背景

> 由于QQ的包体积越来越大，给运营推广带来很大的压力。我们在进行无用资源清除、图片格式转换、资源压缩等等一系列组合拳之后，包体积有了一定的缩减，

但是也到达了一个瓶颈。经过一系列调研之后，决定采用插件化技术，将部分模块使用插件的方式进行下发，从而有效的去缩减包体积大小。

什么是插件化

1.1 模块化和插件化

通常模块化是指一个应用有多个业务模块，每个模块有独立的module，各module之间通过路由或者接口的形式进行通信，但是最终打包的时候，所有module都会被打包到同一个apk里面。

插件化和模块化类似的地方就是，每个模块也是独立的module，不同的是插件所属的模块不会打包进宿主apk，插件模块会打包成独立的apk，然后通过网络下发的方式，让宿主动态去加载插件apk。

1.2 插件化的好处

> 1. 宿主插件互相解藕，
> 2. 减少宿主APK的包体积；
> 3. 插件可以动态下发，升级插件功能或者修复问题非常方便快捷，不依赖发版。

1.3 插件化的知识基础

1.3.1 类加载机制

所有的插件化框架，都需要去使用Classloader去加载插件里面的类文件，所以这里需要大家了解一下Classloader的类加载机制，下图是java的双亲委派加载机制：

相对于 java 的 ClassLoader，双亲委派是同样适用的，只不过类加载器有些出入，下面看一下Android中常用的几个 ClassLoader：

```
public static PluginManager getPluginManager(File apk) {
    final FixedPathPmUpdater fixedPathPmUpdater = new FixedPathPmUpdater(apk);
    File tempPm = fixedPathPmUpdater.getLatest();
    if (tempPm != null) {
        return new DynamicPluginManager(fixedPathPmUpdater);
    }
    return null;
}
```

> 在Android中，我们正常安装到手机里面的APK所包含的类都是使用 PathClassLoader 进行加载的，而 PathClassLoader 所持有的 parent 则是 BootClassLoader。

插件化技术通常都会使用 DexClassLoader 去加载存储卡中的 APK，而在构造一个 DexClassLoader 的适合，需要指定一个 parent，通常都是指定为 PathClassLoader，这样 DexClassLoader、PathClassLoader、BootClassLoader形成了一个从子到父的链，通常也是满足双亲委派机制的（有些场景会继承 DexClassLoader 去修改类加载流程，有可能会破坏双亲委派的链，这里就不做讨论了）。

1.3.2 四大组件的占坑逻辑

Android的四大组件通常都需要在清单文件声明（动态广播除外），所以这里就有个插件化所需要解决的核心问题：插件里的四大组件没有在清单文件声明，那么怎么去启动他们呢？ 各大插件化框架基本上都是使用狸猫换太子的方式来实现的，也就是所谓的占坑。

占坑可以理解为：我们先在宿主的清单文件中声明好提供给插件使用的四大组件坑位，当我们尝试去启动插件Activity 的时候，先使用坑位Activity替换要启动的插件Activity，骗过 Manifest 的校验，然后在加载 Activity 的时候，再去加载真正要启动的插件Activity，从而实现了狸猫换太子的方案。

各大厂商的占坑方案：

为什么要用Shadow

我在做技术调研的时候，主要考虑插件化技术的以下几个方面：

- 兼容性
- 稳定性
- 社区活跃度

所以对目前市面上最大的几个插件化框架做了对比VirtualAPK、Replugin、Shadow

VirtualAPK

VirtualAPK 是滴滴开源的插件化框架，它的稳定性在滴滴已经得到验证，下面我们来看一下它的优缺点。

- 优点:

- 缺点:

1.至少2年以上没有代码提交记录了，处于不维护状态。

2.目前只适配到了Android 9.0

Replugin

360出品的插件化框架，老牌插件化框架，社区也比较活跃。

- 优点

 图片

 图片

- 缺点

1.没有适配AndroidX，虽然我对它做过AndroidX的适配，但是没有上线验证过，所以整体稳定性有待考量。

2.维护较慢，也是很久没有维护了。

 图片

3.目前也是只兼容到了Android9.0版本的系统。

 图片

Replugin 目前只兼容到Android9.0。它通过Hook宿主的ClassLoader去实现偷梁换柱的插件加载逻辑，入侵度一般。占坑太重，生成过多坑位，很多是用不上的。最严重的问题就是没有适配AndroidX，虽然我对它进行了AndroidX的适配改造，但是稳定性还没有得到验证。

Shadow

腾讯出品的插件化框架，在手机QQ得到验证，稳定性值得肯定，w号称 0 Hook（有待商榷），社区还是比较活跃的。

- 优点

介绍

Shadow是一个腾讯自主研发的Android插件框架，经过线上亿级用户量检验。 Shadow不仅开源分享了插件技术的关键代码，还完整的分享了上线部署所需要的所有设计。

与市面上其他插件框架相比，Shadow主要具有以下特点：

- **复用独立安装App的源码**：插件App的源码原本就是可以正常安装运行的。
- **零反射无Hack实现插件技术**：从理论上就已经确定无需对任何系统做兼容开发，更无任何隐藏API调用，和Google限制非公开SDK接口访问的策略完全不冲突。
- **全动态插件框架**：一次性实现完美的插件框架很难，但Shadow将这些实现全部动态化起来，使插件框架的代码成为了插件的一部分。插件的迭代不再受宿主打包了旧版本插件框架所限制。
- **宿主增量极小**：得益于全动态实现，真正合入宿主程序的代码量极小（15KB，160方法数左右）。
- **Kotlin实现**：core.loader，core.transform核心代码完全用Kotlin实现，代码简洁易维护。

支持特性

- 四大组件
- Fragment（代码添加和Xml添加）
- DataBinding（无需特别支持，但已验证可正常工作）
- 跨进程使用插件Service
- 自定义Theme
- 插件访问宿主类
- So加载
- 分段加载插件（多Apk分别加载或多Apk以此依赖加载）
- 一个Activity中加载多个Apk中的View
- 等等.....

1.简介

介绍

Shadow是一个腾讯自主研发的Android插件框架，经过线上亿级用户量检验。 Shadow不仅开源分享了插件技术的关键代码，还完整的分享了上线部署所需要的所有设计。

与市面上其他插件框架相比，Shadow主要具有以下特点：

- **复用独立安装App的源码**：插件App的源码原本就是可以正常安装运行的。
- **零反射无Hack实现插件技术**：从理论上就已经确定无需对任何系统做兼容开发，更无任何隐藏API调用，和Google限制非公开SDK接口访问的策略完全不冲突。
- **全动态插件框架**：一次性实现完美的插件框架很难，但Shadow将这些实现全部动态化起来，使插件框架的代码成为了插件的一部分。插件的迭代不再受宿主打包了旧版本插件框架所限制。
- **宿主增量极小**：得益于全动态实现，真正合入宿主程序的代码量极小（15KB，160方法数左右）。
- **Kotlin实现**：core.loader，core.transform核心代码完全用Kotlin实现，代码简洁易维护。

支持特性

- 四大组件
- Fragment（代码添加和Xml添加）
- DataBinding（无需特别支持，但已验证可正常工作）
- 跨进程使用插件Service
- 自定义Theme
- 插件访问宿主类
- So加载
- 分段加载插件（多Apk分别加载或多Apk以此依赖加载）
- 一个Activity中加载多个Apk中的View
- 等等.....

2.维护非常及时，反馈也非常快

图片

- 缺点

> **由于设计的过于灵活，上手成本增加不少；发布一个插件，至少需要一个PluginManager和一个插件zip，受网络等外部环境影响概率增大；没有发布到maven上，所以暂时只能依靠源码或者自己发布maven。SDK功能不够全面，需要我们进行二次开发。**

总结

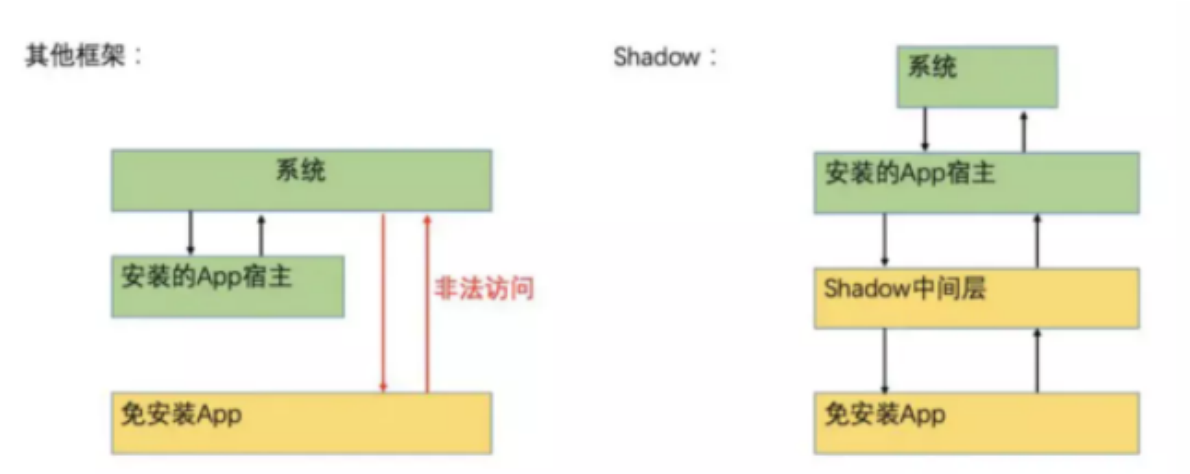
综合对上面三个插件化框架的调研，由于VirtualApk和Replugin最高只适配到了Android9.0系统，并且很久没有维护了，所以这里不做考虑。

而Shadow作为手Q开源的插件化框架，号称采用了**零反射无Hack实现插件技术**，所以对于Android版本的兼容性（尤其是高版本Android系统）应该优于VirtualApk和Replugin，并且社区较活跃，Shadow的开发人员反馈问题也非常及时。

所以我们暂定 Shadow 作为我们插件化技术方案，接下来就是对 Shadow的使用方式和原理进行剖析。

2.1 Shadow简介

图列举了Shadow对比其他插件化框架对宿主代码的增量：



下图简单列举了Shadow与其他插件化框架的区别:

图片

Shadow官方号称 **零反射无Hack实现插件技术**、**全动态插件框架**，基本上将能够动态下发的全部动态下发了，从而实现宿主代码增量极少，并且可以非常灵活的控制插件的下发。但是Shadow真的是对宿主没有任何反射Hack么？这里暂时卖个关子。

四 Shadow原理剖析

4.1 一个Shadow插件组成部分

- ### PluginManager 插件管理

插件管理模块，该模块打包成独立apk下发。宿主通过接口声明+反射和 PluginManager 形成映射关系。宿主所有的插件调用，都需要通过 PluginManager。

- **Plugin-loader**

插件的 loader 模块，该模块会作为独立的apk被打包进插件的zip包中。PluginManager 通过和 loader 的通信，实现对插件的安装、调用等操作。

- **Plugin-runtime**

插件的 runtime 模块，该模块会作为独立的 apk 被打包进插件的zip包中。该模块包含了代理Acitvity、代理ContentProvider等类。

- **Plugin-app**

插件的业务模块，该模块会作为独立的 apk 被打包进插件的zip包中。就是咱们所有的插件业务代码。

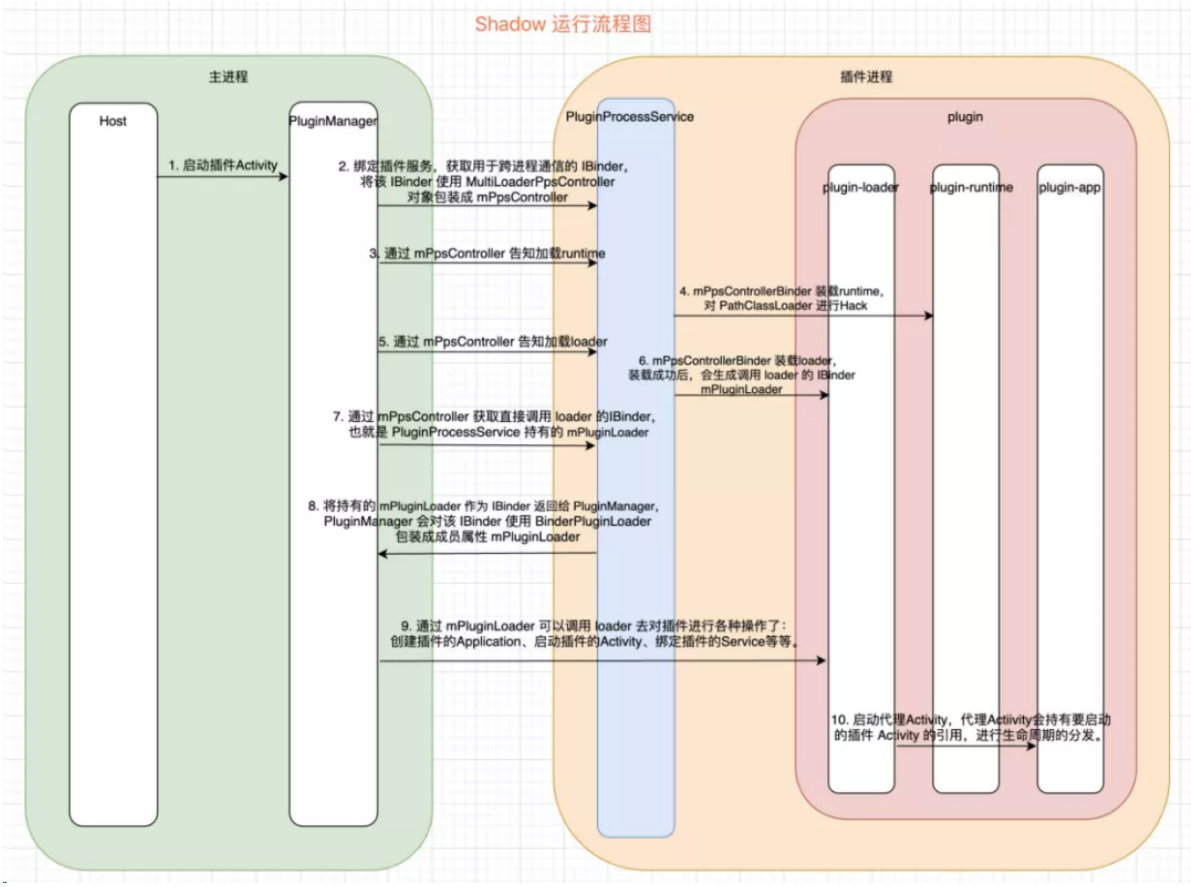
4.2 SDK**结构**

```
├── shadow
|   ├── sdk //Shadow 的 SDK 源码
|   |   ├── coding //用来做编译期的检查
|   |   |   ├── checks     //编译期检查哪些方法不符合 Shadow 规则的，提前抛出
|   |   |   ├── code-generator //编译期代码生成，包括 Activity 相关代码的生成(代理 Activity 和插件内的 Activity 生命周期联动的 Delegate)等等。
|   |   |   ├── lint //lint 检查，指定 lintChecks project(':checks')
|   |   ├── core //shadow 的一些核心库
|   |   |   ├── activity-container //插件 runtime 模块集成，里面包含代理 Activity 及相关逻辑。
|   |   |   ├── common //宿主集成，包含插件安装信息，映射宿主和 插件 loader 的 ContentProvider 代理接口。
|   |   |   ├── gradle-plugin //gradle 插件，依赖 transform 模块，用来生成打包插件的 task，生成插件的 config.json，并将它和 插件 Loader、插件 Runtime、插件 apk 打包到一个压缩包里。
|   |   |   ├── load-parameters //loader 库加载插件所需要用到的参数定义
|   |   |   ├── loader //loader，负责加载插件
|   |   |   ├── manager //pluginManager，管理插件
|   |   |   ├── runtime //runtime，插件的运行时模块，里面包含占位 Activity、Provider 等等。
|   |   |   ├── transform //编译期的 transform，在编译期将插件的 Context、Application、Activity、Service 等继承关系修改为继承 Shadow 自定义的类。
|   |   |   └── transform-kit //transform 所要用到的一些工具
|   |   └── dynamic //插件自身动态化实现，包括一些接口的抽象
|   |   |   ├── dynamic-host //宿主引用的库，声明了宿主和 PluginManager 进行通信的一些类。
|   |   |   ├── dynamic-host-multi-loader-ext //只被 dynamic-manager-multi-loader-ext 以 compileOnly 引用，暂时没看出有啥用。
|   |   |   ├── dynamic-loader //声明了 loader 的接口，作为连接 PluginManager 和 插件 Loader 的桥梁。
|   |   |   ├── dynamic-loader-impl //loader 的具体实现逻辑
|   |   |   ├── dynamic-manager //PluginManager 调用 loader 的具体实现
|   |   |   ├── dynamic-manager-multi-loader-ext //没有任何地方引用，暂时没看出有啥用。
|   |   |   ├── jar-wrapper
|   |   |   |   ├── dynamic-host-debug //将 dynamic-host 打包成 jar 包，让 PluginManger 和 loader 以 compileOnly 方式引用，从而实现和宿主形成映射关系。
|   |   |   |   ├── dynamic-host-realse //同上
```

4.3 源码剖析

4.3.1 Shadow运行流程图

下图是启动插件Activity的大概流程：



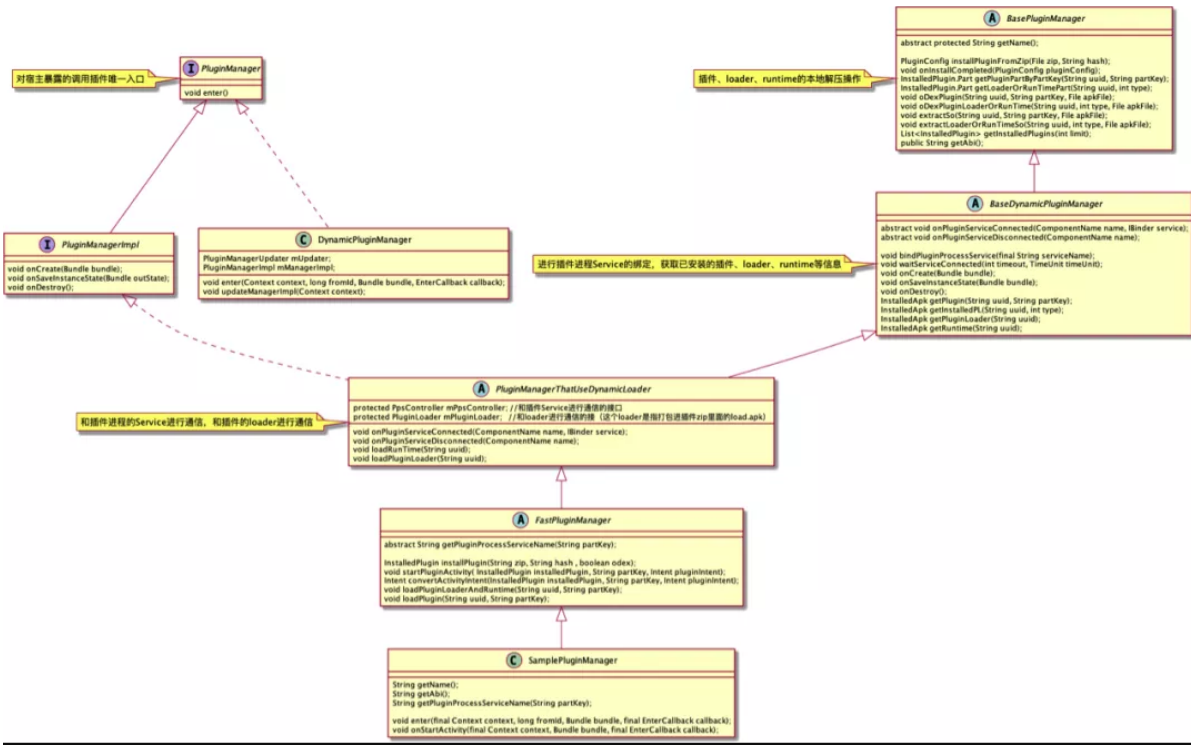
通过上面流程图，我们了解了Shadow每个模块的执行顺序：

- Host 通过 PluginManager 去调用插件；
- PluginManager 又通过 IBinder 和插件服务通信，加载 plugin-runtime、 plugin-loader；
- PluginManager 通过获取操作 plugin-loader 的 IBinder，从而实现了对插件的加载、调用等操作。

了解了 Shadow 模块的执行顺序之后，接下来我们结合源码从每个模块的具体安装、加载、调用的细节来分析一下。

4.3.2 PluginManager

PluginManager 的类图



PluginManager 的加载流程

PluginManager 的加载，在使用的时候，会创建一个 DynamicPluginManager 对象（这个 DynamicPluginManager 是写在宿主里面的）

640 (1042×360)

当我们调用 DynamicPluginManager 的 enter 方法进行插件操作的时候，会调用 updateManagerImpl() 方法，尝试去加载真正的Manager实现类：

```
@Override
public void enter(Context context, String uuid, long fromId, Bundle bundle, EnterCallback callback) {
                    ...
    if (mManagerImpl == null) {
        updateManagerImpl(context);
    }
                    ...
    mManagerImpl.enter(context, uuid, fromId, bundle, callback);
    mUpdater.update();
}

private void updateManagerImpl(Context context) {
                                ...
        ManagerImplLoader implLoader = new ManagerImplLoader(context, latestManagerImplApk);
        PluginManagerImpl newImpl = implLoader.load();
                                ...
                                mManagerImpl = newImpl;
                                ...
}
```

ManagerImplLoader 的构造方法会创建 PluginManager.apk 所释放的路径：

图片

然后再看看 ManagerImplLoader 的 load() 方法：

这个工厂类是通过ApkClassLoader反射加载的 ManagerFactoryImpl，ManagerFactoryImpl 是需要我们把包名和类名丝毫不动的声明在PluginManager.apk中，再看一下 ManagerFactoryImpl 的 buildManager() 调用：

最终，加载出来的是我们自己在 PluginManager.apk 中创建的 SamplePluginManager，这就是我们真正调用的 PluginManager。

4.3.3 plugin-runtime

从 4.3.1 的插件运行流程图，我们可以看到 plugin-runtime 的加载是：PluginManager 通过 IBinder 去跨进程通知插件 Service 加载的，这个插件 Service，就是咱们在宿主中定义的继承 PluginProcessService 的 Service。

所以我们来看一下 PluginProcessService 里面是如何加载 plugin-runtime 模块的：

咱们再去看看 DynamicRuntime.loadRuntime() 是怎么调用的：

这里可以看到一个 hack 方法，shadow不是说0hack么？我们再一探究竟：

总结

从上面的流程可以看出，加载 plugin-runtime 模块，其实就是为了破坏宿主的 ClassLoader parent 引用链，将 RuntimeClassLoader 插入到宿主ClassLoader和其Parent的中间位置，最终修改后的父子关系如下：

--BootClassLoader

----RuntimeClassLoader

------PathClassLoader

通过这种方式，通过类加载的双亲委派的机制，宿主的 PathClassLoader 加载不到的类（如代理 Activity），就回去parent查找，RuntimeClassLoader 则可以加载这个类。

4.3.4 plugin-loader

和 plugin-runtime 模块类似，plugin-loader 模块也是 PluginManager 通过 IBinder 去 PluginProcessService 加载的。

我们这里看一下 PluginProcessService 加载 plugin-loader 的逻辑：

```
    void loadPluginLoader(String uuid) throws FailedException {
            ...
                                //1.获取已安装的 plugin-loader。
                                InstalledApk installedApk;
                                installedApk = mUuidManager.getPluginLoader(uuid);
                                ...
                                //2.加载 plugin-loader 的映射接口 PluginLoaderImpl。
                                PluginLoaderImpl pluginLoader = new LoaderImplLoader().load(installedApk, uuid, getApplica
tionContext());
                                pluginLoader.setUuidManager(mUuidManager);

                                //3.PluginProcessService 成员变量缓存 pluinloader。
                                mPluginLoader = pluginLoader;
    }

            //获取操作 plugin-loader 的 IBinder。
            IBinder getPluginLoader() {
        return mPluginLoader;
    }
```

我们先看一下 PluginLoaderImpl 这个类，它其实就是一个继承了 IBinder 的接口。在 4.3.1 的运行流程图，我们可以看到 PluginManager 是通过 IBinder 方式去 PluginProcessService 获取到了 PluginLoader 的 IBinder，这样 PluginManager 就可以通过 PluginLoader 的 IBinder 直接对 plugin-loader 进行插件操作了，而 PluginLoader 的 IBinder 就是这个 PluginLoaderImpl 的实现：

 图片

我们在看看 LoaderImplLoader 的 load() 方法是如何调用的：

 图片