

2019 届硕士学位论文

分类号: \_\_\_\_\_

学校代码: 10269

密 级: \_\_\_\_\_

学 号: 51164500190



华东师范大学

构建 Android 动态函数调用图:

RunDroid 的设计与实现

院 系: 计算机科学与软件工程学院

专业名称: 计 算 机 技 术

研究方向: 软件方法与程序语言

指导教师: 徐立华 副 教 授

学位申请人: 袁 宇 杰

2018 年 11 月

Dissertation for master degree in 2019  
(Professional)

University Code: 10269  
Student ID: 51164500190

# EAST CHINA NORMAL UNIVERSITY

## **Building Android Dynamic Call Graph: Design and Implement of RunDroid**

Department:	School of Computer Science and Software Engineering
Major:	Computer Technique
Research Direction:	Software Method and Program Language
Supervisor:	Associate Professor Lihua Xu
Candidate:	Yujie Yuan

Nov, 2018

## 华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《构建 Android 动态函数调用图：RunDroid 的设计与实现》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名：\_\_\_\_\_

日期： 年 月 日

## 华东师范大学学位论文著作权使用声明

《构建 Android 动态函数调用图：RunDroid 的设计与实现》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，著作权归本人所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和学校指定的相关机构送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

- ( ) 1. 经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文 \*，于 年 月 日解密，解密后适用上述授权。
- ( ) 2. 不保密，适用上述授权。

导师签名：\_\_\_\_\_

本人签名：\_\_\_\_\_

年 月 日

\* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

## 袁宇杰 硕士专业学位论文答辩委员会成员名单

姓名	职称	单位	备注
X x	教授	华东师范大学	主席
X x	教授	华东师范大学	
X x	教授	华东师范大学	

## 摘 要

在 Android，应用程序的逻辑分散在不同的代码段（例如方法、线程、组件等）中，这使得部分静态分析工具在分析时，得不到精确的结果。为了帮助研究人员和开发人员了解 Android 应用程序的执行过程，我们提供了 RunDroid，一个用于还原 Android 应用程序运行时动态调用图的工具，进而帮助辅助静态工具提供更精确的程序分析结果。RunDroid 利用源程序代码插桩和运行时方法拦截的相结合的方式，捕获应用程序在应用层和系统层的方法执行信息，还原方法间的调用关系。在此基础上，RunDroid 利用对象和方法间的依赖关系，进一步还原方法之间的触发关系（例如），在调用图中展现运行过程中的 Android 特性。

另外，我们还将 RunDroid 和静态分析工具进行对比，分析两种技术在生成函数调用图上的优缺点。

**关键词:** Android, 函数调用图, 动态分析技术, 生命周期, 多线程

# ABSTRACT

In Android, the logic of the application is spread across different code segments (such as methods, threads, components, etc.), which makes some static analysis tools less accurate when analyzing. To help researchers and developers understand the execution of Android applications, we have provided RunDroid, a tool for restoring dynamic call graphs for Android application runtimes, to help assist static tools to provide more accurate program analysis results. RunDroid uses the combination of source code instrumentation and runtime method interception to capture the application execution information of the application at the application layer and system level, and restore the call relationship between methods. On this basis, RunDroid uses the dependencies between objects and methods to further restore the trigger relationship between the methods (for example), and display the Android features in the running graph in the call graph.

In addition, we also compare RunDroid with static analysis tools to analyze the advantages and disadvantages of the two techniques in generating function call graphs.

**Keywords:** *Android, Call Graph, Dynamic Analysis, Multi-Thread*

# 目录

<b>摘要</b> . . . . .	i
<b>ABSTRACT</b> . . . . .	ii
<b>第一章 绪论</b> . . . . .	1
1.1 研究背景 . . . . .	1
1.2 Android 分析技术 . . . . .	3
1.2.1 静态分析技术 . . . . .	3
1.2.2 动态分析技术 . . . . .	4
1.2.3 分析技术的应用 . . . . .	6
1.3 论文研究内容 . . . . .	7
1.4 本文组织结构 . . . . .	8
<b>第二章 Android 系统相关背景介绍</b> . . . . .	9
2.1 Android 系统结构介绍 . . . . .	9
2.2 Android 中的 Activity . . . . .	11
2.3 Android 中的多线程交互 . . . . .	13
2.3.1 基于 Java 的多线程交互 . . . . .	14
2.3.2 基于 Handler 的多线程消息调度 . . . . .	14
2.4 本文遇到的困难与挑战 . . . . .	17
2.5 本章总结 . . . . .	17
<b>第三章 名称解释</b> . . . . .	19
3.1 概念说明 . . . . .	19
3.1.1 关于方法和对象的定义 . . . . .	19
3.1.2 关于方法间关系的定义 . . . . .	20

3.1.3	关于调用图的定义	21
3.2	举例说明	22
3.3	本章总结	22
<b>第四章</b>	<b>RunDroid 的系统设计</b>	<b>23</b>
4.1	整体设计框架	23
4.2	方法信息的捕获	25
4.2.1	用户方法执行信息的获取	25
4.2.2	系统方法执行信息的获取	26
4.3	扩展函数调用图的构建过程	27
4.3.1	构建函数调用图	28
4.3.2	构建 Activity 的生命周期和事件回调	29
4.3.3	构建多线程触发关系	29
4.4	本章总结	32
<b>第五章</b>	<b>RunDroid 的系统实现</b>	<b>33</b>
5.1	相关技术介绍	33
5.1.1	srcML	33
5.1.2	Xposed 框架	33
5.1.3	Neo4j	34
5.2	模块实现	34
5.2.1	预处理器	34
5.2.2	运行时拦截器	34
5.2.3	日志记录器	35
5.2.4	调用图构建器	35
5.3	本章总结	36
<b>第六章</b>	<b>系统测试</b>	<b>37</b>
6.1	应用结果展示	37
6.1.1	函数调用图的构建结果展示	37
6.1.2	Activity 生命周期和事件回调的效果展示	41

6.1.3	多线程触发关系效果展示	43
6.2	RunDroid 在错误定位领域的应用	45
6.2.1	原理简介	45
6.2.2	结果分析	46
<b>第七章</b>	<b>总结与展望</b>	<b>47</b>
7.1	总结	47
7.2	展望	48
<b>参考文献</b>		<b>49</b>
<b>致谢</b>		<b>49</b>
<b>攻读学位期间发表的学术论文</b>		<b>50</b>

## 插 图

图 1.1 Google Play Store 上架的应用总数的变化趋势 . . . . .	1
图 2.1 Android 系统框架图 . . . . .	10
图 2.2 Activity 的生命周期 . . . . .	12
图 2.3 Handler 的使用实例 . . . . .	15
图 2.4 Handler 各 API 之间的调用关系 . . . . .	15
图 2.5 Handler 机制中各组成部分的相互关系 . . . . .	16
图 4.1 RunDroid 的基本思路 . . . . .	24
图 4.2 RunDroid 的工作流程 . . . . .	25
图 5.1 使用 Cypher 查找共用对象 $o_m$ 的方法 $m_{enqueue}$ 、 $m_{dispatch}$ . . . . .	36
图 6.1 MainActivitiy 的代码 . . . . .	38
图 6.2 斐波拉契数列-RunDroid 生成的调用图（局部） . . . . .	39
图 6.3 斐波拉契数列-FlowDroid 生成的调用图（局部） . . . . .	40
图 6.4 Activity 部分-RunDroid 生成的调用图（局部） . . . . .	41
图 6.5 dummyMainMethod-FlowDroid 生成的调用图 . . . . .	42
图 6.6 多线程触发关系-RunDroid 生成的调用图（局部） . . . . .	43
图 6.7 多线程触发关系-FlowDroid 生成的调用图（局部） . . . . .	43

## 表 格

表 4.1 运行时拦截器拦截的系统方法列表 . . . . . 27

# 算 法

算法 4.1 日志代码插桩过程 . . . . .	26
算法 4.2 函数调用图的构建过程 . . . . .	28
算法 4.3 构建 Activity 的生命周期和事件回调 . . . . .	30
算法 4.4 扩展函数调用图的构建过程 . . . . .	31
算法 6.1 错误定位的计算方法 . . . . .	46

# 第一章 绪论

## 1.1 研究背景

在现在社会，人们和移动设备的关系越来越密切，衣食住行几乎都离不开手机。每天，人们只需要打开手机上的应用，就可以完成几乎所有生活需求，从出行打车到在线订餐，从网上购物到房屋租赁。移动应用已经深入到人们生活的方方面面。以移动系统 Android 为例，根据著名网站 statista 的统计 [?] 显示，Android 官方应用平台 Google Play Store 在 2009 年 12 月至 2018 年 6 月期间的应用数量变化如图 1.1 所示。Google Play Store 于 2008 年 8 月上线，截止 2018 年 3 月，在 Google Play Store 上架的应用已经超过 330 万。这个数字在 2013 年 7 月才刚刚突破 100 万。这也从一个侧面反映出最近几年移动应用迅猛的增长趋势。

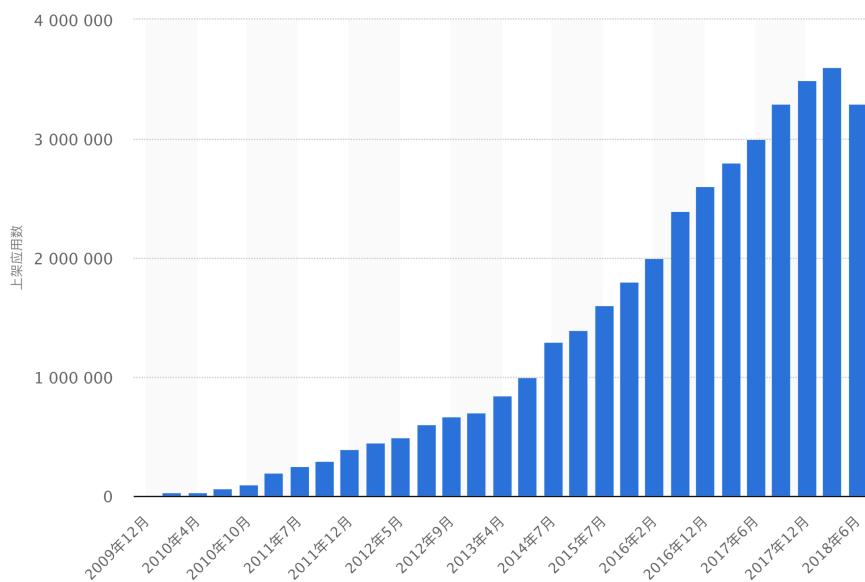


图 1.1: Google Play Store 上架的应用总数的变化趋势

正因为移动应用迅猛的增长趋势，学术界和工业届的相关人员开始研究如何通过技术手段分析移动应用的代码内容，了解应用本身的运行时行为，进行相关学术研究和工业生产。利用程序分析技术，研究人员对应用程序的项目相关源代码、配置文件或者二进制分发文件进行分析，监控程序的运行时行为，总结出应用程序相关特征。结合具体应用场景，我们对这些特征进行归纳总结出相关规律，应用在应用分析、安全风险以及质量保障等领域，进一步提升应用程序的易用性、安全性和可靠性。

根据分析过程中是否需要运行目标程序，我们可以将这些技术手段分为静态分析技术和动态分析技术。如果分析过程不依赖于目标程序的运行，这种分析技术称为静态分析技术，反之则为动态分析技术。静态分析技术通常以二进制程序文件作为研究主体，结合相应的控制流分析、数据流分析技术、指针分析以及程序依赖分析，得出应用程序过程内的控制依赖和数据依赖（两者统称为程序依赖）；根据程序方法内相关函数/方法<sup>1</sup>调用，可以得到函数调用图、UML 类图和序列图；我们将程序依赖数据和函数调用图相结合，可以进一步得到过程间程序依赖[?]，帮助研究人员了解程序整体层面的业务间的依赖关系，进行污点传播分析。相反的，动态分析技术依赖目标程序的运行，通过修改目标程序的运行文件，搭建目标程序的运行环境，记录程序运行过程中相关操作信息，监控目标程序在运行过程中的状态变迁或指标变化，进而得出程序在运行过程中的行为特定，帮助研究人员进行程序安全性分析，提升程序的质量可靠性。

但是，上述两种技术各有各的优劣。静态分析技术有着较为扎实的理论基础，分析结果精确可靠，覆盖范围全面。但是，静态分析技术在枚举所有情况时，往往会遇到状态爆炸的问题，具体实验效果受到实验运行环境的硬件条件和算法实现程度的限制。而且，静态分析技术分析的问题依赖于外部环境（用户实时操作序列、手机所处环境因素，如温度等），分析得到的结果并不是非常准确。动态分析技术却能解决这个问题，通过对程序运行状态的监控，研究人员可以了解程序的

<sup>1</sup>在 Java 语言中，函数称为方法。在本文中，两者可以相互替换，不做区分。

运行行为，掌握程序的安全性信息和可靠性信息。但是，动态分析技术的缺点也非常明显：动态运行环境的搭建往往要设计到相关系统的源代码，构建系统的时间成本大，技术要求高。另外，动态分析技术的分析结果往往只针对一次程序的运行过程，无法直接推广到其他运行情况。

Android 应用程序的特性（例如，基于事件驱动的基础架构、面向组件的开发方式、高度依赖回调函数和多线程交互等）使得传统分析工具无法直接应用在 Android 程序上，对研究人员了解 Android 应用程序执行细节造成了一定的困扰。为了解决这个问题，本文提出了一种静动态相结合的技术方案，通过程序源代码和运行环境进行预处理，获得程序的运行时信息，进而还原出 Android 应用程序的动态函数调用图。在调用图中，除了方法调用关系，我们还提供了方法对象、方法间触发关系等信息，可以帮助研究人员补全函数关系，较为全面地了解应用程序运行时的状态变化。

## 1.2 Android 分析技术

通常的，软件分析技术主要分为静态分析技术和动态分析技术两类。

### 1.2.1 静态分析技术

**缩减此处，再添加 3~5 篇文献** 在不执行应用程序的情况下，静态分析技术通过对应用程序的源代码或者执行文件进行控制流分析和数据流分析，进而推断应用程序在运行过程中可能产生的行为。这方面相关工具包括 Soot[?]、FlowDroid[?]、AmanDroid[?]、IccTA[?]、AndroGuard[?] 等。Soot[?] 是传统的静态分析工具，其思路是将所有的 Java 字节码文件转化成一种中间语言 Jimple，并在 Jimple 的基础上进行常规的控制流分析、数据流分析，理论上适用于所有可以在 Java 虚拟机上运行的语言（例如 Scala、Groovy 等等）的分析。由于 Android 程序本身的字节码 Dalvik 和 Java 字节码在格式上保持一致，因此，Soot 也支持 Android 应用程序的静态分析。但是，Soot 在分析过程中没有考虑一些 Android 的特性难免会出现一

些问题。为此，德国达姆施塔特工业大学的 Steven Arzt 等人在 Soot 的基础上考虑 Android 程序中 Activity 的生命周期特性，推出了一个针对 Android 的静态分析工具 FlowDroid[? ]，可以做到上下文、路径、对象、字段等层面上的敏感。FlowDroid 通过定义数据源点和数据泄漏点，在 Android 应用生命周期的基础上，可以实现数据流敏感的污点分析。但其不足之处在于缺少跨组件通信的分析不考虑多线程调用问题。在 FlowDroid 基础上，卢森堡大学的 Li Li 等人推出了 IccTA[? ]，利用跨组件通信分析工具 IC3 提取跨组件通信（Inter-Component Communication, ICC）的方法调用，并结合 AndroidManifest.xml 文件定义的 Intent Filter 信息，连接 ICC 两端的组件，克服了 FlowDroid 因缺少跨组件通信而导致的数据流上的缺失。因为它是构建在 FlowDroid 之上一个探测敏感信息泄露的，所以受限于 FlowDroid 的局限性。

Yang 等人 [? ]，利用静态分析技术，并将回调函数添加到控制流图（调用图）中，形成了回调控制流图（Callback control-flow graph）。实验结果显示，Yang 的工作比 Gator [? ] 在控件监听器绑定上得到了更为准确结果。

### 1.2.2 动态分析技术

**缩减此处，再添加 3~5 篇文献** 和静态分析技术相对应，动态分析技术通过执行应用程序，获取程序运行过程的相关信息，从而实现对应的研究目的。动态分析技术往往需要对运行环境做适当的修改或者调用特殊的系统接口，记录应用程序运行过程的关键信息，结合数据流追踪等技术，已记录应用程序的运行时行为。这方面的工作代表包括 [? ? ? ? ] 等。Enck 等人提出的 TaintDroid[? ]，是一个高效的系统级的动态污点跟踪和分析系统。它通过修改 Dalvik 虚拟机，利用动态污点分析技术实时监控敏感数据的生成、传播和泄露，实现了变量层面、方法层面、文件层面的数据追踪。此外，TaintDroid 还支持跨进程通信（IPC）层面上的污点分析，因此可以精确分析出应用程序从消费者手机上获取和发布隐私信息的完整传播过程。TaintDroid 提供了较为完备的数据流分析技术，但是不支持控制流追踪，无法给出

相关语句的执行路径。DroidBox[?] 在 TaintDroid 基础上，对 Android Framework 的部分 API 做了修改，可以记录一些开发人员感兴趣的 API（例如文件读写、网络请求、SMS 服务等）的调用，并提供分析结果的可视化。同时，DroidBox 还实现了应用程序的自动安装和执行，弥补了 TaintDroid 在软件测试自动化方面的不足。和 TaintDroid 不同，TraceDroid[?] 采用的是另一种思路，利用字节码插装技术 AspectJ，使得方法在执行时输出相应的日志信息。根据这些信息 TraceDroid 可以还原函数调用图，得到分析结果。由于 Aspect 在进行字节码编织时引入的新的方法会导致方法数 65K 限制问题（即构建 APK 文件的过程中，方法总数超过 65536，进而使得 APK 文件无法成功构建[?]），因此该方案存在不稳定的情况。

另外，研究人员还用动态分析技术查找 Android 应用程序在运行时性能瓶颈。Android 官方性能检测工具 SimplePref[?] 就是其中的一个代表。SimplePref 利用了 Linux 提供的系统接口 *pref\_event\_open*，定时获取到性能监视单元的相关信息（例如 cpu 周期数、执行的指令数、缓存失效次数等）。利用这些信息，SimplePref 可以得到对应时刻的 CPU 状态，还原出各个方法的执行时间和对应的执行路径。根据方法执行时间的长短和对应的执行路径，开发人员可以发现程序的性能瓶颈，进而通过对程序代码做出调整，提升程序的运行性能。但是，该方法受到系统接口回调周期的影响，过于频繁的调度周期会使系统产生过大的开销，影响原有程序的执行；反之，则会丢失部分方法的执行信息。而且，Android 程序关心的性能瓶颈一般都位于主线程，而 SimplePref 会输出所有线程的执行信息，实际开销较大。为此，Uber 的 Nanoscope[?] 采用追踪（Trace）技术在定制化的系统 Nanoscope OS 中运行，在虚拟机解释执行目标方法前后输出相关 Trace 日志，进而得到性能报告。相比 SimplePref，Nanoscope 只输出主线程相关的方法数据信息，大大减低了性能上的开销。但是，nanoscope 的局限性在于构建成本较高，需要配合特定的系统使用。

### 1.2.3 分析技术的应用

上述技术广泛运用在安全性分析、质量保障等领域。

### § 安全性分析：

安全性分析包括隐私泄露、权限机制研究、恶意软件排查等。

**缩减此处，再添加 3~5 篇文献** 为了弥补 Android 官方文档权限说明不完整的状况，多伦多大学的 Kathy Au 等人提出的 PScout [?] 利用 Soot 对 Android 系统程序进行静态分析，构建出 Android 系统的函数调用图，在调用图中标识权限相关的 Binder 跨进程调用，结合逆向可达性分析技术得出相关 API 接口以及对应调用路径上的所有权限检查的映射关系。在 PScout 的基础上，Rahul Pandita 等人将目光聚焦到应用市场上，他们开发的 WhyPer [?] 将自然语言处理技术（NLP）和传统静态分析技术相结合，可以帮人们找出那些应用描述和实际使用到的权限不符的应用。Wei Yang 等人发现一部分恶意应用试图通过模仿正常应用的行为以防止被安全机构识别出来。不同的是恶意软件的行为一般在特定条件下（例如深夜）运行。为此，Wei Yang 等人提出了一种基于静态分析技术的解决方案 AppContext [?]: 在 FlowDroid 提供的函数调用图的基础上，AppContext 结合 Android 常见的系统事件形成 ICFG，提取出安全敏感行为的上下文，并利用 SVM 技术对这些信息进行机器学习，得到最终的安全检测模型。实验结果显示，AppContext 的准确率和召回率分别达到了 87.7% 和 95%，这从一个侧面反映了安全敏感行为的恶意性与该行为的意图（通过上下文反映）更密切相关，验证之前提到的问题。Zhemin Yang 等人的 AppIntent [?] 利用静态分析技术提取应用程序的时间约束条件，再结合符号执行技术，最后得到一串可以导致用户信息泄露的序列。Android 应用程序中存在着大量的网络交互，因此，也有一部分研究 Socket [???] 聚焦于移动应用的网络安全问题。

### § 质量保障

伴随着 Android 应用程序的发展，研究人员也开始思考如何利用技术手段保障应用本身的质量。这方面的工作包括提升软件在测试过程中的代码覆盖率 [???]、程序错误的定位、分析与修复 [???] 以及变异测试 [???] 等。工作 [?] 利用静态分析工具 SCanDroid 得到静态 Activity 迁移图，并结合基于目标探索策略

和深度优先探索策略进而达到了较好的 Activity 覆盖率。Wei Yang 等人的工作 [? ] 通过对 APP 的状态进行建模，在深度优先算法的基础上，使得程序在当前状态无可再遍历状态的情况下进行回溯。相比传统的深度优先算法，Wei Yang 的工作对应的时间开销更小，效率更高。此外，Stoat[? ?] 利用静态、动态分析技术标识出程序本身的状态和事件，从应用中抽象出有限状态机（FSM）模型。基于该模型，Stoat 进行模型变异、测试用例生成与执行、模型迭代，使得应用程序的覆盖率相比之前工作 [? ?] 提升了 17~31%。工作 [? ?] 等将传统软件的错误定位技术——基于频谱的错误定位技术（spectrum-based fault location, SBFL）运用在 Android 应用上，取得了不错的成绩。工作 [? ?] 等主要聚焦如何在 Android 应用程序出现问题时，根据异常的堆栈信息结合软件变异技术或者问答网站生成补丁代码，实现程序的自动修复。Lingling Fan 等人的工作 [?] 通过开源社区中应用的 issue 解析分析，从异常种类（应用异常、系统异常、库（Library）异常）、系统异常分类、错误检查工具以及错误修复方式等若干方面进行较为深入的分析、探讨和总结。

### 1.3 论文研究内容

本文的主要研究工作包括以下：

- Android 动态函数调用图构建系统 RunDroid 的设计与实现：除了传统调用图的方法调用关系，RunDroid 生成的函数调用图还展示方法执行过程相关的对象信息和方法间的触发关系（例如多线程），反映 Android 平台相关的特性（Activity 组件生命周期和事件回调）。
- 静动态分析技术效果对比：详细展示和说明 RunDroid 产生的调用图，并就 Activity 生命周期及事件回调、多线程触发关系等角度和与传统静态分析工具 FlowDroid 产生的静态调用图做了对比分析，并分析两种技术的优劣。
- RunDroid 系统的应用：利用 RunDroid 用于开源 Android 应用的过程分析，反映方法间触发关系在 Android 系统上的常见性。另外，RunDroid 与错误定位技术结合的实验说明 RunDroid 提供的信息，可以提供更多程序依赖信息，有

有助于提高错误定位技术技术结果的准确性。

## 1.4 本文组织结构

本文共分为六章，环绕着 Android 动态函数调用图构建系统的设计与实现展开，各章节内容如下：

**第一章 绪论** 主要介绍了本文的主要研究背景、相关工作以及主要工作内容。

**第二章 Android 系统相关背景介绍** 从 Android 的体系结构出发，介绍了和 Android 系统相关的背景知识。

**第三章 名称解释** 方法和对象的关系、方法间关系、调用图等几个方面介绍本文用到的概念做了符号化的定义，并结合文章的例子对这些概念做解释。

**第四章 RunDroid 的系统设计** 从系统功能、相关挑战、技术路线、技术选型、模块实现等若干方面介绍 RunDroid 的设计与实现。

**第五章 RunDroid 的系统实现**

**第六章 系统测试** 将展示 RunDroid 系统生成的函数调用图的运行结果，并对函数调用图进行详细的阐述。

**第七章 总结与展望** 对本文工作进行总结，并对下一步工作进行展望。

## 第二章 Android 系统相关背景介绍

本章首先会简要介绍 Android 系统架构，其次会对 Android 中的 Activity 组件做基本介绍，接着会着重介绍 Android 中常见的两种多线程交互方式，最后将指出本文系统在实现上的难点。

### 2.1 Android 系统结构介绍

Android 是基于 Linux 内核开发的的开源操作系统，隶属于 Google 公司，主要用于触屏移动设备如智能手机、平板电脑与其他便携式设备，是目前世界上最流行的移动终端操作系统之一。

在系统架构上，Android 自下到上，可以分为四层：Kernel 层、Library 和 Android Runtime(Dalvik/ART)、Framework、Application 等，如图 2.1 所示。Kernel 层是硬件和软件层之间的抽象层，主要是设备的驱动程序，例如：显示驱动、音频驱动、蓝牙驱动、Binder IPC 驱动等。Library 和 Android Runtime(Dalvik/ART)：Library，顾名思义就是向上层提供各种这样的基础功能，例如 SQLite 数据库引擎，Surface Manager 显示系统管理库。Android Runtime 主要是运行 Android 程序的虚拟机，在不同版本的系统上对应着不同的虚拟机，例如在 Android 5.0 及以上是 ART，而在 Android 4.3 及以下是 Dalvik，而 Android 4.4 两者都有。Framework 层主要是系统管理类库，包括 Activity 的管理，消息通知的管理；同时，它是 Application 的基础架构，为应用程序层的开发者提供了 API，其存在简化了程序的开发。而 Application 就是我们平时接触的应用，开发人员公国调用底层 Framework 提供的 API 实现相应的接口。

虽然 Android 应用程序是使用 Java 语言开发的，但是它和传统的 Java 程序有着很大的不同，具体有如下几点：

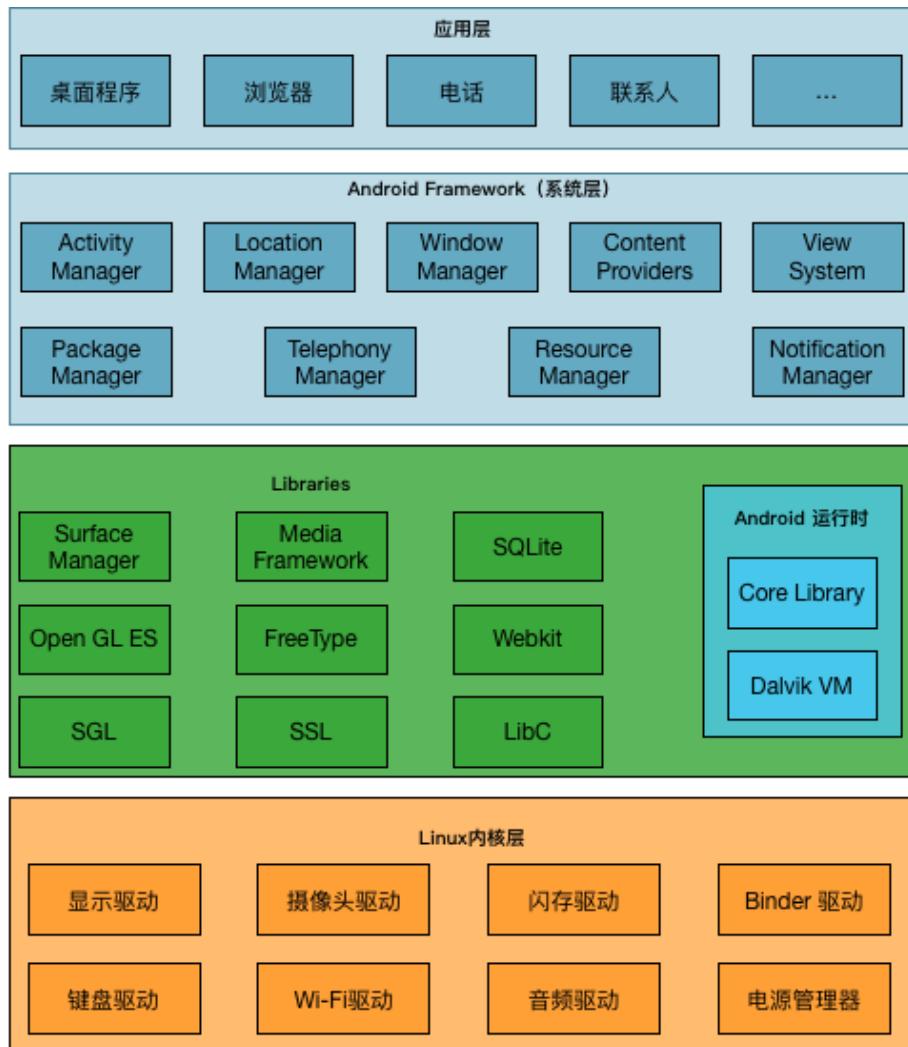


图 2.1: Android 系统框架图

**基于事件驱动的编程模型：**在设计上，Android 应用程序的开发架构采用的是事件驱动架构。在开发过程中，没有传统程序中入口函数 Entry Point 的概念。应用程序中通用的业务逻辑（例如应用程序如何启动退出、应用的窗口如何创建销毁等）存在于 Android Framework 中。这也使得 Android 应用程序的分发文件（即 APK 文件）相对较小。

**面向组件的开发方式：**Android 程序中较为常见的是组件（Component，例如 Activity、Service、Content Provider、Broadcast Receiver），它是应用程序运行的最小单元，受到 Android Framework 的直接调度。开发人员通过继承这些组件，重写对应的生命周期函数，已实现对应的业务需求（界面的布局、页面状态的保存等），

而这些组件的生命周期由 Framework 调度完成。

**大量逻辑实现依赖于回调函数和多线程通信：**由于 Android 应用程序采用的是基于单线程消息队列的事件驱动架构，因此，界面相关的操作只允许出现在主线程（UI Thread）中，耗时操作只能在工作线程（Worker Thread）中进行。通常的，开发人员往往会借助回调函数处理控件的响应事件，利用多线程交互串联界面相关操作和耗时操作，完成对应的业务。

## 2.2 Android 中的 Activity

在 Android 应用程序运行过程中，Activity 向用户展示图形界面，响应用户的反馈，和其他组件一同完成相关业务，扮演着最为重要的作用 [? ]。由于 Android 应用程序在架构选型上采用了事件驱动模型，为了便于协调应用内部状态的管理，Android 组件通常有生命周期的概念，Activity 也不例外。

Android 系统根据 Activity 在运行时和用户的反馈将其状态分为以下四种：

- 运行态：在该状态下，Activity 处于页面最前端时，用户可以与 Activity 进行交互。一般的，我们看到 Activity 均处于这个状态。
- 暂停态：在该状态，Activity 仍然可见，但是失去了窗口的焦点。当一个 Activity 之上出现一个透明的 Activity、Toast 或者对话框时，Activity 就处于这个状态。处于暂停状态的 Activity 仍处于存活状态，保存着所有的内存数据，只有当系统内存极度紧张时，才有可能被系统杀死回收。
- 停止态：当一个 Activity 被其他的 Activity 遮挡时，处于这个状态。处于该状态的 Activity 仍然可以保留所有的状态，只是对用户不可见。系统在需要内存的情况下，可以采用相应的策略对 Activity 进行杀死回收操作。
- 终止态：当 Activity 处于暂停态或者停止态时，系统由于内存原因可能会将上述两种 Activity 杀死回收。处于该状态下的 Activity 将不能直接恢复。

Activity 的生命周期就是以上状态之间的跳转，受到 Activity 在运行时的内存分布、环境状态以及业务逻辑的影响，由 Android 系统直接负责调度。Android *Activity*

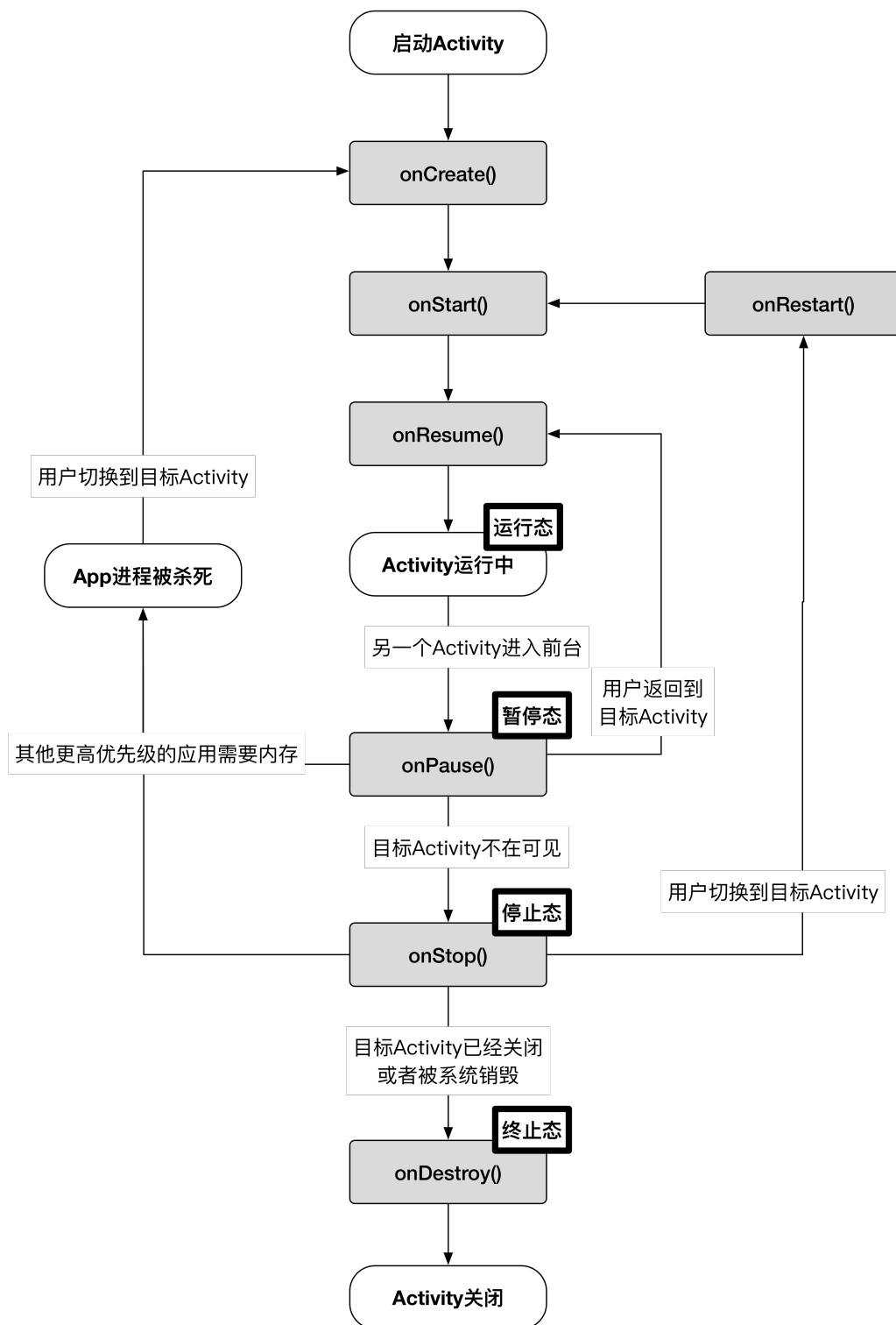


图 2.2: Activity 的生命周期

提供的方法包括 `onCreate()`、`onStart()`、`onResume()`、`onPaused()`、`onRestart()`、`onStoped()` 和 `onDestroy()` 等，方便开发人员在 *Activity* 的状态发生变化时对程序的运行时数据和应用状态做适当的处理操作。对应的 *Activity* 的生命周期具体如图 2.2 所示：

当用户点击应用图标，系统启动应用程序后，系统会创建 *Activity*、启动 *Activity* 并使之可以和用户进行交互。在这个过程中，`onCreate()`、`onStart()`、`onResume()` 等方法被回调，*Activity* 最终处于运行态；

当用户点击“返回键”返回到桌面时，*Activity* 会失去焦点，在用户的视野中消失，直至被系统回收，对应的状态也从运行态经暂停态、停止态，变为终止态变，期间 `onPause()`、`onStop()`、`onDestroy()` 等方法被回调；

当用户从一个界面回到原来的界面时，原有的 *Activity* 从停止态重启，先后出现在设备界面上，获得和用户交互的焦点，期间 `onRestart()`、`onStart()`、`onResume()` 等方法被回调；

当一个 *Activity* 长期处于停止态，但由于内存原因被系统回收时，用户尝试启动它时，系统会像启动一个新的 *Activity* 一样启动它。

### 2.3 Android 中的多线程交互

Android 系统在架构设计上采用了事件驱动架构。在多线程并发访问时，若 UI 控件对于各线程均是可见的，并发对控件做读写操作会使控件处于不可预期的状态；若贸然对控件使用锁机制，访问控件的各个线程之间存在竞争关系，阻塞相关线程业务逻辑的执行，使得应用变得复杂低效。为了避免此类低效率问题，Android 系统在设计事件驱动架构时，采用了单线程的消息队列，即只允许在 UI 线程（也称为主线程，Main Thread）进行界面更新操作，不允许在其他线程（也称为工作线程，Worker Thread）进行界面更新操作。另外，为了保证应用程序界面渲染和事件响应的及时性，任何在主线程上产生的耗时操作（例如加载磁盘上的图片、网络请求等）都是不被允许的。换而言之，当应用程序需要进行耗时操作时，这个操作往

往会在一个新的线程中执行，而不是在主线程中。

Android 系统架构中的单线程消息队列以及主线程的非阻塞性，使得界面更新和耗时操作分散在不同的线程中。只因如此，多线程交互在 Android 应用程序开发过程中十分常见。从整体上，系统提供的交互方式分为两种：基于 Java 的多线程交互和基于 Handler 的交互方式 [? ]。

### 2.3.1 基于 Java 的多线程交互

由于 Android 系统提供的 API 接口兼容 Java 多线程相关的部分 API，因此，在 Android 系统中，开发人员可以采用和 Java 应用相同的调用方式启动工作线程，并在对应的线程上完成业务逻辑。但是，Java API 只能实现业务逻辑从原有线程转移到新的工作线程上，不能重新返回到主线程上。为此，Android 系统在 Java API 的基础上还提供了 API `void runOnUiThread (Runnable runnable)`。该 API 可以帮助开发人员将业务逻辑的执行从工作线程转移到主线程上，该 API 也符合 Android 只允许在主线程上更新界面这一基本设计原则。但是，该 API 也存在着一些弊端，例如 `runOnUiThread(Runnable)` API 的定义位于类 `android.app.Activity`，这也就意味着在 Android 组件 Service 中进行耗时操作时，无法通过该 API 返回到主线程；同时基于接口的函数参数定义方式对于跨线程的参数传递也不是十分友好。为此，Android 提供了基于 Handler 的多线程交互方式。

### 2.3.2 基于 Handler 的多线程消息调度

为了满足开发人员多样化的业务在多线程间的切换，Android 提供了基于 Handler 消息调度的多线程交互方式。

图 2.3 为关于 Handler 的使用示例。图中对应的场景为应用由于业务需要获取一个字符串，并将这个字符串展现在用户界面上。考虑到获取字符串的过程比较耗时，可能会阻塞主线程的相关业务，因此我们会在工作线程执行这部分逻辑（如图 2.3-左所示），而在主线程执行信息展现的逻辑（如图 2.3-右所示）。具体的，在工作线程中，用户生成的字符串和对应的逻辑代码 `MSG_CHANGE_TEXT` 封装到

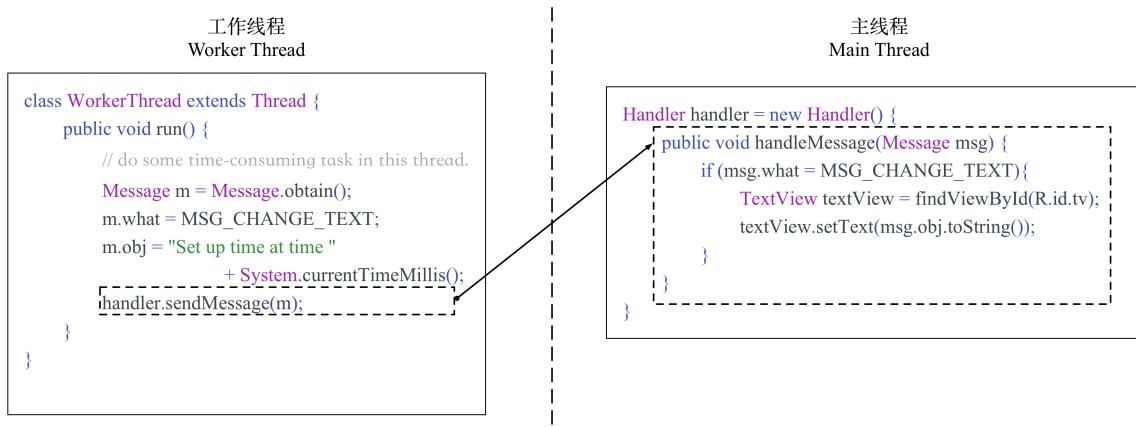


图 2.3: Handler 的使用实例

*Message* 对象中，通过 Handler 发送出去；Handler 在主线程中收到了该 *Message* 对象时，会根据逻辑代码（即 *Message.what* 的值）决定进行对应的逻辑处理（本例中，对应逻辑为更新界面信息）。

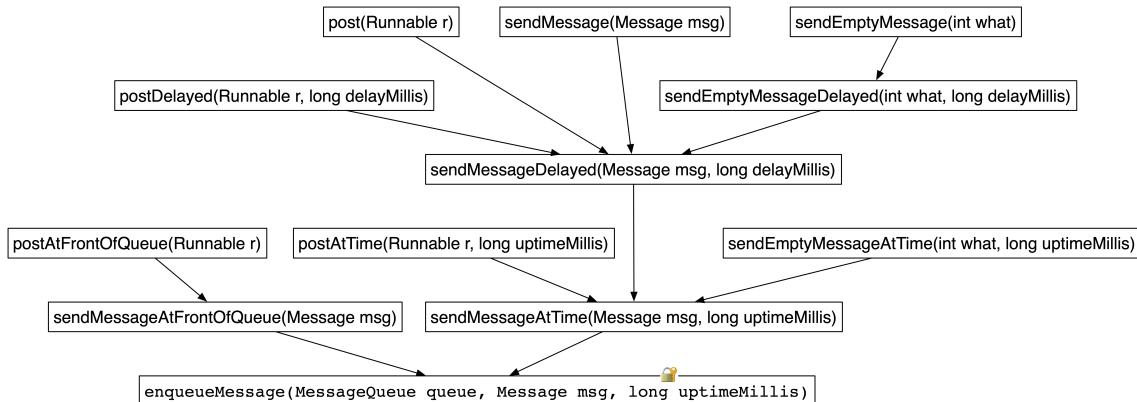


图 2.4: Handler 各 API 之间的调用关系

除了示例中的调用方式，Android SDK 提供的 API[?] 还提供多种 API，同时支持基于 *Runnable* 的消息调度和基于逻辑代码的消息调度：开发人员可以通过 *post(Runnable)*，*postAtTime(Runnable, long)*，*sendMessage(Message)*，*sendMessageDelayed(Message, long)*，*sendMessageAtTime(Message, long)* 和 *sendEmptyMessage(int)* 等多种 API 形式实现消息调度。通过分析 Android 系统相关源代码，我们发现上述 Handler 相关的 API 关系如图 2.4 所示。从图 2.4 中，我们可以发现所有的 API 最后就会调用到 Handler *enqueueMessage(MessageQueue, Message, long)* 方

法。

Handler 机制主要由 Handler、Looper、MessageQueue、Message 等若干部分组成。Message 是多线程交互的核心载体；考虑到移动设备的硬件限制以及 Message 使用的频繁性，Android 系统通过对对象池对 Message 对象进行管理。MessageQueue 是一个双端队列，存放所有待处理的 Message 对象。Looper 负责消息的分发。Handler 在整个过程中承担着消息的发送者和消费者两个身份，负责将消息发送到对应的 MessageQueue 中以及消费来自 Looper 分发下来的消息。

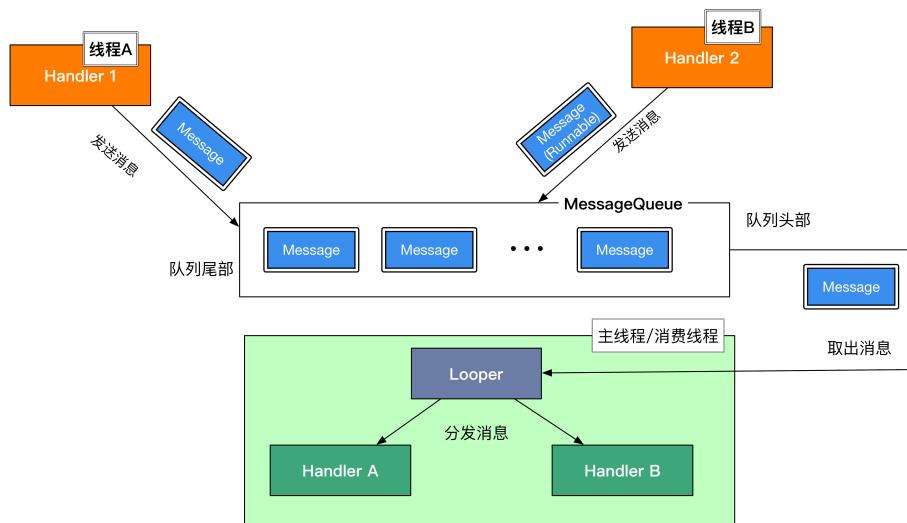


图 2.5: Handler 机制中各组成部分的相互关系

Handler 机制中各组成部分的相互关系如图 2.5 所示。当用户要通过 Handler 传递消息时，用户将调用系统提供的 Handler API，该 API 会将通过分发 *Handler enqueueMessage(MessageQueue, Message, long)* 将消息投递到消息队列 MessageQueue 中；当 Looper 从 MessageQueue 中读取该 Message 对象，分发给对应的 Handler 对象，调用该对象的方法 *Handler.dispatchMessage(Message)*；最终，Handler 对象会调用 *Handler.handleMessage(Message)* 按照 Message 对象进行业务逻辑处理。

基于 Handler 的多线程消息调度，充分利用了 Android 的事件驱动架构，将业务逻辑抽象出 Message 对象，借助消息队列在线程间传递 Message，达到了 Android 多线程交互的目的。Handler 机制，不仅帮助开发人员实现业务逻辑在主线程和工

作线程间的自由转移，而且其灵活的 API 设计降低应用的设计复杂程度，提升了系统架构的可拓展性。正因如此，在 Android 开发过程中，基于 Handler 消息调度的多线程交互十分常见。

## 2.4 本文遇到的困难与挑战

本文解决的关键问题有如下几点：

1) 如何获取应用程序中各个函数的执行信息？

获取应用程序在执行过程中各函数执行信息是本文的基础。从函数分类上，用户定义的方法和系统预定义的方法。对于前者，我们可以通过修改程序源代码实现；但对于后者，由于我们无法直接修改系统程序的源代码或者构建一个符合本文业务需求的系统的成本较高，为此，我们需要寻找对应的解决方案以帮助我们获取系统方法的执行信息。

2) 如何根据程序中各函数的的执行信息，还原应用程序的函数调用图？

基于第一点，我们可以获取程序在运行过程中的执行信息。但仅仅依靠这些执行信息是不够的，我们还需要将这些执行信息组织起来，挖掘执行信息之间的关联关系，从而才能还原成应用程序的函数调用图。这将是本文研究的重点之一。

3) 如何在生成的函数调用图中体现 Android 特性？

正如上文所提的，Android 系统中有很多常规应用中不具备的特性，例如 Activity 的生命周期、基于多线程调用的触发关系。若要在生成的函数调用图上体现上述特性，需要对 Android 系统源代码有一定的了解熟悉，并基于图中的相关信息创建与特性相对应的关系。这既是本文研究的重点，也是本文的创新点。

## 2.5 本章总结

本章主要介绍了 Android 系统的相关背景知识，较为详细的阐述了 Android 的系统结构，详细介绍了 Android 四大组件之一的 Activity 生命周期。同时，本章还介绍了基于 Runnable/Thread、Handler 消息调度两种不同的多线程交互方式，详细

地分析了 Handler 的运行机制，为下文基于函数调用图的多线程触发关系生成做了铺垫。最后，本章还介绍了系统在实现上可能遇到的困难。

## 第三章 名称解释

在本节，我们将给出拓展动态函数调用图构建过程中的基本术语：方法执行、方法对象、调用关系、触发关系、函数调用图、拓展函数调用图等，并将结合具体示例进行解释。

### 3.1 概念说明

方法执行是一个方法执行相关信息的描述，方法对象对应是和方法执行相关的对象；调用关系和触发关系描述了方法执行之间的关系。函数调用图为所有调用关系的集合，在函数调用图上添加方法对象以及触发关系得到拓展函数调用图。相关定义如下：

#### 3.1.1 关于方法和对象的定义

##### 定义 1 方法对象 (*Method Object, MO*)

和方法执行相关的对象称为方法对象，可以体现对象和执行方法的相互关系。在本文中，方法对象通常用符号  $o$  表示。

对于一个方法执行  $m$ ，对象和方法执行的关系有如下几种：

- 参数关系：若对象  $o_p$  是这个方法  $m$  的参数，记为  $o_p \xrightarrow{\text{parameter}} m$ ;
- 返回值关系：若对象  $o_r$  是这个方法  $m$  的返回值，记为  $o_r \xrightarrow{\text{return}} m$ ;
- 实例关系：若方法  $m$  是非静态方法，则方法执行时我们可以获取到关联到的  $this$  指针对象  $o_i$ ，记为  $o_i \xrightarrow{\text{instance}} m$ 。

##### 定义 2 方法执行

方法执行是对方法执行过程中的相关信息的描述，完整的信息包括对应方法的完整签名、执行时所处的线程以及相关的方法对象。在本文中，方法执行通常用符号  $m$  表示。

### 3.1.2 关于方法间关系的定义

#### 定义 3 调用关系 (*Invoke*)

对于程序  $P$  的两个方法  $m_1$  和  $m_2$ ，方法  $m_1$  调用了方法  $m_2$ ，则记作  $m_1 \rightarrow m_2$ ，称为方法  $m_1$  调用方法  $m_2$ 。

在此基础上，对于方法  $m$ ，若存在方法  $m_i$  ( $i = 0, \dots, n, n \geq 0$ )，使得公式 3.1 成立，则记作  $m_0 \xrightarrow{*} m$ ，称为方法  $m_0$  扩展调用方法  $m_n$ 。特殊的，对于方法  $m_1$  和方法  $m_2$ ，若  $m_1 \rightarrow m_2$ ，则  $m_1 \xrightarrow{*} m_2$  也成立。

$$m_0 \rightarrow m_1 \rightarrow \dots m_n \rightarrow m \quad (3.1)$$

**在系统源代码的层面上**，如果对于方法  $m$  和  $m'$ ，方法  $m$  的执行过程总是会调用方法  $m'$ （即  $m \rightarrow m'$  总是成立），可以记为  $m \Rightarrow m'$ ；如果对于方法  $m$  和  $m'$ ， $m \xrightarrow{*} m'$  总是成立，可以记为  $m \xrightarrow{*} m'$ 。

#### 定义 4 触发关系 (*Trigger*)

若方法  $m_a$  和方法  $m_b$  之间同时需要满足以下三个条件，则两个方法存在触发关系，记为  $m_a \hookrightarrow m_b$  或者  $m_a \xrightarrow{\text{因果关系}} m_b$ ，称为  $m_a$  触发调用  $m_b$ ：

- 方法  $m_a$  的执行时间总是在方法  $m_b$  的执行时间之前；
- $m_a \xrightarrow{*} m_b$  不成立；
- $m_a$ 、 $m_b$  之间存在着因果关系，包括但不限于事件回调或多线程交互等。

以多线程中的 `Thread` 为例，方法 `Thread.start()`（记为  $m_{start}$ ）执行会使 JVM / Dalvik 虚拟机创建一个新的线程。最终在这个线程中，虚拟机会在新的线程中回

调该 Thread 对象的方法  $Thread.run()$  (记为  $m_{run}$ )。上述描述可以表示成  $m_{start} \hookrightarrow m_{run}$ 。由于这个触发关系和多线程相关, 也可以记作  $m_{start} \xrightarrow{Thread} m_{run}$ 。同样的, 触发关系也适用于 Ui 事件注册与响应、基于 Handler 的多线程交互。

**在系统源代码的层面上**, 对于方法  $m_a, m_b, m_c$ , 若  $m_a \Rightarrow^* m_b$  和  $m_b \hookrightarrow m_c$  同时成立, 则  $m_a \hookrightarrow m_c$  也成立; 若  $m_a \hookrightarrow m_b$  和  $m_b \Rightarrow^* m_c$  同时成立, 则  $m_a \hookrightarrow m_c$  也成立。

### 3.1.3 关于调用图的定义

#### 定义 5 函数调用图 ( $CallGraph, CG$ )

函数调用图是对程序运行时行为的描述, 用有向图  $CG = (V, E)$  表示。图中的点  $v \in V$  表示一个**方法执行**  $m$ ; 如果方法  $m_1$  调用方法  $m_2$  (即  $m_1 \rightarrow m_2$ ), 则有向边  $e = \langle m_1, m_2 \rangle$  属于集合  $E$ 。

**注意:** 在应用执行过程中, 方法 A 被调用了两次, 方法 A 的每次调用都调用了方法 B, 则对应的函数调用图  $CG$  如公式 3.2 所示。在调用图  $CG$  中,  $m_a$  和  $m_b$  各有两个, 分别对应的两次**方法执行**。 $\langle m_{a_1} \rightarrow m_{b_1} \rangle$  对应的是第一次函数 A 调用函数 B,  $\langle m_{a_2} \rightarrow m_{b_2} \rangle$  对应的是第二次函数 A 调用函数 B,

$$CG = (V, E),$$

$$V = \{m_{a_1}, m_{b_1}, m_{a_2}, m_{b_2}\}, \quad (3.2)$$

$$E = \{\langle m_{a_1} \rightarrow m_{b_1} \rangle, \langle m_{a_2} \rightarrow m_{b_2} \rangle\}$$

#### 定义 6 拓展函数调用图 ( $Extended CallGraph, ECG$ )

在函数调用图 (CG) 的基础上, 添加了方法对象和函数间的触发关系。拓展函数调用图中的节点包括方法执行节点和方法对象节点。图中的边包括描述方法间关系的边和描述方法和对象间的边: 前者的方法间关系包括调用关系和触发关系; 而后者的关系包括和方法对象相关的三个关系。

### 3.2 举例说明

以第二章中的图 2.3 为例，我们将简要阐述上述概念。在线程 WorkerThread 中，方法 `run()` 依次调用了方法 `Message.obtain()` 和方法 `Handler.sendMessage(Message)`，则有  $m_{run} \rightarrow m_{obtain}$  和  $m_{run} \rightarrow m_{send}$ 。对于方法  $m_{obtain}$ ， $o_m \xrightarrow{return} m_{obtain}$  成立。对于方法  $m_{send}$ ， $o_m \xrightarrow{parameter} m_{send}$ 、 $o_{handler} \xrightarrow{instance} m_{send}$  成立。通过对 Android Handler 运行机制的分析，我们知道  $m_{send} \xrightarrow{*} m_{enqueue}$ 、 $m_{enqueue} \leftrightarrow m_{dispatch}$  以及  $m_{dispatch} \xrightarrow{*} m_{handle}$ ，因此， $m_{send} \leftrightarrow m_{handle}$  或  $m_{send} \xrightarrow{Handler} m_{handle}$ 。

### 3.3 本章总结

本章从方法和对象的关系、方法间关系、调用图等几个方面对方法关系、方法执行、调用关系、触发关系、函数调用图、拓展调用图等概念做了符号化的定义，并结合第二章的 Handler 例子简单阐述了上述概念。

## 第四章 RunDroid 的系统设计

为了了解 Android 应用程序在运行阶段的执行过程，本文提出了一个可用于还原 Android 应用程序运行时动态函数调用图的工具——RunDroid。RunDroid 利用源程序代码插桩和运行时方法拦截的相结合的方式，捕获应用程序在应用层面和系统层面的方法执行信息，还原方法间的调用关系。在此基础上，RunDroid 利用对象和方法的关系结合具体的触发规则，进而还原出方法间触发关系，在调用图中展现运行过程中的 Android 特性行为。RunDroid 提供的函数调用图从方法调用关系、方法间的触发关系以及方法执行的相关对象信息等多个方面较为全面地刻画了 Android 应用程序的执行过程，可以为应用程序分析提供更为多样、准确的信息。

### 4.1 整体设计框架

图 4.1-左上为一段示例代码：在 main 函数执行时，程序会依次调用 A、B 两个函数，而 B 函数则会调用了 C、D 两个函数。图 4.1-左下则是程序的运行时函数调用图，即 RunDroid 实际的输出产物。程序执行过程可以看做函数调用图（树）的深度优先遍历过程<sup>1</sup>。还原函数调用图的关键点，在于如何在程序执行过程中输出树的遍历序列，并根据遍历序列进行还原函数调用图。

本文采用的方案：RunDroid 通过对源程序（图 4.1-左上）进行预处理，得到包含日志记录功能的运行代码（图 4.1-右上）；程序在函数执行前后可以输出和方法执行相关的日志信息，（图 4.1-右下）；最后，我们根据这些日志信息构建出函数调

<sup>1</sup>如果我们将运行过程中一个方法执行看做动态函数调用图中的一个节点，那么，动态函数调用图本身就是一颗树，而不是一个图。

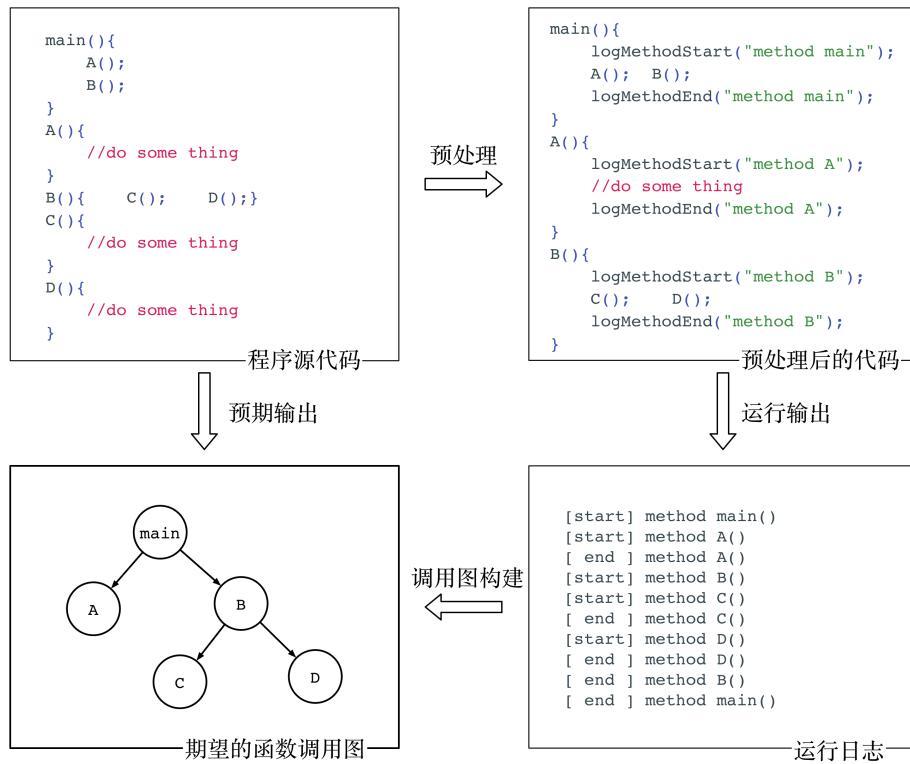


图 4.1: RunDroid 的基本思路

用图（图 4.1-左下）。另外，在函数调用图的基础上，RunDroid 利用日志中包括的方法对象信息，挖掘和方法对象相关联的方法，结合具体触发规则，进而建立方法触发关系。

在技术实现上，RunDroid 主要分为预处理器、运行时拦截器、日志记录器、调用图构建器等 4 个部分，对应的工作流程如图 4.2 所示。从功能上，预处理器和运行时拦截器两者的作用是一致的：在程序运行过程中，当相关方法执行时，触发日志记录器记录相应的信息。预处理器通过源代码插桩技术实现在用户方法运行时触发用户方法的信息记录，而运行时拦截器则是通过方法劫持技术实现对系统方法执行的拦截，进而触发日志记录。在应用运行时，日志记录器会以日志的形式将用户方法和系统方法对应的执行信息记录下来。最后，调用图构建器会根据应用程序运行时输出的日志，构建拓展函数调用图。

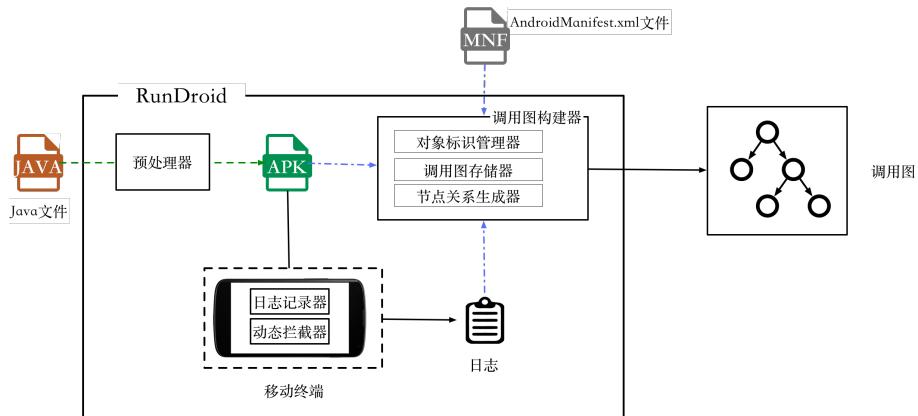


图 4.2: RunDroid 的工作流程

## 4.2 方法信息的捕获

在 RunDroid 中，方法一共分为两类，用户方法和系统方法。用户方法是由用户定义的，直接修改的成本较低，易于实现。而系统方法在系统中定义，属于系统运行环境的一部份，行为修改成本较高。虽然记录这两种方法的执行信息的方式是不一样的。但是两者最终的效果是一致的：当应用程序运行时，相关方法的执行信息均会通过日志记录器记录下来，用于后续调用图的构建。

### 4.2.1 用户方法执行信息的获取

预处理器以 Android 项目源代码作为输入，输出预处理后的 APK 文件。具体处理过程如算法 4.1 所示：预处理器会遍历项目源代码中的 Java 源文件，将源文件和对应的抽象语法树转化成 XML 格式的文件，并以 DOM 的形式加载到程序中（第 1~4 行）。对于抽象语法树中的每一个类和每一个方法，预处理器会计算出编译后所处的类的全限类名及方法签名（第 5~8 行）。全限类名和方法签名的组合 <全限类名, 方法签名> 可以作为方法体的唯一标识，和抽象语法树中的方法体一一对应。在每个方法内部，预处理器会插入用户方法日志记录的代码，实现日志代码编织，达到用户方法运行时信息记录的目的（第 9~11 行）。另外，预处理器还对每个方法体进行了异常捕获处理，防止方法体内部的异常导致日志记录过程的中断，影响函数调用图的构建（第 10 行）。上述操作都是对 DOM 对象 *document* 直接操作

的。当一个 Java 文件修改完毕后，预处理器会将 DOM 对象重新转换成 Java 源文件，并替换原来的文件（第 12 行）。最后，预处理器会利用编织后得到的源代码构建成 APK 文件（第13行）。

---

#### 算法 4.1: 日志代码插桩过程

---

**输入:** *javaFiles*, 应用程序的源代码

**输出:** *apk*, 包含插桩代码的 APK 文件

```

1 Function buildApk(log):
2   for javaFile ∈ javaFiles do
3     transform javaFile to xml-format File xmlFile
4     transform xmlFile to DOM Object document
5     for classEle ∈ document do
6       className ← computeClassSign(classEle,xmlFile);
7       for methodEle ∈ classEle do
8         methodId ← computeMethodId(methodEle,className);
9         addMethodExitLogCode(methodEle,methodId);
10        addMethodCatchLogCode(methodEle,methodId);
11        addMethodEnterLogCode(methodEle,methodId);
12       write document into new Java File newJavaFile
13     apk ← buildApk(newJavaFile)
14   return apk;

```

---

#### 4.2.2 系统方法执行信息的获取

运行时拦截器的主要职责是对 Android 系统方法的执行进行拦截，将相关信息传递给日志记录器并记录下来。运行时拦截器维护着系统方法列表。列表中的方法通常和 Android 特性相关，例如 Activity 的生命周期方法、Java Thread API 以及 Handler 相关的 API，具体如表 4.1 所示。在目标应用程序开始运行前，运行时拦截器会为上述目标方法注册执行回调函数。在目标方法执行前和执行后，对应的回调函数会被执行，进而通过日志记录器记录下这些方法执行的相关信息。在 RunDroid 中，运行时拦截器可以弥补预处理器无法提供系统方法执行信息的不足，

填补调用图缺失的系统方法执行，进而还原出应用层和系统层之间以及系统内部的方法调用，使得产生的调用图更加完整。

表 4.1: 运行时拦截器拦截的系统方法列表

方法签名	说明
Activity.onCreate(Bundle)	
Activit.onStart()	
Activity.onResume()	和 Activity 生命周期相关的方法
Activity.onPause()	
Activity.onStop()	
Activity.onDestroy()	
Thread.start()	和 Java 线程启动相关的方法
Message.obtain()	
Handler.enqueueMessage(MessageQueue,Message,long)	
Handler.dispatchMessage(Message)	
Handler.post(Runnable)	
Handler.postAtTime(Runnable,long)	
Handler.postAtTime(Runnable,Object,long)	
Handler.postDelayed(Runnable,long)	
Handler.postAtFrontOfQueue(Runnable)	和 Handler 机制相关的方法
Handler.sendMessage(Message)	
Handler.sendEmptyMessage(int)	
Handler.sendEmptyMessageDelayed(int,long)	
Handler.sendEmptyMessageAtTime(int,long)	
Handler.sendMessageAtFrontOfQueue(Message)	
Handler.sendMessageDelayed(Message,long)	
Handler.sendMessageAtTime(Message,long)	

### 4.3 扩展函数调用图的构建过程

调用图构建器构建拓展函数调用图的过程分为如下几个阶段：RunDroid 根据程序运行时的日志提取函数间的调用关系，创建函数调用图；根据 AndroidManifest.xml 的 Activity 组件声明，在函数调用图上标识出完整的 Activity 生命周期流转序列；利用方法执行与方法对象的关联关系，结合触发关系规则，补全方法间的触发关系，形成最终的拓展函数调用图。

---

**算法 4.2:** 函数调用图的构建过程

---

**输入:**  $logs$ , 应用程序的运行时日志

**输出:**  $cg$ , 函数调用图

```

1 Function buildCallGraph( $log$ ):
2    $cg \leftarrow \text{new CallGraph}();$ 
3   for  $thread \in logs.\text{threads}$  do
4      $stack \leftarrow \text{new Stack}();$ 
5     for  $log \in logs.\text{get}(thread)$  do
6        $top = stack.\text{peek}();$ 
7       if  $\text{isMethodStartLog}(log)$  then
8          $m \leftarrow \text{generateMethodInfo}(log);$ 
9          $cg.addMethodNode(m);$ 
10         $cg.addMethodObjectsIfNotExists(o_p, o_i);$ 
11         $cg.addMethodObjectRels(\langle o_p \xrightarrow{\text{parameter}} m \rangle, \langle o_i \xrightarrow{\text{instance}} m \rangle);$ 
12        if  $top \neq null$  then
13           $cg.addInvokeRel(\langle top \rightarrow m \rangle);$ 
14           $stack.push(m);$ 
15        else
16           $cg.addMethodObjectIfNotExists(o_r);$ 
17           $cg.addMethodObjectRel(\langle o_r \xrightarrow{\text{return}} m \rangle);$ 
18           $stack.pop();$ 
19      return  $cg;$ 

```

---

#### 4.3.1 构建函数调用图

由于各个线程在执行过程中, 不存在一个调用关系跨越两个线程, 因此, 在整个构建过程中, RunDroid 以产生日志的线程为基本构建单元, 向调用图添加方法调用关系。函数调用图的构建过程如算法 4.2 所示。

对于每个线程, RunDroid 顺序遍历对应的日志, 使用栈  $stack$  的入栈、出栈操作来模拟对应线程的函数执行的过程, 还原调用关系 (第 2~20 行): 当读取到方法执行的开始日志时, 系统会在调用图创建一个节点表示该方法的执行 (第 7~8

行), 同时也在调用图中添加方法参数、方法实例对应的对象节点  $o_p$ 、 $o_i$  以及方法对象和方法的关系  $\langle o_p \xrightarrow{\text{parameter}} m \rangle$ 、 $\langle o_i \xrightarrow{\text{instance}} m \rangle$  (第 9~10 行)。如果此时当前线程栈  $stack$  的栈顶方法元素  $top$  存在, 系统会创建从方法  $top$  到当前方法  $m$  的调用关系,  $\langle top \rightarrow m \rangle$ , 并将当前方法  $m$  压入栈  $stack$  中 (第 11~14 行)。当读取到方法执行的结束日志时, 该日志对应的方法必然是栈顶方法  $top$ , 若栈顶方法  $top$  存在返回对象  $o_r$ , 则只需要将  $o_r$  和  $top$  的关系添加到调用图中即可, 最后弹出栈顶的  $top$  即可 (第 16~18 行)。在上述过程中, 如果待添加的方法对象在调用图中已经存在时, 该对象则无须重新添加, 只需添加方法与对象间的关系即可。

### 4.3.2 构建 Activity 的生命周期和事件回调

#### § 构建 Activity 的生命周期

Android 应用的 Activity 生命周期的构建就是将 Activity 生命周期方法按照时间顺序串联起来, 利用的是 Activity 生命周期方法的签名的不变性。

整个过程以 AndroidManifest 文件作为输入, 在原有的函数调用图的基础上, 添加表示 Activity 生命周期的有向边, 具体过程如算法 4.3 所示: 对于 AndroidManifest 文件中声明的每一个 Activity 组件  $o_{activity}$ , 系统都会遍历以该对象为方法实例的方法  $m$  (即  $m$ 、 $o_{activity}$  满足条件  $o_{activity} \xrightarrow{\text{instance}} m$ ); 若方法  $m$  同时满足三个条件, 则将它添加到列表  $lifecycleList$  中 (第 3~8 行)。最后, 将列表  $lifecycleList$  按照时间顺序进行排序, 并依次连接起来, 即可得到 Activity 的生命周期 (第 9~10 行)。

这三个条件分别为: (1) 方法  $m$  的方法签名和图 2.2 中的任何一个生命周期方法的签名保持一致; (2) 方法  $m$  执行时所处的线程为主线程; (3) 方法  $m$  在调用图  $cg$  中不在调用者, 即在调用图  $cg$  中不存在方法  $m' \in cg$ , 使得  $m' \rightarrow m$  成立。

#### § Android 事件回调的触发关系

### 4.3.3 构建多线程触发关系

基于函数调用图构建多线程触发关系主要分为两个方面, 基于 Java 的多线程交互与基于 Handler 的多线程消息调度, 具体的过程如算法 4.4 所示。

---

**算法 4.3: 构建 Activity 的生命周期和事件回调**

---

输入: *manifestFile*, AndroidManifest 文件  
 输入: *cg*, 函数调用图  
 输出: *cg'*, 包括 Activity 生命周期的函数调用图

```

1 Function patchActivityLifecycleAndUiEvent(cg):
2   patchActivityLifecycle(cg,manifestFile);
3   patchUiEvent(cg);
4   cg' ← cg;
5   return cg';
6 Function patchActivityLifecycle(cg,manifestFile):
7   lifecycleList ← new List();
8   for oactivity ∈ {activity | activity defined in manifestFile} do
9     for m ∈ {m | m is method node in cg and oactivity is instance of m} do
10       if m is the method of Activity's Lifecycle
11         and Thread for m is MainThread
12         and  $\langle m' \rightarrow m \rangle \notin cg$  then
13           lifecycleList.add(m);
14   sortListByTime(lifecycleList);
15   linkItemsByTime(lifecycleList,cg);
16 Function patchUiEvent(cg):
17   for olistener ∈ {o | o is the View.OnClickListener object node ,o ∈ cg} do
18     if  $\langle m_{register} \xrightarrow{\text{instance}} o_{\text{listener}} \rangle \in cg$ 
19       and  $\langle m_{click} \xrightarrow{\text{instance}} o_{\text{listener}} \rangle \in cg$  then
20         cg.addTriggerRel(mregister ↲UiEvent mclick)

```

---

## § 基于 Java 的多线程交互

基于 Java 的多线程交互往往是以 *Runnable* 作为传递对象, 通常通过调用方法 *Thread.start()* (用 *m<sub>start</sub>* 表示) 和 *Activity.runOnUiThread(Runnable)* (用 *m<sub>runOnUiThread</sub>* 表示) 等 API, 进而触发方法 *Runnable.run()* (用 *m<sub>run</sub>* 表示) 的执行。因此, 对于方法 *Thread.start()*, 如果存在一个 *Runnable* 类型的对象 *r*, 它既是方法 *m<sub>start</sub>* 的

---

**算法 4.4: 扩展函数调用图的构建过程**

---

输入:  $cg$ , 函数调用图

输出:  $ecg$ , 拓展函数调用图

```

1 Function buildExtendedCallGraph( $cg$ ):
2    $ecg \leftarrow cg$ ;
3   addThreadTrigger( $ecg$ );
4   addHandlerTrigger( $ecg$ );
5   return  $ecg$ ;
6 Function addThreadTrigger( $ecg$ ):
7   for  $o_r \in \{o \mid o \text{ is the } Runnable \text{ object node}, o \in ecg\}$  do
8     if  $\langle m_{start} \xrightarrow{\text{instance}} o_r \rangle \in ecg$ 
9       and  $\langle m_{run} \xrightarrow{\text{instance}} o_r \rangle \in ecg$  then
10       $ecg.addTriggerRel(m_{start} \xleftarrow{\text{Thread}} m_{run})$ 
11      if  $\langle m_{runOnUiThread} \xrightarrow{\text{parameter}} o_r \rangle \in ecg$ 
12        and  $\langle m_{run} \xrightarrow{\text{instance}} o_r \rangle \in ecg$  then
13           $ecg.addTriggerRel(m_{runOnUiThread} \xleftarrow{\text{runOnUiThread}} m_{run})$ 
14 Function addHandlerTrigger( $ecg$ ):
15   for  $o_m \in \{o \mid o \text{ is the } Message \text{ object node}, o \in ecg\}$  do
16     if  $\langle m_{enqueue} \xrightarrow{\text{parameter}} o_m \rangle \in ecg$ 
17       and  $\langle m_{dispatch} \xrightarrow{\text{parameter}} o_m \rangle \in ecg$  then
18        $m_{send} = calculateSendMethod(ecg, m_{enqueue});$ 
19        $m_{handle} = calculateHandleMethod(ecg, m_{dispatch});$ 
20        $ecg.addTriggerRel(m_{send} \xleftarrow{\text{Handler}} m_{handle})$ 
```

---

实例, 又是方法  $m_{run}$  的实例, 则两个方法间存在触发关系, 即  $m_{start} \hookrightarrow m_{run}$  (第 8~10 行)。同样的, 对于方法  $Activity.runOnUiThread(Runnable)$ , 也存在类似的关系: 如果存在一个  $Runnable$  类型的对象  $r$ , 它既是方法  $m_{runOnUiThread}$  的参数, 又是方法  $m_{run}$  的实例, 则两个方法间存在触发关系, 即  $m_{runOnUiThread} \hookrightarrow m_{run}$  (第 11~13 行)。

### § 基于 **Handler** 的多线程消息调度

根据第二、三章的介绍，我们已经知晓用户通过调用 Handler 提供的 API，将相关业务逻辑借助 Message 对象传递给目标线程的 Handler 对象，在目标线程执行相应的业务逻辑处理。首先，我们利用类 Handler 的方法 *enqueueMessage(Message)*（用  $m_{enqueue}$  表示）和 *dispatchMessage(Message)*（用  $m_{dispatch}$  表示）公用同一个 Message 对象的特点，找到所有的 Handler 底层函数触发关系， $m_{enqueue} \leftrightarrow m_{dispatch}$ （第 16~17 行）；对于每一个触发关系  $m_{enqueue} \rightarrow m_{dispatch}$ ，从  $m_{enqueue}$  顺着调用关系往上找到最上层的 Handler API 方法（即用户调用的 Handler API 方法， $m_{send}$ ）（第 18 行），从  $m_{dispatch}$  顺着调用关系往下找到用户定义的方法 *Handler.handleMessage(Message)*  $m_{handle}$ （第 19 行），最后在调用图中提交方法  $m_{send}$  和  $m_{handle}$  之间的触发关系  $m_{send} \xrightarrow{\text{Handler}} m_{handle}$ （第 20 行）。

## 4.4 本章总结

本章详细介绍了 RunDroid 的设计。首先，本章简要介绍了 RunDroid 实现的基本功能，并通过一个简单的例子介绍了 RunDroid 的整体运行框架。由此，我们提出了 RunDroid 技术路线：RunDroid 利用源代码插桩和运行时拦截器相结合的方式，将运行时的方法执行信息以日志的形式保存下来，通过线程内部各个方法日志的嵌套关系还原出函数调用图，并在此基础上构建 Activity 的生命周期，补全方法间的触发关系，形成最终的拓展函数调用图。

## 第五章 RunDroid 的系统实现

RunDroid 使用 Java 作为开发语言，使用了 srcML、Xposed、Neo4j 等技术，由预处理器、运行时拦截器、日志记录器、调用图构建器等部分组成。本章将详细的介绍上述技术及 RunDroid 各组成部分的实现细节。

### 5.1 相关技术介绍

#### 5.1.1 srcML

srcML [?] 是轻量级源代码分析工具，它可以将程序的抽象语法树以 XML 的形式展现给用户，支持 C、C++、C# 以及 Java 等多个语言的语法解析。通过 srcML 的语法解析器解析得到的 XML 内容保留了源代码中完整的语法树信息。在 RunDroid，srcML 作为 RunDroid 预处理器中的重要组成部分，承担源代码语法解析的主要职责，辅助完成用户方法层面的日志代码的编织。

#### 5.1.2 Xposed 框架

Xposed [?] 是一个基于 Android 系统的运行时行为修改框架。在不修改程序源代码的情况下，基于 Xposed 开发的第三方插件通过将目标方法关联到函数执行回调上，进行方法参数和返回值的重写和额外方法逻辑的添加等操作，达到目标方法行为修改的目的。相比定制化 Android 系统，Xposed 同时兼容大部分 Android 系统版本，实现成本低。利用 Xposed 提供的类似面向切面编程的 API 接口，RunDroid 中的运行时拦截器可以对任意方法（包括用户方法和系统方法）执行过程进行动态拦截，达到系统方法信息的日志记录的目的。

### 5.1.3 Neo4j

Neo4j [?] 是基于 Java 语言开发的图数据库，可以用于存储图结构相关的数据结构。Neo4j 的数据主要分为节点和关系两大类，分别对应图论中的点与边。Neo4j 支持在关系和节点上添加自定义键值属性，为节点指定标签，为关系指定类型等等。在数据操作接口方面，Neo4j 支持 Cypher、Java API 和 RESTful API 等多种方式，同时也提供了友好的数据浏览界面用于数据的展示与修改。在 RunDroid，Neo4j 是调用图构建器的主要组成部分，主要承担着拓展函数调用图的存储、查询、展示的职责。

## 5.2 模块实现

### 5.2.1 预处理器

在 chapter 四中，我们提到预处理器在 RunDroid 中的作用主要是日志代码编织。在现有的技术上，代码编织主要分为两类：基于源代码的代码编织和基于字节码的代码编织。通常采用的是字节码编织技术，例如 Aspect、Emma 等。但是，这种字节码插桩技术在运行过程中会生成过多的方法数，在 Android 应用构建过程中可能存在方法数 65K 限制问题。为此，预处理器采用的方案是基于源代码的代码编织方案。该方案通过修改源程序，将日志记录代码直接写入在方法体内部，避免了新方法的引入，在一定程度上避免了方法数 65K 限制问题。

### 5.2.2 运行时拦截器

在实现上，运行时拦截器是基于 Xposed 的插件，它维护的列表包括了所有需要拦截的系统方法（下称为目标方法）。每当应用程序启动时，运行时拦截器通过 Xposed 提供的 API *XposedHelpers.findAndHookMethod()* 将表 4.1 中的目标方法绑定到方法钩子（即类 HookCallBack）上。在应用程序的运行过程中，HookCallBack 对象的 *beforeHookedMethod(MethodHookParam)* 会在目标方法执行之前被调用，*afterHookedMethod(MethodHookParam)* 会在目标方法执行后调用。最终，HookCallBack

对应会将方法执行的相关信息传递给日志记录器。

### 5.2.3 日志记录器

日志记录器的职责是将方法执行的消息以文件的形式持久化下来。针对不同的方法类型（静态方法与非静态方法、用户方法与系统方法等），日志记录器提供了不同的 API，帮助我们记录相应方法执行的日志信息。在日志内容上，日志记录器还会记录每个方法执行的所处线程、方法签名标识、所处阶段（方法开始执行阶段/方法执行完毕阶段）以及相关的方法对象信息。方法对象信息主要包括对象的类型、属性和全局 ID（通过 Java API `System.identityHashCode(Object)` 获取）等。同时，日志记录器还支持对象信息的自定义：开发人员可以根据自身的需要为不同类型的对象输出不同对象数据信息。

### 5.2.4 调用图构建器

在实现上，调用图构建器如图 4.2 所示，主要有以下几个部分组成：调用图存储器、对象管理器以及触发关系生成器。

调用图存储器将 Neo4j 作为调用图的底层存储引擎，调用图中的方法和对象会以节点的形式出现在调用图中，采用 Neo4j 中的关系表示两者之间的边。

对象管理器负责将日志中的对象信息映射成调用图中的节点。通常情况下，我们认为一个对象和一个全局 ID 一一对应，所以，我们将全局 ID 相同的对象信息映射成调用图中的一个节点。但考虑到有些类型（例如 Handler 机制中的 Message）使用对象池技术，我们还在全局 ID 的基础上引入对象版本号，避免对象在不同生命周期的串用。以 Message 为例，调用图构建过程中，如果对象管理器发现待提交的 Message 对象  $m$  是方法 `Message.obtain()` 的返回值，处理逻辑如下：如果调用图中不存在一个全局 ID 和  $m$  一致的节点，则对象管理器会创建一个全新的节点来表示  $m$ ，设置其对象版本号为 1；而调用图已经存在一个全局 ID 和  $m$  一致，版本号为  $version$  的节点  $m'$ ，则对象管理器会创建一个全新的节点来表示  $m$ ，并将  $version + 1$  作为节点  $m$  的版本号，而不是复用原有节点  $m'$ 。

触发关系生成器是第四章中算法 4.4 的具体实现，基于 Cypher 脚本和 Soot 完成。例如，算法 4.4 第 16~17 行中，我们需要找到使用  $o_m$  作为参数的方法  $m_{enqueue}$ 、 $m_{dispatch}$ ，并建立两者之间的触发关系。传统的实现方式需要对图进行遍历，效率较低。而 RunDroid 在实现上采用了 Cypher 脚本，如图 5.1 所示。Soot 的工作主要是提供应用程序相关的类型查询服务。例如，算法 4.4 第 7 行中，我们需要知道所有的 Runnable 对象，此处便需要通过 Soot 查询所有的 Runnable 子类。另外，在 4.3 中，我们也会将 Soot 和 Cypher 相结合用于查找所有 Activity 的生命周期方法。

---

```
MATCH (m_enqueue:METHOD)-[ :PARAM ] -> (o_m:OBJECT) ,  
      (m_dispatch:METHOD)-[ :PARAM ] -> (o_m:OBJECT)  
return o_m, m_enqueue, m_dispatch
```

---

图 5.1：使用 Cypher 查找共用对象  $o_m$  的方法  $m_{enqueue}$ 、 $m_{dispatch}$

### 5.3 本章总结

本章侧重介绍 RunDroid 的技术实现。本章先对 RunDroid 使用到的技术(srcML、Xposed、Neo4J) 做了简要的介绍，并介绍了这些技术在 RunDroid 中的主要职责。并在此基础上，我们介绍了 RunDroid 中预处理器、运行时拦截器、日志记录器、调用图构建器等模块的实现细节。

## 第六章 系统测试

上一章详细介绍了 xxxx，其中包含 xxx、xxx 各功能模块。为了 xxxx，本章节中，通过 xxx 来展示、评估系统。实验相关配置的环境如下：PC 端对应的型号是 ThinkPad E430，对应的 CPU 型号为 Intel 酷睿 i5 3210M（双核），内存为 12GB。手机型号为小米 MI 5，处理器型号为骁龙 820，RAM 为 3GB，对应的系统版本为 Android 6.0，内置 Xposed 运行环境。

### 6.1 应用结果展示

在本节中，我们会对 RunDroid 运行结果做出相应的展示，并将展现结果和静态分析工具 FlowDroid 做对比，比较两个工具的产出结果，分析各自的优劣。

我们研究的 APK 的主体代码如图 6.1 所示。在图 6.1 中，我们声明了有 Activity *MainActivity*，他是应用层的主 Activity，该 Activity 界面主要有 3 个按钮组成，方便对应的是斐波拉契数量的计算、启动一个 Activity（生命周期相关）以及基于 Handler 的异步事件。

#### 6.1.1 函数调用图的构建结果展示

在应用程序运行时，点击按钮 button1，应用会计算斐波拉契数列中的第 5 项，并将这个数以 Toast 的形式展示给用户。上述过程中，方法调用同时涉及到普通方法调用、递归函数调用。RunDroid 的动态分析结果如图 6.2 所示：每一个红色节点对应的一次方法执行，每一个绿色节点对应是一个对象；如果对象是方法执行的方法对象，则在图中会有一条边从方法指向该对象，并在边上标识两种之间的关系（参数关系、返回值关系、实例关系等）；如果两个方法执行之间存在调用关

```
1 package cn.mijack.rundroidtest;
2
3 public class MainActivity extends Activity implements View.OnClickListener {
4     Button button1, button2, button3;
5     Handler handler = new Handler() {
6         public void handleMessage(Message msg) {
7             if (msg.what == 1) {
8                 Toast.makeText(MainActivity.this, "handle", Toast.LENGTH_SHORT).show();
9             }
10        }
11    };
12    protected void onCreate(Bundle savedInstanceState) {
13        super.onCreate(savedInstanceState);
14        setContentView(R.layout.activity_main);
15        button1 = findViewById(R.id.button1);
16        button1.setOnClickListener(this);      // button2、button3进行相同的操作，此处省略
17    }
18    public void onClick(View view) {
19        switch (view.getId()) {
20            case R.id.button1:
21                doHandleButton1();
22                return;          // button2、button3进行相同的操作，此处省略
23            }
24        }
25        public void doHandleButton1() {
26            int fibonacci = doFibonacci(4);
27            Toast.makeText(this, "fibonacci: " + fibonacci, Toast.LENGTH_SHORT).show();
28        }
29        private int doFibonacci(int i) {
30            if (i < 1)      return -1;
31            if (i == 1 || i == 2)      return 1;
32            return doFibonacci(i - 1) + doFibonacci(i - 2);
33        }
34        public void doHandleButton2() {
35            Intent intent = new Intent(this, NewActivity.class);
36            startActivity(intent);
37        }
38        public void doHandleButton3() {
39            Thread workerThread = new WorkerThread(handler);
40            workerThread.start();
41        }
42    }
```

图 6.1: MainActivitiy 的代码

系，则他们之间会通过从调用发起方指向被调用方的有向边进行连接。

在本案例中，*doHandleButton1()* 调用了方法 *doFibonacci()*，因此前者有条有向边指向后者；通过观察虚线框中各节点的关系，我们知道，对于方法 *doFibonacci()*

, 当参数为 5 时, 对应的结果为 5。FlowDroid 的静态分析结果如图 6.3 所示。通过比较图 6.2 和图 6.3, 我们可以发现以下有趣的结论:

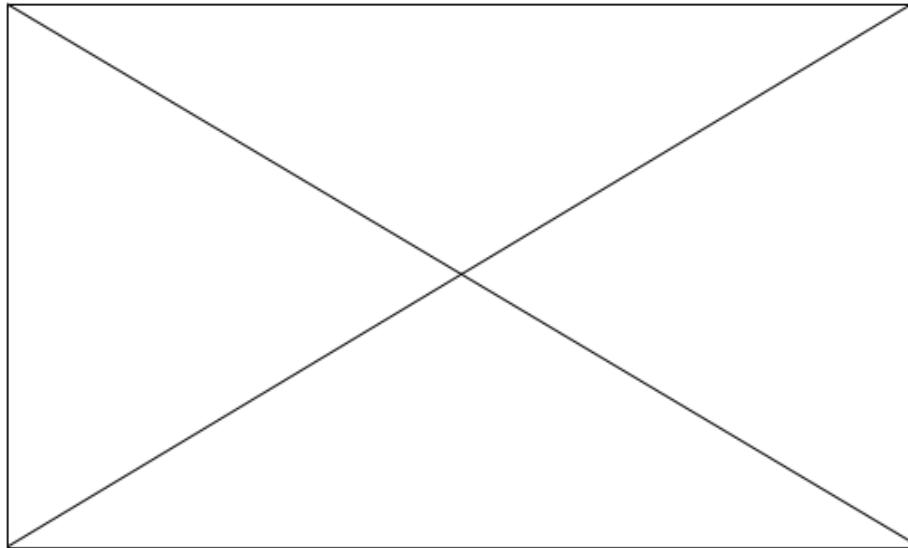


图 6.2: 斐波拉契数列-RunDroid 生成的调用图 (局部)

首先, 两者在方法 *doFibonacci()* 数量上不同的: RunDroid 得到的结果中, 方法节点 *doFibonacci()* 共有 9 个, 即在运行过程中, *doFibonacci()* 被调用了 9 次。由于在一个应用中, 一个方法的方法签名是唯一的, 所以 FlowDroid 给出的结果中 *doFibonacci()* 只有一个节点, 该节点上存在一个指向自己的环, 表示这个方法在执行过程中可能出现递归调用的情况。另外, 静态分析方法 (FlowDroid) 在分析 *doFibonacci()* 这类方法时, 对函数的执行上下文做出准确的判断, 进而给出程序准确的运行时行为。因此, 静态分析技术得到的函数调用图往往以方法体本身作为研究的基本单元, 而动态分析技术的函数调用图可以细化方法执行之间的关系, 在一定程度上可以反映程序执行的具体过程。

其次, 方法 *Toast.makeText(Context,CharSequence,int)* 没有出现在 RunDroid 的结果中, 而 FlowDroid 给出的结果却有展现: 而在运行过程中, 我们却可以看到 Toast 提示。经过分析, 原因如下: *Toast.makeText(Context,CharSequence,int)* 属于系统定义的方法, 而运行时拦截器待拦截的方法列表中并未包含该方法, 因此运行时未收集到相关的方法执行信息, 导致 RunDroid 无法在调用图中还原相应的函

数。这也从一个方面反映了 RunDroid 的劣势：系统函数运行时信息的捕获需要提前将相应的方法告知运行时拦截器，生成的调用图才会包括相应的方法。这使得调用图只包含研究人员关心的方法的执行信息，但也在一定程度上导致部分方法的缺失和调用图的局部不完整。

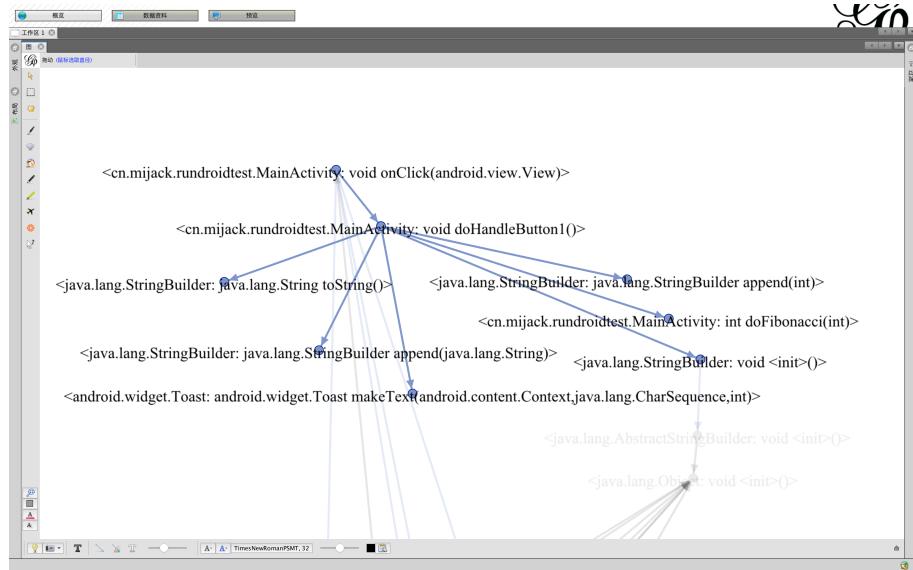


图 6.3: 斐波拉契数列-FlowDroid 生成的调用图（局部）

最后，FlowDroid 给出的结果中包括了大量的 *StringBuilder* 相关的方法，而 RunDroid 的结果并不包括这些方法：对比源码发现，源程序并未直接使用 *StringBuilder*。通过查阅文献 [? ]，我们发现出于提高字符串串联性能的考虑，Java 编译器可以使用类 *StringBuilder* 等技术通过源代码做适当等价的修改，以避免表达式求值过程中产生过多的字符串数量。由于上述过程发生在 Java 程序编译阶段并最后以字节码的形式不存在 APK 文件中，而 FlowDroid 恰好从字节码层面对应用进行分析，因此 FlowDroid 的分析结果会包含该方法。对于 RunDroid，上述方法既在源代码中未出现相关方法定义（无法进行日志代码编织），又不在运行时拦截器的目标方法列表中，所以 RunDroid 给出的调用图自然也不会包括 *StringBuilder* 相关的方法。

### 6.1.2 Activity 生命周期和事件回调的效果展示

在本节中，应用运行时，我们将点击按钮 button2，在 *MainActivity* 启动另一个 *NewActivity*，对比 RunDroid 和 FlowDroid 在 Activity 生命周期方法的呈现效果。最终，我们得到的 RunDroid 的运行结果如图 6.4 所示。针对 Android 组件 *Activity* 的生命周期特性，FlowDroid 进行了针对性的建模，通过构造方法 *dummyMainMethod()* 串联 Android Activity 的生命周期和 UI 事件回调，相应地结果如图 6.5 所示。

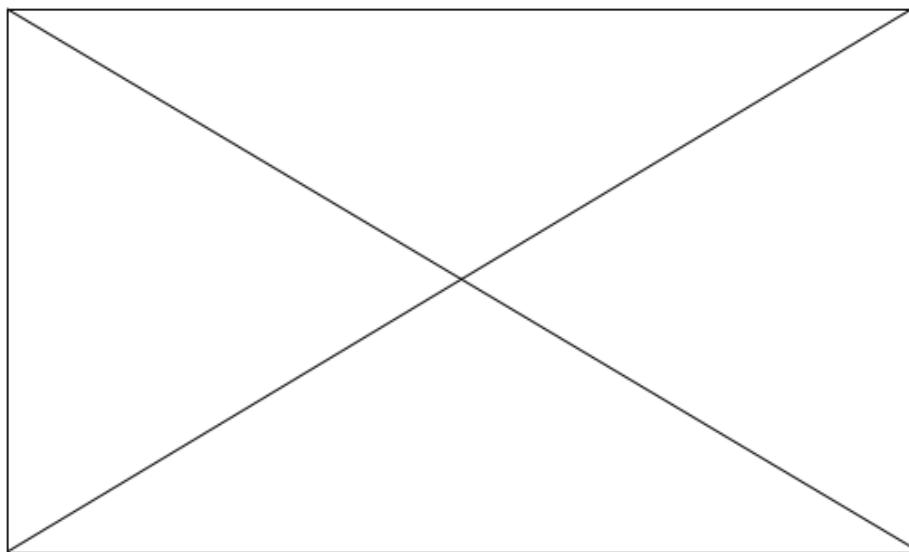


图 6.4: Activity 部分-RunDroid 生成的调用图（局部）

对比图 6.4 和图 6.5，我们可以发现 RunDroid 和 FlowDroid 在组件生命周期和事件回调上的设计思想上是不同的：RunDroid 可以捕获到组件生命周期方法以及回调事件方法的实际执行，因此，RunDroid 对于上述行为的展现主要是按照时间的先后顺序将这些方法节点串联起来，形成完整的事件序列。而 FlowDroid 在生成函数 *dummyMainMethod()* 时，会为 *AndroidManifest.xml* 文件中定义的每一个 *Activity* 单独创建包含生命周期方法和事件回调方法的状态迁移图（图 6.5 中的虚线框部分表示 *Activity* 的整体状态迁移，灰色部分为 *MainActivity* 中的 UI 事件响应方法），最后将这些 *Activity* 的状态迁移串联起来，并将 Action 为 *android.intent.action.MAIN* 并且 category 为 *android.intent.category.LAUNCHER* 的 *Activity* 组件作为结果的默认启动的 *Activity*，最终形成方法 *dummyMainMethod()*。

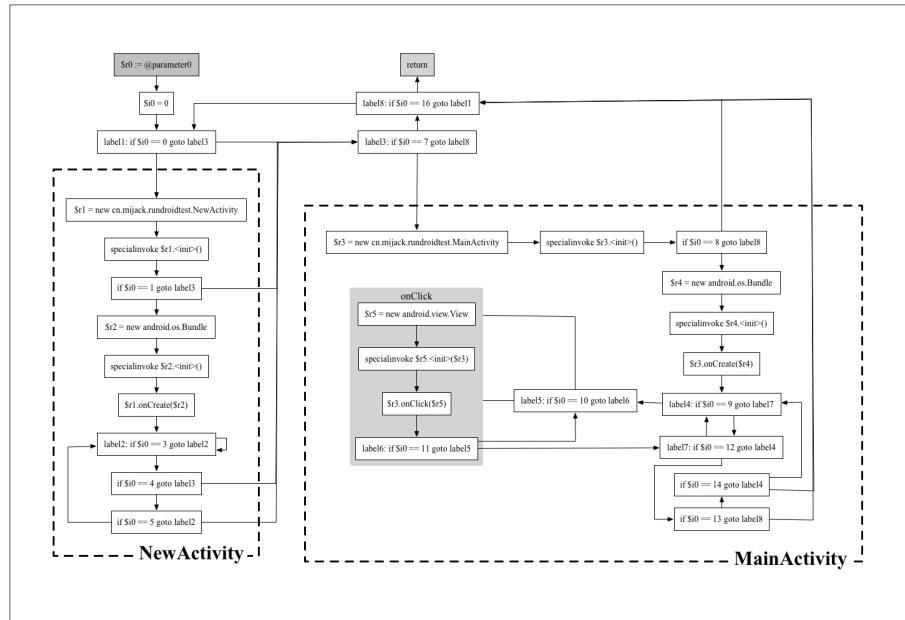


图 6.5: dummyMainMethod-FlowDroid 生成的调用图

在结果展现上，RunDroid 倾向于将和 Activity 生命周期相关的所有方法都关联起来。例如，虽然开发人员在源代码中没有定义 *onResume()* 等方法，但在运行过程中，*MainActivity* 的父类方法 *onResume()* 被调用了，便会在结果中呈现出来。在这点上，FlowDroid 的处理方式恰恰相反，考虑到在未重写系统类方法时的类行为表现基本保持一致性，FlowDroid 并不会在结果中展示父类 Activity 未重载的生命周期方法。

观察图 6.4 中生命周期方法归属的 Activity，我们发现，在应用启动 *NewActivity* 过程中，*MainActivity* 和 *NewActivity* 的生命周期方法是交替出现的，并不是 *MainActivity* 的生命周期方法全部执行完毕后才执行 *NewActivity* 的生命周期方法。而 FlowDroid 给出的结果将 *MainActivity* 和 *NewActivity* 分开处理，因此得到的结果属于后面一种情况。相比 FlowDroid，RunDroid 的运行结果是程序运行时的直接反映，更适合反映应用的状态变化。

### 6.1.3 多线程触发关系效果展示

多线程开发是 Android 开发中经常涉及的开发要求。为此，针对这一场景，我们进行了相关的测试。在本节，我们将点击按钮 button3，获取 `doHandleButton3()` 相关的调用图情况。经过实验，RunDroid 和 FlowDroid 的运行结果分别如图 6.6、图 6.7 所示。两幅图在以下节点上存在不同：

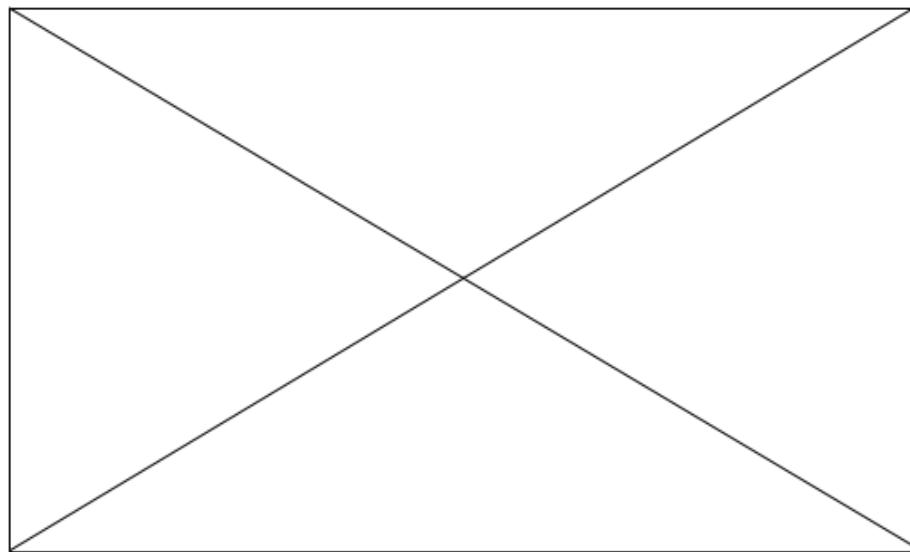


图 6.6: 多线程触发关系-RunDroid 生成的调用图（局部）

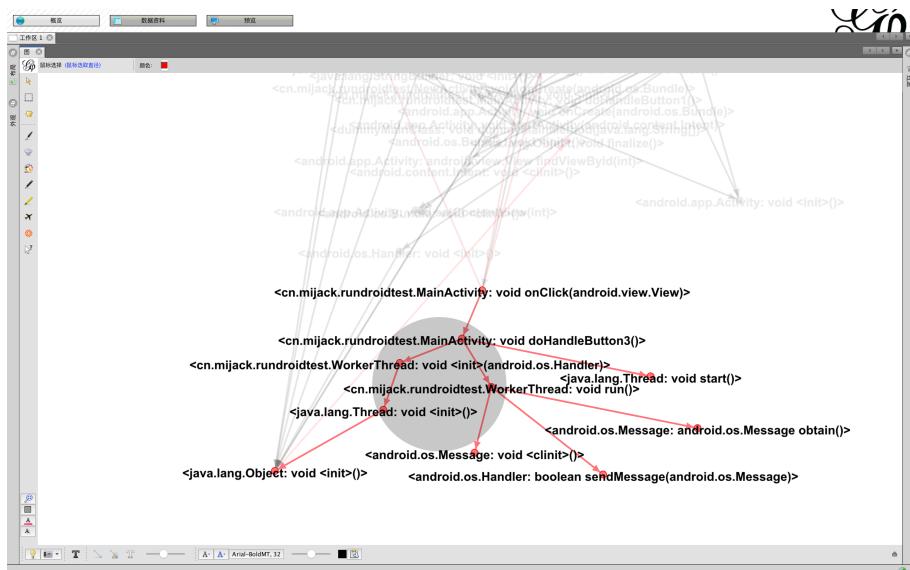


图 6.7: 多线程触发关系-FlowDroid 生成的调用图（局部）

### § Java API 多线程的触发方式

RunDroid 在方法 `Thread.start()` 和方法 `WorkerThread.run()` 之间添加了一条有向边。这条有向边从前者指向后者，表示方法 `Thread.start()` 的执行触发了方法 `WorkerThread.run()` 的执行，边上标识着触发原因（通过 Thread 方式触发，Thread）。FlowDroid 对方法 `doHandleButton3()` 进行推算出图 6.1 第 42 行中的变量 `workerThread` 类型为 `WorkerThread`。同时，FlowDroid 可以推算出方法 `Thread.start()` 的执行会导致方法 `WorkerThread.run()` 的执行。在 FlowDroid 的设计者看来，方法 `Thread.start()` 可以替换成方法 `WorkerThread.run()`。因此，FlowDroid 的结果中，方法 `doHandleButton3()` 和 `WorkerThread.run()` 之间存在调用边。这条有向边并不是像 RunDroid 从 `Thread.start()` 出发。虽然 RunDroid 和 FlowDroid 都可以在函数调用图表现 Java API 多线程的触发方式，但相比之下，RunDroid 的呈现方式更为全面严谨，比 FlowDroid 更符合程序的运行过程。当方法 `Thread.start()` 和 `WorkerThread.run()` 同时在同一个函数中被调用，FlowDroid 在调用图上无法将这两个方法间的调用关系和触发关系区分开。而 RunDroid 针对性地标识了函数间的关系是普通调用关系还是基于线程的触发关系，所以不会在方法关系展现上遇到类似的问题。

### § 基于 Handler 的多线程触发方式

在 `WorkerThread` 的方法 `run()` 中，我们通过 `Handler` 进行了异步 UI 操作，触发了方法 `MainActivity$1.handleMessage(Message)` 的执行。在构建拓展调用图过程中，RunDroid 充分挖掘了这些方法和对应的方法对象的关联关系，进而补全方法 `Handler.sendMessage()` 和 `MainActivity$1.handleMessage(Message)` 之间的触发关系。因此，在 RunDroid 展示的结果中，上述两个方法之间存在一个从前者指向后者的有向边，边上标识着相应的触发原因（通过 Handler 方式触发，Handler）<sup>1</sup>。但是，FlowDroid 分析相关代码时，由于缺少运行时上下文的基本信息，无法分析出 `WorkerThread.run()` 中 `handler` 的具体类型，因此，无法将 `Handler` 相关的两个方法连接起来。

<sup>1</sup> 由于我们定义的 `Handler` 是 `MainActivity` 的匿名内部类，因此编译后的类名为 `MainActivity$1`。

### § 静态方法的使用

另外，在 FlowDroid 给出的结果中，我们发现方法 `doHandleButton3()` 还调用了类 `Message` 的类初始化方法 `<clinit>()`。原因是方法 `doHandleButton3()` 调用类 `Message` 的静态方法 `obtain()`，因此可能需要进行类的初始化。由于在程序运行过程中，方法 `doHandleButton3()` 在执行方法 `Message.obtain()` 前，类 `Message` 已经完成了类的初始化，方法 `Message.<clinit>()` 并没有被调用。因此 RunDroid 的结果中显示上述两个方法不存在调用关系。这从一个侧面方面反映出 FlowDroid 分析的结果和动态运行的过程可能存在一定的偏差。

## 6.2 RunDroid 在错误定位领域的应用

在本节中，我们将采用因果关联模型 [? ?] 作为错误定位技术。相比传统错误定位技术（基于频谱的错误定位）只考虑了程序语句的覆盖情况，基于因果关联模型的错误定位技术还考虑程序在执行过程中的程序依赖（即控制依赖和数据依赖），分析的准确度较高。

### 6.2.1 原理简介

基于因果关联模型的错误定位技术，以程序本身  $P$  和程序在一组测试用例下的覆盖率信息作为输入，通过期望模型（公式 6.1）和线性回归模型（公式 6.2）计算程序  $P$  中的语句  $s$  的相应的错误估计值  $\tau$ 。 $\tau$  的值越大，语句  $s$  是错误的可能性越大。具体过程如算法 6.1 所示。

$$\tau(s) = E[Y = 1|T = 1] - E[Y = 0|T = 0] \quad (6.1)$$

注： $E[Y = 1|T = 1]$ ：实验组执行失败的期望， $E[Y = 0|T = 0]$ ：对照组执行通过的期望。

$$Y = \alpha + \tau T + \beta X \quad (6.2)$$

注：其中  $Y$  表示测试用例是否执行失败（1 为执行失败，0 为执行通过）， $T$  和  $X$  分别为语句  $s$  和  $Pred(s)$  中各语句的覆盖情况， $\beta$  为  $Pred(s)$  中各语句的错误估计值， $\alpha$  为公式中待计算的定值。

对于语句  $s$ , 语句  $s$  的前驱节点集合  $Pred(s)$  指的是语句  $s$  在程序  $P$  中的控制依赖和数据依赖的合集 (第 3 行)。匹配过程中, 每个测试用例按照语句  $s$  前驱节点集合  $Pred(s)$  的覆盖率情况表示成一维向量, 并按照语句  $s$  是否覆盖将测试用例分为实验组和对照组 (实验组 ( $T=1$ ) 和对照组 ( $T=0$ ) 分别与覆盖语句  $s$  的测试用例、未覆盖语句  $s$  的测试用例对应); 如果向量表示结果相同的测试用例同时出现实验组和对照组中, 则保留对应的测试用例到集合  $Mdata(s)$  中, 反之不保留 (第 4 行)。匹配完毕后, 当  $Mdata(s)$  不为空集, 我们将使用公式 6.1 计算语句  $s$  的错误估计值  $\tau(s)$  (第 6 行)。当  $Mdata(s)$  为空集, 我们将通过线性回归公式 6.2 公式计算得到关于  $T$  的斜率  $\tau$  作为语句  $s$  的错误估计值  $\tau(s)$  (第 8 行)。最后, 对于所有的语句, 按照错误估计值逆向排序输出, 算法完毕。

---

#### 算法 6.1: 错误定位的计算方法

---

**输入:**  $P$ , 应用程序

**输入:**  $CoverageInfo$ , 应用程序的覆盖信息

**输出:**  $sorted_{\tau}$ , 程序语句的逆向排序

1 **Function** *FaultLocazilation*( $P, CoverageInfo$ ):

```

2   for  $s \in P$  do
3       计算语句  $s$  在程序  $P$  中前驱节点集合  $Pred(s)$ ;
4       根据  $Pred(s)$  的覆盖率情况筛选待计算的数据  $Mdata(s)$ ;
5       if  $Mdata(s)$  为  $\emptyset$  then
6           使用  $Mdata(s)$  根据公式 6.1 计算  $\tau(s)$  ;
7       else
8           根据公式 6.2 计算  $\tau(s)$  ;
9   对各语句按照  $\tau$  逆向排序得到  $sorted_{\tau}$ ;
10  return  $sorted_{\tau}$ ;

```

---

#### 6.2.2 结果分析

## 第七章 总结与展望

### 7.1 总结

背景：

贡献总结：

系统实现：

实验评估：

**RunDroid** 以帮助用户（开发人员）了解 **Android** 应用程序的执行过程作为基本出发点，让 **Android** 应用在执行时输出相应的执行日志信息，进而利用这些日志信息恢复 **Android** 应用程序执行时的动态调用图。在具体实现上，我们通过对源代码进行日志插桩的方式输出程序在应用层上的执行信息，利用 **Xposed** 可以改变 **Android** 系统行为的特性进行系统方法的拦截处理，从而记录 **Android** 系统内部的执行信息。除了展示方法间的调用关系之外，**RunDroid** 考虑到 **Android** 系统上的多线程调用场景，提供多线程触发关系展示的功能：**RunDroid** 对日志进行初步处理，在 **Neo4j** 图数据库上构建程序调用图，根据具体的多线程调用规则结合 **Soot** 提供的实现类查询服务，生成对应的 **Cypher** 脚本语句，在对应的节点之间创建对应的触发关系，展现了方法间的多线程触发关系。这个方案思路清晰简洁，可以从方法调用、方法间的触发关系、相关对象信息等多个方面较为全面地展现 **Android** 应用的执行过程，具有一定的通用性和拓展性。

## 7.2 展望

本文致力于还原 Android 应用程序的动态函数调用图，反映程序的运行时状态，虽然取得了一定的研究成果，但在实验过程中仍然发以下问题：

(1) RunDroid 在捕获应用用户层方法时，采用的方案是源代码插桩方案。调用图构建的前置条件需要提供 Android 应用的源代码。系统运行对源代码高度依赖，现在阶段的 RunDroid 更适用于企业开发环境，和工业生产环境下应用还存在一定的距离。

(2) 在实验阶段，我们发现，当应用程序长时间运行时，应用程序会产生较多的日志。通常的，移动设备上的存储是有限的。因此，对于一些调用关系较为复杂的应用，RunDroid 的日志方案比较容易遇到日志存储的瓶颈。

(3) RunDroid 中的运行时拦截器是基于 Xposed 框架实现的。Xposed 框架并不是适用于所有的 Android 手机，在一定程度给 RunDroid 的实验环境提出了额外的要求。

整体上，本文提出的 RunDroid 较为准确地还原出 Android 应用程序在运行过程的函数调用图。主要的改进方法：利用字节码修改技术代替源代码修改方案以减少 RunDroid 运行过程中对源代码的依赖；引入基于 JVMTI 的调试环境，借助调试技术实现系统方法执行的拦截，摆脱对 Xposed 环境的依赖；通过静态分析技术确定运行过程的确定性路径，缩减待插桩的用户方法数量，进而减少运行时日志的产出量，同时在构建函数姐需要等。

## 致 谢

## 攻读学位期间发表的学术论文

1. Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu, RunDroid: Recovering Execution Call Graphs for Android Applications. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017 是软件工程领域的国际顶尖学术会议，属于中国计算机协会 (CCF) 评选出的软件工程领域 A 类会议).