南京大學

研 究 生 毕 业 论 文

（申请工程硕士学位）

论 文 题 目 基于动静态分析的安卓测试意图生成技术

作 者 姓 名 张晶

学科、专业名称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 陈振宇 教授，房春荣 助理研究员

2022 年 **5** 月 **19** 日

学　　号：**MF20320218**

论文答辩日期：**2022** 年 **05** 月 **20** 日

指 导 教 师：　　（签字）

Android Test Script Intent Generation Technique based on Dynamic and Static Analysis

by

Jing Zhang

Supervised by

Professor **Zhenyu Chen**，Assistant Research **Chunrong Fang**

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
Master of Engineering
in
Software Engineering

Software Institute
Nanjing University

May 19, 2022

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于动静态分析的安卓测试意图生成技术

工程硕士（软件工程领域）专业 2020 级硕士生姓名：张晶

指导教师（姓名、职称）：陈振宇 教授，房春荣 助理研究员

摘　要

为了降低人工测试活动的成本并保证软件的质量，自动化测试技术被广泛应用于移动端应用上。版本的快速迭代过程中，会积累非常多的测试脚本。开发人员可以通过注释（代码的自然语言描述）来理解这些测试脚本。但是根据我们对 github 上 100 个开源软件的测试脚本注释情况统计得出，只有 34% 的测试脚本有零星的注释，剩余的测试脚本完全没有注释。这些测试脚本注释的缺失，会不易于开发人员对于脚本进行管理、检索以及二次利用。同时，由于测试脚本的语句通过绑定 UI 界面的控件来进行用户交互操作，所以这些测试语句和控件的源代码无法直接关联起来，这也导致了传统的代码理解技术无法直接应用在测试脚本上。

本文提出了一种结合动静态程序分析的测试脚本意图报告生成方法，有效的帮助开发人员快速理解并管理海量测试脚本。其中，意图指用户对特定测试操作的应用行为期望，把期望明确呈现在应用的测试文档中。动静态分析的区别在于分析过程中应用程序是否具备良好的运行环境且需要处于正在运行的状态中，我们将两者结合起来，可以更高效、准确地获取更多有效信息。测试脚本通过绑定安卓界面的控件来进行一些点击、长按或者是输入的操作，有 ID 或者 XPath 两种绑定方法。其中，ID 是开发人员在安卓应用开发过程中定义的控件名称；XPath 是一种 XML 语言，通过 XML 文件中布局的层次结构来唯一确定一个控件。ID 与源代码具有强关联性，但是由于开发人员对于应用代码设计的复杂性和多样性，我们无法直接将测试脚本中的控件 ID 与安卓源码映射起来。针对 ID 类控件，使用了动静态分析相结合的方式去找到对应的 UI 响应函数，再利用 code2seq 模型生成函数对应的意图信息。其中，静态分析是对安卓源代码进行模式匹配，通过安卓开发经验以及对于开源项目的一些调查，我们总结了五种当下最常用的绑定控件的 UI 响应函数的方式，并且根

据优先级情况对匹配顺序进行调整。在静态分析中不能获取 UI 响应函数的 ID 类控件，我们对其进行动态控制流图法分析。我们使用 soot 框架对安卓 APK 包进行插桩，利用安卓日志系统在运行过程中来记录并输出函数调用情况，并得到当前 ID 控件的控制流图（control flow graph），从而在源代码中检索得到 UI 响应函数。针对 XPath 类控件，我们利用 AppiumDriver（Appium 框架提供的接口类）在程序运行态保存应用当前的控件的图片、屏幕截图与其 XML 布局文件。如果为文字类图片我们使用 XPath 路径中的 content-desc 字段提取文字信息，反之我们将其输入编码器解码器模型中得到图片相对应的意图信息。最后，我们将两部分的意图结果进行整合，得到该测试脚本的意图报告。

我们设计了实验来证明该方法的可靠性。实验选取了 10 个安卓移动应用和 50 个测试脚本，其控件的图像/代码映射率为 84.71%，生成意图结果的映射率为 77.65%。实验从机器翻译的四大指标 BLEU，CIDEr，METEOR 以及 ROUGE-L 来验证了其有效性，我们分别计算了 XPath 分析部分、ID 分析部分、以及整个方法的 NLP 指标。比较结果表明，我们提出的生成测试脚本意图报告的方法相比 code2seq 模型的评分更高，这体现了我们模型的可靠性。

关键词： 测试意图生成，动态分析，静态分析，代码语义生成

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Android Test Script Intent Generation Technique based on Dynamic and Static Analysis

SPECIALIZATION: Software Engineering

POSTGRADUATE: Jing Zhang

MENTOR: Professor **Zhenyu Chen**，Assistant Research **Chunrong Fang**

Abstract

To reduce the cost of manual testing activities and to ensure the quality of software, automated testing techniques are widely used on mobile applications. A very large number of test scripts are accumulated during the rapid iteration of versions. Developers can understand these test scripts through annotations (natural language descriptions of the code). However, according to our statistics of 100 open source software test scripts' annotations on github, only 34% of the test scripts have a few annotations, and the remaining test scripts have no annotations at all. The lack of annotations in these test scripts will make it hard for developers to manage, retrieve, and reuse the scripts. At the same time, since the statements of the test script perform user interaction by binding the widgets of the UI interface, the test statements cannot be directly related to the source code of the widgets, which makes it impossible to directly apply traditional code understanding techniques to the test scripts.

In this paper, we propose a test script intent report generation method basing on dynamic and static program analysis, which effectively helps developers to quickly understand and manage a large number of test scripts. The intent refers to the user's expectation of the application behavior for a specific test operation, and the expectation is clearly presented in the test documentation of the application. The difference between dynamic and static analysis is whether the application has a running environment and needs to be in a running state during the analysis, we combine the two to get more effective information accurately. Test scripts bind widgets in the Android interface to perform some clicks, long presses, or input actions, such as binding a widget by ID or XPath. The ID is strongly correlated with the source code, but we cannot directly

map the widget ID in the test script to the Android source code due to the complexity and variety of application code design. We use a combination of dynamic and static analysis to find the corresponding UI response functions for ID widgets, and then use the code2seq model to generate the intent information for the functions.Through our experience in Android development and some surveys on open source projects, we have summarized the five most common ways to bind the UI response functions of widgets, and adjusted the matching order according to the priority,which we call it static pattern matching. We analyze the dynamic control flow graph method for ID widgets that cannot get UI response functions in static analysis. We use the soot framework to stake the Android APK package, use the Android logging system to record and output the function calls during the runtime, and get the control flow graph of the current ID widget to retrieve the UI response function in the source code. For XPath widgets, we use AppiumDriver (an interface class provided by the Appium framework) to save the image, screenshot and XML layout file of the current widget in the runtime of the application. If it is a text image we use the content-desc field in the Xpath path to extract the text information, and vice versa we input it into the encoder-decoder model to get the intent information corresponding to the image. Finally, we integrate the intent results of both parts to get the intent report of the test script.

We design the experiments to demonstrate the reliability of our method. Ten Android mobile applications and 50 test scripts are selected for the experiments, and the image/code mapping rate of their widgets is 84.71% and the mapping rate of the generated intent results is 77.65%. The experiments verify the effectiveness from the four major metrics of machine translation, BLEU, CIDEr, METEOR, and ROUGE-L, while we calculated the XPath analysis part, the ID analysis part, and the whole method separately. The comparison results show that our proposed method for generating test script intent reports has higher scores compared to the code2seq model, which reflects the reliability of our model.

keywords: Test Intent Generation, Dynamic Analysis, Static Analysis, Code Understating, Mobile App Testing

目　　录

插图清单

附表清单

第一章　绪论

1.1　选题的背景和意义

二十一世纪，移动端设备发展迅速。从 2009-2017 年智能手机操作系统占据的全球市场份额 [3] 可以看出，凭借近百分之八十的市场份额，安卓系统遥遥领先于其他移动操作系统，全球有超过十亿台安卓设备。根据 Statista 的数据，2014-2022 年间美国的安卓手机用户逐年增加，到 2022 年有大约 1.3 亿的安卓手机使用者 [2]。安卓移动应用被广泛应用于智能手机、手表等移动设备上和一些嵌入式的内部系统。截止 2021 年 12 月，谷歌应用商店内的上架应用达 260 万个 [1]。由于互联网的高速发展和现代人对于移动端应用的依赖性增强，各个分类的应用程序都需要快速迭代去保持自己的竞争力。与此同时，需要测试去保证应用的质量。一些人工测试方法需要大量人力成本去做相对重复的工作，自动化测试中的白盒测试则可以预先定义用户行为，完成固定的流程，一个脚本可以自动化的多次运行。自动化测试技术广泛应用起来。长此以往，会积累很多自动化的测试脚本。

通过对 github 上的 100 个开源安卓项目的测试脚本进行统计发现，其中，66 个安卓应用的测试模块没有任何注释，只有 1 个应用的测试脚本有非常详尽的注释，剩下的 33 个应用中只有零星的注释。这说明了开发人员在开发脚本的时候注释率极低，这样会导致版本迭代之后其复用性和理解性都较差，不利于开发人员管理脚本。在软件开发和维护中，开发人员会在程序理解上花费接近百分之六十的时间 [5]。恰当并准确的注释可以带来一定程度的帮助，因为程序员可以通过自然语言描述信息来快速定义这段代码在项目中的作用。例如，在一次完整的软件版本迭代之后，研发团队需要对整个应用进行回归测试。由于之前的脚本缺乏注释，无法复用之前没有进行代码修改模块的测试脚本，需要针对所有的应用回归点进行测试脚本的开发。如果之前的脚本所测试的模块代码进行迭代，因为注释的缺乏也无法在原有测试脚本上进行修改。

基于以上问题，本文提出了一种基于动静态的自动化生成安卓应用测试脚本意图信息的技术，可以在保证应用质量的同时节约大部分人力成本，使应用

迭代的过程更加高效。针对安卓源代码、安卓 APK 包、安卓自动化测试脚本等信息，结合动态分析和静态分析两种技术，生成脚本的意图信息。在这个过程中，我们主要通过对用户界面 XML 文件、图像的分析和安卓源代码的分析去提取有用信息。

1.2 国内外研究现状

1.2.1 图形用户界面分析技术

安卓的自动化测试框架是通过用户界面的图标、输入框等 UI 控件去驱动应用。结合计算机视觉技术和人工智能，可以理解应用用户界面的图像信息并作出转换。

由于图形用户界面都是开发人员通过代码去绘制的，所以一些最新的技术通过设计师给出的用户界面可以自动化生成其对应的源代码。Beltramelli 等人 [6] 提出了一种自动将定义的 UI 图形用户界面稿转换为多个移动端的前端代码的方法，名为 pix2code。在这篇文章中，作者利用深度学习模型来进行端到端的训练，用于自动化进行用户界面的逆向工程，并从单一的输入图像中生成对应代码。该方法的显着优势之一是不需要人工标记的数据。网络可以通过在图像序列对上进行训练来对图形组件和相关标记之间的关系进行建模。对于三个不同的移动端平台（即 iOS、Android 和基于 h5 的技术），准确率超过 77%。Chen[7] 等人提出了一个自动化的跨平台 GUI 代码生成框架，可以在两个移动平台（安卓、iOS）之间转换 GUI 代码。框架包含三个阶段，第一阶段是 GUI 组件识别，用图像处理技术提取 UI 页面中的组件，利用深度学习算法（即 CNN 分类）来识别组件的类型；然后是组件类型映射，将确定的组件类型映射到目标平台的相应组件上；三是 UI 代码生成，根据组件的类型和它们的特性，生成 GUI 的实现代码。

一个完整的用户界面截屏不是单一的，而是由许多 UI 部件组成的，识别并针对各个部件进行提取十分重要。Nguyen 等人提出了一种可以识别移动应用的图形用户界面的技术叫做 REMAUI[15]。在一个给定输入的移动端屏幕截图上，REMAUI 可以通过计算机视觉和光学字符识别 (OCR) 技术识别用户界面元素，如输入框、选择框、容器和按钮等。

在自动测试的过程中，用户界面的图片或者文字都包含了许多语义信息。结合机器学习技术，可以通过对图片的识别去提取并获得图片或文字的语义

信息。Xiao 等人提出了 IconIntent 框架 [8]，该框架通过程序分析技术将图标和 UI 部件联系起来，并采用计算机视觉技术将相关的图标分为八个敏感类别。IconIntent 可以与各种隐私分析工具结合起来，例如 GUILeak[4]，用于帮助开发者追踪隐私政策中提到的信息类型图标。Xi 等人提出了 DeepIntent 这个框架 [10]，它使用深度图标行为学习，从大量流行的应用程序中学习图标行为模型并检测出意图和行为的差异。DeepIntent 提供了程序分析技术，将意图（即图标和上下文文本）与用户界面宽度联系起来，并根据程序行为推断 UI 控件的标签。DeepIntent 使用深度学习技术，对图标及其上下文文本进行联合建模来学习图标行为模型，并且通过计算离群值来检测意图与行为之间的差异。Zhang 等人 [16] 引入了启发式方法（如用户界面分组和排序）和额外的模型（例如识别 UI 内容、状态、互动性）识别图像用户界面的 UI 控件并生成语义信息。同时，其建立了屏幕识别来生成可及性元数据以增强 iOS VoiceOver。

1.2.2 代码语义生成技术

安卓应用在执行自动化测试脚本的过程中，其核心是运行了某一部分源代码，故针对于用户响应事件的源代码的语义理解也十分重要。目前的生成代码注释技术主要分为三大类，基于模版匹配的方法、基于信息检索的方法（IR）和基于机器学习的方法。

模版匹配需要预先定义一组自然语言，再使用一些规则去匹配代码从而生成自然语言。Sridhara 等人 [18] 使用软件词汇使用模型（Software Vocabulary Usage Model）提出了一个基于规则的模型，提取了当前输入的 Java 方法的摘要。Moreno 等人 [19] 事先定义了一些规则来筛选有效的信息，并通过将这些信息进行总结来为 Java 类生成摘要。这些基于规则的方法已经被扩展应用到一些包含特殊用途的代码片段，如测试用例 [20] 和代码修改 [21]。模板方法通过从输入的源代码中提取关键信息来生成该段的自然语言描述。

IR 方法被广泛用于代码的摘要生成，通常从类似的代码片段中搜索有用的语义信息。Haiduc 等人 [22] 应用矢量空间模型（VSM）和潜意识语义索引 (LSI) 来为类和方法生成基于术语的注释。他们的工作被 Eddy 等人 [23] 所复制和扩展，他们利用分层主题模型来生成注释。Wong 等人 [24] 应用代码克隆检测技术来寻找类似的代码片断，并使用类似代码片断的注释。这项工作类似于他们之前的工作 AutoCommen[25]，该工作从 Stack Overflow 中挖掘开发人员写的描述以自动生成注释。

最近几年的研究将机器学习技术应用在代码摘要生成上。Iyer 等人 [26] 提出了一种利用 RNN 网络去生成 C# 代码片段和 SQL 语句的自然语言描述信息的方法。该方法将输入的代码段看作纯文本信息，利用 LTSM[49] 建立了注释的条件分布模型。Allamanis 等人 [27] 将一个神经卷积网络模型应用于这个问题，该模型可以将源代码片段极度总结为类似名字的简短摘要。这些基于学习的方法主要是学习源代码中的潜在特征，如语义、格式等。根据这些学习到的特征序列生成自然语言描述信息，这些实验结果证明了深度学习方法在代码理解上的有效性。Hu 等人 [28] 提出了一种自动生成 Java 的代码片段的自然语言描述的方法 DeepCom。DeepCom 应用自然语言处理 (NLP) 技术，根据从代码语料库中学习特征序列并映射到自然语言的语料库中生成代码片段的描述信息。code2seq[30] 提出了一个代码到序列的模型，该模型考虑了源代码的独特句法结构和自然语言的序列模型。其核心思想是对代码片断的抽象语法树中的路径进行采样，生成摘要语法树的路径，用长短期记忆模型 [49] 对这些路径进行编码，并生成目标序列。

1.3 本文的工作

安卓移动应用的快速迭代过程中，Appium 自动化测试脚本大量堆积且注释缺失。其中，Appium 测试框架通过当前运行页面的信息进行控件绑定和用户行为交互，无法直接映射到安卓源代码中，所以传统的代码理解技术无法直接应用。基于这些问题，我们提出了利用动静态分析生成测试脚本意图报告的解决方案，帮助开发人员理解并管理测试脚本。本文的主要工作包括：

（1）XPath 类控件图片文字信息提取：针对 Appium 测试脚本中的 XPath 类控件，我们利用动态分析法提取其对应的图片文字信息。运行测试脚本的过程中，我们对屏幕的截图信息和 XML 文件信息进行保存，利用 XPath 路径找到在 XML 文件中控件对应的坐标和文字描述。

（2）ID 类控件静态模版匹配：针对 Appium 测试脚本中的 ID 类控件，我们先使用静态分析法模版匹配来找到控件对应的 UI 响应函数。根据对安卓 API 的调研和开发过程中的经验，我们总结了五种安卓设置回调的方法，并抽象成模版。五个模版根据其特点和出现频率进行优先级排序。

（3）ID 类控件动态控制流图法：针对在（2）模版匹配中无法找到 UI 响应函数的 ID 控件，我们通过动态分析控制流图法来定位其 UI 响应函数。我们

使用了 soot 框架进行 APK 插桩，将源代码中每个函数开始和结束位置都插入一句安卓日志输出其函数签名信息。运行安卓项目后，我们收集并过滤日志，得到该控件的控制流图并在源代码中定位具体的 UI 响应函数。

（4）图像理解模型：我们基于 [48] 构建了一个编码器解码器结构的深度学习模型。对于图像输入使用了卷积神经网络来提取特征，并输出一个概率向量，映射到语料库中的一个词。

（5）代码理解模型：我们根据 code2seq[30]，实现了一个生成代码语义信息的模型。使用编码器构建代码片段的抽象语法树，在输入解码器中得到目标序列，并在其中加入了注意力机制。

1.4 本文的组织结构

第一章主要介绍了本文的选题背景、主要工作与相关的研究现状。首先调查了在开发过程中安卓应用测试脚本的注释稀缺性，阐述了生成测试脚本意图的必要性。根据安卓应用运行时的图形用户界面和源代码中的用户响应事件两部分特性，介绍了 UI 界面的图片识别技术和代码注释生产技术的研究现状。最后，介绍了本文的组织结构以及整体工作。

第二章针对文章中使用的技术进行介绍。首先是程序的动静态分析技术以及其区别和优缺点。其次介绍了安卓系统以及相关的技术，包括 Android Debug Bridge（adb）、Appium 自动化测试框架、jarsigner（APK 签名工具）、soot 框架（java 优化框架，对 java 项目进行插桩），dex2jar 工具（将 APK 包进行反编译生成 jar 包）。然后是一些机器学习相关的技术，包括了 code2seq 模型（用于生成代码语义）和 VGG-16 模型（生成图像意图中所使用的模型）。

第三章介绍了基于动静态程序分析的安卓测试意图生成技术的整体技术框架以及每部分的模型。根据 Appium 自动化框架的特点，我们将测试语句分为 XPath 和 ID 两种类型进行分析。通过类型特点分别应用不同的方法，ID 类控件我们使用静态模版法和动态控制流图法；XPath 类控件我们提取其图像文本信息，最后将两部分信息输入机器学习模型中生成意图并进行整合。

第四章我们使用 github 上的开源的 10 个安卓应用和对应的 50 个测试脚本进行实验。实验主要通过两部分来证明技术的可行性。第一部分是测试语句对应的 XPath 分析或者 ID 分析的映射率。XPath 的映射是指定位的控件图像和 XML 文件等媒体信息，ID 的映射是指 UI 回调函数的映射，其图像/代码总映

射率达 84.71%。第二部分从 BLEU 指数、CIDEr 指数、METEOR 指数等评分来验证了该方法的有效性，结果显示我们提出的方法优于 code2seq 模型。

第五章对整个系统进行需求分析和架构设计。针对系统，我们分析功能性需求和非功能性需求，给出相关的用例图以及每个用例的需求规格说明。此外，我们还设计了系统的架构和系统的模块划分。

第六章对系统进行详细设计。我们将系统分为文件模块、APK 预处理模块、自动化运行模块和测试脚本分析模块四大部分。针对每个模块给出具体的介绍、实体类图和核心功能的详细代码。

第七章对整个方法进行总结与展望。总结了本文中的各项工作和贡献，并针对本文中不足的地方提出可靠的改进意见。

第二章　相关技术概述

2.1　安卓应用静态分析

静态程序分析通常将程序的源代码或目标代码作为输入，在不执行该代码的情况下对其进行检查，并通过检查代码结构、语句序列以及在不同的函数调用中如何处理变量值来获得结果。静态分析的主要优点是分析会涵盖项目中的所有的代码。这与动态分析不同，在动态分析中，部分代码只能在某些特定条件下执行，而这些条件在分析阶段是不一定会被满足。一个典型的静态分析过程开始于将被分析的应用代码表示为一些抽象的模型，例如，调用图、控制流图或 UML 类图，来用于分析。这些抽象模型实际上提供了一些接口，用于支持上层的分析，如污点分析等。其他信息，如变量的值也可以被收集起来，支持静态分析更深入的验证，例如，数据流分析。

控制流分析。控制流分析是一种技术，用于显示在一个给定的程序中控制的层次流是如何排序的，使程序的所有可能的执行路径都可以被分析。通常，控制序列以控制流图（control flow graph，CFG）的形式表示。控制流图中每个节点代表一个基本的代码块（语句或指令），而每个节点代表语句或指令，而每条有向边表示两个节点之间可能的控制流。

数据流分析。数据流分析 [30] 是一种在程序的每一点上计算一组可能值的技术。这一组取决于使用数据流分析所要解决的问题的种类。例如，在达到定义问题中，人们想知道定义的集合（例如，语句 int x=3;）在每个程序点上可达到的定义集。在这个特定的问题中，程序点 P 的可能值集合是指到达 P 的定义集（也就是说，变量在到达 P 之前没有被重新定义）。点对点分析包括计算一个静态指针表达式（或只是一个变量）在程序运行时可以指向的所有数据的静态抽象。

Sufatrio 等人 [31] 提出了一个关于安卓安全性的检测，包括静态和动态的方法。这项调查首先介绍了一个分类法，其中有五个主要类别，基于现有的 Android 安全解决方案。然后，它将现有的工作归入这五个类别，从而对它们进行比较研究。该调查显示，大多数研究解决方案解决 Android 中的安全问题

是利用静态分析。

2.1.1 soot 框架

soot 是一个用于优化 Java 项目的框架 [34]，使用了静态分析方法。该框架用 Java 实现，输入为支持三种用于表示 Java 字节码的中间表示法。Baf 是 Java 基于堆栈的字节码的精简表示；Jimple 是一种类型化的三地址中间表示法，适合用于代表 Java 的字节码。Grimp，是 Jimple 的一个聚合版本。对类文件的优化方法是，首先将基于堆栈的的字节码转换成 Jimple，一种更适合于传统程序优化的三地址形式；然后再将优化后的 Jimple 转换为的字节码。

soot 中核心的类有 SootClass、SootMethod 和 SootField。其中 SootClass 是 Java 方法的 Soot 表示，其中，SootMethod 包括了这个 SootClass 中的所有方法，SootField 中包括了所有的变量。这个设计类似于 Java 反射。soot 框架的主要贡献有：从字节码到冗长的 Jimple 转换（使用 Clark Verbrugge 的代码作为原型）；通过复制和常量传播来压缩 Jimple 的冗长代码；实现流分析框架和各种流程分析，如实时变量分析；实现局部变量拆分和局部变量打包；实现不可达的代码和无效赋值的检测和删除。

2.2 安卓应用动态分析

与静态分析不同，动态分析是在安卓应用的运行时进行检测的。动态分析通常被认为比静态分析更复杂，因为不仅需要应用程序安装包，还需要程序的可执行环境以及模拟用户事件（即输入、触摸和点击）。常见的动态分析测试有以下几种：模糊测试、导向型随机测试和基于搜索的测试 [36]。它们之间的区别包括测试效果、测试生成和漏洞覆盖。

模糊测试是一种自动化的软件测试技术，包括提供无效的、意想不到的或随机的数据作为输入，以达到测试目的。模糊测试可以监测程序的异常情况如崩溃，或失败的内置代码断言或发现潜在的内存泄漏 [37]。导向型随机测试是一种混合的软件验证技术，融合了符号和具体的执行。符号执行将程序变量视为符号变量，而具体的执行对特定的输入路径进行测试 [38]。基于搜索的测试使用启发式的优化搜索技术来自动完成测试任务，如模拟退火和遗传算法 [39]。

动态分析相对于静态分析来言，测试代码的覆盖率不可能像静态分析一样

达到 100%，其测试所需要的环境也相对复杂。但是，动态分析可以检测出静态分析中无法得到的问题，如内存泄漏和代码漏洞。

2.3 安卓开发相关概念

2.3.1 XML 布局文件

Android 应用程序的用户界面结构由布局来定义。应用程序的用户界面的结构是由布局定义的，这些布局可以用 XML 配置或 Java 代码声明。XML 布局机制提供了一个定义明确的词汇表，对应于视图类、子类以及它们可能的事件处理程序。

2.3.2 Logcat

安卓日志系统提供了一个收集和查看系统调试输出的机制。来自不同应用程序和系统部分的日志被收集在一系列的循环缓冲区，然后可以通过 Logcat 命令查看和过滤。在开发过程中，可以通过安卓 Log 类的静态方法输出日志信息，日志的标签类型都可以由开发人员自定义。日志的重要等级由高到低为错误（Error，e），警告（Warn，w），信息（Info，i），调试（Debug，d），详细（Verbose，v）。

2.3.3 UI 回调函数

Android 中还使用了一些回调方法来处理各种事件。特别是，UI 事件由系统检测，并通过回调方法通知开发者应用程序（例如，当用户点击一个按钮时做出反应）。Android 中定义了多个的这样的回调方法（例如，输入事件、点击事件），与生命周期方法类似，这些回调方法会生成一个更完整的控制流图。

2.4 安卓相关工具

2.4.1 Appium 框架

Appium 是一个开源的自动化测试框架 [58]，同时也是一个跨平台的解决方案，不仅支持 iOS 和 Android 的原生应用，也支持混合应用和移动 Web 应用的测试。Appium 是支持跨平台的测试框架，是因为底层实现同时内置了 iOS 的 XUCITest 和安卓平台的 UiAutomator。原生应用是指完全依靠移动设备部署的代码进行用户交互、数据的传输以及处理，而混合应用是利用移动应用中内置的 WebView 和使用前端技术的 h5 页面进行交互，类似于一个内置的浏览器。由于混合应用可以减轻移动应用包的大小，也可以减轻一部分移动端的资源压力，并随时可以进行 h5 页面的上线部署，在当下十分流行。测试环境支持真实环境和模拟测试，以及支持本地和云部署。Appium 支持多种程序语言开发测试程序，比如：pyhton、java、Ruby、js、php、C# 等。

Appium 的设计理念有四点：测试时的 APP 和最终发布的 APP 应该是相同的，不能为了进行测试就改变 APP 源代码，这说明了 Appium 测试框架对源码的无侵入性。其原理是底层基于安卓和 iOS 的原生测试框架开发的; 能够使用任何语言和框架来写测试，不会被特定编程语言和框架限制; 使用标准化的自动化测试规范和 API，不会因为版本迭代而经常变更;Appium 是一个开源框架。

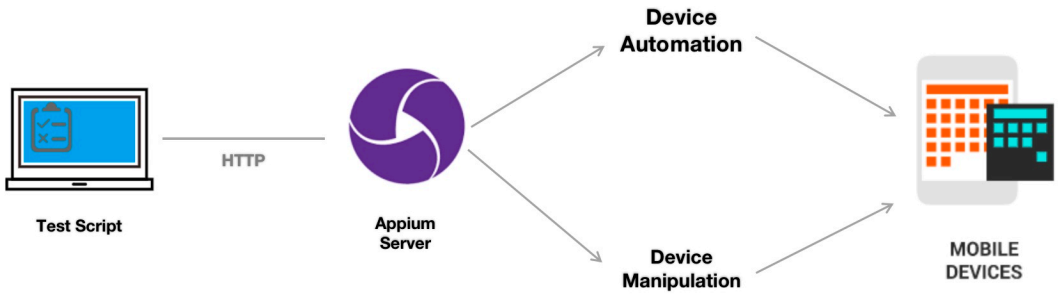

图 2-1: Appium 系统架构

Appium 的整体框架如图 2.1 所示。整体设计是一个 C/S 架构，由三大部分组成。首先，Test Script 是测试脚本，可以使用任何语言进行开发，在测试脚本中可以定义被测应用、测试流程等。Appium Server 是 Appium 在 pc 端的服务器，是一个 JavaScript 代码实现的程序，运行在 Node.js 之中。测试脚本和被

测应用以及设备通过和服务端的 http 连接进行数据的传输。测试脚本以每个库（java、c# 等）的特定方式启动与服务器的会话（session），但它们最终都会向服务器发送一个 POST 请求，其中有一个称为”desired capabilities” 的 JSON 对象。服务器将启动自动化会话，并响应一个会话 ID，用于发送进一步的命令。

2.4.2 jarsigner

jarsigner 工具可以签署 jar 文件，也可以验证签名的 jar 文件的签名和完整性。Java 项目的 jar 包可以将类文件、图像、声音和其他数字数据打包在一个文件中，以便更快、更容易地分发。maven 工具使开发人员能够编译生成 jar 包。从技术上讲，任何压缩文件也可以被认为是一个 jar 文件，尽管当由 jar 命令创建或由 jarsigner 命令处理时，jar 文件也包含一个 META-INF/MANIFEST.MF 文件。

数字签名是由一些数据（被签名的数据）和一个实体（个人、公司等）的私钥计算出来的一串比特。与手写签名类似，数字签名有许多有用的特点。它的真实性可以通过使用与生成签名的私钥相对应的公钥进行计算来验证。只要私钥是保密的，就不能伪造数字签名。同一个数字签名被签名的数据是唯一的。签署的数据也不能被改变，如果数据被篡改，那么其签名就不能被验证为真。要为一个文件生成签名，该实体必须首先有一个与之相关的公/私钥对和一个或多个验证其公钥的证书。证书是一个实体的数字签名声明。

在生成安卓的可发布版本的 APK 包时，必须要手动进行签名，有以下三个原因。一是 APK 使用同一个证书进行签名后，用户可以安全升级迭代之后的应用。二是可以保证应用的模块化。安卓系统可以允许同一个证书签名的多个应用在一个进程中运行，系统把进程当做一个应用。我们可以将我们的应用部署为一个模块，用户可以升级一个独立的模块。这有利于应用的松耦合。三是使用相同的证书签署多个应用程序可以在应用程序之间以安全的方式共享代码和数据。

jarsigner 命令使用钥匙库中的钥匙和证书信息来为 jar 文件生成数字签名。钥匙库是一个私人钥匙及其相关的 X.509 证书链的数据库，可以验证相应的公钥。keytool 命令被用来创建和管理钥匙库。

jarsigner 命令使用一个实体的私钥来生成一个签名。签名的 jar 文件包含来自钥匙库的证书副本，用来签名文件的私钥的公钥。jarsigner 命令可以使用签名的 jar 文件中的证书（在其签名块文件中）验证 jar 文件的数字签名。jarsigner

命令可以生成包含时间戳的签名，让系统或部署者（包括 Java 插件）检查 jar 文件是否在签名证书仍然有效时被签名。此外，API 接口可以允许应用程序获得签名的时间戳信息。

2.4.3 adb 工具

Android Debug Bridge（adb）是一个内置于安卓 SDK 的命令行工具。它可以让用户与移动设备通信并执行各种操作，如安装和调试应用程序，来回复制文件，以及利用 Unix shell 来运行命令等。adb 还可以在程序运行期间，对移动设备进行截屏和录屏等操作。结合 shell 命令，adb 可以存储许多在安卓应用运行期间的数据（如媒体数据、日志信息）。

adb 是一个客户/服务器程序，包括三个部分。

客户端。客户端从 PC 端的终端或 shell 脚本运行 adb 命令。它的作用是向服务器发送命令，与一个或多个模拟器实例进行交互。

Daemon（adbd）。Daemon 在移动设备上作为一个后台进程运行。它的作用是通过 USB 或模拟器的 TCP 与 adb 服务器连接。当设备与守护进程成功连接时，adb 服务器就认为它是在线的。

服务器。服务器在 PC 端上作为一个后台进程运行。目的是在连接或移除设备时感知 USB 端口或者 wifi，维护一个已连接设备的列表，并为每个设备分配不同的状态，如在线、离线等。

当调用 adb 命令时，客户端将首先检查 adb 服务器是否正在运行。如果没有，服务器会尝试与守护程序连接，直到找到它们。然后会设备上收到授权请求，并将其绑定到本地 TCP 端口。之后，将继续监听该特定端口的命令。移动设备可以通过 USB 或者 wifi 两种方式连接到 PC 端。

2.4.4 Dex2jar

Dex2jar 工具允许将 APK 文件转换为 jar 文件并查看源代码。将 APK 的 classes.dex 文件转换为 classes.jar 文件或反之亦然，是 Dex2jar 的核心功能。可以通过 Java 反编译器查看 Android 应用程序的源代码。此外，也可以直接从 classes.dex 文件中获得”.smali ” 文件，用这种格式改变应用程序的源代码。

2.5 VGG-16

VGG-16 是一个应用于 ImageNet（一个用于视觉对象识别软件研究的大型视觉数据库项目）的卷积神经网络（CNN）架构 [57]。VGG-16 架构是由牛津大学的 Karen Simonyan 和 Andrew Zisserman 于 2014 年通过文章”Very Deep Convolutional Networks for Large-Scale Image Recognition ” 研发并介绍的。VGG 的全称是 Visual Geometry Group，是牛津大学的开发这个架构的研究小组，而”16” 代表这个模型有 16 层。

VGG-16 模型在 ImageNet 中排名前五，并有 92.7% 的测试准确率。ImageNet 是一个由超过 1400 万张属于 1000 个类别的图像组成的数据集。它是 2014 年提交给 ImageNet 大规模视觉识别挑战赛（ILSVRC）的著名模型之一。其通过用多个 3×3 核大小的过滤器相继取代大核大小的过滤器（第一和第二卷积层分别为 11 和 5），对 AlexNet 架构进行了改进。VGG-16 使用 NVIDIA Titan Black GPU 进行了数周的训练。

VGG-16 被用于许多深度学习图像分类技术中，由于其易于实现而受到欢迎。由于 VGG-16 具有的优势，它被广泛用于学习应用中。VGG-16 是一个 CNN 架构，曾在 2014 年赢得 ImageNet 大规模视觉识别挑战赛（ILSVRC）。它仍然是迄今为止最好的视觉架构之一。

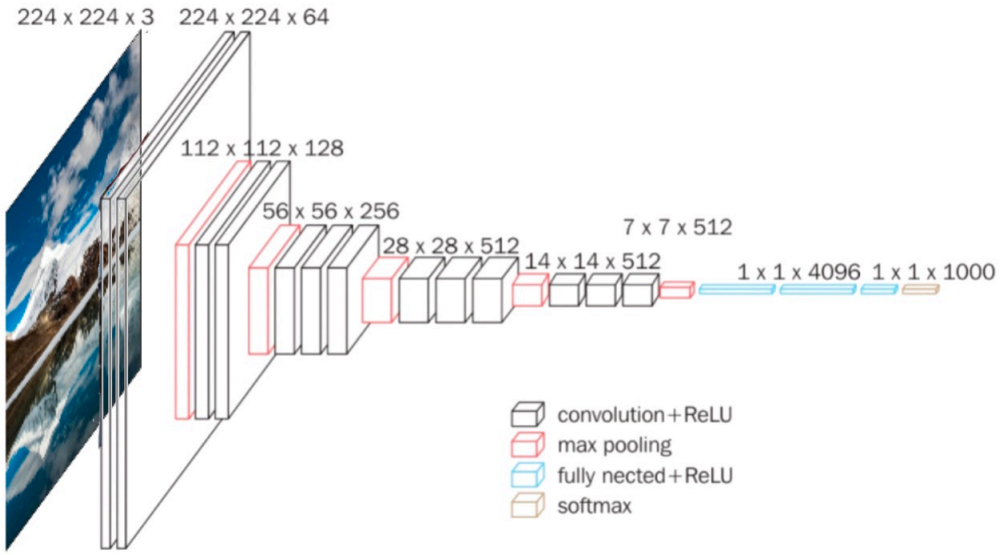

图 2-2: VGG-16 架构

图2-2展示了 VGG-16 模型各层的结构。在训练期间，输入是一个固定大小的 224x224RGB 图像。预处理是从每个像素中减去在训练集上计算的平均 RGB 值。图像通过卷积层的堆叠，其中使用了具有非常小的接受域的过滤器：3×3（这是捕捉左/右、上/下、中心概念的最小尺寸，并且具有与一个 7×7 相同的有效接受域）。在其中一个配置中，还利用了 1×1 卷积滤波器，它可以被看作是输入通道的线性变换（其次是非线性）。3×3 卷积层（convolution layer）的卷积跨度和卷积层输入的空间填充被固定为 1 像素，这确保了卷积后的空间分辨率得以保留。五个最大池化层，紧随一些卷积层之后，帮助进行空间集合。最大池化层（max pooling layer）是在一个 2×2 像素的窗口上进行的，跨度为 2。有三个全连接（fully connected layer）层跟在卷积层的后面（这些层在不同的结构中有不同的深度）：前两个层各有 4096 个通道，第三个层进行 1000 路 ILSVRC 分类，因此包含 1000 个通道（每类一个）。最后一层是 softmax 层。全连接层的配置在所有网络中都是一样的。

2.6 code2seq

code2seq 是由 Alon 等人在 2019 年发表在 ICLR 上的一个代码到序列（sequence）的模型 [30]。该模型考虑了源代码的独特句法结构和自然语言的序列模型。其核心思想是对代码片断的抽象语法树中的路径进行采样。摘要语法树的路径，用 LSTM[49] 对这些路径进行编码，并在生成目标序列时关注它们。长短期记忆（Long short-term memory，LSTM）是一种特殊的 RNN，主要是为了解决长序列训练过程中的梯度消失和梯度爆炸问题。简单来说，就是相比普通的 RNN，LSTM 能够在更长的序列中有更好的表现。实验部分使用 code2seq 来预测三个不同大小的数据集的方法名称，预测给定的部分和简短的代码片段的自然语言标题，并在两种编程语言中生成方法文档。code2seq 模型的表现明显优于最先进的神经网络机器翻译模型。

此外，code2seq 在关注输入路径的同时逐步解码输出序列，同时关注输入路径，因此可以生成未见过的序列。这与 code2vec[60]（Alon 等人，2019）相比，它有一个封闭的词汇表。

2.7 Vue 框架

Vue[50] 是一套用于构建和设计用户界面的渐进式前端框架。该框架是以自下而上的逐个应用设计方法开发的。虽然它与模型-视图-模型（MVVM）模式没有严格的联系，但 Vue 的设计原则部分是受其启发。与其他大多数大型框架不同，Vue 的核心库只关心视图层的技术要求，这使得其很容易上手，也很容易与其他第三方库集成。Vue 还可以与现代工具链和各种支持类库结合使用，为复杂的单页应用提供持续的保证。同时，Vue 兼容现在所有主流的浏览器。与市场上比较流行的 React[51] 框架相比，在保留了 Virtural DOM[52]、响应式组件化的视图组件以及对核心库的关注的同时，其运行时性能得到了优化。与 React 相比，Vue 更强调用户体验，这使得它更容易上手，如 HTML&CSS 的扩展采用了经典的 Web 技术，框架的构建采用了原生渲染，大大方便了程序员的开发工作。

2.8 Django 框架

Django[59] 是一款由 python 编写的开源 Web 应用框架，是 python web 框架中最为流行的应用框架之一。其主要目的是简单、高效开发数据库驱动的网站。同时，Django 还强调代码的复用，可以方便的将多个组件以“插件”的形式部署到整个框架中进行服务。该框架还具有强大的第三方插件，是一款高效便捷的 web 应用开发框架。

2.9 本章小结

本章主要针对在实验过程中，所使用的一些技术进行介绍。首先是动静态分析的区别和各自的优缺点，以及重点介绍了 soot 框架。其次是安卓开发框架内设计到的技术概念，包括 XML 布局文件、安卓日志系统 Logcat 和 UI 控件的回调机制。除此之外，介绍一些衍生的安卓工具，包括 Appium 测试框架、jarsigner 工具、adb 工具和 Dex2jar 工具。最后介绍了用于图像分析的 VGG-16 模型和提取代码语义信息的 code2seq 模型。

第三章　基于动静态分析的安卓测试意图生成技术

3.1　整体概述

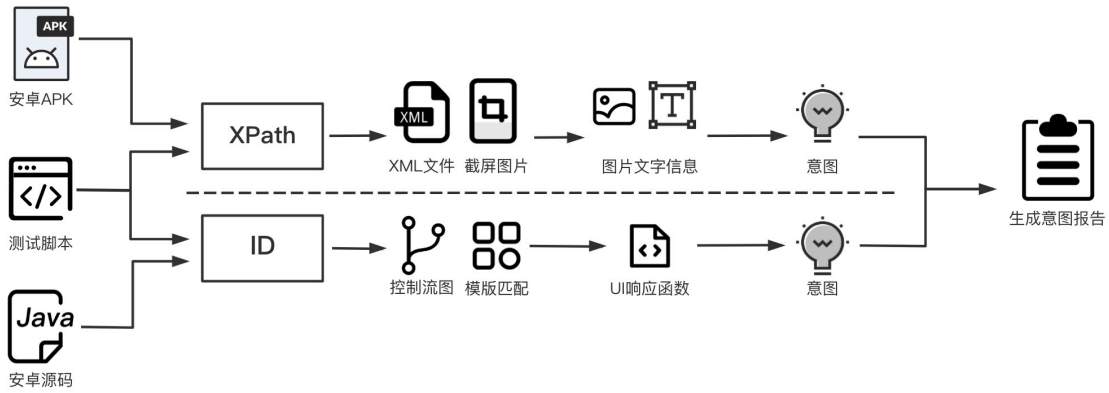

图 3-1: 整体架构

本方法的整体框架如图3-1所示。输入的数据分为三部分，安卓 APK 包、Appium 自动化测试脚本和安卓源码。从开发人员在编写 Appium 测试脚本的习惯来看，模拟用户事件的方法主要分为两大类。

```
1 driver.findElementById(”com.eaglx.callblocker:id/number”)
      .sendKeys(”139”);
2 driver.findElementByXPath(”\t\n” +
      ”/hierarchy/android.widget.FrameLayout/android.widget
      .Linearsayout/android.widet.ImageButton”)
      .click();
```

图 3-2: Appium 测试脚本示例

（1）通过 AppiumDriver.findElementByXpath() 去绑定这个控件（以下简称 XPath），并模拟用户事件。控件的 XPath 是根据安卓应用在运行时图形用户界面所生成的 XML 布局文件（2.3.1节）而动态生成的。

（2）通过 AppiumDriver.findElementById() 去绑定控件（以下简称 ID），主要的参数 ID 是开发人员在绘制安卓应用 UI 时自定义的，同时也会增加相应的 UI 回调事件（2.3.2节）。但是需要注意的是，由于开发人员编码的不规范性，ID 是有可能发生重复的，这种情况我们不予考虑。

3.2 ID 分析

由于控件 ID 与源码的强关联性，我们可以通过定位 ID 的 UI 回调函数去生成这行测试语句的语义信息。获取 UI 回调函数由两种不同的方法组成。第一个方法是静态模版匹配（3.2.1节），匹配的成功率为 48.27%。第二个方法是动态运行应用后通过日志系统生成动态调用图去定位 UI 回调（3.2.2节）。方法二可以补齐方法一在匹配的成功率不足的问题，不直接使用方法二的原因是静态分析的效率更高（2.2节）。两种方法结合使用，可以优化得到 UI 回调的成功率的同时提高效率。

获取到 UI 回调函数之后，将函数输入到编码器解码器模型中，得到语义信息（3.2.3节）。

3.2.1 静态模版匹配

开发人员在新建一个控件 ID 时，如果是非静态 UI 控件（只为用户界面的展示，不用于与用户交互），一定会为这个控件绑定其对应的 UI 回调函数。当用户的手指触摸屏幕时，安卓会将触摸行为分成三类，DOWN（手指放下）、UP（手指抬起）、CANCEL（取消）。通过这三类行为和对应的时常以及坐标来确定用户的点击或者滑动行为，并且处罚对应的 View 控件的 performClick() 方法。performClick() 会获取当前控件的 ClickListener（点击监听器），并触发 OnClick() 方法。通过对安卓 API 的调研和在安卓开发过程中的经验，总结了五个使用率最高的开发习惯并抽象为模版。

```
case R.id.action_pin_recipe_to_widget:
   //response code
return true;
```

图 3-3: 静态模版方法-模版 1 示例

（1）图3-3是模版 1 的一个实例。通常，开发人员会先获取一个父控件

（父控件是许多子控件的父类），通过 switch/case 控制语句去匹配 Id 的值去进行 UI 回调。使用这个方法可以同时处理多个 ID，程序的结构化更好。

（2）图 3-4是模版 2 的一个实例。其思路与模版 1 类似，只是控制语句从 swtich/case 变成了 if/else 语句。

```
if(item.getItemId() == R.id.action_settings) {
   //response code
}
```

图 3-4: 静态模版方法-模版 2 示例

（3）图 3-5是模版 3 的一个实例。在模版 3 中，主要是分为两种情况，第一是 Java 最简单的赋值方法，情况 2 是使用 @BindView 注解给控件的实例赋值。再通过 ID 绑定到 Java 实例变量之后，可以通过 setOnClickListener 给 View 类控件绑定一个监听器去处理控件的用户交互事件。

```
//情况 1
TextView tv = findElementsById(R.id.rv_steps);
//情况 2
@BindView(R.id.rv_steps)
TextView tv;

tv.setOnClickListener{
   //response code
};
```

图 3-5: 静态模版方法-模版 3 示例

（4）图 3-6是模版 4 的一个实例。开发人员可以直接使用安卓 SDK 提供的 AOP 注解 OnClick 去绑定 View 控件，从而绑定其 UI 回调函数。

```
@OnClick(R.id.btn_next_step)
void openNextStep(){
   //response code
}
```

图 3-6: 静态模版方法-模版 4 示例

（5）图 3-7是模版 5 的一个实例。区别于前四种模版是在 Java 文件中绑定了 UI 回调的方法，模版 5 是在 XML 布局文件中绑定方法（an-

```
< Button
...
android:id="@+id/nextButton"
android:onClick="onNext"
...
/>
```

图 3-7: 静态模版方法-模版 5 示例

droid:onClick="onNext"）。具体的 OnNext 函数在 XML 绑定的页面 Java 文件之中。

五个模版会进行优先级排序，保证匹配准确率的同时也提高响应速率、节省硬件的资源消耗。图3-8展示了一种同时满足模版 2 和模版 4 的情况，但是模版 2 的定位更加准确，所以模版 2 的优先级更高。除此之外，出现频率也与优先级的顺序有关。表3-1展示了实验中五个模版在 40 个 Id 的数据集上的匹配准确率。可以看出，模版 3 的出现概率最高，高达 20.51%。结合以上情况，优先级顺序分别为 3.2.1.4.5。提取的 UI 回调函数会将函数名修改为当前应用名称、当前脚本名称、当前 ID 名称的组合。

表 3-1: 五个模版出现的频率

	模版 1	模版 2	模版 3	模版 4	模版 5
频率（%）	10.86	7.24	20.51	7.24	2.41

```
@OnClick({R.id.fab_search, R.id.fab_region})
public void onClick(View view){
   fam.collapse();
   if (view.getId() == R.id.fab_search) {
      Intent intent = new Intent(this,
      SearchActivity.class);
      startActivity(intent);
   } else if (view.getId() ==R.id.fab_region) {
      Intent intent = new Intent(this,RegionActivity.class);
      startActivity(intent);
   }
}
```

图 3-8: 同时满足两种模版的情况

3.2.2 动态控制流图法

控制流图（CFG，Control Flow Graph）是指在程序运行过程中函数间的调用关系图，节点代表了一个函数代码块，边代表了调用关系。控制流图是通过静态分析得出的，可以输出所有可能的执行路径。但是想要获得具体控件 ID 的 UI 回调事件，还需要通过控制流图，也就是在运行过程中的调用关系图得出。借助于上文中提到过安卓的日志系统 Logcat（2.3.2节）可以生成控制流图。在安卓运行过程中，想要动态获得运行时的函数间调用情况，可以通过 Logcat 输出详细信息去辅助定位。

程序插桩

程序插桩，最早是由 J.C.Huang 提出的 [41]，在程序中插入探针并保证被测试代码的逻辑完整性。本质上就是针对代码段进行信息采集，代码段的类型和长度都是可以自定义的函数调用。通过探针的执行并抛出程序运行的数据，通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等信息去揭示程序的内部行为和特征。

插桩方式是把函数的首尾加上 Logcat 日志语句，在参数部分整合函数的签名信息，具体模型如图3-9所示。

$$Parameter_{output1} = Start + Name_{class} + Name_{method} \tag{3-1}$$

$$Parameter_{output2} = End + Name_{class} + Name_{method} \tag{3-2}$$

公式 3-1 是输出参数的计算方式，其中 $Start/end$ 是函数开始/结束的标志，$Name_{class}$ 是函数所属类名的名称，$Name_{method}$ 是函数的签名，包括了函数名、是否有返回值、返回类型以及传入参数的数量以及类型。通过在日志中打印 $Parameter_{output}$ 信息，可以定位到具体的函数。

```
1 public void method(){
2
3       //concret code
4
5 }
```

(a) 插桩前

```
1 public void method(){
2       Log.i("Parameter_output1");
3       //concret code
4       Log.i("Parameter_output2");
5 }
```

(b) 插桩后

图 3-9: 插桩模型

安卓程序插桩是通过 soot 框架实现的。输入是安卓 APK 包，输出是插桩之后的 APK 包。使用 soot 插桩的算法流程如 Algorithm 1 所示。

Algorithm 1: soot 插桩流程

```
1  initSootSetings()
2  addTransformBody()
3  initInputApk()
4  foreach class in javaFile do
5      foreach method in classMethod do
6          filterUnuselessFile () foreach line in method do
7              if line is FirstLine then
8                  creatLogUnit()
9                  insertLogBeforLine()
10             if line is LastLine then
11                 creatLogUnit()
12                 insertLogAfterLine()
```

其中，初始化 soot 框架设置的时候需要设置安卓 jar 包路径，路径下需要包含所有的安卓 SDK 版本号。这是为了满足所有安卓 SDK 版本的 APK 包插桩。在 initInputApk() 步骤，会将 APK 导入内存中并进行反编译的处理。算法会遍历 APK 反编译的所有 Java 文件，在这个过程中需要过滤一些安卓 SDK 源码中的文件。过滤方法是通过匹配包名前缀，如果包名前缀匹配“android”单词，就会忽视这个文件的插桩。这样不仅可以提高插桩效率，也可以在之后的日志中减少冗余信息。算法 9、12 行会生成 Log 的句柄，区别在于其中的开始/结束标志不一样，所传的参数也不同，如公式 3-1、3-2 所示。

之后再利用 soot 框架提供的接口将 Log 句柄插入源码恰当的位置之中。通过 dex2jar 工具可以反编译 APK 包，查看是否插桩成功。这里需要注意的是，在打包 APK 时需要设置 debug 打包参数为不混淆代码，否则通过 soot 进行反编译的时候生成的包名均为混淆过的 a、b、c 等。虽然利用这些信息也可以找到签名函数，但是函数内部的变量大多经过混淆后会丢失大量语义信息。

插桩后输出的 APK 还需要签名才可以在移动设备上运行。签名可以防止应用被恶意篡改并保证用户数据的安全性。首先生成一个使用 RSA 算法的密钥

```
keytool -genkey -alias a.keystore -keyalg RSA -validity 20000
   -keystore a.keystore
jarsigner -verbose -keystore a.keystore -signedjar
   ${app名称_output.apk} ${app名称.apk} a.keystore
```

图 3-10: 签名 APK 命令行

库，再使用 dex2jar 工具（2.4.4节）生成其签名后的 APK。

生成代码对应的日志

当运行已经插桩的 APK 和测试脚本的时候，利用 adb 工具中的“adb logcat”命令行，我们可以看到在这个过程中系统产生的应用日志。由于日志内也包括了此应用在开发过程中的开发人员所输出的信息，我们需要过滤插桩的日志信息。过滤规则主要有两个：

（1）通过在 Log 句柄的参数中加入特殊的 tag，这个 tag 的命名只要与常规的安卓信息区分来即可。这里，我们定义 tag 为“info start – declaringClass:”和“info end – declaringClass:”。

（2）过滤掉第三方包产生的日志信息。在实际的安卓开发过程中，会使用很多著名的第三方库，如 OKHttp、Glide 等等。区分这个库的方法是通过包名。在公式 3-1、3-2 中，我们可以看到有一个参数是所属类信息，其中就包括了包名。例如在应用 CallBlocker 中的原生代码所属的包名是 com.eaglx.callblocker。

结合以上两条过滤规则，我们可以使用“grep”命令来过滤规则。通常应该分别过滤这些信息，但是由于规则（1）（2）所需要过滤的两部分字符串是相邻的，故可以合并这两条过滤规则为 grep declaringClass:com.eaglx.callblocker（以应用 CallBlocker 为例）。针对于 start 和 end 标志，我们在之后的实验中进行区分。

日志切片。在测试脚本运行过程中，产生的日志也是属于这个期间的。需要针对日志进行切片，方便映射到对应的测试语句上。Logcat 日志在输出的过程中会输出精确的时间戳信息，而每条日志的时间戳会有相联关系。我们定义两条日志是相联关系的条件为时间戳的间隔小于 100ms，反之如果大于 100ms，则非相联关系。利用相联关系进行切片，其规则如下：

（1）首先通过相联关系的概念，我们可以将日志分为 $Part_1$ 到 $Part_n$。在每个 $Part_i$ 之中，相邻的两条日志 Log_{ij}、$Log_{i(j+1)}$ 都满足相联关系。同时，在相邻的两个部分 < $Part_i$ >、< $Part_{i+1}$ > 中,< $Part_i$ > 的最后一条日志信息 Log_{im} 和

$< Part_{i+1} >$ 的第一条日志 $Log_{(i+1)m}$ 满足非相临关系。此外，$Part_i$ 的初始时间和结束时间分别由第一条日志和最后一条日志的时间戳来计算。

$$< Part_1 >< Part_2 >< Part_3 > ... < Part_n > \tag{3-3}$$

$$< Part_i >= Log_{i1}Log_{i2}...Log_{im} \tag{3-4}$$

$$TimeStart_{<Part_i>} = TimeStamp_{Log_{i1}} \tag{3-5}$$

$$TimeEnd_{<Part_i>} = TimeStamp_{Log_{im}} \tag{3-6}$$

（2）应用程序在启动的时候，也会执行大量的代码，产生相关的日志。这些日志针对不同的测试脚本来说，相差无几。对于本实验来说，是无效的日志信息，所以需要删除。Appium 项目测试脚本在运行主测试流程之前，会先尝试定位被测应用，这时候测试进程会调用 sleep 进行休眠，休眠时间为 8s。我们会将日志中第一部分 $Part_1$ 的开始时间信息记录下来，按照顺序向后搜索直到找到 $Part_i$ 满足公式 3-7。并将 $Part_i$ 之前的日志全部删除。这里公式 3.7 中使用的 7.8s，考虑到了一部分延迟。

$$TimeStart_{<Part_i>} - TimeStart_{<Part_1>} > 7.8s \tag{3-7}$$

(3) 测试脚本在开发过程中，每行测试语句之间会有 3s 的间隔。我们利用这个特性，将 *Part* 进行分区，如公式 3-8、3-9。其中 $Script_i$ 代表每个测试语句所属的日志切片，由几个相邻的 *Part* 组成。其中，$TimeStart_{<Part_j>}$ 是 $Script_{i+1}$ 的第一个 *Part* 的开始时间戳，满足 $TimeStart_{<Part_j>}$ 与 $TimeStart_{<Part_i>}$ 的间隔大于 2.8s（理由同上）。

$$Script_i = < Part_i >< Part_{i+1} >...< Part_{j-1} > \tag{3-8}$$

$$TimeStart_{<Part_i>} - TimeStart_{<Part_j>} > 2.8s \tag{3-9}$$

通过过滤规则和切片规则，我们得到处理后的日志文件，图3-11为应用ContactManager 的截取的一小部分日志信息。通过 start/end 标志，我们可以得到当前日志的控制流图。控制流图的生成可以通过树来表示。相同函数签名的 start 和 end 标志之间的日志信息为此函数的子调用，其他均为顺序调用。图3-12为部分日志所生成的控制流图。

```
01-12 11:05:07.987 27231 27231 I logger : info start --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:public void <init>()
01-12 11:05:07.987 27231 27231 I logger : info end --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:public void <init>()
01-12 11:05:07.990 27231 27231 I logger : info start --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:protected void onCreate(android.os.Bundle)
01-12 11:05:08.007 27231 27231 I logger : info end --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:protected void onCreate(android.os.Bundle)
01-12 11:05:08.010 27231 27231 I logger : info start --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:private void setListeners()
01-12 11:05:08.010 27231 27231 I logger : info start --
   declaringClass:com.matiboux.grifith.contactmanager.ContactInfo
   methodName:public static java.lang.String getFieldById()
01-12 11:05:08.010 27231 27231 I logger : info end --
   declaringClass:com.matiboux.grifith.contactmanager.ContactInfo
   methodName:public static java.lang.String getFieldById()
01-12 11:05:08.010 27231 27231 I logger : info end --
   declaringClass:com.matiboux.grifith.contactmanager.ListContacts
   methodName:private void setListeners()
```

图 3-11: Logcat 日志截取

生成 UI 回调函数

Log 日志定位函数。通过我们在代码插桩时，新建的 Log 句柄中的参数信息可以对源代码中的函数进行唯一确定。算法 2 是用日志定位函数的具体步骤。首先，需要使用日志信息字符串的切割来提取出类信息和方法签名。我们以图3-11中最后一行日志为例，其中类名是 com.matiboux.griffith.contactmanager.ListContacts。我们可以使用类名转换成全路径去打开这个类文件。需要注意的是，Java 中有些内部类的设计，所以有的类名带有 $ 符号，需要进行特殊的

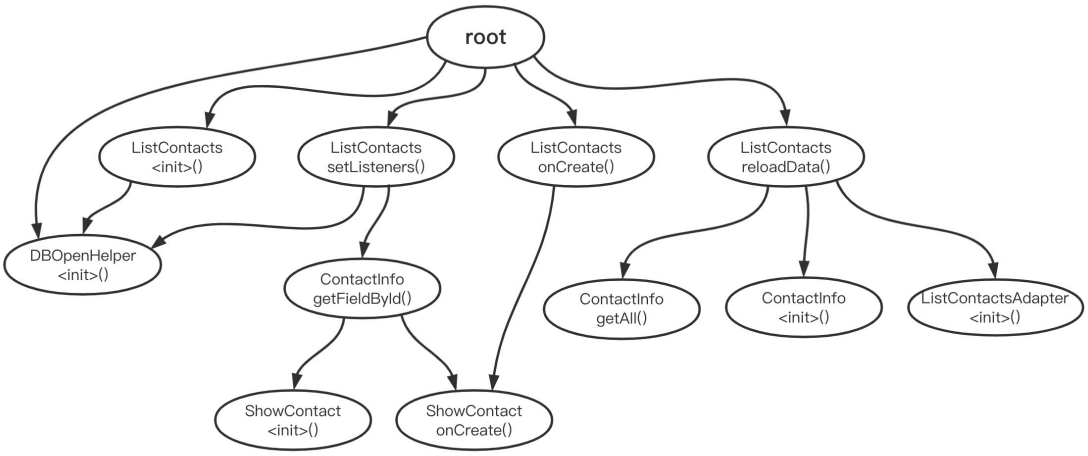

图 3-12: 控制流图

处理和判断。示例所对应的方法名是 setListeners()，返回值是 void，输入参数为空。通过这三个参数可以在 Java 文件中唯一确定一个函数。当前行如果匹配这三个参数后，即可以开始记录函数的内容。函数遍历结束的条件通过判断 leftParenthesis 参数来记录当前的 responseMethod 中所含的左右括号“{}”数量之差来确定。当 leftParenthesis 为 0 的时候，函数的标记即为结束。

合并函数。每行测试语句所对应的日志不止一条，所以需要对每行日志进行函数定位后进行合并。由于每个函数都对应了 start/end 两行日志，所以我们只对 Start 的日志进行统计。合并方法就是将函数进行拼接后，使用测试脚本名和当前测试语句在脚本中的索引对新函数进行命名。由此生成这行测试语句的 UI 回调函数。

3.2.3 生成代码语义信息

前文中提到的结合模版匹配和控制流图两种动静态分析的方法，生成 ID 类测试语句的 UI 回调函数。我们使用了 Alon 等人提出的 code2seq 模型 [30]，输入是 UI 回调函数，输出是语义信息。模型主要分为两步，第一是使用编码器构建代码片段的抽象语法树，第二是将其输入到编码器解码器模型之中得到目标序列。

抽象语法树（AST，Abstract Syntax Tree）代表了代码片段的独一无二的语法结构。语法树的叶子叫做终端，是用于代表开发人员在代码中定义的标识符和变量名称。例如，基础类型 int 等。没有叶子的节点被称作非终端，代表了一个编程语言中的逻辑结构，例如 if/else 语句被当作是 IfStmt。在抽象语法树

Algorithm 2: 定位函数

```
Input: log
Output: responseMethod
class_name = extractClass(log)
method_name, method_returnClass = extractMethod(log)
method_parametersList = getParametersList(log)
file = open(class_name)
foreach line in file do
    flag = false
    leftParenthesis = 0
    if (matches(line,class_name) and matches(line, method_returnClass)
    and matches(line,method_parametersList)) or flag then
        flag = true
        responseMethod += line
        leftParenthesis = calculateLeftParenthesis(responseMethod) -
          calculateRightParenthesis(responseMethod)
        if leftParenthesis == 0 then
            break
    continue
    flag = false
```

中，包括了所有终端之间可能的成对路径，并将其表示为终端与非终端的序列。在其中，我们会抽出 k 条路径作为核心的表示来代表输入的代码片段。每次训练中会多次迭代来选取 k 个路径，保证抽象语法树的偏差最小化。

Code 代表了输入的代码片段。假设终端序列为 $(t_1, t_2, ..., t_n)$, 那么终端序列一共有无限种可能。第 i 次迭代训练，我们都会从 *Code* 的抽象语法树中随机均匀的抽出 k 个终端序列 $\{SEQ_1^i, SEQ_2^i, ..., SEQ_k^i\}$，并成为 AST 路径。在编码器阶段，我们需要给 AST 路径生产一个向量代表 V_i。我们使用双向 LSTM 对路径进行编码，分别表示每个路径。

每个 AST 路径都是由节点和它们的子索引组成，这些子索引来自一个有限的词汇表中的节点和它们的子索引组成，最多有 364 个符号。我们用一个学习过的嵌入矩阵 E_{nodes} 来表示每个节点，然后用双向 LSTM 的最终状态对整个序列进行编码。AST 路径的第一个和最后一个节点是终端，代表 *Code* 中的

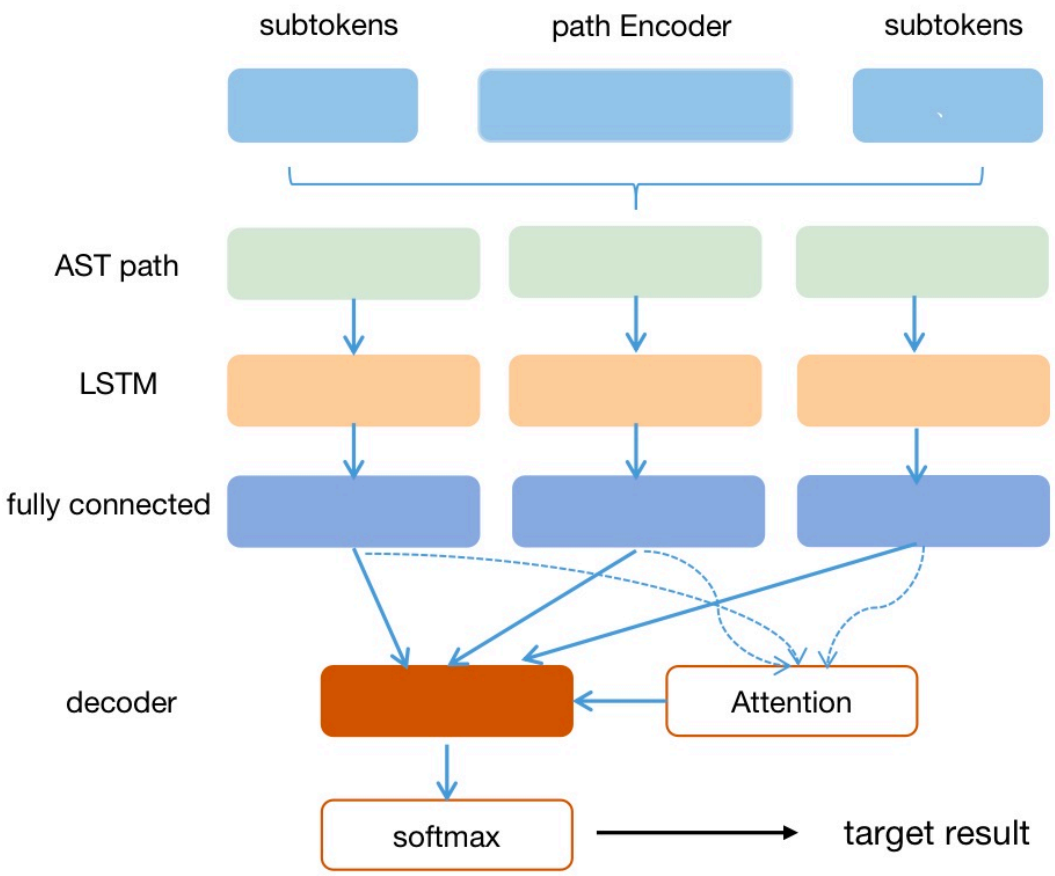

图 3-13: 生成 UI 回调函数意图模型

tokens[45][46]。其中，*tokens* 会被分成子 *tokens*，例如 contentValue 会被分成 content 和 value。例如，一个值为 ArrayList 的令牌将被分解为 Array 和 List。驼峰式编码可以提供了一个明确的每个 *tokens* 的分区。LSTM 也可以在每一步预测子 *tokens*。为了给解码器提供一个初始状态，对给定的所有 k 个路径进行平均。

编码器解码器模型 [42][43][44] 如图 3-13所示。在每个解码步骤中，下一个目标单词的概率与之前产生的 *token* 有关。attention 的模型是用于计算在解码阶段的每一个时间步长和上下文向量，使用 LSTM 进行计算。将时间步长和上下文向量输入多层感知器（MLP）[47]，最后用 softmax 预测下一个单词出现的概率。

在编码器解码器模型中，我们使用 sparse_softmax_cross_entropy 方法作为损失函数。优化方法是 Nesterov 动量法。我们对模型进行了 3,000 次历时训练，批次大小为 512。批量大小为 512。用于生成代码的语料库规模是 27,000。

AST 路径需要一个阈值来限制数量，我们将阈值设置为 9，这意味着一个 AST 路径包含的数量不超过 9 个节点。code2seq 模型是用 9500 个 Java 项目训练，其数据集足够大去适应我们的数据。我们将该模型封装成一个可调用的方法，返回结果字符串。该模型的加载过程是平行调用的。

3.3 XPath 分析

XPath (XML Path Language)，是一种在 XML 文档中搜素和定位元素的查询语言。超过一半的控件在测试脚本中是通过 XPath 定位的。使用 XPath 定位的控件有一部分是因为下文提到的 Adapter 方法导致的许多控件的 ID 一致，而剩下的是因为没有严格遵循编码标准。在通过 XPath 定位的部件中，大约 77% 的控件是用布局的层次结构定位的（公式 3-10），其余的是用 XML 的属性 content-desc，是控件的文本描述内容（公式 3-11）。其中，<widgetType> 是安卓的控件类名，有布局类 Layout、控件类 TextView 等，用于控件布局存在父控件和子控件的关系，所以通过其层次结构进行定位。

$$//<widgetType>[@content-desc=<text>] \tag{3-10}$$

$$/hierarchy/<widgetType>[0]/.../<widgetType>[i] \tag{3-11}$$

事实上，除了这五种开发方法去定义 UI 回调以外（3.2.1 节），还有一种常用的方法。由于现在很多安卓应用的页面都是动态拉取后端数据显示在移动端应用界面上，所以其动态变化导致无法定义所有的 UI 控件。这就需要用 Adapter 类，连接后端数据和前端显示的适配器接口，是数据和 UI 控件之间一个重要的纽带（图 3-15）。在 Adapter 内的所有控件都是动态绘制的，所以也无法使用静态模版方法去匹配得到 UI 响应函数。动态绘制 Adapter 内部控件时，由于数据格式是相同的，每个小部件的 ID 也是相同的（图 3-14）。通常在开发人员在写测试脚本时，也会通过 findElementsByXpath() 去获取控件。

(a) Fig1 (b) Fig2

图 3-14: 极简天气

3.3.1 获取 XPath 控件图像

图 3-16: XPath 截取图像

XPath 的生成是需要依靠 UiAutomator。在一个用户图形界面上（图 3-17），红色框是我们想要定位的控件，右上方蓝色框是该控件的层次结构（可视化的 XML 布局文件），下方的蓝色框是该控件的详细信息（所属类名、文本信息、是否可点击等等属性）。在运行 Appium 测试脚本的时候，我们可以保存其每一步的屏幕截图和 XML 布局文件。在 XML 布局文件中，我们可以通过 XPath 去定位控件，控件中包含了当前控件的横纵坐标。使用坐标在屏幕截图上进行二次截取，可以得到控件对应的图片（图 3-16）。

```
mAdapter = new CityAdapter(this, mList);
   mAdapter.setOnItemClickListener(new
   CityAdapter.OnItemClickListener() {
   @Override
   public void onItemClick(View v, ImageView favo, int position)
       {
       LocationEntity entity = mList.get(position);
       final FavoriteEntity favoriteEntity = new
          FavoriteEntity(entity.getId(),entity.getName(),
          entity.getPath());
       mViewModel.insertFavorite(favoriteEntity);
       entity.setFavorite(true);
       favo.setSelected(true);
   }
});
recyclerView.setLayoutManager(new LinearLayoutManager(this));
recyclerView.setAdapter(mAdapter);
```

图 3-15: Adapter 绑定 UI 回调

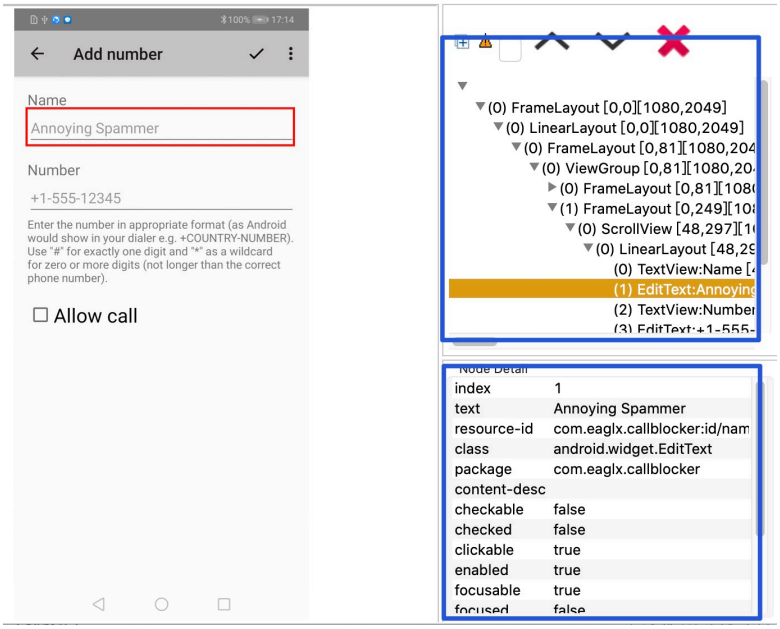

图 3-17: uiAutomator 截图

3.3.2 生成图像语义信息

截图的控件图像可以分为文字类和图片类两大类。针对文字类的图像，我们可以从 XML 布局文件中的“text”属性中提取文本信息。针对图片类，往往是没有直接的语义信息的。如图3-16所示，“+”代表了新建这个操作。为了

理解没有文本信息的控件图像，并其构建语义信息。

我们基于 [48] 构建了一个编码器解码器结构的深度学习模型。如图 3-18所示。编码器部分对图像输入和标题文本，分别是 Widget 特征提取器和序列特征处理器。小部件特征提取器被设计用来从输入的小部件图像中提取视觉特征。我们使用一个卷积神经网络（CNN）来提取图像特征。这个 CNN 模型包含卷积层和池化层。对于一个用于执行分类任务的普通 CNN 模型，其输出是图像的概率向量。然而，我们的目标是提取图像的特征，以便进一步处理。所以我们删除最后一层，直接输出一个 256 维的特征向量。对于文本输入，我们使用一个序列特征处理器来提取特征。最初的序列是一个起始标记。序列被送入一个嵌入编码器，将自然语言编码为向量。嵌入比单热编码的优点有两个：一是对于一个大的语料库来说，onehot 嵌入是相当稀疏的，这是一种计算资源的浪费；二是嵌入考虑了文本的语义。然后，嵌入的词被送入 LSTM 模型 [49]，将嵌入的测试解析为 256 维的向量。

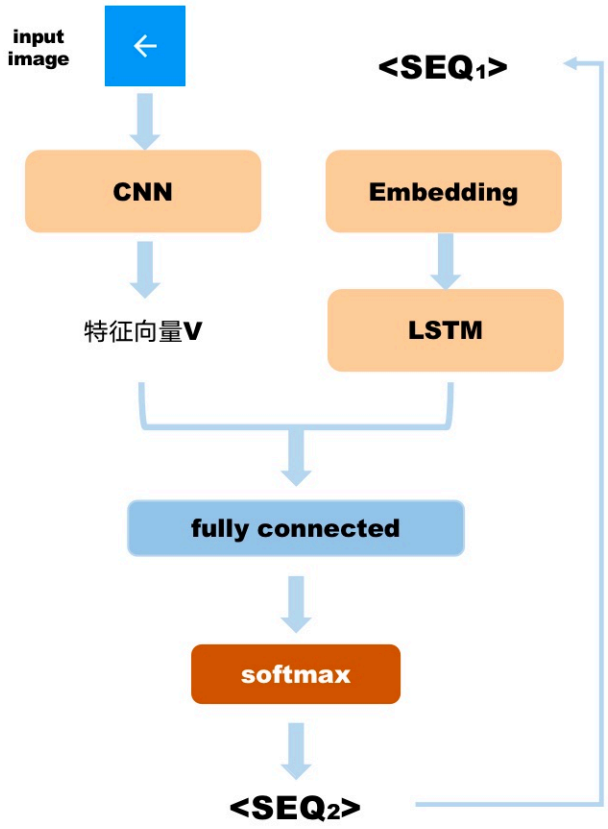

图 3-18: 编码器解码器模型

在获得小部件图像的两个特征向量和当前序列的两个特征向量后，我们将这些向量连接起来，得到一个 512 维的特征向量。该特征向量被送入全连接层以进一步解码该特征。中间结果向量被送入 softmax 层以预测小部件图像标题的下一个字。softmax 层的输出是一个输出概率向量，可以映射到语料库中的一个词。当前的结果将与之前的结果序列相连接，预测过程将重复进行。

该模型是在我们构建的大规模小部件图像数据集的基础上训练的。该数据集包含 2,000 个不同的 icon，并附有意图说明。该数据集是按照 7:2:1 的比例，分别为训练集、验证集和测试集。图像理解模型是由两个编码器和一个解码器组成。我们使用的小部件图像的编码器是 VGG-16 模型 [19]。输入的图像大小为（244，244，3）。该模型是由一系列卷积层、集合层和全连接层组成的。我们删掉了最后一层。

3.4 本章小结

本章主要是介绍了安卓移动应用的 Appium 测试脚本意图报告生成的整体框架。我们从 Appium 脚本绑定控件的方式出发，主要分为通过 ID 查找和通过 XPath 定位两种类型。针对 ID 类型的控件，我们使用了动静态结合的分析方法，具体为模版匹配和调用图分析法。其中，模版匹配是静态方法，主要是利用一些常见的编写 UI 响应函数的规则去匹配目标函数；调用图分析法是动态方法，我们使用 soot 框架对 APK 包进行插桩，通过 Logcat 输出其调用函数信息，定位目标函数。我们将函数输入对应模型之中得到其语义信息。针对 XPath 类控件，我们使用动态分析的方法，获取其截图和 XML 文件得到控件图像，分析 XML 文件得到文字类图像，使用编码器解码器模型分析图像的语义信息。最后，我们将两部分的结果进行整合，得到最终的意图报告。

第四章　实验与评估

4.1　实验目的

在第三章中，我们详细的介绍了基于动静态程序分析的安卓测试脚本意图生成的整体框架以及根据测试语句特点分为 XPath 和 ID 两种类型进行相应的处理。根据第三章中提出的方法框架，我们通过实验来验证其有效性。这里我们提出了两个问题，并在之后的结果分析阶段进行回答。

问题一：XPath 控件和 ID 控件的映射率如何？（关于映射率的定义我们在后文中给出。）

问题二：本实验得出意图报告结果的有效性如何？

4.2　数据准备

我们在 AndroZooOpen 上一共选取了 10 个开源的安卓移动应用，具体的应用名、github 地址和应用大小在表 4-1中给出。

表 4-1: App 数据集

App 名称	github-entry	项目大小
Baking-App	Abdallah-Abdelazim/Baking-App	5.1MB
CallBlocker	eaglx/CallBlocker	433KB
course	CSE-Projects/course-assistant-for-educators	10.3MB
iWeather	AllenWen/android-iWeather	12.6MB
VocableTrainer	0xpr03/VocableTrainer-Android	2.5MB
ContactManager	0matiboux/griffith-android-contactmanager	3.1MB
Kassenschnitt	Pika960/Kassenschnitt	250KB
NerverTooManyBooks	tfonteyn/NeverTooManyBooks	10.1MB
TEFAPTrackers	JDaniel29/TEFAPTracker	506KB
ProgressNote	0Kirby/ProgressNote	17.6MB

移动应用的选取需要符合以下三个条件：

（1）App 源代码需要全部由 Java 组成，不能有 Kotlin 代码。Kotlin 语言是一种基于 Java 开发的语言，但是其语法规则与 Java 差别较大，不适用于我们的模式匹配和代码语义生成。

（2）App 是非游戏类应用。游戏类应用的开发方式与普通应用有较大的区别。

（3）App 的源代码可编译或者是其 APK 包无混淆。我们在 APK 预处理阶段会对其进行插桩，如果是混淆后的代码，插桩之后生成的日志文件会丢失其语义信息，不利于之后语义生成阶段的处理。

针对每个安卓应用，我们编写了五个测试脚本，且每个测试脚本中的语句都具备有准确的意图信息。实验的测试脚本一共为 50 个，我们使用 Huawei Mate20 作为测试手机。

4.3 评估指标

控件映射率。由上文可知，我们将 XPath 类型的控件提取其控件图像、将 ID 类型的控件映射到其源代码的 UI 响应函数上，在这个过程中，我们假设图像/代码映射率为 u，当我们计算 XPath 控件的映射率时，n 为提取的控件图像，N 为样本总数。

$$u_{xpath} = n_{xpath}/N_{xpath} \tag{4-1}$$

同样的，如果计算 ID 的映射率时，N 也是样本总数，但是 n 为静态模版和动态分析方法两种映射到的样本的并集。

$$u_{id} = \frac{n_{dynamic} \cup n_{static}}{N_{id}} \tag{4-2}$$

同样的，我们计算生成的意图结果的映射率。这个是指在 XPath 分析或者 ID 分析的过程中，生成最终意图结果的样本占样本总量的比例。

BLEU（BiLingual Evaluation Understudy，双语评估）[53] 是一种用于评估从一种自然语言机器翻译到另一种语言的文本质量的算法。质量被认为是机器输出与人类输出之间的对应关系，BLEU 的核心思想是机器翻译越接近专业的人工翻译越好。分数是通过与一组高质量的参考译文进行比较来计算单个翻译片段（一般是句子）的分数。我们将这些评分进行平均处理，以达到对翻译的整体质量的估计。可理解性或语法正确性不在考虑范围之内。BLEU 的分数范

围在 [0,1] 之间。这个值表示参考文本和候选文本的相似程度，值越大代表句子间的相似程度越高。如果评估分数达到 1 分，表明候选文本与参考译文其中的一个样本相等。因为有更多的机会进行匹配，增加参考译文的样本数量可以增加 BLEU 评分。我们的意图提取大多数为英文，脚本中测试语句标记的注释为中文，我们将其翻译为多个相似的参考译文，然后使用 BLEU 评分指标来对此实验进行评估。

我们取测试语料库的几何平均值，然后将结果乘以一个指数级的简洁惩罚系数。我们首先计算出修改后的 $n-gram$ 精度的几何平均值，即 p_n。$n-gram$ 指一个语句里面连续的 n 个单词组成的片段。使用和为 1 的长度为 N 的 $n-grams$ 和正数权重 w_n。c 是候选翻译的长度，r 是有效参考语料库的长度。我们计算简洁性惩罚 BP（公式 4-2），BP 是对生成翻译文本过短的一个惩罚因子。$BLEU$ 的值由公式 4-4 计算得出。公式 4-3 和公式 4-4 结合并对 $BLEU$ 取对数后得到公式 4-5。在本实验中，我们取 $n=1$，2，3，4，也就是 BLEU1、BLEU2、BLEU3、BLEU4。

$$BP=\begin{cases} l, & if\, c>r, \\ e^{1-r/c}, & if\, c\le r. \end{cases} \tag{4-3}$$

$$\mathrm{BLEU}=\mathrm{BP}\cdot\exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{4-4}$$

$$\log \mathrm{BLEU}=\min\left(1-\frac{r}{c},0\right)+\sum_{n=1}^{N} w_n \log p_n \tag{4-5}$$

CIDEr[54]（Consensus-Based Image Description Evaluation，基于共识的图像描述评价）主要采用 TF-IDF（Term Frequency Inverse Document 频率）来计算不同参考句子中 n-grams 的权重。基础共识是，n-grams 出现得越少，它可能包含的信息就越多。我们使用 TF-IDF 对每个 n-gram 进行加权处理 [36]。一个 n-gram 的 $_k$ 在参考句子 s_{ij} 中出现的次数用 $h^k(s_{ij})$ 表示，对于候选句子 c_i，用 $h_k(c_i)$ 表示。公式 4-6 计算每个 n-gram 的 $_k$ 的 TF-IDF 权重 $g_k(s_{ij})$。

$$g_k\left(s_{ij}\right)=\frac{h_k\left(s_{ij}\right)}{\sum_{\omega_l\in\Omega} h_l\left(s_{ij}\right)}\log\left(\frac{|I|}{\sum_{I_p\in I}\min\left(1,\sum_q h_k\left(s_{pq}\right)\right)}\right) \tag{4-6}$$

其中 Ω 是所有 n-grams 的词汇，I 是数据集中所有图像的集合。第一项衡量每个 ω_k 的 TF，第二项使用 IDF 来衡量 ω_k 的稀有性。也就是说，IDF 通过以

下方式提供了一个词语突出性的衡量标准并扣除那些可能在视觉上不太重要的流行词。IDF 的计算方法是使用数据集中的图像数量 $|I|$ 除以 $_k$ 出现在任何参考句子中的图像数量的对数。

长度为 n 的 n-grams 的 CIDEr 分数的计算方法是候选句子和参考句子之间的平均余弦相似度，使用了精度和召回率。

$$\mathrm{CIDEr}_n(c_i, S_i) = \frac{1}{m}\sum_j \frac{\boldsymbol{g^n}(c_i)\cdot \boldsymbol{g^n}\left(s_{ij}\right)}{\|\boldsymbol{g^n}(c_i)\|\left\|\boldsymbol{g^n}\left(s_{ij}\right)\right\|} \tag{4-7}$$

$g^n(c_i)$ 是由 $g_k(c_i)$ 组成的向量，$\|g^n(c_i)\|$ 是向量 $g^n(c_i)$ 的值，$g^n(s_i j)$ 同理。

ROUGE[55] 是 Recall Oriented Understudy of Gisting Evaluation 的缩写。它计算了候选语句的基于 n-gram 的召回率与参考语句的关系。它是一个流行的总结性评价的常用指标。与 BLEU 类似，ROUGE 的版本可以通过改变 n-gram 计数来计算。ROUGE 的另外两个版本是 ROUGE-S 和 ROUGE-L。这两个版本计算了一个 F-measure，有一个召回偏差，使用跳过的双关语和最长的的共同子序列来计算每个参考句子之间的 F 值。跳过的词是句子中的所有对在一个句子中的所有有序的词，非连续地取样。考虑到这些分数，他们返回整个参考语句中的最大分数作为判断的依据。本实验选取 ROUGE-L 做为评估指标，其为基于两个文本单元的最长公共序列。其计算方式如下

$$R_{LCS} = \frac{LCS(X,Y)}{m} \tag{4-8}$$

$$P_{LCS} = \frac{LCS(X,Y)}{m} \tag{4-9}$$

$$F_{LCS} = \frac{\left(1+\beta^2\right)R_{LCS}P_{LCS}}{R_{LCS}+\beta^2 P_{LCS}} \tag{4-10}$$

其中，其中 X 为参考语句，长度为 m；Y 为候选语句，长度为 n。β 为精确率和召回率的比值。

METEOR[56]（Metric for Evaluation of Translation with Explicit ORdering）是指用显式排序法评估翻译的度量。与 ROUGE-L 和 ROUGE-S 类似，它也计算基于匹配的 F-measure，并返回一组参考文献中的最大分数作为它的质量的判断。然而，它以一种更复杂的方式来解决词级对应关系，使用精确匹配、词干化和语义相似性。它优化了匹配，使之达到最小化的程度。最大限度地减少意

味着，只要有可能匹配应该是连续的。

METEOR 扩展了 BLEU 有关“共现”的概念，提出了三个统计共现次数的模块：一是“绝对”模块（exact module），即统计待测译文与参考译文中绝对一致单词的共现次数；二是“波特词干”模块（porter stem module），即基于波特词干算法计算待测译文与参考译文中词干相同的词语“变体”的共现次数，如 happy 和 happiness 将在此模块中被认定为共现词；三是“WN 同义词”模块（WN synonymy module），即基于 WordNet 词典匹配候选语句与目标语句中的同义词，计入共现次数，如 sunlight 与 sunshine。

首先需要计算 F-measure，其根据精度 P 和召回率 R 得出。值得注意的是，它设置了有利于召回率而非精度的参数，召回率比重是精度的 9 倍。

$$\text{Fmean} = \frac{10PR}{R+9P} \tag{4-11}$$

在 METEOR 中，我们需要计算惩罚因子 Penalty。其作用是针对单词一致但是乱序的候选语句，需要给出严重的惩罚系数。chunks 的数目越少意味着每个 chunk 的平均长度越长，也就是说候选语句和参考语句的语序越一致。unigrams_matched 表示匹配上的 unigram 个数。

$$\text{Penalty} = \gamma\left(\frac{\text{chunks}}{\text{unigrams\_matched}}\right)^{\theta} \tag{4-12}$$

$$\text{Meteor} = (1 - \text{Penalty}) \cdot F \tag{4-13}$$

最后，METEOR 计算候选语句和参考语句的准确率和召回率的调和平均。

4.4 结果分析

4.4.1 问题一的结果分析

图4-1是生成测试脚本的意图报告方法的映射率。该图主要包括两个部分。左图为图像/代码映射率，右图为生成意图结果映射率。在左图中，XPath 代表了在 XPath 分析中可以正常提取图像的比例，ID_static 代表了在 ID 分析过

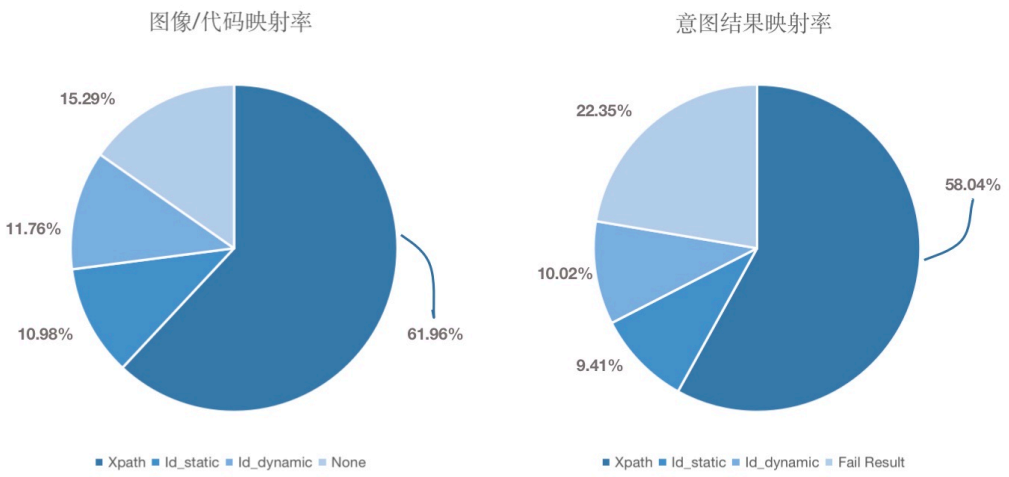

图 4-1: 映射率

程中通过静态模版方法提取到的 UI 回调函数比例，ID_dynamic 代表了在 ID 分析中通过动态调用图法提取到的 UI 回调函数比例，None 代表了 XPath 分析或者 ID 分析均没有产生图像/代码结果的控件比例。左图中反应了在第一阶段的映射中有 15.29% 的控件没有生成其对应的图像/代码；XPath 的总映射率高于 ID 的总映射率，是由于 XPath 控件数量大于 ID 控件数量，XPath 分析和 ID 分析在各自控件类型的映射率为 96.34% 和 69.44%。由此可见，XPath 分析在获取图像和 XML 文件时的映射成功率更高。XPath 分析中失败的映射是由于在运行自动化测试脚本的过程中，连接不稳定性导致图像、XML 文件等信息没有存储下来。在 ID_static 分析中，无法生成 UI 回调函数的原因即为不满足在第三章中提到的五个模版。在 ID_dynamic 分析中，没有生成 UI 回调函数主要有两个原因，一是因为有一部分的控件在用户交互之后，应用并没有针对这一部分控件作出任何响应，而是在点击提交之后才会提取这一部分的数据，比如说一部分的输入控件；二是因为其输出的日志信息无法定位到源代码中的响应函数，比如说我们使用了数据库的注解去实现真正的数据库处理类，这部分代码会在编译之后才会生成，所以我们无法在源代码包中定位 UI 回调函数。

我们通过将左边图中 XPath 分析或者 ID 分析的图像/代码输入到对应模型中，可以生成意图结果，从而得到右图中意图结果的映射率。XPath 代表了在 XML 文件提取文字信息或者在图像生成模型中得到意图结果的比例，ID_static 代表了通过静态模版方法生成的代码意图比例，ID_dynamic 代表了通过动态调用图法生成的代码意图比例，Fail Result 代表了没有产生最终结果

的控件比例。我们可以看出，在经过了第二阶段的分析后，XPath、ID_static、ID_dynamic 三个数值都有所下降，这是因为在模型分析的时候，会有一部分图像/代码无法生成最终的意图结果。ID_static、ID_dynamic 这两部分在生成 UI 回调函数代码的时候所使用的方法不同，但是在代码语义生成阶段的模型是相同的。其中，图像语义生成模型、代码语义生成模型的失败比例分别为 6.33%、17.24%，说明图像语义生成模型的成功率更高。

4.4.2 问题二的结果分析

表 4-2: 评估指标结果

	BLEU1	BLEU2	BLEU3	BLEU4	ROUGE	CIDEr	METEOR
XPath	0.77625	0.32698	0.15508	0.10805	0.30067	0.37013	0.15005
ID_static	0.07174	0.04	0.01422	0	0.20023	0.35278	0
ID_dynamic	0.17347	0.07	0.01929	0	0.41528	0.79154	0
code2seq	0.02006	0.003162	0	0	0.04750	0.00458	0.01431
TestIntent	0.34864	0.15357	0.06561	0.03045	0.27958	0.50520	0.02222
TestIntent_step	0.59469	0.23943	0.11355	0.07912	0.29983	0.37675	0.10778

表 4-2为该方法中各个部分的评估指标数据。XPath 代表了 XPath 分析部分的评估指标结果；ID_static 为 ID 分析部分中静态模版方法；ID_dynamic 为 ID 分析部分中动态调用图法；code2seq 是直接使用论文 [30] 中的模型生成意图结果的评估数据；TestIntent 为 XPath 和 ID 分析后将每一句测试语句的意图结果进行整合之后得到的每个测试脚本的意图报告的结果进行分析得到的评估数据；TestIntent_step 是指在 XPath 和 ID 分析之后针对每一条测试语句意图结果进行分析后得到的评估数据。

总体来说，我们的模型在各个部分的评估数据都高于 code2seq，这说明了我们的模型的有效性。这是因为单纯在测试脚本中，不能反应出每个测试语句复杂的逻辑，脚本只包含用户操作由 ID 和 XPath 定位的应用控件。因此。code2seq 模型不能从测试脚本中提取太多信息。我们的模型将每个测试语句的用户操作映射到图像文字信息或者 UI 响应函数代码，提供了更多有效的信息去生成意图结果，所以在各个评估指标上均优于原始的 code2seq 模型。

纵向来看，在以上六个部分中，BLEUn（n=1，2，3，4）指数随着 n 的增大而减小，这主要是因为我们的意图结果大多为短语。其中，只要小部分的

结果单词大于 3 个，占比小于 10%，这也导致了 BLEU4 在 ID 分析部分的结果为 0。METEOR 指标相对其他指标的评分较低，是因为该指标对于意图结果乱序的情况会有较大的惩罚因子。我们的代码语义生成模型生成的意图结果不考虑单词之间的顺序关系，故在 ID 分析部分评分为 0。ROUGE 指标中，我们考虑将最长公共子序列（LCS），故评分较于 BLEU 和 CIDEr。CIDEr 是将词频逆文档频率（Term Frequency Inverse Document，TF-IDF）应用在句子的相似性判断上，TF-IDF 针对于不经常出现的单词，会有较高的分数。我们的 ID_dynamic 模型在 CIDEr 的评分高达 0.79154，说明动态调用图法挖掘出了频率出现低且较为重要的信息。

横向来看，我们将所有的指标平均后，XPath 部分的指标优于 ID 部分，这说明 XPath 分析的模型较 ID 分析更优，原因可能是从图像或者 XML 中提取的意图信息比 UI 响应函数代码中提取的意图信息更为准确。ID_dynamic 部分的指标数据大部分优于 ID_static 部分，这两个部分所使用的代码意图生成模型是相同的，这说明动态调用图法中定位的 UI 回调函数代码的准确性高于静态模版法。除了 CIDEr 指标，TestIntent 部分的评分都高于 TestIntent_step，这说明我们整合每一步骤的意图结果的方法较差，降低了最终结果。

4.5 本章小结

本章主要是对测试脚本意图生成的技术的有效性进行评估。实验选取了 10 个开源的安卓移动测试应用，应用选取满足我们在文中提出的三个条件。每个应用我们编写了 5 个 Appium 的测试脚本，并且每行测试语句中都具有准确的意图信息注释。该实验中，我们提出了两个问题，问题一是 XPath 分析和 ID 分析中控件的映射率如何，问题二实验结果的有效性如何。针对问题一，我们提出的方法在图像/代码映射率为 84.71%，在生成意图结果的映射率为 77.65%。针对问题二，我们选取了 BLEU 指数、CIDEr 指数、METEOR 指数和 ROUGE-L 指数来评估实验的有效性。按照方法的不同，我们分为 XPath、ID_static 和 ID_dynamic 三部分。按照意图结果粒度划分，我们分为测试脚本意图和测试语句意图两部分。除此之外，我们还计算了 code2seq 模型的指标数据。结果表明，我们的方法在各个评估指标上均优于 code2seq，说明了该方法在测试脚本之外提取了更多有效信息（文本、图像、代码等）。

第五章　系统需求分析与概要分析

5.1　需求分析

5.1.1　功能需求分析

本系统核心功能是生成安卓应用的 Appium 测试脚本的意图报告。基于此，从用户和系统交互的角度分析了多项功能需求。同时，对这些功能需求的具体内容进行描述并针对优先级进行排序。其中，R1 和 R7 是与用户直接交互的需求，剩下的需求都是为生成意图的测试报告主流程所服务。

表 5-1: 功能需求

需求编号	需求名称	需求描述	需求优先级
R1	上传文件	用户上传安卓 APK 包、安卓测试脚本、安卓应用源代码	高
R2	Apk 预处理	将用户上传的 APK 利用 soot 框架对其进行代码插桩的预处理并对其签名处理	高
R3	自动化运行测试脚本	运行用户上传的测试脚本和预处理之后的安卓 APK，并存储相关 XML 文件、截图信息和日志信息	高
R4	测试脚本分析	针对用户上传的测试脚本进行 XPath 或者 ID 的分析处理	高
R5	XPath 控件分析	将测试脚本中的 XPath 类控件的文字或者图像提取出来并生成其意图信息	高
R6	ID 控件分析	使用动静态分析来提取测试脚本中的 ID 类控件的 UI 响应函数，并输入模型中生成其测试意图信息	高
R7	生成意图报告	将 XPath 和 ID 的信息进行整合，生成最终的意图报告	高
R8	下载意图报告	将测试脚本的意图报告生成文件，保存到本地	中

R1 是用户针对文件进行上传操作，为该系统交互的第一步；然后是 R2 针对 APK 进行预处理操作，此步骤检测了 APK 的合规性、并对 APK 进行插桩和签名操作；R3 自动化运行测试脚本，并保存相关信息；R4 是测试脚本分析，将测试脚本处理成 XPath 或者 ID 形式，并调用 R6 或者 R7 完成意图生成步

骤；R8 是在整个流程处理完成之后，用户下载意图报告或者是全部文件的 zip 包。

5.1.2 非功能需求分析

非功能需要需要保证该系统的可靠性。在生成测试脚本意图的同时，也需要保证在长时间内的稳定运行，以及发生故障之后的应急处理。表 5-2列举了本系统所需的非功能需求。

表 5-2: 非功能需求

需求编号	需求名称	需求描述	需求优先级
R9	易用性	用户操作简单	中
R10	可维护性	系统在较长时间内可以保证稳定运行，在发生故障之后，可以保证在 20s 内重启之后恢复服务	高
R11	可维护性	当出现新增的功能需求时，尽可能少的变更之前的代码；出现系统漏洞时，可以快速定位及时修复	中
R12	性能	在用户提交输入的文件之后，需要保证用户在可接受范围内收到结果	中

5.2 系统用例图以及用例描述

本系统的核心功能是生成安卓应用的 Appium 测试脚本对应的意图报告，方便开发人员对该测试脚本的管理和重用。图 5-1是该系统相关的用例图。其中，唯一的角色是用户，其对应的用例一共有三个，分别为上传文件、提交生成测试脚本对应的意图报告和下载意图报告文件。

表 5-3: 用例图

用例编号	用例名称	功能需求编号
UC1	上传文件	R1
UC2	下载文件	R7、R8
UC3	生成意图报告结果	R2、R3、R4、R5、R6、R7

表 5-3列举了用户对应的八个用例的用例编号、用例名称以及相关的功能需求编号。其中，上传文件的用例名称只对应需求 R1，和其他需求无相关联

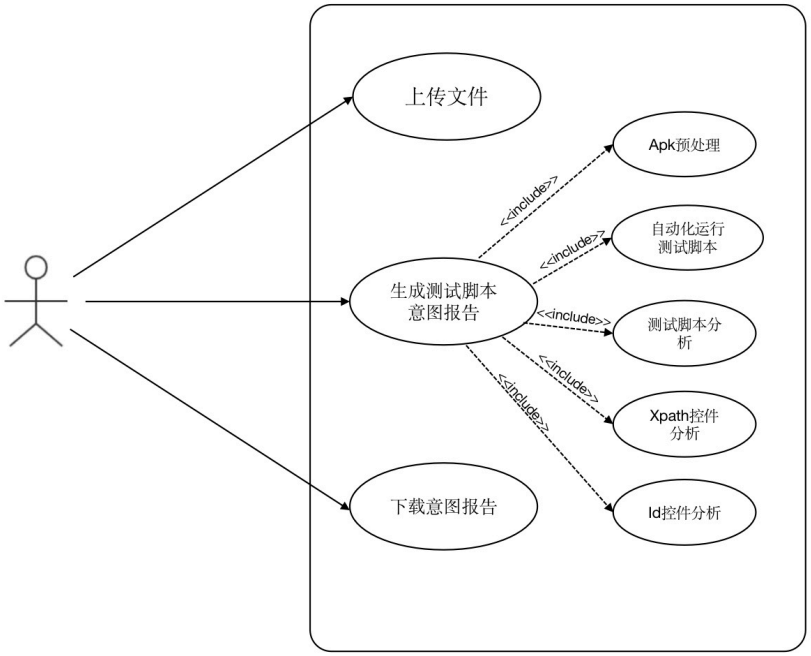

图 5-1: 用例图

性。UC8 生成意图报告，需要用户在提交生成按钮之前，成功上传所需的三个文件，并且完成其前置的所有步骤，所以相关的功能需求编号为 R6、R7。

用例 UC2 下载在测试脚本的意图报告相关联功能需求是 R7 和 R8，因为测试脚本的意图报告需要成功生成，才可以提供文件下载的功能。UC3 是测试脚本的意图报告生成的主流程，与其相关联的需求为 R2-R7。

下面，本文将通过用例表格对三个用例进行具体的描述。

第一个用例是用户上传文件，表 5-4是针对上传文件的用例描述。我们可以看到，上传文件是用户使用本系统的第一步，所以没有前置条件。用户上传的文件一共有三个，分别为安卓 APK 包、安卓应用源代码以及对应的 Appium 测试脚本。这三个文件缺一不可。规格说明中的流程是按顺序上传文件，但用户可以自由选择上传顺序。当文件在上传过程中，由于网络等原因出现上传失败时，系统应该出现提示并且引导用户再次上传。这三个文件会用于之后的测试脚本报告的生成。

表 5-5介绍了第二个 UC2 下载文件的用例描述。该用例主要是用户针对上传的三个文件和生成的测试脚本的意图报告等进行保存下载的操作。用户可以自定义的选择想下载的文件并提交保存请求。如果是测试脚本的意图报告，系统会生成报告对应的文本文件；如果是上传的三个文件，即直接返回服务器储

表 5-4: 用户上传文件用例描述

描述项	说明
用例编号	UC1
用例名称	用户上传文件
参与者	用户
用例描述	用户上传三个文件，其中包括有安卓 APK 包、安卓源代码、对应的 Appium 测试脚本
前置条件	无
正常流程	1. 用户上传安卓 apk 包 2. 系统提示上传成功 3. 用户上传源代码 4. 系统提示上传成功 5. 用户上传安卓的 Appium 测试脚本 6. 系统提示上传成功
后置条件	生成对应测试脚本的意图结果以及每一行测试语句对应的语义信息
异常流程	2. 用户上传安卓 apk 包失败 a) 系统提示失败，引导用户再次上传 4. 用户上传安卓源代码 zip 包失败 a) 系统提示失败，引导用户再次上传 6. 用户上传测试脚本失败 a) 系统提示失败，引导用户再次上传

存的文件。最后将文件保存到用户本地。该用例的前置条件是需要用例 2 的正常生成，如果没有正常生成意图报告的话，无法进行保存，需要系统针对此问题对用户做出提示。

表 5-5: 用户下载文件用例描述

描述项	说明
用例编号	UC3
用例名称	用户下载意图报告文件
参与者	用户
用例描述	用户保存生成的意图报告文件
前置条件	测试脚本的报告生成成功
正常流程	1. 用户选择保存文件 2. 系统生成文件 3. 文件保存到本地
后置条件	无
异常流程	1. 用户选择保存文件不存在 a) 系统提示用户先生成意图报告 2. 系统生成文件失败 a) 系统再次尝试文件生成

表 5-6: 生成意图报告用例描述

描述项	说明
用例编号	UC2
用例名称	提交并生成测试意图结果
参与者	用户
用例描述	用户提交后，生成对应 Appium 脚本的测试意图结果
前置条件	用户成功上传所需的三个文件，并且 APK 插桩成功
正常流程	1. 用户提交生成命令 2. 对 APK 进行插桩 3. 对插桩之后的 APK 进行签名 4. 对 APK 进行检验 5. 系统查看 Appium Server 端 6. 系统检测 adb devices 连接情况 7. 系统运行 Appium 测试脚本 8. 存储相关的媒体信息，包括屏幕截图、XML 布局文件和日志文件 9. 对测试脚本进行分析 9.1 针对 XPath 类型控件进行处理 9.2 对 ID 类型控件进行处理 10. 整合意图报告并写入文件 11. 返回前端数据
后置条件	无
异常流程	2. 启动 Appium Server 端失败 a) 将启动时的错误信息返回 3. 没有可用设备 a) 提示用户等待 4. 系统运行 Appium 测试脚本错误 a) 将错误日志信息返回给用户，提示用户重新上传测试脚本文件

表 5-6介绍了第三个 UC3 生成测试脚本的意图报告的用例描述。根据上文可知，该用例的前置条件是用户成功上传所需的三个文件并且 APK 的预处理结束。所以，当用户提交生成报告的请求时，需要提前检查三个文件的可见性。如果三个文件内有文件缺失，需要系统针对这个问题提示用户，从而保证前置条件。用户在提交生成请求之后，系统需要进行三大步骤，APK 预处理、启动 Appium Server 端和测试脚本分析处理。APK 预处理步骤中，系统需要针对用户上传的 APK 包使用 soot 框架对其进行插桩处理，使其能够输出一个控件的调用关系图；然后需要对新生成的 APK 包进行签名，以便后续能够正常在测试设备上运行。在这个过程中，还会检查 APK 的混淆情况。Appium Server 端是为了在测试机器上运行 Appium 的测试脚本。之后，系统需要检测安卓测试机器的连接情况，只有测试机正常开启 debug 模式之后，才可以运行安卓的 Appium 测试脚本。做好一系列准备工作之后，系统尝试运行用户上传的 Appium 测试脚本，需要注意的是，如果测试脚本有 bug 或者出现测试脚本

的 Appium jar 包与系统本身设定的 Appium jar 包版本不符的情况，会导致测试脚本运行失败。这时，需要将在运行测试脚本错误时的 Java 项目日志信息返回给用户，提示用户更改之后再上传并重新开始此流程。在处理完自动化测试这一流程之后，系统会保存并处理相关的截屏、日志等数据。最后是针对测试脚本的分析与处理，根据上文可知，我们将测试脚本语句分为了 XPath 和 ID 两大类型分别进行分析。针对 XPath 类型的控件，我们对其图像进行分析；ID 类控件，我们对其代码进行分析。将两部分的结果进行整合后，我们得到最终的意图报告结果返回给用户。

5.3 系统概要设计

5.3.1 系统架构设计

基于动静态的安卓测试脚本意图分析系统使用了前后端分离的架构，如图 5-2所示，前端使用了 Vue 框架实现，后端使用了 Django 框架实现。前端通过调用后端 Django 中 API 接口来获取数据。在服务端，我们将模块主要划分为四个，APK 预处理模块、自动化运行模块、意图报告生成模块和文件模块。底层服务的支撑主要有三个，测试设备提供了安卓 Appium 脚本运行的硬件支撑；机器学习模型主要用于生成代码和图像的语义信息；数据部分主要包括数据库和文件存储。

Vue 是一个用于构建用户界面的 JavaScript 框架。它的核心部分主要集中在视图层，非常容易理解。在本系统中，我们要使用的 Vue 版本是 2.0。Vue.js 最大的优势之一就是它的体积小。Vue.js 促进了双向通信，也加快 HTML 块的处理速度。这个功能也被称为双向数据绑定，这意味着无论在 UI 中做什么改变，都会传递给数据，而在数据中做的改变也会反映在 UI 中。

Django 框架是一个轻量级的 Web 后端框架，编程语言是 python。由于我们实验中的许多环节使用 python 实现，所以选取了当下较为流行的 Django 框架。服务端中的用户接口层用于提供 API 将数据传输给前端页面并展示给用户。系统共划分为 4 个模块，其中 APK 预处理模块、自动化运行模块和测试脚本分析模块为生成意图生成报告的主流程模块。APK 预处理模块主要负责对用户上传的 APK 进行动态插桩和 APK 签名，是服务于测试脚本分析模块中 ID 类控件的调用函数日志。同时，在 APK 预处理模块中对 APK 进行校验，校验其是否为未混淆的 APK 包。自动化运行模块主要是将 Appium 服务和运行测

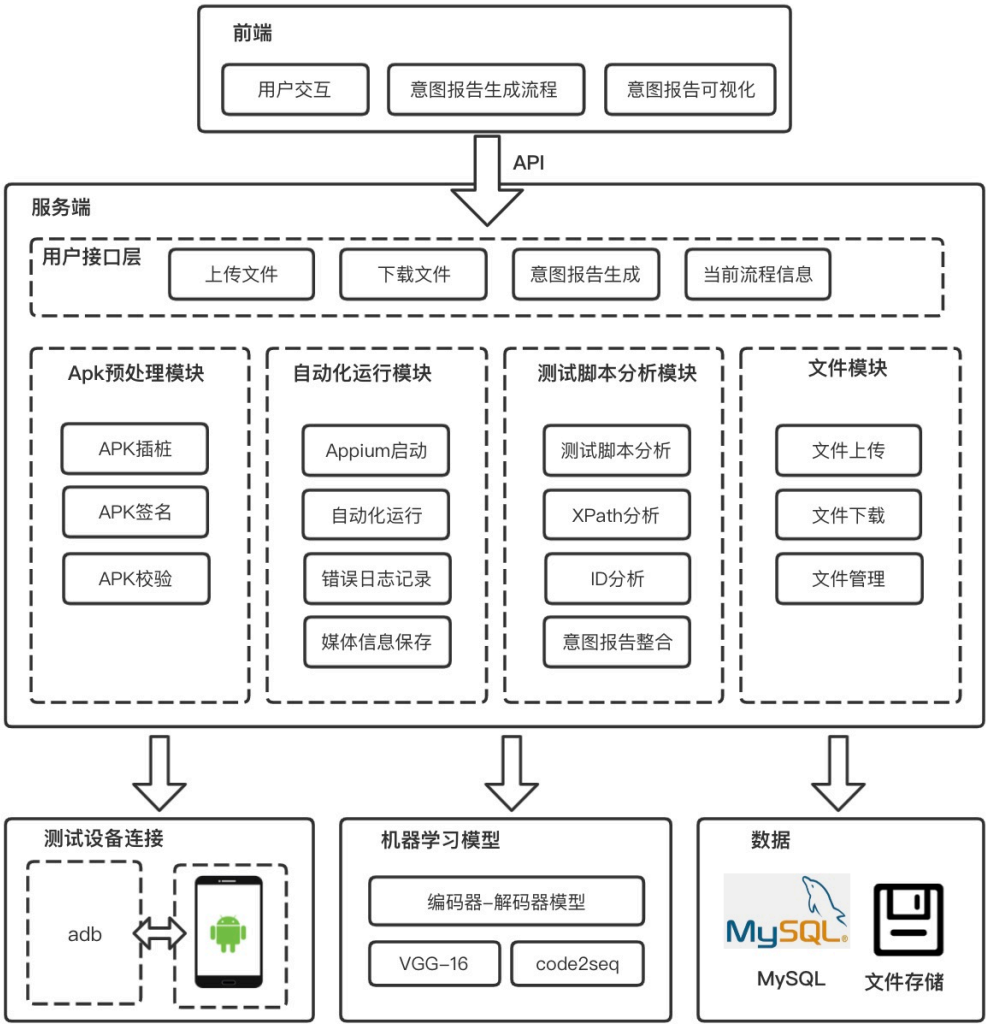

图 5-2: 系统架构

试脚本的逻辑封装起来，同时需要将测试脚本分析模块中所需要的媒体文件、日志文件保存下来。测试脚本分析模块的核心就是生成最终的测试脚本意图报告，主要分为四个子模块，测试脚本分析、XPath 分析、ID 分析和意图报告整合。最后一个模块是文件模块，其主要是负责文件的管理和用户的上传以及下载等。

5.3.2 系统视图模型

本系统主要使用了“4+1”视图模型，一种使用多个并发视图的软件架构描述模型。从系统的多个角度出发，将这些视图结合起来可以形成完整的系统

设计。场景视图在上文中的用例分析中已经有详细介绍，下面将分别从逻辑视图、开发视图、进程视图以及物理视图对基于动静态分析的安卓测试脚本意图生成系统进行多角度的设计与说明。

逻辑视图

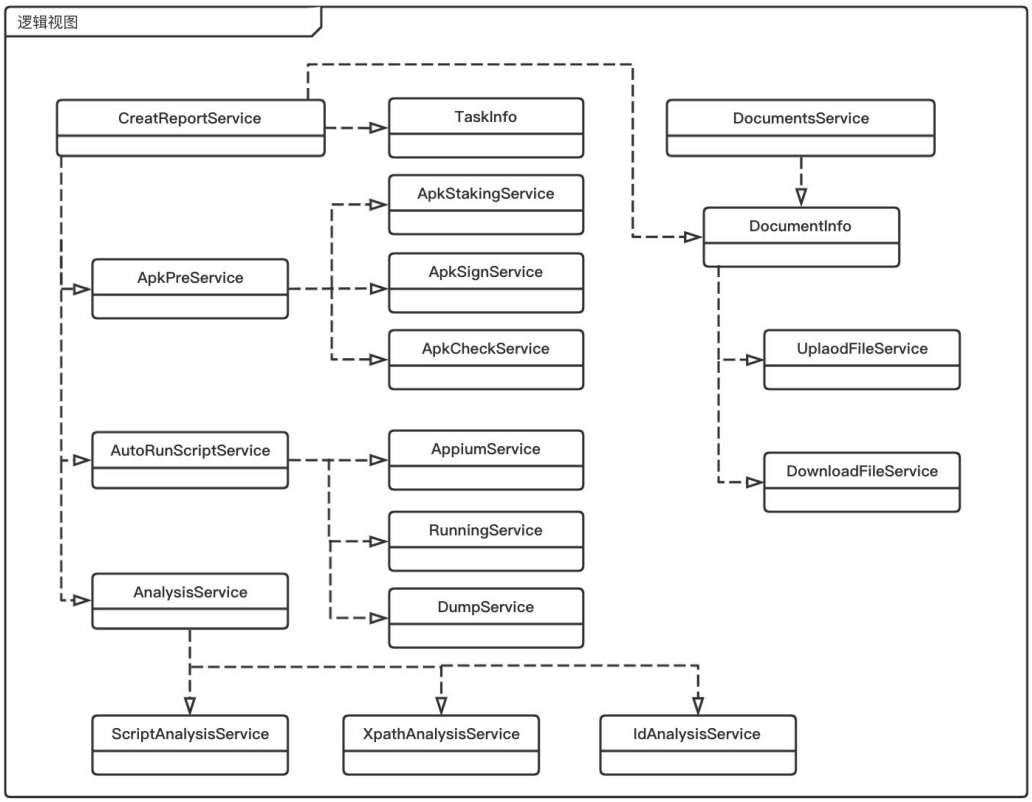

图 5-3: 逻辑视图

图 5-3是系统的逻辑视图。其中，CreateReportService 是本系统的主服务，生成测试脚本的意图报告。这个核心服务需要两部分数据进行支撑，TaskInfo 用于创建一个生成任务时存储相关信息，DocumentInfo 中存储了主服务在执行过程中需要的文件信息，包括了 APK 包文件、源代码文件和 Appium 测试脚本文件。CreateReportService 中的子服务有三个，分别是 ApkPreService、AutoRunScriptService 和 AnalysisService。ApkPreService 主要是提供了 Apk 预处理的服务，其中 ApkStackingService 利用 soot 框架的代码插桩将函数的签名转换成日志输出，为之后的流程提供支撑。ApkSignService 是为 APK 进行日志签名服务，只有被签名的 APK 才可以安装在测试设备之中；ApkCheck 是为了检查 APK 是否混淆。AutoRunScriptService 包括了三个子服务，AppiumService 用于启动 Appium 进程服务，并且检查 adb devices 的连接情况。RunningService 是

将 Appium 测试脚本植入测试脚本的 Java jar 包之中运行测试。DumpService 服务记录了在运行测试过程之中利用 Appium Driver 提供的接口进行截屏信息、XML 文件和日志信息的保存。AnalysisService 是生成意图报告的最后一个服务，先是进行测试脚本的分析，之后对于不同类型的语句进行 XPath 分析和 ID 分析，生成最终的意图报告结果。DocumentService 为主服务提供文件管理的功能，也于用户的文件上传与下载有关。

进程视图

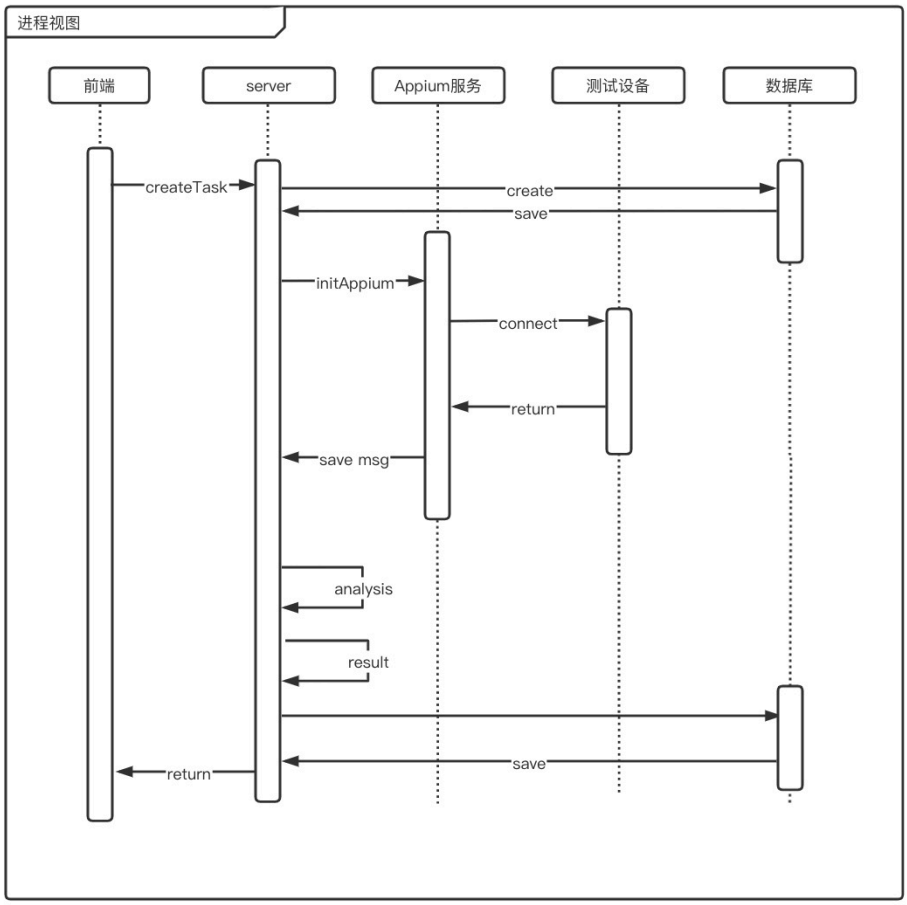

图 5-4: 进程视图

图 5-4介绍了本系统的进程视图。进程视图主要是描述了整个系统不同线程间的工作与交互。首先需要前端发送新建意图报告的任务的请求，服务端在接收到请求之后，需要依次和 Appium 服务端、测试设备和数据库三个线程进行交互。服务端会将用户上传的文件信息在 MySQL 数据库中做持久化处理、新建任务的具体信息也会存储在 MySQL 数据库中。服务端会启动 Appium 服务，然后通过 adb 去检测测试设备的连接情况。在服务端自动化运行 Appium

测试脚本的过程中，实际上是 Appium 服务端与测试设备之间进行交互，并通过我们实现的接口保存了测试设备运行过程中产生的文件。自动化运行测试脚本结束之后，服务端进行一系列的操作来分析测试脚本语句以及生成相关的意图报告，将其存储在 MySQL 服务器之中，最终将结果返回给前端界面，一次完整的交互就完成了。

物理视图

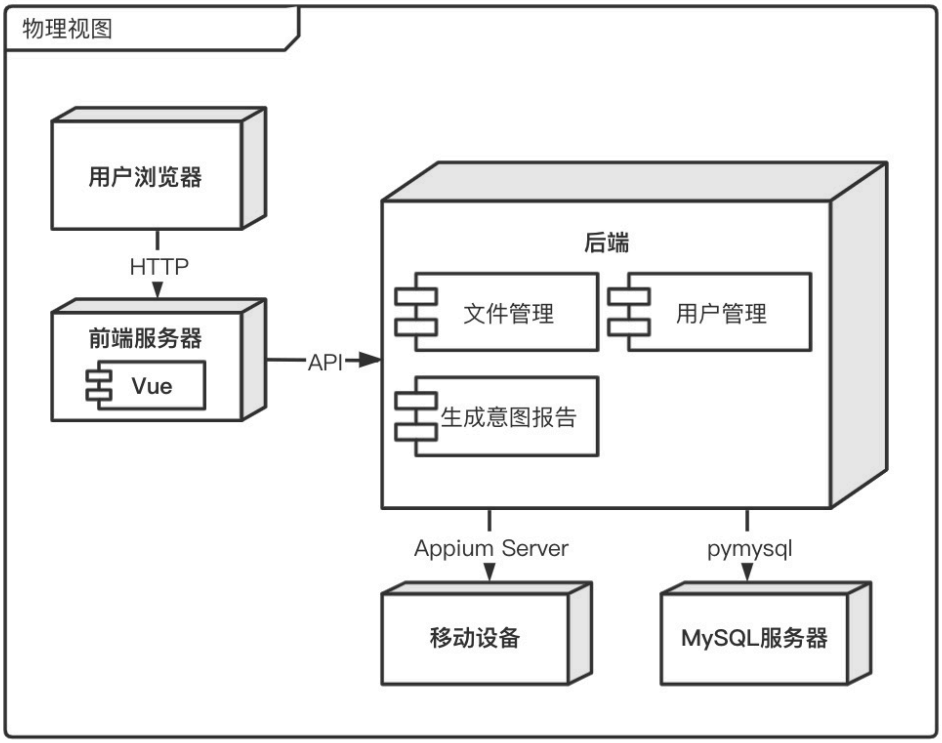

图 5-5: 物理视图

图 5-5介绍了本系统的物理视图。该物理视图涉及到的物理节点有五个，分别为用户浏览器、前端服务器、后端服务器、测试的移动设备和 MySQL 服务器。其中用户浏览器是用于发送 HTTP 请求从而来返回前端界面的 js 代码显示在浏览器界面上，前端服务器使用 Vue 框架进行开发。前端的服务器会根据路由信息调用后端的 Controller 中开发的 API 接口来使用后端服务。后端用 Django 框架实现，其主要包括了文件管理、用户管理和生成意图报告三大功能。支撑后端服务的底层物理节点有两个，移动设备和 MySQL 服务器。测试的移动设备是通过 Appium Server 和 adb 与后端服务器进行交互和数据传输的；python 可以通过 pymsql（python 3.x 中连接 MySql 的库）来请求 MySQL 服务。

开发视图

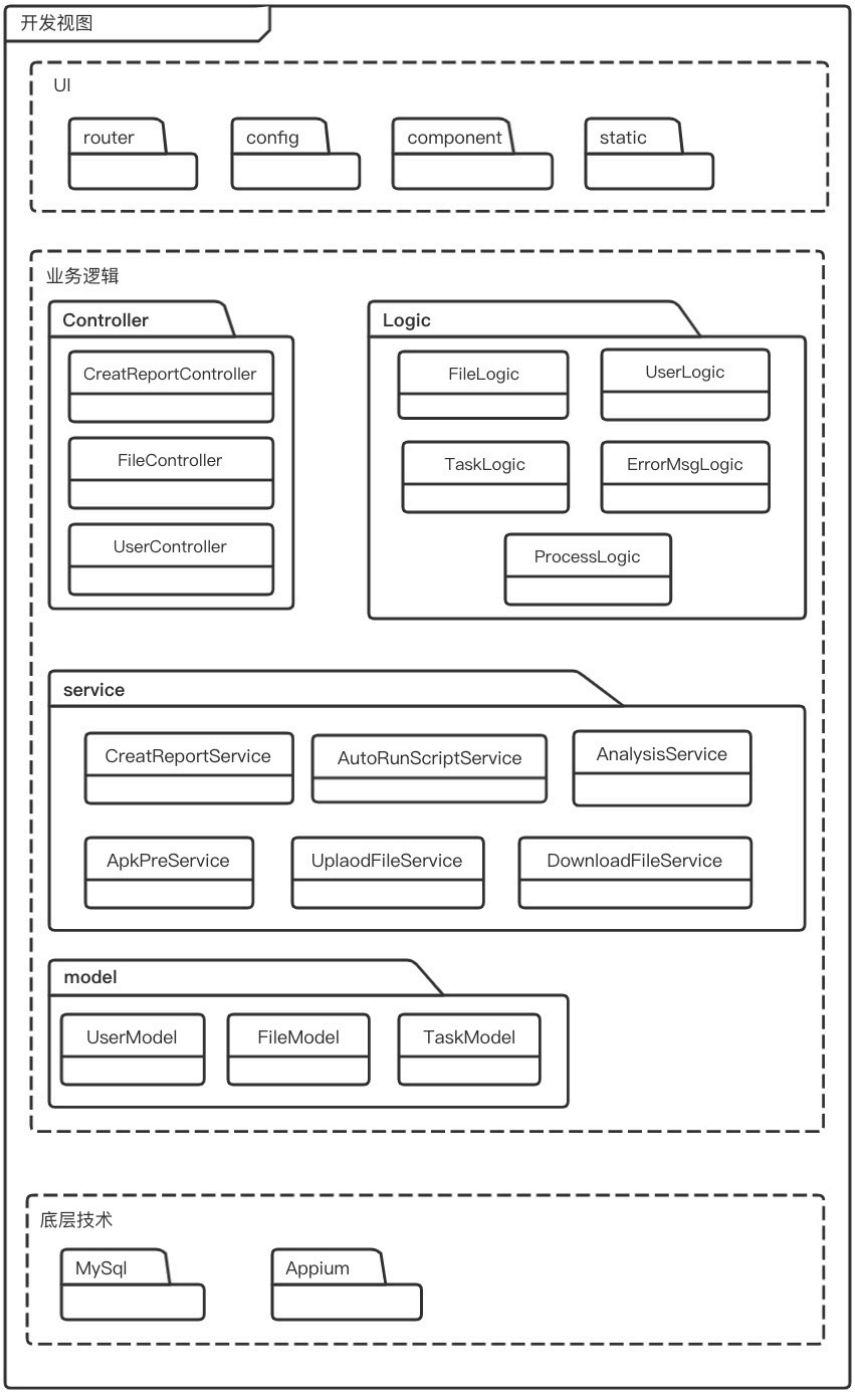

图 5-6: 开发视图

图5-6是系统的开发视图，主要是使用了三层模型，分为了 UI 层、业务逻辑层和底层技术。UI 层使用 Vue.js 框架实现，router 包存放了调用后端 API 的路由文件，config 存放了前端的配置项，component 存放了前端页面的各种页面

组件，static 存放了前端的所有静态资源文件。

业务逻辑层是支撑系统的核心，主要是由 Controller 包、Logic 包、Service 包和 model 包组成。Controller 中定义了所有与前端交互的接口，CreateReport-Controller 主要是提交测试脚本意图报告生成请求的相关接口，FileController 主要是存放了与用户上传和下载文件相关的接口，UserController 中存放了用户个人信息相关的接口。其中，这些 Controller 的接口主要是靠 Logic 包中的逻辑层进行支撑。FileLogic 存放了文件相关的逻辑；UserLogic 存放了用户信息相关的逻辑；TaskLogic 存放了一次测试脚本意图报告生成的逻辑；ErrorMsgLogic 中是在运行测试脚本意图报告过程出现的错误信息，包括 Appium 运行的错误信息等；ProcessLogic 主要是控制了整个意图报告过程的逻辑。Service 中主要是存放了所有的服务，在上文的逻辑视图中已经有详细介绍。Model 层给所有数据提供了持久化存储，包括用户数据、文件数据和任务数据等。底层由 MySQL 数据库和 Appium 服务组成。

5.3.3 系统数据库设计

实体类图

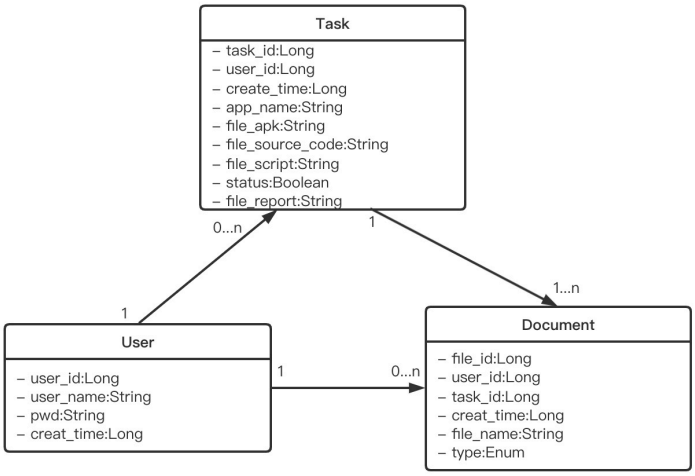

图 5-7: 实体类图

图 5-7介绍了测试脚本的意图报告生成系统的实体类图。本系统一共有三个实体类，分别为 Task、Document 和 User。其中 User 类对用户信息做一个记录，由于不是本系统核心关注的需求，所以设计较为简单。Task 为本系统最核心的类，记录了一次完整的用户发送生成意图报告请求。用户 User 类可以创建多个 Task 类，所以是 1 对 n 的实体关系。Document 类记录了系统所有的文件

信息，其中用户 User 类可以上传多份文件，Task 类也会在生成意图报告的过程中产生多份相关联的文件信息。

数据库表设计

下图是系统的三个数据库实体类的每个字段数据的具体意义。

表 5-7: User 表设计

属性	数据类型	说明
user_id	Long	用户唯一标识 id
user_name	String	用户名
pwd	String	用户密码
create_time	Long	用户创建时间

用户表 User 中 user_id 是主键，代表了用户唯一标识 id。字段 user_name 是用户自定义的用户名，不能超过 20 个字符。字段 pwd 代表了用户设置的登陆密码。字段 create_time 代表了该账户创建的时间戳信息。

表 5-8: Task 表设计

属性	数据类型	说明
task_id	Long	任务唯一标识 id
user_id	Long	用户唯一标识 id
create_time	Long	任务创建时间
app_name	String	待测安卓移动应用的应用名称
file_apk	String	用户上传的 apk 文件
file_source_code	String	用户上传的源代码 zip 包文件
file_script	String	用户上传的 Appium 测试脚本文件
status	Boolean	当前任务完成状态
file_report	String	生成的意图报告文件

Task 表中字段 task_id 是当前任务的唯一标识 id，由雪花算法在用户创建一次生成意图报告命令的时候生成，也是本表的主键。user_id 是 Task 表的外键，依赖于 User 表中的主键 user_id。字段 create_time 代表了创建该任务的时间戳信息。字段 app_name 代表了被测应用的名称。字段 file_apk、file_source_code、

file_script 和 file_report 分别代表了用户上传的 APK 文件、上传的源代码 zip 包、Appium 测试脚本文件和生成的意图报告文件。

需要说明的是 file_apk、file_source_code 和 file_script 是三个用户上传的文件，这里我们不储存其文件唯一标识 id，直接存储文件名。这样设计是方便直接对文件进行检索，不需要再二次查 Document 表。字段 status 记录了当前任务的完成状态，true 为已完成，false 为未完成或者任务失败。

表 5-9: Document 表设计

属性	数据类型	说明
file_id	Long	文件唯一标识 id
user_id	Long	用户名
task_id	Long	任务唯一标识 id
create_time	Long	文件创建时间
file_name	String	文件名
type	Enum	文件类型

Document 表中 file_id 是唯一标识的 id，user_id 和 task_id 都是表的外键，该表用于存储文件信息。user_id 依赖于 User 表，task_id 依赖于 Task 表。字段 create_time 代表了创建该任务的时间戳信息。字段 file_name 存储了当前文件的名称。字段 type 是枚举类型，声明了当前文件的类型。由于系统在运行过程中会产生多个不同种类的文件，所以其类型较多，有 APK 包、源代码包、Appium 测试脚本、预处理后 APK 包、运行产生的日志文件、静态模版提取的 UI 回调函数、生成的控件 UI 截图信息、最后整合的意图报告结果文件。

5.4 本章小结

本章介绍了测试脚本意图报告系统的需求分析和概要分析。5.1 节主要是从用户角度分析了本系统的功能性需求和非功能性需求。5.2 节是针对系统用例进行列举，以及阐述了该系统用例的详细需求规格说明。5.3 节是系统的概要设计，首先是架构设计的说明；其次是利用“4+1”视图模型从不同角度介绍了本系统的一些详细设计；最后针对数据库的设计进行简要阐述。

第六章　系统详细设计与实现

6.1　文件模块的设计与实现

6.1.1　文件模块概述

文件模块主要是存储用户上传和下载的文件、以及对应的在生成意图报告过程中产生的文件进行管理。文件模块包括了用户在上传文件或者下载文件的接口，也包括了文件信息的数据库持久化操作。在生成意图报告的过程中，存在大量中间文件的生成，为了存储中间过程的各种文件，我们也使用文件模块对这些文件进行保存、文件的拷贝删除、文件夹的压缩解压和数据库存储。在生成意图报告之后，我们会将在 XPath 分析和 ID 分析的中间文件进行提取和存储，生成对应的 README 文件供用户下载查阅。

6.1.2　文件模块核心类图

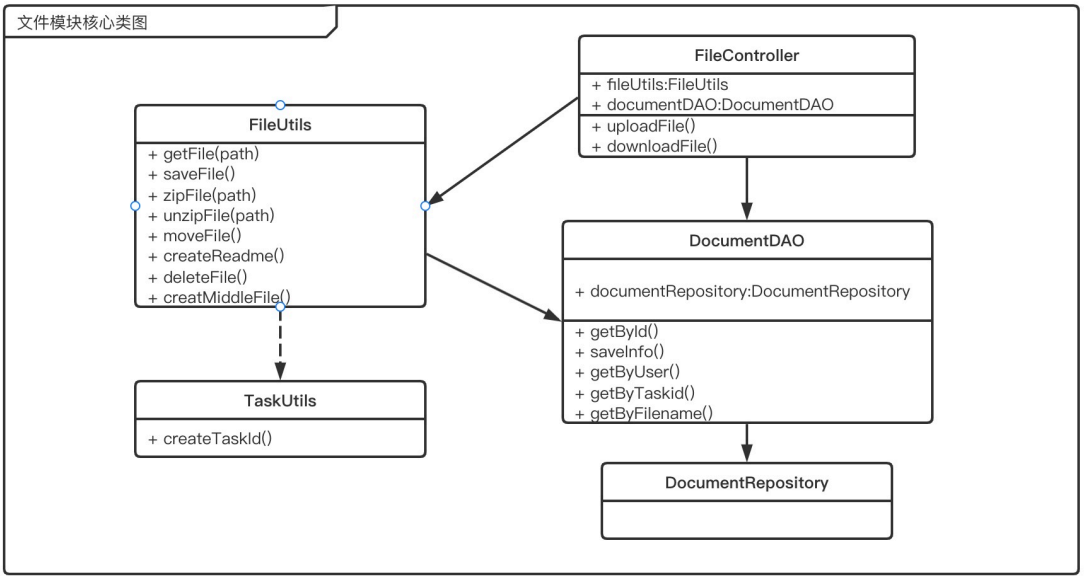

图 6-1: 文件模块核心类图

图 6-1是文件模块的核心类图。FileUtils 类时文件的工具类，给文件提供一些通用的工具方法。TaskUtils 类主要是用于在创建一次测试脚本的意图报告任务时生成全局唯一的任务号 Task_id。DocumentDAO 类是数据库中的文件数据访问对象，DocumentRepository 类是文件数据的仓储层，用于访问 MySQL 服务。FileController 类是文件上传下载相关的控制器，给文件接口提供服务。FileController 类通过使用 FileUtils 类和 DocumentDAO 来实现其方法。在 FileUtils 类中，getFile() 和 saveFile() 分别是获取文件内容和保存文件内容；zipFile() 和 unzipFile() 是对文件或者文件夹进行压缩和解压操作；moveFile() 是对文件进行移动拷贝操作；deleteFile() 是删除文件操作；createMiddleFile() 是根据意图报告结果生成 XPath/Id 分析产生的媒体文件和 UI 回调函数，我们称之为中间文件，该文件夹为 data，子文件夹根据测试脚本的语句按照 index 递增来命名；createReadme() 是在一次完整的意图报告生成之后，用户点击下载所有文件，系统会将 Appium 测试脚本、测试应用 APK、意图报告、源代码 zip 包和中间文件放到一个文件夹中，并产生 README 文件，文件中介绍了以上每个文件的内容。图 6-2是以 ContactManager_1.java 脚本为例产生的文件夹中的 README 文件内容。DocumentDAO 类中是文件数据表的增删改查功能，核心功能有按照文件 Id 查询、按照用户 Id 查询、按照任务 Id 查询和按照文件名进行查询和保存一行文件数据。

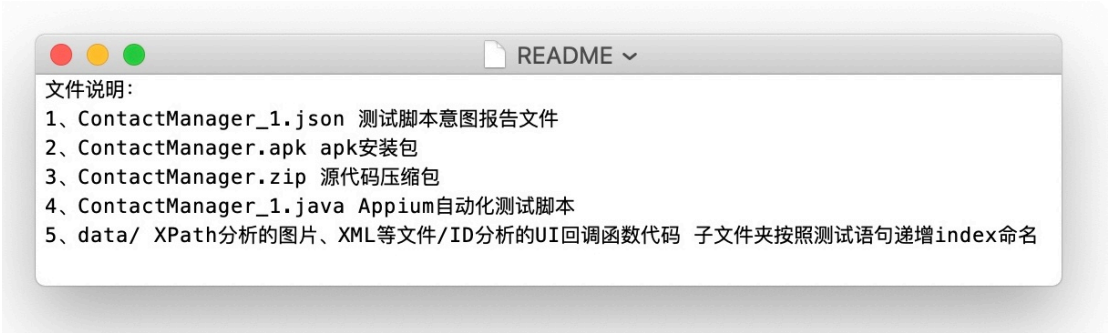

图 6-2: README 文件

6.1.3 文件模块具体实现

文件存储路径

所有的文件均存储在 FILE_ROOT_PATH/document/文件夹下面。表 6-1展示了在一个完整用户交互过程中整个系统产生的文件以及存储路径 (以移动应

用 test 为例)。我们先假设当前用户的一次完整交互所在的根目录为 task_id，在下文中会继续给出 task_id 具体的数值。

表 6-1: 文件存储路径

编号	文件名称	文件描述	所在路径
D1	test.apk	用户上传 APK 包	./document/task_id/apk/test.apk
D2	test.java	用户上传 Appium 测试脚本	./document/task_id/script/test.java
D3	Test/	用户上传的源代码	./document/task_id/AndroidStudioProject/Test/
D4	test_output.apk	插桩处理之后的 APK 文件	./document/task_id/apk_output/test_output.apk
D5	test.apk	test_output.apk 签名处理之后的 APK 文件	./document/task_id/apk_output/test.apk
D6	log	运行 appium 测试脚本产生的日志文件	./document/task_id/log/log
D7	log_deal	针对 D6 的 Log 文件进行分割后的日志文件	./document/task_id/log_deal/
D8	response_method	静态模版方法提取的 UI 回调函数	./document/task_id/response_method/
D9	ui	生成的截图信息和 XML 信息等	./document/task_id/ui/
D10	result	最后整合的意图报告结果	./document/task_id/result

在表 6-1中，D1、D2、D3 是用户上传的文件。D3 是将源代码的 zip 包解压之后的文件，其内部格式应该满足 Test/src/main/java/这样的格式，属于标准的安卓开发项目。D10 是用户可以选择下载的结果文件。除此之外，其他文件都是在运行自动化测试脚本和生成意图报告的结果过程中产生的中间文件。我们将其统一放到一个目录下面方便进行管理。D7 是一个文件夹，其内部的文件应该以测试语句的 index 递增的顺序进行命名。D8 内部的文件以 ID 名称来命名，如果 ID 不存在的话，则文件不存在，代表静态模版匹配不成功的情况。D9 是在运行 Appium 自动化测试脚本过程中，产生的截图信息和 XML 信息。D9 下的子文件夹应该按照 index 递增的顺序进行命名，子文件夹内部的文件包括三个，第一个是当前步骤的屏幕截图、第二个是当前步骤的 XML 布局文件、第三个是前两个文件生成的控件的截图信息。

task_id 生成

每一次使用系统操作时的 task_id 路径应该是唯一的，这里我们使用雪花算法来实现 task_id 的生成。该算法属于半依赖数据源方式，原理是使用 Long 类

型（64 位），按照一定的规则进行填充：时间（毫秒级）+ 集群 ID+ 机器 ID+ 序列号，每部分占用的位数可以根据实际需要分配，其中集群 ID 和机器 ID 这两部分，在实际应用场景中要依赖外部参数配置或数据库记录。本系统不涉及到分布式系统和多台服务器，所以集群 ID 和机器 ID 保持一致就好。保留这两部分，也可以提高系统的可扩展性。雪花算法的优点在于高性能、低延迟、按时间有序、生成效率极高。由于其与当前的时间戳有关系，所以生成的序列是按照时间升序排列的。图 6-3是参考了雪花算法生成 task_id 的具体实现。

```
class create_task_id(object):
   def __init__(self, worker_id, sequence=0):
      self.worker_id = worker_id
      self.sequence = sequence
      self.last_timestamp = -1 # 上次计算的时间戳
   def get_timestamp(self): #生成毫秒级时间戳
      return int(time.time() * 1000)
   def wait_next_millis(self, last_timestamp):# 等到下一毫秒
      timestamp = self.get_timestamp()
      while timestamp <= last_timestamp:
         timestamp = self.get_timestamp()
      return timestamp
   def get_id(self):
      timestamp = self.get_timestamp()
      if timestamp == self.last_timestamp:
         self.sequence = (self.sequence + 1) & SEQUENCE_MASK
         if self.sequence == 0:
            timestamp =
               self.wait_next_millis(self.last_timestamp)
      else:
         self.sequence = 0
      self.last_timestamp = timestamp
      new_id = ((timestamp - TWEPOCH) << TIMESTAMP_LEFT_SHIFT) |
         (self.worker_id << WORKER_ID_SHIFT) | self.sequence
      return new_id
```

图 6-3: 生成 task_id

6.2 APK 预处理模块的设计与实现

6.2.1 APK 预处理模块概述

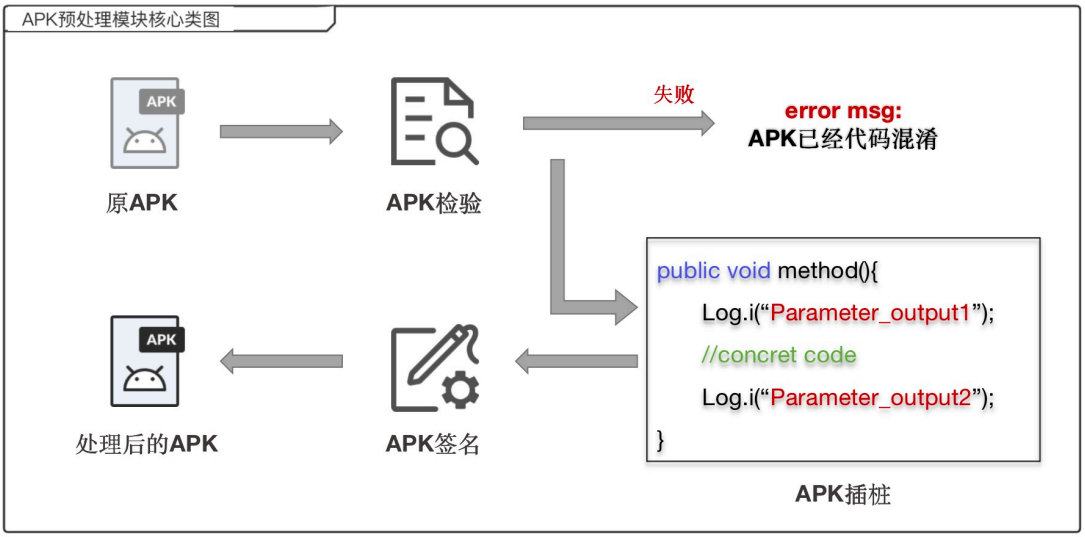

图 6-4: APK 预处理模块流程图

APK 预处理模块是生成意图报告的第一步，主要用于对用户上传的 APK 包进行插桩、签名和校验等操作。用户上传 APK 之后，我们需要利用 soot 框架对 APK 进行插桩处理，这里是将函数的开始和结尾分别插入一句 Log 语句，输出其函数签名、所在包名等信息，方便我们后续对该日志在其源码中查找对应的函数。之后，对已经插桩的 APK 进行签名。最后，我们对 APK 包进行校验，这里主要是验证 APK 是否被混淆。

6.2.2 APK 预处理模块的核心类图

该模块的核心类图如图 6-5所示。可以看出，其核心服务是 ApkPreprocessingService，在这个服务中包含了三个子服务，分别是 APK 插桩服务 ApkStackingService、Apk 签名服务 ApkSignedService 和校验服务 ApkCheckService。除此之外，还包括了 DocumentDAO、DocumentService 和 ErrorMsg 三个类。其中 DocumentDAO 用于将文件信息持久化到数据库中，当我们处理完用户上传的 APK 之后，我们会得到一个新的 APK 包，利用这个 DAO 可以进行数据存储。DocumentService 用于获取文件对象 File，我们使用 Django 自带的文件系统对

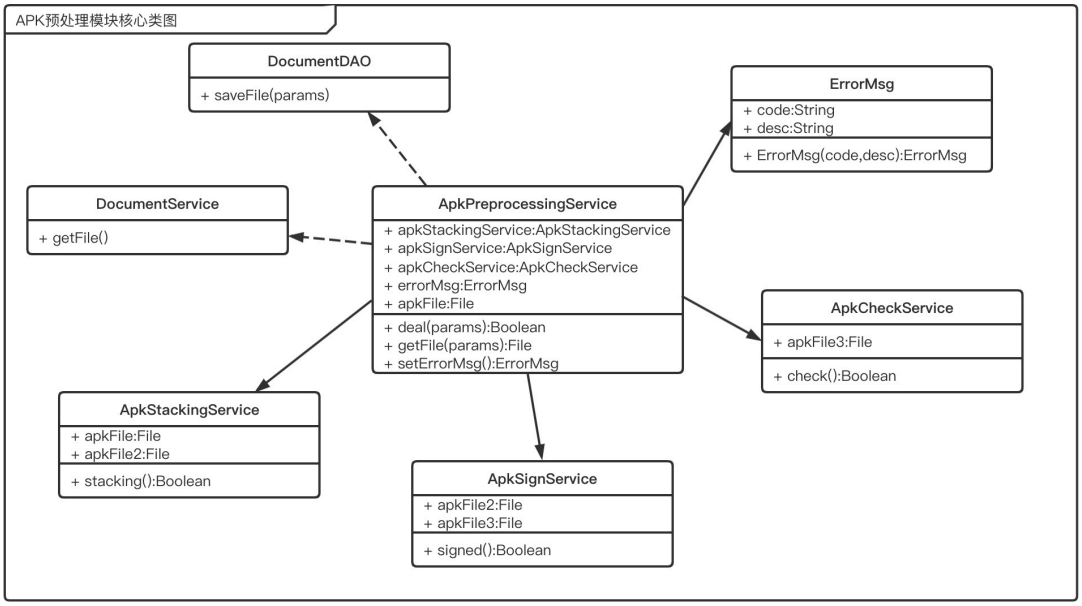

图 6-5: APK 预处理模块核心类图

File 类进行处理，这里包括文件的打开、写入和关闭。ErrorMsg 存储了在这个过程中出现的错误信息，其类型 type 分为三种。第一种是 APK 插桩失败，值为 1；第二种是 APK 签名失败，值为 2；第三种是 APK 已经被混淆，值为 3。ApkStackingService 对 APK 包进行插桩处理，这部分我们使用了著名的 soot 框架，这部分使用 Java 代码实现。我们的调用方式是将 Java 项目打成 jar 包，通过命令行的方式进行穿参和调用。ApkSignedService 是将插桩后的 APK 进行签名，以便后续自动化运行测试脚本的时候可以将 APK 包安装到测试设备上。签名我们使用了 jarsigner 工具，其使用方式也是通过命令行进行调用。ApkCheckService 是针对 APK 进行检验，这里使用了 Dex2jar 工具将 APK 转化为 jar 包后查看其源代码信息，如果是发现命名方式是 a、b、c 等，则该 APK 包已经被混淆，是不符合我们要求的 APK 文件。

6.2.3 APK 预处理模块的具体实现

初始化 soot 框架参数

图 6-6是初始化 soot 框架设置的代码片段。该函数的入参是 APK 的路径。在初始化的开始，我们每次调用 soot 框架提供的服务时，都需要进行 reset 操作。设置 Android jar 包路径是为了满足不同版本的 jar 包，soot 框架会自动提取当前 APK 所使用的 Android 版本号，再通过该路径去调用对应

的 jar 包。输出格式我们设置为 APK 包。值的注意的是，部分 APK 需要设置 process_multiple_dex 为 true。

```
public static void initsoot(String apk) {
      G.reset();

      //设置允许伪类（Phantom
         class），指的是soot为那些在其classpath找不到的类建立的模型
      Options.v().set_allow_phantom_refs(true);

      //将虚拟机的classpath预置到Soot自己的classpath中。
      Options.v().set_prepend_classpath(true);
      //设置soot的输出格式
      Options.v().set_output_format(Options.output_format_dex);
      //设置android jar包路径
      Options.v().set_android_jars(androidJar);
      Options.v().set_src_prec(Options.src_prec_apk);
      Options.v().set_process_dir(Collections.singletonList(APK));
      Options.v().set_force_overwrite(true);
      //有的可能需要
      Options.v().set_process_multiple_dex(true);
      Scene.v().loadNecessaryClasses();
   }
```

图 6-6: 初始化 soot 代码

soot 插桩

使用 soot 框架进行插桩时，我们需要自定义 BodyTransformer，并实现 internalTransform() 方法。该方法的入参有三个，其中 Body 类是将 Java 中一个方法的方法体转化为 soot 自定义的参数类型，也是重写这个方法中主要使用的参数。其中，Body 对应 Java 类中的一个方法，Unit 代表方法的每一行语句。我们首先通过 arg0 变量获取方法体里面的每行语句的迭代器，再获取该方法的方法名 methodName 和所属的类名 declaringClass。我们需要过滤掉安卓原生的类，这部分类我们不做插桩处理，过滤规则是使用包名进行前缀匹配符合“android”或者“com.google.android”的类，我们不予处理。

```
static class MyTransform extends BodyTransformer {
   @Override
   protected void internalTransform(Body arg0, String arg1,
                                    Map<String, String> arg2) {
      //获取Body里所有的units
      Iterator<Unit> unitsIterator =
         arg0.getUnits().snapshotIterator();
      //其中，Body对应Java类中的一个方法，Unit代表方法的每一行语句
      // 获取当前类名
      String declaringClass =
         arg0.getMethod().getDeclaringClass().getName();
      // 获取当前方法签名
      String methodName = arg0.getMethod().getDeclaration();
      Stmt stmtEnd = null;
      Boolean flag = true;
      if (declaringClass.startsWith(”android”) ||
         declaringClass.startsWith(”com.google.android”)) {
         return;
      }
```

图 6-7: soot 自定义插桩方法 1

之后，我们使用 unitsIterator 对每一行语句进行迭代。首先会将 Unit 强制转换为 Stmt,Stmt 为 Jimple 里每条语句的表示。首先，判断其是否为调用语句，这是为了筛选掉注释语句。如果当前 flag 为 true，代表当前的 stmt 为第一行语句，开始的 Log 日志由“info start”来标志，将函数签名信息和当前类信息输入到 makeToast() 函数中来新建一行安卓日志语句。我们使用 insertBefore() 在迭代器的第一行语句前一行插入带有开始标志的安卓日志代码，然后将 flag 置为 false，代表当前函数的开始日志已经插入完成。stmtEnd 用于保存最后一行有效的语句，当前循环结束后，如果当前函数不为空也就是 stmtEnd 不为空，我们生成带有结尾标志的安卓日志代码。这行语句也使用了 makeToast() 函数生成，不同于之前的是，我们使用 info end 作为开始标志，然后我们使用 insertAfter() 方法在迭代器的最后一行语句的后一行插入带有结束标记的安卓 Log 日志。

```
        while (unitsIterator.hasNext()) {
            Stmt stmt = (Stmt) unitsIterator.next();
            //将Unit强制转换为Stmt,Stmt为Jimple里每条语句的表示
            if (stmt.containsInvokeExpr()) {//如果是一条调用语句
                stmtEnd = stmt;
                if (flag) {
                    List<Unit> toastUnits = makeToast(arg0, "info
                        start -- declaringClass:" + declaringClass + "
                        methodName:" + methodName);
                    arg0.getUnits().insertBefore(toastUnits, stmt);
                    //在这条语句之前插入Toast消息
                    flag = false;
                }
            }
        }
        if (stmtEnd != null && !flag) {
            List<Unit> toastUnits = makeToast(arg0, "info end --
                declaringClass:" + declaringClass + " methodName:" +
                methodName);
            //在这条语句之后插入Toast消息
            arg0.getUnits().insertAfter(toastUnits, stmtEnd);
        }
    }
}
```

图 6-8: soot 自定义插桩方法 2

图6-9是上文中生成 Log 日志语句的方法 makeToast()。入参是当前的方法体 Body 变量和当前需要输出的日志信息 toast 字符串，插入的 Log 语句为 Log.i("","")，也是安卓标准的输出日志方式，其中 i 表示当前日志等级为 info。首先，我们新建一个 java 代码语句的列表 unitsList。我们使用了 Scene.v().getMethod() 方法去获取安卓日志所属类 android.util.Log 并赋值给变量 logClass，然后使用 logClass.getMethod() 方法获取方法 int i(java.lang.String,java.lang.String)，输入参数包含了函数名、入参和出参，这里是利用了 Java 反射的原理。Log.i() 是一个静态方法，所以不需要新建 Log 实例，我们使用了

Jimple.v().newStaticInvokeExpr 去生成句柄并转化成 InvokeStmt 类型并加入我们之前定义好的 unitsList 变量中返回。

```
private List<Unit> makeToast(Body body, String toast) {
        List<Unit> unitsList = new ArrayList<Unit>();
        SootClass logClass =
            Scene.v().getSootClass("android.util.Log");
        //获取android.util.Log类
        SootMethod sootMethod = logClass.getMethod("int
            i(java.lang.String,java.lang.String)");
        StaticInvokeExpr staticInvokeExpr =
            Jimple.v().newStaticInvokeExpr(sootMethod.makeRef(),
            StringConstant.v("logger"), StringConstant.v(toast));
        InvokeStmt invokeStmt =
            Jimple.v().newInvokeStmt(staticInvokeExpr);
        unitsList.add(invokeStmt);
        return unitsList;
    }
```

图 6-9: 创建插桩语句句柄

6.3 自动化运行模块的设计与实现

6.3.1 自动化运行模块概述

自动化运行模块主要是自动化运行用户上传的 Appium 测试脚本并记录下相关文件信息。我们需要在该模块开启 Appium 服务端和检查其测试设备的连接情况，将 Appium 测试脚本替换到有 Appium 环境的 Java 项目中，使用 mvn 命令对项目进行编译和运行。这个模块中记录的文件有三部分，第一部分是 Appium 的 Java 项目在编译、运行过程中的控制台信息，第二部分是测试设备在运行该应用时产生的日志文件，第三部分是运行时的媒体信息。

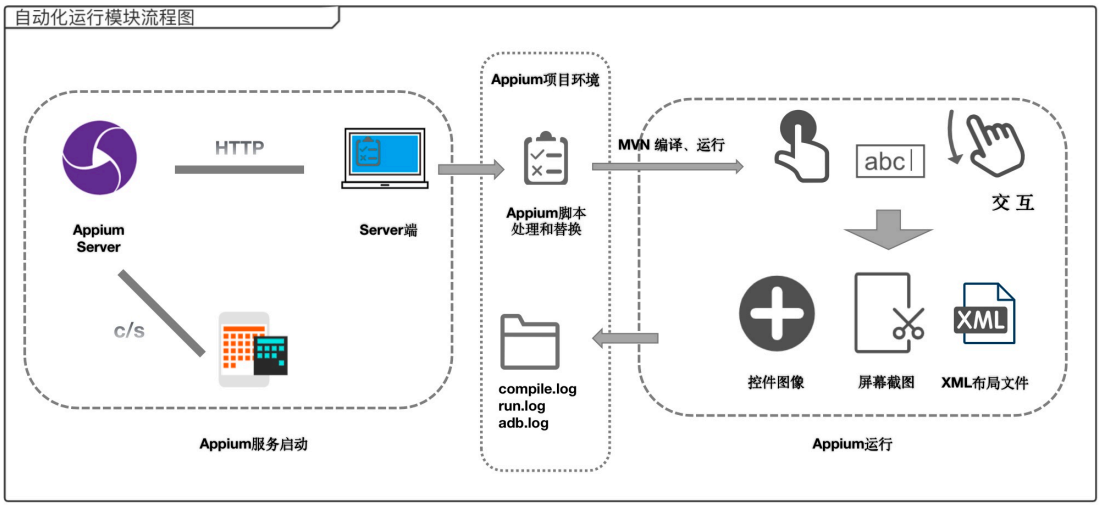

图 6-10: 自动化运行模块流程图

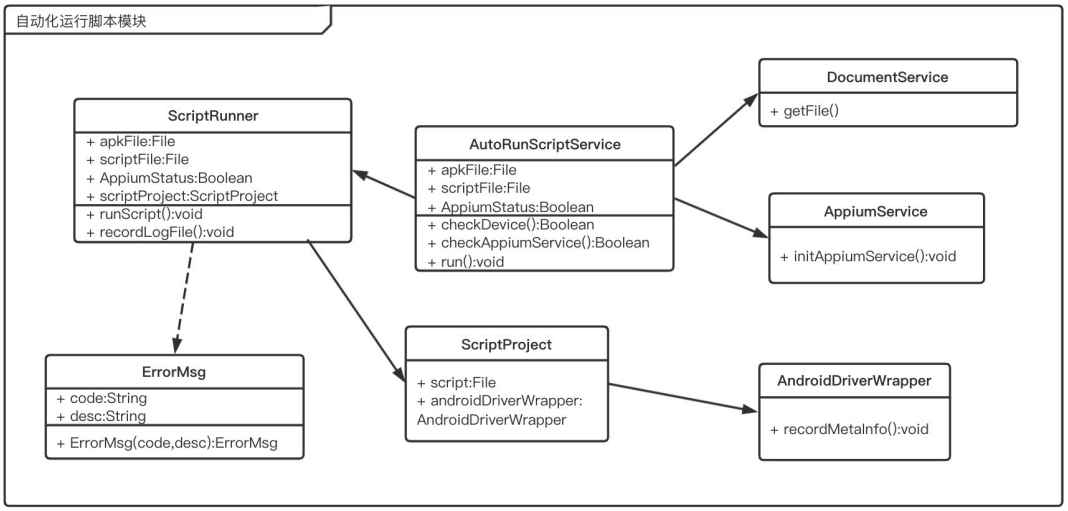

图 6-11: 自动化运行模块的核心类图

6.3.2 自动化运行模块的核心类图

图 6-11 是自动化运行模块的核心类图。该模块的核心服务是 AutoRunScriptService。AutoRunScriptService 包括了三大功能，分别是设备连接情况检测、Appium 服务开启状态检测和自动化运行测试脚本。AutoRunScriptService 需要获取用户上传的 Appium 测试脚本文件和 APK 预处理模块中获得的 APK 包。设备连接服务检测是 checkDevice()，我们通过 adb 命令来判断当前的端口是否和测试设备保持连接状态。方法 checkAppiumService() 是检测 Appium 服务的开启状态，如果当前未开启 Appium 服务的端口号，需要 AppiumService 类中

的 initAppiumService() 方法另外开启一个进程来启动 Appium 服务。方法 run() 是核心功能，自动化运行 Appium 测试脚本。这部分功能我们使用 ScriptRunner 类来实现，其变量除了必要的文件信息外，还有 scriptProject 是具备了 Appium 环境的测试脚本运行项目。其中方法 runScript() 会将用户上传的 Appium 测试脚本替换到 scriptProject 相对应的位置中，并且调用 Java 的命令行对其进行打包，再运行该 jar 项目。方法 recordLogFile() 会记录在运行该 jar 项目时的控制台输出信息（包括编译信息和运行信息）和当前测试机器在运行 Appium 脚本时产生的日志文件。如果在 recordLogFile() 期间产生错误信息，如运行测试脚本失败等，则会转换成错误信息 ErrorMsg 类。ScriptProject 除了具备 Appium 测试脚本的运行环境外，还实现了 AndroidDriverWrapper 提供的接口，这里是为了记录下运行过程中产生的媒体信息，包括每行测试语句的 XML 布局文件、当前页面截图信息和当前控件的图像信息。

6.3.3 自动化运行模块的具体实现

Appium 测试脚本替换

```
private static void runScript(String path, String outDir) {
   // 1. 读取并预处理脚本文件。
   String scriptContents;
   File srcFile = new File(path);
   String srcFileName = srcFile.getName();
   String scriptClassName = srcFileName.substring(0,
      srcFileName.lastIndexOf('.'));
   try (BufferedReader reader = new BufferedReader(
         new FileReader(srcFile)
   )) {
      StringJoiner joiner = new StringJoiner("\n");
      //脚本文件处理操作
      reader.lines().map(s -> s
         .replaceAll("^package test\\..+;$", "package test;")
         .replaceAll("(\\s+)(AppiumDriver|AndroidDriver)",
            "$1AndroidDriverWrapper")
         .replaceAll("(\\s+)" + scriptClassName, "$1Script")
         .replaceAll("^.+\\.setCapability\\(\"app\".+\\);\\s*$","")
         .replaceAll("^import test\\..*;\\s*$", "")
```

```
    ).forEach(joiner::add);
    scriptContents = joiner.toString();
} catch (IOException e) {
    String msg = String.format("Failed to read the script
        file: %s.", path);
    throw new RuntimeException(msg, e);
}
```

图 6-12: Appium 测试脚本替换的详细代码 1

代码片段图6-12和图6-13是将用户上传的 Appium 测试脚本替换到已经具备运行环境的 Java 项目中。第一步我们需要读取脚本文件并进行预处理。srcFile 是当前输入的脚本文件对象，srcFileName 是当前文件的全路径名称，scriptClassName 是当前脚本定义的类名。我们使用 BufferedReader 来进行文件的缓冲读入并处理，如果当前文件的读入发生 IOException 时，会捕获异常并对输出 RuntimeException 异常，异常信息为“Failed to read the script file”文件读入异常和当前文件的所属路径信息。在处理脚本的过程中，我们主要是用 map() 函数来进行以下几个步骤的处理。这里需要添加当前 Java 文件的包信息（我们的 scriptProject 所在包名为 test），将 AppiumDriver 替换为 AndroidDriverWrapper（AndroidDriverWrapper 中定义了一些媒体文件的存储），更改文件名为 Script.java（我们的 scriptProject 中的脚本文件名为 Script.java），移除语句 setCapability("app", String) 和语句 import test.*。

```
// 2. 写入script Project中
String dstPath = RUNNING_DIR +
    "/src/main/java/test/Script.java";
try (FileWriter writer = new FileWriter(dstPath)) {
    writer.write(scriptContents);writer.flush();
} catch (IOException e) {
String msg = String.format("Failed to write to %s.",dstPath);
    throw new RuntimeException(msg, e);
}}
```

图 6-13: Appium 测试脚本替换的详细代码 2

第二步我们将 scriptProject 中的 Script.java 文件打开并使用之前处理好的文本进行覆盖。变量 dstPath 是目标文件的路径。我们使用 FileWriter 进行文件的写入操作，然后调用 writer.flush() 是将缓冲区的字符写入到文件中。如果文件写入发生 IOException 异常，会转换成 RuntimeException 抛出并输出错误信息“Failed to write to”和目标文件路径。

记录运行时的输出信息

在 Appium 测试脚本替换到 scriptProject 之后，我们调用 recordLogFile() 方法对该项目进行编译或运行操作并记录相关的日志信息和当前设备运行测试脚本后的安卓日志信息。

```
private static void recordLogFile(String path, String
   outDir,String scriptClassName) {
   String compileCmd = ”mvn compile”;
   String runCmd = ”mvn exec:java -Dexec.mainClass=test.Main
      -Dexec.args=\”” + new File(outDir).getAbsolutePath()”;
   File workingDir = new File(RUNNING_DIR);
   try {
      //1. 编译脚本项目
      Process compileProcess =
         Runtime.getRuntime().exec(compileCmd, null, workingDir);
      String compileLog = ”logs/” + scriptClassName +
         ”.compile.log”;
      writeToFile(compileLog, compileProcess.getInputStream());
      writeToFile(compileLog, compileProcess.getErrorStream());
```

图 6-14: 记录运行时的输出信息详细代码 1

首先我们预定义 compileCmd 字符串和 runCmd 字符串，分别为编译命令和运行命令。变量 workingDir 为当前的工作路径。其中我们使用“mvn compile”命令对 Java 项目进行编译，使用“mvn exec:java -Dexec.mainClass=test.Main -Dexec.args=”+new File(outDir).getAbsolutePath()”对编译成功的项目进行运行。在编译阶段，我们使用 Runtime.getRuntime().exec() 方法进行 shell 命令的运行，变量 compileLog 为输出的编译项目的日志文件路径。我们将编译过程 compileProcess 中的正常输出流写入 compileLog，如果是编译项目出错则使用 getErrorStream() 方法输出错误流写入编译的日志中。如果 compilePro-

cess.waitFor() 为 0 的话，表明当前的编译流程进行完毕。

然后，我们进行脚本项目的运行操作，与编译的步骤类似。我们也通过 Runtime.getRuntime().exec() 方法运行 shell 命令。变量 runningLog 定义了运行过程中的输出日志路径，同时使用 writeToFile() 方法记录当前 runProcess 的输出流或者错误流并写入 run.log 文件中。如果当前的 runProcess 命令运行出现问题，则会抛出相对应的 RuntimeException，参数为“Failed to run the script.”。

```
if (compileProcess.waitFor() == 0) {
   //2. 运行脚本项目
   Process runProcess = Runtime.getRuntime().exec(runCmd,
      null, workingDir);
   String runningLog = ”logs/” + scriptClassName +
      ”.run.log”;
   writeToFile(runningLog, runProcess.getInputStream());
   writeToFile(runningLog, runProcess.getErrorStream());
   if (runProcess.waitFor() != 0) {
      throw new RuntimeException(”Failed to run the
         script.”);
   }
```

图 6-15: 记录运行时的输出信息详细代码 2

最后，我们使用“adb logcat”命令去获取测试设备的日志信息，同时使用管道去过滤 declaringClass 中前缀符合当前 APK 包名的类。这里的过滤是排除了安卓应用在开发过程中引用第三方库的类的日志信息。我们使用 Runtime.getRuntime().exec() 来运行获取设备日志的命令，并写入到.adb.log 文件中。在这个过程中，这三条命令如果出现运行的错误，会抛出 RuntimeException 的异常信息，并输出相关信息来定位错误出现的阶段。

```
//3. 记录当前测试设备的日志文件
String adbLogFile=”adb logcat|grep declaringClass:”+path;
Process adbLogProcess = Runtime.getRuntime()
.exec(adbLogFile, null, workingDir);
String adbRunningLog = ”logs/” + scriptClassName +
   ”.adb.log”;
writeToFile(adbRunningLog, adbLogProcess.getInputStream());
```

```
      if (adbLogProcess.waitFor() != 0) {
         throw new RuntimeException("Failed to record the log."
            + path);
      }
   } else {
      throw new RuntimeException("Failed to compile the script."
         + path);
   }
} catch (IOException | InterruptedException e) {
   String msg = "Failed to run the script." + path;
   throw new RuntimeException(msg, e);
}
```

图 6-16: 记录运行时的输出信息详细代码 3

记录并存储媒体信息

AndroidDriverWrapper 类是继承了原有的 Appium 驱动 AndroidDriver 类并实现了 Consumer<WebElement> 接口，目的是为了在运行测试脚本的过程中记录我们想要的媒体信息。AndroidDriverWrapper 类中存放了两个变量，其中变量 dumpDir 是媒体文件的存储目录，变量 step 用于从 0 开始依次记录当前步骤的索引数。

```
public class AndroidDriverWrapper extends
   AndroidDriver<WebElement> implements Consumer<WebElement> {
   /** 存放结果的目录 */
   private static String dumpDir = "result";
   /** 步骤计数索引 */
   private static int step = 0;
```

其中，Consumer<WebElement> 接口需要实现的方法是 accept(WebElement e) 。该方法中创建了三个文件，截图文件 screen.png、控件图像信息 widget.png 和当前屏幕的 XML 文件 ui.xml。设备当前的屏幕截图我们调用 this.getScreenshotAs(OutputType.FILE) 来获取；控件图像是调用方法 e.getScreenshotAs(OutputType.FILE)，其中 e 是当前的控件 WebElement；this.getPageSource() 用于获取当前页面的 XML 文件。我们根据 step 值来新建该测试步骤的文件

夹，并复制屏幕截图和控件图像并使用 FileWriter 将 XML 缓冲写入对应文件中。这个过程也会对 IOException 进行捕获。

```
@Override
public void accept(WebElement e) {
   File stepDir = new File(dumpDir, String.valueOf(step));
   File screenSrcFile = this.getScreenshotAs(OutputType.FILE);
   File screenDstFile = new File(stepDir, "screen.png");//截图文件
   File widgetSrcFile = e.getScreenshotAs(OutputType.FILE);
   File widgetDstFile = new File(stepDir, "widget.png");//控件图像
   String pageSrc = this.getPageSource();
   File pageSrcFile = new File(stepDir, "ui.xml");//XML文件
   try {
      FileUtils.deleteDirectory(stepDir);
      FileUtils.forceMkdir(stepDir);
      FileUtils.copyFile(screenSrcFile, screenDstFile);
      FileUtils.copyFile(widgetSrcFile, widgetDstFile);
      try (FileWriter writer = new FileWriter(pageSrcFile)) {
         writer.append(pageSrc).flush();
      }
   } catch (IOException ioException) {
      ioException.printStackTrace();
   }
   step++;
}
```

图 6-17: 记录并存储媒体信息

6.4 测试脚本分析模块的设计与实现

6.4.1 测试脚本分析模块概述

测试脚本分析模块功能是分析 Appium 测试脚本并生成意图报告的结果。根据第三章，我们会将 Appium 测试脚本的语句分为 XPath 控件和 ID 控件两种类型。针对 XPath 类型控件，XPath 类型有两种情况，分别为层次结构加

content-desc 和层次结构加控件类型两种。content-desc 可以直接提取其文本信息，后者我们通过在自动化运行时获取的媒体文件输入模型中得到其意图信息。针对 ID 类型的控件，我们会首先使用静态模版方法去匹配 UI 回调函数，如果没有匹配成功，我们使用日志信息去定位其 UI 回调函数。将 UI 回调函数输入到 code2seq 模型中得到意图信息。最后，我们将这两部分的意图结果进行合并，得到最终的意图报告。

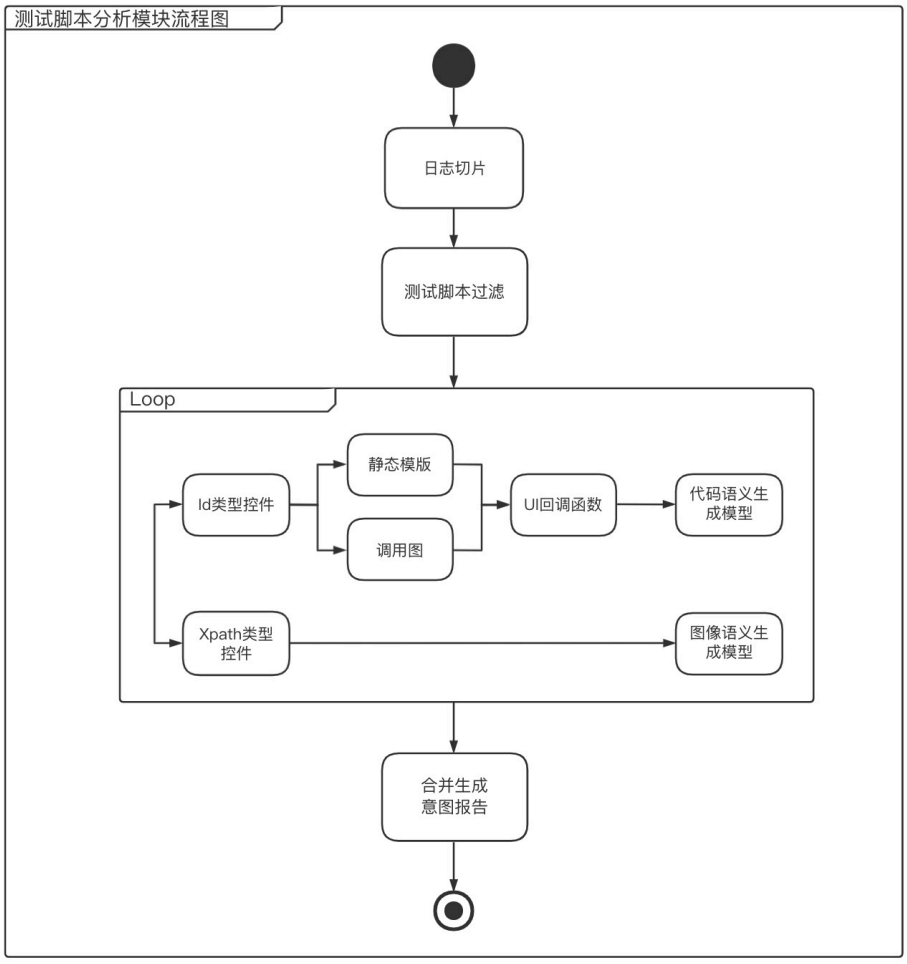

图 6-18: 测试脚本运行模块流程图

6.4.2 测试脚本分析模块的核心类图

测试脚本分析模块的核心类是 ScriptAnalysisService。ScriptAnalysisService 中的核心变量有四个，stmtList 保存了 Appium 测试脚本中的每行测试语句；resultList 中存储了测试语句以及生成的意图结果信息；idAnalysisService 是 ID

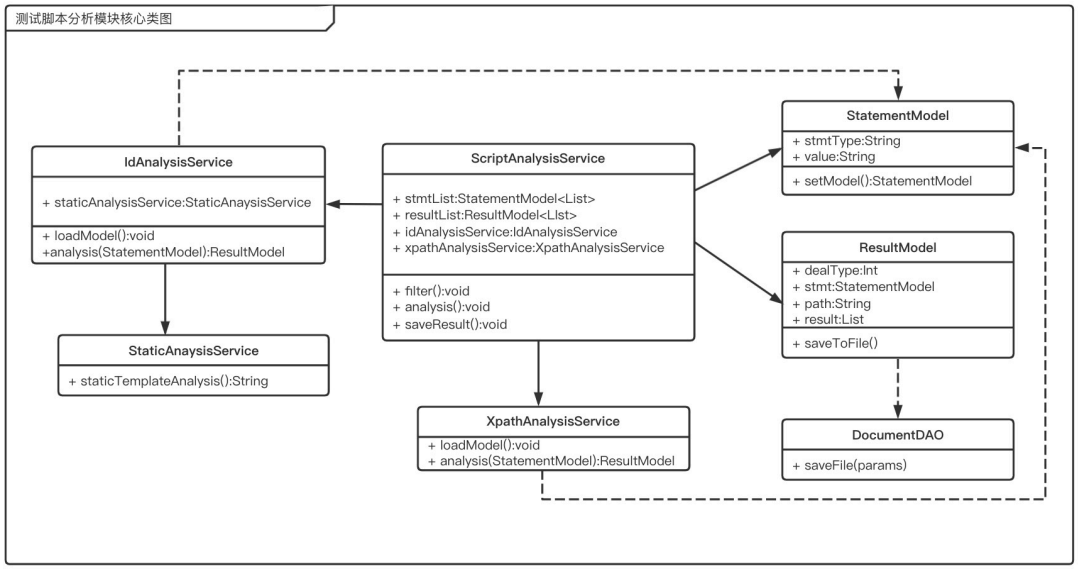

图 6-19: 测试脚本分析模块的核心类图

类型控件的分析；xpathAnalysisService 是 XPath 类控件的分析。方法 logDivision() 是将在自动运行模块的测试设备日志进行预处理以及将其进行分割。方法 filter() 将测试脚本中的测试语句抽象为 StatementModel 类。StatementModel 类主要是存储了当前语句的类型 stmtType、对应的 ID 值或者 XPath 值 value、当前语句的用户交互方法和 sendKeys 值（该值可能为 None）。方法 analysis() 进行分析，分别调用 IdAnalysisService 和 XpathAnalysisService 的服务，并将其转化为 ResultModel 类。IdAnalysisService 是 Id 类控件的分析服务，其中包括了模型的加载 loadModel() 和调用 locateMethodService 类的中对应的定位 UI 响应函数的方法。XPath 类型控件的分析由 XpathAnalysisService 类实现，其功能与 ID 类控件的分析类似。ResultModel 类中存储了分析该语句的类型以及对应的意图结果 result，path 变量存储了当前控件分析之前的信息，这部分信息根据 dealType 有所区别，可能是截图图像也有可能是 UI 响应函数。saveResult() 方法将意图结果进行整合，把 resultList 转换为意图报告文件。

6.4.3 测试脚本分析模块的具体实现

日志切片

我们需要将测试脚本在运行过程中设备产生的日志进行切片。日志切片的规则如3.2.2节中所示，我们通过时间戳的信息来定义两条日志的相关联

关系，将相关联的日志语句划分到一起。变量 source_path 是源日志的所属路径，变量 target_path 是目标日志的所属路径，os.listdir() 用于将工作路径定位到 source_path 路径并返回该路径下的所有文件夹列表。接下来，我们循环该文件夹下的每个测试脚本文件夹，再使用相同的方法得到每个需要处理的日志文件。每个文件的路径为“target_path + apk_name + '/' + log _name + '/'”。变量 index 记录了当前被切片的日志属于第几块，也对应测试脚本中的每行测试语句。变量 souce_file 是待处理的日志文件。变量 content 是当前的日志块信息。变量 cur_time1 代表了当前日志块的最后一条日志的时间戳信息。

```
def handle_log():
   source_path = './log/' # 源日志
   target_path = './log_deal/' # 目标日志
   s_dir = os.listdir(source_path)
   for apk_name in s_dir:
      apk_path = os.listdir(source_path + apk_name)
      os.makedirs(target_path + apk_name + '/')
      for log_name in apk_path:
         os.makedirs(target_path + apk_name + '/' + log_name +
            '/')
         index = 0
         source_file = open(source_path + apk_name + '/' +
            log_name)
         content = ''
         cur_time1 = ''
```

图 6-20: 分割日志片段 1

我们循环获取当前文件的每一条日志 line。通过 get_current_time(line) 方法获取其当前 line 的时间戳。sub() 函数用于判断当前的两条日志的时间间隔秒数。当下一条语句与当前的日志被判定为相关联关系时，我们将当前的 line 加入 content 日志块中；否则，我们会新建一个有 index 信息的文件，并写入之前的日志信息 content，同时 index 自增 1。每条语句 line 处理完毕之后，需要更新当前的最新时间戳信息 cur_time1。最后一步是将最后存储的 content 代码块进行文件的写入与保存。

```
        for line in source_file:
            cur_time2 = get_current_time(line)
            # 当前两条日志为关联日志，应该被分在同一块中
            if sub(cur_time2, cur_time1) <= 2.8:
                content += line
            else:
              # 记录之前的日志信息
                path = target_path + apk_name + '/' + log_name +
                    '/' + log_name + '_' + str(index)
                with open(path, 'w') as target_file:
                    target_file.write(content)
                content = line
                index += 1
            # 更新时间
            cur_time1 = cur_time2
        path = target_path + apk_name + '/' + log_name + '/' +
            log_name + '_' + str(index)
        target_file = open(path, 'w')
        target_file.write(content)
```

图 6-21: 分割日志片段 2

测试脚本信息提取

测试脚本信息提取主要是为了将测试脚本的语句转换成 StatementModel 的列表，方便后续的处理。变量 valid_lines 是一个列表用于存储当前的有效行。打开该测试脚本的 Java 文件后，读取每行代码进行循环。我们会去除该行代码中的空格，然后通过关键词去匹配该行代码。如果代码中有 findElementById 关键词，且该行代码没有被“//”注释，我们判定其为 ID 类型语句。我们使用正则表达式来匹配其相关的值。这里需要提取的值有三个，分别为当前方法的参数（ID 或者 XPath 值）、当前方法进行的用户操作（点击等）还有就是 sendKeys 方法参数（这个参数可能不存在，不存在的话我们将其置为 None）。最后，我们根据提取的值新建一个 StatementModel 对象并将其放在 valid_lines 变量中返回。

```
if ('findElementById' in line) & (re.findall("\)\.(.*?)\(",
   line) != []) & (line.startswith("//") == False):
   # ID类型
   id = re.findall('id/(.*?)"', line)
   action = re.findall("\)\.(.*?)\(", line)
   temp = StatementModel('id',id[0],action[0])
   if (action[0] == "sendKeys"):
      sendKeys = re.findall('sendKeys\("(.*?)"\)', line)
      temp.sendkeys = sendKeys
   else:
      temp.sendkeys = 'None'
   valid_lines.append(temp)
```

图 6-22: 测试脚本信息提取的详细代码 1

如果该行代码中有 findElementByXPath 关键词，且该行代码没有被“//”注释，即为 XPath 类型语句。首先，我们通过 re.findall() 方法来找到 XPath 值；然后用过紧接在后面的方法来判断当前测试语句的用户交互行为，如点击或者输入操作；如果是输入操作，则会有 sendKeys 关键词。我们将这些信息转换为 StatementModel 类。

```
if ('findElementByXPath' in line) & (re.findall("\)\.(.*?)\(",
   line) != []) & (line.startswith("//") == False):
   # XPath类型
   id = re.findall('XPath\("(.*?)"', line)
   action = re.findall("\)\.(.*?)\(", line)
   temp = StatementModel('xpath',id[0],action[0])
   if (action[0] == "sendKeys"):
      sendKeys = re.findall('sendKeys\("(.*?)"\)', line)
      temp.sendkeys = sendKeys
   else:
      temp.sendkeys = 'None'
   valid_lines.append(temp)
```

图 6-23: 测试脚本信息提取的详细代码 2

静态模版匹配 UI 响应函数

```
case R.id.action_pin_recipe_to_widget:
   //response code
return true;
```

图 6-24: 静态模版方法-模版 1 示例

静态模版去匹配 UI 响应函数的方法中一共涉及到五个静态模版，这里我们只给出其中两种模版的详细代码。由于类型 1-4 都是在代码中匹配，我们只给出其中类型 1 的代码，如图6-24所示。我们遍历所有源代码中的所有 Java 文件，并按行读取每个文件内容，当我们找到当前行满足其 ID 值和“case”关键词的时候，我们创建一个文件并开始记录 UI 响应函数。该函数的第一行我们设置为“public void + appname + _ + target + (){”。然后逐行进行写入操作。该 UI 响应函数结束的标志是当前的行有“return”关键词。如果是其他类型的模版匹配，其结束标志为当前内容中左括号“{”、有括号“}”数量相等时。if_finish 是在模版匹配方法中的全局变量，当其值为 1 是代表了模版匹配成功，并且不进行之后类型的模版匹配。

```
def type1(target, root, output):
   appnamelist = root.split("\\")
   appname = appnamelist[len(appnamelist) - 1]
   rootlist = getFileRoot(root)
   for i in range(len(rootlist)):
      if (rootlist[i].endswith(".java")):
         f = open(rootlist[i], 'r', encoding='UTF-8')
         content = f.readlines()
         linenumber = len(content)
         for j in range(linenumber):
            if ((content[j].find(target) != -1) &
                (content[j].find("case") != -1)):
               full_path = create_file(target, output)
               file = open(full_path, 'w')
               while (content[j].find("return") == -1):
                  if (content[j].find("case") != -1):
                     file.write("      public void " + appname +
                        "_" + target + "(){\n")
```

```
                else:
                    if content[j].find("();") != -1:
                        content[j] = content[j].replace(" ", "")
                        content[j] = content[j].replace("();\n",
                            "")
                        dump(content[j], root, file)
                    else:
                        file.write(content[j])
                    j = j + 1
                file.write("      }")
                global if_finish
                if_finish = 1
                file.close()
            j = j + 1
        f.close()
```

图 6-25: 模版匹配中类型 1 的详细代码

类型 5 是在 XML 文件中匹配。我们的目标是在 XML 文件中获取其 UI 响应函数的名称，所以我们会筛选出以“.xml”结尾的文件。我们可以利用该名称和当前 XML 文件名去 Java 文件中搜索，其逻辑与类型 1-4 类似，以上代码已经介绍相关方法，在这里不赘述。在 XML 文件中获取函数名是通过匹配“android:id="@+id/”字段，如果匹配成功的话，我们再去匹配"android:onClick" 字段，并获取该字段的值即为目标函数名。

我们得到目标函数名之后，需要去 Java 文件中寻找当前的 UI 函数响应函数，Java 文件主要需要符合两个规则。规则一是在 UI 资源绑定时绑定了当前的 XML 文件名，规则二是可以找到响应的函数名。

日志定位 UI 响应函数

一行测试设备输出的日志格式如下图所示。首先，我们根据关键词“declaringClass”和“methodName”两个关键词提取其类民和对应的函数签名。在该例中，类名是 com.matiboux.griffith.contactmanager.ListContacts，然后我们提取其包的路径 com.matiboux.griffith.contactmanager 和类名 ListContacts，我们根据这两个信息可以在源代码中找到其对应的 Java 文件和 UI 回调函数。

```
01-12 11:05:07.987 27231 27231 I logger : info start --
declaringClass:com.matiboux.griffith.contactmanager.ListContacts
methodName:public void <init>()
```

这里需要注意的是，有的类为内部类所以输出的日志会带有 $ 符号。我们会对内部类的情况特殊处理。针对函数签名，我们会提取三个部分，分别是函数名、入参、返回参数。

```
def parse_line(one_log):
    one_log = one_log.replace('\n', '')
    sub_class_index = one_log.rfind("declaringClass:")
    method_index = one_log.rfind("methodName:")
    target_class = one_log[sub_class_index+15: method_index-2]
    method = one_log[method_index + 11: len(one_log)]
    if target_class + method in function_set:
        return ''
    else:
        function_set.add(target_class + method)
    # $ 代表内部类的处理
    if target_class.rfind("$") != -1:
        class_file_path = source_path + "/" +
            target_class.split("$")[0].replace('.', '/') + '.java'
        sub_class = target_class.split("$")[-1]
        has_sub_class = True
    else:
        class_file_path = source_path + "/" +
            target_class.replace('.', '/') + '.java'
        sub_class = target_class.split(".")[-1]
    left = method.find('(')
    right = method.find(')')
```

图 6-26: 定位 UI 回调函数的详细代码 1

其中，变量 method 存储了函数名信息；变量 target_class 代表了当前函数所属的文件的类名；function_set 是一个哈希表，存储了已经索引过的函数名，是为了防止日志中的多次调用导致重复对一个函数进行存储。如果当前的

one_log 中含有“$”标志，则代表我们需要针对内部类进行处理，处理的方法是存储一个 sub_class 变量，代表内部类的类名，has_sub_class 为 True 则代表当前需要匹配内部类；如果没有内部类，那么 sub_class 和 target_class 保持一致，区别在于后者后缀为“.java”。

函数类型分为两大类，构造方法和非构造方法，如果当前是一个构造方法，那么函数名是“<init>”而非当前的类名，所以我们需要对匹配字段进行转换；反正，则可以直接使用函数名进行匹配。同时我们会提取当前的返回值。因为 Java 函数存在重载的特点，所以会出现函数名相等的情况，这时候就需要对函数的入参进行提取并转化为列表。

```
# 处理init 构造方法
   if method.find('<init>') >= 0:
      method_name = sub_class
      method_return = 'void'
   else:
      method_2 = method[0:left]
      method_name = method_2.split(" ")[-1]
      method_return = method_2.split(" ")[-2].split(".")[-1]
   has_sub_class = False
   class_file = open(class_file_path)
   left_parenthesis = 0 # 左括号的数量
   content = ''
   if_find_target = False
   first_line = False
   # 处理方法参数
   method_parameters = method[left + 1:right]
   if len(method_parameters) > 0:
      method_parameters = method_parameters.split(",")
```

图 6-27: 定位 UI 回调函数的详细代码 2

我们对文件中的每一行代码进行循环处理，首先需要判断当前行是否含有关键词 method_name 和 method_return，如果这两部分匹配成功，我们才会进行接下来的二次匹配。匹配分为有参数情况和无参数情况两种。如果函数有参数，我们需要提取当前函数的入参并转化为列表和我们之前的目标函数入参

列表进行一一对应。不管是否有参数，匹配成功后我们都将 first_line 标志和 if_find_target 标志置为 True。如果当前 if_find_target 标志为 True，我们可以将当前行加入我们的匹配结果 content 中，并将行号 line_number 自增 1，在这个过程中，我们会记录 content 中的左括号“{”、右括号“}”数量之差。如果当前的左右括号之差等于零代表函数提取结束；还有一种情况也会结束循环，当前行的数量超过了最大行数 MAX_LINE_NUMBER，会退出循环，并补齐右括号“}”的数量。

6.5 运行截图

图 6-28是测试脚本意图生成系统的用户上传界面图。从图中，我们可以看到该系统的使用步骤分为两步，第一步是上传三个文件，其中包括了安卓 APK 包、源代码 zip 包和测试脚本文件。其中，安卓 APK 包需要文件的后缀为“.apk”；源代码文件后缀必须是“.zip”；安卓测试脚本需要是“.java”文件。如果用户上传了不符合要求的文件，会提示文件类型错误并提醒用户重新上传。如果三个文件都符合系统要求并且上传成功，可以进行第二步，点击生成测试脚本意图报告的按钮。

图 6-28: 用户上传界面

图 6-29: 生成意图报告界面

点击生成之后，会跳转到生成意图报告界面，如图6-29所示。该界面的左侧包含了三部分信息，运行环境、基础信息和测试脚本内容。运行环境包括了Appium 版本、Python 版本、TensorFlow 版本和 Keras 版本。基础信息模块主要是展示了当前待测应用名称和脚本名称。测试脚本内容提取了用户上传测试脚本中的有效代码片段，删除掉 Appium 测试脚本中的公共代码。界面的右侧主要展示了运行环境和运行结果。在运行进度部分，进度条会显示当前后台的运行进度，随着以下五个步骤的加载状态来决定。五个步骤是顺序进行，一共有三个状态，正在运行、运行成功和运行失败。如果运行失败的话，无法返回运行结果。运行结果分为了测试脚本意图结果和每行语句具体的意图结果。详细的意图结果由表格展示，分别展示了当前语句的类型、值和意图结构。除此之

外，下方有两个按钮，分别为下载意图报告按钮和下载所有文件按钮。下载的所有文件中不仅包括了用户上传的三个文件和意图结果文件，还包含了中间过程产生的媒体文件（屏幕截图、控件图像、XML 文件）和 ID 控件提取的 UI 响应函数可供用户查看。

6.6 本章小结

本章介绍了测试脚本意图生成系统的模块划分以及每个模块中核心逻辑代码的实现。第一个模块是文件模块，主要负责整个系统的文件存储和用户上传下载文件的接口。第二个模块是 APK 预处理模块，主要是对 APK 进行校验、插桩和签名，并给出了基于 soot 框架对 APK 插桩的核心代码实现。第三个模块是自动化运行模块，在该模块中，我们可以自动化运行用户上传的 Appium 测试脚本并存储相关的日志文件和媒体文件信息。我们给出了封装 Appium 项目以及自动化运行的代码；除此之外，我们还介绍了如何保存日志文件和利用 Appium Driver 提供的接口记录控件图像。最后一个模块是测试脚本分析模块，我们在这个模块中利用 XPath 类型控件和 ID 类型控件的区别来分析得到其意图信息并整合成一个完整的意图报告。在这个过程中，需要日志切片的算法和定位 UI 响应函数的算法，我们在本章中已经详细介绍。

第七章　总结与展望

在移动互联网的高速发展之下，自动化测试脚本的堆积以及注释的缺乏导致开发人员对于大量脚本难以管理和复用，本文提出了一种新颖的结合动静态分析的生成测试脚本意图报告的方法，首次解决了测试脚本缺乏注释的问题。

我们从 Appium 测试脚本的绑定控件方式出发，将测试语句分为了 ID 和 XPath 两种类型。ID 是开发人员在开发一个控件的时候自定义的全局唯一标识；XPath 是在安卓的 XML 布局文件中定位控件路径的语言。针对 ID 控件，我们使用了静态模版和动态控制流图两种方法去获取 ID 控件的 UI 回调函数，我们将代码输入 code2seq 模型中获取其意图结果；针对 XPath 控件，我们使用 Appium Driver 的接口保存运行时的控件截图和 XML 文件，文字类控件我们从 XML 文件中提取其语义信息，图片类控件我们将其输入编码器解码器模型中获取其语义信息。最后，我们整合这两部分得到意图报告结果。

实验部分，我们计算了两部分的数据。第一部分是测试脚本意图报告方法的映射率。XPath 分析和 ID 分析在各自控件类型的图像/代码映射率为 96.34% 和 69.44%，测试意图结果的映射率为 77.65%。第二部分是自然语言描述的 BLEU、CIDEr、ROUGE-L 和 METEOR 四个评分指标。结果显示，我们的方法在各个方面的分数均优于原有的 code2seq 模型。

基于实验的有效性，我们将其封装成一个测试脚本的意图报告生成系统。我们从用户角度分析了该系统的需求以及给出了具体用例，基于“4+1”视图模型给出了系统的框架设计。我们介绍了每个模块的特点，并给出了核心代码的实现。

在本文提出的方法中，也有一些可以改进的地方。首先，code2seq 模型目前对超过百行的函数无法进行代码意图提取输出结果，可以针对代码语义模型进行一些优化；其次，我们针对于不同类型的测试语句进行分析后生成最终的意图报告结果时，我们目前简单的使用了意图拼接的方法，可以考虑去重等方法提高最终的意图结果准确度；最后，我们可以建立检索策略，针对海量脚本和其意图信息进行管理并检索定位其源代码。

参考文献

[1] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] https://www.statista.com/statistics/232786/forecast-of-andrioid-users-in-the-us/

[3] statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/

[4] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, “GUILeak: Tracing privacy policy claims on user input data for android applications,” in International Conference on Software Engineering (ICSE), 2018.

[5] Xia X, Bao L, Lo D, et al. Measuring program comprehension: A large-scale field study with professionals[J]. IEEE Transactions on Software Engineering, 2017, 44(10): 951-976.

[6] Beltramelli T. pix2code: Generating code from a graphical user interface screenshot[C]//Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems. 2018: 1-6.

[7] Chen S, Fan L, Su T, et al. Automated cross-platform GUI code generation for mobile apps[C]//2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile). IEEE, 2019: 13-16.

[8] Xiao X, Wang X, Cao Z, et al. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 257-268.

[9] 周培君, 吴军华. 组合源码结构和语义的代码注释自动生成方法 [J]. 小型微型计算机系统,2021,42(12):2501-2505.

[10] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao,Z. Liu, F. Xu et al., “Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps,”in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 2421–2436.

[11] 杨萍, 舒辉, 康绯, 卜文娟, 黄宇垚. 一种基于语义分析的恶意代码攻击图生成方法 [J]. 计算机科学,2021,48(S1):448-458+463.

[12] 蔡瑞初, 张盛强, 许柏炎. 基于结构感知混合编码模型的代码注释生成方法 [J/OL]. 计算机工程:1-11[2022-03-27].DOI:10.19678/j.issn.1000-3428.0063592.

[13] 宋晓涛, 孙海龙. 基于神经网络的自动源代码摘要技术综述 [J]. 软件学报,2022,33(01):55-77.DOI:10.13328/j.cnki.jos.006337.

[14] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu,Q. Shan, J. Nichols, J. Wu, C. Fleizach et al., “Screen recognition:Creating accessibility metadata for mobile applications from pixels,”arXiv preprint arXiv:2101.04893, 2021.

[15] Nguyen T A, Csallner C. Reverse engineering mobile application user interfaces with remaui (t)[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015: 248-259.

[16] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu,Q. Shan, J. Nichols, J. Wu, C. Fleizach et al., “Screen recognition:Creating accessibility metadata for mobile applications from pixels,”arXiv preprint arXiv:2101.04893, 2021.

[17] 彭斌, 李征, 刘勇, 吴永豪. 基于卷积神经网络的代码注释自动生成方法 [J]. 计算机科学,2021,48(12):117-124.

[18] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K VijayShanker. 2010. Towards automatically generating summary comments for java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 43–52.

[19] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock,and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, 23 – 32.

[20] Sai Zhang, Cheng Zhang, and Michael D Ernst. 2011. Automated documentation inference to explain failed tests. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society,63 – 72.

[21] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 33 – 42.

[22] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code.In Reverse Engineering (WCRE), 2010 17th Working Conference on. IEEE, 35 – 44.

[23] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013.Evaluating source code summarization techniques: Replication and expansion. In Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE,13 – 22.

[24] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, 380 – 389.

[25] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 562 – 567.

[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016.Summarizing Source Code using a Neural Attention Model.In ACL (1).

[27] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In International Conference on Machine Learning. 2091–2100.

[28] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, 2018, pp. 200–20 010.

[29] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1:Long Papers), 2016, pp. 2073–2083.

[30] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in International Conference on Learning Representations, 2019.

[31] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques, Addison Wesley, 1986.

[32] D.J. Tan, T.-W. Chua, V.L. Thing, et al., Securing android: a survey, taxonomy,and challenges, ACM Comput. Surv. 47 (4) (2015) 58.

[33] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., ... Traon, L. (2017). Static analysis of android apps: A systematic literature review. Information and Software Technology, 88, 67-95.

[34] Lam, P., Bodden, E., Lhoták, O., Hendren, L. (2011, October). The Soot framework for Java program analysis: a retrospective. In Cetus Users and Compiler Infastructure Workshop (CETUS 2011) (Vol. 15, No. 35).

[35] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V. (2010). Soot: A Java bytecode optimization framework. In CASCON First Decade High Impact Papers (pp. 214-224).

[36] Mohanty, H.; Mohanty, J.; Balakrishnan, A. Trends in Software Testing; Springer: Berlin/Heidelberg,Germany, 2017.

[37] Mahmood, R.; Esfahani, N.; Kacem, T.; Mirzaei, N.; Malek, S.; Stavrou, A. A whitebox approach for automated security testing of Android applications on the cloud. In Proceedings of the 7th International Workshop on Automation of Software Test, Zurich, Switzerland, 2 – 3 June 2012; pp. 22 – 28.

[38] Edalat, E.; Sadeghiyan, B.; Ghassemi, F. ConsiDroid: A Concolic-based Tool for Detecting SQL Injection Vulnerability in Android Apps. arXiv 2018, arXiv:1811.10448.

[39] McMinn, P. Search-Based Software Testing: Past, Present and Future. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21 – 25 March 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 153 – 163.

[40] Developers, A. Android Debug Bridge (adb). Available online:https://developer.android.com/studio/command-line/adb (accessed on 20 November 2018).

[41] J. C. Huang, ”Program Instrumentation and Software Testing,” in Computer, vol. 11, no. 4, pp. 25-32, April 1978, doi: 10.1109/C-M.1978.218134.

[42] Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Hol- ¨ger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks.In Advances in Neural Information Processing Systems, pages 3104 – 3112, 2014.

[44] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015, pages 1412 – 1421, 2015.

[45] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In Proceedings of the 2015 10th

Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015a. ACM. ISBN 978- 1-4503-3675-8. doi:10.1145/2786805.2786849.

[46] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, pages 2091–2100,2016.

[47] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015, pages 1412–1421, 2015.

[48] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang, "Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning." Association for Computing Machinery, 2020, p. 322–334.

[49] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[50] Wang S, Li X, Duan S, et al. Modeling and Simulation of Radar Klystron Based on the System Vue[C] // 2019 International Conference on Meteorology Observations (ICMO). 2019 : 1 –4.

[51] Wohlgethan E. SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js[D]. Hochschule für Angewandte Wissenschaften Hamburg, 2018.

[52] Saks E. JavaScript Frameworks: Angular vs React vs Vue[J]. 2019.

[53] Papineni K, Roukos S, Ward T, et al. Bleu: a method for automatic evaluation of machine translation[C]//Proceedings of the 40th annual meeting of the Association for Computational Linguistics. 2002: 311-318.

[54] Vedantam R, Lawrence Zitnick C, Parikh D. Cider: Consensus-based image description evaluation[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 4566-4575.

[55] Lin C Y. Rouge: A package for automatic evaluation of summaries[C]//Text summarization branches out. 2004: 74-81.

[56] Banerjee S, Lavie A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments[C]//Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization. 2005: 65-72.

[57] Qassim H, Verma A, Feinzimer D. Compressed residual-VGG16 CNN model for big data places image recognition[C]//2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC). IEEE, 2018: 169-175.

[58] Hans M. Appium Essentials[M]. Packt Publishing Ltd, 2015.

[59] Forcier J, Bissex P, Chun W J. Python web development with Django[M]. Addison-Wesley Professional, 2008.

[60] Alon U, Zilberstein M, Levy O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1-29

致　　谢

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

_____年_____月_____日

论文题名	基于动静态分析的安卓测试意图生成技术				
研究生学号	MF20320218	所在院系	软件学院	学位年度	2022
论文级别	☐ 硕士　☐ 硕士专业学位 ☐ 博士　☐ 博士专业学位 （请在方框内画勾）				
作者 Email	zhang_jing@smail.nju.edu.cn				
导师姓名	陈振宇 教授，房春荣 助理研究员				

论文涉密情况：

☐ 不保密

☐ 保密，保密期 (_____年_____月_____日 至 _____年_____月_____日)

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。