Replugin插件化框架原理简介

版本号:2019-03-11(23:05)

简介

为什么需要插件化?

1、为什么需要插件化?

> 1. 发布不够灵活
> 2. 安装包过大
> 3. 模块不够独立，耦合度高。

导致插件化不稳定的根源在哪?

2、导致插件化框架不稳定的根源在哪?

> 1. Hook太多--- 修改了太多的系统API
> 2. 市面上可以hook的内容:
> 3. AMS
> 4. Instrumentation
> 5. System-Services
> 6. PackageManager
> 7. ContextImpl
> 8. Resources

3、哪些情况可能会导致Hook引起稳定性问题?

> 1. Android升级
> 2. ROM修改
> 3. Hook的使用不当，Hook点选择不正确

4、Hook要越少越好!

5、总结插件化稳定的方法

> 1. Hook一处：宿主的ClassLoader
> 2. DexClassLoader原生
> 3. Resources原生
> 4. PluginContext非Hook，new出来的
> 5. 坚持 真正、长久的稳定 ， 大量适配过、非长久的稳定是伪稳定

如何解决不灵活的问题?

1、灵活的目标

> 1. 插件组件随意增改

2. 新插件直接用
3. 主程序长期不发版本
4. 自由设置进程

大项目才使用插件化?

1、大项目才使用插件化?

1. 插件 = 免安装应用
2. 基础放在主程序中，放心
3. 迁移成本高

2、全面插件化

1. 自由、独立发布
2. 应用“积木化”
3. 宿主/插件交互简单

Hook技术

PathClassLoader的Hook

1、RePlugin如何通过Hook实现插件化?

1. 仅仅Hook ClassLoader
2. 最终使用本身的 `RepluginClassLoader`

```
Application.mBase.mPackageInfo.mClassLoader
    = new RepluginClassLoader();
```

2、RePlugin Hook的ClassLoader仅仅更改了loadClass，其他的都直接透传。

1. 系统原生的四大组件等内容，通过 `Hook的RepluginClassLoader` 的 `loadClass()` 方法转换成插件所需要的内容，并交给系统。
2. `loadClass()` 里面有 `PitManager`、`HookingClassManager` 进行处理

3、ClassLoader一定要继承自 `PathClassLoader`

1. 防止 Android 7.x 因使用 addDexPath 而有问题。
2. 除此之外，此 ClassLoader 所在位置也非常稳定。目前来看，从 Android 2.1 至今都没有发生过位置、名称上的变化，可以长期使用。

Dex的Hook

3、RePlugin中不对Dex进行Hook

> 1. 使用原生的 `DexClassLoader`
>
> - 为了支持使用宿主类，后Hook了 `PluginDexClassLoader`

```
// 1、获取插件apk的路径
mPath = getPluginApkPath();
// 2、获取到ODEX中的目录的路径
String out = mContext.getDir(ODEX_DIR, 0).getPath();
// 3、获取到父ClassLoader
ClassLoader parent = getClass().getClassLoader().getParent();
// 4、创建DexClassLoader
mClassLoader = new DexClassLoader(mPath, out, parent);
```

资源的Hook

4、资源使用原生的Resources

```
// 1、获取包信息
mPackageInfo = pm.getPackageArchiveInfo(mPath, ...);
// 2、获取应用信息
ApplicationInfo ai = mPackageInfo.applicationInfo;
// 3、获取到原生的Resources
mPkgResources = pm.getResourcesForApplication(ai);
```

Context的Hook

5、不对宿主Apk的Context进行Hook，使用原生Context

6、对插件的Context会new PluginContext。是非Hook的?

> 1. Hook为 `PluginContext`

```
mPigContext = new PluginContext(...);
```

> 1. 这里PluginContext并不是Hook的，本身没有改任何系统API，只是交给插件这个PluginContext，去做特殊的工作。

PluginContext

7、RepluginContext是什么?

> 1. 由 `Resources、Dex ClassLoader、Application Context` 组成
> 2. 将 `RepluginContext` 作为 `mBase`
> 3. 也就是 `Activity、Service、...` 这些的基础

8、RepluginContext的工作

功能	方法
插件的原生DexClassLoader	getClassLoader()
插件的原生Resources	getResources()
Resources.Assets	getAssets()
修改LayoutInflater的Factory	getSystemService()
全新的Service方案	startService()/bindService()
重定向到插件目录	getSharedPreferences()
重定向到插件目录	文件操作
插件的Application对象	getApplicationContext()

9、LayoutInflater是什么？有什么用?

10、Resources是什么？有什么用?

坑位方案

1、坑位是什么?非坑位是什么?

> 1. 非坑位就是一一对应关系:
>   `<activity name="xxxActivity">` 对应的就是 `xxxActivity`
> 2. 坑位就是租赁关系：一个坑位对应多个目标。
>   `<activity name="NISTI">` 对应的就是 `xxxActivity` 也可以是其他Activity。
>   - 通常情况会出现 `ClassNotFound` 。但是Activity的坑位方案，可以交给 `PluginClassLoader` 去找到 `Nx` 具体对应的是谁。

Activity坑位方案

2、Activity的启动流程

> 1. startActivity
> 2. AMS
> 3. Path ClassLoader(安卓系统)
> 4. ContextImpl
> 5. xxxActivity

3、通常情况Activity的启动流程中都需要大量的Hook点:

1. startActivity `(Hook)`
2. AMS `(Hook)`
3. Path ClassLoader `(Hook)`
4. ContextImpl `(Hook)`

4、如何避免Hook实现Activity的启动?Replugin的处理流程?

1. PM.startActivity()
   - 非Hook
   - `PluginContext.startActivity()`
   - 帮助处理是打开自己的Activity还是其他插件的Activity，插件直接调用startActivity即可。
   - 如果宿主需要调用 `RePlugin.startActivity()`
2. PitManager坑位的startActivity(Nx)
   - 非Hook
   - 直接调用 `context.startActivity(Nx)`
3. AMS
   - 原生
4. PluginClassLoader
   - **Hook**, 去找到刚才 `Nx` 到底对应的是哪个 `xxxActivity`
   - `classloader.loadClass(xxx)`
5. Dex ClassLoader
   - 原生
6. Plugin Context
   - 非Hook
7. xxxActivity

- 并非走了系统的再通过Hook去修改
- 而是直接先走自己的处理方法

5、Replugin的核心思路

1. 找坑
2. 开坑
3. 找ClassLoader拦截坑
4. 把坑返回给系统，让其显示个真实的

坑位的分层

6、坑位的分层

1. Standard
2. Single Task
3. Full Screen
4. Single Instance

> 5. Single Top

7、坑位的分层越多，支持度越高。

8、坑位满了怎么办?

> 1. 利用方案去完美的腾出坑位
> 2. 清除一些坑位，进行缓存
> 3. 事后进行恢复

9、坑位越多，越稳定。

10、支持Task Affinity

Service方案

1、Service的Hook方案中的启动流程

> 1. startService【Hook】
> 2. AMS【Hook】
> 3. Path ClassLoader【Hook】
> 4. ContextImpl【Hook】
> 5. xxxService

2、Service非Hook方案(ContentProvider进行模拟，只需要一个Service坑位)

> 1. PSC.startService
> 2. ContentProvider(Binder)【借助】
> 3. PSS.startServiceLocked【核心-】
> 4. HostClassLoader
> 5. DexClassLoader
> 6. PluginContext
> 7. xxxService
>
> - 前两步是调用者进程
> - 后两步是目标进程

3、为什么Service不采用Activity那种大量坑位的方法?

> 1. Activity前台可见的Activity有限。Service可能非常大量。
> 2. Service坑位满了，无法想Activity一样进行缓存和恢复。Service要一直在后台执行任务。

旧框架对Activity/Service的额外处理

1、插件中的Activity，内部都继承自PluginActivity

2、插件中的Service，都会进行stopSelf方法重定向

3、插件中的ContentResolver，各方法进行重定向

新框架的处理

1、动态编译方案

> 1. 源码/Jar -> .class -> 【动态编译】-> .class(new) -> .dex

2、动态编译做了哪些事情?

> 1. 将 `Activity\stopSelf()\ContentResolver.query()` 通过 `JavaAssist(gradle)` 进行处理
> 2. 处理为 `PluginActivity\PSC.stopSelf()\PPC.query()`

Component Pit Manager

Replugin设计的内容

静态Receiver

多进程坑位

Service & Provider & App

IPC(跨进程通信)

TaskAffinity & Theme

一行获取宿主/插件接口

SO库随意使用

总结

1、Replugin的优点在于: 稳定 + 灵活

> 1. Hook-1处，全新坑位方案
> 2. 多进程，静态Receiver
> 3. 支持20+特性
> 4. 动态编译方案(不需要开发者有任何工作量)

扩展知识

1、Protobuf是什么?

> 1. Protobuf是一种灵活高效可序列化的数据协议，相于XML，具有更快、更简单、更轻量级等特性。
> 2. 可很轻松地实现数据结构的序列化和反序列化
> 3. 一旦需求有变，可以更新数据结构，而不会影响已部署程序

问题汇总

参考资料

1. Jack Trout(杰克.克劳特)-《定位》系列
2. Android Protobuf应用及原理
3. 《RePlugin 插件化框架漫谈（GMTC 2017，Youku视频）》—— @GMTC（全球移动技术大会）
4. 《RePlugin框架实现原理和最佳实践》