浙江大學

硕 士 研 究 生 读 书 报 告

题目 Android 热修复方案的学习

作者姓名 陈 明

作者学号 21651096

指导教师 李启雷

学科专业 软件工程专业

所在学院 软件学院

提交日期 二〇一七 年 一 月

Study On Android HotFix Schemes

A Dissertation Submitted to

Zhejiang University

in partial fulfillment of the requirements for

the degree of

Master of Engineering

Major Subject: Software Engineering

Advisor: Qilei Li

By

Ming Chen

Zhejiang University, P.R. China

2016

摘要

用户体验是移动产品核心竞争力之一，而 bug 解决也是用户体验的一部分，传统 bug 修复往往伴随着新版本的发布，用户需要重新安装，体验性差。因此，android 开始领域出现各种热修复方案。本文就是对目前市场上的几个流行的 android 热修复方案，包括 QQ 空间补丁技术、微信 tinker 和阿里百川 HotFix，进行学习和对比。

关键词：android，热修复

Abstract

User experience is the core competitiveness of mobile products and bug solving is part of the user experience.The traditional bug fixes are often followed by a new version of the release. Users need to re-install so the user experience is poor. Therefore, android HotFix appeared. This article is aim to learn and contrast several popular android HotFix on the market today, including QQ space patch technology, WeChat tinker and AliBaiChuan HotFix.

Keywords: android,HotFix

1 引言

传统开发流程中，发现 bug 之后，需要进行 bug 修复然后重新上线，用户进行安装，但是，这样带来了一系列的弊端，比如用户下载安装成本高，应用体验差，用户不一定来得及更新等，为此，热修复显得尤为重要。

所谓热修复，简单的说就是通过事先设定的接口从网上下载没有问题的代码来替换有问题的代码从而解决 bug，在这一过程中，无需发布新版本，用户不会有什么感觉，用户体验相比传统开发有了很大的提升。

最近，android 开发领域对于热修复技术的讨论越来越热烈，同时也诞生了很多很好的解决方案，包括 QQ 空间补丁技术、16 年 9 月开源的微信 Tinker 和阿里百川在 16 年 11 月上线的基于 AndFix 的热修复方案等，主要可以分为两类：一类是基于 multidex 的热更新框架，如 QQ 空间补丁技术、微信 Tinker；另一类就是 native hook 方案，如。阿里百川 HotFix，它们在原理上各有不同各有优缺点。本文就现在市场上的 3 个优秀解决方案进行学习，第 2 章到第 4 章分别针对 QQ 空间补丁技术、微信 Tinker、阿里百川 HotFix 原理纵向分析，第 5 章将三种方案进行横向对比。

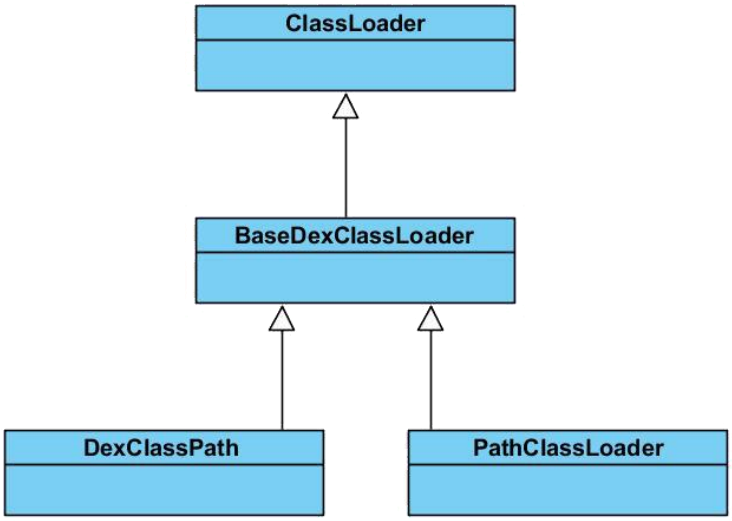
2 QQ 空间补丁技术

2.1 原理

在 android 中，有个很重要的文件就是 dex 文件，dex 文件就是将整个 android 工程中所有的 class 压缩到一个(或几个) dex 文件中，合并了每个 class 的常量、class 版本信息等，例如每个 class 中都有一个相同的字符串，在 dex 中就只存一份就够了。所以，在 android 系统中，虚拟机是无法识别一个普通 class 文件的。

Java 执行程序的时候需要用到虚拟机 jvm 将字节码转换为机器码，这个过程需要用到 ClassLoader 类加载器。同理，Android 也有自己的虚拟机 dalvik 执行同样的功能，它的过程用到 PathClassLoader 类和 DexClassPath 类。通过查询 android 官方的 api 文档可以知道，PathClassLoader 类用于加载系统类和已经安装的类，DexClassPath 类用以加载

jar 包或者未安装的 apk。通过查看两者的源码可以知道，两者均继承在 BaseDexClassLoader 类的基础上重写了构造函数，核心逻辑都放在 BaseDexClassLoader 类中，再深入阅读 BaseDexClassLoader 的源码可以发现其继承了 ClassLoader 类，因此，通过阅读源码，可以得到了如下的类继承关系图[1]。



为了理解类加载的机制，深入阅读 BaseDexClassLoader.java 的源码[2]。可以发现，BaseDexClassLoader 类中有个 findClass 方法用于寻找类。FindClass 类调用了 DexPathList 类的实例化对象 pathList 的 findClass 方法，如果找到就返回该类，否则就抛出 ClassNotFound 异常。这个思路比较容易理解。

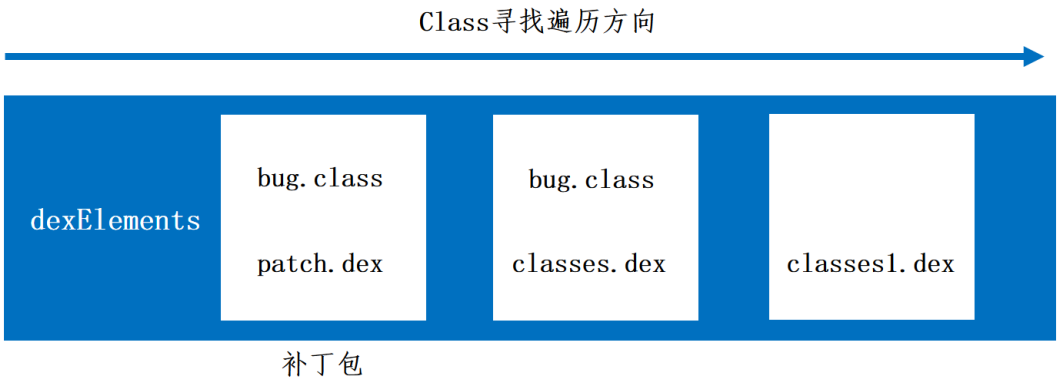
```
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    Class c = pathList.findClass(name, suppressedExceptions);
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException("Didn't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

由于上面的过程调用了 DexPathList 类中的 findClass 方法，所以继续阅读 DexPathList 类[2]中 findClass 的实现源码。一个 ClassLoader 可以包含多个 dex 文件，每个 dex 文件是一个 Element，多个 dex 文件排列成一个有序的数组 dexElements，findClass 的实现方法是遍历 Element 中的所有 dex，然后试着加载这个 dex 中的要寻找的 Class，如果找到要找的 Class 就直接返回，无论后面的 dex 还有没有这个 Class，如果遍历一遍之后还是没找到要找的 Class，就直接返回 null。

```
public Class findClass(String name, List<Throwable> suppressed) {
    for (Element element : dexElements) {
        DexFile dex = element.dexFile;

        if (dex != null) {
            Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
            if (clazz != null) {
                return clazz;
            }
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
    return null;
}
```

基于以上的类加载的过程，HotFix 的原理就是将存在 bug 的类进行修改，单独打包，即所谓的补丁包，然后把这个补丁包的 dex 插入到 dexElements 的最前面，这样，最先加载的就会是修改完的 Class。如下图所示，假设 bug.java 中有部分代码写错了，那么只需要把修改完的 bug.java 编译成 bug.class，然后将其打包，放到指定位置下，然后将其加入到 dexElements 的最前面，在扫描 bug 类的时候，就会先扫描到 patch 中的 bug 类，从而完成 bug 修复。



以上过程在实践的时候，在某些时候可能会出现“Class resolved by unexpected DEX”，通过网上查资料，可以找到出现这种错误的原因。假设 classes.dex 中有两个类 A、B，其中 A 调用了 B，现在发现 B 出现了 bug，于是对 B 进行了修复，同时打成补丁包按照上面的过程放入，在 A 调用 B 的时候，会发现 B 不在自己所在的 dex 包内，于是报错。在 apk 安装的时候，虚拟机会将 dex 优化成 odex 后才拿去执行。在这个过程中会对所有 class 一个校验。 校验方式：假设 A 该类在它的 static 方法，private 方法，构造函数，override 方法中直接引用到 B 类。如果 A 类和 B 类在同一个 dex 中，那么 A 类就会被打上 CLASS_ISPREVERIFIED 标记，被打上这个标记的类不能引用其他 dex 中的类，否则就会报该错误。为了解决这种错误，只需让 A 类不要被打上

CLASS_ISPREVERIFIED 标记即可，根据打上 CLASS_ISPREVERIFIED 标记的规则可以知道，只需让 A 类不要成为第一层引用者或者让 A 类去引用其他 dex 中的方法。网上也有给了相应的解决方法，即在类 A 的构造函数打印其它 dex 中的类。

```
// A 类
public class A{
    public A(){
        //新增部分
        System.out.println(AntilazyLoad.class) ;
    }
}
```

其中，AntilazyLoad 在其他 dex 里面，这样，由于 A 使用了其他 dex 中的类，所以不会被打上 CLASS_ISPREVERIFIED 标记。于是，上面报的错误就能得到解决。

```
// AntilazyLoad 类
public class AntilazyLoad{


}
```

2.2 实践

下面，对 QQ 空间补丁技术进行简单的实践。

第一步 为了可以进行热修复，在 Application 类中进行判断是否有补丁包，如果有补丁包的话，加载补丁包。

```
@Override
public void onCreate() {
    super.onCreate();
    injectHack();
    // 获取补丁，如果存在就执行注入操作
    String dexPath = Environment.getExternalStorageDirectory().getAbsolutePath().concat("/patch_dex.jar");
    File file = new File(dexPath);
    if (file.exists()) {
        inject(dexPath);
    } else {
        Log.e("HotPatchApplication", dexPath + "不存在");
    }
}
```

其中，加载补丁包 inject 函数其实是把补丁包插入到 Element 中，这一过程的流程为：获取 dexElements、获取补丁包、插入补丁包、更新 dexElements。

（1）获取 dexElements

由于 dexElements 外部不能调用，所以使用 Java 的反射机制获得其属性值。

```
/**
 * 通过反射获取对象的属性值
 */
private Object getField(Class<?> cl, String fieldName, Object object) throws NoSuchFieldException,
    IllegalAccessException {
    Field field = cl.getDeclaredField(fieldName);
    field.setAccessible(true);
    return field.get(object);
}
```

（2）获取补丁包

（3）插入补丁包，插入补丁包的时候必须保证是插入到数组头部，才能进行优先加载。

```
private Object combineArray(Object firstArr, Object secondArr) {
    int firstLength = Array.getLength(firstArr);
    int secondLength = Array.getLength(secondArr);
    int length = firstLength + secondLength;

    Class<?> componentType = firstArr.getClass().getComponentType();
    Object newArr = Array.newInstance(componentType, length);
    for (int i = 0; i < length; i++) {
        if (i < firstLength) {
            Array.set(newArr, i, Array.get(firstArr, i));
        } else {
            Array.set(newArr, i, Array.get(secondArr, i - firstLength));
        }
    }
    return newArr;
}
```

（4）更新 dexElements

同样也是通过 Java 的反射机制完成。

```
/**
 * 通过反射设置对象的属性值
 */
private void setField(Class<?> cl, String fieldName, Object object, Object value) throws NoSuchFieldException,
    IllegalAccessException {
    Field field = cl.getDeclaredField(fieldName);
    field.setAccessible(true);
    field.set(object, value);
}
```

根据前面四个过程，可以实现加载补丁包 inject 函数。

```
private void inject(String path) {
    try {
        // 获取classes的dexElements
        Class<?> cl = Class.forName("dalvik.system.BaseDexClassLoader");
        Object pathList = getField(cl, "pathList", getClassLoader());
        Object baseElements = getField(pathList.getClass(), "dexElements", pathList);

        // 获取patch_dex的dexElements（需要先加载dex）
        String dexopt = getDir("dexopt", 0).getAbsolutePath();
        DexClassLoader dexClassLoader = new DexClassLoader(path, dexopt, dexopt, getClassLoader());
        Object obj = getField(cl, "pathList", dexClassLoader);
        Object dexElements = getField(obj.getClass(), "dexElements", obj);

        // 合并两个Elements
        Object combineElements = combineArray(dexElements, baseElements);

        // 将合并后的Element数组重新赋值给app的classLoader
        setField(pathList.getClass(), "dexElements", pathList, combineElements);

    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    }
}
```

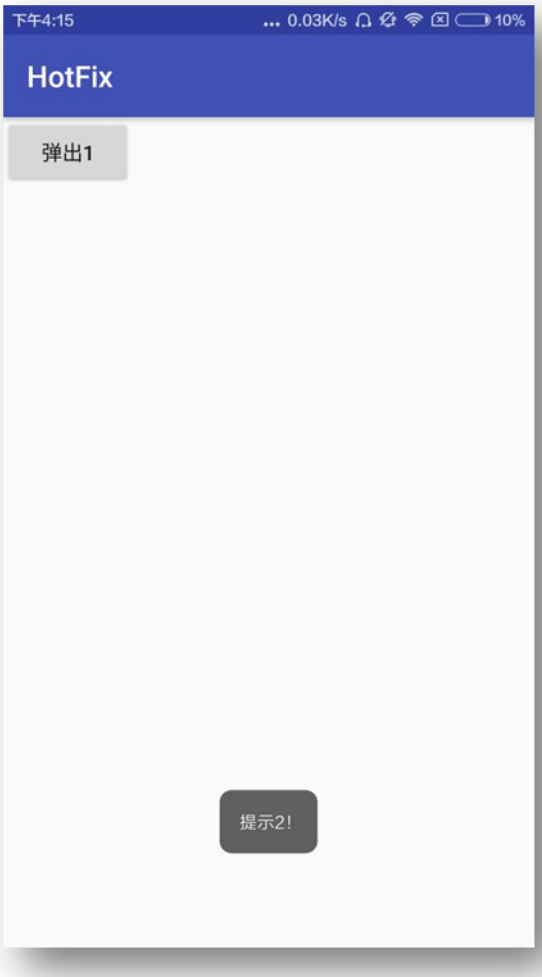
由于前面涉及到外部存储的读和写，所以需要在配置文件 AndroidManifest.xml 中声明相关的权限。

<!--SD 卡读取权限-->

<permission name="android.permission.WRITE_EXTERNAL_STORAGE"/>

<permission name="android.permission.READ_EXTERNAL_STORAGE"/>

第二步 先写一个有错误的简单页面，如下图所示。点击按钮本来应该弹出“提示 1”，结果却弹出“提示 2”。



第三步 修复这个问题，改成“提示 1!”。然后重新编译，将编译完之后把修改后的 class 文件单独提取出来。建一个和项目包一样的路径。

利用 jar 命令打包成 jar 包。

jar –cvf patch.jar chenming

```
C:\Users\陈明\Desktop\hotfix>jar -cvf patch.jar chenming
已添加清单
正在添加: chenming/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: chenming/com/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: chenming/com/hotfix/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: chenming/com/hotfix/MainActivity.class(输入 = 1098) (输出 = 594)(压缩了 45%)
```

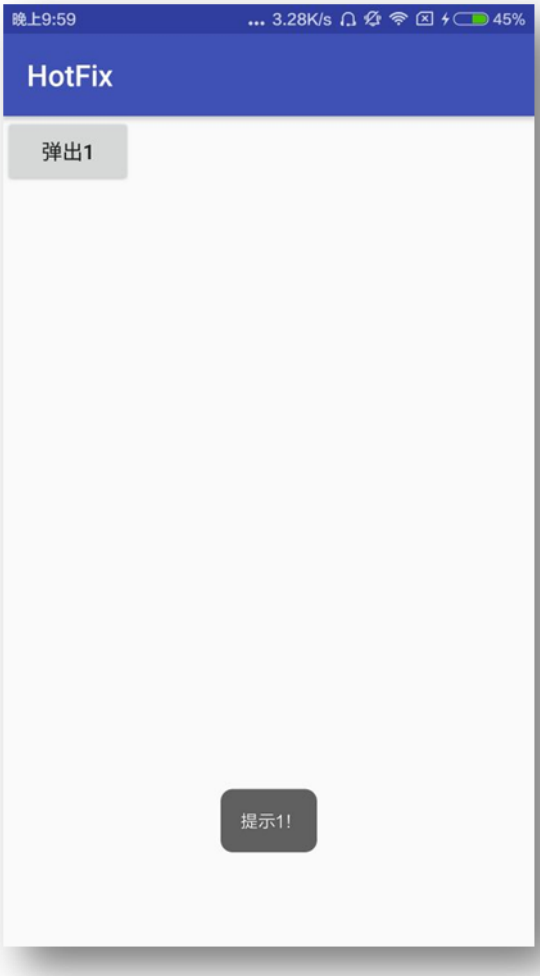
利用 dx 命令打包成补丁包。

dx –dex –output=patch_dex.jar patch.jar

```
C:\Users\陈明\Desktop\hotfix>dx --dex --output=patch_dex.jar patch.jar
C:\Users\陈明\Desktop\hotfix>
```

第四步 将补丁包 patch_dex.jar 放到 SD 卡路径下，如果找不到 SD 卡路径可以用 Environment.getExternalStorageDirectory().getAbsolutePath()打印出路径位置。

第五步 重启应用，可以发现已经修复了前面说到的 bug，点击“弹出 1”弹出的就是预期的“提示 1”。



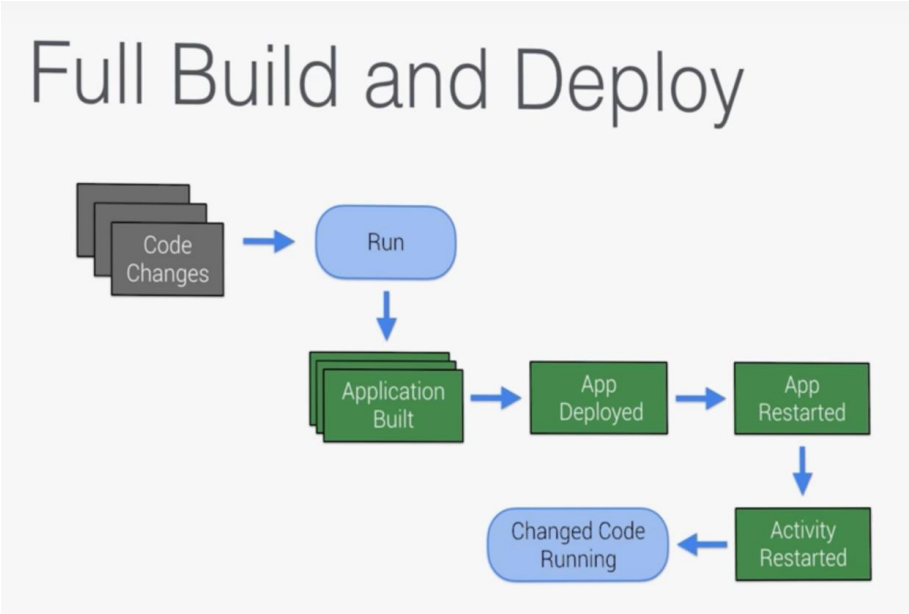
3 微信 Tinker

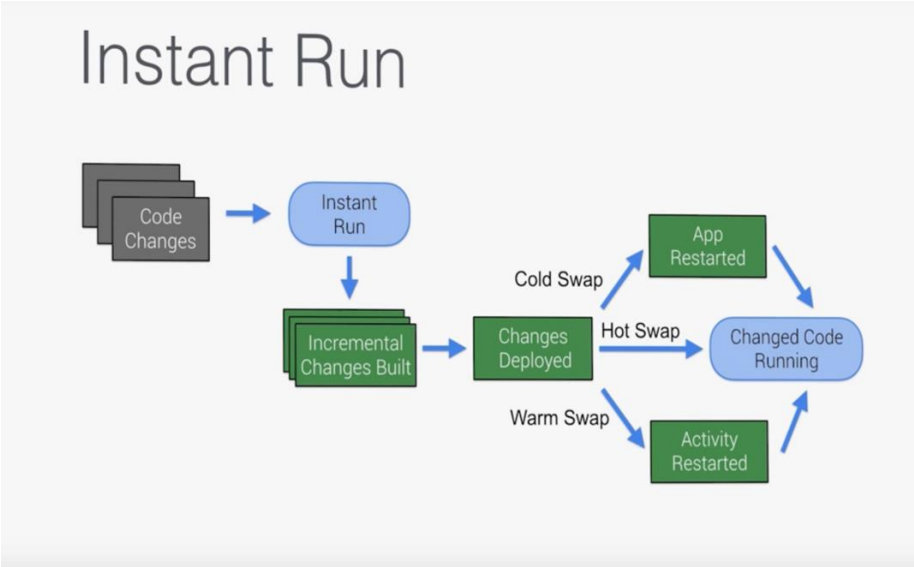
3.1 原理

HotFix 明显会在开启时候有一定的性能影响，为了解决这个问题，微信有了一个新的解决方案，即 Tinker。Tinker 大致的流程就是在编译的时候，通过对比新旧两个 dex 生成差异部分的 patch.dex，在运行时候，将差异的 patch.dex 与原安装包的就 Dex 合并成新的 dex[3]。

其中，合成新的 dex 是基于 android studio2.0 新出的 Instant Run，所谓 Instant Run 字面意思就是即时运行。在没有 Instant Run 时，代码变更之后运行，需要构建 Application，

然后工程的 Mainfest 文件被合并，和资源文件一起被打包到 APK 里面，类似的，Java 文件被编译成字节码，然后转换成 DEX 文件，最后部署 App，重新启动 App，重新启动 Activity，最终变更的代码才能变执行。之间需要花费大量的时间，工程小还好，一旦大了就需要花费几分钟甚至几十分钟时间。和完整构建、停止、重新安装和加载 App 相比，Instant Run 只增量构建和部署代码改变的部分，在很大程度上缩短构建和部署的时间。Instant Run 有 3 种交换类型，分别是 Hot Swap、Warm Swap 和 Cold Swap，Instant Run 会根据改变代码的类型，自动决定使用哪种类型。两者的区别可以从下图中看出来。





同时，为了可以让补丁包尽可能的小，微信提出来一种 DexDiff 算法，充分利用了 Dex 的格式信息来降低差异大小。步骤主要是（1）计算出 new dex 中每项 Section 的大小，比如 string_ids 在 dex 文件中所占大小；（2）根据表中前一项的偏移地址和大小，计算出每项 Section 的偏移地址；（3）调用 DexSectionDiffAlgorithm.execute()，将 new dex 与 old dex 中的每项 section 进行对比，对于每项 Section，遍历其每一项 Item，进行新旧对比，记录 ADD，DEL 标识，存放于 patchOperationList 中。接着遍历 patchOperationList，

添加 REPLACE 标识，最后将 ADD，DEL，REPLACE 操作分别记录到各自的 List 中；
（4）调用 DexPatchGenerator.writePatchOperations()，将记录写入补丁。

这样做的优点在于合成整包，性能得到了提高，但是也和 QQ 空间超级补丁技术存在一样的缺点，不能即时生效，而且需要开启新进程才能合并，容易因为内存不足而导致合并失败。

3.2 源码解读

Tinker 源码目前已经开源，地址[4]：

https://github.com/Tencent/tinker

源码的入口是 TinkerLoader.java，其中的 tryLoad 方法就是加载 dex 的方法。

```
/**
 * only main process can handle patch version change or incomplete
 */
@Override
public Intent tryLoad(TinkerApplication app, int tinkerFlag, boolean tinkerLoadVerifyFlag) {
    Intent resultIntent = new Intent();

    long begin = SystemClock.elapsedRealtime();
    tryLoadPatchFilesInternal(app, tinkerFlag, tinkerLoadVerifyFlag, resultIntent);
    long cost = SystemClock.elapsedRealtime() - begin;
    ShareIntentUtil.setIntentPatchCostTime(resultIntent, cost);
    return resultIntent;
}
```

这个方法首先会统计目前的时间，结束之后统计结束时间，从而计算合并时间。其中最重要的是 tryLoadPatchFilesInternal 方法，这个方法就是最核心的合并 dex 文件的方法。深入这个方法，这个方法实现的代码量比较多。首先，经过了一系列的安全性的检验，在 TinkerLoader.java 文件中的 225 行左右的位置开始用 TinkerDexLoader.loadTinkerJars 的方法加载 jar 包。

```
//now we can load patch jar
if (isEnabledForDex) {
    boolean loadTinkerJars = TinkerDexLoader.loadTinkerJars(app, tinkerLoadVerifyFlag, patchVersionDirectory,
        resultIntent, isSystemOTA);
    if (!loadTinkerJars) {
        Log.w(TAG, "tryLoadPatchFiles:onPatchLoadDexesFail");
        return;
    }
}
```

打开 TinkerDexLoader.java，里面有 loadTinkerJars 方法，遍历 dexList，找出 md5 符合校验通过的。然后调用 SystemClassLoaderAdder 的 installDexes 方法插入。

```
final boolean isArtPlatForm = ShareTinkerInternals.isVmArt();
for (ShareDexDiffPatchInfo info : dexList) {
    //for dalvik, ignore art support dex
    if (isJustArtSupportDex(info)) {
        continue;
    }
    String path = dexPath + info.realName;
    File file = new File(path);

    if (tinkerLoadVerifyFlag) {
        long start = System.currentTimeMillis();
        String checkMd5 = isArtPlatForm ? info.destMd5InArt : info.destMd5InDvm;
        if (!SharePatchFileUtil.verifyDexFileMd5(file, checkMd5)) {
            //it is good to delete the mismatch file
            ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR_LOAD_PATCH_VERSION_DEX_MD5_MISMATCH);
            intentResult.putExtra(ShareIntentUtil.INTENT_PATCH_MISMATCH_DEX_PATH,
                file.getAbsolutePath());
            return false;
        }
        Log.i(TAG, "verify dex file:" + file.getPath() + " md5, use time: " + (System.currentTimeMillis() - start));
    }
    legalFiles.add(file);
}
```

```
try {
    SystemClassLoaderAdder.installDexes(application, classLoader, optimizeDir, legalFiles);
} catch (Throwable e) {
    Log.e(TAG, "install dexes failed");
      e.printStackTrace();
    intentResult.putExtra(ShareIntentUtil.INTENT_PATCH_EXCEPTION, e);
    ShareIntentUtil.setIntentReturnCode(intentResult, ShareConstants.ERROR_LOAD_PATCH_VERSION_DEX_LOAD_EXCEPTION);
    return false;
}
```

其中 SystemClassLoaderAdder 的 installDexes 方法实现如下。很明显，Tinker 对不同的 Android 版本做了不同的处理。

```
public static void installDexes(Application application, PathClassLoader loader, File dexOptDir, List<File> files)
    throws Throwable {

    if (!files.isEmpty()) {
        ClassLoader classLoader = loader;
        if (Build.VERSION.SDK_INT >= 24) {
            classLoader = AndroidNClassLoader.inject(loader, application);
        }
        //because in dalvik, if inner class is not the same classloader with it wrapper class.
        //it won't fail at dex2opt
        if (Build.VERSION.SDK_INT >= 23) {
            V23.install(classLoader, files, dexOptDir);
        } else if (Build.VERSION.SDK_INT >= 19) {
            V19.install(classLoader, files, dexOptDir);
        } else if (Build.VERSION.SDK_INT >= 14) {
            V14.install(classLoader, files, dexOptDir);
        } else {
            V4.install(classLoader, files, dexOptDir);
        }
        //install done
        sPatchDexCount = files.size();
        Log.i(TAG, "after loaded classloader: " + classLoader + ", dex size:" + sPatchDexCount);

        if (!checkDexInstall(classLoader)) {
            //reset patch dex
            SystemClassLoaderAdder.uninstallPatchDex(classLoader);
            throw new TinkerRuntimeException(ShareConstants.CHECK_DEX_INSTALL_FAIL);
        }
    }
}
```

以最新的 V23 即 Android 6 为例，跟上一章一样，源码中首先为了获取到 BaseDexClassLoader 中的 dexPathList 采用了 java 的反射机制得到，然后调用

ShareReflectUtil.expandFieldArray()。

```
private static final class V23 {

    private static void install(ClassLoader loader, List<File> additionalClassPathEntries,
                                File optimizedDirectory)
        throws IllegalArgumentException, IllegalAccessException,
        NoSuchFieldException, InvocationTargetException, NoSuchMethodException, IOException {
        /* The patched class loader is expected to be a descendant of
         * dalvik.system.BaseDexClassLoader. We modify its
         * dalvik.system.DexPathList pathList field to append additional DEX
         * file entries.
         */
        Field pathListField = ShareReflectUtil.findField(loader, "pathList");
        Object dexPathList = pathListField.get(loader);
        ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
        ShareReflectUtil.expandFieldArray(dexPathList, "dexElements", makePathElements(dexPathList,
            new ArrayList<File>(additionalClassPathEntries), optimizedDirectory,
            suppressedExceptions));
        if (suppressedExceptions.size() > 0) {
            for (IOException e : suppressedExceptions) {
                Log.w(TAG, "Exception in makePathElement", e);
                throw e;
            }

        }
    }
```

之后深入阅读 ShareReflectUtil 中的源码，ShareReflectUtil 在 shareutil 文件夹下，从源码可以发现，Tinker 本质仍然是用 dexElements 中位置靠前的 Dex 优先加载类来实现热修复，只是，Tinker 比前一种方法的改进在于，直接替换 dex，这样，避免了第一次启动耗时长的问题。

```
 * Replace the value of a field containing a non null array, by a new array containing the
 * elements of the original array plus the elements of extraElements.
 *
 * @param instance      the instance whose field is to be modified.
 * @param fieldName     the field to modify.
 * @param extraElements elements to append at the end of the array.
 */
public static void expandFieldArray(Object instance, String fieldName, Object[] extraElements)
    throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException {
    Field jlrField = findField(instance, fieldName);

    Object[] original = (Object[]) jlrField.get(instance);
    Object[] combined = (Object[]) Array.newInstance(original.getClass().getComponentType(), original.length + extraElements.
        length);

    // NOTE: changed to copy extraElements first, for patch load first

    System.arraycopy(extraElements, 0, combined, 0, extraElements.length);
    System.arraycopy(original, 0, combined, extraElements.length, original.length);

    jlrField.set(instance, combined);
}
```

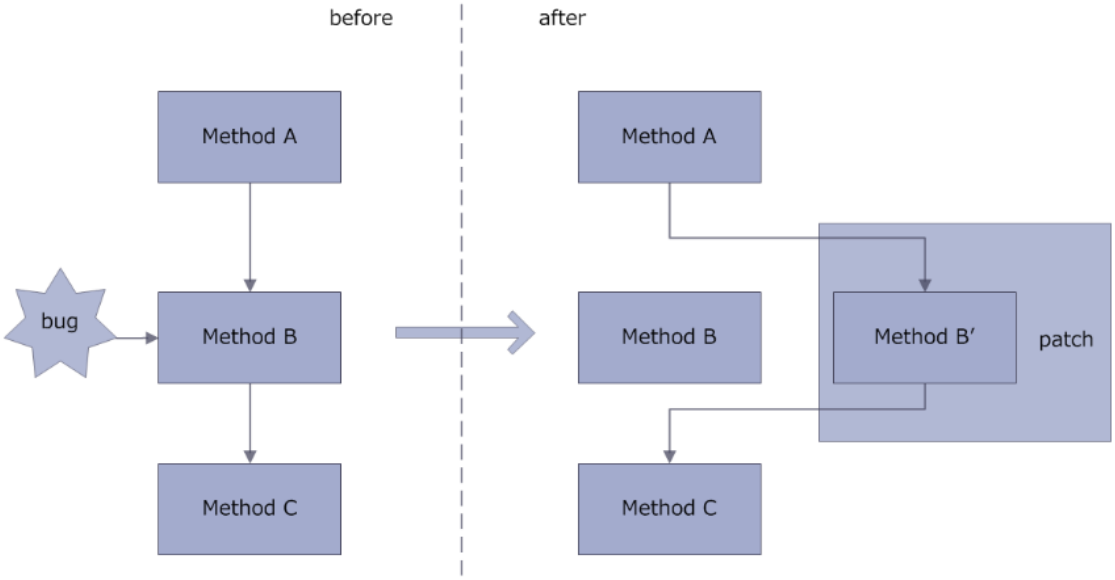
4 阿里百川 HotFix

4.1 原理

阿里百川推出的热修复 HotFix 服务，相对于 QQ 空间超级补丁技术和微信 Tinker 来说，定位于紧急 bug 修复的场景下，能够最及时的修复 bug，下拉补丁立即生效无需等待。该热修复是基于 AndFix 技术的基础上，在产品易用性、安全性方面做了比较深度的优化，因此，除了具有 AndFix 强大的修复能力以外，还提供了 patch 包加密校验、App 安全校验、版本控制管理、加载修复的安全性等等的很多的服务。

阿里百川的热修复技术的有点在于 BUG 修复的即时性，同时，补丁包同样采用差量技术，生成的 PATCH 体积小，而且对应用无侵入，几乎无性能损耗。

android 系统架构从上层到底层共包括四层，分别是应用程序程序层、应用框架层、系统库和 android 运行时和 Linux 内核，其中 Native 层这部分常见一些本地服务和一些链接库等，是运行时的一部分，这一层的一个特点就是通过 C 和 C++语言实现。阿里百川 HotFix 修复技术的原理是在 Native 层改变指针指向，完成方法重定向，从而实现方法的替换。如下图所示。



AndFix 实现方法的替换，需要在 Native 层进行操作，操作的流程如下：

第一步 打开链接库得到操作句柄；获取到 native 层的内部函数，然后得到

ClassObject 对象。

第二步 把访问的权限属性设置为公开 public。

第三步 获取到新旧方法的指针，然后新方法指向目标方法，从而实现方法的替换。

4.2 实践

接下来以 Dalvik 设备为例，完成上面说的三个流程。[5]

第一步 ClassObject 对象的获取，在 delvik_setup 函数中，首先用 dvmThreadSelf( ) 查询当前的线程，然后用 dvmDecodeIndirectRef()来根据当前线程获得 ClassObject 对象，完成第一步。

```
extern jboolean __attribute__ ((visibility ("hidden"))) dalvik_setup(
        JNIEnv* env, int apilevel) {
    //Davik虚拟机实现 是在libdvm.so中
    //dlopen()方法以指定模式打开动态链接库，RTLD_NOW立即打开
    void* dvm_hand = dlopen("libdvm.so", RTLD_NOW);
    if (dvm_hand) {
        //dlsym:通过句柄和连接符名称获取内部函数
        dvmDecodeIndirectRef_fnPtr = dvm_dlsym(dvm_hand,
                apilevel > 10 ?
                        "_Z20dvmDecodeIndirectRefP6ThreadP8_jobject" :
                        "dvmDecodeIndirectRef");
        if (!dvmDecodeIndirectRef_fnPtr) {
            return JNI_FALSE;
        }
        dvmThreadSelf_fnPtr = dvm_dlsym(dvm_hand,
                apilevel > 10 ? "_Z13dvmThreadSelfv" : "dvmThreadSelf");
        if (!dvmThreadSelf_fnPtr) {
            return JNI_FALSE;
        }
        //
        jclass clazz = env->FindClass("java/lang/reflect/Method");
        jClassMethod = env->GetMethodID(clazz, "getDeclaringClass",
                        "()Ljava/lang/Class;");

        return JNI_TRUE;
    } else {
        return JNI_FALSE;
    }
}
```

第二步 修改访问权限。由于 private、protected 的方法和字段不可以被动态库看见和识别，因此，必须修改为 public 才能被使用。

```
extern void dalvik_setFieldFlag(JNIEnv* env, jobject field) {
    Field* dalvikField = (Field*) env->FromReflectedField(field);
    //修改访问权限为public
    dalvikField->accessFlags = dalvikField->accessFlags & (~ACC_PRIVATE)
            | ACC_PUBLIC;
    LOGD("dalvik_setFieldFlag: %d ", dalvikField->accessFlags);
}
```

第三步 方法替换。实确切地说是方法指针指向的改变。替换的流程为：Dalvik 设备、

Native 层找到被替换的类、将类状态设置为初始化、得到新旧方法的指针、操作指针让指针指向新的方法、完成方法的替换。

```
extern void __attribute__ ((visibility ("hidden"))) dalvik_replaceMethod(
        JNIEnv* env, jobject src, jobject dest) {
    //clazz为被替换的类
    jobject clazz = env->CallObjectMethod(dest, jClassMethod);
    //clz 为被替换的类对象
    ClassObject* clz = (ClassObject*) dvmDecodeIndirectRef_fnPtr(
            dvmThreadSelf_fnPtr(), clazz);
    //将类状态设置为初始化完毕
    clz->status = CLASS_INITIALIZED;
    //得到指向新方法的指针
    Method* meth = (Method*) env->FromReflectedMethod(src);
    //得到指向需要修复的目标方法的指针
    Method* target = (Method*) env->FromReflectedMethod(dest);

    //新方法指向目标方法，实现方法的替换
    meth->clazz = target->clazz;
    //访问权限属性public
    meth->accessFlags |= ACC_PUBLIC;
    meth->methodIndex = target->methodIndex;
    meth->jniArgInfo = target->jniArgInfo;
    meth->registersSize = target->registersSize;
    meth->outsSize = target->outsSize;
    meth->insSize = target->insSize;
    meth->prototype = target->prototype;
    //指针指向新的替换方法
    meth->insns = target->insns;
    meth->nativeFunc = target->nativeFunc;
}
```

阿里百川的热修复功能在 11 月已经上线，具体使用步骤如下。

（1）申请成为阿里百川开发者，并在后台中新建应用。

（2）新建项目工程，下载阿里百川 SDK，并把 so 包放到工程的 lib 目录下，然后加入 so 依赖语句。然后，同时，修改 gradle 配置文件。加入依赖包语句：

compile 'com.taobao.android:alisdk-hotfix:1.4.0'

同时，阿里百川提供 maven 管理，因此，还要在 gradle 中加入 maven 仓库管理：

url "http://repo.baichuan-android.taobao.com/content/groups/BaichuanRepositories"

（3）APP 所需权限。阿里百川热修复需要用到网络权限和外设读取权限，因此需要加入以下权限管理。Android 6.0 版本需要进行权限适配。

<!-- 网络权限 -->

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />

<!-- 外部存储读权限 -->

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>

（4）配置 AndroidManifest 文件。方法是在 application 节点下加入节点，内容如下，其中 your-app-secret 和 your-rsa-secret 的值均能在阿里百川后台应用管理查到对应的值。

```
<meta-data
    android:name="com.taobao.android.hotfix.APPSECRET"
    android:value="your-app-secret" />
<meta-data
    android:name="com.taobao.android.hotfix.RSASECRET"
    android:value="your-rsa-secret" />
```

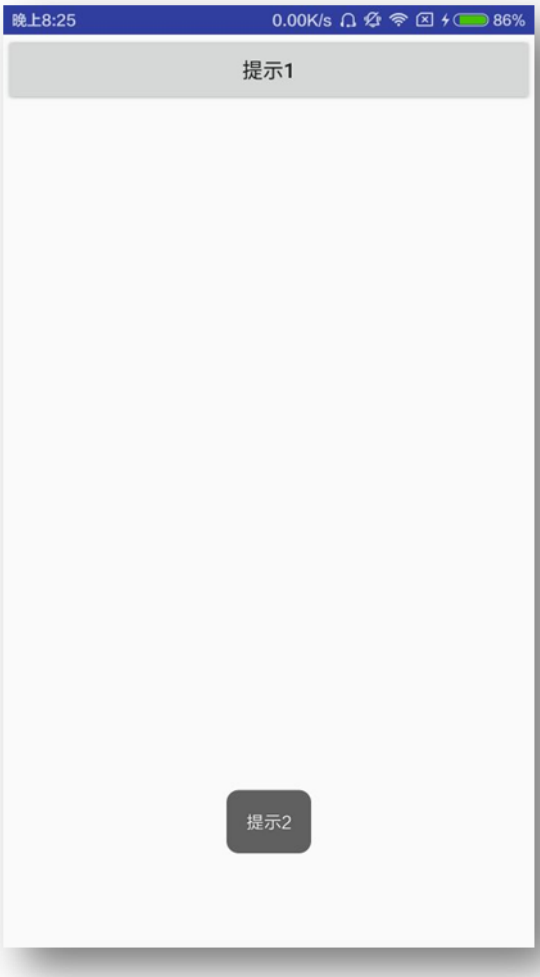
（5）根据官网的文档，initialize 方法必须越早实现越好，所以，新建一个自己的 Application，在 Application 中实现 initialize 最好，同时 Application 中还需要实现 initApp 方法。如下所示。initApp 中的 appId 要换成自己的 id，可以从阿里百川网站的后台管理找到对应的值。Initialize 方法主要是获取 HotFixManager，这是一个单例模式的类，调用 HotFixManager.getInstance()方法获得。

```
private void initApp() {
    this.appId = "84715-1"; //替换掉自己应用的appId
    try {
        this.appVersion = this.getPackageManager().getPackageInfo(this.getPackageName(), 0).versionName;
    } catch (Exception e) {
        this.appVersion = "1.0.0";
    }
}
```

```
private void initHotfix() {
    HotFixManager.getInstance().setContext(this.getApplication())
            .setAppVersion(MainApplication.appVersion)
            .setAppId(MainApplication.appId)
            .setAesKey(null)
            .setSupportHotpatch(true)
            .setEnableDebug(true)
            .setPatchLoadStatusStub(new PatchLoadStatusListener() {
                @Override
                public void onload(final int mode, final int code, final String info, final int handlePatchVersion) {
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            StringBuilder stringBuilder = new StringBuilder();
                            stringBuilder.append("Mode:").append(mode).append(" Code:").append(code).append(" Info:").append(
                                info).append(" HandlePatchVersion:").append(handlePatchVersion);
                            updateConsole(stringBuilder.toString());
                        }
                    });
                }
            }).initialize();

    if (Build.VERSION.SDK_INT >= 23) {
        requestExternalStoragePermission();
    }
}
```

（6）编写自己的代码部分，然后打包成 release 版本，安装。例子和 QQ 空间补丁的项目一样，有 bug 的代码输出了“提示 2”。如下图所示。



接下来就是 Bug 修复过程。

（7）首先把老版本的 APK 保存下来，并且重新命名为“old.apk”，修复 bug，然后重新打包一个 apk，并且重新命名为“new.apk”。

（8）到官网下载补丁生成工具 BCFixPatchTools-1.3.0.jar。把 BCFixPatchTools-1.3.0.jar、old.apk、new.apk 放到同一个文件夹下。把打包用的密钥也放到该文件夹下面。

（9）生成补丁包，在上面的文件夹下面使用 cmd 命令，各个参数的说明如下：

-c, -cmd: 值为 patch: 打补丁命令，默认值为 help 时查看使用说明。

-s, -src_apk：填写本地的原始 APK（即有存在 bug 的 APK），此项必选。

-f, -fixed_apk：已经完成 bug 修复的 APK，此项必选。

-w, -wp：输出 patch 的路径，最后，如果打补丁成功了会在 wp 目录下自动创建的 hotfix-working 目录并且生成 baichuan-hotfix-patch.jar 补丁文件，此项必选。

-k, -sign_file_url：本地的签名文件的路径，不输入就表示不做签名，此项可选。

-p, -sign_file_pass: 证书文件的密码，此项可选。

-a, -sign_alias: 证书的别名，此项可选。

-e, -sign_alias_pass: 证书别名的密码，此项可选。

-y, -aes_key: 自定义 aes 秘钥，必须是 16 位，此项可选。

-l, -filterClassFilePath：本地的白名单类列表文件的路径，放进去的类将不会再被计算。

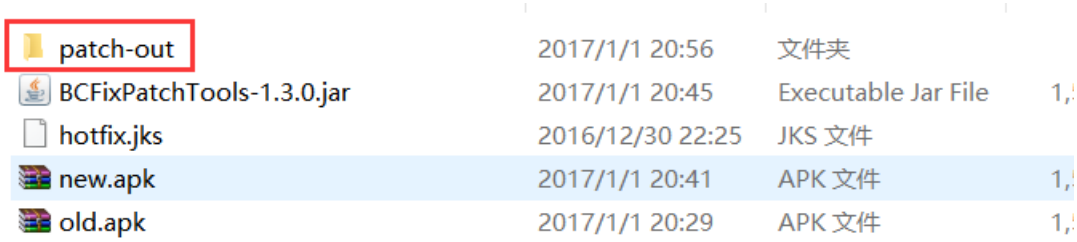
如 java -jar BCFixPatchTools-1.3.0.jar -c patch -s old.apk -f new.apk -w patch-out -k hotfix.jks -p android -a hotfix -e android

执行之后会显示生成成功。成功之后可以在文件夹下面看见一个 patch-out 即是补丁包，里面有 jar 的补丁文件。

```
java –jar BCFixPatchTools-1.3.0.jar –c patch –s old.apk –f new.apk –w patch-out
–k hotfix.jks –p android –a hotfix –e android
```

```
C:\Users\陈明\Desktop\tool>java -jar BCFixPatchTools-1.3.0.jar -c patch -s old.apk -f new.apk -w patch-out -k hotfix.jks
 -p android -a hotfix -e android
start to build alibaba baichuan hotfix patch, tool verison: 1.3.0
add modified method:V  onClick(Landroid/view/View;)  in Class:Lcom/chenming/alihotfix/MainActivity$1;
add modified class: com.chenming.alihotfix.MainActivity$1
add used class:Lcom/chenming/alihotfix/MainActivity$1;
add used method:Landroid/widget/Toast;->makeText:[Landroid/content/Context;|Ljava/lang/CharSequence;|I] Landroid/widget/
Toast;
add used method:Landroid/widget/Toast;->show:[] V
build prepare class template: Lcom/chenming/alihotfix/MainActivity$1;->onClick
start sign patch file
dopatch success, final patch file is: C:\Users\陈明\Desktop\tool\patch-out\hotfix-working\baichuan-hotfix-patch.jar
```

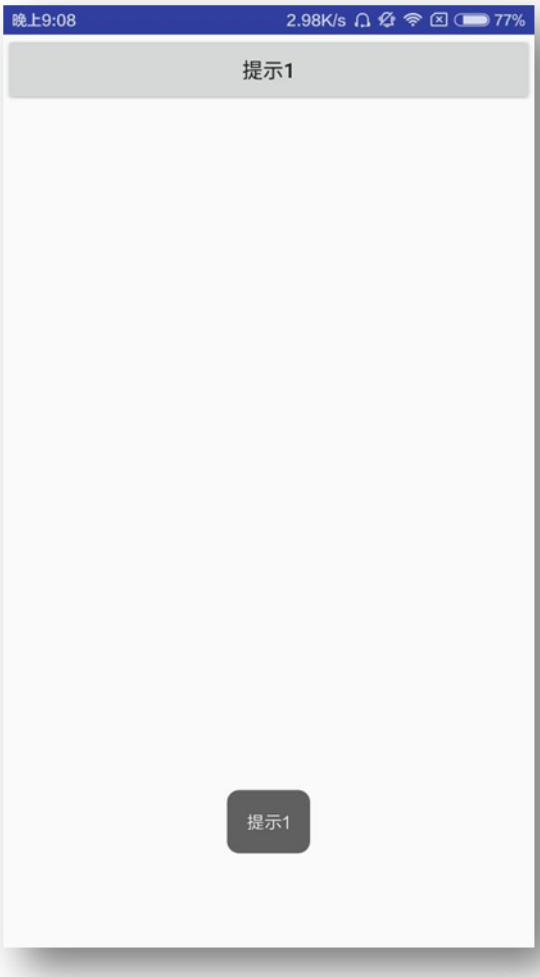
其中，old.apk 和 new.apk 分别是有 bug 的安装包和修复完 bug 的安装包，hotfix.jks 是发布 release 版本所用的密钥文件。



（10）上传补丁包。登陆阿里百川后台应用管理，把刚才生成的 patch-out 文件夹最里面的 baichuan-hotfix-patch.jar 上传上去。

点进查看详情，可以选择全量发布和灰度发布，两者的区别在于全量发布是所有人都发布，而灰度发布只能部分人收到，一般直接选择全量发布，勾选之后点击确认发布按钮，即可看见发布成功，表示补丁包已经发送。

（11）再次点击手机上的“提示 1”按钮，可以发现已经变成了“提示 1”了。可见 bug 修复完成。



5 几种热修复方案的对比

以上三种比较流行的热修复方案原理各不相同，每种热修复都有自己的优缺点。QQ空间超级补丁技术是基于dex分包方案，把bug方法修复以后，放到一个单独的dex里，插入到dexElements数组的最前面，让虚拟机去加载修复完后的方法。这样可以实现类的替代，兼容性高。但是，这么做也带来了一些缺点，比如不能即时生效，必须重启才能生效，而且当修复的类的数量到达一定量的时候，启动时会比较耗时。

微信Tinker的主要原理与QQ空间超级补丁技术有点类似，主要的区别在于不再将patch.dex增加到elements数组中，而是差量的方式给出patch.dex，然后将patch.dex与应用的classes.dex合并，然后整体替换掉旧的dex文件，以达到修复的目的。这样做的优点在于合成整包，性能得到了提高，但是也和QQ空间超级补丁技术存在一样的缺点，不能即时生效，而且需要开启新进程才能合并，容易因为内存不足而导致合并失败。

不同于QQ空间超级补丁和微信Tinker通过改变dex的方案，阿里百川的HotFix提供了一种运行于Native修改Filed指针的方式，来实现方法的替换，从而修复bug。这么做的优点在于修复的bug能即时生效，补丁采用了差量的技术，生成的补丁包体积较小，几乎没有什么性能损耗。缺点在于由于1. 由于厂商的自定义ROM，对少数机型暂不支持。

三种修复方案各有优缺点，所以，具体项目中选用哪种作为热修复的方式还需要根据具体项目而定。

参考文献

[1] QQ 空间终端开发团队.安卓 App 热补丁动态修复技术介绍[2015-10-31].

http://mp.weixin.qq.com/s/xuvHomyTzTA90IEWDrdwgw

[2] 何红辉、关爱民.Android 源码设计模式解析与实战[2015-11]

[3] Bugly.微信 Android 热补丁实践演进之路[2016-6]

http://bugly.qq.com/bbs/forum.php?mod=viewthread&tid=1264

[4] Tencent.tinker 开源代码库[2016-9-18]

https://github.com/Tencent/tinker

[5] 阿里百川.阿里百川 HotFix 接入流程[2016-11]

http://baichuan.taobao.com/docs/doc.htm?spm=a3c0d.7629140.0.0.lLVcPJ&treeId=234&articleId=105456&docType=1