

Android高级进阶

顾浩鑫 / 著

Android高级研发工程师50个必备技能点



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



与本书作者在两家公司都是同事，也有多个Android项目合作的经历，作者基础非常扎实，经常探索并应用Android新技术、新框架，是业界非常优秀的Android架构师。本书是作者在一线互联网企业工作多年的经验沉淀，从不同角度对Android新技术抛砖引玉，将实战中的案例抽象成原型，全书内容通俗易懂，特别推荐给有志在移动端发展的工程师朋友们。

腾讯移动互联网事业部 何朝阳

书中涵盖了Android开发生命周期的各个方面，尤其注重高质量的开发实践。作者从基础、架构、安全、性能优化、新技术、测试等角度，通过简单的代码示例详尽地展示了Android开发技巧。作者对代码的优雅以及对卓越性能的极致追求，足以保证本书成为Android研发工程师不可或缺的参考书。

腾讯移动互联网事业部 揭宗昌

与笔者共事一年多，深刻体会到笔者扎实沉稳的气质，以及探索创新的锐气。长期从事一线软件开发工作使得笔者对“工欲善其事，必先利其器”有着深刻的理解，本书从Android开发涉及的各个方面，各个阶段阐述笔者的经验沉淀，也系统地介绍各类开源工具和开发利器，适合各层次的读者阅读。

华多网络娱乐部 欧阳绍聪

本书作者技术基础深厚、扎实，是新技术的探索和追逐者，与他共事期间，我深刻感受到他对于技术发自内心的热爱，他的Android应用开发技术达到炉火纯青的地步，在APP开发和优化上有丰富的实战经验，同时他还是开源社区的活跃分子和贡献者。作者经历了移动互联网的热潮，见证了APP应用开发技术的萌芽、发展和成熟。现在移动互联网的开发者也逐步开始关注插件化技术、性能优化、行业新技术，系统架构等进阶技术。本书的上市弥补了Android进阶技术系统化解读的空缺，作者对于技术的解读朴实风趣，深入浅出，内容都是在大型互联网公司工作与探索积累下来的精华，是Android工程师的进阶宝典和面试宝典，推荐给广大从事移动开发的朋友们。

阿里移动事业部 顾大辉



博文视点Broadview



@博文视点Broadview



策划编辑：董 英

责任编辑：徐津平

封面设计：李 玲

上架建议：程序设计

ISBN 978-7-121-29845-5



定价：89.00元

Android高级进阶

顾浩鑫 / 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是Android的进阶学习指南，主要为Android初中级开发者进阶所需的知识，高级开发者也可以从本书中发现很多共鸣点。本书从8个方面对50个知识点进行分类讲解，包括基础篇、系统架构篇、经验总结篇、新技术篇、性能优化篇、移动安全篇、工具篇、测试篇。熟练掌握这些知识点后，应该就能够应付实际项目开发中的绝大部分问题了。

本书的主要目的在于给读者一个完整的Android中高级开发者知识图谱。笔者希望通过本书的系统讲解，能够帮助读者在面试和工作中收获自己满意的成绩。

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高级进阶 / 顾浩鑫著. —北京：电子工业出版社，2016.10

ISBN 978-7-121-29845-5

I. ①A… II. ①顾… 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（2016）第208289号

策划编辑：董 英

责任编辑：徐津平

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

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：33.5 字数：643千字

版 次：2016年10月第1版

印 次：2016年11月第2次印刷

印 数：3001～5000册 定价：89.00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zltts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

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

Android 应用开发发展到今天，已经成为一个非常成熟的技术方向，市面上涌现了大量关于 Android 开发的图书，总的来说可以分为以下几类。

- **入门实战类**：这类图书是市面上占比最多的，也是初中级开发者比较青睐的一类图书，基本上介绍的都是Android的基础知识，例如界面开发、数据存储、网络通信、数据库操作、传感器使用等，最后附上一两个简单实战项目的介绍。建议读者在入门的时候买一本评价还不错的这类图书学习一遍，然后结合实际项目进行知识巩固即可，之后这类书基本上扮演的是工具书的角色，就是在忘记某个知识点的时候到书中查一查重新温习一遍。
- **源码分析类**：这类图书又可以分为Framework层源码分析类和Native层源码分析类两种，这类图书主要针对从事Android系统开发的读者。当然，从事应用开发的读者如果对Android Framework等底层的实现原理有所了解的话那也是大有裨益的，而且相对而言更有优势。
- **安全和逆向分析类**：这类图书主要介绍的是Android平台的软件安全、逆向分析及加解密技术等，主要涉及Android软件的静态分析、动态调试、破解及反破解等技术。这个方向的知识既有Android应用开发者所需要的，也有Android系统开发者所需要的，市场上也有专门的Android系统安全相关的职位。
- **系统移植和驱动类**：这类图书主要介绍的是Android内核、移植和驱动开发的整个底层嵌入式开发知识，这一类知识点其实并不能算作移动端开发，本质上属于传统的嵌入式开发领域，毕竟Android底层就是一个Linux系统。

以上便是目前笔者所看到的 Android 图书市场所覆盖的内容，可以说是大而全。但事实上在笔者看来，还有一类图书是目前没有出现的，那就是体现一线互联网公司工程实践中涉及的重要知识点，笔者将之归类为第 5 类：工程实践类。希望本书能够弥补这个空白，并期待能够看到更多此类图书的出现。

本书内容

本书从结构上分为 8 篇，共 50 个知识点。

- **基础篇**：这是占比最大的一个篇章，虽然名为基础篇，但你应该理解为是高级进阶里面的基础知识，而不是Android开发入门的基础知识。本篇主要包含Android View、动画、Support Library、Gradle、NDK、ANR、异步技术、注解、数据序列化和Hybrid等知识点。
- **系统架构篇**：本篇侧重介绍架构和项目整体的相关知识，主要包括UI架构、事件总线、编码规范和技术堆栈等知识点。
- **经验总结篇**：本篇侧重介绍Android工程实践中得出的经验，主要包括64K方法数限制、插件框架、推送原理、APP瘦身、Crash统计等知识点。
- **新技术篇**：本篇侧重介绍新近出现的技术点，当然可能本书出版时已经没那么新了，但并不妨碍其重要性，主要包括函数式编程思想简介、依赖注入、Kotlin、React Native、在线热修复、AOP和Facebook Buck等知识点。
- **性能优化篇**：本篇侧重介绍目前Android平台常见的性能优化相关知识，主要包括代码优化、图片优化、电量优化、布局优化和网络优化等知识点。
- **移动安全篇**：本篇侧重介绍很多应用中经常会忽略的安全知识，主要包括混淆、反编译、密钥隐藏、加固和如何编写安全的代码等知识点。
- **工具篇**：本篇介绍了Android开发中会用到的几个工具，主要包括Android Studio、Stetho、LeakCanary和Redex等知识点。
- **测试篇**：本篇介绍了测试相关的知识，这部分是很多开发人员经常容易忽略的内容，本质上属于测试领域，但开发人员需要有所了解，主要包括Android单元测试、UI自动化测试、静态代码分析和自动化构建等工具。

读者对象

本书的读者对象如下。

- Android应用研发工程师
- 计算机相关专业的学生

致谢

感谢董英编辑找到我并建议我出一本关于 Android 开发相关的图书，不然也不会有这本书的存在；感谢妻子恋恋对我的写作和生活的鼓励和陪伴；感谢父母和老师对我的培养；感谢我曾经就职的公司：华为、百度、平安科技，以及现在所在的平安金融科技；感谢华为 PTN、华为推送、百度文库、百度导航、百度打车、Hyperion、任意门、银行一账通等项目的兄弟姐妹们，是过往的这些公司和团队磨砺了我。

勘误与互动

读者如果发现本书文字、代码和图片等信息存在错误或者纰漏，欢迎反馈给我。对书中内容或者 Android 应用开发中有什么疑问，也可以与我互动，届时将在微信公众号定期发布本书的勘误信息，并解答大家的疑问。我的相关信息如下。

- 微信公众号：ASCE1885
- 微博：http://weibo.com/asce885?is_all=1
- GitHub：<https://github.com/ASCE1885>

目 录

第1篇 基础篇

第1章 Android触摸事件传递机制	2
1.1 触摸事件的类型	2
1.2 事件传递的三个阶段	3
1.3 View的事件传递机制	4
1.4 ViewGroup的事件传递机制	10
第2章 Android View的绘制流程	16
2.1 绘制的整体流程	17
2.2 MeasureSpec	17
2.3 Measure	19
2.4 Layout	22
2.5 Draw	22
第3章 Android 动画机制	25
3.1 逐帧动画 (Frame Animation)	25
3.1.1 XML 资源文件方式	25
3.1.2 代码方式	26
3.2 补间动画 (Tween Animation)	27
3.2.1 插值器 Interpolator	27
3.2.2 AlphaAnimation	29
3.2.3 ScaleAnimation	30
3.2.4 TranslateAnimation	31
3.2.5 RotateAnimation	32
3.2.6 自定义补间动画	34
3.3 属性动画 (Property Animation)	34
3.3.1 Evaluator	35
3.3.2 AnimatorSet	36
3.3.3 ValueAnimator	36
3.3.4 ObjectAnimator	38
3.4 过渡动画 (Transition Animation)	40
第4章 Support Annotation Library 使用详解	46
4.1 Nullness 注解	47

4.2	资源类型注解	48
4.3	类型定义注解	50
4.4	线程注解	52
4.5	RGB 颜色值注解	52
4.6	值范围注解	53
4.7	权限注解	53
4.8	重写函数注解	54
4.9	返回值注解	55
4.10	@VisibleForTesting	55
4.11	@Keep	55
第5章 Percent Support Library使用详解		57
第6章 Design Support Library使用详解		62
6.1	Snackbar	62
6.2	TextInputLayout	63
6.3	TabLayout	64
6.4	NavigationView	65
6.4.1	导航菜单	66
6.4.2	导航头部	67
6.5	FloatingActionButton	70
6.5.1	使用浮动操作按钮	70
6.5.2	其他选项	71
6.5.3	点击事件	71
6.6	CoordinatorLayout	72
6.7	CollapsingToolbarLayout	73
6.8	BottomSheetBehavior	75
第7章 Android Studio中的NDK开发		77
7.1	ABI的基本概念	77
7.2	引入预编译的二进制 C/C++ 函数库	79
7.3	直接从 C/C++ 源码编译	79
7.3.1	配置 ndk.dir 变量	79
7.3.2	在 Gradle 中配置 NDK 模块	79
7.3.3	添加 C/C++ 文件到指定的目录	81
7.4	使用 .so 文件的注意事项	81
7.4.1	使用高平台版本编译的 .so 文件运行在低版本的设备上	81
7.4.2	混合使用不同的C++ 运行时编译的 .so 文件	82
7.4.3	没有为每个支持的 CPU 架构提供对应的 .so 文件	82
7.4.4	将 .so 文件放在错误的地方	82
7.4.5	只提供 armeabi 架构的 .so 文件而忽略其他 ABIs 的	83

第8章 Gradle 必知必会	85
8.1 共享变量的定义	85
8.2 通用配置	87
8.3 aar 函数库的引用	88
8.4 签名和混淆的配置	90
第9章 通过Gradle打包发布函数库到JCenter和Maven Central	92
9.1 Maven Central 和 JCenter	92
9.1.1 Maven Central	93
9.1.2 JCenter	93
9.2 Android Studio 获取函数库的原理	94
9.3 上传函数库到 JCenter	96
9.3.1 步骤一：在 Bintray 网站上注册一个账号	96
9.3.2 步骤二：创建一个 Sonatype 账号	96
9.3.3 步骤三：在 Bintray 网站使能自动签名	97
9.3.4 步骤四：生成 POM 相关的信息	100
9.3.5 步骤五：上传函数库到 Bintray	104
9.3.6 步骤六：发布 Bintray 用户仓库到 JCenter	107
9.3.7 步骤七：同步函数库到 Maven Central	109
第10章 Builder模式详解	110
10.1 经典的 Builder 模式	110
10.2 Builder 模式的变种	113
10.3 变种 Builder 模式的自动化生成	119
10.4 开源函数库的例子	122
第11章 注解在 Android 中的应用	124
11.1 注解的定义	124
11.2 标准注解	125
11.2.1 编译相关注解	125
11.2.2 资源相关注解	125
11.2.3 元注解	125
11.3 运行时注解	127
11.4 编译时注解	127
11.4.1 定义注解处理器	127
11.4.2 注册注解处理器	131
11.4.3 android-apt插件	132
第12章 ANR产生的原因及其定位分析	134
12.1 ANR 产生的原因	135
12.2 典型的 ANR 问题场景	135

12.3	ANR 的定位和分析	136
12.3.1	Logcat 日志信息	136
12.3.2	traces.txt 日志信息	138
12.4	ANR 的避免和检测	141
12.4.1	StrictMode	141
12.4.2	BlockCanary	142
第13章	Android 异步处理技术	144
13.1	Thread	144
13.2	HandlerThread	146
13.3	AsyncQueryHandler	149
13.4	IntentService	150
13.5	Executor Framework	153
13.6	AsyncTask	155
13.7	Loader	156
13.8	总结	159
第14章	Android 数据序列化方案研究	160
14.1	Serializable	160
14.2	Parcelable	166
14.3	SQLiteDatabase	169
14.4	SharedPreferences	170
14.5	JSON	171
14.6	Protocol Buffers 及 Nano-Proto-Buffers	171
14.7	FlatBuffers	171
第15章	Android WebView Java 和 JavaScript 交互详解	173
15.1	Java 调用 JavaScript	173
15.2	JavaScript 调用 Java	174

第2篇 系统架构篇

第16章	MVP 模式及其在 Android 中的实践	180
16.1	MVP 的基本概念	180
16.2	MVP 与 MVC 的区别	181
16.3	MVP 的开源实现	182
16.3.1	Android-Architecture	182
16.3.2	TODO-MVP	182
16.3.3	TODO-MVP-Loaders	183
16.3.4	TODO-MVP-Clean	183
16.3.5	TODO-Databinding	184

16.3.6 其他开源参考实现	184
16.4 MVP 的好处	185
16.5 MVP 存在的问题	185
第17章 MVVM模式及Android DataBinding实战	186
17.1 Data Binding 表达式	187
17.2 数据对象	188
17.3 数据绑定	188
17.4 事件绑定	189
第18章 观察者模式的拓展：事件总线	191
18.1 为何要使用	191
18.2 原理	192
18.3 开源实现	193
18.3.1 EventBus	193
18.3.2 otto	194
18.4 与观察者模式及 Android 广播的区别	196
第19章 书写简洁规范的代码	197
19.1 Java 编码规范	197
19.1.1 源代码文件的定义	197
19.1.2 源代码文件的结构	197
19.1.3 遵循的格式	198
19.1.4 命名约定	200
19.1.5 Javadoc	200
19.2 Android 命名规范	200
19.2.1 布局文件的命名	200
19.2.2 资源文件的命名	201
19.2.3 类的命名	201
19.3 CheckStyle 的使用	202
第20章 基于开源项目搭建属于自己的技术堆栈	203
20.1 APP 的整体架构	203
20.2 技术选型的考量点	205
20.3 日志记录能力	205
20.4 JSON 解析能力	207
20.4.1 gson	207
20.4.2 jackson	207
20.4.3 Fastjson	208
20.4.4 LoganSquare	208

20.5	数据库操作能力	210
20.5.1	ActiveAndroid	210
20.5.2	ormlite	211
20.5.3	greenDAO	211
20.5.4	Realm	212
20.6	网络通信能力	213
20.6.1	android-async-http	213
20.6.2	OkHttp	215
20.6.3	Volley	216
20.6.4	Retrofit	217
20.7	图片缓存和显示能力	217
20.7.1	BitmapFun	218
20.7.2	Picasso	218
20.7.3	Glide	218
20.7.4	Fresco	219
20.7.5	Android-Universal-Image-Loader	219

第3篇 经验总结篇

第21章	64K方法数限制原理与解决方案	222
21.1	64K 限制的原因	222
21.2	使用 MultiDex 解决 64K 限制的问题	223
21.2.1	Android 5.0 之前的版本	223
21.2.2	Android 5.0 及之后的版本	223
21.3	如何避免出现 64K 限制	223
21.4	配置 MultiDex	224
21.5	MultiDex Support Library 的局限性	226
21.6	在开发阶段优化 MultiDex 的构建	227
第22章	Android 插件框架机制研究与实践	230
22.1	基本概念	231
22.1.1	宿主和插件	231
22.1.2	ClassLoader 机制	231
22.2	开源框架	231
22.2.1	android-pluginmgr	232
22.2.2	dynamic-load-apk	232
22.2.3	DynamicAPK	232
22.2.4	DroidPlugin	233
22.2.5	Small	234

第23章 推送机制实现原理详解	235
23.1 推送的开源实现方案	236
23.1.1 基于 XMPP 协议	236
23.1.2 基于 MQTT 协议	236
23.2 推送的第三方平台	236
23.3 自己实现推送功能	237
23.3.1 长连接的建立 (TCPConnectThread)	237
23.3.2 数据的发送 (TCPSendThread)	237
23.3.3 数据的接收 (TCPReceiveThread)	238
23.3.4 心跳包的实现 (TCPHeartBeatThread)	240
第24章 APP 瘦身经验总结	241
24.1 APP 为什么变胖了	241
24.2 从 APK 文件的结构说起	242
24.3 优化图片资源占用的空间	245
24.3.1 无损压缩 [ImageOptim]	246
24.3.2 有损压缩 [ImageAlpha]	246
24.3.3 有损压缩 [TinyPNG]	246
24.3.4 PNG/JPEG 转换为 WebP	246
24.3.5 尽量使用 NinePatch 格式的 PNG 图	247
24.4 使用 Lint 删除无用资源	248
24.5 利用 Android Gradle 配置	248
24.5.1 minifyEnable	248
24.5.2 shrinkResources	249
24.5.3 resConfigs	249
24.5.4 ndk.abiFilters	250
24.6 重构和优化代码	250
24.7 资源混淆	251
24.8 插件化	251
第25章 Android Crash 日志收集原理与实践	252
25.1 Java 层 Crash 捕获机制	253
25.1.1 基本原理	253
25.1.2 线程信息	254
25.1.3 SharedPreferences 信息	255
25.1.4 系统设置	257
25.1.5 Logcat 中的日志记录	261
25.1.6 自定义 Log 文件中的内容	264
25.1.7 MemInfo 信息	266

25.2	Native 层 Crash 捕获机制	267
25.3	Crash 的上报	269

第4篇 新技术篇

第26章	函数式编程思想及其在Android中的应用	272
26.1	代码的简化	274
26.2	Operators 简介	275
第27章	依赖注入及其在Android中的应用	277
27.1	基本概念	277
27.1.1	构造函数注入	278
27.1.2	Setter 函数注入	279
27.1.3	接口注入	279
27.2	为何需要框架	280
27.3	开源框架的选择	280
27.3.1	ButterKnife	280
27.3.2	RoboGuice	282
27.3.3	Dagger	285
27.3.4	Dagger2	288
27.3.5	框架的对比	289
第28章	Android世界的Swift: Kotlin在Android中的应用	290
28.1	选择 Kotlin 的原因	290
28.2	Kotlin 的安装和配置	291
28.3	Kotlin 语言的特性	292
28.3.1	可表达性	292
28.3.2	空类型安全	294
28.3.3	扩展函数	295
28.4	Kotlin 的 Gradle 配置	296
28.5	将 Java 类转换成 Kotlin 类	299
28.6	相关资料	302
第29章	React Native For Android入门指南	304
29.1	环境配置	304
29.1.1	Homebrew	304
29.1.2	nvm	305
29.1.3	Node.js	305
29.1.4	watchman	306
29.1.5	flow	306

29.2	Android 开发环境的要求	306
29.3	React Native 工程配置	307
29.3.1	安装react-native	307
29.3.2	生成工程	307
29.4	Android Studio 工程概览	308
29.5	React Native 依赖库修改为本地	314
29.5.1	下载 react-native.aar	314
29.5.2	react-native.aar 的文件内容	315
29.5.3	Gradle 本地依赖	316
29.5.4	将 node_modules 上传到 svn/git	318
29.6	React Native 学习建议	319
第30章	Android在线热修复方案研究	320
30.1	在线热修复的基本流程	320
30.2	Dexposed	321
30.2.1	如何集成	322
30.2.2	基本用法	323
30.2.3	在线热修复	325
30.2.4	平台的限制	328
30.3	AndFix	329
30.3.1	如何集成	329
30.3.2	补丁包生成工具	331
30.3.3	平台的限制	332
30.4	Nuwa	332
30.4.1	基本原理	332
30.4.2	如何集成	333
30.4.3	补丁生成工具	334
30.4.4	平台的限制	334
30.5	总结	334
第31章	面向切面编程及其在Android中的应用	335
31.1	AOP 的基本概念	335
31.2	代码织入的时机	336
31.3	基于 AspectJ 实现 Android 平台的 AOP	337
31.3.1	Hugo 的用法简介	337
31.3.2	Hugo 的实现原理	339
31.4	其他 AOP 开源框架	344
第32章	基于Facebook Buck改造Android构建系统	345
32.1	Buck环境配置	346
32.1.1	Homebrew 方式	346

32.1.2	手动构建方式	346
32.1.3	安装 Watchman	348
32.1.4	安装 Android SDK 和 Android NDK	348
32.2	快速创建基于 Buck 构建的 Android 工程	349
32.3	Buck 的基本概念	351
32.3.1	构建规则 (Build Rule)	352
32.3.2	构建目标 (Build Target)	354
32.3.3	构建文件 (Build File)	355
32.3.4	构建目标模式 (Build Target Pattern)	356
32.4	项目改造实战	357
32.4.1	步骤一: 手动下载工程依赖的第三方 Jar包或者aar包	357
32.4.2	步骤二: 将 R.* 常量修改为非 final 的	357
32.4.3	步骤三: 创建 BUCK 文件	358
32.4.4	步骤四: 编译 Buck 的 buck-android-support	363
32.4.5	步骤五: Exopackage 的使用	363
32.5	Buck 的自动化改造	366

第5篇 性能优化篇

第33章	代码优化	368
33.1	数据结构的选择	368
33.2	Handler 和内部类的正确用法	370
33.3	正确地使用 Context	373
33.3.1	Context 的种类	374
33.3.2	错误使用 Context 导致的内存泄漏	374
33.3.3	不同 Context 的对比	376
33.4	掌握 Java 的四种引用方式	376
33.5	其他代码微优化	377
33.5.1	避免创建非必要的对象	377
33.5.2	对常量使用 static final 修饰	378
33.5.3	避免内部的 Getters/Setters	378
33.5.4	代码的重构	378
第34章	图片优化	379
34.1	图片的格式	379
34.1.1	JPEG	380
34.1.2	PNG	380
34.1.3	GIF	380
34.1.4	WebP	380
34.2	图片的压缩	380
34.2.1	无损压缩 ImageOptim	381

34.2.2	有损压缩 ImageAlpha	381
34.2.3	有损压缩 TinyPNG	381
34.2.4	PNG/JPEG 转换为 WebP	381
34.2.5	尽量使用 NinePatch 格式的 PNG 图	382
34.3	图片的缓存	382
第35章	电量优化	383
35.1	BroadcastReceiver	383
35.2	数据传输	384
35.3	位置服务	384
35.4	AlarmManager	386
35.5	WakeLock	386
第36章	布局优化	388
36.1	include 标签共享布局	388
36.2	ViewStub 标签实现延迟加载	389
36.3	merge 标签减少布局层次	391
36.4	尽量使用 CompoundDrawable	392
36.5	使用 Lint	393
第37章	网络优化	395
37.1	避免 DNS 解析	395
37.2	合并网络请求	395
37.3	预先获取数据	396
37.4	避免轮询	396
37.5	优化重连机制	396
37.6	离线缓存	396
37.7	压缩数据大小	396
37.8	不同的网络环境使用不同的超时策略	397
37.9	CDN 的使用	397

第6篇 移动安全篇

第38章	Android混淆机制详解	400
38.1	Java 代码的混淆	400
38.1.1	Proguard 的特性	401
38.1.2	Proguard 的使能和配置	401
38.1.3	proguard-rules.pro 文件的编写	404
38.1.4	Proguard 生成的文件	407
38.1.5	Proguard 混淆规则汇总	409
38.2	Native (C/C++) 代码的混淆	409

38.3 资源文件的混淆	409
第39章 Android 反编译机制详解	411
39.1 资源文件的反编译	411
39.1.1 ApkTool 的安装	411
39.1.2 ApkTool 的使用	412
39.2 Java 代码的反编译	413
第40章 客户端敏感信息隐藏技术研究	414
40.1 敏感信息嵌套在 strings.xml 中	415
40.2 敏感信息隐藏在 Java 源代码中	415
40.3 敏感信息隐藏在 BuildConfig 中	417
40.4 使用 DexGuard	418
40.5 对敏感信息进行伪装或者加密	419
40.6 敏感信息隐藏在原生函数库中 (.so 文件)	419
40.7 对APK进行加固处理	419
第41章 Android 加固技术研究	421
41.1 爱加密的主要功能	421
41.1.1 漏洞分析	421
41.1.2 加密服务	422
41.1.3 渠道监测	423
41.2 常见 APP 漏洞及风险	423
41.2.1 静态破解	423
41.2.2 二次打包	424
41.2.3 本地储存数据窃取	424
41.2.4 界面截取	424
41.2.5 输入法攻击	424
41.2.6 协议抓取	424
41.3 Android 程序反破解技术	424
41.3.1 对抗反编译	424
41.3.2 对抗静态分析	425
41.3.3 对抗动态调试	425
41.3.4 防止重编译	425
41.4 加固技术研究知识储备	426
41.4.1 掌握常见的破解分析工具	426
41.4.2 掌握 Dalvik 指令集代码	428
41.4.3 掌握 Dex 和 Odex 文件格式	428
41.4.4 掌握 Smali 文件格式	428
41.4.5 掌握基于 Android 的 ARM 汇编语言基础	428

第42章 Android安全编码	429
42.1 WebView 远程代码执行	429
42.2 WebView 密码明文保存	430
42.3 Android 本地拒绝服务	431
42.3.1 非法序列化对象导致的 ClassNotFoundException	431
42.3.2 空 Action 导致的 NullPointerException	432
42.3.3 强制类型转换导致的 ClassCastException	433
42.3.4 数组越界导致的 IndexOutOfBoundsException	433
42.4 SharedPreferences 全局任意读写	434
42.5 密钥硬编码	434
42.6 AES/DES/RSA 弱加密	434
42.7 随机函数使用错误	437
42.8 WebView 忽略 SSL 证书	438
42.9 HTTPS 证书弱校验	438
42.9.1 自定义 X509TrustManager 未实现安全校验	438
42.9.2 自定义 HostnameVerifier 默认接受所有域名	441
42.9.3 SSLSocketFactory 信任所有证书	442
42.10 PendingIntent 使用不当	443

第7篇 工具篇

第43章 Android调试工具Facebook Stetho	446
43.1 视图布局监视	447
43.2 数据库监视	447
43.3 网络监视	448
43.3.1 网络模块使用的是 HttpURLConnection	449
43.3.2 网络模块使用的是 OkHttp	452
43.4 dumpapp	454
43.4.1 插件的编写	454
43.4.2 插件的集成	456
43.4.3 插件的使用	456
43.5 Javascript 控制台	457
43.6 最佳实践	457
第44章 内存泄漏检测函数库 LeakCanary	460
44.1 基本概念	461
44.2 LeakCanary 的集成	461
44.3 LeakCanary 的原理	465
44.4 LeakCanary 的定制	469
44.4.1 RefWatcher 的自定义	469

44.4.2	通知页面样式的自定义	470
44.4.3	内存泄漏堆栈信息保存个数的自定义	471
44.4.4	Watcher 的延时	471
44.4.5	自定义内存泄漏堆栈信息和 heap dump 的处理方式	471
44.4.6	忽略特定的弱引用	472
44.4.7	不监视特定的 Activity 类	472
第45章	基于Facebook Redex实现Android APK的压缩和优化	474
45.1	转换的时机	474
45.2	管道的思想	475
45.3	减少字节码的意义	475
45.4	混淆和压缩	475
45.5	使用内联函数	476
45.6	无用代码的消除	477
45.7	Redex 的集成和使用	478
45.7.1	依赖的安装	478
45.7.2	下载, 构建和安装	478
45.7.3	使用	478
第46章	Android Studio你所需要知道的功能	479
46.1	Annotate	479
46.2	.ignore 插件	480
46.3	Live Templates	481
46.4	集成 Bug 管理系统	482
第8篇 测试篇		
第47章	Android单元测试框架简介	486
47.1	Java 单元测试框架 JUnit	486
47.2	Android 单元测试框架 Robolectric 3.0	488
47.3	Java 模拟测试框架 Mockito	490
47.3.1	行为的验证	490
47.3.2	Stub (桩函数) 的使用	491
第48章	Android UI自动化测试框架简介	492
48.1	Monkey	492
48.2	MonkeyRunner	493
48.3	UIAutomator	493
48.4	Robotium	494
48.5	Espresso	494
48.6	Appium	494

第49章	Android静态代码分析实战	495
49.1	Java代码规范检查工具CheckStyle	495
49.1.1	Gradle方式	495
49.1.2	Android Studio插件方式	497
49.2	Java静态代码分析工具FindBugs	498
49.2.1	Gradle方式	498
49.2.2	Android Studio插件方式	499
49.3	Java静态代码分析工具PMD	500
49.3.1	Gradle方式	500
49.3.2	Android Studio插件方式	501
49.4	Android代码优化工具Lint	501
49.4.1	Gradle方式	501
49.4.2	Android Studio插件方式	502
第50章	基于Jenkins+Gradle搭建Android持续集成编译环境	503
50.1	Tomcat的下载和启动	503
50.2	Jenkins的下载和运行	505
50.3	Jenkins插件的安装	506
50.4	Jenkins全局配置	507
50.4.1	配置 JDK 环境	507
50.4.2	配置 Android SDK 环境	507
50.4.3	配置 Git 环境	508
50.4.4	配置 SVN 环境	508
50.4.5	配置 Gradle 环境	508
50.5	JOB相关的操作	508
50.5.1	JOB 的创建	508
50.5.2	JOB 的配置	509
50.5.3	Gradle 的配置	510
50.5.4	构建触发器的配置	511
50.5.5	参数化构建	514
50.6	Jenkins预定义的环境变量	514



第1篇 基础篇

- ★ 第 1 章 Android 触摸事件传递机制
- ★ 第 2 章 Android View 的绘制流程
- ★ 第 3 章 Android 动画机制
- ★ 第 4 章 Support Annotation Library 使用详解
- ★ 第 5 章 Percent Support Library 使用详解
- ★ 第 6 章 Design Support Library 使用详解
- ★ 第 7 章 Android Studio 中的 NDK 开发
- ★ 第 8 章 Gradle 必知必会
- ★ 第 9 章 通过 Gradle 打包发布函数库到 JCenter 和 Maven Central
- ★ 第 10 章 Builder 模式详解
- ★ 第 11 章 注解在 Android 中的应用
- ★ 第 12 章 ANR 产生的原因及其定位分析
- ★ 第 13 章 Android 异步处理技术
- ★ 第 14 章 Android 数据序列化方案研究
- ★ 第 15 章 Android WebView Java 和 JavaScript 交互详解

第1章

Android触摸事件传递机制

我们在 Android 开发中经常会遇到多个 View、ViewGroup 嵌套的问题，例如 ViewPager 中嵌套 Fragment，而在 Fragment 中需要实现一个横向滚动的广告栏，这时候，就会遇到广告栏的滑动事件和 ViewPager 的滑动事件相互冲突的问题，想要正确快速地处理这种问题，开发者需要对 View 的事件传递机制有较深入的理解。

本章将会详细介绍 Activity、View、ViewGroup 三者的触摸事件传递机制。一次完整的事件传递主要包括三个阶段，分别是事件的分发、拦截和消费。我们首先来了解下 Android 中触摸事件的主要类型。

1.1 触摸事件的类型

触摸事件对应的是 MotionEvent 类，事件的类型主要有如下三种。

- ACTION_DOWN：用户手指的按下操作，一个按下操作标志着一次触摸事件的开始。
- ACTION_MOVE：用户手指按压屏幕后，在松开之前，如果移动的距离超过一定的阈值，那么会被判定为 ACTION_MOVE 操作，一般情况下，手指的轻微移动都会触发一系列的移动事件。
- ACTION_UP：用户手指离开屏幕的操作，一次抬起操作标志着一次触摸事件的结束。

在一次屏幕触摸操作中，ACTION_DOWN 和 ACTION_UP 这两个事件是必需的，而 ACTION_MOVE 视情况而定，如果用户仅仅是点击了一下屏幕，那么可能只会监测到按下和抬起的动作。

1.2 事件传递的三个阶段

在了解了触摸事件的三种主要类型之后，在讲解 Activity、View、ViewGroup 的事件传递的具体实现之前，我们需要来了解本章开头说的事件传递的三个阶段。

- 分发（Dispatch）：事件的分发对应着 dispatchTouchEvent 方法，在 Android 系统中，所有的触摸事件都是通过这个方法来的，方法原型如下。

```
public boolean dispatchTouchEvent(MotionEvent ev)
```

在这个方法中，根据当前视图的具体实现逻辑，来决定是直接消费这个事件还是将事件继续分发给子视图处理，方法返回值为 true 表示事件被当前视图消费掉，不再继续分发事件；方法返回值为 super.dispatchTouchEvent 表示继续分发该事件。如果当前视图是 ViewGroup 及其子类，则会调用 onInterceptTouchEvent 方法判定是否拦截该事件。

- 拦截（Intercept）：事件的拦截对应着 onInterceptTouchEvent 方法，这个方法只在 ViewGroup 及其子类中才存在，在 View 和 Activity 中是不存在的。方法的原型如下。

```
public boolean onInterceptTouchEvent(MotionEvent ev)
```

同理，这个方法也是通过返回的布尔值来决定是否拦截对应的事件，根据具体的实现逻辑，返回 true 表示拦截这个事件，不继续分发给子视图，同时交由自身的 onTouchEvent 方法进行消费；返回 false 或者 super.onInterceptTouchEvent 表示不对事件进行拦截，需要继续传递给子视图。

- 消费（Consume）：事件的消费对应着 onTouchEvent 方法，方法原型如下。

```
public boolean onTouchEvent(MotionEvent event)
```

该方法返回值为 true 表示当前视图可以处理对应的事件，事件将不会向上传递给父视图；返回值为 false 表示当前视图不处理这个事件，事件会被传递给父视图的 onTouchEvent 方法进行处理。

在 Android 系统中，拥有事件传递处理能力的类有以下三种。

- Activity：拥有 dispatchTouchEvent 和 onTouchEvent 两个方法。
- ViewGroup：拥有 dispatchTouchEvent、onInterceptTouchEvent 和 onTouchEvent 三个方法。
- View：拥有 dispatchTouchEvent 和 onTouchEvent 两个方法。

本章接下来将分别介绍 View 和 ViewGroup 的事件传递机制，Activity 相关的内容穿插其中，将不再单独介绍。

1.3 View的事件传递机制

虽然 ViewGroup 是 View 的子类，但是这里所说的 View 专指除 ViewGroup 外的 View 控件，例如 TextView、Button、CheckBox 等，View 控件本身已经是最小的单位，不能再作为其他 View 的容器。View 控件拥有 dispatchTouchEvent 和 onTouchEvent 两个方法。为了清楚地演示，我们首先来定义一个继承 TextView 的类 MyTextView，如下所示。我们将每个事件的触发都打印了日志，以方便了解事件传递的流程。

```
public class MyTextView extends TextView {

    private static final String TAG = "MyTextView";

    public MyTextView(Context context) {
        super(context);
    }

    public MyTextView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        switch (ev.getAction()) {
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "dispatchTouchEvent ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "dispatchTouchEvent ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "dispatchTouchEvent ACTION_UP");
                break;
            case MotionEvent.ACTION_CANCEL:
```



```

        Log.e(TAG, "dispatchTouchEvent ACTION_CANCEL" );
        break;
    default:
        break;
}
return super.dispatchTouchEvent(ev);
}

```

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onTouchEvent ACTION_DOWN" );
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onTouchEvent ACTION_MOVE" );
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "onTouchEvent ACTION_UP" );
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.e(TAG, "onTouchEvent ACTION_CANCEL" );
            break;
        default:
            break;
    }
    return super.onTouchEvent(event);
}
}

```

同时定义一个 MainActivity 用来展示 MyTextView，在这个 Activity 中，我们为 MyTextView 设置了点击（onClick）和触摸（onTouch）监听，方便跟踪了解事件传递的流程，代码如下。

```
public class MainActivity extends AppCompatActivity implements View.  
OnClickListener, View.OnTouchListener {
```

```
    private static final String TAG = "MainActivity";
```

```
    private MyTextView mTextView;
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        mTextView = (MyTextView) findViewById(R.id.my_text_view);  
        mTextView.setOnClickListener(this); // 设置MyTextView的点击处理  
        mTextView.setOnTouchListener(this); // 设置MyTextView的触摸处理  
    }
```

```
    @Override
```

```
    public boolean dispatchTouchEvent(MotionEvent ev) {  
        switch (ev.getAction()) {  
            case MotionEvent.ACTION_DOWN:  
                Log.e(TAG, "dispatchTouchEvent ACTION_DOWN");  
                break;  
            case MotionEvent.ACTION_MOVE:  
                Log.e(TAG, "dispatchTouchEvent ACTION_MOVE");  
                break;  
            case MotionEvent.ACTION_UP:  
                Log.e(TAG, "dispatchTouchEvent ACTION_UP");  
                break;  
            case MotionEvent.ACTION_CANCEL:  
                Log.e(TAG, "dispatchTouchEvent ACTION_CANCEL");  
                break;  
            default:  
                break;  
        }  
        return super.dispatchTouchEvent(ev);  
    }
```

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onTouchEvent ACTION_DOWN" );
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onTouchEvent ACTION_MOVE" );
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "onTouchEvent ACTION_UP" );
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.e(TAG, "onTouchEvent ACTION_CANCEL" );
            break;
        default:
            break;
    }
    return super.onTouchEvent(event);
}
```

```
@Override
public void onClick(View view) {
    switch (view.getId()) {
        case R.id.my_text_view:
            Log.e(TAG, "MyTextView onClick" );
            break;
        default:
            break;
    }
}
```

```
@Override
public boolean onTouch(View view, MotionEvent motionEvent) {
    switch(view.getId()) {
```

```

        case R.id.my_text_view:
            switch (motionEvent.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    Log.e(TAG, "MyTextView onTouch ACTION_DOWN" );
                    break;
                case MotionEvent.ACTION_MOVE:
                    Log.e(TAG, "MyTextView onTouch ACTION_MOVE" );
                    break;
                case MotionEvent.ACTION_UP:
                    Log.e(TAG, "MyTextView onTouch ACTION_UP" );
                    break;
                default:
                    break;
            }
            break;
        default:
            break;
    }
    return false;
}
}

```

运行上面的代码，点击 MyTextView，在 Logcat 中将打印出如下日志。

```

com.ascel885.viewdemo E/MainActivity: dispatchTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MyTextView: dispatchTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MainActivity: MyTextView onTouch ACTION_DOWN
com.ascel885.viewdemo E/MyTextView: onTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MainActivity: dispatchTouchEvent ACTION_UP
com.ascel885.viewdemo E/MyTextView: dispatchTouchEvent ACTION_UP
com.ascel885.viewdemo E/MainActivity: MyTextView onTouch ACTION_UP
com.ascel885.viewdemo E/MyTextView: onTouchEvent ACTION_UP
com.ascel885.viewdemo E/MainActivity: MyTextView onClick

```

从上面的代码和运行日志可以看出，dispatchTouchEvent、onTouchEvent 这两个方法的返回值可能存在以下三种情况。

- 直接返回 false。
- 直接返回 true。
- 返回父类的同名方法，例如 `super.dispatchTouchEvent`。

不同的返回值会导致事件传递流程相差甚远，通过不断修改这些方法的返回值并查看日志记录，我们最终可以得到屏幕按下操作 ACTION_DOWN 事件的处理流程如图 1-1 所示，ACTION_UP 的流程与图 1-1 类似，只不过事件类型不同而已，我们将不再一一列举。

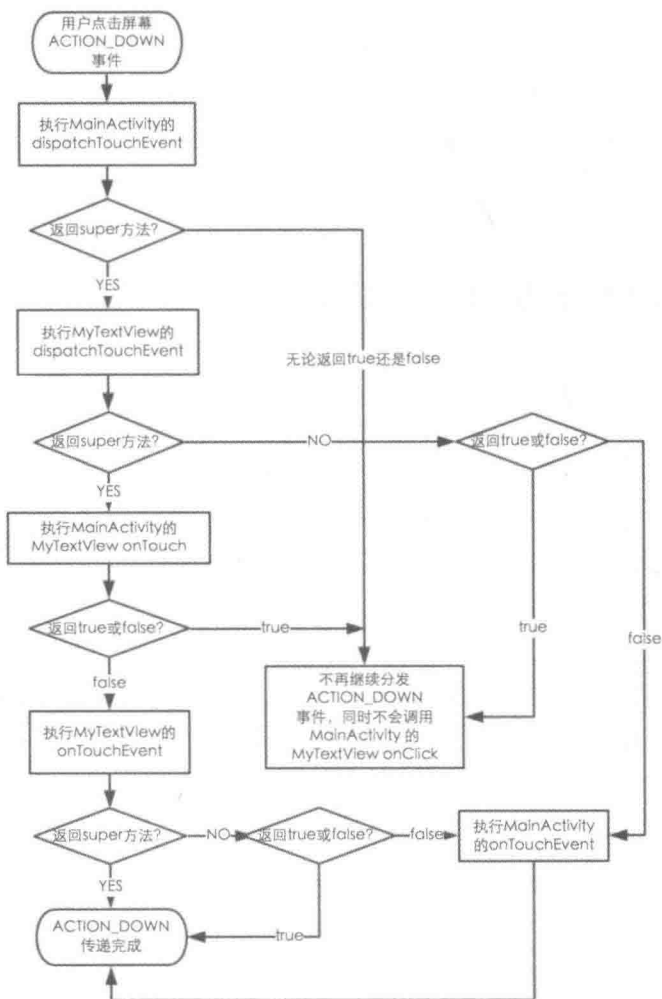


图 1-1

从上面的流程图可以得出以下结论。

- 触摸事件的传递流程是从 `dispatchTouchEvent` 开始的，如果不进行人为干预（也就是默认返回父类的同名函数），则事件将会依照嵌套层次从外层向内层传递，到达最内层的 `View` 时，就由它的 `onTouchEvent` 方法处理，该方法如果能够消费该事件，则返回 `true`，如果处理不了，则返回 `false`，这时事件会重新向外层传递，并由外层 `View` 的 `onTouchEvent` 方法进行处理，依此类推。
- 如果事件在向内层传递过程中由于人为干预，事件处理函数返回 `true`，则会导致事件提前被消费掉，内层 `View` 将不会收到这个事件。
- `View` 控件的事件触发顺序是先执行 `onTouch` 方法，在最后才执行 `onClick` 方法。如果 `onTouch` 返回 `true`，则事件不会继续传递，最后也不会调用 `onClick` 方法；如果 `onTouch` 返回 `false`，则事件继续传递。

1.4 ViewGroup的事件传递机制

`ViewGroup` 是作为 `View` 控件的容器存在的，Android 系统默认提供了一系列 `ViewGroup` 的子类，常见的有 `LinearLayout`、`RelativeLayout`、`FrameLayout`、`ListView`、`ScrollView` 等。`ViewGroup` 拥有 `dispatchTouchEvent`、`onInterceptTouchEvent` 和 `onTouchEvent` 三个方法，可以看出和 `View` 的唯一区别是多了一个 `onInterceptTouchEvent` 方法。为了演示，我们需要自定义如下一个 `ViewGroup`，继承自 `RelativeLayout`，实现一个 `MyRelativeLayout`。

```
public class MyRelativeLayout extends RelativeLayout {  
  
    private static final String TAG = "MyRelativeLayout";  
  
    public MyRelativeLayout(Context context) {  
        super(context);  
    }  
  
    public MyRelativeLayout(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
}
```

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    switch (ev.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "dispatchTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "dispatchTouchEvent ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "dispatchTouchEvent ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.e(TAG, "dispatchTouchEvent ACTION_CANCEL");
            break;
        default:
            break;
    }
    return super.dispatchTouchEvent(ev);
}

@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    switch (ev.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onInterceptTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onInterceptTouchEvent ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "onInterceptTouchEvent ACTION_UP");
            break;
    }
}
```

```

        default:
            break;
    }
    return super.onInterceptTouchEvent(ev);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Log.e(TAG, "onTouchEvent ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.e(TAG, "onTouchEvent ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.e(TAG, "onTouchEvent ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.e(TAG, "onTouchEvent ACTION_CANCEL");
            break;
        default:
            break;
    }
    return super.onTouchEvent(event);
}
}

```

同时将这个 Layout 作为 MyTextView 的容器，也就是修改 XML 布局文件如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<com.ascel885.viewdemo.MyRelativeLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"

```



```

    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.ascel885.viewdemo.MainActivity" >

    <com.ascel885.viewdemo.MyTextView
        android:id="@+id/my_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:text="Hello World!" />
</com.ascel885.viewdemo.MyRelativeLayout>

```

运行, 点击 MyTextView, 在 Logcat 中打印日志如下。

```

com.ascel885.viewdemo E/MainActivity: dispatchTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MyRelativeLayout: dispatchTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MyRelativeLayout: onInterceptTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MyTextView: dispatchTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MainActivity: MyTextView onTouch ACTION_DOWN
com.ascel885.viewdemo E/MyTextView: onTouchEvent ACTION_DOWN
com.ascel885.viewdemo E/MainActivity: dispatchTouchEvent ACTION_UP
com.ascel885.viewdemo E/MyRelativeLayout: dispatchTouchEvent ACTION_UP
com.ascel885.viewdemo E/MyRelativeLayout: onInterceptTouchEvent ACTION_UP
com.ascel885.viewdemo E/MyTextView: dispatchTouchEvent ACTION_UP
com.ascel885.viewdemo E/MainActivity: MyTextView onTouch ACTION_UP
com.ascel885.viewdemo E/MyTextView: onTouchEvent ACTION_UP
com.ascel885.viewdemo E/MainActivity: MyTextView onClick

```

可以看到, 与 View 的事件流程唯一不一样的地方是在 MainActivity 和 MyTextView 之间增加了一层 MyRelativeLayout。同理, 通过打印日志信息, 可以得到如图 1-2 所示的流程图。

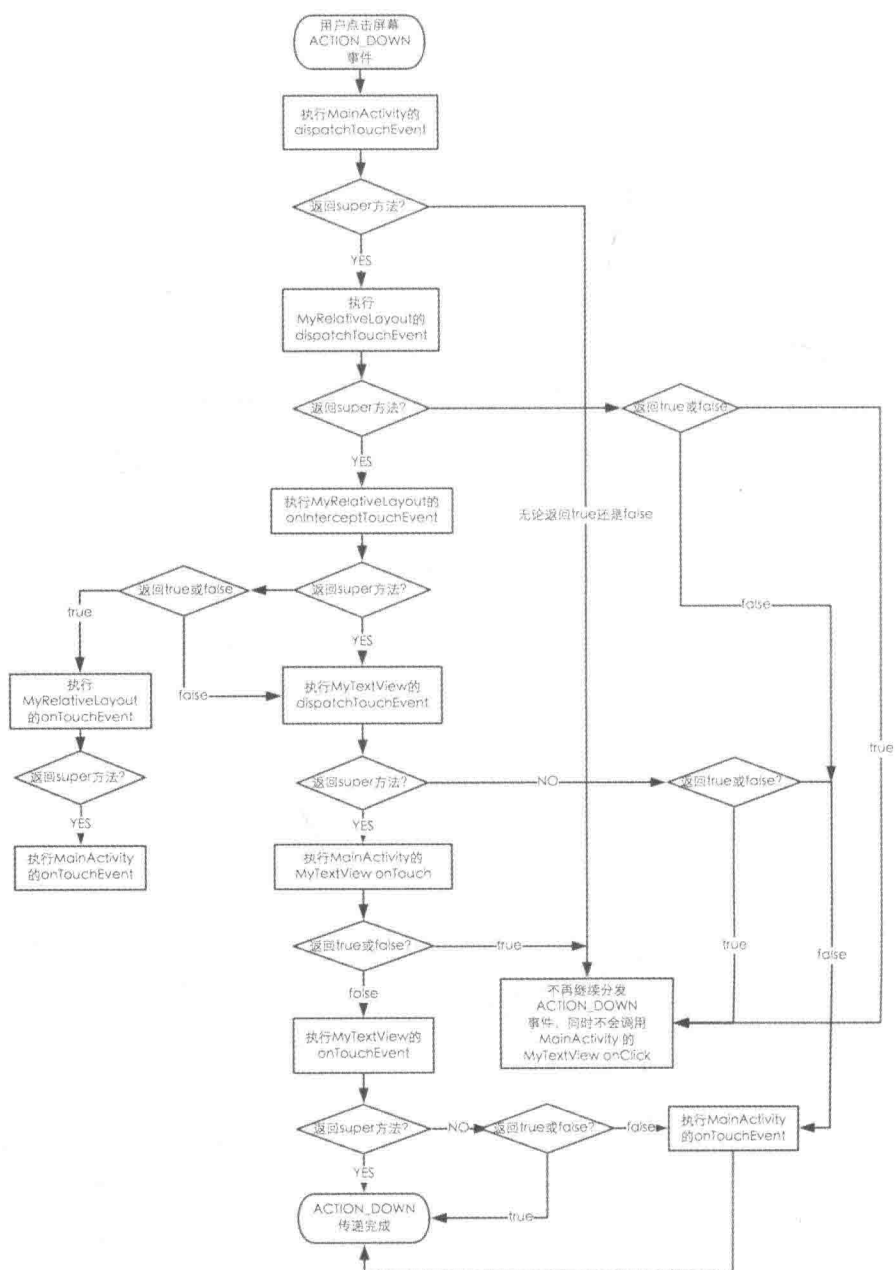


图1-2

从上面的流程图我们可以得出以下结论。

- 触摸事件的传递顺序是由 Activity 到 ViewGroup，再由 ViewGroup 递归传递给它的子 View。
- ViewGroup 通过 `onInterceptTouchEvent` 方法对事件进行拦截，如果该方法返回 `true`，则事件不会继续传递给子 View，如果返回 `false` 或者 `super.onInterceptTouchEvent`，则事件会继续传递给子 View。
- 在子 View 中对事件进行消费后，ViewGroup 将接收不到任何事件。

在掌握了上面的事件传递流程之后，应该可以应对实际项目开发中 90% 以上的工作，如果想更进一步地了解，可以阅读 Framework 层的源码，基本上就是遵循上述流程，这个留给读者当作练习。

第2章

Android View的绘制流程

在项目开发过程中，经常存在需要实现自定义控件的情况，对于比较简单的需求，通过组合系统提供的原生控件即可完成，但碰到设计师脑洞大开的时候，通过简单的组合方式显然满足不了需求，这时候往往需要技术人员自己实现控件的测量、布局和绘制等操作，而这一切的前提是你需要熟练掌握 View 的绘制流程。

Android 中 Activity 是作为应用程序的载体存在的，它代表着一个完整的用户界面，提供了一个窗口来绘制各种视图，当 Activity 启动时，我们会通过 setContentView 方法来设置一个内容视图，这个内容视图就是用户看到的界面。在第 1 章中我们已经知道了 Android 中 View 存在两种形式：一种是单一的 View 控件，另一种是可以包含其他 View 的 ViewGroup 容器。前面所说的内容视图就是以 ViewGroup 的形式存在的。在正式讲解 View 的绘制流程之前，我们有必要先来简单了解一下 Android 的 UI 管理系统的层级关系，如图 2-1 所示。

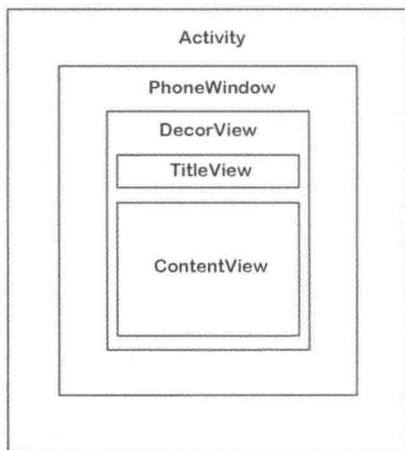


图2-1

PhoneWindow 是 Android 系统中最基本的窗口系统，每个 Activity 会创建一个。PhoneWindow 是 Activity 和 View 系统交互的接口。DecorView 本质上是一个 FrameLayout，是 Activity 中所有 View 的祖先。

2.1 绘制的整体流程

当一个应用启动时，会启动一个主 Activity，Android 系统会根据 Activity 的布局来对它进行绘制。绘制会从根视图 ViewRoot 的 performTraversals() 方法开始，从上到下遍历整个视图树，每个 View 控件负责绘制自己，而 ViewGroup 还需要负责通知自己的子 View 进行绘制操作。视图绘制的过程可以分为三个步骤，分别是测量（Measure）、布局（Layout）和绘制（Draw）。performTraversals() 的核心代码如下。

```
private void performTraversals() {  
    ...  
    int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);  
    int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);  
    ...  
    // 执行测量流程  
    performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);  
    ...  
    // 执行布局流程  
    performLayout(lp, desiredWindowWidth, desiredWindowHeight);  
    ...  
    // 执行绘制流程  
    performDraw();  
}
```

2.2 MeasureSpec

MeasureSpec 表示的是一个 32 位的整型值，它的高 2 位表示测量模式 SpecMode，低 30 位表示某种测量模式下的规格大小 SpecSize。MeasureSpec 是 View 类的一个静态内部类，用来说明应该如何测量这个 View，其核心代码如下。

```
public static class MeasureSpec {
    private static final int MODE_SHIFT = 30;
    private static final int MODE_MASK = 0x3 << MODE_SHIFT;

    // 不指定测量模式
    public static final int UNSPECIFIED = 0 << MODE_SHIFT;

    // 精确测量模式
    public static final int EXACTLY = 1 << MODE_SHIFT;

    // 最大值测量模式
    public static final int AT_MOST = 2 << MODE_SHIFT;

    // 根据指定的大小和模式创建一个 MeasureSpec
    public static int makeMeasureSpec(int size, int mode) {
        if (sUseBrokenMakeMeasureSpec) {
            return size + mode;
        } else {
            return (size & ~MODE_MASK) | (mode & MODE_MASK);
        }
    }

    // 微调某个MeasureSpec的大小
    static int adjust(int measureSpec, int delta) {
        final int mode = getMode(measureSpec);
        if (mode == UNSPECIFIED) {
            // No need to adjust size for UNSPECIFIED mode.
            return makeMeasureSpec(0, UNSPECIFIED);
        }
        int size = getSize(measureSpec) + delta;
        if (size < 0) {
            Log.e(VIEW_LOG_TAG, "MeasureSpec.adjust: new size would be
negative! ( " + size +
                ") spec: " + toString(measureSpec) + " delta: " +
```

```

delta);
        size = 0;
    }
    return makeMeasureSpec(size, mode);
}
}

```

我们需要重点关注代码中的以下三种测量模式，这个在后面的 Measure 阶段会用到。

- UNSPECIFIED：不指定测量模式，父视图没有限制子视图的大小，子视图可以是想要的任何尺寸，通常用于系统内部，应用开发中很少使用到。
- EXACTLY：精确测量模式，当该视图的 layout_width 或者 layout_height 指定为具体数值或者 match_parent 时生效，表示父视图已经决定了子视图的精确大小，这种模式下 View 的测量值就是 SpecSize 的值。
- AT_MOST：最大值模式，当该视图的 layout_width 或者 layout_height 指定为 wrap_content 时生效，此时子视图的尺寸可以是不超过父视图允许的最大尺寸的任何尺寸。

对 DecorView 而言，它的 MeasureSpec 由窗口尺寸和其自身的 LayoutParams 共同决定；对于普通的 View，它的 MeasureSpec 由父视图的 MeasureSpec 和其自身的 LayoutParams 共同决定。

2.3 Measure

Measure 操作用来计算 View 的实际大小，由前面的分析可知，页面的测量流程是从 performMeasure 方法开始的，核心代码如下。

```

private void performMeasure(int childWidthMeasureSpec, int
childHeightMeasureSpec) {
    ...
    mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);
    ...
}

```

可以看到，具体的测量操作是分发给 ViewGroup 的，由 ViewGroup 在它的 measureChild 方法中传递给子 View，代码如下。ViewGroup 通过遍历自身所有的子 View，并逐个调用子 View

的 `measure` 方法实现测量操作。

```
// 遍历测量 ViewGroup 中所有的 View
protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {
    final int size = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < size; ++i) {
        final View child = children[i];
        // 当 View 的可见性处于 GONE 状态时, 不对其进行测量
        if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {
            measureChild(child, widthMeasureSpec, heightMeasureSpec);
        }
    }
}

// 测量某个指定的 View
protected void measureChild(View child, int parentWidthMeasureSpec,
    int parentHeightMeasureSpec) {
    final LayoutParams lp = child.getLayoutParams();

    // 根据父容器的 MeasureSpec 和子 View 的 LayoutParams 等信息计算子 View 的 MeasureSpec
    final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
        mPaddingLeft + mPaddingRight, lp.width);
    final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
        mPaddingTop + mPaddingBottom, lp.height);

    child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
}
```

下面来看看 `View (ViewGroup)` 的 `measure` 方法, 最终的测量是通过回调 `onMeasure` 方法实现的, 这个通常由 `View` 的特定子类自己实现, 开发者也可以通过重写这个方法实现自定义 `View`。

```
public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
    ...
    onMeasure(widthMeasureSpec, heightMeasureSpec);
    ...
}

// 如果需要自定义测量过程, 则子类可以重写这个方法
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // setMeasuredDimension方法用于设置View的测量宽高
    setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(),
widthMeasureSpec),
        getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
}

// 如果 View 没有重写 onMeasure 方法, 则默认会直接调用 getDefaultSize 来获得 View
的宽高
public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    switch (specMode) {
        case MeasureSpec.UNSPECIFIED:
            result = size;
            break;
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
            result = specSize;
            break;
    }
    return result;
}
```

2.4 Layout

Layout 过程用来确定 View 在父容器中的布局位置，它是由父容器获取子 View 的位置参数后，调用子 View 的 layout 方法并将位置参数传入实现的，ViewRootImpl 的 performLayout 代码如下。

```
// ViewRootImpl.java
private void performLayout(WindowManager.LayoutParams lp, int
desiredWindowWidth,
    int desiredWindowHeight) {
    ...
    host.layout(0, 0, host.getMeasuredWidth(), host.getMeasuredHeight());
    ...
}

// View.java
public void layout(int l, int t, int r, int b) {
    ...
    onLayout(changed, l, t, r, b);
    ...
}

// 空方法，子类如果是 ViewGroup 类型，则重写这个方法，实现 ViewGroup 中所有 View 控件
// 布局流程
protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
}
```

2.5 Draw

Draw 操作用来将控件绘制出来，绘制的流程从 performDraw 方法开始，核心代码如下。

```
private void performDraw() {
    ...
}
```

```

        draw(fullRedrawNeeded);
        ...
    }

    private void draw(boolean fullRedrawNeeded) {
        ...
        if (!drawSoftware(surface, mAttachInfo, xOffset, yOffset,
            scalingRequired, dirty)) {
            return;
        }
        ...
    }

    private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int
        xoff, int yoff,
        boolean scalingRequired, Rect dirty) {
        ...
        mView.draw(canvas);
        ...
    }

```

可以看到最终调用到每个 View 的 draw 方法绘制每个具体的 View，绘制基本上可以分为六个步骤，代码如下。

```

public void draw(Canvas canvas) {
    ...
    // 步骤一：绘制 View 的背景
    drawBackground(canvas);

    ...
    // 步骤二：如果需要的话，保存 canvas 的图层，为 fading 做准备
    saveCount = canvas.getSaveCount();
    ...
    canvas.saveLayer(left, top, right, top + length, null, flags);

```

```
...
// 步骤三: 绘制 View 的内容
onDraw(canvas);

...
// 步骤四: 绘制 View 的子 View
dispatchDraw(canvas);

...
// 步骤五: 如果需要的话, 绘制 View 的 fading 边缘并恢复图层
canvas.drawRect(left, top, right, top + length, p);
...
canvas.restoreToCount(saveCount);

...
// 步骤六: 绘制 View 的装饰 (例如滚动条)
onDrawScrollBars(canvas);
}
```

第3章

Android 动画机制

Android 应用开发中经常需要实现各种各样的动画效果，随着交互设计师给出的动画效果日趋复杂和高级，Android 的动画框架也在不断进化以满足需求，Android 发展到今天，有多少种可以使用的动画类型呢？本章将给出答案。

熟练掌握 Android 动画机制并实现出符合设计师要求的动画效果，是一名中高级 Android 应用开发者的基本功。在 Android 3.0 之前的版本，我们能使用的动画类型有两种，分别是逐帧动画和补间动画；在 Android 3.0 发布时，Android SDK 又为开发者带来了更加强大灵活的属性动画，使得实现复杂的动画效果更加容易；随着时间的推移，在 Android 4.4 中，Android SDK 又为开发者带来了 `android.transition` 框架，这使得开发者可以通过一种更直观的方式定义动画效果。下面我们分别来介绍这四种动画类型。

3.1 逐帧动画 (Frame Animation)

逐帧动画也叫 `Drawable Animation`，是最简单最直观的动画类型，它利用人眼的视觉暂留效应——也就是光对视网膜所产生的视觉在光停止作用后，仍会保留一段时间的现象。

在 Android 中实现逐帧动画，就是由设计师给出一系列状态不断变化的图片，开发者可以指定动画中每一帧对应的图片和持续的时间，然后就可以开始播放动画了。具体而言，有两种方式可以定义逐帧动画，分别是采用 XML 资源文件和代码实现。

3.1.1 XML 资源文件方式

这是最常使用的方式，首先我们将每一帧的图片放到 `res/drawable` 目录中，然后在 `res/anim`

目录中新建一个动画 XML 文件，在这个文件中使用 <animation-list> 标签来定义动画帧序列，使用 <item> 标签来定义动画的每一帧，并在其中指定帧的持续时间等属性，格式如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false" >
    <item
        android:drawable="@drawable/common_loading_01"
        android:duration="120" />
    <item
        android:drawable="@drawable/common_loading_02"
        android:duration="120" />
    <item
        android:drawable="@drawable/common_loading_03"
        android:duration="120" />
    <!-- 省略其他类似的帧定义 -->
</animation-list>
```

其中 android:oneshot 用来控制动画是否循环播放，如果取值为 true，表示动画不会循环播放，否则动画将会循环播放；android:duration 用来指定每一帧的播放持续时间。

3.1.2 代码方式

在代码中定义逐帧动画也很简单，但不常用，语句如下。

```
AnimationDrawable animDrawable = new AnimationDrawable();
for(int i=1; i<5; i++) {
    int id = getResources().getIdentifier("common_loading_" + i,
        "drawable", getPackageName());
    Drawable drawable = getResources().getDrawable(id);
    animDrawable.addFrame(drawable, 120);
}
imageView.setBackgroundDrawable(animDrawable);
animDrawable.setOneShot(false);
```

定义好逐帧动画之后，可以在符合某个条件时触发或者停止动画的播放，伪代码如下。

```
// 获取 AnimationDrawable对象实例，用来控制动画的播放和停止
AnimationDrawable animDrawable = (AnimationDrawable) imageView.
getBackground();
// 动画的播放
animDrawable.start();
// 动画的停止
animDrawable.stop();
```

3.2 补间动画 (Tween Animation)

补间动画是指开发者无须定义动画过程中的每一帧，只需要定义动画的开发和结束这两个关键帧，并指定动画变化的时间和方式等，然后交由 Android 系统进行计算，通过在两个关键帧之间插入渐变值来实现平滑过渡，从而对 View 的内容完成一系列的图形变换来实现的动画效果，主要包括四种基本的效果：透明度变化 Alpha、大小变化 Scale、位移变化 Translate 及旋转变换 Rotate，这四种效果可以动态组合，从而实现复杂灵活的动画。同样，定义补间动画也可以分为 XML 资源文件和代码两种方式。不过在这之前，我们首先来认识一下插值器 Interpolator。

3.2.1 插值器 Interpolator

前面说到的 Android 系统会在补间动画的开始和结束关键帧之间插入渐变值，它依据的便是 Interpolator。具体来说，Interpolator 会根据类型的不同，选择不同的算法计算出在补间动画期间所需要动态插入帧的密度和位置，Interpolator 负责控制动画的变化速度，使得前面所说的四种基本动画效果能够以匀速、加速、减速、抛物线等多种速度进行变化。

具体到 Android 代码中，Interpolator 类其实是一个空接口，它继承自 TimeInterpolator，TimeInterpolator 时间插值器允许动画进行非线性运动变换，如加速和减速等，该接口中只有 float getInterpolation(float input) 这个方法。入参是一个 0.0~1.0 的值，返回值可以小于 0.0 也可以大于 1.0，代码如下。

```
public interface Interpolator extends TimeInterpolator {

}
```

```
public interface TimeInterpolator {  
    float getInterpolation(float input);  
}
```

Android SDK 默认提供了几个 Interpolator 的实现类，如下表 3-1 所示。

表3-1

插值器类型	功能说明
AccelerateDecelerateInterpolator	在动画开始与结束的时候速率改变比较慢，在中间的时候加速
AccelerateInterpolator	在动画开始的地方速率改变比较慢，然后开始加速
AnticipateInterpolator	动画开始的时候先向后，然后向前滑动
AnticipateOvershootInterpolator	动画开始的时候先向后，然后向前甩一定值后返回最后的值
BounceInterpolator	动画结束的时候弹起
CycleInterpolator	动画循环播放特定的次数，速率的改变遵循正弦曲线
DecelerateInterpolator	在动画开始的地方速率改变比较快，然后开始变慢
LinearInterpolator	动画以常量速率进行改变
OvershootInterpolator	动画向前甩一定值后再回到原来位置
PathInterpolator	新增的，通过定义路径坐标，动画可以按照路径坐标来运行；注意这里的坐标并不是指十字坐标系，而是单方向，也就是可以从 0~1，然后弹回 0.6 后再弹到 0.8，直到最后时间结束

当然，如果上述默认的插值器不符合我们的实际需求，开发者也可以通过实现 Interpolator 接口来编写自己的插值器。

Android SDK 使用 Animation 类来表示抽的动画类，上面介绍的补间动画四种基本类型分别对应如表 3-2 所示的几个类。

表3-2

动画类型	功能说明
AlphaAnimation	改变透明度的动画，创建动画时需要指定动画开始和结束的透明度，以及动画持续时间，透明度取值范围是 0 到 1
ScaleAnimation	缩放大小的动画，创建动画时需要指定动画开始和结束时在 X 轴和 Y 轴的缩放比，以及动画持续时间；同时由于缩放时以不同的点作为中心会产生不同的效果，因此也需要通过 pivotX 和 pivotY 指定缩放中心的坐标
TranslateAnimation	改变位置的动画，创建动画时需要指定动画开始和结束时的 X、Y 坐标，以及动画的持续时间
RotateAnimation	旋转动画，创建动画时需要指定动画开始和结束时的旋转角度，以及动画持续时间；同时由于旋转时以不同的点作为中心会产生不同的效果，因此也需要通过 pivotX 和 pivotY 指定旋转中心的坐标

3.2.2 AlphaAnimation

XML 方式就是在 `res/anim` 目录中新建 XML 文件，其中的内容如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_decelerate_
interpolator" >

    <translate
        android:duration="200"
        android:fromYDelta="100%p"
        android:toYDelta="0" />
    <!--toAlpha 1.0表示完全不透明 0.0表示完全透明-->
    <alpha
        android:duration="200"
        android:fromAlpha="0.0"
        android:toAlpha="1.0" />
</set>
```

`AlphaAnimation` 的构造函数只有两个参数，分别是初始的透明度和结束的透明度，语句如下。

```
public AlphaAnimation(float fromAlpha, float toAlpha) {
    mFromAlpha = fromAlpha;
    mToAlpha = toAlpha;
}
```

在代码中实现透明度动画很简单，只需创建一个 `AlphaAnimation` 实例，然后设置动画持续时间即可，语句如下。

```
public void alpha() {
    AlphaAnimation anim = new AlphaAnimation(0, 1); // 透明度从0变化到1
    anim.setDuration(500); // 持续时间500毫秒
    anim.setFillAfter(true); // 动画结束后保留结束状态
    mImageView.setAnimation(anim);
}
```

3.2.3 ScaleAnimation

XML 实现示例如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator" >
    <scale
        android:duration="2000"
        android:fromXScale="0.2"
        android:fromYScale="0.2"
        android:pivotX="50%"
        android:pivotY="50%"
        android:toXScale="1.5"
        android:toYScale="1.5" />
</set>
```

在代码实现中，ScaleAnimation 可用的构造函数有三个，代码如下。

```
public ScaleAnimation(float fromX, float toX, float fromY, float toY) {
}

public ScaleAnimation(float fromX, float toX, float fromY, float toY,
    float pivotX, float pivotY) {
}

public ScaleAnimation(float fromX, float toX, float fromY, float toY,
    int pivotXType, float pivotXValue, int pivotYType, float pivotYValue) {
}
```

可以看到，涉及到的人参比较多，具体含义如表 3-3 所示。

ScaleAnimation 的使用如下。

```
public void scale() {
    ScaleAnimation anim = new ScaleAnimation(1.0f, 4.0f, 1.0f, 4.0f,
        Animation.RELATIVE_TO_SELF, 0.0f,
        Animation.RELATIVE_TO_SELF, 0.0f);
}
```

```

anim.setDuration(2000);
anim.setFillAfter(true);
mImageView.setAnimation(anim);

```

```

}

```

表3-3

参 数	含 义 说 明
fromX	动画开始时的 X 坐标的伸缩尺寸
toX	动画结束时的 X 坐标的伸缩尺寸
fromY	动画开始时的 Y 坐标的伸缩尺寸
toY	动画结束时的 Y 坐标的伸缩尺寸
pivotX/pivotXValue	缩放动画的中心点 X 坐标
pivotY/pivotYValue	缩放动画的中心点 Y 坐标
pivotXType	动画在 X 轴的伸缩模式, 也就是中心点相对于哪个物件, 取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
pivotYType	动画在 Y 轴的伸缩模式, 也就是中心点相对于哪个物件, 取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT

3.2.4 TranslateAnimation

XML 实现示例如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:duration="200"
        android:fromXDelta="0"
        android:fromYDelta="0"
        android:toXDelta="0"
        android:toYDelta="1000" />
</set>

```

在代码实现中, TranslateAnimation 可用的构造函数有两个, 代码如下。

```

public TranslateAnimation(float fromXDelta, float toXDelta, float fromYDelta,
float toYDelta) {
}

```

```
public TranslateAnimation(int fromXType, float fromXValue, int toXType, float
toXValue,
    int fromYType, float fromYValue, int toYType, float toYValue) {
}
```

其中，入参的含义如表 3-4 所示。

表3-4

参 数	含义说明
fromXType	动画开始时在 X 轴的位移模式，取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
fromXValue/fromXDelta	动画开始时当前 View 的 X 坐标
toXType	动画结束时在 X 轴的位移模式，取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
toXValue/toXDelta	动画结束时当前 View 的 X 坐标
fromYType	动画开始时在 Y 轴的位移模式，取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
fromYValue/fromYDelta	动画开始时当前 View 的 Y 坐标
toYType	动画结束时在 Y 轴的位移模式，取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
toYValue/toYDelta	动画结束时当前 View 的 Y 坐标

TranslateAnimation 的使用如下。

```
public void translate() {
    TranslateAnimation anim = new TranslateAnimation(Animation.RELATIVE_TO_SELF, 0f,
        Animation.RELATIVE_TO_SELF, 2f, Animation.RELATIVE_TO_SELF,
        0f, Animation.RELATIVE_TO_SELF, 2f);
    anim.setDuration(3000);
    anim.setFillAfter(true);
    mImageView.setAnimation(anim);
}
```

3.2.5 RotateAnimation

XML 实现示例如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <rotate
        android:duration="1000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:startOffset="0"
        android:repeatCount="-1"
        android:repeatMode="restart"
        android:toDegrees="360" />
</set>

```

在代码实现中, RotateAnimation 可用的构造函数有三个, 代码如下。

```

public RotateAnimation(float fromDegrees, float toDegrees) {
}

public RotateAnimation(float fromDegrees, float toDegrees, float pivotX, float
pivotY) {
}

public RotateAnimation(float fromDegrees, float toDegrees, int pivotXType,
float pivotXValue,
    int pivotYType, float pivotYValue) {
}

```

其中, 入参的涵义含义如表 3-5 所示。

表3-5

参 数	含义说明
fromDegrees	动画开始时的旋转角度
toDegrees	动画结束时的旋转角度
pivotXType	动画在 X 轴的旋转模式, 即相对于物件的位置类型, 取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
pivotXValue	动画相对于物件的 X 坐标开始位置
pivotYType	动画在 Y 轴的旋转模式, 即相对于物件的位置类型, 取值是 Animation.ABSOLUTE、Animation.RELATIVE_TO_SELF 或 Animation.RELATIVE_TO_PARENT
pivotYValue	动画相对于物件的 Y 坐标开始位置

RotateAnimation 的使用如下。

```
public void rotate() {  
    RotateAnimation anim = new RotateAnimation(0, -720,  
        RotateAnimation.RELATIVE_TO_SELF, 0.5f,  
        RotateAnimation.RELATIVE_TO_SELF, 0.5f);  
    anim.setDuration(1000);  
    anim.setFillAfter(true);  
    mImageView.setAnimation(anim);  
}
```

3.2.6 自定义补间动画

在实际的项目中，往往会遇到使用上述四种基本动画无法实现的动画需求，这时可以考虑自定义补间动画，只需要继承 Animation，并重写这个抽象基类中的 applyTransformation 方法，在其中实现具体的动画变换逻辑，语句代码如下。

```
public class MyAnimation extends Animation {  
  
    @Override  
    protected void applyTransformation(float interpolatedTime, Transformation  
    transformation) {  
        super.applyTransformation(interpolatedTime, transformation);  
    }  
}
```

其中，interpolatedTime 表示动画的时间进行比，无论动画的实际持续时间是多少，这个参数总是会从 0 变化到 1。transformation 表示补间动画在不同时刻对 View 的变形程度。

3.3 属性动画 (Property Animation)

属性动画是在 Android 3.0 中引入的，在补间动画中，我们只能改变 View 的绘制效果，View 的真实属性是没有变化的，而属性动画则可以直接改变 View 对象的属性值，同时属性动

画几乎可以对任何对象执行动画，而不是局限在 View 对象上，从某种意义上讲，属性动画可以说是增强版的补间动画。与补间动画类似，属性动画也涉及如表 3-6 所示的基本属性。

表3-6

动画属性	说 明
动画持续时间	默认值是 300 毫秒，在资源文件中通过 android:duration 指定
动画插值方式	详见补间动画介绍，在资源文件中通过 android:interpolator 指定
动画重复次数	指定动画重复播放的次数，在资源文件中通过 android:repeatCount 指定
动画重复模式	指定一次动画播放结束后，重复下次动画时，是从开始帧再次播放到结束帧，还是从结束帧反方向播放到开始帧，在资源文件中通过 android:repeatMode 指定
帧刷新频率	指定间隔多长时间播放一帧，默认值是 10 毫秒
动画集合	通过动画集合可以将多个属性动画组合起来，同时通过指定 android:ordering 属性可以控制这组动画是按次序播放还是同时播放，在资源文件中通过 <set>...</set> 来表示

属性动画的基类是 Animator，它是一个抽象类，所以不会直接使用这个类，通常都是继承它并重写其中的相关方法，Android SDK 为开发者默认提供了几个子类，大多数情况下使用这些子类就足够完成开发任务了。

3.3.1 Evaluator

在介绍 Animator 的子类之前，我们首先来了解一个名为 Evaluator 的概念，它是用来控制属性动画如何计算属性值的。它的接口定义是 TypeEvaluator，其中定义了 evaluate 方法，供不同类型的子类实现。

```
public interface TypeEvaluator<T> {
    public T evaluate(float fraction, T startValue, T endValue);
}
```

常见的实现类有 IntEvaluator、FloatEvaluator、ArgbEvaluator 等。下面我们来看一下 ArgbEvaluator 的具体实现，可以看到实现逻辑很简单，就是根据输入的初始值和结束值及一个进度比，计算出每一个进度对应的 ARGB 值。

```
public class ArgbEvaluator implements TypeEvaluator {
    private static final ArgbEvaluator sInstance = new ArgbEvaluator();

    public static ArgbEvaluator getInstance() {
        return sInstance;
    }
}
```

```

    }

    public Object evaluate(float fraction, Object startValue, Object
endValue) {
        int startInt = (Integer) startValue;
        int startA = (startInt >> 24) & 0xff;
        int startR = (startInt >> 16) & 0xff;
        int startG = (startInt >> 8) & 0xff;
        int startB = startInt & 0xff;

        int endInt = (Integer) endValue;
        int endA = (endInt >> 24) & 0xff;
        int endR = (endInt >> 16) & 0xff;
        int endG = (endInt >> 8) & 0xff;
        int endB = endInt & 0xff;

        return (int)((startA + (int)(fraction * (endA - startA))) << 24) |
            (int)((startR + (int)(fraction * (endR - startR))) << 16) |
            (int)((startG + (int)(fraction * (endG - startG))) << 8) |
            (int)((startB + (int)(fraction * (endB - startB))));
    }
}

```

3.3.2 AnimatorSet

AnimatorSet 也是 Animator 的子类，用来组合多个 Animator，并指定这些 Animator 是顺序播放还是同时播放。

3.3.3 ValueAnimator

ValueAnimator 是属性动画最重要的一个类，继承自 Animator。它定义了属性动画大部分的核心功能，包括计算各个帧的属性值、处理更新事件、按照属性值的类型控制计算规则等。

一个完整的属性动画由以下两部分组成。

- 计算动画各个帧的相关属性值。
- 将这些属性值设置给指定的对象。

`ValueAnimator` 为开发者实现了第一部分的功能，第二部分功能由开发者自行设置。`ValueAnimator` 的构造函数是空实现，一般都是使用如下的静态工厂方法来进行实例化。

```
public static ValueAnimator ofInt(int... values) {
    ValueAnimator anim = new ValueAnimator();
    anim.setIntValues(values);
    return anim;
}

public static ValueAnimator ofArgb(int... values) {
    ValueAnimator anim = new ValueAnimator();
    anim.setIntValues(values);
    anim.setEvaluator(ArgbEvaluator.getInstance());
    return anim;
}

public static ValueAnimator ofFloat(float... values) {
    ValueAnimator anim = new ValueAnimator();
    anim.setFloatValues(values);
    return anim;
}

public static ValueAnimator ofPropertyValuesHolder(PropertyValuesHolder...
values) {
    ValueAnimator anim = new ValueAnimator();
    anim.setValues(values);
    return anim;
}

public static ValueAnimator ofObject(TypeEvaluator evaluator, Object...
values) {
    ValueAnimator anim = new ValueAnimator();
    anim.setObjectValues(values);
    anim.setEvaluator(evaluator);
}
```

```

        return anim;
    }

```

获取到实例后，接着需要设置动画持续时间、插值方式、重复次数等属性值，然后启动动画，最后还需要为 ValueAnimator 注册 AnimatorUpdateListener 监听器，并在这个监听器的 onAnimationUpdate 方法中将计算出来的属性值设置给指定对象。我们从 React Native 这个开源框架中可以看到用法如下。

```

int curColor = activity.getWindow().getStatusBarColor();
ValueAnimator colorAnimation = ValueAnimator.ofObject(
    new ArgbEvaluator(), curColor, color);

colorAnimation.addUpdateListener(new ValueAnimator.AnimatorUpdateListener()
{
    @Override
    public void onAnimationUpdate(ValueAnimator animator) {
        activity.getWindow().setStatusBarColor((Integer) animator.
getAnimatedValue());
    }
});
colorAnimation.setDuration(300).setStartDelay(0);
colorAnimation.start();

```

3.3.4 ObjectAnimator

ObjectAnimator 是 ValueAnimator 的子类，封装实现了上面所说的第二部分的功能。因此，在实际开发中用得最多的就是 ObjectAnimator，只有在 ObjectAnimator 实现不了的场景下，才考虑使用 ValueAnimator。ObjectAnimator 和 ValueAnimator 在构造实例时最大的不同是需要指定动画作用的具体对象和对象的属性名，而且一般不需要注册 AnimatorUpdateListener 监听器，示例如下。

```

public class CustomCircleProgressBar extends CircularProgressBar {
    public CustomCircleProgressBar(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}

```

```

@Override
public void setProgressWithAnimation(float progress, int duration) {
    ObjectAnimator objectAnimator = ObjectAnimator.ofFloat(this,
        "progress", progress);
    objectAnimator.setDuration(duration);
    objectAnimator.start();
}
}

```

使用 ObjectAnimator 有以下几点需要注意。

- 需要为对象对应的属性提供 setter 方法。例如，上面的 progress 属性在父类 CircularProgressBar 提供了如下设置方法。

```

public void setProgress(float progress) {
    this.progress = (progress<=100) ? progress : 100;
    invalidate();
}

```

- 如果动画的对象是 View，那么为了能显示动画效果，在某些情况下，可能还需要注册 AnimatorUpdateListener 监听器，并在其回调方法 onAnimationUpdate 中调用 View 的 invalidate 方法来刷新 View 的显示。

上面所说的都是在代码中定义属性动画。事实上，与补间动画类似，属性动画也可以在 XML 文件中定义，在工程的 res/animator 目录中存放的就是属性动画 XML 文件，例子 scale.xml 如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:ordering="together" >

    <objectAnimator
        android:duration="2000"
        android:propertyName="scaleX"
        android:valueFrom="1"
        android:valueTo="0.4"

```

```

        android:valueType=" floatType" >
</objectAnimator>
<objectAnimator
    android:duration=" 2000"
    android:propertyName=" scaleY"
    android:valueFrom=" 1"
    android:valueTo=" 0.4"
    android:valueType=" floatType" >
</objectAnimator>

</set>

```

使用代码加载如下。

```

public void scaleY(View view) {
    Animator anim = AnimatorInflater.loadAnimator(this, R.animator.scale);
    anim.setTarget(mImageView);
    anim.start();
}

```

3.4 过渡动画 (Transition Animation)

过渡动画是在 Android 4.4 引入的新的动画框架，它本质上还是属性动画，只不过是属性动画做了一层封装，方便开发者实现 Activity 或者 View 的过渡动画效果。和属性动画相比，过渡动画最大的不同是需要为动画前后准备不同的布局，并通过对应的 API 实现两个布局的过渡动画，而属性动画只需要一个布局文件。

在使用 Transition Animation 框架实现动画效果之前，我们先来了解这个框架的几个基本概念。

- Scene：定义了页面的当前状态信息，Scene 的实例化一般通过静态工厂方法实现。

```

public static Scene getSceneForLayout(ViewGroup sceneRoot, int layoutId,
Context context) {
}

```

- **Transition**: 定义了界面之间切换的动画信息, 在使用 **TransitionManager** 时没有指定使用哪个 **Transition**, 那么会使用默认的 **AutoTransition**, 源码如下。可以看出 **AutoTransition** 的动画效果就是先隐藏对象变透明, 然后移动指定的对象, 最后显示出来。

```
public class AutoTransition extends TransitionSet {
    public AutoTransition() {
        init();
    }

    public AutoTransition(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    private void init() {
        setOrdering(ORDERING_SEQUENTIAL);
        addTransition(new Fade(Fade.OUT));
        addTransition(new ChangeBounds());
        addTransition(new Fade(Fade.IN));
    }
}
```

- **TransitionManager**: 控制 **Scene** 之间切换的控制器, 切换常用的方法有以下两个, 其中的 **sDefaultTransition** 就是前面说的 **AutoTransition** 的实例。

```
public static void go(Scene scene) {
    changeScene(scene, sDefaultTransition);
}

public static void go(Scene scene, Transition transition) {
    changeScene(scene, transition);
}
```

过渡动画的使用很简单, 首先定义同一个页面的两个布局, 分别是动画前的布局和动画后的布局, 我们将其命名为 `fragment_transition_scene_before.xml` 和 `fragment_transition_scene_after.xml`, 这两个布局文件的根布局具有相同的 `android:id` 值, 动画前的布局文件内容如下。

```
<!--fragment_transition_scene_before.xml-->
<RelativeLayout
    xmlns:android=" http://schemas.android.com/apk/res/android"
    android:id="@+id/scene"
    android:layout_width=" match_parent"
    android:layout_height=" match_parent" >

    <TextView
        android:id="@+id/textView"
        android:text="@string/hello_world"
        android:layout_width=" wrap_content"
        android:layout_height=" wrap_content" />

    <Button
        android:id="@+id/goButton"
        android:text="@string/button_go"
        android:layout_below="@id/textView"
        android:layout_width=" wrap_content"
        android:layout_height=" wrap_content" />
</RelativeLayout>
```

动画后的布局文件内容如下。

```
<!--fragment_transition_scene_after.xml-->
<RelativeLayout
    xmlns:android=" http://schemas.android.com/apk/res/android"
    android:id="@+id/scene"
    android:layout_width=" match_parent"
    android:layout_height=" match_parent" >

    <TextView
        android:id="@+id/textView"
        android:text="@string/hello_world"
        android:layout_width=" wrap_content"
        android:layout_height=" wrap_content" />
```

```

<Button
    android:id="@+id/goButton"
    android:text="@string/button_go"
    android:layout_below="@id/textView"
    android:layout_alignParentBottom="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</RelativeLayout>

```

Transition Animation 使用代码实现的例子如下。

```

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

    if (savedInstanceState == null) {
        getFragmentManager().beginTransaction()
            .add(R.id.container, new TransitionFragment())
            .commit();
    }

    // XML 文件方式
    ViewGroup container = (ViewGroup)findViewById(R.id.container);
    TransitionInflater transitionInflater = TransitionInflater.from(this);
    mTransitionManager = transitionInflater.inflateTransitionManager(R.
transition.transition_manager, container);
    mScene1 = Scene.getSceneForLayout(container, R.layout.fragment_
transition_scene_before, this);
    mScene2 = Scene.getSceneForLayout(container, R.layout.fragment_
transition_scene_after, this);
}

// 代码方式

```

```
private void goToScene(Scene scene) {
    ChangeBounds changeBounds = new ChangeBounds();
    changeBounds.setDuration(2000);
    Fade fadeOut = new Fade(Fade.OUT);
    fadeOut.setDuration(2000);
    Fade fadeIn = new Fade(Fade.IN);
    fadeIn.setDuration(2000);
    TransitionSet transition = new TransitionSet();
    transition.setOrdering(TransitionSet.ORDERING_SEQUENTIAL);
    transition
        .addTransition(fadeOut)
        .addTransition(changeBounds)
        .addTransition(fadeIn);
    TransitionManager.go(scene, transition);
}
```

上面的代码分别演示了 XML 布局和代码实现两种方式，过渡动画的 XML 文件需要放在 res/transition 目录中，其中 transition_manager.xml 定义如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<transitionManager
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <transition
        android:fromScene="@layout/fragment_transition_scene_1"
        android:toScene="@layout/fragment_transition_scene_2"
        android:transition="@transition/slow_auto_transition" />
    <transition
        android:fromScene="@layout/fragment_transition_scene_1"
        android:toScene="@layout/fragment_transition_scene_2"
        android:transition="@transition/slow_auto_transition" />
</transitionManager>
```

其中的 slow_auto_transition 也是定义在 res/transition 目录中的自定义 Transition 动画文件。

```
<?xml version="1.0" encoding="utf-8" ?>
<transitionSet
```



```
xmlns:android="http://schemas.android.com/apk/res/android"
android:transitionOrdering="sequential" >
<fade
    android:fadingMode="fade_out"
    android:duration="1000" />
<changeBounds
    android:duration="2000"
    android:interpolator="@android:interpolator/anticipate_
overshoot" />
<fade
    android:fadingMode="fade_in"
    android:duration="1000" />
</transitionSet>
```

第4章

Support Annotation Library 使用详解

Support Annotation Library 是从 Android Support Library 19.1 开始引入的一个全新的函数包，它包含一系列有用的元注解，用来帮助开发者在编译期间发现可能存在的 Bug。Support Library 本身也使用 Annotation Library 提供的注解来完善自身的代码质量，Android Studio 天然地支持 Annotation Library，并提供可视化的交互以方便开发者发现问题。

在 Android Support Library 22.2 中，新增了 13 种新的 Annotation Library 注解，因此，在实际开发中我们应该尽量使用最新版本的函数包，以便能够使用更多的元注解来提高代码质量。

Android Support Library 发展到现在，已经不止是一个 Jar 包，而是拆分成多个独立的 Jar 包，例如 support-v4、appcompat-v7、gridlayout-v7、design、cardview-v7 等。完整的 support library 包含以下几个系列的 Jar 包。

```
com.android.support:support-annotations:23.1.1
com.android.support:support-v4:23.1.1
com.android.support:support-v13:23.1.1
com.android.support:appcompat-v7:23.1.1
com.android.support:design:23.1.1
com.android.support:gridlayout-v7:23.1.1
com.android.support:mediarouter-v7:23.1.1
com.android.support:cardview-v7:23.1.1
com.android.support:palette-v7:23.1.1
com.android.support:recyclerview-v7:23.1.1
com.android.support:leanback-v17:23.1.1
```

Annotation Library 也是其中之一，默认情况下不会包含在工程中。如果我们的 SDK 已经安装了 Android Support Repository，那么我们可以通过打开工程的 Project Structure 对话框，并选中一个 Module，选中 Dependencies 选项卡，点击“+”按钮，在弹出的 Choose Library Dependency 对话框中轻松找到 Annotation Library，如图 4-1 所示。

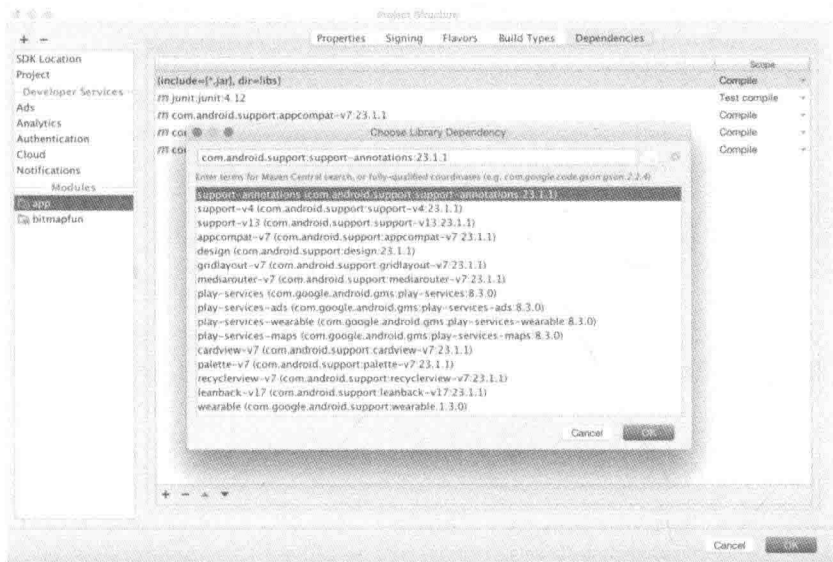


图 4-1

点击添加后，在 Module 的 build.gradle 文件中会新增 Annotation 函数库的依赖代码如下。

```
dependencies {
    compile 'com.android.support:support-annotations:23.1.1'
}
```

在 support-annotation-23.1.1 函数包中，总共包含 39 种注解，下面我们按照类型分别进行介绍。

4.1 Nullness 注解

此类注解包含如下内容。

- @Nullable 作用于函数参数或者返回值，标记参数或者返回值可以为空。

- @NonNull作用于函数参数或者返回值，标记参数或者返回值不可以为空。

当出现这种违反注解标记的代码时，Android Studio 会给出提示，同时使用 Android Lint 进行静态代码扫描，也会显示出错提示。下面以 @NonNull 注解为例进行说明，当 helloWorld 函数的参数传入 null 时，在 Android Studio 中会出现如图 4-2 所示的警告。

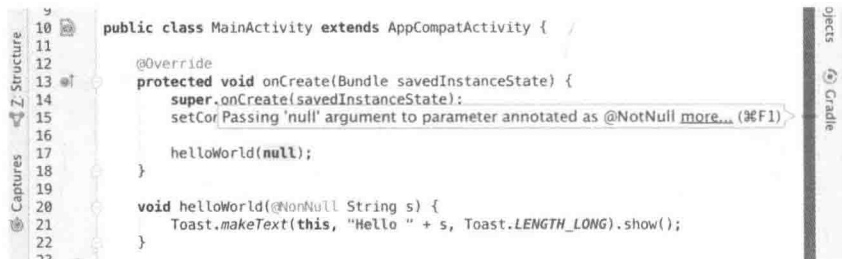


图 4-2

另一方面，如果使用 Android Lint 扫描这个文件的话，在扫描结果中会显示如图 4-3 所示的警告。

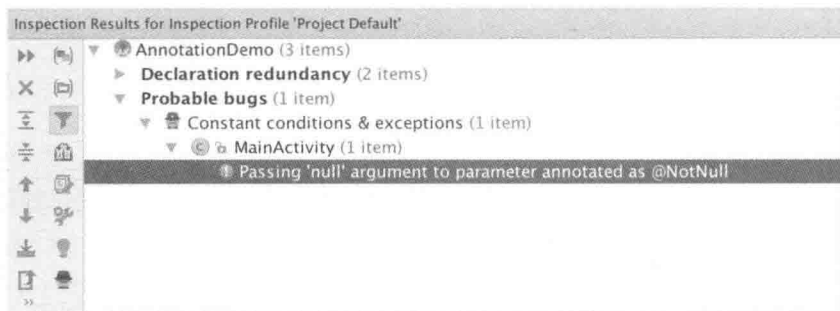


图 4-3

后面其他类型的注解类型也会有这两种类型的提示，我们将不再赘述。

4.2 资源类型注解

我们知道资源在 Android 中通常是以整型值表示的，并保存在 R.java 文件中。这意味着一个需要传入 Layout 资源值的函数，如果传入 String 资源值不会在编译期报错，只有在运行时执行到相应的代码才能发现问题，则使用资源类型注解可以防止这种情况的出现。

资源类型的注解作用于函数参数、返回值及类的变量，在 support-annotations-23.1.1 中，每种资源类型对应一个注解。

- AnimatorRes: 标记整型值是 android.R.animator 类型。
- AnimRes: 标记整型值是 android.R.anim 类型。
- AnyRes: 标记整型值是任何一种资源类型，如果确切知道表示的是哪一种具体资源的话，建议显式指定。
- ArrayRes: 标记整型值是 android.R.array 类型。
- AttrRes: 标记整型值是 android.R.attr 类型。
- BoolRes: 标记整型值是布尔类型。
- ColorRes: 标记整型值是 android.R.color 类型。
- DrawableRes: 标记整型值是 android.R.drawable 类型。
- FractionRes: 标记整型值是 fraction 类型，这个比较少见，这种类型的资源常见于 Animation Xml 中，比如 50%p，表示占 parent 的 50%。
- IdRes: 标记整型值是 android.R.id 类型。
- IntegerRes: 标记整型值是 android.R.integer 类型。
- InterpolatorRes: 标记整型值是 android.R.interpolator 类型。
- LayoutRes: 标记整型值是 android.R.layout 类型。
- MenuRes: 标记整型值是 android.R.menu 类型。
- PluralsRes: 标记整型值是 android.R.plurals 类型，表示复数字符串类型，具体可以参见官方文档¹。
- RawRes: 标记整型值是 android.R.raw 类型。
- StringRes: 标记整型值是 android.R.string 类型。
- StyleableRes: 标记整型值是 android.R.styleable 类型。
- StyleRes: 标记整型值是 android.R.style 类型。

1 <http://developer.android.com/guide/topics/resources/string-resource.html#Plurals>

- TransitionRes: 标记整型值是 transition 类型。
- XmlRes: 标记整型值是 android.R.xml 类型。

我们来看一个例子, support-v7 包中 AppCompatActivity 的 setContentView 函数使用 @LayoutRes 标记它的参数。

```
@Override
public void setContentView(@LayoutRes int layoutResID) {
    getDelegate().setContentView(layoutResID);
}
```

我们的 MainActivity 继承自 AppCompatActivity, 在 onCreate 函数中调用 setContentView 函数, 如果传入的参数不是 R.Layout 类型, 而是其他资源类型 (例如 R.String 类型), 那么 Android Studio 会提示如图 4-4 的错误。

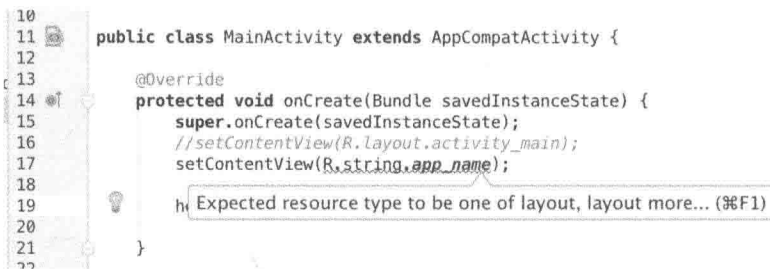


图 4-4

4.3 类型定义注解

在 Android 开发中, 整型值不止经常用来代表资源引用值, 而且经常用来代替枚举值。@IntDef 注解用来创建一个整型类型定义的新注解, 我们可以使用这个新注解来标记自己编写的 API。下面这段代码是从 appcompat 函数包中抽取出来的, 可以看到 @IntDef 的用法。

```
import android.support.annotation.IntDef;
...
public abstract class ActionBar {

    // 告知编译器不要在.class文件中存储注解数据
```

```

@Retention(RetentionPolicy.SOURCE)
// 定义可以接受的常量列表
@IntDef({NAVIGATION_MODE_STANDARD, NAVIGATION_MODE_LIST, NAVIGATION_
MODE_TABS})
// 定义 NavigationMode 注解
public @interface NavigationMode {}

// 常量定义
public static final int NAVIGATION_MODE_STANDARD = 0;
public static final int NAVIGATION_MODE_LIST = 1;
public static final int NAVIGATION_MODE_TABS = 2;

@NavigationMode
public abstract int getNavigationMode();

public abstract void setNavigationMode(@NavigationMode int mode);

...
}

```

在使用 `setNavigationMode` 这个 API 时，如果传入的参数 `mode` 不是三个常量值之一，那么 Android Studio 就会给出警告。

除了类似上面定义，我们也可以定义一个 `flag` 标识位，来识别函数参数或者返回值是否符合某一种模式，语句如下。

```

@IntDef(flag=true, value={
    NAVIGATION_MODE_STANDARD,
    NAVIGATION_MODE_LIST,
    NAVIGATION_MODE_TABS
})
@Retention(RetentionPolicy.SOURCE)
public @interface NavigationMode {}

@NavigationMode
public abstract int getNavigationMode();

public abstract void setNavigationMode(@NavigationMode int mode);

```

如果符合，那么可以如下调用。

```
setNavigationMode(ActionBar.NAVIGATION_MODE_STANDARD | ActionBar.NAVIGATION_MODE_LIST);
```

4.4 线程注解

Android 应用开发过程中，经常会涉及多种线程的使用，界面相关操作必须在主线程，而耗时操作例如文件下载等则需要放到后台线程中。线程相关注解有四种。

- `@UiThread`：标记运行在 UI 线程，一个 UI 线程是 Activity 运行所在的主窗口，对于一个应用而言，可能存在多个 UI 线程，每个 UI 线程对应不同的主窗口。
- `@MainThread`：标记运行在主线程，一个应用只有一个主线程，主线程也是 `@UiThread` 线程。通常情况下，我们使用 `@MainThread` 来注解生命周期相关函数，使用 `@UiThread` 来注解视图相关函数，一般情况下，`@MainThread` 和 `@UiThread` 是可互换使用的。
- `@WorkerThread`：标记运行在后台线程。
- `@BinderThread`：标记运行在 Binder 线程。

一个典型的例子是 `AsyncTask` 的实现，我们截取部分代码如下。

```
@MainThread
protected void onPreExecute() {}

@WorkerThread
protected abstract Result doInBackground(Params... params);

@MainThread
protected void onProgressUpdate(Progress... values) {}
```

4.5 RGB 颜色值注解

在资源类型注解中我们使用 `@ColorRes` 来标记参数类型需要传入颜色类型的资源 id，本节

介绍的 @ColorInt 注解则是标记参数类型需要传入 RGB 或者 ARGB 颜色整型值。在 TextView 的源码中可以找到使用 @ColorInt 的例子。

```
public void setTextColor(@ColorInt int color) {
    mTextColor = ColorStateList.valueOf(color);
    updateTextColors();
}
```

4.6 值范围注解

当函数参数的取值是在一定范围内时,可以使用值范围注解来防止调用者传入错误的参数,这种类型主要有三种注解。

(1) @Size : 对于类似数组、集合和字符串之类的参数,我们可以使用 @Size 注解来表示这些参数的大小。用法如下。

@Size(min=1) // 可以表示集合不可以为空

@Size(max=23) // 可以表示字符串最大字符个数是 23

@Size(2) // 可以表示数组元素个数是 2 个

@Size(multiple=2) // 可以表示数组大小是 2 的倍数

(2) @IntRange : 参数类型是 int 或者 long, 用法如下。

```
public void setAlpha(@IntRange(from=0,to=255) int alpha) { ... }
```

(3) @FloatRange : 参数类型是 float 或者 double, 用法如下。

```
public void setAlpha(@FloatRange(from=0.0, to=1.0) float alpha) { ... }
```

4.7 权限注解

Android 应用在使用某些系统功能时,需要在 AndroidManifest.xml 中声明权限,否则在运行时会提示缺失对应的权限。为了在编译期及时发现缺失的权限,我们可以使用 @RequiresPermission 注解。

- 如果函数调用需要声明一个权限，语句如下。

```
@RequiresPermission(Manifest.permission.SET_WALLPAPER)
public abstract void setWallpaper(Bitmap bitmap) throws IOException;
```

- 如果函数调用需要声明集合中最少一个权限，语句如下。

```
@RequiresPermission(anyOf = {
    Manifest.permission.ACCESS_COARSE_LOCATION,
    Manifest.permission.ACCESS_FINE_LOCATION})
public abstract Location getLastKnownLocation(String provider);
```

- 如果函数调用需要同时声明多个权限，语句如下。

```
@RequiresPermission(allOf = {
    Manifest.permission.READ_HISTORY_BOOKMARKS,
    Manifest.permission.WRITE_HISTORY_BOOKMARKS})
public static final void updateVisitedHistory(ContentResolver cr, String url,
boolean real);
```

- 对于 Intent 调用所需权限，可以在 Intent 的 ACTION 字符串定义处添加注解，语句如下。

```
@RequiresPermission(android.Manifest.permission.BLUETOOTH)
public static final String ACTION_REQUEST_DISCOVERABLE =
    "android.bluetooth.adapter.action.REQUEST_DISCOVERABLE";
```

- 对于 ContentProvider 相关所需的权限，可能同时需要读和写这两个操作，对应不同的权限声明，语句如下。

```
@RequiresPermission.Read(@RequiresPermission(READ_HISTORY_BOOKMARKS))
@RequiresPermission.Write(@RequiresPermission(WRITE_HISTORY_BOOKMARKS))
public static final Uri BOOKMARKS_URI = Uri.parse("content://browser/
bookmarks");
```

4.8 重写函数注解

如果 API 允许调用者重写某个函数，但同时要求重写的函数需要调用被重写的函数，否则

代码逻辑可能会错误，那么可以使用 `@CallSuper` 注解来提示开发者，语句如下。

```
@CallSuper
protected void onCreate(@Nullable Bundle savedInstanceState);
```

4.9 返回值注解

如果我们编写的函数需要调用者对返回值做某些处理，那么可以使用 `@CheckResult` 注解来提示开发者。当然我们没有必要对每个非空返回值的函数都添加这个注解，该注解的主要目的是让调用者在使用 API 时不至于怀疑该函数是否会产生副作用。在 Android 源码中，Context 类的 `checkPermission` 函数使用了该注解。

```
@CheckResult(suggest="#enforcePermission(String,int,int,String)")
@PackageManager.PermissionResult
public abstract int checkPermission(@NonNull String permission, int pid, int
uid);

@CheckResult(suggest="#enforceCallingOrSelfPermission(String,String)")
@PackageManager.PermissionResult
public abstract int checkCallingOrSelfPermission(@NonNull String
permission);
```

这样如果调用者没有检查这两个函数的返回值，那么 Android Studio 将会给出警告，警告信息中包含 `suggest` 属性中的内容。

4.10 @VisibleForTesting

单元测试中可能需要访问到一些不可见的类、函数或者变量，这时可以使用 `@VisibleForTesting` 注解来使其对测试可见。

4.11 @Keep

`@Keep` 注解用来标记在 Proguard 混淆过程中不需要混淆的类或者方法。如果你曾经在编写

混淆文件时使用过，那么 @Keep 的用法很简单。

```
-keep class com.foo.bar { public static <methods> }
```

如果有了 @Keep 注解,则可以在代码编写过程中对不需要混淆的类或者方法直接标记即可。

```
public class AnnotaionDemo {  
    @Keep  
    public void doSomething() {  
        // ...  
    }  
    // ...  
}
```

最后说明一下，如果函数库中使用 Annotation Library，并使用 Gradle 生成 aar 压缩包，那么在编译时 Android Gradle 插件会抽取出这些注解信息并打包在 aar 文件中，以便函数库的调用者正常使用我们的注解信息。aar 文件中的 annotations.zip 文件就是抽取出来的注解信息。

第5章

Percent Support Library 使用详解

Android 系统的碎片化发展导致了 Android 手机多种机型、多种分辨率、多种屏幕密度共存的状态，面对设计师提供的以 px 为单位的标注图，很多开发同学肯定为屏幕适配工作花过不少心思。在布局方面，我们知道 `LinearLayout` 的 `layout_weight` 属性使得我们在线性布局中可以实现按屏幕比例来排列控件，但是对于必须使用相对布局或者帧布局的情况，可能需要多嵌套一层 `LinearLayout`，以便实现按屏幕比例布局。某些情况下，我们甚至使用空白的 `View` 来实现百分比的 `margin`。这不仅使得布局文件复杂，而且会增加渲染的层次，导致性能下降。

2015 年的 8 月，Google 为我们带来了一个全新的百分比布局兼容函数库：Android Percent Support Library，解决了我们上文提到的问题。目前它支持 `RelativeLayout` 和 `FrameLayout` 的百分比布局。不过已经有开发者在 Github 上面开源了对 `LinearLayout` 的百分比支持。因此，是时候在实际项目开发中使用这个函数库了。

从名字可以看出，Percent 函数库是 Support Library 家族的一员，在使用它提供的 API 之前，首先需要在 Gradle 的 `build.gradle` 文件中加入依赖，如下所示。

```
dependencies {  
    compile 'com.android.support:percent:23.2.0'  
}
```

打开下载后的函数库，可以看到其中主要包含如下三个类。

- `PercentFrameLayout`
- `PercentRelativeLayout`

- PercentLayoutHelper

百分比逻辑的实现基本上都在 PercentLayoutHelper 这个类中，感兴趣的同学可以自行阅读。

首先我们需要认识到，PercentFrameLayout 和 PercentRelativeLayout 分别是对 FrameLayout 和 RelativeLayout 的继承。因此，我们仍然可以使用父类既有的属性和功能。同时，PercentLayout 扩展了百分比相关的特性，对应到 XML 文件中，增加了如下配置属性。

- layout_widthPercent: 用百分比来表示宽度。
- layout_heightPercent: 用百分比来表示高度。
- layout_marginPercent: 用百分比来表示 View 之间的间隔。
- layout_marginLeftPercent: 用百分比来表示左边的间隔。
- layout_marginTopPercent: 用百分比来表示顶部的间隔。
- layout_marginRightPercent: 用百分比来表示右边的间隔。
- layout_marginBottomPercent: 用百分比来表示底部的间隔。
- layout_marginStartPercent: 用百分比来表示距离第一个 View 之间的间隔。
- layout_marginEndPercent: 用百分比来表示距离最后一个 View 之间的间隔。
- layout_aspectRatio: 用百分比来表示 View 的宽高比。

下面我们以 PercentRelativeLayout 为例说明百分比布局的使用。

```
<?xml version="1.0" encoding="utf-8" ?>
<android.support.percent.PercentRelativeLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

<TextView
    android:id="@+id/top_left"
    android:layout_width="0dp"
    android:layout_height="0dp"
```

```

android:layout_alignParentTop=" true"
android:background=" @android:color/holo_blue_bright"
android:gravity=" center"
android:text=" 蓝色色块"
android:textSize=" 16dp"
app:layout_heightPercent=" 30%"
app:layout_widthPercent=" 70%" />

```

```
<TextView
```

```

    android:id=" @+id/top_right"
    android:layout_width=" 0dp"
    android:layout_height=" 0dp"
    android:layout_alignParentTop=" true"
    android:layout_toRightOf=" @+id/top_left"
    android:background=" @android:color/holo_red_light"
    android:gravity=" center"
    android:text=" 红色色块"
    android:textSize=" 16dp"
    app:layout_heightPercent=" 30%"
    app:layout_widthPercent=" 30%" />

```

```
<TextView
```

```

    android:id=" @+id/bottom"
    android:layout_width=" match_parent"
    android:layout_height=" 0dp"
    android:layout_below=" @+id/top_left"
    android:background=" @android:color/holo_green_light"
    android:gravity=" center"
    android:text=" 绿色色块"
    android:textSize=" 16dp"
    app:layout_heightPercent=" 70%" />

```

```
</android.support.percent.PercentRelativeLayout>
```

对应的界面如图 5-1 左侧图所示。使用百分比布局，我们可以不指定 `android:layout_width`

和 `android:layout_height` 的取值,但这样的话,Android Studio 会提示没有设置这两个值,导致出现标红警告,所以一般情况下,我们可以指定这两个的取值为 `0dp`。由于新增加的属性并不是位于 Android 命名控件之内的,因此需要使用 `app:` 作为前缀引用。

Percent 库新增加的属性中有一个比较特殊: `layout_aspectRatio`,它使得开发者可以单独指定 View 的宽或者高,然后使用 `app:layout_aspectratio` 属性设置宽高比,从而实现 View 的宽和高按照百分比来设置,代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.percent.PercentRelativeLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/top_left"
        android:layout_width="200dp"
        android:layout_height="0dp"
        android:layout_alignParentTop="true"
        android:background="@android:color/holo_blue_bright"
        android:gravity="center"
        android:text="蓝色色块"
        android:textSize="16dp"
        app:layout_aspectRatio="200%" />

    <TextView
        android:id="@+id/top_right"
        android:layout_width="100dp"
        android:layout_height="0dp"
        android:layout_below="@id/top_left"
        android:layout_marginTop="20dp"
        android:background="@android:color/holo_red_light"
        android:gravity="center"
```



```
android:text=" 红色色块"  
android:textSize=" 16dp"  
app:layout_aspectRatio=" 50%" />
```

```
</android.support.percent.PercentRelativeLayout>
```

实现效果如图 5-1 右侧图所示。

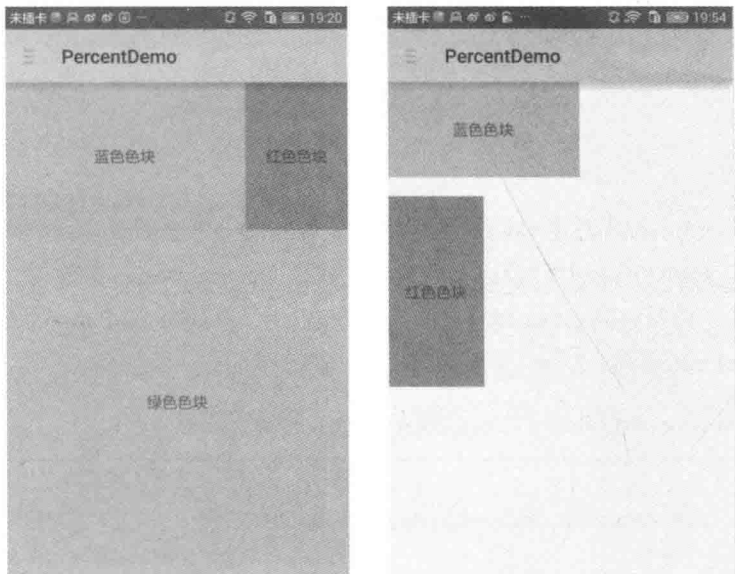


图5-1

android-percent-support-lib-sample¹ 这个百分比开源示例中，增加了一个 PercentLinearLayout 类，实现了 LinearLayout 的百分比布局，也就是说，Android 中主流的三大布局都支持百分比。

1 <https://github.com/JulienGenoud/android-percent-support-lib-sample>

第6章

Design Support Library 使用详解

Design Support Library¹ 是在 Google I/O 2015² 上发布的一个全新兼容函数库，它使得开发者可以在 Android 2.1（API Level 7）及以上的设备中实现 Material Design 的效果，这个函数库提供了一系列的控件，主要包括：Snackbar、Navigation View、FloatActionButton、CoordinatorLayout、CollapsingToolbarLayout 等。

在使用 Design Support Library 之前，首先需要添加如下依赖。

```
dependencies {  
    compile 'com.android.support:design:23.1.1'  
}
```

6.1 Snackbar

Snackbar 是带有动画效果的快速提示栏，它显示在屏幕的底部，是用来替代 Toast 的一个全新控件，它基本上继承了 Toast 的属性和方法，和 Toast 最大的不同是 Snackbar 可以带有按钮。当 Snackbar 显示时，用户可以点击按钮执行对应的操作。Snackbar 支持滑动消失，如果用户没有作任何操作，Snackbar 在到达指定时间之后就会自动消失。Snackbar 的使用很简单，语句如下。

```
Snackbar.make(view, "Here's a Snackbar", Snackbar.LENGTH_LONG)  
    .setAction("Action", null)  
    .show();
```

¹ <http://android-developers.blogspot.com/es/2015/05/android-design-support-library.html>

² <https://events.google.com/io2015/>

显示如图 6-1 所示。

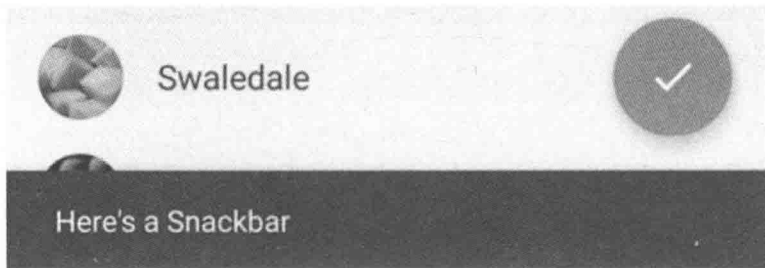


图6-1

6.2 TextInputLayout

TextInputLayout 的主要作用是作为 EditText 的容器,从而为 EditText 默认生成一个浮动的 Label,当用户点击 EditText 之后,EditText 中设置的 hint 字符串会自动移动到 EditText 的左上角。TextInputLayout 的使用很简单,语句如下。将它作为 EditText 的父容器即可。

```
<android.support.design.widget.TextInputLayout
    android:id="@+id/text_input_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    app:errorEnabled="true"
    app:errorTextAppearance="@style/ErrorText" >

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:imeOptions="actionGo"
        android:inputType="text"
        android:singleLine="true" />

</android.support.design.widget.TextInputLayout>
```

EditText 未获得焦点、获得焦点以及输入出错提示分别如图 6-2 所示。

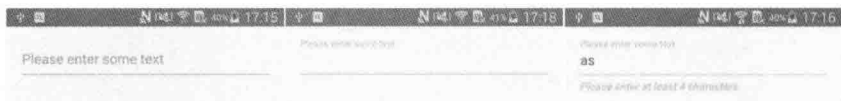


图6-2

6.3 TabLayout

TabLayout 控件用于在应用中轻松地添加 Tab 分组功能，总共有两种类型可供选择。

- 固定 Tabs：对应 xml 配置中的 `app:tabMode="fixed"`。
- 可滑动的 Tabs：对应 xml 配置中的 `app:tabMode="scrollable"`。

分别如图 6-3 所示。



图6-3

布局文件中只需添加如下 TabLayout 声明。

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

```
<android.support.v4.view.ViewPager
    android:id="@+id/viewpager"
    android:paddingTop="45dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

```

<android.support.design.widget.TabLayout
    android:id="@+id/tab_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:tabMode="fixed"
    app:tabGravity="fill" />

```

```

</RelativeLayout>

```

接着在代码中将 TabLayout 和 ViewPager 关联起来。

```

tabLayout.setupWithViewPager(mViewPager);

```

6.4 NavigationView

自从 Material Design 发布后,我们知道如何设计一个符合标准的导航抽屉¹。在开发中遵循这些设计准则相当费时,不过现在有了导航视图,实现起来就简单多了。实现效果如图 6-4 所示。

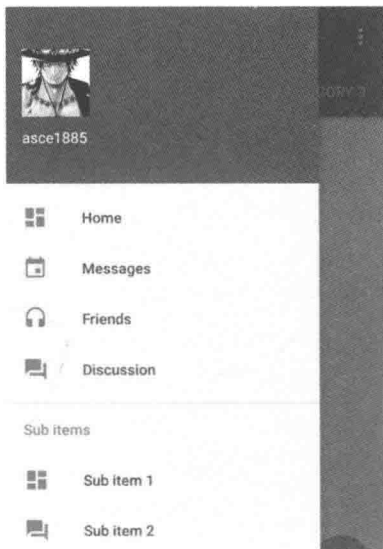


图 6-4

¹ <http://www.google.com/design/spec/patterns/navigation-drawer.html>

使用导航视图需要传入一组参数，一个可选的头部布局，以及一个用于构建导航选项的菜单。完成上面这些步骤之后，就只需要给导航选项添加响应事件的监听器就可以了。

6.4.1 导航菜单

首先我们来创建菜单 draw_view.xml，菜单布局中使用 Group 来指定菜单组，这个组包含四个子菜单，创建一个组时一般会指定同一时间只有一个菜单项可以被选中，代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- 指定单选 -->
    <group android:checkableBehavior="single" >
        <item
            android:id="@+id/nav_home"
            android:icon="@drawable/ic_dashboard"
            android:title="Home" />
        <item
            android:id="@+id/nav_messages"
            android:icon="@drawable/ic_event"
            android:title="Messages" />
        <item
            android:id="@+id/nav_friends"
            android:icon="@drawable/ic_headset"
            android:title="Friends" />
        <item
            android:id="@+id/nav_discussion"
            android:icon="@drawable/ic_forum"
            android:title="Discussion" />
    </group>
    <item android:title="Sub items" >
        <menu>
            <item
                android:icon="@drawable/ic_dashboard"
                android:title="Sub item 1" />
```

```

        <item
            android:icon="@drawable/ic_forum"
            android:title=" Sub item 2" />
    </menu>
</item>
</menu>

```

布局中还可以看到，通过使用一个子菜单（Sub items）作为 item，我们也可以添加包含头部的菜单项。这将会创建一个分割线和一个头部，紧跟着一个 item。

6.4.2 导航头部

导航视图的头部可以根据实际需求定制，我们的例子中头部 nav_header.xml 包含一个用户头像和用户名，语句如下。

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="192dp"
    android:background="?attr/colorPrimaryDark"
    android:paddingTop="30dp"
    android:paddingLeft="16dp"
    android:theme="@style/ThemeOverlay.AppCompat.Dark"
    android:orientation="vertical"
    android:gravity="center|left" >

    <ImageView
        android:id="@+id/avatar"
        android:layout_width="64dp"
        android:layout_height="64dp"
        android:scaleType="centerCrop"
        android:src="@drawable/ic_head" />

    <TextView
        android:layout_width="match_parent"

```

```

        android:layout_height=" wrap_content"
        android:layout_marginTop=" 10dp"
        android:text=" ascel885"
        android:textAppearance=" @style/TextAppearance.AppCompat.Body1" />

```

```
</LinearLayout>
```

接着我们可以给 Activity 布局添加导航视图，同时设置菜单选项和头部布局。这里我不会详细介绍头部，因为它可以是任何你想要的布局。该 Activity 的布局如下。

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout xmlns:android=" http://schemas.
android.com/apk/res/android"
    xmlns:app=" http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_height=" match_parent"
    android:layout_width=" match_parent"
    android:fitsSystemWindows=" true" >

    <include layout="@layout/include_list_viewpager" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_height=" match_parent"
        android:layout_width=" wrap_content"
        android:layout_gravity=" start"
        android:fitsSystemWindows=" true"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/drawer_view" />

</android.support.v4.widget.DrawerLayout>

```

最后就是在 Activity 的 onCreate 函数中添加 Java 代码，首先我们需要给左上角图标的左边加上一个返回的图标。

```

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

```



```
final ActionBar ab = getSupportActionBar();
ab.setHomeAsUpIndicator(R.drawable.ic_menu);
ab.setDisplayHomeAsUpEnabled(true);
```

接着初始化导航抽屉，当导航选项被选中时，将会显示一个 Snackbar，并置选中的菜单项为选中态，同时关闭抽屉。

```
mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);

NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
if (navigationView != null) {
    navigationView.setNavigationItemSelectedListener(
        new NavigationView.OnNavigationItemSelectedListener() {
            @Override
            public boolean onNavigationItemSelected(MenuItem menuItem) {
                Snackbar.make(content, menuItem.getTitle() + " pressed",
Snackbar.LENGTH_LONG).show();
                menuItem.setChecked(true);
                mDrawerLayout.closeDrawers();
                return true;
            }
        });
}
```

最后，当菜单按钮被点击时，打开抽屉。

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            mDrawerLayout.openDrawer(GravityCompat.START);
            return true;
    }
    return super.onOptionsItemSelected(item);
}
```

由于引入了 Design Support Library 和导航视图 (Navigation View), 现在要创建一个符合材料设计标准的导航抽屉是轻而易举的工作。下面我们继续学习一个新的组件: 浮动操作按钮。

6.5 FloatingActionButton

浮动操作按钮 (FAB) 是在 Material Design 准则¹中引入的新组件, 用于强调当前屏幕最重要的一些操作, 它以大胆时尚的方式吸引用户的注意, 如图 6-5 所示。虽然很多人在鼓吹这个组件有多好, 但我们最好在必要时再使用它, 而且它提供的应该是高频操作 (最好在 Activity 中需要一直显示)。

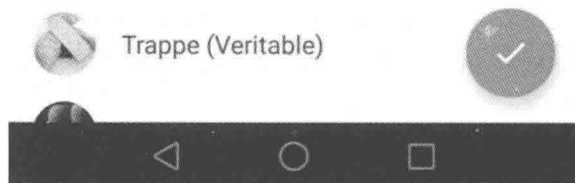


图 6-5

6.5.1 使用浮动操作按钮

FAB 的使用很简单, 首先, 你需要做的唯一的事情就是使用 FloatingActionButton, 它继承自 ImageView, 你之前所掌握的关于 ImageView 的知识点有助于你理解 FAB。你可以按如下语句在 XML 文件中添加 FAB。

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end|bottom"
    android:layout_margin="@dimen/fab_margin"
    app:elevation="6dp"
    app:pressedTranslationZ="12dp"
    android:src="@drawable/ic_done" />
```

¹ <http://www.google.com/design/spec/components/buttons-floating-action-button.html>

如你所见，我们只添加了一个图标、一个 FAB 未按压状态（elevation）的阴影（默认是 6dp）以及一个 FAB 按压状态（pressedTranslationZ）的阴影（默认是 12dp）。同时我也设置了 FAB 按钮位于屏幕右下角，当然这是假设 FAB 处于一个 FrameLayout 容器中，当它处于其他容器中时，位置可以再进行适配。

6.5.2 其他选项

上一节说到的是 FAB 自定义属性中最简单的几种，默认情况下，FAB 会采用主题中定义的 accentColor 作为背景颜色，同时采用 colorControlHighlight 作为波纹颜色（rippleColor），而这两个属性都是可以自定义的。

自定义背景颜色如下。

```
app:backgroundTint="@color/mycolor"
```

自定义波纹颜色如下。

```
app:rippleColor="@android:color/white"
```

你也可以在代码中进行设置，不过设置 backgroundTint 稍微困难一点，因为它使用了颜色状态列表（StateList），需要按如下语句进行设置。

```
fab.setBackgroundTintList(ColorStateList.valueOf("Your color"));
```

FAB 有两个可选的尺寸，通过添加如下属性可以将 FAB 设置为迷你版本。

```
app:fabSize="mini"
```

效果如图 6-6 所示。

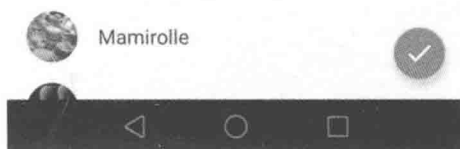


图6-6

6.5.3 点击事件

我们也可以给 FAB 添加一个点击响应事件。

```
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Here's a Snackbar", Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
    }
});
```

6.6 CoordinatorLayout

CoordinatorLayout 是 Design Support Library 引入的一个功能强大的布局，本质上是一个增强型的 FrameLayout，它可以使得不同视图组件直接相互作用，并协调动画效果。CoordinatorLayout 是基于定义在 Behaviors 中的规则集的，我们可以定义 CoordinatorLayout 内部的视图组件是如何相互作用并发生变化的。例如在上面我们使用 FloatingActionButton 和 Snackbar，为了实现在 Snackbar 出现时 FAB 能够自动往上移动，Snackbar 消失时 FAB 自动往下移动，我们就需要将 CoordinatorLayout 作为 FAB 的父容器，代码如下。

```
<android.support.design.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/main_content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true" >

    <android.support.design.widget.FloatingActionButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/fab_margin"
        android:clickable="true"
        android:src="@drawable/ic_discuss"
        app:layout_anchor="@id/appbar"
        app:layout_anchorGravity="bottom|right|end" />
```

```
</android.support.design.widget.CoordinatorLayout>
```

同时需要在代码实现 Snackbar 时将 CoordinatorLayout 实例作为 View 参数传递给 Snackbar, 语句如下。

```
Snackbar.make(mCoordinator, "Here's a Snackbar", Snackbar.LENGTH_SHORT).  
show();
```

6.7 CollapsingToolbarLayout

CollapsingToolbarLayout 控件可以实现当屏幕内容滚动时收缩 Toolbar 的效果, 通常和 AppBarLayout 配合使用, 代码如下。

```
<android.support.design.widget.AppBarLayout  
    android:id="@+id/appbar"  
    android:layout_width="match_parent"  
    android:layout_height="@dimen/detail_backdrop_height"  
    android:fitsSystemWindows="true"  
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >  
  
    <android.support.design.widget.CollapsingToolbarLayout  
        android:id="@+id/collapsing_toolbar"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:fitsSystemWindows="true"  
        app:contentScrim="?attr/colorPrimary"  
        app:expandedTitleMarginEnd="64dp"  
        app:expandedTitleMarginStart="48dp"  
        app:layout_scrollFlags="scroll|exitUntilCollapsed" >  
  
        <ImageView  
            android:id="@+id/backdrop"  
            android:layout_width="match_parent"
```

```

        android:layout_height="match_parent"
        android:fitsSystemWindows="true"
        android:scaleType="centerCrop"
        app:layout_collapseMode="parallax" />

<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_collapseMode="pin"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

</android.support.design.widget.CollapsingToolbarLayout>
</android.support.design.widget.AppBarLayout>

```

其中 `ImageView` 的 `app:layout_collapseMode` 属性是 `CollapsingToolbarLayout` 提供的，有两个取值。

- Pin: 当 `CollapsingToolbarLayout` 完全收缩后，`ToolBar` 还可以保留在屏幕上。
- parallax: 在内容滚动时，`CollapsingToolbarLayout` 中的 `View` 也可以同时滚动，并实现视差滚动效果，视差因子通过 `app:layout_collapseParallaxMultiplier` 属性进行设置。上面代码实现的滚动前和滚动后效果如图6-7所示。

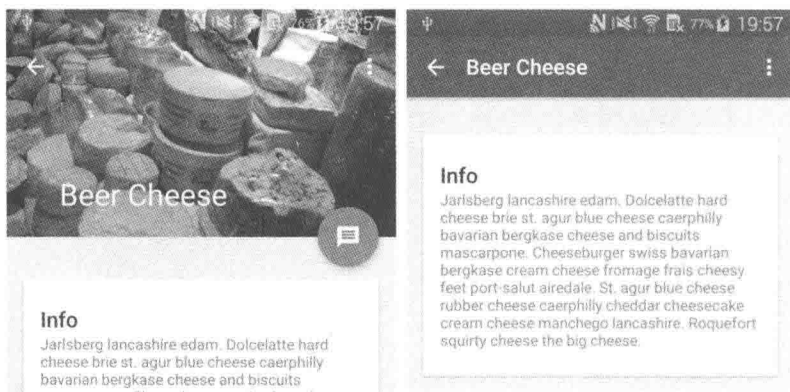


图6-7

6.8 BottomSheetBehavior

BottomSheetBehavior 控件是在 Android Support Library 23.2 中引入的, 它可以轻松地实现底部动作条功能, 底部动作条的引入需要在布局中添加 `app:layout_behavior` 属性, 并将这个布局作为 CoordinatorLayout 的子 View, 语句如下。

```
<android.support.design.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true" >
...
    <LinearLayout
        android:id="@+id/design_bottom_sheet"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_green_light"
        android:orientation="vertical"
        android:paddingTop="8dp"
        app:behavior_peekHeight="32dp"
        app:layout_behavior="android.support.design.widget.
BottomSheetBehavior" >
        ...
    </LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

开发者还可以通过设置 BottomSheetCallback 回调接口实现状态变化的监听, 语句如下。

```
View bottomSheet = findViewById(R.id.design_bottom_sheet);
BottomSheetBehavior behavior = BottomSheetBehavior.from(bottomSheet);
behavior.setBottomSheetCallback(new BottomSheetBehavior.
BottomSheetCallback() {
    @Override
```

```
public void onSlide(@NonNull View bottomSheet, float slideOffset) {  
    // 拖动动作  
    Log.d(TAG, "onSlide: " + slideOffset);  
}  
  
@Override  
public void onStateChanged(@NonNull View bottomSheet, int newState) {  
    // 状态变化  
    Log.d(TAG, "onStateChanged: " + newState);  
}  
});
```

第7章

Android Studio中的 NDK开发

本章不会介绍如何使用 JNI/NDK 一步一步进行 Android 开发，而是尝试理清在 Android Studio 中结合 NDK 进行开发的一些重要知识点。

- ABI 的基本概念。
- 在 Android Studio 中使用 C++ 代码的两种方式。
- 在 Gradle 中添加原生 .so 文件依赖的方式。
- 使用 .so 文件时一些注意事项。

7.1 ABI的基本概念

早期的 Android 系统几乎只支持 ARMv5 的 CPU 架构，而发展到现在，Android 系统目前支持以下 7 种不同的 CPU 架构。

- ARMv5
- ARMv7 (从2010年起)
- x86 (从2011年起)
- MIPS (从2012年起)
- ARMv8
- MIPS6
- x86_64 (从2014年起)

每一种架构关联着一种 ABI，那么什么是 ABI 呢？ABI 是 Application Binary Interface 的缩写，即应用程序二进制接口，它定义了二进制文件（Android 平台上专指 .so 文件）如何运行在相应的系统平台上，包括使用的指令集、内存对齐到可用的系统函数库。在 Android 系统上，每一个 CPU 架构对应一个 ABI，如前所述，总共有 7 种，对应到 Android Studio 中的目录结构如下。

- [module_name]
- --[src]
- ----[main]
- -----[jniLibs]
- -----[armeabi]
- -----[armeabi-v7a]
- -----[x86]
- -----[mips]
- -----[arm64-v8a]
- -----[x86_64]
- -----[mips64]

很多设备都支持多于一种的 ABI。例如 ARM64 和 x86 设备也可以同时运行 armeabi-v7a 和 armeabi 的二进制包。但最好是针对特定平台提供相应平台的二进制包，这种情况下运行时就少了一个模拟层（例如 x86 设备上模拟 arm 的虚拟层），从而得到更好的性能（例如能够利用最近的架构更新，例如硬件 fpu、更多的寄存器、更好的向量化等）。

我们可以通过 Build.SUPPORTED_ABIS 得到根据偏好排序的设备支持的 ABI 列表。但你不应该从你的应用程序中读取它，因为如果在对应的 lib/ABI 目录中存在 .so 文件的话，Android 包管理器安装 APK 时，会自动选择 APK 包中为对应系统 ABI 预编译好的 .so 文件。

在 Android Studio 中为工程添加 C++ 代码的方式有两种。

- 引入预编译的二进制库（自己编译好或者第三方提供的）。
- 在 Android Studio 中直接从 C/C++ 源码编译。

7.2 引入预编译的二进制 C/C++ 函数库

添加预编译的二进制库很简单，默认情况下，Android Studio 会到 jniLibs 目录中查找并拷贝所有的二进制库。假设我们在上面每个 ABI 目录中都有一个名为 libhellondk.so 的二进制库，在 Android 中要使用这个库很简单。

```
String libName = "hellondk";  
System.loadLibrary(libName);
```

注意，使用 NDK 编译名为 MyModuleName 的静态库时，生成的文件名如下。

```
libMyModuleName.so
```

因此，在 Java 中使用 loadLibrary 加载 libhellondk.so 这个库时，需要把头部的 lib 和尾部的后缀 .so 去掉。

7.3 直接从 C/C++ 源码编译

Android Studio 中对 C/C++ 源码编译成 .so 文件的步骤主要如下。

- 配置 ndk.dir 变量。
- 在 Gradle 中配置 NDK 模块。
- 添加 C / C++ 文件到指定目录。

7.3.1 配置 ndk.dir 变量

进行 NDK 开发第一件要做的事情就是打开工程根目录的 local.properties 文件，并在其中配置 ndk 的目录，以便让 Android Studio 能够找到 NDK 的可执行文件等。

```
sdk.dir=/Users/guhaoxin/Library/Android/sdk  
ndk.dir=/Users/guhaoxin/Library/Android/ndk
```

7.3.2 在 Gradle 中配置 NDK 模块

为了让 Gradle 对 C/C++ 代码进行编译，需要配置 module 的 build.gradle 文件，打开 build.

gradle，在其中 defaultConfig 段落中添加如下代码。

```
ndk {
    moduleName "moduleName"
}
```

其中，moduleName 要替换成我们自己的 C/C++ 模块名，例如我的某个 NDK 项目中配置文件内容如下。

```
android {
    compileSdkVersion 20
    buildToolsVersion "22.0.1"

    defaultConfig {
        minSdkVersion 9
        targetSdkVersion 20
        versionCode 1
        versionName "1.0"

        ndk {
            moduleName "HXEngine"
        }
    }
    ...
}
```

接下来，可以配置其他的 NDK 选项，例如 cFlags、stl 和 ldLibs 等。

```
ndk {
    moduleName "HXEngine"
    cFlags "-DANDROID_NDK -D_DEBUG DNULL=0"
    ldLibs "EGL", "GLSv3", "dl", "log"
    stl "stlport_shared"
}
```

7.3.3 添加 C/C++ 文件到指定的目录

默认情况下，Gradle 会到

```
[module]/src/main/jni/
```

目录中查找 C/C++ 文件进行编译，我们只需把 C/C++ 文件放到这个目录中即可。当然我们也可以修改 jni 的目录，例如希望把 C/C++ 文件放到 source 目录中，可以修改 gradle 文件如下。

```
android {  
  
    // ..... 其他设置 .....  
  
    sourceSets.main {  
        jni.srcDirs 'src/main/source'  
    }  
}
```

7.4 使用 .so 文件的注意事项

处理 .so 文件时有一条简单却并不知名的重要的法则：你应该尽可能地提供专为每个 ABI 优化过的 .so 文件，要么全部支持，要么都不支持。我们不应该混合着使用，而应该为每个 ABI 目录提供对应的 .so 文件。

当一个应用安装在设备上，只有该设备支持的 CPU 架构对应的 .so 文件会被安装。在 x86 设备上，libs/x86 目录中如果存在 .so 文件的话，会被安装，如果不存在，则会选择 armeabi-v7a 中的 .so 文件，如果也不存在，则选择 armeabi 目录中的 .so 文件（因为 x86 设备也支持 armeabi-v7a 和 armeabi）。

使用 .so 文件的注意事项主要如下。

7.4.1 使用高平台版本编译的 .so 文件运行在低版本的设备上

使用 NDK 时，你可能会倾向于使用最新的编译平台，但事实上这是错误的，因为 NDK 平

台不是后向兼容的，而是前向兼容的。推荐使用 APP 的 `minSdkVersion` 对应的编译平台。因此，当我们引入一个预编译好的 `.so` 文件时，首先需要检查它被编译所用的平台版本。

7.4.2 混合使用不同的C++ 运行时编译的 .so 文件

`.so` 文件可以依赖于不同的 C++ 运行时，静态编译或者动态加载。混合使用不同版本的 C++ 运行时可能导致很多奇怪的 Crash。作为一个经验法则，当只有一个 `.so` 文件时，静态编译 C++ 运行时是没问题的，但存在多个 `.so` 文件时，应该让所有的 `.so` 文件都动态链接相同的 C++ 运行时。

7.4.3 没有为每个支持的 CPU 架构提供对应的 .so 文件

这一点在前文已经说到了，但你应该真的特别注意它，因为它可能发生在根本没有意识到的情况下。例如：你的 APP 支持 `armeabi-v7a` 和 `x86` 架构，然后使用 Android Studio 新增了一个函数库依赖，这个函数库包含 `.so` 文件并支持更多的 CPU 架构，例如新增 `android-gif-drawable` 函数库。

```
compile 'pl.droidsonroids.gif:android-gif-drawable:1.1.+'

```

发布我们的 APP 后，会发现它在某些设备上会发生 Crash，例如 Galaxy S6，最终可以发现只有 64 位目录下的 `.so` 文件被安装进手机。解决的办法是重新编译我们的 `.so` 文件使其支持缺失的 ABIs，或者设置

```
ndk.abiFilters

```

显式地指定支持的 ABIs。

7.4.4 将 .so 文件放在错误的地方

我们往往很容易对 `.so` 文件应该放在或者生成到哪里感到困惑，下面是一个总结。

- Android Studio 工程放在 `jniLibs/ABI` 目录中（当然也可以通过在 `build.gradle` 文件中的设置 `jniLibs.srcDir` 属性自己指定）。
- Eclipse 工程放在 `libs/ABI` 目录中（这也是 `ndk-build` 命令默认生成 `.so` 文件的目录）。

- AAR 压缩包中位于 jni/ABI 目录中（.so 文件会自动包含到引用 AAR 压缩包的 APK 中）。
- 最终 APK 文件中的 lib/ABI 目录中。
- 通过 PackageManager 安装后，在小于 Android 5.0 的系统中，.so 文件位于 APP 的 nativeLibraryPath 目录中；在大于等于 Android 5.0 的系统中，.so 文件位于 APP 的 nativeLibraryRootDir/CPU_ARCH 目录中。

7.4.5 只提供 armeabi 架构的 .so 文件而忽略其他 ABIs 的

所有的 x86/x86_64/armeabi-v7a/arm64-v8a 设备都支持 armeabi 架构的 .so 文件，因此似乎移除其他 ABIs 的 .so 文件是一个减少 APK 大小的好技巧。但事实上并不是：这将影响到函数库的性能和兼容性。

x86 设备能够很好的运行 ARM 类型函数库，但并不保证 100% 不发生 crash，特别是对旧设备。64 位设备（arm64-v8a、x86_64、mips64）能够运行 32 位的函数库，但是以 32 位模式运行，在 64 位平台上运行 32 位版本的 ART 和 Android 组件，将丢失专为 64 位优化过的性能（ART、webview、media 等）。

以减少 APK 包大小为由是一个错误的借口，因为你也可以选择在应用市场上传指定 ABI 版本的 APK，生成不同 ABI 版本的 APK 可以在 build.gradle 中进行如下配置。

```
android {
    ...
    splits {
        abi {
            enable true
            reset()
            include 'x86', 'x86_64', 'armeabi-v7a', 'arm64-v8a'
        }
    }
    //select ABIs to build APKs for
    universalApk true //generate an additional APK that contains all
    the ABIs
}

// map for the version code
```

```
project.ext.versionCodes = [ 'armeabi' : 1, 'armeabi-v7a' : 2, 'arm64-  
v8a' : 3, 'mips' : 5, 'mips64' : 6, 'x86' : 8, 'x86_64' : 9]  
  
android.applicationVariants.all { variant ->  
    // assign different version code for each output  
    variant.outputs.each { output ->  
        output.versionCodeOverride =  
            project.ext.versionCodes.get(output.getFilter(com.  
android.build.OutputFile.ABI), 0) * 1000000 + android.defaultConfig.  
versionCode  
        }  
    }  
}
```

第8章

Gradle 必知必会

Gradle 是 Android Studio 标配的构建系统，想要熟练使用 Android Studio 进行 Android 开发，必须对 Gradle 有一定程度的认知。相信经过一两年的实践，大家应该都能掌握 Gradle 的基本用法，例如 `settings.gradle`、Project 的 `build.gradle` 和 Module 的 `build.gradle` 等概念以及之间的区别，同时也了解 Gradle 文件的基本结构。本章不会继续讲解这些基础的东西，如果对 Gradle 还不熟悉的话，可以参考《Android Gradle 插件中文指南》¹*Android Plugin DSL Reference*² 和 *Gradle User Guide*³ 这三个文档，本章接下来将要介绍的是一些有用的实例，以补充官方文档。

8.1 共享变量的定义

Gradle 开发中经常会涉及到很多相同的配置，例如不同 module 中都要配置 `compileSdkVersion`、`buildToolsVersion` 等变量值，我们把这些公共的配置项称为共享变量。一般情况下，它们的取值都应该保持一致，如果让每个 module 自己管理这些配置的值，则可能会导致不同 module 取值不同，也可能导致修改某个配置项时需要到每个 module 中都修改一遍。为了解决这个问题，我们定义一个名为 `common_config.gradle` 的文件，并放在工程根目录中，文件内容举例如下。

```
project.ext {  
  
    // Java语言相关  
    javaVersion = 8  
    javaMaxHeapSize = '4G'
```

¹ <http://gradle-guide.books.yourtion.com/index.html>

² <http://google.github.io/android-gradle-dsl/current/>

³ <https://docs.gradle.org/current/userguide/userguide.html>

```

// Android编译版本相关
compileSdkVersion = 23
buildToolsVersion = "23.1.1"
minSdkVersion = 16
targetSdkVersion = 23

// 混淆相关
minifyEnable = true
shrinkResEnable = minifyEnable

// JDK版本兼容
sourceCompatibility = this.&getJavaVersion()
targetCompatibility = this.&getJavaVersion()
}

def getJavaVersion() {
    switch (project.ext.javaVersion) {
        case "6" :
            return JavaVersion.VERSION_1_6
        case "7" :
            return JavaVersion.VERSION_1_7
        case "8" :
            return JavaVersion.VERSION_1_8
        default:
            return JavaVersion.VERSION_1_6
    }
}

```

工程中各个 module 的 build.gradle 文件引用全局配置项如下。

```

apply from: "${project.rootDir}/common_config.gradle"

android {
    compileSdkVersion project.ext.compileSdkVersion
    buildToolsVersion project.ext.buildToolsVersion

```

```

defaultConfig {
    minSdkVersion project.ext.minSdkVersion
    targetSdkVersion project.ext.targetSdkVersion
    versionCode 1
    versionName "1.0"
}
buildTypes {
    release {
        minifyEnabled project.ext.minifyEnable
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
    }
}

compileOptions {
    sourceCompatibility project.ext.sourceCompatibility
    targetCompatibility project.ext.targetCompatibility
}
}

```

8.2 通用配置

在多 Module 项目中，不同 Module 除了共享一些变量值之外，还会共享一些通用的配置，例如在前面的例子中，如果项目中有 8 个 Module，那么我们需要在每个 Module 的 build.gradle 文件中添加对 common_config.gradle 的引用。

```

apply from: "${project.rootDir}/common_config.gradle"

```

这显然是很繁琐的，一种常见解决方案是在工程根目录的 build.gradle 文件中配置 subprojects，语句如下。

```

subprojects {
    apply from: "${project.rootDir}/common_config.gradle"

    dependencies {
        testCompile 'junit:junit:4.12'
    }
}

```

8.3 aar 函数库的引用

aar 是在 Android Studio 中开始引入的一个全新的文件类型，它本质上是一个压缩包，里面包含了 jar 文件和 Android 相关的资源，如图 8-1 所示就是一个解压后的 aar 包。

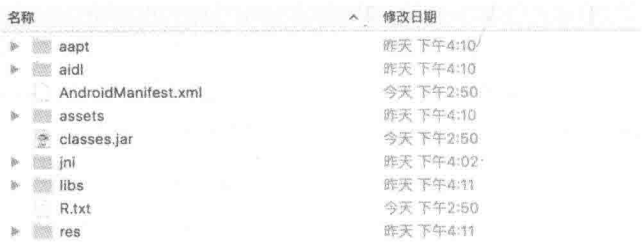


图 8-1

在 Android 工程中引用 aar 的方式和引用 jar 的方式不同，我们以图 8-2 所示工程为例进行说明，这个工程包含两个 module：app 和 thirdparty，其中 app 依赖于 thirdparty，在 thirdparty/libs 目录中存放了 react native 的依赖函数库。



图 8-2

为了在 thirdparty 中正常引用 aar 文件中的类，可以在 thirdparty/build.gradle 文件中增加如下配置。

```
android {
```

```

...

// 为了能够在工程的libs目录中找到其中的aar文件
repositories {
    flatDir {
        dirs 'libs'
    }
}

dependencies {
    ...

    // aar 文件的依赖配置
    compile(name: 'android-jsc-r174650', ext: 'aar')
    compile(name: 'drawee-0.8.1', ext: 'aar')
    compile(name: 'fbcore-0.8.1', ext: 'aar')
    compile(name: 'fresco-0.8.1', ext: 'aar')
    compile(name: 'imagepipeline-0.8.1', ext: 'aar')
    compile(name: 'imagepipeline-okhttp-0.8.1', ext: 'aar')
    compile(name: 'react-native-0.18.0', ext: 'aar')
}

```

但是如果其他模块依赖于 `thirdparty`，只有上面的配置是不够的，Android Studio 会提示找不到 aar 文件的错误，如图 8-3 所示。



图8-3

为了在其他模块使用 `thirdparty` 中提供的 API，以及引用到其中的 `aar` 文件，我们需要在项目根目录中的 `build.gradle` 文件中增加如下配置。

```
allprojects {
    repositories {
        ...
        flatDir {
            dirs '../thirdparty/libs'
        }
    }
}
```

8.4 签名和混淆的配置

APK 在发布时需要进行签名和代码混淆，在项目的 `Application Module` 中的 `build.gradle` 文件中进行少量配置即可实现自动签名和混淆，语句如下。

```
android {
    signingConfigs {

        // *** 需要替换成项目自己的值，其中 ***.keystore 文件和 build.gradle 位于
        同级目录
        myConfig {
            storeFile file( "***.keystore" )
            storePassword "***"
            keyAlias    "***"
            keyPassword "***"
        }
    }

    buildTypes {
        debug {
            // debug 版本可签名也可不签名
            signingConfig signingConfigs.myConfig
        }
    }
}
```

```
// debug版本可以不进行代码混淆，以便于调试
minifyEnabled false
    proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
'proguard-rules.pro'
}
release {
    // 签名配置
    signingConfig signingConfigs.myConfig

    // release版本一定要进行代码混淆
    minifyEnabled true
        proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
'proguard-rules.pro'
    }
}
}
```

第9章

通过Gradle打包发布函数库到JCenter和Maven Central

使用 Android Studio & Gradle 开发 Android 应用的时候，如果工程要引入一个第三方的开源函数库，比如 EventBus，那么只需要在对应模块的 build.gradle 文件中添加如下代码即可。

```
dependencies {  
    compile 'de.greenrobot:eventbus:2.4.0'  
}
```

那么这背后的原理是如何的呢？如果我们自己开发一个函数库，想要以这种方式发布并提供给第三方使用的话，需要如何操作呢？本章将讲解这其中的原理，并以实际例子进行说明。

9.1 Maven Central 和 JCenter

为了理解 Android Studio 如何在线自动获取开源库的原理，我们首先需要了解一些 Maven Repository 的概念。Maven 是 Java 开发中流行的构建工具，它的好处之一是可以减少构建应用程序时所依赖的软件构件的副本，Maven 建议的方式是将所有软件构件存储在一个叫做 Repository 的远程仓库中。经过多年的发展，目前存在两个标准的 Java & Android Maven 仓库：Maven Central 和 JCenter。

Android Studio 正是通过 Maven Repository 服务器来下载所需的函数库，服务器的地址在工程的 build.gradle 文件中定义，后面会说到。早期的 Android Studio 版本使用 Maven Central 作为默认的 Maven 仓库，但由于 Maven Central 对开发者不友好，成功上传函数库是一件很困难的事情，同时出于某种程度上的安全考虑，Android Studio 开发团队已经将默认的 Maven 仓库修

改为 JCenter。相比较 Maven Central, JCenter 具有如下优点。

- 基于 CDN 分发函数库, JCenter 提供了更快的下载速度。
- JCenter 是最大的 Java 仓库, 可以说 Maven Central 是 JCenter 的一个子集, 托管在 Maven Central 中的函数库, 几乎也都托管在 JCenter 上面。
- 上传函数库到 JCenter 上面是一件非常简单的事情, Bintray 的用户界面对用户友好。
- 如果想要同时将函数库上传到 Maven Central 上面, Bintray 网站上通过简单的点击操作就可以完成。

需要注意的一点是, 虽然 Maven Central 和 JCenter 都是标准的 android 函数库仓库, 但它们是由不同的提供商托管在不同的服务器上面的, 这两者并无关系。

9.1.1 Maven Central

托管在 sonatype¹ 上面的 Maven 仓库, 在这个地址² 可以查看上面的所有函数库。

想要在 Android Studio 的 Gradle 工程中使用 Maven Central, 需要在工程的 build.gradle 文件中添加如下配置。

```
allprojects {  
    repositories {  
        mavenCentral()  
    }  
}
```

9.1.2 JCenter

托管在 bintray³ 上面的 Maven 仓库, 在这个地址⁴ 可以查看上面的所有函数库。

想要在 Android Studio 的 Gradle 工程中使用 JCenter, 需要在工程的 build.gradle 文件中添加如下配置。

-
- 1 <https://sonatype.org/>
 - 2 <https://oss.sonatype.org/content/repositories/releases/>
 - 3 <https://bintray.com/>
 - 4 <http://jcenter.bintray.com/>

```
allprojects {
    repositories {
        jcenter()
    }
}
```

9.2 Android Studio 获取函数库的原理

前面说到在工程的 build.gradle 中配置 Maven 仓库的类型，也就是地址，那么 Gradle 使用怎样的模式匹配来下载指定的函数库呢？还是以 EventBus 为例进行说明。

```
compile 'de.greenrobot:eventbus:2.4.0'
```

一个完整的函数库依赖字符串包含三部分。

```
GROUP_ID:ARTIFACT_ID:VERSION
```

对应起来，GROUP_ID 是 de.greenrobot，ARTIFACT_ID 是 eventbus，VERSION 是 2.4.0。

- GROUP_ID: 标识函数库所属的 Group，一个组织（公司或者个人）可能会存在不同功能的函数库，一般有两种方式命名 GROUP_ID，一种是以工程的精确到组织名的包名作为前缀命名，一种是以工程的精确到函数库名的包名作为前缀命名，这两种命名方式的代表分别是 greenrobot 和 squareup。

// greenrobot方式：精确到组织名

```
dependencies {
    compile 'de.greenrobot:eventbus:2.4.0'
    compile 'de.greenrobot:greendao:2.0.0'
    compile 'de.greenrobot:greendao-generator:2.0.0'
    compile 'de.greenrobot:java-common:2.3.0'
}
```

// squareup方式：精确到函数库名

```
dependencies {
    compile 'com.squareup:otto:1.3.7' // 这个其实也是精确到组织名
    compile 'com.squareup.picasso:picasso:2.5.2'
```

```

compile 'com.squareup.okhttp:okhttp:2.4.0'
compile 'com.squareup.retrofit:retrofit:1.9.0'
}

```

- ARTIFACT_ID: 标识函数库的名字。
- VERSION: 标识函数库的版本号, 可以使用任何字符串来表示, 但建议遵循 x.y.z 格式命名, 如果是 beta 版本, 则命名为 x.y.z-beta, 语句如下。

```

dependencies {
    compile 'de.greenrobot:eventbus:3.0.0-beta1'
    provided 'de.greenrobot:eventbus-annotation-processor:3.0.0-beta1'
}

```

Gradle 会根据上面的依赖配置, 向 Maven Repository 服务器查询是否存在该版本的函数库, 如果存在, 则会根据服务器类型拼接下载请求 url 如下。

- JCenter: <http://jcenter.bintray.com/de/greenrobot/eventbus/3.0.0-beta1>
- Maven Central: <https://oss.sonatype.org/content/repositories/releases/de/greenrobot/eventbus/3.0.0-beta1/>

从上面这两个链接可以看到, Gradle 下载的函数库无非就是 jar 包或者 aar 包以及一些配置文件和签名文件, 内容如图 9-1 所示。

Index of /repositories/releases/de/greenrobot/eventbus/3.0.0-beta1

Name	Last Modified	Size	Description
Parent Directory			
eventbus-3.0.0-beta1-javadoc.jar	Tue Jun 02 20:53:54 UTC 2015	111667	
eventbus-3.0.0-beta1-javadoc.jar.asc	Tue Jun 02 20:53:52 UTC 2015	190	
eventbus-3.0.0-beta1-javadoc.jar.asc.md5	Tue Jun 02 20:53:52 UTC 2015	32	
eventbus-3.0.0-beta1-javadoc.jar.asc.sha1	Tue Jun 02 20:53:52 UTC 2015	40	
eventbus-3.0.0-beta1-javadoc.jar.md5	Tue Jun 02 20:53:54 UTC 2015	32	
eventbus-3.0.0-beta1-javadoc.jar.sha1	Tue Jun 02 20:53:55 UTC 2015	40	
eventbus-3.0.0-beta1-sources.jar	Tue Jun 02 20:53:53 UTC 2015	31496	
eventbus-3.0.0-beta1-sources.jar.asc	Tue Jun 02 20:53:52 UTC 2015	190	
eventbus-3.0.0-beta1-sources.jar.asc.md5	Tue Jun 02 20:53:52 UTC 2015	32	
eventbus-3.0.0-beta1-sources.jar.asc.sha1	Tue Jun 02 20:53:53 UTC 2015	40	
eventbus-3.0.0-beta1-sources.jar.md5	Tue Jun 02 20:53:53 UTC 2015	32	
eventbus-3.0.0-beta1-sources.jar.sha1	Tue Jun 02 20:53:53 UTC 2015	40	
eventbus-3.0.0-beta1.jar	Tue Jun 02 20:53:47 UTC 2015	47965	

图 9-1

9.3 上传函数库到 JCenter

前面提到 JCenter 可以说是 Maven Central 的超集，而且 Android Studio 默认使用 JCenter 作为 Maven 仓库，而且 JCenter 托管网站 Bintray 也支持 Maven Central 的发布，因此，本节以 JCenter 为例介绍函数库的上传和发布。

9.3.1 步骤一：在 Bintray 网站上注册一个账号

既然是在 Bintray 网站上完成我们的函数库上传工作，那么首先需要注册一个 Bintray 账号。注册完成并登录后，Bintray 默认会帮我们创建一个 Maven 仓库。

9.3.2 步骤二：创建一个 Sonatype 账号

为了发布函数库到 Maven Central，首先到 Sonatype¹ 网站上面注册一个账号，创建完成后需要获得发布许可，这可以通过在 Sonatype 网站的 JIRA 中新建一个 issue，单击完成上面的 Create 按钮，将弹出如图 9-2 所示的创建 Issue 对话框。

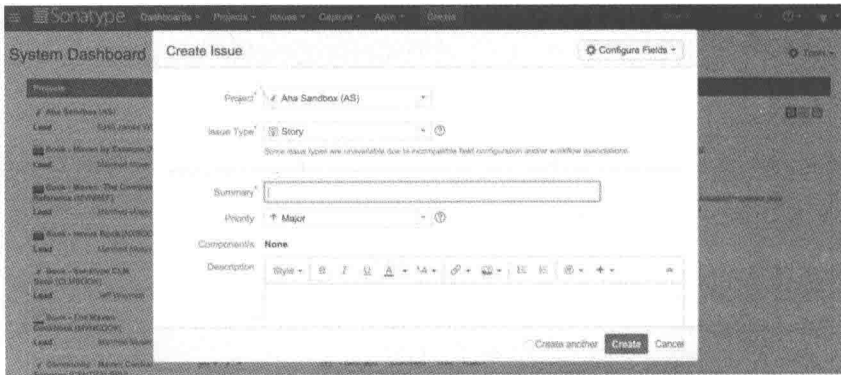


图9-2

必选项填写说明如下。

- Project: Community Support – Open Source Project Repository Hosting (OSSRH)
- Issue Type: New Project

¹ <https://issues.sonatype.org/secure/Dashboard.jspa>

- Summary: 函数库的说明, 例如 The hfjson Library
- Group Id: 设置 GROUP_ID 的根, 例如 com.asce1885, 这样在通过许可后, 以 com.asce1885 开头的函数库都可以上传到仓库中, 例如 com.asce1885.hfjson
- Project URL: 函数库工程的URL地址, 例如 <https://github.com/asce1885/hfjson>
- SCM url: 软件配置管理地址, 例如 <https://github.com/asce1885/hfjson.git>

创建完成后, 可能需要一周左右的时间才能通过审核。审核通过后, 才可以向 Maven Central 发布函数库。

最后一步是在 Bintray 网站上面设置我们的 Sonatype OSS User 的值, 打开 Edit Your Profile¹ 页面, 选择 Accounts 选项, 在其中的 Sonatype OSS User 填入我们的 Sonatype 账户名并点击 Update 按钮即可, 如图 9-3 所示。

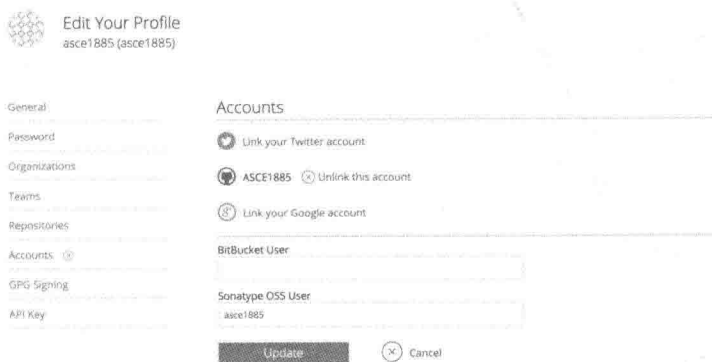


图9-3

9.3.3 步骤三：在 Bintray 网站使能自动签名

如前所述, 我们可以通过 JCenter 向 Maven Central 上传并发布函数库, 但前提是要对函数库进行签名。Bintray 为开发者提供了简单易用的 web 接口来实现对上传的函数库自动签名。

为了实现签名, 我们首先要创建签名所需的密钥, 这需要使用目前最流行、最好用的加密工具之一的 `gpg`² 来实现, 通过在 Terminal 中输入如下命令进行安装。

¹ <https://bintray.com/profile/edit>

² <https://www.gnupg.org/index.html>

```
brew install gpg
```

安装完成后，输入 `gpg --gen-key` 命令按提示生成密钥，命令行交互过程如下。

```
bogon:~ guhaoxin$ gpg --gen-key
gpg (GnuPG) 1.4.19; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

请选择您要使用的密钥种类：

- (1) RSA and RSA (default)
- (2) DSA and Elgamal
- (3) DSA (仅用于签名)
- (4) RSA (仅用于签名)

您的选择？

RSA 密钥长度应在 1024 位与 4096 位之间。

您想要用多大的密钥尺寸？(2048)

您所要求的密钥尺寸是 2048 位

请设定这把密钥的有效期限。

- 0 = 密钥永不过期
- <n> = 密钥在 n 天后过期
- <n>w = 密钥在 n 周后过期
- <n>m = 密钥在 n 月后过期
- <n>y = 密钥在 n 年后过期

密钥的有效期限是？(0)

密钥永远不会过期

以上正确吗？(y/n)yes

您需要一个用户标识来辨识您的密钥；本软件会用真实姓名、注释和电子邮件地址组合成用户标识，如下所示。

```
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"
```

真实姓名: ascel885

电子邮件地址: ascel885@gmail.com

注释: 测试使用

您正在使用 'utf-8' 字符集。

您选定了这个用户标识:

```
"asce1885 (..) <asce1885@gmail.com>"
```

更改姓名(N)、注释(C)、电子邮件地址(E)或确定(O)/退出(Q)? O

您需要一个密码来保护您的私钥。

我们需要生成大量的随机字节。这个时候您可以多做些琐事(像是敲打键盘、移动鼠标、读写硬盘之类的),这会让随机数字发生器有更好的机会获得足够的熵数。

```
.....+++++
```

```
.....+++++
```

我们需要生成大量的随机字节。这个时候您可以多做些琐事(像是敲打键盘、移动鼠标、读写硬盘之类的),这会让随机数字发生器有更好的机会获得足够的熵数。

```
..+++++
```

```
+++++
```

```
gpg: /Users/guhaoxin/.gnupg/trustdb.gpg: 建立了信任度数据库
```

```
gpg: 密钥 448906C5 被标记为绝对信任
```

公钥和私钥已经生成并经签名。

```
gpg: 正在检查信任度数据库
```

```
gpg: 需要 3 份勉强信任和 1 份完全信任, PGP 信任模型
```

```
gpg: 深度: 0 有效性: 1 已签名: 0 信任度: 0-, 0q, 0n, 0m, 0f, 1u
```

```
pub 2048R/448906C5 2016-01-19
```

```
密钥指纹 = B90B F66E 4DB8 B15C 196E 2F8F 8FC2 D18D 4489 06C5
```

```
uid asce1885 (\xe6\xb5\xe8\xaf使\xe7\xa8\x29 <asce1885@gmail.com>
```

```
sub 2048R/E221A587 2016-01-19
```

到这一步,已经生成 RSA 的公钥和私钥,以后可以通过键入 `gpg --list-keys` 查看生成的密钥信息,语句如下。

```
bogon:~ guhaoxin$ gpg --list-keys
```

```
/Users/guhaoxin/.gnupg/pubring.gpg
```

```
-----
```

```
pub 2048R/448906C5 2016-01-19
```

```
uid asce1885 (\xe6\xb5\xe8\xaf使\xe7\xa8\x29 <asce1885@
```

```
gmail.com>
```

```
sub 2048R/E221A587 2016-01-19
```

接下来需要将生成的公钥上传到密钥服务器上使其生效，这是通过如下命令实现的。

```
gpg --keyserver hkp://pool.sks-keyservers.net --send-keys PUBLIC_KEY_ID
```

其中 PUBLIC_KEY_ID 需要替换成上面生成的公钥信息，也就是 pub 2048R/ 后面的数字串：448906C5。接着输入下面的命令从服务器以 ASCII 格式导出公钥和私钥。

```
gpg -a --export yourmail@email.com > public_key_sender.asc
```

```
gpg -a --export-secret-key yourmail@email.com > private_key_sender.asc
```

记得将 yourmail@email.com 替换成前面生成密钥所使用的邮箱地址。执行成功后，生成的两个文件中分别保存了公钥和私钥信息。接着打开 Bintray 的 Edit Your Profile¹ 页面，打开其中的 GPG Signing 选项，将公钥和私钥填入对应的位置即可，最后单击 Update 按钮进行保存，如图 9-4 所示。



图 9-4

9.3.4 步骤四：生成 POM 相关的信息

Maven Central 和 JCenter 的发布都要求提供工程的 POM 信息，这是通过 android-maven-plugin 来生成。首先在工程的 build.gradle 文件中添加这个插件的依赖，语句如下。

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.
```

¹ <https://bintray.com/profile/edit>


```

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.0'
        classpath 'com.github.dcendents:android-maven-gradle-plugin:1.3'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

```

然后在函数库 Module 的 build.gradle 文件中添加一些常量定义，这些常量定义在下面的 install.gradle 和 bintray.gradle 中使用到，build.gradle 文件内容如下。

```

apply plugin: 'com.android.library' // 表明是Android函数库，如果是Java函数库，
则取值为 'java'

```

```

// Android相关配置

```

```

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"
    resourcePrefix "ascel885_"

    defaultConfig {
        minSdkVersion 14
        targetSdkVersion 19
        versionCode 4
        versionName "1.0.3"
    }
}

```

```

    }
    buildTypes {
    }
}

// 常量定义
ext {
    bintrayRepo = 'maven'
    bintrayName = 'hfjson' //发布到JCenter上的项目名字

    publishedGroupId = 'com.ascel885.hfjson' // 函数库的Maven Group ID
    libraryName = 'hfjson' // 函数库名字
    artifact = 'hfjson' // 构件的名字

    libraryDescription = 'This is a JSON Parsing and Serializing Library'
// 函数库的描述

    siteUrl = 'https://github.com/ascel885/hfjson' // 定义函数库的主页
    gitUrl = 'https://github.com/ascel885/hfjson.git' // 定义函数库的
Github仓库地址

    libraryVersion = '1.0.3' // 函数库的版本号

    developerId = 'ascel885'
    developerName = 'ascel885'
    developerEmail = 'ascel885@gmail.com'

    licenseName = 'The Apache Software License, Version 2.0'
    licenseUrl = 'http://www.apache.org/licenses/LICENSE-2.0.txt'
    allLicenses = [ "Apache-2.0" ]
}

apply from: "install.gradle"
apply from: "bintray.gradle"

```

接着在要发布的函数库的 Module 中的根目录中新建 `install.gradle` 文件，并在其中添加生成 POM 的配置，语句如下。

```

apply plugin: 'com.github.dcendents.android-maven'
// 生成JavaDoc和Source Jar需要

group = publishedGroupId

install {
    repositories.mavenInstaller {
        // 生成pom.xml文件
        pom {
            project {
                packaging 'aar' // 指定打包的格式，如果是Jar包，则取值为 'jar'
                groupId publishedGroupId
                artifactId artifact

                name libraryName
                description libraryDescription
                url siteUrl

                // 设置授权许可
                licenses {
                    license {
                        name licenseName
                        url licenseUrl
                        distribution 'repo'
                    }
                }

                // 开发者的信息
                developers {
                    developer {
                        id developerId

```

```

        name developerName
        email developerEmail
    }
}

// 设置软件配置管理
scm {
    connection gitUrl
    developerConnection gitUrl
    url siteUrl
}
}
}
}
```

完成之后，这时如果在 Terminal 中执行 `./gradlew install` 命令时，会在本地的 M2 仓库中安装上面的 aar 文件。

9.3.5 步骤五：上传函数库到 Bintray

首先在工程的 build.gradle 中添加 Bintray 官方的 Gradle 插件，语句如下。

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.
```

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.1'
        classpath 'com.jfrog.bintray.gradle:gradle-bintray-plugin:1.2'
        classpath 'com.github.dcendents:android-maven-plugin:1.2'
    }
}
```

```

    }
}

allprojects {
    repositories {
        jcenter()
    }
}

```

同时，在函数库 Module 的根目录新建 `bintray.gradle` 文件并添加生成 JavaDoc 和 Source Jars 的任务，以及上传到 JCenter 的脚本如下。

```

apply plugin: 'com.jfrog.bintray' // 上传函数库到Bintray时需要用到

// 发布函数库构件时指定的版本号
version = libraryVersion

// 生成Source Jar的任务
task sourcesJar(type: Jar) {
    from android.sourceSets.main.java.srcDirs
    classifier = 'sources'
}

// 生成JavaDoc的任务
task javadoc(type: Javadoc) {
    source = android.sourceSets.main.java.srcDirs
    classpath += project.files(android.getBootClasspath().join(File.
pathSeparator))
}

// 生成JavaDoc.jar的任务
task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
    options.encoding = 'UTF-8'
}

```

```

}

// 指定归档的构件
artifacts {
    archives javadocJar
    archives sourcesJar
}

// 上传到Bintray
Properties properties = new Properties()
properties.load(project.rootProject.file('local.properties').
newDataInputStream())

bintray {
    user = properties.getProperty("bintray.user")
    key = properties.getProperty("bintray.apikey")

    configurations = ['archives']
    pkg {
        repo = bintrayRepo
        name = bintrayName
        websiteUrl = siteUrl
        vcsUrl = gitUrl
        licenses = allLicenses
        publish = true
        publicDownloadNumbers = true
        version {
            desc = libraryDescription
            gpg {
                sign = true //是否使用GPG对文件进行签名，默认是false
                passphrase = properties.getProperty("bintray.gpg.password")
            }
        }
    }
}

```

```

    }
}

```

其中 `bintray.user` 和 `bintray.apikey` 是保存在 `local.properties` 文件中的配置，分别表示 Bintray 的用户名和 APIKey，需要注意的是，这个文件不应该上传到版本控制系统（`svn` 或者 `git`）中，以免造成密钥泄漏。打开 Edit Your Profile¹ 页面，选择 API Key 选项，即可看到对应用户名的 API Key 的值，如图 9-5 所示。

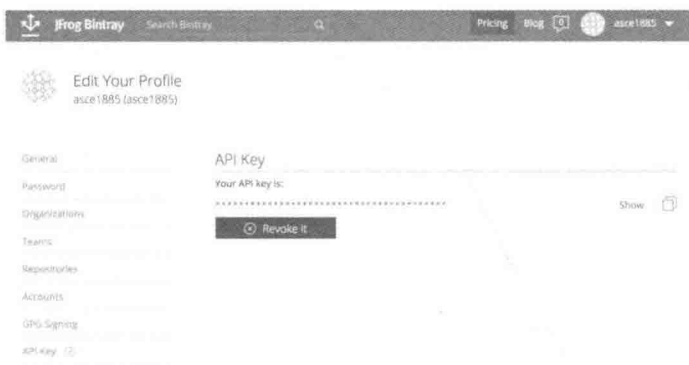


图9-5

`local.properties` 的内容如下。

```

bintray.user=asce1885
bintray.apikey=32e1fde903ac7662c9da8c05512cb5f8107a896e
bintray.gpg.password=***** // 填入你的 GPG 私钥

```

9.3.6 步骤六：发布 Bintray 用户仓库到 JCenter

完成上述配置之后，严格来说，可以分两步完成函数库的上传，首先是在 Android Studio 的命令行中执行 `./gradlew install` 命令，这一步是编译源码并生成 `aar/jar`, `pom` 等文件。如果源码编译没有问题，接下来就可以在 Android Studio 的命令行中执行 `./gradlew bintrayUpload` 命令即可完成函数库的上传。如果这个函数库之前发布过版本，这次只是升级版本号，那么到这一步就全部完成了，可以开始使用这个在线函数库了。如果是发布函数库的第一个版本，那么需要到 Bintray 网站上提交审核，在 Bintray 上进入我们刚刚上传的函数库的主页，在页面右下角找到 Add To JCenter 按钮，如图 9-6 所示。

¹ <https://bintray.com/profile/edit>



图 9-6

在点击进入的页面中填写发布说明，提交后，等待审核通过即可开始使用这个函数库，审核周期一般为 1 至 2 天。



图 9-7

顺便说一句，如何在 Bintray 上面查看函数库包含的文件呢？这可以通过如图 9-8 所示页面查看。

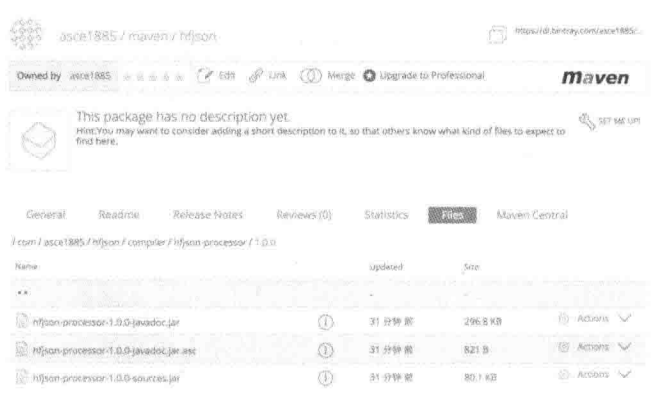


图 9-8

9.3.7 步骤七：同步函数库到 Maven Central

虽然 Android Studio 已经将默认的 Maven 仓库切换为 JCenter，但还是有一些开发者在使用 Maven Central，因此，我们还是有必要将函数库也在 Maven Central 上面同步一份的。在同步到 Maven Central 之前，首先要确保我们的函数库已经完成如下步骤。

- Add To JCenter，并通过审核。
- 在 Sonatype 网站上新建的函数库仓库通过审核。

然后点击图 9-9 所示 Maven Central 选项，并在其中填写 Sonatype 的用户名和密码进行 Sync 即可。

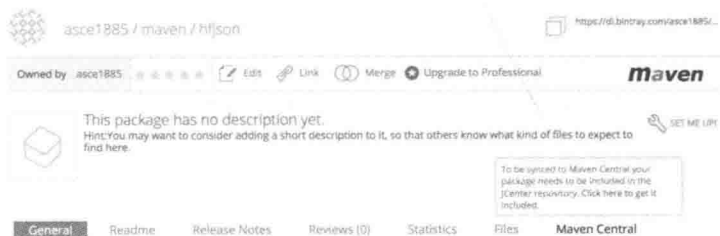


图 9-9

第10章

Builder模式详解

Builder 模式是一种广泛使用的设计模式，最初介绍见于四人帮的经典著作《设计模式：可复用面向对象软件的基础》。它的定义为：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

经过多年的实践演化，在 Android & Java 开发中广泛使用的是它的一个变种。

10.1 经典的 Builder 模式

经典的 Builder 模式主要有四个参与者。

- Product：被构造的复杂对象，ConcreteBuilder 用来创建该对象的内部表示，并定义它的装配过程。
- Builder：抽象接口，用来定义创建 Product 对象的各个组成部件的操作。
- ConcreteBuilder：Builder 接口的具体实现，可以定义多个，是实际构建 Product 对象的地方，同时会提供一个返回 Product 的接口。
- Director：Builder 接口的构造者和使用者。

接下来以代码的形式进行简单说明，首先创建 Product 类。

```
public class Product {  
  
    private String partOne;
```

```

private String partTwo;

public String getPartOne() {
    return partOne;
}

public void setPartOne(String partOne) {
    this.partOne = partOne;
}

public String getPartTwo() {
    return partTwo;
}

public void setPartTwo(String partTwo) {
    this.partTwo = partTwo;
}
}

```

接着创建 Builder 接口。

```

public interface Builder {

    public void buildPartOne();

    public void buildPartTwo();

    public Product getProduct();
}

```

然后实现 Builder 接口，创建两个 ConcreteBuilder。

```

public class ConcreteBuilderA implements Builder {

    private Product product;

```

```
@Override
public void buildPartOne() {

}

@Override
public void buildPartTwo() {

}

@Override
public Product getProduct() {
    return product;
}
}

public class ConcreteBuilderB implements Builder {

    private Product product;

    @Override
    public void buildPartOne() {

}

    @Override
    public void buildPartTwo() {

}

    @Override
    public Product getProduct() {
        return product;
    }
}
```

最后创建 Director 类。

```
public class Director {

    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void buildProduct() {
        this.builder.buildPartOne();
        this.builder.buildPartTwo();
    }

    public Product getProduct() {
        return this.builder.getProduct();
    }
}
```

10.2 Builder 模式的变种

经典的 Builder 模式重点在于抽象出对象创建的步骤，并通过调用不同的具体实现从而得到不同的结果，而变种 Builder 模式的目的在于减少对象创建过程中引入的多个重载构造函数、可选参数以及 setters 过度使用导致的不必要的复杂性。下面我们以一个简单对象 User 的创建过程为例子进行讲解，它具有如下所示属性值，且都是不可变（final）的（编程常识之一：我们应该尽量将属性值定义为不可变的）。

```
public class User {

    private final String mFirstName; // 必选
    private final String mLastName;  // 必选
    private final String mGender;    // 可选
    private final int mAge;           // 可选
    private final String mPhoneNo;   // 可选
}
```

其中, `mFirstName` 和 `mLastName` 是必选的, 其他几个属性是可选的。那么我们要如何构造这个类的实例呢? 这里有两个前提条件。

- 由于我们已经将属性值声明为`final`, 因此必须在构造函数中对这些属性进行赋值, 否则编译通不过。
- 由于属性值有必选和可选之分, 也就是说构造函数需要提供可以选择忽略可选参数的方式。

最直接的一个方案是定义多个重载的构造函数, 其中一个构造函数只接收必选参数, 一个构造函数接收所有必选参数和一个可选参数, 一个构造函数接收所有必选参数和两个可选参数, 依此类推, 代码如下。

```
public User(String firstName, String lastName) {
    this(firstName, lastName, "");
}

public User(String firstName, String lastName, String gender) {
    this(firstName, lastName, gender, 0);
}

public User(String firstName, String lastName, String gender, int age) {
    this(firstName, lastName, gender, age, "");
}

public User(String firstName, String lastName, String gender, int age, String
phoneNo) {
    mFirstName = firstName;
    mLastName = lastName;
    mGender = gender;
    mAge = age;
    mPhoneNo = phoneNo;
}
```

这种构造函数的方式虽然简单, 但只适用于只有少量几个属性的情况, 随着 `User` 类属性个数的增加, 代码将变得越来越难以阅读和维护。更严重的是对于类的使用者而言, 代码将变

得更加难以使用。

另外一种可选的方案是遵循 JavaBeans 的规范，定义一个默认的非参数构造函数，并为类的每个属性提供 getters 和 setters 函数，语句如下。

```
public class User {  
    private String mFirstName; // 必选  
    private String mLastName;  // 必选  
    private String mGender;    // 可选  
    private int mAge;          // 可选  
    private String mPhoneNo;   // 可选  
  
    public int getAge() {  
        return mAge;  
    }  
    public void setAge(int mAge) {  
        mAge = mAge;  
    }  
    public String getFirstName() {  
        return mFirstName;  
    }  
    public void setFirstName(String mFirstName) {  
        mFirstName = mFirstName;  
    }  
    public String getGender() {  
        return mGender;  
    }  
    public void setGender(String mGender) {  
        mGender = mGender;  
    }  
    public String getLastName() {  
        return mLastName;  
    }  
    public void setLastName(String mLastName) {  
        mLastName = mLastName;  
    }  
}
```

```

    }

    public String getPhoneNo() {
        return mPhoneNo;
    }

    public void setPhoneNo(String mPhoneNo) {
        mPhoneNo = mPhoneNo;
    }
}

```

第二种方案的好处是易于阅读和维护，使用者可以创建一个空实例，并只设置感兴趣的属性值，但这个方案存在两个缺点。

- User 类是可变的，从而失去了不可变对象的很多好处。
- User 类的实例状态不连续，如果你想创建一个同时具有五个属性值的类实例，那么直到第五个属性值的 set 函数被调用时，该类实例才具有完整连续的状态。这就意味着类的调用者们可能会看到该类实例的不连续状态。

那么有没有第三种方案同时具有前面两种方案的优点，同时又不存在它们的缺点呢？当然有，它就是本文的主角——变种的 Builder 模式。经过改造后的 User 类如下。

```

public class User {
    private final String mFirstName; // 必选
    private final String mLastName;  // 必选
    private final String mGender;    // 可选
    private final int mAge;           // 可选
    private final String mPhoneNo;   // 可选

    private User(UserBuilder builder) {
        mFirstName = builder.firstName;
        mLastName = builder.lastName;
        mGender = builder.gender;
        mAge = builder.age;
        mPhoneNo = builder.phoneNo;
    }

    public String getFirstName() {

```



```
        return mFirstName;
    }

    public String getLastName() {
        return mLastName;
    }

    public String getGender() {
        return mGender;
    }

    public int getAge() {
        return mAge;
    }

    public String getPhone() {
        return mPhoneNo;
    }

    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private String gender;
        private int age;
        private String phoneNo;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public UserBuilder gender(String gender) {
            this.gender = gender;
            return this;
        }

        public UserBuilder age(int age) {
            this.age = age;
        }
    }
}
```

```
        return this;
    }

    public UserBuilder phone(String phone) {
        this.phoneNo = phone;
        return this;
    }

    public User build() {
        return new User(this);
    }
}
}
```

从上面的代码可以看出以下几点。

- User 类的构造函数是私有的，这意味着调用者不能直接实例化这个类。
- User 类是不可变的，所有必选的属性值都是 final 的并且在构造函数中设置；同时，对属性值只提供 getter 函数。
- UserBuilder 使用 Fluent Interface¹惯用法，使得类的使用者代码可读性更佳。
- UserBuilder 的构造函数只接收必选的属性值作为参数，并且只有必选的属性值设置为 final，以此保证它们在构造函数中设置。

最后，来看一下 User 类实例的使用方法。

```
public User getUser() {
    return new
        User.UserBuilder( "Jack" , "Wilson" )
            .gender( "male" )
            .age(30)
            .phone( "13059679890" )
            .build();
}
```

¹ <http://martinfowler.com/bliki/FluentInterface.html>

10.3 变种 Builder 模式的自动化生成

变种 Builder 模式唯一的缺点就是需要编写很多的样板代码,我们需要在内部类 UserBuilder 中重复外部类 User 的属性定义。在 Android Studio 中,可以通过安装名为 InnerBuilder 的插件来简化 Builder 模式的创建过程,在 Plugins 面板中搜索 builder 即可找到这个插件,如图 10-1 所示。

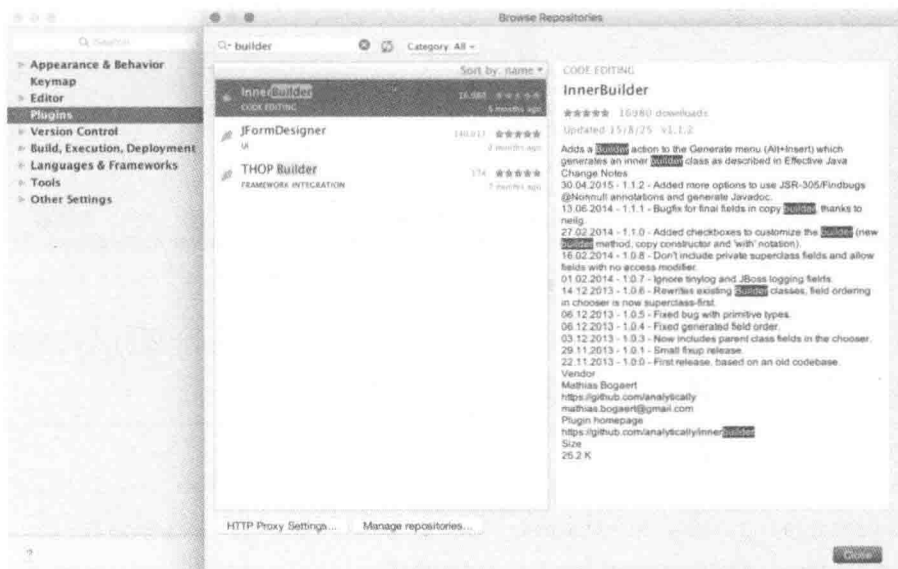


图 10-1

下载完成后,重启 Android Studio 使插件生效。编写 User 类代码时,只需要把属性名确定下来,然后单击鼠标右键,打开 Generate 菜单,选择 Builder 按钮,如图 10-2 所示。



图 10-2

在弹出的 Builder 配置对话框中进行相关配置的勾选，如图 10-3 所示。

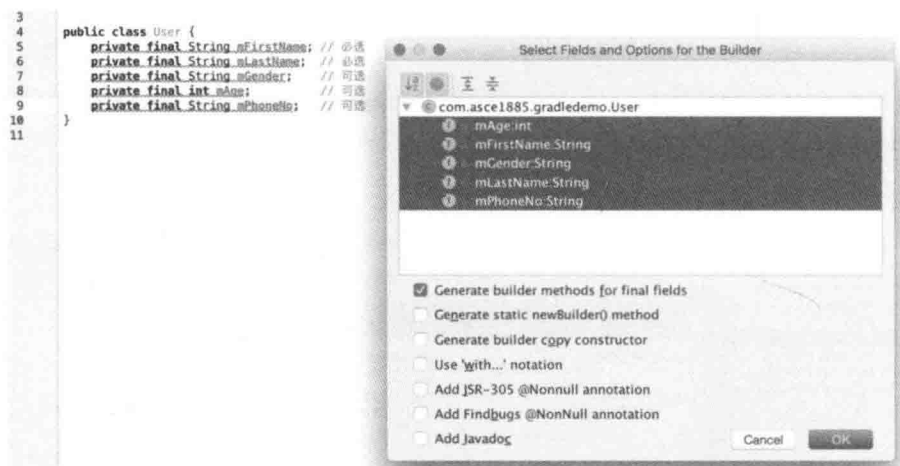


图10-3

完成后，即可自动生成 Builder 相关代码，语句如下。可以看到生成的代码和我们自己编写的代码有些许不同，根据实际需求进行少量修改即可。

```
public class User {
    private final String mFirstName; // 必选
    private final String mLastName; // 必选
    private final String mGender;    // 可选
    private final int mAge;          // 可选
    private final String mPhoneNo;   // 可选

    private User(Builder builder) {
        mAge = builder.mAge;
        mFirstName = builder.mFirstName;
        mLastName = builder.mLastName;
        mGender = builder.mGender;
        mPhoneNo = builder.mPhoneNo;
    }

    public static final class Builder {
        private int mAge;
```

```
private String mFirstName;
private String mLastName;
private String mGender;
private String mPhoneNo;

public Builder() {
}

public Builder mAge(int val) {
    mAge = val;
    return this;
}

public Builder mFirstName(String val) {
    mFirstName = val;
    return this;
}

public Builder mLastName(String val) {
    mLastName = val;
    return this;
}

public Builder mGender(String val) {
    mGender = val;
    return this;
}

public Builder mPhoneNo(String val) {
    mPhoneNo = val;
    return this;
}

public User build() {
```

```

        return new User(this);
    }
}

```

10.4 开源函数库的例子

无论在 Android SDK 还是各种开源函数库中，都经常可以看到变种 Builder 模式的应用，具体的实现方式和我们上面介绍的方案大同小异，感兴趣的可以阅读完整源码。下面我们举几个例子，大家应该可以轻易看出是应用的 Builder 模式。

Android 系统对话框 AlertDialog 的使用，语句如下。

```

AlertDialog alertDialog = new AlertDialog.Builder(this).
    setTitle(“对话框的标题”).
    setMessage(“对话框的内容”).
    setIcon(R.drawable.ic_launcher).
    create();
alertDialog.show();

```

图片缓存函数库 Android-Universal-Image-Loader 的全局配置，语句如下。

```

private ImageLoaderConfiguration(final Builder builder) {
    resources = builder.context.getResources();
    maxImageWidthForMemoryCache = builder.maxImageWidthForMemoryCache;
    maxImageHeightForMemoryCache = builder.maxImageHeightForMemoryCache;
    maxImageWidthForDiskCache = builder.maxImageWidthForDiskCache;
    maxImageHeightForDiskCache = builder.maxImageHeightForDiskCache;
    processorForDiskCache = builder.processorForDiskCache;
    taskExecutor = builder.taskExecutor;
    taskExecutorForCachedImages = builder.taskExecutorForCachedImages;
    threadPoolSize = builder.threadPoolSize;
    threadPriority = builder.threadPriority;
    tasksProcessingType = builder.tasksProcessingType;
    diskCache = builder.diskCache;
}

```

```
memoryCache = builder.memoryCache;
defaultDisplayImageOptions = builder.defaultDisplayImageOptions;
downloader = builder.downloader;
decoder = builder.decoder;

customExecutor = builder.customExecutor;
customExecutorForCachedImages = builder.customExecutorForCachedImages;

networkDeniedDownloader = new NetworkDeniedImageDownloader(downloader);
slowNetworkDownloader = new SlowNetworkImageDownloader(downloader);

L.writeDebugLogs(builder.writeLogs);
}
```

网络请求框架 OkHttp 的请求封装，语句如下。

```
private Request(Builder builder) {
    this.url = builder.url;
    this.method = builder.method;
    this.headers = builder.headers.build();
    this.body = builder.body;
    this.tag = builder.tag != null ? builder.tag : this
}
```

第11章

注解在 Android 中的应用

Android 应用开发中对注解的使用达到了淋漓尽致的地步，无论是运行时注解，还是编译时注解，亦或是标准注解，都被广泛的使用着。Android Support Library 甚至专门推出一个注解支持库 Support Annotation，各种开源函数库中也不乏注解的应用，例如著名的 REST 网络请求函数库 Retrofit 使用运行时注解，依赖注入函数库 Dagger2 使用编译时注解等。

11.1 注解的定义

注解是 Java 语言的特性之一，它是在源代码中插入的标签，这些标签在后面的编译或者运行过程中起到某种作用，每个注解都必须通过注解接口 `@interface` 进行声明，接口的方法对应着注解的元素。我们先来看看 Android 上著名的 View 注入框架 ButterKnife 的 Bind 注解的源码。

```
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.FIELD)
public @interface Bind {
    /** View ID to which the field will be bound. */
    @IdRes int[] value();
}
```

`@interface` 声明会创建一个实际的 Java 接口，与其他任何接口一样，注解也将会编译成 `.class` 文件。注解接口中的元素声明实际上是方法声明，注解接口中的方法没有参数，没有 `throws` 语句，也不能使用泛型。

11.2 标准注解

Java API 中默认定义的注解我们称之为标准注解。它们定义在 `java.lang`、`java.lang.annotation` 和 `javax.annotation` 包中。按照使用场景不同，可以分为如下三类。

11.2.1 编译相关注解

编译相关的注解是给编译器使用的，有以下几种。

- `@Override`：编译器会检查被注解的方法是否真的重载了一个来自父类的方法，如果没有，编译器将会给出错误提示。
- `@Deprecated`：可以用来修饰任何不再鼓励使用或已被弃用的属性、方法等。
- `@SuppressWarnings`：可用于除了包之外的其他声明项中，用来抑制某种类型的警告。
- `@SafeVarargs`：用于方法和构造函数，用来断言不定长参数可以安全使用。
- `@Generated`：一般是给代码生成工具使用，用来表示这段代码不是开发者手动编写的，而是工具生成的。被 `@Generated` 修饰的代码一般不建议手动修改它。
- `@FunctionalInterface`：用来修饰接口，表示对应的接口是带单个方法的函数式接口。

11.2.2 资源相关注解

资源相关注解有四个，一般用在 JavaEE 领域，Android 开发中应该不会使用到，具体如下。

- `@PostConstruct`：用在控制对象生命周期的环境中，例如 Web 容器和应用服务器，表示在构造函数之后应该立即调用被该注解修饰的方法。
- `@PreDestroy`：表示在删除一个被注入的对象之前应该立即调用被该注解修饰的方法。
- `@Resource`：用于 Web 容器的资源注入，表示单个资源。
- `@Resources`：用于 Web 容器的资源注入，表示一个资源数组。

11.2.3 元注解

元注解，顾名思义，就是用来定义和实现注解的注解，总共有如下五种。

- @Target：这个注解的取值是一个 ElementType 类型的数组，用来指定注解所适用的对象范围，总共有十种不同的类型，根据定义的注解进行灵活的组合，如表11-1所示。

表11-1

元素类型	适用于
ANNOTATION_TYPE	注解类型声明
CONSTRUCTOR	构造函数
FIELD	实例变量
LOCAL_VARIABLE	局部变量
METHOD	方法
PACKAGE	包
PARAMETER	方法参数或者构造函数的参数
TYPE	类（包含 enum）和接口（包含注解类型）
TYPE_PARAMETER	类型参数
TYPE_USE	类型的用途

同时支持多种类型的注解定义如下。

```
@Target({ElementType.TYPE, ElementType.PACKAGE})
public @interface CrashReport
```

如果一个注解的定义没有使用 @Target 修饰，那么它可以用在除了 TYPE_USE 和 TYPE_PARAMETER 之外的其他类型声明中。

- @Retention：用来指明注解的访问范围，也就是在什么级别保留注解，有如下三种选择。
 - ❑ 源码级注解：在定义注解接口时，使用 @Retention(RetentionPolicy.SOURCE) 修饰的注解，该类型的注解信息只会保留在 .java 源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的.class 文件中。
 - ❑ 编译时注解：在定义注解接口时，使用 @Retention(RetentionPolicy.CLASS) 修饰的注解，该类型的注解信息会保留在 .java 源码里和 .class 文件里，在执行的时候，会被 Java 虚拟机丢弃，不会加载到虚拟机中。
 - ❑ 运行时注解：在定义注解接口时，使用 @Retention(RetentionPolicy.RUNTIME) 修饰的注解，Java 虚拟机在运行期也保留注解信息，可以通过反射机制读取注解的信息（.java 源码、.class 文件和执行的时候都有注解的信息）。

未指定类型时，默认是 CLASS 类型。

- `@Documented`: 表示被修饰的注解应该被包含在被注解项的文档中（例如用 JavaDoc 生成的文档）。
- `@Inherited`: 表示该注解可以被子类继承的。
- `@Repeatable`: 表示这个注解可以在同一个项上面应用多次，不过这个注解是在 Java 8 中才引入的，前面四个元注解都是在 Java 5 中就已经引入的。

11.3 运行时注解

通过前面的讲解我们知道，要定义运行时注解，只需要在声明注解时指定 `@Retention(RetentionPolicy.RUNTIME)` 即可，运行时注解一般和反射机制配合使用，相比编译时注解性能比较低，但灵活性好，实现起来比较简单，本书不做详细介绍。

11.4 编译时注解

编译时注解能够自动处理 Java 源文件并生成更多的源码、配置文件、脚本或其他可能想要生成的东西。这些操作是通过注解处理器完成的。Java 编译器集成了注解处理，通过在编译期间调用 `javac -processor` 命令可以调起注解处理器，它能够允许我们实现编译时注解的功能，从而提高函数库的性能，很多知名的开源函数库都采用这种做法。

11.4.1 定义注解处理器

在编译期间，编译器会定位到 Java 源文件中的注解，注解处理器会对其感兴趣的注解进行处理，需要注意的是，一个注解处理器只能产生新的源文件，它不能修改一个已经存在的源文件。注解处理器一般通过继承 `AbstractProcessor` 类并实现 `process` 方法，同时需要指定这个处理器能够处理的注解类型以及支持的 Java 版本，语句如下。

```
public class JsonAnnotationProcessor extends AbstractProcessor {  
  
    @Override  
    public synchronized void init(ProcessingEnvironment env) {}  
}
```

```

@Override
public Set<String> getSupportedAnnotationTypes() {}

@Override
public SourceVersion getSupportedSourceVersion() {}

@Override
    public boolean process(Set<? extends TypeElement> elements,
RoundEnvironment env) {}
}

```

下面我们以 `LoganSquare`^[1] 这个函数库为例来说明上面这个方法的使用，`LoganSquare` 是一个实现了编译时注解以提高性能的 JSON 解析函数库。

- `init(ProcessingEnvironment env)`: 初始化方法会被注解处理工具调用，并传入 `ProcessingEnvironment` 类型的参数，这个参数提供包含了很多有用的工具类，例如 `Elements`、`Types`、`Filter` 等。

```

@Override
public synchronized void init(ProcessingEnvironment env) {
    super.init(env);

    mElementUtils = env.getElementUtils();
    mTypeUtils = env.getTypeUtils();
    mFiler = env.getFiler();
    mJsonObjectMap = new HashMap<>();
    mProcessors = Processor.allProcessors(processingEnv);
}

```

- `getSupportedAnnotationTypes()`: 指定这个注解处理器能够处理的注解类型，这个方法返回一个可以支持的注解类型的字符串集合。

```

@Override
public Set<String> getSupportedAnnotationTypes() {
    Set<String> supportTypes = new LinkedHashSet<>();
    for (Processor processor : mProcessors) {

```

```

        supportTypes.add(processor.getAnnotation().getCanonicalName());
    }
    return supportTypes;
}

```

- `getSupportedSourceVersion()`: 指定注解处理器使用的 Java 版本, 通常返回 `SourceVersion.latestSupported()` 即可, 当然也可以明确指定只支持某个版本的 Java, 例如 `SourceVersion.RELEASE_6`。

```

@Override
public SourceVersion getSupportedSourceVersion() {
    return SourceVersion.latestSupported();
}

```

- `process(Set<? extends TypeElement> elements, RoundEnvironment env)`: 在这个方法中实现注解处理器的具体业务逻辑, 根据输入参数 `env` 可以得到包含特定注解的被注解元素, 然后可以编写处理注解的代码最终生成所需的 Java 源文件等信息。

```

@Override
public boolean process(Set<? extends TypeElement> elements, RoundEnvironment env) {
    try {
        for (Processor processor : mProcessors) {
            processor.findAndParseObjects(env, mJsonObjectMap, mElementUtils, mTypeUtils);
        }

        for (Map.Entry<String, JsonObjectHolder> entry : mJsonObjectMap.entrySet()) {
            String fqcn = entry.getKey();
            JsonObjectHolder jsonObjectHolder = entry.getValue();

            if (!jsonObjectHolder.fileCreated) {
                jsonObjectHolder.fileCreated = true;

                try {

```

```

        JavaFileObject jfo = mFile.createSourceFile(fqcn);
        Writer writer = jfo.openWriter();
        writer.write(new ObjectMapperInjector(jsonObjectHolder).
getJavaClassFile());
        writer.flush();
        writer.close();
    } catch (IOException e) {
        error(fqcn, "Exception occurred while attempting to
write injector for type %s. Exception message: %s", fqcn, e.getMessage());
    }
}

return true;
} catch (Throwable e) {
    StringWriter stackTrace = new StringWriter();
    e.printStackTrace(new PrintWriter(stackTrace));
    error("Exception while processing Json classes. Stack trace
incoming:\n%s", stackTrace.toString());
    return false;
}
}

private void error(String message, Object... args) {
    processingEnv.getMessager().printMessage(ERROR, String.format(message,
args));
}

```

从 Java 7 开始, 我们也可以使用注解来代替上面的 `getSupportedAnnotationTypes()` 和 `getSupportedSourceVersion()` 方法, 语句如下。

```

@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes({
    // 该注解处理器支持的所有注解全名字符串类型的集合
})

public class JsonAnnotationProcessor extends AbstractProcessor {

```

```

@Override

public synchronized void init(ProcessingEnvironment env) {}

@Override

public boolean process(Set<? extends TypeElement> elements,
RoundEnvironment env) {}
}

```

11.4.2 注册注解处理器

注解处理器定义好之后，为了让 `javac -processor` 能够对其进行处理，我们需要将注解处理器打包到一个 jar 文件中，同时，需要在 jar 文件中增加一个名为 `javax.annotation.processing.Processor` 的文件来指明 jar 文件中有哪些注解处理器，这个文件最终的路径位于 jar 文件根目录中的 `META-INF/services` 目录中，jar 文件解压后的目录结构如图 11-1 所示。



图 11-1

`javax.annotation.processing.Processor` 文件的内容是注解处理器全路径名，如果存在多个注解处理器，以换行进行分隔，语句如下。

```
com.bluelinelabs.logansquare.processor.JsonAnnotationProcessor
```

上面看到的是编译后的 jar 文件的目录结构，那么我们源文件的目录是怎样的呢？我们只需在和 `src/main/java` 同级目录中新建一个名为 `resources` 的目录，将 `javax.annotation.processing.Processor` 文件放进去就行，如图 11-2 所示。

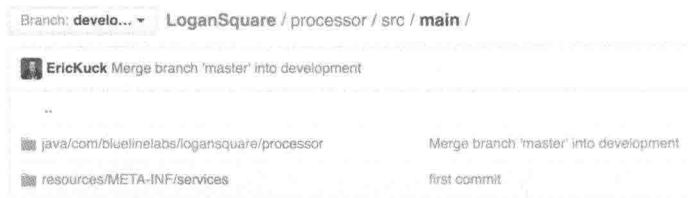


图11-2

值得一提的是，注解处理器所在的 Android Studio 工程必须是 Java Library 类型，而不应该是 Android Library 类型，因为在实现编译时的注解可能会用到 javax 包里面的类，而这个是不包含在 Android Library 的 JDK 中的。

手动执行上面的注册过程是很繁琐的，因此 Google 开源了一个名为 AutoService^[3] 的函数库来解放开发者的双手，引入这个函数库后，我们只需在定义自己的 Processor 时使用 @AutoService 注解标记即可完成上面的注册步骤，修改后的 JsonAnnotationProcessor 定义如下。

```
@AutoService(Processor.class)
@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes({
    // 该注解处理器支持的所有注解全名字符串类型的集合
})
public class JsonAnnotationProcessor extends AbstractProcessor {

    @Override
    public synchronized void init(ProcessingEnvironment env) {}

    @Override
    public boolean process(Set<? extends TypeElement> elements,
        RoundEnvironment env) {}
}
```

11.4.3 android-apt¹插件

我们知道，注解处理器所在的 jar 文件只在编译期间起作用，到应用运行时不会用到，因此，在 build.gradle 中引入依赖时应该以 provided 方式而不是 compile 方式引入，语句如下。

¹ <https://bitbucket.org/hvisser/android-apt>

```
dependencies {
    provided 'com.bluelinelabs:logansquare-compiler:1.3.6'
    compile 'com.bluelinelabs:logansquare:1.3.6'
}
```

当然，我们还有另外一个选择，就是使用 `android-apt` 插件的方式。`android-apt` 是在 Android Studio 中使用注解处理器的一个辅助插件，它的作用主要如下。

- 只在编译期间引入注解处理器所在的函数库作为依赖，不会打包到最终生成的 APK 中。
- 为注解处理器生成的源代码设置好正确的路径，以便 Android Studio 能够正常找到。

`android-apt` 的使用很简单，首先在使用到注解处理器函数库的模块的 `build.gradle` 文件中引入 `apt` 插件。

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        ...
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}

apply plugin: 'com.neenbedankt.android-apt'
```

接着以 `apt` 的方式引入注解处理器函数库作为依赖。

```
dependencies {
    apt 'com.bluelinelabs:logansquare-compiler:1.3.6'
    compile 'com.bluelinelabs:logansquare:1.3.6'
}
```

第12章

ANR产生的原因及其定位分析

ANR 是 Android 中一个独有的概念，每一本介绍 Android 开发的入门书几乎都会对其进行介绍，它的全称是 Application Not Responding（应用程序无响应）。相信从事 Android 开发的同学或多或少都会遇到过。对于高质量的代码，ANR 在开发者自测过程中可能不会经常遇到，但一旦测试人员进行 Monkey 测试，ANR 出现的概率就比较高了，如何快速分析定位并解决，是开发者的必修课。

ANR 的直观体验是用户在操作 APP 的过程中，感觉界面卡顿，比如按下某个按钮，打开某个页面等，当卡顿超过一定时间（一般是 5 秒）时就会出现 ANR 对话框，相信每个 Android 用户都见到过这个对话框，如图 12-1 所示。

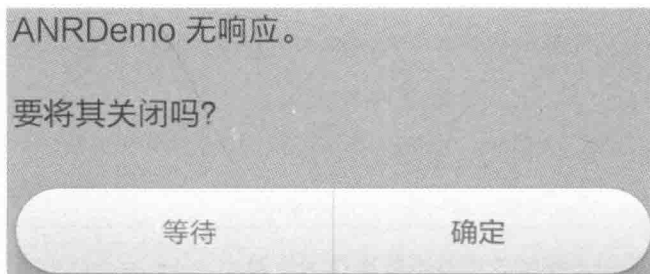


图 12-1

这时查看 Logcat，一般可以发现 ANR 以及 traces.txt 等字样，后面会详细介绍。可以发现，出现 ANR 主要是因为我们在主线程中做了太多耗时操作。这时你可以选择“等待”按钮，等待应用程序结束主线程的耗时操作，或者选择“确定”按钮，结束这个应用程序。Android 的早期用户应该可以发现一些体验比较差的 APP 中经常会出现上面这个对话框，ANR 对于一个应用来说是不能承受之痛，其影响并不比应用发生 Crash 小。

12.1 ANR 产生的原因

只有当应用程序的 UI 线程响应超时才会引起 ANR，超时产生原因一般有两种。

- 当前的事件没有机会得到处理，例如 UI 线程正在响应另外一个事件，当前事件由于某种原因被阻塞了。
- 当前的事件正在处理，但是由于耗时太长没能及时完成。

根据 ANR 产生的原因不同，超时时间也不尽相同，从本质上讲，产生 ANR 的原因有三种，大致可以对应到 Android 中四大组件中的三个（Activity/View、BroadcastReceiver 和 Service）。

KeyDispatchTimeout

最常见的一种类型，原因是 View 的按键事件或者触摸事件在特定的时间（5 秒）内无法得到响应。

BroadcastTimeout

原因是 BroadcastReceiver 的 onReceive() 函数运行在主线程中，在特定的时间（10 秒）内无法完成处理。

ServiceTimeout

比较少出现的一种类型，原因是 Service 的各个生命周期函数在特定时间（20 秒）内无法完成处理。

12.2 典型的 ANR 问题场景

- 应用程序 UI 线程存在耗时操作，例如在 UI 线程中进行网络请求，数据库操作或者文件操作等，可能会导致 UI 线程无法及时处理用户输入等。当然在 Android 4.0 之后，如果在 UI 线程中进行网络操作，将会抛出 NetworkOnMainThreadException 异常。
- 应用程序的 UI 线程等待子线程释放某个锁，从而无法处理用户的输入。
- 耗时的动画需要大量的计算工作，可能导致 CPU 负载过重。

12.3 ANR 的定位和分析

当发生 ANR 时，开发者可以通过结合 Logcat 日志和生成的位于手机内部存储的 `/data/anr/traces.txt` 文件进行分析和定位。

12.3.1 Logcat 日志信息

下面我们通过在主线程中模拟一个耗时操作来使应用发生 ANR，除了在 Logcat 中可以看到生成 `traces.txt` 文件的日志外。

```
Wrote stack traces to '/data/anr/traces.txt'
```

还可以得到如下详细的日志信息。

```
ActivityManager: ANR in com.ascel885.anrdemo (com.ascel885.anrdemo/.
MainActivity)
```

```
PID: 8672
```

```
Reason: Input dispatching timed out (Waiting to send key
event because the focused window has not finished processing all of the input
events that were previously delivered to it. Outbound queue length: 0.
Wait queue length: 21.)
```

```
Load: 13.11 / 11.92 / 7.37
```

```
CPU usage from 0ms to 7361ms later:
```

```
61% 947/system_server: 24% user + 37% kernel / faults: 11884
minor 106 major
```

```
...
```

```
1.3% 8672/com.ascel885.anrdemo: 0.7% user + 0.6% kernel /
faults: 7276 minor 17 major
```

```
54% TOTAL: 21% user + 31% kernel + 0.8% iowait + 0.1%
softirq
```

```
CPU usage from 6373ms to 6910ms later:
```

```
43% 947/system_server: 29% user + 13% kernel / faults: 29
minor
```

```
24% 1156/LazyTaskWriterT: 22% user + 1.7% kernel
```

```
8.6% 1044/ActivityManager: 1.7% user + 6.8% kernel
```

```
...
```

同时如果由于 ANR 导致应用发生崩溃，那么除了打印出上面的信息之外，还会调用 CrashAnrDetector 打印出下面的信息。

```
CrashAnrDetector: Process: com.ascel885.anrdemo
                  Flags: 0xe8be46
                  Package: com.ascel885.anrdemo v1 (1.0)
                  Activity: com.ascel885.anrdemo/.MainActivity
                  Subject: Input dispatching timed out (Waiting to send key
event because the focused window has not finished processing all of the input
events that were previously delivered to it.  Outbound queue length: 0.
Wait queue length: 21.)
                  Build: samsung/kltezn/klte:5.0/LRX21T/G9006VZNU1BOJ4:user/
release-keys
                  CPU usage from 0ms to 7361ms later:
                      61% 947/system_server: 24% user + 37% kernel / faults:
11884 minor 106 major
                      1.3% 8672/com.ascel885.anrdemo: 0.7% user + 0.6% kernel
/ faults: 7276 minor 17 major
                      ...
                      54% TOTAL: 21% user + 31% kernel + 0.8% iowait + 0.1%
softirq
CrashAnrDetector: processName: com.ascel885.anrdemo
CrashAnrDetector: broadcastEvent : com.ascel885.anrdemo data_app_anr
```

可以看到，Logcat 日志信息中主要包含如下内容。

- 导致 ANR 的类名及所在包名：MainActivity, com.ascel885.anrdemo。
- 发生 ANR 的进程名称及 ID：com.ascel885.anrdemo, 8672。
- ANR 产生的原因（类型）：Input dispatching timed out, 属于 KeyDispatchTimeout 类型。
- 系统中活跃进程的 CPU 占用率：1.3% 8672/com.ascel885.anrdemo: 0.7% user + 0.6% kernel / faults: 7276 minor 17 major。

12.3.2 traces.txt 日志信息

从 Logcat 的日志信息我们可以知道引发 ANR 的具体的类信息，以及 ANR 的类型，但是这不足够开发者定位到具体引发问题的代码行，为了获得进一步的信息，我们需要借助于 ANR 过程中生成的堆栈信息文件 /data/anr/traces.txt。这个文件可以通过终端 Terminal 中执行 adb pull 命令从手机的内部存储中拷贝到电脑中，也可通过如下语句拷贝到 MacBook Pro 的桌面上。

```
adb pull /data/anr/traces.txt ~/Desktop/
```

使用文本阅读器打开这个文件，一般在文件开头部分就可以看到可能导致 ANR 的堆栈信息如下。

```
----- pid 8672 at 2016-04-25 21:59:14 -----
Cmd line: com.ascel885.anrdemo
ABI: arm
Build type: optimized
Loaded classes: 19845 allocated classes
Intern table: 5035 strong; 0 weak
JNI: CheckJNI is on; globals=271
...
Heap: 24% free, 16MB/21MB; 53963 objects
...
DALVIK THREADS (13):
"main" prio=5 tid=1 Sleeping
  | group="main" sCount=1 dsCount=0 obj=0x87789ef0 self=0xb4f07800
  | sysTid=8672 nice=0 cgrp=apps sched=0/0 handle=0xb6fe4ec8
  | state=S schedstat=( 246636720 148938849 453 ) utm=18 stm=6 core=3 HZ=100
  | stack=0xbe5bd000-0xbe5bf000 stackSize=8MB
  | held mutexes=
  at java.lang.Thread.sleep!(Native method)
  - sleeping on <0x165e2275> (a java.lang.Object)
  at java.lang.Thread.sleep(Thread.java:1031)
  - locked <0x165e2275> (a java.lang.Object)
  at java.lang.Thread.sleep(Thread.java:985)
  at com.ascel885.anrdemo.MainActivity.triggerAnrWithLongOperation(MainActiv
```

```

ity.java:32)
    at com.ascel1885.anrdemo.MainActivity$$ViewBinder$1.onClick(MainActivity$$ViewBinder.java:17)
    at butterknife.internal.DebouncingOnClickListener.onClick(DebouncingOnClickListener.java:22)
    at android.view.View.performClick(View.java:5155)
    at android.view.View$PerformClick.run(View.java:20747)
    at android.os.Handler.handleCallback(Handler.java:739)
    at android.os.Handler.dispatchMessage(Handler.java:95)
    at android.os.Looper.loop(Looper.java:145)
    at android.app.ActivityThread.main(ActivityThread.java:5835)
    at java.lang.reflect.Method.invoke!(Native method)
    at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:1399)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1194)

```

可以看到 traces.txt 文件中有助于问题定位的信息主要包括如下内容。

- 发生 ANR 的进程名称、ID，以及时间。

```
----- pid 8672 at 2016-04-25 21:59:14 -----
```

```
Cmd line: com.ascel1885.anrdemo
```

- 手机的 CPU 架构：arm。
- 堆内存信息：24% free, 16MB/21MB; 53963 objects。
- 主线程基本信息。
 - ☐ 线程名称：“main”
 - ☐ 线程优先级：prio=5
 - ☐ 线程锁 ID：tid=1
 - ☐ 线程状态：Sleeping
- 主线程的详细信息。

- ❑ 线程组名称: group= "main"
- ❑ 线程被挂起的次数: sCount=1
- ❑ 线程被调试器挂起的次数: dsCount=0, 当线程被调试结束后, sCount 会被重置为0, 它的值会重新根据是否被挂起而增加, 而 dsCount 不会被重置为0。因此, dsCount 可以用来判断这个线程是否被调试器调试过。
- ❑ 线程的 Java 对象地址: obj=0x87789ef0
- ❑ 线程本身的 Native 对象地址: self=0xb4f07800
- 线程的调度信息。
 - ❑ Linux 系统中内核线程 ID: sysTid=8672, 可以看到主线程的线程号和进程号相同
 - ❑ 线程调度优先级: nice=0
 - ❑ 线程调度组: cgrp=apps
 - ❑ 线程调度策略和优先级: sched=0/0
 - ❑ 线程处理函数地址: handle=0xb6fe4ec8
- 线程的上下文信息。
 - ❑ 线程调度状态: state=S
 - ❑ 线程在 CPU 中的执行时间、线程等待时间、线程执行的时间片长度: schedstat=(246636720 148938849 453)
 - ❑ 线程在用户态中调度时间值 (单位: jiffies) : utm=18
 - ❑ 线程在内核态中的调度时间值: stm=6
 - ❑ 最后执行这个线程的 CPU 核序号: core=3
- 线程的堆栈信息。
 - ❑ 堆栈地址和大小: stack=0xbe5bd000-0xbe5bf000 stackSize=8MB
 - ❑ 堆栈信息: 从中可以看到, ANR是由于在MainActivity类中的triggerAnrWithLongOperation函数调用了 Thread.sleep 导致。


```
at java.lang.Thread.sleep(Thread.java:985)
at com.ascel1885.anrdemo.MainActivity.triggerAnrWithLongOperation(MainActivity.java:32)
```

12.4 ANR的避免和检测

为了避免在开发中引入可能导致应用发生 ANR 的问题，除了切记不要在主线程中作耗时操作，我们也可以借助于一些工具来进行检测，从而更有效的避免 ANR 的引入。

12.4.1 StrictMode

严格模式 StrictMode 是 Android SDK 提供的一个用来检测代码中是否存在违规操作的工具类，StrictMode 主要检测两大类问题。

- 线程策略 ThreadPolicy。
 - ❑ detectCustomSlowCalls：检测自定义耗时操作。
 - ❑ detectDiskReads：检测是否存在磁盘读取操作。
 - ❑ detectDiskWrites：检测是否存在磁盘写入操作。
 - ❑ detectNetwork：检测是否存在网络操作。
- 虚拟机策略 VmPolicy。
 - ❑ detectActivityLeaks：检测是否存在 Activity 泄漏。
 - ❑ detectLeakedClosableObjects：检测是否存在未关闭的 Closable 对象泄漏。
 - ❑ detectLeakedSqlLiteObjects：检测是否存在 Sqlite 对象泄漏。
 - ❑ setClassInstanceLimit：检测类实例个数是否超过限制。

可以看到，其中的 ThreadPolicy 可以用来检测可能存在的主线程耗时操作，解决这些检测到的问题能够减少应用发生 ANR 的概率。需要注意的是，我们只能在 Debug 版本中使用它，发布到市场上的版本要关闭掉。StrictMode 的使用很简单，我们只需在应用初始化的地方例如 Application 或者 MainActivity 类的 onCreate 方法中执行如下代码即可。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    if (BuildConfig.DEBUG) {
        // 开启线程模式
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder().
detectAll().penaltyLog().build());
        // 开启虚拟机模式
        StrictMode.setVmPolicy(new VmPolicy.Builder().detectAll().
penaltyLog().build());
    }
    super.onCreate(savedInstanceState);
}
```

上面的初始化代码调用 `penaltyLog` 表示在 Logcat 中打印日志，调用 `detectAll` 方法表示启动所有的检测策略，我们也可以根据应用的具体需求只开启某些策略，语句如下。

```
StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
    .detectDiskReads()
    .detectDiskWrites()
    .detectNetwork()
    .penaltyLog()
    .build());

StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
    .detectActivityLeaks()
    .detectLeakedSqlLiteObjects()
    .detectLeakedClosableObjects()
    .penaltyLog()
    .build());
```

12.4.2 BlockCanary¹

BlockCanary 是一个非侵入式的性能监控函数库，它的用法和 LeakCanary 类似，只不过后者监控应用的内存泄漏，而 BlockCanary 主要用来监控应用主线程的卡顿。它的基本原理是利

¹ <https://github.com/moduth/blockcanary>

用主线程的消息队列处理机制，通过对比消息分发开始和结束的时间点来判断是否超过设定的时间，如果是，则判断为主线程卡顿。它的集成很简单，首先在 build.gradle 中添加在线依赖，语句如下。

```
dependencies {
    compile 'com.github.moduth:blockcanary-android:1.2.1'

    // 仅在debug包启用BlockCanary进行卡顿监控和提示的话，可以这么用
    debugCompile 'com.github.moduth:blockcanary-android:1.2.1'
    releaseCompile 'com.github.moduth:blockcanary-no-op:1.2.1'
}
```

然后在 Application 类中进行配置和初始化即可，语句如下。

```
public class DemoApplication extends Application {
    @Override
    public void onCreate() {
        ...
        // 在主进程初始化调用
        BlockCanary.install(this, new AppBlockCanaryContext()).start();
    }
}

public class AppBlockCanaryContext extends BlockCanaryContext {
    // 实现各种上下文，包括应用标识符、用户uid、网络类型、卡慢判断阈值、Log保存位置等
}
```

更详细的信息可以参见官网说明。

第13章

Android异步处理技术

移动应用的开发要求我们正确的处理好主线程和子线程之间的关系，耗时的操作应该放到子线程中，避免阻塞主线程，导致 ANR。异步处理技术是提高应用性能，解决主线程和子线程之间通信问题的关键。

在 Android 中，异步处理技术有很多种，常见的有 Thread、AsyncTask、Handler & Looper、Executors 等，在实际项目中，我们需要根据具体业务需求进行选择，一个完整的异步处理技术继承树如图 13-1 所示，本章将会一一进行介绍。

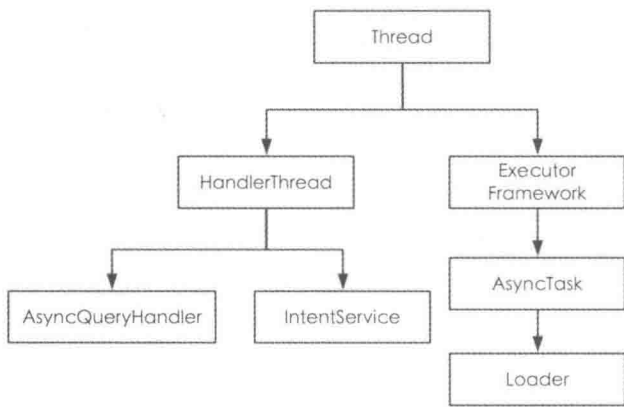


图 13-1

13.1 Thread

线程是 Java 语言的一个概念，它是实际执行任务的基本单元，从图 13-1 中可以看出，

Thread 是 Android 中异步处理技术的基础，相信大家对它并不陌生，创建线程有两种方法。

- 继承 Thread 类并重写 run 方法，语句如下。

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        // 实现具体逻辑，例如文件读写，网络请求等  
    }  
}  
  
public void startThread() {  
    MyThread myThread = new MyThread();  
    myThread.start(); // 使用 start 启动线程  
}
```

- 实现 Runnable 接口并实现 run 方法，语句如下。

```
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        // 实现具体逻辑，例如文件读写，网络请求等  
    }  
}  
  
public void startThread() {  
    MyRunnable runnable = new MyRunnable();  
    Thread thread = new Thread(runnable);  
    thread.start();  
}
```

Android 应用中各种类型的线程本质上都基于 Linux 系统的 pthreads，在应用层可以分为三种类型的线程。

- 主线程：主线程也称为 UI 线程，随着应用启动而启动，主线程用来运行 Android 组件，同时刷新屏幕上的 UI 元素。Android 系统如果检测到非主线程更新 UI 组件，那么就会抛出 CalledFromWrongThreadException 异常，只有主线程才能操作 UI，是因为 Android

的 UI 工具包不是线程安全的。主线程中创建的 Handler 会顺序执行接收到的消息，包括从其他线程发送的消息。因此，如果消息队列中前面的消息没有很快执行完，那么它可能会阻塞队列中的其他消息的及时处理。

- **Binder 线程：**Binder 线程用于不同进程之间线程的通信，每个进程都维护了一个线程池，用来处理其他进程中线程发送的消息，这些进程包括系统服务、Intents、ContentProviders 和 Service 等。在大部分情况下，应用不需要关心 Binder 线程，因为系统会优先将请求转换为使用主线程。一个典型的需要使用 Binder 线程的场景是应用提供一个给其他进程通过 AIDL 接口绑定的 Service。
- **后台线程：**在应用中显式创建的线程都是后台线程，也就是当刚创建出来时，这些线程的执行体是空的，需要手动添加任务。在 Linux 系统层面，主线程和后台线程是一样的。在 Android 框架中，通过 WindowManager 赋予了主线程只能处理 UI 更新以及后台线程不能直接操作 UI 的限制。

13.2 HandlerThread

HandlerThread 是一个集成了 Looper 和 MessageQueue 的线程，当启动 HandlerThread 时，会同时生成 Looper 和 MessageQueue，然后等待消息进行处理，它的 run 方法源码如下。

```
@Override
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

使用 `HandlerThread` 的好处是开发者不需要自己去创建和维护 `Looper`，它的用法和普通线程一样，语句如下。

```
HandlerThread handlerThread = new HandlerThread("HandlerThread");
handlerThread.start();

mHandler = new Handler(handlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        // 处理接收到的消息
    }
};
```

`HandlerThread` 中只有一个消息队列，队列中的消息是顺序执行的，因此是线程安全的，当然吞吐量自然受到一定的影响，队列中的任务可能会被前面没有执行完的任务阻塞。`HandlerThread` 的内部机制确保了在创建 `Looper` 和发送消息之间不存在竞态条件，这个是通过将 `HandlerThread.getLooper()` 实现为一个阻塞操作实现的，只有当 `HandlerThread` 准备好接收消息之后才会返回，源码如下。

```
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }

    // 如果线程已经启动，那么在 Looper 准备好之前应先等待
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }

    return mLooper;
}
```

如果具体业务要求在 `HandlerThread` 开始接收消息之前要进行某些初始化操作的话，可以重写 `HandlerThread` 的 `onLooperPrepared` 函数，例如可以在这个函数中创建与 `HandlerThread` 关联的 `Handler` 实例，这同时也可以对外隐藏我们的 `Handler` 实例，语句如下。

```
public class MyHandlerThread extends HandlerThread {
    private Handler mHandler;

    public MyHandlerThread() {
        super("MyHandlerThread", Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    protected void onLooperPrepared() {
        super.onLooperPrepared();
        mHandler = new Handler(getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                switch(msg.what) {
                    case 1:
                        // Handle message
                        break;
                    case 2:
                        // Handle message
                        break;
                }
            }
        };
    }

    public void publishedMethod1() {
        mHandler.sendMessage(1);
    }

    public void publishedMethod2() {
        mHandler.sendMessage(2);
    }
}
```

13.3 AsyncQueryHandler

AsyncQueryHandler 是用于在 ContentProvider 上面执行异步的 CRUD (Create、Read、Update、Delete) 操作的工具类, CRUD 操作会被放到一个单独的子线程中执行, 当操作结束获取到结果后, 将通过消息的方式传递给调用 AsyncQueryHandler 的线程, 通常就是主线程。AsyncQueryHandler 是一个抽象类, 继承自 Handler, 通过封装 ContentResolver、HandlerThread、Handler 等实现对 ContentProvider 的异步操作, 原理图¹如图 13-2 所示。

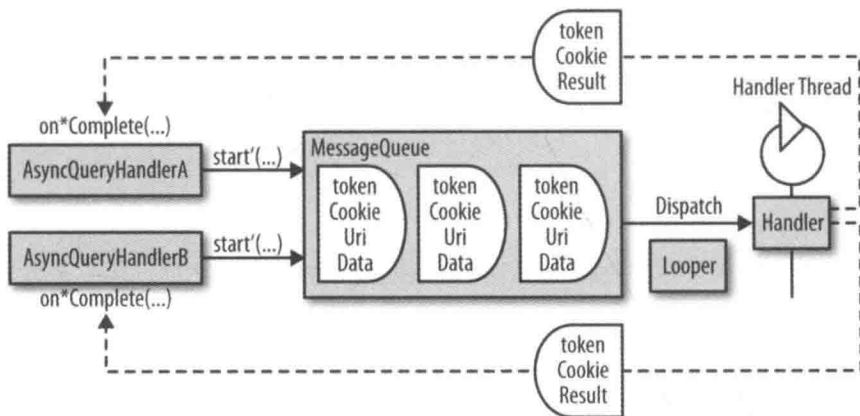


图13-2

AsyncQueryHandler 封装了如下四个方法用来操作 ContentProvider, 分别对应上面说到的 CRUD 操作。

```
final void startDelete(int token, Object cookie, Uri uri, String selection,
String[] selectionArgs);
```

```
final void startInsert(int token, Object cookie, Uri uri, ContentValues
initialValues);
```

```
final void startQuery(int token, Object cookie, Uri uri, String[] projection,
String selection, String[] selectionArgs, String orderBy);
```

¹ Efficient Android Threading

```
final void startUpdate(int token, Object cookie, Uri uri, ContentValues values, String selection, String[] selectionArgs)
```

AsyncQueryHandler 的子类可以根据实际需求实现下面的回调函数，从而得到上面的 CRUD 操作的返回结果。

```
@Override
protected void onDeleteComplete(int token, Object cookie, int result) { ...
}

@Override
protected void onUpdateComplete(int token, Object cookie, int result) { ...
}

@Override
protected void onInsertComplete(int token, Object cookie, Uri result) { ...
}

@Override
protected void onQueryComplete(int token, Object cookie, Cursor result) { ...
}
```

13.4 IntentService

我们知道 Service 的各个生命周期函数是运行在主线程的，因此它本身并不是一个异步处理技术。为了能够在 Service 中实现在子线程中处理耗时任务，Android 引入了一个 Service 的子类：IntentService。IntentService 具有 Service 一样的生命周期，同时也提供了在后台线程中处理异步任务的机制。与 HandlerThread 类似，IntentService 也是在一个后台线程中顺序执行所有的任务，我们通过给 Context.startService 传递一个 Intent 类型的参数可以启动 IntentService 的异步执行，如果此时 IntentService 正在运行中，那么这个新的 Intent 将会进入队列进行排队，直到后台线程处理完队列前面的任务；如果此时 IntentService 没有在运行，那么将会启动一个新的 IntentService，当后台线程队列中所有任务处理完成之后，IntentService 将会结束它的生命周期，因此 IntentService 不需要开发者手动结束。

IntentService 本身是一个抽象类，因此，使用前需要继承它并实现 `onHandleIntent` 方法，在这个方法中实现具体的后台处理业务逻辑，同时在子类的构造方法中需要调用 `super(String name)` 传入子类的名字，语句如下。

```
public class SimpleIntentService extends IntentService {
    public SimpleIntentService() {
        super(SimpleIntentService.class.getName());
        setIntentRedelivery(true);
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        // 这个方法是在后台线程中调用的
    }
}
```

上面代码中的 `setIntentRedelivery` 方法如果设置为 `true`，那么 `IntentService` 的 `onStartCommand` 方法将会返回 `START_REDELIVER_INTENT`。这时，如果 `onHandleIntent` 方法返回之前进程死掉了，那么进程将会重新启动，`intent` 将会重新投递。

当然，类似 `Service`，不要忘记在 `AndroidManifest.xml` 文件中注册 `SimpleIntentService`。

```
<service android:name=".SimpleIntentService" />
```

通过查看 `IntentService` 的源码，我们可以发现事实上 `IntentService` 是通过 `HandlerThread` 来实现后台任务的处理的，代码逻辑很简单，语句如下。

```
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }
}
```

```
    }

    @Override
    public void handleMessage(Message msg) {
        onHandleIntent((Intent)msg.obj);
        stopSelf(msg.arg1);
    }
}

public IntentService(String name) {
    super();
    mName = name;
}

@Override
public void onCreate() {
    super.onCreate();
    HandlerThread thread = new HandlerThread("IntentService[ " + mName
+ "]" );
    thread.start();

    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public void onStart(Intent intent, int startId) {
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    msg.obj = intent;
    mServiceHandler.sendMessage(msg);
}

@Override
```

```

public int onStartCommand(Intent intent, int flags, int startId) {
    onStart(intent, startId);
    return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
}

```

13.5 Executor Framework

我们知道，创建和销毁对象（例如线程），是存在开销的，如果应用中频繁出现线程的创建和销毁，那么会影响到应用的性能。使用 Java Executor 框架可以通过线程池等机制解决这个问题，改善应用的体验。Executor 框架为开发者提供了如下能力。

- 创建工作线程池，同时通过队列来控制能够在这些线程执行的任务的个数。
- 检测导致线程意外终止的错误。
- 等待线程执行完成并获取执行结果。
- 批量执行线程，并通过固定的顺序获取执行结果。
- 在合适的时机启动后台线程，从而保证线程执行结果可以很快反馈给用户。

Executor 框架的基础是一个名为 Executor 的接口定义，Executor 的主要目的是分离任务的创建和它的执行，最终实现上述所说的功能点。

```

public interface Executor {
    void execute(Runnable command);
}

```

开发者可以通过实现 Executor 接口并重写 execute 方法从而实现自己的 Executor 类，最简单的是直接在这个方法中创建一个线程来执行 Runnable。

```

public class SimpleExecutor implements Executor {
    @Override
    public void execute(Runnable runnable) {
        new Thread(runnable).start();
    }
}

```

当然，实际应用中 `execute` 方法很少是这么简单的，通常需要增加类似队列，任务优先级等功能，最终实现一个线程池。线程池是任务队列和工作线程的集合，这两者组合起来实现生产者消费者模式。Executor 框架为开发者提供了预定义的线程池实现，内容如下。

- 固定大小的线程池：通过 `Executors.newFixedThreadPool(n)` 创建，其中 `n` 表示线程池中线程的个数。
- 可变大小的线程池：通过 `Executors.newCachedThreadPool()` 创建，当有新的任务需要执行时，线程池会创建新的线程来处理它，空闲的线程会等待 60秒来执行新任务，当没有任务可执行时就自动销毁，因此可变大小线程池会根据任务队列的大小而变化。
- 单个线程的线程池：通过 `Executors.newSingleThreadExecutor()` 创建，这个线程池中永远只有一个线程来串行执行任务队列中的任务。

预定义的线程池都是基于 `ThreadPoolExecutor` 类之上构建的，而通过 `ThreadPoolExecutor` 开发者可以自定义线程池的一些行为，我们主要来看看这个类的构造函数定义。

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue);
```

- `corePoolSize`：核心线程数，核心线程会一直存在于线程池中，即使当前没有任务需要处理；当线程数小于核心线程数时，即使当前有空闲的线程，线程池也会优先创建新的线程来处理任务。
- `maximumPoolSize`：最大线程数，当线程数大于核心线程数，且任务队列已经满了，这时线程池就会创建新的线程，直到线程数量达到最大线程数为止。
- `keepAliveTime`：线程的空闲存活时间，当线程的空闲时间超过这个之时，线程会被销毁，直到线程数等于核心线程数。
- `unit`：`keepAliveTime` 的单位，可选的有 `TimeUnit` 类中的 `NANOSECONDS`、`MICROSECONDS`、`MILLISECONDS` 和 `SECONDS`。
- `workQueue`：线程池所使用的任务缓冲队列。

13.6 AsyncTask

从文章开头的继承树可以看到，AsyncTask 是在 Executor 框架基础上进行的封装，它实现将耗时任务移动到工作线程中执行，同时提供方便的接口实现工作线程和主线程的通信，使用 AsyncTask 一般会用到如下方法。

```
public class FullTask extends AsyncTask<Params, Progress, Result> {
    @Override
    protected void onPreExecute() { ... }

    @Override
    protected Result doInBackground(Params... params) { ... }

    @Override
    protected void onProgressUpdate(Progress... progress) { ... }

    @Override
    protected void onPostExecute(Result result) { ... }

    @Override
    protected void onCancelled(Result result) { ... }
}
```

其中，除了 `doInBackground` 方法是在工作线程中执行，其他的都是在主线程中执行，具体的使用方法各位应该都很清楚，本章不做赘述，接下来主要看一下使用 AsyncTask 需要注意的地方。首先，我们需要了解，在不同的 Android 系统版本中，AsyncTask 的表现不尽相同，如表 13-1 所示。在不同的系统版本中，AsyncTask 的 `execute` 和 `executeOnExecutor` 方法的运行有些许差别。

表 13-1

API Level	execute 方法	executeOnExecutor 方法
1 ~ 3	串行执行	还没有这个方法
4 ~ 10	并行执行	还没有这个方法
11 ~ 12	并行执行	串行或者并行
13+	串行执行	串行或者并行

可以看到, 如果使用 AsyncTask 执行的任务需要并行执行的话, 那么在 API Level 大于 13 的版本上建议使用 `executeOnExecutor` 代替 `execute`。

一个应用中使用的所有 AsyncTask 实例会共享全局的属性, 也就是说如果 AsyncTask 中的任务是串行执行, 那么应用中所有的 AsyncTask 都会进行排队, 只有等前面的任务执行完成之后, 才会接着执行下一个 AsyncTask 中的任务, 在 `executeOnExecutor(AsyncTask.SERIAL_EXECUTOR)` 或者 API Level 大于 13 的系统上面执行 `execute()` 方法, 都会是这个效果; 如果 AsyncTask 是异步执行, 那么在四核 CPU 系统上, 最多也只有五个任务可以同时进行, 其他任务需要在队列中排队, 等待空闲的线程。之所以会出现这种情况是由于 AsyncTask 中的 `ThreadPoolExecutor` 指定核心线程数是系统 CPU 核数 +1, 语句如下。

```
public abstract class AsyncTask<Params, Progress, Result> {
    private static final int CPU_COUNT = Runtime.getRuntime().
availableProcessors();

    private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
    private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
    private static final int KEEP_ALIVE = 1;

    public static final Executor THREAD_POOL_EXECUTOR
        = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE,
KEEP_ALIVE,
                                TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
}
```

13.7 Loader

Loader 是 Android 3.0 开始引入的一个异步数据加载框架, 它使得在 Activity 或者 Fragment 中异步加载数据变得很简单, 同时它在数据源发生变化时, 能够及时发出消息通知。Loader 框架涉及的 API 主要如下。

- Loader: 加载器框架的基类, 封装了实现异步数据加载的接口, 当一个加载器被激活后, 它就会开始监视数据源并在数据发生改变时发送新的结果。
- AsyncTaskLoader: Loader 的子类, 顾名思义, 它是基于 AsyncTask 实现的异步数据加

载，它是一个抽象类，子类必须实现 `loadInBackground` 方法，在其中进行具体的数据加载操作。

- **CursorLoader**: `AsyncTaskLoader` 的子类，封装了对 `ContentResolver` 的 `query` 操作，实现从 `ContentProvider` 中查询数据的功能。
- **LoaderManager**: 抽象类，`Activity` 和 `Fragment` 默认都会关联一个 `LoaderManager` 的对象，开发者只需要通过 `getLoaderManager` 即可获得。`LoaderManager` 是用来管理一个或者多个加载器对象的。
- **LoaderManager.LoaderCallbacks**: `LoaderManager` 的回调接口，主要有如下三个方法。
 - ❑ `onCreateLoader()`: 初始化并返回一个新的 `Loader` 实例。
 - ❑ `onLoadFinished()`: 当一个加载器完成加载过程之后会回调这个方法。
 - ❑ `onLoaderReset()`: 当一个加载器被重置并且数据无效时会回调这个方法。

一个简单的 `Loader` 使用例子如下。

```
public class ContactActivity extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor> {

    private static final int CONTACT_NAME_LOADER_ID = 0;

    // 这里的 PROJECTION 只获取ID和用户名两个字段
    static final String[] CONTACTS_SUMMARY_PROJECTION = new String[] {
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.DISPLAY_NAME
    };

    SimpleCursorAdapter mAdapter;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        initAdapter();
        // 通过 LoaderManager 初始化 Loader, 这会回调到 onCreateLoader
```

```
        getLoaderManager().initLoader(CONTACT_NAME_LOADER_ID, null, this);
    }

    private void initAdapter() {
        mAdapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_1, null,
            new String[] { ContactsContract.Contacts.DISPLAY_NAME },
            new int[] { android.R.id.text1 }, 0);
        setListAdapter(mAdapter);
    }

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // 实际创建 Loader 的地方，此处使用 CursorLoader
        return new CursorLoader(this, ContactsContract.Contacts.CONTENT_URI,
            CONTACTS_SUMMARY_PROJECTION, null, null,
            ContactsContract.Contacts.DISPLAY_NAME + " ASC");
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
        // 后台线程中加载完数据后，回调这个方法将数据传递给主线程
        mAdapter.swapCursor(c);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        // Loader 被重置后的回调，在这里可以重新刷新页面数据
        mAdapter.swapCursor(null);
    }
}
```

13.8 总结

Android 平台提供了如此多的异步处理技术，我们在进行选择的时候需要根据具体的业务需求而定，综合考量以下几个因素。

- 尽量使用更少的系统资源，例如 CPU 和内存等。
- 为应用提供更好的性能和响应度。
- 实现和使用起来不复杂。
- 写出来的代码是否符合好的设计，是否易于理解和维护。

第14章

Android数据序列化方案研究

数据序列化在 Android 应用开发中占据着举足轻重的位置，无论是进程间通信、本地数据存储还是网络数据传输等，都离不开序列化的支持。针对不同场景选择正确的序列化方案，对应用的性能有着极大的影响。

广义上讲，序列化是将数据结构或者对象转换成可用于存储或者传输的数据格式的过程，在序列化期间，数据结构或者对象将其状态信息写入到临时或者持久性存储区中；反序列化是将序列化过程中生成的数据还原成数据结构或者对象的过程。Android 应用开发有很多种可选的序列化和反序列化方案，本章将逐一进行介绍。

14.1 Serializable

Serializable 是 Java 语言的特性，它是最简单的也是使用最广泛的序列化方案之一，只有实现了 Serializable 接口的 Java 对象才可以实现序列化。这种类型的序列化是将 Java 对象转换成字节序列的过程，而反序列化则是将字节序列恢复成 Java 对象的过程。

Serializable 接口是一种标识接口，也就是无需实现方法，Java 便会对这个对象进行序列化操作。它的缺点是使用反射机制，在序列化的过程中会创建很多临时对象，容易触发垃圾回收，序列化的过程比较慢，对于性能要求很严格的场景不建议使用这种方案。下面以实际项目中一段代码来说明其使用方式。

首先定义 Java 对象类，它必须实现 Serializable 接口。序列化过程中会使用名为 serialVersionUID 的版本号和序列化的类相关联，serialVersionUID 在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类对象，如果接收者加载的对象的 serialVersionUID 和发送者加载的对象的 serialVersionUID 取值不同，则反序列化过程会

出现 `InvalidClassException` 异常, 这种情况在网络通信两个节点间的序列化特别需要注意。最佳实践是显式指定 `serialVersionUID` 的值 (IDE 为我们随机生成), 语句如下。

```
public class WenkuBanner implements Serializable {

    private static final long serialVersionUID = 916501062446384003L;

    public int mAccessTime; // 本次访问时间

    public ArrayList<Image> mContentList; // 轮播图片数组

    public class Image implements Serializable {

        private static final long serialVersionUID = 8723340374426479692L;

        public int mType; // 类型标识: 1-文档列表, 2-文档

        public String mIconUrl; // 图标URL

        public String mValue; // id

    }

    /**
     * 默认构造函数, 作为空对象使用
     */
    public WenkuBanner() {
        mAccessTime = 0;
        mContentList = new ArrayList<Image>();
    }
}
```

如果想要序列化一个对象, 首先需要创建某种类型的 `OutputStream`, 例如 `ByteArrayOutputStream`、`FileOutputStream` 等, 接着将这些 `OutputStream` 封装到一个 `ObjectOutputStream` 对象中, 然后就

可以调用 `ObjectOutputStream` 对象的 `writeObject` 方法将对象进行序列化。需要注意的是，对象的序列化是基于字节的，因此不能使用 `Reader` 和 `Writer` 这种基于字符的方式。

为了将 `WenkuBanner` 对象序列化到磁盘，首先要在磁盘上创建一个文件，用于存储序列化后的数据，然后就是调用 `writeObject` 方法，语句如下。

```
/**
 * 对象序列化
 * @param type 类型
 * @param obj 要进行序列化的对象
 */
private static void writeCache(int type, Object obj) {

    if (!SDCardUtils.isSDCardWritable()) {
        LogUtil.d(TAG, "SDCard not writable");
        return;
    }

    File file = null;
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;

    try {
        switch (type) {
            case TYPE_BANNER:
                if (SDCardUtils.createBannerFolder()) {
                    file = new File(ReaderSettings.DEFAULT_RECOMMEND_BANNER_
CACHE_FOLDER + "/banner");
                }
                break;
            case TYPE_RECOMDDOCS:
                // ...
                break;
        }
    }
```

```

case TYPE_CLASSIFICATION:
    // ...
    break;
default:
    break;
}

if (file != null) {
    fos = new FileOutputStream(file);
    oos = new ObjectOutputStream(fos);
    if (oos != null) {
        oos.writeObject(obj);
        oos.flush();
    }
}

} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (oos != null) {
            oos.close();
        }
        if (fos != null) {
            fos.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

反序列化的过程和序列化过程是相对的，也就是需要将数据转换成某种类型的InputStream，例如ByteArrayInputStream、FileInputStream等，接着封装在ObjectInputStream对象中，最后调用readObject方法即可，语句如下。

```
private static Object readCache(int type) {

    File file = null;
    FileInputStream fis = null;
    ObjectInputStream ois = null;
    Object obj = null;

    try {
        switch (type) {
            case TYPE_BANNER:
                WenkuBanner banner = new WenkuBanner();
                if (!SDCardUtils.isSDCardAvailable()) {
                    return banner;
                }

                file = new File(ReaderSettings.DEFAULT_RECOMMEND_BANNER_CACHE_
FOLDER + "/" + "banner");
                if (!file.exists()) {
                    return banner;
                }
                obj = banner; // 保证返回值不为null，且类型正确
                break;
            case TYPE_RECOMDDOCS:
                // ...
                break;
            case TYPE_CLASSIFICATION:
                // ...
                break;
            default:
                break;
        }
    }
}
```



```
    }

    fis = new FileInputStream(file);
    ois = new ObjectInputStream(fis);
    if (ois != null) {
        obj = ois.readObject();
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (StreamCorruptedException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (ois != null) {
            ois.close();
        }
        if (fis != null) {
            fis.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

return obj;
}
```

14.2 Parcelable

前面介绍的 `Serializable` 是 JDK 提供的接口，这种序列化方式是基于磁盘或者网络的，而 `Parcelable` 是 Android SDK 提供的，它是基于内存的，由于内存读写速度高于磁盘，因此在 Android 中跨进程对象的传递一般使用 `Parcelable`。

`Parcelable` 接口源码如下。

```
public interface Parcelable {

    public static final int PARCELABLE_WRITE_RETURN_VALUE = 0x0001;

    public static final int CONTENTS_FILE_DESCRIPTOR = 0x0001;

    public int describeContents();

    public void writeToParcel(Parcel dest, int flags);

    public interface Creator<T> {

        public T createFromParcel(Parcel source);

        public T[] newArray(int size);
    }

    public interface ClassLoaderCreator<T> extends Creator<T> {

        public T createFromParcel(Parcel source, ClassLoader loader);
    }
}
```

很明显，实现 `Parcelable` 并不容易，需要写大量的模板代码，这使得对象代码变得难以阅读和维护。好在程序员是聪明的，在 Android Studio 中可以安装一个名为 `Android Parcelable code generator` 的插件，如图 14-1 所示。

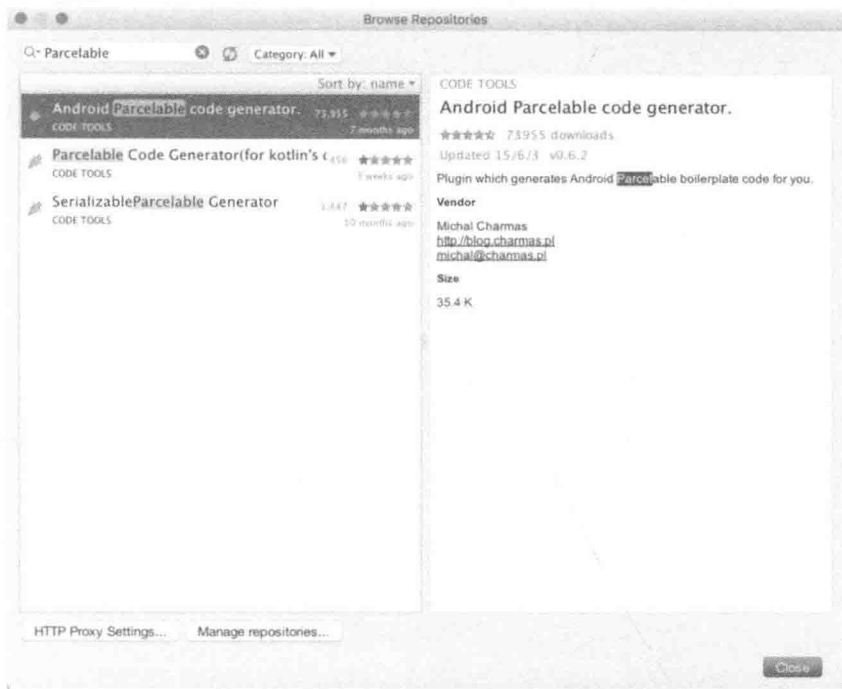


图 14-1

安装完成后,重启 Android Studio 使其生效,接着编写 Java 实体类 BookItem,鼠标右键打开 Generate 对话框,如图 14-2 所示。

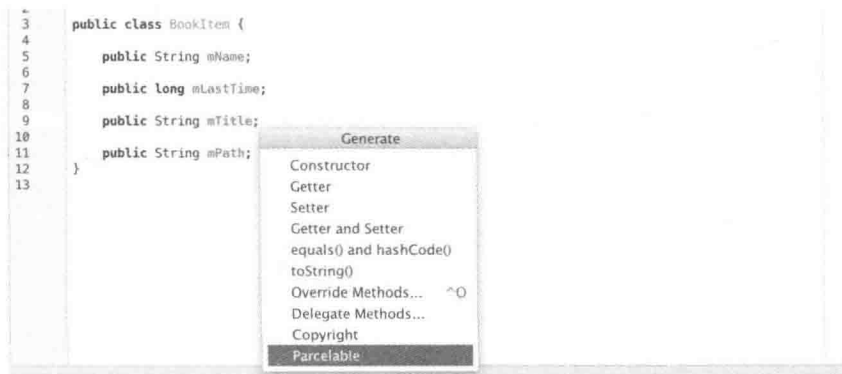


图 14-2

鼠标单击 Parcelable 按钮,该插件就自动帮我们将 BookItem 类转换成实现 Parcelable 接口的形式,免去开发者手动编写的麻烦,生成的代码如下。

```
public class BookItem implements Parcelable {

    public String mName;

    public long mLastTime;

    public String mTitle;

    public String mPath;

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.mName);
        dest.writeLong(this.mLastTime);
        dest.writeString(this.mTitle);
        dest.writeString(this.mPath);
    }

    public BookItem() {
    }

    protected BookItem(Parcel in) {
        this.mName = in.readString();
        this.mLastTime = in.readLong();
        this.mTitle = in.readString();
        this.mPath = in.readString();
    }

    public static final Parcelable.Creator<BookItem> CREATOR = new
```

```

Parcelable.Creator<BookItem>() {
    public BookItem createFromParcel(Parcel source) {
        return new BookItem(source);
    }

    public BookItem[] newArray(int size) {
        return new BookItem[size];
    }
};
}

```

可以看到，实现 Parcelable 接口，需要实现以下几个方法。

- describeContents：接口内容的描述，一般默认返回0即可。
- writeToParcel：序列化的方法，将类的数据写入到 Parcel 容器中。
- 静态的Parcelable.Creator接口，这个接口包含两个方法：
 - ❑ createFromParcel：反序列化的方法，将 Parcel 还原成 Java 对象。
 - ❑ newArray：提供给外部类反序列化这个数组使用。

14.3 SQLiteDatabase

SQLite 是一款轻量级的关系型数据库，它的运算速度极快、占用的资源很少——通常只需要几百 KB 的内存即可，特别适合在移动设备上使用，Android 和 iOS 都内置了 SQLite 数据库。

SQLite 数据库主要用来存储复杂的关系型数据，Android 系统原生支持 SQLite 数据库相关操作，但由于其 API 不友好，需要开发者编写很多样板代码，且容易出错，因此，开源社区出现了很多封装 SQLite 的 ORM 框架，相关介绍可参见本书第 20 章的内容。

安全性方面，由于 Android 应用程序数据库的默认目录是 /data/data/PACKAGE_NAME/database，而这个目录在手机 Root 之后是可以直接访问到的，因此敏感信息存储到数据库之前需要进行加密，使用时再进行解密，一个简单的方法是使用开源的 sqlcipher¹ 函数库来实现。sqlcipher 是一个开源的 SQLite 加密扩展，支持对数据库文件的 256 位 AES 加密。

¹ <https://github.com/sqlcipher/sqlcipher>

14.4 SharedPreferences

SharedPreferences 是 Android 平台提供的一个轻量级的存储 API，一般用来保存应用的一些常用配置信息，其本质是一个键值对存储。SharedPreferences 支持常用数据类型如 boolean、float、int、long 以及 String 的存储和读取。由于无需复杂的数据转换操作，SharedPreferences 相比其他序列化更高效。

使用 SharedPreferences 读取和存储数据，主要可以分为三步。

- 获取 SharedPreferences 对象。

```
SharedPreferences mPreferences = context.getSharedPreferences(PREFERENCES_
NAME, Context.MODE_PRIVATE);
```

- 通过 SharedPreferences 对象读取存储在 SharedPreferences 中的数据。

```
mPreferences.getBoolean(key, defValue);
```

- 获取 SharedPreferences.Editor 对象。

```
SharedPreferences.Editor mEditor = mPreferences.edit();
```

- 通过 SharedPreferences.Editor 对象写入数据到 SharedPreferences 中。

```
mEditor.putBoolean(key, b); // 写入 Boolean 类型数据
```

- 调用 commit 函数将写入的数据提交，从而完成数据存储操作。

```
mEditor.commit();
```

在 Android 系统中，SharedPreferences 中的信息是以 XML 文件的形式保存在 /data/data/PACKAGE_NAME/shared_prefs 目录中的。正常情况下，其他 APP 或者设备使用者是无法访问到这个目录的，但一旦 Android 设备 Root 之后，这个目录就可以轻易访问到了。因此，如果应用要保存敏感信息，在 SharedPreferences 中就必须进行加密存储。一般有两种方案。

- 在对数据进行存储之前进行加密，在读取后进行解密，可以使用 Java 加解密 API 实现，也可以使用成熟稳定的加解密开源库例如 Facebook 的 conceal¹来实现。
- 直接使用 SharedPreferences 的安全封装类 secure-preferences²，无须开发者手动编写加解

¹ <https://github.com/facebook/conceal>

² <https://github.com/scottyab/secure-preferences>

密部分代码，简单易用。

当然，上面两种方案都涉及到加解密所用的密钥如何安全的存储的问题，这一点可以参考本书第 40 章的内容。

14.5 JSON

JSON 的全称是 JavaScript Object Notation，它是一种轻量级的数据交换格式，Android SDK 原生支持 JSON 格式的解析和序列化。JSON 可以说是移动端使用最广泛的数据交换格式，几乎 80% 的 APP 与服务端的通信都是使用 JSON 格式。

由于 Android 原生 JSON 解析 API 性能很差，因此在开源界涌现了一系列简单易用且高性能的函数库，具体可以参考《基于开源项目搭建属于自己的技术堆栈》一文的相关介绍。

14.6 Protocol Buffers及Nano-Proto-Buffers

Protocol Buffers¹ 是 Google 设计的语言无关、平台无关的一种轻便高效的序列化结构数据存储格式，类似 XML，但更小、更快、更简单，很适合做数据存储或者 RPC 数据交换的格式。它可用于通讯协议，数据存储等领域的与语言无关，平台无关，可扩展的序列化结构数据格式。

如果你决定在项目中使用 Protocol Buffers，那么在移动端代码中应该使用 Nano-Proto-Buffers² 版本，因为普通的 Protocol Buffers 会生成非常冗余的代码，可能会增加 APP 内存占用，导致 APK 体积增长、性能下降，不注意的话，会很快遇到 64K 方法数限制问题。

14.7 FlatBuffers

Google FlatBuffers³ 是 Google 为游戏开发或其他对性能敏感的应用程序创建的开源的，跨平台的，高效的序列化函数库，它提供了对 C++/Java 等语言接口的支持。FlatBuffers 是一个注重性能和资源使用的序列化类库。相比较 Protocol Buffers 而言，它更适合移动设备。

1 <https://github.com/google/protobuf>

2 <https://github.com/google/protobuf/tree/master/javanano>

3 <https://github.com/google/flatbuffers>

FlatBuffers 的特性主要有：

- 序列化过程不需要打包和拆包：FlatBuffers 将序列化数据存储在缓存中，这些数据既可以存储在文件中，也可以通过网络进行传输，而无需其他任何解析开销，相比之下，Protocol Buffer 和 JSON 等均需要拆包和解包这两个步骤，FlatBuffers 的结构化数据以二进制形式保存，不存在数据解析的过程。
- 内存占用少、性能高：数据访问时唯一的内存需求就是缓冲区，不需要额外的内存分配。
- 强类型系统设计：在编译阶段就能够发现尽可能多的错误，而不是等到运行期才手动检查和修正。
- 支持跨平台：支持 C++/Java 等，不需要其他依赖库。

FlatBuffers 利用自身特殊的编码格式，在一定程度上减少内存的使用，优化了数据读取性能。同时，对于数据结构的前向后向兼容提供了良好的可扩展性。

第15章

Android WebView Java 和 JavaScript 交互详解

现代的移动应用几乎都是 Hybrid 方式，也就是集合了 Native APP 和 Web APP 的优点，既保证了用户体验，又使得 APP 在一定程度上具备动态更新的能力，同时有利于实现跨平台开发，减少人力成本，Hybrid 实现的关键点在于如何打通 Java 和 Javascript 之间的通信，主要包括两点：Java 如何调用 JavaScript；JavaScript 如何调用 Java。

在 Android 开发中我们是使用 WebView 组件来加载 HTML5 页面的，WebView 默认提供了让 Java 和 HTML5 页面中的 JavaScript 脚本交互的能力。

15.1 Java 调用 JavaScript

Java 调用 JavaScript 中的函数很简单，只需要执行如下代码即可。

```
mWebView.loadUrl("javascript:toast()");
```

其中，toast() 方法是 HTML5 页面中的 JavaScript 函数，语句如下。

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>演示 Java 调用 HTML5 页面中的 JavaScript 方法</title>
</head>
<script language="javascript">
```

```

<!--提供给 Android 的 Java 代码调用-->
function toast() {
    alert( "ascel885" );
}
</script>
<!--...-->
</html>

```

15.2 JavaScript 调用 Java

WebView 提供了一个名为 WebSettings 的工具类来实现让 WebView 中的 JavaScript 脚本调用 Android 应用的 Java 方法。实现起来很简单，只需要三个步骤。

- 调用与 WebView 关联的 WebSettings 实例的 setJavaScriptEnabled 方法来使能 JavaScript 调用的功能。
- 调用 WebView 的 addJavascriptInterface 方法将应用中的 Java 对象暴露给 JavaScript。
- 在 JavaScript 脚本中调用步骤二暴露出来的 Java 对象的方法。

下面以一个实例来演示，先来看一下要加载的 HTML5 页面。

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv=" Content-Type" content=" text/html; charset=utf-8" />
    <title>演示 JavaScript 调用 Android Java 方法</title>
</head>
<body>
    <!--其中的javaObject就是Android暴露出来的Java对象的名字-->
    <input type=" button" value=" 打印Log" onclick=" javaObject.
printLog( 'ascel885' );" />
    <input type=" button" value=" 显示Toast" onclick=" javaObject.
showToast( 'ascel885' );" />
</body>

```

```
</html>
```

接着看一下暴露给 JavaScript 调用的 Java 对象的定义。

```
public class JavaObject {
    private static final String TAG = "JavaObject";
    private Context mContext;

    public JavaObject(Context context) {
        mContext = context;
    }

    public void printLog(String message) {
        Log.d(TAG, message);
    }

    public void showToast(String message) {
        Toast.makeText(mContext, message, Toast.LENGTH_LONG).show();
    }
}
```

最后在 WebView 中的绑定操作语句如下。

```
WebSettings webSettings = mWebView.getSettings();
// 使能JavaScript
webSettings.setJavaScriptEnabled(true);
// 将JavaObject对象暴露给JavaScript脚本，在JavaScript中可以通过javaObject访问到这个对象
mWebView.addJavascriptInterface(new JavaObject(this), "javaObject");
```

上面这种 JavaScript 调用 Java 方法的方式虽然是官方提供的，但不幸的是，在 Android 4.2 之前的系统中，上面用法存在极大的安全隐患，会引起臭名昭著的 WebView 远程代码执行漏洞，具体可参见本书第 42 章的相关介绍。从 Android 4.2 开始，Google 修复了这个漏洞，我们可以安全地使用上述的方式，唯一需要修改的是对暴露给 JavaScript 调用的方法增加 @JavascriptInterface 注解，语句如下。

```
public class JavaObject {
    // 省略其他...
    @JavascriptInterface
```

```

public void printLog(String message) {
    Log.d(TAG, message);
}

@JavascriptInterface
public void showToast(String message) {
    Toast.makeText(mContext, message, Toast.LENGTH_LONG).show();
}
}

```

那么问题来了，在 Android 4.2 之前的系统版本中该如何规避这个安全隐患呢？答案是不再使用 `addJavascriptInterface` 这种方式，转而寻找其他的途径。我们知道，JavaScript 有三种常用的消息提示框，分别是：弹出警告框 `alert`、弹出确认框 `confirm` 和弹出输入框 `prompt`。对应到 Android 的 `WebChromeClient` 类，分别是以下三个方法。

```

public boolean onJsAlert(WebView view, String url, String message,
    JsResult result) {
    return false;
}

public boolean onJsConfirm(WebView view, String url, String message,
    JsResult result) {
    return false;
}

public boolean onJsPrompt(WebView view, String url, String message,
    String defaultValue, JsPromptResult result) {
    return false;
}

```

可以看到这三个方法参数唯一的区别是返回给 Javascript 的结果类型不一样，前两者是 `JsResult` 类型，这个类中带有有一个布尔类型的结果值；而 `onJsPrompt` 是 `JsPromptResult` 类型，它是 `JsResult` 的子类，带有有一个字符串类型的结果值，部分代码如下。

```

public class JsResult {
    private boolean mResult;

    @SystemApi

```

```

public final boolean getResult() {
    return mResult;
}

}

public class JsPromptResult extends JsResult {
    private String mStringResult;

    @SystemApi
    public String getStringResult() {
        return mStringResult;
    }
}

```

很显然，String 类型的结果值可以带上更多的信息，因此，我们选择 onJsPrompt 方法作为解决方案的突破口，通过这个方法，我们能够实现在 JavaScript 中将字符串信息（对应 onJsPrompt 入参中的 message）传递给 Java，而 Java 执行完成后也能够把返回结果的字符串形式（对应 onJsPrompt 的返回值 mStringResult）传递给 JavaScript。基本思路如下。

- 由于我们是通过字符串形式在 JavaScript 和 Java 之间进行通信的，因此，首先需要基于这个字符串定义好通信的协议，可以是 JSON 格式，这个字符串中可能会包含调用的类型 type、方法名 method、方法参数 args 等。
- 在 JavaScript 中封装一个方法，它通过最终调用 prompt 方法实现将上面的文本协议信息传递给 Java 层 WebChromeClient 类的 onJsPrompt 方法，在这个方法中对协议信息进行解析，可以得到类型、方法名、参数等信息，通过 Java 的反射机制可以实现调用到对应的 Java 方法。
- 步骤二的 Java 方法执行完毕后，同理，需要定义好返回值的协议格式，并通过 JsPromptResult 返回给 JavaScript。

当然，具体实现起来还是有很多其他工作需要做的，想要应用到线上项目中，更少不了各种测试。好消息是，国内开发者 pedant¹ 基于上述方案已经实现了一个健壮可靠的开源函数库，名为 safe-java-js-webview-bridge^[2]，我们可以直接拿来用。

¹ <https://github.com/pedant>



第2篇 系统架构篇

- ★ 第 16 章 MVP 模式及其在 Android 中的实践
- ★ 第 17 章 MVVM 模式及 Android DataBinding 实战
- ★ 第 18 章 观察者模式的拓展：事件总线
- ★ 第 19 章 书写简洁规范的代码
- ★ 第 20 章 基于开源项目搭建属于自己的技术堆栈

第16章

MVP模式及其在Android中的实践

Android 应用开发的早些年间，一个 APP 的整体架构并没有得到很好的重视，毕竟当时懂 Android 开发的人并不多，资深的开发者更是少之又少，大家的主要精力都集中在如何更好地使用 Android SDK 提供的 API，来完成 APP 的功能需求。随着多年以来的发展和积累，Android 应用开发的 UI 架构模式历经了 MVC、MVP 到 MVVM 的演进。特别是近一两年，MVP 模式更是受到热烈的追捧，虽然没能像 Java Web 开发领域的 SSH 那样形成统一的标准或者开发框架，但 MVP 的思想已经得到了普及，也涌现了一些不错的开源框架实现。

UI 架构模式是面向开发者的，它在一定程度会存在性能的损耗，但好处是代码具有更高的可阅读性、可测试性、可维护性以及可复用性。

16.1 MVP 的基本概念

传统的 Android 应用开发中，View 层（Activity，Fragment 或者自定义 View）承载了太多的责任，它不仅要完成界面的更新、复杂动画的渲染等 UI 相关的操作，还要处理各种业务逻辑，例如从网络获取数据、将用户输入保存到本地数据库中。由于职责不单一，View 层的代码往往显得很庞大，一个 Activity 或者 Fragment 的代码行数可能要好几千行。这种模式显然不是长久之计，随着一个类的代码量逐渐增加，维护和升级将变得越来越困难，牵一发而动全身。为了更好的组织并对代码进行分层设计，我们有必要引入 MVP 模式。

MVP 的全称是 Model、View、Presenter，顾名思义，它将整个应用分为三层，如图 16-1 所示。

- View 层：视图层，包含界面相关的功能，例如各种 Activity、Fragment、View、Adapter 等，该层专注于用户的交互，实现设计师给出的界面，动画等交互效果。View 层一般会持有 Presenter 层的引用，或者也可以通过依赖注入（例如 Dagger）的方式获得 Presenter 的实例，并将非 UI 的逻辑操作委托给 Presenter。
- Presenter 层：逻辑控制层，充当中间人的角色，用来隔离 View 层和 Model 层，该层是通过从 View 层剥离控制逻辑部分而形成的，主要负责 View 层和 Model 层的控制和交互。例如接收 View 层的网络数据加载请求，并分发给对应的 Model 处理，同时监听 Model 层的处理结果，最终将其反馈给 View 层，从而实现界面的刷新。
- Model 层：封装各种数据来源，例如远程网络数据，本地数据库数据等，对 Presenter 层提供简单易用的接口。

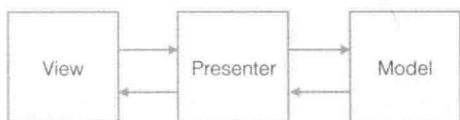


图16-1

16.2 MVP 与 MVC 的区别

MVP 是经典的 MVC 的延伸和改进，MVC 的关系图解如图 16-2 所示，和 MVP 的相比，可以看出最大的不同在于。

- MVP 中 View 层和 Model 层并没有直接通信，而是通过中间人 Presenter 来间接通信；Presenter 和 View 以及 Model 的交互都是通过接口进行的；通常 View 与 Presenter 是一对一的，当然，复杂的 View 可能需要多个 Presenter 来共同处理，这些需要根据具体的业务需求而定。
- MVC 中 Model 层和 View 层是直接通信的，而且 Controller 是基于行为的，可以被多个 View 共享。

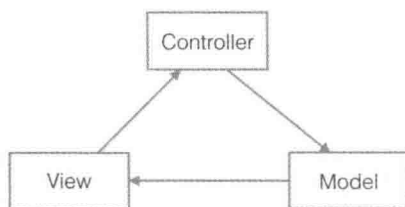


图 16-2

16.3 MVP 的开源实现

16.3.1 Android-Architecture¹

Google 官方出品的 Android UI 架构的一系列例子，几乎都是 MVP 架构的实现，开发者可以根据自己的业务需求进行选择和参考，下面分别进行简单介绍。

16.3.2 TODO-MVP²

MVP 架构的基本实现，没有使用任何其他的架构框架，通过纯手工的依赖注入实现从本地数据源和远程数据源获取数据的 Repository，使用回调方法实现异步任务，如图 16-3 所示。

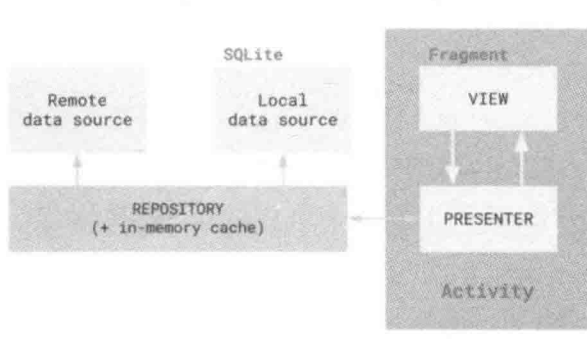


图 16-3

¹ <https://github.com/googlesamples/android-architecture>

² <https://github.com/googlesamples/android-architecture/tree/todo-mvp/todoapp>

16.3.3 TODO-MVP-Loaders¹

在 TODO-MVP 的基础上，通过 Loaders 机制从 Repository 中获取数据，使用 Loaders 机制的好处如下。

- 提供异步加载数据的能力，因此 Repository 不需要提供回调方法。
- 监听 Repository 的数据变化，并在数据变化时发送新的结果。
- 在配置变化导致界面重建时，Loaders 机制会重新与最后的 Loader 建立连接。

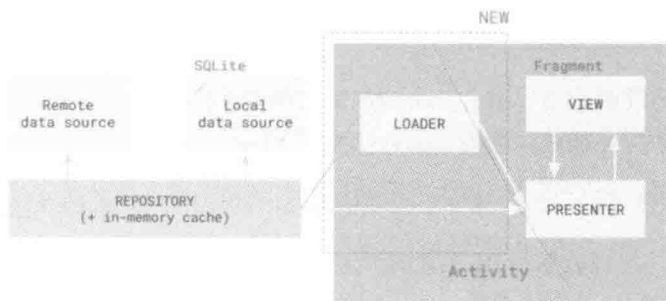


图 16-4

16.3.4 TODO-MVP-Clean²

在 TODO-MVP 的基础上，参考 Clean 架构³的思想，在表现层和 Repository 之间增加了一个 Domain 层，在整体上将 APP 分为了三个层次。

- MVP 层：也称为表现层，包含 View 和 Presenter。
- Domain 层：业务逻辑层，提供名为 use cases 或者 interactors 的类来表示所有可能从 Presenter 发起的动作。
- Repository 层：数据存储层。

¹ <https://github.com/googleamples/android-architecture/tree/todo-mvp-loaders>

² <https://github.com/googleamples/android-architecture/tree/todo-mvp-clean>

³ <https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>

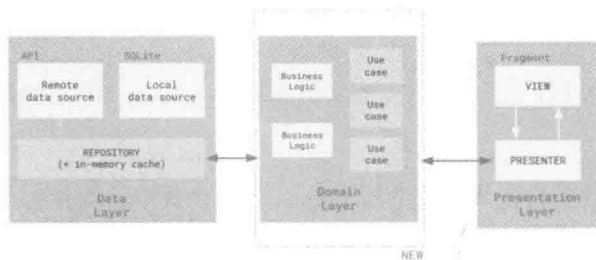


图16-5

16.3.5 TODO-Databinding¹

在 TODO-MVP 的基础上，结合 Data Binding² 函数库实现视图和数据的绑定，它并没有严格遵循 MVP 或者 MVVM 模式，因为它结合使用了 ViewModels 和 Presenters。

Data Binding 函数库的引入，减少了很多样板代码的使用，实现 UI 元素和数据的绑定。

- 使用布局文件实现数据和 UI 元素的绑定。
- 事件也会和一个动作处理器绑定在一起。
- 监听数据并在需要时自动更新。

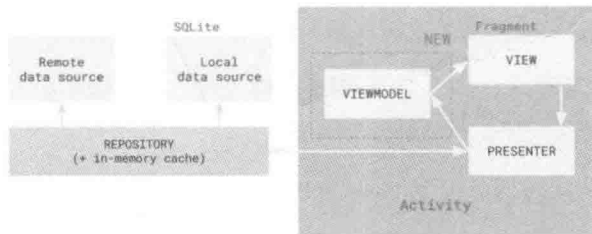


图16-6

16.3.6 其他开源参考实现

在 Google 官方的 MVP 开源实现参考发布之前，其实已经有很多开发者发布了自己的 MVP 实

¹ <https://github.com/googlesamples/android-architecture/tree/todo-databinding>

² http://developer.android.com/tools/data-binding/guide.html#data_objects

现,其中值得关注的有 [androidmvp](https://github.com/antonioleal/androidmvp)¹、[mosby](https://github.com/sockeqwe/mosby)²、[nucleus](https://github.com/konmnik/nucleus)³、[EffectiveAndroidUI](https://github.com/pedrovs/EffectiveAndroidUI)⁴、[MvpCleanArchitecture](https://github.com/glomadrian/MvpCleanArchitecture)⁵等。

16.4 MVP 的好处

使用 MVP 组织代码架构,并对代码实施分层管理,有以下好处。

- 如果界面发生变化,甚至是全新改版,只需修改对应的 View 即可,Presenter 和 Model 层无需改动。
- 如果业务逻辑或者数据获取方式发生变化,只需要修改对应的 Model。
- 如果控制逻辑发生变化,只需修改对应的 Presenter。
- Presenter 层和 View 层以及 Model 层的交互都是基于接口实现的,这有助于对 Presenter 进行单元测试,同时由于是面向接口编程,只需事先定义好接口,每一层的实现可以交由不同的开发人员并行实现,最终再一起联调,能够明显加快某一功能的开发进度。
- 团队的新成员拿到项目的代码,能够很容易的读懂现有的逻辑,快速上手。
- 如果你正在开发一个对外的 SDK,根据市场需求,需要提供带 UI 版本和不带 UI 的纯接口版本,那么使用 MVP 模式,将 UI 部分代码放在 View 层,将接口部分代码放在 Model 层,打包的时候可以轻松实现是否将 View 层打包进去,从而避免纯接口版本混入 UI 相关的代码。

16.5 MVP 存在的问题

- 增加代码类的数量。
- 由于进行了三层划分,函数的调用栈变深了,如果开发人员没能非常清楚的了解哪一层具体该负责哪些功能,那么可能存在因为层次职责辨认不清等原因导致不同层之间的代码乱入,从而没能达到 MVP 充分解耦各层的目的。

1 <https://github.com/antonioleal/androidmvp>

2 <https://github.com/sockeqwe/mosby>

3 <https://github.com/konmnik/nucleus>

4 <https://github.com/pedrovs/EffectiveAndroidUI>

5 <https://github.com/glomadrian/MvpCleanArchitecture>

第17章

MVVM模式及Android DataBinding实战

MVVM (Model — View — ViewModel) 最初是在 2005 年由微软提出的一个 UI 架构概念。相比 MVP 模式, MVVM 将 Presenter 改为了 ViewModel, 同时实现 View 和 ViewModel 的双向绑定。View 层的变化会自动导致 ViewModel 发生变化, ViewModel 的数据变化也会自动实现 View 的刷新, 开发者可以不用直接处理 View 和数据的更新操作, MVVM 框架会完成这一切, MVVM 模式不同层之间关系如图 17-1 所示。

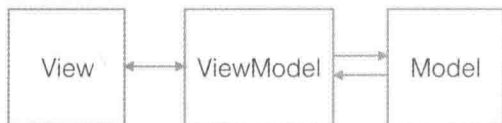


图17-1

在 Google I/O 2015 大会上, Android 开发团队发布了官方的 MVVM 模式支持函数库 Data Binding Library。Data Binding Library 是一个兼容函数库, 可以在 Android 2.1 (API Level 7) 及之后的 Android 系统上面使用。在使用 Data Binding 之前, 需要确保 Gradle 的 Android Studio 插件版本大于或等于 1.5.0-alpha1, 而且 Android Studio 的版本号应该大于或等于 1.3。

Data Binding 函数库的引入很简单, 在使用到 Data Binding 的 android module 的 build.gradle 文件中加入下面的配置即可, Android Studio 将自动为我们下载所需的依赖库。

```
android {  
    ....  
    dataBinding {
```

```

        enabled = true
    }
}

```

17.1 Data Binding 表达式

Data Binding 的布局文件和普通的布局文件有些许不同，它以 `<layout>` 作为根布局标签，里面又包含 `data` 和 `view` 两个标签，其中 `data` 标签用来实现数据绑定，`view` 标签就是在没有使用 Data Binding 时这个页面的布局文件的根标签，语句如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<layout xmlns:android="http://schemas.android.com/apk/res/android" >
    <data>
        <variable name="user" type="com.example.User" />
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}" />
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.lastName}" />
    </LinearLayout>
</layout>

```

`data` 标签内定义了一个名为 `user` 的属性变量，类型是名为 `User` 的 Java 类，在 `layout` 标签中使用 `@{user.firstName}` 来将某个控件的值和 `user` 的成员变量绑定在一起。

17.2 数据对象

上面代码中的 `user` 变量称为数据对象，它可以是如下所示的 POJO 类。

```
public class User {  
    public final String firstName;  
    public final String lastName;  
    public User(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

也可以是如下所示的 JavaBeans 对象。

```
public class User {  
    private final String firstName;  
    private final String lastName;  
    public User(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    public String getFirstName() {  
        return this.firstName;  
    }  
    public String getLastName() {  
        return this.lastName;  
    }  
}
```

对于 Data Binding 而言，上面这两种类的定义是等价的，`@{user.firstName}` 表达式既可以访问到公有的 `firstName` 属性，也可以通过 `getFirstName()` 访问到私有的 `firstName` 属性。当然，如果存在名为 `firstName()` 的方法也是可以访问到的。

17.3 数据绑定

默认情况下，Binding 类的命名以布局文件名加上 Binding 作为后缀组合而成，例如布局文

件名为 `main_activity.xml`，那么生成的 `Binding` 类名为 `MainActivityBinding`。这个类包含了所有的映射，并会自动帮我们实现数据绑定，数据绑定的用法很简单，语句所下。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    MainActivityBinding binding = DataBindingUtil.setContentView(this,
R.layout.main_activity);
    User user = new User("Test", "User");
    binding.setUser(user);
}
```

这样就完成了数据绑定，当然我们也可以通过 `inflate` 方式。

```
MainActivityBinding binding = MainActivityBinding.inflate(getLayoutInflater());
```

如果我们在 `ListView` 或者 `RecyclerView` 的 `adapter` 中使用数据绑定，那么也可以如下方式实现绑定。

```
ListItemBinding binding = ListItemBinding.inflate(layoutInflater, viewGroup,
false);
//or
ListItemBinding binding = DataBindingUtil.inflate(layoutInflater, R.layout.
list_item, viewGroup, false);
```

17.4 事件绑定

在 `Data Binding` 的布局文件中我们也可以进行事件的绑定，类似于 `android:onClick` 可以绑定 `Java` 方法。我们的事件处理类定义如下。

```
public class MyHandlers {
    public void onClickFriend(View view) { ... }
    public void onClickEnemy(View view) { ... }
}
```

在 XML 文件中的绑定代码如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<layout xmlns:android="http://schemas.android.com/apk/res/android" >
    <data>
        <variable name="handlers" type="com.example.Handlers" />
        <variable name="user" type="com.example.User" />
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"
            android:onClick="@{user.isFriend ? handlers.onClickFriend :
handlers.onClickEnemy}" />
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.lastName}"
            android:onClick="@{user.isFriend ? handlers.onClickFriend :
handlers.onClickEnemy}" />
    </LinearLayout>
</layout>
```

第18章

观察者模式的拓展：事件总线

观察者模式是四人帮的《设计模式 可复用面向对象软件的基础》一书中提出的经典设计模式之一，也是使用最为广泛的模式之一。在软件开发领域几乎无人不晓，无论在后端编程，Web 编程还是移动端编程，都有很多应用和变体。《设计模式》一书对观察者模式的描述是：“定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新”。本章就来讲一下观察者模式的一个拓展：事件总线。继承自观察者模式，事件总线也是基于发布订阅的机制来实现事件的发送和接收的。

18.1 为何要使用

Android 应用开发过程中，经常会涉及 Activity、Fragment、Service 等不同组件或者模块之间的消息传递，使用传统的方法实现，往往会写出丑陋的代码，而且不同组件和模块之间耦合严重。随着模块的日益增多、代码逻辑的不断新增和修改，整个代码的架构就会显得越来越混乱，一个模块的改动可能会引起其他模块的连锁反应，为了便于理解，下面举例说明。

例子一：Activity 中不同 Fragment 之间需要进行通信，传统的做法是将 Activity 作为中介，Fragment A 通过 `getActivity()` 获取宿主 Activity 实例进而可以拿到 Fragment B 的实例，从而向 Fragment B 发送消息或者获取数据；好一点的做法是在 Fragment 中编写接口，让宿主 Activity 实现该接口，从而在 Activity 中实现不同 Fragment 之间的数据通信。

例子二：多个 Activity 页面跳转和数据回传的问题，例如 Activity A 跳转到 Activity B，接着跳转到 Activity C，在 C 中执行一系列操作后，需要传递数据或者事件给 Activity A，传统的做法是进行接口回调，这样不仅增加逻辑负责性，而且增大页面间耦合。

为了解决以上问题，实现组件间和模块间的解耦，我们引入了事件总线的概念。

18.2 原理

事件总线，顾名思义，是消息或者说事件流动的管道，不同组件和模块之间的消息传递都是通过总线来实现，组件与组件、模块与模块之间不直接进行通信，一言以蔽之，事件总线就是用来简化 Android 应用中组件或者模块间的通信，从而实现模块间解耦的目的。

事件总线是基于观察者模式的思想实现的，它使用发布订阅的方式支持组件和模块间的通信，摒弃了观察者模式需要显式注册回调的缺点，同时可用于替换 Java 中传统的事件监听方式。事件总线涉及到的关联方如下。

- 事件 Event：一个普通的 POJO 类，只包含数据，不包含对数据的操作。事件有两种类型：普通事件和粘滞事件，粘滞事件的特点是在事件发布后，订阅者才开始订阅该类型事件，那么它依然可以收到这个事件，而普通事件是收不到的。
- 订阅者 Subscriber：订阅某种类型事件的对象，通常会有一个回调函数用于对接收到的事件进行处理，订阅者可以订阅事件，也可以取消订阅的事件，订阅者可以引入优先级的概念，优先级高的订阅者可以优先接收到该事件，并可以决定是否继续传递事件给低优先级的订阅者。
- 发布者 Publisher：事件的源头，发布某种类型事件的对象。
- 总线 EventBus：负责订阅者，事件等信息的存储，同时处理事件的流动和分发，通过总线，订阅者和发布者是解耦的，互相不知道对方的存在。

事件总线的总体架构可以借用 EventBus 这个库的官网截图，如图 18-1 所示。

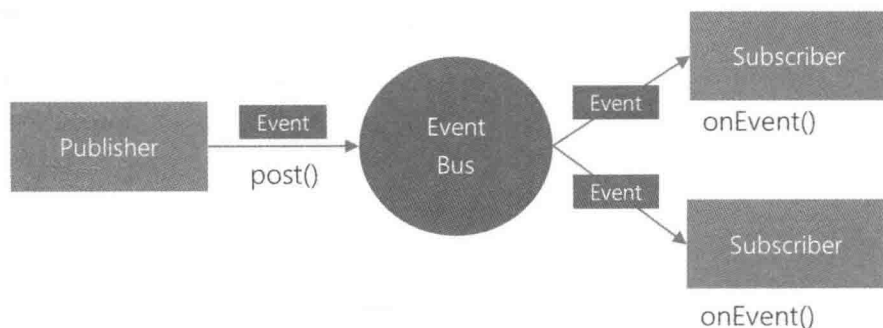


图 18-1

18.3 开源实现

Android 事件总线的开源实现有很多，例如 Google 出品的 Guava 库中就包含一个 EventBus 模块，但由于 Guava 库很庞大，为了避免引入太多无用代码，一般都不建议使用这个库；最使用广泛的主要有 greenrobot 的 EventBus、square 的 otto，当然，国内也有一些开源实现，例如 bboyfeiyu 的 AndroidEventBus¹、mexiaokey 的 xBus²，前者吸收了 greenrobot 的 EventBus 以及 square 的 otto 的优点，并在此基础上做出了相应的改进，使得事件总线框架更适合用户的使用习惯，也使得事件的投递更加的精准、灵活。后者是事件总线的简单实现。下面主要介绍 EventBus 和 otto 这两个代表性的实现。

18.3.1 EventBus³

EventBus 是一个专门为 Android 平台优化定制的事件总线函数库，出于性能考虑，没有使用 Java 注解，因为在 Android 平台上，查询注解是很慢的，尤其是在 Android 4.0 之前的系统上，详情可参见 Google 的这个 Bug 报告⁴。

Android Studio & Gradle 中使用 EventBus 只需如下简单的几个步骤。

在 build.gradle 中添加在线依赖如下。

```
compile 'de.greenrobot:eventbus:2.4.0'
```

定义事件类。

```
public class MessageEvent {
    public String mType;
    public String mContent;
    // ...
}
```

注册事件订阅者，在处理事件的类的合适位置进行事件的订阅和取消订阅，同时实现名为 onEvent 的回调函数。

¹ <https://github.com/bboyfeiyu/AndroidEventBus>

² <https://github.com/mexiaokey/xBus>

³ <https://github.com/greenrobot/EventBus>

⁴ <http://code.google.com/p/android/issues/detail?id=7811>

```
// 把当前类注册为订阅者, 如果是Activity类, 那么可以放在onCreate函数中
EventBus.getDefault().register(this);

// 解除注册当前类 (一定要调用, 否则会内存泄漏), 如果是Activity类, 可以放在onDestroy函数中
EventBus.getDefault().unregister(this);

public void onEvent(Object event) {
    // ...
}
```

在任何一个类中, 如果产生了某个事件, 那么可以通过如下代码进行事件的发布, 这样订阅者就可以收到这个事件的广播了。

```
/**
 * 这里的event类型必须和上面的onEvent()方法的参数类型一致
 */
EventBus.getDefault().post(event);
```

EventBus 基于性能考虑没有使用注解, 而是使用“名称约定优于配置”的思想, 导致的缺点如下。

- 接收事件的函数必须以 onEvent 开头。
- 一般每个事件对应一个 Event 类, 会产生很多样板类, 从而增加 APP 出现 64K 方法数问题的可能。
- 用于接收 EventBus 事件的类不能混淆, 否则会找不到 onEvent 函数。

在本书成稿时, EventBus 已经发布了 3.0 Beta 版本¹, 这个版本使用编译时注解代替“约定优于配置”的思想, 接收事件的函数不再需要以 onEvent 开头, 只要使用注解 @Subscribe 标注即可, 同时较 2.4.0 版本提升了性能, 如图 18-2 所示。

让我们期待 3.0 正式版的发布吧!

18.3.2 otto²

otto 是在 Guava 的 EventBus 模块基础上针对 Android 平台增强定制的事件总线函数库, 和

¹ <https://github.com/greenrobot/EventBus/tree/annotations>

² <https://github.com/square/otto>

greenrobot 的 EventBus 最大的不同是它使用运行时注解的方式，在性能上较 EventBus 差，但使用上比 EventBus 灵活。同时功能方面相比 EventBus 也少了很多特性，从 EventBus 的官网可以看到，无论从功能特性还是性能，EventBus 都比 otto 优秀，如图 18-3 功能对比图、图 18-4 性能对比图所示。

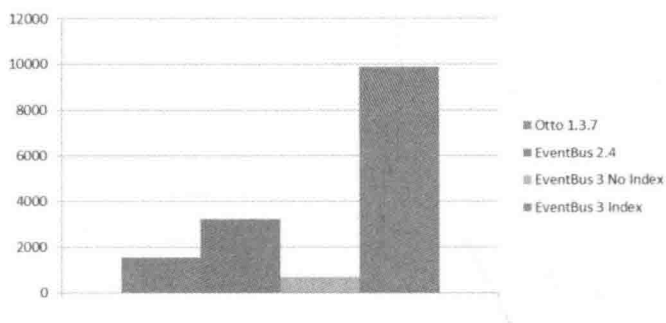


图18-2

	EventBus	Otto
Declare event handling methods	Name conventions	Annotations
Event inheritance	Yes	Yes
Subscriber inheritance	Yes	No
Cache most recent events	Yes, sticky events	No
Event producers (e.g. for coding cached events)	No	Yes
Event delivery in posting thread	Yes (Default)	Yes
Event delivery in main thread	Yes	No
Event delivery in background thread	Yes	No
Asynchronous event delivery	Yes	No

图18-3

	EventBus	Otto
Posting 1000 events, Android 2.3 emulator	~70% faster	
Posting 1000 events, S3 Android 4.0	~110% faster	
Register 1000 subscribers, Android 2.3 emulator	~10% faster	
Register 1000 subscribers, S3 Android 4.0	~70% faster	
Register subscribers cold start, Android 2.3 emulator	~350% faster	
Register subscribers cold start, S3 Android 4.0	About the same	

图18-4

从 otto 官网可以看到如图 18-5 所示的提示。

Deprecated!

This project is deprecated in favor of RxJava and RxAndroid. These projects permit the same event-driven programming model as Otto, but they're more capable and offer better control of threading.

图 18-5

可以看到, otto 已经标记为过时了, 将不再提供新功能和特性, 推荐使用 RxJava¹ 来代替它。当然, RxJava 虽然和事件总线同样都是基于观察者模式实现的, 但功能职责不尽相同, 事件总线主要适合组件和模块间通信使用, 而 RxJava 的使用场景更多的是异步数据流的处理, 基于函数响应式编程方式。

18.4 与观察者模式及 Android 广播的区别

虽然可以使用 Android 的 BroadcastReceiver 来实现各种事件的监听, 但它更适合用于监听 Android 系统级的广播事件, 例如网络状态变化、电量变化等, 对于业务相关的事件变化, 使用 BroadcastReceiver 太重量级了, 而且使用起来也很不方便, 最佳实践应该是把 BroadcastReceiver 限定于监听系统级别的广播事件。

观察者模式用于简单的事件监听没有问题, 但如果 APP 全局都使用观察者模式来解决组件和模块间的消息通信, 那么可能会造成接口膨胀的问题。观察者模式要求开发者自己实现事件的生成、分发和处理, 需要进行很好的设计。同时, 观察者模式不支持粘滞事件, 不支持事件优先级等特性。当然, 性能上来说, 观察者模式要比事件总线性能高, 毕竟它不需要处理很多其他的特性。

事件总线使用起来很方便, 但我们不应该滥用它, 需要严格限定它的使用范围, 只有在组件或者模块间通信时才使用它。对于简单的消息传递, 就选用观察者模式或者事件回调的方式即可。

¹ <https://github.com/ReactiveX/RxJava>

第19章

书写简洁规范的代码

写代码如同写文章，正所谓文如其人，代码可以说是开发者的脸面，写出简洁规范的代码对个人，对团队都是非常重要的。对个人而言，简洁规范的代码不仅体现自己的专业性和技术水平，而且能够赢得别人的赞赏；对团队而言，团队成员遵循统一的规范，能够更好地维护代码库的稳定和谐。

Android 是基于 Java 语言进行开发的，因此说到编码规范，首先需要遵循 Java 的编码规范，比较有名的 Java 编码规范有 Google 的 Java 编码规范¹和 Oracle 的 Java 编码规范²。一份完整的编码规范不仅关注编码格式是否美观，同时也会讨论一些约定和编码标准。

19.1 Java 编码规范

Java 编码规范一般包含的关注点主要包括如下几点。

19.1.1 源代码文件的定义

源代码文件以文件内容中的最顶层的 Java 类命名，而且大小写敏感，文件扩展名为 .java，同时，文件的编码格式统一为 UTF-8。

19.1.2 源代码文件的结构

一个完整的源代码文件由四部分组成。

1 <http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

2 <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

- 版权信息或者许可证，例如 Android Framework 的版权声明如下。

```

/*
 * Copyright (C) 2008 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

```

- Java 文件所在 package 的声明，例如 package android.net。
- Java 类需要使用到的依赖的引入 import 语句，一般不建议使用通配符，例如 import android.annotation.*;，而应该引入具体的类，例如 import android.annotation.SdkConstant;。同时 import 语句要按照类型进行分组，不同的组以空行分隔，例如静态导入的类要独立成组，第三方函数库引入的类也独立成组等，一般使用 IDE 的格式化功能会自动帮我们分好组。
- Java 顶级类的定义，有且只有一个，但可以存在内部类定义。每个类中的成员需要按照某种逻辑进行排序，维护者也需要按照相同的逻辑去添加新的成员，而不是仅仅将新的成员放到最后面，同时，多个重载方法应该按顺序排放在一起，中间不要插入其他方法。

19.1.3 遵循的格式

格式主要涉及代码的排版问题，需要重点关注的主要包括。

- 多使用花括号：例如 if、else、for、do、while 等语句要和花括号一起使用，即使只有

一条语句或者是空的，也要加上花括号，下面的写法是错误的，而且可能会在后面的代码维护中引入 bug。

```
if (isLogin)
    login();
else
    unlogin();
```

正确的做法是加上花括号。

```
if (isLogin) {
    login();
} else {
    unlogin();
}
```

- 列字符个数的限制：根据团队内部使用的显示器以及 IDE 具体问题具体分析，最终目的是保证代码良好的可阅读性，一般常见的是约定一行 80 个或 100 个字符，超过部分 IDE 自动换行。
- 空白的使用：不要把所有的代码都写在一起，需要按照逻辑进行分组，垂直方向的空白原则主要有：（1）方法体内，语句的逻辑分组之间使用空行；（2）类里面连续的成员（字段、构造方法、普通方法、嵌套内部类、静态初始化代码块等）之间加上空行。当然，也没有必要加上多个空行，一个就够了。
- switch 语句：switch 语句除了要注意缩进，空白的使用之外，如果连续两个 case 之间明确不需要加 break 语句时，建议加上 // fall through 注释，方便代码维护者的理解，同时一定要加上 default 语句，一个符合规范的 switch 语句如下。

```
switch (input) {
    case 1:
    case 2:
        prepareOneOrTwo();
        // fall through
    case 3:
        handleOneTwoOrThree();
        break;
```

```
default:
    handleLargeNumber(input);
}
```

- 修饰符的顺序：类和成员的修饰符如果存在多个的话，需要按照 Java 语言规范中的定义排序，语句如下。

```
public protected private abstract static final transient volatile synchronized native strictfp
```

19.1.4 命名约定

类的命名遵循大驼峰命名法 UpperCamelCase，而方法名和变量名的命名遵循小驼峰命名法 lowerCamelCase。常量名使用大写字母表示，单词之间以下划线分隔，例如 `static final int CONNECTION_TIMEOUT = 10000`。

19.1.5 Javadoc

标准的 Javadoc 常见的标记和含义如下。

```
/**
 * Javadoc常见的标记
 *
 * @param 方法参数的说明
 * @return 对方法返回值的说明
 * @throws 方法抛出异常的描述
 * @version 模块的版本号
 * @author 模块的作者
 * @see 参考转向
 * @deprecated 标记是否过时
 */
```

19.2 Android 命名规范

19.2.1 布局文件的命名

布局文件的命名规则使用“前缀_逻辑名”的方式，单词全部小写，例如：

- Activity 的布局文件命名为 `activity_XXX`。
- Fragment 的布局文件命名为 `fragment_XXX`。
- 自定义控件的布局文件命名为 `view_XXX`。
- 对话框的布局文件命名为 `dialog_XXX`。
- 列表项的布局文件命名为 `item_XXX`。

19.2.2 资源文件的命名

资源文件的命名规则使用 前缀_模块名_逻辑名称 的方式，单词全部小写，例如：

- 按钮的命名以 `btn` 作为前缀，例如 `btn_login.png`，当按钮存在多种形态时，需要加上按钮的形态，例如 `btn_login_normal.png`、`btn_login_pressed.png` 等。
- 图标命名以 `ic` 作为前缀，例如 `ic_share.png`。
- 背景图片的命名以 `bg` 作为前缀，例如 `bg_main.png`。
- 分隔线的命名以 `divider` 作为前缀，例如 `divider_gray.png`。

19.2.3 类的命名

类的命名遵循 Java 的类命名规范，也就是使用大驼峰命名法，同时需要根据类的具体用途引入 Android 相关的命名规则，例如：

- Activity 类需要以 `Activity` 作为后缀，例如 `MainActivity`。
- Fragment 类需要以 `Fragment` 作为后缀，例如 `HomeFragment`。
- Service 类需要以 `Service` 作为后缀，例如 `DownloadService`。
- BroadcastReceiver 类需要以 `Receiver` 作为后缀，例如 `PushReceiver`。
- ContentProvider 类需要以 `Provider` 作为后缀，例如 `ContactProvider`。
- 工具类需要以 `Util` 作为后缀，例如 `NetworkUtil`。
- 自定义的公共基础类以 `Base` 开头，例如 `BaseActivity`。

- 单元测试的类以 Test 作为后缀，例如 HashTest。

19.3 CheckStyle 的使用

在 Android Studio 中，我们可以引入 CheckStyle 插件来进行编码规范的检查，每个团队根据自身的编码规范定制 CheckStyle 的规则，然后可以加入到持续构建平台中，定期扫描提交的代码是否符合规范，并给出报告，CheckStyle 的配置可以参见本书第 49 章。

第20章

基于开源项目搭建属于自己的技术堆栈

在 Android 面试过程中，作为面试官的我一般会问面试者这样一道题目：聊聊你所理解的在 Android 应用开发过程中，几乎每个 APP 都具有的与业务无关的底层基础能力，或者说如果你从头开始一个新项目，你如何搭建这个 APP 的基础框架。这个问题本质上是考察面试者的 Android 技术选型以及整体架构方面的能力。

Android 发展到现在，已经处于成熟稳定期，相信每个靠谱的移动互联网公司都已经搭建起自己的一套通用的基础开发框架，服务于公司内部各大产品线的移动端产品。有一定经验的 Android 开发者，也都或多或少在心中或者实践中有自己的一套基础开发工具集。无论是公司还是个人，这套基础开发框架中肯定都包含开源的函数库或者第三方的 SDK 以及自己研发的或者总结的函数库，只不过比例不同而已。为了从整体架构上进行把握，我们先来看看一个完整的 APP 整体架构。

20.1 APP 的整体架构

从较高的层次讲，一个 APP 的整体架构可以分为两层，即应用层和基础框架层。

- 应用层专注于行业领域的实现，例如金融、支付、地图导航、社交等，它直接面向用户，是用户对产品的第一层感知。
- 基础框架层专注于技术领域的实现，提供 APP 公有的特性，避免重复制造轮子，它是用户对产品的第二层感知，例如性能、稳定性等。

一个理想的 APP 架构，首先应该是支持跨平台开发的；其次应该具有清晰的层次划分，同一层模块间充分解耦，模块内部符合面向对象设计六大原则；最后应该在功能、性能、稳定性等方面达到综合最优。基于以上设计原则，我们可以得到如图 20-1 所示的 APP 架构图，最上层是应用层，应用层以下都属于基础框架层，可以看到基础框架层包括：组件层、基础层和跨平台层。



图 20-1

本章重点就是其中的基础层，下面就开始一步一步地阐述如何基于开源函数库搭建属于自己的一个基础技术堆栈。

20.2 技术选型的考量点

首先要明确的是，我们选择开源函数库或者第三方 SDK，一般需要综合考虑以下几个方面。

- 特性：提供的特性是否满足项目的需求。
- 可用性：是否提供了简洁便利的 API，方便开发者集成使用。
- 性能：性能不能太差，否则项目后面性能优化会过不去，可能会出现需要替换函数库的情况。
- 文档：文档应该比较齐全，且可读性强。
- 技术支持：遇到问题或者发现 Bug，是否能够及时得到官方的技术支持是很重要的。
- 大小：引入函数库会增加 APK 的大小，需要慎重抉择。
- 方法数：如果函数库方法数太多，积累起来会导致你的 APP 遇到 64K 问题，应该尽量避免。

20.3 日志记录能力

日志记录无论在服务端开发还是移动端开发，都是一个基础且重要的能力。开发人员在代码调试以及错误定位过程中，大多数时候都要依赖日志信息，一个简洁灵活的日记记录模块是相当重要的。Android 系统提供了 Log 类用来记录日志，在 Android 开发最初的几年，我们几乎都是对这个 Log 类进行简单的封装，例如增加全局控制是否打印日志的开关、增加打印到文件的能力等。直到我们遇到了 Logger¹ 这个开源日志记录库，才认识到日志记录还可以这么玩。Logger 同样是基于系统 Log 类基础上进行的封装，但新增了如下超赞的特性。

- 在 Logcat 中完美的格式化输出，再也不用担心和手机其他 APP 或者系统的日志信息相混淆了。
- 包含线程、类、方法信息，可以清楚地看到日记记录的调用堆栈。
- 支持跳转到源码处。
- 支持格式化输出 JSON、XML 格式信息。

¹ <https://github.com/orhanobut/logger>

给出一张官网 Logcat 截图大家就清楚了，如图 20-2 所示。

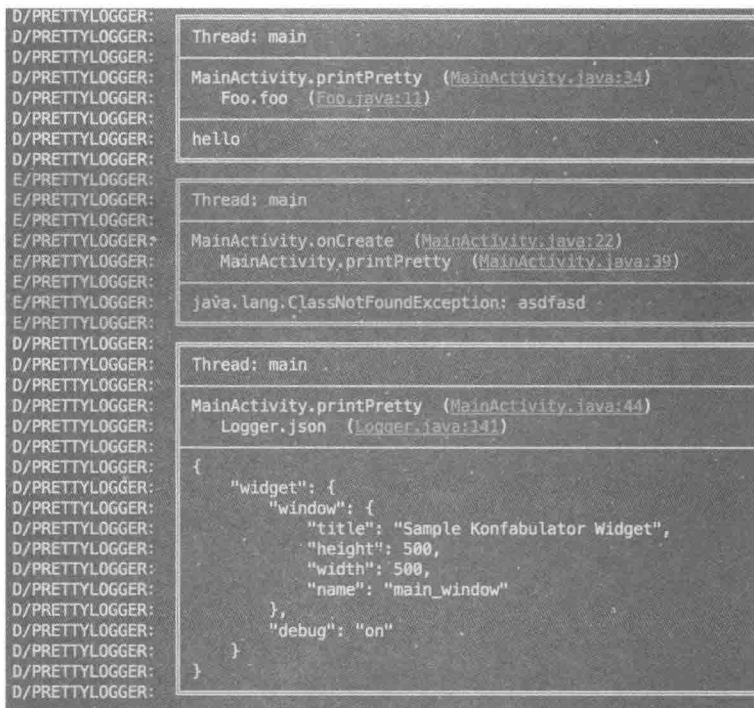


图20-2

当然 Logger 也不是完备的，它虽然支持 JSON、XML 的格式化输出，但并不支持诸如 List、Set、Map 和数组等常见 Java 集合类的格式化输出。那么如何解决呢？如果你不想自己添加，可以参考 LogUtils¹ 这个开源库，它实现了 Logger 缺失的上述特性。

再者，Logger 只支持输出日志到 Logcat，但项目开发中往往还存在将日志保存到磁盘上的需求，如何将两者结合起来呢？这时我们遇到了 timber²。

timber 是 JakeWharton 开源的一个日志记录库，它的特点是可扩展的框架，开发者可以方便快捷地集成不同类型的日志记录方式，例如，打印日志到 Logcat、打印日志到文件、打印日志到网络等，timber 通过一行代码就可以同时调用多种方式。

timber 的思想很简单，就是维护一个森林对象，它由不同类型的日志树组合而成，例如，Logcat 记录树、文件记录树、网络记录树等，森林对象提供对外的接口进行日志打印。每种类

1 <https://github.com/pengwei1024/LogUtils>

2 <https://github.com/JakeWharton/timber>

型的树都可以通过种植操作来把自己添加到森林对象中,或者通过移除操作从森林对象中删除,从而实现该类型日志记录的开启和关闭。

因此,最终我们的日志记录模块将由 timber+Logger+LogUtils 组成,当然轮子找到了,轮子的兼容合并就得靠我们自己实现了,同时我们还得增加打印到文件的日志树和打印到网络的日志树实现。

20.4 JSON 解析能力

移动互联网产品与服务器端通信的数据格式,如果没有特殊需求的话,一般都使用 JSON 格式。Android 系统也原生的提供了 JSON 解析的 API,但是它的速度非常慢,而且没有提供简洁方便的接口来提高开发者的效率和降低出错的可能。因此,通常情况下,我们都会重新选择其他优秀的 JSON 解析实现,用以代替系统的 API。经过多年的发展,目前 JSON 解析的开源实现主要包括如下几种。

20.4.1 gson¹

gson 是 Google 出品的 JSON 解析函数库,可以将 JSON 字符串反序列化为对应的 Java 对象,或者反过来将 Java 对象序列化为对应的 JSON 字符串,免去了开发者手动通过 JSONObject 和 JSONArray 将 JSON 字段逐个进行解析的烦恼,也减少了出错的可能性,增强了代码的质量。使用 gson 解析时,对应的 Java 实体类无需使用注解进行标记,支持任意复杂 Java 对象包括没有源代码的对象。

20.4.2 jackson²

jackson 是 Java 语言的一个流行的 JSON 函数库,在 Android 开发中使用时,主要包含三部分。

- jackson-core³: JSON 流处理核心库。

¹ <https://github.com/google/gson>

² <https://github.com/FasterXML/jackson>

³ <https://github.com/FasterXML/jackson-core>

- `jackson-databind`¹: 数据绑定函数库, 实现 Java 对象和 JSON 字符串流的相互转换。
- `jackson-annotations`²: `databind` 使用的注解函数库。

由于 `jackson` 是针对 Java 语言通用的 JSON 函数库, 并没有为 Android 优化定制过, 因此函数包中包含很多非必需的 API, 相比其他 JSON 函数库, 用于 Android 平台会更显著的增大最终生成的 APK 的体积。

20.4.3 Fastjson³

`Fastjson` 是阿里巴巴公司出品的一个 Java 语言编写的高性能且功能完善的 JSON 函数库。它采用一种“假定有序快速匹配”的算法, 把 JSON Parse 的性能提升到极致, 号称是目前 Java 语言中最快的 JSON 库。`Fastjson` 接口简单易用, 已经被广泛使用在缓存序列化、协议交互、Web 输出、Android 客户端等多种应用场景。

由于是 Java 语言通用的, 因此, 以前在 Android 上使用时, `Fastjson` 不可避免的引入了很多对于 Android 而言冗余的功能, 从而增加了包大小, 很多人使用的就是标准版的 `fastjson`, 但事实上, `fastjson` 还存在一个专门为 Android 定制的版本——`fastjson.android`⁴。和标准版本相比, Android 版本去掉了一些 Android 虚拟机 `dalvik` 不支持的功能, 使得 jar 更小。

20.4.4 LoganSquare⁵

`LoganSquare` 是近两年崛起的快速解析和序列化 JSON 的 Android 函数库, 其底层基于 `jackson` 的 streaming API, 使用 APT (Android Annotation Tool) 实现编译时注解, 从而提高 JSON 解析和序列化的性能。图 20-3 是官网的一张 Benchmark 图, 从中可以看到 `LoganSquare` 和 `gson`、`jackson databind` 的性能对比。

单纯从性能方面看, `LoganSquare` 应该是完胜 `gson` 和 `jackson` 的。如果和 `fastjson` 相比较, 两者应该是不相伯仲的。

上面介绍的四个函数库, 涉及的 Jar 包如图 20-4 所示。

1 <https://github.com/FasterXML/jackson-databind>

2 <https://github.com/FasterXML/jackson-annotations>

3 <https://github.com/alibaba/fastjson>

4 <https://github.com/alibaba/fastjson/wiki/Android%E7%89%88%E6%9C%AC>

5 <https://github.com/bluelinelabs/LoganSquare>



图20-3

名称	修改日期	大小	种类
fastjson-1.1.43.android.jar	今天 上午11:29	256 KB	Java archive
fastjson-1.2.7.jar	今天 下午1:14	417 KB	Java archive
gson-2.5.jar	今天 下午1:38	232 KB	Java archive
jackson-annotations-2.6.4.jar	今天 下午2:08	47 KB	Java archive
jackson-core-2.6.4.jar	今天 下午2:01	259 KB	Java archive
jackson-databind-2.6.4.jar	今天 下午2:05	1.2 MB	Java archive
logansquare-1.3.4.jar	今天 下午1:09	48 KB	Java archive

图20-4

总结起来，四个函数库的包大小分别如下。

- gson: 232KB。
- jackson: $259 + 47 + 1229 = 1.5\text{MB}$ 。
- Fastjson: 417KB。
- Fastjson.android: 256KB。
- LoganSquare: $48 + 259 = 307\text{KB}$ 。

从性能和包大小综合考虑，最终我们会选择 fastjson.android 作为基础技术堆栈中的 JSON 解析和序列化库。

20.5 数据库操作能力

无论是 iOS 平台还是 Android 平台，底层数据库都是基于开源的 SQLite 实现，然后在系统层封装成用于应用层的 API。虽然直接使用系统的数据库 API 性能很高，但是这些 API 接口并不是很方便开发者使用，一不小心就会引入 Bugs，而且代码的视觉效果也不佳。为了解决这个问题，一系列的对象关系映射（ORM）框架涌现了出来，这其中比较知名且广泛使用的有：ActiveAndroid¹、ormlite² 和 greenDAO³。

2014 年 7 月，一个跨平台的移动数据库引擎 Realm⁴ 悄然发布，它是专门为移动应用所设计的数据持久化解决方案，支持 Android 和 iOS 两大平台，并提供了 Objective-C、Swift、Android（Java）三大语言的 API。到今天，它的用户已经涵盖很多主流公司，如图 20-5 所示。



图 20-5

20.5.1 ActiveAndroid

ActiveAndroid 是一种 Active Record 风格的 ORM 框架，Active Record（活动目录）是 Yii、Rails 等框架中对 ORM 实现的典型命名方式。它可以极大的简化数据库的使用，使用面向对象的方式管理数据库，告别手写 SQL 的历史。每一个数据库表都可以被映射为一个类，开发者只需使用类似 save() 或者 delete() 这样的函数即可。

不过 ActiveAndroid 已经基本上处于维护阶段了，最新一个 Release 版本是在 2012 年发布的。

1 <https://github.com/pardom/ActiveAndroid>

2 <https://github.com/j256>

3 <https://github.com/greenrobot/greenDAO>

4 <https://realm.io/>

20.5.2 ormlite

ormlite 是 Java 平台的一个 ORM 框架，支持 JDBC 连接、Spring 和 Android 平台。在 Android 中使用时，它包含两部分。

- ormlite-core¹：核心模块，无论在哪个平台使用，都必须基于这个核心库，是实现 ORM 映射的关键模块。
- ormlite-android²：基于 ormlite-core 封装的针对 Android 平台的适配模块，Android 开发中主要跟这个模块打交道。

与 ActiveAndroid 类似，ormlite 也已经不是一个活跃的开源库，最近一次 Release 版本是在 2013 年发布的。

20.5.3 greenDAO

greenDAO 是一个轻量级且快速的 ORM 框架，专门为 Android 高度优化和定制，它能够支持每秒数千条记录的 CRUD 操作。我们从官网博客³上面一张 Benchmark 图可以看出来它与 ormlite 和 ActiveAndroid 的性能对比，如图 20-6 所示。

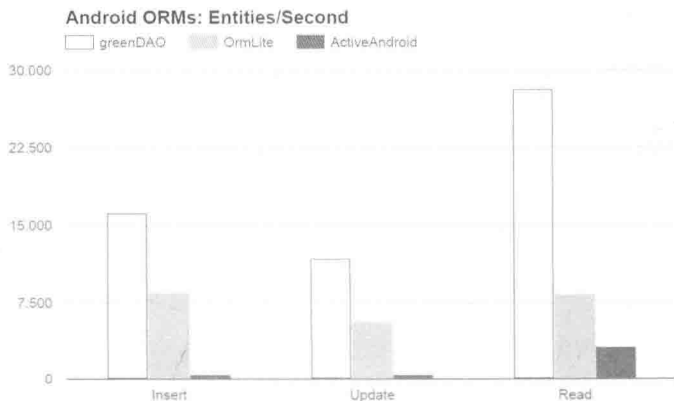


图 20-6

纵轴表示每秒执行的操作数。greenDAO 依然处于高度活跃中，两个月前刚发布了 2.1.0 版本。

1 <https://github.com/j256/ormlite-core>

2 <https://github.com/j256/ormlite-android>

3 <http://greendao-orm.com/2015/09/09/android-orm-performance-in-2015/>

20.5.4 Realm

如前面所说, Realm 是一个全新的移动数据库引擎, 它既不是基于 iOS 平台的 Core Data, 也不是基于 SQLite, 它拥有自己的数据库存储引擎, 并实现了高效快速的数据库构建操作。相比 Core Data 和 SQLite, Realm 操作要快很多, 跟 ORM 框架相比就更不用说了。

开发者应该开始尝试使用 Realm 的理由如下。

- 跨平台: Android 和 iOS 已经是事实上的两大移动互联网操作系统, 绝大多数应用都会支持这两个平台。使用 Realm, Android 和 iOS 开发者无需考虑内部数据的架构, 调用 Realm 提供的 API 即可轻松完成数据的交换, 实现“一个数据库, 两个平台之间的无缝衔接”。
- 用法简单: 相比 Core Data 和 SQLite 所需的入门知识, Realm 可以极大降低开发者的学习成本, 快速实现数据库存储功能。
- 可视化操作: Realm 为开发者提供了一个轻量级的数据库可视化操作工具, 开发者可以轻松查看数据库中的内容, 并实现简单地插入和删除等操作。

最后我们来看下如果引入上述四个函数库的话, 分别会给我们的 APP 增加多大的空间, 如图 20-7 所示。

名称	修改日期	大小	种类
activeandroid-3.0.jar	今天 下午3:32	40 KB	Java archive
greendao-2.1.0.jar	今天 下午4:00	100 KB	Java archive
ormlite-android-4.46.jar	今天 下午3:09	57 KB	Java archive
realm-android-0.87.1.jar	今天 下午4:43	4.2 MB	Java archive

图20-7

前三个 ORM 框架的大小是在正常范围之内的, 但 Realm 的大小一般项目可能无法接受, 我们将这个 Jar 包解压, 可以看到里面的内容如图 20-8 所示。

原来是不同 CPU 架构平台的 .so 文件增加了整个包的大小, 由于 arm 平台的 so 在其他平台上面是能够以兼容模式运行的, 虽然会损失性能, 但可以极大地减少函数库占用的空间。因此, 可以选择只保留 armeabi-v7a 和 x86 两个平台的 .so 文件, 直接删除无用的 .so 文件, 或者通过在工程的 build.gradle 文件中增加 ndk abi 过滤, 语句如下。

```
android {
    ...
    defaultConfig {
```



```

...
ndk {
    abiFilters "armeabi-v7a", "x86"
}
}
}

```

因此,综合考虑性能,包大小以及开源库的可持续发展等因素,我们最终选择 greenDAO。

名称	修改日期	大小	种类
.DS_Store	今天 下午5:07	6 KB	文稿
io	今天 下午5:07	--	文件夹
.DS_Store	今天 下午5:07	6 KB	文稿
realm	2015年12月23日 下午1:23	--	文件夹
lib	今天 下午5:06	--	文件夹
.DS_Store	今天 下午5:07	6 KB	文稿
arm64-v8a	2015年12月23日 下午1:28	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	1.9 MB	Unix 可执行文件
armeabi	2015年12月23日 下午1:25	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	1.2 MB	Unix 可执行文件
armeabi-v7a	2015年12月23日 下午1:27	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	1.1 MB	Unix 可执行文件
mips	2015年12月23日 下午1:30	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	2.4 MB	Unix 可执行文件
x86	2015年12月23日 下午1:32	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	1.8 MB	Unix 可执行文件
x86_64	2015年12月23日 下午1:34	--	文件夹
librealm-jni.so	2015年12月23日 下午1:39	2 MB	Unix 可执行文件
META-INF	2015年12月23日 下午1:39	--	文件夹

图20-8

20.6 网络通信能力

现在的 APP 几乎都需要从服务器获取数据,不可避免的需要具备网络通信的能力,否则会成为一个信息孤岛。

20.6.1 android-async-http¹

Android 最经典的网络异步通信函数库,做过一段时间 Android 开发的读者对这个库应该不陌生,它对 Apache 的 HttpClient² API 的封装使得开发者可以简洁优雅地实现网络请求和响应,并且同时支持同步和异步请求。android-async-http 的特性主要如下。

¹ <https://github.com/loopj/android-async-http>

² <https://hc.apache.org/httpcomponents-client-ga/>

- 支持异步 HTTP 请求，并在匿名回调函数中处理响应。
- 在子线程中发起 HTTP 请求。
- 内部采用线程池来处理并发请求。
- 通过 RequestParams 类实现 GET/POST 参数构造。
- 无须第三方库支持即可实现 Multipart 文件上传。
- 库的大小只有 60KB。
- 支持多种移动网络环境下自动智能的请求重试机制。
- HTTP 响应中实现自动的 gzip 解码，实现快速请求响应。
- 内置多种形式的响应解析，有原生的字节流、String、JSON 对象，甚至可以将 response 写到文件中。
- 可选的永久 cookie 保存，内部实现使用的是 Android 的 SharedPreferences。

笔者个人觉得在 Android 6.0 出来之前，这个函数库是 Android 平台上性价比最高的一个网络库选择，官方在 Android 2.3 开始就推荐开发者使用 HttpURLConnection 代替 Apache 的 HttpClient。但从 Android 6.0 开始，由于系统对开发者隐藏了 HttpClient 的 API，这迫使开发者如果继续使用 android-async-http 的话，就需要引入额外的 HttpClient 函数库，这显著增大了使用 android-async-http 的代价，上面提到的“库的大小只有 60KB”已然成为历史。

至于为何 Android 要移除 HttpClient，可以参见 Android 6.0 Changes[21]，如图 20-9 所示。

[m2mallow/android-6.0-changes.html](http://m2mallow.com/android-6.0-changes.html)

Apache HTTP Client Removal

Android 6.0 release removes support for the Apache HTTP client. If your app is using this client and targets Android 2.3 (API level 9) or higher, use the `HttpURLConnection` class instead. This API is more efficient because it reduces network use through transparent compression and response caching, and minimizes power consumption. To continue using the Apache HTTP APIs, you must first declare the following compile-time dependency in your `build.gradle` file:

```
android {  
    useLibrary 'org.apache.http.legacy'  
}
```

图20-9

可以看到，如果想继续使用 HttpClient，官方推荐的做法是在编译期引入 org.apache.http.legacy 这个库，这个库可以在 Android SDK 目录下的 platforms\android-23\optional 中找到，它的作用是确保在编译时不会出现找不到 HttpClient 相关 API 的错误，在应用运行时可以不依赖这个库，因为 6.0 以上的 Android 系统还没有真正移除 HttpClient 的代码，只不过 API 设置为对开发者不可见。不过通过查看 android-async-http 的源码发现，它并没有使用官网推荐的这个库，而是使用下面这个函数库来代替以前的 Apache 的 HttpClient，因此我们不能保证在 Android 的下个版本会不会把 Apache HttpClient 真正的从系统源码中彻底移除掉。

```
dependencies {
    compile 'cz.msebera.android:httpclient:4.3.6'
```

这样显著增加了 APP 的包大小，从图 20-10 的对比可以看到，Android 6.0 开始，如果想继续使用 android-async-http，那么你的 APP 需要额外增加 1.1MB 左右的大小。

名称	修改日期	大小
android-async-http-1.4.9.jar	2015年12月9日 下午7:13	106 KB
httpclient-4.3.6.jar	2015年12月9日 下午7:13	1.1 MB
org.apache.http.legacy.jar	2015年11月25日 上午9:32	303 KB

图 20-10

20.6.2 OkHttp¹

OkHttp 是一个高效的 HTTP 客户端，具有如下特性。

- 支持 HTTP/2 和 SPDY，对同一台主机的所有请求共享同一个 socket。
- 当 SPDY 不可用时，使用连接池减少请求的延迟。
- 透明的 GZIP 压缩减少下载数据大小。
- 缓存响应避免重复的网络请求。

OkHttp 在网络性能很差的情况下能够很好地工作，它能够避免常见的网络连接问题。如果你的 HTTP 服务有多个 IP 地址，OkHttp 在第一次连接失败时，会尝试其他可选的地址。这对于 IPv4+IPv6 以及托管在冗余数据中心的服务来说是必要的。OkHttp 使用现代的 TLS 特性(SNI, ALPN) 初始化 HTTP 连接，当握手失败时，会降级使用 TLS1.0 尝试初始化连接。

¹ <https://github.com/square/okhttp>

OkHttp 依赖于 okio, okio 作为 java.io 和 java.nio 的补充, 是 square 公司开发的一个函数库。okio 使得开发者可以更方便地访问、存储和处理数据。一开始是作为 OkHttp 的一个组件存在的, 当然我们也可以单独使用它。

使用 OkHttp 需要引入的 Jar 包如图 20-11 所示。

名称	修改日期	大小
okhttp-3.0.1.jar	今天 下午10:17	326 KB
okio-1.6.0.jar	2015年11月16日 上午10:38	66 KB

图20-11

20.6.3 Volley¹

Volley 是 Google 在 Google I/O 2013 上面发布的用于 Android 平台的网络通信库, 能使网络通信更快、更简单、更健壮。在 Google I/O 演讲 PPT 上, 配了一张弓箭发射图用来说明 Volley 特别适用于数据量小等通信频繁的场景, 如图 20-12。



图20-12

具体地讲, Volley 是为了简化网络任务而设计的, 用于帮助开发者处理请求、加载、缓存、多线程、同步等任务。Volley 设计了一个灵活的网络栈适配器, 在 Android 2.2 及之前的版本中, Volley 底层使用 Apache HttpClient, 在 Android 2.3 及以上版本中, 它使用 HttpURLConnection 来发起网络请求, 而且开发者也很容易将网络栈切换成使用 OkHttp。

Volley 官方源码托管在 Google Source 上面, 使用时只能直接以 Jar 包形式引入, 如果想在

¹ <https://android.googlesource.com/platform/frameworks/volley>

Gradle 中使用 compile 在线引入, 可以考虑使用 mcxiaoke 在 Github 上面的 Volley Mirror¹, 然后在 build.gradle 中使用如下语句即可。

```
compile 'com.mcxiaoke.volley:library:1.0.19'
```

20.6.4 Retrofit²

确切地说, Retrofit 并不是一个完整的网络请求函数库, 而是将 REST API 转换成 Java 接口的一个开源函数库, 它要求服务器 API 接口遵循 REST 规范。基于注解使得代码变得很简洁, Retrofit 默认情况下使用 GSON 作为 JSON 解析器, 使用 OkHttp 实现网络请求, 三者通常配合使用, 当然我们也可以将这两者替换成其他的函数库。

通过上面的分析可以发现, HttpURLConnection、Apache HttpClient 和 OkHttp 封装了底层的网络请求, 而 android-async-http、Volley 和 Retrofit 是基于前面三者的基础上二次开发而成。

最后我们来看一下这些函数库的大小, 如图 20-13 所示。

名称	修改日期	大小	种类
android-async-http-1.4.9.jar	2015年12月9日 下午7:13	106 KB	Java JAR 文件
gson-2.3.1.jar	昨天 下午10:33	211 KB	Java JAR 文件
httpclient-4.3.6.jar	2015年12月9日 下午7:13	1.1 MB	Java JAR 文件
okhttp-3.0.1.jar	昨天 下午10:17	326 KB	Java JAR 文件
okio-1.6.0.jar	2015年11月16日 上午10:38	66 KB	Java JAR 文件
org.apache.http.legacy.jar	2015年11月26日 上午8:32	303 KB	Java JAR 文件
retrofit-1.9.0.jar	昨天 下午10:33	122 KB	Java JAR 文件
volley-1.0.19.jar	昨天 下午10:29	94 KB	Java JAR 文件

图20-13

- android-async-http: 106KB+1.1MB = 1.2MB。
- OkHttp: 326KB+66KB = 392KB。
- Volley: 94KB。
- Retrofit: 122KB+211KB = 333KB。

20.7 图片缓存和显示能力

Android 发展到今天, 已经出现很多优秀的图片缓存函数库, 开发人员可以根据项目的实

¹ <https://github.com/mcxiaoke/android-volley>

² <https://github.com/square/retrofit>

实际需求进行选择。传统的图片缓存方案中设置有两级缓存，分别是内存缓存和磁盘缓存。在 Facebook 推出的 Fresco 中，它增加了一级缓存，也就是 Native 缓存，这极大地降低了使用 Fresco 的 APP 出现 OOM 的概率。

20.7.1 BitmapFun¹

BitmapFun 函数库是 Android 官方教程中的一个图片加载和缓存示例，对于简单的图片加载需求来说，使用 BitmapFun 就够了，在早期的 Android APP 开发中使用较多，后来随着越来越多成熟强大的函数库的出现，BitmapFun 也渐渐退出实际项目开发的舞台。但作为一个入门图片缓存的教程，它依然起着不可忽视的作用，毕竟万变不离其宗。

20.7.2 Picasso²

Picasso 是著名的 Square 公司众多开源项目中的一个，以著名画家毕加索为名，连 Sample app 都使用毕加索的名画作为例子。它除了实现图片的下载和二级缓存功能，还解决了一些问题。

- 在 adapter 中正常的处理 Image View 回收和下载的取消。
- 使用尽量小的内存实现复杂的图像变换。

在 Picasso 中，我们使用一行代码即可实现图片下载并渲染到 ImageView 中。

```
Picasso.with(context).load("http://i.imgur.com/DvpvklR.png").into(imageView);
```

20.7.3 Glide³

Glide 是 Google 推荐的用于 Android 平台上的图片加载和缓存函数库。这个库被广泛应用在 Google 的开源项目中，Glide 和 Picasso 有 90% 的相似度，可以说就是 Picasso 的克隆版本，只是在细节上还是存在不少区别。Glide 为包含图片的滚动列表做了尽可能流畅的优化。除了静态图片，Glide 也支持 GIF 格式图片的显示。Gilde 提供了灵活的 API 可以让开发者方便地替

¹ <http://developer.android.com/training/displaying-bitmaps/index.html>

² <https://github.com/square/picasso>

³ <https://github.com/bumptech/glide>

换下载图片所用的网络函数库，默认情况下，它使用 `URLConnection` 作为网络请求模块，开发者也可以根据自己项目的实际需求灵活使用 Google 的 `Volley` 或者 Square 的 `OkHttp` 等函数库进行替换。

`Glide` 的使用也可以使用一行代码来完成，语句如下，跟 `Picasso` 确实非常相似。

```
Glide.with(this).load("http://goo.gl/gEgYUd").into(imageView);
```

20.7.4 Fresco¹

`Fresco` 是 Facebook 开源的功能强大的图片加载和缓存函数库，相比其他图片缓存库，`Fresco` 最显著的特点是具有三级缓存：两级内存缓存和一级磁盘缓存。它的主要特性如下。

- 渐进式地的加载 JPEG 图片。
- 显示 GIF 和 WebP 动画。
- 可扩展，可自定义的图片加载和显示。
- 在 Android 4.x 和以下的系统上，将图片放在 Android 内存一个特殊的区域，从而使得应用运行更流畅，同时极大减低出现 `OutOfMemoryError` 的错误。

20.7.5 Android-Universal-Image-Loader²

`Android-Universal-Image-Loader` 简称 `UIL`，是 Android 平台老牌的图片下载和缓存函数库，功能强大灵活且高度可自定义，它提供一系列配置选项，并能很好地控制图片加载和缓存的过程。使用者甚多，早期的 Android 开发者应该都接触过，现在仍然在很多项目中使用。`UIL` 也支持二级缓存，它的主要特性如下。

- 同步或者异步的多线程图片加载。
- 高度可自定义：线程池、下载器、解码器、内存和磁盘缓存、图片显示选项等。
- 每张图片的显示支持多种自定义选项：默认存根图片、缓存切换、解码选项、`Bitmap` 处理和显示等。
- 图片可缓存在内存或者磁盘（设备的文件系统或者 SD 卡）上。

¹ <https://github.com/facebook/fresco>

² <https://github.com/nostra13/Android-Universal-Image-Loader>

- 可实时监听图片加载流程，包括下载进度。

最后来看一下如果引入这些函数库，会给 APP 增加多大的空间，如图 20-14 所示。



名称	修改日期	大小	种类
BitmapFun.jar	今天 下午9:11	71 KB	Java JAR 文件
bolts-android-1.1.4.jar	2015年9月23日 上午8:51	47 KB	Java JAR 文件
drawee-0.9.0.aar	今天 下午7:48	93 KB	文稿
fbcore-0.9.0.aar	今天 下午7:48	93 KB	文稿
fresco-0.9.0.aar	今天 下午7:48	10 KB	文稿
glide-3.7.0.jar	今天 下午7:46	475 KB	Java JAR 文件
imagepipeline-0.9.0.aar	今天 下午7:49	3 MB	文稿
imagepipeline-base-0.9.0.aar	今天 下午7:49	62 KB	文稿
imagepipeline-okhttp-0.8.1.aar	2015年11月16日 上午10:35	8 KB	文稿
nineoldandroid-2.4.0.jar	2015年9月23日 上午8:51	111 KB	Java JAR 文件
picasso-2.5.2.jar	今天 下午7:52	120 KB	Java JAR 文件
universal-image-loader-1.9.5.jar	今天 下午7:46	162 KB	Java JAR 文件

图20-14

可以看到，Fresco 函数库的依赖库有很多，除其他几个函数库的 Jar 包外，其他的都是使用 Fresco 所需要依赖的。总结起来，四个函数库的包大小分别如下。

- BitmapFun: 71KB。
- Picasso: 120KB。
- Glide: 475KB。
- Fresco: $47\text{KB} + 93\text{KB} + 93\text{KB} + 10\text{KB} + 3\text{MB} + 62\text{KB} + 8\text{KB} + 111\text{KB} = 3.4\text{MB}$ 。
- Android-Universal-Image-Loader: 162KB。

图片函数库的选择需要根据 APP 的具体情况而定，对于严重依赖图片缓存的 APP，例如壁纸类，图片社交类 APP 来说，可以选择最专业的 Fresco。对于一般的 APP，选择 Fresco 会显得比较重，毕竟 Fresco 3.4MB 的体量摆在这。根据 APP 对图片显示和缓存的需求从低到高我们可以对以上函数库做一个排序。

BitmapFun < Picasso < Android-Universal-Image-Loader < Glide < Fresco

值得一提的是，如果你的 APP 计划使用 React Native 进行部分模块功能的开发的话，那么在基础函数库选择方面需要考虑和 React Native 的依赖库的复用，这样可以减少引入 React Native 所增加的 APP 大小，可以复用的函数库有：OkHttp、Fresco、jackson-core。



第3篇 经验总结篇

- ★ 第 21 章 64K 方法数限制原理与解决方案
- ★ 第 22 章 Android 插件框架机制研究与实践
- ★ 第 23 章 推送机制实现原理详解
- ★ 第 24 章 APP 瘦身经验总结
- ★ 第 25 章 Android Crash 日志收集原理与实践

第21章

64K方法数限制原理与解决方案

Android 应用开发中，当产品迭代到一定的版本，业务模块增长到一定规模后，APK 不可避免会遇到 64K 方法数问题，了解其原理并给出解决方案是作为高级开发人员必须具备的能力。

64K 方法数问题也有人称之为 65K 方法数问题，本质上都是指 Android Dalvik 可执行文件 .dex 中的 Java 方法数引用超过 65536，64K 的计算方法是 65536 除以 1024，65K 的计算方法是 65536 除以 1000，Android 官方的叫法是 64K¹。因此，下文就统一使用 64K 这种说法，这也是一种符合计算机科学的说法。

64K 方法数问题的直观表现是在构建 APP 的时候出现编译错误，导致构建失败，在旧版本的构建系统中会提示如下错误。

```
Conversion to Dalvik format failed:  
Unable to execute dex: method ID not in [0, 0xffff]: 65536
```

新版本的构建系统会提示类似如下错误。

```
trouble writing output:  
Too many field references: 131000; max is 65536.  
You may try using --multi-dex option.
```

21.1 64K 限制的原因

Android APK 文件本质上是一个压缩文件，它里面包含的 classes.dex 文件是可执行的

¹ <https://developer.android.com/intl/es/tools/building/multidex.html>

Dalvik 字节码文件，这个 .dex 文件中存放的是所有编译后的 Java 代码。Dalvik 可执行文件规范限制了（事实上是最初设计上的一个失误）单个 .dex 文件最多能引用的方法数是 65536 个，这其中包含了 Android Framework、APP 引用的第三方函数库以及 APP 自身的方法。

21.2 使用 MultiDex 解决 64K 限制的问题

21.2.1 Android 5.0 之前的版本

我们知道在 Android 5.0 (API level 21) 之前，系统使用的是 Dalvik 虚拟机来执行 Android 应用，默认情况下，Dalvik 为每个 APK 只生成一个 classes.dex 文件，为了规避单个 .dex 文件方法数超过 64K 的问题，我们需要拆分这个单一的 classes.dex 文件，拆分后可能存在类似于 classes.dex、classes2.dex、classes3.dex 等多个 .dex 文件，具体有多少个需要看开发者的配置以及应用的方法总数，在应用启动时，会先加载 classes.dex 文件，我们称之为主 (Primary) dex 文件，应用启动后才会依次加载其他 .dex 文件，这些统称为从 (Secondary) dex 文件。为了规避 64K 限制，Google 推出了一个名为 MultiDex Support Library 的函数库，当我们下载了 Android Support Libraries 之后，可以在 <sdk>/extras/android/support/multidex/ 目录中找到这个函数库。

21.2.2 Android 5.0 及之后的版本

从 Android 5.0 (API level 21) 开始，Android 使用名为 ART 的虚拟机来代替 Dalvik 虚拟机，ART 天然支持从 APK 文件中加载多个 .dex 文件，在应用安装期间，它会执行一个预编译操作，扫描 APK 中的 classes(..N).dex 文件并将它们编译成一个单一的 .oat 文件，在应用运行时去加载这个 .oat 文件，而不是一个一个的加载 .dex 文件。

21.3 如何避免出现 64K 限制

当你的 APP 开始触碰到 64K 方法数的天花板时，我并不建议你立即使用 MultiDex Support Library 来将 APK 中单一的 .dex 拆分成多个，从而规避 64K 方法数限制引起的编译错误问题。最佳实践是永远保持应用的方法数低于 64K，永远没有机会使用 MultiDex Support Library，因为使用 MultiDex 是下下策，在大多数情况下会降低应用的性能，这个后面会说到。因此，我们

有必要先了解有哪些方法可以将应用的方法数降低到 64K 以下。

- 检查应用的直接和间接第三方依赖：在公司级项目开发中引入一个第三方函数库，是一件需要非常谨慎的事情，我们需要综合考虑这个库的体积、方法数、性能等，具体可以参见本书第20章。一个常见的反模式是引入一个很大的函数库，但仅仅用到了其中很少的功能，对于这种函数库的引入我们需要坚决抵制，有很多其他的方法可以被用来避免引入这个库，例如使用其他可替代的函数库，或者将这部分功能单独抠出来，或者自己实现这部分功能。目的都只有一个，剔除对这个函数库的依赖，从而较大的减少 APK 的方法数。
- 使用 Proguard 移除无用的代码：配置并在 Release 版本中使能 ProGuard，它的压缩功能通过分析字节码，能够检测并移除没有使用到的类、字段、方法和属性。更详细的信息可以参见本书第28章。

21.4 配置 MultiDex

如果由于各种原因，无法将应用的方法数降低到 64K 以下，那么就只能使用 Google 官方的 MultiDex 解决方案，Android 的 Gradle 插件在 Android Build Tool 21.1 开始就支持使用 MultiDex 了。首先需要配置 Application Module 的 build.gradle 文件，在其中增加对 MultiDex 函数库的依赖，同时使能 MultiDex，语句如下。

```
android {  
    compileSdkVersion 21  
    buildToolsVersion "21.1.0"  
  
    defaultConfig {  
        ...  
        minSdkVersion 14  
        targetSdkVersion 21  
        ...  
  
        // 使能 multidex 的支持  
        multiDexEnabled true  
    }  
}
```

```

    ...
}

dependencies {
    compile 'com.android.support.multidex:1.0.1'
}

```

接着引入 MultiDexApplication，需要根据项目的具体情况决定如何引入。

- 如果项目没有实现自定义的 Application 类，那么只需要在 Application Module 的 AndroidManifest.xml 文件中使用 MultiDexApplication 替换 Application 即可，语句如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.multidex.myapplication" >
    <application
        ...
        android:name="android.support.multidex.MultiDexApplication" >
        ...
    </application>
</manifest>

```

- 如果应用已经有自定义的 Application 类，那么可以让它改而继承 MultiDexApplication，语句如下。

```

public class MyApplication extends MultiDexApplication {
    @Override
    public void onCreate() {
        super.onCreate();
    }
}

```

- 如果应用已经有自定义的 Application 类，而且你不想或者不能修改它的父类，那么也可以通过覆写 attachBaseContext 方法并初始化 MultiDex，语句如下。

```

public class MyApplication extends BaseApplication {
    @Override

```

```
protected void attachBaseContext(Context base) {  
    super.attachBaseContext(base);  
    MultiDex.install(this);  
}  
}
```

为什么要在 `attachBaseContext` 方法中进行 `install` 呢？因为这个方法是在 `onCreate` 之前执行的，是开发者可以控制的应用最早执行的方法。

21.5 MultiDex Support Library 的局限性

前面我们说过，MultiDex Support Library 只是一个不得已而为之的方案，它本身并不是完美的，因此将它集成到项目中，需要经过完整的测试才能上线，可能会出现应用性能下降等问题，MultiDex Support Library 的局限性如下。

- 应用首次启动时 Dalvik 虚拟机会对所有的 `.dex` 文件执行 `dexopt` 操作，生成 ODEX 文件，这个过程很复杂且非常耗时，如果应用的 `dex` 文件太大，可能会导致出现 ANR。
- 在 Android 4.0 (API level 14) 之前的系统上，由于 Dalvik `linearAlloc` 的 bug (见 Issue 22586¹)，使用 MultiDex 的应用可能启动失败。如果你的应用还支持低于 API level 14 的系统版本时，那么在上线之前需要针对这些低版本系统经过严格充分的测试，否则用户可能会在启动你的 APP 时出现错误。理论上，使用 Proguard 的压缩功能可以减少或者消除这些潜在的问题。当然，目前低于 Android 4.0 的系统使用量已经很少了，建议直接将应用的 `minSdkVersion` 设置为 14，这样问题也就不存在了。
- 由于 Dalvik 的线性内存分配器 `linearAlloc` 的限制 (见 Issue 78035²)，使用 MultiDex 的应用在出现很大的内存分配时，可能会导致应用崩溃。根本原因是 Dalvik 虚拟机用来加载类的堆内存大小被硬编码了，Android 2.3 以下是 5M，Android 2.3 是 8M，这个内存分配的限制在 Android 4.0 (API level 14) 已经增加到了 16M，但是在 Android 5.0 (API level 21) 之前的系统上运行的 APP，还是有可能会超出这个限制，从而导致崩溃。当然 Android 5.0 开始由于使用了 ART 虚拟机，因此不再存在 `linearAlloc` 的问题。
- 当引入 MultiDex 机制时，必然会存在主 `dex` 文件和从 `dex` 文件，应用启动所需要的类都

1 <http://b.android.com/22586>

2 <http://b.android.com/78035>

必须放到主 dex 文件中，否则会出现 `NoClassDefFoundError` 的错误。Android 构建工具自动帮我们处理了 Android 系统相关的依赖，但对于应用自己引入的第三方函数库，如果还依赖其他的一些东西，例如通过反射调用 Java 类，或者调用 NDK 层代码的 Java 方法，这些可能就不会被放到主 dex 文件中，如果在应用启动时需要用到，那么必然出现问题。

21.6 在开发阶段优化 MultiDex 的构建

当初次在应用中引入并使能 MultiDex 时，对于较大型的应用，你点击开始构建按钮，然后到楼下的咖啡店买了一杯咖啡，慢悠悠地回到座位上，可能 APK 还没有生成出来。可想而知，MultiDex 在开发阶段将会极大地影响到团队的开发效率。MultiDex 之所以会增加如此显著的构建处理时间，原因在于构建系统需要经过复杂的计算决定哪些类要包含在主 dex 文件中，哪些类可以包含在从 dex 文件中。

为了解决这个问题，减少开发阶段由于引入 MultiDex 所增加的构建时间，我们可以在工程主模块的 `build.gradle` 文件中使用 `productFlavors` 来创建两个 flavor：一个是开发阶段使用的，一个是生产阶段使用的。开发阶段的 flavor 中，设置 `minSdkVersion` 为 21，这使得构建系统使用 ART 支持的格式更快的生成 MultiDex 的输出；生产阶段的 flavor 中，设置 `minSdkVersion` 为应用实际支持的最低版本号，设置代码如下。

```
android {  
    productFlavors {  
        // 定义不同的 flavor  
        dev {  
            minSdkVersion 21  
        }  
        prod {  
            // APP 实际的 minSdkVersion  
            minSdkVersion 14  
        }  
    }  
    // ...  
    buildTypes {
```

```

        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile 'com.android.support:multidex:1.0.1'
}

```

上面的配置完成后，可以使用 devDebug variant 来构建我们的应用，它会结合 dev 的 productFlavor 和 debug 的 buildType 的配置，也就是最终生成的 APK 不会使用 ProGuard，使能了 MultiDex，同时 minSdkVersion 为 21。这一系列配置使得 Android Gradle 插件做了如下工作。

- 将 Android Studio 工程中每个 module 包括它的依赖构建生成一个独立的 .dex 文件，这一步骤通常称为 pre-dexing。
- 将所有 .dex 文件都包含进 APK 中，不做任何修改。
- 最重要的一点是，每个 module 对应的 .dex 文件不会被合并，因此避免了决定哪些类要放到主 dex 文件中的耗时计算。

因此，通过这样的配置，在开发阶段，只有发生改动的 module 的 .dex 文件会重新计算生成并打包到 APK 文件中，其他 module 的 .dex 文件维持不变，从而实现了快速增量的构建。需要注意的是，通过这种方式生成的 APK 文件只能在 Android 5.0 设备上进行测试。

接着我们来介绍下如何使用 devDebug 这个 Build Variant，如果要在命令行中执行 gradle tasks，那么可以使用标准的命令加上 DevDebug 后缀，例如：

```
./gradlew installDevDebug
```

如果是使用 Android Studio，也可以很方便的进行 Build Variant 的切换，首先点击 Android Studio 左侧边栏中的 Build Variant，打开对应的窗口，然后在对应的 module 中选择 devDebug 即可，如图 21-1 所示。

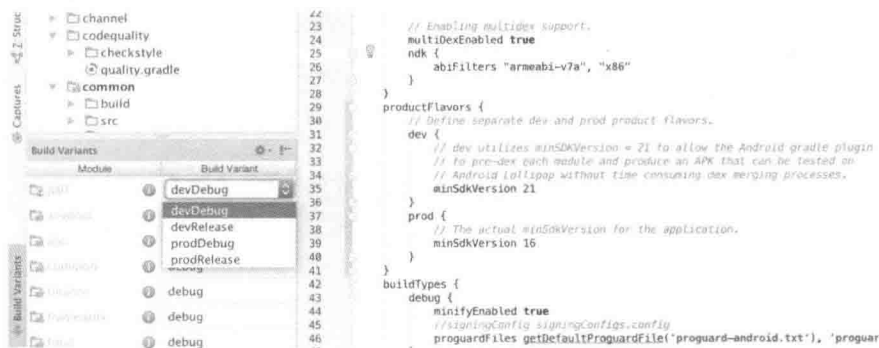


图21-1

第22章

Android 插件框架机制 研究与实践

随着移动端产品模块的不断增加，APK 的体积也在不断增长，APK 的方法数很可能会触及 64K 方法数限制问题，虽然 Google 给出了 MultiDex 的解决方案，但并不完美，其余方面暂且不提，首先是 APK 启动速度都会受到影响。

为了解决上述问题，我们引入插件框架的概念。插件框架的基本形式是将一个 APK 中的不同功能模块进行拆分，并打入到不同的 dex 文件或者 APK 文件中，主工程只是一个空壳，提供了用来加载模块 dex 或者 APK 的框架。使用插件框架的好处有很多。

- 提升 Android Studio 工程的构建速度：每个模块作为一个独立的插件进行开发和调试，Android Studio 只需要加载对应的模块和依赖库即可，不需要再像插件化之前那样加载所有的模块，构建速度得到明显提升。
- 提高应用的启动速度：引入插件化框架，应用启动时可以选择只加载必需的模块，其他非必须模块可以在使用到的再进行加载，同时规避了使用 MultiDex 方案导致的应用启动需要在主线程中执行所有 dex 文件的解压、dexopt 和加载操作，避免了应用启动黑屏和 ANR 的问题。
- 支持多团队并行开发：每个模块独立成单独的插件，不同模块的开发在定义好接口的前提下，可以做到互不干扰的并行开发。
- 在线动态加载或者更新模块：每个插件可以做到在线热更新，从而实现功能的快速上下线，不再需要依赖应用市场的发布。
- 按需加载不同的模块，实现灵活的功能配置：不同插件之间是相互独立的，可以很容易

做到根据业务需求实现插件的热插拔。

在 Android 中实现插件框架，需要解决的问题主要如下。

- 资源和代码的加载。
- Android 生命周期的管理和组件的注册。
- 插件 APK 和宿主 APK 资源引用的冲突解决。

22.1 基本概念

22.1.1 宿主和插件

宿主 APK 实现了一套插件的加载和管理的框架，它作为应用的主工程存在，插件 APK 都是依托于宿主 APK 而存在的。

插件 APK 是每个独立的功能模块，可以通过在线配置和更新实现插件 APK 在宿主 APK 中的上线和下线，以及动态更新等功能。

22.1.2 ClassLoader 机制

Android 中的 ClassLoader 机制主要用来加载 dex 文件，系统提供了两个 API 可供选择：

- PathClassLoader：只能加载已经安装到 Android 系统中的 APK 文件，因此不符合插件化的需求，不作考虑。
- DexClassLoader：支持加载外部的 APK、Jar 或者 dex 文件，正好符合插件化的需求，所有的插件化方案都是使用 DexClassLoader 来加载插件 APK 中的 .class 文件的。

22.2 开源框架

最近一两年涌现了多种插件框架的实现方案，不同的方案存在各自的优缺点，下面简单介绍下目前几个主要插件框架的基本原理和优缺点。

22.2.1 android-pluginmgr¹

android-pluginmgr 的实现原理是使用 DexMaker 的动态热部署功能来生成 Activity，让这个 Activity 继承目标插件所在的 Activity。它的主要特性如下。

- 插件 APP 不需要设置任何规则或者限制。
- 技术方法相对成熟稳定。

它的主要缺点如下。

- 基于热部署实现，因此框架的稳定性有待加强，OOM问题较突出。
- 只支持 Activity，不支持其他组件，通用性较差。

22.2.2 dynamic-load-apk²

dynamic-load-apk 是基于代理的方式实现插件框架的，需要按照一定的规则来开发插件 APK，插件中的组件需要实现经过改造后的 Activity、FragmentActivity、Service 等的子类。它的主要特性如下。

- 插件需要遵循一定的规则，因此安全方面更加可控。
- 方案简单，适用于自身少量代码的插件化改造。

它的主要缺点如下。

- 不支持通过 this 调用组件的方法，需要通过 that 去调用。
- 由于 APK 中的 Activity 没有注册，不支持隐式调用 APK 内部的 Activity。
- 插件编写和改造过程中，需要考虑兼容性问题比较多，联调起来会比较费时费力。

22.2.3 DynamicAPK³

DynamicAPK 是携程实现的一种实现多 APK/DEX 加载的插件框架解决方案，使用这个框架，我们可以实现 Android Studio 多 module 工程并行开发模式，同时可以实现在线热修复功能。

1 <https://github.com/houkx/android-pluginmgr>

2 <https://github.com/singwhatiwanna/dynamic-load-apk>

3 <https://github.com/CtripMobile/DynamicAPK>

它的主要特性如下。

- 很少的修改即可实现插件化改造：DynamicAPK 不是基于代理方式实现 Activity 等组件的生命周期管理，因此改造起来更方便；同时通过修改 aapt 实现插件中资源的管理，使得 R.java 中的资源引用和普通 Android 工程没有区别。
- 提升工程编译速度，更好地实现并行开发。
- 按需下载和更新模块的代码和资源，实现在线热更新和热修复。
- 提高 APP 启动速度：DynamicAPK 实现在 APP 启动时按需加载必需的模块，规避了 MultiDex 方案在启动时需要在主线程中执行所有 dex 的解压、dexopt、加载操作，从而提升了应用的启动速度，并避免可能引起的启动黑屏和 ANR。

它的主要缺点如下。

- 插件工程不支持 Native 代码，例如不支持 so 库。
- 插件工程不支持对 Library 工程、aar、maven 远程仓库的依赖。

22.2.4 DroidPlugin¹

DroidPlugin 是 360 手机助手实现的一种插件框架，它可以直接运行第三方的独立 APK 文件，完全不需要对 APK 进行修改或者安装。它的主要特性如下。

- 支持 Android 四大组件，而且插件中的组件不需要在宿主 APK 中注册。
- 支持 Android 2.3 及以上系统，支持所有的系统 API。
- 插件与插件之间，插件与宿主之间的代码和资源完全隔离。
- 实现了进程管理，插件的空进程会被及时回收，占用内存低。

它的主要缺点如下。

- 插件 APK 中不支持自定义资源的 Notification。
- 插件 APK 中无法注册具有特殊 IntentFilter 的四大组件。
- 缺乏对 Native 层的 Hook 操作，对于某些带有 Native 代码的插件 APK 支持不好，可能无

¹ <https://github.com/DroidPluginTeam/DroidPlugin>

法正常运行。

- 由于插件与插件，插件与宿主之间的代码完全隔离，因此，插件与插件，插件与宿主之间通信只能通过 Android 系统级别的通信方式。

22.2.5 Small¹

Small 的目标是实现最轻巧的跨平台插件化框架，它最低支持 Android API Level 8 和 iOS 7.0。Small 的主要特性如下。

- 所有插件支持内置于宿主包中。
- 插件的编码和资源文件的使用与开发普通应用没有差别。
- 通过设定 URI，宿主以及 Native应用插件，Web 插件，在线网页等能够方便地进行通信。
- 支持 Android、iOS 和 HTML5，三者可以通过同一套 Javascript 接口实现通信。

Small 目前看来唯一的不足是暂不支持 Service 的动态注册，不过这个可以通过将 Service 预先注册在宿主的 AndroidManifest.xml 文件中进行规避，因为 Service 的更新频率通常非常低。

¹ <https://github.com/wequick/Small>

第23章

推送机制实现原理详解

Android 平台的推送功能是一项基础能力，相信大部分 APP 都会直接使用第三方的推送 SDK，少部分 APP 需要自己实现推送功能，本章将会覆盖这两者，最后给出一个实现简单推送 SDK 的例子。

传统的移动端 APP 从服务端获取信息的途径是通过主动向服务端发起 Request 请求，通常称这种模式为 Pull 模式，这种模式移动端和服务端之间维持的是短连接，也就是需要时由移动端主动发起请求建立连接，获取到服务端的数据后，随即断开连接，下次需要时再重新建立；相应的，存在一种服务端主动发送消息给移动端的通信模式，通常称为 Push 模式，也就是本文中所说的推送机制。推送机制要求移动端和服务端保持一个长连接通道，当服务端需要发送消息给移动端时，直接通过这个早已建立好的连接通信即可。在实际开发中，我们需要根据具体的业务需求来决定是采用 Pull 模式还是 Push 模式。

有的 APP 基于前面所说的 Pull 模式，通过轮询的方式实现类似推送的功能，这种方式通过在移动端启动一个定时器，每隔一段时间向服务端发起 Pull 请求，存在数据则拉取，否则继续等待下一次轮询。我们称轮询的方式为伪推送，它比正常的推送需要耗费更多的电量、网络流量，而且不能实时获取数据，一般不推荐这种做法。真正的推送机制是基于 TCP 长连接实现的，并通过间隔性发送心跳包来防止 NAT¹ 超时，同时可以判断与服务端的连接是否断开，最终保证通道的畅通。

Android 平台想要实现推送功能，有一些可选的方案，Google 官方提供的 GCM（Google Cloud Messaging）服务，由于在国内相当不稳定，国内开发者一般不会选用，因此就不作介绍了。

¹ <http://baike.baidu.com/view/16102.htm>

23.1 推送的开源实现方案

23.1.1 基于 XMPP 协议¹

XMPP 协议的全称是可扩展消息与存在协议 (Extensible Messaging and Presence Protocol)，它是基于 XML 格式的协议，通常用于即时消息 (IM) 和在线现场探测。为了保证消息到达的及时性，它的底层是基于 TCP 长连接实现的，因此本质上可以看作是一个推送系统，只不过上层封装了很多即时通信所需的功能和协议，同时保证了稳定性和安全性。基于 XMPP 提供的开源框架 Openfire 和 Smack，androidpn² (Android Push Notification) 这个开源库实现了一套完整的方案，它的服务端基于 Openfire，客户端基于 Smack，如果你感兴趣的话可以去看看它的源码和用法。

XMPP 协议方式实现推送虽然简单和相对稳定，但由于它基于 XML 协议实现的消息通信，对于移动端来说很耗费网络流量，因此一般不建议作为推送的选型。

23.1.2 基于 MQTT 协议³

MQTT (Message Queuing Telemetry Transport，消息队列遥测传输) 是 IBM 公司开发的一个轻量级的消息传输协议，跟 XMPP 协议类似，也是基于发布订阅模式实现即时消息通信，不同的是，MQTT 协议针对低带宽网络、低计算能力设备等做了特殊的优化，它的设计思想是开源，可靠和简单，和 XMPP 使用冗余的 XML 协议不同，MQTT 的传输格式非常精小，最小的数据包只有两个比特。国内有些 APP 就是使用 MQTT 来实现简单的推送功能的。

23.2 推送的第三方平台

除了开源的选择，其实对于小团队而言，使用第三方平台提供的 SDK 才是正道，毕竟小公司研发实力和周期有限，那么有哪些不错的推送平台可供选择呢？常见的有友盟推送、小米推送、百度推送、极光推送等，大家可以根据自己公司的实际情况进行对比选择，本章就不展开讨论了。

¹ <https://xmpp.org/>

² <https://sourceforge.net/projects/androidpn/>

³ <http://mqtt.org/>

23.3 自己实现推送功能

接下来我们来看看自己实现一个简单的客户端推送功能，需要涉及到哪些方面。首先我们需要知道，在 Android 中想要建立 TCP 长连接，就不能使用 `HttpURLConnection` 或者 `HttpClient` 等网络请求 API，因为它们是更上层的，属于 HTTP 协议。推送功能需要使用更底层的 API，才能实现对 TCP 协议那一层的操作，Java 为开发者提供了网络套接字 `Socket`，它封装了很多 TCP 的操作。对于移动端来说，一个推送的基本框架需要包含。

- 和服务端建立连接的功能。
- 发送数据给服务端的功能。
- 从服务端接收推送数据的功能。
- 心跳包的实现。

代码实现中，上面每个功能会分别封装成独立的线程，然后通过一个管理器统筹连接的建立和管理，下面介绍的核心代码都是在线程体中实现的。

23.3.1 长连接的建立（`TCPConnectThread`）

长连接的建立主要是调用 `Socket` 类的 `connect` 方法实现的，核心代码如下。其中 `TCP_URL` 表示服务端的 URL 地址，`TCP_PORT` 表示端口号，`SOCKET_CONNECT_TIMEOUT` 表示此次连接的超时时间，这些需要根据具体需求进行设置。`setKeepAlive` 表示这次连接是长连接。

```
mSocket = new Socket();  
mSocket.connect(new InetSocketAddress(TCP_URL, TCP_PORT), SOCKET_CONNECT_TIMEOUT);  
mSocket.setKeepAlive(true);
```

通过上面的代码，一个 TCP 长连接就建立了，当然，生产代码要比上面复杂得多，例如需要判断连接是否已经建立，已经存在的话需要关闭重新建立；当手机的网络不可用时，需要增加延时重试机制，在建立 `Socket` 连接的过程中出现异常，需要重新建立连接等。

23.3.2 数据的发送（`TCPSendThread`）

长连接建立后，我们需要保存返回的 `Socket` 实例，这个实例代表这个长连接的通道，后续

移动端和服务端的数据通信都是通过这个实例进行的，既然有通信，那么需要定义好前后端通信的数据格式，Socket 通信发送的是字节数据，通常情况下，一个完整的消息至少包含协议头+协议主体内容+校验码，数据发送的核心代码如下（为简单起见，发送时协议格式是：协议主体内容长度（两个字节）+主体内容）。

```
Socket socket = PushSocket.getInstance(); // 从我们的 Socket 单例中获取上面的
Socket 实例
```

```
String content = ...; // 要发送的数据内容
```

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
short len = (short) content.length;
byte b1 = (byte) (0xff & len);
byte b2 = (byte) (0xff & (len >> 8));
out.writeByte(b1); //java short是大端，linux c short是小端
out.writeByte(b2);
out.write(mContent);
out.flush();
```

23.3.3 数据的接收（TCPReceiveThread）

推送数据的接收跟数据的发送流程类似，只不过一个是从 `DataInputStream` 中读取数据，一个是向 `DataOutputStream` 中写入数据，核心代码如下所示（为简单起见，接收时协议格式是：协议主体内容长度（两字节）+消息类型（一字节）+主体内容）。

```
Socket socket = PushSocket.getInstance(); // 获取 Socket 实例
```

```
// 获取服务端推送的数据流
```

```
DataInputStream in;
try {
    in = new DataInputStream(socket.getInputStream());
} catch (IOException e) {
    Log.e(TAG, "IOException:" + e.getMessage());
    return;
}
```

```

}

// 对数据流进行解析
try {
    // 根据协议定义，此处读取前面两字节数据
    // 然后拼接成数据主体内容的长度值
    byte b1 = in.readByte();
    byte b2 = in.readByte();

    short size = 0;
    size = (short) (size | (b2 << 8 & 0xff00));
    size = (short) (size | (b1 & 0xff));

    // 接着读取消息的类型值
    byte type = in.readByte();

    // 读取主体内容
    byte[] buf = new byte[size - 1];
    in.readFully(buf, 0, size - 1);

    // 将消息类型和内容发送给其他线程处理
    if (mHandler != null) {
        Message msg = mHandler.obtainMessage(10 + type, buf);
        msg.sendToTarget();
    }
} catch (EOFException e) {
    mIsRunning = false;
    Log.e(TAG, "EOFException:" + e.getMessage());
} catch (IOException e) {
    mIsRunning = false;
    Log.e(TAG, "IOException:" + e.getMessage());
}

```

23.3.4 心跳包的实现 (TCPHeartBeatThread)

为了保持长连接的可用性，移动端需要每隔一段时间就往 Socket 通道中发送一个心跳包，心跳包说白了就是一个遵循某个自定义协议的数据，除了可以使用心跳包实现长连接保活之外，根据业务的具体需求，我们其实也可以往心跳包中增加一个其他的信息。例如，对于打车类应用，为了获得乘客和司机的当前位置，可以考虑通过心跳包将 GPS 定位信息传递给服务端。当然，为了节省用户的手机网络流量，心跳包的大小需要严格限制。下面的核心代码，我们假定心跳包的协议格式是：心跳类型（一字节）+ GPS 定位信息（十一字节），在 Android 中心跳包的间隔性发送可以通过定时器 AlarmManager 实现，也可以通过在线程中通过 while 循环+ Thread.sleep 方式实现，核心代码如下。其中 TCP_HEARTBEAT_INTERVAL 是心跳间隔时间，可以根据具体需求而定。

```
while (mIsKeepAlive) {
    Thread.sleep(TCP_HEARTBEAT_INTERVAL); // 心跳线程睡眠

    ByteBuffer buf = ByteBuffer.allocate(1+11); // 为心跳数据分配内存空间，总共
    12个子节
    buf.put(type); // 添加心跳类型
    buf.put(LBSService.mLocationBytes); // 添加 GPS 定位信息
    buf.flip();

    // 调用前面介绍的数据发送线程实现心跳包的发送
    TCPSendThread sendTCP = new TCPSendThread(mHandler);
    sendTCP.setContent(buf.array());
    sendTCP.start();
}
```

通过这四个步骤，一个推送 SDK 就初具雏形了。当然，想要实现一个完整的推送 SDK，还有很多工作需要完成。例如推送 SDK 服务的保活，SDK 和服务端通信的安全性保证，SDK 耗费的电量和网络流量等性能指标的优化，同一个手机多个 APP 使用的推送 SDK 服务和 Socket 通道的复用等。

第24章

APP 瘦身经验总结

随着业务的不断发展，移动端产品的功能也随之逐渐增加，APP 的体积也不可避免地呈现上升的趋势，如果不加以重视，几个版本迭代下来，可能你的 APP 体积会达到用户不能忍受的程度。如果你是 SDK 开发者，你的 SDK 包大小是用户决定是否采用的关键因素；如果你的 APP 想要预装到某款手机或者某款 Android 系统中，APP 的体积也会受到很严格的限制。因此，APP 的瘦身是每个移动端产品都会遇到的一个普遍问题，本章将从不同的角度切入，全面介绍 APP 瘦身相关知识。

24.1 APP 为什么变胖了

在 Android 出现的最初几年时间里，除了游戏，我们很难看到动辄十兆或者几十兆的 APP，但现在你只要到应用市场上去走一圈，就能发现十兆以上大小的应用比比皆是，究其原因，主要有以下几种。

- 随着 Android 系统版本的碎片化发展以及手机类型的极大丰富，每个 APP 要支持的主流 dpi 分类越来越多，从最初的 ldpi、mdpi、hdpi，到后来的 xhdpi、xxhdpi、xxxhdpi、tvdpi 等。
- 随着 Android 生态系统的不断发展成熟，出现了很多方便开发者的函数库和 SDK，随着引入的函数库和 SDK 数量的增多，不可避免的引入很多重复功能代码以及资源文件。
- 用户对 APP 视觉要求的不断提高，APP 提供的资源细节越来越丰富，占用的体积也不断上升。

24.2 从 APK 文件的结构说起

到应用市场上面随便下载一个 APK 文件，由于 APK 本身是一个压缩文件，因此我们可以将后缀名由 .apk 改为 .zip，然后解压该文件，一般情况下，如果开发者在发布 APK 时没有对其进行加固等特殊处理，我们应该能够得到如图 24-1 所示的文件夹内容。

名称	修改日期	大小	种类
AndroidManifest.xml	2016年2月1日 上午11:23	68 KB	XML Document
assets	今天 下午9:24	--	文件夹
classes.dex	2016年2月1日 上午11:23	8 MB	文档
classes2.dex	2016年2月1日 上午11:23	6.5 MB	文档
classes3.dex	2016年2月1日 上午11:23	946 KB	文档
com	今天 下午9:24	--	文件夹
lib	今天 下午9:24	--	文件夹
META-INF	今天 下午9:24	--	文件夹
org	今天 下午9:24	--	文件夹
res	今天 下午9:24	--	文件夹
resources.arsc	2016年2月1日 上午11:23	1 MB	文档

图 24-1

在进一步分析如何给 APP 瘦身之前，我们首先来了解下一个 APK 文件所包含的东西，这样才能有的放矢地进行优化。

- **AndroidManifest.xml**：Android 项目的系统清单文件，用来控制 Android 应用的名称、桌面图标、访问权限等全局属性。此外，Android 应用的四大组件 Activity、Service、BroadcastReceiver 和 ContentProvider 也需要在这个文件中声明和配置。
- **assets**：该目录用来存放需要打包到 Android 应用程序的静态资源文件，例如图片资源文件、JSON 配置文件、渠道配置文件、二进制数据文件、HTML5离线资源文件等。与 res/raw 目录不同的是，assets 目录支持任意深度的子目录，同时该目录下面的文件不会生成资源ID。图24-2所示是某个 APP 的 assets 目录内容，可以一窥该目录的作用。

assets	今天 下午9:58	--	文件夹
DS_Store	今天 下午9:58	6 KB	文档
about.html	2016年1月25日 下午5:10	160 KB	HTML document
cfg	今天 下午9:57	--	文件夹
channel	2016年1月25日 下午5:10	8 字节	Unix executable
CMRequire.dat	2016年1月25日 下午5:10	2 KB	文档
config_file.txt	2016年1月25日 下午5:10	251 字节	纯文本文档
icon_bus_station.png	2016年1月25日 下午5:10	1 KB	PNG 图像
icon_end.png	2016年1月25日 下午5:10	2 KB	PNG 图像
icon_line_node.png	2016年1月25日 下午5:10	651 字节	PNG 图像
logo_h.png	2016年1月25日 下午5:10	3 KB	PNG 图像
logo_l.png	2016年1月25日 下午5:10	1 KB	PNG 图像
open_sdk_file.dat	2016年1月26日 下午5:10	726 字节	文档
place	今天 下午9:57	--	文件夹
sapi_cert.cer	2016年1月25日 下午5:10	576 字节	证书
sapi_theme	今天 下午9:57	--	文件夹
service.cfg	2016年1月25日 下午5:10	691 字节	Config.ion file
StyleSheet	今天 下午9:57	--	文件夹
VerDataSet.dat	2016年1月25日 下午5:10	172 字节	文档

图 24-2

- **classes.dex**: 应用程序的可执行文件。Android 代码都打包在这种类型的文件中, 可以通过反编译工具反编译后进行查看, 在上图中可以看到, 这个 APP 有 **classes.dex**、**classes2.dex** 和 **classes3.dex** 三个 dex 文件, 这是因为这个 APP 的方法数已经超过 65K 的限制, 需要进行分包, 对于一般的 APP 来说, 方法数目没有超过 65K, 那么打包后只会存在一个 **classes.dex** 文件。
- **lib**: 该目录存放的是应用程序依赖的不同 ABI 类型的 .so 文件, 如图24-3所示。

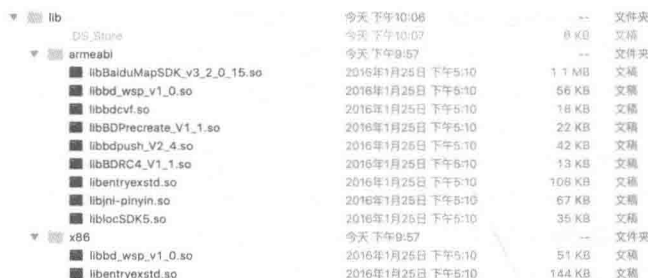


图24-3

- **res**: 该目录存放的都是应用的资源文件, 包括图片资源、字符串资源、颜色资源、尺寸资源等, 这个目录下面的资源都会出现在资源清单文件 **R.java** 的索引中。

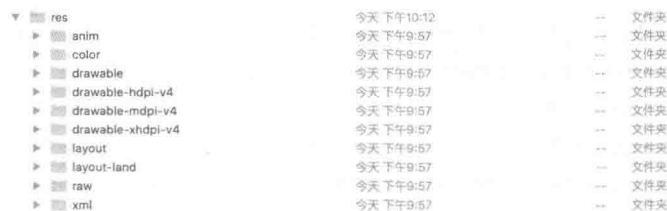


图24-4

- **resources.arsc**: 资源索引表, 用来描述具有ID值的资源的配置信息。
- **META-INF**: 该目录存放的是签名相关的信息, 用于验证 APK 包的完整性以及保证系统的安全。主要包含三个文件:
 - **MANIFEST.MF**: 主要存放 APK 包中每个文件的名字及每个文件的 SHA1 哈希值, 内容如图24-5所示。



图 24-5

□ CERT.SF: 通常每个 APP 会有一个特定的名字, 例如 BDMOBILE.SF、NETDISK_.SF 等, 它保存的是 MANIFEST.MF 的哈希值以及 MANIFEST.MF 文件中每一个哈希项的哈希值, 如图 24-6 所示。

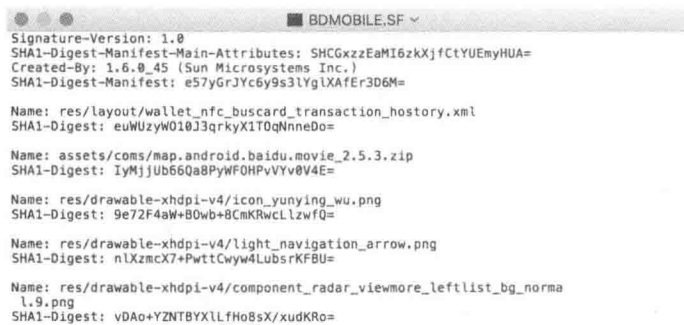


图 24-6

□ CERT.RSA: 这个文件保存了 APK 包的签名和证书的公钥信息。

从上面的分析可以看出, 一个 APK 包的组成中, 影响到最终的 APK 包大小的文件可分为三种类型, 分别如下。

- Java 代码文件: classes*.dex。
- Native 代码文件: lib 目录下面的 .so 文件。
- 资源文件: 包括 assets 目录、res 目录以及 resources.arsc 索引表文件。

下面介绍的 APP 瘦身方法都是基于如何减少以上三种类型文件的大小得出的方案。

24.3 优化图片资源占用的空间

目前移动端 Android 平台原生支持的图片格式主要有:JPEG、PNG、GIF、BMP 和 WebP (自从 Android 4.0 开始支持),但是在 Android 应用开发中能够使用的编解码格式只有其中的三种:JPEG、PNG、WebP,这可以通过查看 Bitmap 类的 CompressFormat 枚举值来确定。

```
public static enum CompressFormat {  
    JPEG,  
    PNG,  
    WEBP;  
  
    private CompressFormat() {  
  
    }  
}
```

如果要在应用层使用 GIF 格式图片,那么需要自己引入第三方函数库进行支持。在对应用中的图片资源进行压缩和优化之前,我们有必要对这几种常见的图片格式做一个简单的了解和区分。

- JPEG: JPEG (发音为/jay-peg/)是一种广泛使用的有损压缩图像的标准格式,它不支持透明和多帧动画,一般摄影类作品最终都是以 JPEG 格式展示。通过控制压缩比,可以调整图片的大小。
- PNG: PNG 是一种无损压缩图片格式,它支持完整的透明通道,从图像处理领域讲, JPEG 只有 RGB 三个通道,而 PNG 有 ARGB 四个通道。由于是无损压缩,因此 PNG 图片一般占用空间比较大,会无形中增加最终 APP 的大小,我们在做 APP 瘦身时一般都要对 PNG 图片进行处理以降低其大小。
- GIF: GIF 是一种古老的图片格式,它诞生于 1987 年,随着初代互联网流行开来。它的特点是支持多帧动画。大家对这种格式肯定不陌生,社交平台上面发送的各种动态表情,大部分都是基于 GIF 来实现的。
- WebP: 相比前面几种图片格式, WebP (发音为 /weppy/) 算是一个初生儿了,它由 Google 在 2010 年发布,它支持有损和无损压缩、支持完整的透明通道、也支持多帧动画,是一种比较理想的图片格式。目前国内很多主流 APP 都已经应用了 WebP,例如微信、微博、淘宝等。在既保证图片质量又要限制图片大小的需求下, WebP 应该是首选。

目前无论 Android 平台还是 iOS 平台，大多数 APP 在搭建界面时使用的几乎都是 PNG 格式图片资源，除非你的项目已经全面支持 WebP 格式，否则你都会面临对 PNG 图片瘦身的要求。在这里，我们可以通过几个工具对 PNG 图片进行压缩来达到瘦身的目的。

24.3.1 无损压缩 [ImageOptim]¹

ImageOptim 是一个无损的压缩工具，它通过优化 PNG 压缩参数，移除冗余元数据以及非必需的颜色配置文件等方式，在不牺牲图片质量的前提下，既减小了 PNG 图片占用的空间，又提高了加载的速度。

24.3.2 有损压缩 [ImageAlpha]²

ImageAlpha 是 ImageOptim 作者的一个有损的 PNG 压缩工具，相比较而言，图片大小得到极大的减低，当然同时图片质量也会受到一定程度的影响，经过该工具压缩的图片，需要经过设计师的检视才能最终上线，否则可能会影响到整个 APP 的视觉效果。

24.3.3 有损压缩 [TinyPNG]³

TinyPNG 也是比较知名的有损 PNG 压缩工具，它以 Web 站点的形式提供，没有独立的 APP 安装包，同所有的有损压缩工具一样，经过压缩的图片，需要经过设计师的检视才能最终上线，否则可能会影响到整个 APP 的视觉效果。

当然还有很多无损压缩工具，例如 [JPEGMini]⁴、[MozJPEG]⁵ 等，大家可以从中选择适合自己项目的一个就行，主要是在图片大小和图片质量之间找到一个折衷点。

24.3.4 PNG/JPEG 转换为 WebP

如果你的 APP 最低支持到 Android 4.0，那么可以直接使用系统提供的能力来支持 WebP，

1 <https://imageoptim.com/>

2 <https://pngmini.com/>

3 <https://tinypng.com/>

4 <http://www.jpegmini.com/>

5 <https://imageoptim.com/mozjpeg/>

如果是 4.0 以下的系统,也可以通过在 APP 中集成第三方函数库,例如 [webp-android-backport]¹ 来实现对 WebP 的支持。

根据 Google 的测试,无损压缩后的 WebP 比 PNG 文件少了 45% 的文件大小,即使这些 PNG 文件经过其他压缩工具例如 ImageOptim 压缩之后,WebP 依然可以减少约 28% 的文件大小。

需要注意的是,对于具有 Alpha 通道的 PNG 图片来说,如果需要在 Android 4.2.1 之前的系统上运行,那么不能转换成 WebP 格式,因为只有在 Android 4.2.1 以上的系统中,才能解析具有 Alpha 通道的 WebP 图片。

WebP 转换的工具可以选择 [智图]² 和 [iSparta]³ 等。

24.3.5 尽量使用 NinePatch 格式的 PNG 图

9.png 图片格式简称 NinePatch 图,本质上仍然是 PNG 格式图片,它是针对 Android 平台的一种特殊 PNG 图片格式,可以在图片指定位置拉伸或者填充内容。NinePatch 图的优点是体积小、拉伸不变形、能够很好的适配 Android 各种机型。Android SDK 自带了 NinePatch 图的编辑工具,位于 sdk/tools/draw9patch 中,点击即可启动。当然,Android Studio 也集成了 PNG 转 NinePatch 的功能,我们只需右键点击某个需要转换的 PNG 图片,在弹出的对话框中选择 Create 9-Patch File... 即可自动完成转换。

最后我们以 TinyPng 为例来直观地观察压缩工具对图片的压缩效果,如图 24-7 所示。



图 24-7

¹ <https://github.com/alexey-pelykh/webp-android-backport>

² <http://zhitu.isux.us/>

³ <http://isparta.github.io/>

24.4 使用 Lint 删除无用资源

Proguard 只会对 Java 代码起作用, 对于 `res/drawable*` 目录中的图片, 如果没有使用到, Proguard 只会移除该图片在 R 类中的引用, 不会删除该图片。这时就需要用到 Android Lint 了。Android Lint 天然集成在 Android Studio 中, 它会分析 `res` 目录下面的资源文件, 但不会分析 `assets` 目录下面的资源文件。具体使用方法可以参考本书第 49 章的相关介绍。对工程执行 Android Lint, 结果出来后, 查看 Android Lint — Unused resources 选项即可得到哪些资源是多余的, 如图 24-8 所示。当然, 我们不能过度依赖工具, 还需要人工确认是否真的是多余的, 例如某些资源是通过 Java 反射机制来使用的, 这时 Android Lint 还是会检查出该资源没有被使用到。

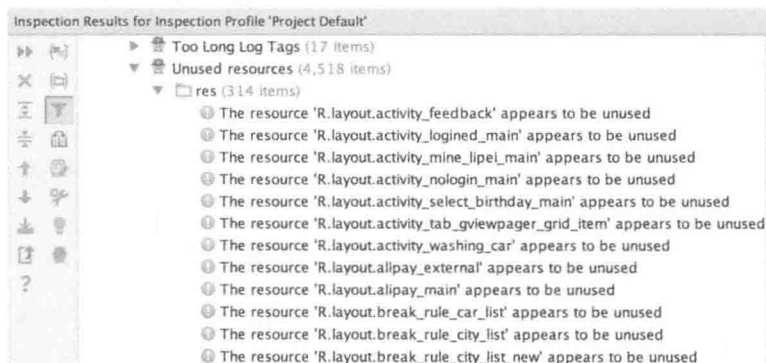


图 24-8

24.5 利用 Android Gradle 配置

在 Android Studio 工程的 `app/build.gradle` 文件中进行一些配置可以进一步缩减最终生成的 APK 大小, 它们分别如下。

24.5.1 minifyEnable

标识是否开启 Proguard 混淆, 设置为 `true` 时需要同时设置 Proguard 配置文件名和规则。Proguard 的作用不仅仅是混淆, 它还具有压缩、优化等功能。它会遍历所有代码并找出没有引用到的代码, 这些无用代码在生成最终的 APK 文件之前会被剔除掉。同时 Proguard 会使用简短的字母组合替换原来的类名、属性名等, 这些都能在一定程度上减少 APP 的大小。

24.5.2 shrinkResources

标识是否去除无用的 resource 文件。需要注意的是, shrinkResources 依赖于 minifyEnable, 需要 minifyEnable 设置为 true 时才会生效。同时, shrinkResources 需要慎重使用, 某些资源是通过反射获取的, 这类资源可能也会被删除掉, 从而在运行时会报 Resources\$NotFoundException 异常。

以上两种方式的代码示例如下。

```
android {  
    // 省略其他  
    buildTypes {  
        release {  
            minifyEnabled true  
            shrinkResources true  
            proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),  
                'proguard-rules.pro'  
        }  
    }  
}
```

24.5.3 resConfigs

Android 开发过程中不可避免的会引入第三方开源函数库或者 SDK, 在不修改它们的前提下, 在最终生成的 APK 中, 我们可能会引入很多其实不需要使用到的资源文件, 主要可以分为两种。

- DPI 目录: Android 从出现到现在, 历经了多个版本, 支持多种不同类型的设备, 屏幕密度、屏幕形状、屏幕大小等都差别很大, 支持的屏幕密度就有 ldpi、mdpi、hdpi、xhdpi、xxhdpi、xxxhdpi 等多种类型。在实际项目开发中, 我们当然不可能为每一种屏幕密度提供对应的一套资源文件, 这不仅没必要而且会显著增加 APP 的体积, 我们需要调研产品的目标用户以及目前市场上主流的手机设备屏幕密度, 满足这些用户和设备即可。因此需要剔除引用的第三方函数库或者 SDK 中可能存在的不需要的 DPI 目录及其文件。

- 国际化文件：Android 开发中不可避免地会引用外部的函数库，这些函数库是为了全世界开发人员服务的，不可避免地会存在很多国际化文件，而对于普通的 APP 来说，可能只需要支持本国语言就可以了。因此，也需要剔除无用的国际化文件。

在 Android Studio 中，我们可以通过在 Gradle 中进行配置，来选择最终打包到 APK 中的资源，这一步是通过指定 `resConfig` 或者 `resConfigs` 的取值，通过如下 DSL（Domain Specific Language）防止 AAPT（Android Asset Packaging Tool）打包不需要的资源。

```
android {
    // 省略其他
    defaultConfig {
        // 省略其他
        resConfigs "en", "fr"
        resConfigs "nodpi", "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"
    }
}
```

24.5.4 ndk.abiFilters

在工程的 `build.gradle` 文件中增加 `ndk.abiFilters` 配置，可以指定我们需要的 ABI 类型，从而可以过滤掉不需要的 ABI 类型的 `.so` 文件。

```
android {
    // 省略其他
    defaultConfig {
        // 省略其他
        ndk {
            abiFilters "armeabi-v7a", "x86"
        }
    }
}
```

24.6 重构和优化代码

保持良好的编码习惯，对你的代码进行持续的优化或者重构，减少重复代码，实现代码复

用,这方面可以多读一读 Martin Fowler 的《重构 – 改善既有代码的设计》¹一书。在项目开发中,建议抽出一个基础库,提供基础的功能,例如网络、数据库、加解密、utils 工具包等,实现不同模块间复用基础的功能,甚至在公司层面维护一个公共库,在不同产品线之间共享。这样如果不同产品之间需要相互集成,复用一套公共库,能在很大程度上减少重复的代码。

24.7 资源混淆

资源文件的混淆方案目前有 [美团]² 和 [微信]³ 两种,前者是通过修改 AAPT 在处理资源文件相关的源码达到资源文件名的替换;后者是通过直接修改 resources.arsc 文件达到资源文件名的混淆。相比之下,微信的方案更优,而且微信已经将这个方案开源出来,地址在 [AndResGuard]⁴,感兴趣的同学通过上面的 README.md 介绍可以很容易集成到自己的项目中。

24.8 插件化

插件化开发也是减少 APP 体积的一个可行的途径,不过首先你需要实现一个插件化的框架,用来在线动态的下载并加载各个插件。

¹ http://www.amazon.cn/%E9%87%8D%E6%9E%84-%E6%94%B9%E5%96%84%E6%97%A2%E6%9C%89%E4%BB%A3%E7%A0%81%E7%9A%84%E8%AE%BE%E8%AE%A1-%E9%A9%AC%E4%B8%81%C2%B7%E7%A6%8F%E5%8B%92/dp/B011LPUB42/ref=sr_1_1?s=books&ie=UTF8&qid=1456545179&sr=1-1&keywords=%E9%87%8D%E6%9E%84%3A%E6%94%B9%E5%96%84%E6%97%A2%E6%9C%89%E4%BB%A3%E7%A0%81%E7%9A%84%E8%AE%BE%E8%AE%A1

² <http://tech.meituan.com/mt-android-resource-obfuscation.html>

³ http://mp.weixin.qq.com/s?__biz=MzAwNDY1ODY2OQ==&mid=208135658&idx=1&sn=ac9bd6b4927e9e82f9fa14e396183a8f#rd

⁴ <https://github.com/shwenzhang/AndResGuard>

第25章

Android Crash 日志收集 原理与实践

Android 应用在开发和测试过程中，如果出现 Crash，我们一般可以通过查看 Logcat 日志信息来获取崩溃的原因，如果是必现的 Crash，可以保留手机的现场信息，很方便地进行调试和分析。但是当应用发布到应用市场后，在用户的手机上出现 Crash，我们显然不可能直接去找用户要日志信息以及手机来进行定位，那么这时我们如何才能得到 Crash 相关的信息呢？这就需要通过 Crash 监控 SDK 来实现。

Android 应用的 Crash 日志收集与上报是每个 APP 都必须具备的基本能力，Crash 对应用的用户留存率、口碑等有着极大的影响，试想如果你使用的 APP 经常性地发生崩溃，你还会继续使用它吗？特别是现在市场上同类产品竞争如此激烈的情况下。因此，一个 APP Crash 率的高低是衡量程序员能力的一个指标，很多的公司更是将它作为程序员的 KPI 指标之一。

大多数的开发者其实都不会面临需要自己研发 Crash SDK 的任务，原因很简单，市场上已经有很多第三方的 Crash SDK 可供选择，例如腾讯的 Bugly、百度的 MTJ、友盟 SDK 等。如果没有特殊需求，我们的主要目标还是应该聚焦在 APP 业务功能的实现与优化上面。话虽如此，我们还是有必要了解 Crash 日志收集与上报的基本原理，这不仅能增加你的知识面，同时如果某一天突然需要自己实现的话，不至于手忙脚乱。

那么实现一个 Crash SDK 包括哪些方面的工作呢？总结起来可以分为三方面。

- Crash 的捕获。
- Crash 堆栈信息的获取。
- Crash 日志的上报。

Android 底层是基于 Linux 操作系统构建的，上层是基于 Java 语言实现的，上层与底层的通信基于 JNI。因此，在 Android 上面做开发，不可避免的需要跟 Java 和 C/C++ 打交道。相应的，Crash 可能发生在这两层，Java 是基于虚拟机运行的，而 C/C++ 更贴近操作系统，这两层的 Crash 机制差别比较大，本文将分两部分分别进行介绍。

25.1 Java 层 Crash 捕获机制

25.1.1 基本原理

Android 应用程序是基于 Java 语言开发的，异常处理也是沿用 Java 语言的机制。在 Java 中异常分为两类：Checked Exception 和 UnCheckedException。CheckedException 又称为编译时异常，它是在编译阶段必须处理的，否则编译失败，一般使用 try...catch 捕获并进行处理即可。UnCheckedException 又称为运行时异常，这种类型的异常是在编译阶段检测不出来的，只有在程序运行时满足某些条件才会触发。当然，也可以使用 try...catch 来捕获这类异常。但事实是，我们往往不知道在什么地方应该添加对这类异常的捕获。

好在 Java API 提供了一个全局异常捕获处理器，Android 应用在 Java 层捕获 Crash 依赖的就是 Thread.UncaughtExceptionHandler 处理器接口，通常情况下，我们只需要实现这个接口，并重写其中的 uncaughtException 方法，在该方法中可以读取 Crash 的堆栈信息，语句如下。

```
public class MyUncaughtExceptionHandler implements Thread.  
UncaughtExceptionHandler {  
  
    @Override  
    public void uncaughtException(Thread thread, Throwable ex) {  
        final Writer result = new StringWriter();  
        final PrintWriter printWriter = new PrintWriter(result);  
  
        // 如果异常是在 AsyncTask 里面的后台线程抛出的  
        // 那么实际的异常仍然可以通过 getCause 获得  
        Throwable cause = ex;  
        while (null != cause) {
```

```

        cause.printStackTrace(printWriter);
        cause = cause.getCause();
    }

    // stacktraceAsString 就是获取的 crash 堆栈信息
    final String stacktraceAsString = result.toString();

    printWriter.close();
}
}

```

为了使用自定义的 `UncaughtExceptionHandler`，我们还需要对它进行注册，以替换应用默认的异常处理器，一般都是在 `Application` 类的 `onCreate` 方法中进行注册，语句如下。

```

public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();

        Thread.setDefaultUncaughtExceptionHandler(new
        MyUncaughtExceptionHandler());
    }
}

```

通常情况下，收集发生 `Crash` 的堆栈信息就已经足够我们分析并定位出崩溃的原因，从而修复这个 `Crash`。但复杂一点的 `Crash`，可能仅有堆栈信息是不够的，我们还需要其它一些信息来辅助问题的定位和解决，这些信息包括如下内容。

25.1.2 线程信息

线程的基本信息包括 `ID`、名字、优先级和所在的线程组，可以根据实际情况收集某些线程的信息，但通常收集发生 `Crash` 的线程信息即可，通用的线程信息收集代码如下。

```

public class ThreadCollector {

```

```

@NonNull
public static String collect(@Nullable Thread thread) {
    StringBuilder result = new StringBuilder();
    if (thread != null) {
        result.append("id=").append(thread.getId()).append("\n");
        result.append("name=").append(thread.getName()).append("\n");
        result.append("priority=").append(thread.getPriority()).
append("\n");
        if (t.getThreadGroup() != null) {
            result.append("groupName=").append(thread.
getThreadGroup().getName()).append("\n");
        }
    }
    return result.toString();
}
}

```

25.1.3 SharedPreferences 信息

某些类型的 Crash 依赖于应用的 SharedPreferences 中的某些信息项。例如某个开关，当打开时，会导致 APP 运行发生 Crash，关闭时不存在问题，这时为了准确复现这个 Crash，如果有收集 SharedPreferences 中的信息，将会极大的加速问题的定位，通用的收集代码如下。

```

final class SharedPreferencesCollector {

    private final Context mContext;

    private String[] mSharedPrefIds;

    public SharedPreferencesCollector(Context context, String[]
sharedPrefIds) {
        mContext = context;
        mSharedPrefIds = sharedPrefIds;
    }
}

```

```

@NonNull
public String collect() {
    final StringBuilder result = new StringBuilder();

    // 收集默认的 SharedPreferences 信息
    final Map<String, SharedPreferences> sharedPrefs = new
    TreeMap<String, SharedPreferences>();
    sharedPrefs.put( "default" , PreferenceManager.getDefaultSharedPrefe
    rences(mContext));

    // 收集应用自定义的 SharedPreferences 信息
    if (mSharedPrefIds != null) {
        for (final String sharedPrefId : mSharedPrefIds) {
            sharedPrefs.put(sharedPrefId, mContext.getSharedPreferences(
            sharedPrefId, Context.MODE_PRIVATE));
        }
    }

    // 遍历所有的 SharedPreferences 文件
    for (Map.Entry<String, SharedPreferences> entry : sharedPrefs.
    entrySet()) {
        final String sharedPrefId = entry.getKey();
        final SharedPreferences prefs = entry.getValue();

        final Map<String, ?> prefEntries = prefs.getAll();

        // 如果 SharedPreferences 文件内容为空
        if (prefEntries.isEmpty()) {
            result.append(sharedPrefId).append( '=' ).
            append( "empty\n" );
            continue;
        }
    }
}

```

```

// 遍历添加某个 SharedPreferences 文件中的内容
for (final Map.Entry<String, ?> prefEntry : prefEntries.
entrySet()) {
    final Object prefValue = prefEntry.getValue();
    result.append(sharedPrefId).append( '.' ).append(prefEntry.
getKey()).append( '=' );
    result.append(prefValue == null ? "null" : prefValue.
toString());
    result.append( "\n" );
}
result.append( '\n' );
}

return result.toString();
}
}

```

25.1.4 系统设置

在 Android 中，许多的系统属性都是在系统设置中进行设置的，比如蓝牙、Wi-Fi 的状态、当前的首选语言、屏幕亮度等。这些信息存放在数据库中，对应的 URI 为 `content://settings/system`、`content://settings/secure`、`content://settings/global` 等。对这些数据库的读写操作对应着 Android SDK 中的 `Settings` 类，我们对系统设置的读写本质上就是对这些数据库表的操作。

- **System:** 以键值对的形式存放系统中各种类型的常规偏好设置，它是可读写的，获取这种类型设置的代码如下，使用反射的方式是为了兼容不同的 API Level。

```

final class SettingsCollector {

    private static final String LOG_TAG = "SettingsCollector";

    private final Context mContext;

```

```
public SettingsCollector(Context context) {
    mContext = context;
}

@NonNull
public String collectSystemSettings() {
    final StringBuilder result = new StringBuilder();
    final Field[] keys = Settings.System.class.getFields();
    for (final Field key : keys) {
        // Avoid retrieving deprecated fields... it is useless, has an
        // impact on prefs, and the system writes many warnings in the
        // logcat.
        if (!key.isAnnotationPresent(Deprecated.class) && key.getType()
            == String.class) {
            try {
                final Object value = Settings.System.getString(mContext.
                    getContentResolver(), (String) key.get(null));
                if (value != null) {
                    result.append(key.getName()).append( "=" ).
                        append(value).append( "\n" );
                }
            } catch (@NonNull IllegalArgumentException e) {
                Log.w(LOG_TAG, "Error : ", e);
            } catch (@NonNull IllegalAccessException e) {
                Log.w(LOG_TAG, "Error : ", e);
            }
        }
    }

    return result.toString();
}
}
```

- Secure: 以键值对的形式存放系统的安全设置, 这个是只读的, 获取这种类型设置的代码如下。

```

@NonNull
public String collectSecureSettings() {
    final StringBuilder result = new StringBuilder();
    final Field[] keys = Settings.Secure.class.getFields();
    for (final Field key : keys) {
        if (!key.isAnnotationPresent(Deprecated.class) && key.getType()
            == String.class && isAuthorized(key)) {
            try {
                final Object value = Settings.Secure.getString(mContext.
                    getContentResolver(), (String) key.get(null));
                if (value != null) {
                    result.append(key.getName()).append( "=" ).
                        append(value).append( "\n" );
                }
            } catch (@NonNull IllegalArgumentException e) {
                Log.w(LOG_TAG, "Error : ", e);
            } catch (@NonNull IllegalAccessException e) {
                Log.w(LOG_TAG, "Error : ", e);
            }
        }
    }

    return result.toString();
}

```

- Global: 以键值对的形式存放系统中对所有用户公用的偏好设置, 它是只读的, 获取这种类型设置的代码如下。

```

@NonNull
public String collectGlobalSettings() {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR1) {
        return "";
    }
}

```

```

    }

    final StringBuilder result = new StringBuilder();
    try {
        final Class<?> globalClass = Class.forName( "android.provider.
Settings$Global" );
        final Field[] keys = globalClass.getFields();
        final Method getString = globalClass.getMethod( "getString",
ContentResolver.class, String.class);
        for (final Field key : keys) {
            if (!key.isAnnotationPresent(Deprecated.class) && key.
getType() == String.class && isAuthorized(key)) {
                final Object value = getString.invoke(null, mContext.
getContentResolver(), key.get(null));
                if (value != null) {
                    result.append(key.getName()).append( "=" ).
append(value).append( "\n" );
                }
            }
        }
    } catch (@NonNull IllegalArgumentException e) {
        Log.w(LOG_TAG, "Error : ", e);
    } catch (@NonNull InvocationTargetException e) {
        Log.w(LOG_TAG, "Error : ", e);
    } catch (@NonNull NoSuchMethodException e) {
        Log.w(LOG_TAG, "Error : ", e);
    } catch (@NonNull SecurityException e) {
        Log.w(LOG_TAG, "Error : ", e);
    } catch (@NonNull ClassNotFoundException e) {
        Log.w(LOG_TAG, "Error : ", e);
    } catch (@NonNull IllegalAccessException e) {
        Log.w(LOG_TAG, "Error : ", e);
    }
}

```



```

        return result.toString();
    }

    private boolean isAuthorized(@Nullable Field key) {
        if (key == null || key.getName().startsWith("WIFI_AP")) {
            return false;
        }
        return true;
    }
}

```

21.1.5 Logcat 中的日志记录

捕获 Logcat 日志的好处是可以清楚地知道 Crash 发生前后的上下文，对于准确定位 Crash 来说提供了更完备的信息，实现代码如下。

```

class LogCatCollector {

    private static final String LOG_TAG = "LogCatCollector";

    private static final int DEFAULT_TAIL_COUNT = 100;
    // 保留 logcat 输出中最后的行数

    private static final int DEFAULT_BUFFER_SIZE_IN_BYTES = 8192;

    public String collectLogCat(@Nullable String bufferName, boolean
logcatFilterByPid, String[] logcatArguments) {
        final int myPid = android.os.Process.myPid();
        String myPidStr = null;
        if (logcatFilterByPid && myPid > 0) { // 只收集当前进程相关的logcat信息
            myPidStr = Integer.toString(myPid) + " ):";
        }

        final List<String> commandLine = new ArrayList<String>();
    }
}

```

```

        commandLine.add( "logcat" );
        if (bufferName != null) {
            commandLine.add( "-b" );
            commandLine.add(bufferName);
        }

        // logcat 的 "-t n" 参数是 API Level 8 才引入的, 对于之前的系统版本
        // 需要做特殊处理来模拟这种情况
        final int tailCount;
        final List<String> logcatArgumentsList = new ArrayList<String>(Arrays.
asList(logcatArguments));

        final int tailIndex = logcatArgumentsList.indexOf( "-t" );
        if (tailIndex > -1 && tailIndex < logcatArgumentsList.size()) {
            tailCount = Integer.parseInt(logcatArgumentsList.get(tailIndex +
1));

            if (Build.VERSION.SDK_INT < Build.VERSION_CODES.FROYO) {
                logcatArgumentsList.remove(tailIndex + 1);
                logcatArgumentsList.remove(tailIndex);
                logcatArgumentsList.add( "-d" );
            }
        } else {
            tailCount = -1;
        }

        final LinkedList<String> logcatBuf = new BoundedLinkedList<String>(ta
ilCount > 0 ? tailCount
            : DEFAULT_TAIL_COUNT);
        commandLine.addAll(logcatArgumentsList);

        BufferedReader bufferedReader = null;

        try {
            final Process process = Runtime.getRuntime().exec(commandLine.

```

```

toArray(new String[commandLine.size()]));

        bufferedReader = new BufferedReader(new
InputStreamReader(process.getInputStream()), DEFAULT_BUFFER_SIZE_IN_BYTES);

        // Dump stderr to null
        new Thread(new Runnable() {
            public void run() {
                try {
                    InputStream stderr = process.getErrorStream();
                    byte[] dummy = new byte[DEFAULT_BUFFER_SIZE_IN_
BYTES];

                    //noinspection StatementWithEmptyBody
                    while (stderr.read(dummy) >= 0)
                        ;

                    } catch (IOException ignored) {
                    }

                }
            }).start();

        while (true) {
            final String line = bufferedReader.readLine();
            if (line == null) {
                break;
            }
            if (myPidStr == null || line.contains(myPidStr)) {
                logcatBuf.add(line + "\n");
            }
        }

    } catch (IOException e) {
        Log.e(LOG_TAG, "LogCatCollector.collectLogCat could not
retrieve data.", e);
    } finally {
        try {

```

```

        if (null != bufferedReader) {
            bufferedReader.close();
        }
    } catch (IOException e) {
        // 至此已经无能为力
    }
}

return logcatBuf.toString();
}
}

```

25.1.6 自定义 Log 文件中的内容

有时候，我们的 APP 会将一些重要的日志信息有选择的存放到内部存储或者外部存储的某个 Log 文件中，当发生 Crash 时，也可以收集这个 Log 文件中的内容并上传到服务器，帮助问题的分析和定位，实现代码如下。可以收集指定文件中指定行数的内容。

```

class LogFileCollector {

    @NonNull
    public String collectLogFile(@NonNull Context context, @NonNull String
fileName, int numberOfLines) throws IOException {
        final BoundedLinkedList<String> resultBuffer = new BoundedLinkedList<
String>(numberOfLines);
        final BufferedReader reader = getReader(context, fileName);
        try {
            String line = reader.readLine();
            while (line != null) {
                resultBuffer.add(line + "\n");
                line = reader.readLine();
            }
        } finally {
            try {

```

```

        reader.close();
    } catch (IOException e) {

    }
}

return resultBuffer.toString();
}

@NonNull
private static BufferedReader getReader(@NonNull Context context, @
NonNull String fileName) {
    try {
        final FileInputStream inputStream;
        if (fileName.startsWith("/") ) {
            // 绝对路径
            inputStream = new FileInputStream(fileName);
        } else if (fileName.contains("/") ) {
            // 相对路径
            inputStream = new FileInputStream(new File(context.
getFilesDir(), fileName));
        } else {
            // 应用内部存储中的某个文件
            inputStream = context.openFileInput(fileName);
        }
        return new BufferedReader(new InputStreamReader(inputStream),
1024);
    } catch (FileNotFoundException e) {
        return new BufferedReader(new InputStreamReader(new
ByteArrayInputStream(new byte[0])));
    }
}
}

```

25.1.7 MemInfo 信息

Crash 发生时的内存使用情况对某些类型的 Crash 定位也是有很大帮助的，通过执行 `dumpsys meminfo` 命令可以获取到当前进程的内存使用信息，语句如下。

```
final class DumpSysCollector {

    private static final String LOG_TAG = "DumpSysCollector";

    private static final int DEFAULT_BUFFER_SIZE_IN_BYTES = 8192;

    @NonNull
    public static String collectMemInfo() {

        final StringBuilder meminfo = new StringBuilder();
        BufferedReader bufferedReader = null;
        try {
            final List<String> commandLine = new ArrayList<String>();
            commandLine.add( "dumpsys" );
            commandLine.add( "meminfo" );
            commandLine.add(Integer.toString(android.os.Process.myPid()));

            final Process process = Runtime.getRuntime().exec(commandLine.
toArray(new String[commandLine.size()]));
            bufferedReader = new BufferedReader(new
InputStreamReader(process.getInputStream()), DEFAULT_BUFFER_SIZE_IN_BYTES);

            while (true) {
                final String line = bufferedReader.readLine();
                if (line == null) {
                    break;
                }
                meminfo.append(line);
                meminfo.append( "\n" );
            }
        } catch (IOException e) {
            return null;
        } finally {
            if (bufferedReader != null) {
                bufferedReader.close();
            }
        }
        return meminfo.toString();
    }
}
```

```

    }

    } catch (IOException e) {
        Log.e(LOG_TAG, "DumpSysCollector.meminfo could not retrieve
data", e);
    }

    try {
        if (null != bufferedReader) {
            bufferedReader.close();
        }
    } catch (IOException e) {

    }

    return meminfo.toString();
}
}

```

25.2 Native 层 Crash 捕获机制

Native 层代码的错误可以分为两种。

C++ 异常：在 Native 层，如果使用 C++ 语言进行开发，而且使用了 C++ 异常机制，那么函数执行可以抛出 `std::exception` 类型的异常；如果使用 C/C++ 语言开发，使用的是错误码机制，那么对于一些导致系统不可用的错误码，我们也可以进行捕获上报。总的来说，C++ 异常通常是可捕获的，一般不会引起 APP Crash，当然如果处理不当，会引起逻辑错误。

Fatal Signal 异常：在 Native 层，由于 C/C++ 野指针或者内存读取越界等原因，导致 APP 整个 Crash 的错误。这种 Crash 一般会在 Logcat 中打印出包含 Fatal signal 字样的日志。对于这种 Crash，前面介绍的 Java 异常捕获类 `Thread.UncaughtExceptionHandler` 是检测不到的。那么如

何捕获这种异常并上报呢？

熟悉 Linux 底层的同学应该很容易看出这种 Crash 是基于 Linux 的信号处理机制。信号（又称为软中断信号，signal）本质上是一种软件层面的中断机制，用来通知进程发生了异步事件。进程之间可以互相通过系统调用 kill 来发送软中断信号；Linux 内核也可以因为内部事件而给进程发送信号，通知进程某个事件的发生。需要注意的是，信号并不携带任何数据，它只是用来通知某进程发生了什么事情。接收到信号后，通常有三种处理方式。

- 自定义处理信号：进程为需要处理的信号提供信号处理函数。
- 忽略信号：进程忽略不感兴趣的信号（SIGKILL 和 SIGSTOP 忽略不了）。
- 使用系统的默认处理：使用内核的默认信号处理函数，默认情况下，系统对大部分信号的缺省操作是终止进程。

了解信号的基本知识后，那么问题就变得很简单了，由于 Native 层 Crash 大部分都是 signal 软中断类型错误，因此只要捕获 signal 并进行处理，得到中断的具体信息就可以很好帮助定位了。这一步可以通过调用 sigaction 注册信号处理函数来完成。

// 要捕获的信号类型

```
const int handledSignals[] = {
    SIGFPE, SIGSEGV, SIGABRT, SIGFPE, SIGILL, SIGBUS, SIGIPE, SIGSTKFLT
};
```

// 信号类型的个数

```
const int handledSignalsNum = sizeof(handledSignals) /
sizeof(handledSignals[0]);
```

// 保存老的信号处理器

```
struct sigaction old_handlers[handledSignalsNum];
```

```
void initCrashHandler()
```

```
{
    struct sigaction handler;
    memset(&handler, 0, sizeof(sigaction));
    handler.sa_sigaction = my_handler;
```



```

handler.sa_flags = SA_RESETHAND;

// 注册信号处理函数的宏定义, 减少冗余代码
#define CATCH_SIG(X) sigaction(handledSignals[X], &handler, &old_handlers[X])

// 遍历所有关注的信号并注册信号处理器
for (int i = 0; i < handledSignalsNum; ++i)
{
    CATCH_SIG(handledSignals[i]);
}
}

```

上面代码中的 `my_handler` 回调函数就是用来处理信号的, 在这个函数中, 我们设法获取 Native Crash 的相关堆栈信息, 然后上报给服务器。但是 Native 层并没有提供像 Java 层那样的 `Throwable.printStackTrace` 函数来获取堆栈信息, 目前来说有两种思路。

- 抓取 Logcat 日志: 前面说过, Native 层发生 fatal signal 导致 APP 崩溃, 也会在 Logcat 中打印出相关的堆栈信息, 因此, 当在 Native 层检测到 fatal signal, 利用我们的信号处理函数 `my_handler` 可以向 Java 层发送消息, 通知它去抓取 Logcat 的日志, 抓取的方式上面我们已经介绍过, 需要注意的一点是, 这时候由于 Crash 应用原有的进程将会很快被结束掉, 因此 Logcat 的抓取应该开启新的进程, 例如启动一个新进程在 Service 中进行操作。
- Google Breakpad: 这是一个跨平台的崩溃转储和分析工具, 支持 Windows、Linux、OS X、Android 等, 通过集成它提供的函数库, 在应用发生崩溃时会将相关堆栈信息写入一个 minidump 格式文件中, 通过将这个文件上传到服务器, 开发人员可以通过 `addr2line` 等工具将 dump 文件中的函数地址转换成对应的代码行数, 从而知道问题发生的具体位置。

25.3 Crash 的上报

Crash 的上报相比捕获和堆栈信息的获取来说, 就很简单了, 基本工作就是和服务端定好通信协议的格式, 通常情况下, Crash 上报的信息除了包含上面说到的堆栈等信息之外, 还需

要包含以下信息。

- 应用的版本号。
- 系统类型及版本号。
- 手机设备型号。
- 手机唯一的设备ID。
- 渠道号。
- Crash 发生的时间。
- 应用的包名。



第4篇 新技术篇

- ★ 第 26 章 函数式编程思想及其在 Android 中的应用
- ★ 第 27 章 依赖注入及其在 Android 中的应用
- ★ 第 28 章 Android 世界的 Swift : Kotlin 在 Android 中的应用
- ★ 第 29 章 React Native For Android 入门指南
- ★ 第 30 章 Android 在线热修复方案研究
- ★ 第 31 章 面向切面编程及其在 Android 中的应用
- ★ 第 32 章 基于 Facebook Buck 改造 Android 构建系统

第26章

函数式编程思想及其在Android中的应用

函数式编程是一种编程范式，它的基础是 lambda 演算。lambda 演算的函数可以接受函数作为输入和输出。和指令式编程相比，函数式编程中强调函数的运算比指令的执行重要，函数是第一等公民。

在面向对象编程中，数据和对数据的操作是耦合在一起的，对象对外隐藏它的操作细节，其他对象通过封装好的接口调用这些操作，核心的抽象模型是数据本身，核心的活动是组合新对象以及拓展存在的对象，而这些是通过添加新的方法实现的。

在函数式编程中，数据与函数是松耦合的，函数隐藏了自身的实现，核心抽象模型是函数，而不是数据结构，核心活动是编写新的函数。函数响应式编程为解决现代编程问题提供了全新的视角。一旦理解它，可以极大地简化项目，特别是处理嵌套回调的异步事件、复杂的列表过滤和变换或时间相关问题。

目前在 Android 开发中使用函数式编程有两个框架可以使用，比较成熟稳定且使用广泛的是 RxJava¹ & RxAndroid²，另外一个 Google 最近推出的 Agera³ 框架，后者有望成为官方标准。

- RxJava：一个在 Java 虚拟机上实现的函数响应式扩展库，它扩展自观察者模式，提供了基于 observable 序列实现的异步调用及基于事件的响应式编程。
- RxAndroid：RxJava 在 Android 平台的扩展，能够简化 Android 上面基于 RxJava 的开发。

1 <https://github.com/ReactiveX/RxJava>

2 <https://github.com/ReactiveX/RxAndroid>

3 <https://github.com/google/agera>

- Agera: Google 推出的帮助 Android 开发者更好地实现函数式、异步和响应式开发范式的框架，可以在 Android SDK 9 及以上版本使用。类似于 RxJava，Agera 也是基于观察者模式实现的。

接下来以 RxJava 为例介绍函数式编程的简单用法。响应式代码的基本组成部分是 Observables 和 Subscribers，Observable 用来发送消息，而 Subscriber 用来接收消息。消息的发送是有固定模式的，Observable 可以发送任意数量的消息，当消息被成功处理或者出错时，流程结束。Observable 会调用它的每个 Subscriber 的 onNext() 方法，并最终以 Subscriber.onComplete() 或者 Subscriber.onError() 结束。

可以看到，RxJava 的基本概念很像标准的观察者模式，一个显著的不同点在于 Observables 一般只有等到 Subscriber 订阅它，才会开始发送消息，如果没有订阅者观察它，那么将不会起任何作用。接下来看看创建一个基本的 Observable 的例子。

```
Observable<String> myObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> sub) {
            sub.onNext("asce1885");
            sub.onCompleted();
        }
    }
);
```

这个 Observable 很简单，只是简单的发送一个字符串 asce1885 然后完成，现在让我们创建一个 Subscriber 来消费这个数据。

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) { Log.d(s); }

    @Override
    public void onCompleted() { }

    @Override
    public void onError(Throwable e) { }
```

```
};
```

Subscriber 只是简单地将收到的消息打印出来，现在既然有了观察者和被观察者，最后一步就是将两者通过 subscribe 方法关联起来：

```
myObservable.subscribe(mySubscriber); // 输出 asce1885
```

当订阅完成后，myObservable 将调用 mySubscriber 的 onNext() 和 onComplete() 方法，然后打印出 asce1885 然后结束运行。

26.1 代码的简化

上面我们完成了 asce1885 字符串的打印需要编写的样板代码有点多，但事实上 RxJava 提供了很多便捷的方式来简化代码的编写。首先让我们来简化 Observable，RxJava 为常见的操作提供了很多内建的 Observable 创建函数，例如，可以使用 Observable.just() 方法来简化上面的 Observable 代码。

```
Observable<String> myObservable =  
    Observable.just("asce1885");
```

接着简化 Subscriber 代码，如果我们不关心 onComplete() 或者 onError() 的处理的话，那么可以使用一个更简单的类来替换 onNext() 方法中需要完成的功能，语句如下。

```
Action1<String> onNextAction = new Action1<String>() {  
    @Override  
    public void call(String s) {  
        Log.d(s);  
    }  
};
```

Actions 可以用来定义 Subscriber 的每一个部分，Observable.subscribe() 方法能够接收一个、两个或者三个 Action 参数，分别表示 onNext()、onError() 和 onComplete() 方法，上面的 Subscriber 可以简化为如下语句。

```
myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);
```

当然，上面的例子我们不需要处理 onError() 和 onComplete()，因此，只需要第一个参数。

```
myObservable.subscribe(onNextAction);
```

最后，我们把上面的函数调用链接起来从而去掉临时变量，结果如下。

```
Observable.just("asce1885")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            Log.d(s);
        }
    });
```

如果使用 Java 8 的 lambda 表达式还可以进一步将代码简化成如下语句。

```
Observable.just("asce1885")
    .subscribe(s -> Log.d(s));
```

26.2 Operators 简介

接下来我们介绍一个名为 Operators 的功能，它在消息发送者 Observable 和消息消费者 Subscriber 之间起到操纵消息的作用。RxJava 默认拥有大量的 Operators，本节我们来介绍 map() operator，它可以用来将已被发送的消息转换成另外一种形式。

```
Observable.just("asce1885")
    .map(new Func1<String, String>() {
        @Override
        public String call(String s) {
            return s + " @HUST";
        }
    })
    .subscribe(s -> Log.d(s));
```

同样可以使用 Java 8 的 lambda 表达式进一步简化代码。

```
Observable.just("asce1885")
    .map(s -> s + " @HUST")
    .subscribe(s -> Log.d(s));
```

`map()` operator 本质上是一个用于转换消息对象的 `Observable`，我们可以级连调用任意多个 `map()` 方法，一层一层的将初始消息转换成 `Subscriber` 需要的数据形式。

上面的例子虽然比较简单，但我们可以得到两个重要的概念。

- `Observable` 和 `Subscriber` 能完成任何你想得到的事情：`Observable` 可以是一个数据库查询，`Subscriber` 获取查询结果并显示在页面上；`Observable` 也可以是从网络上获取数据，`Subscriber` 将其保存到本地磁盘中；`Observable` 还可以是屏幕上面某个控件的点击，而 `Subscriber` 用来监听并响应这个点击事件。
- `Observable` 和 `Subscriber` 与它们之间的一系列转换步骤是相互独立的：我们可以在消息发送者 `Observable` 和消息消费者 `Subscriber` 之间加入任意多个类似 `map()` 的 operator 方法，只要 operators 符合输入输出的数据模型，那么我们可以得到一个无穷尽的调用链。

第27章

依赖注入及其在 Android 中的应用

依赖注入在 JavaEE 开发中是很常见的一种开发范式，近年来 Android 开发中也涌现了很多实用的依赖注入框架，框架的使用有助于开发人员少写很多样板代码，同时能够达到不同类之间的解耦，方便单元测试。

依赖注入让开发者可以编写低耦合的代码，它解耦了依赖者和被依赖者，使得在修改被依赖者的具体实现时，不需要修改依赖者；同时对于依赖者的测试，可以很容易地使用 mock 对象替换被依赖者，可以独立地对依赖者进行单元测试。Android 平台上比较具代表性的依赖注入函数库主要有 ButterKnife、RoboGuice、Dagger 和 Dagger2，本章将一一进行介绍，不过在这之前，为了更好地理解引入框架的意义，我们先来了解依赖注入的基本概念。

27.1 基本概念

在解释依赖注入的概念之前，我们首先来说一说控制反转（Inversion of Control，缩写为 IoC）的涵义，IoC 是 Martin Fowler 在 2004 年总结提炼出来的，它现在是面向对象编程的重要法则之一，旨在减小程序中不同部分的耦合问题，IoC 一般分为两种类型。

- 依赖注入（Dependency Injection，简称 DI）。
- 依赖查找（Dependency Lookup）。

其中，依赖注入使用广泛，因此，一般控制反转和依赖注入可互换使用，下文统一使用依赖注入一词。

在代码编写过程中，经常会遇到需要以组合方式组织不同的类，例如汽车类 Car 是由引擎类 Engine，车轮类 Wheel 等其它的类组合而成，实现可能如下。

```
public class Car {  
    private Engine mEngine;  
    private Wheel mWheel;  
  
    public Car() {  
        mEngine = new PetrolEngine();  
        mWheel = new MichelinWheel();  
    }  
}
```

上面的代码可以正常运行，也没有逻辑上的错误，但缺点是 Car 和 Engine 以及 Wheel 的耦合比较严重。

- 在 Car 的构造函数中，它需要自己创建 Engine（Wheel 也一样）的实例。
- 同时需要知道如何实现 Engine，在上面的代码中，Car 需要知道 PetrolEngine 这个 Engine 子类的存在。
- 一旦需要将 Engine 类型由 Petrol 改为 Diesel，需要修改 Car 的构造函数。

那么如何通过依赖注入的思想来解耦上述代码呢？对应到依赖注入的三种常见的实现方式，我们来改造上面的代码。

27.1.1 构造函数注入

这是最简单最直观的注入方式，就是将 Engine（Wheel）类的实例化从 Car 类的构造函数中抽离出来，Engine（Wheel）的实例在其他地方创建，然后通过构造函数的参数的方式传递给 Car 类就够了，语句如下。

```
public class Car {  
    private Engine mEngine;  
    private Wheel mWheel;  
  
    public Car(Engine engine, Wheel wheel) {
```

```
mEngine = engine;
mWheel = wheel;
}
}
```

27.1.2 Setter 函数注入

Setter 函数注入是在 Car 类构造完成之后，通过设置的方式传入 Engine（Wheel）的实例，同样可以达到解耦的目的。

```
public class Car {
    private Engine mEngine;
    private Wheel mWheel;

    public Car() {

    }

    public void setEngine(Engine engine) {
        mEngine = engine;
    }

    public void setWheel(Wheel wheel) {
        mWheel = wheel;
    }
}
```

27.1.3 接口注入

接口注入相对 Setter 函数注入更为抽象，本质上是通过实现接口中的方法来实现 Setter 的功能。首先，需要创建用于提供注入方法的接口，语句如下。

```
public interface ICarInject {
    void bindEngine(Engine engine);
    void bindWheel(Wheel wheel);
}
```

接着 Car 类实现这个接口。

```
public class Car implements ICarInject {
    private Engine mEngine;
    private Wheel mWheel;

    public Car() {

    }

    public void bindEngine(Engine engine) {
        mEngine = engine;
    }

    public void bindWheel(Wheel wheel) {
        mWheel = wheel;
    }
}
```

27.2 为何需要框架

对于简单的需求，我们可以像上面那样使用工厂方法或者生成器来手动创建类，并传递依赖。但是这并不是长久之计，当依赖的层级变多变复杂，手动传递依赖将是一件繁琐的事情，会写出很多样板代码。这时就需要引入依赖注入框架来帮开发者完成繁琐的依赖管理问题，通过使用框架，我们可以轻松的配置依赖。

27.3 开源框架的选择

27.3.1 ButterKnife¹

ButterKnife 严格意义上不能算是一个依赖注入框架，因为它专注于实现 Android View 的属性和方法的绑定，并不支持其他方面的注入。ButterKnife 是由 JakeWharton 独立开发的，最

¹ <https://github.com/JakeWharton/butterknife>

新的稳定版本 7.0.1 依然是通过运行时反射实现 View 的注入，性能较低下，不过最新的开发版本 8.0.0-SNAPSHOT 使用编译时注解来提升性能，关于编译时注解的知识可以参见本书第 11 章。从中可以看出，优秀的开源库都在一直进化并使用最新的技术提高自身的竞争优势。ButterKnife 使用注解帮开发者生成样板代码，主要提供以下功能。

- 使用 @Bind 注解来代替 findViewById 函数，这是 ButterKnife 最典型的功能。

```
class ExampleActivity extends Activity {
    @Bind(R.id.title) TextView title;
    @Bind(R.id.subtitle) TextView subtitle;
    @Bind(R.id.footer) TextView footer;

    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        ButterKnife.bind(this);
        // TODO Use fields...
    }
}
```

- 将多个 View 组成一个列表或者数组，可以实现一次性对所有 View 的操作。

```
@Bind({ R.id.first_name, R.id.middle_name, R.id.last_name })
List<EditText> nameViews;

// 使用 apply 方法可以实现对所有 View 的统一 Action 操作
ButterKnife.apply(nameViews, DISABLE);
static final ButterKnife.Action<View> DISABLE = new ButterKnife.
Action<View>() {
    @Override public void apply(View view, int index) {
        view.setEnabled(false);
    }
};

// 使用 apply 方法可以实现对所有 View 的统一 Setter 操作
ButterKnife.apply(nameViews, ENABLED, false);
```

```
static final ButterKnife.Setter<View, Boolean> ENABLED = new ButterKnife.
Setter<View, Boolean>() {
    @Override public void set(View view, Boolean value, int index) {
        view.setEnabled(value);
    }
};
```

// 使用 apply 方法设置 View 的属性

```
ButterKnife.apply(nameViews, View.ALPHA, 0.0f);
```

- 通过 @OnClick 注解可以消除代码里面设置监听器所需的匿名内部类定义。

```
@OnClick(R.id.submit)
public void submit(View view) {
    // TODO submit data to server...
}
```

- 在属性上使用资源注解可以代替资源查找方法的使用。

```
@BindString(R.string.login_error)
String loginErrorMessage;
```

27.3.2 RoboGuice¹

RoboGuice 是 Google Guice 在 Android 平台上的移植和定制版本，是 Android 平台比较老牌的依赖注入框架之一，已经在成百上千的应用中使用。RoboGuice 3.0 开始引入 RoboBlender，用于实现部分功能的编译时注解处理，提升性能。RoboGuice 的使用比较具有代码侵入性，它提供了一系列改写后的 Android 组件类，开发者为了使用依赖注入功能，必须修改项目中对应的组件继承这些类。

RoboGuice 提供的主要功能有：

- View 的注入：类似 ButterKnife 提供的功能，不过在 RoboGuice 中实现 View 的注入，需要首先让 Activity 继承 RoboActivity，接着使用 @ContentView 注解设置 ContentView，当然也可以直接使用 setContentView 函数直接设置，最后使用 @InjectView 注解来标记需要

¹ <https://github.com/roboguice/roboguice>

注入的 View，使用 RoboGuice 前的代码如下。

```
public class MyActivity extends Activity {
    TextView textView;

    @Override
    protected void onCreate( Bundle savedInstanceState ) {
        setContentView(R.layout.myactivity_layout);
        textView = (TextView) findViewById(R.id.text1);
        textView.setText( "Hello!" );
    }
}
```

使用 RoboGuice 改造后的代码变得精简了很多。

```
@ContentView(R.layout.myactivity_layout)
public class MyActivity extends RoboActivity {
    @InjectView(R.id.text1) TextView textView;

    @Override
    protected void onCreate( Bundle savedInstanceState ) {
        textView.setText( "Hello!" );
    }
}
```

- 资源的注入：资源的注入同样需要继承 Robo 的子类，例如 RoboActivity，然后使用 @InjectResource 进行注解，语句如下。

```
public class MyActivity extends RoboActivity {
    // 其中, my_animation 是位于 res/anim 目录中的动画文件
    @InjectResource(R.anim.my_animation) Animation myAnimation;
}
```

- 系统服务的注入：系统服务的注入跟 View 的注入类似，而且更简单，只需要在 Robo 的子类中使用 @Inject 注解即可轻松实现，省去了开发者自己调用 getSystemService 的麻烦，语句如下。

```
class MyActivity extends RoboActivity {
    @Inject Vibrator vibrator;
    @Inject NotificationManager notificationManager;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // 可以直接使用注解后的系统服务了
        vibrator.vibrate(1000L);
        notificationManager.cancelAll();
    }
}
```

- POJO 的注入：POJO（Plain Old Java Objects）的注入也是使用 @Inject 注解，默认情况下，RoboGuice 会调用 POJO 类的无参数构造函数来实例化这个类，语句如下。

```
class MyActivity extends RoboActivity {
    @Inject Foo foo; // 相当于调用 new Foo();
}
```

当 POJO 类的构造函数带参数时，一种简单的办法是使用 @Inject 显式注解 POJO 类的构造函数，语句如下。

```
class Foo {
    Bar bar;

    @Inject
    public Foo(Bar bar) {
        this.bar = bar;
    }
}
```

这样，前面所说的 @Inject Foo foo 将会使用 Foo(Bar bar) 构造函数进行实例化，其中参数 Bar 的注入也是类似的流程。

类似于 Activity，BroadcastReceiver、ContentProvider、Service 等都可以使用 RoboGuice 实现依赖注入，本章就不赘述了。

27.3.3 Dagger¹

Dagger 是 Square 公司开源的专为低端设备设计的依赖注入框架。大部分依赖注入框架通过反射来创建和注入依赖，反射机制虽然很有用，但在 Android 平台上面耗时严重，特别是在老的 android 版本上面。因此，Dagger 使用编译时注解功能在编译阶段创建工作所需的所有类。这样一来，就不需要用到反射了。Dagger 相比其他依赖注入器功能稍弱，但效率却是非常高的。在使用 Dagger 之前，我们需要在工程的 build.gradle 文件中添加如下依赖。

```
dependencies {
    compile 'com.squareup.dagger:dagger:1.2.2'
    provided 'com.squareup.dagger:dagger-compiler:1.2.2'
}
```

第一个是 Dagger 函数库，第二个是 Dagger 预编译器函数库，它会创建注入依赖所需的类。通过创建预编译的类可以避免大部分的反射操作。由于我们只需要 Dagger 编译器函数库来编译工程，在应用中不会使用到它，因此我们把它标记为 provided，这样生成的 APK 中就不会包含这个函数库了。

基本用法

Dagger 使用 javax.inject.Inject 注解来标记需要注入的构造函数和属性，使用 @Inject 注解标记构造函数，在需要创建这个类的实例时，Dagger 会获取构造函数所需的参数并调用这个构造函数进行实例化。

```
class Thermosiphon implements Pump {
    private final Heater heater;

    @Inject
    Thermosiphon(Heater heater) {
        this.heater = heater;
    }

    ...
}
```

¹ <http://square.github.io/dagger/>

Dagger 同样使用 `@Inject` 注解对属性进行注入，如果这个类没有同时对构造函数进行注入，那么 Dagger 默认会使用这个类的无参数构造函数进行实例化。

```
class CoffeeMaker {
    @Inject Heater heater;
    @Inject Pump pump;

    ...
}
```

需要注意的是，`@Inject` 不支持对普通方法的注入，同时它也不支持接口 `Interface`、第三方函数库中的类，在这些场景中，我们可以转而使用 `@Provides` 注解，它用来标识函数，表示这个函数作为注入的提供者，同时函数名并不重要，重要的是函数的返回值类型。如下所示，当我们需要一个 `Heater` 的实例时，`provideHeater` 函数将会被调用。

```
@Provides Heater provideHeater() {
    return new ElectricHeater();
}
```

所有被标记为 `@Provides` 的函数都应该放在使用注解 `@Module` 标记的类定义中，语句如下。

```
@Module
class DripCoffeeModule {
    @Provides Heater provideHeater() {
        return new ElectricHeater();
    }

    @Provides Pump providePump(Thermosiphon pump) {
        return pump;
    }
}
```

值得一提的是，默认的命名规范是：`@Provides` 注解的函数名以 `provide` 作为前缀，`@Module` 注解的类名以 `Module` 作为后缀。

构建对象图

对象图是所有依赖存在的地方，它包含被创建的实例，并能够把这些实例注入到相应的对

象中。在前面的例子中，我们看到了经典的依赖注入，注入都是通过构造函数参数传递实现的。但在 Android 开发中，有的类例如 Application、Activity 等，开发者对它们的构造函数并没有控制权，因此需要使用另外的方式来实现依赖注入。语句如下。

```
class CoffeeApp implements Runnable {
    @Inject CoffeeMaker coffeeMaker; // 这个是依赖

    @Override public void run() {
        coffeeMaker.brew();
    }

    public static void main(String[] args) {
        ObjectGraph objectGraph = ObjectGraph.create(new DripCoffeeModule());
        CoffeeApp coffeeApp = objectGraph.get(CoffeeApp.class);
        ...
    }
}
```

我们使用 ObjectGraph.create() 来获取对象图实例，它可以接受一个或者多个 @Module 类实例。同时使用 @Inject 注解来指明谁是依赖，需要注意的是，这些依赖项必须是 public 或者 default 访问范围的，以便 Dagger 可以给它们赋值。最后有一个问题，ObjectGraph 是如何知道 CoffeeApp 的呢？这需要修改 DripCoffeeModule 的定义，增加 CoffeeApp 的注入，语句如下。

```
@Module(
    injects = CoffeeApp.class
)
class DripCoffeeModule {
    ...
}
```

单例模式

在前面定义 @Provides 函数时，如果增加 @Singleton 注解，那么这个函数会一直返回相同的对象实例，语句如下。

```
@Provides @Singleton Heater provideHeater() {
    return new ElectricHeater();
}
```

至此，我们已经了解 Dagger 的基本用法，更详细的内容可以参见官方文档，接下来我们将会学习 Dagger2，它在 Dagger 的基础上更进一步优化。

27.3.4 Dagger2¹

Dagger2 是 Google 基于 Square 的 Dagger 基础上的二次开发，它移除了 Dagger 中所有反射的使用，同时在构建对象图时使用 @Component 注解代替 Dagger 中的 ObjectGraph/Injector 的使用，从而让开发者可以写出更简洁的代码。Dagger2 的编译时注解使用 android-apt²，因此需要在 build.gradle 文件中如下引入依赖。

```
apply plugin: 'com.neenbedankt.android-apt'

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'
    }
}

dependencies {
    apt 'com.google.dagger:dagger-compiler:2.0'
    compile 'com.google.dagger:dagger:2.0'
    ...
}
```

由于继承自 Dagger，因此，Dagger2 的很多用法和 Dagger 一模一样，例如 @Inject、@Provides、@Module 和 @Singleton 等。接下来只介绍不一样的地方。

构建对象图

在 Dagger2 中我们使用 @Component 注解来实现对象图，首先定义一个接口 interface，这个接口提供一个无参数的函数，返回值是所需要的类类型，通过使用 @Component 注解标记这个

¹ <http://google.github.io/dagger/>

² <https://bitbucket.org/hvisser/android-apt>

interface, 并传入 modules 参数, 语句如下。

```
@Component(modules = DripCoffeeModule.class)
interface CoffeeShop {
    CoffeeMaker maker();
}
```

其中, DripCoffeeModule 是 Dagger 一节中介绍的类, 它使用 @Module 标记类定义, 同时使用 @Provides 标记对外提供依赖的方法。经过 @Component 定义后, Dagger2 帮我们生成了以 Dagger 作为前缀的类, 可以通过 builder() 函数获得它的实例, 并设置依赖, 最后调用 build() 函数生成 CoffeeShop 的一个实例, 语句如下。

```
CoffeeShop coffeeShop = DaggerCoffeeShop.builder()
    .dripCoffeeModule(new DripCoffeeModule())
    .build();
```

27.3.5 框架的对比

ButterKnife 是纯粹的 View 注入框架, 并不是严格意义上的依赖注入框架, 如果仅仅需要 View 的注入, 那么建议选择 ButterKnife, 因此不把它列入以下的比较中。

RoboGuice 虽然在 3.0 版本中开始尝试使用编译时注解提升性能, 但框架实现中还是有很多地方用到反射机制, 因此性能提升空间还很大, 而且 RoboGuice 的使用代码侵入性比较大, 包大小也是这几个里面最大的。

Dagger 虽然使用编译时注解处理器实现编译时的注入, 但在对象图的构建时还是使用到了反射机制, 它会在运行的时候检测依赖注入是否正常工作, 因此会损耗一部分的性能, 同时也会导致调试困难。

Dagger2 和 Dagger 最大的区别是对象图的验证, 配置以及预先设置等方面抛弃了反射机制的使用, 转而是在编译阶段完成, 因此相比 Dagger, 性能得到了更大的提升。当然, 由于没有使用反射, Dagger2 没有实现动态机制, 缺乏灵活性。

从性能上面来说, Dagger2 优于 Dagger, Dagger 优于 RoboGuice, 详细的微基准测试可以参见 NimbleDroid 的这个 Demo¹。

¹ <https://github.com/NimbleDroid/DIDemoApps>

第28章

Android世界的Swift： Kotlin在Android中的应用

Kotlin¹ 是一门类似于 Swift 的基于 JVM 的通用编程语言，它由 JetBrains² 设计并开源，可用于服务器程序开发，桌面客户端程序开发以及移动应用开发（Android）。对于 JetBrains，多数人可能不是很熟悉，但如果说起它的知名产品 IntelliJ IDEA，你就应该知道了，而 Android Studio 正是基于 IntelliJ IDEA 修改而来的。因此，Android Studio 可以很容易的集成并使用 Kotlin 进行 Android 应用的开发。使用 Kotlin，我们可以很容易的在 Android 工程中替代 Java，或者与 Java 混合使用。

Kotlin 是一门包含很多函数式编程思想的面向对象编程语言，它生来就是为了弥补 Java 语言缺失的现代语言的特性，并极大地简化代码，使得开发者可以尽量少地编写样板代码。因此，它正受到越来越多的 Java 开发者的青睐。在笔者撰写本书时，适逢 Kotlin 1.0 正式版发布。

28.1 选择 Kotlin 的原因

- 相对而言更平的学习曲线：例如相比 Scala 而言，我们可以学得更快。Kotlin 限制比较多，但如果你之前没有使用过现代编程语言，那么使用 Kotlin 入门会更容易。
- 轻量级：相比其他编程语言，Kotlin 函数库更小。由于 Android 存在 64K 方法数限制问题，这使得这一点更为重要。虽然使用 proguard 或者打包成多个 dex 能够解决这个问题，但是所有这些解决方案都会增加系统复杂性，并增加调试的时间或者 APP 启动时间。Kotlin 函数库方法数小于 7000 个，相当于 support-v4 的大小。

¹ <http://kotlinlang.org/>

² <https://www.jetbrains.com/>

- 高度可互操作且无缝兼容 Java: Kotlin 和 Java 的亲密度非常高,可以和其他 Java 类库很好并且简单地实现互操作。Kotlin 团队在开发这门新语言时正是秉承了这个中心思想,他们希望可以使用 Kotlin 继续开发现有使用 Java 语言编程的工程,而不是重写所有代码。因此 Kotlin 需要能够极好的和 Java 互操作。
- 完美的集成 Android Studio 以及 Gradle: Kotlin 有一个专门用于 Android Studio 的插件,因此在 Android 工程中使用 Kotlin 只需简单的配置。

28.2 Kotlin 的安装和配置

想要在 Android Studio 中进行 Kotlin 的开发很简单,需要安装一个插件。首先打开 Android Studio 的 Preferences 页面,在左边窗口选中 Plugins,这时打开了插件管理页面,如图 28-1 所示。



图 28-1

点击下方的 Install JetBrains plugin... 按钮,打开 JetBrains 插件页面,搜索 Kotlin,可以找到三个 Kotlin 相关的插件,如图 28-2 所示。

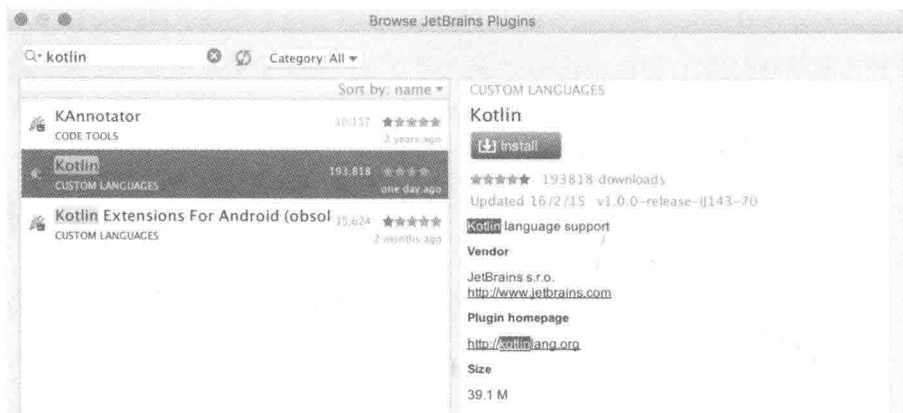


图28-2

其中：

- **Kotlin**: Kotlin 语言的支持包，安装这个插件，我们才能在 Android Studio 中识别 Kotlin 的语法。
- **Kotlin Extensions For Android(obsolete)**: Kotlin 的 Android 扩展插件，方便开发者使用 Kotlin 进行 Android 开发，这个插件被标记为过时的，它提供的功能已经集成在 Kotlin 插件中。
- **KAnnotator**: Kotlin 的注解扩展插件。

可以看到，只有 Kotlin 插件是必须的，其他两个插件基本可以忽略掉。插件安装完成之后，重启 Android Studio，就可以开始使用 Kotlin 编写 Android 应用了。

28.3 Kotlin 语言的特性

28.3.1 可表达性

使用 Kotlin 可以很容易避免样板代码的编写，因为语言本身已经默认覆盖了大多数典型的情况。例如，在 Java 中如果要创建一个典型的数据模型类，我们需要编写（或者至少生成）如下代码。

```
public class Artist {
    private long id;
    private String name;
    private String url;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    @Override public String toString() {
        return "Artist{ " +
            "id=" + id +
            ", name=' " + name + '\'' +
            ", url=' " + url + '\'' +
```

```
        '}' ;
    }
}
```

如果使用 Kotlin 编写呢？语句如下。

```
data class Artist(
    var id: Long,
    var name: String,
    var url: String)
```

28.3.2 空类型安全

当我们使用 Java 进行开发时，大部分代码都是防守型的。如果不想在代码运行时得到非预期的 `NullPointerException` 异常，我们通常需要在之前不断地检测对象是否为空，或者使用一个默认的空对象。类似其他很多编程语言，Kotlin 是空类型安全的，所有变量默认被修饰为“不可为 `null`”，必须显式在类型后添加？修饰符才可赋值为 `null`。因此我们需要使用安全调用操作符显式指明对象是否能够为空，看看下面的例子和注释能够更好地帮助我们理解。

```
// 编译不通过，因为notNullArtist不能为null
var notNullArtist: Artist = null

//编译通过，artist可以为null
var artist: Artist? = null

// 编译不通过，因为artist可能为空，这时我们需要对null进行处理
artist.print()

// 只有当artist != null时才会调用print()函数
artist?.print()

// 这也是一种编译通过的用法
if (artist != null) {
    artist.print()
}
```

// 这种用法只有在确信artist不为null时才可使用, 否则会抛出异常

```
artist!!.print()
```

// 当artist为null时可以指定一个默认值

```
val name = artist?.name ?: "empty"
```

28.3.3 扩展函数

我们可以为任何已存在的类添加新函数, 相比我们工程中普遍存在的传统的工具类, 扩展函数更具可读性。例如, 我们可以为 `Fragment` 类添加一个新函数, 用于显示一个 toast。

```
fun Fragment.toast(message: CharSequence, duration: Int = Toast.LENGTH_LONG)
{
    Toast.makeText(getActivity(), message, duration).show()
}
```

然后可以这样调用。

```
fragment.toast("Hello world!")
```

函数式支持 (Lambda), 函数是一级公民

使用 Java 语言开发 Android 时, 每次当我们创建一个新的 listener 时, 都需要声明一个 `onClick` 函数用于处理监听回调, 如果使用 Kotlin, 我们可以直接编写监听回调的代码而不再通过匿名对象传递 `onClick` 方法, 这个特性被称为 Lambda 表达式, 语句如下。

// Java 的写法

```
view.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View view) {
        Toast.makeText(this, "Hello World!", Toast.LENGTH_LONG).
show();
    }

});
```

// Kotlin 的写法

```
view.setOnClickListener({
    Toast.makeText(this, "Hello World!", Toast.LENGTH_LONG).show();
})
```

28.4 Kotlin 的 Gradle 配置

使用 Android Studio 新建一个空的 Android 工程，用鼠标右键单击工程包名并在弹出的菜单中选择 New → Kotlin File/Class，如图 28-3 所示。



图28-3

在弹出的对话框中选择创建一个名为 KotlinActivity 的 Class，创建完成之后，Android Studio 提示我们需要配置 Kotlin 相关类库和插件，如图 28-4 所示。



图28-4

点击 Configure 按钮，会弹出一个 Kotlin 配置对话框，如图 28-5 所示。



图28-5

在这里我们可以选择使用的 Kotlin 插件版本, 以及把 Kotlin 配置信息应用到哪些 modules 中。配置完成之后, 可以发现 app/build.gradle 文件中新增了如下内容。

```
...
apply plugin: 'kotlin-android'

android {
    ...
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}

dependencies {
    ...
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}

buildscript {
    ext.kotlin_version = '1.0.0'
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_
version"
```

```
    }  
}  
repositories {  
    mavenCentral()  
}
```

可以看到，Android Studio 在 app/build.gradle 中自动配置了 Kotlin 相关插件和变量，主要步骤如下。

- 添加 Kotlin 的 Gradle 插件依赖。

```
buildscript {  
    ext.kotlin_version = '1.0.0'  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_  
version"  
    }  
}
```

- 应用 Kotlin 插件。

```
apply plugin: 'kotlin-android'
```

- 添加 Kotlin 标准函数库到工程的依赖中。

```
dependencies {  
    ...  
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"  
}
```

- 配置 Kotlin 源文件夹（与 Java 源文件夹区分开，需要手动创建）。

```
android {  
    ...  
    sourceSets {  
        main.java.srcDirs += 'src/main/kotlin'    }  
}
```

```

    }
}

```

完成之后，我们得到一个名为 MainActivity.kt 的 Kotlin 类，最终工程的目录结构如图 28-6 所示。

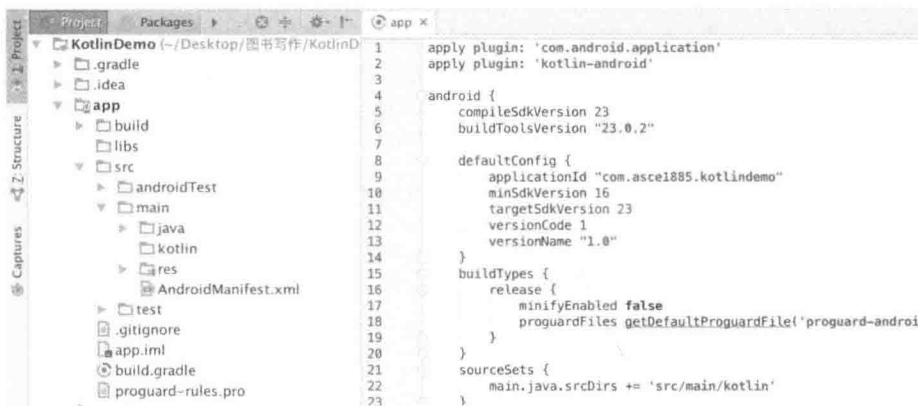


图28-6

28.5 将 Java 类转换成 Kotlin 类

Kotlin 插件能够自动将现有的 Java 类转换成 Kotlin 类，方法很简单，只需要选中 Java 类，并选中菜单 code-Convert Java File to Kotlin File 即可。转换前默认的空 Activity Java 类如下。

```
package com.ascel885.kotlindemo;
```

```
import android.os.Bundle;
```

```
import android.support.v7.app.AppCompatActivity;
```

```
import android.view.Menu;
```

```
import android.view.MenuItem;
```

```
public class MainActivity extends AppCompatActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

转换后对应的 Kotlin 类如下。

```
package com.ascel885.kotlindemo

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
```



```

import android.view.Menu
import android.view.MenuItem

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is
present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        val id = item.itemId

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true
        }

        return super.onOptionsItemSelected(item)
    }
}

```

通过前面的代码，我们可以看出有一些明显的区别。

- 继承的时候使用冒号:而不是使用 extends 关键字:这一点跟 C++ 语言类似。
- 显式的使用 override 关键字:在 Java 中我们使用 @override 注解使得代码更简洁,但 Java 并不强制使用。在 Kotlin 中,必须显式使用。
- 使用 fun 来声明函数:Kotlin 是一种面向对象的函数式语言,因此类似 Scala 等语言,方法将会使用函数来代表。
- 函数参数使用不同的命名法:参数类型和名字顺序是反过来的,中间以冒号:分隔,类似 Swift 语言。
- 分号;是可选的:我们不需要以分号结束当前行,当然如果想要的话也可以。但如果我们不用加分号的话,将节省很多时间,并使代码更简洁。

28.6 相关资料

- Kotlin 是一门开放源码的语言,基于 Apache 2.0 Open-Source license 开发的,代码托管在 Github¹上,感兴趣的同学可以阅读源码并作出自己的贡献。
- Anko函数库²:JetBrains 推出的一个简化基于 Kotlin 进行 Android 开发的函数库,使用 Kotlin 编写实现,最突出的特点在于抛弃了传统的基于 xml 文件实现 UI 的方式,而是使用一种类似 DSL 的方式声明 UI。使用 Anko,如果要声明一个垂直布局,里面包含一个 EditText 和一个 Button,可以如下声明,可以看到,完全看不到 xml 文件的痕迹。

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    verticalLayout {
        padding = dip(30)
        editText {
            hint = "Name"
            textSize = 24f
        }
        editText {

```

1 <https://github.com/JetBrains/kotlin>

2 <https://github.com/Kotlin/anko>

```
        hint = "Password"
        textSize = 24f
    }
    button( "Login" ) {
        textSize = 26f
    }
}
}
```

- 目前 Kotlin 相关的图书有两本: *Kotlin in Action*¹和*Kotlin for Android Developers*²。
- Kotlin相关资料合辑 awesome-kotlin³, 主要包括函数库与框架, 博客及视频资源等。

1 <https://www.manning.com/books/kotlin-in-action>

2 <https://leanpub.com/kotlin-for-android-developers>

3 <https://github.com/JavaBy/awesome-kotlin>

第29章

React Native For Android 入门指南

React Native¹ 是 Facebook 开源的一个框架，基于这套框架，我们可以使用 Javascript 语言编写 Android 和 iOS 的 APP，并且两个平台可以复用 70% ~ 80% 的代码，实现 “Learn once, write anywhere” 的目标。React Native 结合了 Web 应用和 Native 应用的优势，既具有 Web 应用的在线动态更新，又具有 Native 的原生体验，React Native 俨然已经成为移动开发的一个热门技术选型。本章将带大家了解 React Native 的基本知识，以及如何系统地学习这个框架。

29.1 环境配置

在开始 React Native 的学习之前，我们首先需要了解的是，使用 React Native 开发的 APP 只能运行在大于等于 Android 4.1 (API 16) 或大于等于 iOS 7.0 的手机操作系统上面。搭建 React Native 的开发环境涉及几个工具，这里以 Mac OS X 系统为例进行说明。

29.1.1 Homebrew²

Homebrew 是一个方便开发者在 MAC OS X 系统上面安装 Linux 工具包的 ruby 脚本，而 MAC OS X 已经内置了 ruby 的解释环境，因此安装 Homebrew 只需执行以下脚本。

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

¹ <https://facebook.github.io/react-native/>

² <http://brew.sh/>

更多关于 Homebrew 的用法请参见官方文档。关于 Homebrew 的作者，还有一个趣闻：白板编程没写出反转二叉树，Homebrew 作者被谷歌拒掉了¹。

29.1.2 nvm²

Node 版本管理器，是一个简单的 bash 脚本，用来管理同一台电脑上的多个 node.js 版本，并可实现方便的版本间切换。我们可以使用 Homebrew 来安装 nvm：brew install nvm，然后通过命令 vim \$HOME/.bashrc 打开 .bashrc 文件，添加如下配置。

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh" # This loads nvm
```

当然也可以选择官方的安装方法，就不用自己手动写 .bashrc 文件了。

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.26.1/install.sh
| bash
```

或者，

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.26.1/install.
sh | bash
```

这样配置之后，如果在命令行 Terminal 中输入 nvm 命令还是提示 command not found，需要再次输入 ~/.nvm/nvm.sh 命令激活 nvm。

29.1.3 Node.js³

基于 Chrome V8 JavaScript 引擎实现的一个 JavaScript 运行时，可用于方便地搭建响应速度快、易于扩展的网络应用。Node.js 使用事件驱动、非阻塞 I/O 模型而得以轻量 and 高效，非常适合在分布式设备上运行的数据密集型的实时应用。通过 nvm 安装 Node.js 的命令如下。

```
nvm install node && nvm alias default node
```

不过可能由于网络或者服务不稳定，实际上使用这个命令安装可能会失败，就算成功也会

¹ <http://www.nowcoder.com/discuss/572>

² <https://github.com/creationix/nvm>

³ <https://nodejs.org/>

花费较长的时间，因此建议到 Node.js 官网¹去直接下载 pkg 包，如图 29-1 所示。

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Important security releases, please update now!

Download for OS X (x64)

v4.4.0 LTS

Recommended For Most Users

v5.8.0 Stable

Latest Features

图 29-1

29.1.4 watchman²

Facebook 开源的一个文件监控服务，用来监视文件并且记录文件的改动情况，当文件变更时它可以触发一些操作，例如执行一些命令等。安装 watchman 是为了规避 node 文件监控的一个 bug，安装很简单，脚本如下。

```
brew install watchman
```

29.1.5 flow³

Facebook 出品的一个用于 JavaScript 代码静态类型检查的工具，用于找出 JavaScript 代码中的类型错误。Flow 采用 OCaml 语言开发，安装脚本如下。

```
brew install flow
```

安装完成之后，可以执行如下命令更新 Homebrew 的信息，并升级所有可以升级的软件。

```
brew update && brew upgrade
```

29.2 Android 开发环境的要求

打开 Android SDK Manager，确保如下工具和开发包已经安装。

1 <https://nodejs.org/en/>

2 <https://facebook.github.io/watchman/>

3 <http://www.flowtype.org/>

SDK :

- Android SDK Build-tools version 23.0.1。
- Android 6.0 (API 23)。
- Android Support Repository。

模拟器 :

- Intel x86 Atom System Image (for Android 5.1.1 – API 22)。
- Intel x86 Emulator Accelerator (HAXM installer)。

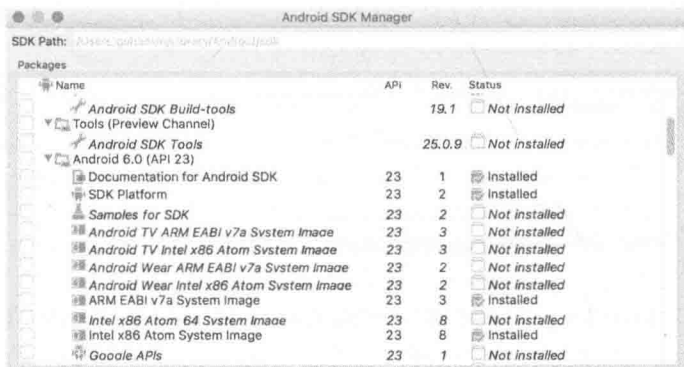


图29-2

29.3 React Native 工程配置

29.3.1 安装react-native

```
npm install -g react-native-cli
```

在 Terminal 中运行以上脚本，成功后，就可以在 Terminal 中使用 react-native 这个命令了，这个脚本只需执行一次。

29.3.2 生成工程

```
react-native init AwesomeProject
```

在 Terminal 中执行以上脚本，它会下载 React Native 工程源码和依赖，并在 AwesomeProject/iOS/AwesomeProject.xcodeproj 目录中创建 XCode 工程，在 AwesomeProject/android/app 创建 Android Studio 工程。生成的示例工程目录如图 29-3 所示。



图 29-3

至此，React Native 配置完成。

29.4 Android Studio 工程概览

使用 Android Studio 打开 AwesomeProject/android/app，Gradle 会去下载一系列依赖的函数包，这个过程视网速而定，可能会比较长时间。通过阅读源码我们可以发现，这些依赖的函数包主要有。

```
compile 'com.android.support:appcompat-v7:23.0.1'
compile 'com.android.support:recyclerview-v7:23.0.1'
compile 'com.facebook.fresco:fresco:0.8.1'
compile 'com.facebook.fresco:imagepipeline-okhttp:0.8.1'
compile 'com.facebook.stetho:stetho:1.2.0'
compile 'com.facebook.stetho:stetho-okhttp:1.2.0'
compile 'com.fasterxml.jackson.core:jackson-core:2.2.3'
compile 'com.google.code.findbugs:jsr305:3.0.0'
compile 'com.squareup.okhttp:okhttp:2.5.0'
compile 'com.squareup.okhttp:okhttp-ws:2.5.0'
compile 'com.squareup.okio:okio:1.6.0'
compile 'org.webkit:android-jsc:r174650'
```


不过如果我们反编译 AwesomeProject 生成的 APK，或者在 Android Studio 中查看 AwesomeProject 的 External Libraries，会发现事实上最终打进 APK 包的函数包不止上面这些，可以看到 External Libraries 的截图如图 29-4。



图 29-4

最终生成的 DEBUG 版本 APK 大小为 7.2M，体积还是比较大的。

当然，打开 app/build.gradle 文件，可以看到该 module 只依赖 react-native 的一个 aar 包，其他依赖的函数包对于开发者来说是透明的。

```
android {
    // ...
    defaultConfig {
        // ...
        ndk {
            abiFilters "armeabi-v7a", "x86"
        }
    }
}

dependencies {
    // ...
    compile 'com.facebook.react:react-native:0.11.+'
```

可以看到，React Native 默认只提供 armeabi-v7a 和 x86 这两个处理器架构的 .so 文件，其它类型架构的手机是以兼容模式运行。打开示例工程唯一的类 MainActivity，可以发现已经针对 React Native 做了一层封装调用，默认帮我们维护了 React Native 的生命周期。

```
package com.awesomeproject;

import android.app.Activity;
import android.os.Bundle;
import android.view.KeyEvent;

import com.facebook.react.LifecycleState;
import com.facebook.react.ReactInstanceManager;
import com.facebook.react.ReactRootView;
import com.facebook.react.modules.core.DefaultHardwareBackBtnHandler;
import com.facebook.react.shell.MainReactPackage;
import com.facebook.soloaders.Soloader;

public class MainActivity extends Activity implements
DefaultHardwareBackBtnHandler {

    private ReactInstanceManager mReactInstanceManager;
    private ReactRootView mReactRootView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mReactRootView = new ReactRootView(this);

        mReactInstanceManager = ReactInstanceManager.builder()
            .setApplication(getApplication())
            .setBundleAssetName("index.android.bundle")
            .setJSMainModuleName("index.android")
            .addPackage(new MainReactPackage())
            .setUseDeveloperSupport(BuildConfig.DEBUG)
```

```

        .setInitialLifecycleState(LifecycleState.RESUMED)
        .build();

        mReactRootView.startReactApplication(mReactInstanceManager,
        "AwesomeProject", null);

        setContentView(mReactRootView);
    }

    @Override
    public boolean onKeyUp(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_MENU && mReactInstanceManager !=
null) {
            mReactInstanceManager.showDevOptionsDialog();
            return true;
        }
        return super.onKeyUp(keyCode, event);
    }

    @Override
    public void invokeDefaultOnBackPressed() {
        super.onBackPressed();
    }

    @Override
    protected void onPause() {
        super.onPause();

        if (mReactInstanceManager != null) {
            mReactInstanceManager.onPause();
        }
    }

    @Override

```

```

protected void onResume() {
    super.onResume();

    if (mReactInstanceManager != null) {
        mReactInstanceManager.onResume(this);
    }
}
}

```

工程目录下的 index.android.js 是基于 React 写的 js 主模块，代码如下。

```

'use strict' ;

// 声明引用
var React = require( 'react-native' );
var {
    AppRegistry,
    StyleSheet,
    Text,
    View,
} = React;

// 这个页面的布局
var AwesomeProject = React.createClass({
    render: function() {
        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>
                    Welcome to React Native!
                </Text>
                <Text style={styles.instructions}>
                    To get started, edit index.android.js
                </Text>
                <Text style={styles.instructions}>
                    Shake or press menu button for dev menu
                </Text>
            </View>
        );
    }
});

```

```

        </Text>
      </View>
    );
  }
});

```

// 这个页面用到的样式

```

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});

```

// 注册这个页面

```
AppRegistry.registerComponent('AwesomeProject', () => AwesomeProject);
```

而 package.json 文件是工程的依赖和元数据配置文件。

```

{
  "name": "AwesomeProject",
  "version": "0.0.1",
  "private": true,

```

```
    "scripts": {
      "start": "node_modules/react-native/packager/packager.sh"
    },
    "dependencies": {
      "react-native": "^0.11.0"
    }
  }
}
```

29.5 React Native 依赖库修改为本地

公司的网络可能由于防火墙等原因，使用 Gradle 在线依赖去下载 React Native（以下简称 RN）的 .aar 包可能一天都下载不下来，如果让团队内其他同事也都每次等一天才下载下来，效率将是极其低下的，况且 RN 版本更新非常频繁。因此，有必要将 RN 的 .aar 包下载下来，并设置为本地依赖模式，同时引入 RN 依赖的第三方开源函数库。

我们先来看看默认的在线依赖 RN 的方式，很简单明了。

```
compile 'com.facebook.react:react-native:0.18.+'
```

29.5.1 下载 react-native.aar

RN 的 .aar 包托管在 Bintray 上面，因此，我们可以自己将其下载下来，从而将“在线依赖+下载”改为“手动下载+本地依赖”。打开 bintray¹ 搜索 react-native，可以查找到 React Native 的官方 bintray 网址²，如图 29-5 所示。在这个主页中，可以看到 RN 的一系列版本，最新的是 0.15.1。



图 29-5

1 <https://bintray.com/>
2 <https://bintray.com/bintray/jcenter/com.facebook.react%3Areact-native/view>

点击某个版本, 进入特定的版本页面, 选择 Files 标签, 可以看到如下图所示的文件列表, 在这个表中可以选择我们所需要的 react-native-0.18.0.aar 下载即可, 如图 29-6 所示。

Name	Updated	Size
react-native-0.18.0-javadoc.jar	2 个月前	1.4 MB
react-native-0.18.0-sources.jar	2 个月前	447.6 KB
react-native-0.18.0.aar	2 个月前	1.6 MB
react-native-0.18.0.pom	2 个月前	3.3 KB

图 29-6

29.5.2 react-native.aar 的文件内容

aar 文件本质上是一个压缩包, 可以将其后缀名由 .aar 修改为 .zip, 然后解压得到如图 29-7 所示目录。

名称	修改日期	大小
aapt	2015年9月24日 下午5:29	--
aidl	2016年1月4日 下午4:50	--
AndroidManifest.xml	2016年1月4日 下午4:50	404 字节
annotations.zip	2016年1月5日 下午6:08	486 字节
assets	2016年1月4日 下午4:50	--
classes.jar	2016年1月5日 下午6:19	754 KB
jni	2016年1月5日 下午6:08	--
libs	2015年9月24日 下午5:29	--
R.txt	2016年1月4日 下午4:50	66 KB
res	2016年1月4日 下午4:51	--

图 29-7

其中, classes.jar 就是 ReactAndroid 的代码, 使用 JD-GUI 工具可以一目了然。

而 jni 目录中存放的是 RN 依赖的 so 库, 可以看到, 为了减小使用 RN 的 APK 的包大小, 只支持了 armeabi-v7a 和 x86 两个处理器架构平台, 如图 29-9 所示。

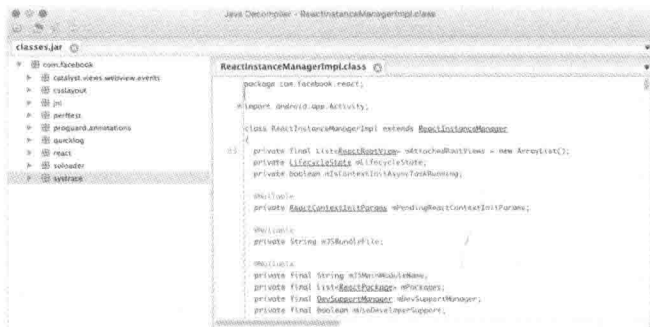


图 29-8

名称	大小	日期	时间	版本
jni	今天	下午 3:13		--
OS_Store	今天	下午 3:13		8 KB
armeabi-v7a	2016年1月5日	下午 6:08		--
libfb.so	2016年1月5日	下午 6:08		14 KB
libfbjni.so	2016年1月5日	下午 6:08		50 KB
libfolly_json.so	2016年1月5日	下午 6:08		112 KB
libglog.so	2016年1月5日	下午 6:08		79 KB
libgustl_shared.so	2016年1月5日	下午 6:08		651 KB
libreactnativejni.so	2016年1月5日	下午 6:08		96 KB
x86	2016年1月5日	下午 6:08		--
libfb.so	2016年1月5日	下午 6:08		5 KB
libfbjni.so	2016年1月5日	下午 6:08		75 KB
libfolly_json.so	2016年1月5日	下午 6:08		190 KB
libglog.so	2016年1月5日	下午 6:08		100 KB
libgustl_shared.so	2016年1月5日	下午 6:08		989 KB
libreactnativejni.so	2016年1月5日	下午 6:08		177 KB

图29-9

29.5.3 Gradle 本地依赖

把下载后的 aar 文件复制到工程的 app/libs 中，接着修改 build.gradle 文件，添加本地 aar 文件依赖的语句如下。

```
dependencies {
    // ...
    compile(name: 'react-native-0.18.0', ext: 'aar')
}
```

使用本地依赖方式，构建后 `react-native-0.18.0.aar` 不会自动下载 RN 所依赖的第三方函数库，需要我们手动配置，这些第三方函数库的下载方式可依法炮制。

那么如何查看 react-native-0.18.0.aar 依赖的第三方函数库呢？这个可以通过阅读 RN 的源码，打开源码路径下面的 ReactAndroid/src/build.gradle¹ 文件查看其 dependencies，语句如下。

1 <https://github.com/facebook/react-native/blob/master/ReactAndroid/build.gradle>


```
dependencies {
    compile fileTree(dir: 'src/main/third-party/java/infer-annotations/',
include: [ '*.jar' ])
    compile 'com.android.support:appcompat-v7:23.0.1'
    compile 'com.android.support:recyclerview-v7:23.0.1'
    compile 'com.facebook.fresco:fresco:0.8.1'
    compile 'com.facebook.fresco:imagepipeline-okhttp:0.8.1'
    compile 'com.facebook.stetho:stetho:1.2.0'
    compile 'com.facebook.stetho:stetho-okhttp:1.2.0'
    compile 'com.fasterxml.jackson.core:jackson-core:2.2.3'
    compile 'com.google.code.findbugs:jsr305:3.0.0'
    compile 'com.squareup.okhttp:okhttp:2.5.0'
    compile 'com.squareup.okhttp:okhttp-ws:2.5.0'
    compile 'com.squareup.okio:okio:1.6.0'
    compile 'org.webkit:android-jsc:r174650'
}
```

当然，完整的依赖项还不止这些，因为这些依赖可能还级连依赖其他的函数库，我们可以通过反编译 AwesomeProject（通过 `react-native init AwesomeProject` 命令生成的）生成的 APK，或者在 Android Studio 中查看 AwesomeProject 的 External Libraries，会发现事实上最终打进 APK 包的函数包如图 29-10 所示。

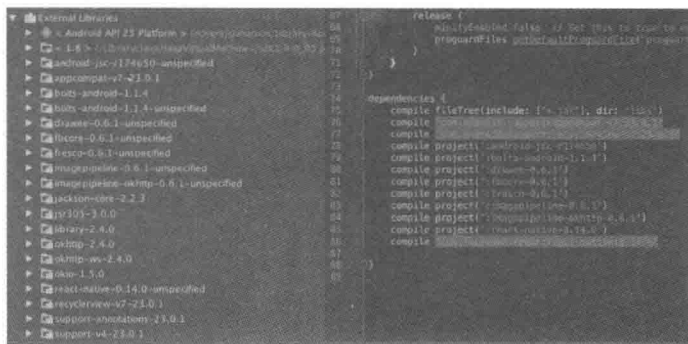


图29-10

因此，最终我们的 RN 本地依赖完整函数包如图 29-11 所示。

名称	修改日期
android-jsc-r174650.aar	昨天 下午8:46
bolts-android-1.1.4.aar	今天 下午1:52
drawee-0.6.1.aar	今天 下午1:43
fbcore-0.6.1.aar	今天 下午1:45
fresco-0.6.1.aar	今天 上午10:26
imagepipeline-0.6.1.aar	今天 下午1:47
imagepipeline-okhttp-0.6.1.aar	今天 上午10:27
jackson-core-2.2.3.jar	昨天 下午8:55
jsr305-3.0.0.jar	昨天 下午8:53
nineoldandroids-2.4.0.jar	今天 下午2:48
okhttp-2.4.0.jar	今天 上午10:28
okhttp-ws-2.4.0.jar	今天 上午10:29
okio-1.5.0.jar	今天 上午10:30
react-native-0.14.0.aar	今天 上午10:25

图29-11

对于 Jar 包，我们只需要将其拷贝到 app/libs 目录中，并在 app/build.gradle 文件中如下设置即可正常引用。

```
dependencies {
    compile fileTree(include: [ '*.jar' ], dir: 'libs' )
}
```

对于 .aar 包的本地依赖如何配置，可以参见本书第 8 章中关于 aar 函数库引用的相关内容。

29.5.4 将 node_modules 上传到 svn/git

我们使用 react-native init 命令得到的 RN 示例中包含一个 node_modules 目录，这个目录是开发 RN 必不可少的，如图 29-12 所示，包含了很多内容，在网络环境差的时候很难完整下载下来，而且以后引用的第三方 JS 函数库也会添加到这个目录中，因此我们也需要把这个目录共享到 svn 或者 git 上面，供团队内其他同事使用。

android	今天 下午4:59	--	文件夹
index.android.js	2015年10月27日 下午2:07	1 KB	JavaScr
index.ios.js	2015年10月27日 下午6:25	2 KB	JavaScr
ios	2015年10月27日 下午2:09	--	文件夹
node_modules	2015年10月20日 下午4:52	--	文件夹
react-native	2015年10月22日 下午2:25	--	文件夹
cli.js	2015年9月14日 下午8:57	129 字节	JavaScr
Examples	2015年10月9日 上午12:49	--	文件夹
jestSupport	2015年10月20日 下午4:50	--	文件夹
Libraries	2015年10月20日 下午4:50	--	文件夹
LICENSE	2015年9月14日 下午8:57	2 KB	文本编辑
local-cli	2015年10月20日 下午4:50	--	文件夹
node_modules	2015年10月20日 下午4:50	--	文件夹
package.json	2015年10月20日 下午4:50	4 KB	JSON
packager	2015年10月20日 下午4:50	--	文件夹
PATENTS	2015年9月14日 下午8:57	2 KB	文本编辑
React	2015年10月20日 下午4:50	--	文件夹
React.podspec	2015年10月9日 上午12:49	4 KB	文档
README.md	2015年9月24日 下午10:32	7 KB	Markd...
package.json	2015年10月27日 下午2:07	205 字节	JSON

图29-12

29.6 React Native 学习建议

React Native 是使用 Javascript 语言进行开发，同时基于 ReactJS 框架语法。因此，想要上手并熟练掌握 React Native 的开发，需要具备以下基本知识。

- 学习 Javascript 语言，同时建议学习最新的 ES6 语法规范。
- 学习 React 框架，了解 React 生命周期等基础知识。
- 学习 React Native 框架。
- 具备 Android、iOS 平台的基本开发常识。

第30章

Android在线热修复方案研究

在移动开发中，几乎没有一个 APP 在发布到应用市场后，在用户的使用过程中可以百分之百不会发生崩溃等严重错误。一般情况下，如果不是必现的崩溃，我们只需要控制 APP 的崩溃率在一定的比例之下就可以，但是某些情况下，由于测试不充分等原因导致发布后的 APP 出现经常性的崩溃，而且比例已经超过正常阈值。在以前面对这种情况，我们只能紧急修复出现问题的代码然后重新打包，测试并发布新版本到应用市场，同时通过 APP 自身的更新功能提示用户下载最新包重新安装。近一两年来随着 Android 在线热修复技术的出现，我们有了更快速方便的解决方案，在不需要重新发布新版本的前提下就可以将导致崩溃的 Bug 修复。

Android 在线热修复在近一两年得到较快的发展和普及，涌现了很多优秀的方案和相应的开源框架，总的来说，有三种可选的方案，它们各有优缺点，在实际项目中需要根据具体的业务需求来选择。解决方案虽然有多种，但在线热修复总的流程是不变的，在介绍这些方案和开源框架之前，我们先来看看热修复的基本流程。

30.1 在线热修复的基本流程

在本书第 25 章中我们了解到，每个 APP 都会集成一个用于捕获并统计线上 Crash 的 SDK，通过这个 SDK，我们可以知道 APP 的健康状况，这也是在线热修复的基础。从线上 Crash 的统计中可以得知 APP 是否存在需要快速在线热修复的严重性 Crash。如果存在，那么接着根据 Crash 日志定位分析出哪里的代码出的问题，这时需要基于这个发布的版本号对应的代码标签拉出一个 bugfix 的分支，并在这个分支上修复这个问题。问题修复后，开发同学自测通过，就可以提测给测试同学，测试同学回归发现问题已经解决后，开发同学就可以在自动化构建平台触发 bugfix 分支的构建，这个构建同时会执行前后两个版本 APK 的对比，生成补丁文件。补丁文件生成后，就需要想办法下发给出问题的 APP。这里有两种方法，一种是如果 APP 实现

了自己的推送功能，也就是和你的服务器保持了一个稳定的长链接，那么可以通过这个通道直接把补丁文件推送给用户手机上面的 APP。一种是你的 APP 并没有自己的长连接，那么只能通过某个时机让 APP 主动向服务器请求是否需要打补丁，如果需要就下载补丁文件，这个一般都是放在 APP 启动时去判断。当补丁文件生成后，就可以开始将 bugfix 分支修复的代码同步合并到主干分支，以确保后续版本不会存在相同的问题。至此，一个完整的在线热修复流程结束了，如图 30-1 所示。

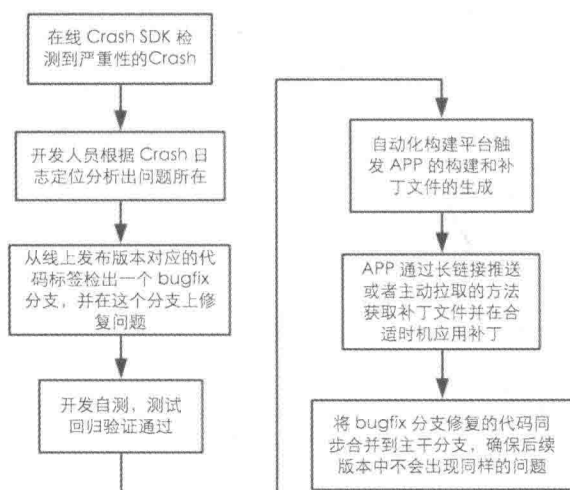


图 30-1

30.2 Dexposed¹

Dexposed 是阿里巴巴无线事业部开源的 Android 平台下的无侵入运行时 AOP 框架，该框架基于 AOP 思想，支持经典的 AOP 使用场景，可应用于日志记录、性能统计、安全控制、事务处理、异常处理等方面。针对 Android 平台，Dexposed 支持函数级别的在线热修复，本节会介绍它在在线热修复方面的应用，至于 AOP 的应用可以参见本书第 27 章。

Dexposed 是基于久负盛名的开源 Xposed² 框架实现的一个 Android 平台上功能强大的无侵入式运行时 AOP 框架，Dexposed 的 AOP 实现是完全非侵入式的，没有使用任何注解处理器，

¹ <https://github.com/alibaba/dexposed>

² <https://github.com/rovo89/Xposed>

编织器或者字节码重写器。集成 Dexposed 框架很简单，只需要在应用初始化阶段加载一个很小的 JNI 库。

Dexposed 实现的 hooking，不仅可以 hook 应用中的自定义函数，也可以 hook 应用中调用的 Android 框架的函数。基于动态类加载技术，运行中的 APP 可以加载一小段经过编译的 Java AOP 代码，在不需要重启 APP 的前提下实现修改目标 APP 的行为。

30.2.1 如何集成

集成方式很简单，只需要在线依赖一个 aar 包，其中包含一个名为 dexposedbridge.jar 的 Jar 包，以及两个 .so 文件：libdexposed.so 和 libdexposed_l.so。Gradle 依赖配置如下。

```
dependencies {
    compile 'com.taobao.android:dexposed:0.1.1@aar'
}
```

接着应该在应用初始化的地方（尽可能早的添加）添加初始化 Dexposed 的代码，例如在 MyApplication 中添加。

```
public class MyApplication extends Application {

    private boolean mIsSupported = false; // 设备是否支持dexposed

    private boolean mIsLDevice = false; // 设备Android系统是否是Android 5.0及以上

    @Override
    public void onCreate() {
        super.onCreate();

        // check device if support and auto load libs
        mIsSupported = DexposedBridge.canDexposed(this);
        mIsLDevice = Build.VERSION.SDK_INT >= 21;
    }
}
```

```

public boolean isSupported() {
    return mIsSupported;
}

public boolean isLDevice() {
    return mIsLDevice;
}
}

```

30.2.2 基本用法

在正式介绍 Dexposed 热修复的应用之前，我们有必要先来了解它的基本用法。在 Dexposed 中，对于某个函数而言，有三个注入点可供选择。

- 函数执行前注入（before）。
- 函数执行后注入（after）。
- 替换函数执行中的代码段（replace）。

分别对应于抽象类 XC_MethodHook 及其子类 XC_MethodReplacement 中的函数，语句如下。

```

public abstract class XC_MethodHook extends XCallback {

    /**
     * Called before the invocation of the method.
     * <p>Can use {@link MethodHookParam#setResult(Object)} and {@link MethodHookParam#setThrowable(Throwable)}
     * to prevent the original method from being called.
     */
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {}

    /**
     * Called after the invocation of the method.
     * <p>Can use {@link MethodHookParam#setResult(Object)} and {@link MethodHookParam#setThrowable(Throwable)}
     */
}

```

```

dHookParam#setThrowable(Throwable) }
    * to modify the return value of the original method.
    */
    protected void afterHookedMethod(MethodHookParam param) throws Throwable
    {}
}

public abstract class XC_MethodReplacement extends XC_MethodHook {

    @Override
    protected final void beforeHookedMethod(MethodHookParam param) throws
    Throwable {
        try {
            Object result = replaceHookedMethod(param);
            param.setResult(result);
        } catch (Throwable t) {
            param.setThrowable(t);
        }
    }

    protected final void afterHookedMethod(MethodHookParam param) throws
    Throwable {
    }

    /**
     * Shortcut for replacing a method completely. Whatever is returned/
    thrown here is taken
     * instead of the result of the original method (which will not be
    called).
     */
    protected abstract Object replaceHookedMethod(MethodHookParam param)
    throws Throwable;
}

```

可以看到，这三个注入回调函数都有一个类型为 `MethodHookParam` 的参数，这个参数包含

了一些很有用的信息。

- MethodHookParam.method: 要 hook 的函数。
- MethodHookParam.thisObject: 这个类的一个实例。
- MethodHookParam.args: 用于传递被注入函数的所有参数。
- MethodHookParam.setResult: 用于修改原函数调用的结果, 如果在beforeHookedMethod 回调函数中调用 setResult, 可以阻止对原函数的调用。但是如果有返回值的话仍然需要通过 hook 处理器进行 return 操作。

30.2.3 在线热修复

在线热更新一般用于修复线上严重的, 紧急的或者安全性的 bug, 这里会涉及到两个 APK 文件, 一个我们称为宿主 APK, 也就是发布到应用市场的 APK, 一个称为补丁 APK。宿主 APK 出现 bug 时, 通过在线下载的方式从服务器下载到补丁 APK, 使用补丁 APK 中的函数替换原来的函数, 从而实现在线修复 bug 的功能。

为了实现这个功能, 需要再引入一个名为 patchloader 的 Jar 包, 这个函数库实现了一个热修复框架, 宿主 APK 在发布时会把这个 Jar 包一起打包进 APK 中, 而补丁 APK 只是在编译时需要这个 Jar 包, 但打包成 APK 时不包含这个 Jar 包, 以免补丁 APK 集成到宿主 APK 中时发生冲突。因此, 补丁 APK 将会以 provided 的形式依赖 dexposedbridge.jar 和 patchloader.jar, 补丁 APK 的 build.gradle 文件中依赖部分脚本如下。

```
dependencies {
    provided files( 'libs/dexposedbridge.jar' )
    provided files( 'libs/patchloader.jar' )
}
```

这里我们假设宿主 APK 的 MainActivity.showDialog 函数出现问题, 需要打补丁, 宿主代码如下 (类完整路径是 com.taobao.dexposed.MainActivity)。

```
package com.taobao.dexposed;

public class MainActivity extends Activity {
```

```

private void showDialog() {
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle( "Dexposed sample" )
        .setMessage(
            "Please clone patchsample project to generate apk,
and copy it to \" /Android/data/com.taobao.dexposed/cache/patch.apk\" " )
        .setPositiveButton( "ok" , new DialogInterface.
OnClickListener() {
            public void onClick(DialogInterface dialog, int
whichButton) {
                }
            }).create().show();
    }
}

```

补丁 APK 只有一个名为 DialogPatch 的类, 实现了 patchloader 函数库中的 IPatch 接口, IPatch 接口代码如下。

```

/**
 * The interface implemented by hotpatch classes.
 */
public interface IPatch {
    void handlePatch(PatchParam lpparam) throws Throwable;
}

```

DialogPatch 类实现 IPatch 的 handlePatch 函数, 在该函数中通过反射得到宿主 APK 中 com.taobao.dexposed.MainActivity 类实例, 然后调用 dexposedbridge 函数库中的 DexposedBridge.findAndHookMethod 函数, 对 MainActivity 中的 showDialog 函数进行 Hook 操作, 替换宿主 APK 中的相应代码, DialogPatch 代码如下。

```

public class DialogPatch implements IPatch {

    @Override
    public void handlePatch(final PatchParam arg0) throws Throwable {
        Class<?> cls = null;
        try {

```

```

        cls= arg0.context.getClassLoader()
            .loadClass( "com.taobao.dexposed.MainActivity" );
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        return;
    }
    DexposedBridge.findAndHookMethod(cls, "showDialog",
        new XC_MethodReplacement() {
            @Override
            protected Object replaceHookedMethod(MethodHookParam param)
throws Throwable {
                Activity mainActivity = (Activity) param.thisObject;
                AlertDialog.Builder builder = new AlertDialog.
Builder(mainActivity);
                builder.setTitle( "Dexposed sample" )
                    .setMessage( "The dialog is shown from patch apk!" )
                    .setPositiveButton( "ok", new DialogInterface.
OnClickListener() {
                        public void onClick(DialogInterface dialog, int
whichButton) {
                            }
                        }).create().show();
                return null;
            }
        });
    }
}

```

最后宿主 APK 通过调用 patchloader 函数库提供的 PatchMain.load 函数来动态加载下载到的补丁 APK，加载代码如下。

```

// Run patch apk
public void runPatchApk(View view) {
    Log.d( "dexposed", "runPatchApk button clicked." );
}

```

```

        if (isLDevice) {
            showLog( "dexposed" , "It doesn' t support this function on L
device." );
            return;
        }
        if (!isSupport) {
            Log.d( "dexposed" , "This device doesn' t support dexposed!" );
            return;
        }
        File cacheDir = getExternalCacheDir();
        if(cacheDir != null){
            String fullpath = cacheDir.getAbsolutePath() + File.separator +
"patch.apk" ;
            PatchResult result = PatchMain.load(this, fullpath, null);
            if (result.isSuccess()) {
                Log.e( "Hotpatch" , "patch success!" );
            } else {
                Log.e( "Hotpatch" , "patch error is " + result.getErrorInfo());
            }
        }
        showDialog();
    }
}

```

30.2.4 平台的限制

虽然 Dexposed 的功能很强大,但由于 Android 系统版本碎片化太严重,目前 Dexposed 支持从 Android 2.3 到 4.4 (除了 3.0) 的所有 dalvid 运行时 arm 架构的设备,稳定性已经经过实践检验,完整的支持列表如下。

支持的 CPU 架构 :

- arm

支持的系统版本 :

- Dalvik 2.3
- Dalvik 4.0~4.4

不支持的系统版本：

- Dalvik 3.0
- ART 5.1
- ART M

测试中的系统版本：

- ART 5.0

未经测试的系统版本：

- Dalvik 2.2

30.3 AndFix¹

AndFix 同样出自阿里巴巴，不过跟 Dexposed 是不同的团队，是内部竞争的产物。AndFix 同样是基于 Xposed 的思想，相比 Dexposed，AndFix 是一个更纯粹的热修复框架，也就是说它唯一的功能就是热修复，而不像 Dexposed 还可以用于日志记录，性能监控等其他 AOP 领域。同时 AndFix 提供了更完善的工具用于生成补丁包，而不是需要开发者自己通过反射的方式去实现补丁函数。

30.3.1 如何集成

AndFix 的集成也很简单，只需在 build.gradle 文件中添加一行在线依赖。

```
dependencies {
    compile 'com.alipay.euler:andfix:0.4.0@aar'
}
```

一次完整的热修复可分为三个步骤，同理，补丁操作应该在应用启动的时候越早应用越好。

¹ <https://github.com/alibaba/AndFix>

- 初始化补丁管理器
- 加载补丁包
- 添加补丁包

代码如下。

```
public class MyApplication extends Application {
    private static final String TAG = "MyApplication";

    private static final String APATCH_PATH = "/out.apatch";

    private PatchManager mPatchManager; // 补丁管理器

    @Override
    public void onCreate() {
        super.onCreate();
        // 初始化补丁管理器
        mPatchManager = new PatchManager(this);
        mPatchManager.init("1.0");
        Log.d(TAG, "inited.");

        // 加载补丁包
        mPatchManager.loadPatch();
        Log.d(TAG, "apatch loaded.");

        // 在运行时动态添加补丁包, 补丁立即生效
        try {
            // 补丁文件 .apatch 的路径
            String patchFileString = Environment.
getExternalStorageDirectory()
                .getAbsolutePath() + APATCH_PATH;
            mPatchManager.addPatch(patchFileString);
        } catch (IOException e) {
            Log.e(TAG, "", e);
        }
    }
}
```

```

    }
}

```

30.3.2 补丁包生成工具

AndFix 提供了一个名为 apkpatch 的补丁包生成工具，在使用这个工具生成补丁包之前，我们需要准备。

- 发布到应用市场上的 APK 包，这个 APK 中存在需要修复的 bug。
- 修复了 bug 的新的 APK 包。
- 应用的 keystore 签名信息。

接着就可以执行 apkpatch 脚本，生成 .apatch 补丁文件，用法如下。

```
usage: apkpatch -f <new> -t <old> -o <output> -k <keystore> -p <***> -a
<alias> -e <***>
-a,--alias <alias>      keystore entry alias.
-e,--epassword <***>    keystore entry password.
-f,--from <loc>         new Apk file path.
-k,--keystore <loc>     keystore path.
-n,--name <name>        patch name.
-o,--out <dir>          output dir.
-p,--kpassword <***>    keystore password.
-t,--to <loc>           old Apk file path.
```

如果你的团队是模块化并行开发模式，那么如果某次热修复需要同时修复多个模块中的 bug，这时不同模块会生成相应的 .apatch 文件，在最终发布之前，可以使用如下命令来合并多个 .apatch 文件，用法如下。

```
usage: apkpatch -m <apatch_path...> -o <output> -k <keystore> -p <***> -a
<alias> -e <***>
-a,--alias <alias>      keystore entry alias.
-e,--epassword <***>    keystore entry password.
-k,--keystore <loc>     keystore path.
-m,--merge <loc...>    path of .apatch files.
```

```
-n,--name <name>      patch name.  
-o,--out <dir>         output dir.  
-p,--kpassword <***>  keystore password.
```

30.3.3 平台的限制

虽然都是基于 Xposed 思想，但 Dexposed 是直接基于 Xposed 基础上实现的，而 AndFix 只是参考其 Hook 的思想，目前 AndFix 比 Dexposed 做到了更好的系统兼容性。

支持的 CPU 架构：

- arm
- x86

支持的系统版本：

Android 2.3 到 6.0，同时支持 Dalvik 和 Art 虚拟机。

30.4 Nuwa¹

Nuwa 这种实现热修复的方案基于 ClassLoader 加载 dex 文件，来源于 QQ 空间终端开发团队，类似的开源框架还有 HotFix² 和 DroidFix³ 等，由于 Nuwa 的实现最完善，因此以它为例进行说明。

30.4.1 基本原理

Nuwa 的方案基于 Android 的 dex 分包基础上，如果你的 APP 曾经达到过 64K 方法数天花板问题的话，那么你应该对 dex 分包不陌生，简单来说，就是将一个 dex 文件拆分成多个 dex 文件，在应用启动时，其中一个作为主 dex 进行加载，应用启动后，将逐个加载其他从 dex。多个 dex 文件会排列成一个有序的数组，在 Dalvik 虚拟机加载类的过程中，会顺序遍历这些 dex 文件，在每个 dex 文件中查找对应的类，如果找到就返回，没有找到就继续在下一个 dex

¹ <https://github.com/jasonross/Nuwa>

² <https://github.com/dodola/HotFix>

³ <https://github.com/bunnyblue/DroidFix>

文件中查找。因此，理论上，如果多个 dex 文件中存在相同的类，那么排在数组前面的 dex 文件中的类将会被优先选择。因此，Nuwa 实现热修复就是通过将修复后的代码所在的 dex 插入到 dex 数组的最前面使用新类替换旧类的目的。

30.4.2 如何集成

首先在工程根目录的 build.gradle 文件中添加 Nuwa 的 Gradle 插件路径，语句如下。

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'cn.jiajixin.nuwa:gradle:1.2.2'
    }
}
```

接着在 Module 中的 build.gradle 文件中使用上述插件，同时添加 Nuwa 的在线依赖。

```
apply plugin: "cn.jiajixin.nuwa"

dependencies {
    compile 'cn.jiajixin.nuwa:nuwa:1.0.0'
}
```

在代码中使用 Nuwa 加载补丁文件很简单，为了确保尽可能早的初始化它，我们在应用的 Application 类的 attachBaseContext 函数中调用它的初始化函数。

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    Nuwa.init(this);
}
```

接着在合适的时机加载补丁文件，从而实现热修复。

```
Nuwa.loadPatch(this, patchFile);
```

30.4.3 补丁生成工具

Nuwa 提供一个名为 Nuwa Gradle¹ 的插件来实现补丁的生成，可以很方便的集成到自动化构建系统中，主要提供了两个 Gradle Task 来生成补丁。

- `nuwaPatches`: 默认为所有 Variant 分别生成一个补丁文件 `patch.jar`。
- `nuwa${variant.name.capitalize()}Patch`: 为指定的 Variant 生成一个补丁文件 `patch.jar`。

30.4.4 平台的限制

Nuwa 支持的平台从 Android 2.3 到 Android 6.0，同时支持 Dalvik 和 Art 虚拟机，对处理器架构也没有作限制。

30.5 总结

以上讨论的三种方案各有千秋，需要根据自己的业务需求进行选择，其中 Dexposed 暂时由于只支持 Android 5.0 以下的系统版本，不能覆盖所有的系统，因此，虽然功能很强大，但并不适合用来作通用的热修复选型。

AndFix 和 Nuwa 同时支持 Android 2.3 到 Android 6.0 系统，但从稳定性方面来讲，由于 AndFix 是通过修复 JNI 层函数实现函数替换的，因此稳定性不如 Nuwa，而且从函数库的大小上，Nuwa 也占足了优势，Nuwa 唯一的缺点是需要在下一次应用启动后热修复才能生效，而且会稍微影响到应用的启动速度，但总的来说，Nuwa 的方案应该是首选。

¹ <https://github.com/jasonross/NuwaGradle>

第31章

面向切面编程及其在 Android 中的应用

AOP，全称为 Aspect Oriented Programming，即面向切面编程，在 JavaEE 领域是一个常见且广泛使用的概念，但在 Android 开发中 AOP 的使用并不多见，不过一旦你在 Android 开发中引入 AOP，你可以实现很多意想不到的功能。

31.1 AOP 的基本概念

AOP 是软件开发中的一个编程范式，通过预编译方式或者运行期动态代理等实现程序功能的统一维护的一种技术，它是 OOP（面向对象编程）的延续，利用 AOP 开发者可以实现对业务逻辑中的不同部分进行隔离，从而进一步降低耦合，提高程序的可复用性，进而提高开发的效率。AOP 能够实现将日志记录、性能统计、埋点统计、安全控制、异常处理等代码从具体的业务逻辑代码中抽取出来，放到统一的地方进行处理。AOP 涉及到得基本概念如下。

- 横切关注点（Cross-cutting concerns）：在面向对象编程中，经常需要在不同的模块代码中添加一些类似的代码，例如在函数入口处打印日志，在 View 的点击处添加点击事件的埋点统计，或者对一个函数进行性能监控。查看它的执行耗时等。在 AOP 中把软件系统分成两个部分：核心关注点和横切关注点，核心关注点就是业务逻辑处理的主要流程，而横切关注点就是上面所说的经常发生在核心关注点的多个地方，且基本相似的日志记录、埋点统计等。
- 连接点（Joint point）：在核心关注点中可能会存在横切关注点的地方，例如方法调用的入口，View 的点击处理等地方，在 AOP 中习惯称为连接点。

- 通知（Advice）：特点连接点处所执行的动作，也就是 AOP 织入的代码，典型的有。
 - before：在目标方法执行之前的动作。
 - around：替换目标代码的动作。
 - after：在目标方法执行之后的动作。
- 切入点（Pointcut）：连接点的集合，这些连接点可以确定什么时机触发一个通知。切入点通常使用正则表达式或者通配符语法表示，可以指定执行某个方法，也可以指定多个方法，例如指定标记了某个注解的所有方法。
- 切面（Aspect）：切入点和通知可以组合成一个切面。
- 织入（Weaving）：将通知注入到连接点的过程。

31.2 代码织入的时机

AOP 中代码的织入根据类型的不同，主要可以分为三类。

- 编译时织入：在 Java 类文件编译的时候进行织入，这需要通过特定的编译器来实现，例如使用 AspectJ 的织入编译器。
- 类加载时织入：通过自定义类加载器 ClassLoader 的方式在目标类被加载到虚拟机之前进行类的字节代码的增强。
- 运行时织入：切面在运行的某个时刻被动态织入，基本原理是使用 Java 的动态代理技术。

相应的，存在一系列不同的工具方便开发者实现上述各种织入方式，主要有：

- AspectJ¹：是一个面向切面的框架，它扩展了 Java 语言，定义了一套 AOP 语法，实现了一个专门的编译器来在编译期生成遵守 Java 字节码规范的 .class 文件。
- Javassist-android²：Javassist 是一个开源的用于分析、编辑和创建 Java 字节码的类库，Android 平台有相应的移植版本。

¹ <http://www.eclipse.org/aspectj/downloads.php>

² <https://github.com/crimsonwoods/javassist-android>

- ASMDEX¹: 类似于 ASM², 都是实现对字节码进行操作的函数库, 不同的是 ASM 操作的是 Java 字节码, 而 ASMDEX 操作的是 Android 平台的 DEX 字节码。
- DexMaker³: DexMaker 是运行在 Android Dalvik 虚拟机上, 用于编译期或者运行期生成适用于 Dalvik 虚拟机的字节码的函数库。

31.3 基于 AspectJ 实现 Android 平台的 AOP

本章接下来主要介绍如何使用 AspectJ 来实现 Android 平台上面的 AOP 编程, 选择 AspectJ 进行介绍, 除了因为其功能强大, 是一个完整的 AOP 框架之外, 还考虑到它支持编译期的代码织入, 这样不会影响到应用的性能, 同时 AspectJ 还提供了易于使用的 API。为了便于大家的理解, 我们选择 JakeWharton 的 Hugo⁴ 函数库作为例子。Hugo 是一个基于 AspectJ 实现的 AOP 日志记录函数库, 它提供了很好的关于如何在 Android 平台使用 AspectJ 实现 AOP 的例子。

31.3.1 Hugo 的用法简介

为了有直观的感受, 我们先来看看 Hugo 的使用步骤, 首先当然是引入 Hugo 所需的 Gradle 插件, 语句如下。

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.jakewharton.hugo:hugo-plugin:1.2.1'
    }
}

// 添加 Hugo 的 Gradle 插件
```

```
apply plugin: 'com.jakewharton.hugo' // 应用 Hugo 插件
```

- 1 <http://asm.ow2.org/asmdex-index.html>
- 2 <http://asm.ow2.org/index.html>
- 3 <https://github.com/crittercism/dexmaker>
- 4 <https://github.com/JakeWharton/hugo>

就这么简单，之后这个插件会帮我们下载一些依赖库，分别是：

- aspectjrt.jar: aspectJ 运行时的依赖库，想要使用 aspectJ 的功能都需要引入这个库，只有 119 KB 大小。
- hugo-annotations: Hugo 的注解库，后面会介绍到，只有 4KB 大小。
- hugo-runtime: Hugo 的运行时库，是实现 Hugo 日志功能的核心库，只有 8KB 大小。

可以看到，基于 aspectJ 实现 AOP 功能对应用的安装包大小影响微乎其微。Hugo 的使用很简单，在需要进行日志记录的类名或者方法名处使用 `@DebugLog` 注解标记即可，之后 Hugo 就会在编译时织入打印日志的代码，从而省去了开发者手动编写日志代码的繁琐。例如下面这个方法使用 `@DebugLog` 注解。

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

    printArgs( "The" , "Quick" , "Brown" , "Fox" );
}

@DebugLog
private void printArgs(String... args) {
    for (String arg : args) {
        Log.i( "Args" , arg);
    }
}
```

在程序运行的时候会打印出下面的日志信息，其中 `→ printArgs(args=["The" , "Quick" , "Brown" , "Fox"])` 和 `← printArgs [0ms]` 是 Hugo 这个函数库为我们自动添加的日志信息。

```
com.ascel885.hugodemo V/MainActivity: → printArgs(args=[ "The" , "Quick" ,
"Brown" , "Fox" ])
com.ascel885.hugodemo I/Args: The
com.ascel885.hugodemo I/Args: Quick
com.ascel885.hugodemo I/Args: Brown
```

```
com.ascel885.hugodemo I/Args: Fox
com.ascel885.hugodemo V/MainActivity: ← printArgs [0ms]
```

31.3.2 Hugo 的实现原理

通过查看 Hugo 的源码,我们发现在 Android Studio 中 Hugo 主要分为四个 Module,分别是。

- hugo-annotations: 包含 hugo 提供给开发者使用的注解定义,是一个纯的 Java Library。
- hugo-plugin: 包含 hugo 的 gradle 插件实现,基于 groovy 语言实现,是一个 Groovy Library。
- hugo-runtime: 包含 hugo 核心功能的实现,是一个 Android Library。
- hugo-example: 包含 hugo 的使用示例,是一个 Android 应用。

注解的定义

Hugo 以注解的方式来标记哪些地方需要使用日志功能,目前只有一个名为 @DebugLog 的注解,定义如下。可以看到,这个注解是编译时注解类型,可以作用于类(包含 enum)、接口(包含注解类型)、方法以及构造函数。

```
@Target({TYPE, METHOD, CONSTRUCTOR})
@Retention(CLASS)
public @interface DebugLog {
}
```

核心功能的实现

Hugo 核心功能的实现在于如何定义切面,也就是定义好切入点和通知,这些都要借助于 aspectjrt 函数库提供的 API,语句如下。

```
// 切面的定义
@Aspect
public class Hugo {

    // 切入点的定义
    @Pointcut("within(@hugo.weaving.DebugLog *)")
```

```

public void withinAnnotatedClass() {
}

@Pointcut( "execution(* *(..)) && withinAnnotatedClass()" )
public void methodInsideAnnotatedType() {
}

// 切入点是可以组合的
@Pointcut( "execution(*.new(..)) && withinAnnotatedClass()" )
public void constructorInsideAnnotatedType() {
}

@Pointcut( "execution(@hugo.weaving.DebugLog * *(..)) ||
methodInsideAnnotatedType()" )
public void method() {
}

@Pointcut( "execution(@hugo.weaving.DebugLog *.new(..)) || constructorIn
sideAnnotatedType()" )
public void constructor() {
}

// 通知的定义, 此处只需要使用到Around通知, 其他的还有Before, After等类型
@Around( "method() || constructor()" )
public Object logAndExecute(ProceedingJoinPoint joinPoint) throws
Throwable {
    enterMethod(joinPoint); // 这里在原来方法执行之前打印Hugo的日志

    long startNanos = System.nanoTime();
    Object result = joinPoint.proceed(); // 这里是调用原来的方法
    long stopNanos = System.nanoTime();
    long lengthMillis = TimeUnit.NANOSECONDS.toMillis(stopNanos -
startNanos);
}

```



```
exitMethod(joinPoint, result, lengthMillis); // 这里在原来方法执行之后打印Hugo的日志
```

```
        return result;
    }
    ...
}
```

Gradle 插件的实现

Hugo Gradle 插件的主要功能是利用 aspectJ 编译器对工程中 .java 源文件编译后的 .class 文件进行代码的织入处理。

```
import com.android.build.gradle.AppPlugin
import com.android.build.gradle.LibraryPlugin
import org.aspectj.bridge.IMessage
import org.aspectj.bridge.MessageHandler
import org.aspectj.tools.ajc.Main
import org.gradle.api.Plugin
import org.gradle.api.Project
import org.gradle.api.tasks.compile.JavaCompile

class HugoPlugin implements Plugin<Project> {
    // 在使用者的build.gradle中调用: apply plugin: 'com.jakewharton.hugo'
    // 实际上就是调用到插件的这个apply函数
    @Override
    void apply(Project project) {
        // Android Studio 工程中是否存在 Application Module
        // 即在build.gradle中应用了apply plugin: 'com.android.application' 的
module
        def hasApp = project.plugins.withType(AppPlugin)

        // Android Studio 工程中是否存在 Library Module
        // 即在build.gradle中应用了apply plugin: 'com.android.library' 的
module
```

```

def hasLib = project.plugins.withType(LibraryPlugin)
if (!hasApp && !hasLib) {
    throw new IllegalStateException( "' android' or 'android-
library' plugin required." )
}

final def log = project.logger
final def variants
// 获取module的build.gradle文件中定义的variants
if (hasApp) {
    variants = project.android.applicationVariants
} else {
    variants = project.android.libraryVariants
}

// 使用这个插件的工程需要引入的依赖，可以看到，Hugo只在Debug版本起作用
project.dependencies {
    debugCompile 'com.jakewharton.hugo:hugo-runtime:1.2.1'
    debugCompile 'org.aspectj:aspectjrt:1.8.5'
    compile 'com.jakewharton.hugo:hugo-annotations:1.2.1'
}

// 遍历每个variant
variants.all { variant ->
    // 对于非Debug版本，不进行代码织入
    if (!variant.buildType.isDebuggable()) {
        log.debug( "Skipping non-debuggable build type '${variant.
buildType.name}' ." )
        return;
    }

    JavaCompile javaCompile = variant.javaCompile
    // 在 Java 编译的最后阶段（模块中的字节码.class文件已经生成）实现代码的织入

```

```

javaCompile.doLast {
    // 调用aspectj的编译器ajc需要传入的参数
    String[] args = [
        "-showWeaveInfo",
        "-1.5",
        "-inpath", javaCompile.destinationDir.toString(),
        "-aspectpath", javaCompile.classpath.asPath,
        "-d", javaCompile.destinationDir.toString(),
        "-classpath", javaCompile.classpath.asPath,
        "-bootclasspath", project.android.bootClasspath.
join(File.pathSeparator)
    ]
    log.debug "ajc args: " + Arrays.toString(args)

    MessageHandler handler = new MessageHandler(true);
    new Main().run(args, handler);
// 调用aspectj的编译器ajc实现具体的代码织入

    // 打印消息
    for (IMessage message : handler.getMessages(null, true)) {
        switch (message.getKind()) {
            case IMessage.ABORT:
            case IMessage.ERROR:
            case IMessage.FAIL:
                log.error message.message, message.thrown
                break;
            case IMessage.WARNING:
                log.warn message.message, message.thrown
                break;
            case IMessage.INFO:
                log.info message.message, message.thrown
                break;
            case IMessage.DEBUG:
                log.debug message.message, message.thrown

```

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

31.4 其他 AOP 开源框架

通过上面的分析，我们应该可以在 Android Studio 中自己实现一个基于 AspectJ 的 AOP 框架，除了 Hugo，目前还有下面几个不错的开源函数库可供参考。

- [gradle-android-aspectj-plugin](#)¹
- [RoboAspectJ](#)²
- [gradle_plugin_android_aspectjx](#)³

实现的基本原理与上面分析的 Hugo 差不多，只不过有的增加了对 aar 包、Jar 包中代码的 AspectJ 支持，以及 Kotlin 的支持。

1 <https://github.com/uPhyca/gradle-android-aspectj-plugin>

2 <https://github.com/meituan/RoboAspectJ>

3 https://github.com/HujiangTechnology/gradle_plugin_android_aspectjx

第32章

基于Facebook Buck 改造Android构建系统

自从 Android 的 IDE 由 Eclipse 切换到 Android Studio，构建 APK 所用的工具也由 Ant 转换为 Gradle。从那时候起，我们就一直使用 Gradle 进行项目的构建，随着工程 Module 的不断增多，代码的一处改动，都要花费几分钟的时间编译并重新生成 APK，实在是浪费时间，一个可选的方案是引入插件化框架，每个模块独立成一个 APK，然后由插件化框架将所有模块的 APK 组装成统一的 APK，并对外提供服务，这方面的内容可以参见本书第 22 章。另一个方案就是替换构建工具，也就是使用本文的主角 Facebook Buck，来替换现有的 Gradle，这不是一件容易的事情。

Facebook Buck¹ 是一个构建系统，以 Google 内部构建系统 blaze 为模型，它是由前 Google，现 Facebook 工程师开发并在 Github 上面开源的，官网上关于 Gradle 和 Buck 的构建速度的一个对比如表 32-1。

表32-1

	Gradle	Buck	Speed Up
clean build	31s	6s	5x
incremental build	13s	1.7s	7.5x
no-op build	3s	0.2s	15x
clean install	7.2s	7.2s	1x
incremental install	7.2s	1.5s	4.8x

¹ <https://buckbuild.com/>

需要注意的是, Buck 当前只支持 Mac OS X 和 Linux。本章以 Mac OS X 平台为例进行介绍。

32.1 Buck环境配置

有两种方式可以下载 Buck。

32.1.1 Homebrew 方式

OS X 系统使用 Homebrew 方式安装 Buck 之前, 需要首先确保安装了 XCode 和 Command Line Tool¹, 并更新到最新版本, 接着在 Terminal 中执行如下命令获取最新版本的 Buck。

```
$ brew update
$ brew tap facebook/fb
$ brew install --HEAD buck
```

如果因为种种原因, 这种方式走不通的话, 建议尝试手动构建方式。

32.1.2 手动构建方式

手动构建就是从 Buck 源码进行编译安装, 首先需要确保你的 OS X 满足以下条件。

- Oracle JDK 7²
- Apache Ant 1.8 及以上版本³
- Python 2.6 或者 2.7⁴
- Git⁵
- C 编译器: gcc⁶或者 clang⁷

1 <https://developer.apple.com/xcode/download/>

2 <http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

3 <http://ant.apache.org/>

4 <https://www.python.org/downloads/>

5 <http://git-scm.com/download>

6 <http://gcc.gnu.org/>

7 http://clang.llvm.org/get_started.html

在具备以上环境之后，就可以从 Github 上面检出 Buck 的源码然后进行编译安装了，在 Terminal 中执行如下命令。

```
$ git clone https://github.com/facebook/buck.git
$ cd buck
$ ant
$ ./bin/buck --help
```

其中 Buck 的源码比较大，压缩包接近 200M，所以网络不佳的话 git clone 可能会等待很长时间，也很容易出错，需要耐心等待和重试。

如果一切正常的话，你将会在 Terminal 中得到如下日志信息。

```
buck build tool
usage:
  buck [options]
  buck command --help
  buck command [command-options]

available commands:
  audit      lists the inputs for the specified target
  build      builds the specified target
  clean      deletes any generated files
  fetch      downloads remote resources to your local machine
  install    builds and installs an application
  project    generates project configuration files for an IDE
  query      provides facilities to query information about the target

nodes graph
  quickstart generates a default project directory
  server      query and control the http server
  targets     prints the list of buildable targets
  test        builds and runs the tests for the specified target
  uninstall  uninstalls an APK

options:
  --help      : Shows this screen and exits.
  --version (-V) : Show version number.
```

由于以后会频繁使用到 Buck 安装目录下面的 bin/buck 命令，所以为了方便在 Terminal 中直接执行 Buck 命令，需要将该目录添加到环境变量中，在 Terminal 中进入到用户主目录，打开 .bash_profile 文件，命令如下。

```
$ cd ~  
$ vim ~/.bash_profile
```

添加配置信息。

```
export PATH=$HOME/buck/bin:$PATH
```

接着执行如下命令使该配置立即生效。

```
$ source ~/.bash_profile
```

32.1.3 安装 Watchman¹

Watchman 是 Facebook 开源的一个文件监控服务，用来监视文件并且记录文件的改动情况，当文件变更它可以触发一些操作，例如执行一些命令等。安装 Watchman，是为了避免 Buck 每次都去解析 build files²，同时可以缓存其他一些东西，减少编译时间。Watchman 安装很简单，脚本如下。

```
$ brew install watchman
```

32.1.4 安装 Android SDK 和 Android NDK

相信作为 Android 开发者，电脑上应该都已经有了这两个函数库，需要注意的一点是，使用 Buck 编译 Android 应用的代码，需要在用户主目录的 .bash_profile 文件中配置这两个函数库的环境变量，语句如下。

```
export ANDROID_HOME=/Users/guhaoxin/Library/Android/sdk/  
export ANDROID_NDK=/Users/guhaoxin/Library/Android/android-ndk-r10/
```

如果你的系统存在多个版本的 NDK，那么也可以配置 ANDROID_NDK_REPOSITORY 变量指向包含多个 NDK 版本的目录。

¹ <https://facebook.github.io/watchman/>

² https://buckbuild.com/concept/build_file.html

32.2 快速创建基于 Buck 构建的 Android 工程

使用 `touch .buckconfig && buck quickstart` 命令可以快速创建一个基于 Buck 构建的 Android 工程，该命令执行过程中会要求你补全如下两个参数的值。

- `--dest-dir`: 生成的 Android 工程的目录。
- `--android-sdk`: 电脑上 Android SDK 的目录。

Terminal 中执行的日志信息如下。

```
guhaoxindeMacBook-Pro:~ guhaoxin$ touch .buckconfig && buck quickstart
Buck does not appear to have been built -- building Buck!
All done, continuing with build.
Using watchman.
Using buckd.
Enter the directory where you would like to create the project:  /Users/
guhaoxin/Desktop/BuckDemo
Enter your Android SDK's location: /Users/guhaoxin/Library/Android/sdk/
Thanks for installing Buck!
```

In this quickstart project, the file `apps/myapp/BUCK` defines the build rules.

At this point, you should move into the project directory and try running:

```
buck build //apps/myapp:app
```

or:

```
buck build app
```

See `.buckconfig` for a full list of aliases.

If you have an Android device connected to your computer, you can also try:

```
buck install app
```

This information is located in the file README.md if you need to access it later.

guhaoxindeMacBook-Pro:~ guhaoxin\$

生成的 Android 工程目录结构如图 32-2 所示。

名称	修改日期	大小	种类
apps	今天 下午5:45	--	文件夹
myapp	今天 下午5:45	--	文件夹
AndroidManifest.xml	今天 下午5:34	526 字节	XML 文本
BUCK	今天 下午5:34	316 字节	文本编辑 文稿
debug.keystore	今天 下午5:34	2 KB	文稿
debug.keystore.properties	今天 下午5:34	73 字节	Xcode 文稿
java	今天 下午5:45	--	文件夹
com	今天 下午5:45	--	文件夹
example	今天 下午5:45	--	文件夹
activity	今天 下午5:45	--	文件夹
BUCK	今天 下午5:34	197 字节	文本编辑 文稿
MyFirstActivity.java	今天 下午5:34	246 字节	Java 源代码
local.properties	今天 下午5:34	44 字节	Xcode 文稿
README.md	今天 下午5:34	460 字节	Markdown
res	今天 下午5:45	--	文件夹
AndroidManifest.xml	今天 下午5:34	666 字节	XML 文本
com	今天 下午5:45	--	文件夹
example	今天 下午5:45	--	文件夹
activity	今天 下午5:45	--	文件夹
BUCK	今天 下午5:34	198 字节	文本编辑 文稿
res	今天 下午5:45	--	文件夹

图32-2

可以看到，每个目录下面都有一个名为 BUCK 的配置文件，我们先预览下 myapp 目录下面的 BUCK 文件内容，目前有个初步印象就好。

```
android_binary(  
  name = 'app',  
  manifest = 'AndroidManifest.xml',  
  keystore = ':debug_keystore',  
  deps = [  
    '//java/com/example/activity:activity',  
  ],  
)
```

```
keystore(  
  name = 'debug_keystore',
```

```

store = 'debug.keystore' ,
properties = 'debug.keystore.properties' ,
)

project_config(
    src_target = ':app' ,
)

```

进入到工程根目录，在 Terminal 中输入如下命令创建 IntelliJ 工程。

```
$ buck project --ide IntelliJ
```

日志记录如下，表明 IntelliJ 工程创建成功。

```

Using buckd.
Waiting for Watchman command [/usr/local/bin/watchman watch /Users/guhaoxin/
Desktop/BuckDemo/.]...
Timed out after 10000 ms waiting for Watchman command [/usr/local/bin/
watchman watch /Users/guhaoxin/Desktop/BuckDemo/.]. Disabling Watchman.
[-] PROCESSING BUCK FILES...FINISHED 0.3s
[+] GENERATING PROJECT...0.4s
Modified 8 IntelliJ project files.
:: Please resynchronize IntelliJ via File->Synchronize or Cmd-Opt-Y (Mac)
or Ctrl-Alt-Y (PC/Linux)
=== Did you know ===
* You can run `buck project <target>` to generate a minimal project just
for that target.
* This will make your IDE faster when working on large projects.
* See buck project --help for more info.
---* Knowing is half the battle!

```

32.3 Buck 的基本概念

使用 Facebook Buck 对已有的 Android 项目进行改造，首先需要理解 Buck 的一些基本概念，在这个的基础上，我们才能既快又好的进行改造工作。本节介绍四个主要的概念，它们对于编

写 Buck 构建脚本至关重要。

- 构建规则（Build Rule）。
- 构建目标（Build Target）。
- 构建文件（Build File）。
- 构建目标模式（Build Target Pattern）。

32.3.1 构建规则（Build Rule）

构建规则是从一组输入文件中生成输出文件的过程，Buck 已经内建了一系列构建规则，定义了构建 Android 代码时会用到的公共操作，例如：

- 编译基于 Android SDK 编写的 Java 代码，可以使用 `android_library`。¹
- 生成 Android APK 文件，可以使用 `android_binary`。²

构建规则是一个泛称，在构建文件中使用内建 Python 函数定义的规则以及创建的用于执行这个操作的 Java 对象，都称之为构建规则。每个构建规则至少需要具备如下三个参数：

- `name`：构建规则的名字，在一个构建文件中必须保持唯一性。
- `deps`：构建规则的依赖项，以一个构建目标的列表的形式表示。
- `visibility`：可见性，表明其他构建规则是否可以依赖该构建规则，以一个构建目标模式的列表形式表示。

在 Buck 中，每一个构建规则可以产出零个或者一个输出文件，其他构建规则可以声明依赖这些输出文件。例如：

- 构建规则 `android_library` 的输出是一个 jar 文件，因此，构建规则 `android_binary` 可以声明依赖于规则 `android_library`，这样就可以在最终生成的 APK 文件中包含 `android_library` 输出的 Jar 包。
- 构建规则 `android_library B` 依赖于另一个规则 `android_library A`，那么在 B 编译时，会将 A 生成的 Jar 文件包含进来。

1 https://buckbuild.com/rule/android_library.html

2 https://buckbuild.com/rule/android_binary.html

3 https://buckbuild.com/concept/build_target_pattern.html

讲了这么多，我们还是来看一下构建规则的示例，这个示例是一个 `android_binary` 规则，它依赖于 `android_resource` 和 `android_library`，分别表示生成 APK 所需的资源和 Java 代码文件。

资源文件

```
android_resource(
    name = 'res' ,
    res = 'res' ,
    assets = 'assets' ,
)
```

Java 库文件

```
android_library(
    name = 'src' ,
    srcs = glob([ 'src/**/*.java' ]),
    deps = [
        ':res' ,
    ],
)
```

构建这个规则将会生成一个名为 `messenger.apk` 的文件

```
android_binary(
    name = 'messenger' ,
    manifest = 'AndroidManifest.xml' ,
    keystore = '//keystores:prod' ,
    package_type = 'release' ,
    proguard_config = 'proguard.cfg' ,
    deps = [
        ':res' ,
        ':src' ,
    ],
)
```

有一点需要明确，构建规则依赖的规则，会先于构建规则本身进行构建。构建规则和它的依赖之间是一个有向图，Buck 要求这个图是无环的，也就是最终形成一个有向无环图。这样

Buck 就可以对独立的子图进行并行构建，从而提高构建效率。

32.3.2 构建目标 (Build Target)

构建目标是一个字符串，用于标识工程中的某个构建规则，一个完整的构建目标示例是 `//java/com/facebook/share:ui`，主要有三个组成部分：

- `//` 前缀表示路径是相对于工程根目录。
- `java/com/facebook/share` 表示构建文件 BUCK 位于 `java/com/facebook/share` 目录中。
- 冒号后面的 `ui` 表示构建文件里面定义的构建规则名字，在一个构建文件中，构建规则的名字是唯一的。

在同一个构建文件中，可以使用构建目标的相对路径来引用它。构建目标的相对路径以冒号 `:` 开始，紧跟着三大组成部分中的第三部分：构建规则的名字。例如在构建文件 `java/com/facebook/share/BUCK` 中，`:ui` 可以用来引用 `//java/com/facebook/share:ui`。

```
# 这是 java/com/facebook/share/BUCK 文件的内容
java_binary(
  name = 'ui_jar' ,
  deps = [
    # 跟使用 '//java/com/facebook/share:ui' 效果一样
    ':ui' ,
  ],
)
```

构建目标经常作为构建规则的参数，并在 Buck 的命令行界面中使用。完整的情况下，使用命令行构建 Buck 时，我们需要输入完整的构建目标如下。

```
buck build //java/com/facebook/share:share
```

幸运的是，Buck 在解析命令行的构建目标时很宽松（解析构建文件中的构建目标时很严格），所以可以把前缀 `//` 去掉，一样可以正常构建。

```
buck build java/com/facebook/share:share
```

如果在冒号前面多了一个 `/`，同样也会被忽略，因此我们可以这么写。

```
buck build java/com/facebook/share/:share
```

可以注意到，其中的 `java/com/facebook/share/` 可以通过 `tab` 键帮我们自动补齐，不用手动一个字母一个字母的输入了。更进一步，如果冒号后面的构建规则名字和最后一个路径相同，也可以忽略。

```
# 相当于 //java/com/facebook/share:share.
```

```
buck build java/com/facebook/share/
```

32.3.3 构建文件 (Build File)

构建文件用于定义一个或者多个构建规则，统一命名为 BUCK。工程中的 Java 源文件只能被离它最近的 BUCK 文件引用，这里的“最近”指的是文件目录树中离该 Java 源文件所在目录最近的。例如，如果我们的工程中有如下所示的 BUCK 文件。

```
java/com/facebook/base/BUCK
java/com/facebook/common/BUCK
java/com/facebook/common/collect/BUCK
```

那么这些 BUCK 文件中定义的构建规则有如下限制：

- `java/com/facebook/base/BUCK` 文件中的构建规则只能引用 `java/com/facebook/base/` 目录中的源文件。
- `java/com/facebook/common/BUCK` 文件中的构建规则可以引用该目录中除了子目录 `java/com/facebook/common/collect/` 之外的所有Java源文件。
- `java/com/facebook/common/collect/` 目录中的Java源文件只能被 `java/com/facebook/common/collect/BUCK` 这个文件中的构建规则所引用。

构建文件所能引用到的源文件范围我们称之为构建包 `build package`，想要引用到其他构建包的源文件，需要在构建规则中使用 `deps` 引用这些文件。回到前面的例子，位于 `java/com/facebook/common/concurrent/` 的代码需要依赖 `java/com/facebook/common/collect/` 包下面的代码，假设 `java/com/facebook/common/collect/BUCK` 文件中有一个构建规则如下。

```
java_library(
    name = 'collect',
    srcs = glob([ '*.java' ]),
```

```

deps = [
    '//java/com/facebook/base:base' ,
],
)

```

那么 `java/com/facebook/common/BUCK` 文件应该定义规则如下。

```

java_library(
    name = 'concurrent' ,
    srcs = glob([ 'concurrent/*.java' ]),
    deps = [
        '//java/com/facebook/base:base' ,
        '//java/com/facebook/common/collect:collect' ,
    ],
)

```

特别强调一下，直接在 `srcs` 中引用 `concurrent` 包中的源文件是无效的。

```

java_library(
    name = 'concurrent' ,
    srcs = glob([ 'collect/*.java' , 'concurrent/*.java' ]),
    deps = [
        '//java/com/facebook/base:base' ,
    ],
)

```

32.3.4 构建目标模式 (Build Target Pattern)

构建目标模式是一个用来匹配一个或者多个构建目标的字符串，一般作为参数传递给构建规则中的 `visibility` 属性。一个构建目标其实也是一个构建目标模式，只是它匹配的是它自己。

```

# 匹配到 '//apps/myapp:app' .
'//apps/myapp:app'

```

以冒号 : 结尾的构建目标模式可以匹配到同一目录中的构建目标。

```

# 匹配到 '//apps/myapp:app_debug' 和 '//apps/myapp:app_release' .
'//apps/myapp:'

```

以 /... 结尾的构建目标模式可以匹配到该目录及其子目录中的构建目标。

```
# 匹配到 '//apps:common' and '//apps/myapp:app' .
'//apps/...'
```

32.4 项目改造实战

我们现有的工程是使用 Gradle 进行构建的，工程代码量比较庞大，方法数早已经突破 65K 限制，之前已经基于 Android Studio 的 Module 进行多模块的划分，已经形成十几个 Module，这将极大影响工程构建的速度，目前只要修改一两处代码，重新构建就要花费五六分钟，这简直不能忍了，开始 Buck 的改造吧！

首先我们来介绍一个概念：Exopackage。这是 Buck 的一个高级特性，正是基于这个特性，才实现了高速的迭代式 Android 构建。Exopackage 是一个很小的壳程序，它包含用于启动和加载一个 Android 应用所需的最少代码和资源，在运行时加载应用的代码可以避免一处 Java 代码修改就必须重启 APP 的困境，这显著地减少了修改验证应用功能的周期。将一个成熟的 Android 工程由 Gradle 构建修改为 Buck 构建，一般需要包含以下这些步骤。

32.4.1 步骤一：手动下载工程依赖的第三方 Jar包或者aar包

与 Gradle 不同，Buck 不支持在线引用第三方的 Jar 包或者 aar 包，也就是之前在 Gradle 中使用的 compile 在线依赖方式需要改成本地依赖方式，Maven Central 和 JCenter 方式在 Buck 中不可用。关于如何修改可以参考本书第 29 章。

32.4.2 步骤二：将 R.* 常量修改为非 final 的

在 Android Studio 的 Module 中其实我们已经进行过这方面的改造了，原因是 Android Library Module 的 R 类中的常量不是 final 的，因此，不能用在 switch...case 中的 case 语句中。遍历 Library Module 里面的 switch...case 语句，如果 case 语句中使用了 R.* 的值，那么需要将这个语句修改为 if...else 方式，下面是修改前的代码。

```
int id = view.getId();
switch (id) {
```

```
case R.id.button1:
    action1();
    break;
case R.id.button2:
    action2();
    break;
case R.id.button3:
    action3();
    break;
default:
    break;
}
```

修改后变成。

```
int id = view.getId();
if (id == R.id.button1) {
    action1();
} else if (id == R.id.button2) {
    action2();
} else if (id == R.id.button3) {
    action3();
}
```

原理可以参考 *Non-constant Fields in Case Labels*¹ 这篇文章。在 Buck 中，不仅 Library Module 需要进行这样的改造，Application Module 也一样需要。

32.4.3 步骤三：创建 BUCK 文件

在 Buck 中，构建规则是定义在名为 BUCK 的文件中的，下面我们以 Buck 官方提供的例子工程 AntennaPod² 为例来讲解完成一个完整的 BUCK 文件的创建的过程。

¹ <http://tools.android.com/tips/non-constant-fields>

² <https://github.com/AntennaPod/AntennaPod>

创建一个名为 all-jars 的 android_library 规则

将项目中用到的第三方 Jar 包统一放在项目根目录中的 libs 目录中，然后创建如下所示的 all-jars 规则，这个规则将暴露出 libs 目录下面的所有 Jar 包，并作为其他规则的依赖。

```
import re

jar_deps = []
for jarfile in glob([ 'libs/*.jar' ]):
    name = 'jars__' + re.sub(r' ^.*?/([^\/]+)\.jar$', r' \1' , jarfile)
    jar_deps.append( ':' + name)
    prebuilt_jar(
        name = name,
        binary_jar = jarfile,
    )

android_library(
    name = 'all-jars' ,
    exported_deps = jar_deps,
)
```

为每个本地 .aar 文件创建一个规则

使用 android_prebuilt_aar 规则来包装本地的 .aar 文件，语句如下。

```
android_prebuilt_aar(
    name = 'appcompat' ,
    aar = 'libs/appcompat-v7-23.2.0.aar' ,
)
```

定义从 .aidl 文件生成 .java 文件的规则

最后以 android_library 形式提供给使用者进行依赖。

```
presto_gen_aidls = []
for aidlfile in glob([ 'src/com/aocate/presto/service/*.aidl' ]):
    name = 'presto_aidls__' + re.sub(r' ^.*?/([^\/]+)\.aidl$', r' \1' ,
```

```
aidlfile)
    presto_gen_aidls.append( ':' + name)
    gen_aidl(
        name = name,
        aidl = aidlfile,
        import_path = 'src' ,
    )
```

```
android_library(
    name = 'presto-aidls' ,
    srcs = presto_gen_aidls,
)
```

创建一个 android_build_config 类型的规则

该规则用于生成 BuildConfig 文件，同样以 android_library 形式对外提供，这个类对于 Exopackage 的创建很重要。

```
android_build_config(
    name = 'build-config' ,
    package = 'de.danoeh.antennapod' ,
)
```

将工程中每个 Library Module 以 android_library 形式封装

Module dslv 资源的封装

```
android_resource(
    name = 'dslv-res' ,
    package = 'com.mobeta.android.dslv' ,
    res = 'submodules/dslv/library/res' ,
)
```

Module dslv 代码的封装

```
android_library(
    name = 'dslv-lib' ,
    srcs = glob([ 'submodules/dslv/library/src/**/*.java' ]),
```

```

    deps = [
        ':all-jars' ,
        ':dslv-res' ,
    ],
)

# Module presto 代码的封装
android_library(
    name = 'presto-lib' ,
    srcs = glob([ 'src/com/aocate/**/*.*.java' ]),
    deps = [
        ':all-jars' ,
        ':presto-aidls' ,
    ],
)

```

定义统一的 android_resource 和 android_library，封装所有 Module 的资源 and 代码

```

android_resource(
    name = 'res' ,
    package = 'de.danoeh.antennapod' ,
    res = 'res' ,
    assets = 'assets' ,
    deps = [
        ':appcompat' ,
        ':dslv-res' ,
    ],
)

android_library(
    name = 'main-lib' ,
    srcs = glob([ 'src/de/**/*.*.java' ]),
    deps = [
        ':all-jars' ,
    ],
)

```

```

        ':appcompat' ,
        ':build-config' ,
        ':dslv-lib' ,
        ':presto-lib' ,
        ':res' ,
    ],
)

```

一个完整的 APK 在安装到设备之前,我们需要使用 keystore 文件对其进行签名,在 Buck 中,定义如下。

```

keystore(
    name = 'debug_keystore' ,
    store = 'keystore/debug.keystore' ,
    properties = 'keystore/debug.keystore.properties' ,
)

```

打包 APK

BUCK 文件编写的最后一步是使用 android_binary 规则生成 APK 文件,如下所示。

```

android_binary(
    name = 'antennapod' ,
    manifest = 'AndroidManifest.xml' ,
    keystore = ':debug_keystore' ,
    deps = [
        ':main-lib' ,
    ],
)

```

最后为了便于在命令行中使用 Buck 构建我们的代码,同时提高构建的缓存,我们还需要在工程根目录中新建 .buckconfig 文件,并添加如下内容。

```

[alias]
    antennapod = //:antennapod
[cache]
    mode = dir

```

```

dir_max_size = 1GB
[android]
target = Google Inc.:Google APIs:19

```

至此，我们可以在命令行中执行 `buck build antennapod` 来构建这个 APP，或者使用 `buck install antennapod` 将 APK 安装到设备或者模拟器上。

32.4.4 步骤四：编译 Buck 的 buck-android-support

为了使用 Buck 的 Exopackage 来加速 Android 项目构建的过程，需要使用 Buck 提供的一个支持库，这个名为 buck-android-support 的支持库包含在 Buck 的源码中，在 Buck 源码的根目录中执行 `buck build buck-android-support` 命令即可编译生成这个支持库的 jar 文件。生成后，将其复制到 AntennaPod 工程根目录下面的 libs 目录即可。

32.4.5 步骤五：Exopackage 的使用

使用 Exopackage 需要对我们的工程进行一系列的改造，首先需要改造的是自定义的 Application 类，如果你的工程中存在的话。在这个例子工程中，自定义了一个名为 PodcastApp 的 Application 类。

```

public class PodcastApp extends Application {
    // ...
}

```

需要将其改造成如下格式。

```

public class PodcastApp extends DefaultApplicationLike {
    private final Application appContext;

    public PodcastApp(Application appContext) {
        this.appContext = appContext;
    }
}

```

也就是将原来的 PodcastApp 类由 Application 的子类改造成委托类。接下来我们需要创建一个新的继承自 ExopackageApplication 的 Application 类，语句如下。

```
public class AppShell extends ExopackageApplication {

    public AppShell() {
        super("de.danoeh.antennapod.PodcastApp",
            de.danoeh.antennapod.BuildConfig.EXOPACKAGE_FLAGS);
    }
}
```

其中，第一个参数中的字符串表示自定义 Application 的委托类，第二个参数中的 BuildConfig 的包名必须和前面在 android_build_config 规则中定义的包名一致。如果我们现有的工程中没有使用自定义的 Application 类，那么可以省略前面继承 ApplicationLike 的步骤，直接新建一个继承自 ExopackageApplication 的子类即可，语句如下。

```
public class AppShell extends ExopackageApplication {
    public AppShell() {
        super(de.danoeh.antennapod.BuildConfig.EXOPACKAGE_FLAGS);
    }
}
```

接下来，开始改造前面步骤三的 BUCK 文件，新建如下 android_library 来编译 ExopackageApplication 类。

```
APP_CLASS_SOURCE = 'src/de/danoeh/antennapod/AppShell.java'

android_library(
    name = 'application-lib',
    srcs = [APP_CLASS_SOURCE],
    deps = [
        # 这是前面创建的 android_build_config() 规则
        ':build-config',

        # 这是前面创建的包裹 buck-android-support.jar 的规则
        ':jars__buck-android-support',
    ],
)
```

如果前面步骤创建的 `android_library` 规则的 `glob()` 有包含 `ExopackageApplication` 类文件，那么需要排除它。

```
srcs = glob([ 'src/de/**/*.*java' ]), # 这是修改前
srcs = glob([ 'src/de/**/*.*java' ], excludes = [APP_CLASS_SOURCE]), # 这是修改后
```

最后修改 `android_binary` 规则，这是变化最大的地方。

```
android_binary(
    name = 'antennapod' ,
    manifest = 'AndroidManifest.xml' ,
    keystore = ':debug_keystore' ,
    use_split_dex = True,
    exopackage_modes = [ 'secondary_dex' ],
    primary_dex_patterns = [
        '^de/danoeh/antennapod/AppShell^' ,
        '^de/danoeh/antennapod/BuildConfig^' ,
        '^com/facebook/buck/android/support/exopackage/' ,
    ],
    deps = [
        ':application-lib' ,
        ':main-lib' ,
    ],
)
```

其中：

- `primary_dex_patterns` 指明 `deps` 中依赖中哪些 `.class` 文件是放到主 `dex` 中。
- `exopackage = ['secondary_dex']` 是为了保证 `BuildConfig.EXOPACKAGE_FLAGS` 被正确设置。
- `use_split_dex = True` 是因为使用 `Exopackage` 要求 `APP` 拆分成多个 `dex` 文件。

最后需要更新 `AndroidManifest.xml` 文件，将其中的 `Application` 名字修改一下，语句如下。

```
android:name=" de.danoeh.antennapod.PodcastApp" # 修改前  
android:name=" de.danoeh.antennapod.AppShell" # 修改后
```

至此，Buck 的改造已经接近尾声，我们可以执行 `buck install --run antennapod` 命令安装我们的例子工程，并享受 Buck 代码的高速构建。

32.5 Buck 的自动化改造

通过前面的学习可以看到，要将现有的工程从 Gradle 改造成 Buck 是存在一定的难度和工作量的，如果能够自动化或者半自动化改造过程，那将是一件大幸事，Piasy¹ 开源了一个名为 OkBuck² 的工具帮助实现了这一点。强烈建议在熟悉 Buck 基本概念和用法的基础上，使用 OkBuck 这个工具来减少工作量，OkBuck 的原理和使用参见 *OkBuck, underneath the hood*³ 和《手把手 OkBuck 教程：应用到 AndroidTDDBootstrap 项目》⁴ 这两篇文章以及工具的 README。

1 <https://github.com/Piasy>

2 <https://github.com/Piasy/OkBuck>

3 <http://blog.piasy.com/OkBuck-Underneath-the-hood/>

4 <http://blog.piasy.com/AndroidTDDBootstrap-Use-OkBuck/>



第5篇 性能优化篇

- ★ 第 33 章 代码优化
- ★ 第 34 章 图片优化
- ★ 第 35 章 电量优化
- ★ 第 36 章 布局优化
- ★ 第 37 章 网络优化

第33章

代码优化

代码级别的优化是最基本的能力，每个开发人员都应该不断努力提高自己的编码能力，从而写出高效的代码。使用本章介绍的方法进行代码优化后，并不能让应用性能得到显著的提升，但不积跬步，无以至千里。根据 Android 官方的建议，编写高效代码的两个基本准则如下。

- 不要做冗余的工作。
- 尽量避免次数过多的内存分配操作。

在这里我要加上第三个准则：深入的理解所用语言特性和系统平台的 API，具体到 Android 开发，就是要熟练掌握 Java 语言，并对 Android SDK 所提供的 API 了如指掌。

33.1 数据结构的选择

正确的选择合适的数据结构是很重要的，对 Java 中常见的数据结构例如 ArrayList 和 LinkedList、HashMap 和 HashSet 等，需要做到对它们的联系与区别有较深入的理解，这样在编写代码中面临选择时才能作出正确的选择，下面我们以在 Android 开发中使用 SparseArray 代替 HashMap 为例进行说明。SparseArray 是 Android 平台特有的稀疏数组的实现，它是 Integer 到 Object 的一个映射，在特定场合可用于代替 HashMap<Integer, <E>>，提高性能。它的核心实现是二分查找算法。

```
// This is Arrays.binarySearch(), but doesn't do any argument validation.
static int binarySearch(int[] array, int size, int value) {
    int lo = 0;
    int hi = size - 1;
```

```

while (lo <= hi) {
    final int mid = (lo + hi) >>> 1;
    final int midVal = array[mid];

    if (midVal < value) {
        lo = mid + 1;
    } else if (midVal > value) {
        hi = mid - 1;
    } else {
        return mid; // value found
    }
}
return ~lo; // value not present
}

```

SparseArray 家族目前有以下四类。

```

// 用于代替HashMap<Integer, Boolean> booleanMap = new HashMap<Integer,
Boolean>();
SparseBooleanArray booleanArray = new SparseBooleanArray();

// 用于代替HashMap<Integer, Integer> booleanMap = new HashMap<Integer,
Integer>();
SparseIntArray intArray = new SparseIntArray();

// 用于代替HashMap<Integer, Long> booleanMap = new HashMap<Integer, Long>();
SparseLongArray longArray = new SparseLongArray();

// 用于代替HashMap<Integer, String> booleanMap = new HashMap<Integer,
String>();
SparseArray<String> stringArray = new SparseArray<String>();

```

需要注意的几点如下。

- SparseArray 不是线程安全的。

- 由于要进行二分查找，因此，SparseArray 会对插入的数据按照 Key 值大小顺序插入。
- SparseArray 对删除操作做了优化，它并不会立即删除这个元素，而是通过设置标识位（DELETED）的方式，后面尝试重用。

在 Android 工程中运行 Lint 进行静态代码分析，会有一个名为 AndroidLintUseSparseArrays 的检查项，如果违规，它会提示。

HashMap can be replaced with SparseArray

这样可以很轻松地找到工程中可优化的地方。

33.2 Handler 和内部类的正确用法

Android 代码中涉及线程间通信的地方经常会使用 Handler，典型的代码结构如下。

```
public class HandlerActivity extends Activity {

    // 可能引入内存泄漏的用法
    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // ...
        }
    };
}
```

使用 Android Lint 分析这段代码，会违反检查项 AndroidLintHandlerLeak，得到如下提示。

This Handler class should be static or leaks might occur.

那么产生内存泄漏的原因可能是什么呢？我们知道，Handler 是和 Looper 以及 MessageQueue 一起工作的，在 Android 中，一个应用启动后，系统默认会创建一个为主线程服务的 Looper 对象，该 Looper 对象用于处理主线程的所有 Message 对象，它的生命周期贯穿于整个应用的生命周期。在主线程中使用的 Handler 都会默认绑定到这个 Looper 对象。在主线程中创建 Handler 对象时，它会立即关联主线程 Looper 对象的 MessageQueue，这时发送到 MessageQueue 中的 Message 对象都会持有这个 Handler 对象的引用，这样在 Looper 处理消息

时才能回调到 Handler 的 handleMessage 方法。因此，如果 Message 还没有被处理完成，那么 Handler 对象也就不会被垃圾回收。

在上面的代码中，将 Handler 的实例声明为 HandlerActivity 类的内部类。而在 Java 语言中，非静态内部匿名类会持有外部类的一个隐式的引用，这样就可能会导致外部类无法被垃圾回收。因此，最终由于 MessageQueue 中的 Message 还没处理完成，就会持有 Handler 对象的引用，而非静态的 Handler 对象会持有外部类 HandlerActivity 的引用，这个 Activity 无法被垃圾回收，从而导致内存泄漏。

一个明显的会引入内存泄漏的例子如下。

```
public class HandlerActivity extends Activity {

    // 可能引入内存泄漏的用法
    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // ...
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // 延迟5分钟发送消息
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() { /* ... */ }
        }, 1000 * 60 * 5);
    }
}
```

由于消息延迟 5 分钟发送，因此，当用户进入这个 Activity 并退出后，在消息发送并处理

完成之前，这个 Activity 是会被系统回收的（系统内存确实不够使用的情况例外）。

如何解决呢？有两个方案。

- 在子线程中使用 Handler，这时需要开发者自己创建一个 Looper 对象，这个 Looper 对象的生命周期同一般的 Java 对象，因此这种用法没有问题。
- 将 Handler 声明为静态的内部类，前面说过，静态内部类不会持有外部类的引用，因此，也不会引起内存泄漏，经典用法的代码如下。

```
public class HandlerActivity extends Activity {

    /**
     * 声明一个静态的Handler内部类，并持有外部类的弱引用
     */
    private static class InnerHandler extends Handler {

        private final WeakReference<HandlerActivity> mActivity;

        public InnerHandler(HandlerActivity activity) {
            mActivity = new WeakReference<HandlerActivity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            HandlerActivity activity = mActivity.get();
            if (activity != null) {
                // ...
            }
        }
    }

    private final InnerHandler mHandler = new InnerHandler(this);

    /**
     * 静态的匿名内部类不会持有外部类的引用
```



```

*/
private static final Runnable sRunnable = new Runnable() {
    @Override
    public void run() { /* ... */ }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // 延迟5分钟发送消息
    mHandler.postDelayed(sRunnable, 1000 * 60 * 5);
}
}

```

33.3 正确地使用 Context

Context 应该是每个入门 Android 开发的程序员第一个接触到的概念，它代表当前的上下文环境，可以用来实现很多功能的调用，语句如下。

```

// 获取资源管理器对象，进而可以访问到例如 string, color 等资源
Resources resources = context.getResources();

// 启动指定的 Activity
context.startActivity(new Intent(this, MainActivity.class));

// 获取各种系统服务
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.
TELEPHONY_SERVICE);

// 获取系统文件目录
File internalDir = context.getCacheDir();
File externalDir = context.getExternalCacheDir();

// 更多.....

```

可见，正确理解 Context 的概念是很重要的，虽然应用开发中随处可见 Context 的使用，但并不是所有的 Context 实例都具备相同的功能，在使用上需要区别对待，否则很可能会引入问题。我们首先来总结下 Context 的种类。

33.3.1 Context 的种类

根据 Context 依托的组件以及用途不同，我们可以将 Context 分为如下几种。

- **Application**: Android 应用中的默认单例类，在 Activity 或者 Service 中通过 `getApplication()` 可以获取到这个单例，通过 `context.getApplicationContext()` 可以获取到应用全局唯一的 Context 实例。
- **Activity/Service**: 这两个类都是 `ContextWrapper` 的子类，在这两个类中可以通过 `getBaseContext()` 获取到它们的 Context 实例，不同的 Activity 或者 Service 实例，它们的 Context 都是独立的，不会复用。
- **BroadcastReceiver**: 和 Activity 以及 Service 不同，`BroadcastReceiver` 本身并不是 Context 的子类，而是在回调函数 `onReceive()` 中由 Android 框架传入一个 Context 的实例。系统传入的这个 Context 实例是经过功能裁剪的，它不能调用 `registerReceiver()` 以及 `bindService()` 这两个函数。
- **ContentProvider**: 同样的，`ContentProvider` 也不是 Context 的子类，但在创建时系统会传入一个 Context 实例，这样在 `ContentProvider` 中可以通过调用 `getContext()` 函数获取。如果 `ContentProvider` 和调用者处于相同的应用进程中，那么 `getContext()` 将返回应用全局唯一的 Context 实例。如果是其他进程调用的 `ContentProvider`，那么 `ContentProvider` 将持有自身所在进程的 Context 实例。

33.3.2 错误使用 Context 导致的内存泄漏

错误地使用 Context 可能会导致内存泄漏，典型的例子是在实现单例模式时使用 Context，如下代码是可能会导致内存泄漏的单例实现。

```
public class SingleInstance {  
  
    private Context mContext;
```

```

private static SingleInstance sInstance;

private SingleInstance(Context context) {
    mContext = context;
}

public static SingleInstance getInstance(Context context) {
    if (sInstance == null) {
        sInstance = new SingleInstance(context);
    }

    return sInstance;
}
}

```

如果使用者调用 `getInstance` 时传入的 `Context` 是一个 `Activity` 或者 `Service` 的实例，那么在应用退出之前，由于单例一直存在，会导致对应的 `Activity` 或者 `Service` 被单例引用，从而不会被垃圾回收，`Activity` 或者 `Service` 中关联的其他 `View` 或者数据结构对象也不会被释放，从而导致内存泄漏。正确的做法是使用 `Application Context`，因为它是应用唯一的，而且生命周期是跟应用一致的，正确的单例实现如下。

```

public class SingleInstance {

    private Context mContext;

    private static SingleInstance sInstance;

    private SingleInstance(Context context) {
        mContext = context;
    }

    public static SingleInstance getInstance(Context context) {
        if (sInstance == null) {

```

```
        sInstance = new SingleInstance(context.getApplicationContext());
// 这一句是关键
    }

    return sInstance;
}
}
```

33.3.3 不同 Context 的对比

不同组件中的 Context 能提供的功能不尽相同，总结起来，如表 33-1 所示¹。

表33-1

功 能	Application	Activity	Service	BroadcastReceiver	ContentProvider
显示 Dialog	NO	YES	NO	NO	NO
启动 Activity	NO[1]	YES	NO[1]	NO[1]	NO[1]
实现 Layout Inflation	NO[2]	YES	NO[2]	NO[2]	NO[2]
启动 Service	YES	YES	YES	YES	YES
绑定 Service	YES	YES	YES	YES	NO
发送 Broadcast	YES	YES	YES	YES	YES
注册 Broadcast	YES	YES	YES	YES	NO[3]
加载资源 Resource	YES	YES	YES	YES	YES

其中 NO[1] 标记表示对应的组件并不是真的不可以启动 Activity，而是建议不要这么做，因为这些组件会在新的 Task 中创建 Activity，而不是在原来的 Task 中。

NO[2] 标记也是表示不建议这么做，因为在非 Activity 中进行 Layout Inflation，会使用系统默认的主题，而不是应用中设置的主题。

NO[3] 标记表示在 Android 4.2 及以上的系统上，如果注册的 BroadcastReceiver 是 null 时是可以的，用来获取 sticky 广播的当前值。

33.4 掌握 Java 的四种引用方式

掌握 Java 的四种引用类型对于写出内存使用良好的应用是很关键的，同时它也是技术面

¹ <https://possiblemobile.com/2013/06/context/>

试中经常会出现的题目，属于语言基础。

- 强引用：Java 里面最广泛使用的一种，也是对象默认的引用类型。如果一个对象具有强引用，那么垃圾回收器是不会对它进行回收操作的，当内存空间不足时，Java 虚拟机将会抛出 `OutOfMemoryError` 错误，这时应用将会终止运行。
- 软引用：一个对象如果只有软引用，那么当内存空间充足时，垃圾回收器不会对它进行回收操作，只有当内存空间不足时，这个对象才会被回收。软引用可以用来实现内存敏感的高速缓存，如果配合引用队列（`ReferenceQueue`）使用，当软引用指向的对象被垃圾回收器回收后，Java 虚拟机将会把这个软引用加入到与之关联的引用队列中。
- 弱引用：在面试过程中，很多面试者经常会将弱引用和软引用这两者搞混，其实从名字上可以看出，弱引用是比软引用更弱的一种引用类型，只有弱引用指向的对象的生命周期更短，当垃圾回收器扫描到只具有弱引用的对象时，不论当前内存空间是否不足，都会对弱引用对象进行回收。弱引用也可以和一个引用队列配合使用，当弱引用指向的对象被回收后，Java 虚拟机会将这个弱引用加入到与之关联的引用队列中。
- 虚引用：和软引用和弱引用不同，虚引用并不会对所指向的对象生命周期产生任何影响，也就是对象还是会按照它原来的方式被垃圾回收器回收，虚引用本质上只是一个标记作用，主要用来跟踪对象被垃圾回收的活动，虚引用必须和引用队列配合使用，当对象被垃圾回收时，如果存在虚引用，那么 Java 虚拟机会将这个虚引用加入到与之关联的引用队列中。

或许大多数人对于将软引用、弱引用、虚引用加入到引用队列之后有什么作用很疑惑，面试过程中也发现很多人知道四种引用的定义，但对于引用队列的作用却一无所知，引用队列的应用例子可以参见本书第 44 章。

33.5 其他代码微优化

33.5.1 避免创建非必要的对象

对象的创建需要内存分配，对象的销毁需要垃圾回收，这些都会一定程度上影响到应用的性能。因此一般来说，最好是重用对象而不是在每次需要的时候去创建一个功能相同的新对象，特别是注意不要在循环中重复创建相同的对象。

33.5.2 对常量使用 static final 修饰

对于基本数据类型和 String 类型的常量,建议使用 static final 修饰,因为 final 类型的常量会在进入静态 dex 文件的域初始化部分,这时对基本数据类型和 String 类型常量的调用不会涉及类的初始化,而是直接调用字面量。

33.5.3 避免内部的 Getters/Setters

在面向对象编程中,Getters/Setters 的作用主要是对外屏蔽具体的变量定义,从而达到更好的封装性。但如果在类内部还使用 Getters/Setters 函数访问变量的话,会降低访问的速度。根据 Android 官方文档,在没有 JIT (Just In Time) 编译器时,直接访问变量的速度是调用 Getter 方法的 3 倍;在 JIT 编译时,直接访问变量的速度是调用 Getter 方法的 7 倍。当然,如果你的应用中使用了 ProGuard 的话,那么 ProGuard 会对 Getters/Setters 进行内联操作,从而达到直接访问的效果。

33.5.4 代码的重构

代码的重构是一项长期的持之以恒的工作,需要依靠团队中每一个成员来维护代码库的高质量,如何有效的进行代码重构,除了需要对你所在项目有较深入的理解之外,你还需要一定的方法论指导,强烈推荐《重构 - 改善既有代码的设计》¹和《代码整洁之道》²这两本书。

¹ https://www.amazon.cn/%E9%87%8D%E6%9E%84-%E6%94%B9%E5%96%84%E6%97%A2%E6%9C%89%E4%BB%A3%E7%A0%81%E7%9A%84%E8%AE%BE%E8%AE%A1-%E9%A9%AC%E4%B8%81%C2%B7%E7%A6%8F%E5%8B%92/dp/B011LPUB42/ref=sr_l_1?ie=UTF8&qid=1462089891&sr=8-1&keywords=%E9%87%8D%E6%9E%84-%E6%94%B9%E5%96%84%E6%97%A2%E6%9C%89%E4%BB%A3%E7%A0%81%E7%9A%84%E8%AE%BE%E8%AE%A1

² https://www.amazon.cn/%E4%BB%A3%E7%A0%81%E6%95%B4%E6%B4%81%E4%B9%8B%E9%81%93-%E9%A9%AC%E4%B8%81/dp/B0031M9GHC/ref=sr_l_1?ie=UTF8&qid=1462089927&sr=8-1&keywords=%E7%AE%80%E6%B4%81%E4%BB%A3%E7%A0%81

第34章

图片优化

图片的加载和显示是每个商业 APP 都避免不了的问题，对于图片重度依赖类 APP，例如壁纸类应用，图片社交类应用，对于图片的处理将会影响到整个 APP 的用户体验。

在正式了解 Android 中如何优化图片相关的内容之前，我们先来聊聊 Android 系统支持的图片格式。

34.1 图片的格式

目前移动端 Android 平台原生支持的图片格式主要有:JPEG、PNG、GIF、BMP 和 WebP (自从 Android 4.0 开始支持)，但是在 Android 应用开发中能够使用的编解码格式只有其中的三种：JPEG、PNG、WebP，图片格式可以通过查看 Bitmap 类的 CompressFormat 枚举值来确定。

```
public static enum CompressFormat {  
    JPEG,  
    PNG,  
    WEBP;  
  
    private CompressFormat() {  
  
    }  
}
```

如果要在应用层使用 GIF 格式图片，那么需要自己引入第三方函数库进行支持。

34.1.1 JPEG¹

JPEG（发音为 /jay-peg/）是一种广泛使用的有损压缩图像标准格式，它不支持透明和多帧动画，一般摄影类作品最终都是以 JPEG 格式展示。通过控制压缩比，可以调整图片的大小。

34.1.2 PNG²

PNG 是一种无损压缩图片格式，它支持完整的透明通道，从图像处理领域讲，JPEG 只有 RGB 三个通道，而 PNG 有 ARGB 四个通道。由于是无损压缩，因此 PNG 图片占用空间一般比较大，会无形中增加最终 APP 的大小，在做 APP 瘦身时一般都要对 PNG 图片进行处理以减小其占用的体积。

34.1.3 GIF

GIF 是一种古老的图片格式，它诞生于 1987 年，随着初代互联网流行开来。它的特点是支持多帧动画。大家对这种格式肯定不陌生，社交平台上面发送的各种动态表情，大部分都是基于 GIF 来实现的。

34.1.4 WebP

相比前面几种图片格式，WebP（发音为 /weppy/）算是一个初生儿了，由 Google 在 2010 年发布，它支持有损和无损压缩、支持完整的透明通道、也支持多帧动画，是一种比较理想的图片格式。目前国内很多主流 APP 都已经应用了 WebP，例如微信、微博、淘宝等。在既保证图片质量又要限制图片大小的需求下，WebP 应该是首选。

34.2 图片的压缩

目前无论 Android 平台还是 iOS 平台，大多数 APP 在搭建界面时使用的几乎都是 PNG 格式图片资源，除非你的项目已经全面支持 WebP 格式，否则你都会面临对 PNG 图片瘦身的要求，在这里，我们可以通过几个工具对 PNG 图片进行压缩来达到瘦身的目的。

1 <https://jpeg.org/>

2 <http://www.libpng.org/pub/png/>

34.2.1 无损压缩 ImageOptim¹

ImageOptim 是一个无损的压缩工具，它通过优化 PNG 压缩参数，移除冗余元数据以及非必需的颜色配置文件等方式，在不牺牲图片质量的前提下，既减小了 PNG 图片占用的空间，又提高了加载的速度。

34.2.2 有损压缩 ImageAlpha²

ImageAlpha 是 ImageOptim 作者开发的一个有损的 PNG 压缩工具，相比较而言，图片大小得到极大的降低，当然图片质量同时也会受到一定程度的影响，经过该工具压缩的图片，需要经过设计师的检视才能最终上线，否则可能会影响到整个 APP 的视觉效果。

34.2.3 有损压缩 TinyPNG³

TinyPNG 也是比较知名的有损 PNG 压缩工具，它以 Web 站点的形式提供，没有独立的 APP 安装包，同所有的有损压缩工具一样，经过压缩的图片，需要经过设计师的检视才能最终上线，否则可能会影响到整个 APP 的视觉效果。

其实还有很多无损压缩工具，例如 JPEGMini⁴、MozJPEG⁵ 等，大家可以从中选择适合自己项目的一个就行，主要是在图片大小和图片质量之间找到一个折衷点。

34.2.4 PNG/JPEG 转换为 WebP

如果你的 APP 最低支持到 Android 4.0，那么可以直接使用系统提供的能力来支持 WebP，如果是 4.0 以下的系统，也可以通过在 APP 中集成第三方函数库例如 webp-android-backport⁶ 来实现对 WebP 的支持。根据 Google 的测试，无损压缩后的 WebP 比 PNG 文件少了 45% 的文件大小，即使这些 PNG 文件经过其他压缩工具例如 ImageOptim 压缩之后，WebP 依然可以减少约 28% 的文件大小。

1 <https://imageoptim.com/>

2 <https://pngmini.com/>

3 <https://tinypng.com/>

4 <http://www.jpegmini.com/>

5 <https://imageoptim.com/mozjpeg/>

6 <https://github.com/alexey-pelykh/webp-android-backport>

WebP 转换的工具可以选择智图¹和 iSparta²等。

34.2.5 尽量使用 NinePatch 格式的 PNG 图

.9.png 图片格式简称 NinePatch 图，本质上仍然是 PNG 格式图片，它是针对 Android 平台的一种特殊 PNG 图片格式，可以在图片指定位置拉伸或者填充内容。NinePatch 图的优点是体积小，拉伸不变形，能够很好的适配 Android 各种机型。Android SDK 自带了 NinePatch 图的编辑工具，位于 `sdk/tools/draw9patch`，点击即可启动；当然，Android Studio 也集成了 PNG 转 NinePatch 的功能，我们只需右键点击某个需要转换的 PNG 图片，在弹出的对话框中选择 `Create 9-Patch File...` 即可自动完成转换。

34.3 图片的缓存

图片的缓存可以参见本书第 20 章中关于图片的显示和缓存的内容，为了达到应用中图片缓存的最优，我们需要根据具体的业务需求，来选择对应的开源框架。

¹ <http://zhitu.isux.us/>

² <http://isparta.github.io/>

第35章

电量优化

手机的耗电量历来是衡量一款 APP 的重要指标之一，特别对于地图、导航、运动类 APP，耗电量有可能成为用户选择你家的产品还是竞品的一个关键因素。

Android 应用开发中存在很多比较耗电的特性，例如网络、定位、传感器等，同时对于一些关键 API 的正确使用也是有效降低应用耗电量的手段，例如 BroadcastReceiver、AlarmManager、WakeLock 等。

35.1 BroadcastReceiver

为了减少应用损耗的电量，我们在代码实现中需要尽量避免无用操作代码的执行。例如，当应用退到后台，一切的界面刷新都是没有意义而且浪费内存和电量的，广播接收器是一个典型的例子。如果应用中存在一个监听网络状态变化的广播接收器并会执行一些动作，例如弹出 Toast 提示用户网络环境的切换，那么当应用位于后台时，我们需要禁用掉这个提示功能，因为这时它不仅影响用户使用其他应用而且还在后台默默地消耗着设备上本来就不多的电量，通常的做法是在界面 onPause 之后取消广播监听器的监听操作，同时根据具体业务需求选择当应用位于后台时是否禁用广播接收器，代码如下。

```
private void enableBroadcastReceiver(boolean isEnabled, Class<?> receiver) {
    PackageManager pm = getPackageManager();
    ComponentName receiverName = new ComponentName(this, receiver);

    int newState;
    if (isEnabled) {
```

```
        newState = PackageManager.COMPONENT_ENABLED_STATE_ENABLED;
    } else {
        newState = PackageManager.COMPONENT_ENABLED_STATE_DISABLED;
    }

    pm.setComponentEnabledSetting(receiverName, newState, PackageManager.
DONT_KILL_APP);
}
```

35.2 数据传输

Android 中的数据传输方式有很多种，常见的有。

- 蓝牙传输。
- Wi-Fi 传输。
- 移动网络传输。

无论哪一种传输方式，为了更好地延长电池的使用时间，我们在使用过程中都需要重点关注两件事情。

- 后台数据传输的管理：根据具体业务需求，严格限制应用位于后台时是否禁用某些数据传输，尽量能够避免无效的数据传输。
- 数据传输的频度问题：通过经验值或者数据统计的方法确定好数据传输的频度，避免冗余重复的数据传输，同时参考本书第37章的内容，数据传输过程中要压缩数据大小，合并网络请求，避免轮询等。

35.3 位置服务

位置服务已经成为很多应用必备的功能之一，特别是地图类、电商类和社交类应用，位置服务更是关键的功能之一，能否正确有限地使用位置服务，是应用耗电量的一个关键点。Android 中常见的位置服务有两种：GPS 定位和网络定位。GPS 定位服务需要 `ACCESS_FINE_LOCATION` 权限，网络定位服务需要 `ACCESS_COARSE_LOCATION` 或者 `ACCESS_FINE_`

LOCATION 权限。代码中使用位置服务时，通常需要关注以下几个方面。

- 有没有及时注销位置监听器：和使用广播监听器一样，位置监听器也需要做到及时的注销，因为长时间的监听位置更新会耗费大量的电量，通常可以选择在页面的 onPause 中进行注销操作，更好用且全局有效的做法是禁用位置监听器，代码如下。

```
private void disableLocaltionListener(LocationListener listener) {
    LocationManager locationManager = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_
FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this, Manifest.permission.
ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        return;
    }
    locationManager.removeUpdates(listener);
}
```

- 位置更新监听频率的设定：根据具体的业务需求设置一个合适的更新频率值，通常需要在定位精度和耗电量之间综合考虑。Android SDK 提供了 requestLocationUpdates 方法进行设置，这个方法有两个主要的参数：
 - ❑ minTime：用来指定位置更新通知的最小时间间隔，单位是毫秒。
 - ❑ minDistance：用来指定位置更新通知的最小距离，单位是米。
- 多种位置服务的选择：Android 系统为开发者提供了三种定位服务：
 - ❑ GPS 定位：通过接收全球定位系统的卫星提供的经纬度坐标信息实现位置服务，精度是最高的，通常在10米以内。当然 GPS 定位在时间和电量的消耗上也是最高的。
 - ❑ 网络定位：通过移动通信的基站信号差异来计算出手机所在的位置，精度比 GPS 定位差很多，通常在几百米范围内。
 - ❑ 被动定位：最省电的定位服务，如果应用使用被动定位服务，说明它想知道位置更新信息但又不想主动获取，也就是这个应用会等待手机中其他应用、服务或者系统组件发出定位请求，并和这些组件的监听器一起接收位置更新。

实际应用中需要综合考虑应用的具体需求在不同时机采用不同的定位服务，实际开发中很

多应用通常会选择第三方的定位 SDK，例如百度定位、高德定位，因为这些 SDK 无论在定位时间，定位精度还是定位耗电量方面都作了专门的优化。

35.4 AlarmManager

AlarmManager 是 Android SDK 提供的一个唤醒 API，它是系统级别的服务，可以在特定的时刻广播一个指定的 Intent，这个 PendingIntent 可以用来启动 Activity、Service 或 BroadcastReceiver。例如后台上传统计信息，可以通过一个 AlarmManager 来定时检查是否满足条件并上传记录。AlarmManager 提供了三个常用的方法。

- set：设置一次性的闹钟操作。
- setRepeating：设置重复性的闹钟操作。
- setInexactRepeating：也是设置重复性的闹钟操作，只不过两个相连的闹钟执行的间隔时间不是固定的。

AlarmManager 的唤醒操作也是比较耗电的，通常情况下需要保证两次唤醒操作的时间间隔不要太短，在不需要使用唤醒功能的情况下尽早取消 AlarmManager，否则应用会一直处于耗电状态。

35.5 WakeLock

WakeLock 是为了保持设备处于唤醒状态的 API，因为在某些情况下，即使用户长时间不与设备交互，仍然需要阻止设备进入休眠状态，从而保证良好的用户体验，例如用户使用手机观看电影的时候，需要保证屏幕保持开启状态。WakeLock 的锁类型又很多种，不同的锁类型对 CPU、屏幕和键盘的影响不相同，具体情况如下。

- PARTIAL_WAKE_LOCK：保持 CPU 正常运转，但屏幕和键盘灯可能是关闭的。
- SCREEN_DIM_WAKE_LOCK：保持 CPU 正常运转，允许屏幕点亮但可能是置灰的，键盘灯可能是关闭的。
- SCREEN_BRIGHT_WAKE_LOCK：保持 CPU 正常运转，允许屏幕高亮显示，键盘灯可能是关闭的。

- **FULL_WAKE_LOCK**: 保持 CPU 正常运转, 保持屏幕高亮显示, 键盘灯也保持亮度。
- **ACQUIRE_CAUSES_WAKEUP**: 强制屏幕和键盘灯亮起, 这种锁针对一些必须通知用户的操作。
- **ON_AFTER_RELEASE**: 当 WakeLock 被释放后, 继续保持屏幕和键盘灯开启一定时间。

WakeLock 的使用很简单, 语句如下。

```

PowerManager.WakeLock wakeLock = null;

// 获取唤醒锁
private void acquireWakeLock() {
    if (null == wakeLock) {
        PowerManager pm = (PowerManager) this.getSystemService(Context.
POWER_SERVICE);
        wakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK |
PowerManager.ON_AFTER_RELEASE, "PlayService");
        if (null != wakeLock) {
            wakeLock.acquire();
        }
    }
}

// 释放唤醒锁
private void releaseWakeLock() {
    if (null != wakeLock) {
        wakeLock.release();
        wakeLock = null;
    }
}

```

使用 WakeLock 时, 需要切记及时释放锁, 有的应用在使用完 WakeLock 之后会忘记释放它, 从而导致屏幕一直显示很长时间, 快速的耗费了手机的电量。通常情况下, 我们要尽早地释放 WakeLock, 例如在播放视频时获取 WakeLock 保持屏幕常亮, 在暂停播放时就应该及时地释放锁, 而不是等到停止播放才释放, 在应用恢复播放时再次获取 WakeLock 即可。

第36章

布局优化

在进行 Android 应用的界面编写时，如果创建的布局层次结构比较复杂，View 树嵌套的层次比较深，那么将会使得页面展现的时间比较长，导致应用运行起来越来越慢。Android 布局的优化是实现应用响应灵敏的基础。遵循一些通用的编码准则有利于实现这个目标。

36.1 include 标签共享布局

在使用 XML 文件编写 Android 应用的界面布局时，经常会遇到在不同的页面中需要实现相同的布局，这时候就会写出重复的代码。例如由于几乎每个页面都有标题栏，因此，在每个页面中都要实现标题栏的 XML 代码。这种方式显然是不建议的，最佳实践是将通用的布局抽取出来，独立成一个 XML 文件，然后在需要用到的页面中使用 include 标签引入进来，不仅减少了代码量，而且在需要修改标题栏的时候，只需修改这个独立的文件，而不是到每个页面中进行重复的修改劳动。假设我们抽取出来的标题栏布局名为 layout_common_title.xml，语句如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <!-- 此处只是为了演示，实际项目代码要比这个复杂一些 -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```



```

        android:text=" 标题栏"
        android:layout_centerInParent=" true" />

```

```

</RelativeLayout>

```

在页面 MainActivity 的布局中，使用到这个标题栏，引入的方式如下。

```

<?xml version=" 1.0" encoding=" utf-8" ?>
<RelativeLayout xmlns:android=" http://schemas.android.com/apk/res/android"
    xmlns:tools=" http://schemas.android.com/tools"
    android:layout_width=" match_parent"
    android:layout_height=" match_parent"
    android:paddingBottom=" @dimen/activity_vertical_margin"
    android:paddingLeft=" @dimen/activity_horizontal_margin"
    android:paddingRight=" @dimen/activity_horizontal_margin"
    android:paddingTop=" @dimen/activity_vertical_margin"
    tools:context=" com.ascel885.md5test.MainActivity" >

    <!-- 引入外部共享布局 layout_common_title -->
    <include layout=" @layout/layout_common_title"
        android:layout_height=" 56dp"
        android:layout_alignParentTop=" true"
        android:layout_width=" match_parent" />

    <TextView
        android:layout_width=" wrap_content"
        android:layout_height=" wrap_content"
        android:text=" Hello World!" />

</RelativeLayout>

```

36.2 ViewStub 标签实现延迟加载

ViewStub 是一种不可视并且大小为 0 的视图，它可以延迟到运行时才填充（inflate）布局资源。当 ViewStub 设置为可见或者被 inflate 之后，就会填充布局资源，之后这个 ViewStub 就

会被填充的视图所代替，跟普通的视图不再有任何区别。ViewStub 的使用场景在于某个页面需要根据用户交互或者其他条件动态调整显示的视图，例如某个页面是从服务端获取数据进行展现的，正常情况下会展示获取到的数据，但在网络不可用的情况下，需要展示一张不可用的图片和相关提示信息，大多数情况下，这个不可用的视图是不需要显示出来的，如果不使用 ViewStub，只是把不可用视图隐藏，那么它还是会被 inflate 并占用资源，如果使用 ViewStub，则可以在需要显示的时候才会进行视图的填充，从而实现延迟加载的目的。假设页面加载数据失败时会展示一张网络出错的图片，这个图片的布局如下所示，它就是我们的 ViewStub 会显示的视图。

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <!-- 此处只是为了演示，实际项目代码要增加其他提示信息 -->
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:src="@drawable/network_error" />

</RelativeLayout>
```

页面使用 ViewStub 加载这个布局的代码如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.ascel885.md5test.MainActivity" >
```

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />

<ViewStub
    android:id="@+id/error_stub"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout="@layout/layout_network_error"
    android:inflatedId="@+id/error_view" />

</RelativeLayout>

```

其中的 `android:inflatedId` 的值是在 Java 代码中调用 `ViewStub` 的 `inflate()` 或者 `setVisibility()` 方法时返回的 ID，这个 ID 就是被填充的 `View` 的 ID。

36.3 merge 标签减少布局层次

`merge` 标签在某些场景下可以用来减少布局的层次，由于 `Activity` 的 `ContentView` 的最外层容器是一个 `FrameLayout`，因此，当一个独立的布局文件最外层是 `FrameLayout`，且这个布局不需要设置背景 `android:background` 或者 `android:padding` 等属性时，可以使用 `<merge>` 标签来代替 `<FrameLayout>` 标签，从而减少一层多余的 `FrameLayout` 布局。另外一种可以使用 `<merge>` 标签的情况是当前布局作为另外一个布局的子布局，使用 `<include>` 标签引入时，我们使用前面 `include` 一节的例子，可以考虑把 `layout_common_title.xml` 文件中的 `RelativeLayout` 替换成 `merge`，代码如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<merge xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

```

```
<!-- 此处只是为了演示，实际项目代码要比这个复杂一些 -->
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="标题栏"
    android:layout_centerInParent="true" />

</merge>
```

36.4 尽量使用 CompoundDrawable

在 LinearLayout 布局中，如果存在相邻的 ImageView 和 TextView，语句如下。

```
<?xml version="1.0" encoding="utf-8" ?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:orientation="vertical"
    android:paddingTop="@dimen/activity_vertical_margin" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <ImageView
        android:id="@+id/image_view"
        android:layout_marginTop="10dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```

        android:src="@mipmap/ic_launcher" />
</LinearLayout>

```

那么一般来说可以使用 compound drawable 合二为一成为一个 TextView, ImageView 中的图片变成 TextView 的如下属性之一: drawableTop, drawableLeft, drawableRight 或者 drawableBottom。原来两个 View 之间的间隔使用 TextView 的属性 drawablePadding 来代替, 语句如下。

```

<?xml version="1.0" encoding="utf-8" ?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:orientation="vertical"
    android:paddingTop="@dimen/activity_vertical_margin" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="wrap_content"
        android:drawablePadding="10dp"
        android:drawableBottom="@mipmap/ic_launcher"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>

```

36.5 使用 Lint

Android Lint 除了可以对 Java 代码进行静态检查之外, 也可以用来为检查应用的布局是否存在可优化的地方。Lint 的如下规则是专门为优化布局设置的:

- AndroidLintUseCompoundDrawables: 就是前面介绍的尽量使用 CompoundDrawable。
- MergeRootFrame: 就是前面介绍的merge 标签减少布局层次。

- **TooManyViews**: 单个布局中存在太多的 View, 默认情况下, 单个布局中 View 的个数最多只能是 80 个, 可以考虑使用 CompoundDrawables 等来减少 View 的个数。
- **TooDeepLayout**: 避免过深的布局嵌套, 默认情况下, 单个布局中最多层级是10, 可以考虑使用 RelativeLayout 来减少布局的层次。
- **UselessParent**: 当一个布局满足以下条件时, 可以将它移除掉, 并将它的子 View 移动到它的父容器中, 以得到更扁平的布局层次: (1) 这个布局中只有一个子 View, 也就是子 View 不存在兄弟 View; (2) 这个布局不是一个 ScrollView 或者根布局; (3) 这个布局没有设置背景 android:background。
- **NestedWeights**: android:layout_weight 属性会使得 View 控件被测量两次, 当一个 LinearLayout 拥有非 0dp 值的 android:layout_weight 属性, 这时如果将它嵌套在另一个拥有非 0dp 的android:layout_weight 的 LinearLayout, 那么这时测量的次数将呈指数级别增加。
- **UselessLeaf**: 一个布局如果既没有子 View 也没有设置背景, 那么它将是不可见的, 通常可以移除它, 这样可以得到更扁平和高效的布局层级。
- **InefficientWeight**: 当 LinearLayout 中只有一个子 View 定义了 android:layout_weight 属性, 更高性能的做法是使用 0dp 的 android:layout_height 或者 android:layout_width 来替换它, 这样这个子 View 就不需要测量它自身对应的大小。

第37章

网络优化

移动端 APP 几乎都是联网的，通过网络请求从服务端获取数据，网络的延迟等会对 APP 的性能产生较大的影响，如何优化网络是本章的重点。

网络优化除了可以节省网络流量、节省电量，还可以提高应用的响应。在 Android 平台上，可以采取以下措施来改善应用的网络请求。

37.1 避免 DNS 解析

DNS 是域名系统，它的主要功能是根据应用请求所用的域名 URL 去网络上面映射表中查找对应 IP 地址，这个过程可能会需要上百毫秒的时间，而且可能会存在 DNS 劫持的危险，根据具体的业务需求，我们可以使用 IP 直连的方式来代替域名访问的方式，从而达到更快的网络请求。使用 IP 的坏处是不够灵活，当对应的服务因为某些原因 IP 地址发生变化后，客户端就访问不了了。因此 IP 方式需要增加动态更新的能力，或者在 IP 方式访问失败时，切换到域名访问方式。

37.2 合并网络请求

一次完整的 HTTP 请求，首先需要进行 DNS 查找，接着通过 TCP 三次握手，从而建立连接；如果是 HTTPS 请求，那么还需要经过 TLS 握手成功后连接才建立。因此对于网络请求应该尽量减少请求的接口，能够合并的网络请求就尽量合并。

37.3 预先获取数据

预先获取数据能够将网络请求集中在一次，这样其他时间段手机就可以切换到空闲状态，从而避免经常性的唤醒和空闲，起到节省电量的作用。

37.4 避免轮询

轮询是指客户端每隔一段时间就向服务端主动发起的网络请求，如果存在需要的数据就拉取，没有的话就继续等待下一次轮询。一般情况下不建议在应用中使用轮询操作，能够使用推送来替换的尽量使用推送。在不得已的情况下，也要避免使用 `Thread.sleep()` 函数来循环等待，比较好的做法是使用系统 `AlarmManager` 来实现定时轮询，`AlarmManager` 可以保证在系统休眠时 CPU 也可以得到休眠，在下次需要发起网络请求的时候才唤醒。

37.5 优化重连机制

尽量避免在网络请求失败时，无限制的循环重试连接，可以设定一个最大重连次数，超过次数限制之后结束重连，等一段较长时间后再尝试连接，或者把是否重连的问题抛给使用者根据具体的业务需求确定。

37.6 离线缓存

对于类似图片，文件等数据，可以使用内存缓存+外存缓存的方式实现二级缓存策略，当在缓存中命中对应的图片或者文件时，直接从缓存中读取，无需走网络请求，不仅避免了网络延迟，还节省了用户的流量。在 Android 中，典型的是使用 `LruCache` 实现内存缓存，`DiskLruCache` 实现外存缓存。对于图片，已经有很多开源框架可供选择，参见本书第 20 章的内容。

37.7 压缩数据大小

从节省网络流量和提高应用响应度等方面出发，我们需要减少网络上传输的数据，对于

客户端来说，可以对发送给服务端的数据进行 gzip 压缩；同时可以选用更优的数据传输格式，例如可以使用二进制方式代替 JSON 格式，使用 WebP 图片格式代替 JPEG 或者 PNG 图片格式等。关于数据格式的选择可以参见本书第 14 章的内容。

37.8 不同的网络环境使用不同的超时策略

应用中应该根据当前的网络类型来设置不同的网络超时时间，常见的网络类型有 2G、3G、4G 和 Wi-Fi。为了实时更新当前网络类型，可以通过监听 `ConnectivityManager.CONNECTIVITY_ACTION` 的变化来获取最新的网络类型，并动态调整网络超时时间。

37.9 CDN 的使用

CDN 全称是内容分发网络，它的基本思想是尽可能避开网络上可能影响数据传输速度和稳定性的环节，从而实现更快、更稳定的数据传输。CDN 加速能够缓解电信核心网络延迟带来的影响。



第6篇 移动安全篇

- ★ 第 38 章 Android 混淆机制详解
- ★ 第 39 章 Android 反编译机制详解
- ★ 第 40 章 客户端敏感信息隐藏技术研究
- ★ 第 41 章 Android 加固技术研究
- ★ 第 42 章 Android 安全编码

第38章

Android混淆机制详解

混淆是增加逆向工程和破解的难度，防止 APP 知识产权被窃取的一个有力手段，高级的代码混淆甚至可以有效地保护存储在客户端的密钥，同时混淆也有很多需要注意的地方。

从广义上讲，Android 中的混淆包括三种类型：Java 代码的混淆、Native（C & C++）代码的混淆以及资源文件的混淆，下面分别进行介绍。

38.1 Java 代码的混淆

Java 代码的混淆在 Android 中是最为常见的一种混淆方式，一般依赖于 Proguard 或者 DexGuard 工具，其中 Proguard 是免费且开源的，DexGuard 是付费的，它们的特性对比如图 38-1 所示。

	ProGuard	DexGuard	DexGuard Enterprise
Availability	Open source	Individual developers	Larger companies
Size improvements	✓	✓	✓
Performance improvements	✓	✓	✓
Name obfuscation	✓	✓	✓
Code protection		✓	✓
Resource protection			✓
Application integrity			✓
Platform integrity			✓
Communication hardening			✓

图38-1

可以看到，付费工具功能还是强大许多，但一般情况下，使用 Proguard 就够了，使用 Proguard 混淆过的代码如图 38-2 所示。

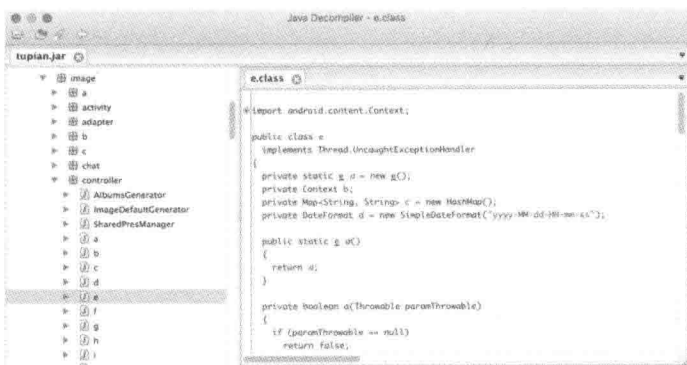


图38-2

下面主要对 Proguard 进行简单的介绍。

38.1.1 Proguard 的特性

首先我们需要知道的是,在 Android 中使用 Proguard,它不仅为我们提供代码混淆的功能,事实上 Proguard 主要提供了 4 个功能特性。

- 压缩: Java 源代码通常被编译为字节码,虽然字节码比源代码更简洁,但它本身仍然会包含很多无用的代码,Proguard 的压缩功能通过分析字节码,能够检测并移除没有使用到的类、字段、方法和属性。
- 优化: 优化 Java 字节码,同时移除没有使用到的指令。
- 混淆: 使用无意义的简短字母组合对类名、字段名和方法名进行重命名。
- 预检验: 对上述处理后的代码进行预检验。

38.1.2 Proguard 的使能和配置

在 Android Studio 中 Proguard 的使能很简单,只需要在进行混淆的 Module 中的 build.gradle 文件中添加如下配置即可(默认生成的 build.gradle 文件中已经对 Proguard 进行配置,只不过还没有使能)。

```
android {
```

```
...
```

```

buildTypes {
    release {
        minifyEnabled true // true表示使能Proguard
        proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
        'proguard-rules.pro'
    }
}
}

```

其中，proguard-android.txt 是 Proguard 默认的混淆配置文件，位于 Android SDK 的 tools/proguard 目录中，该文件是对 Proguard 基本的配置，几乎是每个 APP 都要用到的配置，当然，这个默认文件的配置远远不够，我们需要根据自身 APP 的特殊性增加或者减少相关的配置，proguard-android.txt 文件内容如下。

```

# 混淆时不使用大小写混合类名
-dontusemixedcaseclassnames

# 指定不去忽略非公共的库类
-dontskipnonpubliclibraryclasses

# 混淆过程打印日志时的级别
-verbose

# dex打包时会自己做优化操作，不希望代码经过Proguard的优化和预校验，因此，在这里需要对这两项去使能
# 不使用优化方案
-dontoptimize
# 混淆时不做预校验
-dontpreverify

# 如果项目中使用到注解，需要保留注解属性
-keepattributes *Annotation*

-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService

```

保持native 方法不作混淆

```
-keepclasseswithmembernames class * {
    native <methods>;
}
```

保持 Views 中的 setter 和 getter 方法不混淆, 保证动画能够正常执行

```
-keepclassmembers public class * extends android.view.View {
    void set*(**);
    ** get*();
}
```

保持 Activity 中参数是 View 类型的函数, 保证在 XML 文件中配置的 onClick 属性的值能够正常调用到

```
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}
```

保持枚举类型中的函数

```
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}
```

保持 Parcelable 类中的 CREATOR 不被混淆

```
-keepclassmembers class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator CREATOR;
}
```

```
-keepclassmembers class **.R$* {
    public static <fields>;
}
```

support library 中包含对新的系统平台版本号的引用, 如果是在旧的系统平台上链接 support

library 的话，不要显示警告信息
`-dontwarn android.support.**`

如果默认的 `proguard-android.txt` 文件的配置不满足 APP 的具体需求，那么需要配置 module 根目录下面的 `proguard-rules.pro` 文件，用这个文件的配置信息替代 `proguard-android.txt` 文件的配置信息。

38.1.3 proguard-rules.pro 文件的编写

混淆文件的规则可以大致分为三种类型：

- 公共的混淆规则：每个APP都通用的，主要是对 Proguard 的基本配置，以及 Android SDK 中 API 设置的规则，例如Activity、Parcelable 等。
- App 特有的混淆规则：根据APP自身的特点进行设置，例如某些类会被反射调用，如果被混淆，那么反射就找不到了。
- 第三方函数库或者 SDK 的混淆规则：如果 APP 引入了第三方开源函数库或者 SDK，那么需要查看这些函数库或者 SDK 的使用说明，将需要去混淆的地方加上去。

下面是一个比较完整的 `proguard-rules.pro` 文件的内容。

```
# 代码迭代优化的次数，取值范围0~7，默认为5
-optimizationpasses 5

# 混淆时不使用大小写混合的方式，这样混淆后都是小写字母的组合
-dontusemixedcaseclassnames

# 混淆时不做预校验，由前面的介绍可以知道，预校验是 Proguard 四大功能之一，在 Android 中
一般不需要预校验，这样可以加快混淆的速度
-dontpreverify

# 混淆时记录日志，同时会生成映射文件，Android Studio 中，生成的默认映射文件是 `build/
outputs/mapping/release/mapping.txt`
-verbose
```


指定生成的映射文件的路径和名称

```
-printmapping build/outputs/mapping/release/mymapping.txt
```

混淆时所采用的算法, 参数是 Google 官方推荐的过滤器算法

```
-optimizations !code/simplification/arithmetic,!field/*,!class/merging/*
```

如果项目中使用到注解, 需要保留注解属性

```
-keepattributes *Annotation*
```

不混淆泛型

```
-keepattributes Signature
```

保留代码行号, 这在混淆后代码运行中抛出异常信息时, 有利于定位出问题的代码

```
-keepattributes SourceFile,LineNumberTable
```

保持 Android SDK 相关 API 类不被混淆

```
-keep public class * extends android.app.Activity
```

```
-keep public class * extends android.app.Application
```

```
-keep public class * extends android.app.Service
```

```
-keep public class * extends android.content.BroadcastReceiver
```

```
-keep public class * extends android.content.ContentProvider
```

```
-keep public class * extends android.app.backup.BackupAgentHelper
```

```
-keep public class * extends android.preference.Preference
```

```
-keep public class com.android.vending.licensing.ILicensingService
```

保留 R 类

```
-keep class **.R$* {
```

```
    *;
```

```
}
```

保持 native 方法不被混淆

```
-keepclasseswithmembernames class * {
```

```
    native <methods>;
```

```
}
```

保持自定义控件类不被混淆

```
-keepclasseswithmembers class * {  
    public <init>(android.content.Context, android.util.AttributeSet);  
}
```

```
-keepclasseswithmembers class * {  
    public <init>(android.content.Context, android.util.AttributeSet, int);  
}
```

保持 Activity 中参数是 View 类型的函数, 保证在 Layout XML 文件中配置的 onClick 属性的值能够正常调用到

```
-keepclassmembers class * extends android.app.Activity {  
    public void *(android.view.View);  
}
```

保持枚举 enum 类不被混淆

```
-keepclassmembers enum * {  
    public static **[] values();  
    public static ** valueOf(java.lang.String);  
}
```

保持 Parcelable 不被混淆

```
-keep class * implements android.os.Parcelable {  
    public static final android.os.Parcelable$Creator *;  
}
```

保持 Serializable 序列化类相关方法和字段不被混淆

```
-keepclassmembers class * implements java.io.Serializable {  
    static final long serialVersionUID;  
    private static final java.io.ObjectStreamField[] serialPersistentFields;  
    !static !transient <fields>;  
    private void writeObject(java.io.ObjectOutputStream);  
    private void readObject(java.io.ObjectInputStream);  
}
```

```

java.lang.Object writeReplace();
java.lang.Object readResolve();
}

```

保持自定义控件不被混淆

```

-keep public class * extends android.view.View {
    public <init>(android.content.Context);
    public <init>(android.content.Context, android.util.AttributeSet);
    public <init>(android.content.Context, android.util.AttributeSet, int);
    public void set*(...);
    *** get*();
}

```

引入 Logansquare 开源库所需增加的混淆配置

```

-keep class com.bluelinelabs.logansquare.** { *; }
-keep @com.bluelinelabs.logansquare.annotation.JsonObject class *
-keep class **$$Json ObjectMapper { *; }

```

38.1.4 Proguard 生成的文件

在 Android Studio 中使用 Proguard 对代码进行混淆，会在 build/outputs/mapping/release 目录中生成四个文件，分别如下。

- dump.txt：列出生成的APK文件中所有class文件的内部结构，如图38-3所示。

```

+ Program class: org/apache/http/entity/mime/MultipartFormEntity
  Superclass:   java/lang/Object
  Major version: 0x31
  Minor version: 0x0
    = target 1.5
  Access flags: 0x20
    = class org.apache.http.entity.mime.MultipartFormEntity extends java.lang.Object

Interfaces (count = 1):
  + Class [org/apache/http/HttpEntity]

Constant Pool (count = 73):
  + String [Content-Type]
  + String [Multipart form entity does not implement #getContent()]
  + String [Streaming entity does not implement #consumeContent()]
  + Class [java/io/IOException]
  + Class [java/lang/Object]

```

图38-3

- mapping.txt: 列出混淆前的Java源码和混淆后的类、方法和属性名字之间的映射, 如图38-4所示。

```
com.asce1885.safetykeyboardlibrary.KeyBoardView -> com.asce1885.safetykeyboardlibrary.a:
    android.widget.Button btn1 -> a
    android.widget.Button btn2 -> b
    android.widget.Button btn3 -> c
    android.widget.Button btn4 -> d
    android.widget.Button btn5 -> e
    android.widget.Button btn6 -> f
    android.widget.Button btn7 -> g
    android.widget.Button btn8 -> h
    android.widget.Button btn9 -> i
    android.widget.Button btn0 -> j
    android.view.View lineardel -> k
    android.widget.Button btnpoint -> l
    android.widget.ImageView exitKey -> m
    com.asce1885.safetykeyboardlibrary.KeyBoardView$onKeyDownBtnListener onKeyDownBtnListener -> n
    com.asce1885.safetykeyboardlibrary.KeyBoardView$onKeyExitBtnListener onKeyExitBtnListener -> o
    android.widget.Button linearBtn0 -> p
    android.widget.Button linearBtnPoint -> q
    android.widget.Button linearLinearDel -> r
    android.widget.Button linearABC -> s
```

图38-4

- seeds.txt: 列出未混淆的类和成员, 如图38-5所示。

```
com.asce1885.safetykeyboardlibrary.KeyBoardView -> com.asce1885.safetykeyboardlibrary.a:
    android.widget.Button btn1 -> a
    android.widget.Button btn2 -> b
    android.widget.Button btn3 -> c
    android.widget.Button btn4 -> d
    android.widget.Button btn5 -> e
    android.widget.Button btn6 -> f
    android.widget.Button btn7 -> g
    android.widget.Button btn8 -> h
    android.widget.Button btn9 -> i
    android.widget.Button btn0 -> j
    android.view.View lineardel -> k
    android.widget.Button btnpoint -> l
    android.widget.ImageView exitKey -> m
    com.asce1885.safetykeyboardlibrary.KeyBoardView$onKeyDownBtnListener onKeyDownBtnListener -> n
    com.asce1885.safetykeyboardlibrary.KeyBoardView$onKeyExitBtnListener onKeyExitBtnListener -> o
    android.widget.Button linearBtn0 -> p
    android.widget.Button linearBtnPoint -> q
    android.widget.Button linearLinearDel -> r
    android.widget.Button linearABC -> s
```

图38-5

- usage.txt: 列出从apk文件中剥离的代码, 如图38-6所示。

```
com.asce1885.safetykeyboardlibrary.BuildConfig
com.asce1885.safetykeyboardlibrary.KeyBoardView:
    private static final int FIRST_DELETE_DELAYED_TIME
    private static final int SECOND_DELETE_DELAYED_TIME
    private static final int SENDEVENT
    514:514:public int getKeyBoradType()
    522:523:public void setBtnpoint(android.widget.Button)
    1236:1236:public android.widget.EditText getEditText()
com.asce1885.safetykeyboardlibrary.KeyView:
    33:34:public void setTextView(android.widget.TextView)
com.asce1885.safetykeyboardlibrary.PAKeyboardEditText:
    public static final int KEYBOARD
    public static final int LETTERKEYBOARD
    private static final char DOT
    237:237:public boolean isDecimal()
    270:270:public boolean isPassword()
```

图38-6

38.1.5 Proguard 混淆规则汇总

Proguard 混淆规则合集有国外的 android-proguard-snippets¹，以及国内的 android-proguard-cn²。

38.2 Native (C/C++) 代码的混淆

Android 开发中经常需要在客户端保存敏感信息，相比较将敏感信息写在 Java 层，将其下移到 NDK 层是个更好的选择，虽然 NDK 层存储敏感信息安全性很好，但仍然避免不了被破解，为了进一步增强破解的难度，我们需要对 NDK 层的代码也进行混淆保护。

相比较 Java 层的代码混淆而已，Native 层代码混淆并没有一个标准的方案或者函数库，常见且比较简单的方法是使用花指令，使得 Native 代码在被反汇编时出错，从而让破解者无法清晰正确的反汇编出代码的内容。

下面我们介绍的 Obfuscator-LLVM³ 是一个值得关注的项目，这个项目专注于 LLVM 编译器，这一点使得它可移植性很高，兼容 LLVM 支持的所有语言（C、C++、Objective-C、Ada and Fortran）和平台（x86、x86-64、PowerPC、PowerPC-64、ARM、Thumb、SPARC、Alpha、CellSPU、MIPS、MSP430、SystemZ、XCore）。我们可以自己从源码开始编译出二进制包，或者可以直接使用由 Ryan Welton⁴ 预先编译的包，具体可参见 AndroidObfuscation-NDK⁵ 这个例子工程。

38.3 资源文件的混淆

跟 Native 代码的混淆类似，资源文件的混淆也没有一个统一的方案。事实上，对资源文件进行混淆这一操作并不常见，因为资源文件的保密性没有代码那么高。虽然如此，在项目中对资源文件进行混淆还是有如下好处。

- 提高 APP 被破解的难度：资源文件虽然不会暴露代码的逻辑，但是如果资源文件不进行混淆的话，破解者可以通过 apktool 这个工具轻易的得到原始的资源文件，破解者通

1 <https://github.com/krschultz/android-proguard-snippets>

2 <https://github.com/msdx/android-proguard-cn>

3 <https://github.com/obfuscator-llvm/obfuscator>

4 <https://github.com/fuzion24>

5 <https://github.com/Fuzion24/AndroidObfuscation-NDK>

过资源文件的命名以及内容作为辅助，可以加速对代码的理解和破解。

- 减少 APP 的最终包大小：资源文件的混淆类似 Java 代码的混淆，也是通过使用无意义的字母来代替完整的命名实现的，它的一个副作用就是能够一定程度上减少 APP 的包大小。

资源文件的混淆方案目前有美团¹和微信²两种，前者是通过修改 AAPT 在处理资源文件相关的源码达到资源文件名的替换；后者是通过直接修改 `resources.arsc` 文件达到资源文件名的混淆。相比之下，微信的方案更优，而且微信已经将这个方案开源出来，地址在 AndResGuard³，感兴趣的同学通过上面的 README.md 介绍可以很容易集成到自己的项目中。

1 <http://tech.meituan.com/mt-android-resource-obfuscation.html>

2 http://mp.weixin.qq.com/s?__biz=MzAwNDY1ODY2OQ==&mid=208135658&idx=1&sn=ac9bd6b4927e9e82f9fa14e396183a8f#rl

3 <https://github.com/shwenzhang/AndResGuard>

第39章

Android 反编译机制详解

从事 Android 开发有一定年限的程序员，应该都遇到过产品经理或者设计师拿着竞品或者某个做的不错的第三方 APP 跟你说要实现它里面的某某功能或者效果。一般在这种情况下，我们都会通过反编译工具去看看这个 APP 是如何实现这个功能的，一来可以有一定的借鉴作用，减少自己研发的时间成本，二来也可以在对方实现的基础上精益求精。本章就来介绍如何对 APP 进行反编译。

一个 APP 的包中包含代码文件和资源文件，因此反编译也相应得分为两部分：对资源文件的反编译以及对 Java 代码的反编译。

39.1 资源文件的反编译

Android 中对资源文件的反编译一般使用 ApkTool¹ 这个工具，它可以反编译 resources.arsc、9.png 和 XML 等文件。同时它也可以将经过修改的反编译文件重新打包成 APK 文件。

39.1.1 ApkTool 的安装

ApkTool 支持 Windows / Linux / Mac OS X，下面我们以 Mac OS X 平台上面的安装方法为例进行说明，其它平台类似，具体可参见官网上面的安装说明。

- 打开 wrapper script²页面，将该页面的内容保存为 apktool 文件。
- 到 ApkTool 托管地址³下载最新的版本，并重命名为 apktool.jar。

1 <http://ibotpeaches.github.io/Apktool/>

2 <https://raw.githubusercontent.com/iBotPeaches/Apktool/master/scripts/osx/apktool>

3 <https://bitbucket.org/iBotPeaches/apktool/downloads>

- 将 apktool.jar 和 apktool 这两个文件移动到 /usr/local/bin 目录中（需要有 root 权限），同时通过命令 `chmod +x` 命令给这两个文件增加可执行权限。

其中，apktool 这个脚本是为了方便我们在命令行 Terminal 中可以直接执行 apktool 命令而增加的，不使用这个脚本，那么运行 Apktool 工具的命令会比较繁琐，语句如下。

```
java -jar apktool.jar
```

39.1.2 ApkTool 的使用

ApkTool 安装完成之后，我们可以到应用市场上面随便下载一个 APK 文件，重命名为 test.apk。在命令行中进入这个文件所在目录，执行命令 `apktool d test.apk` 进行资源的反编译，成功的话，我们在 APK 同级目录中可以得到一个新的目录，里面的内容如图 39-1 所示。

名称	修改日期	大小	种类
OS_Stats	今天 下午6:54	6 KB	文图
AndroidManifest.xml	今天 下午6:51	27 KB	XML Doc
apktool.yml	今天 下午6:51	516 字节	YAML
assets	今天 下午6:51	--	文件夹
lib	今天 下午6:53	--	文件夹
original	今天 下午6:53	--	文件夹
res	今天 下午6:54	--	文件夹
smali	今天 下午6:54	--	文件夹
unknown	今天 下午6:53	--	文件夹

图 39-1

其中包括明文形式的图片资源、布局文件资源、动画资源以及 APK 代码的 smali 文件等，这样我们就可以对这些文件进行分析研究，例如可以知道 APK 中某个页面是如何布局的。而且我们可以通过修改其中的某些资源文件或者 smali 文件，然后通过命令 `apktool b test` 重新打包成 APK。

ApkTool 的主要功能如上面所介绍，借用官网的一张图也可以看出这个工具的用途，如图 39-2 所示。

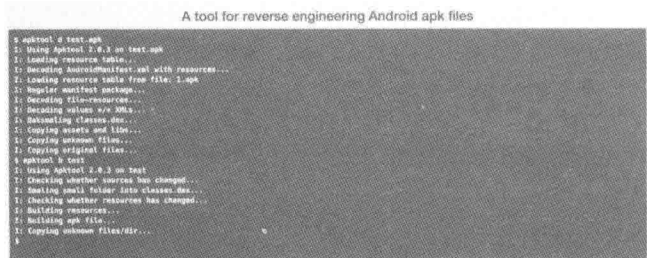


图39-2

39.2 Java 代码的反编译

反编译 Java 代码的常用工具是 dex2jar¹，这个工具的输入是 APK 文件中的 classes.dex 文件，输出是一个 jar 文件。我们把前面的 test.apk 文件解压后，将其中的 classes.dex 文件拷贝到 dex2jar 根目录中，在命令行 Terminal 中执行如下命令进行字节码的反编译。

```
./d2j-dex2jar.sh classes.dex
```

完成之后，可以在 dex2jar 目录中得到一个 jar 文件，我们可以使用 JD-GUI² 来打开它，如图 39-3 所示。

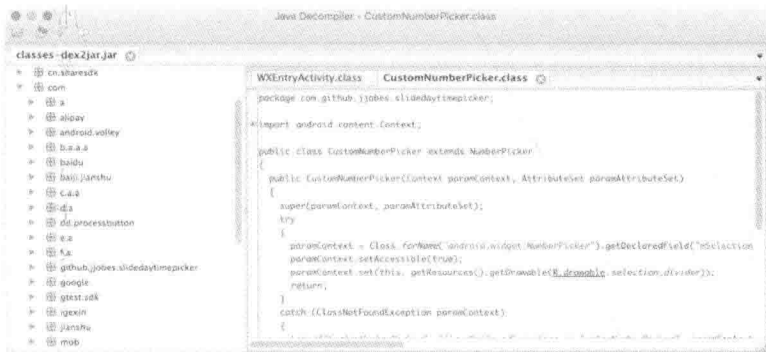


图39-3

¹ <https://github.com/pxb1988/dex2jar>

² <http://jd.benow.ca/>

第40章

客户端敏感信息隐藏技术研究

Android 开发中经常会遇到需要在客户端对一些敏感信息进行保存的情况，例如我们使用开源库 SQLCipher¹ 对数据库进行加密；我们使用 Facebook 的 Conceal² 对磁盘上大文件的加密。无论哪种情况，都会涉及加密所用的密钥如何在客户端安全存放的问题。另外，现代的 APP 一般都会集成很多第三方的 SDK，用于实现诸如支付、统计、推送等功能，这时一般都需要向这些第三方 SDK 官方网站申请 API Key，我们当然不希望这些密钥或者 API Key 能被第三方通过反编译很轻易地获取。因此，如何在客户端保存敏感信息，是关系到整个 APP 是否安全的关键问题。

常见的敏感信息隐藏策略主要有。

- 敏感信息嵌套在 strings.xml 中。
- 敏感信息隐藏在 Java 源代码中。
- 敏感信息隐藏在 BuildConfig 中。
- 使用 DexGuard。
- 对敏感信息进行伪装或者加密。
- 敏感信息隐藏在原生函数库中（.so 文件）。
- 对 APK 进行加固处理。

1 <https://github.com/sqlcipher/sqlcipher>

2 <https://github.com/facebook/conceal>

40.1 敏感信息嵌套在 strings.xml 中

由于密钥或者 API Key 一般都是以字符串的形式存在，最常见的一种保存方法是将其存放在工程的 res/values/strings.xml 文件中。

```
<resources>
    <string name=" statistic_appkey" >512b45e0527015964a0000a7</string>
</resources>
```

这种方法的安全性是极低的，第三方只需要使用 ApkTool 等工具对 APK 的资源文件进行反编译，就能轻易地拿到这个 strings.xml 文件，里面的内容是一目了然的。

40.2 敏感信息隐藏在 Java 源代码中

将密钥或者 API Key 以字符串或者字符数组的形式硬编码到代码中，这也是普通开发者常用的一种方式，代码如下。

```
public class MainActivity extends AppCompatActivity {

    // 以字符串形式在源代码中存放统计SDK的API Key
    private static final String STATISTIC_APP_KEY =
        "512b45e0527015964a0000a7" ;

    // 以字符数组形式存放，相比字符串形式安全一些
    private static final byte[] STATISTIC_APP_KEY_ARRAY = new byte[] {
        '5', '1', '2', 'b', '4', '5', 'e', '0', '5', '2',
        '7', '0',
        '1', '5', '9', '6', '4', 'a', '0', '0', '0', '0',
        'a', '7'
    };

    ...
}
```

同样的，这种方式的安全性也是极低的，通过使用 dex2jar 等工具对 classes.dex 文件进行

反编译，可以得到 classes-dex2jar.jar 包，这时使用 JD-GUI 可以查看反编译后的源码如下。

```
public class MainActivity extends AppCompatActivity
{
    private static final String STATISTIC_APP_KEY =
        "512b45e0527015964a0000a7" ;

    private static final byte[] STATISTIC_APP_KEY_ARRAY = arrayOfByte;

    static
    {
        byte[] arrayOfByte = new byte[24];
        arrayOfByte[0] = 53;
        arrayOfByte[1] = 49;
        arrayOfByte[2] = 50;
        arrayOfByte[3] = 98;
        arrayOfByte[4] = 52;
        arrayOfByte[5] = 53;
        arrayOfByte[6] = 101;
        arrayOfByte[7] = 48;
        arrayOfByte[8] = 53;
        arrayOfByte[9] = 50;
        arrayOfByte[10] = 55;
        arrayOfByte[11] = 48;
        arrayOfByte[12] = 49;
        arrayOfByte[13] = 53;
        arrayOfByte[14] = 57;
        arrayOfByte[15] = 54;
        arrayOfByte[16] = 52;
        arrayOfByte[17] = 97;
        arrayOfByte[18] = 48;
        arrayOfByte[19] = 48;
        arrayOfByte[20] = 48;
        arrayOfByte[21] = 48;
```

```

        arrayOfByte[22] = 97;
        arrayOfByte[23] = 55;
    }
}

```

40.3 敏感信息隐藏在 BuildConfig 中

Android Gradle Plugin 为我们提供了 BuildConfig 文件，我们可以把敏感信息存放在这里，同时通过将敏感信息存放在工程的 gradle.properties 中，可以避免将其上传到版本控制系统（svn 或者 git 等）上，从而将敏感信息控制在少数人手里，而不是暴露给所有具有 svn 或者 git 权限的人。

要将信息保存到 BuildConfig 文件中，需要在 app/build.gradle 文件中设置如下。

```

android {
    ...

    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
            'proguard-rules.pro'
            buildConfigField "String", "statisticAppKey",
            "\" ${statisticAppKey}\" "
        }
    }
}

```

其中 buildConfigField 函数用于给 BuildConfig 文件添加一个字段，原型如下。

```

// 用于给 BuildConfig 文件添加一个字段定义
public void buildConfigField(
    @NonNull String type,
    @NonNull String name,
    @NonNull String value) {

```

```

ClassField alreadyPresent = getBuildConfigFields().get(name);
if (alreadyPresent != null) {
    logger.info("BuildType(${getName()}): buildConfigField '$name'
value is being replaced: ${alreadyPresent.value} -> $value");
}
addBuildConfigField(AndroidBuilder.createClassField(type, name, value));
}

```

而“\” `$(statisticAppKey)\`”的取值定义在工程的 `gradle.properties` 文件中。

```
statisticAppKey=512b45e0527015964a0000a7
```

这种方式安全级别也是很低的，我们对生成的 APK 进行反编译之后，查看 `BuildConfig.class` 文件，可以直接看到 API Key 的取值。

```

public final class BuildConfig {
    ...

    public static final int VERSION_CODE = 1;
    public static final String VERSION_NAME = "1.0";
    public static final String statisticAppKey = "512b45e0527015964a0000a7";
}

```

40.4 使用 DexGuard

我们知道，Android 已经默认集成了 ProGuard，它是一个免费的用于压缩、优化和混淆 Java 字节码的工具，混淆的功能主要是使用简短的无意义的字母组合来对代码中的类、字段、方法和属性进行重命名，但它无法对字符串进行混淆。也就是说，使用 ProGuard 之后，我们还是可以看到反编译后代码中完整的字符串定义。为了实现更高级的混淆和加密功能，我们可以选择商业版本的 ProGuard——DexGuard¹。DexGuard 对代码、资源、字符串、`AndroidManifest.xml` 等进行了全面的加密和混淆，相比 ProGuard，功能强大了不少。官方的一个比较图如图 40-1 所示。

¹ <http://www.guardsquare.com/dexguard>

	ProGuard	DexGuard	DexGuard Enterprise
Availability	Open source	Individual developers	Larger companies
Size improvements	✓	✓	✓
Performance improvements	✓	✓	✓
Name obfuscation	✓	✓	✓
Code protection		✓	✓
Resource protection			✓
Application integrity			✓
Platform integrity			✓
Communication hardening			✓

图40-1

40.5 对敏感信息进行伪装或者加密

虽然 DexGuard 功能强大,也能帮助我们对敏感字符串信息进行加密,但毕竟需要付费使用,并不是每个公司都会愿意付费。因此多数情况下,只能靠我们对 Java 代码中的敏感信息进行伪装或者加密,以加大第三方破解的难度。最简单的我们可以使用 Base64 对字符串进行编码,然后将编码后的字符串和另外一串密钥进行异或操作,从而得到伪装后的字符串。当然,这个用于异或的密钥如何存放,就是一个先有鸡还是先有蛋的问题了。

40.6 敏感信息隐藏在原生函数库中 (.so文件)

为了增大第三方获取敏感信息的难度,我们可以进一步把敏感信息的存放从 Java 层下移到 Native 层,也就是存放在 .so 文件中。然后通过 jni 封装对敏感信息的获取接口。至于敏感信息在 C/C++ 层如何存放,我们可以重复 Java 层的伪装和加密方式,另外可以在 C/C++ 层通过花指令的方式来使得反汇编 .so 文件的时候出错,来增加 .so 文件被破解的难度。

40.7 对APK进行加固处理

近年来,涌现了很多 APK 加固平台,APK 的加固处理极大地增加了反编译的难度,经过加固平台的加固后,想使用普通的 dex2jar、Apktool 等工具进行反编译几乎都是失败的,这使

得普通开发者再也无法轻易地窥视发布在应用市场上面的 APP 的代码或者资源了。

上面介绍的敏感信息隐藏技术，从破解难度上面讲，是从易到难的顺序。当然，正所谓道高一尺魔高一丈，这些技术本身无论难度如何，都存在被破解的可能，在客户端进行敏感信息隐藏的一个目标就是尽可能地增加破解的难度，只要破解所需花费的精力超过所得的好处，那就已经做的很不错的了。

第41章

Android 加固技术研究

Android 应用加固是指在 APK 的外面加一层壳, 并对 APK 里面的 dex 文件进行加密, 它可以有效防止 APP 被反编译, 是保证 APP 安全的最后一道防线。近几年涌现了很多应用加固平台, 比较知名的有梆梆加固¹、爱加密²、娜迦³、360 加固保⁴、腾讯加固⁵、百度加固⁶等。不同的平台提供的加固功能相差无几, 原理也大同小异。下面我们以爱加密为例进行说明, 帮助大家一窥应用加固的常见内容。

41.1 爱加密的主要功能

41.1.1 漏洞分析

- 文件检查: 检查 dex、res 文件是否存在源代码、资源文件被窃取、替换等安全问题。
- 漏洞扫描: 扫描签名、XML 文件是否存在安全漏洞、存在被注入、嵌入代码等风险。
- 后门检测: 检测 APP 是否存在被二次打包, 然后植入后门程序或第三方代码等风险。
- 一键生成: 一键生成 APP 关于源码、文件、权限、关键字等方面的安全风险分析报告。

1 <http://www.bangcle.com/>

2 <http://www.ijiami.cn/>

3 <http://www.nagain.com/>

4 <http://jiagu.360.cn/>

5 <http://www.qqcloud.com/wiki/%E5%BA%94%E7%94%A8%E5%8A%A0%E5%9B%BA>

6 <http://apkprotect.baidu.com/>

41.1.2 加密服务

- DEX 加壳保护：DEX 文件加壳保护对 DEX 文件进行加壳防护，防止被静态反编译工具破解获取源码。
- 内存防 dump 保护：防止通过使用内存 dump 方法对应用进行非法破解。
- 资源文件保护：应用的资源文件被修改后将无法正常运行。
- 防二次打包保护：保护应用在被非法二次打包后不能正常运行。
- 防调试器保护：防止通过使用调试器工具（例如：zjdroid）对应用进行非法破解。
- 多渠道打包：上传 APK 时，通过选择 android:name 和填写 android:value 来实现对每一个渠道的包的生成和加密。
- 漏洞分析服务：漏洞分析采用文件检查、漏洞扫描、后门检测等技术方向对 APK 进行静态分析并支持一键生成分析报告。
- 渠道监测服务：监控国内 400 多个渠道市场入口，对应用的各渠道的下载量、版本信息、正盗版进行一站监控。
- 签名工具：爱加密提供纯绿色签名工具，支持 Windows、Linux 和 MAC 系统，同时支持批量签名。
- DEX 专业加壳保护：本服务是对安卓 DEX 文件进行加壳保护，有效防止所有静态调试器对 APK 进行破解。
- DEX 专业加花保护：本服务对安卓 DEX 文件进行加入花指令（无效字节码）保护。
- 资源文件指纹签名保护：对资源文件指纹签名进行验证保护，有效防止资源文件被篡改。
- 高级防二次打包保护：本服务对 APK 进行防止二次打包保护，防止 APK 被使用非法手段修改替换文件后进行二次打包。
- 高级防调试器保护：防止通过使用调试器工具（如：zjdroid、APK改之理、ida 等）对应用进行非法破解。
- 高级内存保护：本服务是对内存数据的专业高级保护，可防止内存调试，防止通过 dump 获取源码，防止内存修改。

- 截屏防护：防止黑客通过截屏形式获取应用账号、应用密码、支付银行卡号、支付银行卡密码,支持安卓所有机型。
- 本地数据文件保护：对 APK 应用的网络缓存数据、本地储存数据(提供SDK)进行深度保护。
- 源码优化：（1）一键清除Log（开发日志）信息；（2）一键优化缩小加密后增大的源用包大小。
- 防止脚本：本服务提供防止脚本 SDK，用户根据开发帮助文档进行二次开发，此保护项可有效防止游戏非法使用脚本。
- 防止加速器：防止游戏使用加速器，破坏游戏公平(如：防八门神器和葫芦侠中的加速器功能)。
- 防止模拟器运行：防止模拟器非法运行（可以防止运行在 PC 上的任何类型的 Android 模拟器）。
- 防止内购破解：防止游戏被内购破解（如：游戏内部有支付项，可以防止支付项相关内容被破解）。
- .so文件保护：对.so文件进行优化压缩、源码加密隐藏、防止调试器逆向分析。

41.1.3 渠道监测

- 渠道数据监控。
- 精准识别渠道正盗版。
- 盗版APP详情分析。

41.2 常见 APP 漏洞及风险

41.2.1 静态破解

通过工具 Apktool、dex2jar、jd-gui、DDMS、签名工具，可以对任何一个未加密应用进行静态破解，窃取源码。

41.2.2 二次打包

通过静态破解获取源码，嵌入恶意病毒、广告等行为再利用工具打包、签名，形成二次打包应用。

41.2.3 本地储存数据窃取

通过获取 root 权限，对手机中应用储存的数据进行窃取、编辑、转存等恶意行为，直接威胁用户隐私。

41.2.4 界面截取

通过 adb shell 命令或第三方软件获取 root 权限，在手机界面截取用户填写的隐私信息，随后进行恶意行为。

41.2.5 输入法攻击

通过对系统输入法攻击，从而对用户填写的隐私信息进行截获、转存等恶意操作，窃取敏感信息。

41.2.6 协议抓取

通过设置代理或使用第三方抓包工具，对应用发送与接收的数据包进行截获、重发、编辑、转存等恶意操作。

41.3 Android 程序反破解技术

41.3.1 对抗反编译

对抗反编译是指 APK 文件无法通过反编译工具（例如 ApkTool、BakSmali、dex2jar 等）对其进行反编译，或者反编译后无法得到软件正确的反汇编代码。基本思路是寻找反编译工具在

处理 APK 或者 DEX 文件时的缺陷，然后在自己的代码中加以利用，让反编译工具在处理 APK 文件的时候抛出异常或者反编译失败，有两种方法可以找到反编译工具的缺陷。

- 阅读反编译工具的源码。
- 压力测试。

41.3.2 对抗静态分析

反编译工具一直在改进，因此即使你在版本 2.1 发现它的缺陷并加以利用，使反编译你的 APK 失败，但很可能在版本 2.2 就把这个缺陷解决了，因此，不要指望反编译工具永远无法反编译你的 APK，我们还需要使用其他方法来防止 APK 被破解。

- 代码混淆技术，ProGuard 提供了压缩（Shrinking）、混淆（Obfuscation）、优化（Optimisation）Java 代码和反混淆栈跟踪（ReTrace）的功能。
- NDK 保护：逆向 NDK 程序的汇编代码比逆向 Java 代码枯燥和困难很多，同时使用 C++ 也可以对敏感字符串和代码进行加密。
- 外壳保护：针对 NDK 编写的 Native 代码。

41.3.3 对抗动态调试

- 检测调试器：动态调试使用调试器来挂钩 APK，获取 APK 运行时的数据，因此，我们可以在 APK 中加入检测调试器的代码，当检测到 APK 被调试器连接时，终止 APK 的运行。
- 检测模拟器：APK 发布后，如果发现其运行在模拟器中，很有可能是有人试图破解或者分析它，因此这时我们也要终止 APK 的运行。

41.3.4 防止重编译

- 检查 APK 的签名。
- 校验 APK 的完整性。
- 校验 classes.dex 文件的完整性。

41.4 加固技术研究知识储备

前面已经分析了目前主流的第三方解决方案，并总结了常见的 APP 漏洞和风险，最终提出了反破解措施，下面是对 Android 程序反破解技术的进一步细化，看看如何系统地掌握应用加固的知识。在对 APK 进行加固之前，首先需要了解如何破解 APK，对破解技术掌握得越多越深，就能够更好地对其进行反破解，也就是加固。而要得到这一点，需要一系列的知识储备。

41.4.1 掌握常见的破解分析工具

1. Smali and Baksmali

Smali 和 baksmali 分别是 dex 格式文件的汇编器和反汇编器。

2. Androguard

Androguard 是使用 python 编写的 Android 逆向工具，主要支持如下文件的逆向。

- DEX、ODEX
- APK
- Android 的二进制形式的 XML

3. Apktool

Apktool 用于编译和反编译 Android APK，主要有如下特性。

- 反汇编资源文件（包括 resources.arsc、classes.dex、9.png 和 XML）。
- 将资源文件重新编译成二进制的 APK / JAR。
- 根据应用程序框架的资源组织和处理 APK 文件。
- 支持单步调试 smali 文件。

4. dex2jar

dex2jar 是一个用于操作 Android 的 .dex 文件和 .class 文件的工具包，它是使用 Java 语言编写的，主要提供了如下工具。

- dex-reader/writer: 用于读写 Dalvik 可执行 .dex 文件格式. 包含一个类似ASM的轻量级 API。
- d2j-dex2jar: 将 .dex 文件转换成 .class 文件集合（压缩成为一个Jar包）。
- smali/baksmali: 将 .dex 文件反汇编成 .smali 文件，将 .smali 文件汇编成 .dex 文件。与 smali工具功能一致，但是对中文更友好。
- 其他工具如：d2j-decrypt-string。

5. jad

jad 是一个闭源的 Java 反编译器，而且已经不再维护了。它提供命令行接口将 .class 文件还原成 Java 源文件。

6. JD-GUI

JD-GUI 是一个闭源的 Java 反编译器，用于从 .class 文件重新构建还原出 Java 源文件，同时提供一个用户界面以方便浏览反编译后的源代码。通常和 jad 互为补充，因为有时一个反编译器产生的结果可能比另一个好，反之亦然。

7. JEB

JEB 是一个闭源的、商业的 Dalvik 字节码反编译器，用于从 Android DEX 文件生成可读的 Java 源代码。

8. Radare2

Radare2 是开源的，可移植的用于处理二进制文件的逆向框架。

9. IDA Pro 和 Hex-Rays 反编译器

IDA PRO 简称 IDA（Interactive Disassembler），是一个世界顶级的交互式反汇编工具，有两种可用版本。标准版（Standard）支持二十多种处理器。高级版（Advanced）支持 50 多种处理器。

Hex-Rays 反编译器是 IDA Pro 的一个插件，用于将 IDA Pro 反汇编后得到的结果转换为可读的 C 语言伪代码。

41.4.2 掌握 Dalvik 指令集代码

Dalvik 虚拟机为自己专门设计了一套指令集，并且制定了自己的指令格式和调用规范。Dalvik 指令是 DEX 文件最主要的组成部分，因此静态分析 DEX 文件首先需要熟悉这套指令集。

41.4.3 掌握 Dex 和 Odex 文件格式

在实际分析 Android APK 的过程中，dex 或者 odex 文件无疑是最常见的，因此这个也是基础功。

41.4.4 掌握 Smali 文件格式

使用 Apktool 反编译 APK 文件后，会在反编译工程目录中生成一个 smali 文件夹，里面存放的就是反编译后得到的 smali 文件。无论是分析 APK 的代码逻辑，还是对 APK 进行二次修改，抑或是对 smali 进行乱序，都需要掌握 smali 文件格式。

41.4.5 掌握基于 Android 的 ARM 汇编语言基础

Android NDK 允许开发人员通过 C/C++ 代码来编写 APP 的核心功能代码，这些代码通过编译会生成基于特定处理器的可执行文件，对于使用 ARM 处理器的 Android 手机而言，它最终会生成 ARM elf 可执行文件。进行 APK 破解或者加固时，需要分析这个 elf 文件，因此，无论是静态分析还是动态调试这个 ARM elf 代码都需要具备基本的 ARM 反汇编代码阅读能力。

第42章

Android安全编码

随着移动端安全问题重要性日益提高，编写安全的代码是检验一个工程师是否合格的基本标准，因此，我们有必要了解目前主流的 Android 漏洞原理和规避方案，让读者在编码的同时有意识地写出安全的代码。

42.1 WebView 远程代码执行

在 Android API Level 16 (Android 4.2) 及之前的系统上，如果使用 WebView.addJavascriptInterface 方法来实现通过 JavaScript 调用应用本地 Java 接口时，由于系统没有对注册的 Java 类方法调用做任何限制，导致攻击者可以通过使用 Java 反射 API 利用该漏洞来执行任意 Java 对象的方法，从而达到攻击的目的。在 Javascript 中注入 Java 对象 injectedObject 的漏洞代码如下。

```
mWebView = new WebView(this);
mWebView.getSettings().setJavaScriptEnabled(true);
mWebView.addJavascriptInterface(this, "injectedObject");
mWebView.loadUrl("http://www.ascel885.com/index.html");
```

攻击者页面的 HTML 代码如下，通过 injectedObject 对象可以获得 Android API 的 getRuntime 方法，从而可以执行一系列 shell 命令，为所欲为。

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" dir="ltr" >
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```

<script type="text/javascript">
    function execute(cmdargs) {
        return injectedObject.getClass().forName("java.lang.Runtime").
getMethod("getRuntime", null).invoke(null, null).exec(cmdArgs);
    }
</script>
</head>
...
</html>

```

该漏洞的解决方案如下。

- 对于 API Level > 16 的系统来说，Android 给出了解决方案，就是规定允许被调用的本地 Java 函数必须使用 @JavascriptInterface 注解进行标记，新的用法如下。

```

class JsNativeObject {
    @JavascriptInterface
    public void showShareWindow() {...}
}

mWebView = new WebView(this);
mWebView.getSettings().setJavaScriptEnabled(true);
mWebView.addJavascriptInterface(new JsNativeObject(), "injectedObject");
mWebView.loadUrl("http://www.ascel885.com/index.html");

```

- 对于 API Level <= 16 的系统来说，强烈建议不要再使用 addJavascriptInterface 接口，转而使用 safe-java-js-webview-bridge^[1] 这个函数库。
- 移除 Android 系统内部的如下默认接口。

```

removeJavascriptInterface("searchBoxJavaBridge_");
removeJavascriptInterface("accessibility");
removeJavascriptInterface("accessibilityTraversal");

```

42.2 WebView 密码明文保存

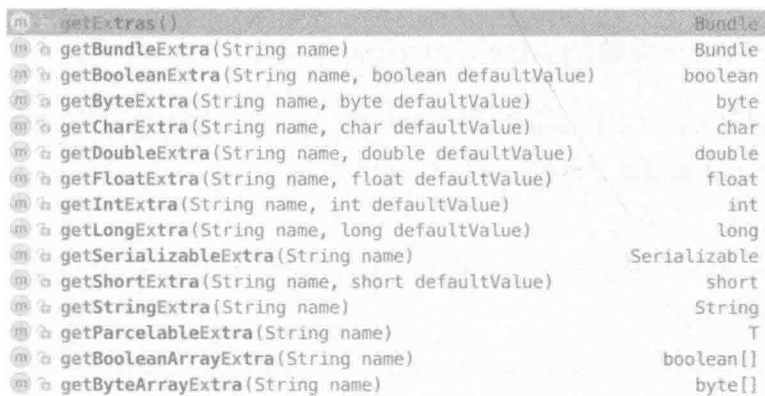
默认情况下，WebView 会保存用户输入的用户名和密码信息，以明文形式保存在 data/data/

应用的包名 /databases/webview.db 文件中。如果手机被 root，那么密码信息很容易泄漏。因此，在 WebView 初始化时，一定要记得显式设置不在 WebView 中保存密码。

```
mWebView.getSettings().setSavePassword(false);
```

42.3 Android 本地拒绝服务

Android 提供了 Intent 机制来实现四大组件间的交互和通信，Intent 中包含了要通信的组件，以及需要传递给它的数据，Android 系统根据 Intent 实现对组件的调用。本地拒绝服务的根本原因在于 Intent.get***Extra() 系列函数，其中的一部分如图 42-1 所示。



getExtras()	Bundle
getBundleExtra(String name)	Bundle
getBooleanExtra(String name, boolean defaultValue)	boolean
getByteExtra(String name, byte defaultValue)	byte
getCharExtra(String name, char defaultValue)	char
getDoubleExtra(String name, double defaultValue)	double
getFloatExtra(String name, float defaultValue)	float
getIntExtra(String name, int defaultValue)	int
getLongExtra(String name, long defaultValue)	long
getSerializableExtra(String name)	Serializable
getShortExtra(String name, short defaultValue)	short
getStringExtra(String name)	String
getParcelableExtra(String name)	T
getBooleanArrayExtra(String name)	boolean[]
getByteArrayExtra(String name)	byte[]

图42-1

攻击者可以通过构造自定义序列化类对象，并通过 Intent.put***Extra() 函数设置给 Intent，攻击者使用该 Intent 调起对应的组件时，组件由于无法找到序列化类对象的定义，从而抛出异常，导致 APP 发生 Crash。本地拒绝服务可以绕过手机中安装的安全卫士等 APP 的扫描，从而被恶意竞争对手利用来使得你的 APP 在用户手机上一启动就 Crash，当初打车软件恶习竞争阶段就曾经出现过类似事件，当然原理可能不同。

本地拒绝服务存在的情况多种多样，下面举例说明。

42.3.1 非法序列化对象导致的 ClassNotFoundException

当应用使用 getSerializableExtra 函数获取序列化类对象时，如果没有增加异常捕获逻辑，

那么第三方攻击者可以通过构造应用中不存在的序列化类并作为 extra 传入 Intent，那么在调用你的应用时将发生 Crash，漏洞代码如下。

```
Intent intent = getIntent();
intent.getSerializableExtra( "serializable_object_key" );
```

攻击代码如下。

```
Intent intent = new Intent();
intent.setComponent( new ComponentName( yourPackageName, yourClassName ) );
intent.putExtra( "serializable_object_key", new ThirdPartyObject() );
startActivity(intent);
```

42.3.2 空 Action 导致的 NullPointerException

当需要使用到 Intent 中的 action，但如果在获取 action 时没有进行判空处理，那么就会被攻击者利用来达到拒绝服务的目的，编写的代码如下。

```
Intent intent = getIntent();
if ( intent.getAction().equals( "android.intent.action.ascel885.share" ) ) {
    // do something
}
```

攻击者构造的代码如下。

```
Intent intent = new Intent();
intent.setComponent( new ComponentName( yourPackageName, yourClassName ) );
startActivity(intent);
```

修复这个漏洞的方法很简单，除了对 getAction 返回值进行判空之外，如果平时养成比较时将常量写在前面的好习惯，其实不判空也不会出现问题。

```
Intent intent = getIntent();
if ( "android.intent.action.ascel885.share".equals(intent.getAction()) ) {
    // do something
}
```

42.3.3 强制类型转换导致的 ClassCastException

在需要强制类型转换的函数调用中，例如 `getSerializableExtra` 函数。

```
User user = (User)intent.getSerializableExtra( "key_user" );
```

如果 `Intent` 中传入的 `key_user` 的值不是 `User` 类型，那么会出现 `ClassCastException` 异常，导致应用 Crash。

攻击代码如下。

```
Intent intent = new Intent();
intent.setComponent(new ComponentName(yourPackageName, yourClassName));
intent.putExtra( "key_user", 10);
```

42.3.4 数组越界导致的 IndexOutOfBoundsException

使用 `Intent.get***ArrayExtra` 系列函数时，如果没有对数组边界做判断，而是使用过约定的数组长度，那么会被攻击者利用，漏洞代码如下。

```
long[] array = intent.getLongArrayExtra( "long_array" );
if (null != array) {
    for (int i=0; i<50; ++i) {
        Log.d(TAG, "array value: " + array[i]);
    }
}
```

如果数组元素个数少于 50 个，那么上述代码运行必然会遇到 `IndexOutOfBoundsException` 异常。

本地拒绝服务解决方案总结起来有：

- 将不需要给其它 APP 调用的组件在 `AndroidManifest.xml` 文件中设置 `exported="false"`。
- 使用 `Intent` 获取 extra 数据时增加 `try...catch` 异常捕获。
- 注意 `getAction` 为空的处理。
- 使用 `Intent` 获取数组，列表等数据时要做长度校验。

- 强制类型转换时要增加 try...catch 异常捕获。

42.4 SharedPreference 全局任意读写

创建 SharedPreferences 时可以指定多种模式，其中包括：

- Context.MODE_PRIVATE：私有读写，只有自己的 APP 可以访问。
- Context.MODE_WORLD_READABLE：全局可读，其他 APP 也可以读取。
- Context.MODE_WORLD_WRITEABLE：全局可写，其他 APP 也可以写入。

如果设置为全局可读写的模式，那么攻击者可能会读取、篡改、伪造其中存放的内容，从而实行诈骗等行为，造成用户财产的损失。正确的做法是以 MODE_PRIVATE 模式创建，语句如下。

```
SharedPreferences safePref = getSharedPreferences("my_pref", Context.MODE_PRIVATE);
```

创建的文件位于“/data/data/APP 包名 /shared_prefs/my_pref.xml”中，如果手机被 root 之后，用户可以随意查看这个文件，因此，我们不应该在这个文件中以明文形式存放敏感信息，例如用户的密码等数据，必须确保以密文形式存放。

42.5 密钥硬编码

用于敏感数据传输和本地数据加密的密钥，不能硬编码在代码中，更详细的内容可以参见本书第 40 章。

42.6 AES/DES/RSA 弱加密

在使用加密算法时，如果对加解密相关知识和 API 没有一个正确完整的了解，可能会弱化加密的效果，从而增大被黑客攻击的可能性，常见的问题主要有以下几个方面。

- 使用 DES 而不是 AES 加密算法：DES 是一种弱对称加密算法，一般不再建议在商业产

品中使用，而是应该改而使用更安全的 AES，使用 AES 加密的代码如下。

```
/**
 * 加密
 *
 * @param content 需要加密的内容
 * @param password 加密密码
 * @return
 */
public static byte[] encrypt(String content, String password) {
    try {
        KeyGenerator kgen = KeyGenerator.getInstance("AES");
        kgen.init(128, new SecureRandom(password.getBytes()));
        SecretKey secretKey = kgen.generateKey();
        byte[] enCodeFormat = secretKey.getEncoded();
        SecretKeySpec key = new SecretKeySpec(enCodeFormat, "AES");
        Cipher cipher = Cipher.getInstance("AES");// 创建密码器
        byte[] byteContent = content.getBytes("utf-8");
        cipher.init(Cipher.ENCRYPT_MODE, key);// 初始化
        byte[] result = cipher.doFinal(byteContent);
        return result; // 加密
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (NoSuchPaddingException e) {
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    }
}
```

```

    return null;
}

```

- AES 使用不安全的加密模式：AES 的加密模式有 ECB、CBC、CFB 等，其中 ECB 模式是不安全的，如下是存在风险的代码。

```

SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
cipher.init(Cipher.ENCRYPT_MODE, key);

```

- 使用不安全的密钥长度：如果使用 RSA 加密，密钥长度小于 512 bit，那么也是不安全的，如下是正确的初始化 RSA 密钥的方法。

```

/**
 * 生成公钥和私钥
 * @throws NoSuchAlgorithmException
 *
 */
public static HashMap<String, Object> getKeys() throws
NoSuchAlgorithmException {
    HashMap<String, Object> map = new HashMap<String, Object>();
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
    keyPairGen.initialize(1024);
    KeyPair keyPair = keyPairGen.generateKeyPair();
    RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
    RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
    map.put("public", publicKey);
    map.put("private", privateKey);
    return map;
}

```

- 硬编码初始化向量：使用初始化向量时，将其硬编码到代码或者配置文件中，语句如下。

```

static String e = "9238513401340235"; // 硬编码方式容易泄漏初始化向量

public static String Encrypt(String src, String key) throws Exception {

```



```

    if (key == null) {
        return null;
    }
    if (key.length() != 16) {
        return null;
    }

    byte[] raw = key.getBytes();
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding"); // “算法/模式/补码方式”
    IvParameterSpec iv = new IvParameterSpec(e.getBytes()); // 使用CBC模式，需要一个向量iv，可增加加密算法的强度
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);
    byte[] encrypted = cipher.doFinal(src.getBytes());
    return Base64.encodeBytes(encrypted); // 此处使用BASE64做转码功能，同时能起到2次加密的作用。
}

```

- RSA 加密时没有使用 padding: 使用 RSA 公钥时通常会绑定一个 padding, 这是因为 no padding 方式的 RSA 更容易遭到攻击, 语句如下。

```

Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding"); // 安全
Cipher cipher = Cipher.getInstance("RSA/ECB/NoPadding"); // 不安全

```

42.7 随机函数使用错误

Java API 中的随机函数类 `SecureRandom` 的随机性是通过设置给它的 `seed` 值来保证的, 但在 Android 4.2 之前的系统上如果输入相同的 `seed` 值, 则会生成重复的随机数, 如下是存在问题的代码。

```

SecureRandom secureRandom = new SecureRandom();
byte[] b = new byte[] { 2, 5, 7, 4 };
secureRandom.setSeed(b);

```

而在 Android 4.2 及之后的系统上，这个漏洞修复了。在 SecureRandom 生成随机数时，如果不调用 setSeed 方法，SecureRandom 会从系统中位于 /dev/urandom 的随机源中获取 seed 值，使用系统生成的这个默认 seed 值可以保证随机数的随机性。

42.8 WebView 忽略 SSL 证书

Android WebView 在加载网页如果遇到证书认证错误时，会回调到 WebViewClient 类的 onReceivedSslError 方法，如下所示是存在漏洞的代码。

```
mWebView.setWebViewClient(new WebViewClient() {  
    @Override  
    public void onReceivedSslError(WebView view, SslErrorHandler handler,  
SslError error) {  
        handler.proceed();  
    }  
});
```

可以看到，回调函数中调用了 handler.proceed() 方法来忽略该证书的错误，这会导致中间人攻击，从而导致用户隐私数据的泄漏。正确的做法是调用默认的 handler.cancel() 方法来终止页面的加载。

42.9 HTTPS 证书弱校验

在实际项目开发中，一般涉及敏感信息的传输我们都会使用 HTTPS 协议来和服务器端进行安全的通信，虽然意图是好的，但往往很多人在实现时对 HTTPS 相关的 Java API 没有全面的掌握，导致代码编写中出现很多安全漏洞，其中以数字证书的使用最为典型，从而导致遭受中间人攻击。漏洞的形式主要有如下三种。

42.9.1 自定义 X509TrustManager 未实现安全校验

使用 HttpURLConnection 发起 HTTPS 请求时，需要提供一个自定义的 X509TrustManager，不过如果实现不当，可能会引入安全问题，如下所示是存在风险的代码。

```

private static class MyTrustManager implements X509TrustManager {

    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
        // 默认接受任意的客户端证书
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
        // 默认接受任意的服务端证书
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
};

```

Android 系统的共享证书机制规定，如果目标 URL 服务器下发的证书不在已信任证书列表中，或者该证书不是权威机构颁发的而是自签名的，那么会出现 `SSLHandshakeException` 异常。正确的做法应该是提供与服务器对应的证书，并进行校验，语句如下。

```

public void init(Context context) throws CertificateException, IOException,
        NoSuchProviderException, NoSuchAlgorithmException,
        KeyManagementException {
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    InputStream caInput = new BufferedInputStream(context.getAssets().
open("ascel885.crt")); // 证书位于assets目录中
    final Certificate ca;
    try {
        ca = cf.generateCertificate(caInput);
    } finally {
        caInput.close();
    }
}

```

```

    }

    MyTrustManager trustManager = new MyTrustManager(ca);
    SSLContext sslContext = SSLContext.getInstance("TLSv1", "AndroidOpenSSL");
    sslContext.init(null, new TrustManager[]{trustManager}, null);
}

private static class MyTrustManager implements X509TrustManager {
    private Certificate mCa;

    public MyTrustManager(Certificate ca) {
        mCa = ca;
    }

    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {
        // 默认接受任意的客户端证书
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {
        if (null == chain || 0 == chain.length) {
            Log.e(TAG, "Certificate chain is invalid.");
        } else if (null == authType || 0 == authType.length()) {
            Log.e(TAG, "Authentication type is invalid.");
        } else {
            for (X509Certificate cert : chain) {
                // 校验证书有效期
                cert.checkValidity();

                // 校验证书的签名
            }
        }
    }
}

```

```

        try {
            cert.verify(mCa.getPublicKey());
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } catch (NoSuchProviderException e) {
            e.printStackTrace();
        } catch (SignatureException e) {
            e.printStackTrace();
        }
    }
}

@Override
public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[0];
}
}

```

42.9.2 自定义 HostnameVerifier 默认接受所有域名

在 SSL 握手期间，如果 URL 对应的主机名和服务器端标识的主机名不匹配，那么 HTTPS 验证机制会回调到 HostnameVerifier 接口的 verify 方法，如果该方法中没有做任何校验而直接返回 true，那么会允许所有域名，从而导致漏洞的存在，以下是漏洞代码。

```

HostnameVerifier hv = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        // 返回 true 表示接受任意域名服务器
        return true;
    }
};

```

```
HttpsURLConnection.setDefaultHostnameVerifier(hv);
```

正确的做法是根据具体的业务需求实现域名匹配规则，只保留符合规则的域名访问，过滤掉可能存在风险的域名，语句如下。

```
HostnameVerifier hv = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        if (isSecureHost()) { // 实现自定义过滤规则
            return true;
        } else {
            HostnameVerifier hv = HttpsURLConnection.
getDefaultHostnameVerifier();
            return hv.verify(hostname, session);
        }
    }
};
```

```
HttpsURLConnection.setDefaultHostnameVerifier(hv);
```

42.9.3 SSLSocketFactory 信任所有证书

信任所有证书极易导致中间人攻击，并被控制整个通信内容，这是业界公认高危漏洞，代码如下。

```
SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

修复这个问题很简单，代码如下。

```
SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

42.10 PendingIntent 使用不当

Intent 是附着在启动它的组件例如 Activity 之上，它是立即执行的。而 PendingIntent 是对 Intent 的封装，用来处理即将要发生的事件，典型的用法是在使用 Notification 发送通知时，PendingIntent 用来封装点击通知后的跳转页面，获取 PendingIntent 实例的方式有如下几种。

```
PendingIntent intent;  
intent = PendingIntent.getBroadcast();  
intent = PendingIntent.getActivity();  
intent = PendingIntent.getService();  
intent = PendingIntent.getActivities()
```

PendingIntent 保存了当前 APP 的 Context，这使得其它 APP 具有跟当前 APP 中一样的权限执行 PendingIntent 封装的 Intent。在使用 PendingIntent 时，其中封装的 Intent 如果是隐式的，那么很容易遭到攻击者嗅探或者劫持，导致 Intent 中的内容泄漏，漏洞如下所示。

```
Intent intent = new Intent( "com.ascel885.action.SHARE" ); // 隐式 Intent  
intent.addFlags(10);  
intent.putExtra( "key", "ascel885" );  
PendingIntent.getBroadcast(this, 0, intent, 0);
```

规避这个漏洞的方案是在 PendingIntent 中使用显式的 Intent。



第7篇 工具篇

- ★ 第 43 章 Android 调试工具 Facebook Stetho
- ★ 第 44 章 内存泄漏检测函数库 LeakCanary
- ★ 第 45 章 基于 Facebook Redex 实现 Android APK 的压缩和优化
- ★ 第 46 章 Android Studio 你所需要知道的功能

第43章

Android调试工具 Facebook Stetho

Stetho¹ 是 Facebook 出品的功能强大的 Android 调试工具，在应用中集成 Stetho 之后，打开 Chrome DevTools 就可以方便地查看 APP 的界面布局、网络请求数据、SQLite 数据库、SharedPreferences 等信息，而且完全不需要对你的手机设备进行 root 操作。

Stetho 的集成非常简单，首先在 Gradle 中添加对 Stetho 函数库的依赖。

```
dependencies {  
    // 这是使用 Stetho 必须引入的核心函数库  
    compile 'com.facebook.stetho:stetho:1.3.1'  
}
```

接着在工程的 Application 类的 onCreate 函数中添加 Stetho 初始化代码。

```
public class MyApplication extends Application {  
    public void onCreate() {  
        super.onCreate();  
        Stetho.initializeWithDefaults(this); // 使用默认值初始化 Stetho  
    }  
}
```

Stetho.Stetho.initializeWithDefaults(this) 等价于如下代码。

```
Stetho.initialize(Stetho.newInitializerBuilder(this)  
    .enableDumpapp(Stetho.defaultDumperPluginsProvider(this))
```

¹ <https://github.com/facebook/stetho>

```
.enableWebKitInspector(Stetho.defaultInspectorModulesProvider(this))
.build();
```

Stetho 的基本配置完成后,就具备了查看数据库,查看 View Hierarchy,使用默认 dumpapp 工具的能力。这时通过打开 Chrome 浏览器,在地址栏中输入 chrome://inspect 打开 DevTools,将手机通过 USB 连接到电脑,打开集成 Stetho 的 APP,即可在 Chrome 中看到如图 43-1 所示的页面。



图 43-1

43.1 视图布局监视

可以看到当前打开的应用的包名,点击旁边的 inspect 按钮即可以打开监视页面(如果是首次打开这个页面,需要首先科学上网,否则看到的是一个空白页面)。

如图 43-2 所示,这个默认的页面就是 APP 对应页面的 View Hierarchy,不仅可以看到 Android Native 页面,也可以看到 HTML5 页面的布局。

43.2 数据库监视

在图 42-3 中,单击 Resourcecs 选项卡,可以看到 APP 中的资源文件,这其中包括数据库表,SharePreference 存储,WebView 的 Session Storage, Cookies 以及应用的缓存等,我们可以对这些值进行读写操作。例如修改 SharePreference 某一项的取值。



图 43-2

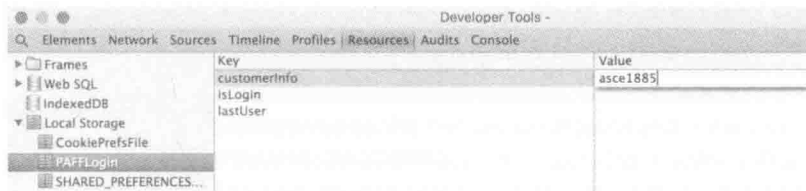


图 42-3

单击数据库名，例如 anydoor.db，可以在右边输入框中直接执行 SQL 语句，如图 42-4 所示。

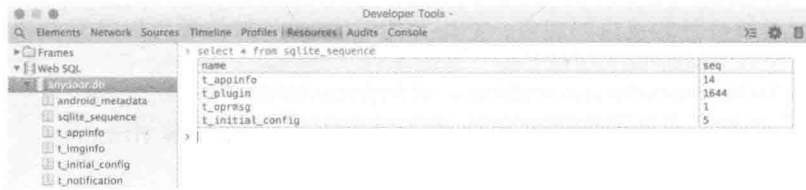


图 42-4

43.3 网络监视

Stetho 的网络访问监视功能可以查看网络请求链接、请求的头部、JSON 数据、请求的响应等信息。使能网络监视，根据项目中使用的网络模块不同，可分为两种方式。

43.3.1 网络模块使用的是 HttpURLConnection

在工程的 Gradle 文件依赖中添加网络拦截相关的函数库 stetho-urlconnection。

```
dependencies {
    compile 'com.facebook.stetho:stetho-urlconnection:1.3.1'
}
```

这个函数库提供了 StethoURLConnectionManager 类实现网络请求和响应的监视，这个类主要提供以下四个函数来完成这一过程。

```
// 调用时机在 HttpURLConnection 已经配置完成
// 但还没有发起实际的网络请求，比如说 connect。
public void preConnect();

// 调用时机在 HttpURLConnection 成功和服务器交换了
// HTTP 消息（请求头部+实体，以及响应头部），但还没有
// 处理响应的实体数据
public void postConnect();

// 负责拦截 HttpURLConnection.getInputStream() 的数据并发送给 Stetho
public InputStream interpretResponseStream();

// 标识在 preConnect 和 interpretResponseStream 这两个生命周期之间发生了不可恢复的
// 错误
public void httpExchangeFailed();
```

下面来看一下 StethoURLConnectionManager 实际使用的例子。

```
private class HttpRequestTask implements Runnable {
    private final HttpRequest request;
    private final Callback callback;
    private final StethoURLConnectionManager stethoManager;

    public HttpRequestTask(HttpRequest request, Callback callback) {
        this.request = request;
        this.callback = callback;
    }
}
```

```

        stethoManager = new StethoURLConnectionManager(request.friendlyName);
    }

    @Override
    public void run() {
        try {
            HttpResponse response = doFetch();
            callback.onResponse(response);
        } catch (IOException e) {
            callback.onFailure(e);
        }
    }

    private HttpResponse doFetch() throws IOException {
        HttpURLConnection conn = configureAndConnectRequest();
        try {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            InputStream rawStream = conn.getInputStream();
            try {
                // Let Stetho see the raw, possibly compressed stream.
                rawStream = stethoManager.interpretResponseStream(rawStream);
                InputStream decompressedStream = applyDecompressionIfApplicable(conn, rawStream);
                if (decompressedStream != null) {
                    copy(decompressedStream, out, new byte[1024]);
                }
            } finally {
                if (rawStream != null) {
                    rawStream.close();
                }
            }
            return new HttpResponse(conn.getResponseCode(), out.toByteArray());
        } finally {
            conn.disconnect();
        }
    }

```

```

    }
}

private HttpURLConnection configureAndConnectRequest() throws IOException
{
    URL url = new URL(request.url);

    // Note that this does not actually create a new connection so it is
    appropriate to
        // defer preConnect until after the HttpURLConnection instance is
    configured. Do not
        // invoke connect, conn.getInputStream, conn.getOutputStream, etc
    before calling
        // preConnect!
    HttpURLConnection conn = (HttpURLConnection)url.openConnection();
    try {
        conn.setReadTimeout(READ_TIMEOUT_MS);
        conn.setConnectTimeout(CONNECT_TIMEOUT_MS);
        conn.setRequestMethod(request.method.toString());

        // Adding this disables transparent gzip compression so that we can
    intercept
        // the raw stream and display the correct response body size.
        requestDecompression(conn);

        SimpleRequestEntity requestEntity = null;
        if (request.body != null) {
            requestEntity = new ByteArrayRequestEntity(request.body);
        }

        stethoManager.preConnect(conn, requestEntity);
        try {
            if (request.method == HttpMethod.POST) {
                if (requestEntity == null) {

```

```

        throw new IllegalStateException( "POST requires an entity" );
    }
    conn.setDoOutput(true);

    requestEntity.writeTo(conn.getOutputStream());
}

    // Ensure that we are connected after this point.  Note that
getOutputStream above will
    // also connect and exchange HTTP messages.
    conn.connect();

    stethoManager.postConnect();

    return conn;
} catch (IOException inner) {
    // This must only be called after preConnect.  Failures before
that cannot be
    // represented since the request has not yet begun according to
Stetho.
    stethoManager.httpExchangeFailed(inner);
    throw inner;
}
} catch (IOException outer) {
    conn.disconnect();
    throw outer;
}
}
}

```

43.3.2 网络模块使用的是 OkHttp

相比较 HTTPURLConnection 的网络拦截, 使用 OkHttp 就显得简单很多, 同样的, 首先需

要在工程的 Gradle 文件依赖中添加网络相关的函数库。

```
dependencies {

    // 如果 OkHttp 版本是 3.x
    compile 'com.facebook.stetho:stetho-okhttp3:1.3.1'

    // 如果 OkHttp 版本是 2.2.x+
    compile 'com.facebook.stetho:stetho-okhttp:1.3.1'
}
```

接着使用如下代码添加 Stetho 拦截器来实现对 APP 网路请求响应的监听。

```
OkHttpClient client = new OkHttpClient();
client.networkInterceptors().add(new StethoInterceptor());
```

// 或者如下代码

```
new OkHttpClient.Builder()
    .addNetworkInterceptor(new StethoInterceptor())
    .build();
```

网络监视页面内容如图 43-5 所示。



图 43-5

43.4 dumpapp

dumpapp 以命令行的方式提供对 APP 的监视功能，该程序位于 stetho 代码路径下的 script/dumpapp 中，目前只支持类 Unix 系统（Linux、Mac OSX），执行 dumpapp 之前，需要确保你的电脑安装了 Python3，如果未安装，可以通过 HomeBrew 方式直接安装。

```
brew install python3
```

安装完成后，在命令行进入 script 目录中执行 .dumpapp -l 可以看到命令行中输出如下内容。

```
crash
files
hprof
prefs
```

这些是 dumpapp 默认提供的插件，用于实现对 APP 的 dump 操作，下面分别进行简单的介绍。

- HprofDumperPlugin: 在 sdcard 上生成 hprof 文件，并将文件的绝对路径显示给开发者。
- SharedPreferencesDumperPlugin: 以命令行方式提供对 SharePreferene 的读写操作。
- CrashDumperPlugin: 模拟 APP Crash，可用于测试代码对 Crash 的处理流程，包含三种不同的 Crash 策略：throw、kill 和 exit。
- FilesDumperPlugin: 提供对 APP 内部存储的文件操作功能，包括查看内部存储文件、文件目录结构，以及导出内部存储的文件到可访问的外部存储上。

dumpapp 的最强大处并不是它默认提供的插件功能，而是它提供的可扩展插件编写能力，开发者可以根据项目的实际需求，开发自己的插件，从而更灵活的操作 APP，插件的编写很简单，下面以一个 HelloWorld 为例进行讲解。

43.4.1 插件的编写

dumpapp 提供了一套插件规范，自定义的插件只需要实现 DumperPlugin 接口并重写该接口提供的 getName 和 dump 方法即可，代码如下。

```
public class HelloWorldDumperPlugin implements DumperPlugin {
    private static final String NAME = "hello";
```

```

@Override
public String getName() {
    return NAME;
}

@Override
public void dump(DumperContext dumpContext) throws DumpException {
    PrintStream writer = dumpContext.getStdout();
    Iterator<String> args = dumpContext.getArgsAsList().iterator();

    String helloToWhom = ArgsHelper.nextOptionalArg(args, null);
    if (helloToWhom != null) {
        doHello(dumpContext.getStdin(), writer, helloToWhom);
    } else {
        doUsage(writer);
    }
}

// 在命令行中根据用户的输入，输出对应的回复
private void doHello(InputStream in, PrintStream writer, String name)
throws DumpException {
    if (TextUtils.isEmpty(name)) {
        // This will print an error to the dumpapp user and cause a non-zero
        exit of the
        // script.
        throw new DumpUsageException( "Name is empty" );
    } else if ( "-" .equals(name)) {
        try {
            name = new BufferedReader(new InputStreamReader(in)).readLine();
        } catch (IOException e) {
            throw new DumpException(e.toString());
        }
    }
}

```

```

        writer.println( "Hello " + name + "!" );
    }

    // 命令用法提示
    private void doUsage(PrintStream writer) {
        writer.println( "Usage: dumpapp " + NAME + " <name>" );
        writer.println();
        writer.println( "If <name> is '-', the name will be read from
stdin." );
    }
}

```

可以看出，HelloWorldDumperPlugin 是一个简单的 echo 程序，它的名字是 hello，这对应着前面调用 ./dumpapp -l 后输出的插件名。

43.4.2 插件的集成

插件编写好了，下一步就是集成到 APP 中，这一步需要修改前面初始化 Stetho 的代码，语句如下。

```

Stetho.initialize(Stetho.newInitializerBuilder(context)
    .enableDumpapp(new DumperPluginsProvider() {
        @Override
        public Iterable<DumperPlugin> get() {
            return new Stetho.DefaultDumperPluginsBuilder(context)
                .provide(new HelloWorldDumperPlugin())
                .finish();
        }
    })
    .build());

```

43.4.3 插件的使用

至此，简单的两步即完成了自定义插件的开发，插件的使用跟默认插件一样，命令行操作

信息如图 43-6 所示。

```
C02PC5C1FVH5:scripts guhaoxin$ ./dumpapp hello
Usage: dumpapp hello <name>

If <name> is '-', the name will be read from stdin.
C02PC5C1FVH5:scripts guhaoxin$ ./dumpapp hello asce1885
Hello asce1885!
```

图 43-6

43.5 Javascript 控制台

开发者可以通过在 Javascript 控制台中执行 Javascript 代码，来和 APP 设置 Android SDK 进行交互。想要使用这个功能，需要在工程的 Gradle 文件依赖中添加如下函数库。

```
compile 'com.facebook.stetho:stetho-js-rhino:1.3.1'
```

之后就可以在控制台中输入 Javascript 代码了，如图 43-7 所示。

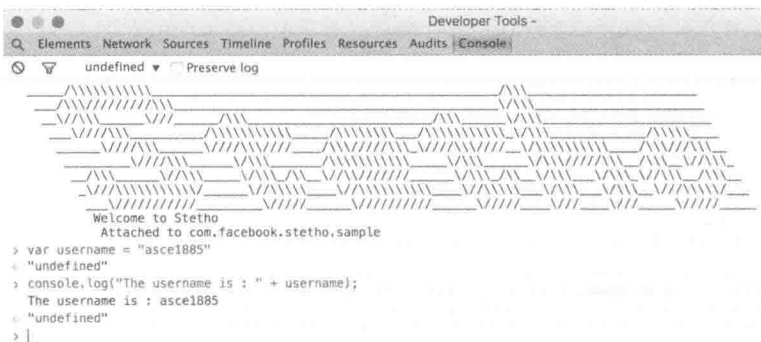


图 43-7

43.6 最佳实践

前面引入 Stetho 相关的函数库的方式是这样的。

```
compile 'com.facebook.stetho:stetho:1.3.1'
compile 'com.facebook.stetho:stetho-urlconnection:1.3.1'
```

```
compile 'com.facebook.stetho:stetho-okhttp3:1.3.1'
compile 'com.facebook.stetho:stetho-js-rhino:1.3.1'
```

但事实上, Stetho 只是用于项目开发过程中方便调试才引入的, 不应该以 `compile` 的方式引入, 否则这些函数库会出现在最终的 Release 包中, 这会显著增加 APP 的体积, 因此, 最佳实践是保证只有调试版本才引入这些函数库, 语句如下。

```
debugCompile 'com.facebook.stetho:stetho:1.3.1'
debugCompile 'com.facebook.stetho:stetho-urlconnection:1.3.1'
debugCompile 'com.facebook.stetho:stetho-okhttp3:1.3.1'
debugCompile 'com.facebook.stetho:stetho-js-rhino:1.3.1'
```

同时, 我们是在应用的 `MyApplication` 类的 `onCreate` 函数中进行 Stetho 的初始化的, 为了不在发布的 APP 中混入这些调试的代码, 需要将 Stetho 初始化操作与线上的代码隔离开来。

首先新建一个 `src/debug/java` 的源文件夹, 专门用来存放调试相关的代码, 而主文件夹 `src/main/java` 则存放 Release 版本的源文件。接着添加一个 Debug 版本的 Application, 它继承自主文件夹的 `MyApplication` 类。

```
public class MyDebugApplication extends MyApplication {
    public void onCreate() {
        super.onCreate();
        Stetho.initializeWithDefaults(this); // 使用默认值初始化 Stetho
    }
}
```

为了实现在 Debug 版本使用的 Application 类是 `MyDebugApplication`, 还需要在 `src/debug` 目录中新建一个 `AndroidManifest.xml` 文件, 文件主体内容如下。

```
<?xml version="1.0" encoding="utf-8" ?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.facebook.stetho.sample" >

    <application
        tools:replace="android:name"
```

```
android:name=".MyDebugApplication" />
```

```
</manifest>
```

Gradle 在构建时,如果是 debug 版本,那么会将 src/debug/AndroidManifest.xml 文件内容合入 src/main/AndroidManifest.xml 文件中,同时由于使用了 tools:replace 属性,所以 android:name 标签的值 MyDebugApplication 会替换 MyApplication。

最终工程的目录结构如图 43-8 所示。



图 43-8

第44章

内存泄漏检测函数库 LeakCanary

LeakCanary¹ 是 Square² 公司开源的一个检测内存泄漏的函数库，可以方便地和你的项目进行集成，在 Debug 版本中监控 Activity、Fragment 等的内存泄漏。使用这个函数库后，Square 修复了很多内存泄漏问题，甚至发现了 Android SDK 中的泄漏，号称解决了自家产品中 94% 的 OOM (OutOfMemoryError) 问题。集成 LeakCanary 到工程之后，在检测到内存泄漏时，会在发送消息到系统通知栏，点击后会打开名为 DisplayLeakActivity 的页面，并显示泄漏的跟踪信息，Logcat 上面也会有对应的日志输出。同时，如果觉得跟踪信息不足以定位时，DisplayLeakActivity 还为开发者默认保存了最近的 7 个 dump 文件到 APP 的目录中，可以使用 MAT 等工具对 dump 文件作进一步分析。DisplayLeakActivity 界面如图 44-1 所示。

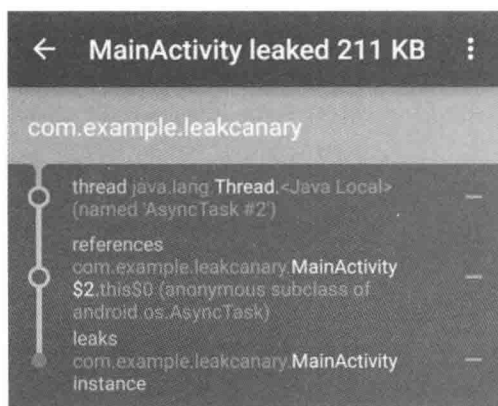


图 44-1

1 <https://github.com/square/leakcanary>

2 <https://corner.squareup.com/>

44.1 基本概念

在继续讲解之前，我们有必要先来温习几个基本概念：

- 垃圾回收：和 Java 一样，Android 也是基于垃圾回收（Garbage Collection，简称 GC）机制实现内存的自动回收的。目前最基本的垃圾回收算法有四种：标记—清除算法（Mark-Sweep）、标记—压缩算法（Mark-Compact）、复制算法（Copying）以及引用计数算法（Reference Counting）。现代流行的垃圾收集算法一般是由这四种中的其中几种算法组合而成。在 Android 虚拟机中，无论是 Dalvik 还是 Art，都是使用标记—清除（Mark-Sweep）算法进行垃圾回收的。
- 内存泄漏：在 Android 中，内存泄漏是指不再使用的对象依然占用内存，或者它们占用的内存没有及时得到释放，从而造成内存空间的不断减少的现象。由于 Android 应用可使用的内存较少，发生内存泄漏会使得内存使用更加紧张，甚至最终由于内存耗尽而发生 OOM（OutOfMemoryError）错误，导致应用崩溃。
- 软引用：使用 SoftReference 关联的对象，用来表示一些有用但不是必需的对象，被 SoftReference 关联的对象，只有在内存不足的时候才会被垃圾回收。
- 弱引用：使用 WeakReference 关联的对象，用来表示非必需的对象，在虚拟机进行垃圾回收时，无论内存是否充足，这类对象都会被回收。
- 引用队列：引用队列 ReferenceQueue 一般是作为 WeakReference（SoftReference）的构造函数参数传入，在 WeakReference（SoftReference）指向的对象被垃圾回收后，ReferenceQueue 就是用来保存这些已经被回收的 Reference。

44.2 LeakCanary 的集成

了解完内存泄漏相关基本概念后，先来看看在项目中如何集成 LeakCanary，如果使用 LeakCanary 默认配置（只检测 Activity 的内存泄漏）的话，集成很简单，只有两个步骤，首先在 build.gradle 文件中添加如下所示的依赖。

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3.1' // or
    1.4-beta1
```

```

        releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1'
// or 1.4-beta1
        testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1' //
or 1.4-beta1
    }

```

可以看到，只有在 Debug 版本中才使用 LeakCanary 的功能，在 Release 和 Test 版本中，使用的是 LeakCanary 的 no-op 版本，也就是没有实际代码和操作的 Wrapper 版本，这样不会对生成的 APK 包体积和应用的性能造成影响。no-op 版本只包含 LeakCanary 和 RefWatcher 这两个类的空实现，代码如下。

```

public final class LeakCanary {
    public static RefWatcher install(Application application) {
        return RefWatcher.DISABLED;
    }

    private LeakCanary() {
        throw new AssertionError();
    }
}

public final class RefWatcher {
    public static final RefWatcher DISABLED = new RefWatcher();

    private RefWatcher() {
    }

    public void watch(Object watchedReference) {
    }

    public void watch(Object watchedReference, String referenceName) {
    }
}

```

接着在工程的 Application 类的 onCreate 函数中进行 LeakCanary 的初始化，这样就完成

LeakCanary 的基本配置。

```
public class ExampleApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        LeakCanary.install(this);
    }
}
```

通过查看 LeakCanary.install 函数的源码，可以发现，这个函数会启动一个 ActivityRefWatcher，它会帮我们自动监控应用中调用 Activity.onDestroy() 之后发生泄漏的 Activity。

```
public static RefWatcher install(Application application,
    Class<? extends AbstractAnalysisResultService> listenerServiceClass,
    ExcludedRefs excludedRefs) {
    if (isInAnalyzerProcess(application)) {
        return RefWatcher.DISABLED;
    }
    enableDisplayLeakActivity(application);
    HeapDump.Listener heapDumpListener =
        new ServiceHeapDumpListener(application, listenerServiceClass);
    RefWatcher refWatcher = androidWatcher(application, heapDumpListener,
excludedRefs);
    ActivityRefWatcher.installOnIcsPlus(application, refWatcher);
    return refWatcher;
}
```

那么如果想监控其他的对象例如 Fragment 呢？这可以通过获取 RefWatcher 的实例来实现，由上面的代码可知，LeakCanary.install() 会返回一个预定义的 RefWatcher 实例，我们需要保存这个实例，并在应用中其他地方使用，一般保存在 Application 类中，语句如下。

```
public class ExampleApplication extends Application {

    public static RefWatcher getRefWatcher(Context context) {
```

```

        ExampleApplication application = (ExampleApplication) context.
getApplicationContext();
        return application.refWatcher;
    }

    private RefWatcher refWatcher;

    @Override public void onCreate() {
        super.onCreate();
        refWatcher = LeakCanary.install(this);
    }
}

```

然后在某个 Fragment 中就可以使用它来实现内存泄漏的监控了，语句如下。如果 Fragment 的 onDestroy 被调用后，这个 Fragment 的实例还没有被销毁，那么就可以在通知栏或者 Logcat 中看到相关的泄漏信息。

```

public abstract class BaseFragment extends Fragment {

    @Override public void onDestroy() {
        super.onDestroy();
        RefWatcher refWatcher = ExampleApplication.
getRefWatcher(getActivity());
        refWatcher.watch(this);
    }
}

```

RefWatch 的 watch 函数源码如下。

```

public void watch(Object watchedReference) {
    watch(watchedReference, "");
}

```

从方法签名可以看出，可以使用 watch 来监控任何你认为应该已经销毁的对象。

44.3 LeakCanary 的原理

根据官方的描述，LeakCanary 的工作原理如下。

- `RefWatcher.watch()` 函数会为被监控的对象创建一个 `KeyedWeakReference` 弱引用对象，`KeyedWeakReference` 是 `WeakReference` 的子类，增加了键值对信息，后面会根据指定的键 `Key` 找到弱引用对象，源码如下。

```
final class KeyedWeakReference extends WeakReference<Object> {
    public final String key;
    public final String name;

    KeyedWeakReference(Object referent, String key, String name,
        ReferenceQueue<Object> referenceQueue) {
        super(checkNotNull(referent, "referent"), checkNotNull(referenceQueue,
            "referenceQueue"));
        this.key = checkNotNull(key, "key");
        this.name = checkNotNull(name, "name");
    }
}
```

- 在后台线程 `AndroidWatchExecutor` 中，检查 `KeyedWeakReference` 弱引用是否已经被清除，如果还存在，则触发一次垃圾回收。垃圾回收之后，如果弱引用对象依然存在，说明发生了内存泄漏，这时 `LeakCanary` 会将 `Heap` 内存导出到 `.hprof` 文件中，这个文件存放在 APP 的文件目录中。

```
/**
 * 调用两次 removeWeaklyReachableReferences 确保内存泄漏检测的准确性
 * @param reference
 * @param watchStartNanoTime
 */
void ensureGone(KeyedWeakReference reference, long watchStartNanoTime) {
    long gcStartNanoTime = System.nanoTime();

    // 计算从调用 watch 函数开始到检测的时间间隔
```

```

        long watchDurationMs = NANoseconds.toMillis(gcStartNanoTime -
watchStartNanoTime);

// 根据引用队列中的被回收弱引用信息，第一次刷新 retainedKeys
removeWeaklyReachableReferences();

// 如果弱引用对象的内存已被回收，或者当前应用处于调试模式，则退出
if (gone(reference) || debuggerControl.isDebuggerAttached()) {
    return;
}

// 调用系统GC，尝试垃圾回收
gcTrigger.runGc();

// 根据引用队列中的被回收弱引用信息，第二次刷新 retainedKeys
removeWeaklyReachableReferences();

// 经过两次确认，到这里可以说明出现内存泄漏了
if (!gone(reference)) {
    long startDumpHeap = System.nanoTime();
    long gcDurationMs = NANoseconds.toMillis(startDumpHeap -
gcStartNanoTime);

    // 将当前 Heap 内容 dump 到文件中
    File heapDumpFile = heapDumper.dumpHeap();

    if (heapDumpFile == HeapDumper.NO_DUMP) {
        return;
    }

    long heapDumpDurationMs = NANoseconds.toMillis(System.nanoTime() -
startDumpHeap);
    heapDumpListener.analyze(
        new HeapDump(heapDumpFile, reference.key, reference.name,
excludedRefs, watchDurationMs, gcDurationMs, heapDumpDurationMs));
}

```

```

    }
}

/**
 * 判断指定的弱引用是否已经被回收
 * @param reference
 * @return
 */
private boolean gone(KeyedWeakReference reference) {
    return !retainedKeys.contains(reference.key);
}

/**
 * 遍历弱引用队列中被回收的引用，并从 retainedKeys 中移除掉
 * 保证 retainedKeys 中存放的是还没有被垃圾回收的弱引用对象的 Key
 */
private void removeWeaklyReachableReferences() {
    KeyedWeakReference ref;
    while ((ref = (KeyedWeakReference) queue.poll()) != null) {
        retainedKeys.remove(ref.key);
    }
}

```

- 接着在一个独立的进程中启动 HeapAnalyzerService 服务，这时 HeapAnalyzer 会使用开源库 [HAHA]¹解析上面的 Heap dump 信息。基于唯一的 reference key，HeapAnalyzer 可以在 heap dump 中找到对应的 KeyedWeakReference，并定位到发生内存泄漏的对象引用。HeapAnalyzer 会计算到 GC Roots 的最短强引用路径，并判断是否存在泄漏，并构建出导致泄漏的对象引用链。

```

/**
 * 根据 referenceKey 在 heap dump 文件中搜索对应的 KeyedWeakReference 对象实例
 * 然后计算这个对象到 GC roots 的最短强引用路径
 * @param heapDumpFile heap dump 文件
 * @param referenceKey 引用 key

```

¹ <https://github.com/square/haha>

```

    * @return
    */
    public AnalysisResult checkForLeak(File heapDumpFile, String referenceKey) {
        long analysisStartNanoTime = System.nanoTime();

        if (!heapDumpFile.exists()) {
            Exception exception = new IllegalArgumentException("File does not exist: " + heapDumpFile);
            return failure(exception, since(analysisStartNanoTime));
        }

        try {
            // 使用 HAHA 开源库分析 heap dump 文件
            HprofBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);
            HprofParser parser = new HprofParser(buffer);
            Snapshot snapshot = parser.parse();
            deduplicateGcRoots(snapshot);

            // 根据引用 key 查找对应的弱引用对象
            Instance leakingRef = findLeakingReference(referenceKey, snapshot);

            // 如果为空, 说明这个弱引用对象在之前的检查 key 和 heap dump 操作之间被回收
            if (leakingRef == null) {
                return noLeak(since(analysisStartNanoTime));
            }

            // 构建出导致泄漏的对象引用链
            return findLeakTrace(analysisStartNanoTime, snapshot, leakingRef);
        } catch (Throwable e) {
            return failure(e, since(analysisStartNanoTime));
        }
    }
}

```

-
- 内存泄漏检测的结果会传递给位于 APP 主进程中的 DisplayLeakService 服务, 由这个服

务来发出泄漏的通知栏消息。

44.4 LeakCanary 的定制

LeakCanary 提供了良好的接口供使用者进行自定义操作，主要有以下这些方面。

44.4.1 RefWatcher 的自定义

在前面介绍过，leakcanary-android-no-op 依赖库是用在 Release 版本的，它包含了 LeakCanary 和 RefWatch 这两个类的空实现，如果要自定义 LeakCanary，需要确保自定义部分代码只影响到 Debug 版本，因为有可能会引用到 leakcanary-android-no-op 依赖库中没有的 API，从而导致发布版本出错。正确的做法是将 Release 和 Debug 部分的代码分离，例如定义 ExampleApplication 用于 Release 版本，DebugExampleApplication 用于 Debug 版本，它继承自 ExampleApplication，这两个类的代码如下。

```
// Release 版本的 Application 类
public class ExampleApplication extends Application {
    public static RefWatcher getRefWatcher(Context context) {
        ExampleApplication application = (ExampleApplication) context.
getApplicationContext();
        return application.refWatcher;
    }

    private RefWatcher refWatcher;

    @Override public void onCreate() {
        super.onCreate();
        refWatcher = installLeakCanary();
    }

    protected RefWatcher installLeakCanary() {
        return RefWatcher.DISABLED;
    }
}
```

```

}

// Debug 版本的 Application 类
public class DebugExampleApplication extends ExampleApplication {
    protected RefWatcher installLeakCanary() {
        RefWatcher refWatcher = ? // Build a customized RefWatcher
        return refWatcher;
    }
}

```

这两个类及其对应的 AndroidManifest.xml 文件要放在不同的目录中，具体可以参考本书第 43 章。

44.4.2 通知页面样式的自定义

内存泄漏通知页面 DisplayLeakActivity 默认的图标和标签这两个值，可以通过在 APP 中提供对应的值进行覆盖，图标的定义如下。

```

res/
drawable-hdpi/
    __leak_canary_icon.png
drawable-mdpi/
    __leak_canary_icon.png
drawable-xhdpi/
    __leak_canary_icon.png
drawable-xxhdpi/
    __leak_canary_icon.png
drawable-xxxhdpi/
    __leak_canary_icon.png

```

标签的自定义如下。

```

<?xml version=" 1.0" encoding=" utf-8" ?>
<resources>
    <string name=" __leak_canary_display_activity_label" >MyLeaks</string>
</resources>

```

44.4.3 内存泄漏堆栈信息保存个数的自定义

默认情况下, DisplayLeakActivity 在 APP 目录中最多保存 7 个 HeapDump 文件和泄漏的堆栈信息 (Leak Trace), 可以通过在 APP 中定义 R.integer.__leak_canary_max_stored_leaks 来进行修改, 语句如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
    <integer name="__leak_canary_max_stored_leaks">20</integer>
</resources>
```

44.4.4 Watcher 的延时

在 LeakCanary 1.4.* 版本开始, 可以通过定义 R.integer.leak_canary_watch_delay_millis 来修改弱引用对象被认为出现内存泄漏的延时时间, 默认是 5 秒, 如下可修改为更加严格的 1.5 秒。

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
    <integer name="leak_canary_watch_delay_millis">1500</integer>
</resources>
```

44.4.5 自定义内存泄漏堆栈信息和 heap dump 的处理方式

默认情况下, 通过 DisplayLeakActivity 可以查看和分享内存泄漏堆栈信息和 HeapDump 文件, 但可以通过继承 DisplayLeakService 并重写其中的 afterDefaultHandling 函数来实现定制化的操作。例如发送到服务端, 甚至上传到 Slack 或者 HipChat, 语句如下。

```
public class LeakUploadService extends DisplayLeakService {
    @Override
    protected void afterDefaultHandling(HeapDump heapDump, AnalysisResult
result, String leakInfo) {
        if (!result.leakFound || result.excludedLeak) {
            return;
        }
    }
}
```

```

        myServer.uploadLeakBlocking(heapDump.heapDumpFile, leakInfo);
    }
}

```

接着需要修改 LeakCanary 的初始化代码，语句如下。

```

public class DebugExampleApplication extends ExampleApplication {
    protected RefWatcher installLeakCanary() {
        return LeakCanary.install(app, LeakUploadService.class,
        AndroidExcludedRefs.createAppDefaults().build());
    }
}

```

最后为了使 LeakUploadService 生效，不要忘了在 AndroidManifest.xml 文件中进行注册。

44.4.6 忽略特定的弱引用

通过实现自己的 ExcludedRefs，我们可以实现对某些特定弱引用对象的忽略，不对其进行内存泄漏的监视，代码如下。

```

public class DebugExampleApplication extends ExampleApplication {
    protected RefWatcher installLeakCanary() {
        ExcludedRefs excludedRefs = AndroidExcludedRefs.createAppDefaults()
        .instanceField("com.example.ExampleClass", "exampleField")
        .build();
        return LeakCanary.install(this, DisplayLeakService.class,
        excludedRefs);
    }
}

```

44.4.7 不监视特定的 Activity 类

LeakCanary 的 install 函数默认会安装 ActivityRefWatcher，从而实现对应用中所有 Activity 内存泄漏的监视，但实际开发中，往往存在某些需要排除在监视范围之外的 Activity，这可以通过自定义安装过程实现。

```

public class DebugExampleApplication extends ExampleApplication {
    @Override
    protected RefWatcher installLeakCanary() {
        if (LeakCanary.isInAnalyzerProcess(this)) {
            return RefWatcher.DISABLED;
        } else {
            ExcludedRefs excludedRefs = AndroidExcludedRefs.
createAppDefaults().build();
            LeakCanary.enableDisplayLeakActivity(this);
            ServiceHeapDumpListener heapDumpListener = new
ServiceHeapDumpListener(this, DisplayLeakService.class);
            final RefWatcher refWatcher = LeakCanary.androidWatcher(this,
heapDumpListener, excludedRefs);
            registerActivityLifecycleCallbacks(new
ActivityLifecycleCallbacks() {
                public void onActivityDestroyed(Activity activity) {
                    if (activity instanceof ThirdPartyActivity) {
                        return;
                    }
                    refWatcher.watch(activity);
                }
                // ...
            });
            return refWatcher;
        }
    }
}

```

LeakCanary 对 Activity 的监听默认只支持 Android 4.0 及以上系统，因为 ActivityRefWatcher 在监听每个 Activity 时需使用 Application 类中的 ActivityLifecycleCallbacks，这个回调接口是在 4.0 系统上开始引入的；如果想在 4.0 以下系统使用 LeakCanary 监听 Activity 的内存泄漏，可以在应用的 Activity 基类的 onDestroy 函数中手动添加对 Activity 的监听，类似之前说过的 Fragment 方式。

第45章

基于Facebook Redex实现Android APK的压缩和优化

最近 Facebook 开源了一个名为 Redex¹ 的工具包，专门用于 Android 字节码的优化，经过 Redex 转换后的 APK，体积变得更小，运行速度变得更快。Redex 基于管道的方式来优化 Android 的 .dex 文件，一个源 .dex 文件通过管道进行一系列的自定义转换后，将得到一个优化的 .dex 文件。接下来将带大家简单快速地了解 Redex 是什么，以及它的基本原理和使用方法。

45.1 转换的时机

我们知道 Android 的编译过程首先是通过 javac 工具将 .java 文件编译成 .class 文件，接着将所有的 .class 文件合并成 Dalvik 虚拟机的可执行文件 .dex，最后再跟其他资源等文件一起压缩成 APK 文件，大致流程如图 45-1 所示。

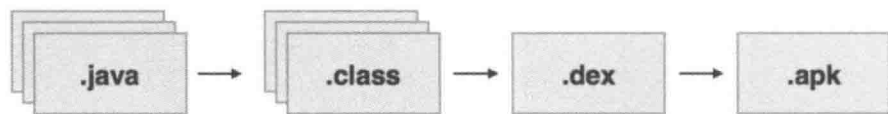


图45-1

Redex 选择基于字节码文件而不是 Java 源码进行优化，是因为字节码相比 Java 源码而言，可以进行更为全局的、类与类之间的优化，而不是单个类文件的局部优化；选择基于 dex 字节码而不是 Java 字节码进行优化，是因为某些优化只能在 .dex 文件中进行。

¹ <https://github.com/facebook/redex>

45.2 管道的思想

鉴于随着时间的推移,开发人员可能会不断得到新的优化 idea,为了方便地将新的优化点加入既有代码中,同时也方便不同开发人员并行开发优化点。所以 Redex 选择基于管道的思想来实现 dex 的优化,这样每一个优化的 idea 可以通过插件的形式集成到管道中,实现即插即用,同事不会影响其他的优化插件,整体优化流程如图 45-2 所示。



图 45-2

45.3 减少字节码的意义

减少 APP 中字节码的大小有很多好处,最明显的是可以减少 APP 下载和安装的时间,也可以减少 APP 安装后占用的存储空间。同时,理论上更少的字节码也意味着需要执行的指令更少,需要加载进内存的代码页发生缺页的情况也更少,这些显然对于资源密集型使用场景(例如应用冷启动)起到了很好的性能优化作用。在 Redex 中,基于管道的思想实现了一系列旨在减少和优化字节码的转换器,下面来了解其中的三种。

45.4 混淆和压缩

代码混淆广泛应用于 Web 开发语言(例如 Javascript 和 HTML)中,在不改变功能的前提下,通过使用无意义和简短的字符来替换完整的字符串信息,从而减少总的代码大小,Android 中使用 Proguard 也是为了达到类似的目的。

在开发阶段,代码中可读性强的字符串信息是非常重要的,例如类的完整路径,源文件的路径,函数名称等。但对于编译后的字节码而言,这些完整的字符串信息占用了较大的空间,更重要的是,虚拟机运行字节码的时候并不关心这些字符串信息是否可读性强,abc() 和 MyFooSuccessCallback() 这两者对虚拟机的处理来说并无区别。因此,我们就可以将字节码中可

读性强但占用空间的字符串替换成无意义但简短的字符串，如图 45-3 所示。

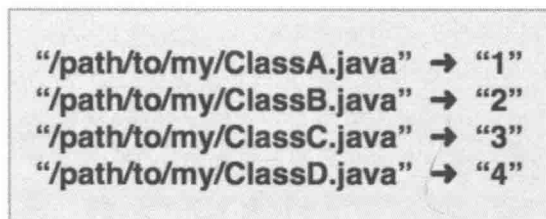


图 45-3

跟 Java 层代码使用 Proguard 混淆后需要生成 mapping.txt 文件类似，字节码的混淆也需要生成对应的映射文件，以便在 APP 运行过程中出现问题需要定位的时候，可以将混淆后的日志信息通过映射文件还原成可读的字符串信息，映射文件内容类似这样。

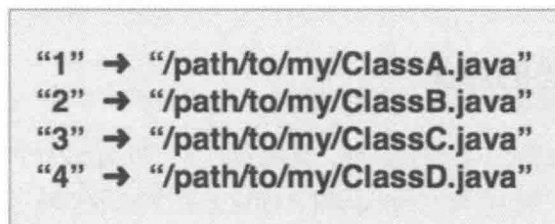


图 45-4

45.5 使用内联函数

内联函数是在编译期间将函数体直接嵌入该函数的调用处，也就是在编译时不具备函数的性质，不存在执行函数调用产生的开销，从而得到提高代码运行性能的目的。同时，如果正确地应用它，还可以减小编译后生成的文件大小。软件工程的最佳实践鼓励开发者要具备封装的思想，要明确类的职责，这样往往会导致需要对一个类按功能和职责进行拆分等操作。在实际开发中，这种思想是很重要的，但同时最终生成的字节码中也留有进一步优化的空间。

最简单的一个例子是适配器类型的函数，这些函数通常是用来封装一些小函数，以提供更简洁统一的 API 接口，或者是由于参数列表不同而存在的多个重载函数，亦或是 setter/getter 函数等，这些函数在最终生成的 APK 中有的可能根本不会被调用到。因此在 .dex 文件阶段对

这些函数进行内联操作，是很大的一个优化点。

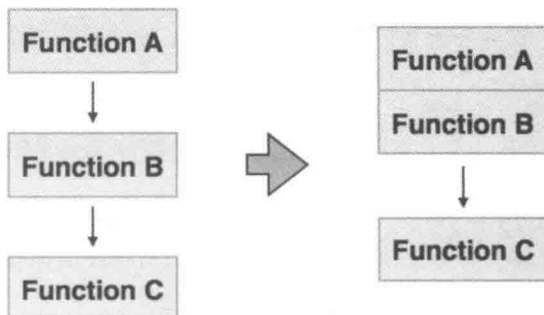


图 45-5

45.6 无用代码的消除

在大型项目的源代码中，不可避免地会存在很多无用的代码，移除这些无用代码在减少最终 APK 的大小的同时也不会带来任何副作用。在某些方面，无用代码的移除类似于标记清除（Mark-Sweep）垃圾回收算法。我们从某个明确会调用的入口，例如 MainActivity 开始遍历各个条件分支和函数调用，在生成的图中标记访问到的代码，在遍历了所有的条件分支和函数调用之后，就可以判定那些没有被标记的函数是无用的代码，可以安全的删除，如图 45-6 所示。

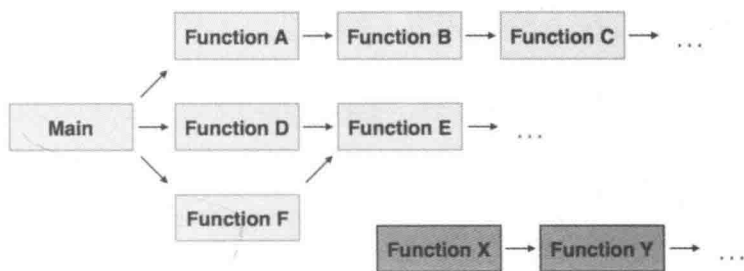


图 45-6

当然，理论上很简单，但在 Android 中还需要处理类似反射，或者 XML 布局文件中对代码的引用等异常情况。

45.7 Redex 的集成和使用

在使用 Redex 之前，首先要配置好编译环境，Redex 目前支持 Mac OS X 和 Linux 系统，下面以 Mac OS X 为例说明。

45.7.1 依赖的安装

打开 Terminal 窗口，执行以下的 HomeBrew 命令安装 Redex 的依赖。

```
brew install autoconf automake libtool python3
brew install boost double-conversion gflags glog libevent
```

45.7.2 下载，构建和安装

Redex 的依赖成功安装后，接着使用 Git 将 Redex 的源码 checkout 到电脑上，由于 Redex 是以子模块的方式引入 folly¹ 的，因此需要执行如下命令初始化子模块。

```
git submodule update --init
```

最后，通过 autoreconf 和 make 命令来编译 Redex。

```
autoreconf -ivf && ./configure && make && make install
```

成功后，就可以开始使用 Redex 来转换现有的 APK 文件中的 .dex 文件了。

45.7.3 使用

Redex 的使用很简单，如下所示，只需指定源 APK 和 生成的 APK 的路径就可以。

```
redex path/to/your.apk -o path/to/output.apk
```

¹ <https://github.com/facebook/folly>

第46章

Android Studio你所需要知道的功能

“工欲善其事，必先利其器”，Android 开发中 Android Studio 已经是事实上的标准开发工具，相信你对其常见的用法已经非常熟悉，本章介绍的几个 Android Studio 的功能或者插件也许能够进一步提高你的工作效率。

46.1 Annotate

在团队协作开发过程中，很多时候需要知道某行代码最近一次是谁修改的，因为什么原因而修改，我们当然可以选择在团队即时通信群中询问，或者查看这个文件的版本提交记录，但这样通常非常耗时，群里可能过了一天都没人会响应你的提问，而如果这行代码是很久以前修改的，可能需要查阅到很久以前的一次提交。一个好的方法是使用 Android Studio 提供的 Annotate 功能（当然，前提是你的团队使用 Git），如图 46-1 所示，用鼠标右键单击代码编辑框左边栏，在弹出的菜单中就能看到 Annotate。

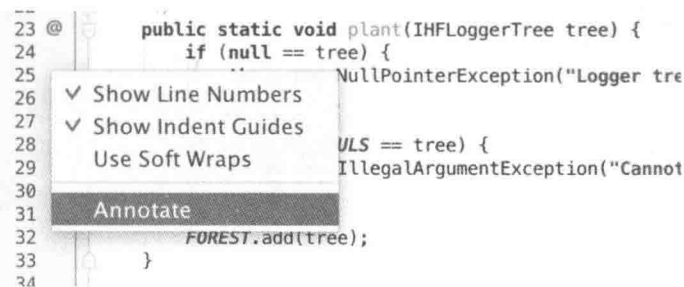


图46-1

单击 Annotate 按钮，首次使用时会弹出对话框要求输入这段代码所在的版本控制系统的用户名和密码，输入成功后，会显示出如图 46-2 所示的结果，可以看到，谁最后改动了哪一行代码一目了然，其中包括代码提交的日期，commit id 和作者名字，将鼠标漂浮在对应的行上面，还会出现这次提交的简介信息。



图 46-2

单击对应的行，还会弹出包含这次提交更详细信息（提交涉及的所有文件）的对话框。

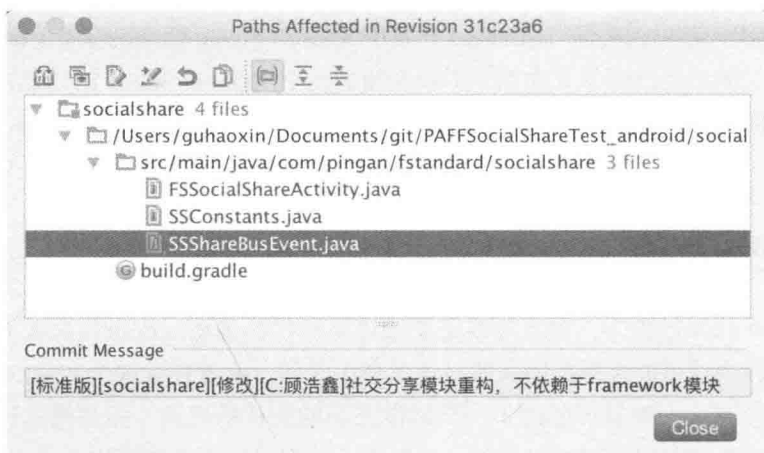


图 46-3

46.2 .ignore 插件

.ignore 插件可以方便的生成不同版本控制系统中需要忽略的文件，这样可以在提交代码的时候得到一个干净的文件修改记录，在 Android Studio 的插件页面中搜索 .ignore 并安装即可。完成后右键 Android 工程，选择 New → .ignore file 选择你使用的版本控制系统即可生成对应的 .ignore 文件，如图 46-4 所示。

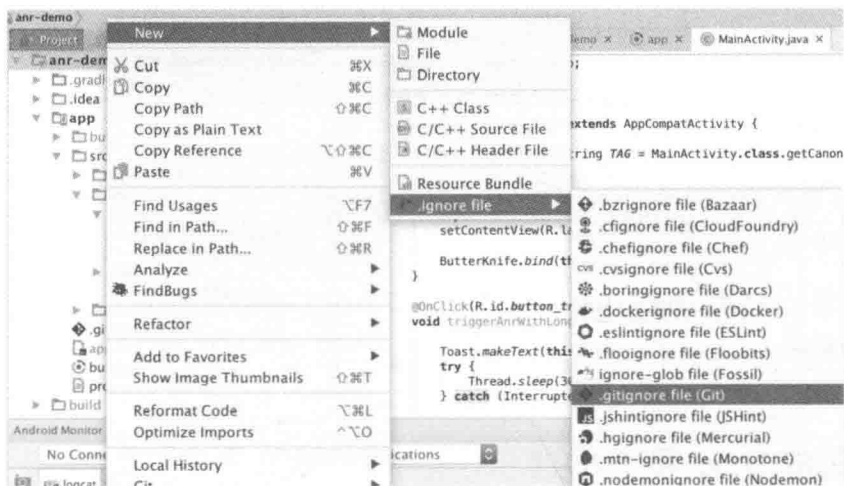


图 46-4

46.3 Live Templates

Android 开发中经常需要会需要敲一些样板代码，例如 `findViewById`、`Toast.makeText` 等，每次手动输入这些样板代码只是浪费时间而已，一个好的办法是使用缩略词来输入，并让 Android Studio 自动生成，这是 Live Templates 提供的能力了。在 Android Studio 的 Preferences → Editor → Live Templates 中可以看到 IDE 默认为开发者提供的配置，如图 46-5 所示。

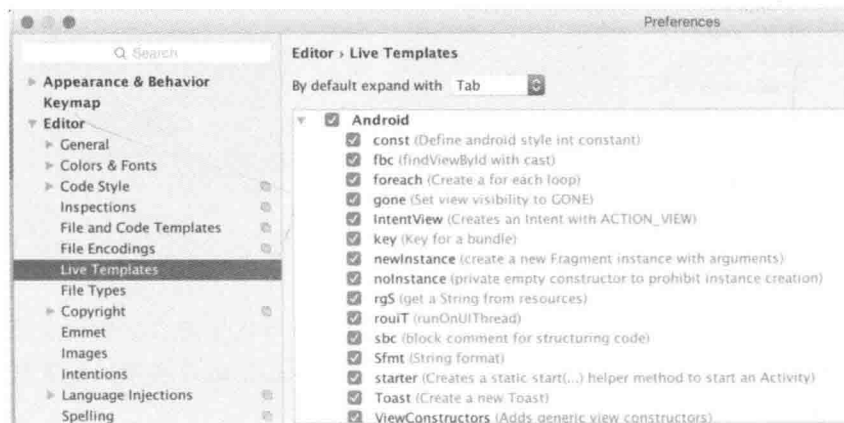


图 46-5

使用方法很简单，输入前面的缩略词，然后单击 `tab` 键，IDE 就会自动帮我们扩展出完整的代码，例如，输入 `fb` + `tab` 键，IDE 帮忙补全如下。

```
() findViewById(R.id.);
```

输入 `Toast` + `tab` 键，IDE 帮忙补全如下。

```
Toast.makeText(MainActivity.this, "", Toast.LENGTH_SHORT).show();
```

我们只需要将空白的地方填空就行。除了使用系统默认提供的规则，我们也可以非常方便的自定义缩略词。

46.4 集成 Bug 管理系统

Android Studio 中可以轻松集成各种常见的 Bug 管理系统，点击 `Tools-Tasks & Contexts-Configure Servers`，弹出如图 46-6 所示的 Bug 管理服务对话框。

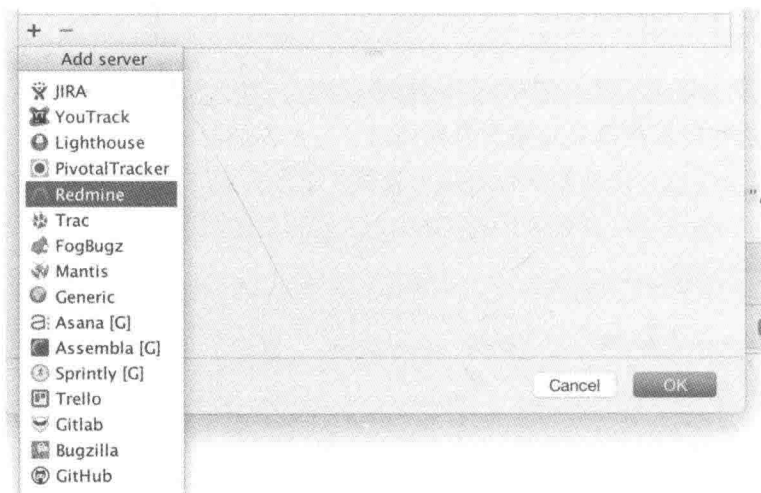


图46-6

可以看到，默认支持 JIRA、Redmine、Gitlab、GitHub 等常见的 Bug 管理系统，可以根据团队具体使用的系统进行选择，例如选择 Redmine，弹出如图 46-7 所示的配置页面，需要配置 Bug 系统的地址、你的用户名和密码等信息。

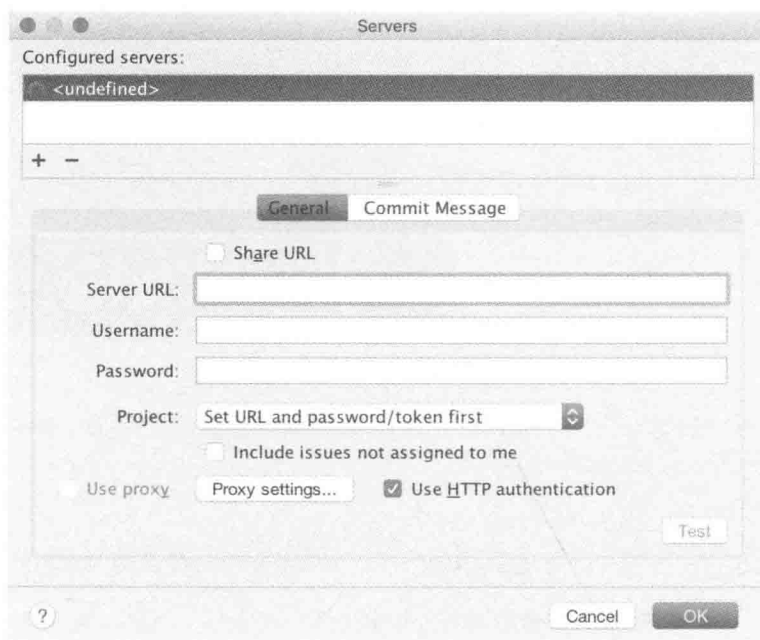


图 46-7



第8篇 测试篇

- ★ 第 47 章 Android 单元测试框架简介
- ★ 第 48 章 Android UI 自动化测试框架简介
- ★ 第 49 章 Android 静态代码分析实战
- ★ 第 50 章 基于 Jenkins + Gradle 搭建 Android 持续集成编译环境

第47章

Android单元测试框架简介

单元测试（又称为模块测试，Unit Testing）是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。——维基百科

首先需要明确的是，单元测试是开发人员而不是测试人员的职责，在国外流行的编程实践测试驱动开发（Test Driven Develop，TDD）中，要求开发人员先写单元测试，然后再开始编写代码，编码的代码要保证能够通过单元测试的验证，当所有的单元测试用例通过之后，代码也就编写完成了。

虽然单元测试有着种种的好处，但这一编程方式在国内并没有很好的流行起来，原因有很多，首先国内大多数开发人员没有编写单元测试的意识，也没有进行这方面的正确实践，导致对单元测试没有正确完整的认识；其次移动开发中，由于产品需求的不断变化和版本的快速迭代，开发人员基本上除了加班加点赶需求，并没有多余的时间去编写单元测试用例；如果碰上产品经常性改版，那么之前编写的单元测试基本上也需要重写。

测试驱动开发的利弊和实际可操作性我们不作深入的讨论，但单元测试至少对于一个基础库是必不可少的，基础库的好处是不会经常发生太大的变化，单元测试有助于及早发现修改和重构引入的 bug，从而保证基础库的稳定和可用。

47.1 Java 单元测试框架 JUnit¹

JUnit 是使用最广泛，也是最基础的一个 Java 语言单元测试框架，使用 Android Studio 的模

¹ <http://www.junit.org/>

版创建一个应用工程时，默认会生成单元测试目录，同时会给出一个使用 JUnit 的简单例子，如图 47-1 所示，src/test 目录就是 Java 单元测试目录，它里面的包目录结构和 src/main 类似。



图 47-1

其中 ExampleUnitTest 内容如下。

```
import org.junit.Test;
import static org.junit.Assert.*;

public class ExampleUnitTest {
    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }
}
```

可以看到：

- 单元测试类的命名一般以 Test 结尾。
- 单元测试方法需要使用 @Test 注解标记。
- 验证单元测试的结果一般使用 Assert 类中的方法，例如 assertEquals、assertTrue、assertNotEquals、fail 等。

在使用 JUnit 提供的 API 之前，我们需要在 build.gradle 文件中加入如下依赖。

```
dependencies {
    testCompile 'junit:junit:4.12'
}
```

在JUnit中，除了@Test，常用的注解还有：

- @Before：表示在每个被@Test标记的测试方法调用之前，都要先调用被@Before标记的方法。
- @After：表示在每个被@Test标记的测试方法调用之后，都要接着调用被@After标记的方法。
- @BeforeClass：在执行一个测试类的所有测试方法之前，会调用一次这个测试类中被@BeforeClass标记的方法。
- @AfterClass：执行完一个测试类中所有的测试方法之后，会调用一次这个测试类中被@AfterClass标记的方法。
- @Ignore：测试类中某些测试方法可能暂时处于不可测试阶段，那么为了保证不影响整个测试类的正常运行，可以使用@Ignore标记这些测试方法。

47.2 Android 单元测试框架 Robolectric 3.0¹

Robolectric 是 Android 平台上最好用的单元测试框架，它的设计思想是通过实现一套 JVM 能够运行的 Android 代码，从而实现脱离 Android 环境进行测试。Robolectric 的最新版本是 3.0，在 Android Studio 中集成很简单，首先在 Build.gradle 文件中引入依赖，由于 Robolectric 依赖于 JUnit，因此也需要一同引入。

```
testCompile "junit:junit:4.10"
testCompile "org.robolectric:robolectric:3.0"
```

然后在单元测试类中使用注解和 RobolectricGradleTestRunner 关联起来。

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class)
public class SandwichTest {
```

¹ <http://robolectric.org/getting-started/>

}

Robolectric 的工作目录和 JUnit 一样，都是 src/test 目录。使用 Robolectric 可以方便地对 Android 中的 Activity、Fragment、Service、BroadcastReceiver 等组件进行单元测试，对 Activity 的测试例子如下，titleIsCorrect 方法通过判断 Activity 的标题是否正确来判断 Activity 是否正常启动。

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class)
public class MainActivityTest {

    @Test
    public void titleIsCorrect() throws Exception {
        Activity activity = Robolectric.setupActivity(MainActivity.class);
        assertTrue(activity.getTitle().toString().equals("Deckard"));
    }
}
```

当然，还可以测试 Activity 的生命周期，然后通过判断特定生命周期执行前后某些状态量的变化来判断是否正常运行，下面的例子通过判断一个 TextView 变量取值是否正确来决定是否通过测试。

```
@Test
public void testLifecycle() {
    ActivityController<MainActivity> activityController = Robolectric.
buildActivity(MainActivity.class).create().start();
    Activity activity = activityController.get();
    TextView textview = (TextView) activity.findViewById(R.id.tv_lifecycle_
value);
    assertEquals("onCreate", textview.getText().toString());
    activityController.resume();
    assertEquals("onResume", textview.getText().toString());
}
```

更多的使用例子可以参见官方文档，Robolectric 不仅支持对 UI 组件的测试，同时还支持像日志输出、网络请求、数据库等方面的测试。

47.3 Java 模拟测试框架 Mockito¹

在单元测试过程中，往往存在一些很难生成或者根本不可能生成的对象，为了正常进行测试，需要通过创建一个虚假的对象来模拟这些真实的对象，从而达到正常测试的目的。Mockito 是一个强大的用于 Java 开发的模拟测试框架，在使用之前，首先引入函数库。

```
repositories {  
    jcenter()  
}  
  
dependencies {  
    testCompile "org.mockito:mockito-core:1.+"  
}
```

接着就可以开始使用 Mockito 的强大功能了。

47.3.1 行为的验证

行为的验证用来检测某个 Mock 对象是否执行了某个操作，例如往数组里面添加数据项。Mock 对象的创建通过 Mock 方法或者 @Mock 注解实现，创建完成之后，就可以像使用原对象那样使用 Mock 对象，最后通过 verify 方法验证对应的操作是否执行过，内容如下。

```
import static org.mockito.Mockito.*;  
  
// 创建接口 List 的 mock 对象  
List mockedList = mock(List.class);  
  
// 使用 mock 对象执行操作  
mockedList.add("one");  
mockedList.clear();  
  
// 验证对应的操作是否执行过  
verify(mockedList).add("one");  
verify(mockedList).clear();
```

¹ <http://mockito.org/>

47.3.2 Stub（桩函数）的使用

Stub 和 Mock 都是用来模拟系统外部依赖的，Mock 侧重于判断测试是通过还是失败，而 Stub 则是用来完全模拟外部依赖，Mockito 中使用 Stub 的例子如下。

```
// 创建类 LinkedList 的 mock 对象
LinkedList mockedList = mock(LinkedList.class);

// 在 mock 对象正在执行某个操作之前，插入桩函数
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

// 打印出 "first"
System.out.println(mockedList.get(0));

// 抛出运行时异常
System.out.println(mockedList.get(1));

// 由于 mock 对象没有对 get(999) 操作进行插桩，因此返回 "null"
System.out.println(mockedList.get(999));
```

Mockito 的功能非常丰富，除了上面的 Mock 和 Stub，它还支持 Mock 参数匹配、对真实的对象执行 Spy 操作等，具体可以参见官方文档¹。

¹ <http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html>

第48章

Android UI自动化测试 框架简介

自动化测试是解放测试人员双手，将之前需要人为执行的测试操作转变为机器自动执行的过程。自动化测试能够让测试人员将更多的精力投入到更重要的事情中，而不是做一些重复的测试。自动化测试不仅减轻了测试人员的负担，还提高了测试效率。

如果团队中开发和测试分工明确的话，作为 Android 开发人员可能平时接触自动化测试的机会并不多，但起码的了解还是应该具备的。Android 发展到今天，涌现了很多自动化测试框架，常见的有 Monkey、MonkeyTalk、Robotium、UIAutomator、Appium 等，下面分别做一个简单的介绍。

48.1 Monkey

Monkey 是 Android SDK 自带的命令行工具，它可以运行在模拟器或者真实设备中。它的基本原理是通过向系统发送伪随机的用户事件流从而实现对应应用进行压力测试的目的，Monkey 测试是一种为了测试应用稳定性和健壮性的快速有效的方法。测试完成后会输出日志信息帮助开发者定位问题，Monkey 测试是发现应用中可能存在 ANR 的一个手段。Monkey 测试有三个基本特点。

- 只能对应用的 APK 包进行测试，存在一定的局限性。
- 测试使用的事件流数据是随机生成的，不能自定义。
- 可以针对 Monkey 测试的对象，事件的数量，类型，频率等进行设置。

配置好 Android SDK 的环境变量后，在命令行中执行 `adb shell monkey` 即可启动 Monkey。

48.2 MonkeyRunner

MonkeyRunner 也是 Android SDK 提供的工具，相比 Monkey 而言，它是一个 API 工具包，使用 MonkeyRunner 提供的 API 可以实现在 Android 代码之外定义特定命令和事件控制 Android 真实设备和模拟器。MonkeyRunner 的主要设计目的是用来测试功能以及框架水平上的应用程序和设备，或者用来运行单元测试套件等。

MonkeyRunner 为 Android 的测试提供了以下特性。

- 多设备控制：MonkeyRunner API 可以跨越不同的设备和模拟器实施测试套件，我们可以一次连接上所有的真实设备或者模拟器，然后执行一个或者多个测试。
- 功能测试：MonkeyRunner 可以为应用自动执行一次功能测试，测试者只需要提供触摸等事件的输入值，然后观察输出结果的截屏即可。
- 回归测试：MonkeyRunner 可以运行某个应用，并将其结果截屏后与已知的正确结果的截屏进行对比，从而对应用进行功能的回归。
- 可扩展的自动化：MonkeyRunner 是一个 API 工具包，测试者可以使用 Python 模块和程序开发一套系统来控制 Android 设备，MonkeyRunner 使用 Jython，从而实现 MonkeyRunner API 与 Android 框架轻松地进行交互。

值得一提的是 Monkey 和 MonkeyRunner 都是基于坐标点的测试框架。

48.3 UIAutomator

UIAutomator 是 Android 4.1 发布的 Android UI 自动化测试工具，基本上支持所有 Android 事件操作，例如点击、滑动、文本输入等，它不需要使用者了解代码的实现细节。UIAutomator 的缺点要求 Android 版本高于 4.0，而且在 4.3 以下的系统无法根据控件 ID 进行操作，对 Web 页面只支持根据坐标进行点击操作。

Android SDK 4.1 中提供了以下工具来实现 UI 自动化测试。

- `uiautomatorviewer`：一个图形界面工具用来扫描和分析应用中的 UI 控件。
- `uiautomator`：一个用于实现自动化测试的函数库，包含创建 UI 测试所需的各种 API 和执行自动化测试的引擎，这个函数库的使用需要满足两个条件。

❑ Android SDK Tools, Revision 21 or higher。

❑ Android SDK Platform, API 16 or higher。

48.4 Robotium¹

Robotium 是一款开源的针对 Android 平台的自动化测试框架，它主要是对应用进行黑盒测试，提供的 API 封装了很多便捷的方法，使得开发者可以模拟各种手势操作，例如点击、长按、滑动、给控件赋值等，同时提供了查找和断言等 API 接口，方便对应用的控件进行各种操作。

Robotium 既可以针对源码进行测试，也可以针对 APK 文件进行测试，不过在对 APK 进行测试之前，需要确保被测试的 APK 和测试项目具有相同的签名。

UIAutomator 和 Robotium 都是基于控件的测试框架。

48.5 Espresso²

Espresso 是 Google 官方在 2013 年推出的 Android 自动化测试框架，相对于其他工具，Espresso 的 API 更加精确，并且规模更小更简洁，容易学习。它的目标是让开发者写出更简洁的针对 Android 应用的 UI 测试代码。Espresso 集成在 Testing Support Library³ 中，可以很方便地引入。

48.6 Appium⁴

Appium 是目前功能最强大且最受开发者喜爱的移动端自动化框架，它支持 Native app、Hybrid app 和 Web app 的测试，同时支持 Android 和 iOS 平台，Appium 可以非常快捷地创建自动化测试用例。由于 Appium 基于 Selenium 的 client 库，因此它可以使用多种语言编写测试用例，包括 Java、Objective-C、Python、Ruby、NodeJs 等，开发者可以自由选择自己擅长的语言。

Appium 的缺点是环境部署相比其他框架要更复杂和烦琐，而且运行时依赖较多。Appium 的社区支持良好，有超过 100 名的开发者在 Github 上面维护和优化它的源码，其中文资料和书籍也都比较齐全。

1 <http://www.robotium.org>

2 <https://code.google.com/p/android-test-kit/wiki/Espresso>

3 <https://developer.android.com/tools/support-library/index.html>

4 <http://appium.io/>

第49章

Android静态代码分析实战

实际项目开发中，程序员的工作是将产品经理的需求以代码的形式实现出来，而代码编写的完成仅仅是一切工作的开始，后面会接着进行前后端功能的联调、bugs 的修复、代码质量的优化等工作。本章就来说说代码质量优化的一项基本的工作：使用静态代码质量分析工具来扫描代码的规范以及存在的隐患。

在 Android 开发中，有四大经典的静态代码扫描和分析工具可供使用，它们分别是代码规范检查工具 CheckStyle¹、Java 静态代码分析工具 FindBugs²、Java 静态代码分析工具 PMD³、Android 代码优化工具 Lint⁴。

49.1 Java代码规范检查工具CheckStyle

CheckStyle 是一个针对 Java 语言的代码规范检查工具，默认情况下，它遵循 Google 的 Java 编码规范和 Sun 的代码规范，同时它又是高度可配置的，不同的团队可以根据自身的情况对检查规则进行裁剪或者新增。

49.1.1 Gradle方式

首先我们需要根据 CheckStyle 的官方文档进行检查规则的配置，主要有两个配置文件。

- checkstyle.xml：检查规则配置文件。

1 <https://github.com/checkstyle/checkstyle>

2 <http://findbugs.sourceforge.net/>

3 <http://sourceforge.net/projects/pmd/files/pmd-eclipse/update-site/>

4 <http://tools.android.com/tips/lint>

- suppressions.xml: 工程中某些特殊文件不需要检查的可以在这个文件中进行配置。

完成规则的配置是最重要的也是每个开发团队需要定制的部分, 接下来我们在工程根目录中新建一个名为 config 的目录, 该目录用于存放整个工程的一些配置信息, 例如主题样式配置信息, 模块化配置信息, 渠道配置信息, 以及我们的代码质量检查配置信息等, 每种配置信息独立为一个子目录, 并配置对应的 .gradle 文件, 例如代码质量目录命名为 quality, gradle 文件命名为 quality.gradle, 同时在 quality 目录中新建四个子目录, 分别用来存放四大检查工具的配置文件, 最终的目录结构如图 49-1 所示。



图 49-1

规则配置文件的目录结构确定后, 接着就开始编写在 Gradle 中执行 CheckStyle 检查的 Task, 语句如下。

```
apply plugin: 'checkstyle'

task checkstyle(type: Checkstyle) {
    configFile file("${project.rootDir}/config/quality/checkstyle/checkstyle.xml")
    configProperties.checkstyleSuppressionsPath = file("${project.rootDir}/config/quality/checkstyle/suppressions.xml").absolutePath
    source 'src'
    include '**/*.java'
    exclude '**/gen/**'
```

```
classpath = files()
}
```

在工程的 Application Module 的 build.gradle 文件中应用 quality.gradle 文件即可生效。

```
apply from: '../config/quality.gradle'
```

点击图 49-2 中的 CheckStyle Task 即可执行编码规范的检查。

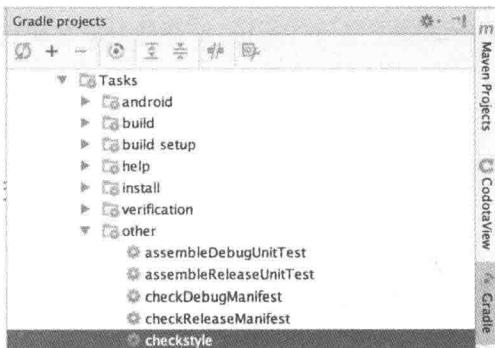


图 49-2

49.1.2 Android Studio插件方式

打开 Android Studio 的 Preferences 面板, 选择其中的 Plugins 选项, 搜索 CheckStyle, 在弹出的搜索结果页面中选择 CheckStyle-IDEA 插件进行安装即可, 如图 49-3 所示。

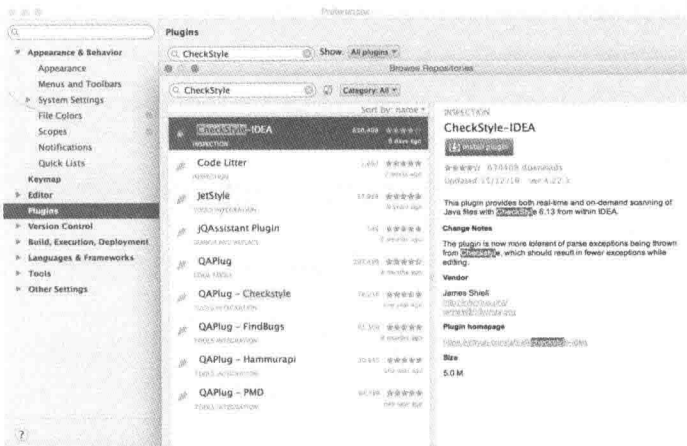


图 49-3

插件方式相比 Gradle 方式的好处是检查出的违规项可以直接点击跳转到源码处，Gradle 方式更适合在持续集成例如 Jenkins 中使用。

49.2 Java静态代码分析工具FindBugs

FindBugs 是一个针对 Java 字节码的静态分析工具，它通过将字节码与一组缺陷模式进行对比来发现代码中可能存在的问题。由于 FindBugs 是针对字节码进行分析的，因此，需要首先将 .java 文件编译为 .class 文件，之后才能开始使用 FindBugs。同样的，FindBugs 也有两种使用方式。

49.2.1 Gradle方式

FindBugs 的规则配置文件名为 findbugs-filter.xml，用于过滤对于团队来说不重要或者可以忽略不计的缺陷或者不需要分析的文件类型。FindBugs 的 Gradle Task 定义如下。

```
apply plugin: 'findbugs'

task findbugs(type: FindBugs, dependsOn: assembleDebug) {
    ignoreFailures = false
    effort = "max"
    reportLevel = "high"
    excludeFilter = new File("${project.rootDir}/config/quality/findbugs/
findbugs-filter.xml")
    classes = files("${project.rootDir}/app/build/intermediates/classes")

    source 'src'
    include '**/*.java'
    exclude '**/gen/**'

    reports {
        xml.enabled = false
        html.enabled = true
        xml {
```

```

        destination "$project.buildDir/reports/findbugs/findbugs.xml"
    }

    html {
        destination "$project.buildDir/reports/findbugs/findbugs.html"
    }
}

classpath = files()
}

```

从 Task 的定义中可以看出，FindBugs 的输出文件类型可选择 xml 或者 html 格式，通过配置可以实现其使能或者去使能。一般我们都会选择输出 html 格式，因为可读性更强，执行 FindBugs Task 的方法跟 CheckStyle Task 方法一样。

49.2.2 Android Studio插件方式

同理，通过在 Android Studio 的 Preferences — Plugins 中搜索 FindBugs 即可找到并安装 FindBugs 插件，如图 49-4 所示。

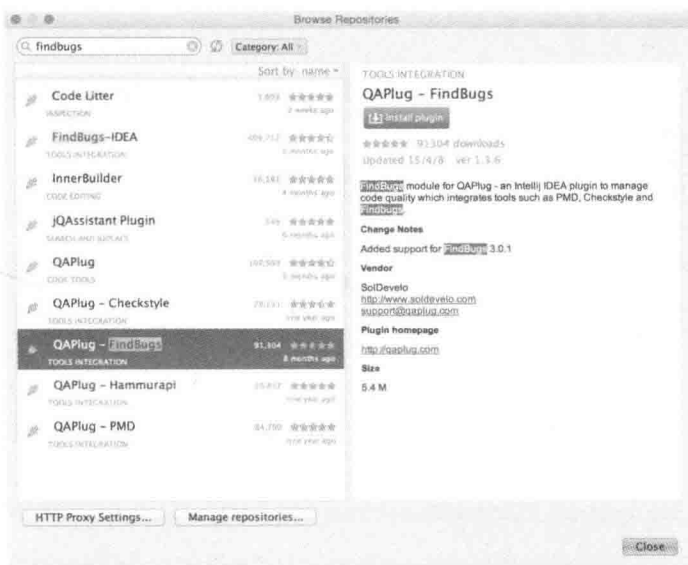


图 49-4

49.3 Java静态代码分析工具PMD

PMD 是一个针对 Java 源代码的静态代码分析工具，其功能与 FindBugs 非常类似，只不过一个是针对 Java 源代码进行分析，一个是针对 Java 字节码进行分析，两者功能有重叠，但并不是可代替的，而是相互补充的关系。

49.3.1 Gradle方式

PMD 的规则配置文件名为 `pmd-ruleset.xml`，Gradle Task 定义如下所示。同样的，输出文件格式可选择 `xml` 或者 `html`，从可读性来讲，我们一般选择输出 `html` 格式。

```
apply plugin: 'pmd'

task pmd(type: Pmd) {
    ignoreFailures = false
    ruleSetFiles = files("${project.rootDir}/config/quality/pmd/pmd-ruleset.xml")
    ruleSets = []

    source 'src'
    include '**/*.java'
    exclude '**/gen/**'

    reports {
        xml.enabled = false
        html.enabled = true
        xml {
            destination "$project.buildDir/reports/pmd/pmd.xml"
        }
        html {
            destination "$project.buildDir/reports/pmd/pmd.html"
        }
    }
}
```

49.3.2 Android Studio插件方式

同理,通过在 Android Studio 的 Preferences — Plugins 中搜索 pmd 即可找到并安装 pmd 插件,如图 49-5 所示。

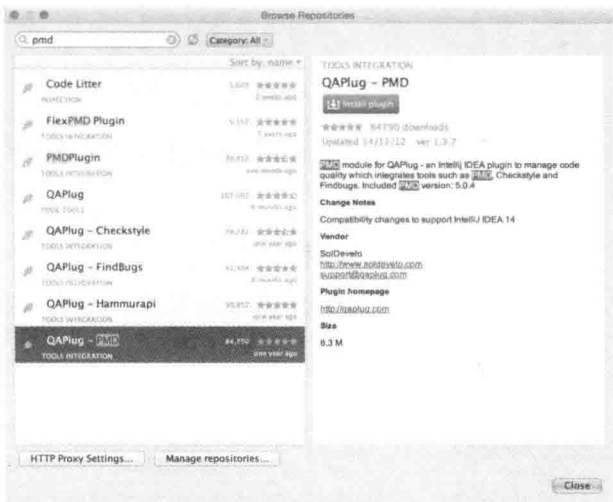


图 49-5

49.4 Android代码优化工具Lint

Android Lint 工具是 Android Studio 自带的静态代码检查工具,跟前面几个工具不同的是,Android Lint 是专门针对 Android 定制的检查规则,因此可以检查出很多 Android 特有的代码缺陷。建议在开发过程中,经常性地检查编写的代码是否存在问题,每次功能模块提测前运行 Lint 是一个不错的时机。

49.4.1 Gradle方式

Android Lint 的规则配置文件名为 lint.xml, Gradle Task 定义如下。

```
android {
    lintOptions {
        abortOnError true
    }
}
```

```

xmlReport false
htmlReport true
lintConfig file( "${project.rootDir}/config/quality/lint/lint.xml" )
htmlOutput file( "${project.buildDir}/reports/lint/lint-result.html" )
xmlOutput file( "${project.buildDir}/reports/lint/lint-result.xml" )
}
}

```

49.4.2 Android Studio插件方式

Android Studio 自身集成了 Android Lint，用鼠标右键单击工程中的任何一个文件夹，在弹出的菜单中选择 Analyze → Inspect Code...，即可执行 Lint 检查，如图 49-6 所示。

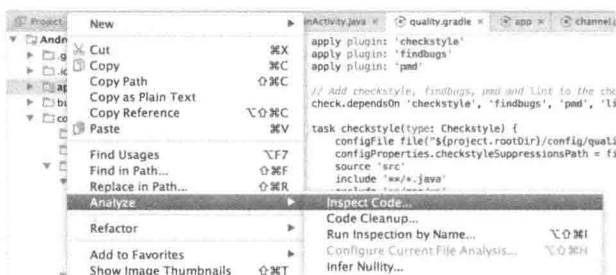


图 49-6

检查结果如图 49-7 所示，缺陷类型为我们很好地进行了分类。



图 49-7

第50章

基于Jenkins + Gradle搭建Android持续集成编译环境

持续集成编译环境是敏捷开发中很重要的一个组成部分，它能够有效地提高整个团队的生产效率，最大化的减少人为出错的可能。例如，通过代码的持续提交，可以减少代码合并的痛苦，更快地和其他人代码集成；通过持续编译，能够及早地发现代码库存在的错误，并支持产品、测试等人员及时取包进行功能验证。

实现自动化构建起码需要具备以下两个条件。

- 一台持续集成服务器：这台服务器的任务是从代码托管服务器（svn 或者 git）自动拉取最新的代码，并进行代码的编译打包并最终输出 app 的安装包，同时发邮件通知团队成员持续集成结果，考虑到目前移动端产品都拥有 Android 和 iOS 两个平台，因此这台持续集成服务器必须是安装了 Max OSX 系统的。
- 一个大家可访问的 Web 页面：可以为不同的项目创建不同的构建机制，提供定时自动构建（Daily Build）和用户手动构建两种触发方式。

为了满足上面的条件，目前绝大多数公司都是采用 Mac + Jenkins 的方式来实现持续集成。

50.1 Tomcat的下载和启动

Jenkins 是一个 Web 应用程序，为了让团队中其他人能够访问到 Jenkins，首先要将它部署到 Web 服务器中，通常情况下，我们选择 Apache Tomcat¹，到官网²下载最新的版本即可，如图

¹ <http://tomcat.apache.org/>

² <http://tomcat.apache.org/download-80.cgi>

50-1 所示。我们选择最新的发布版 8.0.32，并根据系统下载对应的压缩包，对于 Mac 系统，下载 zip 或者 tar.gz 都可以。



图50-1

下载完成并进行安装或者解压后，可以看到 Tomcat 的目录结构如图 50-2 所示。



图50-2

一般情况下，我们都需要对 Tomcat 进行各种自定义的配置，以优化性能并符合自身的业务需求，不过对于部署 Jenkins 而言，由于是团队内部人员使用，功能也简单，因此一般无需进行其他配置。启动 Tomcat，只需在 Terminal 中进入 bin 目录，执行如下命令。

```
./startup.sh
```

即可启动 Tomcat 服务器，如果出现如下错误。

```
-bash: ./startup.sh: Permission denied
```

说明 Tomcat/bin 目录下面的 .sh 文件没有执行权限, 需要手动修改, 在 bin 目录执行如下命令。

```
chmod u+x *.sh
```

正常启动 Tomcat 后, 在 Terminal 中可以看到如下日志。

```
C02PC5C1FVH5:bin guhaoxin$ ./startup.sh
Using CATALINA_BASE:   /Users/guhaoxin/Downloads/apache-tomcat-8.0.32
Using CATALINA_HOME:   /Users/guhaoxin/Downloads/apache-tomcat-8.0.32
Using CATALINA_TMPDIR: /Users/guhaoxin/Downloads/apache-tomcat-8.0.32/temp
Using JRE_HOME:        /Library/Java/JavaVirtualMachines/jdk1.7.0_75.jdk/
Contents/Home/
Using CLASSPATH:       /Users/guhaoxin/Downloads/apache-tomcat-8.0.32/bin/
bootstrap.jar:/Users/guhaoxin/Downloads/apache-tomcat-8.0.32/bin/tomcat-
juli.jar
Tomcat started.
```

这时在浏览器地址栏中访问 localhost:8080 可以看到如图 50-3 所示页面。

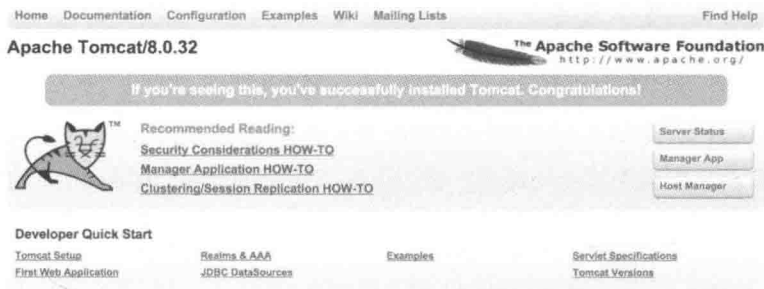


图50-3

50.2 Jenkins的下载和运行

在 Mac OSX 系统上安装 Jenkins, 可以到 Jenkins¹ 官网下载 Java Web Archive (.war) 包, 下载完成后会得到 jenkins.war 文件, 将这个文件复制到 Tomcat 的 webapps 目录下面, 重启 Tomcat, 在浏览器中访问地址 <http://localhost:8080/jenkins/> 即可打开 Jenkins 首页, 如图 50-4 所示。

¹ <http://jenkins-ci.org/>



图 50-4

50.3 Jenkins插件的安装

为了实现 Jenkins 构建 Android Studio 工程，我们需要安装以下基础插件。

- Gradle plugin：用于支持 Jenkins 执行 Gradle 构建脚本。
- GIT plugin：如果代码库是以 Git 方式托管的话，用于支持 Jenkins 拉取远程代码托管服务器的 Git 仓库。
- Subversion Plug-in：如果代码库是以 SVN 方式托管的话，用于支持 Jenkins 拉取远程代码托管服务器的 SVN 仓库。
- SSH Credentials Plugin：SSH 证书插件，用于支持 Jenkins 本地存储 SSH 证书。

点击 Jenkins 首页“系统管理”按钮可以打开管理 Jenkins 页面，在这个页面中进行 Jenkins 的一系列配置，例如配置系统全局路径、管理插件、安全设置等，如图 50-5 所示。



图 50-5

点击“管理插件”按钮，进入插件管理页面，在“可选插件”选项卡中搜索上面的插件进行安装，安装完成后重启 Jenkins 使其生效即可，插件安装和更新过程图 50-6 所示。

安装/更新 插件中

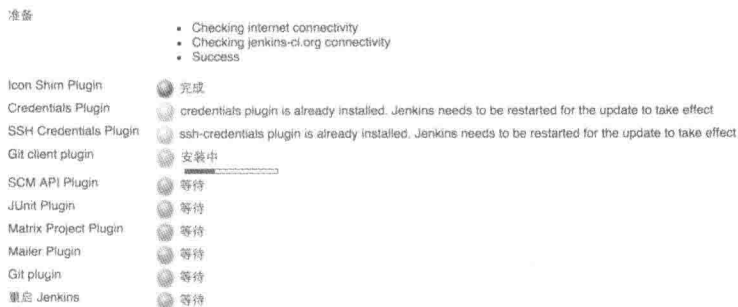


图50-6

50.4 Jenkins全局配置

点击 Jenkins 首页“系统管理—系统设置”，可以进入 Jenkins 的全局配置页面，在这个页面中，我们需要进行 JDK、Android SDK、Git、SVN 和 Gradle 这些环境的配置。

50.4.1 配置 JDK 环境

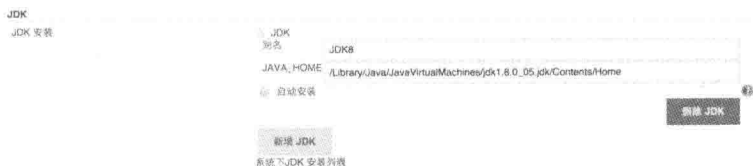


图50-7

50.4.2 配置 Android SDK 环境



图50-8

50.4.3 配置 Git 环境



图 50-9

50.4.4 配置 SVN 环境



图 50-10

50.4.5 配置 Gradle 环境



图 50-11

50.5 JOB相关的操作

50.5.1 JOB 的创建

在 Jenkins 首页点击左边栏的“新建”按钮，选择“构建一个自由风格的软件项目”并输入 Item 名称即可创建一个新的名为 EventBusProject 的 JOB，如图 50-12 所示。



图50-12

50.5.2 JOB 的配置

创建完成后，在 Jenkins 首页可以看到新建的这个 JOB，点击进入 EventBusProject 页面，接着点击“配置”按钮，打开 EventBusProject 这个 JOB 的配置页面，首先我们需要配置代码仓库地址信息，下面以 Git 为例进行说明，如果是以 HTTPS 方式访问 Git，最基础的只需要配置 Repositories URL，并指定构建的分支（Branches to build），如图 50-13 所示。



图50-13

如果是以 SSH 方式访问 Git，则还需要配置 Credentials，点击上图中的 Add 按钮，在弹出的页面中选择 SSH Username with private key 方式，配置如图 50-14 所示。

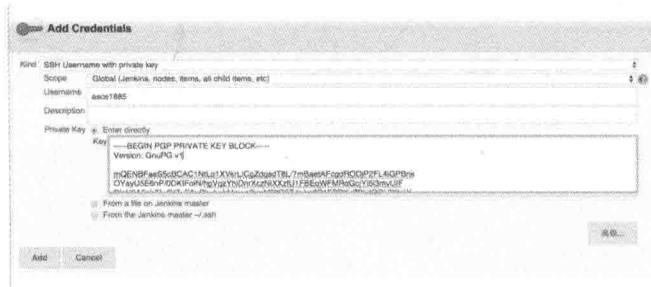


图50-14

返回 JOB 配置主页面，选择刚才新添加的名为 asce1885 的 Credentials 即可使用 SSH 方式访问 Git 仓库了，如图 50-15 所示。



图 50-15

50.5.3 Gradle 的配置

由于我们的工程是使用 Gradle 进行构建的，因此需要增加 Gradle 配置相关信息，通过单击“增加构建步骤→Invoke Gradle script”，如图 50-16 所示。



图 50-16

可以打开配置对话框，主要配置 Tasks、Root Build script 和 Build File 这三项。

- Tasks：填入需要执行的 Gradle 任务列表。
- Root Build script：工程的根目录，如果工程遵循 Android Studio 标准的项目结构，可以不显式指定。
- Build File：指定构建文件的位置，如果构建文件遵循默认的build.gradle命名，则此处可以不显式指定。

最终的取值如图 50-17 所示。

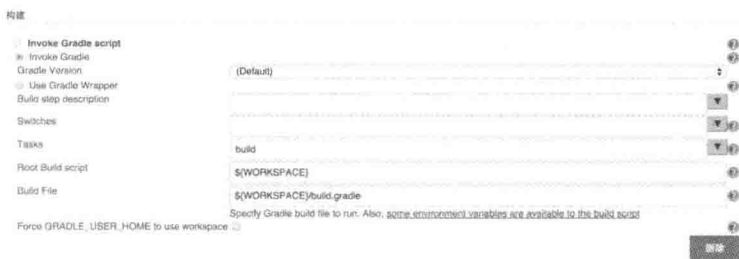


图50-17

其中 `$(WORKSPACE)` 表示 JOB 的工作区，`WORKSPACE` 是 Jenkins 预定义的环境变量。对于 `EventBusProject` 这个 JOB 的工作区内容如图 50-18 所示，这些文件是从上面配置的 Git 地址自动拉取的。



图50-18

50.5.4 构建触发器的配置

Jenkins 支持自动触发 JOB 的构建，这是通过配置构建触发器来实现的，主要有两种策略。

- Poll SCM：设置定时器定时检测代码服务器是否有代码更新，如果有则拉取并构建，否则不执行构建，如图50-19所示，我们设置每隔10分钟检查一次。



图50-19

- Build periodically: 设置定时器周期性的执行构建任务，如图50-20所示，我们设置每天8点准时构建。



图50-20

日程表的格式要求可以参加 Jenkins 的提示，点击上图日程表后面的问号，就会显示出日期格式的说明。

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

MINUTE HOUR DOM MONTH DOW

MINUTE Minutes within the hour (0-59)

HOUR The hour of the day (0-23)

DOM The day of the month (1-31)

MONTH The month (1-12)

DOW The day of the week (0-7) where 0 and 7 are Sunday.

To specify multiple values for one field, the following operators are available. In the order of precedence,

* specifies all valid values

M-N specifies a range of values

M-N/X or */X steps by intervals of X through the specified range or whole valid range

A,B,...,Z enumerates multiple values

To allow periodically scheduled tasks to produce even load on the system, the symbol H (for "hash") should be used wherever possible. For example, using 0 0 * * * for a dozen daily jobs will cause a large spike at midnight. In contrast, using H H * * * would still execute each job once a day, but not all at the same time, better using limited resources.

The H symbol can be used with a range. For example, H H(0-7) * * * means some time between 12:00 AM (midnight) to 7:59 AM. You can also use step

intervals with H, with or without ranges.

The H symbol can be thought of as a random value over a range, but it actually is a hash of the job name, not a random function, so that the value remains stable for any given project.

Beware that for the day of month field, short cycles such as */3 or H/3 will not work consistently near the end of most months, due to variable month lengths. For example, */3 will run on the 1st, 4th, ...31st days of a long month, then again the next day of the next month. Hashes are always chosen in the 1-28 range, so H/3 will produce a gap between runs of between 3 and 6 days at the end of a month. (Longer cycles will also have inconsistent lengths but the effect may be relatively less noticeable.)

Empty lines and lines that start with # will be ignored as comments.

In addition, @yearly, @annually, @monthly, @weekly, @daily, @midnight, and @hourly are supported as convenient aliases. These use the hash system for automatic balancing. For example, @hourly is the same as H * * * * and could mean at any time during the hour. @midnight actually means some time between 12:00 AM and 2:59 AM.

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
H/15 * * * *
# every ten minutes in the first half of every hour (three times, perhaps at
:04, :14, :24)
H(0-29)/10 * * * *
# once every two hours every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38
PM, 4:38 PM)
H 9-16/2 * * 1-5
# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *
```

50.5.5 参数化构建

多数情况下，JOB 的构建需要支持可动态配置参数，也就是说在开始构建之前，可以修改某些参数的取值，从而得到不同的构建结果，例如根据输入参数，决定此次构建是编译 Release 版本还是 Debug 版本等，这就需要通过参数化构建来实现。在 JOB 配置页面点击“参数化构建过程→添加参数”，如图 50-21 所示。

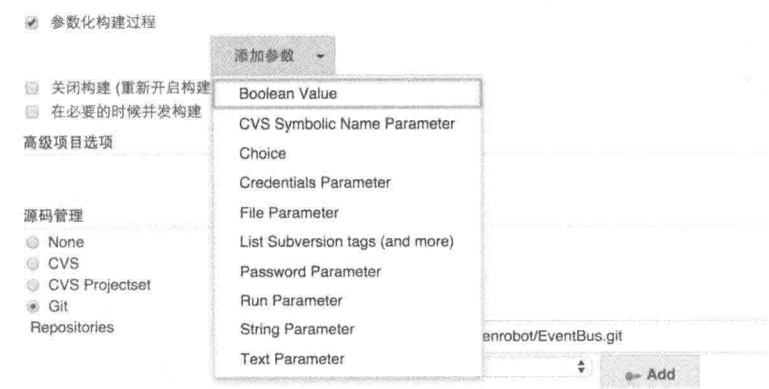


图50-21

接着根据参数的类型选择对应的选项，配置完成后，每次构建之前，都要先设置这些参数的取值，如图 50-22 所示。



图50-22

50.6 Jenkins预定义的环境变量

最后列举一下 Jenkins 预先定义好的环境变量，在配置 JOB 时使用这些变量能够使得配置更简单。

BUILD_NUMBER

The current build number, such as “153”

BUILD_ID

The current build ID, identical to BUILD_NUMBER for builds created in 1.597+, but a YYYY-MM-DD_hh-mm-ss timestamp for older builds

BUILD_DISPLAY_NAME

The display name of the current build, which is something like “#153” by default.

JOB_NAME

Name of the project of this build, such as “foo” or “foo/bar”. (To strip off folder paths from a Bourne shell script, try: `${JOB_NAME##*/}`)

BUILD_TAG

String of “jenkins-`${JOB_NAME}`-`${BUILD_NUMBER}`”. Convenient to put into a resource file, a jar file, etc for easier identification.

EXECUTOR_NUMBER

The unique number that identifies the current executor (among executors of the same machine) that’s carrying out this build. This is the number you see in the “build executor status”, except that the number starts from 0, not 1.

NODE_NAME

Name of the slave if the build is on a slave, or “master” if run on master

NODE_LABELS

Whitespace-separated list of labels that the node is assigned.

WORKSPACE

The absolute path of the directory assigned to the build as a workspace.

JENKINS_HOME

The absolute path of the directory assigned on the master node for Jenkins to store data.

JENKINS_URL

Full URL of Jenkins, like `http://server:port/jenkins/` (note: only available if Jenkins URL set in system configuration)

BUILD_URL

Full URL of this build, like `http://server:port/jenkins/job/foo/15/` (Jenkins URL must be set)

JOB_URL

Full URL of this job, like `http://server:port/jenkins/job/foo/` (Jenkins URL must be set)

SVN_REVISION

Subversion revision number that's currently checked out to the workspace, such as "12345"

SVN_URL

Subversion URL that's currently checked out to the workspace.
