爱奇艺基线App-插件化原理

陈晓峰

iQIYI 爱奇艺

悦 享 品 质

前言

定义

所谓**插件，就是一个模**块（比如一个dex文件，或一个apk），可以在运行期，被宿主App动态安装，使用。

优点

用户端：

1. 不用重新安装 APK 就能升级应用功能，提升用户体验。
2. 用户可以按需加载不同的模块，实现灵活的功能配置。

开发端：

1. 可快速修复线上 BUG 和更新功能，减少宿主app发版频率。
2. 模块化、解耦合、并行开发
3. 减少宿主apk大小

前言

基线app对插件的管理：启动请求插件列表，插件有版本号，发现有新版本**或从未下**载过，则下载。

url: https://iface2.iqiyi.com/fusion/3.0/plugin?


```
▼ object {2}
    code : 0
  ▼ data {1}
    ▼ plugins {1}
      ▼ plugin [25]
        ▶ 0 {29}
        ▶ 1 {29}
        ▶ 2 {29}
        ▶ 3 {30}
        ▶ 4 {30}
        ▶ 5 {29}
        ▶ 6 {29}
        ▶ 7 {29}
        ▶ 8 {30}
        ▼ 9 {13}
            baseplugins : android.app.fw
            crc : 8A922C62
            scrc : 8A922C62
            url : https://cdndata.video.iqiyi.com/cdn/qiyiapp/20190525/231626w5ed68c92166601ff616eefca8e1f546c/com.iqiyi.imall.apk
            priority : 2
            plugin_icon_url : http://pic0.iqiyipic.com/common/20171226/94ebe59fc35c4b8e98898072dbb67985.png
            c_dl_at : 1
            uninstall : 0
            plugin_id : 6958
            plugin_name : 爱奇艺电商
```


插件地址，可以是apk，也可以是放了dex的zip包

前言

问题点

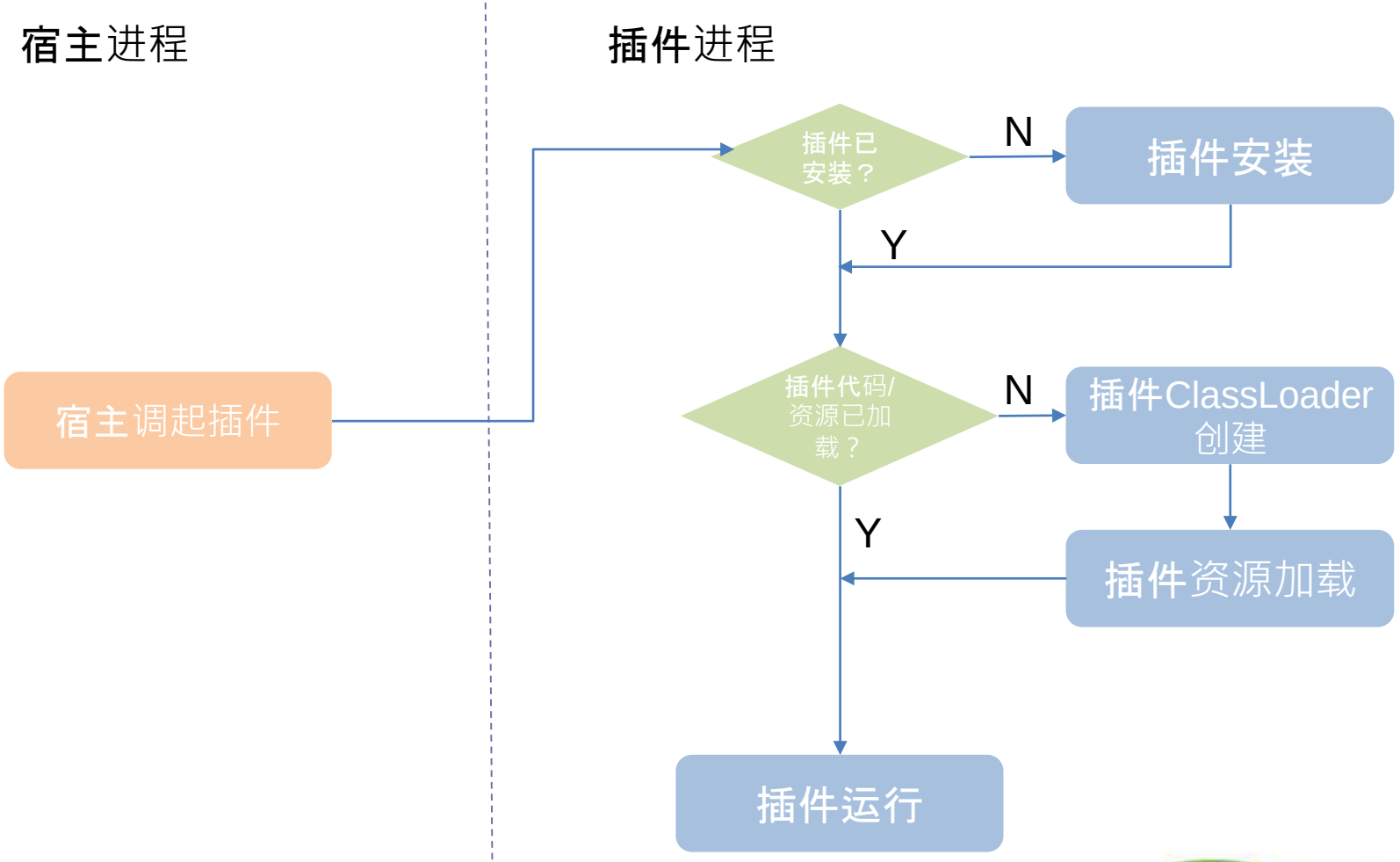
插件中的类和资源在宿主App中都不存在，四大组件也没有在AndroidManifest中注册，自然也不会被系统安装，只是放置在宿主App设备上的一个apk包。所以插件框架都需要解决以下4个问题：

1.插件代码的加载
2.插件资源的加载
3.插件组件的生命周期
4.插件和宿主互不影响

思路

1. 新建DexClassLoader对象，专门加载插件dex文件的组件类
2. 新建Resources对象，专门管理插件apk的资源
3. 用新建的ClassLoader实例化插件的组件(如XXActivity)，用一个在宿主注册的代理activity，间接管理XXXActivity的生命周期。
4. 使用Android多进程功能，让插件和宿主处于不同进程，互不影响。

整体流程



插件安装

插件的安装分为内置插件（asset**目**录）**和**线上插件两部分。

•**内置插件**：

放宿主apk**打包**，约定**存放在***assets/pluginapp/<plugin_pkg_name>.apk***形式**，**安装**时解压到*/data/data/<host_pkg_name>/app_pluginapp***目**录；

•线**上插件**：

宿主在运行期，**从后台下**载到sdcard**目**录上，**安装**时解压到*/data/data/<host_pkg_name>/app_pluginapp***目**录下；

插件安装

具体安装过程：

1. 拷贝apk到内置存储区，重命名为<plugin_pkg_name>.apk
2. 解压apk中的so库到app_pluginapp/<plugin_pkg_name>/lib目录
3. DexOptimizer解压apk中的dex并优化
   (Android N以上版本,使用*dex2oat*命令优化，目的是加快在ART虚拟机上的运行)

插件ClassLoader创建

Java中的类都是通过ClassLoader加载的,其子类**DexClassLoader**，可以加载指定路径的dex、apk或jar文件的Class，它是插件化的技术基础。

具体加载代码，后面详细给出

插件资源加载

Android APP运行除了类还有资源，对于Android来说，资源是通过AssetManager和Resources这两个类管理。App在运行时查找资源是通过当前Context的Resource实例中查找，在Resource内部是通过AssetManager管理当前的资源，AssetManager维护了资源包路径的数组。插件化的原理，就是将插件的资源路径添加到AssetManager的资源路径数组中，通过反射AssetManager的隐藏方法*addAssetPath*实现插件资源的加载。

插件资源加载

关键代码

1. 创建AssetManager & Resources， 资源指向解压的apk路径。

```
class PluginLoadedApk {
...
private void createPluginResource() {
    try{
        AssetManager am = AssetManager.class.newInstance();
        Method addAssetPath = AssetManager.class.getDeclaredMethod("addAssetPath", String.class);
        addAssetPath.setAccessible(true);
        addAssetPath.invoke(am, pluginApkPath);
        mPuginResources = new Resources(am, hostResource.getDisplayMetrics(), hostResources.getConfiguration());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public Resources getResources() {
    return mPuginResources;
}

...
```

通过反射添加路径，pluginApkPath是插件apk的绝对路径

插件运行

也就是运行插件各种组件：Activtiy、Service、BroadcastReceiver, ContentProvider。

接下来重点介绍Activity的运行流程。

插件运行

Activity运行

在Android中，我们一般要启动一个Activity，会调用类似的代码

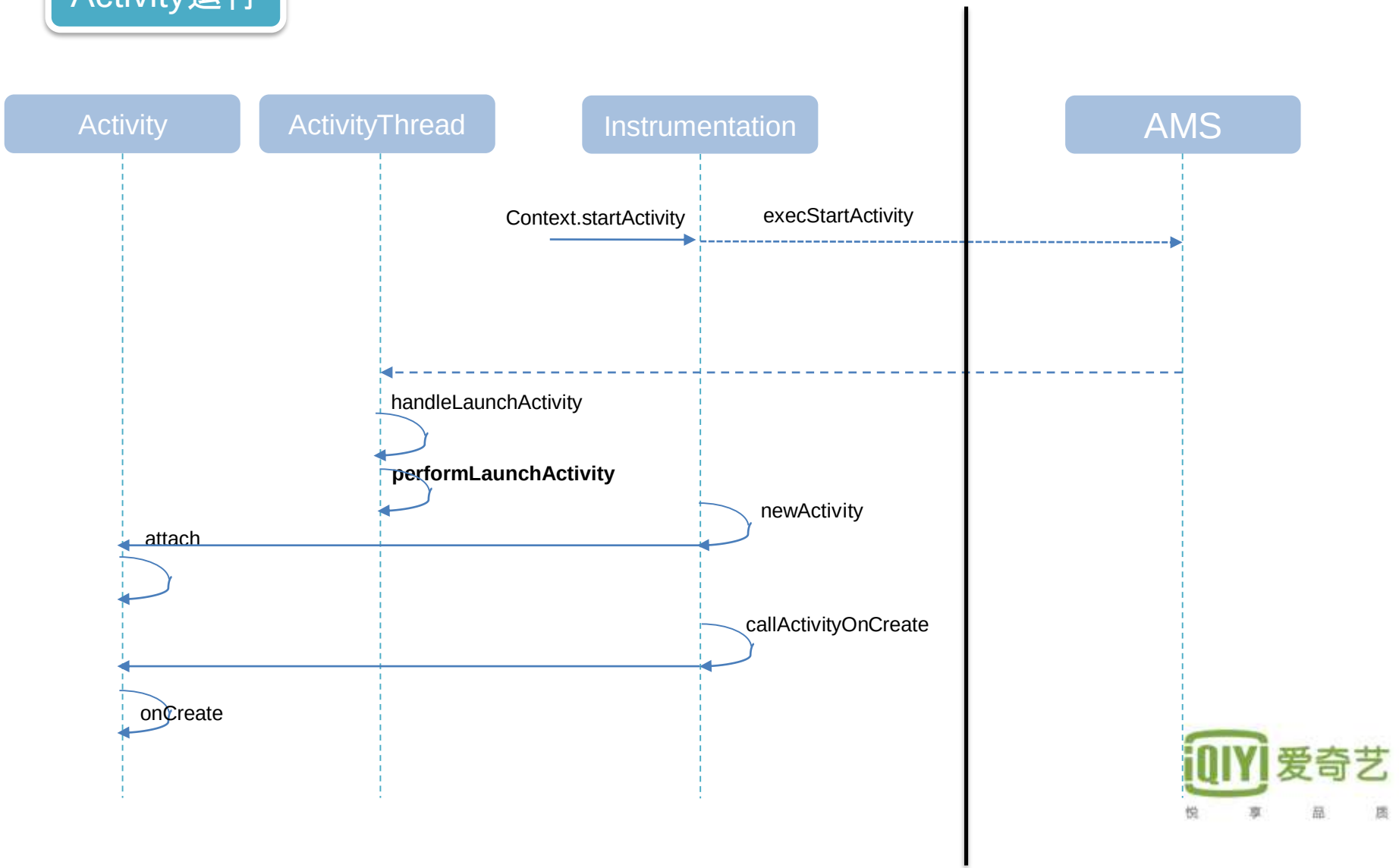
```
Intent intent = new Intent(ActivityA.this, ActivityB.class);
context.startActivity(intent);
```

Activity启动过程主要流程有如下几步：

- 1. 调用startActivity，直接或间接执行Instrumentation的execStartActivity()方法
- 2. Instrumentation跨进程通信，调用到系统服务AMS进程的startActivity方法
- 3. AMS会先判断权限，校验Activity的合法性，然后调用ActivityStackSupervisor和ActivityStack进行一系列的交互来确定和管理Activity的栈
- 4. AMS通过ApplicationThread进行跨进程通信，重新回调到App进程的ActivityThread端
- 5. 通过ActivityThread中的H类（一个Handler实例）发送消息驱动，执行ActivityThread#performLaunchActivity()方法，会再次调用Instrumentation#newActivity()来创建Activity并实例化
- 6. 调用Activity#attach()方法初始化Activity的基本信息，接着调用callActivityOnCreate来唤醒Activity，走到Activity#onCreate()生命周期回调

插件运行

Activity运行



插件运行

Activity运行

ActivityThread#**performLaunchActivity**比较关键，后续的newActivity, attach, callActivityOnCreate**都由它触**发。关键代码如下：

```
private Activity performLaunchActivity() {
    ...
    ClassLoader cl = appContext.getClassLoader();
    activity = this.mInstrumentation.newActivity(cl, component.getClassName(), r.intent);//实例化一个类
    Application app = r.packageInfo.makeApplication(false, this.mInstrumentation);//没有application，则创建
    activity.attach(appContext, this, this.getInstrumentation()....);//各种变量传给activity，然后创建PhoneWindow
    activity.setTheme(theme);//设置主题
    mInstrumentation.callActivityOnCreate(activity, r.state);//调用Activity的onCreate
    ...
}
```

插件运行

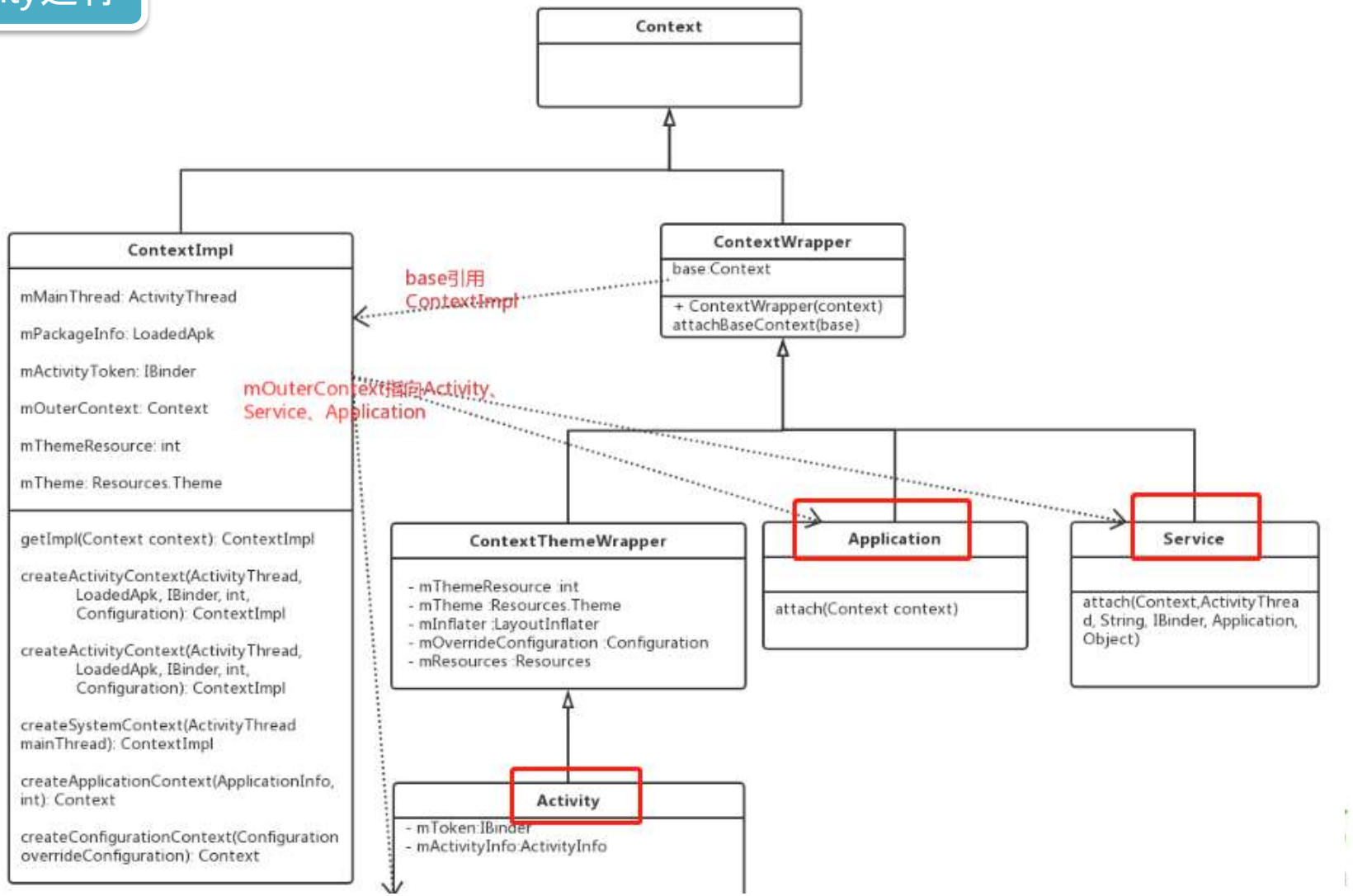
Activity运行

ps:
ActivityThread是app的主进程管理类，带main入口。
Instrumentation是在创建ActivityThread时，实例化的，也是单例。
如果一个app有多个进程，则会对应创建多个ActivityThread & Instrumentation。

context.startActivity(intent);
这个context，既可能是Service，也可能是Activity，或Application，如何都调用到Instrumentation？

插件运行

Activity运行



插件运行

Activity运行

Context有startActivity抽象函数，由ContextImpl实现。Service/Appliction调用时，通过父类的ContextWrapper间接调用ContextImpl。Activity则有本地mInstrument变量指向Instrumentation对象，可以直接调用。

```
public void startActivity(Intent intent, Bundle options) {
    this.warnIfCallingFromSystemProcess();
    if ((intent.getFlags() & 268435456) == 0 && options != null && Activi
        throw new AndroidRuntimeException("Calling startActivity() from o
    } else {
        this.mMainThread.getInstrumentation().execStartActivity(this.getO
    }
}
```

插件运行

Activity运行

根据前面的流程，要直接启动插件activity，则第3步，AMS所做的校验通过不了（判断当前需要启动的Activity是否被系统安装过）。

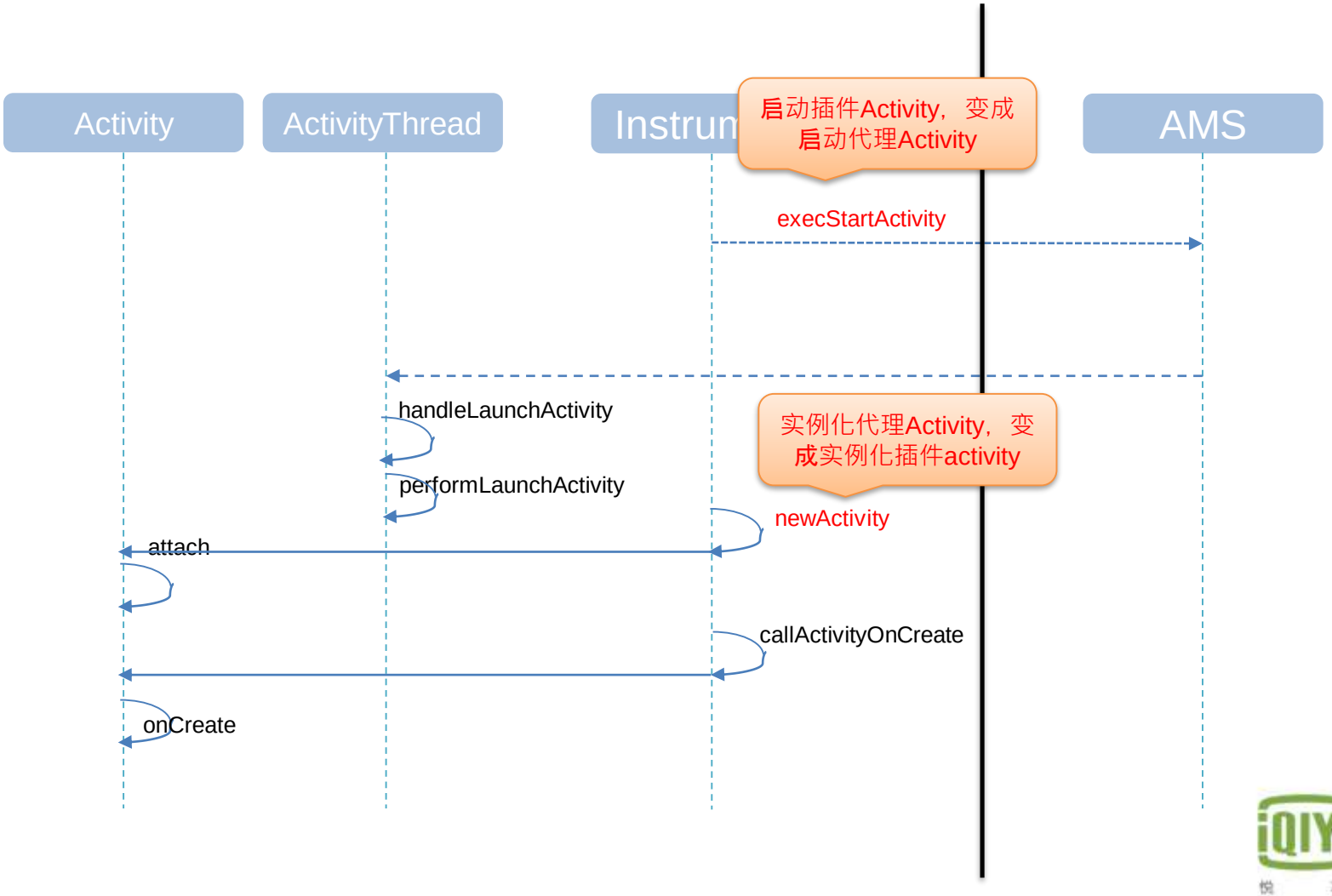
解决思路:

修改Instrumentation的execStartActivity()， 把intent中的插件Activity替换成代理的Activity，骗过系统AMS进程的校验；

等Intent再次传递到App端创建Activity对象时，不要创建代理Activity对象，而是插件Activity对象，从而给插件Activity注入了生命周期。

插件运行

解决思路:



插件运行

完整流程

1. Host app调起PluginManager#launchPlugin(包名)
2. 启动ServiceProxy1，该service声明在插件进程，将启动新进程，且回调执行host的Application#onCreate，这里会执行Neptune#init，反射替换ActivityThread的mInstrumentation为新建的NeptuneInstrument对象。
3. ServiceProxy1调用PluginManager#loadPluginAsync异步加载插件，创建PluginLoadedApk，内部创建插件的DexClassLoader和插件的Resource
4. PluginManager#doRealLaunch，真正启动插件，查找插件的主Activity*(即Manifest声明action为MAIN, category为LAUNCHER)*，执行mContext.startActivity(PluginActivity1)
5. 走到NeptuneInstrument#execStartActivity，内部执行ComponentFinder#switchToActivityProxy，将插件activity替换为代理activity，然后调起AMS（ActivityManager.getService().startActivity）。
6. AMS鉴权，各种操作，回调ActivityThread# performLaunchActivity，内部执行NeptuneInstrument#newActivity。
7. NeptuneInstrument#newActivity使用第3步建的DexClassLoader，实例化插件activity对象，并将第3步建的Resource，赋值给该对象。
8. 执行插件activity#attach，赋值各种activity变量，并创建window。
9. 第5步的performLaunchActivity，回调执行activity#onCreate，开始生命周期
10. ActivityThread的mActivities列表，包含插件activity，其Resume, Pause等生命周期，将由ActivityThread触发

插件运行

完整流程

PS:
当已经启动一个插件activity，如PluginActivity1，再继续跳转插件其他页面，即startActivity(PluginActivity1.this, PluginActivity2.class)，则流程从第5步开始，NeptuneInstrument继续为PluginActivity2寻找一个代理，一步步继续下去。

插件运行

Activity运行　　关键代码

这里给出一些有意思的关键代码：

1. 第二步的，Neptune#init，hook ActivityThread的mInstrument变量

```
private static void hookInstrumentation() {
    ActivityThread activityThread = ActivityThread.currentActivityThread();
    PluginInstrument pluginInstrument = new NeptuneInstrument(hostInstr);
    ReflectionUtils.on(activityThread).set("mInstrumentation", pluginInstrument);
}
```

其中NeptuneInstrument继承Instrument，且重写execStartActivity(), newActivity()等函数

插件运行

Activity运行　　关键代码

2. 第三步，插件的ClassLoader的创建

```
public PluginLoadedApk() {
    //创建插件ClassLoader
    createNewClassLoader();
    // 创建插件资源
    createPluginResource();
    // 注册静态广播
    installStaticReceiver();
}
```

```
private boolean createNewClassLoader() {
    //PluginClassLoader 继承 DexClassLoader, 支持MultiDex
    mPluginClassLoader = new PluginClassLoader(mPluginPackageInfo, mPluginPath,
                        optDir.getAbsolutePath(), mPluginPackageInfo.getNativeLibraryDir(), mParent);
    sAllPluginClassLoader.put(mPluginPackageName, mPluginClassLoader);
    return handleNewDependencies();
}
```

ClassLoader传递参数，包括apk路径，dex解压路径，so库路径。例如：

/data/user/0/com.iqiyi.host.sample/app_pluginapp/com.iqiyi.plugin.sample.1.2.apk
/data/user/0/com.iqiyi.host.sample/app_pluginapp/com.iqiyi.plugin.sample
/data/user/0/com.iqiyi.host.sample/app_pluginapp/com.iqiyi.plugin.sample/lib

插件运行

Activity运行　　关键代码

3. **第五步**，Activity偷梁换柱。

```
public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activity target,
        Intent intent, int requestCode, Bundle options) {
    ComponentFinder.switchToActivityProxy(mPkgName, intent, requestCode, who);
    try {
        Class<?>[] paramTypes = new Class[]{Context.class, IBinder.class, IBinder.class, Activity.class,
                Intent.class, int.class, Bundle.class};
        return mInstrumentRef.call( name: "execStartActivity", sMethods, paramTypes, who, contextThread, token,
    } catch (Exception e) {
        // ignore
    }
    return null;
}
```

将intent指向的插件activity，换为代理activity

继续走正常路程，调起AMS

插件运行

Activity运行　　关键代码

4. 第七步， NeptuneInstrument#newActivity再次偷梁换柱，实例化插件activity。

```
public Activity newActivity(ClassLoader cl, String className, Intent intent) throws InstantiationException, Illegal
    if (className.startsWith(ComponentFinder.DEFAULT_ACTIVITY_PROXY_PREFIX)) {
        // 插件代理Activity，替换回插件真实的Activity
        String[] result = IntentUtils.parsePkgAndClsFromIntent(intent);
        String packageName = result[0];
        String targetClass = result[1];

        PluginDebugLog.runtimeLog(TAG, msg: "newActivity: " + className + ", targetClass: " + targetClass);
        if (!TextUtils.isEmpty(packageName)) {
            PluginLoadedApk loadedApk = PluginManager.getPluginLoadedApkByPkgName(packageName);
            if (loadedApk != null && targetClass != null) {
                Activity activity = mHostInstr.newActivity(loadedApk.getPluginClassLoader(), targetClass, intent);
                activity.setIntent(intent);

                if (!dispatchToBaseActivity(activity)) {
                    // 这里需要替换Resources，是因为ContextThemeWrapper会缓存一个Resource对象，而在Activity#attach(
                    // Activity#onCreate()之间，系统会调用Activity#setTheme()初始化主题，Android 4.1+
                    ReflectionUtils.on(activity).setNoException("mResources", loadedApk.getPluginResource());
                }
```

获得之前存的插件class和包名

实例化插件activity

反射，设置插件activity的mResources

插件运行

Activity运行　小问题

插件activity，能否访问宿主app资源或其他插件资源？

可以。AssetManager可以记录多个资源路径。只需调用addAssetPath即可。不过这是private函数，需要反射调用。

```
// 添加宿主的资源到插件的AssetManager
String hostSourceDir = mHostContext.getApplicationInfo().sourceDir;
ReflectionUtils.on(mAssetManager).call("addAssetPath", sMethods, paramTypes,
        hostSourceDir);
```

ps: 传递的路径，是apk的绝对路径，AssetManager会自动解析内部的resources.asrc以及资源文件，维护资源ID与资源名称的对应关系。

插件运行

Activity运行　小问题

当然，如果资源要共享，则编译插件apk期间，需要依赖宿主/其他插件的aar，但可以不打包到插件apk。

此外，插件共享，会引入资源冲突，原因在于不同apk的资源id可能相同。

解决办法：

资源id是由8位16进制数表示，表示为0xPPTTNNNN。PP段用来区分包空间，默认只区分了应用资源和系统资源，TT段为资源类型，NNNN段在同一个APK中从0000递增。

列别	PP段	TT段	NNNN段
应用资源	0x7f	04	0000
系统资源	0x01	04	0000

所以方案是，编译期间，配置资源ID的PP段，对于不同的插件使用不同的PP段，从而区分不同插件的资源。

插件运行

Activity运行　小问题

插件activity，能否访问宿主app或其他插件apk的代码？

可以。如果当前插件需要依赖其他插件，可以实例化对应的ClassLoader，然后在findClass时，遍历所有的ClassLoader。

```
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    // 根据Java ClassLoader的双亲委托模型，执行到此在parent ClassLoader中没有找到
    // 类似的，我们优先在依赖的插件ClassLoader中查找
    for (ClassLoader classLoader : dependencies) {
        try {
            Class<?> c = classLoader.loadClass(name);
            if (c != null) {
                // find class in the dependency
                return c;
            }
        } catch (ClassNotFoundException e) {
            // ClassNotFoundException thrown if class not found ini dependency class lo
        }
    }
    // If still not found, find in this class loader
    try {
        return super.findClass(name);
    } catch (ClassNotFoundException e) {
```

dependencies是其他插件/宿主的classLoader

dependencies找不到，就从自己身上找

插件运行

Service运行

Service的启动有两种方式startService和bindService。流程和Activity非常类似：

1. 调用startService, 会执行ContextImpl的startService方法
2. ContextImpl通过binder跨进程通信，调用到系统服务AMS进程的startService
3. AMS进程校验Service的权限和合法性，通过校验之后会通过ApplicationThread再次回调到App进程
4. ActivityThread的H类（Handler实现）发送消息驱动，调用ActivityThread#handleCreateService创建Service实例， 回调Service#onCreate方法

插件运行

Service运行　完整方案

1. PluginManager#launchPlugin(intent)，将intent的targetService，替换为代理service。
2. 调用context.startService()，跨进程调起AMS，鉴权，回调ActivityThread，创建代理Service对象。
3. 代理Service在onStartCommand/onBind中解析targetService。反射创建targetService对象，并调用attach方法将代理的token传给targetService。
4. 代理Service在各种生命周期的回调，间接调起targetService的回调。

```
@Override
public void onStart(Intent intent, int startId) {
    PluginDebugLog.log(TAG, msg: "ServiceProxy1>>>>>onStart():" + (intent == null ? "null" : int
    if (intent == null) {
        super.onStart( intent: null, startId);
        return;
    }
    String targetClassName = IntentUtils.getTargetClass(intent);
    String targetPackageName = IntentUtils.getTargetPackage(intent);
    PluginServiceWrapper currentPlugin = loadTargetService(targetPackageName, targetClassName);

    if (currentPlugin != null && currentPlugin.getCurrentService() != null) {
        currentPlugin.updateServiceState(PluginServiceWrapper.PLUGIN_SERVICE_STARTED);
        currentPlugin.getCurrentService().onStart(intent, startId);
    }
```

间接控制插件service的声明周期

插件运行

BroadCastReceiver运行

BroadcastRecevier分为静态广播和动态广播两种。静态广播是声明在AndroidManifest之中的，在apk安装到设备上之后，PMS自动注册到系统中的；而动态广播是我们在代码中显式调用Context#registerReceiver()注册的。

插件的静态广播是没有声明在宿主的AndroidManifest中，系统自然无法为其注册。

解决思路：加载插件时，解析插件AndroidManifest中的静态广播，转换为动态广播进行注册。

插件运行

关于多进程

前文还提到了，让插件可以在独立进程中运行。
实现方案如下：
在Manifest中，设置代理activity/service**在独立**进程中运行

```
<activity
    android:name="org.qiyi.pluginlibrary.component.InstrActivityProxy1"
    android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSiz
    android:process=":plugin1"
    android:screenOrientation="portrait" />
```

```
<service
    android:name="org.qiyi.pluginlibrary.component.ServiceProxy1"
    android:process=":plugin1" />
```

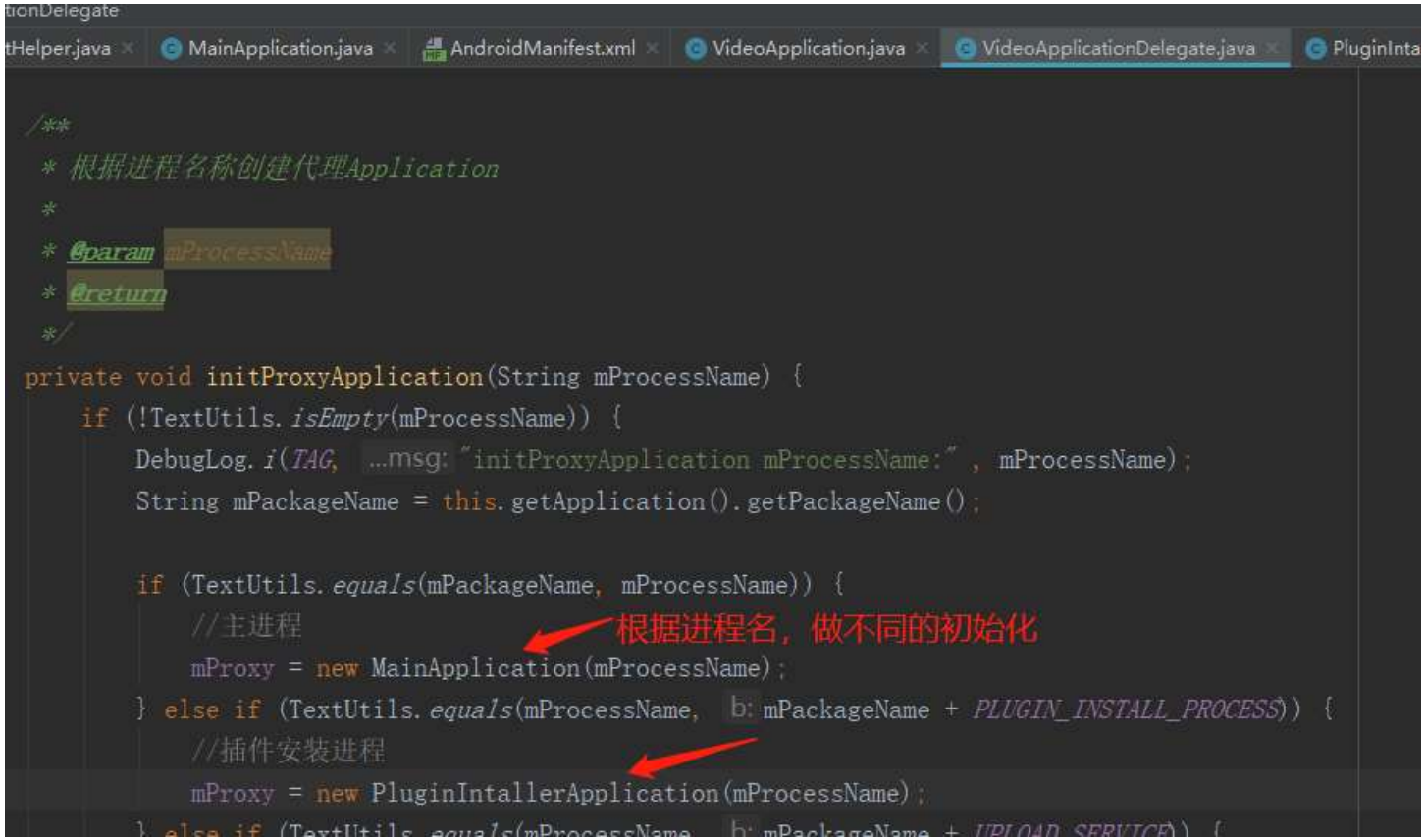
基线app开启独立进程的模块，有不少，常见的如下：

```
6433 u0_a673  10 -10   7% S   226 2265028K 237688K  ta com.qiyi.video:plugin1
5191 u0_a673  20   0   1% S   295 2488116K 369880K  fg com.qiyi.video
5556 u0_a673  20   0   0% S   120 1977280K  97392K  fg com.qiyi.video:downloader
```

插件运行

关于多进程

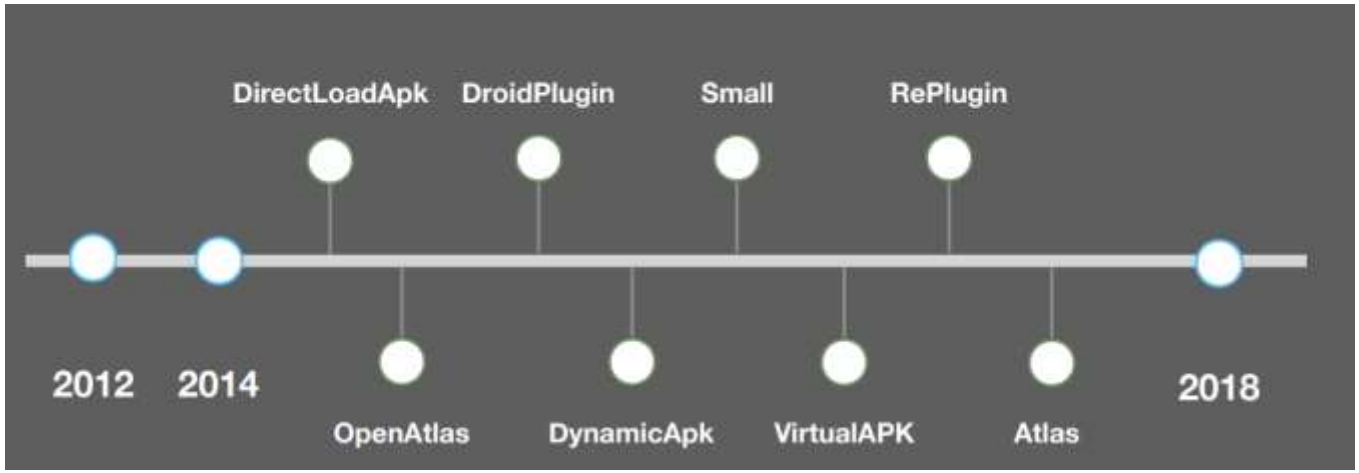
多进程的好处是，增加内存，且隔离业务（A进程崩溃，不会影响B进程）。
但多进程，Application 被多次创建，需要在创建时，根据进程名，做不同的处理。
基线app是这样处理的：



后话

基线插件库地址：http://gitlab.qiyi.domain/mobile-android/Neptune

各大公司，也有推出插件方案，原理不尽相同




iQIYI
爱奇艺
悦 享 品 质