Android App-Hook and Plug-In Technology
# Contents

Acknowledgments, xvii  
About the Author, xix  
About the Translators, xxi  
Introduction, xxiii

**Chapter 1** - Plug-Ins from the Past to the Future  
1.1 ANDROID PLUG-INS IN CHINA  
1.2 HISTORY OF ANDROID PLUG-IN TECHNIQUES  
1.3 USAGE OF PLUG-INS  
1.4 ANOTHER CHOICE: REACT NATIVE  
1.5 DO ALL COMPONENTS REQUIRE PLUG-INS?  
1.6 DOUBLE-OPENING AND VIRTUAL MACHINE  
1.7 FROM NATIVE TO HTML5  
1.8 SUMMARY

**Chapter 2** - The Underlying Knowledge of Android  
2.1 OVERVIEW OF UNDERLYING ANDROID KNOWLEDGE  
2.2 BINDER  
2.3 AIDL  
2.4 ACTIVITYMANAGERSERVICE  
2.5 ACTIVITY WORKING PRINCIPLES  
  2.5.1 How to Launch an App  
  2.5.2 Starting the App Is Not So Simple
2.5.2.1 Click the App Icon in Launcher and Send a Message to the AMS
2.5.2.2 The AMS Handles the Information from the Launcher
2.5.2.3 The Launcher Goes to Sleep and Informs the AMS Again
2.5.2.4 The AMS Creates a New Process
2.5.2.5 Start a New Process and Inform the AMS
2.5.2.6 The AMS Tells the New App Which Activity to Launch
2.5.2.7 The Amazon App Starts an Activity

2.6 NAVIGATION IN APP
2.7 FAMILY OF CONTEXT
2.8 SERVICE
  2.8.1 Start Service in a New Process
    2.8.1.1 The App Sends a Message to the AMS to Launch Service
    2.8.1.2 The AMS Creates a New Process
    2.8.1.3 Start a New Process and Inform the AMS
    2.8.1.4 The AMS Sends Information to the New Process
    2.8.1.5 New Process to Launch Service
  2.8.2 Start a Service in the Same Process
  2.8.3 Bind a Service in the Same Process
    2.8.3.1 The App Sends a Message to the AMS to Bind a Service
    2.8.3.2 The AMS Sends Two Messages to the App Process
    2.8.3.3 The App Receives the First Message
    2.8.3.4 The App Receives the Second Message and Sends a Binder Object to the AMS
    2.8.3.5 AMS Informs the App
2.9 **BROADCASTRECEIVER** 47
   - 2.9.1 Registration 48
   - 2.9.2 Send a Broadcast 49

2.10 **CONTENTPROVIDER** 51
   - 2.10.1 The Essence of the *ContentProvider* 54
   - 2.10.2 The ASM 54
   - 2.10.3 Communication between *ContentProvider* and the AMS 56

2.11 **THE PMS AND APP INSTALLATION PROCESS** 57
   - 2.11.1 PMS Introduction 57
   - 2.11.2 App Installation Process 58
   - 2.11.3 PackageParser 59
   - 2.11.4 ActivityThread and PackageManager 60

2.12 **CLASSLOADER** 61

2.13 **PARENT-DELEGATION** 63

2.14 **MULTIDEX** 63

2.15 **A MUSIC PLAYER APP** 65
   - 2.15.1 A Music Player Based on Two Receivers 65
   - 2.15.2 A Music Player Based on One Receiver 71

2.16 **SUMMARY** 77

---

**Chapter 3 ▪ Reflection** 79

3.1 **BASIC REFLECTION** 79
   - 3.1.1 Get the Class Using a String 80
     - 3.1.1.1 Get the Class Using a String 80
     - 3.1.1.2 Class.forName 80
     - 3.1.1.3 Property class 80
     - 3.1.1.4 Property TYPE 80
   - 3.1.2 Get the Property and Method of the Class 81
     - 3.1.2.1 Get the Constructor of the Class 81
     - 3.1.2.2 Invoke a Private Method of the Class 83
3.1.2.3 Invoke a Private and Static Method of the Class

3.1.2.4 Get a Private Field of the Class and Modify Its Value

3.1.2.5 Get the Private Static Field of the Class and Modify Its Value

3.1.3 Generics and Singleton<T>

3.2 jOOR

3.2.1 Get a Class from a String

3.2.1.1 Get a Class from a String

3.2.1.2 Get a Class by Using on and get

3.2.2 Get the Property and Method of a Class

3.2.2.1 Get a Constructor of a Class

3.2.2.2 Get the Private Method of the Class

3.2.2.3 Get the Private and Static Method of the Class

3.2.2.4 Get the Private Field of the Class

3.2.2.5 Get the Private and Static Field of the Class

3.2.3 Generics and Singleton<T>

3.3 ENCAPSULATED CLASSES OF THE BASIC REFLECTION

3.3.1 Get a Constructor

3.3.2 Invoke Instance Methods

3.3.3 Invoke Static Methods

3.3.4 Get the Field of the Class and Set Its Value

3.3.5 Handle Generics

3.4 FURTHER ENCAPSULATION OF THE REFLECTION

3.4.1 Reflect a Method with Only One Parameter or without Parameters

3.4.2 Replace String with Class Type

3.4.3 Differences between the Static and Instance Fields

3.4.4 Optimization of the Field Reflection

3.5 SUMMARY
## Chapter 4 ▪ Proxy Pattern

4.1 WHAT IS A PROXY PATTERN? 105
   4.1.1 Remote Proxy 106
   4.1.2 Write Log 108
4.2 STATIC-PROXY AND DYNAMIC-PROXY 109
4.3 A HOOK ON THE AMN 111
4.4 A HOOK ON THE PMS 113
4.5 SUMMARY 114

## Chapter 5 ▪ Hooking startActivity()

5.1 INVOKE STARTACTIVITY() IN TWO WAYS 115
5.2 HOOKING STARTACTIVITY() OF THE ACTIVITY 116
   5.2.1 Solution 1: Hooking the Method startActivityForResult of Activity 118
   5.2.2 Solution 2: Hooking the Field mInstrumentation of Activity 118
   5.2.3 Solution 3: Hooking the Method getDefault() of AMN 121
   5.2.4 Solution 4: Hooking the Field mCallback of H 125
   5.2.5 Solution 5: Hooking Instrumentation Again 128
5.3 HOOKING THE METHOD STARTACTIVITY OF CONTEXT 131
   5.3.1 Solution 6: Hooking the Field mInstrumentation of ActivityThread 131
   5.3.2 Which Solution Is the Best? 133
5.4 LAUNCH AN ACTIVITY NOT DECLARED IN ANDROIDMANIFEST.XML 133
   5.4.1 How to Hook AMN 133
   5.4.2 First Half of the Hook 135
   5.4.3 Second Half of the Hook: Hooking the Field mCallback of H 139
   5.4.4 Second Half of the Hook: Hooking the mInstrumentation Field of ActivityThread 141
5.5 SUMMARY 143
### Chapter 6 • The Basic Concepts of Plug-In Techniques

- **6.1 Loading External Dex** 145
- **6.2 Interface-Oriented Programming** 148
- **6.3 Plug-In Thinning** 151
- **6.4 Debugging in Plug-Ins** 154
- **6.5 Application Plug-In Solutions** 156
- **6.6 Summary** 158

### Chapter 7 • Resources in Plug-In

- **7.1 How to Load Resources in Android** 159
  - 7.1.1 Types of Resources 159
  - 7.1.2 Resources and AssetManager 160
- **7.2 Plug-In Solutions of Resources** 161
- **7.3 Solutions for Changing Skins** 166
- **7.4 Another Plug-In Solution for Changing Skins** 178
- **7.5 Summary** 179

### Chapter 8 • The Plug-In Solution of Four Components

- **8.1 The Simplest Plug-In Solution** 181
  - 8.1.1 Pre-Declare Activity and Service of the Plug-In in the HostApp’s AndroidManifest.xml 182
  - 8.1.2 Combine the Dex 183
  - 8.1.3 Start a Service of the Plug-In 184
  - 8.1.4 Resources in Activity 185
- **8.2 A Plug-In Solution for Activity** 188
  - 8.2.1 Launch an Activity of a Plug-In Not Declared in the AndroidManifest.xml of the HostApp 188
  - 8.2.2 Solution 1: Based on Dynamic-Proxy 193
    - 8.2.2.1 The Process of Launching an Activity 193
    - 8.2.2.2 Add a Plug-In Activity to the Cache 196
    - 8.2.2.3 Solution 1 of Loading Class in a Plug-In: Create DexClassLoader for Each Plug-In apk 201
  - 8.2.3 Hooking More Classes 202
8.2.3 Solution 2: Merge All the Plug-In Dexes into One Array 205
8.2.4 Plug-In Solution of Resources 208
8.2.5 Support LaunchMode in Plug-In 208
8.2.6 Solution 3: Hook ClassLoader 212
8.3 THE PLUG-IN SOLUTION FOR SERVICE 216
8.3.1 The Relationship Between Service and Activity 216
8.3.2 StubService 218
8.3.3 Plug-In Solution to startService() 220
8.3.4 Plug-In Solution of bindService 226
8.4 A PLUG-IN SOLUTION FOR BROADCASTRECEIVER 229
8.4.1 Receiver Overview 229
8.4.2 A Plug-In Solution for Dynamic Receiver 231
8.4.3 A Plug-In Solution for Static Receiver 231
8.4.4 A Final Plug-In Solution for Static Receiver 233
8.5 A PLUG-IN SOLUTION FOR CONTENTPROVIDER 239
8.5.1 The Basic Concept of ContentProvider 239
8.5.2 A Simple Example of ContentProvider 239
8.5.3 A Plug-In Solution for ContentProvider 242
8.5.4 The Right Time to Install a ContentProvider Plug-In 245
8.5.5 The Forwarding Mechanism of ContentProvider 246
8.6 SUMMARY 247

Chapter 9 - A Plug-In Solution Based on Static-Proxy 249
9.1 A PLUG-IN SOLUTION FOR ACTIVITY BASED ON STATIC-PROXY 249
9.1.1 The Idea of Static-Proxy 249
9.1.2 The Simplest Example of Static-Proxy 250
9.1.2.1 Jump from the HostApp to the Plug-In 251
9.1.2.2 Communication between ProxyActivity and Plug-In Activity 252
9.1.2.3 The Logic of Activity in the Plug-In 255
9.1.3 Jump in the Plug-In 255
9.1.4 Eliminate the Keyword “that” 256
9.1.5 Jump Out 259
  9.1.5.1 Preparation for Jumping Out 259
  9.1.5.2 Jump to Another Plug-In 260
  9.1.5.3 Jump to the HostApp 260
9.1.6 Use Interface-Oriented Programming in Static-Proxy 261
9.1.7 Support for LaunchMode 267
  9.1.7.1 Overview of LaunchMode 267
  9.1.7.2 Plug-In Solutions for LaunchMode 269
9.2 THE PLUG-IN SOLUTION FOR SERVICE AND
BROADCASTRECEIVER BASED ON STATIC-PROXY 271
  9.2.1 Static-Proxy in Service 271
    9.2.1.1 Issue 1 276
    9.2.1.2 Issue 2 277
    9.2.1.3 Issue 3 278
  9.2.2 Support bindService() 278
  9.2.3 StubService 280
  9.2.4 The Last Solution for Service Plug-Ins: Integration
    with Dynamic-Proxy and Static-Proxy 283
    9.2.4.1 Parse Service in the Plug-In 283
    9.2.4.2 Create a Service Object Using Reflection 285
    9.2.4.3 ProxyService and ServiceManager 287
    9.2.4.4 bindService() and unbindService() 294
  9.2.5 Static-Proxy in BroadcastReceiver 301
9.3 SUMMARY 305

CHAPTER 10  ■  Related Plug-In Techniques 307
10.1 RESOLVE THE CONFLICTS BETWEEN RESOURCES OF
THE PLUG-INS 307
  10.1.1 The Process of App Packaging 307
  10.1.2 Hook aapt 308
    10.1.2.1 Modify and Generate a New aapt Command 308
    10.1.2.2 Using This New aapt Command in
      the Project 314
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.3</td>
<td>public.xml</td>
<td>316</td>
</tr>
<tr>
<td>10.1.4</td>
<td>Plug-In Uses Resources in the HostApp</td>
<td>318</td>
</tr>
<tr>
<td>10.2</td>
<td>A PLUG-IN FRAMEWORK BASED ON FRAGMENT</td>
<td>320</td>
</tr>
<tr>
<td>10.2.1</td>
<td>AndroidDynamicLoader Overview</td>
<td>320</td>
</tr>
<tr>
<td>10.2.2</td>
<td>A Simple Plug-In Sample Based on Fragment</td>
<td>321</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Jumping Between Fragments</td>
<td>322</td>
</tr>
<tr>
<td>10.2.4</td>
<td>Jump from the Plug-In</td>
<td>324</td>
</tr>
<tr>
<td>10.3</td>
<td>DOWNGRADE</td>
<td>326</td>
</tr>
<tr>
<td>10.3.1</td>
<td>From Activity to HTML5</td>
<td>328</td>
</tr>
<tr>
<td>10.3.2</td>
<td>From HTML5 to Activity</td>
<td>334</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Support for the Backpress Button</td>
<td>340</td>
</tr>
<tr>
<td>10.4</td>
<td>PROGUARD FOR PLUG-INS</td>
<td>341</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Basic Obfuse Rules for Plug-Ins</td>
<td>341</td>
</tr>
<tr>
<td>10.4.2</td>
<td>Obfuse Without a Common Library</td>
<td>342</td>
</tr>
<tr>
<td>10.4.3</td>
<td>Obfusing with a Common Library</td>
<td>345</td>
</tr>
<tr>
<td>10.4.3.1</td>
<td>Use MultiDex</td>
<td>346</td>
</tr>
<tr>
<td>10.4.3.2</td>
<td>Modify the ProGuard File</td>
<td>349</td>
</tr>
<tr>
<td>10.4.3.3</td>
<td>Remove Redundant Dexes from plugin1.apk</td>
<td>350</td>
</tr>
<tr>
<td>10.5</td>
<td>INCREMENTAL UPDATE</td>
<td>352</td>
</tr>
<tr>
<td>10.5.1</td>
<td>The Basic Concept of an Incremental Update</td>
<td>352</td>
</tr>
<tr>
<td>10.5.2</td>
<td>Create an Incremental Package</td>
<td>353</td>
</tr>
<tr>
<td>10.5.3</td>
<td>Apply Permissions</td>
<td>353</td>
</tr>
<tr>
<td>10.5.4</td>
<td>Merge Incremental Package</td>
<td>354</td>
</tr>
<tr>
<td>10.6</td>
<td>A PLUG-IN SOLUTION FOR SO FILES</td>
<td>356</td>
</tr>
<tr>
<td>10.6.1</td>
<td>Write a Hello-World SO</td>
<td>356</td>
</tr>
<tr>
<td>10.6.1.1</td>
<td>Download NDK</td>
<td>356</td>
</tr>
<tr>
<td>10.6.1.2</td>
<td>Create a Project to Generate SO</td>
<td>357</td>
</tr>
<tr>
<td>10.6.2</td>
<td>Using SO</td>
<td>362</td>
</tr>
<tr>
<td>10.6.3</td>
<td>The Principle of Loading SO</td>
<td>363</td>
</tr>
<tr>
<td>10.6.3.1</td>
<td>Compiling SO</td>
<td>364</td>
</tr>
<tr>
<td>10.6.3.2</td>
<td>The Process of Loading SO</td>
<td>365</td>
</tr>
</tbody>
</table>
11.4 A PLUG-IN SOLUTION FOR ACTIVITY 407
11.5 A PLUG-IN SOLUTION FOR RESOURCES 407
11.6 USE FRAGMENT IN THE PLUG-IN 408
11.7 PLUG-IN SOLUTIONS FOR SERVICE, CONTENTPROVIDER, AND BROADCASTRECEIVER 409
  11.7.1 A Plug-In Solution for Service 409
  11.7.2 A Plug-In Solution for BroadcastReceiver 409
  11.7.3 A Plug-In Solution for ContentProvider 410
11.8 SUMMARY 410

APPENDIX A: SAMPLE CODE LIST, 411
INDEX, 415
Acknowledgments

I OWE A BIG THANKS TO Yi Wu, the editor of this book in Chinese, for helping me contact CRC Press to publish this book in English.

I thank Manyun Guo, my wife, for accompanying and encouraging me during the period I spent writing.

Special thanks to my 21 friends from Android forums for helping me translate this Chinese book into English.

I thank Yong Zhang, Yugang Ren, Lody, Guangliang Lin, Jian Huang, and a lot of other friends. Without your endless support I wouldn’t have been able to talk as deeply about this technique domain.
Jianqiang Bao is a senior Android app developer. For more than 15 years, he has developed enterprise solutions using Silverlight, ASP.NET, WP7, Android, and iOS. He has worked at HP, Microsoft, Tuniu and Qunar. He has a technique blog at http://www.cnblogs.com/jax; his GitHub is https://github.com/BaoBaoJianqiang.
About the Translators

Many people have participated in this book’s translation from Chinese to English, the list is as follows:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Translator</th>
<th>Reviewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hongwei Cao</td>
<td>Han Yan</td>
</tr>
<tr>
<td>2</td>
<td>Chunfei Shi, Xuelong Wang, Xiaohui Li</td>
<td>Fangxiang Deng</td>
</tr>
<tr>
<td>3</td>
<td>Wenpeng Li</td>
<td>Jianqiang Bao</td>
</tr>
<tr>
<td>4</td>
<td>Xizhi Pan</td>
<td>Jinyu Guo</td>
</tr>
<tr>
<td>5</td>
<td>Jian Feng</td>
<td>Guiming Zou</td>
</tr>
<tr>
<td>6</td>
<td>Xiaohui Li</td>
<td>Zelong Gong</td>
</tr>
<tr>
<td>7</td>
<td>Jinyu Guo</td>
<td>Tong Peng</td>
</tr>
<tr>
<td>8</td>
<td>Tianhong Han Guiming Zou</td>
<td>Wenhan Xiao</td>
</tr>
<tr>
<td></td>
<td>Xuelong Wang</td>
<td>Sheng Li</td>
</tr>
<tr>
<td></td>
<td>Yupeng Wang</td>
<td>Jian Feng</td>
</tr>
<tr>
<td>9</td>
<td>Alan Pan T</td>
<td>Xizhi Pan</td>
</tr>
<tr>
<td></td>
<td>Siyang Long</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Shuaifeng Ma</td>
<td>Tong Peng</td>
</tr>
<tr>
<td></td>
<td>Fangxiang Deng</td>
<td>Jian Feng</td>
</tr>
<tr>
<td></td>
<td>Tong Peng</td>
<td>Fangxiang Deng</td>
</tr>
<tr>
<td></td>
<td>Zhaoyun</td>
<td>Siyang Long</td>
</tr>
<tr>
<td></td>
<td>Zelong Gong</td>
<td>Guiming Zou</td>
</tr>
<tr>
<td></td>
<td>Hao Yang</td>
<td>Xizhi Pan</td>
</tr>
<tr>
<td></td>
<td>Jinyu Guo</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Jinyu Guo</td>
<td>Siyang Long</td>
</tr>
</tbody>
</table>
Introduction

Welcome to the first edition of Android App-Hook and Plug-In Technology

WHAT THIS BOOK WILL TEACH YOU

This book will teach you everything you need to know to master Android plug-in techniques.

This book introduces the Android plug-in technique. An app can be downloaded as an apk file in a zip file from the remote server. We call this zip file a plug-in. This app can invoke a class in this plug-in. This means that we can update the content of the app without republishing it again.

Google Play has a strict app auditing strategy. It forbids any app from downloading another app to prevent it from downloading malicious content or pornographic and violent content. Thus, we cannot publish an app using this plug-in technique on Google Play.

This book will teach you the underlying knowledge of the Android system, which help you master Android technique at a high level.

After reading this book, you will be familiar with several aspects of the Android system, including the following content:

- Binder and AIDL mechanisms.
- The working mechanisms of Activity, Service, ContentProvider, and BroadcastReceiver.
- Communication between ActivityManagerService and four components.
- How to launch an app.
- LaunchMode.
- The families of Context and ClassLoader.
**Introduction**

- *MultiDex.*
- How to load SO.
- *PackageManagerService* and how to install an app in the Android system.
- Reflection.
- The *Proxy.newProxyInstance()* method for adding an external function to the original API.

**WHO IS THIS BOOK FOR?**

Don’t use the plug-in techniques introduced in this book on Google Play; it’s forbidden.

This book introduces a lot of knowledge about the Android system which is useful to app developers.

**THE SAMPLE CODE**

The sample code in this book is on my Github: https://github.com/BaoBaoJianqiang/.

There are 74 demos in this book, and I list the address of each demo in the corresponding chapter and section.

In Appendix A, I list all the demos with their corresponding chapter and section.

**THE BOOK’S STRUCTURE**

This book consists of 11 chapters: chapters 1 to 5 introduces the basic knowledge of plug-in techniques; chapters 6 to 10 introduces several solutions for plug-in programming issues; Chapter 11 is an overall summary.

Below is a brief overview of the chapters.

Chapter 1 introduces the history of Android plug-in techniques.

Chapter 2 introduces the underlying Android system, including *Binder* and *AIDL*, *ActivityManagerService*, *PackageManagerService*, *ActivityThread*, *LoadedApk*, and so on. As this book is written for app developers, I illustrate these concepts with a series of pictures rather than code.

Chapter 3 introduces the syntax of reflection, and the encapsulation of the reflection, including *jOOR*, a famous Java reflection framework. Reflection is the basis of plug-in techniques.
Chapter 4 introduces proxy patterns, including Static-Proxy and Dynamic-Proxy, these two proxies generate two important plug-in frameworks, DroidPlugin and DL. `Proxy.newProxyInstance()` is a widely used plug-in, and we use this method to hook `IActivityManager` and `IPackageManager` in this chapter.

Chapter 5 introduces how to start an Activity not declared in the `AndroidManifest.xml`, based on the `Proxy.newProxyInstance()` introduced in Chapter 4.

Chapter 6 introduces the basic knowledge of plug-ins, including how to debug from the Hostapp to the plug-in, and interface-oriented programming.

Chapter 7 introduces how to load Resources into plug-ins. `AssetManager` and `Resources` are key points, especially the method `addAssetPath()` of `AssetManager`. Based on this technique, we can dynamically change a skin.

Chapter 8 introduces plug-in solutions for `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. A different mechanism of these four components results in different plug-in solutions.

Chapter 9 introduces a plug-in framework based on Static-Proxy. The creator of this framework invented a new keyword “that,” so this framework is also called “That.” “That” is a very smart framework; it’s also well known as Puppet.

Chapter 10 considers other related plug-in techniques. Including how to resolve conflicts between the resource ID in plug-ins, how to use fragments in plug-ins, how to replace HTML5 with `Activity`, how to use `ProGuard` in plug-ins, how to reduce the size of plug-ins, how to download a SO file dynamically, and how to support the Android O and P systems with plug-ins.

Chapter 11 summarizes all the plug-in techniques mentioned in this book.

CONTACTING THE AUTHORS

If you have suggestions, remarks, or questions on plug-in techniques and sample code, please contact the author on: 16230091@qq.com.
Chapter 1

Plug-Ins from the Past to the Future

Google Play has a strict app auditing strategy. For example, it forbids any app to download another app to prevent it from downloading bad content, pornographic and violent content, for example.

In addition, Google Play forbids app developers from modifying the original behavior of the API of the Android system, which is not open to the app developers. For example, the method `addAssetPath()` of the `AssetManager`, and the method `currentActivityThread()` of the `ActivityThread`. Also, Android P launched a new mechanism named the grey-list and black-list. If the developer modifies the APIs through the two lists above, these APIs will print a warning or return `null` directly.

The auditing strategy in the Chinese app market is less strict. Downloading and launching are allowed and there are two main techniques widely used in China; one is plug-in, the other one is hot-fix.

1.1 ANDROID PLUG-INS IN CHINA

The plug-in technique separates one app into a lot of small apps for different business purposes; the OTA* app, for example, consists of hotels, flights, cars, and other domains. We can separate these domains into several small apps, such as a hotel app, flight app, and car app, and all these small apps are called plug-in apps. As all the businesses are separated into

* OTA: Online Travel Agent
different plug-in apps, only the home page is left in the main app (and is called the Hostapp). When users click the button in the Hostapp, it will navigate to the small apps.

In traditional app coding strategy, all the code and logic should be in one app. When we find bugs in the app, there is only one way to solve this online bug; it is to package this app again and submit it to the Android app market. However, the users must download the latest version of this app to remedy the bugs. This is not a good solution; it’s not user-friendly. Most users don’t want to waste time updating an app.

Android plug-ins are a very good solution to the problem above. If there is a bug in one plug-in, we just need to package this plug-in app again, and then put this new plug-in on the remote server. When the app user opens the Hostapp, it will download this new plug-in in the background thread automatically. When downloaded successfully, the user needs to restart the app and the bugs will have been eliminated from the app.

The plug-in technique is not only used to fix bugs but is also suitable for rapid software development. In traditional app development, you launch a new version of the app every month. It is very common for a very big company to have 100 product requirements needing to be coded within one month. Any delay in development causes some products to launch later than planned. By using the plug-in technique, the different apps can be launched individually; there will be no time limit.

In China, the hot-fix technique was developed using similar ideas to plug-ins. The hot-fix technique is useful for fixing online bugs. When app developers find online bugs, they can fix the codes and then package the code modification into a zip file; then upload this zip file to the remote server, so that users can download this zip file dynamically. After the users have downloaded the zip successfully, the app will decompose this zip file and substitute the old code with the new code in the zip file.

The plug-in technique and hot-fix technique were developed using very similar ideas. The plug-in technique loads outside the apps by hooking the Android internal system API. The hook occurs in the Java code, where the hot-fix is happening in NDK, which means C++. A hot-fix replaces the pointer of the old method with a pointer of the new method.

This book focuses on the plug-in technique.

1.2 HISTORY OF ANDROID PLUG-IN TECHNIQUES

On July 27, 2012, the first milestone in Android plug-in technology was reached. Yimin Tu, who worked for Dianping.com, released the first
Android plug-in open source project, *AndroidDynamicLoader*, and the Dianping.com app was based on this framework. This plug-in framework is based on Fragment. The app has only one activity; all the pages are implemented by fragments and loaded by this activity. Some fragment pages are plug-ins, which can be downloaded dynamically. This plug-in framework was the first time anyone used the method `addAssetPath()` of the `AssetManager` to handle the `Resources` in the plug-in app.

In 2013, 23Code appeared. 23Code provides a container where plug-ins can be dynamically downloaded and run. We can write a variety of `UserControls` and run them in 23Code. It is an Android plug-in framework, but without source code and not widely known.

On March 27, 2013, Bokui, the developer of the Taobao app, shared technical information on Taobao’s plug-in framework. The name of this plug-in framework is Atlas. In this topic, he introduced a way to modify the internal API of Android, incremental update, downgrade, compatibility, and so on. It’s a pity that this plug-in framework is not an open source project. We can’t learn more from this topic.

At 8:20 on March 30, 2014, the second milestone of Android plug-in technology was reached. Yugang Ren published an Android plug-in project named `dynamic-load-apk`, which was not the same as the other plug-in projects. It did not modify the internal methods of the Android system, but solved problems from the application layer of the app by creating a class named `ProxyActivity` to distribute and start the activity of the plug-in. Yugang Ren invented a keyword called “that” in this framework, it’s also called the “That” framework in this book. In fact, the creator does not like this nickname and named it `DL` for short. When he developed this framework, there were so many difficulties, because there was not enough information on Android plug-in technology that could be referred to, especially before 2014.

The “That” framework only has the implementation of `Activity` at the beginning. With the contribution of Xiao Tian and Siyu Song, the implementation of `Service` was available later. In April 2015, the “That” framework was stable.

At the same time, Tao Zhang was also contributing to the implementation of plug-in technology. In May 2014, after reading all the source codes of `DL`,

---

* https://github.com/mmin18/AndroidDynamicLoader
† http://v.youku.com/v_show/id_XNTMzMjYzMzM2.html
‡ https://github.com/singwhatiwanna/dynamic-load-apk
he released his first plug-in framework, \textit{CJFrameForAndroid}*. This design was similar to the “That” framework. In addition, the \textit{CJFrameForAndroid} framework provided a plug-in solution called \textit{LaunchMode}, which was a very important contribution to plug-in techniques.

In November 2014, Houkx released a plug-in project named \textit{android-pluginmgr} on GitHub†. This framework first proposed registering a \textit{StubActivity} in the \texttt{AndroidManifest.xml} to cheat the AMS but opened an \textit{Activity} in a plug-in. At the same time, Houkx also found that all the permissions should be declared in the \texttt{AndroidManifest.xml} of the plug-in in advance.

On December 8, 2014, there was good news, Android Studio V1.0 was available. Android developers began to gradually abandon Eclipse to use Android Studio. Android Studio is compiled and packaged with Gradle, which makes the design of plug-in frameworks much simpler, eliminating the inconvenience of using Eclipse to run the Android SDK.

Then, though, 2015 was coming. Lody, an 18-year-old boy, began using Android in 2015 when he was a senior high school student. He had studied the source code of the Android system for 3 years. His first well-known open source project was \textit{TurboDex}‡, which could quickly load \texttt{dex} with high speed. This is a very useful plug-in framework because it usually takes a long time to load all the plug-ins for initialization.

At the end of March 2015, Lody released the plug-in project: \textit{Direct-Load-Apk}§. This framework combined two thoughts mentioned earlier; one was \textit{Static-Proxy}, from Yugang Ren’s “That” framework, the other one was to cheat the AMS, from Houkx’s \textit{pluginmgr} framework. \textit{Direct-Load-Apk} is not widely known, because Lody had too much school homework.

The legend of Lody wasn’t finished yet; he spent a lot of time on \textit{VirtualApp}. \textit{VirtualApp} is like a virtual machine on the Android system. It can install and run other apps. We’ll discuss \textit{VirtualApp} in Section 1.6.

In May 2015, Limpoxe released the plug-in framework: \textit{Android-Plug-in-Framework}¶.

\footnotesize
\begin{itemize}
\item [*] \url{https://github.com/kymjs/CJFrameForAndroid}
\item [†] \url{https://github.com/houkx/android-pluginmgr}
\item [‡] \url{https://github.com/asLody/TurboDex}
\item [§] \url{http://git.oschina.net/oycocean/Direct-Load-apk}
\item [¶] \url{https://github.com/limpoxe/Android-Plugin-Framework}
\end{itemize}
In July 2015, Kaedea released the plug-in framework: *Android-dynamical-loading*.

On August 27, 2015, the third milestone of Android plug-in technology was reached, Yong Zhang’s plug-in framework *DroidPlugin* came out. Yong Zhang was a developer at Qihoo360, and *DroidPlugin* was used in his team. The magic of this framework is that any app can be loaded into the HostApp. You can write a HostApp based on this framework, and then load apps written by others as plug-ins.

*DroidPlugin* is a powerful plug-in framework, but its disadvantage is obvious. It modifies too many internal APIs of the Android system. Due to the lack of literature on the *DroidPlugin* framework, it is difficult to understand. There are many articles about *DroidPlugin* on blogs and forums, but the best one is written by WeiShu Tian. He also worked at Qihoo360 and had the opportunity to talk about *DroidPlugin* with its creator. He then wrote a series of articles about the *DroidPlugin*, including the principles of Binder, AIDL and the plug-in mechanism for the Activity, Service, BroadcastReceiver, and ContentProvider.

The year 2015 was the first year of Android plug-in development. Not only the “that” framework and *DroidPlugin* but many other plug-in frameworks were also born at that time.

The project *OpenAtlas* was released on GitHub in May and was later renamed *ACDD*. It proposes modifying the command aapt so that the resource ID of the plug-in is no longer a fixed value of 0x7f, but can be modified to a value such as 0x71. aapt is a command line tool supplied by Android. It’s used to generate resource IDs during the packaging process of an Android app. This technique solves the problem of resource ID conflict after merging the resources of the plug-in and the Hostapp together.

*OpenAtlas* hooks the method execStartActivity() of Instrumentation to load the Activity of the plug-in dynamically.

In addition, *OpenAtlas* also modifies ContextWrapper, and rewrites the method getResources(). Because the Activity is the subclass of the ContextWrapper, the Activity of the plug-in inherits the method getResources() to get the resources of the plug-in. We can’t use this method anymore; we need to create a parent class named BasePluginActivity, all the

---

* https://github.com/kaedea/android-dynamical-loading
† https://github.com/Qihoo360/DroidPlugin
‡ Blog address: http://weishu.me
Activities of the plug-in should inherit `BasePluginActivity` and override the method `getResources()` to fetch the resources of the plug-in.

Ctrip.com released their plug-in framework `DynamicAPK`* in October 2015, which was based on the `OpenAtlas` framework.

At the end of December 2015, Guangliang Lin published his plug-in framework, `Small`. At that time, he worked for a car trading platform and his framework was used for this platform.

`Small` has a lot of interesting features:

- Merges all `dex` files of the plug-in into a `dex` array of the Hostapp. This means the HostApp can load any class of plug-in.

- Declares `StubActivity` in the `AndroidManifest.xml` to cheat the AMS. This solution is the same as the `DroidPlugin`.

- Pre-declares `Service`, `Receiver`, and `ContentProvider` of the plug-in in the `AndroidManifest.xml` of the Hostapp to support the other three components.

- Invokes the method `addAssetPath()` of the `AssetManager`, and all the resources of the plug-in are merged into the resources of the Hostapp. If a resource ID conflict occurs, `Small` modifies `R.java` and `resource.arsc` in the packaging process. After these two files are generated, `Small` will change the resource ID in these two files; for example, from 0x7f to 0x71.

At the end of 2015, all the technical issues of plug-ins were resolved. That year, the plug-in technology was very varied. A lot of open source plug-in frameworks were born in 2015. These frameworks were almost all invented by individuals. It was basically divided into two categories. Dynamic-Hook was invented by Yong Zhang in `DroidPlugin`, and Static-Proxy was invented by Yugang Ren in the “That” framework.

In 2015, Android hot-fix technology and `React Native` appeared in the app developer’s world, which provided the same advantages as Android plug-in technology. Android plug-in technology was no longer the only choice for app developers.

---

* https://github.com/CtripMobile/DynamicAPK
† https://github.com/wequick/Small
Since 2016, many internet companies have released their plug-in frameworks. These frameworks focus on stability and compatibility to benefit their millions of daily users.

Let us enumerate these frameworks in order:

- In August 2016, iReader published its plug-in framework named ZeusPlugin*.
- In March 2017, Alibaba published its plug-in framework named Atlas†.

In June 26, 2017, Qihoo360 published its second plug-in framework named RePlugin‡, which is different from DroidPlugin.

On June 29, 2017, Didi published its plug-in framework named VirtualApk§.

All the plug-in frameworks published on github by internet companies focus on:

- Plug-in compatibility, including the impact of the upgrade of the Android system on the plug-in framework, and the impact of different mobile phone ROMs.
- Plug-in stability, for example, crashes.
- Management of plug-ins, including installation and uninstallation.

In spite of the fact that a few years have passed, various plug-in frameworks have gradually become stable. Developers who are now engaged in plug-in technology only need to pay attention to the annual upgrade of the Android system and add code to be compatible with the latest updates.

With the maturity of the plug-in framework, many authors of plug-in technology have begun to change their focus. Some of them are still sticking with Android; for example, Yugang Ren is still working in the Android domain; and some of them have moved to Blockchain where they focus on writing smart contracts with the Go language every day.

* https://github.com/iReaderAndroid/ZeusPlugin
† https://github.com/alibaba/atlas
‡ https://github.com/Qihoo360/RePlugin
§ https://github.com/didi/VirtualAPK
Thanks to those who have contributed to plug-in technology, including the authors of plug-in frameworks, as well as the authors who have written a series of articles to introduce this technique.

### 1.3 USAGE OF PLUG-INS

Once upon a time, we naively thought that Android plug-ins were intended to add new features.

It took a lot of time and effort, but after the project was implemented with plug-in frameworks, we discovered that 80% of the usage of plug-ins was to fix online bugs. At this point, it has the same capabilities as hot-fix techniques such as AndFix* and Tinker†, and even better than these hot-fix frameworks.

The app always releases a new version every two weeks. Generally, the new feature goes online at this point. On the other hand, in some companies, the release strategy of Android app is affected by iOS app releases in the App Store.

In the time before plug-in frameworks, we were scared to write bugs; if a serious bug appeared, we had to release a new version to fix this bug. The app users would have to update the app to the newest version, and it’s not user-friendly to have to make frequent downloads to keep the app running smoothly.

With the plug-in framework, developers don’t have to worry about the quality of the code—if something is wrong, you can release a new version to fix it quickly. After the app is released, each plug-in will have one or two new releases every day.

The Android plug-in framework has become a bug-fixing tool and this is something that we don’t want to see.

In fact, the plug-in framework is more suitable for MMORPG games. There will always be new skins, or a new hero role available every few days, even for adjusting the attributes. All of these do not need to be released as a new version.

There is another use for plug-in technology, which is the ABTest, but it is not used widely. When the product manager wants to determine which will be selected from two styles of design, there will be two strategies which are made into the two plug-in packages; 50% of the users download strategy A, the other 50% download strategy B. Checking the results after

* https://github.com/alibaba/AndFix
† https://github.com/Tencent/tinker
a week, such as the page conversion rate, will tell you which strategy is better. This is called a data-driven product.

In the previous section, the componentization of Android was mentioned. That is, with the independence of the business unit, the Android and iOS teams are split into their own business and have their own organization relationships. Therefore, it is necessary to split the different services of hotels and flights into different modules. In the componentization of Android, modules are still dependent on the aar file; we can use Maven to manage these aar files.

The componentized model of Android is only applicable to the development stage. Once there is a bug in the online release, or new features are to be released, all modules must be packaged together again to deploy the new release.

Plug-in technology is the final solution based on Android componentization. At this point, each service module is no longer an aar file, but an apk file, placed in the folder assets of the Host app. In this way, after a release, some modules are updated, only the code of this module is packaged again, an incremental package is generated, and it is put on the remote server for the app users to download.

1.4 ANOTHER CHOICE: REACT NATIVE

In 2015, React Native was born. At that time, few people paid attention to it because it was still immature with only a few basic functions. Then, with the next iteration of the React Native project, the functions were much improved. Although there has not been a release version 1.0 so far, we find that it is a perfect “plug-in” framework to support both Android and iOS systems.

React Native is written based on JavaScript, packaged, and put on the remote server for Android and iOS apps to download and use.

For small-sized or medium-sized companies and startups, who don’t have the human and financial resources to develop a plug-in framework, generally adopt a relatively stable, open source, and continuous updated plug-in framework. However, it seems that iOS has no technical framework, especially after the jsPatch (a hot-fix solution) was forbidden by the App Store. Their best choice is React Native. Once the JavaScript is recruited, it will be able to quickly iterate and release.

On React Native, there are already some books available. This book mainly introduces Android plug-in technology. This section shows some points that Android plug-in is not as good a technique as React Native.
1.5 DO ALL COMPONENTS REQUIRE PLUG-INS?

In Android, Activity, Service, ContentProvider, and BroadcastReceiver are the four major components.

Do all four components need to convert to plug-ins? Over the years, I have been working on plug-in technology with this question.

I have worked in OTA companies for several years. This kind of app is similar to e-commerce ones, including a complete set of payment processes, and Activity is the most used; 200 or 300 Activities are not surprising. The other three components are rarely used.

Most apps in China have the same situation.

According to the technology stack, the app is divided into four domains:

- Game App. People have their own online update process. Many of them use scripts like Lua.
- Mobile assistants, mobile phone guards, and the use of such applications for Service, Receiver, and ContentProvider.
- Music, video, and live video applications are very dependent on Service and Receiver, in addition to more Activities.
- E-Commerce, social, news, and reading apps use a lot of Activities. The use of the other three major components is not enough.

Different plug-in frameworks are suitable for different requirements. If an app uses a lot of the Service, BroadcastReceiver, and ContentProvider components, the plug-in framework for this app must support all four components in Android, but this plug-in framework is hard to maintain. Otherwise, a plug-in framework which only supports Activity is enough for a simple app.

1.6 DOUBLE-OPENING AND VIRTUAL MACHINE

Since plug-ins will be replaced by React Native in the following years, what is the future of plug-ins? The answer is virtual machine technology.

Some engineers already have experience of installing a virtual machine on a computer. If the computer’s memory is large enough, you can open multiple virtual machines at the same time. On each virtual machine, you can log into Twitter with a different account, and can then chat with yourself. Of course, chatting with yourself doesn’t make any sense.
Can we support installing one or more virtual machines on an Android system? Lody, a college student, is doing such work. He has a very famous open source project, VirtualApp, which is now in commercial operation*

With such a virtual machine system, we can open two Twitter apps with a different account in only one mobile phone and chat with ourselves.

The multiple instances of technology that opens an app at the same time is called double-opening. Some mobile phone systems in China now support double-opening technology. You can see this option in the settings of Android phones.

The technology of double-opening and virtual machines is outside of this book’s scope.

1.7 FROM NATIVE TO HTML5

Through the immense prosperity of app technology, I got my first job in 2004; when the computer assembly industry was transitioning from CS (Client/Server) to BS (Browser/Server) architecture. For example, if you installed MSN on your computer, you can chat with your friends through this software. After the technology of the internet grew up, you started to move the original system into the website. This is BS architecture.

Compared to CS, BS was a thin client; many features were not supported by BS. Designers came up with the concept of a SmartClient, which is a CS pattern. Outlook is a good sample. You can read and write email offline without a network; it sends a written email automatically.

Then, Flash became hot. Flash was the originator of the web-rich client. Based on Flash, there is Flex, which is now being adopted by more and more companies. At this time, Microsoft also created Silverlight, which is like Flash but on the web. At the same time, JavaScript was also working hard and became the final winner in the rich client space. A book on JavaScript was very popular at that time; it is called “JavaScript Design Patterns.”

JavaScript was only used for web visual effects in 2004. From 2005 onward, JavaScript has experienced additions with Ajax, jQuery, ECMAScript 1 to 6, and so on. The frameworks of Angular, React, and Vue have become extremely powerful and have been packaged due to JavaScript as an “object-oriented” language.

* https://github.com/asLody/VirtualApp
Compared to web technology, the app is taking the same development path. In the first five years of mobile development, app developers used basic syntax to write each app. The experience was not user-friendly. The networking speed was not fast and many white screens were found in the apps, and there were a lot of bugs and crashes. Over the following five year of mobile development, however, more and more mobile techniques were invented by app developers, such as RxJava, ButterKnife, jsPatch, Tinker, and a lot of plug-in frameworks. The next stage is the transition from CS to BS. The hybrid technology is the above-mentioned BS, but there are many defects, especially the bad performance of web browsers. Then there is React Native. HTML5 is also slow, but you can translate HTML5 into Native code. I don’t know which techniques will appear in the next five years, but the trend is obvious. Android and iOS technology will not die; on the other hand, HTML5 will become the main method for app development in the coming years.

1.8 SUMMARY
In this chapter, we reviewed the history of Android plug-in technology, which is basically divided into two parts, Dynamic-Proxy and Static-Proxy. All plug-in frameworks are based on these two parts. After the history review, we found that plug-in technology was not accomplished at one stroke, but it has experienced a process of gradual improvement.

Plug-in technology is not only used to fix bugs and dynamically release new features. In the process of researching plug-in technology, we have developed the Android virtual machine and double-opening technology. This is a new technology area that can bypass Android system limitations and run the app faster.

React Native was also mentioned, which also fixes bugs and dynamically releases new features, just like Android plug-in technology. Which technology is selected depends on whether the R&D team is based on HTML5 or Android and depends on whether it will be released on Google Play.