

分类号	
UDC	
密 级	
学 号	2141220004

西安理工大学

硕士学位论文

Android App 功能插件化机制的研究与实现

学 科 门 类: 工 学

学 科 名 称: 计算机系统结构

指 导 教 师: 孙钦东 教授

申 请 日 期: 2017 年 6 月

论文题目: Android App 功能插件化机制的研究与实现

学科名称: 计算机系统结构

研究生: 熊建芬

签名: 熊建芬

指导教师: 孙钦东 教授

签名: 孙钦东

摘要

随着智能手机的大众化, 移动应用层出不穷, 要在众多的应用中得到用户的青睐, 除了功能实用外, 该应用的用户体验势必也是绝佳。按照传统的开发方式, 应用需要频繁更新, 而每更新一次, 都需要用户进行重新下载整个安装包进行安装, 对于大部分用户而言可能嫌费流量、麻烦, 从而拒绝更新应用。这种现象也就给移动应用快速迭代和 bug 的修复造成了严重的影响。目前已经开源的插件化框架有不少, 但是大多数针对性比较强, 要么针对的是新功能的增加, 要么是 bug 的修复, 并没有一个框架对两者进行融合; 并且这些框架也没有采取任何安全机制来保障应用的安全。

正是基于上述原因, 本文提出了一种 Android App 插件化机制。这种插件机制按照插件化的粒度分成模块化更新和热修复。模块化更新针对的是新功能的增加, 而热修复针对的是类文件中方法级别的修复。在这种机制下用户无需手动安装新模块, App 通过动态加载的方式就可以进行更新, 大大提高了用户的体验性, 更为开发者部署应用、更新应用、修复应用的 bug 提供了很大的方便。本文所研究的机制具有以下特点: 一是可以在用户无感知的情况下下载更新模块、对新功能模块更新和修复类文件中的 bug; 二是每个插件模块都是独立的, 插件之间互不影响, 并且可以进行热插拔; 三是热修复模块与模块化更新结合, 既实现了大粒度的模块更新, 又实现了小粒度的热修复功能; 四是具有一定的安全机制, 可以在一定程度上保障应用的安全。

本文通过分析 Android 系统框架和源码得出插件化的位置与各大组件插件化的思路, 再结合动态加载技术与相关机制来实现 Android App 的插件化。最后, 经过 Demo 的测试, 该插件机制可以实现模块化的动态加载和 bug 的修复, 并且模块可以随时的安装和卸载, 与传统的开发方式相比, 以插件化开发方式开发的 App 不论在开发效率还是用户的体验性上都会有较大的提高。

关键词: Android App; 插件化机制; 粒度; 安全机制; 模块化更新; 热修复

Title: RESEARCH AND IMPLEMENTATION OF PLUGIN MECHNISM OF ANDROID APP FUNCTION

Major: Computer Architecture

Name: Jianfen Xiong

Signature: Jianfen Xiong

Supervisor: Prof. Qindong Sun

Signature: Qindong Sun

Abstract

Smart phones are very common in social. Mobile applications is various. To get the user's favor in so many mobile applications, the user experience of application must be bound to be excellent in addition to the function. In accordance with the traditional development methods, applications need to be updated frequently and each time users need to re-download the entire installation package and install. Users may feel costly, trouble and then refuse to update the application. This phenomenon also has a serious impact on the rapid iteration of mobile applications and bug fixes. Now there are many plug-in frameworks that are already open source. Some of them is for new features, the others are for bug fixes. But no one integrates both of them. Also none of these frameworks takes some security mechanism to protect the application security.

For the above reasons, this paper presents an Android App plug-in mechanism. This plug-in mechanism is divided into modular updates and hot fixes according to the plug-in granularity. The modular update is for the addition of new features, and the hot fixes is for the method level fixed in the class file. In this mechanism, users do not need to manually install the new module any more. App can be updated through dynamic loading. This approach not only greatly improve the user's experience, but also offers great convenience for developers deploying applications, updating applications, fixing bugs. The mechanism of this paper has the following characteristics: First, users can be unconscious to download the update module, and then update the new modules and fix class file bugs; Second, each plug-in module is independent. Plug-ins do not affect each other and can install or uninstall; Third, Combining the hot fixes and module update, it can achieve updating new features and hot patch function; Fourth, with a security mechanism to maintain the security of the application.

This paper analyzes Android system framework and source code to get plug-in location and the train of thought of the major components for plugin. And it combines dynamic loading technology with related mechanisms to achieve Android App plug-in. Finally, after testing the

Demo, it proves that the plug-in mechanism can achieve modular dynamic loading and fix bugs, and the module can be installed and uninstalled. Comparing with the traditional development methods, App using plug-in method to develop regardless of the development efficiency or the user's experience have a greater improvement.

Key Word:Android App; plugin mechanism; granularity; security mechanism; modular update; hot fixed

目录

1 绪论.....	1
1.1 研究背景与意义.....	1
1.2 国内外研究现状.....	3
1.3 主要工作及研究内容.....	4
1.4 本文的组织结构.....	5
2 动态加载技术与相关机制分析.....	7
2.1 动态加载技术.....	7
2.1.1 Java 类的动态加载机制.....	7
2.1.2 Android 虚拟机类加载机制.....	8
2.2 HOOK 机制.....	9
2.3 Binder 机制.....	11
2.4 本章小结.....	13
3 粒度化的安全功能插件化实现机制分析.....	15
3.1 插件化的位置.....	15
3.2 插件加载技术.....	16
3.2.1 代码加载方式分析.....	17
3.2.2 资源编译加载方式分析.....	17
3.3 大粒度插件化模块化更新.....	19
3.3.1 模块隔离性分析.....	19
3.3.2 四大组件插件化分析.....	20
3.4 小粒度插件化热修复.....	25
3.4.1 机制中热修复的原理分析.....	25
3.4.2 动态代码注入解决 Multidex 失败的问题.....	26
3.5 插件安全机制分析.....	27
3.5.1 APK 签名机制.....	27
3.5.2 dex 文件的保护.....	27
3.6 本章小结.....	28
4 粒度化的安全功能插件化实现机制设计.....	29
4.1 总体设计.....	29
4.2 插件加载模块.....	30
4.2.1 资源加载.....	31
4.2.2 代码加载.....	32
4.3 模块化更新.....	32

4.3.1 APK 解析	32
4.3.2 AndroidManifest 预注册	33
4.3.3 四大组件的插件化.....	33
4.3.4 HOOK 模块	34
4.4 热修复模块.....	36
4.5 缓存机制与插件包管理.....	37
4.6 插件启动.....	38
4.6.1 进程管理.....	38
4.6.2 组件生命周期管理.....	39
4.7 插件化机制安全模块.....	40
4.7.1 APK 签名	40
4.7.2 验证 APK 签名.....	41
4.7.3 DES 算法加解密 dex 文件	42
4.8 本章小结.....	44
5 功能插件化机制的实现与验证.....	45
5.1 AndroidManifest 预注册的实现	45
5.2 APK 解析的实现	46
5.3 四大组件启动的实现.....	47
5.3.1 ClassLoader 加载代码.....	47
5.3.2 Activity 的启动	47
5.3.3 Service 的启动.....	48
5.3.4 BroadcastReceiver 的启动	48
5.3.5 ContentProvider 的启动	49
5.4 热修复的实现.....	49
5.5 插件化安全机制的实现.....	50
5.6 插件化机制的验证与对比.....	51
5.6.1 机制验证.....	51
5.6.2 相关机制对比.....	55
5.7 本章小结.....	56
6 总结与展望.....	57
6.1 工作总结.....	57
6.2 工作展望.....	57
致谢.....	59
参考文献.....	61
在校学习期间所发表的论文、专利、获奖及社会评价.....	65

1 绪论

1.1 研究背景与意义

自 2007 年苹果谷歌开始发布智能手机系统，从此智能移动设备不断发展，如今市场上已经出现了大量的智能移动设备系统，其发展势头也已经远远超过了传统的 PC^[1]。但是目前使用最广泛的系统还是 Android，iOS 和 Windows Phone，其他操作系统的使用量很少。表 1-1 是各个系统在市场上所占份额。

表 1-1 2016 年移动设备操作系统全球市场份额

Table 1-1 The global market share of 2016 mobile device operating system

移动操作系统市场份额	2016 年份额
Android	66.01%
iOS	27.84%
Windows Phone	2.79%
Symban	1.03%
BlackBerry	0.85%
其他	1.48%
总和	100%

从表中可以看出 Android 系统仍然占据着绝对主导的市场。Android 作为 Linux 内核的新继承者，自然就继承了 Linux 的特性^[2]。同时它具有一个最大的优势那就是代码的开源性，也正是因为这个优势 Android 系统得到巨大的完善和丰富^[3]。由于 Android 向开发人员提供了成熟的开发平台和方便的应用发布市场，因此大大简化了开发和推广流程，可以让开发人员更多的去关注应用本身的开发。

目前从全球看来，iOS 和 Android 的应用市场规模不断扩大，已经超过 1 万亿美元^[9]。截止到 2016 年，应用商店的总的营收额将突破 460 亿美元，其中包括有付费下载，应用内购买、订阅、和广告服务，已经远远超过了 2011 年的 85 亿美元。随着移动端信息技术的不断高速发展，应用市场的规模将不断扩大，预计到 2020 年移动应用的下载量将达到 1500 亿次^[4-6]。我国正在逐步的加大对移动互联网基础设施的建设，基础设施将会越来越完善，移动应用也将随之越来越普及，所涉及到的领域也会越来越多，这样就导致应用的规模越来越庞大。开发者往往会因为用户需求的不断变化，对应用进行频繁的更新^[6]。对于传统的开发模式而言，每一次应用的更新都需要用户去重新下载安装整个应用包。这样的开发模式对于用户来说，体验是十分不友好，并且浪费用户的流量。因此传统的开发模式很难再胜任移动端应用的开发工作了。

模块化的开发不仅可以提高开发效率，同时也可以减少开发的成本、增加软件的灵活性、对于延长软件的生命周期也起到了很大的作用。从软件生命周期的各个阶段就能看出：

- (1) 在需求分析阶段，与用户沟通时可以快速切入用户所感兴趣的业务，让用户能

够更加直观的感受软件的雏形，方便用户提出各种需求。

(2) 在软件设计阶段，软件架构十分关键。而随着软件业务和规模的庞大，软件架构自然就越来越复杂。模块化的开发可以让开发者暂时忽略模块内的设计细节，将更多的精力放在软件架构的设计上^{[11][12]}。

(3) 在软件编码阶段，可以提高软件模块的复用率，避免了大量代码的重复使用。因此可以提高软件编码的效率。

(4) 在软件部署阶段，可以进行模块化的更新，让用户先下载需要的功能，再后续去更新下载新功能。

(5) 在软件维护阶段，一旦应用出现问题，开发人员可以针对模块进行测试，同时不会影响其他模块，因此也可以减少软件维护的成本^[10]。

为了适应现代移动应用发展的需求，就需要开发者采用其他的开发模式来取代传统的开发模式。

2016 年 Android 插件化技术取得了突飞猛进的成就，随着业务需求的不断扩大，各大厂商都面临着 Android Native 平台的瓶颈^[12]。从技术的层面来分析，业务逻辑的不断复杂导致代码的急剧膨胀，各大厂商陆续的触到 65535 方法数的天花板。与此同时，以运营为主的时代，对于模块更新的需求提出了更大的挑战^[16]。从业务逻辑上来分析，功能模块之间的解耦以及维护开发团队的分离也是一种大趋势。每个团队维护着不同的应用模块。

利用 H5 和 Hybird 的混合开发方式能够解决这些问题。H5 应用其实是由一个个网页构成运行在手机浏览器上，通常应用会做一个套壳的浏览器运行页面。因为 H5 的一些特性，很多应用会使用半原生半 H5 的 Hybird App 开发模式。H5 应用相对于原生应用有着其明显的优势。在机型的适配问题上，完全无需考虑用户机型和适配的问题，还可以直接在网页上调试和修改，大大改善了原生开发平台碎片化、开发的人力成本、时间成本^[8]；在版本升级方面，具有的优势更加明显，应用的升级与用户无关，无需下载安装更新，可以实时的送到用户手中；在上线后的维护成本方面，上线的时间稳定快速，不需要开发市场审核，并且如果有收入成分的开发市场还可以绕过收入成分。除此之外 H5 应用还有一些优势，例如在视频媒体方面。虽然 H5 应用有如此多的优点，但也不能忽略其缺点。H5 应用的上网速度与响应速度与原生应用相差十分大并且在加载大图片时性能下降，尤其当大量用户同时访问同一个应用时，性能下降的十分明显，同时它也不能自动处理动画反复交互的问题^[17]。H5 还有两个致命的缺点：第一是在性能支持问题上，原生应用开发远远超过了 H5 应用，并且在用户体验方面原生比 H5 响应速度要流畅和快；第二是功能问题，原生应用可以很轻易的调用硬件摄像头、陀螺仪、麦克风等硬件，而 H5 对这些硬件的支持较差，因此当 App 需要使用这些硬件时，H5 就不容易扩展，速度也比较慢。

正是为了避免上述缺点，国外使用 FaceBook 推出的 react-native；而国内大部分使用纯 native 的插件化技术。

1.2 国内外研究现状

早在上世纪 60 年代 H.Simon 就提出了“模块”这一概念^[21]。他认为插件化模块在复杂系统的演化中是动态均衡之后的一种特殊框架^[21]。插件式的开发方式在开发应用程序的模式中是一种构架上的模型，而不是通用的某种约定的技术标准。在应用程序的开发过程中，“模块”只是逻辑上的概念，把一个应用程序的整体划分为互相独立的单独的宿主应用，外加多个应用外的功能模块^[22]。图 1-1 即是其体系结构图。

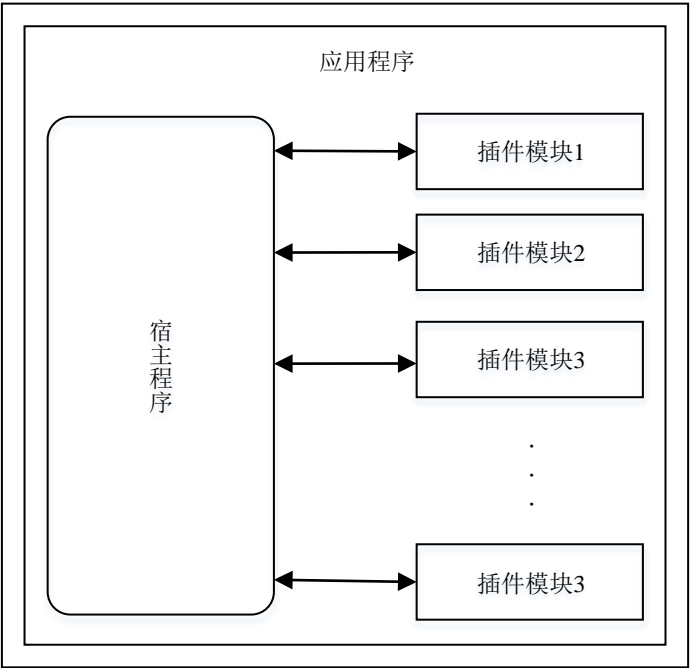


图 1-1 插件化体系结构图

Figure1-1 Plug-in architecture diagram

从图 1-1 中可以看出模块与宿主程序的特点：

- （1）插件模块可以进行动态独立的安装。
- （2）宿主模块对各个插件模块进行统一的管理，其中包括注册，安装等过程。
- （3）宿主模块应该具有向后的兼容性，也就是说新版的宿主程序可以正常的装载旧版的插件模块。

Baldwin 和 Clark 提出了模块设计中的两种规则：第一种是显性规则，它定义了外部模块与模块之间的关系和它们是以什么样的方式组合在一起的，也被称为外部规则；第二种是隐性规则，是在遵循显性规则基础上可以自由定义块的内部，也被称为内部规则。设计的显性规则可以确保采用模块化开发的系统正常运行，而隐性规则可以不断的完善模块本身的功能，最终提高整个系统的性能^[24]。因此可以得出隐性设计规则封装了功能模块的技术细节，而显性设计规则确保了软件外部的稳定。从而采用更先进的软件开发技术软件整体仍旧可以稳定持续使用，这样也就使软件的生命周期得到了延长^[23]。

模块化在众多软件程序的开发过程中是一种很常见的软件开发架构，尤其是在 PC 端，其主要的开发方法包括：动态链接库，组件对象模型^[22]。在开发 Android 应用时，模块化

的开发方式也已经兴起了一段时间,目前各大互联网公司都在加紧开发属于自己的模块化框架用来适应日益多样化的开发需求^{[25][26]}。现在已经开源的插件化框架并且相对比较成熟的主要有: DynamicLoadApk、AndroidDynamicLoader、DroidPlugin。虽然这些插件化框架已经相对比较成熟稳定,但是不可避免的还是存在自身的某些缺陷。DynamicLoadApk 的核心思想是通过代理来实现的,而通过代理反射方式来实现,势必其效率不会太高; AndroidDynamicLoader 是通过 Fragment 以及 schema 的方式实现的,这种方式的局限性在于要求 Activity 必须通过 Fragment 实现; DroidPlugin 虽然避免了上述两种方案的短板。但是它也具有缺陷,在加载插件时并不能很好的去辨别插件的安全性。热修复的框架在开源上也有不少,但是它们大多数对系统的兼容性不是很好,尤其是 art 系统。

现如今面对用户不断变化的需求和移动应用不断迭代的环境,开发人员与用户都热切盼望有一个成熟、稳定、安全、完善的插件化框架。但是目前不论是国内还是国外都缺乏一个比较完善稳定的插件化框架。鉴于这个原因,本文将设计一个全新的 Android 平台下的插件实现机制,这个机制相比于现在的开源框架最大的不同是根据插件化粒度的大小分成模块化更新和热修复,并且设计了相关的安全机制从一定程度上来保障应用的安全。

插件安全性的检测是非常必要的,如果不对插件的安全性进行检测,一旦插件模块在网络传输中被劫持替换,产生的后果将会是十分严重的。目前开源的插件化框架也可以进行模块化更新和热修复的工作,但是并没有将这两者进行细分并且它们针对性比较强,要么是針對新功能的增加,要么针对的是 bug 的修复。热修复与模块化更新实质上都是插件化,但是它们还是有区别的,模块化更新是进行功能上的更新,是一个从无到有的过程;而热修复是修复类文件中的 bug,是应用中原本就一直存在的功能。在本文实现的插件化机制中两者接入宿主应用的方式上有不同。将这两种插件化的方式进行细分可以一定程度上提高应用更新的效率。

1.3 主要工作及研究内容

目前插件化框架越来越多,其中一部分框架主要是针对新功能的增加,另一部分针对的是类文件中 bug 的修复,但是还没有任何框架将两者结合在一起。两者的实质都是进行插件化,但是插件化的粒度不同。本文根据插件化粒度的大小分成模块化更新和热修复,并使用 Java 动态加载技术对 Android 系统组件进行 HOOK 替换来达到组件的动态加载和热修复的效果,同时通过检测 APK 的签名来提高应用的安全性,通过保护 dex 文件来保障代码的安全性。主要的研究工作如下:

(1) 研究与分析相关技术和机制。其中包括动态加载技术、HOOK 机制和 Binder 机制。在插件化的过程中这些技术与机制都起到了核心作用,本文利用动态加载技术加载代码,利用 HOOK 机制与 Binder 的结合让插件能够正常访问系统的资源和服务。

(2) 研究和分析 Android 系统框架和源码,得出插件化在系统中的位置与代码、资源的加载思路。通过对系统框架的分析得出插件化的位置只能介于应用层与 Framework

层之间,在其他层进行插件化是不可行的。通过分析系统加载资源和各大组件启动运行的过程,了解每个过程的特点,针对各自的特点制定不同的插件化方法。

(3) 研究插件模块的接入方式。根据接入宿主应用方式的不同来区别两种粒度的插件化。针对小粒度的插件化,使用 dex 文件接入方式;而大粒度的插件化采取 APK 包接入方式。

(4) 研究 App 的签名技术。本文利用自动化生成工具生成签名文件。在 Android 应用的版本升级安装过程中,系统都会对应用的签名进行验证,通过辨别签名的一致性与完整性来判定应用的安全性。利用 App 的这一特点对插件模块的签名进行验证维护应用的安全。

(5) 研究代码的安全技术。通过研究 dex 文件的保护技术,选择相应的措施保护 dex 文件,防止应用被反编译^[21]。dex 文件中包含了 App 代码的所有信息,而 dex 文件的破解并不困难,一旦破解,应用和代码的安全性就得不到保障。因此采用 DES 对 dex 文件进行加密。DES 密钥根据每个插件模块随机产生,从而更进一步保护 dex 文件。

(6) 研究动态代码注入技术和动态代码注入的相关工具,利用动态代码注入解决单个 dex 文件的引用问题^[14]。在系统优化 dex 文件的过程中,系统会对当前类与直接引用的类是否在同一个 dex 文件中进行检测,若都在同一个 dex 文件中,就会被加上特殊标志,从而导致热修复的失败。

(7) 实现模块化更新和热修复功能。将从网络上加载下来并存放在固定目录下的 APK 文件或者 dex 文件动态加载至内存,然后利用上面分析和研究的相关技术方法实现宿主应用新模块增加和 bug 修复功能。

1.4 本文的组织结构

本文的主要内容可以分为六章:

第一章是绪论部分,本章主要是介绍 Android App 插件化的研究背景、现状、意义,以及接下来所要研究的内容。

第二章是动态加载技术与相关机制分析,本章主要针对在插件化过程中需要用到的关键技术或者机制进行分析,通过对它们的分析得出它们的使用方法和时机。

第三章是粒度化的安全功能插件化实现机制分析,本章主要通过分析插件化机制,得出插件化在系统中的位置、各组件插件化的思路、热修复的原理及思路等。

第四章是粒度化的安全功能插件化实现机制设计,本章主要首先对插件化机制的总体进行设计,接着是对各个模块进行详细的设计。

第五章是功能插件化机制的实现,本章主要是对第四章所设计的机制进行具体的实现与验证。

第六章是总结与展望,本章主要是对现在研究实现的工作进行总结,对本文所研究内容的一些不足进行一个总结,并为后面的完善提供一个思路。

2 动态加载技术与相关机制分析

2.1 动态加载技术

实现本文插件机制的基础就是动态加载机制^[27]。通常应用已经发布后，需要兼容扩展系统的功能，而动态加载技术就可以满足这种需求。动态加载技术的实现一般有两种方式：一种是重定位的加载目标模块，其实质是 API 的重定位^[29]；另一种则是利用 Java 虚拟机的特性，Java 虚拟机本身的类加载机制提供了动态加载的可能性^{[29][30]}。因为本文所研究的是 Android App 的插件化，大部分 App 是利用 Java 来编码的，所以本文对 Java 类的动态加载机制进行研究与分析，利用 Java 虚拟机与 Android 虚拟机的相似性得出适合 Android 应用的动态加载机制。

2.1.1 Java 类的动态加载机制

.java 文件被 Java 编译器编译成.class 文件，该文件是以字节码的形式存在的。Java 虚拟机 JVM 是通过加载.class 文件进行解释执行的，这种方式实现了 Java 程序的跨平台。类被 JVM 加载进内存之后才能被其他类引用，将类加载进内存的工具是 Java 的类加载器。Java 的类加载机制为 Java 提供了三个默认的加载器，其中负责加载 JDK 的核心类库的是 Bootstrap ClassLoader 也叫启动类加载器；负责加载 Java 的扩展类是 Extension ClassLoader 也叫扩展类加载器，它默认加载的是 lib/ext/目录下所有的 Jar 包；负责加载 App 中 classpath 目录下的所有 jar 文件和 class 文件的是 App ClassLoader，也叫系统类加载器^[33]。Java 除了默认的三个加载器外，开发者也可以通过自定义加载器来加载自己的代码。如图 2-1 是 Java 默认加载器的加载机制。

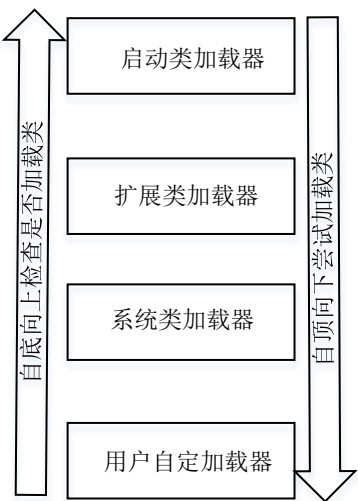


图 2-1 Java 类加载机制

Figure2-1 Java class loading mechanism

ClassLoader 采用的是双亲委托模型来加载类的，每个类都有父类加载器，它们之间是一种包含关系而不是类的继承关系^[35]。例如 JVM 中内置的启动类加载器，可以被用作其他 Classloader 的父类加载器，但是本身是没有父类加载器。当一个 class 文件要被加载时，

首先加载它的 `ClassLoader` 会委托它的父类加载器去加载这个文件，然后父类加载器再交给它自己的上一级，一直到最顶层的启动类加载器。假如没有加载成功，则下一级加载器加载，并且就这样依次向下一级递交任务。如果 `class` 文件的加载器的发起者都加载不到这个文件那么系统就会报 `ClassNotFoundException` 异常，如果这个 `class` 文件被成功加载到了内存中，系统就会将这个 `class` 在内存中的实例对象返回。Java 虚拟机中类加载的层次如下图 2-2。

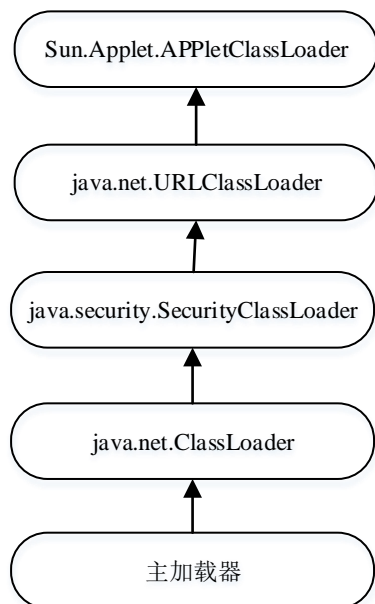


图 2-2 Java 虚拟机类加载的层次

Figure2-2 The level of the Java Virtual Machine class loading

Java 类的加载可能是利用不同的 `ClassLoader` 来实现的，但是采用的方式都是双亲委托，这种类的加载机制为实现动态加载提供了可能性。开发者可以很容易的构造自己的加载器或者是获取系统的加载器来加载自己的类扩展系统的功能。

2.1.2 Android 虚拟机类加载机制

Android App 大部分的编码语言采用的是 Java, Java 代码的运行都是依靠 Java 虚拟机解释执行的^[29]。因此 Android 程序的运行环境也必须是虚拟机，只是 Android 程序运行的虚拟机与普通的虚拟机还是存在一些差别。JVM 运行的是字节码，而 Android 的虚拟机运行的是 `dex` 文件。`dex` 文件本质是经过特殊处理的一种字节码，这种字节码是 Android 虚拟机可以执行的代码，它需要利用 `dx` 工具将 Java Compiler 编译之后的 `class` 文件进行转换。Android 虚拟机本身就是从 Java 虚拟机的基础上演变过来的，所以在设计之初，Java 程序就可以无缝衔接到 Android 虚拟机中运行^[36]。因此 Android 虚拟机加载类的机制与 Java 中的是类似的，只是 Android 与 Java 虚拟机加载的类的 `ClassLoader` 不相同，加载的层次不同。如图 2-3 是类在 Android 虚拟机中加载的层次。

从图中也可以发现 Java 虚拟机中类加载的机制与 Android 虚拟机中类加载层次是类似的，所以 Java 中动态加载的机制完全可以运用到 Android App 的开发中。

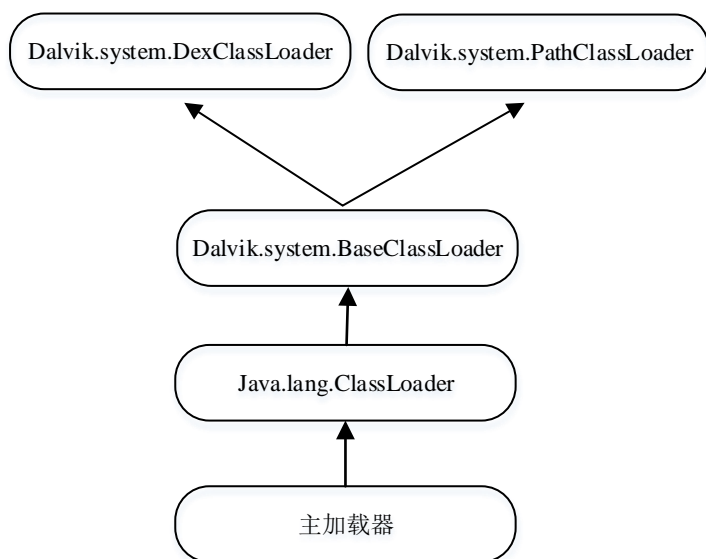


图 2-3 Android 虚拟机类加载的层次

Figure2-3 The level of the Android virtual machine class loading

2.2 HOOK 机制

HOOK 的英文翻译为“钩子”，钩子的意思是时间在抵达终点之前截获和监控事件的传输过程，当一个钩子钩上需要的事件时，根据自己的处理逻辑去处理特定的事件。这个钩子可以将自己的代码“注入”被 HOOK 的进程中从而成为目标进程的一部分。图 2-4 是 HOOK 技术的原理图。

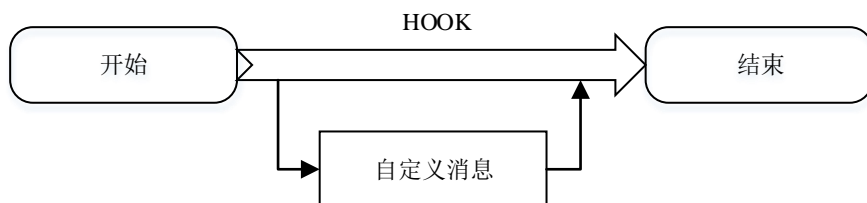


图 2-4 HOOK 技术原理图

Figure2-4 Schematic diagram of HOOK technology

HOOK 技术是插件化过程中的关键技术，对于插件来说它并未在系统中进行安装和注册，而 Android 系统中存在一项安全措施就是未注册安装的组件是没有权限访问系统的服务和资源的。因此插件要正常运行，就必须进行 HOOK，欺瞒系统进行正常服务和资源的访问。

Android 的开发模式分为 Native 模式和 Java 模式，这两种模式可以通过 jni 技术来进行交互^[30]。在一般情况下，Java 模式下可以通过调用 native 方法来干涉 Native 模式，但是 Native 模式不能在 Java 的支持下来干涉 Java 模式。所以在 HOOK 时就可以从两个方面来进行，理解 ELF 文件是对 Native 层 HOOK 的难点；而了解虚拟机的特性和 Java 反射的使用则是 HOOK Java 层的必要条件。Native 层的 HOOK 是针对 NDK 开发出来的.so 库文件中的方法（包括 Android 系统底层的 Linux 方法）来进行重定向的。该重定向对 so

库文件中的全局偏移量表或者符号表进行修改,从而实现重定向的目标;Java 层的 HOOK 是通过改变虚拟机调用加载 dex 文件的方式,这种方式的改变是利用 Android 平台的虚拟机注入和 Java 反射来实现的,从而达到重定向的目的。利用 Java 语言自身的特性完成 HOOK 过程,通常需要两个步骤:

(1) 寻找 HOOK 点,HOOK 点的寻找往往是比较困难的,通常都是通过分析系统源码得到相应的 HOOK 点。一般类的单例方法或者是静态变量是很好的 HOOK 点,因为利用 Java 的反射机制很容易就获取到它们的对象,因此这两种类型 HOOK 起来比较简单方便。

(2) 构造 HOOK 的原始对象的代理类。通常代理类包括两种:动态代理和静态代理。静态代理类在没有任何条件限制的情况下每个方法在调用之前必须先调用原始对象的方法,它维护的是原始对象中的成员变量;动态代理相对于静态代理要复杂一些,因为动态代理类有个规则那就是原始类需要实现接口,否则就不能进行相关操作。动态代理的实质是生成一个代理类的字节码,原始类中的所有接口方法这个类都会自动的去实现,再利用反射机制来调用接口中的一切方法。图 2-5 是动态代理的过程。

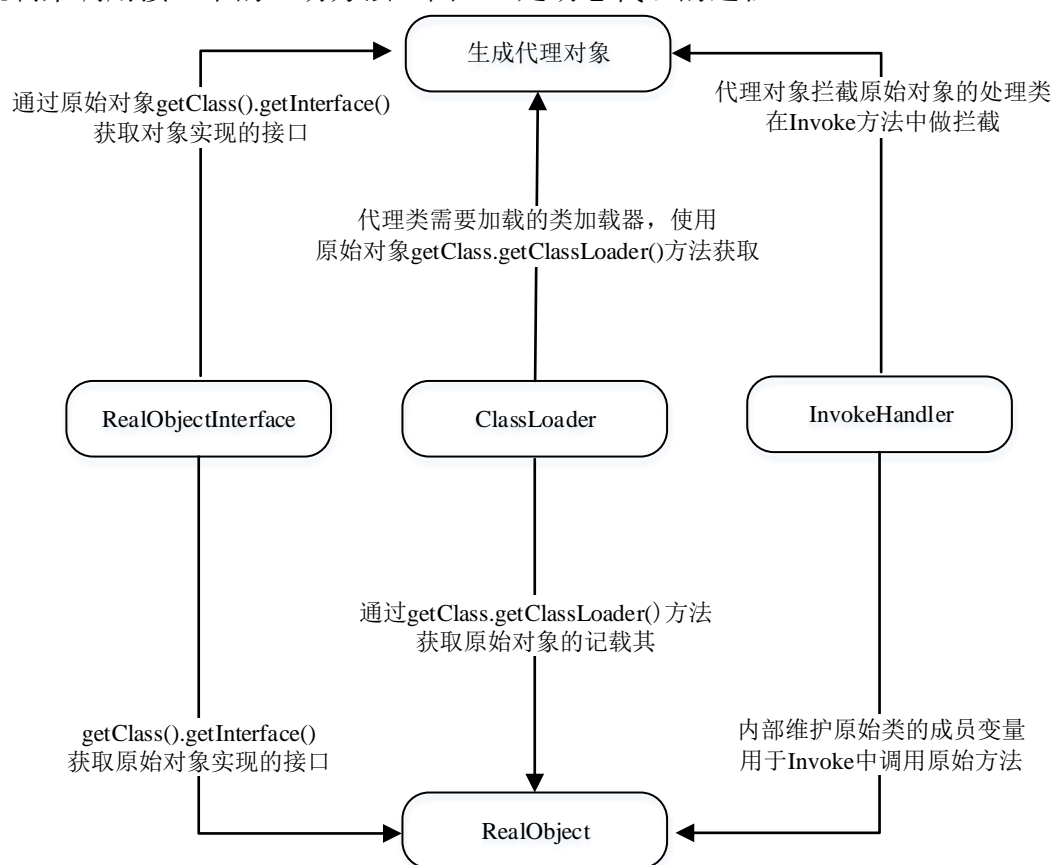


图 2-5 动态代理过程

Figure2-5 The flow of dynamic agent

HOOK 机制的实现需要利用上述两个步骤,其中寻找 HOOK 点是比较麻烦的。由于插件中的组件的未安装和注册,要使它们能像宿主中已经注册安装的组件无异,必须寻找合适的 HOOK 点进行相应的 HOOK,这就要求认真的查看研究 Android 系统的源码,在

源码里寻找各个 HOOK 点。

2.3 Binder 机制

Binder 是 Android 系统进程间通信的方式之一，Android 系统是基于 Linux 内核开发的^[37]。Linux 进程间的通信方式有管道、信号量、Socket 等，但是 Android 系统并没有选择其中的任何一种，而是选择了 Binder 机制。其中的主要原因在于移动设备的硬件性能比较差、内存不足。

Binder 所使用的通信方式是 Client-Sever，其中 Client 端一般包含多个进程，它们会向 Server 端发出请求以获得对应的服务；而 Sever 端只有一个进程是作为提供某种服务的一端。Client-Server 端的通信需要具有下面两个特点：第一点为方便来接收 Client 的请求，Server 端需要给出确切的访问接入点或者地址，同时 Sever 的地址必须通过某种方式来告知 Client；另一点是双方通信需要指定相关协议来进行数据的传输。在 Server 端，Client 向 Server 请求某种服务时就是利用 Binder 来实现的，Server 为 Client 提供的服务的接入点就相当于 Binder 在通信方式中的位置；在 Client 端，Binder 充当的角色则像是通向 Server 的一条管道而不是服务的接入点，如果要与 Server 进行通信就必须先和 Binder 建立连接然后才能通过 Binder 与 Server 通信获得所需要的服务。

ServiceManager、Server、Client 和 Binder 驱动是 Binder 的通信模型框架的四种角色。其中前三种都是运行在用户空间，只有 Binder 驱动是运行在内核空间。通信模型中的四种角色对应到互联网中：Client 对应客户端，Server 对应服务器，驱动则对应路由器，ServiceManager 对应域名解析器^[14]。下图是 Binder 的通信模型。

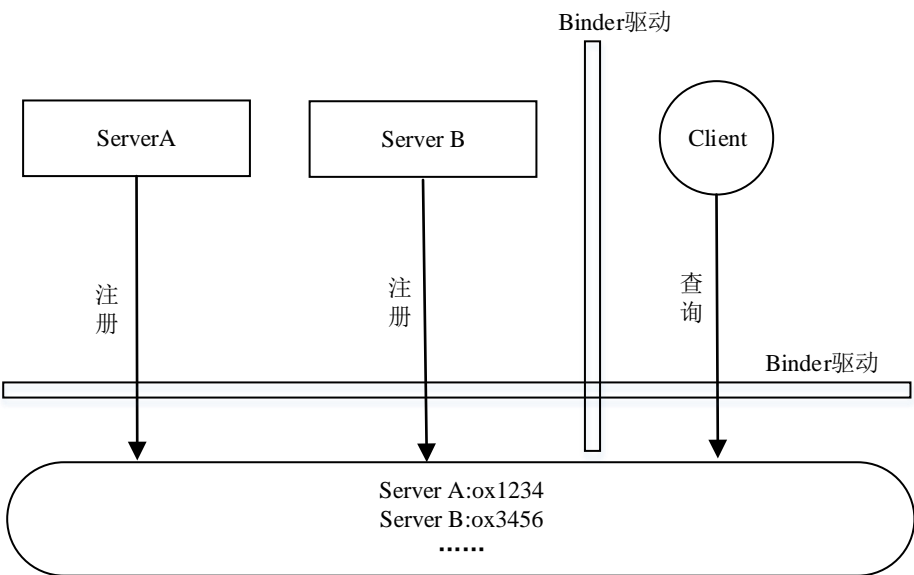


图 2-6 Binder 的通信模型

Figure2-6 The communication model of Binder

从图中可以看到通信过程为第一步创建通信录（下面简称 SM），首先需要建立 SM，Server 进程向 Binder 驱动提出申请生成 SM。Binder 驱动处理之后，ServiceManager 进程

开始负责管理 Service，但是此时的 SM 是空的；第二步，每个 Server 进程在启动完成后都需要向 SM 请求进行注册，SM 将进程的名字与地址记录下来，完成注册；最后一步就是 Server 与 Client 进行通信，Client 首先向 SM 请求获取所需服务的地址，Client 根据 SM 返回的地址对相应的服务进行访问。

从图 2-6 中可以看到 Client 与 Server 的通信过程，但是对于 Binder 是如何跨进程进行通信的，上图中并没有说明。处于内核态的进程可以访问所有用户态进程的数据，但是处于用户态的两个进程并不能相互访问彼此的数据，因此最简单的方式就是进程之间需要进行数据的交互时，首先将其中一个进程的数据复制到内核空间，接着再将内核空间的数据复制到另一个进程中，这种方式就是将内核作为一个中转，实现数据在进程间的传递。而 Binder 机制的数据传输并不是利用上述方式完成的，而是通过 Binder 驱动来完成进程间的数据的交互。图 2-7 是 Binder 跨进程通信图。

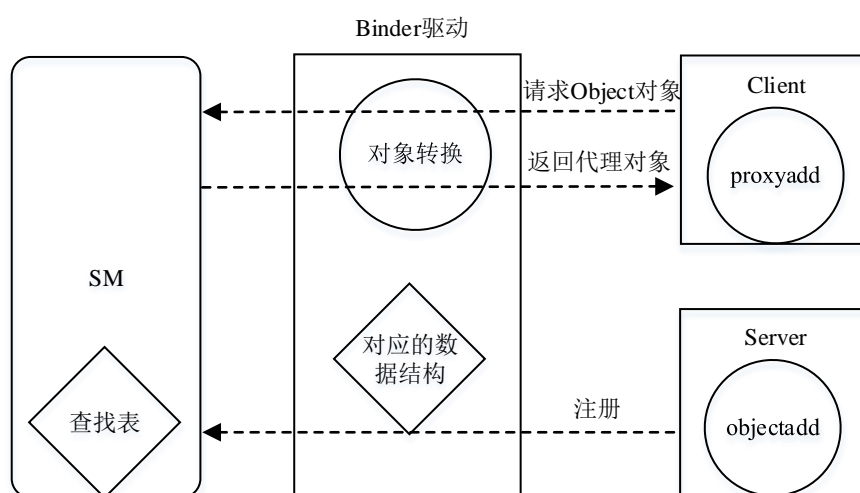


图 2-7 Binder 跨进程通信

Figure2-7 The cross-process communication of Binder

从上图中可以看出，当 Server 进程向 SM 请求注册时，会通过 Object 对象的 add 方法将对象添加进 SM 中；当 Client 需要请求某种服务时，先向 SM 进行查询，找到对应的 object 对象。进程之间所有的数据都需要通过内核空间的驱动，在数据通过驱动的时候，驱动并没有一成不变的将数据原封不动的进行传递，它并不会对 Client 进程返回一个真实的 object 对象，而是返回一个类似于 object 的代理对象，这个代理对象也具有一个 add 方法，但是这个方法的功能并不是具有 Server 进程中真正 object 对象 add 方法的功能，它只是一个代理，它的功能只是将参数包装好后交给 Binder 驱动。驱动获得消息后，通过查找表然后通知 Server 进程，调用 Server 进程的 add 方法，Server 进程将执行后的结果返回给 Binder 驱动，驱动再把结果返回给 Client 进程。这样 Binder 的跨进程通信就结束。

Binder 使用的 CS 的通信方式具有简单高效、安全性好的优点，是 Android 进程通信的中流砥柱^[41]。在 Android 系统中，应用想要获取系统服务、资源或者权限，几乎都是通过 Binder 的跨进程传输实现的。而需要获取到系统的访问权限，系统要求应用必须是经

过安装注册的，未经安装注册的应用系统会通过一系列的检测而拒绝访问。通过上面 Binder 通信机制的分析，伪造代理的 Binder 对象，绕过系统的检测与系统进程进行通信，等到再次回到应用进程中再将对象还原的这种方式是可以实现插件中组件的运行。

2.4 本章小结

本章首先是对插件化过程中会用到的技术和相关机制进行分析。首先是对动态加载技术进行分析，从分析 Java 类的动态加载机制过程中，得到 Android 虚拟机也可以实现 Java 虚拟机下的动态加载机制；接着是对 HOOK 机制进行了详细的分析，得出在 Android 系统中 HOOK 的原因以及 HOOK 的过程；最后是对 Binder 的通信机制进行了详细的分析，得出在插件化中 HOOK Binder 至关重要。

3 粒度化的安全功能插件化实现机制分析

本文所研究的插件机制根据插件化粒度的大小分成了两个大类：第一类大粒度的插件化，本文称之为模块化更新；第二类是小粒度的插件化，是热修复功能。模块化的更新也可以修复 bug，但是热修复关注的是方法级别的修复，粒度会更小。这两种方式实现原理还是有比较大的差别。目前开源的插件化框架针对的都是大粒度的模块更新，并且都没有使用相应的安全机制来保护应用的安全。本文在研究插件机制的同时，通过分析 Android 应用的安全机制，为插件机制设计了相应的安全机制。对于大粒度的插件化模块化更新本文采取的是 APK 接入宿主应用，对小粒度的插件化热修复本文则选择 dex 文件接入。

3.1 插件化的位置

Android 系统本身就是基于 Linux 内核开发的程序，它不仅拥有最高权限，还可以对硬件资源进行调配^[41]。因此 Android 系统的实质是一个运行在 Linux 内核之上的应用程序。Android 系统框架图如图 3-1。

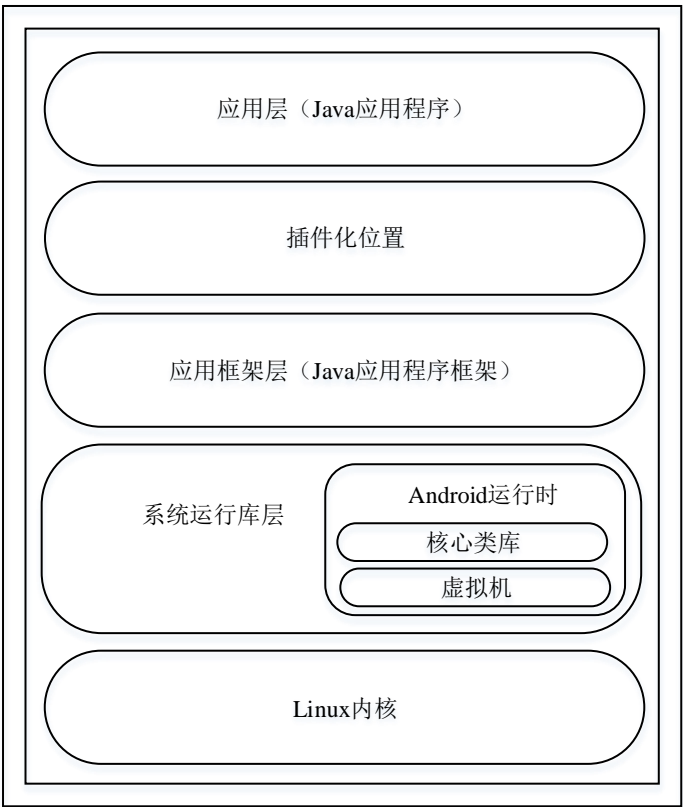


图 3-1 Android 系统框架

Figure 3-1 Android system framework

Android 系统框架有四层，从低到高依次是：操作系统（OS）、库（Libraries）和运行环境（RunTime）、应用程序框架（Application Framework）、应用程序层（Application）^[43]。

（1）Application 层是用来与用户交互的，用户可以看见应用界面并且可以进行相应的操作。这类应用通常是基于 Java 语言开发的，能够独立的完成某些功能。在这层上开

发人员可以通过使用框架提供的 API 来开发应用程序。

(2) Application Framework 层, 在这层开发人员可以使用 Android 基本应用程序的 API 并和复用应用框架中的模块和服务, 从而开发出更优秀的应用。

(3) Libraries 和 RunTime 这层涉及到底层, 开发普通的应用不会涉及到对该层的操作。

(4) OS 的核心服务依赖于 Linux2.6, 该层为 Android 提供了安全性、内存管理、进程管理、网络堆栈等服务。驱动程序模型包含有 Display Driver、Keypad Driver、Camera Driver、WiFi Driver、Flash Memory Driver、Audio Driver、Binder Driver、Power Management。

Android 操作系统的最上层是应用层, 运行各种程序的, 系统为每个应用都分配一个独立的虚拟机, 保证应用的安全性和独立性; Framework 层主要是为了方便开发者的开发提供了一些核心组件, 同时还提供了许多管理组件的服务; 库和运行环境层涉及到的是底层, 在日常的开发中是用不到的; 最底层就属于 Linux 核心层, 这层主要是硬件的驱动程序, 因此本文考虑从应用层和 Framework 层之间着手实现插件化, 其核心思想就是通过运用 HOOK 机制在插件与 Android 系统间开辟一个供插件运行的独立的进程空间。

3.2 插件加载技术

通常在 Android 系统中应用都是以 APK 安装包的形式进行安装的, 系统通过解析 APK, 获取对应的资源和 dex 文件, 再将 dex 文件中的类加载进入应用内存中^[44]。插件接入宿主程序的方式有两种: 一种是 dex 文件, 另一种是 APK 包。对于热修复模块, 它针对的是类文件的修复, 是对 Java 代码的修复, 不需要打包成 APK 包, 可以省去解析 APK 的过程, 从而加速应用的启动, 因此可以直接打包成 dex 文件; 对于模块化更新, 本文采用的是 APK 的接入宿主应用的方式, 因为模块化的更新, 往往都是新功能的扩充, 通常伴随有代码和资源的更新。

APK 文件其实质是一个压缩文件。其中包含有 Manifest 文件、META-INF 目录、classes.dex 文件、res 目录、resources.arsc。

(1) AndroidManifest.xml 文件中记录了应用的版本、名字、权限、等信息, 每个应用中必须定义和包含这个文件, 否则应用不能正常运行。在应用安装或者需要把 APK 上传至某个应用市场时, 这个文件是必不可少的, 同时也是非常重要的。

(2) META-INF 目录对于保证 APK 包的完整性和系统安全非常重要, 里面存放的是签名信息。

(3) dex 文件是唯一可以直接加载并运行在 Android 虚拟机上的文件。dex 文件的产生通常都是利用 ADT 复杂的编译过程, 经过 java 源代码转换成的。而运行 dex 文件效率相对较高是因为在该文件中共用了许多类的名称、常量字符串、从而使它体积大大减小。

(4) res 目录是用来存放资源文件的, 主要包括图片资源, 布局文件。

(5) Resources.arsc 是编译后得到的二进制文件^[46]。通常这个文件中存储了是本地化

和汉化资源。

Android 系统在安装的过程中对 APK 进行解析，获取到对应的信息，将信息缓存到系统中。这些信息是应用中组件注册、启动、运行的关键。

3.2.1 代码加载方式分析

编译后的 Java 代码都在 dex 文件中，要将 dex 文件中的类加载进入应用的内存中，从第二章中知道需要构造类的加载器。Java 的动态加载技术与 Android 系统的动态加载技术虽然有些差异，但是本质上是同一种技术。因此 Java 动态加载机制同样适用于 Android 虚拟机。

DexClassLoader 和 PathClassLoader 是 Android 源码中 BaseDexClassLoader 的两个子类，一般使用这两个加载器来加载类。这两个加载器的区别在于 PathClassLoader 只能加载系统中已经安装过的 APK，而 DexClassLoader 不仅可以加载 jar/apk/dex，还可以加载在 SD 卡中没有安装的 APK。在进行动态加载开发时，DexClassLoader 已经可以满足开发者的需求。但是开发者为了让模块具有更强的隔离性也可以通过自定义的 ClassLoader 加载相应的类。

本文通过构造插件自己的加载器来对插件中的代码进行加载，这样也可以将插件中的类与宿主类进行隔离，而不影响应用的正常运行。因此利用 Android 虚拟机自身动态加载的机制可以实现代码的动态加载。

3.2.2 资源编译加载方式分析

在开发过程中，应用程序资源是应用程序十分重要的一部分。assets 和 res 的目录都会存在于创建的应用工程中，这些目录下存放的就是应用程序资源。在开发时需要获取这些资源时都必须要先获取 Resources 对象，再根据 Resources 访问资源。获取 Resources 资源对象有如下两种方式：

(1) Context 获取 Resources 对象：通常在 Service 与 Activity 中都是使用 getResources() 方法来获取 Resources 对象的。但是在这两个组件中调用该方法的实质是调用 ContextImpl 中对应的方法，因为该方法存在于抽象的 Context 内部，并且调用该方法的实质是调用实现 Context 接口的 ContextImpl 中的方法。图 3-2 是通过 Context 获取 Resources 的过程。

从图中可以看出 Resources 对象的获取最后都是通过 ActivityThread 对象调用 getTopLevelResources() 来实现的。并且是由 ActivityThread 对象利用 HashMap 保存起来，但是为了避免内存溢出的发生，在内存紧张时可以及时释放 Resources 对象所占用的内存，Resources 对象都是以弱引用的形式保存下来的。HashMap 的 key 是根据资源的路径生成的。AssetManager 对象是创建 HashMap 的值 Resources 对象时必须创建的另外一个对象，在初始化该对象时是通过资源对应的路径，这个路径包括了应用程序资源的路径和 Framework 资源路径。getXXX(int id) 访问资源是 AssetManager 中许多获取资源的方法中的一种，系统资源用参数 id 小于 0x10000000 表示，应用资源的参数 id 则 0x70000000

以上的数值来表示。因此在 aapt 编译应用时，系统资源被 aapt 编译得到的 id 都会是小于 0x10000000 的。

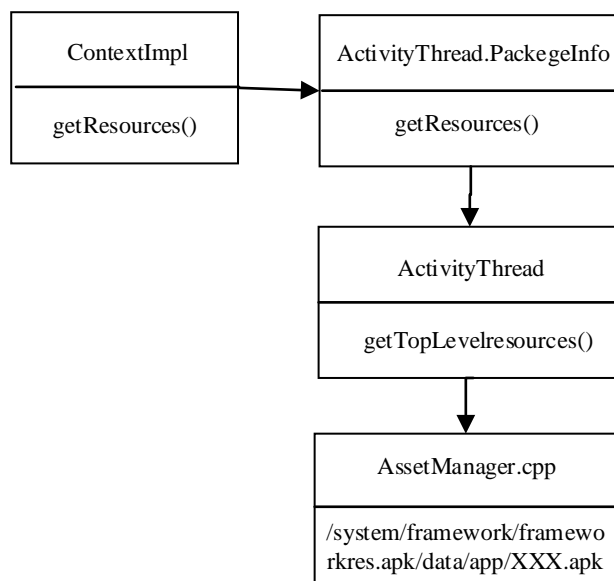


图 3-2 Context 获取 Resources

Figure3-2 Getting Resources via Context

(2) PackageManager 获取 Resources 对象：通过这种方式主要获取的是第三方应用程序中的资源,比如说主题等。如图 3-3 是系统源码利用 PackageManager 来获取 Resources 对象的过程。

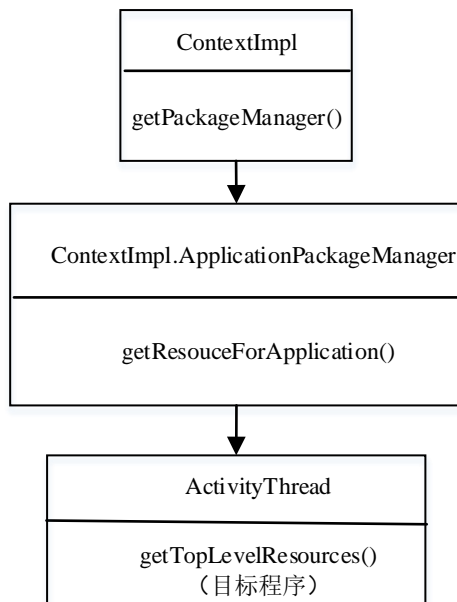


图 3-3 PackageManager 获取 Resources

Figure3-3 Getting Resources via PackageManager

从上图可以看出 Resources 的获取也是调用 ActivityThread 中 getTopLevelResources() 方法来获取的。后面保存 Resources 对象和访问具体资源的过程与 Context 一致。

从上面两种加载资源的方式可以看出，最终访问资源的方式都需要通过 AssetManager

对象和资源路径来构建 Resources 对象访问资源。因此在宿主程序加载插件资源时，可以通过 Java 反射机制来构建这两个关键因素，从而加载插件模块中的资源。

在 Android 系统中，aapt 编译的结果都是一个资源对应一个常量 id，这些在 Java 源码层面生成的 id 都会记录到对应的 R.java 文件中，参与到后面的代码编译过程。在应用工程进行编译的过程中会生成所有资源的 id，这些 id 会被作为常量统一存放在 R 类中供其他代码引用。因为资源的 id 都是由系统自动编译生成，因此插件生成的资源 id 与宿主程序生成的 id 很可能产生冲突，而导致应用访问资源失败。

资源的编译是由系统按照 0xPPTTNNNN 的规则自动生成的。其中 PP 段是用来标记应用 APK 的包名，并且默认情况下应用程序的值是 07f，而系统资源的值是 01f；TT 段是用来标记如图片、布局等资源类型，资源类型相同的，TT 值是相同的，但是同样的 TT 值不代表是同一类的资源，因为系统每次编译的时候可能同一类资源的值不同；资源类型的 id 则是 NNNN 来表示的，它默认是从 1 开始累加的。为了避免宿主程序与插件模块 id 的冲突，可以利用上述规则来固定 id 的生成。因此本文采用固定 TT 值的方式来避免 id 的冲突。固定 TT 值的方法可以通过提供一份 public.xml 的方式，在这个文件中指明每种类型资源的 TT 值开头。

3.3 大粒度插件化模块化更新

3.3.1 模块隔离性分析

Android 系统是以 Linux 内核为基础，辅以定制的 C 语言本地库和内嵌独有的 Dalvik 虚拟机的 Android runtime 环境^[2]。由于 Android 系统和系统上运行的应用都是运行在 Linux 内核之上，因此 Linux 进程负责 Android 虚拟机的初始化。每个 Android 应用的启动都意味着一个新进程的启动，对应着一个虚拟机的实例。因此在 Android 系统运行的过程中会出现大量的独立的进程和虚拟机的实例，为解决虚拟机分配的问题，google 特地设计了一个孵化虚拟机进程的孵化器 Zygote，通过这个孵化器加快虚拟机的启动和分配。Linux 内核启动后，Zygote 接着启动起来，然后它会利用本身创建的监听接口对新进程请求进行监听，一旦出现创建新进程的请求，Zygote 孵化器就会通过自我复制来迅速的创建新进程。当自我复制结束后，就会传递对应的参数对新进程进行初始化，通过这种方式，新进程就可以独立的运行，也就是每个虚拟机可以独立的运行，从而达到 Android App 沙箱隔离性的效果。

本文设计的模块化更新考虑到 App 的稳定性，让插件与插件之间，插件与宿主程序之间隔离开来，实现崩溃和模块的隔离。本文设计这样的机制，在这个机制中每个插件模块都会启动一个新进程，这个新进程是宿主应用进行随机分配的，通过这种方式来实现插件独立的运行在自己的进程中。虽然各个模块独立的运行在各自的进程中，但是在形式上它们仍然属于同一应用，同一应用中避免不了需要进行数据的共享。因此可以利用不同进程间共享 id 得方式来实现数据得共享，也可以利用 Intent 实现小数据的交互。

3.3.2 四大组件插件化分析

Android 的四大组件开发者都再熟悉不过了，四大组件也是应用程序的核心组成部分，每个应用程序都是由其中的一种或者多种组成的。因此在开发一个插件化框架时，这些组件就是插件化的最基本单元，因而只有处理好这些组件插件化过程中遇到的问题，才可能顺利地实现插件机制。四大组件各有各的特点，其中的某些特点可能会为插件化带来限制，只有分析出它们的特点，才能制定相应的解决办法。下面是四大组件的几个特点：

（1）注册制：由于 Android 应用进程的产生都是利用 Zygote 孵化器孵化而来的，在孵化的过程中需要获取应用的关键信息，系统才能进行相应的初始化工作。而系统需要掌握的信息就包括四大组件的相关信息，并且在安装每个应用程序时，就需要完成四大组件的注册，通过注册的形式来告知系统各个组件的信息。这个注册的形式是在 AndroidManifest 文件中配置声明，如果没有在该文件中进行注册的组件，系统是找不到对应的组件的。在四大组件中，广播的注册比较特殊，它具有两种形式的注册方式：包括静态和动态。通过 AndroidManifest 方式注册属于静态，不管系统是否运行，只要系统监听到特定的广播事件发生，系统就将广播对应的行为唤醒；通过代码所注册的广播叫动态广播，只在代码注册之后，注销之前才能接收到相应事件。

（2）激活方法：Activity，Service，Broadcast 这三大组件的特点是都是使用 Intent 的异步消息机制来进行匹配和激活的。Action，Category，Data，ComponentName 是 Intent 匹配的关键项。这三大组件都是通过上述几个关键信息来组合匹配需要激活的组件。而 ContentProvider 是根据 Uri 来定位内容的提供者^[47]。

（3）关闭方法：Activity 是前台可视化的窗口载体，因此调度和使用 Activity 相对而言会复杂不少，并且 Activity 的实例并不会自动被虚拟机 gc，这就要求开发者在不需要 Activity 时主动并及时调用 finish() 方法关闭相应的 Activity 实例。Service 是处于后台长时间运行的组件，它的关闭方式是根据启动 Service 的方式来决定的。Service 的启动方式包括 startService() 和 bindService() 两种。如果使用 startService() 进行启动，则需要使用 stopService() 来关闭；使用 bindService() 进行启动，则需要使用 unbindService() 关闭。内容广播接收者 Broadcast 与提供者 Content Provider 它们的关闭方式比较特殊，它们一旦被激活就会按照对应的逻辑执行代码，一旦执行完毕，它们的生命周期也就结束了，所以它们不需要手动进行关闭。

（4）任务栈：任务栈主要是为了满足 Activity 复杂的生命周期而设计的^[2]。因为在 Android 应用程序中 Activity 的启动模式有多种。启动模式的选择都是根据不同的场景来决定的，不同的启动模式对任务栈的操作也是不同的。Android 系统是一个多任务操作系统，为了满足应用的性能，方便用户的切换，就需要将暂时使用不到的任务进行缓存，同时缓存在控制后台的任务又不能过多，因此为了满足使用环境下最大限度的不积压过量的任务信息，才设计了不同的启动模式来保障应用性能。Activity 因为任务栈的存在而具有

一个独有的机制，被称为生命周期。

3.3.2.1 Activity

Activity 是展示型组件，主要是向用户去展示一个个界面和接收用户的输入信息进行相应的交互^[48]。四大组件中 Activity 是最重要的组件。站在用户的角度，他们只能感知到 Activity，所以在他们看来 Activity 就是应用的全部。应用程序都是在不同的 Activity 之间进行跳跃的。在前面也讲到 Activity 是通过任务栈来统一进行管理的，而任务栈的管理则是由系统进程 ActivityMangagerServer（下面简称 AMS）管理的，这个系统进程从开机初始化就在系统中运行了。所有 Activity 都是有任务栈来进行管理的。图 3-4 是 Activity 的任务栈结构。正是因为任务栈的关系，使 Activity 具有多个不同状态，这些状态的变迁就构成了 Activity 的生命周期，其中任务栈中 Activity 状态的变化就对应着 Activity 生命周期中状态的转换。图 3-5 是 Activity 生命周期的切换图。

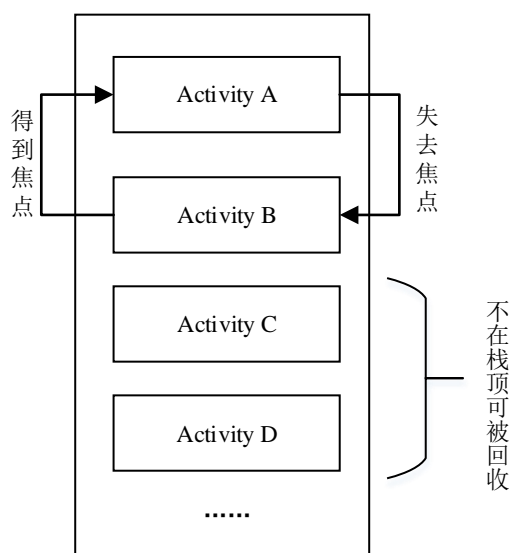


图 3-4 Activity 的任务栈结构

Figure3-4 Activity stack structure of the activity

Activity 的启动是由 startActivity()方法负责的，并且系统只能找到在 AndroidManifest 文件中已经注册了的 Activity 进行启动。启动 Activity 的方式有两种：一种是通过 Launcher，也就是用户通常启动 App 的方式；另一种就是通过 Intent 的异步消息来启动的。但是两种启动方式最终都是通过 startActivity()方法启动。

对于 Activity 的插件化，管理 Activity 的周期是十分关键的。Activity 周期的切换，不光是靠系统来控制的，用户的操作也会影响 Activity 的生命周期。因此要达到 Activity 插件化的效果就必须将 Activity 生命周期的管理委托给系统，必须寻找合适的 HOOK 点对系统的关键模块进行 HOOK。

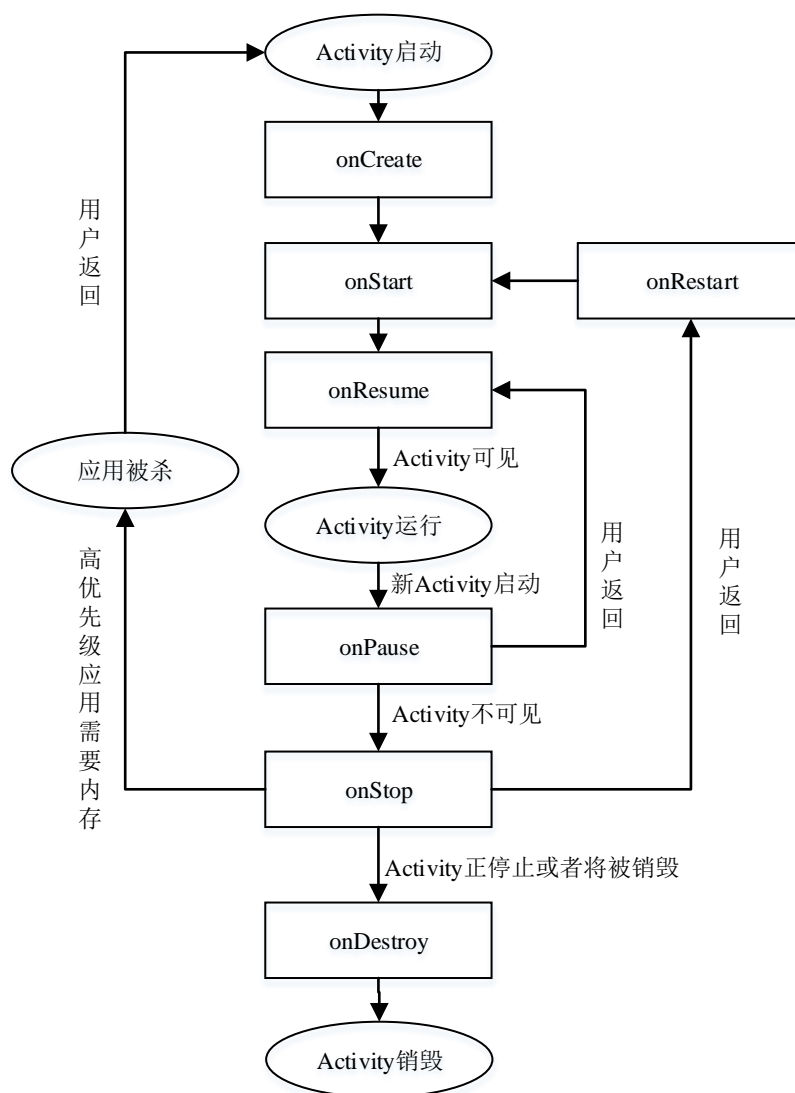


图 3-5 Activity 生命周期的切换流程

Figure 3-5 The switching flow of Activity cycle

Android 应用的启动就是孵化一个虚拟机的实例，在 Android 系统中，Zygote 就是虚拟机实例的孵化器。在用户的角度来看，他们所能感知的是 Activity，所以应用的启动，最终展示的是 Activity。总的来说，应用的启动需要先通过 Zygote 孵化出一个虚拟机的实例，创建进程，然后再去启动 Activity。通常启动一个 Android 应用都是点击桌面的快捷方式，这种启动 Activity 的方式是利用 Launcher 来启动，Launcher 的实质也是一个 Activity^[44]。图 3-6 是应用启动流程也是 Activity 的启动流程。

从应用的启动流程图也可以看出来，应用的启动实质也是 Activity 的启动。而 Activity 的启动并不是由一个进程完成的，而是应用进程和系统进程 AMS 配合完成的。因而若要实现 Activity 的插件化就可以先在 Manifest 文件中注册代理的 Activity 组件，然后通过李代桃僵的方式，完成插件 Activity 启动的过程。因为 Activity 具有多个启动模式，因此在注册代理组件时，需要为每个启动模式的 Activity 进行注册。

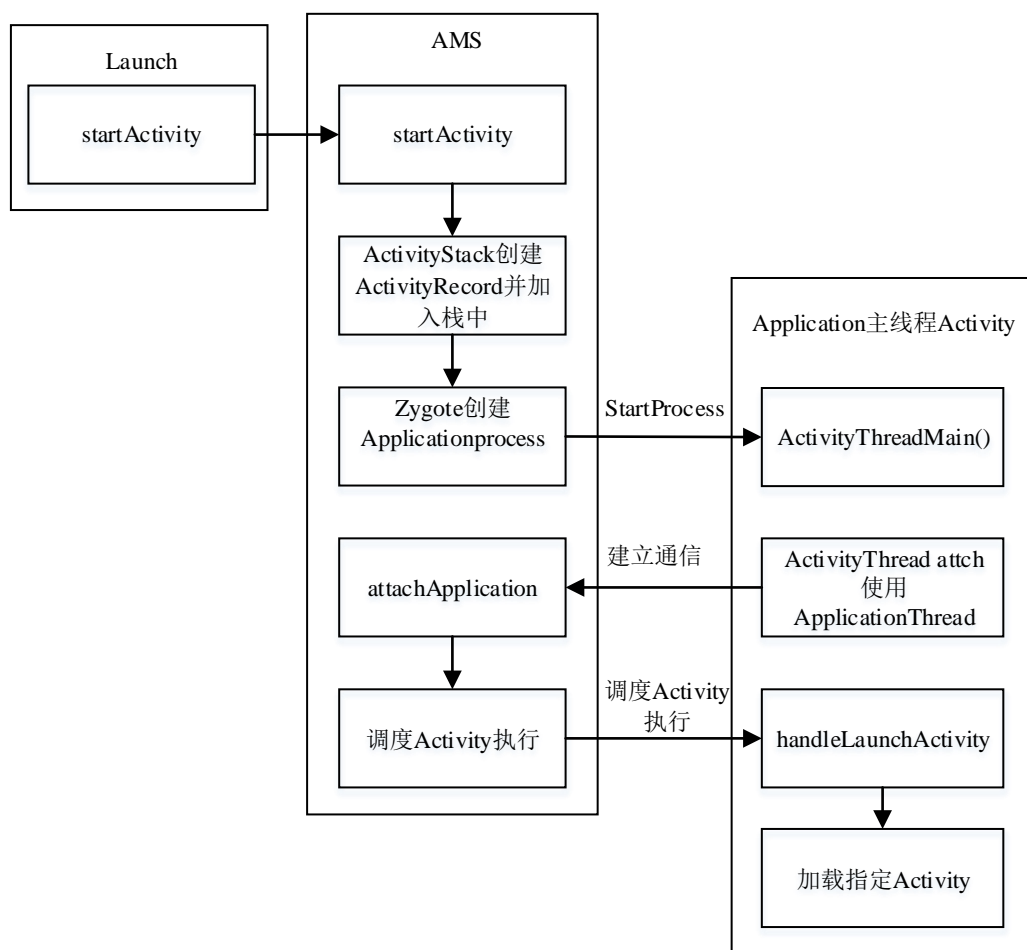


图 3-6 应用启动流程

Figure3-6 The startup flow of application

3.3.2.2 Service

Service 是一种计算型组件，主要用于后台执行一系列的计算任务^[33]。用户感受不到 Service 的存在，是因为 Service 组件是工作在后台的组件。Service 与 Activity 也不同，它只有一种运行模式。系统中有大量的 Service 来对接系统的各类服务。而 Service Manager 就是用来定位所需要的服务具体是哪一个。当客户端需要向服务端请求某种服务时，就会先访问 Service Manager，Service Manager 根据客户端需要获取的服务的名称在服务注册表中找到对应的服务的地址，接着再去访问需要的服务。事实上 Service Manager 也是一个 Service，只是它的地址固定为 0。与 Activity 的启动方式不同的是 Service 的启动调用的方法有两个，一种是通过 startService()方法直接启动，另一种是通过 bindService()与某个 Activity 的生命周期绑定启动，而这两种启动方式的最大不同也就是通信方式和生命周期。图 3-7 是 Service 两种启动方式的生命周期的切换。

虽然 Service 的启动和 Activity 的启动是相似的，都是 ActivityManager 进行调度管理的，不同的是 Service 多了向 Service Manager 注册这一步骤，但是 Service 不存在任务栈的概念，所以并没有那么复杂的生命周期。因此 Service 的插件化相比 Activity 要简单的多。因为 Service 的生命周期都是通过系统来控制的，因此它的生命周期完全可以通过自身来管理；Service 不存在任务栈的概念，不需要创建多种模式的 Service，只需创建一个

代理 Service，利用代理 Service 对插件中的 Service 进行分发。因此本文 Service 的插件化思路就是代理分发。

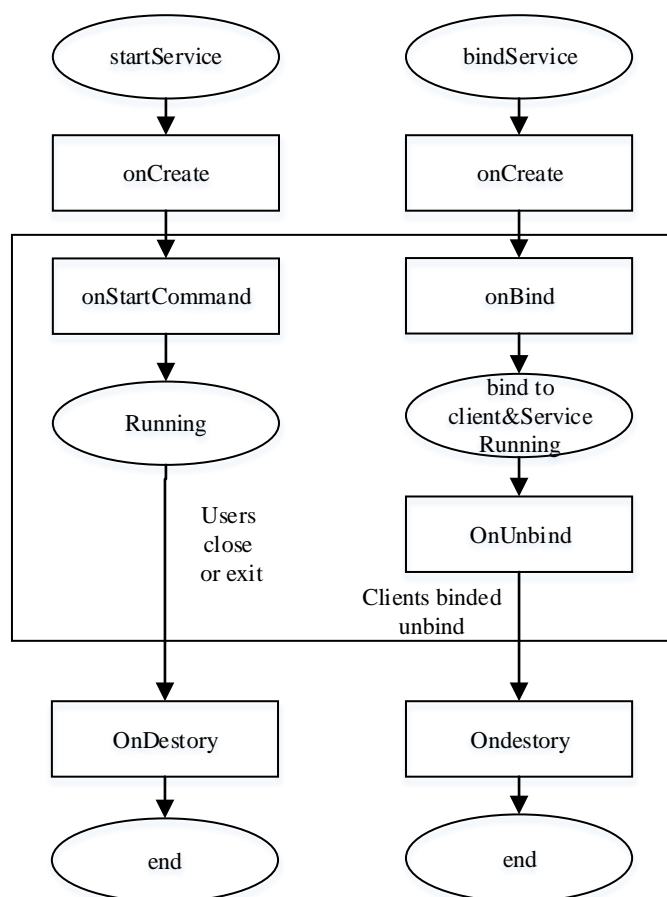


图 3-7 Service 两种启动方式的生命周期

Figure3-7 Service cycle of two startup modes

3.3.2.3 BroadcastReciever

BroadcastReciever 是一种消息型组件，用于不同组件或不同应用之间的消息通信^[33]。BroadcastReciever 工作在系统的内部，所以其并不能被用户感知。BroadcastReciever 的注册方式有两种，一种是静态注册，一种是动态注册。静态注册是在 Manifest 文件中进行注册，通过这种方式进行注册的广播，安装时会被系统自动解析，可以在应用没启动的情况下启动应用并接收相应的广播。动态注册广播是通过代码来进行注册，在不使用的时候也是通过代码来解除相应的广播，这种广播则要求应用必须启动，否则无法接收到对应的广播。因为应用不启动，广播就无法注册，没有进行注册的广播是接收不到其对应的消息的。注册了的广播接收者都会被添加到一个队列之中，在广播被发送后系统会遍历这个队列，广播的接收者对被发送的广播感兴趣就会接收到该广播，通过这种方式 BroadcastReciever 组件实现了低耦合的观察者模式，观察者和被观察者之间可以没有任何的耦合。

BroadcastReciever 的生命周期就是从接收到广播到触发相应的逻辑处理。逻辑处理结束，其生命周期自然就结束。

对于广播的插件化并不像 Activity 与 Service 两大组件一样，它并没有相应的周期，只要想办法让广播的 onRecieve 方法被触发，那么就可以正常启动插件广播。从上面的分析中可以了解插件中的静态广播不可能真的像宿主中的静态广播一样，所以在插件模块中本文采用静态广播转动态广播进行注册的方式。因此对于两种广播的插件化实际上就可以归为一种，都是动态广播的插件化。

3.3.2.4 ContentProvider

ContentProvider 是数据共享型组件，是用来向其它组件或者应用提供共享数据的^[33]。与 Sevice 和 BroadcastReciever 一样，也不能被用户感知。ContentProvider 可以实现应用之间共享大量数据，避免了进程之间传递大量数据时不方便的缺点。它是 Android 系统提供的匿名存储空间共享 Ashmem(Anounymous Shared Memery)方便进程间共享大量数据。

ContentProvider 是由 SeviceManager 来统一管理的。它的生命周期不像 Activity 和 Service 有那么明显的分割线，而是与 BroadcastReciever 一样，在它的逻辑处理结束后，生命周期也结束了。

正因为 ContentProvider 生命周期的单一性，它的插件化会相对简单些。但是 ContentProvider 又具有共享性，在插件化时，就必须进行安装，否则第三方的应用找不到，就失去了 ContentProvider 原本的意义。因此要实现 ContentProvider 组件的插件化，就必须对 ContentProvider 的安装过程进行 HOOK，让第三方的应用能够将插件中的 ContentProvider 启动起来。

3.4 小粒度插件化热修复

3.4.1 机制中热修复的原理分析

市场上 Android 系统版本多，机型也是各种各样。每个市场的政策和审核速度也各不相同。每发布一个版本时，对于开发者都是一种漫长的等待^[49]。iOS 的覆盖速度是比较快的，一般两三天就能达到 80%，但是对于 Android 应用的版本要达到相同的覆盖率却要等到两周以上。这样 Android 版本的更新就被严重阻碍了，导致了市场上应用版本太过于分散，这对于开发人员处理 bug 和投诉就成为十分麻烦的一件事。在开发过程中经常会遇见修复 bug 需要等到下一个版本发布的情况；老版本升级速度太慢，上线后频繁的提醒用户，影响用户体验。这些问题都可以通过热修复技术在用户无感知的情况下加速 bug 的修复和版本的更新来改善。正常的开发流程如图 3-8。

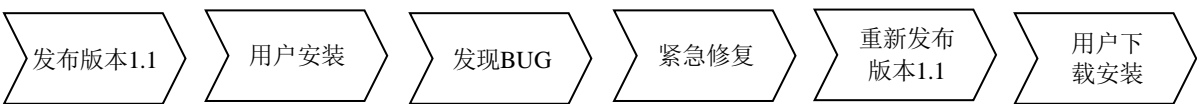


图 3-8 正常应用开发流程

Figure3-8 Normal application development flow

图 3-9 则是热修复的开发流程。



图 3-9 热修复开发流程

Figure3-9 hofix development flow

从上面两种开发的流程可以看出来热修复的开发流程相比正常应用的开发过程有巨大的优势，热修复不需要用户主动下载新的安装包进行安装，可以在用户毫无感知的情况下对 bug 进行修复。

目前开源上也有一些热修复的框架比如说：Dexposed、AndFix 等。Dexposed 不支持 Art 系统，并且写补丁比较困难，粒度也比较细，需要替换的方法又比较多，因此工作量比较大；AndFix 与 Dexposed 相比更简洁一些，打补丁时也不需要重新启动，但缺点是它跳过了类的初始化，对于静态函数、静态成员和构造函数会存在问题，很可能会直接崩掉。因为这些框架的缺点，因此本文将采用另外一种模式。

通过分析 Android 应用加载流程的源码，可以发现最终是利用 `DexPathList.findClass` 方法去查找类的，它是通过遍历 `dexElements` 的这个列表去获取对应的类并加载，而 `dexElements` 列表的初始化是在 `DexPathList` 类的构造函数中完成的。因此一个类如果成功的加载进入内存，那么这个类 `dex` 肯定会在 `dexElements` 中。出现在 `dexElements` 前面的类会被优先加载，所以类在 `dexElements` 中出现的顺序是十分重要的。本文的想法是将需要更新的类放在 `dexElements` 列表的前端，从而达到 bug 修复的作用。

3.4.2 动态代码注入解决 Multidex 失败的问题

需要实现热修复，那么就需要应用程序在运行时动态的更改 `PathClassLoader.PathList.dexElements`。但是因为这些属性是 `private` 属性，需要更改的话，那么就需要用到反射，在构造 `dexElements` 时可以通过构造自定义的 `DexClassLoader` 来加载 `dex` 文件，然后把自定义的 `dexClassLoader.pathList.dexelements` 插入到系统默认的 `dexelements` 前端，通过遍历该数组就可以实现系统优先加载新类，从而实现热修复。以上从理论上已经完成了热修复的方案，但是在实际操作中会报出错误。原因是在 `dex` 文件的优化过程，从 `dex` 文件优化成 `odex` 时，系统会进行校验。系统校验的内容是对直接引用到的类和当前 `class` 是不是在同一个 `dex` 文件中，假如是的话就会被加上 `CLASS_ISPREVERIFIED` 标志^[9]。如果某个类被加上了这个标志，一旦出现了 `Multidex` 那么这种热修复方案就会失败。

为了解决上述问题，最简单有效的方式就是让一个类去引用其他 `dex` 中的类，但是如果仍然使用源码的方式引用，那么是毫无作用的，因为它们始终会在同一个 `dex` 文件中。因此必须采用动态注入的方式，只有在运行时动态的注入相应代码才可以使代码顺利编译成功并且同时可以实现两个互相引用的类放到不同的 `dex` 文件的效果^[29]。JVM 会解释执行 Java 代码编译生成的 `.class` 文件。`.class` 是由字节码构成的，而字节码的解释执行是在

程序运行时进行的,因此本文就通过手工去编写字节码动态的插入至目标文件,通过这种方式来完成一个类对另一个 dex 文件中类的引用。目前这种动态代码注入的工具不少,例如 bcel、asm、javassist。它们都是编辑、创建 Java 字节码和分析的开源类库,利用这些工具可以实现代码的动态注入。

3.5 插件安全机制分析

Android 是为移动设备专门设计的系统,手机上的个人隐私和安全顾虑相对于 PC 是只多不少的,因此对于安全性的要求就会更高些。Android 系统本身就设了许多权限来保证应用的安全。本文所研究的插件化机制,新的插件模块需要在网络上进行传输,因此保证代码的安全就显得更为重要。本文将通过下面两种方式来维护代码和应用的安全。

3.5.1 APK 签名机制

Android APK 签名是对 APK 完整性和对发布机构证书唯一性的一种校验机制^[8]。每个发布的 Android 应用在进行升级都需要使用最初版本应用的签名文件进行签名,如果发布的 APK 包被篡改使用不一致的签名,系统就会拒绝进行升级。同时 Android 系统提供了一个权限机制是基于签名的,这个权限是具有相同签名的应用,可以互相公开各自的功能,因此可以在应用间进行安全的数据共享。Android APK 中的 CERT.RSA 证书与其它的不同,可以在本地机器上自行生成签名文件,而并不需要第三方权威机构发布认证。

将未签名的 APK 与签名的 APK 文件进行对比可以发现,签名的 APK 文件多了 3 个文件: MIFEST.MF, CERT.SF, CERT.RSA。MIFEST.MF 中保存了除了 META.INF 文件以外的 SHA1 和 BASE64 后的编码;CERT.SF 保存了 MANIFEST.MF 文件的 SHA1+base64 编码和 MANIFEST.MF 各子项内容 SHA1+Base64 编码后的值;CERT.RSA 是用来保存通过私钥计算出来的 CERT.SF 文件的数字签名、有效期、所有者、公钥、算法、发证机构等信息。

利用 Android 系统的 APK 签名机制来保护应用的安全,是本文所要研究机制的一个增加应用程序安全性的重要措施,同时本文也将通过加密 dex 文件来保护代码。因为现在 APK 签名工具比较方便,不再需要手动去打包 APK 并签名,所以本文的 APK 的签名打包是直接使用了 APK 签名工具,而对于 dex 文件本文采取的是手动打包的方式。在插件化过程中本文通过验证插件模块的签名文件的一致性与完整性来保障应用的安全。

3.5.2 dex 文件的保护

dex 文件是可以在 Android 虚拟机上直接运行的文件格式。Java 源码通过 Java 编译器编译成 smali 文件,这个过程本身是一种优化。与.class 文件相比,dex 文件的体积小、运行效率高、被编译后可读性弱。smali 到 dex 文件是一个加壳保护的过程。由于 google 设计的这种加壳方式早已被黑客攻破,导致这层加壳过程早已失效。dex 文件的反编译工具有很多并且反编译 dex 文件也是十分简单的事情,一旦 dex 文件反编译成功,应用的源码

就处于暴露当中。攻击者就可以通过源码来分析 App 的设计流程，因此可以很容易对应用实施恶意篡改、恶意注入代码甚至是盗版应用的危险行为。为了避免插件模块被劫持后，导致关键代码和敏感信息的泄露，甚至恶意代码的植入，对 dex 文件应该做出相应的保护措施。否则将对使用这个机制的应用安全产生巨大威胁。因此保护 dex 文件也将是维护应用安全的一个重要措施。

既然 smali 到 dex 文件这个加壳过程被攻破，那么可以在 dex 文件上继续进一步进行保护，对于 dex 文件的保护方式有多种，包括有 dex 文件加壳、dex 文件加花、内存防 dump。

(1) dex 文件的加壳，利用特殊的算法对文件进行压缩加密。

(2) dex 文件加花，由于绝大部分逆向工具都是通过线性的方式来读取字节码再解析的，一旦在读取的过程中遇到无效字节码时就会引起反编译工具字节码解析失败。因此这种方式就是在 dex 文件中插入无效的字节码。

(3) 内存防 dump，对于各种内存 dump，在应用运行时加上内存保护来防止内存被 dump。

上面几种方式都可以实现对 dex 文件的保护。

3.6 本章小结

本章主要是对实现插件化机制的实现思路进行了研究和分析。所实现的插件化机制中的插件化方式按照粒度的大小分为模块化更新和热修复。其中本章针对插件化的位置、加载插件代码、模块化更新、热修复、应用安全进行了详细的研究与分析。利用本章的分析，下一章进行相关的设计。

4 粒度化的安全功能插件化实现机制设计

在第三章中本文将插件机制按照粒度的大小分成了两大部分，分别是模块化更新和热修复。图 4-1 分别是模块化更新的流程图，热修复的流程。

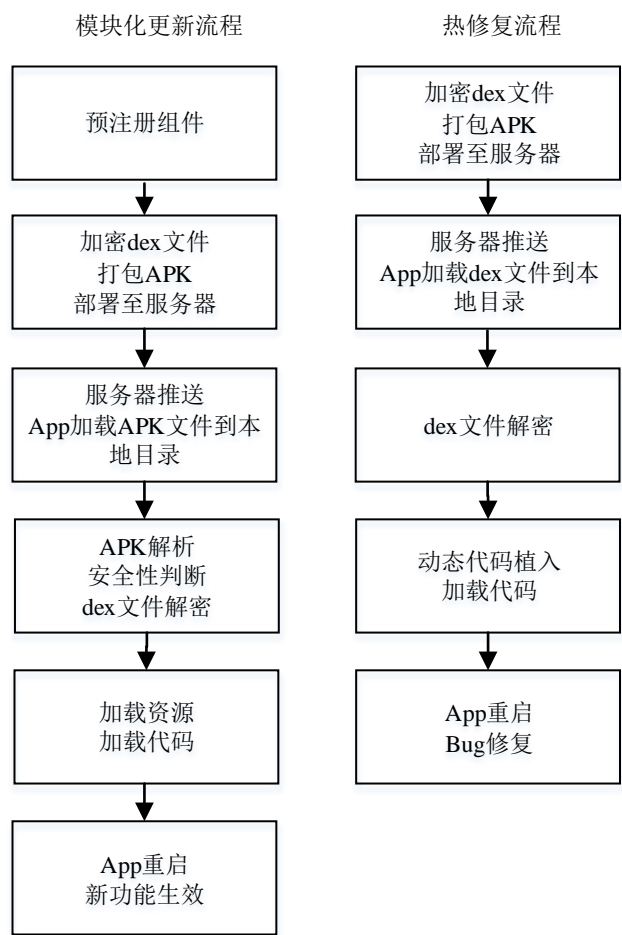


图 4-1 插件化机制流程

Figure4-1 The flow of plugin mechanism

从图中可以看出模块化更新与热修复在很多过程上是相似的，所以在第四章的设计中将它们相似的部分放在一起进行设计，不同的地方再分别进行详细设计。

4.1 总体设计

本文所设计的插件化机制是从大小两个粒度进行插件化的。该机制不仅可以让插件像普通 Android 应用加载进入宿主应用进行模块化更新，还可以对宿主应用中出错的文件进行修复，这样既减少了开发的成本也提高了用户的体验效果。整个插件化机制开发完毕之后可以实现应用的开发与普通 Android 应用一样。对于插件化的开发方式，保证模块代码在网络上的安全性是十分重要的，尤其对于一些敏感性比较高的应用，安全就显的尤为重要。目前开源的插件化框架对安全性这方面并没有做相应的保障。基于上述的这些需求，本文所研究的机制将通过 HOOK 技术接管系统的部分管理机制，实现动态加载 APK 或者是单个 dex 文件，同时让这些插件化模块能够正常运行。四大组件插件化都会预先注册代

理组件，然后利用这个代理组件向系统发送数据请求，接着利用借尸还魂的方式还原成真正要启动的组件。在插件化的过程中利用 HOOK 模块 HOOK 系统各类关键的 API 和 Binder 来获取系统的资源以及管理各插件中组件的生命周期。设计 APK 的解析模块，通过这个模块来解析出 APK 中的关键信息，其中 META.INF 下的签名文件对于判断插件模块的安全性起到了关键性的作用。对于保障应用安全方面，本文采用对比签名文件信息，验证签名是否一致和对 dex 文件加壳的方式来维护。为实现热修复，解决 mutildex 的问题，本文使用了 javassist 动态代码注入技术。图 4-2 是插件化机制的总体设计图。

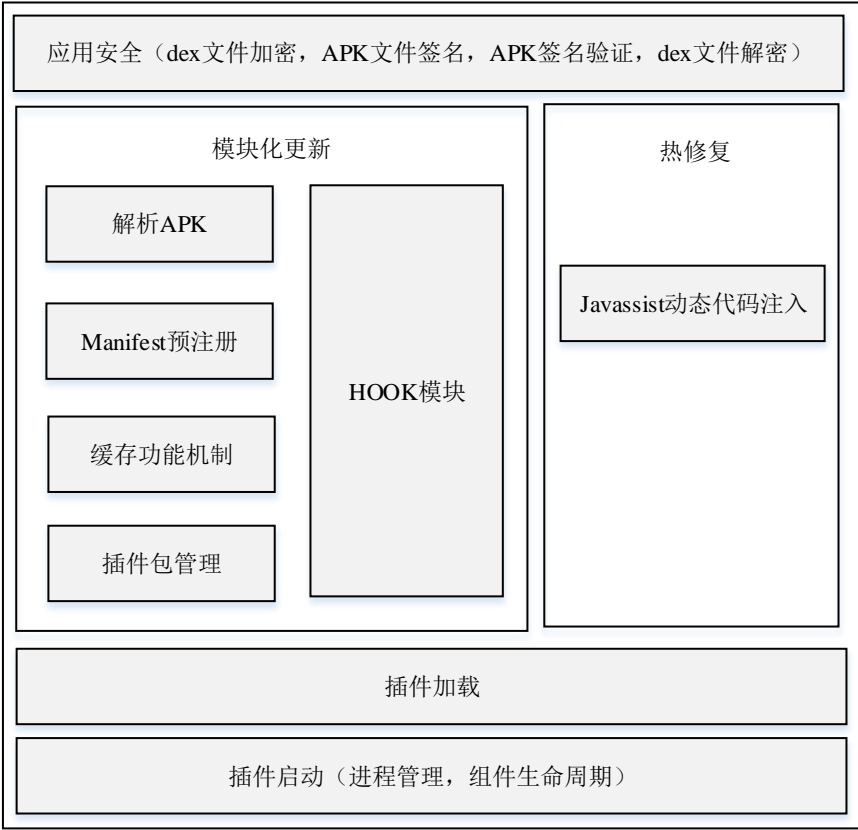


图 4-2 插件化机制总体框架

Figure 4-2 Overall design of the plug-in mechanism

本文研究的机制从插件化粒度大小来划分，分为模块化更新和热修复。模块化更新是增加应用新功能，热修复是从方法级别来修复应用中出错的单个类文件。利用这个框架可以让应用在相对安全的情况下实现模块化更新和热修复功能，并且也可以让系统认为插件模块与宿主应用程序始终为一体。使用该机制开发的应用，App 在用户体验方面将会有质的提升。下面将依次介绍各个模块的设计。

4.2 插件加载模块

在加载插件之前，本文会将从网络上获取的更新文件下载到本地的固定目录下。因此在加载插件时可以直接遍历这个固定目录，若该目录下存在符合条件的 APK 文件或是 dex 文件，系统就会将这些文件加载进入内存。图 4-3 是加载插件模块的流程。

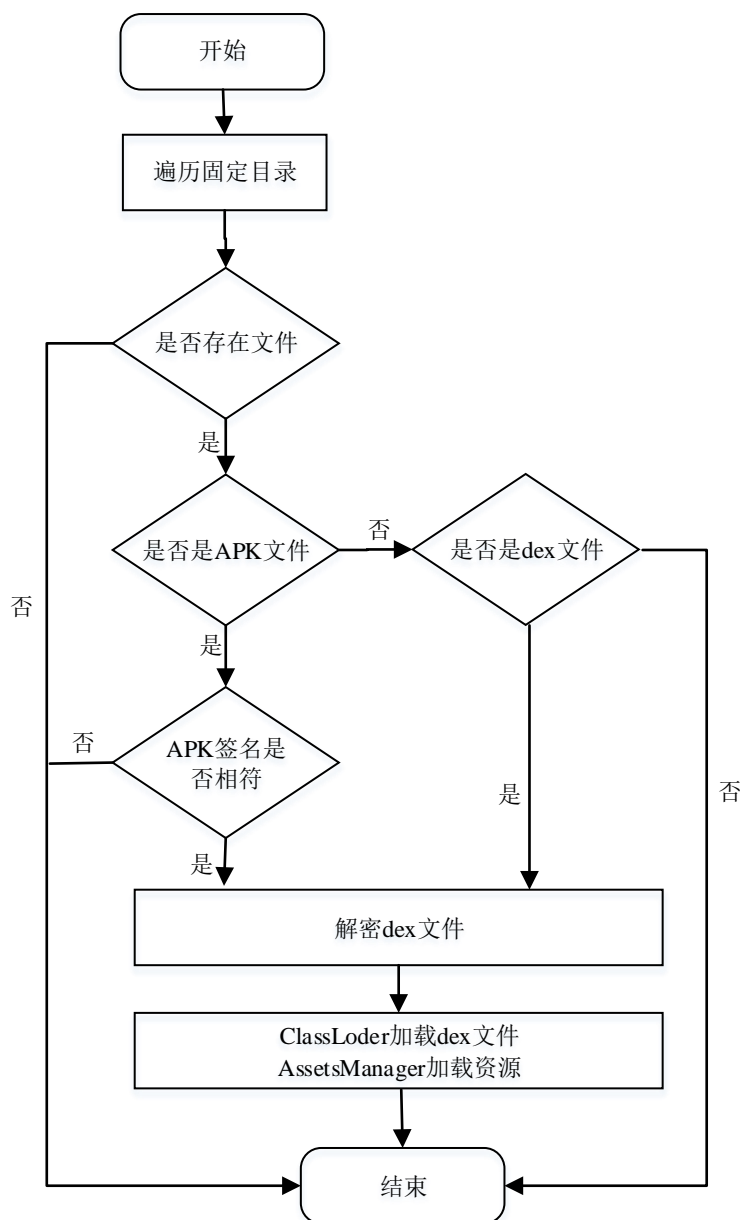


图 4-3 加载插件模块流程

Figure4-3 The flow of loading the plug-in module

4.2.1 资源加载

由于本文所研究的插件化机制中，只有模块化更新才会涉及到资源的加载。模块化更新接入宿主应用所采用的都是 APK 无安装的方式，因此在加载插件中的资源时，不能按照已安装的应用加载资源的方式直接根据 ContextImpl 的 getResources()方法来获取。首先从网络上将插件模块下载到本地的固定目录，然后进行解析。在第三章的插件加载中提到 Resources 对象是通过 AssetManager 来创建的。因此本文通过构造出一个 AssetManager 对象，再利用动态代理反射的方式重写 addAssetsPath 方法将插件的资源和宿主程序的资源路径添加进系统，再利用 AssetManager 对象构造出 Resource 对象，接着利用 Resource 对象通过 getIdentifier()方法获取具体的资源。上面这种方式用来加载的仅仅只是

Drawable 资源,对于 Layout 资源则需要通过反射将当前 Activity 中的 Resource 对象替换为本文插件中构造的 Resource 对象。对与宿主程序与插件程序生成资源 id 的冲突问题,在第三章中也说明了解决方案是通过提供一份 public.xml 的方式来固定资源类型的 id,这样不仅可以解决资源的 id 冲突的问题,也可以实现宿主资源与插件资源的分离。

4.2.2 代码加载

插件模块的 APK 与宿主程序并不属于同一个应用程序,所以正常情况下用宿主应用的加载器是加载不到插件中的代码。在第三章中提到可以利用仿照 Java 类加载机制自定义的加载器来加载插件中的代码。

对于机制中的模块化更新采用的是 APK 文件接入的方式,因为每个模块都是全新的模块,因此本文为每个新模块自定义一个新的 ClassLoader,利用这种方式可以使宿主程序类与插件之间的类具有非常好的隔离性。这种隔离性也正符合了 Android 系统本身的沙箱安全机制的要求,又使应用的健壮性得到了提高。

对于热修复模块是用来修复出错的类,在加载代码时,相对而言会复杂一些,因为除了宿主模块可能出现错误,各插件模块也会出现错误。通过研究类加载的源码可以发现宿主程序的 ClassLoader 都是继承自 BaseDexClassLoader,而 BaseDexClassLoader 中是利用静态类 DexPathList 来查找类,在这个静态类中存在一个数组 dexElements,在这个数组中包含了所有需要加载类的信息,因此查找类的过程就是遍历 dexElements 的数组的过程,于是本文就利用 java 的反射技术将类的信息插入到 dexElements 数组中靠前的位置,根据前面的类的加载机制,应用就会加载本文所修改的类而不会再去加载出错的类了。由于插件和插件之间以及宿主程序间所使用的 ClassLoader 都不一样,所以必须在加载类文件时将相应的 ClassLoader 保存,当该类需要修复时就可以找到对应的 ClassLoader,利用该加载器来加载新类。

通过上述的设计代码已经成功的加载进入了宿主应用。对于热修复模块,因为它的粒度小,针对的是类文件,修复的是已经存在的类,如果修复的是组件类,该组件也是已经可以正常访问系统资源和服务了;但是对于模块化更新,因为该插件模块并没有安装注册,所以对于系统的相关权限或者是资源还不能正常进行访问。因此需要对系统的一些关键模块进行 HOOK,从而欺上瞒下的去访问相应的资源获取相应的权限。本文将在后面 HOOK 模块的设计进行详细的介绍。

4.3 模块化更新

4.3.1 APK 解析

对于普通的应用都是以 APK 包进行安装的,APK 在正常安装的过程中,系统会对 APK 进行自动解析,并且将信息缓存到系统中,在应用请求启动某个组件时,会自动匹配组件信息并初始化组件,启动组件。但是本文中的 APK 并不能通过正常方式进行安装,

因此必须对 Android 系统这一过程进行 HOOK，然后执行本文的逻辑，完成这一过程。在第三章中提到需要对 dex 文件进行保护并且验证 APK 签名，因此可以在该模块中利用解析 APK 文件的过程获取 dex 文件和签名文件，并对签名文件进行验证和对 dex 文件进行解密。

通过上面的分析，APK 包的解析将是本文所研究机制中必须设计的一个模块。当有新的插件模块下载到本地固定目录时，该模块就会自动解析出对应的信息，并且将其放入系统的缓存目录下。通过对应用安装过程源码的分析，可以得出 APK 的解析是由 PackageParser 这个类来负责的，因此本文通过继承该类，实现自己的逻辑来解析插件的 APK 包。

4.3.2 AndroidManifest 预注册

Android 四大组件的启动都需要在 AndroidManifest.xml 中进行注册^[43]。因此想要启动插件中的四大组件势必需要在宿主程序中进行预注册。对于插件模块中四大组件的数量在开发宿主应用时并不能预测出，但是因为系统本身的限制，能同时运行的进程数量是有限的，同时运行的插件数量也就有限，因此本文对四大组件中代理组件的注册数就可以有限化。

为了让每个插件都能够独立的运行在自己的进程中，必须对预注册的组件的 process 属性进行赋值，用来区分不同的进程。因为开发 App 时，在 AndroidManifest 中注册的组件可以通过指定 process 属性来指定该组件所在的进程。对于 Activity 而言它的启动模式有四种，除了由于 Standard 模式每次启动都是新的 Activity，在一个插件中只需为其预先注册一个 Activity，其余模式下启动的 Activity 则必须预注册多个。广播也具有一定的特殊性，它具有两种注册方式，本文采取的方式是静态转动态注册，所以并不需要对其进行预注册。对于其他的两个组件并不像 Activity 的生命周期和启动模式那样复杂，它们只需在每个插件中预注册一个代理组件。

4.3.3 四大组件的插件化

Android 系统中四大组件的启动都需要在 AndroidManifest 中进行注册，由于插件模块中的组件并未在宿主应用的文件中进行注册，所以必须使用上述预先注册的四大组件作为代理，利用代理组件向系统发出相应的请求，在获得相应的应答后再还原成插件中对应的目标组件，在此过程中本文利用 HOOK 技术结合动态代理进行拦截和替换。四大组件拦截替换过程如图 4-4。

因为 Activity 启动模式复杂，在图中的拦截替换过程中，本机制会建立一个映射关系，方便系统在做任务栈管理时能够正确的识别目标组件从而实现不同的启动模式。对于 Service 与 ContentProvider 它们的启动不像 Activity 那样复杂，本文只需将目标组件替换为代理组件，最后再进行代理分发还原。而对于广播，有动态和静态之分，对于静态本文将它转动态进行注册的。所以本文在 APK 进行解析时，将获取的广播信息通过 HOOK 的

方式加入到动态广播的队列当中，然后当广播被发送后，系统会对该队列中对该广播感兴趣的接收者回调相应的方法。

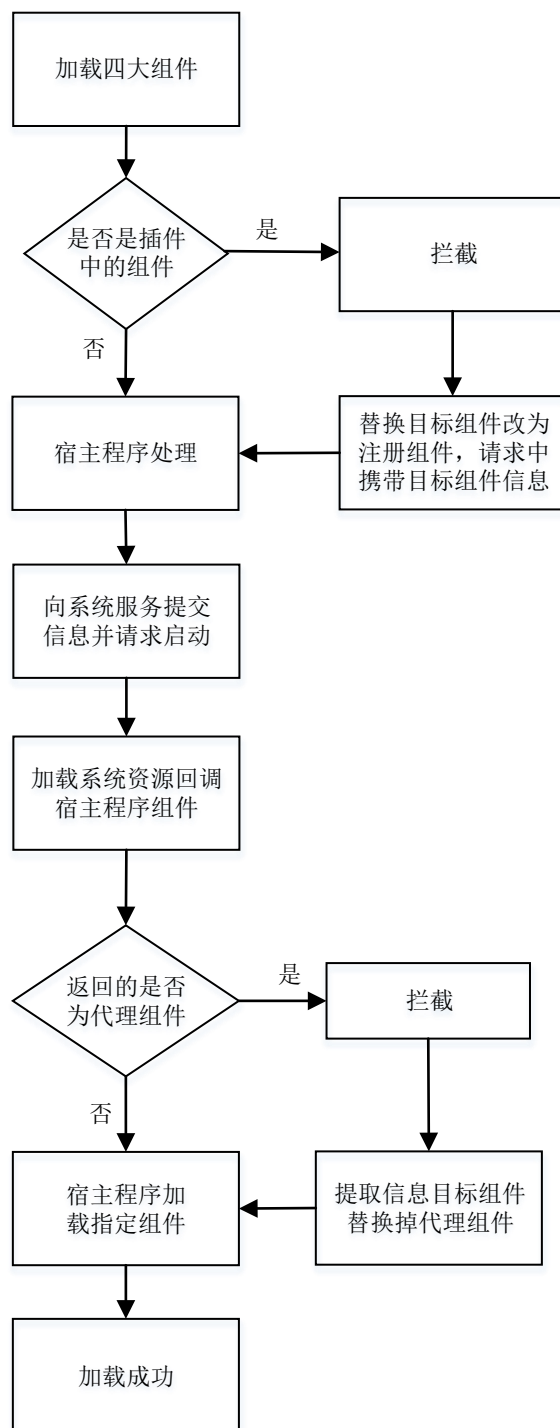


图 4-4 四大组件加载替换流程

Figure4-4 The loading and replacement flow of four components

4.3.4 HOOK 模块

HOOK 模块的设计主要是因为系统在通信过程中插件模块没有进行安装和注册，导致无法正常访问系统资源或是加载组件。本文通过 HOOK 技术和动态代理的技术来解决这个问题，从而保障插件模块的正常运行。这项技术主要分为以下两块：第一块是用于

PMS 和 AMS，应用的启动和运行都与 PMS 和 AMS 相关，尤其是应用与系统进程进行通信，更是与 AMS 息息相关，因此 HOOK PMS 和 AMS 是至关重要的；第二块是 Binder，Android 中进程间通信使用最广泛的一种方式就是 Binder，请求系统服务和获取系统权限等都需要 Binder。因此 HOOK Binder 对与启动插件也是不可或缺的。图 4-5 是 HOOK 模块的设计图。

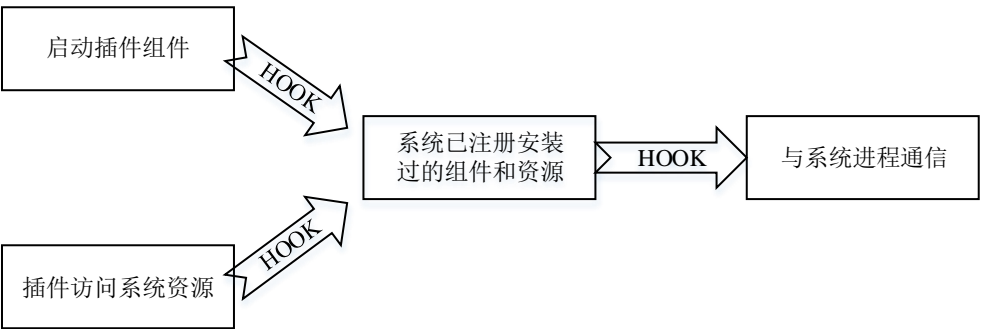


图 4-5 HOOK 模块设计

Figure4-5 HOOK module design

PMS 是 PackageManagerService 的简称（下文简称 PMS）。它在系统中的作用是完成信息校验、APK 信息获取和四大组件信息获取的等重要功能。在 APK 解析、系统启动组件等过程中系统都会检查对应包的信息，因为插件并没有进行安装注册，就会抛出异常。因此要正常启动插件就必须对 PMS 进行 HOOK。从源码中可以看到真正获取 PMS 的方法是 ContextImpl 中的 getPackageManager()。从该方法中可以看到 PMS 对象是在 ActivityThread 中并且还 被 ApplicationPackageManager 包裹了一层。继续查看 ActivityThread 的源码从 ActivityThread 中可以看出 PMS 的代理对象是一个静态对象，可以对这个对象进行 HOOK。因此可以将此作为一个 HOOK 点。

AMS 是 ActivityManagerService 的简称（下文简称 AMS）。AMS 对于 FrameWork 层的作用是非常重要的，四大组件启动运行都和它有着密切的关系。四大组件都是通过调用 AMS 与系统进程进行通信。因此 HOOK AMS 显得尤为关键。通过查看 startActivity()的源码，启动 Activity 最终都是 execStartActivity()方法，这个方法存在于 Instrumentation 类中。从该方法中能够发现其实 ActivityManagerNative 是 AMS 的一个远程代理对象。查看 ActivityManagerNative 中的源码又可以知道 AMS 的这个代理对象是一个单例。因此本文就将此作为一个 HOOK 点对 AMS 的这个代理对象进行 HOOK。

在第二章的介绍中，Binder 可以被理解成在 Client 端的 Binder 本地对象也就是代理对象，也可以被理解成为在 Server 端的 Binder 的真实对象。而在本文研究的机制中主要是插件请求系统服务或资源，是以 Client 端的身份进行 HOOK Binder 的，因此本文所要 HOOK 的对象是 Binder 的代理对象。通过对系统源码的分析可知获取系统的服务需要如下两个步骤：

```
// 获取原始 IBinder 对象
IBinder b = ServiceManager.getService("service_name");
```

```
// 转换为 Service 接口
IXXInterface in = IXXInterface.Stub.asInterface(b);
//asInterface()方法中返回的对象。
android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
```

根据上述代码将 `asInterface()` 方法返回的对象替换成本文 HOOK 后的对象，采用动态代理的方式将 `queryLocalInterface()` HOOK 掉，从而执行本文的逻辑，返回本文伪造的本地 Binder 对象。系统为方便通信，避免每次都需要进行跨进程通信，都会将 Binder 的本地对象缓存在 Map 表中，因此本文将伪造的 Binder 对象加入到这张表中。本文利用的是反射的方式将 Binder 对象加入到表中。采取这种方式修改 Binder 对象，这样 `getService()` 返回的就是伪造的 Binder。

当插件化机制需要访问对应的系统服务、获取求权限、访问系统资源时，都需要用到 HOOK 模块，因此这是插件化的核心模块。

4.4 热修复模块

当应用出现需要修复的 bug 并需要立即更新时，本文会将需要更新的文件打包成 dex 文件，而不再是一个 APK 包。因此热修复文件不需要经过机制中的 APK 解析模块，可以直接进行加载。新类与 bug 类的包名、类名必须一致，同时，新类在数组中的位置又必须优先于应用中的 bug 类，根据类加载机制的双亲委托模型可知，系统就不再会加载 bug 类了。同时为了解决 dex 文件的引用问题，本文在加载 dex 文件的过程中需要进行动态代码的注入。如图 4-6 是热修复的流程。

在 Java 层面通常用到的动态特性是反射，但是反射的效率比较低，并且在 java 层面使用动态代码注册，也不可能实现一个类去引用其它 dex 文件的类。

因此为实现上面的需求，本文使用 `javassist` 类库进行代码注入。利用 `Javassist` 类库中提供的方法来直接在运行时操作 Java 字节码。相比于其他动态注入代码的方式，`javassist` 的性能虽然略低，但是它提供了一层抽象，相比于直接操作字节码的方式 `javassist` 源码级别的 api 成本要低，因此本文才选择使用 `javassist` 来进行代码的注入。在宿主程序开发完毕后，本文的机制要求去打包第二个 dex 文件，这个 dex 文件的目的是让类去引用不同 dex 文件中的类，避免被打上特殊的标志。下面是的 dex 文件的打包过程，如图 4-7。

在宿主应用程序中利用 `javassist` 类库编写相应的代码对需要代码注入的类进行动态注入。

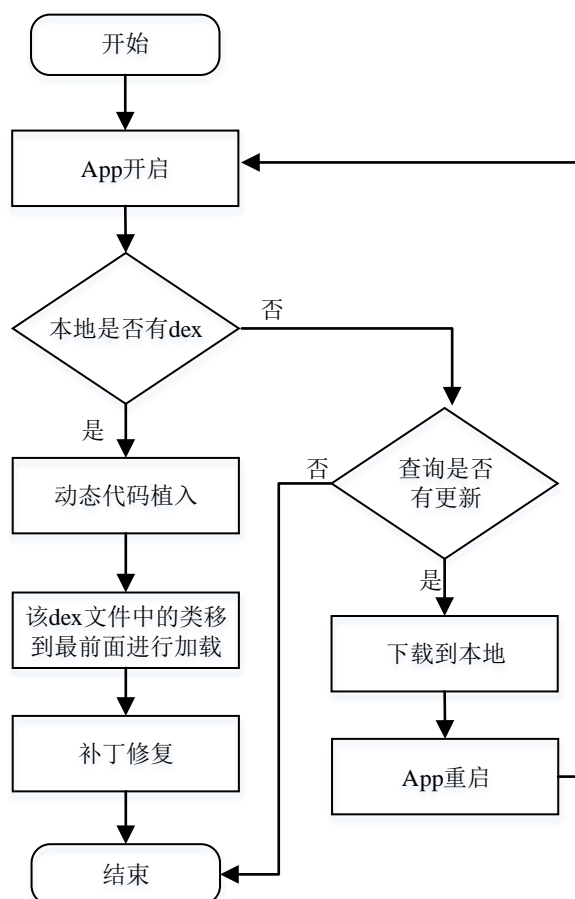


图 4-6 热修复流程

Figure4-6 The flow of hotfix

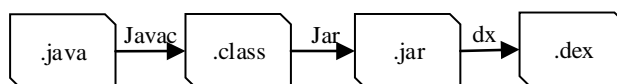


图 4-7 dex 文件的打包

Figure4-7 dex file pack

4.5 缓存机制与插件包管理

Android 应用的沙箱机制，是 Android 系统为了保证应用安全专门设计的^[2]。它的这种安全机制会为每个成功安装的 Android 应用分配一个以应用包名为路径的私有空间，程序中很多本地存储都会选择该路径下的私有空间来进行数据的存储。但是因为插件并没有进行安装注册，所以系统并不会为插件分配这样的空间，而插件本身肯定会遇到需要将数据存储到本地的境况，因此本文通过结合 HOOK 技术与 Java 的反射机制来获取应用程序的 IO 路径，在宿主程序私有空间下为插件创建一个空间。然后重定向到这个 IO 路径。获取到 IO 路径后，插件就可以进行本地存储。

插件在安装完成后，App 根据 IO 路径找到从服务器下载的文件，将文件删除，从而节省更多的内存空间。插件 APK 解析完成后，将解析出来的信息缓存到本地目录，以便应用加载插件，同时也需要将一些关键的对象进行缓存，例如加载插件的 ClassLoder。在插件进行卸载的同时，需要根据插件的包名找到对应的本地目录，将该插件在本地的所有

缓存清空。

4.6 插件启动

热修复是用来修复应用中的 bug 类，所以它的启动是将目标文件进行一个替换，系统会自动的运行该文件的目标代码。但是对于本机制中的 APK 文件，它是一个新的模块，并且没有进行安装与注册，它的启动不是加载到应用内存中就能正常运行的，还需要利用本机制代系统进行相应管理，才能使插件正常启动运行。

4.6.1 进程管理

一个 APK 在 Android 系统中默认就是一个进程，并且这个进程的名就是 AndroidManifest.xml 中的包名，但是系统也允许一个 APK 存在多个进程的情况。可以在 AndroidManifest.xml 注册组件时，指明 process 属性来实现应用内多个进程的情形。因为 Android 系统的沙箱特性正是利用进程的隔离性来实现的，所以通过这种方式可以将宿主程序与插件隔离达到沙箱的效果。本文所研究的插件化机制正是通过启动插件模块中的组件来开启系统的新进程。宿主程序中预先注册了若干个进程，因此就需要制定一种进程启动的方案，本文采取的随机选择的方式。采取上述方式插件就可以在独立的进程中运行，对宿主进程完全不影响，实现了插件崩溃与宿主应用崩溃的隔离。并且因为使用的 HOOK 技术可以使启动插件中四大组件与启动宿主应用中的组件无差别，用户也就无法感知。

本文所研究的插件化机制设计了一个独立的进程管理模块，在启动插件中的组件时，会为该插件模块随机分配预注册进程，该模块也会在进程不可见或者是空进程出现时进行及时回收。因为应用中同时独立运行数个进程的可能性很大，容易导致系统卡顿，影响用户的体验，因此必须使用最优分配法来分配这些进程。进程管理模块可以对运行的进程进行检测，一旦检测到空进程或者不可见进程就对该进程所在的插件的缓存清空并杀死该进程，以腾出更多的空间保证系统的流畅性。

进程的启动会消耗系统的不少资源，对于频繁启动和关闭进程都会影响用户的体验效果，为了更好的对进程进行分配，最大程度的减少系统资源的使用，，本文为进程的管理模块设计了一个进程的启动、匹配和死亡算法。算法流程如图 4-8。

当未启动进程队列为空时，说明本文预注册的进程已经全部启动，现在新的模块需要启动就会选择最早启动的进程杀死并且清空这个进程的所有缓存为新的插件模块重新启动该进程做准备。

当新的插件进程启动后，进程之间需要进行数据的共享。在数据量比较少时，本文直接使用 Bundle 进行数据的传输；在数据量大时，使用共享 id 的方式。

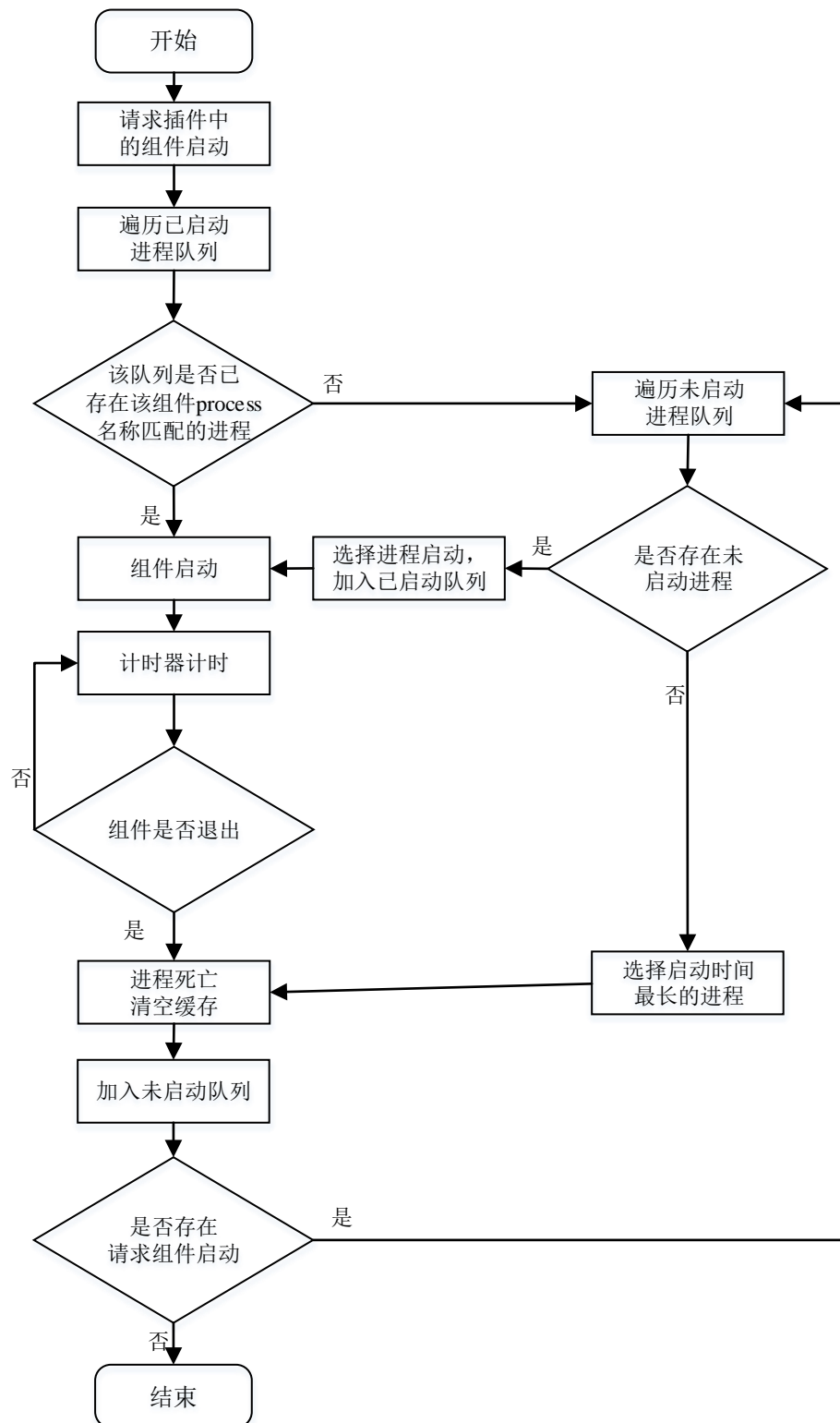


图 4-8 插件进程启动到死亡流程
Figure4-8 The plugin process starts to the death flow

4.6.2 组件生命周期管理

管理组件的生命周期主要针对的时 Activity 和 Service 两大组件，其它两大组件的生命周期仅仅是代码逻辑执行的时间无需进行单独的管理。对于插件中的 Activity 和 Service

组件，可否拥有和宿主中组件一样的生命周期是否十分关键的，这也代表着插件化机制是否成功。

Activity 的生命周期的切换不仅与系统息息相关，而且与用户的操作也是密切相连，这就迫使本文只能将生命周期的管理委托给系统。通过分析 Activity 组件启动源码可以发现生命周期切换的过程中都是通过下面的代码来获取 ActivityClientRecord：

```
ActivityClientRecord r = mActivities.get(token);
```

Instrument 类在获取到 Activity 的 Record 后回调对应的方法来完成生命周期的切换。代码中的 token 其实质是 Binder 对象。token 是 Activity 中的成员变量并且能够唯一的标识 Activity 对象，而 AMS 与应用之间对于 Activity 生命周期的交互正是通过 token 来进行的，并不是通过 Activity 对象直接来实现的。因此本文对这个 Binder 对象进行 HOOK，从而实现插件中 Activity 生命周期的切换。从系统源码来分析，能够看出 AMS 都是使用 token 这个标识对 Activity 任务栈进行处理的，而 Activity 在任务栈中位置的变化正体现了它生命周期的变化。因为需要让插件中的组件启动就必须 HOOK 掉它的本地对象，这就导致 AMS 所在的进程中的 token 是代理 Activity，而应用进程里面的是插件中的 Activity，通过这种方式让插件中的 Activity 具有正常的生命周期，实现 Activity 的插件化。

因为 Service 工作在后台，用户并不能感知到它的存在，所以它的生命周期的切换只是与系统相关。因而相对于 Activity，Service 生命周期的管理没有那么复杂。Service 本身生命周期通常有两种，相对都比较简单。其中非绑定的服务生命周期的开始都是从 startService 的调用，结束则是调用 stopSelf 或者 stopService；绑定的服务则分别对应调用 bindService 和 unbindService。因此对于人为的控制 Service 的生命周期是可行的。对于非绑定启动的对象可以直接利用启动的 Service 对象来回调生命周期交互调用的方法；对于绑定启动的对象，除了像非绑定的方式来回掉对应的方法外，还需要回调 ServiceConnection 对象来完成启动。但是启动的 Service 对象必须是一个真正的具有生命周期的对象，所以利用宿主程序中预注册的代理 Service 组件的各个生命周期回调方法进行代理分发。通过这种方式插件中的组件就具有了和宿主应用中一样的生命周期。

4.7 插件化机制安全模块

本文采用验证 APK 签名文件和对 dex 文件在打包时进行加密、加载前进行解密这两种方式维护应用的安全。

4.7.1 APK 签名

目前大部分的开发都是使用 jarsign 与 signapk 两种工具对 APK 进行签名。两种工具的签名算法都是一致的，它们的区别在于前者是 java 自带的签名工具可以 jar 包签名，后者专门为 APK 签名而开发的；并且两者使用的签名文件不同，jarsign 使用的是 keystore，而 signapk 使用的是 pk8、x509.pem。因为 jarsign 本身已经包含在开发工具中，不用在打包 APK 后进行手动签名，所以本文选择使用它对 APK 进行签名。生成签名文件的步骤

如下：

(1) 利用开发工具生成对应的 keystore 文件。开发工具利用数字签名算法 RSA 生成对应的私钥和公钥保存在该文件中。

(2) 提取 keystore 中的密钥信息，利用 java 的 KeyStore 类 getEntry() 或者 getKey() 获取密钥信息。

(3) 生成 MANIFEST.MF 文件。签名工具遍历 APK 中所有的非文件夹和非签名文件利用一种安全的哈希算法 SHA1 把文件内容按照散列算法生成摘要信息，接着再用 Base64 位进行编码。

(4) 生成 CERT.SF 文件。对前面生成的文件使用 SHA-RSA 算法进行私钥签名。RSA 是一种非对称的加密算法，利用它对消息摘要进行加密。解密则使用公钥。

(5) 生成 CERT.RSA 文件。该文件中保存公钥信息和所采用的加密算法。

(6) 将生成的三个签名文件与源 APK 重新打包生成新的 APK。

4.7.2 验证 APK 签名

前一节介绍了 APK 签名的过程，在 APK 中生成了三个对应的签名文件，分别是 MIFEST.MF，CERT.SF，CERT.RSA。在 Android 系统应用的升级过程中，往往利用这几个签名文件来检验应用的安全性，本文利用这一特点来验证插件模块的安全性。图 4-9 是验证签名一致性和完整性的过程。

JarVerifier.VerifierEntry.verify 对 APK 文件中包含的所有文件对应的摘要值与 MIFEST.MF 的值进行比较，验证 APK 中的文件是否被修改；JarVeirifer.verifyCertificate 使用 RSA 证书文件检验签名文件.SF 文件是否被修改过的；JarVeirifer.verifyCertificate 中使用签名文件 CERT.SF，检验 MANIFEST.MF 文件中的内容是否被篡改过。正是从这上面几方面来确保 APK 的唯一性和完整性。

如果改变了 APK 包中的任何一个文件，在模块 APK 进行解析验证时，改变之后的摘要信息与 MANIFEST.MF 的检验信息不一致，那么本文就拒绝加载该模块；如果更进一步，将更改后的文件计算出对应的新摘要值，接着再去更改 MANIFEST.MF 文件里面对应的属性值，但是这样会导致与 CERT.SF 文件中算出的摘要值不一致，同样拒绝加载；继续更进一步，计算处 MANIFEST.MF 的摘要值，更改 CERT.SF 中对应的值，此时数字签名的值就会与 CERT.RSA 文件的值不一致，也会拒绝加载；而数字签名是不能进行伪造的，因为获取不到对应数字证书的私钥。

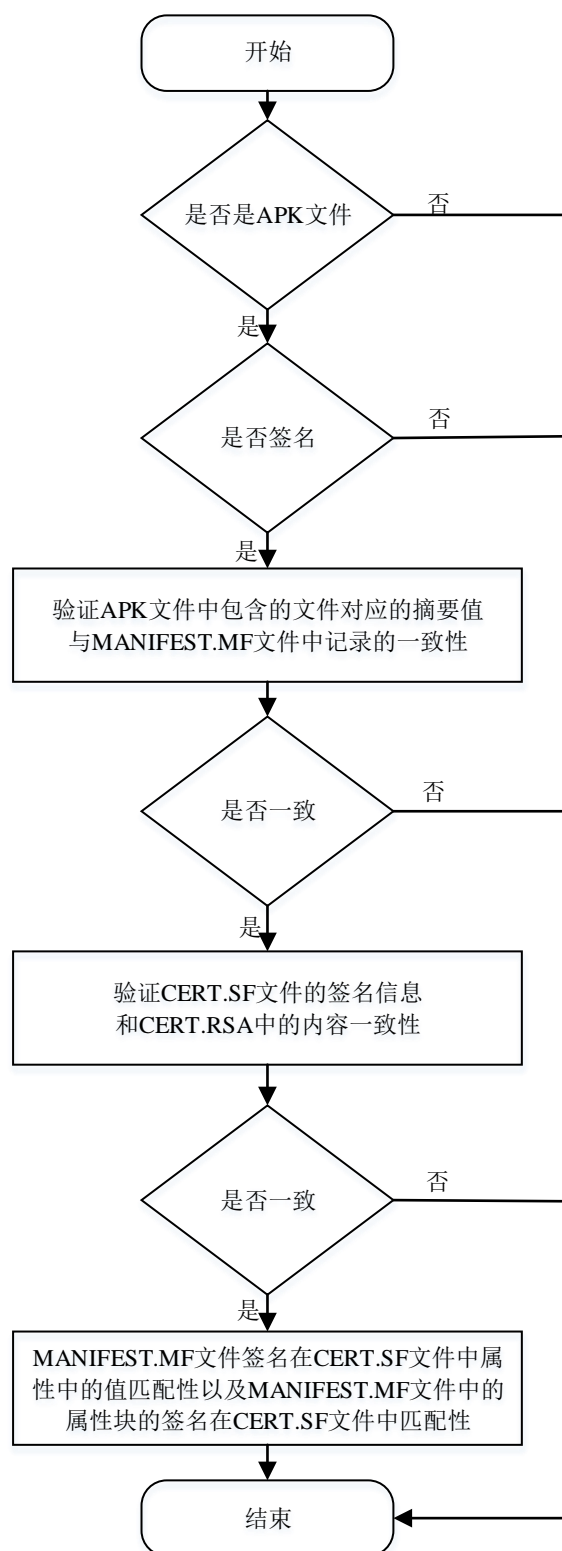


图 4-9 验证 APK 签名
Figure4-9 Verify APK signature

4.7.3 DES 算法加解密 dex 文件

在第三章中介绍了多种 dex 文件的保护方法，最终本文选择加壳的方式。对 dex 文件的加壳可以片面的理解成对 dex 文件进行加密操作。通过加壳程序和算法对原始的文件进

行加密处理，在加载 dex 文件时将 dex 文件进行解密。为了保护原始数据，使原来的数据失去原有的结构和特性就必须在加密过程中使用相应的算法进行处理。加壳原本是将加密后的数据存放在解壳程序的尾部，或者是文件的头部，并且同时在解壳程序末尾添加加壳后数据的相关信息。因为本文针对的是插件模块，为了方便，直接手动对 dex 文件利用加密程序加密，再在本机制中对 dex 文件解密，并没有将解壳程序与数据置于一体。

加壳最重要的就是对 dex 文件加密，因此加密算法的选择显得尤为重要。也正因为如此对加密算法安全性的要求就相对较高。DES 除了用穷举的方法来攻击之外，还没有特别有效的方法，并且它的加解密速度相对其他算法比较快。

DES 是一个分组的加密算法，典型的它以 64 位为分组对数据进行加密，并且加解密为同一套算法。密钥的长度为 64 位，但事实上是 56 位参与运算，密文组的形成是利用分组后的明文组和 56 位的密钥按位替代或交换的方法而得到的。

DES 加密 dex 文件第一步是将 64 位明文即 dex 文件进行初始置换，输出 L_0 和 R_0 两个部分，各 32 位；第二步是生成密钥，将输入的 64 位密钥根据缩小选择换位 1，忽略每行的最后一位得到 56 位，再将这 56 位分为两个 28 位 C_0 和 D_0 ，分别循环左移得到 C_1 和 D_2 ，左移的位数是根据左移表来确定的，然后将得到的结果合并经过缩小选择换位 2 得到密钥 K_0 ，依此类推就可以得到后面的 15 个密钥；第三步是将置换后的左右 32 位数进行 16 次迭代，其迭代公式如上图，因为密钥是 48 位的，所以还需要将 32 位的数据扩展成 48 位其中扩展算法是 $f(R_i, K_i)$ ， f 算法是由 8 个 S 盒将 4 位变成 6 位；第四步就是将得到的 16 次结果根据逆初始值换表进行逆初始置换得到最后的密文也就是加密后的 dex 文件。图 4-10 是 DES 加密 dex 文件的流程图。DES 的解密过程与加密过程是类似的，只是密钥的次序发生了变化，使用的次序相反。

从 DES 加解密的过程中，可以看出 DES 算法的复杂性比较高，它的保密性也不依赖于算法，而是依赖于密钥的保密。虽然 DES 密钥的长度只有 56 位，但是破解起来还是有一定难度，况且对于密钥也可以通过经常更新的方式来保证安全。本文设计的 DES 密钥的更新方式是在每次应用更新时，重新生成密钥，在加密 dex 文件的同时将新密钥写入文件末尾；在终端加载新模块时，首先读取 dex 文件中的密钥，同时删除 dex 文件中的密钥，再利用密钥解密 dex 文件。由于利用密钥更新的方式来保障密钥的安全性，同时又因为相对于其它的加密算法，DES 加解密的速度相对较快，对应用的启动速度影响不大，本文选择了 DES 算法加密 dex 文件。图 4-11 是 DES 密钥的更新过程。

dex 文件不能直接查看，通常都是利用反编译工具反编译后查看 smali 文件。对 dex 文件进行了加密，在未解密的情况下，反编译时出现异常而导致失败，因此在网络中截取到该模块也很难获取到密钥解密 dex 文件。

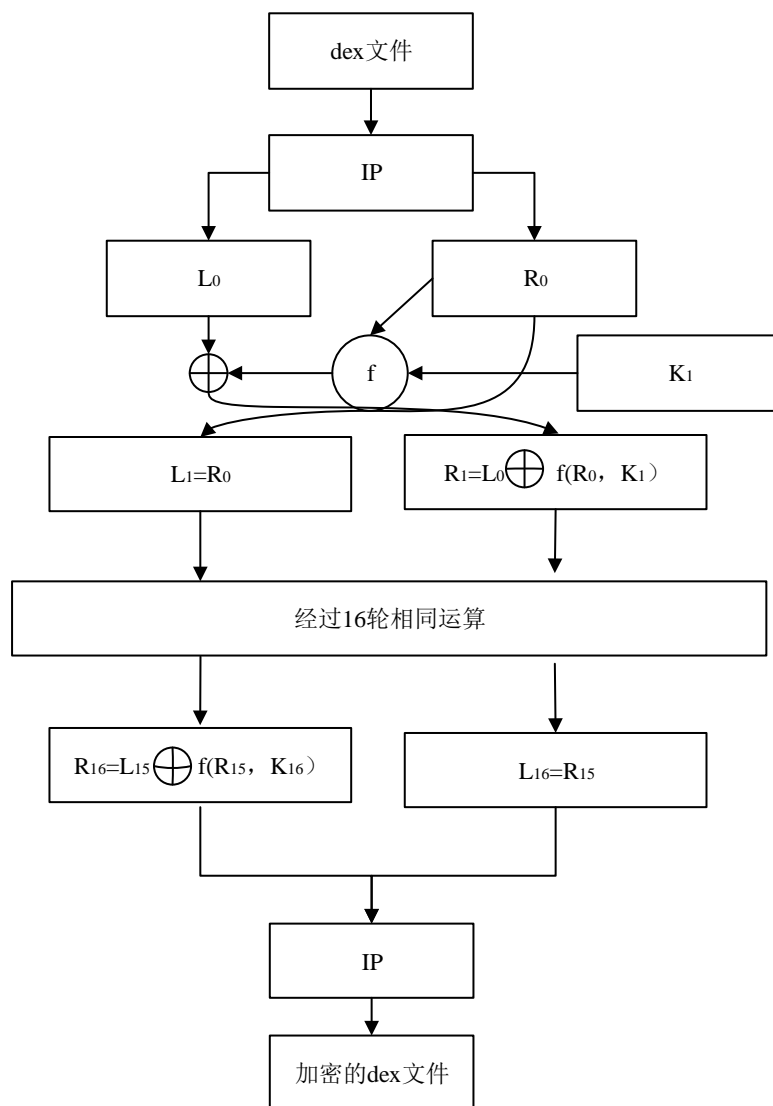


图 4-10 DES 加密的流程

Figure4-10 The flow of DES encryption



图 4-11 密钥更新流程

Figure4-11 The flow of updating key

4.8 本章小结

本章主要是对插件化机制中各模块进行了详细设计。首先是对插件机制整体上进行了介绍，接着对插件化机制的各模块分别进行讲解。本章主要是从以 APK 文件形式来加载新模块，以 dex 文件形式来进行热修复，插件的加载，插件的启动，插件安全机制这几个方面来进行详细的设计。

5 功能插件化机制的实现与验证

5.1 AndroidManifest 预注册的实现

通过上一章的设计和分析，Android 四大组件都需要预先在宿主程序中的进行注册，并且四大组件是按照不同的进程名进行注册，每四种不同的组件的进程名是一致的。对于 Activity 而言它的启动模式有四种，分别是：SingleTop、SingleInstance、SingleTask、Standard。Standard 与其它模式的不同在于，它每次启动都会重新启动一个新的 Activity。鉴于这种原因，本文对于 Standard 模式启动的 Activity 只为其预注册一个，对于其它三种方式启动的 Activity 将各为其预先注册 5 个，编号从 A-E。考虑到可能有数个插件同时运行，所以本文将最先预注册 6 个进程，进程编号从 00-05。下面是按照上述要求进行预注册的代码：

```
<!-->按照这种方式再注册 01-05 进程<!--!>
<activity
<!-->按照这种方式注册 B-E 的 SingleTask<!--!>
    android:name = "com.plugin.mydynamic.ActivityStub$P00$SingleTaskA"
<!-->按照这种方式注册 SingleTop 和 SingleInstance<!--!>
    android:launchMode = "singleTask"
android:process = "com.plugin.mydynamic:plugin00">
    <intent-filter>
        <action android:name = "android.intent.action.MAIN" />
        <category android:name = "com.plugin.mydynamic.PROXY_STUB" />
    </intent-filter>
    <meta-data
        android:name = "com.plugin.mydynamic.ACTIVITY_STUB"
        android:value = "0" />
</activity>
```

就 Service 而言，它的启动方式相对而言比较简单，只有系统可以控制它的生命周期。ContentProvider，它更不具有生命周期。对于这两大组件只需在每个进程中预先注册一个代理组件，下面是 Service 和 ContentProvider 的预注册代码：

```
<!-->按照这种方式再注册 01-05 进程<!--!>
<service
    android:name = "com.plugin.mydynamic.ServiceStub$P00"
android:process = "com.plugin.mydynamic:plugin00">
    <intent-filter>
        <action android:name = "android.intent.action.MAIN" />
        <category android:name = "com.plugin.mydynamic.PROXY_STUB" />
```



```

        </intent-filter>
    </service>
    <provider
        android:name = "com.plugin.mydynamic.ContentProviderStub$P00"
        android:process = "com.plugin.mydynamic:plugin00"/>

```

对于广播而言本文采取的是静态转动态注册，通过 APK 的解析获取到静态注册的广播信息，再根据这些信息进行注册。其相关注册代码如下：

```

try {
    List<IntentFilter> filters=
    PluginManager.getInstance().getReceiverIntentFilter(information);
    for (IntentFilter myfilter : filters) {
        BroadcastReceiver broadcastreceiver=(BroadcastReceiver)
        clazz.loadClass(information.name).newInstance();
        context.registerReceiver(broadcastreceiver, myfilter );
    }
} catch (Exception e) {

}

```

5.2 APK 解析的实现

因为插件 APK 没有进行安装，所以就需要实现解析 APK 模块，解析出每个组件的信息并进行缓存。通过查看和研究系统源码发现系统中有一个 `PackageParser` 类，系统正是使用了这个类来解析 APK 文件的。因此本文采用继承该类，并对该类的方法进行重写的方式解析 APK。下面是实现的一些关键代码：

```

packageParserClass=Class.forName("android.content.pm.PackageParser");
permissionClass=Class.forName("android.content.pm.PackageParser$Permission");
permissionGroupClass=
Class.forName("android.content.pm.PackageParser$PermissionGroup");
instrumentationClass=Class.forName("android.content.pm.PackageParser$Instrumentation");
activityClass=Class.forName("android.content.pm.PackageParser$Activity");
serviceClass=Class.forName("android.content.pm.PackageParser$Service");
providerClass=Class.forName("android.content.pm.PackageParser$Provider");
try {
    sArraySetClass=Class.forName("android.util.ArraySet");
} catch (ClassNotFoundException e) {

```

```
}
```

上述的代码主要是通过反射的方式来获取对应解析类的对象,下面代码利用这些类对象解析相应的组件信息。

```
Method method = MethodUtils.getAccessibleMethod(packageParserClass,
"generateActivityInfo", activityClass, int.class, packageUserStateClass, int.class);
return (ActivityInfo) method.invoke(null, activity, flags, mDefaultPackageUserState, mUserId);
```

上述代码是解析 Activity 组件的代码,对于其他组件的解析只要将方法中的参数做相应的改变。

5.3 四大组件启动的实现

四大组件在启动前,本文会根据宿主程序中预注册的多个进程和进程的匹配算法选择最合适的进程启动。

5.3.1 ClassLoader 加载代码

系统本不会主动的加载插件中的代码,因此本文实现了自定义的加载器 PluginClassLoader,这个加载器继承自 BaseDexClassLoader,并且重写了 loadClass()方法。同时定义了 PluginManager,在这个类中定义了 installAPK()方法。

(1) loadClass()方法是用于从当前 ClassLoader 中查找类是否加载,没有加载则从 List<PluginClassLoader>中遍历查找。

(2) installAPK()方法用于加载插件代码,为每个插件新建 PluginClassLoader 加载器加载代码,实现插件之间代码的隔离性。

通过这种方式实现了插件代码加载到宿主应用内存种,并且很好的实现了代码的隔离性。在加载代码同时将各个组件的对象缓存在内存中,方便后面在启动插件时寻找各组件的对象。

5.3.2 Activity 的启动

首先本文先将 AMS 的本地代理中 startActivity 进行 HOOK,并替换掉 AMS 的 intent 对象,将插件中的目标 Activity 替换成本文预注册的 Activity。关键代码如下:

```
ComponentName componentName = new ComponentName(targetPackage,
StubActivity.class.getCanonicalName());
newIntent.setComponent(componentName);
```

通过这种方式绕过系统的检查并利用伪造的 Binder 对象与系统服务进行通信,系统中的 AMS 开始真正执行 startActivity 方法,应用中要正常启动还需要将代理的 Activity 替换回目标 Activity 且必须再进行一次跨进程通信,从系统进程再回到应用的主进程,系统回到应用的进程同样是通过 Binder 实现的。在系统进程中是利用 Handler 发消息到应用进程中,而 Handler 的实质也是 Binder。然后再在此 Handler 中实现拦截将代理 Activity

替换回目标 Activity。因为在启动插件时，本文会将这个 Activity 对象进行保存，因此在替换的过程中很容易就可以找到对应的对象进行替换。关键代码如下：

```
Intent target =  
raw.getParcelableExtra(HookHelper.EXTRA_TARGET_INTENT);  
raw.setComponent(target.getComponent());
```

5.3.3 Service 的启动

与 Activity 启动的实现一样，需要将 AMS 的本地代理中的 startService 和 stopService 方法进行 HOOK，对于绑定服务则是 HOOK onBind 和 onUnbind 两个方法。在 HOOK startService 和 onBind 方法时，需要把目标 Service 替换为代理 Service。关键代码如下：

```
ComponentName componentName = new ComponentName(stubPackage,  
ProxyService.class.getName());  
newIntent.setComponent(componentName);
```

在 stopService 与 onUnbind 方法中利用类似的方式进行相同的过程。通过这种方式，本文已经将插件目标 Service 重定位到代理 Service 上，因此在启动 Service 时，不再是目标 Service 收到 onStart 或者是 onServiceConnected 的回调，而是代理 Service 相应的方法，接着本文采取代理分发技术，在这个回调中将任务交回给目标 Service。在前面本文会将各个组件对象共享一个缓存。因此在回调中很容易获取目标对象，替换回目标对象执行对应的逻辑。关键代码如下：

```
for (ComponentName componentName : mServiceInfoMap.keySet()) {  
    if (componentName.equals(pluginIntent.getComponent())) {  
        return mServiceInfoMap.get(componentName);  
    }  
}
```

对于 Service 的各个生命周期的回调本文采取的都是上述的代理分发方式。

5.3.4 BroadcastReceiver 的启动

对于静态广播的启动，本文已经在 AndroidManifest.xml 进行注册了，并且经过 APK 解析模块已经获取了静态广播的相关信息，然后将静态广播转为动态注册。因为广播并没有生命周期，因此不必像 Activity 和 Service 一样进行插件化，只需要将代码加载进入内存，创建一个 java 对象，通过某种方式让对象回调 onReceive 方法。对于静态广播插件化的实现，在 APK 解析时本文已经获取到了静态广播信息，还需要 HOOK AMS，利用 AMS 将本文获取到的广播信息注册到系统中，让插件中的静态广播顺利接收到系统广播。下面是关键代码：

```
BroadcastReceiver broadcastReceiver =  
(BroadcastReceiver)clazz.loadClass(activityInformation.name).newInstance();
```

```
context.registerReceiver(broadcastReceiver, intentFilter);
```

至此，静态广播已经转化为动态广播注册进入到系统中，将这些广播通过 Java 的动态代理，将插件中的广播加入到广播接收者的队列中，然后在广播被发送时，系统自然会去遍历这个队列，对其中订阅了该广播的接收者回调 `onReceive()` 方法。从而完成了广播的插件化。

5.3.5 ContentProvider 的启动

对于插件模块中 `ContentProvider` 的启动，相对于广播的启动会复杂，因为该组件需要分享到第三方应用。本文是通过预先安装宿主中的插件中的 `ContentProvider`，在第三方应用请求时，它所请求的会是代理 `ContentProvider`。只是预先安装的过程需要把握时机，否则若是代理组件启动调用插件组件时，插件组件还并未安装完成，那么应用肯定会报错，所以本文选择在 `Application` 类的 `attachBaseContext` 中完成插件组件的安装，因为在前面 `APK` 解析中已经获取到了对应插件的信息，然后就可以通过 `HOOK AMS` 来实现组件的安装。

第三方应用在封装请求时会封装对应插件的信息，然后通过拦截解析请求，利用代理组件来进行分发，由具体的插件组件来完成相应的响应，这个过程与 `Service` 启动过程的代理分发技术是一致的。关键代码如下：

```
String rawAuth = raw.getAuthority();
String uriString = raw.toString();
uriString = uriString.replaceAll(rawAuth + '/', "");
Uri newUri = Uri.parse(uriString);
```

5.4 热修复的实现

热修复中最重要的一步就是动态代码的注入。动态代码注入的实现如下：

(1) 打包一个 `hack_dex.dex` 文件，这个文件中包含一个类文件，这个 `dex` 文件的作用就是让宿主或者插件 `dex` 文件引用，避免被打上 `CLASS_ISPREVERIFIED` 的标记。

(2) 编写 `InjectHack` 类，这个类的作用是用来将代码注入到需要修复的类中，不过在编写时注意导入 `javassist-*.jar` 的包。

(3) 在 `Application` 中判断是否是热更新，如果是则调用 `InjectHack` 类进行代码注入，如果不是则执行正常的程序逻辑。

通过上述方式，代码已经注入到类中，实现热修复只需要将更新的类加载进入内存中，只要该类优先于需要修复的类加载就可以成功。不过在实现类的加载过程中，本文需要选择正确的加载器来加载这个更新类。本文在加载各个插件时，把每个加载器的对象以键值对的方式保存起来，键值就是各个插件的包名。然后，本文再通过 `dex` 文件获取到类名，根据类名获取到包名，通过这个包名去获取到加载器加载更新文件。

在加载器加载类之前需要将这个更新类的信息插入到 `dexElements` 数组前面，其关键

代码如下：

```
// 创建一个数组，用来替换原始的数组
Object[] newElements = (Object[]) Array.newInstance(elementClass, dexElements.length + 1);
Constructor<?> constructor = elementClass.getConstructor(File.class, boolean.class, File.class,
DexFile.class);
Object o = constructor.newInstance(apkFile, false, apkFile,
DexFile.loadDex(apkFile.getCanonicalPath(), optDexFile.getAbsolutePath(), 0));
Object[] toAddElementArray = new Object[] { o };
// 复制原始的 elements
System.arraycopy(dexElements, 0, newElements, 0, dexElements.length);
// 复制插件 element
System.arraycopy(toAddElementArray, 0, newElements, dexElements.length,
toAddElementArray.length);
// 替换
dexElementArray.set(pathListObj, newElements);
```

5.5 插件化安全机制的实现

插件化机制安全的保证是通过验证签名文件的一致性、完整性与 dex 文件的加密来实现的。通过 APK 解析，本文可以获取到 APK 签名的信息，然后验证签名的一致性与完整性。关键代码如下：

```
//获取签名信息
packageParser.collectCertificates(pkg,0);
//访问签名信息
pkg.signatures = new Signature[myCerts.length];
    for (int i=0; i<count; i++){
        pkg.signatures[i] = new Signature(
            myCerts[i].getEncoded());
    }
//对比签名信息
private boolean IstheSame(Signature[] first, Signature[] second) {};
```

根据验证 APK 中文件是否被篡改的结果来确定是否进行 dex 文件的解密，若 APK 文件已经被修改，则不进行下一步的 dex 文件解密，放弃下面的步骤，因为该插件模块已经不安全；若 APK 文件未修改，则进行 dex 文件解密。dex 文件加解密的关键代码如下：

```
//DES 解密
public static byte[] decryptDES(byte[] source, byte[] key) throws Exception {
```

```

// 创建随机数源
SecureRandom secureRandom = new SecureRandom();
DESKeySpec desKeySpec = new DESKeySpec(key);
// 生成密钥工厂，再把 DESKeySpec 对象转换成 SecretKey 对象
SecretKeyFactory kF = SecretKeyFactory.getInstance(DES);
SecretKey sK = kF .generateSecret(desKeySpec );
Cipher myCipher = Cipher.getInstance(DES);
// 密钥初始化 Cipher 对象
myCipher.init(Cipher.DECRYPT_MODE, sK , secureRandom );
// 进行解密
return myCipher.doFinal(source);
}

```

DES 的解密工作是在加载器加载类之前进行的，DES 的加密则是在打包 dex 文件时进行的。DES 的加解密过程是类似的，只是密钥的顺序相反，所以在上面代码的实现上只是截取了 DES 解密的关键代码。

5.6 插件化机制的验证与对比

5.6.1 机制验证

在前面本文对插件化机制进行了详细的设计与实现，本节通过演示示例程序来测试本文所研究的插件机制。本文将对插件的安装，卸载，各组件插件化的结果进行演示。在演示的过程中插件放在固定目录下进行加载。在插件化模块中演示的是各大组件的插件化，在宿主程序中有一个测试热修复的入口，用来测试热修复模块。

(1) 插件加载



图 5-1 安装页面

Figure5-1 Installation page

通过点击上面的安装按钮，测试安装组件模块，安装成功后跳至安装页面，如图 5-2。



图 5-2 安装成功页面

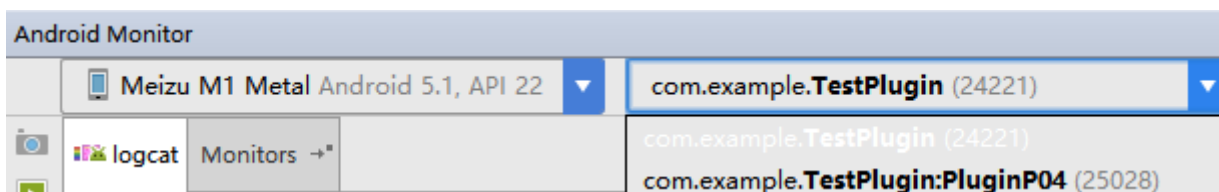
Figure5-2 Installation Success Page

成功跳转至安装成功页面，说明插件模块已经成功加载进入了宿主应用内存，接下来通过点击打开按钮启动插件。

(2) 进程启动



5-3(a) 插件主页



5-3(b) 进程启动

图 5-3 插件启动

Figure5-3 Plug-in starts

从图 5-3 中的图 b 中可以看到插件启动起来，一个新的进程也启动了。在插件的主页面有四大组件测试的入口，分别点击四个按钮进入各个组件的插件测试。

(3) 组件插件化

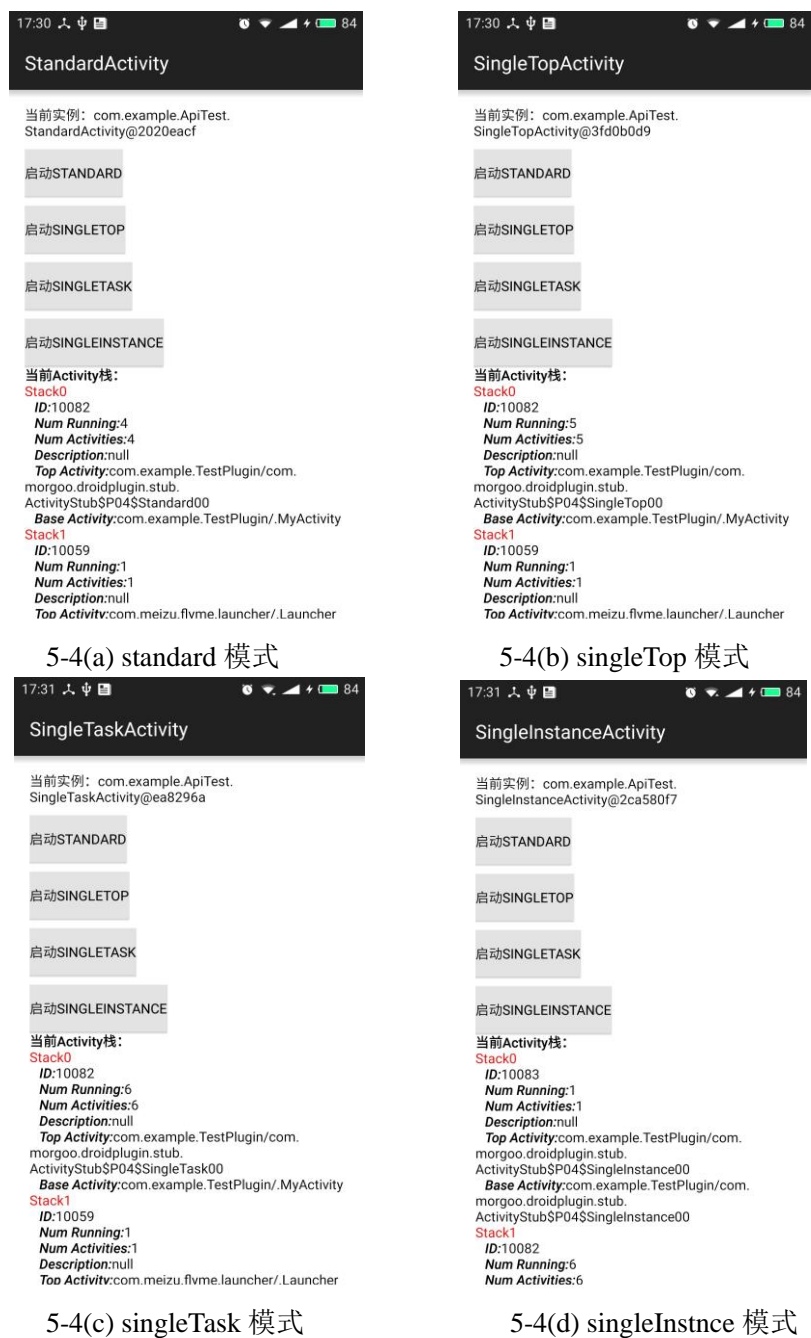


图 5-4 Activity 插件化
Figure5-4 Activity plugin

从图 5-4 中可以看到不同模式下各个 Activity 插件化结果。从各个模式启动的 Activity 打印出的任务栈的信息，可以看到栈中的 Activity 的信息都是相应代理组件的信息。通过同样的方式可以测试其他三大组件，因为 Activity 存在任务栈的概念，本文直接通过获取任务栈的信息，然后打印出来。对于 Service 和其他组件可以根据 adb shell 的方式查看对应的信息，发现启动起来的都是本文预先注册的各个组件。

(4) 热修复



图 5-5 热修复
Figure5-5 Hot Fixed

从图 5-5 中可以发现与插件化模块的更新不同，热修复只是更新文件中的方法或者是变量而不是增加一个新功能。从旧版本变更到新版本需要经过一个重新启动的过程。

插件机制的安全性包括了两点，一个是 APK 的签名机制，一个是 dex 文件的加密。对于 APK 的签名验证，对于模块化更新是在打包 APK 文件时加入签名机制，从而在 APK 文件中生成对应的签名文件。如图 5-6 是通过解压 APK 文件获取的签名文件。

名称	修改日期	类型	大小
 CERT.RSA	2017/3/13 16:03	RSA 文件	2 KB
 CERT.SF	2017/3/13 16:03	SF 文件	37 KB
 MANIFEST.MF	2017/3/13 16:03	MF 文件	37 KB

图 5-6 APK 签名文件
Figure5-6 APK signature file

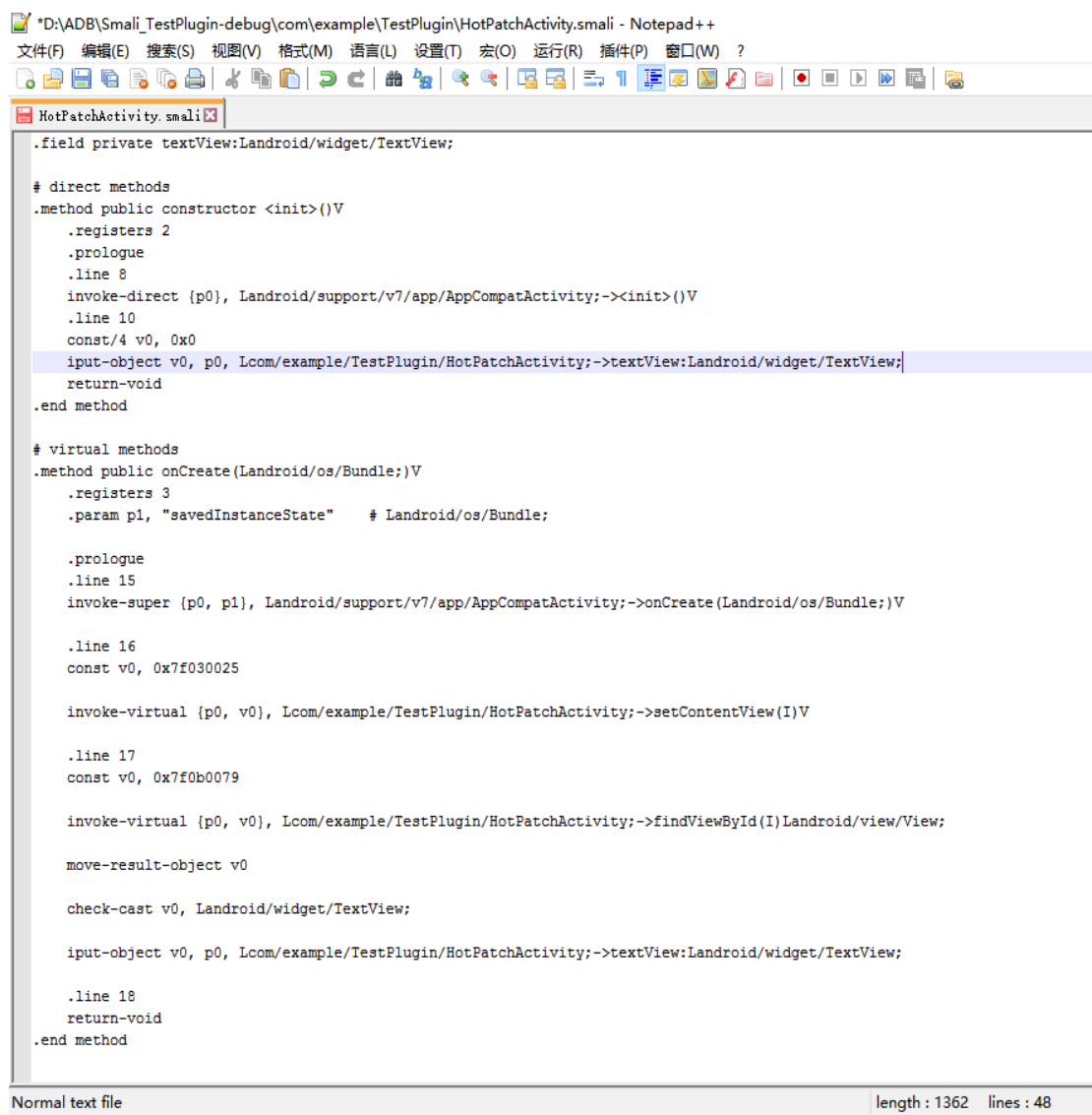
在通过 HOOK 方式解析 APK 的过程中，利用 JarVerifier.VerifierEntry.verify 对 APK 文件中包含的所有文件对应的摘要值与 MIFEST.MF 的值进行比较，验证 APK 中的文件是否被修改；使用 RSA 证书文件检验签名文件.SF 文件是否被修改过的；使用签名文件 CERT.SF，检验 MANIFEST.MF 文件中的内容是否被篡改过。



图 5-7 修改 APK 文件安装失败
Figure5-7 Install APK failure by updating

如果 APK 文件中出现任何的修改就会出现上图 5-7 的情形，导致加载模块失败。因为 dex 文件不能直接查看，通常都是通过反编译工具反编译后来查看 smali 文件。

因此对于 dex 文件加密的验证方式只能利用工具来将 dex 文件进行反编译,通过查看 smali 文件来验证该 dex 文件的安全性。本文将利用 APKDB 来对 dex 文件进行反编译获取 smali 文件。如果 dex 文件未进行加密,利用该工具可以顺利编译获取到 smali 文件,如图 5-7。



```
*D:\ADB\Smali_TestPlugin-debug\com\example\TestPlugin\HotPatchActivity.smali - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(T) 宏(O) 运行(R) 插件(P) 窗口(W) ?

HotPatchActivity.smali
.field private textView:Landroid/widget/TextView;

# direct methods
.method public constructor <init>()V
    .registers 2
    .prologue
    .line 8
    invoke-direct {p0}, Landroid/support/v7/app/CompatActivity;-><init>()V
    .line 10
    const/4 v0, 0x0
    iput-object v0, p0, Lcom/example/TestPlugin/HotPatchActivity;->textView:Landroid/widget/TextView;
    return-void
.end method

# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .registers 3
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    .line 15
    invoke-super {p0, p1}, Landroid/support/v7/app/CompatActivity;->onCreate(Landroid/os/Bundle;)V

    .line 16
    const v0, 0x7f030025

    invoke-virtual {p0, v0}, Lcom/example/TestPlugin/HotPatchActivity;->setContentView(I)V

    .line 17
    const v0, 0x7f0b0079

    invoke-virtual {p0, v0}, Lcom/example/TestPlugin/HotPatchActivity;->findViewById(I)Landroid/view/View;

    move-result-object v0

    check-cast v0, Landroid/widget/TextView;

    iput-object v0, p0, Lcom/example/TestPlugin/HotPatchActivity;->textView:Landroid/widget/TextView;

    .line 18
    return-void
.end method

Normal text file                                     length : 1362   lines : 48
```

图 5-8 反编译 smali 文件

Figure5-8 The decompile smali file

通过分析该文件可以得到代码的关键信息,也可以再利用工具将 smali 文件直接转化成 java 代码,对于未加密的 dex 文件的破译是比较简单的。对于已经加密的文件在使用反编译工具进行编译时会自动退出不会生成对应的文件。因此利用此方式可以有效的维护插件化模块的安全性。

5.6.2 相关机制对比

目前市场上应用相对比较广泛和稳定的插件化框架包括手机淘宝的 ACDD、360 手机助手使用的 DroidPlugin 以及携程的 Dynamic load apk (DLA)。本文将利用是否支持四大组件、组件是否具有完整的生命周期、Notification 能否插件化、是否支持.so 库、是否

具有独立进程、能否实现热修复以及是否采取相关安全措施维护应用的安全性这几个特征与本文所设计的机制进行对比。

表 5-1 市场上各插件化机制对比

Table 5-1 The Comparison of plug-in frameworks in the market

—	支持四大组件	组件完整生命周期	Notification	独立进程	支持 .so 库	安全机制	热修复
ACDD	是	是	否	否	否	否	否
DLA	否	否	否	否	否	否	否
DroidPlugin	是	是	否	是	否	否	否
本文机制	是	是	否	是	否	是	是

从上表可以看出本文所设计的机制具有明显的优势。本文的机制最大的特点就是具有热修复的功能，同时采用了相应的安全机制来保障整个应用的安全。上表的几个框架都不能插件化 Notification 以及 .so 库的插件化，因此这也是本机制需要在后续完善的地方。

插件化机制是否支持四大组件、组件是否具有完整生命周期是十分重要的，对于一个完整的应用通常都包含有这四大组件，因此在 App 后续的更新过程中不可避免的会遇到需要新增相应组件的情况，并且 Activity 与 Service 是否具有完整的生命周期将影响到插件时候能够独立的运行。插件运行在独立的进程中可以将各个模块的崩溃进行隔离，从而增强 App 的健壮性。在机制中增加热修复将功能模块的更新与 bug 的修复结合起来了，使机制的功能更完善。机制中增加相应的安全机制则可以更好的维护应用数据隐私的安全。

5.7 本章小结

本章主要是对插件化制进行实现和验证。首先是从 AndroidManifest 预注册、APK 解析、四大组件启动、热修复机制、安全机制这几个方面详细的介绍了插件机制的实现，接着是对插件机制验证，主要是插件的安装卸载、插件的启动、组件的启动、安全机制方面进行演示。最后将本文设计的机制与市场上常见的框架进行了对比，得出了本文机制的优势所在。

6 总结与展望

6.1 工作总结

随着移动互联网的迅猛发展,人们已经意识到手机应用带来的巨大方便,越来越离不开手机了,这种现象直接导致移动互联网的使用已经远远超过了传统的互联网。智能手机已经遍布了绝大部分的人群,各种各样的移动应用也是层出不穷,更新升级的速度也越来越快。如果每次的更新与升级都需要用户重新下载整个应用的安装包,然后进行安装,这种方式不仅浪费用户流量,对于用户的体验大打折扣,甚至直接导致用户卸载该应用,而且对于应用的发布者也是很很不方便的。针对上面的这些情况,市场上也有一些解决方案,例如: H5 开发、react-native 和 native 的插件化。因为 H5 和 react-native 的一些缺点,本文选择了 native 插件化的方式。

本文首先通过研究与分析插件化过程中所要用到的主要技术和机制,对 Android 系统中通信方式 Binder 的通信原理有了深入的了解,同时也意识到 Binder 在插件机制中起到了至关重要的作用。要想实现插件化,就需要寻找合适 HOOK 点,然后对 Binder 进行 HOOK,通过 Binder 的欺上瞒下的方式,来实现插件模块对系统资源和服务的访问。

接着对所要实现的插件机制进行了分析。本文根据插件化粒度的大小将插件化划分为大粒度插件化的模块化更新和小粒度的热修复。并且对插件化的位置、代码的加载和应用安全做了逐一的分析。随后还对插件机制进行总体和详细的设计。其中主要是对插件的加载模块、大粒度插件化模块化更新模块、热修复模块、插件启动、以及插件机制的安全进行了详细的设计。在整个设计的过程中利用进程的启动、匹配、死亡的过程对进程进行合理的分配,使用 DES 算法对 dex 文件进行加密,其中 DES 算法密钥的安全性是采取随机为每个模块生成不同的密钥的方式保障。

最后再对设计的插件机制进行了实现和验证,主要是从四大组件插件化预注册、APK 解析和热修复来介绍了插件机制的实现,同时对插件的安装与卸载、组件的启动、进程的随机分配、插件的安全性等进行验证。

6.2 工作展望

虽然插件机制已经得到了实现,但是依然存在一些不足,这些不足需要在后面的工作中进行改进。下面是本文插件机制的不足点:

(1) 在本文研究的机制中并没有设计对 .so 库的插件化,通常在游戏底层进行交互的都是 native 层的交互,但是本文的插件机制本没有对该 native 层进行 HOOK。

(2) 插件中四大组件无法通过 Intentfilter 与除宿主程序以外的其它第三程序通信。因为系统本身也存在一个限制,那就是未安装的组件,系统是找不到,就无法进行匹配。

(3) 不能在插件中使用自定义资源的 Notification,比如带自定义的 RemoteLayout

的 Notification 等。

虽然本文设计的框架还具有一定的局限性,但是这个机制包含了插件模块化更新和热修复两大模块,并且提出了相应的安全机制,相比开源上的一些插件化框架,本文设计的机制功能更多,安全性更好。不过可以预测在以后插件化框架将会越来越多,功能越来越全面。

致谢

三年的研究生生活转眼即逝，回想起来自己已经在西安理工大学度过了七年，从大一懵懵懂懂的少年已经成长为一个研三能为自己未来负责的知识青年。这一切都要归功于我的亲人、朋友、老师、同学无私的帮助和母校这七年来给予我的教育和培养。

在这里首先要衷心感谢我的导师孙钦东教授。从本科开始孙钦东教授就是我的导师，在本科和研究生期间一直都为我提供了一个良好的学习锻炼的环境，让我在锻炼中不断成长，从而更好地掌握所学知识。孙老师谦逊的为人、渊博的学识以及严谨的科研作风深深的影响着我，让我学会一切从实际出发，实事求是。生活中，孙老师宽厚的待人处事方式也让我感到由衷的敬佩。在这里再次向不管是生活还是学习上给予我巨大指导和帮助的孙老师表示衷心的感谢。

其次感谢已经毕业的王健师兄、年刚师兄和付雷师兄在从事 Android 开发的过程中遇到技术难点，给予我指导和帮助。感谢王楠师兄、张景鹏师兄、王艳师姐、贺毅师兄在生活和学习上的帮助。感谢我们同级的沈奇、马松松、段惊园、肖会娟、张泽琳在我迷茫时给我建议，让我做出正确的选择。感谢沈奇、蔡亚妮师妹、刘金环师妹共同探讨、研究技术难点，都为追求完美的结果付出了辛勤的劳动。感谢实验室所有的师弟师妹们，是你们让我的研究生三年充满欢乐，谢谢你们。

最后，感谢在百忙之中为本文审稿的老师，您辛苦了，谢谢！

参考文献

- [1] 陆海洋. 移动 QQ Android 版本个人中心的设计与实现[D].哈尔滨工业大学,2013.
- [2] 华保健,周艾亭,朱洪军. Android 内核钩子的混合检测技术[J]. 计算机应用, 2014, 34(11):3336-3339.
- [3] 葛志忠. Android 应用搜索的设计与实现[D].复旦大学,2014.
- [4] 王频. 中国移动 Mobile Market 发展战略分析[D].北京交通大学,2011.
- [5] 李欢. 基于移动互联的 IM 交互设计研究[D].北京交通大学,2013.
- [6] 佳都新太科技股份有限公司. 基于 Android 的应用插件化实现方案: 中国, CN201310317921.6[P].2013-11-20.
- [7] 谢晋. 基于 Android 的阿里巴巴移动客户端的设计与实现[D].哈尔滨工业大学,2012.
- [8] 袁向英. 基于 Android 系统的数据库开发和插件技术的应用开发[J]. 电脑编程技巧与维护,2014
- [9] 丁丽萍. Android 操作系统的安全性分析[J]. 信息安全,2012.
- [10] Arnold K, Gosling J, Holmes D. Java(TM) Programming Language, The (4th Edition)[M]. Addison-Wesley Professional, 2005.
- [11] 马珂. 基于虚拟机的内核模块行为分析技术研究[D].湘潭大学,2014..
- [12] Svendsen K, Birkedal L, Parkinson M. Modular Reasoning about Separation of Concurrent Data Structures[M]// Programming Languages and Systems. Springer Berlin Heidelberg, 2013:169-188.
- [13] 王中玉,金鑫. 基于 APK 动态加载的 Android 插件化实现方法、装置及交互方法: , CN104216741A[P]. 2014.
- [14] 韩超、梁泉.Android 系统原理及开发要点详解.电子工业出版社.2010.01.
- [15] 京奇虎科技有限公司,奇智软件(北京)有限公司.一种 Android 程序中的组件管理方法和装置: 中国,CN201510148427.0[P].2015-7-1.
- [16] 南京华设科技股份有限公司.Android 模块化开发方法:中国, CN201210163461. 1[P].2012-10-10.
- [17] Lin Q,Wei-tong H,Qi W,et al. An integratted information retrieval support system for campus network[J]. Wuhan University journal of Natural Sciences,2006,11(1):42-46.
- [18] González A. A Hypervisor Based Platform to Support Real-Time Safety Critical Embedded Java Applications[J]. Computer Systems Science & Engineering, 2013, 28(28):157-168.
- [19] Language J P. Java(TM) Programming Language, The (4th Edition) (The Java Series)[J]. Addison-Wesley, 2013.
- [20] 童时中. 模块化研究及实践的现状和发展[J]. 电子机械工程, 2011, 27(2):1-8.
- [21] 赵静. Android 系统架构及应用程序开发研究[J]. 自动化与仪器仪表, 2017(1):86-87.
- [22] Rodseth L. From Bachelor Threat to Fraternal Security: Male Associations and Modular Organization in Human Societies[J]. International Journal of Primatology, 2012, 33(5):1194-1214.
- [23] Baldwin C Y,Clark K B. Design rules:The power of modularity[J]. Managing in the Modular

Age:Architetectures,Networks,and Orgnizations,2003,149.

- [24] 范德辉,高杉,倪萍. 物联网智能终端适配中间件开发模式[J]. 计算机系统应用,2015,03:151-155.
- [25] 蔡杰,郭兵. Android 动态加载方案的研究与实现[J]. 现代计算机(专业版),2017,(01):42-45.
- [26] 常煜,邓飞. Android 动态加载技术[J]. 电脑知识与技术,2016,(23):49-50+53.
- [27] 王贝. 浅谈计算机软件插件技术应用研究[J]. 西部皮革, 2016, 38(16):13-13.
- [28] <http://www.cnblogs.com/over140/archive/2011/11/23/2259367.html>.
- [29] Yoshiura N, Wei W. Static Data Race Detection for Java Programs with Dynamic Class Loading[M]// Internet and Distributed Computing Systems. Springer International Publishing, 2014:161-173.
- [30] Ehringer D. The dalvik virtual machine architecture[J]. Techn Report, 2010.
- [31] Stanley M. The Mobile Internet report: Ramping faster than desktop Internet, the mobile Internet will be bigger than most think[J]. 2009.
- [32] Herlihy M, Shavit N. The Art of Multiprocessor Programming, Revised Reprint[M]// The art of multiprocessor programming =. China Machine Press, 2013:1-2.
- [33] 安達,文幸. History and future perspective of cellular mobile communications technology[J]. Technical Report of Ieice Rcs, 2013, 113:85-90.
- [34] 王智恒. 基于动态加载技术的 Android 插件化开发框架研究与实现[D].云南大学,2016.
- [35] 朱晓佳. 基于 OSGi 的 Android 模块动态加载技术研究[D].中国海洋大学,2014.
- [36] 宫向一. 基于 OSGi 的 Android 应用模块动态加载框架设计与实现[D].中国海洋大学,2015.
- [37] Binder 机制详解—Binder 系统架构, <http://www.cloudchou.com/android/post-507.html>.
- [38] openbinder, <http://www.angryredplanet.com/hackbod/openbinder/docs/html>.
- [39] Ehringer D. The dalvik virtual machine architecture[J]. Techn Report, 2010.
- [40] <https://code.google.com/archive/p/android-application-plug-ins-frame-work/>.
- [41] Sylve J, Case A, Marziale L, et al. Acquisition and analysis of volatile memory from android devices[J]. Digital Investigation, 2012, 8(3-4):175-184.
- [42] Android A Programmer' s Guide . Jerome (J.F.) DiMarzio DOI: 10.1036、0071599886
- [43] 世纪龙信息网络有限责任公司. 安卓系统 Activity 插件的创建、调用方法及系统:中国,CN201310738209.3[P].2014-4-23.
- [44] Yellin F. The Java Virtual Machine Specification, Java SE 7 Edition[C]// Addison-Wesley Professional, 2014:27-59.
- [45] 佐冰冰. Android 平台下 Launcher 启动器的设计与实现[D]. 哈尔滨工业大学,2012:108-150.
- [46] 李刚. 疯狂 android 讲义. 电子工业出版社. 2013.3.
- [47] 刘吉成. 基于 OSGI 的 Android 动态模块加载技术的研究 [J]. 信息技术与信息化, 2015, (09):185-187.
- [48] Activity, <http://developer.android.com/reference/android/app/Activity.html>.
- [49] Gosling J. The Java Language Specification, Java SE 7 Edition[J]. 2013, 14(2-3):133-158.

- [50] Lewis J, Loftus W. Java Software Solutions: Foundations of Program Design[J]. 2008.
- [51] Portillo-Dominguez A O, Perry P, Magoni D, et al. TRINI: an adaptive load balancing strategy based on garbage collection for clustered Java systems[J]. Software Practice & Experience, 2016, 46(12):1705-1733.
- [52] Fixmo I. Managing containerized applications on a mobile device while bypassing operating system implemented inter process communication[J]. 2016.

在校学习期间所发表的论文、专利、获奖及社会评价

1. 软件著作权

云相册量产管理系统云相册量产管理系统 v1.0，登记号：2016SR168069。

2. 学术论文

Qindong Sun,Jianfen Xiong,Yan Wang.A method of image segmentation based on the JPEG file stream, the third issue of Journal of Computational Method in Sciences and Engineering.

3. 获得奖励

影巨人智能网络数码相框参加首届中国“互联网+”大学生创新创业大赛获得陕西省银奖。