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

对于InstantRun不了解的同学可以去查看它的官方文档。

另外，这个和HotPatch有什么关系呢？可以这么说，InstantRun就是Android上HotPatch的一种形式，了解了InstantRun的工作原理之后，我们可以更好的理解，甚至改进现有的HotPatch框架。

了解InstantRun

首先，先带没有用过InstantRun的同学了解一下如何使用这个功能。

在使用InstantRun之前，你必须要保证你的AS是2.0以上的版本，并且gradle的版本也要是2.0.0以上。

在你的Preference中点击Instant Run并且勾选对应的选项。

preference

在你运行你的程序之前，你可以看到你的run和debug图标是这样的：

run&debug-before

当你点击运行之后，它变成了这样：

run&debug-after

这意味着如果你修改完代码之后，直接点击对应的run按钮，你的程序不用构建，就直接可以运行。

举个例子，你现在界面有一个Button，点击之后改变TextView的文字。

public class MainActivity extends AppCompatActivity {

private TextView tv;

private String changeStr;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

tv = (TextView)findViewById(R.id.tv);

}

public void change(View view){

changeStr = "some errors!!";

tv.setText(changeStr);

}

}

当你运行程序，点击Button之后TextView显示的是some errors，这个时候你发现了错误，修改了代码变成下面这样：

public class MainActivity extends AppCompatActivity {

private TextView tv;

private String changeStr;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

tv = (TextView)findViewById(R.id.tv);

}

public void change(View view){

changeStr = "fix it!!";

tv.setText(changeStr);

}

}

如果是原来，你点击了run之后，你的程序会重新构建并且重新部署到手机上，而如果你开启了InstantRun，在你点击run之后，程序不会重新构造，而当你点下Button的时候，你会惊讶的发现功能已经修复了，TextView的文字改成了fix it。

当然，并不是所有的修复都可以这样［无缝修改］的，有一些需要重启对应的Activity，有一些则要重启整个app。所以InstantRun的修复分为三类：hot swap，warm swap和cold swap。

hot swap是三种类型中最快生效的，它的可以作用在一般代码的修改上，比如上面的例子。

warm swap是针对资源的修改，需要你重启对应的Activity。

cold swap是最慢的一种，它需要你重启整个app，并且需要你的Android API在21或者以上，对于API20以下的，则会和原来一样，重新构建并部署应用。

这样的功能为我们开发者节省了构建程序和安装程序的过程，可谓造福人类啊！

看到这里，有些同学可能会说，这不就是HotPatch吗！是的，InstantRun就是一个HotPatch。并且它和现在一些主流的HotPatch框架的实现原理是有所不同的，所以希望大家在看完这篇文章之后会有所收获。

深入源码

首先，为什么使用InstantRun要把gradle的版本升级到2.0.0以上呢，因为在1.5的时候，gradle增加了transform api。并且谷歌在2.0.0的时候利用这个api做了一些事，实现了InstantRun。

首先我们看一下当我们在第一次构建应用的时候对应的message输出。

firstBuild

在一大堆的输出中，最后有几个transform开头的task，其中有几个带有InstantRun的标示：

transform

下面再来看看经过修改之后，再次点击run按钮以后的输出。

secondBuild

可以看到还是有几个同样带有transform和InstantRun标示的task，这里对于gradle task和transform api我不做具体的讲解，大家可以自行查阅相关资料。上这么多图给大家看的原因就是想要告诉大家，InstantRun确实是通过transform这样的一个api去实现的。

大致了解了一下InstantRun之后，我们就应该从源码角度去分析了。在分析之前，我先告诉大家它的一个大概工作流程，这里分析的是hot swap，也就是一般代码的修复：

(1) 在第一次构建app的时候，它利用了transform去在每一个类注入了一个字段叫做change，它实现了IncrementalChange接口，并且在每一个方法中插入了一个逻辑，如果change不为空，就执行的change的accessdispatch方法，否则执行原方法的原来逻辑。对应的类在app/build/intermediates/transforms/instantRun/debug/folders/1/5目录下。这里多说一句，InstantRun操作字节码用的是asm。

(2) 当你修改完对应的代码点击run按钮之后，InstantRun会去生成对应的patch文件，在app/build/intermediates/transforms/instantRun/debug/folders/4000/5目录下。而对应patch文件中的补丁类的名字是你修改的那个类的名字后面加$override，并且实现了IncrementalChange接口。

(3) 生成一个纪录类AppPatchesLoaderImpl，用来记录哪些类被修改过。

(4) 通过AppPatchesLoaderImpl类将修改过的类中的赋值成中生成的change赋值成(2)中生成的xxxxoverride。

以我们上面的例子看来。

(1)中通过asm修改字节码后的MainActivity：

public class MainActivity extends AppCompatActivity {

private TextView tv;

private String changeStr;

public MainActivity() {

IncrementalChange var1 = $change;

if(var1 != null) {

Object[] var2;

Object[] var10003 = var2 = new Object[1];

var10003[0] = var2;

Object[] var3 = (Object[])var1.access$dispatch("init$args.([Ljava/lang/Object;)Ljava/lang/Object;", var10003);

this(var3, (InstantReloadException)null);

} else {

super();

}

if(var1 != null) {

var1.access$dispatch("init$body.(Lzjutkz/com/instantrundemo/MainActivity;)V", new Object[]{this});

}

}

public void onCreate(Bundle savedInstanceState) {

IncrementalChange var2 = $change;

if(var2 != null) {

var2.access$dispatch("onCreate.(Landroid/os/Bundle;)V", new Object[]{this, savedInstanceState});

} else {

super.onCreate(savedInstanceState);

this.setContentView(2130968601);

this.tv = (TextView)this.findViewById(2131492944);

}

}

public void change(View view) {

IncrementalChange var2 = $change;

if(var2 != null) {

var2.access$dispatch("change.(Landroid/view/View;)V", new Object[]{this, view});

} else {

this.changeStr = "error!!";

this.tv.setText(this.changeStr);

}

}

MainActivity(Object[] var1, InstantReloadException var2) {

String var3 = (String)var1[0];

switch(var3.hashCode()) {

case -2089128195:

super();

return;

case 584748498:

this();

return;

default:

throw new InstantReloadException(String.format("String switch could not find \'%s\' with hashcode %s in %s", new Object[]{var3, Integer.valueOf(var3.hashCode()), "zjutkz/com/instantrundemo/MainActivity"}));

}

}

}

(2)中patch文件中的补丁类：

public class MainActivity$override implements IncrementalChange {

public MainActivity$override() {

}

public static Object init$args(Object[] var0) {

Object[] var1 = new Object[]{"android/support/v7/app/AppCompatActivity.()V"};

return var1;

}

public static void init$body(MainActivity $this) {

}

public static void onCreate(MainActivity $this, Bundle savedInstanceState) {

Object[] var2 = new Object[]{savedInstanceState};

MainActivity.access$super($this, "onCreate.(Landroid/os/Bundle;)V", var2);

$this.setContentView(2130968601);

AndroidInstantRuntime.setPrivateField($this, (TextView)$this.findViewById(2131492944), MainActivity.class, "tv");

}

public static void change(MainActivity $this, View view) {

AndroidInstantRuntime.setPrivateField($this, "fix it!!", MainActivity.class, "changeStr");

((TextView)AndroidInstantRuntime.getPrivateField($this, MainActivity.class, "tv")).setText((String)AndroidInstantRuntime.getPrivateField($this, MainActivity.class, "changeStr"));

}

public Object access$dispatch(String var1, Object... var2) {

switch(var1.hashCode()) {

case -1630101479:

return init$args((Object[])var2[0]);

case -641568046:

onCreate((MainActivity)var2[0], (Bundle)var2[1]);

return null;

case 106989371:

change((MainActivity)var2[0], (View)var2[1]);

return null;

case 1753553473:

init$body((MainActivity)var2[0]);

return null;

default:

throw new InstantReloadException(String.format("String switch could not find \'%s\' with hashcode %s in %s", new Object[]{var1, Integer.valueOf(var1.hashCode()), "zjutkz/com/instantrundemo/MainActivity"}));

}

}

}

(3)中生成的记录类：

public class AppPatchesLoaderImpl extends AbstractPatchesLoaderImpl {

public AppPatchesLoaderImpl() {

}

public String[] getPatchedClasses() {

return new String[]{"zjutkz.com.instantrundemo.MainActivity"};

}

大致关于hot swap的流程就是这样，下面让我们从最开始的地方出发，走一遍InstantRun的流程，并且了解下warm swap和cold swap的机制。

1.替换application

首先，大家看一下app/build/intermediates/bundles/debug/instant-run目录下的AndroidMenifest文件。

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="zjutkz.com.instantrundemo"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="14"

android:targetSdkVersion="23" />

<application

android:name="com.android.tools.fd.runtime.BootstrapApplication"

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme" >

<activity android:name="zjutkz.com.instantrundemo.MainActivity" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

可以看到对应的Application被替换了，变成了BootstrapApplication。这个Application在哪里呢，在app/build/intermediates/incremental-runtime-classes/debug目录下的instant-run.jar中，这个jar包大家可以通过JD-GUI去打开。

下面让我们看看BootstrapApplication，首先看的肯定是attchBaseContext方法。

protected void attachBaseContext(Context context)

{

if (!AppInfo.usingApkSplits) {

String apkFile = context.getApplicationInfo().sourceDir;

long apkModified = apkFile != null ? new File(apkFile).lastModified() : 0L;

createResources(apkModified);

setupClassLoaders(context, context.getCacheDir().getPath(), apkModified);

}

createRealApplication();

super.attachBaseContext(context);

if (this.realApplication != null)

try {

Method attachBaseContext = ContextWrapper.class.getDeclaredMethod("attachBaseContext", new Class[] { Context.class });

attachBaseContext.setAccessible(true);

attachBaseContext.invoke(this.realApplication, new Object[] { context });

} catch (Exception e) {

throw new IllegalStateException(e);

}

}

可以看到最前面有一个setupClassLoaders方法。

private static void setupClassLoaders(Context context, String codeCacheDir, long apkModified)

{

List dexList = FileManager.getDexList(context, apkModified);

Class server = Server.class;

Class patcher = MonkeyPatcher.class;

if (!dexList.isEmpty()) {

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", new StringBuilder().append("Bootstrapping class loader with dex list ").append(join('\n', dexList)).toString());

}

ClassLoader classLoader = BootstrapApplication.class.getClassLoader();

String nativeLibraryPath;

try {

nativeLibraryPath = (String)classLoader.getClass().getMethod("getLdLibraryPath", new Class[0]).invoke(classLoader, new Object[0]);

if (Log.isLoggable("InstantRun", 2))

Log.v("InstantRun", new StringBuilder().append("Native library path: ").append(nativeLibraryPath).toString());

}

catch (Throwable t) {

Log.e("InstantRun", new StringBuilder().append("Failed to determine native library path ").append(t.getMessage()).toString());

nativeLibraryPath = FileManager.getNativeLibraryFolder().getPath();

}

IncrementalClassLoader.inject(classLoader, nativeLibraryPath, codeCacheDir, dexList);

}

}

可以看到它IncrementalClassLoader使用了IncrementalClassLoader.inject方法，而在这个方法里面做的工作是把IncrementalClassLoader作为当前classLoader的父loader，我们都知道java的类加载模型是［双亲委托］的，所以之后加载类都会从IncrementalClassLoader中加载。

回到attachBaseContext方法，之后调用了createRealApplication方法去创建真正的Application，也就是我们应用的Application，比如你自定义的MyApplication并且反射调用它的attachBaseContext。

下面让我们看看onCreate方法。

public void onCreate()

{

if (!AppInfo.usingApkSplits) {

MonkeyPatcher.monkeyPatchApplication(this, this, this.realApplication, this.externalResourcePath);

MonkeyPatcher.monkeyPatchExistingResources(this, this.externalResourcePath, null);

}

else

{

MonkeyPatcher.monkeyPatchApplication(this, this, this.realApplication, null);

}

super.onCreate();

if (AppInfo.applicationId != null) {

try {

boolean foundPackage = false;

int pid = Process.myPid();

ActivityManager manager = (ActivityManager)getSystemService("activity");

List processes = manager.getRunningAppProcesses();

boolean startServer;

if ((processes != null) && (processes.size() > 1))

{

boolean startServer = false;

for (ActivityManager.RunningAppProcessInfo processInfo : processes) {

if (AppInfo.applicationId.equals(processInfo.processName)) {

foundPackage = true;

if (processInfo.pid == pid) {

startServer = true;

break;

}

}

}

if ((!startServer) && (!foundPackage))

{

startServer = true;

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Multiprocess but didn't find process with package: starting server anyway");

}

}

}

else

{

startServer = true;

}

if (startServer)

Server.create(AppInfo.applicationId, this);

}

catch (Throwable t) {

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Failed during multi process check", t);

}

Server.create(AppInfo.applicationId, this);

}

}

if (this.realApplication != null)

this.realApplication.onCreate();

}

首先调用了MonkeyPatcher.monkeyPatchApplication方法，这个方法我们就不跟进去看了，具体作用是：1.把对应的application替换成我们真正的applictaion。2.替换资源路径，老的资源路径是/data/app/[package name]-1.apk，新的资源路径是/data/data/[applicationId]/files/instant-run/resources.ap\_。3.把真正application的LoadedApk替换成BootstrapApplication的，为什么要这么做呢，因为LoadedApk中持有了ClassLoader，这样替换以后，我们程序中加载类都会使用BootstrapApplication的LoadedApk，从而使用它的ClassLoader，而在之前我们已经把ClassLoader的父loader设置成了IncrementalClassLoader，绕了这么一大圈，其实就是为了［使用IncrementalClassLoader去加载类］。那为什么要使用IncrementalClassLoader去加载类呢，因为我们生成的patch文件是不能直接通过程序的ClassLoader去加载的，而IncrementalClassLoader把patch的路径传了进去，这样就可以加载了～

之后的monkeyPatchApplication是通过反射修改了和资源加载相关的东西，例如addAssetPath方法，AssetManager等，将其指向新的资源路径。

到此替换Application的前半部分就讲完了，它的重要作用是为了创建一个IncrementalClassLoader用来加载patch文件中的补丁类。

通过前面的分析我们知道了为什么应用可以去加载patch文件中的补丁类，下面让我们继续。

首先还是看BootstrapApplication类的onCreate函数。在后面调用了Server.create方法。而在Server的构造函数中，调用了startServer方法。

private void startServer()

{

try {

Thread socketServerThread = new Thread(new SocketServerThread(null));

socketServerThread.start();

}

catch (Throwable e)

{

if (Log.isLoggable("InstantRun", 6))

Log.e("InstantRun", "Fatal error starting Instant Run server", e);

}

}

我们看看SocketServerThread的run方法。

public void run()

{

while (true)

try

{

LocalServerSocket serverSocket = Server.this.mServerSocket;

if (serverSocket == null) {

break;

}

LocalSocket socket = serverSocket.accept();

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Received connection from IDE: spawning connection thread");

}

Server.SocketServerReplyThread socketServerReplyThread = new Server.SocketServerReplyThread(Server.this, socket);

socketServerReplyThread.run();

if (Server.sWrongTokenCount > 50) {

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Stopping server: too many wrong token connections");

}

Server.this.mServerSocket.close();

break;

}

} catch (Throwable e) {

if (Log.isLoggable("InstantRun", 2))

Log.v("InstantRun", "Fatal error accepting connection on local socket", e);

}

}

其中使用了Socket，这下我们就明白了，原来InstantRun内部使用了Socket来进行通信。也就是说当我们修改完程序点击run之后，AndroidStudio会通过socket将数据传递给我们，最终调用的是handlePatches方法。

private int handlePatches(List<ApplicationPatch> changes, boolean hasResources, int updateMode)

{

if (hasResources) {

FileManager.startUpdate();

}

for (ApplicationPatch change : changes) {

String path = change.getPath();

if (path.endsWith(".dex")) {

handleColdSwapPatch(change);

boolean canHotSwap = false;

for (ApplicationPatch c : changes) {

if (c.getPath().equals("classes.dex.3")) {

canHotSwap = true;

break;

}

}

if (!canHotSwap) {

updateMode = 3;

}

}

else if (path.equals("classes.dex.3")) {

updateMode = handleHotSwapPatch(updateMode, change);

} else if (isResourcePath(path)) {

updateMode = handleResourcePatch(updateMode, change, path);

}

}

if (hasResources) {

FileManager.finishUpdate(true);

}

return updateMode;

}

看到这里大家有没有眼前一亮，这个方法会根据文件的后缀名去执行对应的方法，而对应的方法正是

handleHotSwapPatch(对应 hot swap)，handleResourcePatch(对应 warm swap)和handleColdSwapPatch(对应 cold swap)。

2.hot swap

让我们先看比较熟悉的hot swap。

private int handleHotSwapPatch(int updateMode, ApplicationPatch patch) {

if (Log.isLoggable("InstantRun", 2))

Log.v("InstantRun", "Received incremental code patch");

try

{

String dexFile = FileManager.writeTempDexFile(patch.getBytes());

if (dexFile == null) {

Log.e("InstantRun", "No file to write the code to");

return updateMode;

}if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Reading live code from " + dexFile);

}

String nativeLibraryPath = FileManager.getNativeLibraryFolder().getPath();

DexClassLoader dexClassLoader = new DexClassLoader(dexFile, this.mApplication.getCacheDir().getPath(), nativeLibraryPath, getClass().getClassLoader());

Class aClass = Class.forName("com.android.tools.fd.runtime.AppPatchesLoaderImpl", true, dexClassLoader);

try

{

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Got the patcher class " + aClass);

}

PatchesLoader loader = (PatchesLoader)aClass.newInstance();

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Got the patcher instance " + loader);

}

String[] getPatchedClasses = (String[])aClass.getDeclaredMethod("getPatchedClasses", new Class[0]).invoke(loader, new Object[0]);

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Got the list of classes ");

for (String getPatchedClass : getPatchedClasses) {

Log.v("InstantRun", "class " + getPatchedClass);

}

}

if (!loader.load())

updateMode = 3;

}

catch (Exception e) {

Log.e("InstantRun", "Couldn't apply code changes", e);

e.printStackTrace();

updateMode = 3;

}

} catch (Throwable e) {

Log.e("InstantRun", "Couldn't apply code changes", e);

updateMode = 3;

}

return updateMode;

}

逻辑比较多，其中最核心的就是通过dexPath创建一个ClassLoader，并且通过它去创建一个AppPatchesLoaderImpl，然后执行AppPatchesLoaderImpl的load方法。AppPatchesLoaderImpl这个类大家还记得吧，就是之前的那个［记录类］。

public class AppPatchesLoaderImpl extends AbstractPatchesLoaderImpl {

public AppPatchesLoaderImpl() {

}

public String[] getPatchedClasses() {

return new String[]{"zjutkz.com.instantrundemo.MainActivity"};

}

}

它继承自AbstractPatchesLoaderImpl，也就是说load的逻辑在AbstractPatchesLoaderImpl中。

public boolean load()

{

try

{

for (String className : getPatchedClasses()) {

ClassLoader cl = getClass().getClassLoader();

Class aClass = cl.loadClass(className + "$override");

Object o = aClass.newInstance();

Class originalClass = cl.loadClass(className);

Field changeField = originalClass.getDeclaredField("$change");

changeField.setAccessible(true);

Object previous = changeField.get(null);

if (previous != null) {

Field isObsolete = previous.getClass().getDeclaredField("$obsolete");

if (isObsolete != null) {

isObsolete.set(null, Boolean.valueOf(true));

}

}

changeField.set(null, o);

if ((Log.logging != null) && (Log.logging.isLoggable(Level.FINE)))

Log.logging.log(Level.FINE, String.format("patched %s", new Object[] { className }));

}

}

catch (Exception e) {

if (Log.logging != null) {

Log.logging.log(Level.SEVERE, String.format("Exception while patching %s", new Object[] { "foo.bar" }), e);

}

return false;

}

return true;

}

它通过getPatchedClasses方法拿到对应修改过的类，这里就是我们的MainActivity。

后面逻辑已经很清晰了，大家对应之前我讲的(1)(2)(3)(4)去看就行了。

这样，我们就完成了hot swap，不用重新构建app，不用重启进程，甚至不用重启Activity！

3.warm swap

接下来让我们看资源替换。

private static int handleResourcePatch(int updateMode, ApplicationPatch patch, String path)

{

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Received resource changes (" + path + ")");

}

FileManager.writeAaptResources(path, patch.getBytes());

updateMode = Math.max(updateMode, 2);

return updateMode;

}

调用了FileManager.writeAaptResources方法。

public static void writeAaptResources(String relativePath, byte[] bytes)

{

File resourceFile = getResourceFile(getWriteFolder(false));

File file = resourceFile;

File folder = file.getParentFile();

if (!folder.isDirectory()) {

boolean created = folder.mkdirs();

if (!created) {

if (Log.isLoggable("InstantRun", 2)) {

Log.v("InstantRun", "Cannot create local resource file directory " + folder);

}

return;

}

}

if (relativePath.equals("resources.ap\_"))

{

writeRawBytes(file, bytes);

}

else

writeRawBytes(file, bytes);

}

可以看到它去获取了对应的资源文件，就是我们在上面提到的/data/data/[applicationId]/files/instant-run/resources.ap\_，InstantRun直接对它进行了字节码操作，把通过Socket传过来的修改过的资源传递了进去。对Android上的资源打包不了解的同学可以去看老罗的[Android应用程序资源的编译和打包过程分析这篇文章。很可惜，writeRawBytes这个方法在反编译的情况下看不到，具体的源码我还在寻找当中。。

4.cold swap

对于cold swap，其实就是把数据写进对应的dex中，所以在art的情况下需要重启app，而对于API20以下的只能重新构建和部署了。

private static void handleColdSwapPatch(ApplicationPatch patch) {

if (patch.path.startsWith("slice-")) {

File file = FileManager.writeDexShard(patch.getBytes(), patch.path);

if (Log.isLoggable("InstantRun", 2))

Log.v("InstantRun", "Received dex shard " + file);

}

}

public static File writeDexShard(byte[] bytes, String name){

File dexFolder = getDexFileFolder(getDataFolder(), true);

if (dexFolder == null) {

return null;

}

File file = new File(dexFolder, name);

writeRawBytes(file, bytes);

return file;

}

对比热修复

讲完了InstantRun的原理，不知道大家是不是看的眼睛痛了呢，其实我想说，下面这个才是重头戏！因为在我看来，了解源码的目的是为了去利用它，只是单单去知道一个库的工作原理有什么？又不是你写的，大家说是吧。

大家也知道现在有很多优秀的HotPatch开源库，代表的就是Nuwa。

看过Nuwa源码的同学都知道，它的原理是将patch的dexPahList中的Element数组插入到宿主的Element数组之前。这种方案呢，是基于单ClassLoader的，也就是说整个应用中只有一个ClassLoader，这样一来，如果一个类被加载了那么在程序运行的时间呢，它是不会再去通过ClassLoader加载一遍的，所以就导致了这样的HotPatch框架［每次打patch以后要重启应用才会生效］，但是对于InstantRun的hot swap是不存在这样的限制的，为什么呢？因为它是基于多ClassLoader的，前面源码中也有提到，它的每一个patch都有一个ClassLoader，这就意味着如果你想更新patch，它都会创建一个ClassLoader，而在java中不同ClassLoader创建的类被认为是不同的，所以会重新加载新的patch中的补丁类。

另外，现在的HotPatch框架对资源替换的支持做的都比较一般，但是看了上面的源码，大家会发现InstantRun对资源的支持是比较好的，核心逻辑就是替换资源目录并且去操作resources.ap\_文件，通过这种方式就能达到warm swap的目的，具体的实现方案我还在研究中。

还有一点，如果大家使用过Nuwa，你会发现Application类是无法打patch的，具体原因可以去看区长写的聊聊Android 热修复Nuwa有哪些坑。但是通过替换Application这样的方式，我们把我们真正的Application变成DelegateApplication而使用PorxyApplication去执行patch框架的初始化并且加载DelegateApplication，有可能可以解决这一问题。具体可不可以还是要实践了才知道。如果不可以的话。。。当我瞎说的吧～

最后，说了InstantRun的优点，那么它对于现在的HotPatch框架有什么缺点呢？这里我说一点吧，InstantRun利用了transform api去生成字节码，这样的方式不是说不好，只能说不灵活，因为所有的transform操作是由TransformManager管理的，也就是说它执行的时机是固定的，如果涉及到混淆，dex等操作，这些task的顺序都是不可变的，这样的就会踩出很多坑来，我们可以换一种方式，像Nuwa一样，自己去写一个task，并且通过依赖的方式插入到你想要插入的task链的位置，非常灵活。

如果能结合现有的HotPatch框架和InstantRun，我们就可以打造一个功能完善的官方版的热修复，想想还有些小激动呢～