

```
l{1})'          all(c
t=ler,con['rateDetail'][r
oad(sys) sys.setdefault
etail_rate.htm?itemId=3564&
ent=1&tagId=&posi=&picture=
.Cf0B9Qn # -*- cod  utf-8 -*-
import requests import json import
t sys reload(sys) sys.setdefaultencoding
ist_detail_rate.htm?itemId=356489673!
```

```
.n5Ock_3ZCf0B9Qn9GeC4%3D%7CU2xMHDJ7C
/VoV2pV_FvWGVHeOZ%62FRHFMeUB4QHxCdkh8
'RijLDXU_lk3YTc%3D%7CVmhIGCUFOBgkGIMXN
/P29VbFZ_snBKdiAAPR0zHT0BOQI8A1UD%7CWG
J%2BASE_LBksDDAEoGAIYzU%3D%7CWmjCEjw%
JudiBpM_g%3D%3D%7CW2JfykJ%2FX2BaFEV5W
A2111Cs_AAAE6&_ksTS=1440490222698_2142&
ile(r'\w+[1]({})D){1}') content=re.findall(cont)[0]
ontent,"k")count=len(con['rateDetail']['rateList']
3 -*- im rt sys reload(sys) sys.setdefaultencoding
" .. ist_detail_rate.htm?itemId=356489673
*ent=1&tagId=&posi=&picture=
9Qn9Ge   ling: utf-8
ort reque  ort json ir
eload(sys) defaulter
etail_rate.l Id=3564
-1&tar  =&pict'
```

深入探索 Android 热修复技术原理

业界首部 全方位系统介绍热修复原理书籍，
从阿里Sophix方案开发过程入手权威解读！



深入探索 Android 热修复技术原理

业界首部 全方位系统介绍热修复原理书籍，
从阿里Sophix方案开发过程入手权威解读！

| 推荐语

自 2014 年至今，手淘定义和引领了业界 Android 组件化和热修复技术风潮，至于后来者 Instant App 或多或少也受了国内技术风气影响。今天看到团队同学将这块技术认真系统化整理成书，非常欣喜。在这本书里，既能看到对热修复技术风潮的发展历史系统深入总结，看到国内程序员在 Android 系统级技术持续突破上的不懈努力，更看到国内程序员坚持打造世界级优秀专业移动技术产品的雄心壮志！

——手机淘宝基础平台部负责人，阿里巴巴资深技术专家

吴志华（天施）

业内少有的讲解 Android 热修复的深度书籍，对于原理、代码讲解的非常的清晰和深入，值得 Android 工程师研读。

——手机淘宝基础架构团队负责人，阿里巴巴资深技术专家

倪生华（玄黎）

应用热修复是一项略带神秘而又颇具争议的技术，但是它的确赋予应用开发者“驾着飞机修引擎”的能力。本书从 Android 应用热修复技术的原理及代码实现、多种方案进行比较的角度，系统化地阐述了 Android 平台上的应用热修复技术。对 Android 应用热修复有好奇心的技术人员，这本专题书不容错过。

——计算机技术领域著名作家，阿里巴巴飞猪事业部首席架构师
潘爱民

阿里无线 3 年前在业界首次推出 Android 热修复技术 Dexposed，为 Android 底层技术服务业务痛点需求点亮了一个崭新的技术方向，点燃了业界百花齐放的探索热潮。Sophix 的发布让我们再次看到了阿里无线在这个技术领域的自我迭代和锐意创新。这是一个技术改变格局的时代，同时也是一个能人辈出的时代！

——安卓绿色联盟发起人，手机淘宝前架构师
冯森林

| 推荐序

2016, A Year of No Significance.

随着无线互联网在各领域大行其道，我们再一次见证计算机操作系统这一平台技术的周期性发展规律，既：从一个操作系统的兴起，到平台上应用和开发者的繁荣，再到安全领域技术逐步应用到日常应用研发领域，最后进入到平滑发展或者走向衰退。作为平台真正进入巅峰期的一个不起眼的标志——安全领域类的技术逐步融入到应用研发领域，这一现象在 Windows、Linux 到 Android 上已被多次验证，屡试不爽。以热修复技术为例，作为安全类技术的标志性衍生产物，其进入到应用化领域开始大行其道时，标志着平台发展开始迈入新阶段，这也意味着应用市场的繁荣程度、应用开发者的思维和研发模式也进入到游戏的下半程。

回首过去，Android 热修复技术在 2016 年如火如荼，尽管与插件化 / 动态化技术存在一些技术交集，但热修复更倾向于对问题的修护和流量的维护，需要尽可能减少对用户的打扰。因此热修复出现的逻辑蕴含了一个事实：无线应用已从如何获取大量廉价流量向如何更好维护昂贵流量转变。我们发现过往各种热修复立足点或多或少都在强调修复之术或如何尽可能修复，而忽略了一个最大的前提，即对开发者以及接入的各种微妙影响。为解决这一大前提问题，达到更为系统性地理解和使用热修复的目的，我们从集团 Andfix 这样一个具有一定特点的即时热修复技术开始，尝试打造一

个对于开发者友好的系统性热修复产品，这里没有对系统组件的偷梁换柱、也没有运行期对补丁构建的大动干戈、既不需要定制编译工具、也不要求改变资源的排布、甚至不需要参与到 APK 的构建中。最终我们还对系统底层有了一些自己的原创性发现，实践了保留及时性、坚持无侵入性、具备高兼容性这三大原则，最终基本完成了当初做热修复产品的架构和设想。

然而尽管如此，从历史的长河回首，2016 年依然将是平淡的一年。这一年中热修复、插件化等动态性黑科技方案层出不穷，直播 /AR/VR 的方兴未艾反应了人们对未来技术的渴望，也暴露出当下的焦虑，我们在 2016 年做出的种种技术和方向决策，其结果和影响都将投射到历史的长河当中，随波逐流。或许 2016 会因为当时各种技术决策而成为一个承上启下的锚点，供未来审视，至少对 Android 热修复来说，会是这样，这或许是我们的收获。除此之外，从本系列文章中，读者还可以获取一些我们对于热修复技术的思考和解读，能够更深层次了解我们对热修复技术背后本质的剖析，从而得到一些更为具体的体验。

杨青（所为）

手机淘宝技术团队

2017.6

| 前言

热修复技术，可以看做是 Android 平台发展成熟至一定阶段的必然产物。随着移动端业务的复杂程度的增加，传统的发版更新流程显然无法满足业务和开发者的需求，热修复技术的推出很大程度上改善了这一局面。热修复技术在近年来飞速发展，尤其是在 Instant Run 方案推出后，各种热修复技术百花盛开，国内大部分成熟的主流 APP 都拥有自己的热更新技术，像手淘、支付宝、微信、QQ、饿了么、美团等等。可以说，一个好的热修复技术，将为你的 APP 助力百倍。

虽然方案很多，但是深入系统地讲解热修复技术细节的书籍基本没有，市面上国内外的各种 Android 书籍大部分只是泛泛地囊括 Android 开发的基础知识，然而基本都没有包含热修复技术的章节，最多只是一笔带过。即使有很多开源的热修复方案，要自己硬啃代码还是要花费不少时间和精力。如果只有开源代码就足够的话，为什么还需要这么多技术书籍和文档呢？与其看一个晦涩难懂的项目，不如找一本专业详实的书，这将会帮你更系统全面理解这项技术涉及的难点与关键点。

出于回馈业界的考虑，我们把阿里移动热修复方案 Sophix 开发过程中的技术细节进行了整理归纳，以免费电子书的形式与广大 Android 开发者进行分享。对于每一个想在 Android 开发领域有所造诣的开发者，掌握热修复技术更是必备的素质。

本书结构

本书各章节是以热修复所涉及的各个技术面进行编排的，结构分明、循序渐进。推荐以章节顺序进行阅读，当然如果对某些方面感兴趣，也可进行跳读。对于日常工作中遇到的问题，也可以通过阅读本书来寻求答案。

第 1 章 热修复技术介绍

热修复技术的演进与技术发展，Sophix 方案的简介。

第 2 章 代码热修复技术

从底层替换热修复和冷启动修复两方面进行详细解析。

第 3 章 资源热修复技术

资源修复的技术细节与思考。

第 4 章 SO 库热修复技术

SO 库修复的探索与实践。

第 5 章 热修复未来展望

我们对于热修复技术未来的畅想与期盼。

你将得到什么

读完本书，你将会对 Android 热修复技术有很深刻的认识，不仅能很大改进工作效率，对于系统底层原理的理解和今后的开发工

作都有很大帮助。并且，目前热修复原理还是很多高级 Android 技术岗位的面试常客，对付它们你也将得心应手。你还可以通过本书的知识自己实现一个完善的热修复框架，当然，想直接使用的话可以访问[阿里移动热修复 Sophix 官方地址](#)，马上就能够拥有安全可靠的全方位热修复功能。

致谢

Sophix 的推出与本书的发行是阿里巴巴许多同学共同努力的成果。

在此，要感谢团队老大所为，还有其他共同为 Sophix 的研发和推广做出贡献的悟二、查郁、泽胤、潇衍、荻朵同学，以及之前百川项目里面共同参与热修复项目开发的其他手淘同学。还要感谢阿里云事业部的德泰、鸿厚、冷茗、军陌、连珂同学的辛苦付出。

本书的发行也离不开集团技术发展部的孝杨、荣茂、漫远同学的辛勤工作与大力支持，在此诚挚感谢。

甘晓霖（万壑）

手机淘宝技术团队

2017.6

目录

推荐语	ii
推荐序	iv
前言	vi
第1章 热修复技术介绍	1
1.1 什么是热修复	1
1.2 技术积淀	2
1.3 详细比较	3
1.4 技术概览	4
1.5 本章小结	12
第2章 代码热修复技术	13
2.1 底层热替换原理	13
2.2 你所不知的 Java	33
2.3 冷启动类加载原理	73
2.4 多态对冷启动类加载的影响	86
2.5 Dalvik 下完整 DEX 方案的新探索	94
2.6 本章小结	105
第3章 资源热修复技术	106
3.1 普遍的实现方式	106
3.2 资源文件的格式	111

3.3 运行时资源的解析	114
3.4 另辟蹊径的资源修复方案	118
3.5 更优雅地替换 AssetManager	122
3.6 本章小结	127
第 4 章 SO 库热修复技术	128
4.1 SO 库加载原理	128
4.2 SO 库热部署实时生效可行性分析	130
4.3 SO 库冷部署重启生效实现方案	137
4.4 如何正确复制补丁 SO 库	140
4.5 本章小结	143
第 5 章 热修复未来展望	144
热修复的专业性	144
对 Android 的生态的影响	144
Android 与 iOS 热修复的不同	145
未来，无限可能！	146
附 录	147
Sophix 方案纵向比较	147
Sophix 方案横向比较	148

第1章 热修复技术介绍

1.1 什么是热修复

对于广大的移动开发者而言，发版更新是最为寻常不过的事了。然而，如果你发现刚发出去的包有紧急的 BUG 需要修复，那你就必须需要经过下面这样的流程：



图 1-1 正常开发流程

这就是传统的更新流程，步骤十分繁琐，用户需要点击进入应用商店，下载完整新版本安装包，花费大量时间等待安装完成，才能重新打开使用修复完 BUG 的 APP。如果这个 BUG 十分严重，如果有些用户可能就会对你失去耐心而直接卸载，你就再也没有更新修复的机会了。

因此，传统流程存在这几大弊端：

- 重新发布版本代价太大
- 用户下载安装成本太高
- BUG 修复不及时，用户体验太差

相应的，许多开发者找到了比较合适的解决办法。其中一种办法，就是采用 Hybrid 方案。也就是把需要经常变更的业务逻辑以 H5 的方式独立出来。而这种方案，需要传统的 java 开发者学习前端语言，不仅增加了学习成本，而且还要对原先的逻辑进行合适的抽象和转换。并且，对于无法转为 H5 形式的代码仍旧是无法修复的。

还会有人会选择使用插件化方案来解决问题，像 Atlas 或者 DroidPlugin 方案。而这类方式，移植成本非常高，且不说要先学习整套插件化工具，对原先老代码的改造也不是短时间内能够完成的。对于中小型 APP 而言，明显太过笨重。

于是，热修复技术应运而生了。

采用热修复技术，你可以把更新补丁上传到云端，此时 APP 就可以直接从云端下拉补丁直接应用生效。因此，更新流程就变为了这样：



图 1-2 热修复开发流程

可见，热修复的开发流程显得更加灵活，它有以下这几大优势：

- 无需重新发版，实时高效热修复
- 用户无感知修复，无需下载新的应用，代价小
- 修复成功率高，把损失降到最低

1.2 技术积淀

罗马不是一天建成的，阿里巴巴对 Android 热修复技术已经进行了多年的探索。

最开始，是手淘基于 Xposed 进行了改进，产生了针对 Android Dalvik 虚拟机运行时的 Java Method Hook 技术——Dexposed。但这个方案由于对底层 Dalvik 结构过于依赖，最终无法继续兼容 Android 5.0 以后 ART 虚拟机，因此作罢。

后来支付宝提出了新的热修复方案 Andfix。Andfix 同样是一种底层结构替换的方案，也达到了运行时生效即时修复的效果，并且重要的是，做到了 Dalvik 和 ART 环境的全版本兼容。阿里百川结合手淘在实际工程中使用 Andfix 的经验，对相关业

务逻辑解耦后，推出了阿里百川 Hotfix 方案，并得到了良好的反响。

此时的百川 Hotfix 已经是一个很不错的产品了，对于基本的代码修复需求都可以解决，安全性和易用性都做的比较好。然而，它所依赖的基石，Andfix 本身，是有局限性的。且不说其底层固定结构的替换方案稳定性不好，其使用范围也存在着诸多限制，虽然可以通过改造代码绕过限制来达到相同的修复目的，但这种方式既不优雅也不方便。而更大的问题是，Andfix 只提供了代码层面的修复，对于资源和 so 的修复都还未能实现。

而在 Android 平台上，业界除阿里系之外，比较著名的热修复还有：腾讯 QQ 空间的超级补丁技术、微信的 Tinker、饿了么的 Amigo、美团的 Robust 等等。不过他们各自有自身的局限性，或者不够稳定，或者补丁过大，或者效率低下，或者使用起来过于繁琐，大部分技术上看起来似乎可行，但实际体验并不好。

终于在 2017 年 6 月，阿里巴巴手机淘宝技术团队联合阿里云正式发布了新一代非侵入式 Android 热修复方案——Sophix。

Sophix 的横空出世，打破了各家热修复技术纷争的局面。因为我们可以满怀信心地说，在 Android 热修复的三大领域：代码修复、资源修复、so 修复方面，以及方案的安全性和易用性方面，Sophix 都做到了业界领先。

1.3 详细比较

下面的这张表格，从几个热修复最重要的维度，把 Sophix 和另外两个主要商业化热修复方案进行了比较。

方案对比	Sophix	Tinker	Amigo
DEX 修复	同时支持即时生效和冷启动修复	冷启动修复	冷启动修复
资源更新	差量包, 不用合成	差量包, 需要合成	全量包, 不用合成
SO 库更新	插桩实现, 开发透明	替换接口, 开发不透明	插桩实现, 开发透明
性能损耗	低, 仅冷启动情况下有些损耗	高, 有合成操作	低, 全量替换
四大组件	不能增加	不能增加	能增加
生成补丁	直接选择已经编好的新旧包在本地生成	编译新包时设置基线包	上传完整新包到服务端
补丁大小	小	小	大
接入成本	傻瓜式接入	复杂	一般
Android 版本	全部支持	全部支持	全部支持
安全机制	加密传输及签名校验	加密传输及签名校验	加密传输及签名校验
服务端支持	支持服务端控制	支持服务端控制	支持服务端控制

可以看到, Sophix 在各个指标上全面占优。而其中唯一不支持的地方就是四大组件的修复, 这是因为如果要修复四大组件, 必须在 `AndroidManifest` 里面预先插入代理组件, 并且尽可能声明所有权限, 而这么做就会给原先的 app 添加很多臃肿的代码, 对 app 运行流程的侵入性很强, 所以, 本着对开发者透明与代码极简的原则, 我们没有做这种多余的处理。

直接看表格的话, 其中有些技术细节可能还看不太明朗, 那么接下来, 我将从各个角度, 深度解读 Sophix 的技术优势以及它与同类技术的差别。

1.4 技术概览

Sophix 的诞生, 起初是对原先的阿里百川 Hotfix 1.X 版本进行升级衍进。

原先百川 Hotfix 服务端的整套请求控制流程，以及安全检查这部分，是与热修复功能相对分离的，因此我们依旧保留了这部分的逻辑。

而原本的热修复方案，主要限制在于 Andfix 本身，我们最开始也是从突破原先修复限制入手，希望能够基于原先的 Andfix 代码做一些必要的改进。然而最终发现，Andfix 自身限制几乎是无法绕过的，在运行时对原有类的结构是已经固化在内存中的，它的一些动态属性和很难进行扩展。并且由于 Android 系统的碎片化，厂商的虚拟机底层结构都不是确定的，因此直接基于原先机制进行扩展的风险很大。

所以我们绕开了具体的技术实现细节，直接从修复的原理入手，对原先的代码修复技术进行深挖和改良。

回顾为期九个多月的探索与开发，这其中无处不体现着我们对易用性和优雅性的极致追求，在技术先进性与易用性上我们达到了完美的平衡。所以，当我们再回头看目前市面上的其他热修复技术，真的有一种“曾经沧海难为水”的感觉。

1.4.1 设计理念

Sophix 的核心设计理念，就是非侵入性。

我们的打包过程不会侵入到 apk 的 build 流程中。我们所需要的，只有已经生成完毕的新旧 apk，而至于 apk 是如何生成的——是 Android Studio 打包出来的、还是 Eclipse 打包出来的、或者是自定义的打包流程，我们一律不关心。在生成补丁的过程中既不会改变任何打包组件，也不插入任何 AOP 代码，我们极力做到了——不添加任何超出开发者预期的代码，以避免多余的热修复代码给开发者带来困扰。

在 Sophix 中，唯一需要的就是初始化和请求补丁两行代码，甚至连入口 Application 类我们都不做任何修改，这样就给了开发者最大的透明度和自由度。我们甚至重新开发了打包工具，使得补丁工具操作图形界面化，这种所见即所得的补丁生成方式也是阿里热修复独家的。因此，Sophix 的接入成本也是目前市面上所有方案里最低的。

这种非侵入式热更新理念，是我们在设计过程中从用户使用角度进行了深入思考而提炼出的核心思想。

这里的用户，指的自然是广大的开发者。对于开发者而言，热修复应该是一个与业务无关的 SDK 组件，在整个开发过程中感知不到它的存在。最理想的情况，就是开发者拿过来两个 apk，一个是已经安装在手机上的 apk，另一个是将要发布出去的 apk。我们直接通过工具，就可以根据这两个 apk 生成补丁，然后把这个补丁下发给已经安装的旧 app 上，就可以直接加载，使旧 app 重生为新的 app。而这个加载了补丁包新 app，在功能和使用上，将会和直接安装新 apk 别无二致。

至于 Sophix 这个名字，读音是 ['sɔfiks]，是来源于 Sophic（明智的）+ FIX，一个更明智的热修复方案。

1.4.2 代码修复

代码修复有两大主要方案，一种是阿里系的底层替换方案，另一种是腾讯系的类加载方案。

这两类方案各有优劣：

- 底层替换方案限制颇多，但时效性最好，加载轻快，立即见效。
- 类加载方案时效性差，需要重新冷启动才能见效，但修复范围广，限制少。

底层替换方案

底层替换方案是在已经加载了的类中直接替换掉原有方法，是在原来类的基础上进行修改的。因而无法实现对与原有类进行方法和字段的增减，因为这样将破坏原有类的结构。

一旦补丁类中出现了方法的增加和减少，就会导致这个类以及整个 Dex 的方法数的变化。方法数的变化伴随着方法索引的变化，这样在访问方法时就无法正常地索

引到正确的方法了。如果字段发生了增加和减少，和方法变化的情况一样，所有字段的索引都会发生变化。并且更严重的问题是，如果在程序运行中间某个类突然增加了一个字段，那么对于原先已经产生的这个类的实例，它们还是原来的结构，这是无法改变的。而新方法使用到这些老的实例对象时，访问新增字段就会产生不可预期的结果。

这是这类方案的固有限制，而底层替换方案最为人诟病的地方，在于底层替换的不稳定性。

传统的底层替换方式，不论是 Dexposed、Andfix 或者其他安全界的 Hook 方案，都是直接依赖修改虚拟机方法实体的具体字段。例如，改 Dalvik 方法的 jni 函数指针、改类或方法的访问权限等等。这样就带来一个很严重的问题，由于 Android 是开源的，各个手机厂商都可以对代码进行改造，而 Andfix 里 ArtMethod 的结构是根据公开的 Android 源码中的结构写死的。如果某个厂商对这个 ArtMethod 结构体进行了修改，就和原先开源代码里的结构不一致，那么在这个修改过了的设备上，通用性的替换机制就会出问题。这便是不稳定的根源。

而我们也对代码的底层替换原理重新进行了深入思考，从克服其限制和兼容性入手，以一种更加优雅的替换思路，实现了即时生效的代码热修复。

我们实现的是一种无视底层具体结构的替换方式，这种方式不仅解决了兼容性问题，并且由于忽略了底层 ArtMethod 结构的差异，对于所有的 Android 版本都不再需要区分，代码量大大减少。即使以后的 Android 版本不断修改 ArtMethod 的成员，只要保证 ArtMethod 数组仍是以线性结构排列，就能直接适用于将来的 Android 8.0、9.0 等新版本，无需再针对新的系统版本进行适配了。事实也证明确实如此，当我们拿到 Google 刚发不久的 Android O(8.0) 开发者预览版的系统时，hotfix demo 直接就能顺利地加载补丁跑起来了，我们并没有做任何适配工作，稳定性极好。

类加载方案

类加载方案的原理是在 app 重新启动后让 Classloader 去加载新的类。因为在 app 运行到一半的时候，所有需要发生变更的类已经被加载过了，在 Android 上是无法对一个类进行卸载的。如果不重启，原来的类还在虚拟机中，就无法加载新类。因此，只有在下次重启的时候，在还没走到业务逻辑之前抢先加载补丁中的新类，这样后续访问这个类时，就会 Resolve 为新类。从而达到热修复的目的。

再来看看腾讯系三大类加载方案的实现原理。QQ 空间方案会侵入打包流程，并且为了 hack 添加一些无用的信息，实现起来很不优雅。而 QFix 的方案，需要获取底层虚拟机的函数，不够稳定可靠，并且有个比较大的问题是无法新增 public 函数。

微信的 Tinker 方案是完整的全量 dex 加载，并且可谓是将补丁合成做到了极致，然而我们发现，精密的武器并非适用于所有战场。Tinker 的合成方案，是从 dex 的方法和指令维度进行全量合成，整个过程都是自己研发的。虽然可以很大地节省空间，但由于对 dex 内容的比较粒度过细，实现较为复杂，性能消耗比较严重。实际上，dex 的大小占整个 apk 的比例是比较低的，一个 app 里面的 dex 文件大小并不是主要部分，而占空间大的主要还是资源文件。因此，Tinker 方案的时空代价转换的性价比不高。

其实，dex 比较的最佳粒度，应该是在类的维度。它既不像方法和指令维度那样的细微，也不像 bsbiff 比较那般的粗糙。在类的维度，可以达到时间和空间平衡的最佳效果。基于这个准则，我们另辟蹊径，实现了一种完全不同的全量 dex 替换方案。

我们采用的也是全量合成 dex 的技术，这个技术是从[手淘插件化框架 Atlas](#)汲取的。我们会直接利用 Android 原先的类查找和合成机制，快速合成新的全量 dex。这么一来，我们既不需要处理合成时方法数超过的情况，对于 dex 的结构也不用进行破坏性重构。

我们重新编排了包中 dex 的顺序。这样，在虚拟机查找类的时候，会优先找到 classes.dex 中的类，然后才是 classes2.dex、classes3.dex，也可以看做是 dex

文件级别的类插桩方案。这个方式十分巧妙，它对旧包与补丁包中 classes.dex 的顺序进行了打破与重组，最终使得系统可以自然地识别到这个顺序，以实现类覆盖的目的。这将会大大减少合成补丁的开销。

双剑合璧

既然底层替换方案和类加载方案各有其优点，把他们联合起来不是最好的选择吗？Sophix 的代码修复体系正是同时涵盖了这两种方案。两种方案的结合，可以实现优势互补，完全兼顾的作用，可以灵活地根据实际情况自动切换。

这两种方案我们都进行了重大的改进，并且从补丁生成到应用的各个环节都进行了研究，使得二者能很好地整合在一起。在补丁生成阶段，补丁工具会根据实际代码变动情况进行自动选择，针对小修改，在底层替换方案限制范围内的，就直接采用底层替换修复吗，这样可以做到代码修复即时生效。而对于代码修改超出底层替换限制的，会使用类加载替换，这样虽然及时性没那么好，但总归可以达到热修复的目的。

另外，运行时阶段，Sophix 还会再判断所运行的机型是否支持热修复，这样即使补丁支持热修复，但由于机型底层虚拟机构造不支持，还是会走类加载修复，从而达到最好的兼容性。

1.4.3 资源修复

目前市面上的很多资源热修复方案基本上都是参考了 Instant Run 的实现。实际上，Instant Run 的推出正是推动这次热修复浪潮的主因，各家热修复方案，在代码、资源等方面的实现，很大程度上地参考了 Instant Run 的代码，而资源修复方案正是被拿来用到最多的地方。

简要说来，Instant Run 中的资源热修复分为两步：

1. 构造一个新的 AssetManager，并通过反射调用 addAssetPath，把这个完整的新资源包加入到 AssetManager 中。这样就得到了一个含有所有新资源的 AssetManager。
2. 找到所有之前引用到原有 AssetManager 的地方，通过反射，把引用处替换为 AssetManager。

我们发现，其实大量代码都是在处理兼容性问题和找到所有 AssetManager 的引用处，真正的替换的逻辑其实很简单。

我们的方案没有直接使用 Instant Run 的技术，而是另辟蹊径，构造了一个 package id 为 0x66 的资源包，这个包里只包含改变了的资源项，然后直接在原有 AssetManager 中 addAssetPath 这个包就可以了。由于补丁包的 package id 为 0x66，不与目前已经加载的 0x7f 冲突，因此直接加入到已有的 AssetManager 中就可以直接使用了。补丁包里面的资源，只包含原有包里面没有而新的包里面有新增资源，以及原有内容发生了改变的资源。并且，我们采用了更加优雅的替换方式，直接在原有的 AssetManager 对象上进行析构和重构，这样所有原先对 AssetManager 对象的引用是没有发生改变的，所以就不需要像 Instant Run 那样进行繁琐的修改了。

可以说，我们的资源修复方案，优越性超过了 Google 官方的 Instant Run 方案。整个资源替换的方案优势在于：

1. 不修改 AssetManager 的引用处，替换更快更完全。（对比 Instanat Run 以及所有 copycat 的实现）
2. 不必下发完整包，补丁包中只包含有变动的资源。（对比 Instanat Run、Amigo 等方式的实现）
3. 不需要在运行时合成完整包。不占用运行时计算和内存资源。（对比 Tinker 的实现）

所以，我们不要被所谓的“官方实现”束缚住手脚，其实 Instant Run 的开发团

队和 Android framework 的开发团队并不是同一个团队，他们对于 Android 系统机制的理解未必十分深入。只要你认真研读系统代码，实现一个比官方更好的方案绝非难事。所以我想说的是，要想实现技术方案的突破，首先就需要破除所谓“权威”的观念。

1.4.4 SO 库修复

SO 库的修复本质上是对 native 方法的修复和替换。

我们采用的是类似类修复反射注入方式。把补丁 so 库的路径插入到 nativeLibraryDirectories 数组的最前面，就能够达到加载 so 库的时候是补丁 so 库，而不是原来 so 库的目录，从而达到修复的目的。

采用这种方案，完全由 Sophix 在启动期间反射注入 patch 中的 so 库。对开发者依然是透明的。不用像某些其他方案需要手动替换系统的 System.load 来实现替换目的。

1.5 本章小结

本章介绍了热修复技术的主要使用场景和为业界带来的变化。详细说明了阿里巴巴推出的热修复解决方案 Sophix 的由来，同时与其他各大主流方案进行了比较。另外，粗略介绍了热修复所涉及的各个方面，并引导概述后续各个章节。

第2章 代码热修复技术

2.1 底层热替换原理

在各种 Android 热修复方案中，Andfix 的**即时生效**令人印象深刻，它稍显另类，并不需要重新启动，而是在加载补丁后直接对方法进行替换就可以完成修复，然而它的使用限制也遭遇到更多的质疑。

2.1.1 Andfix 回顾

我们先来看一下，为何唯独 Andfix 能够做到即时生效呢？

原因是这样的，在 app 运行到一半的时候，所有需要发生变更的分类已经被加载过了，在 Android 上是无法对一个分类进行卸载的。而腾讯系的方案，都是让 Classloader 去加载新的类。如果不重启，原来的类还在虚拟机中，就无法加载新类。因此，只有在下次重启的时候，在还没走到业务逻辑之前抢先加载补丁中的新类，这样后续访问这个类时，就会 Resolve 为新的类。从而达到热修复的目的。

Andfix 采用的方法是，在已经加载了的类中直接在 native 层替换掉原有方法，是在原来类的基础上进行修改的。我们这就来看一下 Andfix 的具体实现。

其核心在于 replaceMethod 函数

```
@AndFix/src/com/alipay/euler/andfix/AndFix.java  
private static native void replaceMethod(Method src, Method dest);
```

这是一个 native 方法，它的参数是在 Java 层通过反射机制得到的 Method 对象所对应的 jobject。src 对应的是需要被替换的原有方法。而 dest 对应的就是新方

法，新方法存在于补丁包的新类中，也就是补丁方法。

```
@AndFix/jni/andfix.cpp

static void replaceMethod(JNIEnv* env, jclass clazz, jobject src,
    jobject dest) {
    if (isArt) {
        art_replaceMethod(env, src, dest);
    } else {
        dalvik_replaceMethod(env, src, dest);
    }
}
```

Android 的 java 运行环境，在 4.4 以下用的是 dalvik 虚拟机，而在 4.4 以上用的是 art 虚拟机。

```
@AndFix/jni/art/art_method_replace.cpp

extern void __attribute__((visibility ("hidden"))) art_replaceMethod(
    JNIEnv* env, jobject src, jobject dest) {
    if (apilevel > 23) {
        replace_7_0(env, src, dest);
    } else if (apilevel > 22) {
        replace_6_0(env, src, dest);
    } else if (apilevel > 21) {
        replace_5_1(env, src, dest);
    } else if (apilevel > 19) {
        replace_5_0(env, src, dest);
    }else{
        replace_4_4(env, src, dest);
    }
}
```

我们以 art 为例，对于不同 Android 版本的 art，底层 Java 对象的数据结构是不同的，因而会进一步区分不同的替换函数，这里我们以 Android 6.0 为例，对应的就是 `replace_6_0`。

```
@AndFix/jni/art/art_method_replace_6_0.cpp

void replace_6_0(JNIEnv* env, jobject src, jobject dest) {

    // %% 通过 Method 对象得到底层 Java 函数对应 ArtMethod 的真实地址。
    art::mirror::ArtMethod* smeth =
```

```

    (art::mirror::ArtMethod*) env->FromReflectedMethod(src);

    art::mirror::ArtMethod* dmeth =
        (art::mirror::ArtMethod*) env->FromReflectedMethod(dest);

    . . .

// % 把旧函数的所有成员变量都替换为新函数。
smeth->declaring_class_ = dmeth->declaring_class_;
smeth->dex_cache_resolved_methods_ = dmeth->dex_cache_resolved_methods_;
smeth->dex_cache_resolved_types_ = dmeth->dex_cache_resolved_types_;
smeth->access_flags_ = dmeth->access_flags_;
smeth->dex_code_item_offset_ = dmeth->dex_code_item_offset_;
smeth->dex_method_index_ = dmeth->dex_method_index_;
smeth->method_index_ = dmeth->method_index_;

smeth->ptr_sized_fields_.entry_point_from_interpreter_ =
dmeth->ptr_sized_fields_.entry_point_from_interpreter_;

smeth->ptr_sized_fields_.entry_point_from_jni_ =
dmeth->ptr_sized_fields_.entry_point_from_jni_;
smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_ =
dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_;

LOGD("replace_6_0: %d , %d",
     smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_,
     dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_);
}

```

每一个 Java 方法在 art 中都对应着一个 ArtMethod，ArtMethod 记录了这个 Java 方法的所有信息，包括所属类、访问权限、代码执行地址等等。

通过 `env->FromReflectedMethod`，可以由 Method 对象得到这个方法对应的 ArtMethod 的真正起始地址。然后就可以把它强转为 ArtMethod 指针，从而对所有成员进行修改。

这样全部替换完之后就完成了热修复逻辑。以后调用这个方法时就会直接走到新方法的实现中了。

2.1.2 虚拟机调用方法的原理

为什么这样替换完就可以实现热修复呢？这需要从虚拟机调用方法的原理说起。

在 Android 6.0，art 虚拟机中 ArtMethod 的结构是这个样子的：

```
@art/runtime/art_method.h

class ArtMethod FINAL {
    ...

protected:
    // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
    // The class we are a part of.
    GcRoot<mirror::Class> declaring_class_;

    // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
    GcRoot<mirror::PointerArray> dex_cache_resolved_methods_;

    // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
    GcRoot<mirror::ObjectArray<mirror::Class>> dex_cache_resolved_types_;

    // Access flags; low 16 bits are defined by spec.
    uint32_t access_flags_;

    /* Dex file fields. The defining dex file is available via declaring_
     * class_->dex_cache_ */

    // Offset to the CodeItem.
    uint32_t dex_code_item_offset_;

    // Index into method_ids of the dex file associated with this method.
    uint32_t dex_method_index_;

    /* End of dex file fields. */

    // Entry within a dispatch table for this method. For static/direct methods
    the index is into
    // the declaringClass.directMethods, for virtual methods the vtable and for
    interface methods the
    // ifTable.
    uint32_t method_index_;

    // Fake padding field gets inserted here.

    // Must be the last fields in the method.
    // PACKED(4) is necessary for the correctness of
```

```

// RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
struct PACKED(4) PtrSizedFields {
    // Method dispatch from the interpreter invokes this pointer which may
    cause a bridge into
    // compiled code.
    void* entry_point_from_interpreter_;

    // Pointer to JNI function registered to this method, or a function to
    resolve the JNI function.
    void* entry_point_from_jni_;

    // Method dispatch from quick compiled code invokes this pointer which
    may cause bridging into
    // the interpreter.
    void* entry_point_from_quick_compiled_code_;
} ptr_sized_fields_;

...
}

```

这其中最重要的字段就是 `entry_point_from_interpreter_` 和 `entry_point_from_quick_compiled_code_` 了，从名字可以看出来，他们就是方法的执行入口。我们知道，Java 代码在 Android 中会被编译为 Dex Code。

art 中可以采用解释模式或者 AOT 机器码模式执行。

解释模式，就是取出 Dex Code，逐条解释执行就行了。如果方法的调用者是以解释模式运行的，在调用这个方法时，就会取得这个方法的 `entry_point_from_interpreter_`，然后跳转过去执行。

而如果是 AOT 的方式，就会先预编译好 Dex Code 对应的机器码，然后运行期直接执行机器码就行了，不需要一条条地解释执行 Dex Code。如果方法的调用者是以 AOT 机器码方式执行的，在调用这个方法时，就是跳转到 `entry_point_from_quick_compiled_code_` 执行。

那我们是不是只需要替换这几个 `entry_point_*` 入口地址就能够实现方法替换了呢？

并没有这么简单。因为不论是解释模式或是 AOT 机器码模式，在运行期间还会

需要用到 ArtMethod 里面的其他成员字段。

就以 AOT 机器码模式为例，虽然 Dex Code 被编译成了机器码。但是机器码并不是可以脱离虚拟机而单独运行的，以这段简单的代码为例：

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    ...
}
```

编译为 AOT 机器码后，是这样的：

```
7: void com.patch.demo.MainActivity.onCreate(android.os.Bundle) (dex_
method_idx=20639)
DEX CODE:
0x0000: 6f20 4600 1000          | invoke-super {v0, v1}, void
android.app.Activity.onCreate(android.os.Bundle) // method@70
0x0003: 0e00                  | return-void

CODE: (code_offset=0x006fdbac size_offset=0x006fdb8 size=96)
...
0x006fdb8: f94003e0 ldr x0, [sp]           ;x0 = MainActivity.onCreate 对
应的 ArtMethod 指针
0x006fdb4: b9400400 ldr w0, [x0, #4]      ;w0 = [x0 + 4] = dex_cache_
resolved_methods_ 字段
0x006fdb8: f9412000 ldr x0, [x0, #576]   ;x0 = [x0 + 576] = dex_cache_
resolved_methods_ 数组的第 72 (=576/8) 个元素，即对应 Activity.onCreate 的 ArtMethod
指针
0x006fdbec: f940181e ldr lr, [x0, #48]   ;lr = [x0 + 48] = Activity.
onCreate 的 ArtMethod 成员的 entry_point_from_quick_compiled_code_ 执行入口点
0x006fdbf0: d63f03c0 blr lr              ;调用 Activity.onCreate
...
```

这里面我去掉了一些校验之类的无关代码，可以很清楚看到，在调用一个方法时，取得了 ArtMethod 中的 dex_cache_resolved_methods_，这是一个存放 ArtMethod* 的指针数组，通过它就可以访问到这个 Method 所在 Dex 中所有的 Method 所对应的 ArtMethod*。

Activity.onCreate 的方法索引是 70，由于是 64 位系统，因此每个指针的大小为 8 字节，又由于 ArtMethod* 元素是从这个数组的第 0x2 个位置开始存放的，因此偏移 $(70 + 2) * 8 = 576$ 的位置正是 Activity.onCreate 的 ArtMethod 指针。

这是一个比较简单的例子，而在实际代码中，有许多更为复杂的调用情况。很多情况下还需要用到 dex_code_item_offset_ 等字段。由此可以看出，AOT 机器码的执行过程，还是会有对于虚拟机以及 ArtMethod 其他成员字段的依赖。

因此，当把一个旧方法的所有成员字段都换成新方法后，执行时所有数据就可以保持和新方法的一致。这样在所有执行到旧方法的地方，会取得新方法的执行入口、所属 class、方法索引号以及所属 dex 信息，然后像调用旧方法一样顺滑地执行到新方法的逻辑。

2.1.3 兼容性问题的根源

然而，目前市面上几乎所有的 native 替换方案，比如 Andfix 和其他安全界的 Hook 方案，都是写死了 ArtMethod 结构体，这会带来巨大的兼容性问题。

从刚才的分析可以看到，虽然 Andfix 是把底层结构强转为了 art::mirror::ArtMethod，但这里的 art::mirror::ArtMethod 并非等同于 app 运行时所在设备虚拟机底层的 art::mirror::ArtMethod，而是 Andfix 自己构造的 art::mirror::ArtMethod。

```
@AndFix/jni/art/art_6_0.h

class ArtMethod {
public:

    // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
    // The class we are a part of.
    uint32_t declaring_class_;
    // Short cuts to declaring_class_->dex_cache_ member for fast compiled
    code access.
    uint32_t dex_cache_resolved_methods_;
    // Short cuts to declaring_class_->dex_cache_ member for fast compiled
    code access.
    uint32_t dex_cache_resolved_types_;
```

```

// Access flags; low 16 bits are defined by spec.
uint32_t access_flags_;
/* Dex file fields. The defining dex file is available via declaring_
class_->dex_cache_ */
// Offset to the CodeItem.
uint32_t dex_code_item_offset_;
// Index into method_ids of the dex file associated with this method.
uint32_t dex_method_index_;
/* End of dex file fields. */
// Entry within a dispatch table for this method. For static/direct
methods the index is into
// the declaringClass.directMethods, for virtual methods the vtable and
for interface methods the
// ifTable.
uint32_t method_index_;
// Fake padding field gets inserted here.
// Must be the last fields in the method.
// PACKED(4) is necessary for the correctness of
// RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
struct PtrSizedFields {
    // Method dispatch from the interpreter invokes this pointer which
may cause a bridge into
    // compiled code.
    void* entry_point_from_interpreter_;
    // Pointer to JNI function registered to this method, or a function
to resolve the JNI function.
    void* entry_point_from_jni_;
    // Method dispatch from quick compiled code invokes this pointer
which may cause bridging into
    // the interpreter.
    void* entry_point_from_quick_compiled_code_;
} ptr_sized_fields_;
};

```

我们再来看看 Android 开源代码里面 art 虚拟机里的 ArtMethod:

```

@art/runtime/art_method.h

class ArtMethod FINAL {
    ...
protected:
    // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
    // The class we are a part of.
    GcRoot<mirror::Class> declaring_class_;

    // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
    GcRoot<mirror::PointerArray> dex_cache_resolved_methods_;

```

```
// Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
GcRoot<mirror::ObjectArray<mirror::Class>> dex_cache_resolved_types_;

// Access flags; low 16 bits are defined by spec.
uint32_t access_flags_;

/* Dex file fields. The defining dex file is available via declaring_
class_->dex_cache_ */

// Offset to the CodeItem.
uint32_t dex_code_item_offset_;

// Index into method_ids of the dex file associated with this method.
uint32_t dex_method_index_;

/* End of dex file fields. */

// Entry within a dispatch table for this method. For static/direct methods
the index is into
// the declaringClass.directMethods, for virtual methods the vtable and for
interface methods the
// ifTable.
uint32_t method_index_;

// Fake padding field gets inserted here.

// Must be the last fields in the method.
// PACKED(4) is necessary for the correctness of
// RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
struct PACKED(4) PtrSizedFields {
    // Method dispatch from the interpreter invokes this pointer which may
cause a bridge into
    // compiled code.
    void* entry_point_from_interpreter_;

    // Pointer to JNI function registered to this method, or a function to
resolve the JNI function.
    void* entry_point_from_jni_;

    // Method dispatch from quick compiled code invokes this pointer which
may cause bridging into
    // the interpreter.
    void* entry_point_from_quick_compiled_code_;
} ptr_sized_fields_;

.....
}
```

可以看到，ArtMethod 结构里的各个成员的大小是和 AOSP 开源代码里完全一致的。这是由于 Android 源码是公开的，Andfix 里面的这个 ArtMethod 自然是遵照 android 虚拟机 art 源码里面的 ArtMethod 构建的。

但是，由于 Android 是开源的，各个手机厂商都可以对代码进行改造，而 Andfix 里 ArtMethod 的结构是根据公开的 Android 源码中的结构写死的。如果某个厂商对这个 ArtMethod 结构体进行了修改，就和原先开源代码里的结构不一致，那么在这个修改过了的设备上，替换机制就会出问题。

比如，在 Andfix 替换 `declaring_class_` 的地方，

```
smeth->declaring_class_ = dmeth->declaring_class_;
```

由于 `declaring_class_` 是 andfix 里 ArtMethod 的第一个成员，因此它和以下这行代码等价：

```
* (uint32_t*) (smeth + 0) = * (uint32_t*) (dmeth + 0)
```

如果手机厂商在 ArtMethod 结构体的 `declaring_class_` 前面添加了一个字段 `additional_`，那么，`additional_` 就成为了 ArtMethod 的第一个成员，所以 `smeth + 0` 这个位置在这台设备上实际就变成了 `additional_`，而不再是 `declaring_class_` 字段。所以这行代码的真正含义就变成了：

```
smeth->additional_ = dmeth->additional_;
```

这样就和原先替换 `declaring_class_` 的逻辑不一致，从而无法正常执行热修复逻辑。

这也正是 Andfix 不支持很多机型的原因，很大的可能，就是因为这些机型修改了底层的虚拟机结构。

2.1.4 突破底层结构差异

知道了 native 替换方式兼容性问题的原因，我们是否有办法寻求一种新的方式，不依赖于 ROM 底层方法结构的实现而达到替换效果呢？

我们发现，这样 native 层面替换思路，其实就是替换 ArtMethod 的所有成员。那么，我们并不需要构造出 ArtMethod 具体的各个成员字段，只要把 ArtMethod 的作为整体进行替换，这样不就可以了吗？

也就是把原先这样的逐一替换

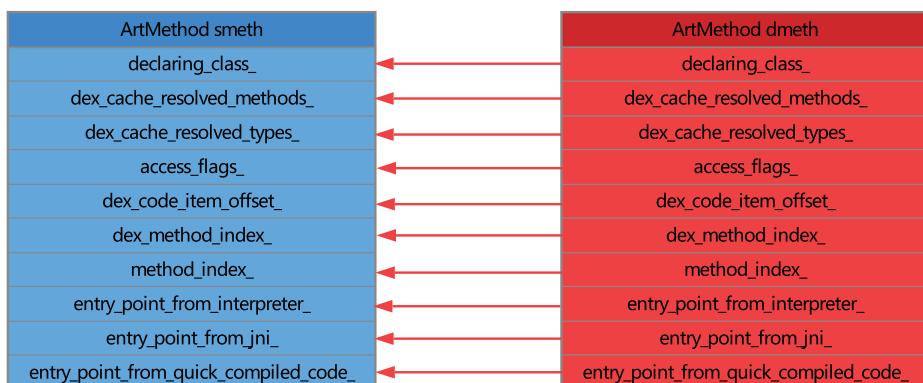


图 2-1

变成了这样的整体替换

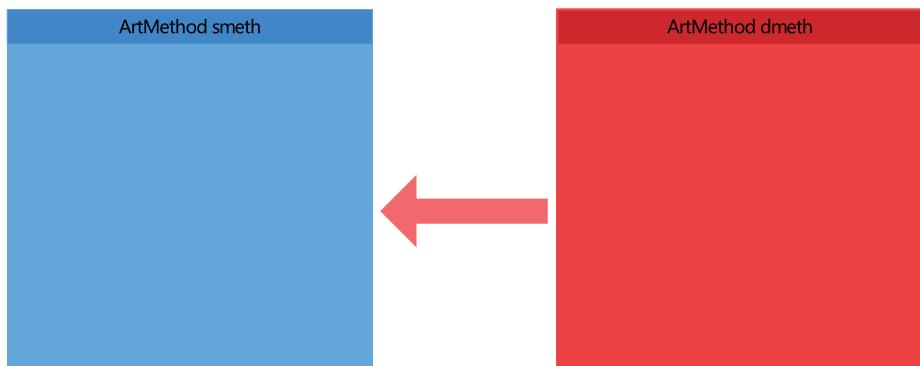


图 2-2

因此 Andfix 这一系列繁琐的替换：

```
// %% 把旧函数的所有成员变量都替换为新函数的。
smeth->declaring_class_ = dmeth->declaring_class_;
smeth->dex_cache_resolved_methods_ = dmeth->dex_cache_resolved_methods_;
smeth->dex_cache_resolved_types_ = dmeth->dex_cache_resolved_types_;
smeth->access_flags_ = dmeth->access_flags_;
smeth->dex_code_item_offset_ = dmeth->dex_code_item_offset_;
smeth->dex_method_index_ = dmeth->dex_method_index_;
smeth->method_index_ = dmeth->method_index_;
...
```

其实可以浓缩为：

```
memcpy(smeth, dmeth, sizeof(ArtMethod));
```

就是这样，一句话就能取代上面一堆代码，这正是我们深入理解替换机制的本质之后研发出的新替换方案。

刚才提到过，不同的手机厂商都可以对底层的 ArtMethod 进行任意修改，但即使他们把 ArtMethod 改得六亲不认，只要我像这样把整个 ArtMethod 结构体完整替换了，就能够把所有旧方法成员自动对应地换成新方法的成员。

但这其中最关键的地方，在于 sizeof(ArtMethod)。如果 size 计算有偏差，导致部分成员没有被替换，或者替换区域超出了边界，都会导致严重的问题。

对于 ROM 开发者而言，是在 art 源代码里面，所以一个简单的 `sizeof(ArtMethod)` 就行了，因为这是在编译期就可以决定的。

但我们是上层开发者，app 会被下发给各式各样的 Android 设备，所以我们是需要在运行时动态地得到 app 所运行设备上面的底层 ArtMethod 大小的，这就没那么简单了。

想要忽略 ArtMethod 的具体结构成员直接取得其 size 的精确值，我们还是需要从虚拟机的源码入手，**从底层的数据结构及排列特点探寻答案。**

在 art 里面，初始化一个类的时候会给这个类的所有方法分配空间，我们可以看到这个分配空间的地方：

```
@android-6.0.1_r62/art/runtime/class_linker.cc

void ClassLinker::LoadClassMembers(Thread* self, const DexFile& dex_file,
                                    const uint8_t* class_data,
                                    Handle<mirror::Class> klass,
                                    const OatFile::OatClass* oat_class) {
    ...
    ArtMethod* const direct_methods = (it.NumDirectMethods() != 0)
        ? AllocArtMethodArray(self, it.NumDirectMethods())
        : nullptr;
    ArtMethod* const virtual_methods = (it.NumVirtualMethods() != 0)
        ? AllocArtMethodArray(self, it.NumVirtualMethods())
        : nullptr;
    ...
}
```

类的方法有 direct 方法和 virtual 方法。direct 方法包含 static 方法和所有不可继承的对象方法。而 virtual 方法就是所有可以继承的对象方法了。

AllocArtMethodArray 函数分配了他们的方法所在区域。

```
@android-6.0.1_r62/art/runtime/class_linker.cc

ArtMethod* ClassLinker::AllocArtMethodArray(Thread* self, size_t length) {
    const size_t method_size = ArtMethod::ObjectSize(image_pointer_size_);
    uintptr_t ptr = reinterpret_cast<uintptr_t>(
        Runtime::Current()->GetLinearAlloc()->Alloc(self, method_size *
length));
    CHECK_NE(ptr, 0u);
    for (size_t i = 0; i < length; ++i) {
        new(reinterpret_cast<void*>(ptr + i * method_size)) ArtMethod;
    }
    return reinterpret_cast<ArtMethod*>(ptr);
}
```

可以看到，ptr 是这个方法数组的指针，而方法是一个接一个紧密地 new 出来排列在这个方法数组中的。这时只是分配出空间，还没填入真正的 ArtMethod 的各个成员值，不过这并不影响我们观察 ArtMethod 的空间结构。

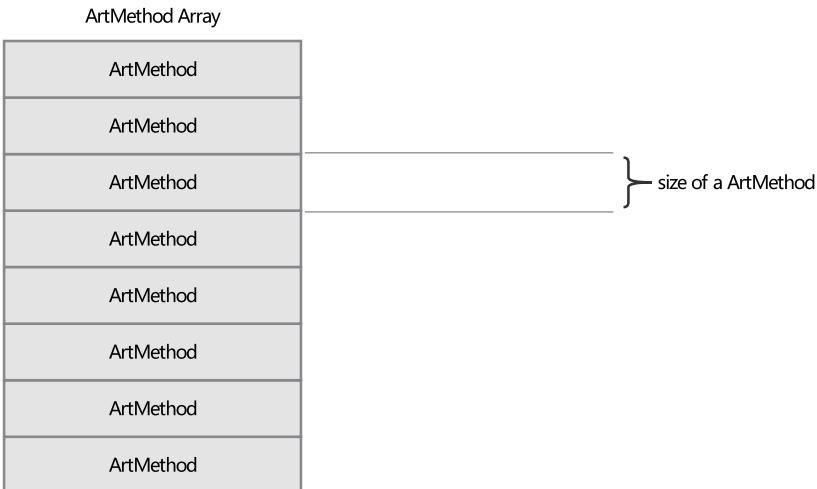


图 2-3

正是这里给了我们启示，ArtMethod 们是紧密排列的，所以一个 ArtMethod 的大小，不就是相邻两个方法所对应的 ArtMethod 的起始地址的差值吗？

正是如此。我们就从这个排列特点入手，自己构造一个类，以一种巧妙的方式获取到这个差值。

```
public class NativeStructsModel {
    final public static void f1() {}
    final public static void f2() {}
}
```

由于 f1 和 f2 都是 static 方法，所以都属于 direct ArtMethod Array。由于 NativeStructsModel 类中只存在这两个方法，因此它们肯定是相邻的。

那么我们就可以在 JNI 层取得它们地址的差值：

```
size_t firMid = (size_t) env->GetStaticMethodID(nativeStructsModelClazz,
"f1", "()V");
size_t secMid = (size_t) env->GetStaticMethodID(nativeStructsModelClazz,
"f2", "()V");
size_t methSize = secMid - firMid;
```

然后，就以这个 `methSize` 作为 `sizeof(ArtMethod)`，代入之前的代码。

```
memcpy(smeth, dmeth, methSize);
```

问题就迎刃而解了。

值得一提的是，由于忽略了底层 `ArtMethod` 结构的差异，对于所有的 Android 版本都不再需要区分，而统一以 `memcpy` 实现即可，代码量大大减少。即使以后的 Android 版本不断修改 `ArtMethod` 的成员，只要保证 `ArtMethod` 数组仍是以线性结构排列，就能直接适用于将来的 Android 8.0、9.0 等新版本，无需再针对新的系统版本进行适配了。事实也证明确实如此，当我们拿到 Google 刚发不久的 Android O(8.0) 开发者预览版的系统时，`hotfix demo` 直接就能顺利地加载补丁跑起来了，我们并没有做任何适配工作，稳定性极好。

2.1.5 访问权限的问题

方法调用时的权限检查

看到这里，你可能会有疑惑：我们只是替换了 `ArtMethod` 的内容，但新替换的方法的所属类，和原先方法的所属类，是不同的类，被替换的方法有权限访问这个类的其他 `private` 方法吗？

以这段简单的代码为例

```
public class Demo {
    Demo() {
        func();
    }

    private void func() {
    }
}
```

`Demo` 构造函数调用私有函数 `func` 所对应的 Dex Code 和 Native Code 为

```

void com.patch.demo.Demo.<init>() (dex_method_idx=20628)
DEX CODE:
...
0x0003: 7010 9550 0000 | invoke-direct {v0}, void com.patch.
demo.Demo.func() // method@20629
...
CODE: (code_offset=0x006fd86c size_offset=0x006fd868 size=140)...
...
0x006fd8c4: f94003e0 ldr x0, [sp] ; x0 = <init> 的
ArtMethod*
0x006fd8c8: b9400400 ldr w0, [x0, #4] ; w0 = dex_cache_
resolved_methods_
0x006fd8cc: d2909710 mov x16, #0x84b8 ; x16 = 0x84b8
0x006fd8d0: f2a00050 movk x16, #0x2, lsl #16 ; x16 = 0x84b8 + 0x20000
= 0x284b8 = (20629 + 2) * 8, ; 也就是 Demo.func 的
ArtMethod* 相对于表头 dex_cache_resolved_methods_ 的偏移。
0x006fd8d4: f8706800 ldr x0, [x0, x16] ; 得到 Demo.func 的
ArtMethod*
0x006fd8d8: f940181e ldr lr, [x0, #48] ; 取得其 entry_point_from_
quick_compiled_code_
0x006fd8dc: d63f03c0 blr lr ; 跳转执行
...

```

这个调用逻辑和之前 Activity 的例子大同小异，需要注意的地方是，在构造函数调用同一个类下的私有方法 `func` 时，没有做任何权限检查。也就是说，这时即使我把 `func` 方法的偷梁换柱，也能直接跳过去正常执行而不会报错。

可以推测，在 dex2oat 生成 AOT 机器码时是有做一些检查和优化的，由于在 dex2oat 编译机器码时确认了两个方法同属一个类，所以机器码中就不存在权限检查的相关代码。

同包名下的权限问题

但是，并非所有方法都可以这么顺利地进行访问的。我们发现补丁中的类在访问同包名下的类时，会报出访问权限异常：

```

Caused by: java.lang.IllegalAccessError:
Method 'void com.patch.demo.BaseBug.test()' is inaccessible to class 'com.
patch.demo.MyClass' (declaration of 'com.patch.demo.MyClass')

```

```
appears in /data/user/0/com.patch.demo/files/baichuan.fix/patch/patch.jar)
```

虽然 `com.patch.demo.BaseBug` 和 `com.patch.demo.MyClass` 是同一个包 `com.patch.demo` 下面的，但是由于我们替换了 `com.patch.demo.BaseBug.test`，而这个替换了的 `BaseBug.test` 是从补丁包的 Classloader 加载的，与原先的 base 包就不是同一个 Classloader 了，这样就导致两个类无法被判别为同包名。具体的校验逻辑是在虚拟机代码的 `Class::IsInSamePackage` 中：

```
android-6.0.1_r62/art/runtime/mirror/class.cc

bool Class::IsInSamePackage(Class* that) {
    Class* klass1 = this;
    Class* klass2 = that;
    if (klass1 == klass2) {
        return true;
    }
    // Class loaders must match.
    if (klass1->GetClassLoader() != klass2->GetClassLoader()) {
        return false;
    }
    // Arrays are in the same package when their element classes are.
    while (klass1->IsArrayClass()) {
        klass1 = klass1->GetComponentType();
    }
    while (klass2->IsArrayClass()) {
        klass2 = klass2->GetComponentType();
    }
    // trivial check again for array types
    if (klass1 == klass2) {
        return true;
    }
    // Compare the package part of the descriptor string.
    std::string temp1, temp2;
    return IsInSamePackage(klass1->GetDescriptor(&temp1), klass2-
>GetDescriptor(&temp2));
}
```

关键点在于，`Class loaders must match` 这行注释。

知道了原因就好解决了，我们只要设置新类的 Classloader 为原来类就可以了。而这一步同样不需要在 JNI 层构造底层的结构，只需要通过反射进行设置。这样仍旧能够保证良好的兼容性。

实现代码如下：

```
Field classLoaderField = Class.class.getDeclaredField("classLoader");
classLoaderField.setAccessible(true);
classLoaderField.set(newClass, oldClass.getClassLoader());
```

这样就解决了同包名下的访问权限问题。

反射调用非静态方法产生的问题

当一个非静态方法被热替换后，在反射调用这个方法时，会抛出异常。

比如下面这个例子：

```
// BaseBug.test 方法已经被热替换了。
...
BaseBug bb = new BaseBug();
Method testMeth = BaseBug.class.getDeclaredMethod("test");
testMeth.invoke(bb);
```

invoke 的时候就会报：

```
Caused by: java.lang.IllegalArgumentException:
Expected receiver of type com.patch.demo.BaseBug,
but got com.patch.demo.BaseBug
```

这里面，expected receiver 的 BaseBug，和 got 到的 BaseBug，虽然都叫 com.patch.demo.BaseBug，但却是不同的类。

前者是被热替换的方法所属的类，由于我们把它的 ArtMethod 的 declaring_class_ 替换了，因此就是新的补丁类。而后者作为被调用的实例对象 bb 的所属类，是原有的 BaseBug。两者是不同的。

在反射 invoke 这个方法时，在底层会调用到 InvokeMethod：

```
jobject InvokeMethod(const ScopedObjectAccessAlreadyRunnable& soa, jobject
javaMethod,
jobject javaReceiver, jobject javaArgs, size_t
```

```

num_frames) {
    ...
    if (!VerifyObjectIsClass(receiver, declaring_class)) {
        return nullptr;
    }
    ...
}

```

这里面会调用 VerifyObjectIsClass 函数做验证。

```

inline bool VerifyObjectIsClass(mirror::Object* o, mirror::Class* c) {
    if (UNLIKELY(o == nullptr)) {
        ThrowNullPointerException("null receiver");
        return false;
    } else if (UNLIKELY(!o->InstanceOf(c))) {
        InvalidReceiverError(o, c);
        return false;
    }
    return true;
}

```

o 表示 Method.invoke 传入的第一个参数，也就是作用的对象。

c 表示 ArtMethod 所属的 Class。

因此，只有 o 是 c 的一个实例才能够通过验证，才能继续执行后面的反射调用流程。

由此可知，这种热替换方式所替换的非静态方法，在进行反射调用时，由于 VerifyObjectIsClass 时旧类和新类不匹配，就会导致校验不通过，从而抛出上面那个异常。

那为什么方法是非静态才有这个问题呢？因为如果是静态方法，是在类的级别直接进行调用的，就不需要接收对象实例作为参数。所以就没有这方面的检查了。

对于这种反射调用非静态方法的问题，我们会采用另一种冷启动机制对付，本文在最后会说明如何解决。

2.1.6 即时生效所带来的限制

除了反射的问题，像本方案以及 Andfix 这样直接在运行期修改底层结构的热修复，都存在着一个限制，那就是只能支持方法的替换。而对于补丁类里面存在方法增加和减少，以及成员字段的增加和减少的情况，都是不适用的。

原因是这样的，一旦补丁类中出现了方法的增加和减少，就会导致这个类以及整个 Dex 的方法数的变化。方法数的变化伴随着方法索引的变化，这样在访问方法时就无法正常地索引到正确的方法了。

而如果字段发生了增加和减少，和方法变化的情况一样，所有字段的索引都会发生变化。并且更严重的问题是，如果在程序运行中间某个类突然增加了一个字段，那么对于原先已经产生的这个类的实例，它们还是原来的结构，这是无法改变的。而新方法使用到这些老的实例对象时，访问新增字段就会产生不可预期的结果。

不过新增一个完整的、原先包里面不存在的新类是可以的，这个不受限制。

总之，只有两种情况是不适用的：

1. 引起原有了类中发生结构变化的修改
2. 修复了的非静态方法会被反射调用

而对于其他情况，这种方式的热修复都可以任意使用。

虽然有着一些使用限制，但一旦满足使用条件，这种热修复方式是十分出众的，它补丁小，加载迅速，能够实时生效无需重新启动 app，并且具有着完美的设备兼容性。对于较小程度的修复再适合不过了。针对小修改可以采用本文这种即时生效的热修复，并且可以结合资源修复，做到资源和代码的即时生效。而如果触及了本文提到的热替换使用限制，**对于比较大的代码改动以及被修复方法反射调用情况**，Sophix 也提供了另一种完整 DEX 修复机制（将在 2.3 章节进行讲解），不过是需要 app 重新冷启动，来发挥其更加完善的修复及更新功能。从而可以做到无感知的应用更新。

2.2 你所不知的 Java

和业界很多热修复方案不同，Sophix 热修复一直秉承粒度小、注重快捷修复、无侵入适合原生工程。因为坚持这个原则，我们在研发过程中遇到很多编译期的问题，这些问题对我们最终方案的实施和热部署也带来或多或少地影响，令人印象深刻。

本节列举了我们在项目实战中遇到的一些挑战，这些都是 Java 语言在编译实现上的一些特点，虽然这些特点与热修复没有直接关系，但深入研究它们对 Android 及 Java 语言的理解都颇有裨益。

2.2.1 内部类编译

有时候我们会发现，修改外部类某个方法逻辑为访问内部类的某个方法时，最后打出来的补丁包竟然提示新增了一个方法，这真的很匪夷所思。所有我们有必要了解一下内部类在编译期间是怎么编译的，首先我们要知道**内部类会在编译期会被编译为跟外部类一样的顶级类**。

静态内部类 / 非静态内部类区别

静态内部类 / 非静态内部类的区别大家应该都很熟悉，非静态内部类持有外部类的引用，静态内部类不持有外部类的引用。所以在 android 性能优化中建议 handle 的实现尽量使用静态内部类，防止外部类 Activity 类不能被回收导致可能 OOM。我们反编译为 smali 看下两者的不同点

```
// 静态内部类
# direct methods
.method constructor <init>()V
    return-void
.end method

// 非静态内部类，编译期间会自动合成 this$0 域表示的就是外部类的引用。
.field final synthetic this$0:Lcom/taobao/patch/demo/DexFixDemo;
# direct methods
.method constructor <init>(Lcom/taobao/patch/demo/DexFixDemo;)V
    .registers 1
    .param V, "dexfixdemo"
    .vreg 0, V
    .ldc v0, Lcom/taobao/patch/demo/DexFixDemo;
    .putstatic this$0, Lcom/taobao/patch/demo/DexFixDemo;
    .return
```

```

.locals 1
.param p1, "this$0"    # Lcom/taobao/patch/demo/DexFixDemo;

    input-object p1, p0, Lcom/taobao/patch/demo/DexFixDemo$A;->this$0:Lcom/
taobao/patch/demo/DexFixDemo;
    return-void
.end method

```

内部类和外部类互相访问

既然内部类实际上跟外部类一样都是顶级类，既然都是顶级类，那是不是意味着对方 private 的 method/field 是没法被访问得到的，事实上外部类为了访问内部类私有的域 / 方法，编译期间自动会为内部类生成 access&** 相关方法

```

public class BaseBug {
    public void test(Context context) {
        InnerClass inner = new InnerClass("old apk");
        Toast.makeText(context.getApplicationContext(), inner.s, Toast.
LENGTH_SHORT).show();
    }
    class InnerClass {
        private String s;
        private InnerClass(String s) {
            this.s = s;
        }
    }
}

```

此时外部类 BaseBug 为了能访问到内部类 InnerClass 的私有域 s，所以编译器自动为 InnerClass 这个内部类合成 access&100 方法，这个方法的实现简单返回私有域 s 的值。同样的如果此时匿名内部类需要访问外部类的 private 属性 / 方法，那么外部类也会自动生成 access&** 相关方法提供给内部类访问使用。

热部署解决方案

所以有这样一种场景，patch 前的 test 方法没访问 inner.s，patch 后的 test 方法访问了 inner.s，那么补丁工具最后检测到了新增了 access&100 方法。那么我们只要防止生成 access&** 相关方法，就能走热部署，也就是底层替换方式热修复。

所以只要满足以下条件，就能避免编译器自动生成 access&** 相关方法

- 一个外部类如果有内部类，把所有 method/field 的 private 访问权限改成 protected 或者默认访问权限或 public。
- 同时把内部类的所有 method/field 的 private 访问权限改成 protected 或者默认访问权限或 public。

2.2.2 匿名内部类编译

匿名内部类其实也是个内部类，所以自然也有上一小节说明情况的影响，但是我们发现新增一个匿名类（补丁热部署模式下是允许新增类），同时规避上一节的情况，但是匪夷所思的还是提示了 method 的新增，所以接下来看下匿名内部类跟非匿名内部类相比，又有怎么样的特殊性。

匿名内部类编译命名规则

匿名内部类顾名思义就是没名字的。匿名内部类的名称格式一般是**外部类 &numble;**，后面的 numble，编译期根据该匿名内部类在外部类中出现的先后关系，依次累加命名。

```
public class DexFixDemo {
    public static void test(Context context) {
        /*new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                Log.d("BCHotfix", "OnClickListener");
            }
        };*/
        new Thread("thread-1") {
            @Override
            public void run() {
                Log.d("BCHotfix", "thread-1 thread");
            }
        }.start();
    }
}
```

修复后的 apk 新增 `DialogInterface.OnClickListener` 这么一个匿名内部类，但是最后补丁工具发现新增了 `onClick` 方法，因为 patch 前只有一个 Thread 匿名内部类，此时该类的名称是 `DexFixDemo&1`，然后 patch 后在 test 方法中新增了 `DialogInterface.OnClickListener` 一个匿名内部类。此时 `DialogInterface.OnClickListener` 匿名内部类名称是 `DexFixDemo&1`，Thread 匿名内部类名称是 `DexFixDemo&2`，所以前后 DexFixDemo&1 类进行对比差异，这个时候已经完全乱套了。同样的道理，**减少一个匿名内部类也存在相同的情况。**

热部署解决方案

新增 / 减少匿名内部类，实际上对于热部署来说是无解的，因为补丁工具拿到的已经是编译后的 .class 文件，所以根本没法去区分 `DexFixDemo&1/DexFixDemo&2` 类。所以这种情况下，如果有补丁热部署的需求，应该极力避免插入一个新的匿名内部类。当然如果是**匿名内部类是插入到外部类的末尾，那么是允许的。**

2.2.3 有趣的域编译

静态 field，非静态 field 编译

实际上在热部署方案中除了不支持 method/field 的新增，同时也是不支持 `<clinit>` 的修复，这个方法会在 Dalvik 虚拟机中类加载的时候进行类初始化时候调用。在 java 源码中本身并没有 clinit 这个方法，这个方法是 android 编译器自动合成的方法。通过测试发现，静态 field 的初始化和静态代码块实际上就会被编译器编译在 `<clinit>` 这个方法，所以我们有必要去了解一下 field/ 代码块到底是怎么编译的。

来看个简单的示例。

```
public class DexFixDemo {
{
    i = 2;
}
```

```

private int i = 1;

private static int j = 1;
static {
    j = 2;
}
}

```

反编译为 smali 看下

```

.method static constructor <clinit>()V -->类初始化方法
    const/4 v0, 0x1
    sput v0, Lcom/taobao/patch/demo/DexFixDemo;->j:I
    const/4 v0, 0x2
    sput v0, Lcom/taobao/patch/demo/DexFixDemo;->j:I
    return-void
.end method

.method public constructor <init>()V -->构造方法
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V -->首先调用父类的默认构造函数
    const/4 v0, 0x2
    iput v0, p0, Lcom/taobao/patch/demo/DexFixDemo;->i:I
    const/4 v0, 0x1
    iput v0, p0, Lcom/taobao/patch/demo/DexFixDemo;->i:I
    return-void
.end method

```

静态 field 初始化，静态代码块

上面的示例中，能够很明显静态 field 初始化和静态代码块被编译器翻译在 `<clinit>` 方法中。静态代码块和静态域初始化在 `clinit` 中的先后关系就是两者出现在源码中的先后关系，所以上述示例中最后 `j==2`。前面说过，类加载然后进行类初始化的时候，会去调用 `clinit` 方法，一个类仅加载一次。以下三种情况都会尝试去加载一个类：

1. new 一个类的对象 (`new-instance` 指令)；
2. 调用类的静态方法 (`invoke-static` 指令)；
3. 获取类的静态域的值 (`sget` 指令)。

首先判断这个类有没有被加载过，如果没有加载过，执行的流程 `dvmResolve-`

`Class->dvmLinkClass->dvmInitClass`, 类的初始化时在 `dvmInitClass`。`dvmInitClass` 这个函数首先会尝试会对父类进行初始化，然后调用本类的 `clinit` 方法，所以此时静态 field 得到初始化和静态代码块得到执行。

非静态 field 初始化，非静态代码块

上面的示例中，能够很明显的看到非静态 field 初始化和非静态代码块被编译器翻译在 `<init>` 默认无参构造函数中。非静态 field 和非静态代码块在 `init` 方法中的先后顺序也跟两者在源码中出现的顺序一致，所以上述示例中最后 `i==1`。实际上如果存在有参构造函数，那么每个有参构造函数都会执行一个非静态域的初始化和非静态代码块。

构造函数会被 android 编译器自动翻译成 `<init>` 方法

前面介绍过 `clinit` 方法在类加载初始化的时候被调用，那么 `<init>` 构造函数方法肯定是对类对象进行初始化时候被调用的，简单来说 `new` 一个对象就会对这个对象进行初始化，并调用这个对象相应的构造函数，看下这行代码 `String s = new String("test");` 编译之后的样子。

```
new-instance v0, Ljava/lang/String;
invoke-direct {v0}, Ljava/lang/String;-><init>()V
```

首先执行 `new-instance` 指令，主要为对象分配堆内存，同时如果类如果之前没加载过，尝试加载类。然后执行 `invoke-direct` 指令调用类的 `init` 构造函数方法执行对象的初始化。

热部署解决方案

由于我们不支持 `<clinit>` 方法的热部署，所以任何静态 field 初始化和静态代码块的变更都会被翻译到 `clinit` 方法中，导致最后热部署失败，只能冷启动生效。如上所见，非静态 field 和非静态代码块的变更被翻译到 `<init>` 构造函数中，热部署模式下只是视为一个普通方法的变更，此时对热部署是没有影响的。

2.2.4 final static 域编译

`final static` 域首先是一个静态域，所以我们自然认为由于会被翻译到 `clinit` 方法中，所以自然热部署下面也是不能变更。但是测试发现，`final static` 修饰的基本类型 /String 常量类型，匪夷所思的竟然没有被翻译到 `clinit` 方法中，见以下分析。

final static 域编译规则

`final static` 静态常量域。看下 `final static` 域被编译后的样子。

```
public class DexFixDemo
    static Temp t1 = new Temp();
    final static Temp t2 = new Temp();

    final static String s1 = new String("heihei");
    final static String s2 = "haha";

    static int i1 = 1;
    final static int i2 = 2;
}
```

看下反编译得到的 smali 文件

```
.field static i1:I = 0x0
.field static final i2:I = 0x2
.field static final s1:Ljava/lang/String;
.field static final s2:Ljava/lang/String; = "haha"
.field static t1:Lcom/taobao/patch/demo/Temp;
.field static final t2:Lcom/taobao/patch/demo/Temp;

# direct methods
.method static constructor <clinit>()V
    new-instance v0, Lcom/taobao/patch/demo/Temp;
    invoke-direct {v0}, Lcom/taobao/patch/demo/Temp;-><init>()V
    sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->t1:Lcom/taobao/patch/
demo/Temp;

    new-instance v0, Lcom/taobao/patch/demo/Temp;
    invoke-direct {v0}, Lcom/taobao/patch/demo/Temp;-><init>()V
    sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->t2:Lcom/taobao/patch/
demo/Temp;

    new-instance v0, Ljava/lang/String;
```

```

const-string v1, "heihei"
invoke-direct {v0, v1}, Ljava/lang/String;-><init>(Ljava/lang/String;)V
sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->s1:Ljava/lang/String;
const/4 v0, 0x1
sput v0, Lcom/taobao/patch/demo/DexFixDemo;->i1:I

return-void
.end method

```

我们发现，`final static int i2 = 2` 和 `final static String s2 = "haha"` 这两个静态域竟然没在 `clinit` 中被初始化。其它的非 `final` 静态域均在 `clinit` 函数中得到初始化。这里注意下 `"haha"` 和 `new String("heihei")` 的区别，前者是字符串常量，后者是引用类型。那这两个 `final static` 域 (`i2` 和 `s2`) 究竟在何处得到初始化？事实上，类加载初始化 `dvmInitClass` 在执行 `clinit` 方法之前，首先会先执行 `initSFields`，这个方法的作用主要就是给 `static` 域赋予默认值。如果是引用类型，那么默认初始值为 `NULL`。010Editor 工具查看 dex 文件结构，我们能看到在 dex 的类定义区，每个类下面都有这么一段数据，图中 `encoded_array_item`。

▼ struct class_def_item class_def[1363]	public com.taobao.patch.demo.DexFixDemo
int64 pos	455148
uint class_idx	(0x984) com.taobao.patch.demo.DexFixDemo
enum ACCESS_FLAGS access_flags	(0x1) ACC_PUBLIC
uint superclass_idx	(0x9DE) java.lang.Object
uint interfaces_off	0
uint source_file_idx	(0x868) "DexFixDemo.java"
uint annotations_off	0
uint class_data_off	711815
► struct class_data_item class_data	6 static fields, 0 instance fields, 3 direct methods, 0 virtual methods
uint static_values_off	3317723
▼ struct encoded_array_item static_values	4 items: [int: 0, int: 2, NULL: NULL, string: "haha"]
▼ struct encoded_array value	[int: 0, int: 2, NULL: NULL, string: "haha"]
► struct uleb128 size	0x4
► struct encoded_value values[0]	int: 0
► struct encoded_value values[1]	int: 2
► struct encoded_value values[2]	NULL: NULL
► struct encoded_value values[3]	string: "haha"

图 2-4 `encoded_array_item` 结构

上述代码示例中，那块区域有 4 个默认初始值，分别是 `t1==NULL`，`t2==NULL`，`s1==NULL`，`s2=="haha"`，`i1==0`，`i2==2`。其中 `t1/t2/s2/i1` 在 `initSFields` 中首先赋值了默认初始化值，然后在随后的 `clinit` 中赋值了程序设置的值。而 `i2/s2` 在 `initSFields` 得到的默认值就是程序中设置的值。

现在我们知道了 static 和 final static 修饰 field 的区别了。简单来说：

- final static 修饰的**原始类型和 string 类型域 (非引用类型)**，在并不会被翻译在 clinit 方法中，而是在类初始化执行 initSFields 方法时得到了初始化赋值。
- final static 修饰的**引用类型**，初始化仍然在 clinit 方法中。

final static 域优化原理

另外一方面，我们经常会看到 android 性能优化相关文档中有说过，如果一个 field 是常量，那么推荐尽量使用 `static final` 作为修饰符。**很明显这句话不对，得到优化的仅仅是 final static 原始类型和 String 类型域 (非引用类型)，如果是引用类型，实际上不会得到任何优化的。**还是接着上面的示例，Temp 直接引用 DexFixDemo 的 static 变量

```
class Temp {
    public static void test() {
        int i1 = DexFixDemo.i1;
        int i2 = DexFixDemo.i2;

        Temp t1 = DexFixDemo.t1;
        Temp t2 = DexFixDemo.t2;

        String s1 = DexFixDemo.s1;
        String s2 = DexFixDemo.s2;
    }
}
```

看下反编译后的 Temp.smali 文件

```
.method public static test()V
    sget v0, Lcom/taobao/patch/demo/DexFixDemo;->i1:I
    .local v0, "i1":I
    const/4 v1, 0x2
    .local v1, "i2":I
    sget-object v4, Lcom/taobao/patch/demo/DexFixDemo;->t1:Lcom/taobao/patch/
demo/Temp;
    .local v4, "t1":Lcom/taobao/patch/demo/Temp;
    sget-object v5, Lcom/taobao/patch/demo/DexFixDemo;->t2:Lcom/taobao/patch/
demo/Temp;
```

```

.local v5, "t2":Lcom/taobao/patch/demo/Temp;
sget-object v2, Lcom/taobao/patch/demo/DexFixDemo;->s1:Ljava/lang/String;
.local v2, "s1":Ljava/lang/String;
const-string v3, "haha"
.local v3, "s2":Ljava/lang/String;
return-void
.end method

```

首先来看下 Temp 怎么拿到 DexFixDemo.i2(final static 域)，直接通过 const/4 指令

```

const/4 vA, #+B // 前一个字节是 opcode，后一个字节前 4 位是寄存器 v1，后 4 位就是立即数的
值 "0x02"

HANDLE_OPCODE(OP_CONST_4 /*vA, #+B*/) {
    s4 tmp;
    vdst = INST_A(inst);
    tmp = (s4) (INST_B(inst) << 28) >> 28; // sign extend 4-bit value
    ILOGV("const/4 v%d,#0x%02x", vdst, (s4)tmp);
    SET_REGISTER(vdst, tmp);
}
FINISH(1);
OP_END

```

const/4 指令的执行很简单，操作数在 dex 中的位置就是在 opcode 后一个字节。怎么拿到 DexFixDemo.i1(非 final 域)，通过 sget 指令。

sget vAA, field@BBBB // 前一个字节是 opcode，后一个字节是寄存器 v0，后两个字节是 DexFixDemo.i1 这个 field 在 dex 文件结构中 field 区的索引值

```

#define HANDLE_SGET_X(_opcode, _opname, _ftype, _regsize)
HANDLE_OPCODE(_opcode /*vAA, field@BBBB*/) {
    StaticField* sfield;
    vdst = INST_AA(inst);
    ref = FETCH(1);
    sfield = (StaticField*)dvmDexGetResolvedField(methodClassDex, ref);
    if (sfield == NULL) {
        EXPORT_PC();
        sfield = dvmResolveStaticField(curMethod->clazz, ref);
        if (sfield == NULL)
            GOTO_exceptionThrown();
        if (dvmDexGetResolvedField(methodClassDex, ref) == NULL) {
            JIT_STUB_HACK(dvmJitEndTraceSelect(self, pc));
        }
    }
}

```

```

    SET_REGISTER##_regsize(vdst, dvmGetStaticField##_ftype(sfield));
}
FINISH(2);

```

首先调用 `dvmDexGetResolvedField` 看这个域之前是否被解析过，如果没有被解析过，调用 `dvmResolveStaticField` 尝试解析域，如果这个静态域所在的类没有被解析还会调用 `dvmResolveClass` 解析类。然后拿到这个 `sfield` 静态域，然后调用 `dvmGetStaticFieldInt(sfield)`，得到这个静态域的值。可见此时 `sget` 指令比 `const/4` 指令的解释要重的多，所以 final static 基本类型可以得到优化。

final static String 类型的引用 `const-string` 指令的解释执行比 `sget-object` 指令的解释要快速的一些。本质上其实跟 final static 基本类型差不多，只是有点略微的区别，final static String 类型的变量，编译期间引用会被优化成 `const-string` 指令，`const/4` 拿到的值因为是个**立即数**，但是 `const-string` 指令拿到的只是**字符串常量在 dex 文件结构中字符常量区的索引 id**，所以需要额外的一次字符串查找。`dex` 文件中有一块区域存储着程序所有的字符串常量，最终这块区域会被虚拟机完整加载到内存中，这块也就是通常说的 "字符串常量区" 内存。

final static 引用类型没有得到优化，是因为不管是不是 final，最后都是通过 `sget-object` 指令去获取该值，所以此时实际上从虚拟机运行性能方面来说得不到任何优化，此时 final 的作用，仅仅是让编译器能在编译期间检测到该 final 域有没有被修改。修改过编译期间就直接报错。

热部署解决方案

所有我们可以得到最后的结论：

- 修改 final static 基本类型或者 String 类型域（非引用类型）域，由于编译期间引用到基本类型的地方被立即数替换，引用到 String 类型（非引用类型）的地方被常量池索引 id 替换，所以在热部署模式下，最终所有引用到该 final

static 域的方法都会被替换。实际上此时仍然可以走热部署。

- 修改 final static 引用类型域，是不允许的，因为这个 field 的初始化会被翻译到 clinit 方法中，所以此时没法走热部署。

2.2.5 有趣的方法编译

应用混淆方法编译

除了以上的内部类 / 匿名内部类可能会造成 method 新增之后，我们发现项目如果应用了混淆，由于可能导致方法的内联和裁剪，那么最后也可能导致 method 的新增 / 减少，以下介绍哪些场景会造成方法的内联和裁剪。

方法内联

实际上有好几种情况可能导致方法被内联掉。

1. 方法没有被其它任何地方引用到，毫无疑问，该方法会被内联掉
2. 方法足够简单，比如一个方法的实现就只有一行，该方法会被内联掉，那么任何调用该方法的地方都会被该方法的实现替换掉
3. 方法只被一个地方引用到，这个地方会被方法的实现替换掉。

举个简单的例子进行说明下。

```
public class BaseBug {
    public static void test(Context context) {
        Toast.makeText(context.getApplicationContext(), "old apk...", Toast.LENGTH_SHORT).show();
        print("haha");
    }
    public static void print(String s) {
        ...
        Log.d("BaseBug", s);
    }
}
```

此时假如 print 方法足够复杂，同时只在 test 方法中被调用，假设 test 方法没被内联，print 方法由于只有一个地方调用此时 print 方法会被内联。查看下生成的 mapping.txt 文件，印证了这个结论，此时没有看到 print 方法的映射，说明这个方法被内联掉了。

```
com.taobao.hotfix.demo.BaseBug -> com.taobao.hotfix.demo.a:  
void test(android.content.Context) -> a
```

如果恰好将要 patch 的一个方法调用了 print 方法，那么 print 被调用了两次，在新的 apk 中不会被内联，补丁工具检测到新增了 print 方法。那么该补丁只能走冷启动方案。

方法裁剪

```
public class BaseBug {  
    public static void test(Context context) {  
        Log.d("BaseBug", "test");  
    }  
}
```

查看下生成的 mapping.txt 文件

```
com.taobao.hotfix.demo.BaseBug -> com.taobao.hotfix.demo.a:  
void test$faab20d() -> a
```

此时 test 方法 context 参数没被使用，所以 test 方法的 context 参数被裁剪，混淆任务首先生成 `test$faab20d()` 裁剪过后的无参方法，然后再混淆。所以如果将要 patch 该 test 方法，同时恰好用到了 context 参数，那么 test 方法的 context 参数不会被裁剪，那么补丁工具检测到新增了 `test(context)` 方法。那么该补丁只能走冷启动方案。

怎么让该参数不被裁剪，当然是有办法的，参数引用住，不让编译器在优化的时候认为这是一个无用的参数就好了，可以采取的方法很多，这里介绍一种最有效的方法：

```

public static void test(Context context) {
    if (Boolean.FALSE.booleanValue()) {
        context.getApplicationContext();
    }
    Log.d("BaseBug", "test");
}

```

注意这里不能用基本类型 false，必须用包装类 Boolean，因为如果写基本类型这个 if 语句也很可能会被优化掉的。

热部署解决方案

实际上只要混淆配置文件加上 `-dontoptimize` 这项就不会去做方法的裁剪和内联。一般情况下项目的混淆配置都会使用到 android sdk 默认的混淆配置文件 `proguard-android-optimize.txt` 或者 `proguard-android.txt`，两者的区别就是后者应用了 `-dontoptimize` 这一项配置而前者没应用。

有关混淆库不进行详细解析，有需要的详细了解的参考 [ProGuard manual](#)。

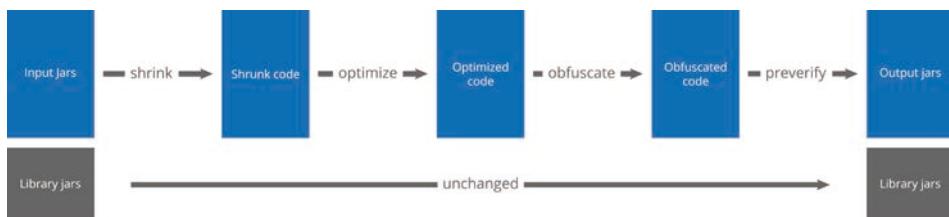


图 2-5 ProGuard_build_process

实际上上述几个步骤都是可以选的，简单对混淆库可能给热部署带来的影响进行说明，主要在 optimization 阶段

optimization step: 进一步优化代码，不是入口点的类和方法可以被设置为 private、static 或 final，无用的参数可能被移除，并且一些方法可能会被内联

可以看到 optimization 阶段，除了会做方法的裁剪和内联可能导致方法的新增 /

减少之外。还可能把方法的修饰符优化成 private/static/final，那么此时补丁工具发现这个方法的实现发生了变更，会对这个方法做毫无意义的 patch。所以补丁热部署模式下，混淆配置最好都加上 `-dontoptimize` 这项。

preverification step: 针对 .class 文件的预校验，在 .class 文件中加上 StackMa/StackMapTable 信息，这样 Hotspot VM 在类加载时候执行类校验阶段会省去一些步骤，因此类加载将更快

我们知道 android 虚拟机执行的 dex 文件格式，编译期间 dx 工具会把所有的 .class 文件优化成 .dex 文件，所以混淆库的预编译这一步在 android 中是没有任何意义的，反而会拖累打包速度，android 虚拟机中有自己的一套代码校验逻辑 (dvmVerifyClass)。所以 android 中混淆配置一般都需要加上 `-dontpreverify` 这一项。

还有混淆库对反射的处理也特别有意思，有需要的可以看 proguard.jar 源码。

```
Class cls = Class.forName("com.taobao.test.Temp");
Method method = cls.getDeclaredMethod("test", new Class[]{Context.class,
String.class});
```

com.taobao.test.Temp 类不会被移除，test 方法有可能在 shrinking 阶段被移除或者在 obfuscation 阶段方法可能被重命名，最后导致反射调用失败。

```
Method method = com.taobao.test.Temp.class.getDeclaredMethod("test", new Class[]
{Context.class, String.class});
```

com.taobao.test.Temp 类不会被移除，test 方法不会被移除，obfuscation 阶段这个方法和 .class 这一行的字节码同时发生变更，所以不会失败。

2.2.6 switch case 语句编译

由于在实现资源修复方案热部署的过程中（第3章将进行详解），我们要做新旧资源 id 的替换，我们发现竟然存在 switch case 语句中的 id 不会被替换掉的情况，

所以有必要来探索下 `switch case` 语句编译的特殊性。

switch case 语句编译规则

```
public void testContinue() {
    int temp = 2;
    int result = 0;
    switch (temp) {
        case 1:
            result = 1;
            break;
        case 3:
            result = 3;
            break;
        case 5:
            result = 5;
            break;
    }
}
public void testNotContinue() {
    int temp = 2;
    int result = 0;
    switch (temp) {
        case 1:
            result = 1;
            break;
        case 3:
            result = 3;
            break;
        case 10:
            result = 10;
    }
}
```

看下 `testContinue/testNotContinue` 方法编译出来有何不同。

```
# virtual methods
.method public testContinue()V
    const/4 v1, 0x2
    .local v1, "temp":I
    const/4 v0, 0x0
    .local v0, "result":I
    packed-switch v1, :pswitch_data_0

:pswitch_0
return-void
```

```

:pswitch_1
const/4 v0, 0x1
:pswitch_2
const/4 v0, 0x3
:pswitch_3
const/4 v0, 0x5

:pswitch_data_0
.packed-switch 0x1
    :pswitch_1
    :pswitch_0
    :pswitch_2
    :pswitch_0
    :pswitch_3
.end packed-switch
.end method

.method public testNotContinue()V
    const/4 v1, 0x2
    .local v1, "temp":I
    const/4 v0, 0x0

    .local v0, "result":I
    sparse-switch v1, :sswitch_data_0

    :sswitch_0
    const/4 v0, 0x1
    :sswitch_1
    const/4 v0, 0x3
    :sswitch_2
    const/16 v0, 0xa

    :sswitch_data_0
    .sparse-switch
        0x1 -> :sswitch_0
        0x3 -> :sswitch_1
        0xa -> :sswitch_2
.end sparse-switch
.end method

```

testContinue 方法的 switch case 语句被翻译成 packed-switch 指令,
testNotContinue 方法的 switch case 语句被翻译成 sparse-switch 指令。
 比较下差异, testContinue 的 switch 语句的 case 项是**连续的几个值比较相近的值 1,3,5**。所以被编译期翻译为 packed-switch 指令, 可以看到对这几个连续的数中间的差值用 :pswitch_0 补齐, :pswitch_0 标签处直接 return-void。

testNotContinue 的 switch 语句的 case 项分别是 `1, 3, 10`, 很明显**不够连续**, 所以被编译期翻译为 `sparse-switch` 指令。怎么才算连续的 case 值这个是由编译器来决定的。

热部署解决方案

一个资源 id 肯定是 `const final static` 变量, 此时恰好 switch case 语句被翻译成 `packed-switch` 指令, 所以这个时候如果不做任何处理就存在资源 id 替换不完全的情况。解决方案其实很简单暴力, 修改 smali 反编译流程, 碰到 `packed-switch` 指令强转为 `sparse-switch` 指令, `:pswitch_N` 等相关标签指令也需要强转为 `:sswitch_N` 指令。然后做资源 id 的暴力替换, 然后再回编译 smali 为 dex。再做类方法变更的检测, 所以就需要经过**反编译 -> 资源 id 替换 -> 回编译**的过程, 这也会使得打补丁变得稍慢一些。

2.2.7 泛型编译

泛型是 java5 才开始引入的, 我们发现泛型的使用, 也可能导致 method 的新增, 所以是时候深入了解一下泛型的编译过程了。

为什么需要泛型?

- Java 语言中的**泛型基本上完全在编译器中实现**, 由编译器执行类型检查和类型推断, 然后生成普通的非泛型的字节码, 就是虚拟机完全无感知泛型的存在。这种实现技术称为擦除 (erasure)。编译器使用泛型类型信息保证类型安全, 然后在生成字节码之前将其清除。
- Java5 才引入泛型, 所以扩展虚拟机指令集来支持泛型被认为是无法接受的, 因为这会为 Java 厂商升级其 JVM 造成难以逾越的障碍。因此采用了可以完全在编译器中实现的擦除方法。

我们知道了以上两点，其中最重要的就是**类型擦除**的理解，我们先来通过一个简单的例子说明 java 为什么需要泛型。java5 之前，要实现类似“泛型”的功能，由于 java 类都是以 Object 为最上层的父类别，所以用 Object 来可以用来实现类似“泛型”的功能。

```
public class ObjectFoo {
    private Object foo;
    public void setFoo(Object foo) {
        this.foo = foo;
    }
    public Object getFoo() {
        return foo;
    }
}
```

代码调用示例

```
ObjectFoo foo1 = new ObjectFoo();
foo1.setFoo(new Boolean(true));
Boolean b = (Boolean) foo1.getFoo(); // 正确
String s = (String) foo1.getFoo(); // 运行时，类型转换失败 ClassCastException 异常
```

由于语法上并没有错误，所以编译器检查不出上面的程式有错误，真正的错误要在执行期才会发生，这时恼人的 `ClassCastException` 就会出来搞怪，Boolean 类型当然不能转换为 String 类型。

所以我们可以看到使用 Object 来实现“泛型”存在一些问题，因为必须要强制类型转换，所以很多人可能忘记强制类型转换，或者是强转用错类型（本该用 Boolean 却用了 Integer），但由于语法上是可以的，所以编译器检查不出错误，因而执行时期就会发生 `ClassCastException`，在 java5 之后，提出了针对泛型设计的解决方案。**该方案在编译时进行类型安全检测，允许程序员在编译时就能检测到非法的类型**，泛型解决方案如下：

```
public class GenericFoo<T> {
    private T foo;
    public void setFoo(T foo) {
```

```

        this.foo = foo;
    }
    public T getFoo() {
        return foo;
    }
}

```

代码调用示例

```

GenericFoo<Boolean> genericFoo = new GenericFoo();
genericFoo.setFoo(new Boolean(true));
Boolean b = genericFoo.getFoo(); // 正确
String s = (String) genericFoo.getFoo(); // 编译不通过, inconvertible types

```

很明显此时使用泛型的优势就体现出来了，编译期间就检查到了可能的异常，我们发现 `Boolean b = genericFoo.getFoo()` 这里并不需要做强制类型转换，实际上编译器会在字节码中自动加上强制类型转换。这里先留下一个坑，见后面小节的分析。

泛型类型擦除

前面说过，Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉，这个过程就称为类型擦除。反编译看下字节码就很容易看到了。

```

.method public getFoo()Ljava/lang/Object;
.method public setFoo(Ljava/lang/Object;)V

```

所以如果此时再定义一个 `setFoo(Object foo)` 方法肯定是行不通的，编译期间报重复方法定义。如果这样的 `<T extends Number>`，限定了泛型，那么是这样的。

```

.method public getFoo()Ljava/lang/Number;
.method public setFoo(Ljava/lang/Number;)V

```

所以我们可以知道 `new T()`，这样使用泛型是编译不过的，因为类型擦除会导致实际上是 `new Object()`，所以是错误的。

类型擦除与多态的冲突和解决

```

class A<T> {
    private T t;
    public T get() {
        return t;
    }
    public void set(T t) {
        this.t = t;
    }
}

class B extends A<Number> {
    private Number n;
    @Override // 跟父类返回值不一样，为什么重写父类 get 方法？
    public Number get() {
        return n;
    }

    @Override // 跟父类方法参数类型不一样，为什么重写父类 set 方法？
    public void set(Number n) {
        this.n = n;
    }
}

class C extends A {
    private Number n;
    @Override // 跟父类返回值不一样，为什么重写父类 get 方法？
    public Number get() {
        return n;
    }
    //@Override 重载父类 get 方法，因为方法参数类型不一样，这里没问题。
    public void set(Number n) {
        this.n = n;
    }
}

```

按照前面类型擦除的分析，为什么类 B 的 `set` 和 `get` 方法可以用 `@Override` 而不报错。`@Override` 表明这个方法是重写，我们知道重写的意思是子类中的方法与父类中的某一方法具有相同的方法名，返回类型和参数表。但是根据前面的分析，

基类 A 由于类型擦除的影响，`set(T t)` 在字节码中实际上是 `set(Object t)`，那么 B 的方法 `set(Number n)` 方法参数不一样，此时类 B 的 set 方法理论上来说应该重载而不是重写了基类的 set 方法。但是我们的本意是进行重写，实现多态，可是类型擦除后，只能变为了重载。这样，类型擦除就和多态有了冲突。实际上 JVM 采用了一个特殊的方法，来完成这项重写功能，那就是 bridge 方法。看下类 B 的字节码表示

```
.method public get()Ljava/lang/Number;
.method public bridge synthetic get()Ljava/lang/Object;
    invoke-virtual {p0}, Lcom/taobao/test/B;->get()Ljava/lang/Number;
    move-result-object v0
    return-object v0
.end method

.method public set(Ljava/lang/Number;)V
.method public bridge synthetic set(Ljava/lang/Object;)V
    check-cast p1, Ljava/lang/Number;
    invoke-virtual {p0, p1}, Lcom/taobao/test/B;->set(Ljava/lang/Number;)V
    return-void
.end method
```

我们可以发现编译器为我们自动合成 `set(Ljava/lang/Object;)` 和 `get()` `Ljava/lang/Object;` 这两个 bridge 方法来重写父类方法，同时这两个桥方法实际上调用 `B.print(Ljava/lang/Number;)` 和 `B.get() Ljava/lang/Number` 这两个重载方法。那么我们可以最终得到最后的结论：

- 子类中真正重写基类方法的是编译器自动合成的 bridge 方法。而类 B 定义的 get 和 set 方法上面的 @Override 只不过是假象，bridge 方法的内部实现去调用我们自己重写的 print 方法而已。所以，**虚拟机巧妙使用了桥方法的方式，来解决了类型擦除和多态的冲突**

这里或许也许会有疑问，类 B 中的方法 `get() Ljava/lang/Number;` 和 `get() Ljava/lang/Object;` 是同时存在的，这就颠覆了我们的认知，如果是我们自己编写 Java 源代码，这样的代码是无法通过编译器的检查的，方法的重载只能以方法参数而无法以返回类型别作为函数重载的区分标准，但是虚拟机却是允许这样

做的，因为虚拟机通过参数类型和返回类型共同来确定一个方法，所以编译器为了实现泛型的多态允许自己做这个看起来“不合法”的事情，然后交给虚拟机自己去区别处理了。

泛型类型转换

同时前面我们还留了一个坑，泛型是可以不需要强制类型转换。

```
//class GenericFoo<T>

GenericFoo<Boolean> genericFool = new GenericFoo<Boolean>(); //new
GenericFoo<Boolean>() 其中 <Boolean> 非必须。但是 GenericFoo<Boolean> genericFool
的 <Boolean> 是必须要的。
genericFool.setFoo(new Boolean(true));
Boolean b = genericFool.getFoo(); // 正确

GenericFoo genericFoo2 = new GenericFoo();
genericFoo2.setFoo(new Boolean(true));
Boolean b = (Boolean) genericFoo2.getFoo(); // 必须强制类型转换，否则编译不过。
```

代码示例中，第一个不需要强制类型转换，但是第二个必须强制类型转换否则编译期报 `inconvertible types` 错误。反编译看下 smali

```
.method public test()V
    .....
    invoke-virtual {v2}, Lcom/taobao/test/GenericFoo;->getFoo()Ljava/lang/
Object;
    move-result-object v0
    check-cast v0, Ljava/lang/Boolean;

    .....
    invoke-virtual {v3}, Lcom/taobao/test/GenericFoo;->getFoo()Ljava/lang/
Object;
    move-result-object v1
    check-cast v1, Ljava/lang/Boolean;
.end method
```

字节码文件很意外，两者其实并没有什么区别，实际上编译期间，编译器发现如果有任何一个变量的申明加上了泛型类型的话，编译器自动加上 `check-cast` 类型转换，而不需要程序员在源码文件中进行强制类型转换，这里不需要并不意味着不会类型转

换，可以发现其实只是类型转换编译器自动帮我们完成了而已。

热部署解决方案

前面类型擦除中说过，如果由 `B extends A` 变成了 `B extends A<Number>`，那么就可能会新增对应的桥方法。此时新增方法了，只能走冷部署了。这种情况下，如果要走热部署，应该避免类似上面那种的修复。

另外一方面，实际上泛型方法内部会生成一个 `dalvik/annotation/Signature` 这个系统注解

```
class A {
    public <A extends Number> void getA(A t) {
        System.out.println("t:" + t);
    }
}

//getA 方法内部注解
.annotation system Ldalvik/annotation/Signature;
    value = {
        "<A:>",
        "Ljava/lang/Number;",
        ">(TA;)V"
    }
.end annotation
```

如果现在把方法的签名换成 `<B extends Number> void getA(B t)`，前面我们说过，泛型类型擦除所以方法的逻辑实际上没有发生任何变化，但是这个方法的注解却发生了变化。

```
.annotation system Ldalvik/annotation/Signature;
    value = {
        "<B:>",
        "Ljava/lang/Number;",
        ">(TB;)V"
    }
.end annotation
```

我们的补丁工具发现这个方法发生了变化（**不会排除注解的变化**），然后对这个

方法进行 patch 此时是多余的没有任何意义的，所以热部署下，我们需要避免这种会导致浪费性能的修复。

2.2.8 Lambda 表达式编译

Lambda 表达式是 java7 才引入的一种表达式，类似于匿名内部类实际上又与匿名内部类有很大的区别，我们发现 Lambda 表达式的使用也可能导致方法的新增 / 减少，导致最后走不了热部署模式。所以是时候深入了解一下 Lambda 表达式的编译过程了。

Lambda 表达式编译规则

首先简单介绍下 lambda 表达式，lambda 为 Java 添加了缺失的函数式编程特点，Java 现在提供的最接近闭包的概念便是 Lambda 表达式。gradle 就是基于 groovy 存在大量闭包。函数式接口具有两个主要特征，**是一个接口，这个接口具有唯一的一个抽象方法，我们将满足这两个特性的接口称为函数式接口**。比如 Java 标准库中的 `java.lang.Runnable` 和 `java.util.Comparator` 都是典型的函数式接口。跟匿名内部类的区别如下：

- 关键字 this 匿名类的 this 关键字指向匿名类，而 lambda 表达式的 this 关键字指向包围 lambda 表达式的类。
- 编译方式，Java 编译器将 lambda 表达式编译成类的私有方法，使用了 Java7 的 invokedynamic 字节码指令来动态绑定这个方法。Java 编译器将匿名内部类编译成外部类 &numble 的新类。

通过一个代码示例来讲解 lambda 表达式，匿名内部类前面已经详细介绍过。

```
public interface TInterface { // 自定义函数式接口
    int test(String s);
}
public class Test {
    private static int temp = 2;
```

```

public static void main(String args[]) throws Exception {
    new Thread(() -> {
        System.out.println("java8 Thread lamada...");
    }).start();

    test(s -> temp + 1);
}
public static void test(TInterface tInterface) {
    System.out.println("java8 TInterface lamada..." + tInterface.
test(1));
}
}
}

```

`javap -p -v Test.class` 反编译 .class 文件查看下，截取关键内容。

```

public class com.demo.Test
  SourceFile: "Test.java"
  InnerClasses:
    public static final #103= #102 of #106; //Lookup=class java/lang/
invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #50 invokestatic java/lang/invoke/LambdaMetafactory.
metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/
lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #51 ()V
      #52 invokestatic com/demo/Test.lambda$main$0:()V
      #51 ()V
    1: #50 invokestatic java/lang/invoke/LambdaMetafactory.
metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/
lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #56 (I)I
      #57 invokestatic com/demo/Test.lambda$main$1:(I)I
      #56 (I)I
  Constant pool:
    #3 = InvokeDynamic      #0:#53          //  #0:run:()Ljava/lang/Runnable;
    #6 = InvokeDynamic      #1:#58          //  #1:test:()Lcom/demo/TInterface;
    ...
  public static void main(java.lang.String[]) throws java.lang.Exception;
    flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
      0: new           #2                  // class java/lang/Thread
      3: dup

```

```

        4: invokedynamic #3, 0          // InvokeDynamic #0:run:()Ljava/
lang/Runnable;
        9: invokespecial #4           // Method java/lang/
Thread."<init>":(Ljava/lang/Runnable;)V
       12: invokevirtual #5          // Method java/lang/Thread.
start:()V
       15: invokedynamic #6, 0          // InvokeDynamic #1:test:()Lcom/
demo/TInterface;
       20: invokestatic #7           // Method test:(Lcom/demo/
TInterface;)V

private static int lambda$main$1(int);
private static void lambda$main$0();

```

这里我们可以发现了几点：

- 编译期间自动生成私有静态的 `lambda$main$**(*)` 方法，这个方法的实现其实就是 lambda 表达式里面的逻辑。
- `invokedynamic` 指令执行 lambda 表达式
- 相比较匿名内部类的区别，发现并没有在磁盘上生成外部类 `&numble` 的新类

那么我们应该思考的是 `invokedynamic` 指令的执行，最后为什么就调用到了 Test 的私有静态 `lambda$main$**(*)` 方法。我们先来看下 jvm 中关于 `invokedynamic` 指令的介绍：在 java7 JVM 中增加了一个新的指令 `invokedynamic`，用于支持动态语言，即允许方法调用可以在运行时指定类和方法，不必在编译的时候确定。字节码中每条 `invokedynamic` 指令出现的位置称为一个动态调用点，`invokedynamic` 指令后面会跟一个指向常量池的调用点限定符 (#3, #6)，这个限定符会被解析为一个动态调用点

上面的反编译后的内容中我们发现，`invokedynamic` 指令执行时实际上会去调用 `java/lang/invoke/LambdaMetafactory` 的 `metafactory` 这个静态方法。这个静态方法实际上会在运行时生成一个实现函数式接口的具体类，然后具体类会调用 Test 的私有静态 `lambda$main$**(*)` 方法。我们可以通过添加 `-Djdk.internal.lambda.dumpProxyClasses` 这个虚拟机运行参数，那么运行时会将生成的新类 `.class` 内容输出到一个文件中，文件内容如下：

```
// $FF: synthetic class
final class Test$$Lambda$1 implements Runnable {
    private Test$$Lambda$1() {
    }
    @Hidden
    public void run() {
        Test.lambda$main$0();
    }
}
// $FF: synthetic class
final class Test$$Lambda$2 implements TIInterface {
    private Test$$Lambda$2() {
    }
    @Hidden
    public int test(int var1) {
        return Test.lambda$main$1(var1);
    }
}
```

Sun/Oracle Hotspot VM 大概就是这么解释 .class 文件中 lambda 表达式的。那我们来看下 android 虚拟机下是怎么解释 lambda 表达式，探索下两者的异同点。

我们知道 android 虚拟机首先通过 `javac` 把源代码编译成 `.class`，然后再通过 `dx` 工具优化成适合移动设备的 `dex` 字节码文件。但是 Android 中如果要使用新的 Java8 语言特性，还需使用新的 Jack 工具链来替换掉老的工具链来编译。新的 Android 工具链将 Java 源语言编译成 Android 可读取的 Dalvik 可执行文件字节码，且有其自己的 `.jack` 库格式，Jack 是 Java Android Compiler Kit 的缩写，它可以将 Java 代码直接编译为 Dalvik 字节码，并负责 `Minification`, `Obfuscation`, `Repackaging`, `Multidexing`, `Incremental compilation`。它试图取代 `javac/dx/proguard/jarjar/multidex` 库等工具。

以下是构建 Android Dalvik 可执行文件可用的两种工具链的对比：

- 旧版 javac 工具链：

javac (.java → .class) → dx (.class → .dex)

- 新版 Jack 工具链：

Jack (.java → .jack → .dex)

图 2-6 Jack 工具链

更多关于 Jack 的介绍，参考以下相关文档：

- [gradle 中怎么如何使用 Lambda](#)
- [Android 新一代编译 toolchain Jack & Jill 简介](#)
- [官方文档 Compiling with Jack](#)

上面我们知道为了使用 java8 中的 lambda 表达式特性，那么必须使用新的 Jack 工具链，新的 Jack 工具链编译产物中是没有 .class 文件的。所有我们直接对新的 Jack 工具链编译出来的 .dex 进行反编译。接着使用上面的代码示例。

```
//Test.smali 内容
.method static synthetic lambda$-com_taobao_test_Test_lambda$1()V // 合成的方法
.method static synthetic lambda$-com_taobao_test_Test_lambda$2(I)I // 合成的方法

.method public static main([Ljava/lang/String;)V
    move-object v0, p0

    .local v0, "args":[Ljava/lang/String;
    new-instance v1, Ljava/lang/Thread;
    new-instance v2, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String_args_LambdaImpl0;
    invoke-direct {v2}, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String_args_LambdaImpl0;-><init>()V
    invoke-direct {v1, v2}, Ljava/lang/Thread;-><init>(Ljava/lang/Runnable;)V
    invoke-virtual {v1}, Ljava/lang/Thread;->start()V

    new-instance v3, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String_args_LambdaImpl1;
    invoke-direct {v3}, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String_args_LambdaImpl1;-><init>()V
```

```

    invoke-static {v3}, Lcom/taobao/test/Test;->test(Lcom/taobao/test/
TInterface;)V

    return-void
.end method

//Test$$Lambda$-void_main_java_lang_String_args_LambdaImpl0.smali 内容
.class final synthetic Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String_args_LambdaImpl0;
.super Ljava/lang/Object;
# interfaces
.impliments Ljava/lang/Runnable;

# virtual methods
.method public run()V
    invoke-static {}, Lcom/taobao/test/Test;->lambda$-com_taobao_test_Test_
lambda$1()V
    return-void
.end method

//Test$$Lambda$-void_main_java_lang_String_args_LambdaImpl1.smali

```

很明显可以看到 .dex 字节码文件和 .class 字节码文件对 lambda 表达式处理的异同点。

- 共同点：编译期间都会为外部类合成一个 static 辅助方法，该方法内部逻辑实现 lambda 表达式。
- 不同点：1. .class 字节码中通过 invokedynamic 指令执行 lambda 表达式。而 .dex 字节码中执行 lambda 表达式跟普通方法调用没有任何区别。2. .class 字节码中运行时生成新类。.dex 字节码中编译期间生成新类。

热部署解决方案

有了以上知识点做基础，同时我们知道我们打补丁是通过反编译为 smali 然后新 apk 跟基线 apk 进行差异对比，得到最后的补丁包。所以首先：

新增一个 lambda 表达式，会导致外部类新增一个辅助方法，所以此时不支持走热部署方案，还有另外一方面，可以看下合成类的命名规则 `Test$$Lambda$-void_main_java_lang_String_args_LambdaImpl0.smali`: 外部类名

+Lambda+Lambda 表达式所在方法的签名 +LambdaImpl+ 出现的顺序号。构成这个合成类。所以此时如果不是在末尾插入了一个新的 Lambda 表达式，那么就会导致跟前面说明匿名内部类一样的问题，会导致类方法比较乱套。减少一个 lambda 表达式热部署情况下也是不允许的，也会导致类方法比较乱套。

那么如果只是修改 lambda 表达式内部的逻辑，此时看起来仅仅相当于修改了一个方法，所以此时是看起来是允许走热部署的。事实上并非如此。我们忽略了一种情况，lambda 表达式访问外部类非静态 field/method 的场景。

前面我们知道 .dex 字节码中 lambda 表达式在编译期间会自动生成新的辅助类。注意该辅助类是非静态的，所以该辅助类如果为了访问 " 外部类 " 的非静态 field/method 就必须持有 " 外部类 " 的引用。如果该辅助类没有访问 " 外部类 " 的非静态 field/method，那么就不会持有 " 外部类 " 的引用。这里注意这个辅助类和内部类的区别。我们前面说过如果是非 static 内部类的话一定会持有外部类的引用的！下面的示例很容易说明这个问题。

lambda 表达式没有访问 " 外部类 " 的非静态 field/method，可以看到并没有持有 " 外部类 " 引用。

```
public synthetic constructor <init>()V
```

lambda 表达式访问了 " 外部类 " 的非静态 field/method，可以看到持有了 " 外部类 " 引用。

```
private synthetic val$this:Lcom/taobao/test/Test; // 合成的变量 val$this，" 外部类 " 引用
public synthetic constructor <init>(Lcom/taobao/test/Test;)V
    input-object p1, p0, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_String__args_LambdaImpl1;->val$this:Lcom/taobao/test/Test;
```

所以此时存在这么个情况，基线 apk 中 lambda 表达式中没有访问非静态 field/method，修复后的 apk 中 lambda 表达式中访问非静态 field/method。那么会导致发现新增了 field。此时热部署也会失败。

最后对这一小节简短的总结一下：

- 增加 / 减少一个 lambda 表达式会导致类方法比较错乱，所以都会导致热部署失败。
- 修改一个 lambda 表达式基于前面的分析，可能导致新增 field，所以此时也会导致热部署失败。

2.2.9 访问权限检查对热替换的影响

2.1.5 访问权限的问题中有提到权限问题对于底层热替换的影响，下面我们就来深入剖析虚拟机下权限控制可能给我们的热修复方案带来的影响，下面代码示例仅演示 Dalvik 虚拟机。

类加载阶段父类 / 实现接口访问权限检查

一个类的加载，必须经历 resolve->link->init 三个阶段，父类 / 实现接口权限控制检查主要发生在 link 阶段。代码如下：

```
bool dvmLinkClass(ClassObject* clazz) {
    ...
    if (clazz->status == CLASS_IDX) {
        ...
        if (clazz->interfaceCount > 0) {
            for (i = 0; i < clazz->interfaceCount; i++) {
                assert(interfaceIdxArray[i] != kDexNoIndex);
                clazz->interfaces[i] =
                    dvmResolveClass(clazz, interfaceIdxArray[i], false);
                ...
                /* are we allowed to implement this interface? */
                if (!dvmCheckClassAccess(clazz, clazz->interfaces[i])) { -->
                    检查所有实现的接口的访问权限
                    dvmLinearReadOnly(clazz->classLoader, clazz->interfaces);
                    ALOGW("Interface '%s' is not accessible to '%s'",
                          clazz->interfaces[i]->descriptor, clazz-
                          >descriptor);
                    dvmThrowIllegalAccessError("interface not accessible");
                    goto bail;
                }
            }
        }
    }
}
```

```

}
...
if (strcmp(clazz->descriptor, "Ljava/lang/Object;") == 0) {
    ...
} else {
    if (clazz->super == NULL) {
        dvmThrowLinkageError("no superclass defined");
        goto bail;
    }
    /* verify */
    else if (!dvmCheckClassAccess(clazz, clazz->super)) { --> 检查父类的访问
权限
        ALOGW("Superclass of '%s' (%s) is not accessible",
              clazz->descriptor, clazz->super->descriptor);
        dvmThrowIllegalAccessError("superclass not accessible");
        goto bail;
    }
}

```

上述代码示例我们可以看到，会依次对当前类实现的接口和父类进行访问权限检查。接下来看下 dvmCheckClassAccess 的逻辑

```

bool dvmCheckClassAccess(const ClassObject* accessFrom,
    const ClassObject* clazz) {
    if (dvmIsPublicClass(clazz)) --> 如果父类是 public 类，直接 return true.
        return true;
    return dvmInSamePackage(accessFrom, clazz);
}
bool dvmInSamePackage(const ClassObject* class1, const ClassObject* class2) {
    /* quick test for intra-class access */
    if (class1 == class2)
        return true;

    /* class loaders must match */
    if (class1->classLoader != class2->classLoader) --> classLoader 不一致，直接
return false
        return false;

    /*
     * Switch array classes to their element types. Arrays receive the
     * class loader of the underlying element type. The point of doing
     * this is to get the un-decorated class name, without all the
     * "[[L...;" stuff.
     */
    if (dvmIsArrayClass(class1))
        class1 = class1->elementClass;
    if (dvmIsArrayClass(class2))

```

```

class2 = class2->elementClass;

/* check again */
if (class1 == class2)
    return true;

/*
 * We have two classes with different names. Compare them and see
 * if they match up through the final '/'.
 *
 * Ljava/lang/Object; + Ljava/lang/Class;          --> true
 * LFoo;           + LBar;                         --> true
 * Ljava/lang/Object; + Ljava/io/File;            --> false
 * Ljava/lang/Object; + Ljava/lang/reflect/Method; --> false
 */
int commonLen;

commonLen = strcmpCount(class1->descriptor, class2->descriptor);
if (strchr(class1->descriptor + commonLen, '/') != NULL ||
    strchr(class2->descriptor + commonLen, '/') != NULL)
{
    return false;
}

return true;
}

```

我们可以看到如果当前类和实现接口 / 父类是非 public, 同时负责加载两者的是 classLoader 不一样的情况下，直接 return false。所以如果此时不进行任何处理的话，那么在类加载阶段就报错。我们当前的代码热修复方案是基于新 classLoader 加载补丁类，所以在 patch 的过程中就会报类似如下的错误。

```
Superclass of '%s' (%s) is not accessible
```

类校验阶段访问权限检查

上一小节我们可以得出结论，在补丁加载阶段，我们就能得到异常，所以我们完全可以在补丁加载失败的情况下，兜底走冷部署方案。事实上并非如此简单，如果补丁类中存在非 public 类的访问 / 非 public 方法 / 域的调用，那么都会导致失败。更为致命的是，在补丁加载阶段是检测不出来的，补丁会被视为正常加载，但是在运行阶

段会直接 crash 异常退出。接下来我们深入分析下。

由于补丁类在单独的 dex 中，所以如果要加载这个 dex 的话，肯定要进行 dexopt 的，dexopt 过程中会执行 dvmVerifyClass 校验 dex 中的每个类。方法调用链：dvmVerifyClass 校验类 → verifyMethod 校验类中的每个方法 → (dvmVerifyCodeFlow→doCodeVerification) 对每个方法的逻辑进行校验 → verifyInstruction 实际上就是校验每个指令

```

static bool verifyInstruction(const Method* meth, InsnFlags* insnFlags,
    RegisterTable* regTable, int insnIdx, UninitInstanceMap* uninitMap,
    int* pStartGuess) {
    ...
    case OP_NEW_INSTANCE:
        resClass = dvmOptResolveClass(meth->clazz, decInsn.vB, &failure);

    case OP_INVOKE_VIRTUAL:
    case OP_INVOKE_SUPER:
    case OP_INVOKE_DIRECT:
    case OP_INVOKE_STATIC:
        calledMethod = verifyInvocationArgs(meth, workLine, insnRegCount,
            &decInsn, uninitMap, METHOD_VIRTUAL, isRange,
            isSuper, &failure);
        if (!VERIFY_OK(failure))
            break;
    ...
    if (!VERIFY_OK(failure)) {
        if (failure == VERIFY_ERROR_GENERIC || gDvm.optimizing) { // 失败的错误
原因只要不是VERIFY_ERROR_GENERIC就会执行else分支。
            /* immediate failure, reject class */
            LOG_VFY METH(meth, "VFY: rejecting opcode 0x%02x at 0x%04x",
                decInsn.opcode, insnIdx);
            goto bail;
        } else {
            /* replace opcode and continue on */
            ALOGD("VFY: replacing opcode 0x%02x at 0x%04x",
                decInsn.opcode, insnIdx);
            if (!replaceFailingInstruction(meth, insnFlags, insnIdx,
                failure)) { // 指令替换
                LOG_VFY METH(meth, "VFY: rejecting opcode 0x%02x at 0x%04x",
                    decInsn.opcode, insnIdx);
                goto bail;
            }
            /* IMPORTANT: meth->insnns may have been changed */
            insnns = meth->insnns + insnIdx;
        }
    }
}

```

```
        /* continue on as if we just handled a throw-verification-error */
*/  
failure = VERIFY_ERROR_NONE;  
nextFlags = kInstrCanThrow;  
}  
}
```

OP_NEW_INSTANCE 指令的校验会通过 dvmOptResolveClass->dvmCheckClassAccess 去检查补丁类所引用的类的访问权限，dvmCheckClassAccess 这个方法前面介绍过，不重复介绍，我们着重介绍下 OP_INVOKE_VIRTUAL 指令的校验。上面我们可以看到调用的是 verifyInvocationArgs->dvmResolveMethod

```
/*
 * Alternate version of dvmResolveMethod() .
 *
 * Doesn't throw exceptions, and checks access on every lookup.
 *
 * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
 */
Method* dvmOptResolveMethod(ClassObject* referrer, u4 methodIdx,
    MethodType methodType, VerifyError* pFailure)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    Method* resMethod;

    assert(methodType == METHOD_DIRECT ||
           methodType == METHOD_VIRTUAL ||
           methodType == METHOD_STATIC);

    resMethod = dvmDexGetResolvedMethod(pDvmDex, methodIdx);
    if (resMethod == NULL) {
        const DexMethodId* pMethodId;
        ClassObject* resClass;

        pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

        resClass = dvmOptResolveClass(referrer, pMethodId->classIdx,
pFailure);
        .....
    }

    bool allowed = dvmCheckMethodAccess(referrer, resMethod); --> 方法的访问权限
}
```

```

untweakLoader(referrer, resMethod->clazz);
if (!allowed) {
    IF ALOGI() {
        char* desc = dexProtoCopyMethodDescriptor(&resMethod->prototype);
        ALOGI("DexOpt: illegal method access (call %s.%s %s from %s)",
              resMethod->clazz->descriptor, resMethod->name, desc,
              referrer->descriptor);
        free(desc);
    }
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_ACCESS_METHOD;
    return NULL;
}

return resMethod;
}

```

我们看到此时调用 dvmCheckMethodAccess 检查调用方法的访问权限。

```

bool dvmCheckMethodAccess(const ClassObject* accessFrom, const Method* method) {
    return checkAccess(accessFrom, method->clazz, method->accessFlags);
}
/*
 * Validate method/field access.
 */
static bool checkAccess(const ClassObject* accessFrom,
    const ClassObject* accessTo, u4 accessFlags) {
    /* quick accept for public access */
    if (accessFlags & ACC_PUBLIC)
        return true;

    /* quick accept for access from same class */
    if (accessFrom == accessTo)
        return true;

    /* quick reject for private access from another class */
    if (accessFlags & ACC_PRIVATE)
        return false;

    /*
     * Semi-quick test for protected access from a sub-class, which may or
     * may not be in the same package.
     */
    if (accessFlags & ACC_PROTECTED)
        if (dvmIsSubClass(accessFrom, accessTo))
            return true;
}

```

```

/*
 * Allow protected and private access from other classes in the same
 * package.
 */
return dvmInSamePackage(accessFrom, accessTo);
}

```

如果该方法是 public, checkAccess 直接返回 true, 如果是 protected, 那么检查调用方法所在的类和当前类是否是父子类的关系, 如果不是调用 dvmInSamePackage, 这个方法我们前面已经介绍过, 如果补丁类和方法所在的类不是 classLoader, 那么返回 false。

所以最后我们能够得到结论, 补丁类如果引用了非 public 类, 那么 verifyInstruction 方法执行的结果 pFailure = VERIFY_ERROR_ACCESS_METHOD, 那么 verifyInstruction 方法接下来就会执行 replaceFailingInstruction 指令替换流程。 replaceFailingInstruction->dvmUpdateCodeUnit 方法更新潜在错误的 **opcode 指令码**, 此时会把上述权限检查失败的指令码比如 invoke-virtual/new-instance 等指令被替换成为了 **OP_THROW_VERIFICATION_ERROR** 指令。

```

static bool replaceFailingInstruction(const Method* meth, InsnFlags*
insnFlags,
int insnIdx, VerifyError failure)
{
    VerifyErrorRefType refType;
    u2* oldInsns = (u2*) meth->insns + insnIdx;
    int width;

    Opcode opcode = dexOpcodeFromCodeUnit(*oldInsns);
    switch (opcode) {

        case OP_NEW_INSTANCE:
            refType = VERIFY_ERROR_REF_CLASS;
            break;

        case OP_IGET: // insn[1] == field ref, 2 bytes
            refType = VERIFY_ERROR_REF_FIELD;
            break;

        case OP_INVOKE_VIRTUAL: // insn[1] == method ref, 3 bytes
        case OP_INVOKE_SUPER:
        case OP_INVOKE_DIRECT:

```

```

        case OP_INVOKE_STATIC:
        case OP_INVOKE_INTERFACE:
            refType = VERIFY_ERROR_REF_METHOD;
            break;
    }

/* encode the opcode, with the failure code in the high byte */
assert(width == 2 || width == 3);
u2 newVal = OP_THROW_VERIFICATION_ERROR |
    (failure << 8) | (refType << (8 + kVerifyErrorRefTypeShift));
dvmUpdateCodeUnit(meth, oldInsns, newVal);
return true;
}

```

注意此时 verifyInstruction 返回的仍然是 true, 所以 dvmVerifyClass 返回的也是 true, 所以在补丁加载的时候是成功的, 我们并不会收到任何的异常。这也是跟补丁类的父类 / 实现的接口非 public 情况下补丁加载的情况下就报异常最大的区别。那么我们接下来看下 `OP_THROW_VERIFICATION_ERROR` 指令会发生什么?

```

HANDLE_OPCODE(OP_THROW_VERIFICATION_ERROR)
EXPORT_PC();
vsrc1 = INST_AA(inst);
ref = FETCH(1); /* class/field/method ref */
dvmThrowVerificationError(curMethod, vsrc1, ref); // 调用
dvmThrowVerificationError 方法
GOTO_exceptionThrown();
OP_END

void dvmThrowVerificationError(const Method* method, int kind, int ref){
    switch ((VerifyError) errorKind) {
    case VERIFY_ERROR_ACCESS_CLASS:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = classNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); // 类权限异常信息 ...
        break;
    case VERIFY_ERROR_ACCESS_FIELD:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = fieldNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); // 域权限异常信息 ...
        break;
    case VERIFY_ERROR_ACCESS_METHOD:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = methodNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); // 方法权限异常信息 ...
        break;
    }
}

```

```
        break;
    ...
}

dvmThrowException(exceptionClass, msg.c_str());
}
```

so, 我们知道了在执行的过程中 `dvmThrowException` 抛异常，程序中断异常退出。

2.2.10 <clinit>方法

由于补丁热部署模式下的特殊性——不允许类结构变更以及不允许变更 `<clinit>` 方法，所以我们的补丁工具如果发现了这几种限制情况，那么此时只能走冷启动重启生效，冷启动几乎是无任何限制的，可以做到任何场景的修复。可能有时候在源码层面上来看并没有新增 / 减少 method 和 field，但是实际上由于要满足 Java 各种语法特性的需求，所以编译器会在编译期间为我们自动合成一些 method 和 field，最后就有可能触发了这几个限制情况。以上列举的情况可能并不完全详细，这些分析也只能是一个抛砖引玉的作用，具体情况还需要具体分析，同时一些难以理解的 java 语法特性或许从编译的角度去分析可能就无处遁形了。

2.3 冷启动类加载原理

前面我们提到热部署修复方案有诸多特点(有关热替换方案的实现请看2.1章节)。其根本原理是基于native层方法的替换，所以当类结构变化时，如新增减少类method/field在热部署模式下会受到限制。但冷部署能突破这种约束，可以更好地达到修复目的，再加上冷部署在稳定性上具有的独特优势，因此可以作为热部署的有力补充而存在。

2.3.1 冷启动实现方案概述

冷启动重启生效，现在一般有以下两种实现方案，同时给出他们各自的优缺点：

	QQ空间	Tinker
原理	为了解决Dalvik下unexpected dex problem异常而采用插桩的方式，单独放一个帮助类在独立的dex中让其他类调用，阻止了类被打上CLASS_ISPREVERIFIED标志从而规避问题的出现。最后加载补丁dex得到dexFile对象作为参数构建一个Element对象插入到dex-Elements数组的最前面。	提供dex差量包，整体替换dex的方案。差量的方式给出patch.dex，然后将patch.dex与应用的classes.dex合并成一个完整的dex，完整dex加载得到的dexFile对象作为参数构建一个Element对象然后整体替换掉旧的dex-Elements数组。
优点	没有合成整包，产物比较小，比较灵活	自研dex差异算法，补丁包很小，dex merge成完整dex，Dalvik不影响类加载性能，Art下也不存在必须包含父类/引用类的情况
缺点	Dalvik下影响类加载性能，Art下类地址写死，导致必须包含父类/引用，最后补丁包很大	dex合并内存消耗在vm heap上，容易OOM，最后导致dex合并失败。

上面的表格，我们能清晰的看到两个方案的缺点都很明显。这里对tinker方案dex merge缺陷进行简单说明一下：

dex merge操作是在java层面进行，所有对象的分配都是在java heap上，如果此时进程申请的java heap对象超过了vm heap规定的大小，那么进程发生OOM，那么系统memory killer可能会杀掉该进程，导致dex合成失败。另外一方面我们知道jni层面C++ new/malloc申请的内存，分配在native heap，native

heap 的增长并不受 vm heap 大小的限制，只受限于 RAM，如果 RAM 不足那么进程也会被杀死导致闪退。所以如果只是从 dex merge 方面思考，在 jni 层面进行 dex merge，从而可以避免 OOM 提高 dex 合并的成功率。理论上当然可以，只是 jni 层实现起来比较复杂而已

文章的开头我们说过，我们的需求是冷启动模式是热部署模式的补充兜底方案，所以这两个方案使用的应该是同一套补丁，另外一个方面跟代码修复热部署方案一样，我们追求的是不侵入打包。上述两种方案都需要侵入应用打包过程，同时补丁的结构也不一样，这两套方案对我们来说都是不适用。所以我们需要另辟蹊径冷启动修复，寻求一种既能无侵入打包又能做热部署模式下兜底补充的解决方案，下面将对 Dalvik 虚拟机和 Art 虚拟机的冷启动方案分别进行介绍。

2.3.2 插桩实现的前因后果

众所周知，如果仅仅把补丁类打入补丁包中而不做任何处理的话，那么运行时类加载的时候就会异常退出，接下来先来看下抛这个异常的前因后果。

加载一个 dex 文件到本地内存的时候，如果不存在 odex 文件，那么首先会执行 dexopt，dexopt 的入口在 `davilk/opt/OptMain.cpp` 的 main 方法，最后调用到 `verifyAndOptimizeClass` 执行真正的 verify/optimize 操作。

```
/*
 * Verify and/or optimize a specific class.
 */
static void verifyAndOptimizeClass(DexFile* pDexFile, ClassObject* clazz,
    const DexClassDef* pClassDef, bool doVerify, bool doOpt) {
    const char* classDescriptor;
    bool verified = false;
    classDescriptor = dexStringByTypeId(pDexFile, pClassDef->classIdx);

    if (doVerify) {
        if (dvmVerifyClass(clazz)) { // 执行类的 Verify
            ((DexClassDef*)pClassDef)->accessFlags |= CLASS_ISPREVERIFIED; // 类被打上 CLASS_ISPREVERIFIED 标志
            verified = true;
        }
    }
}
```

```

    }

    if (doOpt) {
        bool needVerify = (gDvm.dexOptMode == OPTIMIZE_MODE_VERIFIED || gDvm.
dexOptMode == OPTIMIZE_MODE_FULL);
        if (!verified && needVerify) {
            .....
        } else {
            dvmOptimizeClass(clazz, false); // 执行类的 Optimize
            ((DexClassDef*)pClassDef)->accessFlags |= CLASS_ISOPTIMIZED; // 类
被打上 CLASS_ISOPTIMIZED 标志
        }
    }
}

```

apk 第一次安装的时候，会对原 dex 执行 dexopt，此时假如 apk 只存在一个 dex，所以 dvmVerifyClass(clazz) 结果为 true。所以 apk 中所有的类都会被打上 CLASS_ISPREVERIFIED 标志，接下来执行 dvmOptimizeClass，类接着被打上 CLASS_ISOPTIMIZED 标志。

- dvmVerifyClass: 类校验，类校验的目的简单来说就是为了防止类被篡改校验类的合法性。此时会对类的每个方法进行校验，这里我们只需要知道如果类的所有方法中直接引用到的类（第一层级关系，不会进行递归搜索）和当前类都在同一个 dex 中的话，dvmVerifyClass 就返回 true。
- dvmOptimizeClass: 类优化，简单来说这个过程会把部分指令优化成虚拟机内部指令，比如方法调用指令：`invoke-*` 指令变成了 `invoke-*-quick`，`quick` 指令会从类的 vtable 表中直接取，vtable 简单来说就是类的所有方法的一张大表（包括继承自父类的方法）。因此加快了方法的执行速率。

现在假如 A 类是补丁类，所以补丁 A 类在单独的 dex 中。类 B 中的某个方法引用到补丁类 A，所以执行到该方法会尝试解析类 A

```

ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool
fromUnverifiedConstant) {
    .....
    if (!fromUnverifiedConstant && IS_CLASS_FLAG_SET(referrer, CLASS_
ISPREVERIFIED)){ // 如果类被打上了 CLASS_ISPREVERIFIED 标志

```

```

    if (referrer->pDvmDex != resClassCheck->pDvmDex && resClassCheck-
>classLoader != NULL) {
    dvmThrowIllegalAccessError("Class ref in pre-verified class
resolved to unexpected implementation"); // 抛异常
    return NULL;
}
}
.....
}

```

上面的代码很容易看出来，类 B 由于被打上了 CLASS_ISPREVERIFIED 标志，接下来 referrer 是类 B，resClassCheck 是补丁类 A，他们属于不同的 dex。所以 dvmThrowIllegalAccessError。为了解决这个问题，一个单独无关帮助类放到一个单独的 dex 中，原 dex 中所有类的构造函数都引用这个类，一般的实现方法都是侵入 dex 打包流程，利用 .class 字节码修改技术，在所有 .class 文件的构造函数中引用这个帮助类，插桩由此而来。根据前面的介绍，dexopt 过程中 dvmVerifyClass 类校验返回 false，原 dex 中所有的类都没有 CLASS_ISPREVERIFIED 标志，因此解决运行时这个异常。

但是插桩是会给类加载效率带来比较严重的影响的。熟悉 Dalvik 虚拟机的同学知道，一个类的加载通常有三个阶段，dvmResolveClass->dvmLinkClass->dvmInitClass，这个三个阶段不一一详细进行说明。dvmInitClass 阶段在类解析完毕尝试初始化类的时候执行，这个方法主要完成父类的初始化，当前类的初始化，static 变量的初始化赋值等等操作。

```

bool dvmInitClass(ClassObject* clazz) {
    if (clazz->status < CLASS_VERIFIED) { // 如果类没被打上 CLASS_ISPREVERIFIED 标志
        clazz->status = CLASS_VERIFYING;
        if (!dvmVerifyClass(clazz)) { // 类 Verify
            ...
        }
        clazz->status = CLASS_VERIFIED;
    }
    if (!IS_CLASS_FLAG_SET(clazz, CLASS_ISOPTIMIZED) && !gDvm.optimizing) {
        // 如果类没被打上 CLASS_ISOPTIMIZED 标志
        dvmOptimizeClass(clazz, essentialOnly); // 类 Optimize
        SET_CLASS_FLAG(clazz, CLASS_ISOPTIMIZED);
    }
}

```

```
    } ....
```

可以看到除了上面说的类初始化之外，如果类没被打上 CLASS_ISPREVERIFIED/CLASS_ISOPTIMIZED 标志，那么类的 Verify 和 Optimize 都将在类的初始化阶段进行。正常情况下类的 Verify 和 Optimize 都仅仅只是在 apk 第一次安装执行 dexopt 的时候进行，类的 Verify 实际上是很重的，因为会对类的所有方法中的所有指令都进行校验，单个类加载来看类 Verify 并不耗时，但是如果同一时间点加载大量类的情况下，这个耗时就会被放大。所以这也是插桩给类的加载效率带来比较大影响的后果，接下来来看下具体会给类加载带来多大的影响。

2.3.3 插桩导致类加载性能影响

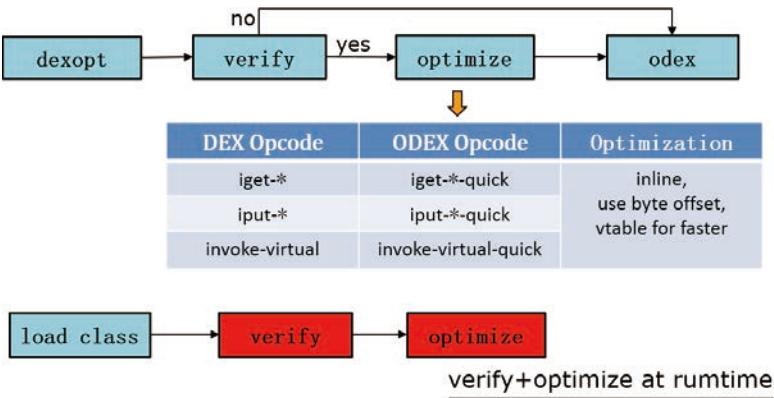


图 2-7 dexopt 过程

上一小节的介绍，我们知道若采用插桩导致所有类都非 preverify，这导致 verify 与 optimize 操作会在加载类时触发。这就会导致类加载有一定的性能损耗，微信做过一次测试，分别采用优化和不优化两种方式做过两种测试，分别采用插桩与不插桩两种方式进行两种测试，一是连续加载 700 个 50 行左右的类，一是统计应用启动完成的整个耗时。

	不插桩	插桩
700 个类	84ms	685ms
启动耗时	4934ms	7240ms

平均每个类 verify+optimize(跟类的大小有关系) 的耗时并不长，而且这个耗时每个类只有一次 (类只会加载一次)。但由于应用刚启动时这种场景下一般会同时加载大量的类，在这个情况影响还是比较大的，启动的时候就容易白屏，这点是没法容忍的。

2.3.4 避免插桩的 QFix 方案

手 Q 热补丁轻量级 QFix 方案提供了一种不同的思路，简单来讲：

```

ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
    bool fromUnverifiedConstant)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const char* className;

    /*
     * Check the table first -- this gets
     * methods.
     */
    resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
    if (resClass == NULL)
        return resClass;

    className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
    if (className[0] != '\0' && className[1] == '\0') {
        /* primitive type */
        resClass = dvmFindPrimitiveClass(className[0]);
    } else {
        resClass = dvmFindClassNoInit(className, referrer->classLoader);
    }

    // 提前把patch类加入到pDvmDex.pResClasses
// 数组, retClass结果不为null
// 设为true, 跳过dex一致性校验
    if (resClass != NULL) {
        if (!fromUnverifiedConstant &&
            IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
        {
            ClassObject* resClassCheck = resClass;
            if (dvmIsArrayClass(resClassCheck))
                resClassCheck = resClassCheck->elementClass;

            if (referrer->pDvmDex != resClassCheck->pDvmDex &&
                resClassCheck->classLoader != NULL)
            {
                dvmThrowIllegalAccessError(
                    "Class ref in pre-verified class resolved to unexpected "
                    "implementation");
                return NULL;
            }
        }
        dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
    }
}

return resClass;
}

```

图 2-8 dvmResolveClass 绕过

怎么让 dvmDexGetResolvedClass 返回的结果不为 null，只要调用过一次 `dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);` 就行了，举个例子简单说明下。

```
public class B {
    public static void test() {
        A.a();
    }
}
```

我们此时需要 patch 的类是类 A，所以类 A 被打入到一个独立的补丁 dex 中。那么执行到类 B 的 test 方法时，执行到 A.a() 这行代码时就会尝试去解析类 A，此时 `dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool fromUnverifiedConstant)`

- referrer: 实际上就是类 B
- classIdx: 类 A 在原 dex 文件结构类区中的索引 id
- fromUnverifiedConstant: 是否 const-class/instance-of 指令

此时是调用的是 A 的静态 a 方法，`invoke-static` 指令不属于 `const-class/instance-of` 这两个指令中的一个。不做任何处理的话，`dvmDexGetResolvedClass` 一开始是 null 的。然后 A 是从补丁 dex 中解析加载，B 是在原 Dex 中，A 在补丁 dex 中，所以 `B->pDvmDex != A->pDvmDex`，接下来执行到 `dvmThrowIllegalAccessError` 从而导致运行时异常。所以我们要做的是，必须要在一开始的时候，就把补丁 A 类添加到原来 dex(pDvmDex) 的 pResClasses 数组中。这样就确保了执行 B 类 test 方法的时候，`dvmDexGetResolvedClass` 不为 null，就不会执行后面类 A 和类 B 的 dex 一致性校验了。

具体实现，首先我们通过补丁工具反编译 dex 为 smali 文件拿到：

- preResolveClz: 需要 patch 的类 A 的描述符，非必须，为了调试方便加上该参数而已。--> Lcom/taobao/patch/demo/A;

- refererClz: 需要 patch 的类 A 所在的 dex 的任何一个类描述符, 注意这里不限定必须是引用补丁类 A 的某个类, 实际上只要同一个 dex 中的任何一个类都可以。所以我们直接拿原 dex 中的第一个类即可。--> Landroid/support/annotation/AnimRes;
- classIdx: 需要 patch 的类 A 在原来 dex 文件中的类索引 id。--> 2425

然后通过 dlopen 拿到 libdvm.so 库的句柄, 然后通过 dlsym 拿到该 so 库的 `dvmResolveClass/dvmFindLoadedClass` 函数指针。首先需要预加载引用类 ->`android/support/annotation/AnimRes`, 这样 `dvmFindLoadedClass ("android/support/annotation/AnimRes")` 才不为 null, `dvmFindLoadedClass` 执行结果得到的 ClassObject 做为第一个参数执行 `dvmResolveClass (AnimRes, 2425, true)` 即可。

简单看下 JNI 层代码部分实现。实际上可以看到 `preResolveClz` 参数是非必须的

```
jboolean resolveColdPatchClasses(JNIEnv *env, jclass clz, jstring
preResolveClz, jstring refererClz,
                                    jlong classIdx, dexstuff_t *dexstuff) {
    LOGD("start resolveColdPatchClasses");
    ClassObject *refererObj = dexstuff->dvmFindLoadedClass_fnPtr(
        Jstring2CStr(env, refererClz)); // 调用 dvmFindLoadedClass
    LOGD("referrer ClassObject: %s\n", refererObj->descriptor);
    if (strlen(refererObj->descriptor) == 0) {
        return JNI_FALSE;
    }

    ClassObject *resolveClass = dexstuff->dvmResolveClass_fnPtr(refererObj,
classIdx, true); // 调用 dvmResolveClass
    LOGD("classIdx ClassObject: %s\n", resolveClass->descriptor);
    if (strlen(resolveClass->descriptor) == 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

完美解决。这个思路与前面方案一的 native hook 方式不同, 不会去 hook 某个系统方法, 而是从 native 层直接调用, 同时更不需要插桩。具体实现需要注意以下三点:

- dvmResolveClass 的第三个参数 fromUnverifiedConstant 必须为 true。
- apk 多 dex 情况下，dvmResolveClass 第一个参数 referrer 类必须跟需要 patch 的类在同一个 dex，但是他们两个类不需要存在任何引用关系，任何一个在同一个 dex 中的类作为 referrer 都可以。
- referrer 类必须提前加载。

然而，QFix 的方案有它独特的缺陷，由于是在 dexopt 后进行绕过的，dexopt 会改变原先的很多逻辑，许多 odex 层面的优化会写死字段和方法的访问偏移，这就会导致比较严重的 BUG，2.4 章节会详细解释这一影响。最后我们采用的是我们自研的全量 DEX 方案，具体可看后面的 2.5 章节。

2.3.5 Art 下冷启动实现

前面说过补丁热部署模式下是一个完整的类，补丁的粒度是类。现在我们的需求是补丁既能走热部署模式也能走冷启动模式，为了减少补丁包的大小，并没有为热部署和冷启动分别准备一套补丁，而是同一个热部署模式下的补丁能够降级直接走冷启动，所以我们不需要做 dex merge。但是前面我们知道为了解决 Art 下类地址写死的问题，tinker 通过 dex merge 成一个全新完整的新 dex 整个替换掉旧的 dexElements 数组。事实上我们并不需要这样做，Art 虚拟机下面默认已经支持多 dex 压缩文件的加载了。

我们分别来看下 Dalvik 下和 Art 下对 `DexFile.loadDex` 尝试把一个 dex 文件解析加载到 native 内存都发生了什么，实际上都是调用了 `DexFile.openDexFileNative` 这个 native 方法。看下 Native 层对应的 c/c++ 代码具体实现。

Dalvik 虚拟机下面：

```
static void Dalvik_dalvik_system_DexFile_openDexFileNative(const u4* args,
JValue* pResult) {
    if (hasDexExtension(sourceName)
        && dvmRawDexFileOpen(sourceName, outputName, &pRawDexFile, false)
== 0) { // 加载一个原始 dex 文件
```

```

ALOGV("Opening DEX file '%s' (DEX)", sourceName);

pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
pDexOrJar->isDex = true;
pDexOrJar->pRawDexFile = pRawDexFile;
pDexOrJar->pDexMemory = NULL;
} else if (dvmJarFileOpen(sourceName, outputName, &pJarFile, false) == 0)
{ // 加载一个压缩文件
    ALOGV("Opening DEX file '%s' (Jar)", sourceName);

    pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
    pDexOrJar->isDex = false;
    pDexOrJar->pJarFile = pJarFile;
    pDexOrJar->pDexMemory = NULL;
} else {
    ALOGV("Unable to open DEX file '%s'", sourceName);
    dvmThrowIOException("unable to open DEX file");
}
}

int dvmJarFileOpen(const char* fileName, const char* odexOutputName,
JarFile** ppJarFile, bool isBootstrap){
...
else {
    ZipEntry entry;
tryArchive:
    /*
     * Pre-created .odex absent or stale. Look inside the jar for a
     * "classes.dex".
     */
    entry = dexZipFindEntry(&archive, kDexInJarName); //  

kDexInJarName=="classes.dex", 说明只加载一个dex
    ....
}
}
}

```

static const char* kDexInJarName = "classes.dex"; 很明显 Dalvik 尝试加载一个压缩文件的时候只会去把 `classes.dex` 加载到内存中。如果此时压缩文件中有多 dex，那么除了 `classes.dex` 之外的其它 dex 被直接忽略掉

Art 虚拟机下面：方法调用链 `DexFile_openDexFileNative-> OpenDex-FilesFromOat -> LoadDexFiles`

```

std::vector<std::unique_ptr<const DexFile>> OatFileAssistant::LoadDexFiles(
    const OatFile& oat_file, const char* dex_location) {

```

```

// Load the primary dex file.
const OatFile::OatDexFile* oat_dex_file = oat_file.GetOatDexFile(
    dex_location, nullptr, false);
std::unique_ptr<const DexFile> dex_file = oat_dex_file->OpenDexFile(&error_
msg);
dex_files.push_back(std::move(dex_file));

// Load secondary multidex files
for (size_t i = 1; ; i++) {
    std::string secondary_dex_location = DexFile::GetMultiDexLocation(i, dex_
location);
    oat_dex_file = oat_file.GetOatDexFile(secondary_dex_location.c_str(),
nullptr, false);
    dex_file = oat_dex_file->OpenDexFile(&error_msg);
    dex_files.push_back(std::move(dex_file));
}
return dex_files;
}

```

上面代码我们大概可以看出来 Art 下面默认已经支持加载压缩文件中包含多个 dex，首先肯定优先加载 primary dex 其实就是 `classes.dex`，后续会加载其它的 dex。所以补丁类只需要放到 `classes.dex` 即可，后续出现在其它 dex 中的 "补丁类" 是不会被重复加载的。所以我们得到 Art 下最终的冷启动解决方案：我们只要把补丁 `dex` 命名为 `classes.dex`。原 apk 中的 `dex` 依次命名为 `classes(2,3,4...).dex` 就好了，然后一起打包为一个压缩文件。然后 `DexFile.loadDex` 得到 `DexFile` 对象，最后把该 `DexFile` 对象整个替换旧的 `dexElements` 数组就可以了。

一张图来看下我们的方案和 tinker 方案的不同：

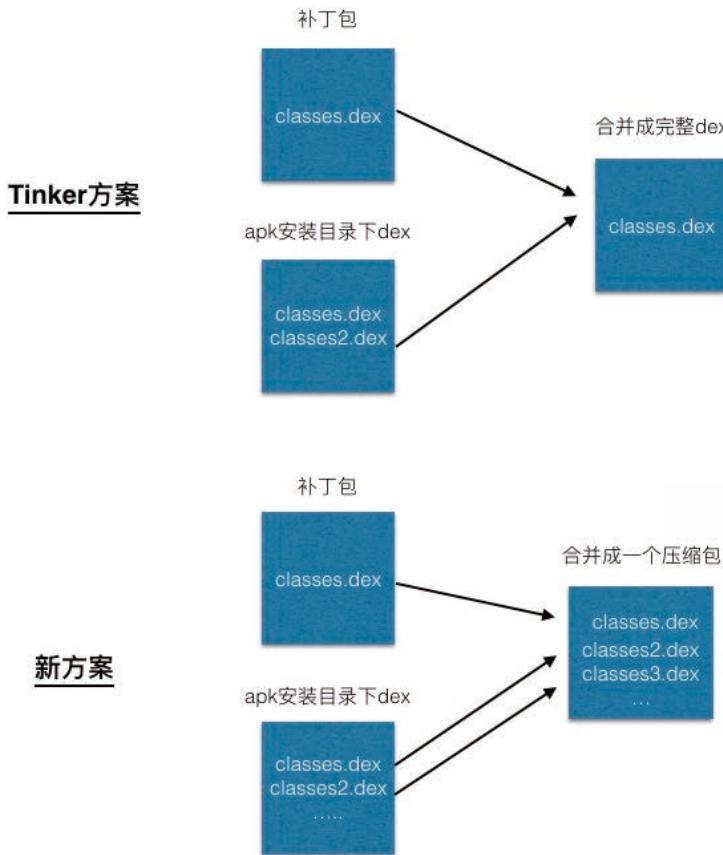


图 2-9 方案不同点

需要注意一点：

- 补丁 dex 必须命名为 `classes.dex`
- `loadDex` 得到的 DexFile 完整替换掉 `dexElements` 数组而不是插入

2.3.6 不得不说的其它点

我们知道 `DexFile.loadDex` 尝试把一个 dex 文件解析并加载到 native 内存，在加载到 native 内存之前，如果 dex 不存在对应的 odex，那么 Dalvik 下会执行

dexopt, Art 下会执行 dexoat, 最后得到的都是一个优化后的 odex。实际上最后虚拟机执行的是这个 `odex` 而不是 `dex`。

现在有这么一个问题, 如果 dex 足够大那么 dexopt/dexoat 实际上是很耗时的, 根据上面我们提到的方案, Dalvik 下实际上影响比较小, 因为 loadDex 仅仅是补丁包。但是 Art 下影响是非常大的, 因为 loadDex 是补丁 dex 和 apk 中原 dex 合并成的一个完整补丁压缩包, 所以 dexoat 非常耗时。所以如果优化后的 odex 文件没生成或者没生成一个完整的 odex 文件, 那么 loadDex 便不能在应用启动的时候进行的, 因为会阻塞 loadDex 线程, 一般是主线程。所以为了解决这个问题, 我们把 `loadDex` 当做一个事务来看, 如果中途被打断, 那么就删除 `odex` 文件, 重启的时候如果发现存在 `odex` 文件, `loadDex` 完之后, 反射注入 / 替换 `dexElements` 数组, 实现 `patch`。如果不存在 `odex` 文件, 那么重启另一个子线程 `loadDex`, 重启之后再生效。

另外一方面为了 patch 补丁的安全性, 虽然对补丁包进行签名校验, 这个时候能够防止整个补丁包被篡改, 但是实际上因为虚拟机执行的是 odex 而不是 dex, 还需要对 `odex` 文件进行 md5 完整性校验, 如果匹配, 则直接加载。不匹配, 则重新生成一遍 odex 文件, 防止 odex 文件被篡改。

2.3.7 完整的方案考虑

代码修复冷启动方案由于它的高兼容性, 几乎可以修复任何代码修复的场景, 但是注入前被加载的类(比如 Application 类)肯定是不能被修复的。所以我们把它作为一个兜底的方案, 在没法走热部署或者热部署失败的情况下, 最后都会走代码冷启动重启生效, 所以我们的补丁是同一套的。具体实施方案对 Dalvik 下和 Art 下分别做了处理:

- Dalvik 下采用我们自行研发的全量 DEX 方案。
- Art 下本质上虚拟机已经支持多 dex 的加载, 我们要做的仅仅是把补丁 dex 作为主 dex(classes.dex) 加载而已。

2.4 多态对冷启动类加载的影响

前面我们知道冷启动方案几乎是可以修复任何场景的，但 Dalvik 下 QFix 方案存在很大的限制，下面将深入介绍下目前方案下为什么会有这些限制，同时给出具体的解决方案。

2.4.1 重新认识多态

实现多态的技术一般叫做动态绑定，是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。多态一般指的是非静态非 private 方法的多态。field 和静态方法不具有多态性。示例如下。

```
public class B extends A {
    String name = "B name";

    @Override
    void a_t1() {
        System.out.println("B a_t1...");
    }
    void b_t1(){}
}

public static void main(String[] args) {
    A obj = new B();
    System.out.println(obj.name);
    obj.a_t1();
}
}

class A {
    String name = "A name";

    void a_t1() {
        System.out.println("A a_t1...");
    }
    void a_t2(){}
}
```

输出结果：`A name/B a_t1...`，可以看到 name 这个 field 没有多态性，print 这个方法具有多态性，这里首先分析下方法多态性的实现。首先 `new B()` 的执行会尝试加载类 B，方法调用链 `dvmResolveClass->dvmLinkClass->creat-`

`eVtable`, 此时会为类 B 创建一个 vtable, 其实在虚拟机中加载每个类都会为这个类生成一张 `vtable` 表, `vtable` 表说白了就是当前类的所有 `virtual` 方法的一个数组, 当前类和所有继承父类的 `public/protected/default` 方法就是 `virtual` 方法。因为 `public/protected/default` 修饰的方法是可以被继承的。`private/static` 方法不属于这个范畴, 因为不能被继承。

```

/*
 * Create the virtual method table.
 *
 * The top part of the table is a copy of the table from our superclass,
 * with our local methods overriding theirs. The bottom part of the table
 * has any new methods we defined.
 */
static bool createVtable(ClassObject* clazz){
    bool result = false;
    int maxCount;
    int i;

    /* the virtual methods we define, plus the superclass vtable size */
    maxCount = clazz->virtualMethodCount;
    if (clazz->super != NULL) {
        maxCount += clazz->super->vtableCount;// 如果父类不为 null, 那么当前类的
        vtable 表的大小就是当前类的 virtual 方法总数加上父类的 vtableCount
    }
    /*
     * Over-allocate the table, then realloc it down if necessary. So
     * long as we don't allocate anything in between we won't cause
     * fragmentation, and reducing the size should be unlikely to cause
     * a buffer copy.
    */
    dvmLinearReadWrite(clazz->classLoader, clazz->virtualMethods);
    clazz->vtable = (Method**) dvmLinearAlloc(clazz->classLoader,
    sizeof(Method*) * maxCount); // 内存分配。vtable 的大小为 maxCount

    if (clazz->super != NULL) { // 存在父类 ...
        int actualCount;

        memcpy(clazz->vtable, clazz->super->vtable,
               sizeof(*(clazz->vtable)) * clazz->super->vtableCount); // 很关键的
        // 一步, 一开始把父类所有的 vtable 都复制给子类。所有子类的 vtable 中存在所有继承自父类的方法
        actualCount = clazz->super->vtableCount;

        /*
         * See if any of our virtual methods override the superclass.
        */
    }
}

```

```

        for (i = 0; i < clazz->virtualMethodCount; i++) { // 遍历子类自身所有的
virtual 方法
            Method* localMeth = &clazz->virtualMethods[i];
            int si;

            for (si = 0; si < clazz->super->vtableCount; si++) {
                Method* superMeth = clazz->vtable[si];

                if (dvmCompareMethodNamesAndProtos(localMeth, superMeth) ==
0) { // 如果子类 virtual 方法签名和父类一样，说明该方法被覆盖了。子类重写方法覆盖掉 vtable
中父类的方法
                    .....
                    clazz->vtable[si] = localMeth; // 所以 vtable 中 si 的索引必须
换成子类覆盖的方法 !!!
                    localMeth->methodIndex = (u2) si; //methodIndex 也就是方法在
vtable 中的索引值 ...
                    break;
                }
            }

            if (si == clazz->super->vtableCount) { // 方法签名和父类的不一致，说明
是子类自己的方法
                /* not an override, add to end */
                clazz->vtable[actualCount] = localMeth; // 添加到子类 vtable 的末尾
                localMeth->methodIndex = (u2) actualCount;
                actualCount++;
            }
        }

        .....
    } else { // 没有继承自任何类 .....
        /* java/lang/Object case */
        int count = clazz->virtualMethodCount;
        if (count != (u2) count) {
            ALOGE("Too many methods (%d) in base class '%s'", count,
                  clazz->descriptor);
            goto bail;
        }

        for (i = 0; i < count; i++) {
            clazz->vtable[i] = &clazz->virtualMethods[i];
            clazz->virtualMethods[i].methodIndex = (u2) i;
        }
        clazz->vtableCount = clazz->virtualMethodCount;
    }
}
}

```

上面的代码注释解释的很清楚，子类 vtable 的大小等于子类 virtual 方法数 + 父类 vtable 的大小。

- 整个复制父类 vtable 到子类的 vtable
- 遍历子类的 virtual 方法集合，如果方法原型一致，说明是重写父类方法，那么相同索引位置处，子类重写方法覆盖掉 vtable 中父类的方法
- 方法原型不一致，那么把该方法添加到 vtable 的末尾

所以上述示例中，假如父类 A 的 vtable 是 `vtable[0]=A.a_t1, vtable[1]=A.a_t2` 方法，那么 B 类的 vtable 就是 `vtable[0]=B.a_t1, vtable[1]=A.a_t2, vtable[2]=B.b_t1`。接下来看下 `obj.a_t1()` 发生了什么，`invokeVirtual` 指令的解释

```

GOTO_TARGET(invoker, bool methodCallRange, bool) {
    Method* baseMethod;
    Object* thisPtr;

    EXPORT_PC();

    vsrcl = INST_AA(inst);      /* AA (count) or BA (count + arg 5) */
    ref = FETCH(1);            /* method ref */
    vdst = FETCH(2);           /* 4 regs -or- first reg */

    /*
     * The object against which we are executing a method is always
     * in the first argument.
     */
    if (methodCallRange) {
        thisPtr = (Object*) GET_REGISTER(vdst);
    } else {
        thisPtr = (Object*) GET_REGISTER(vdst & 0x0f); // 当前对象
    }

    /*
     * Resolve the method. This is the correct method for the static
     * type of the object. We also verify access permissions here.
     */
    baseMethod = dvmDexGetResolvedMethod(methodClassDex, ref); // 是否已经解析过
    该方法
    if (baseMethod == NULL) {
        baseMethod = dvmResolveMethod(curMethod->clazz, ref, METHOD_VIRTUAL);
    // 没有解析过该方法调用 dvmResolveMethod, baseMethod 得到的当然是 A.a_t1 方法。
        if (baseMethod == NULL) {
            ILOGV("+ unknown method or access denied");
            GOTO_exceptionThrown();
        }
    }
}

```

```

    }

    /*
     * Combine the object we found with the vtable offset in the
     * method.
     */
    assert(baseMethod->methodIndex < thisPtr->clazz->vtableCount);
    methodToCall = thisPtr->clazz->vtable[baseMethod->methodIndex]; //A.a_t1
方法在类 A 的 vtable 中的索引去类 B 的 vtable 中查找

.....
GOTO_invokeMethod(methodCallRange, methodToCall, vsrcl, vdsl);
}
GOTO_TARGET_END

```

首先 obj 引用类型是基类 A，所以上述代码中 baseMethod 拿到的 A.a_t1，
 baseMethod->methodIndex 是该方法在类 A 的 vtable 中的索引 0，obj 的**实际类型**
 是类 B，所以 thisPtr->clazz 就是类 B，那么 B.vtable[0] 就是 B.a_t1 方法，
 所以 obj.a_t1() 实际最后调用的是 B.a_t1 方法。这样就实现了方法的多态。

至于 field/static 方法为什么不具有多态性，这里不详细代码分析，有需要的可以看 `iget/invoke-static` 指令的解释，简单来讲，**是从当前变量的引用类型而不是实际类型中查找，如果找不到再去父类中递归查找**。所以 field 和 static 方法不具备多态性。

2.4.2 冷启动方案限制

我们来看下如果新增了一个 public/protected/default 方法，会出现什么情况。

```

public class Demo {
    public static void test_addMethod() {
        A obj = new A();
        obj.a_t2();
    }
}
class A {
    int a = 0;

    // 新增 a_t1 方法
}

```

```

void a_t1() {
    Log.d("Sophix", "A a_t1");
}
void a_t2() {
    Log.d("Sophix", "A a_t2");
}
}

```

修复后的 apk 中新增了 `a_t1()` 方法，Demo 类不做任何修复，测试发现应用补丁后 `Demo.test_addMethod()` 得到的结果竟然是 D/Sophix: A a_t1，这表明 `obj.a_t2` 执行的竟然是 `a_t1` 方法，简直匪夷所思。下面深入分析下本质原因

在 2.3 章节我们有提到过，dex 文件第一次加载的时候，会执行 dexopt，dexopt 有两个过程：verify+optimize。

- `dvmVerifyClass`: 类校验，类校验的目的简单来说就是为了防止类被篡改校验类的合法性。此时会对类的每个方法进行校验，这里我们只需要知道如果类的所有方法中直接引用到的类（第一层级关系，不会进行递归搜索）和当前类都在同一个 dex 中的话，`dvmVerifyClass` 就返回 true。
- `dvmOptimizeClass`: 类优化，简单来说这个过程会把部分指令优化成虚拟机内部指令，比如方法调用指令：`invoke-virtual-quick`，`quick` 指令会从类的 `vtable` 表中直接取，`vtable` 简单来说就是类的所有方法的一张大表（包括继承自父类的方法）。因此加快了方法的执行速率。

这里主要介绍下 optimize 阶段

```

void dvmOptimizeClass(ClassObject* clazz, bool essentialOnly) {
    int i;
    for (i = 0; i < clazz->directMethodCount; i++) {
        optimizeMethod(&clazz->directMethods[i], essentialOnly);
    }
    for (i = 0; i < clazz->virtualMethodCount; i++) {
        optimizeMethod(&clazz->virtualMethods[i], essentialOnly);
    }
}
static void optimizeMethod(Method* method, bool essentialOnly) { // 优化类的方法
    .....
}

```

```

/*
 * non-essential substitutions:
 * invoke-{virtual,direct,static} [/range] --> execute-inline
 * invoke-{virtual,super} [/range] --> invoke-*-quick
 */
if (!matched && !essentialOnly) {
    switch (opc) {
        case OP_INVOKE_VIRTUAL:
            if (!rewriteExecuteInline(method, insns, METHOD_VIRTUAL)) {
                rewriteVirtualInvoke(method, insns, OP_INVOKE_VIRTUAL_
QUICK); // 重写 invoke-virtual 为虚拟机内部指令 invoke-virtual-quick
            }
            break;
        ...
    }
}
.....
}

```

注释已经很清楚了，重写 `invoke-virtual` 为虚拟机内部指令 `invoke-virtual-quick`，这个指令后面跟的立即数就是该方法在类 `vtable` 中索引值。

```

GOTO_TARGET(invokerVirtualQuick, bool methodCallRange) {
    Object* thisPtr;
    EXPORT_PC();
    vsrc1 = INST_AA(inst);      /* AA (count) or BA (count + arg 5) */
    ref = FETCH(1);           /* vtable index */
    vdst = FETCH(2);          /* 4 regs -or- first reg */

    /*
     * The object against which we are executing a method is always
     * in the first argument.
     */
    if (methodCallRange) {
        assert(vsrc1 > 0);
        ILOGV("| invoke-virtual-quick-range args=%d @0x%04x {regs=v%d-v%d}",
              vsrc1, ref, vdst, vdst+vsrc1-1);
        thisPtr = (Object*) GET_REGISTER(vdst);
    } else {
        assert((vsrc1>>4) > 0);
        ILOGV("| invoke-virtual-quick args=%d @0x%04x {regs=0x%04x %x}",
              vsrc1 >> 4, ref, vdst, vsrc1 & 0x0f);
        thisPtr = (Object*) GET_REGISTER(vdst & 0x0f);
    }

    /*
     * Combine the object we found with the vtable offset in the
     * method.
     */
}

```

```

*/
assert(ref < (unsigned int) thisPtr->clazz->vtableCount);
methodToCall = thisPtr->clazz->vtable[ref]; // 直接从变量的实际类型的 vtable
中拿方法
.....
GOTO_invokeMethod(methodCallRange, methodToCall, vsrcl, vdst);
}
GOTO_TARGET_END

```

`invoke-virtual-quick` 很明显相比 `invoke-virtual` 效率更高，直接从实际类型的 vtable 中拿到调用方法的指针而省略了 `dvmResolveMethod` 从变量的引用类型拿到该方法在 vtable 索引 id 的步骤，所以更高效。

现在我们很容易知道上面代码示例，方法调用错乱的发生的本质原因了。patch 前类 A 的 vtable 值是 `vtable[0]=a_t2`。patch 后类 A 新增了 `a_t1` 方法，那么类 A 的 vtable 值 `vtable[0]=a_t1, vtable[1]=a_t2`。但是 `obj.a_t2()` 这行代码在 odex 中的指令其实是 `invoke-virtual-quick A.vtable[0]`，所以 patch 前调用的是 `a_t2` 方法，patch 后调用的是 `a_t1` 方法，导致了方法调用错乱。

2.4.3 终极解决方案

可见，由于多态的影响，QFix 的方案走不通了。我们最后的希望就寄托于类似 tinker 方案的完整 DEX 解决方案了。

利用 google 已经开源的 dexmerge 方案，把补丁 dex 和原 dex 合并成一个完整的 dex 似乎可行的，但仅仅这样还是不够的，多 dex 下如果 dexmerge 会有 65535 方法数超了的异常，dexmerge 会导致内存风暴，内存不足情况下容易失败。完整的 DEX 合成要求在移动端进行，实现较为复杂。

因此，我们最后自研了一套完善的完整 DEX 方案，具体是如何实现的呢？就让我们进入 2.5 章节，来揭开这一谜底。

2.5 Dalvik 下完整 DEX 方案的新探索

2.5.1 冷启动类加载修复

对于 Android 下的冷启动类加载修复，最早实现的方案是 QQ 空间提出的 dex 插入方案。该方案的主要思想，就是把插入新 dex 插入到 ClassLoader 索引路径的最前面。这样在 load 一个 class 时，就会优先找到补丁中的。后来微信的 Tinker 和手 Q 的 QFix 都基于该方案做了改进，而这类插入 dex 的方案，都会遇到一个主要的问题，就是如何解决 Dalvik 虚拟机下类的 pre-verify 问题。

如果一个类中直接引用到的所有非系统类都和该类在同一个 dex 里的话，那么这个类就会被打上 `CLASS_ISPREVERIFIED`，具体判定代码可见虚拟机中的 `verifyAndOptimizeClass` 函数。

我们先来看看腾讯的三大热修复方案是如何解决这个问题的：

- QQ 空间的处理方式，是在每个类中插入一个来自其他 dex 的 `hack.class`，由此让所有类里面都无法满足 pre-verified 条件。
- Tinker 的方式，是合成全量的 dex 文件，这样所有 class 的都在全量 dex 中解决，从而消除 class 重复而带来的冲突。
- QFix 的方式，是取得虚拟机中的某些底层函数，提前 resolve 所有补丁类。以此绕过 Pre-verify 检查。

以上的三种方案里面，QQ 空间方案会侵入打包流程，并且为了 `hack` 添加一些臃肿的代码，实现起来很不优雅。而我们一开始采用的 QFix 的方案，需要获取底层虚拟机的函数，不够稳定可靠。并且，和空间方案一样，有个比较大的问题是无法新增 `public` 函数，具体原因后续还将有文章进行详解。

现在看来比较好的方式，就是像 Tinker 那样全量合成完整新 dex。他们的合成方案，是从 dex 的方法和指令维度进行全量合成，虽然可以很大地节省空间，但由于

对 dex 内容的比较粒度过细，实现较为复杂，性能消耗比较严重。实际上，dex 的大小占整个 apk 的比例是比较低的，而占空间大的主要还是 apk 中的资源文件。因此，Tinker 方案的时空代价转换的性价比不高。

其实，dex 比较的最佳粒度，应该是在类的维度。它既不像方法和指令维度那样的细微，也不像 bsbiff 比较那般的粗糙。在类的维度，可以达到时间和空间平衡的最佳效果。

2.5.2 一种新的全量 Dex 方案

一般来说，合成完整 dex，思路就是把原来的 dex 和 patch 里的 dex 重新合并成一个。

然而我们的思路是反过来的。

我们可以这样考虑，既然补丁中已经有变动的类了，那只要在原先基线包里的 dex 里面，去掉补丁中也有的 class。这样，补丁 + 去除了补丁类的基线包，不就等于了新 app 中的所有类了吗？

参照 Android 原生 multi-dex 的实现再来看这个方案，会很好理解。multi-dex 是把一个 apk 里用到的所有类拆分到 `classes.dex`、`classes2.dex`、`classes3.dex`、... 之中，而每个 dex 都只包含了部分的类的定义，但单个 dex 也是可以加载的，因为只要把所有 dex 都 load 进去，本 dex 中不存在的类就可以在运行期间在其他的 dex 中找到。

因此同理，在基线包 dex 里面在去掉了补丁中 class 后，原先需要发生变更的旧的 class 就被消除了，基线包 dex 里就只包含不变的 class。而这些不变的 class 要用到补丁中的新 class 时会自动地找到补丁 dex，补丁包中的新 class 在需要用到不变的 class 时也会找到基线包 dex 的 class。这样的话，基线包里面不使用补丁类的 class 仍旧可以按原来的逻辑做 odex，最大地保证了 dexopt 的效果。

这么一来，我们不再需要像传统合成的思路那样判断类的增加和修改情况，而且也不需要处理合成时方法数超过的情况，对于 dex 的结构也不用进行破坏性重构。

现在，合成完整 dex 的问题就简化为了一——如何在基线包 dex 里面去掉补丁包中包含的所有类。接下来我们看一下在 dex 中去除指定类的具体实现。

首先，来看 dex 文件中 header 的结构：

```
/*
 * Direct-mapped "header_item" struct.
 */
struct DexHeader {
    uint8_t magic[8];           /* includes version number */
    uint32_t checksum;          /* adler32 checksum */
    uint8_t signature[kSHA1DigestLen]; /* SHA-1 hash */
    uint32_t fileSize;          /* length of entire file */
    uint32_t headerSize;        /* offset to start of next section */
    uint32_t endianTag;
    uint32_t linkSize;
    uint32_t linkOff;
    uint32_t mapOff;
    uint32_t stringIdsSize;
    uint32_t stringIdsOff;
    uint32_t typeIdsSize;
    uint32_t typeIdsOff;
    uint32_t protoIdsSize;
    uint32_t protoIdsOff;
    uint32_t fieldIdsSize;
    uint32_t fieldIdsOff;
    uint32_t methodIdsSize;
    uint32_t methodIdsOff;
    uint32_t classDefsSize;
    uint32_t classDefsOff;
    uint32_t dataSize;
    uint32_t dataOff;
};
```

由 dex header 就可以取得 dex 的各个重要属性，这些属性在文件中的分布如下所示：

Name	Format	Description
header	header_item	the header
string_ids	string_id_item[]	string identifiers list. These are identifiers for all the strings used by this file, either for internal naming (e.g., type descriptors) or as constant objects referred to by code. This list must be sorted by string contents, using UTF-16 code point values (not in a locale-sensitive manner), and it must not contain any duplicate entries.
type_ids	type_id_item[]	type identifiers list. These are identifiers for all types (classes, arrays, or primitive types) referred to by this file, whether defined in the file or not. This list must be sorted by string_id index, and it must not contain any duplicate entries.
proto_ids	proto_id_item[]	method prototype identifiers list. These are identifiers for all prototypes referred to by this file. This list must be sorted in return-type (by type_id index) major order, and then by argument list (lexicographic ordering, individual arguments ordered by type_id index). The list must not contain any duplicate entries.
field_ids	field_id_item[]	field identifiers list. These are identifiers for all fields referred to by this file, whether defined in the file or not. This list must be sorted, where the defining type (by type_id index) is the major order, field name (by string_id index) is the intermediate order, and type (by type_id index) is the minor order. The list must not contain any duplicate entries.
method_ids	method_id_item[]	method identifiers list. These are identifiers for all methods referred to by this file, whether defined in the file or not. This list must be sorted, where the defining type (by type_id index) is the major order, method name (by string_id index) is the intermediate order, and method prototype (by proto_id index) is the minor order. The list must not contain any duplicate entries.
class_defs	class_def_item[]	class definitions list. The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. Furthermore, it is invalid for a definition for the same-named class to appear more than once in the list.
call_site_ids	call_site_id_item[]	call site identifiers list. These are identifiers for all call sites referred to by this file, whether defined in the file or not. This list must be sorted in ascending order of call_site_off. This list must not contain any duplicate entries.
method_handles	method_handle_item[]	method handles list. A list of all method handles referred to by this file, whether defined in the file or not. This list is not sorted and may contain duplicates which will logically correspond to different method handle instances.
data	ubyte[]	data area, containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment.
link_data	ubyte[]	data used in statically linked files. The format of the data in this section is left unspecified by this document. This section is empty in unlinked files, and runtime implementations may use it as they see fit.

这里我们是打算去除 dex 里的 Class，因此我们最关心的自然是这里面的 class_defs。

需要注意的是，我们并不是要把某个 Class 的所有信息都从 dex 移除，因为如果这么做，可能会导致 dex 的各个部分都发生变化，从而需要大量调整 offset，这样就变得就费时费力了。我们要做的，仅仅是让在解析这个 dex 的时候找不到这个 Class 的定义就行了。**因此，只需要移除定义的入口，对于 Class 的具体内容不进行删除，这样可以最大可能地减少 offset 的修改。**

我们来看虚拟机在 dexopt 的时候是如何找到某个 dex 的所有类定义的。

```
@android-4.4.4_r2/dalvik/vm/analysis/DexPrepare.cpp

/*
 * Verify and/or optimize all classes that were successfully loaded from
 * this DEX file.
 */
static void verifyAndOptimizeClasses(DexFile* pDexFile, bool doVerify,
                                     bool doOpt)
{
    u4 count = pDexFile->pHeader->classDefsSize;
    u4 idx;

    for (idx = 0; idx < count; idx++) {
        const DexClassDef* pClassDef;
        const char* classDescriptor;
        ClassObject* clazz;

        pClassDef = dexGetClassDef(pDexFile, idx);
        classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef->classIdx);

        /* all classes are loaded into the bootstrap class loader */
        clazz = dvmLookupClass(classDescriptor, NULL, false);
        if (clazz != NULL) {
            verifyAndOptimizeClass(pDexFile, clazz, pClassDef, doVerify,
                                   doOpt);
        } else {
            // TODO: log when in verbose mode
            ALOGV("DexOpt: not optimizing unavailable class '%s',
                  classDescriptor);
        }
    }
}
```

正是 `dexGetClassDef` 函数返回了类的定义。

```
@android-4.4.4_r2/dalvik/libdex/DexFile.h

/* return the DexClassDef with the specified index */
DEX_INLINE const DexClassDef* dexGetClassDef(const DexFile* pDexFile, u4 idx)
{
    assert(idx < pDexFile->pHeader->classDefsSize);
    return &pDexFile->pClassDefs[idx];
}
```

而这里 `pClassDefs` 是怎么来的呢？

```
/*
 * Set up the basic raw data pointers of a DexFile. This function isn't
 * meant for general use.
 */
void dexFileSetupBasicPointers(DexFile* pDexFile, const u1* data) {
    DexHeader *pHeader = (DexHeader*) data;

    ...
    pDexFile->pClassDefs = (const DexClassDef*) (data + pHeader-
>classDefsOff);
    ...
}
```

由此可以看出，一个类的所有 `DexClassDef`，也就是类定义，是从 `pHeader->classDefsOff` 偏移处开始，一个接一个地线性排列着的，一个 dex 里面一共有 `pHeader->classDefsSize` 个类定义。

由此，我们就可以直接找到 `pHeader->classDefsOff` 偏移处，一个个地遍历所有的 `DexClassDef`，如果发现这个 `DexClassDef` 的类名包含在我们的补丁中，就把它移除，实现效果如下：

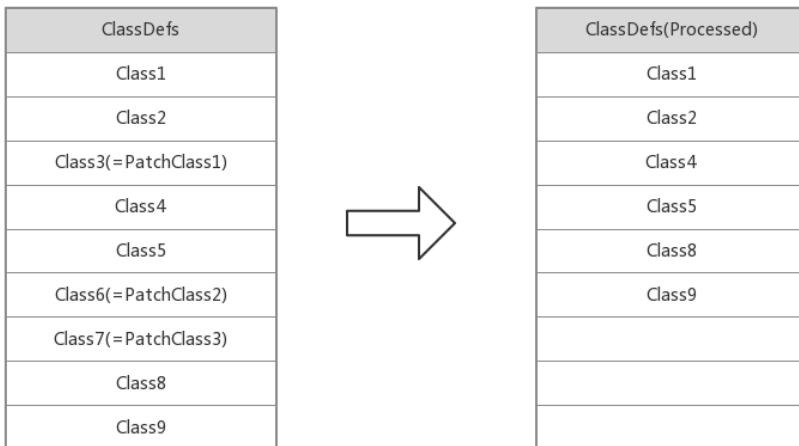


图 2-10 Dex 中 ClassDef 的移除

接着，只要修改 `pHeader->classDefssiz`，把 dex 中类的数目改为去除补丁中类之后的数目即可。

我们只是去除了类的定义，而对于类的方法实体以及其他 dex 信息不做移除，虽然这样会把这个被移除类的无用信息残留在 dex 文件中，但这些信息占不了太多空间，并且对 dex 的处理速度是提升很大的，这种移除类操作的方式就变得十分轻快。

2.5.3 对于 Application 的处理

由此，我们实现了完整的 dex 合成。但仍然有个问题，这个问题所有完整 dex 替换方案都会遇到，那就是对于 Application 的处理。

众所周知，Application 是整个 app 的入口，因此，在进入到替换的完整 dex 之前，一定会通过 Application 的代码，因此，Application 必然是加载在原来的老 dex 里面的。只有在补丁加载后使用的类，会在新的完整 dex 里面找到。

因此，在加载补丁后，如果 Application 类使用其他在新 dex 里的类，由于不在同一个 dex 里，如果 Application 被打上了 pre-verified 标志，这时就会抛出异常：

```
FATAL EXCEPTION: main
Process: com.taobao.worktest, PID: 2481
java.lang.IllegalAccessError: Class ref in pre-verified class resolved to
unexpected implementation
    at com.taobao.test.MyApplication.test(MyApplication.java:59)
    at com.taobao.test.MyApplication.onCreate(MyApplication.java:39)
    at android.app.Instrumentation.callApplicationOnCreate(Instrumentation.
java:1007)
    at android.app.ActivityThread.handleBindApplication(ActivityThread.
java:4328)
    at android.app.ActivityThread.access$1500(ActivityThread.java:135)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1256)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:136)
    at android.app.ActivityThread.main(ActivityThread.java:5001)
    at java.lang.reflect.Method.invokeNative(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:515)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:785)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:601)
    at dalvik.system.NativeStart.main(Native Method)
```

对此，我们的解决办法很简单，既然被设上了 pre-verified 标志，那么，清除掉它就是了。

类的标志，位于 `ClassObject` 的 `accessFlags` 成员。

```
struct ClassObject : Object {
    /* leave space for instance data; we could access fields directly if we
       freeze the definition of java/lang/Class */
    u4           instanceData[CLASS_FIELD_SLOTS];

    /* UTF-8 descriptor for the class; from constant pool, or on heap
       if generated ("[C") */
    const char*   descriptor;
    char*        descriptorAlloc;

    /* access flags; low 16 bits are defined by VM spec */
    u4           accessFlags;

    /* VM-unique class serial number, nonzero, set very early */
    u4           serialNumber;

    /* DexFile from which we came; needed to resolve constant pool entries */
    /* (will be NULL for VM-generated, e.g. arrays and primitive classes) */
    DvmDex*      pDvmDex;
```

```

/* state of class initialization */
ClassStatus     status;

...
}

```

而 pre-verified 标志的定义是

```
CLASS_ISPREVERIFIED      = (1<<16), // class has been pre-verified
```

因此，我们只需要在 jni 层清除掉它即可

```
clazzObj->accessFlags &= ~CLASS_ISPREVERIFIED;
```

这样，在 `dvmResolveClass` 找到了新 dex 里的类后，由于 `CLASS_ISPREVERIFIED` 标志被清空，就不会判断所在 dex 是否相同，从而成功避免抛出异常。

```

ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
    bool fromUnverifiedConstant) {
    ...
    // 条件不满足，不进行 check
    if (!fromUnverifiedConstant &&
        IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
    {
        ClassObject* resClassCheck = resClass;
        if (dvmIsArrayClass(resClassCheck))
            resClassCheck = resClassCheck->elementClass;
        // 判断是否在同一 dex 中
        if (referrer->pDvmDex != resClassCheck->pDvmDex &&
            resClassCheck->classLoader != NULL)
        {
            ALOGW("Class resolved by unexpected DEX:
                  \"%s(%p):%p ref [%s] %s(%p):%p\",
                  referrer->descriptor, referrer->classLoader,
                  referrer->pDvmDex,
                  resClass->descriptor, resClassCheck->descriptor,
                  resClassCheck->classLoader, resClassCheck->pDvmDex);
            ALOGW("(%s had used a different %s during pre-verification)",
                  referrer->descriptor, resClass->descriptor);
            dvmThrowIllegalAccessError(
                "Class ref in pre-verified class resolved to unexpected "
                "implementation");
            return NULL;
        }
    }

```

```

    }
...
}
```

接下来，我们来对比一下目前市面上其他完整 dex 方案是怎么做的。

Tinker 的方案，是在 `AndroidManifest.xml` 声明中就要求开发者将自己的 Application 直接换成 `TinkerApplication`。而对于真正 app 的 Application，要在初始化 `TinkerApplication` 时作为参数传入。这样 `TinkerApplication` 会接管这个传入的 Application，在生命周期回调时通过反射的方式调用实际 Application 的相关回调逻辑。这么做确实很好地将入口 Application 和用户代码隔离开来了，不过需要改造原先存在的 Application，如果对 Application 有更多扩展，接入成本也是比较高的。

Amigo 的方案，是在编译过程中，用 Amigo 自定义的 gradle 插件将 app 的 Application 替换成了 Amigo 自己的另一个 Application，并且将原来的 Application 的 name 保存起来，该修复的都修复完了的时候再调用之前保存的的 Application 的 `attach(context)`，然后将它设回到 `loadedApk` 中，最后调用它的 `onCreate()`，执行原有 Application 中的逻辑。这种方式只是开发者的代码层面无感知，但其实是在编译期间偷偷帮用户做了替换，有点掩耳盗铃的意味，并且这种对系统做反射替换本身也是有一定风险的。

相比之下，我们的 Application 处理方案既没有侵入编译过程，也不需要进行反射替换，所有的兼容操作都在运行期间都自动做好。接入过程极其顺滑。

2.5.4 dvmOptResolveClass 问题与对策

然而我们这种清除标志的方案并非一帆风顺，开发过程中我们发现，如果这个入口 Application 是没有 `pre-verified` 的，反而有更大的问题。

这个问题是，Dalvik 虚拟机如果发现某个类没有 `pre-verified`，就会在初始化这个类时做 Verify 操作，这将扫描这个类的所有代码，在扫描过程中对这个类代码里使

用到的类都要进行 `dvmOptResolveClass` 操作。

而这个 `dvmOptResolveClass` 正是罪魁祸首，它会在 Resolve 的时候对使用到的类进行初始化，而这个逻辑是发生在 Application 类初始化的时候。此时补丁还没进行加载，所以就会提前加载到原始 dex 中的类。接下来当补丁加载完毕后，这些已经加载的类如果用到了新 dex 中的类，并且又是 pre-verified 时就会报错。

这里最大的问题在于，我们无法把补丁加载提前到 `dvmOptResolveClass` 之前，因为在一个 app 的生命周期里，没有可能到达比入口 Application 初始化更早的时期了。

而这个问题常见于多 dex 情形，当存在多 dex 时，无法保证 Application 的用到的类和它处于同个 dex 中。如果只有一个 dex，一般就不会有这个问题。

多 dex 情况下要想解决这个问题，有两种办法：

- 第一种办法，让 Application 用到的所有非系统类都和 Application 位于同一个 dex 里，这样就可以保证 pre-verified 标志被打上，避免进入 `dvmOptResolveClass`，而在补丁加载完之后，我们再清除 pre-verified 标志，使得接下来使用其他类也不会报错。
- 第二种办法，把 Application 里面除了热修复框架代码以外的其他代码都剥离开，单独提出放到一个其他类里面，这样使得 Application 不会直接用到过多非系统类，这样，保证这个单独拿出来的类和 Application 处于同一个 dex 的几率还是比较大的。如果想要更保险，Application 可以采用反射方式访问这个单独类，这样就彻底把 Application 和其他类隔绝开了。

第一种方法实现较为简单，因为 Android 官方 multi-dex 机制会自动将 Application 用到的类都打包到主 dex 中，因此只要把热修复初始化放在 `attachBaseContext` 的最前面，大多都没问题。而第二种方法稍加繁琐，是在代码架构层面进行重新设计，不过可以一劳永逸地解决问题。

2.6 本章小结

本章主要讲解了代码热修复的底层替换实现和冷启动类加载实现原理，并详细说明了在我们开发中遇到的疑难问题及解决方式。Sophix 采用了两种方案相结合的方式，实现了按需选择、完全兼顾的效果。

第3章 资源热修复技术

3.1 普遍的实现方式

Android 资源的热修复，就是在 app 不重新安装的情况下，利用下发的补丁包直接更新本 app 中的资源。

目前市面上的很多资源热修复方案基本上都是参考了 Instant Run 的实现。

首先，我们简单来看一下 Instant Run 是怎么做到资源热修复的。

Instant Run 资源热修复的核心代码就是这个 monkeyPatchExistingResources 方法：

```
@com/android/tools/fd/runtime/MonkeyPatcher.java

public static void monkeyPatchExistingResources(@Nullable Context context,
                                                 @Nullable String
externalResourceFile,
                                                 @Nullable
Collection<Activity> activities) {

    if (externalResourceFile == null) {
        return;
    }

    try {
        // % Part 1. 创建一个新的 AssetManager，并通过反射调用 addAssetPath 添加 /
sdcard 上的新资源包。
        // 这样就构造出了一个带新资源的 AssetManager
        // Create a new AssetManager instance and point it to the resources
installed under
        // /sdcard
        AssetManager newAssetManager = AssetManager.class.getConstructor().newInstance();
        Method mAddAssetPath = AssetManager.class.
getDeclaredMethod("addAssetPath", String.class);
```

```

        mAddAssetPath.setAccessible(true);
        if (((Integer) mAddAssetPath.invoke(newAssetManager,
externalResourceFile)) == 0) {
            throw new IllegalStateException("Could not create new
AssetManager");
        }

        // Kitkat needs this method call, Lollipop doesn't. However, it
doesn't seem to cause any harm
        // in L, so we do it unconditionally.
        Method mEnsureStringBlocks = AssetManager.class.
getDeclaredMethod("ensureStringBlocks");
        mEnsureStringBlocks.setAccessible(true);
        mEnsureStringBlocks.invoke(newAssetManager);

        // %% Part 2. 反射得到 Activity 中 AssetManager 的引用处，全部换成刚才新构建的
newAssetManager
        if (activities != null) {
            for (Activity activity : activities) {
                Resources resources = activity.getResources();

                try {
                    Field mAssets = Resources.class.
getDeclaredField("mAssets");
                    mAssets.setAccessible(true);
                    mAssets.set(resources, newAssetManager);
                } catch (Throwable ignore) {
                    Field mResourcesImpl = Resources.class.
getDeclaredField("mResourcesImpl");
                    mResourcesImpl.setAccessible(true);
                    Object resourceImpl = mResourcesImpl.get(resources);
                    Field implAssets = resourceImpl.getClass().
getDeclaredField("mAssets");
                    implAssets.setAccessible(true);
                    implAssets.set(resourceImpl, newAssetManager);
                }
                ...
            }

            pruneResourceCaches(resources);
        }
    }

    // %% Part 3. 得到 Resources 的弱引用集合，把他们的 AssetManager 成员替换成
newAssetManager
    // Iterate over all known Resources objects
    Collection<WeakReference<Resources>> references;
    if (SDK_INT >= KITKAT) {
        // Find the singleton instance of ResourcesManager
        Class<?> resourcesManagerClass = Class.forName("android.app.

```

```

ResourcesManager");
    Method mGetInstance = resourcesManagerClass.
getDeclaredMethod("getInstance");
    mGetInstance.setAccessible(true);
    Object resourcesManager = mGetInstance.invoke(null);
    try {
        Field fMActiveResources = resourcesManagerClass.
getDeclaredField("mActiveResources");
        fMActiveResources.setAccessible(true);
        @SuppressWarnings("unchecked")
        ArrayMap<?, WeakReference<Resources>> arrayMap =
            (ArrayMap<?, WeakReference<Resources>>)
fMActiveResources.get(resourcesManager);
        references = arrayMap.values();
    } catch (NoSuchFieldException ignore) {
        Field mResourceReferences = resourcesManagerClass.
getDeclaredField("mResourceReferences");
        mResourceReferences.setAccessible(true);
        //noinspection unchecked
        references = (Collection<WeakReference<Resources>>)
mResourceReferences.get(resourcesManager);
    }
} else {
    Class<?> activityThread = Class.forName("android.app.
ActivityThread");
    Field fMActiveResources = activityThread.
getDeclaredField("mActiveResources");
    fMActiveResources.setAccessible(true);
    Object thread = getActivityThread(context, activityThread);
    @SuppressWarnings("unchecked")
    HashMap<?, WeakReference<Resources>> map =
        (HashMap<?, WeakReference<Resources>>) fMActiveResources.
get(thread);
    references = map.values();
}
for (WeakReference<Resources> wr : references) {
    Resources resources = wr.get();
    if (resources != null) {
        // Set the AssetManager of the Resources instance to our
brand new one
        try {
            Field mAssets = Resources.class.
getDeclaredField("mAssets");
            mAssets.setAccessible(true);
            mAssets.set(resources, newAssetManager);
        } catch (Throwable ignore) {
            Field mResourcesImpl = Resources.class.
getDeclaredField("mResourcesImpl");
            mResourcesImpl.setAccessible(true);
        }
    }
}

```

```
        Object resourceImpl = mResourcesImpl.get(resources);
        Field implAssets = resourceImpl.getClass() .
getDeclaredField("mAssets");
        implAssets.setAccessible(true);
        implAssets.set(resourceImpl, newAssetManager);
    }

    resources.updateConfiguration(resources.getConfiguration(),
resources.getDisplayMetrics());
}
}
} catch (Throwable e) {
    throw new IllegalStateException(e);
}
}
```

简要说来，Instant Run 中的资源热修复分为两步，

1. 构造一个新的 AssetManager，并通过反射调用 addAssetPath，把这个完整的新资源包加入到 AssetManager 中。这样就得到了一个含有所有新资源的 AssetManager。
 2. 找到所有之前引用到原有 AssetManager 的地方，通过反射，把引用处替换为 AssetManager。

其实仔细看可以发现，大量代码都是在处理兼容性问题和找到**所有** AssetManager 的引用处。真正的实现逻辑其实很简单。

这其中的重点，自然是 addAssetPath 这个函数。现在我们来看一下它的底层实现逻辑。

以 Android 6.0 为例，`addAssetPath` 最终调用到了 native 方法。

```
@frameworks/base/core/java/android/content/res/AssetManager.java

/**
 * Add an additional set of assets to the asset manager. This can be
 * either a directory or ZIP file. Not for use by applications.

```

```

public final int addAssetPath(String path) {
    synchronized (this) {
        int res = addAssetPathNative(path);
        makeStringBlocks(mStringBlocks);
        return res;
    }
}

...
private native final int addAssetPathNative(String path);

```

Java 层的 AssetManager 只是个包装，真正关于资源处理的所有逻辑，其实都位于 native 层由 C++ 实现的 AssetManager。

执行 addAssetPath 就是解析这个格式，然后构造出底层数据结构的过程。整个解析资源的调用链是：

- public final int addAssetPath(String path)
- android_content_AssetManager_addAssetPath
- AssetManager::addAssetPath
- AssetManager::appendPathToResTable
- ResTable::add
- ResTable::addInternal
- ResTable::parsePackage

解析的细节比较繁琐，就不细细说明了，有兴趣的可以一层层追下去。

大致过程就是，通过传入的资源包路径，先得到其中的 resources.arsc，然后解析它的格式，存放在底层的 AssetManager 的 mResources 成员中。

```

@frameworks/base/include/androidfw/AssetManager.h
class AssetManager : public AAssetManager {

...
mutable ResTable* mResources;
...

```

AssetManager 的 mResources 成员是一个 ResTable 结构体：

```
class ResTable
{
    mutable Mutex          mLock;
    // 互斥锁，用于多进程间互斥操作。

    status_t               mError;

    ResTable_config         mParams;

    // Array of all resource tables.
    Vector<Header*>       mHeaders;
    // 表示所有 resources.arsc 原始数据，这就等同于所有通过 addAssetPath 加载进来的路径
    // 的资源 id 信息。

    // Array of packages in all resource tables.
    Vector<PackageGroup*>   mPackageGroups;
    // 资源包的实体，包含所有加载进来的 package id 所对应的资源。

    // Mapping from resource package IDs to indices into the internal
    // package array.
    uint8_t                 mPackageMap[256];
    // 索引表，表示 0~255 的 package id，每个元组分别存放 该 id 所属 PackageGroup 在
    // mPackageGroups 中的 index

    uint8_t                 mNextPackageId;
};
```

一个 Android 进程只包含一个 ResTable，ResTable 的成员变量 mPackageGroups 就是所有解析过的资源包的集合。任何一个资源包中都含有 resources.arsc，它记录了所有资源的 id 分配情况以及资源中的所有字符串。这些信息是以二进制方式存储的。底层的 AssetManager 做的事就是解析这个文件，然后把相关信息存储到 mPackageGroups 里面。

3.2 资源文件的格式

整个 resources.arsc 文件，实际上是由一个个 ResChunk (以下简称 chunk) 拼接起来的。从文件头开始，每个 chunk 的头部都是一个 ResChunk_header 结

构，它指示了这个 chunk 的大小和数据类型。

```
/*
 * Header that appears at the front of every data chunk in a resource.
 */
struct ResChunk_header
{
    // Type identifier for this chunk. The meaning of this value depends
    // on the containing chunk.
    uint16_t type;

    // Size of the chunk header (in bytes). Adding this value to
    // the address of the chunk allows you to find its associated data
    // (if any).
    uint16_t headerSize;

    // Total size of this chunk (in bytes). This is the chunkSize plus
    // the size of any data associated with the chunk. Adding this value
    // to the chunk allows you to completely skip its contents (including
    // any child chunks). If this value is the same as chunkSize, there is
    // no data associated with the chunk.
    uint32_t size;
};
```

通过 ResChunk_header 中的 type 成员，可以知道这个 chunk 是什么类型，从而就可以知道应该如何解析这个 chunk。

解析完一个 chunk 后，从这个 chunk + size 的位置开始，就可以得到下一个 chunk 起始位置，这样就可以依次读取完整个文件的数据内容。

一般来说，一个 resources.arsc 里面包含若干个 package，不过默认情况下，由打包工具 aapt 打出来的包只有一个 package。这个 package 里包含了 app 中的所有资源信息。

资源信息主要是指每个资源的名称以及它对应的编号。我们知道，Android 中的每个资源，都有它唯一的编号。

编号是一个 32 位数字，用十六进制来表示就是 0xPPTTEEEE。PP 为 package id，TT 为 type id，EEEE 为 entry id。

它们代表什么？在 resources.arsc 里是以怎样的方式记录的呢？

- 对于 package id，每个 package 对应的是类型为 RES_TABLE_PACKAGE_TYPE 的 ResTable_package 结构体，ResTable_package 结构体的 id 成员变量就表示它的 package id。
- 对于 type id，每个 type 对应的是类型为 RES_TABLE_TYPE_SPEC_TYPE 的 ResTable_typeSpec 结构体。它的 id 成员变量就是 type id。但是，该 type id 具体对应什么类型，是需要到 package chunk 里的 Type String Pool 中去解析得到的。比如 Type String Pool 中依次有 attr、drawable、mipmap、layout 字符串。就表示 attr 类型的 type id 为 1，drawable 类型的 type id 为 2，mipmap 类型的 type id 为 3，layout 类型的 type id 为 4。所以，每个 type id 对应了 Type String Pool 里的字符顺序所指定的类型。
- 对于 entry id，每个 entry 表示一个资源项，资源项是按照排列的先后顺序自动被标机编号的。也就是说，一个 type 里按位置出现的第一个资源项，其 entry id 为 0x0000，第二个为 0x0001，以此类推。因此我们是无法直接指定 entry id 的，只能根据排布顺序决定。资源项之间是紧密排布的，没有空隙，但是可以指定资源项为 ResTable_type::NO_ENTRY 来填入一个空资源。

举个例子，我们随便找个带资源的 apk，用 aapt 解析一下，看到其中的一行是：

```
$ aapt d resources app-debug.apk
...
spec resource 0x7f040019 com.taobao.patch.demo:layout/activity_main:
flags=0x00000000
...
```

这就表示，activity_main.xml 这个资源的编号是 0x7f040019。它的 package id 是 0x7f，资源类型的 id 为 0x04，Type String Pool 里的第四个字符串正是 layout 类型，而 0x04 类型的第 0x0019 个资源项就是 activity_main 这个资源。

3.3 运行时资源的解析

默认由 Android SDK 编出来的 apk，是由 aapt 工具进行打包的，其资源包的 package id 就是 0x7f。

系统的资源包，也就是 framework-res.jar，package id 为 0x01。

在走到 app 的第一行代码之前，系统就已经帮我们构造好一个已经添加了安装包资源的 AssetManager 了。

```
@frameworks/base/core/java/android/app/ResourcesManager.java

    Resources getTopLevelResources(String resDir, String[] splitResDirs,
        String[] overlayDirs, String[] libDirs, int displayId,
        Configuration overrideConfiguration, CompatibilityInfo
compatInfo) {
    ...
    ...

    AssetManager assets = new AssetManager();
    // resDir 就是安装包 apk
    if (resDir != null) {
        if (assets.addAssetPath(resDir) == 0) {
            return null;
        }
    }
    ...
}
```

因此，这个 AssetManager 里就已经包含了系统资源包以及 app 的安装包，就是 package id 为 0x01 的 framework-res.jar 中的资源和 package id 为 0x7f 的 app 安装包资源。

如果此时直接在原有 AssetManager 上继续 addAssetPath 的完整补丁包的话，由于补丁包里面的 package id 也是 0x7f，就会使得同一个 package id 的包被加载两次。这会有怎样的问题呢？

在 Android L 之后，这是没问题的，他会默默地把后来的包添加到之前的包的同一个 PackageGroup 下面。

而在解析的时候，会与之前的包比较同一个 type id 所对应的类型，如果该类型下的资源项数目和之前添加过的不一致，会打出一条 warning log，但是仍旧加入到该类型的 TypeList 中。

```

status_t ResTable::parsePackage(const ResTable_package* const pkg,
                               const Header* const header)
{
    ...
    TypeList& typeList = group->types.editItemAt(typeIndex);
    if (!typeList.isEmpty()) {
        const Type* existingType = typeList[0];
        if (existingType->entryCount != newEntryCount &&
            idmapIndex < 0) {
            ALOGW("ResTable_typeSpec entry count inconsistent:
given %d, previously %d",
                  (int) newEntryCount, (int) existingType-
>entryCount);
            // We should normally abort here, but some legacy
apps declare
            // resources in the 'android' package (old bug in
AAPT).
        }
    }

    Type* t = new Type(header, package, newEntryCount);
    t->typeSpec = typeSpec;
    t->typeSpecFlags = (const uint32_t*)(
        ((const uint8_t*)typeSpec) + dtohs(typeSpec->header.
headerSize));
    if (idmapIndex >= 0) {
        t->idmapEntries = idmapEntries[idmapIndex];
    }
    typeList.add(t);
}
...

```

但是在 get 这个资源的时候呢？

```

status_t ResTable::getEntry(
    const PackageGroup* packageGroup, int typeIndex, int entryIndex,
    const ResTable_config* config,
    Entry* outEntry) const
{
    const TypeList& typeList = packageGroup->types[typeIndex];
    ...
    // %% 从第一个 type 开始遍历，也就是说会先取得安装包的资源，然后才是补丁包。
    // Iterate over the Types of each package.
}

```

```

const size_t typeCount = typeList.size();
for (size_t i = 0; i < typeCount; i++) {
    const Type* const typeSpec = typeList[i];

    int realEntryIndex = entryIndex;
    int realTypeIndex = typeIndex;
    bool currentTypeIsOverlay = false;

    if (static_cast<size_t>(realEntryIndex) >= typeSpec->entryCount) {
        ALOGW("For resource 0x%08x, entry index(%d) is beyond type
entryCount(%d)",
              Res_MAKEID(packageGroup->id - 1, typeIndex, entryIndex),
              entryIndex, static_cast<int>(typeSpec->entryCount));
        // We should normally abort here, but some legacy apps declare
        // resources in the 'android' package (old bug in AAPT).
        continue;
    }

    const size_t numConfigs = typeSpec->configs.size();
    for (size_t c = 0; c < numConfigs; c++) {
        ...
        if (bestType != NULL) {
            // Check if this one is less specific than the last found.
        If so,
            // we will skip it. We check starting with things we most
care
            // about to those we least care about.
            if (!thisConfig.isBetterThan(bestConfig, config)) {
                if (!currentTypeIsOverlay || thisConfig.
compare(bestConfig) != 0) {
                    continue;
                }
            }
        }
        bestType = thisType;
        bestOffset = thisOffset;
        bestConfig = thisConfig;
        bestPackage = typeSpec->package;
        actualTypeIndex = realTypeIndex;

        // If no config was specified, any type will do, so skip
        if (config == NULL) {
            break;
        }
    }
}
}

```

在获取某个 Type 的资源时，会从前往后遍历，也就是说先得到原有安装包里的资源，除非后面的资源的 config 比前面的更详细才会发生覆盖。而对于同一个 config 而言，补丁中的资源就永远无法生效了。所以在 Android L 以上的版本，在原有 AssetManager 上加入补丁包，是没有任何作用的，补丁中的资源无法生效。

而在 Android 4.4 及以下版本，addAssetPath 只是把补丁包的路径添加到了 mAssetPath 中，而真正解析的资源包的逻辑是在 app 第一次执行 AssetManager::getResTable 的时候。

```
@android-4.4.4_r2/frameworks/base/libs/androidfw/AssetManager.cpp

const ResTable* AssetManager::getResTable(bool required) const
{
    // %% mResources 已存在，直接返回，不再往下走。
    ResTable* rt = mResources;
    if (rt) {
        return rt;
    }

    // Iterate through all asset packages, collecting resources from each.

    AutoMutex _l(mLock);

    if (mResources != NULL) {
        return mResources;
    }

    if (required) {
        LOG_FATAL_IF(mAssetPaths.size() == 0, "No assets added to
AssetManager");
    }

    if (mCacheMode != CACHE_OFF && !mCacheValid)
        const_cast<AssetManager*>(this)->loadFileNameCacheLocked();

    const size_t N = mAssetPaths.size();
    for (size_t i=0; i<N; i++) {
        // ... %% 真正解析 package 的地方 ...
    }

    if (required && !rt) ALOGW("Unable to find resources file resources.
arsc");
    if (!rt) {
        mResources = rt = new ResTable();
    }
}
```

```

    }
    return rt;
}

```

而在执行到加载补丁代码的时候，getResTable 已经执行过了无数次了。这是因为就算我们之前没做过任何资源相关操作，Android framework 里的代码也会多次调用到那里。所以，以后即使是 addAssetPath，也只是添加到了 mAssetPath，并不会发生解析。所以补丁包里面的资源是完全不生效的！

所以，像 Instant Run 这种方案，一定需要一个全新的 AssetManager 时，然后再加入完整的新资源包，替换掉原有的 AssetManager。

3.4 另辟蹊径的资源修复方案

而一个好的资源热修复方案是怎样的呢？

首先，补丁包要足够小，像直接下发完整的补丁包肯定是不行的，很占用空间。

而像有些方案，是先进行 bsdiff，对资源包做差量，然后下发差量包，在运行时合成完整包再加载。这样确实减小了包的体积，但是却在运行时多了合成的操作，耗费了运行时间和内存。合成后的包也是完整的包，仍旧会占用磁盘空间。

而如果不采用类似 Instant Run 的方案，市面上许多实现，是自己修改 aapt，在打包时将补丁包资源进行重新编号。这样就会涉及到修改 Android SDK 工具包，即不利于集成也无法很好地对将来的 aapt 版本进行升级。

针对以上几个问题，一个好的资源热修复方案，既要保证补丁包足够小，不在运行时占用很多资源，又要不侵入打包流程。我们提出了一个目前市面上未曾实现的方案。

简单来说，我们构造了一个 package id 为 0x66 的资源包，这个包里只包含改变了的资源项，然后直接在原有 AssetManager 中 addAssetPath 这个包。然后，

就可以了。

真的这么简单？

没错！由于补丁包的 package id 为 0x66，不与目前已经加载的 0x7f 冲突，因此直接加入到已有的 AssetManager 中就可以直接使用了。补丁包里面的资源，只包含原有包里面**没有**而新的包里面**有的**新增资源，以及原有内容发生了改变的资源。

而资源的改变包含增加、减少、修改这三种情况，我们分别是如何处理的呢？

- 对于新增资源，直接加入补丁包，然后新代码里直接引用就可以了，没什么好说的。
- 对于减少资源，我们只要不使用它就行了，因此不用考虑这种情况，它也不影响补丁包。
- 对于修改资源，比如替换了一张图片之类的情况。我们把它视为新增资源，在打入补丁的时候，代码在引用处也会做相应修改，也就是直接把原来使用旧资源 id 的地方变为新 id。

用一张图来说明补丁包的情况，是这样的：

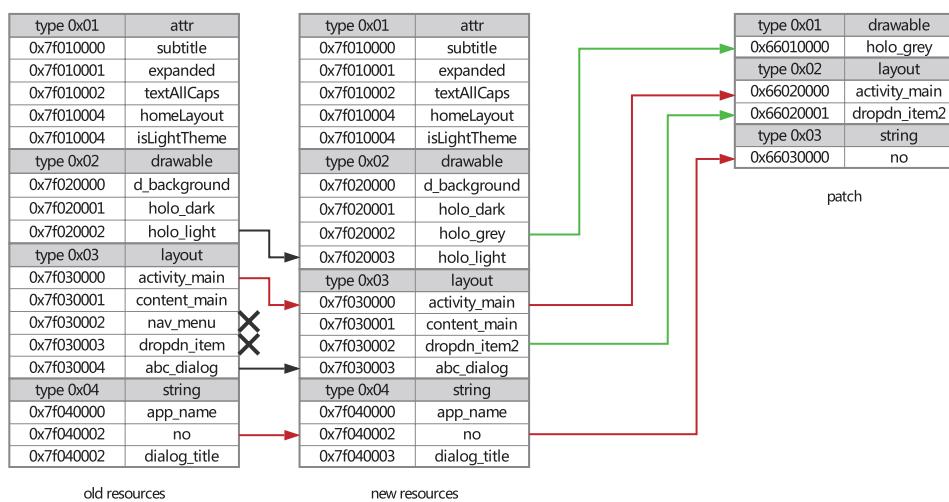


图 3-1

图中绿线表示新增资源。红线表示内容发生修改的资源。黑线表示内容没有变化，但是 id 发生改变的资源。× 表示删除了的资源。

3.4.1 新增的资源及其导致 id 偏移

可以看到，新的资源包与旧资源包相比，新增了 holo_grey 和 dropdn_item2 资源，新增的资源被加入到 patch 中。并分配了 0x66 开头的资源 id。

而新增的两个资源导致了在它们所属的 type 中跟在它们之后的资源 id 发生了位移。比如 holo_light，id 由 0x7f020002 变为 0x7f020003，而 abc_dialog 由 0x7f030004 变为 0x7f030003。新资源插入的位置是随机的，这与每次 aapt 打包时解析 xml 的顺序有关。发生位移的资源不会加入 patch，但是在 patch 的代码中会调整 id 的引用处。

比如说在代码里，我们是这么写的

```
imageView.setImageResource(R.drawable.holo_light);
```

这个 R.drawable.holo_light 是一个 int 值，它的值是 aapt 指定的，对于开发者透明，即使点进去，也会直接跳到对应 res/drawable/holo_light.png，无法查看。不过可以用反编译工具，看到它的真实值是 0x7f020002。所以这行代码其实等价于：

```
imageView.setImageResource(0x7f020002);
```

而当打出了一个新包后，对开发者而言，holo_light 的图片内容没变，代码引用处也没变。但是新包里面，同样是这句话，由于新资源的插入导致的 id 改变，对于 R.drawable.holo_light 的引用已经变成了：

```
imageView.setImageResource(0x7f020003);
```

但实际上这种情况并不属于资源改变，更不属于代码的改变，所以我们在对比新旧代码之前，会把新包里面的这行代码修正回原来的 id。

```
imageView.setImageResource(0x7f020002);
```

然后再进行后续代码的对比。这样后续代码对比时就不会检测到发生了改变。

3.4.2 内容发生改变的资源

而对于内容发生改变的资源（类型为 layout 的 activity_main，这可能是我们修改了 activity_main.xml 的文件内容。还有类型为 string 的 no，可能是我们修改了这个字符串的值），它们都会被加入到 patch 中，并重新编号为新 id。

而相应的代码，也会发生改变，比如，

```
setContentView(R.layout.activity_main);
```

实际上也就是

```
setContentView(0x7f030000);
```

在生成对比新旧代码之前，我们会把新包里面的这行代码变为

```
setContentView(0x66020000);
```

这样，在进行代码对比时，会使得这行代码所在函数被检测到发生了改变。于是相应的代码修复会在运行时发生，这样就引用到了正确的新内容资源。

3.4.3 删除了的资源

对于删除的资源，不会影响补丁包。

这很好理解，既然资源被删除了，就说明新的代码中也不会用到它，那资源放在那里没人用，就相当于不存在了。

3.4.4 对于 type 的影响

可以看到，由于 type0x01 的所有资源项都没有变化，所以整个 type0x01 资源都没有加入到 patch 中。这也使得后面的 type 的 id 都往前移了一位。因此 Type String Pool 中的字符串也要进行修正，这样才能使得 0x01 的 type 指向 drawable，而不是原来的 attr。

所以我们可以看到，所谓简单，指的是运行时应用 patch 变的简单了。

而真正复杂的地方在于构造 patch。我们需要把新旧两个资源包解开，分别解析其中的 resources.arsc 文件，对比新旧的不同，并将它们重新打成带有新 package id 的新资源包。这里补丁包指定的 package id 只要不是 0x7f 和 0x01 就行，可以是任意 0x7f 以下的数字，我们默认把它指定为 0x66。

构造这样的补丁资源包，需要对整个 resources.arsc 的结构十分了解，要对二进制形式的一个一个 chunk 进行解析分类，然后再把补丁信息一个一个重新组装成二进制的 chunk。这里面很多工作与 aapt 做的类似，实际上开发打包工具的时候也是参考了很多 aapt 和系统加载资源的代码。

3.5 更优雅地替换 AssetManager

对于 Android L 以后的版本，直接在原有 AssetManager 上应用 patch 就行了。并且由于用的是原来的 AssetManager，所以原先大量的反射修改替换操作就完全不需要了，大大提高了加载补丁的效率。

但之前提到过，在 Android KK 和以下版本，addAssetPath 是不会加载资源的，必须重新构造一个新的 AssetManager 并加入 patch，再换掉原来的。那么我们不就又要和 Instant Run 一样，做一大堆兼容版本和反射替换的工作了吗？

对于这种情况，我们也找到了更优雅的方式，不需要再如此地大费周章。

在 AssetManager 的源码里面，有一个有趣的东西。

```
@frameworks/base/core/java/android/content/res/AssetManager.java
public final class AssetManager {
    ...
    private native final void destroy();
    ...
}
```

明显，这个是用来销毁 AssetManager 并释放资源的函数，我们来看看它具体做了什么吧。

```
static void android_content_AssetManager_destroy(JNIEnv* env, jobject clazz)
{
    AssetManager* am = (AssetManager*)
        (env->GetIntField(clazz, gAssetManagerOffsets.mObject));
    ALOGV("Destroying AssetManager %p for Java object %p\n", am, clazz);
    if (am != NULL) {
        delete am;
        env->SetIntField(clazz, gAssetManagerOffsets.mObject, 0);
    }
}
```

可以看到，首先，它析构了 native 层的 AssetManager，然后把 java 层的 AssetManager 对 native 层的 AssetManager 的引用设为空。

```
AssetManager::~AssetManager(void)
{
    int count = android_atomic_dec(&gCount);
    //ALOGI("Destroying AssetManager in %p # %d\n", this, count);

    delete mConfig;
    delete mResources;

    // don't have a String class yet, so make sure we clean up
    delete[] mLocale;
    delete[] mVendor;
}
```

native 层的 AssetManager 析构函数会析构它的所有成员，这样就会释放之前加载了的资源。

而现在，java 层的 AssetManager 已经成为了空壳。我们就可以调用它的 init 方法，对它重新进行初始化了！

```
@frameworks/base/core/java/android/content/res/AssetManager.java
public final class AssetManager {
    ...
    private native final void init();
    ...
}
```

这同样是个 native 方法，

```
static void android_content_AssetManager_init(JNIEnv* env, jobject clazz)
{
    AssetManager* am = new AssetManager();
    if (am == NULL) {
        jniThrowException(env, "java/lang/OutOfMemoryError", "");
        return;
    }

    am->addDefaultAssets();

    ALOGV("Created AssetManager %p for Java object %p\n", am, clazz);
    env->SetIntField(clazz, gAssetManagerOffsets.mObject, (jint)am);
}
```

这样，在执行 init 的时候，会在 native 层创建一个没有添加过资源，并且 mResources 没有初始化过的 AssetManager。然后我们再对它进行 addAssetPath，之后由于 mResource 没有初始化过，就可以正常走到解析 mResources 的逻辑，加载所有此时 add 进去的资源了！

```
@android-4.4.4_r2/frameworks/base/libs/androidfw/AssetManager.cpp

const ResTable* AssetManager::getResTable(bool required) const
{
    ResTable* rt = mResources;
    // %% mResources 没有初始化过，为空，因此不会 return。
    if (rt) {
        return rt;
    }
    ...
}
```

```
// %% 这时就会走到这里，进行所有 add 进去的 path 的加载。
const size_t N = mAssetPaths.size();
for (size_t i=0; i<N; i++) {
    // ... 解析 package ...
}

...
return rt;
}
```

这个方案的实现代码如下：

```
...
Method initMeth = assetManagerMethod("init");
Method destroyMeth = assetManagerMethod("destroy");
Method addAssetPathMeth = assetManagerMethod("addAssetPath", String.
class);

// %% 析构 AssetManager
destroyMeth.invoke(am);

// %% 重新构造 AssetManager
initMeth.invoke(am);

// %% 置空 mStringBlocks
assetManagerField("mStringBlocks").set(am, null);

// %% 重新添加原有 AssetManager 中加载过的资源路径
for (String path : loadedPaths) {
    LogTool.d(TAG, "pexyResources" + path);
    addAssetPathMeth.invoke(am, path);
}

// %% 添加 patch 资源路径
addAssetPathMeth.invoke(am, patchPath);

// %% 重新对 mStringBlocks 赋值
assetManagerMethod("ensureStringBlocks").invoke(am);

}

private Method assetManagerMethod(String name, Class<?>...
parameterTypes) {
try {
    Method meth = Class.forName("android.content.res.AssetManager")
```

```
        .getDeclaredMethod(name, parameterTypes);
    meth.setAccessible(true);
    return meth;
} catch (Exception e) {
    LogTool.e(TAG, "assetManagerMethod", e);
    return null;
}
}

private Field assetManagerField(String name) {
try {
    Field field = mAssetManagerClass.getDeclaredField(name);
    field.setAccessible(true);
    return field;
} catch (Exception e) {
    LogTool.e(TAG, "assetManagerField", e);
    return null;
}
}
```

这里需要注意的地方是 `mStringBlocks`。它记录了之前加载过的所有资源包的 String Pool，因此很多时候访问字符串是通过它来找到的。如果不进行重新构造，在后面使用到它时就会导致崩溃。

由于我们是直接对原有的 `AssetManager` 进行析构和重构，所有原先对 `AssetManager` 对象的引用是没有发生改变的，这样，就不需要像 Instant Run 那样进行繁琐的修改了。

顺带一提，类似 Instant Run 的完整替换资源的方案，在替换 `AssetManager` 这一步，也可以采用我们这种方式进行替换，省时省力又省心。

3.6 本章小结

总结一下，相比于目前市面上的资源修复方式，我们提出的资源修复的优势在于：

- 不侵入打包，直接对比新旧资源即可产生补丁资源包。（对比修改 aapt 方式的实现）
- 不必下发完整包，补丁包中只包含有变动的资源。（对比 Instant Run、Amigo 等方式的实现）
- 不需要在运行时合成完整包。不占用运行时计算和内存资源。（对比 Tinker 的实现）

唯一有个需要注意的地方就是，因为对新的资源的引用是在新代码中，所有资源修复是需要代码修复的支持的。也因此所有资源修复方案必然是附带代码修复的。而之前提到过，本方案在进行代码修复前，会对资源引用处进行修正。而修正就是需要找到旧的资源 id，换成新的 id。查找旧 id 时是直接对 int 值进行替换，所以会找到 `0x7f??????` 这样的需要替换 id。但是，如果有开发者使用到了 `0x7f??????` 这样的数字，而它并非资源 id，可是却和需要替换的 id 数值相同，这就会导致这个数字被错误地替换。

但这种情况是极为罕见的，因为很少会有人用到这样特殊的数字，并且还需要碰巧这数字和资源 id 相等才行。即使出现，开发者也可以用拼接的方式绕过这类数字的产生。所以基本可以不用担心这种情况，只是需要注意它的存在。

第4章 SO 库热修复技术

4.1 SO 库加载原理

Java API 提供以下两个接口加载一个 so 库

- `System.loadLibrary(String libName)`: 传进去的参数: so 库名称, 表示的 so 库文件, 位于 apk 压缩文件中的 libs 目录, 最后复制到 apk 安装目录下。
- `System.load(String pathName)`: 传进去的参数: so 库在磁盘中的完整路径。加载一个自定义外部 so 库文件。

上述两种方式加载一个 so 库, 实际上最后都调用 `nativeLoad` 这个 native 方法去加载 so 库, 这个方法的参数 `fileName`: so 库在磁盘中的完整路径名。

代码 + 图文的方式简述 so 库加载原理, 下面的代码示例, `stringFromJNI -> Java_com_taobao_jni_MainActivity_stringFromJNI` 静态注册的 native 方法, `test -> test` 动态注册的 native 方法。

```
public class MainActivity extends Activity {
    static {
        System.loadLibrary("jnitest");
    }
    public static native String stringFromJNI();
    public static native void test();
}
// 静态注册 stringFromJNI 本地方法
extern "C" jstring Java_com_taobao_jni_MainActivity_stringFromJNI(JNIEnv*
*env, jclass clazz) {
    std::string hello = "jni stringFromJNI old....";
    return env->NewStringUTF(hello.c_str());
}
```

```
// 动态注册 test 方法
void test(JNIEnv *env, jclass clazz) {
    LOGD("jni test old....");
}
JNINativeMethod nativeMethods[] = {
    {"test", "()V", (void *) test}
};
#define JNIREG_CLASS "com/taobao/jni/MainActivity"
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
    LOGD("old JNI_OnLoad");
    ....
    jclass clz = env->FindClass(JNIREG_CLASS);
    if (env->RegisterNatives(clz, nativeMethods, sizeof(nativeMethods) /
sizeof(nativeMethods[0])) != JNI_OK) {
        return JNI_ERR;
    }
    return JNI_VERSION_1_4;
}
```

我们知道 JNI 编程中，动态注册的 native 方法必须实现 `JNI_OnLoad` 方法，同时实现一个 `JNINativeMethod[]` 数组，静态注册的 native 方法必须是 `Java+类完整路径 + 方法名` 的格式。

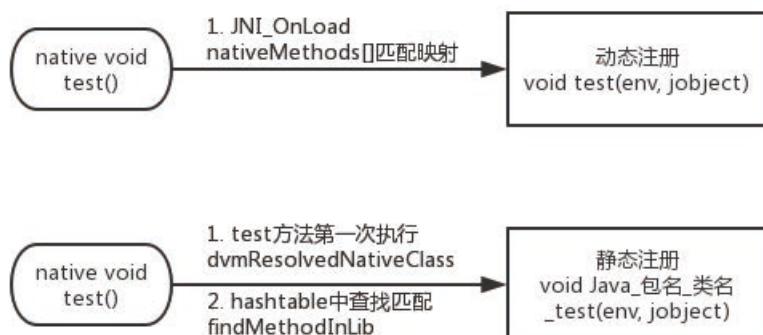


图 4-1 native 方法映射逻辑

总结下：

- 动态注册的 native 方法映射通过加载 so 库过程中调用 `JNI_OnLoad` 方法调用完成。

- 静态注册的 native 方法映射是在该 `native` 方法第一次执行的时候才完成映射，当然前提是该 so 库已经 load 过。

4.2 SO 库热部署实时生效可行性分析

4.2.1 动态注册 native 方法实时生效

前面我们分析过 so 库的加载原理，我们知道动态注册的 native 方法调用一次 `JNI_OnLoad` 方法都会重新完成一次映射，所以我们是否只要先加载原来的 so 库，然后再加载补丁 so 库，就能完成 Java 层 native 方法到 native 层 patch 后的新方法映射，这样就完成动态注册 native 方法的 patch 实时修复。一张图说明

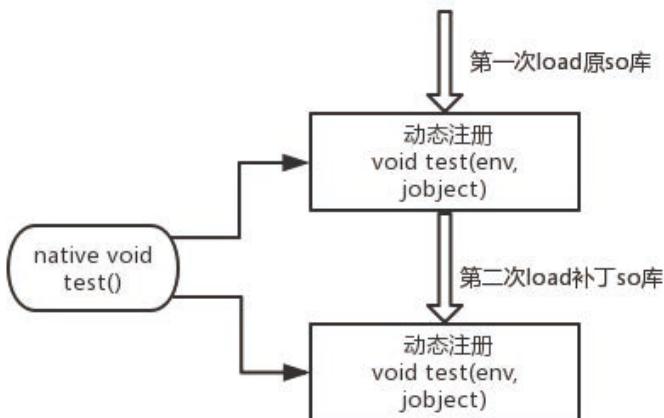


图 4-2 动态注册 native 方法实时生效

实测发现 art 下这样是可以做到实时生效的，但是 Dalvik 下做不到实时生效，通过代码测试我们发现，实际上 Dalvik 下第二次 load 补丁 so 库，执行的仍然是原来 so 库的 `JNI_OnLoad` 方法，而不是补丁 so 库的 `JNI_OnLoad` 方法，所以 Dalvik 下做不到实时生效。我们来简单分析下，既然拿到的是原来 so 库的 `JNI_OnLoad` 方

法，那么我们首先怀疑以下两个函数是否有问题。

- `dlopen()`: 返回给我们一个动态链接库的句柄
- `dlsym()`: 通过一个 `dlopen` 得到的动态连接库句柄，来查找一个 symbol

首先来看下 Dalvik 虚拟机下面 `dlopen` 的实现，源码在 `/bionic/linker/dlfcn.cpp` 文件，方法调用链路: `dlopen-> do_dlopen -> find_library -> find_library_internal`

```
static soinfo* find_library_internal(const char* name) {
    soinfo* si = find_loaded_library(name);
    if (si != NULL) { //so 库已经加载过
        if (si->flags & FLAG_LINKED) {
            return si; // 直接返回该 so 库的句柄
        }
        DL_ERR("OOPS: recursive link to \'%s\'", si->name);
        return NULL;
    }

    TRACE("[ '%s' has not been loaded yet. Locating...]", name);
    si = load_library(name); //so 库从未加载过，load_library 执行加载
    if (si == NULL) {
        return NULL;
    }
    return si;
}
```

`findloadedlibrary` 方法判断 `name` 表示的 so 库是否已经被加载过，如果加载过直接返回之前加载 so 库的句柄，没有加载过，调用 `load_library` 尝试加载 so 库

```
static soinfo *find_loaded_library(const char *name) {
    soinfo *si;
    const char *bname;

    // TODO: don't use basename only for determining libraries
    // http://code.google.com/p/android/issues/detail?id=6670
    bname = strrchr(name, '/');
    bname = bname ? bname + 1 : name;

    for (si = solist; si != NULL; si = si->next) {
        if (!strcmp(bname, si->name)) {
```

```

        return si;
    }
}
return NULL;
}

```

看代码注释，也知道其实这是 Dalvik 虚拟机下的一个 bug，这里它是通过 basename 做出查找，传进来的参数 name 实际上是 so 库所在磁盘的完整路径，比如此时修复后的 so 库的路径为 /data/data/com.taobao.jni/files/libnative-lib.so。但是此时是通过 bname:libnative-lib.so 作为 key 去查找，我们知道第一次加载原来的 so 库 System.loadLibrary("native-lib"); 实际上已经在 solist 表中存在了 native-lib 这个 key，所以 Dalvik **下面加载修复后的补丁 so 拿到的还是原 so 库文件的句柄，所以执行的仍然是原来 so 库的 JNI_OnLoad 方法，Art 下不存在这个问题，是因为 Art 下这个地方是以 name 作为 key 去查找而不是 bname，所以 art 下重新 load 一遍补丁 so 库，拿到的是补丁 so 库的句柄，然后执行补丁 so 库的 JNI_OnLoad**

所以为了解决 Dalvik 下面的这个问题，那么如果尝试对补丁 so 进行改名，比如此处补丁 so 库的完整路径修改之后变成 /data/data/com.taobao.jni/files/libnative-lib-123333.so，后面一串数字是当前时间戳，确保这个 bname 是全局唯一的，按照上面的分析，在 solist 中查找的 key 已经是唯一的，所以此时可以做到 Dalvik 下面动态注册的 native 方法的实时生效。

4.2.2 静态注册 native 方法实时生效

上面通过尝试对补丁 so 库进行重命名为全局唯一的名称可以确保第二次加载补丁 so 库可以做到 Dalvik 下和 Art 下动态注册方法的实时生效，但要做到静态注册 native 方法的实时生效还需要更多工作。

前面我们说过静态注册 native 方法的映射是在 native 方法第一次执行的时候就完成了映射，所以如果 native 方法在加载补丁 so 库之前已经执行过了，那么是否这

种时候这个静态注册的 native 方法一定得不到修复？幸运的是，系统 JNI API 提供了解注册的接口。

```

static jint UnregisterNatives(JNIEnv* env, jclass jclazz) {
    ClassObject* clazz = (ClassObject*) dvmDecodeIndirectRef(ts.self(),
jclazz);
    dvmUnregisterJNINativeMethods(clazz);
    return JNI_OK;
}
/*
 * Un-register all JNI native methods from a class.
 */
void dvmUnregisterJNINativeMethods(ClassObject* clazz) {
    unregisterJNINativeMethods(clazz->directMethods, clazz-
>directMethodCount);
    unregisterJNINativeMethods(clazz->virtualMethods, clazz-
>virtualMethodCount);
}
static void unregisterJNINativeMethods(Method* methods, size_t count) {
    while (count != 0) {
        count--;
        Method* meth = &methods[count];
        if (!dvmIsNativeMethod(meth))
            continue;
        if (dvmIsAbstractMethod(meth))      /* avoid abstract method stubs */
            continue;

        dvmSetNativeFunc(meth, dvmResolveNativeMethod, NULL); //meth-
>nativeFunc 重新指向 dvmResolveNativeMethod
    }
}

```

`UnregisterNatives` 函数会把 `jclazz` 所在类的所有 native 方法都重新指向为 `dvmResolveNativeMethod`，所以调用 `UnregisterNatives` 之后不管是静态注册还是动态注册的 native 方法之前是否执行过在加载补丁 so 的时候都会重新去做映射。所以我们只需要以下调用。

```

static void patchNativeMethod(JNIEnv *env, jclass cls) {
    env->UnregisterNatives(cls);
}

```

这里有一个难点，因为 native 方法的修改是在 SO 库中，所以我们的补丁工具很难检测出到底是哪个 Java 类需要解注册 native 方法。这个问题暂且放下。假设

我们能知道哪个类需要解注册 native 方法，然后 load 补丁 so 库之后，再次执行该 native 方法，这样看起来是可以让该 native 方法实时生效，但是测试发现，在补丁 so 库重命名的前提下，java 层 native 方法可能映射到原 so 库的方法，也可能映射到补丁 so 库的修复后的新方法。

首先静态注册的 native 方法之前从未执行，首先尝试解析该方法。或者调用了 `unregisterJNINativeMethods` 解注册方法，那么该方法将指向 `meth->nativeFunc = dvmResolveNativeMethod`，那么真正运行该方法的时候，实际上执行的是 `dvmResolveNativeMethod` 函数。这个函数主要完成 java 层 native 方法和 native 层方法的映射逻辑。

```

void dvmResolveNativeMethod(const u4* args, JValue* pResult,
    const Method* method, Thread* self) {
    ClassObject* clazz = method->clazz;
    ....
    /* now scan any DLLs we have loaded for JNI signatures */
    void* func = lookupSharedLibMethod(method); // 调用 lookupSharedLibMethod 方法，拿到 so 库文件对应的 native 方法函数指针。
    if (func != NULL) {
        /* found it, point it at the JNI bridge and then call it */
        dvmUseJNIBridge((Method*) method, func);
        (*method->nativeFunc)(args, pResult, method, self);
        return;
    }
    ....
    dvmThrowUnsatisfiedLinkError("Native method not found", method);
}

static void* lookupSharedLibMethod(const Method* method){
    return (void*) dvmHashForeach(gDvm.nativeLibs, findMethodInLib,
        (void*) method);
}

int dvmHashForeach(HashTable* pHashTable, HashForeachFunc func, void* arg){
    int i, val, tableSize;
    tableSize = pHashTable->tableSize;

    for (i = 0; i < tableSize; i++) {
        HashEntry* pEnt = &pHashTable->pEntries[i];
        if (pEnt->data != NULL && pEnt->data != HASH_TOMBSTONE) {
            val = (*func)(pEnt->data, arg);
        }
    }
}

```

```

        if (val != 0)
            return val;
    }
}
return 0;
}

```

`gDvm.nativeLibs` 是一个全局变量，它是一个 hashtable，存放着整个虚拟机加载 so 库的 SharedLib 结构指针。然后该变量作为参数传递给 `dvmHashForEach` 函数进行 hashtable 遍历。执行 `findMethodInLib` 函数看是否找到对应的 native 函数指针，如果第一个找到就直接 return，不在进行下次的查找。

这个结构很重要，在虚拟机中大量使用到了 hashtable 这个数据结构，hashtable 的实现源码在 `dalvik/vm/Hash.h` 和 `dalvik/vm/Hash.cpp` 文件中，有兴趣可以自行查看源码，这里不进行详细分析。hashtable 的遍历和插入都是在 `dvmHashTableLookup` 方法中实现，简单说下 `java.hashtable` 和 `c.hashtable` 的异同点：

- 共同点：两者实际上都是**数组实现**，hashtable 容量如果超过默认值都会进行扩容，都是对 `key` 进行 `hash` 计算然后跟 `hashtable` 的长度进行取模作为 `bucket`。
- 不同点：Dalvik 虚拟机下 hashtable put/get 操作实现方法，实际上实现要比 java hashmap 的实现要简单一些，java hashmap 的 put 实现需要处理 hash 冲突的情况，一般情况下会通过在**冲突节点上新增一个链表处理冲突**，然后 `get` 实现会遍历这个链表通过 `equals` 方法比较 `value` 是否一致进行**查找**，dalvik 下 hashtable 的 put 实现上 (`doAdd=true`) 只是简单的把指针下移直到下一个空节点。get 实现 (`doAdd=false`) 首先根据 hash 值计算出 bucket 位置，然后通过 `cmpFunc` 函数比较值是否一致，不一致，指针下移。hashtable 的遍历实际就是数组遍历实现

知道了 dalvik 下 hashtable 的实现原理，那我们再来看下前面提到的：补丁 so 库重命名的前提下，为什么 java 层 native 方法可能映射到原 so 库的方法也可能映

射到补丁 so 库的修复后的新方法。一张图说明情况

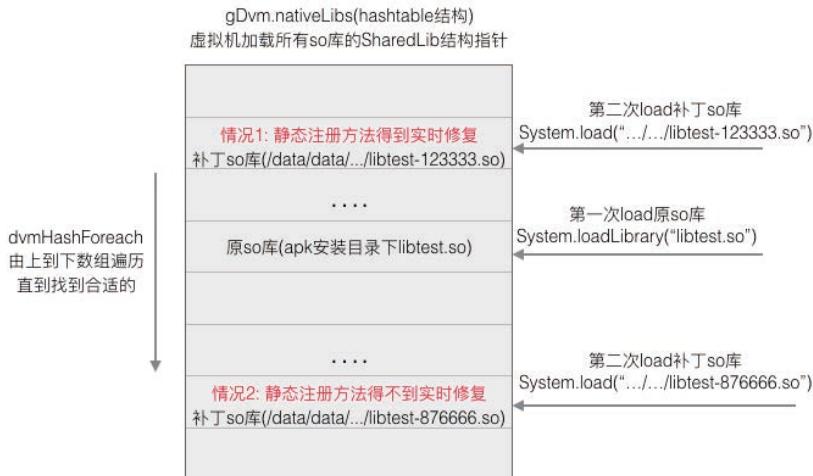


图 4-3 静态注册 native 方法实时生效

所以我们可以得到结论：

- 对补丁 so 库进行重命名后，如果这个补丁 so 库在 hashtable 中的位置比原 so 库的位置靠前，那么这个静态注册 native 方法就能够得到修复，位置如果靠后就得不到修复。

4.2.3 SO 实时生效方案总结

基于上面的分析，so 库的实时生效必须满足以下几点：

1. so 库为了兼容 Dalvik 虚拟机下动态注册 native 方法的实时生效，必须对 so 文件进行改名。
2. 针对 so 库静态注册 native 方法的实时生效，首先需要解注册静态注册的 native 方法，这个也是难点，因为我们很难知道 so 库中哪几个静态注册的 native 方法发生了变更。假设就算我们知道如果静态注册的 native 方法需要

解注册，重新load补丁so库也有可能被修复也有可能不被修复。

3. 上面对补丁so进行了第二次加载，那么肯定是多消耗了一次本地内存，如果补丁so库够大，补丁so够多，那么JNI层的OOM也不是没可能
4. 另外一方面补丁so如果新增了一个动态注册的方法而dex中没有相应方法，直接去加载这个补丁so文件会报NoSuchMethodError异常，具体逻辑在dvmRegisterJNIMethod中。我们知道如果dex如果新增了一个native方法，那么走不了热部署只能冷启动重启生效，所以此时补丁so就不能第二次load了。这种情况下so库的修复严重依赖于dex的修复方案。

可以看到SO库实时生效方案，对于静态注册的native方法有一定的局限性，不能满足一般的通用性，所以最后我们放弃了so库的实时生效需求，转而求次实现so库修复的冷部署重启生效方案。

4.3 SO库冷部署重启生效实现方案

为了更好的兼容通用性，我们尝试通过冷部署重启生效的角度分析下补丁so库的修复方案。

4.3.1 接口调用替换方案

sdk提供接口替换System默认加载so库接口

```
SOPatchManager.loadLibrary(String libName) -> 代替 System.loadLibrary(String libName)
```

SOPatchManager.loadLibrary接口加载so库的时候优先尝试去加载sdk指定目录下的补丁so，加载策略如下：

- 如果存在则加载补丁so库而不会去加载安装apk安装目录下的so库。
- 如果不存在补丁so，那么调用System.loadLibrary去加载安装apk目录下的so库。

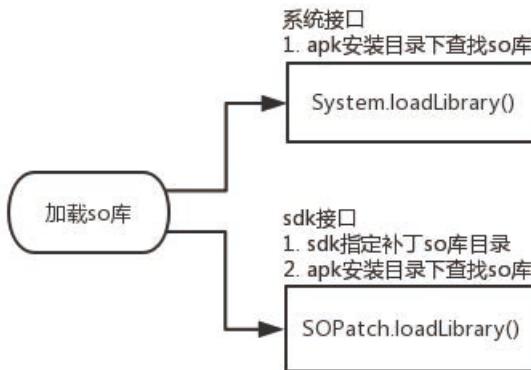


图 4-4 接口调用替换实现

我们可以很清楚的看到这个方案的优缺点：

- 优点：不需要对不同 sdk 版本进行兼容，因为所有的 sdk 版本都有 System.loadLibrary 这个接口。
- 缺点：调用方需要替换掉 System 默认加载 so 库接口为 sdk 提供的接口，如果是已经编译混淆好的三方库的 so 库需要 patch，那么是很难做到接口的替换。

虽然这种方案实现简单，同时不需要对不同 sdk 版本区分处理，但是有一定的局限性没法修复三方包的 so 库同时需要强制侵入接入方接口调用，接着我们来看下反射注入方案。

4.3.2 反射注入方案

前面介绍过 `System.loadLibrary("native-lib");` 加载 so 库的原理，其实 `native-lib` 这个 so 库最终传给 native 方法执行的参数是 `so 库在磁盘中的完整路径`，比如：`/data/app-lib/com.taobao.jni-2/libnative-lib.so`，so 库会在 `DexPathList.nativeLibraryDirectories/nativeLibraryPathElements` 变量所表示的目录下去遍历搜索。

sdk<23 DexPathList.findLibrary 实现如下

```
/** 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)) { //path文件存在同时
canOpenReadOnly, 返回该path
            return path;
        }
    }
    return null;
}
```

可以发现会遍历 `nativeLibraryDirectories` 数组，如果找到了 `IoUtils.canOpenReadOnly(path)` 返回为 true，那么就直接返回该 path，`IoUtils.canOpenReadOnly(path)` 返回为 true 的前提肯定是需要 path 表示的 so 文件存在的。那么我们可以采取类似类修复反射注入方式，只要把我们的补丁 so 库的路径插入到 `nativeLibraryDirectories` 数组的最前面就能够达到加载 so 库的时候是补丁 so 库而不是原来 so 库的目录，从而达到修复的目的。

sdk>=23 DexPathList.findLibrary 实现如下

```
/** List of native library path elements. */
private final Element[] nativeLibraryPathElements;
public String findLibrary(String libraryName) {
    String fileName = System.mapLibraryName(libraryName);
    for (Element element : nativeLibraryPathElements) {
        String path = element.findNativeLibrary(fileName);
        if (path != null) {
            return path;
        }
    }
    return null;
}
```

sdk23 以上 `findLibrary` 实现已经发生了变化，如上所示，那么我们只需要把补丁 so 库的完整路径作为参数构建一个 `Element` 对象，然后再插入到 `nativeLibraryPathElements` 数组的最前面就好了。

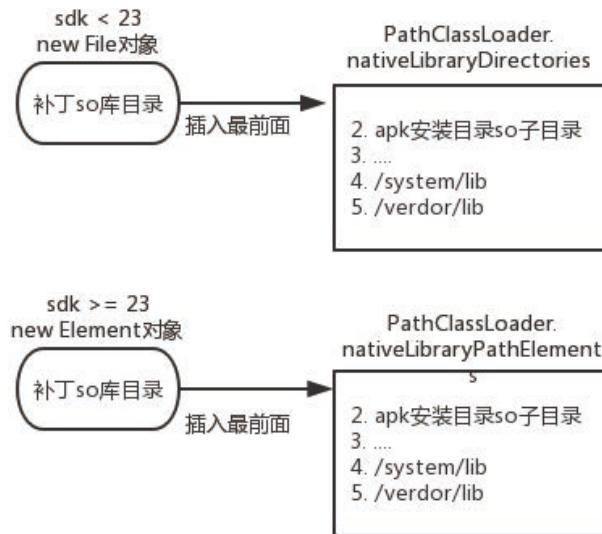


图 4-5 反射注入实现

- 优点：可以修复三方库的 so 库。同时接入方不需要像方案 1 一样强制侵入用户接口调用。
- 缺点：需要不断的对 sdk 进行适配，如上 sdk23 为分界线，findLibrary 接口实现已经发生了变化。

我们知道在不管是在补丁包中还是 apk 中一个 so 库都存在多种 cpu 架构的 so 文件，比如 "armeabi", "arm64-v8a", "x86" 等。加载肯定是加载其中一个 so 库文件的，如何选择机型对应的 so 库文件将是重点所在。

4.4 如何正确复制补丁 SO 库

上面提到的一个问题，这里不打算详细介绍。有需要的参考文档：[Android 动态链接库加载原理及 HotFix 方案介绍](#)，这篇文档有些观点不尽正确，但是我也能知道虚拟机究竟选择哪个 `abis` 目录作为参数构建 `PathClassLoader` 对象，一张图简单了解下原理：

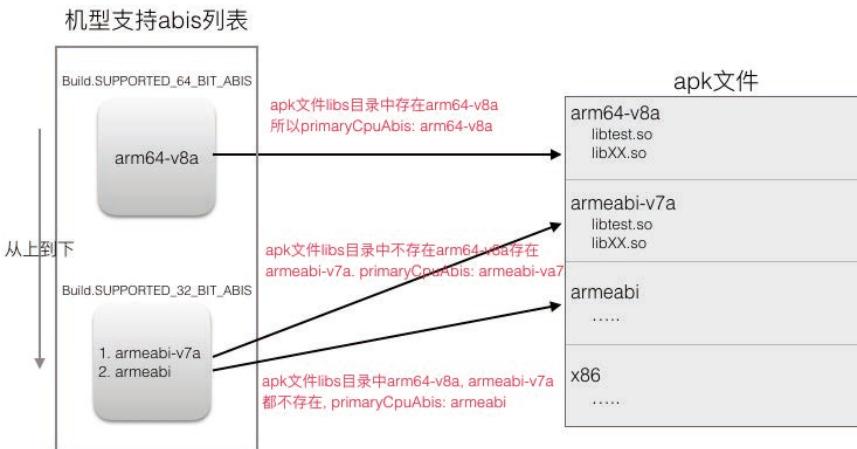


图 4-6 如何选择 primaryCpuAbis

实际上补丁 so 也存在类似的问题，我们的补丁 so 库文件放到补丁包的 libs 目录下面，`libs` 目录和 `.dex` 文件和 `res` 资源文件一起打包成一个压缩文件作为最后的补丁包，`libs` 目录可能也包含多种 abis 目录。所以我们需要选择手机最合适的 `primaryCpuAbi`，然后从 `libs` 目录下面选择这个 `primaryCpuAbi` 子目录插入到 `nativeLibraryDirectories/nativeLibraryPathElements` 数组中。所以怎么选择 `primaryCpuAbi` 是关键，来看下我们 sdk 具体的实现

```

static {
    try {
        PackageManager pm = mApp.getPackageManager();
        if (pm != null) {
            ApplicationInfo mAppInfo = pm.getApplicationInfo(mApp.
getPackageName(), 0);
            if (mAppInfo != null) {
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.
LOLLIPOP) { //sdk>=21
                    Field thirdFiled = ApplicationInfo.class.
getDeclaredField("primaryCpuAbi");
                    thirdFiled.setAccessible(true);
                    String cpuAbi = (String) thirdFiled.get(mAppInfo);
                    primaryCpuAbis = new String[]{cpuAbi};
                } else { //sdk<21
                    primaryCpuAbis = new String[]{

```

```
        Build.CPU_ABI, Build.CPU_ABI2
    );
}
}
}
} catch (Throwable t) {
LogTool.e(TAG, "SOPatchManager static block", t);
}
}
```

- sdk>=21 时，直接反射拿到 ApplicationInfo 对象的 primaryCpuAbi 即可
- sdk<21 时，由于此时不支持 64 位，所以直接把 Build.CPU_ABI, Build.CPU_ABI2 作为 primaryCpuAbi 即可

4.5 本章小结

对于SO库的修复方案目前更多采取的是接口调用替换方式，需要强制侵入用户接口调用。目前我们的SO文件修复方案采取的是反射注入的方案，重启生效。具有更好的普遍性。如果有SO文件修复实时生效的需求，也是可以做到的，只是有些限制情况。

第 5 章 热修复未来展望

热修复的专业性

热修复是一个与业务完全无关的模块，开发者如果要自己实现一套可靠的热修复框架，将花费大量时间和精力。虽然市面上已经有很多开源的热修复实现，然而其中的很多坑，往往要踩过才知道，等你把这些坑一一踩过之后，可能大量的用户已经对你失去信心。所以，依靠一个稳定可靠、而且简单实用的商业版本，反而能使各方面的成本降到最低。并且，热修复并不是简单的客户端 SDK，它还包含了安全机制和服务端的控制逻辑，这整条链路也不是短时间内可以快速完成的。

还是那句老话，专业是事交给专业的人去做。开发者应该把更多时间精力放到自己的核心业务之中。

因此，我们把 Sophix 研发过程中遇到的问题和技术细节与大家进行分享，揭开这一简单优雅的方案背后所付出的艰辛与探索。大家可以直接访问[阿里云移动热修复官网](#)进行体验，文档详实，几分钟就能搞定。Sophix 提供了一套更加完美的客户端服务端一体的热更新方案。做到了图形界面一键打包、加密传输、签名校验和服务端控制发布与灰度功能，让你用最少的时间实现最强大可靠的全方位热更新。并且在代码修复、资源修复、SO 库修复方面，都做到了业界最佳。

对 Android 的生态的影响

很多人会把热修复技术跟其他国内厂商的“黑科技”混为一谈。有人说，你们国内开发者就是瞎搞，就不能给我们 Android 用户一个更加纯净的环境吗？

这里我需要澄清一下。热修复技术不同于其他国内的 Android “黑科技”。就比如，国内 Android 进程保活，是让 app 持续驻留在后台避免被系统杀死，这既耗费手机电量又占内存，浪费了很多手机资源。再比如，app 自行定制的推送服务，无节操地对用户进行信息轰炸。还有更过分的全家桶，一个 app 同时拉起一票 app，并且长期占着内存，使得手机卡顿不堪。总归，这些技术都是为了 app 厂商的利益而损害手机使用者的实际体验。

而热修复技术是完全不同的，它达到的是一个手机用户和开发者双赢的目的。不仅厂商可以快速迭代更新 app，使得功能能最快上线。并且由于热更新过程是毫无感知的，手机用户也减少了繁琐的更新步骤，节省了大量等待更新的时间。这实际上是改善了 Android 的生态环境。只是这其中最重要的，是要保证热修复功能的稳定性。而 Sophix 的稳定性，是经过了无数开发者检验的，并且还有手淘多年深厚的技术沉淀作为保障。

| Android 与 iOS 热修复的不同

前段时间，苹果封杀了 iOS 的热修复功能，这给 iOS 的开发者带来了很大困扰。

热修复功能被禁止，会使得很多 app 不得不靠直接发版进行更新，这样一旦新版本出了问题，整个更新迭代过程变得十分漫长。并且一些试验性功能无法进行灰度，这就使得一个重要功能的更新将直接全量发版，如果功能不够稳定，波及范围就变得非常广。而且，用户需要重新下载整个 app，不仅流程漫长，原本不到 1MB 的补丁就能解决的事，现在不得不下载几十甚至上百 MB 的完整包才能更新。

苹果这一政策的推出，使得很多人也因此不看好 Android 的热修复技术了。在这里，我们可以打消这种错误的观念。因为 Android 的情况和 iOS 是有极大不同的。主要有两个方面：

再看回 Android 的情况，Android 热修复和 iOS 是有极大不同的。主要有两个方面：

- 谷歌和苹果在中国的地位不同
- Android 和 iOS 的开放性不同

谷歌在中国没有像苹果那样的控制力，即使它想要封杀也不可能，国内是有各个安卓应用市场的，没有统一的 app 安装渠道。另外，Android 是开源的，各个厂商都可以做定制，想统一各家的安装渠道几乎是不可能的。

未来，无限可能！

我们对于未来是很乐观的，Android 的热修复领域不仅不会受到封杀，反而还有很大的发展空间。我们正在尝试支持各大加固厂商，目前阿里聚安全修复已经支持了 Sophix，热修复结合安全加固，将会使得 app 的稳定性和安全性更加坚不可摧。甚至后续还可以与系统厂商合作，对系统 app 乃至系统组件进行修复，这样就可以避免频繁 OTA 升级。

因此，热修复所能发挥的价值将是十分巨大的。热修复还可以与其他领域进行碰撞，引发无限的可能性。在这里，我们非常期待携手广大应用厂商以及 ROM 厂商，共同推动 Android 的生态更加完善。

附录

Sophix 方案纵向比较

方案对比	Andfix开源版本	阿里Hotfix 1.X	阿里Hotfix 2.0
方法替换	支持, 除部分情况 ^[0]	支持, 除部分情况	全部支持
方法增加减少	不支持	不支持	以冷启动方式支持 ^[1]
方法反射调用	只支持静态方法	只支持静态方法	以冷启动方式支持
即时生效	支持	支持	视情况支持 ^[2]
多 DEX	不支持	支持	支持
资源更新	不支持	不支持	支持
so 库更新	不支持	不支持	支持
Android 版本	支持 2.3~7.0	支持 2.3~6.0	全部支持包含 7.0 以上
已有机型	大部分支持 ^[3]	大部分支持	全部支持
安全机制	无	加密传输及签名校验	加密传输及签名校验
性能损耗	低, 几乎无损耗	低, 几乎无损耗	低, 仅冷启动情况下有些损耗
生成补丁	繁琐, 命令行操作	繁琐, 命令行操作	便捷, 图形化界面
补丁大小	不大, 仅变动的类	小, 仅变动的方法	不大, 仅变动的资源和代码 ^[4]
服务端支持	无	支持服务端控制 ^[5]	支持服务端控制

说明:

[0] 部分情况指的是构造方法、参数数目大于 8 或者参数包括 long,double,float 基本类型的方法。

[1] 冷启动方式, 指的是需要重启 app 在下次启动时才能生效。

[2] 对于 Andfix 及 Hotfix 1.X 能够支持的代码变动情况, 都能做到即时生效。而对于 Andfix 及 Hotfix 1.X 不支持的代码变动情况, 会走冷启动方式, 此时就无法做到即时生效。

[3] Hotfix 1.X 已经支持绝大部分主流手机, 只是在 X86 设备以及修改了虚拟机底层结构的 ROM 上不支持。

[4] 由于支持了资源和库, 如果有这些方面的更新, 就会导致的补丁变大一些, 这个是很正常的。并且由于只包含差异的部分, 所以补丁已经是最大程度的小了。

[5] 提供服务端的补丁发布和停发、版本控制和灰度功能, 存储开发者上传的补丁包。

Sophix 方案横向比较

方案对比	Sophix	Tinker	Amigo
DEX 修复	同时支持即时生效修复和冷启动修复	冷启动修复	冷启动修复
资源更新	差量包，不用合成	差量包，需要合成	全量包，不用合成
SO 库更新	插桩实现，开发透明	替换接口，开发不透明	插桩实现，开发透明
性能损耗	低，仅冷启动情况下有些损耗	高，有合成操作	低，全量替换
生成补丁	直接选择已经编好的新旧包在本地生成	编译新包时设置基线包	上传完整新包到服务端
补丁大小	小	小	大
接入成本	傻瓜式接入	复杂	一般
Android 版本	全部支持	全部支持	全部支持
安全机制	加密传输及签名校验	加密传输及签名校验	加密传输及签名校验
服务端支持	支持服务端控制	支持服务端控制	支持服务端控制



Sophix 答疑及
技术交流钉钉群



关注「阿里技术」公众号
把握前沿技术脉搏
