2019 届硕士专业学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51164500190

華東師範大學

XXXXXX

XXXX

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：	**移动应用分析**
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 University Code: 10269

(Professional) Student ID: 51164500190

EAST CHINA NORMAL UNIVERSITY

xxxxxxxxxx

Department:	School of Computer Science and Software Engineering
Major:	Computer Technique
Research Direction:	Mobile Software Analysis
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《xxxxxxx》，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

《xxxxx》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，本论文的研究成果归华东师范大学所有。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，并向主管部门和相关机构如国家图书馆、中信所和“知网”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
			主席
		华东师范大学	
		华东师范大学	

摘　要

关键词: Android，函数调用图，日志生成

ABSTRACT

Keywords:

目录

插图

表格

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的生活需求，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。

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

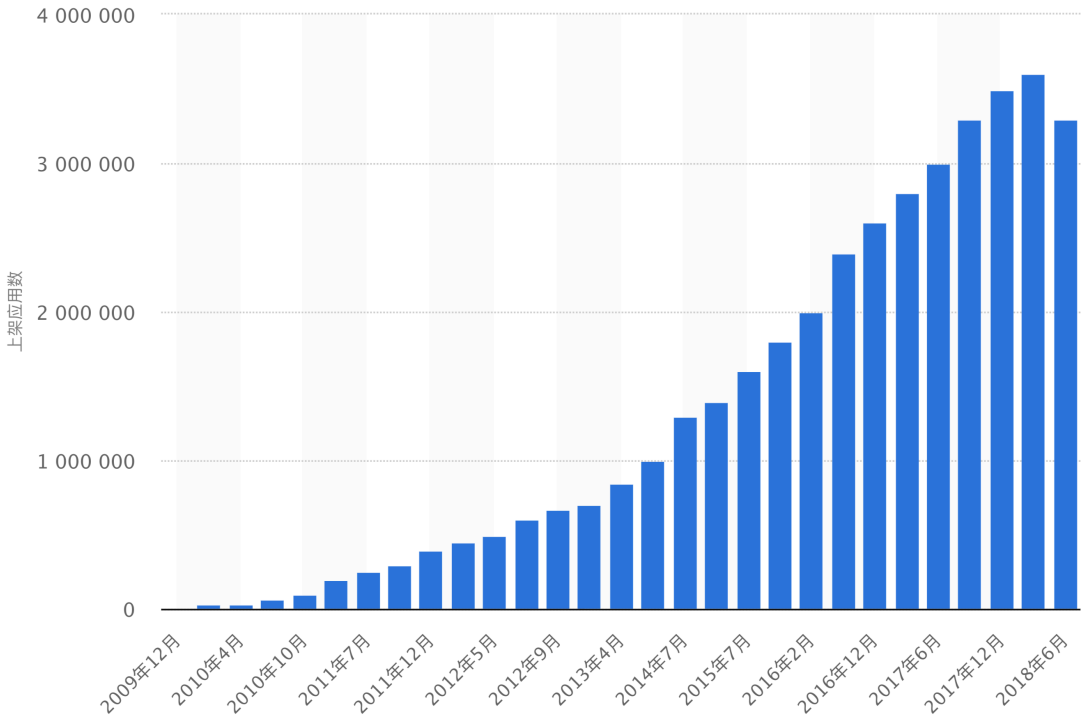

图 1.1: Google Play Store 上架的应用总数的变化趋势

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[1]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

[1]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。

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

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 Davlik 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 之上的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 gragh）。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。法国和意大利的学者 [**?**] 通过对 Java 字节码静态分析器 Julia 进行扩展，使得其支持对 Android 应用程序的静态分析；他们通过改写 Android 库中 Activity、LayoutInflater 等类的代码逻辑，规避了 Android 系统分析常见难点 (如程序的事件机制，基于反射的视图加载等)，实现了包括死代码检查、空指针检查在内的 7 种静态分析技术。

1.2.2 动态分析技术

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 TaintDroid [**?**]、DroidBox [**?**]、TraceDroid [**?**]、DroidScope [**?**] 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生成源 APK 文件中的方法总数超过 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、应用分析等领域。

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。

IOS Detecting privacy leaks in ios applications

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 系统程序源代码进行静态分析，构建出整个 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。相比其他类似工作，由于考虑了 Android 系统特性，PScout 分析的到结果更加完整，但受到可达性分析的精度限制，存在部分错误的映射关系。此外，该团队对 Android 2.1 ∼ 4.0 等四个版本的系统源代码进行了广泛的分析，就权限设计的冗余性、无文档说明 API 的权限要求、权限在 Android 系统升级的演变等多个研究问题进行深入的探讨。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，例如恶意软件用到了和正常软件经常使用到的发送短信的功能，不同的是恶意软件一般在特定的时间点（深夜）运行，以防止引起用户的注意。为了筛选出这种类型的恶意软件，Wei Yang 等人提出了一个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，提取出安全敏感行为及对应的上下文，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。

AppIntent [**?**]

也有一部分研究聚焦移动应用在网络传输上的安全问题。Socket [**?**]

质量保障:

Exception fault localization in Android applications [**?**]

machado2013mzoltar [**?**]

Suting [**?**]

Fan [**? ?**]

现有的性能测试的工作中，Mario Linares-Vasquez[25] 等提出通过对开源社区 App 的分析，详细归纳了 App 性能中由线程问题、GUI 渲染问题、内存问题导致 App 运行异常的原因。

应用分类:

Simapp [**?**]

mobile app tagging [**?**]

1.3 本文的主要工作

本文的主要研究工作包括以下:

1) 调研最近几年 Android 应用分析领域的静动态分析工具，了解各项工具的优劣以及相关的应用案例。

2) 提出并实现 Android 动态函数调用图构建系统，包含传统函数调用图的构建以及在此基础之上的多线程函数调用关系的构建。

3) 对上述系统设计对应的实验方案，评估对应的实验效果。

1.4 本文的组织结构

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：主要介绍了本文的主要研究背景、主要工作内容以及研究意义，最后对本文的各章的内容做了阐述。

第二章：从 Android 的体系结构出发，并由此展开介绍了最近几年 Android 领域相关的静态分析技术和动态分析技术，以及这些技术在各自领域的运用，简单描述本文可能遇到的困难。

第三章：从系统功能、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。

第四章：从应用程序的构建效率、日志本身记录效率以及系统对程序运行的影响等三个方面设计并开展相关的测试实验。

第五章：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。

第六章：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：

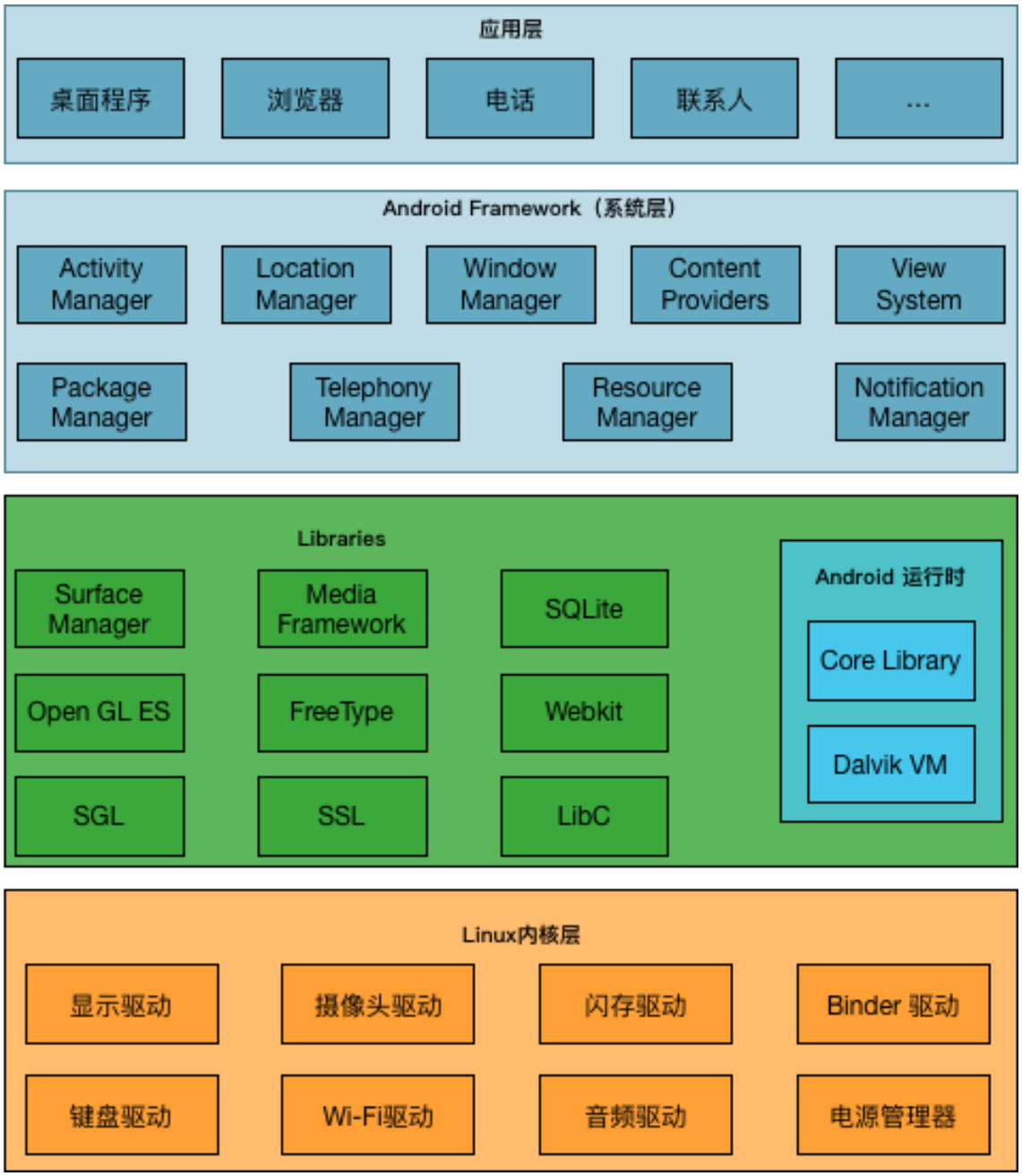

图 2.1: Android 系统框架图

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：

1. 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。

2. 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。

3. 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。

4. 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 系统为 Activity 提供了 onCreate(), onStart(), onResume(), onPaused(), onRestart(), onStoped() 和 onDestroy() 等方法，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。

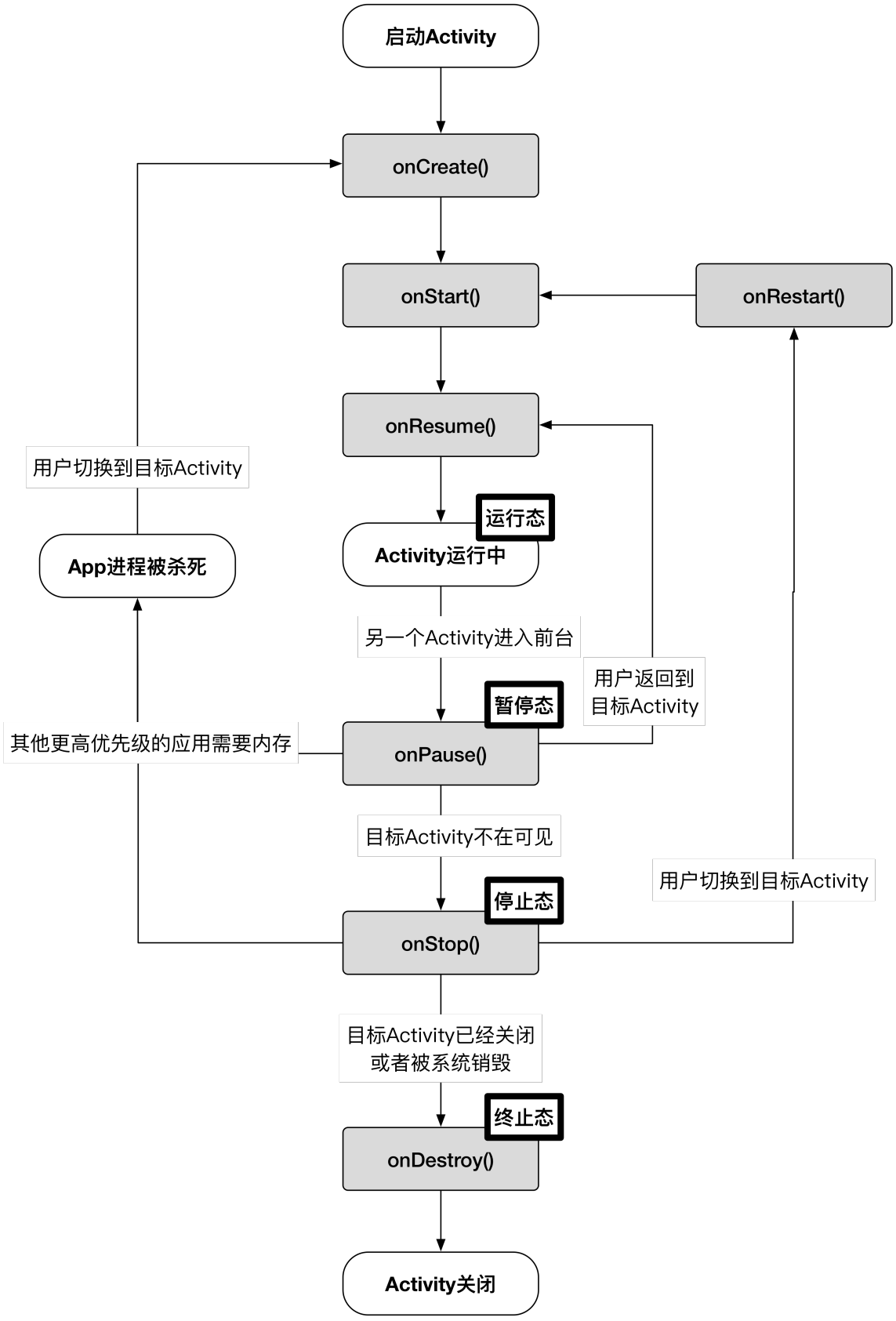

图 2.2: Activity 的生命周期

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，这将会使阻塞部分线程业务逻辑的执行，使得应用变得复杂低效。上述情况对于应用程序都是不可接受的。为了避免多线程操作之间的竞争关系带来的低效率问题，Android 系统在设计事件驱动架构时，采用了单线程的消息队列，即只允许在主线程（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。

当应用程序出现耗时操作（例如加载磁盘上的图片、网络请求等）时，应用程序往往需要在一个新的线程中执行上述逻辑。当应用程序界面中的某些控件需要根据耗时操作的结果（例如渲染得到的图片对象、网络请求得到的 JSON 字段）更新界面状态时，开发人员需要切换到主线程进行界面的更新。

从整体上，开发者可以需要的交互方式分为基于 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 void runOnUiThread (Runnable action) API。runOnUiThread 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 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。当开发人员需要当前业务逻辑转移到其他线程时，通过方法 Message.obtain() 获取一个 Message，将对应的业务逻辑封装成 Runnable 对象传递给 Message 中对应的字段，或者将对应的参数传递给 Message 中的参数字段，最后通过 Handler 对象发送给指定的消息队列。当目标线程的消息队列读取到这条消息时，便会在该线程中执行预定的业务逻辑。

图 2.3为 Handler 的简单示例：用户在工作线程执行一项耗时任务（生成一个字符串），将生成的字符串传递给 Message 对象，并通过 Handler 对象通知主线程进行界面更新。

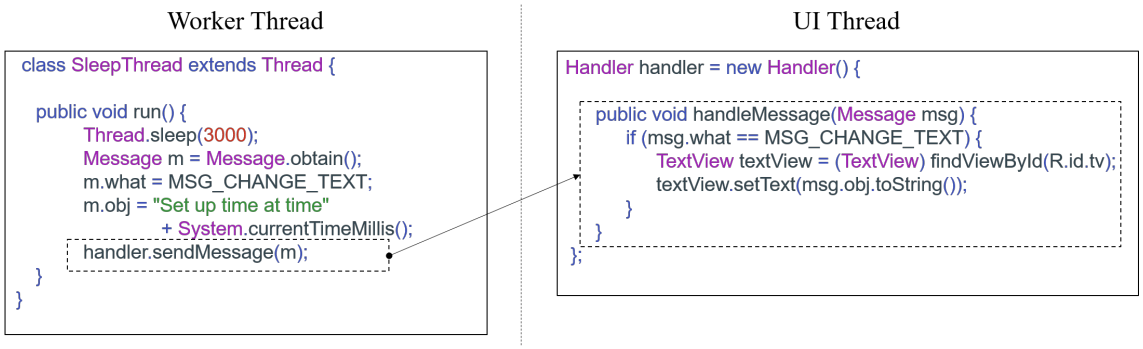

图 2.3: Handler 的使用实例

从 Android SDK 提供的 API 来看，开发人员可以通过 post(Runnable)，postAtTime(Runnable，long)，sendMessage(Message)，postDelayed(Runnable, Object, long)，sendMessageDelayed(Message，long)，sendMessageAtTime(Message，long) 和 sendEmptyMessage(int) 等多种 API 形式实现消息调度。通过查阅和分析 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 android.os.Handler enqueueMessage(MessageQueue, Message, long) 方法。

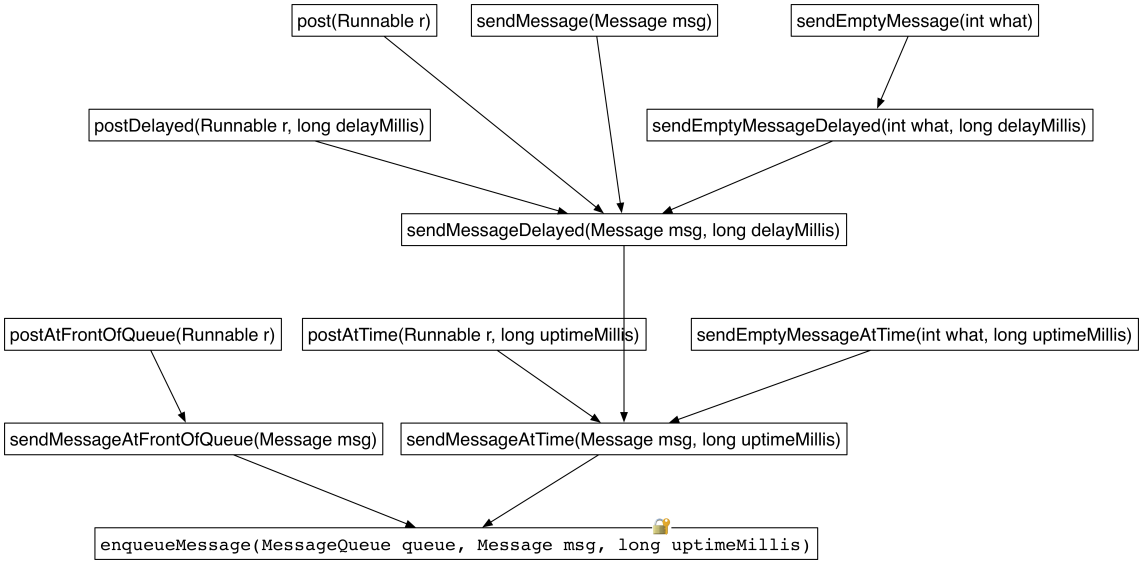

图 2.4: Handler 各 API 之间的调用关系

从底层实现上看，Handler 机制主要由 Handler、Looper、MessageQueue、Message 等若干部分组成。Message 是跨线程交互的主要载体，Android 系统采用对象池的设计模式来管理 Message 对象；无论开发人员以何种形式调用了 Handler 发送消息，传递的参数最后均会封装到 Message 对象中。MessageQueue 则存放着所有待处理的 Message 对象，它是一个双端队列，开发人员可以根据具体业务场景在消息队列的头部、尾部或者适当位置插入消息队列。Looper 则负责以 epoll 的方式从 MessageQueue 中循环读取 Message 对象，分发给对应的 Handler 对象使得业务可以在对应恰当的线程上被处理；一个线程最多只允许只有一个 Looper 对象，他只能绑定一个与之对应的 MessageQueue；其中最为常见的就是位于主线程的 MainLooper，它主要负责 Android 系统的日常调度（例如 Activity 的生命周期、控件的点击事件响应等）。Handler 对象则负责将消息发送到对应的 MessageQueue 中（扮演着消息的生产者角色）以及消费来自 Looper 分发下来的 Message（扮演着消息的消费者角色）；在一个应用中，Handler 可以存在多个对象，一个 Handler 对象也可以同时扮演生产者和消费者两个角色。

从原理上看，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。该消息对象通过 Handler 的对应接口发

送至在目标线程所对应的消息队列 MessageQueue 中，再由 Looper 对象在目标线程运行时从消息队列中取出，分发给对应的 Handler 执行，达到了 Android 跨线程交互的目的。具体地，Handler 的工作原理如 图 2.5所示：

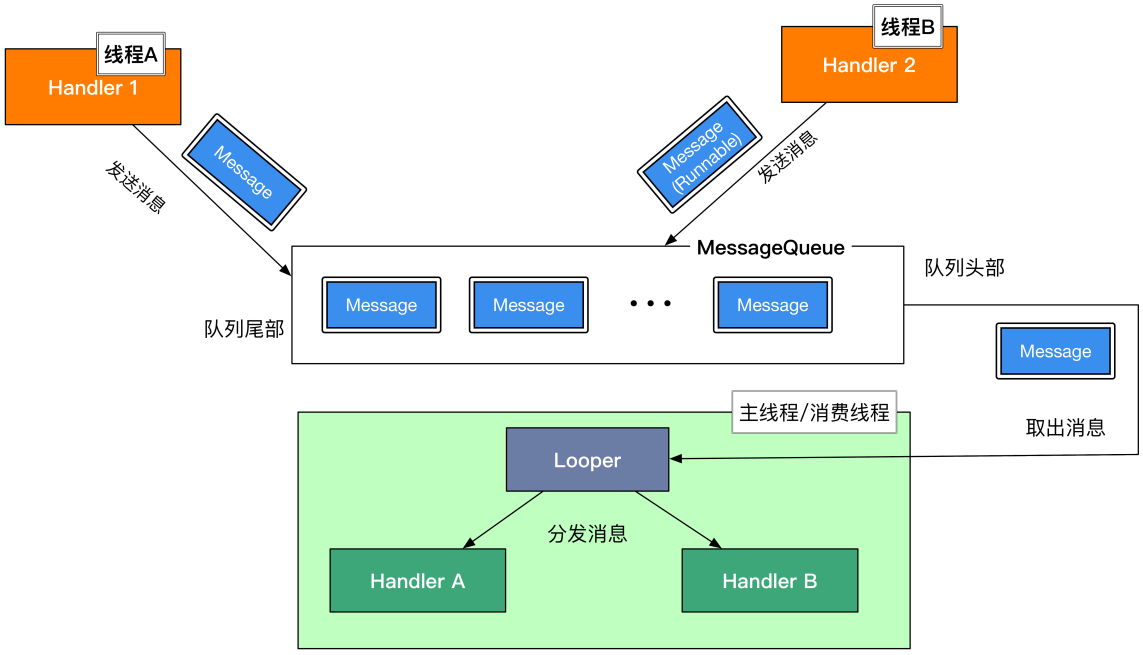

图 2.5: Handler 的工作原理

综上所述，基于 Handler 消息调度的多线程交互方式，不仅可以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 设计还帮助开发人员降低应用的设计复杂程度，提升了系统架构的可拓展性。因此，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 本章小结

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 机器生命周期。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，较为详细地分析了 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　基本术语

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。

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：

定义 1. *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。

定义 2. *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：

- 若对象 o_p 是这个方法 m 的参数，记为 $o_p \overset{parameter}{\longrightarrow} m$，或者 $rel(m, o_p) = parameter$；
- 若对象 o_r 是这个方法 m 的返回值，记为 $o_r \overset{return}{\longrightarrow} m$，或者 $rel(m, o_r) = return$；
- 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 $o_i \overset{instance}{\longrightarrow} m$，或者 $rel(m, o_i) = instance$。

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 $m_1 \to m_2$，称为方法 m_1 调用方法 m_2。

$$m_0 \to m_1 \to \dots m_n \to m \tag{3.1}$$

在此基础上，对于方法 m，若存在方法 m_i（$i = 0, \dots, n, n \geqslant 0$），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 $m_1 \to m_2$，则 $m_1 \xrightarrow{*} m_2$ 也成立。

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 $m_a \hookrightarrow m_b$，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 之间存在着一定的因果关系，包括但不限于生命周期事件，UI 交互事件或多线程通信等。

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 $m_1 \to m_2$），则有向边 $e = (m_1, m_2)$ 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**。$(m_{a_{(1)}} \to m_{b_{(1)}})$ 对应的是第一次函数 A 调用函数 B，$(m_{a_{(2)}} \to m_{b_{(2)}})$ 对应的是第二次函数 A 调用函数 B，

$$
\begin{aligned}
CG =& (V, E), \\
V =& \{m_{a_{(1)}}, m_{b_{(1)}}, m_{a_{(2)}}, m_{b_{(2)}}\}, \\
E =& \{(m_{a_{(1)}} \rightarrow m_{b_{(1)}}), (m_{a_{(2)}} \rightarrow m_{b_{(2)}})\}
\end{aligned}
\tag{3.2}
$$

定义 6. 拓展函数调用图（*Extended Dynamic CallGraph*，*EDCG*）

在函数调用图（DCG）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。具体定义如 公式 3.3所示：

$$
\begin{aligned}
EDCG =& (V_{EDCG}, E_{EDCG}), \\
DCG =& (V_{DCG}, E_{DCG}), \\
V_{EDCG} =& V_{method} \bigcup V_{object}, \\
V_{method} =& V_{DCG}, \\
G_{EDCG} =& G_{method} \bigcup G_{object}, \\
G_{method} =& E_{DCG} \bigcup \{(m_1, m_2) \mid m_1 \hookrightarrow m_2\}
\end{aligned}
\tag{3.3}
$$

第四章　设计与实现

4.1　系统功能介绍

本文以帮助研究人员和开发人员了解 Android 应用程序的执行过程作为基本出发点，通过设计与实现 Android 动态函数调用图构建系统 RunDroid，生成 Android 应用程序运行时对应的动态函数调用图，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展现 Android 应用的执行过程，为应用程序分析提供更为多样、准确的信息。另外，系统具备一定的可拓展性，可以方便相关人员根据自身的业务需求对系统进行扩展，完成相应的需求。

4.2　基本思想

以 图 4.1为例， 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-左下则是期望输出——函数调用图。在本质上，函数调用图属于树。程序执行过程可以看做树的深度优先遍历过程。函数调用图还原的关键点，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。通常的，树的遍历分为中序遍历、前序遍历以及后序遍历三种。不幸的，两颗结构完全不同的树对应的遍历序列可能是一样的，这也就意味着上述三种遍历方法均不能直接还原出函数调用图。

若在程序执行前后均记录日志（即一个方法的执行会输出两条日志：方法开始日志、方法结束日志）时，我们得到的遍历序列是唯一的，可以直接用于函数调用图的还原。

为此，我们采用的基本思路如下：通过对源程序（ 图 4.1-左上）进行预处理，

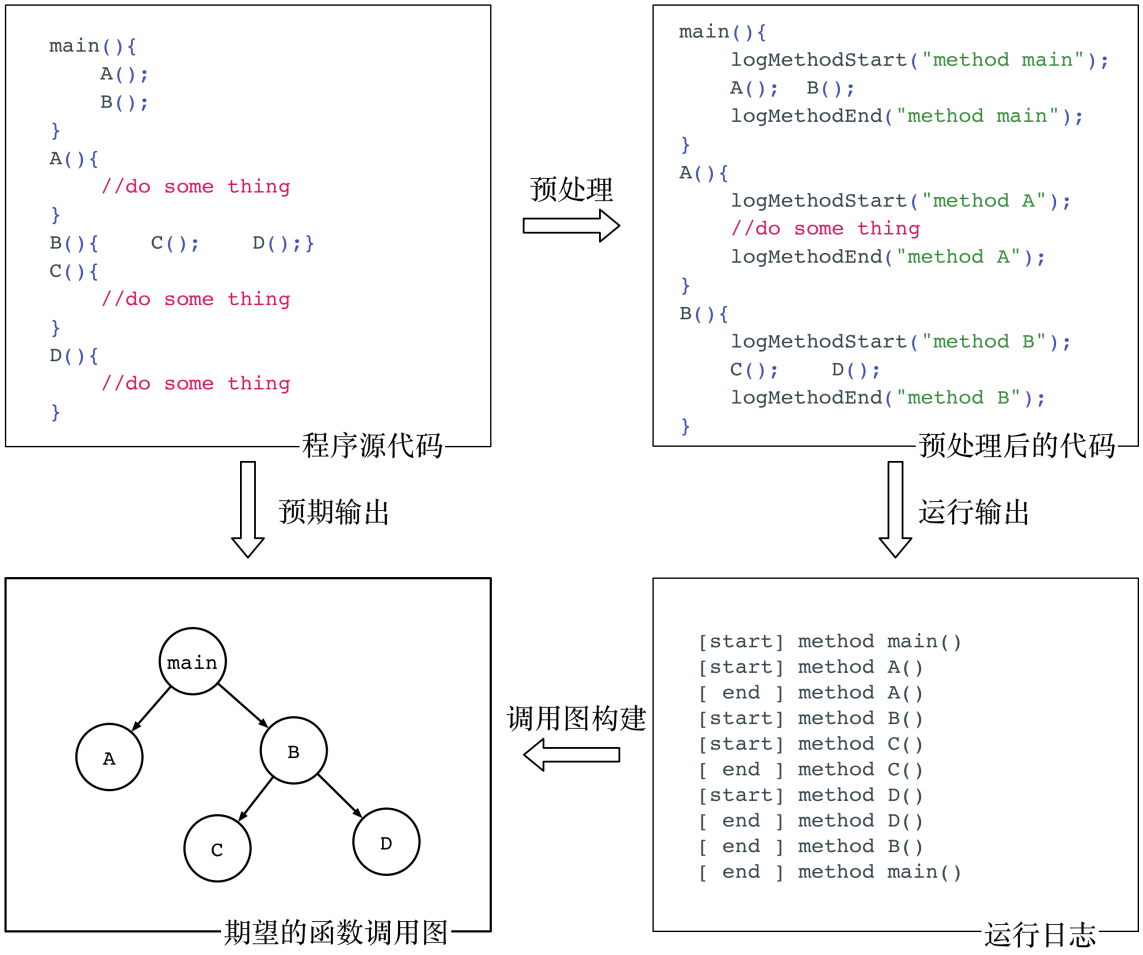

图 4.1: RunDroid 的基本思路

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用图（ 图 4.1-左下）。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。

4.3 技术路线

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。对应技术路线如下：预处理器通过源代码插桩技术实现用户方法层面的信息记录，而运行时拦截器则负责拦截系统方法的执行。在应用

运行时，日志记录器会记录用户方法和系统方法对应的执行信息，以日志的形式记录下来。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。对应的工作流程如 图 4.2所示。

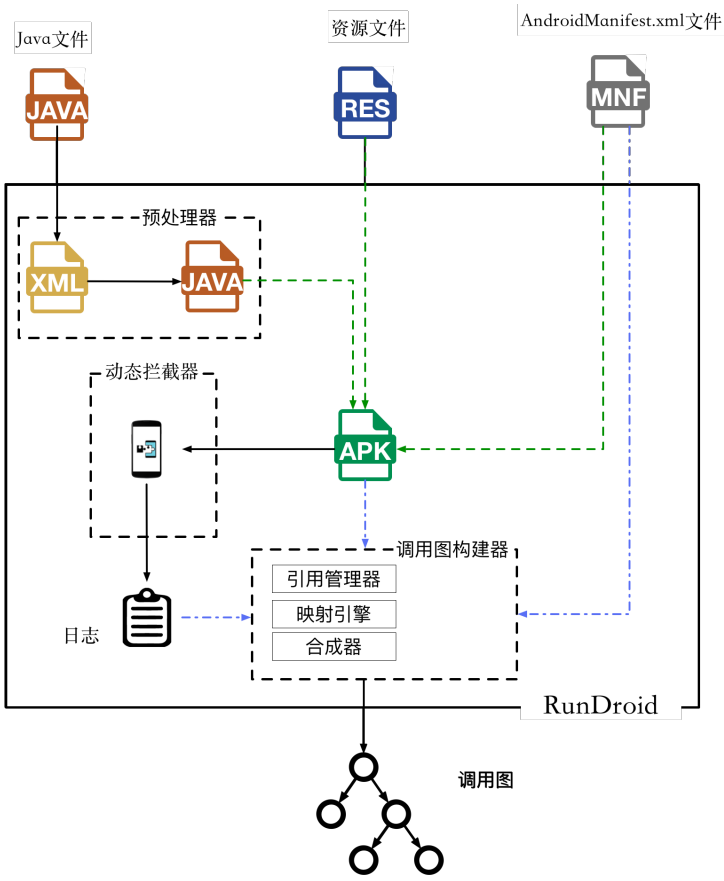

图 4.2: RunDroid 的工作流程

4.4 相关技术介绍

4.4.1 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。srcML 的语法解析器采用的是基于 LL(*) 算法实现的语法解析器 ANTLR 2.7.7，解析得到的 XML 内容保留了源代码中完整的语法树信息。同时，srcML 还提供了一个强大工具集，支持对生成内容的查询、分析以及修改，可以用于架构设计、语言研究、软

件重构等场景，应用于软件工程、编程语言、并行和分布式处理等多个领域。在RunDroid，srcML 承担的主要职责为源代码的语法解析。

4.4.2 Xposed 框架

Xposed [**?**] 是由 rovo89 主导开发的第三方框架。基于 Xposed 开发的第三方插件，可以在不修改系统和应用程序源代码的情况下，改变他们的运行行为。Xposed 框架可以运行在不同版本的 Android 系统上，开发过程十分便利，而且易于撤销。Xposed 的实现原理具体如下：由于 Android 系统的所有的应用程序进程都是由 Zygote 进程孵化而来，Xposed 通过替换程序*/system/bin/app_process*，使得系统在启动过程中加载 Xposed 的相关文件，将所有的目标方法指向 Native 方法 xposedCallHandler，维护目标方法和对应的钩子方法（Hook Function）的映射关系，从而实现对 Zygote 进程及 Dalvik 虚拟机的劫持；当程序执行到目标方法时，xposedCallHandler 会完成目标方法的原有代码和对应钩子方法的调度，达到对目标方法劫持的目的。使用 Xposed，可以帮助我们实现类似面向切面编程（Aspect-Oriented Programming, AOP）的功能，完成系统层面的方法执行情况的记录。

4.4.3 Neo4j

Neo4j [**?**] 是基于 Java 语言开发的图数据库。与传统的基于关系模型的存储结构不同，Neo4j 的存储模型是基于图论开发的，遵循属性图数据模型。Neo4j 的数据主要分为节点（Node）和关系（Relationship）两大类，分别对应图论中的节点与边。另外，Neo4j 还可以在关系和节点上添加 key-value 形式的属性，为节点指定一个或者多个标签，为关系指定类型等等。Neo4j 以 Lucence 作为索引支撑，支持完整的 ACID（原子性，一致性，隔离性和持久性）事务规则，提供了基于 Cypher 脚本、Native Java API 和 REST Http API 等多种方式帮助开发人员进行数据开发工作。同时，Neo4j 还提供了友好的浏览器界面，具有十分友好的交互体验。由于基于属性图数据模型，Neo4j 通常适用于和图关系有着密切关系的应用场景：例如社交网络分析，公共交通网络研究以及地图网络拓扑等场景。在 RunDroid，Neo4j 主

要承担着拓展函数调用图的数据存储和查询的主要职责。

4.5 模块实现介绍

4.5.1 预处理器

4.5.2 运行时拦截器

4.5.3 日志记录器

4.5.4 调用图构建器

4.6 扩展函数调用图的构建过程

拓展函数调用图的构建分为两个阶段：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；在函数调用图的基础上，利用方法执行与方法对象的关联关系创建拓展函数调用图。具体过程如下：

4.6.1 函数调用图的构建过程

4.6.2 如何构建 Activity 的生命周期

4.6.3 如何构建多线程触发关系

4.7 本章小结

Algorithm 1 函数调用图的构建过程

输入: $logs$，应用程序在运行过程中的日志输出

输出: cg，函数调用图

$cg \leftarrow$ new CallGraph()
for thread $\in logs.threads$ **do**
  $stack \leftarrow$ new Stack()
  for $log \in logs.get(thread)$ **do**
    top = $stack$.peek()
    if isMethodStartLog(log) **then**
      $m \leftarrow$ generateMethodInfo(log)
      cg.addMethodNode(m)
      ▷ 在调用图中提交方法节点
      if $top \neq null$ **then**
        cg.addInvokeRel($(top \rightarrow m)$)
        cg.addMethodObjectRel($(m_p \xrightarrow{parameter} m)$,$(m_i \xrightarrow{instance} m)$)
        ▷ 在调用图中提交方法对象节点（此处只涉及参数关系和实例关系）
        $stack$.push(m)
      end if
    else
      cg.addMethodObject($(m_r \xrightarrow{return} m)$)
      ▷ 在调用图中提交方法对象节点（此处只涉及返回值关系）
      $stack$.pop()
    end if
  end for
end for
return cg

第五章　系统测试

5.1　实验简介

5.2　构建效率对比实验

5.3　日志效率对比实验

5.4　运行效率对比实验

第六章　应用结果展示

第五章 5.1

6.1　函数调用图的构建结果展示

6.2　Activity 的生命周期效果展示

6.3　多线程触发关系效果展示

6.4　应用程序的状态效果展示

第七章　总结与展望

致　　谢

二〇一八年十月

攻读硕士学位期间发表论文

■ 已公开发表论文

1. Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu,RunDroid: Recovering Execution Call Graphs for Android Applications. In Proc. ESEC/FSE, 2017