转载请注明链接: https://blog.csdn.net/feather_wch/article/details/88368431

插件化前置知识和自定义简易插件框架

版本：2019-03-13(0:10)

> 要先掌握插件化所涉及的到的方方面面的基础知识，才能真正理解差简化、热修复的原理。

Binder机制(3题)

1、Binder是什么?

> 1. 用于进程间通信
> 2. Android中各个app、系统服务运行于不同进程中，需要借助Binder进行通信。
> 3. 从Linux内核角度看，Linux存在进程隔离和虚拟地址空间，导致进程间无法共享数据。所以需要借助IPC

2、Binder机制的步骤分析

> 进程A借助Binder机制获取到进程B的服务结果，步骤分析：
>
> 1. 进程B创建Binder对象
> 2. 进程A接收到进程B的Binder对象
> 3. 进程A利用进程B传来的对象发起请求
> 4. 进程B收到并处理进程A的请求
> 5. 进程A获取到进程B返回的处理结果

3、Binder机制参考资料

> 1. Android IPC
> 2. Binder机制详解
> 3. Android进程间通信详解

hook(11题)

代理

1、代理模式也成为委托模式，分为两种

> 1. 静态代理
> 2. 动态代理

2、代理模式有什么用?

> 1. 限制了对象的访问，对内部对象进行保护
> 2. 如果内部对象的字段发生变化，对于代理类来说，不影响对外的接口，只需要内部进行适配。

静态代理

3、AIDL中的代理模式?代理模式的应用场景?

> 1. AIDL中采用 静态代理
> 2. 进程A获取到服务进程B的Binder对象时, 生成一个代理对象，通过该代理对象直接进行 加减等操作 、
> 3. 给进程A一种假象，获取到了进程B的对象，并进行了操作 。本质是 内部进行了传入数据给B进行处理，最终从B中获取到结果 等一系列操作。

动态代理

4、动态代理是什么？如何实现?

> 1. 动态代理就是 编译阶段 不知道具体代理类，在 运行阶段 才指定了 代理类
> 2. 通过 Java的InvocationHandler类

```
// 1、自定义类实现 InvocationHandler接口
// 2、重写`invoke()方法`
// 3、调用 Proxy.newProxyInstance(classloader, claz[], invocationHandler); 返回一个代理类
```

hook

5、什么是Hook?

> 1. Hook-钩子，是指对一些方法进行拦截，这些方法调用时，能够执行 自定义的代码逻辑
> 2. 是一种 面向切面编程的思想AOP

6、Android中进行Hook的思路

> 1. 找到需要Hook的方法，所属的 系统类
> 2. 利用代理模式来 代理系统类 ，拦截并执行 自定义逻辑
> 3. 利用反射，将 系统类 替换为 代理类

反射

7、Java中的反射机制是什么?

> 1. 在运行状态中，对于任意一个类，都能知道 类的所有属性和方法
> 2. 并且能够调用其方法或者获取其属性

【实例】Hook剪切板服务

8、利用Hook实现剪切板服务的复制和粘贴，粘贴的内容都是我们指定的内容。

剪切板服务的使用

9、剪切板服务是如何使用的?

> 点击按钮后将自定义内容，放入到系统剪切板中。

```
mEditText = findViewById(R.id.clipboard_edittext);
findViewById(R.id.clipboard_btn).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        // 1、获取到剪切板服务
        ClipboardManager clipboardManager = (ClipboardManager) getSystemService(Context
        // 2、创建String类型的clipData
        ClipData clipData = ClipData.newPlainText("label", mEditText.getText().toString
        // 3、将ClipData内容放到系统剪切板里面
        clipboardManager.setPrimaryClip(clipData);
    }
});
```

剪切板的原理

10、获得系统剪切板服务的源码流程

> 本质是从 `ServiceManager` 的 `sCache(HashMap)` 获取到 剪切板服务的`Binder`对象(实现了`IBinder`接口)

```java
// Activity.java-getSystemService(Context.CLIPBOARD_SERVICE)
public Object getSystemService(String name) {
    // 1、WindowManager
    if (WINDOW_SERVICE.equals(name)) {
        return mWindowManager;
    }
    // 2、搜索服务Manager
    else if (SEARCH_SERVICE.equals(name)) {
        ensureSearchManager();
        return mSearchManager;
    }
    // 3、进一步寻找“剪切板服务”
    return super.getSystemService(name);
}

// ContextThemeWrapper.java
public Object getSystemService(String name) {
    // 1、布局加载服务
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        mInflater = LayoutInflater.from(getBaseContext()).cloneInContext(this);
        return mInflater;
    }
    // 2、进一步寻找“剪切板服务”
    return getBaseContext().getSystemService(name);
}

// ContextImpl.java
public Object getSystemService(String name) {
    return SystemServiceRegistry.getSystemService(this, name);
}

// SystemServiceRegistry.java - 从HashMap中取出
private static final HashMap<String, ServiceFetcher<?>> SYSTEM_SERVICE_FETCHERS = new HashM
public static Object getSystemService(ContextImpl ctx, String name) {
    ServiceFetcher<?> fetcher = SYSTEM_SERVICE_FETCHERS.get(name);
    return fetcher != null ? fetcher.getService(ctx) : null;
}

/**=========================================
 * 问题：HashMap(SYSTEM_SERVICE_FETCHERS)是如何初始化的？ClipBoard服务是何时放入该HashMap中的
 *   1. 调用registerService方法对剪切板服务进行注册
 * // SystemServiceRegistry.java
 *=========================================*/
registerService(Context.CLIPBOARD_SERVICE, ClipboardManager.class,new CachedServiceFetcher<
    @Override
    public ClipboardManager createService(ContextImpl ctx) throws ServiceNotFoundException
        // 1、创建剪切板管理器
        return new ClipboardManager(ctx.getOuterContext(), ctx.mMainThread.getHandler());
    }});
// SystemServiceRegistry.java-将剪切板服务放入到SYSTEM_SERVICE_FETCHERS中
private static <T> void registerService(String serviceName, Class<T> serviceClass, ServiceF
    SYSTEM_SERVICE_NAMES.put(serviceClass, serviceName);
    SYSTEM_SERVICE_FETCHERS.put(serviceName, serviceFetcher);
}
```

```
// ClipboardManager.java - 最终通过ServiceManager获取到目标服务
private final IClipboard mService;
public ClipboardManager(Context context, Handler handler) throws ServiceNotFoundException {
    // xxx
    // 1、从ServiceManager内部的缓存的HashMap中找，有就返回该IClipboard的Binder对象，要么创建新
    mService = IClipboard.Stub.asInterface(ServiceManager.getServiceOrThrow(Context.CLIPBOA
}

// ServiceManager.java
public static IBinder getServiceOrThrow(String name) throws ServiceNotFoundException {
    final IBinder binder = getService(name);
    if (binder != null) {
        return binder;
    } else{
        throw new ServiceNotFoundException(name);
    }
}
// ServiceManager.java
private static HashMap<String, IBinder> sCache = new HashMap<String, IBinder>();
public static IBinder getService(String name) {
    // 1、返回HashMap-sCache中剪切服务对应的IBinder
    IBinder service = sCache.get(name);
    if (service != null) {
        return service;
    }
    return null;
}

// ServiceManager.java - 得到 ClipBoard的Binder對象的代理對象
public static android.content.IClipboard asInterface(android.os.IBinder obj) {
    if ((obj == null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); // Hook点
    if (((iin != null) && (iin instanceof android.content.IClipboard))) {
        return ((android.content.IClipboard) iin);
    }
    return new android.content.IClipboard.Stub.Proxy(obj);
}
```

Hook剪切板服务

11、Hook剪切板服务的流程和实现代码

> 1. 获取到剪切板服务(ClipboardManager)的代理Binder对象
> 2. 将其存入到HashMap中。
> 3. 该Binder对象的剪切板操作的方法逻辑按照我们的需求进行修改。

```
/**====================
 * 1、Hook剪切板服务
 *======================*/
public static void hookClipboard() throws ClassNotFoundException, NoSuchMethodException, Ir
    // 1. 获取系统的ServiceManager
    Class<?> serviceManagerClass = Class.forName("android.os.ServiceManager");
    // 2. 获得ServiceManager的getService方法
    Method getServiceMethod = serviceManagerClass.getMethod("getService", String.class);
    // 3. 通过该方法获取到原系统ClipBoard服务
    IBinder binder = (IBinder) getServiceMethod.invoke(null, Context.CLIPBOARD_SERVICE);
    /**============================================
     * 4. 动态代理，创建自己的代理对象。
     *    1) 将为Clipboard的IBinder对象提供代理，该代理类MyClipboardProxy会实现IBinder的接口
     *    2) 调用queryLocalInterface()方法时，获取到我们剪切板的代理Binder对象
     *    3) 该方法就是生成一个新类，该类实现了IBinder的接口。
     *============================================*/
    IBinder myBinder = (IBinder) Proxy.newProxyInstance(serviceManagerClass.getClassLoader(
    // 5. 拿到ServiceManager中的缓存IBinder数组
    Field field = serviceManagerClass.getDeclaredField("sCache");
    field.setAccessible(true);
    Map<String, IBinder> map = (Map<String, IBinder>) field.get(null);
    // 6. 将我们自定义的ClipboardManager放入Map中
    map.put(Context.CLIPBOARD_SERVICE, myBinder);
}

public static class MyClipboardProxy implements InvocationHandler{

    private final IBinder mSystemBinder;

    public MyClipboardProxy(IBinder iBinder){
        mSystemBinder = iBinder;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 1、拦截原系统类的查询本地是否有“剪切服务代理类”的方法
        if("queryLocalInterface".equals(method.getName())){
            // 2. 拿到系统的Stub
            Class<?> mStubClass = Class.forName("android.content.IClipboard$Stub");
            // 3. 拿到系统的Clipboard本地类
            Class<?> mIClipboard = Class.forName("android.content.IClipboard");
            /**==================================================================
             * 4. 将其代理为我的Clipboard。该类实现了IClipboard所有剪切板操作的接口。获取剪切板
             *==================================================================*/
            return Proxy.newProxyInstance(mIClipboard.getClassLoader(), new Class[]{mIClipb
        }
        // 5. 其余方法走原系统的执行
        return method.invoke(mSystemBinder, args);
    }
}

public static class MyClipboard implements InvocationHandler{

    private Object mBase;
```

```
        public MyClipboard(IBinder systemBinder, Class stub){
            // 拿到asInteface方法，因为源码中执行了这一句，我们也要执行这一句
            try {
                Method asInterface = stub.getDeclaredMethod("asInterface", IBinder.class);
                mBase = asInterface.invoke(null, systemBinder);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            if("setPrimaryClip".equals(method.getName())){
                if(args.length > 0){
                    if(args[0] instanceof ClipData){
                        ClipData clipData = (ClipData) args[0];
                        if(clipData.getItemCount() > 0){
                            // 1、获取到原文本，附加上修改的内容
                            String srcText = clipData.getItemAt(0).getText().toString();
                            if(!TextUtils.isEmpty(srcText)){
                                // 2、将参数替换为新的ClipData，继续执行原逻辑的方法
                                args[0] = ClipData.newPlainText("label", srcText + "---转自猎羽
                            }
                        }
                    }
                }
            }
            // 3、当前和其他方法还是交给系统原逻辑处理
            return method.invoke(mBase, args);
        }
    }
```

> 调用

```
        try {
            hookClipboard();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
```

Proxy.newProxyInstance

反射调用

ClassLoader和dex加载过程(4题)

1、Android中具有两个主要的ClassLoader

> 1. PathClassLoader
> 2. DexClassLoader
> 3. 这两者都继承自 `BaseDexClassLoader`

PathClassLoader

2、PathClassLoader的作用

> 1. Android系统通过 `PathClassLoader` 加载 系统类 和 主Dex中的类
> 2. 使用继承自 `BaseDexClassLoader` 的 `findClass()`进行加载
> 3. 只能加载系统中已经安装过的apk

DexClassLoader

3、DexClassLoader的作用

> 1. 可以用来从未安装.jar和.apk文件中加载class。
> 2. 使用继承自 `BaseDexClassLoader` 的 `findClass()`进行加载

4、PathClassLoader和DexClassLoader的区别

> 1. DexClassLoader：能够从未安装的jar/apk加载class
> 2. PathClassLoader：只能从系统中已经安装过的apk加载class

四大组件(25题)

1、Activity必须在AndroidManifest中进行注册，如何绕开不注册就抛出异常的问题?

2、插件化如何管理插件中四大组件的生命周期?

3、如何加载插件的apk?

Instrumentation

4、Instrumentation的作用是: 监控应用与系统相关的交互行为

5、相关的方法:

> 1. mInstrumentation.execStartActivity(): 启动Activity
> 2. mInstrumentation.newActivity(): 使用类加载器创建Activity对象

3. mInstrumentation.callActivityOnCreate()/callActivityOnResume()等等生命周期的方法: 生命周期

LoadedApk

6、LoadedApk的作用

1. LoadedApk对象是APK文件在内存中的表示。 Apk文件的相关信息，ApplicationInfo、主线程、包名、Application对象都是局部变量。
2. 具有创建Application对象的方法。
3. 具有内部类：LoadedApk.ServiceDispatcher
4. 具有内部类: LoadedApk.ReceiverDispatcher

7、LoadedApk内部类ServiceDispatcher的作用?

1. LoadedApk.java的内部类ServiceDispatcher
2. 如何让远程服务端调用客户端的ServiceConnection中的方法?
3. 需要借助Binder才能让远程服务端回调自己的方法
4. ServiceDispatcher的内部类InnerConnection就起到了Binder的作用

8、LoadedApk内部类ReceiverDispatcher的作用?

LoadedApk.ReceiverDispatcher.InnerReceiver起到了Binder的作用

9、相关方法

1. makeApplication()：创建Application对象，该对象唯一，不会重复创建。

如何加载插件apk

构造LoadedApk修改ClassLoader

10、如何加载插件Apk方案1(Hook思想，DroidPlugin框架实现方法)

1-加载插件apk需要一个ClassLoader
2-系统的ClassLoader通过LoadedApk对象获得。需要构建LoadedApk，修改ClassLoader对象，通过Hook方法将自我构建的LoadedApk添加到ActivityThread的mPackages(HashMap)中。

```
final ArrayMap<String, WeakReference<LoadedApk>> mPackages = new ArrayMap<>();
```

3-该方案除了需要Hook LoadedApk，在构造LoadedApk中还用到ApplicationInfo对象，需要进行Hook
4-ApplicaitonInfo的解析还涉及到手动解析插件中的AndroidManifest文件，过程复杂。

借助系统的ClassLoader

11、如何加载插件Apk方案2

> 1. 借助系统的ClassLoader
> 2. 将插件Apk的信息告诉ClassLoader，让系统帮我们加载
> 3. 将插件apk的dex文件，插入到DexPathList类中的 `dexElements数组` 即可

如何解决插件中Activity未注册的问题?

12、如何解决插件中Activity未注册的问题?

> 1. Hook startActivity()，借助占坑的方法。去加载傀儡Activity。
> 2. 在AndroidManifest中注册一个傀儡Activity，骗过系统层的校验
> 3. 真正创建Activity时，Hook拦截Activity的创建方法，去创建真正的插件apk中的Activity

13、流程分析

```
// 1、创建一个属于插件的ClassLoader，传入插件Apk的路径。父类加载器是宿主的类加载器。
String cachePath = MainActivity.this.getCacheDir().getAbsolutePath();          // dex优化
String apkPath = Environment.getExternalStorageDirectory().getAbsolutePath() + "/plugin
DexClassLoader mClassLoader = new DexClassLoader(apkPath, cachePath, cachePath, getClas
// 2、将宿主apk和插件apk的dex文件加入到新建的dexElements数组中
PathClassLoader pathLoader = (PathClassLoader) getApplicationContext().getClassLoader()
// 3、将数组设置给宿主的DexPathList对象
// 4、拦截Activity的启动。需要去代理AMS的代理类(Binder机制，只能处理AMS的代理类)
// 5、系统检查完Activity合法性后，拦截系统创建Activity的方法，将傀儡Activity替换为我们需要的
```

启动流程

14、启动流程

```
//Instrumentation.java
public ActivityResult execStartActivity(Context who, IBinder contextThread, IBinder token, Acti
    int result = ActivityManager.getService() //Binder对象
                .startActivity(whoThread, ... ,options);
    // xxx
}
// ActivityManager.java - AMS的代理类
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}
// ActivityManager.java - 单例模式
private static final Singleton<IActivityManager> IActivityManagerSingleton = new Singleton<IAct
                @Override
                protected IActivityManager create() {
                    final IBinder b = ServiceManager.getService(Context.ACTIVITY_SERVICE);
                    final IActivityManager am = IActivityManager.Stub.asInterface(b);
                    return am;
                }
};
```

15、Hook AMS的代理类

> 1-Hook AMS的代理类，就是代理 `ActivityManager.getService()` 的返回值就可以了
> 2-获得AMS的代理类ActivityManager

```
        // 1、获取ActivityManager类
        Class<?> activityManagerClass = Class.forName("android.app.ActivityManager");

        // 2、取出getService方法
        Method getServiceMethod = activityManagerClass.getDeclaredMethod("getService", null);

        // 3、调用getService，获得Singleton<IActivityManager> IActivityManagerSingleton
        Object activityManagerSingleton = getServiceMethod.invoke(null, null);

        // 4、获得AMS的代理对象
        Class<?> singleton = Class.forName("android.util.Singleton"); // 是一个 android.util.Si
        Field mInstanceField = singleton.getDeclaredField("mInstance"); // 取出mInstance字段
        mInstanceField.setAccessible(true);
        Object activityManager = mInstanceField.get(activityManagerSingleton);
```

> 3-Hook IActivityManager的startActivity方法

ActivityManager

16、ActivityManager是AMS的代理类

为什么占坑Activity的生命周期能给予真正的Activity?

17、为什么占坑Activity的生命周期能给予真正的Activity?

1. AMS与ActivityThread之间对于Activity的生命周期的交互，并没有直接使用Activity对象进行交互
2. 而是使用一个token来标识，这个token是binder对象，因此可以方便地跨进程传递。
3. Activity里面有一个成员变量mToken代表的就是它，token可以唯一地标识一个Activity对象
4. 只替换了要启动Activity的信息，并没有替换token，所以系统并不知道当前运行的已经是真正的Activity

【实例】加载插件的Activity

18、为什么需要使用宿主的ClassLoader作为我们的DexClassLoader的父加载器

1. JVM提供了三种类加载器，还有用户自定义的类加载器。这些加载器之间的层次关系被称为 类加载器的双亲委派模型。
2. 该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过 组合（`Composition`）关系 来实现，而不是通过继承（Inheritance）。
3. 双亲委派，从父类加载器层层加载，防止出现自定义类加载器也有一个Object类，和系统的Object类出现混乱。

19. Hook方案中startActivity是如何将目标Activity替换为占坑Activity的?

1. 只需要将参数中的Intent的ComponentName替换为占坑Activity即可
2. 并且将原来的ComponentName，通过Intent.putExtra()对其进行保存。

20、Hook方案中系统检查完Activity的合法性后，又是如何将占坑Activity恢复为目标Activity的?

1. 根据Activity的启动流程，最终是在 `ActivityThread`的`Handler H` 中处理的消息。

Application中进行Hook

21、在自定义Application进行插件的加载、Hook Activity的启动流程

1. 尽早进行处理，可以在Activity或者Application的 `attachBaseContext()` 中进行处理
2. 这里在Application中进行处理

```
public class HostApplication extends Application{

    /**
     * 尽早进行插件的加载。可以在Application中也可以在Activity的attachBaseContext()方法中调用
     */
    @Override
    protected void attachBaseContext(Context newBase) {
        super.attachBaseContext(newBase);
        try {
            // 1. 加载插件apk(将插件apk中所有dex文件放入到宿主的pathList中的dexElements中)
            loadPluginApk("plugin.apk");
            // 2. Hook ActivityManager: 将插件apk的目标Activity，替换为占坑Activity
            hookActivityManager();
            // 3. Hook ActivityThread中的Handler H。将占坑Acitivty，替换为目标Activity。
            hookActivityThreadHandlerH();
            // 4. 创建属于插件的资源库【这里是资源加载部分所需要的】
            preparePluginResources("plugin.apk");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

> AndroidManifest.xml: 1. 指定Application 2. 提前注册占坑Activity

```
<manifest package="com.hao.featherpluginhost">

    <application
        android:name=".HostApplication"
        xxx>
        <activity android:name=".HostActivity">xxx</activity>
        <activity android:name=".PitActivity"></activity>
    </application>

</manifest>
```

加载插件Apk

22、加载插件Apk

```
/**================================================
 * @function: 加载插件apk到宿主中
 *  1. 下载Apk到宿主应用的数据目录
 *  2. 加载插件Apk的Dex文件
 *================================================*/
private void loadPluginApk(String pluginApkName) throws ClassNotFoundException, NoSuchFiel

    /**=======================================================
     * 0、模拟下载插件的过程。
     *    1. 从assets/external中将插件apk，移动到，app的数据目录下
     *=======================================================*/
    // 1. apk名
    String apkName = pluginApkName;
    // 2. apk目前位于assets/external/路径中
    String apkAssetsPath = "external" + File.separator + apkName;
    // 3、如果/data/data/webview.apk文件已经存在，需要删除后重新复制(模拟下载流程)
    String pluginFilePath = getFilesDir().getAbsolutePath() + File.separator + apkName;
    File pluginFile = new File(pluginFilePath);
    if(!pluginFile.exists()){
        // 4、不存在插件apk时，将assets/external中的文件复制到app的文件下
        copyAssetsFileToAppFiles(apkAssetsPath, apkName);
    }
    // 5、安装。如果数据目录下不存在该插件apk文件，报错。
    if(!pluginFile.exists()){
        Log.e("feather", "Download plugin failed!");
        return;
    }

    /**=======================================================
     * 1、获取到插件Apk的路径
     *    1. 可以从外部存储获取
     *    2. 可以从网络上下载到本地，然后从这个本地地址中获取【本次是模拟网络下载的过程】
     *=======================================================*/
    String pluginApkPath = pluginFilePath;
    // String pluginApkPath = Environment.getExternalStorageDirectory().getAbsolutePath() +

    /**=======================================================
     * 2、创建我们的DexClassLoader来加载“插件Apk的dex文件”
     *    1. 可以从外部存储获取
     *    2. 可以从assets\plugin中获取【就从此处取，毕竟是Demo】
     *    3. 可以从网络上下载到本地，然后从这个本地地址中获取
     *=======================================================*/
    String cachePath = getCacheDir().getAbsolutePath(); // 获取到缓存目录
    DexClassLoader pluginDexClassLoader = new DexClassLoader(pluginApkPath, // apk or jar包
            cachePath,  // optimizedDirectory，经过优化的dex文件（odex）文件输出目录。
            cachePath,  // librarySearchPath，包含native library的目录。（将被添加到app动态库
            getClassLoader()); // 用宿主的ClassLoader作为父类加载器

    /**=======================================================
     * 3、加载“插件Apk”的Dex文件(一): 获取宿主和插件的Dex路径列表: DexPathList pathList
     *    1. 可以从外部存储获取
     *    2. 可以从assets\plugin中获取【就从此处取，毕竟是Demo】
     *    3. 可以从网络上下载到本地，然后从这个本地地址中获取
     *=======================================================*/
```

```
    // 1、拿到宿主的ClassLoader
    PathClassLoader hostPathLoader = (PathClassLoader) getClassLoader();
    // 2、反射出BaseDexClassLoader的字段pathList(DexPathList)
    Class<?> baseDexClassLoaderClass = Class.forName("dalvik.system.BaseDexClassLoader");
    Field pathListField = baseDexClassLoaderClass.getDeclaredField("pathList"); // 具有成员
    pathListField.setAccessible(true);
    // 3、获取到宿主的pathList
    Object hostPathList = pathListField.get(hostPathLoader); // 从【宿主】的类加载器中，获取
    // 4、获取到插件的pathList
    Object pluginPathList = pathListField.get(pluginDexClassLoader); // 从【插件】的类加载器

    /**=======================================================
     * 4、加载“插件Apk”的Dex文件(二): 合并宿主和插件的pathList中的dexElements数组
     *=======================================================*/
    // 1、获取到DexPathList的字段dexElements
    Field dexElementsField = hostPathList.getClass().getDeclaredField("dexElements");
    dexElementsField.setAccessible(true);
    // 2、获取到【宿主】的dex数组
    Object hostDexElements = dexElementsField.get(hostPathList);
    // 3、获取到【插件】的dex数组
    Object pluginDexElements = dexElementsField.get(pluginPathList);
    // 4、合并数组
    Class<?> localClass = hostDexElements.getClass().getComponentType(); //获取Host数组类型
    int hostArrayLength = Array.getLength(hostDexElements); //获取Host数组长度
    int newArrayLength = hostArrayLength + Array.getLength(pluginDexElements); //宿主数组si
    Object newDexElements = Array.newInstance(localClass, newArrayLength); //创建新的数组
    for (int i = 0; i < newArrayLength; ++i) { //将宿主和插件的dex文件设置到新数组中
        if (i < hostArrayLength) {
            Array.set(newDexElements, i, Array.get(hostDexElements, i)); // 先存入宿主的数组
        } else {
            Array.set(newDexElements, i, Array.get(pluginDexElements, i - hostArrayLength))
        }
    }

    /**=======================================================
     * 5、加载“插件Apk”的Dex文件(三): 将新的Dex数组放入到【宿主】的pathList中
     *=======================================================*/
    dexElementsField.set(hostPathList, newDexElements);
}

/**=============================================================================
 * @function 将“assets/external”下的外置插件复制到宿主app的数据目录中，模拟外置插件的下载
 * @param assetFileName assets目录中的文件路径名，例如"assets/external/webview.apk"中的"exter
 * @param newFileName 插件名如"webview.apk"
 *=============================================================================*/
private void copyAssetsFileToAppFiles(String assetFileName, String newFileName){

    // 1、assets/external中的外置插件apk作为输入
    try(InputStream inputStream = this.getAssets().open(assetFileName);
        // 2、宿主appp数据目录作为输出
        FileOutputStream fileOutputStream = this.openFileOutput(newFileName, MODE_PRIVATE))

        // 3、IO进行文件的复制
        int byteCount = 0;
        byte[] buffer = new byte[1024];
```

```
            while((byteCount = inputStream.read(buffer)) != -1){
                fileOutputStream.write(buffer, 0, byteCount);
            }
            // 4、刷新输出流
            fileOutputStream.flush();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

目标Activity替换为占坑Activity: Hook ActivityManager

23、Hook ActivityManager将目标Activity替换为占坑Activity

> 1. 替换 `IActivityManagerSingleton` 中的IActivityManager为我们的代理类。

```
public void hookActivityManager() throws ClassNotFoundException, NoSuchMethodException, InvocationTarge
    // 1、获取ActivityManager类
    Class<?> activityManagerClass = Class.forName("android.app.ActivityManager");
    // 2、取出IActivityManagerSingleton
    Field iActivityManagerSingletonField = activityManagerClass.getDeclaredField("IActivityManagerSingl
    iActivityManagerSingletonField.setAccessible(true);
    Object activityManagerSingleton = iActivityManagerSingletonField.get(null);
    // 3、获得AMS的代理对象
    Class<?> singleton = Class.forName("android.util.Singleton"); // 是一个 android.util.Singleton对象;
    Field mInstanceField = singleton.getDeclaredField("mInstance"); // 取出mInstance字段
    mInstanceField.setAccessible(true);
    Object activityManager = mInstanceField.get(activityManagerSingleton);
    // 4、创建AMS的代理对象的【代理对象】
    Class<?> iActivityManagerInterface = Class.forName("android.app.IActivityManager");
    Object proxyActivityManager = Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader()
            new Class<?>[] { iActivityManagerInterface }, new IActivityManagerInvocationHandler(activit
    // 5、将IActivityManagerSingleton的“mInstace字段”替换为"ActivityManager的代理对象"，进行我们需要的将插件
    mInstanceField.set(activityManagerSingleton, proxyActivityManager);
}

public static final String EXTRA_TARGET_INTENT = "extra_target_intent";

/**=========================================================
 * @功能: 用于拦截ActivityManager的startActivity操作
 *=========================================================*/
public class IActivityManagerInvocationHandler implements InvocationHandler {
    private static final String TAG = "feather";

    Object mActivityManager;
    public IActivityManagerInvocationHandler(Object activityManager) {
        mActivityManager = activityManager;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

        // 1、startActivity以外的操作直接透传给“原ActivityManager”处理
        if (!"startActivity".equals(method.getName())) {
            return method.invoke(mActivityManager, args);
        }else{
            // 2、拦截startActivity方法
            Log.e(TAG,"startActivity方法拦截了");

            // 3、找到参数里面的第一个Intent对象。
            Intent intent = null;
            for (Object arg : args) {
                if (arg instanceof Intent) {
                    intent = (Intent) arg;

                    Log.e(TAG,"原Intent = " + intent);
                    /**=====================================
                     * 4、获取到原Activity的信息，并以key-value形式保存到Intent的extra中
                     *====================================*/
                    // 1. 获取到原componentName(原Activity信息)
```

```
                    ComponentName rawComponentName = intent.getComponent();
                    // 2. 保存到Extra中
                    intent.putExtra(EXTRA_TARGET_INTENT, rawComponentName); // "extra_target_intent"
                    /**=====================================
                     * 5、将目标Activity替换为占坑Activity
                     *====================================*/
                    // 1. 宿主apk的"包名"
                    String stubPackage = getPackageName();
                    // 2. 以宿主包名 + 占坑Activity的信息构建ComponentName
                    ComponentName componentName = new ComponentName(stubPackage, PitActivity.class.getN
                    // 3. 替换Component
                    intent.setComponent(componentName);

                    Log.e(TAG,"新Intent = " + intent);
                    break;
                }
            }
            Log.e(TAG,"startActivity方法 hook 成功");
            // 4、以修改后的intent参数去startActivity
            return method.invoke(mActivityManager, args);
        }
    }
}
```

占坑Activity还原为目标Activity: Hook ActivityThread的Handler H

24、占坑Activity还原为目标Activity: Hook ActivityThread的Handler H

```
/**=======================================================
 * @function 通过系统验证后，将占坑Activity换回目标Activity
 *  1. 给ActivityThead的Handler H设置一个回调接口Callback
 *  2. 在Handler调用handleMessage()前被我们的Callback进行拦截处理
 *=======================================================*/
public static void hookActivityThreadHandlerH() throws Exception {

    // 1、获取到ActivityThread类
    Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
    // 2、调用“currentActivityThread()”获取到ActivityThread对象
    Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread")
    currentActivityThreadMethod.setAccessible(true);
    Object activityThread = currentActivityThreadMethod.invoke(null);
    // 3、获取到ActivityThread对象的字段mH(Handler H)
    Field mHField = activityThreadClass.getDeclaredField("mH");
    mHField.setAccessible(true);
    Handler mH = (Handler) mHField.get(activityThread);
    /**=========================================================
     * 4、根据消息机制，给mH设置Callback，该Callback优先于handlerMessage()执行。在activity启动的流程中，替换
     *   1. Handler具有成员变量mCallback
     *   2. 更改mH的mCallback，为我们自定义的Handler Callback
     *=========================================================*/
    Field mCallbackField = Handler.class.getDeclaredField("mCallback");
    mCallbackField.setAccessible(true);
    mCallbackField.set(mH, new ActivityThreadHandlerHCallback(mH));
}

/**=======================================
 * @功能: 自定义Callback，重点处理Activity
 *  1. 将系统返回的Intent中的ComponentName替换为原来的内容
 *=======================================*/
public static class ActivityThreadHandlerHCallback implements Handler.Callback {
    // 和源码一致，按道理需要反射获得
    private static final int EXECUTE_TRANSACTION = 159;

    Handler mHandlerH;
    public ActivityThreadHandlerHCallback(Handler rawHandler){
        mHandlerH = rawHandler;
    }
    @Override
    public boolean handleMessage(Message msg) {
        switch (msg.what){
            case EXECUTE_TRANSACTION:
                try {
                    // 1、msg.obj需要是ClientTransaction的对象
                    Class<?> clientTransactionClass = Class.forName("android.app.servertransaction.Clie
                    if(clientTransactionClass.isInstance(msg.obj)){
                        // 2、从ClientTransaction对象中取出: List<ClientTransactionItem> mActivityCallba
                        Field mActivityCallbacksField = clientTransactionClass.getDeclaredField("mActiv
                        mActivityCallbacksField.setAccessible(true);
                        List mActivityCallbacks = (List) mActivityCallbacksField.get(msg.obj);

                        // 3、遍历Callbacks列表，取出LaunchActivityItem。
                        for (Object callback : mActivityCallbacks) {
```

```
                            // 4、如果Callback是LaunchActivityItem类型的，表明是startActivity的后续处理
                            Class<?> launchActivityItemClass = Class.forName("android.app.servertransac
                            if(launchActivityItemClass.isInstance(callback)){
                                // 5、获取到Intent
                                Field mIntentField = launchActivityItemClass.getDeclaredField("mIntent"
                                mIntentField.setAccessible(true);
                                Intent mIntent = (Intent) mIntentField.get(callback);
                                // 6、获取到之前缓存的ComponentName，并放回Intent
                                ComponentName componentName = mIntent.getParcelableExtra(EXTRA_TARGET_I
                                if(componentName != null){
                                    mIntent.setComponent(componentName);
                                }
                                // 7、将LaunchActivityItem的成员变量mIntent进行替换
                                mIntentField.set(callback, mIntent);
                            }
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
            }
            // 8、交给ActivityThread里面的“Handler H”进行处理。
            mHandlerH.handleMessage(msg);
            // 9、不再需要后续的处理，避免再去执行Handler的handleMessage(导致重复处理)
            return true;
        }
    }
```

跳转到插件Activity

25、宿主Activity中跳转到插件的Activity

> 1. 利用Intent跳转到插件Activity
> 2. 一定要使用ComponentName进行跳转

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_host);

    // 打开插件的Activity
    findViewById(R.id.login_btn).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent();
            intent.setComponent(new ComponentName("com.hao.featherpluginlogin", "com.hao.fe
            startActivity(intent);
        }
    });
}
```

资源加载

1、插件Apk的资源问题

> 1. 宿主中的LoadedApk存储了宿主Apk的相关信息，包括资源。
> 2. 通过宿主和插件的dex合并，能够让宿主跳转到插件的Activity
> 3. 但是合并之后，插件Activity调用原插件的资源，确实用的是宿主的资源库，会出现找不到该资源，导致崩溃。

Context.getResources()

2、无论是通过R文件还是代码中Context.getResources()方法，本质都是通过 `Context.getResources()` 实现的

```
// ContextImpl.java
public Resources getResources() {
    return mResources;
}
// ContextImpl.java - ResoucesManager是一个单例
ResourcesManager mResourcesManager = ResourcesManager.getInstance();
Resources mResources;
```

3、让插件Apk资源包生效的思路

> 1. 构造自己的AssetManager，调用addAssetPath添加插件的资源路径
> 2. 构造属于该插件的Resources
> 3. 返回给插件Activity

【实例】加载插件资源

4、实例解决如何不会因为插件Activity加载插件Resources导致崩溃

> 1. 在宿主创建属于插件的AssetManager和Resources
> 2. 重写宿主Application的getAssets()/getResources()方法
> 3. 插件Activity中去获取宿主Application的AssetManager和Resources(均包含插件资源)，而不是使用宿主资源库。最终解决该问题。
>
> - 宿主Application: 构建属于插件的AssetManager和Resources

```
protected void attachBaseContext(Context newBase) {
    super.attachBaseContext(newBase);
    // ........省略其他Hook操作........
    // 1、创建属于插件的资源库
    preparePluginResources("plugin.apk");
}

AssetManager mPluginAssetManager; // 创建属于插件的AssetManager
Resources mPluginResources; // 创建属于插件的Resources
Resources.Theme mPluginTheme; // 创建属于插件的Theme
/**==========================================
 * @功能: 创建属于插件的AssetManager和Resources
 *  1. 通过addAssetPath将插件资源包添加到AssetManager中
 *  2. 通过创建的插件AssetManager来创建插件Resources
 *==========================================*/
private void preparePluginResources(String pluginApkName) throws InstantiationException, I
    String pluginFilePath = getFilesDir().getAbsolutePath() + File.separator + pluginApkNam
    File pluginFile = new File(pluginFilePath);
    if(!pluginFile.exists()){
        Log.e("feather", "插件Apk不存在！");
    }
    // 1、创建新的AssetManager
    mPluginAssetManager = AssetManager.class.newInstance();
    // 2、调用AssetManager的addAssetPath将插件的路径添加进去
    Method addAssetPathMethod = mPluginAssetManager.getClass().getDeclaredMethod("addAssetP
    addAssetPathMethod.setAccessible(true);
    addAssetPathMethod.invoke(mPluginAssetManager, pluginFilePath);
    // 3、创建属于插件的Resources
    Resources hostResources = getResources();
    mPluginResources = new Resources(mPluginAssetManager, hostResources.getDisplayMetrics()
    // 4、创建属于插件的Theme
    mPluginTheme = mPluginResources.newTheme();
    mPluginTheme.setTo(super.getTheme());
}
@Override
public Resources getResources() {
    return mPluginResources == null ? super.getResources() : mPluginResources;
}
@Override
public AssetManager getAssets() {
    return mPluginAssetManager == null ? super.getAssets() : mPluginAssetManager;
}
@Override
public Resources.Theme getTheme() {
    return mPluginTheme == null ? super.getTheme() : mPluginTheme;
}
```

插件Activity重构getAssets()/getResources()

5、插件的Activity使用默认的获取资源方法，会获取到宿主的资源库，导致加载插件中的资源都会报错，需要让插件去获取插件的资源库。

> 1. 重写getAssets()
> 2. 重写getResources()
> 3. 重写getTheme()

```
public class LoginActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    // 1、获取到宿主Application的插件AssetManager
    @Override
    public AssetManager getAssets() {
        if(getApplication() != null && getApplication().getAssets() != null){
            return getApplication().getAssets();
        }
        return super.getAssets();
    }
    // 2、获取到宿主Application的插件esources
    @Override
    public Resources getResources() {
        if(getApplication() != null && getApplication().getResources() != null){
            return getApplication().getResources();
        }
        return super.getResources();
    }
    // 3、获取到宿主Application的插件Theme
    @Override
    public Resources.Theme getTheme() {
        if(getApplication() != null && getApplication().getTheme() != null){
            return getApplication().getTheme();
        }
        return super.getTheme();
    }
}
```

知识扩展

1、getDeclaredField()和getField()的区别?

参考资料

1. 插件化框架学习系列博客