Sophix 介绍与实践

1. 介绍

   移动热修复（Sophix）是阿里云提供的 Android 平台 App 热修复服务方案

2. Sophix 优势

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**

   通过比较可以发现 Sophix 相比其他平台的实现更具优势

   A．对于部分修改，补丁即时生效，不需要应用重启（这个还得看情况，较大的修改或者修改类结构之类的，会导致重启）

   B．补丁包同样采用差异技术，生成的 PATCH 体积小

   C．对应用无侵入，几乎没有性能损耗

   D．傻瓜式接入

   E．能够实现代码修复、资源修复、so 修复等方面，因此更加全面的满足修复要求。

   缺点：

   A．不支持四大组件的修复，这里的修复是新增，或更改。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的权限，这么做会给原先的 App 添加很多臃肿的代码，对 app 的侵入性很强，因此不支持四大组件新增和更改的修复，但是四大组件里面的代码是可以修复的。

3. Sophix 原理简单介绍

   Sophix 的代码修复体系**同时涵盖了底层替换方案和类加载方案**，两种方案的结合，可以实现优势互补，完全兼容的作用，可以灵活的根据实际情况启动切换。在生成补丁时，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，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，虽然及时性不是很好，但是可以达到热

修复目的。

4.阿里巴巴提供的快速接入文档：
https://help.aliyun.com/document_detail/53240.html?spm=5176.doc53238.6.546.T3HWbI

4. 补丁工具下载地址

   Window：
   http://ams-hotfix-repo.oss-cn-shanghai.aliyuncs.com/SophixPatchTool_windows.zip?spm=5176.doc53247.2.2.2ur263&file=SophixPatchTool_windows.zip

   Mac
   http://ams-hotfix-repo.oss-cn-shanghai.aliyuncs.com/SophixPatchTool_macos.zip?spm=5176.doc53247.2.1.2ur263&file=SophixPatchTool_macos.zip

   Linux
   http://ams-hotfix-repo.oss-cn-shanghai.aliyuncs.com/SophixPatchTool_linux.zip?spm=5176.doc53247.2.3.2ur263&file=SophixPatchTool_linux.zip

5. 补丁工具使用文档
   https://help.aliyun.com/document_detail/53247.html?spm=5176.doc53248.6.548.lHabEj

6. 调试工具下载地址
   http://ams-hotfix-repo.oss-cn-shanghai.aliyuncs.com/hotfix_debug_tool-release.apk?spm=5176.doc53247.2.4.2ur263&file=hotfix_debug_tool-release.apk

7. 调试工具使用文档：
   https://help.aliyun.com/document_detail/53248.html?spm=5176.doc53247.6.549.zPEvTh

Demo 实践

在新建项目之前，需要提前在阿里云移动热修复中申请账号，并创建相应的 App。审核通过后再进行剩余操作。

1. 新建 Sophix 工程
2. 在工程中配置 Sophix 依赖

   A. 在根节点下配置阿里云仓库

```
repositories {
    //阿里云仓库
    maven {
        url
"http://maven.aliyun.com/nexus/content/repositories/releases"
    }
}
```

   B. 在依赖(dependencies)中添加

```
    compile 'com.aliyun.ams:alicloud-android-hotfix:3.0.5'
```

C. 本 Demo 为了兼容 Android6.0，增加了 v7 兼容包

```
compile 'com.android.support:appcompat-v7:25.0.1'
```

3. 配置混淆

```
#基线包使用，生成 mapping.txt
-printmapping mapping.txt
#生成的 mapping.txt 在 app/buidl/outputs/mapping/release 路径下，
移动到/app 路径下
#修复后的项目使用，保证混淆结果一致
#-applymapping mapping.txt
#hotfix
-keep class com.taobao.sophix.**{*;}
-keep class com.ta.utdid2.device.**{*;}
#防止 inline
-dontoptimize
```

4. 配置相关权限

```
<!-- 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，调试工具加载本地补丁需要 -->
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

5. 在 Application 节点中添加阿里云热修复服务后台申请的 App id ， App Secrets 和 RSA 密钥

```
    <meta-data
    android:name="com.taobao.android.hotfix.IDSECRET"
    android:value="app id" />
<meta-data
    android:name="com.taobao.android.hotfix.APPSECRET"
    android:value="app secrets" />
<meta-data
android:name="com.taobao.android.hotfix.RSASECRET"
android:value="app rsa 密钥" />
```

6. 建立 Application 的子类，并在 Manifest 中添加。在 Application 子类的 attachBaseContext 方法最前面或者 onCreate 方法最前面完成 SophixManager 的初始化。

```java
@Override
protected void attachBaseContext(Context base) { super.attachBaseContext(base); }
/**
 * 初始化SophixManager
 * 这个方法建议在attachBaseContext方法最前面或onCreate方法最前面执行
 */
private void initSophixManager() {
    // initialize最好放在attachBaseContext最前面
    String appVersion = getAppVersion();
    SophixManager.getInstance().setContext(this)
            .setAppVersion(appVersion)
            .setAesKey("ccandroid2017712")
            .setEnableDebug(false)
            .setEnableFixWhenJit()
            .setPatchLoadStatusStub((mode, code, info, handlePatchVersion) -> {
                // 补丁加载回调通知
                if (code == PatchStatus.CODE_LOAD_SUCCESS) {
                    // 表明补丁加载成功
                    Toast.makeText(SophixApplication.this, "补丁加载成功", Toast.LENGTH_SHORT).show();
                } else if (code == PatchStatus.CODE_LOAD_RELAUNCH) {
                    // 表明新补丁生效需要重启. 开发者可提示用户或者强制重启;
                    // 建议: 用户可以监听进入后台事件, 然后应用自杀
                    Toast.makeText(SophixApplication.this, "补丁需要应用重启才能生效", Toast.LENGTH_SHORT).show();
                    //让进程自杀
                    SophixManager.getInstance().killProcessSafely();
                } else if (code == PatchStatus.CODE_LOAD_FAIL) {
                    // 内部引擎异常, 推荐此时清空本地补丁, 防止失败补丁重复加载
                    SophixManager.getInstance().cleanPatches();
                } else {
                    // 其它错误信息, 查看PatchStatus类说明
                }
            }).initialize();
}
/**
 * 获取当前app的versionName
 *
 * @return versionName
 */
private String getAppVersion() {
    String appVersion;
    try {
        appVersion = this.getPackageManager().getPackageInfo(this.getPackageName(), 0).versionName;
    } catch (Exception e) {
        appVersion = "1.0";
    }
    return appVersion;
}
@Override
public void onCreate() {
    super.onCreate();
    initSophixManager();
}
```

下面是初始化方法中的介绍（来自阿里巴巴 Sohpix 接入文档）

initialize 方法

- initialize(): **<必选>**

  该方法主要做些必要的初始化工作以及如果本地有补丁的话会加载补丁，但不会自动请求补丁。因此需要自行调用 queryAndLoadNewPatch 方法拉取补丁。这个方法调用需要尽可能的早，推荐在 Application 的 onCreate 方法中调用, initialize()方法调用之前你需要先调用如下几个方法，方法调用说明如下:

- setContext(this): **<必选>** Application 上下文 context
- setAppVersion(appVersion): **<必选>** 应用的版本号
- setEnableDebug(true/false): **<可选>** 默认为 false，是否调试模式，调试模式下会输出日志以及不进行补丁签名校验. 线下调试此参数可以设置为 true，查看日志过滤 TAG:Sophix，同时强制不对补丁进行签名校验，所有就算补丁未签名或者签名失败也发现可以加载成功. 但是正式发布该参数必须为 false, false 会对补丁做签名校验，否则就可能存在安全漏洞风险
- setEnableFixWhenJit(): **<可选>** 默认情况下会在 Android N 以后的版本发现 jit 后跳过，这会使得部分 7.0 以上设备不进行修复。而如果想要此时不跳过，需要打开这个选项进行配置。打开后，需要做对 Application 进行改造。要尽可能避免 Application 类与和它同包名的类互相访问，如果确实需要访问，接口应设为 public 权限，详见常见问题文档，也可寻

求群里技术支持解决。Android 7.0 后带来的 jit 问题很隐蔽，只有频繁使用的 app 会由系统进行 jit，该接口可以彻底解决 Android N 带来的 jit 问题。

- setAesKey(aesKey): **<可选>** 用户自定义 aes 秘钥，会对补丁包采用对称加密。这个参数值必须是 16 位数字或字母的组合，是和补丁工具设置里面 AES Key 保持完全一致，补丁才能正确被解密进而加载。此时平台无感知这个秘钥，所以不用担心阿里云移动平台会利用你们的补丁做一些非法的事情。
- setPatchLoadStatusStub(new PatchLoadStatusListener()): **<可选>** 设置 patch 加载状态监听器，该方法参数需要实现 PatchLoadStatusListener 接口，接口说明见 1.3.2.2 说明
- setUnsupportedModel(modelName, sdkVersionInt):**<可选>** 把不支持的设备加入黑名单，加入后不会进行热修复。modelName 为该机型上 Build.MODEL 的值，这个值也可以通过 adb shell getprop | grep ro.product.model 取得。sdkVersionInt 就是该机型的 Android 版本，也就是 Build.VERSION.SDK_INT，若设为 0，则对应该机型所有安卓版本。

PatchLoadStatusListener 接口

该接口需要自行实现并传入 initialize 方法中，补丁加载状态会回调给该接口，参数说明如下:

- mode: 补丁模式, 0:正常请求模式 1:扫码模式 2:本地补丁模式
- code: 补丁加载状态码，详情查看 PatchStatusCode 类说明
- info: 补丁加载详细说明，详情查看 PatchStatusCode 类说明
- handlePatchVersion: 当前处理的补丁版本号, 0:无 -1:本地补丁 其它:后台补丁

这里列举几个常见的 code 码说明,

- code: 1 补丁加载成功
- code: 6 服务端没有最新可用的补丁
- code: 11 RSASECRET 错误，官网中的密钥是否正确请检查
- code: 12 当前应用已经存在一个旧补丁，应用重启尝试加载新补丁
- code: 13 补丁加载失败，导致的原因很多种，比如 UnsatisfiedLinkError 等异常，此时应该严格检查 logcat 异常日志
- code: 16 APPSECRET 错误，官网中的密钥是否正确请检查
- code: 18 一键清除补丁
- code: 19 连续两次 queryAndLoadNewPatch()方法调用不能短于 3s

7. 在接口中可能会调用的方法：

killProcessSafely 方法

可以在 PatchLoadStatusListener 监听到 CODE_LOAD_RELAUNCH 后在合适的时机，调用此方法杀死进程。注意，不可以直接 Process.killProcess(Process.myPid())来杀进程，这样会扰乱 Sophix 的内部状态。因此如果需要杀死进程，建议使用这个方法，它在内部做一些适当处理后才杀死本进程。

cleanPatches()方法

清空本地补丁，并且不再拉取被清空的版本的补丁。

8. **补丁在后台发布之后，并不会主动下行推送到客户端，需要手动调用 queryAndLoadNewPatch 方法查询后台补丁是否可用.**
9. 在本 demo 中，通过按钮点击，调用 **queryAndLoadNewPatch 方法，实际开发视实际情况而定。**
10. 在布局中添加两个按钮，一个点击会 Crash 掉，一个用于 **queryAndLoadNewPatch。**
11. 在 MainActivity 中，完成按钮初始化，并定义点击事件
内容如下：

```
public class MainActivity extends CheckPermissionsActivity
        implements View.OnClickListener {
    Button mbtnBug;
    Button mbtnLoadPatch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mbtnBug = (Button) findViewById(R.id.id_main_btn_bug);
        mbtnLoadPatch= (Button) findViewById(R.id.id_main_load_patch);
        mbtnBug.setOnClickListener(this);
        mbtnLoadPatch.setOnClickListener(this);
    }
    /**
     * 测试mbtnBug点击事件
     *
     * @param v
     */
    @Override
    public void onClick(View v) {
        switch (v.getId()){
            case R.id.id_main_btn_bug:
                //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="sophix";
                if (!str.isEmpty()) {
                    Toast.makeText(MainActivity.this, str, Toast.LENGTH_SHORT).show();
                }
                break;
            case R.id.id_main_load_patch:
                SophixManager.getInstance().queryAndLoadNewPatch();
                break;
            default:
                break;
        }

    }
}
```

再 onClick 方法中，R.id.id_main_btn_bug:下
有问题的版本是：
使用 String str = null;注释: String str="sophix";
修复后的版本是：
使用 String str="sophix";注释: String str = null;

SophixManager.getInstance().queryAndLoadNewPatch()的详细介绍

该方法主要用于查询服务器是否有新的可用补丁. SDK 内部限制连续两次 queryAndLoadNewPatch()方法调用不能短于 3s，否则的话就会报 code:19 的错误码. 如果查询到可用的话，首先下载补丁到本地，然后

- 应用原本没有补丁，那么如果当前应用的补丁是热补丁，那么会立刻加载(不管是冷补丁还是热补丁)．如果当前应用的补丁是冷补丁，那么需要重启生效．
- **应用已经存在一个补丁，请求发现有新补丁后，本次不受影响。并且在下次启动时补丁文件删除，下载并预加载新补丁。在下下次启动时应用新补丁。**

  补丁在后台发布之后，并不会主动下行推送到客户端，需要手动调用 queryAndLoadNewPatch 方法查询后台补丁是否可用．

- 只会下载补丁版本号比当前应用存在的补丁版本号高的补丁，比如当前应用已经下载了补丁版本号为 5 的补丁，那么只有后台发布的补丁版本号>5 才会重新下载．

  同时 1.4.0 以上版本服务后台上线了“一键清除”补丁的功能，所以如果后台点击了“一键清除”那么这个方法将会返回 code:18 的状态码．此时本地补丁将会被强制清除，同时不清除本地补丁版本号

12. 生成两款 Apk，一款由 bug，一款修复了 bug。使用补丁工具生成补丁包
补丁工具使用见：
https://help.aliyun.com/document_detail/53247.html?spm=5176.doc53240.6.548.p42vuJ
13. 上传补丁包，阿里云后台创建的版本要和当前 App 版本一致。
14. 推荐操作：
发布前请严格按照：扫码内测 => 灰度发布 => 全量发布的流程进行，以保证补丁包能够正常在所有 Android 版本的机型上生效。为了保险起见，理论上应该对每个版本的 android 手机都测一遍是否生效会比较好。不过，其实只需测试通过以下具有代表性的 Android 版本就基本没什么大问题了：2.3、4.4、5.1、7.0
15. 调试工具（Apk，需要安装到手机上）
http://ams-hotfix-repo.oss-cn-shanghai.aliyuncs.com/hotfix_debug_tool-release.apk?spm=5176.doc53248.2.1.apER0c&file=hotfix_debug_tool-release.apk
16. 调试工具扫码内测操作流程见
https://help.aliyun.com/document_detail/53248.html?spm=5176.doc53247.6.549.Xbsn42
17. 版本说明情况：

  说明一：patch 是针对客户端具体某个版本的，patch 和具体版本绑定

- eg. 应用当前版本号是 1.1.0，那么只能在后台查询到 1.1.0 版本对应发布的补丁，而查询不到之前 1.0.0 旧版本发布的补丁．

  说明二：针对某个具体版本发布的 5 新补丁，必须包含所有的 bugfix，而不能依赖补丁递增修复的方式，因为应用仅可能加载一个补丁

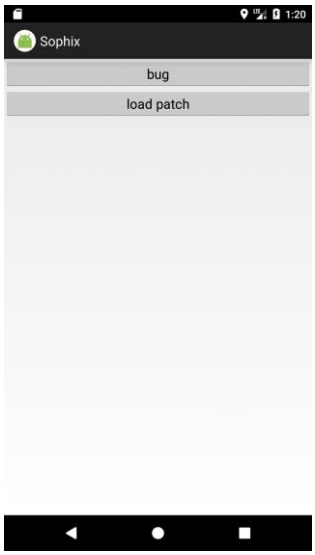
- eg. 针对 1.0.0 版本在后台发布了一个补丁版本号为 1 的补丁修复了 bug1，然后发现此时针对这个版本补丁 1 修复的不完全，代码还有 bug2，在后台重新发布一个补丁版本号为 2 的补丁，那么此时补丁 2 就必须同时包含 bug1 和 bug2 的修复才行，而不是只包含 bug2 的修复(bug1 就没被修复了)

18. 其中 MainActivity 继承的父类 CheckPermissionActivity 主要是进行动态权限的检查。内容如下：

```java
/**
 * 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,
            Manifest.permission.ACCESS_WIFI_STATE,
            Manifest.permission.ACCESS_NETWORK_STATE,
            Manifest.permission.INTERNET,
            };
    /**
     * 子Activity传入的perimission
     */
    protected  String[] mInputPermission;

    private static final int PERMISSON_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;
        }
    }
```

```java
    @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()]),
                    PERMISSON_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;
    }

}
```

19. 运行程序



1.点击第一个按钮 bug，应用会由于 NPE 而 Crash。
2.点击 load patch 查询后台是否有可用补丁，应用会因为补丁需要冷启动才能生效，而进行应用自杀，强制让用户重启。
3.重启应用后，再次点击 bug，NPE 问题已经修复。