 节一开始提出的第二个问题，论文做出如下回答：反射对象 Field 等只是 Java 语言层面的描述，系统存储元数据的地点不在语言层。考虑到一个 Java 对象在虚拟机内部是用多个 C++对象来描述，可以推论系统的元数据应该是从更底层的虚拟机内部注入到这些反射对象中。这个推论的基础，需要对 Java 虚拟机的概念模型有所了解，这部分知识连同反射对象被创建和注入的时机将在 4.8 节进行讲解。

4.3.3 元数据操作及其 API

4.2.1 节中给出了 AccessibleObject 的权限检查发生时机，具体来说，发生在以下的三种时刻：

(1) 对一个 Field 对象进行设值/取值操作时；

(2) 动态调用一个 Method 对象描述的方法时；

(3) 创建并初始化一个 Constructor 类描述的类的实例时。

这恰好是 Java 元对象协议所提供的三个动态操作元信息的能力（存取字段、动态调用、创建类实例）。它们的 API 已经在 3.5 节中讲解，具体的代码实现中将涉及元数据存取、安全与权限检查两个层面的问题，是 Smith 博士提出的反射系统三大要素

的其中两个。论文将在接下来的 4.3.4 节进行讲解。

4.3.4 简单连系的实现

3.5 节中给出了 Java 元对象协议中动态层面各个 API 的概念流程，而这些方法具体是如何实现的呢？方法实现是通过何种方式来动态存取元数据的？方法如何实现复杂的权限检查流程呢？本节将回答实现代码存取元数据的方法。

本节将仍然以 Field 类的代码来讲解简单连系，Method 类和 Constructor 类的实现原理与 Field 类大同小异。本节选择 Field 类对引用类型的 get 方法进行讲解，其他设值/取值方法的实现与本方法并无二样，感兴趣的读者可以自行参阅源代码。

get 方法在 Field.java 源文件的第 370-380 行：

```
370     @CallerSensitive
371     public Object get(Object obj)
372         throws IllegalArgumentException, IllegalAccessException
373     {
374         if (!override) {
375             if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
376                 checkAccess(Reflection.getCallerClass(), clazz, obj, modifiers);
377             }
378         }
379         return getFieldAccessor(obj).get(obj);
380     }
```

这段代码背后是整个 Java 标准类库对反射机制很精华的两个设计：一是其存取器模式的设计，二是其权限检查机制的设计。虽然这段代码节选自 Field 类，但 Method 类和 Constructor 类的实现如出一辙，设计思想是相通的。阅读代码第 380 行，它没有直接去存取字段的值，而是通过一个存取器 FieldAccessor 来完成存取。

观察 Field 对象的设计，它并没有在类中维护一个字段，用来表示该字段的值。笔者认为，原因在于元数据并非在 Java 语言层面维护的，就算在此提供一个数据结构来描述，还必须把语言层的变化反映到 VM 层以维护系统与其自描述的一致性。因此，不如把操作元数据的地点往底层推，减少维护成本。

存取器模式的设计本质上是一层抽象，通过提供一个统一的接口，使得语言层不必关心具体存取字段的类型。具体的 Accessor 实现会在使用的时候被注入进来。在设计上，FieldAccessor 是个接口，它的实现是由 sun.reflect 包的 ReflectionFactory 工具类注入的。由于存取器模式的具体实现在 Oracle 公司 sun 包下的私有实现中，因此其实现分析也放到 4.4.2 节中讲解。

4.4 标准类库的内部实现

在 Java 标准类库中使用了一部分内部实现，这部分包位于 sun 包下。java.*、javax.* 和 org.*包是 JavaSE 的官方 API 标准，是对外承诺维护的 Java 开发接口，其实现与平台无关，而且为了保持向后兼容性，一般不会轻易做出大的修改。而 sun.*是 Oracle 自己对 JDK 平台标准类库的私有实现，不是 Java 对外承诺的标准接口，其实现是与平台相关的，而且在后续的版本可能有大的变动。因此 Oracle 公司并不鼓励编程人员使用这个内部实现，否则因此带来的兼容性损失需由开发者自行负责。

Java 为何能通过提供一个统一的对外接口来达到平台无关性？那是因为它通过底层的实现把平台的差异性向上屏蔽了，但在不同的平台上是需要针对平台编写不同的代码的。OpenJDK 项目下的 jdk/src 路径下有一个 share 文件夹，表示各平台共享的逻辑，这个包通过提供统一的接口来达到平台无关，而在具体的实现中，则需要根据不同的平台来注入不同的平台实现。不同平台的实现代码在其他的包中，如 bsd 平台、linux 平台、macosx 平台及 windows 平台等。

在上一节 JDK 标准类库的实现中我们提到了一些 sun 私有实现的类，比如 LangReflectAccess 类，字段和方法的存取器 Accessor，权限检查 Reflection 工具类等。它们的作用将在这一节讨论。

首先，先来看一下 sun.reflect 包的基本结构，这个包下有很多类，还有几个包，它们从功能上可以大致分为如图 4-4 所示的几个模块。

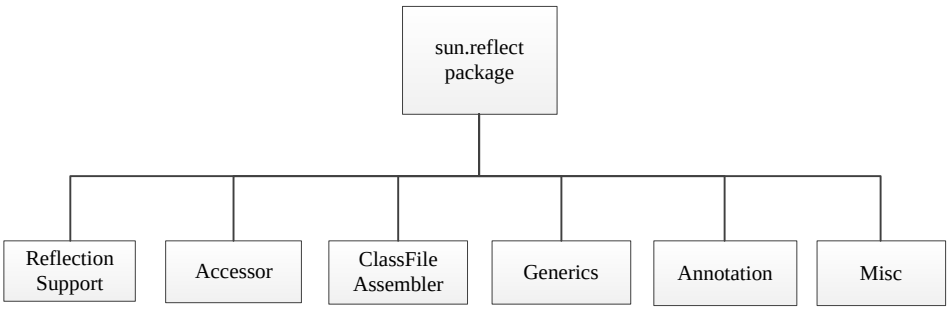

图 4-4 sun.reflect 包模块结构

ReflectionSupport 模块主要是对 Java 标准类库反射实现功能的补充，Accessor 模块是对 Java 标准类库中存取器模式的具体实现，ClassFileAssembler 模块主要是实现 class 静态文件到运行时字节数组的转换。Generics 模块和 Annotation 模块分别是对 Java 标准类库下泛型和注解的实现，本论文将不涉及。Misc 模块是其他繁杂的支持，如 UTF-8，签名迭代器等，本论文也不过多涉及。

4.4.1 安全与反射工厂

4.3.2 节曾提到，对反射对象接入权限的安全检查非常严格，但同时反射包下又提供了一个 ReflectAccess 类作为接口，为外界提供反射对象的生成服务。这个设计看起

来是矛盾的，但进一步思考，这个设计又是符合情理而且非常精妙的。不过首先要弄清楚的是，系统对反射对象的创建和获取有什么需求呢？

首先，作为一个实现了简单连系的反射系统，任何关于元数据的修改都会真实及时地反映到系统中。语言层的反射对象作为 Java 最核心的三个反射元素，它们提供了系统三个重要层面（字段、方法、构造方法）的自描述，任何对它们的修改都会反映到系统中去。因此，反射对象是不允许用户自己创建的，反射对象的构造方法都被设计为包私有访问权限，否则用户有意或无意的修改都可能导致系统的不一致。那么将它们直接设置为私有构造方法行不行呢？

这要考虑到系统对反射对象的另一个需求。虽说反射对象不允许用户自己创建，但在系统内部实现的一些层面可能还是需要创建反射对象的，比如持久化时可能需要创建临时的字段对象。这时如果把构造方法设置为私有，显示无法满足这样的需求。但这样的话另外一个问题又来了，既要不允许用户的外部访问，又要允许系统内部特定部分的访问，这个权限控制的粒度应该怎么设计呢？

回答已经在源代码中。Java 因此设计了这个 ReflectAccess 类作为内外沟通的接口，专门处理反射对象的生成事务，而安全考虑已经融入到设计本身——这个类的位置和访问权限。

4.4.1.1 反射工厂——ReflectionFactory

在进一步探索 ReflectAccess 类的设计考虑之前，先来看看 sun.reflect 包下一个重要的类，它是所有反射对象生成的主工厂。它提供了 java.lang.reflect 包下的 Field、Method 和 Constructor 反射对象以及它们对应存取器（Accessor）的生成服务。

且慢，刚刚不是说反射对象都是包私有的构造方法？为什么这个工厂可以获取反射对象的生成权限？源码之前，了无秘密。论文就以这个类中的 newField()方法作为例子来讲解，其代码在 OpenJDK 项目的 jdk/src/share/classes/sun/reflect 路径下 ReflectionFactory 类的第 207-222 行：

```
207  public Field newField(Class<?> declaringClass,
                           …) // 传入了很多参数，此处省略
214      {
215          return langReflectAccess().newField(declaringClass,
                                                 …); // 很多参数，省略。
222      }
```

可以发现，它又转调了 langReflectAccess()方法拿到一个对象，再把新建字段的请求委托给这个对象。这个对象是什么呢，它在源代码的第 55 行声明，是一个 LangReflectAccess 接口类型的对象。这个接口只有一个实现，就是 java.lang.reflect 包下的 ReflectAccess 类。因为 ReflectAccess 被放置在 java.lang.reflect 包下，它拥有对

反射对象构造方法的调用权限。它作为一个中介者，用以处理 ReflectionFactory 生成反射对象的请求，其工作原理如图 4-5 所示。

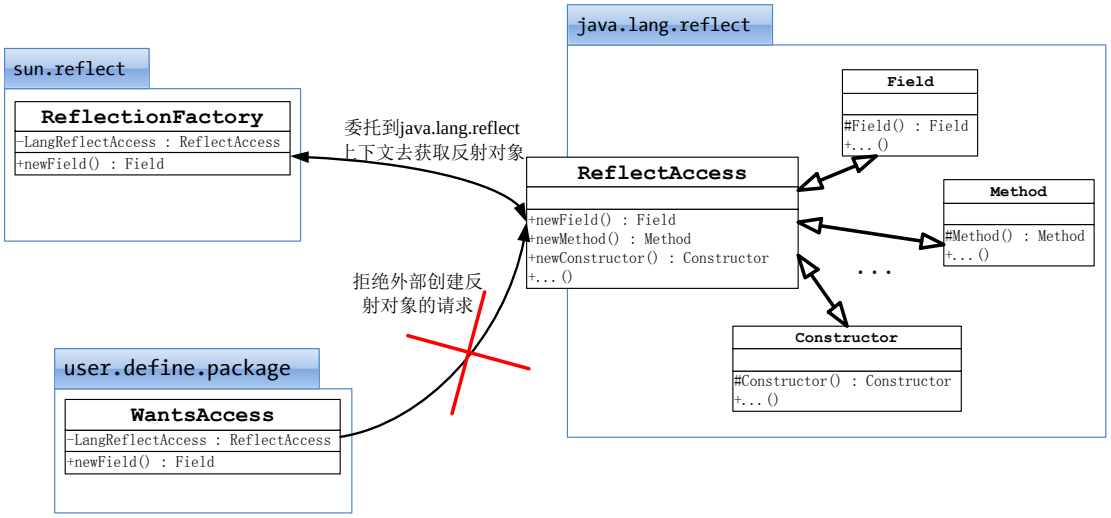

图 4-5 中介者 ReflectAccess 工作原理

4.4.1.2 安全控制

4.4.1.1 节解答了一个问题，即反射工厂最终是调用 ReflectAccess 类来负责反射对象事务。图 4-5 还有另外一个部分，即 ReflectAccess 类会拒绝用户自定义类的反射对象请求。事实上，用户的类根本就“看不见”ReflectAccess 这个类，因为它是包私有的访问权限，在 Java 的官方 API 文档也查不到这个类。但如果 ReflectAccess 类是包私有的，外部的反射工厂 ReflectionFactory 类又怎么能访问到它呢？

答案依然在源代码中。见 ReflectionFactory 类的第 408 行：

```
408     private static LangReflectAccess langReflectAccess() {
409         if (langReflectAccess == null) {
410             Modifier.isPublic(Modifier.PUBLIC);
411         }
412         return langReflectAccess;
413     }
```

在代码的 409 行判断 langReflectAccess 是否为空，如果是，调用 Modifier 类的静态方法 isPublic()，然后返回 langReflectAccess。难道 isPublic 这个方法会返回一个 LangReflectAccess 对象？查看其源代码，不会，它正常地返回一个布尔值。但在 410 行一定发生了什么事并注入了一个 ReflectAccess 对象，而且这一步是整个反射对象安全机制的关键，它把操作反射对象的权限授权给了这个反射工厂。

ReflectionFactory 类有一个 langReflectAccess 对象的公有设值方法，有可能是在 Modifier 中调用了这个设值方法，而且 Modifier 类也在 java.lang.reflect 包下，它确实有创建 ReflectAccess 类的权限。基于这样的猜测，阅读一下 Modifier 类的源代码，在

代码 53 行。

```
53          static {
54              sun.reflect.ReflectionFactory factory =
55                  AccessController.doPrivileged(
56                      new ReflectionFactory.GetReflectionFactoryAction());
57              factory.setLangReflectAccess(new java.lang.reflect.ReflectAccess());
58          }
```

代码第 49 行的注释中提到，这段代码是 java.lang 包和 java.lang.reflect 的初始化协议。可以看到，在代码的 57 行确实是注入了一个 ReflectAccess 类，至此反射工厂就被授权拥有了对 java.lang.reflect 包下反射对象的新建权限。而用户自定义的类由于没有经过这段代码的授权，根本连 ReflectAccess 这个类都看不到。整个内部授权的机制就是这么完成的，它的初始化顺序如图 4-6 所示。

总结起来，java.lang.reflect 包提供外界接入反射对象的安全机制设计是非常精巧的，其设计粒度提供了：

(1) 反射对象构造方法的访问权限都是包私有，因此 java.lang.reflect 包以外的包不可能通过构造方法直接生成反射对象；

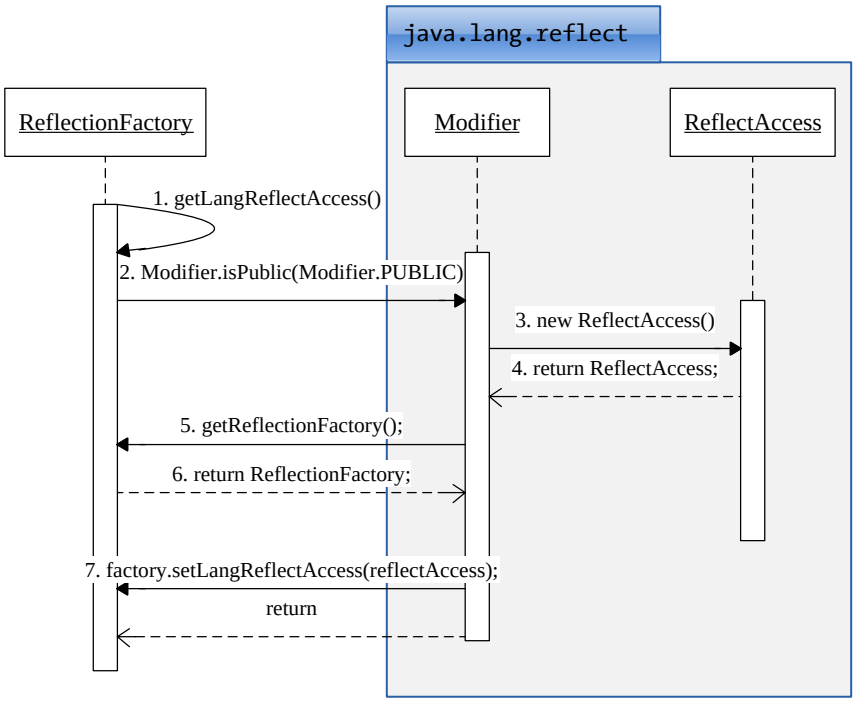

图 4-6 ReflectionFactory 初始化顺序图

(2) 为了提供外界访问，ReflectAccess 被设计出来作为与外界的接口，但它也是一个包私有的类，不能被用户自定义的包和类直接使用。该类主要提供给系统内部使用，并且必须经过特别的授权；

(3) 目前仅对 sun.reflect 包下的 ReflectionFactory 类进行了授权。授权是通过反射包下的公共类 Modifier 的静态初始代码完成的，这段代码会将 ReflectAccess 类通过设值方法注入到反射工厂中。

4.4.2 存取器架构设计与实现

在 4.3.4 节中，我们提到在 Java 语言层的反射对象中，存取元信息通过的是一个 Accessor。当一个反射对象的 Accessor 尚不存在时，它会向 ReflectionFactory 请求一个。基本的 Accessor 体系如图 4-7 所示。

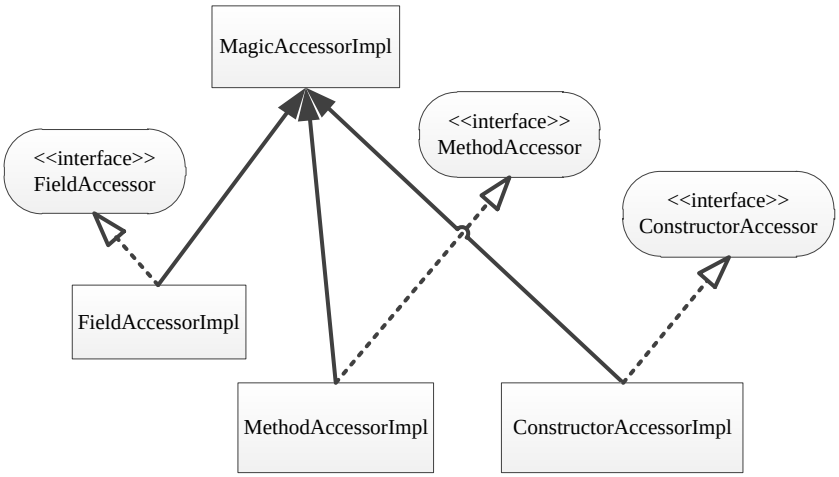

图 4-7 存取器模式的基本架构图

所有的存取器实现（AccessorImpl）都继承了 MagicAccessorImpl 这个类，它是一个标记类，并不是任何类的实现，其命名只是为了和继承它的各个存取器实现的命名保持一致。它的“魔术”之处在于，继承它的所有子类均可以无条件通过 VM 的权限验证，这部分权限验证的代码是写在 VM 中的，因此虚拟机“认识”这个类。如果要对这个类的名称做出改动，那么也必须对相应部分的虚拟机代码做出修改。也因其权限之高，其实现者选择把部分安全和权限检查放在 Java 语言层面，在调用存取器之前完成相应的安全检查。

4.4.2.1 字段存取器体系

字段存取器的类继承体系如图 4-8 所示。顶层的 UnsafeFieldAccessorImpl 是个抽象类，它继承自 FieldAccessorImpl 类，后者定义了与 java.lang.reflect.Field 类相对应的 18 个动态存取字段的设值/取值方法，也就是 3.5.5.1 节给出的对字段动态存取的 18 个 API。

上图的类体系看起来非常复杂，其实主要的分支只有三个。即 UnsafeStaticFieldAccessorImpl 、 UnsafeQualifiedFieldAccessorImpl 以及对 UnsafeFieldAccessorImpl 的一套直接实现。每个分支下各有 9 个类，分别处理 Java 语言中 9 种不同的字段类型（8 中基本类型加上对象类型）。具体每个类的实现都很巧妙，

它只负责其基类 18 个 API 与自己相关的 2 个（一个设值方法，一个取值方法），对于其余的 10 几个 API 一律抛出 IllegalArgumentException 或 IllegalAccessException 异常。关于将异常封装到类或方法中这种设计模式，读者可在[50]一书中找到更详细的资料。

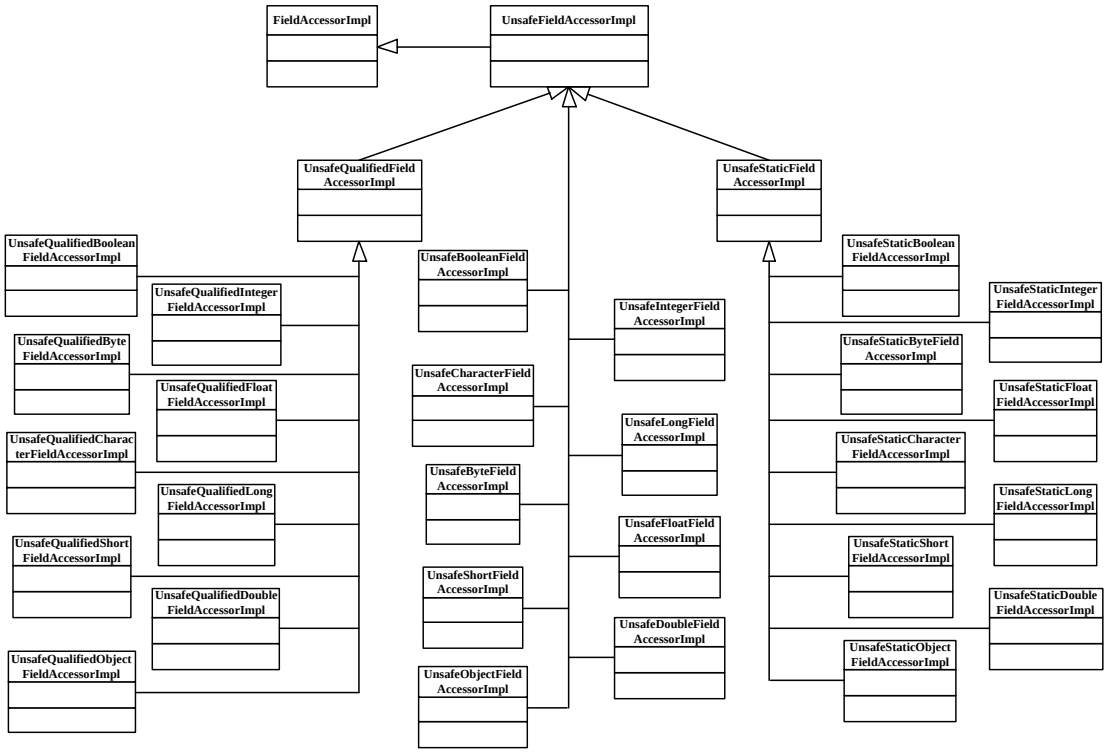

图 4-8 字段存取器类继承体系

4.4.2.2 方法存取器体系

相比与字段存取器体系，方法存取器的体系就简单得多。基类 MethodAccessorImpl 依然实现了 3.5.5.2 节展示的 java.lang.reflect.Method 类 API，不过只有一个 invoke() 方法。其子类也只有两个，相应的类继承体系如图 4-9 所示。

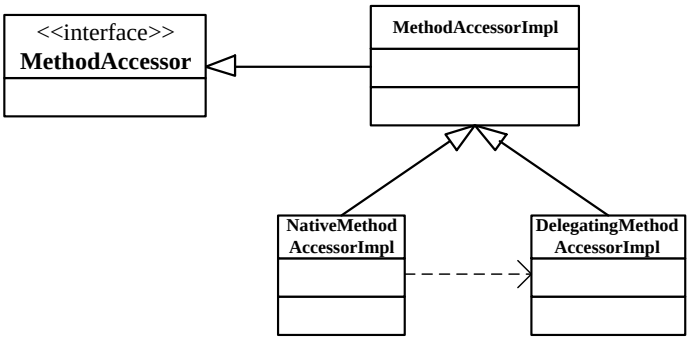

图 4-9 方法存取器类继承体系

这里设计了两个方法存取器的实现。一般情况下，方法存取器都是通过一个生成器 MethodAccessorGenerator 实时根据方法来生成其字节码和存取器，这个方法在正常情况下会非常快。但是在系统启动的初始，类的本地字节码是第一次生成，此时采用

本地生成的速度反而要慢 3-4 倍。对于一些频繁使用反射机制来自启动的框架来说，启动时间可能会受到影响。因此，在方法存取器的实现上，设计团队采用了一种“inflation”机制，即在系统启动的过程中，前面几次反射方法的调用采用更快的方法存取器来实现，等到系统启动完毕后再恢复直接生成存取器的实现。Inflation 机制由 ReflectionFactory 类支持，有兴趣的读者可以参阅该类的源代码。

为了保证这种切换能够平滑进行，不影响系统其他部分的设计，一个 DelegatingMethodAccessorImpl 被引入了。两种不同的实现都会被委托给这个类来调用，通过这个统一的接口来屏蔽实现的不同。

4.4.2.3 构造方法存取器体系

构造方法存取器的体系跟方法存取器体系类似，同样使用了一个相似的本地构造器实现来加快系统启动时的反射速度。此外，在用户试图获取一个抽象类或者 Class 类的构造方法存取器时，一个异常存取器将会被返回，这同样是异常类设计模式的应用，可以避免在获取构造方法存取器的时候因为异常路径而将注意力转移到异常处理流程上，是甚为精巧的设计。构造方法存取器的体系如图 4-10 所示。某种程度上，Java 语言层的三个反射对象对应的委托存取器的接口设计，完整地反映了 Java 这个元对象体系的全部协议。

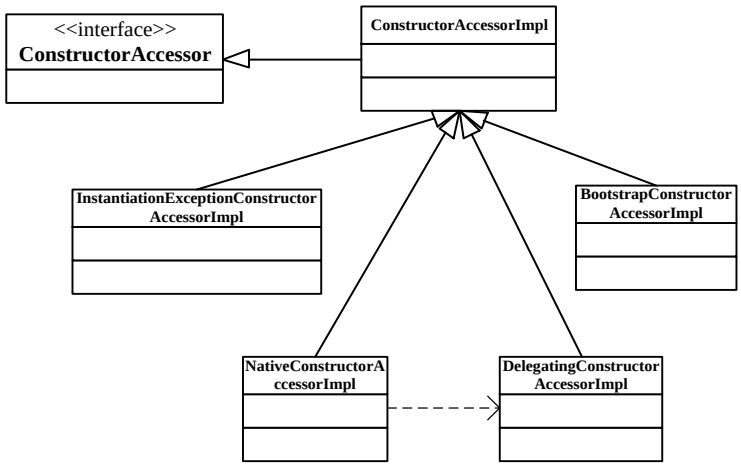

图 4-10 构造方法存取器类继承体系

4.4.3 类文件组装模块

这个模块主要由几个类组成：ByteVector、ByteVectorImpl、ByteVectorFactory、UTF、ClassFileConstants、ConstantPool 和 ClassFileAssembler。前三者构成一个封闭的系统，仅以工厂为一个对外的接口，而 ByteVector 是对字节数组操作的一个简单封装。ClassFileConstants 提供了与 Java 虚拟机规范对接的字节码内容声明。ConstantPool 负责从底层的 constantPoolOop 对象中获取数据，它涉及了 jvm 底层的实现和一些敏感的数据及操作，通过 Reflection 类注册要求数据保护和过滤。

4.5 元数据仓库——class 文件

看到上面的解析，我们离反射底层使用的核心数据结构又进了一步，有些心急的读者可能已经想接着 native 本地代码进一步挖下去了。别急，在进一步深入源代码之前，我们需要介绍一些基本的知识。本节和下一节将分别讲一台 Java 虚拟机的外观以及它赖以执行的.class 文件，也就是我们说的字节码文件。

关于字节码文件的内容和设计，已有很多书籍和资料可以参考。笔者在此本着“够用”的原则，只叙述与反射相关且重要的部分和设计思想，不会照搬规范的内容。如果读者想更深入地研究这部分内容，可以参见文献[20]和[24]。

Sun 公司坚决地将其文档规范分成了两份：语言规范[19]和虚拟机规范[20]，并且在虚拟机规范中提到，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。任意语言都可以使用有效的 class 文件来存储它们的数据并为虚拟机所接受处理。Java 虚拟机作为一个通用的、机器无关的执行平台，任何其他语言的实现者都可以选择将虚拟机作为其语言产品的交付媒介。

在 Java 虚拟机规范中，一个 class 文件对应着唯一一个类或接口的定义信息，每每个 class 文件都由以 8 位为单位的字节流组成。它是一个重要的数据仓库，Java 类或接口被编译后都以 class 文件的形式存在，这个文件包含了类或接口的很多重要信息，比如字段信息、方法信息、类继承体系信息等。

4.5.1 class 文件结构

每个 class 文件都对应着如图 4-11 所示的 ClassFile 结构。

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

图 4-11 class 文件结构

关于这个文件格式更详细的讨论请参见其他相关的资料，这里只叙述跟反射实现联系紧密的一些域。

4.5.1.1 magic（魔数）

魔数唯一的作用是用于确定这个文件是否为一个能被 Java 虚拟机所接受的 class 文件。其固定值为 0xCAFEBABE，不会改变。

4.5.1.2 constant_pool_count（常量池计数器）

常量池计数器的值等于常量池表中的成员数加 1。常量池表中的索引值只有在大于 0 且小于 constant_pool_count 的时候才认为是有效的，但常量池表的 0 索引可以被用来表达“不引用任何一个常量池项的常量池数据结构”的意思。

4.5.1.3 constant_pool（常量池）

常量池是一种表结构，包含 class 文件结构及其子结构中引用的所有字符常量，包括类和接口名、字段和方法描述符等。是一个极为重要的数据结构，其精华在类加载阶段将被虚拟机的类加载子系统吸取，并成为一个运行时常量池的一部分，供类或接口各种访问的需要。

4.5.1.4 access_flags（访问标志）

访问标志。它是一种掩码标志，用于表示某个类或接口的访问权限及属性。

4.5.1.5 this_class（本类索引）

其值必须是对常量池表中项的一个有效索引值，常量池在这个索引处的成员必须为 CONSTANT_Class_info 类型常量（见 4.5.2.1 节）。

4.5.1.6 super_class（基类索引）

其值必须为 0 或者对常量池表中项的一个有效索引值，此时常量池在这个索引处的成员必须为 CONSTANT_Class_info 类型常量。0 值则表示本类没有基类，那么它只可能是 Object 类。

4.5.1.7 interfaces_count（接口计数器）

接口计数器。其值为当前类或接口的直接父接口的数量。

4.5.1.8 interfaces（接口表）

接口表。其每个成员的值必须是一个对常量池表中项的一个有效索引值，常量池在这个索引处的成员必须为 CONSTANT_Class_info。该表的长度为 interfaces_count。

4.5.1.9 fields_count（字段计数器）

字段计数器。其值为当前 class 文件 fields 表的成员个数。

4.5.1.10 fields（字段表）

字段表。其每个成员的值都必须是一个 field_info 结构的数据项，用于表示当前类或接口中某个字段的完整描述。该表描述当前类或接口所声明的所有字段，但不包括从父类或父接口继承下来的字段。完整的结构见 4.5.3 节。

4.5.1.11 methods_count（方法计数器）

方法计数器。其值为当前 class 文件 methods 表的成员个数。

4.5.1.12 methods（方法表）

方法表。其每个成员都必须是一个 method_info 结构的数据项，用于表示当前类或接口中某个方法的完整描述。该表描述了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<init>和类或接口初始化方法<clinit>，但不包括从父类或父接口继承下来的方法。

对于反射而言，我们关心的是类/接口、字段方法、和常量池。以下 4.5.2 节将讲解常量池和类/接口结构，4.5.3 节讲解字段结构，4.5.4 节讲解方法结构，其他的属性（比如 class 文件中的 attribute 属性等）限于篇幅无法详细讲解。请有兴趣的读者自行参阅文献[20]。

4.5.2 常量池

所有的常量池表的项的格式都必须相同，均一个单字节“tag”项开头，指示该常量池表项的类型。常量池结构如图 4-12 所示，其 tag 字段的有效取值及相关含义读者同样可以在文献[20]中查阅到，此处限于篇幅仅在行文中列出必须的值。

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

图 4-12 常量池表项的结构

4.5.4.1 CONSTANT_Class_info 结构

CONSTANT_Class_info 结构用于表示类或接口，其格式如图 4-13 所示。

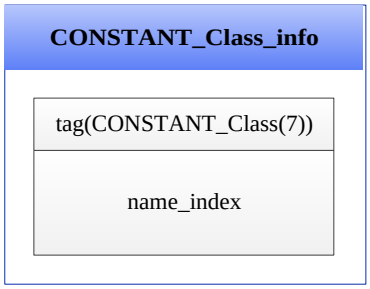

图 4-13 CONSTANT_Class_info 结构

其中，tag 项的值为 CONSTANT_Class(7)；name_index 是一个 u2 类型变量，其值必须是对常量池表的一个有效索引。常量池表在该索引处的成员必须是一个 CONSTANT_Utf8_info 结构，代表一个有效的类或接口二进制名称的内部形式。其与 CONSTANT_Utf8_info 常量池项的关系如图所示。

4.5.4.2 CONSTANT_Utf8_info 结构

CONSTANT_ Utf8_info 结构用于表示字符常量的值，其格式如图 4-14 所示。

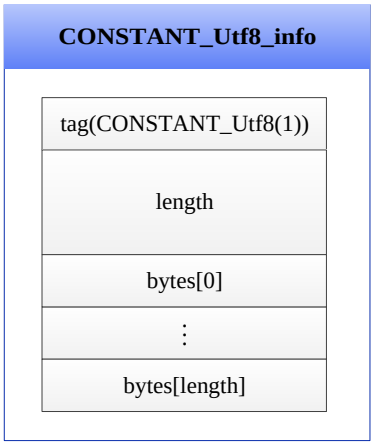

图 4-14 CONSTANT_Utf8_info 结构

length 是 u2 类型的变量，指示 bytes 数组的长度（而不是 C 等语言使用‘\0’等字符来区分字符串末尾的方式）。bytes 是 u1 类型的数组，字符常量采用改进过的 Utf-8 编码表示。具体的差异可以参阅文献[20]，此处不多做介绍，可以认为它们是一般的 Utf-8，对本文的理解没有太大影响。关于 Utf-8 编码的相关知识请参阅[44]。

4.5.4.3 CONSTANT_Fieldref_info、CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 结构

CONSTANT_Fieldref_info、CONSTANT_InterfaceMethodref_info 和 CONSTANT_Methodref_info 结构分别用来描述一个字段、接口方法、方法的信息，它们由类似的结构表示，如图 4-15 所示。

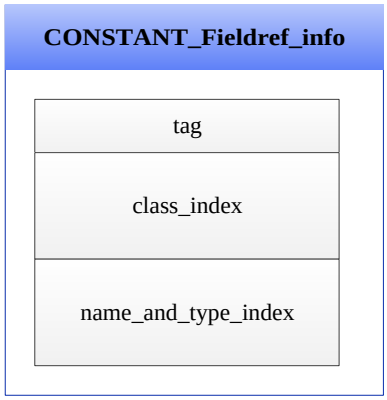

图 4-15 CONSTANT_Fieldref_info、CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 结构

其中 tag 项的值视类型的不同而不同，字段引用的值为 9，方法引用的值为 10，接口方法的引用为 11。其实这些引用值只起一个标记的作用，只要相互之间不重复即可，并没有太多的含义。class_index 项必须是对一个类或接口项的引用；

name_and_type_index 项必须是对一个常量池表项 CONSTANT_NameAndType_info 结构的引用，用于表示当前字段或方法的名字和描述符。

4.5.4.4 CONSTANT_NameAndType_info 结构

CONSTANT_NameAndType_info 结构用来表示字段或方法，但它没有标识出它们所属的类或接口，仅仅表示名字和描述符，它通常与其他结构配合使用。其结构如图 4-16 所示。

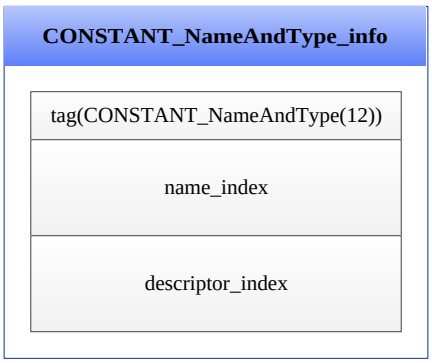

图 4-16 CONSTANT_NameAndType_info 结构

其中，name_index 必须是对一个常量池表项 CONSTANT_Utf8_info 结构的引用，该结构要么表示特殊的方法名<init>，要么表示一个有效的字段或方法的非全限定名；descriptor_index 必须是对一个常量池表项 CONSTANT_Utf8_info 结构的引用，该结构表示一个个有效的字段描述符或方法描述符。

到这里整个常量池中与本文相关的数据结构就都介绍完了。光是讲解数据结构未免枯燥，也未必能建立起其之间的相互关联。在此笔者通过图 4-17 来描述它们之间的联系，希望能够多少让读者感受到这个数据结构如此设计的用意所在。

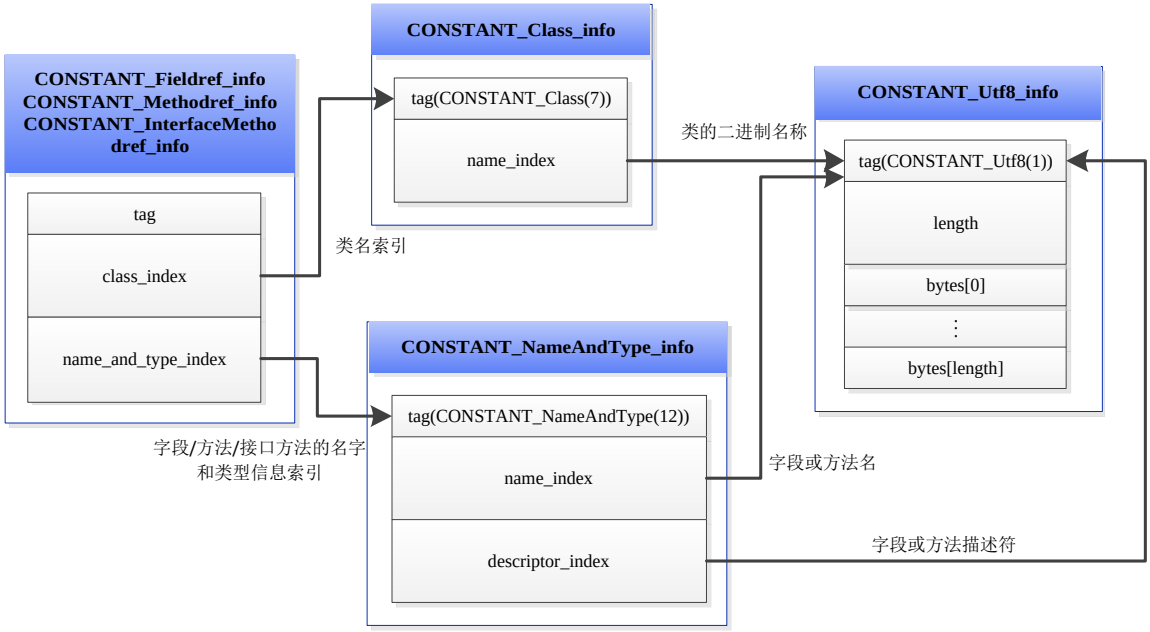

图 4-17 class 文件各结构的相互引用

由上图可以看到，各种常量池结构最终引用到的都是 CONSTANT_Utf8_info 结构，即最终各种信息都是以字符串的信息来表示。比如，对于一个字段引用的结构，它又包含了两个引用，分别指向其声明类的信息描述和自身信息描述，后者又指向其自身的名字和描述符，以此来完整地描述一个字段。方法和接口方法的描述道理是相同的。

细心的读者可能注意到图中对类名的标识是通过其二进制名称，这些名称是语言规范和虚拟机规范为了准确无歧义地引用到一个唯一的字段（方法或其他程序元素）所使用的命名约定。

4.5.3 字段表与方法表

字段结构 field_info 和方法结构 method_info 其内部组成均是相同的，在此将它们合并到一节。读者需要注意，对于字段和方法，其部分结构的取值可能是不同的，比如 tag 值、descriptor_index 的指向（字段结构要求指向一个字段描述符，方法结构要求指向一个方法描述符）等。它们共同的结构如图 4-18 所示。需要注意的是，这个结构与 4.8.3 节提到的 FieldInfo 数据结构类似，但它们是不同层级的数据结构。

field_info/method_info

access_flags	访问标志
name_index	字段/方法名索引
descriptor_index	描述符索引
attribute_count	属性表计数器
attributes[0]	属性表
...	

图 4-18 field_info 及 method_info 结构

4.6 反射部分的 JNI 机制

在 4.3 和 4.4 节中，论文从 Java 标准类库和 Oracle 公司对标准类库的私有实现出发，初步探索了 Java 语言层反射机制的运作原理。论文在 4.3.2 节提出的这些问题依然未得到解答：既然系统的自描述数据不存在于语言层面，那么它们起源于何处？是以何种方式被组织？又是何时何地被创建出来？

在 4.2.1 节中，论文提到通往反射的入口即是通过每个 Object 或其子类的对象来调用其 getClass()方法，拿到一个元层的元对象来操纵元信息。这个方法在 Object.java 的第 64 行：

```
64    public final native Class<?> getClass();
```

但这个方法没有方法体。类似的方法也可以在 Class.java 类中找到，它们是 Java 元信息的重要来源，但是却看不到实现。以下代码节选自 Class.java 类：

```
127         private static native void registerNatives();
```

```
128     static {
129         registerNatives();
130     }
        …
2866        private native Field[]          getDeclaredFields0(boolean publicOnly);
2867        private native Method[]         getDeclaredMethods0(boolean publicOnly);
2868        private native Constructor<T>[] getDeclaredConstructors0(boolean publicOnly);
2869        private native Class<?>[]    getDeclaredClasses0();
```

这是个什么方法？native 这个关键字又代表着什么？为什么要使用这种方式？这个方法存在于何处？4.6.1 节和 4.6.2 节将试图回答以上几个问题。

4.6.1 JNI 机制与本地方法概述

JNI 全称为 Java Native Interface，即 Java 本地接口。这个机制是 Java 平台的一部分，可以实现 Java 代码与其他语言实现（如 C/C++等）进行交互。它扩充了 Java 平台的能力，弥补了其平台无关性带来的对本地特定环境操作的不足。原则上，在需要用到 Java API 不提供的与操作系统平台相关的特性，或者在程序对时间和性能特别敏感的场合下，我们必须使用 JNI 机制来与更底层的语言（如汇编、C/C++等）直接交互[45]。

native 关键字说明其修饰的方法是一个本地方法，其实现体不在声明这个方法的当前文件中，而是在对应的其他语言实现的文件里。准确来说，JDK 的本地代码是采用 C 语言来实现的。

4.6.2 本地方法位置

在下载的 OpenJDK 项目源码的子项目 jdk 中，在路径 src/share 下有两个文件夹，classes 和 natives。在 classes 包中包含了 Java 标准类库的实现和 Oracle 公司对 Java 标准类库的私有实现，在这些代码中可能存在着对本地代码的调用（如 4.6 节中 Object 类和 Class 类中的 native 方法）。这些本地代码大部分可以在 natives 包中对应的目录结构中找到，它们存在于与 classes 包下声明了本地方法的文件同名的 c 文件下。

如今论文的代码分析从 Java 语言层的类库实现，走入了更深处的本地接口层的实现。JNI 机制充当着一个中介者的角色，位于类库实现与虚拟机实现中间。只要分析清楚这个中间层的机制，清楚它如何实现上层类库到下层虚拟机的方法调用的映射，更多背后实现的迷雾将被拨开。不过，要弄清楚这个机制仍然有一些工作要做。论文以 Class.c 的源码开始讲解，见源代码第 87-93 行。

```
87  JNIEXPORT void JNICALL
```

```
88  Java_java_lang_Class_registerNatives(JNIEnv *env, jclass cls)
89  {
90      methods[1].fnPtr = (void *)(*env)->GetSuperclass;
91      (*env)->RegisterNatives(env, cls, methods,
92                              sizeof(methods)/sizeof(JNINativeMethod));
93  }
```

这个方法是原汁原味的 C 语言实现，包含了底层的函数指针、指针类型转换、结构体数组等特性。这个方法就是标准类库中 registerNatives 方法的本地实现了，但方法本身还存在一些现今还不太清楚其作用的元素。论文研究源码实现时，将尝试回答如下的问题：

(1) JNIEnv 是个什么类型的数据结构？它有什么样的作用？

(2) methods 这个结构体数组有什么作用？在整个 Class.c 文件中并没有看到语言层的 Class.java 声明的所有本地方法对应的实现，那么它们的函数体在什么地方呢？

(3) jclass 是个什么类型的变量？在 Java 语言层面调用这个方法的时候并没有传入这两个参数，那么它们是什么时候被注入的？是否用来描述 Java 层的对象？有什么样的作用？

(4) JNIEXPORT 和 JNICALL 看起来是两个宏，它们的作用是什么？

(5) 方法的命名有什么样的规则？是 JNI 机制所强制使用的规则吗？系统是通过命名来识别并应用这些方法的吗？

(6) JNIEnv 的成员函数 RegisterNatives()是如何实现函数“注册”的？它们什么时候被注册？注册信息以什么形式存在？由谁来维护？系统如何找到这些注册的方法？

接下来安排 4.6.3～4.6.6 这 4 个小节来尝试解决第 1 到第 4 个问题，而第 5 个和第 6 个问题将放在 4.8 节来解答，因为它涉及了 JVM 的底层细节。

4.6.3 JNI 中枢——JNIEnv 结构

JNIEnv 是一个线程相关的变量，每一个线程都有一个独立的 JNIEnv 变量。它在 OpenJDK 项目下的 hotspot 项目下的 jni.h 头文件中被声明，具体的路径是 hotspot/src/share/vm/prims/jni.h，见代码第 190-198 行：

```
190     struct JNINativeInterface_;
192     struct JNIEnv_;
193
194     #ifdef __cplusplus
195         typedef    JNIEnv_ JNIEnv;
```

```
196     #else
197         typedef const struct JNINativeInterface_ *JNIEnv;
198     #endif
```

上述代码声明了两个数据结构：JNINativeInterface_和 JNIEnv_。其后又根据项目文件中是否使用 C++（#ifdef __cplusplus 宏），在不同的语言下使用相同的 JNIEnv 名称来指向这两个数据结构。不同的是，C 语言下定义的 JNIEnv 多了一重指针包装，目的在于保留 C 和 C++不同的编码风格。（<jni.h>注释 L770~L780）。在 C 语言中，拿到这个 JNIEnv 还需要对它解封装才能使用：(*env)->function()，而在 C++中，则可以直接使用如下的成员运算符进行操作：env->function()。

其实 JNINativeInterface_和 JNIEnv_本质上是一样的数据结构，JNIEnv_结构中所有的函数及功能都是直接转调 JNINativeInterface_结构来实现的。JNINativeInterface_这个数据结构在<jni.h>文件的第 214 行定义：

```
214  struct JNINativeInterface_ {
215      void *reserved0;
216      void *reserved1;
217      void *reserved2;
218
219      void *reserved3;
220      jint (JNICALL *GetVersion)(JNIEnv *env);
221
222      jclass (JNICALL *DefineClass)
223          (JNIEnv *env, const char *name, jobject loader, const jbyte *buf,
224           jsize len);
     …
```

JNINativeInterface_是一个巨大的数据结构，其声明一共有 550 多行。它预留了 4 个 void 指针的位置，接下来的每个变量都是一个指向 JNI 调用的函数指针。其数据结构如图 4-19 所示。

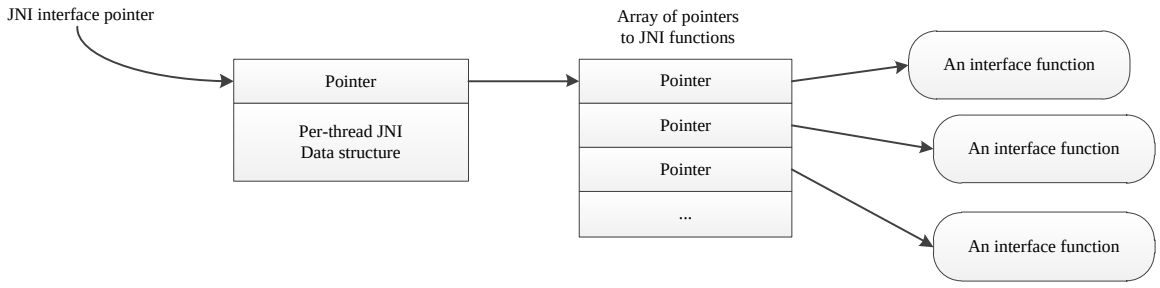

图 4-19 JNINativeInterface 的结构

4.6.4 本地方法注册

在 4.6.3 节中留下了两个问题：

(1) JNINativeInterface_这个结构什么时候被新建？什么时候被填充具体的函数地址？被系统的什么部分使用来寻址？其使用者如何寻址？

(2) 这些方法的具体实现在何处？

首先看一下第一个问题。4.7.2 节中出现的 methods 数组，在<Class.c>文件的第 54 行被定义：

```
static JNINativeMethod methods[] = {
    {"getName0",            "()" STR,           (void *)&JVM_GetClassName},
    {"getSuperclass",       "()" CLS,           NULL},
    …
    {"getDeclaredFields0","(Z)[" FLD,           (void *)&JVM_GetClassDeclaredFields},
    …
};
```

由这个声明可以知道 methods 数组的元素是 JNINativeMethod 类型的变量，它的定义在<jni.h>头文件的第 180 行：

```
180     typedef struct {
181         char *name;
182         char *signature;
183         void *fnPtr;
184     } JNINativeMethod;
```

该结构体定义了三个成员，分别是方法名、方法签名和函数起始地址指针。

在 methods 数组的声明中使用到了一些辅助签名生成的宏，它们在<Class.h>文件的 L44 行定义：

```
44  #define OBJ "Ljava/lang/Object;"
45  #define CLS "Ljava/lang/Class;"
46  #define CPL "Lsun/reflect/ConstantPool;"
47  #define STR "Ljava/lang/String;"
48  #define FLD "Ljava/lang/reflect/Field;"
49  #define MHD "Ljava/lang/reflect/Method;"
50  #define CTR "Ljava/lang/reflect/Constructor;"
51  #define PD  "Ljava/security/ProtectionDomain;"
52  #define BA  "[B"
```

它会将 Java 层同样名称和签名的本地方法“注册”到某个地方，并记录下这个方

法的物理地址。在语言层调用这个本地方法的时候，就通过这个指针去调用函数真实的实现。这个巨大的 JNINativeMethod 结构在虚拟机启动时，会被 JavaMain 方法初始化，并装填好各个方法的真实入口地址[25]。

4.6.5 JNI 数据类型转换

之所以存在这样的数据类型转换，原因在于 Java 语言中的数据类型是一种“规范”，它定义了这种数据结构必须满足的约束（位数、取值范围、运算含义等），只要能满足其外观表现，理论上它可以采用任意一种实现。而 Java 虚拟机本身大部分的代码都是使用 C++写成的，C++对数据模型的实现是接近于机器模型的，虽然它本身也依赖于编译器、处理器字长和操作系统等因素[46]。C++11 标准中仅仅规定了数据类型的取值范围及不同类型之间的大小关系[47]，并未规定具体的实现。具体的约束在<limits>头文件中。

因此，JNI 机制身处 Java 语言层和虚拟机层沟通要道的交点上，自然要为数据类型的转换负起责任。在其头文件<jni.h>中，它通过数据类型重命名的方式来做出数据类型的变换：

```
57        typedef unsigned char     jboolean;
58        typedef unsigned short    jchar;
59        typedef short             jshort;
60        typedef float             jfloat;
61        typedef double            jdouble;
62
63        typedef jint              jsize;
```

而 Java 语言中的 byte、int 和 long 类型实现则根据平台的不同映射到不同的数据类型上。其定义在头文件<jni_md.h>第 27 行。<jni_x86.h>文件的第 57 行则声明了 x86 平台下的 int 和 long 等数据类型的本地映射，它将一个 Java 语言的 jint 类型直接映射为虚拟机平台的整数 int 类型。其他数据类型也是根据平台和环境的不同，分别做了不同的映射。

```
57      typedef int jint;
58      typedef __int64 jlong;
59  #endif
60
61  typedef signed char jbyte;
```

除了基本类型以外，JNI 还对引用类型定义了一个类型体系，如图 4-20 所示。具体的代码在<jni.h>文件的第 65-116 行，有兴趣的读者可以自行参考。

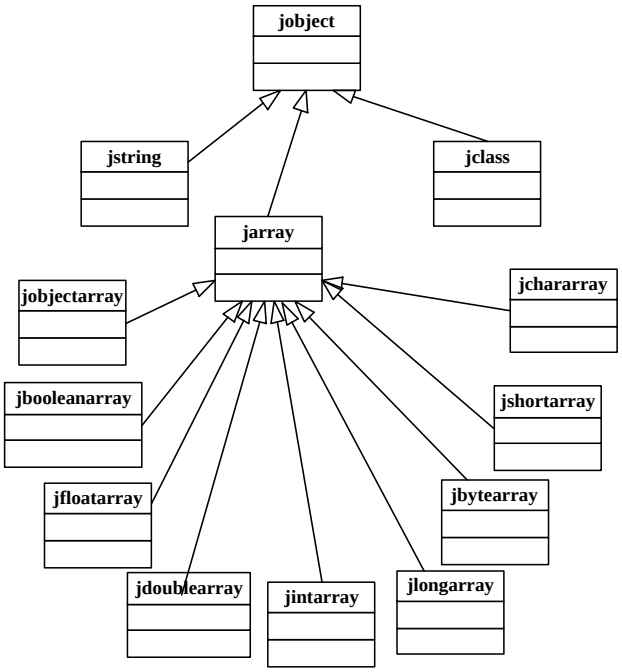

图 4-20 JNI 数据类型体系

4.6.6 JNI 相关的宏定义

这两个宏在<jni_md.h>头文件中被引入，<jni_md.h>又根据平台不同会把不同的头文件包含进来。论文节选了<jni_x86.h>头文件的一段代码，其中根据操作系统和编译器的不同，采用不同的方式来定义 JNIEXPORT 和 JNICALL 两个宏。代码在源文件的第 29-59 行：

```
29  #if defined(SOLARIS) || defined(LINUX) || defined(_ALLBSD_SOURCE)
30
31  #if defined(__GNUC__) && (__GNUC__ > 4)||(__GNUC__ == 4) && (__GNUC_MINOR__ > 2)
32      #define JNIEXPORT       __attribute__((visibility("default")))
33      #define JNIIMPORT       __attribute__((visibility("default")))
34  #else
35      #define JNIEXPORT
36      #define JNIIMPORT
37  #endif
38
39      #define JNICALL
        …
52  #else
53      #define JNIEXPORT __declspec(dllexport)
```

```
54      #define JNIIMPORT __declspec(dllimport)
55      #define JNICALL __stdcall
…
59  #endif
```

在不同的平台下，JNIEXPORT 有不同的宏定义，在 Linux 和 Solaris 等平台下且编译器为 GNU gcc 的环境下，它们被定义为__attribute__((visibility("default")))，而在其他平台下（如 Windows 等），它们被定义为__declspec(dllexport)。定义不同，但其作用是相同的，都用于通知编译器将其所修饰的函数及符号表等信息导出到 lib 库和 dll 文件中，以提供给外部使用。

JNICALL 宏在 Windows 平台下被定义为__stdcall，其他平台下则是空定义，仅起标记作用。这个宏用于告诉编译器函数参数的入栈方式，这些都是比较底层的内容，用于兼容不同平台不同编译器甚至不同的文件（PE/COFF 文件等）环境，对于本文的研究论题不是十分重要，因此在此处仅简单提及。读者可简单认为它们仅起标记函数的作用，并认为标记函数按照其概念模型运行即可。其中，JNIEXPORT 标记着它可以被导出到外部使用，JNICALL 表明函数是一个 JNI 本地函数。

至此，论文回答了 4.6 节开头提出的前 4 个问题。在 JNI 机制下，Java 语言层的标准类库中声明的本地方法都通过调用 RegisterNatives()方法来注册相应的本地方法（见 4.6.2 节），而 RegisterNatives()方法则使用了一个 JNINativeMethod 结构体数组来完成注册。该结构体中存放了每个方法及与其对应的 JVM 层方法的函数地址。在语言层标准类库中的本地方法被调用时，它会通过 JNI 层拿到这个虚拟机 VM 层的函数地址并进行调用。在 4.8 节将讲解一个具体调用的全流程，读者可以大概看到一个调用的每个层面。

4.7 数据结构的核心——oop/klass 二分模型

通过 4.1-4.6 节对标准类库实现、JNI 中间层的讲解，论文的研究终于深入到虚拟机的实现层。OpenJDK 的虚拟机 HotSpot 主要是使用 C++语言进行开发的，部分处理器相关或对性能敏感（如字节码或机器代码生成）的代码是用汇编写成。HotSpot 虚拟机的实现符合 4.5 节描述的虚拟机概念模型，但实现可能与概念模型有所出入[26]。

一个 Java 对象在 JVM 层面是用至少两个对象来表示的，类的实例数据采用一个 instanceOopDesc 类来描述，而类本身使用的是 instanceKlass 来描述。前者只负责存取实例对象的数据，而类的字段和方法等属性是存放在 instanceKlass 中的。

instanceKlass 是一个比较大的数据结构，如图 4-21 所示。它包含了一个普通的 oop 对象的对象头和指向元数据的指针_metadata，紧接着是一个 C++的虚函数表，类的一些属性，以及类字段数、方法数、本地方法入口等信息。

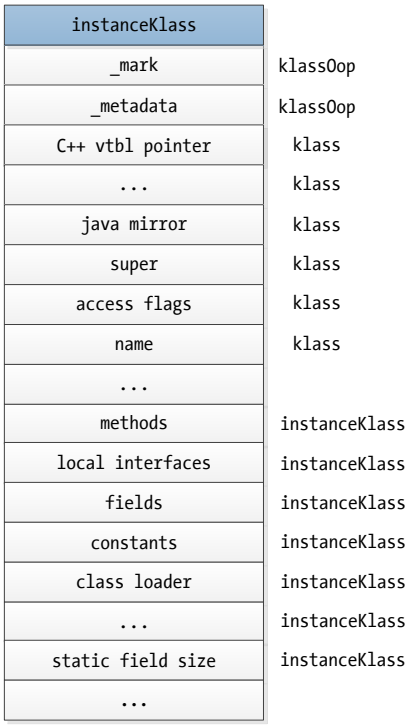

图 4-21 instanceKlass 结构

4.8 一个完整反射调用的实现

具备了前面的铺垫，本节将选取 Class.java 类的一个方法 getDeclaredFields()来讲解一个反射调用的全过程。该方法位于 Class.java 类的第 1801 行。

```
1801    public Field[] getDeclaredFields() throws SecurityException {
        …
1805        checkMemberAccess(Member.DECLARED, Reflection.getCallerClass(), true);
1806        return copyFields(privateGetDeclaredFields(false));
1807    }
```

这个方法用来拿到本 Class 对象所描述类声明的所有字段，包括包可见和私有的字段，但不包含继承于基类或父接口的字段。代码第 1805 行进行了一些权限检查，详细的安全检查见 4.3.5 节。然后，代码第 1806 行又调用了 privateGetDeclaredFields()，该方法位于 Class.java 类的第 2380 行，在此节选方法体中的一部分代码：

```
2380    private Field[] privateGetDeclaredFields(boolean publicOnly) {
2381        checkInitted();
2382        Field[] res = null;
2383        if (useCaches) {
                …// 省略的代码，返回已缓存的 Fields
2395        }
```

```
2396            // 若无缓存 Fields 对象，向 VM 发起请求
2397            res = Reflection.filterFields(this, getDeclaredFields0(publicOnly));
                // 省略的代码，缓存取得的 Fields 以供下次调用快速返回
2405            return res;
2406    }
```

代码第 2397 行中又进一步调用了 getDeclaredFields0()方法，并向 Reflection 类请求过滤掉获得的字段对象的数组。。

getDeclaredFields0()方法在 Class.java 类的第 2866 行声明：

```
2866    private native Field[]          getDeclaredFields0(boolean publicOnly);
```

它是一个本地方法，其方法的实现体已脱离本 Class.java 文件的范围。在 OpenJDK 项目的 jdk/src/share/natives/java/lang 路径下可以找到其实现文件 Class.c。其向 JNI 机制的注册代码在第 54 行：

```
54  static JNINativeMethod methods[] = {
    …
65  {"getDeclaredFields0","(Z)[" FLD,           (void *)&JVM_GetClassDeclaredFields},
…
77  }
```

可见，其实现最终调用到 VM 层的 JVM_GetClassDeclaredFields 函数。函数声明在<jvm.h>头文件中，该文件在 OpenJDK 项目的 hotspot/src/share/vm/prims 路径下。声明代码在第 544 行：

```
543 JNIEXPORT jobjectArray JNICALL
544 JVM_GetClassDeclaredFields(JNIEnv *env, jclass ofClass, jboolean publicOnly);
```

该函数实现在<jvm.cpp>文件的第 1729 行，其函数体有 60 行之多，此处只给出它实现的流程图，如图 4-22 所示。

4.8.1 JVM 实现相关的宏定义

在代码第 1729 行中，函数的开头是个奇怪的形式：JVM_ENTRY(jobjectArray, JVM_GetClassDeclaredFields(JNIEnv *env, jclass ofClass, jboolean publicOnly)){…。读者可能会感到疑惑，这个宏 JVM_ENTRY 是干什么的？可以声明这种形式的函数吗？

问题的解答还在这个宏本身的定义。JVM_ENTRY 宏在 hotspot 模块 src/share/runtime 路径下的 interfaceSupport.hpp 头文件中被声明，代码从第 579 行开始：

```
579 #define JVM_ENTRY(result_type, header)                                  \
580 extern "C" {                                                            \
581     result_type JNICALL header {                                        \
```

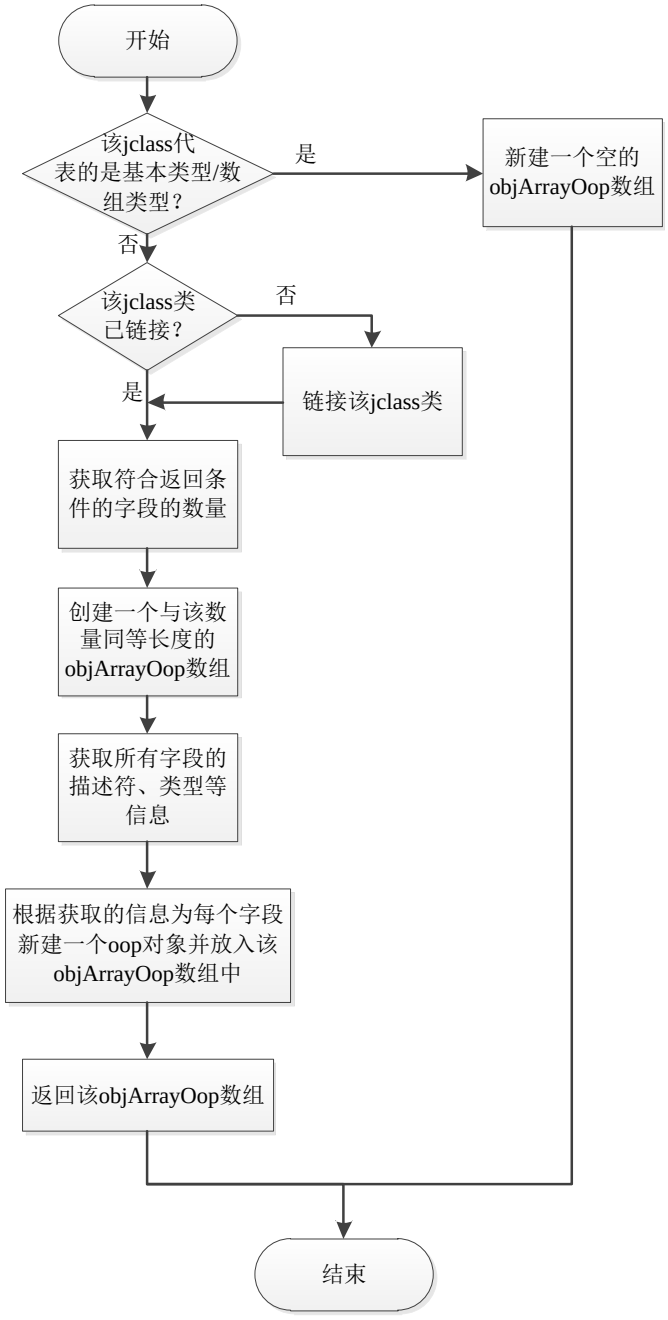

图 4-22 JVM_GetClassDeclaredFields 函数流程图

```
582     JavaThread* thread=JavaThread::thread_from_jni_environment(env); \
583     ThreadInVMfromNative __tiv(thread);                              \
584     debug_only(VMNativeEntryWrapper __vew;)                          \
585     VM_ENTRY_BASE(result_type, header, thread)
```

其传入的两个参数 result_type 和 header 分别是函数的返回值和函数声明。见代码第 581 行，这行代码：JVM_ENTRY(jobjectArray, JVM_GetClassDeclaredFields(JNIEnv *env, jclass ofClass, jboolean publicOnly)){被宏展开后将变成：

```
extern "C" {
    jobjectArray JNICALL JVM_GetClassDeclaredField(JNIEnv* env, jclass ofClass, jboolean
publicOnly) {
    …
```

就变成正常的函数声明形式了。宏的其他部分还声明了一些提供做 JNI 接口的 JVM 函数共有的特征，比如是否支持调试等。这些论文不做深入探讨，有兴趣的读者可以自行参阅 HotSpot 源码。

另外，JVM_END 宏的定义则非常简单，在<interfaceSupport.hpp>头文件的第 613 行。它用于关闭 extern “C”区块和函数声明：

```
613       #define JVM_END } }
```

4.8.2 简单封装——句柄设计模式

handle 实质上是对对象引用的一层简单封装，是个中间层。可以看见在这段代码中并没有直接对 oop 对象或者 klass 对象进行操作，而是把各种 oop 对象和 klass 对象作为参数传入到相应的 handle 对象中，再对 handle 进行操作。这样设计的好处的主要原因是要与虚拟机的 GC 子系统（Garbage Collection，垃圾回收）更好配合[48]。当 GC 要了解是否有需要回收的 oop 或 klass 对象，只需要扫描整个 handle 表即可；其次，GC 的工作可能会改变一些对象在内存中的位置，而使用了 handle 则只需要更改其中持有的对象指针即可，客户端的调用代码不会受到影响[49]。

那么这个 handle 体系的定义在何处呢？笔者使用了各种方法，比如查看<jvm.cpp>引用的所有头文件、使用 Everything 等搜索软件搜索并查看 OpenJDK 项目下所有名字带有“handle”的文件、查看所有系统的关键文件（如 systemDictionary.hpp、symbolDictionary.hpp、oopsHierarchy.hpp 等）、百度+谷歌等方法，均没有找到 handle 体系的定义，也没有找到任何类型重定义（typedef）的线索。笔者仔细思考了一下，最后终于在 hotspot 项目下的 src/share/vm/runtime 路径上找到了这个文件：Handles.hpp。handles 体系的定义在源代码的第 163 行：

```
163  #define DEF_HANDLE(type, is_a)                                  \
164     class type##Handle;                                          \
165     class type##Handle: public Handle {                          \
166       protected:                                                 \
167         type##Oop       obj() const              { return (type##Oop)Handle::obj(); } \
169         type##Oop non_null_obj() const {return (type##Oop)Handle::non_null_obj();}\
170         …
```

宏定义，它的定义是通过字符串拼接而成的。代码的第 190 行开始使用了这个宏：

```
190  DEF_HANDLE(instance              , is_instance              )
191  DEF_HANDLE(method                , is_method                )
192  DEF_HANDLE(constMethod           , is_constMethod           )
193  DEF_HANDLE(methodData            , is_methodData            )
194  DEF_HANDLE(array                 , is_array                 )
195  DEF_HANDLE(constantPool          , is_constantPool          )
     …
```

以第 190 行的代码为例，展开后的效果即是：

```
class instanceHandle;
class instanceHandle: public Handle {
 protected:
   instanceOop    obj() const { return (instanceOop)Handle::obj(); }
   instanceOop non_null_obj() const { return (instanceOop)Handle::non_null_obj(); }
…
```

4.8.3 字段迭代器和信息容器——JavaFieldStream 和 FieldInfo

在代码的第 1756 和 1773 行，一个信息流类 JavaFieldStream 被用于迭代取出每个字段的相关信息。它继承自 FieldStream 这个基类。该类用来迭代类已定义的字段数组，根据所需迭代类型的不同，它提供了不同的子类可供实现，其中 JavaFieldStream 用于迭代普通的 Java 字段，是更被鼓励使用的迭代器。InternalFieldStream 用于取得 JVM 插入的字段，而 AllFieldStream 仅在极少数的情形下需要使用，它可以取得所有类型的字段。

JavaFieldStream 的工作原理如图 4-23 所示，从其调用者来看，它只是一个迭代器，提供 next()、done()等方法来取得下一个索引、指示迭代结束等。同时，可以取得迭代器在迭代索引处的值，也就是一个字段。而从 JavaFieldStream 的内部构造来看，关于字段的信息它全部都委托给 FieldInfo 这个类来获取。它们的分工很明确，JavaFieldStream 只是一个迭代器，其迭代的对象 FieldInfo 才是真正的信息容器。

那么 FieldInfo 是如何维护字段信息的呢？在其文件<fieldInfo.hpp>第 46-57 行声明了如图 4-24 所示的这个数据结构。

这个数据结构出现了我们关心的一些信息，如访问标志、字段名和字段签名（其中又包含字段类型信息）等。但它是一个 16 位的无符号整数，明显不可能存放一个完整的字段值（如 int 型、long 型整数等）。正如该字段名所暗示，它仅是一个“索引”（index），那么它又是如何使用这个索引来获得字段真实的信息（字段名，字段值等）的呢？其最终的数据来源又是何处呢？要回答这个问题，还必须从源代码查起。见源

代码第 90 行的 name()方法。

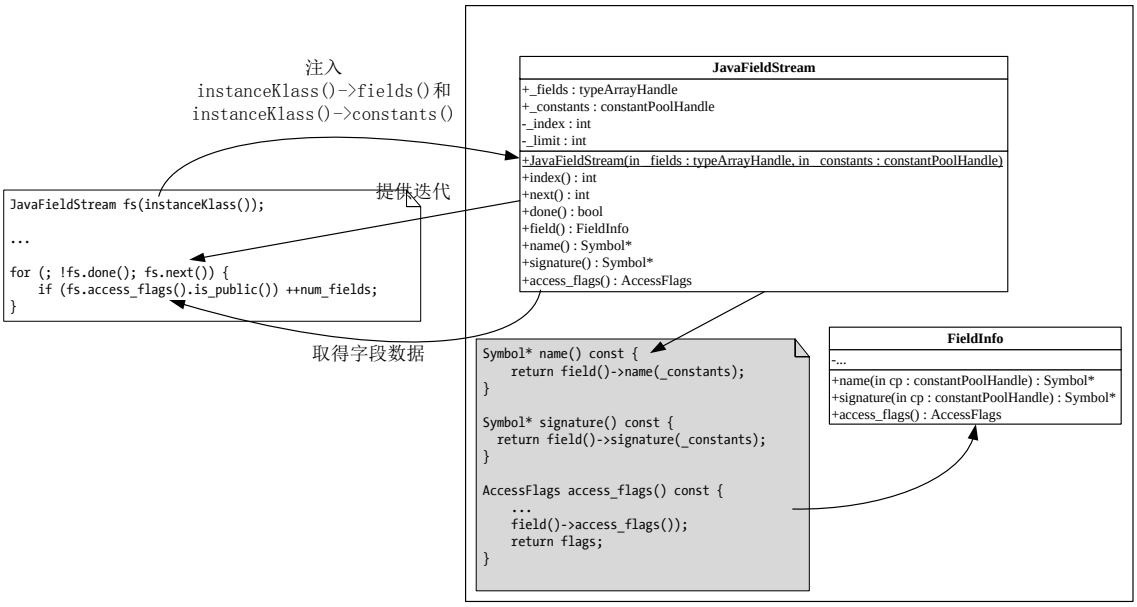

图 4-23 JavaFieldStream 模块提供的使用接口

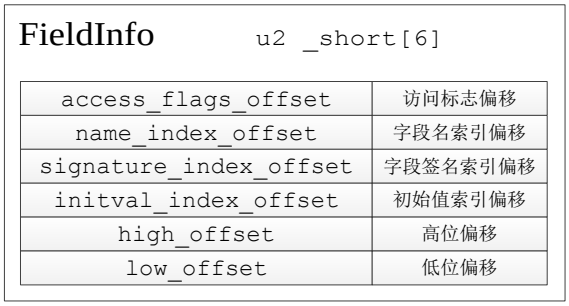

图 4-24 信息容器 FieldInfo 的结构

```
90      Symbol* name(constantPoolHandle cp) const {
91        int index = name_index();
92        if (is_internal()) {
93          return lookup_symbol(index);
94        }
95        return cp->symbol_at(index);
96      }
```

可以看见方法参数中注入了一个 constantPoolHandle 变量，在 JavaFieldStream 的工作原理图中可以看到这个 constantPoolHandle 字段正是 JavaFieldStream 注入进来的。如果该字段不是 JVM 插入的字段，那么代码第 95 行将使用该常量池句柄的 symbol_at(index)方法去获取真正的字段名，返回一个 Symbol 类型的指针。真正的数据源，来自运行时的常量池。运行时的常量池中存储了由 class 文件加载得到的类元信息，并在类加载阶段创建并准备好相应的 oop 和 klass 对象。

4.9 本章小结

本章总结了介绍了 OpenJDK 项目的源代码结构，给出了一个反射调用涉及的五个层面：标准类库、标准类库的实现、JNI 本地实现、JVM 实现及最底层的数据结构。以下各节分述了这五个层面。4.2～4.3 节讲述了标准类库相关知识；4.4 节讲解了标准类库的内部实现；4.5 及 4.7 节讲解了用于保存各种元数据的编译期及运行期数据结构：class 文件及 oop/klass 二分模型；4.6 节讲解了反射调用过程涉及的 JNI 部分；4.8 节讲解了 JVM 层面的实现。

结　　论

论文一共完成了以下工作：

(1) 一般性反射模型的研究。这部分是研究 Java 反射模型的基础，论文主要参考并给出了一个关于反射的定义、总结了 B.C.Smith 博士提到的反射模型的三个要素：自描述、casual connection 和安全性，研究了反射的两种分类：结构反射和计算反射以及它们的实现和对比；

(2) Java 反射模型的研究。论文从模型的体系架构和实现两方面进行了研究。基于一个一般性反射模型的研究，论文研究了 Java 反射机制的基本模型、其元数据的组织、其简单连系的实现，总结了 Java 元对象协议的静态层面和动态层面。论文研究并给出了由 Java 标准类库层面一直贯通到 JVM 层面的反射实现，研究了 Java 反射从调用到实现的全过程。内容涉及 Java 语言层面的存取器模式实现、安全与权限检查、元数据在 class 静态文件和运行时核心数据结构 oop/klass 二分模型中的组织、JNI 层面对本地方法的注册以及查找和调用机制，以及 JVM 层面最底层的实现。

由于专业能力和时间的限制，本论文还存在如下不足：

(1) 论文对使用 Java 反射实现的框架进行搜集研究的样本还不够，以至于最后删去了这部分。如果能把 Java 反射放到实践中去检验它的性能和使用，可以更贴合软件开发现状地对它进行更深入的研究；

(2) 论文对 Java 反射模型的理论研究还达不到融会贯通的高度。在这个领域有很多论文，涉及很多层面，包括结构反射和计算反射这些和本文比较贴合的方面。如果能够更好地理解已经被提出的一些理论和实现方法的异同，或可作为本文更加厚重的理论支撑；

(3) 笔者对底层技术的掌握还不够深入。在 Java 反射机制这个题目上，如果能够具有良好的汇编、操作系统、编译原理、计算机体系结构以及脚本编写等各方面的基础，那么题目可以做到虚拟机的字节码执行引擎以及平台相关的内存实现模型上。限于能力不足，论文还未能通过反汇编虚拟机的指令集和机器指令来观察其执行引擎；

(4) 验证部分还不够全面和深入。因为笔者在搭建虚拟机上花费了很多时间，最后也没有成功，多少影响了本论文关于底层机制实现验证的数据支撑强度。好在笔者还有其他工具来完成对反射机制关键部分的内存查看和验证。

综上，论文研究了基本的反射模型，分析了 Java 的反射模型。论文研究了 Java 反射机制实现涉及的一些方面，并挖掘代码对部分方面的实现做了分析。

致　　谢

毕业设计的这半年有太多的事情值得感恩感谢。

感谢我的导师胡晓鹏。去年 11 月份我就自己拿着几个题目去找老师商讨毕业设计的选题，老师肯定了 Java 反射机制这个选题的意义，没有老师这半年我就没有机会做我真正喜欢的底层工作。在这个过程老师给予了我和我们小组的成员很多宝贵的意见和支持，也留下了很多很多洞见。同时，老师对学术的较真和敏锐的感觉，真的是我学习过程中感到被支持的动力。

感谢我的老爸林少文。你关心着我大四以后的两件大事，找工作和毕业设计，并针对我自身的情况给予了很多无比珍贵的建议。你给予了我足够的空间和自由去做我喜欢的事情，后来我才慢慢发现这需要很大的宽容和爱。同样的包容还发生在我的家庭中，我看事角度有时过于单一理想化，又有一些锐气，妈妈个性也比较分明，是爸爸你一直承担家里有时意见的冲突，宽容着包容着我们的个性。这些爱和不易让我觉得你是世界上最伟大的爸爸，我永远爱你，也想要自己逐渐奋斗成长到足够去给你们爱和保护。谢谢你。

感谢我的妈妈林洁珊。妈妈乐观的生活态度一直给我鼓励和喜悦。你教会我如何去乐观地对待生活，尽管同时要有一个悲观的世界观。能够给予我亲近的爱，其实也是在教我如何爱人和接受爱。你和爸爸这个温暖的家是我觉得永远感恩的福分，同时也希望与你一起经历苦乐，有能力为家里承担些许东西。谢谢妈妈，我也爱你。

谢谢我的女朋友小耳朵。四年的异地恋我们一直支持彼此，这和她的理解包容是分不开的。燕玲同学心意善良，善于自省和思考，同时身体力行地给出别人爱和温暖。在与你相处的过程中我发现也改进了很多自身的东西，发现自己自我的很多方面，是你让我及时觉察，也是你在我因为工作烦躁的时候一直鼓励我让我平静给我关心。这篇论文的完成，也倾注了你的心意。谢谢你，有你是我的幸运。

谢谢我们小组的好队友，每个星期的笔记都整理得很给力，学习氛围也很融洽。

谢谢同个寝室的仨宿友。是你们接触最日常真实的我，在我因为工作进度不好烦躁的时候和我一起说话开玩笑，有了好的软件或者技术文章也会无私分享。谢谢你们。

谢谢远在 Pune 的 Dhanesh，Juhi 以及 Melbourne 的 Michal。你们给了我最真诚的 feedback，表达你们对我的喜欢。我感受到这份肯定，会一直记住，十分感谢。

谢谢苏打绿、木苜和贺斯琴。是你们让我一直真诚地面对自己，青峰的歌经常触动我心里真诚文艺的一部分。你们给的温暖和勇气我一直收着，谢谢相遇。

谢谢这半年所有的缘分。

参考文献

[1] TIOBE Index for June 2015[OL]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2015.

[2] Oracle Technology Network. Java SE Technologies[OL]. http://www.oracle.com/technetwork/java/javase/tech/index.html, 2014.

[3] Ira R.Forman, Nate Forman. Java Reflection in Action[M]. Manning Publications Co., 2005.

[4] Brian Foote, Ralph E.Johnson. Reflective Facilities in Smalltalk-80[C]. ACM Sigplan Notices, 1989: 327-335.

[5] Microsoft Developer Network. Reflection(C# and Visual Basic)[OL]. https://msdn.microsoft.com/en-us/library/ms173183.aspx, 2013.

[6] Google The Java Tutorials. Trail: The Reflection API[OL]. http://docs.oracle.com/javase/tutorial/reflect/index.html.

[7] Brian C.Smith. Reflection and Semantics in a Procedural Language[D]. Ph.D. thesis, Massachusetts Institute of Technology, LCS TR-272, 1982.

[8] Daniel P.Friedman, Mitchell Wand. Reification: Reflection without Metaphysics [C]. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, 1984: 348-355.

[9] Pattie Maes. Concepts and Experiments in Computational Reflection[J]. OOPSLA'87 Conference Proceedings, 1987: 147-155.

[10] Pattie Maes. Computational Reflection[J]. The Knowledge Engineering Review, 1988, 3: 01-19.

[11] Jacques Ferber. Computatinal Reflection In Class-Based Object-Oriented Programming[C]. ACM Sigplan Notices, 1989: 317-326.

[12] Ira R.Forman, Scott H.Danforth. Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming[M]. Addison Wesley Longman, Inc, 1998.

[13] Danforth, S.H., I.R.Forman. Reflections on Metaclass Programming in SOM[C], ACM Sigplan Notices, 1994: 440-452.

[14] Danforth, S.H., I.R.Forman. Derived Metaclasses in SOM[C], ACM Sigplan Notices, 1994: 63-73.

[15] Forman, I.R., S.H.Danforth. Inheritance of metaclass constraints in SOM[C]. Proceedings, Reflection, 1996.

[16] Forman, I.R., S.H.Danforth, H.H.Madduri. Composition of Before/After Metaclasses in SOM[C], ACM Sigplan Notices, 1994: 427-439.

[17] Kiczales, G., J. des Rivieres, D.G.Bobrow. The Art of Metaobject Protocol[M]. The MIT Press, 1991.

[18] Stanley B.Lippman. Inside the C++ Object Model[M], Addison-Wesley, 1996.

[19] Gosling, James. The Java Language Specification, SE7 Edition[S]. Addison-Wesley Professional, 2000.

[20] Tim Lindholm. The Java Virtual Machine Specification, SE7 Edition[S]. Pearson Education, 2014.

[21] Liang Sheng. The Java Native Interface: Programmer's Guide and Specification [S]. Addison-Wesley Professional, 1999.

[22] 曹玲. 基于元层技术的反射分析方法研究[D]. 硕士论文. 河海大学, 2003.

[23] 李冰. 反射的分布式元对象协议的设计与实现[D]. 硕士论文. 天津大学, 2004-1.

[24] 周志明. 深入理解 Java 虚拟机：JVM 高级特性与最佳实践[M], 第 2 版. 机械工业出版社, 2014-7.

[25] 陈涛. HotSpot 实战[M]. 人民邮电出版社, 2014-3.

[26] 莫枢. Java Program in Action[OL]. http://elastos.org/elorg_files/FreeBooks/java/JVM%E5%88%86%E4%BA%AB20100621.pdf, 2010[2015-04-20].

[27] 侯捷. Java 反射机制[DB/OL]. http://jjhou.boolan.com/javatwo-2004-reflection.pdf, 2004.

[28] Wikipedia. List of Java Virtual Machines[OL]. https://en.wikipedia.org/wiki/List_of_Java_virtual_machines, 2014.

[29] 侯捷. 庖丁解牛:侯捷自序[OL]. http://jjhou.boolan.com/jjwbooks-tass-foreward.htm, 2001[2015-04-20].

[30] P.Maes, D.Nardi edit. Meta-Level Architectures and Reflection[M]. North Holland, 1988.

[31] L.Steels. Meaning in Knowledge Representation[M] in [30]: 51-59.

[32] Kim Mens, Sebastián González. Computational Reflection and Context-Oriented Programming[OL]. https://released.info.ucl.ac.be/courses/r+cop/material/04-T-Reflection%20in%20Smalltalk.pdf, 2012.

[33] David. VM Types For SmallTalk Objects[OL]. https://code.google.com/p/strongtalk/wiki/VMTypesForSmalltalkObjects, 2007.

[34] John Rose, J. Duke. CompressedOops[OL]. https://wiki.openjdk.java.net/display/HotSpot/CompressedOops, 2012.

[35] P. Cointe. A Tutorial Introduction to Metaclass Architecture as Provided by Class Oriented Lauguages[C]. Proceedings of the International Conference on Fifth Generation Computer Systems, 1988: 592-608.

[36] Brian Cantwell Smith. On The Origin of Objects[M]. MIT Press, 1996.

[37] 莫枢. OpenJDK 源码阅读导航[OL]. http://rednaxelafx.iteye.com/blog/1549577, 2012.

[38] Oracle Java SE Documentation. Enhancements to the Java Reflection API[OL]. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/enhancements.html, 2011.

[39] Joe Darchy. OpenJDK State[OL]. https://blogs.oracle.com/darcy/resource/OSCON/oscon2011_OpenJDKState.pdf, 2011.

[40] Jon Masamitsu. JEP 122: Remove the Permanent Generation[OL]. http://openjdk.java.net/jeps/122, 2010.

[41] Pardeep Kumar. Metaspace in Java 8[OL]. http://java-latte.blogspot.in/2014/03/metaspace-in-java-8.html, 2014.

[42] 莫枢. [资料整合]Oracle HotSpot VM 计划移除 PermGen[OL]. http://rednaxelafx.iteye.com/blog/905273, 2011.

[43] http://hg.openjdk.java.net/jdk7u/jdk7u/raw-file/tip/README-builds.html[OL], 2011.

[44] The Unicode Consortium. http://www.unicode.org[OL], 2015.

[45] 宁静致远. Java 基础之理解 JNI 原理[OL]. http://www.cnblogs.com/mandroid/archive/2011/06/15/2081093.html, 2011-6.

[46] Jérôme. What does the C++ standard state the size of int, long type to be?[OL]. http://stackoverflow.com/questions/589575, 2009-2.

[47] Vijay. Does the size of an int depend on the compiler and/or processor[OL]. http://stackoverflow.com/questions/2331751, 2010-2.

[48] 莫枢. 找出栈上的指针/引用[OL]. http://rednaxelafx.iteye.com/blog/1044951, 2011.

[49] 莫枢. 答“有关 JVM 中一些基本数据结构的疑问”[OL]. http://hllvm.group.iteye.com/group/topic/37605, 2013.

[50] Martin Fowler. Refactoring: Improving the Design of Existing Code[M]. 熊节. Pearson Education, 2010.

[51] Pierre Cointe. Metaclasses are First Class: the ObjVlisp Model. OOPSLA’87 Conference Proceedings, 1987: 156-167.