2015 届研究生硕士学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：5200000

華東師範大學

East China Normal University

硕士学位论文

DOCTORAL DISSERTATION

题目

****

院　　系：软件学院

专　　业：

研 究 方 向：

指 导 教 师：姓名　教授

学位申请人：姓名

2015 年 **5** 月 **30** 日

Dissertation for doctoral degree in 2015 University Code: 10269
Student ID: 5200000

East China Normal University

Title

****

Department:	**Software Engineering Institute**
Major:	**Software Engineering**
Research direction:	**Formal Methods**
Supervisor:	**Prof. Name**
Candidate:	**Name**

May, 2015

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《***》，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

《***》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。

*** 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
***	教授	大学	主席
***	教授	大学	
***	教授	大学	
***	教授	大学	
***	教授	大学	

摘　　要

关键词: X，Y

ABSTRACT

Key Words: *X, Y*

目录

插图

表格

第一章　绪论

1.1　研究背景

1.2　研究意义

在 Android 应用的开发过程中，经常使用的面向对象编程、回调函数以及多线程交互等开发技术，会干扰 Android 应用静态分析工具的分析过程，导致函数调用关系的错误或者缺失。因此，仅仅依赖静态分析工具，无法准确的了解应用程序执行的具体细节。但是，传统动态分析工具对环境构建、使用成本等方面需要投入的成本大，导致开发人员难以快速上手。

为此，本文提出的解决方案，以较为便捷的方式对应用程序进行预处理，并结合具体的运行环境，从而构建 Android 应用程序的动态函数调用图，为研究人员提供更多样更准确的分析信息，辅助其他工具完成对 Android 应用程序的分析工作。这也是本文的研究意义。

1.3　Android 分析技术介绍

伴随着移动互联网的兴起，国内外的研究人员开始关注移动应用分析，通过对移动应用运用软件分析技术，了解应用程序在运行过程中的行为表现。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。

1.3.1　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 等。但是，静态分析工具在分析过程中虽然可以对应用程序进行较为全面的分析，覆盖应用程序的所有代码，但由于缺少和程序执行过程相关的部分必要信息（应用程序的执行序列、和设备所处环境相关的传感器（如 GPS、温度等）信息等），可能导致部分情况下分析结果的不精确。为了解决这一问题，研究人员提出了动态分析

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。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 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。由此可见，静态分析工具在分析过程中虽然可以对应用程序进行较为全面的分析，覆盖应用程序的所有代码，但由于缺少和程序执行过程相关的部分必要信息（应用程序的执行序列、和设备所处环境相关的传感器（如 GPS、温度等）信息等），可能导致部分情况下分析结果的不精确。

1.3.2 动态分析技术

为了解决这一问题，研究人员提出了动态分析技术。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。这一项技术通常用于恶意软件的检测和排查。这方面的工作代表包括 TaintDroid 、DroidBox、TraceDroid、DroidScope 等。Enck 等人提出的 TaintDroid 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。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。另外，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技术实现了 AOP，进而根据生成的日志文件得到分析结果。

另外，还可以介绍 simplepref 和 uber 的 nanoscope。

Dynamic Analysis Platforms In this section, we further explore existing dynamic analysis platforms and compare their implementations against TraceDroid.

1.3.3 基于其他技术的分析方法

1.4 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。

1.5 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 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：

异于常规程序的开发方式：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

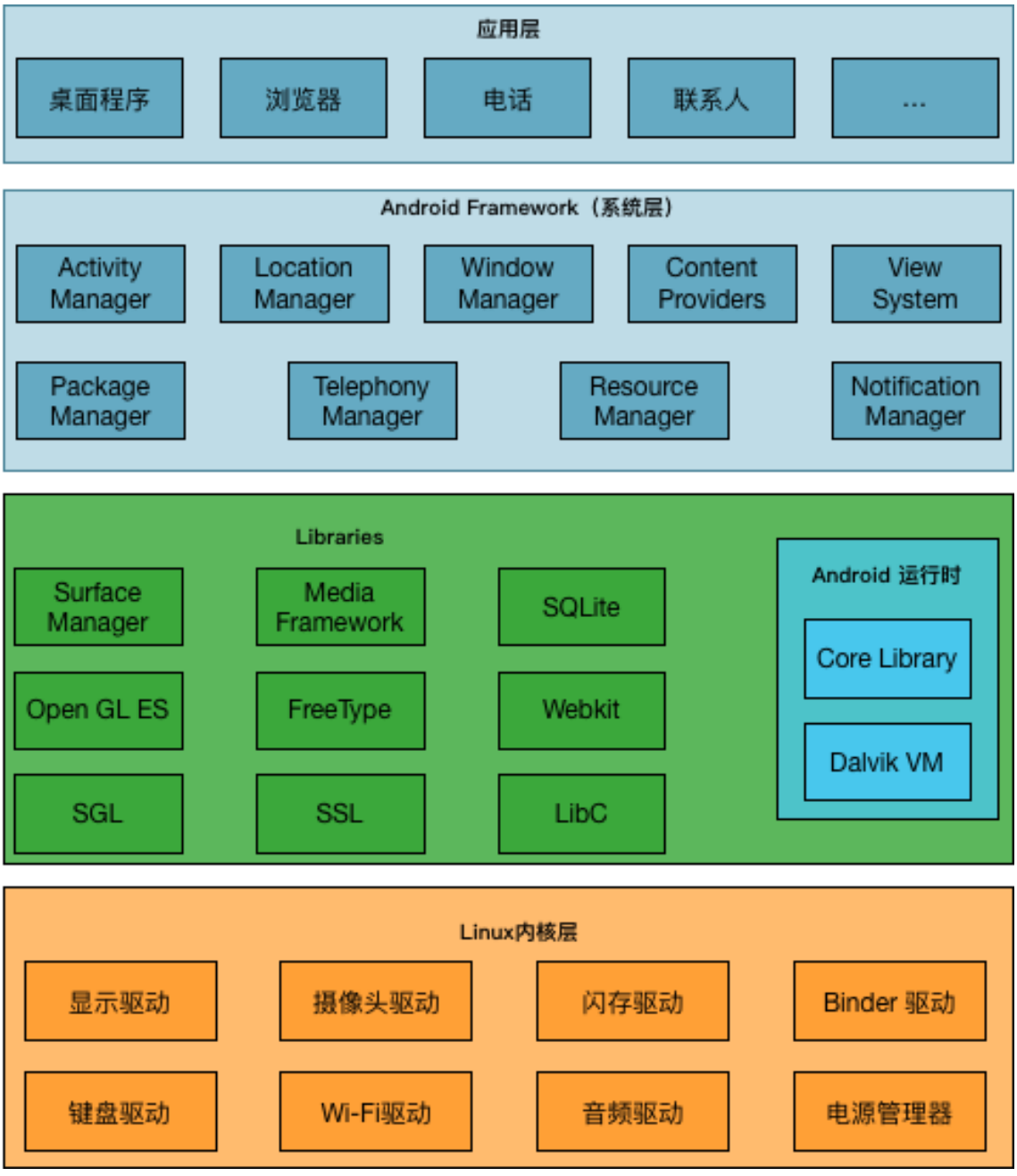

图 2.1: Android 系统框架图

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 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；

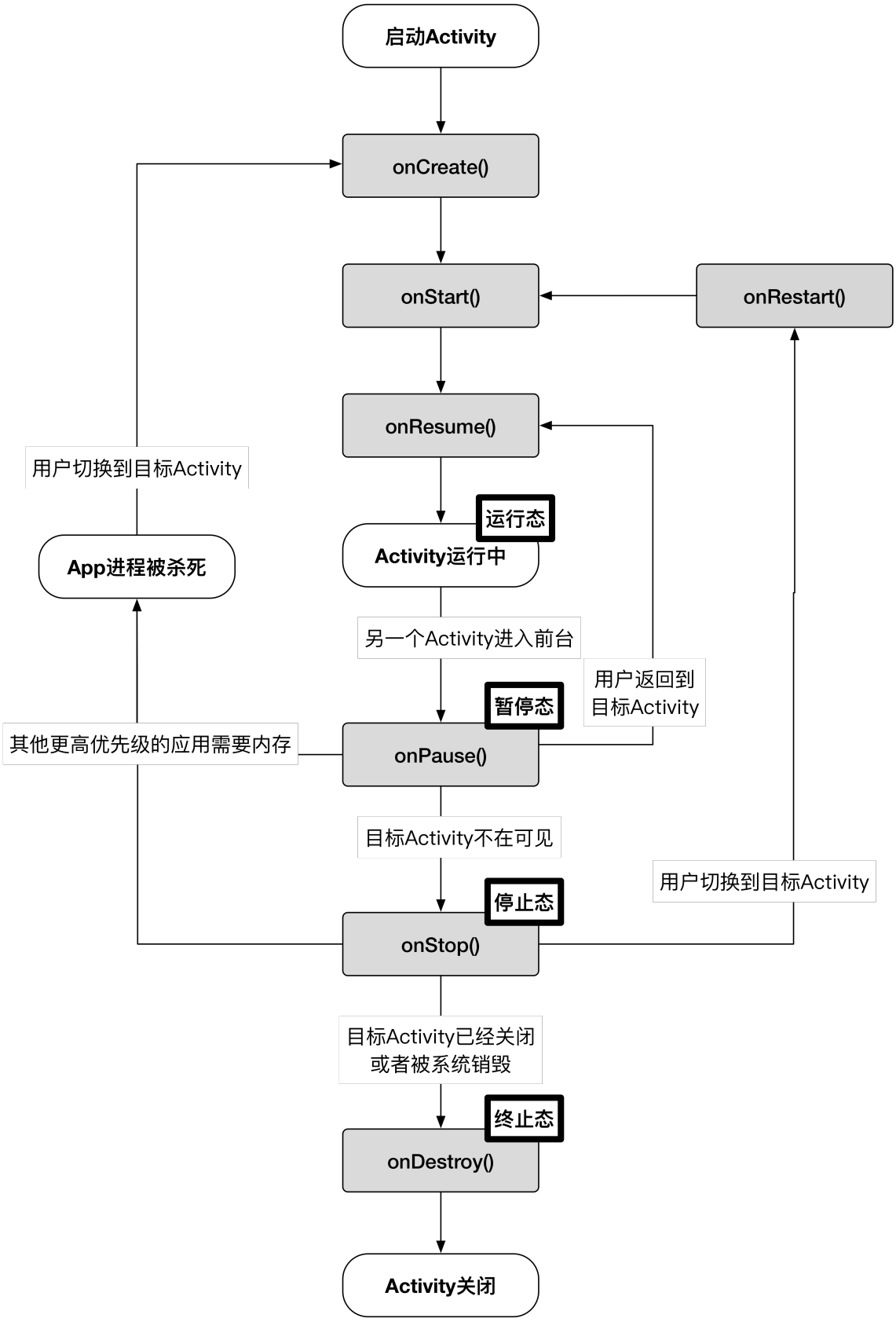

图 2.2: Activity 的生命周期

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，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

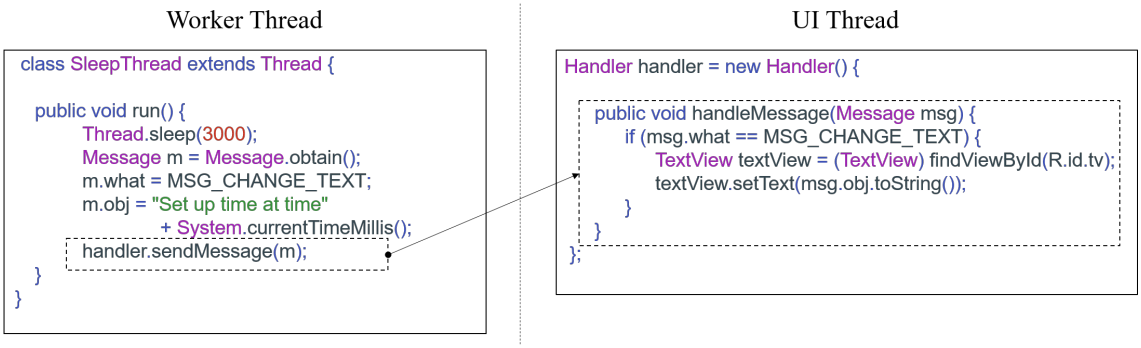

图 2.3: Handler 的使用实例

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 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。

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 4 所示。

从图 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。

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

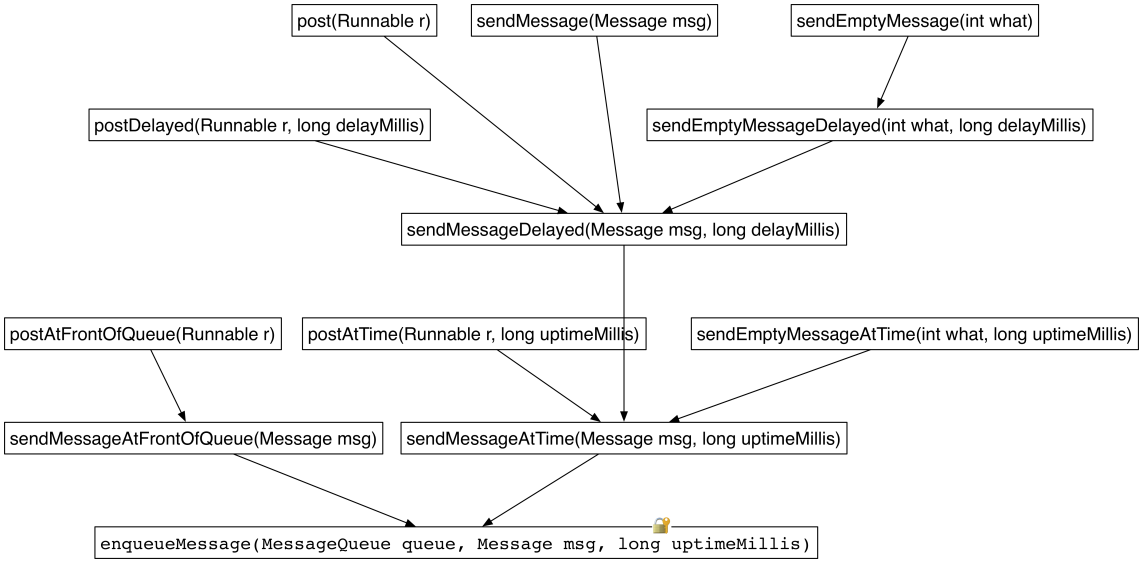

图 2.4: Handler 各 API 之间的调用关系

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 5 所示：

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，

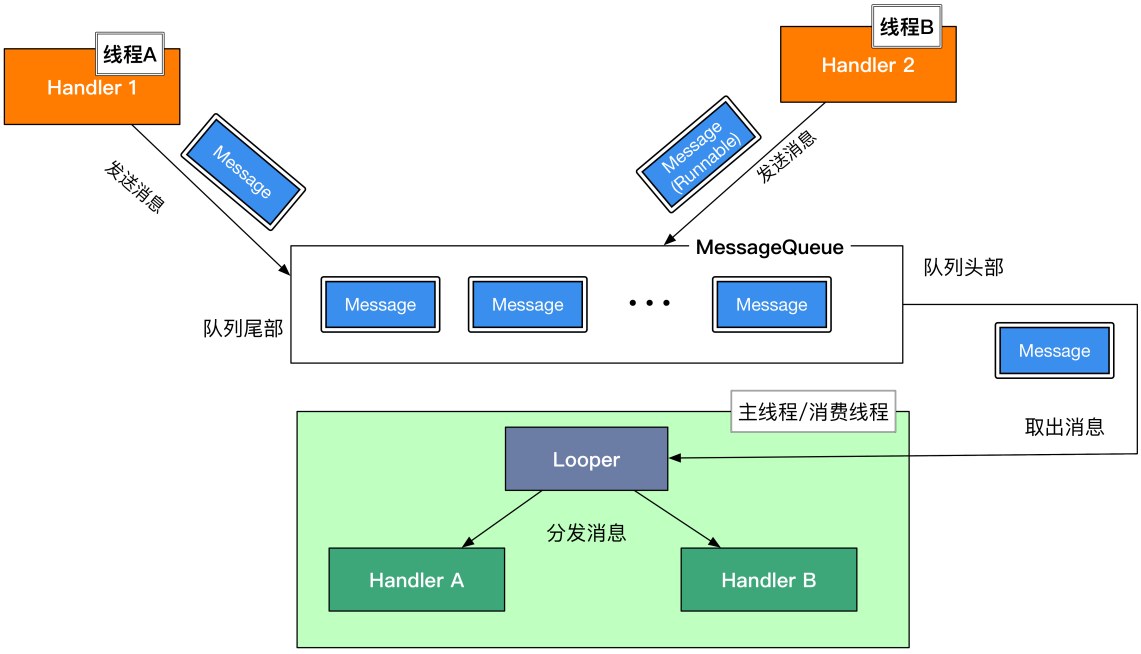

图 2.5: Handler 的工作原理

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　RunDroid 系统的设计与实现

3.1　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。

3.2　概念定义

定义 3.2.1. 调用关系（*Invoke*）：对于程序 P 的两个方法 m_1 和 m_2，如果在方法 m_1 中调用了方法 m_2，则记作 $m_1 \to m_2$，称为方法 m_1 调用方法 m_2。

$$m_0 \to m_1 \to \ldots m_{n-1} \to m_n \tag{3.1}$$

在此基础上，对于方法 m_0 和方法 m_n，若存在方法 $m_i (i = 1, \ldots, n-1\ n > 1)$，使得 3.1成立，，则记作 $m_0 \xrightarrow{*} m_n$，称为方法 m_0 扩展调用方法 m_n。

补充的，对于方法 m_1 和方法 m_2，若 $m_1 \to m_2$，也可以记为 $m_1 \xrightarrow{*} m_2$。

定义 3.2.2. 函数调用图（*CallGraph*，*CG*）：函数调用图 $CG = (V, E)$ 是一个有向图（*DAG*），图中的点 $v \in V$ 表示一个**方法实体** 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。

定义 3.2.3. 动态函数调用图（*Dynamic CallGraph*，*DCG*）：动态函数调用图是在调用图的基础上演化而来，是对程序运行时行为的描述。同样的，动态函数调用图也是一个有向图（$DCG = (V, E)$）。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：函数调用图和动态函数调用图是分别从静态分析角度和动态分析角度对程序运行过程的描述。前者分析的整体是方法实体，是静态分析的结果，每次产生的结果是一样的；后台则为方法执行，是对动态行为的描述，每次产生的结果和具体的执行过程相关，不一定是一样的。

例如，方法 A 调用了方法 B，函数调用图 CG 如 3.2所示。但若，在实际执行过程中，方法 A 被调用了两次，这对应的动态函数调用图 DCG 如 3.3所示。可以看出，DCG 中，$method_a$ 和 $method_b$ 各有两个，分别对应的两次**函数执行**。

$$CG = (method_a, method_b, (method_a \rightarrow method_b)) \tag{3.2}$$

$$\begin{aligned} DCG =& (V, G), \\ V =& \{method_{a_{(1)}}, method_{b_{(1)}}, method_{a_{(2)}}, method_{b_{(2)}}\}, \\ G =& \{(method_{a_{(1)}} \rightarrow method_{b_{(1)}}), (method_{a_{(2)}} \rightarrow method_{b_{(2)}})\} \end{aligned} \tag{3.3}$$

定义 3.2.4. *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。具体的，在一次具体的方法执行过程中，如果对象 p 是这个方法 m 的参数，记为 $p \overset{parameter}{\longrightarrow} m$；如果对象 r 是这个方法 m 的返回值，记为 $r \overset{return}{\longrightarrow} m$；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 i，记为 $i \overset{instance}{\longrightarrow} m$；*

定义 3.2.5. *触发关系（Trigger）：如果对于动态函数调用图 DCG 中两个方法（不妨记为 m_a 和 m_b，$m_a \in DCG$，$m_b \in DCG$），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 触发了 m_b：*

1. *方法 m_a 的执行时间总是在方法 m_b 的执行时间之前；*
2. *$m_a \overset{*}{\rightarrow} m_b$ 不成立；*
3. *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 3.2.6. *拓展动态函数调用图（Extended Dynamic CallGraph，EDCG）：在动态函数调用图（DCG）的基础上，添加了方法对象和函数间的触发关系。拓展动态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*

```
void A(){   // do something A   }
void C(){   // do something C   }
void D( ) {   // do something D  }
void B( ) {
   C();   D();
}
int main( ) {
   A();   B();
}
```

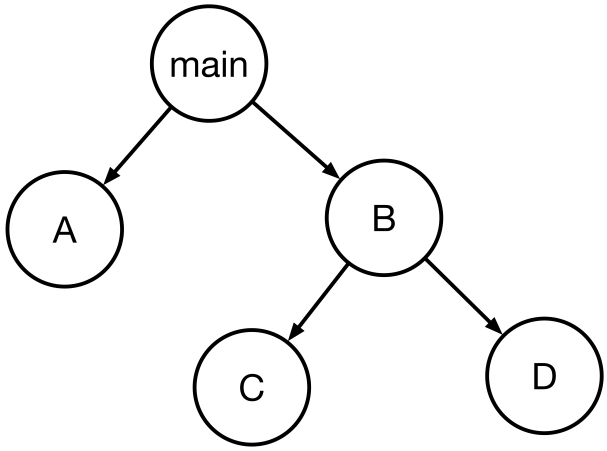

图 3.1: 面向过程编程语言的函数调用的示例

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 *3.4*所示：

$$\begin{aligned} EDCG =& (V_{EDCG}, E_{EDCG}), \\ DCG =& (V_{DCG}, E_{DCG}), \\ V_{EDCG} =& V_{method} \bigcup V_{object}, \\ V_{method} =& V_{DCG}, \\ G_{EDCG} =& G_{method} \bigcup G_{object}, \\ G_{method} =& E_{DCG} \bigcup \{(m_1, m_2) \mid m_1 \hookrightarrow m_2\} \end{aligned} \tag{3.4}$$

3.3　设计思路介绍

以下面一段代码里为例，我们将简要介绍一下 RunDroid 还原 Android 应用程序动态函数调用图的基本思路： 3.1-左为一段面向过程编程的示例代码，当 main 函数执行时，会一次调用 A、B 两个函数，而 B 函数有调用了 C、D 两个函数，对应动态函数调用图如 3.1-右所示。

从函数调用图的构建过程可以看出，程序的执行过程就是对函数调用图自上而下的深度优先遍历过程。由此可见，若要还原出图 6-右中的函数调用图，本文采用的基本思路是以日志方式输出对右图中的函数调用图的深度优先遍历序列，并基于得到的遍历序列还原出函数调用图。由于 Android 是由面向对象编程语言 Java 开发的系统，系统还需要考虑面向对象编程的特性——多态性（即同一个行为在不同的对象下的表现可以不同）。为此，RunDroid 还会在函数调用图将函数执行和对应的对象进行关联，更好地体现面向对象编程的函数调用关系。基于上述的函数调用关系的信息，RunDroid 根据函数调用之间的关系进一步挖掘，进一步挖掘 Android 系统中的特性（例如组件 Activity 的生命周期、多线程的交互

方式）。

3.4 技术路线（偏工程性）

本技术路线拟利用语法分析工具，对 Android 应用程序进行了应用源代码层面的执行日志插桩工作，利用非侵入式系统行为修改插件获取系统层面的函数执行信息。结合以上日志信息，方案对日志进行初步处理，在图数据库上构建原始的 Android 应用程序的动态函数调用图。通过阅读分析 Android 系统中多线程相关的源代码，制定具体的多线程分析插件，进而在函数调用图中标识出多线程相关的方法间触发关系，全面地展现 Android 应用的执行过程。

3.5 相关技术介绍

3.5.1 Soot

3.5.2 srcML

srcML[1] 是基于 1：an infrastructure for the exploration, analysis, and manipulation of source code. 2：an XML format for source code. 3：a lightweight, highly scalable, robust, multi-language parsing tool to convert source code into srcML. 4：a free software application licensed under GPL.

3.5.3 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/system/bin/app_process 程序，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。

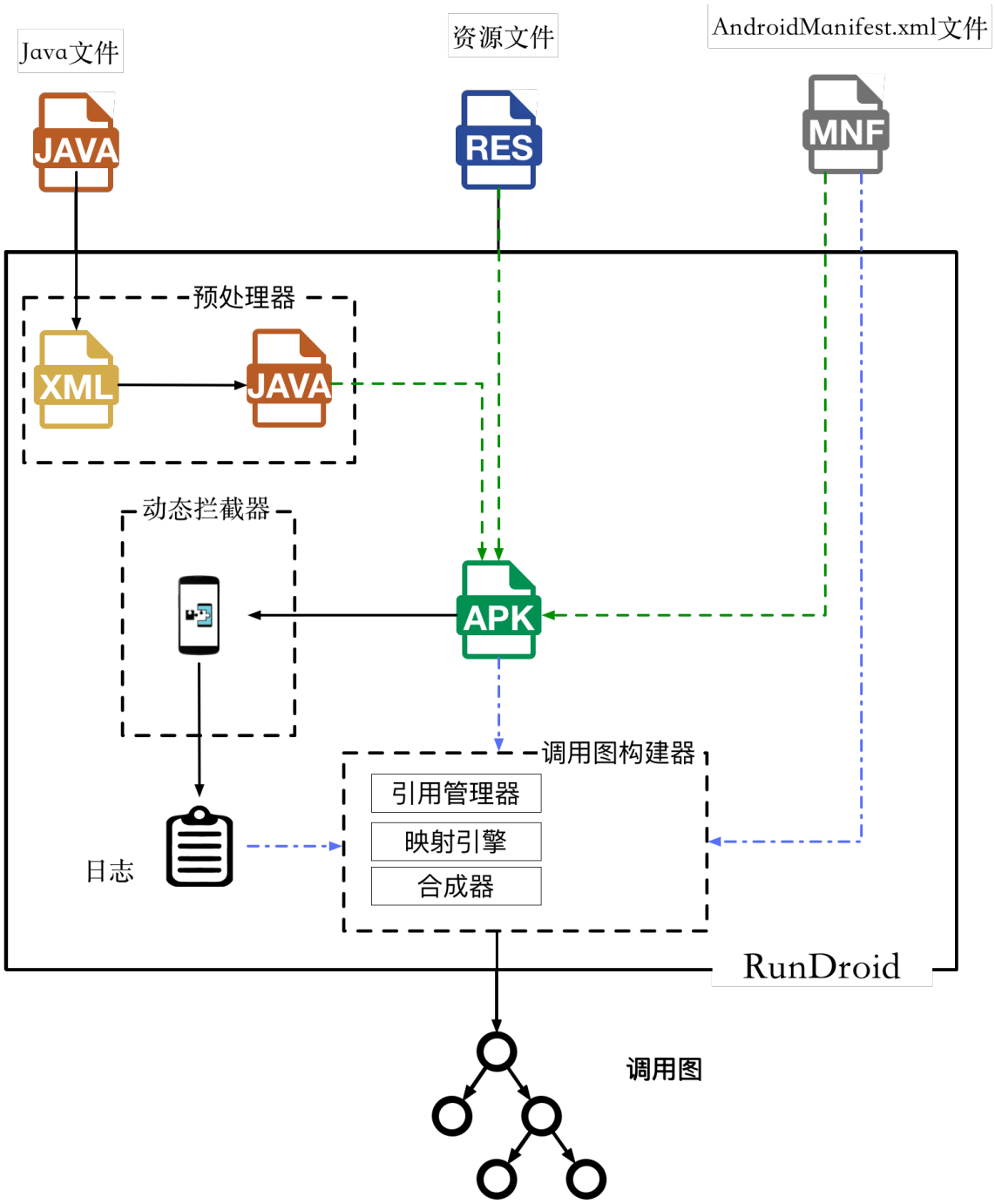

图 3.2: RunDroid 基本架构图

3.5.4 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，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 主要承担着函数调用图的数据存储和查询的主要职责。

3.6 模块实现介绍

3.6.1 源代码预处理组件（Preprocessor）

3.6.2 运行时日志记录器（Runtime Logger）

3.6.3 调用图构建器（Call-graph Builder）

3.7 构建过程介绍

3.7.1 如何构建函数调用图

3.7.2 如何构建 Activity 的生命周期

3.7.3 如何构建多线程触发关系

3.8 本章小结

算法 3.1: Call graph construction

```
Input: logs, the log entries for an app execution
Output: cg, the call graph
cg ← new CallGraph()
for thread ∈ logs.threads do
    stack ← new Stack()
    for log ∈ logs.get(thread) do
        if isMethodEntry(log) then
            n ← cg.addMethodNode(log)
            if stack is not empty then
                top = stack.peek()
                cg.addInvokeRel(top, n)
            end
            stack.push(n)
        else
            stack.pop(){method exit}
        end
    end
end
return cg
```

第四章　X

第五章　X

第六章　X

第七章　X

参考文献

[1] COLLARD M, DECKER M, MALETIC J. srcml: An infrastructure for the exploration, analysis, and manipulation of source code[C]//Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM'13) Tool Demonstration Track, Eindhoven, The Netherlands. [S.l.: s.n.], 2013: 1-4.

致　　谢

感谢亲爱的家人及亲友的理解和鼎力支持。你们的关爱和默默付出使我能安心学业并顺利完成论文。

二零一五年五月

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