Android Dalvik虚拟机结构及机制剖析

——第2卷 Dalvik虚拟机各模块机制分析

吴艳霞 张国印 编著

清华大学出版社

Android Dalvik虚拟机结构及机制剖析

——第2卷 Dalvik虚拟机各模块机制分析

吴艳霞 张国印 编著

清华大学出版社

内 容 简 介

本系列丛书共分 2 卷，本书为第 2 卷，在第 1 卷的基础上，采有情景分析的方式对 Android Dalvik 虚拟机的源代码进行了有针对性的分析，围绕类加载、解释器、即时编译、本地方法调用、内存管理及反射机制等功能模块展开分析，主要帮助读者从微观上更深入地理解 Dalvik 虚拟机中各功能模块的实现原理及运行机制。

第 2 卷共 6 章，第 1 章介绍类加载机制，包括其整体的工作流程和机制，详细讲解了其中的三个阶段，并以一个实例验证了源码分析的结果；第 2 章介绍了 Dalvik 虚拟机中至关重要的内存管理机制，详细讲解了其实现的两种算法；第 3 章分析了 JNI 模块的实现原理，在分析源码的基础上，细致入微地介绍了为何用 JNI 编程会提升程序的执行效率；第 4 章以反射机制的一个代码示例开始，介绍了其涉及的 API，并从宏观到微观详细介绍了反射机制；第 5 章介绍了实现解释器的两种不同的技术，比较了 Fast 解释器和 Portable 解释器的不同及各自的优劣势，第 6 章从介绍最近在解释器中非常火的 JIT（即时编译）开始，到 JIT 的所谓的前端分析，再到 JIT 的后端代码生成，为本书画上一个圆满的句号。

通过阅读本书，读者可以了解 Dalvik 虚拟机在 Android 应用程序运行过程中所扮演的重要角色及其不可替代的价值。通过阅读本系列丛书，读者可以对 Android 应用程序的执行过程有更加细致的了解，可以帮助读者优化自己编写的应用程序，更加合理地设计应用程序结构，有效提高应用程序的运行速度。

图书在版编目（CIP）数据

Android Dalvik 虚拟机结构及机制剖析. 第 2 卷，Dalvik 虚拟机各模块机制分析/吴艳霞，张国印编著. --北京：清华大学出版社，2014
ISBN 978-7-302-36108-4

Ⅰ. ①A… Ⅱ. ①吴… ②张… Ⅲ. ①移动终端－应用程序－程序设计－虚拟处理机－研究 Ⅳ. ①TP338

中国版本图书馆 CIP 数据核字（2014）第 069718 号

责任编辑：袁勤勇　薛　阳
封面设计：傅瑞学
责任校对：焦丽丽
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：10.75　　**字　　数**：262 千字
版　　次：2014 年 8 月第 1 版　　**印　　次**：2014 年 8 月第 1 次印刷
印　　数：1～2000
定　　价：29.00 元

产品编号：056943-01

前言

读者在看过第1卷之后，应该已经大致了解了Dalvik虚拟机，包括其模块组成、Dex文件格式以及使用到的工具。本书作为第2卷，在第1卷的基础上，将要细致地分析Dalvik内部的组成原理，采用情景分析的方式，有针对性地分析Android Dalvik虚拟机的源代码，围绕类加载、解释器、即时编译、本地方法调用、内存管理及反射机制等功能模块逐个击破，帮助各位读者从微观上更深入地理解Dalvik虚拟机中各功能模块的实现原理及运行机制。在其中展示的所有源码，都来自于Android 4.0.4的源码，并在此基础上添加了些许注释，以便理解源码。源码篇幅可能有点长，但大多数都是其中最主要的内容，如果读者手边有源码，可以对照着看，效果更好。

此时，在翻开这第2卷时，您手边应该已经准备好了一个能正常调试Dalvik的环境。现在，就进入本卷，边学习边实践，揭开Dalvik内部的神秘面纱。

全书共分为6章：

第1章介绍类加载机制，包括其整体的工作流程和机制，详细讲解其中的三个阶段，并以一个实例验证了源码分析的结果；

第2章介绍Dalvik虚拟机中至关重要的内存管理机制，详细讲解其实现的两种算法；

第3章分析JNI模块的实现原理，在分析源码的基础上，细致入微地介绍为何用JNI编程会提升程序的执行效率；

第4章以反射机制的一个代码示例开始，介绍其涉及的API，并从宏观（实现的三个模块）到微观（具体实现细节）详细介绍了反射机制；

第5章介绍实现解释器的两种不同的技术，比较Fast解释器和Portable解释器的不同及各自的优势和劣势；

第6章从介绍最近在解释器中非常火的JIT（即时编译）开始，到JIT的所谓的前端分析，再到JIT的后端代码生成，为本书画上一个圆满的句号。

本书主要由哈尔滨工程大学吴艳霞、张国印负责编写，参与本书编写和校核工作的还有汪永峰、王彦璋、谢东良、于成、张婷婷、苗施亮、许圣明、檀凯，这里对他们的辛勤工作表示衷心的感谢。

本书主要是针对高级Android应用开发工程师、Android系统开发工程师、Android移植工程师及对Android Dalvik虚拟机源码实现感兴趣的读者参考使用。

编　者

2014年5月

目 录

第 1 章

类加载模块的原理及实现

本章内容提要

- 什么是类加载机制?
- 类加载机制具有怎样的功能?
- 类加载的输入输出是什么?
- 类加载机制的工作流程是什么?
- 类加载机制的函数具体实现是怎样的?
- 类加载机制是如何与解释器进行交互的?

本章主要围绕以下 6 点问题展开讨论,1.1 节主要回答了什么是类加载、类加载的具体功能以及机制的输入与输出;1.2 节从一个整体宏观的角度介绍了类加载机制的整体工作流程,给出了类加载机制的工作阶段划分;在接下来的三节中,分别对类加载机制的三个关键阶段进行详细的分析阐述,详细讲述了三个工作阶段的功能原理以及相关的函数源码分析;最后一节将会通过一个虚拟机运行实例,介绍类加载机制的产物是如何被解释器引用并执行,并从这一角度展示类加载机制与其他功能模块交互的方式。

1.1 类加载机制概述

谈起类加载,想必大多数读者比较陌生,类加载机制究竟具有怎样的功能以及它在虚拟机中是如何工作的是本章具体讨论的内容。

我们都知道程序是由机器指令和数据组成的,对于一个真实的机器,通常由人工将指令和数据交付给机器,再由机器按照指令去对数据进行计算。但是对于虚拟机来说,在其模拟真实机器执行程序之前,需要一种加载机制将程序的指令和数据装载进入虚拟机内部的运行时环境,使虚拟机中的执行模块可以根据程序执行的需要随时取得目标指令和相关数据,以完成程序的执行任务。对于 Java 虚拟机而言,这一种将类数据装载进入虚拟机运行时环境的过程就称为类加载。具体到 Dalvik 虚拟机,类加载机制的主要功能就是将应用程序中 Dalvik 操作码以及程序数据提取并加载到虚拟机内部,以保证程序的正确运行。

类加载机制在应用程序和执行模块之间建立起了一个桥梁,对于整个 Dalvik 虚拟机架构来说,类加载机制处在一个承上启下的位置。图 1.1 反映了类加载机制在 Dalvik 虚拟机执行过程中所承担的重要作用。

类加载机制作为 Dalvik 虚拟机一个重要的功能模块,其输入输出是什么呢? 在第 1 卷第 3 章中,已经对 Dex 文件进行了介绍,在此就不再赘述。简单来说,Dex 文件就是 Android

应用程序的可执行文件，里面封装了 Dalvik 字节码以及程序数据，而类加载机制就是负责提取装载 Dex 文件中的指令与数据。因此，Dex 文件是 Dalvik 虚拟机的输入文件，更是类加载机制的主要处理对象。既然 Dex 文件是类加载机制的输入文件，那么类加载机制的输出又是什么呢？事实上，类加载机制的输出是一个名为 ClassObject 的数据结构实例对象，类加载将目标类的各项资源数据与 ClassObject 数据结构下的各个成员变量以指针的形式进行关联，执行模块只需通过该数据结构即可获得目标类所有的运行时数据，使程序的顺利执行成为可能。

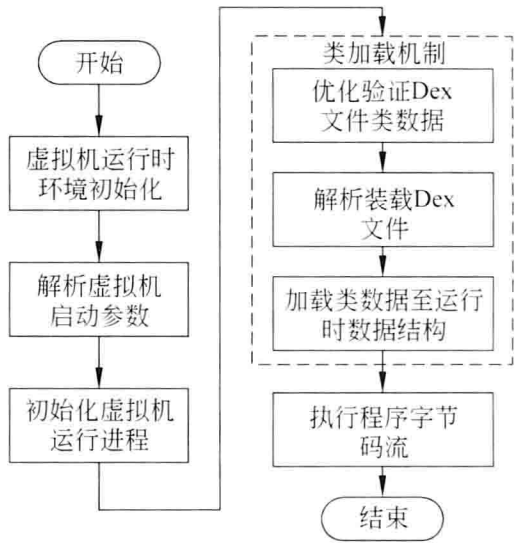

图 1.1　Dalvik 虚拟机程序执行流程图

点拨　程序是由指令序列组成的，告诉计算机如何完成一个具体的任务。程序是软件开发人员根据用户需求开发的、用程序设计语言描述的适合计算机执行的指令(语句)序列。而对于 Dalvik 虚拟机来说，它所能“读懂”的指令是 Dalvik 操作码，其介绍可以参见第 1 卷第 3 章。

1.2　类加载机制整体工作流程介绍

在了解了类加载机制的主要功能以及输入输出产物后，我们不禁要问，类加载机制在工作过程中主要有哪几项关键工作以及其工作流程又是什么？经过对类加载机制源码的分析研究，类加载机制工作的主要内容以及其整体的工作流程主要分为以下三点。

(1) 对 Dex 文件进行验证并优化，验证的目的是对 Dex 文件中的类数据进行安全性、合法性检验，为虚拟机的安全稳定运行提供保证；而优化的目的则是根据当前设备平台特性对程序中的字节码进行优化替换并为 Dex 文件增加辅助信息，最后输出经过优化的 Odex 文件，使之可以代替原有的 Dex 文件更加高效地被虚拟机执行。

(2) 对优化后的 Odex 文件进行解析，其目标就是通过在内存中创建专用的数据结构描述表示该 Odex 文件，使虚拟机对目标 Odex 文件中各个部分的类数据都是可达的，为随后实际加载某一指定类做好数据准备。

(3) 对指定类进行实际加载，其功能是实时根据 Dalvik 虚拟机执行需要从已被解析的 Dex 文件中提取二进制 Dalvik 字节码并将其封装进运行时数据结构，以供解释器解释执

行。而该运行时数据结构实际上是一个 ClassObject 结构体对象，也称为类对象，该数据结构用于封装程序类的所有运行时数据信息。当虚拟机执行一个类方法时，解释器将引用并执行类对象中封装的方法操作码，进而达到完成程序要求的执行目标。由此可见，类对象实例在程序运行的过程中承担着不可替代的重要作用。

图 1.2 简要描述了类加载机制的整体工作流程。

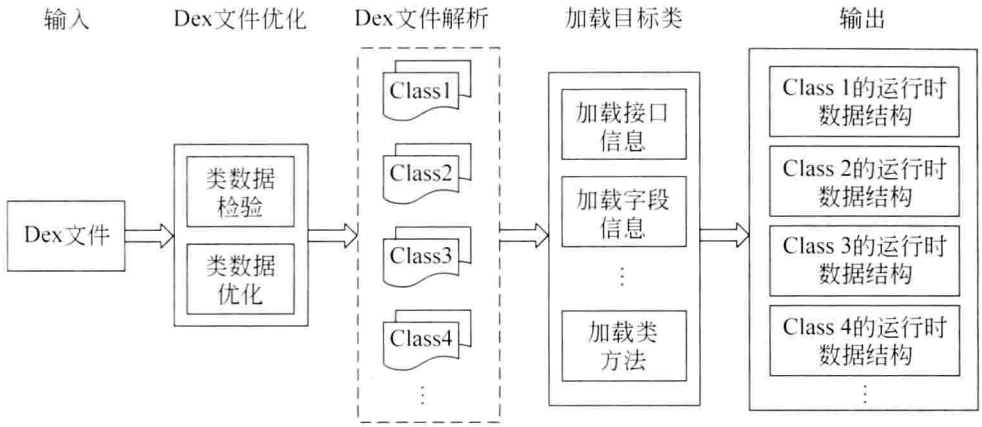

图 1.2　类加载机制整体工作概况图

在随后的内容中将分别从 Dex 文件的优化与验证、Dex 文件解析以及类的实际加载等三方面（关键内容）展开介绍，以帮助读者对类加载机制的整体功能、设计原理以及函数工作流程有一个较为深入的理解。

1.3　Dex 文件的优化与验证

事实上，Dex 文件中类数据的优化与验证是 Dex 文件解析工作的一部分，但由于该优化与验证工作在 Dalvik 虚拟机中被前置，使之成为 Dalvik 区别于其他 Java 虚拟机的重要特征，因此这部分工作具有较高的重要性，其工作效果的好坏在一定程度上决定了 Dalvik 虚拟机是否可以高效安全地运行，因此本书特将这部分内容抽取出来单独介绍。同时，Dalvik 虚拟机的优化技术一直是业界关注的焦点，如何进一步提高 Dalvik 虚拟机在嵌入式设备上的性能表现应该是未来工程开发人员工作的重点。

在这一节中，主要讲解了 Dex 文件的优化与验证技术的设计原理、关键数据结构以及其函数执行流程。希望通过对这三方面内容的介绍，可以让读者由表及里地了解该技术的功能原理以及具体实现。

1.3.1　Dex 文件优化验证的原理与实现

依据 Google 提供的开发文档的描述，在通常情况下，优化和验证 Dex 文件最安全且最便捷的方式就是在虚拟机中直接加载目标 Dex 文件并运行其中包含的所有类，因为一旦加载失败或是运行失败，就意味着 Dex 文件优化验证的失败。因此，虚拟机究竟是使用何种方法，以较为高效的手段实现了对 Dex 文件的验证与优化，这将是一个非常值得深入研究的问题。为了得到最为直观、真实的结论，需要从 Android 源码入手，以彻底弄清问题的本质。

根据对优化机制的源码研究发现：Dalvik 虚拟机的优化验证工作独立于程序的执行，同时优化验证这两部分功能也被整合成为一个功能模块并且类加载机制在加载工作的初期通过类似接口调用的方式调用这个优化模块对目标 Dex 文件进行优化。为了解决资源占用问题，Android 系统将会新建一个虚拟机用于实现相应的功能，在 Dex 文件的优化验证工作结束并正常输出优化文件后，系统将释放该虚拟机占用的所有资源。

同时，为了更大程度地保证原 Dex 文件的数据的安全以及优化机制的独立性，优化机制并不直接改写原 Dex 文件，而是重新创建一个后缀为.Odex 的空文件并以严格的格式要求将所有的优化信息写入该文件，主要包括依赖库关系、寄存器映射关系以及类的索引关系，这些关系的建立会大大提高类加载机制的执行效率，同时，在优化过程中还会根据平台特性对原 Dex 文件中部分字节码进行替换（例如，对字段的访问方式由查找改为直接引用）以提高程序执行速率，最后再将重写的 Dex 文件也写入 Odex 文件（1.3.2 节将详细介绍 Odex 文件结构以及各部分功能）。Odex 文件作为优化机制的输出将会取代原 Dex 文件并作为直接的可执行文件被其他功能模块（例如，内存管理模块、解释器模块等）调用。Dex 文件的优化机制大致的工作流程如图 1.3 所示。

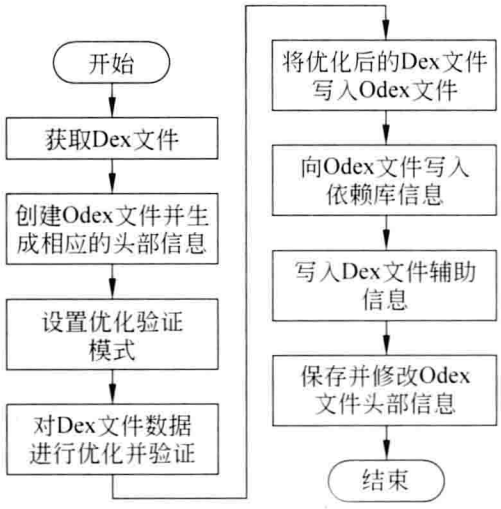

图 1.3　优化机制工作流程图

Dex 文件的优化与验证机制在 Dalvik 虚拟机中被设计成一个独立的系统工具，这样做的好处是使该机制具有比较高的独立性，使整个机制模块性更好，在一定程度上降低了整个 Android 系统的冗余。同时，也提供了一个技术参考，在适应低性能的移动平台时，应该尽量采用这种优化手段以提高效率。

点拨　Android 设备在第一次启动程序时往往耗费较长的时间，实际上，在这期间虚拟机对目标 Dex 文件进行了验证与优化，并为之生成了相应的 Odex 文件。当用户再次启动应用程序时，新生成的 Odex 文件将会代替原有的 Dex 的文件被虚拟机引用执行，由于不需要再次对程序数据进行验证优化，极大地缩短了启动时间。

1.3.2　Odex 文件结构分析

直观上 Odex 文件在 Dex 文件的原有结构上进行了扩充，即在 Dex 文件前拼接了 Odex 文件头部信息，还在 Dex 文件尾部拼接了依赖库、寄存器映射关系以及类的哈希索引等辅助信息。其结构对比如图 1.4 所示。

通过 Odex 文件的头部信息可以更好地了解一下 Odex 的文件结构以及各部分数据含义，表 1.1 为 Odex 文件头 DexOptHeader 在 Dexfile.h 文件中的定义。

在表 1.1 中，DexOptHeader 结构中的 magic 字段与 DexHeader 结构中的 magic 字段类似，都是用于标识文件；dexOffset 字段表示原 Dex 文件起始位置的偏移量，实际上它就等于 DexOptHeader 结构体的大小 0x28；dexLength 字段表示 Dex 文件的总长度，通过这两个字段可以非常快速定位并读取 Dex 文件；depsOffset 字段表示依赖库起始的偏移量；

depsLength 表示依赖库的总长度；optOffset 字段表示优化数据的起始偏移量；optLength 字段表示优化信息的总长度，而对于类加载机制非常关键的类索引信息就封装在这部分优化信息中；flags 字段为一个标识，其用于标示 Dalvik 虚拟机加载 Odex 文件时优化与验证选项；checksum 字段为 Odex 文件的校验和。

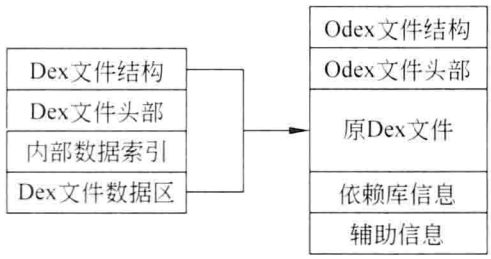

图 1.4　Dex 文件与 Odex 文件结构对比图

表 1.1　DexOptHeader 数据结构定义

变量类型	变量名称	描　述
u1	magic[8]	Odex 文件版本标识
u4	dexOffset	Dex 文件头偏移量
u4	dexLength	Dex 文件总长度
u4	depsOffset	Odex 文件依赖库列表偏移量
u4	depsLength	依赖库信息总长度
u4	optOffset	优化数据信息偏移量
u4	optLength	优化数据总长度
u4	flags	标识位
u4	checksum	文件校验和

通过这个头部信息，虚拟机可以非常高效地查找 Dex 文件中的各类信息，极大提高了执行效率，另外，Dalvik 虚拟机对 Dex 文件所进行的优化工作主要体现在依赖库和辅助信息两部分上，因此，下文将会对这两部分内容的功能进行介绍。

1. 依赖库信息

依赖库顾名思义，就是指该 Dex 文件所需要链接的本地函数库，Dalvik 虚拟机在程序执行前期通过优化机制将这部分整合到 Odex 文件中，可以在一定程度上提高程序的执行效率，表 1.2 为依赖库 Dependence 结构体定义。

表 1.2　Dependence 数据结构定义

变量类型	变量名称	描　述
u4	modWhen	时间戳
u4	crc	校验信息
u4	DALVIK_VM_BUILD	虚拟机版本号
u4	numDeps	依赖库个数
u4	len	Name 长度
u4	name[len]	依赖库名称
KSHA1DigestLen	signature	SHA-1 值

在表 1.2 中，modWhen 用来记录 Dex 文件优化前的时间戳，crc 为 Dex 文件优化前的 crc 校验值，DALVIK_VM_BUILD 值表示的是虚拟机版本号；不同版本的 Android 系统定义不同，例如，Android 2.2.3 为 19、Android 2.3 为 23 以及 Android 4.0.4 则为 27。numDeps 字段所代表的含义为该 Dex 文件的依赖库个数，其中 table 结构体的个数正是由

numDeps 决定的，也可以理解为每个依赖库都对应一个 table 结构体对象，在该结构体中，len 表示依赖库名称的长度、name 为依赖库名以及 signature 表示 SHA-1 签名。

2. 类索引信息

类索引信息的建立是优化机制的重要工作之一，在该索引表中，优化机制为 Dex 文件中的每一个类配置了一个 table 结构体对象，在这个对象中记录了类描述符哈希值、类描述符在 Dex 文件中偏移地址以及类定义区的偏移地址，类加载机制通过这些信息可以非常快速地定位类资源地址并加载类。同时，通过哈希查找的方式极大地提高了类加载机制的查找效率，表 1.3 为 DexClassLookup 结构体定义。

表 1.3 DexClassLookup 数据结构定义

变量类型	变量名称	描　述
int	size	表大小
int	numEntries	表项入口数量
u4	classDescriptorHash	类描述符的哈希值
u4	classDescriptorOffset	Dex 文件中该类描述符的偏移位置
u4	classDefOffset	Dex 文件中该类定义偏移位置

在表 1.3 中，numEntries 是一个比较特别的数目，它虽然表示的是表的项数，但实际上这个数值是通过 dexRoundUpPower2()函数生成。

点拨 dexRoundUpPower2()函数是源自斯坦福大学的一个算法——用于求比一个数大的最小的 2 的整数次幂，例如：当数为 6 时，该算法计算得到 8。这样做的结果会比 Dex 文件中类的数量大，但好处是降低了哈希冲突率。

1.3.3 函数执行流程

Dex 文件的优化始于 Android 源码中 frameworks 层的 PackageManagerService 类，该类实际上是通过 Installer 类实现对 apk 文件的安装、优化以及卸载等工作。而 Installer 类通过与 c 层的 installd 建立 socket 连接，使得在上层的 install、remove、dexopt 等功能最终由 installd 在底层实现。在 installd 中，do_dexopt 函数负责完成对 Dex 文件的优化，而 do_dexopt 函数将会调用 dexopt 函数去完成实际的优化工作。在 dexopt 函数中，首先完成一些必要的准备工作，比如声明关键变量，分析文件路径，而最关键的是创建了一个空文件并以 Odex 后缀结尾，由此，可以认定 dexopt 函数用于产生 Odex 文件，在完成准备工作后，将会委托 run_dexopt 函数完成实际的优化工作。

在 run_dexopt 函数中，可以看到这样一行代码：

```
execl(DEX_OPT_BIN, DEX_OPT_BIN, "-zip", zip_num, Odex_num, input_file_name, dexopt_
flags, (char * ) NULL);
```

run_dexopt 函数通过使用 Execl 函数将优化工作委托给了由第一参数 DEX_OPT_BIN 宏定义所指出的可运行程序，DEX_OPT_BIN 宏定义的内容是：

```
static const char * DEX_OPT_BIN="/system/bin/dexopt";
```

而/system/bin/dexopt 中的 dexopt 程序的源码位于 Android 系统源码的 dalvik\dexopt 目录下，从目录结构上即可反映出 dexopt 优化程序是 Dalvik 虚拟机下第一级子程序，也正说明优化机制在 Dalvik 虚拟机中确实为一个相对独立的功能模块。

在清楚了优化机制的源头后，dexopt 优化程序的工作流程成为我们比较关注的一个问题，因此，本文将在此着重分析优化机制的具体实现过程并介绍优化机制中关键的技术点以及有代表性的实现细节。

dexopt 的主程序代码位于 dalvik\dexopt\OptMain.cpp 文件中，其中 extractAndProcessZip() 函数用于处理并优化 apk/jar/zip 文件中的 classes.dex，因此该函数将作为优化机制的主控函数，extractAndProcessZip()函数的实现代码如下。

代码清单 1.1　dalvik\dexopt\OptMain.cpp：extractAndProcessZip()函数源代码

```
static int extractAndProcessZip( int zipFd,int cacheFd,const char * debugFileName,
bool isBootstrap,const char * bootClassPath,const char * dexoptFlagStr)
{
    /* 函数在执行初期声明相关的中间变量 */
    ZipArchive zippy;                    //用于描述 ZIP 压缩文件的数据结构
    ZipEntry zipEntry;                   //用于表示一个 ZIP 入口
    ...
    off_t dexOffset;                     //用于表示在 Odex 文件中，原 Dex 文件的起始地址
    int err;                             //标示符
    int result=-1;                       //函数返回值
    int dexoptFlags=0;                   //优化标示符
    /* 设置默认的优化模式 */
    DexClassVerifyMode verifyMode=VERIFY_MODE_ALL;
    DexOptimizerMode dexOptMode=OPTIMIZE_MODE_VERIFIED;
    memset(&zippy,0,sizeof(zippy));      //对 zippy 对象进行置 0 操作
    /* 对入口参数 cacheFd 文件描述符所代表的输入文件进行为空判断，该文件必须保证为空，因
       为在后期要将优化后的数据写入该文件中 */
    if (lseek(cacheFd,0,SEEK_END) !=0) {
        LOGE("DexOptZ: new cache file '%s' is not empty",debugFileName);
        goto bail;
    }
   /* 当 cacheFd 所指文件为空，那么为其创建一个 Odex 文件的头部 */
    err=dexOptCreateEmptyHeader(cacheFd);
    if (err !=0)                         //对函数执行结果进行判断，如果失败则将返回
        goto bail;
  /* 取得 Odex 文件中原 Dex 文件的起始位置，实际就是一个 Odex 文件头部的长度，并将结果赋值
     给变量 dexOffset */
    dexOffset=lseek(cacheFd,0,SEEK_CUR);
    if (dexOffset<0)
        goto bail;
    /* 打开 ZIP 对象，在其中查找目标 Dex 文件 */
    if (dexZipPrepArchive(zipFd,debugFileName,&zippy) !=0) {
```

```
        LOGW("DexOptZ: unable to open zip archive '%s'",debugFileName);
        goto bail;
    }
/* 获取目标 Dex 文件的解压入口 */
    zipEntry=dexZipFindEntry(&zippy,kClassesDex);
    if (zipEntry==NULL) {
        LOGW("DexOptZ: zip archive '%s' does not include %s",
            debugFileName,kClassesDex);
        goto bail;
    }
    /* 获取相关 ZIP 入口信息 */
    if (dexZipGetEntryInfo(&zippy,zipEntry,NULL,&uncompLen,NULL,
                        NULL,&modWhen,&crc32) !=0)
    {
        LOGW("DexOptZ: zip archive GetEntryInfo failed on %s",
                debugFileName);
        goto bail;
    }
    :
    /* 从 ZIP 文件将目标 Dex 文件解压出来,并写入 cacheFd 所指文件,此时 cacheFd 所指文件
       非空,包括一个 Odex 文件头部加上一个原始的 Dex 文件 */
        if (dexZipExtractEntryToFile(&zippy,zipEntry,cacheFd) !=0) {
        LOGW("DexOptZ: extraction of %s from %s failed",
            kClassesDex,debugFileName);
        goto bail;
    }
/* 根据入口参数 dexoptFlagStr,对验证优化需求进行分析,dexoptFlagStr
     实际上是一个字符串,记录了验证优化的要求 */
    if (dexoptFlagStr[0]!='\0') {
        const char* opc;
        const char* val;
            /* 设置验证模式 */
        opc=strstr(dexoptFlagStr,"v=");     /* verification */
        if (opc !=NULL) {
         switch (*(opc+2)) {
         case 'n': verifyMode=VERIFY_MODE_NONE;          break;
         case 'r': verifyMode=VERIFY_MODE_REMOTE;        break;
         case 'a': verifyMode=VERIFY_MODE_ALL;           break;
         default:                                        break;
            }
        }
        /* 设置优化模式 */
        opc=strstr(dexoptFlagStr,"o=");     /* optimization */
        if (opc !=NULL) {
```

```
        switch (*(opc+2)) {
        case 'n': dexOptMode=OPTIMIZE_MODE_NONE;         break;
        case 'v': dexOptMode=OPTIMIZE_MODE_VERIFIED;     break;
        case 'a': dexOptMode=OPTIMIZE_MODE_ALL;          break;
        case 'f': dexOptMode=OPTIMIZE_MODE_FULL;         break;
        default:                                         break;
            }
        }
⋮
}
/* 当完成了原 Dex 文件的提取以及验证优化选项的设置,即可以开始真正的优化工作,需要初始
   化一个虚拟机专门用于验证优化工作 */
    if (dvmPrepForDexOpt(bootClassPath,dexOptMode,verifyMode,
            dexoptFlags) != 0)
    {
        LOGE("DexOptZ: VM init failed");
        goto bail;
    }
/* 调用 dvmContinueOptimization 函数完成对 Dex 文件的验证与优化工作 */
    if (!dvmContinueOptimization(cacheFd,dexOffset,uncompLen,
                    debugFileName,modWhen,crc32,isBootstrap))
    {
        LOGE("Optimization failed");
        goto bail;
    }
    result=0;                                          //设置返回值,0 表示成功
    ⋮
    return result;                                     //函数返回}
```

从上面的源码中可以看到,extractAndProcessZip()函数首先会调用 dexOptCreateEmptyHeader()函数为 Odex 文件创建一个文件头用于描述 Odex 文件内容,随后主函数将调用 dexZipFindEntry()函数检查目标文件中是否拥有 classes.dex 文件,如果目标 Dex 文件存在,则通过 dexZipGetEntryInfo()函数读取 Dex 文件相关的验证信息,接着调用 dexZipExtractEntryToFile()函数提取 classes.dex 文件并写入 Odex 文件,在写入完毕后,主程序将会根据入口参数 dexoptFlagStr 解析检验与优化模式并将优化选项 dexOptMode 与验证 verifyMode 写入到全局变量中。至此,优化机制的准备工作基本结束。

在准备工作完成后,主函数调用 dvmPrepForDexOpt()启动并初始化一个虚拟机进程,而以后的所有优化操作都会在这个进程中完成,当 Odex 文件正常输出后,这个进程的所有资源都会被释放。当优化的进程准备完毕后,主函数将调用 dvmContinueOptimization()函数开始真正的验证与优化工作。

点拨　启用一个专门的进程用于负责 Dex 文件的优化与验证,这样做的好处是在一定程度上保证了虚拟机的安全运行,因为一个未经安全验证的程序,不能保证对其他虚拟机进程绝对安全。例如,恶意篡改共享数据、造成内存溢出等。

代码清单 1.2 dalvik\vm\analysis\DexPrepare.cpp：dvmContinueOptimization()函数源代码

```
bool dvmContinueOptimization(int fd,off_t dexOffset,long dexLength,
            const char* fileName,u4 modWhen,u4 crc,bool isBootstrap)
{
    /*声明相关中间变量*/
    DexClassLookup* pClassLookup=NULL;
    RegisterMapBuilder* pRegMapBuilder=NULL;
    assert(gDvm.optimizing);
    LOGV("Continuing optimization (%s,isb=%d)",fileName,isBootstrap);
    assert(dexOffset >=0);                              //判断输入文件长度非 0
    /*对目标文件进行合法性检验*/
    if (dexLength< (int) sizeof(DexHeader)) {
    /*一个 Dex 文件的长度不能小于其文件头的长度*/
        LOGE("too small to be DEX");
        return false;
    }
    /*Odex 文件中的 Dex 文件的起始偏移量不能小于 Odex 文件头的长度*/
    if (dexOffset< (int) sizeof(DexOptHeader)) {
        LOGE("not enough room for opt header");
        return false;
    }
    ⋮
        /*将 fd 所指文件映射到某一位置,该位置的起始地址为 mapAddr,其大小就为 fd 所指文件大小,即一个 Odex 文件头部加上一个 Dex 文件长度*/
        mapAddr=mmap(NULL,dexOffset+dexLength,
             PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
        if (mapAddr==MAP_FAILED) {
            LOGE("unable to mmap DEX cache: %s",strerror(errno));
            goto bail;
        }
        /*设置相关的优化验证选项*/
        bool doVerify,doOpt;
        if (gDvm.classVerifyMode==VERIFY_MODE_NONE) {
            doVerify=false;
        } else if (gDvm.classVerifyMode==VERIFY_MODE_REMOTE) {
            doVerify=!gDvm.optimizingBootstrapClass;
        } else /* if (gDvm.classVerifyMode==VERIFY_MODE_ALL) */ {
            doVerify=true;
        }
        if (gDvm.dexOptMode==OPTIMIZE_MODE_NONE) {
            doOpt=false;
        } else if (gDvm.dexOptMode==OPTIMIZE_MODE_VERIFIED ||
```