Broadview
www.broadview.com.cn

Android进阶三部曲第三部

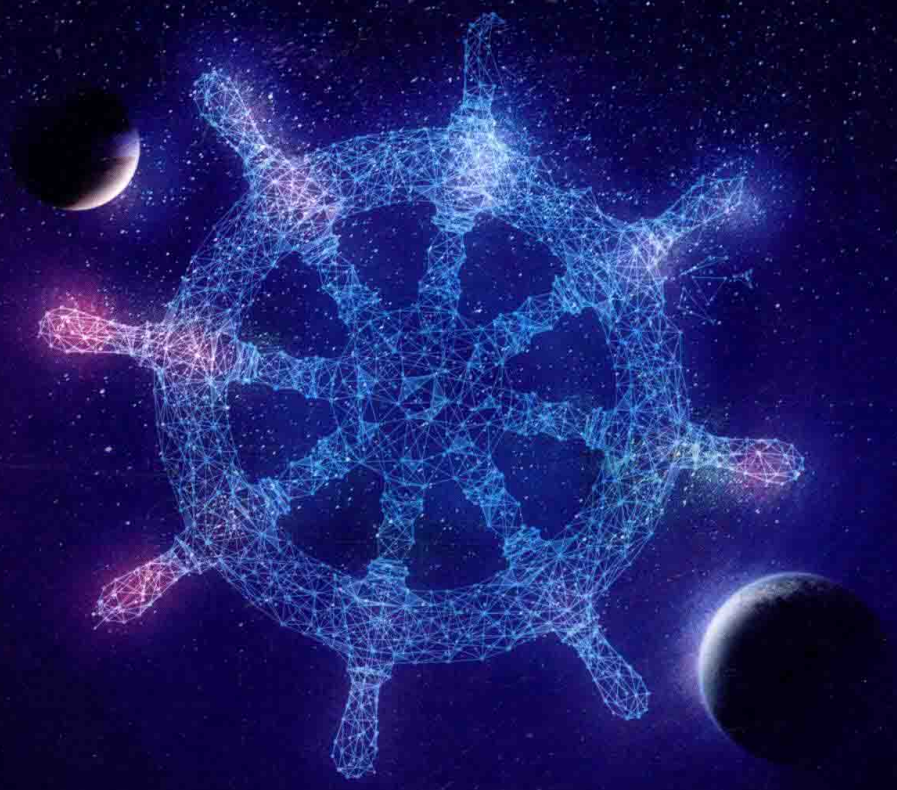
畅销书《Android进阶之光》《Android进阶解密》作者又一力作!

第三部结合前两部，共同构建Android进阶三部曲知识体系。

Android进阶三部曲是专门为应用开发进阶和面试打造的系列图书。

Android 进阶指北

刘望舒 / 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn

博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Android进阶三部曲
专门为Android应用开发进阶和面试打造的系列图书

- **第一部《Android进阶之光》**
  对Android开发进阶要点进行深入讲解，为工程师的进阶之路带来指引和光明。

Android新特性
OkHttp/Retrofit
Dagger2
……
View体系
RxJava
MVP/MVVM

- **第二部《Android进阶解密》**
  将系统源码和应用开发结合讲解，破解Android应用开发进阶奥秘。

系统、应用程序进程启动原理
AMS/WMS
ClassLoader
……
四大组件原理
JNI原理
插件化、热修复原理

- **第三部《Android进阶指北》**
  结合前两部，构建Android进阶三部曲知识体系。

系统源码的下载、编译、调试
PMS/IMS
Jetpack架构组件
……
Binder原理
Gradle核心思想
Flutter

读者服务
微信扫码回复：39375
· 获取博文视点学院在线课程、电子书20元代金券
· 获取免费增值资源
· 加入读者交流群，与本书作者互动
· 获取精选书单推荐

上架建议：移动开发>Android

ISBN 978-7-121-39375-4
9 787121 393754 >
定价：109.00元

责任编辑：付　睿
封面设计：李　玲　罗　铭

Android 进阶指北

刘望舒 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是一本 Android 进阶图书，结合《Android 进阶之光》和《Android 进阶解密》两本书介绍的知识，一同建立起 Android 应用开发进阶知识体系。全书共分 13 章，主要从 3 个方面来组织内容：（1）Android 应用开发所需要掌握的系统源码知识，包括系统源码的下载、编译、调试，包管理机制，输入系统，Native Binder，Java Binder；（2）应用开发必需的知识点，包括 Groovy 基础、Gradle 核心思想、Gradle 的 Android 插件、Android Jetpack 架构组件；（3）与跨平台开发相关的知识点，包括跨平台技术演进、Dart 基础、Flutter 基础。

本书适合有一定基础的 Android 应用开发工程师和对 Android 系统源码感兴趣的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Android 进阶指北 / 刘望舒著. —北京：电子工业出版社，2020.9

ISBN 978-7-121-39375-4

Ⅰ. ①A… Ⅱ. ①刘… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2020）第 147446 号

责任编辑：付　睿　　特约编辑：田学清
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：30.25　　字数：642 千字
版　　次：2020 年 9 月第 1 版
印　　次：2020 年 9 月第 1 次印刷
定　　价：109.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei. com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前言

为什么写这本书

Android 进阶三部曲包括《Android 进阶之光》、《Android 进阶解密》和《Android 进阶指北》，写这本书的原因和前两本书有些关联，主要有以下几点：

（1）我一直想建立一个知识体系，但是《Android 进阶之光》和《Android 进阶解密》不能形成具有一定规模的知识体系，而本书的出版将会打破这一点，三本书初步形成了知识体系，即 Android 进阶三部曲知识体系。

（2）目前市面上的关于 Android 系统源码分析的书大多不是专门为应用开发编写的，而本书中的系统源码分析是专门为应用开发编写的。

（3）目前市面上的关于 Android 应用综合类进阶的图书很少，2017 年有《Android 进阶之光》，2019 年有《Android 进阶解密》，2020 年我仍要贡献出自己的力量。

（4）目前市面上的关于 Android 源码分析类的图书大多基于 Android 6.0 之前的版本，需要有一本书来对更新版本的系统源码进行分析。

（5）《Android 进阶之光》和《Android 进阶解密》覆盖的知识点还远远不够，我希望这本书能覆盖更多的知识点，让更多的人受益。

本书内容

本书共分 13 章，各章内容简介如下。

- 第 1 章介绍系统源码的下载、编译、调试，带领读者走进 Android 系统源码的世界。
- 第 2 章介绍包管理机制和 PMS，PMS 是 Android 应用进阶所需要了解的三大 MS（Manage Service，管理服务）之一，三大 MS 分别为 AMS、WMS 和 PMS。

- 第 3 章介绍输入系统和 IMS，IMS 是除三大 MS 之外，应用开发人员所需要掌握的知识。
- 第 4 章介绍 IPC 机制，为讲解 Binder 原理进行铺垫。
- 第 5 章介绍 Native Binder 原理，掌握这一章的内容可以深入理解 Java Binder 原理。
- 第 6 章介绍 Java Binder 原理，包括 Binder 类的注册、JavaBBinder 原理、Java Binder 架构。
- 第 7 章介绍 Groovy 基础，为讲解 Gradle 进行铺垫。
- 第 8 章介绍 Gradle 核心思想，为讲解 Gradle 的 Android 插件进行铺垫。
- 第 9 章结合第 7 章和第 8 章的内容，讲解 Gradle 的 Android 插件。
- 第 10 章介绍 Android Jetpack 架构组件，包括 Lifecycle、LiveData、ViewModel 的使用方法和原理。
- 第 11 章介绍跨平台技术演进。
- 第 12 章介绍 Dart 基础，为讲解 Flutter 基础进行铺垫。
- 第 13 章介绍 Flutter 基础。

本书特色

（1）Android 进阶三部曲是目前市面上少有的成知识体系的源码分析类图书。

（2）本书的知识点自成体系并且环环相扣，但每一章节都或多或少与本书或者系列图书的其他章节有所关联。

（3）本书中的 Android 系统源码分析都是针对应用开发来编写的。

（4）本书是目前市面上少有的介绍 Android 新技术的图书。

读者对象

作为 Android 进阶三部曲的第三部，本书的深度要深于《Android 进阶之光》，本书适合以下读者阅读。

- 有一定基础的 Android 应用开发工程师。
- 对 Android 系统源码感兴趣的读者。

致谢

感谢本书的责任编辑付睿，她在 CSDN 博客上发现了我，并积极推动本书的出版进度，使得本书能够及时出版。感谢所有关注我的朋友们，你们的鼓励和认可为我写书及写博客带来了动力。

勘误与互动

本人虽已竭尽全力，但书中难免存在不足之处，欢迎大家向我反馈，我也会在独立博客和 CSDN 博客中定期发布本书的勘误信息。

本书互动地址

独立博客：http://liuwangshu.cn

CSDN 博客：https://liuwangshu.blog.csdn.net

GitHub：https://github.com/henrymorgen

【读者服务】

扫码回复：**39375**

- 获取博文视点学院 20 元付费内容抵扣券
- 获取免费增值资源
- 加入 Android 读者交流群，与更多读者互动
- 获取精选书单推荐

目录

Android 进阶三部曲知识体系 1

第 1 章　系统源码的下载、编译、调试 5

1.1　使用 VirtualBox 安装 Ubuntu 5

1.1.1　安装 Ubuntu 的步骤 6

1.1.2　安装增强工具 12

1.2　AOSP 源码下载 14

1.2.1　AOSP 简介 14

1.2.2　下载 repo 15

1.2.3　下载源码 15

1.3　内核源码下载 16

1.4　系统源码的整编和单编 18

1.4.1　编译系统概述 18

1.4.2　编译源码的方式 20

1.4.3　准备编译环境 20

1.4.4　源码整编 21

1.4.5　源码单编 24

1.5　Source Insight 导入系统源码 25

1.5.1　新建源码项目 25

1.5.2　定位文件 27

1.5.3 全局搜索 28
1.6 Android Studio 导入系统源码 29
1.7 Android Studio 调试系统源码 33
1.7.1 AOSP 源码项目调试 34
1.7.2 导入部分源码调试 38
1.7.3 新建 Android 项目调试 39
1.8 本章小结 40
第 2 章 理解包管理机制和 PMS 41
2.1 PackageInstaller 的初始化 41
2.1.1 PackageManager 简介 42
2.1.2 APK 的文件结构和安装方式 42
2.1.3 寻找 PackageInstaller 入口 43
2.1.4 PackageInstallerActivity 解析 47
2.1.5 PackageInstaller 初始化的过程 51
2.2 PackageInstaller 安装 APK 过程 52
2.2.1 PackageInstaller 中的处理 52
2.2.2 Java 框架层的处理 57
2.2.3 PackageInstaller 安装 APK 的过程 59
2.3 PMS 处理 APK 的安装过程 59
2.3.1 PackageHandler 处理安装消息 60
2.3.2 复制 APK 的处理 66
2.3.3 PMS 处理 APK 安装 70
2.3.4 PMS 处理 APK 安装的步骤 76
2.4 PMS 的创建过程 77
2.4.1 SystemServer 处理部分 77
2.4.2 PMS 构造方法 80
2.5 APK 解析过程 91
2.5.1 引入 PackageParser 91
2.5.2 PackageParser 解析 APK 92

2.5.3 Package 的数据结构 99
2.6 本章小结 100
第 3 章 理解输入系统和 IMS 102
3.1 输入事件传递流程的组成部分 103
3.2 IMS 的诞生 105
3.2.1 SystemServer 处理部分 105
3.2.2 InputManagerService 构造方法 107
3.3 IMS 的启动过程 110
3.4 InputDispatcher 的启动过程 112
3.5 InputReader 处理事件的过程 113
3.6 输入事件的处理总结 119
3.7 InputReader 的加工类型 120
3.8 InputDispatcher 的分发过程 123
3.8.1 唤醒 InputDispatcherThread 123
3.8.2 InputDispatcher 进行分发 124
3.9 事件分发到目标窗口的过程 128
3.9.1 为事件寻找合适的分发目标 128
3.9.2 处理点击形式的事件 132
3.9.3 向目标窗口发送事件 135
3.9.4 Motion 事件分发过程总结 136
3.10 本章小结 138
第 4 章 IPC 机制 139
4.1 Linux 和 Android 的 IPC 机制 139
4.1.1 Linux 中的 IPC 机制 139
4.1.2 Android 中的 IPC 机制 141
4.2 开启多进程 142
4.2.1 开启多进程举例 142
4.2.2 开启多进程引出的问题 144

4.3 用 Messenger 进行进程间通信 146
4.4 使用 AIDL 实现跨进程方法调用 150
4.4.1 创建 AIDL 文件 150
4.4.2 创建服务端 152
4.4.3 客户端调用 153
4.5 用 ContentProvider 进行进程间通信 155
4.5.1 建立数据库，方便 ContentProvider 使用 155
4.5.2 使用 ContentProvider 对数据库进行操作 156
4.5.3 在 Activity 中调用另一个进程的 GameProvider 方法 158
4.6 用 Socket 实现跨进程聊天程序 160
4.7 本章小结 167

第 5 章 Native Binder 原理 168

5.1 学习 Binder 的前置知识点 168
5.1.1 Linux 和 Binder 的 IPC 通信原理 169
5.1.2 使用 Binder 的原因 173
5.1.3 学习 Binder 的原因 174
5.2 ServiceManager 中的 Binder 机制 175
5.2.1 基于 Binder 通信的 C/S 架构 175
5.2.2 MediaServer 的 main 函数 176
5.2.3 每个进程唯一的 ProcessState 实例 177
5.2.4 解析 ServiceManager 中的 Binder 机制 180
5.2.5 小结 187
5.3 系统服务的注册过程 187
5.3.1 从调用链角度说明 MediaPlayerService 的注册过程 187
5.3.2 从进程角度说明 MediaPlayerService 的注册过程 195
5.3.3 小结 196
5.4 ServiceManager 的启动过程 196
5.4.1 打开 binder 设备 198
5.4.2 注册成为 Binder 机制的上下文管理者 201

5.4.3 循环等待和处理客户端发来的请求 203
5.5 系统服务的获取过程 206
5.5.1 客户端 MediaPlayerService 请求获取服务 206
5.5.2 服务端 ServiceManager 处理请求 212
5.6 本章小结 218
第 6 章 Java Binder 原理 219
6.1 Java Binder 的 JNI 注册 219
6.1.1 Binder 类的注册 223
6.1.2 BinderInternal 类的注册 224
6.2 Java Binder 中系统服务的注册过程 225
6.2.1 将 AMS 注册到 ServiceManager 226
6.2.2 引出 JavaBBinder 230
6.2.3 解析 JavaBBinder 232
6.2.4 Java Binder 架构 234
6.3 本章小结 235
第 7 章 Groovy 基础 236
7.1 Groovy 概述 236
7.2 Groovy 编写和调试 237
7.3 变量 237
7.4 方法 237
7.5 类 238
7.6 语句 239
7.6.1 断言 239
7.6.2 for 循环 239
7.6.3 switch 语句 240
7.7 数据类型 241
7.7.1 字符串 241
7.7.2 List 242

7.7.3 Map 242
7.7.4 闭包（Closure） 243
7.8 I/O 操作 244
7.8.1 文件读取 244
7.8.2 文件写入 244
7.9 其他 245
7.9.1 asType 245
7.9.2 判断是否为真 245
7.9.3 安全取值 245
7.9.4 with 操作符 246
7.10 本章小结 246
第 8 章 Gradle 核心思想 247
8.1 Gradle 概述 247
8.1.1 项目自动化 248
8.1.2 构建工具 248
8.1.3 Gradle 的特性 253
8.2 Gradle 基础知识 255
8.2.1 配置 Gradle 环境 255
8.2.2 实现 Hello World 257
8.2.3 Gradle 的任务 258
8.2.4 Gradle 日志级别 260
8.2.5 Gradle 命令行 261
8.3 被忽视的 Gradle Wrapper 263
8.3.1 为什么需要 Gradle Wrapper 263
8.3.2 构建 Gradle Wrapper 264
8.3.3 配置 Gradle Wrapper 265
8.3.4 使用 Gradle Wrapper 266
8.3.5 升级 Gradle Wrapper 267
8.3.6 自定义 Gradle Wrapper 268

8.4 Gradle 插件基础 268
8.4.1 Gradle 插件概述 268
8.4.2 应用 Gradle 插件 270
8.4.3 插件 DSL 272
8.4.4 自定义对象插件 273
8.4.5 Gradle 插件的作用和优点 274
8.5 自定义 Gradle 插件 274
8.5.1 在 build.gradle 中编写 275
8.5.2 在 buildSrc 工程项目中编写 277
8.5.3 在独立项目中编写 278
8.6 本章小结 281
第 9 章 Gradle 的 Android 插件 282
9.1 什么是 Gradle 的 Android 插件 282
9.2 Android Studio 的模块类型和项目视图 283
9.3 项目 build.gradle 284
9.4 模块 build.gradle 285
9.4.1 Gradle 的 Android 插件类型 286
9.4.2 Android 块 286
9.4.3 dependencies 块 291
9.5 Android 签名文件配置 292
9.5.1 自定义签名信息文件 292
9.5.2 本地添加签名信息文件 293
9.6 Gradle 的库依赖 294
9.7 Gradle 的库依赖管理 295
9.7.1 Gradle 的依赖传递 296
9.7.2 Gradle 的依赖检查 296
9.7.3 Gradle 的依赖冲突 299
9.8 本章小结 301

第 10 章　Android Jetpack 架构组件 302
10.1　Android Jetpack 简介 303
10.2　Android Jetpack 分类 303
10.2.1　Architecture 304
10.2.2　Foundationy 305
10.2.3　Behavior 306
10.2.4　UI 307
10.3　在应用中引入 Jetpack 相关组件 307
10.4　使用 Lifecycle 307
10.4.1　为什么需要 Lifecycle 307
10.4.2　如何使用 Lifecycle 309
10.4.3　Lifecycle 应用举例 312
10.4.4　自定义 LifecycleOwner 315
10.5　Lifecycle 的原理 316
10.5.1　Lifecycle 的生命周期状态事件和状态 316
10.5.2　Lifecycle 如何观察 Activity 和 Fragment 的生命周期 317
10.5.3　Lifecycle 关联类 327
10.6　使用 LiveData 328
10.6.1　什么是 LiveData 328
10.6.2　LiveData 的基本用法 329
10.6.3　更改 LiveData 中的数据 330
10.6.4　合并多个 LiveData 数据源 332
10.6.5　拓展 LiveData 对象 333
10.7　LiveData 的原理 335
10.7.1　LiveData 如何观察组件生命周期变化 335
10.7.2　LiveData 的 observe 方法回调 336
10.7.3　postValue/setValue 方法分析 339
10.7.4　Transformations.map 方法分析 341
10.7.5　LiveData 关联类 343

10.8 使用 ViewModel 344
10.8.1 什么是 ViewModel 345
10.8.2 ViewModel 的基本使用方法 345
10.8.3 ViewModel 的生命周期 347
10.9 ViewModel 的原理 348
10.10 本章小结 351
第 11 章 跨平台技术演进 352
11.1 Web App 353
11.2 Hybrid App 353
11.2.1 Cordova 354
11.2.2 Ionic 355
11.2.3 VasSonic 355
11.2.4 微信小程序 356
11.3 语言编译转换 357
11.4 原生渲染 358
11.4.1 React Native 358
11.4.2 Weex 358
11.4.3 快应用 359
11.5 自绘 UI 360
11.5.1 Qt 360
11.5.2 Flutter 361
11.6 本章小结 362
第 12 章 Dart 基础 363
12.1 Dart 开发环境搭建 363
12.2 Dart 概述 364
12.3 Dart 的特性和重要概念 365
12.4 Dart 关键字 366
12.5 变量 367

12.6 基本数据类型 367
12.6.1 Number 367
12.6.2 String 368
12.6.3 Boolean 368
12.6.4 List 368
12.6.5 Set 369
12.6.6 Map 369
12.7 函数 369
12.7.1 可选参数 370
12.7.2 默认参数值 370
12.7.3 main 函数 371
12.7.4 匿名函数 371
12.8 流程控制语句 371
12.8.1 for 循环 372
12.8.2 switch 和 case 372
12.9 捕获异常 373
12.10 为类添加新的功能 373
12.11 库的使用方法 374
12.12 异步支持 375
12.13 使类可调用 376
12.14 创建实例 376
12.15 本章小结 376

第 13 章 Flutter 基础 377

13.1 Flutter 开发环境搭建和 Hello World 377
13.1.1 Flutter 概述 378
13.1.2 Flutter 开发环境搭建 378
13.1.3 运行一个 Flutter Demo 383
13.1.4 Hello World 385
13.2 Widget 基础 386

13.2.1 Widget 概述 387
13.2.2 Widget 的分类 387
13.2.3 Widget 的状态分类 388
13.2.4 根 Widget 的种类 388
13.3 Basic Widget 389
13.3.1 代码模板和主题 389
13.3.2 文本 390
13.3.3 图片 391
13.3.4 凸起按钮 394
13.3.5 其他 Widget 395
13.4 Material 组件 397
13.4.1 MaterialApp 398
13.4.2 Scaffold 401
13.4.3 AppBar 403
13.4.4 BottomNavigationBar 405
13.4.5 TabBar 407
13.4.6 Drawer 411
13.5 Scrolling Widget 413
13.5.1 ListView 413
13.5.2 GridView 418
13.5.3 PageView 419
13.6 手势相关 Widget 421
13.6.1 指针事件 421
13.6.2 手势 422
13.6.3 使用 GestureDetector 423
13.6.4 使用 Dismissible 424
13.7 资源和图片 426
13.7.1 指定 asset 426
13.7.2 加载文本 427
13.7.3 加载图片 430

13.8 布局 Widget 433
13.8.1 线性布局 Widget 433
13.8.2 流式布局 Widget 434
13.8.3 层式布局 Widget 436
13.8.4 弹性布局 Widget 438
13.9 网络请求库 Dio 440
13.9.1 Dio 的使用入门 440
13.9.2 Dio 访问网络 441
13.10 JSON 数据解析 443
13.10.1 使用 json.decode 方法 443
13.10.2 手动编写实体类 444
13.10.3 自动生成实体类 445
13.11 路由与数据传递 447
13.11.1 新建路由 447
13.11.2 使用注册路由 450
13.11.3 路由间数据传递 451
13.12 Flutter 与 Android 的相互通信 456
13.12.1 PlatformChannel 概述 456
13.12.2 Flutter 调用 Android 458
13.12.3 Android 调用 Flutter 461
13.13 本章小结 465

Android 进阶三部曲知识体系

我从 2011 年开始写技术博客，从 2016 年开始，我写的文章都是系列文章，这些系列文章从点到面组合在一起成为一个知识体系。这个知识体系在国内来说是形成比较早的，并且受到了众多开发者的关注和好评，后面有很多人开始模仿，也算是带了个好头。

现在我也要通过书建立一个知识体系，这在应用开发领域是开创性的，这个知识体系通过一本书是无法实现的，因为应用开发涉及的知识点很多，所以我写了 Android 进阶三部曲：《Android 进阶之光》《Android 进阶解密》《Android 进阶指北》。通过 Android 进阶三部曲来实现知识体系。

1．Android 进阶三部曲知识体系

我将 Android 进阶三部曲知识体系分为三部分，分别是 Android 应用开发、Android 系统源码、Java 和跨平台基础，下面以思维导图的形式对每个部分进行了整理，其也可以作为对三本书内容的索引。

（1）Android 应用开发。

如图 1 所示，这一部分的内容在三本书中都有所涉及，毕竟 Android 进阶三部曲是写给应用开发的。这一部分除了应用实践，更注重每个知识点的原理。

（2）Android 系统源码。

如图 2 所示，这一部分的内容主要集中在《Android 进阶解密》和《Android 进阶指北》中，相关内容都是应用开发人员所需要掌握的。

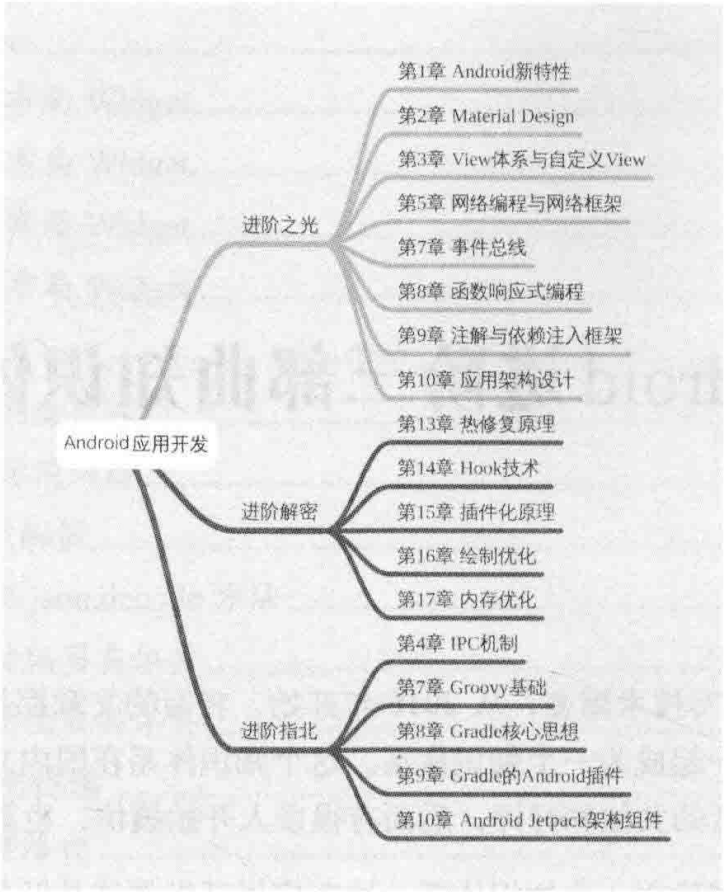


图 1　Android 应用开发部分

图 2　Android 系统源码部分

（3）Java 和跨平台基础。

如图 3 所示，这一部分的内容不多，按严格意义来说，这些内容不属于 Android 技术范畴，但都是应用开发需要掌握的内容。

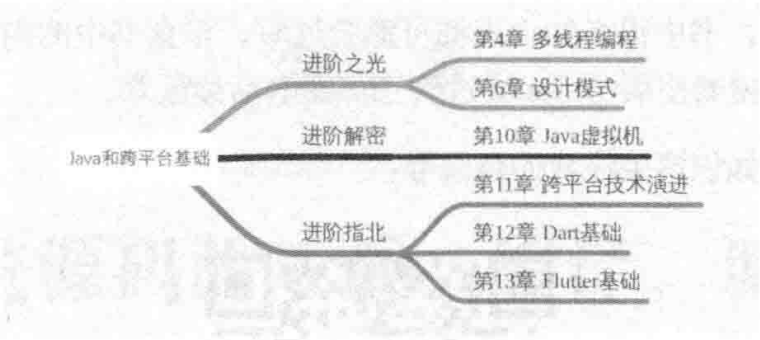


图 3　Java 和跨平台基础部分

2．Android 进阶三部曲的关联章节

在《Android 进阶解密》和《Android 进阶指北》中的大部分章节前都设有关联章节。比如本书的第 3 章“理解输入系统和 IMS”，它的关联章节见图 4。

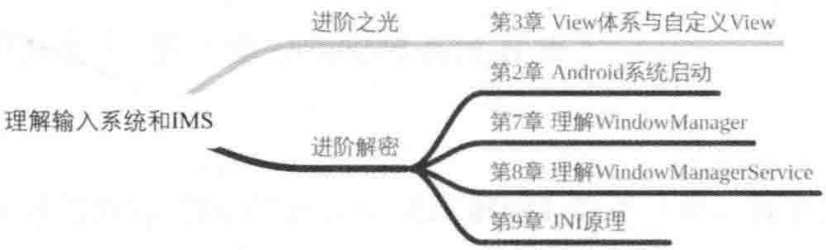


图 4　理解输入系统和 IMS 的关联章节

再比如本书的第 6 章“Java Binder 原理”，它的关联章节见图 5。

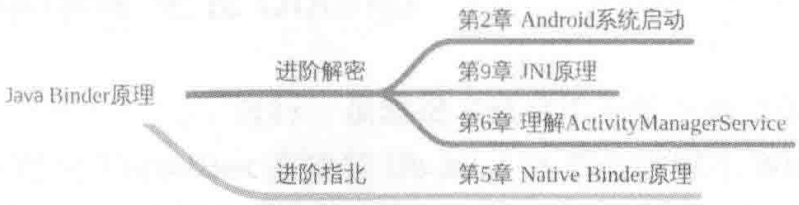


图 5　Java Binder 原理的关联章节

很多章节并不是孤立的，而是相互关联的，这些知识点由点到面组合在一起成为 Android 进阶三部曲知识体系。

3．Android 进阶三部曲的后续更新

随着技术的发展，书中很多知识点都可能会过时，毕竟书中的内容可能会过时，如果条件允许，那么我会接着出版 Android 进阶三部曲的后续版本。

想要获取更多的知识请关注我的公众号：

第 1 章

系统源码的下载、编译、调试

关联章节：

《Android 进阶解密》 第 1 章 Android 系统架构

《Android 进阶解密》 第 2 章 Android 系统启动

在 Android 进阶三部曲第二部《Android 进阶解密》的第 1 章，我介绍了两种阅读源码的方式，其中一种是从百度网盘将源码下载下来，然后用 SouceInsight 查看，这种方式很便捷，适合阅读源码，但是这种方式有两个弊端，一个是无法获取最新的源码，另一个是不能够编译源码。本章讲解的就是下载、编译、调试源码的相关知识。

1.1 使用 VirtualBox 安装 Ubuntu

编译源码需要在 Linux 环境下进行，虽然很多读者的系统不是 Ubuntu，但是可以在 Windows 和 macOS 上使用 VirtualBox 来安装 Ubuntu。本节会讲解在 Windows 10 环境下安装 Ubuntu 的过程，在 macOS 环境下大同小异。另外，在 Windows 10 环境下也可以在 Microsoft Store 中下载 Ubuntu 软件。

需要下载两个文件：

（1）VirtualBox，我的版本为 VirtualBox 6.0.4。

（2）Ubuntu 桌面版，我的版本为 Ubuntu 18.04.2 LTS 。

1.1.1 安装 Ubuntu 的步骤

VirtualBox 的安装过程很简单，一路单击下一步按钮即可。

（1）单击 VirtualBox 的新建按钮，新建虚拟电脑，如图 1-1 所示。

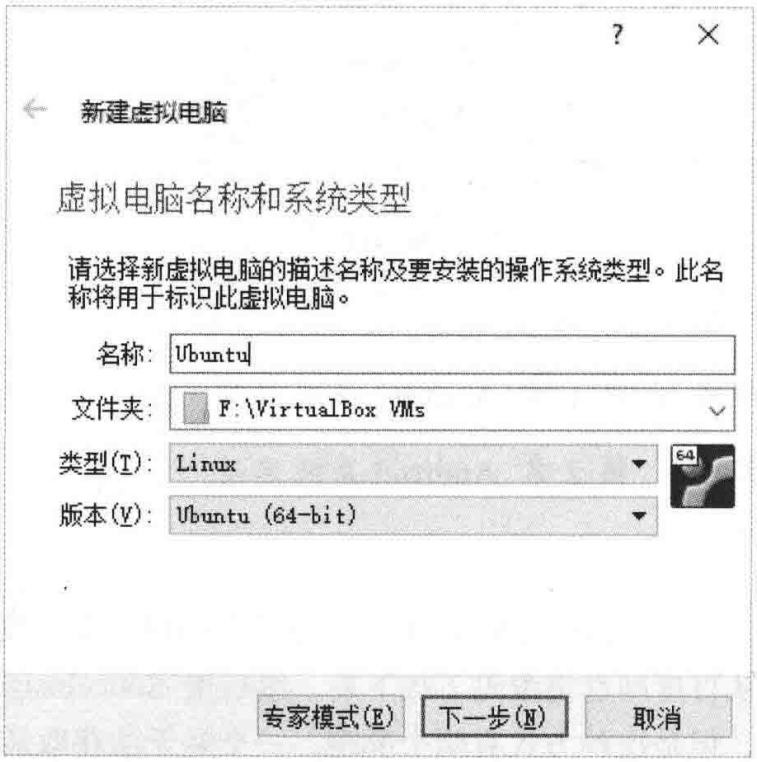


图 1-1　新建虚拟电脑

可能在版本选项中没有 64 位的选项，这时需要进入 BIOS，开启 CPU 虚拟化。不同的主板进入 BIOS 的方式不同（具体方式请自行查询），比如我的电脑是惠普台式机，在开机时连续按 F10 键就会进入 BIOS，依次选择安全→系统安全性→虚拟化技术并设置启用即可。

（2）设置分配给虚拟电脑的内存大小，建议的内存大小为 1024MB，也可以根据自己电脑的配置来决定，建议最少分配 4096MB，如图 1-2 所示。

（3）创建虚拟硬盘的大小，一路单击下一步按钮就可以了，如图 1-3、图 1-4、图 1-5 所示。

图 1-2　设置内存大小

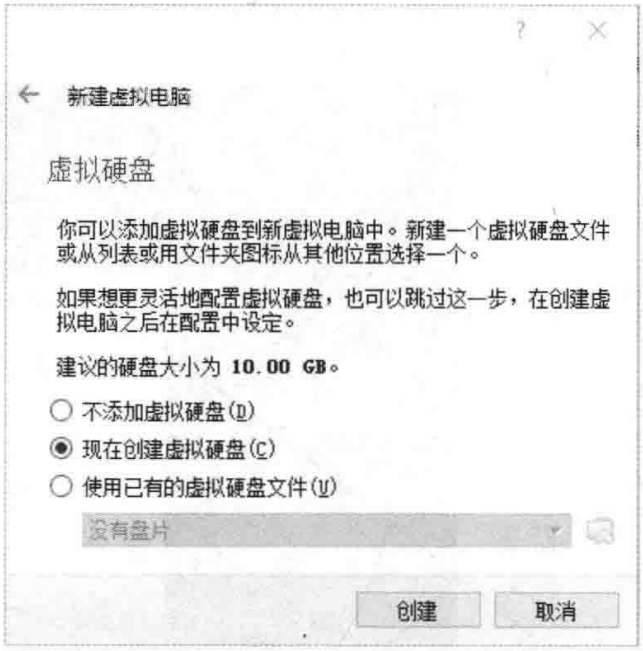


图 1-3　设置虚拟硬盘

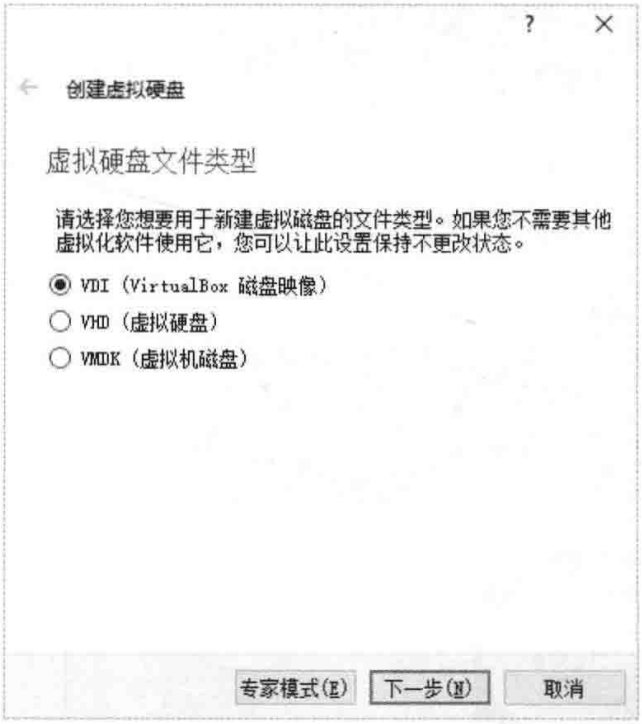


图 1-4　设置虚拟硬盘文件类型

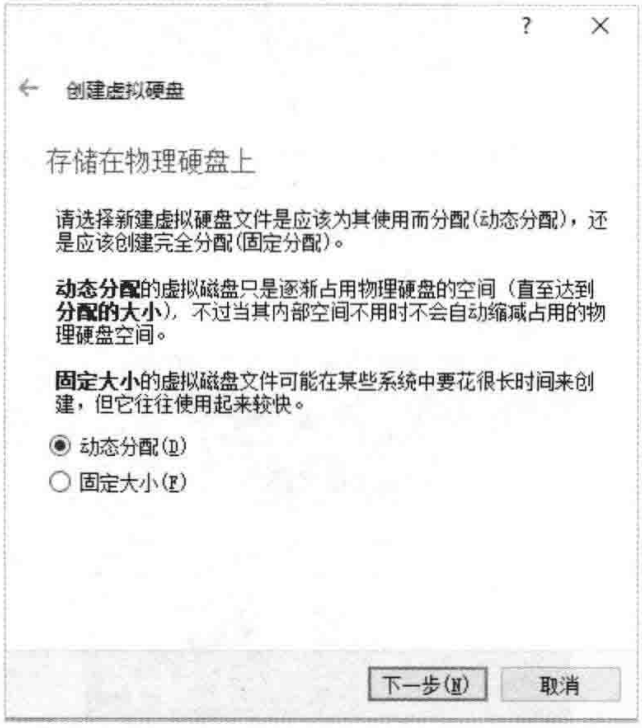


图 1-5　设置动态分配或固定大小

（4）虽然建议虚拟硬盘的大小为 10GB，但是如果要进行 Android 源码编译，则需要更大的硬盘空间，建议最少分配 200GB，我在这里分配了 200GB，如图 1-6 所示。

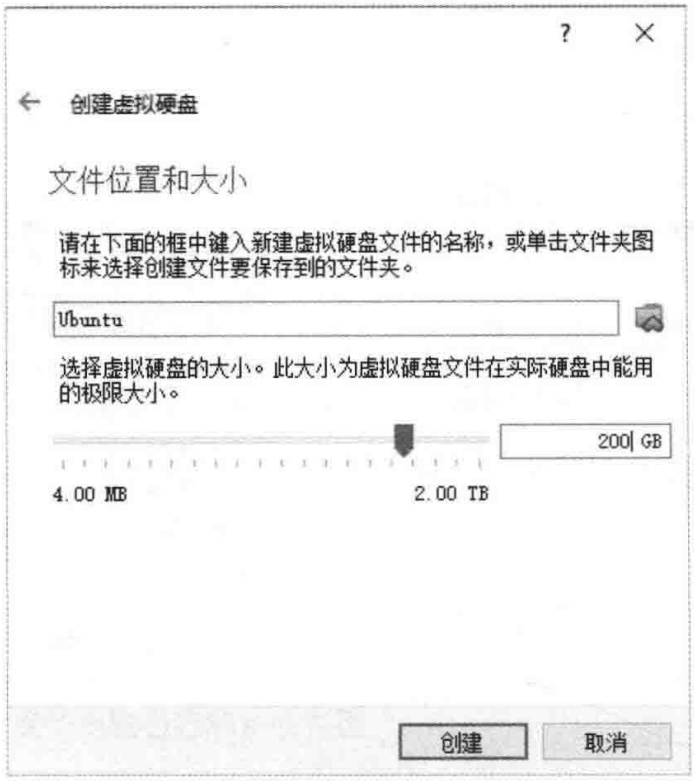


图 1-6　设置文件位置和大小

（5）选择 Ubuntu，单击启动按钮，如图 1-7 所示。

图 1-7　单击启动按钮

（6）选择已经下载的 Ubuntu 镜像文件，单击启动按钮，如图 1-8 所示。

图 1-8　选择 Ubuntu 镜像文件

（7）选择系统语言，单击安装 Ubuntu 按钮，如图 1-9 所示。

图 1-9　选择系统语言

（8）下面按照步骤进行操作就可以了，如图 1-10、图 1-11 所示。

图 1-10　设置键盘布局

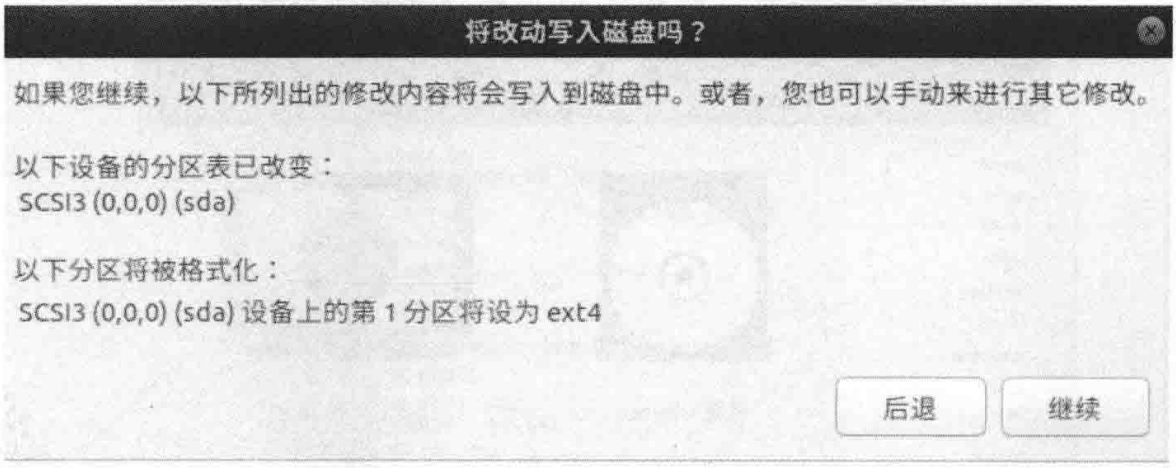


图 1-11　设置改动是否写入磁盘

（9）填入姓名、密码等信息，单击继续按钮，如图 1-12 所示。

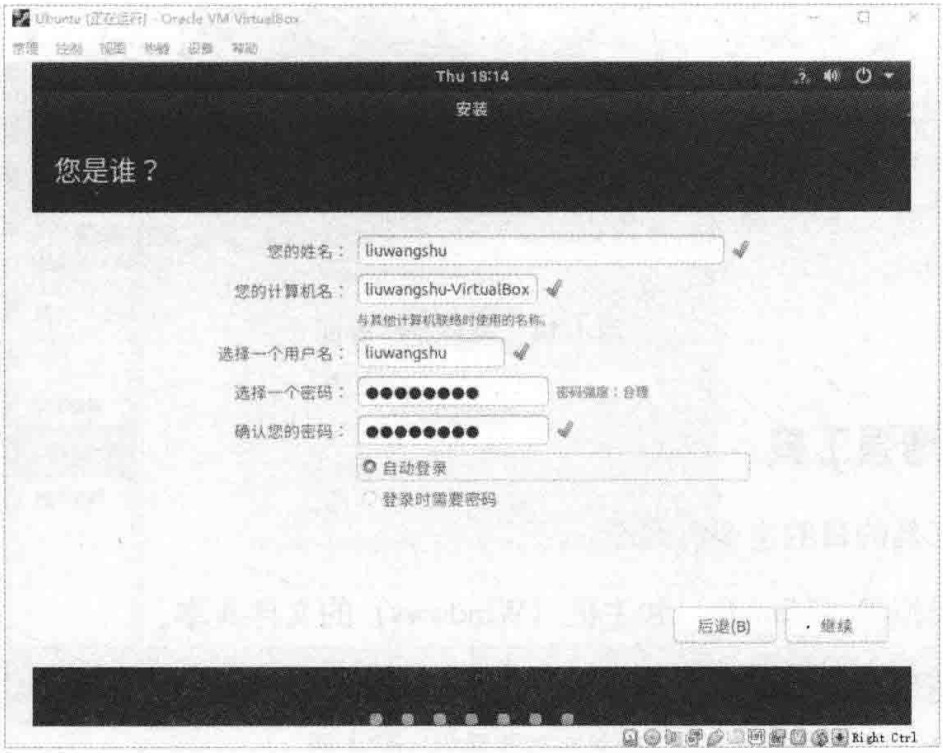


图 1-12　设置姓名和密码等信息

(10) 开始安装 Ubuntu，安装完成单击现在重启按钮就可以了，如图 1-13、图 1-14 所示。

图 1-13　Ubuntu 安装进度界面

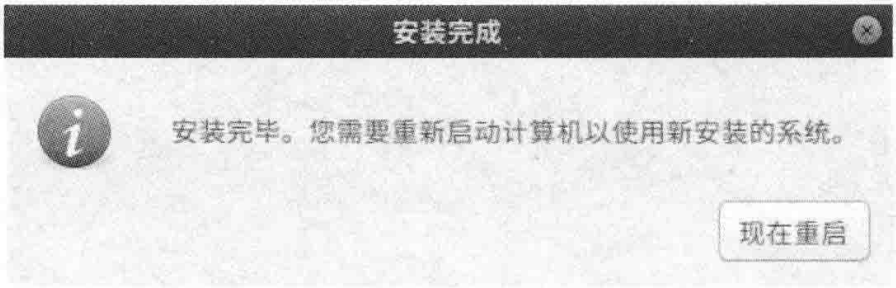


图 1-14　安装完毕界面

1.1.2　安装增强工具

安装增强工具的目的主要有两个：

（1）实现虚拟机（Ubuntu）和主机（Windows）的文件共享。

（2）可以使用全屏模式。

1. 配置 Ubuntu

在用户界面中勾选在屏幕顶部显示，如图 1-15 所示。

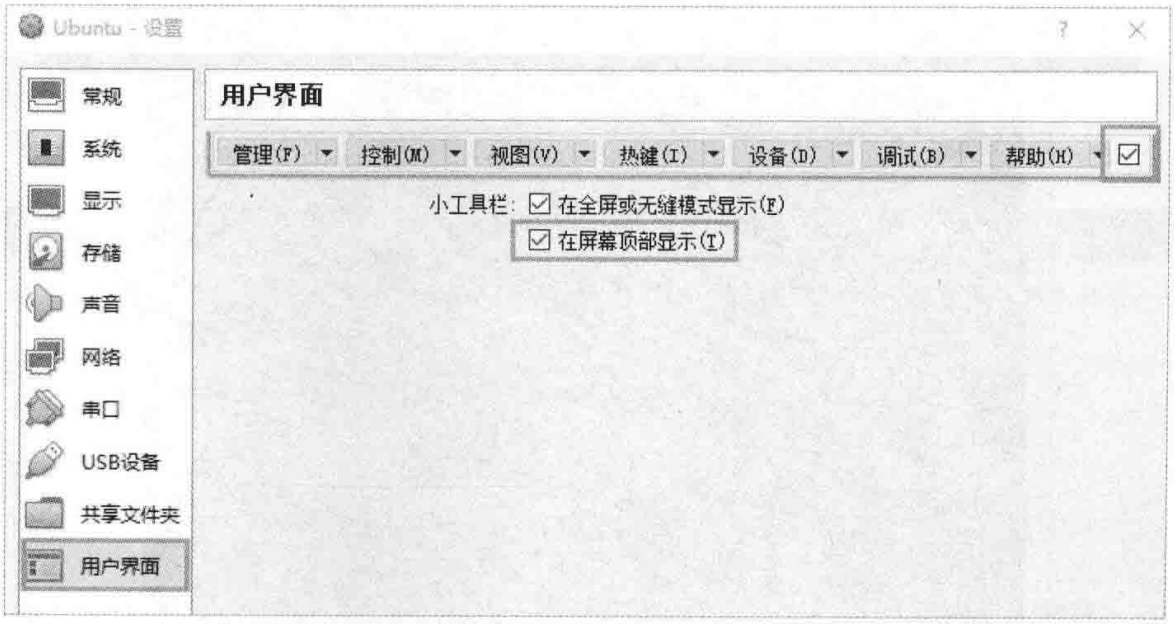


图 1-15　Ubuntu 设置界面

新建一个共享文件夹，勾选自动挂载和固定分配，如图 1-16 所示。

在模拟器屏幕顶部选择设备→安装增强功能，然后会在桌面生成一个 VBox_GAs_6.0.4 的盘符，接着进行配置。

图 1-16　设置共享文件夹

2. 安装增强工具

先安装 gcc 编译器和 perl 库：

```
sudo apt-get install gcc perl make
```

从桌面或者 Ubuntu 的主文件中进入 VBox_GAs_6.0.4 目录，单击运行软件按钮，安装增强工具，如图 1-17 所示。

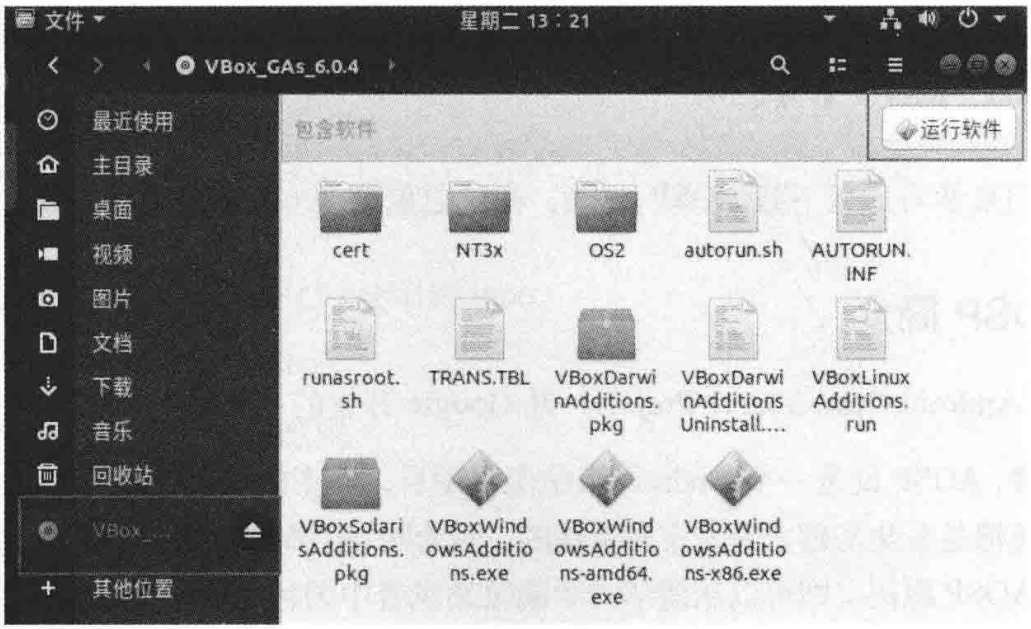


图 1-17　安装增强工具

出现如下提示就说明安装成功了，然后重启虚拟机，如图 1-18 所示。

```
Verifying archive integrity... All good.
Uncompressing VirtualBox 6.0.4 Guest Additions for Linux........
VirtualBox Guest Additions installer
Removing installed version 6.0.4 of VirtualBox Guest Additions...
update-initramfs: Generating /boot/initrd.img-4.18.0-15-generic
Copying additional installer modules ...
Installing additional modules ...
VirtualBox Guest Additions: Building the VirtualBox Guest Additions kernel
modules.  This may take a while.
VirtualBox Guest Additions: To build modules for other installed kernels, run
VirtualBox Guest Additions:   /sbin/rcvboxadd quicksetup <version>
VirtualBox Guest Additions: Building the modules for kernel 4.18.0-15-generic.
update-initramfs: Generating /boot/initrd.img-4.18.0-15-generic
VirtualBox Guest Additions: Running kernel modules will not be replaced until
the system is restarted
VirtualBox Guest Additions: Starting.
Press Return to close this window...
```

图 1-18　安装成功提示

这时在桌面上会生成一个盘符，通过它就可以和主机进行文件共享了，如图 1-19 所示。

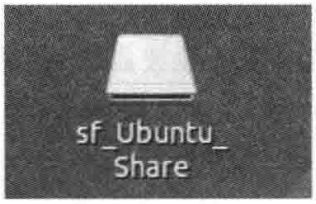


图 1-19　在桌面上生成盘符

除了安装增强工具，还有一个功能是很常用的，在设置→常规→高级中的共享粘贴板和拖放选项中选择双向，使虚拟机和主机的交互更加简单。

1.2　AOSP 源码下载

本节我们来学习如何下载 AOSP 源码，在此之前需要知道什么是 AOSP。

1.2.1　AOSP 简介

AOSP（Android Open Source Project）是 Google 开发的 Android 开源项目。

通俗来讲，AOSP 就是一个 Android 系统源码项目，通过它可以定制 Android 操作系统，国内手机厂商都是在此基础上开发定制系统的。因为网络限制的缘故，如果无法连接 Google 服务器获取 AOSP 源码，则可以从清华大学镜像站或者中国科学技术大学镜像站下载。下面以清华大学镜像站为例进行讲解。

1.2.2　下载 repo

Android 源码包含数百个 Git 库，下载这么多的 Git 库是一项繁重的任务，因此 Google 开发了 repo，它是一个用于管理 Android 版本库的工具，使用 Python 对 Git 进行了一定的封装，简化了对多个 Git 版本库的管理任务。

安装 Git，在 Ubuntu 中输入如下命令。

```
sudo apt-get install git
```

可能会报如下错误，如图 1-20 所示。

```
E: 无法获得锁 /var/lib/dpkg/lock-frontend - open (11: 资源暂时不可用)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), is a
nother process using it?
```

图 1-20　安装 repo 报错

出现这个问题的原因是另一个程序正在运行，导致资源被锁不可用，输入如下命令来解决。

```
sudo rm /var/cache/apt/archives/lock
sudo rm /var/lib/dpkg/lock
```

接下来创建 bin，并将其加入 PATH 中。

```
mkdir ~/bin
PATH=~/bin:$PATH
```

安装 curl 库。

```
sudo apt-get install curl
```

下载 repo 并设置权限。

```
curl https://mirrors.tuna.tsinghua…… > ~/bin/repo
chmod a+x ~/bin/repo
```

最后安装 Python，在初始化时会用到 repo。

```
sudo apt-get install python
```

重启虚拟机后，开始下载源码。

1.2.3　下载源码

建立工作目录。

```
mkdir aosp
cd aosp
```

repo 在运行过程中会尝试访问官方的 Git 源来更新自己，如果想使用 tuna 的镜像源进行更新，则可以将如下内容复制到~/.bashrc 中。

```
export REPO_URL='https://mirrors.tuna.tsinghua……'
```

设置身份，添加自己的邮箱和姓名。

```
git config --global user.email "piratemorgen@gmail.com"
git config --global user.name "piratemorgen"
```

初始化仓库。

```
repo init -u https://aosp.tuna.tsinghua……
```

初始化并指定版本。

```
repo init -u https://aosp.tuna.tsinghua……
```

同步源码。

```
repo sync
```

源码下载完成后，看到下面的信息就说明 AOSP 下载成功。整个源码的大小为 36.5GB，如图 1-21 所示。

```
正在检出文件: 100% (73/73), 完成.
正在检出文件: 100% (14/14), 完成.
正在检出文件: 100% (4084/4084), 完成.
正在检出文件: 100% (10157/10157), 完成.
正在检出文件: 100% (1150/1150), 完成.
正在检出文件: 100% (1699/1699), 完成.检出文件:  72% (1228/1699)
正在检出文件: 100% (891/891), 完成.在检出文件:  12% (111/891)
正在检出文件: 100% (67/67), 完成.正在检出文件:  68% (46/67)
正在检出文件: 100% (18572/18572), 完成.
正在检出文件: 100% (149/149), 完成.在检出文件:  69% (103/149)
正在检出文件: 100% (17/17), 完成.正在检出文件:  41% (7/17)
Syncing work tree: 100% (668/668), done.
```

图 1-21　AOSP 下载成功

1.3　内核源码下载

AOSP 源码中并不包括内核源码，需要单独下载，内核源码有很多版本，比如 common 是通用的 Linux 内核源码，msm 是用于使用高通 MSM 芯片的 Android 设备的源码，goldfish 是用于 Android 模拟器的内核源码，这里以 goldfish 为例进行讲解。

和下载 AOSP 源码一样，我们需要先建立工作目录。

```
mkdir kernel
cd kernel
```

使用清华大学镜像站，git 命令如下所示。

```
git clone https://aosp.tuna.tsinghua……
```

内核源码下载成功如图 1-22 所示。

```
liuwangshu@liuwangshu-VirtualBox:~/kernel$ git clone https://aosp.tuna.tsinghua.edu.cn/kernel/goldfish.git
正克隆到 'goldfish'...
remote: Counting objects: 6044583, done.
remote: Total 6044583 (delta 0), reused 0 (delta 0)
接收对象中: 100% (6044583/6044583), 1.29 GiB | 5.37 MiB/s, 完成.
处理 delta 中: 100% (5087755/5087755), 完成.
```

图 1-22　内核源码下载成功

这时在 kernel 目录中会生成一个 goldfish 文件夹，进入 goldfish 文件夹并使用 git 命令。

```
cd goldfish
git branch -a
```

这时界面上会列出有哪些内核的版本分支可以下载，如图 1-23 所示。

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-3.10
  remotes/origin/android-3.18
  remotes/origin/android-3.4
  remotes/origin/android-4.14
  remotes/origin/android-4.4
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.10
  remotes/origin/android-goldfish-3.10-k-dev
  remotes/origin/android-goldfish-3.10-l-mr1-dev
  remotes/origin/android-goldfish-3.10-m-dev
  remotes/origin/android-goldfish-3.10-n-dev
  remotes/origin/android-goldfish-3.18
  remotes/origin/android-goldfish-3.18-dev
  remotes/origin/android-goldfish-3.4
  remotes/origin/android-goldfish-3.4-l-mr1-dev
  remotes/origin/android-goldfish-4.14-dev
  remotes/origin/android-goldfish-4.14-dev.120914098
  remotes/origin/android-goldfish-4.14-dev.143174688
  remotes/origin/android-goldfish-4.14-dev.20190417
  remotes/origin/android-goldfish-4.14-dev.backup
  remotes/origin/android-goldfish-4.4-dev
  remotes/origin/android-goldfish-4.9-dev
  remotes/origin/b12
  remotes/origin/b120914098
  remotes/origin/heads/for/android-goldfish-3.18-dev
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master
  remotes/origin/test
  remotes/origin/tparker-recovery
```

图 1-23　内核的版本分支

在这里选择下载 goldfish 3.4 版本。

```
git checkout remotes/origin/android-goldfish-3.4
```

1.4 系统源码的整编和单编

我们顺利地将 AOSP 下载了下来，很多时候不仅需要查看源码，还有以下几个需求：

- 动态调试 Android 系统源码。
- 定制 Android 系统。
- 将最新版本的 Android 系统刷入自己的 Android 设备中。
- 将系统源码导入 Android Studio 中。

为了实现这些需求，我们需要编译系统源码。

1.4.1 编译系统概述

学习以下概念会帮助我们了解 Android 编译系统。

1. Makefile

Android 平台的编译系统，其实就是用 Makefile 写的一个独立项目。它定义了编译的规则，实现了“自动化编译”，不仅能把分散在数百个 Git 库中的代码整合起来进行统一编译，而且还能把产物分门别类地输出到一个目录中，打包成手机 ROM，还可以生成应用开发时所使用的 SDK、NDK 等。

因此，采用 Makefile 编写的编译系统，也可以称为 Makefile 编译系统。

2. Android.mk

Makefile 编译系统的一部分 Android.mk，定义了一个模块的必要参数，使模块随着平台编译。通俗来讲，就是告诉编译系统以什么样的规则来编译源码，并生成对应的目标文件。

3. Ninja

Ninja 是一个致力于提升速度的小型编译系统，如果把其他编译系统看作高级语言，那么 Ninja 的目标就是汇编语言。

4. Soong

Soong 是 Google 用来替代此前 Makefile 编译系统的替代品，负责解析 Android.bp 文件，并将之转换为 Ninja 文件。

5. Blueprint

Blueprint 用来解析 Android.bp 文件，并将其翻译为 Ninja 语法文件。

6. kati

kati 是 Google 专门为 Android 开发的一个基于 Golang 和 C++的小项目，其用处是将 Android 中的 Makefile 转换成 Ninja 文件。

7. Android.bp

Android.bp 是用来替换 Android.mk 的配置文件。

Android.mk、Ninja、Soong、Blueprint、kati、Android.bp 等概念之间的联系如图 1-24 所示。

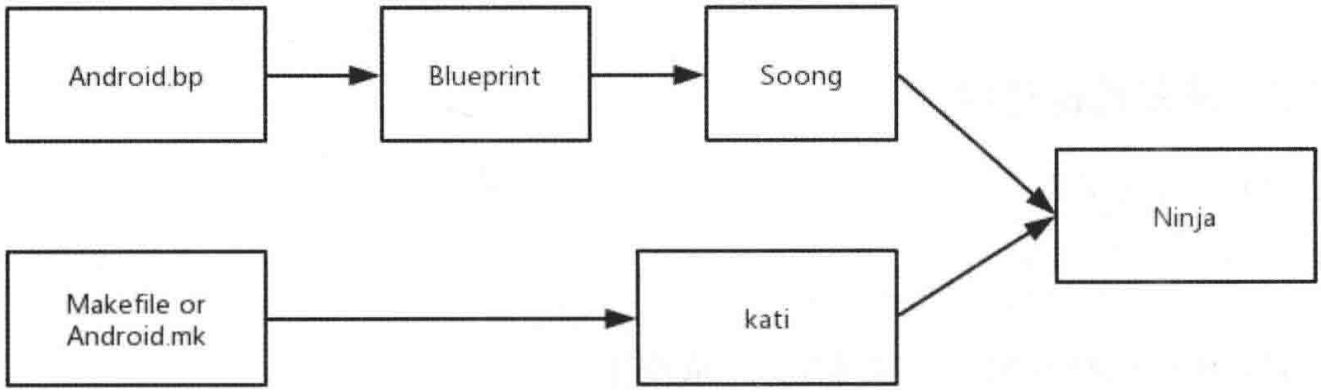


图 1-24　Android.mk、Ninja、Soong、Blueprint、kati、Android.bp 等概念之间的联系

Blueprint 负责解析 Android.bp 文件的内容，Blueprint 类似一个处理相关语法的库文件，Soong 则用于定义具体如何处理相应的语法及命令实现。通俗来讲，就是 Soong 借助于 Blueprint 定义的 Android.bp 语法，完成对 Android.bp 的解析，并最终将其转换成 Ninja 文件。Makefile 文件会通过 kati 转换为 Ninja 文件。

随着 Android 工程越来越大，采用 Makefile 编译系统花费的时间也越来越长，因此 Google 在 Android 7.0 开始引入了 Ninja 来编译系统。相对于 Makefile，在大的项目管理任务中，Ninja 在速度和并行性方面有突出的优势。

Makefile 默认的文件名为 Makefile 或 makefile，也常用.make 或.mk 作为文件后缀。Ninja 的默认文件名为 build.ninja，其他文件以.ninja 作为文件后缀。Makefile 与 Ninja 的区别在于，Makefile 是为开发编写的，而 Ninja 则用于生成其他程序。如果说 Makefile 是 Java 语言，那么 Ninja 就是汇编语言。

1.4.2 编译源码的方式

Android 系统源码编译有很多种方式，其中主要有以下几种方式：

- 在 Linux 中直接进行系统源码编译（Android 官方支持）。
- 在 macOS 中直接进行系统源码编译（Android 官方支持）。
- 使用 Docker 编译，支持 macOS 和 Windows。

其中需要注意的是，Docker 支持的最低版本为 Windows 7，建议在 Windows 10 环境下使用 Docker，因为在 Windows 7 环境下还需要借助 Docker Toolbox 和 VirtualBox 中的容器进行通信，效率相对低些。

考虑到大多数人的设备和上手难易程度，下面讲解如何在 Linux 中直接进行系统源码编译。

1.4.3 准备编译环境

（1）安装 JDK 8。

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk
```

（2）使用 Ubuntu 14+，需要安装以下依赖包。

```
sudo apt-get install git-core gnupg flex bison gperf build-essential zip curl
zlib1g-dev gcc-multilib g++-multilib libc6-dev-i386 lib32ncurses5-dev x11proto-core-
dev libx11-dev lib32z-dev ccache libgl1-mesa-dev libxml2-utils xsltproc unzip
```

（3）设置处理器数量。

依次选择设置→系统→处理器，设置处理器数量，建议选择能设置的最大值，这里设置的值是 6。

1.4.4　源码整编

整编就是编译整个 Android 源码，整编主要有三个步骤，下面分别对其进行介绍。

1. 初始化环境

在 AOSP 的根目录下输入如下命令。

```
source build/envsetup.sh
// 编译前删除 build 文件夹
make clobber
```

使用 build 目录中的 envsetup.sh 脚本初始化环境，这个脚本会引入其他执行脚本。

2. 选择编译目标

输入如下命令。

```
lunch
```

lunch 命令是 envsetup.sh 中定义的命令，用来让用户选择编译目标。

输入 lunch 命令，会输出以下信息。

```
You're building on Linux

Lunch menu... pick a combo:
    1. aosp_arm-eng
    2. aosp_arm64-eng
    3. aosp_mips-eng
    4. aosp_mips64-eng
    5. aosp_x86-eng
    6. aosp_x86_64-eng
    7. aosp_car_arm-userdebug
    8. aosp_car_arm64-userdebug
    9. aosp_car_x86-userdebug
    10. aosp_car_x86_64-userdebug
    11. mini_emulator_arm64-userdebug
    12. m_e_arm-userdebug
    13. m_e_mips64-eng
    14. m_e_mips-userdebug
    15. mini_emulator_x86_64-userdebug
    16. mini_emulator_x86-userdebug
    17. uml-userdebug
    18. aosp_cf_x86_auto-userdebug
    19. aosp_cf_x86_phone-userdebug
```

```
20. aosp_cf_x86_tablet-userdebug
21. aosp_cf_x86_tablet_3g-userdebug
22. aosp_cf_x86_tv-userdebug
23. aosp_cf_x86_wear-userdebug
24. aosp_cf_x86_64_auto-userdebug
25. aosp_cf_x86_64_phone-userdebug
26. aosp_cf_x86_64_tablet-userdebug
27. aosp_cf_x86_64_tablet_3g-userdebug
28. aosp_cf_x86_64_tv-userdebug
29. aosp_cf_x86_64_wear-userdebug
30. cf_x86_auto-userdebug
31. cf_x86_phone-userdebug
32. cf_x86_tablet-userdebug
33. cf_x86_tablet_3g-userdebug
34. cf_x86_tv-userdebug
35. cf_x86_wear-userdebug
36. cf_x86_64_auto-userdebug
37. cf_x86_64_phone-userdebug
38. cf_x86_64_tablet-userdebug
39. cf_x86_64_tablet_3g-userdebug
40. cf_x86_64_tv-userdebug
41. cf_x86_64_wear-userdebug
42. aosp_marlin-userdebug
43. aosp_marlin_svelte-userdebug
44. aosp_sailfish-userdebug
45. aosp_walleye-userdebug
46. aosp_walleye_test-userdebug
47. aosp_taimen-userdebug
48. hikey-userdebug
49. hikey64_only-userdebug
50. hikey960-userdebug

Which would you like? [aosp_arm-eng]
```

其目的是选择编译目标的格式，编译目标的格式组成为 BUILD-BUILDTYPE，比如 aosp_arm-eng 的 BUILD 为 aosp_arm，BUILDTYPE 为 eng。

其中 BUILD 表示编译出的镜像可以在什么环境下运行，aosp 代表 Android 开源项目，arm 表示系统运行在 ARM 架构的处理器上。

BUILDTYPE 指的是编译类型，其类型有以下三种：

- user：是正式发布到市场上的版本，其权限受限，如果没有 root 权限，则不能调试，adb 默认处于停用状态。

- userdebug：在 user 版本的基础上开放了 root 权限和 debug 权限，adb 默认处于启用状态。一般用于调试真机。
- eng：开发工程师使用的版本，拥有最大权限（root 权限等），具有额外调试工具的开发配置。一般用于模拟器。

如果没有 Nexus 设备，只想编译源码后在模拟器运行并查看，那么 BUILD 可以选择 aosp_x86，BUILDTYPE 可以选择 eng，在 Which would you like? [aosp_arm-eng]后面直接输入对应序号 5。

也可以直接指定编译的目标：

```
lunch aosp_x86-eng
```

或者（不同的系统版本，对应的序号会不同，不建议直接用序号）

```
lunch 5
```

3. 开始编译

通过-j*N* 参数来设置编译的并行任务数，以提高编译速度，此前我的 CPU 核心数为 6，这里 *N* 值最好选在 6 ～12 之间，我们设置 6 个并行任务并进行编译。

```
make -j6
```

整编成功后会打印如图 1-25 所示的内容。

```
The new table will be used at the next reboot.
The operation has completed successfully.
Setting name!
partNum is 0
REALLY setting name!
Warning: The kernel is still using the old partition table.
The new table will be used at the next reboot.
The operation has completed successfully.
```

图 1-25　整编成功

由于采用的是虚拟机整编，因此编译速度会慢一些，这里花费了 2 小时 46 分钟。

最终在 out/target/product/generic_x86/目录下生成了三个重要的镜像文件：system.img、userdata.img、ramdisk.img。下面大概介绍一下这三个镜像文件。

- system.img：系统镜像，其中包含了 Android 系统的主要目录和文件。
- userdata.img：用户镜像，是 Android 系统中用来存放用户数据的地方。

- ramdisk.img：根文件系统镜像，包含一些启动 Android 系统的重要文件，比如 init.rc。

在编译完成之后运行模拟器，就可以通过以下命令运行 Android 虚拟机了，命令如下。

```
source build/envsetup.sh
lunch 5
emulator
```

如果在编译完成后运行 Android 虚拟机，由于之前已经执行过 source 和 lunch 命令了，则可以直接运行。

```
emulator
```

如果不报错就会启动模拟器，如果报错，则可以上网查询获取解决方案。

1.4.5 源码单编

我们要编译系统的 Settings 应用模块，在 AOSP 根目录中执行。

```
source build/envsetup.sh
lunch 5
```

进入 Settings 的目录。

```
cdpackages/apps/Settings
```

mm 编译当前目录下的模块，不编译依赖模块。mm 编译成功后系统会提示生成文件的存放路径，如图 1-26 所示。

```
Preparing output jar [/home/liuwangshu/aosp/out/target/common/obj/APPS/Settings_
intermediates/classes-proguard.jar]
  Copying resources from program jar [/home/liuwangshu/aosp/out/target/common/ob
j/APPS/Settings_intermediates/classes.jar]
[100% 692/692] Copy: out/target/produc...riv-app/Settings/oat/x86/Settings.odex
```

图 1-26　单编成功

除了 Settings.odex 文件，还会在源码目录 out/target/product/generic_x86/system/priv-app/Settings 下生成 Settings.apk。

此外还有以下命令可以进行单编。

- mmm：编译指定目录下的模块，不编译它所依赖的其他模块。
- mma：编译当前目录下的模块及其依赖项。

- mmma：编译指定路径下的所有模块，并且包含其依赖项。

如果修改了源码，想查看生成的 apk 文件，则有以下两种方式。

（1）通过 adb push 或者 adb install 来安装 APK。

（2）使用 make snod 命令，重新生成 system.img，然后运行模拟器。

1.5　Source Insight 导入系统源码

在前面的章节中，我们对系统源码进行了编译，本节接着来学习如何将系统源码导入编辑器中，以便查看和调试源码。查看源码可以使用 Android Studio、Eclipse、Sublime、Source Insight 等软件，这里我推荐使用 Android Studio。由于 Source Insight 是 Windows 平台的软件，macOS 平台不能使用，所以使用 Android Studio 是一个不错的选择，而且使用 Android Studio 还可以调试源码，我们先来学习 Source Insight 导入系统源码。

在《Android 进阶解密》的第 1 章中，我讲解了 Source Insight 如何导入系统源码，可能有的读者没有这本书，这里再来简单讲解一下。

Source Insight 只能查看源码，不能调试源码，如果只想在 Source Insight 中查看源码，那么可以直接从百度网盘将源码下载下来。如果想在 Android Studio 中查看源码，那么最好还是在 Linux 环境下将 AOSP 源码下载下来。

1.5.1　新建源码项目

安装软件后，首先要新建源码项目。选择菜单项 Project→New Project 会弹出提示框，如图 1-27 所示。

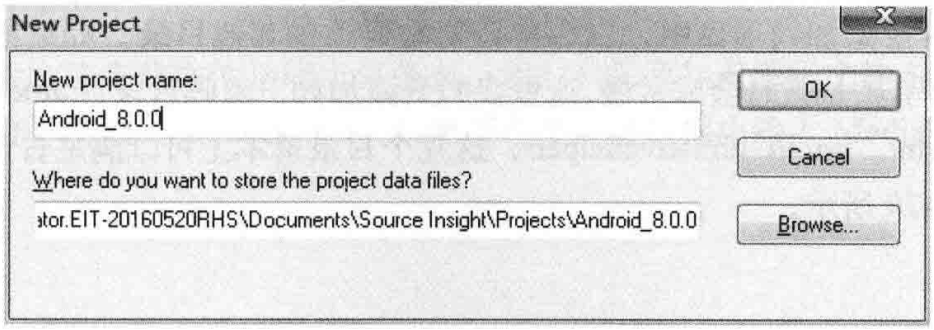


图 1-27　新建源码项目

这里我们指定源码项目的名称为 Android_8.0.0，然后单击 OK 按钮进入 New Project Settings 界面，如图 1-28 所示。

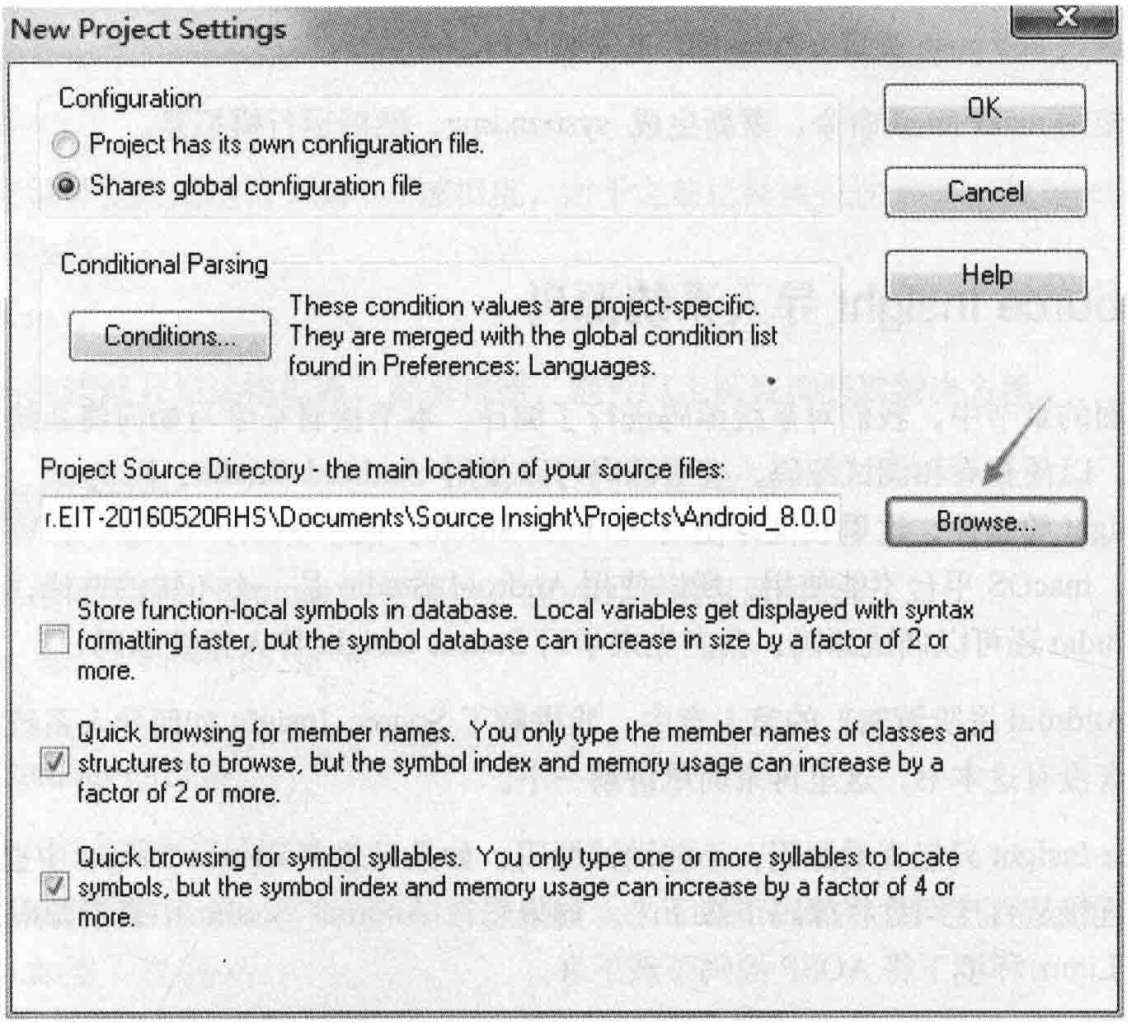


图 1-28　指定源码项目的名称

单击图 1-28 中箭头指向的 Browse 按钮来选择本地系统源码所在的路径，比如我的系统源码路径为 D:/Android/android-8.0.0r1。选择加载路径后单击 OK 按钮，会进入 Add and Remove Project Files 界面，在这个界面中可以给项目添加整个 Android 系统源码，也可以只把源码部分目录添加到项目中，以后再根据需要添加其他目录。如果给项目添加整个 Android 系统源码，则加载时会非常慢，这里我们只添加如下源码目录：frameworks/、libcore/、packages/、system/、art/和 libnativehelper/，这几个目录基本上可以满足日常的系统源码阅读需求，如图 1-29 所示。

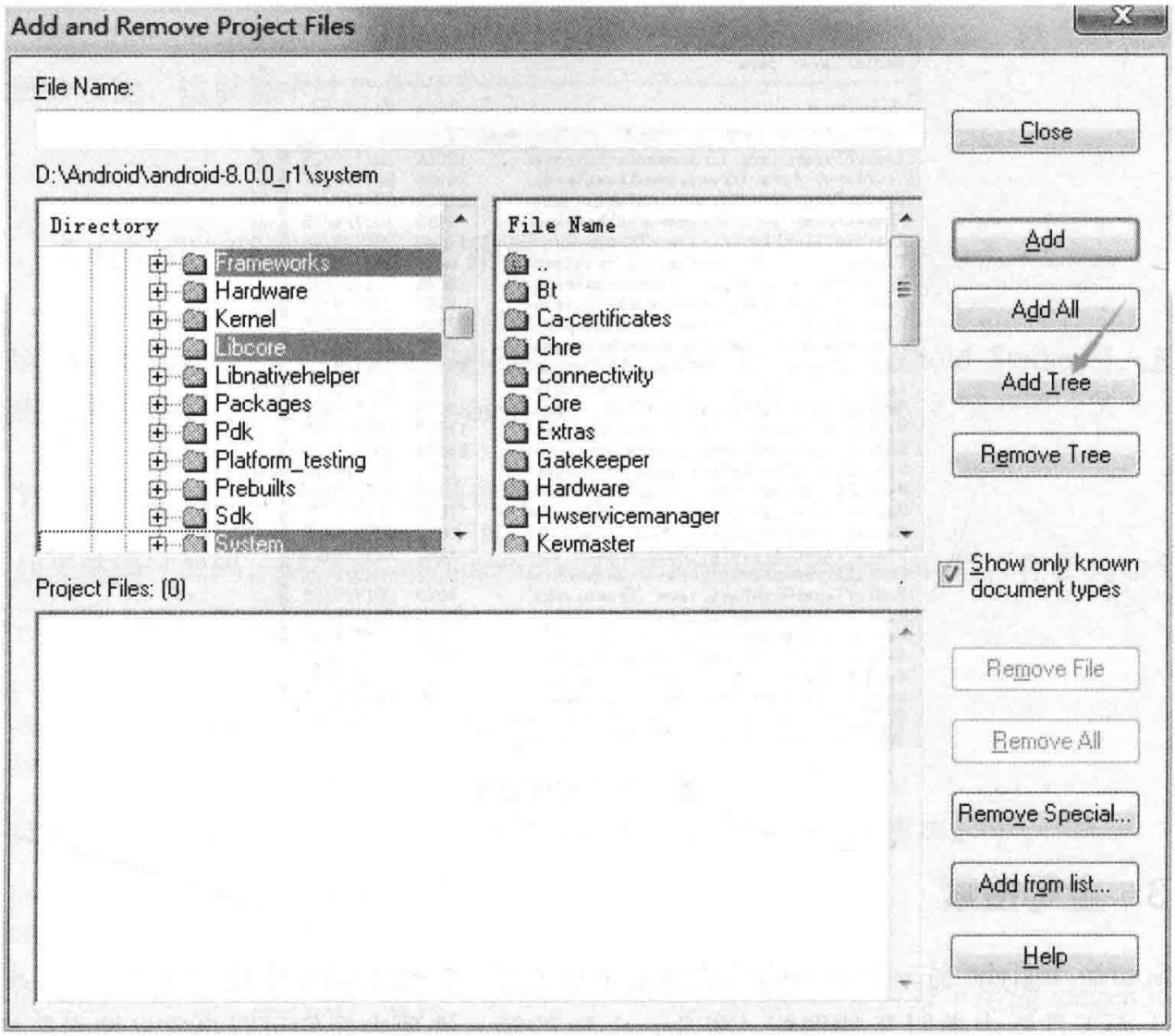


图 1-29　选择导入的源码目录

单击 Add Tree 按钮就会将选择的目录源码加载到 Android_8.0.0 项目中，这时候会弹出加载进度条，加载完毕后单击窗口的 Close 按钮就可以了。

1.5.2　定位文件

Source Insight 的定位文件功能十分强大，我们只需要知道源码文件名就可以轻松找到文件，比如我们要找 MediaPlayer.java，只需要在文件搜索框中输入 MediaPlayer.java 即可，如图 1-30 所示。

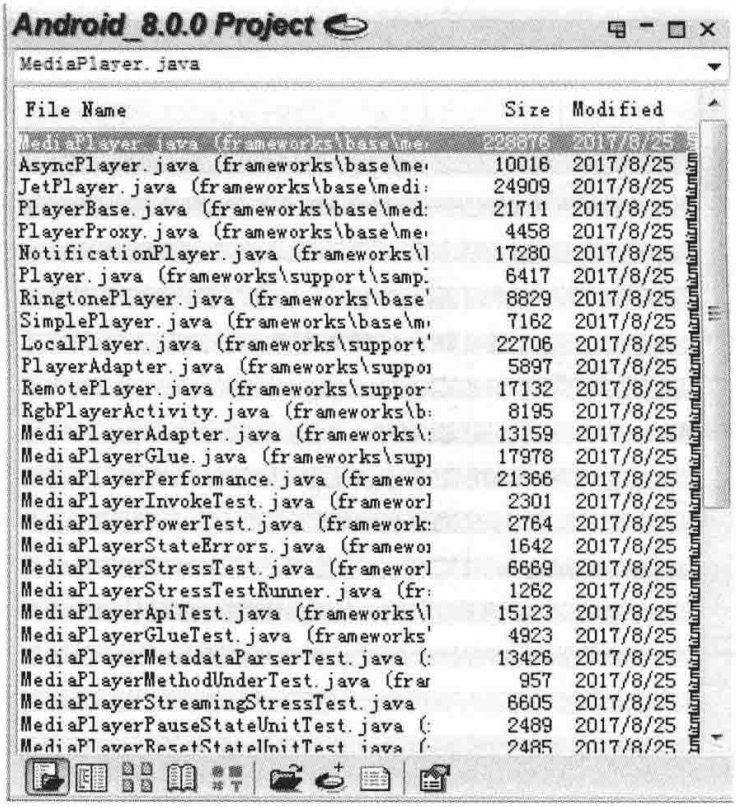


图 1-30　定位文件

1.5.3　全局搜索

Source Insight 另一个好用的功能就是全局搜索，默认的功能快捷键为 Ctrl+/，或者单击最上面工具栏中类似 R 的图标。在 Search In 的输入选项中我们可以自定义搜索范围，比如想查找所有 Java 文件中引用 MediaPlayer 类的情况，就可以像图 1-31 一样进行操作。

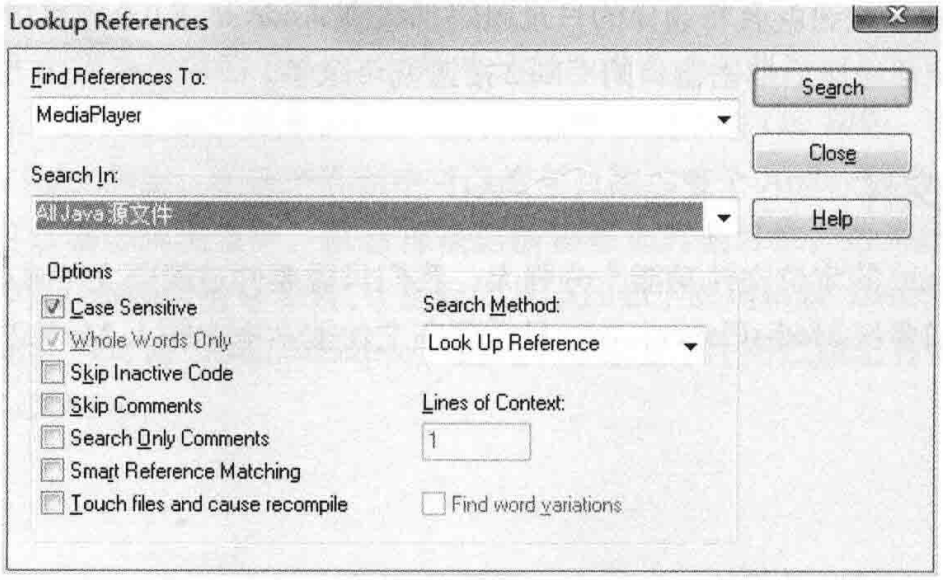


图 1-31　全局搜索

当然，Source Insight 的功能不只有以上几种，随着使用次数的增多，就会熟练掌握它的大部分功能，这里就不过多介绍了。

1.6　Android Studio 导入系统源码

Source Insight 导入系统源码不需要对源码进行编译，但是 Android Studio 导入整个系统源码需要对源码进行编译，生成 Android Studio 的项目配置文件。

1．生成 Android Studio 的项目配置文件

如果整编过源码，则查看 out/host/linux-x86/framework/idegen.jar 是否存在，如果其不存在，则进入源码根目录执行如下命令。

```
source build/envsetup.sh
lunch [选择整编时选择的参数或者数字]
mmm development/tools/idegen/
```

如果没有整编过源码，则可以直接执行如下命令单编 idegen 模块。

```
source build/ensetup.sh
make idegen
```

成功编译 idegen 模块后，会在 out/host/linux-x86/framework 目录下生成 idegen.jar，执行如下命令。

```
sudo development/tools/idegen/idegen.sh
```

这时会在源码根目录中生成 android.iml 和 android.ipr 两个文件，这两个文件一般是只读模式的，这里建议改成可读可写的，否则，在更改一些项目配置时可能会出现无法保存的情况。

```
sudo chmod 777 android.iml
sudo chmod 777 android.ipr
```

2．配置 Android Studio 的项目配置文件

将所有源码导入 Android Studio 会导致第一次加载很慢，可以在 android.iml 中修改 excludeFolder 配置，将不需要查看的源码排除掉。等源码项目加载完成后，还可以通过 Android Studio 对 Exclude 的 Module 进行调整。如果你的电脑性能很好，则可以不用进行配置。

在 android.iml 中搜索 excludeFolder，在 excludeFolder 下面加入这些配置。

```
<excludeFolder url="file://$MODULE_DIR$/bionic" />
<excludeFolder url="file://$MODULE_DIR$/bootable" />
<excludeFolder url="file://$MODULE_DIR$/build" />
<excludeFolder url="file://$MODULE_DIR$/cts" />
<excludeFolder url="file://$MODULE_DIR$/dalvik" />
<excludeFolder url="file://$MODULE_DIR$/developers" />
<excludeFolder url="file://$MODULE_DIR$/development" />
<excludeFolder url="file://$MODULE_DIR$/device" />
<excludeFolder url="file://$MODULE_DIR$/docs" />
<excludeFolder url="file://$MODULE_DIR$/external" />
<excludeFolder url="file://$MODULE_DIR$/hardware" />
<excludeFolder url="file://$MODULE_DIR$/kernel" />
<excludeFolder url="file://$MODULE_DIR$/out" />
<excludeFolder url="file://$MODULE_DIR$/pdk" />
<excludeFolder url="file://$MODULE_DIR$/platform_testing" />
<excludeFolder url="file://$MODULE_DIR$/prebuilts" />
<excludeFolder url="file://$MODULE_DIR$/sdk" />
<excludeFolder url="file://$MODULE_DIR$/system" />
<excludeFolder url="file://$MODULE_DIR$/test" />
<excludeFolder url="file://$MODULE_DIR$/toolchain" />
<excludeFolder url="file://$MODULE_DIR$/tools" />
<excludeFolder url="file://$MODULE_DIR$/.repo" />
```

3. 将系统源码导入 Android Studio 中

在 Android Studio 安装目录的 bin 目录下，打开 studio64.vmoptions 文件，根据自己电脑的实际情况进行设置，这里将数值修改为如下值。

```
-Xms1024m
-Xmx1024m
```

如果是在 VirtualBox 中下载系统源码的，那么可以将 VirtualBox 中的系统源码复制到共享文件夹中，这样源码就会自动添加到 Windows 或者 macOS 上。通过 Android Studio 的 Open an existing Android Studio project 选项选择 android.ipr 就可以导入源码，这里我用了大概 7 分钟就完成导入工作。导入源码后将工程目录切换为 Project 选项，这样就可以查看源码了，如图 1-32 所示。

图 1-32　打开 android.ipr

4．配置项目的 JDK、SDK

由于我们下载的是 9.0 版本的 AOSP 源码，SDK 版本也应该是对应的 API 28，如果没有对应的 API 28 文件就去 SDK Manager 下载即可。

依次选择 File→Project Structure→SDKs 来配置项目的 JDK、SDK。

创建一个新的 JDK，将它取名为 1.8(No Libraries)，删除其中 Classpath 选项卡下面的所有 jar 文件，如图 1-33 所示。

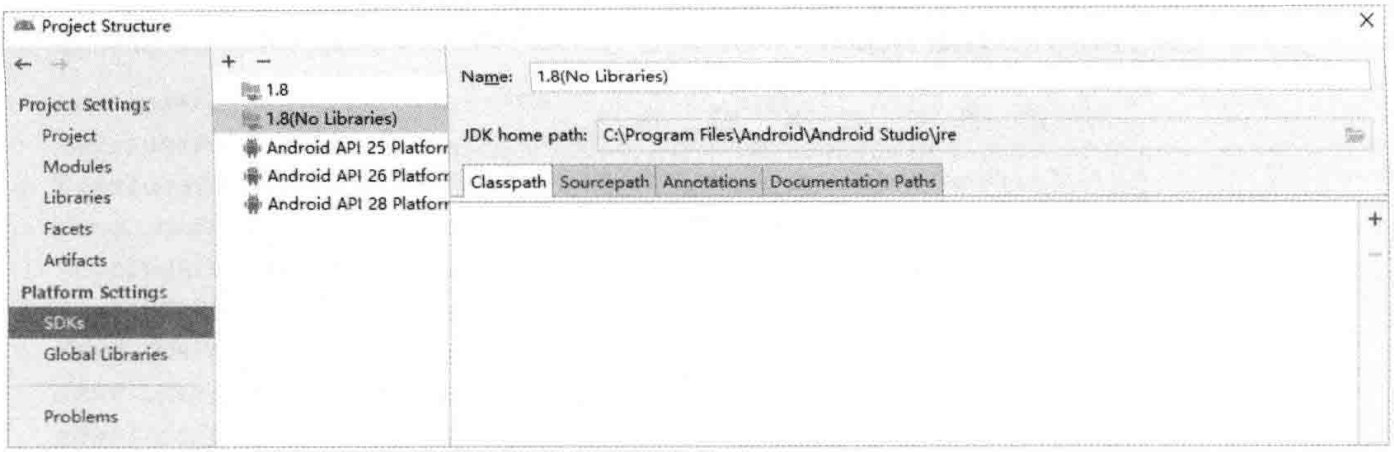


图 1-33　新建 JDK

然后将 Android SDK 的 Java SDK 设置为 1.8(No Libraries)，这样 Android 源码使用的 Java API 就是 Android 源码中的 Java API，如图 1-34 所示。

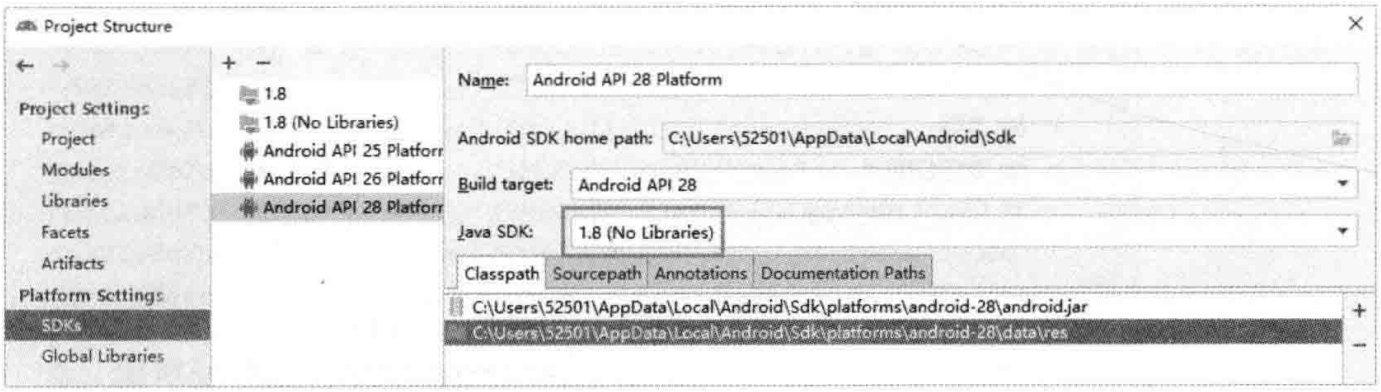


图 1-34　设置 Java SDK

确保项目的 SDK 为 Android 源码对应的 SDK，如图 1-35 所示。

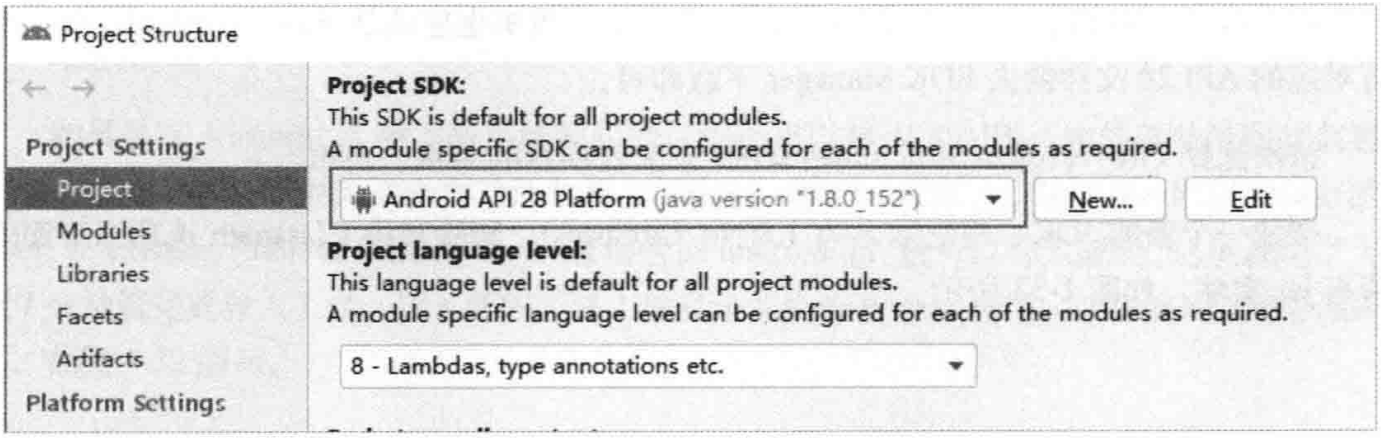


图 1-35　项目的 SDK 为 Android 源码对应的 SDK

5. 排除不需要的代码目录

在 File→Project Structure→Modules 中通过 Excluded 来筛选代码目录，比如我们选择 bionic 目录，单击 Excluded 按钮，bionic 目录会变为橙色，bionic 字段会出现在右侧视图中，说明该目录已经被排除了，通俗来讲就是被排除在工程之外了。如果不希望 bionic 目录被排除，则再次单击 Excluded 按钮，这时 bionic 目录会变为灰色，如图 1-36 所示。

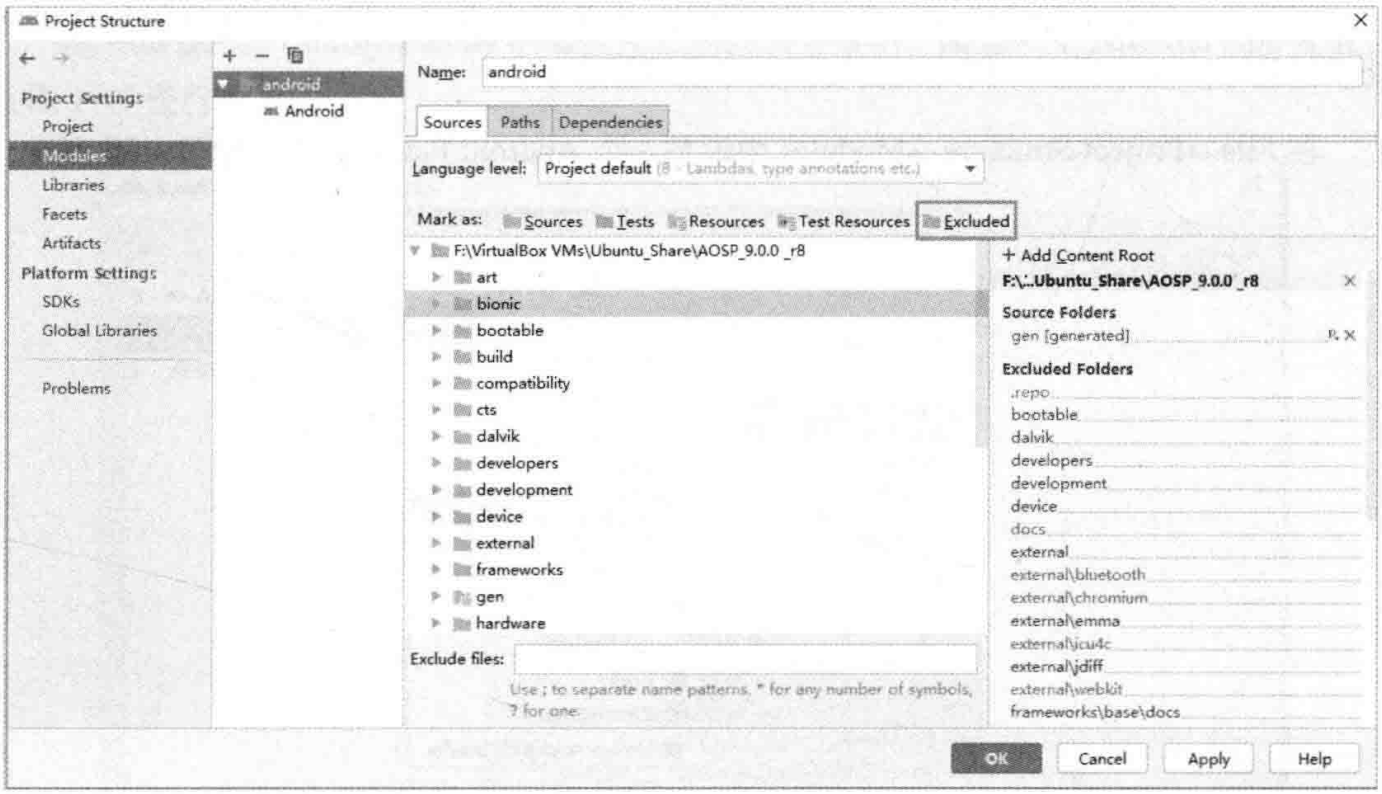


图 1-36　排除不需要的代码目录

1.7　Android Studio 调试系统源码

作为应用开发人员，我们将 AOSP 源码下载完毕后，已经可以查看源码了，如果想更进一步学习，可以学习调试源码，这样有助于了解系统源码及其原理。调试系统源码的方式有很多种，本节介绍适合应用开发的三种方法，分别是 AOSP 源码项目调试、导入部分源码调试、新建 Android 项目调试。

需要注意的是，这三种方法都不需要整编系统源码，甚至有的方法只需要下载部分源码就可以了。哪个方法更适合呢？我们接着往下看。

1.7.1 AOSP 源码项目调试

首先我们需要配置 SDK 和 JDK，这个知识点在 1.6 节已讲过，接下来新建 Android 工程配置。

1.7.1.1 新建 Android 工程配置

实际上不按如下配置操作也可以调试源码，但是打开 Android Studio 时会在 Event Log 处报错且不能消除提醒，因此还需要按如下配置操作。

在 File→Project Structure→Modules 中添加一个 Android Framework，如图 1-37 所示。

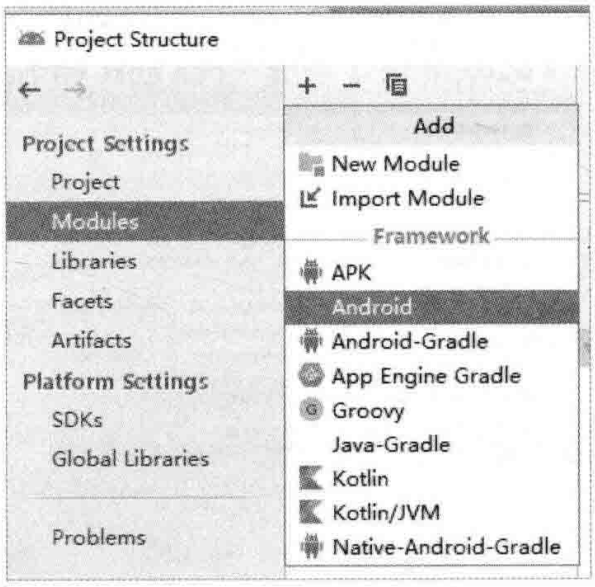


图 1-37　添加一个 Android Framework

将 Manifest file 等路径修改为图 1-38 中类似的路径，因为我们常调试这个目录，所以这样设置，也可以设置为其他路径，如图 1-38 所示。

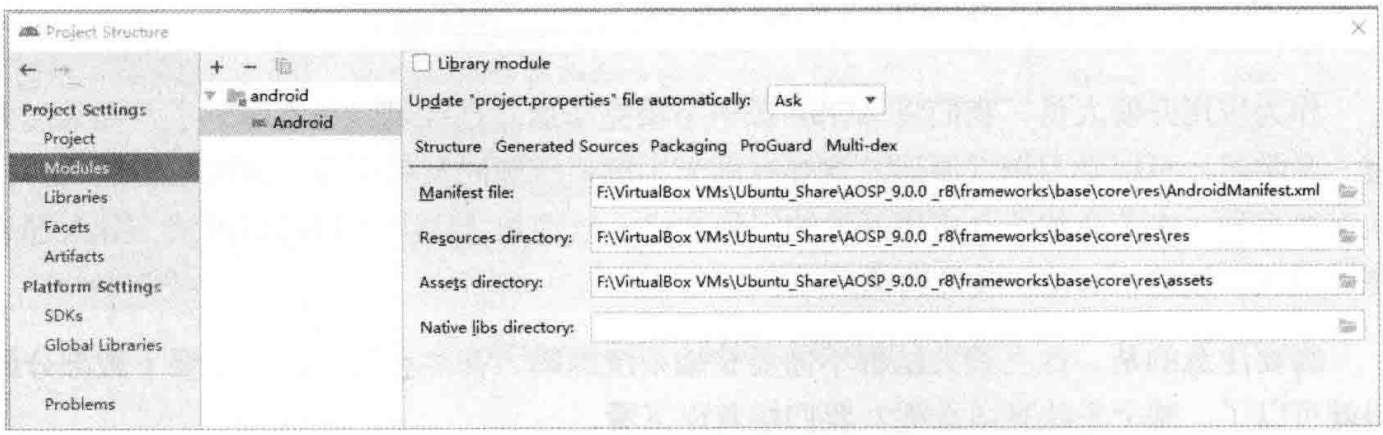


图 1-38　修改 Manifest file 等路径

在 Android Studio 顶部的工具条中选择 Edit Configurations，如图 1-39 所示。

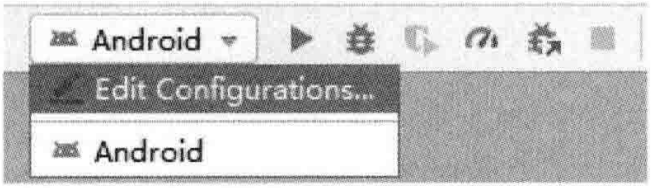


图 1-39　选择 Edit Configurations

在 Run/Debug Configurations 中单击左上角的加号按钮，添加一个 Android App 配置，按图 1-40 进行配置。

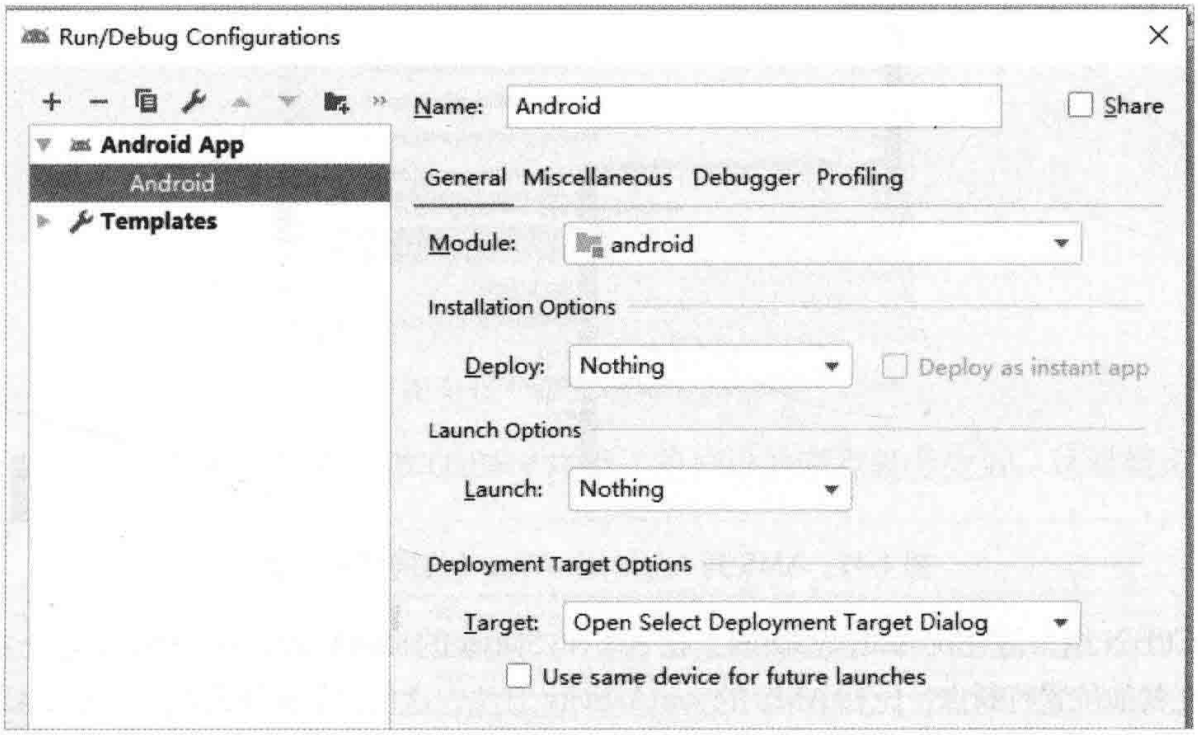


图 1-40　添加一个 Android App 配置

1.7.1.2　调试源码

打开 Genymotion，运行 Android 9.0 的模拟器。

应用程序的启动过程是大部分开发人员都熟悉的知识点，如果还不熟悉这些内容，可以参考《Android 进阶解密》第 4 章。本节的调试源码部分都以应用程序的启动过程点为基础进行调试。应用程序在启动时会调用 ActivityStarter 的 startActivityMayWait 方法，图 1-41 是 AMS 到 ApplicationThread 的调用时序图。

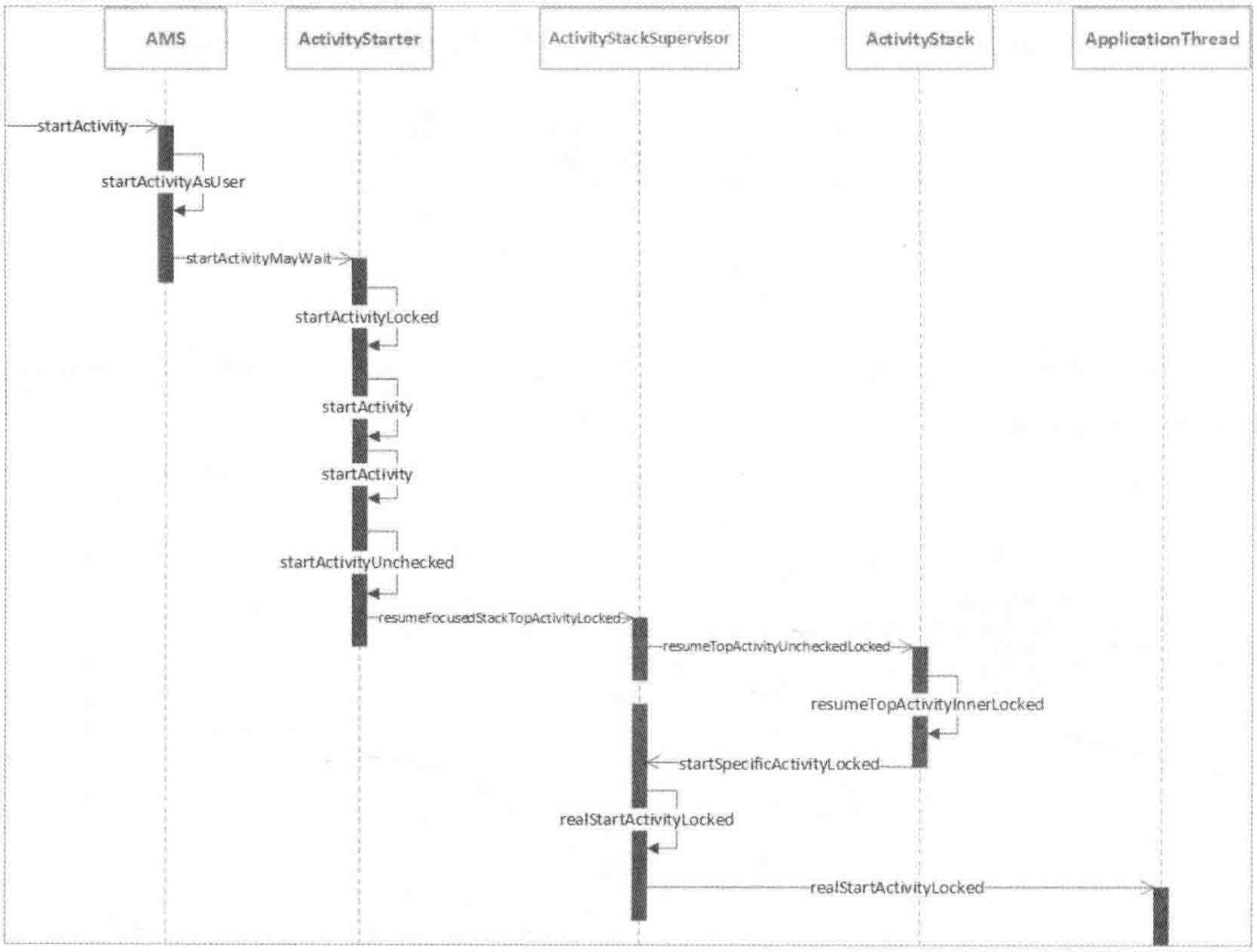


图 1-41　AMS 到 ApplicationThread 的调用时序图

按 Ctrl+N 组合键查找 ActivityStarter，在 ActivityStarter 的 startActivityMayWait 方法上打断点，也可以在其他位置打断点，比如 AMS 的 startActivity 方法，这里只是举例说明，如图 1-42 所示。

```
945     private int startActivityMayWait(IApplicationThread caller, int callingUid,
946             String callingPackage, Intent intent, String resolvedType,
947             IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
948             IBinder resultTo, String resultWho, int requestCode, int startFlags,
949             ProfilerInfo profilerInfo, WaitResult outResult,
950             Configuration globalConfig, SafeActivityOptions options, boolean ignoreTargetSecurity,
951             int userId, TaskRecord inTask, String reason,
952             boolean allowPendingRemoteAnimationRegistryLookup) {
953         // Refuse possible leaked file descriptors
954         if (intent != null && intent.hasFileDescriptors()) {
955             throw new IllegalArgumentException("File descriptors passed in Intent");
956         }
957         mSupervisor.getActivityMetricsLogger().notifyActivityLaunching();
958         boolean componentSpecified = intent.getComponent() != null;
```

图 1-42　在 ActivityStarter 的 startActivityMayWait 方法上打断点

选择菜单项 Run→Attach Debugger to Android Process 或者单击上方工具条的 Attach Debugger to Android Process 图标，勾选 Show all processes，选择 system_process，如图 1-43 所示。

图 1-43　选择 system_process

在 Genymotion 模拟器中选择 Gallery 应用，我们设的断点就会生效，这时就可以进行调试了，如图 1-44 所示。

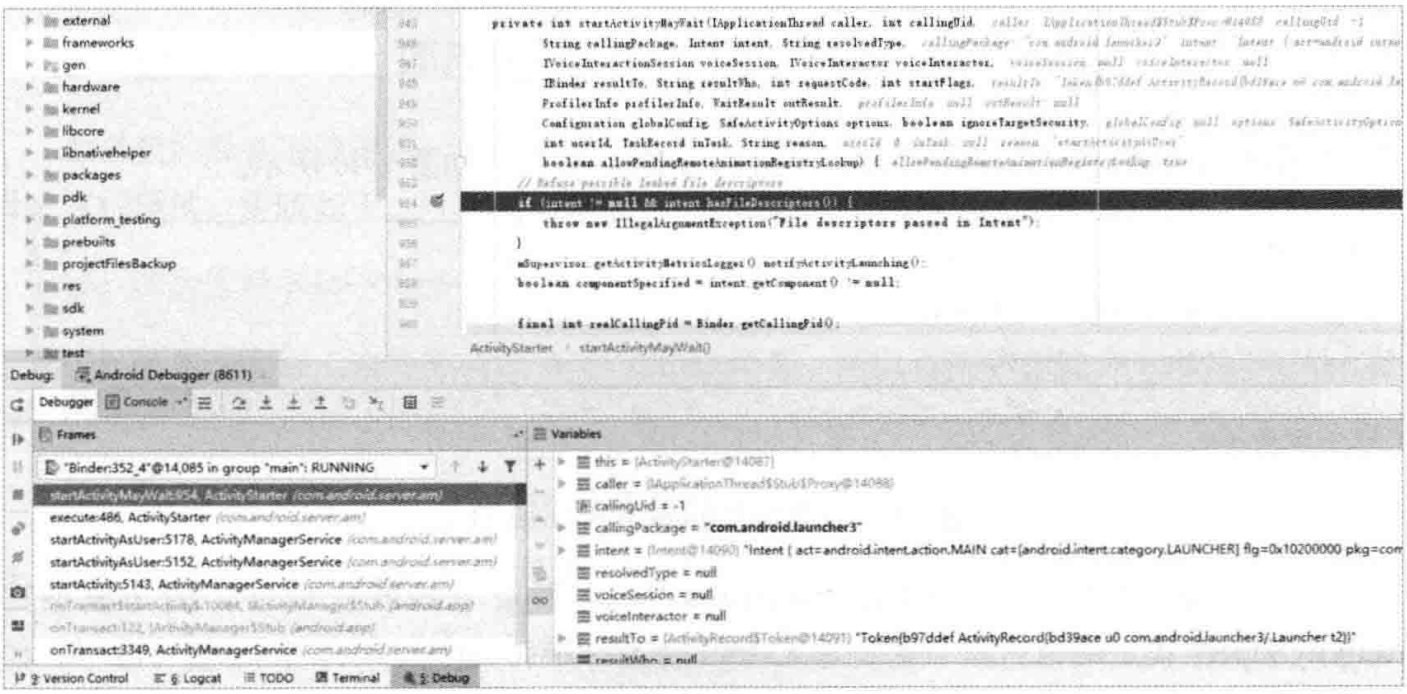


图 1-44　断点生效，进行调试

1.7.2 导入部分源码调试

AOSP 源码项目调试需要对源码的 idegen 模块进行单编，生成 android.ipr，以便 Android Studio（也简称为 AS）导入 AOSP 的全部源码。事实上，我们调试源码时用不到 AOSP 的全部源码，导入部分源码也可以调试。这里以导入常用的 frameworks/base 目录为例。

在当前项目中依次选择 File→New→Import project，或者关闭当前项目，选择 Import project，如图 1-45 所示。

图 1-45　选择 Import project

选择导入 frameworks/base 目录，一路单击 Next 按钮，如图 1-46 所示。

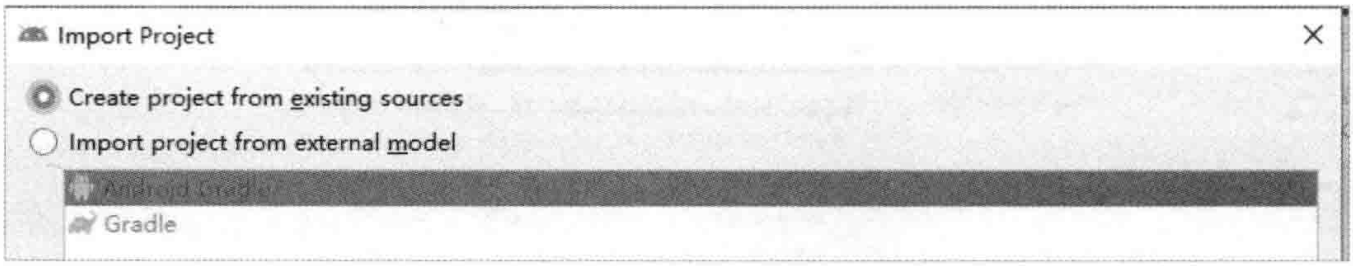


图 1-46　选择导入 frameworks/base 目录

这时 Android Studio 会列出 frameworks\base 目录下的所有项目，因为要调试 ActivityStarter 类，所以这里只需要导入 frameworks\base\services\core\java 就可以了，如图 1-47 所示。

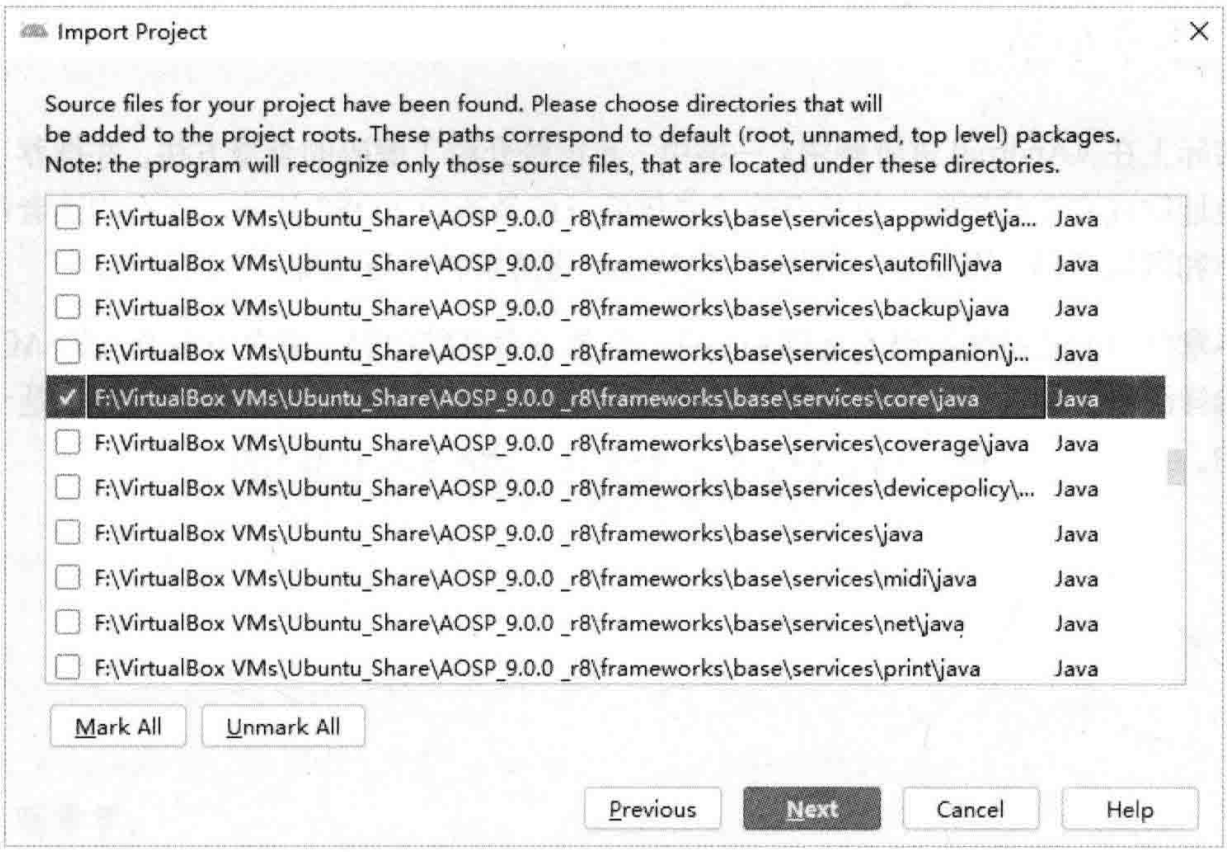


图 1-47　导入 frameworks\base\services\core\java

接着一路单击 Next 按钮就可以了，项目加载进来后，配置项目的 JDK、SDK。接下来的调试步骤和之前讲到的是一样的，这里就不赘述了。

1.7.3　新建 Android 项目调试

如果我们没有下载源码，或者不想导入那么多源码，那么也可以新建一个 Android 项目来进行调试，步骤如下。

（1）依次选择 File→New→New Project，一路单击 Next 按钮就可以了。

（2）新建一个包，因为要调试 ActivityStarter 类，所以包的名称就为 ActivityStarter 类的包名 com.android.server.am。

（3）将 9.0 版本的 ActivityStarter 类复制到包中。

（4）按照 1.7.1 节的内容开始调试。

本节将介绍三种调试源码的方法，对应用开发人员来说，这三种调试方法就够用了，总体来说都大同小异，适合不同需求的开发人员。按照我个人的理解，推荐使用 AOSP 源码项目调试，这种方法可以一劳永逸。

1.8 本章小结

实际上在《Android 进阶解密》一书中，我已经介绍了源码的查看方法，并推荐了一个网盘地址供读者下载源码。但是有的读者想要下载最新的 AOSP 源码，还有的读者希望能够编译和调试源码，因此这一章是对《Android 进阶解密》遗漏知识点的补充。

本章学习的是系统源码的基础知识点，想要学会系统源码，需要下载合适的 AOSP 源码、编译源码、调试源码，那么掌握 Android Studio 和 Source Insight 的相关知识是十分有必要的。

第 2 章

理解包管理机制和 PMS

关联章节：

《Android 进阶解密》 第 2 章 Android 系统启动

包管理机制是 Android 中的重要机制，是应用开发和系统开发需要掌握的知识点之一。

包指的是 apk、jar 和 so 等文件，它们被加载到 Android 内存中，由一个包转换成可执行的代码，需要一个机制来进行包的加载、解析和管理等操作，这就是包管理机制。包管理机制由许多类组成，其中核心为 PackageManagerService（PMS），它负责对包进行管理，如果直接讲 PMS，会比较难以理解，因此我们需要一个切入点，这个切入点就是常见的 APK 的安装，先从安装器 PackageInstaller 开始讲起。

2.1 PackageInstaller 的初始化

在讲 PackageInstaller 之前，首先需要了解 PackageManager、APK 的文件结构和安装方式。

2.1.1 PackageManager 简介

与 ActivityManager 和 AMS 的关系类似，PMS 也有一个对应的管理类 PackageManager，用于向应用程序进程提供一些功能。PackageManager 是一个抽象类，它的具体实现类为 ApplicationPackageManager，ApplicationPackageManager 中的方法会通过 IPackageManager 与 AMS 进行进程间通信，因此 PackageManager 所提供的功能最终是由 PMS 来实现的，这么设计的主要目的是避免系统服务 PMS 被直接访问。PackageManager 提供了一些功能，主要有以下几个。

（1）获取一个应用程序的所有信息（ApplicationInfo）。

（2）获取四大组件的信息。

（3）查询 permission 的相关信息。

（4）获取包的信息。

（5）安装、卸载 APK。

2.1.2 APK 的文件结构和安装方式

APK 是 AndroidPackage 的缩写，即 Android 安装包，它实际上是 zip 格式的压缩文件。在一般情况下，APK 解压后的文件结构如表 2-1 所示。

表 2-1 APK 解压后的文件结构

目录/文件	描　述
assert	存放的原生资源文件，通过 AssetManager 类访问
lib	存放库文件
META-INF	保存应用的签名信息，签名信息可以验证 apk 文件的完整性
res	存放资源文件。res 中除了 raw 子目录，其他子目录都参与编译，这些子目录下的资源可以通过编译出的 R 类在代码中访问
AndroidManifest.xml	用来声明应用程序的包名称、版本、组件和权限等数据。APK 中的 AndroidManifest.xml 经过压缩，可以通过 AXMLPrinter2 工具解开
classes.dex	Java 源码编译后生成的 Java 字节码文件
resources.arsc	编译后的二进制资源文件

APK 主要有以下四种安装方式。

- 通过 adb 命令安装，adb 命令包括 adb push/install。
- 用户下载了 APK，通过系统安装器 PackageInstaller 安装该 APK。PackageInstaller 是系统内置的应用程序，用于安装和卸载应用程序。
- 系统开机时安装系统应用。
- 通过电脑或者手机上的应用商店自动安装。

这四种方式最终都是由 PMS 来进行处理的，与此前的调用链是不同的，这里会介绍第二种方式，也就是通过系统安装器 PackageInstaller 安装 APK；对开发者来说，这是调用链比较长的安装方式，开发者能学到更多知识。

2.1.3　寻找 PackageInstaller 入口

在 Android 7.0 之前，我们可以通过如下代码来安装指定路径中的 APK。

```
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    intent.setDataAndType(Uri.parse("file://" +
path),"application/vnd.android.package-archive");
    context.startActivity(intent);
```

如果 Android 7.0 或更高版本再这么做，就会报 FileUriExposedException 异常。这是因为 StrictMode API 政策禁止应用程序将 file:// Uri 暴露给另一个应用程序，如果包含 file:// Uri 的 Intent 离开应用，就会报 FileUriExposedException 异常。为了解决这个问题，Google 提供了 FileProvider，FileProvider 继承自 ContentProvider，使用它可以将 file://Uri 替换为 content://Uri，具体怎么使用 FileProvider 并不是重点，只要知道无论是在 Android 7.0 之前版本还是在 Android 7.0 及更高版本中，都会调用如下代码。

```
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setDataAndType(xxxxx, "application/vnd.android.package-archive");
```

Intent 的 Action 属性为 ACTION_VIEW，Type 属性指定 Intent 的数据类型为 application/vnd.android.package-archive。

能隐式匹配的 Activity 为 InstallStart，需要注意的是，这里分析的源码基于 Android 8.0，Android 7.0 能隐式匹配的 Activity 为 PackageInstallerActivity，如下所示。

packages/apps/PackageInstaller/AndroidManifest.xml

```
<activity android:name=".InstallStart"
            android:exported="true"
            android:excludeFromRecents="true">
        <intent-filter android:priority="1">
            <action android:name="android.intent.action.VIEW" />
            <action android:name="android.intent.action.INSTALL_PACKAGE" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:scheme="file" />
            <data android:scheme="content" />
            <data android:mimeType="application/vnd.android.package-archive" />
        </intent-filter>
    ...
    </activity>
```

InstallStart 是 PackageInstaller 中的入口 Activity，其中 PackageInstaller 是系统内置的应用程序，用于安装和卸载应用。当我们调用 PackageInstaller 来安装应用时，会跳转到 InstallStart，并调用它的 onCreate 方法，如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallStart.java

```
@Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
  ...
      if (PackageInstaller.ACTION_CONFIRM_PERMISSIONS.equals(intent.
getAction())) {//1
          nextActivity.setClass(this, PackageInstallerActivity.class);
      } else {
          Uri packageUri = intent.getData();
          if (packageUri == null) {//2
              Intent result = new Intent();
              result.putExtra(Intent.EXTRA_INSTALL_RESULT,
                    PackageManager.INSTALL_FAILED_INVALID_URI);
              setResult(RESULT_FIRST_USER, result);
              nextActivity = null;
          } else {
              if (packageUri.getScheme().equals(SCHEME_CONTENT)) {//3
                  nextActivity.setClass(this, InstallStaging.class);
              } else {
                  nextActivity.setClass(this, PackageInstallerActivity.class);
              }
          }
      }
      if (nextActivity != null) {
          startActivity(nextActivity);
```

```
        }
        finish();
    }
```

注释 1 处判断 Intent 的 Action 是否为 CONFIRM_PERMISSIONS，根据应用情景显然不是；注释 2 处判断 packageUri 是否为空，也不成立；注释 3 处判断 Uri 的 Scheme 协议是否是 content，如果是 content 就跳转到 InstallStaging，如果不是 content 就跳转到 PackageInstallerActivity。在本节的应用情景中，对于 Android 7.0 及更高版本，我们会使用 FileProvider 来处理 URI，FileProvider 会隐藏共享文件的真实路径，并将路径转换成 content://Uri 路径，这样就会跳转到 InstallStaging。InstallStaging 的 onResume 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallStaging.java

```
@Override
  protected void onResume() {
      super.onResume();
      if (mStagingTask == null) {
          if (mStagedFile == null) {
              try {
                  mStagedFile = TemporaryFileManager.getStagedFile(this);//1
              } catch (IOException e) {
                  showError();
                  return;
              }
          }
          mStagingTask = new StagingAsyncTask();
          mStagingTask.execute(getIntent().getData());//2
      }
  }
```

在注释 1 处，如果 File 类型的 mStagedFile 为 null，则创建 mStagedFile，mStagedFile 用于存储临时数据。注释 2 处启动 StagingAsyncTask，并传入了 content 协议的 Uri，如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallStaging.java

```
private final class StagingAsyncTask extends AsyncTask<Uri, Void, Boolean> {
    @Override
    protected Boolean doInBackground(Uri... params) {
        if (params == null || params.length <= 0) {
            return false;
        }
        Uri packageUri = params[0];
        try (InputStream in = getContentResolver().openInputStream(packageUri))
{
```

```
                if (in == null) {
                    return false;
                }
                try (OutputStream out = new FileOutputStream(mStagedFile)) {
                    byte[] buffer = new byte[4096];
                    int bytesRead;
                    while ((bytesRead = in.read(buffer)) >= 0) {
                        if (isCancelled()) {
                            return false;
                        }
                        out.write(buffer, 0, bytesRead);
                    }
                }
            } catch (IOException | SecurityException e) {
                Log.w(LOG_TAG, "Error staging apk from content URI", e);
                return false;
            }
            return true;
        }
        @Override
        protected void onPostExecute(Boolean success) {
            if (success) {
                Intent installIntent = new Intent(getIntent());
                installIntent.setClass(InstallStaging.this,
PackageInstallerActivity.class);
                installIntent.setData(Uri.fromFile(mStagedFile));
                installIntent
                        .setFlags(installIntent.getFlags() &
~Intent.FLAG_ACTIVITY_FORWARD_RESULT);
                installIntent.addFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION);
                startActivityForResult(installIntent, 0);
            } else {
                showError();
            }
        }
    }
}
```

在 doInBackground 方法中，将 packageUri（content 协议的 Uri）的内容写入 mStagedFile 中，如果写入成功，则在 onPostExecute 方法中会跳转到 PackageInstallerActivity 中，并将 mStagedFile 传进去。绕了一圈又回到了 PackageInstallerActivity 中，从这里可以看出 InstallStaging 主要起转换作用，将 content 协议的 Uri 转换为 file 协议，然后跳转到 PackageInstallerActivity 中，这样就可以像此前的版本（Android 7.0 之前版本）一样启动安装流程了。

2.1.4 PackageInstallerActivity 解析

从功能上说，PackageInstallerActivity 才是应用安装器 PackageInstaller 真正的入口 Activity，PackageInstallerActivity 的 onCreate 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        if (icicle != null) {
            mAllowUnknownSources = icicle.getBoolean(ALLOW_UNKNOWN_SOURCES_KEY);
        }
        mPm = getPackageManager();
        mIpm = AppGlobals.getPackageManager();
        mAppOpsManager = (AppOpsManager) getSystemService(Context.
APP_OPS_SERVICE);
        mInstaller = mPm.getPackageInstaller();
        mUserManager = (UserManager) getSystemService(Context.USER_SERVICE);
        ...
        //根据 Uri 的 Scheme 进行预处理
        boolean wasSetUp = processPackageUri(packageUri);//1
        if (!wasSetUp) {
            return;
        }
        bindUi(R.layout.install_confirm, false);
        //判断是否是未知来源的应用，如果开启允许安装未知来源选项，则直接进行初始化安装
        checkIfAllowedAndInitiateInstall();//2
    }
```

首先初始化安装所需的各种对象，比如 PackageManager、IPackageManager、AppOpsManager 和 UserManager 等，它们的描述如表 2-2 所示。

表 2-2　PackageManager、IPackageManager 等类的描述

类　名	描　述
PackageManager	用于向应用程序进程提供一些功能，最终的功能是由 PMS 来实现的
IPackageManager	一个 AIDL 的接口，用于和 PMS 进行进程间通信
AppOpsManager	用于权限动态检测，从 Android 4.3 开始被引入
PackageInstaller	提供安装、升级和删除应用程序的功能
UserManager	用于多用户管理

注释 1 处的 processPackageUri 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
    private boolean processPackageUri(final Uri packageUri) {
        mPackageURI = packageUri;
        final String scheme = packageUri.getScheme();//1
        switch (scheme) {
            case SCHEME_PACKAGE: {
                try {
                 ...
            } break;
            case SCHEME_FILE: {
                File sourceFile = new File(packageUri.getPath());//2
                //得到 sourceFile 的包信息
                PackageParser.Package parsed = PackageUtil.getPackageInfo(this,
sourceFile);//3
                if (parsed == null) {
                    Log.w(TAG, "Parse error when parsing manifest. Discontinuing
installation");
                    showDialogInner(DLG_PACKAGE_ERROR);
                    setPmResult(PackageManager.INSTALL_FAILED_INVALID_APK);
                    return false;
                }
                //对 parsed 进行进一步的处理并得到包信息 PackageInfo
                mPkgInfo = PackageParser.generatePackageInfo(parsed, null,
                        PackageManager.GET_PERMISSIONS, 0, 0, null,
                        new PackageUserState());//4
               mAppSnippet = PackageUtil.getAppSnippet(this,mPkgInfo.applicationInfo,
sourceFile);
            } break;
            default: {
                Log.w(TAG, "Unsupported scheme " + scheme);
                setPmResult(PackageManager.INSTALL_FAILED_INVALID_URI);
                finish();
                return false;
            }
        }
        return true;
    }
```

首先在注释 1 处得到 packageUri 的 Scheme 协议，然后根据这个 Scheme 协议分别对 package 协议和 file 协议进行处理，如果不是这两个协议，就会关闭 PackageInstallerActivity 并返回 false。我们主要来看 file 协议的处理过程，注释 1 处根据 packageUri 创建了一个新

的 File。注释 2 处的内部会用 PackageParser 的 parsePackage 方法解析这个 File（这个 File 其实是 apk 文件），得到 apk 文件的包信息 Package，Package 包含了该 apk 文件的所有信息。注释 3 处会将 Package 根据 uid、用户状态信息和 PackageManager 的配置等变量对包信息 Package 进行进一步的处理并得到 PackageInfo。

回到 PackageInstallerActivity 的 onCreate 方法的注释 2 处，checkIfAllowedAndInitiateInstall 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
    private void checkIfAllowedAndInitiateInstall() {
        //判断是否允许安装未知来源的 APK 或者根据 Intent 判断得出该 APK 不是未知来源的
        if (mAllowUnknownSources || !isInstallRequestFromUnknownSource
(getIntent())) {//1
            //初始化安装
            initiateInstall();//2
            return;
        }
        // 如果管理员限制来自未知来源的安装流程，就弹出提示对话框或者跳转到设置界面
        if (isUnknownSourcesDisallowed()) {
            if ((mUserManager.getUserRestrictionSource(UserManager.DISALLOW_
INSTALL_UNKNOWN_SOURCES,Process.myUserHandle()) & UserManager.RESTRICTION_
SOURCE_SYSTEM) != 0)
                {showDialogInner(DLG_UNKNOWN_SOURCES_ RESTRICTED_FOR_USER);
                return;
            } else {
                startActivity(new Intent(Settings.ACTION_SHOW_ADMIN_SUPPORT_
DETAILS));
                finish();
            }
        } else {
            handleUnknownSources();//3
        }
    }
```

注释 1 处判断是否允许安装未知来源的 APK 或者根据 Intent 判断得出该 APK 不是未知来源的，就调用注释 2 处的 initiateInstall 方法来进行初始化安装。如果管理员限制来自未知来源的安装，就弹出提示对话框或者跳转到设置界面，否则就调用注释 3 处的 handleUnknownSources 方法来处理未知来源的 APK。注释 2 处的 initiateInstall 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
    private void initiateInstall() {
        String pkgName = mPkgInfo.packageName;//1
        String[] oldName = mPm.canonicalToCurrentPackageNames(new String[]
{ pkgName });
        if (oldName != null && oldName.length > 0 && oldName[0] != null) {
            pkgName = oldName[0];
            mPkgInfo.packageName = pkgName;
            mPkgInfo.applicationInfo.packageName = pkgName;
        }
        try {
            //根据包名获取应用程序信息
            mAppInfo = mPm.getApplicationInfo(pkgName,
                    PackageManager.MATCH_UNINSTALLED_PACKAGES);//2
            if ((mAppInfo.flags&ApplicationInfo.FLAG_INSTALLED) == 0) {
                mAppInfo = null;
            }
        } catch (NameNotFoundException e) {
            mAppInfo = null;
        }
        //初始化安装的确认界面
        startInstallConfirm();//3
    }
```

注释 1 处得到了包名，注释 2 处根据包名获取应用程序信息 ApplicationInfo，注释 3 处的 startInstallConfirm 方法如下所示。

/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
    private void startInstallConfirm() {
        //省略初始化界面代码
        ...
        AppSecurityPermissions perms = new AppSecurityPermissions(this,
mPkgInfo);//1
        final int N = perms.getPermissionCount(AppSecurityPermissions.WHICH_ALL);
        if (mAppInfo != null) {
            msg = (mAppInfo.flags & ApplicationInfo.FLAG_SYSTEM) != 0
                    ? R.string.install_confirm_question_update_system
                    : R.string.install_confirm_question_update;
            mScrollView = new CaffeinatedScrollView(this);
            mScrollView.setFillViewport(true);
            boolean newPermissionsFound = false;
            if (!supportsRuntimePermissions) {
                newPermissionsFound =
                    (perms.getPermissionCount(AppSecurityPermissions.WHICH_NEW) > 0);
```

```
            if (newPermissionsFound) {
                permVisible = true;
                mScrollView.addView(perms.getPermissionsView(
                        AppSecurityPermissions.WHICH_NEW));//2
            }
        }
    ...
}
```

在 startInstallConfirm 方法中，首先初始化安装确认界面，就是我们平常安装 APK 时出现的界面，界面上有确认和取消按钮，并会列出安装该 APK 需要访问的系统权限。需要注意的是，不同厂商定制的 Android 系统会有不同的安装确认界面。

注释 1 处创建了 AppSecurityPermissions，它会提取出 APK 中的权限信息并展示出来，这个负责展示的 View 是 AppSecurityPermissions 的内部类 PermissionItemView。注释 2 处调用 AppSecurityPermissions 的 getPermissionsView 方法来获取 PermissionItemView，并将 PermissionItemView 添加到 CaffeinatedScrollView 中，这样安装该 APK 需要访问的系统权限就可以全部展示出来了，PackageInstaller 的初始化工作就完成了。

2.1.5　PackageInstaller 初始化的过程

现在来总结 PackageInstaller 初始化的过程如下。

（1）根据 Uri 的 Scheme 协议不同，跳转到不同的界面，content 协议跳转到 InstallStart，其他的跳转到 PackageInstallerActivity。在本节的应用场景中，如果使用的是 Android 7.0 及更高版本，则会跳转到 InstallStart。

（2）InstallStart 将 content 协议的 Uri 转换为 file 协议，然后跳转到 PackageInstallerActivity。

（3）PackageInstallerActivity 会分别对 package 协议和 file 协议的 Uri 进行处理，如果是 file 协议，则会解析 apk 文件得到包信息 PackageInfo。

（4）PackageInstallerActivity 会对未知来源的 APK 进行处理，如果允许安装未知来源的 APK 或者根据 Intent 判断得出该 APK 不是未知来源的，就会初始化安装确认界面。如果管理员限制来自未知来源的安装流程，就会弹出提示对话框或者跳转到设置界面。

2.2 PackageInstaller 安装 APK 过程

我们接着学习 PackageInstaller 是如何安装 APK 的。

2.2.1 PackageInstaller 中的处理

紧接着 2.1 节的内容，在 PackageInstallerActivity 调用 startInstallConfirm 方法初始化安装确认界面后，这个安装确认界面就会呈现给用户，用户如果想要安装这个应用程序，就会单击确定按钮，这时就会调用 PackageInstallerActivity 的 onClick 方法，如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
public void onClick(View v) {
    if (v == mOk) {
        if (mOk.isEnabled()) {
            if (mOkCanInstall || mScrollView == null) {
                if (mSessionId != -1) {
                    mInstaller.setPermissionsResult(mSessionId, true);
                    finish();
                } else {
                    startInstall();//1
                }
            } else {
                mScrollView.pageScroll(View.FOCUS_DOWN);
            }
        }
    } else if (v == mCancel) {
        ...
        finish();
    }
}
```

onClick 方法中分别对确定和取消按钮进行了处理，下面主要查看对确定按钮的处理，上面代码注释 1 处调用的 startInstall 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/PackageInstallerActivity.java

```
private void startInstall() {
    Intent newIntent = new Intent();
    newIntent.putExtra(PackageUtil.INTENT_ATTR_APPLICATION_INFO,
            mPkgInfo.applicationInfo);
    newIntent.setData(mPackageURI);//1
    newIntent.setClass(this, InstallInstalling.class);
```

```
        String installerPackageName = getIntent().getStringExtra(
              Intent.EXTRA_INSTALLER_PACKAGE_NAME);
        if (mOriginatingURI != null) {
           newIntent.putExtra(Intent.EXTRA_ORIGINATING_URI, mOriginatingURI);
        }
        ...
        if(localLOGV) Log.i(TAG, "downloaded app uri="+mPackageURI);
        startActivity(newIntent);
        finish();
     }
```

startInstall 方法用于跳转到 InstallInstalling 的 Activity，并关闭当前的 PackageInstallerActivity。InstallInstalling 主要用于向包管理器发送包的信息并处理包管理的回调。InstallInstalling 的 onCreate 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallInstalling.java

```
     @Override
      protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.install_installing);
        ApplicationInfo appInfo = getIntent()
              .getParcelableExtra(PackageUtil.INTENT_ATTR_APPLICATION_INFO);
        mPackageURI = getIntent().getData();
        if ("package".equals(mPackageURI.getScheme())) {
           try {
              getPackageManager().installExistingPackage(appInfo.packageName);
              launchSuccess();
           } catch (PackageManager.NameNotFoundException e) {
              launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR, null);
           }
        } else {
           //根据 mPackageURI 创建一个对应的 File
           final File sourceFile = new File(mPackageURI.getPath());
           PackageUtil.initSnippetForNewApp(this, PackageUtil.getAppSnippet(this,
appInfo,sourceFile), R.id.app_snippet);
           //如果 savedInstanceState 不为 null，则获取此前保存的 mSessionId 和 mInstallId
           if (savedInstanceState != null) {//1
              mSessionId = savedInstanceState.getInt(SESSION_ID);
              mInstallId = savedInstanceState.getInt(INSTALL_ID);
              //向 InstallEventReceiver 注册一个观察者
              try {
                 InstallEventReceiver.addObserver(this, mInstallId,
                       this::launchFinishBasedOnResult);//2
              } catch (EventResultPersister.OutOfIdsException e) {
```

```
                }
            } else {
                PackageInstaller.SessionParams params = new
PackageInstaller.SessionParams(
                        PackageInstaller.SessionParams.MODE_FULL_INSTALL);//3
                params.referrerUri =
getIntent().getParcelableExtra(Intent.EXTRA_REFERRER);
                params.originatingUri = getIntent()
                        .getParcelableExtra(Intent.EXTRA_ORIGINATING_URI);
                params.originatingUid =
getIntent().getIntExtra(Intent.EXTRA_ORIGINATING_UID,
                        UID_UNKNOWN);
                File file = new File(mPackageURI.getPath());//4
                try {
                    PackageParser.PackageLite pkg =
PackageParser.parsePackageLite(file, 0);//5
                    params.setAppPackageName(pkg.packageName);
                    params.setInstallLocation(pkg.installLocation);
                    params.setSize(
                            PackageHelper.calculateInstalledSize(pkg, false,
params.abiOverride));
                } catch (PackageParser.PackageParserException e) {
                    ...
                }
                try {
                    mInstallId = InstallEventReceiver
                            .addObserver(this, EventResultPersister.GENERATE_NEW_ID,
                                    this::launchFinishBasedOnResult);//6
                } catch (EventResultPersister.OutOfIdsException e) {
                    launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR,
null);
                }
                try {
                    mSessionId = getPackageManager().getPackageInstaller().
createSession(params);//7
                } catch (IOException e) {
                    launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR, null);
                }
            }
            ...
            mSessionCallback = new InstallSessionCallback();
        }
    }
```

onCreate 方法中会分别对 package 和 content 协议的 Uri 进行处理，我们来看 content 协议的 Uri 处理部分。

如果注释 1 处的 savedInstanceState 不为 null，则获取此前保存的 mSessionId 和 mInstallId，其中 mSessionId 是安装包的会话 id，mInstallId 是等待的安装事件 id。

注释 2 处根据 mInstallId 向 InstallEventReceiver 注册一个观察者，launchFinishBasedOnResult 会接收到安装事件的回调，无论安装成功还是失败，都会关闭当前的 Activity(InstallInstalling)。如果 savedInstanceState 为 null，代码的逻辑也是类似的，则在注释 3 处创建 SessionParams，它用来代表安装会话的参数，注释 4、5 处根据 mPackageUri 对包（APK）进行轻量级的解析，并将解析的参数赋值给 SessionParams。

注释 6 处和注释 2 处类似，向 InstallEventReceiver 注册一个观察者并返回一个新的 mInstallId，其中 InstallEventReceiver 继承自 BroadcastReceiver，用于接收安装事件并回调给 EventResultPersister。注释 7 处 PackageInstaller 的 createSession 方法内部会通过 IPackageInstaller 与 PackageInstallerService 进行进程间通信，最终调用的是 PackageInstallerService 的 createSession 方法来创建并返回 mSessionId。

InstallInstalling 的 onCreate 方法就分析到这里，接着查看 InstallInstalling 的 onResume 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallInstalling.java

```
    @Override
     protected void onResume() {
        super.onResume();
        if (mInstallingTask == null) {
           PackageInstaller installer = getPackageManager().getPackageInstaller();
           PackageInstaller.SessionInfo sessionInfo =
installer.getSessionInfo(mSessionId);//1
           if (sessionInfo != null && !sessionInfo.isActive()) {//2
              mInstallingTask = new InstallingAsyncTask();
              mInstallingTask.execute();
           } else {
              mCancelButton.setEnabled(false);
              setFinishOnTouchOutside(false);
           }
        }
     }
```

注释 1 处根据 mSessionId 得到 SessionInfo，SessionInfo 代表安装会话的详细信息。如

果注释 2 处的 sessionInfo 不为 null 并且不是活动的，就创建并执行 InstallingAsyncTask。InstallingAsyncTask 的 doInBackground 方法中会根据包（APK）的 Uri，将 APK 的信息通过 I/O 流的形式写入 PackageInstaller.Session 中。InstallingAsyncTask 的 onPostExecute 方法如下所示。

packages/apps/PackageInstaller/src/com/android/packageinstaller/InstallInstalling.java

```
@Override
 protected void onPostExecute(PackageInstaller.Session session) {
    if (session != null) {
       Intent broadcastIntent = new Intent(BROADCAST_ACTION);
       broadcastIntent.setPackage(
             getPackageManager().getPermissionControllerPackageName());
      broadcastIntent.putExtra(EventResultPersister.EXTRA_ID, mInstallId);
       PendingIntent pendingIntent = PendingIntent.getBroadcast(
             InstallInstalling.this,
             mInstallId,
             broadcastIntent,
             PendingIntent.FLAG_UPDATE_CURRENT);
       session.commit(pendingIntent.getIntentSender());//1
       mCancelButton.setEnabled(false);
       setFinishOnTouchOutside(false);
    } else {

getPackageManager().getPackageInstaller().abandonSession(mSessionId);
       if (!isCancelled()) {
          launchFailure(PackageManager.INSTALL_FAILED_INVALID_APK, null);
       }
    }
 }
```

注释 1 处创建了一个 PendingIntent，并将该 PendingIntent 的 IntentSender 通过 PackageInstaller.Session 的 commit 方法发送了出去，发送去哪了呢？我们接着查看 PackageInstaller.Session 的 commit 方法。

frameworks/base/core/java/android/content/pm/PackageInstaller.java

```
public void commit(@NonNull IntentSender statusReceiver) {
    try {
       mSession.commit(statusReceiver);
    } catch (RemoteException e) {
       throw e.rethrowFromSystemServer();
    }
 }
```

mSession 的类型为 IPackageInstallerSession，这说明要通过 IPackageInstallerSession 进行进程间的通信，最终会调用 PackageInstallerSession 的 commit 方法，这样代码逻辑就到了 Java 框架层。

2.2.2　Java 框架层的处理

Java 框架层的处理如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageInstallerSession.java

```
    @Override
      public void commit(IntentSender statusReceiver) {
         Preconditions.checkNotNull(statusReceiver);
         ...
         mActiveCount.incrementAndGet();
        final PackageInstallObserverAdapter adapter = new PackageInstallObserverAdapter
(mContext,statusReceiver, sessionId, mIsInstallerDeviceOwner, userId);
         mHandler.obtainMessage(MSG_COMMIT, adapter.getBinder()).
sendToTarget();//1
      }
```

commit 方法中会将包的信息封装为 PackageInstallObserverAdapter，它在 PMS 中被定义。注释 1 处会向 Handler 发送一个类型为 MSG_COMMIT 的消息，其中 adapter.getBinder() 会得到 IPackageInstallObserver2.Stub 类型的观察者，由类型可知这个观察者是可以跨进程进行回调的。处理该消息的代码如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageInstallerSession.java

```
     private final Handler.Callback mHandlerCallback = new Handler.Callback() {
         @Override
         public boolean handleMessage(Message msg) {
            final PackageInfo pkgInfo = mPm.getPackageInfo(
                  params.appPackageName, PackageManager.GET_SIGNATURES
                        | PackageManager.MATCH_STATIC_SHARED_LIBRARIES /*flags*/,
userId);
            final ApplicationInfo appInfo = mPm.getApplicationInfo(
                  params.appPackageName, 0, userId);
            synchronized (mLock) {
               if (msg.obj != null) {
                  mRemoteObserver = (IPackageInstallObserver2) msg.obj;//1
               }
               try {
                  commitLocked(pkgInfo, appInfo);//2
```

```
                } catch (PackageManagerException e) {
                    final String completeMsg = ExceptionUtils.getCompleteMessage(e);
                    Slog.e(TAG, "Commit of session " + sessionId + " failed: " +
completeMsg);
                    destroyInternal();
                    dispatchSessionFinished(e.error, completeMsg, null);//3
                }
                return true;
            }
        }
    };
```

注释 1 处获取了 IPackageInstallObserver2 类型的观察者 mRemoteObserver，注释 2 处的 commitLocked 方法如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageInstallerSession.java

```
private void commitLocked(PackageInfo pkgInfo, ApplicationInfo appInfo)
        throws PackageManagerException {
    ...
    mPm.installStage(mPackageName, stageDir, stageCid, localObserver, params,
            installerPackageName, installerUid, user, mCertificates);
}
```

commitLocked 方法比较长，这里仅截取了主要信息，其会调用 PMS 的 installStage 方法，这样代码逻辑就进入 PMS 的代码逻辑中。

回到 mHandlerCallback 的 handleMessage 方法，如果 commitLocked 方法出现 PackageManagerException 异常，就会调用注释 3 处的 dispatchSessionFinished 方法，它的实现如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageInstallerSession.java

```
private void dispatchSessionFinished(int returnCode, String msg, Bundle extras)
{
    mFinalStatus = returnCode;
    mFinalMessage = msg;
    if (mRemoteObserver != null) {
        try {
            mRemoteObserver.onPackageInstalled(mPackageName, returnCode, msg,
extras);//1
        } catch (RemoteException ignored) {
        }
    }
    ...
```

```
    }
```

注释 1 处会调用 IPackageInstallObserver2 的 onPackageInstalled 方法，其具体实现位于 PackageInstallObserver 类中。

frameworks/base/core/java/android/app/PackageInstallObserver.java

```
public class PackageInstallObserver {
    private final IPackageInstallObserver2.Stub mBinder = new
IPackageInstallObserver2.Stub() {
        ...
        @Override
        public void onPackageInstalled(String basePackageName, int returnCode,
            String msg, Bundle extras) {
            PackageInstallObserver.this.onPackageInstalled(basePackageName,
returnCode, msg,
                extras);//1
        }
    };
```

注释 1 处调用了 PackageInstallObserver 的 onPackageInstalled 方法，实现这个方法的类为 PackageInstallObserver 的子类和前面提到的 PackageInstallObserverAdapter。总结一下就是 dispatchSessionFinished 方法会通过 mRemoteObserver 的 onPackageInstalled 方法将 Complete 方法出现的 PackageManagerException 异常信息回调给 PackageInstallObserverAdapter。

2.2.3　PackageInstaller 安装 APK 的过程

简单来讲，PackageInstaller 安装 APK 的过程有以下两步。

（1）将 APK 的信息通过 I/O 流的形式写入 PackageInstaller.Session 中。

（2）调用 PackageInstaller.Session 的 commit 方法，将 APK 的信息交由 PMS 处理。

2.3　PMS 处理 APK 的安装过程

我们已经学习了 PackageInstaller 是如何安装 APK 的，最后会将 APK 的信息交由 PMS 处理，那么 PMS 如何处理 APK 的信息呢？本节会给出答案。

2.3.1 PackageHandler 处理安装消息

APK 的信息交由 PMS 后，PMS 通过向 PackageHandler 发送消息来驱动 APK 的复制和安装工作。

首先来查看 PackageHandler 处理安装消息的调用时序图，如图 2-1 所示。

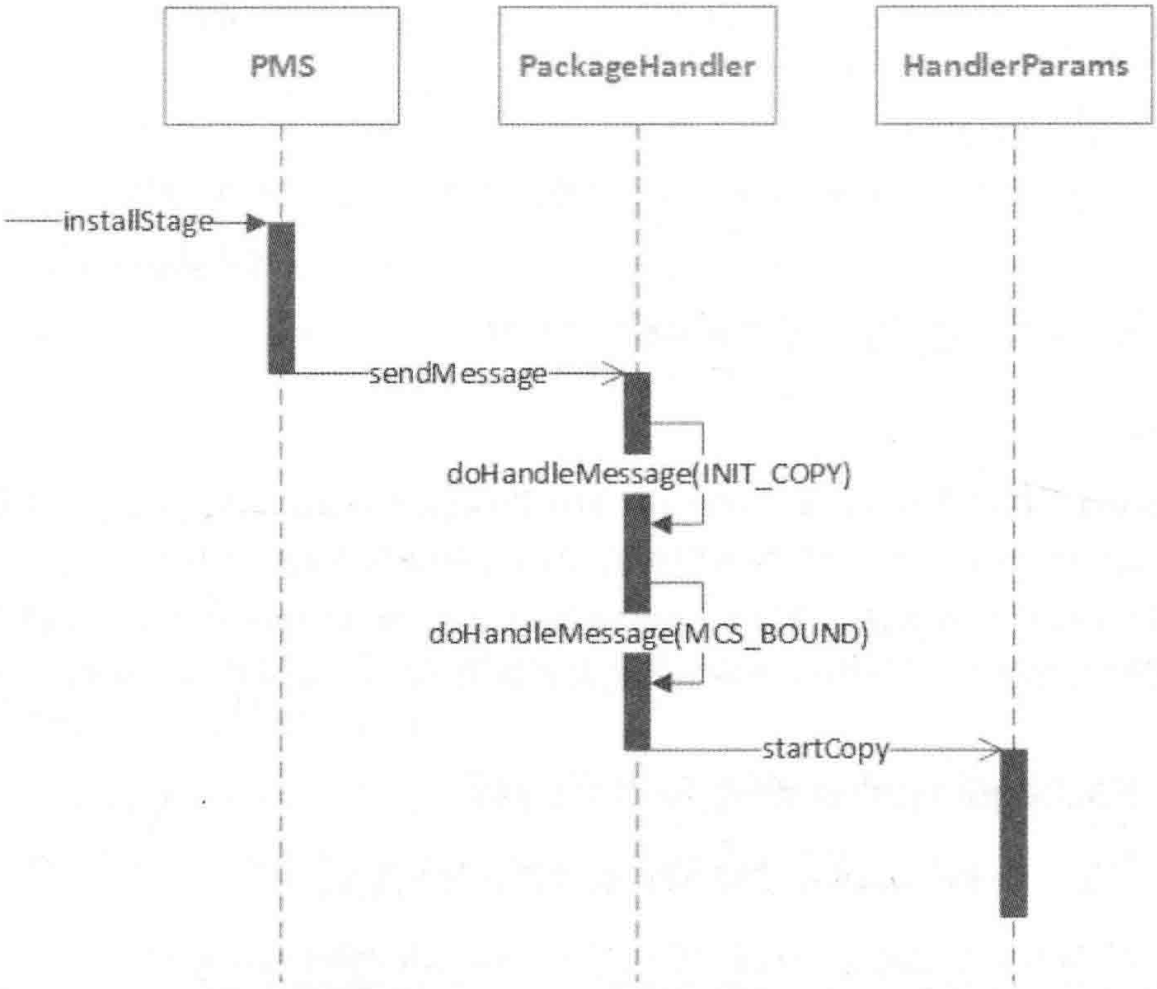


图 2-1 PackageHandler 处理安装消息的调用时序图

接着前面章节的代码逻辑来查看 PMS 的 installStage 方法。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
void installStage(String packageName, File stagedDir, String stagedCid,
        IPackageInstallObserver2 observer, PackageInstaller.SessionParams
sessionParams,String installerPackageName, int installerUid, UserHandle user,
        Certificate[][] certificates) {
    ...
    final Message msg = mHandler.obtainMessage(INIT_COPY);//1
    final int installReason = fixUpInstallReason(installerPackageName,
installerUid,sessionParams.installReason);
```

```
        final InstallParams params = new InstallParams(origin, null, observer,
sessionParams.installFlags, installerPackageName,
sessionParams.volumeUuid,verificationInfo, user, sessionParams.abiOverride,
sessionParams. grantedRuntimePermissions, certificates, installReason);//2
        params.setTraceMethod("installStage").setTraceCookie
(System.identityHashCode(params));
        msg.obj = params;
        ...
        mHandler.sendMessage(msg);//3
    }
```

注释 1 处创建了类型为 INIT_COPY 的消息。注释 2 处创建了 InstallParams，它对应包的安装数据。注释 3 处通过消息将 InstallParams 发送出去。

2.3.1.1　对 INIT_COPY 类型的消息进行处理

处理 INIT_COPY 类型的消息的代码如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java#PackageHandler

```
        void doHandleMessage(Message msg) {
            switch (msg.what) {
                case INIT_COPY: {
                    HandlerParams params = (HandlerParams) msg.obj;
                    int idx = mPendingInstalls.size();
                    if (DEBUG_INSTALL) Slog.i(TAG, "init_copy idx=" + idx + ": " +
params);//mBound 用于标识是否绑定了服务，其默认值为 false
                    if (!mBound) {//1
                        Trace.asyncTraceBegin(TRACE_TAG_PACKAGE_MANAGER,
"bindingMCS",System.identityHashCode(mHandler));
                        //如果没有绑定服务，则重新绑定，如果 connectToService 方法内部绑定服务成功，
                        //则会将 mBound 设置为 true
                        if (!connectToService()) {//2
                            Slog.e(TAG, "Failed to bind to media container service");
                            params.serviceError();
                            Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER,
"bindingMCS",System.identityHashCode(mHandler));
                            if (params.traceMethod != null) {
                                Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER,
params.traceMethod,params.traceCookie);
                            }
                            //绑定服务失败则返回
                            return;
```

```
                } else {
                    //绑定服务成功，将请求添加到 ArrayList 类型的 mPendingInstalls 中，
                    //等待处理
                    mPendingInstalls.add(idx, params);
                }
            } else {
                //已经绑定服务
                mPendingInstalls.add(idx, params);
                if (idx == 0) {
                    mHandler.sendEmptyMessage(MCS_BOUND);//3
                }
            }
            break;
        }
        ....
        }
    }
}
```

PackageHandler 继承自 Handler，它被定义在 PMS 中，doHandleMessage 方法用于处理各个类型的消息，并查看对 INIT_COPY 类型消息的处理。注释 1 处的 mBound 用于标识是否绑定了 DefaultContainerService，其默认值为 false。DefaultContainerService 用于检查和复制可移动文件的服务，这是一个比较耗时的操作，因此 DefaultContainerService 没有和 PMS 运行在同一进程中，它运行在 com.android.defcontainer 进程中，通过 IMediaContainerService 和 PMS 进行 IPC 通信，如图 2-2 所示。

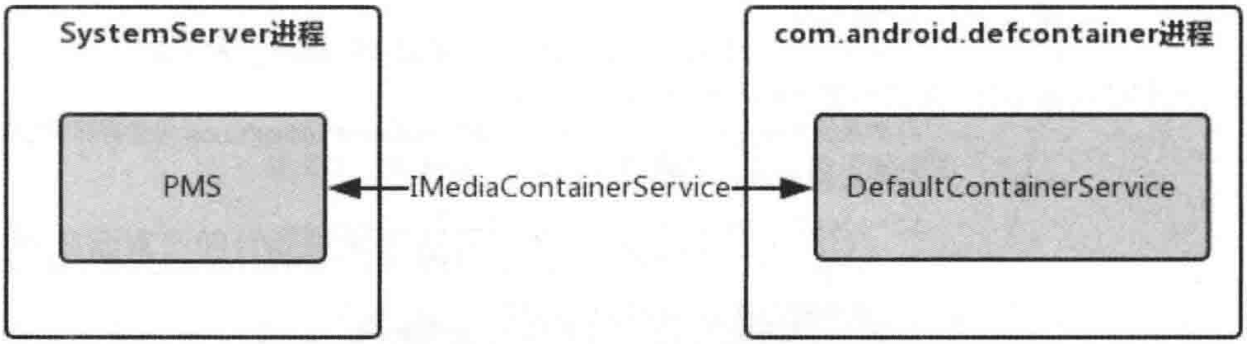


图 2-2　PMS 和 DefaultContainerService 进行 IPC 通信

上面代码注释 2 处的 connectToService 方法用于绑定 DefaultContainerService，注释 3 处发送了 MCS_BOUND 类型的消息，触发处理安装请求。

查看 connectToService 方法如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java# PackageHandler

```
private boolean connectToService() {
        if (DEBUG_SD_INSTALL) Log.i(TAG, "Trying to bind to" +
              " DefaultContainerService");
        Intent service = new Intent().setComponent(DEFAULT_CONTAINER_COMPONENT);
        Process.setThreadPriority(Process.THREAD_PRIORITY_DEFAULT);
        if (mContext.bindServiceAsUser(service, mDefContainerConn,
              Context.BIND_AUTO_CREATE, UserHandle.SYSTEM)) {//1
           Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
           mBound = true;//2
           return true;
        }
        Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
        return false;
    }
```

如果注释 2 处绑定 DefaultContainerService 成功，则 mBound 会设置为 ture。注释 1 处的 bindServiceAsUser 方法会传入 mDefContainerConn，bindServiceAsUser 方法的处理逻辑和我们调用 bindService 是类似的，在建立服务连接后，会调用 onServiceConnected 方法，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
class DefaultContainerConnection implements ServiceConnection {
    public void onServiceConnected(ComponentName name, IBinder service) {
        if (DEBUG_SD_INSTALL) Log.i(TAG, "onServiceConnected");
        final IMediaContainerService imcs = IMediaContainerService.Stub
              .asInterface(Binder.allowBlocking(service));
        mHandler.sendMessage(mHandler.obtainMessage(MCS_BOUND, Object));//1
    }
    public void onServiceDisconnected(ComponentName name) {
        if (DEBUG_SD_INSTALL) Log.i(TAG, "onServiceDisconnected");
    }
}
```

注释 1 处发送了 MCS_BOUND 类型的消息，与 PackageHandler.doHandleMessage 方法的注释 3 处不同的是，这里发送的消息带 Object 类型的参数，接下来会对这两种情况进行讲解，一种是消息不带 Object 类型的参数，另一种是消息带 Object 类型的参数。

2.3.1.2 对 MCS_BOUND 类型的消息进行处理

1. 消息不带 Object 类型的参数

查看对 MCS_BOUND 类型消息的处理，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
case MCS_BOUND: {
          if (DEBUG_INSTALL) Slog.i(TAG, "mcs_bound");
          if (msg.obj != null) {//1
             mContainerService = (IMediaContainerService) msg.obj;//2
             Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER, "bindingMCS",
                   System.identityHashCode(mHandler));
          }
          if (mContainerService == null) {//3
             if (!mBound) {//4
                  Slog.e(TAG, "Cannot bind to media container service");
                  for (HandlerParams params : mPendingInstalls) {
                     params.serviceError();//5
                     Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER,
"queueInstall",System.identityHashCode(params));
                     if (params.traceMethod != null) {
                     Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER,
                      params.traceMethod, params.traceCookie);
                     }
                     return;
                  }
                     //绑定失败，清空安装请求队列
                     mPendingInstalls.clear();
                } else {
                     //继续等待绑定服务
                     Slog.w(TAG, "Waiting to connect to media container service");
                }
          } else if (mPendingInstalls.size() > 0) {
            ...
            else {
                Slog.w(TAG, "Empty queue");
                }
          break;
       }
```

如果消息不带 Object 类型的参数，则无法满足注释 1 处的条件，注释 2 处的 IMediaContainerService 类型的 mContainerService 也无法被赋值，这样就满足了注释 3 处的

条件。

如果满足注释 4 处的条件，说明还没有绑定服务，而此前已经在 PackageHandler.doHandleMessage 方法的注释 2 处调用绑定服务的方法了，这显然是不正常的，因此在注释 5 处负责处理服务发生错误的情况。如果不满足注释 4 处的条件，则说明已经绑定服务了，就会打印出系统日志，告知用户等待系统绑定服务。

2. 消息带 Object 类型的参数

消息带 Object 类型的参数的代码如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
case MCS_BOUND: {
        if (DEBUG_INSTALL) Slog.i(TAG, "mcs_bound");
        if (msg.obj != null) {
        ...
        }
        if (mContainerService == null) {//1
         ...
        } else if (mPendingInstalls.size() > 0) {//2
                  HandlerParams params = mPendingInstalls.get(0);//3
                 if (params != null) {
                    Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER,
"queueInstall",System.identityHashCode(params));
                    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "startCopy");
                    if (params.startCopy()) {//4
                       if (DEBUG_SD_INSTALL) Log.i(TAG,
                             "Checking for more work or unbind...");
                       //如果APK安装成功，则删除本次安装请求
                       if (mPendingInstalls.size() > 0) {
                          mPendingInstalls.remove(0);
                       }
                       if (mPendingInstalls.size() == 0) {
                          if (mBound) {
                             //如果没有安装请求了，则发送解绑服务的请求
                             if (DEBUG_SD_INSTALL) Log.i(TAG,
                                   "Posting delayed MCS_UNBIND");
                             removeMessages(MCS_UNBIND);
                             Message ubmsg = obtainMessage(MCS_UNBIND);
                             sendMessageDelayed(ubmsg, 10000);
                          }
                       } else {
                          if (DEBUG_SD_INSTALL) Log.i(TAG,
```

```
                                "Posting MCS_BOUND for next work");
                            //如果还有其他安装请求，则接着发送MCS_BOUND消息继续处理剩余
                            //的安装请求
                             mHandler.sendEmptyMessage(MCS_BOUND);//5
                        }
                    }
                    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
                }else {
                Slog.w(TAG, "Empty queue");//6
            }
        break;
    }
```

如果 MCS_BOUND 类型的消息带 Object 类型的参数，就不会满足注释 1 处的条件，就会调用注释 2 处的判断语句。如果安装请求数不大于 0，就会打印出注释 6 处的日志，说明安装请求队列是空的。安装完一个 APK 后，就会在注释 5 处发出 MSC_BOUND 消息，继续处理剩下的安装请求，直到安装请求队列为空。

注释 3 处得到安装请求队列中的第一个请求 HandlerParams，如果 HandlerParams 不为 null，就会调用注释 4 处的 HandlerParams 的 startCopy 方法，用于复制 APK 的流程。

2.3.2 复制 APK 的处理

先来查看复制 APK 的时序图，如图 2-3 所示。

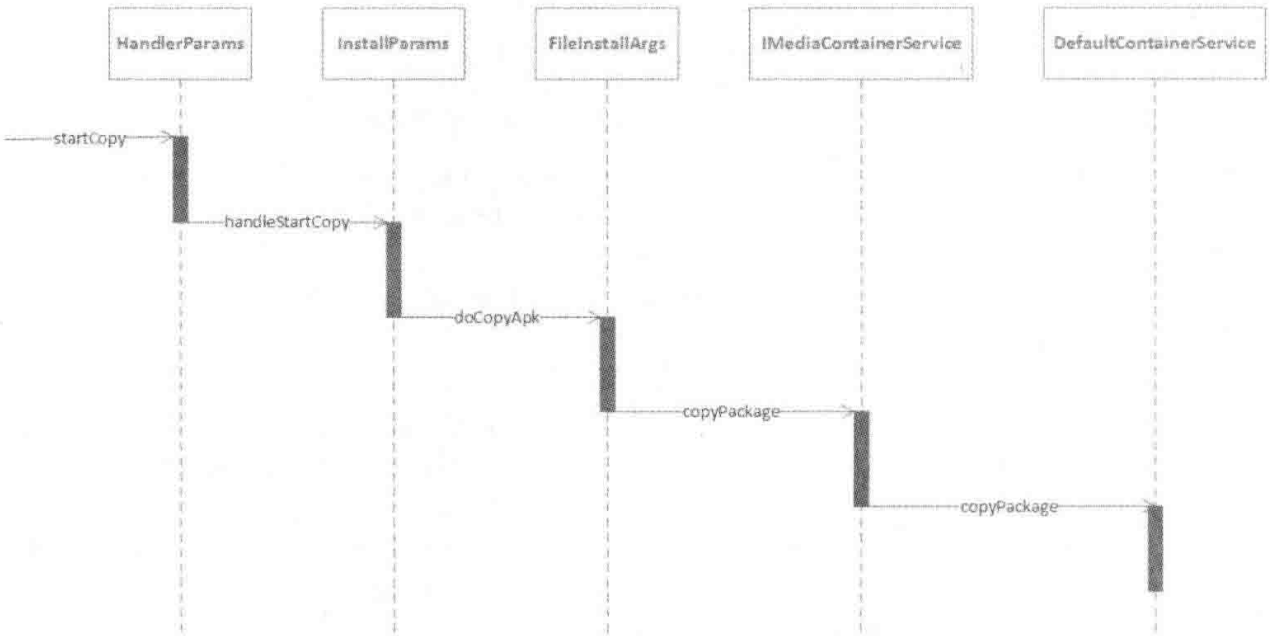


图 2-3　复制 APK 的时序图

HandlerParams 是 PMS 中的抽象类，它的实现类为 PMS 的内部类 InstallParams。HandlerParams 的 startCopy 方法如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java# HandlerParams

```
    final boolean startCopy() {
            boolean res;
            try {
                if (DEBUG_INSTALL) Slog.i(TAG, "startCopy " + mUser + ": " + this);
                //如果 startCopy 方法尝试的次数超过了 4 次，就放弃这个安装请求
                if (++mRetries > MAX_RETRIES) {//1
                    Slog.w(TAG, "Failed to invoke remote methods on default container
service. Giving up");
                    mHandler.sendEmptyMessage(MCS_GIVE_UP);//2
                    handleServiceError();
                    return false;
                } else {
                    handleStartCopy();//3
                    res = true;
                }
            } catch (RemoteException e) {
                if (DEBUG_INSTALL) Slog.i(TAG, "Posting install MCS_RECONNECT");
                mHandler.sendEmptyMessage(MCS_RECONNECT);
                res = false;
            }
            handleReturnCode();//4
            return res;
        }
```

注释 1 处的 mRetries 用于记录调用 startCopy 方法的次数，调用 startCopy 方法时会先自动加 1，如果尝试的次数大于 4 次就放弃这个安装请求，在注释 2 处发送 MCS_GIVE_UP 类型消息，将第一个安装请求（本次安装请求）从安装请求队列 mPendingInstalls 中移除。注释 4 处用于处理复制 APK 后的安装 APK 逻辑，后面会再次提到它。注释 3 处调用了抽象方法 handleStartCopy，它的实现在 InstallParams 中，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java# InstallParams

```
    public void handleStartCopy() throws RemoteException {
        ...
        //确定 APK 的安装位置。onSd: 安装到 SD 卡中; onInt: 安装到内部存储（即 Data 分区）中;
        //ephemeral: 安装到临时存储中（Instant Apps 安装）
        final boolean onSd = (installFlags & PackageManager.INSTALL_EXTERNAL) != 0;
```

```
            final boolean onInt = (installFlags & PackageManager.INSTALL_INTERNAL) != 0;
            final boolean ephemeral = (installFlags & PackageManager.INSTALL_INSTANT_
APP) != 0;
            PackageInfoLite pkgLite = null;
            if (onInt && onSd) {
               // APK 不能同时安装到 SD 卡和 Data 分区中
               Slog.w(TAG, "Conflicting flags specified for installing on both
internal and external");
               ret = PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION;
            //安装标志冲突，Instant Apps 不能安装到 SD 卡中
            } else if (onSd && ephemeral) {
               Slog.w(TAG,  "Conflicting flags specified for installing ephemeral on
external");
               ret = PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION;
            } else {
                //获取 APK 的少量信息
               pkgLite = mContainerService.getMinimalPackageInfo(origin.
resolvedPath, installFlags, packageAbiOverride);//1
               if (DEBUG_EPHEMERAL && ephemeral) {
                  Slog.v(TAG, "pkgLite for install: " + pkgLite);
               }
            ...
            if (ret == PackageManager.INSTALL_SUCCEEDED) {
                //判断安装的位置
               int loc = pkgLite.recommendedInstallLocation;
               if (loc == PackageHelper.RECOMMEND_FAILED_INVALID_LOCATION) {
                  ret = PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION;
               } else if (loc == PackageHelper.RECOMMEND_FAILED_ALREADY_EXISTS) {
                  ret = PackageManager.INSTALL_FAILED_ALREADY_EXISTS;
               }
               ...
               }else{
                loc = installLocationPolicy(pkgLite);//2
                ...
               }
            }
            //根据 InstallParams 创建 InstallArgs 对象
            final InstallArgs args = createInstallArgs(this);//3
            mArgs = args;
            if (ret == PackageManager.INSTALL_SUCCEEDED) {
                  ...
               if (!origin.existing && requiredUid != -1
                     && isVerificationEnabled(
```

```
                        verifierUser.getIdentifier(), installFlags,
installerUid)) {
                    ...
                } else{
                    ret = args.copyApk(mContainerService, true);//4
                }
            }
            mRet = ret;
        }
```

handleStartCopy 方法的代码很多，这里仅截取了关键部分。注释 1 处通过 IMediaContainerService 跨进程调用了 DefaultContainerService 的 getMinimalPackageInfo 方法，该方法轻量解析 APK 并得到 APK 的少量信息，轻量解析的原因是这里不需要得到 APK 的全部信息，APK 的少量信息会被封装到 PackageInfoLite 中。接着在注释 2 处确定 APK 的安装位置。注释 3 处创建了 InstallArgs，InstallArgs 是一个抽象类，定义了 APK 的安装逻辑，比如复制和重命名 APK 等，InstallArgs 有 3 个子类，都被定义在 PMS 中，如图 2-4 所示。

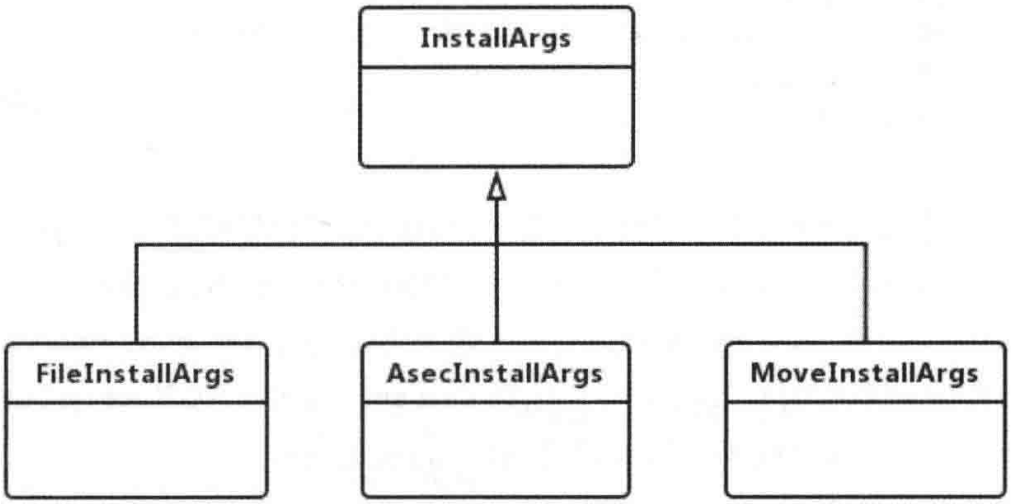


图 2-4　InstallArgs 的子类

其中，FileInstallArgs 用于处理安装到非 ASEC 的存储空间的 APK，也就是内部存储空间（Data 分区），AsecInstallArgs 用于处理安装到 ASEC 中（mnt/asec）（即 SD 卡）中的 APK。MoveInstallArgs 用于处理已安装 APK 在存储中移动的逻辑。

对 APK 进行检查后就会在注释 4 处调用 InstallArgs 的 copyApk 方法进行安装。

不同的 InstallArgs 子类会有不同的处理方法，这里以 FileInstallArgs 为例。FileInstallArgs 的 copyApk 方法直接返回了 FileInstallArgs 的 doCopyApk 方法，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java#FileInstallArgs

```
    private int doCopyApk(IMediaContainerService imcs, boolean temp) throws
RemoteException {
            ...
            try {
                final boolean isEphemeral = (installFlags & PackageManager.INSTALL_
INSTANT_APP) != 0;
                //创建临时文件存储目录
                final File tempDir =
                        mInstallerService.allocateStageDirLegacy(volumeUuid,
isEphemeral);//1
                codeFile = tempDir;
                resourceFile = tempDir;
            } catch (IOException e) {
                Slog.w(TAG, "Failed to create copy file: " + e);
                return PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
            }
            ...
            int ret = PackageManager.INSTALL_SUCCEEDED;
            ret = imcs.copyPackage(origin.file.getAbsolutePath(), target);//2
            ...
            return ret;
        }
```

注释 1 处创建了临时文件存储目录，比如/data/app/vmdl18300388.tmp，其中 18300388 是安装的 sessionId。注释 2 处通过 IMediaContainerService 跨进程调用了 DefaultContainerService 的 copyPackage 方法，这个方法会在 DefaultContainerService 所在的进程中将 APK 复制到临时存储目录中，比如/data/app/vmdl18300388.tmp/base.apk。截至目前，APK 的复制工作就完成了，接着就是 APK 的安装过程了。

2.3.3 PMS 处理 APK 安装

首先来查看安装 APK 的时序图，如图 2-5 所示。

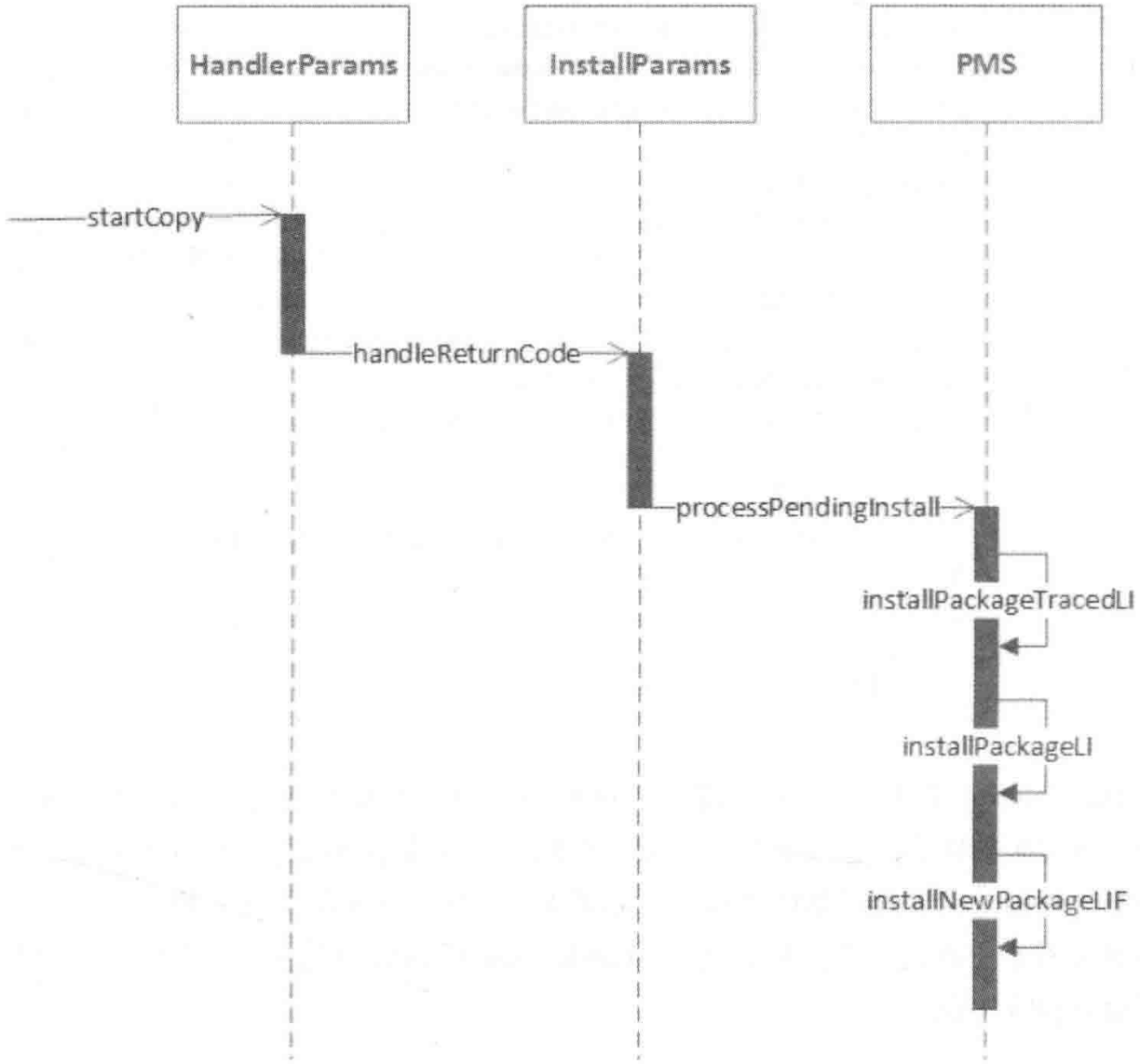


图 2-5　安装 APK 的时序图

回到 APK 的复制调用链的头部方法：HandlerParams 的 startCopy 方法，注释 4 处会调用 handleReturnCode 方法，它的实现在 InstallParams 中，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
    void handleReturnCode() {
      if (mArgs != null) {
         processPendingInstall(mArgs, mRet);
      }
   }

      private void processPendingInstall(final InstallArgs args, final int
currentStatus) {
         mHandler.post(new Runnable() {
            public void run() {
```

```
                mHandler.removeCallbacks(this);
                PackageInstalledInfo res = new PackageInstalledInfo();
                res.setReturnCode(currentStatus);
                res.uid = -1;
                res.pkg = null;
                res.removedInfo = null;
                if (res.returnCode == PackageManager.INSTALL_SUCCEEDED) {
                    //安装前处理
                    args.doPreInstall(res.returnCode);//1
                    synchronized (mInstallLock) {
                        installPackageTracedLI(args, res);//2
                    }
                    //安装后收尾
                    args.doPostInstall(res.returnCode, res.uid);//3
                }
            ...
            }
        });
    }
```

在 handleReturnCode 方法中只调用了 processPendingInstall 方法。注释 1 处会检查 APK 的状态，在安装前需要确保安装环境可靠，如果安装环境不可靠，则会清除复制的 APK 文件。注释 3 处会处理安装后的收尾操作，如果安装不成功，则删除安装相关的目录与文件。下面主要来看看上面代码中注释 2 处的 installPackageTracedLI 方法，其内部会调用 PMS 的 installPackageLI 方法，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {
    ...
    PackageParser pp = new PackageParser();
    pp.setSeparateProcesses(mSeparateProcesses);
    pp.setDisplayMetrics(mMetrics);
    pp.setCallback(mPackageParserCallback);
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parsePackage");
    final PackageParser.Package pkg;
    try {
        //解析 APK
        pkg = pp.parsePackage(tmpPackageFile, parseFlags);//1
    } catch (PackageParserException e) {
        res.setError("Failed parse during installPackageLI", e);
        return;
    } finally {
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
```

```
        }
        ...
        pp = null;
        String oldCodePath = null;
        boolean systemApp = false;
        synchronized (mPackages) {
            // 检查APK是否存在
            if ((installFlags & PackageManager.INSTALL_REPLACE_EXISTING) != 0) {
                //获取没被改名前的包名
                String oldName = mSettings.getRenamedPackageLPr(pkgName);
                if (pkg.mOriginalPackages != null
                        && pkg.mOriginalPackages.contains(oldName)
                        && mPackages.containsKey(oldName)) {
                    pkg.setPackageName(oldName);//2
                    pkgName = pkg.packageName;
                    replace = true;//设置标志位，表示替换安装
                    if (DEBUG_INSTALL) Slog.d(TAG, "Replacing existing renamed package:
oldName=" + oldName + " pkgName=" + pkgName);
                }
                ...
            }
            PackageSetting ps = mSettings.mPackages.get(pkgName);
            //查看Settings中是否存在要安装的APK的信息，如果有就获取签名信息
            if (ps != null) {//3
                if (DEBUG_INSTALL) Slog.d(TAG, "Existing package: " + ps);
                PackageSetting signatureCheckPs = ps;
                if (pkg.applicationInfo.isStaticSharedLibrary()) {
                    SharedLibraryEntry libraryEntry =
getLatestSharedLibraVersionLPr(pkg);
                    if (libraryEntry != null) {
                        signatureCheckPs = mSettings.getPackageLPr(libraryEntry.apk);
                    }
                }
                //检查签名的正确性
                if (shouldCheckUpgradeKeySetLP(signatureCheckPs, scanFlags)) {
                    if (!checkUpgradeKeySetLP(signatureCheckPs, pkg)) {
                        res.setError(INSTALL_FAILED_UPDATE_INCOMPATIBLE, "Package "
                                + pkg.packageName + " upgrade keys do not match the "
                                + "previously installed version");
                        return;
                    }
                }
                ...
            }
```

```
            int N = pkg.permissions.size();
            for (int i = N-1; i >= 0; i--) {
                //遍历每个权限，对权限进行处理
                PackageParser.Permission perm = pkg.permissions.get(i);
                BasePermission bp = mSettings.mPermissions.get(perm.info.name);

                }
            }
        }
        if (systemApp) {
            if (onExternal) {
                //系统App不能在SD卡上替换安装
                res.setError(INSTALL_FAILED_INVALID_INSTALL_LOCATION,
                        "Cannot install updates to system apps on sdcard");
                return;
            } else if (instantApp) {
                //系统App不能被Instant App替换
                res.setError(INSTALL_FAILED_INSTANT_APP_INVALID,
                        "Cannot update a system app with an instant app");
                return;
            }
        }
        ...
        //重命名临时文件
        if (!args.doRename(res.returnCode, pkg, oldCodePath)) {//4
            res.setError(INSTALL_FAILED_INSUFFICIENT_STORAGE, "Failed rename");
            return;
        }

        startIntentFilterVerifications(args.user.getIdentifier(), replace, pkg);

        try (PackageFreezer freezer = freezePackageForInstall(pkgName, installFlags,
                "installPackageLI")) {

            if (replace) {//5
                //替换安装
                ...
                replacePackageLIF(pkg, parseFlags, scanFlags | SCAN_REPLACING,
args.user, installerPackageName, res, args.installReason);
            } else {


                //安装新的APK
```

```
            installNewPackageLIF(pkg, parseFlags, scanFlags |
SCAN_DELETE_DATA_ON_FAILURES, args.user, installerPackageName, volumeUuid, res,
args.installReason);
        }
    }

    synchronized (mPackages) {
        final PackageSetting ps = mSettings.mPackages.get(pkgName);
        if (ps != null) {
            //更新应用程序的所属用户
            res.newUsers = ps.queryInstalledUsers(sUserManager.getUserIds(), true);
            ps.setUpdateAvailable(false /*updateAvailable*/);
        }
        ...
    }
}
```

installPackageLI 方法的代码有将近 500 行，这里仅截取了代码的核心部分，主要做了以下几件事。

（1）创建 PackageParser，解析 APK。

（2）检查 APK 是否存在，如果存在就获取此前没被改名前的包名并在注释 1 处赋值给 PackageParser.Package 类型的 pkg，注释 3 处将标志位 replace 设置为 true，表示替换安装。

（3）在注释 3 处，如果 Settings 中存在要安装的 APK 的信息，说明此前安装过该 APK，则需要校验 APK 的签名信息，确保进行替换安装是安全的。

（4）在注释 4 处，重命名临时文件，比如将前面提到的/data/app/vmdl18300388.tmp/base.apk 重命名为/data/app/包名-1/base.apk。

（5）系统 App 的更新安装会有两个限制，一个是系统 App 不能在 SD 卡上进行替换安装，另一个是系统 App 不能被 Instant App 替换。

（6）在注释 5 处，根据 replace 来进行区分处理，如果是替换安装，就调用 replacePackageLIF 方法，其方法内部还会对系统 App 和非系统 App 进行区分处理，如果是安装新的 APK，就调用 installNewPackageLIF 方法。

这里我们以安装新的 APK 并调用 PMS 的 installNewPackageLIF 方法为例，代码如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
    private void installNewPackageLIF(PackageParser.Package pkg, final int
policyFlags, int scanFlags, UserHandle user, String installerPackageName, String
volumeUuid, PackageInstalledInfo res, int installReason) {
        ...
        try {
            //扫描 APK
            PackageParser.Package newPackage = scanPackageTracedLI(pkg, policyFlags,
scanFlags, System.currentTimeMillis(), user);
            //更新 Settings 信息
            updateSettingsLI(newPackage, installerPackageName, null, res, user,
installReason);
            if (res.returnCode == PackageManager.INSTALL_SUCCEEDED) {
                //如果安装成功，就为新安装的应用程序准备数据
                prepareAppDataAfterInstallLIF(newPackage);

            } else {
                //如果安装失败，则删除 APK
                deletePackageLIF(pkgName, UserHandle.ALL, false, null,
                        PackageManager.DELETE_KEEP_DATA, res.removedInfo, true, null);
            }
        } catch (PackageManagerException e) {
            res.setError("Package couldn't be installed in " + pkg.codePath, e);
        }
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    }
```

installNewPackageLIF 方法主要做了以下三件事。

（1）扫描 APK，将 APK 的信息存储在 PackageParser.Package 类型的 newPackage 中，一个 Package 的信息包含了一个 base APK 及零个或者多个 split APK。

（2）更新该 APK 对应的 Settings 信息，Settings 用于保存所有包的动态设置。

（3）如果安装成功，就为新安装的应用程序准备数据；如果安装失败，就删除 APK。

安装 APK 的过程就讲到这里，感兴趣的读者可以接着深入研究。

2.3.4 PMS 处理 APK 安装的步骤

本节主要讲解了 PMS 是如何处理 APK 安装的，主要有以下几个步骤：

（1）PackageInstaller 安装 APK 时会将 APK 的信息交由 PMS 处理，PMS 通过向

PackageHandler 发送消息来驱动 APK 的复制和安装工作。

(2) PMS 发送 INIT_COPY 和 MCS_BOUND 类型的消息，控制 PackageHandler 来绑定 DefaultContainerService，完成复制 APK 等工作。

(3) 复制完 APK 后，开始进行安装 APK 的流程，包括安装前的检查、安装 APK 和安装后的收尾工作。

2.4　PMS 的创建过程

PMS 的创建过程分为两个部分来进行讲解，分别是 SystemServer 处理部分和 PMS 构造方法。其中 SystemServer 处理部分和 AMS、WMS 的创建过程是类似的，可以将它们进行对比，这样可以深入理解和记忆这一知识点。

2.4.1　SystemServer 处理部分

PMS 是在 SystemServer 进程中被创建的，SystemServer 进程用来创建系统服务，不了解 SystemServer 的读者可以查看《Android 进阶解密》第 2 章。

从 SystemServer 的入口方法 main 方法开始讲起，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
      new SystemServer().run();
   }
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {
   try {
      ...

      //创建消息 Looper
       Looper.prepareMainLooper();
      //加载了动态库 libandroid_servers.so
      System.loadLibrary("android_servers");//1
      performPendingShutdown();
```

```
            //创建系统的 Context
            createSystemContext();
            //创建 SystemServiceManager
            mSystemServiceManager = new SystemServiceManager(mSystemContext);//2
            mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
            LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
            SystemServerInitThreadPool.get();
        } finally {
            traceEnd();
        }
        try {
            traceBeginAndSlog("StartServices");
            //启动引导服务
            startBootstrapServices();//3
            //启动核心服务
            startCoreServices();//4
            //启动其他服务
            startOtherServices();//5
            SystemServerInitThreadPool.shutdown();
        } catch (Throwable ex) {
            Slog.e("System", "******************************************");
            Slog.e("System", "************ Failure starting system services", ex);
            throw ex;
        } finally {
            traceEnd();
        }
        ...
    }
```

注释 1 处加载了动态库 libandroid_servers.so。注释 2 处创建了 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。注释 3 处的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService 和 PackageManagerService 等服务。注释 4 处的 startCoreServices 方法中则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService。注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService 和 VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、4、5 的方法中可以看出，官方把系统服务分为三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要的和不需要立即启动的服务。这些系统服务总共有 100 多个，我们熟知的 AMS 属于引导服务，WMS 属于其他服务。

本节讲的 PMS 属于引导服务，引导服务及其作用如表 2-3 所示。

表 2-3　引导服务及其作用

引 导 服 务	作　　用
Installer	系统安装 APK 时的一个服务类，启动完成 Installer 服务之后，才能启动其他系统服务
ActivityManagerService	负责四大组件的启动、切换和调度
PowerManagerService	计算系统中和 Power 相关的计算任务，决策系统应该如何反应
LightsService	管理和显示背光 LED
DisplayManagerService	管理所有显示设备
UserManagerService	多用户模式管理
SensorService	为系统提供各种感应器服务
PackageManagerService	对 APK 进行安装、解析、删除和卸载等操作

查看启动引导服务的注释 3 处的 startBootstrapServices 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void startBootstrapServices() {
    ...
     String cryptState = SystemProperties.get("vold.decrypt");//1
     if (ENCRYPTING_STATE.equals(cryptState)) {
          Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
          mOnlyCore = true;
       } else if (ENCRYPTED_STATE.equals(cryptState)) {
          Slog.w(TAG, "Device encrypted - only parsing core apps");
          mOnlyCore = true;
       }
    ...
    traceBeginAndSlog("StartPackageManagerService");
    mPackageManagerService = PackageManagerService.main(mSystemContext,
installer, mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);//2
    mFirstBoot = mPackageManagerService.isFirstBoot();//3
    mPackageManager = mSystemContext.getPackageManager();
    traceEnd();
    ...
}
```

注释 1 处读取 init.rc 的 vold.decrypt 属性，如果它的值为"trigger_restart_min_framework"，说明我们加密了设备，这时 mOnlyCore 的值为 true，表示只运行“核心”程序，这是为了创建一个极简的启动环境。

注释 2 处的 PMS 的 main 方法主要用来创建 PMS。注释 3 处获取了 boolean 类型的变

量 mFirstBoot，它表示 PMS 是否首次被启动。mFirstBoot 是后续 WMS 创建时所需的参数，从这里就可以看出系统服务之间是有依赖关系的，它们的启动顺序不能随意被更改。

2.4.2 PMS 构造方法

PMS 的 main 方法如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
public static PackageManagerService main(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
    PackageManagerServiceCompilerMapping.checkProperties();
    PackageManagerService m = new PackageManagerService(context, installer,
            factoryTest, onlyCore);
    m.enableSystemUserPackages();
    ServiceManager.addService("package", m);
    return m;
}
```

main 方法主要有两个作用，一个是创建 PMS 对象，另一个是将 PMS 注册到 ServiceManager 中。

PMS 的构造方法大概有 600 行代码，分为 5 个阶段，每个阶段会打印出相应的 EventLog（EventLog 用于打印 Android 系统的事件日志），如下所示。

（1）BOOT_PROGRESS_PMS_START（开始阶段）。

（2）BOOT_PROGRESS_PMS_SYSTEM_SCAN_START（扫描系统阶段）。

（3）BOOT_PROGRESS_PMS_DATAS_CAN_START（扫描 Data 分区阶段）。

（4）BOOT_PROGRESS_PMS_SCAN_END（扫描结束阶段）。

（5）BOOT_PROGRESS_PMS_READY（准备阶段）。

下面对每个阶段进行分析。

2.4.2.1 开始阶段

PMS 的构造方法中会获取一些包管理所需的属性，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
```

```
        LockGuard.installLock(mPackages, LockGuard.INDEX_PACKAGES);
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "create package manager");
        //打印开始阶段日志
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
              SystemClock.uptimeMillis())
        ...
        //用于存储屏幕的相关信息
        mMetrics = new DisplayMetrics();
        //Settings 用于保存所有包的动态设置
        mSettings = new Settings(mPackages);
        //在 Settings 中添加多个默认的 sharedUserId
        mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID,
ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);//1
        mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        ...
        mInstaller = installer;
        //创建 Dex 优化工具类
        mPackageDexOptimizer = new PackageDexOptimizer(installer, mInstallLock,
context, "*dexopt*");
        mDexManager = new DexManager(this, mPackageDexOptimizer, installer,
mInstallLock);
        mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());
        mOnPermissionChangeListeners = new OnPermissionChangeListeners(
              FgThread.get().getLooper());
        getDefaultDisplayMetrics(context, mMetrics);
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "get system config");
        //得到全局系统配置信息
        SystemConfig systemConfig = SystemConfig.getInstance();
        //获取全局的 groupId
        mGlobalGids = systemConfig.getGlobalGids();
        //获取系统权限
        mSystemPermissions = systemConfig.getSystemPermissions();
        mAvailableFeatures = systemConfig.getAvailableFeatures();
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        mProtectedPackages = new ProtectedPackages(mContext);
        //安装 APK 时需要的锁，保护所有对 installd 的访问
        synchronized (mInstallLock) {//2
        //更新 APK 时需要的锁，保护内存中已经解析的包信息等内容
        synchronized (mPackages) {//3
           //创建后台线程 ServiceThread
           mHandlerThread = new ServiceThread(TAG,
```

```
                Process.THREAD_PRIORITY_BACKGROUND, true /*allowIo*/);
            mHandlerThread.start();
            //创建 PackageHandler，绑定到 ServiceThread 的消息队列
            mHandler = new PackageHandler(mHandlerThread.getLooper());//4
            mProcessLoggingHandler = new ProcessLoggingHandler();
            //将 PackageHandler 添加到 Watchdog 的检测集中
            Watchdog.getInstance().addThread(mHandler, WATCHDOG_TIMEOUT);//5

            mDefaultPermissionPolicy = new DefaultPermissionGrantPolicy(this);
            mInstantAppRegistry = new InstantAppRegistry(this);
            //在 Data 分区中创建一些目录
            File dataDir = Environment.getDataDirectory();//6
            mAppInstallDir = new File(dataDir, "app");
            mAppLib32InstallDir = new File(dataDir, "app-lib");
            mAsecInternalPath = new File(dataDir, "app-asec").getPath();
            mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
            //创建多用户管理服务
            sUserManager = new UserManagerService(context, this,
                new UserDataPreparer(mInstaller, mInstallLock, mContext,
mOnlyCore), mPackages);
             ...
              mFirstBoot = !mSettings.readLPw(sUserManager.getUsers(false))//7
          ...
    }
```

在开始阶段创建了很多 PMS 中的关键对象并赋值给 PMS 中的成员变量，下面来简单介绍这些成员变量。

- mSettings：用于保存所有包的动态设置。注释 1 处将系统进程的 sharedUserId 添加到 mSettings 中，sharedUserId 用于进程间共享数据，比如两个 App 之间的数据是不共享的，如果它们有了共同的 sharedUserId，就可以运行在同一个进程中共享数据。
- mInstaller：Installer 继承自 SystemService，和 PMS、AMS 一样是系统服务，PMS 的很多操作都是由 Installer 来完成的，比如 APK 的安装和卸载。在 Installer 内部，通过 IInstalld 和 installd 进行 Binder 通信，由位于 Native 层的 installd 来完成具体的操作。
- systemConfig：用于得到全局系统的配置信息，比如系统的权限就可以通过 SystemConfig 来获取。
- mPackageDexOptimizer：Dex 优化的工具类。
- mHandler（PackageHandler 类型）：PackageHandler 继承自 Handler，在注释 4 处，它绑定了后台线程 ServiceThread 的消息队列。PMS 通过 PackageHandler 驱动 APK

的复制和安装工作。

- 如果 PackageHandler 处理的消息队列过于繁忙，则有可能导致系统卡住，因此在注释 5 处将它添加到 Watchdog 的检测集中。
- Watchdog 主要有两个用途：一个是定时检测系统关键服务（AMS 和 WMS 等）是否可能发生死锁，还有一个是定时检测线程的消息队列是否长时间处于工作状态（可能阻塞并等待了很长时间）。如果出现上述问题，则 Watchdog 会将日志保存起来，必要时还会杀掉自己所在的进程，也就是 SystemServer 进程。
- sUserManager（UserManagerService 类型）：多用户管理服务。

除了创建这些关键对象，在开始阶段还有一些关键代码需要讲解。

- 注释 2 处和注释 3 处加了两个锁，其中 mInstallLock 是安装 APK 时需要的锁，保护所有对 installd 的访问；mPackages 是更新 APK 时需要的锁，保护内存中已经解析的包信息等内容。
- 注释 6 处的代码创建了一些 Data 分区中的子目录，比如/data/app。
- 注释 7 处会解析 packages.xml 等文件的信息，并将其保存到 Settings 的对应字段中。packages.xml 中记录系统中所有安装的应用信息，包括基本信息、签名和权限。如果 packages.xml 中有安装的应用信息，那么注释 7 处 Settings 的 readLPw 方法会返回 true，然而 mFirstBoot 的值为 false，说明 PMS 不是首次被启动。

2.4.2.2　扫描系统阶段

扫描系统阶段的代码如下所示。

```
...
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
...
        //打印扫描系统阶段日志
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
              startTime);
        ...
        //在/system 中创建 framework 目录
       File frameworkDir = new File(Environment.getRootDirectory(), "framework");
        ...
        //扫描/vendor/overlay 目录下的文件
        scanDirTracedLI(new File(VENDOR_OVERLAY_DIR), mDefParseFlags
              | PackageParser.PARSE_IS_SYSTEM
              | PackageParser.PARSE_IS_SYSTEM_DIR
```

```
                    | PackageParser.PARSE_TRUSTED_OVERLAY, scanFlags |
SCAN_TRUSTED_OVERLAY, 0);
            mParallelPackageParserCallback.findStaticOverlayPackages();
            //扫描/system/framework 目录下的文件
            scanDirTracedLI(frameworkDir, mDefParseFlags
                    | PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR
                    | PackageParser.PARSE_IS_PRIVILEGED,
                    scanFlags | SCAN_NO_DEX, 0);
            final File privilegedAppDir = new File(Environment.getRootDirectory(),
"priv-app");
            //扫描 /system/priv-app 目录下的文件
            scanDirTracedLI(privilegedAppDir, mDefParseFlags
                    | PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR
                    | PackageParser.PARSE_IS_PRIVILEGED, scanFlags, 0);
            final File systemAppDir = new File(Environment.getRootDirectory(),
"app");
            //扫描/system/app 目录下的文件
            scanDirTracedLI(systemAppDir, mDefParseFlags
                    | PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);
            File vendorAppDir = new File("/vendor/app");
            try {
               vendorAppDir = vendorAppDir.getCanonicalFile();
            } catch (IOException e) {
            }
            //扫描 /vendor/app 目录下的文件
            scanDirTracedLI(vendorAppDir, mDefParseFlags
                    | PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

            //扫描/oem/app 目录下的文件
            final File oemAppDir = new File(Environment.getOemDirectory(), "app");
            scanDirTracedLI(oemAppDir, mDefParseFlags
                    | PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

            //这个列表代表可能有升级包的系统App
            final List<String> possiblyDeletedUpdatedSystemApps = new
ArrayList<String>();//1
            if (!mOnlyCore) {
              Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();
```

```
                while (psit.hasNext()) {
                    PackageSetting ps = psit.next();
                    if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0) {
                        continue;
                    }
                    final PackageParser.Package scannedPkg = mPackages.get(ps.name);
                    if (scannedPkg != null) {
                        if (mSettings.isDisabledSystemPackageLPr(ps.name)) {//2
                            ...
                            //将这个系统 App 的 PackageSetting 从 PMS 的 mPackages 中移除
                            removePackageLI(scannedPkg, true);
                            //将升级包的路径添加到 mExpectingBetter 列表中
                            mExpectingBetter.put(ps.name, ps.codePath);
                        }
                        continue;
                    }

                    if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
                        ...
                    } else {
                     final PackageSetting disabledPs = mSettings.getDisabledSystemPkgLPr
(ps.name);
                        //这个系统 App 升级包信息在 mDisabledSysPackages 中,但是没有发现这个升
                        //级包存在
                        if (disabledPs.codePath == null || !disabledPs.codePath.
exists()) {//3
                            possiblyDeletedUpdatedSystemApps.add(ps.name);
                        }
                    }
                }
            }
            ...
    }
```

/system 可以称为 System 分区，其中主要存储 Google 和其他厂商提供的 Android 系统相关文件和框架。Android 系统架构分为应用层、应用框架层、系统运行库层（Native 层）、硬件抽象层（HAL 层）和 Linux 内核层，除了 Linux 内核层在 Boot 分区，其他层的代码都在 System 分区。表 2-4 列出了 System 分区的部分子目录。

表 2-4　System 分区的部分子目录

目　录	含　义
app	存放系统 App，包括 Google 内置的 App，也有厂商或者运营商提供的 App
framework	存放应用框架层的 jar 包
priv-app	存放特权 App
lib	存放 so 文件
fonts	存放系统字体文件
media	存放系统的各种声音，比如铃声、提示音及系统启动播放的动画

前面的代码还涉及/vendor 目录，它用来存储厂商对 Android 系统的定制部分。

系统扫描阶段的主要工作有以下三点。

（1）创建/system 的子目录，比如/system/framework、/system/priv-app 和/system/app 等。

（2）扫描系统文件，比如/vendor/overlay、/system/framework 和/system/app 等目录下的文件。

（3）对扫描到的系统文件进行后续处理。

这里主要来讲第 3 点，一次 OTA 升级对于一个系统 App 会出现以下三种情况。

- 这个系统 App 无更新。
- 这个系统 App 有更新。
- 在新的 OTA 版本中，这个系统 App 已经被删除。

在系统 App 升级后，PMS 会将该系统 App 的升级包设置数据（PackageSetting）存储到 Settings 的 mDisabledSysPackages 列表中（具体见 PMS 的 replaceSystemPackageLIF 方法），mDisabledSysPackages 的类型为 ArrayMap<String, PackageSetting>。mDisabledSysPackages 中的信息会被 PMS 保存到 packages.xml 中的<updated-package>标签下（具体见 Settings 的 writeDisabledSysPackageLPr 方法）。

扫描系统阶段代码的注释 2 处说明这个系统 App 有升级包，那么就将该系统 App 的 PackageSetting 从 mDisabledSysPackages 列表中移除，并将系统 App 的升级包路径添加到 mExpectingBetter 列表中等待后续处理。mExpectingBetter 的类型为 ArrayMap<String, File>。

在扫描系统阶段代码的注释 3 处，如果这个系统 App 的升级包信息存储在 mDisabledSysPackages 列表中，但是没有发现这个升级包存在，则将它加入 possiblyDeletedUpdatedSystemApps 列表中，意为“系统 App 的升级包可能被删除”，之所以是

“可能”，是因为系统还没有扫描 Data 分区，只能将它暂放到 possiblyDeletedUpdatedSystemApps 列表中，等扫描完 Data 分区后再进行处理。

2.4.2.3　扫描 Data 分区阶段

扫描 Data 分区阶段的代码如下所示。

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
   ...
   mSettings.pruneSharedUsersLPw();
   //如果没有加密设备，那么开始扫描 Data 分区
   if (!mOnlyCore) {
      //打印扫描 Data 分区阶段日志
      EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
            SystemClock.uptimeMillis());
      //扫描/data/app 目录下的文件
      scanDirTracedLI(mAppInstallDir, 0, scanFlags | SCAN_REQUIRE_KNOWN, 0);
      //扫描/data/app-private 目录下的文件
      scanDirTracedLI(mDrmAppPrivateInstallDir, mDefParseFlags
            | PackageParser.PARSE_FORWARD_LOCK,
            scanFlags | SCAN_REQUIRE_KNOWN, 0);
      //扫描完 Data 分区后，处理 possiblyDeletedUpdatedSystemApps 列表
      for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
         PackageParser.Package deletedPkg = mPackages.get(deletedAppName);
         //从 mSettings.mDisabledSysPackages 变量中移除此应用
         mSettings.removeDisabledSystemPackageLPw(deletedAppName);
         String msg;
        //1: 如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，则说明其是残留的 App 信息，
        //后续会删除它的数据
         if (deletedPkg == null) {
            msg = "Updated system package " + deletedAppName
                  + " no longer exists; it's data will be wiped";
         } else {
         //2: 如果这个系统 App 在 mPackages 中，则说明其存在于 Data 分区中，不属于系统 App，
         //那么移除其系统权限
            msg = "Updated system app + " + deletedAppName
                  + " no longer present; removing system privileges for "
                  + deletedAppName;
            deletedPkg.applicationInfo.flags &= ~ApplicationInfo.FLAG_SYSTEM;
            PackageSetting deletedPs = mSettings.mPackages.get(deletedAppName);
            deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;
         }
```

```
            logCriticalInfo(Log.WARN, msg);
        }
        //遍历 mExpectingBetter 列表
        for (int i = 0; i < mExpectingBetter.size(); i++) {
            final String packageName = mExpectingBetter.keyAt(i);
            if (!mPackages.containsKey(packageName)) {
                //得到系统 App 的升级包路径
                final File scanFile = mExpectingBetter.valueAt(i);
                logCriticalInfo(Log.WARN, "Expected better " + packageName
                        + " but never showed up; reverting to system");
                int reparseFlags = mDefParseFlags;
                //3: 根据系统 App 所在的目录设置扫描的解析参数
                if (FileUtils.contains(privilegedAppDir, scanFile)) {
                    reparseFlags = PackageParser.PARSE_IS_SYSTEM
                            | PackageParser.PARSE_IS_SYSTEM_DIR
                            | PackageParser.PARSE_IS_PRIVILEGED;
                }
                ...
                //将 packageName 对应的包设置数据（PackageSetting）添加到 mSettings 的
                //mPackages 中
                mSettings.enableSystemPackageLPw(packageName);//4
                try {
                    //扫描系统 App 的升级包
                  scanPackageTracedLI(scanFile, reparseFlags, scanFlags, 0, null);//5
                } catch (PackageManagerException e) {
                    Slog.e(TAG, "Failed to parse original system package: "
                            + e.getMessage());
                }
            }
        }
    }
    //清除 mExpectingBetter 列表
    mExpectingBetter.clear();
...
}
```

/data 可以称为 Data 分区，它用来存储所有用户的个人数据和配置文件。表 2-5 列出了 Data 分区部分子目录。

表 2-5　Data 分区部分子目录

目　录	含　义
app	存储用户自己安装的 App
data	存储所有已安装的 App 数据，每个 App 都有自己单独的子目录

续表

目　录	含　义
app-private	App 的私有存储空间
app-lib	存储所有 App 的 Jni 库
system	存放系统配置文件
anr	用于存储 ANR 发生时系统生成的 traces.txt 文件

扫描 Data 分区阶段的主要工作有以下三点。

（1）扫描/data/app 和/data/app-private 目录下的文件。

（2）遍历 possiblyDeletedUpdatedSystemApps 列表，在扫描 Data 分区阶段代码的注释 1 处，如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，则说明其是残留的 App 信息，后续会删除它的数据。在注释 2 处，如果这个系统 App 的包信息在 mPackages 中，则说明其存在于 Data 分区中，不属于系统 App，那么移除其系统权限。

（3）遍历 mExpectingBetter 列表。注释 3 处根据系统 App 所在的目录设置扫描的解析参数。注释 4 处的方法内部会将 packageName 对应的包设置数据（PackageSetting）添加到 mSettings 的 mPackages 中。注释 5 处扫描系统 App 的升级包，最后清除 mExpectingBetter 列表。

2.4.2.4　扫描结束阶段

扫描结束阶段的代码如下所示。

```
//打印扫描结束阶段日志
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
              SystemClock.uptimeMillis());
        Slog.i(TAG, "Time to scan packages: "
              + ((SystemClock.uptimeMillis()-startTime)/1000f)
              + " seconds");
        int updateFlags = UPDATE_PERMISSIONS_ALL;
        //如果当前平台的SDK版本和上次启动时的SDK版本不同，则重新更新APK的授权
        if (ver.sdkVersion != mSdkVersion) {
           Slog.i(TAG, "Platform changed from " + ver.sdkVersion + " to "
                 + mSdkVersion + "; regranting permissions for internal storage");
           updateFlags |= UPDATE_PERMISSIONS_REPLACE_PKG |
UPDATE_PERMISSIONS_REPLACE_ALL;
        }
        updatePermissionsLPw(null, null, StorageManager.UUID_PRIVATE_INTERNAL,
updateFlags);
```

```
            ver.sdkVersion = mSdkVersion;
            //如果是第一次启动或者是Android M升级后的第一次启动，则需要初始化所有用户定义的默认
            //首选App
            if (!onlyCore && (mPromoteSystemApps || mFirstBoot)) {
                for (UserInfo user : sUserManager.getUsers(true)) {
                    mSettings.applyDefaultPreferredAppsLPw(this, user.id);
                    applyFactoryDefaultBrowserLPw(user.id);
                    primeDomainVerificationsLPw(user.id);
                }
            }
            ...
            //OTA升级后的第一次启动，会清除代码缓存目录
            if (mIsUpgrade && !onlyCore) {
                Slog.i(TAG, "Build fingerprint changed; clearing code caches");
                for (int i = 0; i < mSettings.mPackages.size(); i++) {
                    final PackageSetting ps = mSettings.mPackages.valueAt(i);
                    if (Objects.equals(StorageManager.UUID_PRIVATE_INTERNAL,
ps.volumeUuid)) {
                        clearAppDataLIF(ps.pkg, UserHandle.USER_ALL,
                            StorageManager.FLAG_STORAGE_DE |
StorageManager.FLAG_STORAGE_CE | Installer.FLAG_CLEAR_CODE_CACHE_ONLY);
                    }
                }
                ver.fingerprint = Build.FINGERPRINT;
            }
            ...
            //把Settings的内容保存到packages.xml中
            mSettings.writeLPr();
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
```

扫描结束阶段的主要工作有以下几点。

（1）如果当前平台的 SDK 版本和上次启动时的 SDK 版本不同，则重新更新 APK 的授权。

（2）如果是第一次启动或者是 Android M 升级后的第一次启动，则需要初始化所有用户定义的默认首选 App。

（3）OTA 升级后的第一次启动，会清除代码缓存目录。

（4）把 Settings 的内容保存到 packages.xml 中，这样以后 PMS 再创建时会读到此前保存的 Settings 的内容。

2.4.2.5　准备阶段

准备阶段的代码如下所示。

```
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
            SystemClock.uptimeMillis());
    ...
    mInstallerService = new PackageInstallerService(context, this);//1
    ...
    Runtime.getRuntime().gc();//2
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "loadFallbacks");
    FallbackCategoryProvider.loadFallbacks();
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    mInstaller.setWarnIfHeld(mPackages);
    LocalServices.addService(PackageManagerInternal.class, new
PackageManagerInternalImpl());//3
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
}
```

注释 1 处创建了 PackageInstallerService，PackageInstallerService 是用于管理安装会话的服务，它会为每次的安装过程分配一个 SessionId。注释 2 处进行了一次垃圾收集。注释 3 处将 PackageManagerInternalImpl（PackageManager 的本地服务）添加到 LocalServices 中，LocalServices 用于存储运行在当前进程中的本地服务。

2.5　APK 解析过程

在前面的章节中，我介绍了 PackageInstaller 的初始化和安装 APK 的过程、PMS 处理 APK 的安装和 PMS 的创建过程，在这些过程中经常会涉及一个类，那就是 PackageParser，它用来在 APK 的安装过程中解析 APK，那么 APK 是如何被解析的呢？本节中会给出答案。

2.5.1　引入 PackageParser

Android 世界中有很多包，比如应用程序的 APK、Android 运行环境的 jar 包（比如 framework.jar）和组成 Android 系统的各种动态库 so 等，由于包的种类和数量较多，这就需要进行包管理，但是包管理需要在内存中进行，而这些包都是以静态文件的形式存在的，需要一个工具类将这些包转换为内存中的数据结构，这个工具就是包解析器 PackageParser。

在 2.3 节中，我们得知安装 APK 时需要调用 PMS 的 installPackageLI 方法，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {
    ...
    PackageParser pp = new PackageParser();//1
    pp.setSeparateProcesses(mSeparateProcesses);
    pp.setDisplayMetrics(mMetrics);
    pp.setCallback(mPackageParserCallback);
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parsePackage");
    final PackageParser.Package pkg;
    try {
        pkg = pp.parsePackage(tmpPackageFile, parseFlags);//2
    }
    ...
}
```

可以看到安装 APK 时，需要先在注释 1 处创建 PackageParser，然后在注释 2 处调用 PackageParser 的 parsePackage 方法来解析 APK。

2.5.2 PackageParser 解析 APK

Android 5.0 引入了 Split APK 机制，其目的是解决 65536 上限及 APK 安装包越来越大等问题。Split APK 机制可以将一个 APK 拆分成多个独立的 APK。

在引入了 Split APK 机制后，APK 有以下两种分类。

- Single APK：一个完整的 APK 安装文件，即 base APK。Android 称其为 Monolithic。
- Mutiple APK：在一个文件目录中安装文件，其内部有多个被拆分的 APK，这些 APK 由一个 base APK 和一个或多个 split APK 组成。Android 称其为 Cluster。

了解了 APK 后，继续学习 PackageParser 解析 APK，查看 PackageParser 的 parsePackage 方法，如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
public Package parsePackage(File packageFile, int flags, boolean useCaches)
        throws PackageParserException {
    Package parsed = useCaches ? getCachedResult(packageFile, flags) : null;
    if (parsed != null) {
        return parsed;
    }
    if (packageFile.isDirectory()) {//1
```

```
            parsed = parseClusterPackage(packageFile, flags);
        } else {
            parsed = parseMonolithicPackage(packageFile, flags);
        }
        cacheResult(packageFile, flags, parsed);

        return parsed;
    }
```

在注释 1 处，如果要解析的 packageFile 是一个目录，则说明 APK 的种类为 Mutiple APK，这时就需要调用 parseClusterPackage 方法来解析，而如果是 Single APK，则调用 parseMonolithicPackage 方法来解析。这里以复杂的 parseClusterPackage 方法为例，了解了这个方法后，自然也能够看懂 parseMonolithicPackage 方法。PackageParser 解析 APK 的简图如图 2-6 所示。

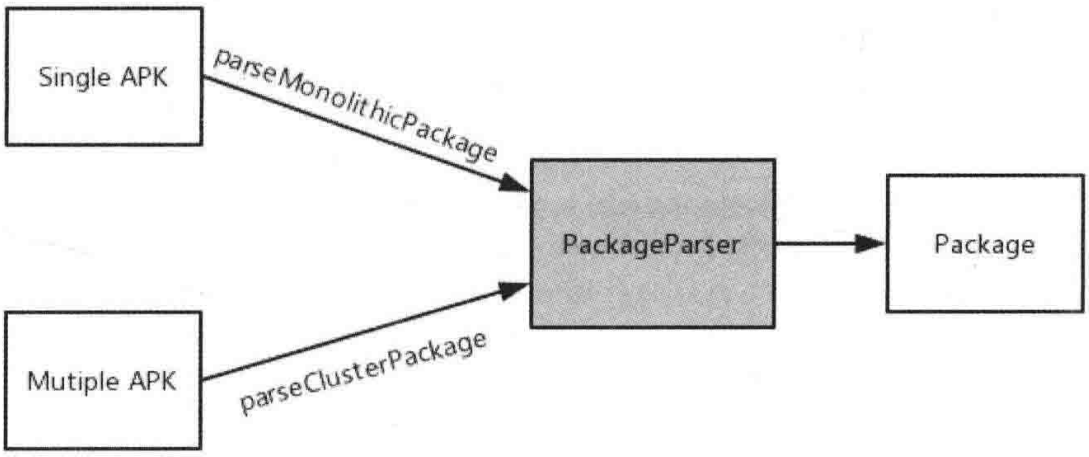


图 2-6　PackageParser 解析 APK 的简图

来查看 parseClusterPackage 方法是如何解析 APK 的。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
    private Package parseClusterPackage(File packageDir, int flags) throws
PackageParserException {
        final PackageLite lite = parseClusterPackageLite(packageDir, 0);//1
        if (mOnlyCoreApps && !lite.coreApp) {//2
            throw new
PackageParserException(INSTALL_PARSE_FAILED_MANIFEST_MALFORMED,
                    "Not a coreApp: " + packageDir);
        }
        ...
        try {
            final AssetManager assets = assetLoader.getBaseAssetManager();
```

```
            final File baseApk = new File(lite.baseCodePath);
            final Package pkg = parseBaseApk(baseApk, assets, flags);//3
            if (pkg == null) {
               throw new PackageParserException(INSTALL_PARSE_FAILED_NOT_APK,
                     "Failed to parse base APK: " + baseApk);
            }
            if (!ArrayUtils.isEmpty(lite.splitNames)) {
               final int num = lite.splitNames.length;//4
               pkg.splitNames = lite.splitNames;
               pkg.splitCodePaths = lite.splitCodePaths;
               pkg.splitRevisionCodes = lite.splitRevisionCodes;
               pkg.splitFlags = new int[num];
               pkg.splitPrivateFlags = new int[num];
               pkg.applicationInfo.splitNames = pkg.splitNames;
               pkg.applicationInfo.splitDependencies = splitDependencies;
               for (int i = 0; i < num; i++) {
                  final AssetManager splitAssets =
assetLoader.getSplitAssetManager(i);
                  parseSplitApk(pkg, i, splitAssets, flags);//5
               }
            }
            pkg.setCodePath(packageDir.getAbsolutePath());
            pkg.setUse32bitAbi(lite.use32bitAbi);
            return pkg;
         } finally {
            IoUtils.closeQuietly(assetLoader);
         }
      }
```

注释 1 处调用了 parseClusterPackageLite 方法，用于轻量级解析目录文件，之所以要轻量级解析，是因为解析 APK 是一个复杂且耗时的工作，这里的逻辑并不需要 APK 的所有信息。parseClusterPackageLite 方法内部会通过 parseApkLite 方法解析每个 Mutiple APK，得到每个 Mutiple APK 对应的 ApkLite（轻量级 APK 信息），然后将这些 ApkLite 封装为一个 PackageLite（轻量级包信息）并返回。

注释 2 处的 mOnlyCoreApps 用于指示 PackageParser 是否只解析“核心”应用，“核心”应用指的是 AndroidManifest 中属性 coreApp 的值为 true，只解析“核心”应用是为了创建一个极简的启动环境。在创建 PMS 时 mOnlyCoreApps 一路传递过来，如果我们加密了设备，mOnlyCoreApps 的值就为 true，另外可以通过 PackageParser 的 setOnlyCoreApps 方法来设置 mOnlyCoreApps 的值。

. lite.coreApp 表示当前包是否包含“核心”应用，如果不满足注释 2 的条件就会抛出

异常。

注释 3 处的 parseBaseApk 方法用于解析 base APK，注释 4 处获取了 split APK 的数量，根据这个数量在注释 5 处遍历地调用 parseSplitApk 来解析每个 split APK。这里主要查看 parseBaseApk 方法，如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
    private Package parseBaseApk(File apkFile, AssetManager assets, int flags)
            throws PackageParserException {
        final String apkPath = apkFile.getAbsolutePath();
        String volumeUuid = null;
        if (apkPath.startsWith(MNT_EXPAND)) {
            final int end = apkPath.indexOf('/', MNT_EXPAND.length());
            volumeUuid = apkPath.substring(MNT_EXPAND.length(), end);//1
        }
        ...
        Resources res = null;
        XmlResourceParser parser = null;
        try {
            res = new Resources(assets, mMetrics, null);
            parser = assets.openXmlResourceParser(cookie,
ANDROID_MANIFEST_FILENAME);
            final String[] outError = new String[1];
            final Package pkg = parseBaseApk(apkPath, res, parser, flags,
outError);//2
            if (pkg == null) {
                throw new PackageParserException(mParseError,
                        apkPath + " (at " + parser.getPositionDescription() + "): " +
outError[0]);
            }
            pkg.setVolumeUuid(volumeUuid);//3
            pkg.setApplicationVolumeUuid(volumeUuid);//4
            pkg.setBaseCodePath(apkPath);
            pkg.setSignatures(null);
            return pkg;
        } catch (PackageParserException e) {
            throw e;
        }
        ...
    }
```

在注释 1 处，如果 APK 的路径以/mnt/expand/开头，就截取该路径来获取 volumeUuid，注释 3 处的代码用于以后标识这个解析后的 Package，注释 4 处的代码用于标识该 App 所

在的存储卷 UUID。

注释 2 处又调用了 parseBaseApk 的重载方法，可以看出当前的 parseBaseApk 方法主要用于获取和设置 volumeUuid。parseBaseApk 的重载方法如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
    private Package parseBaseApk(String apkPath, Resources res, XmlResourceParser
parser, int flags, String[] outError) throws XmlPullParserException, IOException {
        ...
        final Package pkg = new Package(pkgName);//1
        //从资源中提取自定义属性集 com.android.internal.R.styleable.AndroidManifest
        //并得到 TypedArray
        TypedArray sa = res.obtainAttributes(parser,
              com.android.internal.R.styleable.AndroidManifest);//2
        //使用 Typedarray 获取 AndroidManifest 中的 versionCode 并赋值给 Package 的对应属性
        pkg.mVersionCode = pkg.applicationInfo.versionCode = sa.getInteger(
              com.android.internal.R.styleable.AndroidManifest_versionCode, 0);
        pkg.baseRevisionCode = sa.getInteger(
              com.android.internal.R.styleable.AndroidManifest_revisionCode, 0);
        pkg.mVersionName = sa.getNonConfigurationString(
              com.android.internal.R.styleable.AndroidManifest_versionName, 0);
        if (pkg.mVersionName != null) {
           pkg.mVersionName = pkg.mVersionName.intern();
        }
        pkg.coreApp = parser.getAttributeBooleanValue(null, "coreApp", false);//3
        //获取资源后要回收
        sa.recycle();
        return parseBaseApkCommon(pkg, null, res, parser, flags, outError);
    }
```

注释 1 处创建了 Package 对象，注释 2 处从资源中提取了自定义属性集 com.android.internal.R.styleable.AndroidManifest 并得到 TypedArray，这个属性集所在的源码位置为 frameworks/base/core/res/res/values/attrs_manifest.xml。接着用 TypedArray 读取 APK 的 AndroidManifest 中的 versionCode、revisionCode 和 versionName 的值并将这些值赋给 Package 的对应属性。注释 3 处读取了 APK 的 AndroidManifest 中的 coreApp 的值。

最后会调用 parseBaseApkCommon 方法，这个方法主要用来解析 APK 的 AndroidManifest 中的各个标签，比如 application、permission、uses-sdk 和 feature-group 等，其中四大组件的标签在 application 标签下，解析 application 标签的方法为 parseBaseApplication，如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
    private boolean parseBaseApplication(Package owner, Resources res,
            XmlResourceParser parser, int flags, String[] outError)
        throws XmlPullParserException, IOException {
        ...
        while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
              && (type != XmlPullParser.END_TAG || parser.getDepth() > innerDepth))
{
            if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
               continue;
            }
            String tagName = parser.getName();
            if (tagName.equals("activity")) {//1
               Activity a = parseActivity(owner, res, parser, flags, outError, false,
                     owner.baseHardwareAccelerated);//2
               if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                  return false;
               }
               owner.activities.add(a);//3
            } else if (tagName.equals("receiver")) {
               Activity a = parseActivity(owner, res, parser, flags, outError, true,
false);
               if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                  return false;
               }
               owner.receivers.add(a);
            } else if (tagName.equals("service")) {
               Service s = parseService(owner, res, parser, flags, outError);
               if (s == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                  return false;
               }
               owner.services.add(s);
            } else if (tagName.equals("provider")) {
               Provider p = parseProvider(owner, res, parser, flags, outError);
               if (p == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                  return false;
               }
               owner.providers.add(p);
            ...
```

```
            }
        }
    ...
}
```

parseBaseApplication 方法大概有 500 行代码，这里只截取了与解析四大组件相关的代码。如果注释 1 处的标签名为 activity，则调用注释 2 处的 parseActivity 方法解析 activity 标签并得到一个 Activity 对象（PackageParser 的静态内部类），这个方法有 300 多行代码，仅解析一个 activity 标签就如此烦琐，activity 标签只是 application 众多标签中的一个，而 application 只是 AndroidManifest 众多标签中的一个，这让我们更容易理解为什么此前解析 APK 时要使用轻量级解析。注释 3 处将解析得到的 Activity 对象保存在 Package 的列表 activities 中。其他四大组件也是类似的逻辑。

PackageParser 解析 APK 的代码逻辑非常庞大，但基本上了解了本节所讲的内容就足够了，感兴趣的读者可以自行查看源码。

parseBaseApk 方法主要的解析结构如图 2-7 所示。

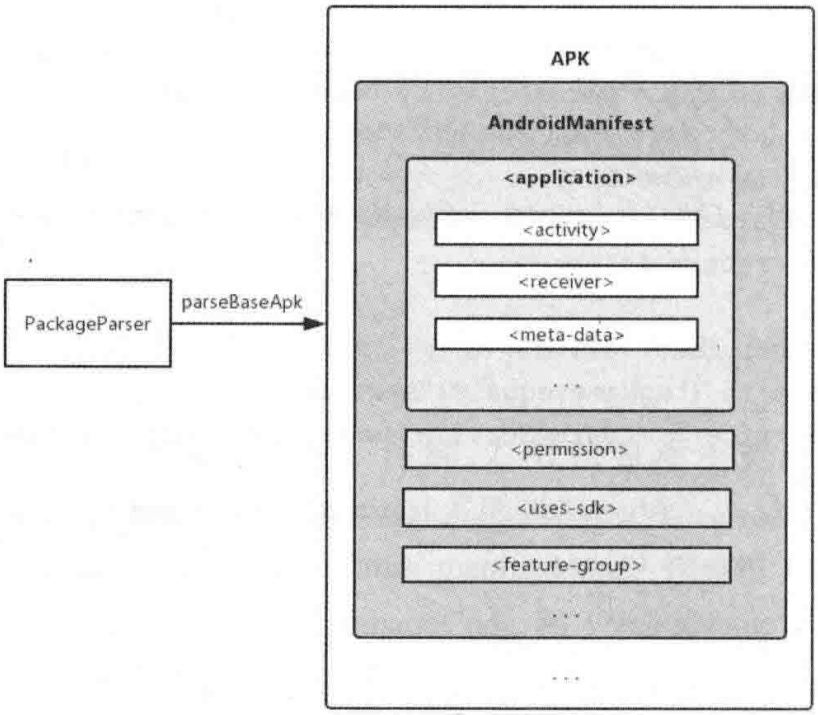


图 2-7　parseBaseApk 方法主要的解析结构

2.5.3　Package 的数据结构

包被解析后，最终显示内存中的是 Package，Package 是 PackageParser 的内部类，它的部分成员变量如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
public final static class Package implements Parcelable {
    public String packageName;
    public String manifestPackageName;
    public String[] splitNames;
    public String volumeUuid;
    public String codePath;
    public String baseCodePath;
    ...
    public ApplicationInfo applicationInfo = new ApplicationInfo();
    public final ArrayList<Permission> permissions = new ArrayList<Permission>(0);
    public final ArrayList<PermissionGroup> permissionGroups = new
ArrayList<PermissionGroup>(0);
    public final ArrayList<Activity> activities = new ArrayList<Activity>(0);//1
    public final ArrayList<Activity> receivers = new ArrayList<Activity>(0);
    public final ArrayList<Provider> providers = new ArrayList<Provider>(0);
    public final ArrayList<Service> services = new ArrayList<Service>(0);
    public final ArrayList<Instrumentation> instrumentation = new
ArrayList<Instrumentation>(0);
...
}
```

在注释 1 处，activities 列表中存储了类型为 Activity 的对象，需要注意这个 Activity 并不是我们常用的 Activity，而是 PackageParser 的静态内部类，Package 中的其他列表也都是如此。Package 的数据结构简图如图 2-8 所示。

从这个简图中可以了解 Package 的数据结构是如何设计的。

- Package 中有许多组件，比如 Activity、Provider 和 Permission 等，它们都继承自基类 Component。
- 每个组件都包含一个 info 数据，比如 Activity 类中包含了成员变量 ActivityInfo，这个 ActivityInfo 才是真正的 Activity 数据。

- 四大组件的标签内可能包含<intent-filter>，其用来过滤 Intent 信息，因此需要 IntentInfo 来保存组件的 Intent 信息，组件基类 Component 依赖 IntentInfo，IntentInfo 有三个子类，即 ActivityIntentInfo、ServiceIntentInfo 和 ProviderIntentInfo，不同组件依赖的 IntentInfo 会有所不同，比如 Activity 继承自 Component<ActivityIntentInfo>，Permission 继承自 Component<IntentInfo>。

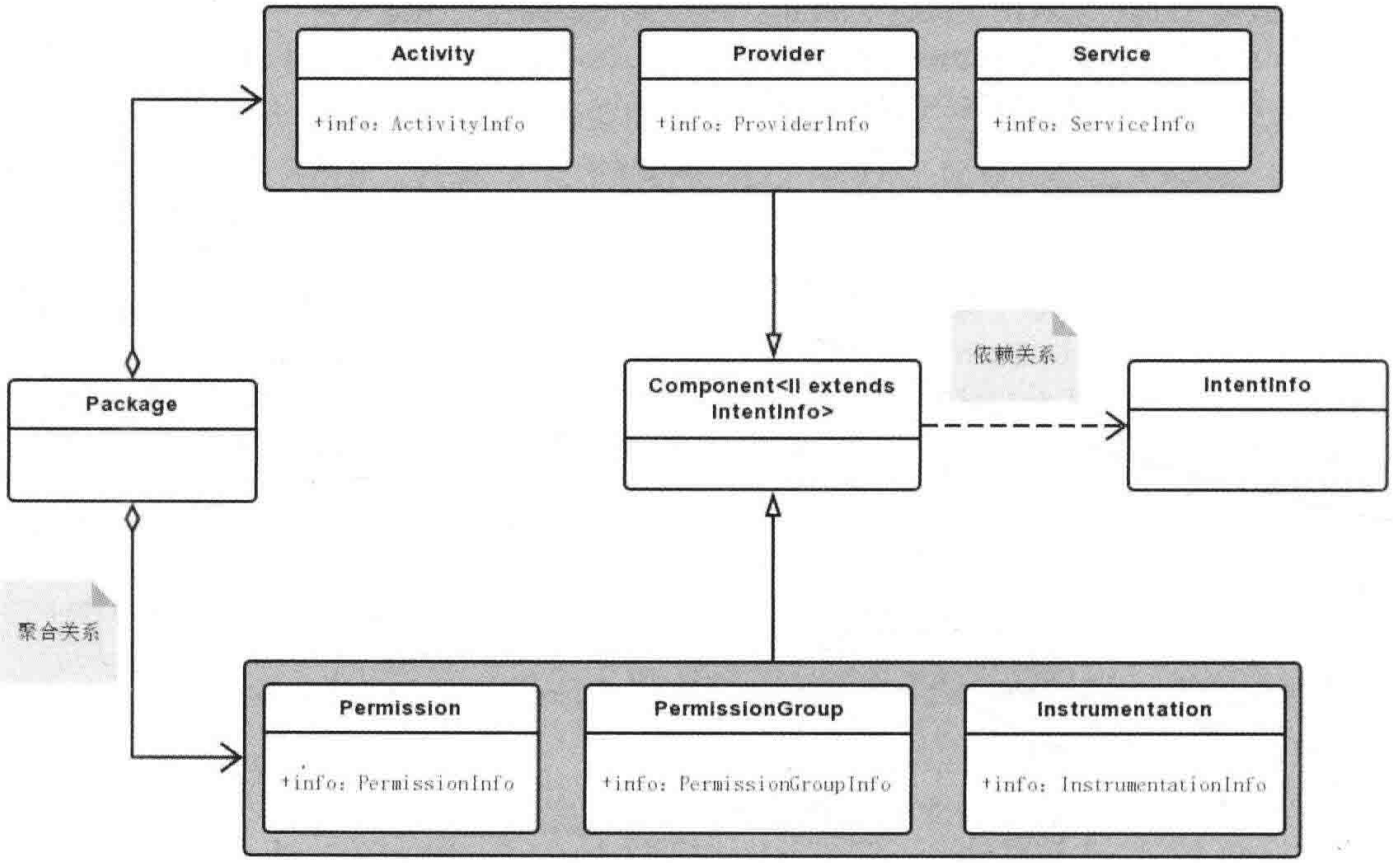


图 2-8　Package 的数据结构简图

最终解析的数据会封装到 Package 中，除此之外，在解析过程中还有两个轻量级数据结构，即 ApkLite 和 PackageLite，因为这两个数据结构和 Package 没有太大的关联，所以就没有在图 2-8 中体现。

2.6　本章小结

AMS、WMS 和 PMS 是现阶段应用开发人员需要掌握的三大系统服务，在《Android 进阶解密》一书中我介绍了 AMS 和 WMS，没有介绍 PMS，因此在本书中进行了补充。

如果直接讲 PMS 会比较难以理解，因此需要一个切入点，这个切入点就是常见的 APK 的安装过程，本章除了介绍 APK 的安装过程，还介绍了 PMS 的创建过程和 APK 的解析过程，这些都是包管理机制的核心知识点，应用开发人员掌握这些知识点就足够用了，如果想要深入了解，则还可以查看具体的源码。

第 3 章

理解输入系统和 IMS

关联章节：

《Android 进阶之光》 第 3 章 View 体系与自定义 View

《Android 进阶解密》 第 2 章 Android 系统启动

《Android 进阶解密》 第 7 章 理解 WindowManager

《Android 进阶解密》 第 8 章 理解 WindowManagerService

《Android 进阶解密》 第 9 章 JNI 原理

很多读者可能会提出这样的问题，输入系统是否和 View 的事件分发有些关联呢？确实是有些关联的，只不过 View 事件分发只能算是输入系统事件传递的一部分。本章所讲解的输入系统主要是我们不经常接触但还需要了解的内容。

输入系统的核心是 InputManagerService，可以说它是除系统服务 AMS、WMS 和 PMS 之外，应用开发人员还需要掌握的系统服务之一，至于原因，我们学习了本章之后就知道了。

3.1　输入事件传递流程的组成部分

输入系统是外界与 Android 设备交互的基础，仅凭输入系统是无法完成输入事件传递的，因此需要输入系统和 Android 系统的其他成员来共同实现输入事件传递。输入系统事件传递需要经过如图 3-1 所示的流程。

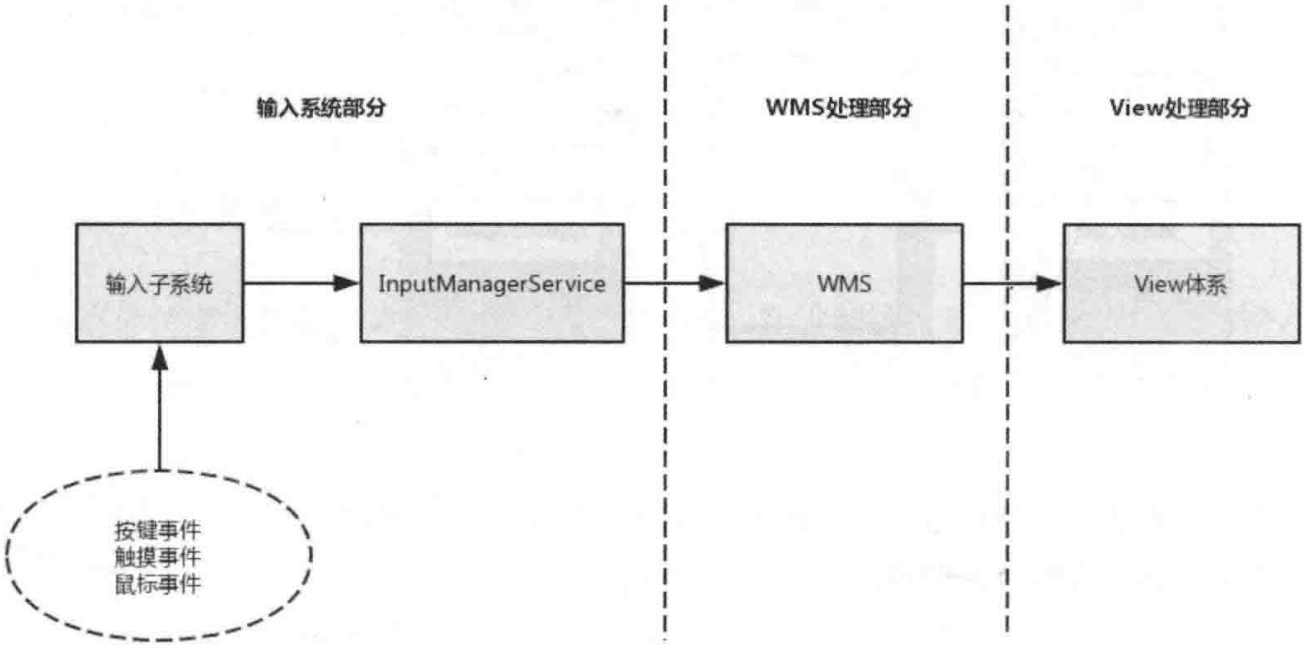


图 3-1　输入系统事件传递流程

输入事件传递流程可以分为三个部分，分别是输入系统部分、WMS 处理部分和 View 处理部分。下面分别对这三个部分进行简单介绍。

1. 输入系统部分

输入系统部分主要分为输入子系统和 InputManagerService（以下简称 IMS），在 Android 中还有一个 IMS（IP Multimedia Subsystem），意为 IP 多媒体子系统。

Android 的输入设备有很多种，比如触摸屏、键盘、鼠标、游戏手柄和操纵杆等，其中应用开发接触最多的是触摸屏。当输入设备可用时，Linux 内核会在/dev/input 中创建对应的设备节点。用户操作这些输入设备时会产生各种事件，比如按键事件、触摸事件和鼠标事件等。

输入事件产生的原始信息会被 Linux 内核中的输入子系统采集，原始信息由 Kernel space 的驱动层一直传递到 User space 的设备节点。

Android 提供了 getevent 和 sendevent 两个工具帮助开发者从设备节点读取输入事件和写入输入事件。

输入系统部分如图 3-2 所示。

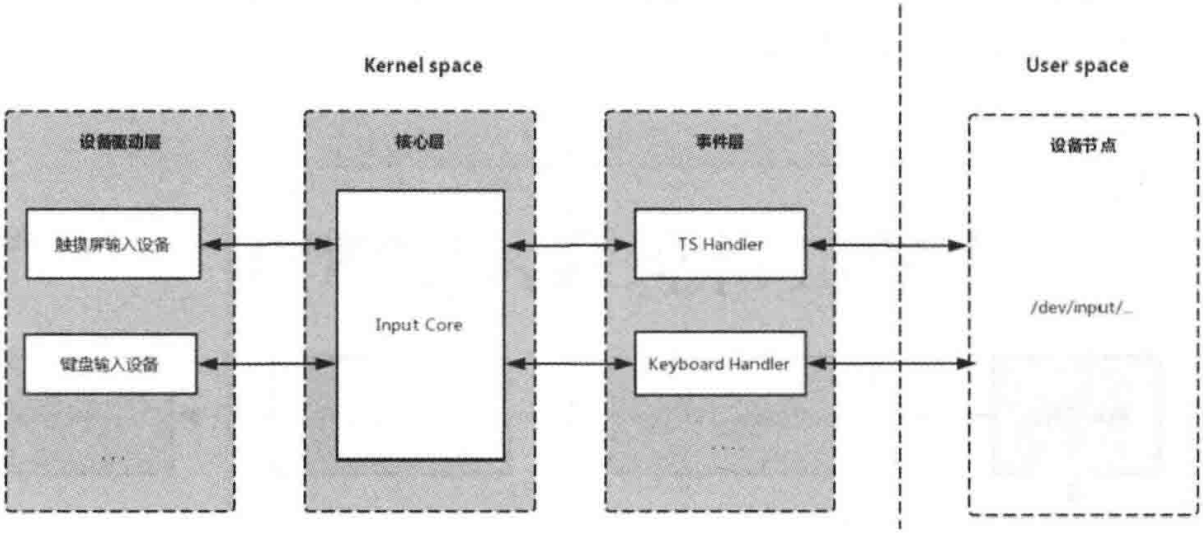


图 3-2　输入系统部分

IMS 所做的工作就是监听/dev/input 下的所有设备节点，当设备节点有数据时会将数据进行加工处理并找到合适的窗口，将输入事件派发给它。

2. WMS 处理部分

在《Android 进阶解密》的第 8 章中曾介绍过 WMS 的职责有四种，如图 3-3 所示。

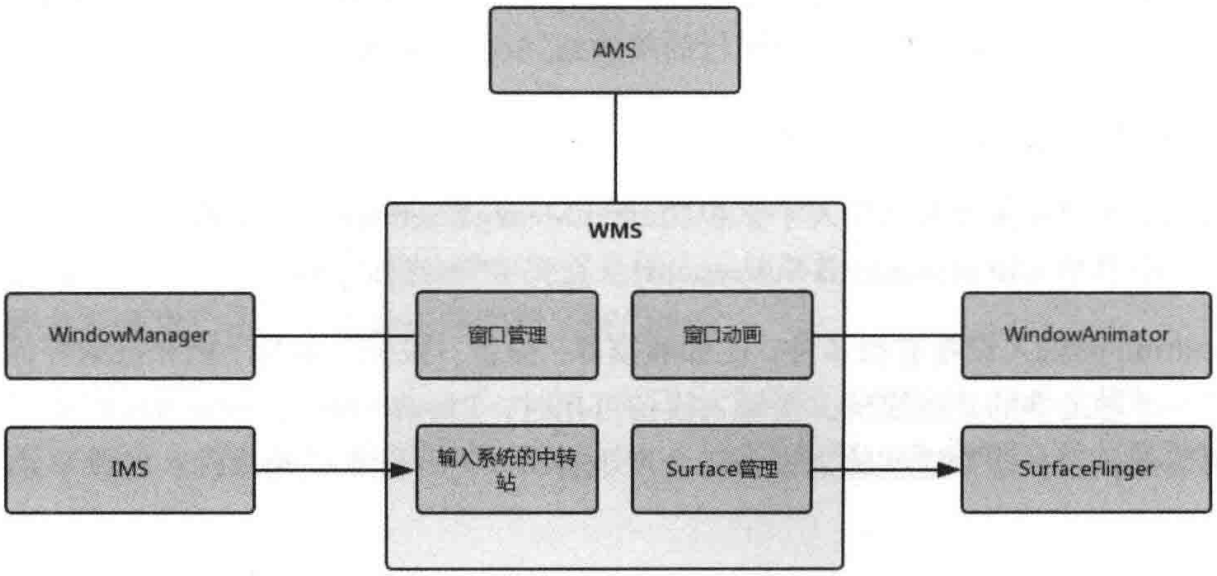


图 3-3　WMS 的职责

WMS 的职责之一就是输入系统的中转站，WMS 作为窗口的管理者，会配合 IMS 将输入事件交由合适的窗口来处理。

3. View 处理部分

View 处理部分应该是大家很熟悉的内容了，在一般情况下，输入事件最终会交由 View 来处理，应用开发人员可以通过一些回调方法轻松得到这个事件的封装类并对其进行处理，比如 onTouchEvent(MotionEvent ev)方法。

3.2　IMS 的诞生

输入事件传递流程的组成部分我们已经了解了，在讲解输入系统部分中 IMS 对输入事件的处理之前，需要了解 IMS 的诞生。

3.2.1　SystemServer 处理部分

与 AMS、WMS、PMS 一样，IMS 是在 SystemServer 进程中创建的，SystemServer 进程用来创建系统服务，关于 SystemServer 进程的内容可以查看《Android 进阶解密》的 2.3 节。

下面从 SystemServer 的入口方法 main 方法开始讲起，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
      new SystemServer().run();
  }
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {
  ...
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices();//1
        //启动核心服务
        startCoreServices();//2
        //启动其他服务
```

```
        startOtherServices();//3
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "*****************************************");
        Slog.e("System", "************ Failure starting system services", ex);
        throw ex;
    } finally {
        traceEnd();
    }
    ...
}
```

注释 1 处的 startBootstrapServices 方法用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService 和 PackageManagerService 等服务。注释 2 处的 startCoreServices 方法则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService 等服务。注释 3 处的 startOtherServices 方法启动了 CameraService、AlarmManagerService 和 VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 1、2、3 中的方法可以看出，官方把系统服务分为三种类型，分别是引导服务、核心服务和其他服务，其中，其他服务是一些非紧要的或不需要立即启动的服务。系统服务总共有 100 多个，我们熟知的 AMS 和 PMS 属于引导服务，WMS 属于其他服务。

本节要讲的 IMS 属于其他服务，表 3-1 列出了部分其他服务及其作用。

表 3-1　部分其他服务及其作用

其他服务	作　用
CameraService	摄像头相关服务
AlarmManagerService	全局定时器管理服务
InputManagerService	管理输入事件
WindowManagerService	窗口管理服务
VrManagerService	VR 模式管理服务
BluetoothService	蓝牙管理服务
NotificationManagerService	通知管理服务
DeviceStorageMonitorService	存储相关管理服务
LocationManagerService	定位管理服务
AudioService	音频相关管理服务
……	……

查看启动其他服务的注释 3 处的 startOtherServices 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
    private void startOtherServices() {
    ...
            inputManager = new InputManagerService(context);//1
            traceEnd();
            traceBeginAndSlog("StartWindowManagerService");
            ConcurrentUtils.waitForFutureNoInterrupt(mSensorServiceStart,
START_SENSOR_SERVICE);
            mSensorServiceStart = null;
            wm = WindowManagerService.main(context, inputManager,
                  mFactoryTestMode != FactoryTest.FACTORY_TEST_LOW_LEVEL,
                  !mFirstBoot, mOnlyCore, new PhoneWindowManager());//2
            ServiceManager.addService(Context.WINDOW_SERVICE, wm);
            ServiceManager.addService(Context.INPUT_SERVICE, inputManager);
            traceEnd();
    ...

    }
```

注释 1 处创建了 IMS，注释 2 处执行了 WMS 的 main 方法，其内部会创建 WMS。需要注意的是，main 方法的其中一个参数就是注释 1 处创建的 IMS，WMS 是输入系统的中转站，其内部包含了 IMS 引用。然后将 WMS 和 IMS 添加到 ServiceManager 中进行统一管理。

3.2.2　InputManagerService 构造方法

接着来查看 IMS 的构造方法。

frameworks/base/services/core/java/com/android/server/input/InputManagerService.java

```
   public InputManagerService(Context context) {
         this.mContext = context;
         this.mHandler = new InputManagerHandler(DisplayThread.get().
getLooper());//1
         mUseDevInputEventForAudioJack =
               context.getResources().getBoolean(R.bool.config_
useDevInputEventForAudioJack);
         Slog.i(TAG, "Initializing input manager, mUseDevInputEventForAudioJack="
               + mUseDevInputEventForAudioJack);
         mPtr = nativeInit(this, mContext, mHandler.getLooper().getQueue());//2
```

```
        ...
    }
```

注释 1 处用 android.display 线程的 Looper 创建了 InputManagerHandler，这样 InputManagerHandler 会运行在 android.display 线程中，android.display 线程是系统共享的单例前台线程，这个线程内部执行了 WMS 的创建，具体见《Android 进阶解密》的 8.2 节。

注释 2 处调用了 nativeInit 方法，很明显要通过 JNI 调用 Native 方法，如下所示。

frameworks/base/services/core/jni/com_android_server_input_InputManagerService.cpp

```
static jlong nativeInit(JNIEnv* env, jclass /* clazz */,
        jobject serviceObj, jobject contextObj, jobject messageQueueObj) {
    sp<MessageQueue> messageQueue = android_os_MessageQueue_getMessageQueue(env,
messageQueueObj);
    if (messageQueue == NULL) {
        jniThrowRuntimeException(env, "MessageQueue is not initialized.");
        return 0;
    }
    NativeInputManager* im = new NativeInputManager(contextObj, serviceObj,
            messageQueue->getLooper());//1
    im->incStrong(0);
    return reinterpret_cast<jlong>(im);
}
```

注释 1 处创建了 NativeInputManager，调用 reinterpretcast 运算符将 NativeInputManager 指针强制转换并返回（重新解释比特位）。NativeInputManager 的构造函数如下所示。

frameworks/base/services/core/jni/com_android_server_input_InputManagerService.cpp

```
NativeInputManager::NativeInputManager(jobject contextObj,
        jobject serviceObj, const sp<Looper>& looper) :
        mLooper(looper), mInteractive(true) {
    ...
    sp<EventHub> eventHub = new EventHub();
    mInputManager = new InputManager(eventHub, this, this);
}
```

NativeInputManager 构造函数中创建了 EventHub 和 InputManager，EventHub 通过 Linux 内核的 INotify 与 Epoll 机制监听设备节点，通过 EventHub 的 getEvent 函数读取设备节点的增删事件和原始输入事件。InputManager 的构造函数如下所示。

frameworks/native/services/inputflinger/InputManager.cpp

```
InputManager::InputManager(
        const sp<EventHubInterface>& eventHub,
```

```
        const sp<InputReaderPolicyInterface>& readerPolicy,
        const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
    mDispatcher = new InputDispatcher(dispatcherPolicy);
    mReader = new InputReader(eventHub, readerPolicy, mDispatcher);
    initialize();
}

void InputManager::initialize() {
    mReaderThread = new InputReaderThread(mReader);
    mDispatcherThread = new InputDispatcherThread(mDispatcher);
}
```

在 InputManager 构造函数中创建了 InputReader 和 InputDispatcher，InputReader 会不断循环读取 EventHub 中的原始输入事件，将这些原始输入事件进行加工后交由 InputDispatcher 处理，InputDispatcher 中保存了 WMS 的所有窗口信息（WMS 会将窗口信息实时处理并更新到 InputDispatcher 中），这样 InputDispatcher 就可以将输入事件派发给合适的窗口。InputReader 和 InputDispatcher 都是耗时操作，因此在 initialize 函数中创建了供它们运行的 InputReaderThread 和 InputDispatcherThread 线程。

InputManagerService 构造方法描绘了如图 3-4 所示的 IMS 架构简图。

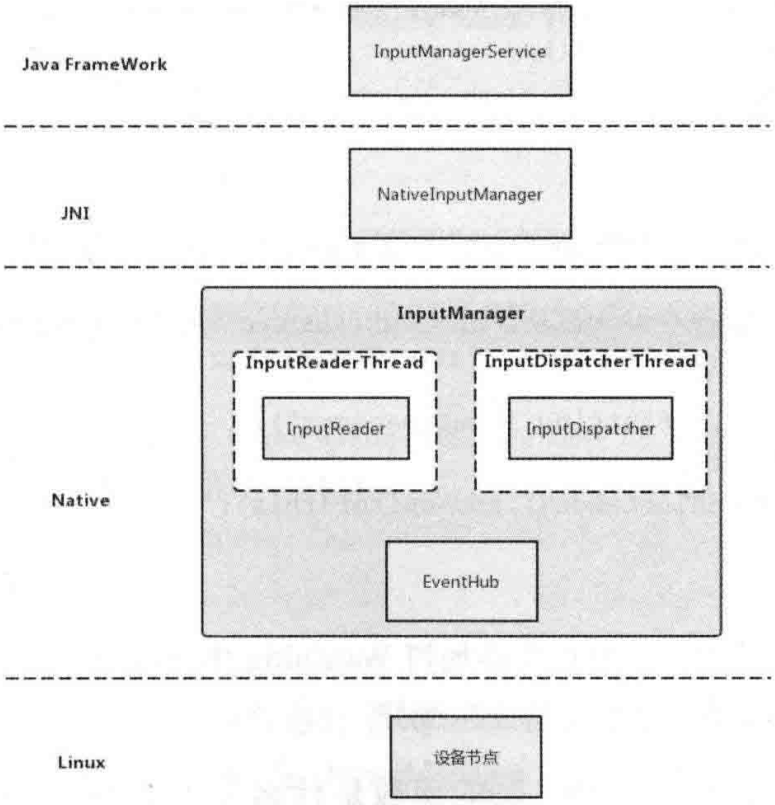


图 3-4　IMS 架构简图

从图 3-4 可以看出，IMS 的主要工作都在 Native 层中，这些内容会在后续章节进行介绍，我们在这里先有一个印象，后续随着相关知识的讲解，会慢慢丰富这张简图。

3.3 IMS 的启动过程

在 3.2 节中，我们学习了 IMS 的创建，IMS 创建后还会启动，在这一节继续来学习 IMS 的启动过程和输入事件的处理。

在 SystemServer 的 startOtherServices 方法中进行 IMS 的创建，代码如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void startOtherServices() {
...
        traceBeginAndSlog("StartInputManagerService");
        inputManager = new InputManagerService(context);
        traceEnd();
...
        traceBeginAndSlog("StartInputManager");
        inputManager.setWindowManagerCallbacks(wm.getInputMonitor());
        inputManager.start();
        traceEnd();

}
```

创建 IMS 后就会执行 IMS 的启动。IMS 的 start 方法如下所示。

frameworks/base/services/core/java/com/android/server/input/InputManagerService.java

```
public void start() {
        Slog.i(TAG, "Starting input manager");
        nativeStart(mPtr);
        Watchdog.getInstance().addMonitor(this);
      ...
    }
```

IMS 的 start 方法中，会将自身添加到 Watchdog 中进行监控，用于定时检测系统关键服务（AMS 和 WMS 等）是否可能发生死锁。

nativeStart 方法对应的 JNI 层的函数是什么呢？查看 com_android_server_input_InputManagerService 的 gInputManagerMethods 数组，代码如下所示。

frameworks/base/services/core/jni/com_android_server_*input*_InputManagerService.cpp

```
static const JNINativeMethod gInputManagerMethods[] = {
...
  { "nativeStart", "(J)V",
        (void*) nativeStart },
...
}
```

nativeStart 方法对应的 JNI 函数为 nativeStart 代码，如下所示。

frameworks/base/services/core/jni/com_android_server_input_InputManagerService.cpp

```
static void nativeStart(JNIEnv* env, jclass /* clazz */, jlong ptr) {
   NativeInputManager* im = reinterpret_cast<NativeInputManager*>(ptr);
   status_t result = im->getInputManager()->start();//1
   if (result) {
      jniThrowRuntimeException(env, "Input manager could not be started.");
   }
}
```

用 reinterpret_cast 操作符将 jlong 类型的 ptr 强制转换为原类型（NativeInputManager 指针类型）。注释 1 处调用了 InputManager 的 start 函数，代码如下所示。

frameworks/native/services/inputflinger/InputManager.cpp

```
status_t InputManager::start() {
   status_t result = mDispatcherThread->run("InputDispatcher",
PRIORITY_URGENT_DISPLAY);
   if (result) {
      ALOGE("Could not start InputDispatcher thread due to error %d.", result);
      return result;
   }
   result = mReaderThread->run("InputReader", PRIORITY_URGENT_DISPLAY);
   if (result) {
      ALOGE("Could not start InputReader thread due to error %d.", result);
      mDispatcherThread->requestExit();
      return result;
   }
   return OK;
}
```

可以看到 InputManager 的 start 函数运行了 InputReaderThread 和 InputDispatcherThread，这两个线程在 3.2.2 节讲过，它们在 InputManager 的构造函数中被创建，其中在 InputReaderThread 中运行了 InputReader，在 InputDispatcherThread 中运行了 InputDispatcher。

3.4 InputDispatcher 的启动过程

先来回顾一下 InputDispatcher 和 InputReader 是在哪里创建的，InputManager 的构造函数如下所示。

frameworks/native/services/inputflinger/InputManager.cpp

```
InputManager::InputManager(
      const sp<EventHubInterface>& eventHub,
      const sp<InputReaderPolicyInterface>& readerPolicy,
      const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
   mDispatcher = new InputDispatcher(dispatcherPolicy);
   mReader = new InputReader(eventHub, readerPolicy, mDispatcher);
   initialize();
}
```

由代码可以看到 InputDispatcher 和 InputReader 是有关联的，InputDispatcher 会作为一个参数传入 InputReader 中。InputDispatcher 是在 InputReader 之前创建的，这个顺序不能改变，因为要确保 InputReader 将加工后的输入事件交给 InputDispatcher 时，InputDispatcher 已经被创建。

InputDispatcher 的定义如下所示。

frameworks/native/services/inputflinger/InputDispatcher.h

```
class InputDispatcherThread : public Thread {
public:
   explicit InputDispatcherThread(const sp<InputDispatcherInterface>&
dispatcher);
   ~InputDispatcherThread();
private:
   virtual bool threadLoop();
   sp<InputDispatcherInterface> mDispatcher;
};
}
```

InputDispatcher.h 中定义了 threadLoop 纯虚函数，InputDispatcher 继承了 Thread。native 的 Thread 内部有一个循环，当线程运行时，会调用上面代码的 threadLoop 函数，如果它返回 true 并且没有调用 requestExit 函数，就会接着循环调用 threadLoop 函数。

查看 InputDispatcherThread 的 threadLoop 函数是如何实现的，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
bool InputDispatcherThread::threadLoop() {
    mDispatcher->dispatchOnce();
    return true;
}
```

threadLoop 函数中只调用了 InputDispatcher 的 dispatchOnce 函数，代码如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::dispatchOnce() {
    nsecs_t nextWakeupTime = LONG_LONG_MAX;
    {
        AutoMutex _l(mLock);
        mDispatcherIsAliveCondition.broadcast();
        if (!haveCommandsLocked()) {//1
            dispatchOnceInnerLocked(&nextWakeupTime);//2
        }
        if (runCommandsLockedInterruptible()) {
            nextWakeupTime = LONG_LONG_MIN;
        }
    }
    nsecs_t currentTime = now();//3
    int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);//4
    mLooper->pollOnce(timeoutMillis);
}
```

注释 1 处用于检查 InputDispatcher 的缓存队列中是否有等待处理的命令，如果没有就会执行注释 2 处的 dispatchOnceInnerLocked 函数，它用来将输入事件分发给合适的窗口。注释 3 处获取当前的时间，结合注释 4 处，得出 InputDispatcherThread 需要睡眠的时间为 timeoutMillis。最后调用 Looper 的 pollOnce 函数使 InputDispatcherThread 进入睡眠状态，并将它的最长睡眠的时间设置为 timeoutMillis。当有输入事件产生时，InputReader 就会将睡眠状态的 InputDispatcher 唤醒，InputDispatcher 会重新开始分发输入事件。

那么 InputReader 是如何唤醒 InputDispatcherThread 的呢？我们继续往下学习。

3.5　InputReader 处理事件的过程

InputReader 是在 InputReaderThread 中启动的，InputReaderThread 和 InputDispatcherThread 的定义是类似的，也是继承了 Thread 并定义了 threadLoop 纯虚函数。如果处理的事件为键

盘输入事件，则其调用时序图如图 3-5 所示。

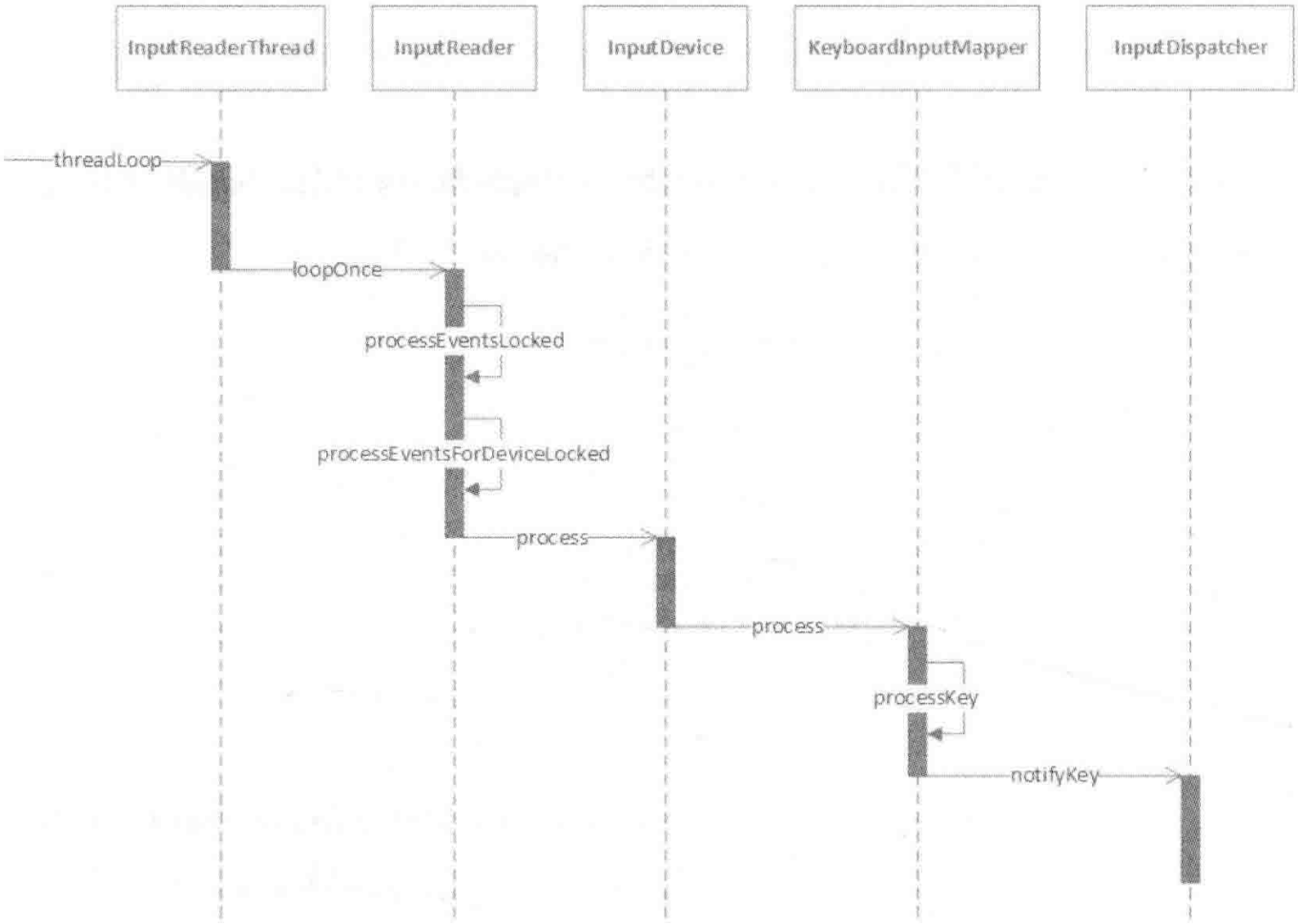


图 3-5　键盘输入事件的调用时序图

InputReaderThread 的 threadLoop 函数如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
bool InputReaderThread::threadLoop() {
    mReader->loopOnce();
    return true;
}
```

threadLoop 函数中只调用了 InputReader 的 loopOnce 函数，如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void InputReader::loopOnce() {
  ...
    //通过 EventHub 的 getEvents 函数获取事件信息并存放在 mEventBuffer 中
    size_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer,
EVENT_BUFFER_SIZE);//1
```

```
        AutoMutex _l(mLock);
        mReaderIsAliveCondition.broadcast();
        if (count) {
            //如果有事件信息，则调用 processEventsLocked 函数对事件进行加工处理
            processEventsLocked(mEventBuffer, count);//2
        }
    ...
}
```

注释 1 处调用 EventHub 的 getEvents 函数来获取设备节点的事件信息并存放在 mEventBuffer 中，事件信息主要有两种，一种是设备节点的增删事件（设备事件），另一种是原始输入事件。注释 2 处的 processEventsLocked 函数用于对 mEventBuffer 中的原始输入事件信息进行加工处理，加工后的输入事件会交由 InputDispatcher 来处理，processEventsLocked 函数如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void InputReader::processEventsLocked(const RawEvent* rawEvents, size_t count) {
    //遍历所有事件
    for (const RawEvent* rawEvent = rawEvents; count;) {
        int32_t type = rawEvent->type;
        size_t batchSize = 1;
        //事件类型分为原始输入事件和设备事件，这个条件语句对原始输入事件进行处理
        if (type < EventHubInterface::FIRST_SYNTHETIC_EVENT) {
            int32_t deviceId = rawEvent->deviceId;
            while (batchSize < count) {
                if (rawEvent[batchSize].type >= EventHubInterface::FIRST_SYNTHETIC_
EVENT
                        || rawEvent[batchSize].deviceId != deviceId) {
                    break;
                }
                batchSize += 1;
            }
#if DEBUG_RAW_EVENTS
            ALOGD("BatchSize: %d Count: %d", batchSize, count);
#endif
            //处理 deviceId 所对应的设备的原始输入事件
            processEventsForDeviceLocked(deviceId, rawEvent, batchSize);//1
        } else {
            //对设备事件进行处理
            switch (rawEvent->type) {
            case EventHubInterface::DEVICE_ADDED:
                addDeviceLocked(rawEvent->when, rawEvent->deviceId);
                break;
```

```
            case EventHubInterface::DEVICE_REMOVED:
                removeDeviceLocked(rawEvent->when, rawEvent->deviceId);
                break;
            case EventHubInterface::FINISHED_DEVICE_SCAN:
                handleConfigurationChangedLocked(rawEvent->when);
                break;
            default:
                ALOG_ASSERT(false);
                break;
            }
        }
        count -= batchSize;
        rawEvent += batchSize;
    }
}
```

InputReader 的 processEventsLocked 函数首先遍历了所有事件，这些事件用 RawEvent 对象来表示，将原始输入事件和设备事件分开处理，其中设备事件分为 DEVICE_ADDED（设备添加事件）、DEVICE_REMOVED 和 FINISHED_DEVICE_SCAN，这些事件是在 EventHub 的 getEvent 函数中生成的。如果是 DEVICE_ADDED 事件，则 InputReader 会新建 InputDevice 对象，用来存储设备信息，并且将 InputDevice 存储在 KeyedVector 类型的容器 mDevices 中。

同一个设备的输入事件交给 processEventsForDeviceLocked 函数来处理，如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void InputReader::processEventsForDeviceLocked(int32_t deviceId,
        const RawEvent* rawEvents, size_t count) {
    ssize_t deviceIndex = mDevices.indexOfKey(deviceId);//1
    if (deviceIndex < 0) {
        ALOGW("Discarding event for unknown deviceId %d.", deviceId);
        return;
    }
    InputDevice* device = mDevices.valueAt(deviceIndex);//2
    if (device->isIgnored()) {
        return;
    }
    device->process(rawEvents, count);
}
```

注释 1 处根据 deviceId 从 mDevices 中获取对应的 deviceIndex，注释 2 处再根据 deviceIndex 从 mDevices 中获取对应的 InputDevice。最后调用 InputDevice 的 process 函数，

如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void InputDevice::process(const RawEvent* rawEvents, size_t count) {*
    size_t numMappers = mMappers.size();
    //遍历处理该 InputDevice 的所有事件
    for (const RawEvent* rawEvent = rawEvents; count--; rawEvent++) {
#if DEBUG_RAW_EVENTS
        ALOGD("Input event: device=%d type=0x%04x code=0x%04x value=0x%08x
when=%lld",
                rawEvent->deviceId, rawEvent->type, rawEvent->code, rawEvent->value,
                rawEvent->when);
#endif
      //mDropUntilNextSync 的值默认为 false，如果设备的输入事件缓冲区溢出，则这个值会设置为 true
        if (mDropUntilNextSync) {
            ...
        } else {
            for (size_t i = 0; i < numMappers; i++) {//1
                InputMapper* mapper = mMappers[i];
                mapper->process(rawEvent);//2
            }
        }
    }
}
```

在上述代码中，首先遍历 InputDevice 中的所有事件，真正加工原始输入事件的是 InputMapper 对象，由于原始输入事件的类型很多，因此在 InputMapper 中有很多子类，用于加工不同的原始输入事件，比如 KeyboardInputMapper 用于处理键盘输入事件，TouchInputMapper 用于处理触摸输入事件。

注释 1 处遍历所有 InputMapper，在注释 2 处，将原始输入事件交由这些 InputMapper 来处理，至于是哪个 InputMapper 来处理的，InputReader 并不关心。

这里以处理键盘输入事件为例，KeyboardInputMapper 的 process 函数如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void KeyboardInputMapper::process(const RawEvent* rawEvent) {
    switch (rawEvent->type) {
    case EV_KEY: {//1
        int32_t scanCode = rawEvent->code;
        int32_t usageCode = mCurrentHidUsage;
        mCurrentHidUsage = 0;
        if (isKeyboardOrGamepadKey(scanCode)) {
```

```
            processKey(rawEvent->when, rawEvent->value != 0, scanCode,
usageCode);//2
        }
        break;
    }
    ...
    }
}
```

在注释 1 处，如果事件的类型为按键类型，就会调用注释 2 处的 KeyboardInputMapper 的 processKey 函数，如下所示。

frameworks/native/services/inputflinger/InputReader.cpp

```
void KeyboardInputMapper::processKey(nsecs_t when, bool down, int32_t scanCode,
        int32_t usageCode) {
    ...
    NotifyKeyArgs args(when, getDeviceId(), mSource, policyFlags,
            down ? AKEY_EVENT_ACTION_DOWN : AKEY_EVENT_ACTION_UP,
            AKEY_EVENT_FLAG_FROM_SYSTEM, keyCode, scanCode, keyMetaState, downTime);
    getListener()->notifyKey(&args);//1
}
```

processKey 函数将加工后的键盘输入事件封装为 NotifyKeyArgs，将 NotifyKeyArgs 通知给 InputListenerInterface。

InputDispatcher 继承了 InputDispatcherInterface，而 InputDispatcherInterface 继承了 InputListenerInterface，因此注释 1 处实际上调用了 InputDispatcher 的 notifyKey 函数，将 NotifyKeyArgs 交由 InputDispatcher 处理，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::notifyKey(const NotifyKeyArgs* args) {
    ...
    bool needWake;
    {
        mLock.lock();
        if (shouldSendKeyToInputFilterLocked(args)) {
            mLock.unlock();
            policyFlags |= POLICY_FLAG_FILTERED;
            if (!mPolicy->filterInputEvent(&event, policyFlags)) {
                return;
            }
            mLock.lock();
        }
```

```
        int32_t repeatCount = 0;
        KeyEntry* newEntry = new KeyEntry(args->eventTime,
              args->deviceId, args->source, policyFlags,
              args->action, flags, keyCode, args->scanCode,
              metaState, repeatCount, args->downTime);//1
        needWake = enqueueInboundEventLocked(newEntry);//2
        mLock.unlock();
    }
    if (needWake) {
        mLooper->wake();
    }
}
```

代码块中采用 Mutex 互斥锁的形式，在注释 1 处，根据 NotifyKeyArgs 重新封装一个 KeyEntry 对象，代表一次按键数据。在注释 2 处，根据 KeyEntry 来判断是否需要将睡眠中的 InputDispatcherThread 唤醒，如果需要唤醒，就调用 Looper 的 wake 函数来唤醒，InputDispatcherThread 被唤醒后就会重新对输入事件进行分发。

3.6　输入事件的处理总结

输入事件的处理涉及了四个关键的类，分别是 IMS、EventHub、InputDispatcher 和 InputReader，它们做了如下工作：

（1）IMS 启动了 InputDispatcherThread 和 InputReaderThread，它们分别用来运行 InputDispatcher 和 InputReader。

（2）InputDispatcher 先于 InputReader 被创建，InputDispatcher 的 dispatchOnceInnerLocked 函数用来将事件分发给合适的窗口。InputDispatcher 没有输入事件处理时会进入睡眠状态，等待 InputReader 来通知唤醒。

（3）InputReader 通过 EventHub 的 getEvents 函数获取事件信息，如果是原始输入事件，就将这些原始输入事件交由不同的 InputMapper 来处理，最终交由 InputDispatcher 来进行分发。

（4）在 InputDispatcher 的 notifyKey 函数中会根据按键数据来判断 InputDispatcher 是否要被唤醒，InputDispatcher 被唤醒后，会重新调用 dispatchOnceInnerLocked 函数将输入事件分发给合适的窗口。

IMS 的架构图如图 3-6 所示。

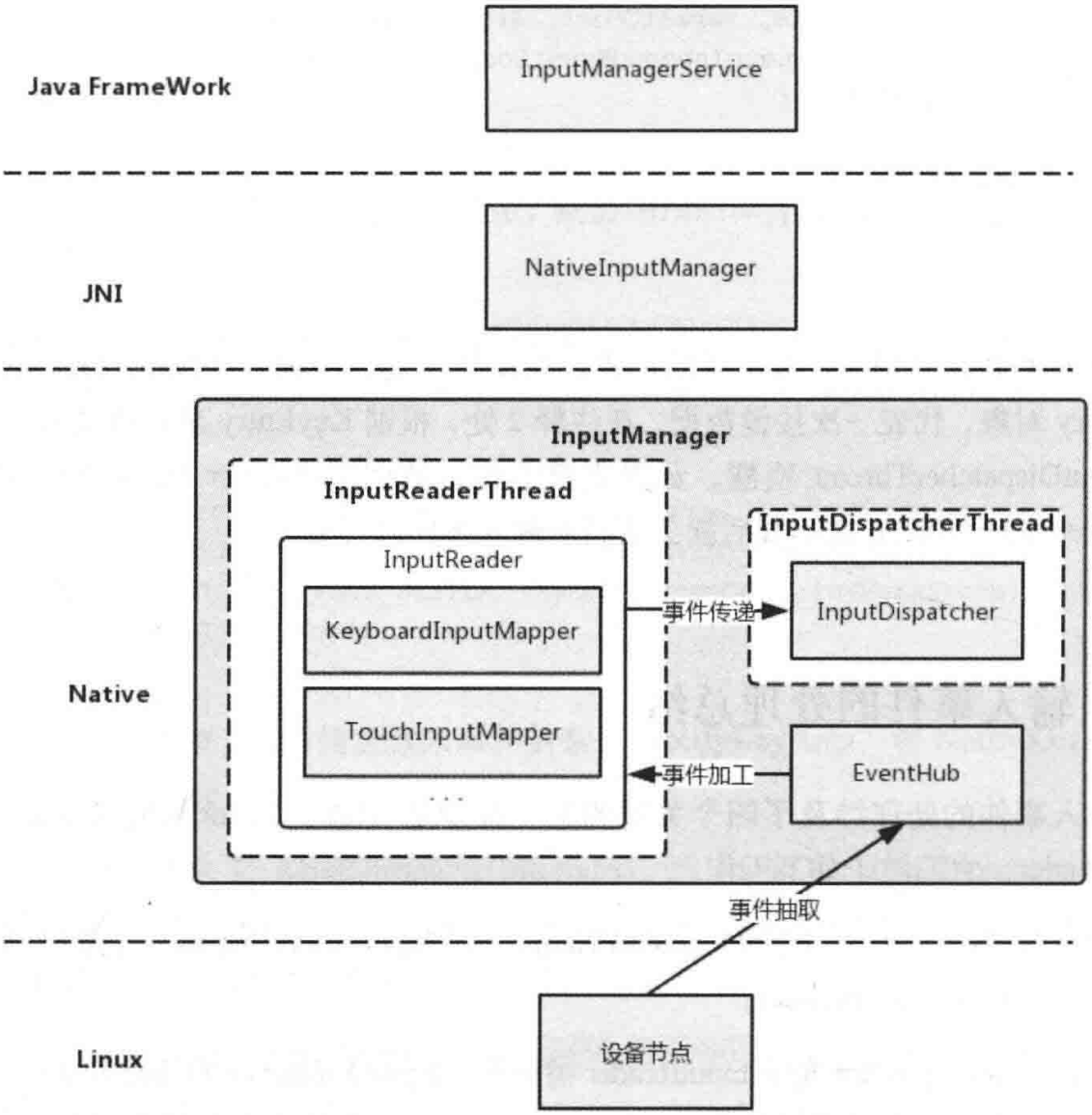


图 3-6　IMS 架构图

3.7　InputReader 的加工类型

在前面章节中，我们得知 InputReader 会对原始输入事件进行加工，如果事件的类型为按键类型，就会调用如下代码。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::notifyKey(const NotifyKeyArgs* args) {
  ...
    bool needWake;
    {
    ...
    }
    if (needWake) {
       mLooper->wake();
    }
}
```

InputDispatcher 的 notifyKey 方法用于唤醒 InputDispatcherThread，它的参数 NotifyKeyArgs 是 InputReader 对按键类型的事件加工后得到的，如下所示。

frameworks/native/services/inputflinger/InputListener.h

```
struct NotifyKeyArgs : public NotifyArgs {
    nsecs_t eventTime;
    int32_t deviceId;
    uint32_t source;
    uint32_t policyFlags;
    int32_t action;
    int32_t flags;
    int32_t keyCode;
    int32_t scanCode;
    int32_t metaState;
    nsecs_t downTime;
    inline NotifyKeyArgs() { }
    NotifyKeyArgs(nsecs_t eventTime, int32_t deviceId, uint32_t source, uint32_t
policyFlags, int32_t action, int32_t flags, int32_t keyCode, int32_t scanCode,
            int32_t metaState, nsecs_t downTime);
    NotifyKeyArgs(const NotifyKeyArgs& other);
    virtual ~NotifyKeyArgs() { }
    virtual void notify(const sp<InputListenerInterface>& listener) const;
};
```

由上述代码可以看到，NotifyKeyArgs 结构体继承自 NotifyArgs 结构体，NotifyArgs 有三个子类，分别是 NotifyKeyArgs、NotifyMotionArgs 和 NotifySwitchArgs，如图 3-7 所示。

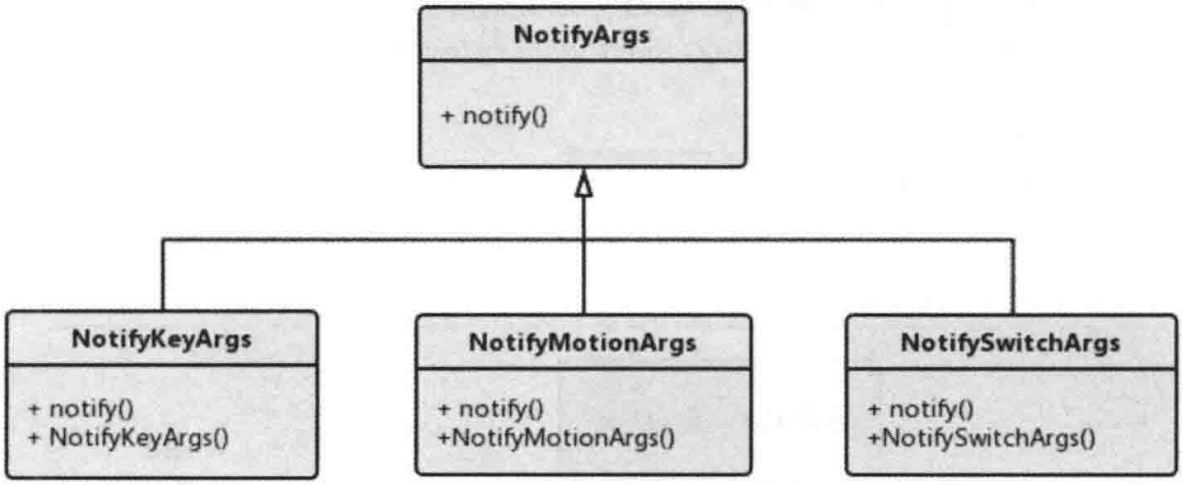


图 3-7　NotifyArgs 的三个子类

InputReader 对原始输入事件进行加工后，最终得出三种事件类型，分别是 key 事件、Motion 事件和 Switch 事件，将这些事件交由 InputDispatcher 进行分发，如图 3-8 所示。

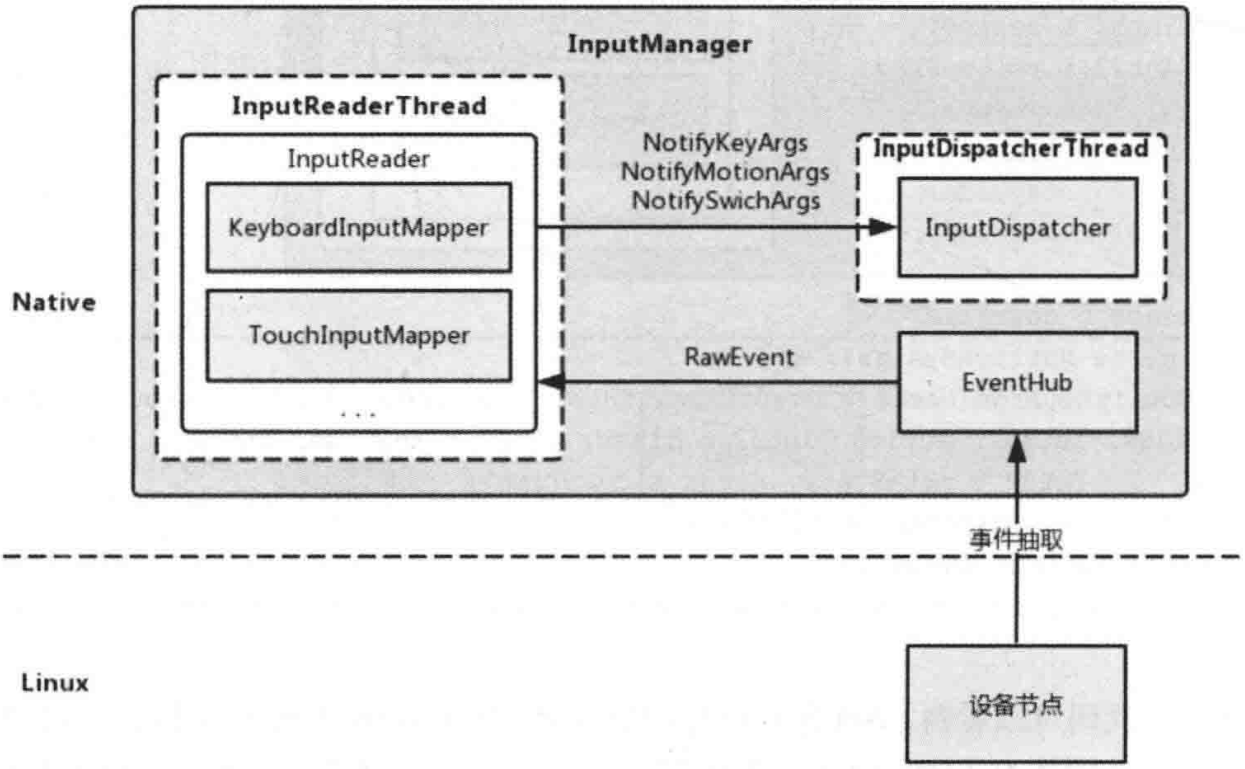


图 3-8　三种事件类型

3.8 InputDispatcher 的分发过程

不同的事件类型有不同的分发过程，其中 Switch 事件的处理是没有分发过程的，在 InputDispatcher 的 notifySwitch 函数中会将 Switch 事件交由 InputDispatcherPolicy 来处理。本章一直讲解 key 事件相关案例，这次以 Motion 事件的分发过程来举例，对 key 事件的分发过程感兴趣的读者可以自行去查看源码，二者本质上都差不多。

3.8.1 唤醒 InputDispatcherThread

InputDispatcher 的 notifyMotion 函数用来唤醒 InputDispatcherThread，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::notifyMotion(const NotifyMotionArgs* args) {
#if DEBUG_INBOUND_EVENT_DETAILS
...
#endif
    //检查 Motion 事件的参数是否有效
    if (!validateMotionEvent(args->action, args->actionButton,
            args->pointerCount, args->pointerProperties)) {//1
        return;
    }
    uint32_t policyFlags = args->policyFlags;
    policyFlags |= POLICY_FLAG_TRUSTED;
    mPolicy->interceptMotionBeforeQueueing(args->eventTime, /*byref*/
policyFlags);
    bool needWake;
    {
        mLock.lock();
        //Motion 事件是否需要交由 InputFilter 过滤
        if (shouldSendMotionToInputFilterLocked(args)) {//2
            mLock.unlock();
            MotionEvent event;
            //初始化 MotionEvent，将 NotifyMotionArgs 中的参数信息赋值给 MotionEvent 中的参数
            event.initialize(args->deviceId, args->source, args->action,
args->actionButton, args->flags, args->edgeFlags, args->metaState, args->buttonState,
                0, 0, args->xPrecision, args->yPrecision,
                args->downTime, args->eventTime,
                args->pointerCount, args->pointerProperties, args->pointerCoords);
            //表示已经过滤了
            policyFlags |= POLICY_FLAG_FILTERED;
```

```
            //开始过滤，如果返回值为 false，则直接 return，这次事件不再进行分发
            if (!mPolicy->filterInputEvent(&event, policyFlags)) {//3
                return;
            }
            mLock.lock();
        }
        /**
        * 4
        */
        MotionEntry* newEntry = new MotionEntry(args->eventTime,
                args->deviceId, args->source, policyFlags,
                args->action, args->actionButton, args->flags,
                args->metaState, args->buttonState,
                args->edgeFlags, args->xPrecision, args->yPrecision, args->downTime,
                args->displayId,
                args->pointerCount, args->pointerProperties, args->pointerCoords, 0, 0);
        needWake = enqueueInboundEventLocked(newEntry);//5
        mLock.unlock();
    }
    if (needWake) {
        mLooper->wake();//6
    }
}
```

注释 1 处用于检查 Motion 事件的参数是否有效，其内部会检查触控点的数量 pointerCount 是否在合理范围内（小于 1 或者大于 16 都是不合理的），以及触控点的 ID 是否在合理范围内（小于 0 或者大于 31 都是不合理的）。

在注释 2 处，如果 Motion 事件需要交由 InputFilter 过滤，就会初始化 MotionEvent，其作用就是用 NotifyMotionArgs 中的事件参数信息构造一个 MotionEvent，接着 MotionEvent 会交由注释 3 处的方法进行过滤，如果返回值为 false，则这次 Motion 事件就会被忽略掉。

在注释 4 处，用 NotifyMotionArgs 中的事件参数信息构造一个 MotionEntry 对象。注释 5 处将 MotionEntry 传入 enqueueInboundEventLocked 函数中，其内部会将 MotionEntry 添加到 InputDispatcher 的 mInboundQueue 队列的末尾，并返回一个值 needWake，代表 InputDispatcherThread 是否需要被唤醒，如果需要被唤醒就调用注释 6 处的代码来唤醒 InputDispatcherThread。

3.8.2 InputDispatcher 进行分发

InputDispatcherThread 被唤醒后，会执行 InputDispatcherThread 的 threadLoop 函数，如

下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
bool InputDispatcherThread::threadLoop() {
    mDispatcher->dispatchOnce();
    return true;
}
```

threadLoop 函数中只调用了 InputDispatcher 的 dispatchOnce 函数，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::dispatchOnce() {
    nsecs_t nextWakeupTime = LONG_LONG_MAX;
    {
        AutoMutex _l(mLock);
        mDispatcherIsAliveCondition.broadcast();
        if (!haveCommandsLocked()) {//1
            dispatchOnceInnerLocked(&nextWakeupTime);//2
        }
        if (runCommandsLockedInterruptible()) {
            nextWakeupTime = LONG_LONG_MIN;
        }
    }
    nsecs_t currentTime = now();//3
    int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);//4
    mLooper->pollOnce(timeoutMillis);
}
```

注释 1 处用于检查 InputDispatcher 的缓存队列中是否有等待处理的命令，如果没有就会执行注释 2 处的 dispatchOnceInnerLocked 函数，用来将输入事件分发给合适的窗口。注释 3 处获取当前的时间，结合注释 4 处，得出 InputDispatcherThread 需要睡眠的时间为 timeoutMillis。最后调用 Looper 的 pollOnce 函数使 InputDispatcherThread 进入睡眠状态，并将它的最长睡眠时间设置为 timeoutMillis。当有输入事件产生时，InputReader 就会将睡眠状态的 InputDispatcherThread 唤醒，InputDispatcher 会重新开始分发输入事件。查看注释 2 处的 dispatchOnceInnerLocked 函数是如何进行事件分发的，代码如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {
    ...
    // 如果 InputDispatcher 被冻结，则不进行派发操作
    if (mDispatchFrozen) {
#if DEBUG_FOCUS
```

```
        ALOGD("Dispatch frozen.  Waiting some more.");
#endif
        return;
    }
    //如果 isAppSwitchDue 为 true，则说明没有及时响应 Home 键等操作
    bool isAppSwitchDue = mAppSwitchDueTime <= currentTime;//1
    if (mAppSwitchDueTime < *nextWakeupTime) {//2
        *nextWakeupTime = mAppSwitchDueTime;
    }
    //如果还没有待分发的事件，则在 mInboundQueue 中取出一个事件
    if (! mPendingEvent) {
        //如果 mInboundQueue 为空，并且没有待分发的事件，就 return
        if (mInboundQueue.isEmpty()) {
            ...
            if (!mPendingEvent) {
                return;
            }
        } else {
            //如果 mInboundQueue 不为空，则取队列头部的 EventEntry 赋值给 mPendingEvent
            mPendingEvent = mInboundQueue.dequeueAtHead();
            traceInboundQueueLengthLocked();
        }
        if (mPendingEvent->policyFlags & POLICY_FLAG_PASS_TO_USER) {
            pokeUserActivityLocked(mPendingEvent);
        }
        resetANRTimeoutsLocked();
    }
    ALOG_ASSERT(mPendingEvent != NULL);
    bool done = false;
    DropReason dropReason = DROP_REASON_NOT_DROPPED;//3
    ...
    switch (mPendingEvent->type) {//4
    ...
    case EventEntry::TYPE_MOTION: {
        MotionEntry* typedEntry = static_cast<MotionEntry*>(mPendingEvent);
        //如果没有及时响应窗口切换操作
        if (dropReason == DROP_REASON_NOT_DROPPED && isAppSwitchDue) {
            dropReason = DROP_REASON_APP_SWITCH;
        }
        //事件过期
        if (dropReason == DROP_REASON_NOT_DROPPED
                && isStaleEventLocked(currentTime, typedEntry)) {
            dropReason = DROP_REASON_STALE;
        }
```

```
        //阻碍其他窗口获取事件
        if (dropReason == DROP_REASON_NOT_DROPPED && mNextUnblockedEvent) {
            dropReason = DROP_REASON_BLOCKED;
        }
        done = dispatchMotionLocked(currentTime, typedEntry,
                &dropReason, nextWakeupTime);//5
        break;
    }
    default:
        ALOG_ASSERT(false);
        break;
    }

    if (done) {
        if (dropReason != DROP_REASON_NOT_DROPPED) {
            dropInboundEventLocked(mPendingEvent, dropReason);
        }
        mLastDropReason = dropReason;
        //释放本次事件处理的对象
        releasePendingEventLocked();//6
        //使 InputDispatcher 能够快速处理下一个分发事件
        *nextWakeupTime = LONG_LONG_MIN;//7
}
```

InputDispatcher 的 dispatchOnceInnerLocked 函数的代码比较长，这里截取了和 Motion 事件的分发相关的主要源码。dispatchOnceInnerLocked 函数主要做了以下几件事。

（1）InputDispatcher 的冻结处理。

如果当前 InputDispatcher 被冻结，则不进行分发操作，InputDispatcher 有三种状态，分别是正常状态、冻结状态和禁用状态，可以通过 InputDispatcher 的 setInputDispatchMode 函数来设置。

（2）窗口切换操作处理。

注释 1 处的 mAppSwitchDueTime 代表了 App 最近发生窗口切换操作时（比如按 Home 键、挂断电话），该操作事件最迟的分发时间。如果 mAppSwitchDueTime 小于或等于当前系统时间，说明没有及时响应窗口切换操作，则 isAppSwitchDue 的值为 true。

在注释 2 处，如果 mAppSwitchDueTime 小于 nextWakeupTime（下一次 InputDispatcherThread 醒来的时间），就将 mAppSwitchDueTime 赋值给 nextWakeupTime，这样当 InputDispatcher 处理完分发事件后，就会第一时间处理窗口切换操作。

（3）取出事件。

如果没有待分发的事件，就从 mInboundQueue 中取出一个事件；如果 mInboundQueue 为空，并且没有待分发的事件，就调用 return；如果 mInboundQueue 不为空，则取队列头部的 EventEntry 赋值给 mPendingEvent，mPendingEvent 的类型为 EventEntry 对象指针。

（4）事件丢弃。

注释 3 处的 dropReason 代表了事件丢弃的原因，它的默认值为 DROP_REASON_NOT_DROPPED，其代表事件不被丢弃。

注释 4 处根据 mPendingEvent 的 type 进行区分处理，这里主要截取了对 Motion 类型的处理。经过过滤会调用注释 5 处的 dispatchMotionLocked 函数为这个事件寻找合适的窗口。

（5）后续处理。

如果注释 5 处的事件分发成功，则会在注释 6 处调用 releasePendingEventLocked 函数，其内部会将 mPendingEvent 的值设置为 Null，并将 mPendingEvent 指向的对象内存释放掉。注释 7 处将 nextWakeupTime 的值设置为 LONG_LONG_MIN，这是为了使 InputDispatcher 能够快速处理下一个分发事件。

3.9 事件分发到目标窗口的过程

在 3.8 节中我们学习了 InputDispatcher 的分发过程，这一过程还有一部分没有讲解，即事件分发到目标窗口的过程。

3.9.1 为事件寻找合适的分发目标

我们先来回顾 3.8 节中讲解的 InputDispatcher 的 dispatchOnceInnerLocked 函数，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {
    ...
    DropReason dropReason = DROP_REASON_NOT_DROPPED;//1
   ...
```

```
    switch (mPendingEvent->type) {//2
    ...
    case EventEntry::TYPE_MOTION: {
        MotionEntry* typedEntry = static_cast<MotionEntry*>(mPendingEvent);
        //如果没有及时响应窗口切换操作
        if (dropReason == DROP_REASON_NOT_DROPPED && isAppSwitchDue) {
            dropReason = DROP_REASON_APP_SWITCH;
        }
        //事件过期
        if (dropReason == DROP_REASON_NOT_DROPPED
                && isStaleEventLocked(currentTime, typedEntry)) {
            dropReason = DROP_REASON_STALE;
        }
        //阻碍其他窗口获取事件
        if (dropReason == DROP_REASON_NOT_DROPPED && mNextUnblockedEvent) {
            dropReason = DROP_REASON_BLOCKED;
        }
        done = dispatchMotionLocked(currentTime, typedEntry,
                &dropReason, nextWakeupTime);//3
        break;
    }
    default:
        ALOG_ASSERT(false);
        break;
    }
    ...
}
```

dispatchOnceInnerLocked 函数主要做了 5 件事，这里只截取了其中的 1 件事：事件丢弃。

注释 1 处的 dropReason 代表了事件丢弃的原因，它的默认值为 DROP_REASON_NOT_DROPPED，代表事件不被丢弃。

注释 2 处根据 mPendingEvent 的 type 进行区分处理，这里主要截取了对 Motion 类型的处理。经过条件语句过滤，会调用注释 3 处的 dispatchMotionLocked 函数为 Motion 事件寻找合适的窗口，如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
bool InputDispatcher::dispatchMotionLocked(
        nsecs_t currentTime, MotionEntry* entry, DropReason* dropReason, nsecs_t*
nextWakeupTime) {
    if (! entry->dispatchInProgress) {
```

```
        //标记当前已经进入分发的过程
        entry->dispatchInProgress = true;
        logOutboundMotionDetailsLocked("dispatchMotion - ", entry);
    }
    //如果事件是需要丢弃的，则返回true，不会为该事件寻找合适的窗口
    if (*dropReason != DROP_REASON_NOT_DROPPED) {//1
        setInjectionResultLocked(entry, *dropReason == DROP_REASON_POLICY
                ? INPUT_EVENT_INJECTION_SUCCEEDED : INPUT_EVENT_INJECTION_FAILED);
        return true;
    }
    bool isPointerEvent = entry->source & AINPUT_SOURCE_CLASS_POINTER;
    //目标窗口信息列表会存储在inputTargets中
    Vector<InputTarget> inputTargets;//2
    bool conflictingPointerActions = false;
    int32_t injectionResult;

    if (isPointerEvent) {
        //处理触摸形式的事件，比如触摸屏幕
        injectionResult = findTouchedWindowTargetsLocked(currentTime,
                entry, inputTargets, nextWakeupTime, &conflictingPointerActions);//3
    } else {
        //处理非触摸形式的事件，比如轨迹球
        injectionResult = findFocusedWindowTargetsLocked(currentTime,
                entry, inputTargets, nextWakeupTime);//4
    }
    //输入事件被挂起，说明找到了窗口并且窗口无响应
    if (injectionResult == INPUT_EVENT_INJECTION_PENDING) {
        return false;
    }
    setInjectionResultLocked(entry, injectionResult);
    //输入事件没有分发成功，说明没有找到合适的窗口
    if (injectionResult != INPUT_EVENT_INJECTION_SUCCEEDED) {
        if (injectionResult != INPUT_EVENT_INJECTION_PERMISSION_DENIED) {
            CancelationOptions::Mode mode(isPointerEvent ?
                    CancelationOptions::CANCEL_POINTER_EVENTS :
                    CancelationOptions::CANCEL_NON_POINTER_EVENTS);
            CancelationOptions options(mode, "input event injection failed");
            synthesizeCancelationEventsForMonitorsLocked(options);
        }
        return true;
    }
    //将分发的目标添加到inputTargets列表中
    addMonitoringTargetsLocked(inputTargets);//5
    if (conflictingPointerActions) {
```

```
        CancelationOptions options(CancelationOptions::CANCEL_POINTER_EVENTS,
              "conflicting pointer actions");
        synthesizeCancelationEventsForAllConnectionsLocked(options);
    }
    //将事件分发给 inputTargets 列表中的目标
    dispatchEventLocked(currentTime, entry, inputTargets);//6
    return true;
}
```

注释 1 处说明事件是需要丢弃的，这时就会直接返回 true，不会为该事件寻找窗口，这次的分发任务就没有完成，会在下一次 InputDispatcherThread 的循环中再次尝试分发。注释 3 处和注释 4 处会对触摸形式的事件和非触摸形式的事件进行处理，将事件处理的结果交由 injectionResult 处理。后面会判断 injectionResult 的值，如果 injectionResult 的值为 INPUT_EVENT_INJECTION_PENDING，则说明找到了窗口并且窗口无响应，这时输入事件被挂起，就会返回 false；如果 injectionResult 的值不为 INPUT_EVENT_INJECTION_SUCCEEDED，则说明没有找到合适的窗口，输入事件没有分发成功，这时就会返回 true。

注释 5 处会将分发的目标添加到 inputTargets 列表中，最终在注释 6 处将事件分发给 inputTargets 列表中的目标。

从注释 2 处可以看出在 inputTargets 列表中存储的是 InputTarget 结构体，代码如下所示。

frameworks/native/services/inputflinger/InputDispatcher.h

```
struct InputTarget {
  enum {
    //此标记表示事件正在交付给前台应用程序
    FLAG_FOREGROUND = 1 << 0,
    //此标记指示 MotionEvent 位于目标区域内
    FLAG_WINDOW_IS_OBSCURED = 1 << 1,
    ...
};
    //InputDispatcher 与目标窗口的通信管道
    sp<InputChannel> inputChannel;//1
    //事件派发的标记
    int32_t flags;
    //屏幕坐标系相对于目标窗口坐标系的偏移量
    float xOffset, yOffset;//2
    //屏幕坐标系相对于目标窗口坐标系的缩放系数
    float scaleFactor;//3
    BitSet32 pointerIds;
}
```

InputTarget 结构体可以说是 InputDispatcher 与目标窗口的转换器，其分为两大部分，一部分是枚举中存储的 InputDispatcher 与目标窗口交互的标记，另一部分是 InputDispatcher 与目标窗口交互的参数，比如注释 1 处的 inputChannel，它实际上是一个 SocketPair，SocketPair 用于进程间双向通信，SocketPair 非常适合 InputDispatcher 与目标窗口之间的通信，InputDispatcher 不仅要将事件分发到目标窗口，同时 InputDispatcher 还需要得到目标窗口对事件的响应。注释 2 处的 xOffset 和 yOffset 是屏幕坐标系相对于目标窗口坐标系的偏移量，MotionEntry(MotionEvent)中存储的坐标是屏幕坐标系，因此就需要注释 2 处和注释 3 处的参数，将屏幕坐标系转换为目标窗口坐标系。

3.9.2 处理点击形式的事件

在 InputDispatcher 的 dispatchMotionLocked 函数的注释 3 处和注释 4 处，分别对 Motion 事件中的触摸形式事件和非触摸形式事件进行了处理，由于非触摸形式事件不是很常见的，这里只对触摸形式事件进行解析。InputDispatcher 的 findTouchedWindowTargetsLocked 函数有 400 多行代码，这里截取了需要了解的部分，并且分两个部分来讲解。

（1）findTouchedWindowTargetsLocked 函数 part1。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
int32_t InputDispatcher::findTouchedWindowTargetsLocked(nsecs_t currentTime,
        const MotionEntry* entry, Vector<InputTarget>& inputTargets, nsecs_t*
nextWakeupTime,
        bool* outConflictingPointerActions) {
    ...
    if (newGesture || (isSplit && maskedAction ==
AMOTION_EVENT_ACTION_POINTER_DOWN)) {
      //从 MotionEntry 中获取坐标点
        int32_t pointerIndex = getMotionEventActionPointerIndex(action);
        int32_t x = int32_t(entry->pointerCoords[pointerIndex].
              getAxisValue(AMOTION_EVENT_AXIS_X));
        int32_t y = int32_t(entry->pointerCoords[pointerIndex].
              getAxisValue(AMOTION_EVENT_AXIS_Y));
        sp<InputWindowHandle> newTouchedWindowHandle;
        bool isTouchModal = false;
        size_t numWindows = mWindowHandles.size();//1
        //遍历窗口，找到触摸过的窗口和窗口之外的外部目标
        for (size_t i = 0; i < numWindows; i++) {//2
           //获取 InputDispatcher 中代表窗口的 windowHandle
           sp<InputWindowHandle> windowHandle = mWindowHandles.itemAt(i);
```

```
            //得到窗口信息 windowInfo
            const InputWindowInfo* windowInfo = windowHandle->getInfo();
            if (windowInfo->displayId != displayId) {
            //如果 displayId 不匹配，则开始下一次循环
                continue;
            }
            //获取窗口的 flag
            int32_t flags = windowInfo->layoutParamsFlags;
            //如果窗口是可见的
            if (windowInfo->visible) {
                //如果窗口的 flag 不为 FLAG_NOT_TOUCHABLE（窗口是 touchable）
                if (! (flags & InputWindowInfo::FLAG_NOT_TOUCHABLE)) {
                    //如果窗口是 focusable 或者 flag 不为 FLAG_NOT_FOCUSABLE，则说明该窗口是
                    //可触摸模式
                    isTouchModal = (flags & (InputWindowInfo::FLAG_NOT_FOCUSABLE
                            | InputWindowInfo::FLAG_NOT_TOUCH_MODAL)) == 0;//3
                    //如果窗口是可触摸模式或者坐标点落在窗口之上
                    if (isTouchModal || windowInfo->touchableRegionContainsPoint(x, y))
    {
                        newTouchedWindowHandle = windowHandle;//4
                        break;
                    }
                }
                if (maskedAction == AMOTION_EVENT_ACTION_DOWN
                        && (flags & InputWindowInfo::FLAG_WATCH_OUTSIDE_TOUCH)) {
                    //将符合条件的窗口放入 TempTouchState 中，以便后续处理
                    mTempTouchState.addOrUpdateWindow(
                            windowHandle, InputTarget::FLAG_DISPATCH_AS_OUTSIDE,
BitSet32(0));//5
                }
            }
        }
```

上述代码先从 MotionEntry 中获取坐标点，以便后面筛选窗口使用。注释 1 处获取列表 mWindowHandles 的 InputWindowHandle 数量，InputWindowHandle 中保存了 InputWindowInfo，InputWindowInfo 中又包含了 WindowManager.LayoutParams 定义的窗口标志，除了窗口标志，InputWindowInfo 中还包含了 InputChannel 和窗口的各种属性，InputWindowInfo 描述了可以接收输入事件的窗口属性。这样看来，InputWindowHandle 和 WMS 中的 WindowState 很相似。通俗来讲，WindowState 用来代表 WMS 中的窗口，而 InputWindowHandle 用来代表输入系统中的窗口。

那么输入系统是如何得到窗口信息的呢？这是因为 mWindowHandles 列表就是 WMS

更新到 InputDispatcher 中的。

注释 2 处开始遍历 mWindowHandles 列表中的窗口，找到触摸过的窗口和窗口之外的外部目标。在注释 3 处，如果窗口是 focusable 或者 flag 不为 FLAG_NOT_FOCUSABLE，则说明该窗口是可触摸模式。经过层层筛选，如果窗口是可触摸模式或者坐标点落在窗口之上，则会在注释 4 处将 windowHandle 赋值给 newTouchedWindowHandle。最后在注释 5 处将 newTouchedWindowHandle 添加到 TempTouchState 中，以便后续处理。

（2）findTouchedWindowTargetsLocked 函数 part2。

我们接着查看 findTouchedWindowTargetsLocked 函数剩余部分的代码。

```
...
    //确保所有触摸过的前台窗口都为新的输入做好了准备
    for (size_t i = 0; i < mTempTouchState.windows.size(); i++) {
        const TouchedWindow& touchedWindow = mTempTouchState.windows[i];
        if (touchedWindow.targetFlags & InputTarget::FLAG_FOREGROUND) {
            //检查窗口是否准备好接收更多输入
            String8 reason = checkWindowReadyForMoreInputLocked(currentTime,
                    touchedWindow.windowHandle, entry, "touched");//1
            if (!reason.isEmpty()) {//2
                //如果窗口没有准备好，则将原因赋值给 injectionResult
                injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
                        NULL, touchedWindow.windowHandle, nextWakeupTime,
reason.string());//3
                //不进行后续处理，直接跳到 Unresponsive 标签
                goto Unresponsive;//4
            }
        }
    }
    ...
    //代码运行到这里，说明窗口已经查找成功
    injectionResult = INPUT_EVENT_INJECTION_SUCCEEDED;//5
    //遍历 TempTouchState 中的窗口
    for (size_t i = 0; i < mTempTouchState.windows.size(); i++) {
        const TouchedWindow& touchedWindow = mTempTouchState.windows.itemAt(i);


        //为每个 mTempTouchState 中的窗口生成 InputTargets
        addWindowTargetLocked(touchedWindow.windowHandle,
touchedWindow.targetFlags,
                touchedWindow.pointerIds, inputTargets);//6
    }
```

```
        //在下一次迭代中，删除外部窗口或悬停触摸窗口
        mTempTouchState.filterNonAsIsTouchWindows();
    ...
    Unresponsive:
        //重置 TempTouchState
        mTempTouchState.reset();
        nsecs_t timeSpentWaitingForApplication =
getTimeSpentWaitingForApplicationLocked(currentTime);
        updateDispatchStatisticsLocked(currentTime, entry,
              injectionResult, timeSpentWaitingForApplication);
    #if DEBUG_FOCUS
        ALOGD("findTouchedWindow finished: injectionResult=%d,
injectionPermission=%d, "
              "timeSpentWaitingForApplication=%0.1fms",
              injectionResult, injectionPermission, timeSpentWaitingForApplication /
1000000.0);
    #endif
        return injectionResult;
    }
```

注释 1 处用于检查窗口是否准备好接收更多输入，并将结果赋值给 reason。在注释 2 处，如果 reason 的值不为空，则说明该窗口无法接收更多输入，注释 3 处的 handleTargetsNotReadyLocked 函数会得到无法接收更多输入的原因，将原因赋值给 injectionResult，其函数内部会计算窗口处理的时间，如果超时（默认为 5 秒），就会报 ANR，并将 nextWakeupTime 的值设置为 LONG_LONG_MIN，强制 InputDispatcherThread 在下一次循环中立即被唤醒，InputDispatcher 会重新开始分发输入事件。这时，injectionResult 的值为 INPUT_EVENT_INJECTION_PENDING。由于窗口无法接收更多输入，因此会在注释 4 处调用 goto 语句跳到 Unresponsive 标签，Unresponsive 标签中会调用 TempTouchState 的 reset 函数来重置 TempTouchState。

如果代码已经运行到了注释 5 处，则说明窗口已经查找成功，会遍历 TempTouchState 中的窗口，在注释 6 处为每个 TempTouchState 中的窗口生成 inputTargets。

在 3.9.1 节中，InputDispatcher 的 dispatchMotionLocked 函数的注释 6 处，会调用 InputDispatcher 的 dispatchEventLocked 函数，将事件分发给 inputTargets 列表中的分发目标，接下来我们来查看这是如何实现的。

3.9.3　向目标窗口发送事件

InputDispatcher 的 dispatchEventLocked 函数如下所示。

frameworks/native/services/inputflinger/InputDispatcher.cpp

```
void InputDispatcher::dispatchEventLocked(nsecs_t currentTime,
      EventEntry* eventEntry, const Vector<InputTarget>& inputTargets) {
#if DEBUG_DISPATCH_CYCLE
   ALOGD("dispatchEventToCurrentInputTargets");
#endif
   ALOG_ASSERT(eventEntry->dispatchInProgress);
   pokeUserActivityLocked(eventEntry);
   //遍历 inputTargets 列表
   for (size_t i = 0; i < inputTargets.size(); i++) {
      const InputTarget& inputTarget = inputTargets.itemAt(i);
      //根据 inputTarget 内部的 inputChannel 来获取 Connection 的索引
      ssize_t connectionIndex =
getConnectionIndexLocked(inputTarget.inputChannel);//1
      if (connectionIndex >= 0) {
         //获取保存在 mConnectionsByFd 容器中的 Connection
         sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);
         //根据 inputTarget，开始事件发送循环
         prepareDispatchCycleLocked(currentTime, connection, eventEntry,
&inputTarget);//2
      } else {
#if DEBUG_FOCUS
         ALOGD("Dropping event delivery to target with channel '%s' because it "
               "is no longer registered with the input dispatcher.",
               inputTarget.inputChannel->getName().string());
#endif
      }
   }
}
```

遍历 inputTargets 列表，获取每一个 inputTarget，注释 1 处根据 inputTarget 内部的 inputChannel 来获取 Connection 的索引，再根据这个索引作为 key 值来获取保存在 mConnectionsByFd 容器中的 Connection。Connection 可以理解为 InputDispatcher 和目标窗口的连接，其内部包含了连接的状态、InputChannel、InputWindowHandle 和事件队列等。注释 2 处调用了 prepareDispatchCycleLocked 函数，根据当前的 inputTarget，开始事件发送循环。最终会通过 inputTarget 中的 inputChannel 来和窗口进行进程间通信，最终将 Motion 事件发送给目标窗口。

3.9.4 Motion 事件分发过程总结

结合本章的内容总结一下 Motion 事件分发过程，如图 3-9 所示。

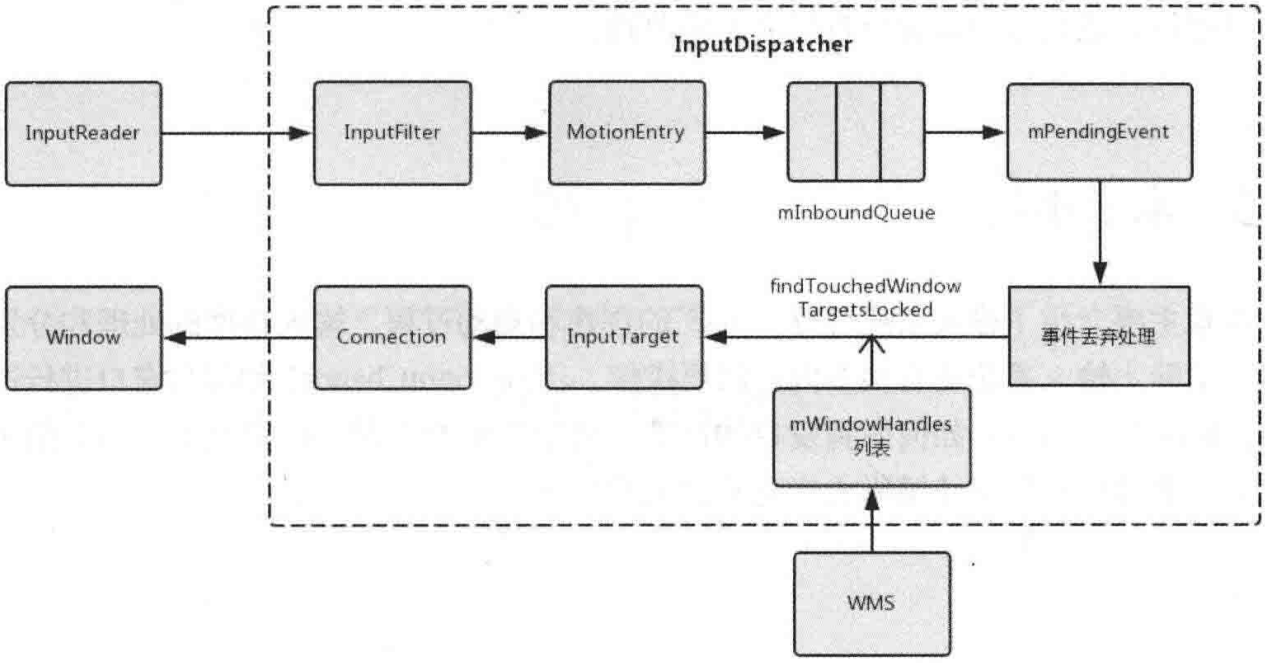


图 3-9　Motion 事件分发过程

（1）Motion 事件在 InputReaderThread 线程的 InputReader 中进行加工，加工完毕后会判断是否要唤醒 InputDispatcherThread，如果需要唤醒，则会在 InputDispatcherThread 的线程循环中不断地用 InputDispatcher 来分发 Motion 事件。

（2）将 Motion 事件交由 InputFilter 过滤，如果返回值为 false，则这次 Motion 事件就会被忽略掉。

（3）InputReader 对 Motion 事件加工后的数据结构为 NotifyMotionArgs，在 InputDispatcher 的 notifyMotion 函数中，用 NotifyMotionArgs 中的事件参数信息构造一个 MotionEntry 对象，这个 MotionEntry 对象会被添加到 InputDispatcher 的 mInboundQueue 队列的末尾。

（4）如果 mInboundQueue 不为空，则取出 mInboundQueue 队列头部的 EventEntry，并将其赋值给 mPendingEvent。

（5）根据 mPendingEvent 的值进行事件丢弃处理。

（6）调用 InputDispatcher 的 findTouchedWindowTargetsLocked 函数，在 mWindowHandles 列表中为 Motion 事件找到目标窗口，并为该窗口生成 InputTarget。

（7）根据 InputTarget 获取一个 Connection，依赖 Connection 将输入事件发送给目标窗口。

这里只是简单总结了 Motion 事件分发过程，与 Motion 事件类似的还有 key 事件，这里不再讲解，感兴趣的读者可自行去阅读源码。

3.10 本章小结

本章主要介绍了输入系统部分、IMS 的创建和启动过程、输入事件的处理和分发等知识点。实际上输入系统还有很多内容需要讲解，比如 inputChannel 如何和窗口进行进程间通信，InputDispatcher 如何得到窗口的反馈，由于篇幅的原因，这里就不过多介绍了，感兴趣的读者可以自行阅读源码或者查阅相关的博客。

第 4 章

IPC 机制

Binder 原理是掌握系统底层原理的基石，也是进阶高级工程师的必备知识点，这一章本应该介绍 Binder 原理，但是讲到 Binder 原理，不得不提到 IPC 机制，在这一章会先讲解 Linux 和 Android 的 IPC 机制种类，然后简单来介绍 Android 应用开发中接触到的 IPC 机制：Messenger、AIDL、ContentProvider、Socket。总体来说本章的内容比较简单，为第 5 章讲解 Binder 原理做铺垫。

4.1 Linux 和 Android 的 IPC 机制

IPC 全称为 Inter-Process Communication，含义为进程间通信，是指两个进程之间进行数据交换的过程。在 Android 和 Linux 中都有各自的 IPC 机制，这里分别来进行介绍。

4.1.1 Linux 中的 IPC 机制

Linux 中提供了很多进程间通信机制，主要有管道（Pipe）、信号（Sinal）、信号量（Semophore）、消息队列（Message）、共享内存（Share Memory）和套接字（Socket）等。

1. 管道

管道是 Linux 由 UNIX 继承过来的进程间的通信机制，它是 UNIX 早期的一个重要通信

机制。管道的主要思想是在内存中创建一个共享文件，从而使通信双方利用这个共享文件来传递信息。这个共享文件比较特殊，它不属于文件系统并且只存在于内存中。另外，管道采用半双工通信方式，数据只能在一个方向上流动。管道的简单模型如图 4-1 所示。

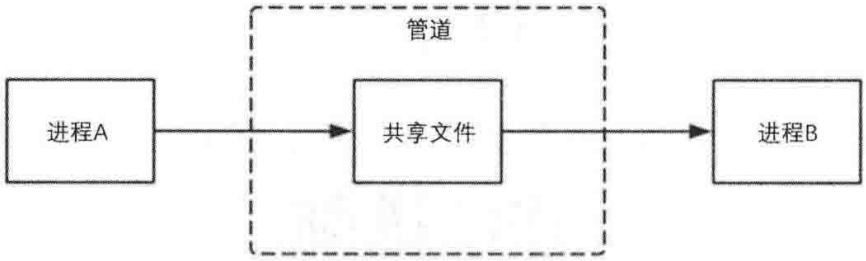


图 4-1　管道的简单模型

2．信号

信号是软件层次上对中断机制的一种模拟，信号是一种异步通信方式，进程不必通过任何操作来等待信号的到达。信号可以在用户空间进程和内核之间直接交互，内核可以利用信号来通知用户空间的进程发生了哪些系统事件。信号不适用于信息交换，比较适用于进程中断控制。

3．信号量

信号量是一个计数器，用来控制多个进程对共享资源的访问。信号量常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。信号量主要作为进程间及同一进程内不同线程之间的同步手段。

4．消息队列

消息队列是消息的链表，具有特定的格式，消息队列存放在内存中由消息队列标识符进行标识，并且允许一个或多个进程向它写入与读取消息。使用消息队列会使信息复制两次，因此对于频繁通信或者信息量大的通信不宜使用消息队列。

5．共享内存

共享内存的多个进程可以直接读写一块内存空间，是针对其他通信机制运行效率较低而设计的。为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间。这样，进程就可以直接读写这一块内存而不需要进行数据的复制，从而大幅度提高效率。

6．套接字

套接字是更基础的进程间通信机制，与其他通信机制不同的是，套接字可用于不同机器之间的进程间通信。

4.1.2　Android 中的 IPC 机制

Android 系统是基于 Linux 内核的，在 Linux 内核基础上又拓展出了一些 IPC 机制。Android 系统除了支持套接字，还支持序列化、Messenger、AIDL、Bundle、文件共享、ContentProvider 和 Binder 等。Binder 会在第 5 章进行介绍，下面先来了解 IPC 机制。

1．序列化

序列化指的是 Serializable/Parcelable 接口，Serializable 接口是 Java 提供的一个序列化接口，是一个空接口，为对象提供标准的序列化和反序列化操作。Parcelable 接口是 Android 中的序列化方式，更适合在 Android 平台上使用。虽然 Parcelable 接口用起来比较麻烦，但是其效率很高。

2．Messenger

Messenger 在 Android 应用开发中的使用频率不高，可以在不同进程中传递 Message 对象，在 Message 中加入我们想要传递的数据就可以在进程间进行数据传递了。Messenger 是一种轻量级的 IPC 方案，并对 AIDL 进行了封装。

3．AIDL

AIDL 全称为 Android Interface Definition Language，即 Android 接口定义语言。Messenger 是以串行的方式来处理客户端发来的信息的，如果有大量的消息发送到服务端，那么服务端仍然逐个处理再响应客户端显然是不合适的。虽然 Messenger 可以用于进程间数据传递，但是却不能满足跨进程的方法调用，这个时候就需要使用 AIDL 了。

4．Bundle

Bundle 实现了 Parcelable 接口，所以它可以在不同的进程间传输。Activity、Service、Receiver 都是在 Intent 中通过 Bundle 来进行数据传递的。

5. 文件共享

两个进程通过读写同一个文件来进行数据共享，共享的文件可以是文本、XML、JSON。文件共享适用于对数据同步要求不高的进程间通信。

6. ContentProvider

ContentProvider 为存储和获取数据提供统一的接口，它可以在不同的应用程序之间共享数据，ContentProvider 本身就是适合进程间通信的。ContentProvider 底层实现也是 Binder，但是使用起来比 AIDL 要容易许多。系统中很多操作都采用了 ContentProvider，例如通讯录、音视频等，这些操作本身就是跨进程进行通信的。

4.2 开启多进程

既然要讲解 Android 应用开发中的 IPC 机制，那么就需要代码的示例，而这些示例都需要开启多进程，本节我们来学习开启多进程。

Android 应用要开启多进程的原因，主要有以下几点：

- 单进程所分配的内存不够，需要更多的内存。早期的 Android 系统只为一个单进程的应用分配了 16MB 的可用内存，随着手机硬件的提升和 Android 系统的改进，虽然可分配内存越来越多，但仍旧可以通过开启多进程来获取更多内存来处理自己 App 的业务。
- 独立运行的组件，比如个推，它的服务会另开启一个进程。
- 运行一些“不可见人”的操作，比如获取用户的隐私数据，比如防止双守护进程被用户杀掉。

4.2.1 开启多进程举例

首先我们写一个 Activity 并启动一个服务，代码如下所示。

```
public class MyProcessActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my_process);
```

```
        Intent myServiceIntent=new Intent(MyProcessActivity.this,MyService.class);
        this.startService(myServiceIntent);
    }
}
```

服务 MyService 的代码如下所示。

```
public class MyService extends Service {

    private static final String TAG = "wangshu";
    @Override
    public void onCreate() {
        Log.i(TAG,"MyService is oncreate");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i(TAG, "MyProcessActivity is created: ");
        return START_STICKY;
    }
    @Override
    public void onDestroy() {
        Log.i(TAG,"OnDestory");
    }
    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

}
```

然后我们只需要在 AndroidManifest.xml 中配置 android:process 就可以了。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android……"
    package="com.example.liuwangshu.myprogress" >

    <application
        android:allowBackup="true"
        android:name=".MyApplication"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name=".MyProcessActivity" >
            <intent-filter>
```

```
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service
        android:name=".MyService"
        android:label="@string/app_name"
        android:process=":remote">
    </service>
  </application>
</manifest>
```

这里选择 remote 是主观的，也可以命名为其他名字。冒号“:”代替当前应用的包名，所以 MyService 运行在名为 com.example.liuwangshu.myprogress：remote 的进程中。我们也可以设置 android:process="com.example.liuwangshu.myprogress.remote"，这样 MyService 运行在名为 com.example.liuwangshu.myprogress.remote 的进程中。这两种命名方式也是有区别的，如果被设置的进程名是以一个冒号开头的，则这个新的进程对这个应用来说是私有的，其他应用的组件不能和新的进程运行在同一进程中，当新的进程被需要或者这个服务需要在新进程中运行的时候，这个新进程将会被创建。如果这个进程的名字是以小写字母开头的，则这个服务将运行在一个以这个名字命名的全局的进程中，当然前提是它有相应的权限，将允许不同应用中的组件运行在同一个进程中，从而减少资源的占用。

我们运行应用会发现：开启了两个进程，如图 4-2 所示。

htc-htc_m8et-HC542Y900916	Online	4.4.4
com.example.liuwangshu.myprogress	27933	8606
com.example.liuwangshu.myprogress:remote	27951	8610

图 4-2　开启两个进程

4.2.2　开启多进程引出的问题

开启多进程会使 Application 运行两次，我们继承 Application，在 oncreate 方法中添加 log 并运行程序，效果如图 4-3 所示。

```
public class MyApplication extends Application {
    private static final String TAG = "wangshu";
    @Override
    public void onCreate() {
        super.onCreate();
        int pid = android.os.Process.myPid();
        Log.i(TAG, "MyApplication is oncreate=====+"pid="+pid);
    }
```

```
12-29 19:30:53.917 29932-29932/? I/wangshu: MyApplication is oncreate=====pid=29932
12-29 19:30:54.007 29946-29946/? I/wangshu: MyApplication is oncreate=====pid=29946
12-29 19:30:54.007 29946-29946/? I/wangshu: MyService is oncreate
12-29 19:30:54.007 29946-29946/? I/wangshu: MyProcessActivity is created:
```

图 4-3　添加 log 并运行程序

在 log 中我们发现开启的两个进程都会执行 oncreate 方法。现在很多开发者都习惯在 Application 中进行初始化操作及数据的传递操作，这显然是不妥当的，解决的方法就是得到每个进程的名称，如果进程的名称和我们应用的进程名称相同则进行应用的操作，如果不相同则进行其他进程的操作。运行程序后的 log 信息如图 4-4 所示。

```
public class MyApplication extends Application {
    private static final String TAG = "wangshu";
    @Override
    public void onCreate() {
        super.onCreate();
        int pid = android.os.Process.myPid();
        Log.i(TAG, "MyApplication is oncreate=====+"pid="+pid);
        String processNameString = "";
        ActivityManager mActivityManager =
(ActivityManager)this.getSystemService(getApplicationContext().ACTIVITY_SERVICE);
        for (ActivityManager.RunningAppProcessInfo appProcess :
mActivityManager.getRunningAppProcesses()) {
            if (appProcess.pid == pid) {
                processNameString = appProcess.processName;
            }
        }
        if("com.example.liuwangshu.myprogress".equals(processNameString)){
            Log.i(TAG, "processName="+processNameString+"-----work");
        }else{
            Log.i(TAG, "processName="+processNameString+"-----work");
        }
    }
}
```

```
12-29 19:41:18.147 31023-31023/? I/wangshu: MyApplication is oncreate=====pid=31023
12-29 19:41:18.147 31023-31023/? I/wangshu: processName=com.example.liuwangshu.myprogress-----work
12-29 19:41:18.227 31039-31039/? I/wangshu: MyApplication is oncreate=====pid=31039
12-29 19:41:18.237 31039-31039/? I/wangshu: processName=com.example.liuwangshu.myprogress:remote-----work
12-29 19:41:18.237 31039-31039/? I/wangshu: MyService is oncreate
12-29 19:41:18.237 31039-31039/? I/wangshu: MyProcessActivity is created:
```

图 4-4　运行程序后的 log 信息

从图 4-4 中可以看到不同的进程执行了不同的操作。

4.3 用 Messenger 进行进程间通信

Messenger 可以在不同进程中传递 Message 对象，在 Message 中加入我们想要传递的数据就可以在进程间的进行数据传递了。Messenger 是一种轻量级的 IPC 方案并对 AIDL 进行了封装，它实现起来比较容易，下面我们来看看如何实现用 Messenger 进行进程间通信。

首先我们先写服务端（MessengerService.java），在 onBind 方法中创建 Messenger，关联接收消息的 Handler 调用 getBinder 来获取 Binder 对象，在 handleMessage 方法中接收客户端发来的信息。

MessengerService.java

```
import android.app.Service;
import android.content.Intent;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.util.Log;

public class MessengerService extends Service {
    public static final String TAG = "MoonMessenger";
    public static final int MSG_FROMCLIENT=1000;

    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_FROMCLIENT:
                    Log.i(TAG,"收到客户端信息-------"+msg.getData().get("msg"));
                    break;
            }
        }
    };

    @Override
    public IBinder onBind(Intent intent) {
        return new Messenger(mHandler).getBinder();
    }
}
```

需要注意的是注册服务时要另开启一个进程：

```
<service android:name=".MessengerService" android:process=":remote"></service>
```

接下来创建客户端（MessengerActivity.java），绑定另一个进程服务，绑定成功后根据服务端返回的 Binder 对象创建 Messenger，并用 Messenger 向服务端发送信息。

MessengerActivity.java

```
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MessengerActivity extends AppCompatActivity {
   private Messenger mMessenger;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_messenger);
      Intent intent=new Intent(MessengerActivity.this,MessengerService.class);
      bindService(intent,mServiceConnection, Context.BIND_AUTO_CREATE);
   }

private ServiceConnection mServiceConnection=new ServiceConnection() {
   @Override
   public void onServiceConnected(ComponentName name, IBinder service) {
      mMessenger=new Messenger(service);
      Message mMessage=Message.obtain(null,MessengerService.MSG_FROMCLIENT);
      Bundle mBundle=new Bundle();
      mBundle.putString("msg","这里是客户端，服务端收到了吗");
      mMessage.setData(mBundle);
      try {
         mMessenger.send(mMessage);
      } catch (RemoteException e) {
         e.printStackTrace();
      }
   }
```

```
        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };
        @Override
        protected void onDestroy() {
            super.onDestroy();
            unbindService(mServiceConnection);

        }
    }
```

运行程序后的 log 信息如图 4-5 所示。

```
01-02 16:34:50.191 16039-16039/? I/MoonMessenger: 收到客户端信息-------这里是客户端，服务端收到了吗
```

图 4-5　运行程序后的 log 信息

服务端收到了客户端的信息，但是服务端现在无法回应客户端，现在我们来修改代码使服务端回应客户端，客户端也能收到服务端的回应。

首先我们修改服务端，在 handleMessage 回调中收到客户端信息时，调用 Message.replyTo 得到客户端传递过来的 Messenger 对象，创建消息并通过 Messenger 发送给客户端。

MessengerService.java

```
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_FROMCLIENT:
                    Log.i(TAG,"收到客户端信息-------"+msg.getData().get("msg"));
                    //得到客户端传来的 Messenger 对象
                    Messenger mMessenger=msg.replyTo;
                    Message mMessage=Message.obtain(null,MessengerService.MSG_
FROMCLIENT);
                    Bundle mBundle=new Bundle();
                    mBundle.putString("rep","这里是服务端，我们收到信息了");
                    mMessage.setData(mBundle);
                    try {
                        mMessenger.send(mMessage);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }
```

```
                break;
            }
        }
    };
```

然后修改客户端，客户端需要创建一个 Handler 来接收服务端的信息，如下所示。

MessengerActivity.java

```
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what){
                case MessengerService.MSG_FROMCLIENT:
                    Log.i(MessengerService.TAG, "收到服务端信息-------" +
msg.getData().get("rep"));
                    break;
            }
        }
    };
```

前面我们在服务端调用 Message.replyTo 得到客户端传递过来的 Messenger 对象，可是客户端并没有传递 Messenger 对象，现在加上这段代码将 Messenger 对象传递给服务端，当然需要关联定义的 Handler。

MessengerActivity.java

```
    private ServiceConnection mServiceConnection=new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            mMessenger=new Messenger(service);
            Message mMessage=Message.obtain(null,MessengerService.MSG_FROMCLIENT);
            Bundle mBundle=new Bundle();
            mBundle.putString("msg", "这里是客户端，服务端收到了吗");
            mMessage.setData(mBundle);
            //将 Messenger 传递给服务端
            mMessage.replyTo=new Messenger(mHandler);
            try {
                mMessenger.send(mMessage);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
```

最后运行代码查看 log，运行程序后的 log 信息如图 4-6 所示。

```
01-02 17:07:08.611 19416-19416/? I/MoonMessenger: 收到客户端信息-------这里是客户端，服务端收到了吗
01-02 17:07:08.651 19403-19403/? I/MoonMessenger: 收到服务端信息-------这里是服务端，我们收到信息了
```

图 4-6　运行程序后的 log 信息

4.4　使用 AIDL 实现跨进程方法调用

在 4.3 节中，我们介绍了使用 Messenger 来进行进程间通信的方法，但是我们能够发现 Messenger 是以串行的方式来处理客户端发来的信息的，如果有大量的消息发到服务端，服务端仍然逐个处理再响应客户端显然是不合适的。虽然 Messenger 用来进程间进行数据传递，但是却不能满足跨进程的方法调用，接下来我们来使用 AIDL 来实现跨进程方法调用。

4.4.1　创建 AIDL 文件

我们将项目的目录结构调整为 Android 模式，在 java 同级目录下创建 aidl 文件夹，在文件夹中创建一个包，其包名和应用包名一致，如图 4-7 所示。

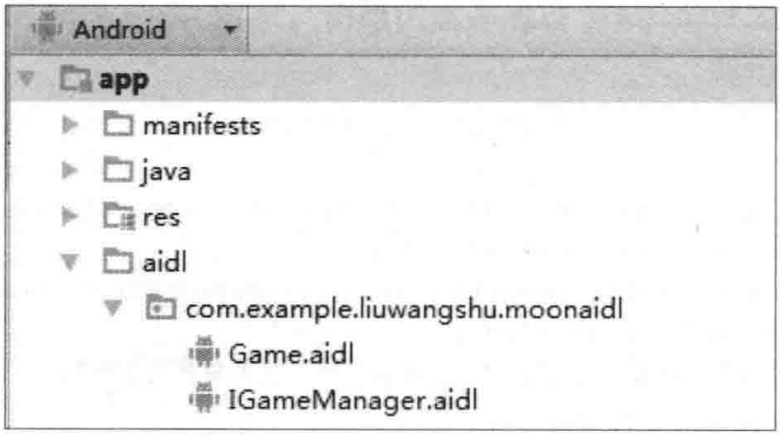


图 4-7　创建 AIDL 相关文件

我们先创建一个 IGameManager.aidl 的文件，这里面有两个方法，分别是 addGame 方法和 getGameList 方法，代码如下所示。

IGameManager.aidl

```
package com.example.liuwangshu.moonaidl;
import com.example.liuwangshu.moonaidl.Game;
interface IGameManager {
  List<Game>getGameList();
```

```
 void addGame(in Game game);
}
```

在 AIDL 文件中支持的数据类型如下。

- 基本数据类型。
- String 和 CharSequence。
- List：只支持 ArrayList，里面的元素都必须被 AIDL 支持。
- Map：只支持 HashMap，里面的元素都必须被 AIDL 支持。
- 实现 Parcelable 接口的对象。
- 所有 AIDL 接口。

在 IGameManager.aidl 中我们用到了 Game 类，这个类实现了 Parcelable 接口，在 AIDL 文件中我们需要 import 来查看 Game 类。

Game.java

```
package com.example.liuwangshu.moonaidl;
import android.os.Parcel;
import android.os.Parcelable;
public class Game implements Parcelable {
   public String gameName;
   public String gameDescribe;
   public Game(String gameName,String gameDescribe){
      this.gameName=gameName;
      this.gameDescribe=gameDescribe;
   }
   protected Game(Parcel in) {
      gameName=in.readString();
      gameDescribe=in.readString();
   }
   public static final Creator<Game> CREATOR = new Creator<Game>() {
      @Override
      public Game createFromParcel(Parcel in) {
         return new Game(in);
      }
      @Override
      public Game[] newArray(int size) {
         return new Game[size];
      }
   };
   @Override
   public int describeContents() {
      return 0;
```

```
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(gameName);
        dest.writeString(gameDescribe);
    }
}
```

在这里不讲解如何去实现 Parcelable 接口，在上面的 IGameManager.aidl 文件中我们用到了 Game 类，所以要创建 Game.aidl 来申明 Game 类实现了 Parcelable 接口。

Game.aidl

```
package com.example.liuwangshu.moonaidl;
parcelable Game;
```

现在重新编译程序，工程就会自动生成 IGameManager.aidl 对应的接口文件，这个文件生成的位置和 Eclipse 的位置不同，我们将项目的目录结构调整为 Project 模式，在 app→build→generated→soure→aidl→debug 目录下找到自己的文件，在文件中有一个接口文件 IGameManager，如图 4-8 所示。

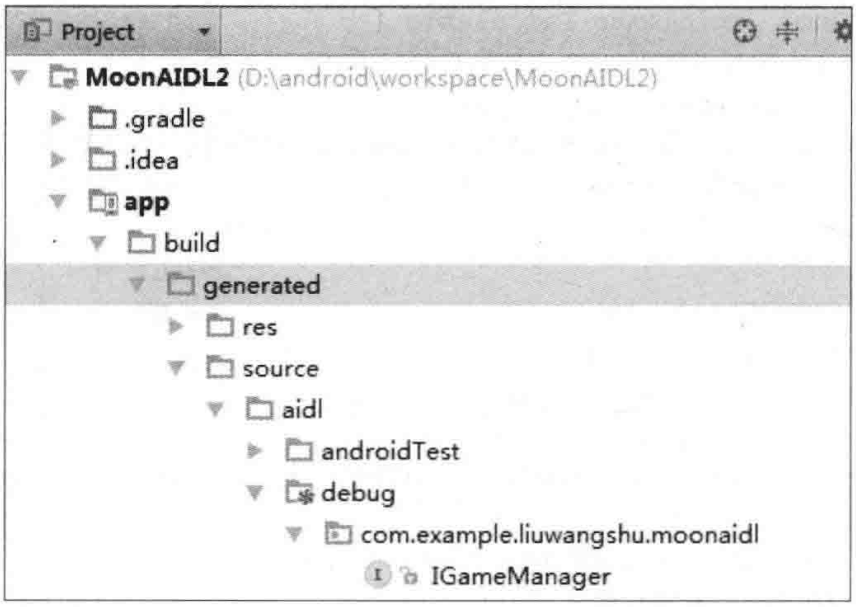


图 4-8　接口文件 IGameManager 的所在位置

4.4.2　创建服务端

在服务端 onCreate 方法中创建了两个游戏的信息，并创建 Binder 对象实现了 AIDL 的接口文件中的方法，在 onBind 方法中将 Binder 对象返回。

AIDLService.java

```
package com.example.liuwangshu.moonaidl;
import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import android.os.RemoteException;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
public class AIDLService extends Service{
    private CopyOnWriteArrayList<Game> mGameList=new
CopyOnWriteArrayList<Game>();
    private Binder mBinder= new IGameManager.Stub() {
        @Override
        public List<Game> getGameList() throws RemoteException {
            return mGameList;
        }
        @Override
        public void addGame(Game game) throws RemoteException {
            mGameList.add(game);
        }
    };
    @Override
    public void onCreate() {
       super.onCreate();
        mGameList.add(new Game("九阴真经 ol", "最好玩的武侠网游"));
        mGameList.add(new Game("大航海时代 ol","最好玩的航海网游"));

    }
    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
}
```

当然这个服务端应该运行在另一个进程，在 AndroidManifest.xml 文件中配置服务。

```
<service android:name=".AIDLService" android:process=":remote"></service>
```

4.4.3 客户端调用

最后在客户端 onCreate 方法中调用 bindService 方法绑定远程服务端，绑定成功后将返回的 Binder 对象转换为 AIDL 接口，这样就可以通过这个接口来调用远程服务端的方法了。

AIDLActivity.java

```
package com.example.liuwangshu.moonaidl;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.os.RemoteException;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import java.util.List;
public class AIDLActivity extends AppCompatActivity {
   private final static String TAG="AIDLActivity";
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_aidl);
      Intent mIntent=new Intent(AIDLActivity.this,AIDLService.class);
      bindService(mIntent,mServiceConnection, Context.BIND_AUTO_CREATE);
   }
  private ServiceConnection mServiceConnection=new ServiceConnection() {
     @Override
     public void onServiceConnected(ComponentName name, IBinder service) {
        IGameManager iGameManager=IGameManager.Stub.asInterface(service);
        Game game=new Game("月影传说","最好玩的武侠单机游戏");
        try {
           iGameManager.addGame(game);
           List<Game> mList=iGameManager.getGameList();
           for(int i=0;i<mList.size();i++){
              Game mGame=mList.get(i);
              Log.i(TAG,mGame.gameName+"---"+mGame.gameDescribe);
           }
        } catch (RemoteException e) {
           e.printStackTrace();
        }
     }
     @Override
     public void onServiceDisconnected(ComponentName name) {
     }
  };
   @Override
   protected void onDestroy() {
      super.onDestroy();
```

```
        unbindService(mServiceConnection);

    }
}
```

绑定成功后，我们创建了一个新的 Game 类调用远程服务端的 addGame 方法将新游戏添加进去，然后调用循环将远端服务中的所有游戏打印出来，运行程序打印结果如图 4-9 所示。

```
01-09 16:30:50.744 28741-28741/? I/AIDLActivity: 九阴真经ol---最好玩的武侠网游
01-09 16:30:50.744 28741-28741/? I/AIDLActivity: 大航海时代ol---最好玩的航海网游
01-09 16:30:50.754 28741-28741/? I/AIDLActivity: 月影传说---最好玩的武侠单机游戏
```

图 4-9　运行程序打印结果

打印出了远程服务端的所有游戏，这样就成功地在客户端通过 AIDL 来调用远程服务端的方法了。

4.5　用 ContentProvider 进行进程间通信

ContentProvider 为存储和获取数据提供统一的接口，它可以在不同的应用程序之间共享数据，本身就是适合进程间通信的。ContentProvider 底层实现也是 Binder，但是使用起来比 AIDL 要容易许多。

4.5.1　建立数据库，方便 ContentProvider 使用

创建数据库并创建表"game_provider.db"，里面有两个字段，分别用来存储游戏的名字和游戏的描述。

DbOpenHelper.java

```
package com.example.liuwangshu.mooncontentprovider;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
public class DbOpenHelper extends SQLiteOpenHelper {
   private static final String DB_NAME="game_provider.db";
    static final String GAME_TABLE_NAME="game";
   private static final int DB_VERSION=1;
```

```
        private String CREATE_GAME_TABLE="create table if not exists " + GAME_TABLE_NAME
+"(_id integer primary key," + "name TEXT, "+"describe TEXT)";
        public DbOpenHelper(Context context) {
            super(context, DB_NAME, null, DB_VERSION);
        }
        @Override
        public void onCreate(SQLiteDatabase db) {
            db.execSQL(CREATE_GAME_TABLE);
        }
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

        }
    }
```

4.5.2 使用 ContentProvider 对数据库进行操作

在 initProvoder 方法中，开启线程对数据库进行操作，删除表的所有数据，再添加我们想要添加的数据，接着实现 query 方法和 insert 方法。

GameProvider.java

```
package com.example.liuwangshu.mooncontentprovider;
import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.net.Uri;
public class GameProvider extends ContentProvider {
    public static final String AUTHORITY = "com.example.liuwangshu.mooncontentprovide.
GameProvider";
    public static final Uri GAME_CONTENT_URI = Uri.parse("content://" + AUTHORITY
+ "/game");
    private static final UriMatcher mUriMatcher = new UriMatcher(UriMatcher.NO_
MATCH);
    private SQLiteDatabase mDb;
    private Context mContext;
    private String table;
    static {
        mUriMatcher.addURI(AUTHORITY, "game", 0);
    }
    @Override
```

```
    public boolean onCreate() {
        table = DbOpenHelper.GAME_TABLE_NAME;
        mContext = getContext();
        initProvoder();
        return false;
    }
    private void initProvoder() {
        mDb = new DbOpenHelper(mContext).getWritableDatabase();
        new Thread(new Runnable() {
            @Override
            public void run() {
                mDb.execSQL("delete from " + DbOpenHelper.GAME_TABLE_NAME);
                mDb.execSQL("insert into game values(1,'九阴真经ol','最好玩的武侠网游
');");
            }
        }).start();
    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs, String sortOrder) {
        String table = DbOpenHelper.GAME_TABLE_NAME;
        Cursor mCursor = mDb.query(table, projection, selection, selectionArgs, null,
sortOrder, null);
        return mCursor;
    }
    @Override
    public String getType(Uri uri) {
        return null;
    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        mDb.insert(table, null, values);
        mContext.getContentResolver().notifyChange(uri, null);
        return null;
    }
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs) {
        return 0;
    }
}
```

在 manifest 文件中，要使 ContentProvider 运行在另一个进程，代码如下所示。

```
<provider android:authorities="com.example.liuwangshu.mooncontentprovide.
GameProvider"
            android:name=".GameProvider"
            android:process=":provider"
        ></provider>
```

4.5.3 在 Activity 中调用另一个进程的 GameProvider 方法

用 GameProvider 插入一条数据（此前 GameProvider 初始化时已经插入了一条数据），然后调用 GameProvider 的 query 方法来查询数据库中有几条数据并将其打印出来，代码如下所示。

```
package com.example.liuwangshu.mooncontentprovider;

import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
public class ContentProviderActivity extends AppCompatActivity {
    private final static String TAG = "ContentProviderActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_content_provider);
        Uri uri = Uri.parse("content://com.example.liuwangshu.mooncontentprovide.
GameProvider");
        ContentValues mContentValues = new ContentValues();
        mContentValues.put("_id", 2);
        mContentValues.put("name", "大航海时代 ol");
        mContentValues.put("describe", "最好玩的航海网游");
        getContentResolver().insert(uri, mContentValues);
        Cursor gameCursor = getContentResolver().query(uri, new String[]{"name",
"describe"}, null, null, null);
        while (gameCursor.moveToNext()) {
            Game mGame = new Game(gameCursor.getString(0), gameCursor.getString(1));
            Log.i(TAG, mGame.gameName + "---" + mGame.gameDescribe);
        }
    }
}
```

Bean 文件 Game.java 在 4.4.1 节使用过，因此直接拿过来用，代码如下所示。

```
package com.example.liuwangshu.mooncontentprovider;
import android.os.Parcel;
import android.os.Parcelable;
public class Game implements Parcelable {
   public String gameName;
   public String gameDescribe;

   public Game(String gameName, String gameDescribe) {
      this.gameName = gameName;
      this.gameDescribe = gameDescribe;
   }
   protected Game(Parcel in) {
      gameName = in.readString();
      gameDescribe = in.readString();
   }
   public static final Creator<Game> CREATOR = new Creator<Game>() {
      @Override
      public Game createFromParcel(Parcel in) {
         return new Game(in);
      }
      @Override
      public Game[] newArray(int size) {
         return new Game[size];
      }
   };
   @Override
   public int describeContents() {
      return 0;
   }
   @Override
   public void writeToParcel(Parcel dest, int flags) {
      dest.writeString(gameName);
      dest.writeString(gameDescribe);
   }
}
```

我们运行程序，发现 GameProvider 运行在另一个进程中，如图 4-10 所示。

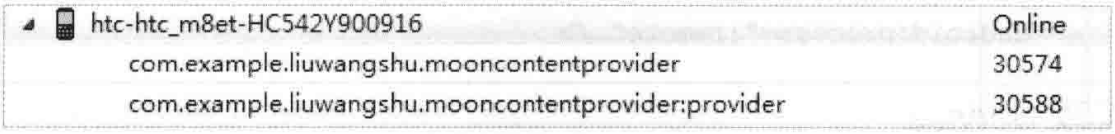


图 4-10　GameProvider 运行的进程

日志中也打印出了我们想要的结果，即两条游戏信息，如图 4-11 所示。

```
01-24 16:18:17.839 32460-32460/? I/ContentProviderActivity: 九阴真经ol---最好玩的武侠网游
01-24 16:18:17.839 32460-32460/? I/ContentProviderActivity: 大航海时代ol---最好玩的航海网游
```

图 4-11　运行程序后的日志信息

4.6　用 Socket 实现跨进程聊天程序

Socket 是位于应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口，供应用层调用以实现进程在网络中通信。Socket 分为流式套接字和数据包套接字，分别对应网络传输控制层的 TCP 协议和 UDP 协议。TCP 协议是一种面向连接的、可靠的、基于字节流的传输层通信协议，它使用三次握手协议建立连接，并且提供了超时重传机制，具有很高的稳定性。UDP 协议则是一种无连接的协议，且不对传送数据包进行可靠性保证，它适合一次传输少量数据，UDP 传输的可靠性由应用层负责。在网络质量令人十分不满意的环境下，UDP 协议数据包丢失会比较严重。但是由于 UDP 协议的特性：它不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用 UDP 协议较多。

接下来我们用 Socket 来实现跨进程聊天程序。

1．配置

首先我们来实现服务端，当然想要使用 Socket，需要在 AndroidManifest.xml 文件中声明如下权限。

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

需要实现一个远程的服务来当作聊天程序的服务端，在 AndroidManifest.xml 文件中配置 service。

```
<service
  android:name=".SocketServerService"
  android:process=":remote" />
```

2．实现 Service

接下来我们在 Service 启动时，在线程中建立 TCP 服务，监听 8688 端口，等待客户端

连接，当客户端连接时就会生成 Socket。通过每次创建的 Socket 就可以和不同的客户端通信了。当客户端断开连接时，服务端也会关闭 Socket 并结束通话线程。服务端首先会向客户端发送一条消息："您好，我是服务端"，并接收客户端发来的消息，将收到的消息进行加工再返回给客户端。

```
package com.example.liuwangshu.moonsocket;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.text.TextUtils;
import android.util.Log;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
public class SocketServerService extends Service {
    private boolean isServiceDestroyed = false;
    @Override
    public void onCreate() {
        new Thread(new TcpServer()).start();
        super.onCreate();
    }
    @Override
    public IBinder onBind(Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
    private class TcpServer implements Runnable {
        @Override
        public void run() {
            ServerSocket serverSocket;
            try {
                //监听 8688 端口
                serverSocket = new ServerSocket(8688);
            } catch (IOException e) {

                return;
            }
            while (!isServiceDestroyed) {
                try {
```

```
                    // 接受客户端请求，并且阻塞直到接收到消息
                    final Socket client = serverSocket.accept();
                    new Thread() {
                        @Override
                        public void run() {
                            try {
                                responseClient(client);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                    }.start();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    private void responseClient(Socket client) throws IOException {
        // 用于接收客户端消息
        BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        // 用于向客户端发送消息
        PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(client.getOutputStream())), true);
        out.println("您好，我是服务端");
        while (!isServiceDestroyed) {
            String str = in.readLine();
            Log.i("moon", "收到客户端发来的信息" + str);
            if (TextUtils.isEmpty(str)) {
                //客户端断开了连接
                Log.i("moon", "客户端断开连接");
                break;
            }
            String message = "收到了客户端的信息为: " + str;
            // 从客户端收到的消息加工再发送给客户端
            out.println(message);
        }
        out.close();
        in.close();
        client.close();
    }
    @Override
    public void onDestroy() {
```

```
        isServiceDestroyed = true;
        super.onDestroy();
    }
}
```

3. 实现聊天程序客户端

客户端 Activity 会在 onCreate 方法中启动服务端，并开启线程连接服务端 Socket。为了确保服务端 Socket 能连接成功，采用了超时重连的策略，每次连接失败时都会重新建立连接。连接成功后，客户端会收到服务端发送的消息："您好，我是服务端"，我们也可以在 EditText 输入字符并发送到服务端。

```
package com.example.liuwangshu.moonsocket;
import android.content.Intent;
import android.os.SystemClock;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
public class SocketClientActivity extends AppCompatActivity {
    private Button bt_send;
    private EditText et_receive;
    private Socket mClientSocket;
    private PrintWriter mPrintWriter;
    private TextView tv_message;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_socket);
        initView();
        Intent service = new Intent(this, SocketServerService.class);
        startService(service);
        new Thread() {
            @Override
```

```
            public void run() {
                connectSocketServer();
            }
        }.start();

    }
    private void initView() {
        et_receive= (EditText) findViewById(R.id.et_receive);
        bt_send= (Button) findViewById(R.id.bt_send);
        tv_message= (TextView) this.findViewById(R.id.tv_message);
        bt_send.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                final String msg = et_receive.getText().toString();
                //向服务器发送信息
                if(!TextUtils.isEmpty(msg)&&null!=mPrintWriter) {
                    mPrintWriter.println(msg);
                    tv_message.setText(tv_message.getText() + "\n" + "客户端: " + msg);
                    et_receive.setText("");
                }
            }
        });
    }
    private void connectSocketServer() {
        Socket socket = null;
        while (socket == null) {
            try {
                //选择和服务器相同的端口 8688
                socket = new Socket("localhost", 8688);
                mClientSocket = socket;
                mPrintWriter = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())), true);
            } catch (IOException e) {
                SystemClock.sleep(1000);
        }
        }
        try {
            //接收服务器端的消息
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.
getInputStream()));
            while (!isFinishing()) {
                final String msg = br.readLine();
                if (msg != null) {
                    runOnUiThread(new Runnable() {
```

```
                                    @Override
                                    public void run() {
                                        tv_message.setText(tv_message.getText() + "\n" +
"服务端: " + msg);
                                    }
                                }
                        );
                    }
                }
                mPrintWriter.close();
                br.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
```

布局如下所示。

activity_socket.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android……"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/tv_message"
        android:layout_width="match_parent"
        android:layout_height="400dp" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_alignParentBottom="true"
        android:orientation="horizontal">
        <EditText
            android:id="@+id/et_receive"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="2"
            />
        <Button
            android:id="@+id/bt_send"
```

```
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:text="向服务器发消息" />
    </LinearLayout>
</RelativeLayout>
```

4. 运行聊天程序

运行程序，我们可以看到客户端和服务端的进程信息，如图 4-12 所示。

meizu-m1-79MEAL823RJT	Online
com.example.liuwangshu.moonsocket	10361
com.example.liuwangshu.moonsocket:remote	10388

图 4-12　客户端和服务端的进程信息

客户端首先会收到服务端的信息："您好，我是服务端"，接下来我们向服务端发送"我想要怒放的生命"，这时候服务端收到了这条信息并将该信息加工后返回客户端。运行程序效果如图 4-13 所示。

图 4-13　运行程序效果

4.7　本章小结

本章比较简单，主要讲解了 Linux 和 Android 的 IPC 机制种类，以及 Android 应用开发涉及的 IPC 机制：Messenger、AIDL、ContentProvider、Socket。

第 5 章

Native Binder 原理

关联章节：

第 4 章 IPC 机制

第 6 章 Java Binder 原理

本章我们来学习 Native Binder 原理，根据 Android 系统的分层，Binder 机制可以分为 Java Binder、Native Binder、Kernel Binder。Native Binder 指的是 Native 层的 Binder，是在学习 Java Binder 原理前必须要学习的知识点。要讲解 Native Binder 原理还需要涉及 Kernel Binder 的内容，本章的 Kernel Binder 基于 goldfish3.4。

5.1 学习 Binder 的前置知识点

在讲解 Native Binder 原理前，需要先学习 Binder 的前置知识点，比如 Linux 和 Binder 的 IPC 通信原理、使用 Binder 的原因、学习 Binder 的原因。

5.1.1　Linux 和 Binder 的 IPC 通信原理

在讲到 Linux 的进程通信原理之前，首先需要了解 Linux 中的几个概念，进程通信简单模型如图 5-1 所示。

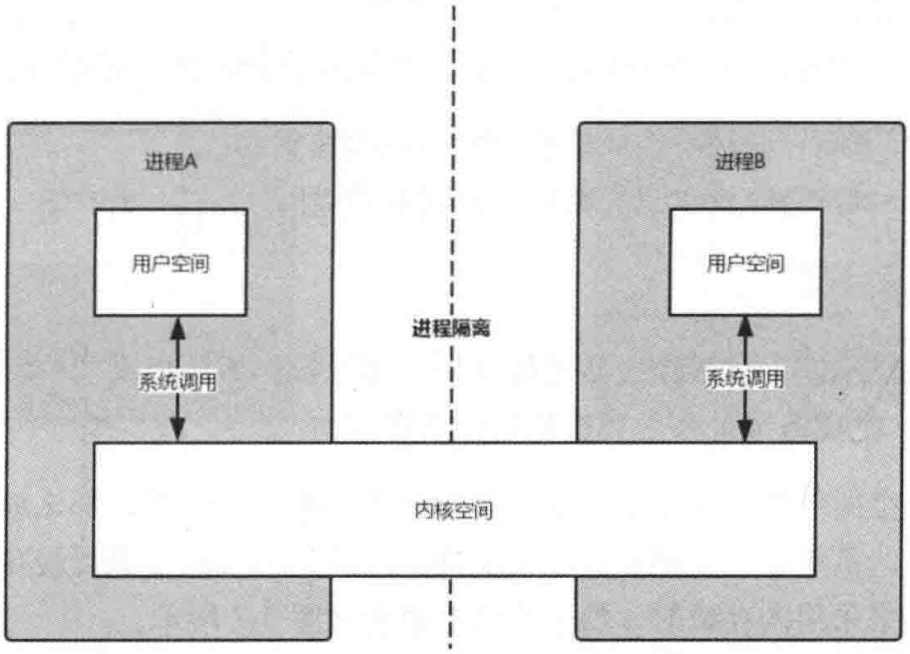


图 5-1　进程通信简单模型

1．内核空间和用户空间

当我们接触到 Linux 时，会听到两个术语：User space（用户空间）和 Kernel space（内核空间），那么它们的含义是什么呢？

为了保护用户进程，不能直接操作内核，以保证内核的安全，操作系统从逻辑上将虚拟空间划分为用户空间和内核空间。Linux 操作系统将较高的 1GB 供内核使用，称为内核空间，将较低的 3GB 供各进程使用，称为用户空间。

内核空间是 Linux 内核的运行空间，用户空间是用户程序的运行空间。为了保证内核的安全，它们是隔离的，即使用户的程序崩溃了，内核也不会受到影响。内核空间的数据是可以进程间共享的，而用户空间的数据则不可以。比如在图 5-1 中，进程 A 的用户空间是不能和进程 B 的用户空间共享的。

2．进程隔离

进程隔离指的是一个进程不能直接操作或者访问另一个进程，也就是进程 A 不可以直接访问进程 B 的数据。

3．系统调用

用户空间需要访问内核空间，就需要借助系统调用来实现。系统调用是用户空间访问内核空间的唯一方式，保证了所有资源访问都是在内核的控制下进行的，避免了用户程序对系统资源的越权访问，提升了系统的安全性和稳定性。

进程 A 和进程 B 的用户空间可以通过如下系统函数和内核空间交互。

- copy_*from*_user：将用户空间的数据复制到内核空间。
- copy_*to*_user：将内核空间的数据复制到用户空间。

4．内存映射

由于应用程序不能直接操作设备硬件地址，所以操作系统提供了一种机制：内存映射（Memory Map），把设备地址映射到进程虚拟内存区。

例如，用户空间需要读取磁盘的文件，如果不采用内存映射，那么就需要在内核空间建立一个页缓存，页缓存去复制磁盘中的文件，然后用户空间复制页缓存的文件，这就需要两次复制。如果采用内存映射，那么其映射模型如图 5-2 所示。

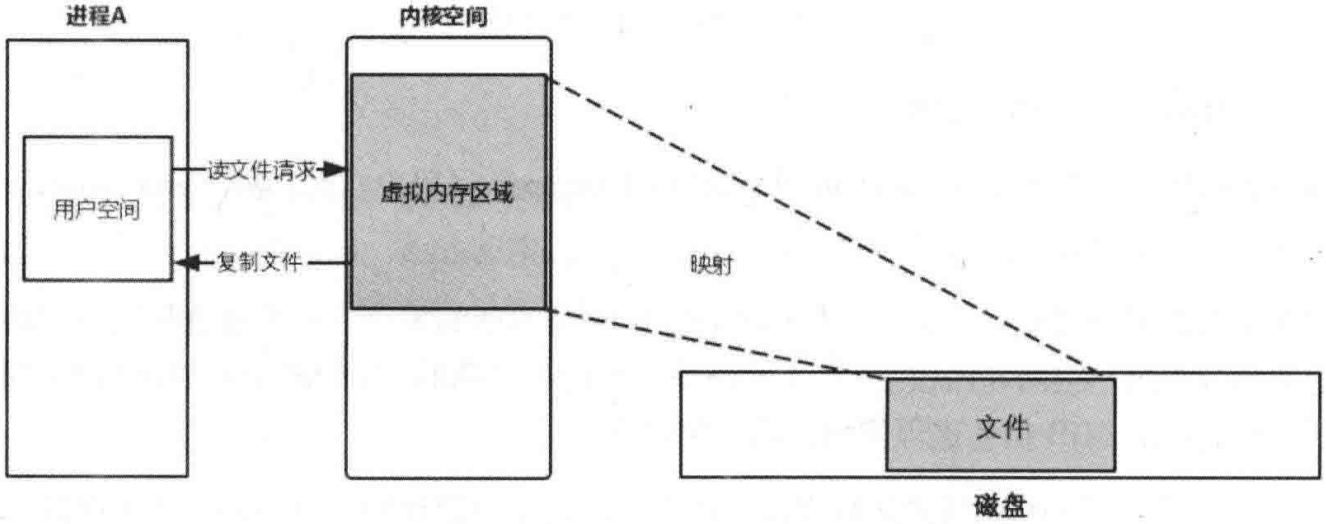


图 5-2　内存映射模型

由于新建了虚拟内存区域，那么磁盘文件和虚拟内存区域就可以直接映射，少了一次复制。

在 Linux 中通过系统调用函数 mmap 来实现内存映射。将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反映到内核空间，反之亦然。内存映射能减少数据复制次数，实现用户空间和内核空间的高效互动。

5. Linux 的 IPC 通信原理

了解了 Linux 中的几个概念后，就可以学习 Linux 的 IPC 通信原理了，Linux 的 IPC 通信模型 1 如图 5-3 所示。

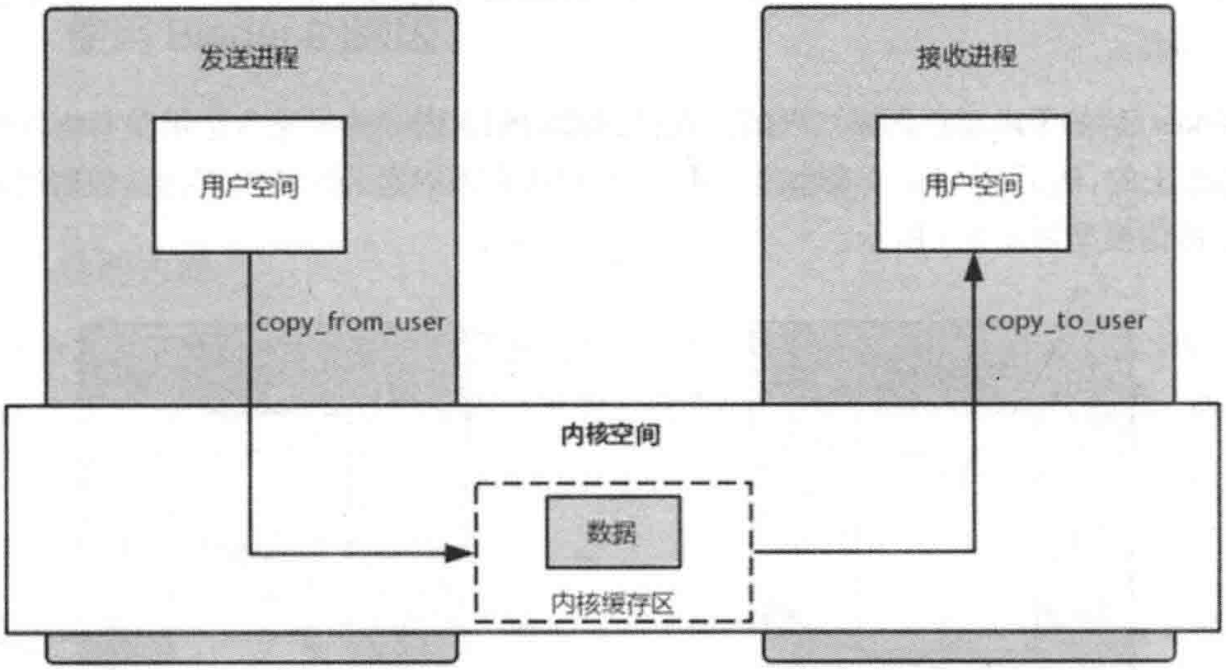


图 5-3　Linux 的 IPC 通信模型 1

内核程序在内核空间分配内存并开辟一块内核缓存区，发送进程通过 copy_*from*_user()函数将数据复制到内核空间的缓冲区中。同样，接收进程接收数据时在自己的用户空间开辟一块内核缓存区，然后内核程序调用 copy_*to*_user()函数将数据从内核缓存区复制到接收进程。这样数据发送进程和数据接收进程就完成了一次数据传输，也就是一次进程间的通信。

Linux 的 IPC 通信原理有以下两个问题。

（1）一次数据传递需要经历：用户空间→内核缓存区→用户空间，需要两次数据复制，这样效率不高。

（2）接收数据的缓存区由数据接收进程提供，但是接收进程并不知道需要多大的空间来存放将要传递过来的数据，因此只能开辟尽可能大的内存空间或者先调用 API 接收消息头来获取消息体的大小，以免浪费了空间或者时间。

6. Binder 的通信原理

Binder 是基于开源的 OpenBinder 实现的，OpenBinder 最早并不是由 Google 公司开发的，而是由 Be Inc 公司开发的，接着由 Palm Inc 公司负责开发。后来 OpenBinder 的作者 Dianne Hackborn 加入了 Google 公司，并负责 Android 平台的开发工作，把这项技术也带进了 Android。

Binder 是基于内存映射来实现的，在前面我们知道内存映射通常是用在有物理介质的文件系统上的，然后 Binder 没有物理介质，它使用内存映射是为了跨进程传递数据的，Linux 的 IPC 通信模型如图 5-4 所示。

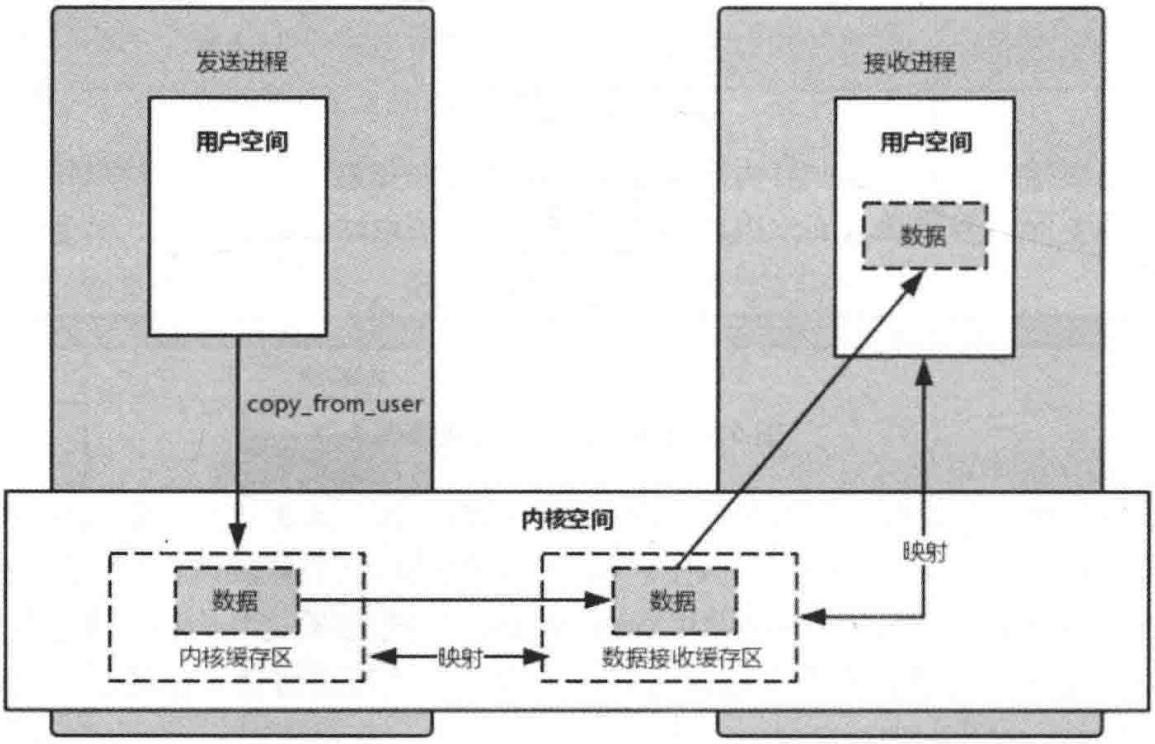


图 5-4　Linux 的 IPC 通信模型 2

Binder 通信的步骤如下所示。

（1）Binder 驱动在内核空间创建一个数据接收缓存区。

（2）在内核空间开辟一块内核缓存区，建立内核缓存区和数据接收缓存区之间的映射关系，以及数据接收缓存区和接收进程用户空间地址的映射关系。

（3）发送方进程通过 copy_*from*_user()函数将数据复制到内核中的内核缓存区，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程

的用户空间，这样便完成了一次进程间的通信。

整个过程只使用了一次复制，不会因为不知道数据的大小而浪费空间或者时间，这样效率更高。

5.1.2　使用 Binder 的原因

Android 是基于 Linux 内核的，Linux 提供了很多 IPC 机制，而 Android 自己却设计了 Binder 来进行通信，主要原因有以下四个方面。

1．性能方面

性能方面主要影响的因素是数据复制次数，管道、消息队列、Socket 的复制次数都是两次，性能不是很好，共享内存不需要复制，性能很好，Binder 的复制次数为一次，性能仅次于内存复制。

2．稳定性方面

Binder 是基于 C/S 架构的，这个架构通常采用两层结构，在技术上已经很成熟了，稳定性也是没有问题的。共享内存没有分层，难以控制，并发同步访问临界资源时，可能还会产生死锁。从稳定性的角度讲，Binder 是优于共享内存的。

3．安全方面

Android 是一个开源的系统，并且拥有开放性的平台，市场上应用来源很广，因此安全性对 Android 平台而言极其重要。

传统的 IPC 接收方无法获得对方可靠的进程用户 ID/进程 ID（UID/PID），无法鉴别对方身份。Android 为每个安装好的 App 分配了自己的 UID，通过进程的 UID 来鉴别进程身份。另外，Android 系统中的服务端会判断 UID/PID 是否满足访问权限，而对外只暴露客户端，加强了系统的安全性。

4．语言方面

Linux 是基于 C 语言的，C 语言是面向过程的，Android 应用层和 Java Framework 是基于 Java 语言的，Java 语言是面向对象的。Binder 本身符合面向对象的思想，因此作为 Android 的通信机制更合适。

从这四个方面来看，Linux 提供的大部分 IPC 机制无法和 Binder 相比较，而共享内

存只在性能方面优于 Binder，其他方面都劣于 Binder，这些就是 Android 要使用 Binder 来进行进程间通信的原因，当然系统中并不是所有进程通信都采用了 Binder，而是根据场景选择最合适的，比如 Zygote 进程与 AMS 通信使用的是 Socket，Kill Process 采用的是信号。

5.1.3 学习 Binder 的原因

Binder 机制在 Android 中的地位举足轻重，我们需要掌握的很多原理都和 Binder 有关。

（1）系统中的各个进程是如何通信的。

（2）Android 系统启动过程。

（3）AMS、PMS 的原理。

（4）四大组件的原理，比如 Activity 是如何启动的。

（5）插件化原理。

（6）系统服务的客户端和服务端是如何通信的（比如 MediaPlayer 和 MeidaPlayerService）。

以上只是列举了一小部分，简单来讲，比如系统在启动时，SystemServer 进程启动后会创建 Binder 线程池，其目的是通过 Binder 可以使在 SystemServer 进程中的服务能够和其他进程间通信。再比如我们常说的 AMS、PMS 都是基于 Binder 来实现的，拿 PMS 来说，PMS 运行在 SystemServer 进程，如果它想要和 DefaultContainerService 通信（用于检查和复制可移动文件的系统服务），就需要通过 Binder 来实现，因为 DefaultContainerService 运行在 com.android.defcontainer 进程。

还有一个比较常见的 C/S 架构间通信的问题，客户端的 MediaPlayer 和服务端的 MeidaPlayerService 不是运行在一个进程中的，同样需要 Binder 来实现通信。

可以说 Binder 机制是掌握系统底层原理的基石。根据 Android 系统的分层，Binder 机制主要分为 Java Binder、Native Binder、Kernel Binder，如图 5-5 所示。

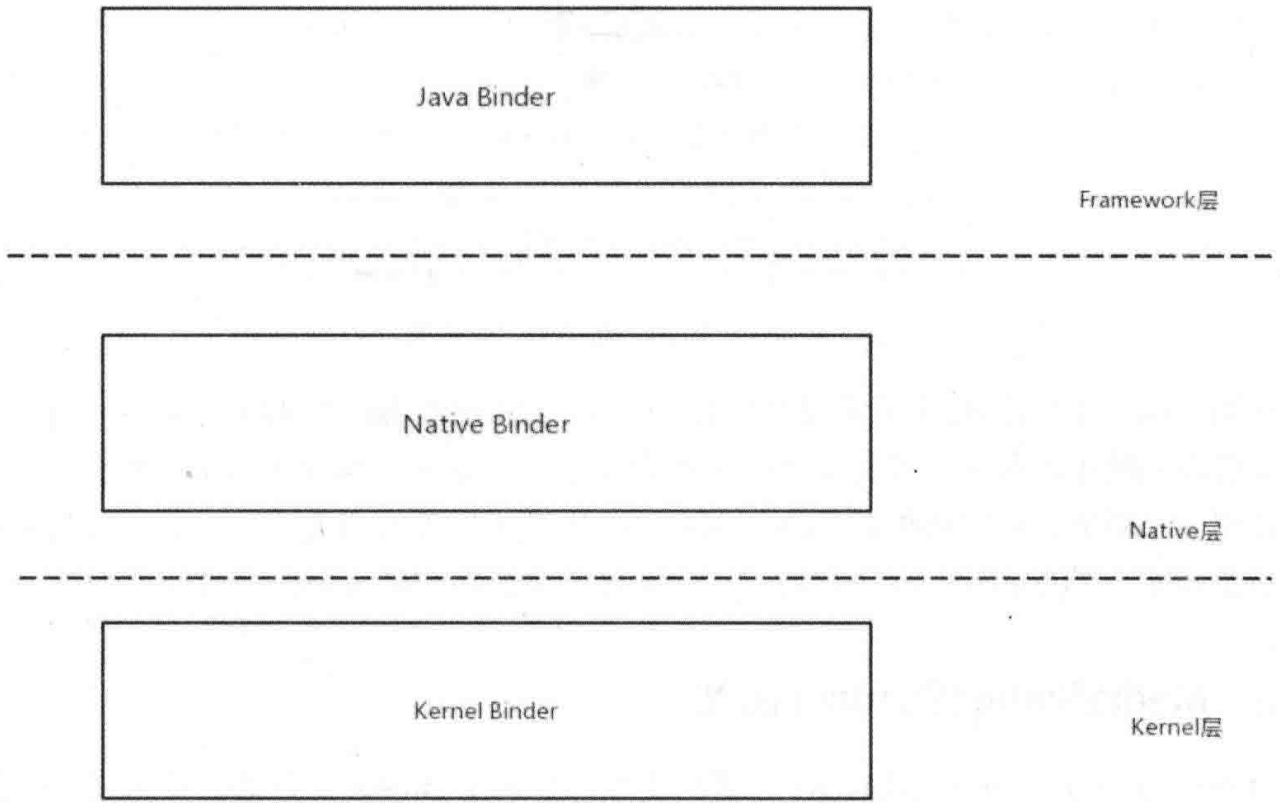


图 5-5　Binder 机制的组成

图 5-5 并没有给出 Binder 机制的具体细节，而是先给出了一个简单的概念图，随着后续知识的讲解，最终会完善这张图。

5.2　ServiceManager 中的 Binder 机制

Java Binder 是需要借助 Native Binder 来工作的，因此需要先了解 Native Binder，Native Binder 架构的原型就是基于 Binder 通信的 C/S 架构的，因此我们先从 C/S 架构开始学习。

5.2.1　基于 Binder 通信的 C/S 架构

在 Android 系统中，Binder 进程间通信的使用是很普遍的，在《Android 进阶之光》的最后一章，讲解了 MediaPlayer 框架，这个框架基于 C/S 架构，并采用 Binder 来进行进程间通信，如图 5-6 所示。

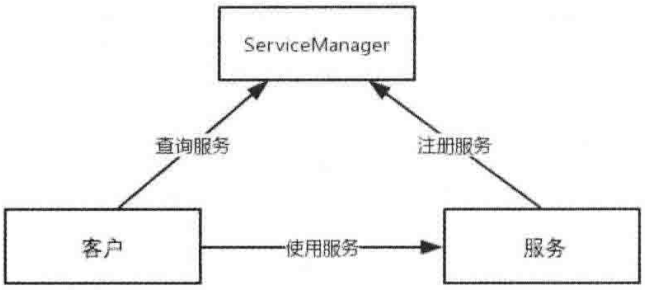


图 5-6　基于 Binder 通信的 C/S 架构

从图 5-6 中可以看出，除了常规 C/S 架构的客户端和服务端，还包括了 ServiceManager，它用于管理系统中的服务。首先 Server 进程会注册一些服务到 ServiceManager 中，如果客户要使用某个服务，则需要先到 ServiceManager 中查询服务的相关信息，然后根据服务的相关信息与服务所在的 Server 进程建立通信通路，这样客户就可以使用服务了。

5.2.2　MediaServer 的 main 函数

Client、Server、ServiceManager 三者的交互都是基于 Binder 通信的，那么任意两者的交互都可以说明 Binder 通信的原理，可以说 Native Binder 的原理的核心就是 ServiceManager 的原理，为了更好地了解 ServiceManager，这里使用 MediaPlayer 框架来举例，它也是学习多媒体时必须要掌握的知识点。

MediaPlayer 框架的简单框架图如图 5-7 所示。

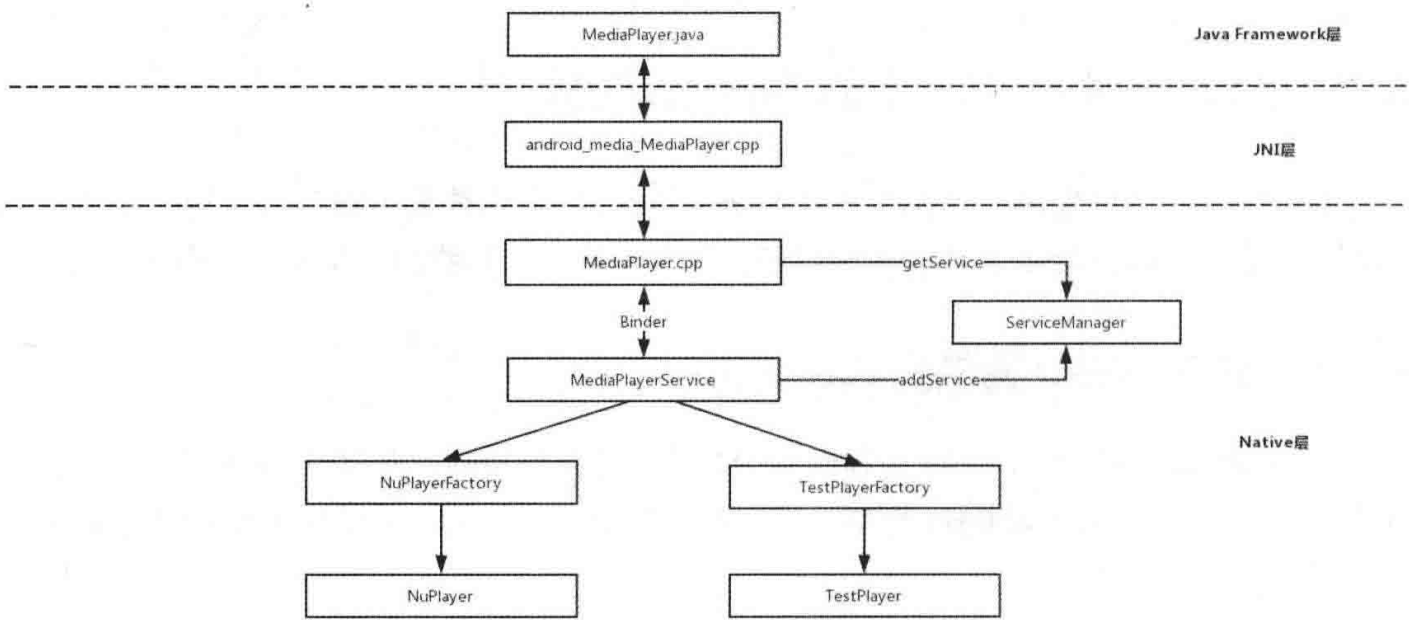


图 5-7　MediaPlayer 框架的简单框架图

由图 5-7 可知，MediaPlayer 和 MediaPlayerService 是通过 Binder 来进行通信的，MediaPlayer 是客户端，MediaPlayerService 是服务端，MediaPlayerService 是系统多媒体服务的一种，系统多媒体服务是由 MediaServer 服务进程提供的，它是一个可执行程序，在 Android 系统启动时，MediaServer 也被启动，它的入口函数如下所示。

frameworks/av/media/mediaserver/main_mediaserver.cpp

```
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);
    //获取 ProcessState 实例
    sp<ProcessState> proc(ProcessState::self());//1
    sp<IServiceManager> sm(defaultServiceManager());//2
    ALOGI("ServiceManager: %p", sm.get());
    InitializeIcuOrDie();
    //注册 MediaPlayerService
    MediaPlayerService::instantiate();//3
    ResourceManagerService::instantiate();
    registerExtensions();
    //启动 Binder 线程池
    ProcessState::self()->startThreadPool();
    //当前线程加入线程池
    IPCThreadState::self()->joinThreadPool();
}
```

注释 1 处用来获取 ProcessState 实例，在这一过程中会打开/dev/binder 设备，并使用 mmap 为 Binder 驱动分配一个虚拟地址空间用来接收数据。

注释 2 处用来得到 IServiceManager，通过 IServiceManager，其他进程就可以和当前的 ServiceManager 交互，这里就用到了 Binder 通信。

注释 3 处用来注册 MediaPlayerService。

除了注释 3 处的知识点会在 5.3 节进行介绍，注释 1 处和注释 2 处的内容，本节会分别对其进行介绍，先介绍 ProcessState 实例。

5.2.3　每个进程唯一的 ProcessState 实例

ProcessState 实例代表进程的状态，先来查看 ProcessState 实例的 self 函数，代码如下所示。

frameworks/native/libs/binder/ProcessState.cpp

```
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState("/dev/binder");//1
    return gProcess;
}
```

这里采用了单例模式，确保每个进程只有一个 ProcessState 实例。注释 1 处创建了一个 ProcessState 实例，参数为/dev/binder。接着来查看 ProcessState 实例的构造函数，代码如下所示。

frameworks/native/libs/binder/ProcessState.cpp

```
ProcessState::ProcessState(const char *driver)
    : mDriverName(String8(driver))
    , mDriverFD(open_driver(driver))//1
    , mVMStart(MAP_FAILED)
    , mThreadCountLock(PTHREAD_MUTEX_INITIALIZER)
    , mThreadCountDecrement(PTHREAD_COND_INITIALIZER)
    , mExecutingThreadsCount(0)
    , mMaxThreads(DEFAULT_MAX_BINDER_THREADS)
    , mStarvationStartTimeMs(0)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
mDriverFD, 0);//2
        if (mVMStart == MAP_FAILED) {
            ALOGE("Using %s failed: unable to mmap transaction memory.\n",
mDriverName.c_str());
            close(mDriverFD);
            mDriverFD = -1;
            mDriverName.clear();
        }
    }
    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.
Terminating.");
```

```
}
```

ProcessState 实例的构造函数中调用了很多函数，需要注意注释 1 处打开了/dev/binder 设备。

注释 2 处的 mmap 函数会在内核虚拟地址空间中申请一块与用户虚拟内存相同大小的内存，然后申请物理内存，将同一块物理内存分别映射到内核虚拟地址空间和用户虚拟内存空间，实现了内核虚拟地址空间和用户虚拟内存空间的数据同步操作，也就是内存映射。

mmap 函数用于对 Binder 设备进行内存映射，除了 mmap 函数还有 open 函数、ioctl 函数，接下来学习这两个函数的作用。

注释 1 处的 open_driver 函数的代码如下所示。

frameworks/native/libs/binder/ProcessState.cpp

```
static int open_driver(const char *driver)
{
    int fd = open(driver, O_RDWR | O_CLOEXEC);//1
    if (fd >= 0) {
        ...
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);//2
        if (result == -1) {
            ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
        }
    } else {
        ALOGW("Opening '%s' failed: %s\n", driver, strerror(errno));
    }
    return fd;
}
```

注释 1 处打开了/dev/binder 设备并返回文件操作符 fd，这样就可以操作内核的 Binder 驱动了。注释 2 处的 ioctl 函数的作用就是和 Binder 设备进行参数的传递，这里的 ioctl 函数将 binder 支持的最大线程数设定为 15（maxThreads 的值为 15）。最终 open_driver 函数返回文件操作符 fd。

总体来说，ProcessState 实例有以下两个重要作用。

（1）打开/dev/binder 设备并设定 Binder 最大的支持线程数。

（2）通过 mmap 为 binder 分配一块虚拟地址空间，达到内存映射的目的。

5.2.4 解析 ServiceManager 中的 Binder 机制

回到 5.2.2 节的 MediaServer 的入口函数，注释 2 处调用了 defaultServiceManager 函数，代码如下所示。

frameworks/native/libs/binder/IServiceManager.cpp

```
sp<IServiceManager> defaultServiceManager()
{
   if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

   {
      AutoMutex _l(gDefaultServiceManagerLock);
      while (gDefaultServiceManager == NULL) {
         gDefaultServiceManager = interface_cast<IServiceManager>(
            ProcessState::self()->getContextObject(NULL));//1
         if (gDefaultServiceManager == NULL)
            sleep(1);
      }
   }

   return gDefaultServiceManager;
}
```

从 IServiceManager 所在的文件路径可以知道，ServiceManager 中不但使用了 Binder 通信，而且它自身也是属于 Binder 体系的。defaultServiceManager 中同样使用了单例，注释 1 处的 interface_cast 函数生成了 gDefaultServiceManager，其内部调用了 ProcessState 实例的 getContextObject 函数，代码如下所示。

frameworks/native/libs/binder/ProcessState.cpp

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /*caller*/)
{
   return getStrongProxyForHandle(0);
}

sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
   sp<IBinder> result;
   AutoMutex _l(mLock);
   handle_entry* e = lookupHandleLocked(handle);//1
   if (e != NULL) {
      IBinder* b = e->binder;
      if (b == NULL || !e->refs->attemptIncWeak(this)) {
         if (handle == 0) {
```

```
            Parcel data;
            status_t status = IPCThreadState::self()->transact(
                  0, IBinder::PING_TRANSACTION, data, NULL, 0);
            if (status == DEAD_OBJECT)
              return NULL;
        }
        b = BpBinder::create(handle);//2
        e->binder = b;//3
        if (b) e->refs = b->getWeakRefs();
        result = b;
      } else {
        result.force_set(b);
        e->refs->decWeak(this);
      }
    }

    return result;
  }
```

getContextObject 函数中直接调用了 getStrongProxyForHandle 函数，它的参数的值为 0，那么 handle 的值就为 0，handle 是一个资源标识。注释 1 处查询了这个资源标识对应的资源（handleentry）是否存在，如果不存在就会在注释 2 处新建 BpBinder，并在注释 3 处赋值给 handleentry 的 binder，最终返回的 result 的值为 BpBinder。

5.2.4.1　BpBinder 和 BBinder

讲到 BpBinder 就要提到 BBinder，它们是 Binder 通信的“双子星”，都继承了 IBinder。BpBinder 是客户端与服务端交互的代理类，而 BBinder 则代表了服务端。BpBinder 和 BBinder 是一一对应的，BpBinder 会通过 handle 来找到对应的 BBinder。

我们知道在 ServiceManager 中创建了 BpBinder，通过 handle（值为 0）可以找到对应的 BBinder，如图 5-8 所示。

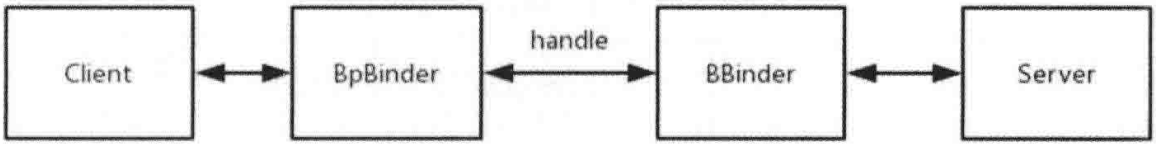


图 5-8　BpBinder 和 BBinder 一一对应

分析完 ProcessState 实例的 getContextObject 函数，接着回到 interface_cast 函数。

```
gDefaultServiceManager = interface_cast<IServiceManager>(
```

```
            ProcessState::self()->getContextObject(NULL));
```

interface_cast 具体实现如下所示。

frameworks/native/libs/binder/include/binder/IInterface.h

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
   return INTERFACE::asInterface(obj);
}
```

当前的场景中，INTERFACE 的值为 IServiceManager，那么替换后代码如下所示。

```
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
   return IServiceManager::asInterface(obj);
}
```

我们接着来分析 IServiceManager。

5.2.4.2 解密 IServiceManager

BpBinder 和 BBinder 负责 Binder 的通信，而 IServiceManager 用于处理 ServiceManager 的业务，IServiceManager 定义在 IServiceManager.h 中。

frameworks/native/libs/binder/include/binder/IServiceManager.h

```
class IServiceManager : public IInterface
{
public:
   DECLARE_META_INTERFACE(ServiceManager)//1
   ...
   //一些操作 Service 的函数
   virtual sp<IBinder>       getService( const String16& name) const = 0;
   virtual sp<IBinder>       checkService( const String16& name) const = 0;
   virtual status_t addService(const String16& name, const sp<IBinder>& service,
                        bool allowIsolated = false,
                        int dumpsysFlags = DUMP_FLAG_PRIORITY_DEFAULT) = 0;
   virtual Vector<String16> listServices(int dumpsysFlags =
DUMP_FLAG_PRIORITY_ALL) = 0;
   enum {
      GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
      CHECK_SERVICE_TRANSACTION,
      ADD_SERVICE_TRANSACTION,
      LIST_SERVICES_TRANSACTION,
```

```
    };
};
```

可以看到 IServiceManager 继承了 IInterface，其内部定义了一些常量和一些操作 Service 的函数，在注释 1 处调用了 DECLARE_META_INTERFACE 宏，它的定义在 IInterface.h 中，如下所示。

frameworks/native/libs/binder/include/binder/IInterface.h

```
#define DECLARE_META_INTERFACE(INTERFACE)                                \
    static const ::android::String16 descriptor;                         \
    static ::android::sp<I##INTERFACE> asInterface(                      \
            const ::android::sp<::android::IBinder>& obj);               \
    virtual const ::android::String16& getInterfaceDescriptor() const;   \
    I##INTERFACE();                                                      \
    virtual ~I##INTERFACE();
```

其中 INTERFACE 的值为 ServiceManager，经过替换后的代码如下所示。

```
    static const ::android::String16 descriptor;
    //定义 asInterface 函数
    static ::android::sp<IServiceManager> asInterface(
            const ::android::sp<::android::IBinder>& obj);
    virtual const ::android::String16& getInterfaceDescriptor() const;
    //定义 IServiceManager 构造函数
    IServiceManager();
    //定义 IServiceManager 析构函数
    virtual ~IServiceManager();
```

从 DECLARE_META_INTERFACE 宏的名称和上面的代码中，可以发现它主要声明了一些函数和一个变量。那么这些函数和变量的实现在哪里呢？答案还在 IInterface.h 中，IMPLEMENT_META_INTERFACE 宏，代码如下所示。

frameworks/native/libs/binder/include/binder/IInterface.h

```
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                        \
    const ::android::String16 I##INTERFACE::descriptor(NAME);            \
    const ::android::String16&                                           \
            I##INTERFACE::getInterfaceDescriptor() const {               \
        return I##INTERFACE::descriptor;                                 \
    }                                                                    \
    ::android::sp<I##INTERFACE> I##INTERFACE::asInterface(               \
            const ::android::sp<::android::IBinder>& obj)                \
    {                                                                    \
        ::android::sp<I##INTERFACE> intr;                                \
        if (obj != NULL) {                                               \
```

```
        intr = static_cast<I##INTERFACE*>(                          \
            obj->queryLocalInterface(                               \
                I##INTERFACE::descriptor).get());                   \
        if (intr == NULL) {                                         \
            intr = new Bp##INTERFACE(obj);                          \
        }                                                           \
    }                                                               \
    return intr;                                                    \
}                                                                   \
I##INTERFACE::I##INTERFACE() { }                                    \
I##INTERFACE::~I##INTERFACE() { }                                   \
```

DECLARE_META_INTERFACE 宏和 IMPLEMENT_META_INTERFACE 宏是配合使用的，很多系统服务都使用了它们，IServiceManager 使用 IMPLEMENT_META_INTERFACE 宏只有一行代码，如下所示。

frameworks/native/libs/binder/IServiceManager.cpp

```
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

IMPLEMENT_META_INTERFACE 宏的 INTERFACE 值为 ServiceManager，NAME 值为"android.os.IServiceManager"，进行替换后的代码如下所示。

```
    const ::android::String16
IServiceManager::descriptor("android.os.IServiceManager");
    const ::android::String16&
        IServiceManager::getInterfaceDescriptor() const {
    return IServiceManager::descriptor;
    }
    //实现了 asInterface 函数
    ::android::sp<IServiceManager> IServiceManager::asInterface(
        const ::android::sp<::android::IBinder>& obj)
    {
    ::android::sp<IServiceManager> intr;
    if (obj != NULL) {
        intr = static_cast<IServiceManager>(
            obj->queryLocalInterface(
                IServiceManager::descriptor).get());
        if (intr == NULL) {
            intr = new BpServiceManager(obj);//1
        }
    }
    return intr;
    }
    IServiceManager::IServiceManager() { }
```

```
IServiceManager::~IServiceManager() { }
```

注释 1 处新建了一个 BpServiceManager，传入的参数 obj 的值为 BpBinder。到这里我们也就明白了，asInterface 函数就是用 BpBinder 为参数创建了 BpServiceManager，从而推断出 interface_cast 函数创建了 BpServiceManager，再继续往上推断，IServiceManager 的 defaultServiceManager 函数返回的就是 BpServiceManager。

BpServiceManager 的作用是什么呢？先来看 BpServiceManager 的构造函数。

frameworks/native/libs/binder/IServiceManager.cpp

```
class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    explicit BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }
...
}
```

impl 的值其实就是 BpBinder，BpServiceManager 的构造函数调用了基类 BpInterface 的构造函数。

frameworks/native/libs/binder/include/binder/IInterface.h

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
...
};
```

BpInterface 继承了 BpRefBase，BpRefBase 的实现如下所示。

frameworks/native/libs/binder/Binder.cpp

```
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);
        mRefs = mRemote->createWeak(this);
    }
}
```

mRemote 是一个 IBinder* 指针，它最终的指向为 BpBinder，也就是说 BpServiceManager 的 mRemote 指向了 BpBinder。那么 BpServiceManager 的作用就是实现了 IServiceManager，并且通过 BpBinder 来实现通信。

5.2.4.3 IServiceManager 家族

可能前面讲的内容会使我们有些头晕，这是因为对各个类的关系还不明确，通过学习图 5-9 的 IServiceManager 家族 UML 图也许就会豁然开朗。

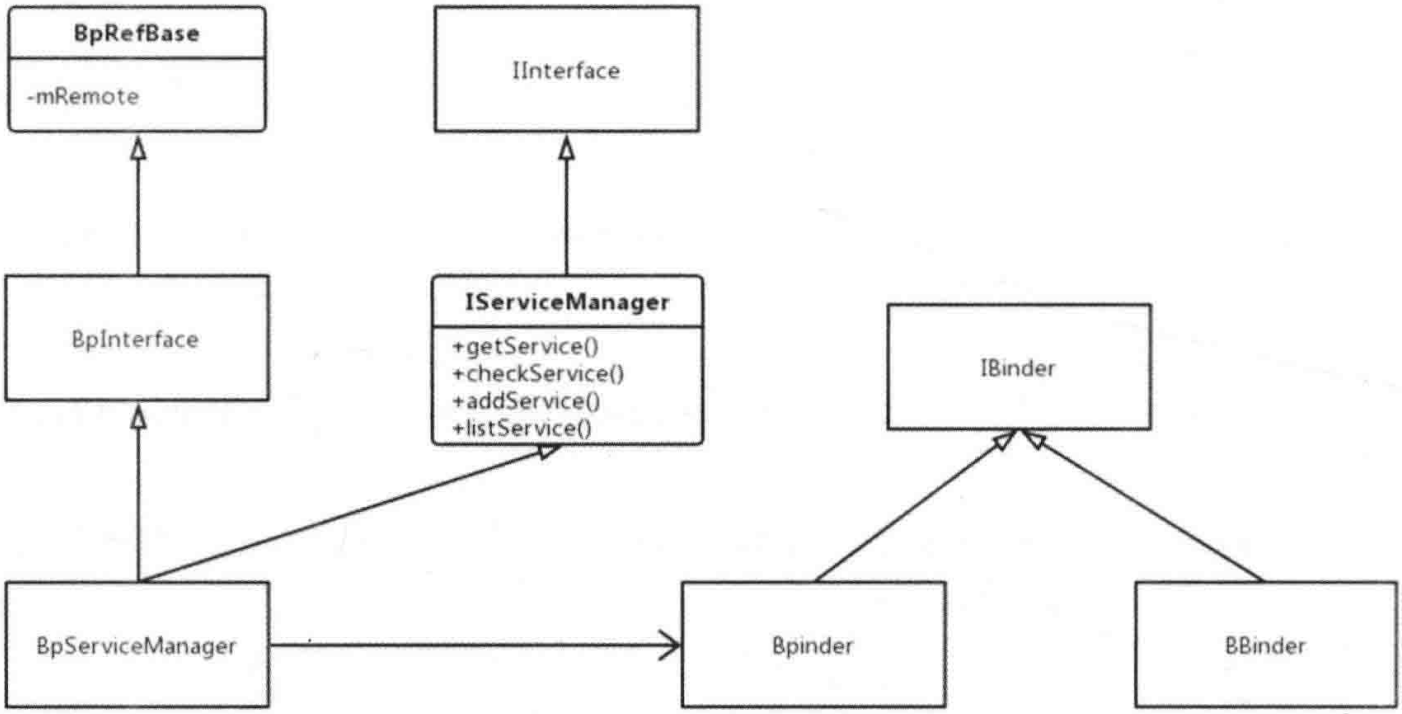


图 5-9 IServiceManager 家族 UML 图

由图 5-9 可知：

（1）BpBinder 与 BBinder 都和通信有关，它们都继承自 IBinder。

（2）BpServiceManager 派生自 IServiceManager，它们都和业务有关。

（3）BpRefBase 包含了 mRemote，通过不断派生，BpServiceManager 也同样包含 mRemote，它指向了 BpBinder，通过 BpBinder 来实现通信。

5.2.5　小结

本节我们学习了 Binder 通信的 C/S 架构，也知道了 Native Binder 的原理的核心其实就是 ServiceManager 的原理，为了讲解 ServiceManager 的原理，我们需要一个框架来举例，那就是 MediaPlayer 框架。在讲解 MediaServer 的入口函数时，我们遇到了三个问题，其中前两个问题相关的知识点，即 ProcessState 实例和 IServiceManager 都已经讲解了，5.3 节会讲解第三个问题，即 MediaPlayerService 是如何注册的。

5.3　系统服务的注册过程

通过了解 MediaPlayerService 是如何注册的，可以得知系统服务的注册过程。

5.3.1　从调用链角度说明 MediaPlayerService 的注册过程

先来查看 MediaServer 的入口函数，代码如下所示。

frameworks/av/media/mediaserver/main/mediaserver.cpp

```
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);
    //获取 ProcessState 实例
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm(defaultServiceManager());
    ALOGI("ServiceManager: %p", sm.get());
    InitializeIcuOrDie();
    //注册 MediaPlayerService
    MediaPlayerService::instantiate();//1
    ResourceManagerService::instantiate();
    registerExtensions();
    //启动 Binder 线程池
    ProcessState::self()->startThreadPool();
    //当前线程加入线程池
    IPCThreadState::self()->joinThreadPool();
}
```

这段代码中的很多内容在 5.2.1 节都已经介绍过了，接着分析注释 1 处的代码。

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.player"), new MediaPlayerService, ());
}
```

defaultServiceManager 返回的是 BpServiceManager，参数是一个字符串和 MediaPlayerService，类似于 Key/Value 的形式来完成注册，接着查看 addService 函数。

frameworks/native/libs/binder/IServiceManager.cpp

```
 virtual status_t addService(const String16& name, const sp<IBinder>& service,
                             bool allowIsolated, int dumpsysPriority) {
        Parcel data, reply;                //数据包
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);          //name 值为"media.player"
        data.writeStrongBinder(service);//service 值为 MediaPlayerService
        data.writeInt32(allowIsolated ? 1 : 0);
        data.writeInt32(dumpsysPriority);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
&reply);//1
        return err == NO_ERROR ? reply.readExceptionCode() : err;
    }
```

data 是一个数据包，后面的代码会不断将数据写入 data 中，注释 1 处的 remote()指的是 mRemote，也就是 BpBinder。addService 函数的作用就是将请求数据打包成 data，然后传给 BpBinder 的 transact 函数，代码如下所示。

frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder 将逻辑处理交给 IPCThreadState，先来了解 IPCThreadState::self()的作用。

frameworks/native/libs/binder/IPCThreadState.cpp

```
IPCThreadState* IPCThreadState::self()
{
   //首次进来 gHaveTLS 的值为 false
   if (gHaveTLS) {
restart:
      const pthread_key_t k = gTLS;//1
      IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);//2
      if (st) return st;
      return new IPCThreadState;//3
   }
   ...
   pthread_mutex_unlock(&gTLSMutex);
   goto restart;
}
```

注释 1 处的 TLS 的全称为 Thread local storage，它指的是线程本地存储空间，在每个线程中都有 TLS，并且其在线程间不共享。注释 2 处用于获取 TLS 中的内容并赋值给 IPCThreadState* 指针。注释 3 处会新建一个 IPCThreadState，这里可以得知 IPCThreadState::self()实际是创建 IPCThreadState 的，它的构造函数如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

```
IPCThreadState::IPCThreadState()
   : mProcess(ProcessState::self()),
     mStrictModePolicy(0),
     mLastTransactionBinderFlags(0)
{
   pthread_setspecific(gTLS, this);//1
   clearCaller();
   mIn.setDataCapacity(256);
   mOut.setDataCapacity(256);
}
```

注释 1 处的 pthread_setspecific 函数用于设置 TLS，将 IPCThreadState::self()获得的 TLS 和自身传入。IPCThreadState 中还包含 mIn 和 mOut，其中 mIn 用来接收来自 Binder 驱动的数据，mOut 用来存储发往 Binder 驱动的数据，它们默认大小都为 256 字节。

知道了 IPCThreadState 的构造函数，再回来查看 IPCThreadState 的 transact 函数。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
```

```
{
    status_t err;

    flags |= TF_ACCEPT_FDS;
    ...
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);//1

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        ...
        if (reply) {
            err = waitForResponse(reply);//2
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        ...
    } else {
        //不需要等待 reply 的分支
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

调用 BpBinder 的 transact 函数实际上就是调用 IPCThreadState 的 transact 函数。注释 1 处的 writeTransactionData 函数用于传输数据，其中第一个参数 BC_TRANSACTION 代表向 Binder 驱动发送的命令协议，该命令协议都以 BC_开头，而 Binder 驱动返回的命令协议都以 BR_开头。这个命令协议我们在后面会再次提到它。

现在分别来分析注释 1 处的 writeTransactionData 函数和注释 2 处的 waitForResponse 函数。

1. writeTransactionData 函数分析

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
```

```
    binder_transaction_data tr;//1

    tr.target.ptr = 0;
    tr.target.handle = handle;//2
    tr.code = code;
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();//3
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(binder_size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = reinterpret_cast<uintptr_t>(statusBuffer);
        tr.offsets_size = 0;
        tr.data.ptr.offsets = 0;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

注释 1 处的 binder_transaction_data 结构体（tr 结构体）是向 Binder 驱动通信的数据结构的，注释 2 处将 handle 传递给 target 的 handle，用于标识目标，这里的 handle 的值为 0，代表了 ServiceManager。注释 3 处对数据 data 进行错误检查，如果没有错误就将数据赋值给对应的 tr 结构体。最后会将 BC_TRANSACTION 和 tr 结构体写入 mOut 中。

上述代码调用链的时序图如图 5-10 所示。

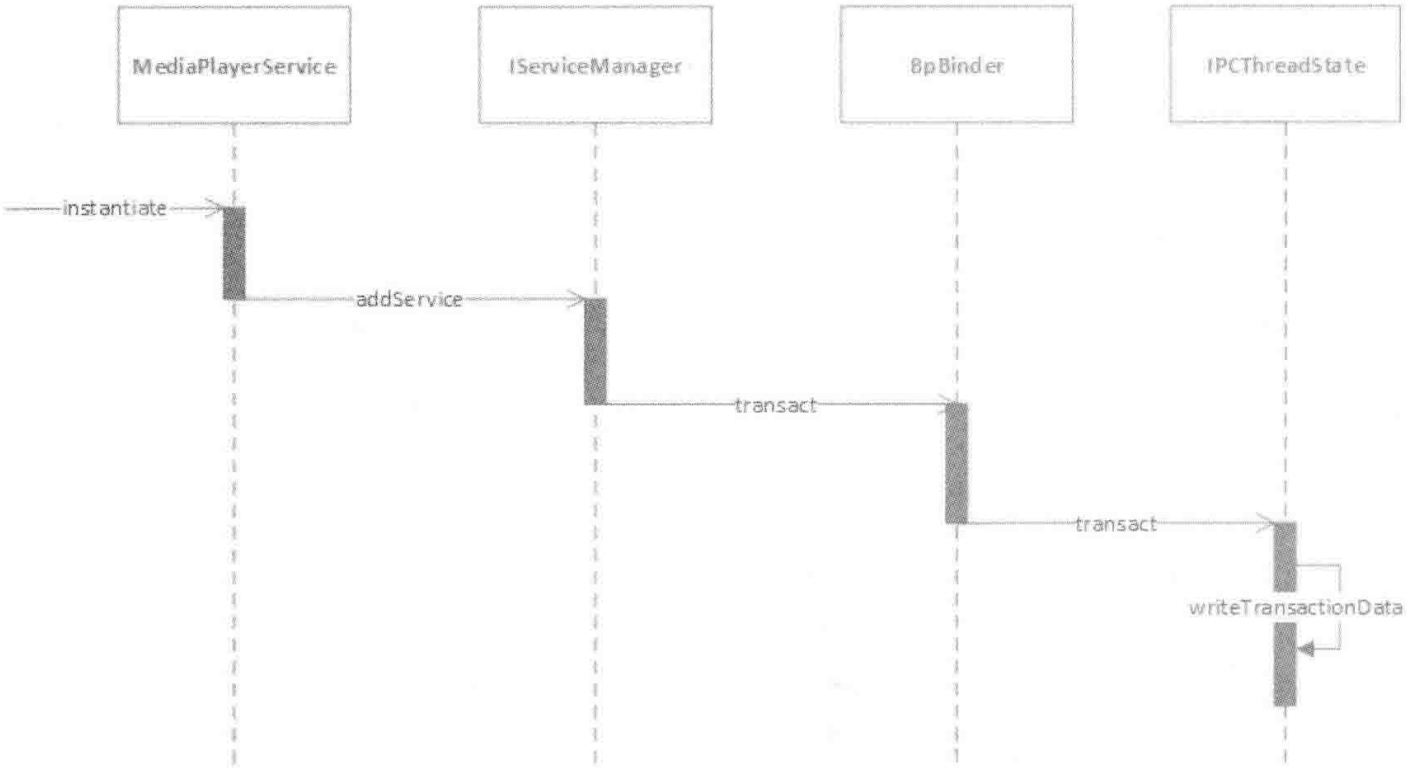


图 5-10　代码调用链的时序图

2. waitForResponse 函数分析

接着来查看 waitForResponse 函数的作用，waitForResponse 函数中的 case 语句有很多，这里截取部分代码。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;//1
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
        cmd = (uint32_t)mIn.readInt32();
        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
```

```
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;
        ...
        default:
            //处理各种命令协议
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }
    finish:
        ...
        return err;
    }
```

注释 1 处的 talkWithDriver 函数的内部通过 ioctl 函数与 Binder 驱动进行通信，代码如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }
    //和 Binder 驱动通信的结构体
    binder_write_read bwr; //1
    //mIn 是否有可读的数据，接收的数据存储在 mIn
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();//2
    //这时 doReceive 的值为 true
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();//3
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
    ...
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
```

```
    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(__ANDROID__)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)//4
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
    ...
    } while (err == -EINTR);
    ...
    return err;
}
```

注释 1 处的 binder_write_read 是和 Binder 驱动通信的结构体，在注释 2 和注释 3 处将 mOut、mIn 赋值给 binder_write_read 的相应字段，最终通过注释 4 处的 ioctl 函数和 Binder 驱动进行通信，这一部分涉及 Kernel Binder 的内容就不再详细介绍了，我们只需要知道在 Kernel Binder 中会记录服务名和 handle，用于后续的服务查询。

3. 小结

从调用链的角度来看，MediaPlayerService 是如何注册的并不复杂，因为这里只是简单地介绍了一个调用链分支，可以总结为以下几个步骤。

（1）addService 函数将数据打包发送给 BpBinder 来进行处理。

（2）BpBinder 新建一个 IPCThreadState 对象，并将通信的任务交给 IPCThreadState。

（3）IPCThreadState 的 writeTransactionData 函数用于将命令协议和数据写入 mOut 中。

（4）IPCThreadState 的 waitForResponse 函数主要有两个作用，一个作用是通过 ioctl 函数操作，mOut 和 mIn 与 Binder 驱动进行数据交互，另一个作用是处理各种命令协议。

5.3.2　从进程角度说明 MediaPlayerService 的注册过程

MediaPlayerService 的注册还涉及了进程，如图 5-11 所示。

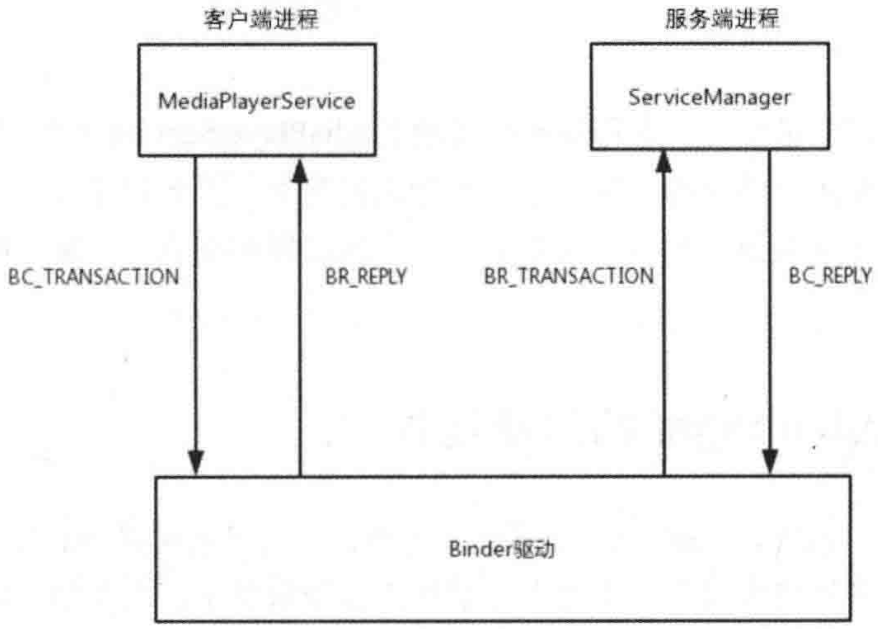


图 5-11　MediaPlayerService 的注册过程

从图 5-11 中可以看出 MediaPlayerService 的注册过程是以 C/S 架构为基础的，注册过程是在 MediaPlayerService 进行的，它是客户端，用于请求添加系统服务。而服务端则指的是 ServiceManager，用于完成系统服务的添加。

客户端和服务端分别运行在两个进程中，通过 Binder 来进行通信。更详细点描述，就是两端通过向 Binder 驱动发送命令协议来完成系统服务的添加。这其中命令协议非常多，过程也比较复杂，这里对命令协议进行了简化，只涉及了四个命令协议，其中 BC_TRANSACTION 和 BR_TRANSACTION 过程是一个完整的事务，BC_REPLY 和 BR_REPLY 是一个完整的事务。

简化步骤如下所示。

（1）客户端向 Binder 驱动发送 BC_TRANSACTION 命令。

（2）Binder 驱动接收到请求后生成 BR_TRANSACTION 命令，唤醒服务端的线程后将 BR_TRANSACTION 命令发送给 ServiceManager。

（3）服务端中的服务注册完成后，生成 BC_REPLY 命令发送给 Binder 驱动。

（4）Binder 驱动生成 BR_REPLY 命令，唤醒客户端的线程后将 BR_REPLY 命令发送

给客户端。

通过这些协议命令来驱动并完成系统服务的注册。

5.3.3 小结

本节分别从调用链角度和进程角度来讲解 MediaPlayerService 是如何注册的，间接地总结出了系统服务是如何注册的。这两个角度都比较复杂，因此这里分别对这两个角度进行简化，作为应用开发人员，我们不需要注重太多的过程和细节，了解大概的步骤即可。

5.4 ServiceManager 的启动过程

在 5.3 节中，我们以 MediaPlayerService 为例，讲解了系统服务的注册过程，接下来要讲解的是系统服务的获取过程，但是在了解获取系统服务前，首先要了解 ServiceManager 的启动过程，这样更有助于理解系统服务的注册过程和获取过程。

另外还有一点需要说明的是，如果想要了解 ServiceManager 的启动过程，就需要查看 Kernel Binder 部分的源码，这部分代码在内核源码中，AOSP 源码是不包括内核源码的，因此需要单独下载，具体的内容见 1.3 节。

ServiceManager 是 init 进程负责启动的，具体是在解析 init.rc 配置文件时启动的，init 进程是在系统启动时启动的，因此 ServiceManager 亦是如此，不理解 init 进程和 init.rc 配置文件的可以看《Android 进阶解密》第 2 章。

rc 文件内部由 Android 初始化语言编写（Android Init Language）的脚本，它主要包含五种类型语的句：Action、Commands、Services、Options 和 Import。

从 Android 7.0 开始，对 init.rc 配置文件进行了拆分，每个 init.rc 配置文件服务一个 rc 文件。ServiceManager 的启动脚本在 servicemanager.rc 中，代码如下所示。

frameworks/native/cmds/servicemanager/servicemanager.rc

```
service servicemanager /system/bin/servicemanager
   class core animation
   user system  //1
   group system readproc
   critical //2
   onrestart restart healthd
```

```
onrestart restart zygote
onrestart restart audioserver
onrestart restart media
onrestart restart surfaceflinger
onrestart restart inputflinger
onrestart restart drm
onrestart restart cameraserver
onrestart restart keystore
onrestart restart gatekeeperd
writepid /dev/cpuset/system-background/tasks
shutdown critical
```

service 用于通知 init 进程创建名为 servicemanager 的进程，这个 servicemanager 进程执行程序的路径为/system/bin/servicemanager。

注释 1 处的关键字 user 说明 servicemanager 是以用户 system 的身份运行的，注释 2 处的 critical 说明 servicemanager 是系统中的关键服务，关键服务是不会退出的，如果退出了，系统就会重启，当系统重启时就会启动用 onrestart 关键字修饰的进程，比如 zygote、media、surfaceflinger 等。

servicemanager 的入口函数在 servicemanager.c 中，如下所示。

frameworks/native/cmds/servicemanager/servicemanager.c

```
int main(int argc, char** argv)
{
   struct binder_state *bs;//1
   union selinux_callback cb;
   char *driver;

   if (argc > 1) {
      driver = argv[1];
   } else {
      driver = "/dev/binder";
   }
   bs = binder_open(driver, 128*1024);//2
   ...
   if (binder_become_context_manager(bs)) {//3
      ALOGE("cannot become context manager (%s)\n", strerror(errno));
      return -1;
   }
   ...
   if (getcon(&service_manager_context) != 0) {
      ALOGE("SELinux: Failed to acquire service_manager context. Aborting.\n");
```

```
        abort();
    }
    binder_loop(bs, svcmgr_handler);//4

    return 0;
}
```

注释 1 处的 binder_state 结构体用来存储 binder 的三个信息。

```
struct binder_state
{
    int fd;             //binder 设备的文件描述符
    void *mapped;       //binder 设备文件映射到进程的地址空间
    size_t mapsize;     //内存映射后，系统分配的地址空间的大小，默认为 128KB
};
```

main 函数主要有以下三个作用：

（1）注释 2 处调用了 binderopen 函数，用于打开 binder 设备文件，并申请 128KB 大小的内存空间。

（2）注释 3 处调用了 binder_become_context_manager 函数，将 servicemanager 注册成为 Binder 机制的上下文管理者。

（3）注释 4 处调用了 binder_loop 函数，循环等待和处理客户端发来的请求。

现在对这三个作用分别进行讲解。

5.4.1 打开 binder 设备

binder_open 函数用于打开 binder 设备文件，并将它映射到进程的地址空间，代码如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
struct binder_state *binder_open(const char* driver, size_t mapsize)
{
    struct binder_state *bs;
    struct binder_version vers;

    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return NULL;
```

```
    }

    bs->fd = open(driver, O_RDWR | O_CLOEXEC);//1
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open %s (%s)\n",
                driver, strerror(errno));
        goto fail_open;
    }
    //获取Binder的version
    if ((ioctl(bs->fd, BINDER_VERSION, &vers) == -1) ||
        (vers.protocol_version != BINDER_CURRENT_PROTOCOL_VERSION)) {//2
        fprintf(stderr,
                "binder: kernel driver version (%d) differs from user space version
(%d)\n",
                vers.protocol_version, BINDER_CURRENT_PROTOCOL_VERSION);
        goto fail_open;
    }

    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);//3
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
        goto fail_map;
    }
    return bs;

fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return NULL;
}
```

注释 1 处打开了 binder 设备文件。注释 2 处的 ioctl 函数用于获取 Binder 的版本，如果获取不到或者内核空间和用户空间的 binder 不是同一个版本就会直接 goto 到 fail_open 标签，释放 binder 的内存空间。注释 3 处调用 mmap 函数进行内存映射，通俗来讲就是将 binder 设备文件映射到进程的地址空间，地址空间的大小为 mapsize，也就是 128KB。映射完成后会将地址空间的起始地址和大小保存在 binderstate 结构体的 mapped 变量和 mapsize 变量中。

这里着重讲一下 open 函数，它会调用 Kernel Binder 部分的 binder_open 函数，这部分源码位于内核源码中，这里展示的代码版本为 goldfish 3.4。

Intel 的 X86 架构的 CPU 提供了 0～3 共四个特权级，特权级的数字越小，权限越高，Linux 操作系统中主要采用了 0 和 3 两个特权级，分别对应的就是内核态与用户态。用户态的特权级别低，因此进程在用户态下不经过系统调用是无法主动访问到内核空间中的数据的，这样用户无法随意进入所有进程共享的内核空间，起到了保护的作用。接下来介绍用户态和内核态。

若一个进程在执行用户自己的代码时处于用户态，比如 open 函数，它运行在用户空间，当前的进程处于用户态。当一个进程因为系统调用进入内核代码中执行时就处于内核态，比如 open 函数通过系统调用（_open 函数），查找到了 open 函数在 Kernel Binder 对应的函数为 binder_open，这时 binder_open 函数运行在内核空间，当前的进程由用户态切换到内核态，代码如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
static int binder_open(struct inode *nodp, struct file *filp)
{//代表 Binder 进程
struct binder_proc *proc;//1
binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
        current->group_leader->pid, current->pid);
//分配内存空间
proc = kzalloc(sizeof(*proc), GFP_KERNEL);//2
if (proc == NULL)
    return -ENOMEM;
get_task_struct(current);
proc->tsk = current;
INIT_LIST_HEAD(&proc->todo);
init_waitqueue_head(&proc->wait);
proc->default_priority = task_nice(current);
//binder 同步锁
binder_lock(__func__);

binder_stats_created(BINDER_STAT_PROC);
hlist_add_head(&proc->proc_node, &binder_procs);
proc->pid = current->group_leader->pid;
INIT_LIST_HEAD(&proc->delivered_death);
filp->private_data = proc;//3
//binder 同步锁释放
binder_unlock(__func__);
...
return 0;
}
```

注释 1 处的 binder_proc 结构体代表 binder 进程，用于管理 binder 的各种信息。注释 2 处用于为 binder_proc 分配内存空间。注释 3 处将 binder_proc 赋值给 file 指针的 private_data 变量，后面的 5.4.2 节会再次提到这个 private_data 变量。

5.4.2　注册成为 Binder 机制的上下文管理者

binder_become_context_manager 函数用于将 servicemanager 注册成为 Binder 机制的上下文管理者，这个管理者在整个系统中只有一个，代码如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

ioctl 函数会调用 Binder 驱动的 binder_ioctl 函数，binder_ioctl 函数的代码比较多，这里截取 BINDER_SET_CONTEXT_MGR 的处理部分，代码如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
int ret;
struct binder_proc *proc = filp->private_data; //1
struct binder_thread *thread;
unsigned int size = _IOC_SIZE(cmd);
void __user *ubuf = (void __user *)arg;
trace_binder_ioctl(cmd, arg);

ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error
< 2);
if (ret)
    goto err_unlocked;

binder_lock(__func__);
thread = binder_get_thread(proc);//2
if (thread == NULL) {
    ret = -ENOMEM;
    goto err;
}

switch (cmd) {
    ...
```

```
case BINDER_SET_CONTEXT_MGR:
    if (binder_context_mgr_node != NULL) {//3
        printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
        ret = -EBUSY;
        goto err;
    }
    ret = security_binder_set_context_mgr(proc->tsk);
    if (ret < 0)
        goto err;
    if (binder_context_mgr_uid != -1) {//4
        if (binder_context_mgr_uid != current->cred->euid) {//5
            printk(KERN_ERR "binder: BINDER_SET_"
                  "CONTEXT_MGR bad uid %d != %d\n",
                  current->cred->euid,
                  binder_context_mgr_uid);
            ret = -EPERM;
            goto err;
        }
    } else
        binder_context_mgr_uid = current->cred->euid;//6
    binder_context_mgr_node = binder_new_node(proc, NULL, NULL);//7
    if (binder_context_mgr_node == NULL) {
        ret = -ENOMEM;
        goto err;
    }
    binder_context_mgr_node->local_weak_refs++;
    binder_context_mgr_node->local_strong_refs++;
    binder_context_mgr_node->has_strong_ref = 1;
    binder_context_mgr_node->has_weak_ref = 1;
    break;
 ...
err_unlocked:
trace_binder_ioctl_done(ret);
return ret;
}
```

注释 1 处将 file 指针中的 private_data 变量赋值给 binder_proc，这个 private_data 变量在 binder_open 函数中讲过，是一个 binder_proc 结构体。注释 2 处的 binder_get_thread 函数用于获取 binder_thread，binder_thread 结构体指的是 binder 线程，binder_get_thread 函数内部会从传入的参数 binder_proc 中查找 binder_thread，如果查询到则直接返回，如果查询不到则会创建一个新的 binderthread 并返回。

注释 3 处的全局变量 binder_context_mgr_node 代表 Binder 机制的上下文管理者对应的

一个 Binder 对象，如果它不为 NULL，说明此前自身已经被注册为 Binder 的上下文管理者了，Binder 的上下文管理者是不能重复注册的，因此会 goto 到 err 标签。

注释 4 处的全局变量 binder_context_mgr_uid 代表注册了 Binder 机制上下文管理者的进程的有效用户 ID，如果它的值不为-1，说明此前已经有进程注册 Binder 的上下文管理者了，因此在注释 5 处判断当前进程的有效用户 ID 是否等于 binder_context_mgr_uid，如果不等于 binder_context_mgr_uid 就 goto 到 err 标签。

如果不满足注释 4 处的条件，说明此前没有进程注册 Binder 机制的上下文管理者，就会在注释 6 处将当前进程的有效用户 ID 赋值给全局变量 binder_context_mgr_uid，另外还会在注释 7 处调用 binder_new_node 函数创建一个 Binder 对象并赋值给全局变量 binder_context_mgr_node。

5.4.3　循环等待和处理客户端发来的请求

servicemanager 成功注册成为 Binder 机制的上下文管理者后，servicemanager 就是 Binder 机制的“总管”了，它需要在系统运行期间处理客户端的请求，由于客户端的请求不确定何时发送，因此需要通过无限循环来实现，实现这一需求的函数就是 binder_loop。

frameworks/native/cmds/servicemanager/binder.c

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    uint32_t readbuf[32];

    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;

    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(uint32_t));//1

    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (uintptr_t) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);//2
```

```
        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }

        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);//3
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

注释 1 处将 BC_ENTER_LOOPER 指令通过 binder_write 函数写入 Binder 驱动中，这样当前线程（ServiceManager 的主线程）就成为一个 Binder 线程，这样就可以处理进程间的请求了。

在无限循环中不断调用注释 2 处的 ioctl 函数，ioctl 函数使用 BINDER_WRITE_READ 指令查询 Binder 驱动中是否有新的请求，如果有新的请求就交给注释 3 处的 binder_parse 函数处理。如果没有新的请求，当前线程就会在 Binder 驱动中睡眠，等待新的进程间请求。

由于 binder_write 函数的调用链中涉及了内核空间和用户空间的交互，因此这里着重讲解这部分内容，代码如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
int binder_write(struct binder_state *bs, void *data, size_t len)
{
    struct binder_write_read bwr;//1
    int res;

    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (uintptr_t) data;//2
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);//3
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
```

```
            strerror(errno));
    }
    return res;
}
```

注释 1 处定义了 binder_write_read 结构体，接下来的代码对 bwr 进行赋值，其中需要注意的是，注释 2 处 data 的值为 BC_ENTER_LOOPER。注释 3 处的 ioctl 函数会将 bwr 中的数据发送给 binder 驱动，我们已经知道了 ioctl 函数在 Kernel Binder 中对应的函数为 binder_ioctl，此前分析过这个函数，这里截取 BINDER_WRITE_READ 命令处理部分，代码如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    ...
    void __user *ubuf = (void __user *)arg;
    ...
switch (cmd) {
case BINDER_WRITE_READ: {
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
        ret = -EINVAL;
        goto err;
    }
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {//1
        ret = -EFAULT;
        goto err;
    }
    binder_debug(BINDER_DEBUG_READ_WRITE,
            "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
            proc->pid, thread->pid, bwr.write_size, bwr.write_buffer,
            bwr.read_size, bwr.read_buffer);

    if (bwr.write_size > 0) {//2
        ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer,
bwr.write_size, &bwr.write_consumed);//3
        trace_binder_write_done(ret);
        if (ret < 0) {
            bwr.read_consumed = 0;
            if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                ret = -EFAULT;
            goto err;
        }
```

```
        }
       ...
        binder_debug(BINDER_DEBUG_READ_WRITE,
                "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
                proc->pid, thread->pid, bwr.write_consumed, bwr.write_size,
                bwr.read_consumed, bwr.read_size);
        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {//4
            ret = -EFAULT;
            goto err;
        }
        break;
    }
      ...
    return ret;
    }
```

注释 1 处的 copy_from_user 函数用于将用户空间数据 ubuf 复制出来保存到内核数据 bwr（binder_write_read 结构体）中。

注释 2 处的 bwr 的输入缓存区有数据时，会调用注释 3 处的 binder_thread_write 函数来处理 BC_ENTER_LOOPER 协议，其内部会将目标线程的状态设置为 BINDER_LOOPER_STATE_ENTERED，这样目标线程就是一个 Binder 线程。

注释 4 处通过 copyto_user 函数将内核空间数据 bwr 复制到用户空间。

5.5 系统服务的获取过程

本节仍以 MediaPlayerService 为例，来讲解系统服务的获取过程。本节内容分为两个部分，分别是客户端 MediaPlayerService 请求获取服务和服务端 ServiceManager 处理请求，下面先来学习第一部分。

5.5.1 客户端 MediaPlayerService 请求获取服务

想要获取 MediaPlayerService，需要先调用 getMediaPlayerService 函数，代码如下所示。

frameworks/av/media/libmedia/IMediaDeathNotifier.cpp

```
IMediaDeathNotifier::getMediaPlayerService()
{
   ALOGV("getMediaPlayerService");
```

```
    Mutex::Autolock _l(sServiceLock);
    if (sMediaPlayerService == 0) {
        sp<IServiceManager> sm = defaultServiceManager();//1
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.player"));//2
            if (binder != 0) {//3
                break;
            }
            ALOGW("Media player service not published, waiting...");
            usleep(500000); //4
        } while (true);

        if (sDeathNotifier == NULL) {
            sDeathNotifier = new DeathNotifier();
        }
        binder->linkToDeath(sDeathNotifier);
        sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);//5
    }
    ALOGE_IF(sMediaPlayerService == 0, "no media player service!?");
    return sMediaPlayerService;
}
```

注释 1 处的 defaultServiceManager 返回的是 BpServiceManager，注释 2 处获取名为"media.player"的系统服务，其返回的值为 BpBinder。由于这时 MediaPlayerService 可能还没有向 ServiceManager 注册，那么就不能满足注释 3 的条件，在注释 4 处休眠 0.5s 后继续调用 getService 函数，直到获取对应的服务为止。

注释 5 处的 interface_cast 函数用于将 BpBinder 转换成 BpMediaPlayerService，其原理就是通过 BpBinder 的 handle 来找到对应的服务，即 BpMediaPlayerService。

注释 2 处的获取服务是重点，BpServiceManager 的 getService 函数如下所示。

frameworks/native/libs/binder/IServiceManager.cpp::BpServiceManager

```
virtual sp<IBinder> getService(const String16& name) const
    {
        ...
        int n = 0;
        while (uptimeMillis() < timeout) {
            n++;
            if (isVendorService) {
                ALOGI("Waiting for vendor service %s...", String8(name).string());
                CallStack stack(LOG_TAG);
```

```
        } else if (n%10 == 0) {
            ALOGI("Waiting for service %s...", String8(name).string());
        }
        usleep(1000*sleepTime);

        sp<IBinder> svc = checkService(name);//1
        if (svc != NULL) return svc;
    }
    ALOGW("Service %s didn't start. Returning NULL", String8(name).string());
    return NULL;
}
```

getService 函数的主要作用就是循环查询服务是否存在，如果不存在就继续查询，查询服务用到了注释 1 处的 checkService 函数，代码如下所示。

frameworks/native/libs/binder/IServiceManager.cpp::BpServiceManager

```
virtual sp<IBinder> checkService( const String16& name) const
{
    Parcel data, reply;//1
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);//2
    remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);//3
    return reply.readStrongBinder();
}
```

注释 1 处的 data 是一个数据包，后续也会将数据写入 data 中。注释 2 处将字符串"media.player"写入 data 中。注释 3 处的 remote()指的是 mRemote，也就是 BpBinder，BpBinder 的 transact 函数如下所示。

frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder 将逻辑处理交给 IPCThreadState，后面的调用链在 5.3 节讲过，

IPCThreadState::self()会创建 IPCThreadState，IPCThreadState 的 transact 函数如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err;

    flags |= TF_ACCEPT_FDS;
    ...
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);//1

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        ...
        if (reply) {
            err = waitForResponse(reply);//2
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        ...
    } else {
        //不需要等待 reply 的分支
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

调用 BpBinder 的 transact 函数实际上就是调用 IPCThreadState 的 transact 函数。注释 1 处的 writeTransactionData 函数用于传输数据，其中第一个参数 BC_TRANSACTION 代表向 Binder 驱动发送命令协议。

注释 1 处的 writeTransactionData 用于准备发送的数据，其内部会将 BC_TRANSACTION 和 bindertransactiondata 结构体写入 mOut 中。

接着查看 waitForResponse 函数，代码如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;//1
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
        cmd = (uint32_t)mIn.readInt32();
        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;
       ...
        default:
            //处理各种命令协议
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
}
finish:
    ...
    return err;
}
```

注释 1 处的 talkWithDriver 函数的内部通过 ioctl 函数与 Binder 驱动进行通信，代码如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }
    //和 Binder 驱动通信的结构体
    binder_write_read bwr; //1
    //mIn 是否有可读的数据，接收的数据存储在 mIn
```

```
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();//2
    //这时 doReceive 的值为 true
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();//3
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
   ...
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(__ANDROID__)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)//4
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
     ...
    } while (err == -EINTR);
    ...
    return err;
}
```

注释 1 处的 binder_writer_ead 是和 Binder 驱动通信的结构体，在注释 2 处和注释 3 处将 mOut、mIn 赋值给 binder_write_read 的相应字段，最终通过注释 4 处的 ioctl 函数和 Binder 驱动进行通信。MediaPlayerService 请求获取服务调用的时序图如图 5-12 所示。

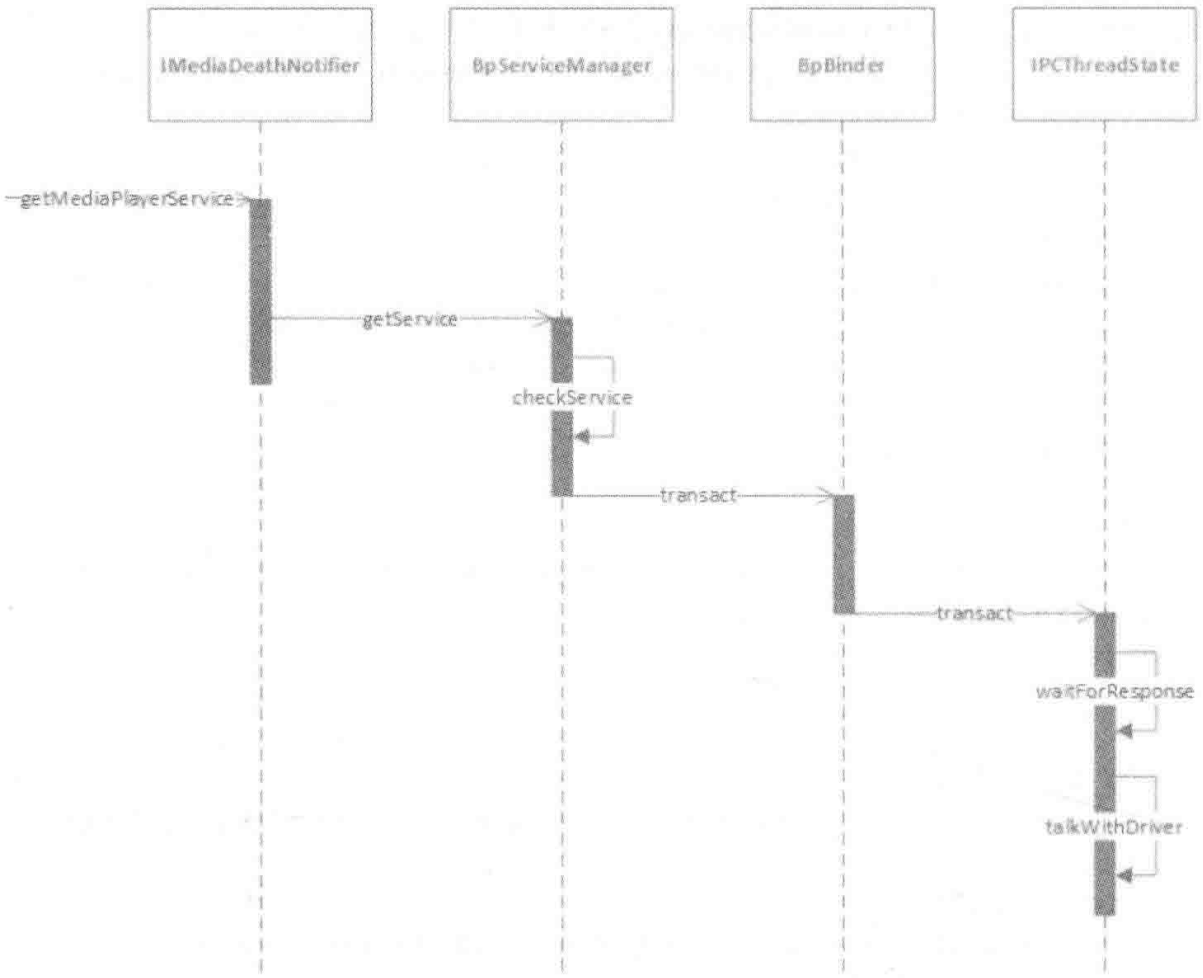


图 5-12　MediaPlayerService 请求获取服务调用的时序图

这时我们需要再次查看图 5-10，从这张简化的流程图中可以看出，当前分析的是客户端进程的流程，当 MediaPlayerService 向 Binder 驱动发送 BC_TRANSACTION 命令后，Binder 驱动会向 ServiceManager 发送 BR_TRANSACTION 命令，接下来我们来查看服务端 ServiceManager 是如何处理获取服务请求的。

5.5.2　服务端 ServiceManager 处理请求

提到服务端 ServiceManager 处理请求，就要想到 ServiceManager 的启动过程，这里简单回顾 servicemanager 的入口函数，如下所示。

frameworks/native/cmds/servicemanager/servicemanager.c

```
int main(int argc, char** argv)
{
   ...
    bs = binder_open(driver, 128*1024);
    ...
```

```
    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    ...
    if (getcon(&service_manager_context) != 0) {
        ALOGE("SELinux: Failed to acquire service_manager context. Aborting.\n");
        abort();
    }
    binder_loop(bs, svcmgr_handler);//1
    return 0;
}
```

main 函数主要有三个作用，其中最后一个作用就是调用 binder_loop 函数，这里需要注意，它的第二个参数为 svcmgr_handler，后面会再次提到 svcmgr_handler。

binder_loop 函数如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
...
    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (uintptr_t) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

在无限循环中不断调用 ioctl 函数，其内部不断使用 BINDER_WRITE_READ 指令查询 Binder 驱动中是否有新的请求，如果有就交给 binder_parse 函数处理；如果没有，当前线程就会在 Binder 驱动中睡眠，等待新的进程间通信请求。

binder_parse 函数如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
             uintptr_t ptr, size_t size, binder_handler func)
{
   int r = 1;
   uintptr_t end = ptr + (uintptr_t) size;

   while (ptr < end) {
      uint32_t cmd = *(uint32_t *) ptr;
      ptr += sizeof(uint32_t);
#if TRACE
      fprintf(stderr,"%s:\n", cmd_name(cmd));
#endif
      switch(cmd) {
      ...
      case BR_TRANSACTION: {
         struct binder_transaction_data *txn = (struct binder_transaction_data *)
ptr;
         if ((end - ptr) < sizeof(*txn)) {
            ALOGE("parse: txn too small!\n");
            return -1;
         }
         binder_dump_txn(txn);
         if (func) {
            unsigned rdata[256/4];
            struct binder_io msg;
            struct binder_io reply;
            int res;
            bio_init(&reply, rdata, sizeof(rdata), 4);
            bio_init_from_txn(&msg, txn);
            res = func(bs, txn, &msg, &reply);//1
            if (txn->flags & TF_ONE_WAY) {
               binder_free_buffer(bs, txn->data.ptr.buffer);
            } else {
               binder_send_reply(bs, &reply, txn->data.ptr.buffer, res);
            }
         }
         ptr += sizeof(*txn);
```

```
            break;
        }
        ...
    }

    return r;
}
```

这里截取了 BR_TRANSACTION 命令的处理部分，注释 1 处的 func 函数通过一路传递指向的是 svcmgr_handler 函数，svcmgr_handler 函数如下所示。

frameworks/native/cmds/servicemanager/servicemanager.c

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_transaction_data *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    ...
    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        if (s == NULL) {
            return -1;
        }
        handle = do_find_service(s, len, txn->sender_euid, txn->sender_pid);
        if (!handle)
            break;
        bio_put_ref(reply, handle);
        return 0;

    ...
    default:
        ALOGE("unknown code %d\n", txn->code);
        return -1;
    }

    bio_put_uint32(reply, 0);
    return 0;
}
```

当要获取服务时，会调用 do_find_service 函数，代码如下所示。

frameworks/native/cmds/servicemanager/servicemanager.c

```
uint32_t do_find_service(const uint16_t *s, size_t len, uid_t uid, pid_t spid)
{
    struct svcinfo *si = find_svc(s, len);//1

    if (!si || !si->handle) {
        return 0;
    }

    if (!si->allow_isolated) {
        uid_t appid = uid % AID_USER;
        if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END) {
            return 0;
        }
    }
    if (!svc_can_find(s, len, spid, uid)) {
        return 0;
    }

    return si->handle;
}
```

注释 1 处的 find_svc 函数用于查询服务，返回的 svcinfo 是一个结构体，其内部包含了服务的 handle 值，最终会返回服务的 handle 值。接着来看 find_svc 函数：

frameworks/native/cmds/servicemanager/servicemanager.c

```
struct svcinfo *find_svc(const uint16_t *s16, size_t len)
{
    struct svcinfo *si;

    for (si = svclist; si; si = si->next) {
        if ((len == si->len) &&
            !memcmp(s16, si->name, len * sizeof(uint16_t))) {
            return si;
        }
    }
    return NULL;
}
```

系统服务的注册流程中，在 Kernel Binder 中会调用 do_add_service 函数，其内部会将包含服务名和 handle 值的 svcinfo 保存到 svclist 列表中。同样，获取服务的流程中，find_svc 函数会遍历 svclist 列表，根据服务名查找对应服务是否已经注册，如果已经注册就会返回对应的 svcinfo，如果没有注册就返回 NULL。

Native Binder 的架构如图 5-13 所示。

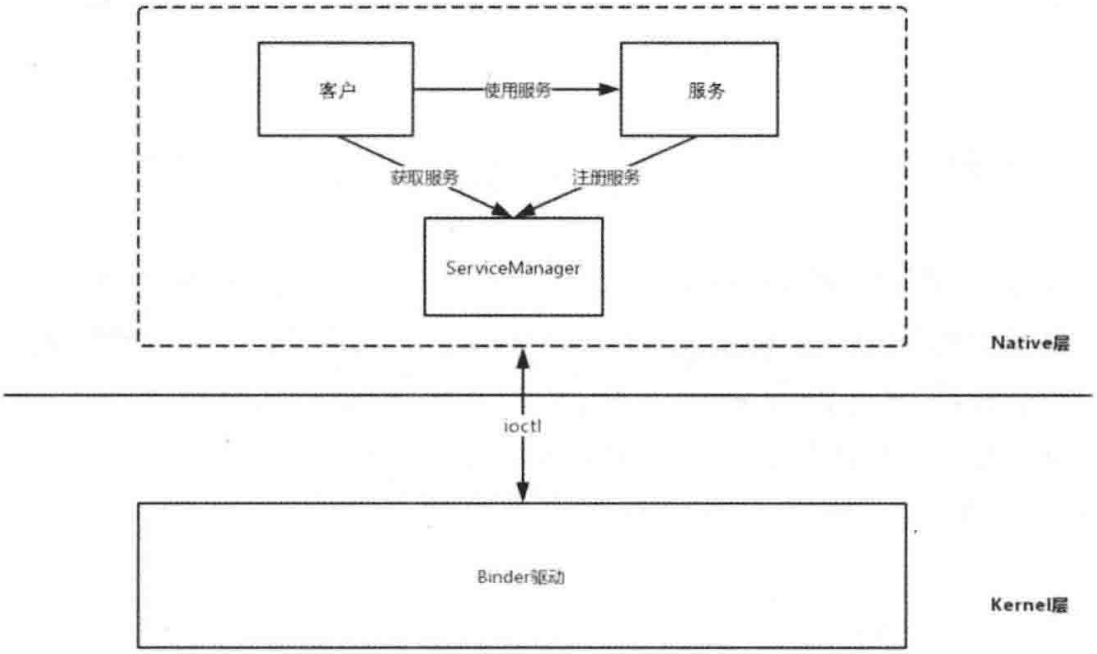


图 5-13　Native Binder 的架构

BpBinder 是客户端与服务端交互的代理类，而 BBinder 则代表了服务端，那么图 5-13 就可以改为图 5-14 的形式。

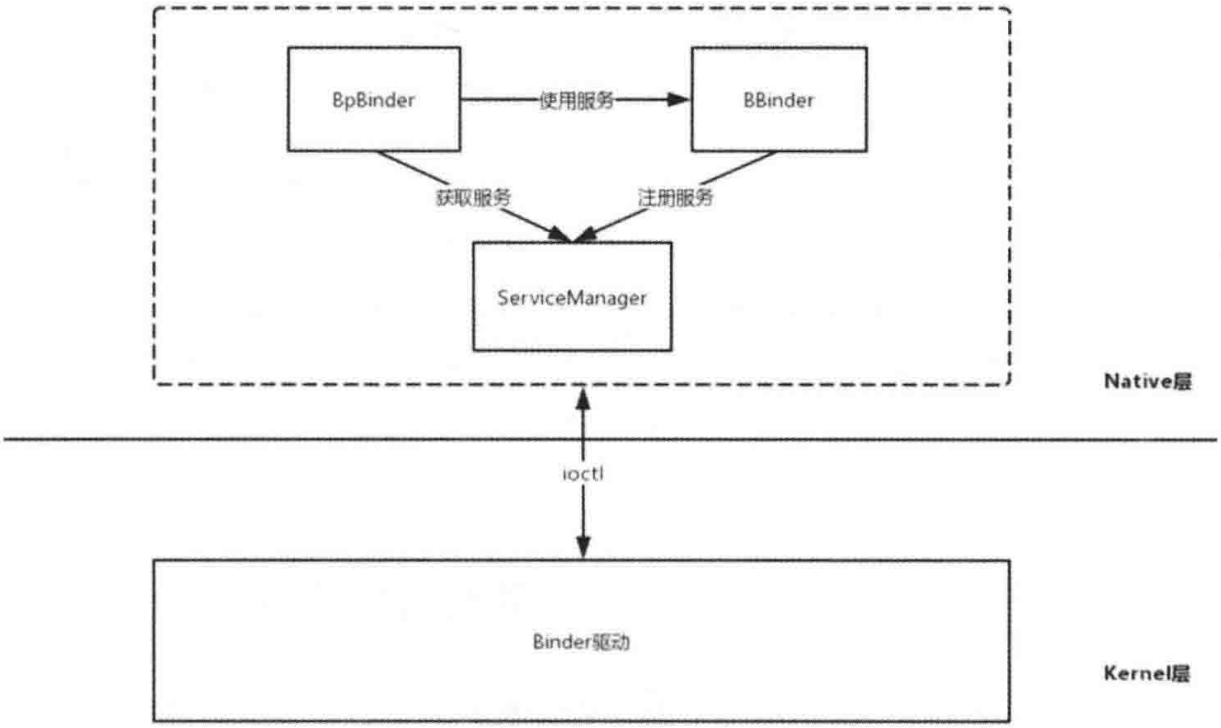


图 5-14　修改后的 Native Binder 的架构

从图 5-14 可以看到，Native Binder 实际是基于 C/S 架构的，BpBinder 代表了客户端，BBinder 代表了服务端，Native Binder 通过 ioctl 函数和 Binder 驱动进行数据交互。

5.6 本章小结

Native Binder 的内容实际有很多，本章介绍了应用开发需要掌握的知识点，比如学习 Binder 原理的前置的知识点，ServiceManager 中的 Binder 机制、ServiceManager 的启动过程、系统服务的注册过程和获取过程。可以说 Native Binder 的核心都是围绕着 ServiceManager 的，只要理解了 ServiceManager 的相关内容，就已经掌握了 Native Binder 的关键部分。

第 6 章

Java Binder 原理

关联章节：

第 5 章 Native Binder 原理

《Android 进阶解密》 第 2 章 Android 系统启动

《Android 进阶解密》 第 6 章 理解 ActivityManagerService

《Android 进阶解密》 第 9 章 JNI 原理

Java Binder 是需要借助 Native Binder 来进行工作的，因此 Java Binder 在设计上也是一个 C/S 架构，可以说 Java Binder 是 Native Binder 的一个镜像，所以在学习 Java Binder 前，首先要学习第 5 章讲解的 Native Binder 的内容。本章主要介绍 Java Binder 是如何初始化的（Java Binder 的 JNI 注册）和系统服务的注册过程。

6.1 Java Binder 的 JNI 注册

Java Binder 和 Native Binder 进行通信前，需要通 JNI，JNI 的注册是在 Zygote 进程启动过程中进行的，代码如下所示。

frameworks/base/core/jni/AndroidRuntime.cpp

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options,
bool zygote)
{
    ...
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env, zygote) != 0) {//1
        return;
    }
    onVmCreated(env);
    if (startReg(env) < 0) {//2
        ALOGE("Unable to register all android natives\n");
        return;
    }
    ...
}
```

注释 1 处用于启动 Java 虚拟机，注释 2 处 startReg 函数用于完成虚拟机的 JNI 注册，关于 AndroidRuntime 的 start 函数的具体分析可以见《Android 进阶解密》2.2 节。

startReg 函数如下所示。

frameworks/base/core/jni/AndroidRuntime.cpp

```
/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    ATRACE_NAME("RegisterAndroidNatives");
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

    ALOGV("--- registering native functions ---\n");
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {//1
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);
    return 0;
}
```

注释 1 处的 register_jni_procs 函数的作用就是循环调用 gRegJNI 数组的成员所对应的方法，如下所示。

```
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
{
   for (size_t i = 0; i < count; i++) {
      if (array[i].mProc(env) < 0) {
#ifndef NDEBUG
         ALOGD("----------!!! %s failed to load\n", array[i].mName);
#endif
         return -1;
      }
   }
   return 0;
}
```

gRegJNI 数组中有 100 多个成员变量：

```
static const RegJNIRec gRegJNI[] = {
   REG_JNI(register_com_android_internal_os_RuntimeInit),
   REG_JNI(register_com_android_internal_os_ZygoteInit_nativeZygoteInit),
   REG_JNI(register_android_os_SystemClock),
   ...
   REG_JNI(register_android_os_Binder),//1
  ...
};
```

其中 REG_JNI 是一个宏定义：

```
   #define REG_JNI(name)      { name }
   struct RegJNIRec {
      int (*mProc)(JNIEnv*);
   };
```

实际上就是调用参数名所对应的函数。负责 Java Binder 和 Native Binder 通信的函数为注释 1 处的 register_android_os_Binder，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
int register_android_os_Binder(JNIEnv* env)
{
   //注册 Binder 类
   if (int_register_android_os_Binder(env) < 0)
      return -1;
   //注册 BinderInternal 类
   if (int_register_android_os_BinderInternal(env) < 0)
      return -1;
   //注册 BinderProxy 类
   if (int_register_android_os_BinderProxy(env) < 0)
```

```
        return -1;
    ...
    return 0;
}
```

register_android_os_Binder 函数有三个作用：注册 Binder 类、注册 BinderInternal 类、注册 BinderProxy 类。它们是 Java Binder 关联类的一小部分，它们的关系如图 6-1 所示。

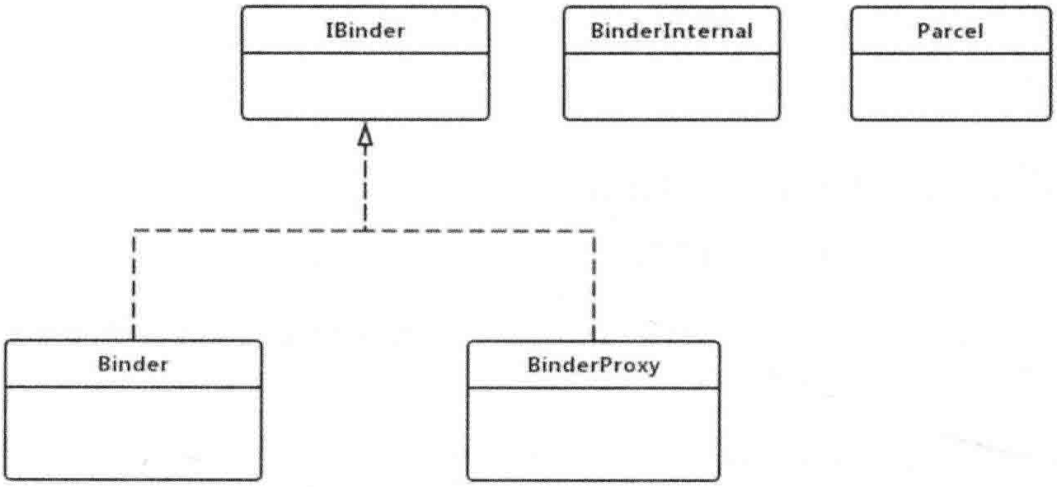


图 6-1　Java Binder 关联类

IBinder 接口中定义了很多整型变量，其中定义一个叫作 FLAG_ONEWAY 的整形变量。客户端在发起调用时一般会阻塞，直到服务端返回结果。设置 FLAG_ONEWAY 后，客户端只需要把请求发送到服务端就可以立即返回，而不需要等待服务端的结果，这是一种非阻塞方式。

- Binder 和 BinderProxy 实现了 IBinder 接口，Binder 是服务端的代表，而 BinderProxy 是客户端的代表。
- BinderInternal 只在 Binder 框架中被使用，其内部类 GcWatcher 用于处理 Binder 的垃圾回收。
- Parcel 是一个数据包装器，它可以在进程间进行传递，Parcel 既可以传递基本数据类型又可以传递 Binder 对象，Binder 通信就是通过 Parcel 来实现客户端与服务端数据交互的。Parcel 的实现既有 Java 部分，又有 Native 部分，其具体实现在 Native 部分中。

下面分别对 Binder、BinderInternal 这两个类的注册进行分析。

6.1.1　Binder 类的注册

调用 int_register_android_os_Binder 函数来完成 Binder 类的注册，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
static const JNINativeMethod gBinderMethods[] = {
    { "getCallingPid", "()I", (void*)android_os_Binder_getCallingPid },
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid },
    { "clearCallingIdentity", "()J",
(void*)android_os_Binder_clearCallingIdentity },
    { "restoreCallingIdentity", "(J)V",
(void*)android_os_Binder_restoreCallingIdentity },
    { "setThreadStrictModePolicy", "(I)V",
(void*)android_os_Binder_setThreadStrictModePolicy },
    { "getThreadStrictModePolicy", "()I",
(void*)android_os_Binder_getThreadStrictModePolicy },
    { "flushPendingCommands", "()V",
(void*)android_os_Binder_flushPendingCommands },
    { "getNativeBBinderHolder", "()J",
(void*)android_os_Binder_getNativeBBinderHolder },
    { "getNativeFinalizer", "()J", (void*)android_os_Binder_getNativeFinalizer },
    { "blockUntilThreadAvailable", "()V",
(void*)android_os_Binder_blockUntilThreadAvailable }
};
const char* const kBinderPathName = "android/os/Binder";//1
static int int_register_android_os_Binder(JNIEnv* env)
{
    jclass clazz = FindClassOrDie(env, kBinderPathName);//2

    gBinderOffsets.mClass = MakeGlobalRefOrDie(env, clazz);//3
    gBinderOffsets.mExecTransact = GetMethodIDOrDie(env, clazz, "execTransact",
"(IJJI)Z");//4
    gBinderOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObject", "J");//5

    return RegisterMethodsOrDie(
        env, kBinderPathName,
        gBinderMethods, NELEM(gBinderMethods));
}
```

注释 1 处的 kBinderPathName 的值为"android/os/Binder"，这是 Binder 在 Java Binder 中的全路径名。

注释 2 处根据这个路径名获取 Binder 的 Class 对象，并赋值给 jclass 类型的变量 clazz，clazz 是 Java 层的 Binder 在 JNI 层的代表。

注释 3 处通过 MakeGlobalRefOrDie 函数将本地引用 clazz 转变为全局引用并赋值给 gBinderOffsets.mClass。

注释 4 处找到了 Java 层的 Binder 的成员方法 execTransact 并赋值给 gBinderOffsets.mExecTransact。

注释 5 处找到了 Java 层的 Binder 的成员变量 mObject 并赋值给 gBinderOffsets.mObject。

最后一行代码通过 RegisterMethodsOrDie 函数注册 gBinderMethods 中定义的函数，其中 gBinderMethods 是 JNINativeMethod 类型的数组，其中存储的是 Binder 的 Native 方法（Java 层）与 JNI 层函数的对应关系。

gBinderMethods 的定义如下所示。

```
static struct bindernative_offsets_t
{
    jclass mClass;
    jmethodID mExecTransact;
    jfieldID mObject;

} gBinderOffsets;
```

使用 gBinderMethods 来保存成员变量和方法有以下两个原因。

（1）为了提高效率。如果每次调用相关的方法时都需要查询方法和变量，那么显然效率比较低。

（2）这些成员变量和方法都是本地引用，在 int_register_android_os_Binder 函数返回时，这些本地引用会被自动释放，因此用 gBinderOffsets 来保存，以便后续使用。

6.1.2 BinderInternal 类的注册

调用 int_register_android_os_BinderInternal 函数来完成 BinderInternal 类的注册，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
const char* const kBinderInternalPathName =
"com/android/internal/os/BinderInternal";
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
    jclass clazz = FindClassOrDie(env, kBinderInternalPathName);
```

```
    gBinderInternalOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderInternalOffsets.mForceGc = GetStaticMethodIDOrDie(env, clazz,
"forceBinderGc", "()V");
    gBinderInternalOffsets.mProxyLimitCallback = GetStaticMethodIDOrDie(env,
clazz, "binderProxyLimitCallbackFromNative", "(I)V");

    jclass SparseIntArrayClass = FindClassOrDie(env,
"android/util/SparseIntArray");
    gSparseIntArrayOffsets.classObject = MakeGlobalRefOrDie(env,
SparseIntArrayClass);
    gSparseIntArrayOffsets.constructor = GetMethodIDOrDie(env,
gSparseIntArrayOffsets.classObject,
                                                    "<init>", "()V");
    gSparseIntArrayOffsets.put = GetMethodIDOrDie(env,
gSparseIntArrayOffsets.classObject, "put",
                                            "(II)V");

    BpBinder::setLimitCallback(android_os_BinderInternal_proxyLimitcallback);

    return RegisterMethodsOrDie(
        env, kBinderInternalPathName,
        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

和 int_register_android_os_Binder 函数的实现类似，BinderInternal 类主要有以下三个作用。

（1）获取 BinderInternal 在 JNI 层的代表 clazz。

（2）将 BinderInternal 类中有用的成员变量和方法存储到 gBinderInternalOffsets 中。

（3）注册 BinderInternal 类的 Native 方法对应的 JNI 函数。

还有一个 BinderProxy 类的注册，它和 Binder、BinderInternal 的注册过程类似，这里就不再赘述了，有兴趣的读者可以自行去看源码。

6.2　Java Binder 中系统服务的注册过程

第 5 章介绍的是 Native Binder 中的系统服务的注册过程，这一过程的核心是 ServiceManager，在 Java Binder 中，也有一个 ServiceManager，然而这个 ServiceManager 是 Java 文件。

既然要将系统服务注册到 ServiceManager，那么就需要选择一个系统服务，这里以常

见的 AMS 为例。

6.2.1 将 AMS 注册到 ServiceManager

在 AMS 的 setSystemProcess 方法中，会调用 ServiceManager 的 addService 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
    public void setSystemProcess() {
        try {
            ServiceManager.addService(Context.ACTIVITY_SERVICE, this, /*
allowIsolated= */ true,
                    DUMP_FLAG_PRIORITY_CRITICAL | DUMP_FLAG_PRIORITY_NORMAL |
DUMP_FLAG_PROTO);//1
            ....
        } catch (PackageManager.NameNotFoundException e) {
            throw new RuntimeException(
                    "Unable to find android system package", e);
        }
    ...
    }
```

注释 1 处的 Context.ACTIVITY_SERVICE 的值为"activity"，其作用就是将 AMS 注册到 ServiceManager 中。接着来看 ServiceManager 的 addService 方法，如下所示。

frameworks/base/core/java/android/os/ServiceManager.java

```
    public static void addService(String name, IBinder service, boolean
allowIsolated,
            int dumpPriority) {
        try {
            getIServiceManager().addService(name, service, allowIsolated,
dumpPriority);
        } catch (RemoteException e) {
            Log.e(TAG, "error in addService", e);
        }
    }
```

主要分析 getIServiceManager 方法返回的是什么，代码如下所示。

frameworks/base/core/java/android/os/ServiceManager.java

```
    private static IServiceManager getIServiceManager() {
        if (sServiceManager != null) {
```

```
            return sServiceManager;
        }
        sServiceManager = ServiceManagerNative
                .asInterface(Binder.allowBlocking(BinderInternal.getContextObject(
)));
        return sServiceManager;
    }
```

到这里，有以下几个函数需要分析。

- BinderInternal.getContextObject
- ServiceManagerNative.asInterface
- getIServiceManager().addService

1. BinderInternal.getContextObject

Binder.allowBlocking 的作用是将 BinderProxy 的 sWarnOnBlocking 的值设置为 false，其主要来分析 BinderInternal.getContextObject 的作用，这是一个 Native 方法，找到它对应的函数。

frameworks/base/core/jni/androidutil/Binder.cpp

```
static const JNINativeMethod gBinderInternalMethods[] = {
    { "getContextObject", "()Landroid/os/IBinder;",
(void*)android_os_BinderInternal_getContextObject },
    ...
};
```

其对应的函数为 android_os_BinderInternal_getContextObject。

frameworks/base/core/jni/androidutil_Binder.cpp

```
static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject
clazz)
{
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);//1
    return javaObjectForIBinder(env, b);
}
```

ProcessState::self 的作用是创建 ProcessState，注释 1 处最终返回的是 BpBinder。

BpBinder 是 Native Binder 中的客户端，这说明 Java 层的 ServiceManager 需要 Native 层的 BpBinder，但是这个 BpBinder 在 Java 层是无法直接使用的，那么就需要传入 javaObjectForIBinder 函数来进行处理，其内部会创建一个 BinderProxy 对象，这样

BinderInternal.getContextObject 最终得到的是 BinderProxy，BinderProxy 是 Java Binder 的客户端的代表。

需要注意的是，这个传入的 BpBinder 会保存到 BinderProxy 的成员变量 mObject 中，后面章节会再次提到这部分内容。

2. ServiceManagerNative.asInterface

提到 asInterface 方法，在 Native Binder 中也有一个 asInterface 函数，它的作用是以 BpBinder 作为参数创建 BpServiceManager。那么在 Java Binder 中的 asInterface 方法的作用又是什么？

frameworks/base/core/java/android/os/ServiceManagerNative.java

```
static public IServiceManager asInterface(IBinder obj)
 {
    if (obj == null) {
       return null;
    }
    IServiceManager in =
       (IServiceManager)obj.queryLocalInterface(descriptor);
    if (in != null) {
       return in;
    }
    return new ServiceManagerProxy(obj);
 }
```

我们知道 obj 的值为 BinderProxy，那么 asInterface 方法的作用就是以 BinderProxy 作为参数创建 ServiceManagerProxy 的。

BinderProxy 和 BpBinder 分别在 Jave Binder 和 Native Binder 中作为客户端的代表，BpServiceManager 通过 BpBinder 来实现通信，ServiceManagerProxy 也会将业务的请求交给 BinderProxy 来处理。

分析到这里，如下代码：

```
    sServiceManager = ServiceManagerNative
          .asInterface(Binder.allowBlocking(BinderInternal.getContextObject(
)));
```

可以理解为：

```
    sServiceManager = new ServiceManagerProxy(BinderProxy);
 }
```

3. getIServiceManager().addService

getIServiceManager 返回的是 ServiceManagerProxy，ServiceManagerProxy 是 ServiceManagerNative 的内部类，它实现了 IServiceManager 接口。

接着来查看 ServiceManagerProxy 的 addService 方法，代码如下所示。

frameworks/base/core/java/android/os/ServiceManagerNative.java::ServiceManagerProxy

```
    public void addService(String name, IBinder service, boolean allowIsolated, int
dumpPriority)
            throws RemoteException {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        data.writeInterfaceToken(IServiceManager.descriptor);
        data.writeString(name);
        data.writeStrongBinder(service);//1
        data.writeInt(allowIsolated ? 1 : 0);
        data.writeInt(dumpPriority);
        mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);//2
        reply.recycle();
        data.recycle();
    }
```

注释 1 处的 data.writeStrongBinder 很关键，后面章节会进行分析。这里的 Parcel 是一个数据包装器，它能够将请求数据写入 Parcel 类型的对象 data 中，通过注释 2 处的 mRemote.transact 发送出去，mRemote 实际上是 BinderProxy，BinderProxy.transact 是 native 函数，实现的函数如下所示。

frameworks/base/core/jni/android*util*/Binder.cpp

```
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
        jint code, jobject dataObj, jobject replyObj, jint flags)
{
    if (dataObj == NULL) {
        jniThrowNullPointerException(env, NULL);
        return JNI_FALSE;
    }
    Parcel* data = parcelForJavaObject(env, dataObj);//1
    if (data == NULL) {
        return JNI_FALSE;
    }
    Parcel* reply = parcelForJavaObject(env, replyObj);//2
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
```

```
        }
        IBinder* target = getBPNativeData(env, obj)->mObject.get();//3
        if (target == NULL) {
            jniThrowException(env, "java/lang/IllegalStateException", "Binder has been
finalized!");
            return JNI_FALSE;
        }
       ...
        status_t err = target->transact(code, *data, reply, flags);//4
        return JNI_FALSE;
    }
```

注释 1 处和注释 2 处将 Java 层的 Parcel 对象转化成为 Native 层的 Parcel 对象。我们知道 BpBinder 会保存到 BinderProxy 的成员变量 mObject 中，因此在注释 3 处，从 BinderProxy 的成员变量 mObject 中获取 BpBinder。最终会在注释 4 处调用 BpBinder 的 transact 函数，向 Binder 驱动发送数据，可以看出 Java Binder 是需要 Native Binder 支持的，最终的目的就是向 Binder 驱动发送数据和接收数据。

6.2.2 引出 JavaBBinder

继续来分析 6.2.1 节的 data.writeStrongBinder(service)，代码如下所示。

frameworks/base/core/java/android/os/Parcel.java

```
    public final void writeStrongBinder(IBinder ll) {
        nativeWriteStrongBinder(mNativePtr, val);
    }
```

nativeWriteStrongBinder 是 Native 方法，实现的函数为 android_os_Parcel_writeStrongBinder，代码如下所示。

frameworks/base/core/jni/android/os/Parcel.cpp

```
    static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jclass clazz, jlong
nativePtr, jobject object)
    {
        Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
        if (parcel != NULL) {
            const status_t err = parcel->writeStrongBinder(ibinderForJavaObject(env,
object));//1
            if (err != NO_ERROR) {
                signalExceptionForError(env, clazz, err);
            }
        }
```

```
}
```

接着查看注释 1 处 ibinderForJavaObject 函数，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
   if (obj == NULL) return NULL;
   if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {//1
      JavaBBinderHolder* jbh = (JavaBBinderHolder*)
         env->GetLongField(obj, gBinderOffsets.mObject);
      return jbh->get(env, obj);//2
   }
   if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {//3
      return getBPNativeData(env, obj)->mObject;
   }

   ALOGW("ibinderForJavaObject: %p is not a Binder object", obj);
   return NULL;
}
```

在注释 3 处，如果 obj 是 Java 层的 BinderProxy 类，则返回 BpBinder。

在注释 1 处，如果 obj 是 Java 层的 Binder 类，那么先获取 JavaBBinderHolder 对象，然后在注释 2 处调用 JavaBBinderHolder 的 get 函数，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp::JavaBBinderHolder

```
class JavaBBinderHolder
{
public:
   sp<JavaBBinder> get(JNIEnv* env, jobject obj)
   {
      AutoMutex _l(mLock);
      sp<JavaBBinder> b = mBinder.promote();//1
      if (b == NULL) {
         //obj 是一个 Java 层的 Binder 对象
         b = new JavaBBinder(env, obj);//2
         mBinder = b;
         ALOGV("Creating JavaBinder %p (refs %p) for Object %p, weakCount=%" PRId32
"\n",
             b.get(), b->getWeakRefs(), obj, b->getWeakRefs()->getWeakCount());
      }
      return b;
   }
```

```
    sp<JavaBBinder> getExisting()
    {
        AutoMutex _l(mLock);
        return mBinder.promote();
    }
private:
    Mutex         mLock;
    wp<JavaBBinder> mBinder;
};
```

成员变量 mBinder 是 wp<JavaBBinder>类型的弱引用，在注释 1 处得到 sp<JavaBBinder>类型的强引用 b，在注释 2 处创建 JavaBBinder 并赋值给 b。那么 JavaBBinderHolder 的 get 函数返回的是 JavaBBinder。

data.writeStrongBinder(service)在这里等价于：

```
data.writeStrongBinder(new JavaBBinder(env, Binder))。
```

讲到这里我们可以得知 ServiceManager.addService()传入的并不是 AMS 本身，而是 JavaBBinder。

6.2.3 解析 JavaBBinder

查看 JavaBBinder 的构造函数，代码如下所示。

frameworks/base/core/jni/android_util_Binder.cpp::JavaBBinderHolder::JavaBBinder

```
class JavaBBinder : public BBinder
{
public:
    JavaBBinder(JNIEnv* env, jobject /* Java Binder */ c)
        : mVM(jnienv_to_javavm(env)), mObject(env->NewGlobalRef(object))
    {
        ALOGV("Creating JavaBBinder %p\n", this);
        gNumLocalRefsCreated.fetch_add(1, std::memory_order_relaxed);
        gcIfManyNewRefs(env);
    }
...
```

由上述代码可以知道 JavaBBinder 继承了 BBinder，那么 JavaBBinder 的作用是什么呢？当 Binder 驱动得到客户端的请求，紧接着会将响应发送给 JavaBBinder，这时会调用 JavaBBinder 的 onTransact 函数，代码如下所示。

frameworks/base/core/jni/androidutil/Binder.cpp::JavaBBinderHolder::JavaBBinder

```
    virtual status_t onTransact(
          uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
      {
          JNIEnv* env = javavm_to_jnienv(mVM);
          ALOGV("onTransact() on %p calling object %p in env %p vm %p\n", this, mObject,
env, mVM);
          IPCThreadState* thread_state = IPCThreadState::self();
          const int32_t strict_policy_before = thread_state->getStrictModePolicy();
          jboolean res = env->CallBooleanMethod(mObject,
gBinderOffsets.mExecTransact,
             code, reinterpret_cast<jlong>(&data), reinterpret_cast<jlong>(reply),
flags);//1

          ...
          return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
      }
```

在注释 1 处会调用 Java 层 Binder 的 execTransact 函数，代码如下所示。

frameworks/base/core/java/android/os/Binder.java

```
      private boolean execTransact(int code, long dataObj, long replyObj,
             int flags) {
   ...
          try {
             if (tracingEnabled) {
                Trace.traceBegin(Trace.TRACE_TAG_ALWAYS, getClass().getName() + ":"
+ code);
             }
             res = onTransact(code, data, reply, flags);//1
          } catch (RemoteException|RuntimeException e) {
             ...
          }
         ...
          return res;
      }
```

上述代码的关键点是注释 1 处的 onTransact 函数，AMS 实现了 onTransact 函数，从而完成业务实现。

从这里可以看出，JavaBBinder 并没有实现什么业务，当它接收到请求时，会调用 Binder 类的 execTransact 函数，execTransact 函数内部又调用了 onTransact 函数，系统服务会重写 onTransact 函数来实现自身的业务功能。

6.2.4 Java Binder 架构

Binder 架构简图如图 6-2 所示。

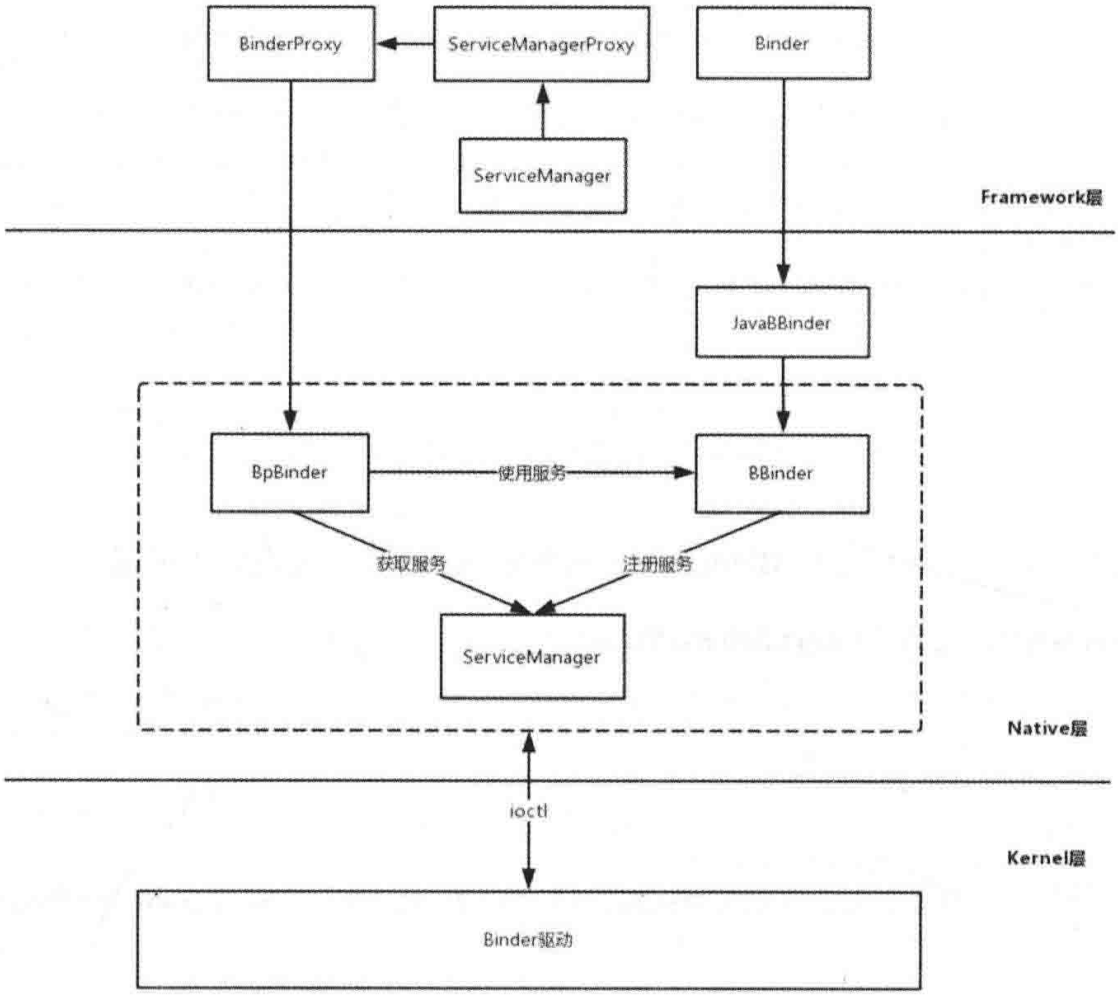


图 6-2 Binder 架构简图

Native Binder 的部分在前面章节已经讲过，这里主要来讲解 Java Binder 部分，从图 6-2 中可以看到。

（1）Binder 是服务端的代表，JavaBBinder 继承 BBinder，JavaBBinder 通过 mObject 变量指向 Binder。

（2）BinderProxy 是客户端的代表，ServiceManager 的 addService 等方法会交由 ServiceManagerProxy 来处理。

（3）ServiceManagerProxy 的成员变量 mRemote 指向 BinderProxy 对象，所以 ServiceManagerProxy 的 addService 等方法会交由 BinderProxy 来处理。

（4）BinderProxy 的成员变量 mObject 指向 BpBinder 对象，因此 BinderProxy 可以通过 BpBinder 和 Binder 驱动发送数据。

6.3　本章小结

本章的内容不是很多，一方面是因为第 5 章讲解的 Native Binder 为本章做好了铺垫，另一方面是我们要探究的关键点是 Java Binder 的架构，我们了解 Java Binder 的 JNI 注册和系统服务的注册过程就可以了，如果读者想要了解更多内容，可以关注一些博客。

第 7 章

Groovy 基础

关联章节：

第 8 章 Gradle 核心思想

本来这一章应该介绍 Gradle 的知识点，但是想要学好 Gradle，Groovy 是必须要掌握的。仅凭一个章节是介绍不完 Groovy 的，本章会带读者快速入门 Groovy，主要讲解 Groovy 和 Java 不同的部分，想要更多了解 Groovy 可以查看 Groovy 官方文档和 Groovy API 文档。

7.1 Groovy 概述

Groovy 是 Apache 旗下的一种基于 JVM 的面向对象的编程语言，既可以用于面向对象编程，又可以用作纯粹的脚本语言。在语言的设计上，Groovy 吸纳了 Python、Ruby 和 Smalltalk 语言的优秀特性，比如动态类型转换、闭包和元编程支持。

Groovy 与 Java 可以互相调用并结合编程，比如在写 Groovy 的时候忘记了语法可以直接按 Java 的语法继续写，也可以在 Java 中调用 Groovy 脚本。与 Java 相比，Groovy 语法更加灵活和简洁，可以用更少的代码来实现与 Java 实现的相同功能。

7.2　Groovy 编写和调试

Groovy 的代码可以在 Android Studio 和 IntelliJ IDEA 等 IDE 中进行编写和调试，这种方式的缺点是需要配置环境，这里推荐在文本中编写代码并结合命令行进行调试（文本推荐使用 Sublime Text），关于命令行请查看 8.2.5 节。

Groovy 编写和调试的具体操作步骤就是：在一个目录中新建 build.gradle 文件，在 build.gradle 文件中新建一个 task，在 task 中编写 Groovy 代码，用命令行进入这个 build.gradle 文件所在的目录，运行 gradle task 名称等命令行对代码进行调试，本节中的示例都是这样编写和调试的。

7.3　变量

Groovy 中用 def 关键字来定义变量，可以不指定变量的类型，其默认访问修饰符是 public。

```
def a = 1;
def int b = 1;
def c = "hello world";
```

7.4　方法

方法使用返回类型或 def 关键字定义，方法可以接收任意数量的参数，这些参数可以不申明类型，如果不提供可见性修饰符，则该方法为 public。

用 def 关键字定义方法。

```
task method <<{
add (1,2)
minus 1,2 //1
}
def add(int a,int b) {
 println a+b //2
}
def minus(a,b) {//3
 println a-b
```

```
}
```

如果指定了方法返回类型，则可以不需要 def 关键字来定义方法。

```
task method <<{
def number=minus 1,2
println number
}
int minus(a,b) {
  return a-b
}
```

如果不使用 return，则方法的返回值为最后一行代码的执行结果。

```
int minus(a,b) {
  a-b //4
}
```

从上面两段代码中可以知道 Groovy 中有很多省略的部分：

（1）语句后面的分号可以省略。

（2）方法的括号可以省略，比如注释 1 处和注释 2 处。

（3）参数类型可以省略，比如注释 3 处。

（4）return 可以省略，比如注释 4 处。

7.5 类

Groovy 类与 Java 类非常类似。

```
task method <<{
def p = new Person()
   p.increaseAge 5
println p.age
}
class Person {
   String name
   Integer age =10
   def increaseAge(Integer years) {
      this.age += years
   }
}
```

运行 gradle method 的打印结果为：

```
15
```

Groovy 类与 Java 类有以下区别：

（1）默认类的修饰符为 public。

（2）没有可见性修饰符的字段会自动生成对应的 setter 方法和 getter 方法。

（3）类不需要与它的源文件有相同的名称，但还是建议采用相同的名称。

7.6　语句

7.6.1　断言

Groovy 断言和 Java 断言不同，Groovy 断言一直处于开启状态，是进行单元测试的首选方式。

```
task method <<{
  assert 1+2 == 6
}
```

输出结果为：

```
Execution failed for task ':method'.
> assert 1+2 == 6
         | |
         3 false
```

当断言的条件为 false 时，程序会抛出异常，不再执行下面的代码，从输出可以很清晰地看到发生错误的地方。

7.6.2　for 循环

Groovy 支持 Java 的 for(int i=0;i<N;i++)和 for(int i :array)形式的循环语句，另外还支持 for in loop 形式，支持遍历范围、列表、Map、数组和字符串等多种类型。

```
//遍历范围
def x = 0
for ( i in 0..3 ) {
   x += i
```

```
}
assert x == 6
//遍历列表
def x = 0
for ( i in [0, 1, 2, 3] ) {
   x += i
}
assert x == 6
//遍历 Map 中的值
def map = ['a':1, 'b':2, 'c':3]
x = 0
for ( v in map.values() ) {
   x += v
}
assert x == 6
```

7.6.3 switch 语句

Groovy 中的 switch 语句不仅能够兼容 Java 代码，还可以处理更多类型的 case 表达式。

```
task method <<{
def x = 16
def result = ""

switch ( x ) {
   case "ok":
      result = "found ok"
   case [1, 2, 4, 'list']:
      result = "list"
      break
   case 10..19:
      result = "range"
      break
   case Integer:
      result = "integer"
      break
   default:
      result = "default"
}
assert result == "range"
}
```

case 表达式可以是字符串、列表、范围和 Integer 等，由于篇幅原因，这里只列举出了一小部分。

7.7　数据类型

Groovy 中的数据类型主要有以下几种：

- Java 中的基本数据类型。
- Groovy 中的容器类。
- 闭包。

7.7.1　字符串

Groovy 中的基本数据类型和 Java 大同小异，这里主要介绍字符串类型。在 Groovy 中有两种字符串类型，即普通字符串 String（java.lang.String）和插值字符串 GString（groovy.lang.GString）。其中普通字符串包括单引号字符串、双引号字符串、三引号字符串。

1．单引号字符串

在 Groovy 中单引号字符串和双引号字符串都可以定义一个字符串常量，只不过单引号字符串不支持插值。

```
'Android进阶解密'
```

2．双引号字符串

想要插值可以使用双引号字符串，插值指的是替换字符串中的占位符，占位符表达式为${}，或者以$为前缀。

```
def name = 'Android进阶之光'
println "hello ${name}"
println "hello $name"
```

3．三引号字符串

三引号字符串可以保留文本的换行和缩进格式，不支持插值。

```
task method <<{
def name = '''Android进阶之光
     Android进阶解密
Android进阶? '''
println name
}
```

其打印结果为：

```
Android进阶之光
      Android进阶解密
Android进阶?
```

String 是不可变的，GString 却是可变的，GString 和 String 即使有相同的字面量，它们的 hashCodes 的值也可能不同，因此应该避免使用 GString 作为 Map 的 key。

```
assert "one: ${1}".hashCode() != "one: 1".hashCode()
```

当双引号字符串中包含插值表达式时，字符串类型为 GString，因此上面的断言为 true。

7.7.2 List

Groovy 没有定义自己的集合类，它在 Java 集合类的基础上进行了增强和简化。Groovy 的 List 对应 Java 中的 List 接口，默认的实现类为 Java 中的 ArrayList。

```
def number = [1, 2, 3]
assert number instanceof List
def linkedList = [1, 2, 3] as LinkedList
assert linkedList instanceof java.util.LinkedList
```

可以使用 as 操作符来显式指定 List 的实现类为 java.util.LinkedList。

Groovy 获取元素比 Java 要简洁些，使用[]来获取 List 中具有正索引或负索引的元素。

```
task method <<{
def number  = [1, 2, 3, 4]
assert number [1] == 2
assert number [-1] == 4 //1

number << 5    //2
assert number [4] == 5
assert number [-1] == 5
}
```

注释 1 处的索引-1 是列表末尾的第一个元素。注释 2 处使用<<运算符在列表末尾追加一个元素。

7.7.3 Map

创建 Map 同样使用[]，需要同时指定键和值，默认的实现类为 java.util.LinkedHashMap。

```
def name = [one: '魏无羡', two: '杨影枫', three: '张无忌']
assert name['one']  == '魏无羡'
assert name.two  == '杨影枫'
```

Map 还有一个键关联的问题：

```
def key = 'name'
def person = [key: '魏无羡'] //1
assert person.containsKey('key')
person = [(key): '魏无羡'] //2
assert person.containsKey('name')
```

注释 1 处魏无羡的键值是 key 这个字符串，而不是 key 变量的值 name。如果想要以 key 变量的值为键值，则需要像注释 2 处一样使用 (key)，用来告诉解析器我们传递的是一个变量，而不是定义一个字符串键值。

7.7.4　闭包（Closure）

Groovy 中的闭包是一个开放的、匿名的、可以接受参数和返回值的代码块。

1．定义闭包

闭包的定义遵循以下语法。

```
{ [closureParameters -> ] statements }
闭包分为两个部分，分别是参数列表部分[closureParameters -> ]和语句部分 statements 。
```

参数列表部分是可选的，如果闭包只有一个参数，则参数名是可选的，Groovy 会隐式指定 it 作为参数名，如下所示。

```
{ println it }    //使用隐式参数 it 的闭包
```

当需要指定参数列表时，需要用->将参数列表和闭包体相分离，如下所示。

```
{ it -> println it }  //it 是一个显示参数
{ String a, String b ->
   println "${a} is a ${b}"
}
```

闭包是 groovy.lang.Cloush 类的一个实例，可以将闭包赋值给变量或字段，如下所示。

```
//将闭包赋值给一个变量
def println ={ it -> println it }
assert println instanceof Closure
//将闭包赋值给 Closure 类型变量
```

```
Closure do= { println 'do!' }
```

2. 调用闭包

闭包既可以当作方法来调用，又可以显示调用 call 方法。

```
def code = { 123 }
assert code() == 123          //闭包当作方法调用
assert code.call() == 123     //显示调用 call 方法
def isOddNumber = { int i -> i%2 != 0 }
assert isOddNumber(3) == true //调用带参数的闭包
```

7.8 I/O 操作

Groovy 的 I/O 操作要比 Java 的更简洁。

7.8.1 文件读取

我们可以在 PC 上新建一个 name.txt，在其中输入一些内容，然后用 Groovy 来读取该文件的内容。

```
def filePath = "D:/Android/name.txt"
def file = new File(filePath) ;
file.eachLine {
   println it
}
```

可以看出 Groovy 的文件读取是很简洁的，还可以更简洁些。

```
def filePath = "D:/Android/name.txt"
def file = new File(filePath) ;
println file.text
```

7.8.2 文件写入

文件写入同样十分简洁。

```
def filePath = "D:/Android/name.txt"
def file = new File(filePath);

file.withPrintWriter {
```

```
    it.println("三井寿")
    it.println("仙道彰")
}
```

7.9 其他

7.9.1 asType

asType 可以用于数据类型转换。

```
String a = '23'
int b = a as int
def c = a.asType(Integer)
assert c instanceof java.lang.Integer
```

7.9.2 判断是否为真

```
if (name != null && name.length > 0) {}
```

可以替换为

```
if (name) {}
```

7.9.3 安全取值

在 Java 中，要安全获取某个对象的值可能需要大量的 if 语句来进行非空判断。

```
if (school != null) {
    if (school.getStudent() != null) {
        if (school.getStudent().getName() != null) {
            System.out.println(school.getStudent().getName());
        }
    }
}
```

Groovy 中可以使用?.来进行安全取值。

```
println school?.student?.name
```

7.9.4 with 操作符

对同一个对象的属性进行赋值时，可以这样做。

```
task method <<{
Person p = new Person()
p.name = "杨影枫"
p.age = 19
p.sex = "男"
println p.name
}
class Person {
   String name
   Integer age
   String sex
}
```

使用 with 来进行简化。

```
Person p = new Person()
p.with {
  name = "杨影枫"
  age= 19
  sex= "男"
 }
println p.name
```

7.10 本章小结

本章简单介绍了 Groovy 的一些语法，其中包括变量、方法和数据类型等，相比 Groovy 的官方文档，本章介绍的内容并不多，但本章的目标是学习与 Android 相关的 Gradle，Groovy 并不是重点，在这里我们只需要了解本章所介绍的内容就可以了，如果遇到不懂的内容，则可以去查找 Groovy 官方文档和 Groovy API 文档。

第 8 章

Gradle 核心思想

关联章节：

第 7 章 Groovy 基础

第 9 章 Gradle 的 Android 插件

本章我们来学习 Gradle，Gradle 是一个基于 JVM 的新一代构建工具，关于 Gradle 完全可以写一本书，本章会针对 Android 开发来对 Gradle 的知识进行精简讲解。目前 Gradle 已经应用于多个 Android 开发的技术体系中，比如构建系统、插件化、热修复和组件化等，如果我们不了解 Gradle，那么对于上述技术体系的了解会大打折扣。

本章 Gradle 的内容都尽量不与 Android 和 Android Studio 有所关联，其目的是摆脱 Android 的束缚，Gradle 本身就是一门技术，当我们了解了 Gradle 的核心思想后，再去学习 Android Gradle（Gradle 的 Android 插件）时会豁然开朗。

8.1 Gradle 概述

在学习 Gradle 之前，先来了解现在要用 Gradle 的原因。

8.1.1 项目自动化

Gradle 是一个构建工具，那么为什么要用构建工具？这就需要先从项目自动化开始讲起。

在开发软件时，会面临相似的情况，我们需要用 IDE 来进行编码，当完成一些功能时会进行编译、单元测试和打包等工作，这些工作都需要开发人员手动来实现。而一般的软件都是迭代式开发的，一个版本接着一个版本，每个版本又可能有很多功能，如果开发每次实现功能时都需要手动进行编译、单元测试和打包等工作，那么显然会非常耗时而且也容易出现问题，因此项目自动化应运而生，它有以下优点：

（1）项目自动化可以尽量防止开发手动介入从而节省了开发的时间并减少错误的发生。

（2）项目自动化可以自定义有序的步骤来完成代码的编译、测试和打包等工作，使重复的步骤变得简单。

（3）IDE 可能受到不同操作系统的限制，而自动化构建是不会依赖于特定的操作系统和 IDE 的，它具有平台无关性。

8.1.2 构建工具

构建工具用于实现项目自动化，是一种可编程的工具，我们可以用代码来控制构建流程，最终生成可交付的软件。构建工具可以帮助我们创建一个重复的、可靠的、无须手动介入的、不依赖于特定操作系统和 IDE 的构建。这里以 APK 的构建过程来举例。

1．APK 的构建过程

APK 的构建过程可以根据官方提供的流程图，如图 8-1 所示。

APK 构建的过程主要分为以下几个步骤。

（1）通过 AAPT（Android Asset Packaging Tool）打包 res 资源文件，比如 AndroidManifest.xml 和 xml 布局文件等，将这些 xml 文件编译为二进制，其中 assets 文件夹和 raw 文件夹的文件不会被编译为二进制，最终会生成 R.java 文件和 resources.arsc 文件。

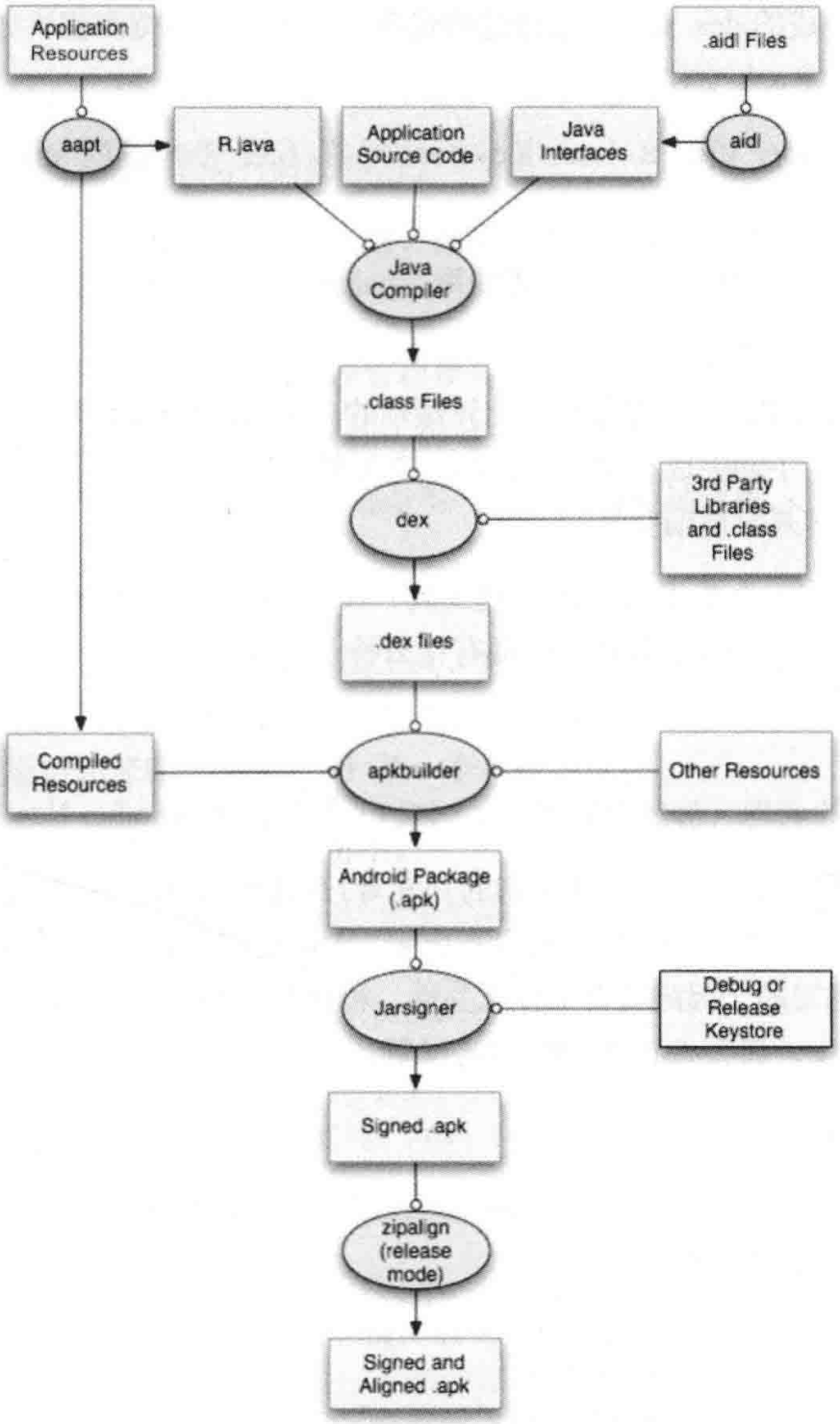


图 8-1　APK 的构建过程

(2) AIDL 工具会将所有 aidl 接口转化为对应的 Java 接口。

(3) 所有 Java 代码，包括 R.java 文件和 Java 接口都会被 Java 编译器编译成.class 文件。

(4) Dex 工具会将第 (3) 步生成的.class 文件、第三方库和其他.class 文件都编译成.dex 文件。

（5）第（4）步编译生成的.dex 文件、编译过的资源、无须编译的资源都（如图片等）会被 ApkBuilder 工具打包成 apk 文件。

（6）使用 Debug Keystore 或者 Release Keystore 对第（5）步生成的 apk 文件进行签名。

（7）如果是对 APK 正式签名，那么还需要使用 zipalign 工具对 APK 进行对齐操作，这样应用运行时会减少内存的开销。

从以上步骤可以看出，APK 的构建过程是比较烦琐的，而且这个构建过程又是时常重复的，如果没有构建工具，手动完成构建工作，那么毫无疑问对于开发人员是个折磨，也会产生诸多问题，导致项目开发周期变长。

在 Gradle 出现之前，有三个基于 Java 的构建工具：Ant、Gant 和 Maven，它们被应用在 Java 或者 Android 开发中，我们来看看它们都有什么特点。

2. Apache Ant

Ant 在这里不是蚂蚁的意思（虽然它的图标是蚂蚁），其指的是 Another Neat Tool。

它是由 James Duncan Davidson（Tomcat 最初的开发者）开发的，最初是用来构建 Tomcat 的。在 2000 年，Ant 成为一个独立的项目并被发布出来。Ant 是由 Java 编写的构建工具，它的核心代码是由 Java 编写的，因此具有平台无关性，构建脚本是 XML 格式的（默认为 build.xml），如果熟悉 XML，那么 Ant 就比较容易理解。

Ant 构建脚本的样式如下所示。

build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="test" default="hello">
   <echo message="running build.xml which is equivalent to build.gant"/>
   <property file="build.properties"/>
   <target name="init"  description="init target" >
      <echo message="Executing init target"/>
   </target>
   <target name="hello" depends="init" description="say hello target">
      <echo message="${echo.msg}"/>
   </target>
</project>
```

Ant 的构建脚本由三个基本元素组成：一个 project（工程）、多个 target（目标）和可

用的 task（任务）。

Apache Ant 有以下缺点。

（1）Ant 无法获取运行时的信息。

（2）XML 作为构建脚本的语言，如果构建逻辑复杂，那么构建脚本就会很长且难以维护。

（3）Ant 需要配合 Ivy（一种管理项目依赖工具），否则 Ant 很难管理依赖。

（4）Ant 在如何组织项目结构方面没有给出任何指导，这导致 Ant 虽然灵活性高，但这样的灵活性导致每个构建脚本都是唯一的而且很难被理解。

3. Gant

Gant 是一个基于 Ant 的构建工具，它在 Ant 的基础上用 Groovy 写的 DSL（领域特定语言）。如果用 Ant 实现构建，但是不喜欢用 XML 来编写构建脚本或者现有的 XML 构建脚本很难维护和管理，那么 Gant 是一个不错的选择。

Gant 构建文件的样式如下所示。

build.gant

```
Ant.echo(message : 'running build.gant')
Ant.property(file : 'build.properties')
def antProperty = Ant.project.properties
target(init : 'init target') {
   echo(message : 'Executing init target')
}
target(hello : 'say hello target') {
   depends(init)
   echo(message : antProperty.'echo.msg')
}
setDefaultTarget(hello)
```

这个 build.gant 等同于此前 Ant 的 build.xml。

4. Apache Maven

Maven 于 2004 年发布，它的目标是改进开发人员在使用 Ant 时面临的一些问题。Maven 最初是为了简化 Jakarta Turbine 项目的构建的，它经历了从 Maven 到 Maven3 的发展，Maven 作为后来者，继承了 Ant 的项目构建功能，同样采用了 XML 作为构建脚本的格式。Maven

具有依赖管理和项目管理的功能，并提供了中央仓库，能够帮助我们自动下载库文件。

Maven 的构建脚本的样式如下所示。

pom.xml

```
<project xmlns="http://maven.apache.……" xmlns:xsi="http://www.w3.……"
  xsi:schemaLocation="http://maven.apache…… http://maven.apache……">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache……</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Maven 相比 Ant 有以下优点。

（1）Ant 是过程式的，开发者需要显示的指定每个目标，以及完成该目标所需要执行的任务。开发者需要重新编写每一个项目的过程，这样会产生大量的重复。而 Maven 是声明式的，项目的构建过程和过程中的各个阶段都由插件实现，开发者只需要声明项目的基本元素就可以了，这很大程度消除了重复。

（2）Ant 本身是没有依赖管理的，需要配合 Ivy 来进行依赖管理，而 Maven 本身就提供了依赖管理。

（3）Maven 使用的是约定而不是配置，它为工程提供了合理的默认行为，项目会知道去哪个目录寻找源码及构建运行时有哪些任务去执行，如果我们项目遵从默认值，那么只需要写几行 XML 配置脚本就可以了，而 Ant 是使用配置且没有默认行为的。

Maven 也有以下缺点。

（1）Maven 提供了默认的结构和生命周期，这些可能不适合我们的项目需求。

（2）为 Maven 定制的扩展过于烦琐。

（3）Maven 的中央仓库比较混乱，当无法从中央仓库中得到需要的类库时，我们可以手工下载并复制到本地仓库中，也可以建立组织内部的仓库服务器。

（4）国内连接 Maven 的中央仓库比较慢，需要连接国内的 Maven 镜像仓库。

（5）Maven 缺乏相关的技术文档，不便于使用和理解。

8.1.3　Gradle 的特性

Gradle 基于 JVM，是一款专注于灵活性和性能的开源构建工具，它的特性如图 8-2 所示。

图 8-2　Gradle 的特性

从图 8-2 可以看出，Gradle 结合 Ant 和 Maven 等构建工具的最佳特性。它有约定优于配置的方法、强大的依赖管理，它的构建脚本使用 Groovy 或 Kotlin 编写，Gradle 是 Android 的官方构建工具。Gradle 的构建脚本的样式如下所示。

build.gradle

```
apply plugin:'java'
group='com.mycompany.app'
archivesBaseName='my-app'
version='1.0-SNAPSHOT'

repositories{
  mavenCentral()
```

```
}

dependencies{
  testCompile 'junit:4.11'
}
```

这个 build.gradlet 等同于此前 Maven 的 pom.xml。由此可以看出 Groovy 编写构建脚本代码量更少，可读性更强。

下面列出 Gradle 与竞争对手不同的特性。

1. 轻松的可拓展性

Gradle 有非常良好的拓展性。如果我们想要在多个构建或者项目中分享可重用代码，Gradle 的插件会帮助我们实现。将 Gradle 插件应用在项目中，它会在项目构建过程中提供很多帮助：为我们添加项目的依赖的第三方库、添加有用的默认设置和约定（源码位置、单元测试代码位置）。其中 Android Gradle 插件继承 Java Gradle 插件，在本章 8.5 节会介绍插件的内容。

2. 采用了 Groovy

Ant 和 Maven 的构建脚本是由 XML 来编写的，如果 XML 逻辑复杂且内容太多就不容易维护。Gradle 可以使用 Groovy 来构建脚本，Groovy 是一种基于 JVM 的动态语言，它的语法和 Java 非常相似并兼容 Java，因此我们不用担心学习 Groovy 的成本。Groovy 在 Java 的基础上增加了很多动态类型和灵活的特性，相比 XML，Gradle 更具有表达性和可读性。

3. 强大的依赖管理

Gradle 提供了可配置的、可靠的依赖管理方案。一旦依赖的库被下载并存储到本地缓存中，我们的项目就可以使用了。依赖管理很好地实现了在不同的平台和机器上产生相同的构建结果。

4. 灵活的约定

Gradle 可以为构建项目提供引导和默认值，如果使用这种约定，那么 Gradle 构建脚本代码量比较少。相比 Ant，Gradle 不仅提供了约定，还可以轻松打破约定。

5. Gradle Wrapper

Gradle Wrapper 是对 Gradle 的包装，它的作用是简化 Gradle 本身的下载、安装和构建，

比如它会在我们没有安装 Gradle 的情况下，去下载指定版本的 Gradle 并进行构建。Gradle 的版本很多，所以有可能出现版本兼容的问题，这就需要 Gradle Wrapper 去统一 Gradle 的版本，避免开发团队因为 Gradle 版本不一致而产生问题。

6. 可以和其他构建工具集成

Gradle 可以和 Ant、Maven 和 Ivy 进行集成，比如我们可以把 Ant 的构建脚本导入 Gradle 的构建中。

7. 底层 API

Gradle 显然无法满足所有企业级构建的要求，但是可以通过 Hook Gradle 的生命周期来监控和配置构建脚本。

8. 社区的支持和推动

Gradle 是一个开源的项目，它遵循了 Apache License 2.0 协议。Gradle 的优良特性吸引了很多开发者并形成了 Gradle 社区，很多开源软件的开发者为 Gradle 的核心代码做出了共享。

8.2　Gradle 基础知识

在 8.1 节中，我们学习了常见的构建工具，以及 Gradle 与这些构建工具相比有哪些不同的特性。本节我们继续来学习 Gradle 入门需要掌握的基础知识，包括配置环境、实现 Hello World、Gradle 的任务、Gradle 日志和 Gradle 命令行，掌握这些能够更好地理解 Gradle。

8.2.1　配置 Gradle 环境

安装 Gradle 前要确保系统已经配置好 JDK 的环境，要求 JDK 的版本在 1.7 或更高的版本。

Gradle 的安装有以下两种形式。

（1）通过包管理安装，比如 Windows 平台的 Chocolatey 和 Scoop，macOS 平台的 MacPortsl 和 Homebrew 等。

（2）手动安装。

关于包管理器安装可以查看官方文档，这部分内容主要介绍手动安装，在 Gradle 官方网站中下载我们想要 Gradle 版本的 binary-only。

1. Windows 平台安装

与配置 Java 环境类似，在编辑系统变量中添加 GRADLE_HOME，如图 8-3 所示。

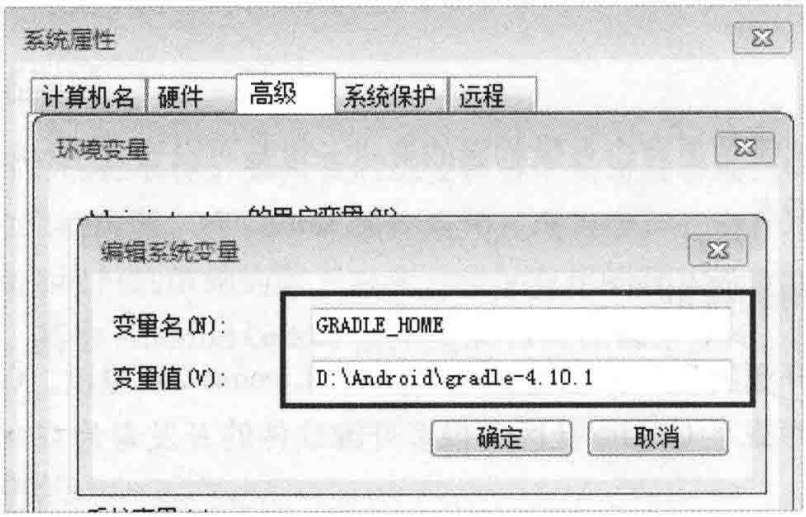


图 8-3　在编辑系统变量中添加 GRADLE_HOME

然后再将%GRADLE_HOME%\bin 添加到 path 系统变量中就可以了，运行 gradle -v 命令来进行验证。

2. macOS 平台安装

macOS 平台安装步骤如下。

（1）打开终端运行 open -e .bash_profile，打开.bash_profile 文件来配置环境变量。

（2）在.bash_profile 文件中将 gradle 路径添加到 path 环境变量中，比如：

```
export PATH=$PATH:~/develop/gradle-4.10.1/bin
```

（3）保存.bash_profile 文件，在终端中运行 source ~/.bash_profile，更新.bash_profile 文件。

（4）在终端上运行 gradle -v 命令，并查看是否配置成功，如图 8-4 所示。

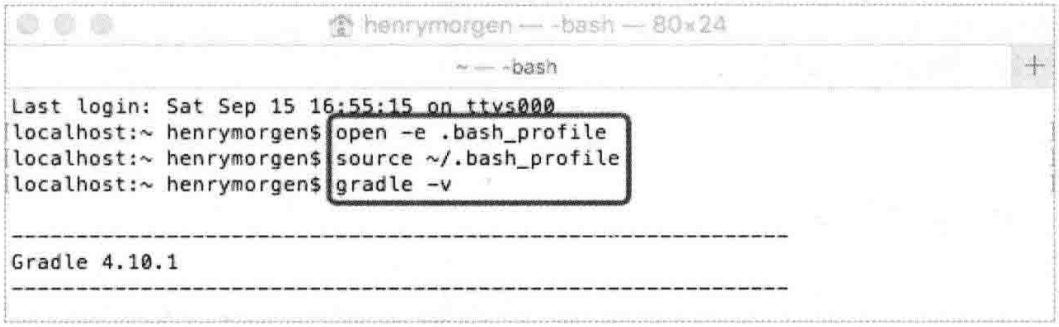


图 8-4　在终端上运行 gradle -v 命令

8.2.2　实现 Hello World

配置完成 Gradle 环境后，按照惯例要实现 Gradle 的 Hello World，这里以 Windows 平台为例。

build.gradle 为 Gradle 默认的构建脚本文件，运行 Gradle 命令时，会从当前目录下寻找 build.gradle 文件来执行构建。

我们首先新建一个目录，比如 D:\Android\gradle_demo，在这个目录中新建一个 build.gradle 文件，输入以下内容：

```
task hello {
   doLast {
      println 'Hello world!'
   }
}
```

然后在该文件的所在目录下运行 gradle -q hello 构建脚本，就会在控制台打印出'Hello world!'。

项目构建比较复杂，为了使用各种开发语言的开发者都能够快速构建项目，专家们开发出了 Gradle 这个基于 Groovy 的 DSL(Domain Specifc Language)，DSL 意为领域特定语言，只用于某个特定的领域。我们只要按照 Groovy 的 DSL 语法来写，就可以轻松构建项目了。

task（任务）和 action（动作）是 Gradle 的重要元素。上述代码中，task 代表一个独立的原子性操作，比如复制一个文件，编译一次 Java 代码，这里我们简单地将其定义一个名为 hello 的任务。doLast 代表 task 执行的最后一个 action，通俗来讲就是 task 执行完毕后会回调 doLast 中的代码，在上面这个示例中就会打印 'Hello world!'

上面的示例还可以写的更简洁一些，操作符<<是 doLast 方法的快捷版本，它们有相同的作用，如下所示。

```
task hello << {
   println 'Hello world!'
}
```

8.2.3 Gradle 的任务

为了讲解后面的 Gradle 命令行，这里简单介绍下 Gradle 的任务，包括创建任务、任务依赖、动态定义任务和任务的分组和描述。

8.2.3.1 创建任务

除了 8.2.2 节实现 Hello World 的示例采用的创建任务方式，还有以下 3 种创建任务方式。

（1）直接用任务名称创建。

```
def Task hello=task(hello)
hello.doLast{
 println "hello world"
}
```

（2）任务名称+任务配置创建。

```
def Task hello=task(hello,group:BasePlugin.BUILD_GROUP)
hello.doLast{
 println "hello world"
}
```

其中 group 为任务配置项，它代表了分组。

（3）TaskContainer 的 create 方法创建。

```
tasks.create(name: 'hello') << {
   println "hello world"
}
```

此前创建任务的方式最终都会调用 tasks 的 create 方法，其中 tasks 类型为 TaskContainer。

8.2.3.2 任务依赖

任务依赖会决定任务运行的先后顺序，被依赖的任务会在定义依赖的任务之前执行。创建任务间的依赖关系如下所示。

```
task hello << {
```

```
    println 'Hello world!'
}
task go(dependsOn: hello) << {
    println "go for it"
}
```

在 hello 任务的基础上增加了一个名为 go 的任务，通过 dependsOn 来指定依赖的任务为 hello，因此 go 任务在 hello 之后运行。

运行 gradle -q go 构建脚本，打印结果如下。

```
Hello world!
go for it
```

8.2.3.3　动态定义任务

动态定义任务指的是在运行时来定义任务的名称，如下所示。

```
3.times {number ->
    task "task$number" << {
        println "task $number"
    }
}
```

这里用到了 Groovy 语法，times 是 Groovy 在 java.lang.Number 中拓展的方法，是一个定时器。3.times 中循环创建了三个新任务，隐式变量 number 的值为 0、1、2，任务的名称由 task 加上 number 的值组成，达到了动态定义任务的目的。

运行 gradle -q task0 构建脚本，打印结果如下。

```
task 0
```

8.2.3.4　任务的分组和描述

Gradle 有任务组的概念，可以为任务配置分组和描述，以便于管理任务，拥有良好的可读性。改造 8.2.3.2 节的示例，为 hello 任务添加分组和描述，如下所示。

```
task hello {
group = 'build'
description = 'hello world'
    doLast {
     println "任务分组: ${group}"
        println "任务描述: ${description}"
    }
}
```

```
task go(dependsOn: hello) << {
   println "go for it"
}
```

也可以采用 8.2.3.1 节中的创建任务方式来为任务添加分组和描述，如下所示。

```
def Task hello=task(hello)
hello.description ='hello world'
hello.group=BasePlugin.BUILD_GROUP
hello.doLast{
println "任务分组: ${group}"
   println "任务描述: ${description}"
}
task go(dependsOn: hello) << {
   println "go for it"
}
```

8.2.4 Gradle 日志级别

和 Android 一样，Gradle 也定义了日志级别，如表 8-1 所示。

表 8-1 Gradle 日志级别

级　别	用　于
ERROR	错误消息
QUIET	重要的消息
WARNING	警告消息
LIFECYCLE	进度消息
INFO	信息性消息
DEBUG	调试消息

前面章节我们通过 gradle -q +任务名称来运行一个指定的 task，这个 q 是命令行开关选项，通过开关选项可以控制输出的日志级别，如表 8-2 所示。

表 8-2 Gradle 开关选项

开关选项	输出日志级别
无日志选项	LIFECYCLE 及更高级别
-q 或者 --quiet	QUIET 及更高级别
-i 或者 --info	INFO 及更高级别
-d 或者 --debug	DEBUG 及更高级别

8.2.5　Gradle 命令行

从命令行的角度来看，Gradle 和 Git 类似，命令都可以用一些 IDE 或图形工具来代替，但是如果我们对 Gradle 命令行熟悉，会帮助我们更好地理解 Gradle，高效地运用 Gradle。

1．获取所有任务信息

本节的命令行以 8.2.3 节的代码为例，此前我们通过 gradle -q +任务名称来运行一个指定的任务，如果不知道任务的名称，则可以通过运行 gradle -q tasks 命令来获取所有任务信息，这样就不需要打开源码了，如下所示。

```
Build tasks
-----------
hello - hello world

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root projec
t 'gradle_demo'.
components - Displays the components produced by root project 'gradle_demo'. [in
cubating]
dependencies - Displays all dependencies declared in root project 'gradle_demo'.

dependencyInsight - Displays the insight into a specific dependency in root proj
ect 'gradle_demo'.
dependentComponents - Displays the dependent components of components in root pr
oject 'gradle_demo'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'gradle_demo'. [incubat
ing]
projects - Displays the sub-projects of root project 'gradle_demo'.
properties - Displays the properties of root project 'gradle_demo'.
tasks - Displays the tasks runnable from root project 'gradle_demo'.

To see all tasks and more detail, run gradle tasks --all

To see more detail about a task, run gradle help --task <task>
```

在默认情况下，只会显示那些被分组的任务的名称和描述。比如 Build tasks（Build 任

务组）中有我们定义的 hello 任务，Build Setup tasks 中有 init 和 wrapper，Help tasks 中有 buildEnvironment 和 components 等。

2. 排除任务

如果我们不想运行 go 任务，则可以运行 gradle hello -x go 命令，如下所示。

```
> Task :hello
任务分组: build
任务描述: hello world

Deprecated Gradle features were used in this build, making it incompatible with
Gradle 5.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle……
mand_line_warnings

BUILD SUCCESSFUL in 2s
1 actionable task: 1 executed
```

由上述代码可以看出，并没有运行 go 任务。

3. 获取任务帮助信息

通过运行 gradle -q help --task hello 命令来显示 hello 任务的帮助信息，如下所示。

```
Detailed task information for hello

Path
    :hello

Type
    Task (org.gradle.api.Task)

Description
    hello world

Group
    build
```

由上述代码可以看到 hello 任务的路径、类型、描述和分组。

4. 多任务调用

如果想要多任务调用，则可以使用如下代码。

```
task helloWorld << {
   println 'Hello world!'
}
task goForit<< {
   println "go for it"
}
```

通过命令行一次执行多个任务，每个任务通常只会执行一次，无论是在命令行中指定任务还是任务依赖，上面的示例中，我们在运行 gradle helloWorld goForit 时，会先执行 helloWorld 任务后再执行 goForit 任务。

5. 任务名称缩写

可以对使用驼峰命名的任务进行缩写，对于名称特别长的任务，这个特性非常有用，比如多任务调用中的示例只需要执行 gradle hW gF 就可以了，不过需要注意的是，任务名称的缩写必须是唯一的，如果多任务调用示例中第二个任务的名称为 helloWangshu，那么就会报错。

8.3　被忽视的 Gradle Wrapper

我们已经学习了为什么要用 Gradle、Gradle 的基础、Groovy 的基础，这些内容为 Gradle 的入门打下了基础，这一节我们接着学习 Gradle Wrapper。了解 Gradle Wrapper 可以更好地理解 Gradle，Gradle Wrapper 在日常开发中看似“不起眼”，实则非常重要。

8.3.1　为什么需要 Gradle Wrapper

Gradle Wrapper 被称为 Gradle 包装器，是对 Gradle 的一层包装。为什么需要 Gradle Wrapper 呢？比如在一个开发团队中，如果每进来一个成员都需要在计算机中安装 Gradle，那么这时运行 Gradle 的环境和版本就会给构建结果带来不确定性。针对这个问题，Gradle 提供了一个解决方案，即 Gradle Wrapper，它是一个脚本，可以在计算机没有安装 Gradle 的情况下运行 Gradle 构建，并且能够指定 Gradle 的版本，开发人员可以快速启动并运行 Gradle 项目，而不必手动安装，这样就标准化了项目，从而提高了开发效率。Android Studio 在新建项目时会自带 Gradle Wrapper，这也是我们很少单独去下载安装 Gradle 的原因。Gradle Wrapper 的工作流程如图 8-5 所示。

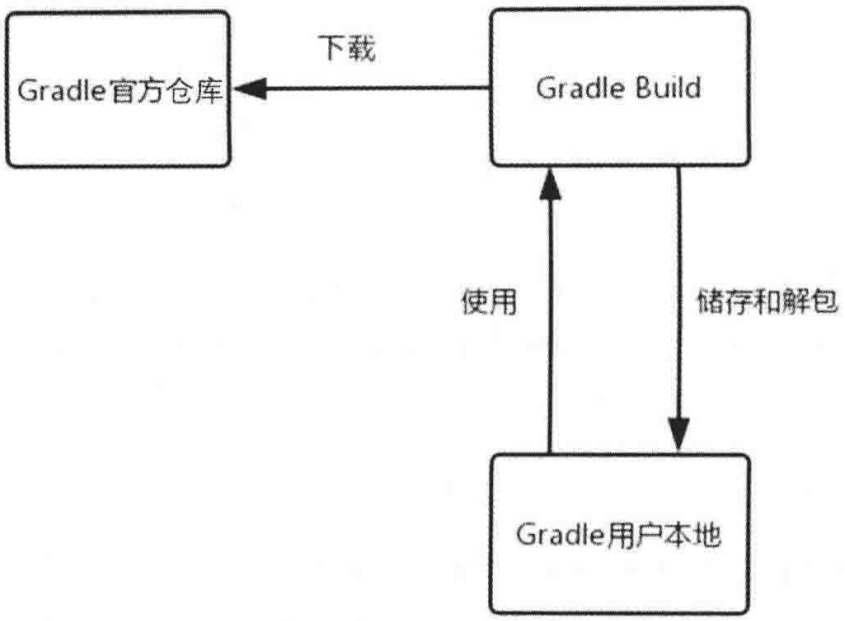


图 8-5　Gradle Wrapper 的工作流程

当使用 Gradle Wrapper 启动 Gradle 时，如果指定版本的 Gradle 没有被下载关联，那么会先从 Gradle 官方仓库将该版本的 Gradle 下载到用户本地，进行解包并执行批处理文件。后续的构建运行都会重用这个解包的运行时安装程序。

8.3.2　构建 Gradle Wrapper

首先要确保计算机中配置了 Gradle 的环境，Gradle 已经内置了 Wrapper Task，执行 Wrapper Task 就可以在项目目录中生成 Gradle Wrapper 的目录文件。在项目根目录中执行 gradle wrapper 就可以了，如下所示。

```
$ gradle wrapper
> Task :wrapper

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

这时会在项目根目录中生成如下文件：

```
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
└── gradlew.bat
```

每个文件的含义如下。

- gradle-wrapper.jar：包含 Gradle 运行时的逻辑代码。
- gradle-wrapper.properties：负责配置包装器运行时行为的属性文件，用来配置使用哪个版本的 Gradle 等属性。
- gradlew：在 Linux 平台下，用于执行 Gralde 命令的包装器脚本。
- gradlew.bat：在 Windows 平台下，用于执行 Gralde 命令的包装器脚本。

当生成了这些目录与文件后，用户就可以将工程 push 到远程，当其他用户 clone 下来后就可以直接进行项目的构建，节省了用户单独下载 Gradle 的时间，并且可以确保 Gradle 版本的一致。

也可以用 gradle 命令行选项来生成 gradle wrapper。

--gradle-version：用于下载和执行指定的 gradle 版本。

--distribution-type：指定下载 Gradle 发行版的类型，可用选项有 bin 和 all，默认值是 bin，-bin 发行版只包含运行时，但不包含源码和文档。

--gradle-distribution-url：指定下载 Gradle 发行版的完整 URL 地址。

--gradle-distribution-sha256-sum：使用的 SHA 256 散列和验证下载的 Gradle 发行版。

比如使用命令行：gradle wrapper --gradle-version 4.2.1 --distribution-type all，就可以生成版本为 4.2.1 的包装器，并使用-all 发行版。

8.3.3　配置 Gradle Wrapper

gradle-wrapper.properties 是 Gradle Wrapper 的属性文件，用来配置 Gradle Wrapper，Gradle 4.2.1 版本对应的 gradle-wrapper.properties 如下所示。

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle……
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

其中字段的含义如下。

- distributionBase：Gradle 解包后存储的主目录。
- distributionPath：distributionBase 指定目录的子目录。distributionBase+distributionPath

就是 Gradle 解包后的存放位置。

- distributionUrl：Gradle 发行版压缩包的下载地址。
- zipStoreBase：Gradle 压缩包存储主目录。
- zipStorePath：zipStoreBase 指定目录的子目录。zipStoreBase+zipStorePath 就是 Gradle 压缩包的存放位置。

这里我们需要关注的是 distributionUrl 字段，如果官方的地址下载不了或者下载缓慢，则可以将这个地址换为其他镜像地址，或者把 Gradle 发行版压缩包放在服务器上以供下载。

8.3.4 使用 Gradle Wrapper

使用 Gradle Wrapper 不是用 Gradle 命令，而是用 gradlew 和 gradlew.bat 脚本。在 build.gradle 中加入如下语句。

```
task test {
   doLast {
      println 'Hello world!'
   }
}
```

以 Windows 平台为例，我们进入项目所在的根目录执行 gradlew.bat test。

```
f:\app>gradlew.bat test
Downloading https://services.gradle……
.............................................................
Starting a Gradle Daemon (subsequent builds will be faster)

> Task :test
Hello world!
```

如果计算机中没有 Gradle 发行版，则 Gradle 包装器会将 Gradle 发行版压缩包下载到本地中并进行解压，比如计算机中的存储路径为：C:\Users\52501.gradle\wrapper\dists\gradle-4.2.1-bin\dajvke9o8kmaxbu0kc5gcgeju\gradle-4.2.1。

如果此后 Gradle 属性文件的 distributionUrl 属性不变，就会一直使用本地的 Gradle 发行版。如果我们再次执行 gradlew.bat test，就会和调用 Gradle 命令一样。

```
f:\app>gradlew.bat test
Starting a Gradle Daemon (subsequent builds will be faster)

> Task :test
Hello world!
```

8.3.5　升级 Gradle Wrapper

升级 Gradle Wrapper 有两种方式，第一种方式是设置 Gradle 属性文件的 distributionUrl 属性，第二种方式是运行 wrapper 任务，推荐使用第二种方式。当前本地的 Gradle 版本为 4.2.1，如果想升级为 5.1.1，则只需要运行 gradlew wrapper --gradle-version 5.1.1 命令即可。

```
f:\app>gradlew wrapper --gradle-version 5.1.1

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

运行 gradlew -v 命令来检查 Gradle 的版本。

```
f:\app>gradlew -v
Downloading https://services.gradle……
.........................................................................
Unzipping
C:\Users\52501\.gradle\wrapper\dists\gradle-5.1.1-bin\90y9l8txxfw1s2o6ctiqeruwn\gr
adle-5.1.1-bin.zip to
C:\Users\52501\.gradle\wrapper\dists\gradle-5.1.1-bin\90y9l8txxfw1s2o6ctiqeruwn

Welcome to Gradle 5.1.1!

Here are the highlights of this release:
 - Control which dependencies can be retrieved from which repositories
 - Production-ready configuration avoidance APIs

For more details see https://docs.gradle……

------------------------------------------------------------
Gradle 5.1.1
------------------------------------------------------------

Build time:   2019-01-10 23:05:02 UTC
Revision:     3c9abb645fb83932c44e8610642393ad62116807

Kotlin DSL:   1.1.1
Kotlin:       1.3.11
Groovy:       2.5.4
Ant:          Apache Ant(TM) version 1.9.13 compiled on July 10 2018
JVM:          1.8.0_191 (Oracle Corporation 25.191-b12)
OS:           Windows 10 10.0 amd64
```

由于本地的版本不是 Gradle 5.1.1，那么会将下载下来的 Gradle 压缩包存储起来并进行解包，具体内容见上述打印日志。

8.3.6 自定义 Gradle Wrapper

Gradle 已经内置了 Wrapper Task，因此构建 Gradle Wrapper 会生成 Gradle Wrapper 的属性文件，这个属性文件可以通过自定义 Wrapper Task 来设置。比如我们想要将下载的 Gradle 版本修改为 4.2.1，可以设置成如下形式。

```
task wrapper(type: Wrapper) {
   gradleVersion = '4.2.1'
}
```

也可以设置 Gradle 发行版压缩包的下载地址和 Gradle 解包后的本地存储路径等配置。

```
task wrapper(type: Wrapper) {
   gradleVersion = '4.2.1'
   distributionUrl = '../../gradle-4.2.1-bin.zip'
   distributionPath=wrapper/dists

}
```

distributionUrl 属性可以设置为本地的项目目录，也可以设置为网络地址。

8.4 Gradle 插件基础

8.4.1 Gradle 插件概述

提到 Gradle 插件前，我们首先要了解什么是插件。在《Android 进阶解密》中，讲解了插件化的原理，并讲解了什么是插件，下面来看初始的机器人的功能，如图 8-6 所示。

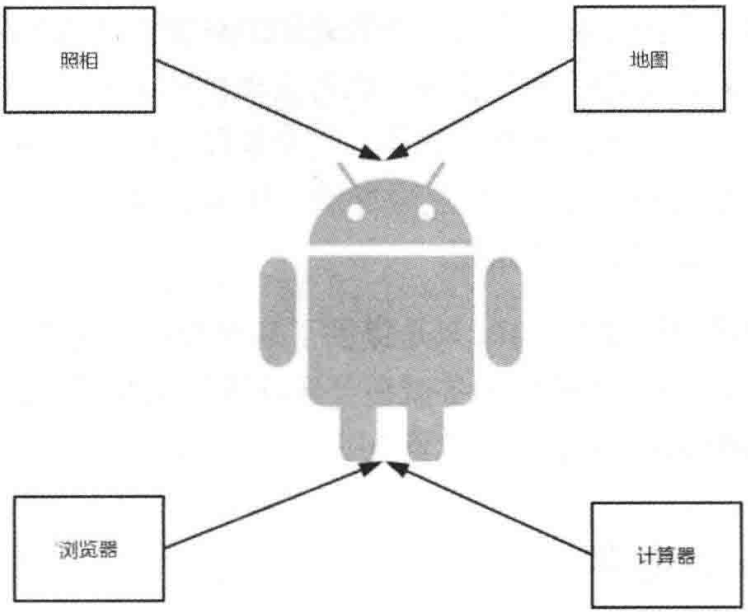


图 8-6　初始的机器人的功能

由图 8-6 可知，可以看到初始的机器人的功能只有照相、地图、浏览器、计算机，这显然是比较乏味的，我们可以给这个机器人安装很多其他应用，使它能够提供更多功能，如图 8-7 所示。

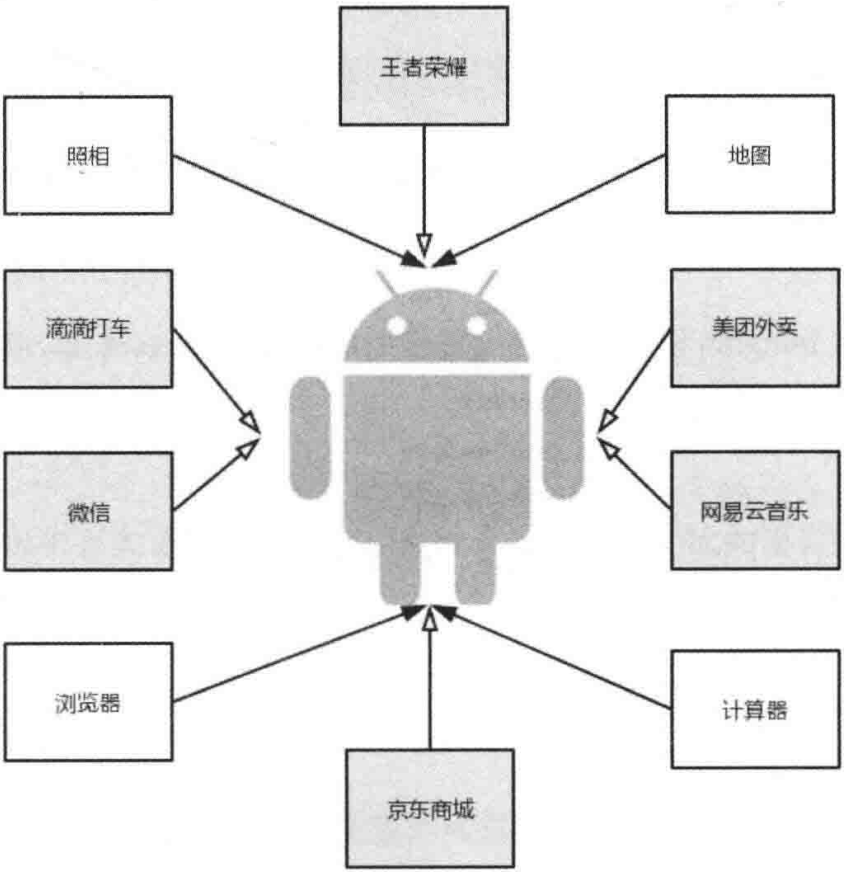


图 8-7　机器人安装很多其他应用后的功能

我们给这个机器人安装了很多应用，这些应用不仅覆盖了人们的衣食住行，还提供了娱乐功能，可以玩游戏、听音乐和购物等，机器人也得到了极大的提升，能够为人类提供更多服务。这些安装的应用可以理解为插件，这个插件可以自由插拔，比如我们需要玩游戏时可以安装王者荣耀，如果不好玩就把它卸载。其实 Android、iOS、macOS 等操作系统采用的都是这种思想，Gradle 也是如此。

Gradle 本身和初始的机器人一样，只是提供了基本的核心功能，其他特性比如编译 Java 源码的能力、编译 Android 工程的能力等就需要通过插件来实现了。这里主要讲的是 Gradle 插件，而不是 Android Gradle 插件。

8.4.2 应用 Gradle 插件

应用 Gradle 插件主要有两个步骤，第一步是解析插件，第二步是把插件应用到项目中，应用插件通过 Project.apply 方法来完成。

在 Gradle 中一般有两种类型的插件，分别为脚本插件和对象插件。脚本插件是额外的构建脚本，它会进一步配置构建，可以把它理解为一个普通的 build.gradle。对象插件又叫二进制插件，是实现了 Plugin 接口的类，下面分别介绍如何使用脚本插件和对象插件。

8.4.2.1 脚本插件

在 8.3 节的示例基础上定义一个 other.gradle，该示例的脚本目录结构如图 8-8 所示。

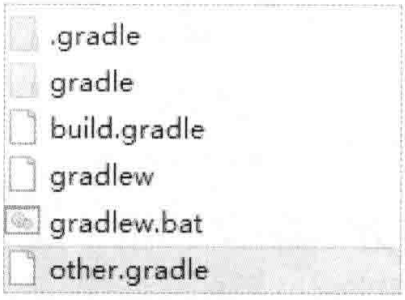


图 8-8　脚本目录结构

other.gradle 的代码如下所示。

```
ext{
verson='1.0'
url='http://liuwangshu……'
}
```

这实际上不算是一个真正的脚本插件，就是一个简单的脚本插件，主要是用于演示脚

本插件是如何被应用的。我们在 build.gradle 中应用这个脚本插件，如下所示。

build.gradle

```
apply from: 'other.gradle'
task test {
   doLast {
      println "版本为: ${verson},地址为: ${url}"
   }
}
```

apply 是 Gradle project 中提供的方法，用于配置项目中的插件。执行 gradlew.bat test，会打印出想要的结果。

8.4.2.2　对象插件

我们知道对象插件就是实现了 org.gradle.api.plugins<Project>接口的插件，对象插件可以分为内部插件和第三方插件。

1．内部插件

如果我们想要应用 Java 插件可以这样写。

build.gradle

```
apply plugin: org.gradle.api.plugins.JavaPlugin
```

Gradle 默认导入了 org.gradle.api.plugins 包，因此我们也可以去掉包名。

```
apply plugin: JavaPlugin
```

实现了 org.gradle.api.plugins 接口的插件还有 pulginid，使用 pulginid 是十分简洁且常用的方式：

```
apply plugin: 'java'
```

Gradle 的发行包中有大量的插件，这些插件有很多类型，比如语言插件、集成插件、软件开发插件等，如果我们向项目添加 C++源码编译功能，那么可以这样写。

```
apply plugin: 'cpp'
```

2．第三方插件

第三方的对象插件通常是 jar 文件，要想让构建脚本知道第三方插件的存在，需要使用 buildscrip 来设置依赖。

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle……"
    }
  }
  dependencies {
    classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:1.8.4"
  }
}
apply plugin: "com.jfrog.bintray"
```

在 buildscrip 中来定义插件所在的原始仓库和插件的依赖，再通过 apply 方法配置就可以了。Android Gradle 插件也属于第三方插件，如果我们想要引入 Android Gradle 插件，那么可以这样写。

```
buildscript {
   repositories {
      jcenter()
   }
   dependencies {
      classpath 'com.android.tools.build:gradle:2.3.2'
   }
}
apply plugin: 'com.android.application'
```

这样我们就可以使用 Android Gradle 插件通过 apply 方法来使用 App 工程插件，这样项目会编译成为一个 APK，这里涉及了 Android 相关的知识，脱离了本章的讨论范围，具体内容请见后续章节。

8.4.3 插件 DSL

Gradle 的特性有四种状态，分别是 Internal、Incubating、Public、Deprecated，插件 DSL 属于 Incubating 状态（即孵化状态）。这也导致插件 DSL 的特性在将来的 Gradle 版本中可能会发生变化，直到它不再孵化为止。

使用 Java 插件可以这样写。

build.gradle

```
plugins {
   id 'java'
}
```

这种代码很简洁，当然这是使用内部插件，如果外部插件被托管在 gradle plugin 上，那么也可以这样写。

build.gradle

```
plugins {
  id "com.jfrog.bintray" version "1.8.4"
}
```

这样就不需要再配置 buildscript 了，直接配置 plugins 来使用插件。

8.4.4　自定义对象插件

对象插件是实现了 org.gradle.api.plugins<Project>接口的插件，这个接口中只定义了一个简单的 apply 方法，想要自定义插件就需要实现 org.gradle.api.plugins<Project>接口。

实现一个简单的自定义插件，为了方便测试，不再采用文本编辑，而是使用 IntelliJ 来编辑（Android Studio 也可以），用 IntelliJ 来打开 8.4.2.1 节的示例，改写 build.gradle 文件，如下所示。

build.gradle

```
apply plugin:CustomPlugin
class CustomPlugin implements Plugin<Project> {
   @Override
   void apply(Project project) {
      project.task('CustomPluginTask') {
         doLast {
            println "自定义插件"
         }
      }
   }
}
```

在 build.gradle 中自定义了一个插件 CustomPlugin，在 apply 方法中创建一个名称为 CustomPluginTask 的任务。在 IntelliJ 的 Terminal 中输入 gradlew.bat CustomPluginTask 来执行 CustomPluginTask 任务，运行结果如图 8-9 所示。

```
Terminal:  Local ×  +
F:\app>gradlew.bat CustomPluginTask

> Task :CustomPluginTask
自定义插件

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

图 8-9　运行结果

这个示例只能在自己的项目中使用，而且比较简单，复杂的插件会在 8.5 节进行介绍。

8.4.5　Gradle 插件的作用和优点

Gradle 插件的作用主要有以下几点。

- 为项目配置依赖。
- 为项目配置约定，比如约定源码的存放位置。
- 为项目添加任务，完成测试、编译和打包等任务。
- 为项目中的核心对象和其他插件的对象添加拓展类型。

使用 Gradle 插件主要有以下优点。

- 重用和减少维护在多个项目类似的逻辑的开销。
- 更高程度的模块化。
- 封装必要的逻辑，并允许构建脚本尽可能是声明性的。

8.5　自定义 Gradle 插件

在 8.4 节中，介绍了什么是 Gradle 插件、如何使用 Gradle 插件、Gradle 插件的作用和优点，实际上还有一个重要的知识点没有提到，那就是自定义 Gradle 插件（自定义 Gradle 对象插件）。自定义 Gradle 插件主要有三种方式，分别是在 build.gradle 中编写、在 buildSrc 工程项目中编写、在独立项目中编写。

8.5.1　在 build.gradle 中编写

Groovy、Java、Kotlin 都可以作为实现插件的语言，在本节的示例中，使用 Groovy 作为实现语言。

在实际工作中我们很少会在 build.gradle 中编写自定义插件，这里是为了带领读者学习写简单的示例，可以先来了解什么是自定义插件。

1．简单的自定义插件

这里使用 IntelliJ 来编辑（Android Studio 也可以），首先新建一个 Groovy 工程，如图 8-10 所示。

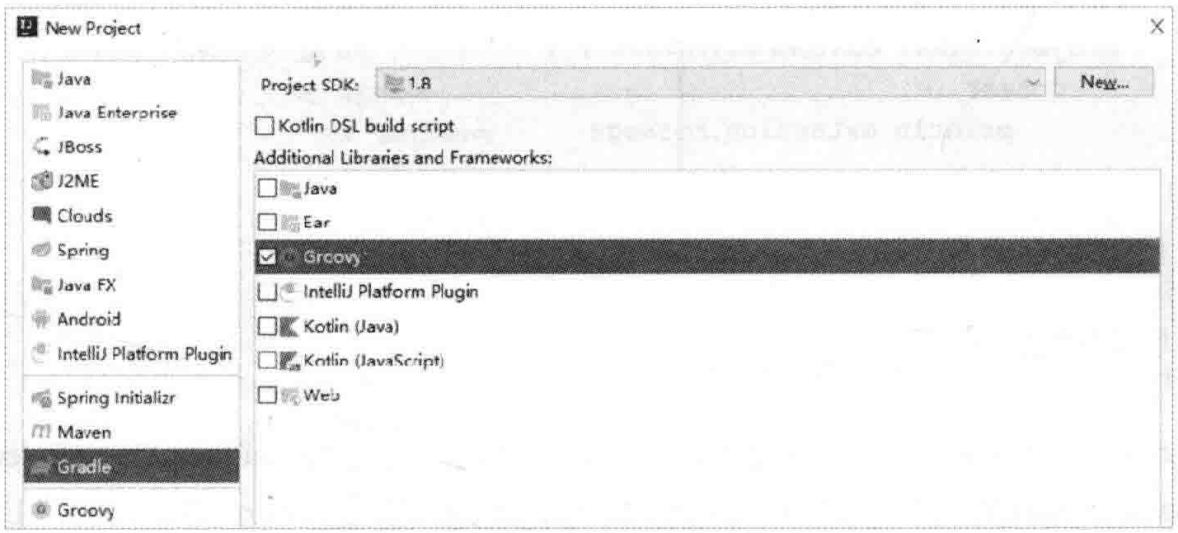


图 8-10　新建一个 Groovy 工程

定义项目的 GroupId 和 ArtifactId，如图 8-11 所示。

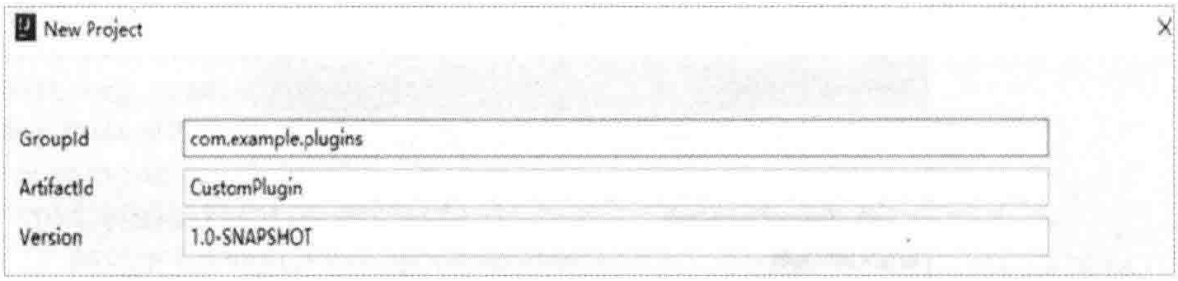


图 8-11　定义项目的 GroupId 和 ArtifactId

build.gradle 文件和运行结果见 8.4.4 节。

2. 自定义插件扩展

再举一个简单的插件拓展示例，通过插件拓展来配置 CustomPluginTask 的输出字符串，如下所示。

build.gradle

```
class CustomPluginPluginExtension {
   String message = 'from CustomPlugin'
}
class CustomPlugin implements Plugin<Project> {
   @Override
   void apply(Project project) {
      def extension = project.extensions.create('custom',
CustomPluginPluginExtension)//1
      project.task('CustomPluginTask') {
         doLast {
            println extension.message
         }
      }
   }
}
apply plugin: CustomPlugin
custom.message = "自定义插件拓展"//2
```

CustomPluginPluginExtension 类中定义了 message 变量，CustomPluginPluginExtension 是一个 Groovy Bean（类似于 JavaBean）。注释 1 处用于将拓展插件 CustomPluginPluginExtension 添加到插件列表中，其名称为 custom。注释 2 处设置 CustomPluginPluginExtension 的 message 值。

在 Terminal 中输入 gradlew.bat CustomPluginTask 来执行 CustomPluginTask 任务，运行结果如图 8-12 所示。

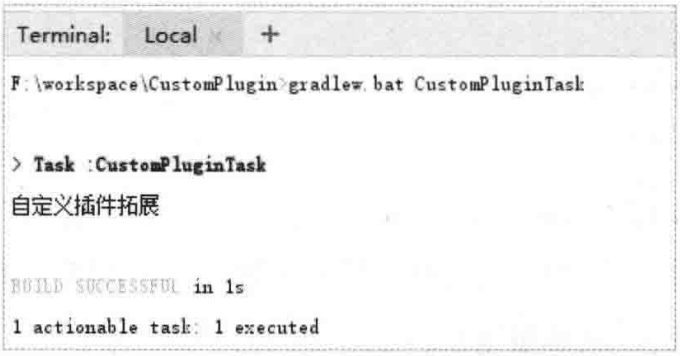


图 8-12　执行 CustomPluginTask 任务的运行结果

8.5.2　在 buildSrc 工程项目中编写

除了在 build.gradle 中编写的自定义插件，还可以将插件的源码放在 rootProjectDir/buildSrc/src/main/groovy 目录中，Gradle 会自动识别来完成编译和测试。

在工程根目录下建立/buildSrc/src/main/groovy 目录，如图 8-13 所示。

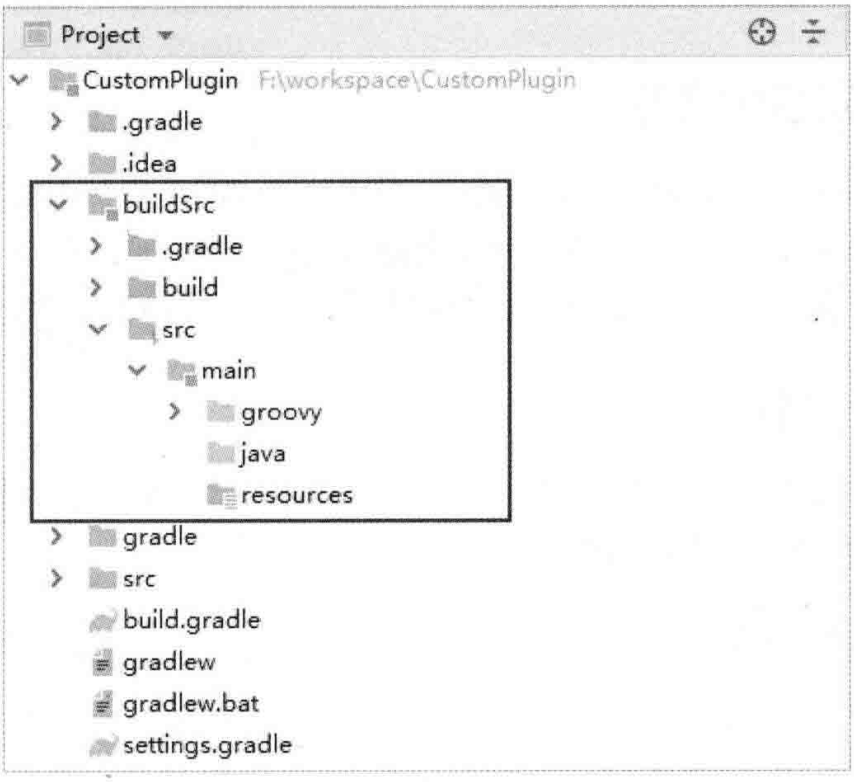


图 8-13　建立/buildSrc/src/main/groovy 目录

在 groovy 目录中创建一个 groovy 文件，如下所示。

buildSrc/src/main/groovy/CustomPlugin.groovy

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;
class CustomPlugin implements Plugin<Project> {
   @Override
   void apply(Project project) {
      project.task('CustomPluginTask') {
         doLast {
            println "自定义插件"
         }
      }
   }
}
```

修改 build.gradle 为如下内容。

build.gradle

```
apply plugin: CustomPlugin
```

在 Terminal 中输入 gradlew.bat CustomPluginTask 来执行 CustomPluginTask 任务，会打印出我们想要的结果。

8.5.3 在独立项目中编写

无论是在 build.gradle 中编写自定义插件，还是在 buildSrc 项目中编写自定义插件，都只能在自己的项目中使用。如果想要分享给其他人使用，那么可以在一个独立的项目中编写插件，这个项目会生成一个包含插件类的 JAR 文件，有了 JAR 文件就可以进行分享了。

8.5.3.1 自定义插件

为了和前两种方式进行区分，这里用 IntelliJ 新建一个 Groovy 工程，我的工程名为 CustomPluginShare，后面代码会用到。

1. 配置 build.gradle

```
apply plugin: 'groovy'

dependencies {
   compile gradleApi()
   compile localGroovy()
}
```

应用 Groovy 插件，并将 Gradle API 添加为编译时依赖项，build 工程会在 External Libraries 中生成三个 jar 文件，如图 8-14 所示。

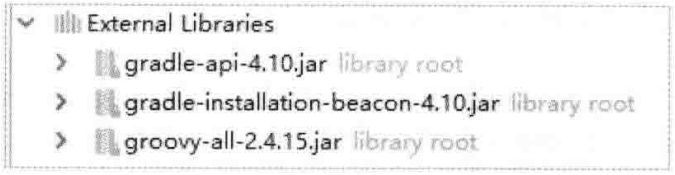


图 8-14　在 External Libraries 中生成三个 jar 文件

这样我们就可以在非 buildSrc 工程项目中使用 groovy 语法和 Gradle 的 API 了。

2. 创建自定义插件

在 src/main/groovy/ 目录中创建一个包，在 com.example.plugins 创建一个 groovy 文件，如下所示。

src/main/groovy/com/example/plugins/CustomPlugin.groovy

```
package com.example.plugins
import org.gradle.api.Plugin
import org.gradle.api.Project
class CustomPlugin implements Plugin<Project> {
   @Override
   void apply(Project project) {
      project.task('CustomPluginTask') {
         doLast {
            println "自定义插件"
         }
      }
   }
}
```

3. 配置 properties 文件

在 buildSrc 工程项目中，Gradle 可以自动识别插件，独立项目中的插件是如何被 Gradle 识别的呢？答案是需要在生成的 jar 文件中提供一个属性文件，这个属性文件名要与插件 id 相匹配。在 resources 中创建 src/main/resources/META-INF/gradle-plugins/com.example.plugins.customplugin.properties，这个属性文件的名称实际就是插件的 id，将 properties 文件的内容修改如下。

src/main/resources/META-INF/gradle-plugins/com.example.plugins.properties

```
implementation-class=com.example.plugins.CustomPlugin
```

implementation-class 属性为自定义插件的名称。

4. 上传插件

这里为了方便举例直接将插件上传到本地，如果想要发布到 Maven、ivy 等仓库或者想要发布到 Gradle 插件门户上，那么可以查看相关的文档。

在 build.gradle 文件中添加如下内容。

```
apply plugin: 'maven'
group = 'com.example.plugins'
```

```
version = '1.0.0'
uploadArchives {
   repositories {
      mavenDeployer {
         repository(url: uri('../repo'))
      }
   }
}
```

这段代码会将生成的插件上传到项目的平级目录 repo 下。这段代码定义了 group 和 version 的名称，这些值会在其他项目依赖该插件时用到。我们 Build 后会在 Gradle 窗口中看到 uploadArchives，如图 8-15 所示。

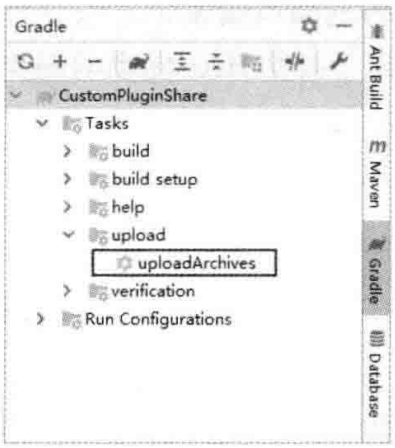


图 8-15　Gradle 窗口

单击 uploadArchives 会在本地生成插件相关的文件，如图 8-16 所示。

图 8-16　在本地生成插件相关的文件

图 8-16 中的 CustomPluginShare-1.0.0.rar 就是我们需要的插件 jar 包。

8.5.3.2　在另一个项目中使用插件

新建一个 Groovy 项目，将其命名为 Project，build.gradle 的代码如下。

```
apply plugin: 'com.example.plugins.customplugin'
```

```
buildscript {
   repositories {
      maven {
         url uri('../repo')
      }
   }
   dependencies {
      classpath 'com.example.plugins:CustomPluginShare:1.0.0'
   }
}
```

其中 com.example.plugins 是 group，CustomPluginShare 是自定义插件的名称，1.0.0 是版本号，也可以这样写。

```
  dependencies {
      classpath group: 'com.example.plugins', name: 'CustomPluginShare',
            version: '1.0.0'
   }
```

在 Terminal 中输入 gradlew.bat CustomPluginTask 来执行 CustomPluginTask 任务，就大功告成了。

如果我们将自定义插件发布到 Gradle 插件门户上，就可以使用插件 DSL 了。

build.gradle

```
plugins {
   id 'com.example.plugins.customplugin' version '1.0.0'
}
```

8.6　本章小结

本章中 Gradle 的内容尽量不与 Android 和 Android Studio 有所关联，其目的是摆脱 Android 的束缚。关于 Gradle 的内容完全可以写一本书，这里精简了一些应用开发需要掌握的内容，尤其是 Gradle 的插件这一部分，为后面章节介绍 Gradle 的 Android 插件打下了基础。

第 9 章

Gradle 的 Android 插件

关联章节：

第 7 章 Groovy 基础

第 8 章 Gradle 核心思想

Android Gradle 或者 Gradle for Android，实际上指的就是 Gradle 的 Android 插件，本章学习它的关键知识点。

9.1 什么是 Gradle 的 Android 插件

Gradle 有很多插件，为了支持 Android 项目的构建，Google 为 Gradle 编写了 Android 插件，新的 Android 构建系统就是由 Gradle 的 Android 插件组成的，Gradle 是一个高级构建工具包，它管理依赖项并允许开发者自定义构建逻辑。Android Studio 使用 Gradle wrapper 来集成 Gradle 的 Android 插件。需要注意的是，Gradle 的 Android 插件也可以独立于 Android Studio 运行。

在 Android 的官方网站中提到了新的 Android 构建系统主要有以下几个特点。

- 代码和资源易于重用。
- 无论是针对多个 APK 发行版还是针对不同风格的应用程序，都可以很容易创建应用程序的多个不同版本。
- 易于配置、扩展和自定义构建过程。
- 良好的 IDE 集成。

Gradle 的 Android 插件结合 Android Studio 成为目前十分流行的 Android 构建系统。

9.2　Android Studio 的模块类型和项目视图

Android Studio 中的每个项目包含一个或多个含有源码文件和资源文件的模块，这些模块可以独立构建、测试或调试，一个 Android Studio 的模块类型可以有以下几种。

1．Android 应用程序模块

Android 应用程序模块可能依赖于库模块，尽管许多 Android 应用程序只包含一个应用程序模块，构建系统会将其生成一个 APK。

2．Android 库模块

Android 库模块包含可重用的特定于 Android 的代码和资源，构建系统会将其生成一个 AAR。

3．App 引擎模块

App 引擎模块包含应用程序引擎集成的代码和资源。

4．Java 库模块

Java 库模块包含可重用的代码，构建系统会将其生成一个 JAR 包。Android Studio3.3.2 中的 Android 项目视图如图 9-1 所示。

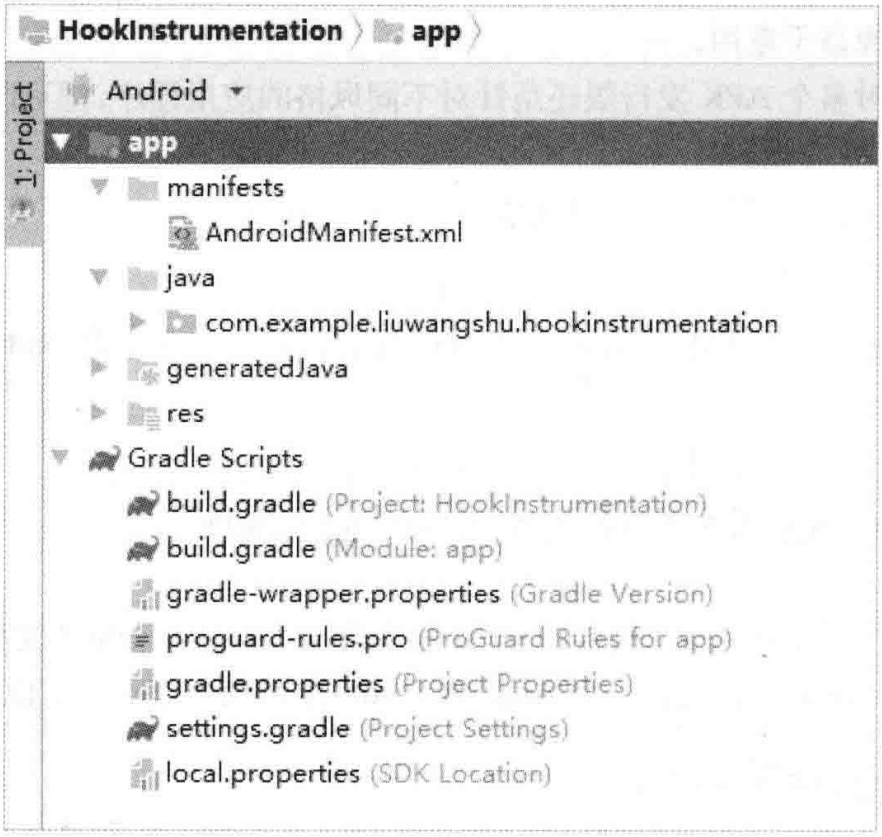


图 9-1　Android Studio3.3.2 中的 Android 项目视图

所有构建文件在 Gradle Scripts 层级下显示，这些文件的作用如下。

- 项目 build.gradle：配置项目的整体属性，比如指定使用的代码仓库、依赖的 Gradle 插件版本等。
- 模块 build.gradle：配置当前 Module 的编译参数。
- gradle-wrapper.properties：用于配置 Gradle Wrapper，具体配置可以查看 8.3 节。
- gradle.properties：配置 Gradle 的编译参数。具体配置见 Gradle 官方文档。
- settings.gradle：配置 Gradle 的多项目管理。
- local.properties：一般用来存放该 Android 项目的私有属性配置，比如 Android 项目的 SDK 路径。

本章主要介绍项目 build.gradle 和模块 build.gradle。

9.3　项目 build.gradle

新建一个 Android 项目，它的项目 build.gradle 的内容如下。

```
buildscript {
   repositories {
      google()
      jcenter()
}
   dependencies {
      classpath 'com.android.tools.build:gradle:3.3.2' //1
   }
}

allprojects {
   repositories {
      google()
      jcenter()
   }
}

task clean(type: Delete) {
   delete rootProject.buildDir
}
```

注释 1 处配置依赖的是 Gradle 插件版本，Gradle 插件属于第三方插件，因此这里在 buildscrip 块中配置 Google 的 Maven 库和 JCenter 库，这样 Gradle 系统才能找到对应的 Gradle 插件。

如果使用 google()报 not found: 'google()'错误，则可以用如下代码代替。

```
maven { url 'https://maven.google……' }
```

9.4　模块 build.gradle

新建一个 Android 项目，它的模块 build.gradle 的内容如下。

```
apply plugin: 'com.android.application'

android {
   compileSdkVersion 28
   defaultConfig {
      applicationId "com.example.myapplication"
      minSdkVersion 15
      targetSdkVersion 28
      versionCode 1
```

```
            versionName "1.0"
            testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
        }
        buildTypes {
            release {
                minifyEnabled false
                proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
            }
        }
    }

    dependencies {
        implementation fileTree(dir: 'libs', include: ['*.jar'])
        implementation 'com.android.support:appcompat-v7:28.0.0'
        implementation 'com.android.support.constraint:constraint-layout:1.1.3'
        testImplementation 'junit:junit:4.12'
        androidTestImplementation 'com.android.support.test:runner:1.0.2'
        androidTestImplementation
'com.android.support.test.espresso:espresso-core:3.0.2'
    }
```

9.4.1 Gradle 的 Android 插件类型

apply 引入的插件 id 为 com.android.application，说明当前模块是一个应用程序模块，Gradle 的 Android 插件有以下多个类型。

- 应用程序插件：插件 id 为 com.android.application，会生成一个 APK。
- 库插件：插件 id 为 com.android.library，会生成一个 AAR，用于提供给其他应用程序模块使用。
- 测试插件：插件 id 为 com.android.test，用于测试其他模块。
- feature 插件：插件 id 为 com.android.feature，是创建 Android Instant App 时需要用到的插件。
- Instant App 插件：插件 id 为 com.android.instantapp，是 Android Instant App 的入口。

9.4.2 Android 块

Android 块用于描述该 Module 构建过程中所用到的所有参数。

- compileSdkVersion：配置编译该模块的 SDK 版本。
- buildToolsVersion：Android 构建工具的版本。

9.4.2.1　defaultConfig 块

Android 块中的 defaultConfig 块用于默认配置，常用的属性如表 9-1 所示。

表 9-1　defaultConfig 块常用的属性

属　性	描　述
applicationId	指定 App 的包名
minSdkVersion	App 最低支持的 SDK 版本
targetSdkVersion	基于哪个 SDK 版本开发
versionCode	App 内部的版本号，用于控制 App 升级
versionName	App 版本名称，也就是发布的版本号
testApplicationId	配置测试 App 的包名
testInstrumentationRunner	配置单元测试使用的 Runner，默认为 android. test.InstrumentationTestRunner
proguardFile	ProGuard 混淆所使用的 ProGuard 配置文件
proguardFiles	同时配置多个 ProGuard 配置文件
signingConfig	配置默认的签名信息

9.4.2.2　buildTypes 块

buildTypes 块用于配置构建不同类型的 APK。

当我们新建一个项目时，在 Android 块中已经默认配置了 buildTypes 块。

```
buildTypes {
      release {
         minifyEnabled false
         proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
      }
   }
```

在 Android Studio 的 Terminal 中执行 gradlew.bat build 命令，会在该模块的 build/outputs/apk 目录中生成 release 和 debug 的 APK，虽然只配置了 release ，但 release 和 debug 是默认配置，即使我们不配置也会生成。我们也可以修改默认的 release 和 debug，甚至可以自定义构建类型，代码如下所示。

```
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
        }
        debug {
            debuggable true
        }
        privitedebug{
            applicationIdSuffix ""
        }
    }
```

这时会在 build/outputs/apk 目录中生成 release、debug、privitedebug 的 APK。

buildTypes 块还可以配置很多属性，常用的属性如表 9-2 所示。

表 9-2　buildTypes 块常用的属性

属　性	描　述
applicationIdSuffix	配置 applicationId 的后缀
debuggable	表示是否支持断点调试
jniDebuggable	表示是否可以调试 NDK 代码
buildConfigField	配置不同的开发环境，比如测试环境和正式环境
shrinkResources	是否自动清理未使用的资源，默认值为 false
zipAlignEnabled	是否开启 zipalign 优化，提高 APK 的运行效率
proguardFile	ProGuard 混淆所使用的 ProGuard 配置文件
proguardFiles	同时配置多个 ProGuard 配置文件
signingConfig	配置默认的签名信息
multiDexEnabled	是否启用自动拆分多个 Dex 的功能

9.4.2.3　signingConfigs 块

signingConfigs 块用于配置签名设置，一般用来配置 release 模式，signingConfigs 块常用的属性如表 9-3 所示。

表 9-3　signingConfigs 块常用的属性

属　性	描　述
storeFile	签名证书文件
storePassword	签名证书文件的密码

续表

属　性	描　述
storeType	签名证书的类型
keyAlias	签名证书中密钥的别名
keyPassword	签名证书中密钥的密码

signingConfigs 块的配置如下所示。

```
signingConfigs {
      release {
         storeFile file('C:/Users/liuwangshu/.android/release.keystore')
         storePassword 'android'
         keyAlias 'androidreleasekey'
         keyPassword 'android'

      }
```

9.4.2.4　其他配置块

android 块中除了 defaultConfig 块、buildTypes 块、signingConfigs 块还有其他配置块，这里列举一些其他配置块，如表 9-4 所示。

表 9-4　其他配置块

块	描　述
sourceSets	配置目录指向
productFlavors	多个渠道配置
lintOptions	Lint 配置
dexOptions	DEX 工具配置
adbOptions	adb 配置
packagingOptions	打包时的相关配置

更多配置块请参考 Gradle 官方文档。

9.4.2.5　全局配置

如果多个 module 的配置是一样的，可以将这些配置提取出来，也就是使用全局配置。全局配置有多种方式，这里介绍其中的两种。

1. 使用 ext 块配置

在项目 build.gradle 中使用 ext 块，如下所示。

```
ext{
   compileSdkVersion =28
   buildToolsVersion ="28.0.3"
   minSdkVersion =15
   targetSdkVersion =28
}
```

在某个 module 的 build.gradle 中使用配置，如下所示。

```
apply plugin: 'com.android.application'
android {
   compileSdkVersion rootProject.ext.compileSdkVersion
   buildToolsVersion rootProject.ext.buildToolsVersion
   defaultConfig {
      applicationId "com.example.liuwangshu.hookinstrumentation"
      minSdkVersion rootProject.ext.minSdkVersion
      targetSdkVersion rootProject.ext.targetSdkVersion
      versionCode 1
      versionName "1.0"
      testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
   }
  ...
}
...
```

2. 使用 config.gradle 配置

首先在根目录下创建 config.gradle 文件来进行配置，如下所示。

config.gradle

```
ext{
   android=[
         applicationId:"com.example.liuwangshu.hookinstrumentation",
         compileSdkVersion :28,
         buildToolsVersion :"28.0.3",
         minSdkVersion : 15,
         targetSdkVersion : 28,
   ]

   dependencies =[
```

```
            "appcompat-v7" : "com.android.support:appcompat-v7:28.0.0",
            "constraint"  :
"com.android.support.constraint:constraint-layout:1.1.3",
        ]
    }
```

然后在项目 build.gradle 中添加 apply from: "config.gradle"，这样项目的所有 module 都能用 config.gradle 中定义的参数。

最后在 module 的 build.gradle 中使用配置，如下所示。

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion rootProject.ext.android.compileSdkVersion
    buildToolsVersion rootProject.ext.android.buildToolsVersion
    defaultConfig {
        applicationId rootProject.ext.android.applicationId
        minSdkVersion rootProject.ext.android.minSdkVersion
        targetSdkVersion rootProject.ext.android.targetSdkVersion
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }
    ...
dependencies {
    implementation rootProject.ext.dependencies["constraint"]
    implementation rootProject.ext.dependencies["appcompat-v7"]
    ...
}
```

9.4.3　dependencies 块

dependencies 块用于配置该 module 构建过程中所依赖的所有库。Gradle 插件 3.4 版本新增了 API 和 implementation 来代替 compile 配置依赖，其中 API 与 compile 是一样的。dependencies 和 API 主要有以下区别。

- implementation 可以让 module 在编译时隐藏自己使用的依赖，但是在运行时这个依赖对所有模块是可见的，而 API 与 compile 一样，无法隐藏自己使用的依赖。
- 使用 API，如果一个 module 发生变化，那么这条依赖链上的所有 module 都需要重新编译，而使用 implemention，只有直接依赖这个 module 需要重新编译。

9.5 Android 签名文件配置

在一般公司中，当团队比较小的时候，App 的签名信息都是放到项目中的，可能还会上传到 GitHub 上，这样做很方便。但随着团队人数的增多，这样做的风险会越来越大，因为签名信息是重要的资源，这样就不能将签名上传到 GitHub 上了，也就不应该在 build.gradle 中直接配置签名。

针对这个问题，主要有以下两种解决方法。

（1）自定义一个签名配置文件。

（2）在本地~/.gradle/gradle.properties 文件中配置签名信息。

9.5.1 自定义签名信息文件

首先在工程的目录下新建一个文件夹，内部存储签名文件和签名信息文件。签名文件为 gradledemo.jks，签名信息文件为 keystore.properties。keystore.properties 中的配置如下所示。

```
STORE_FILE=../signfiles/gradledemo.jks
KEY_ALIAS=gradle
STORE_PASSWORD=jinjiesanbuqu
KEY_PASSWORD=jinjiesanbuqu
```

在.gitignore 中将 gradledemo.jks 和 keystore.properties 忽略掉。然后在模块 build.gradle 中进行配置，代码如下所示。

```
apply plugin: 'com.android.application'
android {
 ...
}
def setSigningProperties(){
   def propFile = file('../signfiles/keystore.properties')
   if (propFile.canRead()){
      def Properties props = new Properties()
      props.load(new FileInputStream(propFile))
      if (props!=null && props.containsKey('STORE_FILE') &&
props.containsKey('STORE_PASSWORD') &&props.containsKey('KEY_ALIAS') &&
props.containsKey('KEY_PASSWORD')) {
         android.signingConfigs.release.storeFile = file(props['STORE_FILE'])
         android.signingConfigs.release.storePassword = props['STORE_PASSWORD']
         android.signingConfigs.release.keyAlias = props['KEY_ALIAS']
```

```
            android.signingConfigs.release.keyPassword = props['KEY_PASSWORD']
        } else {
            throw new Exception("some key missing")
        }
    }else {
        throw new Exception("keystore.properties not found:" +
propFile.absolutePath)

    }
}
```

setSigningProperties 方法用于读取 keystore.properties 文件中的签名文件的信息。

最后在模块 build.gradle 中的 signingconfigs 块中调用 setSigningProperties 方法即可，代码如下所示。

```
apply plugin: 'com.android.application'

android {
 ...
   signingConfigs {
      release {
         setSigningProperties()
      }
   }
}
```

9.5.2　本地添加签名信息文件

还可以将签名文件和签名信息文件放到本地。比如将签名文件放到~/.gradle/gradledemo. jks，签名信息文件放到~/.gradle/keystore.properties。这样签名文件和签名信息文件都不会提交到 GitHub 上。

keystore.properties 的内容如下所示。

```
GRADLEDOME_RELEASE_STORE_FILE=~/.gradle/release-key.keystore
GRADLEDOM_RELEASE_KEY_ALIAS=key-alias
GRADLEDOM_RELEASE_STORE_PASSWORD=pass
GRADLEDOM_RELEASE_KEY_PASSWORD=pass
```

在模块 build.gradle 中的 signingconfigs 块中配置签名，如下所示。

```
signingConfigs {
      release {
         storeFile file(GRADLEDOME_RELEASE_STORE_FILE)
```

```
            storePassword GRADLEDOME_RELEASE_STORE_PASSWORD
            keyAlias GRADLEDOME_RELEASE_KEY_ALIAS
            keyPassword GRADLEDOME_RELEASE_KEY_PASSWORD
        }
    }
```

除了这两种方法，还可以将签名文件和签名信息文件放在专门打包的服务器上，在打包的时候读取即可。

9.6 Gradle 的库依赖

现在一个 Android 项目都需要引入其他库，比如 jar、aar 和 Module 等，现在分别对其进行介绍。如果不特意说明，则下面示例的代码均是写在模块 build.gradle 中的。

1. Gradle 的本地库依赖

关于 jar 依赖可以按照如下这种形式写，可以指定一个 jar 也可以指定多个 jar。

```
//依赖引入 libs 下的所有 jar
implementation fileTree(dir:'libs',include:['*.jar'])

//指定依赖某一个或几个 jar
implementation files('libs/XXX.jar','libs/XXX.jar')
```

aar 依赖需要额外增加一些语句，如下所示。

```
android {
    ...
    repositories {
        flatDir {
            dirs "libs"
        }
    }
}
dependencies {
implementation fileTree(dir:'libs',include:['*.aar'])
implementation(name:'XXX',ext:'aar')
}
```

2. Gradle 的本地 Module 依赖

当项目中有多个 Module 时，我们需要先在 settings.gradle 中引入 Module，如下所示。

```
include ':app'
include ':library1', ':library2'
```

然后在模块 build.gradle 中引入。

```
implementation project(':library1')
```

3. Gradle 的远程库依赖

当在 Android Studio 中新建一个项目时，会在项目 build.gradle 中有如下代码。

```
buildscript {
   repositories {
      google()
      jcenter()

   }
   dependencies {
      classpath 'com.android.tools.build:gradle:3.4.0'
   }
}

allprojects {
   repositories {
      google()
      jcenter()

   }
}
```

这些代码都是默认的，在 buildscript 块和 allprojects 块中，通过 repositories 来引入 Google 的 Maven 库和 JCenter 库。首先会从 Google 的 Maven 库去寻找相应的库，如果找不到就会到 JCenter 库中去寻找，然后在模块 build.gradle 中加入如下代码，就可以引入远程库。

```
implementation group:'com.android.support',name:'appcompat-v7',version:'28.0.0'
//简写
implementation 'com.android.support:appcompat-v7:28.0.0'
```

9.7　Gradle 的库依赖管理

随着 Gradle 依赖的库越来越多，就会产生一些问题，比如依赖冲突，为了解决依赖冲突，我们首先需要了解 Gradle 的库依赖管理的几个技术点。

9.7.1 Gradle 的依赖传递

Gradle 默认是支持依赖传递的，所以当用到 Gradle 依赖时一定会涉及依赖传递，这是必须要知道的一个知识点。那么什么是依赖传递呢？这里举一个简单的示例。

projectC 依赖 projectB，projectB 依赖 projectA，那么 projectC 就依赖 projectA。

依赖传递会产生一些问题，比如重复依赖、依赖错误等，我们可以通过 transitive 来禁止依赖传递，如下所示。

```
implementation('com.xxx.xxx:xxx:3.6.3') {
    transitive false
 }
```

上述代码禁止了 com.xxx.xxx:xxx:3.6.3 库的依赖传递，还可以使用如下语句来关闭当前模块的所有库的依赖传递。

```
configurations.all {
  transitive = false
}
```

只是这样就需要手动添加当前模块的每个库的依赖项，一般不会这么做。

9.7.2 Gradle 的依赖检查

有了依赖检查，我们可以解决依赖产生的问题。依赖检查有很多种方式，下面分别来介绍这几种方式。

1. 使用 Gradle 的命令行

可以直接使用 Gradle 的命令行来进行依赖检查，以 Windows 平台为例，使用 cmd 进入项目的根目录，执行 gradle :app:dependencies 即可，其中 app 是我们新建工程时默认的模块的名称。日志输出很多，下面截取其中的一部分。

```
+--- com.android.support:appcompat-v7:28.0.0
|    +--- com.android.support:support-annotations:28.0.0 //1
|    +--- com.android.support:support-compat:28.0.0 //2
|    |    +--- com.android.support:support-annotations:28.0.0
|    |    +--- com.android.support:collections:28.0.0
|    |    |    \--- com.android.support:support-annotations:28.0.0
|    |    +--- android.arch.lifecycle:runtime:1.1.1
|    |    |    +--- android.arch.lifecycle:common:1.1.1
|    |    |    |    \--- com.android.support:support-annotations:26.1.0 -> 28.0.0 //3
```

```
|    |    |    +--- android.arch.core:common:1.1.1
|    |    |    |    \--- com.android.support:support-annotations:26.1.0 -> 28.0.0
|    |    |    \--- com.android.support:support-annotations:26.1.0 -> 28.0.0
|    |    \--- com.android.support:versionedparcelable:28.0.0
|    |         +--- com.android.support:support-annotations:28.0.0
|    |         \--- com.android.support:collections:28.0.0 (*)
```

上面是 appcompat-v7:28.0.0 库的依赖树的一部分，appcompat-v7:28.0.0 依赖了注释 1 处和注释 2 处的库，注释 2 处的库又依赖了 com.android.support:support-annotations:28.0.0 和 com.android.support:collections:28.0.0，因此当我们引入 appcompat-v7:28.0.0 时，会自动下载所有它依赖传递的库。注释 3 处说明 Gradle 在依赖传递时，会自动提升依赖传递的库的版本，默认使用最高版本的库。

2. 使用 Gradle 面板

除了命令行，还可以使用 Android Studio 中的右侧的 Gradle 面板，找到 app 模块，将其展开后找到 help 目录中的 dependencies，如图 9-2 所示。

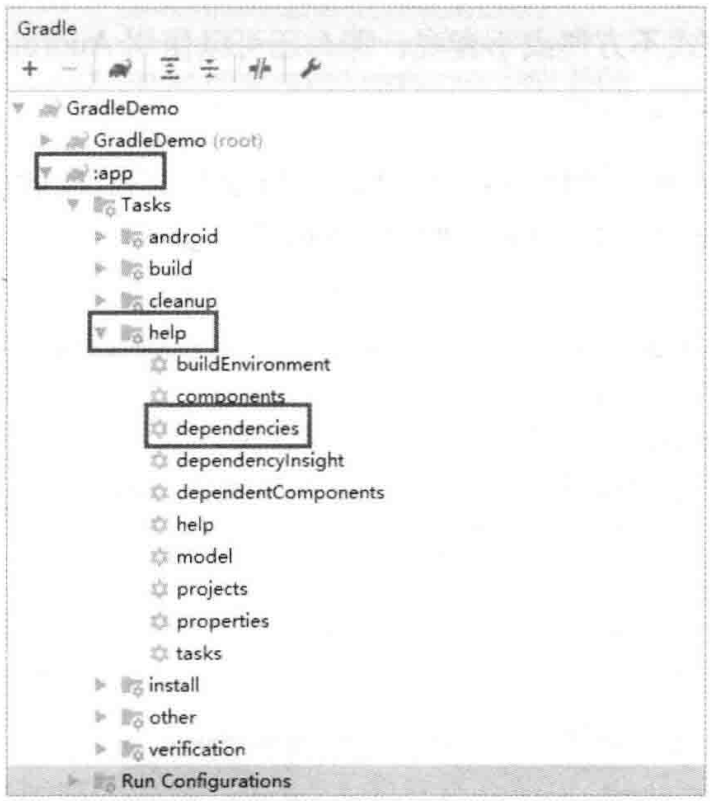


图 9-2　Gradle 面板视图

双击或者在右键菜单中选择第一个选项即可执行命令，日志就会在 Android Studio 中的 Run 窗口中打印出来。现在以我们熟悉的 retrofit 为例，在模块 build.gradle 中引入 retrofit。

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
```

执行依赖检查命令，打印关于 retrofit 的日志如下。

```
+--- com.squareup.retrofit2:retrofit:2.6.0
|    \--- com.squareup.okhttp3:okhttp:3.12.0
|         \--- com.squareup.okio:okio:1.15.0
```

可以很清楚地看到 retrofit:2.6.0 依赖 okhttp:3.12.0，okhttp:3.12.0 依赖 okio:1.15.0。

这时我们使用 transitive 试着修改 build.gradle。

```
implementation ('com.squareup.retrofit2:retrofit:2.6.0') {
    transitive false
}
```

执行依赖检查命令，打印关于 retrofit 的日志如下。

```
+--- com.squareup.retrofit2:retrofit:2.6.0
```

3. 使用 Gradle View 插件

如果觉得前两种方式不方便或不直观，那么还可以使用 Android Studio 的 Gradle View 插件。

在 Android Studio 中依次选择 File→Settings→Plugins，然后搜索 gradle view，找到 Gradle View 插件安装并重启 Android Studio，如图 9-3 所示。

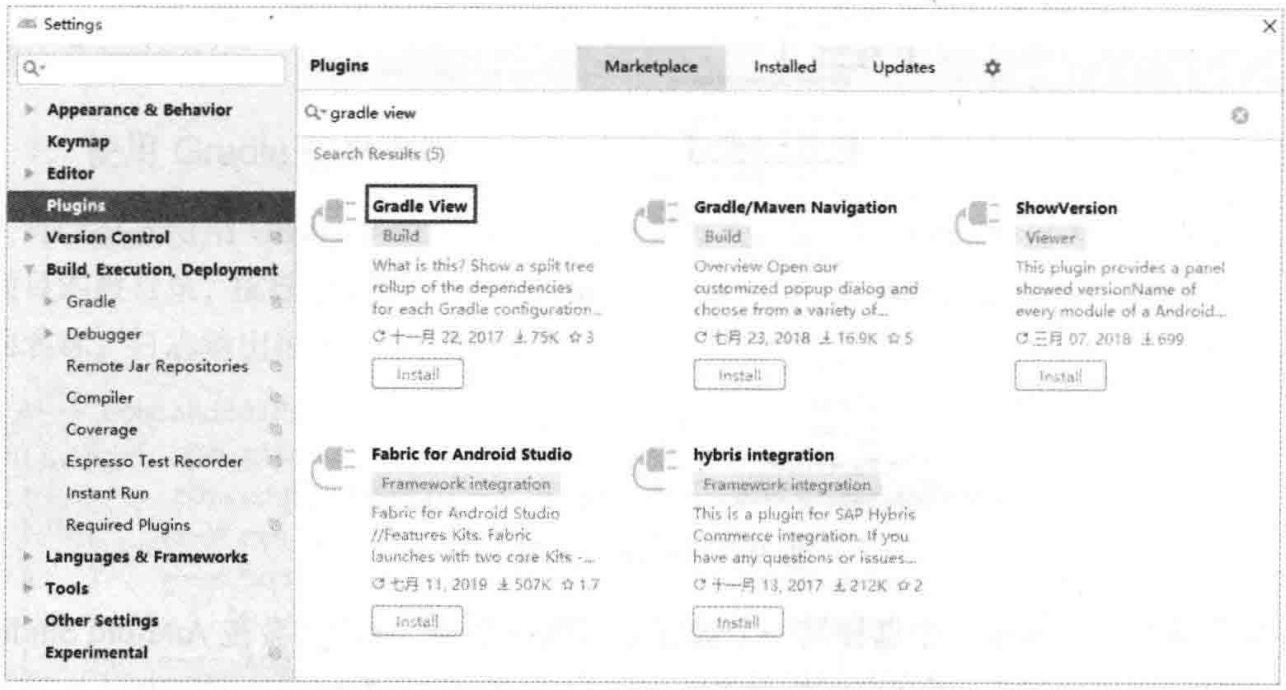


图 9-3　安装 gradle view 插件

接下来依次选择 View→Tools Windows→Gradle View，这时就可以在 Android Studio 的底部发现 Gradle View 窗口，里面会显示当前项目的所有依赖树，如图 9-4 所示。

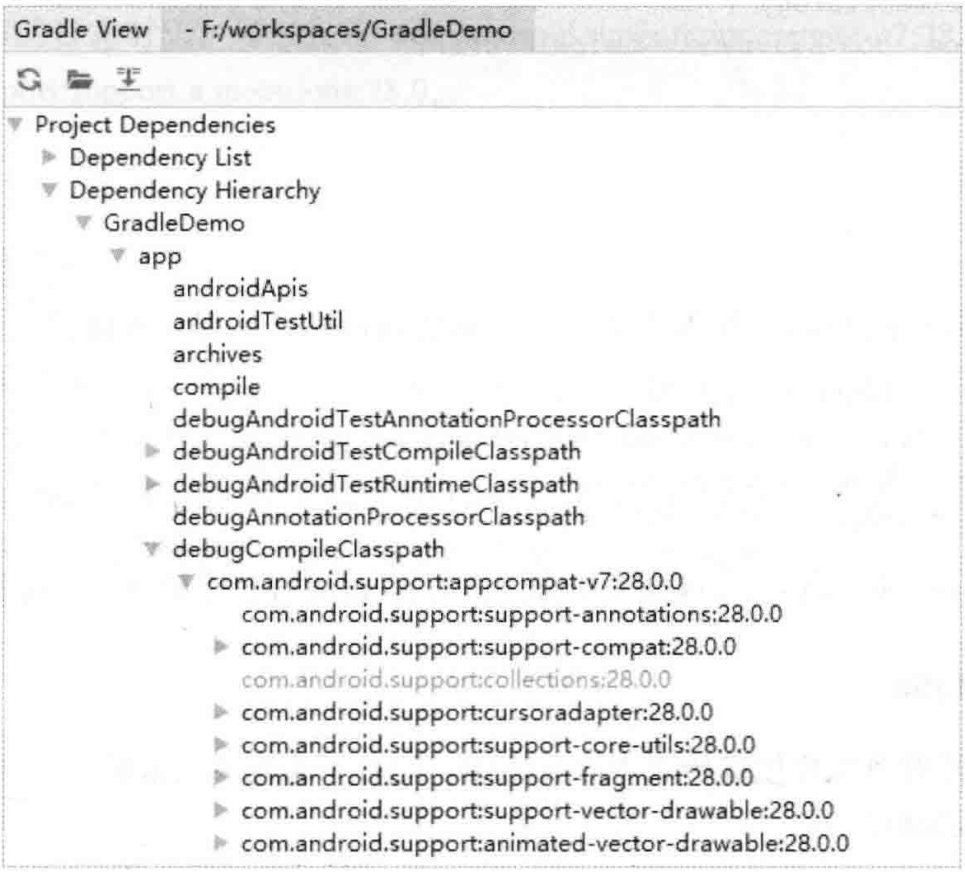


图 9-4　当前项目的所有依赖树

9.7.3　Gradle 的依赖冲突

依赖冲突产生的原因大多是库的版本问题，比如，在 build.gradle 中这样写。

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation 'com.squareup.okio:okio:1.14.0'
```

在 9.7.2 节中，我们知道 retrofit:2.6.0 依赖的 okio 的版本是 1.15.0，而这里引入的 okio 的版本为 1.14.0，引入的版本不同就会产生依赖冲突。依赖冲突的解决关键有两点，一个是 Gradle 的依赖检查，这个在 9.7.2 节已经讲过了；另一个是利用 Gradle 的关键字，合理利用它们是解决依赖冲突的关键，在 9.7.1 节已经介绍了 transitive，现在介绍其余的关键字。

9.7.3.1　force

有时候我们不是想要排除某个库，而是需要强制使用统一的库的版本，force 可以强制

设置模块的库的版本，在模块 build.gradle 中加入如下代码。

```
configurations.all {
   resolutionStrategy {
      force 'com.squareup.okio:okio:2.1.0'
   }
}
dependencies {
...
}
```

强制当前模块的 okio 的版本为 2.1.0，使用依赖检查来查看 retrofit 的依赖如下所示。

```
+--- com.squareup.retrofit2:retrofit:2.6.0
|    \--- com.squareup.okhttp3:okhttp:3.12.0
|         \--- com.squareup.okio:okio:1.15.0 -> 2.1.0
\--- com.squareup.okio:okio:1.14.0 -> 2.1.0
```

可以看到 okio 的版本都被强制升级到了 2.1.0，这样就可以解决一些依赖冲突的问题。

9.7.3.2 exclude

有些时候需要排除库依赖传递中涉及的库，此时不能靠关闭依赖传递来解决问题，这时可以使用 exclude。

我们知道 com.android.support:appcompat-v7:28.0.0 依赖于 com.android.support:support-annotations:28.0.0、com.android.support:support-compat:28.0.0 和 com.android.support:cursoradapter:28.0.0 等库，这时我们不想再依赖 support-annotations 库，如下所示。

```
configurations {
   all*.exclude group: 'com.android.support', module: 'support-annotations'
}
dependencies {
...
}
```

使用依赖检查来查看 com.android.support:appcompat-v7:28.0.0 的依赖，如下所示。

```
+--- com.android.support:appcompat-v7:28.0.0
|    +--- com.android.support:support-compat:28.0.0
|    |    +--- com.android.support:collections:28.0.0
|    |    +--- android.arch.lifecycle:runtime:1.1.1
|    |    |    +--- android.arch.lifecycle:common:1.1.1
|    |    |    \--- android.arch.core:common:1.1.1
|    |    \--- com.android.support:versionedparcelable:28.0.0
```

```
|    |        \--- com.android.support:collections:28.0.0
|    +--- com.android.support:collections:28.0.0
|    +--- com.android.support:cursoradapter:28.0.0
```

和 9.7.2 节的日志对比，可以发现 com.android.support:appcompat-v7:28.0.0 不再依赖 com.android.support:support-annotations:28.0。

9.8　本章小结

Gradle 的 Android 插件的知识点很多也很琐碎，本章在第 8 章的基础上讲解了 Gradle 的 Android 插件的常用的知识点，包括 Android Studio 的模块类型和项目视图、build.gradle、Android 签名文件配置、Gradle 的库依赖和 Gradle 的库依赖管理等，基本覆盖了应用开发需要了解的知识点，如果读者想要了解更多请查看官方文档或者相关图书。

第 10 章

Android Jetpack 架构组件

关联章节：

第 8 章 Gradle 核心思想

《Android 进阶之光》第 6 章 设计模式

《Android 进阶之光》第 10 章 应用架构设计

Android 发展至今，可以说是比较成熟的技术了，刚开始时框架很少，也没有什么规范，所有代码都要自己写，比如网络请求、数据库请求和数据解析等。后来出现了一些框架来帮助开发者进行快速开发，比如 XUtils、Volley、OKHttp 和 EventBus 等，随着框架越来越多，一个应用可以有多种技术选型，直接导致应用开发越来越不规范，做出来的应用质量参差不齐，这显然不是 Google 想看到的。Google 随后推出了 MVP 和 MVVM 相关的官方示例，其效果很一般，而且覆盖面只是在架构上，Google 推出的 Android Jetpack 有望解决以上问题。

Android Jetpack 架构组件所包含的组件比较多，由于篇幅的原因，本章只介绍比较常用的 Lifecycle、LiveData、ViewModel 的使用方法和原理。

10.1　Android Jetpack 简介

Jetpack 并不是一个框架或者组件，它是一套库、工具和指南的集合，可以帮助开发者更轻松地编写优质应用。这些组件可帮助开发者遵循最佳做法、摆脱编写样板代码的工作并简化复杂任务，以便开发者能够将精力集中放在所需的代码上。

Android Jetpack 中的很多组件并不都是新开发的，而是很早就存在的，比如在 Google I/O 2017 大会上推出了 Android Architecture Component（AAC），它包括了 LifeCycle、LiveData、ViewModel、Room。在 Google I/O 2018 大会上，用 AndroidX 替代了 Android Support Library，并在 Android Architecture Component 的基础上发布了 Android Jetpack，AndroidX 也属于 Android Jetpack。

Jetpack 有以下三个主要特性。

（1）加速开发。

组件可以单独使用，也可以协同工作，当使用 kotlin 语言特性时，可以提高效率。

（2）消除样板代码。

Android Jetpack 可管理复杂的 Activity（如后台任务、导航和生命周期管理）。

（3）构建高质量的强大应用。

Android Jetpack 组件围绕现代化设计实践构建而成，具有向后兼容性，可以减少崩溃和内存泄漏。

10.2　Android Jetpack 分类

Android Jetpack 的分类如图 10-1 所示。

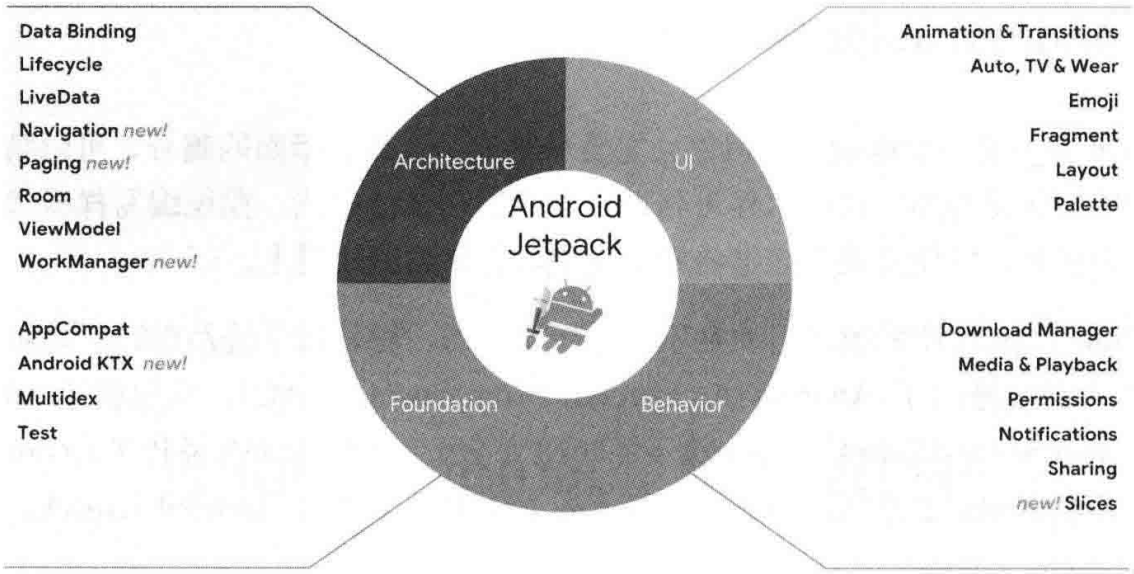


图 10-1　Android Jetpack 的分类

Google 仍旧不断完善 Android Jetpack，截至 2019 年 8 月已经有很多新加的组件，但是它们并没有显示在 10-1 图中，比如 CameraX 等。Android Jetpack 分类有 4 种，分别是 Architecture（架构组件）、Foundationy（基础组件）、Behavior（行为组件）、UI（界面组件），下面分别对其进行介绍。

10.2.1　Architecture

Architecture 能够帮助开发者设计稳健、可测试且易维护的应用。Architecture 可以说是对应用开发帮助很大的组件，本节也是围绕着 Architecture 进行讲解的。

Data Binding：以声明的方式将可观察数据绑定到界面元素，通常和 ViewModel 配合使用。

Lifecycle：用于管理 Activity 和 Fragment 的生命周期，可以帮助开发者生成更易于维护的轻量级代码。

LiveData：在底层数据库更改时通知视图。它是一个可观察的数据持有者，与常规 observable 不同，LiveData 是有生命周期感知的。

Navigation：用于处理应用内导航。

Paging：可以帮助开发者一次加载和显示小块数据，按需加载部分数据可减少网络带宽和系统资源的使用。

Room：友好、流畅的访问 SQLite 数据库。它在 SQLite 的基础上提供了一个抽象层，

允许更强大的数据库访问。

ViewModel：以生命周期的方式管理与界面相关的数据，通常和 Data Binding 配合使用，为开发者实现 MVVM 架构提供了强有力的支持。

WorkManager：管理 Android 的后台作业，即使应用程序退出或设备重新启动也可以运行可延迟的异步任务。

官方推荐的 Android 应用架构如图 10-2 所示。

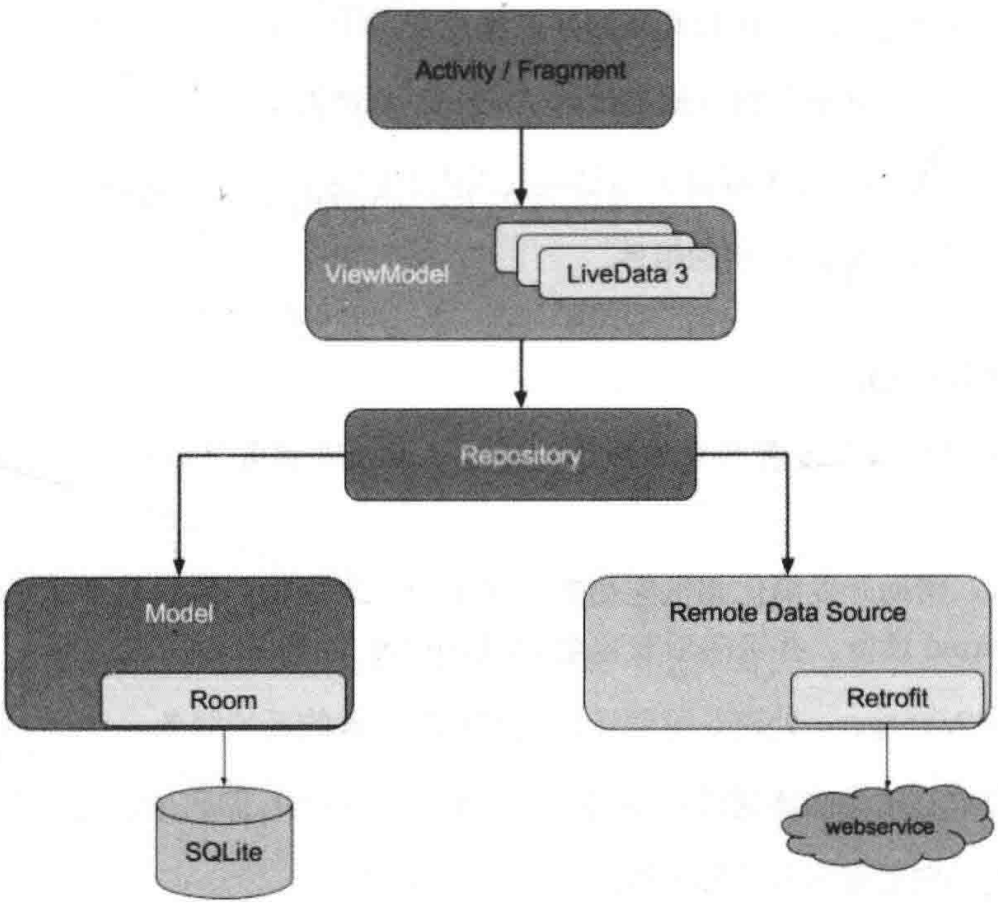


图 10-2　官方推荐的 Android 应用架构

这些架构组件既可以配合使用又可以单独使用，建议还是尽量按照官方推荐的来进行操作。

10.2.2　Foundationy

Foundationy 提供横向功能，例如向后兼容性、测试、安全、Kotlin 语言支持，并包括多个平台开发的组件。

Android KTX：优化了供 Kotlin 使用的 Jetpack 和 Android 平台 API。帮助开发者以更简洁、更愉悦、更惯用的方式使用 Kotlin 进行 Android 开发。

AppCompat：帮助较低版本的 Android 系统进行兼容。

Auto：开发 Android Auto 应用的组件，提供了适用于所有车辆的标准化界面和用户交互。

检测：从 Android Studio 中快速检测基于 Kotlin 或 Java 的代码。

多 Dex 处理：为具有多个 Dex 文件应用提供支持。

安全：安全读写加密文件和共享偏好设置。

测试：用于单元和运行时界面测试的 Android 测试框架。

TV：构建可使用户在大屏幕上体验沉浸式内容的应用。

Wear OS：开发 Wear 应用的组件。

10.2.3 Behavior

Behavior 可帮助开发者开发的应用与标准 Android 服务（如通知、权限、分享）相集成。

CameraX：帮助开发简化相机应用的开发工作。它提供一致且易于使用的界面，适用于大多数 Android 设备，并可向后兼容至 Android 5.0（API 21）。

Download Manager：处理长时间运行的 HTTP 下载的系统服务。

媒体和播放：用于媒体播放和路由（包括 Google Cast）的向后兼容 API。

通知：提供向后兼容的通知 API，支持 Wear 和 Auto。

权限：用于检查和请求应用权限的兼容性 API。

设置：创建交互式设置，建议使用 AndroidX Preference Library 库将用户可配置设置集成到应用中。

分享操作：可以更轻松地实现友好的用户分享操作。

切片：切片是一种 UI 模板，创建可在应用外部显示应用数据的灵活界面元素。

10.2.4　UI

Animation & Transitions：该框架包含用于常见效果的内置动画，并允许开发者创建自定义动画和生命周期回调。

Emoji Compatibility：即便用户没有更新 Android 系统也可以获取最新的表情符号。

Fragment：组件化界面的基本单位。

布局：在 XML 中声明 UI 元素或者在代码中实例化 UI 元素。

调色板：从调色板中提取出有用的信息。

10.3　在应用中引入 Jetpack 相关组件

在项目 build.gradle 中添加 google() 库，如下所示。

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

如果想引入 Room，那么可以在模块 build.gradle 中这样写。

```
implementation  "android.arch.persistence.room:runtime:1.1.1"
annotationProcessor  "android.arch.persistence.room:compiler:1.1.1 "
androidTestImplementation "android.arch.persistence.room:testing:1.1.1"
```

10.4　使用 Lifecycle

Lifecycle 用于帮助开发者管理 Activity 和 Fragment 的生命周期，由于 Lifecycle 是 LiveData 和 ViewModel 的基础，所以需要先学习 Lifecycle。

10.4.1　为什么需要 Lifecycle

在应用开发中，处理 Activity 或者 Fragment 组件的生命周期相关代码是必不可少的，

官方文档中列举了一个示例，这里将其简化，在 Activity 中写一个监听，在 Activity 的不同生命周期方法中调用这个监听，如下所示。

```
public class MainActivity extends AppCompatActivity {
   private MyListener myListener;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      myListener = new MyListener(MainActivity.this);
   }

   @Override
   protected void onStart() {
      super.onStart();
      myListener.start();
   }

   @Override
   protected void onStop() {
      super.onStop();
      myListener.stop();
   }
}

class MyListener {
   public MyListener(Context context) {
   ...
   }
   void start() {
   ...
   }
   void stop() {
   ...
   }
}
```

再列举个 MVP 中常见的示例，如下所示。

```
public class MainActivity extends AppCompatActivity {
   private MyPresenter myPresenter;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);
        myPresenter = new MyPresenter();
    }
    @Override
    protected void onResume() {
        super.onResume();
        myPresenter.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
        myPresenter.onPause();
    }
}

class MyPresenter{
    void onResume() {
    ...
    }
    void onPause() {
    ...
    }
}
```

这两个示例的写法很普遍，实现起来也不难，但在实际开发中，可能会有多个组件在 Activity 的生命周期中进行回调，这样 Activity 的生命周期的方法中可能就需要放大量的代码，这使得它们难以维护。

如果我们在组件中做了耗时操作（比如 onStart 方法），那么这种写法就无法保证组件在 Activity 或者 Fragment 停止之前完成启动。因此我们需要一个能够管理 Activity 和 Fragment 的生命周期的库，这个库就是 Lifecycle。

10.4.2　如何使用 Lifecycle

下面分别来介绍依赖 Lifecycle 库和 Lifecycle 基本用法。

1．依赖 Lifecycle 库

官方网站给出的依赖代码如下所示。

```
dependencies {
    def lifecycle_version = "2.0.0"
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_version"
```

```
        implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
        implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
        implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"
        annotationProcessor
"androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
        implementation
"androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"
        implementation
"androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"
        testImplementation "androidx.arch.core:core-testing:$lifecycle_version"
    }
```

官方网站用的是 AndroidX，这里为了大家能更快掌握，使用的是 lifecycleandroid.arch.lifecycle 库，如下所示。

```
    dependencies {
        def lifecycle_version = "1.1.1"

        // 包含 ViewModel 和 LiveData
        implementation "android.arch.lifecycle:extensions:$lifecycle_version"
        // 仅包含 ViewModel
        implementation "android.arch.lifecycle:viewmodel:$lifecycle_version"
        // 仅包含 LiveData
        implementation "android.arch.lifecycle:livedata:$lifecycle_version"
        // 仅包含 Lifecycles
        implementation "android.arch.lifecycle:runtime:$lifecycle_version"
        annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"
        // 如果用 Java8，则它可用于替代 compiler
        implementation "android.arch.lifecycle:common-java8:$lifecycle_version"

        // 可选，ReactiveStreams 对 LiveData 的支持
        implementation "android.arch.lifecycle:reactivestreams:$lifecycle_version"

        // 可选，LiveData 的测试
        testImplementation "android.arch.core:core-testing:$lifecycle_version"
    }
```

实际上不需要把这些代码全部写进 build.gradle 中，当然全部写进去也不会报错，Gradle 默认是支持依赖传递的。

我们直接添加如下依赖就可以满足日常的工作，如果缺少哪个库，则去单独添加即可。

```
implementation "android.arch.lifecycle:extensions:1.1.1"
```

添加这一句代码就依赖了如图 10-3 中的库。

图 10-3　依赖的库

2. Lifecycle 基本用法

暂且不讲在 Activity 和 Fragment 中如何使用 Lifecycle，先举一个 Lifecycle 的简单示例，如下所示。

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void connectListener() {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void disconnectListener() {
        ...
    }
}

myLifecycleOwner.getLifecycle().addObserver(new MyObserver());//1
```

新建一个 MyObserver 类，它实现了 LifecycleObserver 接口，说明 MyObserver 成为一个 Lifecycle 的观察者。

然后在注释 1 处将 MyObserver 添加到 LifecycleOwner 中。LifecycleOwner 是一个接口，其内部只有一个 getLifecycle 方法，getLifecycle 方法用于获取 Lifecycle，这样就可以将 MyObserver 添加到 Lifecycle 中，当 Lifecycle 的生命周期发生变化时，MyObserver 就能够观察到。

如果使用 Java8，那么可以使用 DefaultLifecycleObserver 来替代 LifecycleObserver，如下所示。

```
class MyObserver implements DefaultLifecycleObserver {
    @Override
    public void onCreate(LifecycleOwner owner) {
```

```
        ...
    }
}
```

除此之外，要在 build.gradle 中添加"androidx.lifecycle:common-java8:<version>"

10.4.3 Lifecycle 应用举例

这里准备了两个示例，一个是在 Activity 中使用的，另一个是在 10.4.1 节的 MVP 示例上进行改进的。

1. 在 Activity 中使用

```
package com.example.lifecycledemo1;

import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleObserver;
import android.arch.lifecycle.OnLifecycleEvent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getLifecycle().addObserver(new MyObserver());//1
    }

    public class MyObserver implements LifecycleObserver{

        @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
        void onResume(){
            Log.d(TAG, "Lifecycle call onResume");
        }
        @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
        void onPause(){
            Log.d(TAG, "Lifecycle call onPause");
        }
```

```
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");

    }
}
```

先实现 MyObserver，对 ON_CREATE 和 ON_RESUME 事件进行监听。因为在 Android Support Library 26.1.0 及之后的版本中，Activity 和 Fragment 已经默认实现了 LifecycleOwner 接口，所以在注释 1 处可以直接使用 getLifecycle 方法获取 Lifecycle 对象，这样 MyObserver 就可以观察 MainActivity 的生命周期变化了，LifecycleOwner 可以理解为被观察者，MainActivity 默认实现了 LifecycleOwner 接口，也就是说 MainActivity 是被观察者。

运行程序，打印的日志如下所示。

```
D/MainActivity: onResume
D/MainActivity: Lifecycle call onResume
D/MainActivity: Lifecycle call onPause
D/MainActivity: onPause
```

只要在 MainActivity 的 onCreate 方法中添加 MyObserver，那么 MyObserver 就可以观察到 MainActivity 各个生命周期的变化。

2. 在 MVP 中使用

改写 10.4.1 节 MVP 的示例，首先实现 MyPresenter，如下所示。

```
public class MyPresenter implements IPresenter {
    private static final String TAG = "test";

    @Override
    public void onResume() {
        Log.d(TAG, "Lifecycle call onResume");
    }
```

```
    @Override
    public void onPause() {
        Log.d(TAG, "Lifecycle call onPause");
    }
}

interface IPresenter extends LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    void onResume();

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    void onPause();
}
```

IPresenter 接口继承自 LifecycleObserver 接口，MyPresenter 又实现了 IPresenter 接口，这样 MyPresenter 就成为一个观察者。

然后在 MainActivity 中加入 MyPresenter，如下所示。

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "test";
    private IPresenter mPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mPresenter = new MyPresenter();
        getLifecycle().addObserver(mPresenter);
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");

    }
}
```

MainActivity 成为被观察者，当它的生命周期发生变化时，MyPresenter 就可以观察到了，这样就不需要在 MainActivity 的多个生命周期方法中调用 MyPresenter 的方法了。

打印的日志如下所示。

```
D/test: onResume
D/test: Lifecycle call onResume
D/test: Lifecycle call onPause
D/test: onPause
```

10.4.4　自定义 LifecycleOwner

如果想实现自定义 LifecycleOwner，那么可以使用 LifecycleRegistry，它是 Lifecycle 的实现类。在 Android Support Library 26.1.0 及之后的版本中，Activity 和 Fragment 已经默认实现了 LifecycleOwner 接口，因此我们可以这样写。

```
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleRegistry;
import android.support.annotation.NonNull;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MyActivity extends AppCompatActivity {
   private LifecycleRegistry lifecycleRegistry;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      lifecycleRegistry = new LifecycleRegistry(this);
      lifecycleRegistry.markState(Lifecycle.State.CREATED);
   }

   @Override
   public void onStart() {
      super.onStart();
      lifecycleRegistry.markState(Lifecycle.State.STARTED);
   }

   @NonNull
   @Override
   public Lifecycle getLifecycle() {
      return lifecycleRegistry;
```

```
    }
}
```

通过新建 LifecycleRegistry，为 LifecycleRegistry 设置 Lifecycle 的各种状态，并通过 getLifecycle 方法返回该 LifecycleRegistry。

本节介绍了 Lifecycle 的基本用法，并通过两个示例来帮助读者消化理解，具体在项目中的使用也不难，其难点是 Lifecycle 的原理，下一节我们就来学习 Lifecycle 的原理。

10.5 Lifecycle 的原理

10.5.1 Lifecycle 的生命周期状态事件和状态

Lifecycle 使用两个枚举来跟踪其关联组件的生命周期状态，这两个枚举分别是 Event 和 State。State 代表 Lifecycle 的生命周期所处的状态。Event 代表 Lifecycle 生命周期对应的事件，这些事件会映射到 Activity 和 Fragment 的回调事件中。

Android 9.0 的 Lifecycle 的源码如下所示。

frameworks/support/lifecycle/common/src/main/java/androidx/lifecycle/Lifecycle.java

```
public abstract class Lifecycle {

    @MainThread
    public abstract void addObserver(@NonNull LifecycleObserver observer);

    @MainThread
    public abstract void removeObserver(@NonNull LifecycleObserver observer);

    @MainThread
    @NonNull
    public abstract State getCurrentState();

    @SuppressWarnings("WeakerAccess")
    public enum Event {
        ON_CREATE,
        ON_START,
        ON_RESUME,
        ON_PAUSE,
        ON_STOP,
        ON_DESTROY,
```

```
        ON_ANY
    }

    @SuppressWarnings("WeakerAccess")
    public enum State {
        DESTROYED,
        INITIALIZED,
        CREATED,
        STARTED,
        RESUMED;
        public boolean isAtLeast(@NonNull State state) {
            return compareTo(state) >= 0;
        }
    }
}
```

Lifecycle 是一个抽象类，其内部不仅包括了添加和移除观察者的方法，还包括了 Event 和 State 两个枚举。可以看到 Event 中的事件和 Activity 的生命周期几乎是对应的，除了 ON_ANY，它还可用于匹配所有事件。

States 与 Events 的关系如图 10-4 所示。

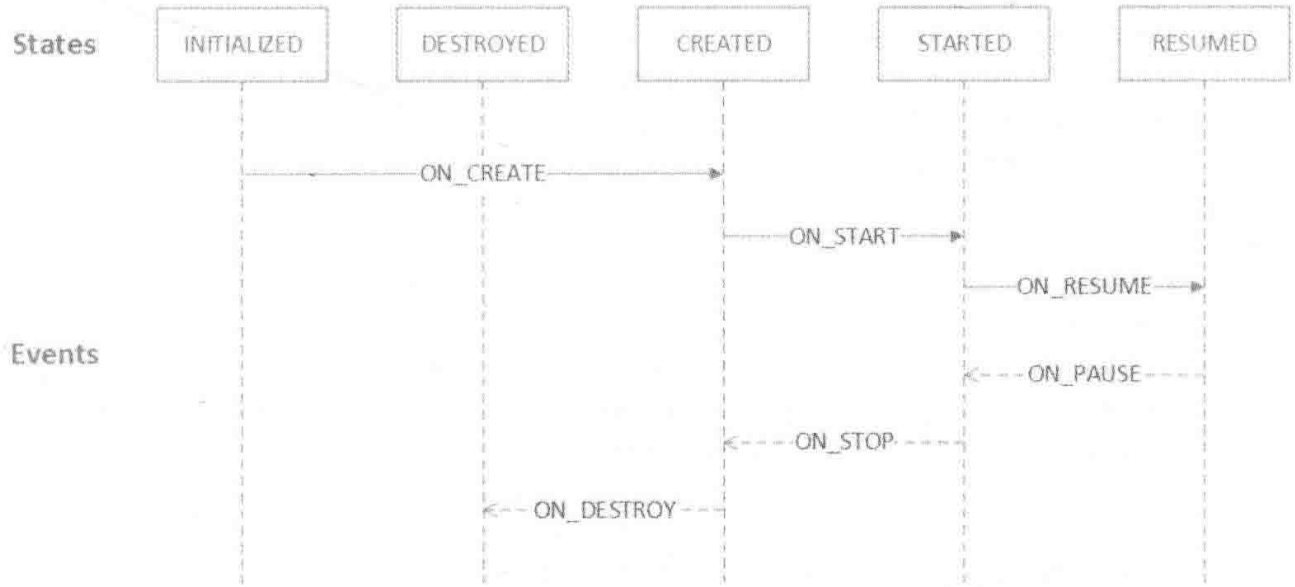


图 10-4　States 与 Events 的关系

10.5.2　Lifecycle 如何观察 Activity 和 Fragment 的生命周期

在 Android Support Library 26.1.0 及之后的版本中，Activity 和 Fragment 已经默认实现了 LifecycleOwner 接口，LifecycleOwner 可以理解为被观察者，那么 Lifecycle 如何观察

Activity 和 Fragment 的生命周期呢？

在 10.4 节的示例中，MainActivity 继承了 AppCompatActivity，而 AppCompatActivity 继承了 FragmentActivity。在 Android 8.0 时，FragmentActivity 继承自 SupportActivity，而在 Android 9.0 时，FragmentActivity 继承自 ComponentActivity 。SupportActivity 和 ComponentActivity 的代码区别不大，这里以 ComponentActivity 为例，如下所示。

frameworks/support/compat/src/main/java/androidx/core/app/ComponentActivity.java

```
@RestrictTo(LIBRARY_GROUP)
public class ComponentActivity extends Activity implements LifecycleOwner {
   private SimpleArrayMap<Class<? extends ExtraData>, ExtraData> mExtraDataMap =
         new SimpleArrayMap<>();

   private LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);//1

   @RestrictTo(LIBRARY_GROUP)
   public void putExtraData(ExtraData extraData) {
      mExtraDataMap.put(extraData.getClass(), extraData);
   }

   @Override
   @SuppressWarnings("RestrictedApi")
   protected void onCreate(@Nullable Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      ReportFragment.injectIfNeededIn(this);//2
   }

   @CallSuper
   @Override
   protected void onSaveInstanceState(Bundle outState) {
      mLifecycleRegistry.markState(Lifecycle.State.CREATED);//3
      super.onSaveInstanceState(outState);
   }

   @RestrictTo(LIBRARY_GROUP)
   public <T extends ExtraData> T getExtraData(Class<T> extraDataClass) {
      return (T) mExtraDataMap.get(extraDataClass);
   }

   @Override
   public Lifecycle getLifecycle() {
      return mLifecycleRegistry;//4
   }
```

```
    @RestrictTo(LIBRARY_GROUP)
    public static class ExtraData {
    }
}
```

注释 1 处创建了 LifecycleRegistry，它是 Lifecycle 的实现类。注释 4 处实现了 LifecycleOwner 接口定义的 getLifecycle 方法，并返回了 LifecycleRegistry。在注释 3 处，将 Lifecycle 的 State 设置为 CREATED。正常来说应该在 ComponentActivity 的各个生命周期方法中改变 Lifecycle 的 State，显然在 ComponentActivity 中没有做这些，而是将这个任务交给了 ReportFragment，注释 2 处将 ComponentActivity 注入 ReportFragment 中。

frameworks/support/lifecycle/runtime/src/main/java/androidx/lifecycle/ReportFragment.java

```
@RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)
public class ReportFragment extends Fragment {
    private static final String REPORT_FRAGMENT_TAG = "androidx.lifecycle"
            + ".LifecycleDispatcher.report_fragment_tag";
    public static void injectIfNeededIn(Activity activity) {
        android.app.FragmentManager manager = activity.getFragmentManager();
        if (manager.findFragmentByTag(REPORT_FRAGMENT_TAG) == null) {
            manager.beginTransaction().add(new ReportFragment(),
REPORT_FRAGMENT_TAG).commit();
            manager.executePendingTransactions();
        }
    }
    static ReportFragment get(Activity activity) {
        return (ReportFragment) activity.getFragmentManager().findFragmentByTag(
                REPORT_FRAGMENT_TAG);
    }
    ...
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        dispatchCreate(mProcessListener);
        dispatch(Lifecycle.Event.ON_CREATE);
    }

    @Override
    public void onStart() {
        super.onStart();
        dispatchStart(mProcessListener);
        dispatch(Lifecycle.Event.ON_START);//1
    }
```

```
        @Override
        public void onResume() {
            super.onResume();
            dispatchResume(mProcessListener);
            dispatch(Lifecycle.Event.ON_RESUME);
        }

    ...
        private void dispatch(Lifecycle.Event event) {
            Activity activity = getActivity();
            if (activity instanceof LifecycleRegistryOwner) {//2
                ((LifecycleRegistryOwner)
activity).getLifecycle().handleLifecycleEvent(event);
                return;
            }

            if (activity instanceof LifecycleOwner) {//3
                Lifecycle lifecycle = ((LifecycleOwner) activity).getLifecycle();
                if (lifecycle instanceof LifecycleRegistry) {
                    ((LifecycleRegistry) lifecycle).handleLifecycleEvent(event);
                }
            }
        }
    ...
    }
```

ReportFragment 的 onStart 方法中会调用注释 1 处的 dispatch 方法。在 dispatch 方法的注释 2 处，判断 Activity 是否实现了 LifecycleRegistryOwner 接口，LifecycleRegistryOwner 继承了 LifecycleOwner 接口，这两个接口有所不同，LifecycleRegistryOwner 定义的 getLifecycle 方法返回的是 LifecycleRegistry 类型，而 LifecycleOwner 定义的 getLifecycle 方法返回的是 Lifecycle 类型。在注释 3 处，如果 Activity 实现了 LifecycleOwner 接口，那么会调用 LifecycleRegistry 的 handleLifecycleEvent 方法。

frameworks/support/lifecycle/runtime/src/main/java/androidx/lifecycle/LifecycleRegistry.java

```
    public void handleLifecycleEvent(@NonNull Lifecycle.Event event) {
        State next = getStateAfter(event);
        moveToState(next);
    }
```

getStateAfter 方法会获取“即将的事件”：当前事件执行后，即将会处于什么事件，代码如下所示。

```
static State getStateAfter(Event event) {
    switch (event) {
        case ON_CREATE:
        case ON_STOP:
            return CREATED;
        case ON_START:
        case ON_PAUSE:
            return STARTED;
        case ON_RESUME:
            return RESUMED;
        case ON_DESTROY:
            return DESTROYED;
        case ON_ANY:
            break;
    }
    throw new IllegalArgumentException("Unexpected event value " + event);
}
```

上述代码和图 10-4 对照看会比较容易理解，如果当前执行了 ON_CREATE 事件或者 ON_STOP 事件，那么状态就会处于 CREATED。回到 handleLifecycleEvent 方法，其内部还会调用 moveToState 方法。

```
private void moveToState(State next) {
    if (mState == next) {
        return;
    }
    mState = next;
    if (mHandlingEvent || mAddingObserverCounter != 0) {
        mNewEventOccurred = true;
        return;
    }
    mHandlingEvent = true;
    sync();
    mHandlingEvent = false;
}
```

如果当前所处的状态和即将要处于的状态一样就不用进行任何操作，sync 方法如下所示。

```
private void sync() {
    LifecycleOwner lifecycleOwner = mLifecycleOwner.get();
    if (lifecycleOwner == null) {
        Log.w(LOG_TAG, "LifecycleOwner is garbage collected, you shouldn't try
dispatch "
                + "new events from it.");
```

```
            return;
        }
        while (!isSynced()) {
            mNewEventOccurred = false;
            if (mState.compareTo(mObserverMap.eldest().getValue().mState) < 0) {
                backwardPass(lifecycleOwner);
            }
            Entry<LifecycleObserver, ObserverWithState> newest =
mObserverMap.newest();
            if (!mNewEventOccurred && newest != null
                    && mState.compareTo(newest.getValue().mState) > 0) {
                forwardPass(lifecycleOwner);
            }
        }
        mNewEventOccurred = false;
    }
```

sync 方法会根据当前状态和 mObserverMap 中的 eldest 和 newest 的状态进行对比，判断当前状态是向前的还是向后的，比如由 STARTED 到 RESUMED 是状态向前的，反过来就是状态向后的，这个不要和 Activity 的生命周期混淆。状态向前还是状态向后的代码大同小异，这里以状态向后为例，如下所示。

```
    private void forwardPass(LifecycleOwner lifecycleOwner) {
        Iterator<Entry<LifecycleObserver, ObserverWithState>> ascendingIterator =
                mObserverMap.iteratorWithAdditions();
        while (ascendingIterator.hasNext() && !mNewEventOccurred) {
            Entry<LifecycleObserver, ObserverWithState> entry =
ascendingIterator.next();
            ObserverWithState observer = entry.getValue();//1
            while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred
                    && mObserverMap.contains(entry.getKey()))) {
                pushParentState(observer.mState);
                observer.dispatchEvent(lifecycleOwner,
upEvent(observer.mState));//2
                popParentState();
            }
        }
    }
```

注释 1 处获取了 ObserverWithState，后面章节会再提到它。

注释 2 处的 upEvent 方法会得到当前状态的向前状态。ObserverWithState 的 dispatchEvent 方法如下所示。

frameworks/support/lifecycle/runtime/src/main/java/androidx/lifecycle/LifecycleRegistry.java

```
static class ObserverWithState {
    State mState;
    GenericLifecycleObserver mLifecycleObserver;

    ObserverWithState(LifecycleObserver observer, State initialState) {
        mLifecycleObserver = Lifecycling.getCallback(observer);//1
        mState = initialState;
    }

    void dispatchEvent(LifecycleOwner owner, Event event) {
        State newState = getStateAfter(event);
        mState = min(mState, newState);
        mLifecycleObserver.onStateChanged(owner, event);
        mState = newState;
    }
}
```

从名称就可以看出来，ObserverWithState 内部包括了 State 和 GenericLifecycleObserver，GenericLifecycleObserver 是一个接口，它继承了 LifecycleObserver 接口。

ReflectiveGenericLifecycleObserver 和 CompositeGeneratedAdaptersObserver 是 GenericLifecycleObserver 的实现类，这里主要查看 ReflectiveGenericLifecycleObserver 的 onStateChanged 方法是如何实现的，如下所示。

frameworks/support/lifecycle/common/src/main/java/androidx/lifecycle/ReflectiveGenericLifecycleObserver.java

```
class ReflectiveGenericLifecycleObserver implements GenericLifecycleObserver {
    private final Object mWrapped;
    private final CallbackInfo mInfo;

    ReflectiveGenericLifecycleObserver(Object wrapped) {
        mWrapped = wrapped;
        mInfo = ClassesInfoCache.sInstance.getInfo(mWrapped.getClass());
    }

    @Override
    public void onStateChanged(LifecycleOwner source, Event event) {
        mInfo.invokeCallbacks(source, event, mWrapped);//1
    }
}
```

注释 1 处会调用 CallbackInfo 的 invokeCallbacks 方法，在讲这个方法前，首先需要知

道 CallbackInfo 是由 createInfo 方法创建的，如下所示。

```
    private CallbackInfo createInfo(Class klass, @Nullable Method[] declaredMethods)
{
        Class superclass = klass.getSuperclass();
        Map<MethodReference, Lifecycle.Event> handlerToEvent = new HashMap<>();
        ...
        Method[] methods = declaredMethods != null ? declaredMethods :
getDeclaredMethods(klass);
        boolean hasLifecycleMethods = false;
        for (Method method : methods) {
            OnLifecycleEvent annotation =
method.getAnnotation(OnLifecycleEvent.class);//1
            if (annotation == null) {
                continue;
            }
            hasLifecycleMethods = true;
            Class<?>[] params = method.getParameterTypes();
            int callType = CALL_TYPE_NO_ARG;
            if (params.length > 0) {
                callType = CALL_TYPE_PROVIDER;
                if (!params[0].isAssignableFrom(LifecycleOwner.class)) {
                    throw new IllegalArgumentException(
                            "invalid parameter type. Must be one and instanceof
LifecycleOwner");
                }
            }
            Lifecycle.Event event = annotation.value();//2
            ...
            MethodReference methodReference = new MethodReference(callType,
method);//3
            verifyAndPutHandler(handlerToEvent, methodReference, event, klass);//4
        }
        CallbackInfo info = new CallbackInfo(handlerToEvent);//5
        mCallbackMap.put(klass, info);
        mHasLifecycleMethods.put(klass, hasLifecycleMethods);
        return info;
    }
```

关键点在注释 1 处，不断遍历各个方法，获取方法上的 OnLifecycleEvent 注解，这个注解正是实现 LifecycleObserver 接口时用到的。注释 2 处获取该注解的值，也就是在 @OnLifecycleEvent 中定义的事件。注释 3 处新建了一个 MethodReference，其内部包括使用了该注解的方法。注释 4 处的 verifyAndPutHandler 方法用于将 MethodReference 和对应的

Event 存入类型为 Map<MethodReference, Lifecycle.Event>的 handlerToEvent 中。

注释 5 处新建 CallbackInfo，并将 handlerToEvent 传进去。

继续看 CallbackInfo 的 invokeCallbacks 方法，代码如下所示。

frameworks/support/lifecycle/common/src/main/java/androidx/lifecycle/ClassesInfoCache.java

```
static class CallbackInfo {
        final Map<Lifecycle.Event, List<MethodReference>> mEventToHandlers;
        final Map<MethodReference, Lifecycle.Event> mHandlerToEvent;
        CallbackInfo(Map<MethodReference, Lifecycle.Event> handlerToEvent) {
           mHandlerToEvent = handlerToEvent;
           mEventToHandlers = new HashMap<>();
           for (Map.Entry<MethodReference, Lifecycle.Event> entry :
handlerToEvent.entrySet()) {//1
              Lifecycle.Event event = entry.getValue();
              List<MethodReference> methodReferences =
mEventToHandlers.get(event);
              if (methodReferences == null) {
                 methodReferences = new ArrayList<>();
                 mEventToHandlers.put(event, methodReferences);
              }
              methodReferences.add(entry.getKey());
           }
        }
        @SuppressWarnings("ConstantConditions")
        void invokeCallbacks(LifecycleOwner source, Lifecycle.Event event, Object
target) {
           invokeMethodsForEvent(mEventToHandlers.get(event), source, event,
target);//2
           invokeMethodsForEvent(mEventToHandlers.get(Lifecycle.Event.ON_ANY),
source, event,
                 target);
        }

        private static void invokeMethodsForEvent(List<MethodReference> handlers,
              LifecycleOwner source, Lifecycle.Event event, Object mWrapped) {
           if (handlers != null) {
              for (int i = handlers.size() - 1; i >= 0; i--) {
                 handlers.get(i).invokeCallback(source, event, mWrapped);
              }
           }
        }
```

注释 1 处的循环的意义在于将 handlerToEvent 进行数据类型转换，转换为一个

HashMap，key 的值为事件，value 的值为 MethodReference。注释 2 处的 invokeMethodsForEvent 方法会传入 mEventToHandlers.get(event)，也就是事件对应的 MethodReference 的集合。在 invokeMethodsForEvent 方法中会遍历 MethodReference 的集合，调用 MethodReference 的 invokeCallback 方法，如下所示。

frameworks/support/lifecycle/common/src/main/java/androidx/lifecycle/ClassesInfoCache.java

```
    @SuppressWarnings("WeakerAccess")
      static class MethodReference {
         final int mCallType;
         final Method mMethod;
         MethodReference(int callType, Method method) {
            mCallType = callType;
            mMethod = method;
            mMethod.setAccessible(true);
         }
         void invokeCallback(LifecycleOwner source, Lifecycle.Event event, Object
target) {
            try {
               switch (mCallType) {
                  case CALL_TYPE_NO_ARG:
                     mMethod.invoke(target);
                     break;
                  case CALL_TYPE_PROVIDER:
                     mMethod.invoke(target, source);
                     break;
                  case CALL_TYPE_PROVIDER_WITH_EVENT:
                     mMethod.invoke(target, source, event);
                     break;
               }
            } catch (InvocationTargetException e) {
               throw new RuntimeException("Failed to call observer method",
e.getCause());
            } catch (IllegalAccessException e) {
               throw new RuntimeException(e);
            }
         }
        ...
      }
```

MethodReference 类中有两个变量，一个变量是 callType，它代表调用方法的类型；另一个变量是 Method，它代表方法，无论是哪种 callType 都会通过 invoke 对方法进行反射。

简单来说，实现 LifecycleObserver 接口的类中，注解修饰的方法和事件会被保存起来，

通过反射对事件的对应方法进行调用，这个调用链的时序图如图 10-5 所示。

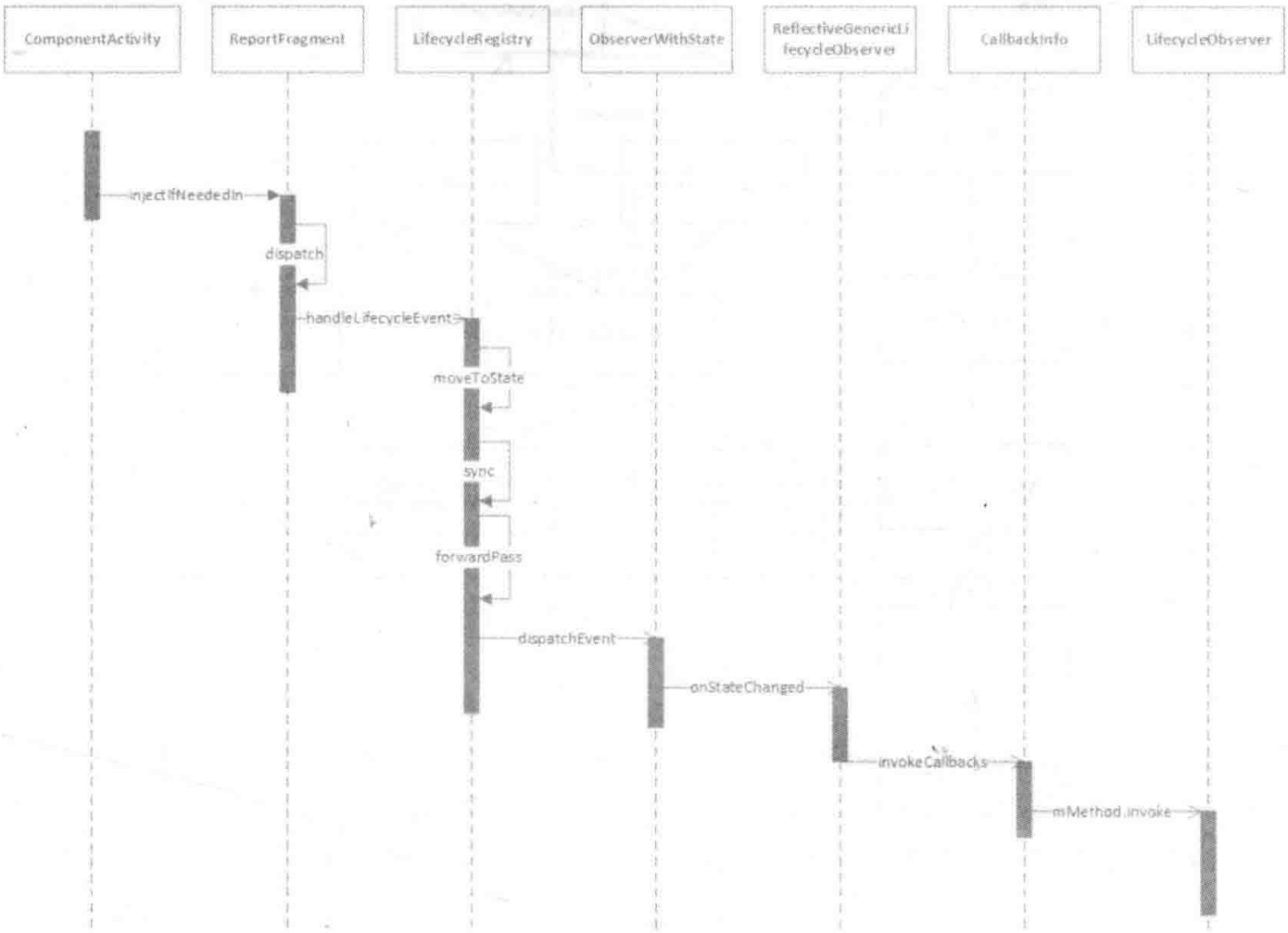


图 10-5　调用链的时序图

10.5.3　Lifecycle 关联类

10.5.2 节是以一个调用链的形式来介绍 Lifecycle 的，调用链中涉及的类具体都是什么关系很难整体去把握，因此本节再以 UML 类图的形式给出 Lifecycle 关联类，其版本为 Android 9.0，如图 10-6 所示。

AppCompatActivity 继承自 FragmentActivity，这两个类都不算是 Lifecycle 关联类。FragmentActivity 实现了 LifecycleOwner 接口，LifecycleOwner 和 Lifecycle 有一个“弱拥有的关系”，LifecycleRegistry 继承自 Lifecycle，剩下的内容就不多介绍了，看图一目了然。如果不理解 UML 类图，那么可以阅读《大话设计模式》这本书。

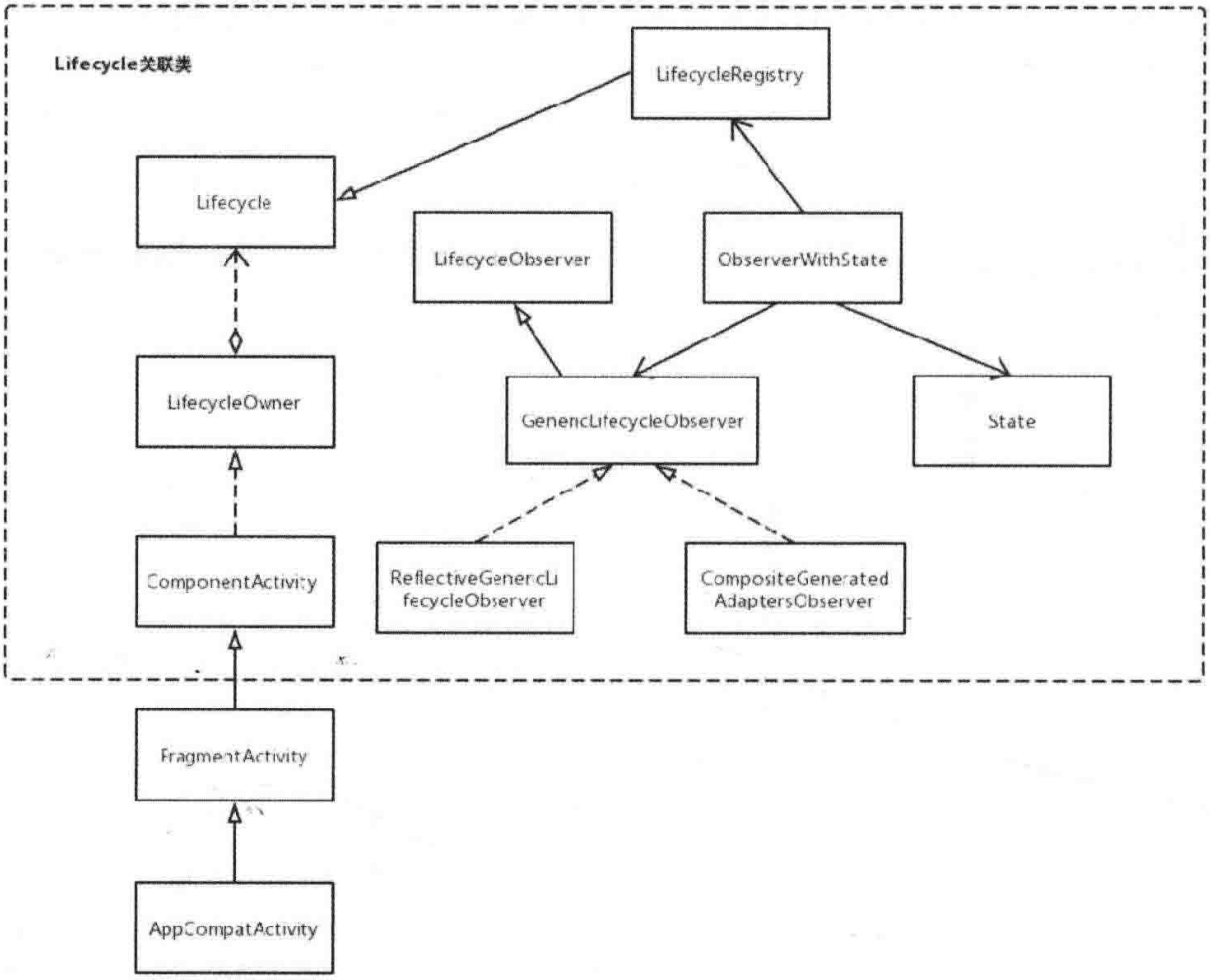


图 10-6　Lifecycle 关联类

10.6　使用 LiveData

在 2017 年前后，RxJava 一直很受欢迎，我在《Android 进阶之光》中就介绍了 RxJava 的使用和原理。Google 推出的 LiveData 和 RxJava 类似，也是基于观察者的，可以认为 LiveData 是轻量级的 RxJava。起初 LiveData 并不被看好，随着 Google 的大力推广，LiveData 也慢慢进入了大家的视野。一般来说，LiveData 很少单独使用，它更多的是和 Android Jetpack 的其他组件搭配使用，比如和 ViewModel 搭配使用。本节就来介绍 LiveData 的使用。

10.6.1　什么是 LiveData

LiveData 是一个可观察的数据持有者，和常规的 observable 不同，LiveData 是具有生

命周期感知的，这意味着它能够在 Activity、Fragment、Service 中正确地处理生命周期，LiveData 通知模型如图 10-7 所示。

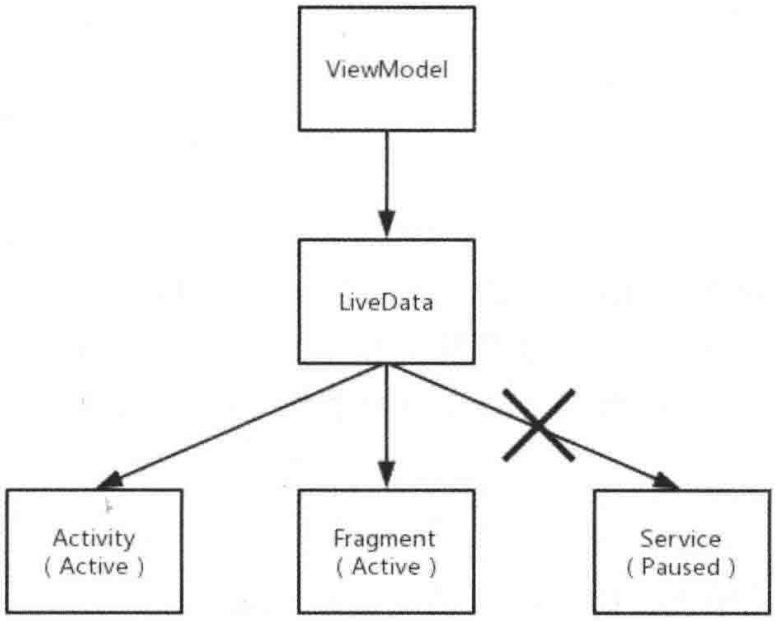


图 10-7　LiveData 通知模型

LiveData 的数据源一般是 ViewModel，也可以是其他可以更新 LiveData 的组件。当数据更新后，LiveData 就会通知它的所有观察者。与 RxJava 的方法不同的是，LiveData 并不是通知所有观察者，它只会通知处于 Active 状态的观察者，如果一个观察者处于 Paused 状态或 Destroyed 状态，那么它将不会收到通知。

这对于 Activity 和 Service 特别有用，因为它们可以安全地观察 LiveData 对象而不用担心内存泄漏的问题。开发者也不需要在 onPause 或 onDestroy 方法中解除对 LiveData 的订阅。还有一点需要注意的是，一旦观察者重新恢复 Resumed 状态，它将会重新收到 LiveData 的最新数据。

10.6.2　LiveData 的基本用法

LiveData 是一个抽象类，它的简单的实现类为 MutableLiveData，这里举个简单的示例。

```
public class MainActivity extends AppCompatActivity {
  private static final String TAG="MainActivity";
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      MutableLiveData<String> mutableLiveData  = new MutableLiveData<>();
```

```
        mutableLiveData.observe(this, new Observer<String>() {//1
            @Override
            public void onChanged(@Nullable final String s) {
                Log.d(TAG, "onChanged:"+s);
            }
        });
        mutableLiveData.postValue("Android进阶三部曲");//2
    }
}
```

注释 1 处的 observe 方法有两个参数，分别是 LifecycleOwner 和 Observer<T>，第一个参数就是 MainActivity 本身，第二个参数新建了一个 Observer<String>，在 onChanged 方法中得到回调。注释 2 处的 postValue 方法会在主线程中更新数据，这样就会得到打印的结果，如下所示。

```
D/MainActivity: onChanged:Android进阶三部曲
```

在大多数情况下，LiveData 的 observe 方法会放在 onCreate 方法中，如果放在 onResume 方法中，则会出现多次调用的问题。除了 MutableLiveData 的 postValue 方法，还可以使用 setValue 方法，它们的区别是，setValue 方法必须在主线程使用，如果是在工作线程中更新 LiveData 的，则可以使用 postValue 方法。

10.6.3 更改 LiveData 中的数据

如果我们想要在 LiveData 对象分发给观察者之前对其中存储的值进行更改，则可以使用 Transformations.map 方法和 Transformations.switchMap 方法，下面通过简单的示例来讲解它们。

1. Transformations.map 方法

如果想要在 LiveData 对象分发给观察者之前对其中存储的值进行更改，则可以使用 Transformations.map 方法。

```
public class MainActivity extends AppCompatActivity {
  private static final String TAG="MainActivity";
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      MutableLiveData<String> mutableLiveData  = new MutableLiveData<>();
      mutableLiveData.observe(this, new Observer<String>() {
```

```
            @Override
            public void onChanged(@Nullable final String s) {
                Log.d(TAG, "onChanged1:"+s);
            }
        });
        LiveData transformedLiveData =Transformations.map(mutableLiveData, new
Function<String, Object>() {
            @Override
            public Object apply(String name) {
               return name + "+Android进阶解密";
            }
        });
        transformedLiveData.observe(this, new Observer() {
            @Override
            public void onChanged(@Nullable Object o) {
                Log.d(TAG, "onChanged2:"+o.toString());
            }
        });
        mutableLiveData.postValue("Android进阶之光");
    }
}
```

通过 Transformations.map 方法，在 mutableLiveData 的基础上又加上了字符串"+Android 进阶解密"。

打印结果如下。

```
D/MainActivity: onChanged1:Android进阶之光
D/MainActivity: onChanged2:Android进阶之光+Android进阶解密
```

2. Transformations.switchMap 方法

如果我们想要手动控制监听其中一个数据的变化，并能根据需要随时切换监听，这时可以使用 Transformations.switchMap 方法，它和 Transformations.map 方法的使用方式类似，只不过 switchMap 方法必须返回一个 LiveData 对象。

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    MutableLiveData<String> mutableLiveData1;
    MutableLiveData<String> mutableLiveData2;
    MutableLiveData<Boolean> liveDataSwitch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mutableLiveData1 = new MutableLiveData<>();
        mutableLiveData2 = new MutableLiveData<>();
        liveDataSwitch = new MutableLiveData<Boolean>();//1

        LiveData transformedLiveData= Transformations.switchMap(liveDataSwitch,
new Function<Boolean, LiveData<String>>() {
            @Override
            public LiveData<String> apply(Boolean input) {
                if (input) {
                    return mutableLiveData1;
                } else {
                    return mutableLiveData2;
                }
            }
        });

        transformedLiveData.observe(this, new Observer<String>() {
            @Override
            public void onChanged(@Nullable final String s) {
                Log.d(TAG, "onChanged:" + s);
            }
        });
        liveDataSwitch.postValue(false);//2
        mutableLiveData1.postValue("Android 进阶之光");
        mutableLiveData2.postValue("Android 进阶解密");
    }
}
```

注释 1 处新建一个 MutableLiveData<Boolean>()来控制切换并赋值给 liveDataSwitch，当 liveDataSwitch 的值为 true 时返回 mutableLiveData1，否则返回 mutableLiveData2。注释 2 处将 liveDataSwitch 的值更新为 false，这样输出的结果为"Android 进阶解密"，达到了切换监听的目的。

10.6.4 合并多个 LiveData 数据源

MediatorLiveData 继承自 mutableLiveData，它可以将多个 LiveData 数据源合并起来，达到一个组件监听多个 LiveData 数据变化的目的。

```
public class MainActivity extends AppCompatActivity {
  private static final String TAG="MainActivity";
   @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        MutableLiveData<String> mutableLiveData1  = new MutableLiveData<>();
        MutableLiveData<String> mutableLiveData2  = new MutableLiveData<>();
        MediatorLiveData liveDataMerger = new MediatorLiveData<String>();
        liveDataMerger.addSource(mutableLiveData1, new Observer() {
            @Override
            public void onChanged(@Nullable Object o) {
                Log.d(TAG, "onChanged1:"+o.toString());
            }
        });

        liveDataMerger.addSource(mutableLiveData2, new Observer() {
            @Override
            public void onChanged(@Nullable Object o) {
                Log.d(TAG, "onChanged2:"+o.toString());
            }
        });
        liveDataMerger.observe(this, new Observer() {
            @Override
            public void onChanged(@Nullable Object o) {
                Log.d(TAG, "onChanged:"+o.toString());
            }
        });
        mutableLiveData1.postValue("Android进阶之光");
    }
}
```

为了更直观的举例，将 LiveData 和 MediatorLiveData 放到了同一个 Activity 中。通过 MediatorLiveData 的 addSource 将两个 MutableLiveData 合并到一起，这样当任何一个 MutableLiveData 数据发生变化时，MediatorLiveData 都可以观察到。

打印的结果如下。

```
D/MainActivity: onChanged1:Android进阶之光
```

10.6.5　拓展 LiveData 对象

如果观察者的生命周期处于 STARTED 或 RESUMED 状态，则 LiveData 会将观察者视为处于 Active 状态。关于如何扩展 LiveData，官方网站的示例是比较简洁的，如下所示。

```
public class StockLiveData extends LiveData<BigDecimal> {
    private static StockLiveData sInstance;
```

```
    private StockManager stockManager;

    private SimplePriceListener listener = new SimplePriceListener() {
        @Override
        public void onPriceChanged(BigDecimal price) {
            setValue(price);
        }
    };

    @MainThread
    public static StockLiveData get(String symbol) {
        if (sInstance == null) {
            sInstance = new StockLiveData(symbol);
        }
        return sInstance;
    }

    private StockLiveData(String symbol) {
        stockManager = new StockManager(symbol);
    }

    @Override
    protected void onActive() {
        stockManager.requestPriceUpdates(listener);
    }

    @Override
    protected void onInactive() {
        stockManager.removeUpdates(listener);
    }
}
```

上述代码是一个观察股票变动的示例，对 LiveData 进行了拓展，并实现了 LiveData 的两个空方法：onActive 和 onInactive。当 Active 状态的观察者数量从 0 变为 1 时会调用 onActive 方法，通俗来讲，就是当 LiveData 对象具有 Active 状态的观察者时调用 onActive 方法，应该在 onActive 方法中开始观察股票价格的更新。当 LiveData 对象没有任何 Active 状态的观察者时调用 onInactive 方法，在这个方法中，断开与 StockManager 服务的连接。

在 Fragment 中使用 StockLiveData，如下所示。

```
public class MyFragment extends Fragment {
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        StockLiveData.get(symbol).observe(this, price -> {
```

```
        });
    }
}
```

10.7　LiveData 的原理

在 10.6 节中我们学习了 LiveData 的基本用法，LiveData 是一个可观察的数据持有者，它是具有组件生命周期感知的，那么它是如何观察组件生命周期变化的呢？LiveData 和 RxJava 不同的是，LiveData 并不会通知所有观察者，它只会通知处于 Active 状态的观察者，如果一个观察者处于 DESTROYED 状态，那么它将不会收到通知，这一点又是如何做到的？Transformations 的 map 方法其内部进行了什么操作？这些问题会在本节中进行讲解。

10.7.1　LiveData 如何观察组件生命周期变化

通过调用 LiveData 的 observe 方法来注册观察者，LiveData 的 observe 方法如下所示。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
@MainThread
    public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super T>
observer) {
        assertMainThread("observe");
        //如果被观察者的当前的状态是 DESTROYED，就调用 return
        if (owner.getLifecycle().getCurrentState() == DESTROYED) {//1
            return;
        }
        LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner,
observer);//2
        ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);//3
        if (existing != null && !existing.isAttachedTo(owner)) {
            throw new IllegalArgumentException("Cannot add the same observer"
                    + " with different lifecycles");
        }
        if (existing != null) {
            return;
        }
        owner.getLifecycle().addObserver(wrapper);//4
    }
```

注释 1 处的 owner 实际上就是注册时传进来的组件，比如 Activity，获取组件当前的状

态，如果组件当前的状态为 DESTROYED，那么直接 return，这说明 DESTROYED 状态的组件是不允许被注册的。

注释 2 处新建了一个 LifecycleBoundObserver 包装类，并将 owner 和 observer 传入这个包装类中。

注释 3 处将 observer 和 LifecycleBoundObserver 存储到 SafeIterableMap <Observer<? super T>, ObserverWrapper>mObservers 中，putIfAbsent 方法和 put 方法是有区别的，如果传入 key 对应的 value 已经存在，则返回存在的 value，不进行替换。如果传入 key 对应的 value 不存在，则添加 key 和 value，返回 null。

如果 existing 等于 null，在注释 4 处会将 LifecycleBoundObserver 添加到 Lifecycle 中完成注册，这样当我们调用 LiveData 的 observe 方法时，实际上是 LiveData 内部完成了 Lifecycle 的观察者的添加，这样 LiveData 自然也就有了观察组件生命周期变化的能力。

10.7.2 LiveData 的 observe 方法回调

LifecycleBoundObserver 是 LiveData 的内部类，代码如下所示。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
    class LifecycleBoundObserver extends ObserverWrapper implements
GenericLifecycleObserver {
        @NonNull final LifecycleOwner mOwner;

        LifecycleBoundObserver(@NonNull LifecycleOwner owner, Observer<? super T>
observer) {
            super(observer);
            mOwner = owner;
        }

        @Override
        boolean shouldBeActive() {
            return mOwner.getLifecycle().getCurrentState().isAtLeast(STARTED);
        }

        @Override
        public void onStateChanged(LifecycleOwner source, Lifecycle.Event event) {
            if (mOwner.getLifecycle().getCurrentState() == DESTROYED) {
                removeObserver(mObserver);//1
                return;
            }
```

```
        activeStateChanged(shouldBeActive());//2
    }

    @Override
    boolean isAttachedTo(LifecycleOwner owner) {
        return mOwner == owner;
    }

    @Override
    void detachObserver() {
        mOwner.getLifecycle().removeObserver(this);
    }
}
```

LifecycleBoundObserver 继承了 ObserverWrapper 类，重写了 shouldBeActive 方法，用于判断当前传入的组件的状态是否是 Active 状态的，Active 状态包括 STARTED 状态和 RESUMED 状态。

LifecycleBoundObserver 实现了 GenericLifecycleObserver 接口，当组件状态发生变化时，会调用 onStateChanged 方法，当组件处于 DESTROYED 状态时，会调用注释 1 处的 removeObserver 方法来移除 observer。这样一个观察者（组件）处于 DESTROYED 状态时，它将不会收到通知。

接着会调用注释 2 处的 activeStateChange 方法，代码如下所示。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
private abstract class ObserverWrapper {
    final Observer<? super T> mObserver;
    boolean mActive;
    int mLastVersion = START_VERSION;

    ObserverWrapper(Observer<? super T> observer) {
        mObserver = observer;
    }

    abstract boolean shouldBeActive();

    boolean isAttachedTo(LifecycleOwner owner) {
        return false;
    }

    void detachObserver() {
```

```
        }

        void activeStateChanged(boolean newActive) {
            if (newActive == mActive) {
                return;
            }
            mActive = newActive;
            boolean wasInactive = LiveData.this.mActiveCount == 0;
            LiveData.this.mActiveCount += mActive ? 1 : -1;
            if (wasInactive && mActive) {
                onActive();
            }
            if (LiveData.this.mActiveCount == 0 && !mActive) {
                onInactive();
            }
            if (mActive) {
                dispatchingValue(this);//1
            }
        }
    }
```

activeStateChanged 方法定义在抽象类 ObserverWrapper 中，它是 Observer 的包装类，activeStateChanged 方法会根据 Active 状态和处于 Active 状态的组件的数量，对 onActive 方法和 onInactive 方法进行回调，这两个方法用于拓展 LiveData 对象。在注释 1 处，如果处于 Active 状态，则会调用 dispatchingValue 方法，并将自身传入其中。

```
    private void dispatchingValue(@Nullable ObserverWrapper initiator) {
        //正在处于分发状态中
        if (mDispatchingValue) {
            //分发无效
            mDispatchInvalidated = true;//1
            return;
        }
        mDispatchingValue = true;
        do {
            //分发有效
            mDispatchInvalidated = false;
            if (initiator != null) {
                considerNotify(initiator);
                initiator = null;
            } else {
                for (Iterator<Map.Entry<Observer<? super T>, ObserverWrapper>>
iterator =
                        mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
```

```
                considerNotify(iterator.next().getValue());
                if (mDispatchInvalidated) {
                    break;
                }
            }
        }
    } while (mDispatchInvalidated);
    //标记不处于分发状态
    mDispatchingValue = false;
}
```

mDispatchingValue 用于标记当前是否处于分发状态中，如果处于该状态，则在注释 1 处标记当前分发无效，直接 return。一路调用过来，ObserverWrapper 是不为 null 的，ObserverWrapper 为 null 的情况在 10.7.3 节会讲到，无论是哪种情况，都会调用 considerNotify 方法，代码如下所示。

```
private void considerNotify(ObserverWrapper observer) {
    if (!observer.mActive) {//1
        return;
    }
    if (!observer.shouldBeActive()) {
        observer.activeStateChanged(false);//2
        return;
    }
    if (observer.mLastVersion >= mVersion) {
        return;
    }
    observer.mLastVersion = mVersion;
    observer.mObserver.onChanged((T) mData);//3
}
```

considerNotify 方法中进行了多次判断，在注释 1 处，如果 ObserverWrapper 的 mActive 值不为 true，就直接 return。注释 2 处，在如果当前 observer 对应组件的状态不是 Active，就会再次调用 activeStateChanged 方法，并传入 false，其方法内部会再次判断是否执行 onActive 方法和 onInactive 方法回调。如果判断条件都满足则会调用 Observer 的 onChanged 方法，这个方法正是使用 LiveData 的 observe 方法的回调。

10.7.3　postValue/setValue 方法分析

当调用 MutableLiveData 的 observe 方法后，还需要通过 postValue/setValue 方法来更新

数据，如下所示。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
...
 private final Runnable mPostValueRunnable = new Runnable() {
     @Override
     public void run() {
        Object newValue;
        synchronized (mDataLock) {
           newValue = mPendingData;
           mPendingData = NOT_SET;
        }
        setValue((T) newValue);//1
     }
  };
  ...
  protected void postValue(T value) {
     boolean postTask;
     synchronized (mDataLock) {
        postTask = mPendingData == NOT_SET;
        mPendingData = value;
     }
     if (!postTask) {
        return;
     }
     ArchTaskExecutor.getInstance().postToMainThread(mPostValueRunnable);//2
  }

  @MainThread //3
  protected void setValue(T value) {
     assertMainThread("setValue");
     mVersion++;
     mData = value;
     dispatchingValue(null);
  }
```

postValue/setValue 方法都定义在 LiveData 中，根据注释 1 处和注释 2 处，可以发现 postValue 方法实际上就是将 setValue 方法切换到主线程调用。注释 3 处说明 setValue 方法是运行在主线程中的，其内部调用了 dispatchingValue 方法，这个方法在 10.7.2 节中介绍过，也就是 dispatchingValue 方法的参数 ObserverWrapper 为 null 的情况。

从这里我们可以知道，无论是 LiveData 的 observe 方法还是 LiveData 的 postValue/setValue 方法都会调用 dispatchingValue 方法。

10.7.4　Transformations.map 方法分析

除了前面章节讲的常用的方法，还可能会使用到 Transformations.map 方法和 Transformations.switchMap 方法，这里以 Transformations.map 方法为例。这个方法用来在 LiveData 对象分发给观察者之前对其中存储的值进行更改，代码如下所示。

frameworks/support/lifecycle/livedata/src/main/java/androidx/lifecycle/Transformations.java

```
@MainThread
   public static <X, Y> LiveData<Y> map(
         @NonNull LiveData<X> source,
         @NonNull final Function<X, Y> mapFunction) {
      final MediatorLiveData<Y> result = new MediatorLiveData<>();//1
      result.addSource(source, new Observer<X>() {
         @Override
         public void onChanged(@Nullable X x) {
            result.setValue(mapFunction.apply(x));
         }
      });
      return result;
   }
```

Transformations.map 方法运行在主线程，注释 1 处创建了 MediatorLiveData，然后调用了它的 addSource 方法，如下所示。

support/lifecycle/livedata/src/main/java/androidx/lifecycle/MediatorLiveData.java

```
   */
   @MainThread
   public <S> void addSource(@NonNull LiveData<S> source, @NonNull Observer<? super
S> onChanged) {
      Source<S> e = new Source<>(source, onChanged);//1
      Source<?> existing = mSources.putIfAbsent(source, e);
      if (existing != null && existing.mObserver != onChanged) {
         throw new IllegalArgumentException(
               "This source was already added with the different observer");
      }
      if (existing != null) {
         return;
      }
      if (hasActiveObservers()) {
         e.plug();//2
      }
   }
```

注释 1 处将传进来的 LiveData 和 onChanged 封装到 Source 类中，注释 2 处调用了 Source 的 plug 方法，如下所示。

```
private static class Source<V> implements Observer<V> {
    final LiveData<V> mLiveData;
    final Observer<? super V> mObserver;
    int mVersion = START_VERSION;

    Source(LiveData<V> liveData, final Observer<? super V> observer) {
        mLiveData = liveData;
        mObserver = observer;
    }

    void plug() {
        mLiveData.observeForever(this);//1
    }

    void unplug() {
        mLiveData.removeObserver(this);
    }

    @Override
    public void onChanged(@Nullable V v) {
        if (mVersion != mLiveData.getVersion()) {
            mVersion = mLiveData.getVersion();
            mObserver.onChanged(v);//2
        }
    }
}
```

在注释 2 处可以看到，Transformations.map 方法传入的 Observer 的回调在这里进行处理。在注释 1 处，Source 的 plug 方法会调用 LiveData 的 observeForever 方法，这与 10.7.2 节所讲的内容有什么区别呢？我们再看如下代码。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
@MainThread
public void observeForever(@NonNull Observer<? super T> observer) {
    assertMainThread("observeForever");
    AlwaysActiveObserver wrapper = new AlwaysActiveObserver(observer);//1
    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
    if (existing != null && existing instanceof LiveData.LifecycleBoundObserver)
{
        throw new IllegalArgumentException("Cannot add the same observer"
                + " with different lifecycles");
```

```
        }
        if (existing != null) {
            return;
        }
        wrapper.activeStateChanged(true);
    }
```

注释 1 处用 AlwaysActiveObserver 来对 Observer 进行包装，然后调用 AlwaysActiveObserver 的 activeStateChanged 方法，其内部实际调用的是 ObserverWrapper 的 activeStateChanged 方法，这个在 10.7.2 节已经分析了，这里就不再赘述了。接着来看 AlwaysActiveObserver 类是如何定义的，如下所示。

frameworks/support/lifecycle/livedata-core/src/main/java/androidx/lifecycle/LiveData.java

```
private class AlwaysActiveObserver extends ObserverWrapper {
    AlwaysActiveObserver(Observer<? super T> observer) {
        super(observer);
    }
    @Override
    boolean shouldBeActive() {
        return true;
    }
}
```

AlwaysActiveObserver 是 LiveData 的内部类，它继承自 ObserverWrapper，和 ObserverWrapper 的区别就是，AlwaysActiveObserver 一直是处于 Active 状态的。

10.7.5　LiveData 关联类

LiveData 关联类的 UML 图如图 10-8 所示，看明白该图之后，再来重新阅读本节，也许会收获更多。

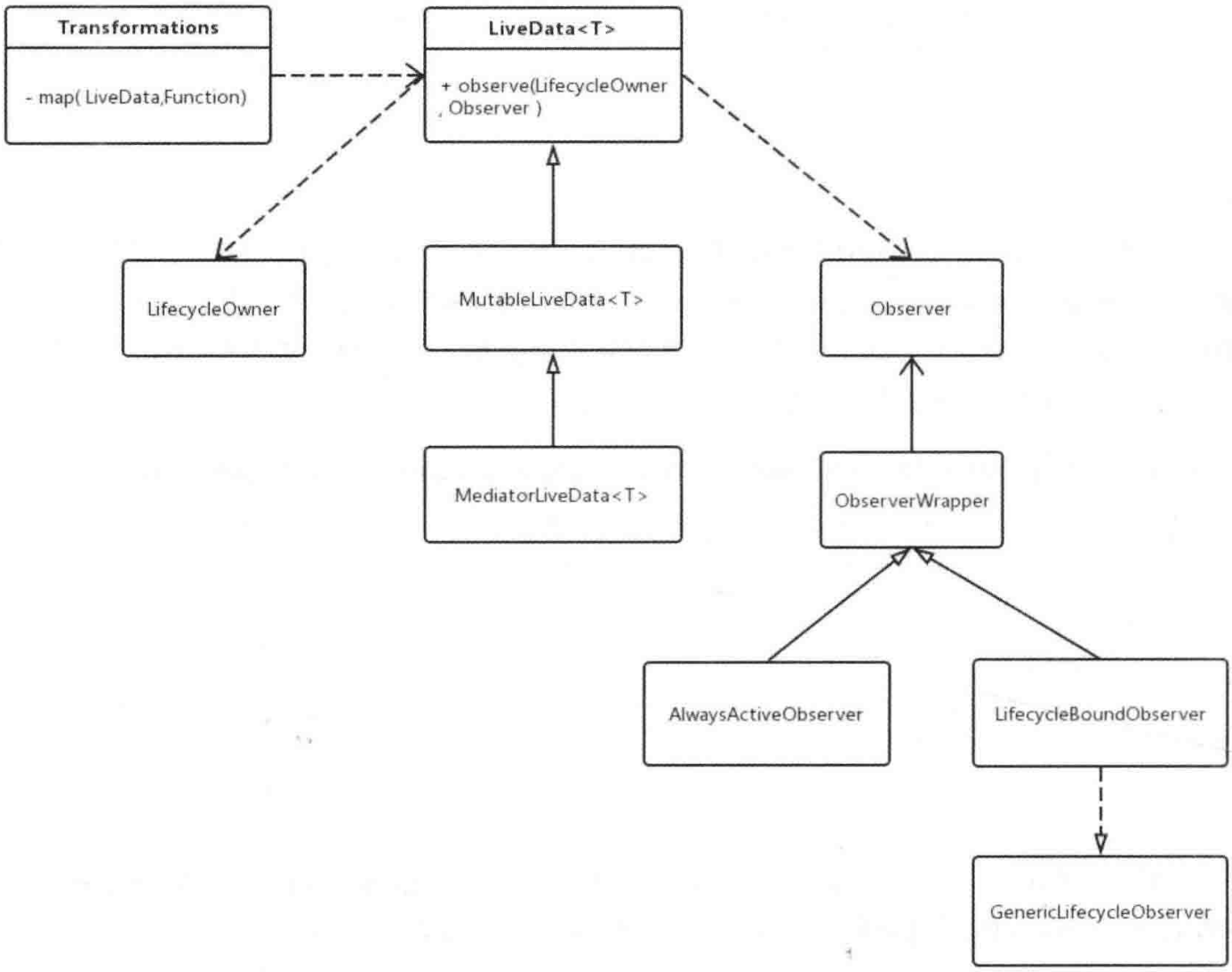


图 10-8　LiveData 关联类的 UML 图

由图 10-8 可知，在讲解 LiveData 时涉及的大部分类都在这个图中，其中 MutableLiveData 继承自 LiveData，LifecycleOwner、Observer 和 LiveData 有相关联的关系，ObserverWrapper 是 Observer 的包装类，其他类看图就可以知道了。

10.8　使用 ViewModel

LiveData 和 ViewModel 是一对好搭档，本节我们一起来学习什么是 ViewModel 及 ViewModel 的基本使用方法。

10.8.1　什么是 ViewModel

在讲 ViewModel 前，我们首先需要了解 MVVM 和 DataBinding，在《Android 进阶之光》一书中，我简单介绍了 MVVM 和 DataBinding。MVVM 最早于 2005 年被微软的架构师 John Gossman 提出，在 2015 年的 Google I/O 大会上发布了 MVVM 的支持库 DataBinding，DataBinding 遭到了一些开发者的诟病，主要是在绑定数据时如果出现问题会很难排查，这样就没有一个合适的 ViewModel 规范帮助开发者实现 MVVM。

在 2017 年的 Google I/O 大会上，Google 推出了 ViewModel 组件，用于规范化 ViewModel 层。ViewModel 是以感知生命周期的形式来存储和管理视图相关数据的。ViewModel 主要有以下特点。

（1）当 Activity 被销毁时，我们可以使用 onSaveInstanceState 方法恢复其数据，这种方法仅适用于恢复少量的支持序列化、反序列化的数据，不适用于恢复大量数据，如用户列表或位图。而 ViewModel 不仅支持恢复大量数据，还不需要序列化、反序列化操作。

（2）Activity/Fragment（视图控制器）主要用于显示视图数据，如果它们也负责数据库或者网络加载数据等操作，那么一旦逻辑过多，会导致视图控制器代码臃肿，ViewModel 可以更容易、更有效地将视图数据相关逻辑和视图控制器分离。

（3）视图控制器经常需要一些时间才可能返回异步调用，视图控制器需要管理这些调用，并在合适的时候清理它们，以确保它们的生命周期不会大于自身，避免内存泄漏。而 ViewModel 恰恰可以避免内存泄漏。

10.8.2　ViewModel 的基本使用方法

1．添加依赖

在前面章节讲过，在一般情况下只需要添加如下代码即可。

```
implementation "android.arch.lifecycle:extensions:1.1.1"
```

由于 Gradle 默认支持依赖传递，添加这一句代码就依赖了很多库，如图 10-9 所示。

图 10-9　依赖的库

如果需要其他特性，比如 kotlin 的支持，那么再另行添加。

2. 自定义 ViewModel

继承 ViewModel，实现自定义 ViewModel，如下所示。

```
import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.MutableLiveData;
import android.arch.lifecycle.ViewModel;

public class MyViewModel extends ViewModel {
   private MutableLiveData<String> name;
   public LiveData<String> getName() {
      if (name == null) {
         name = new MutableLiveData<String>();
         addName();
      }
      return name;
   }
   private void addName() {
      name.setValue("Android进阶解密");
   }
}
```

在 getName 方法中创建一个 MutableLiveData，并通过 MutableLiveData 的 setValue 方法来更新数据。

3. 使用 ViewModel

然后就可以在 Activity 中使用 MyViewModel 了，如下所示。

```
import android.arch.lifecycle.Observer;
import android.arch.lifecycle.ViewModelProviders;
import android.support.annotation.Nullable;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {
   private static final String TAG = "MainActivity";

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
      model.getName().observe(this, new Observer<String>() {
         @Override
         public void onChanged(@Nullable String s) {
               Log.d(TAG, "畅销书: "+s);
         }
      });
   }
}
```

通过 ViewModelProviders 就可以获得 MyViewModel 的实例，然后配合 LiveData 就可以观察 Name 的变化。打印结果如下。

```
D/MainActivity: 书名为Android进阶解密
```

10.8.3　ViewModel 的生命周期

在旋转设备屏幕时，Activity 会被销毁重新创建，而 ViewModel 却不会这样，它的生命周期如图 10-10 所示。

由图 10-10 可以看到，Activity 的生命周期不断变化，经历了被销毁后重新创建，而 ViewModel 的生命周期没有发生变化。

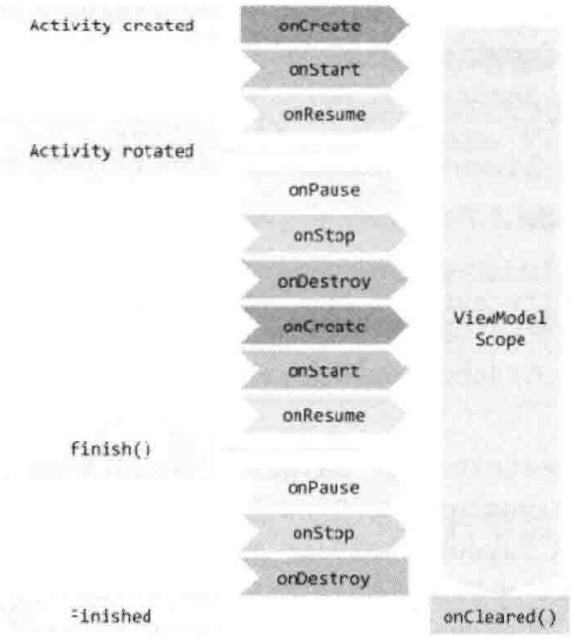


图 10-10　ViewModel 的生命周期

10.9　ViewModel 的原理

要讲解原理，我们首先需要从 10.8.2 节示例中的一个点入手。

```
MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
```

由于是在 Activity 中调用的，因此 this 的值为 Activity，我们还可以在 Fragment 中调用该方法，那么 this 的值为 Fragment，因此 ViewModelProviders.of 方法有多个构造方法，以在 Activity 中调用为例，如下所示。

frameworks/support/lifecycle/extensions/src/main/java/androidx/lifecycle/ViewModelProviders.java

```
@NonNull
@MainThread
public static ViewModelProvider of(@NonNull FragmentActivity activity) {
    return of(activity, null);
}

@NonNull
@MainThread
public static ViewModelProvider of(@NonNull FragmentActivity activity,
        @Nullable Factory factory) {
    Application application = checkApplication(activity);//1
    if (factory == null) {
```

```
            factory = ViewModelProvider.AndroidViewModelFactory.getInstance
(application);//2
        }
        return new ViewModelProvider(activity.getViewModelStore(), factory);
    }
```

注释 1 处内部会调用 activity.getApplication 方法来返回该 Activity 对应的 Application。

注释 2 处的代码用于创建 AndroidViewModelFactory 实例。最后会新建一个 ViewModelProvider，将 AndroidViewModelFactory 作为参数传入，AndroidViewModelFactory 的代码如下所示。

```
    public static class AndroidViewModelFactory extends
ViewModelProvider.NewInstanceFactory {

        private static AndroidViewModelFactory sInstance;
        @NonNull
        public static AndroidViewModelFactory getInstance(@NonNull Application
application) {
            if (sInstance == null) {
                sInstance = new AndroidViewModelFactory(application);
            }
            return sInstance;
        }
        private Application mApplication;
        public AndroidViewModelFactory(@NonNull Application application) {
            mApplication = application;
        }

        @NonNull
        @Override
        public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
            if (AndroidViewModel.class.isAssignableFrom(modelClass)) {
                try {
                    return
modelClass.getConstructor(Application.class).newInstance(mApplication);
                } catch (NoSuchMethodException e) {
                    throw new RuntimeException("Cannot create an instance of " +
modelClass, e);
                } catch (IllegalAccessException e) {
                    throw new RuntimeException("Cannot create an instance of " +
modelClass, e);
                } catch (InstantiationException e) {
                    throw new RuntimeException("Cannot create an instance of " +
modelClass, e);
```

```
                } catch (InvocationTargetException e) {
                    throw new RuntimeException("Cannot create an instance of " +
modelClass, e);
                }
            }
            return super.create(modelClass);
        }
    }
```

AndroidViewModelFactory 是一个单例，在讲这个类之前，需要知道 ViewModel 类本身是一个抽象类，通过继承 ViewModel 来实现自定义 ViewModel，那么 AndroidViewModelFactory 的 create 方法的作用就是通过反射生成 ViewModel 的实现类的。

继续来看 ViewModelProvider.get 方法。

frameworks/support/lifecycle/viewmodel/src/main/java/androidx/lifecycle/ViewModelProvider.java

```
@NonNull
    @MainThread
    public <T extends ViewModel> T get(@NonNull Class<T> modelClass) {
        String canonicalName = modelClass.getCanonicalName();//1
        if (canonicalName == null) {
            throw new IllegalArgumentException("Local and anonymous classes can not
be ViewModels");
        }
        return get(DEFAULT_KEY + ":" + canonicalName, modelClass);//2
    }
    @NonNull
    @MainThread
    public <T extends ViewModel> T get(@NonNull String key, @NonNull Class<T>
modelClass) {
        ViewModel viewModel = mViewModelStore.get(key);//3

        if (modelClass.isInstance(viewModel)) {
            return (T) viewModel;
        } else {
            if (viewModel != null) {
            }
        }

        viewModel = mFactory.create(modelClass);
        mViewModelStore.put(key, viewModel);
        return (T) viewModel;
    }
```

注释 1 处得到类的名称，对这个名称进行字符串拼接，作为注释 2 处方法的参数，DEFAULT_KEY 的值为 "androidx.lifecycle.ViewModelProvider.DefaultKey"。

因此，注释 3 处的 key 值实际上就是"androidx.lifecycle.ViewModelProvider.DefaultKey"+类名。根据这个 key 值从 ViewModelStore 获取 ViewModel（ViewModel 的实现类）。如果 ViewModel 能转换为 modelClass 类的对象，则直接返回该 ViewModel。否则会通过 Factory 创建一个 ViewModel，并将其存储到 ViewModelStore 中。这里的 Factory 指的是 AndroidViewModelFactory，它在 ViewModelProvider 创建时作为参数传进来。

到此，我们已经知道 ViewModel 的实现类是如何创建的了。

当创建完 ViewModel 的实现类后，如同 10.8.2 节，我们还会调用如下代码来观察 name 的变化。

```
MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
    model.getName().observe(this, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            Log.d(TAG, "畅销书: "+s);
        }
    });
```

model.getName()会返回一个 MutableLiveData，接着调用了 MutableLiveData 的 observe 方法，这部分内容在 10.7 节中讲过，这里就不再赘述。

10.10　本章小结

随着 Android Jetpack 架构组件的发展，这一知识点已经是 Android 应用开发势必要掌握的了，但是 Android Jetpack 架构组件内容很多，足够写一本书了，这里只介绍了比较常用的 Lifecycle、LiveData、ViewModel 的使用，在现阶段，一个高级工程师仅了解它们的使用是不够的，本章还介绍了它们的原理。至于其他架构组件，由于不常用或者发展的不成熟，本章没有对其进行介绍，读者可以阅读官方文档和相关的博客。

第 11 章

跨平台技术演进

关联章节：

第 13 章 Flutter 基础

我们在了解 Flutter 前，不妨了解一下跨平台技术的演进，这样更有助于学习 Flutter，也能认清 Flutter 的优势和本质。

在 2010 年了解 Android 和 iOS 开发的人很少，所有人都在“摸着河底过河”，项目更没有第三方框架，大都是自己写的，不像现在各种框架很多。随着移动开发的发展，互联网公司也层出不穷，有些公司迫于竞争，想要更迅速、更省成本的进行开发，就不再满足 Android 端一套代码和 iOS 端一套代码了。与此同时，其他技术领域和各大公司也纷纷推出相关的技术，这样跨平台技术应运而生，并且开始在互联网公司中生根发芽。

Android 和 iOS 生态太大了，可以把它们比作第一级生态，曾经想要颠覆这两个系统的都失败了，因此建立次级生态是比较稳妥的策略，Android 平台更加开放，因此次级生态的中心就是 Android，次级生态的形式多种多样，比如在 Android 系统的基础上建立自己的生态，再或者推出各种跨平台技术建立生态。跨平台技术产生的框架实在太多了，很多还没等我们去了解，它们就没落了，这些框架成为跨平台技术发展的一个过渡产物。跨平台技术的产物是不靠谱还是趋势，学习了本章内容我们会有自己的理解。

跨平台技术的分类没有标准的答案，这里把它们为 5 种，分别为 Web App、Hybrid App、语言编译转换、原生渲染、自绘 UI。下面分别对其进行介绍。

11.1　Web App

Web App 是指基于 Web 的应用，其运行于网络和标准浏览器上，相当于一个网页加一个 App 的壳。2014 年 HTML5 的标准规范制定完成，在网络舆论上 Web App 大有取代 Native App 的气势，但 Web App 有以下缺点。

- 性能低，操作体验不好。
- 无法调用原生 API，很多功能无法实现。
- 依赖于网络，网速慢时体验很差，并且没有离线功能，优化不好会消耗流量。
- 只能作为一个临时的入口，用户留存率低。

在 Web App 的基础上，又出现了几个进化者，这里主要介绍 PWA。

PWA（Progressive Web App）意为渐进式增强 Web 应用，它不是一门技术，而是一个概念，使用以下多种技术来增强 Web App 的功能。

- 用 Service Worker + HTTPS +Cache API + indexedDB 等一系列 Web 技术实现离线加载和缓存。
- 实现了推送和通知。
- 可以直接添加到手机桌面上。
- 使用 Service Worker 可以进行后台同步。

总结起来，PWA 的主要能力就是离线、推送、桌面访问，可以说 PWA 赋予 Web App 原生的体验，但是 PWA 一直不温不火的原因主要有以下几点。

- 浏览器对 PWA 技术支持还不够全面，不是每一款浏览器都能 100%的支持 PWA。
- 国内一些手机厂商对 Android 系统各种魔改，对 PWA 的兼容性不好，甚至不支持 PWA。
- 平台的竞争。iOS 对 PWA 的支持力度远远低于 Android，所以 PWA 在 iOS 上的体验打了折扣。PWA 在面对类似的微信小程序和快应用的竞争中，并没有优势。

11.2　Hybrid App

除了采用原生和 Web 开发 App，还可以采用 HTML5+原生来进行混合开发，这就是 Hybrid。

关于 Hybrid 的诞生有一个小故事，某二线互联网公司的 App 是以原生为主的，随着应用越来越复杂，终于有一天发现原生有一个方法最大数限制，一些页面需要内嵌 HTML5 的页面，于是原生+HTML5 团队一起做了第一个 Hybrid 项目，这一套代码兼容三端并且效率很高，因此 Hybrid App 就成了这个公司的主流，业界其他公司也纷纷效仿。

原生 App 的架构图如图 11-1 所示。

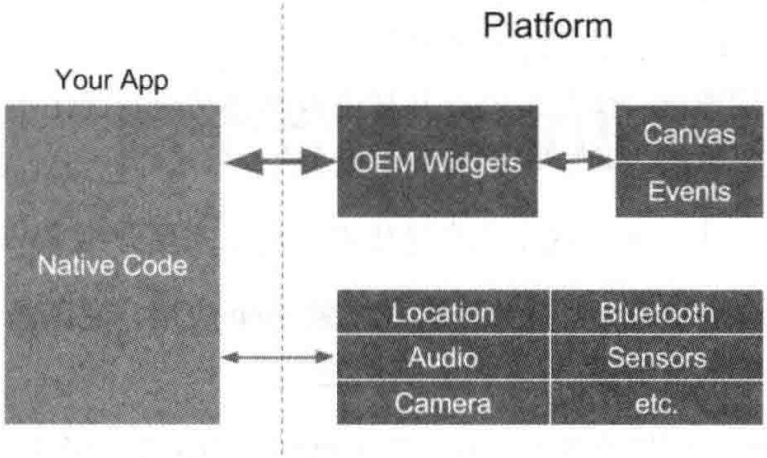


图 11-1　原生 App 的架构图

通过原生 SDK 提供的 API，App 可以与系统底层通信，以创建 UI 组件或访问系统服务。这些组件被渲染到手机屏幕，屏幕产生的相应的事件会被传回给组件。因为每个平台的系统组件是不同的，需要为每个平台开发单独的 App，而 Hybrid App 不必这样，Hybrid App 的原生 UI 组件用来展示交互复杂和渲染要求高的界面，其他可以交给 HTML5 来展示。

Hybrid App 虽然开发效率高，可以跨平台开发，但是 Hybrid 体验比不上原生，对于需要快速试错、快速占领市场的团队，Hybrid App 是一个不错的选择，后期团队稳定下来后，最好还是要做体验更好的原生 App 或者使用其他体验更好的跨平台技术。

Hybrid 相关的技术有很多，比如 PhoneGap、Cordova、Ionic、VasSonic 等，我们大概来了解这些相关的技术。

11.2.1　Cordova

讲到 Cordova，首先就要了解它的前身 PhoneGap，PhoneGap 面向 Web 开发人员，通过使用 HTML、CSS 和 Javascript 构建跨平台 App。2011 年，Apache 收购了 Nitobi Software 和它的 PhoneGap 产品，并对 PhoneGap 进行开源，PhoneGap 2.0 版本时，产品更名为 Apache Cordova。目前 Cordova 支持的平台有 Android、iOS、Windows、OS X、Electron。

Cordova 的体系结构图如图 11-2 所示。

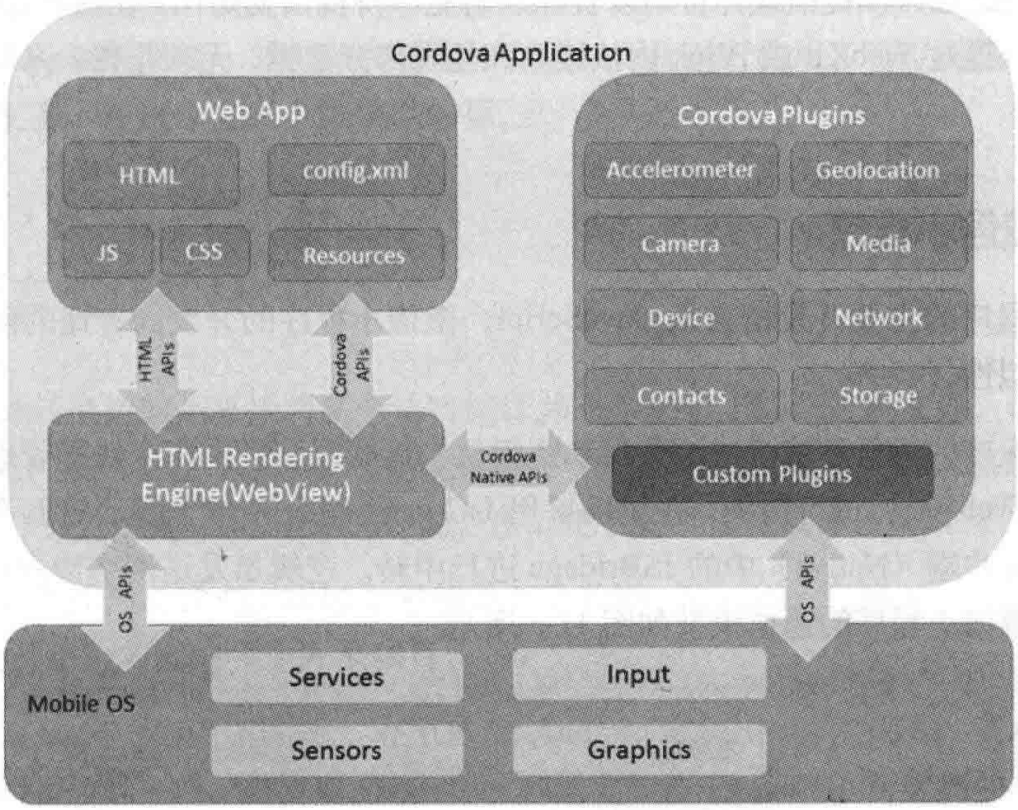


图 11-2　Cordova 的体系结构图

Cordova 同样使用 WebView 来展示界面，插件是 Cordova 中不可或缺的一部分，Apache Cordova 维护了 Core Plugins 插件，这些核心插件为 App 提供访问设备功能，如电池、相机、联系人等。除了核心插件，还有一些第三方插件可以使用，我们也可以开发一个自己的插件。

11.2.2　Ionic

Ionic Framework 是一个开源 UI 工具包，最早的目标是使用 HTML、CSS 和 JavaScript 等 Web 技术开发移动应用程序。由于 Web 技术的这一基础，Ionic 可以在网络运行的任何地方运行，比如 iOS、Android、浏览器、Electron、PWA 等。

目前，Ionic Framework 已与 Angular 正式集成，但对 Vue 和 React 的支持正在开发中。

11.2.3　VasSonic

VasSonic 是由腾讯 VAS 团队开发的轻量级高性能混合框架，其目的是加速在 Android 和 iOS 平台上运行的 H5 首屏。VasSonic 不仅支持服务器呈现的静态网站或动态网站，而

且还完美兼容 Web 离线资源。VasSonic 使用自定义的 url 连接而不是原始网络连接来请求索引 html，因此它可以提前或并行请求资源以避免等待视图初始化。在这种并行的情况下，VasSonic 可以通过 WebKit 或 Blink 内核读取和呈现部分数据，无须花费太多时间等待数据流的结束。

11.2.4 微信小程序

微信小程序的主要开发语言是 JavaScript，微信小程序的开发同普通的网页开发相比有很大的相似性。

微信小程序的运行环境分成渲染层和逻辑层，这两层分别由两个线程管理，渲染层的界面使用了 WebView 进行渲染，逻辑层采用 JsCore 线程运行 JS 脚本。这两个线程的通信会经由微信客户端（Native）中的 JSBridage 进行中转。逻辑层发送网络请求也经由 Native 进行转发，微信小程序的通信模型如图 11-3 所示。

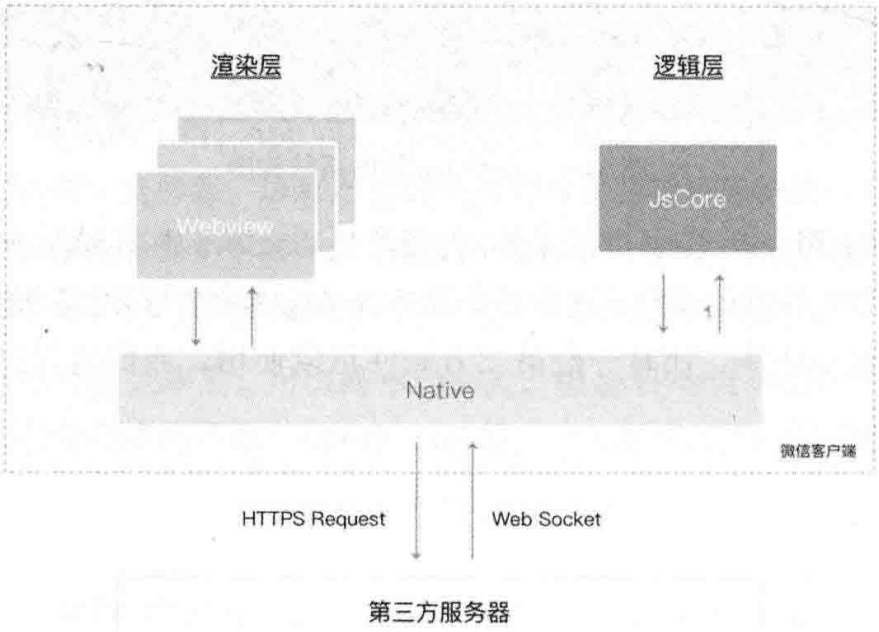


图 11-3　微信小程序的通信模型

其中，WXML 模板和 WXSS 样式工作在渲染层，JS 脚本工作在逻辑层。

微信小程序和 PWA 都是基于 Web 技术的，二者原理的区别是微信小程序类似 Hybrid 架构，WebView 渲染基本的网页内容，对渲染性能要求较高的组件，通过原生组件来实现，比如相机、视频、地图等，另外传统 Web 无法访问的本地能力，需要通过 JS SDK 来实现，而 PWA 则使用多种技术增强 Web 能力以达到接近 Native 应用的体验。

微信小程序本身和 App 就不是竞争关系，更多的是一个推广渠道，微信小程序更像是一张海报，用于快速推广倒流，而 App 则是要推广的对象。微信小程序的缺点很明显，体验上无法跟 App 相提并论，功能依托受限于微信，无法进行拓展。可以说微信小程序就是建立了次级生态，在这个生态中微信说的算。

11.3　语言编译转换

语言编译转换指的是直接将某个语言编译为一个平台下的二进制文件。比较有名的是 Xamarin 框架，虽然它在 Android 平台是内嵌了 Mono 虚拟机来实现的，但是在 iOS 平台下是以 AOT 的方式编译为二进制文件的，所以把它归到语言编译转换类型。

Xamarin 始创于 2011 年，2016 年被微软正式收购。Xamarin 是 Mono 项目的一个分支，基于.NET 的跨平台实现的一个开源项目。

与 PhoneGap 等框架不同的是，在 iOS 和 Android 刚推出新的功能时，Xamarin 可以在第一时间调用相应的 API，而使用 PhoneGap 则需要等待 PhoneGap 封装新的功能后才可以调用相应的 API。

Xamarin 的 Android 实现原理如图 11-4 所示。

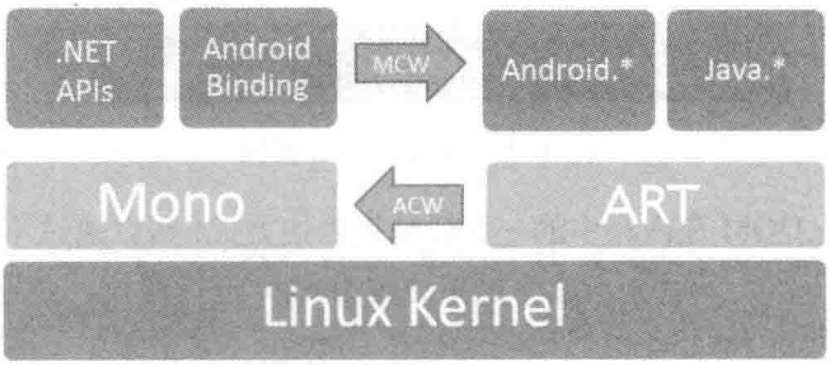


图 11-4　Xamarin 的 Android 实现原理

C#代码写的 Android 应用运行在 Mono 虚拟机中，ART 可以通过 ACWs（Android Callable Wrappers）的方式执行到 Mono 中的 C#代码。如果 C#代码想要调用系统功能或者 Java 的实现类库，那么可以借助 MCW（Managed Callable Wrapper）的方式来实现。MCW 是 JNI 的桥梁，可以使用托管代码调用 Android 代码。

11.4 原生渲染

原生渲染在本节中指的是由 JavaScript 开发并且由原生控件渲染的，其代表有 React Native、Weex、快应用。

11.4.1 React Native

Facebook 曾在移动端举步维艰，他们认为可以不借助任何原生开发手段来实现 Facebook 的移动应用，因此在早期选择了 HTML5，后来发现 HTML5 的效率始终无法和原生相比，因此在 2015 年发布了 React Native。

React Native 是 Facebook 早先开源的 Web UI 框架 React 在原生移动应用平台的衍生产物，底层对 Android 和 iOS 平台的原生代码进行封装，通过使用 JavaScript 就可以编写出原生代码。React Native 架构如图 11-5 所示。

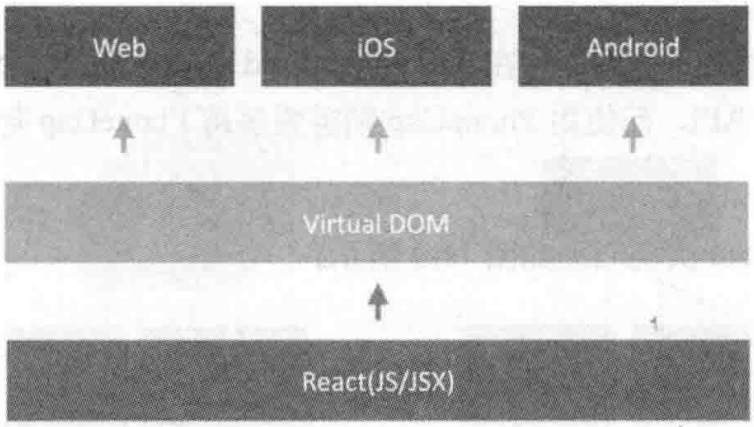


图 11-5　React Native 架构

Virtual DOM 是 DOM 在内存中的一种轻量级表达方式，可以通过不同的渲染引擎生成不同平台下的 UI。React Native 与原生框架通过 Bridge 进行通信，如果使用 Chrome 浏览器进行调试，那么所有 JavaScript 代码将运行在 Chrome V8 引擎中，通过 WebSocket 和原生代码进行通信。

11.4.2 Weex

Weex 是阿里开源的一款跨平台移动开发工具，它能够完美兼顾性能与动态性，让移动开发者通过简捷的前端语法写出原生级别的性能体验，并支持 iOS、Android、YunOS 及 Web 等多端部署。目前 Vue.js 和 Rax 这两个前端框架被广泛应用于 Weex 页面开发，因

此 Weex 支持 Vue 语法和 Rax 语法，而 React Native 只支持 JSX 语法。Weex 架构如图 11-6 所示。

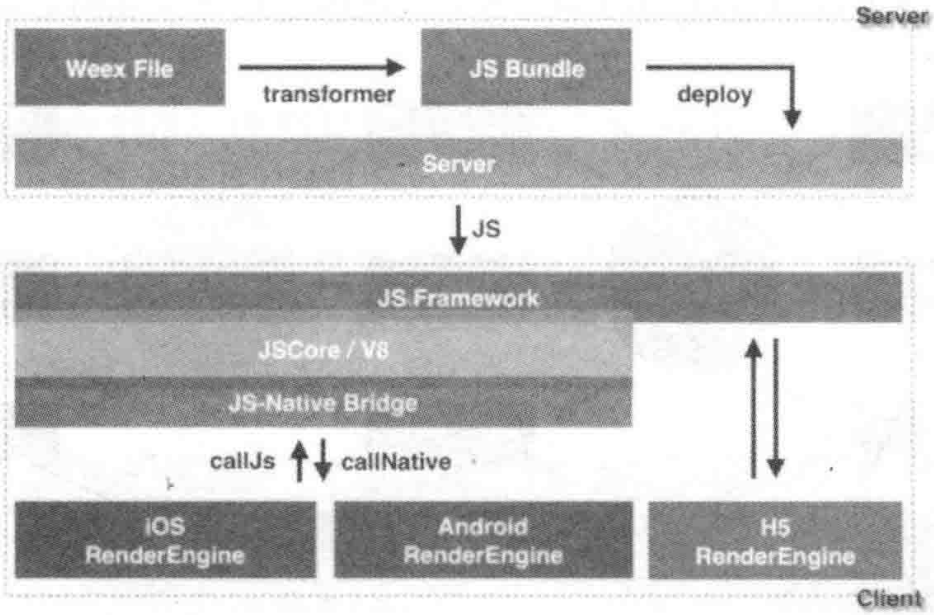


图 11-6　Weex 架构

Weex 首先将编写的 Weex 源码通过 transformer 转换成 JS Bundle。然后将 JS Bundle 部署在服务器，当接收到终端（Android、Web 端、iOS 端）的 JS Bundle 请求时，将 JS Bundle 下发给终端。在终端中，由 Weex 的 JS Framework 接收和执行 JS Bundle 代码，并且执行数据绑定、模板编译等操作，然后输出 JSON 格式的 Virtual DOM，JS Framework 将渲染指令发送给 Native，提供 callNative 和 callJS 接口，以方便 JS Framework 和 Native 的通信。

11.4.3　快应用

2018 年 3 月份，由小米、OPPO、vivo、华为等 10 家国内主流厂商成立了快应用联盟。快应用介于移动网页和原生应用之间，第三方应用以移动网页的形式进行开发，最终得到原生渲染的效果体验。快应用框架深度集成进各手机厂商的手机操作系统中，可以在操作系统层面形成用户需求与应用服务的无缝连接，很多只能在原生应用中才能使用的功能，在快应用中可以很方便的实现，享受原生应用体验，同时不用担心分发留存等问题，资源消耗也比较少。对于每台手机设备，应用可以从多个系统入口，引导用户体验产品。

快应用与 React Native 和 Weex 相比主要有两点不同。

- 快应用自身不支持 Vue 或 React 语法，它采用的是 JavaScript 开发。
- React Native 和 Weex 的渲染引擎是集成到框架中的，每一个 App 都需要打包一份，

其安装包体积较大；快应用渲染引擎是集成到 ROM 中的，应用中无需打包，其安装包体积小。

和微信小程序类似，快应用本质上也是要建立次级生态的，快应用的架构如图 11-7 所示。

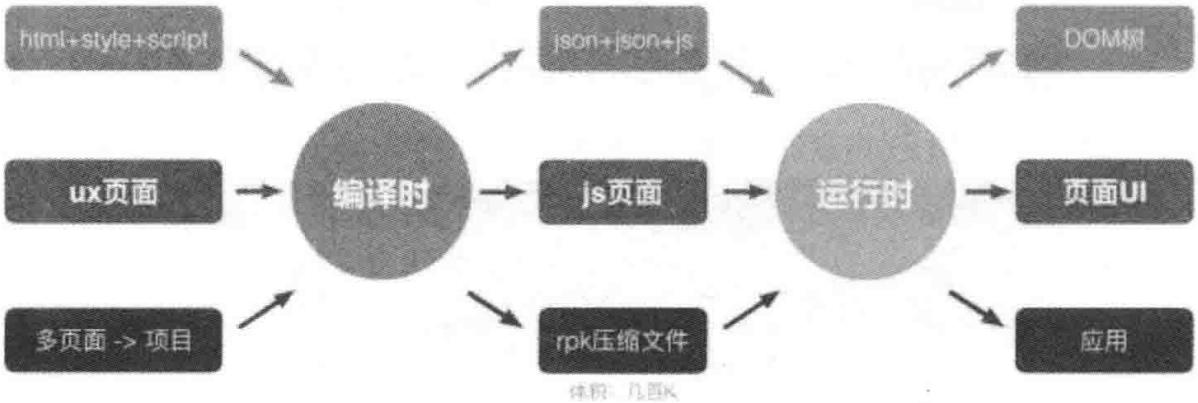


图 11-7　快应用的架构

快应用实现分为编译时和运行时两个方面，UX 页面源码经过编译时得到 JS，然后经过运行时得到界面 UI。每一个页面由 HTML+CSS+JS 组成，编译运行后得到内存中的 DOM 树。多个页面组成一个项目，编译后得到 RPK 文件，最终运行时以应用形态呈现。

快应用推出后不温不火，面对微信小程序，快应用在流量和入口等关键数据方面都无法与小程序匹敌，其未来发展堪忧。

11.5　自绘 UI

自绘 UI 指的是通过在不同平台实现一个统一接口的渲染引擎来绘制 UI，而不依赖系统平台的原生控件，这样可以保证不同平台 UI 的一致性。不用像 React Native 一样，随着不同平台系统版本的变化，开发者还需要处理不同平台的差异，甚至有些特性只能在单个平台上实现，这样无法保证不同平台 UI 的一致性。自绘 UI 框架的代表有 Qt 和 Flutter。

11.5.1　Qt

Qt 产生的时间很早，Qt 第一版于 1991 年由 Trolltech 发布。后来在 2008 年，Nokia 斥资 1.5 亿美元收购 TrollTech，将 Qt 应用于 Symbian 程序开发。2012 年 8 月，Nokia 将 Qt 以 400 万欧元的价格出售给 Digia。2016 年 Qt Group Plc 从 Digia 分拆出来，2014 年 Qt

开始支持移动端的 Android、iOS、Window Phone 平台。虽然 Qt 在 PC 领域发展良好，但是在移动端表现不佳，很少有人提及或者用 Qt 去开发移动端。

11.5.2　Flutter

Flutter 是 Google 的移动 UI 框架，可以快速在 Android 和 iOS 上构建高质量的原生用户界面，它的前身是 Google 试验项目 Sky。

Futter 提出了一切皆 Widget 的概念，除了基本的文本、图片、卡片、输入框，布局方式和动画等也都是由 Widget 组成的。通过使用不同类型的 Widget，就可以实现界面 Flutter 框架，如图 11-8 所示。

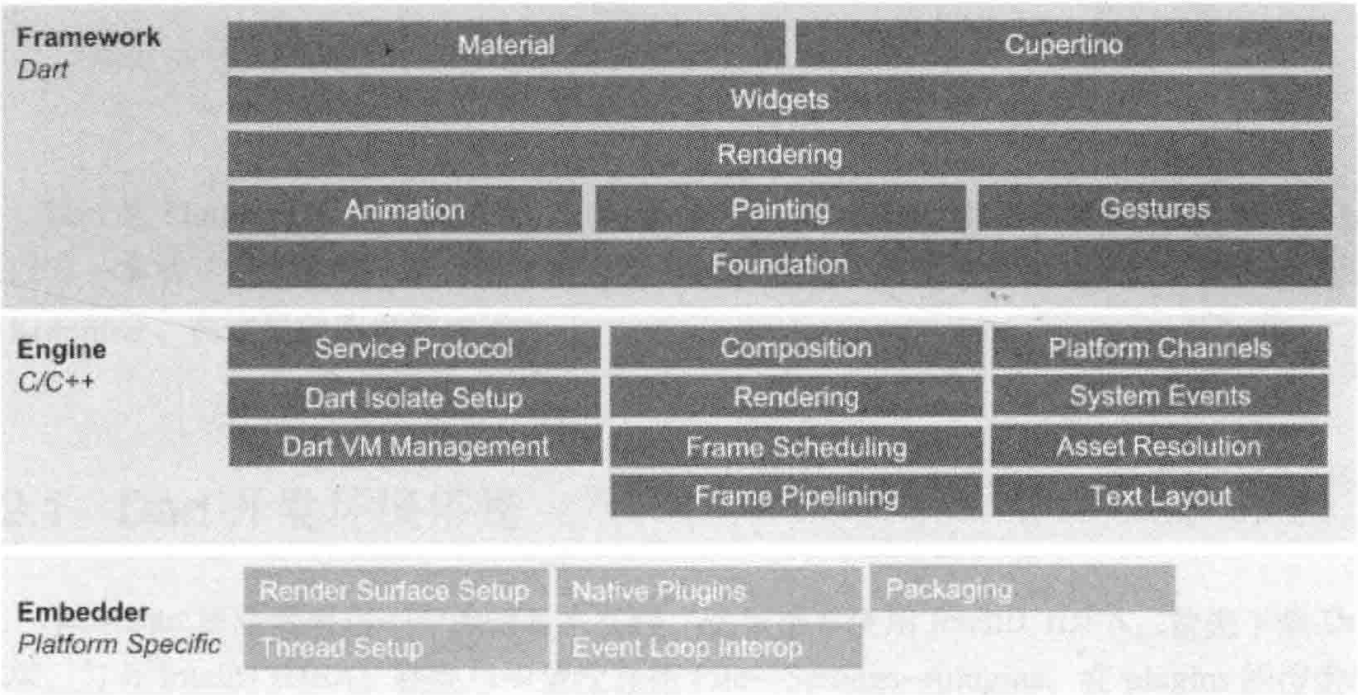


图 11-8　Flutter 框架

由图 11-8 可知，Flutter 框架采用了分层设计，此设计的目的是帮助开发者使用更少的代码完成更多的工作。例如，Material 层是由 Widgets 层的普通 Widget 组成的，而 Widgets 层本身是通过来自 Rendering 层的低级对象构建的。

目前在 Flutter 基础上开发的框架已经出现了，这也证明了业界普遍开始认可 Flutter，并开始进行尝试。

11.6 本章小结

跨平台技术的分类没有标准的答案，这里也只是粗略进行分类，并对每个分类的主流框架进行介绍，实际上还有很多框架没有提到，它们其中有的没落了，有的缺点明显难以使用，再就是大公司的 KPI 产物。跨平台技术的演进如同百家争鸣，极大地促进了跨平台技术的发展。在我看来，这些技术可以使不同技术分支的程序员都可以参与到移动开发中，并享受移动开发的乐趣，从这个角度来看这些跨平台技术的优劣之分是很难去评判的。

第 12 章

Dart 基础

Dart 是 Flutter SDK 指定的语言，因此要学习 Flutter，Dart 是必须要掌握的。关于 Dart 可以写一本书了，这里用一章的内容来介绍 Dart 的精髓，带领读者快速入门。和 Java 语言类似的部分，在这里就不再赘述了。

12.1 Dart 开发环境搭建

学习 Dart 语法需要用一个编辑器来实践，这里推荐使用 IntelliJ IDEA。首先下载 Dart SDK。打开 IntelliJ IDEA，在菜单中依次选择 File→Settings→plugins，在 plugins 的搜索框中搜索 Dart 并安装，然后重启 IntelliJ IDEA。依次选择 File→New Project→Dart，按照图 12-1 配置 Dart SDK。

注意要选择 Console Application，否则会默认创建一个 Web 项目。单击 Next 按钮然后配置项目的名称就可以创建项目了。

图 12-1　配置 Dart SDK

在项目中的 bin/main.dart 中加入如下测试代码。

```
void main() {
  print("Hello World");
}
```

选择菜单中的 Run→Run'main.dart'或者单击工具条的运行图标，就能在控制台看到输出的结果，如图 12-2 所示。

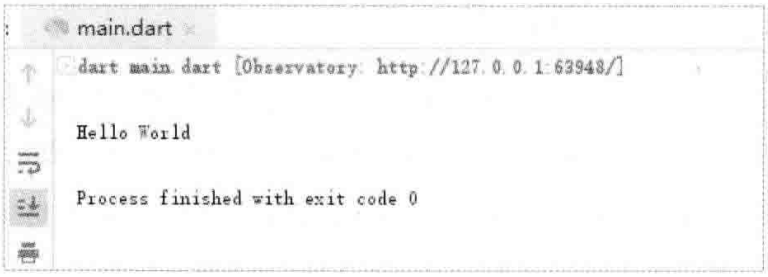


图 12-2　控制台输出的结果

12.2　Dart 概述

Dart 是 Google 开发的计算机编程语言，亮相于 2011 年 10 月，最新的版本是 Dart 2。Dart 诞生的原因是 Google 的工程师出于对 JavaScript 的不满意，诞生的初期也赢得了部分

前端开发者的青睐，但是这时 JavaScript 借着 NodeJS 火了起来，在前端、后端、移动端无孔不入，Dart 就渐渐被人遗忘了，其实 Dart 本身是具有很强的实力的，只是不走运。

Google 并没有放弃 Dart，不遗余力地推广 Dart，Google 的 Angular 提供了 Dart 版本，指定 Dart 为新系统 Fuchsia 的官方开发语言，并指定 Dart 为移动 UI 框架 Flutter 的开发语言，因此 Dart 又重新回到了人们的视野中。

在通常情况下，Dart 运行在 DartVM 上，但是在特定情况下它也可以编译成本地代码运行在硬件上，比如 Flutter 会将代码编译成指定平台的本地代码来提高性能。

12.3　Dart 的特性和重要概念

Dart 的特性主要有以下几点。

（1）执行速度快，Dart 是 AOT(Ahead Of Time)编译的，可以编译成快速的、可预测的本地代码，这使得 Flutter 几乎都可以使用 Dart 来编写，也可以采用 JIT（Just In Time）编译。

（2）易于移植，Dart 可编译成 ARM 和 X86 代码，这样 Dart 可以在 Android、iOS 和其他地方运行。

（3）容易理解，Dart 充分吸收了高级语言特性，如果你已经熟悉 C++、C、Java，那么可以在短短几天内用 Dart 来开发。

（4）易于阅读，Dart 使 Flutter 不需要单独的声明式布局语言（XML 或 JSX），或者单独的可视化界面构建器，这是因为 Dart 的声明式编程布局易于阅读。

（5）避免抢占式调度，Dart 可以在没有锁的情况下进行对象分配和垃圾回收，和 JavaScript 一样，Dart 避免了抢占式调度和共享内存，因此不需要锁。

Dart 的重要概念有以下几点。

（1）在 Dart 中，一切都是对象，每个对象都是一个类的实例，所有对象都继承自 Object。

（2）Dart 在运行前解析所有代码，指定数据类型和编译时常量，可以使代码运行更快。

（3）与 Java 不同，Dart 不具备关键字 public、protected、private。如果一个标识符以下画线开始，那么它和它的库都是私有的。

（4）Dart 支持顶级的函数如 main()，也支持类或对象的静态和实例方法，还可以在函数内部创建函数。

（5）Dart 支持顶级的变量，也支持类或对象的静态变量和实例变量，实例变量有时被称为字段或属性。

（6）Dart 支持泛型类型，如 `List<int>`（整数列表）或 `List<dynamic>`（任何类型的对象列表）。

（7）Dart 工具可以报告两种问题：警告和错误。警告只是说明代码可能无法正常工作，但不会阻止程序执行。错误可以是编译时的，也可以是运行时的，编译时错误会阻止代码执行；运行时错误会导致代码执行时报出异常。

12.4 Dart 关键字

Dart 的关键字如表 12-1 所示。

表 12-1 Dart 关键字

abstract	dynamic	implements	show
as	else	import	static
assert	enum	in	super
async	export	interface	switch
await	extends	is	sync
break	external	library	this
case	factory	factory	factory
catch	false	new	true
class	class	null	try
const	finally	on	typedef
continue	for	operator	var
covariant	Function	part	part
default	get	rethrow	while
deferred	hide	return	with
do	if	set	set

12.5　变量

变量声明使用 var 关键字，未初始化的变量的初始值为 null，即便是数字类型的变量也是 null。

```
var name = 'liuwangshu';
```

name 变量的类型被推断为 String，也可以显示声明。

```
String name = 'liuwangshu' ;
```

如果对象不限于单一类型，那么可以指定 Object 或 dynamic。

```
Object name = 'liuwangshu' ;
```

如果定义的变量不会变化，那么可以使用 final 或 const 来代替 var，final 变量只能设置一次。

```
final name = 'liuwangshu'
//name = 'zhangwuji' ; //会报错
```

const 变量为编译时常量，如果 const 变量在类级别，那么可以使用 static const。

```
const pi = 3.1415926;
const area = pi * 60 * 60;
```

不仅可以使用 const 定义常量，也可以使用 const 创建常量的值。

```
var foo = const []; final bar = const []; const baz = [];//相当于'const []'
```

12.6　基本数据类型

Dart 的基本数据类型包括 Number、String、Boolean、List、Set、Map。

12.6.1　Number

Number 类型分为以下两类。

- int：整数值不大于 64 位，具体取决于平台。在 Dart VM 上，值可以是-2 ^63 到 2 ^63-1，如果编译为 JavaScript，那么允许值为-2^53 到 2^53-1。
- double：64-bit（双精度）浮点数，符合 IEEE 754 标准。

12.6.2 String

Dart 字符串是 UTF-16 编码的字符序列，可以使用单引号或者双引号来创建字符串。

```
var s1 = '单引号适用于字符串文字';
var s2 = "双引号同样有效";
```

可以在字符串中使用表达式，用法是：${expression}。如果表达式是一个标识符，那么可以省略 {}。

```
var s = '乾坤大挪移';
assert('张无忌的$s' ==
      '张无忌的乾坤大挪移');
```

使用三个单引号或者三个双引号可以创建多行字符串对象。

```
var s1 = '''
第一行
第二行
''';

var s2 = """第一行
第二行""";
```

12.6.3 Boolean

Dart 是强 bool 类型检查，只有 true 对象才被认为是 true。

```
var name = '张无忌';
if (name) {
  print('明教教主');
}
```

上面的代码编译不能通过，因为 name 是一个字符串，而不是 bool 类型。

12.6.4 List

下面是一个 List 的示例。

```
var list = [1, 2, 3];
```

List 的第一个元素的索引是 0，最后一个元素的索引是 list.length - 1，如下所示。

```
var list = [1, 2, 3, 4, 5, 6];print(list.length);
print(list[list.length-1]);
```

12.6.5　Set

Dart 中的 Set 是一组无序的集合，如下所示。

```
var hero = ['张无忌', '风清扬', '张三丰', '独孤求败', '萧峰'];
```

要创建一个空集，可以在{}前面带有类型参数，如下所示。

```
var heros= <String> {};
```

使用 add()或 addAll()方法将条目添加到现有集中，如下所示。

```
var heros = <String>{};
heros.add('石破天');
heros.addAll(hero);
```

12.6.6　Map

Map 是一个键值对相关的对象，键和值可以是任何类型的对象，每个键都是唯一的，而一个值可以出现多次，如下所示。

```
var player= {
  '20' : '斯诺',
  '3': '艾弗森',
  '40' : '希尔',
  '8' : '麦基',
  '55' : '穆托姆博'
};
```

使用 Map 构造函数也可以实现同样的功能，如下所示。

```
var player = new Map();
player['20'] = '斯诺';
player['3'] = '艾弗森';
player['40'] = '希尔';
```

12.7　函数

Dart 是一个真正面向对象的语言，函数属于 Function 对象。这意味着，函数可以赋值给变量，也可以当作其他函数的参数，如下所示。

```
void printName(String name) {
```

```
  print('name is $name');
 }
```

12.7.1 可选参数

可选参数可以是可选位置参数，也可以是可选命名参数，但二者不能同时使用。

1. 可选命名参数

调用方法的时候，可以使用 paramName: value 的形式来指定参数的名称，这样就可以根据 paramName 得知参数的含义，提高代码的可读性，如下所示。

```
coffeeFlavor (sugar : true , sugar : false );
```

定义函数时，使用{param1, param2, ...}的形式来指定命名参数，如下所示。

```
coffeeFlavor ({bool sugar , bool sugar}) {

}
```

2. 可选位置参数

把函数的参数放到 [] 中就变成可选位置参数了，如下所示。

```
String go(String to, [String who]) {
  var result = 'go to the $to';
  if (who != null) {
    result = '$result with $who';
  }
  return result;
}
```

12.7.2 默认参数值

可以使用 = 来定义可选参数的默认值， 默认值必须是编译时常量。 如果没有提供默认值，则默认值为 null，如下所示。

```
String go(String to, [String who= 'liuwangshu']) {
  var result = 'go to the $to';
  if (who != null) {
    result = '$result with $who';
  }
  return result;
}
```

```
String result= go ('beijing');
```

12.7.3　main 函数

每个应用都需要有一个顶级的 main 函数来作为入口才能执行。main()函数的返回值为 void 并且有个可选的 List<String> 参数。此前我们列举的示例都是在 main()函数中运行才得已验证的，如下所示。

```
void main(){
 void printName(String name) {
  print('name is $name');
 }
 printName('liuwangshu');
}
```

12.7.4　匿名函数

大部分函数都有名字，例如 main 或者 printElement。可以创建没有名字的匿名方法，格式如下所示。

```
([[Type] param1[, …]]) {
 codeBlock;
};
```

下面的代码定义了一个参数为 i（该参数没有指定类型）的匿名函数。 list 中的每个元素都会调用这个函数并打印出来，如下所示。

```
var list = ['张无忌', '风清扬', '张三丰', '独孤求败', '萧峰'];
list.forEach((i) {
 print(list.indexOf(i).toString() + ': ' + i);
});
```

12.8　流程控制语句

Dart 的流程控制语句如下。

- if 和 else
- for 循环
- while 和 do-while 循环

- break 和 continue
- switch 和 case
- assert

这些语句大部分都和 Java 类似，这里主要讲解 for 循环和 switch 语句。

12.8.1 for 循环

标准的 for 循环，如下所示。

```
var message = new StringBuffer("张无忌");
for (var i = 0; i < 3; i++) {
  message.write('!');
}
```

List 和 Set 等实现了 Iterable 接口的类还支持 for-in 形式的遍历，如下所示。

```
var hero = ['张无忌', '风清扬', '张三丰', '独孤求败', '萧峰'];
for (var h in hero) {
  print(h);
}
```

12.8.2 switch 和 case

Dart 中 switch 语句通过使用 == 来比较整型、字符串或者编译时常量。被比较的对象必须都是同一个类的实例（不能是其子类），并且这个类不允许覆写 ==。另外，枚举类型很适合在 switch 语句中使用，如下所示。

```
String today='Friday';
switch(today){
  case 'Monday':
    print('星期一');
    break;
  case 'Friday':
    print('星期五');
    break;
}
```

12.9　捕获异常

捕获异常可以避免异常继续传递，如下所示。

```
try {
  ...
} on OutOfLlamasException {
  ...
} on Exception catch (e) {
  print('Unknown exception: $e');
} catch (e) {
  print('Something really unknown: $e');
}
```

使用 on 或者 catch 来声明捕获语句，也可以同时使用。其中 on 用来指定异常类型，catch 用来捕获异常对象。

确保某些代码不管是否出现异常都会执行，可以使用 finally 语句来实现，如下所示。

```
try {
  ...
} catch(e) {
  print('Error: $e');
} finally {
  ...
}
```

12.10　为类添加新的功能

Dart 是一个面向对象的编程语言，支持基于 Mixin 的继承机制。Mixin 可以理解为多继承，在 with 关键字的后面为一个类或者多个类，如下所示。

```
class Person{
  run(){
    print('跑');
  }
}

class Wushu{
  use(){
  print('乾坤大挪移');
```

```
  }
}

class Zhangwuji extends Person with Wushu{
int age;
Zhangwuji(int age){
  this.age=age;
 }
}

void main() {
  var zhangwuji=new Zhangwuji(30);
  zhangwuji.run();
  zhangwuji.use();
}
```

通过上述代码的验证，Zhangwuji 类拥有了 Person 和 Wushu 这两个类的方法。

12.11 库的使用方法

使用 import 来引入一个库，对于 Dart 语言内置的库，使用 dart: scheme。 对于第三方的库，可以使用文件系统路径或者 package: scheme，如下所示。

```
import 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

1. 指定库前缀

如果导入的两个库的名字有冲突，则可以使用库的前缀来进行区分。例如，如果 library1 和 library2 都有一个名字为 Element 的类，则可以这样使用，如下所示。

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
...
Element element1 = new Element();              //使用 lib1 中的 Element
lib2.Element element2 = new lib2.Element();    //使用 lib2 中的 Element
```

2. 导入库的一部分

如果只使用库的一部分功能，则可以选择需要导入的部分内容。其中 show 代表只导入

指定的部分，hide 代表除了指定的部分都导入，如下所示。

```
// 只导入 foo
import 'package:lib1/lib1.dart' show foo;

// 除了 foo，其他部分都导入
import 'package:lib2/lib2.dart' hide foo;
```

3. 延迟加载库

延迟加载库意味着应用程序可以在需要的时候再加载库，使用延迟加载库的场景主要有以下几点。

- 减少 App 的初始启动时间。
- 执行 A/B 测试，例如尝试各种算法的不同实现。
- 加载很少使用的功能。

要延迟加载一个库，需要先使用 deferred as 来导入，如下所示。

```
import 'package:deferred/hello.dart' deferred as hello;
```

当需要使用库的时候，调用 loadLibrary 函数来加载库，如下所示。

```
greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

12.12　异步支持

Dart 库中包含许多返回 Future 或 Stream 对象的函数。这些函数是异步的，它们在基本操作后会返回，而不等待该操作完成，例如读取一个文件，在打开文件后就返回了。

虽然看起来有点像同步代码，但是 async 和 await 的代码的确是异步的，如下所示。

```
await readFile()
```

要使用 await，其方法必须带有 async 关键字。

```
FileOperate() async {
var file= await readFile()
//其他处理
}
```

12.13 使类可调用

如果 Dart 中的类实现了 call 函数，那么这个类可以当作方法来调用，如下所示。

```
class JointFunction {
  call(String a, String b, String c, String d) => '$a $b $c $d';
}

main() {
  var jf = new JointFunction();
  var out = jf("放","手","去","做");//1
  print('$out');
}
```

在上述代码中，JointFunction 类定义了一个 call 函数，它接收三个字符串并将它们进行拼接。这样在注释 1 处就可以调用 JointFunction 类了。

12.14 创建实例

在 Java 中创建实例可以用 new，在 Dart 中可以选择用 new，也可以选择不用，如下所示。

```
Element element = Element();
```

对 Android 开发来说，用 new 可能更习惯一些，其可读性也比较好，不用 new 显得更简洁，至于是否用 new 就看团队的要求和个人的习惯。

12.15 本章小结

Dart 的知识点有很多，这里只介绍了一部分需要重点掌握的内容，如果读者想了解更多内容，则可以查看官方文档，关于 Dart 的学习可以结合 Flutter 边写边学，不要只局限于 Dart 的细节，本章也为下一章做了铺垫。

第 13 章

Flutter 基础

关联章节：

第 11 章 跨平台技术演进

第 12 章 Dart 基础

Flutter 是 Google 开发的一套全新的跨平台开源 UI 框架，支持 iOS 和 Android 开发，它也是构建未来的 Fuchsia 系统应用的主要方式。

Flutter 本身并不属于 Android 技术范畴，但随着技术发展和迭代，Flutter 已经是 Android 开发需要掌握的技术之一，关于 Flutter 完全可以写一本书，本章内容带领读者了解 Flutter。

13.1 Flutter 开发环境搭建和 Hello World

学习 Flutter 之前，要先学习环境的搭建，可以在 Windows、macOS、Linux 上进行 Flutter 开发，由于本章内容是面向 Android 开发来编写的，因此选择大部分人都使用的 Windows 平台。

13.1.1 Flutter 概述

Flutter 是 Google 的移动 UI 框架，可以快速在 Android 和 iOS 上构建高质量的原生用户界面，它的前身是 Google 试验项目 Sky。Flutter 主要有以下几个特点。

（1）符合各个平台的用户体验。

Flutter 内置了 Material Design 和 Cupertino 风格（iOS 风格）的 Widgets，使得我们可以开发出符合 Android 和 iOS 风格的应用，另外 Flutter 提供了丰富的 motion API、平滑而自然的滑动效果和平台感知，为用户带来全新体验。

（2）响应式框架。

使用 Flutter 的响应式框架和一系列基础 Widget，可以轻松构建用户界面。使用功能强大且灵活的 API（针对 2D、动画、手势、效果等）可以帮助开发者解决艰难的 UI 挑战。

（3）跨平台自绘引擎。

Flutter 与 Hybrid App、React Native 这些跨平台技术不同，Flutter 既没有使用 WebView，又没有使用各个平台的原生控件，而是本身实现一个统一接口的渲染引擎来绘制 UI，这样可以保证不同平台 UI 的一致性。另外采用自绘引擎后，Flutter 在布局过程中不需要像 RN 那样，要在 JavaScript 和 Native 之间通信，可以节省性能开销。Flutter 使用 Skia 作为其 2D 渲染引擎，Skia 是 Google 的一个 2D 图形处理函数库，包含字形、坐标转换，以及点阵图都有高效且简洁的表现，Skia 是跨平台的，并提供了非常友好的 API。

（4）采用 Dart 开发。

Flutter App 采用 Dart 来开发。与 JavaScript 相比，Dart 在 JIT（即时编译）模式下，其速度与 JavaScript 基本持平。但是当 Dart 以 AOT 模式运行时，Dart 的性能要高于 JavaScript。

（5）访问本地功能和 SDK 并支持插件。

Flutter 可以复用 Java、Swift 或 ObjC 代码，访问 Android 和 iOS 上的原生系统功能。另外通过 Flutter 插件也可以访问原生系统功能，比如蓝牙、相机、Wi-Fi 等。

13.1.2 Flutter 开发环境搭建

在 Windows 平台开发，官方的环境要求是 Windows 7 SP1 或更高版本（64 位），至少 400 MB 的硬盘空间，如果在 macOS 和 Linux 上搭建开发环境，那么请查看官方文档。

13.1.2.1　安装

如果此前安装过 Git for Windows 和 PowerShell 5.0，按 win + R 快捷键输入 cmd，在 cmd 中输入 git 命令来验证下，比如 git -- version。

PowerShell 是微软推出的代替 cmd 的更方便的命令行工具，Windows 7、Windows 8、Windows 10 系统都自带这个命令行工具。右键单击 Windows 10 的开始按钮搜索 PowerShell。

13.1.2.2　下载 Flutter SDK

下载 Flutter SDK 有以下多种方法。

（1）Flutter 官方网站下载最新 Beta 版本进行安装。

（2）在 Flutter GitHub 项目中下载。

（3）通过 git 命令来下载 Flutter 的 Beta 版本。

```
git clone -b beta https://github……
```

这里下载最新的 Beta 版本即可。

将下载下来的 zip 安装包解压到想要安装 Flutter SDK 的路径，比如 F:\Flutter。需要注意的是，不要将 Flutter 安装到需要一些高权限的路径，比如 C:\Program Files\。

13.1.2.3　配置环境变量

1．配置 Flutter 环境变量

如果想在命令行中使用 Flutter 命令，则需要将 Flutter 的路径添加到系统变量 Path 中。

我的 Flutter 路径为 F:\Flutter\flutter，那么就将 F:\Flutter\flutter\bin 添加到系统变量 Path 中，如图 13-1 所示。

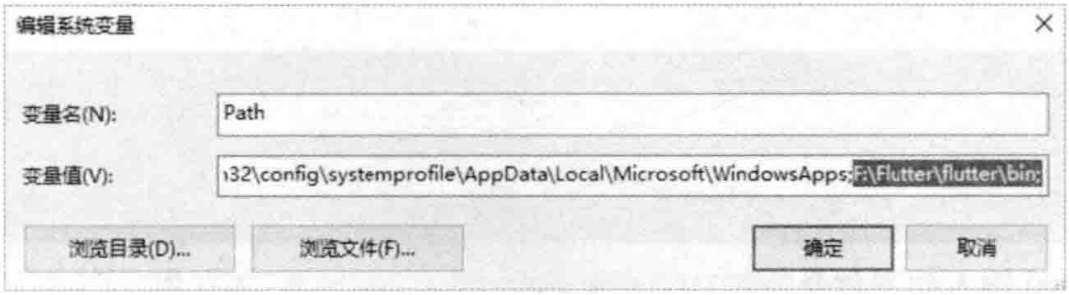


图 13-1　配置系统变量

在 cmd 中输入 flutter --version 来验证是否配置成功。

2. 配置国内镜像

如果在国内安装或使用 Flutter，则使用 Flutter 的官方镜像或者其他镜像能够免于限制并且加快速度，Flutter 的官方镜像如下。

```
PUB_HOSTED_URL=https://pub.flutter……
FLUTTER_STORAGE_BASE_URL=https://storage.flutter……
```

这两个镜像是会变的，具体内容可以查看官方说明。

除此之外，还有上海交通大学 Linux 用户组，如下所示。

```
PUB_HOSTED_URL: https://dart-pub.mirrors.sjtug……
FLUTTER_STORAGE_BASE_URL: https://mirrors.sjtug.sjtu……
```

这里使用 Flutter 的官方镜像，将这两个 url 配置到用户环境变量中，如图 13-2 和图 13-3 所示。

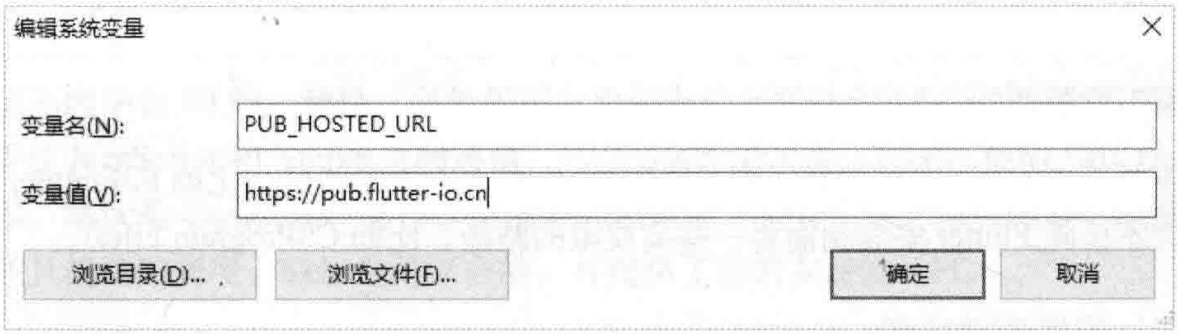


图 13-2　配置到用户环境变量 1

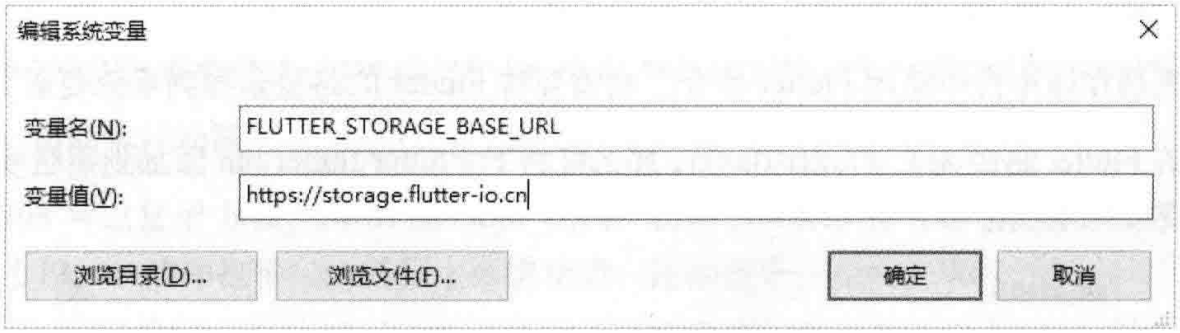


图 13-3　配置到用户环境变量 2

13.1.2.4　安装和检查各种依赖项

打开 cmd 输入如下命令。

```
flutter doctor
```

这时会打印一些提示信息，如图 13-4 所示。

```
C:\Users\52501>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel beta, v1.4.9-hotfix.1, on Microsoft Windows [Version 10.0.17134.706], locale zh-CN)

[!] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
    ! Some Android licenses not accepted.  To resolve this, run: flutter doctor --android-licenses
[!] Android Studio (version 3.3)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[!] IntelliJ IDEA Ultimate Edition (version 2018.3)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[!] Connected device
    ! No devices available

! Doctor found issues in 4 categories.
```

图 13-4　打印的提示信息

由图 13-4 可以看到有五个类别的依赖项，如果安装的 IDE 不同，则类别内容也会不同。其中，“√”是没有问题的类别，“!”是有问题的类别。图 13-4 的类别含义如下所示。

（1）Flutter 的版本和运行环境。

（2）Android 的工具链的情况。

（3）Android Studio 的版本，以及 Flutter 插件安装情况。

（4）IntelliJ IDEA Community 的版本，以及 Flutter 插件安装情况。

（5）设备的连接情况。

安装缺少的依赖项后，可以再次运行 flutter doctor 命令来验证是否已正确设置所有内容。比如第二项有问题，图 13-4 提示可以运行 flutter doctor --android-licenses 来解决，然后，再次运行 flutter doctor 命令，打印的提示信息会发生变化，如图 13-5 所示。

```
C:\Users\52501>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel beta, v1.4.9-hotfix.1, on Microsoft Windows [Version 10.0.17134.706], locale zh-CN)
[√] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
[!] Android Studio (version 3.3)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[!] IntelliJ IDEA Ultimate Edition (version 2018.3)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[!] Connected device
    ! No devices available

! Doctor found issues in 3 categories.
```

图 13-5　再次运行 flutter doctor 命令后打印的提示信息

13.1.2.5 设置编辑器

（1）安装 Android Studio 或者 IntelliJ IDEA。

安装 Android Studio 需要安装 3.0 或者更高的版本。

除了 Android Studio 还可以使用 IntelliJ IDEA：

IntelliJ IDEA Community，2017.1 版本或更高版本。

IntelliJ IDEA Ultimate，2017.1 版本或更高版本。

（2）安装 Flutter 和 Dart 插件。

依次选择 Android Studio 的 File→settings→Plugins→Browse repositories，在搜索框输入 Flutter 和 Dart 来安装它们，如图 13-6 所示。

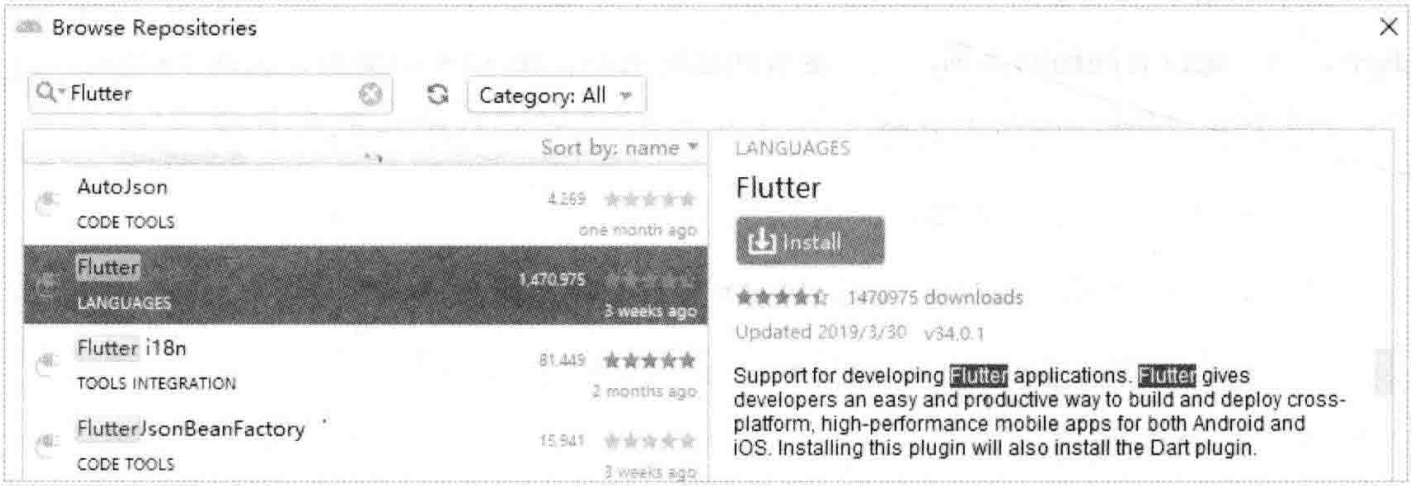


图 13-6 搜索 Flutter

安装成功后重启 Android Studio。这时在 cmd 中输入 flutter doctor 命令，就会看到提示信息的 Android Studio 的版本依赖项为“√”。

13.1.2.6 设置 Android 设备和模拟器

设置 Android 设备和开发 Android 一样，需要注意的是，这个设备的版本应该为 Android 4.1（API 级别 16）或更高版本。选择 Android Studio 和 Genymotion 创建的模拟器都可以，推荐使用 x86 或者 x86_64 镜像。关于如何连接设备和创建模拟器就不进行讲解了。

13.1.3　运行一个 Flutter Demo

在 Android Studio 菜单栏选择 File→New Flutter Project，选择 Flutter Application，如图 13-7 所示。

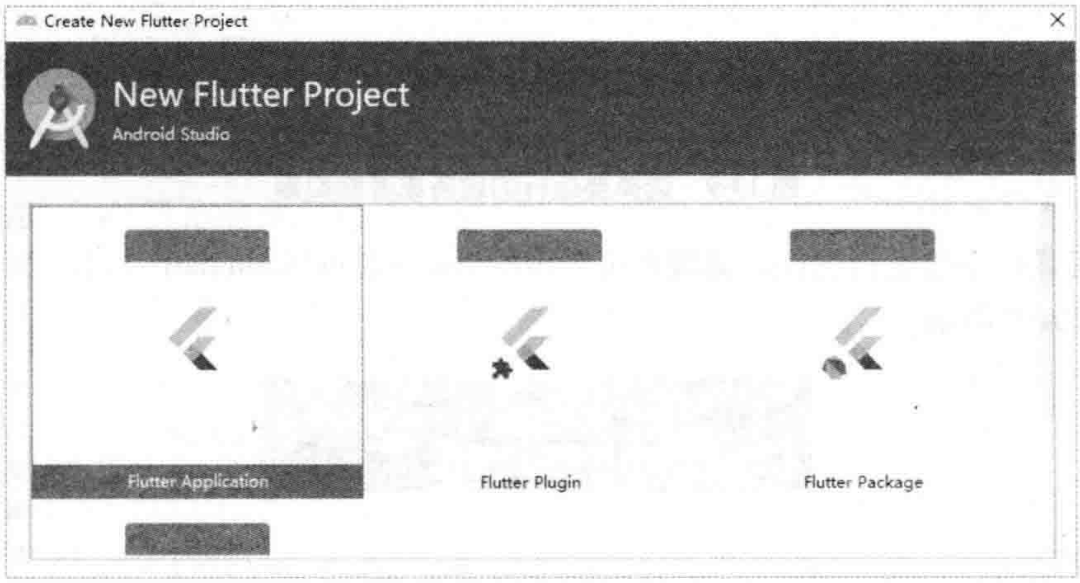


图 13-7　选择 Flutter Application

Project name 需要小写，接着设置 Flutter SDK 的位置和项目的位置，如果没有下载 Flutter SDK，则可以单击 install SDK 下载，如图 13-8 所示。

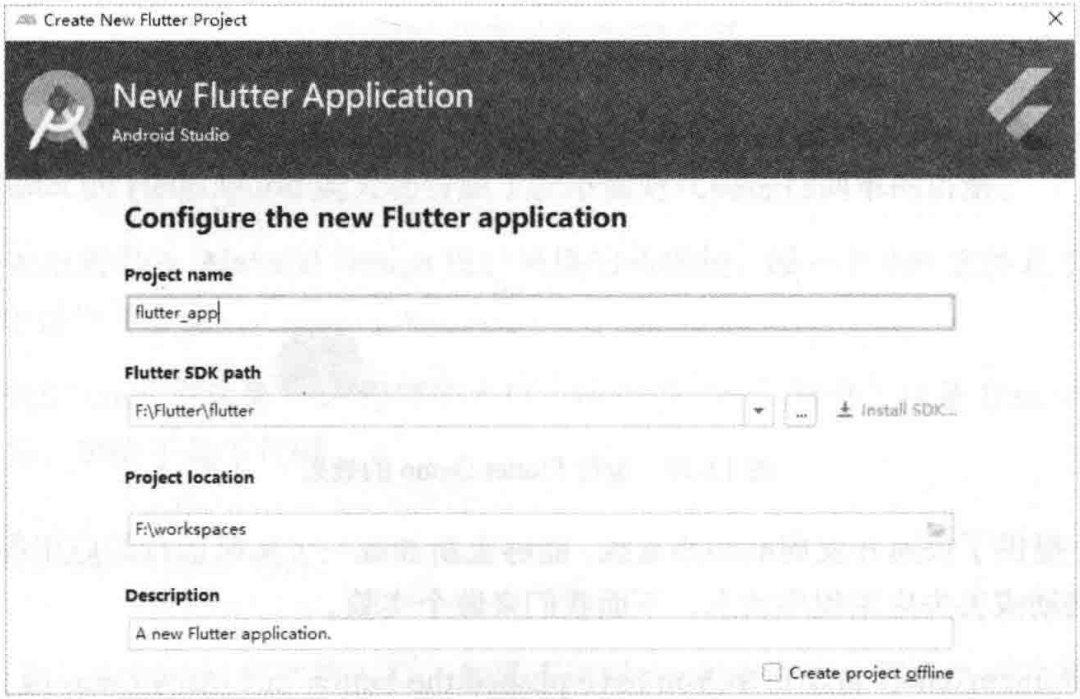


图 13-8　设置 Flutter 工程

项目创建完成后，在 Android Studio 顶部的工具栏中的目标选择器里选择要运行的设备或者模拟器，我在这里选择的是 Genymotion 创建的 Custom Phone(Android 9.0)，如图 13-9 所示。

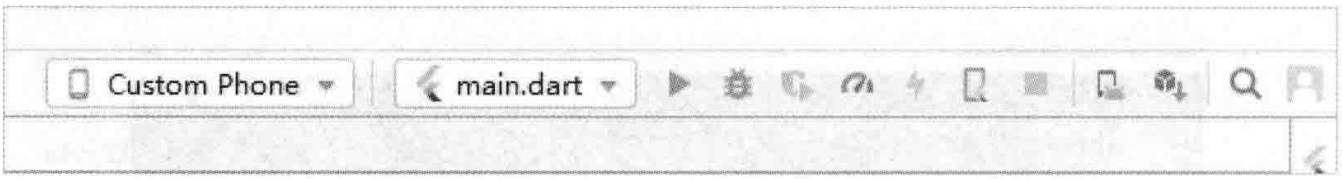


图 13-9　选择要运行的设备或者模拟器

单击工具栏中的运行图标，或调用菜单项的 Run→Run ‘main.dart’ 运行 Flutter Demo，效果如图 13-10 所示。

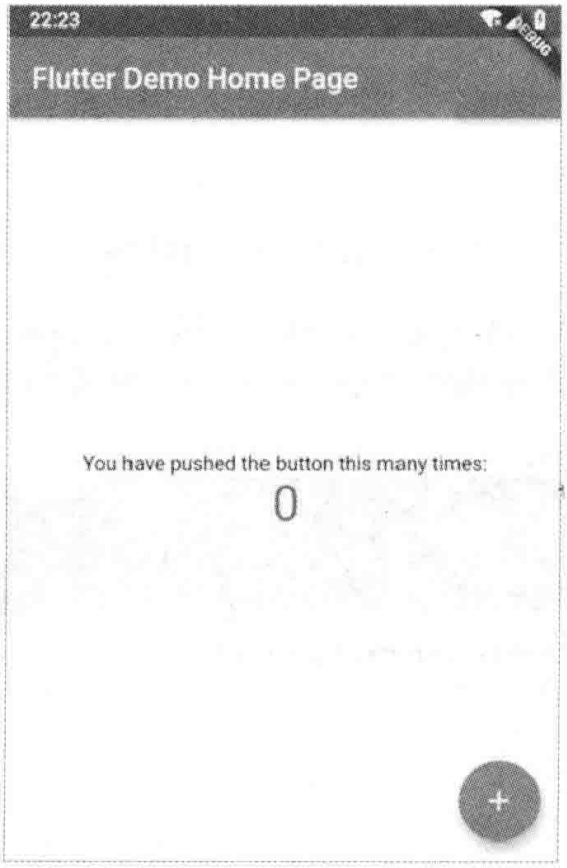


图 13-10　运行 Flutter Demo 的效果

Flutter 提供了快速开发周期和热重载，能够重新加载一个实时运行的应用程序的代码，无须重新启动或丢失应用程序状态，下面我们来做个实验。

打开 lib/main.dart，将字符串 You have pushed the button this many times 改为 You have clicked the button this many times。

然后单击 Save All 或者单击工具栏中的黄色闪电图标，就可以发现字符串更新了修改的值。

13.1.4　Hello World

将 lib/main.dart 中的所有代码替换为如下内容。

```
import 'package:flutter/material.dart';//1

void main() => runApp(MyApp());//2

class MyApp extends StatelessWidget {//3
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
        ),
        body: Center(//4
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

这时 Flutter 的 Hello World 就大功告成了。下面对代码进行简单的讲解。

注释 1 处用来引入 Material Design 设计风格的基础包，每一个 dart 文件几乎都会引入它，Flutter 中提供了丰富的 Material widgets。

注释 2 处的 main 方法是 Dart 程序的入口，使用了“=>”符号，这是 Dart 中单行函数或方法的简写，等价于如下代码。

```
void main(){
return runApp(Widget app);
}
```

runApp 方法是 Flutter 框架的入口，如果不返回 runApp 方法，那么执行的是一个 Dart 控制台应用。

注释 3 处说明 MyApp 继承了 StatelessWidget，这会使应用本身也成为一个 Widget。

Flutter 中有一切皆 Widget 的概念，Widget 分为 StatelessWidget（无状态 Widget）和 StatefulWidget（有状态 Widget）。

build 方法用来描述如何用其他较低级别的 Widget 来显示自身。

MaterialApp 包含了 App 实现 Material Design 所需要的一些 Widget，其中 title 为在任务管理窗口中所显示的应用名字，home 为应用默认显示的界面 Widget。

Scaffold 是 Material Design 布局结构的基本实现，Scaffold 内部包含了 AppBar 和 body，AppBar 为应用程序栏，它在 Scaffold 的顶部；body 为 Scaffold 的主要内容，通俗来讲就是屏幕的大部分。在注释 4 处，body 包含了一个 Center Widget，Center Widget 中有一个子 Widget Text，这样字符 Hello World 就会显示在 body 的中心，运行效果如图 13-11 所示。

图 13-11　运行效果

13.2　Widget 基础

Flutter 的 UI 界面就是由 Widget 组成的，Widget 的数量很多，因此我会用几节的内容来专门介绍它，本节就来介绍 Widget 基础。

13.2.1　Widget 概述

Flutter 的 Widget 的设计灵感来自 React，其主要目的就是使用 Widget 构建 UI。Widget 根据其当前配置和状态来描述视图，当 Widget 的状态发生更改时，Widget 会重建其描述。framework 将根据前面的描述进行对比，以确定底层渲染树从一个状态转换到下一个状态所需的最小更改。

在 Flutter 中，除了 Basics 的文本、图片、卡片、输入框这些基础控件，布局方式和动画等也都是由 Widget 组成的。使用不同类型的 Widget，就可以实现复杂的界面。

Widget 可以翻译为部件，相当于 Android 中的 View。Widget 和 View 不同的是：Widget 具有不同的生命周期，它是不可变的，每当 Widget 或者其状态发生变化时，Flutter 的框架都会创建一个新的 Widget 实例树。相比之下，Android 中的 View 会被绘制一次，并且在 invalidate 调用之前不会重绘。

13.2.2　Widget 的分类

Widget 的分类有很多，每个类别又包含很多 Widget，其主要类别如下。

- Basics：在构建第一个 Flutter 应用程序之前，需要知道 Basic Widget。
- Material Components：Material Design 风格的 Widget。
- Cupertino：iOS 风格的 Widget。
- Accessibility：辅助功能 Widget。
- Animation and Motion：动画和动作 Widget。
- Async：Flutter 应用程序的异步 Widget。
- Input：除了在 Material Components 和 Cupertino 中输入的 Widget，还可以接受用户输入的 Widget。
- Interaction Models：响应触摸事件并将用户路由到不同的视图中。
- Layout：用于布局的 Widget。
- Painting and effects：在不改变布局、大小、位置的情况下，为子 Widget 应用视觉效果。
- Scrolling：滚动相关的 Widget。
- Styling：主题填充相关的 Widget。
- Text：显示文本和文本样式。

Basics 有些特殊，它是由 Flutter 官方从其他 Widget 分类中选取的一些 Widget 组成的，

这些 Widget 是官方建议开发者构建第一个 Flutter 应用程序之前需要知道的，其目的是让开发者更快入门。比如 Row 属于 Layout 分类，它就被选入 Basics 中。

Widget 更多的是以组合的形式存在的，比如 Container 属于 Layout 中的一个 Widget，而 Container 又由 LimitedBox、 ConstrainedBox、Align、Padding、DecoratedBox、Transform 部件组成。如果要实现 Container 的自定义效果，可以组合上述这些 Widget 及其他简单的 Widget，而不是将 Container 进行子类化实现。

13.2.3 Widget 的状态分类

在 Android 中，我们可以通过直接更改 View 来更新视图。但是在 Flutter 中，Widget 是不可变的并且不会直接更新，而必须使用 Widget 的状态。

Widget 有两种状态分类，分别是无状态的 StatelessWidget 和有状态的 StatefulWidget，StatelessWidget 是不可变的，设置以后就不可再变化，所有值都是最终的设置。StatefulWidget 可以保存自己的状态，但是 Widget 是不可变的，因此需要配合 State 来保存状态。

State 拥有自己的生命周期，如表 13-1 所示。

表 13-1 State 的生命周期

名　称	状　态
initState	create 之后被 insert 到渲染树时调用的，只会调用一次
didChangeDependencies	state 依赖的对象发生变化时调用
didUpdateWidget	Widget 状态发生改变时调用，可能会调用多次
build	构建 Widget 时调用
deactivate	移除渲染树时调用
dispose	即将销毁 Widget 时调用

13.2.4 根 Widget 的种类

```
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
     ...
       ),
     ),
```

```
    );
  }
}
```

上述代码中的 MaterialApp 就是一个根 Widget，也就是 Flutter 应用程序的第一个 Widget，根 Widget 的种类如下。

- WidgetsApp：如果需要自定义风格，那么可以使用 WidgetsApp。
- MaterialApp：Material Design 风格的 Widget。
- CupertinoApp：iOS 风格的根 Widget。

如果公司没有特殊要求，那么这里建议使用 MaterialApp 作为根 Widget 即可。

13.3　Basic Widget

Basic Widget 也就是 13.2 节提到的 Basics，主要有以下几种。

- Container：一个便利的容器 Widget，可以设置 Widget 的背景、尺寸、定位。
- Row：在水平方向上布置子窗口 Widget 列表。
- Column：在垂直方向上布置子窗口 Widget 列表。
- Image：显示图像的 Widget。
- Text：单一样式的文本。
- Icon：符合 Material Design 设计规范的图标。
- RaisedButton：符合 Material Design 设计规范的凸起按钮。
- Scaffold：实现 Basics 的 Material Design 布局结构。
- Appbar：Material Design 的应用栏。
- FlutterLogo：以 Widget 形式展示一个 Flutter 图标，可以调整样式。
- Placeholder：绘制一个框，为将来添加的 Widget 占位。

这里选择一些我们必须要掌握的 Basics Widget 来进行讲解。

13.3.1　代码模板和主题

为了理解这些 Basic Widget，需要写一些示例，这些示例需要一个代码模板，以方便测试和学习，如下所示。

```
import 'package:flutter/material.dart';
```

```
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(//1
      title: 'Welcome to Flutter',
      home: Scaffold(//2
        appBar: AppBar(//3
          title: Text('Basics Widget'),
        ),
        body:
          Padding(
            padding: EdgeInsets.all(40.0),
            child:  Text('在这里编写和测试其他基本 Widget'),
          ),
      ),
    );
  }
}
```

上述代码稍微改动了官方的 Hello World 代码，以便于测试，具体的代码含义已经在 13.2.4 节中讲过了，这里结合本节要讲的内容再讲一些细节。注释 1 处的 MaterialApp 属于 Material Components 类别中的 Widget，MaterialApp 中包含了实现 Material Design 的应用程序所需要的 Widget。

注释 2 处和注释 3 处的 Scaffold 和 AppBar 同样也是 Material Components 类别中的 Widget，Scaffold 实现了 Material Design 布局结构，AppBar 是 Material Design 的应用栏，运行效果如图 13-12 所示。

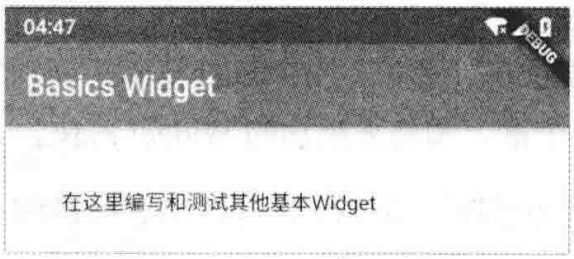


图 13-12　运行效果

13.3.2　文本

Text 还可以定义样式，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Basics Widget'),
        ),
        body: Padding(
          padding: EdgeInsets.all(60.0),
          child: Text(
            '文本样式',
            style: TextStyle(
              fontSize: 16.0,
              color: Colors.indigo,
              fontStyle: FontStyle.normal,
              fontWeight: FontWeight.bold,
            ),
          ),
        ),
      ),
    );
  }
}
```

这里为了便于理解列出了全部代码，此后的举例只列出改变的部分。

通过 TextStyle 来定义文本的样式，效果如图 13-13 所示。

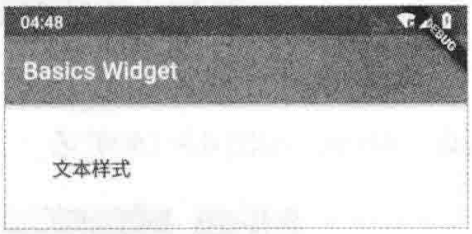


图 13-13　运行 Flutter Demo 效果

13.3.3　图片

Image 的构造函数如下。

- new Image：从 ImageProvider 中获取图片。
- new Image.asset：使用 key 从 AssetBundle 中获取图片。
- new Image.network：加载网络图片。
- new Image.file：从文件中获取图片。
- new Image.memory：用于从 Uint8List 中获取图片。

Image 的属性有很多种，其主要属性为 fit，用于表示图片的填充模式，参数类型为 BoxFit，BoxFit 的取值主要有以下几种，示例图片均来自 Flutter 官方。

contain

全图显示，保持原比例，如图 13-14 所示。

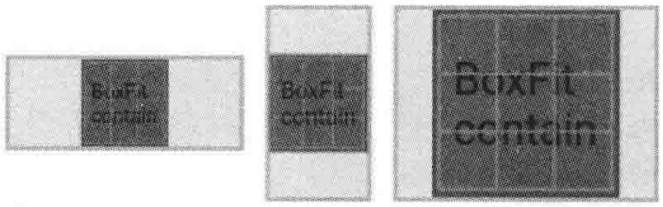


图 13-14　contain 属性

cover

全图充满，可能拉伸也可能被裁剪，如图 13-15 所示。

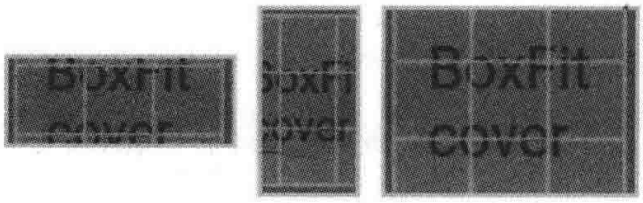


图 13-15　cover 属性

fill

全图显示，通过拉伸来充满目标框，如图 13-16 所示。

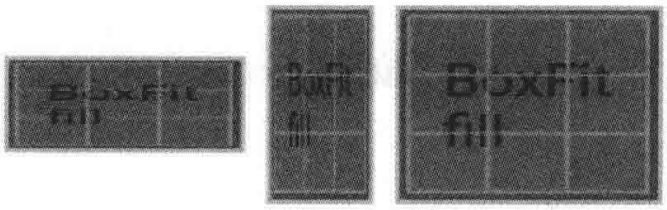


图 13-16　fill 属性

fitHeight

图片高度充满目标框，可能拉伸也可能被裁剪，如图 13-17 所示。

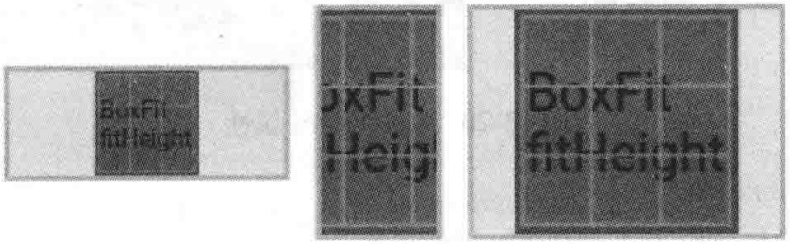


图 13-17　fitHeight 属性

fitWidth

图片宽度充满目标框，可能拉伸也可能被裁剪，如图 13-18 所示。

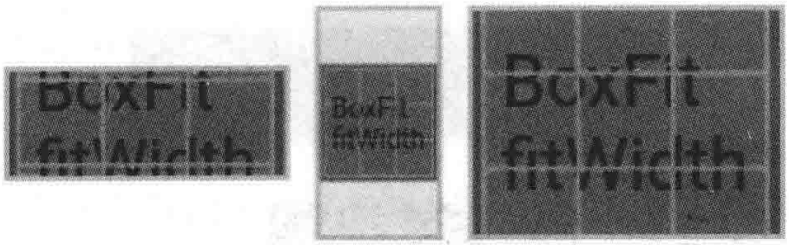


图 13-18　fitWidth 属性

none

保持图片的原始大小，剪裁掉位于目标框外的图片部分，如图 13-19 所示。

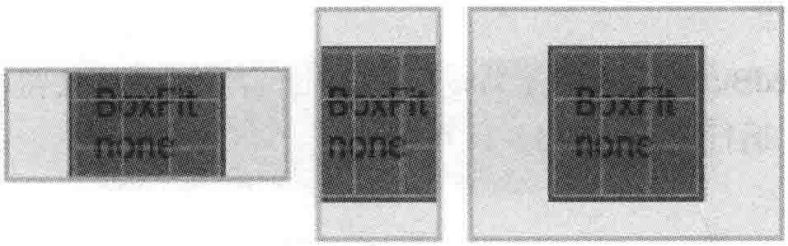


图 13-19　none 属性

scaleDown

与 contain 缩小图像的方式相同，只不过会在必要时缩小以确保图片完全在目标框内，如果不缩小则等同于 none，如图 13-20 所示。

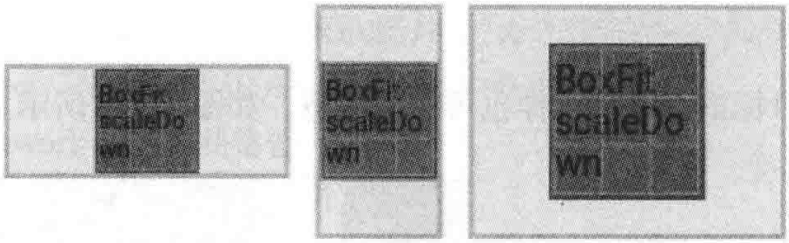


图 13-20　scaleDown 属性

```
child: Image.network(
          "https://upload-images.jianshu……",
          width: 260,
          height: 100,
          fit: BoxFit.fill,
        ),
```

fill 属性的效果如图 13-21 所示。

图 13-21　fill 属性的效果

13.3.4　凸起按钮

凸起按钮（RaisedButton）是符合 Material Design 设计规范的按钮，可以通过 onPressed 来回调按钮的单击，运行效果如图 13-22 所示。

```
RaisedButton(
          onPressed: () => print("onPressed"),
          color: Colors.lightBlueAccent,
          child: Text('RaisedButton', style: TextStyle(fontSize: 10)),
        ),
```

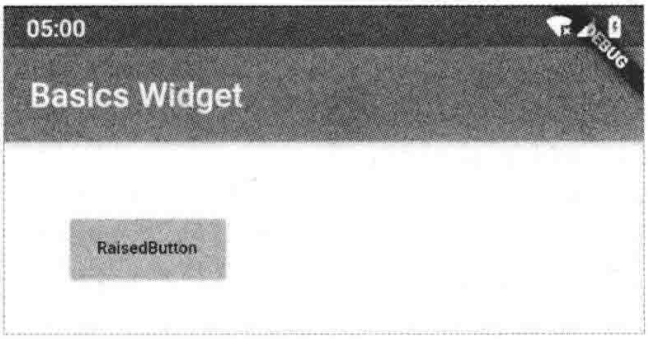


图 13-22　运行 RaisedButton 效果

除了使用 RaisedButton，Flutter 还提供了其他按钮，比如 FlatButton、IconButton、FloatingActionButton 等，它们的使用方法和 RaisedButton 大同小异，这里就不再赘述了。

13.3.5　其他 Widget

Basic Widget 中还有 Row、Column、Container 等 Widget，这里分别对其进行简单介绍。

1．Row

Row 用于在水平方向显示数组中的子元素 Widget，效果如图 13-23 所示。

```
child: Row(
      children: <Widget>[
        Icon(Icons.access_alarm),
        Icon(Icons.add_a_photo),
        Icon(Icons.add_call),
      ],
    ),
```

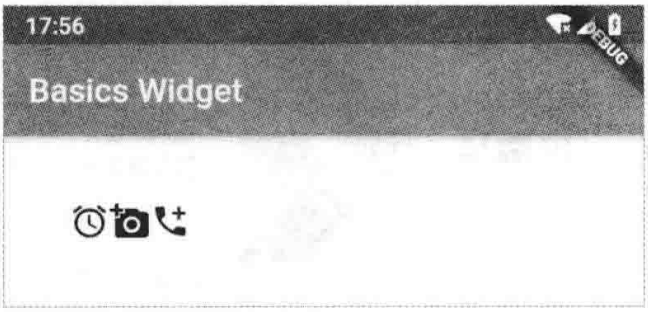


图 13-23　Row 示例效果

Column 用于在垂直方向显示数组中的子元素 Widget，其使用方法和 Row 一样。这里需要提到的是 Expanded，Expanded 可以用来配合 Row 和 Column 使用，用于填充剩余的空间，如下所示。

```
Row(
  children: <Widget>[
    Icon(Icons.access_alarm),
    Icon(Icons.add_a_photo),
    Icon(Icons.add_call),
    Expanded(
      child: FittedBox(
        fit: BoxFit.contain,
        child: const FlutterLogo(),
      ),
      flex: 2,
    ),
    Expanded(
      child: Text(
        "占剩余部分的三分之一",
      ),
      flex: 1,
    ),
  ],
),
```

其中，Expanded 的作用是在自己的尺寸范围内缩放并且调整 child 位置，使 child 适合其尺寸。FlutterLogo 是 Basics Widget 中的一种，用于展示 Flutter 图标。使用 flex 可以调整两个 Expanded 的占比。Expanded 示例效果如图 13-24 所示。

图 13-24　Expanded 示例效果

2. Container

一个便利的容器 Widget，可以设置 Widget 的背景、尺寸、定位。Container 描述起来有些抽象，它和 Android 中的 ViewGroup 类似，如下所示。

```
Container(
      decoration:BoxDecoration(
        color: Colors.lightGreen
      ),
    child: Text('Container'),
    padding: EdgeInsets.all(36.0),
    margin: EdgeInsets.all(10.0),
  ),
```

Container 的 padding 和 margin 属性与 Android 中的作用是类似的，效果如图 13-25 所示。

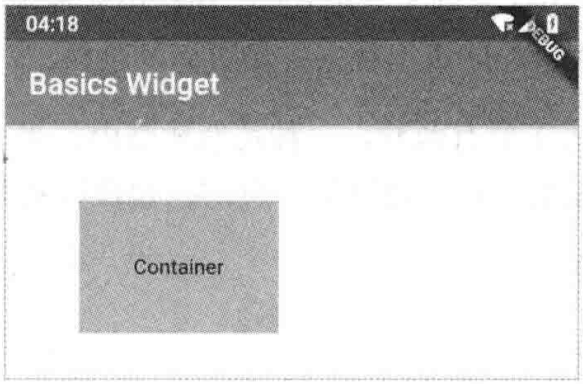


图 13-25　Container 示例效果

13.4　Material 组件

除了 Basic Widget，我们还需要了解 Material Components，也就是 Material 组件。它提供了实现 Material Design 准则的视觉、行为和动作的 Widget。

官方将 Material 组件分为以下几个类型。

- 应用程序结构和导航。
- Button。
- 输入和选择。
- 对话框，警告弹框和面板。
- 信息显示。
- 布局。

由于篇幅的原因，这里只介绍应用程序结构和导航分类中的 MaterialApp、Scaffold、AppBar、BottomNavigationBar、TabBar、Drawer，其余的 Widget 可以查看官方文档。

13.4.1 MaterialApp

讲到 Material 组件，就要提到 MaterialApp，它包含了很多 Widget，这些 Widget 通常是实现 Material Design 的应用程序所需的。MaterialApp 在此前的章节中都用过，这里就不介绍了，下面简单介绍下路由。

在 Android 开发中，我们使用 Intent 来进行界面跳转，也称之为原生路由，后来出现了一些路由框架，比如 ARouter。

在 Flutter 中进行界面跳转的就是路由，路由用 Route 类表示，Navigator 是对 Route 进行管理的 Widget。Navigator 不仅管理了一堆 route，还提供管理堆栈的方法，即 Navigator.push 和 Navigator.pop，通过路由对象的进出栈来控制页面的跳转。

Flutter 路由的使用方式主要有两种，一种是新建路由，另一种是注册路由。我们分别用这两种方式举例：首屏是第一页，通过第一页的按钮跳转到第二页，单击第二页的按钮回到第一页。

1. 新建路由

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Material Components',
      home: FirstPage(),
    );
  }
}

class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('第一页'),
      ),
      body: Padding(
        padding: EdgeInsets.all(30.0),
        child: RaisedButton(
```

```
        child: Text('跳转到第二页'),
        onPressed: () {
          Navigator.push(//1
            context,
            MaterialPageRoute(builder: (context) => SecondPage()),
          );
        },
      ),
    ),
  );
 }
}

class SecondPage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text('第二页'),
   ),
   body: Padding(
    padding: EdgeInsets.all(30.0),
    child: RaisedButton(
      child: Text('回到上一页'),
      onPressed: () {
       Navigator.pop(context);//2
      }),
   ),
  );
 }
}
```

注释 1 处调用了 Navigator.push，它将新建的路由添加到 Navigator 管理的 route 堆栈的栈顶，这个路由我们可以自定义，但是建议使用 MaterialPageRoute，它是一个模态路由，可以适应各个平台进行页面替换，并提供了相应的页面切换动画。在 Android 平台时，页面进入动画是向上滑动并淡出的，退出是相反的动画，如果在 iOS 平台，那么页面进入动画是从右侧滑入的，退出是相反的动画。

单击“跳转到第二页”按钮时会跳转到 SecondPage。注释 2 处的 Navigator.pop 用于弹出 route 堆栈最顶层的 Route，效果如图 13-26 和图 13-27 所示。

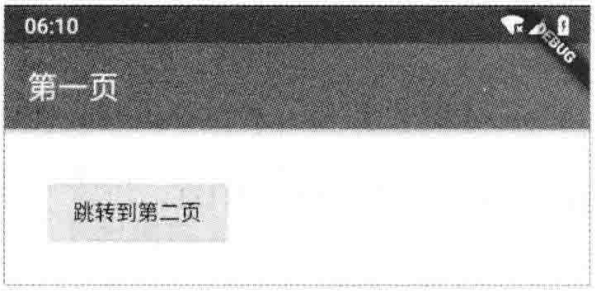


图 13-26　第一页效果

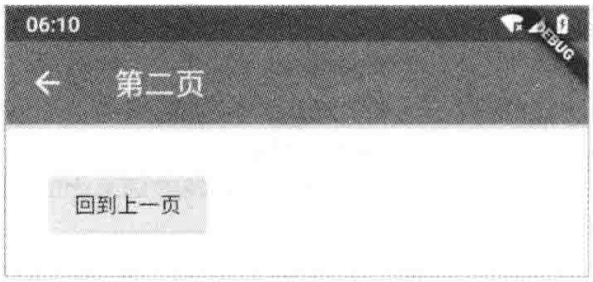


图 13-27　第二页效果

2. 注册路由

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Material Components',
      home: FirstPage(),
      routes: <String, WidgetBuilder>{//1
        '/first': (BuildContext context) => FirstPage(),
        '/second': (BuildContext context) => SecondPage(),
      },
      initialRoute: '/first' ,
    );
  }
}

class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('第一页'),
      ),
      body: Padding(
        padding: EdgeInsets.all(30.0),
        child: RaisedButton(
          child: Text('跳转到第二页'),
          onPressed: () {
            Navigator.pushNamed(context, '/second');//2
          },
        ),
      ),
```

```
    );
  }
}

class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('第二页'),
      ),
      body: Padding(
        padding: EdgeInsets.all(30.0),
        child: RaisedButton(
            child: Text('回到上一页'),
            onPressed: () {
              Navigator.of(context).pop();
            }),
      ),
    );
  }
}
```

注释 1 处的 routes 用于初始化一个路由列表，当推送路由时，将在 routes 中查找路径名称，如果该名称存在，则关联的 WidgetBuilder 用于构造 MaterialPageRoute。注释 2 处的 Navigator.pushNamed 和 Navigator.push 作用类似，只不过 pushNamed 的参数为路由的名称。

13.4.2　Scaffold

Scaffold 同样属于 Material 组件，它实现了 Material Design 的基本布局结构，因此它经常会作为 MaterialApp 的子 Widget。Scaffold 会自动填充可用的空间，这意味着它将占据整个窗口或屏幕，并且 Scaffold 会自动适配屏幕。我们的布局就是在 Scaffold 中进行编写的，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Scaffold 示例'),
        ),
        body: Padding(
          padding: EdgeInsets.all(30.0),
          child: Text('Scaffold'),
        ),
        bottomNavigationBar: BottomAppBar(
          child: Container(height: 50),
        ),
        drawer: Drawer(
          child: Center(
            child: Text('抽屉'),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.add),
        ),
      ),
    );
  }
}
```

Scaffold 有很多属性，上述代码中用了以下几个属性。

- appBar：用于设置顶部的标题栏。
- body：用于显示 Scaffold 的主要内容。
- bottomNavigationBar：用于设置 Scaffold 的底部导航栏。
- drawer：用于设置抽屉效果。
- floatingActionButton：用于设置位于右下角的按钮。

Scaffold 示例效果如图 13-28 所示。

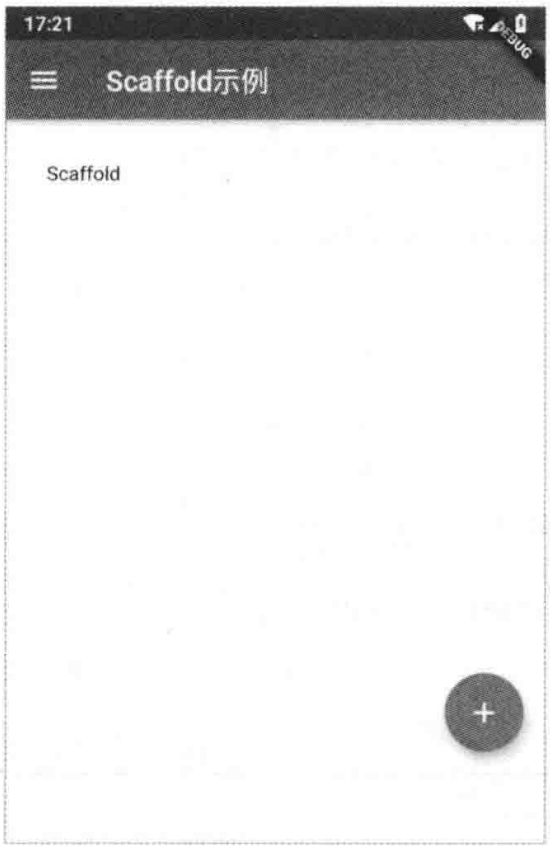


图 13-28　Scaffold 示例效果

可以看到在 AppBar 上有个抽屉的按钮，单击该按钮就会滑出抽屉。

13.4.3　AppBar

AppBar 由 toolbar 和其他可选的 Widget 组成，比如 TabBar 和 FlexibleSpaceBar。

AppBar 会在顶部显示 leading、title、actions 等内容，底部 bottom 通常显示 TabBar，图 13-29 展示了这些内容的位置分布。

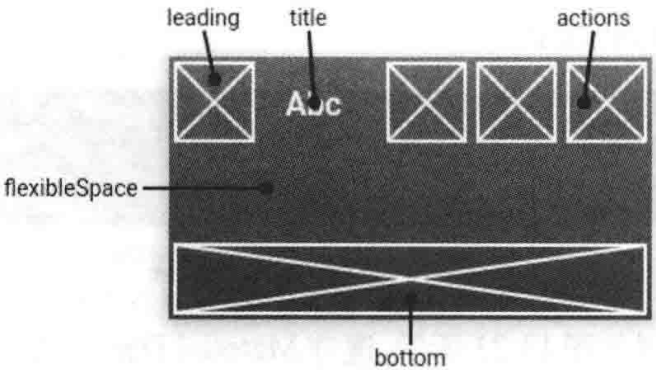


图 13-29　AppBar 中的内容位置分布

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyScaffld(),
    );
  }
}

class MyScaffld extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('AppBar 示例'),
        leading: FlutterLogo(colors: Colors.lightGreen),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.share),
            onPressed: () {
              print('添加按钮');
            },
          ),
        ],
      ),
    );
  }
}
```

这次没有把所有代码写在 MyApp 类中，而是将 Scaffld 的定义放在了 MyScaffld 类中。AppBar 示例效果如图 13-30 所示。

图 13-30　AppBar 示例效果

上述代码的 Widget 树如图 13-31 所示，遵守 Material Design 准则的 Flutter 应用的 Widget 树大致也是如此。

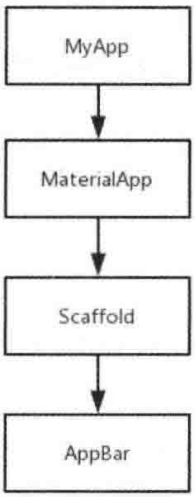


图 13-31　Widget 树

13.4.4　BottomNavigationBar

BottomNavigationBar 是底部的导航栏，用于在 3～5 个少量视图中进行选择。在一般情况下，导航栏的选项卡由文本标签、图标或二者结合的形式组成。

底部导航栏通常与 javaScaffold 结合使用，它会作为 Scaffold.bottomNavigationBar 的参数，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  static const String _title = 'Flutter Code Sample';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyStatefulWidget(),
    );
  }
}

class MyStatefulWidget extends StatefulWidget {
  MyStatefulWidget({Key key}) : super(key: key);

  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();//1
}
```

```
class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  int _selectedIndex = 0;
  static const List<Widget> _widget = <Widget>[
    Text(
      'Index 0:首页',
    ),
    Text(
      'Index 1: 通讯录',
    ),
    Text(
      'Index 2: 设置',
    ),
  ];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('BottomNavigationBar 示例'),
      ),
      body: Center(
        child: _widget.elementAt(_selectedIndex),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            title: Text('首页'),
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.contacts),
            title: Text('通讯录'),
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.build),
            title: Text('设置'),
          ),
        ],
        currentIndex: _selectedIndex,
        selectedItemColor: Colors.amber,
        onTap: _onItemTapped, //2
      ),
    );
```

```
  }
  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
    });
  }
}
```

由于使用的 Widget 需要在 Widget 的生命周期中改变状态，因此 MyStatefulWidget 继承了 StatefulWidget。注释 1 处的 createState 方法会为此 Widget 创建可变状态。注释 2 处的 onTap 属性会在单击其中一个选项卡时调用，它的值由_onItemTapped 方法定义，在这个方法中设置当前的索引并赋值给_selectedIndex，这样通过_selectedIndex 的值就可以切换选项卡了，实现的效果如图 13-32 所示，可以通过单击选项卡来切换界面。

图 13-32　实现的效果

13.4.5　TabBar

TabBar 用于显示水平的选项卡，与 Android 中的 TabLayout 类似。TabBar 通常需要配合 TabBarView 和 TabController 来使用。其中 TabBarView 用于显示与当前所选的选项卡对应的 Widget 视图；TabController 顾名思义就是 TabBarView 和 TabBar 的控制器，是这两个 Widget 的桥梁。实现 TabController 有两种方式，一种是使用 DefaultTabController，另一种

是自定义 TabController。

1. 使用 DefaultTabController

使用 DefaultTabController 方便快捷，直接新建一个 DefaultTabController 即可，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyTabController(),
    );
  }
}
class MyTabController extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 3,
      child: Scaffold(
        appBar: AppBar(
          title: Text('DefaultTabController 示例'),
          bottom: TabBar(
            tabs: <Widget>[
              Tab(
                text: '热点',
              ),
              Tab(
                text: '体育',
              ),
              Tab(
                text: '科技',
              ),
            ],
          ),
        ),
        body: TabBarView(
          children: <Widget>[
            Center(child: Text('热点')),
```

```
          Center(child: Text('体育')),
          Center(child: Text('科技')),
        ],
      ),
    ),
  );
 }
}
```

2. 自定义 TabController

如果想要切换动画或者监听切换的交互，那么可以自定义 TabController，自定义 TabController 需要实现 SingleTickerProviderStateMixin，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: MyStatefulWidget(),
  );
 }
}

class MyStatefulWidget extends StatefulWidget {
 MyStatefulWidget({Key key}) : super(key: key);
 @override
 _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget>
  with SingleTickerProviderStateMixin {
 TabController _tabController;

 void initState() {
  super.initState();
  _tabController = TabController(vsync: this, length: 3);
 }

 @override
 Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(
        title: Text('自定义 TabController'),
        bottom: TabBar(
          tabs: <Widget>[
            Tab(
              text: '热点',
            ),
            Tab(
              text: '体育',
            ),
            Tab(
              text: '科技',
            ),
          ],
          controller: _tabController,//1
        ),
      ),
      body: TabBarView(
        controller: _tabController,
        children: <Widget>[
          Center(child: Text('热点')),
          Center(child: Text('体育')),
          Center(child: Text('科技')),
        ],
      ),
    );
  }
  @override
  void dispose() {
    _tabController.dispose();
    super.dispose();
  }
}
```

自定义 TabController 与使用 DefaultTabController 有两点不同，一点是使用了 StatefulWidget，另一点是在注释 1 处将 TabBar 的 controller 设置为新建的 TabController。自定义 TabController 效果如图 13-33 所示，可以通过滑动界面和单击选项卡来切换界面。

图 13-33　自定义 TabController 效果

13.4.6　Drawer

Drawer 的含义是抽屉，这里表示可以实现拉出推入的效果，与 Android 中的 DrawerLayout 类似。Drawer 通常与 Scaffold.drawer 属性一起使用，抽屉的子项通常是 ListView，其第一个子项是头部，头部主要有以下两个 Widget 可以实现。

- DrawerHeader：展示基本的信息。
- UserAccountsDrawerHeader：展示用户头像、用户名、Email 等信息。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  static const String _title = 'Flutter Code Sample';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyStatefulWidget(),
    );
  }
```

```
}

class MyStatefulWidget extends StatefulWidget {
  MyStatefulWidget({Key key}) : super(key: key);

  @override
  _DrawerState createState() => _DrawerState();
}

class _DrawerState extends State<MyStatefulWidget> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Drawer 示例'),
      ),
      drawer: _drawer,
    );
  }
  get _drawer => Drawer(
       child: ListView(//1
         padding: EdgeInsets.zero,
         children: <Widget>[
           UserAccountsDrawerHeader(//2
             accountName: Text('刘望舒'),
             accountEmail: Text('liuwangshu.gmail.com'),
             currentAccountPicture: CircleAvatar(
               child: Text('X'),
             ),
           ),
           ListTile(
             leading: Icon(Icons.local_post_office),
             title: Text('邮件'),
           ),
           ListTile(
             leading: Icon(Icons.settings),
             title: Text('设置'),
           )
         ],
       ),
     );
}
```

与以往示例不同的是，由于 drawer 属性的代码比较多，为了提高可读性，我将 drawer

属性的值抽取出来，通过 getter 的形式进行获取。注释 1 处可以看出 Drawer 的子项为 ListView，ListView 通过 ListTile 来显示每一个列表项。注释 2 处使用的是 UserAccountsDrawerHeader，可以很轻松地设置用户的姓名、邮箱、图片等。Drawer 效果如图 13-34 所示。

图 13-34　Drawer 效果

13.5　Scrolling Widget

我们知道 Widget 的种类有很多，比如 Basics、Material Components、Cupertino、Scrolling 等，在前面的章节中我介绍了 Basics 和 Material Components，本节讲解 Scrolling 分类中比较常用的 ListView、GridView、PageView。

13.5.1　ListView

ListView 可以说是 Flutter 中十分常用的 Scrolling Widget。ListView 有以下四种构造函数。

（1）默认构造函数 ListView。

（2）ListView.builder：适用于具有大量（或无限）列表项的 ListView。

（3）ListView.separated：可以配置分割线，适用于具有固定数量列表项的 ListView。

（4）ListView.custom：提供了自定义子 Widget 的能力。

1．默认构造函数

使用默认构造函数来实现一个简单的列表，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('ListView 示例'),
        ),
        body: ListView(
          children: <Widget>[
            ListTile(
              leading: Icon(Icons.access_time),
              title: Text('第 1 行'),
            ),
            ListTile(
              leading: Icon(Icons.access_time),
              title: Text('第 2 行'),
            ),
          ],
        ),
      ),
    );
  }
}
```

ListTile 是 Material Components 中的 Widget，它通常用于填充 ListView，可以满足基本的图片+文字的列表需求，如图 13-35 所示。

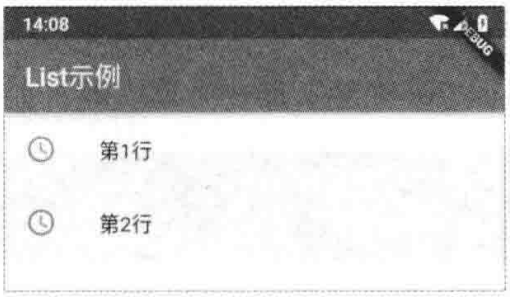


图 13-35　ListView 默认构造函数

2. ListView.builder 创建

如果要展示大量列表项，那么可以使用 ListView.builder，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp(
    items: new List<String>.generate(300, (i) => "第$i 行"), //1
  ));

class MyApp extends StatelessWidget {
 final List<String> items;

 MyApp({@required this.items});
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Flutter',
   home: Scaffold(
    appBar: AppBar(
     title: Text('ListView 示例'),
    ),
    body: ListView.builder( //2
     itemCount: items.length,
     itemBuilder: (context, index) {
      return ListTile(
       leading: Icon(Icons.access_time),
       title: Text('${items[index]}'),
      );
     },
    ),
   ),
  );
 }
}
```

注释 1 处生成了 300 个数据，并将其传入 MyApp 中，注释 2 处通过 ListView.builder 创建 ListView，最终会通过 ListTile 来显示每一个列表项，如图 13-36 所示。

图 13-36　ListView.builder 创建 ListView

3. ListView.separated 创建

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp(
    items: new List<String>.generate(300, (i) => "第$i 行"),
  ));

class MyApp extends StatelessWidget {
 final List<String> items;

 MyApp({@required this.items});

 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Test',
   home: Scaffold(
    appBar: AppBar(
```

```
        title: Text('ListView 示例'),
      ),
      body: ListView.separated(
        itemCount: items.length,
        itemBuilder: (context, index) {
          return ListTile(
            leading: Icon(Icons.access_time),
            title: Text('${items[index]}'),
          );
        },
        separatorBuilder: (context, index) {//1
          return Container(
            constraints: BoxConstraints.tightFor(height: 1),
            color: Colors.black45,
          );
        },
      ),
    ),
  );
 }
}
```

这种创建方式与 ListView.builder 类似，二者不同的是，通过注释 1 处的 separatorBuilder 可以设置分割线及分割线的样式，如图 13-37 所示。

图 13-37　ListView.separated 创建 ListView

13.5.2 GridView

GridView 的使用方法与 ListView 类似，它有以下 5 种构造函数。

（1）默认构造函数 GridView。

（2）GridView.count：适用于在横轴方向上具有固定数量的 GridView。

（3）GridView.extent：适用于在横轴方向上具有最大范围的 GridView。

（4）GridView.builder：适用于具有大量（或无限）列表项的 GridView。

（5）GridView.custom：提供了自定义子 Widget 的能力。

这里以第 2 种构造函数为例，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: new Text('GridView 示例'),
        ),
        body: GridView.count(
          crossAxisCount: 3, //1
          children: <Widget>[
            ListTile(
              title: Text('item1'),
            ),
            ListTile(
              title: Text('item2'),
            ),
            ListTile(
              title: Text('item3'),
            ),
            ListTile(
              title: Text('item4'),
            ),
            ListTile(
              title: Text('item5'),
```

```
          ),
          ListTile(
            title: Text('item6'),
          ),
          ListTile(
            title: Text('item7'),
          ),
          ListTile(
            title: Text('item8'),
          ),
          ListTile(
            title: Text('item9'),
          ),
        ],
      ),
    ),
  );
 }
}
```

注释 1 处的 crossAxisCount 用于设置横轴 item 的数量。GridView 示例效果如图 13-38 所示。

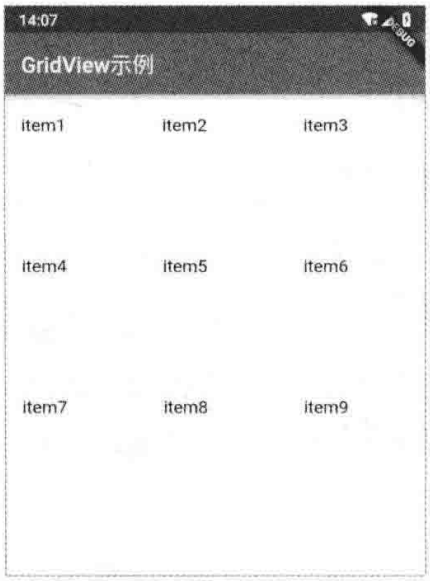


图 13-38　GridView 示例效果

13.5.3　PageView

PageView 是一个可逐页滚动的列表，与 Android 中 ViewPage 类似。PageView 有以下 3 种构造函数。

（1）默认构造函数 PageView。

（2）PageView.builder：适用于具有大量（或无限）列表项的 PageView。

（3）PageView.custom：提供了自定义子 Widget 的能力。

这里以默认构造函数为例，代码如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('PageView 示例'),
        ),
        body: PageView(
          onPageChanged: (index) {//1
            print('当前为第 $index 页');
          },
          children: <Widget>[
            ListTile(
              title: Text('第 0 页'),
            ),
            ListTile(
              title: Text('第 1 页'),
            ),
            ListTile(
              title: Text('第 2 页'),
            ),
          ],
        ),
      ),
    );
  }
}
```

通过注释 1 处的 onPageChanged 属性可以得知当前滑动到了第几页。PageView 示例效果如图 13-39 所示。

图 13-39　PageView 示例效果

由于篇幅的原因，本节并没有介绍每个 Widget 的构造函数，我们掌握了其中一种构造函数基本就可以举一反三了。Scrolling Widget 还包含了一些其他 Widget，比如 NestedScrollView、Scrollbar、CustomScrollView、NotificationListener 等，感兴趣的读者可以自行查看官方文档。

13.6　手势相关 Widget

在移动开发中，用户交互是一个重要的环节，在 Android 中，触摸、单击、滑动等事件处理都提供了相关的 API，在 Flutter 中也是一样的，它们是由 Widget 来实现的。

Flutter 中的手势系统有两个独立的层。第一层是原始指针事件（pointer events），它描述了屏幕上的指针，比如触摸、鼠标、触控笔的位置和移动。 第二层是手势，由一个或多个指针移动组成的动作会被识别为不同的手势。

13.6.1　指针事件

指针表示用户与设备屏幕交互的原始数据。指针事件有如下四种类型。

- PointerDownEvent：指针接触到屏幕的特定位置。
- PointerMoveEvent：指针已从屏幕上的一个位置移动到另一个位置。
- PointerUpEvent：指针已停止接触屏幕。

- PointerCancelEvent：此指针的输入不再指向此应用，通俗来讲就是事件取消。

在指针按下时，Flutter 框架会对当前应用程序执行命中测试，以确定指针与屏幕接触的位置在哪个 Widget 上，然后将 PointerDownEvent 事件（及该指针的后续事件）调度到命中测试最里面的 Widget，事件的分配路径为：从最里面的 Widget 到树的根路径上的所有 Widget。

13.6.2 手势

手势表示由一个或多个指针移动组成的动作。手势主要有以下几种。

单击

onTapDown：指针已经在特定位置与屏幕接触。

onTapUp：指针停止在特定位置与屏幕接触。

onTap：单击事件触发。

onTapCancel：先前指针触发的 onTapDown 不会再触发单击事件。

双击

onDoubleTap：用户快速连续两次在同一位置轻敲屏幕。

长按

onLongPress：指针在相同位置长时间保持与屏幕接触。

垂直拖动

onVerticalDragStart：指针已经与屏幕接触并可能开始垂直移动。

onVerticalDragUpdate：指针与屏幕接触并已沿垂直方向移动。

onVerticalDragEnd：先前与屏幕接触并垂直移动的指针不再与屏幕接触，并且在停止接触屏幕时以特定速度移动。

水平拖动

onHorizontalDragStart：指针已经接触到屏幕并可能开始水平移动。

onHorizontalDragUpdate：指针与屏幕接触并已沿水平方向移动。

onHorizontalDragEnd：先前与屏幕接触并水平移动的指针不再与屏幕接触，并在停止接触屏幕时以特定速度移动。

检测这些手势可以使用 GestureDetector。

13.6.3　使用 GestureDetector

要想检测单击、双击、垂直拖动等手势，只要使用 GestureDetector 嵌套要检测的手势 Widget 并实现想要监听的手势的方法即可，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(GestureDetectorWidget());

class GestureDetectorWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: Scaffold(
        appBar: AppBar(
          title: Text("GestureDetector 示例"),
        ),
        body: Center(
          child: GestureDetector(
            child: Text('手势识别'),
            onTap: () {
              print('单击');
            },
            onDoubleTap: () {
              print('双击');
            },
            onLongPress: () {
              print('长按');
            },
            onHorizontalDragStart: (DragStartDetails details) {
              print('水平拖动');
            },
          ),
        ),
      ),
    );
  }
}
```

只需要在手势识别这个文字上进行操作，对应的手势就会被打印出来，运行效果如图 13-40 所示。

图 13-40　使用 GestureDetector 的运行效果

13.6.4　使用 Dismissible

滑动删除这个操作很常见，比如在一个列表中，我们向左滑动，就会直接删除一个条目或者弹出删除提示选项。Flutter 提供了 Dismissible 来帮助我们实现滑动删除，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(DismissibleWidget(
    items: new List<String>.generate(300, (i) => "第$i 行"),
  ));

class DismissibleWidget extends StatelessWidget {
  final List<String> items;

  DismissibleWidget({@required this.items});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter',
      home: Scaffold(
        appBar: AppBar(
```

```
      title: Text('Dismissible 示例'),
    ),
    body: ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        final item = items[index];
        return Dismissible(
          key: Key(item),
          onDismissed: (direction) {
            items.removeAt(index);
            print(index);
          },
          child: ListTile(
            leading: Icon(Icons.access_time),
            title: Text('${items[index]}'),
          ),
        );
      },
    ),
  ),
);
  }
}
```

上述代码示例和ListView的示例类似，其主要的变化就是用Dismissible来嵌套ListTile。当执行删除操作时，ListView 中的 onDismissed 方法会被回调，我们可以直接在 onDismissed 方法中将被删除的 item 从 List 中移除，运行效果如图 13-41 所示。

图 13-41　使用 Dismissibl 的运行效果

我们向左滑动第一个 item 和第二个 item，会出现删除的动画，结果如图 13-42 所示。

图 13-42　删除 item

13.7　资源和图片

和 Android 开发一样，Flutter 也有 asset 这一概念，asset 是打包到程序安装包中的，可在运行时访问。常见的 asset 类型包括静态数据（例如 JSON 文件）、配置文件、图标和图片（JPEG、WebP、GIF、动画 WebP / GIF、PNG、BMP 和 WBMP）。本节简单介绍如何在 Flutter 中添加资源和图片。

13.7.1　指定 asset

要想使用 asset，需要先让 asset 被识别，在项目根目录中的 pubspec.yaml 中定义图片就可以了，如下所示。

```
flutter:
  assets:
    - assets/1.png
    - assets/2.png
```

在构建期间，Flutter 会将 asset 放置到名称为 asset bundle 的特殊存档中，应用程序可

以在运行时读取它们。 如果想要包含某一个目录中的所有资源，则直接指定目录的名称，如下所示。

```
flutter:
  assets:
    - assets/
```

这种指定只包含直接位于目录中的文件，如果想要添加位于子目录中的文件，那么就需要在 pubspec.yaml 中为每个目录添加一个条目。

在示例中我们使用的是 asset 目录，这个目录可以是任意文件夹，比如可以在根目录中创建一个 images 目录来存储图片，可以这样写。

```
flutter:
  assets:
    - images/1.png
    - images/2.png
```

13.7.2　加载文本

每个 Flutter 应用程序都有一个 rootBundle 对象，rootBundle 对象可以轻松访问主资源包，也就是使用 package:flutter/services.dart 中全局静态的 rootBundle 对象来加载 asset。

不过官网建议使用 DefaultAssetBundle.of 来获取当前 BuildContext 的 AssetBundle。 这种方法不是使用应用程序构建的默认 asset，而是允许父窗口 Widget 在运行时替换不同的 AssetBundle，这对于本地化或测试场景非常有用。

接下来写一个示例来加载文本。在根目录中新建一个 assets 文件，并新建一个 swordsmen.json 文件，内容如下所示。

assets/swordsmen.json

```
[
  {
    "name": "张无忌",
    "gongfu":"乾坤大挪移",
  },
  {
    "name": "令狐冲",
    "gongfu": "独孤九剑",
  }
]
```

然后在 pubspec.yaml 配置该 asset，如下所示。

```
flutter:
  assets:
    - assets/swordsmen.json
```

最后加载这个 json 文件，并把它们显示到界面上，如下所示。

```
import 'package:flutter/material.dart';
import 'dart:convert' show json;
void main() => runApp(AssetsWidget());

class AssetsWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: Scaffold(
        appBar: AppBar(
          title: Text("加载文本示例"),
        ),
        body:JsonWidget(),
      ),
    );
  }
}
class JsonWidget extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return _JsonWidgetState();
  }
}
class _JsonWidgetState extends State<JsonWidget> {
  @override
  Widget build(BuildContext context) {
    return FutureBuilder(//1
      future:
DefaultAssetBundle.of(context).loadString("assets/swordsmen.json"),//2
      builder: (context, snapshot) {
        if (!snapshot.hasData) {//3
          return Center(
            child: CircularProgressIndicator(),
          );
        }else{
          List<dynamic> data = json.decode(snapshot.data.toString());//4
```

```
        return ListView.builder(
          itemCount: data.length,
          itemBuilder: (BuildContext context, int index) {
            return Card(
              child: Column(
                crossAxisAlignment: CrossAxisAlignment.stretch,
                children: <Widget>[
                  Text("名字: ${data[index]["name"]}"),
                  Text("绝学: ${data[index]["gongfu"]}"),
                ],
              ),
            );
          },
        );
      }
    },
  );
}
}
```

关键代码都在_JsonWidgetState 中，注释 1 处的 FutureBuilder 是异步模型，使用它可以很容易得到当前 Widget 的状态，并在加载数据时显示不同的内容，比如网络请求数据，如果数据没有返回就显示加载界面，如果数据返回就显示列表，如果数据加载失败就显示失败界面。注释 2 处获取当前 BuildContext 的 AssetBundle，然后加载 assets/swordsmen.json 中的数据，将数据和状态等信息存储在快照 snapshot 中。注释 3 处如果 snapshot 没有数据，就显示 CircularProgressIndicator，它是一个圆形进度条。如果有数据就在注释 4 处将 snapshot 中的数据解析并存储到 List 中，用 ListView 来显示。上述代码中用到了 dart:convert 库，它是在不同数据之间转换的编码器和解码器，这里用于将 Josn 数据转换为 List。

“加载文本示例”效果如图 13-43 所示。

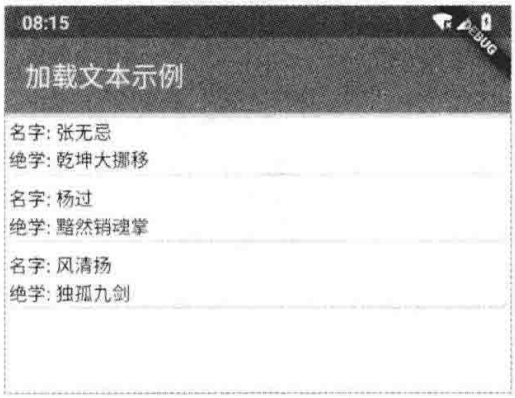


图 13-43 “加载文本示例”效果

在 Widget 上下文之外，或者当 AssetBundle 的句柄不可用时，也可以使用 rootBundle 直接加载此类资源，如下所示。

```
import 'dart:async' show Future;
import 'package:flutter/services.dart' show rootBundle;

Future<String> loadAsset() async {
  return await rootBundle.loadString('assets/swordsmen.json');
}
```

对应于本节开头加载文本的示例，修改注释 2 处的代码，并引入'package:flutter/services.dart' 包就可以了。

```
import 'package:flutter/services.dart' show rootBundle;
...
future: rootBundle.loadString("assets/swordsmen.json"),
...
```

13.7.3 加载图片

在项目根目录中创建 images 文件夹，然后放入两张图片，并在 pubspec.yaml 中配置，如下所示。

```
flutter:
  assets:
    - images/light.png
    - images/decode.png
```

接下来我们在代码中加载图片文件，如下所示。

```
import 'package:flutter/material.dart';
void main() => runApp(AssetsWidget());

class AssetsWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: Scaffold(
        appBar: AppBar(
          title: Text("加载图片示例"),
        ),
        body: ImageWidget(),
      ),
```

```
    );
  }
}

class ImageWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        children: <Widget>[
          Image.asset(
            'images/decode.png',
            height: 200.0,
          ),
          Image.asset(
            'images/light.png',
            height: 200.0,
          ),
        ],
      ),
    );
  }
}
```

通过 Image.asset 就可以加载 images 文件夹中的图片，效果如图 13-44 所示。

图 13-44　通过 Image.asset 加载图片

1. 加载不同分辨率的图片

假设主要资源对应的分辨率为 1.0，比如 72px × 72px 的图片，如果还有 144px × 144px 和 216px × 216px 的图片，那么可以在 images 文件夹下创建两个文件夹，2.0x 和 3.0x，将对应分辨率的图片复制进去，2.0x 对应的是 144px × 144px，3.0x 对应的是 216px × 216px。

…images/decode.png

…images/2.0x/decode.png

…images/3.0x/decode.png

在 pubspec.yaml 配置对应图片资源：

```
flutter:
 assets:
  - images/decode.png
```

Flutter 可以为当前设备加载适合其分辨率的图像，如果设备像素比率为 1.8，那么会选择…images/2.0x/decode.png ，如果设备像素比率为 2.8，那么会选择…images/3.0x/decode.png，设备选择的是相近的图片资源。

2. 加载依赖包中的图片

假设我们的应用程序依赖于一个名为 best_icons 的包，它具有以下目录结构。

.../pubspec.yaml

.../icons/add.png

.../icons/1.5x/phone.png

.../icons/2.0x/phone.png

可以使用 AssetImage 来加载图片。

```
AssetImage('icons/phone.png', package: 'best_icons')
```

本节介绍了加载图片，除了加载目录和依赖包的图片，还可以使用平台的图片，具体的内容见官方文档。

13.8　布局 Widget

Flutter 的布局在本章使用过，但是没有具体讲解，Flutter 的布局实际上也由 Widget 来控制的，在 Flutter 官网上并没有对布局 Widget 进行分类，这里将布局 Widget 根据子元素排列方式分为以下几种。

- 线性布局 Widget。
- 流式布局 Widget。
- 层式布局 Widget。
- 弹性布局 Widget。

下面分别介绍这几种布局 Widget。

13.8.1　线性布局 Widget

线性布局类似于 Android 中的 LinearLayout，可以垂直排列也可以水平排列，Flutter 的线性布局有以下两个。

- Row：水平方向的线性布局。
- Column：垂直方向的线性布局。

Row 和 Column 的使用方法类似，能够掌握 Row，Column 自然而然也就掌握了。这里以 Row 为例，代码如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(RowWidget());

class RowWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: Scaffold(
        appBar: AppBar(
          title: Text("Row 示例"),
        ),
        body: Padding(
          padding: EdgeInsets.all(40.0),
          child: Row(
```

```
          children: <Widget>[
            Text("Android进阶之光"),
            Text("Android进阶解密"),
          ],
        ),
      ),
    ),
  );
 }
}
```

Row 示例运行效果如图 13-45 所示。

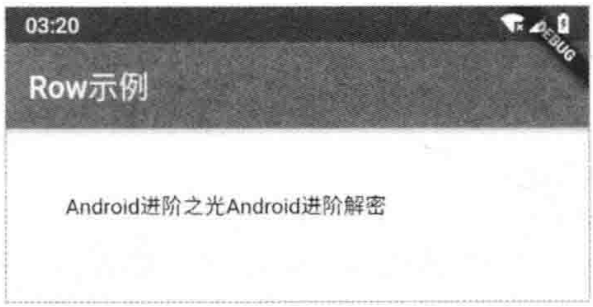


图 13-45　Row 示例运行效果

13.8.2　流式布局 Widget

流式布局就是把超出屏幕显示范围的控件自动换行，在 Android 中并没有现成的流式布局控件，都是开发人员自己封装的，比如鸿洋封装的 FlowLayout。Flutter 中提供了 Wrap 来实现流式布局，代码如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(ChipWidget());

class ChipWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: "Flutter",
   home: Scaffold(
    appBar: AppBar(
     title: Text("流式布局示例"),
    ),
    body: Padding(
     padding: EdgeInsets.all(20.0),
```

```
        child: Wrap(
          direction: Axis.horizontal,  //主轴的方向
          spacing: 8.0,                // 主轴方向的间距
          runSpacing: 12.0,            // 交叉轴方向的间距
          children: <Widget>[
            Chip( //1
              avatar: CircleAvatar(
                 backgroundColor: Colors.blue, child: Text('1')),
              label: Text('Android进阶之光'),
            ),
            Chip(
              avatar: CircleAvatar(
                 backgroundColor: Colors.blue, child: Text('2')),
              label: Text('Android进阶解密'),
            ),
            Chip(
              avatar: CircleAvatar(
                 backgroundColor: Colors.blue, child: Text('3')),
              label: Text('Android进阶? '),
            ),
          ],
        ),
      ),
    ),
  );
 }
}
```

注释 1 处 Chip 的含义为碎片，实际上它是一个标签 Widget，这个代码中定义了三个标签来体现流式布局。

Wrap 有很多属性，它的构造函数如下。

```
Wrap({
  Key key,
  this.direction = Axis.horizontal, //1
  this.alignment = WrapAlignment.start,//2
  this.spacing = 0.0,
  this.runAlignment = WrapAlignment.start,
  this.runSpacing = 0.0,
  this.crossAxisAlignment = WrapCrossAlignment.start,
  this.textDirection,
  this.verticalDirection = VerticalDirection.down,
  List<Widget> children = const <Widget>[],
```

```
  }) : super(key: key, children: children);
```

注释 1 处的 direction 属性用于设定主轴的方向（子 Widget 排列的方向），默认值为 Axis.horizontal，Wrap 属性参数类型和含义如表 13-2 所示。

表 13-2　Wrap 属性参数类型和含义

参数名称	参数类型	含　义
direction	Axis	主轴的方向
alignment	WrapAlignment	子 Widget 在主轴上的对齐方式
runAlignment	WrapAlignment	每行或每列的对齐方式
runSpacing	double	每行或每列之间的间距
crossAxisAlignment	WrapCrossAlignment	子 Widget 在交叉轴上的对齐方式
textDirection	TextDirection	子 Widget 在主轴方向上的布局顺序
verticalDirection	VerticalDirection	子 Widget 在交叉轴方向上的布局顺序

流式布局示例运行效果如图 13-46 所示。

图 13-46　流式布局示例运行效果

13.8.3　层式布局 Widget

层式布局类似于 Android 中的 FrameLayout，在 Flutter 可以通过 Stack 来实现层式布局，其子 Widget 会按照添加顺序确定显示层级，后面添加的会覆盖在前面添加的 Widget 上面。

为了确定子 Widget 到父容器四个角的位置，Stack 将子 Widget 分为以下两类。

- Positioned：Positioned 嵌套的子 Widget，Positioned 可以控制子 Widget 到父容器四个边的距离。

- non-positioned：没有用 Positioned 嵌套的子 Widget，使用 Stack 的 alignment 属性来控制自身在父容器中的位置。

```
import 'package:flutter/material.dart';

void main() => runApp(StackWidget());

class StackWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: "Flutter",
   home: Scaffold(
    appBar: AppBar(
     title: Text("层式布局示例"),
    ),
    body: Stack(
     fit: StackFit.expand,
     children: <Widget>[
      Positioned(
       left: 90.0,
       top: 120.0,
       child: Image.asset(
        "images/decode.png",
        width: 200.0,
        fit: BoxFit.cover,
       ),
      ),
      Container(
       child: Text('Android进阶三部曲第二部',
          style: TextStyle(
             color: Colors.lightBlue, fontSize: 20.0)),
       height: 50.0,
       width: 100.0,
       alignment: Alignment.center,
      )
     ],
    ),
   ),
  );
 }
}
```

上述代码中，Stack 分为两部分，一部分是 Positioned Widget，用于显示一个图片；另

一部分是 non-positioned Widget（Text），用于显示一行文字，并嵌套在 Container 中。层式布局示例效果如图 13-47 所示。

图 13-47　层式布局示例效果

13.8.4　弹性布局 Widget

Flutter 中的弹性布局是 Flex，类似于 Android 中的 FlexboxLayout。为了避免子 Widget 在 Flex、Row、Column 中超界，可以使用 Flexible 和 Expanded，它们可以使 Flex、Row、Column 的子 Widget 具有弹性能力，下面以 Flex 和 Expanded 搭配使用为例。

```
import 'package:flutter/material.dart';

void main() => runApp(FlexWidget());

class FlexWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: Scaffold(
        appBar: AppBar(
          title: Text("弹性布局示例"),
        ),
        body: Flex(
          direction: Axis.horizontal,//1
          children: <Widget>[
```

```
          Expanded(
            flex: 1,
            child: Container(
              height: 60.0,
              width: 30.0,
              color: Colors.red,
            ),
          ),
          Expanded(
            flex: 3,
            child: Container(
              height: 60.0,
              width: 30.0,
              color: Colors.yellow,
            ),
          ),
          Expanded(
            flex: 1,
            child: Container(
              height: 60.0,
              width: 30.0,
              color: Colors.blue,
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

注释 1 处的 direction 属性用于设置主轴的方向，这是必须要设置的。Flex 的子 Widget 是三个 Expanded，每个 Expanded 负责显示一个矩形色块，用 flex 来设置弹性系数，三个矩形色块的比例为 1：3：1，效果如图 13-48 所示。

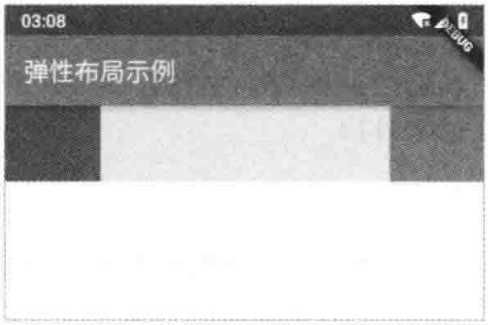


图 13-48　弹性布局示例效果

13.9 网络请求库 Dio

在 Android 开发中，如果我们想要请求网络，那么可以使用 HttpClent 和 HttpURLConnection，但在项目中一般都会使用 OkHttp 和 Retrofit。在 Flutter 中也是如此，系统提供了 HttpClient，但在项目中一般会使用第三方库，比如 http 和 Dio，这一节我们来学习 Dio 的基本使用。

13.9.1 Dio 的使用入门

Dio 是一个强大的 Dart Http 请求库，支持拦截器、全局配置、FormData、请求取消、文件下载、超时等。

在 pubspec.yaml 中添加依赖。

```
dependencies:
  dio: 2.1.7
```

官方网站给出了一个简单的示例，如下所示。

```
import 'package:dio/dio.dart';
void getHttp() async {
  try {
    Response response = await Dio().get("http://liuwangshu……");
    print(response);
  } catch (e) {
    print(e);
  }
}
```

如果想要发送一个 GET 请求：

```
Response response;
response=await dio.get("/test?id=3&name=liuwangshu")
print(response.data.toString());
// 请求参数也可以通过对象传递，上面的代码等同于：
response=await dio.get("/test",data:{"id":3,"name":"liuwangshu"})
print(response.data.toString());
```

发送一个 POST 请求：

```
response=await dio.post("/test",data:{"id":3,"name":"liuwangshu"})
```

发送一个 FormData：

```
FormData formData = new FormData.from({
   "name": "liuwangshu",
   "age": 18,
  });
response = await dio.post("/info", data: formData);
```

虽然还有很多示例在 GitHub 上，但是本节的目的不是重复介绍这些示例，而是写一个网络访问的示例带领读者快速入门。

13.9.2　Dio 访问网络

Dio 访问网络的代码如下所示。

```
import 'dart:convert';
import 'dart:io';
import 'package:dio/dio.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key}) : super(key: key);
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  var _ipAddress = 'Unknown';

  _getIPAddress() async {//1
    var url = 'https://httpbin……';
    Dio _dio = Dio();
    String result;
    try {
```

```
      var response = await _dio.get(url);//2
      if (response.statusCode == HttpStatus.ok) {
        var data= jsonDecode(response.toString());//3
        result = data['origin'];
      } else {
        result =
        'Error getting IP status ${response.statusCode}';
      }
    } catch (exception) {
      result =exception.toString();
    }
    if (!mounted) return;
    setState(() {
      _ipAddress = result;//4
    });
  }

  @override
  Widget build(BuildContext context) {
    var spacer = SizedBox(height: 10.0);
    return Scaffold(
      body: Padding(
        padding: EdgeInsets.all(100.0),
        child: Column(
          children: <Widget>[
            Text('IP 地址为: '),
            spacer,
            Text('$_ipAddress'),
            spacer,
           RaisedButton(
              onPressed: _getIPAddress,
              child: Text('请求网络'),
            ),
          ],
        ),
      ),
    );
  }
}
```

async 和 await 是 Dart 语言用来支持异步编程的关键字，注释 1 处的 async 关键字使 getIPAddress 方法变为了异步方法，注释 2 处的 await 关键字只能在异步方法中使用，它使注释 2 处后面的代码等到 get 请求返回后再执行。

如果网络请求返回的状态码为 200，那么就在注释 3 处进行 Json 解析，将结果赋值给 result 变量。

当我们单击界面上的“请求网络”按钮时，会调用_getIPAddress 方法，在注释 4 处将请求的结果赋值给_ipAddress 并显示在 Text 中。

网络请求成功后，效果如图 13-49 所示。

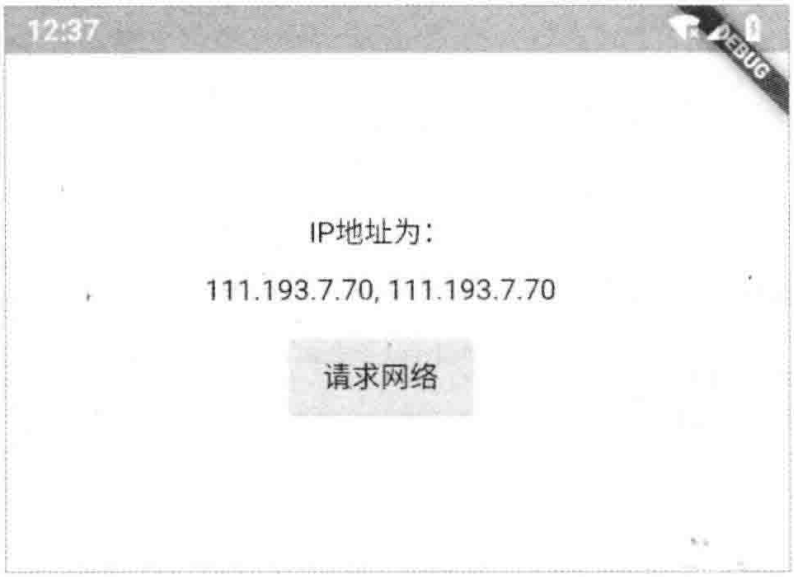


图 13-49　网络请求成功

13.10　JSON 数据解析

常用的 JSON 数据解析方式主要有三种，下面分别对其进行介绍。

13.10.1　使用 json.decode 方法

13.9.2 节中的示例用到的就是 json.decode 方法，需要引入 dart:convert 库。json.decode 方法会将 String 类型数据解析成 Map 数据结构：Map<String, dynamic>，取数据的格式为 object[key]。

13.9.2 节中返回的 JSON 数据为如下。

```
{
  "origin": "111.193.7.70, 111.193.7.70"
}
```

对于不复杂的 JSON 数据，使用 json.decode 方法是一个不错的选择，但实际项目中的 JSON 数据会复杂一些，如果每次取数据都用 object[key]，那么很容易出错。

13.10.2 手动编写实体类

用一个 model 来保存数据，使得数据序列化。

我们可以新建一个 ip.dart。

```
class Ip {
  String origin;

  Ip(this.origin);

  Ip.fromJson(Map<String, dynamic> json) : origin = json['origin'];

  Map<String, dynamic> toJson() => {
      "origin": origin,
    };
}
```

然后修改 main.dart。

```
import 'dart:convert';
import 'dart:io';
import 'package:dio/dio.dart';
import 'package:flutter/material.dart';
import 'model/ip.dart';
...
class _MyHomePageState extends State<MyHomePage> {
  var _ipAddress = 'Unknown';
  _getIPAddress() async {
    var url = 'https://httpbin……';
    Dio _dio = Dio();
    String result;
    try {
      var response = await _dio.get(url);
      if (response.statusCode == HttpStatus.ok) {
        var data= jsonDecode(response.toString());
        var ip=Ip.fromJson(data);//1
        result = ip.origin;
      }
    ...
  }
```

```
...
}
```

注释 1 处通过 Ip.fromJson 就可以获取 ip 的实体，这样就可以完成赋值等操作了。

13.10.3　自动生成实体类

在一般项目中 Json 数据会比较多，每次重复写实体类的模板代码显然枯燥无意义，可以使用一些工具来生成实体类。

比如使用网页自动生成实体类（搜索 quicktype）。转换的实体类如下所示。

```
import 'dart:convert';

Ip ipFromJson(String str) => Ip.fromJson(json.decode(str));

String ipToJson(Ip data) => json.encode(data.toJson());

class Ip {
   String origin;

   Ip({
      this.origin,
   });

   factory Ip.fromJson(Map<String, dynamic> json) => new Ip(
      origin: json["origin"],
   );

   Map<String, dynamic> toJson() => {
      "origin": origin,
   };
}
```

除了使用网页生成实体类，还可以使用 json_serializable。

json_serializable 是一个自动化的代码生成器，生成 JSON 实体类很方便。

首先在 pubspec.yaml 加入如下依赖。

```
dependencies:
  json_annotation: ^2.4.0

dev_dependencies:
  build_runner: ^1.0.0
```

```
  json_serializable: ^3.0.0
```

在 Android Studio 的 Terminal 中运行 flutter packages get 命令，该命令用来在项目中使用这些新的依赖项。然后实现 ip 实体类。

```
import 'package:json_annotation/json_annotation.dart';

// ip.g.dart 将在我们运行生成命令后自动生成
part 'ip.g.dart';

//告诉生成器，这个类需要生成 Model 类
@JsonSerializable()
class Ip{
  Ip(this.origin);

  String origin;

  factory Ip.fromJson(Map<String, dynamic> json) => _$IpFromJson(json);
  Map<String, dynamic> toJson() => _$IpToJson(this);
}
```

上述代码的关键在于@JsonSerializable()，它用于告诉生成器，Ip 类需要生成 Model 类。在 Terminal 窗口运行如下命令生成 ip.g.dart 文件。

```
flutter packages pub run build_runner build
```

如何使用见 13.10.2 节即可，使用 json_serializable 可以让我们忽略 Ip 类中的任何手动 JSON 序列化。代码生成器会在同一个目录下生成 ip.g.dart 文件，如下所示。

```
part of 'ip.dart';
Ip _$IpFromJson(Map<String, dynamic> json) {
  return Ip(json['origin'] as String);
}

Map<String, dynamic> _$IpToJson(Ip instance) =>
    <String, dynamic>{'origin': instance.origin};
```

该文件有必需的序列化逻辑，目前 Flutter 官方推荐的就是使用 json_serializable。

13.11　路由与数据传递

在 Android 开发中使用 Intent 来进行页面跳转，我们称之为原生路由，后来出现了一些路由框架，比如 ARouter。

在 Flutter 中进行界面跳转的就是路由，路由用 Route 类来表示，Navigator 是对 Route 进行管理的 Widget。这一节我们来学习路由和数据传递。Flutter 路由的使用方式主要有两种，一种是新建路由，另一种是注册路由。

13.11.1　新建路由

创建两个页面，第一个页面有一个按钮，单击这个按钮跳转到第二个页面。首先来实现第一个页面，代码如下所示。

```
class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => SecondPage(), maintainState: false));
          },
        ),
      ),
    );
  }
}
```

通过 Navigator.push 方法将一个 Route 对象添加到 Navigator 管理的 Route 堆栈中，这里的 Route 对象是一个 MaterialPageRoute，它自带页面切换动画，并且适配了 Android 和 iOS，如果是 Android，那么页面进入动画是向上滑动并淡出的，退出是相反的；如果是 iOS，那么页面进入动画是从右侧滑入的，退出同样是相反的。一般来说使用 MaterialPageRoute

就够用了，如果不能满足需求，那么可以实现自定义 Route。

接着来实现第二个页面，代码如下所示。

```
class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第二页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("返回到第一页"),
          onPressed: () {
            Navigator.pop(context);
          },
        ),
      ),
    );
  }
}
```

Navigator.pop 方法用于关闭当前页面，返回上一个页面，并将当前的 Route 对象从 Navigator 管理的 Route 堆栈中移除。

完整的代码如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      home: FirstPage(),
    );
  }
}

class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => SecondPage(),
                maintainState: false,
              ),
            );
          },
        ),
      ),
    );
  }
}

class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第二页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("返回到第一页"),
          onPressed: () {
            Navigator.pop(context);
          },
        ),
      ),
    );
  }
}
```

路由跳转界面如图 13-50 所示。

图 13-50　路由跳转界面

13.11.2　使用注册路由

如果很多页面都跳转到了同一个页面，那么每次都要新建路由，会写很多重复的代码，这时使用注册路由就可以简化。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      initialRoute: '/First',//1
      routes: {
        '/First': (context) => FirstPage(),
        "/Second": (context) => SecondPage()
      },
      home: FirstPage(),
```

```
    );
  }
}

class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            Navigator.pushNamed(context, '/Second');//2
          },
        ),
      ),
    );
  }
}
...
```

这里省略了第二页的代码，因为第二页没有变化。代码改变的是我们需要定义的路由表，路由表的格式为 Map<String, WidgetBuilder> routes；其中 key 为路由名称，value 是 builder 回调函数，用于生成相应的路由 Widget。通过路由名称入栈新路由时，应用会根据路由名称在路由表中找到对应的 WidgetBuilder 回调函数，通过调用该回调函数生成路由 Widget 并返回。

注释 1 处用来定义初始路由的页面，接着注册路由表，里面包含了两个页面，分别是 FirstPage 和 SecondPage。注释 2 处通过路由名称来打开 SecondPage。

13.11.3　路由间数据传递

页面跳转的时候经常需要传递数据，传递数据一般有两种情况，一种是页面跳转数据传递，另一种是页面跳转返回数据，这里分别对其进行介绍。

1．页面跳转数据传递

一个页面跳转到一个新的页面，数据传递的方式主要有以下两种。

- 通过 Navigator.push 方法或者 Navigator.pushNamed 方法传递数据。
- 通过 Widget 的构造函数传递数据。

一般来说会使用第一种方式传递数据，下面来进行举例。

```
class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            Navigator.pushNamed(context, '/Second',arguments:
CustomArgumnets('Android进阶之光'));
          },
        ),
      ),
    );
  }
}
class CustomArgumnets {
  String content;
  PassArgumnets(this.content);
}
```

我们通过 arguments 属性来传递参数，这里自定义了一个 CustomArgumnets，它用来传递参数。

改写 13.11.1 节 SecondPage 的代码用来获取数据，如下所示。

```
class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final CustomArgumnets customArgumnets
=ModalRoute.of(context).settings.arguments;
    return Scaffold(
      appBar: AppBar(
        title: Text("第二页"),
      ),
      body: Center(
```

```
      child: Column(
        children: <Widget>[
          Text('第一页的数据为: '),
          Text(customArgumnets.content),
          RaisedButton(
            onPressed: () {
              Navigator.pop(context);//1
            },
            child: Text('返回第一页'),
          ),
        ],
      ),
    ),
  );
 }
}
```

运行代码，跳转到第二页，效果如图 13-51 所示。

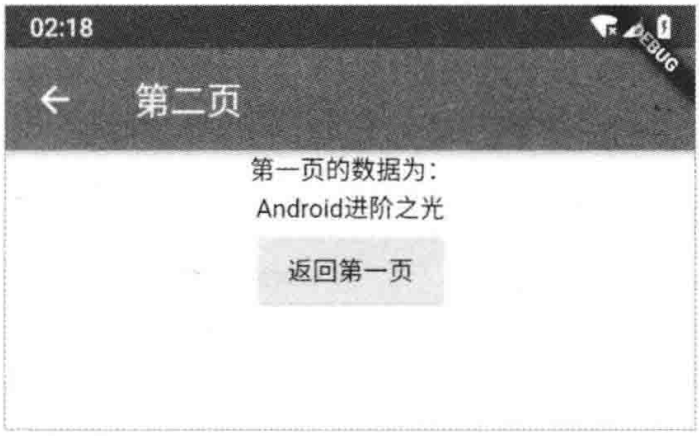


图 13-51　页面跳转数据传递

2. 页面跳转返回数据

除了页面跳转需要传递数据，有时还需要从第二个页面返回数据，改写页面跳转数据传递的 SecondPage 注释 1 处的代码。

```
Navigator.pop(context,CustomArgumnets('Android 进阶解密'));
```

接着改写 FirstPage，如下所示。

```
class FirstPage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return Scaffold(
```

```
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            _navigateToSecondPage(context);
          },
        ),
      ),
    );
  }
  _navigateToSecondPage(BuildContext context) async {
    dynamic customArgumnets = await Navigator.pushNamed(context, '/Second',
        arguments: CustomArgumnets('Android进阶之光'));//1
    print(customArgumnets.content);
  }
}
```

注释 1 处的代码可以接收 java 返回的结果，并且使用异步来防止阻塞 UI。最后贴上完整的代码，如下所示。

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter",
      initialRoute: '/First',
      routes: {
        '/First': (context) => FirstPage(),
        "/Second": (context) => SecondPage()
      },
      home: FirstPage(),
    );
  }
}

class FirstPage extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第一页"),
      ),
      body: Center(
        child: RaisedButton(
          child: Text("跳转到第二页"),
          onPressed: () {
            _navigateToSecondPage(context);
          },
        ),
      ),
    );
  }
  _navigateToSecondPage(BuildContext context) async {
    dynamic customArgumnets = await Navigator.pushNamed(context, '/Second',
        arguments: CustomArgumnets('Android进阶之光'));
    print(customArgumnets.content);
  }
}

class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final CustomArgumnets customArgumnets =ModalRoute.of(context).settings.
arguments;
    return Scaffold(
      appBar: AppBar(
        title: Text("第二页"),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            Text('第一页的数据为: '),
            Text(customArgumnets.content),
            RaisedButton(
              onPressed: () {
                Navigator.pop(context,CustomArgumnets('Android进阶解密'));
              },
              child: Text('返回第一页'),
            ),
          ],
```

```
        ),
      ),
    );
  }
}
class CustomArgumnets {
  String content;
  CustomArgumnets(this.content);
}
```

运行程序，当我们单击“返回第一页”，在控制台中会打印出 Android 进阶解密。

13.12 Flutter 与 Android 的相互通信

由于 Flutte 还在发展阶段，它的 Native 功能并不完善，所以开发者可以编写一些插件来为 Flutter 提供 Native 功能，也可以在 Native 工程中嵌入 Flutter 进行混合开发。Flutter 与 Android 的相互通信是 Flutter 插件开发和混合开发的基础，这一节我们来学习 Flutter 与 Android 的相互通信。

13.12.1 PlatformChannel 概述

由于插件开发和混合开发需要 Flutter 与 Native 进行通信，Flutter 提供了一套 Platform Channel 的机制，来满足 Flutter 与 Native 通信的需求。

下面是 PlatformChannel 架构，如图 13-52 所示。

由图 13-52 可以看到，Flutter 是 Client 端，Native 是 Host 端，Client 和 Host 通信是通过 PlatformChannel 进行的，Client 通过 PlatformChannel 向 Host 发送消息，Host 监听 PlatformChannel 并接收消息，然后将响应结果发送给 Client。消息和响应以异步方式传递，以确保 UI 不阻塞。另外，PlatformChannel 是双工的，这意味着 Flutter 和 Native 可以交替进行 Client 和 Host。

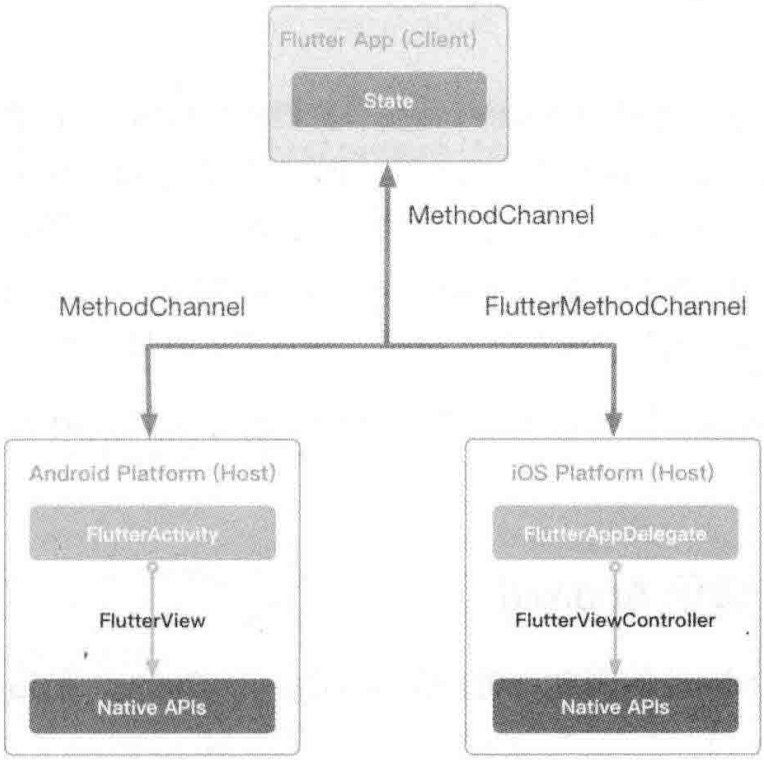


图 13-52　PlatformChannel 架构

Flutter 定义了如下三种不同类型的 PlatformChannel。

- MethodChannel：用于传递方法调用，是比较常用的 PlatformChannel。
- EventChannel：用于传递事件。
- BasicMessageChannel：用于传递数据。

这三种类型 PlatformChannel 的用法都不难，这里以比较常用的 MethodChannel 来举例。在此之前我们首先要了解 BinaryMessenger、Codec、Handler 的概念。

1．BinaryMessenger

BinaryMessenger 是 PlatformChannel 与 Flutter 端的通信工具，其通信使用的消息格式为二进制格式数据，BinaryMessenger 在 Android 中是一个接口，它的实现类为 FlutterNativeView。

2．Codec

Codec 是消息编解码器，主要用于将二进制格式的数据转化为 Handler 能够识别的数据，Flutter 定义了两种 Codec：MessageCodec 和 MethodCodec。MessageCodec 用于二进制格式数据与基础数据之间的编解码，BasicMessageChannel 所使用的编解码器是 MessageCodec。MethodChannel 和 EventChannel 所使用的编解码器均为 MethodCodec。

3. Handler

Flutter 定义了三种类型的 Handler，它们与 PlatformChannel 类型一一对应，分别是 MessageHandler、MethodHandler 和 StreamHandler。在使用 PlatformChannel 时，会为它注册一个 Handler，PlatformChannel 会将该二进制数据通过 Codec 解码器转化为 Handler 能够识别的数据，并交给 Handler 处理。当 Handler 处理完消息后，会通过回调函数返回 result，将 result 通过编解码器编码为二进制格式数据，通过 BinaryMessenger 发送回 Flutter 端。

MethodChannel 可以实现 Flutter 调用 Android，也可以实现 Android 调用 Flutter，这里分别来进行举例。

13.12.2 Flutter 调用 Android

这里实现一个 Android 的简单功能：弹出一个 AlertDialog，然后在 Flutter 中调用这一功能。

1. Android 端实现

先在 MainActivity 中实现该功能，如下所示。

```
package com.example.platform_channel;

import android.app.AlertDialog;
import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        GeneratedPluginRegistrant.registerWith(this);//1

        MethodChannel methodChannel = new MethodChannel(getFlutterView(),
"com.example.platform_channel/dialog");//2
        methodChannel.setMethodCallHandler(new MethodChannel.MethodCallHandler()
{//3
            @Override
```

```
            public void onMethodCall(MethodCall methodCall, MethodChannel.Result
result) {
                if ("dialog".equals(methodCall.method)) {
                    if (methodCall.hasArgument("content")) {
                        showAlertDialog();
                        result.success("弹出成功");
                    } else {
                        result.error("error", "弹出失败", "content is null");
                    }
                } else {
                    result.notImplemented();
                }
            }
            private void showAlertDialog() {
                AlertDialog.Builder builder = new
AlertDialog.Builder(MainActivity.this);
                builder.setPositiveButton("确定", null);
                builder.setTitle("Flutter 调用 Android");
                builder.show();
            }
        });
    }
}
```

注释 1 处用于注册插件，这个是创建 Flutter 工程时 MainActivity 自带的。

注释 2 处创建一个 MethodChannel，它有两个参数，一个是 getFlutterView 方法，用于获取 FlutterView，FlutterView 实现了 BinaryMessenger 接口。另一个是 MethodChannel 的 Name，这个 Name 要保证是唯一的，在 Flutter 端实现中会用到这个 Name。

注释 3 处为 methodChannel 注册一个 MethodCallHandler，用于监听回调的数据。

onMethodCall 方法中的 result 是 Flutter 端传来的数据，我们需要对数据进行判断，然后向 Flutter 端发送数据。

向 Flutter 端发送数据有以下方法。

```
result.success(Object result)
```

如果结果成功，那么将 result 返回给 Flutter 端。

```
result.error(String errorCode,String errorMsg,Object errorDetails)
```

如果结果失败，那么将 errorCode、errorMsg、errorDetails 返回给 Flutter 端。

```
result.notImplemented()
```

Android 端没有实现 Flutter 端需要的方法，它会将 notImplemented 返回给 Flutter 端。

2. Flutter 端实现

在 main.dart 中加入如下代码。

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  static const platformChannel =
      const MethodChannel('com.example.platform_channel/dialog');//1

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Flutter Demo",
      home: Scaffold(
        appBar: AppBar(
          title: Text("Flutter 调用 Android"),
        ),
        body: Padding(
          padding: EdgeInsets.all(40.0),
          child: RaisedButton(
            child: Text("调用 Dialog"),
            onPressed: () {
              showDialog("Flutter 调用 AlertDialog");
            },
          ),
        ),
      ),
    );
  }

  void showDialog(String content) async {
    var arguments = Map();
    arguments['content'] = content;
    try {
      String result = await platformChannel.invokeMethod('dialog', arguments);//2
      print('showDialog ' + result);
    } on PlatformException catch (e) {
```

```
      print('showDialog ' + e.code + e.message + e.details);
    } on MissingPluginException catch (e) {
      print('showDialog ' + e.message);
    }
  }
}
```

注释 1 处创建了 MethodChannel，它需要传入 MethodChannel 的 Name，这个 Name 要保证和 Android 端设置的 Name 是一样的。当单击按钮时会触发 showDialog 方法。注释 2 处用于调用 Android 中的方法，第一个参数是方法的名称，第二个参数 arguments 只能是 Map 或者 JSON 类型的，它是我们需要传递给 Android 端的数据。

运行程序，当我们单击“调用 Dialog”按钮时，效果如图 13-53 所示。

图 13-53　单击“调用 Dialog”按钮时的效果

13.12.3　Android 调用 Flutter

有时 Flutter 调用 Android 后，Android 还会将结果返回给 Flutter，虽然有时可以用 result 来实现，但是 Android 端的处理可能是异步的，result 对象也不能长期持有，这时就需要 Android 来调用 Flutter。

因为页面 UI 是 Flutter 端绘制的，我们很难在页面中控制 Android 端，要实现 Android 调用 Flutter，可以利用 Android 的 Activity 的生命周期，如果将应用切到后台再切回前台，那么 Activity 的 onResume 方法就会被调用，我们在 onResume 方法中实现调用 Flutter 的功能就可以了。

1. Android 端实现

```
package com.example.platform_channel;

import android.os.Bundle;
import android.util.Log;
import java.util.HashMap;
import java.util.Map;
import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
    public static final String MAIN_ACTIVITY = "MainActivity";
    MethodChannel methodChannel;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        GeneratedPluginRegistrant.registerWith(this);

        methodChannel = new
MethodChannel(getFlutterView(),"com.example.platform_channel/text");//1
    }

    @Override
    protected void onResume() {
        super.onResume();
        Map map = new HashMap();
        map.put("content","Android 进阶三部曲");
        methodChannel.invokeMethod("showText", map, new MethodChannel.Result() {//2
            @Override
            public void success(Object o) {
                Log.d(MAIN_ACTIVITY,(String)o);
            }
            @Override
            public void error(String errorCode, String errorMsg, Object errorDetail)
{
```

```
                Log.d(MAIN_ACTIVITY,"errorCode:"+errorCode+" errorMsg:"+errorMsg+"
errorDetail:"+(String)errorDetail);
            }
            @Override
            public void notImplemented() {
                Log.d(MAIN_ACTIVITY,"notImplemented");
            }
        });
    }
}
```

与 Flutter 调用 Android 的代码是类似的，在注释 1 处创建 MethodChannel，然后在注释 2 处调用 Flutter 端的 showText 方法，并将数据以 Map 的形式传递过去。MethodChannel.Result 的回调里有三个方法，通过这三个方法可以得到 Android 调用 Flutter 的结果。

2. Flutter 端实现

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return MyAppState();
  }
}

class MyAppState extends State<MyApp> {
  static const platformChannel =
      const MethodChannel('com.example.platform_channel/text');

  String textContent = 'Flutter 端初始文字';

  @override
  void initState() {
    super.initState();
    platformChannel.setMethodCallHandler((methodCall) async {
      switch (methodCall.method) {
        case 'showText':
          String content = await methodCall.arguments['content'];
```

```
        if (content != null && content.isNotEmpty) {
          setState(() {
            textContent = content;
          });
          return 'success';
        } else {
          throw PlatformException(
              code: 'error', message: '失败', details: 'content is null');
        }
        break;
      default:
        throw MissingPluginException();
    }
  });
}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: "Flutter Demo",
    home: Scaffold(
      appBar: AppBar(
        title: Text('Android 调用 Flutter'),
      ),
      body: Padding(
        padding: EdgeInsets.all(40.0),
        child: Text(textContent),
      ),
    ),
  );
}
}
```

因为要实现 Flutter 页面的改变，所以就需要在 initState 方法中为 MethodChannel 添加回调，如果 Android 端传递过来的方法名称为 showText，就获取 Android 端传来的 content 的值，并赋值给 Text 来改变页面的状态。

运行程序后，将程序切到后台再切回前台，效果如图 13-54 所示。

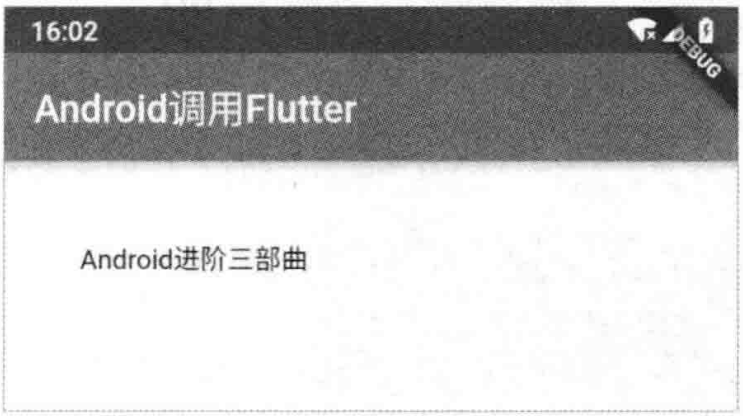


图 13-54　将程序切到后台再切回前台的效果

13.13　本章小结

按理来说 Flutter 不算是 Android 技术，但是作为 Android 应用开发还是需要掌握 Flutter 的。关于 Flutter 的知识点完全可以写出一本书甚至几本书，这里精简地介绍了学习 Flutter 必备的知识点，其中包括环境搭建、Basic Widget、Scrolling Widget、网络请求库 Dio、路由与数据传递等。由于本书不是专门写 Flutter 的书，因此最后并没有介绍 Flutter 的实战，感兴趣的读者可以读一些专门写 Flutter 的书或者博客。