



# 南京大學

## 本科畢業論文

院 系 軟件學院

專 業 軟件工程

題 目 安卓不可見控件内存泄漏的自动检测

年 级 2016 级 学 号 161250136

学生姓名 王冬杨

指导老师 马骏 职 称 副教授

提交日期 2020 年 5 月 29 日



# 南京大学本科生毕业论文(设计、作品)中文摘要

题目：安卓不可见控件内存泄漏的自动检测

院系：软件学院

专业：软件工程

本科生姓名：王冬杨

指导老师（姓名、职称）：马骏副教授

摘要：

在安卓应用中，服务和广播得到了广泛应用，在绝大多数的商店应用中，都可以找到服务和广播的应用，开发者可以使用这些组件方便地为应用提供诸如下载，数据更新，跨应用通信等不可缺少的重要功能。但是，服务和广播的广泛使用也具有一定的安全问题：由于服务和广播的开发具有碎片化以及不可见的特性，以及开发者经常忽视这些不可见控件的生命周期管理，因此在此类组件中发生内存泄漏的几率很高。而正由于这些组件的不可见的特性，使得内存泄漏问题的发现，解决的速度通常都比较慢，产生的影响的时间会比较长。

安卓官方也在逐渐重视这些不可见控件的生命周期管理：近期安卓系统在版本更新时，着重推出了“电池管理策略”，并在后续的版本更新中，对该策略进行了进一步的完善和升级。“电池管理策略”的主要目的是通过重点对后台不可见组件的行为进行了规范和约束，尽可能的节约电量消耗。这一策略会尽可能地促使开发者对不可见组件进行有效的生命周期控制，以减少内存泄漏发生的可能性，也会使开发者加大对不可见控件的测试力度，提前发现和解决更多的问题。但另一方面，这项更新会使得服务和广播的注册、启动方式都有了一定变化，这将会影响一些应用在新版本上的兼容性，同时也加重了碎片化现象。

在本文中，讨论并实现了一套针对安卓不可见控件的内存泄漏问题进行自动化检测诊断的工具，该工具可以在最新的安卓版本上进行检测，使用此自动检测工具可以帮助开发者快速方便的诊断应用中的服务和广播是否存在内存泄漏的问题，生成内存泄漏控件清单，以辅助开发者进行测试和调试。

同时本文在应用市场中下载了若干真实应用，使用本文开发的自动测试工具进行实验，发现在安卓应用中，组件的碎片化和内存泄漏问题较为普遍的存在。

在具体的实现上，该自动检测工具借助了开源工具 **apktool**<sup>[1]</sup> 进行 apk 的反编译以得到应用的控件清单、**mat**<sup>[2]</sup> 进行内存对象的分析以寻找内存泄漏的实

例；使用安卓 sdk 中的 **adb** 和 **emulator** 等工具完成测试的流程；并使用 **socket** 来完成工作站与安卓模拟器的通信。

**关键词：** 安卓系统；内存泄漏；安卓服务；安卓广播

# 目 录

目 录 .....	III
<b>1 绪论 .....</b>	<b>1</b>
1.1 研究背景 .....	1
1.2 相关工作 .....	2
1.3 主要挑战 .....	2
1.4 现有研究的不足 .....	3
1.5 本文贡献 .....	3
1.6 本文主要工作 .....	3
1.7 本文结构 .....	3
<b>2 背景 .....</b>	<b>5</b>
2.1 安卓系统 .....	5
2.1.1 发展历史 .....	5
2.1.2 主要组件 .....	5
2.2 安卓服务 .....	6
2.2.1 服务的启动方式 .....	8
2.2.2 服务的生命周期 .....	8
2.2.3 服务的注册方式 .....	8
2.2.4 服务的内存泄漏 .....	9
2.3 安卓广播接收器 .....	10
2.3.1 广播接收器的启动方式 .....	11
2.3.2 广播接收器的生命周期 .....	11
2.3.3 广播接收器的注册方式 .....	11
2.3.4 广播接收器的内存泄漏 .....	12
2.4 电池管理限制 .....	13
2.4.1 简介 .....	13
2.4.2 对实验方法的影响 .....	14
2.4.3 电池管理限制白名单 .....	14

2.5 小结 .....	17
<b>3 技术方案 .....</b>	<b>19</b>
3.1 系统架构 .....	19
3.1.1 基于测试效率的考量 .....	19
3.1.2 具体设计 .....	20
3.2 Apk 文件解析器 .....	20
3.3 测试驱动应用 .....	21
3.4 测试主流程 .....	22
3.5 内存分析器 .....	23
3.6 其他实现细节 .....	24
3.7 小结 .....	25
<b>4 实验 .....</b>	<b>27</b>
4.1 仿真测试 .....	27
4.1.1 仿真应用 .....	27
4.1.2 实验结果 .....	27
4.2 真实测试 .....	28
4.2.1 实验结果 .....	29
4.2.2 应用崩溃主要原因分析 .....	30
4.3 结果分析 .....	31
4.3.1 数据分析 .....	31
4.3.2 实验数据局限性 .....	32
4.4 小结 .....	32
<b>5 结论 .....</b>	<b>33</b>
<b>参考文献 .....</b>	<b>35</b>
<b>致    谢 .....</b>	<b>37</b>

# 第一章 绪论

本章将介绍一些与本文研究相关研究背景，简要的总结前人的相关工作，并指出前人工作的不足之处。进而说明本文的主要贡献，以及本文在实现时面临的几个主要的挑战，最后说明本文的结构。

## 1.1 研究背景

在安卓应用中，服务（**Services**）和广播（**Broadcast**）得到了广泛的使用。服务可以在安卓应用的后台保持长期运行，提供诸如下载、数据更新等重要功能。然而，正因为服务长期运行于后台的特点，使其往往容易产生异常（**Errors**）。如果服务的编写人员缺少警惕性，服务中出现的错误（**Bug**）可能会导致诸多问题，严重者可能引起应用崩溃，甚至系统死机；广播是一种常被用来进行跨应用的通信的通信手段，应用在使用广播进行与系统或者与其他应用进行通讯时，应用需要编写广播接收器（**Broadcast Receiver**）。负责运行广播接收器的时应用的主线程种，因此在广播接收器中不适合进行耗时操作，通常会在广播接收器中启动服务来进行后续的处理，因此广播接收器也可能通过服务或者自身导致内存泄漏。

安卓应用中的内存泄露指资源（内存对象、句柄、服务等）将不再被使用，但却无法被安卓系统的垃圾回收器（**Garbage Collector**）所回收，同时也是服务中的一大类常见问题。服务如果出现内存泄露，将会导致内存使用量意外大幅度增加，进而使得系统效率降低，严重影响用户体验。

有一类服务被称为**公开服务**，即指定了“**exported:true**”属性的服务。其他应用在满足一定条件时（满足权限要求等）可以启动应用的公开服务，因此内存泄露的问题将会变得更加复杂。

由于在目前的安卓版本（安卓 10）中，安卓操作系统中的“电池优化策略”<sup>[3]</sup>会禁止跨应用启动后台服务<sup>[4]</sup>，而这种测试方式在早先的安卓版本中（安卓 7 及更早）是可行的。因此在现有的最新安卓系统（安卓 10）中，组件的内存泄漏检测方法与先前的方法<sup>[5]</sup>将会有所差别。

## 1.2 相关工作

Erika 等人在早先的安卓版本中，编写了一个检测跨应用通信安全问题的工具 **Com Droid**<sup>[6]</sup>，文中阐述的方法对于跨应用测试具有借鉴意义。

刘洁瑞等人在 2016 年，针对安卓系统资源（如相机等）的内存泄漏问题，构造了 **35** 组标准测试集并使用 **ResLeakBench**<sup>[7]</sup> 进行了较为广泛的测试，该文所研究的系统资源泄露的场景，对本文研究的组件内存泄漏的场景有一定的借鉴意义。

在早先的安卓版本中，跨应用启动服务这一行为是被允许的，南京大学的马骏等人实现过一个公开服务（**Exported Services**）内存泄漏的检测工具 **LES Droid**<sup>[5]</sup>，文中采用的方式分为四步：

1. 使用 **apktool** 反编译工具<sup>[1]</sup> 对被测试应用进行反编译，获取被测试应用的安卓组件清单（**AndroidManifest.xml**）文件，解析获取应用中所有的公开服务的包名和类名。
2. 将测试驱动应用、被测试应用通过 **adb** 安装到模拟器中，启动测试驱动应用。
3. 测试驱动应用重复启动、关闭被测试的服务，在满足一定测试强度之后，导出被测试应用的堆镜像文件（**.hprof files**）。
4. 基于 **MAT** 内存分析库<sup>[2]</sup> 编写堆镜像文件的分析工具，自动检测内存泄漏并统计泄露的入口等。

文中的数据指出：在 **41537** 个被测试应用中，共在其中 **28662**（**69%**）个应用中检测出 **74831** 个服务，其中 **3934**（**13.7%**）个应用拥有公开服务。经过去重、安装测试以及应用商店评分筛选，有 **375** 个实际测试应用，最终通过不同的测试配置，最终检测到在 **18.7%** 的应用中有 **16.8%** 的服务存在内存泄漏问题。

## 1.3 主要挑战

本文面临的挑战主要有：

1. 如何驱动测试流程自动化进行，对被测试组件进行自动化的创建和销毁。



2. 寻找办法能解决“电池管理限制”对测试流程和方法带来的多种限制。
3. 如何对内存文件进行自动化进行分析，找出内存泄漏的组件实例。

## 1.4 现有研究的不足

在测试的方法上：在安卓系统的升级中，着重推出了“电池管理限制”，而这一限制使得原有的测试方式在最新的安卓版本中无法直接使用，需要针对“电池管理限制”进行相应的修改。

在测试对象上：已有的研究只针对安卓的系统资源，以及安卓的公开服务进行了内存泄漏的测试，对于其他不可见组件（广播接收器等）并没有进行系统的测试。

## 1.5 本文贡献

本文将会在最新的安卓版本（安卓 10）上对服务和广播进行内存泄漏的检测和测试。探索了一种解除电源管理限制的方法。分析实验数据，评价“电池管理限制”带来的影响。

## 1.6 本文主要工作

本文旨在探索一套适用于安卓 10 版本的公开服务和静态注册广播接收器的内存泄漏检测方法。主要工作如下：

1. 对原有的测试方法进行修改，使得该方法能够适用于安卓 10 版本。
2. 在桩应用上进行测试，验证自动检测工具的可行性和正确性。
3. 在应用商店中下载真实应用，进行自动化测试分析实验结果。

## 1.7 本文结构

本文的各章节组织结构如下：第一章**绪论 1**简要的说明了本文的研究对象，和采用的测试方法，并总结了相关论文的工作等。第二章**背景 2**详细介绍了安卓广播和服务的注册、启动、生命周期等基础知识，并且对安卓组件内存泄漏的原理做了举例，还简要介绍了新版本安卓中的“电池管理限制”。第三

章**技术方案 3**中具体地解释了自动化检测工具的所有流程和实现细节，并通过追溯安卓源码发现了后台启动服务的方法（后台启动服务默认为不允许的行为）。第四章**实验 4**进行了仿真测试与真实测试，仿真测试目的在于验证自动检测工具的正确性和可行性，真实测试从安卓应用市场中下载了真实应用，使用本文的自动检测工具进行了测试。第五章**结论 5**总结了本文的主要工作，并通过实验数据发现在新版本的安卓系统中，应用的组件内存泄漏问题依然较为广泛的存在。

## 第二章 背景

本章将主要介绍与本文研究相关的背景知识：安卓系统，安卓服务，安卓广播接收器，以及电池管理限制。具体的将会介绍安卓系统的历史和四大组件，两种安卓不可见控件的启动方式、注册方式、生命周期、内存泄漏的成因等，以及电池管理限制对安卓组件带来的新变化。

首先将简单介绍安卓操作系统；接下来重点本文的测试对象：安卓服务以及安卓广播接收；最后介绍电池管理限制。

### 2.1 安卓系统

#### 2.1.1 发展历史

安卓是一个基于 **Linux kernel** 以及众多开源软件的移动设备操作系统，是一个为触屏设备定制的操作系统。

安卓系统诞生于 2013 年，由安迪鲁宾等人开发制作，于 2015 年被谷歌 (Google) 公司收购，现在由谷歌 (Google) 公司进行持续的更新迭代和开发。

现在安卓已经以 **Apache 免费开放源代码许可** 方式进行了开源授权，这一措施使得厂商可以让自己的设备搭载定制化的安卓系统，进而大大加速了安卓系统在移动设备上的普及。目前安卓设备已经超越微软 **Windows** 操作系统，成为全球第一的操作系统。

#### 2.1.2 主要组件

安卓应用中有四大基本组件：

1. **Activity** Activity（即活动）是安卓应用中的图形化界面，在安卓应用中几乎所有与用户的交互都是通过它来完成，是安卓应用中使用最多的一种重要组件，活动之间可以实现切换和通信，从而给用户带来更好的使用体验。

在应用中，系统会有一个活动栈持有所有活动的引用，当新的活动被创建时，将会进入栈的顶部，而当一个活动被销毁时，将会被从栈中弹出，借

助活动栈，可以实现页面跳转，回退等操作。

活动共有四种状态：**运行态**，**暂停态**，**停止态**，**销毁**，开发者可以通过生命周期函数对活动的状态进行有效的管理。

2. **Service** Service（即服务）是安卓应用的后台组件，服务并没有可视化的界面，通常不被用户所感受到，但服务可以在后台为应用提供重要功能，例如数据更新，播放音乐等。

由于活动负责响应用户的交互，因此需要保证活动处于高响应优先级状态，因此例如网络下载，文件读写等耗时的操作需要在不影响主线程的其他线程中完成，这就是服务的存在意义。

服务的状态有两种：**启动状态**和**绑定状态**

本文接下来将在 2.2 章节中对服务进行详细的讲述。

3. **Broadcast** Broadcast（即广播）也是安卓应用的后台组件，广播的存在主要目的是为跨应用通信和系统与应用通信提供了一种通用的方法。应用可以主动发出广播，也可以被动接收广播，也可以接收来自系统的广播。为了使用广播，应用需要注册广播接收器，在接收到广播时进行一系列的响应动作，这就完成了跨应用通信，这使得用户安装的众多应用之间，可以产生联动交互，提升了用户体验。

本文接下来将在 2.3 章节对广播进行详细的讲述。

4. **ContentProvider** ContentProvider（即内容提供者）是安卓系统中专门为不同应用之间提供数据通信的组件，是对**标准查询语句**的高层封装。应用可以设置部分数据与其他应用共享，这样其他应用通过内容提供者就可以获取到这一部分共享数据。需要注意的是，在使用内容提供者时，也需要对生命周期进行有效的管理，否则容易导致内存泄漏问题，过度占用系统的后台资源。

## 2.2 安卓服务

在安卓系统中，每个安卓应用都对应着一个主线程，这个主线程将负责处理与界面相关的计算和渲染、负责响应用户的交互以及负责响应生命周期事件等。这些任务对主线程产生了快速响应的要求，如果主线程响应过慢，会导致糟糕的用户体验。换言之，在主线程中只能进行不耗时（几毫秒级别）的计算和操作，而任何耗时较长的计算和操作都必须在主线程之外的单独的后台线程之上来完成。这样可以使得用户在积极的与应用交互时，应用在后台可以积极

的响应和运行。

然而，后台任务需要运行，则不可避免地会占用（消耗）安卓设备的系统资源：例如占用若干内存容量和消耗一定电池电量等。如果后台任务操作不当（如占用大量内存导致系统卡顿，长期进行密集计算导致电池电量下降变快），也可能导致用户体验下降；同时如果安卓服务的开发人员在开发时引入了**程序缺陷**，使得服务在运行时产生了**内存泄漏**，将会使得前面的现象变得更加严重。因此随着安卓系统的更新，对服务组件也进行了更多的限制，以尽可能地对服务进行优化。

- **安卓 6.0（API 级别 23）** 此版本的安卓系统中，新增了重要的**低电量模式**。在关闭安卓设备的屏幕，而且安卓设备的位置信息没有发生显著变化的情境下，系统认为设备处于闲置状态，低电量模式将被启动，对正在运行中的应用行为进行适当的限制，以减少电量的消耗。例如：限制网络访问，限制同步等。即当开启低电量模式时，服务的行为会受到一定程度的限制。
- **安卓 7.0（API 级别 24）** 此版本的安卓系统禁止应用注册隐式广播，而是需要应用显式地进行广播的注册，这样应用的开发者将会更多关注广播接收器的生命周期，同时安卓系统也便于对广播进行管理。另外一个变动在于将上个版本中推出的低耗电模式推广成为一个随时进行的系统耗电优化：即当屏幕一段时间处于未唤醒状态且未接入电源时，低耗电模式就会自动启动，而无需手动开启。即只要设备处于闲置状态时，服务的行为随时都会受到限制。
- **安卓 8.0（API 级别 26）** 此版本的安卓系统禁止处于后台的应用启动新的服务，后台应用若要启动服务，必须显式经过用户许可，启动前台服务。这项限制对跨应用测试的研究带来了不便（跨应用测试一般由测试人员编写的测试驱动应用去启动其他（后台）应用的组件）。
- **安卓 9.0（API 级别 28）** 新增了**应用待机存储分区**。系统将会根据应用的使用模式，给应用动态分配优先级，系统在分配系统资源时，优先级将会是一个重要的指标。

本节接下来将会详细介绍服务的启动方式，生命周期，注册方式，以及内存泄漏的成因。

### 2.2.1 服务的启动方式

安卓应用中的服务可以通过两种方式启动<sup>[8]</sup>:

**start 模式** 通过此方式启动的服务，必须为显式方式：即构造出特定的 **Intent** 对象，接着调用 **startService()** API 来启动目标服务。

**bind 模式** 通过此方式绑定的服务，通过调用 **bindService()** API 将目标服务与特定的系统组件绑定。被绑定的服务提供接口供其他组件与之交互。

一个服务可以同时通过以上两种方式启动。

### 2.2.2 服务的生命周期

服务的生命周期根据启动方式不同分为两种<sup>[8]</sup>:

**start 模式** 通过 **startService()** API 启动的服务，在此之后服务将始终处于运行状态，直到调用 **stopSelf()** 方法将自身停止运行，或者通过构造特定的 **Intent** 实例，调用系统的 **stopService(Intent)** API 将服务停止运行。

停止运行的服务将会被 **GC(Garbage Collector)** 回收。

**bind 模式** 通过 **bindService()** API 启动的服务，在启动后，其他组件可以通过 **IBinder** 接口与之进行交互，直到其他组件调用 **unbindService()** API 解除绑定。

同一个服务可以同时绑定到多个组件之上，由于服务是单例模式（即同一个服务只能有一个实例在系统中运行），因此若要使得该服务停止运行，并被 **GC** 回收，需要满足的条件是：所有组件都解除了绑定。

每个安卓应用都关联一个 **ActivityThread** 实例，负责调度和执行该应用的各种组件。**ActivityThread** 有一个 **ArrayMap** 类型的成员变量 **mServices**，其中保存了所有仍在运行的服务的引用。一旦某个服务的实例停止运行，其引用将会从 **mServices** 中删除，之后该实例会被 **GC** 回收。

### 2.2.3 服务的注册方式

通常，每个服务都要在 **AndroidManifest.xml** 中注册一个 **<service>** 标签（参考 Listing. 1 中的样例）。同时服务可以通过设置 “**android:exported**” 属性来指定该服务是否将被导出。若希望服务可以被其他应用启动，则指定 **android:exported = “true”**，反之亦然。

**Listing 1** 服务的注册方式

```

1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6      <application ...>
7          ...
8          <service android:name=".Service1"
9              android:enabled="true"
10             android:exported="true">
11          </service>
12          <service
13              android:name = ".Service2"
14              android:enabled = "true"
15              android:exported = "false">
16              <intent-filter>
17                  <category android:name = "cat1"/>
18                  <action android:name = "act2"/>
19              </intent-filter>
20          </service>
21          <service android:name = ".Service3"
22              android:enabled = "true"
23              android:permission = "Permission1">
24              <intent-filter>
25                  <action android:name = "act3"/>
26                  <category android:name = "cat2"/>
27                  <data android:scheme = "Scheme1"/>
28                  <data android:scheme = "Scheme2"/>
29              </intent-filter>
30          </service>
31      </application>
32  </manifest>

```

## 2.2.4 服务的内存泄漏

通常，一个服务的实例不再被使用时应该被 **GC(Garbage Collector)** 回收，并释放资源。然而在某些情况下，一个停止运行的服务可能会意外的被引用，从而使得 **GC** 无法将其回收并释放资源，这样就造成了服务的内存泄漏。

例如在游戏 *com.siendas.games2048* 中，就出现了原理如图（见 **Listing. 2**）的内存泄漏。具体导致内存泄露的原理为：在 **LeakedService** 的实例被构造的时候，**onCreate()** 生命周期函数将会被调用，在该方法中延迟 **1000ms** 启动了一个匿名计时器，该计时器将以 **3000ms** 为周期打印调试信息，可以看到在 **TimerTask** 类中持有了 **LeakedService** 的引用。而在该服务停止运行时，其 **onDestroy()** 生命周期函数将会被调用，而在该方法中并没有对该匿名计时器进行销毁。因此在该服务停止运行后，将会一直存在一个匿名计时器持有该服务

## Listing 2 服务的内存泄漏

```

1  public class LeakedService extends Service{
2      private static final String TAG = "LeakedService";
3      //服务实例的启动方法
4      public void onCreate() {
5          ...
6          new Timer().scheduleAtFixedRate(()->{
7              Log.d(TAG, LeakedService.this.getPackageName() +
8                  "\n.LeakedService is running!");
9              },1000L,3000L);
10         }
11         //服务实例的销毁方法
12         public void onDestroy() {
13             ...
14         }
15     }

```

的引用，从而使得垃圾回收器（GC）无法回收该服务资源，从而形成了该服务实例的内存泄漏。

## 2.3 安卓广播接收器

安卓系统中，可以在安卓应用之间、以及安卓应用与安卓系统之间实现通讯，这种通讯的手段被称为广播<sup>[9]</sup>，广播组件的设计采用了发布-订阅的设计模式。在特定的事件发生时，与之相关的广播将会被发送，例如：安卓系统将在各种系统事件（如插拔耳机，插拔电源等）发生时进行对应的系统广播的发送；再如：一个安卓应用可以发送自定义的广播来通知其他的安卓应用，通过广播来进行跨应用的通信。

在使用广播时，应用需要注册接收特定的广播（称之为订阅）的广播接收器。在相应的广播发出时，将由安卓系统负责将广播发送给订阅此种广播的应用。因此在安卓应用开发者的角度，只需要编写广播接收器，并订阅特定的广播，在接收到广播时做出相应的响应动作即可。

一般而言，在跨应用通信时，广播是被广泛使用的一种方便的手段。但是在使用广播时需要格外小心，如果应用不正确的使用广播，可能会导致系统运行变迟钝。如果开发人员在编写广播接收器的时候引入程序缺陷，将会导致更严重的问题（如应用崩溃等）。

本节接下来将会介绍广播接收器的启动方式，生命周期，注册方式，以及内存泄漏的成因。



### 2.3.1 广播接收器的启动方式

安卓应用中的广播接收器亦有两种方式启动<sup>[10]</sup>:

**清单声明的接收器** 通过在 **AndroidManifest.xml** 中添加 **<receiver>** 标签类型。在应用安装时，将会由软件包管理器在系统中注册此类接收器。此后，此广播接收器会变为应用的一种独立的启动方式，也就是说：即使应用未处于运行状态，系统在接到其订阅的广播后，也可启动应用并将这条广播发送给应用。在收到特定广播后，系统会创建一个专门的 **BroadcastReceiver** 实例，并将广播发送给此实例，由它来完成收到广播后的自定义行为。接收器实例将在 **onReceive(Context, Intent)** 方法种获取所订阅的广播内容，并完成相关操作，在从此方法种返回之后，系统便会对接收器进行销毁和资源回收。

**上下文注册的接收器** 通过在代码中构造出 **BroadcastReceiver** 实例，以及 **IntentFilter** 实例来指定该接收器所订阅的广播类型，调用 **registerReceiver(BroadcastReceiver, IntentFilter)** API 在安卓系统中完成此接收器的注册。通过这种方式注册的接收器的生命周期只与注册的上下文有关，只要上下文有效，则接收器可以继续工作，持续接收广播。如果要停止接收广播，需要使用 **unregisterReceiver(BroadcastReceiver)** API 来注销此广播接收器，之后系统将会对接收器进行销毁和资源回收。

### 2.3.2 广播接收器的生命周期

广播接收器的生命周期根据启动方式不同亦分为两种<sup>[10]</sup>:

**清单声明的接收器** 静态注册的接收器生命周期不限于 **Activity** 甚至整个应用。即使应用并不在运行，接收器也可以接收到订阅的广播。接收器将会在 **onReceive()** 方法结束后被销毁。

**上下文注册的接收器** 上下文注册的接收器，其生命周期仅限于注册的上下文，例如在 **Activity** 上下文注册的接收器，在整个 **Activity** 存活期间可以持续接收广播；在应用上下文中注册的接收器，则会在整个应用运行期间都可以接收广播。需要注意的是：这种方式启动的接收器必须手动进行销毁，即调用 **unregisterReceiver()** API，否则在上下文失效时，系统会抛出异常（并不会导致应用崩溃），同时接收器会引发泄露（见图.2-1）。

### 2.3.3 广播接收器的注册方式

```
2020-04-21 18:06:39.557 17978-17978/com.example.myapplication E/ActivityThread: Activity
com.example.myapplication.SecondActivity has leaked IntentReceiver com.example.myapplication
.LeakReceiver@e03b545 that was originally registered here. Are you missing a call to
unregisterReceiver()?
```

图 2-1: 没有回收接收器将会导致异常以及泄露

### Listing 3 广播接收器的注册方式

```
1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6      <application ...>
7          ...
8          <receiver
9              android:name = ".Receiver1">
10             <intent-filter>
11                 <action android:name = "act1" />
12             </intent-filter>
13         </receiver>
14         <receiver
15             android:name = ".Receiver2"
16             android:exported = "false"
17             android:enabled = "true">
18             <intent-filter>
19                 <category android:name = "cat1" />
20                 <action android:name = "act2" />
21             </intent-filter>
22         </receiver>
23     </application>
24 </manifest>
```

一般而言，清单声明的广播接收器（见 2.3.1）需要在 **AndroidManifest.xml** 文件中添加 **<receiver>** 标签（参考 Listing. 3），在 **<intent-filter>** 子标签中可以指定订阅的广播类型，也可以通过设置 **“android:exported”** 属性来指定该广播是否将被导出。而上下文注册的广播接收器（见 2.3.1）则不需要进行这样的操作。

#### 2.3.4 广播接收器的内存泄漏

广播接收器的内存泄漏原理类似于服务内存泄漏（见 2.2.4 小节）。但是由广播接收器引起的内存泄漏往往比服务更为严重，因为广播接收器被设计成只进行不耗时的操作（如果超过 10s 未从 **onReceive()** 方法中返回，将抛出 **ANR(Application Not Response)** 异常），因此通常广播接收器在接到广播后，会启动其他的 **Service** 进行后续的耗时操作。因此如果在广播接收器中产生了

内存泄漏，可能会使他启动的服务也同时泄露，进而导致更严重的后果。

例如图中（见 **Listing.4**）所示的广播接收器，不仅本身会导致内存泄漏，而且还会启动一个会导致内存泄漏的服务（见 **Listing.2**），因此后果将会更加严重。

**Listing 4** 广播接收器的内存泄漏

```

1  public class LeakReceiver extends BroadcastReceiver {
2      private final String TAG = "LeakReceiver";
3      private final int ID = new Random().nextInt();
4      @Override
5      public void onReceive(Context context, Intent intent) {
6          ...
7          context.startService(new
8              ↳ Intent(context, LeakService.class));
9          new Timer().scheduleAtFixedRate(()->{
10              Log.i(TAG, LeakReceiver.this.ID + " is running!");
11          }, 1000L, 3000L);
12      }
13  }

```

## 2.4 电池管理限制

本节将对电池管理限制进行介绍，同时为了理解电池管理限制的具体生效方式，将对安卓启动服务的源码进行追溯，找到办法绕过电池管理限制。

### 2.4.1 简介

在安卓 9 的更新中，引入了**电池管理**<sup>[1]</sup>，这项更新是先前版本若干相关更新的集成和拓展，目的是帮助节约电量消耗，使得安卓设备的续航得到改善。

其主要限制有两种：

**应用待机群组**系统会根据用户使用应用的时间和频率，将应用分成**活跃**、**工作集**、**常用**、**极少使用**、**从未使用**五个群组，这五个群组将会拥有不同的资源分配优先级和权限。具体来说，系统会根据应用所处的群组，来限制应用的行为：例如限制网络连接能力，设定资源分配优先级等。

**省电模式改进**省电模式是在先前的版本中就已经存在的功能，在安卓 9 中继续完善并扩展了这一功能：如禁止所有后台应用启动服务和使用网络访问服务等。

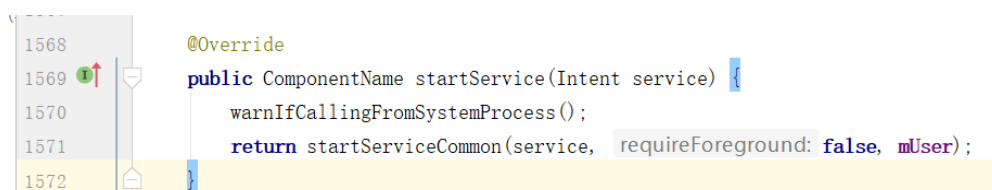
## 2.4.2 对实验方法的影响

由于省电模式中，禁止后台应用启动服务。而在原测试方法中，将由测试驱动应用来启动/停止被测试应用的组件，因此被测试应用的组件的启动行为是后台应用启动服务，于是在最新的安卓版本上，这种原有的测试方式会受到安卓系统的阻止，而下一小节将会介绍如何绕过电池管理限制。

## 2.4.3 电池管理限制白名单

在本小节中，将会追溯安卓源码，在源码的判定逻辑中寻找办法绕过电池管理限制。

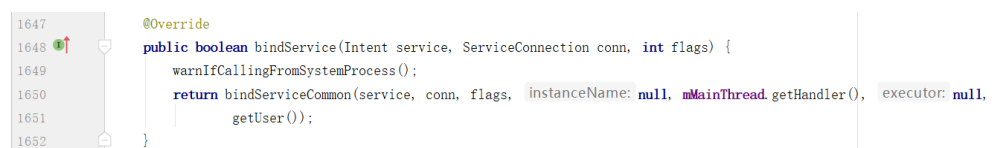
首先，启动服务时，需要调用 `startService()`（见图.2-2）或者 `bindService()`（见图.2-3）API，当调用这两个 API 时，会进入安卓源码的 `ContextImpl.java` 类对应的方法中，可以发现这两个 API 会分别继续调用 `startServiceCommon()` 和 `bindServiceCommon()` 方法，于是我们需要进一步查阅这两个方法。



```

1568
1569 ① ↑
1570
1571
1572
@Override
public ComponentName startService(Intent service) {
    warnIfCallingFromSystemProcess();
    return startServiceCommon(service, requireForeground: false, mUser);
}
    
```

图 2-2: 调用 `startService()` API



```

1647
1648 ① ↑
1649
1650
1651
1652
@Override
public boolean bindService(Intent service, ServiceConnection conn, int flags) {
    warnIfCallingFromSystemProcess();
    return bindServiceCommon(service, conn, flags, instanceName: null, mMainThread.getHandler(), executor: null,
        getUser());
}
    
```

图 2-3: 调用 `bindService()` API

通过阅读 `startServiceCommon()`（见图.2-4）和 `bindServiceCommon()`（见图.2-5）发现了当尝试进行后台启动服务时会抛出的异常信息：**Not allowed to start service ...**，从而确定对后台启动服务的限制具体是由 AMS 来负责处理，进而定位到具体的判定逻辑位于 `ActiveServices.java`（见图.2-6）以及 `ActivityManagerService.java`（见图.2-7）两个类中。

在 `ActiveServices.java`（见图.2-6）的 `startServiceLocked()` 方法的第 494 行调用了 `ActivityManagerService.java`（见图.2-7）的 `getAppStartModeLocked()`

```

1596 private ComponentName startServiceCommon(Intent service, boolean requireForeground,
1597     UserHandle user) {
1598     try {
1599         validateServiceIntent(service);
1600         service.prepareToLeaveProcess(this);
1601         ComponentName cn = ActivityManager.getService().startService(
1602             mMainThread.getApplicationThread(), service, service.resolveTypeIfNeeded(
1603                 getContentResolver()), requireForeground,
1604                 getOpPackageName(), user.getIdentity());
1605         if (cn != null) {
1606             if (cn.getPackageName().equals("!")) {
1607                 throw new SecurityException(
1608                     "Not allowed to start service " + service
1609                     + " without permission " + cn.getClassName());
1610             } else if (cn.getPackageName().equals("!!")) {
1611                 throw new SecurityException(
1612                     "Unable to start service " + service
1613                     + ": " + cn.getClassName());
1614             } else if (cn.getPackageName().equals("?")) {
1615                 throw new IllegalStateException(
1616                     "Not allowed to start service " + service + ": " + cn.getClassName());
1617             }
1618         }
1619         return cn;
1620     } catch (RemoteException e) {
1621         throw e.rethrowFromSystemServer();
1622     }
1623 }

```

图 2-4: startServiceCommon() API

```

1707 private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags,
1708     String instanceName, Handler handler, Executor executor, UserHandle user) {
1709     // Keep this in sync with DevicePolicyManager.bindDeviceAdminServiceAsUser.
1710     IServiceConnection sd;
1711     if (conn == null) {...}
1714     if (handler != null && executor != null) {...}
1717     if (mPackageInfo != null) {...} else {...}
1726     validateServiceIntent(service);
1727     try {
1728         IBinder token = getActivityToken();
1729         if (token == null && (flags & BIND_AUTO_CREATE) == 0 && mPackageInfo != null
1730             && mPackageInfo.getApplicationInfo().targetSdkVersion
1731             < android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
1732             flags |= BIND_WAIVE_PRIORITY;
1733         }
1734         service.prepareToLeaveProcess(this);
1735         int res = ActivityManager.getService().bindIsolatedService(
1736             mMainThread.getApplicationThread(), getActivityToken(), service,
1737             service.resolveTypeIfNeeded(getContentResolver()),
1738             sd, flags, instanceName, getOpPackageName(), user.getIdentity());
1739         if (res < 0) {
1740             throw new SecurityException(
1741                 "Not allowed to bind to service " + service);
1742         }
1743         return res != 0;
1744     } catch (RemoteException e) {...}
1747 }

```

图 2-5: bindServiceCommon() API

方法，在该方法的第 5955 行的注释发现，安卓会维护一个应用白名单，在白名单中的应用将不会受到“电池管理限制”的影响。

```

484 ComponentName startServiceLocked(IApplicationThread caller, Intent service, String resolvedType,
485 int callingPid, int callingUid, boolean fgRequired, String callingPackage,
486 final int userId, boolean allowBackgroundActivityStarts)
487 throws TransactionTooLargeException {
488
489     // If this isn't a direct-to-foreground start, check our ability to kick off an
490     // arbitrary service
491     if (forcedStandby || (!r.startRequested && !fgRequired)) {
492         // Before going further -- if this app is not allowed to start services in the
493         // background, then at this point we aren't going to let it period.
494         final int allowed = mAm.getAppStartModeLocked(r.appInfo.uid, r.packageName,
495             r.appInfo.targetSdkVersion, callingPid, alwaysRestrict: false, disabledOnly: false, forcedStandby);
496         if (allowed != ActivityManager.APP_START_MODE_NORMAL) {
497             Slog.w(TAG, "Background start not allowed: service "
498                 + service + " to " + r.shortInstanceName
499                 + " from pid=" + callingPid + " uid=" + callingUid
500                 + " pkg=" + callingPackage + " startFg?" + fgRequired);
501             if (allowed == ActivityManager.APP_START_MODE_DELAYED || forceSilentAbort) {
502                 // In this case we are silently disabling the app, to disrupt as
503                 // little as possible existing apps.
504                 return null;

```

图 2-6: getAppStartModeLocked() 方法

```

5938 // Unified app-op and target sdk check
5939 int appRestrictedInBackgroundLocked(int uid, String packageName, int packageTargetSdk) {
5940     // Apps that target 0+ are always subject to background check
5941     if (packageTargetSdk >= Build.VERSION_CODES.O) {...}
5942     // ...and legacy apps get an AppOp check
5943     int appop = mAppOpsService.noteOperation(AppOpsManager.OP_RUN_IN_BACKGROUND,
5944         uid, packageName);
5945     if (DEBUG_BACKGROUND_CHECK) {...}
5946     switch (appop) {
5947         case AppOpsManager.MODE_ALLOWED:
5948             // If force-background-check is enabled, restrict all apps that aren't whitelisted.
5949             if (mForceBackgroundCheck &&
5950                 !UserHandle.isCore(uid) &&
5951                 !isOnDeviceIdleWhitelistLocked(uid, /*allowExceptIdleToo=*/ allowExceptIdleToo: true)) {
5952                 if (DEBUG_BACKGROUND_CHECK) {
5953                     Slog.i(TAG, "Force background check: " +
5954                         uid + "/" + packageName + " restricted");
5955                 }
5956                 return ActivityManager.APP_START_MODE_DELAYED;
5957             }
5958             return ActivityManager.APP_START_MODE_NORMAL;
5959         case AppOpsManager.MODE_IGNORED:
5960             return ActivityManager.APP_START_MODE_DELAYED;

```

图 2-7: appRestrictedInBackgroundLocked() 方法

至此，在理论上已经找到了一种绕过电池管理限制的方法：将应用加入到电池管理白名单中。

最终，经过大量的尝试和探索，在安卓的系统设置中，发现了此白名单的添加方式：系统设置 - 应用和通知 - 特殊应用权限 - 电池优化（见图.2-8）。经过实验，将应用添加至此白名单中，确实可以突破“电池管理限制”，可以正常地在后台进行服务的启动，而不会引发异常。

在本文后续的测试过程中，所有的应用都将会添加至此白名单中。

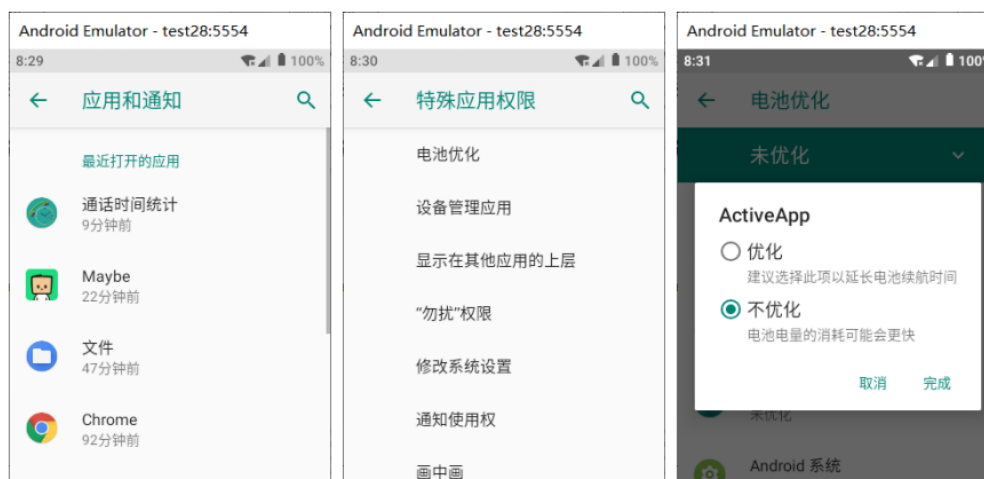


图 2-8: 电池优化白名单添加方式

## 2.5 小结

本章首先介绍了安卓系统的发展历史，以及安卓应用的四大主要组件，意在说明不可见控件在安卓应用中的重要作用。

接下来重点说明了两类不可见组件（服务和广播）的注册、启动方式，及其生命周期，然后举例说明了这两类组件产生内存泄漏的原因，以及内存泄漏将会导致的严重后果，并为后文对内存泄漏组件实例地检测提供依据。

最后通过追溯安卓源码理解了电池管理的具体限制方式，并通过白名单方式绕过了电池管理限制，让应用可以正常地从后台启动服务，从而使得原有的测试方式可以继续使用。





## 第三章 技术方案

本章将介绍安卓控件检测内存泄露的原理。首先将从宏观上对安卓不可见控件内存泄漏的自动化检测工具进行全面的讲述，提出测试的主要流程和步骤。接下来在后续的章节中对各个模块进行具体的细节讲述。

### 3.1 系统架构

首先，测试的资料为由其他开发者所开发的`.apk`文件，本文的测试对象是服务和广播接收器，由于是黑盒测试，不会获取到应用的源码。如前文中对服务和广播接收器的描述，我们需要寻找合适的工具对`apk`文件进行解析反编译，目的是得到`AndroidManifest.xml`清单文件。在得到清单文件后，需要编写脚本对该清单进行解析，从而得到本文关注的测试对象（公开服务和清单中注册的静态广播接收器），生成一份测试配置。

然后，为了实现安卓系统上的跨应用组件的测试，需要设计实现一个安卓应用，这个安卓应用的作用是驱动整个在安卓模拟器上的测试步骤。例如：读取测试配置，进行跨应用组件的启动与关闭，处理测试时发生的异常(Exception)等等。后文将称此应用为测试驱动应用。

接下来，当测试完成时，我们需要判定被测试应用是否发生了内存泄露，以及找到所有内存泄漏组件的实例。因此必须要对被测试应用的堆进行分析，这一步我们可以将堆文件镜像导出，然后对这一时刻的堆中的对象设计算法进行分析。后文将称此分析工具为内存分析器。

#### 3.1.1 基于测试效率的考量

由于测试时，必然要启动安卓模拟器，而安卓模拟器的运行原理为虚拟机，将会占用固定数额的电脑内存等资源，因此如果所有步骤都在安卓模拟器上完成将会使效率大打折扣。

结合上文的方案概述，可以将若干与主要测试流程不相关、或者安卓模拟器上很难实现的步骤放到工作站上来进行，即解析`apk`文件和内存分析。

因此需要建立安卓模拟器和工作站的通讯，由于通讯在本文中只需要简单的消息发送的功能，因此可以使用 python 建立简单的 socket 连接即可。

### 3.1.2 具体设计

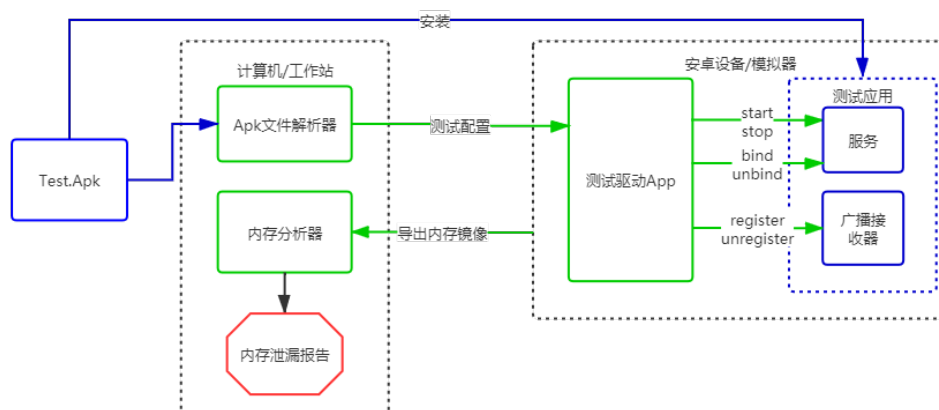


图 3-1: 自动检测工具原理

如图 3-1 所示，测试将在两个环境（工作站、安卓模拟器）中完成：

1. 在工作站中，使用 **Apk 文件解析器**（详见 3.2）对被测试应用 **Test.apk** 进行反编译，并解析 **AndroidManifest.xml** 清单，得到在清单中注册的公开服务以及清单声明的广播接收器，将这些组件作为测试对象生成一份测试配置文件发送到安卓模拟器上。
2. 重置安卓模拟器，保证测试时使用相同的环境。在安卓模拟器上安装好测试驱动 **App**（详见 3.3）以及被测试应用 **Test.apk**。通过 **ADB** 向安卓模拟器发送指令启动测试驱动 **App**，它会读取步骤 1 中发送来的测试配置文件，然后执行测试主流程（详见 3.4）
3. 在完成测试主流程之后，通知工作站将被测试应用的堆镜像文件（.hprof）文件导出，使用**内存分析器**（详见 3.5）进行内存泄漏的检测，最终生成一份**内存泄漏报告**显示被测试应用的服务以及广播接收器有无内存泄漏情况，以及具体的内存泄漏的控件清单。

## 3.2 Apk 文件解析器

Apk 文件解析器旨在通过将被测试应用反编译得到 **AndroidManifest.xml** 清单，对该清单进行分析，从而得到在清单中声明的**公开服务**以及**广播接收器**

作为测试对象。

### Listing 5 使用 apktool 工具进行 apk 的反编译

```
1 apktool d -f $test_apk$ -o temp
```

使用 apktool 工具<sup>[1]</sup>，执行 **Listing.5** 中的命令即可完成被测试应用的反编译得到 **AndroidManifest.xml** 清单。

### Listing 6 使用 python 解析 xml 输出测试配置

```
1 from xml.dom.minidom import parse
2 def get_exported_services(xml_path):
3     exported_services = []
4     manifest = parse(xml_path).documentElement
5     for service in manifest.getElementsByTagName('service'):
6         exported = service.getAttribute('android:exported')
7         enabled = service.getAttribute('android:enabled')
8         if (exported == 'true' and (enabled == '' or enabled ==
9             ↪ 'true')):
10             exported_services.append(service)
11     return exported_services
12
13 def get_static_receivers(xml_path):
14     exported_receivers = []
15     domTree = parse(xml_path)
16     manifest = domTree.documentElement
17     for receiver in manifest.getElementsByTagName('receiver'):
18         enabled = receiver.getAttribute('android:enabled')
19         if (enabled == '' or enabled == 'true'):
20             exported_receivers.append(receiver)
21     return exported_receivers;
```

接下来使用 **python** 中的 **xml** 库解析此 **xml** 文件（详见 **Listing.6**），筛选出所有指定 **android:exported = “true”** 以及 **android:enabled = “true”**（或者不指定，默认值为“true”）的服务，以及在清单中声明的广播接收器。将这些控件的包名，类名，以及所需的权限等信息写入测试配置文件中，至此完成了对被测试应用的组件分析。

## 3.3 测试驱动应用

测试驱动应用负责控制所有安卓模拟器上的测试行为（即测试主流程 **3.4**）。首先测试驱动应用会读取测试配置文件，从中获得要进行测试的服

务和广播接收器，之后就会进入到**测试主流程**，在完成测试流程之后，使用**Socket**与**工作站**进行通信，通知**工作站**测试任务已经完成，此时**工作站**将会导出被测试应用的堆镜像文件进行最后的分析。

### 3.4 测试主流程

---

#### 算法 3.1 测试主流程：公开服务测试

---

```

1: 读取测试配置
2: for 测试配置中的公开服务 do
3:   重置安卓模拟器
4:   重置计数器
5:   if 使用 bind 方式测试 then
6:     启动仿真 Activity
7:   end if
8:   while 计数器 < 测试重复次数 do
9:     if 使用 start 方式测试 then
10:      调用 startService() API 启动服务
11:      调用 stopService() API 停止服务
12:     else
13:      调用 bindService() API 将服务绑定到 Activity 上
14:      调用 unbindService() API 解除服务绑定
15:     end if
16:     计数器自增 1
17:   end while
18:   if 使用 bind 方式测试 then
19:     销毁仿真 Activity
20:   end if
21:   通知工作站导出堆镜像
22: end for
```

---



---

#### 算法 3.2 测试主流程：清单声明的广播接收器

---

```

1: 读取测试配置
2: for 测试配置中的广播接收器 do
3:   重置安卓模拟器
4:   重置计数器
5:   while 计数器 < 测试重复次数 do
6:     构造特定广播，定向发送给该接收器
7:     计数器自增 1
8:   end while
9:   等待 10s{等待所有广播接收器从 onReceive() 方法中返回}
10:  通知工作站导出堆镜像
11: end for
```

---

在测试主流程中，对服务和接收器进行逐一测试：

**服务测试**（见算法 3.1）测试原理为对被测试应用进行大量重复的启动和关闭（根据测试要求不同分为 start/bind 和 stop/unbind），确保潜在的内存泄漏被触发，之后将被测试应用的堆镜像导出，送入**内存分析器**（详见 3.5）进行检测和分析。

**广播接收器测试**（见算法 3.2）广播接收器的测试不需要事先启动应用，因为在清单中注册的广播接收器在应用安装时就会在系统中进行注册，在任何时候都可以接收广播（即使应用未运行），因此只需要构造接收器订阅的广播，定向发送给该接收器就可以触发该接收器。同样的为了确保潜在的内存泄漏被触发，需要重复发送大量的广播。在结束测试时，需要额外等待 10s，目的是让所有广播接收器从 **onReceive()** 方法中返回，进而使得广播接收器可以正常地被销毁。最终将被测试应用的堆镜像导出，送入**内存分析器**（详见 3.5）进行检测和分析。

## 3.5 内存分析器

---

### 算法 3.3 内存分析器：服务分析

---

```

1: 导入并解析.hprof 文件
2: 查询所有 Service 的衍生类集合
3: for Service 的衍生类集合 do
4:   if 该实例已经被销毁 then
5:     获得所有该实例到 GC Root 的路径集合
6:     去除路径集合中不合理的路径
7:     if 路径集合为空 then {证明该实例确实已经被销毁}
8:       该实例不存在内存泄漏
9:     else {该实例仍然被引用，产生内存泄漏}
10:      该实例产生内存泄漏
11:      if 该实例内存泄漏原因为已知的安卓系统 BUG then
12:        不判定构成人为内存泄漏
13:      else
14:        由开发者造成的人为内存泄漏
15:      end if
16:    end if
17:  end if
18: end for

```

---

内存分析器负责从堆镜像文件（.hprof 文件）中识别和统计服务（广播接收器）的内存泄漏实例，基于开源工具 MAT<sup>[2]</sup> 进行定制化的开发，以自动检测内存泄漏组件的实例。具体原理（详见算法 3.3 及 3.4）为：首先导入并解析堆镜像文件，找到所有的服务（广播接收器）实例，接下来对于每个找到的实

---

**算法 3.4** 内存分析器：服务分析

---

```

1: 导入并解析.hprof 文件
2: 查询所有 BroadcastReceiver 的衍生类集合
3: for BroadcastReceiver 的衍生类集合 do
4:   if 该实例已经被销毁 then
5:     获得所有该实例到 GC Root 的路径集合
6:     去除路径集合中不合理的路径
7:     if 路径集合为空 then {证明该实例确实已经被销毁}
8:       该实例不存在内存泄漏
9:     else {该实例仍然被引用，产生内存泄漏}
10:      该实例产生内存泄漏
11:      if 该实例内存泄漏原因为已知的安卓系统 BUG then
12:        不判定构成人为内存泄漏
13:      else
14:        由开发者构成的人为内存泄漏
15:      end if
16:    end if
17:  end if
18: end for

```

---

例，逐个检测判断是否产生了内存泄露，对于产生内存泄漏的控件，生成一份内存泄漏分析报告。

**服务的内存泄露判定** 所有仍然活跃的服务都会被 **ActivityThread** 的 **mServices** 变量所引用，而所有被销毁的服务都会被从 **mServices** 中删除。一个对象的实例如果实际上处于存活状态，则一定会拥有至少一条有效的 **GC Root Path**。因此服务内存泄漏的判定充要条件为：拥有至少一条有效的 **GC Root Path** 且不在 **mServices** 中。

**广播接收器的内存泄漏判定** 由于广播接收器被成绩称不耗时控件（超过 10s 未完成 **onReceive()** 方法，将会抛出 **ANR Exception** 导致应用崩溃），因此对于广播接收器而言，只需要检测是否有实际处于存活状态的实例即可。即：拥有至少一条有效的 **GC Root Path**。

**安卓系统自身原因产生内存泄漏** 由于安卓系统自身可能存在 **BUG** 导致正常的控件产生内存泄漏，此类问题并非由于安卓应用开发者的失误而导致，因此要将此类内存泄漏排除。例如：由输入法聚焦导致的系统级内存泄漏。

## 3.6 其他实现细节

**后台服务限制**<sup>[4]</sup> 在最新的安卓版本（安卓 10）中，“电池管理限制”不允许后台应用创建后台服务，因此直接跨应用对服务进行测试将会失败。解决

办法为：将应用加入到电池管理白名单中，或者在测试时，将被测试应用置于前台。

**广播限制**<sup>[12]</sup> 在**安卓 10**中，禁止应用将隐式广播注册为清单声明的广播接收器。而显示广播和需要签名授权的广播接收器不受限制，可以继续注册为清单声明的广播接收器。因此需要给测试驱动应用申请足够的权限。**注意：**随着安卓系统的更新，很多隐式广播已经不再受到此规定的限制，具体的广播列表可以参考隐式广播例外<sup>[13]</sup>。

**超级用户限制** 由于安卓是基于 **Linux kernel** 开发而来，而在 **Linux** 中，超级用户是一个拥有系统最高权限的用户。在先前的安卓系统中，开发者经常会为系统增加超级用户权限，以便于进行测试（亦成为 **root**）。大致的方法为将兼容的二进制文件 **su** 拷贝到安卓设备中，使安卓设备可以执行超级用户指令 **sudo**，进而为测试带来便利。然而在**安卓 10**中，由于用于 **root** 操作的二进制文件 **su** 维护团队相继解散，由非安卓系统开发人员获得超级用户权限变得困难，因此本文的自动化检测工具的实现不要求对系统进行 **root**，而将使用 **socket** 建立安卓设备与工作站的通信，由工作站使用 **adb** 指令完成各种跨应用的特殊权限操作：比如**跨应用导出堆镜像文件**操作需要应用拥有超级用户权限，而使用 **adb** 指令则不需要超级用户权限即可导出任意应用的堆镜像文件。

### 3.7 小结

本章中主要讨论了自动化检测工具实现的流程、细节以及若干基于测试效率的考量，然后说明了各个模块的实现细节，以及各种检测分析的算法原理。最后说明了一些方案不可行的原因。





## 第四章 实验

为了验证自动化检测工具的可行性、正确性，在本章中将设计实验进行验证。

首先将会设计仿真测试，在仿真测试中，会设计轻量级的、最简单的仿真应用，以检验自动检测工具是否能够正确检测出内存泄漏的组件，同时不出现误检的行为。

接下来为了评价在安卓 10 上服务和广播接收器的内存泄漏情况，将会从安卓应用市场中下载真实的应用，使用本文所述的工具进行测试，得出结论。

### 4.1 仿真测试

#### 4.1.1 仿真应用

仿真应用（见 **Listing. 10**）应同时具有两个公开服务以及两个清单声明的广播接收器。其中每个控件之一需要人为制造内存泄漏（称为 **LeakedService(Listing. 7)** 与 **LeakedReceiver(Listing. 8)**），另一个则需要确保不存在内存泄漏（称为 **NormalService(Listing. 9)** 与 **NormalReceiver(Listing. 9)**）。在对仿真应用进行测试时，预期的实验结果为：能够检测到 **LeakedService** 和 **LeakedReceiver** 的泄露实例，以证明该工具可以发现内存泄漏问题；而检测不到 **NormalService** 和 **NormalReceiver** 的泄露实例，以证明该工具不会将正常的组件误检。

#### 4.1.2 实验结果

实验结果（见 **图. 4-2**及 **图. 4-1**）能正确检测到 **LeakedService** 和 **LeakedReceiver** 的内存泄漏实例，而没有误检 **NormalService** 以及 **NormalReceiver**，证明检测工具实际有效。

### Listing 7 LeakedService 主体代码

```

1  public class LeakedService extends Service {
2      private static final String TAG = "LeakedService";
3      @Override
4      public void onCreate() {
5          super.onCreate();
6          new Timer().scheduleAtFixedRate(new TimerTask() {
7              @Override
8              public void run() {
9                  Log.i(TAG, LeakedService.this.getPackageName() +
10                      "\n LeakedService running ");
11              }
12          }, 1000L, 3000L);
13      }
14  }

```

### Listing 8 LeakedReceiver 主体代码

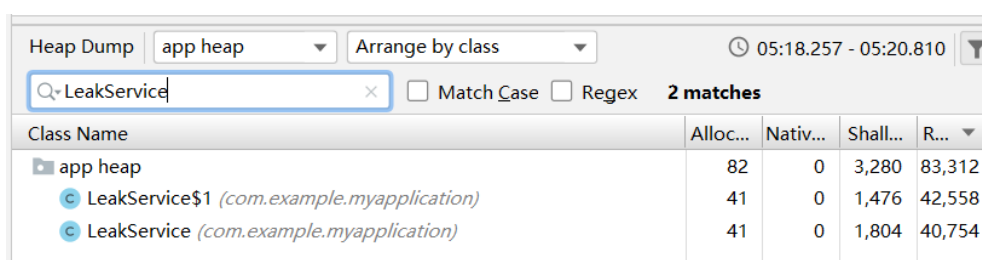
```

1  public class LeakedReceiver extends BroadcastReceiver {
2      private static final String TAG = "LeakedReceiver";
3      private final Random random = new Random();
4      @Override
5      public void onReceive(Context context, Intent intent) {
6          new Timer().scheduleAtFixedRate(new TimerTask() {
7              @Override
8              public void run() {
9                  Log.i(TAG, LeakedReceiver.this.random.nextInt() +
10                      "\n LeakedReceiver running ");
11              }
12          }, 1000L, 3000L);
13      }
14  }

```

## 4.2 真实测试

本文选取了 **AppChina 应用市场**<sup>[14]</sup> 作为测试应用的来源，在其中的视频, 游戏, 工具三个类别中，选取了每个分类中当月下载量最高的若干应用进行测试



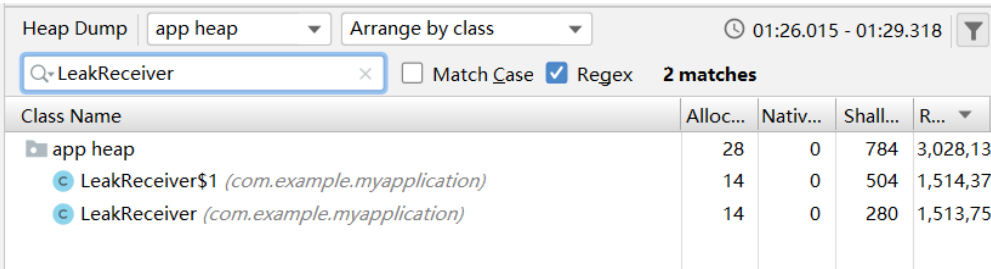
The screenshot shows a memory dump analysis interface. At the top, there are tabs for 'Heap Dump' and 'app heap'. A search bar contains the text 'LeakedService'. Below the search bar, there are checkboxes for 'Match Case' and 'Regex', and a result count of '2 matches'. The main table displays the following data:

Class Name	Alloc...	Nativ...	Shall...	R...
app heap	82	0	3,280	83,312
LeakedService\$1 (com.example.myapplication)	41	0	1,476	42,558
LeakedService (com.example.myapplication)	41	0	1,804	40,754

图 4-1: 检测到 **LeakedService** 内存泄漏实例

Listing 9 NormalReceiver 与 NormalService 主体代码

```
1 public class NormalReceiver extends BroadcastReceiver {
2     private static final String TAG = "NormalReceiver";
3     private final Random random = new Random();
4     @Override
5     public void onReceive(Context context, Intent intent) {
6         Log.i(TAG, NormalReceiver.this.random.nextInt() +
7             ↳ ".NormalReceiver running ");
8     }
9 }
10
11 public class NormalService extends Service{
12     private static final String TAG = "NormalService";
13     @Override
14     public void onReceive(Context context, Intent intent) {
15         Log.i(TAG, NormalService.this.getPackageName() +
16             ↳ ".LeakService running ");
17     }
18 }
```



The screenshot shows the 'Heap Dump' tool in Android Studio. The search bar contains 'LeakReceiver'. The results table shows two matches for the class 'LeakReceiver' from the package 'com.example.myapplication'.

Class Name	Alloc...	Nativ...	Shall...	R...
app heap	28	0	784	3,028,13
LeakReceiver\$1 (com.example.myapplication)	14	0	504	1,514,37
LeakReceiver (com.example.myapplication)	14	0	280	1,513,75

图 4-2: 检测到 LeakedReceiver 内存泄漏实例

试，共计 22 个。对这 22 个应用的测试结果详见 4.2.1 小节：

4.2.1 实验结果

表现正常 12(54.5%) 这些应用的组件表现正常，并没有出现内存泄露问题。

内存泄漏 3(13.6%) 这些应用包含了存在内存泄漏问题的组件，并成功被检测工具检测到。

应用崩溃 7(31.8%) 这些应用在测试时抛出了 ANR(Application Not Response) 异常，导致应用崩溃，无法完成测试。

**Listing 10** 仿真应用的 AndroidManifest.xml 清单

```

1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6
7      <permission
8          android:name="app.custom.permission"
9          android:protectionLevel="signature" />
10     <application ...>
11         <activity android:name=".MainActivity">
12             <intent-filter>
13                 <action
14                     ↪ android:name="android.intent.action.MAIN"
15                     ↪ />
16                 <category
17                     ↪ android:name="android.intent.category.LAUNCHER"
18                     ↪ />
19             </intent-filter>
20         </activity>
21
22         <receiver
23             android:name=".LeakedReceiver"
24             android:exported="true">
25             <intent-filter>
26                 <action android:name="TestActionForLeaked" />
27             </intent-filter>
28         </receiver>
29
30         <receiver
31             android:name = ".NormalReceiver"
32             android:exported = "true">
33             <intent-filter>
34                 <action android:name = "TestActionForNormal" />
35             </intent-filter>
36         </receiver>
37
38         <service
39             android:name=".LeakedService"
40             android:exported="true">
41         </service>
42
43         <service
44             android:name = ".NormalService"
45             android:exported = "true">
46         </service>
47     </application>
48 </manifest>

```

## 4.2.2 应用崩溃主要原因分析

**无法正常启动** 部分应用在启动时即发生了崩溃，导致应用停止运行。这类应用崩溃的原因可能是应用的版本和模拟器系统的版本不兼容，例如使用了

不再符合开发规范的接口，某些接口不再支持，或没有在 **AndroidManifest.xml** 中声明所需的权限（缺少 **<uses-permission>** 标签）等。

**在测试过程中崩溃** 部分应用可以正常启动，但是在**主测试流程**中发生了应用崩溃问题。这类应用崩溃的原因可能是因为应用的组件启动流程存在问题，比如：进行了风险操作，进行了线程不安全操作，没有对启动环境进行检查等；也可能是由于应用的组件确实存在**内存泄漏**问题，而且泄露表现的十分严重，触发了安卓系统的安全限制，导致安卓系统介入将应用停止运行。

**空指针异常** 部分应用的组件在启动时会抛出 **NullPointerException**，该异常表示，在组件的启动过程中，对未实例化的对象进行了读写操作。这类问题大多数是因为组件的开发人员的疏忽，导致的程序缺陷。可能的成因有组件之间使用了共享资源，但是并没有进行专门管理，也有可能是因为这类组件的启动严格遵循**状态自动机**，但是测试时无法得知组件启动需要满足的前置条件，导致运行出错。

## 4.3 结果分析

本小节会将本文的测试结果（见 4.2.1）计算指标来进行分析。

### 4.3.1 数据分析

表 4-1: 实验数据及指标

	表现正常	内存泄漏	运行异常	泄露正常比
指标	54.5%	13.6%	31.8%	0.250

实验数据的对比表明：

- 存在内存泄漏的应用比例为 **13.6%**，说明组件内存泄漏现象较为普遍。
- 运行异常的应用比例为 **31.8%**，说明安卓组件碎片化问题比较严重，兼容性问题相对于内存泄漏问题更为明显。兼容性问题具体原因有：安卓版本升级后，系统权限的收紧，导致应用无法像先前版本一样正常获取响应权限；组件的开发规范变更，导致原有代码抛出异常，导致应用崩溃等。
- 泄露正常比（检测到内存泄漏问题的应用数量与表现正常的的应用数量之比）为 **0.250**，这项指标相较于**内存泄漏应用的比例**更有实际意义，它说明了在**经过成熟测试的稳定可用的应用中**，组件的内存泄漏问题依然较为普

遍，这要求开发者在进行不可见组件开发的时候，需要投入更多的精力进行测试和调试，以确保这些组件中不会出现程序缺陷。

### 4.3.2 实验数据局限性

由于实验设备和实验环境的限制（参考表.4-2），本文只能使用效率较慢的方式对少量小规模的应用进行串行的测试（参考表.4-3）。因此实验结果具有较大的局限性。

表 4-2: 工作站配置

	操作系统	内存容量	处理器型号	模拟器系统版本
测试配置	Windows 10	8 GB	Intel(R) Core(TM) i7-8565U	Android 10

表 4-3: 测试方法

	并发能力	模拟器内存	模拟器 SD 卡容量	测试强度
测试方法	单线程	2 GB	512 MB	每个应用测试 2 次

## 4.4 小结

在本章中，设计了仿真测试实验，以及真实测试实验。前者验证了自动化检测工具的正确性和可行性，而后者研究了安卓 10 中，应用的组件内存泄漏的发生几率，一定程度上验证了安卓组件开发碎片化的问题，同时证明安卓不可见控件内存泄漏的问题依然广泛存在。

## 第五章 结论

本文实现了一个自动化的检测工具，它可以帮助安卓应用的开发人员对应用进行测试，帮助发现存在内存泄露问题的公开服务以及清单声明的广播接收器，尽量减少应用中的缺陷和错误。

在本文的解决方案中，首先对被测试应用进行反编译，接下来编写了脚本分析应用组件清单，得到公开服务和清单声明的广播接收器的列表，作为被测试对象。之后启动安卓模拟器，将被测试应用、测试驱动应用安装到模拟器上，并启动测试驱动应用来负责执行整个测试任务：对被测试组件进行反复的启动/关闭，绑定/解绑定，注册/解注册操作，最大程度确保内存泄漏问题被实际触发。最后寻找内存泄漏的组件，将被测试应用的堆镜像文件导出到工作站中，基于开源工具开发了检测安卓组件内存泄漏的定制工具，最终导出组件的内存泄漏情况报告，以指导开发者对应用进行进一步的测试和调试工作。

但是本文的工作也存在相应的不足：如在实验的测试环节，由于工作站配置的制约，选取的应用数量和规模都很小，因此测试数据具有较大的偶然性；由于难度较大，本文针对广播接收器的测试中，只选取了在清单中注册的广播接收器进行测试，而没有对动态注册的广播接收器进行测试。

在未来的工作中，针对本文的上述不足大致有两个改进的方向：第一，针对测试不充分的问题，可以将本文的测试方法进行并行化的改进，然后使用更大量的应用进行充分的测试，来统计实验数据，这样会更有说服力。第二，对于动态注册的广播接收器，由于无法在清单中自动分析出这些接收器，且注册的过程相对比较复杂，可以寻找一些新的方式去对这一部分接收器进行测试；同理，内容提供者也属于安卓应用的不可见控件，而且也很容易出现内存泄漏的问题，因此对于内容提供者内存泄漏的检测也是未来工作的方向之一。





## 参考文献

- [1] CONNOR TUMBLESON R W. A Tool for Reverse Engineering Android Apk Files[K/OL]. 2019 [2020-04-06].  
<https://ibotpeaches.github.io/Apktool/>.
- [2] AndrodMat 2012.[K/OL]. 2012 [2020-04-06].  
<https://github.com/joebowbeer/andromat>.
- [3] Android Power Saver[K/OL]. 2020 [2020-05-24].  
<https://developer.android.com/about/versions/pie/power#battery-saver>.
- [4] Android Background Service Limit[K/OL]. 2020 [2020-04-27].  
<https://developer.android.google.cn/about/versions/oreo/background#services>.
- [5] JUN M, SHAOCONG L, JIANG Y, et al. LESDroid-A Tool for Detecting Exported Service Leaks of Android Applications[C] // 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). 2018 : 244 – 24410.
- [6] CHIN E, FELT A P, GREENWOOD K, et al. Analyzing inter-application communication in Android[C] // Proceedings of the 9th international conference on Mobile systems, applications, and services. 2011 : 239 – 252.
- [7] 刘洁瑞, 巫雪青, 严俊, et al. 针对 Android 资源泄漏的基准测试集的构造与评测 [J]. 计算机应用, 2016 : 0 – 0.
- [8] Android Service Guide[K/OL]. 2020 [2020-04-20].  
<https://developer.android.com/guide/components/services.html>.
- [9] Android Broadcasts Guide[K/OL]. 2020 [2020-4-29].  
<https://developer.android.google.cn/guide/components/broadcasts>.
- [10] Android Broadcast Guide[K/OL]. 2020 [2020-04-21].  
<https://developer.android.com/guide/components/broadcasts>.

- 
- [11] Android Power Management[K/OL]. 2020 [2020-05-24].  
<https://developer.android.google.cn/about/versions/pie/power>.
- [12] Android Broadcast Receiver Limit[K/OL]. 2020 [2020-04-27].  
<https://developer.android.google.cn/about/versions/oreo/background#broadcasts>.
- [13] Implicit Broadcast Exceptions[K/OL]. 2020 [2020-04-27].  
<https://developer.android.google.cn/guide/components/broadcast-exceptions>.
- [14] AppChina: An App Market for Chinese Mobile Users[K/OL]. 2020 [2020-04-28].  
<http://www.appchina.com/>.

## 致 谢

由衷地感谢我的导师马骏副教授对我进行了悉心的指导，恩施思维敏捷，脚踏实地，对我实验过程中遇到的苦难和疑惑，都进行了仔细耐心的讲解和指导，对我的毕业设计的完成给与了巨大的帮助。

我也要感谢实验室的同学们，在三年的训练中，我们一起互相帮助，同甘共苦，攻坚克难，是我奋斗路上可靠的队友。愿我们的友谊地久天长，愿你们今后取得更高的成就。

另外，我还要感谢开源社区中无数勤劳无私的贡献者们，没有你们开发的众多的开源工具，本文的工作无法完成。在此感谢开源工作者，以及众多博主的技术博客，让我学到了很多。

最后，还要感谢我的家人，无论我身在哪里，家人都一如既往的支持鼓励着我，是我前进路上最坚强的后盾。

感谢所有支持帮助我的人，也祝各位同学前途无量。