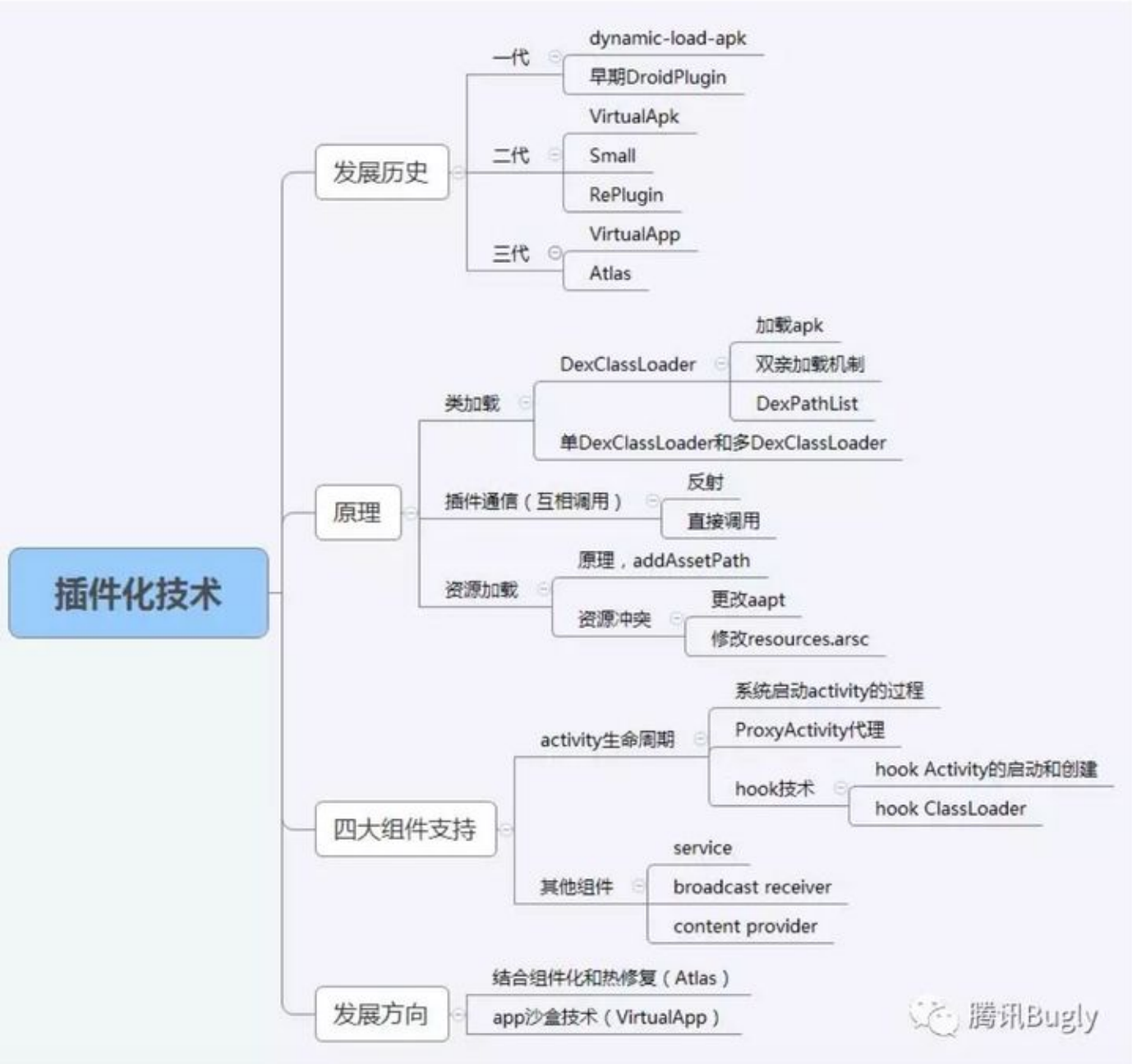
深入理解Android插件化技术

阿里云云...
已认证的官方帐号

117 人赞同了该文章

插件化技术可以说是Android高级工程师所必须具备的技能之一，从2012年插件化概念的提出（Android版本），到2016年插件化的百花争艳，可以说，插件化技术引领着Android技术的进步。

插件化提要



可以说，插件化技术涉及得非常广泛，其中最核心的就是Android的类加载机制和反射机制，相关原理请大家自行百度。

插件化发展历史

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可以理解为插件。支持插件化的app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。想要实现插件化，主要是解决下面三个问题：

- 插件中代码的加载和与主工程的互相调用
- 插件中资源的加载和与主工程的互相访问
- 四大组件生命周期的管理

下面是比较出名的几个开源的插件化框架，按照出现的时间排序。研究它们的实现原理，可以大致看出插件化技术的发展，根据实现原理可以将这几个框架划分成了三代。

第一代	dynamic-load-apk	早期DroidPlugin	
第二代	VirtualAPK	Small	RePlugin
第三代	VirtualApp	Atlas	腾讯Bugly

第一代：dynamic-load-apk最早使用ProxyActivity这种静态代理技术，由ProxyActivity去控制插件中PluginActivity的生命周期。该种方式缺点明显，插件中的activity必须继承PluginActivity，开发时要小心处理context。而DroidPlugin通过Hook系统服务的方式启动插件中的Activity，使得开发插件的过程和开发普通的app没有什么区别，但是由于hook过多系统服务，异常复杂且不够稳定。

第二代：为了同时达到插件开发的低侵入性（像开发普通app一样开发插件）和框架的稳定性，在实现原理上都是趋近于选择尽量少的hook，并通过在manifest中预埋一些组件实现对四大组件的插件化。另外各个框架根据其设计思想都做了不同程度的扩展，其中Small更是做成了一个跨平台，组件化的开发框架。

第三代：VirtualApp比较厉害，能够完全模拟app的运行环境，能够实现app的免安装运行和双开技术。Atlas是阿里今年开源出来的一个结合组件化和热修复技术的一个app基础框架，其广泛的应用与阿里系的各个app，其号称是一个容器化框架。

插件化原理

类加载

Android中常用的有两种类加载器，DexClassLoader和PathClassLoader，它们都继承于BaseDexClassLoader。相关源码如下：

```
// DexClassLoaderpublic class DexClassLoader extends BaseDexClassLoader {    public DexClassLoader(String dexPath,
            String libraryPath, ClassLoader parent) {        super(dexPath, new File(optimizedDirectory), libraryP
    }
}
// PathClassLoader

public class PathClassLoader extends BaseDexClassLoader {    public PathClassLoader(String dexPath, ClassLoader pa
   super(dexPath, null, null, parent);
    }

 public PathClassLoader(String dexPath, String libraryPath,
            ClassLoader parent) {
    super(dexPath, null, libraryPath, parent);
    }
}
```

区别在于调用父类构造器时，DexClassLoader多传了一个optimizedDirectory参数，这个目录必须是内部存储路径，用来缓存系统创建的Dex文件。而PathClassLoader该参数为null，只能加载内部存储目录的Dex文件。所以我们可以用DexClassLoader去加载外部的apk，用法如下：

```
//第一个参数为apk的文件目录
//第二个参数为内部存储目录
//第三个为库文件的存储目录
//第四个参数为父加载器
new DexClassLoader(apk.getAbsolutePath(), dexOutputPath, libsDir.getAbsolutePath(), parent)
```

其实，关于类加载更详细的内容，笔者也深入剖析过，可以查看下面的链接：类加载机制详解

双亲委托机制

ClassLoader调用loadClass方法加载类，代码如下：

```
protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
       //首先从已经加载的类中查找
        Class<?> clazz = findLoadedClass(className);
    if (clazz == null) {
            ClassNotFoundException suppressed = null;
           try {
                //如果没有加载过，先调用父加载器的loadClass
               clazz = parent.loadClass(className, false);
            } catch (ClassNotFoundException e) {
               suppressed = e;
            }
        if (clazz == null) {
                try {

                  //父加载器都没有加载，则尝试加载
                    clazz = findClass(className);
                } catch (ClassNotFoundException e) {
                    e.addSuppressed(suppressed);
                     throw e;
                }
            }
        }
            return clazz;
    }
```

可以看出ClassLoader加载类时，先查看自身是否已经加载过该类，如果没有加载过会首先让父加载器去加载，如果父加载器无法加载该类时才会调用自身的findClass方法加载，该机制很大程度上避免了类的重复加载。

DexPathList

这里要重点说一下DexClassLoader的DexPathList。DexClassLoader重载了findClass方法，在加载类时会调用其内部的DexPathList去加载。DexPathList是在构造DexClassLoader时生成的，其内部包含了DexFile。如下图所示：

DexPathList的loadClass会去遍历DexFile直到找到需要加载的类。

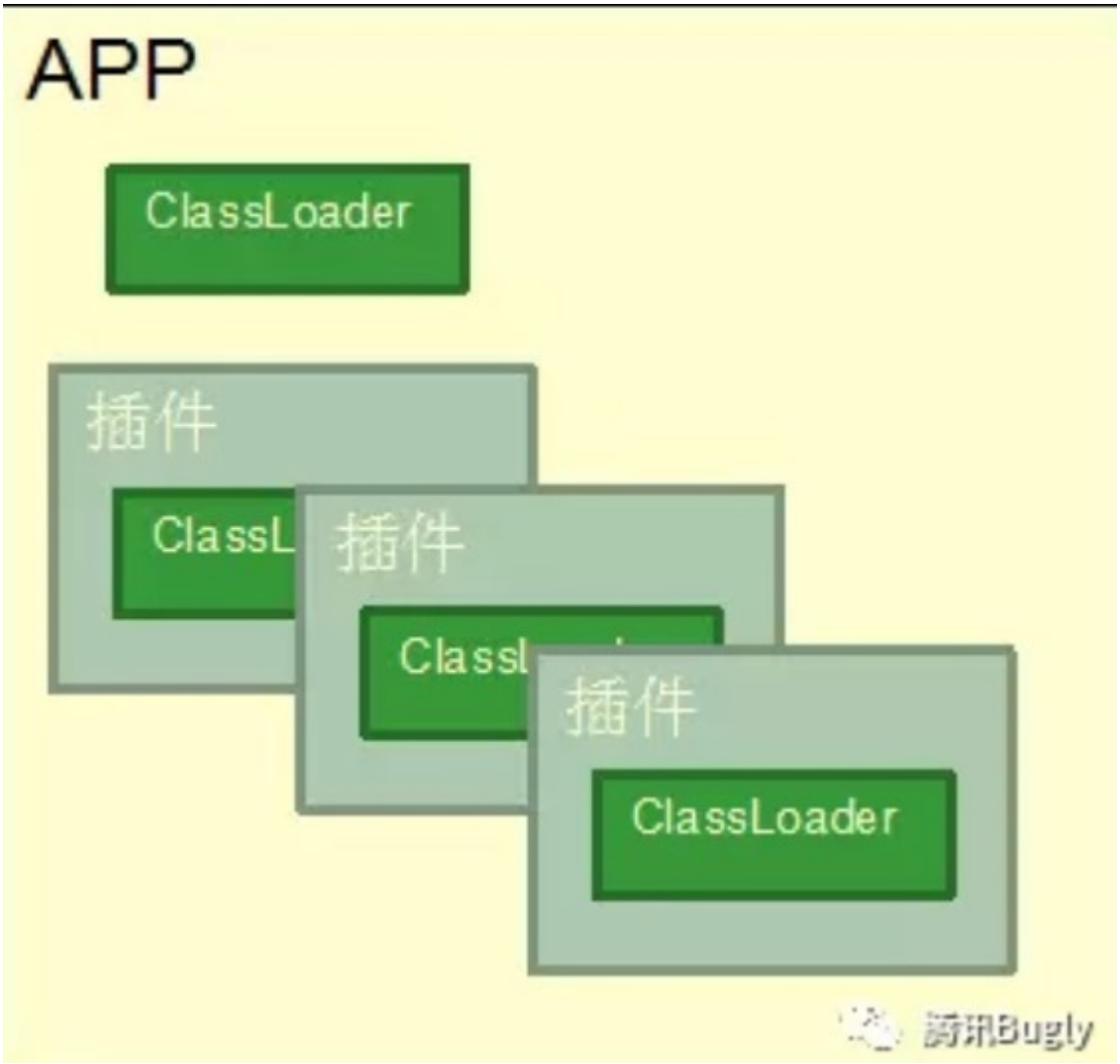
```
public Class findClass(String name, List<Throwable> suppressed) {
       //循环dexElements，调用DexFile.loadClassBinaryName加载class
        for (Element element : dexElements) {
            DexFile dex = element.dexFile;
        if (dex != null) {
                Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
                     if (clazz != null) {
                       return clazz;
                }
            }
        }
      if (dexElementsSuppressedExceptions != null) {
            suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
        }
         return null;
    }
```

腾讯的qq空间热修复技术正是利用了DexClassLoader的加载机制，将需要替换的类添加到dexElements的前面，这样系统会使用先找到的修复过的类。

单DexClassLoader与多DexClassLoader

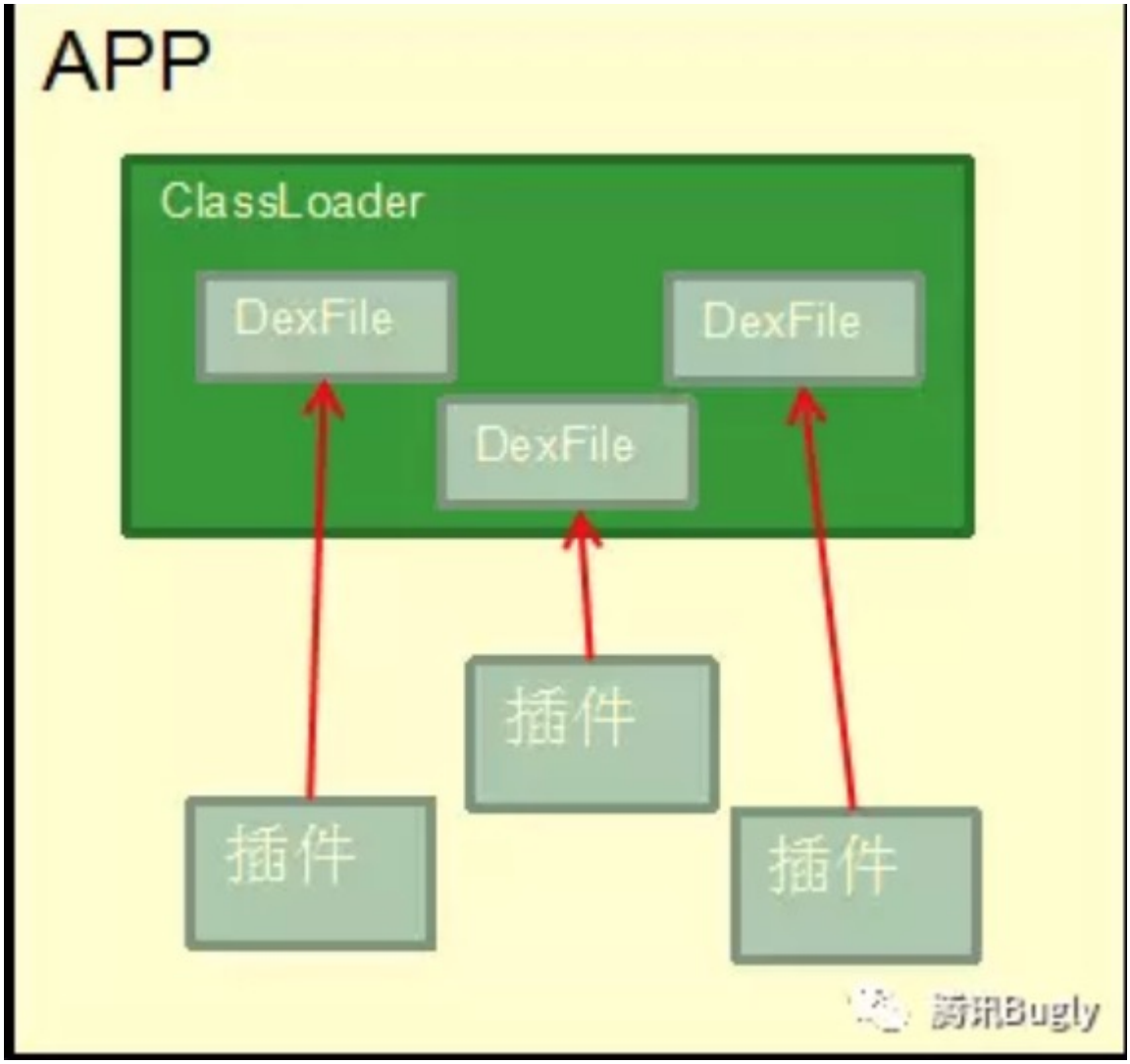
通过给插件apk生成相应的DexClassLoader便可以访问其中的类，这边又有两种处理方式，有单DexClassLoader和多DexClassLoader两种结构。

对于多DexClassLoader结构来说，可以用下面的模型来标识。



对于每个插件都会生成一个DexClassLoader，当加载该插件中的类时需要通过对应DexClassLoader加载。这样不同插件的类是隔离的，当不同插件引用了同一个类库的不同版本时，不会出问题，RePlugin采用的就是此方案。

对于单DexClassLoader来说，其模型如下：



将插件的DexClassLoader中的pathList合并到主工程的DexClassLoader中。这样做的好处时，可以在不同的插件以及主工程间直接互相调用类和方法，并且可以将不同插件的公共模块抽出来放在一个common插件中直接供其他插件使用。Small采用的是这种方式。

插件和主工程的互相调用涉及到以下两个问题：

插件调用主工程

在构造插件的ClassLoader时会传入主工程的ClassLoader作为父加载器，所以插件是可以直接可以通过类名引用主工程的类。

主工程调用插件

- 若使用多ClassLoader机制，主工程引用插件中类需要先通过插件的ClassLoader加载该类再通过反射调用其方法。插件化框架一般会通过统一的入口去管理对各个插件中类的访问，并且做一定的限制。
- 若使用单ClassLoader机制，主工程则可以直接通过类名去访问插件中的类。该方式有个弊病，

若两个不同的插件工程引用了一个库的不同版本，则程序可能会出错，所以要通过一些规范去避免该情况发生。

关于双亲委托更详细的资料，大家也可以访问我博客之前的介绍：classloader双亲委托模式

资源加载

Android系统通过Resource对象加载资源，下面代码展示了该对象的生成过程。

```
//创建AssetManager对象
AssetManager assets = new AssetManager();
 //将apk路径添加到AssetManager中
  if (assets.addAssetPath(resDir) == 0){
    return null;
}
 //创建Resource对象

r = new Resources(assets, metrics, getConfiguration(), compInfo);
```

因此，只要将插件apk的路径加入到AssetManager中，便能够实现对插件资源的访问。

具体实现时，由于AssetManager并不是一个public的类，需要通过反射去创建，并且部分Rom对创建的Resource类进行了修改，所以需要考虑不同Rom的兼容性。

资源路径的处理

和代码加载相似，插件和主工程的资源关系也有两种处理方式：

- 合并式：addAssetPath时加入所有插件和主工程的路径；
- 独立式：各个插件只添加自己apk路径

方式	优点	缺点
合并式	插件和主工程能够直接相互访问资源	会引入资源冲突
独立式	资源隔离，不存在资源冲突	资源共享比较麻烦

合并式由于AssetManager中加入了所有插件和主工程的路径，因此生成的Resource可以同时访问插件和主工程的资源。但是由于主工程和各个插件都是独立编译的，生成的资源id会存在相同的情况，在访问时会产生资源冲突。

独立式时，各个插件的资源是互相隔离的，不过如果想要实现资源的共享，必须拿到对应的Resource对象。

Context的处理

通常我们通过Context对象访问资源，光创建出Resource对象还不够，因此还需要一些额外的工作。 对资源访问的不同实现方式也需要不同的额外工作。以VirtualAPK的处理方式为例。

第一步：创建Resource

```
if (Constants.COMBINE_RESOURCES) {
    //插件和主工程资源合并时需要hook住主工程的资源
    Resources resources = ResourcesManager.createResources(context, apk.getAbsolutePath());
    ResourcesManager.hookResources(context, resources);
      return resources;
} else {
      //插件资源独立，该resource只能访问插件自己的资源
    Resources hostResources = context.getResources();
    AssetManager assetManager = createAssetManager(context, apk);
        return new Resources(assetManager, hostResources.getDisplayMetrics(), hostResources.getConfiguration());
}
```

第二步：hook主工程的Resource

对于合并式的资源访问方式，需要替换主工程的Resource，下面是具体替换的代码。

```
public static void hookResources(Context base, Resources resources) {
   try {
            ReflectUtil.setField(base.getClass(), base, "mResources", resources);
            Object loadedApk = ReflectUtil.getPackageInfo(base);
            ReflectUtil.setField(loadedApk.getClass(), loadedApk, "mResources", resources);

            Object activityThread = ReflectUtil.getActivityThread(base);
            Object resManager = ReflectUtil.getField(activityThread.getClass(), activityThread, "mResourcesManager
     if (Build.VERSION.SDK_INT < 24) {
                Map<Object, WeakReference<Resources>> map = (Map<Object, WeakReference<Resources>>) ReflectUtil.ge
                Object key = map.keySet().iterator().next();
                map.put(key, new WeakReference<>(resources));
            } else {                // still hook Android N Resources, even though it's unnecessary, then nobody w
                Map map = (Map) ReflectUtil.getFieldNoException(resManager.getClass(), resManager, "mResourceImpls
                Object key = map.keySet().iterator().next();
                Object resourcesImpl = ReflectUtil.getFieldNoException(Resources.class, resources, "mResourcesImpl
                map.put(key, new WeakReference<>(resourcesImpl));
            }
    } catch (Exception e) {
        e.printStackTrace();
```

注意下上述代码hook了几个地方，包括以下几个hook点：

替换了主工程context中LoadedApk的mResource对象。

将新的Resource添加到主工程ActivityThread的mResourceManager中，并且根据Android版本做了不同处理。

第三步：关联resource和Activity

```
Activity activity = mBase.newActivity(plugin.getClassLoader(), targetClassName, intent);
activity.setIntent(intent);
//设置Activity的mResources属性，Activity中访问资源时都通过mResources

ReflectUtil.setField(ContextThemeWrapper.class, activity, "mResources", plugin.getResources());
```

上述代码是在Activity创建时被调用的（后面会介绍如何hook Activity的创建过程），在activity被构造出来后，需要替换其中的mResources为插件的Resource。由于独立式时主工程的Resource不能访问插件的资源，所以如果不做替换，会产生资源访问错误。

做完以上工作后，则可以在插件的Activity中放心的使用setContentView，inflater等方法加载布局了。

解决资源冲突

合并式的资源处理方式，会引入资源冲突，原因在于不同插件中的资源id可能相同，所以解决方法就是使得不同的插件资源拥有不同的资源id。

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。如下表所示：

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

所以思路是修改资源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。具体实现方式有两种：

- 修改aapt源码，编译期修改PP段。
- 修改resources.arsc文件，该文件列出了资源id到具体资源路径的映射。

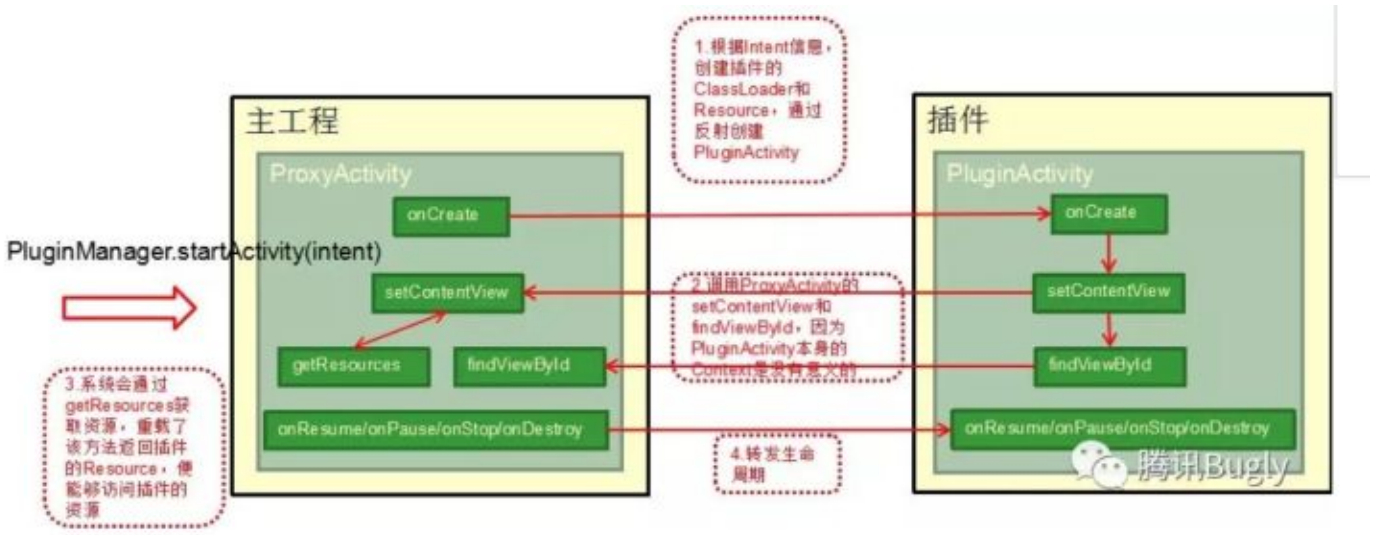
四大组件支持

Android开发中有一些特殊的类，是由系统创建的，并且由系统管理生命周期。如常用的四大组件，Activity，Service，BroadcastReceiver和ContentProvider。 仅仅构造出这些类的实例是没用的，还需要管理组件的生命周期。其中以Activity最为复杂，不同框架采用的方法也不尽相同。下面以Activity为例详细介绍插件化如何支持组件生命周期的管理。 大致分为两种方式：

- ProxyActivity代理
- 预埋StubActivity，hook系统启动Activity的过程

ProxyActivity代理

ProxyActivity代理的方式最早是由dynamic-load-apk提出的，其思想很简单，在主工程中放一个ProxyActivy，启动插件中的Activity时会先启动ProxyActivity，在ProxyActivity中创建插件Activity，并同步生命周期。下图展示了启动插件Activity的过程。



具体的过程如下：

1. 首先需要通过统一的入口（如图中的PluginManager）启动插件Activity，其内部会将启动的插件Activity信息保存下来，并将intent替换为启动ProxyActivity的intent。
2. ProxyActivity根据插件的信息拿到该插件的ClassLoader和Resource，通过反射创建PluginActivity并调用其onCreate方法。
3. PluginActivty调用的setContentView被重写了，会去调用ProxyActivty的setContentView。由于ProxyActivity重写了getResource返回的是插件的Resource，所以setContentView能够访问到插件中的资源。同样findViewById也是调用ProxyActivity的。
4. ProxyActivity中的其他生命周期回调函数中调用相应PluginActivity的生命周期。

理解ProxyActivity代理方式主要注意两点：

- ProxyActivity中需要重写getResouces，getAssets，getClassLoader方法返回插件的相应对象。生命周期函数以及和用户交互相关函数，如onResume，onStop，onBackPressedon，KeyUponWindow，FocusChanged等需要转发给插件。
- PluginActivity中所有调用context的相关的方法，如setContentView，getLayoutInflater，getSystemService等都需要调用ProxyActivity的相应方法。

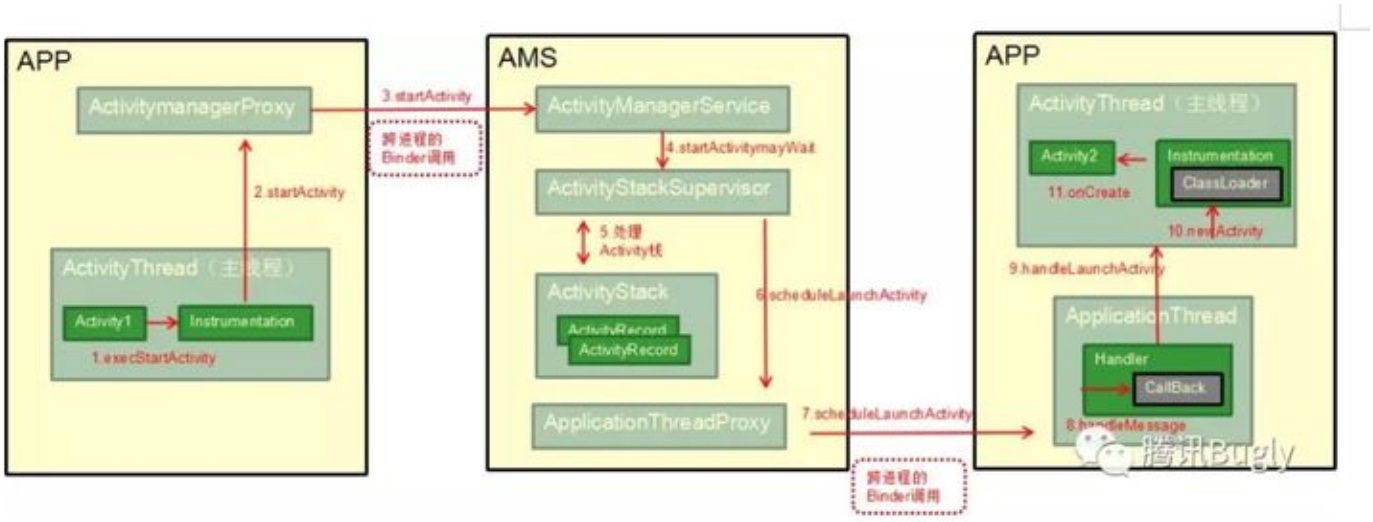
缺点

- 插件中的Activity必须继承PluginActivity，开发侵入性强。
- 如果想支持Activity的singleTask，singleInstance等launchMode时，需要自己管理Activity栈，实现起来很繁琐。
- 插件中需要小心处理Context，容易出错。
- 如果想把之前的模块改造成插件需要很多额外的工作。

该方式虽然能够很好的实现启动插件Activity的目的，但是由于开发式侵入性很强，dynamic-load-apk之后的插件化方案很少继续使用该方式，而是通过hook系统启动Activity的过程，让启动插件中的Activity像启动主工程的Activity一样简单。

hook方式

在介绍hook方式之前，先用一张图简要的介绍下系统是如何启动一个Activity的。



上图列出的是启动一个Activity的主要过程，具体步骤如下：

1. Activity1调用startActivity，实际会调用Instrumentation类的execStartActivity方法，Instrumentation是系统用来监控Activity运行的一个类，Activity的整个生命周期都有它的影子。

2. 通过跨进程的binder调用，进入到ActivityManagerService中，其内部会处理Activity栈。之后又通过跨进程调用进入到Activity2所在的进程中。
3. ApplicationThread是一个binder对象，其运行在binder线程池中，内部包含一个H类，该类继承于类Handler。ApplicationThread将启动Activity2的信息通过H对象发送给主线程。
4. 主线程拿到Activity2的信息后，调用Instrumentation类的newActivity方法，其内通过ClassLoader创建Activity2实例。

下面介绍如何通过hook的方式启动插件中的Activity，需要解决以下两个问题：

- 插件中的Activity没有在AndroidManifest中注册，如何绕过检测。
- 如何构造Activity实例，同步生命周期

解决方法有很多种，以VirtualAPK为例，核心思路如下：

1. 先在Manifest中预埋StubActivity，启动时hook上图第1步，将Intent替换成StubActivity。
2. hook第10步，通过插件的ClassLoader反射创建插件Activity
3. 之后Activity的所有生命周期回调都会通知给插件Activity

替换系统Instrumentation

VirtualAPK在初始化时会调用hookInstrumentationAndHandler，该方法hook了系统的Instrumentaiton类，由上文可知该类和Activity的启动息息相关。

```
private void hookInstrumentationAndHandler() {
    try {
          //获取Instrumentation对象
        Instrumentation baseInstrumentation = ReflectUtil.getInstrumentation(this.mContext);
              //构造自定义的VAInstrumentation
        final VAInstrumentation instrumentation = new VAInstrumentation(this, baseInstrumentation);
                    //设置ActivityThread的mInstrumentation和mCallBack
        Object activityThread = ReflectUtil.getActivityThread(this.mContext);
        ReflectUtil.setInstrumentation(activityThread, instrumentation);
        ReflectUtil.setHandlerCallback(this.mContext, instrumentation);
         this.mInstrumentation = instrumentation;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

该段代码将主线程中的Instrumentation对象替换成了自定义的VAInstrumentation类。在启动和创建插件activity时，该类都会偷偷做一些手脚。

hook activity启动过程

VAInstrumentation类重写了execStartActivity方法，相关代码如下：

```
public ActivityResult execStartActivity(
    //省略了无关参数
    Intent intent) {
//转换隐式intent
    mPluginManager.getComponentsHandler().transformIntentToExplicitAsNeeded(intent);
   if (intent.getComponent() != null) {
     //替换intent中启动Activity为StubActivity
       this.mPluginManager.getComponentsHandler().markIntentIfNeeded(intent);
    }

    //调用父类启动Activity的方法}
public void markIntentIfNeeded(Intent intent) {
   if (intent.getComponent() == null) {
        return;
    }

    String targetPackageName = intent.getComponent().getPackageName();
    String targetClassName = intent.getComponent().getClassName();    // search map and return specific launchmode
    if (!targetPackageName.equals(mContext.getPackageName()) && mPluginManager.getLoadedPlugin(targetPackageName)
        intent.putExtra(Constants.KEY_IS_PLUGIN, true);
        intent.putExtra(Constants.KEY_TARGET_PACKAGE, targetPackageName);
        intent.putExtra(Constants.KEY_TARGET_ACTIVITY, targetClassName);
        dispatchStubActivity(intent);
    }
}
```

execStartActivity中会先去处理隐式intent，如果该隐式intent匹配到了插件中的Activity，将其转换成显式。之后通过markIntentIfNeeded将待启动的的插件Activity替换成了预先在AndroidManifest中占坑的StubActivity，并将插件Activity的信息保存到该intent中。其中有个dispatchStubActivity函数，会根据Activity的launchMode选择具体启动哪个StubActivity。VirtualAPK为了支持Activity的launchMode在主工程的AndroidManifest中对于每种启动模式的Activity都预埋了多个坑位。

hook Activity的创建过程

上一步欺骗了系统，让系统以为自己启动的是一个正常的Activity。当来到图 3.2的第10步时，再将插件的Activity换回来。此时调用的是VAInstrumentation类的newActivity方法。

```
@Override
public Activity newActivity(ClassLoader cl, String className, Intent intent){
    try {
        cl.loadClass(className);
    } catch (ClassNotFoundException e) {
        //通过LoadedPlugin可以获取插件的ClassLoader和Resource
        LoadedPlugin plugin = this.mPluginManager.getLoadedPlugin(intent);
                //获取插件的主Activity
        String targetClassName = PluginUtil.getTargetActivity(intent);
                if (targetClassName != null) {
                   //传入插件的ClassLoader构造插件Activity
            Activity activity = mBase.newActivity(plugin.getClassLoader(), targetClassName, intent);
            activity.setIntent(intent);
                    //设置插件的Resource，从而可以支持插件中资源的访问
            try {
                ReflectUtil.setField(ContextThemeWrapper.class, activity, "mResources", plugin.getResources());
            } catch (Exception ignored) {
                                // ignored.
            }
          return activity;
        }
    }    return mBase.newActivity(cl, className, intent);
}
```

由于AndroidManifest中预埋的StubActivity并没有具体的实现类，所以此时会发生ClassNotFoundException。之后在处理异常时取出插件Activity的信息，通过插件的ClassLoader反射构造插件的Activity。

其他操作

插件Activity构造出来后，为了能够保证其正常运行还要做些额外的工作。

```
@Override
public void callActivityOnCreate(Activity activity, Bundle icicle) {
   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, icicle);
}
```

这段代码主要是将Activity中的Resource，Context等对象替换成了插件的相应对象，保证插件Activity在调用涉及到Context的方法时能够正确运行。

经过上述步骤后，便实现了插件Activity的启动，并且该插件Activity中并不需要什么额外的处理，和常规的Activity一样。那问题来了，之后的onResume，onStop等生命周期怎么办呢？答案是所有和Activity相关的生命周期函数，系统都会调用插件中的Activity。原因在于AMS在处理Activity时，通过一个token表示具体Activity对象，而这个token正是和启动Activity时创建的对象对应的，而这个Activity被我们替换成了插件中的Activity，所以之后AMS的所有调用都会传给插件中的Activity。

其他组件

四大组件中Activity的支持是最复杂的，其他组件的实现原理要简单很多，简要概括如下：

- **Service**：Service和Activity的差别在于，Activity的生命周期是由用户交互决定的，而Service的生命周期是我们通过代码主动调用的，且Service实例和manifest中注册的是一一对应的。实现Service插件化的思路是通过在manifest中预埋StubService，hook系统startService等调用替换启动的Service，之后在StubService中创建插件Service，并手动管理其生命周期。
- **BroadCastReceiver**：解析插件的manifest，将静态注册的广播转为动态注册。
- **ContentProvider**：类似于Service的方式，对插件ContentProvider的所有调用都会通过一个在manifest中占坑的ContentProvider分发。

小结

VirtualAPK通过替换了系统的Instrumentation，hook了Activity的启动和创建，省去了手动管理插件Activity生命周期的繁琐，让插件Activity像正常的Activity一样被系统管理，并且插件Activity在开发时和常规一样，即能独立运行又能作为插件被主工程调用。

其他插件框架在处理Activity时思想大都差不多，无非是这两种方式之一或者两者的结合。在hook时，不同的框架可能会选择不同的hook点。如360的RePlugin框架选择hook了系统的ClassLoader，即图3.2中构造Activity2的ClassLoader，在判断出待启动的Activity是插件中的时，会调用插件的ClassLoader构造相应对象。另外RePlugin为了系统稳定性，选择了尽量少的hook，因此它并没有选择hook系统的startActivity方法来替换intent，而是通过重写Activity的startActivity，因此其插件Activity是需要继承一个类似PluginActivity的基类的。不过RePlugin提供了一个Gradle插件将插件中的Activity的基类换成了PluginActivity，用户在开发插件Activity时也是没有感知的。