Shadow插件框架分析

1、简介

`Shadow`是一个腾讯自主研发的`Android`插件框架，主要有以下特点：

- **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支持**：`core.loader`，`core.transform`核心代码完全用`Kotlin`实现，代码简洁易维护。

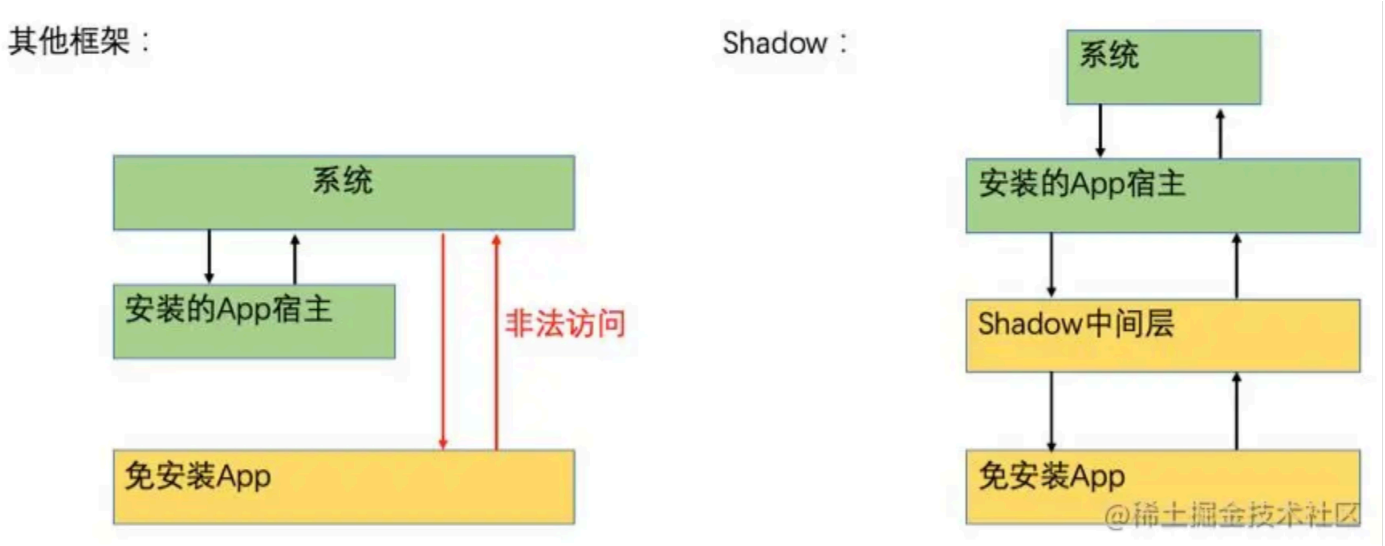
`Shadow`主要解决了两个大问题:

- 问题一：`Android 9.0`开始限制非公开SDK接口访问，机型适配问题出现。
- 问题二：插件框架不完善，其本身的代码需要更新、修复。

2、实现原理

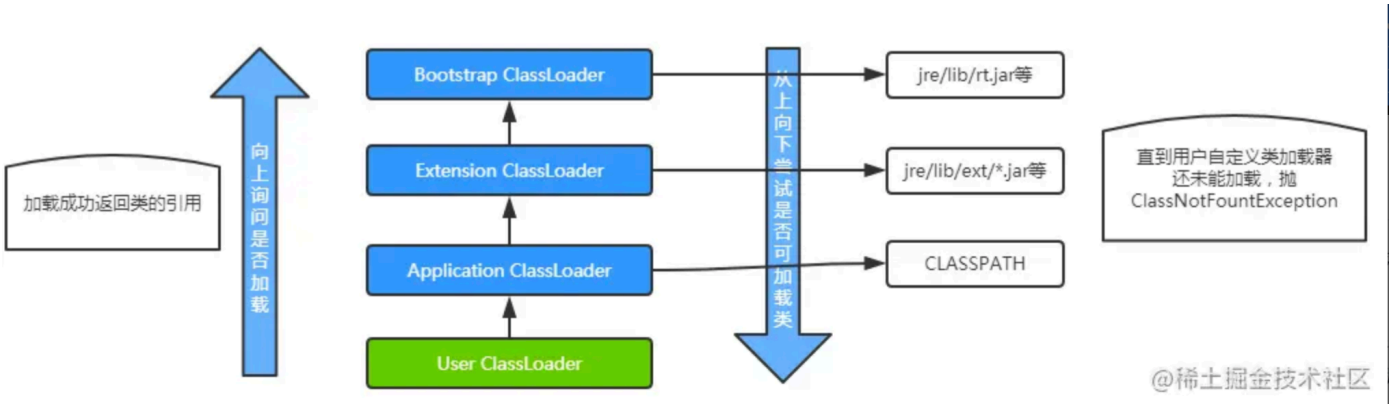
`Shadow`的原则是不去跟系统对抗。既然只是限制非公开SDK接口访问，而没有限制动态加载代码。插件技术的目的本质上来说还是动态加载代码。

`Shadow`通过运用AOP思想，利用字节码编辑工具，在编译期把插件中的所有`Activity`的父类都改成一个普通类，然后让壳子持有这个普通类型的父类去转调它就不用Hack任何系统实现。



3、类加载机制

我们编译好的`class`文件，需要先加载到虚拟机然后才会执行，这个过程是通过`ClassLoader`来完成的。



3.1、双亲委派模型：

- 1.加载某个类的时候，这个类加载器不会自己立刻去加载，它会委托给父类去加载
- 2.如果这个父类还存在父类加载器，则进一步委托，直到最顶层的类加载器
- 3.如果父类加载器可以完成加载任务，就成功返回，否则就再委派给子类加载器
- 4.如果都未加载成功就抛出`ClassNotFoundException`

3.2、双亲委派作用：

1.避免类的重复加载。
比如有两个类加载器，他们都要加载同一个类，这时候如果不是委托而是自己加载自己的，则会将类重复加载到方法区。
2.避免核心类被修改。
比如我们在自定义一个`java.lang.String`类，执行的时候会报错，因为`String`是`java.lang`包下的类，应该由启动类加载器加载。

`JVM`并不会一开始就加载所有的类，它是当你使用到的时候才会去通知类加载器去加载。

3.3、Android类加载机制

当我们`new`一个类时，首先是`Android`的虚拟机（`Dalvik/ART`虚拟机）通过`ClassLoader`去加载`dex`文件到内存。`Android`中的`ClassLoader`主要是`PathClassLoader`和`DexClassLoader`,这两者都继承自`BaseDexClassLoader`。它们都可以理解成应用类加载器。

`PathClassLoader`和`DexClassLoader`的区别：

- `PathClassLoader`只能指定加载apk包路径，不能指定dex文件解压路径。该路径是写死的在`/data/dalvik-cache/`路径下。所以只能用于加载已安装的apk。
- `DexClassLoader`可以指定apk包路径和dex文件解压路径（加载jar、apk、dex文件）

4、框架全动态分析

全动态指的就是除了插件代码之外，插件框架本身的所有逻辑代码也都是动态的。`Shadow`将框架分为四部分：`Host`、`Manager`、`Loader`和`Runtime`。其中除`Host`外，`Manager`、`Loader`、`Runtime`都是动态的。

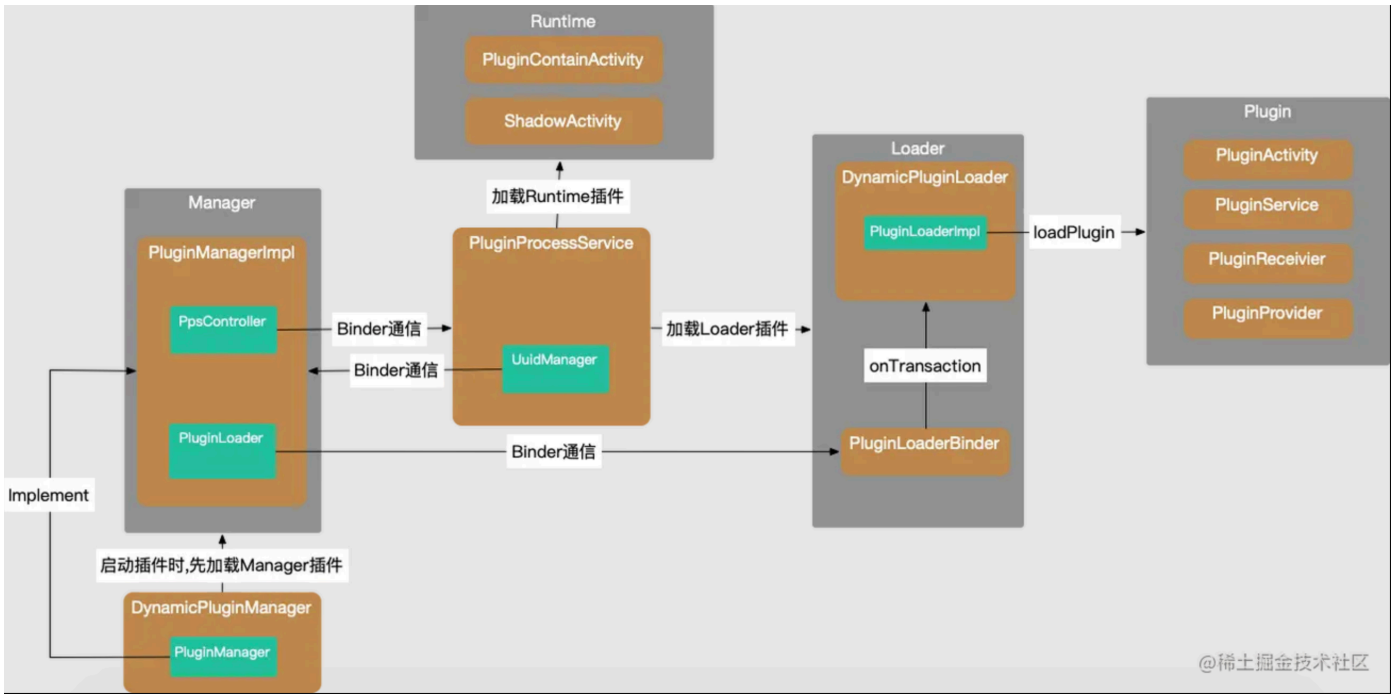
`Host`：`Host`打包在宿主中，它负责两件事情：

1. 为`Manger`、`Loader`、`Runtime`运行提供接口。
2. 加载`Manager`、`Loader`、`Runtime`等插件。它有两个重要的类`DynamicPluginManager`和`PluginProcessService`。`DynamicPluginManager`的`enter()`方法是程序的入口，该方法实现了加载`Manager`的逻辑。`PluginProcessService`加载`Loader`和`Runtime`。

`Manager`：管理插件，包括插件的下载逻辑、入口逻辑、预加载逻辑等。反正就是一切还没有进入到`Loader`之前的所有事情。开源的部分主要包括插件的安装（`apk`存放指定路径、`odex`优化、解压`so`库等）。

`Loader`：`Loader`是框架的核心部分。主要负责加载插、管理四大组件的生命周期、`Application`的生命周期等功能

`Runtime`：`Runtime`这一部分主要是注册在`AndroidManifest.xml`中的一些壳子类。Shadow作者对这部分的描述是被迫动态化。原因是宿主对合入代码的增量要求极其严格，而壳子类会引入大量的方法增量，因此被迫把这部分做成动态化。这也迫使Shadow引入了整套方案中唯一一处Hook系统的API。



4.1、Manager的动态化实现

整个框架的入口是`DynamicPluginManager`的`enter()`方法。我们从`enter()`开始分析`Manager`的动态化。`enter()`主要逻辑如下：

1. 加载`Manager`插件。
2. 通过反射实例化`Manager`内的`PluginManagerImpl`。
3. 将处理逻辑委托给`PluginMangerImpl`的`enter()`。

```
final class ManagerImplLoader extends ImplLoader {
    private static final String MANAGER_FACTORY_CLASS_NAME =
"com.tencent.shadow.dynamic.impl.ManagerFactoryImpl";
    PluginManagerImpl load() {
        //1. 加载Manager插件
        ApkClassLoader apkClassLoader = new ApkClassLoader(
                installedApk, //Manager插件APK。
                getClass().getClassLoader(),
                loadWhiteList(installedApk),
                1
        );
        //支持Resource相关和ClassLoader，允许Manager插件使用资源。
        Context pluginManagerContext = new ChangeApkContextWrapper(
                applicationContext,  //Applicaton
                installedApk.apkFilePath,
                apkClassLoader
        );
        try {
            //2.反射得到Manager中的类com.tencent.shadow.dynamic.impl.ManagerFactoryImpl的
实例，调用buildManager生成PluginManagerImpl。
            ManagerFactory managerFactory = apkClassLoader.getInterface(
                    ManagerFactory.class,
                    MANAGER_FACTORY_CLASS_NAME
            );
            return managerFactory.buildManager(pluginManagerContext);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

随后调用`PluginManagerImpl`的`enter()`方法。`DynamicPluginManager`只是一个代理类，真正的处理类是`Manager`插件中的`PluginManagerImpl`。

4.2、Runtime的动态化实现

`Runtime`的加载过程是在`PluginProcessService`的`loadRuntime`中完成的。

```
public class PluginProcessService extends Service {
  //与宿主通信的代理对象
  private UuidManager mUuidManager;
  void loadRuntime(String uuid) throws FailedException {
      //...省略代码
        //获取Runtime的安装信息
        InstalledApk installedRuntimeApk = mUuidManager.getRuntime(uuid);
        //加载Runtime.
        boolean loaded = DynamicRuntime.loadRuntime(installedRuntimeApk);
      //...省略代码
    }
}
```

`loadRuntime`主要完成了两件事情:

1. 通过`mUuidManage`获取`Runtime`的安装信息。之前说过`Runtime`的安装是在`Manager`中完成的，而`Manager`运行在宿主进程中，因此需要`Binder`通信。
2. 调用`DynamicRuntime.loadRuntime()`加载`Runtime`。

```
public class DynamicRuntime{

    public static boolean loadRuntime(InstalledApk installedRuntimeApk) {
        ClassLoader contextClassLoader = DynamicRuntime.class.getClassLoader();
        RuntimeClassLoader runtimeClassLoader = getRuntimeClassLoader();
        if (runtimeClassLoader != null) {
            String apkPath = runtimeClassLoader.apkPath;
            if (TextUtils.equals(apkPath, installedRuntimeApk.apkFilePath)) {
                //已经加载相同版本的runtime了,不需要加载
                return false;
            } else {
                //版本不一样，说明要更新runtime，先恢复正常的classLoader结构
                recoveryClassLoader();
            }
        }
        try {
             //正常处理，将runtime 挂到pathclassLoader之上
            hackParentToRuntime(installedRuntimeApk, contextClassLoader);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return true;
    }

    private static void hackParentToRuntime(InstalledApk installedRuntimeApk,
ClassLoader contextClassLoader) throws Exception {
        //RuntimeClassLoader加载Runtime插件。
```

```
        RuntimeClassLoader runtimeClassLoader = new
RuntimeClassLoader(installedRuntimeApk.apkFilePath, installedRuntimeApk.oDexPath,
                installedRuntimeApk.libraryPath, contextClassLoader.getParent());
        //这是全框架中唯一一处反射系统API的地方。但这是Java层提供的API，相对来说风险较小。
        Field field = getParentField();
        if (field == null) {
            throw new RuntimeException("在ClassLoader.class中没找到类型为ClassLoader的
parent域");
        }
        field.setAccessible(true);
        field.set(contextClassLoader, runtimeClassLoader);
    }
}
```

`loadRuntime`用到了全框架中唯一一处反射调用系统的私有API，它的作用是将加载`Runtime`的`RuntimeClassLoader`挂载到系统的`PathClassLoader`之上。也就是将`RuntimeClassLoader`作为`PathClassLoader`的父加载器。为什么这样处理呢？因为`Runtime`主要是一些壳子类，例如壳子`Activity`。在系统启动插件中的`Activity`时，其实是启动这些壳子`Activity`。这就要保证系统的`PathClassLoader`必须能找到壳子`Activity`。简单的方式就是利用双亲委派模型，把`PathClassLoader`的父加载器设置成`RuntimeClassLoader`。

4.3、Loader的动态化实现

`Loader`的加载过程与`Runtime`一样，也是在`PluginProcessService`中完成的。主要流程如下：

1. 获取`Loader`的安装信息。
2. 在`LoaderImplLoader`中加载`Loader`插件。

```
public class PluginProcessService extends Service {

    private PluginLoaderImpl mPluginLoader;
    private UuidManager mUuidManager;

    void loadPluginLoader(String uuid) throws FailedException {
        //获取Loader的安装信息
        InstalledApk installedApk = mUuidManager.getPluginLoader(uuid);
        //交给LoaderImplLoader.load()加载loader。
        PluginLoaderImpl pluginLoader = new LoaderImplLoader().load(installedApk, uuid,
getApplicationContext());
        pluginLoader.setUuidManager(mUuidManager);
        mPluginLoader = pluginLoader;
    }

    IBinder getPluginLoader() {
        return mPluginLoader;
    }
}
```

```
public class PpsController {
    public IBinder getPluginLoader() throws RemoteException {
        Parcel _data = Parcel.obtain();
        Parcel _reply = Parcel.obtain();
        IBinder _result;
        try {
            _data.writeInterfaceToken(PpsBinder.DESCRIPTOR);
            mRemote.transact(PpsBinder.TRANSACTION_getPluginLoader, _data, _reply, 0);
            _reply.readException();
            _result = _reply.readStrongBinder();
        } finally {
            _reply.recycle();
            _data.recycle();
        }
        return _result;
    }
}
```

`LoaderImplLoader.load()`返回的`PluginLoaderImpl`是一个`Binder`对象。我们的`Manager`可以通过`PpsController.getPluginLoader()`得到其代理。

5、项目目录结构

```
├── projects
│   ├── sample // 示例代码
│   │   ├── README.md
│   │   ├── maven
│   │   ├── sample-constant // 定义一些常量
│   │   ├── sample-host // 宿主实现
│   │   ├── sample-manager // PluginManager 实现
│   │   └── sample-plugin // 插件的实现
│   ├── sdk // 框架实现代码
│   │   ├── coding // lint
│   │   ├── core
│   │   │   ├── common
│   │   │   ├── gradle-plugin // gradle 插件
│   │   │   ├── load-parameters
│   │   │   ├── loader // 负责加载插件
│   │   │   ├── manager // 装载插件，管理插件
│   │   │   ├── runtime // 插件运行时需要，包括占位 Activity，占位 Provider 等等
│   │   │   ├── transform // Transform 实现，打包时用于替换插件 Activity 父类等等
│   │   │   └── transform-kit
│   │   └── dynamic // 插件自身动态化实现，包括一些接口的抽象
```

项目框架结构图

6、源码分析

6.1、在AndroidManifest中注册

首先需要在宿主`AndroidMainfest.xml`文件中注册壳子`Activity`：`SightSingleTaskActivity`，对应插件进程：`MainPluginProcessService`，壳子Provider:`PluginContainerContentProvider`。

```
 <service android:name="com.run.sports.sight.MainPluginProcessService" />

<activity android:name="com.run.sports.runtime.SightSingleTaskActivity"/>

<provider

android:name="com.tencent.shadow.core.runtime.container.PluginContainerContentProvider"
    android:authorities="${applicationId}.contentprovider.authority.dynamic"
    android:grantUriPermissions="true" />
```

6.2、宿主调用插件入口

在宿主代码中调用`PluginManager`的`enter()`方法如下

```
pluginManager = new DynamicPluginManager(new FixedPathPmUpdater(new File(validFolder,
FILE_NAME_PLUGIN_MANAGER)));
Bundle bundle = new Bundle();
bundle.putString(EXTRA_KEY_PLUGIN_ZIP_PATH, new
 File(validFolder,FILE_NAME_PLUGIN_ZIP_FILE).getAbsolutePath());
pluginManager.enter(HSApplication.getContext(), FROM_ID_START_SIGHT, bundle, new
EnterCallback() {

        //...

        });
```

在`DynamicPluginManager`中`enter()`方法为

```
    @Override
    public void enter(Context context, long fromId, Bundle bundle, EnterCallback
callback) {
        if (mLogger.isInfoEnabled()) {
            mLogger.info("enter fromId:" + fromId + " callback:" + callback);
        }
        //加载插件管理器PluginManager，为SightPluginManager
        updateManagerImpl(context);
        //调用SightPluginManager的enter()方法
        mManagerImpl.enter(context, fromId, bundle, callback);
        mUpdater.update();
    }
```

```
private void updateManagerImpl(Context context) {

        //...
        //调用管理器的加载类，加载插件内设置的白名单
        ManagerImplLoader implLoader = new ManagerImplLoader(context,
latestManagerImplApk);
        //加载插件管理器
        PluginManagerImpl newImpl = implLoader.load();

        //...

    }
}
```

`ManagerImplLoade`r的核心代码为

```
final class ManagerImplLoader extends ImplLoader {
```

```
    // ....

    PluginManagerImpl load() {
        //Apk插件加载专用ClassLoader，用于隔离插件和宿主app
        ApkClassLoader apkClassLoader = new ApkClassLoader(
                installedApk,
                getClass().getClassLoader(),
                loadWhiteList(installedApk),
                1
        );
        //修改Context的apk路径的Wrapper。可将原Context的Resource和ClassLoader重新修改为新的Apk。
        Context pluginManagerContext = new ChangeApkContextWrapper(
                applicationContext,
                installedApk.apkFilePath,
                apkClassLoader
        );

        try {
            ManagerFactory managerFactory = apkClassLoader.getInterface(
                    ManagerFactory.class,
                    MANAGER_FACTORY_CLASS_NAME
            );
            //构建Manager
            return managerFactory.buildManager(pluginManagerContext);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

上述代码都在宿主`app`中，

```
/**
 * 此类包名及类名固定
 */
public final class ManagerFactoryImpl implements ManagerFactory {
    @Override
    public PluginManagerImpl buildManager(Context context) {
        return new SightPluginManager(context);
    }
}
```

上述所说的`mManagerImpl.enter(context, fromId, bundle, callback);`方法就是调用

`SightPluginManager`方法中的`enter()`方法

```
  @Override
```

```
    public void enter(final Context context, long fromId, Bundle bundle, final
EnterCallback callback) {

            //...

            executorService.execute(() -> {
                try {

                    //解压插件包，获取内部插件信息，并保存到数据库中
                    InstalledPlugin installedPlugin = installPlugin(pluginZipPath,
null, true);
                    //根据插件key，加载Runtime、PluginLoader
                    loadPlugin(installedPlugin.UUID, "plugin-sight");
                    //启动插件app
                    callApplicationOnCreate("plugin-sight");
                } catch (Exception e) {
                    if (BuildConfig.DEBUG) {
                        throw new RuntimeException(e);
                    }
                }
            });

            return;
        }
        if (BuildConfig.DEBUG) {
            throw new IllegalArgumentException("unknown fromId, from id = " + fromId);
        }

    }
```

在`enter`方法中是内部启了一个线程去加载插件。

```
  private void loadPluginLoaderAndRuntime(String uuid, String partKey) throws
RemoteException, TimeoutException, FailedException {
        if (mPpsController == null) {
            //绑定service 启动插件进程
            bindPluginProcessService(getPluginProcessServiceName(partKey));
            waitServiceConnected(10, TimeUnit.SECONDS);
        }
        //加载runtime时需要的数据
        loadRunTime(uuid);
        //把插件代码加载到classLoader中并生成pluginLoader
        loadPluginLoader(uuid);
    }

    protected void loadPlugin(String uuid, String partKey) throws RemoteException,
TimeoutException, FailedException {

        loadPluginLoaderAndRuntime(uuid, partKey);
```

```
        Map map = mPluginLoader.getLoadedPlugin();
        if (!map.containsKey(partKey)) {
            mPluginLoader.loadPlugin(partKey);
        }
    }
```

6.3、加载`Runtime`

主要作用：加载`runtime.apk`，修改父类的`classLoader`。

```
    public final void loadRunTime(String uuid) throws RemoteException, FailedException {
        if (mLogger.isInfoEnabled()) {
            mLogger.info("loadRunTime mPpsController:" + mPpsController);
        }

        PpsStatus ppsStatus = mPpsController.getPpsStatus();
        if (!ppsStatus.runtimeLoaded) {
            mPpsController.loadRuntime(uuid);
        }
    }
```

```
void loadRuntime(String uuid) throws FailedException {
    checkUuidManagerNotNull();
    setUuid(uuid);

    try {
        // ...

        InstalledApk installedRuntimeApk = new InstalledApk(installedApk.apkFilePath,
installedApk.oDexPath, installedApk.libraryPath);

        // 将runtime apk加载到DexPathClassLoader，形成如下结构的classLoader树结构 ---
BootClassLoader ----        RuntimeClassLoader ------PathClassLoader
        boolean loaded = DynamicRuntime.loadRuntime(installedRuntimeApk);
        if (loaded) {
            DynamicRuntime.saveLastRuntimeInfo(this, installedRuntimeApk);
        }
        mRuntimeLoaded = true;
    } catch (RuntimeException e) {
        if (mLogger.isErrorEnabled()) {
            mLogger.error("loadRuntime发生RuntimeException", e);
        }
        throw new FailedException(e);
    }
}
```

在`DynamicRuntime`的`loadRuntime()`方法中生成了`RuntimeClassLoader`

```
public static boolean loadRuntime(InstalledApk installedRuntimeApk) {
    ClassLoader contextClassLoader = DynamicRuntime.class.getClassLoader();
    RuntimeClassLoader runtimeClassLoader = getRuntimeClassLoader();

    // ...

    //正常处理，将runtime 挂到pathclassLoader之上
    try {
        hackParentToRuntime(installedRuntimeApk, contextClassLoader);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    return true;
}
```

```
private static void hackParentToRuntime(InstalledApk installedRuntimeApk, ClassLoader
contextClassLoader) throws Exception {
        RuntimeClassLoader runtimeClassLoader = new
RuntimeClassLoader(installedRuntimeApk.apkFilePath,     installedRuntimeApk.oDexPath,
                installedRuntimeApk.libraryPath, contextClassLoader.getParent());
        hackParentClassLoader(contextClassLoader, runtimeClassLoader);
    }

static void hackParentClassLoader(ClassLoader classLoader,
                                      ClassLoader newParentClassLoader) throws
Exception {
        Field field = getParentField();
        if (field == null) {
            throw new RuntimeException("在ClassLoader.class中没找到类型为ClassLoader的
parent域");
        }
        field.setAccessible(true);
        field.set(classLoader, newParentClassLoader);
    }
```

将原本的`BootClassLoader <- PathClassLoader`结构变为`BootClassLoader <- RuntimeClassLoader <- PathClassLoader`，

这里是唯一一处使用反射的位置，零反射是和传统插件框架解决动态加载Activity等组件时是否使用反射来对比的。

6.4、加载`Loader`

主要作用：通过进程间通信加载`pluginLoader`，并把插件加载到`classloader`中

```
public final void loadPluginLoader(String uuid) throws RemoteException, FailedException
{
        if (mLogger.isInfoEnabled()) {
```

```
            mLogger.info("loadPluginLoader mPluginLoader:" + mPluginLoader);
        }
        if (mPluginLoader == null) {
            //内部进行通信
            PpsStatus ppsStatus = mPpsController.getPpsStatus();
            if (!ppsStatus.loaderLoaded) {
                mPpsController.loadPluginLoader(uuid);
            }
            IBinder iBinder = mPpsController.getPluginLoader();
            mPluginLoader = new BinderPluginLoader(iBinder);
        }
    }
```

在`loadPluginLoader()`中进行进程间通信，最终是调用抽象类`ShadowPluginLoader`的`loadPlugin`

```
    @Throws(LoadPluginException::class)
     open fun loadPlugin(
         installedApk: InstalledApk
     ): Future<*> {
         val loadParameters = installedApk.getLoadParameters()
         // ...
         return LoadPluginBloc.loadPlugin(
             mExecutorService,
             mComponentManager,
             mLock,
             mPluginPartsMap,
             mHostAppContext,
             installedApk,
             loadParameters
         )
     }
```

```
object LoadPluginBloc {
    @Throws(LoadPluginException::class)
    fun loadPlugin(
        //...
        loadParameters: LoadParameters
    ): Future<*> {

            //...
            val buildClassLoader = executorService.submit(Callable {
                lock.withLock {
                    //该方法把插件代码加入到classLoader
                    LoadApkBloc.loadPlugin(installedApk, loadParameters,
pluginPartsMap)
                }
            })
```

```
            //...
            return buildRunningPlugin
        }
    }
}
```

6.5 、启动插件`app`

调用`callApplicationOnCreate()`方法，主要作用：启动插件`app`

```
    protected void callApplicationOnCreate(String partKey) throws RemoteException {

        Map map = mPluginLoader.getLoadedPlugin();
        Boolean isCall = (Boolean) map.get(partKey);
        if (isCall == null || !isCall) {
            mPluginLoader.callApplicationOnCreate(partKey);
        }
    }
```

`callApplicationOnCreate()`方法先判断插件是否已经加载，没有加载过则加载。

```
fun callApplicationOnCreate(partKey: String) {
        fun realAction() {
            val pluginParts = getPluginParts(partKey)
            pluginParts?.let {
                val application = pluginParts.application
                application.attachBaseContext(mHostAppContext)
                mPluginContentProviderManager.createContentProviderAndCallOnCreate(
                    application, partKey, pluginParts
                )
                //调用插件application的onCreate()方法，启动插件。
                application.onCreate()
            }
        }

      //...
    }
```

上述是加载和启动插件的整个过程。

6.6 、插件中如何启动`Activity`

在宿主`AndroidManifest.xml`文件中注册的有一个壳子`Activity`，在插件中有`ComponentManage`r类管理插件中的`Activity`与壳子之间的关系

```
class FastPluginManager {
    public void startPluginActivity(Context context, InstalledPlugin installedPlugin,
String partKey, Intent pluginIntent) throws RemoteException, TimeoutException,
FailedException {
        Intent intent = convertActivityIntent(installedPlugin, partKey, pluginIntent);
        if (!(context instanceof Activity)) {
            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        }
        context.startActivity(intent);
    }
}
```

通过调用`convertActivityIntent()`方法把

```
internal class DynamicPluginLoader(hostContext: Context, uuid: String) {

    //...

    fun convertActivityIntent(pluginActivityIntent: Intent): Intent? {
        return
mPluginLoader.mComponentManager.convertPluginActivityIntent(pluginActivityIntent)
    }
}
```

```
abstract class ComponentManager : PluginComponentLauncher{

  override fun convertPluginActivityIntent(pluginIntent: Intent): Intent {
          //判断是否是插件内的一个组件  是的话
          return if (pluginIntent.isPluginComponent()) {
              pluginIntent.toActivityContainerIntent()
          } else {
              pluginIntent
          }
   }

  private fun Intent.toActivityContainerIntent(): Intent {
        val bundleForPluginLoader = Bundle()
        val pluginActivityInfo = pluginActivityInfoMap[component]!!
        bundleForPluginLoader.putParcelable(CM_ACTIVITY_INFO_KEY, pluginActivityInfo)
        return toContainerIntent(bundleForPluginLoader)
  }

  private fun Intent.toContainerIntent(bundleForPluginLoader: Bundle): Intent {

      // ...
      //传入数据把插件内Intent转换成宿主内启动壳子Activity所需Intent
      val containerComponent = componentMap[component]
```

```
            ?: throw IllegalArgumentException("已加载的插件中找不到${component}对应的
ContainerActivity")
        val containerIntent = Intent(this)
        containerIntent.component = containerComponent

        bundleForPluginLoader.putString(CM_CLASS_NAME_KEY, className)
        bundleForPluginLoader.putString(CM_PACKAGE_NAME_KEY, packageName)

        containerIntent.putExtra(CM_EXTRAS_BUNDLE_KEY, pluginExtras)
        containerIntent.putExtra(CM_BUSINESS_NAME_KEY, businessName)
        containerIntent.putExtra(CM_PART_KEY, partKey)
        containerIntent.putExtra(CM_LOADER_BUNDLE_KEY, bundleForPluginLoader)
        containerIntent.putExtra(LOADER_VERSION_KEY, BuildConfig.VERSION_NAME)
        containerIntent.putExtra(PROCESS_ID_KEY, DelegateProviderHolder.sCustomPid)
        return containerIntent
    }
}
```

7、我在开发中遇到的问题

1、插件加载宿主资源问题以及资源混淆问题，在插件中加载`expressAd`，而`goldeye`初始化是在宿主中注册

解决方法：避免插件加载宿主的资源进行混淆，把加入到`andResGuard`的资源白名单中，插件中调用通过`context.getIdentifier()`方法获取对应id

2、`ActivityStart`库启动`Reward`视频广告页，启动失败

解决方法：在宿主中回调并启动`reward`视频页。