Shadow特点：零反射、框架自身动态化；

对于Shadow来说，插件业务生成的插件包，不仅仅只有一个插件业务的apk。往往是一个由如下4部分组成：

1.manger.apk 专门用于联通宿主apk。当宿主apk启动插件的时候，需要启动manager模块，加载 loader.apk,runtime.apk,以及业务插件apk。

2.loader.apk 专门用于映射坑位Activity/Broadcast/Service/ContentProvider 与 业务插件中四大组件对应的类名

3.runtime.apk 用于申明四大组件的类。注意runtime.apk中的AndroidManifest 中不需要声明。这些坑位需要直接声明到宿主中

4.插件业务生成的apk 业务的实现

插件化需要跨越的三大问题：

越过AndroidManifest的校验；

查找插件的Class；

加载插件资源；

三个问题的解决方案：

对于第一个问题：因为是插件库把Activity转化成一个不继承任何类的PluginActivity，并依赖坑位Activity的行为。因此只需要声明坑位的Activity，只是这个坑位的Activity需要做特殊处理，可以注入PluginActivity，并调用公有回调

对于第二个问题：因为不是通过内置的ClassLoader查找Class。而是通过自己生成的ClassLoader 查找Class

对于第三个问题：Context都是自己创建的了，为什么不可以自己在自己的Context做好资源管理呢

Shadow 使用

为了避免插件库影响原来的业务逻辑，一般会常见一个壳模块。这个模块作为主模块，被业务工程依赖。

在工程根目录的build.gradle 新增classpath：com.tencent.shadow.core:gradle-plugin 插件是将所有的四大组件，替换成Shadow的对应的代理类。而这些代理类会把工作交给runtime中真正的Activity组件实现。

创建插件工程，在插件工程里设置Task生成对应的插件包，里面包含了runtime的apk + loade的apk + 插件业务的apk。

创建runtime工程，在runtime工程里面创建站位的Activity，然后将它们注册到宿主的AndroidManifest文件中，且设置为跟MainPluginProcessService一样的进程android:process=":plugin"。

创建loader工程，并创建CoreLoaderFactoryImpl，它的类名必须固定，因为shadow必须通过manager反射来加载它。loader会将写好的代理Activity跟插件业务的Activity进行一一对应。

创建manager工程，它是宿主跟插件交互的桥梁，里面进行了有关插件包的桥接工作。

shadow源码分析

宿主想要启动插件中的某个Activity需要经历如下几个步骤：

1.通过 new DynamicPluginManager(fixedPathPmUpdater); 声明DynamicPluginManager对象

1. 调用DynamicPluginManager的enter方法启动插件中manager.apk的相同id的行为。
2. 将manager.apk抽象成InstalledApk类，进而封装成ApkClassLoader用于查找manager.apk的类、ChangeApkContextWrapper用于获取资源和获取ApkClassLoader。
3. 最后通过ApkClassLoader 查找这个apk中的com.tencent.shadow.dynamic.impl.ManagerFactoryImpl 类并实例化后，调用buildManager 生成PluginManagerImpl对象。因此在manager工程中需要 写一个类名ManagerFactoryImpl，作为manager.apk逻辑的入口。因此此时在宿主中返回的PluginManagerImpl 实际上就是自己编写的manager工程中返回的ManagerFactory实现类。
4. 进入manager.apk的逻辑。
5. manager.apk会尝试启动我们在插件包中定义的插件业务。而这个插件包存在着三个apk，loader.apk,runtime.apk,$插件业务.apk.因此可以说manager.apk管理了当前插件的加载逻辑。
6. 安装压缩包的runtime和loader的apk转化成dex并复制到目标文件夹；安装记录so的位置并复制到目标文件夹；把业务的apk转化成dex并复制到目标目录。
7. 1.启动一个Service服务，并等待服务10秒。在这10秒中会创建一个mPluginLoader对象；2.loadRunTime 通过mPluginLoader 加载runtime.apk的数据；3.loadPluginLoader 通过mPluginLoader 加载loader.apk的数据；4.mPluginLoader.loadPlugin 加载插件；5.mPluginLoader.callApplicationOnCreate 调用插件的Application的onCreate方法；6.convertActivityIntent 检查坑位和插件Activity的映射关系；7.mPluginLoader.startActivityInPluginProcess 启动Activity在插件进程；
8. 通过宿主的Context的bindService方法，启动一个固定类名的Service。并监听onServiceConnected方法，回调抽象方法onPluginServiceConnected。这个Service就是我们在宿主定义的Service。我们需要在manager工程中返回这个类的固定包名。之后所有的操作都将由manager转交给远端进程的Service进行处理。所以所有的占坑组件都需要跟Service在同一个进程才能进行操作。
9. 在Service进程中把runtime.apk 的路径生成一个RuntimeClassLoader，并通过反射，设置为为当前进程ClassLoader父ClassLoader。由于ClassLoader的双亲委托机制，当前进程要查找runtime中设置的坑位时候，通过委托机制会优先交给RuntimeClassLoader进行查找。这样宿主的AndroidManifest.xml注册了坑位，且在插件进程的ClassLoader 也注入了Class。这样就保证了在插件进程，看起来就像AndroidManifest和Class 都在同一模块一样。这么做的好处是显而易见的，可以拆开实现，这样就能在插件包中自定义坑位逻辑，而不需要更新宿主。
10. 通过loader.apk生成一个ApkClassLoader，并获取一个固定的类名LoaderFactoryImpl并实例化，最后调用buildLoader方法。
11. 先反射loader.apk中的CoreLoaderFactoryImpl对象，并调用build方法。此时在DynamicPluginLoader中的mPluginLoader,实际上就是指我们自定义的SamplePluginLoader。注意这个对象是继承于ShadowPluginLoader.
12. 在Manager进程中做了两个步骤进行加载插件：1.获取PluginProcessService 的 PluginLoaderImpl 对象；2.调用PluginLoaderImpl的loadPlugin。
13. 转换插件apk的路径为一个PluginClassLoader，通过packageManager.getPackageArchiveInfo读取该apk中所有的四大组件相关的信息。注意在这个过程中会校验宿主的Context的packageName是否和插件包的xml中的包名一直。只有一致才会继续解析出插件信息。因为在Android的资源管理中，有多重缓存。其次就是动态权限相关的，在动态权限中所有的信息都保存在一个xml文件中。这个xml权限将以packageName+permissionName等为key记录相关信息。在Shadow看来，一来插件并不是真正的安装在Android系统中不符合规范，而来实际上插件对于宿主来说也是一种延伸。因此需要保证宿主的packageName和插件记录在AndroidManifest中的packageName一致。
14. 更新宿主的ResourceImpl。调用到ResourceManager的getOrCreateResources方法中。那么此时会在ResourceManager中为当前的apk资源创建一个全新的ResourceImpl。创建一个ShadowAppComponentFactory,用于对于xml中的appComponentFactory。根据以上plugin信息实例化一个插件的Application，但是插件的Application的已经通过gradle插件转化为ShadowApplication了。对坑位和插件的Activity进行映射。对runtime中的坑位和实际插件业务Activity之间的映射关系。最后把这些信息都保存到PluginPartInfoManager。
15. 接下来在Manager进程就会调用PluginLoader.callApplicationOnCreate.而这个PluginLoader对象实际上是一个Binder对象，最后会调用到DynamicPluginLoader中。而这个方法最后又会调用到ShadowPluginLoader中。
16. 依次调用业务插件Application的attachBaseContext，CP的attachInfo,Application的onCreate。
17. Shadow，在启动插件Activity的时候，直接写入类名到Intent中。接着会在DynamicPluginLoader中检查之前通过addPluginInfo 完成的映射关系。把插件的Activity对应的坑位变成目标启动的Activity。而原来插件的Activity将会存到Intent中。等待坑位的处理。此时我们假设是映射的是一个PluginDefaultProxyActivity.那么PluginDefaultProxyActivity(继承了PluginContainerActivity)这个Activity声明在AndroidManifest.xml的进程刚好是plugin。就是通过这个手段达到的跨进程启动Activity的手段。
18. 实例化插件的Activity对象后。initPluginActivity 初始化插件的Activity。注意插件的Activity已经被替换成了ShadowActivity了。这个方法中为插件Activity设置了classLoader，application等核心参数。插件的Activity监听来自坑位的Window.Callback.以及处理setSoftInputMode。调用插件Activity的onCreate。

