转载请注明链接:https://blog.csdn.net/feather_wch/article/details/88544574

Replugin源码详解

版本号:2019-03-13(23:59)

Replugin的职责划分

1、Replugin分为四个部分的源码

Replugin官方库工程	职责
replugin-host-gradle	【宿主的gradle插件】 1.动态修改AndroidManifest，生成占坑的四大组件 2.动态生成宿主的配置项 `HostBuildConfig.java` 3.解析内置插件

Replugin官方库工程	职责
replugin-host-library	【宿主依赖的核心库】1.初始化整个框架 2.hook ClassLoader 3.加载/启动/管理插件
replugin-plugin-gradle	【插件的gradle插件】1. 利用 `Transfrom API` 和 `Javassit` 动态修改字节码， 让插件工程的Activity全部继承自Replugin库的xxxActivity 等等
replugin-plugin-library	【插件依赖的库】1. 利用反射来使用宿主的接口和功能

2、replugin-host-gradle的职责

> 1.编译过程中，动态修改AndroidManifest，生成占坑的四大组件。
>
> - 位于 `xxx\app\build\intermediates\manifests\full\debug\AndroidManifest`
>
> 2.动态生成宿主的配置项 `HostBuildConfig.java`
>
> - 位于 `xxx\app\build\generated\source\buildConfig\debug\com\qihoo360\replugin\gen`

```
// RePluginHostConfig.java
public class RePluginHostConfig {
    // 常驻进程名字
    public static String PERSISTENT_NAME = ":GuardService";
    // 是否使用“常驻进程”（见PERSISTENT_NAME）作为插件的管理进程。若为False，则会使用默认进程
    public static boolean PERSISTENT_ENABLE = true;
    // 背景透明的坑的数量（每种 launchMode 不同）
    public static int ACTIVITY_PIT_COUNT_TS_STANDARD = 2;
    public static int ACTIVITY_PIT_COUNT_TS_SINGLE_TOP = 2;
    public static int ACTIVITY_PIT_COUNT_TS_SINGLE_TASK = 2;
    public static int ACTIVITY_PIT_COUNT_TS_SINGLE_INSTANCE = 3;
    // 背景不透明的坑的数量（每种 launchMode 不同）
    public static int ACTIVITY_PIT_COUNT_NTS_STANDARD = 6;
    public static int ACTIVITY_PIT_COUNT_NTS_SINGLE_TOP = 2;
    public static int ACTIVITY_PIT_COUNT_NTS_SINGLE_TASK = 3;
    public static int ACTIVITY_PIT_COUNT_NTS_SINGLE_INSTANCE = 2;
    // TaskAffinity 组数
    public static int ACTIVITY_PIT_COUNT_TASK = 2;
    // 是否使用 AppCompat 库
    public static boolean ACTIVITY_PIT_USE_APPCOMPAT = true;
    //------------------------------------------------------------
    // 主程序支持的插件版本范围
    //------------------------------------------------------------
    // HOST 向下兼容的插件版本
    public static int ADAPTER_COMPATIBLE_VERSION = 10;
    // HOST 插件版本
    public static int ADAPTER_CURRENT_VERSION = 12;
}
```

> 3.解析内置插件，扫描 `assets/plugins` ， 解析包含文件名、包名、版本、路径的 `plugins-builtin.json`
>
> - 位于 `xxx\app\build\intermediates\assets\debug`

```
// plugins-builtin.json
[
    {
        "high":null,
        "frm":null,
        "ver":1,
        "low":null,
        "pkg":"com.hao.repluginlogin",
        "path":"plugins/login.jar",
        "name":"login"
    }
]
```

3、replugin-host-library的职责

> 1. 初始化整个框架
> 2. hook ClassLoader
> 3. 加载/启动/管理插件

4、replugin-plugin-gradle的职责

> 1. 利用 `Transfrom API` 和 `Javassit` 动态修改字节码，让插件工程的Activity全部继承自Replugin库的xxxActivity
> 2. 动态将插件Apk中调用 `LocalBroadcastManager` 的地方替换为Replugin库中的 `PluginLocalBroadcastManager`
> 3. 动态将 `ContentReolver`、`ContentProviderClient` 的调用替换成Replugin库中对应的调用
> 4. 动态修改插件工程中所有调用 `Resource.getIdentifier()`方法 的地方，将第三个参数修改为 `插件工程的包名`

5、replugin-plugin-library的职责

> 1. 利用反射来使用宿主的接口和功能

初始化

核心原理

1、Replugin的核心底层原理？如何做到宿主和插件之间的通信和数据共享？

> 1. 使用Binder机制
> 2. 类似于 `ServiceManager`和`AMS`

3. 将一个常驻进程或者宿主的主进程作为 Server端 (根据配置选择)
4. 其他的 插件进程 和 宿主进程 全部属于 Client端
5. Server端 ： 创建了一个Provider，通过 Provider的query 方法返回 Binder对象 来实现 多进程之间、宿主和插件之间的通信和数据共享
6. Server端作为插件管理进程，任务包括：插件的安装、卸载、更新、状态判断

2、Server端做了哪些事情?

1. 创建了一个Provider，通过Provider的query()方法返回 Binder对象 来实现 多进程之间、宿主和插件之间的通信和数据共享
2. 处理: 插件的安装、卸载、更新、状态判断、提取优化dex文件
3. 分配坑位
4. 启动坑位

3、Replugin通过Hook ClassLoader如何处理Activity的启动?

1. 能够对系统的类加载过程进行拦截
2. 将之前分配的坑位信息替换为 真正要启动的组件信息
3. 并且使用该组件对应的ClassLoader进行类的加载

4、Replugin 启动四大组件的大致流程

1. Client端，通知Server端，进行：
    1. 检查插件是否安装
    2. 安装插件
    3. 提取并优化dex文件
    4. 分配坑位
    5. 启动坑位(将真实组件的信息替换为坑位的信息)
2. 利用坑位欺骗系统的检查
3. 通过检查后Client开始真正加载目标组件
4. 用Hook的ClassLoader对系统的类加载过程进行拦截
5. 将坑位信息替换为真实组件的信息，并且用与其对应的ClassLoader进行类的加载

5、为什么要用与真实组件对应的ClassLoader去加载该组件?

关键字

Binder

ServiceManager和AMS

Server/Client

ContentProvider

ClassLoader

RePluginApplication

1、App在应用端所有的回调方法中，最早的一个回调方法是什么?

> 1. Application的 `attachBaseContext()` 中
> 2. 该方法时 `ContextWrapper` 中的方法，在Application创建后，调用 `attach()` 时回调该方法
> 3. 因此插件化框架中对系统ClassLoader等一切系统组件的Hook，都必须该方法中处理。避免不必要的故障。

2、RePluginApplication的源码分析

> 1. 提供创建配置对象的方法
> 2. 提供可创建RepluginCallbacks回调的空方法
> 3. attachBaseContext()
> 4. onCreate()

```
public class RePluginApplication extends Application {
  // 1、创建配置对象，提供可选的配置。
    protected RePluginConfig createConfig() {
        return new RePluginConfig();
    }
  // 2、创建RePluginCallbacks，能用于处理，例如插件不存在，启动下载流程，下载插件Apk
    protected RePluginCallbacks createCallbacks() {
        return null;
    }

    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);

        RePluginConfig c = createConfig();
        if (c == null) {
            c = new RePluginConfig();
        }

        RePluginCallbacks cb = createCallbacks();
        if (cb != null) {
            c.setCallbacks(cb);
        }

        RePlugin.App.attachBaseContext(this, c);
    }

    @Override
    public void onCreate() {
        super.onCreate();
        RePlugin.App.onCreate();
    }
}
```

RePluginApplication.attachBaseContext()

3、RePluginApplication.attachBaseContext()的源码解析

> 1. 核心操作1: `IPC.init()` 标记了当前进程的类型，包括: 进程名、进程id、宿主包名、设置常驻进程名、最后标记当前进程是UI进程还是常驻进程
> 2. 核心操作2: `PMF.init(app)` Hook了系统的PathClassLoader
> 3. 核心操作3: `PMF.callAttach()` 加载内置插件

```
// RePluginApplication.java
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);

    // 1、创建RePluginConfig
    RePluginConfig c = new RePluginConfig();
    // 2、设置RePluginCallbacks
    RePluginCallbacks cb = createCallbacks();
    c.setCallbacks(cb);
    // 3、RePlugin的静态内部类App的方法
    RePlugin.App.attachBaseContext(this, c);
}

// RePlugin.java的内部类App
public static void attachBaseContext(Application app, RePluginConfig config) {

    // 1、保证只会初始化一次
    // 2、存储Application对象
    RePluginInternal.init(app);
    // 3、存储RePluginConfig
    sConfig = config;
    /**=========================
     * 4、存储App所在文件目录的路径
     *  1. 存储App所在文件目录的路径
     *  2. 如果RePluginCallbacks为空创建一个
     *  3. 如果RePluginEventCallbacks为空创建一个
     *==========================*/
    sConfig.initDefaults(app);

    /**=====================================================
     * 5、标记当前进程的类型(如是不是常驻进程)
     *   1. 后续方法根据进程的类型执行不同的逻辑
     *   2. 不同进程的创建会导致【attachBaseContext】调用多次
     *=====================================================*/
    IPC.init(app);

    // 6、初始化HostConfigHelper（通过反射HostConfig来实现）
    HostConfigHelper.init();

    // 7、用于管理插件的运行状态，包括：正常运行、被禁用等
    PluginStatusController.setAppContext(app);

    /**=======================================
     * 8、【最重要】
     *    1. 真正初始化Replugin框架
     *    2. Hook了系统的PathClassLoader
     *=======================================*/
    PMF.init(app);
    // 9、加载默认插件
    PMF.callAttach();
}
```

IPC.init()

4、IPC.init()源码分析

1. 通过proc文件获取当前进程名
2. 获取当前进程pid、宿主程序包名
3. 设置当前进程是不是UI进程、是不是常驻进程的标志(两个布尔值)

```
public static void init(Context context) {
    // 1、通过proc文件获取当前进程名
    sCurrentProcess = SysUtils.getCurrentProcessName();
    // 2、获取当前进程pid
    sCurrentPid = Process.myPid();
    // 3、获取宿主程序包名
    sPackageName = context.getApplicationInfo().packageName;
    /**================================================
     * 4、判断是否使用“常驻进程”（见PERSISTENT_NAME）作为插件的管理进程
     *   1. 并设置常驻进程名称，默认常驻进程名称是以:GuardService结尾。也可使用当前进程名称。
     *   2. 可以通过宿主module下的build.gradle的repluginHostConfig{}中设置，很多参数参考宿主生
     *================================================*/
    if (HostConfigHelper.PERSISTENT_ENABLE) {
        //设置cppn名称为:GuardService
        String cppn = HostConfigHelper.PERSISTENT_NAME;
        if (!TextUtils.isEmpty(cppn)) {
            if (cppn.startsWith(":")) {
                //常驻进程名称为 包名:GuardService
                sPersistentProcessName = sPackageName + cppn;
            } else {
                sPersistentProcessName = cppn;
            }
        }
    } else {
        //如果不使用常驻进程管理插件，则使用当前进程名称
        sPersistentProcessName = sPackageName;
    }
    // 5、判断当前进程是否是主进程
    sIsUIProcess = sCurrentProcess.equals(sPackageName);
    // 6、判断当前线程是不是常驻进程
    sIsPersistentProcess = sCurrentProcess.equals(sPersistentProcessName);
}
```

5、为什么要标记出当前进程是不是UI进程，是不是常驻进程?

1. 每个进程在创建后，都会调用ActivityThread的main方法，开启消息循环，绑定AMS，创建Application，加载Provider，再回调Application生命周期方法
2. 不同进程根据这些标志，能进行对应的逻辑。

PMF.init()

6、PMF.init()源码分析

```
// PMF.java
public static final void init(Application application) {

    // 1、创建一个内部的主线程的Handler。便于后面post任务到主线程中。
    PluginManager.init(application);
    /**============================================================================
     * 2、Server端和Client端的初始化工作。
     *  1. 如果常驻进程作为插件管理进程。则常驻进程进行Server端的初始化。其他进程进行Client端的
     *  2. 如果宿主进程作为插件管理进程。宿主进程进行Server端的初始化。再将其他进程+宿主进程都作
     *============================================================================*/
    sPluginMgr = new PmBase(application);
    sPluginMgr.init();

    Factory.sPluginManager = PMF.getLocal();
    Factory2.sPLProxy = PMF.getInternal();

    PatchClassLoaderUtils.patch(application);
}
```

PmBase

1、PmBase的构造和init

```
// PmBase.java - 存储占坑Activity、Service、Providers的信息到HashSet中。存储服务端Binder对象在
private final HashSet<String> mContainerProviders = new HashSet<String>();  // 坑位Provider
private final HashSet<String> mContainerServices = new HashSet<String>();   // 坑位Services
private final HashSet<String> mContainerActivities = new HashSet<String>();      // 坑位Act
PmBase(Context context) {
    // 1、判断当前进程的类型
    if (PluginManager.sPluginProcessIndex == IPluginManager.PROCESS_UI || PluginManager.isP
        String suffix;
        if (PluginManager.sPluginProcessIndex == IPluginManager.PROCESS_UI) {
            // 1. 例如: "com.hao.repluginhost.loader.p.ProviderN1"
            suffix = "N1";
        } else {
            // 2. 例如: "com.hao.repluginhost.loader.p.Provider0"
            // 3. 例如: "com.hao.repluginhost.loader.p.Provider1"
            // 4. 例如: "com.hao.repluginhost.loader.p.Provider+后缀值"
            suffix = "" + PluginManager.sPluginProcessIndex;
        }
        // 2、存储“坑位Provider”的信息
        mContainerProviders.add(IPC.getPackageName() + ".loader.p.Provider" + suffix);
        // 3、存储“坑位Service”的信息
        mContainerServices.add(IPC.getPackageName() + ".loader.s.Service" + suffix);
    }

    /**================================================
     * 4、代表了当前Client进程
     *   1. 创建了【PluginServiceServer】 是Server端服务的提供方。
     *   2. 创建了【PluginContainers】: 用于插件容器管理，包括请求分配坑等操作
     *================================================*/
    mClient = new PluginProcessPer(context, this, PluginManager.sPluginProcessIndex, mConta

    // 5、负责宿主和插件、插件之间的胡同
    mLocal = new PluginCommImpl(context, this);

    // 6、Replugin框架中内部逻辑使用的很多方法都在这里，包括插件中通过“反射”调用的内部逻辑
    mInternal = new PluginLibraryInternalProxy(this);
}

// PmBase.java - 进行Server端或者Client端的初始化工作
void init() {
    // 1、“常驻进程”作为插件管理进程，则常驻进程作为Server，其余进程作为Client
    if (HostConfigHelper.PERSISTENT_ENABLE) {
        if (IPC.isPersistentProcess()) {
            // 1. Server端: 初始化“Server”所做工作
            initForServer();
        } else {
            // 2. Client端: 连接到Server
            initForClient();
        }
    }
    // 2、“UI进程”作为插件管理进程，则UI进程既可以作为Server也可以作为Client
    else {
        if (IPC.isUIProcess()) {
            // 1. Server端: 尝试初始化Server所做工作，
            initForServer();
```

```
                // 2. Client端：注册该进程信息到“插件管理进程”中。这里无需再做initForClient，因为不
                PMF.sPluginMgr.attach();
            } else {
                // 3. 其他进程，Client端：连接到Server
                initForClient();
            }
        }
    }
```

PluginProcessPer

PluginServiceServer

1、PluginServiceServer的源码

```
public PluginServiceServer(Context context) {
    mContext = context;
    mStub = new Stub();
}
class Stub extends IPluginServiceServer.Stub {
    @Override
    public ComponentName startService(Intent intent, Messenger client) throws RemoteException {
        return PluginServiceServer.this.startServiceLocked(intent, client);
    }
    @Override
    public int stopService(Intent intent, Messenger client) throws RemoteException {
        return PluginServiceServer.this.stopServiceLocked(intent);
    }
    @Override
    public int bindService(Intent intent, IServiceConnection conn, int flags, Messenger client)
        return PluginServiceServer.this.bindServiceLocked(intent, conn, flags, client);
    }
    @Override
    public boolean unbindService(IServiceConnection conn) throws RemoteException {
        return PluginServiceServer.this.unbindServiceLocked(conn);
    }
    @Override
    public String dump() throws RemoteException {
        return PluginServiceServer.this.dump();
    }
}
```

IPluginServiceServer.Stub

2、IPluginServiceServer.Stub

```
public abstract static class Stub extends Binder implements IPluginServiceServer {
    private static final String DESCRIPTOR = "com.qihoo360.replugin.component.service.serve
    static final int TRANSACTION_startService = 1;
    static final int TRANSACTION_stopService = 2;
    static final int TRANSACTION_bindService = 3;
    static final int TRANSACTION_unbindService = 4;
    static final int TRANSACTION_dump = 5;

    public Stub() {
        this.attachInterface(this, "com.qihoo360.replugin.component.service.server.IPluginS
    }

    public static IPluginServiceServer asInterface(IBinder obj) {
        IInterface iin = obj.queryLocalInterface("com.qihoo360.replugin.component.service.s
        return (IPluginServiceServer)(iin != null && iin instanceof IPluginServiceServer ?
    }

    public IBinder asBinder() {
        return this;
    }

    public boolean onTransact(int code, Parcel data, Parcel reply, int flags)  {
        // xxx
        switch(code) {
            case 1: reply.writeInt(1);
                    // xxx
        }
        // xxx
    }

    private static class Proxy implements IPluginServiceServer {
        private IBinder mRemote;

        Proxy(IBinder remote) {
            this.mRemote = remote;
        }

        public IBinder asBinder() {
            return this.mRemote;
        }

        public String getInterfaceDescriptor() {
            return "com.qihoo360.replugin.component.service.server.IPluginServiceServer";
        }

        public ComponentName startService(Intent intent, Messenger client) throws RemoteExc
            Parcel _data = Parcel.obtain();
            Parcel _reply = Parcel.obtain();
            ComponentName _result;
            // xxx
            _data.writeInt(1);
            intent.writeToParcel(_data, 0);
            client.writeToParcel(_data, 0);
            // xxx
```

```
                this.mRemote.transact(1, _data, _reply, 0);
                // xxx
                _result = (ComponentName)ComponentName.CREATOR.createFromParcel(_reply);
                return _result;
            }

            public int stopService(Intent intent, Messenger client) throws RemoteException {
                // xxx
                return _result;
            }
        }
    }
```

PluginContainers

问题汇总

参考资料