分类号　　密级

中国地质大学（北京）

本 科 毕 业 设 计

题　　目　安卓 App 热补丁动态修复技术

英文题目　Android HotPatch

学生姓名　王彬龙　　**院（系）**　信息工程学院

专　　业　软件工程　　**学　　号**　1004126121

指导教师　姚国清　　**职　　称**　教授

二〇一六 年 五 月

中国地质大学（北京）本科毕业设计（论文）开题报告

学号	1004126121	姓名	王彬龙	专业	软件工程
导师姓名	姚国清	职称	教授	单位	信息工程学院
课题性质	设计√ 论文		课 题 来源	科研教学生产其它√	
毕业设计（论文）题目	安卓 App 热补丁动态修复技术				

开题报告（阐述课题的目的、意义、研究现状、研究内容、研究方案、预期结果等）

目的：使用热补丁动态修复技术，向用户下发 Patch，在用户无感知的情况下，修复 bug

意义：App 发布之后，突然发现了严重 bug 需要进行紧急修复，公司各方就会忙得焦头烂额：重新打包 App、测试、向各个应用市场和渠道换包、提示用户升级、用户下载、覆盖安装。有时候仅仅是为了修改了一行代码，也要付出巨大的成本进行换包和重新发布。采用热补丁动态修复技术就不再需要重新发布 App，不再需要用户重新下载，覆盖安装这些影响用户体验的操作，做到在线热更新。

研究内容：android 热修复方案

研究方案：把补丁文件嵌入到 app 的 classloader 之中，动态加载 dex 中的源码

预期结果：运行中的应用通过网络后台用户无感知地下载补丁文件，之后动态替换掉应用中的 bug。

指导教师意见： 指导教师签名： 年　　月　　日
评议小组意见： 审 查 结 果：□ 同 意　　□ 不 同 意 评议小组组长：成员： 年　　月　　日

摘　要

APP 发布到应用市场之后，出现了一个严重 bug。需要进行紧急修复，于是开启流程，包括：修改 bug、重新打包 App、测试、向各个应用市场和渠道换包、提示用户升级、用户下载、覆盖安装。在项目复盘的时候，发现只是为了修改几行代码，但是付出的代价是惨重的。

当 bug 可以通过 patch 的方式动态修复,不再需要重新发布、提示用户下载等一系列动作。保证应用的正常运行。本文完成了 Bug 热修复 SDK 的制作，开发程序只需要集成这个 SDK，然后配合后台就可以完成热修复的功能。

利用 Android 的 ClassLoader 体系动态加载补丁，在实际运行中实现“覆盖”有 bug 的 dex 文件。一个 ClassLoader 可以包含多个 dex 文件，每个 dex 文件是一个 Element，多个 dex 文件排列成一个有序的数组 dexElements，当找类的时候，会按顺序遍历 dex 文件，然后从当前遍历的 dex 文件中找类，如果找类则返回，如果找不到从下一个 dex 文件继续查找。Sdk 所做的就是将补丁中的 dex 文件覆盖掉之前的 dex 文件，这样就达到热修复的目的。

关键词：Android； bug 热修复； classloader

ABSTRACT

After a APP release, suddenly there was a serious need for urgent bug fixes, this time the entire project team will be bruised and battered: Modify the bug, repackage App, test, market and channel to each application to change the package, the user is prompted to upgrade, download, covering installation. Replay of the project, when it is possible to find just to modify a few lines of code, but the cost is heavy. So there is an idea: Can dynamic way to patch repair urgent Bug, no longer need to re-release, prompting the user to download the series allows users to experience downward movement. Bug fixes thermal paper completed the production SDK, developers only need to integrate the SDK program, and then with the background to complete the repair of the thermal features. Under the principle of generalization, is the use of Android system ClassLoader dynamic loading patchdex, realized in the actual operation "cover" a bug's dex. A ClassLoader dex can contain multiple files, each file is a dex Element, multiple dex files arranged in an ordered array dexElements, when looking for class, will be sequentially through dex file and dex files from the current traversal Get in class, if the class is looking to return, if not found continue to find from a dex file. Sdk does is patchdex on bugdex of the "front", so that to achieve the purpose of covering.

Key words： Android; hotpatch; classloader

目　录

1. 引言

一款应用成熟之后，需求非常巨大，维护成本非常高，此时就需要将项目代码按照功能模块进行拆分，也就是所谓的插件化。插件化的基础就是代码的组件化和代码的线上热部署，所有这些概念的起源就是代码的热修复。

现在越来越多的 app 开始采用本地 app 结合 html5 的设计结构,这样做不仅可以跨平台,并且在某个业务或模块出了严重的 bug,可以直接在 html5+后台实时修复,省去了发布新包和新版本的步骤。但是 html5 的渲染性能比起本地代码还是有一定差距，所以也有部分应用在进行热修复探索的时候，采用的是本地代码进行。

本文主要实现将热修复框架制作成 SDK，之后结合后台服务器，方便开发人员修复线上 bug。用户不用重新经历 apk 安装的安装过程,而是直接下载一个补丁包,通过补丁来替换一些出现 bug,下载补丁的过程用户一般是感觉不到的,表面上看是直接修复了 bug。

1.1 背景

热修复补丁不会作为常规补丁随系统自动更新，一般通过网络来通知用户有关热补丁的消息，用户可以在应用提供者的后台服务器免费下载补丁程序。和直接升级软件版本相比，热修复的主要优势是不会使设备当前正在运行的业务中断，也就是在不重启设备的情况下，可以对设备当前应用版本的缺陷进行修复。

作者在公司实习期间负责 APP 主站中登录注册模块的开发。为了防止黑客模拟手机号攻击服务器接口，将手机号进行了加密处理，并且为了防止应用被反编译然后被拿到加密方式，将加密算法通过 NDK 写入 Native 层。在没有进行单元测试的情况下，匆匆就上线了。上线后，果不其然地出现加密错误--加密算法出现问题导致服务器拿不到真正的手机号。要知道正常情况下，用户安装 app 后，只能通过重新安装的方式才可以更新代码。然后事情就开始了：先将加密算法修复并通过单元测试、重新打包 App、测试、向各个应用市场和渠道换包、提示用户升级、用户下载、覆盖安装等，最后再向团队做检讨。痛定思痛，如何可以让自己的错误可以挽回？于是开始调研当前 Android 客户端的 bug 热修复技术。

1.2 国内外研究现状

在热修复方面，国内大厂都有自己开源实现：QQ 空间推出 ClassLoader 方案，把多个 dex 放进 app 的 classloader 之中，这样使得应用中所有类都能被找到。淘宝提供的 Dexposed 方案，基于 Xposed 的 AOP 框架，方法级粒度。主要思路是在 native 层中先找到要修复的 Java 函数对应的 Method 对象，修改它变为 native 方法，把它的 nativeFunc 指向 hookedMethodCallback。这样对当前 java 函数的调用就转为调用 hookedMethodCallbacknative 函数了，然后再用这个 native 函数回调 java 层实现的统一接口来处理。这个统一接口是 XC_MethodReplacement 类，它主要有 beforeHookedMethod 、afterHookedMethod 和 replaceHookedMethod 等几个方法，前两个在执行原 java 函数前后做一些事，replaceHookedMethod 则是替换原 java 方法。比较成熟且影响力比较大的框架有 360 的 droidplugin 方案，特点有：支持 Androd 2.3 以上系统；插件 APK 完全不需做任何修改，可以独立安装运行、也可以做插件运行。要以插件模式运行某个 APK，你无需重新编译、无需知道其源码；插件的四大组件完全不需要在 Host 程序中注册，支持 Service、Activity、BroadcastReceiver、ContentProvider 四大组件；插件之间、Host 程序与插件之间会互相认为对方已经”安装”在系统上了；API 低侵入性：极少的 API。HOST 程序只是需要一行代码即可集成 Droid Plugin；超强隔离：插件之间、插件与 Host 之间完全的代码级别的隔离：不能互相调用对方的代码。通讯只能使用 Android 系统级别的通讯方法；支持所有系统 API 资源完全隔离：插件之间、与 Host 之间实现了资源完全隔离，不会出现资源窜用的情况；实现了进程管理，插件的空进程会被及时回收，占用内存低。

国外 google 提供的 multidex 方案，起初这个方案是为了解决另一个问题：方法数超过 65kAndroid 应用程序包含的可执行文件 Dex,其中包含用于运行开发者的应用程序的可执行字节码文件。Dalvik 可执行规范限制了可以在单个 dex 文件内引用 65536：包括 Android 框架方法，库方法，并在自己的代码中实现的方法总数。突破这个限制需要配置开发者的应用程序构建过程，生成多个 dex 文件，被称为 multidex 配置。在实现 multidex 的过程中，Google 采用方案是 ClassLoader 先加载主 dex，然后再加载次 dex，原理类似于热修复，也是本篇论文的理论来源。

AndroidStudio 从 2.0 开始，加入了一个功能叫做 InstantRun，顾名思义，这个功能的作用就是让开发者能够立即运行自己的程序。具体点说，就是不用再像以前那样每次修改完代码都要重新构建整个 app，而是可以直接点击运行，修改的代码就可以作用于当前的 app。

在论文编写期间，Google 举办的 2016 大会上面官方提供了一种热修复的方法，Instant APPS 用户只需要点击一个链接就可以动态化地打开一个应用的部分内容。前提是应用的代码是高度组件化。目前正在研究这项技术的原理。

1.3 研究内容

本次研究的目标为以软件工程的方式设计并实现一套 bug 热修复 SDK。

Android 中有三个 ClassLoader, 分别为 URLClassLoader、PathClassLoader、DexClassLoader。其中 URLClassLoader 只能用于加载 jar 文件，但是由于 dalvik 不能直接识别 jar，所以在 Android 中无法使用这个加载器。PathClassLoader 它只能加载已经安装的 apk。因为 PathClassLoader 只会去读取 /data/dalvik-cache 目录下的 dex 文件。例如我们安装一个包名为 com.hujiang.xxx 的 apk,那么当 apk 安装过程中，就会在/data/dalvik-cache 目录下生产一个名为 data@app@com.hujiang.xxx-1.apk@classes.dex 的 ODEX 文件。在使用 PathClassLoader 加载 apk 时，它就会去这个文件夹中找相应的 ODEX 文件，如果 apk 没有安装，自然会报 ClassNotFoundException。

DexClassLoader 是最理想的加载器。它的构造函数包含如下参数，分别为：

(1) dexPath,指目标类所在的 APK 或 jar 文件的路径.类装载器将从该路径中寻找指定的目标类,该类必须是 APK 或 jar 的全路径.如果要包含多个路径,路径之间必须使用特定的分割符分隔,特定的分割符可以使用 System.getProperty(“path.separtor”)获得.

(2) dexOutputDir,由于 dex 文件被包含在 APK 或者 Jar 文件中,因此在装载目标类之前需要先从 APK 或 Jar 文件中解压出 dex 文件,该参数就是制定解压出的 dex 文件存放的路径.在 Android 系统中,一个应用程序一般对应一个 Linux 用户 id,应用程序仅对属于自己的数据目录路径有写的权限,因此,该参数可以使用该程序的数据路径。

(3) libPath,指目标类中所使用的 C/C++库存放的路径。

可以看到 dexElements 注释，dexElements 就是一个 dex 列表，可以把每个 Element 当成是一个 dex。DexClassLoader 包含有一个 dex 数组 Element[] dexElements，其中每个 dex 文件是一个 Element，当需要加载类的时候会遍历 dexElements，如果找到类则加载，如果找不到从下一个 dex 文件继续查找。

那么热修复的实现就是把这个插件 dex 插入到 Elements 的最前面，这么做的好处是不仅可以动态的加载一个类，并且由于 DexClassLoader 会优先加载靠前的类，同时实现了宿主 apk 的热修复功能。

类被动态加载之后，还需要考虑三个问题：

(1) android 中许多组件类是需要在 Manifest 文件里面注册后才能工作的，所以即使动态加载了一个新的组件类进来，没有注册的话还是无法工作。

(2) Res 资源。在编译时期，资源与 R.id 对应好。运行时通过这些 id 从 Resource 实例中获取对应的资源。如果是运行时动态加载进来的新类，用到 R.id 的地方将会抛出异常。核心要解决的问题是：如何给外部的新类提供上下文环境？

(3). 阻止类打上 CLASS_ISPREVERIFIED 的标志。否则加载其他 dex 的时候会报错。当一个 apk 在安装的时候，apk 中的 classes.dex 会被虚拟机(dexopt)优化成 odex 文件，然后

才会拿去执行。虚拟机在启动的时候，会有许多的启动参数，其中一项就是 verify 选项，当 verify 选项被打开的时候，就会执行 dvmVerifyClass 进行类的校验，如果 dvmVerifyClass 校验类成功，那么这个类会被打上 CLASS_ISPREVERIFIED 的标志。

Sdk 提供：patch 的安全下载，安全运行，动态替换 bug 所在类，在用户无感知的情况下完成 bug 的修复。

2. 需求分析

2.1 系统需求概述

在应用编写完成之前，sdk 需要为应用提供一个完整的文件下载，校验，接入的功能。随着应用编写完成，发布到应用市场。出现 bug 后，首先分析 bug 出现的原因，整理解决方案；然后，打包生成补丁，并且管理好之前的补丁版本更替，把补丁托管到服务器；应用本地判断服务器是否有补丁文件，下载下来之后，判断补丁的版本信息，然后进行完整性校验；最后查找应用内部类型文件的信息并且动态替换掉对应的类型信息。

2.2 系统功能分析

本部分将对系统所要实现的功能进行分析。分析系统的用例可知，系统整体功能可以分为文件操作、网络加载、安全校验、动态替换。如系统功能图所示：

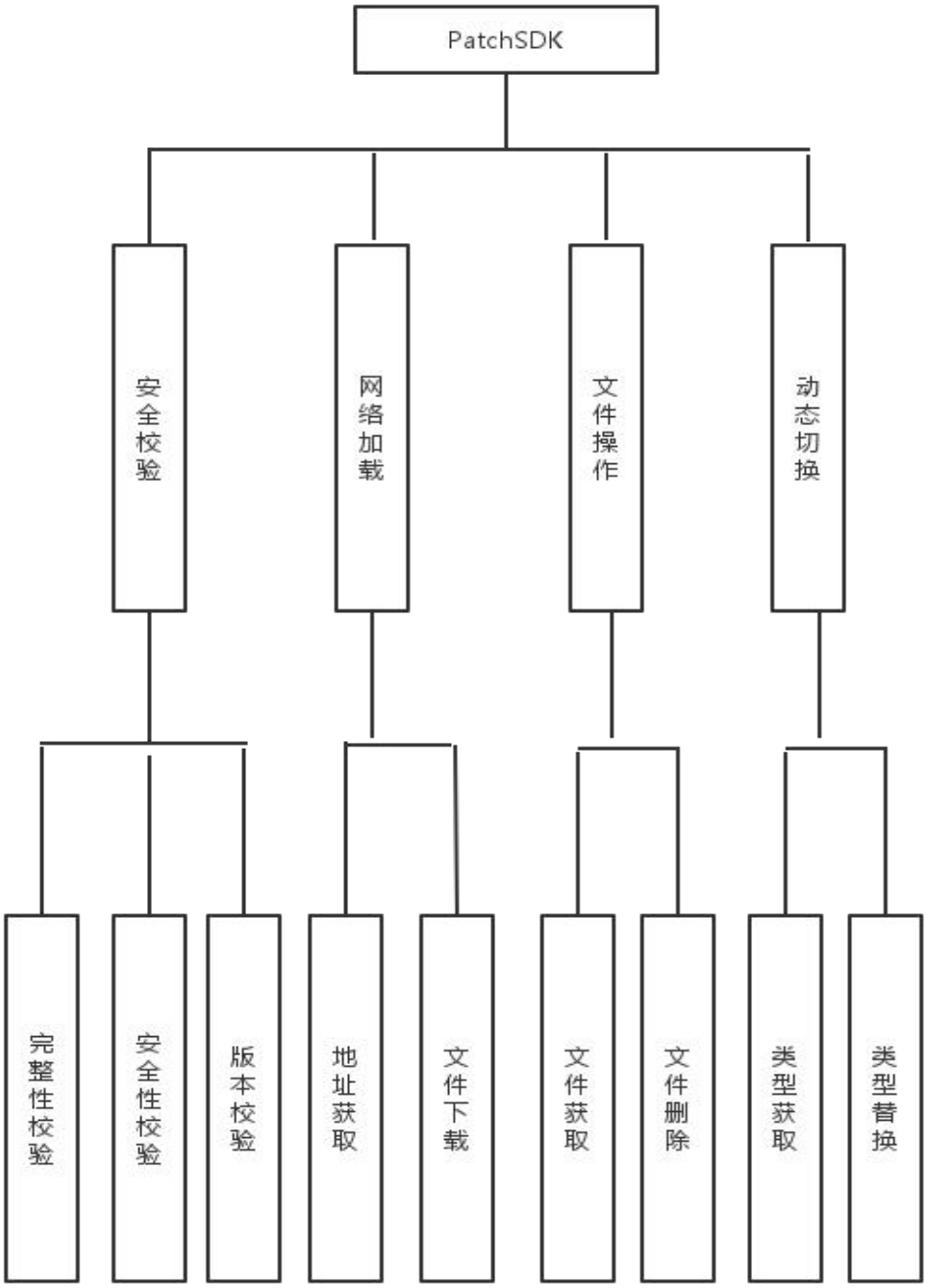

图 1 系统功能图

网络加载：向服务器请求加载补丁的操作。网络支持 SPDY、连接池、GZIP 和 HTTP 缓存。会自动处理常见的网络问题，像二次连接、SSL 的握手问题。OkHttp 使用 Okio 来大大简化数据的访问与存储封装了 okhttp 的网络框架，目前 Get，Post 的请求已经完成，支持大文件上传下载，上传进度回调，下载进度回调，表单上传（多文件和多参数一起上传），链式调用，整合 Gson，自动解析返回对象，支持 Https 和自签名证书，支持 cookie 自动管理，后期将要实现的功能，统一的上传管理和下载管理。

安全校验：对 patch 代码进行加密避免被窃取。对 patch 代码的所有者进行校验防止执行其他的 patch 代码。

文件操作：通过 jni 层将 patch 代码解密并存储在应用私有目录下。

动态切换：运行时通过反射动态覆盖 bug 方法。负责系统运行时类型信息的查找，包括应用中使用到的所有的类信息。然后动态地覆盖掉有 bug 的类，使运行时的类是正确的类信息。

2.3 Bug 修复流程分析

发现一个线上 bug 后，一般的处理流程是：首先分析 bug 出现的原因，整理解决方案；然后，打包生成补丁，并且管理好之前的补丁版本更替，把补丁托管到服务器；应用本地判断服务器是否有补丁文件，下载下来之后，保存到应用私有文件目录下判断补丁的版本信息，然后进行完整性校验；最后查找应用内部类型文件的信息并且动态替换掉对应的类型信息。

2.4 三个关键问题

android 中许多组件类是需要在 Manifest 文件里面注册后才能工作的，所以即使动态加载了一个新的组件类进来，没有注册的话还是无法工作。另外，最重要的一点：生命周期如何控制。目前还真的没什么办法能够处理这个问题，一个 Activity 的启动，如果不采用标准的 Intent 方式，没有经历过 Android 系统 Framework 层级的一系列初始化和注册过程，它的生命周期方法是不会被系统调用的（除非开发者能够修改 Android 系统的一些代码，而这已经是另一个领域的话题了，这里不展开）。那把插件 APK 里所有 Activity 都注册到主项目的 Manifest 里，再以标准 Intent 方式启动。但是事先主项目并不知道插件 Activity 里会新增哪些 Activity，如果每次有新加的 Activity 都需要升级主项目的版本，那不是本末倒置了，不如把插件的逻辑直接写到主项目里来得方便。那就绕绕弯吧，生命周期不就是系统对 Activity 一些特定方法的调用嘛，可以在主项目里创建一个 ProxyActivity，再由它去代理调用插件 Activity 的生命周期方法（这也是代理模式叫法的由来）。用 ProxyActivity（一个标准的 Activity 实例）的生命周期同步控制插件 Activity（普通类的实例）的生命周期，同步的方式可以有下面两种：在 ProxyActivity 生命周期里用反射调用插件 Activity 相应生命周期的方法，简单粗暴。把插件 Activity 的生命周期抽象成接口，在 ProxyActivity 的生命周期里调用。另外，多了这一层接口，也方便主项目控制插件 Activity。这里补充说明下，Fragment 自带生命周期，用 Fragment 来代替 Activity 开发可以省去大部分生命周期的控制工作，但是会使得界面跳转比较麻烦，而且 Honeycomb 以前没有 Fragment，无法在 API11 以前的系统使用。

Res 资源。在编译时期，资源与 R.id 对应好。运行时通过这些 id 从 Resource 实例中获取对应的资源。如果是运行时动态加载进来的新类，用到 R.id 的地方将会抛出异常。解决 Res 资源加载的一种方式：首先将插件中的资源解压出来，然后通过文件流去读取资源，这样做理论上是可行的，但是实际操作起来还是有很大难度 的。首先不同资源有不同的文件流格式，比如图片、XML 等，其次针对不同设备加载的资源可能是不一样的，如何选择合适的资源也是一个需要解决的问题，基于这两点，这种方法也不建议使用，因为它实现起来有较大难度。为了方便地对插件进行资源管理，下面给出一种合理的方式。一个进程是可以同时加载多个应用程序的，也就是可以同时加载多个 APK 文件。每一个 APK 文件在进程中都对应有一个全局的 Resourses 对象以及一个全局的 AssetManager 对象。其中，这个全局的 Resourses 对象保存在一个对应的 ContextImpl 对象的成员变量 mResources 中，而这个全局的 AssetManager 对象保存在这个全局的 Resourses 对象的成员变量 mAssets 中。上述 ContextImpl、Resourses 和 AssetManager 的关系如图所示：

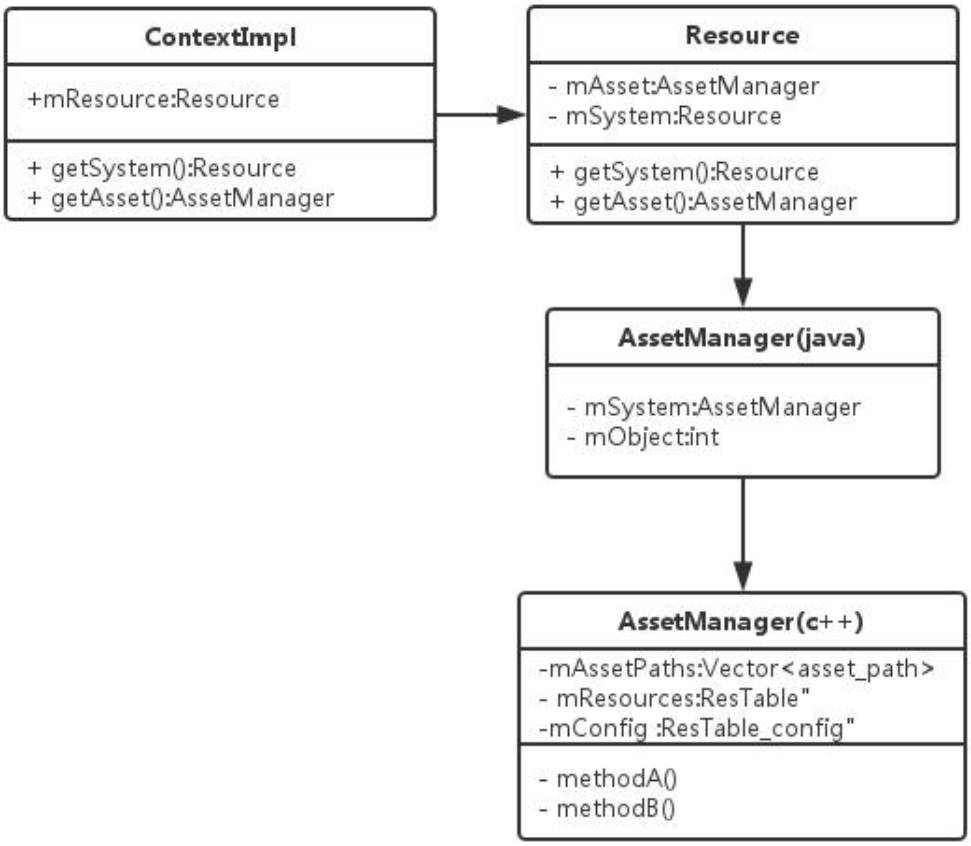

图 2 应用资源关系图

Resources 类有一个成员函数 getAssets，通过它就可以获得保存在 Resources 类的成员变量 mAssets 中的 AssetManager，例如，ContextImpl 类的成员函数 getAssets 就是通过调用其成员变量 mResources 所指向的一个 Resources 对象的成员函数 getAssets 来获得一个可以用来访问应用程序的非编译资源文件的 AssetManager。

Android 应用程序除了要访问自己的资源之外，还需要访问系统的资源。系统的资源打包在/system/framework/framework-res.apk 文件中，它在应用程序进程中是通过一个单独的 Resources 对象和一个单独的 AssetManager 对象来管理的。这个单独的 Resources 对象就保存在 Resources 类的静态成员变量 mSystem 中，然后可以通过 Resources 类的静态成员函数 getSystem 就可以获得这个 Resources 对象，而这个单独的 AssetManager 对象就保存在 AssetManager 类的静态成员变量 sSystem 中，可以通过 AssetManager 类的静态成员函数 getSystem 同样可以获得这个 AssetManager 对象。

AssetManager 类除了在 Java 层有一个实现之外，在 C++层也有一个对应的实现，而 Java 层的 AssetManager 类的功能就是通过 C++层的 AssetManager 类来实现的。Java 层的每一个 AssetManager 对象都有一个类型为 int 的成员变量 mObject，它保存的便是在 C++层对应的 AssetManager 对象的地址，因此，通过这个成员变量就可以将 Java 层的 AssetManager 对象与 C++层的 AssetManager 对象关联起来。

C++层的 AssetManager 类有三个重要的成员变量 mAssetPaths、mResources 和 mConfig。其中，mAssetPaths 保存的是资源存放目录，mResources 指向的是一个资源索引表，而 mConfig 保存的是设备的本地配置信息，例如屏幕密度和大小、国家地区和语言等等配置信息。有了这三个成员变量之后，C++层的 AssetManager 类就可以访问应用程序的资源了。

主要就是创建和初始化用来访问应用程序资源的 AssetManager 对象和 Resources 对象，

其中，初始化操作包括设置 AssetManager 对象的资源文件路径以及设备配置信息等。有了这两个初始化的 AssetManager 对象和 Resources 对象之后，就可以查找应用程序资源了。

创建一个 AssetManager，执行 addAssetPath 方法将资源文件路径加载进去。最后实例化一个 Resource。

虚拟机在安装 app 期间为了提高性能：当一个类只引用当前 dex 的文件，就会打上一个 CLASS_ISPREVERIFIED 标志。在运行的时候如果使用另一个 dex 里面的文件就会抛出异常，直接崩溃！解决这个问题的唯一方法就是防止类被打上这个标志。也就是在目标类上面引用其他 dex 里面的类就可以解决这个问题。方法很简单：先创建一个 java 文件，编译成 class 文件，打包成单独的 dex。之后采用 javassist 在目标类的构造方法中调用该 dex 文件中的类。

2.5 系统用例分析

本系统有一个用例就是项目 app,主要用例有安全校验、网络下载、文件操作、类替换。安全校验包含：完整性校验、安全性校验、版本校验。网络下载包含：地址获取、文件下载。文件操作包括：文件拷贝、文件删除、文件获取。类替换包含：类型获取、类型替换。系统用例图如下：

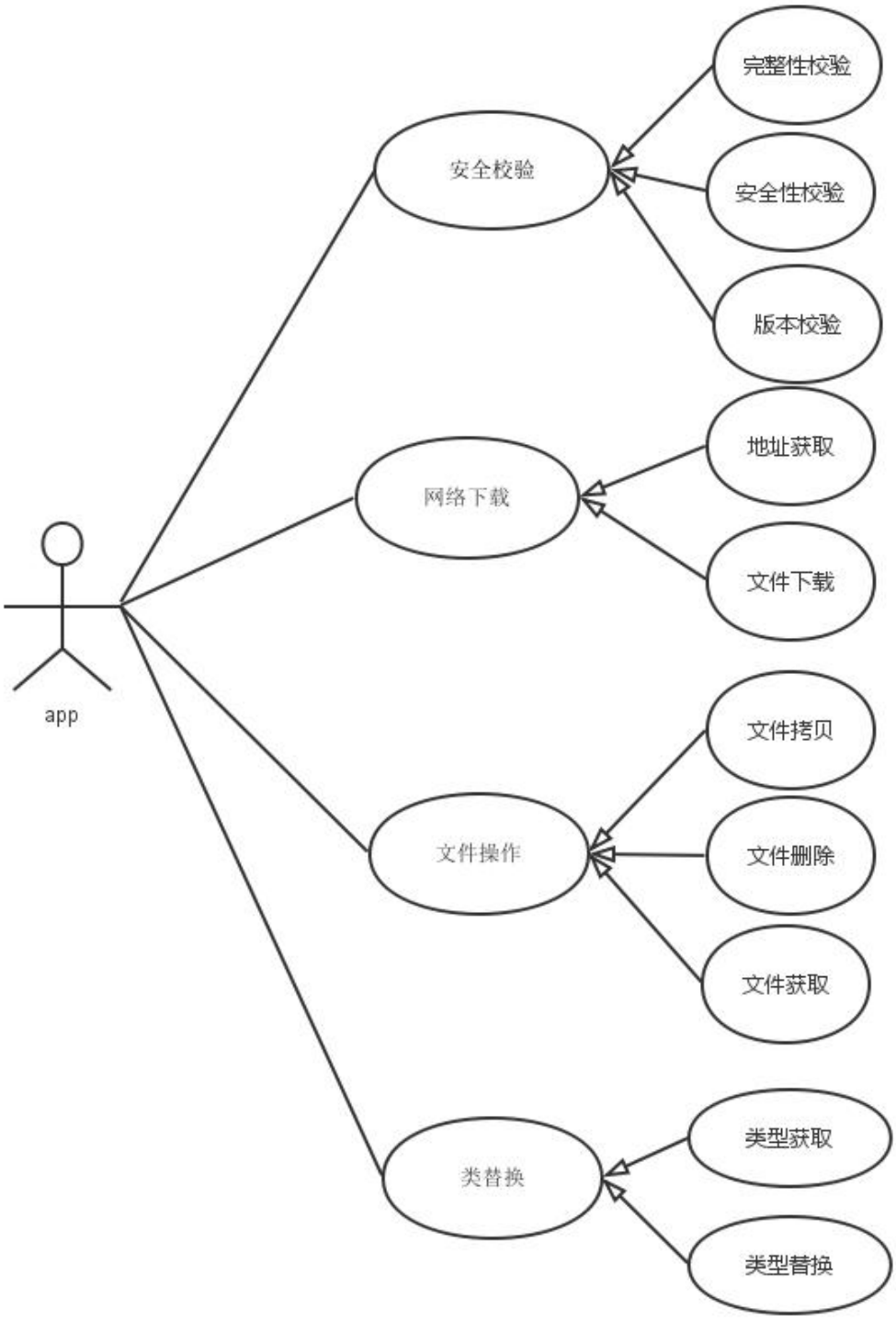

图 3 系统用例图

以下表格为用例说明：

表 1 用例“安全校验”的描述

用例名称	安全校验
标识符	VertifyManager

用例名称	安全校验
用例描述	用于系统对文件的完整性，安全性，版本校验
参与者	系统
优先级	1
状态	进行中
前置条件	用户打开 app
后置条件	无
基本操作流程	打开 app 进行补丁修复
可选操作流程	无
被泛化的用例	无
被包含的用例	1、 完整性校验 2、 安全性校验 3、 版本校验
被扩展的用例	无
修改历史记录	无

表 2 用例“网络下载”的描述

用例名称	网络下载
标识符	DownloadManager
用例描述	用于系统获取 patch 地址，并且下载下来
参与者	系统
优先级	1
状态	进行中
前置条件	用户打开 app
后置条件	无
基本操作流程	打开 app 进行补丁修复
可选操作流程	无
被泛化的用例	无
被包含的用例	1、文件拷贝 2、文件删除 3、文件获取
被扩展的用例	无
修改历史记录	无

表 3 用例“文件操作”的描述

用例名称	文件操作
标识符	FileManager
用例描述	用于系统获取 patch 地址，并且下载下来
参与者	系统
优先级	1
状态	进行中
前置条件	用户打开 app
后置条件	无

用例名称	文件操作
基本操作流程	打开 app 进行补丁修复
可选操作流程	无
被泛化的用例	无
被包含的用例	4、地址获取 5、文件下载
被扩展的用例	无
修改历史记录	无

表 4 用例“类替换”的描述

用例名称	类替换
标识符	PatchManager
用例描述	用于系统获取查找类型信息，动态覆盖
参与者	系统
优先级	1
状态	进行中
前置条件	用户打开 app
后置条件	无
基本操作流程	打开 app 进行补丁修复
可选操作流程	无
被泛化的用例	无
被包含的用例	1、类型查找 2、类型替换
被扩展的用例	无
修改历史记录	无

3 关键原理

3.1 运行时

在 4.4 前，系统采用 Dalcik 虚拟机运行程序。4.4 之后逐步过渡到 ART 运行时。

Zygote 进程中的 Dalvik 虚拟机是从 AndroidRuntime::start 开始创建。主要做了三件事：创建一个 JniInvocation 实例，并且调用它的成员函数 init 来初始化 JNI 环境；调用 AndroidRuntime 类的成员函数 startVm 来创建一个虚拟机及其对应的 JNI 接口，即创建一个 JavaVM 接口和一个 JNIEnv 接口；有了上述的 JavaVM 接口和 JNIEnv 接口之后，就可以在 Zygote 进程中加载指定的 class。

JniInvocation 类的成员函数 init 所做的事情很简单。它首先是读取系统属性 persist.sys.dalvik.vm.lib 的值。前面提到，系统属性 persist.sys.dalvik.vm.lib 的值要么等于 libdvm.so，要么等于 libart.so。因此，接下来通过函数 dlopen 加载到进程来的要么是 libdvm.so，要么是 libart.so。无论加载的是哪一个 so，都要求它导出 JNI_GetDefaultJavaVMInitArgs、JNI_CreateJavaVM 和 JNI_GetCreatedJavaVMs 这三个接口，并且分别保存在 JniInvocation 类的三个成员变量 JNI_GetDefaultJavaVMInitArgs_JNI_CreateJava 和 JNI_GetCreatedJavaVMs_ 中。这三个接口也就是前面提到的用来抽象 Java 虚拟机的三个接口。

```
#ifdef HAVE_ANDROID_OS
static const char* kLibrarySystemProperty = "persist.sys.dalvik.vm.lib";
#endif
static const char* kLibraryFallback = "libdvm.so";

bool JniInvocation::Init(const char* library) {
#ifdef HAVE_ANDROID_OS
  char default_library[PROPERTY_VALUE_MAX];
  property_get(kLibrarySystemProperty, default_library, kLibraryFallback);
#else
  const char* default_library = kLibraryFallback;
#endif
  if (library == NULL) {
    library = default_library;
  }

  handle_ = dlopen(library, RTLD_NOW);
  if (handle_ == NULL) {
    if (strcmp(library, kLibraryFallback) == 0) {
      // Nothing else to try.
      ALOGE("Failed to dlopen %s: %s", library, dlerror());
      return false;
    }
    // Note that this is enough to get something like the zygote
    // running, we can't property_set here to fix this for the future
    // because we are root and not the system user. See
    // RuntimeInit.commonInit for where we fix up the property to
    // avoid future fallbacks. http://b/11463182
    ALOGW("Falling back from %s to %s after dlopen error: %s",
```

```
            library, kLibraryFallback, dlerror());
      library = kLibraryFallback;
      handle_ = dlopen(library, RTLD_NOW);
      if (handle_ == NULL) {
        ALOGE("Failed to dlopen %s: %s", library, dlerror());
        return false;
      }
    }
    if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetDefaultJavaVMInitArgs_),
                    "JNI_GetDefaultJavaVMInitArgs")) {
      return false;
    }
    if (!FindSymbol(reinterpret_cast<void**>(&JNI_CreateJavaVM_),
                    "JNI_CreateJavaVM")) {
      return false;
    }
    if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetCreatedJavaVMs_),
                    "JNI_GetCreatedJavaVMs")) {
      return false;
    }
    return true;
  }
```

AndroidRuntime 类的成员函数 startVm 最主要就是调用函数 JNI_CreateJavaVM 来创建一个 JavaVM 接口及其对应的 JNIEnv 接口。

```
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  return JniInvocation::GetJniInvocation().JNI_CreateJavaVM(p_vm, p_env, vm_args);
}
```

JniInvocation 类的静态成员函数 GetJniInvocation 返回的便是前面所创建的 JniInvocation 实例。有了这个 JniInvocation 实例之后，就继续调用它的成员函数 JNI_CreateJavaVM 来创建一个 JavaVM 接口及其对应的 JNIEnv 接口。

```
jint JniInvocation::JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  return JNI_CreateJavaVM_(p_vm, p_env, vm_args);
}
```

JniInvocation 类的成员变量 JNI_CreateJavaVM_指向的就是前面所加载的 libdvm.so 或者 libart.so 所导出的函数 JNI_CreateJavaVM，因此，JniInvocation 类的成员函数 JNI_CreateJavaVM 返回的 JavaVM 接口指向的要么是 Dalvik 虚拟机，要么是 ART 虚拟机。

通过上面的分析，就很容易得出，Android 系统通过将 ART 运行时抽象成一个 Java 虚拟机，以及通过系统属性 persist.sys.dalvik.vm.lib 和一个适配层 JniInvocation，就可以无缝地将 Dalvik 虚拟机替换为 ART 运行时。这个替换过程设计非常巧妙，因为涉及到的代码修改是非常少的[1]。

以上就是 ART 虚拟机的启动过程，接下来分析应用程序在安装过程中将 dex 字节码翻译为本地机器码的过程。

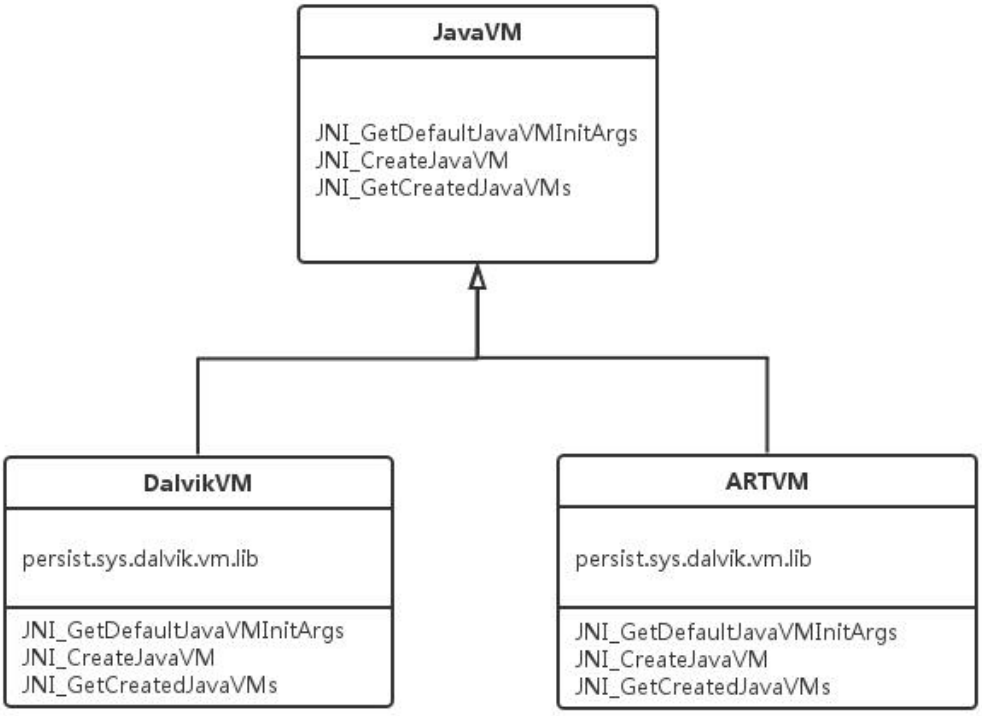

图 4 虚拟机关系图

从图可以知道，Dalvik 虚拟机和 ART 虚拟机都实现了三个用来抽象 Java 虚拟机的接口：

(1). JNI_GetDefaultJavaVMInitArgs -- 获取虚拟机的默认初始化参数

(2). JNI_CreateJavaVM -- 在进程中创建虚拟机实例

(3). JNI_GetCreatedJavaVMs -- 获取进程中创建的虚拟机实例

在 Android 系统中，Davik 虚拟机实现在 libdvm.so 中，ART 虚拟机实现在 libart.so 中。也就是说，libdvm.so 和 libart.so 导出了 JNI_GetDefaultJavaVMInitArgs、JNI_CreateJavaVM 和 JNI_GetCreatedJavaVMs 这三个接口，供外界调用。 此外，Android 系统还提供了一个系统属性 persist.sys.dalvik.vm.lib，它的值要么等于 libdvm.so，要么等于 libart.so。当等于 libdvm.so 时，就表示当前用的是 Dalvik 虚拟机，而当等于 libart.so 时，就表示当前用的是 ART 虚拟机。

以上描述的 Dalvik 虚拟机和 ART 虚拟机的共同之处。不同的地方就在于，Dalvik 虚拟机执行的是 dex 字节码，ART 虚拟机执行的是本地机器码。这意味着 Dalvik 虚拟机包含有一个解释器，用来执行 dex 字节码，当然，Android 从 2.2 开始，也包含有 JIT（Just-In-Time），用来在运行时动态地将执行频率很高的 dex 字节码翻译成本地机器码，然后再执行。通过 JIT，就可以有效地提高 Dalvik 虚拟机的执行效率。但是，将 dex 字节码翻译成本地机器码是发生在应用程序的运行过程中的，并且应用程序每一次重新运行的时候，都要做重做这个翻译工作的。因此，即使用采用了 JIT，Dalvik 虚拟机的总体性能还是不能与直接执行本地机器码的 ART 虚拟机相比。

那么，ART 虚拟机执行的本地机器码是从哪里来的呢？Android 的运行时从 Dalvik 虚拟机替换成 ART 虚拟机，并不要求开发者要将重新将自己的应用直接编译成目标机器码。也就是说，开发者开发出的应用程序经过编译和打包之后，仍然是一个包含 dex 字节码的 APK 文件。而 ART 虚拟机需要的是本地机器码，这就必然要有一个翻译的过程。这个翻译的过程当然不能发生应用程序运行的时候，否则的话就和 Dalvik 虚拟机的 JIT 一样了。在计算机的世界里，与 JIT 相对的是 AOT。AOT 进 Ahead-Of-Time 的简称，它发生在程序运行之前。静态语言（例如 C/C++）开发的应用程序，编译器直接就把它们翻译成目标机器码。这种静

态语言的编译方式也是 AOT 的一种。但是前面提到，ART 虚拟机并不要求开发者将自己的应用直接编译成目标机器码。这样，将应用的 dex 字节码翻译成本地机器码的最恰当 AOT 时机就发生在应用安装的时候。

没有 ART 虚拟机之前，应用在安装的过程，其实也会执行一次“翻译”的过程。只不过这个“翻译”的过程是将 dex 字节码进行优化，也就是由 dex 文件生成 odex 文件。这个过程由安装服务 PackageManagerService 请求守护进程 installd 来执行的。从这个角度来说，在应用安装的过程中将 dex 字节码翻译成本地机器码对原来的应用安装流程基本上就不会产生什么影响。

3.2 Multidex 分包

伴随着 Android 平台的继续增长，Android 应用程序的尺寸也在变大。当应用程序或者引用库达到一定规模，开发者会碰到一个 build 错误：应用到达了 Android 应用构架架构的最大限制。

Android 应用程序包含的可执行文件 Dex,其中包含用于运行开发者的应用程序的可执行字节码文件。Dalvik 可执行规范限制了可以在单个 dex 文件内引用 65536：包括 Android 框架方法，库方法，并在自己的代码中实现的方法总数。突破这个限制需要配置开发者的应用程序构建过程，生成多个 dex 文件，被称为 multidex 配置。

通过查看 MultiDex 的源码，可以发现 MultiDex 在冷启动时因为需要安装 Dex 文件，如果 Dex 文件过大时，处理时间过长，很容易引发 ANR 采用 MultiDex 方案的应用因为 linearAlloc 的 BUG，可能不能在 2.x 设备上启动。

采用 Google 的方案开发者不需要关心 Dex 分包，开发工具会自动的分析依赖关系，把需要的 class 文件及其依赖 class 文件放在 Main Dex 中，因此如果产生了多个 Dex 文件，那么 classes.dex 内的方法数一般都接近 65535 这个极限，剩下的 class 才会被放到 Other Dex 中。如果开发者可以减小 Main Dex 中的 class 数量，是可以加快冷启动速度的。

通过研究源码发现 Multidex 的实现是通过在 Applicaion 实例化之后，会检查系统版本是否支持 multidex，classes2.dex 是否需要安装。如果需要安装则会从 APK 中解压出 classes2.dex 并将其拷贝到应用的沙盒目录下。通过反射将 classes2.dex 注入到当前的 classloader 中。

3.3 加载器 Classloader

在运行时将会有多个 classloader 被加载进来。通常情况下，类加载器分为逻辑上的树，子类加载器委托给父类加载器所有请求。只有当父类加载器不能满足要求时，子类加载器才会试图加载。

classloader 是一个实现类加载器通用功能的抽象类。

加载类的实例时，先查询当前 classloader 是否加载过？有，就直接返回；没有，则查询父类是否加载过？有，则返回；没有，就执行当前 classloader 的 findclass 方法。这样做的结果是：如果类被父类 ClassLoader，那么之后整个系统的生命周期中，永远不会再加载。所以想要实现自定义加载 class，需要复写 ClassLoader 的 loadclass()。作用：共享功能，一些 Framework 层级的类一旦被顶层的 ClassLoader 加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。除此之外还有隔离功能，不同继承路线上的 ClassLoader 加载的类肯定不是同一个类，这样的限制避免了用户自己的代码冒充核心类库的类访问核心类库包可见成员的情况[2]。

```
    protected Class<?> loadClass(String className, boolean resolve) throws
ClassNotFoundException {
            Class<?> clazz = findLoadedClass(className);

            if (clazz == null) {
                ClassNotFoundException suppressed = null;
```

```
                try {
                    clazz = parent.loadClass(className, false);
                } catch (ClassNotFoundException e) {
                    suppressed = e;
                }

                if (clazz == null) {
                    try {
                        clazz = findClass(className);
                    } catch (ClassNotFoundException e) {
                        e.addSuppressed(suppressed);
                        throw e;
                    }
                }
            }

            return clazz;
        }
```

ClassLoader 的内部类 BootClassLoader--系统唯一明确的根 ClassLoader。显然 BootClassLoader 实现了 ClassLoader 的 loadClass()。BootClassLoader 将会通过 Class 类的 native 方法去加载框架需要的类[3]。

静态内部类 SystemClassLoader,创建一个 PathClassLoader,并且指定父加载器为 BootClassLoader。

Android 提供了几个具体的实现：

BaseDexClassLoader.加载 dex 文件的父类，提供基础功能。没有重写 ClassLoader 的 loadclass()，说明依然遵循双亲委托机制。

(1).返回给定 package 的信息。

```
       @Override
        protected synchronized Package getPackage(String name) {
            if (name != null && !name.isEmpty()) {
                Package pack = super.getPackage(name);
                if (pack == null) {
                    pack = definePackage(name, "Unknown", "0.0", "Unknown", "Unknown", "0.0",
"Unknown", null);
                }
                return pack;
            }
            return null;
        }
```

(2).重写 findClass(),通过 pathList 去查找类的方法。

```
      @Override
        protected Class<?> findClass(String name) throws ClassNotFoundException {
            List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
            Class c = pathList.findClass(name, suppressedExceptions);
            if (c == null) {
                ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \""
+ name + "\" on path: " +
```

```
    pathList);
                for (Throwable t : suppressedExceptions) {
                    cnfe.addSuppressed(t);
                }
                throw cnfe;
            }
            return c;
        }
```

(3).遍历 dex 或 resource 的列表。通过 DexFile 的 loadClassBinaryName()返回 Class。将 Class 的 name 与 ClassLoader 进行绑定，唯一确定一个类。PathClassLoader, 继承 BaseDexClassLoader.系统默认的加载安装应用中声明的类的 ClassLoader。

(1).加载包含类和资源的 jar 包或者 apk 文件

```
public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }
```

(2).加载包含 dex 的 JAR、ZIP、APK 文件或者单独的 dex 文件。另外还有一个 native 的 library 文件。

```
public PathClassLoader(String dexPath, String libraryPath, ClassLoader parent) {
        super(dexPath, null, libraryPath, parent);
    }
```

(3)DexClassLoader,继承 BaseDexClassLoader.加载不是应用中的代码。存放在应用私有文件目录下，读取 jar、apk 文件的列表，优化缓存到指定文件夹下。

```
public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), libraryPath, parent);
    }
```

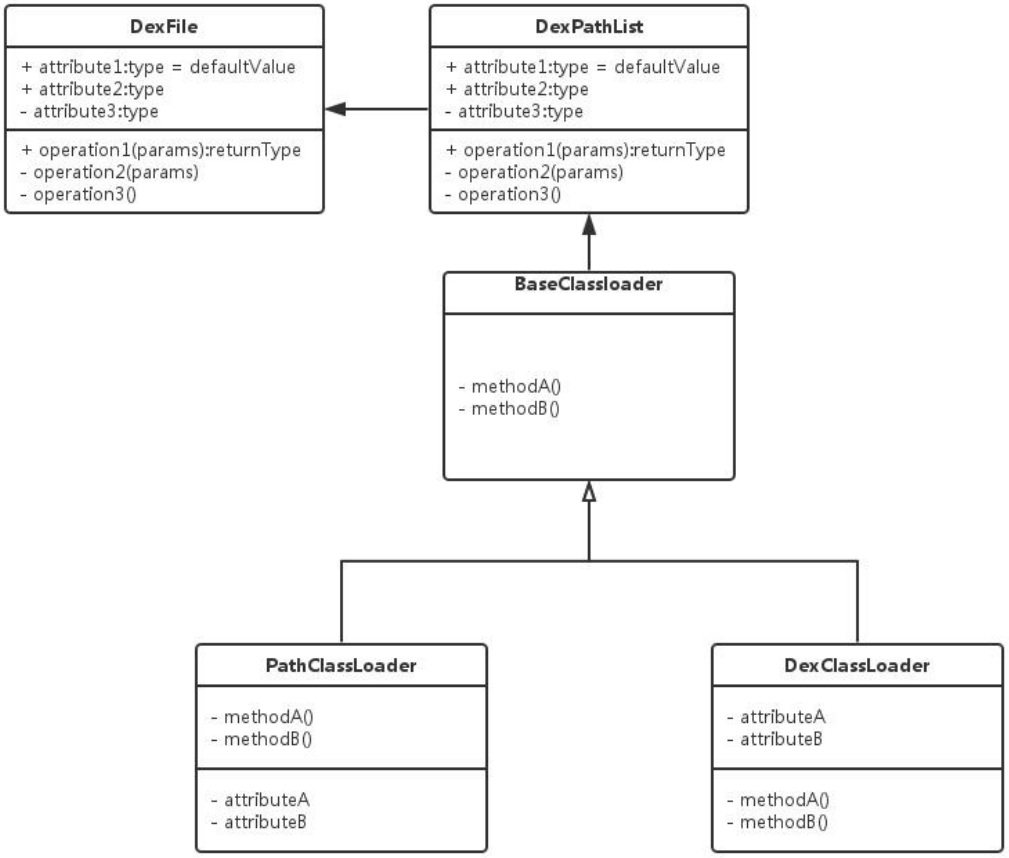

图 5 ClassLoader 关系图

两者区别：

PathClassLoader 是通过 new DexFile(path)来产生 DexFile 对象。DexClassLoader 通过 DexFile 通过静态方法 LoadDex(path,outputpath,0) 得到 DexFile 对象。也就是说，PathClassLoader 不能主动从 zip 包中释放 dex,只支持操作 dex 文件；而 DexClassLoader 可以支持.apk、.jar 和.dex 文件，并且会在指定的 outpath 路径释放出 dex 文件[4-5]。

```
       private static DexFile   loadDexFile(File file, File optimizedDirectory) throws
IOException {
           if (optimizedDirectory == null) {
               return new DexFile(file);
           } else {
               String optimizedPath = optimizedPathFor(file, optimizedDirectory);
               return DexFile.loadDex(file.getPath(), optimizedPath, 0);
           }
   }
```

应用运行加载一个类是这样的过程：BootClassLoader 加载基础类，如 java.lang.String。在这个过程中，通过 BaseClassLoader 的成员变量 DexPathList 去后者的成员变量 Element 的数组中加载类，并且是按照顺序去读取，并且只有在真正的从 dex 加载的时候才会指定类所对应的 ClassLoader。

同理：系统加载 native library 时的流程相似：

```
    @Override
       public String findLibrary(String name) {
           return pathList.findLibrary(name);
```

```
}
   /**
     * List of native library directories.
     */
private final File[] nativeLibraryDirectories;
public String findLibrary(String libraryName) {
        String fileName = System.mapLibraryName(libraryName);
        for (File directory : nativeLibraryDirectories) {
            String path = new File(directory, fileName).getPath();
           if (IoUtils.canOpenReadOnly(path)) {
              return path;
           }
        }
        return null;
}
```

如果能做到将修复的 dex 文件所在的 Element 或者 nativeLibraryDirectories 在 bug 类所在 Element 或者 nativeLibraryDirectories 之前，所有问题就迎刃而解。这里有很多并且，这些并且非一不可，成为了此次能够热修复成功的关键。

4. 系统设计

4.1 系统总体类图设计

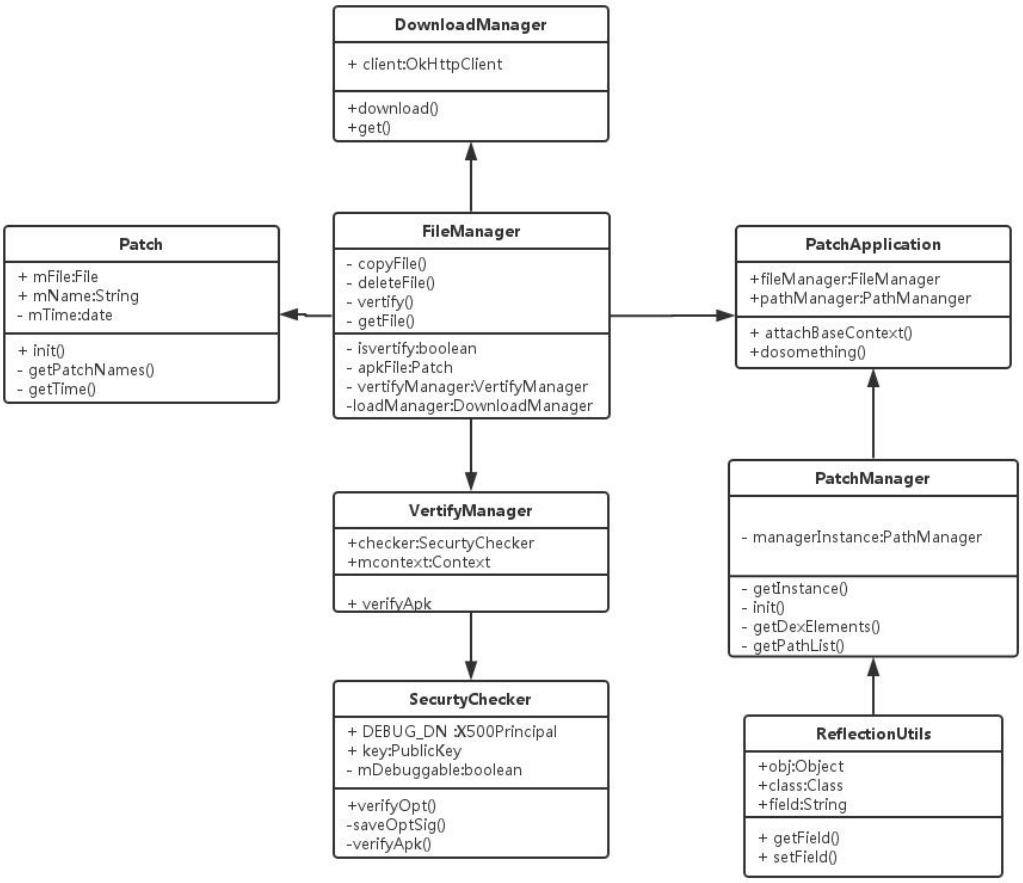

图 6 系统总类图

系统核心实现主要是四个管理类和相应的工具类。DownLoadManager 负责网络加载补丁；VertifyMnager 负责校验方面的操作；FileManager 负责补丁文件相关的操作；PatchManager 负责替换的主要逻辑。其中一个比较重要的工具类 ReflectionUtils，负责通过反射获取类型的相关信息，并且提供修改类型的属性信息功能[6]。Patch 负责标识补丁的基本信息等。

表 5 系统类图信息表

类名	类说明
PatchApplicaiton	系统入口，负责调度文件下载、校验、与类型替换工作
PatchManager	补丁管理类，负责类型替换
ReflectionUtils	通过反射获取类型的 field,并且可以动态替换属性
FileMangaer	负责文件的复制，删除，获取
VertifyManager	负责文件的完整性校验、安全性校验、版本检测
DownloadManager	负责文件地址获取与文件下载
SecurtyChecker	负责文件校验
Patch	补丁文件的基本信息：文件内容，文件日期，文件名称

4.2 系统详细设计

主要按照系统功能模块的划分，通过各个功能模块的类图、时序图设计详细介绍相应的原理。 系统主要分为四个模块：网络加载模块，文件下载模块，文件校验模块，动态替换模块。最后再简要介绍下与服务器的交互模块。

4.2.1 网络加载模块

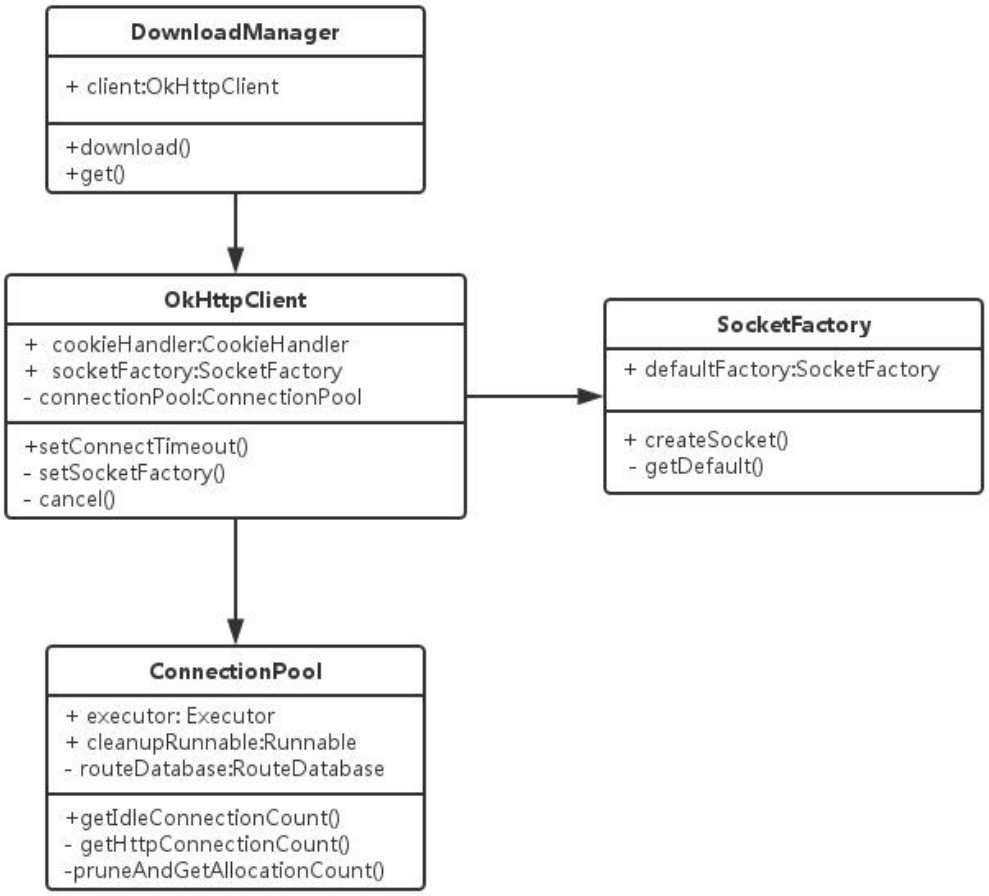

图 7 网络加载类图

本模块使用 Google 官方提供的网络访问工具--OkHttpOkHttpUtils - OkHttp 是一个现代，快速，高效的 Http client，支持 HTTP/2 以及 SPDY，它提供很多功能。诸如连接池，gziping，缓存等就知道网络相关的操作是多么复杂了。OkHttp 扮演着传输层的角色。

OkHttp 使用 Okio 来大大简化数据的访问与存储封装了 okhttp 的网络框架，目前 Get，Post 的请求已经完成，支持大文件上传下载，上传进度回调，下载进度回调，表单上传（多文件和多参数一起上传），链式调用，整合 Gson，自动解析返回对象，支持 Https 和自签名证书，支持 cookie 自动管理，后期将要实现的功能，统一的上传管理和下载管理[7]。

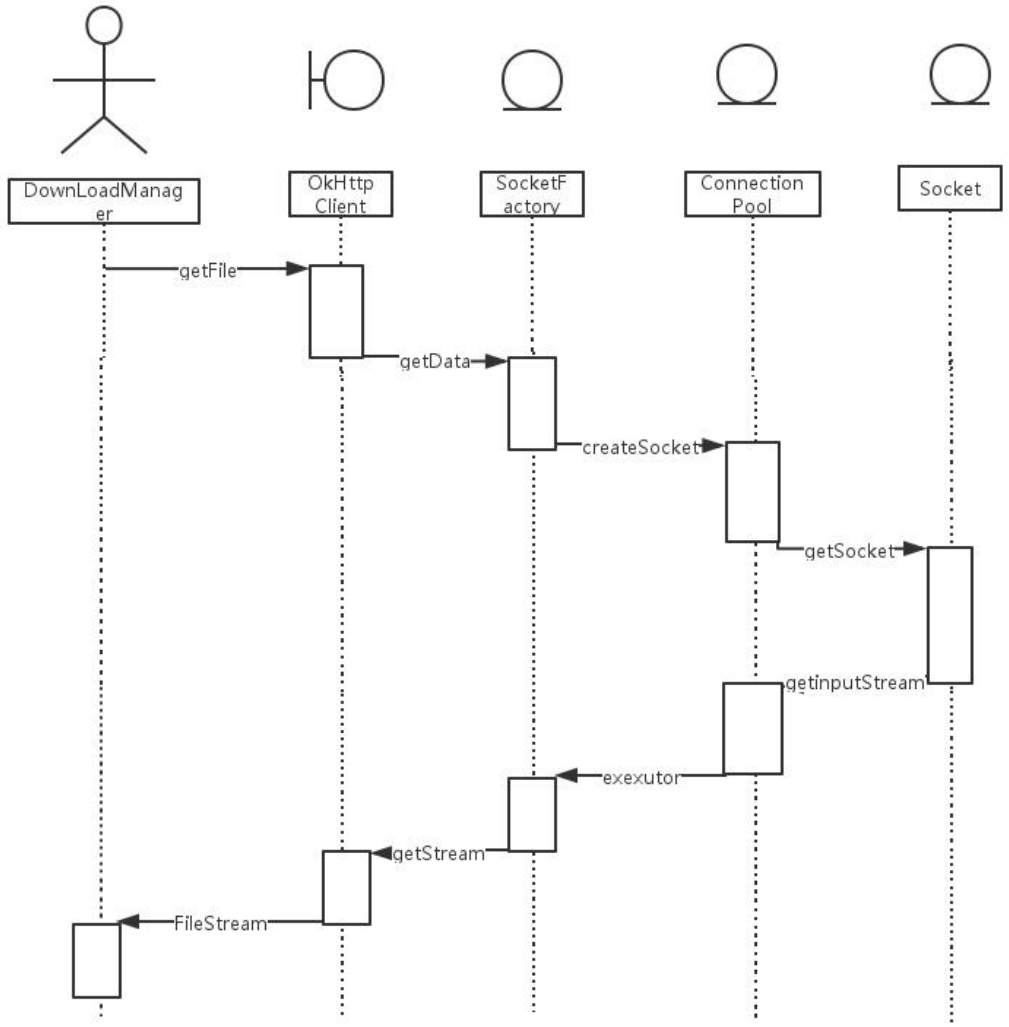

图 8 网络加载时序图

如图，App 获取补丁文件后，创建 OkHttpClient 准备下载文件。既然是 get 请求，当然得先构造好开发者要请求的 URL。有了请求 URL，紧接着就是需要通过这个 URL 构造一个请求对象 Request。当然有时候可能开发者需要对这个 Http 请求添加一些自定义的请求头信息 header，这时开发者在构造 Request 对象之前通过 Request.Builder builder = new Request.Builder() 创建的 builder 对象来添加自己需要添加的请求头信息 builder.addHeader(key, value)。OkHttp 是自带请求缓存控制策略的，如果开发者想改变某个请求的缓存控制策略，开发者也可以通过 builder 对象来修改缓存策略 builder.cacheControl()。通过上述步骤构造好请求对象 Request 之后，通过 OkHttpClient 创建一个 Call 任务对象，这个对象有 execute()和 cancel()等方法对 Call 任务对象进行执行和取消。如果是同步阻塞请求的话，直接执行 Call 对象的 execute()方法即可得到请求结果[8-9]。如果是异步请求的话，就需要执行 Call 对象的 enqueue(new Callback(){})方法，将任务对象添加到任务请求调度队列中，同时添加请求回调接口。请求成功之后，可以得到一个 Response 对象，如果想获得返回的字符串结果则可以通过 response.body().string()，如果想获得返回结果的二进制数据的话可以通过 response.body().bytes()，如果想获得返回的 InputStream 的话可以通过 response.body().byteStream()。

4.2.2 文件操作模块

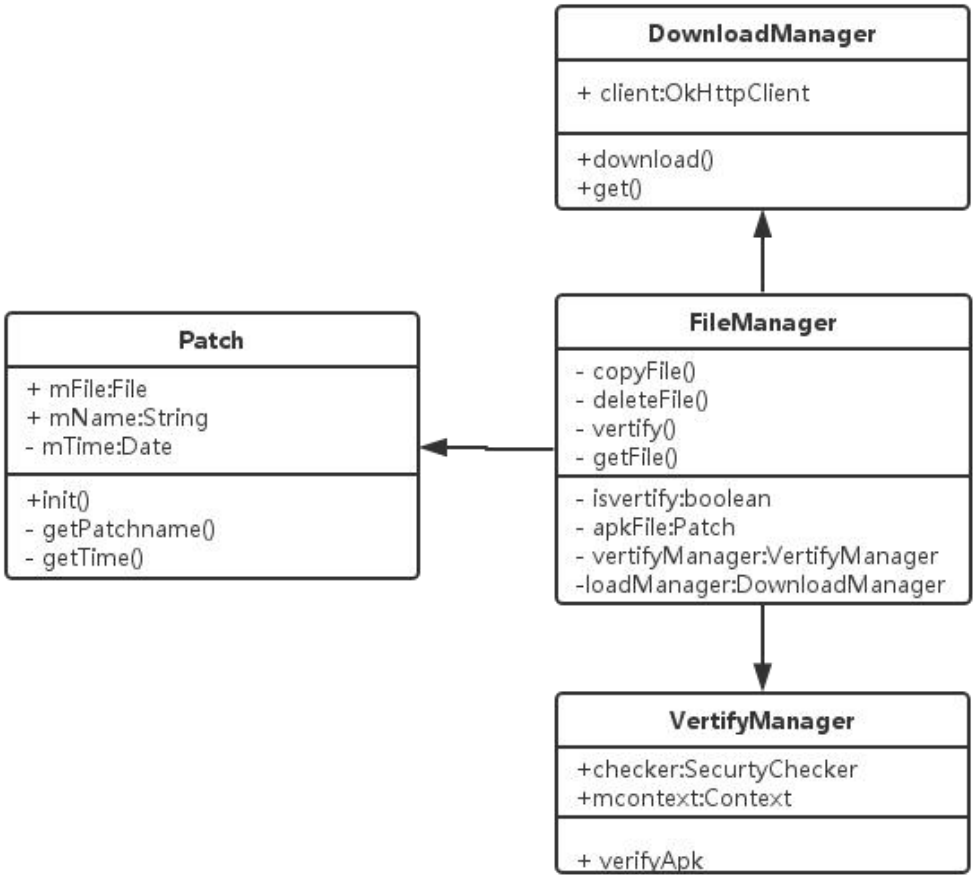

图 9 文件操作类图

如图所示，文件是最常见的数据源之一，在程序中经常需要将数据存储到文件中，例如图片文件、声音文件等数据文件，也经常需要根据需要从指定的文件中进行数据的读取。

内部存储不是内存。内部存储位于系统中很特殊的一个位置，如果开发者想将文件存储于内部存储中，那么文件默认只能被开发者的应用访问到，且一个应用所创建的所有文件都在和应用包名相同的目录下。也就是说应用创建于内部存储的文件，与这个应用是关联起来的。当一个应用卸载之后，内部存储中的这些文件也被删除。从技术上来讲如果开发者在创建内部存储文件的时候将文件属性设置成可读，其他 app 能够访问应用的数据，前提是开发者知道这个应用的包名，如果一个文件的属性是私有（private），那么即使知道包名其他应用也无法访问。 内部存储空间十分有限，因而显得可贵，另外，它也是系统本身和系统应用程序主要的数据存储所在地，一旦内部存储空间耗尽，手机也就无法使用了。所以对于内部存储空间，开发者要尽量避免使用。Shared Preferences 和 SQLite 数据库都是存储在内部存储空间上的[10]。内部存储一般用 Context 来获取和操作。

外部存储，如果说 pc 上也要区分出外部存储和内部存储的话，那么自带的硬盘算是内部存储，U 盘或者移动硬盘算是外部存储，因此开发者很容易带着这样的理解去看待安卓手机，认为机身固有存储是内部存储，而扩展的 T 卡是外部存储。比如开发者任务 16GB 版本的 Nexus 4 有 16G 的内部存储，普通消费者可以这样理解，但是安卓的编程中不能，这 16GB 仍然是外部存储。

所有的安卓设备都有外部存储和内部存储，这两个名称来源于安卓的早期设备，那个时

候的设备内部存储确实是固定的，而外部存储确实是可以像 U 盘一样移动的。但是在后来的设备中，很多中高端机器都将自己的机身存储扩展到了 8G 以上，他们将存储在概念上分成了"内部 internal" 和"外部 external" 两部分，但其实都在手机内部。所以不管安卓手机是否有可移动的 sdcard，他们总是有外部存储和内部存储。最关键的是，开发者都是通过相同的 api 来访问可移动的 sdcard 或者手机自带的存储（外部存储）。

应用获取一个文件之后，先进行文件校验，最后将文件放入应用私有文件目录供之后的动态替换模块使用。

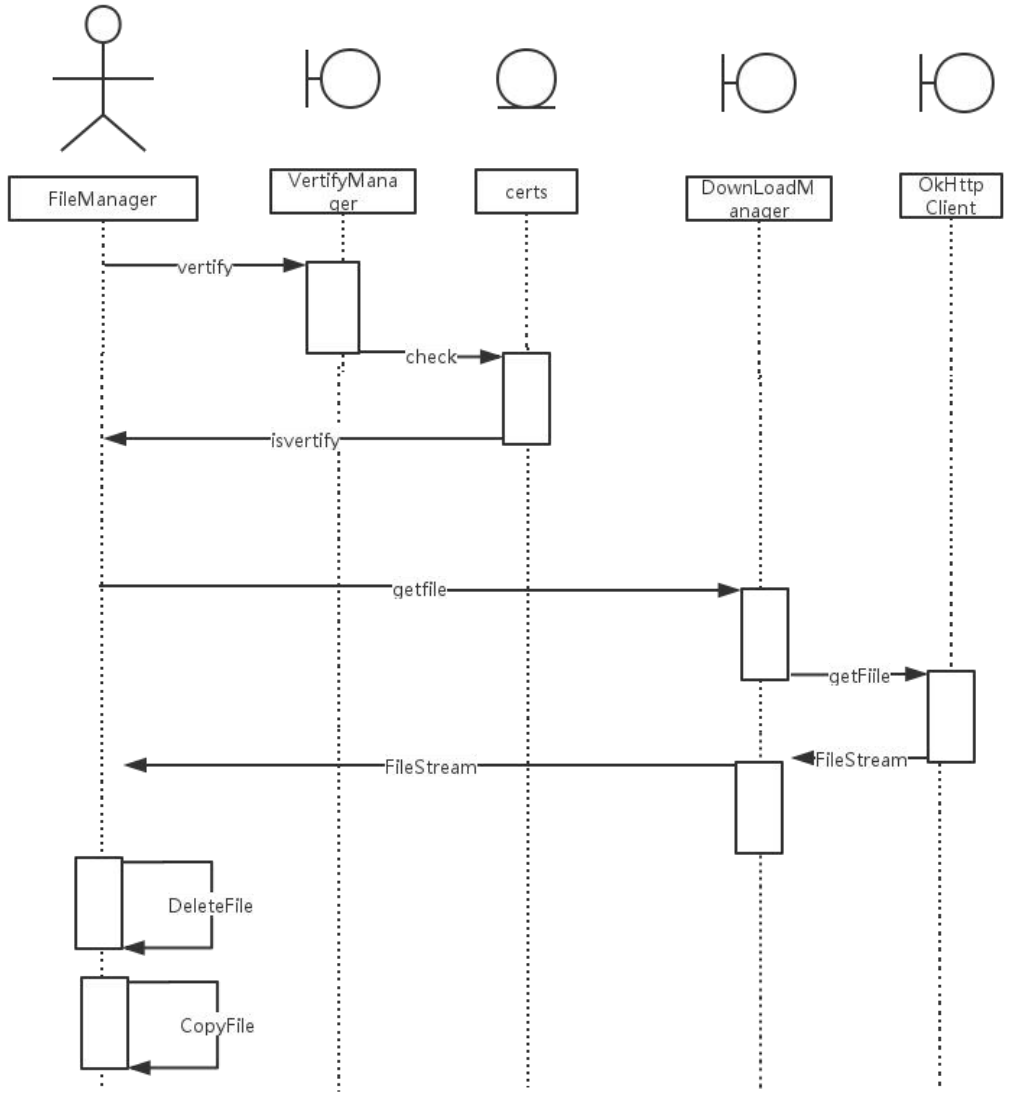

图 10 文件操作时序图

如图，应用获取一个文件之后，VertifyManager 进行文件校验，如果没有通过校验，就将该文件删除掉；如果通过校验，先将信息写入 Patch,之后将文件放入应用私有文件目录保证安全性，并且维护补丁文件的版本管理。应用启动之后，将文件读入内存，供之后的动态替换模块使用。

4. 2. 3 文件校验模块

本模块负责代码校验：包括代码完整性校验、安全校验、版本信息校验。

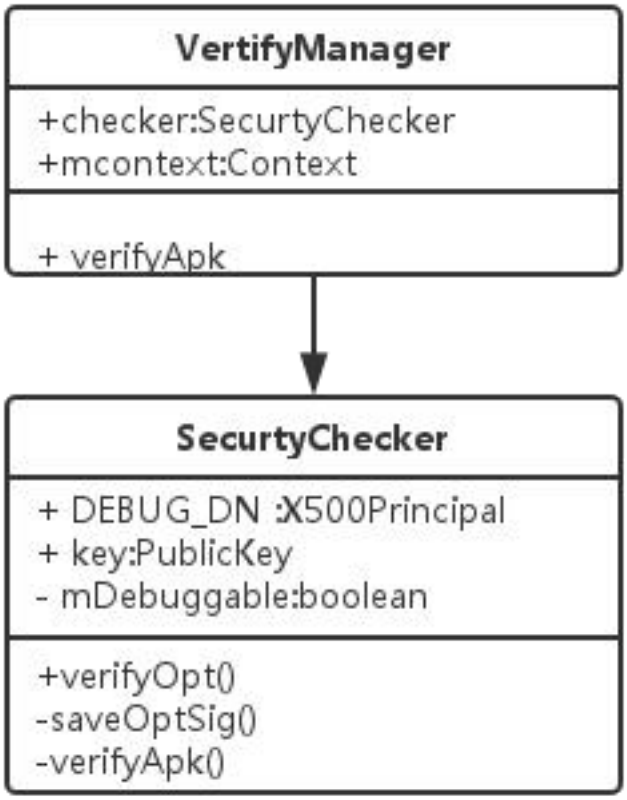

图 11 文件校验类图

Android 对每一个 Apk 文件都会进行签名，在 Apk 文件安装时，系统会对其签名信息进行比对，判断程序的完整性，从而决定该 Apk 文件是否可以安装，在一定程度上达到安全的目的。给定一个包含补丁的 Apk 文件，解压，可以看到一个 META-INFO 文件夹，在该文件夹下有三个文件：分别为 MANIFEST.MF、CERT.SF 和 CERT.RSA。这三个文件分别表征以下含义：

(1).MANIFEST.MF：这是摘要文件。程序遍历 Apk 包中的所有文件(entry)，对非文件夹非签名文件的文件，逐个用 SHA1 生成摘要信息，再用 Base64 进行编码。如果开发者改变了 apk 包中的文件，那么在 apk 安装校验时，改变后的文件摘要信息与 MANIFEST.MF 的检验信息不同，于是程序就不能成功安装。说明：如果攻击者修改了程序的内容，有重新生成了新的摘要，那么就可以通过验证，所以这是一个非常简单的验证。

(2).CERT.SF：这是对摘要的签名文件。对前一步生成的 MANIFEST.MF，使用 SHA1-RSA 算法，用开发者的私钥进行签名。在安装时只能使用公钥才能解密它。解密之后，将它与未加密的摘要信息（即，MANIFEST.MF 文件）进行对比，如果相符，则表明内容没有被异常修改。

(3).CERT.RSA 文件中保存了公钥、所采用的加密算法等信息。META-INFO 里面的说那个文件环环相扣，从而保证补丁程序代码的安全性。

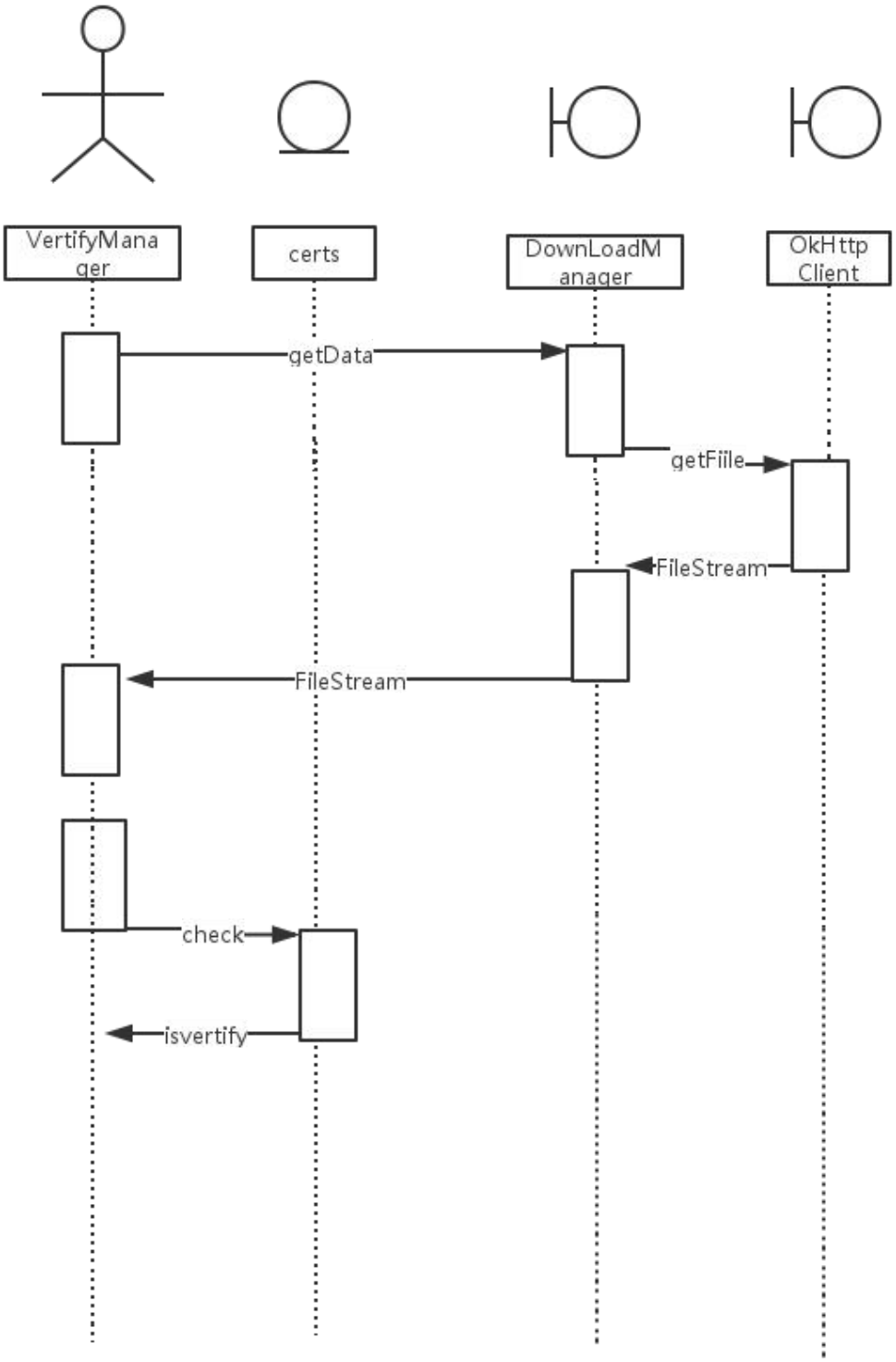

图 12 文件校验时序图

通过代码验证 patch 的完整性。patch 签名信息获取以及两个过签名漏洞检测代码.首先在代码中完成校验值比对的逻辑，此部分代码后续不能再改变，否则 CRC 值会发生变化;从生成的 patch 文件中提取出 classes.dex 文件，计算其 CRC 值，其他 hash 值类似;安全性是为了防止应用加载到其他的代码，然后运行到程序当中造成逻辑错误或者崩溃。版本信息校验是对补丁版本进行控制，方便下载和回滚[11]。

4. 2. 4 动态替换模块

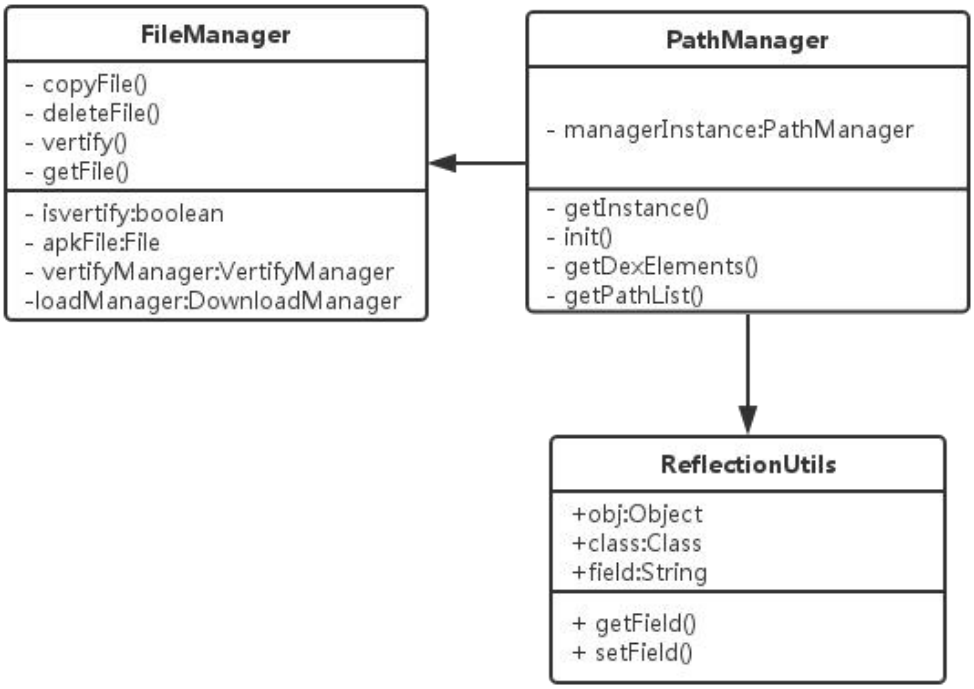

图 13　动态替换类图

本模块是热修复的重点模块。负责系统运行时类型信息的查找，包括应用中使用到的所有的类信息。然后动态地覆盖掉有 bug 的类，使运行时的类是正确的类信息

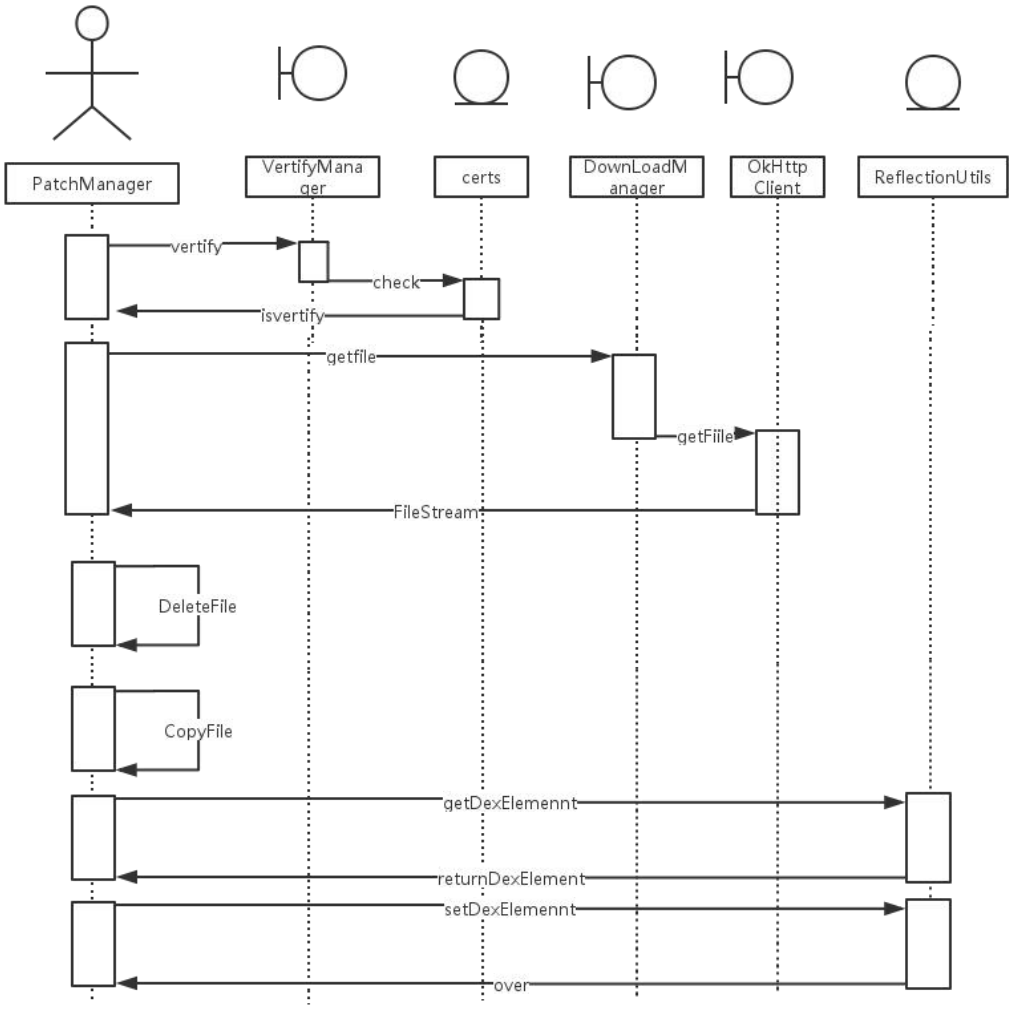

图 14 动态替换时序图

首先，有必要说明一下 java 中化腐朽为神奇的工具--反射。这个工具贯穿整个框架的搭建。Java 反射机制在程序运行时，对于任意一个类或者对象，都能够知道这个类的所有属性和方法(包括私有，静态，final 等任任意属性方法)，这种动态获取的信息以及动态调用对象的方法的功能称为 java 的反射机制。

以下获取属性方法或者给属性赋值，都是借助反射。读到这里，可以发现前面的 ClassLoader 就是反射的核心实现。

(1).自定义 DexClassLoader，指定文件名与缓存 odex 的目录，并且可以还可以指定需要替换的 nativfe Lib 的目录。

```
DexClassLoader cl = new DexClassLoader(outputPath + File.separator + "test.jar",
                        outputPath, null, context.getClassLoader().getParent());
```

(2).获取当前系统的 PathClassLoader 和 DexClassLoader 的 DexElements。

```
Object baseDexElements = getDexElements(getPathList(context.getClassLoader()));
Object newDexElements = getDexElements(getPathList(cl));
```

(3).合并两个数组，并赋值给 PathClassLoader.

```
 Object pathList = getPathList(context.getClassLoader());
 ReflectionUtils.setField(pathList, pathList.getClass(), "dexElements", allDexElements);
```

4.2.5 管理后台模块

本模块属于系统之外的模块，热修复框架在应用端部署后，需要有相应的后台管理支持。更多的情况希望可以在线下载补丁，并且在补丁有新版本的时候，app 要从服务器下载最新的补丁替换本地已经存在的旧补丁。为此，系统应该有一个管理后台，大概有以下功能：

1) 上传不同版本的补丁，并向 APP 项目提供补丁信息查询功能和下载功能。
2) 管理在线的补丁，并能向不同版本号的 APP 主项目提供最合适的补丁 patch。
3) 万一最新的补丁出现紧急 BUG，要提供旧版本回滚功能。
4) 出于安全考虑应该对 APP 项目的请求信息做一些安全性校验。

通过四大模块在流程上的配合，完成热修复的系统设计。保证在安全、及时的情况下，将文件读到本地，运行时加载到内存中，完成补丁获取与补丁加载运行。最后，结合后台模块完成在线更新补丁文件。

5. 系统实现

本小节主要展示系统的执行结果，主要包括创建应用，编译、打包应用，安装应用，运行应用，bug 修复等。

Android 提供了丰富的应用程序框架，它允许开发者建立创新的应用和游戏在 Java 语言环境中的移动设备。Android 应用提供多个入口点。App 由多个可以单独调用的组件组成。比如：一个单独的 Activity 提供一个 UI，一个 service 在后台运行。在一个组件中通过 intent 启动另一个组件（可以是不同应用程序的组件）。这种模式对于单一的应用程序提供多个入口点，并默认允许其他 app 调用。

Apps 可以适配不同的设备。Android 提供了一个自适应的应用程序框架，它允许您为不同的设备配置提供独特的资源。例如，您可以创建不同的屏幕大小不同的 XML 布局文件和系统确定应用基于当前设备的屏幕尺寸，其布局。您可以在运行时查询设备功能的可用性，如果任何应用功能需要特定的硬件，如相机。如果需要，您还可以声明功能，您的应用程序需要这样的应用市场，如谷歌 Play 商店不允许在不支持该功能的设备的安装[12]。

正常的工作流是这样的：创建，编译，测试，打包。

5.1 创建应用

开发者先创建一个 app 如下：

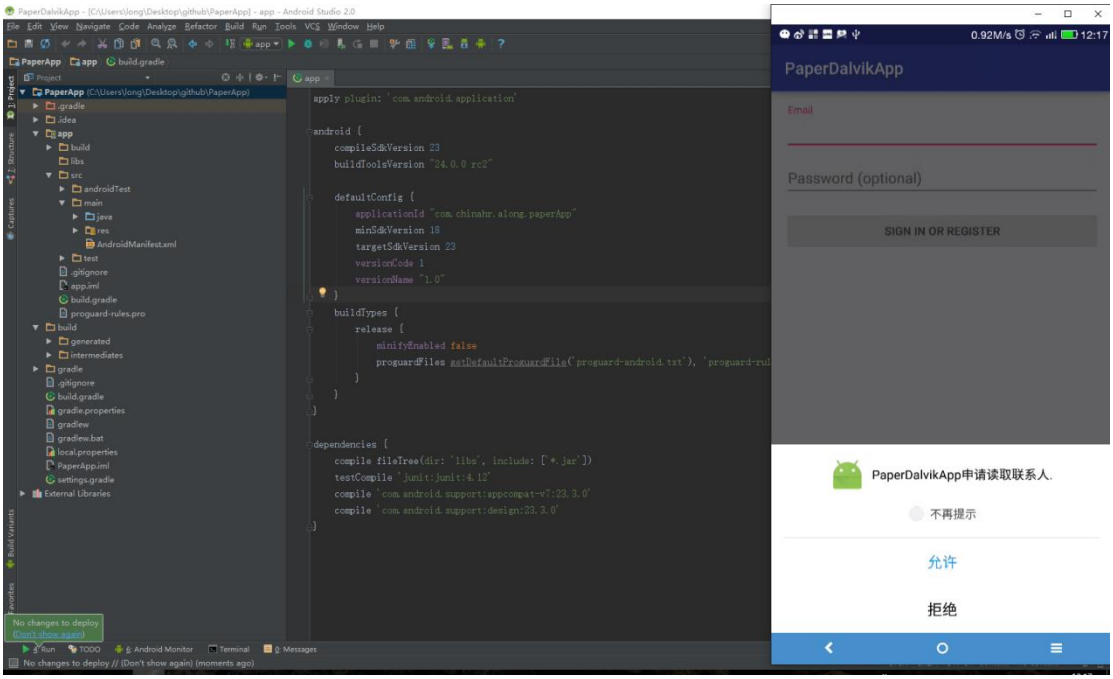

图 15 项目创建图

通过 AndroidStudio 创建一个 app 如图，包括项目文件区，负责编码的工作区，与运行应用的虚拟机。

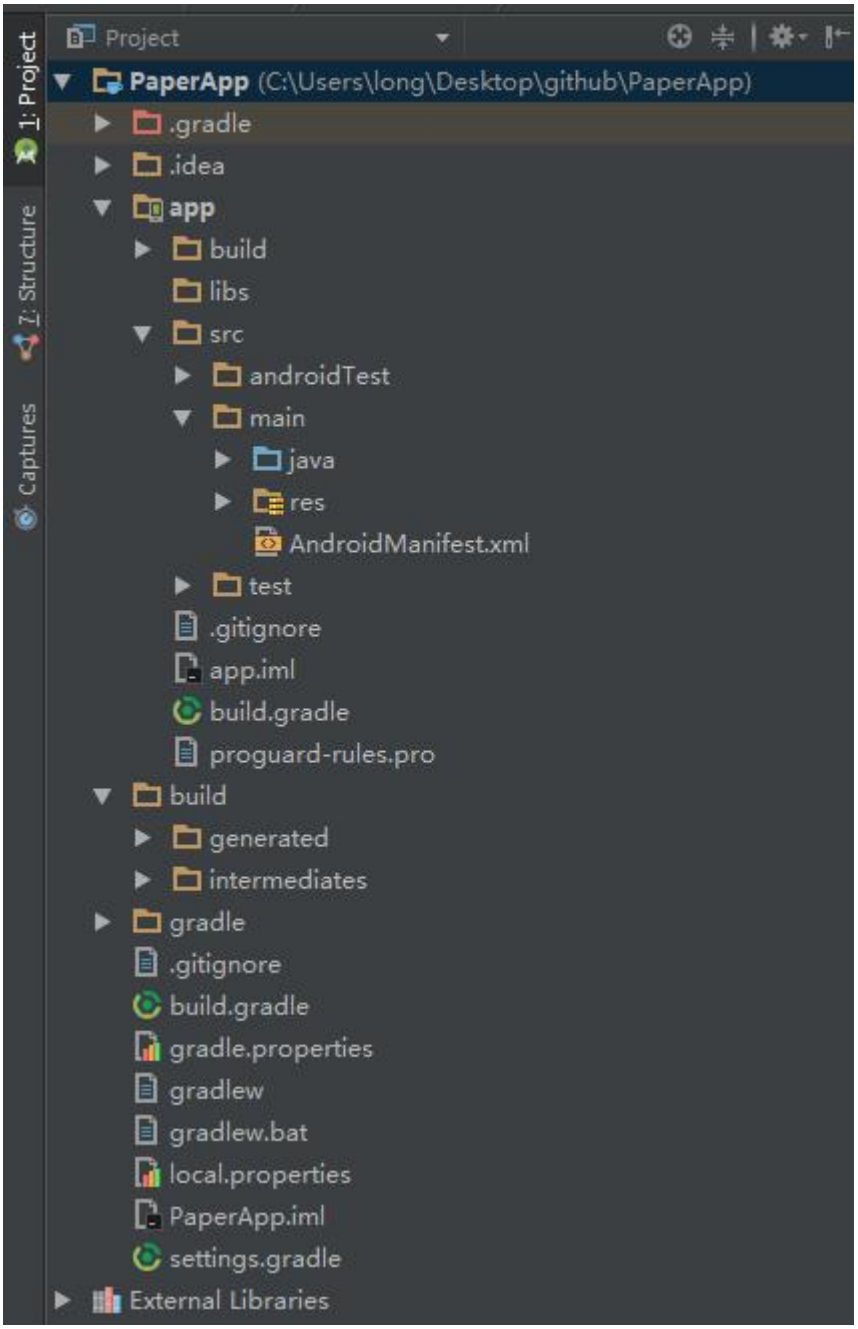

图 16项目文件结构图

如图所示，项目文件分为 3 块：

(1).编译系统：

gradle 版本由 gradle 中的 wrapper 控制,通过 gradlew 执行

(2).配置文件：

Gradle.properties 与 gradle 相关属性的配置

Build.gradle 与 gradle 编译相关的配置

Local.properties 与本地环境配置相关

(3).应用模块：

Build: 编译后的文件存在的位置（包括最终生成的 apk 也在这里面）

Libs: 依赖的库所在的位置（jar 和 aar)

Src: 源代码所在的目录

src/main: 主要代码所在位置（src/androidTest)就是测试代码所在位置了

src/main/assets: android 中附带的一些文件

src/main/java: 最最重要的，开发者的 java 代码所在的位置

src/main/jniLibs: jni 的一些动态库所在的默认位置(.so 文件)

src/main/res: android 资源文件所在位置

src/main/AndroidManifest.xml: AndroidManifest 不用介绍了吧~

build.gradle: 和这个项目有关的 gradle 配置，相当于这个项目的 Makefile，一些项目的依赖就写在这里面

proguard.pro: 代码混淆配置文件

5.2 编译，打包应用

构建过程使用许多工具，生成很多中间文件。在 Android Studio 中，运行 gradle 的编译任务会完成这一切的构建。构建过程非常灵活可配置。下图描述了参与编译的工具和流程：

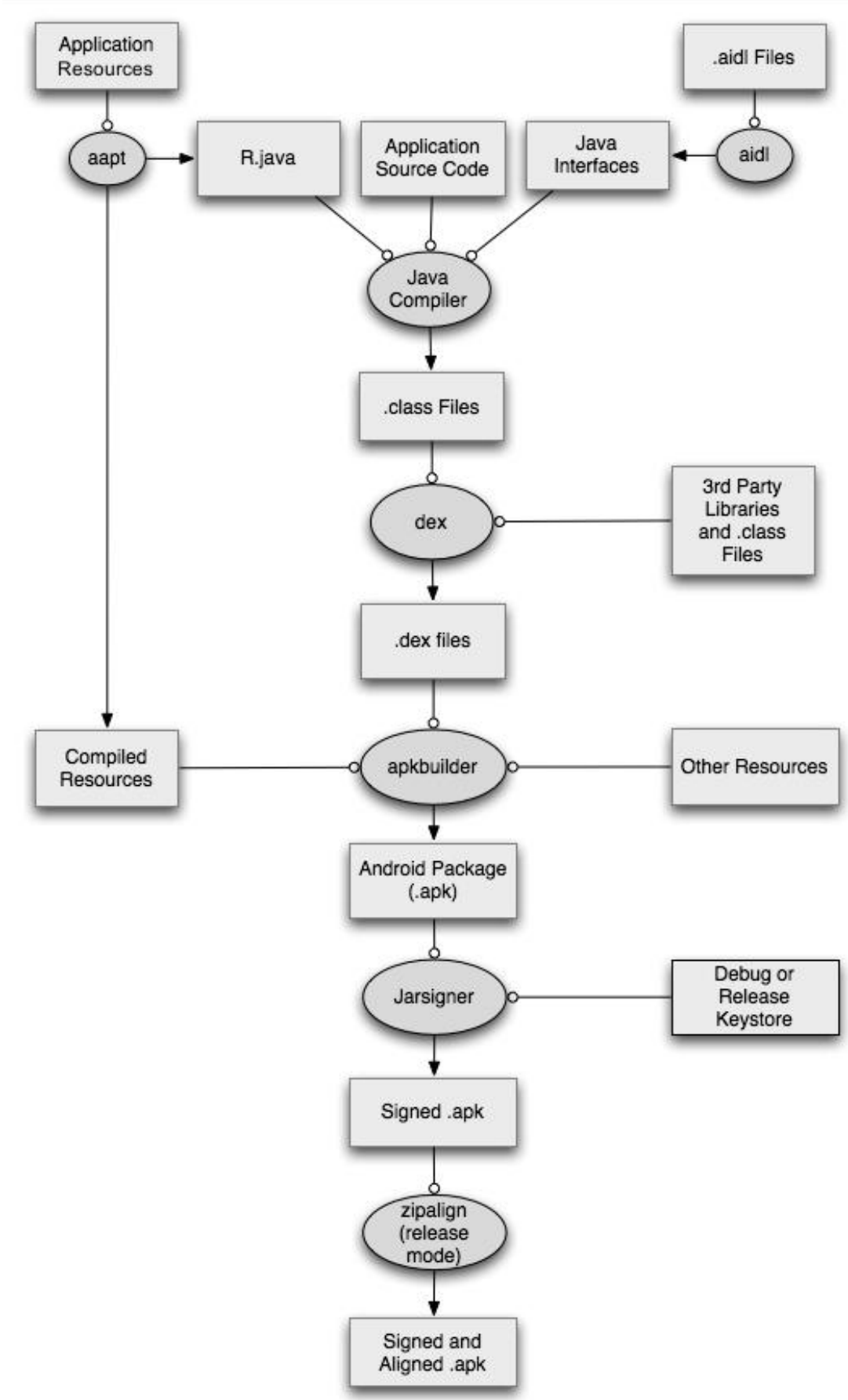
Application Resources
.aidl Files
aapt
R.java
Application Source Code
Java Interfaces
aidl
Java Compiler
.class Files
dex
3rd Party Libraries and .class Files
.dex files
Compiled Resources
apkbuilder
Other Resources
Android Package (.apk)
Jarsigner
Debug or Release Keystore
Signed .apk
zipalign (release mode)
Signed and Aligned .apk

图 17应用编译打包图

如图，标准的构建系统会合并配置在 product flavors, build types, dependencies 上的资源。过程如下：

(1).Android Asset 打包工具（AAPT）编译应用程序资源文件，如 AndroidManifest.xml 文件，activity 的布局 XML 文件。之后生成 R.java，这样就可以从 Java 代码中引用资源。

(2).aidl 工具将.aidl 文件转化成 java 的接口。

(3).编译 java 源码生成 class 文件。Dex 工具将第三方 library 和 cllass 文件转化成 dex 文件。

(4).在标准 Java 环境， Java 源代码被编译成 Java 字节码，其存储在.class 文件。.class 文件是在运行时由 JVM 读取。在 Java 代码中.java 生成一个.class 文件。比如说，java 源文件包含一个公共类，一个静态内部类和三个匿名类，编译过程将输出 5 个 .class 文件。在 Android 平台上， Java 源代码仍然编译成.class 文件。然后通过 dx 工具将。Class 文件转换成.dex 即 Dalvik 虚拟机执行文件。一个.class 文件只包含一个类，一个.dex 文件包含多个类。

(5).Apkbuilder 将编译的资源，没有编译的资源，dex 文件打包成 apk 文件。

(6).Apk 文件创建成功后需要签名才能安装到设备上面。最后对齐 apk，使应用程序运行时占用更少的内存。如果应用程序达到方法个数 65k 的限制，需要进行分包处理。

(7).编译后生成的 apk 输出在 build 文件中。

5.3 安装应用

将一个 apk 发送或者下载到设备后，系统会执行一下几个操作：

(1).拷贝 apk 文件到指定目录。在 Android 系统中，apk 安装文件是会被保存起来的，默认情况下，用户安装的 apk 首先会被拷贝到 /data/app 目录下。/data/app 目录是用户有权限访问的目录，在安装 apk 的时候会自动选择该目录存放用户安装的文件，而系统出厂的 apk 文件则被放到了 /system 分区下，包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有 Root 权限的用户才能访问，这也就是为什么在没有 Root 手机之前，开发者无法删除系统出厂的 app 的原因了[13]。

(2).解压 apk，拷贝文件，创建应用的数据目录为了加快 app 的启动速度，apk 在安装的时候，会首先将 app 的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml 文件、cache、二进制的 so 动态库等。验证 dex 文件是否符合规范，并优化 dex 生成 odex；接着会将 odex 文件解析成易于被 dalvik 加载和执行的 dexFile 结构体。然后通过 java 层的 DexFile 去执行加载操作。

(3).解析 apk 的 AndroidManifinest.xml 文件。Android 系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何 apk 文件，都会更新这个文件。这个文件位于如下目录：/data/system/packages.xml。系统在安装 apk 的过程中，会解析 apk 的 AndroidManifinest.xml 文件，提取出这个 apk 的重要信息写入到 packages.xml 文件中，这些信息包括：权限、应用包名、APK 的安装位置、版本、userID 等等。由此，开发者就知道了为啥一些应用市场和软件管理类的 app 能够很清楚地知道当前手机所安装的所有的 app，以及这些 app 的详细信息了[14-16]。

(4).其他操作：与 windows 应用安装类似，部分 apk 的安装也会向 Launcher 应用申请添加创建快捷方式。

5.4 运行应用

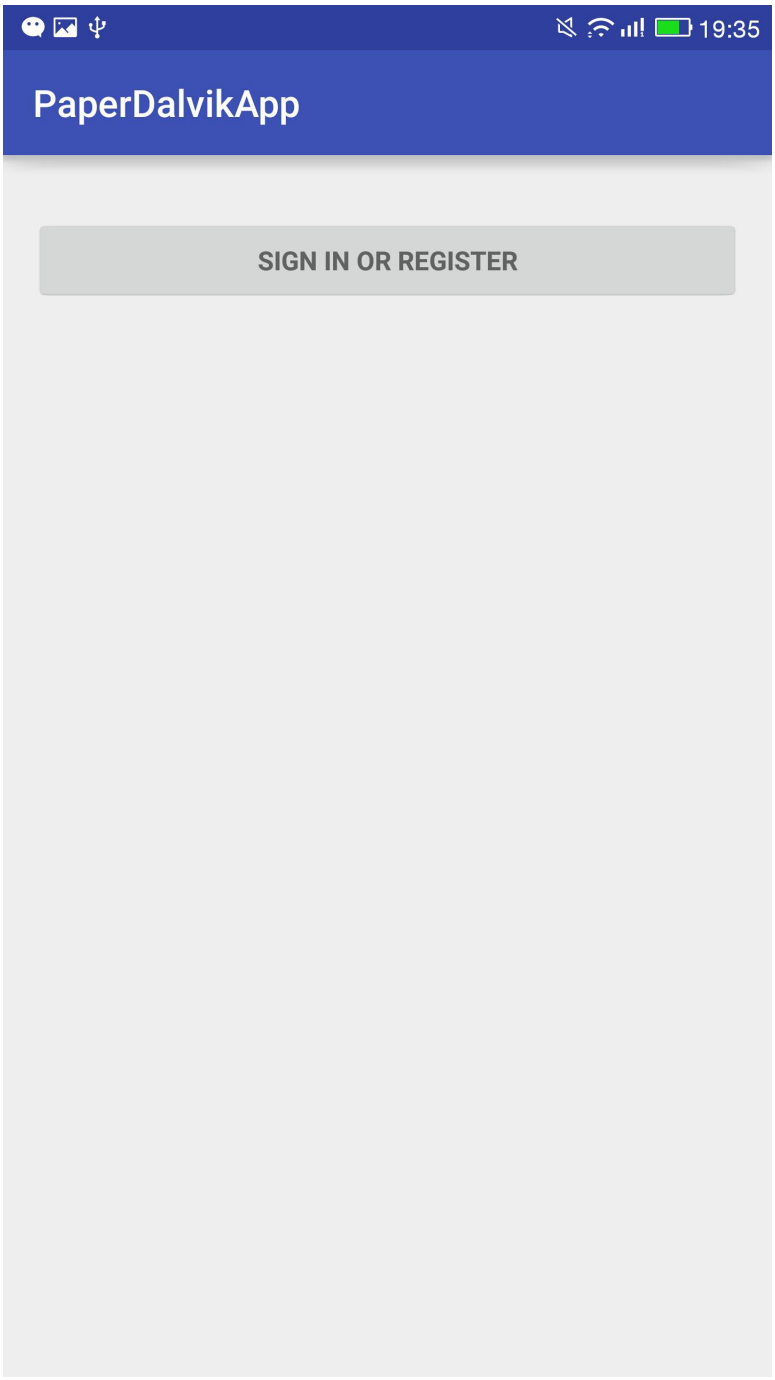

图 18 Bug 应用界面

运行程序后发现出现了一个严重的错误：在登陆页面缺少了 Email 地址与密码两个布局，造成用户无法使用。如此严重的问题，交给 sdk 去处理。

5.5 bug 修复

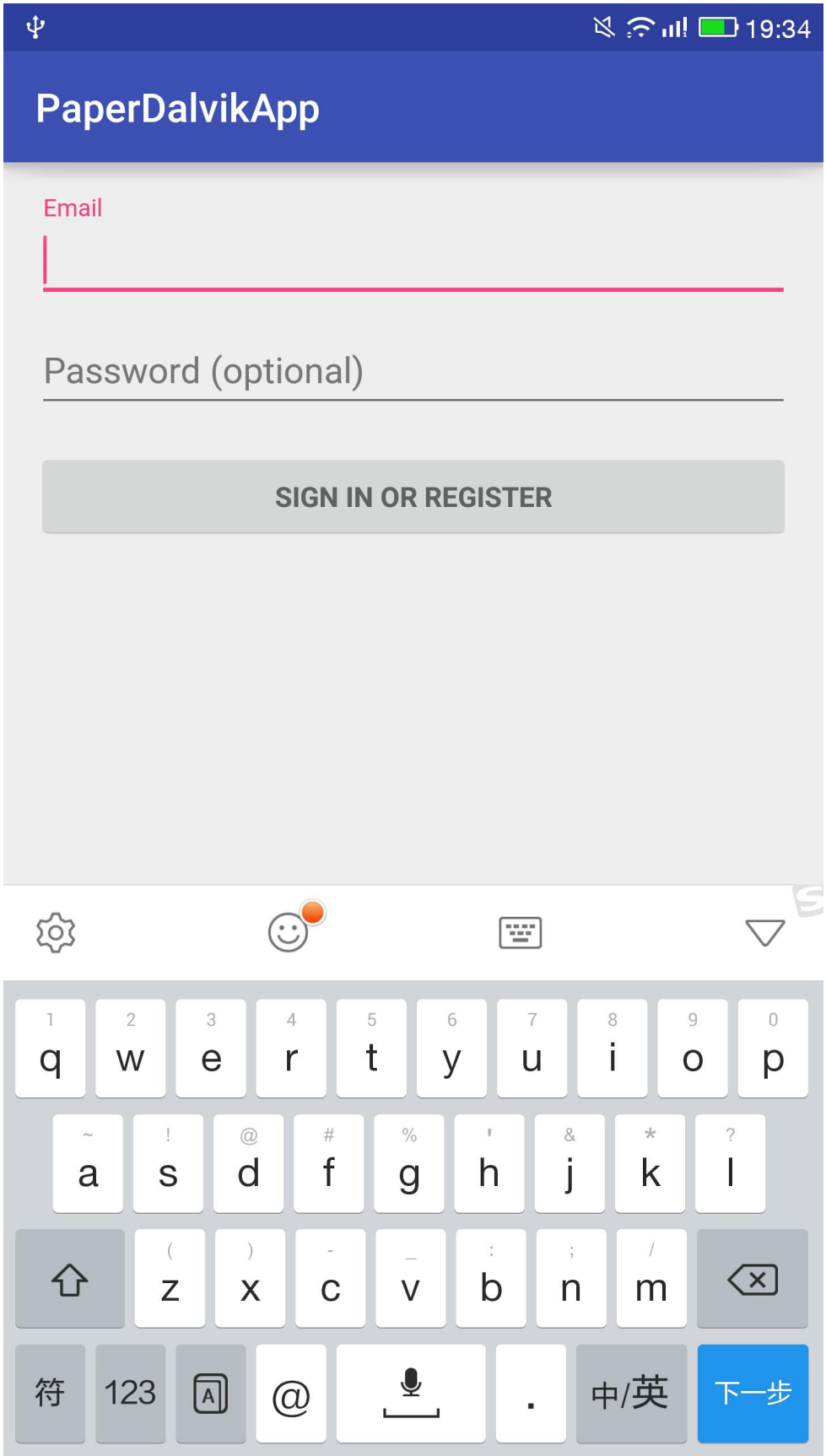

图 19Bug 修复图

首先分析 bug 出现的原因，整理解决方案；然后，打包生成补丁，并且管理好之前的补丁版本更替，把补丁托管到服务器；应用本地判断服务器是否有补丁文件，下载下来之后，

保存到应用私有文件目录下判断补丁的版本信息，然后进行完整性校验；最后查找应用内部类型文件的信息并且动态替换掉对应的类型信息

可以看到程序在动态下载补丁之后，运行程序恢复到正常的状态，现在用户就可以通过填写 Email 地址和密码进行登录操作。

结　论

此次 Android 客户端 bug 热修复框架的制作主要经历需求分析，原理分析，系统设计，系统实现四个模块。从需求出发，需要一个免安装在线修复 bug 的框架。主要提供如下功能：

(1).网络加载：采用 OkHttp 框架一个现代，快速，高效的 Http client，支持 HTTP/2 以及 SPDY，它提供很多功能。诸如连接池，gziping，缓存等

(2).文件操作，保证文件的完整性，完全性，版本信息的准确替换

(3).代码动态修复，完成类型加载的流程分析，找到动态修复的关键点，在应用启动时完成类型覆盖。

完成这些功能之后，结合后台服务器提供一整套补丁管理流程，最终完成热修复方案。

最终得出一个基本的流程：线上出现 bug 后，首先分析 bug 出现的原因，整理解决方案；然后，打包生成补丁，并且管理好之前的补丁版本更替，把补丁托管到服务器；应用本地判断服务器是否有补丁文件，下载下来之后，判断补丁的版本信息，然后进行完整性校验；最后查找应用内部类型文件的信息并且动态替换掉对应的类型信息。

致　谢

临近毕业，心中有很多感恩。完成关于软件工程的一系列课程，从 C 语言，计算机操作系统，java 到设计模式，软件体系结构，测试等。非常感谢各位专业课老师孜孜不倦地为我们传授新的知识和技能。

经过 4 年的软件工程课程学习，掌握了一门同世界交流的工具。让自己可以得到锻炼。在理论知识的学习后，经过姚国清导师的精心指导完成毕业设计及论文的撰写。在此非常感谢姚老师在毕业设计完成过程中给我的悉心指导和关怀。同时感谢同学和舍友对我的支持和关心。对王亚威、何必、龚康和唐镇的合作和帮助表示感谢。

最后，我谨向百忙当中来参加我的论文答辩工作的所有老师致以衷心的感谢。

参考文献

[1] 金泰延 宋亨周.Android 框架揭秘[M].北京：人民邮电出版社,2012.04.

[2] （P.尚）Patrick Chan （R.李）Rosanna Lee 玄伟剑.Java 类库手册[M].北京：北京大学出版社,1997.10.

[3] 刘艳贤;杨凯.JAVA 平台上装载远程类文件的实现[J].河北理工学院学报 2003 , 第 4 期.

[4] 张子言.深入解析 Android 虚拟机[M].北京：清华大学出版社,2014.01.

[5] Joseph Weber 旭日工作室.JAVA l.1 使用大全 3rd ed[M]. 第 3 版.北京：电子工业出版社 ,1998.01.

[6] 李忠良.Android 源码分析实录[M].北京：清华大学出版社,2015.04.

[7] 吴艳霞 张国.Android Dalvik 虚拟机结构及机制剖析 Dalvik 虚拟机各模块机制分析 第 2 卷[M].北京：清华大学出版社,2014.08.

[8] 杨云君.Android 的设计与实现 卷 1[M].北京：机械工业出版社,2013.05 .

[9] 房晓溪.Java 网络程序设计[M].北京：中国铁道出版社,2005.03.

[10] Paul S.Wang 杜一民 赵小燕.Java 面向对象程序设计[M].北京：清华大学出版社,2003.07.

[11] 唐友.Java 语言程序设计[M].哈尔滨：哈尔滨工业大学出版社 , 2013.01 .

[12] 邓凡平.深入理解 Android 卷 I[M]..北京：机械工业出版社,2011.09.

[13] 郭昱.实现 Java 的动态类载入机制[N].计算机世界, 1998.00.00.

附录

附录一：DexPathlist 源码

/** * Creates a {@code DexClassLoader} that finds interpreted and native * code. Interpreted classes are found in a set of DEX files contained * in Jar or APK files. * * The path lists are separated using the character specified by * the "path.separator" system property, which defaults to ":". * * @param dexPath * the list of jar/apk files containing classes and resources * @param dexOutputDir * directory where optimized DEX files should be written * @param libPath * the list of directories containing native libraries; may be null * @param parent * the parent class loader */

```
public DexClassLoader(String dexPath, String dexOutputDir, String libPath,
        ClassLoader parent) {
    super(parent);
    if (dexPath == null || dexOutputDir == null)
        throw new NullPointerException();
    mRawDexPath = dexPath;
    mDexOutputPath = dexOutputDir;
    mRawLibPath = libPath;

    String[] dexPathList = mRawDexPath.split(":");
    int length = dexPathList.length;
    //System.out.println("DexClassLoader: " + dexPathList);
    mFiles = new File[length];
    mZips = new ZipFile[length];
    mDexs = new DexFile[length];
    /* open all Zip and DEX files up front */
    for (int i = 0; i < length; i++) {
        //System.out.println("My path is: " + dexPathList[i]);
        File pathFile = new File(dexPathList[i]);
        mFiles[i] = pathFile;
        if (pathFile.isFile()) {
            try {
                mZips[i] = new ZipFile(pathFile);
            } catch (IOException ioex) {
                // expecting IOException and ZipException
                System.out.println("Failed opening '" + pathFile
                        + "': " + ioex);
                //ioex.printStackTrace();
```

```
                }
                /* we need both DEX and Zip, because dex has no resources */
                try {
                    String outputName =
                        generateOutputName(dexPathList[i], mDexOutputPath);
                    mDexs[i] = DexFile.loadDex(dexPathList[i], outputName, 0);
                } catch (IOException ioex) {
                    // might be a resource-only zip
                    System.out.println("Failed loadDex '" + pathFile
                        + "': " + ioex);
                }
            } else {
                if (VERBOSE_DEBUG)
                    System.out.println("Not found: " + pathFile.getPath());
            }
        }

        /* * Prep for native library loading. */
        String pathList = System.getProperty("java.library.path", ".");
        String pathSep = System.getProperty("path.separator", ":");
        String fileSep = System.getProperty("file.separator", "/");
        if (mRawLibPath != null) {
            if (pathList.length() > 0) {
                pathList += pathSep + mRawLibPath;
            }
            else {
                pathList = mRawLibPath;
            }
        }
    mLibPaths = pathList.split(pathSep);
        length = mLibPaths.length;

        // Add a '/' to the end so we don't have to do the property lookup
        // and concatenation later.
        for (int i = 0; i < length; i++) {
            if (!mLibPaths[i].endsWith(fileSep))
                mLibPaths[i] += fileSep;
            if (VERBOSE_DEBUG)
```

```
                System.out.println("Native lib path " +i+ ": " + mLibPaths[i]);
            }
        }
```

附录二：热修复核心类代码

```
public class PatchManager {
    private static PatchManager managerInstance = new PatchManager();
    public static PatchManager getInstance() {
        return managerInstance;
    }
    public void init(String outputPath, Context context) {
        try {
            DexClassLoader cl = new DexClassLoader(outputPath + File.separator +
"test.jar",
                    outputPath, null, context.getClassLoader().getParent());
//              Log.e("longtianlove",cl.toString());
//              Log.e("longtianlove",context.getClassLoader().toString());
            Object baseDexElements =
getDexElements(getPathList(context.getClassLoader()));
            Object newDexElements = getDexElements(getPathList(cl));
            Object allDexElements = combineArray(newDexElements,
baseDexElements);
            Object pathList = getPathList(context.getClassLoader());
            ReflectionUtils.setField(pathList, pathList.getClass(), "dexElements",
allDexElements);
        } catch (Exception e) {
            Log.e("dragon---patch", e.toString() + "***********");
        }
    }
    private static Object getDexElements(Object paramObject)
            throws IllegalArgumentException, NoSuchFieldException,
IllegalAccessException {
        return ReflectionUtils.getField(paramObject, paramObject.getClass(),
"dexElements");
    }
    private static Object getPathList(Object baseDexClassLoader)
            throws IllegalArgumentException, NoSuchFieldException,
IllegalAccessException, ClassNotFoundException {
        return ReflectionUtils.getField(baseDexClassLoader,
```

```
Class.forName("dalvik.system.BaseDexClassLoader"), "pathList");
        }
        private static Object combineArray(Object firstArray, Object secondArray) {
            Class<?> localClass = firstArray.getClass().getComponentType();
            int firstArrayLength = Array.getLength(firstArray);
            int allLength = firstArrayLength + Array.getLength(secondArray);
            Object result = Array.newInstance(localClass, allLength);
            for (int k = 0; k < allLength; ++k) {
                if (k < firstArrayLength) {
                    Array.set(result, k, Array.get(firstArray, k));
                } else {
                    Array.set(result, k, Array.get(secondArray, k - firstArrayLength));
                }
            }
            return result;
        }

        private PatchManager() {
        }
    }
```