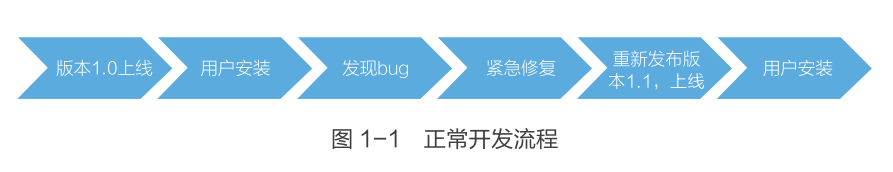


Android热修复(热更新)调研

**中国移动互联网基地融合通信项目组**

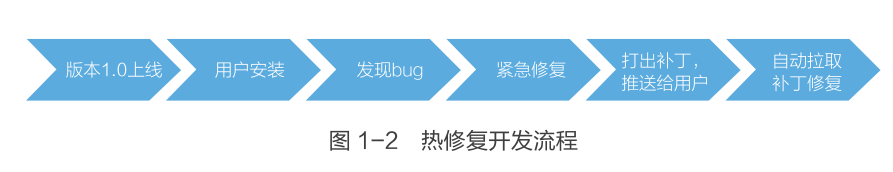
**2018年01月29日**

# 技术背景



从流程来看，传统的开发流程存在很多弊端：

* 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



而热修复的开发流程显得更加灵活，优势很多：

* 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

# 热修复技术发展

更新的类或者插件粒度较小的时候，我们会称之为热修复，一般用于修复bug！比如更新一个bug方法或者紧急修改lib包，甚至一个类等。2016年 Google 的 Android Studio 推出了Instant Run 功能 同时提出了3个名词；

1. “**热部署**” – 方法内的简单修改，无需重启app和Activity。
2. “**暖部署**” – app无需重启，但是activity需要重启，比如资源的修改。
3. “**冷部署**” – app需要重启，比如继承关系的改变或方法的签名变化等。

热更新方案发展至今，有很多团队开发过不同的解决方案，包括**Dexposed**、**AndFix**，(**HotFix**)**Sophix**，**Qzone**超级补丁的类**Nuwa**方式，微信的**Tinker**, 大众点评的**nuwa**、百度金融的**rocooFix**, 饿了么的**amigo**以及美团的**robust**、腾讯的**Bugly**热更新(注：苹果公司现在已经禁止了热更新)。



**热修复开源方案和使用情况**

要弄清热修复技术的原理，就要先弄清Android的ClassLoader机制，相关文章可以阅读[Android ClassLoader详解](http://blog.csdn.net/xiangzhihong8/article/details/52880327?spm=5176.100239.blogcont184258.19.LQkbFH)，Android的ClassLoader分为PathClassLoader和DexClassLoader，它们都都继承自BaseDexClassLoader，其中PathClassLoader用来加载系统类和应用类；DexClassLoader用来加载jar、apk、dex文件。

**热修复就是要做一件事--替换。当替换的东西属于大块内容的时候，就是模块化了，当你去替换方法的时候，叫热修复，当你替换类的时候，叫热插件**，而且从某种意义上讲，所有的热修复方案，都是一种热插件，因为热修复方案就是在app之外去干这个事。就这么简单的理解。无论是替换一个类，还是一个方法，都是在干替换这件事请。这里的替换，也算是几种hook操作，无论在什么代码等级上，都是一种侵入性的操作。

## Dexposed & AndFix & (HotFix)SopHix --阿里热修复方案

1. **对Dexposed(阿里系热修复方案一)**

"Dexposed" 是大厂阿里以前的一个开源热修复项目，基于 Xposed "Xposed"的AOP框架，方法级粒度，可以进行AOP编程、插桩、热补丁、SDK hook等功能。

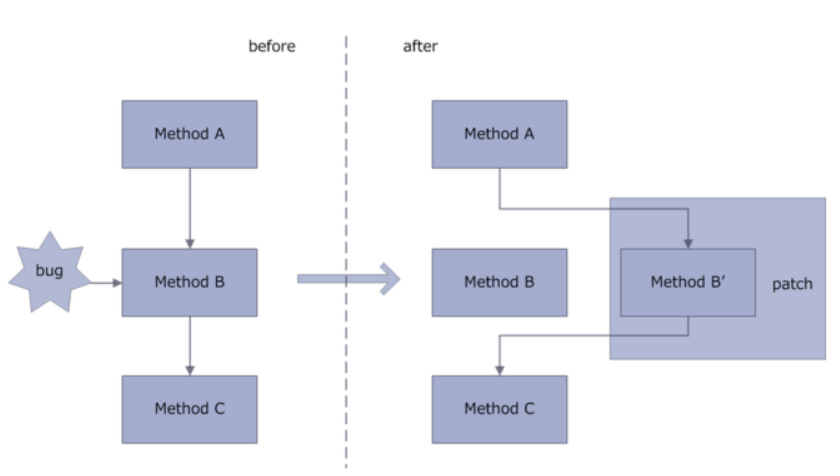
但Xposed 有一个缺陷就是需要 root ，而 Dexposed 就是一个不需要 root 权限的 hook 框架。以前阿里的主流 app ，例如手机淘宝，支付宝，天猫都使用了 Dexposed 支持在线热更新，现在已经不用了，用最新的 Sophix 了。

Dexposed 中的 AOP 原理来自于 Xposed。在 Dalvik 虚拟机下，主要是通过改变一个方法对象方法在 Dalvik 虚拟机中的定义来实现，具体做法就是将该方法的类型改变为 native 并且将这个方法的实现链接到一个通用的 Native Dispatch 方法上。这个 Dispatch 方法通过 JNI 回调到 Java 端的一个统一处理方法，最后在统一处理方法中调用 before、after 函数来实现AOP。在 Art 虚拟机上目前也是是通过改变一个 ArtMethod 的入口函数来实现。

可惜 android 4.4之后的版本都用 Art 取代了 Dalvik ,所以要 hook Android4.4 以后的版本就必须去适配 Art 虚拟机的机制。于是阿里已经抛弃了 Dexposed，原因很明显，4.4 以后不支持了。

1. **AndFix （阿里热修复方案二）**

AndFix 是一个 Android App 的在线热补丁框架。使用此框架，我们能够在不重复发版的情况下，在线修改 App 中的 Bug 。AndFix 就是 “Android Hot-Fix”的缩写，支持 Android 2.3到6.0版本，并且支持 arm 与 X86 系统架构的设备。完美支持 Dalvik 与 ART 的 Runtime。AndFix 的补丁文件是以 .apatch 结尾的文件。它从服务器分发到客户端来修复你 App 的 bug 。



**AndFix 更新实现过程**

 首先添加依赖

`compile 'com.alipay.euler:andfix:0.3.1@aar'`

 然后在 Application.onCreate() 中添加以下代码

`patchManager = new PatchManager(context);``patchManager.init(appversion);//current version``patchManager.loadPatch();`

 可以用这句话获取 appversion,每次 appversion 变更都会导致所有补丁被删除,如果 appversion 没有改变，则会加载已经保存的所有补丁。

`String appversion= getPackageManager().getPackageInfo(getPackageName(), 0).versionName;`

然后在需要的地方调用 PatchManager 的 addPatch 方法加载新补丁，比如可以在下载补丁文件之后调用。

之后就是打补丁的过程了，首先生成一个 apk 文件，然后更改代码，在修复 bug 后生成另一个 apk。通过官方提供的工具 apkpatch 生成一个 .apatch 格式的补丁文件，需要提供原 apk，修复后的 apk，以及一个签名文件。

通过网络传输或者 adb push 的方式将 apatch 文件传到手机上，然后运行到 addPatch 的时候就会加载补丁。

**AndFix 更新的原理：**

1. 首先通过虚拟机的 JarFile 加载补丁文件，然后读取 PATCH.MF 文件得到补丁类的名称
2. 使用 DexFile 读取 patch 文件中的 dex 文件，得到后根据注解来获取补丁方法，然后根据注解中得到类名和方法名，使用 classLoader 获取到 Class，然后根据反射得到 bug 方法。
3. jni 层使用 C++ 的指针替换 bug 方法对象的属性来修复 bug。

具体的实现主要都是我们在 Application 中初始化的**PatchManager**中。

public PatchManager(Context context) {

   mContext = context;

   mAndFixManager = new AndFixManager(mContext);*//初始化AndFixManager*

   mPatchDir = new File(mContext.getFilesDir(), DIR);*//初始化存放patch补丁文件的文件夹*

   mPatchs = new ConcurrentSkipListSet<Patch>();*//初始化存在Patch类的集合,此类适合大并发*

mLoaders = new ConcurrentHashMap<String, ClassLoader>();*//初始化存放类对应的类加载器集合*

}

其中**mAndFixManager = new AndFixManager(mContext);**的实现：

public AndFixManager(Context context) {

   mContext = context;

   mSupport = Compat.isSupport();*//判断Android机型是否适支持AndFix*

   if (mSupport) {

       mSecurityChecker = new SecurityChecker(mContext);*//初始化签名判断类*

       mOptDir = new File(mContext.getFilesDir(), DIR);*//初始化patch文件存放的文件夹*

       if (!mOptDir.exists() && !mOptDir.mkdirs()) {*// make directory fail*

           mSupport = false;

           Log.e(TAG, "opt dir create error.");

       } else if (!mOptDir.isDirectory()) {*// not directory*

           mOptDir.delete();*//如果不是文件目录就删除*

           mSupport = false;

       }

   }

}

然后是对版本的初始化**mPatchManager.init(appversion)**，**init(String appVersion)**

public void init(String appVersion) {      
    if (!mPatchDir.exists() && !mPatchDir.mkdirs()) {*// make directory fail*

       Log.e(TAG, "patch dir create error.");          
        return;

   } else if (!mPatchDir.isDirectory()) {*// not directory*

       mPatchDir.delete();          
        return;

   }

   SharedPreferences sp = mContext.getSharedPreferences(SP\_NAME,

           Context.MODE\_PRIVATE);*//存储关于patch文件的信息*

*//根据你传入的版本号和之前的对比，做不同的处理*

   String ver = sp.getString(SP\_VERSION, null);      
    if (ver == null || !ver.equalsIgnoreCase(appVersion)) {

       cleanPatch();*//删除本地patch文件*

       sp.edit().putString(SP\_VERSION, appVersion).commit();*//并把传入的版本号保存*

   } else {

       initPatchs();*//初始化patch列表，把本地的patch文件加载到内存*

   }

}

init 初始化主要是对 patch 补丁文件信息进行保存或者删除以及加载。

那么 patch 补丁文件是如何加载的呢？其实 patch 补丁文件本质上是一个 jar 包，使用 JarFile 来读取即可：

public Patch(File file) throws IOException {

   mFile = file;

   init();

}@SuppressWarnings("deprecation")private void init() throws IOException {

   JarFile jarFile = null;

   InputStream inputStream = null;      
    try {

       jarFile = new JarFile(mFile);*//使用JarFile读取Patch文件*

       JarEntry entry = jarFile.getJarEntry(ENTRY\_NAME);*//获取META-INF/PATCH.MF文件*

       inputStream = jarFile.getInputStream(entry);

       Manifest manifest = new Manifest(inputStream);

       Attributes main = manifest.getMainAttributes();

       mName = main.getValue(PATCH\_NAME);*//获取PATCH.MF属性Patch-Name*

       mTime = new Date(main.getValue(CREATED\_TIME));*//获取PATCH.MF属性Created-Time*

       mClassesMap = new HashMap<String, List<String>>();

       Attributes.Name attrName;

       String name;

       List<String> strings;          
    for (Iterator<?> it = main.keySet().iterator(); it.hasNext();) {

           attrName = (Attributes.Name) it.next();

           name = attrName.toString();              
            *//判断name的后缀是否是-Classes，并把name对应的值加入到集合中，对应的值就是class类名的列表*

           if (name.endsWith(CLASSES)) {

               strings = Arrays.asList(main.getValue(attrName).split(","));                  
    if (name.equalsIgnoreCase(PATCH\_CLASSES)) {

                   mClassesMap.put(mName, strings);

               } else {

                   mClassesMap.put(

                           name.trim().substring(0, name.length() - 8),*// remove*

*// "-Classes"*

                           strings);

               }

           }

       }

   } finally {        if (jarFile != null) {

           jarFile.close();

       }        if (inputStream != null) {

           inputStream.close();

       }

   }

}

然后就是最重要的**patchManager.loadPatch()**：

public void loadPatch() {

   mLoaders.put("\*", mContext.getClassLoader());*// wildcard*

   Set<String> patchNames;

   List<String> classes;      
    for (Patch patch : mPatchs) {

       patchNames = patch.getPatchNames();          
    for (String patchName : patchNames) {

           classes = patch.getClasses(patchName);*//获取patch对应的class类的集合List*

           mAndFixManager.fix(patch.getFile(), mContext.getClassLoader(),

                   classes);*//修复bug方法*

       }

   }

}

从上面的 bug 修复源码可以看出，就是在找补丁包中有 @MethodReplace 注解的方法，然后反射获取原 apk 中方法的位置，最后进行替换。

而最后调用的 replaceMethod(Method dest,Method src) 则是 native 方法，源码中有两个 replaceMethod：

extern void dalvik\_replaceMethod(JNIEnv\* env, jobject src, jobject dest);*//Dalvik*extern void art\_replaceMethod(JNIEnv\* env, jobject src, jobject dest);*//Art*

从源码的注释也能看出来，因为安卓 4.4 版本之后使用的不再是 Dalvik 虚拟机，而是 Art 虚拟机，所以需要对不同的手机系统做不同的处理。

首先看 Dalvik 替换方法的实现：

extern void \_\_attribute\_\_ ((visibility ("hidden"))) dalvik\_replaceMethod(

   JNIEnv\* env, jobject src, jobject dest) {

   jobject clazz = env->CallObjectMethod(dest, jClassMethod);

   ClassObject\* clz = (ClassObject\*) dvmDecodeIndirectRef\_fnPtr(

       dvmThreadSelf\_fnPtr(), clazz);

   clz->status = CLASS\_INITIALIZED;

   Method\* meth = (Method\*) env->FromReflectedMethod(src);

   Method\* target = (Method\*) env->FromReflectedMethod(dest);

   LOGD("dalvikMethod: %s", meth->name);

   meth->jniArgInfo = 0x80000000;

   meth->accessFlags |= ACC\_NATIVE;*//把Method的属性设置成Native方法*

   int argsSize = dvmComputeMethodArgsSize\_fnPtr(meth);      
    if (!dvmIsStaticMethod(meth))

   argsSize++;

   meth->registersSize = meth->insSize = argsSize;

   meth->insns = (void\*) target;

   meth->nativeFunc = dalvik\_dispatcher;*//把方法的实现替换成native方法*

}

Art 替换方法的实现：

*//不同的art系统版本不同处理也不同*extern void \_\_attribute\_\_ ((visibility ("hidden"))) art\_replaceMethod(

       JNIEnv\* env, jobject src, jobject dest) {      
    if (apilevel > 22) {

       replace\_6\_0(env, src, dest);

   } else if (apilevel > 21) {

       replace\_5\_1(env, src, dest);

   } else {

       replace\_5\_0(env, src, dest);

   }

}*//以5.0为例：*void replace\_5\_0(JNIEnv\* env, jobject src, jobject dest) {

   art::mirror::ArtMethod\* smeth =

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

   art::mirror::ArtMethod\* dmeth =

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

   dmeth->declaring\_class\_->class\_loader\_ =

           smeth->declaring\_class\_->class\_loader\_; *//for plugin classloader*

   dmeth->declaring\_class\_->clinit\_thread\_id\_ =

           smeth->declaring\_class\_->clinit\_thread\_id\_;

   dmeth->declaring\_class\_->status\_ = (void \*)((int)smeth->declaring\_class\_->status\_-1);      
    *//把一些参数的指针给补丁方法*

   smeth->declaring\_class\_ = dmeth->declaring\_class\_;

   smeth->access\_flags\_ = dmeth->access\_flags\_;

   smeth->frame\_size\_in\_bytes\_ = dmeth->frame\_size\_in\_bytes\_;

   smeth->dex\_cache\_initialized\_static\_storage\_ =

           dmeth->dex\_cache\_initialized\_static\_storage\_;

   smeth->dex\_cache\_resolved\_types\_ = dmeth->dex\_cache\_resolved\_types\_;

   smeth->dex\_cache\_resolved\_methods\_ = dmeth->dex\_cache\_resolved\_methods\_;

   smeth->vmap\_table\_ = dmeth->vmap\_table\_;

   smeth->core\_spill\_mask\_ = dmeth->core\_spill\_mask\_;

   smeth->fp\_spill\_mask\_ = dmeth->fp\_spill\_mask\_;

   smeth->mapping\_table\_ = dmeth->mapping\_table\_;

   smeth->code\_item\_offset\_ = dmeth->code\_item\_offset\_;

   smeth->entry\_point\_from\_compiled\_code\_ =

           dmeth->entry\_point\_from\_compiled\_code\_;

   smeth->entry\_point\_from\_interpreter\_ = dmeth->entry\_point\_from\_interpreter\_;

   smeth->native\_method\_ = dmeth->native\_method\_;*//把补丁方法替换掉*

   smeth->method\_index\_ = dmeth->method\_index\_;

   smeth->method\_dex\_index\_ = dmeth->method\_dex\_index\_;

   LOGD("replace\_5\_0: %d , %d", smeth->entry\_point\_from\_compiled\_code\_,

           dmeth->entry\_point\_from\_compiled\_code\_);

}

其实这个替换过程可以看做三步完成

1.打开链接库得到操作句柄，获取 native 层的内部函数，得到 ClassObject 对象

2.修改访问权限的属性为 public

3.得到新旧方法的指针，新方法指向目标方法，实现方法的替换。

如果我们想知道补丁包中到底替换了哪些方法，可以直接方便易 patch 文件，然后看到的所有含有 @ReplaceMethod 注解的方法基本上就都是需要替换的方法了。

好的，现在 AndFix 我们分析了一遍它的实现过程和原理，**其优点是不需要重启即可应用补丁，遗憾的是它还是有不少缺陷的，这直接导致阿里再次抛弃了它，缺陷如下：**

1. 并不能支持所有的方法修复



1. 不支持 YunOS
2. 无法添加新类和新的字段
3. 需要使用加固前的 apk 制作补丁，但是补丁文件很容易被反编译，也就是修改过的类源码容易泄露
4. 使用加固平台可能会使热补丁功能失效（看到有人在 360 加固提了这个问题，自己还未验证）
5. **Sophix （HotFix升级版阿里系热修复终极版）**





注：横向比较即与阿里系历史产品作比较，纵向比价即与同类竞品作比较。



阿里系的热修复框架发展史：Dexposed-->AndFix-->HotFix-->Sophix

总结：可以说在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，具体可以参考[Sophix](https://www.aliyun.com/product/hotfix?spm=a2c4e.11153959.blogcont103527.10.lfmb6e)。

业界首部安卓热修复原理书籍——《深入探索Android热修复技术原理》

本书下载链接：http://pan.baidu.com/s/1sl2l7Bn 密码：70js



## Android热修复技术——Tinker方案解析

1. **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：

* native--底层替换方案限制颇多，但时效性最好，加载轻快，立即见效。
* java--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。

1. **资源修复**

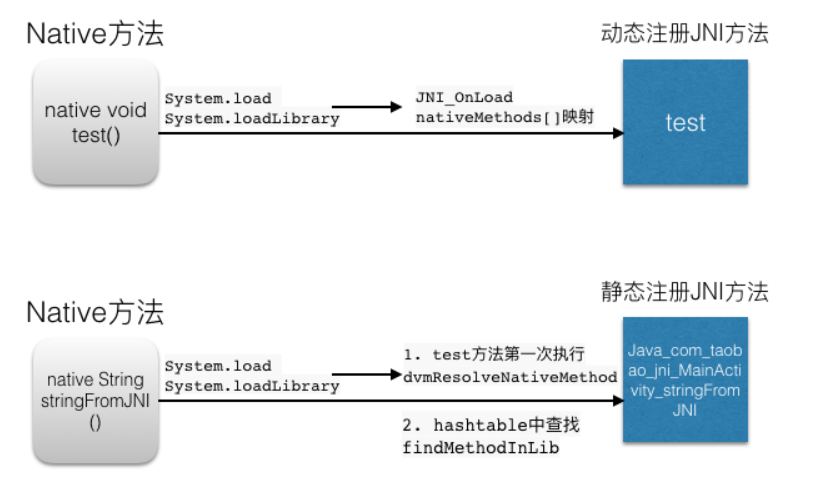
目前市面上的很多资源热修复方案基本上都是参考了Instant Run的实现。实际上，Instant Run的推出正是推动这次热修复浪潮的主因，各家热修复方案，在代码、资源等方面的实现，很大程度上地参考了Instant Run的代码，而资源修复方案正是被拿来用到最多的地方。

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。
3. **so库修复**

so库的修复本质上是对native方法的修复和替换。

我们知道JNI编程中，native方法可以通过动态注册和静态注册两种方式进行。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。



热修复主要应用于以上三个区域，再来看看腾讯的Tinker方案。

1. Dex分包机制

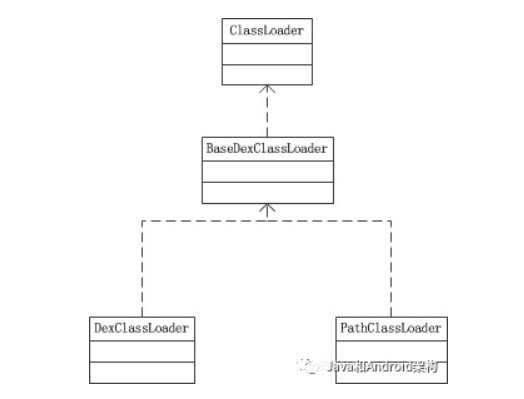
我们开发的代码在被编译成class文件后会被打包成一个dex文件。但是dex文件有一个限制，由于方法id是一个short类型，所以导致了一个dex文件最多只能存放65536个方法。随着现今App的开发日益复杂，导致方法数早已超过了这个上限。为了解决这个问题，Google提出了multidex方案，即一个apk文件可以包含多个dex文件。

不过值得注意的是，除了第一个dex文件以外，其他的dex文件都是以资源的形式被加载的，换句话说就是在Application.onCreate()方法中被注入到系统的ClassLoader中的。这也就为热修复提供了一种可能：将修复后的代码打成补丁包，然后发送到客户端，客户端在启动的时候到指定路径下加载对应dex文件即可。

根据Android虚拟机的类加载机制，同一个类只会被加载一次，所以要让修复后的类替换原有的类就必须让补丁包的类被优先加载。接下来看下Android虚拟机的类加载机制。

**2. 类加载机制**

Android的类加载机制和jvm加载机制类似，都是通过ClassLoader来完成，只是具体的类不同而已：



Android系统通过PathClassLoader来加载系统类和主dex中的类。而DexClassLoader则用于加载其他dex文件中的类。上述两个类都是继承自BaseDexClassLoader，具体的加载方法是findClass:

public class BaseDexClassLoader extends ClassLoader {

private final DexPathList pathList;

/\*\*

\* Constructs an instance.

\*

\* @param dexPath the list of jar/apk files containing classes and

\* resources, delimited by {@code File.pathSeparator}, which

\* defaults to {@code ":"} on Android

\* @param optimizedDirectory directory where optimized dex files

\* should be written; may be {@code null}

\* @param libraryPath the list of directories containing native

\* libraries, delimited by {@code File.pathSeparator}; may be

\* {@code null}

\* @param parent the parent class loader

\*/

public BaseDexClassLoader(String dexPath, File optimizedDirectory,

String libraryPath, ClassLoader parent) {

super(parent);

this.pathList = new DexPathList(this, dexPath, libraryPath, optimizedDirectory);

}

@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;

}

}

从代码中可以看到加载类的工作转移到了pathList中，pathList是一个DexPathList类型，从变量名和类型名就可以看出这是一个维护Dex的容器：

/\*package\*/ final class DexPathList {

private static final String DEX\_SUFFIX = ".dex";

private static final String JAR\_SUFFIX = ".jar";

private static final String ZIP\_SUFFIX = ".zip";

private static final String APK\_SUFFIX = ".apk";

/\*\* class definition context \*/

private final ClassLoader definingContext;

/\*\*

\* List of dex/resource (class path) elements.

\* Should be called pathElements, but the Facebook app uses reflection

\* to modify 'dexElements' (http://b/7726934).

\*/

private final Element[] dexElements;

/\*\*

\* Finds the named class in one of the dex files pointed at by

\* this instance. This will find the one in the earliest listed

\* path element. If the class is found but has not yet been

\* defined, then this method will define it in the defining

\* context that this instance was constructed with.

\*

\* @param name of class to find

\* @param suppressed exceptions encountered whilst finding the class

\* @return the named class or {@code null} if the class is not

\* found in any of the dex files

\*/

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));

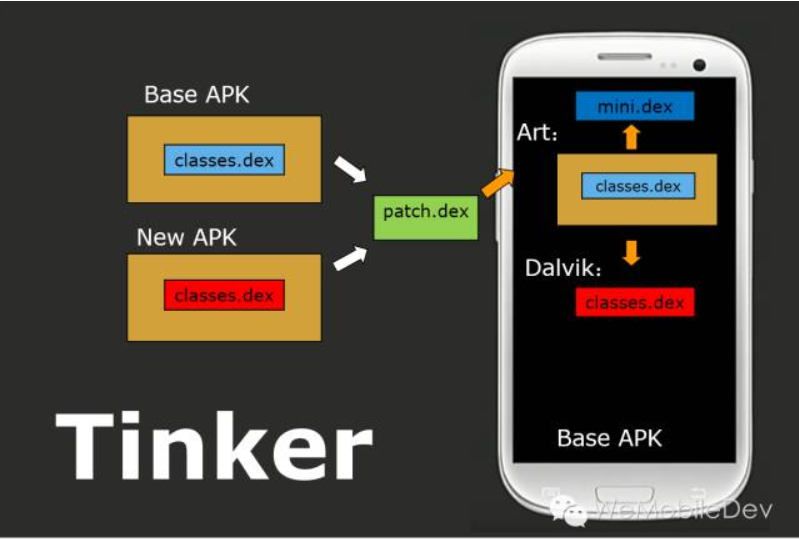
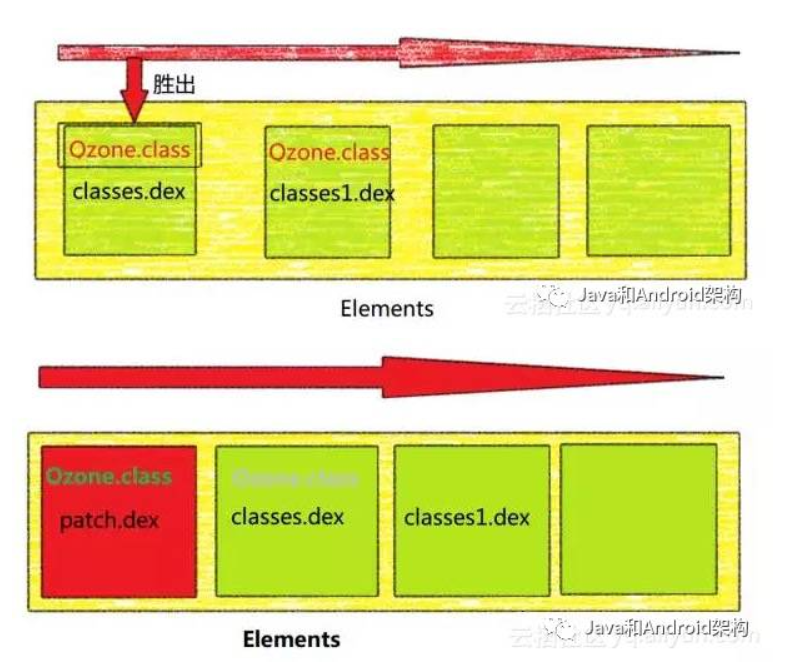
}

return null;

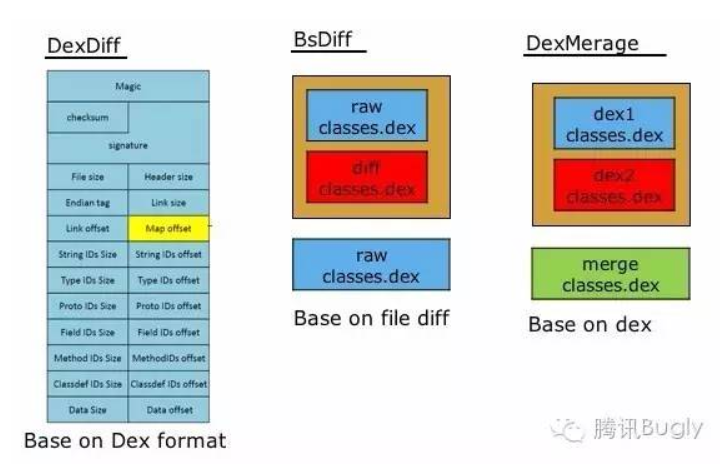
}

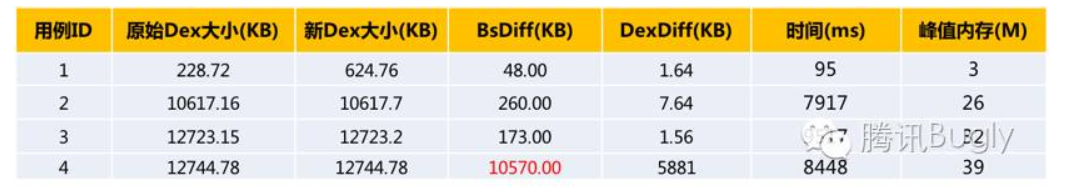
}

DexPathList的findClass也很简单，dexElements是维护dex文件的数组，每一个item对应一个dex文件。DexPathList遍历dexElements，从每一个dex文件中查找目标类，在找到后即返回并停止遍历。所以要想达到热修复的目的就必须让补丁dex在dexElements中的位置先于原有dex：



Tinker差异包的生成是在编译时通过新旧两个Dex生成差异path.dex。在运行时，将差异patch.dex重新跟原始安装包的旧Dex还原为新的Dex。这个过程可能比较耗费时间与内存，所以我们是单独放在一个后台进程:patch中。为了补丁包尽量的小，微信自研了DexDiff算法，它深度利用Dex的格式来减少差异的大小。它的粒度是Dex格式的每一项，可以充分利用原本Dex的信息，而BsDiff的粒度是文件，AndFix/QZone的粒度为class。





通过DexDiff算法的实现，既解决了Dalvik平台的性能损耗问题，又解决了Art平台补丁包过大的问题。但这套方案的缺点在于占Rom体积比较大，微信考虑到移动设备的存储空间提升比较快，增加几十M的Rom空间这个代价可以接受。

but Tinker也开始收费了，具体可以参考：<http://www.tinkerpatch.com/Price>



# 参考资料：

【1】[Android热修复方案比较](https://www.jianshu.com/p/eec0ab6800a4)

【2】[最全面的Android热修复技术——Tinker、nuwa、AndFix、Dexposed](http://www.bbs0101.com/archives/1465.html)

【3】[Android热修复技术总结](https://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649551099&idx=1&sn=3ec0599c93bef164eb3696fe2a6552df&chksm=f1181b86c66f9290eeec68fa2f2c5735ad590a9b584571bac8325b25947eb4d78c71d1707d39&mpshare=1&scene=1&srcid=01292lVT5zH6F19qptRsdOr2#rd)

【4】[阿里Android热修复技术选型——三大流派解析](https://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649548089&idx=1&sn=939f45217087f332f945a703c658bbe1&chksm=f1180e44c66f8752b54ff55e2f8994eec97fcca8e30c9f5a10e3503dcc029483fd630d22dd4e&mpshare=1&scene=1&srcid=0129NMq9vVlHzFODa8Ty2XNq#rd)

【5】[Android热更新技术的研究与实现](https://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649550244&idx=1&sn=c60b051ba16f8a0cde3bfd4c52365d40)

【6】[业界首个非侵入式热修复方案Sophix重磅推出，颠覆移动端传统更新流程！](https://yq.aliyun.com/articles/103527)

【7】[Sophix-阿里第三代非侵入式热修复](https://www.jianshu.com/p/8ea4d653a53e)

【8】[阿里热修复方案Sophix](http://blog.csdn.net/u013795543/article/details/73250166)

【9】[震惊！阿里热更新框架 Sophix 背后隐藏的真相](https://www.jianshu.com/p/853dae4092d7)

【10】[Android插件化与热修复(六)-微信Tinker原理分析](https://www.jianshu.com/p/2216554d3291)

【11】[Android热修复技术——Tinker方案解析](https://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649548879&idx=1&sn=32754304690812bd834742d7bc09666c&chksm=f1180332c66f8a24f69495938b517aa6eac0d80e4952cd36b4bef7c31d17efdfc6b052c3b697&mpshare=1&scene=1&srcid=0129uxI6dE8bwRHzuu8Jxdm8#rd)

【12】[微信热补丁Tinker -- 补丁流程](http://blog.csdn.net/qq_19711823/article/details/53199045)

【13】[微信Tinker的一切都在这里，包括源码(一)](http://dev.qq.com/topic/57ecdf2d98250b4631ae034b)

【14】[Tinker接入及原理分析](http://blog.csdn.net/joye123/article/details/54564339)

【15】[Tencent/tinker](https://github.com/Tencent/tinker)