插件化

前言

插件化技术最初源于免安装运行 apk 的想法，这个免安装的 apk 就可以理解为插件，而支持插件的 app 我们一般叫宿主。宿主可以在运行时加载和运行插件，这样便可以将 app 中一些不常用的功能模块做成插件，一方面减小了安装包的大小，另一方面可以实现 app 功能的动态扩展。

插件化的开源框架

插件化发展到现在，已经出现了非常多的框架，下表列出部分框架：

特性	DynamicAPK	dynamic-load-apk	Small	DroidPlugin	RePlugin	VirtualAPK
作者	携程	任玉刚	wequick	360	360	滴滴
支持四大组件	只支持Activity	只支持Activity	只支持Activity	全支持	全支持	全支持
组件无需在宿主manifest中预注册	×	√	√	√	√	√
插件可以依赖宿主	√	√	√	×	√	√
支持PendingIntent	×	×	×	√	√	√
Android特性支持	大部分	大部分	大部分	几乎全部	几乎全部	几乎全部
兼容性适配	一般	一般	中等	高	高	高
插件构建	部署aapt	无	Gradle插件	无	Gradle插件	Gradle插件

我们在选择开源框架的时候，需要根据自身的需求来，如果加载的插件不需要和宿主有任何耦合，也无须和宿主进行通信，比如加载第三方 App，那么推荐使用 RePlugin，其他的情况推荐使用 VirtualApk。

插件化的实现

我们如何去实现一个插件化呢?

首先我们要知道，插件apk是没有安装的，那我们怎么加载它呢？不知道。。。

没关系，这儿我们还可以细分下，一个 apk 主要就是由代码和资源组成，所以上面的问题我们可以变为：如何加载插件的类？如何加载插件的资源？这样的话是不是就有眉目了。

然后我们还需要解决类的调用的问题，这个地方主要是四大组件的调用问题。我们都知道，四大组件是需要注册的，而插件的四大组件显然没有注册，那我们怎么去调用呢?

所以我们接下来就是解决这三个问题，从而实现插件化

1. 如何加载插件的类?
2. 如何加载插件的资源?

3. 如何调用插件类?

类加载（ClassLoader）

我们在学 java 的时候知道，java 源码文件编译后会生成一个 class 文件，而在 Android 中，将代码编译后会生成一个 apk 文件，将 apk 文件解压后就可以看到其中有一个或多个 classes.dex 文件，它就是安卓把所有 class 文件进行合并，优化后生成的。

java 中 JVM 加载的是 class 文件，而安卓中 DVM 和 ART 加载的是 dex 文件，虽然二者都是用的 ClassLoader 加载的，但因为加载的文件类型不同，还是有些区别的，所以接下来我们主要介绍安卓的 ClassLoader 是如何加载 dex 文件的。

ClassLoader的实现类

ClassLoader是一个抽象类，实现类主要分为两种类型：系统类加载器和自定义加载器。

其中系统类加载器主要包括三种：

- BootClassLoader

  用于加载Android Framework层class文件。

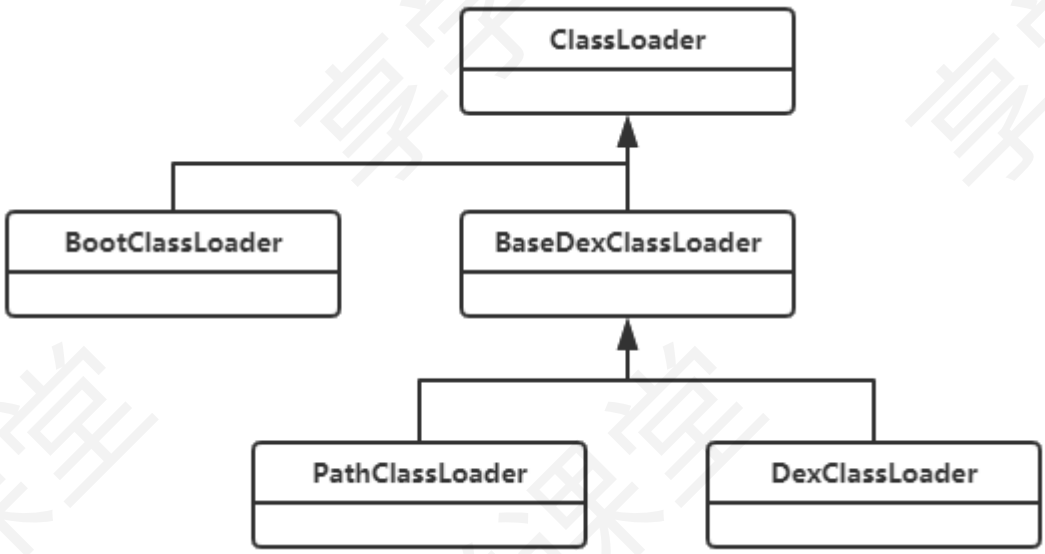
- PathClassLoader

  用于Android应用程序类加载器。可以加载指定的dex，以及jar、zip、apk中的classes.dex

- DexClassLoader

  用于加载指定的dex，以及jar、zip、apk中的classes.dex

类继承关系如下图：



我们先来看下 PathClassLoader 和 DexClassLoader。

```
// /libcore/dalvik/src/main/java/dalvik/system/PathClassLoader.java
public class PathClassLoader extends BaseDexClassLoader {
    // optimizedDirectory 直接为 null
    public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }
```

```
    // optimizedDirectory 直接为 null
    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent) {
        super(dexPath, null, librarySearchPath, parent);
    }
}

// API 小于等于 26/libcore/dalvik/src/main/java/dalvik/system/DexClassLoader.java
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
                          String librarySearchPath, ClassLoader parent) {
        // 26开始，super里面改变了，看下面两个构造方法
        super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
    }
}

// API 26/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java
 public BaseDexClassLoader(String dexPath, File optimizedDirectory,
                           String librarySearchPath, ClassLoader parent) {
    super(parent);
    // DexPathList 的第四个参数是 optimizedDirectory，可以看到这儿为 null
    this.pathList = new DexPathList(this, dexPath, librarySearchPath, null);
}

// API 25/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java
public BaseDexClassLoader(String dexPath, File optimizedDirectory,
                          String librarySearchPath, ClassLoader parent) {
    super(parent);
    this.pathList = new DexPathList(this, dexPath, librarySearchPath, optimizedDirectory);
}
```

根据源码了解到，PathClassLoader 和 DexClassLoader 都是继承自 BaseDexClassLoader，且类中只有构造方法，它们的类加载逻辑完全写在 BaseDexClassLoader 中。

其中我们值的注意的是，在8.0之前，它们二者的唯一区别是第二个参数 optimizedDirectory，这个参数的意思是生成的 odex（优化的dex）存放的路径，PathClassLoader 直接为null，而 DexClassLoader 是使用用户传进来的路径，而在8.0之后，二者就完全一样了。

下面我们再来了解下 BootClassLoader 和 PathClassLoader 之间的关系。

```
// 在 onCreate 中执行下面代码
ClassLoader classLoader = getClassLoader();
while (classLoader != null) {
    Log.e("leo", "classLoader:" + classLoader);
    classLoader = classLoader.getParent();
}
Log.e("leo", "classLoader:" + Activity.class.getClassLoader());
```

打印结果：

```
classLoader:dalvik.system.PathClassLoader[DexPathList[[zip file
"/data/user/0/com.enjoy.pluginactivity/cache/plugin-debug.apk", zip file
"/data/app/com.enjoy.pluginactivity-T4YwTh-
8gHWWDDS19IkHRg==/base.apk"],nativeLibraryDirectories=[/data/app/com.enjoy.pluginactivity-
T4YwTh-8gHWWDDS19IkHRg==/lib/x86_64, /system/lib64, /vendor/lib64]]]
classLoader:java.lang.BootClassLoader@a26e88d
classLoader:java.lang.BootClassLoader@a26e88d
```

通过打印结果可知，应用程序类是由 PathClassLoader 加载的，Activity 类是 BootClassLoader 加载的，并且 BootClassLoader 是 PathClassLoader 的 parent，这里要注意 parent 与父类的区别。这个打印结果我们下面还会提到。

加载原理

那我们如何使用类加载器去加载一个类呢?

非常的简单，例如：我们有一个apk文件，路径是 apkPath，然后里面有个类 com.enjoy.plugin.Test，那么我们可以通过如下方式去加载 Test 类：

```
DexClassLoader dexClassLoader = new
DexClassLoader(dexPath,context.getCacheDir().getAbsolutePath(),     null,
context.getClassLoader());
Class<?> clazz = dexClassLoader.loadClass("com.enjoy.plugin.Test");
```

因为我们需要将插件的 dex 文件加载到宿主里面，所以我们接下来分析源码，看 DexClassLoader 类加载器到底是怎么加载一个 apk 的 dex 文件的。

通过查找发现，DexClassLoader 类中没有 loadClass 方法，一路向上查找，最后在 ClassLoader 类中找到了改方法，源码如下：（后续源码如无标明，都是 API 26 Android 8.0）

```
// /libcore/ojluni/src/main/java/java/lang/ClassLoader.java
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException{
    // 检测这个类是否已经被加载  --> 1
    Class<?> c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                // 如果parent不为null，则调用parent的loadClass进行加载
                c = parent.loadClass(name, false);
            } else {
                // 正常情况下不会走这儿，因为 BootClassLoader 重写了 loadClass 方法，结束了递归
                c = findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException e) {
        }

        if (c == null) {
            // 如果仍然找不到，就调用 findClass 去查找 --> 2
            c = findClass(name);
        }
```

```
    }
    return c;
}

// -->1 检测这个类是否已经被加载
protected final Class<?> findLoadedClass(String name) {
    ClassLoader loader;
    if (this == BootClassLoader.getInstance())
        loader = null;
    else
        loader = this;
    // 最后通过 native 方法实现查找
    return VMClassLoader.findLoadedClass(loader, name);
}

// -->2 加载器一般都会重写这个方法，定义自己的加载规则
protected Class<?> findClass(String name) throws ClassNotFoundException {
    throw new ClassNotFoundException(name);
}
```

首先检测这个类是否已经被加载了，如果已经加载了，直接获取并返回。如果没有被加载，parent 不为 null，则调用parent的loadClass进行加载，依次递归，如果找到了或者加载了就返回，如果即没找到也加载不了，才自己去加载。这个过程就是我们常说的 **双亲委托机制。**

根据前面的打印结果可以知道，BootClassLoader 是最后一个加载器，所以我们来看下它是如何结束向上递归查找的。

```
class BootClassLoader extends ClassLoader {
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        return Class.classForName(name, false, null);
    }

    @Override
    protected Class<?> loadClass(String className, boolean resolve)
        throws ClassNotFoundException {
        Class<?> clazz = findLoadedClass(className);

        if (clazz == null) {
            clazz = findClass(className);
        }

        return clazz;
    }
}
```

我们发现 BootClassLoader 重写了 findClass 和 loadClass 方法，并且在 loadClass 方法中，不再获取 parent，从而结束了递归。

接着我们再来看下，在所有 parent 都没加载成功的情况下，DexClassLoader 是如何加载的。通过查找我们发现在它的父类 BaseDexClassLoader 中，重写了 findClass 方法。

```
// /libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    // 在 pathList 中查找指定的 Class
    Class c = pathList.findClass(name, suppressedExceptions);
    return c;
}

public BaseDexClassLoader(String dexPath, File optimizedDirectory,
                          String librarySearchPath, ClassLoader parent) {
    super(parent);
    // 初始化 pathList
    this.pathList = new DexPathList(this, dexPath, librarySearchPath, null);
}
```

接着再来看 DexPathList 类中的 findClass 方法。

```
private Element[] dexElements;

public Class<?> findClass(String name, List<Throwable> suppressed) {
    //通过 Element 获取 Class 对象
    for (Element element : dexElements) {
        Class<?> clazz = element.findClass(name, definingContext, suppressed);
        if (clazz != null) {
            return clazz;
        }
    }
    return null;
}
```

我们发现 Class 对象就是从 Element 中获得的，而每一个 Element 就对应一个 dex 文件，因为我们的 dex 文件可能有多个，所以这儿使用数组 Element[]。到这儿我们的思路就出来了，分为以下几步：

1. 创建插件的 DexClassLoader 类加载器，然后通过反射获取插件的 dexElements 值。
2. 获取宿主的 PathClassLoader 类加载器，然后通过反射获取宿主的 dexElements 值。
3. 合并宿主的 dexElements 与 插件的 dexElements，生成新的 Element[]。
4. 最后通过反射将新的 Element[] 赋值给宿主的 dexElements 。

具体代码如下：

```
public static void loadClass(Context context) {
    try {
        // 1.获取 pathList 的字段
        Class baseDexClassLoader = Class.forName("dalvik.system.BaseDexClassLoader");
        Field pathListField = baseDexClassLoader.getDeclaredField("pathList");
        pathListField.setAccessible(true);

        /**
         * 获取插件的 dexElements[]
         */
        // 2.获取 DexClassLoader 类中的属性 pathList 的值
```

```
        DexClassLoader dexClassLoader = new DexClassLoader(apkPath,
                context.getCacheDir().getAbsolutePath(), null, context.getClassLoader());
        Object pluginPathList = pathListField.get(dexClassLoader);

        // 3.获取 pathList 中的属性 dexElements[] 的值--- 插件的 dexElements[]
        Class pluginPathListClass = pluginPathList.getClass();
        Field pluginDexElementsField = pluginPathListClass.getDeclaredField("dexElements");
        pluginDexElementsField.setAccessible(true);
        Object[] pluginDexElements = (Object[]) pluginDexElementsField.get(pluginPathList);

        /**
         * 获取宿主的 dexElements[]
         */
        // 4.获取 PathClassLoader 类中的属性 pathList 的值
        PathClassLoader pathClassLoader = (PathClassLoader) context.getClassLoader();
        Object hostPathList = pathListField.get(pathClassLoader);

        // 5.获取 pathList 中的属性 dexElements[] 的值--- 宿主的 dexElements[]
        Class hostPathListClass = hostPathList.getClass();
        Field hostDexElementsField = hostPathListClass.getDeclaredField("dexElements");
        hostDexElementsField.setAccessible(true);
        Object[] hostDexElements = (Object[]) hostDexElementsField.get(hostPathList);

        /**
         * 将插件的 dexElements[] 和宿主的 dexElements[] 合并为一个新的 dexElements[]
         */
        // 6.创建一个新的空数组，第一个参数是数组的类型，第二个参数是数组的长度
        Object[] dexElements = (Object[]) Array.newInstance(
                hostDexElements.getClass().getComponentType(),
                pluginDexElements.length + hostDexElements.length);

        // 7.将插件和宿主的 dexElements[] 的值放入新的数组中
        System.arraycopy(pluginDexElements, 0, dexElements, 0, pluginDexElements.length);
        System.arraycopy(hostDexElements, 0, dexElements, pluginDexElements.length,
hostDexElements.length);
        /**
         * 将生成的新值赋给 "dexElements" 属性
         */
        hostDexElementsField.set(hostPathList, dexElements);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

资源加载

在项目中，我们一般通过 Resources 去访问 res 中的资源，使用 AssetManager 访问 assets 里面的资源。如下：

```
String appName = getResources().getString(R.string.app_name);
InputStream is = getAssets().open("icon_1.png");
```

实际上，Resources 类也是通过 AssetManager 类来访问那些被编译过的应用程序资源文件的，不过在访问之前，它会先根据资源 ID 查找得到对应的资源文件名。 而 AssetManager 对象既可以通过文件名访问那些被编译过的，也可以访问没有被编译过的应用程序资源文件。

我们来看下 Resources 调用 getString 的代码实现过程：

```
// android/content/res/Resources.java
public String getString(@StringRes int id) throws NotFoundException {
    return getText(id).toString();
}
public CharSequence getText(@StringRes int id) throws NotFoundException {
    CharSequence res = mResourcesImpl.getAssets().getResourceText(id);
    if (res != null) {
        return res;
    }
    throw new NotFoundException("String resource ID #0x"
                                + Integer.toHexString(id));
}
```

通过上面的代码知道 Resources 的实现类是 ResourceImpl 类，getAssets() 返回的是 AssetManager，所以也就证实了资源的加载实际是通过 AssetManager 来加载的。

接下来我们看下 AssetManager 是如何创建和初始化的，又是如何加载 apk 资源的，只有掌握了原理，我们才知道如何去加载另一个 apk 的资源。

```
// android/app/LoadedApk
public Resources getResources() {
    if (mResources == null) {
        // 获取 ResourcesManager 对象的单例，然后调用 getResources 方法去获取 Resources 对象 --> 1
        mResources = ResourcesManager.getInstance().getResources(null, mResDir,
            splitPaths, mOverlayDirs, mApplicationInfo.sharedLibraryFiles,
            Display.DEFAULT_DISPLAY, null, getCompatibilityInfo(),getClassLoader());
    }
    return mResources;
}
```

```
// android/app/ResourcesManager
// --> 1
public @Nullable Resources getResources(@Nullable IBinder activityToken,
            @Nullable String resDir,
            @Nullable String[] splitResDirs,
            @Nullable String[] overlayDirs,
            @Nullable String[] libDirs,
            int displayId,
            @Nullable Configuration overrideConfig,
            @NonNull CompatibilityInfo compatInfo,
            @Nullable ClassLoader classLoader) {
    try {
        final ResourcesKey key = new ResourcesKey(
            resDir, // 这个就是 apk 文件路径
            splitResDirs,
```

```
            overlayDirs,
            libDirs,
            displayId,
            overrideConfig != null ? new Configuration(overrideConfig) : null, // Copy
            compatInfo);
        classLoader = classLoader != null ? classLoader : ClassLoader.getSystemClassLoader();
        // 获取或者创建 Resources 对象 --> 2
        return getOrCreateResources(activityToken, key, classLoader);
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_RESOURCES);
    }
}

// --> 2
private @Nullable Resources getOrCreateResources(@Nullable IBinder activityToken,
            @NonNull ResourcesKey key, @NonNull ClassLoader classLoader) {
    // 创建 ResourcesImpl 对象 --> 3
    ResourcesImpl resourcesImpl = createResourcesImpl(key);

    // resources 是 ResourcesImpl 的装饰类
    return resources;
}

// --> 3
private @Nullable ResourcesImpl createResourcesImpl(@NonNull ResourcesKey key) {
    // 创建 AssetManager 对象 --> 4
    final AssetManager assets = createAssetManager(key);
    if (assets == null) {
        return null;
    }
    // 将 assets 对象传入到 ResourcesImpl 类中
    final ResourcesImpl impl = new ResourcesImpl(assets, dm, config, daj);
    return impl;
}

// --> 4
protected @Nullable AssetManager createAssetManager(@NonNull final ResourcesKey key) {
    AssetManager assets = new AssetManager();
    if (key.mResDir != null) {
        // 通过 addAssetPath 方法添加 apk 文件的路径
        if (assets.addAssetPath(key.mResDir) == 0) {
            Log.e(TAG, "failed to add asset path " + key.mResDir);
            return null;
        }
    }
    return assets;
}
```

通过上面代码的分析，我们知道了 apk 文件的路径是通过 assets.addAssetPath 方法设置的，所以如果我们想将插件的 apk 文件添加到宿主中，就可以通过反射修改这个地方。

实现步骤：

1. 创建一个 AssetManager 对象，并调用 addAssetPath 方法，将插件 apk 的路径作为参数传入。
2. 将第一步创建的 AssetManager 对象作为参数，创建一个新的 Resources 对象，并返回给插件使用。

具体代码如下：

```
public static Resources loadResource(Context context) {
    try {
        AssetManager assetManager = AssetManager.class.newInstance();
        Method addAssetPathMethod = assetManager.getClass()
            .getDeclaredMethod("addAssetPath", String.class);
        addAssetPathMethod.setAccessible(true);
        addAssetPathMethod.invoke(assetManager, apkPath);
        Resources resources = context.getResources();
        // 用来加载插件包中的资源
        return new Resources(assetManager, resources.getDisplayMetrics(),
                             resources.getConfiguration());
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

然后在宿主的自定义 Application 类中添加如下代码：

```
// 宿主代码
private Resources resources;

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

    // 获取新建的 resources 资源
    resources = LoadUtil.loadResource(this);
}

// 重写该方法，当 resources 为空时，相当于没有重写，不为空时，返回新建的 resources 对象
@Override
public Resources getResources() {
    return resources == null ? super.getResources() : resources;
}
```

接着在插件中，创建BaseActivity，如下：

```
// 插件中代码
public abstract class BaseActivity extends AppCompatActivity {

    @Override
    public Resources getResources() {
        if (getApplication() != null && getApplication().getResources() != null) {
            // 因为宿主重写了该方法，所以获取的将是新创建的 resources 对象
            return getApplication().getResources();
        }
        return super.getResources();
    }
}
```

然后让插件的 Activity 都继承自 BaseActivity，这样，插件在获取资源时，使用的就是在宿主中新创建的 resources 对象，也就可以拿到资源了。

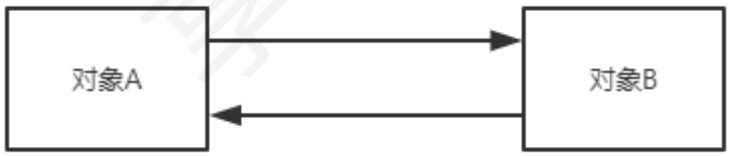
宿主启动插件的Activity

Activity 是需要在清单文件中注册的，显然，插件的 Activity 没有在宿主的清单文件中注册，那我们如何来启动它呢?

这里我们就需要使用 Hook 技术，来绕开系统的检测。可能有些同学不知道 Hook 是什么，所以我们先简单的介绍下 Hook 技术。

Hook

正常情况下对象A调用对象B，对象B处理后将数据返回给对象A，如下图：



而加入Hook后的流程就变成了下图形式：

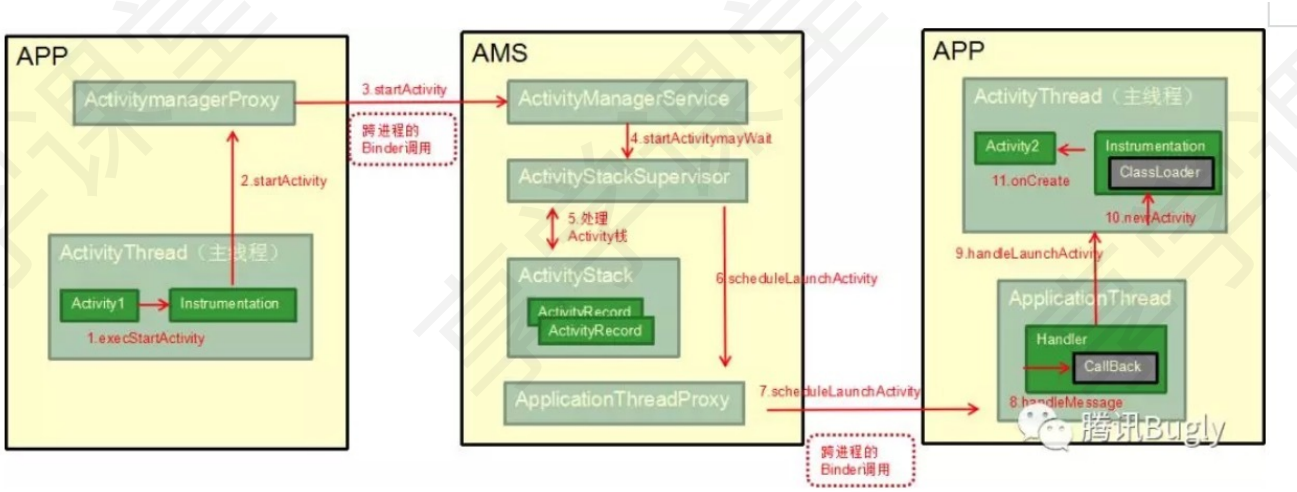
Hook可以是一个方法或者一个对象，它就像一个钩子一样挂在对象B上面，当对象A调用对象B之前，Hook可以做一些处理，起到“欺上瞒下”的作用。而对象B就是我们常说的Hook点。为了保证Hook的稳定性，Hook点一般选择容易找到并且不易变化的对象，如静态变量和单例。

那么思路就来了，首先我们在宿主里面创建一个 ProxyActivity 继承自 Activity，并且在清单中注册。当启动插件 Activity 的时候，在系统检测前，找到一个Hook点，然后通过 Hook 将插件 Activity 替换成 ProxyActivity，等到检测完了后，再找一个Hook点，使用 Hook 将它们换回来，这样就实现了插件 Activity 的启动。思路是不是非常的简单。

如何查找 Hook 点呢？这就需要我们了解 Activity 的启动流程了。

Activity的启动流程

首先我们来看一张 Activity 启动流程的简单示意图，如下：



通过这张图我们可以确定 Hook 点的大致位置。

1. 在进入 AMS 之前，找到一个 Hook 点，用来将插件 Activity 替换为 ProxyActivity。
2. 从 AMS 出来后，再找一个 Hook 点，用来将 ProxyActivity 替换为插件 Activity。

在看源码之前，我们再想一个问题，看源码是要找什么东西作为 Hook 点呢?

我们在项目中一般通过 `startActivity(new Intent(this,PluginActivity.class));` 启动 PluginActivity，如果我想换成启动 ProxyActivity，调用方法 `startActivity(new Intent(this,ProxyActivity.class));` 这样就可以了。是不是已经知道答案了！！！没错，我们只要找到能够修改 Intent 的地方，就可以作为 Hook 点，从这儿也可以看出 Hook 点并不是唯一的。

好的，下面我们进入源码

```
// android/app/Activity.java
@Override
public void startActivity(Intent intent) {
    this.startActivity(intent, null);
}

@Override
public void startActivity(Intent intent, @Nullable Bundle options) {
    startActivityForResult(intent, -1, options);
}

public void startActivityForResult(@RequiresPermission Intent intent, int requestCode,
            @Nullable Bundle options) {
    Instrumentation.ActivityResult ar = mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode, options);
}
```

```
// android/app/Instrumentation.java
public ActivityResult execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity target,
            Intent intent, int requestCode, Bundle options) {
    // 这儿就是我们的 Hook 点，替换传入 startActivity 方法中的 intent 参数
    int result = ActivityManager.getService()
                .startActivity(whoThread, who.getBasePackageName(), intent,
                        intent.resolveTypeIfNeeded(who.getContentResolver()),
                        token, target != null ? target.mEmbeddedID : null,
                        requestCode, 0, null, options);
}
```

既然 Hook 点找到了，那我们怎么修改呢?

答案是动态代理，所以我们要生成一个代理对象，显然，我们要代理的是 `ActivityManager.getService()` 返回的对象。

那下面我们就来看下它返回的是什么?

首先我们看下 `ActivityManager.getService()` 返回的是一个什么类的对象

```
// android/app/ActivityManager.java
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}
```

可以看到，它返回的是 IActivityManager 类的对象。下面我们就生成代理对象，并且当执行的方法是 startActivity 的时候，替换它的参数 intent。代码如下:

```
Object mInstanceProxy = Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
        new Class[]{iActivityManagerClass}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                // 当执行的方法是 startActivity 时作处理
                if ("startActivity".equals(method.getName())) {
                    int index = 0;
                    // 获取 Intent 参数在 args 数组中的index值
                    for (int i = 0; i < args.length; i++) {
                        if (args[i] instanceof Intent) {
                            index = i;
                            break;
                        }
                    }
                    // 得到原始的 Intent 对象 -- 唐僧（插件）的Intent
                    Intent intent = (Intent) args[index];

                    // 生成代理proxyIntent -- 孙悟空（代理）的Intent
                    Intent proxyIntent = new Intent();
                    proxyIntent.setClassName("com.enjoy.pluginactivity",
                            ProxyActivity.class.getName());
                    // 保存原始的Intent对象
```

```
                    proxyIntent.putExtra(TARGET_INTENT, intent);
                    // 使用proxyIntent替换数组中的Intent
                    args[index] = proxyIntent;
                }
                return method.invoke(mInstance, args);
            }
        });
```

接着我们再使用反射将系统中的 IActivityManager 对象替换为我们的代理对象 mInstanceProxy。那如何替换了？我们接着看源码。

```
// android/app/ActivityManager.java
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}

private static final Singleton<IActivityManager> IActivityManagerSingleton =
     new Singleton<IActivityManager>() {
     @Override
     protected IActivityManager create() {
         final IBinder b = ServiceManager.getService(Context.ACTIVITY_SERVICE);
         final IActivityManager am = IActivityManager.Stub.asInterface(b);
         return am;
     }
 };
```

通过上面的代码，我们知道 IActivityManager 是调用的 Singleton 里面的 get 方法，所以下面我们再看下 Singleton 是怎么样的。

```
// android/util/Singleton
public abstract class Singleton<T> {
    private T mInstance;

    protected abstract T create();

    public final T get() {
        synchronized (this) {
            if (mInstance == null) {
                mInstance = create();
            }
            return mInstance;
        }
    }
}
```

可以看出，IActivityManagerSingleton.get() 返回的实际上就是 mInstance 对象。所以接下来我们要替换的就是这个对象。代码如下：

```
// 获取 Singleton<T> 类的对象
Class<?> clazz = Class.forName("android.app.ActivityManager");
Field singletonField = clazz.getDeclaredField("IActivityManagerSingleton");
singletonField.setAccessible(true);
Object singleton = singletonField.get(null);
// 获取 mInstance 对象
Class<?> singletonClass = Class.forName("android.util.Singleton");
Field mInstanceField = singletonClass.getDeclaredField("mInstance");
mInstanceField.setAccessible(true);
final Object mInstance = mInstanceField.get(singleton);
// 使用代理对象替换原有的 mInstance 对象
mInstanceField.set(singleton, mInstanceProxy);
```

到这儿我们的第一步就实现了，接着我们来实现第二步，在出来的时候，将它们换回去。

还记得前面那张图吗？在出来的时候，会调用 Handler 的 handleMessage，所以下面我们看下 Handler 的源码。

```
public void handleMessage(Message msg) {
}

public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

当 mCallback != null 时，首先会执行 mCallback.handleMessage(msg)，再执行 handleMessage(msg)，所以我们可以将 mCallback 作为 Hook 点，创建它。ok，现在问题就只剩一个了，就是找到含有 intent 的对象，没办法，只能接着看源码。

```
// android/app/ActivityThread.java
public void handleMessage(Message msg) {
    switch (msg.what) {
        case LAUNCH_ACTIVITY: {
            final ActivityClientRecord r = (ActivityClientRecord) msg.obj;
            handleLaunchActivity(r, null, "LAUNCH_ACTIVITY");
        } break;
    }
}

static final class ActivityClientRecord {
    Intent intent;
}
```

可以看到，在 ActivityClientRecord 类中，刚好就有个 intent，而且这个类的对象，我们也可以获取到，就是 msg.obj。接下来就简单了，实现代码如下，如果有兴趣的同学也可从 handleLaunchActivity 方法一路跟下去，看看 ActivityClientRecord 的 intent 到底在哪使用的，这儿我们就不赘叙了。

```
// 获取 ActivityThread 类的 对象
Class<?> clazz = Class.forName("android.app.ActivityThread");
Field activityThreadField = clazz.getDeclaredField("sCurrentActivityThread");
activityThreadField.setAccessible(true);
Object activityThread = activityThreadField.get(null);

// 获取 Handler 对象
Field mHField = clazz.getDeclaredField("mH");
mHField.setAccessible(true);
final Handler mH = (Handler) mHField.get(activityThread);

// 设置 Callback 的值
Field mCallbackField = Handler.class.getDeclaredField("mCallback");
mCallbackField.setAccessible(true);
mCallbackField.set(mH, new Handler.Callback() {
    @Override
    public boolean handleMessage(Message msg) {
        switch (msg.what) {
            case 100:
                try {
                    // 获取 proxyIntent
                    Field intentField = msg.obj.getClass().getDeclaredField("intent");
                    intentField.setAccessible(true);
                    Intent proxyIntent = (Intent) intentField.get(msg.obj);

                    // 目标 intent 替换 proxyIntent
                    Intent intent = proxyIntent.getParcelableExtra(TARGET_INTENT);
                    proxyIntent.setComponent(intent.getComponent());
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
        }
        return false;
    }
});
```

总结

插件化涉及的技术其实是非常多的，比如应用程序启动流程、四大组件启动流程、AMS原理、ClassLoader原理、Binder机制，动态代理等等。最后送给大家一句话，路漫漫其修远兮，吾将上下而求索。