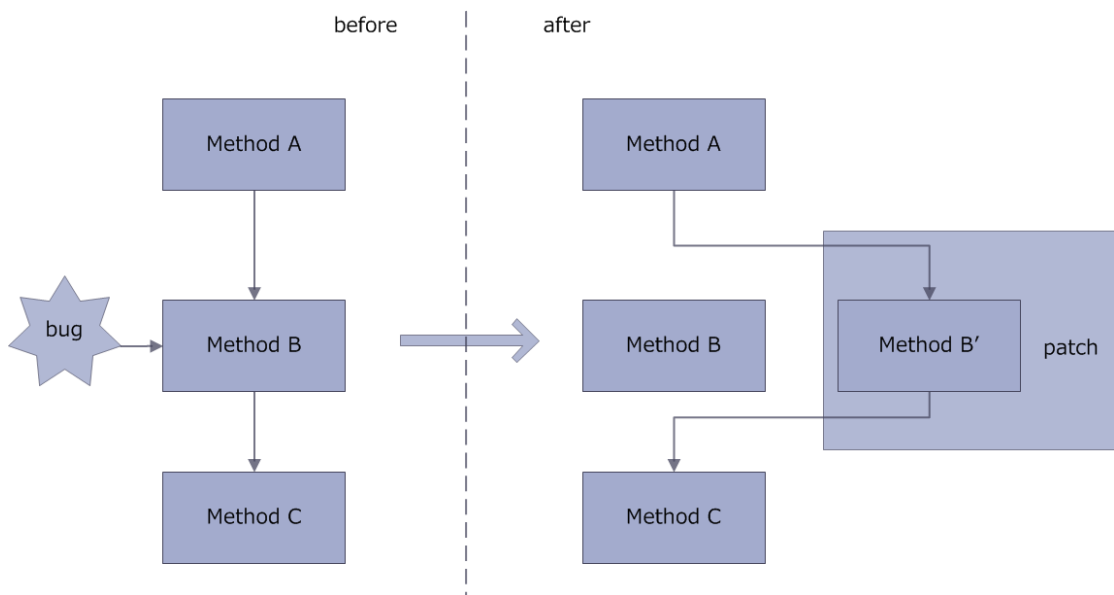


热修复 Andfix 原理及实践

1. 原理

热修复 Andfix 的原理简单的说就是方法体的替换。



它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。核心在 **native** 层的 `replaceMethod` 函数。

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

参数是在 **java** 层通过反射机制得到的 **Method** 对象所对应的 **jobject**，**src** 对应是需要被替换的方法，**dest** 对应新方法，新方法在补丁包中。

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。因此替换有两种方法：

```
@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);  
    }  
}
```

因为相同的虚拟机一样存在不同的 **Android** 版本，比如相同的 **art** 虚拟机存在 5.0, 6.0 等，而且不同版本的底层 **java** 对象是不同的，因此需要进一步的区分不同的替换函数。

```
@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);
    }
}
```

下面以 6.0 为例。

每个 java 方法在 art 虚拟机中都对应着一个 ArtMethod。ArtMethod 记录了这个 java 方法所有的信息，包括所属类，访问权限，代码执行地址等。

所以上面所说的方法的替换，实际上就是将新方法中的 ArtMethod 所有成员替换旧方法 ArtMethod 中的所有成员。

以 replace_6_0()为例：

```
@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_;

    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_);
}
```

我们通过 `env->FromReflectedMethod`，可以有 `Method` 对象得到这个方法对应的 `ArtMethod` 的真正起始地址，然后就可以把把强转为 `ArtMethod` 指针，从而对其所有成员进行修改。

但是!!! 但是!!! 但是!!! 这里的 `ArtMethod` 结构是 alibaba 根据 Android 开源代码中的 `ArtMethod` 结构就行重写的，和实际运行设备中的 `ArtMethod` 结构可能会不一样，这种情况发生在手机厂商没有完全使用 Android 开源代码中的 `ArtMethod`，而是对其进行了修改，那么这种情况下用自己构造的 `ArtMethod` 结构替换成手机厂商修改过的 `ArtMethod`，那么里面的成员就会发生错乱，导致热修复失败。

可能大家会有疑问 `ArtMethod` 什么时候存在的？

类加载完成后，得到的是一个 `Class` 对象。这个 `Class` 对象关联有一系列的 `ArtField` 对象和 `ArtMethod` 对象。其中，`ArtField` 对象描述的是成员变量，而 `ArtMethod` 对象描述的是成员函数。



因此这就导致了很大的局限性，手机厂商修改了 `ArtMethod` 就无法实现热修复了。这也是 Andfix 不支持很多机型的原因。

使用 Andfix 进行热修复还有其他的局限性：

1. 发生 bug 的类，修 bug 后形成的新类不能改变方法的数量，也不能改变成员的数量，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。字段的增加和减少同理。
2. 修复了的非静态方法会被反射调用。这种情况下是什么原因呢？通过反射可以得到 `Method`，而这个 `Method` 是新类中的，包含的类信息也是新类的，当 `invoke` 的时候，传递进去的却是旧类的实例，这时候由于两个类的 `ClassLoader` 不一致，但是无法正确的 `invoke`，进而抛出异常，这个的解决方法是将新类的 `ClassLoader` 修改为旧类的 `ClassLoader`。

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

Demo 实践

Andfix Github: <https://github.com/alibaba/AndFix>

- 1.新建一个 Android Studio Project Andfix
- 2.在 app 的 Gradle 文件中添加必要的依赖，点击同步

compile 'com.alipay.euler:andfix:0.5.0@aar'

```
compile 'com.android.support.constraint:constraint-layout:1.0.2'
compile 'com.alipay.euler:andfix:0.5.0@aar'
compile 'com.android.support:appcompat-v7:25.0.1'
testCompile 'junit:junit:4.12'
```

其中 compile 'com.android.support:appcompat-v7:25.0.1'只是为了实现兼容，动态申请权限使用。

- 3.新建 Application 子类 AndFixApplication，并在 Manifest 中添加
- 4.在 onCreate 方法中实例化 PatchManager 并初始化，并 load 加载。

```
patchManager = new PatchManager(this);
//初始化版本
patchManager.init("1.0");
//You should load patch as early as possible, generally, in the initialization phase of your
//application(such as Application.onCreate()).
patchManager.loadPatch();
```

```
@Override
public void onCreate() {
    super.onCreate();
    initPatchManager();
}

private void initPatchManager() {
    patchManager = new PatchManager(this);
    //初始化版本
    patchManager.init("1.0");
    //You should load patch as early as possible, generally, in the initialization phase of your application(such as Application.onCreate()).
    patchManager.loadPatch();
    //path of the patch file that was downloaded, 这个可以使用定时的判断服务器是否有可更新的补丁包进而下载，addPatch进行热修复
    //也可以使用服务器推送，接受推送并下载后，使用patchManager.addPath(String path)，传入下载的补丁包的路径即可实现热修复
    // patchManager.addPatch();
}
```

- 5.接着通过 parchManager.addPath(String path)方法传入补丁包的路径即可，便可完成热修复。补丁包的获取：

- 1.可以使用定时的判断服务器是否有可更新的补丁包进而下载，parchManager.addPath(String path)进行热修复，传入下载的补丁包的路径。
- 2.也可以使用服务器推送，接受推送并下载后，使用 parchManager.addPath(String path)，传入下载的补丁包的路径。
- 6.实际应用中，可以通过以上方法进行修复，而在本 demo 中，是直接将补丁包放到--/Android/data/本应用包名(com.example.andfix)/file 中，因此在 AndFixApplication 定义了补丁包的路径为 String final

```
//默认的补丁路径
public final static String PATCH_PATH =
"/storage/emulated/0/Android/data/com.example.andfix/files/";
```

另外为了让这个文件夹存在，在 initPatchManager 方法中添加了 this.getExternalFilesDir(null).getAbsolutePath();

因此 AndFixApplication 的所有内容是：

```

public class AndFixApplication extends Application {

    public static PatchManager patchManager;

    //默认的补丁路径
    public final static String PATCH_PATH = "/storage/emulated/0/Android/data/com.example.andfix/files/";

    @Override
    public void onCreate() {
        super.onCreate();
        initPatchManager();
    }

    private void initPatchManager() {
        patchManager = new PatchManager(this);
        //初始化版本
        patchManager.init("1.0");
        //You should load patch as early as possible, generally, in the initialization phase of your application(such as Application.onCreate()).
        patchManager.loadPatch();
        //path of the patch file that was downloaded, 这个可以使用定时的判断服务器是否有可更新的补丁包进而下载, addPatch进行热修复
        //也可以使用服务器推送, 接受推送并下载后, 使用patchManager.addPath(String path), 传入下载的补丁包的路径即可实现热修复
        // 用于生成这个文件夹
        patchManager.addPatch(
            this.getExternalFilesDir(null).getAbsolutePath());
    }
}

```

7. 因为需要访问本地存储。所以需要添加访问权限。

```

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

```

由于 Android6.0 需要动态权限的申请, 所以添加了 CheckPermissionsActivity 类, 是之后添加的 Activity 的父类, CheckPermissionsActivity 类的主要作用就是动态判断当前 Activity 需要的权限是否已经获取, 不获取并动态申请。CheckPermissionsActivity 的内容

```

/**
 * 继承了Activity, 实现Android6.0的运行时权限检测
 * 需要进行运行时权限检测的Activity可以继承这个类
 */
public class CheckPermissionsActivity extends Activity
    implements
        ActivityCompat.OnRequestPermissionsResultCallback {

    /**
     * 需要进行检测的权限数组
     */
    protected String[] needPermissions = {
        Manifest.permission.READ_EXTERNAL_STORAGE,
        Manifest.permission.WRITE_EXTERNAL_STORAGE,
    };

    /**
     * 子Activity传入的permission
     */
    protected String[] mInputPermission;

    private static final int PERMISSION_REQUESTCODE = 0;

    /**
     * 判断是否需要检测, 防止不停的弹框
     */
    private boolean isNeedCheck = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /**
         * 通过bundle获取子Activity传入的permission
         */
        try {
            mInputPermission = savedInstanceState.getStringArray("permission");
        } catch (NullPointerException e) {
            {
                mInputPermission = null;
            }
        }
    }
}

```

```

@Override
protected void onResume() {
    super.onResume();
    if(isNeedCheck){
        if(mInputPermission!=null) {
            checkPermissions(mInputPermission);
        }else{
            checkPermissions(needPermissions);
        }
    }
}

/**
 *
 * @param permissions
 * @since 2.5.0
 */
private void checkPermissions(String... permissions) {
    List<String> needRequestPermissonList = findDeniedPermissions(permissions);
    if (null != needRequestPermissonList
        && needRequestPermissonList.size() > 0) {
        ActivityCompat.requestPermissions(this,
            needRequestPermissonList.toArray(
                new String[needRequestPermissonList.size()]),
            PERMISSION_REQUESTCODE);
    }
}

/**
 * 获取权限集中需要申请权限的列表
 *
 * @param permissions
 * @return
 * @since 2.5.0
 */
private List<String> findDeniedPermissions(String[] permissions) {
    List<String> needRequestPermissonList = new ArrayList<>();
    for (String perm : permissions) {
        if (ContextCompat.checkSelfPermission(this,
            perm) != PackageManager.PERMISSION_GRANTED) {
            needRequestPermissonList.add(perm);
        } else {
            if (ActivityCompat.shouldShowRequestPermissionRationale(
                this, perm)) {
                needRequestPermissonList.add(perm);
            }
        }
    }
    return needRequestPermissonList;
}
}

```

8.接着修改 MainActivity 继承自 CheckPermissionsActivity，并且让 Button 控件 mbtnBug 的点击事件中触发 bug—NPE。

内容如下：

```

public class MainActivity extends CheckPermissionsActivity
    implements View.OnClickListener {
    Button mbtnBug;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mbtnBug = (Button) findViewById(R.id.id_main_btn_bug);
        mbtnBug.setOnClickListener(this);
        fixBug();
    }
    /**
     * 判断andfix.apatch补丁包是否存在，如果存在便进行热修复
     */
    private void fixBug() {
        String fixBugApkName = "andfix.apatch";
        String patchPath = AndFixApplication.PATCH_PATH + fixBugApkName;
        File patchFile = new File(patchPath);
        if (patchFile != null) {
            try {
                AndFixApplication.patchManager.addPatch(patchPath);
            } catch (IOException e) {
                //文件找不到或者类型出错
            }
        }
    }
    /**
     * 测试mbtnBug点击事件
     */
    @Override
    public void onClick(View v) {
        //测试bug，此时使用str会发生空指针异常
        String str = null;
        //修复bug，使用这个，并将上面的注释。
        //String str="andfix";
        if (!str.isEmpty()) {
            Toast.makeText(MainActivity.this, str, Toast.LENGTH_SHORT).show();
        }
    }
}

```

9.接着 build 一个带签名的 apk 并命名为 andfix-old,接着装在手机上。

点击按钮后，应用会 Crash 掉。





10.接在将 MainActivity 中的 onClick 方法里面的//String str="andfix";注释去掉并将 String str = null;注释，也就是修复了 bug。

11.此时再用相同的签名证书 build 带签名的 apk，并命名为 andfix-new。

12.此时下载 apkpatch 工具

<https://github.com/alibaba/AndFix/raw/master/tools/apkpatch-1.0.3.zip>

解压后：

	_MACOSX	2015/9/22 11:47	文件夹	
	apkpatch	2015/9/22 11:23	Windows 批处理...	1 KB
	apkpatch	2015/9/22 11:23	Shell Script	1 KB
	apkpatch-1.0.3	2015/9/22 11:45	Executable Jar File	5,750 KB

13 进入 cmd，并 cd 到解压的目录

通过以下命令生成.patch 文件。

```

usage: apkpatch -f <new> -t <old> -o <output> -k <keystore> -p <***> -a
<alias> -e <***>

```

```

-a,--alias <alias>    keystore entry alias.
-e,--epassword <***>  keystore entry password.
-f,--from <loc>       new Apk file path.
-k,--keystore <loc>   keystore path.
-n,--name <name>      patch name.
-o,--out <dir>        output dir.
-p,--kpassword <***>  keystore password.
-t,--to <loc>         old Apk file path.

```

生成.patch 文件

```

C:\Users\peiyu_wang\Downloads\apkpatch-1.0.3>apkpatch -f C:\Users\peiyu_wang\Desktop\andfix\andfix-old.apk -t C:\Users\p
ei_yu_wang\Desktop\andfix\andfix-new.apk -o C:\Users\peiyu_wang\Desktop\andfix\andfix -k C:\Users\peiyu_wang\Desktop\andf
ix\andfix.jks -p ccandfix -a cc -e ccandfix
add modified Method:V onClick(Landroid/view/View;) in Class:Lcom/example/andfix/MainActivity;
C:\Users\peiyu_wang\Downloads\apkpatch-1.0.3>

```

补充:

如果有多个 bug, 多个.patch, 可以使用一下命令将.patch 整合

```

usage: apkpatch -m <apatch_path...> -o <output> -k <keystore> -p <***>
-a <alias> -e <***>
-a,--alias <alias>    keystore entry alias.
-e,--epassword <***>  keystore entry password.
-k,--keystore <loc>   keystore path.
-m,--merge <loc...>  path of .apatch files.
-n,--name <name>      patch name.
-o,--out <dir>        output dir.
-p,--kpassword <***>  keystore password.

```

14 因为在程序中判断.patch 文件是否文件, 的文件名是 andfix.apatch, 所以将生成的 patch 文件重新命名为 andfix.apatch, 然后将该文件拷贝到.../Android/data/com.example.andfix/files/文件夹中, 此时重新打开应用点击按钮, 不会 Crash, 已经可以正常弹出 andfix。

15demo 展示结束。