

插件化方案研究和实践

VirtualApk分析

- [插件化方案研究和实践](#)
- [VirtualApk分析](#)
 - [PluginManager初始化](#)
 - [坑位管理者ComponentsHandler](#)
 - [初始化prepare\(\)](#)
 - [hook系统行为加载没注册Activity](#)
 - [自定义Instrumentation行为](#)
 - [插件Apk的加载和解析](#)
 - [解析apk相关信息](#)
 - [生成插件Resource](#)
 - [创建插件Classloader](#)
 - [创建插件使用的Context](#)
 - [其他](#)
 - [关于Theme和AppCompatActivity](#)
- [Replugin分析总结](#)
 - [Replugin的框架的初始化](#)
 - [IPC.init](#)
 - [PMF](#)
 - [PMF.init](#)
 - [PMF.callAttach](#)
 - [PmBase](#)
 - [PmBase的创建](#)
 - [PmBase.init](#)
 - [在UI进程分支中](#)
 - [在常驻进程分支中](#)
 - [Replugin的hook原理分析](#)
 - [插件安装和加载](#)
 - [首先是在宿主工程中hook系统Classloader](#)
 - [加载插件基本信息](#)
 - [解析插件和构造插件资源](#)
 - [1.解析apk信息](#)
 - [2.创建插件资源](#)

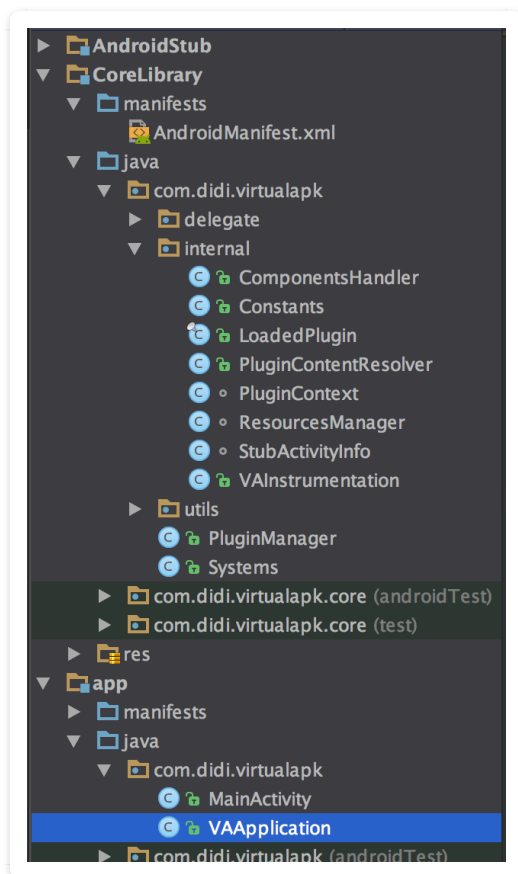
- [3.创建ClassLoader](#)
 - [4.创建插件的Context](#)
- [启动坑位Activity但创建插件Activity](#)
- [一起其他功能支持](#)
 - [加载插件中Fragment](#)
 - [插件和宿主通信](#)

VirtualApk的实现和代码逻辑比较清晰，由于插件Activity是没有在宿主Manifest中注册，所以先使用提前注册的坑位Activity替换插件中Activity骗过AMS的检查，然后hook系统行为启动插件中Activity，最后在插件中使用插件自己的Resouce，Context。

我们可以跟着流程走一遍可以理解的它的实现。

主要关注以下三个方面：

- 如何管理和使用坑位Activity？
- 如何hook系统启动插件Activity？
- 如何使用插件中相关资源和信息？



PluginManager初始化

首先在Application的创建时调用PluginManager的init () 方法。

- 这里PluginManager.getInstance(base)应该改为PluginManager.getInstance () ，因为只有第一次时会将宿主的context传给PluginManager，同时这个context最好是

ApplicationContext。

- init方法不必在Applicaiont.attachBaseContext中调用，只要在加载插件前调用即可。

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    long start = System.currentTimeMillis();

    PluginManager.getInstance(base).init();
    Log.d("ryg", "use time:" + (System.currentTimeMillis() - start));
}

public static PluginManager getInstance(Context base) {
    if (sInstance == null) {
        synchronized (PluginManager.class) {
            if (sInstance == null)
                sInstance = new PluginManager(base);
        }
    }
    return sInstance;
}

private PluginManager(Context context) {
    Context app = context.getApplicationContext();
    if (app == null) {
        this.mContext = context;
    } else {
        this.mContext = ((Application)app).getBaseContext();
    }
    prepare();
}
```

```
public void init() {
    mComponentsHandler = new ComponentsHandler(this);
}
```

prepare方法我们一会再看，我们先看看init方法。

init方法也很简单，创建了ComponentsHandler对象，这个ComponentsHandler其实就是Activity，service等坑位管理者（后面讲解只针对Activity）。

坑位管理者ComponentsHandler

在CoreLibrary的AndroidManifest中声明了一系列坑位Activity。

```
<application>
  <!-- Stub Activities -->
  <activity android:name=".A$1" android:launchMode="standard"/>
  <activity android:name=".A$2" android:launchMode="standard"
    android:theme="@android:style/Theme.Translucent" />

  <!-- Stub Activities -->
  <activity android:name=".B$1" android:launchMode="singleTop"/>
  <activity android:name=".B$2" android:launchMode="singleTop"/>
  <activity android:name=".B$3" android:launchMode="singleTop"/>
  <activity android:name=".B$4" android:launchMode="singleTop"/>
  <activity android:name=".B$5" android:launchMode="singleTop"/>
  <activity android:name=".B$6" android:launchMode="singleTop"/>
  <activity android:name=".B$7" android:launchMode="singleTop"/>
  <activity android:name=".B$8" android:launchMode="singleTop"/>

  <!-- Stub Activities -->
  <activity android:name=".C$1" android:launchMode="singleTask"/>
  <activity android:name=".C$2" android:launchMode="singleTask"/>
  <activity android:name=".C$3" android:launchMode="singleTask"/>
  <activity android:name=".C$4" android:launchMode="singleTask"/>
  <activity android:name=".C$5" android:launchMode="singleTask"/>
  <activity android:name=".C$6" android:launchMode="singleTask"/>
  <activity android:name=".C$7" android:launchMode="singleTask"/>
  <activity android:name=".C$8" android:launchMode="singleTask"/>

  <!-- Stub Activities -->
  <activity android:name=".D$1" android:launchMode="singleInstance"/>
  <activity android:name=".D$2" android:launchMode="singleInstance"/>
  <activity android:name=".D$3" android:launchMode="singleInstance"/>
  <activity android:name=".D$4" android:launchMode="singleInstance"/>
  <activity android:name=".D$5" android:launchMode="singleInstance"/>
  <activity android:name=".D$6" android:launchMode="singleInstance"/>
```

```
public class ComponentsHandler {

    public static final String TAG = "PluginManager";

    private Context mContext;
    private PluginManager mPluginManager;
    private StubActivityInfo mStubActivityInfo = new StubActivityInfo(
);
```

其中StubActivityInfo对应着上面的坑位Activity。主要是根据启动模式和主题的不同，将要启动的目标Activity映射到坑位Activity，并将这种对应关系保存下来。只有一个核心方法，逻辑也很清楚。

```

public String getStubActivity(String className, int launchMode, Theme
theme) {
    String stubActivity= mCachedStubActivity.get(className);
    if (stubActivity != null) {
        return stubActivity;
    }

    TypedArray array = theme.obtainStyledAttributes(new int[]{
        android.R.attr.windowIsTranslucent,
        android.R.attr.windowBackground
    });
    boolean windowIsTranslucent = array.getBoolean(0, false);
    array.recycle();
    if (Constants.DEBUG) {
        Log.d("StubActivityInfo", "getStubActivity, is transparent
theme ? " + windowIsTranslucent);
    }
    stubActivity = String.format(STUB_ACTIVITY_STANDARD, corePacka
ge, usedStandardStubActivity);
    switch (launchMode) {
        case ActivityInfo.LAUNCH_MULTIPLE: {
            stubActivity = String.format(STUB_ACTIVITY_STANDARD, c
orePackage, usedStandardStubActivity);
            if (windowIsTranslucent) {
                stubActivity = String.format(STUB_ACTIVITY_STANDAR
D, corePackage, 2);
            }
            break;
        }
        case ActivityInfo.LAUNCH_SINGLE_TOP: {
            usedSingleTopStubActivity = usedSingleTopStubActivity
% MAX_COUNT_SINGLETOP + 1;
            stubActivity = String.format(STUB_ACTIVITY_SINGLETOP,
corePackage, usedSingleTopStubActivity);
            break;
        }
        case ActivityInfo.LAUNCH_SINGLE_TASK: {
            usedSingleTaskStubActivity = usedSingleTaskStubActivit
y % MAX_COUNT_SINGLETASK + 1;
            stubActivity = String.format(STUB_ACTIVITY_SINGLETASK,
corePackage, usedSingleTaskStubActivity);
            break;
        }
        case ActivityInfo.LAUNCH_SINGLE_INSTANCE: {
            usedSingleInstanceStubActivity = usedSingleInstanceStu
bActivity % MAX_COUNT_SINGLEINSTANCE + 1;
            stubActivity = String.format(STUB_ACTIVITY_SINGLEINSTA

```

```

NCE, corePackage, usedSingleInstanceStubActivity);
        break;
    }

    default:break;
}

mCachedStubActivity.put(className, stubActivity);
return stubActivity;
}

```

当我们启动一个插件中的Activity时，会自动将其替换为坑位Activity以达到欺骗AMS的作用。

```

public void markIntentIfNeeded(Intent intent) {
    String targetPackageName = intent.getComponent().getPackageName();
    String targetClassName = intent.getComponent().getClassName()
    // search map and return specific launchmode stub activity
    if (!targetPackageName.equals(mContext.getPackageName()) && mPluginManager.getLoadedPlugin(targetPackageName) != null) {
        intent.putExtra(Constants.KEY_IS_PLUGIN, true);
        intent.putExtra(Constants.KEY_TARGET_PACKAGE, targetPackageName);
        intent.putExtra(Constants.KEY_TARGET_ACTIVITY, targetClassName);
        dispatchStubActivity(intent);
    }

    private void dispatchStubActivity(Intent intent) {
        ComponentName component = intent.getComponent();
        String targetClassName = intent.getComponent().getClassName();
        LoadedPlugin loadedPlugin = mPluginManager.getLoadedPlugin(intent);
        ActivityInfo info = loadedPlugin.getActivityInfo(component);
        if (info == null) {
            throw new RuntimeException("can not find " + component);
        }
        int launchMode = info.launchMode;
        Resources.Theme themeObj = loadedPlugin.getResources().newTheme();
        themeObj.applyStyle(info.theme, true);
        String stubActivity = mStubActivityInfo.getStubActivity(targetClassName, launchMode, themeObj);
        Log.i(TAG, String.format("dispatchStubActivity, [%s -> %s]", targetClassName, stubActivity));
        intent.setClassName(mContext, stubActivity);
    }
}

```

```
}
```

初始化prepare()

现在我们再看看prepare方法到底做了什么，这个方法主要是hook了系统的Instrumentation行为，我们知道Activity的创建和声明周期都依赖它，利用Hook修改其行为加上坑位Activity，我们可以打开没有在Manifest中声明的Activity，也就是插件中的Activity。

```
private void prepare() {  
    Systems.sHostContext = getHostContext();  
    this.hookInstrumentationAndHandler();  
}
```

hook系统行为加载没注册Activity

插件中的Activity都是没有宿主的Manifest中注册过的，直接启动会报错。跟踪startActivity的调用流程，会发现最终进入Instrumentation的execStartActivity方法，然后再通过ActivityManagerProxy与AMS进行交互，而Activity是否存在的校验是发生在AMS端，所以我们可以提前将Activity的ComponentName替换成占坑的名字就可以解决上面的问题。

使用上面的方案解决了AMS校验问题，但这样启动的就是占坑的Activity而不是插件中的Activity，所以我们还得在合适的时机修改成启动插件中的Activity。

AMS在处理完启动Activity后，会调用：app.thread.scheduleLaunchActivity，即调用ApplicationThread.scheduleLaunchActivity方法，在其内部会调用mH类的sendMessage方法，传递的标识为H.LAUNCH_ACTIVITY，进入调用到ActivityThread的handleLaunchActivity方法，进入调用mInstrumentation.newActivity()。

所以我们还应该在newActivity中替换启动插件中实际需要启动的Activity。不过此时使用的Resource，Context等还是使用的宿主，也需要在合适时机替换掉，可以在Instrumentation的callActivityOnCreate中替换。

hook Instrumentation方案大同小异，我们可以看看具体实现。

- 首先获取ActivityThread，ActivityThread中有个静态成员变量sCurrentActivityThread，我们可以直接通过反射获取。

```
@UiThread  
public static Object getActivityThread(Context base) {  
    if (sActivityThread == null) {
```

```

        try {
            Class<?> activityThreadClazz = Class.forName("android.
app.ActivityThread");
            Object activityThread = null;
            try {
                activityThread = ReflectUtil.getField(activityThre
adClazz, null, "sCurrentActivityThread");
            } catch (Exception e) {
                // ignored
            }
            if (activityThread == null) {
                activityThread = ((ThreadLocal<?>) ReflectUtil.get
Field(activityThreadClazz, null, "sThreadLocal")).get();
            }
            sActivityThread = activityThread;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    return sActivityThread;
}

```

- 在调用ActivityThread的getInstrumentation得到Instrumentation

```

    public static Instrumentation getInstrumentation(Context base) {
        if (getActivityThread(base) != null) {
            try {
                sInstrumentation = (Instrumentation) ReflectUtil.invok
e(
                    sActivityThread.getClass(), sActivityThread, "
getInstrumentation");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        return sInstrumentation;
    }
}

```

- 创建我们自己的Instrumentation并hook掉系统的Instrumentation。

```

final VAInstrumentation instrumentation = new VAInstrumentation(this,

```



```

baseInstrumentation);
Object activityThread = ReflectUtil.getActivityThread(this.mContext)
ReflectUtil.setInstrumentation(activityThread, instrumentation);

    public static void setInstrumentation(Object activityThread, Instr
umentation instrumentation) {
        try {
            ReflectUtil.setField(activityThread.getClass(), activityTh
read, "mInstrumentation", instrumentation);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

在上面的自定Instrumentation的execStartActivity, newActivity, callActivityOnCreate等方法中可以执行我们上面说的替换和还远以及替换资源和context等操作。

自定义Instrumentation行为

- 使用上面提到
mPluginManager.getComponentHandler().markIntentIfNeeded(intent)方法获取坑位Activity的Intent并启动坑位Activity。

```

    public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activit
y target,
        Intent intent, int requestCode, Bundle options) {
        mPluginManager.getComponentHandler().transformIntentToExplici
tAsNeeded(intent);
        // null component is an implicitly intent
        if (intent.getComponent() != null) {
            // resolve intent with Stub Activity if needed
            this.mPluginManager.getComponentHandler().markIntentIfNee
ded(intent);
        }
        ActivityResult result = realExecStartActivity(who, contextThre
ad, token, target,
            intent, requestCode, options);
        return result;
    }
}

```

- 启动插件中实际要打开的Activity

```

    public Activity newActivity(ClassLoader cl, String className, Intent intent) throws InstantiationException, IllegalAccessException, ClassNotFoundException {
        try {
            cl.loadClass(className);
        } catch (ClassNotFoundException e) {
            LoadedPlugin plugin = this.mPluginManager.getLoadedPlugin(intent);
            String targetClassName = PluginUtil.getTargetActivity(intent);
            if (targetClassName != null) {
                Activity activity = mBase.newActivity(plugin.getClassLoader(), targetClassName, intent);
                activity.setIntent(intent);
                try {
                    // for 4.1+
                    ReflectUtil.setField(ContextThemeWrapper.class, activity, "mResources", plugin.getResources());
                } catch (Exception ignored) {
                    // ignored.
                }
                return activity;
            }
        }
        return mBase.newActivity(cl, className, intent);
    }
}

```

- 将Resource, Context等替换成插件中相应对象。

```

@Override
public void callActivityOnCreate(Activity activity, Bundle icle) {
    final Intent intent = activity.getIntent();
    if (PluginUtil.isIntentFromPlugin(intent)) {
        Context base = activity.getBaseContext();
        try {
            LoadedPlugin plugin = this.mPluginManager.getLoadedPlugin(intent);
            ReflectUtil.setField(base.getClass(), base, "mResources", plugin.getResources());
            ReflectUtil.setField(ContextWrapper.class, activity, "mBase", plugin.getPluginContext());
            ReflectUtil.setField(Activity.class, activity, "mApplication", plugin.getApplication());
            ReflectUtil.setFieldNoException(ContextThemeWrapper.class, activity, "mBase", plugin.getPluginContext());
        }
    }
}

```

```

        // set screenOrientation
        ActivityInfo activityInfo = plugin.getActivityInfo(PluginUtil.getComponent(intent));
        if (activityInfo.screenOrientation != ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED) {
            activity.setRequestedOrientation(activityInfo.screenOrientation);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
mBase.callActivityOnCreate(activity, icycle);
}

```

插件Apk的加载和解析

加载apk的过程通常是通过插件apk文件解析出PackageInfo, ApplicationInfo, ComponentInfo (ActivityInfo, ServiceInfo等)。生成插件的Resource, 插件的ClassLoader, 插件的Context等。

解析apk相关信息

VirtualApk使用的方案是通过系统的PackageParser得到Package, 需要考虑版本兼容性和反射相关方法。然后再创建PackageInfo, 使用Package中相关信息初始化后使用。

```

this.mPackage = PackageParserCompat.parsePackage(context, apk, PackageParser.PARSE_MUST_BE_APK);
this.mPackage.applicationInfo.metaData = this.mPackage.mAppMetaData;
this.mPackageInfo = new PackageInfo();
this.mPackageInfo.applicationInfo = this.mPackage.applicationInfo;
this.mPackageInfo.applicationInfo.sourceDir = apk.getAbsolutePath();
// Cache activities
Map<ComponentName, ActivityInfo> activityInfos = new HashMap<ComponentName, ActivityInfo>();
for (PackageParser.Activity activity : this.mPackage.activities) {

    activityInfos.put(activity.getComponentName(), activity.info);
}
this.mActivityInfos = Collections.unmodifiableMap(activityInfos);
this.mPackageInfo.activities = activityInfos.values().toArray(new ActivityInfo[activityInfos.size()]);

```

...

```
    public static final PackageParser.Package parsePackage(final Context context, final File apk, final int flags) throws PackageParser.PackageParserException {
        if (Build.VERSION.SDK_INT >= 24) {
            return PackageParserV24.parsePackage(context, apk, flags);
        } else if (Build.VERSION.SDK_INT >= 21) {
            return PackageParserLollipop.parsePackage(context, apk, flags);
        } else {
            return PackageParserLegacy.parsePackage(context, apk, flags);
        }
    }

    private static final class PackageParserV24 {

        static final PackageParser.Package parsePackage(Context context, File apk, int flags) throws PackageParser.PackageParserException {
            PackageParser parser = new PackageParser();
            PackageParser.Package pkg = parser.parsePackage(apk, flags);

            ReflectUtil.invokeNoException(PackageParser.class, null, "collectCertificates",
                new Class[]{PackageParser.Package.class, int.class}, pkg, flags);
            return pkg;
        }
    }
}
```

实际上应该可以通过PackageManager直接得到的吧？Replugin也是这样做的，并且跟踪PackageManager解析apk的过程也是在内部通过PackageParse解析的。

```
PackageManager pm = mContext.getPackageManager();
mPackageInfo = pm.getPackageArchiveInfo(mPath, PackageManager.GET_ACTIVITIES | PackageManager.GET_META_DATA);
mPackageInfo.applicationInfo.sourceDir = mPath;
...

    public PackageInfo getPackageArchiveInfo(String archiveFilePath, @PackageInfoFlags int flags) {
        final PackageParser parser = new PackageParser();
        final File apkFile = new File(archiveFilePath);
        try {
            if ((flags & (MATCH_DIRECT_BOOT_UNAWARE | MATCH_DIRECT_BOOT_
```

```

T_AWARE)) != 0) {
    // Caller expressed an explicit opinion about what encryption
    // aware/unaware components they want to see, so fall through and
    // give them what they want
} else {
    // Caller expressed no opinion, so match everything
    flags |= MATCH_DIRECT_BOOT_AWARE | MATCH_DIRECT_BOOT_UNAWARE;
}

PackageParser.Package pkg = parser.parseMonolithicPackage(
apkFile, 0);
if ((flags & GET_SIGNATURES) != 0) {
    PackageParser.collectCertificates(pkg, 0);
}
PackageUserState state = new PackageUserState();
return PackageParser.generatePackageInfo(pkg, null, flags,
0, 0, null, state);
} catch (PackageParserException e) {
    return null;
}
}

```

生成插件Resource

VirtualApk支持单独使用插件资源和宿主插件资源合并使用。宿主插件资源合并使用hook替换宿主的Resource已经修改打包插件，我们以后再看。单独使用插件资源和我们之前使用的插件资源使用类似。

```

@WorkerThread
private static Resources createResources(Context context, File apk) {
    if (Constants.COMBINE_RESOURCES) {
        Resources resources = ResourcesManager.createResources(context, apk.getAbsolutePath());
        ResourcesManager.hookResources(context, resources);
        return resources;
    } else {
        Resources hostResources = context.getResources();
        AssetManager assetManager = createAssetManager(context, apk);
        return new Resources(assetManager, hostResources.getDisplayMetrics(), hostResources.getConfiguration());
    }
}

```

```

    }
}

private static AssetManager createAssetManager(Context context, File apk) {
    try {
        AssetManager am = AssetManager.class.newInstance();
        ReflectUtil.invoke(AssetManager.class, am, "addAssetPath",
            apk.getAbsolutePath());
        return am;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

上面的hook过程其实也是参照了系统加载生成Resource的过程。

```

public Resources getTopLevelResources(String resDir, String[] splitRes
    Dirs,
    String[] overlayDirs, String[] libDirs, int displayId,
    Configuration overrideConfiguration, CompatibilityInfo compatInfo,
    IBinder token) {
    final float scale = compatInfo.applicationScale;
    ResourcesKey key = new ResourcesKey(resDir, displayId, overrideConfiguration, scale, token);
    Resources r;
    synchronized (this) {
        //判断是否加载过该资源
        WeakReference<Resources> wr = mActiveResources.get(key);
        r = wr != null ? wr.get() : null;
        if (r != null && r.getAssets().isUpToDate()) {
            //已加载过，直接返回
            return r;
        }
    }
    //没有加载过，构建AssetManager对象
    AssetManager assets = new AssetManager();
    //将APK路径添加到AssetManager的资源路径中
    if (resDir != null) {
        if (assets.addAssetPath(resDir) == 0) {
            return null;
        }
    }
    //屏幕分辨率
    DisplayMetrics dm = getDisplayMetricsLocked(displayId);

```

```

//设备配置
Configuration config;
//创建Resources
r = new Resources(assets, dm, config, compatInfo, token);
if (false) {
    Slog.i(TAG, "Created app resources " + resDir + " " + r + ": "
        + r.getConfiguration() + " appScale="
        + r.getCompatibilityInfo().applicationScale);
}
synchronized (this) {
    //缓存资源
    mActiveResources.put(key, new WeakReference<Resources>(r));
    return r;
}
}

```

除了类似于上面创建Resource的过程，先反射创建AssertManager，在反射调用其方法addAssertPath。然后再使用这个AssertManager创建Resource。系统还提供了一个getResourcesForApplication的api，我们只要有ApplicationInfo就可以得到对应的Apk中的resource，不用反射和hook。Replugin使用的就是这个方案。

```

PackageManager pm = mContext.getPackageManager();
mPkgResources = pm.getResourcesForApplication(mPackageInfo.application
Info);

// ApplicationPackageManager
@Override
public Resources getResourcesForApplication(@NonNull ApplicationInfo a
pp){

    //...
    //省略了部分源码
    final Resources r = mContext.mainThread.getTopLevelResources(
        sameUid ? app.sourceDir : app.publicSourceDir,
        sameUid ? app.splitSourceDirs : app.splitPublicSourceDirs,
        app.resourceDirs, app.sharedLibraryFiles, Display.DEFAULT_
DISPLAY,
        null, mContext.mPackageInfo);
    if (r != null) {
        return r;
    }

}
}

```

创建插件ClassLoader

创建ClassLoader方案都差不多，因为只用DexClassLoader才能加载未安装的apk中的dex。

```
private static ClassLoader createClassLoader(Context context, File
apk, File libsDir, ClassLoader parent) {
    File dexOutputDir = context.getDir(Constants.OPTIMIZE_DIR, Con
text.MODE_PRIVATE);
    String dexOutputPath = dexOutputDir.getAbsolutePath();
    DexClassLoader loader = new DexClassLoader(apk.getAbsolutePath
(), dexOutputPath, libsDir.getAbsolutePath(), parent);

    if (Constants.COMBINE_CLASSLOADER) {
        try {
            DexUtil.insertDex(loader);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    return loader;
}
```

创建插件使用的Context

创建给插件使用的Context并修改为使用插件中相关资源。

```
@Override
public Resources getResources() {
    return this.mPlugin.getResources();
}

@Override
public AssetManager getAssets() {
    return this.mPlugin.getAssets();
}

@Override
public Resources.Theme getTheme() {
    return this.mPlugin.getTheme();
}

...

```


这样在需要的时候就可以使用这些插件中信息。比如在Instrumentation中创建插件Activity时：

```
@Override
public Activity newActivity(ClassLoader cl, String className, Intent intent) throws InstantiationException, IllegalAccessException, ClassNotFoundException {
    try {
        cl.loadClass(className);
    } catch (ClassNotFoundException e) {
        LoadedPlugin plugin = this.mPluginManager.getLoadedPlugin(intent);
        String targetClassName = PluginUtil.getTargetActivity(intent);

        Log.i(TAG, String.format("newActivity[%s : %s]", className, targetClassName));

        if (targetClassName != null) {
            Activity activity = mBase.newActivity(plugin.getClassLoader(), targetClassName, intent);
            activity.setIntent(intent);

            try {
                // for 4.1+
                ReflectUtil.setField(ContextThemeWrapper.class, activity, "mResources", plugin.getResources());
            } catch (Exception ignored) {
                // ignored.
            }

            return activity;
        }
    }

    return mBase.newActivity(cl, className, intent);
}
```

在Activity创建后替换使用插件自己的Resource资源等。

```
@Override
public void callActivityOnCreate(Activity activity, Bundle icle) {
    final Intent intent = activity.getIntent();
    if (PluginUtil.isIntentFromPlugin(intent)) {
```

```

        Context base = activity.getBaseContext();
        try {
            LoadedPlugin plugin = this.mPluginManager.getLoadedPlugin(intent);
            ReflectUtil.setField(base.getClass(), base, "mResources", plugin.getResources());
            ReflectUtil.setField(ContextWrapper.class, activity, "mBase", plugin.getPluginContext());
            ReflectUtil.setField(Activity.class, activity, "mApplication", plugin.getApplication());
            ReflectUtil.setFieldNoException(ContextThemeWrapper.class, activity, "mBase", plugin.getPluginContext());

            // set screenOrientation
            ActivityInfo activityInfo = plugin.getActivityInfo(PluginUtil.getComponent(intent));
            if (activityInfo.screenOrientation != ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED) {
                activity.setRequestedOrientation(activityInfo.screenOrientation);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    mBase.callActivityOnCreate(activity, icle);
}

```

其他

VirtualApk工程源码地址:<https://github.com/didi/VirtualAPK>

使用VirtualApk原理实践的工程地址: <https://github.com/liyonganStudy/plugin-vademo>

关于Theme和AppCompatActivity

当插件Activity继承自AppCompatActivity时, 因为坑位activity是没有定义theme的, 会报如下异常:

```

java.lang.RuntimeException: Unable to start activity ComponentInfo{com.netease.cloumusic.pluginvademo/com.netease.cloumusic.pluginengin.A$1}:

```

java.lang.IllegalStateException: You need to use a Theme.AppCompat **the me** (or descendant) with **this** activity.

```
// ActivityThread
private Activity performLaunchActivity(ActivityClientRecord r, Intent
customIntent) {
    activity.mStartedActivity = false;
    int theme = r.activityInfo.getThemeResource();
    if (theme != 0) {
        activity.setTheme(theme);
    }
    activity.mCalled = false;
    if (r.isPersistable()) {
        mInstrumentation.callActivityOnCreate(activity, r.
state, r.persistentState);
    } else {
        mInstrumentation.callActivityOnCreate(activity, r.
state);
    }
}
```

我们可以在mInstrumentation.callActivityOnCreate在设置theme。

```
public void callActivityOnCreate(Activity activity, Bundle icle) {
    ...
    activity.setTheme(plugin.getThemeId());
    ...
}
```

这时会报如下错误：

java.lang.RuntimeException: Unable to start activity ComponentInfo{com.netease.cloumusic.pluginvdemo/com.netease.cloumusic.pluginengin.A\$1}:

android.content.res.Resources\$NotFoundException: Resource ID #0x7f030017

at android.support.v7.app.AppCompatActivityDelegateImplV9.createSubDecor(AppCompatActivityDelegateImplV9.java:
at android.support.v7.app.AppCompatActivityDelegateImplV9.ensureSubDecor(AppCompatActivityDelegateImplV9.java:

```
at android.support.v7.app.AppCompatActivityDelegateImplV9.setContentView(AppCompatActivityDelegateImplV9.java:
at android.support.v7.app.AppCompatActivity.setContentView(AppCompatActivity.java:140)
```

上面我们已经在callActivityOnCreate反射将Resource， context等替换为插件中相应的Resource了，为什么还是会报资源找不到呢？跟踪源码发现是support包中的控件和相关资源找不到。

同时在inflate这些类时也没有调用插件Context中的getSystemService获取LayoutInflater。推测这些view的创建还是使用的宿主的Resource和LayoutInflater，所以存在资源找不到问题。

```
@Override
public Object getSystemService(String name) {
    // intercept CLIPBOARD_SERVICE,NOTIFICATION_SERVICE
    if (name.equals(Context.CLIPBOARD_SERVICE)) {
        return getHostContext().getSystemService(name);
    } else if (name.equals(Context.NOTIFICATION_SERVICE)) {
        return getHostContext().getSystemService(name);
    } else if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        if (mInflater == null) {
            LayoutInflater inflater = (LayoutInflater) super.getSystemService(name);
            // 新建一个，设置其工厂
            mInflater = inflater.cloneInContext(this);
            mInflater.setFactory(mFactory);
            // 再新建一个，后续可再次设置工厂
            mInflater = mInflater.cloneInContext(this);
        }
        return mInflater;
    }

    return super.getSystemService(name);
}
```

在mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);上面我们发现会先调用activity.attach

```
final void attach(Context context, ActivityThread aThread,
    Instrumentation instr, IBinder token, int ident,
    Application application, Intent intent, ActivityInfo info,
    CharSequence title, Activity parent, String id,
```

```

        NonConfigurationInstances lastNonConfigurationInstances,
        Configuration config, String referrer, IVoiceInteractor vo
iceInteractor,
        Window window) {
    attachBaseContext(context);

    mFragments.attachHost(null /*parent*/);

    mWindow = new PhoneWindow(this, window);
    mWindow.setWindowControllerCallback(this);
    mWindow.setCallback(this);
    mWindow.setOnWindowDismissedCallback(this);
    mWindow.getLayoutInflater().setPrivateFactory(this);

```

我们发现在我们发射替换Resource之前的Activity.attach方法中就已经发生了mWindow.getLayoutInflater()创建上面的window层级的View。

所以我们应该在activity.attachBaseContext时就应该替换使用插件的Context和Resource等对象。

这也是为什么VirtualApk没有支持AppCompatActivity的原因。这就导致插件工程也需要做些改变在attachBaseContext时使用插件对应的Context。

```

@Override
protected void attachBaseContext(Context newBase) {
    if (mBase != null) {
        super.attachBaseContext(mBase);
    } else {
        super.attachBaseContext(newBase);
    }
}

```

Replugin分析总结

Replugin框架还是使用占坑的方式实现的插件化，打开插件Activity时会启动坑位Activity以欺骗系统达到不在AndroidManifest.xml注册的效果。

不过它没有hook系统的Instrumentation.newActivity行为启动插件中的Activity，而是Hook了宿主的ClassLoader，增加了当load的class是坑位Activity时会去查找该坑位对应的Activity的逻辑，实现创建插件Activity的目的。

```

// ActivityThread
private Activity performLaunchActivity(ActivityClientRecord r, Intent

```

```
customIntent) {
    ...
    java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
    //创建活动
    activity = mInstrumentation.newActivity(
        cl, component.getClassName(), r.intent);
    ...
}
```

Replugin唯一的hook点其实就是上面这个cl（Classloader）。

Replugin的整体框架使用了Binder机制来进行宿主和多插件之间交互通信和数据共享。默认会使用一个常驻的进程作为Server端，插件之间和宿主之间沟通和数据共享，插件的安装，卸载，更新，状态判断等全部都在这个Server端完成。其他插件进程和宿主进程都属于Client端。这也是它整个代码框架比较复杂的原因。当然整体框架由于版本迭代导致的代码结构混乱也是导致其比较复杂的原因。

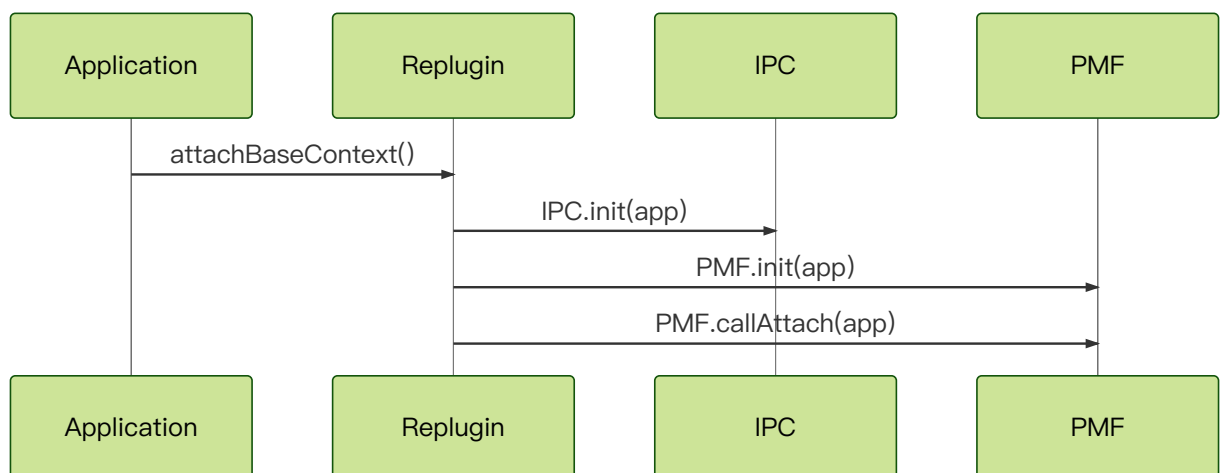
Replugin的插件管理机制也是比较复杂，设计到版本管理，坑位管理，多进程通信和同步。不过它解析apk流程以及构造插件Resource，加载插件资源和使用xml创建插件中的view这些细节的处理都很值得借鉴，同样也坚持了没有hook。

下面我们先看看Replugin的框架的初始化即使用了Binder机制来进行宿主和多插件之间交互通信和数据。

如果对Replugin分析不感兴趣，可以直接看插件的安装和加载部分，里面有使用Replugin原理实践的结果。

Replugin的框架的初始化

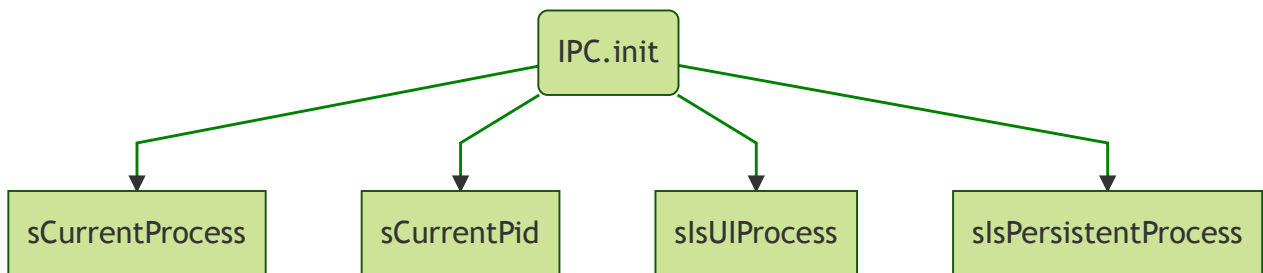
在使用Replugin的应用的Application创建时会回调attachBaseContext方法，在这里调用Replugin相应的方法进行框架的初始化工作。



Replugin: attachBaseContext()

IPC.init

初始化进程信息，判断当前进程是UI进程还是Persistent进程。主要就是通过proc文件获取当前进程名、进程id和宿主包名,然后设置常驻进程的名称，最后标记当前进程是否是ui进程和是不是常驻进程，我们知道每一个进程在创建后都会调用该进程的ActivitiThread的main方法，开启消息循环，绑定AMS，创建Application，加载provider，然后回调Application生命周期方法。所以说这里会被调用多次，多次调用导致了IPC中判断当前是否是插件管理进程时的值不同而影响后面的执行逻辑不同。

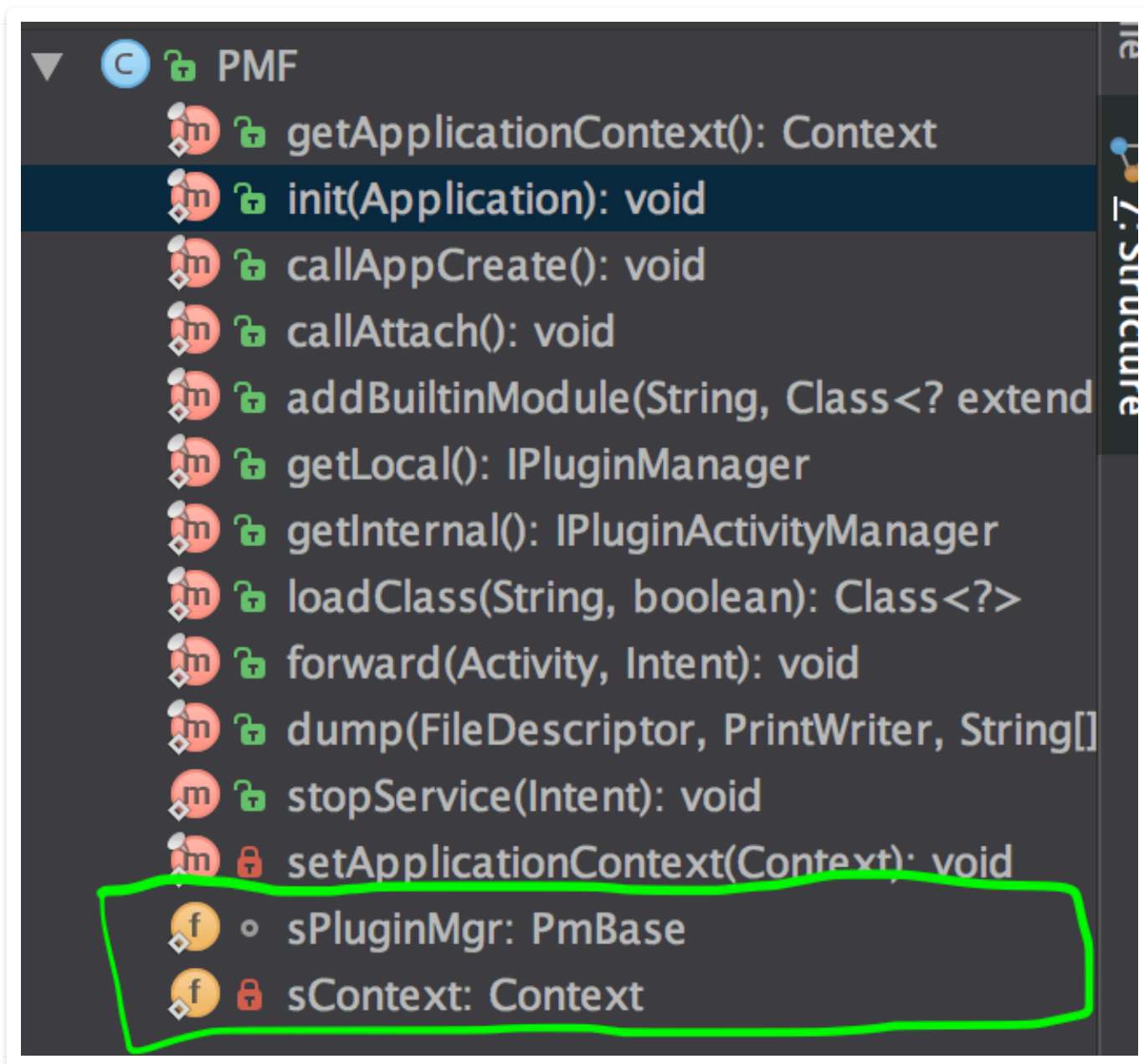


replugin有两个进程：UI进程和常驻进程。通过进程名判断是在哪个进程。

```
D lya      : =====IPC.init()
D lya      : sCurrentProcesscom.qihoo360.replugin.sample.host
D lya      : sCurrentPid5422
D lya      : sPackageNamecom.qihoo360.replugin.sample.host
D lya      : =====IPC.init()
D lya      : sCurrentProcesscom.qihoo360.replugin.sample.host:GuardService
D lya      : sCurrentPid5439
D lya      : sPackageNamecom.qihoo360.replugin.sample.host
```

PMF

PMF是插件管理对外开放的接口。PMF的初始化时创建初始了Replugin需要核心类和hook系统的ClassLoader都在这里。

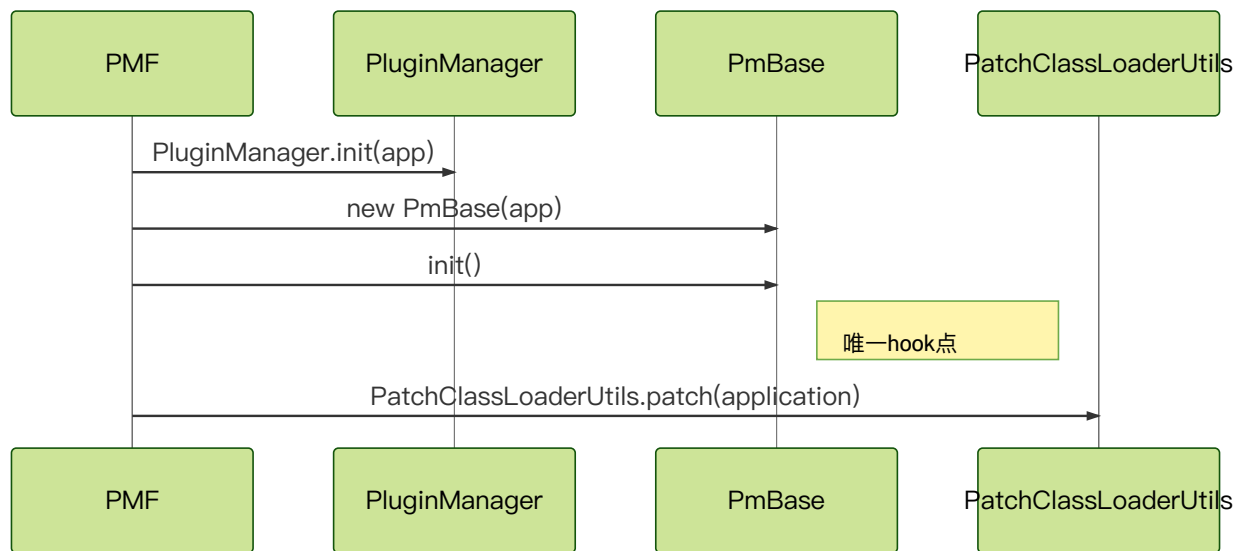


PMF.init

这个函数会做两件事情，初始化PmBase以及Hook系统的PathClassLoader。

```
public static final void init(Application application) {
    setApplicationContext(application);
    PluginManager.init(application);

    sPluginMgr = new PmBase(application);
    sPluginMgr.init();
    .....
    PatchClassLoaderUtils.patch(application);
    //Hook系统Loader，这里是系统唯一Hook点
}
```

PMF.init()

PMF.callAttach

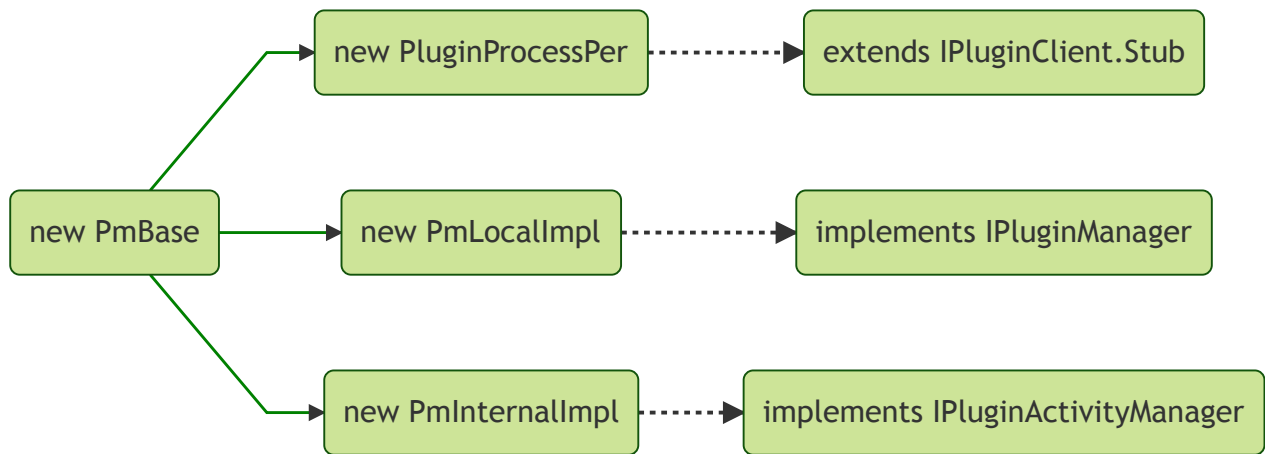
直接调用PmBase相应的方法。

```
public static final void callAttach() {
    sPluginMgr.callAttach();
}
```

PmBase

PmBase的创建和初始化，其实这也是Replugin框架核心初始化的开始， PmBase它本身和它内部引用的其他对象掌握了Replugin中很多重要的功能,例如：分配坑位、初始化插件信息、Client端连接Server端、加载插件、更新插件、删除插件、等等。

PmBase的创建

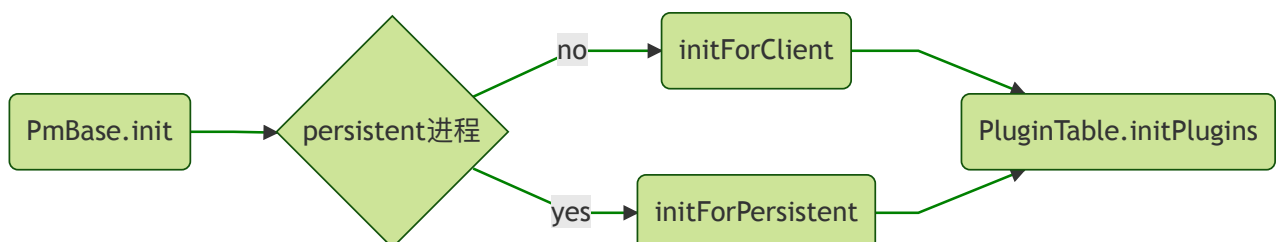


PmBase.init

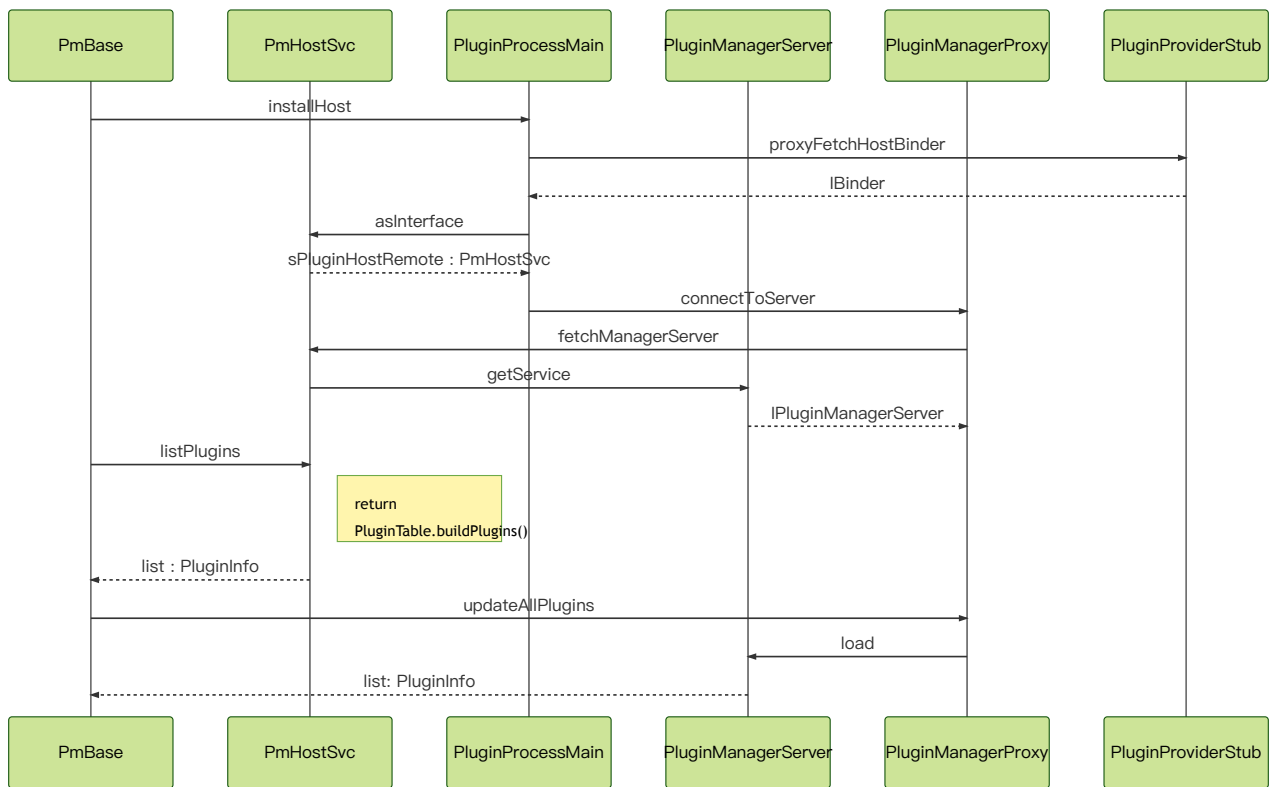
Replugin中对插件的管理是运行在独立的进程中的，所以会调用两次Applicaion的创建过程，同样也会调用两个init方法，所以这里会根据不同进程存在两个分支。

```

void init() {
    if (IPC.isPersistentProcess()) {
        mHostSvc = new PmHostSvc(mContext, this);
        PluginProcessMain.installHost(mHostSvc);
        PluginProcessMain.schedulePluginProcessLoop(PluginProcessM
ain.CHECK_STAGE1_DELAY);
        initForPersistent();
    } else {
        PluginProcessMain.installHost();
        initForClient();
    }
    // 最新快照
    PluginTable.initPlugins(mPlugins);
}
  
```



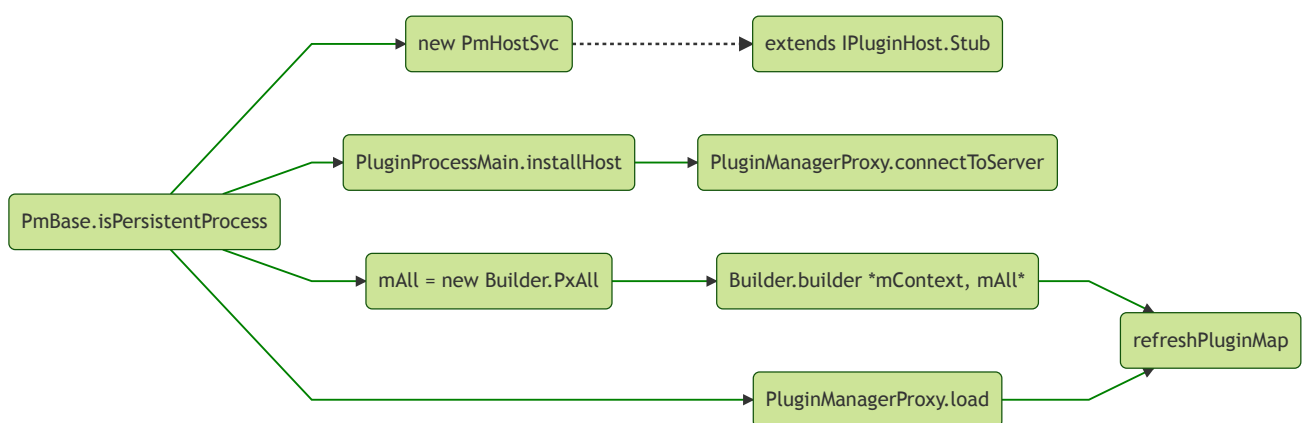
在UI进程分支中

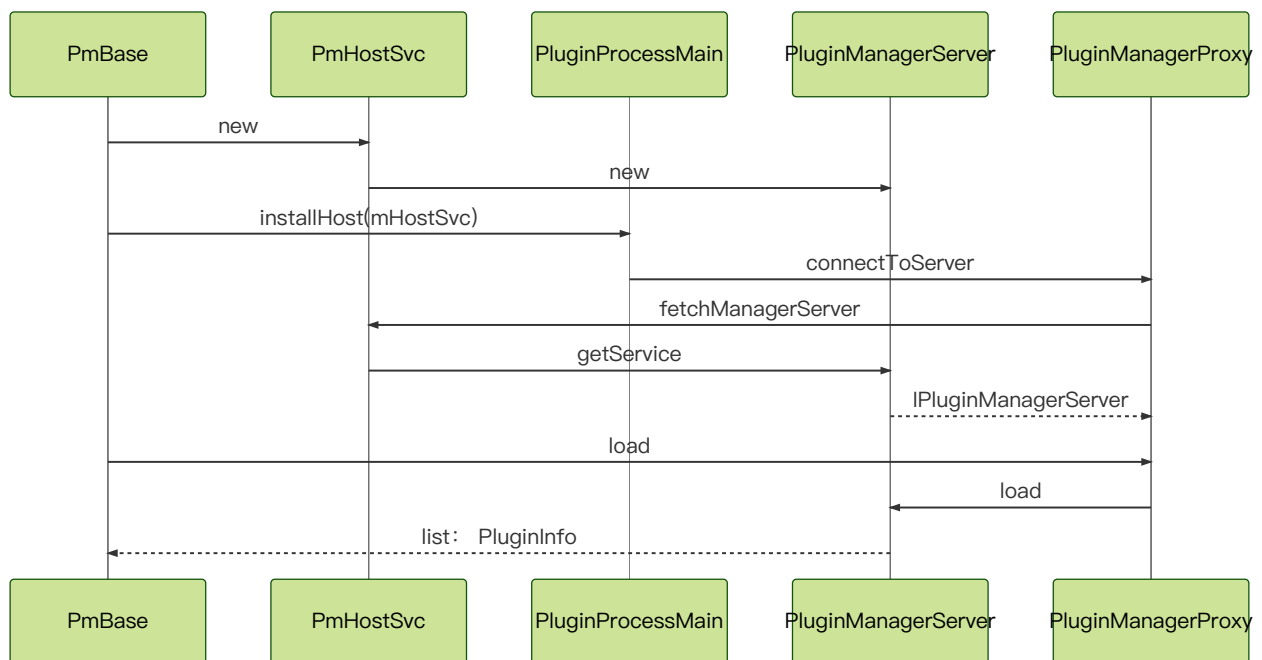


PmBase.initForClient

- PmHostSvc是两个进程间通信的Binder对象，在UI进程中通过上面的初始化过程持有其代理对象，并通过该对象与常驻进程通信。
- PmHostSvc中有个PluginManagerServer对象，用来进行插件管理的，通过上面过程，各进程可以统一使用PluginManagerProxy调用相应方法。

在常驻进程分支中





PmBase.initForPersistent

总结一下：

- 1、首先创建了一个PmHostSvc对象，这个类继承IPluginHost.Stub，是一个IPluginHost类型的Binder对象，可以说所有的插件的管理工作都是直接或者间接由它处理的，PmHostSvc它代表了Server端要处理的事情，也就是插件管理进程处理的事情
- 2、在PmHostSvc的构造方法中又创建了两个对象，一个是PluginServiceServer，这个类是用来管理插件Service的远程Server端，还有一个是PluginManagerServer，这个类在创建的时候在构造中又创建了一个继承自IPluginServiceServer.Stub的Stub对象，Stub也是一个Binder对象，通过后来查看IPluginServiceServer的代码，发现这个类掌管了所有对插件的操作，例如插件的安装、加载、卸载、更新等等
- 3、调用PluginProcessMain.installHost(mHostSvc)方法将PmHostSvc对象也就是IPluginHost类型赋值给PluginProcessMain中的字段sPluginHostLocal，这个IPluginHost是Binder对象。接着调用了IPluginHost.fetchManagerServer()方法将PluginManagerServer中的Stub对象，也就是IPluginServiceServer类型的Binder对象赋值给PluginManagerProxy类中的字段sRemote，这个IPluginServiceServer类型的Binder对象掌握了对插件的安装、卸载、更新等等的操作

如果对这方面感兴趣可以看源码或者[唯一插件化Replugin源码及原理深度剖析--初始化之](#)

[框架核心](#)的分析，不过我个人觉得Replugin这里写的过度封装，比如IPluginManagerServer是管理插件的安装加载等，只会在Service端运行，Replugin还是将它用AIDL设计。导致这部分代码结构比较混乱。

Replugin的hook原理分析

上面我们也提到，在Application的attachBaseContext中我们hook了系统的ClassLoader。

```
// PMF.init(application)
// PatchClassLoaderUtils.patch(application);
public static boolean patch(Application application) {
    Context oBase = application.getBaseContext();
    Object oPackageInfo = FieldUtils.readField(oBase, "mPackageInfo",
true);
    ClassLoader oClassLoader = (ClassLoader) FieldUtils.readField(oPackageInfo, "mClassLoader", true);
    ClassLoader cl = new RePluginClassLoader(parent, original);
    FieldUtils.writeField(oPackageInfo, "mClassLoader", cl, true);
    Thread.currentThread().setContextClassLoader(cl);
}
```

这个RePluginClassLoader只是在加载class时做了一些特别处理，其他都是调用原来的ClassLoader中的相应方法。

```
@Override
protected Class<?> loadClass(String className, boolean resolve)
throws ClassNotFoundException {
    Class<?> c = null;
    c = PMF.loadClass(className, resolve);
    if (c != null) {
        return c;
    }
    try {
        c = mOrig.loadClass(className);
        return c;
    } catch (Throwable e) {
        //
    }
    return super.loadClass(className, resolve);
}
```

当宿主加载类时会被Replugin框架拦截，进过层层调用会到PmBase,如果这个class是坑位activity的话，会被找到这个坑位activity对应的Activity，然后加载这个Activity。

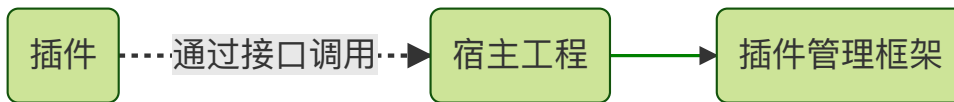
```
RePlugin.java × PmBase.java × RePluginClassLoader.java × PMF.java ×
518      * @return
519      */
520      final Class<?> loadClass(String className, boolean resolve) {
521          // 加载Service中介坑位
522          if (className.startsWith(PluginPitService.class.getName())) {
523              if (LOG) {
524                  LogDebug.i(TAG, "loadClass: Loading PitService Class... clz=" +
525                      className);
526              }
527              return PluginPitService.class;
528          }
529          //
530          if (mContainerActivities.contains(className)) {
531              Class<?> c = mClient.resolveActivityClass(className);
532              if (c != null) {
533                  return c;
534              }
535          }
536      }
```

```
/**
 * 类加载器根据容器解析到目标的activity
 * @param container
 * @return
 */
final Class<?> resolveActivityClass(String container) {
    PluginContainers.ActivityState state = mACM.lookupByContainer(container);
    plugin = state.plugin;
    activity = state.activity;
    Plugin p = mPluginMgr.loadAppPlugin(plugin);
    ClassLoader cl = p.getClassLoader();
    Class<?> c = null;
    c = cl.loadClass(activity);
}
```

关于它是如何hook系统ClassLoader和为什么这样可以保证能完全hook住系统ClassLoader，可以参见源码以及[唯一插件化Replugin源码及原理深度剖析--唯一Hook点原理](#)这篇分析文章。

插件安装和加载

Replugin插件的安装加载逻辑设计到版本迭代，以及他们几套并存的插件管理机制（包括内置插件，p-n插件，外置插件。。）和多进程机制特别复杂，为了学习和借鉴它的原理和实现，我自己新建了工程，主要遵从hook系统ClassLoader，使用占坑插件化方案，使用单进程加载管理插件。源码地址：<https://github.com/liyonganStudy/plugin-master>



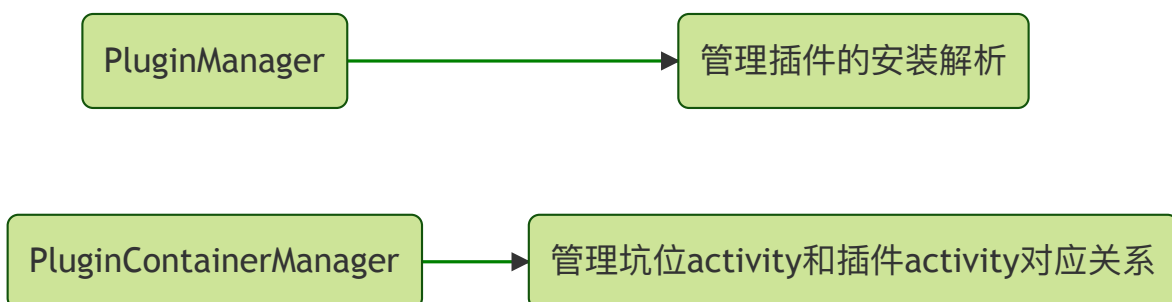
首先是在宿主工程中hook系统ClassLoader

```
public class MyApplication extends Application implements IPluginHost{
    private Handler mHandler = new Handler();

    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        PluginEngine.getInstance().attachBaseContext(this, this);
    }
}
```

```
public void attachBaseContext(Application base, IPluginHost pluginHost) {
    PatchClassLoaderUtils.patch(base);
    mPluginManager = new PluginManager(base);
    mContainerManager = new PluginContainerManager();
    mPluginLibraryHelper = new PluginLibraryHelper();
    mContext = base;
    mPluginHost = pluginHost;
}
```

其中PluginEngine就是插件管理框架对外暴露的接口，所有插件相关的操作都会调用它。**PatchClassLoaderUtils.patch(base)**就是hook系统ClassLoader，原理参照Replugin。同时创建了PluginManager和PluginContainerManager对象。



加载插件基本信息

```
private void simulateInstallExternalPlugin() {
    String demo3Apk= "demo3.apk";

    // 文件是否存在? 直接删除重来
    String pluginFilePath = getFilesDir().getAbsolutePath() + File.separator + demo3Apk;
    File pluginFile = new File(pluginFilePath);
    if (pluginFile.exists()) {
        FileUtils.deleteQuietly(pluginFile);
    }
    copyAssetsFileToAppFiles(demo3Apk, demo3Apk);
    PluginInfo info = null;
    if (pluginFile.exists()) {
        info = PluginEngine.getInstance().install(pluginFilePath);
    }
    if (info != null) {
        Intent intent = PluginEngine.getInstance().getBasicIntent(info.getPackageName(), "com.lya.testplugin.MainActivity");
        PluginEngine.getInstance().startActivity(MainActivity.this, intent);
    }
}
```

这里我把插件apk放在宿主工程的assets目录下再复制到宿主工程目录下模仿下载插件的过程。

接着调用PluginEngine.getInstance().install(pluginFilePath)安装插件。

```
public PluginInfo install(String path) {
    if (TextUtils.isEmpty(path)) {
        throw new IllegalArgumentException();
    }
    // 判断文件合法性
    File file = new File(path);
    if (!file.exists()) {
        LogUtils.log("install: File not exists. path=" + path);
        return null;
    } else if (!file.isFile()) {
        LogUtils.log("install: File not exists. path=" + path);
        return null;
    }
    return mPluginManager.pluginDownloaded(path);
}
```

```
public PluginInfo pluginDownloaded(String path) {
    // 读取apk内容
    PackageInfo packageInfo = mContext.getPackageManager().getPackageArchiveInfo(path, PackageManager.GET_META_DATA);
    if (packageInfo == null) {
        LogUtils.log("pluginDownloaded: not a valid apk. path=" + path);
        return null;
    }
    // 使用apk的MetaData初始化PluginInfo
    PluginInfo pluginInfo = PluginInfo.parseFromPackageInfo(packageInfo, path);
    // 根据PluginInfo生成Plugin对象, 更新内存表
    newPluginFound(pluginInfo);
    return pluginInfo;
}

private void newPluginFound(PluginInfo pluginInfo) {
    Plugin plugin = mPlugins.get(pluginInfo.getPackageName());
    if (plugin == null) {
        plugin = Plugin.build(pluginInfo);
    }
    if (mHostClassLoader == null) {
        mHostClassLoader = PluginManager.class.getClassLoader();
    }
    plugin.attach(mContext, mHostClassLoader);
    mPlugins.put(pluginInfo.getPackageName(), plugin);
}
```

这里其实只是解析了插件里的meta信息构造出插件基本信息。还没有完整解析插件已经构造插件的资源, context和ClassLoader。

解析插件和构造插件资源

直到真正打开插件Activity时才会解析和构造插件资源。

```
public boolean startActivity(Context context, Intent intent) {
    String pkg = intent.getComponent().getPackageName();
    String activity = intent.getComponent().getClassName();
    ActivityInfo activityInfo = mPluginManager.getActivityInfo(pkg, activity);
    if (activityInfo == null) {
        return false;
    }
    intent.putExtra(Constants.INTENT_KEY_THEME_ID, activityInfo.theme);
    ComponentName containerComponentName = mContainerManager.loadPluginActivity(activityInfo, intent, pkg, activity);
    if (containerComponentName == null) {
        return false;
    }
    intent.setComponent(containerComponentName);
    context.startActivity(intent);
    return true;
}
```

启动Activity首先会先解析插件获取目标Activity相关信息已经构造插件相关资源，然后通过坑位管理器将目标Activity替换为坑位Activity并保存该对应关系。接下来先看解析插件apk：

```
public ActivityInfo getActivityInfo(String pkg, String activity) {
    Plugin plugin = loadPlugin(pkg, Constants.LOAD_APP);
    if (plugin == null) {
        return null;
    }
    ActivityInfo activityInfo = null;
    if (!TextUtils.isEmpty(activity)) {
        activityInfo = plugin.getActivityInfo(activity);
    }
    return activityInfo;
}

private Plugin loadPlugin(String pluginName, int type) {
    Plugin plugin = mPlugins.get(pluginName);
    if (plugin == null) {
        return null;
    }
    if (!plugin.load(type)) {
        return null;
    }
    return plugin;
}
```

```
public boolean load(int type) {
    if (mLoader == null) {
        mLoader = new Loader(mPluginInfo.getPath(), mPluginInfo.getPackageName(), mPluginInfo, mHostClassLoader);
        if (!mLoader.loadDex(mHostContext, type)) {
            return false;
        }
        if (type == Constants.LOAD_APP) {
            mPlugin = mLoader.invokeEntryCreate();
        }
    }
    return true;
}
```

解析Apk的工作解耦给单独的Loader处理，我们接着看Loader.loadDex解析apk。

1.解析apk信息

```
try {
    PackageManager pm = context.getPackageManager();
    PackageInfo packageInfo = pm.getPackageArchiveInfo(mPath, PackageManager.GET_ACTIVITIES | PackageManager.GET_META_DATA);
    if (packageInfo == null || packageInfo.applicationInfo == null) {
        LogUtils.log("get package archive info null");
        return false;
    }
    packageInfo.applicationInfo.sourceDir = mPath;
    packageInfo.applicationInfo.publicSourceDir = mPath;
    if (TextUtils.isEmpty(packageInfo.applicationInfo.processName)) {
        packageInfo.applicationInfo.processName = packageInfo.applicationInfo.packageName;
    }
    if (mComponents == null) {
        mComponents = new ComponentList(packageInfo, mPath, mInfo);
        // 调整插件中 Activity 的 TaskAffinity
        adjustPluginTaskAffinity(mPluginName, mPackageInfo.applicationInfo);
    }
}
```

上面在分析VirtualApk是如何解析apk信息时已经提过。ComponetList存放的是插件Activity相关信息。

```
public ComponentList(PackageInfo pi, String path, PluginInfo pli) {
    if (pi.activities != null) {
        for (ActivityInfo ai : pi.activities) {
            ai.applicationInfo.sourceDir = path;
            if (ai.processName == null) {
                ai.processName = ai.applicationInfo.processName;
            }
            if (ai.processName == null) {
                ai.processName = ai.packageName;
            }
            mActivities.put(ai.name, ai);
        }
    }
    // 不知道下面code作用?
    // 解析 Apk 中的 AndroidManifest.xml
    // String manifest = getManifestFromApk(path);
    // 生成组件与 IntentFilter 的对应关系
    // ManifestParser.INS.parse(pli, manifest);

    mApplication = pi.applicationInfo;
}
```

2.创建插件资源

```
}
if (mPkgResources == null) {
    try {
        if (BuildConfig.DEBUG) {
            // 如果是Debug模式的话,防止与Instant Run冲突,资源重新New一个
            Resources r = pm.getResourcesForApplication(packageInfo.applicationInfo);
            mPkgResources = new Resources(r.getAssets(), r.getDisplayMetrics(), r.getConfiguration());
        } else {
            mPkgResources = pm.getResourcesForApplication(packageInfo.applicationInfo);
        }
    } catch (PackageManager.NameNotFoundException e) {
        return false;
    }
    if (mPkgResources == null) {
        return false;
    }
}
```

不同于一般通过addPath反射构造Resource, 这里通过pm.getResourcesForApplication避

免反射hook系统隐藏api。它的实现原理在上面已经分析过。

3.创建ClassLoader

```
if (mClassLoader == null) {
    String out = mInfo.getDexParentDir().getPath();
    ClassLoader parent;
    if (BuildConfig.DEBUG) {
        // 因为Instant Run会替换parent为IncrementalClassLoader, 所以在DEBUG环境里
        // 需要替换为BootClassLoader才行
        // Added by yangchao-xy & Jiongxuan Zhang
        parent = ClassLoader.getSystemClassLoader();
    } else {
        // 线上环境保持不变
        parent = getClass().getClassLoader().getParent(); // TODO: 这里直接用父类加载器
    }
    String soDir = packageInfo.applicationInfo.nativeLibraryDir;
    mClassLoader = new PluginDexClassLoader(mPath, out, soDir, parent, mHostClassLoader);
}
```

注意这里使用的是宿主Classloader的parent作为插件Classloader的parent。特别是想将宿主和插件隔离时，可以避免想加载插件的类却加载到宿主中的类的尴尬。音箱插件化也是在遇到一些坑后改成了这种生成插件Classloader的方案。

4.创建插件的Context

```
public class PluginContext extends ContextThemeWrapper {
    private final ClassLoader mNewClassLoader;
    private final Resources mNewResources;
    private final String mPlugin;
    private final Loader mLoader;
    private final Object mSync = new Object();
    private File mFilesDir;
    private File mCacheDir;
    private File mDatabasesDir;
    private LayoutInflater mInflater;
    // 不在插件中的类, 比如LinearLayout等
    private HashSet<String> mIgnores = new HashSet<>();
    HashMap<String, Constructor<?>> mConstructors = new HashMap<>();

    // private ContextInjector mContextInjector;

    LayoutInflater.Factory mFactory = (name, context, attrs) -> {
        return handleCreateView(name, context, attrs);
    };

    public PluginContext(Context base, int themeres, ClassLoader cl, Resources r, String plugin, Loader loader) {
        super(base, themeres);

        mNewClassLoader = cl;
        mNewResources = r;
        mPlugin = plugin;
        mLoader = loader;
    }
}
```

这样在插件activity的attachBaseContext时attach插件自己的context，可以找到插件自己的资源，主题，ClassLoader等。

```

@Override
public ClassLoader getClassLoader() {
    if (mNewClassLoader != null) {
        return mNewClassLoader;
    }
    return super.getClassLoader();
}

@Override
public Resources getResources() {
    if (mNewResources != null) {
        return mNewResources;
    }
    return super.getResources();
}

@Override
public AssetManager getAssets() {
    if (mNewResources != null) {
        return mNewResources.getAssets();
    }
    return super.getAssets();
}

```

这里还有一个需要处理的就是创建xml中的View时使用LayoutInflater

```

public static LayoutInflater from(Context context) {
    LayoutInflater LayoutInflater =
        (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    if (LayoutInflater == null) {
        throw new AssertionError("LayoutInflater not found.");
    }
    return LayoutInflater;
}

```

```

@Override
public Object getSystemService(String name) {
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        if (mInflater == null) {
            mInflater = LayoutInflater.from(getBaseContext()).cloneInContext(this);
        }
        return mInflater;
    }
    return getBaseContext().getSystemService(name);
}

```

这里我们需要复写修改在创建view时使用插件的ClassLoader和资源。

```

@Override
public Object getSystemService(String name) {
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        if (mInflater == null) {
            LayoutInflater inflater = (LayoutInflater) super.getSystemService(name);
            // 新建一个，设置其工厂
            mInflater = inflater.cloneInContext(this);
            mInflater.setFactory(mFactory);
            // 再新建一个，后续可再次设置工厂
            mInflater = mInflater.cloneInContext(this);
        }
        return mInflater;
    }
    return super.getSystemService(name);
}

```

```

// 构造器缓存
Constructor<?> construct = mConstructors.get(name);

// 缓存失败
if (construct == null) {
    // 找类
    Class<?> c = null;
    boolean found = false;
    try {
        c = mNewClassLoader.loadClass(name);
    } catch (ClassNotFoundException e) {
        // 失败，不管
        LogUtils.log("handleCreateView plugin classLoader not find: class " + name);
    }
    // 找构造器
    try {
        construct = c.getConstructor(Context.class, AttributeSet.class);
        mConstructors.put(name, construct);
    } catch (Exception e) {
        throw new InflateException(attrs.getPositionDescription() + ": Error inflating
    }
}

// 构造
try {
    // 只有开启“详细日志”才会输出，防止“刷屏”现象
    return (View) construct.newInstance(context, attrs);
} catch (Exception e) {
}

```

启动坑位Activity但创建插件Activity

```

ComponentName containerComponentName = mContainerManager.loadPluginActivity(activityInfo, intent, pkg, activity);
if (containerComponentName == null) {
    return false;
}
intent.setComponent(containerComponentName);
context.startActivity(intent);

```

接下来会通过坑位管理器找到合适的坑位Activity,并保存下坑位Activity和实际要启动的Activity之间的对应关系。

之后在ActivityThread创建Activity时，由于我们hook了宿主的ClassLoader，所以下面的cl其实就是我们hook的ClassLoader。

```

ContextImpl appContext = createBaseContextForActivity(r),
Activity activity = null;
try {
    java.lang.ClassLoader cl = appContext.getClassLoader();
    if (appContext.getApplicationContext() instanceof Application) {
        activity = ((Application) appContext.getApplicationContext())
            .instantiateActivity(cl, component.getClassName(), r.intent);
    }
    if (activity == null) {
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent);
    }
}

```

```

@Override
protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
    Class<?> c;
    c = PluginEngine.getInstance().loadClass(className, resolve);
    if (c != null) {
        return c;
    }
    try {
        c = mOrig.loadClass(className);
        LogUtils.logForClassLoader("load class: " + className + " in hostclassloader use origin classl
        return c;
    } catch (Throwable e) {
        e.printStackTrace();
        LogUtils.log("use default classloader load class: " + className + " fail");
    }
    return super.loadClass(className, resolve);
}

```

我们hook的ClassLoader在加载任何类时都会被PluginEngine.getInstance().loadClass拦截。特别是如果是坑位Activity时我们会用插件立面的ClassLoader加载，这样就完成了启动坑位Activity,骗过AMS的校验，创建的确实插件Activity的效果。

```

public Class<?> loadClass(String name, boolean resolve) {
    if (mContainerManager.isContainerActivity(name)) {
        return mContainerManager.resolveActivityClass(name);
    }
    return null;
}

```

```

public Class<?> resolveActivityClass(String activity) {
    ActivityState activityState = mStates.get(activity);
    String pluginName = activityState.pluginName;
    String activityName = activityState.activityName;
    return PluginEngine.getInstance().resolveActivityClass(pluginName, activityName);
}

```

```

public Class<?> resolveActivityClass(String pluginName, String activityName) {
    Plugin plugin = loadPlugin(pluginName, Constants.LOAD_APP);
    if (plugin == null) {
        return null;
    }
    return plugin.resolveActivityClass(activityName);
}

```



```

public Class<?> resolveActivityClass(String activityName) {
    ClassLoader cl = mLoader.getClassLoader();
    Class<?> c = null;
    try {
        c = cl.loadClass(activityName);
        LogUtils.logForClassLoader("=====PluginDexClassLoader resolveActivityClass: " + a
    } catch (Throwable e) {
    }
    return c;
}

```

一起其他功能支持

加载插件中Fragment

歌词模板以前是加载Fragment实现的，不过对xml的支持不行，不能再xml中使用资源是硬伤，使用上面的LayoutInflater方案可以加载Fragment并在xml中使用资源。

```

private void installFragment(String packageName) {
    //代码使用插件Fragment
    ClassLoader d1ClassLoader = PluginEngine.getInstance().getPluginClassLoader(packageName); //获取插件的ClassLoader
    try {
        Fragment fragment = d1ClassLoader.loadClass(packageName + ".MainFragment").asSubclass(Fragment.class).newInstance()
        getSupportFragmentManager().beginTransaction().add(R.id.fragmentContainer, fragment).commit(); //添加Fragment到UI
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

```


```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    Context context;
    try {
        IPluginEngine pluginEngine = Entry.getInstance().getPluginEngine();
        context = pluginEngine.getPluginContext(Constants.PACKAGE_NAME);
    } catch (Throwable throwable) {
        context = getContext();
    }
    return LayoutInflater.from(context).inflate(R.layout.fragment_main, container, false);
}


```

inflater使用插件的Context就可以加载xml中的资源和插件自定义控件等。

插件和宿主通信



```
public interface IPlugin {  
    void sendToPlugin(int command, Serializable content);  
  
    void attach(IPluginHost iPluginHost);  
}
```



```
public interface IPluginHost {  
    void sentToHost(String pluginName, int code, Serializable content);  
  
    IPluginEngine getPluginEngine();  
}
```

亲测使用接口可以进行通信。