GreedyPorter(GPT)技术文档

刘海涛

liuhaitao@baidu.com

目录

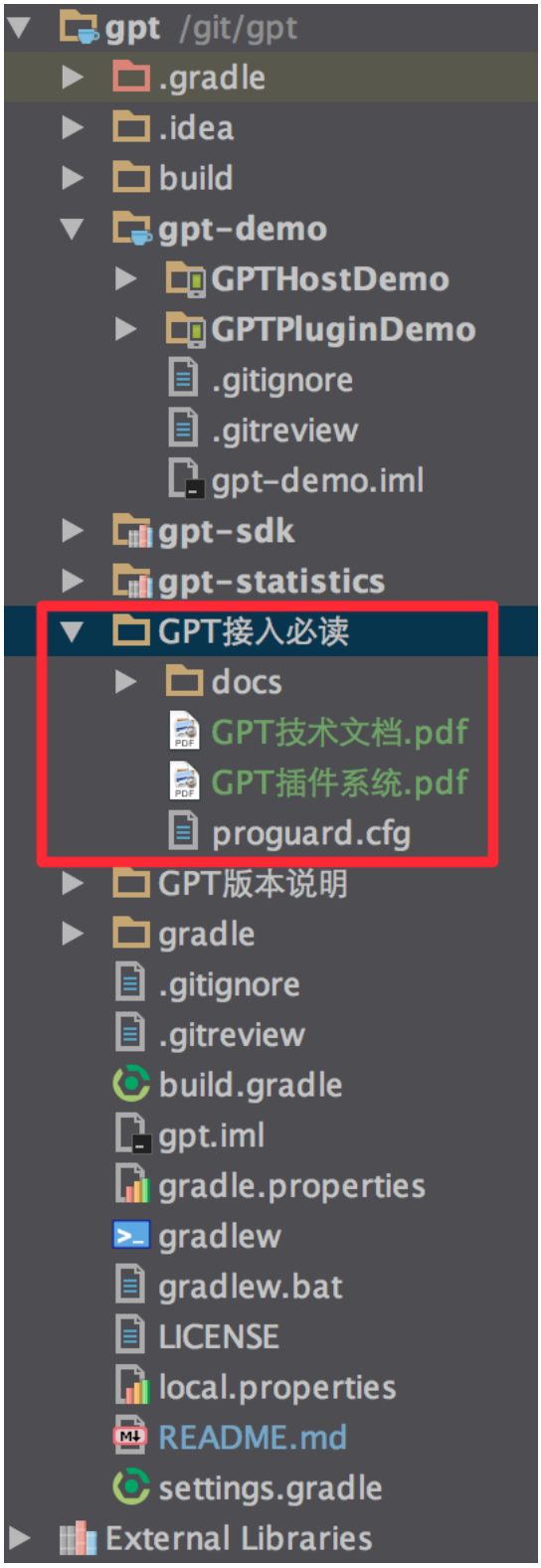
一．GreedyPorter(GPT)插件系统简介

- GPT 插件系统是借鉴 OSGI，AOP 等技术实现的一个 Android 平台重量级插件系统。

- 目前已接入的百度产品包括:百度手机助手,百度网盘,百度卫士,度秘,拾相,91 助手,安卓市场等。

- 目前仅百度手机助手已成功接入过的插件数目 50+。

- 更多内容可关注 http://blog.csdn.net/dffd001 和 https://www.jianshu.com/u/2306ba8f1c59 后续文章更新。或参考源码工程"GPT 接入必读"和相关代码注释说明。


gpt /git/gpt
.gradle
.idea
build
gpt-demo
GPTHostDemo
GPTPluginDemo
.gitignore
.gitreview
gpt-demo.iml
gpt-sdk
gpt-statistics
GPT接入必读
docs
GPT技术文档.pdf
GPT插件系统.pdf
proguard.cfg
GPT版本说明
gradle
.gitignore
.gitreview
build.gradle
gpt.iml
gradle.properties
gradlew
gradlew.bat
LICENSE
local.properties
README.md
settings.gradle
External Libraries

二. GPT 主要特性

- 基于 GPT 的插件开发比较简单，就是一个普通的 APK。

- 插件开发基于标准 Android API，无需重新学习。

- 插件可以 APK 独立运行、方便调试测试，也可以插件形式运行、扩展宿主功能。

- 共用一套代码，无需单独开发维护多套代码，减少开发维护成本。

- 支持 Android 四大组件。

- 支持 Intent 等标准调起方法。

- 支持数据库，Preference 等数据存储。

- 支持未写死路径的第三方 Jar 包通用库。

- 支持多种插件和宿主的交互形式(取决于实际产品需求)。

- 插件默认独立进程安装,减少对主进程的影响。

- 插件默认运行在独立进程减少对主进程的影响，也可在插件的 manifest 中简单声明<meta-data android:name="gpt_union_process" android:value="true"/>以便插件和宿主运行在同进程。

```
<application
    android:name=".MyApplication"
    android:icon="@drawable/ic_launcher"
    android:label="GPTPluginDemo">

    <!--TODO value:true 和宿主运行在同进程,value:false 单独运行在gpt插件进程-->
    <meta-data
        android:name="gpt_union_process"
        android:value="true" />
```

- 插件运行时每个插件的 classloader 互相独立，避免类冲突和类兼容性问题;同时更可以根据需求在插件编包时依赖宿主接口或公告库,而在实际出包时简单配置排除,进而保证插件可以复用宿主的最新接口功能和公共库,极大的减小程序包大小和保证核心功能的宿主一致性。

三．插件开发

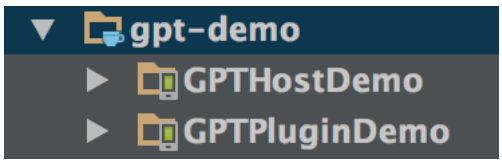
- 基于 GPT 的插件开发比较简单，就是一个普通的 apk，在排除特殊限制的情况下可以独立运行也可以插件方式运行，代码可

以完全共用一套。

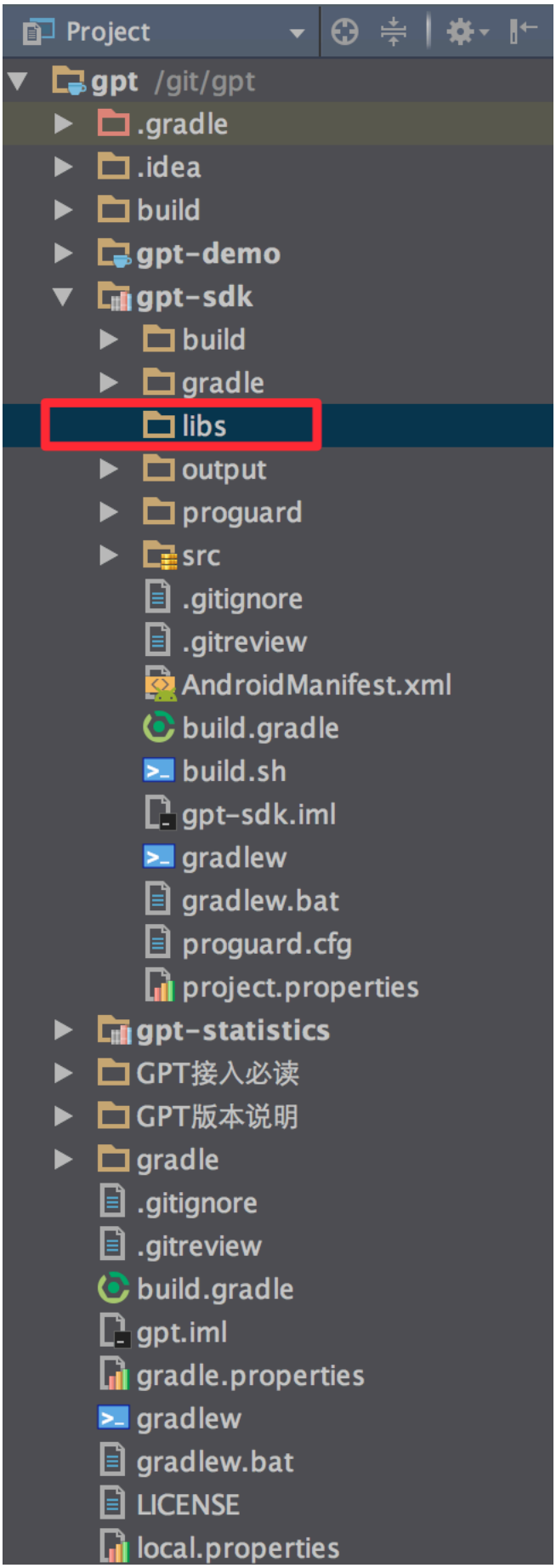
- minSDK 最低支持 8，建议在 10 以上开发。

- 插件主体开发过程就是一个普通的 Android App 开发,极大地减少插件开发二次学习成本并有效降低插件和独立 App 的代码复用与问题修改同步成本。

- 具体可参考本工程附带的完整 Demo 并根据产品实际需求详细阅读并开发修改 TODO 和注释逻辑内容。



- 注意:本开源工程上传代码时因"gpt-sdk"下的"libs"目录并无实际文件而 git 无法单独上传空文件夹,所以下载完本工程代码后只需要在"gpt-sdk"下新建一空的"libs"文件夹后即可编译运行不同目标工程。

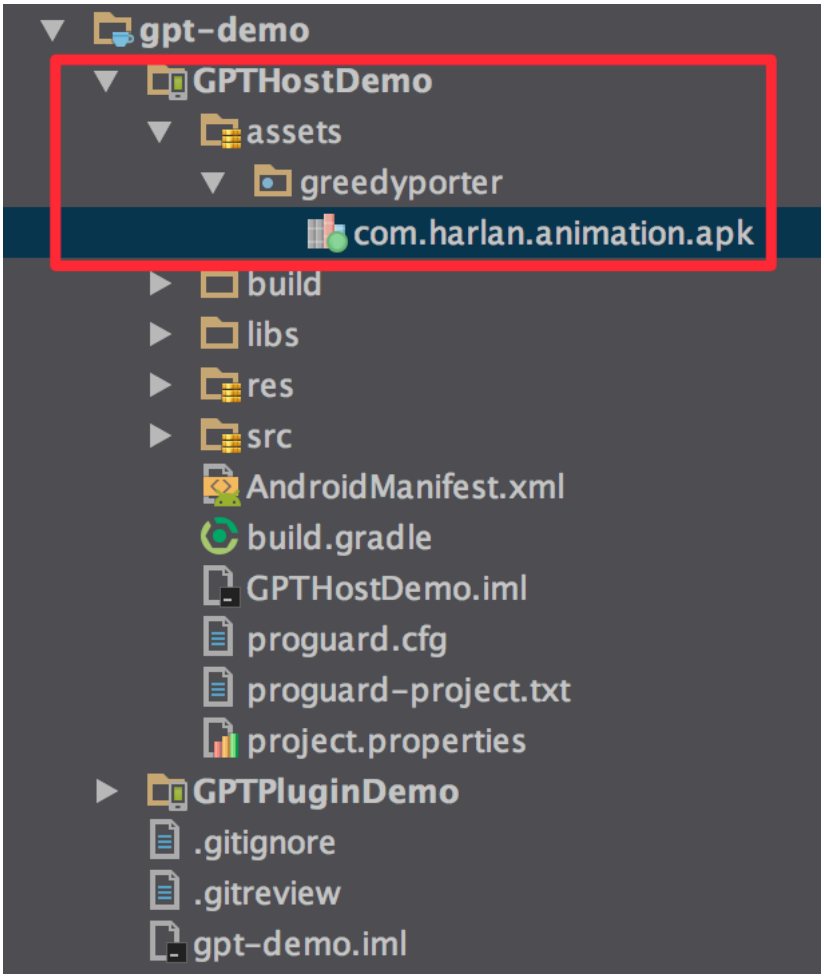

Project
gpt /git/gpt
.gradle
.idea
build
gpt-demo
gpt-sdk
build
gradle
libs
output
proguard
src
.gitignore
.gitreview
AndroidManifest.xml
build.gradle
build.sh
gpt-sdk.iml
gradlew
gradlew.bat
proguard.cfg
project.properties
gpt-statistics
GPT接入必读
GPT版本说明
gradle
.gitignore
.gitreview
build.gradle
gpt.iml
gradle.properties
gradlew
gradlew.bat
LICENSE
local.properties

- "gpt-demo"下的"GPTHostDemo"为宿主接入 GPT 插件系统的可运行 Demo,同时提供了插件系统相关功能调试测试等附属功能,可根据实际需求增改。

- "gpt-demo"下的"GPTPluginDemo"为实际插件的可运行 Demo,同时提供了插件相关功能调试测试等附属功能,并可根据实际需求增改。

- 为方便用户使用 Demo,用户可直接运行安装"GPTHostDemo",本工程已默认内置安装对应的"GPTPluginDemo"以方便用户操作测试。

- 用户也可自行修改"GPTPluginDemo"运行出包后替换"GPTHostDemo"->"assets"下内置插件以包名命名的"com.harlan.animation.apk",内置插件需要以包名命名并放置在"assets"下。



- 内置插件的安装需要调用如下方法,可根据产品实际需求在 Application 的 GPT 初始化设置后或具体 ActivityUI 界面显示点击后调用如下方法以执行内置插件安装。

```
// 为方便 HostDemo 直接体验插件功能,默认先安装内置在 assets/greedyporter 目录下的内置插件 APK(),
// 内置 APK 必须以插件的 packageName 命名，比如 com.harlan.animation.apk
// TODO 如不需要可删除对应路径、插件文件和下面这句代码。
GPTPackageManager.getInstance(getActivity()).installBuildinApps();
```

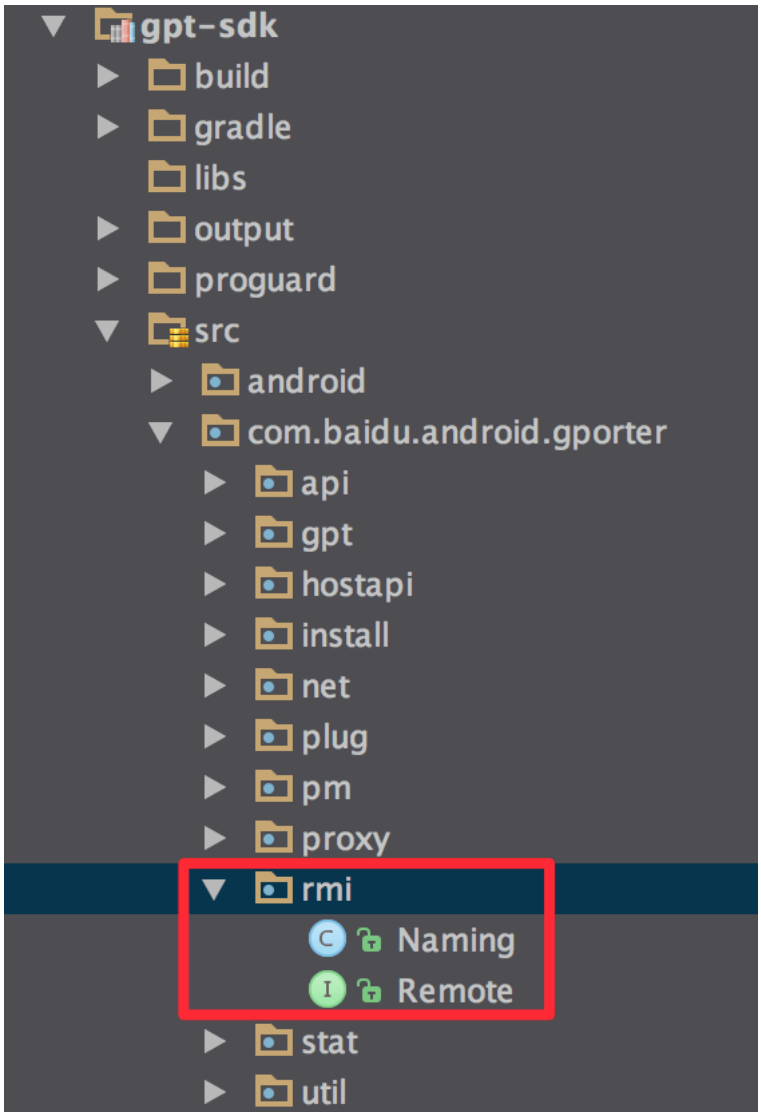
- 用户也可以自行开发插件并随意命名成 xxx.apk 后放在手机的"/sdcard/baidu_plugin_test/"路径下并点击"GPTPluginDemo"的"扫描加载插件"功能进行新插件的独立安装和运行。

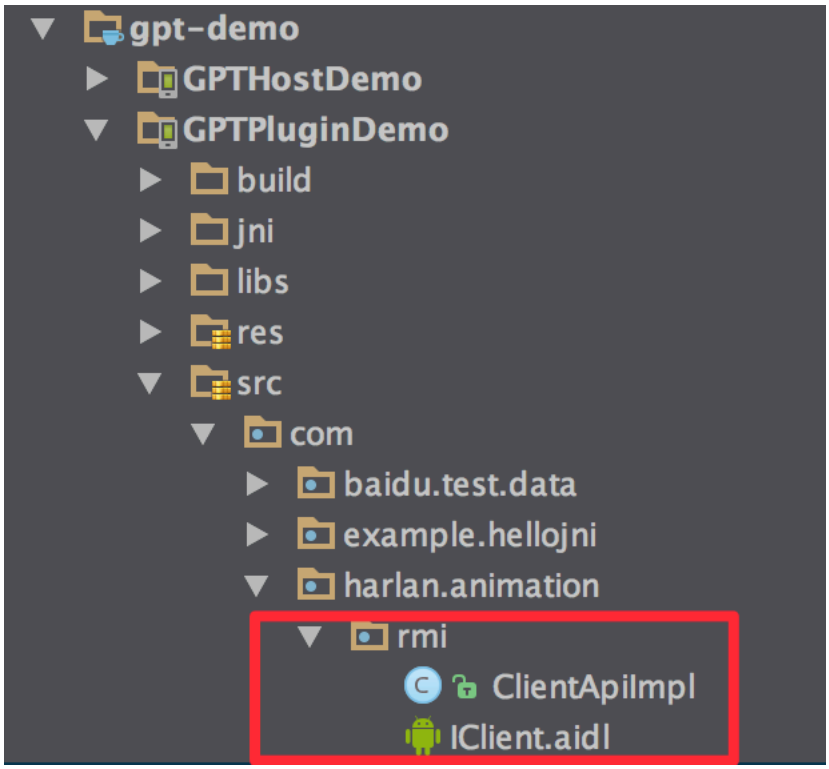
```
dffd001V-MacPro:/ liuhaitao$ adb shell
shell@fortunalte:/ $ cd sdcard
shell@fortunalte:/sdcard $ mkdir baidu_plugin_test
shell@fortunalte:/sdcard $ exit
dffd001V-MacPro:/ liuhaitao$ adb push /Users/baidu/Downloads/com.baidu.appsearch.batterymanager.apk /sdcard/baidu_p
lugin_test/com.baidu.appsearch.batterymanager.apk
[100%] /sdcard/baidu_plugin_test/com.baidu.appsearch.batterymanager.apk
dffd001V-MacPro:/ liuhaitao$ adb shell
shell@fortunalte:/ $ cd sdcard/baidu_plugin_test
shell@fortunalte:/sdcard/baidu_plugin_test $ ls
com.baidu.appsearch.batterymanager.apk
shell@fortunalte:/sdcard/baidu_plugin_test $
```

四．插件主要接口

- 为了解决插件在独立进程中运行时，需要和主进程进行通信的需要，提供了跨进程通信的简单接口,具体功能需求时可兼容测试并扩展开发。

- 插件独立进程和主进程接口调用接口：com.baidu.android.gporter.rmi.Nameing，该接口是基于 aidl 方式通信， 具体可参考 javadoc 和代码注释说明以及"gpt-demo"中的"GPTHostDemo"和"GPTPluginDemo"对应使用实例。


gpt-sdk
build
gradle
libs
output
proguard
src
android
com.baidu.android.gporter
api
gpt
hostapi
install
net
plug
pm
proxy
rmi
Naming
Remote
stat
util

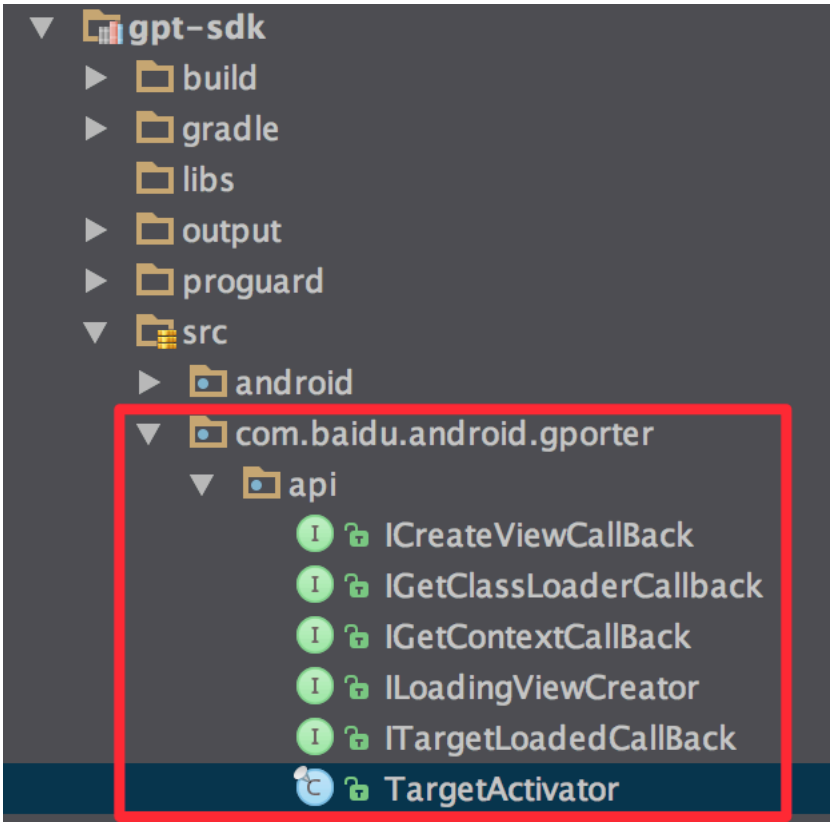


- 插件默认运行在独立进程，如果插件想和主程序运行在一个进程，需要在插件的 manifest 中设置如下：

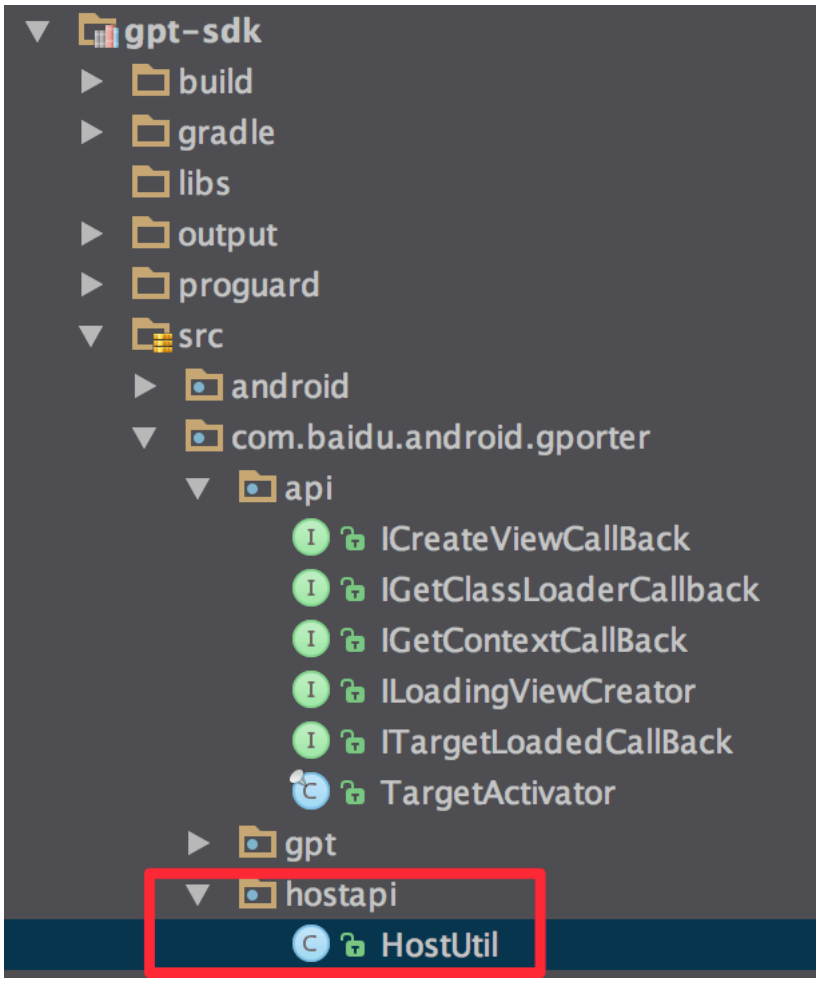
```
<application
    android:name=".MyApplication"
    android:icon="@drawable/ic_launcher"
    android:label="GPTPluginDemo">

    <!--TODO value:true 和宿主运行在同进程,value:false 单独运行在gpt插件进程-->
    <meta-data
        android:name="gpt_union_process"
        android:value="true" />
```

- 插件安装、启动等主要 API 和自定义监听回调可参看"gpt-sdk"中"com.baidu.android.gporter.api"下的"TargetActivator"相关类和接口方法说明。

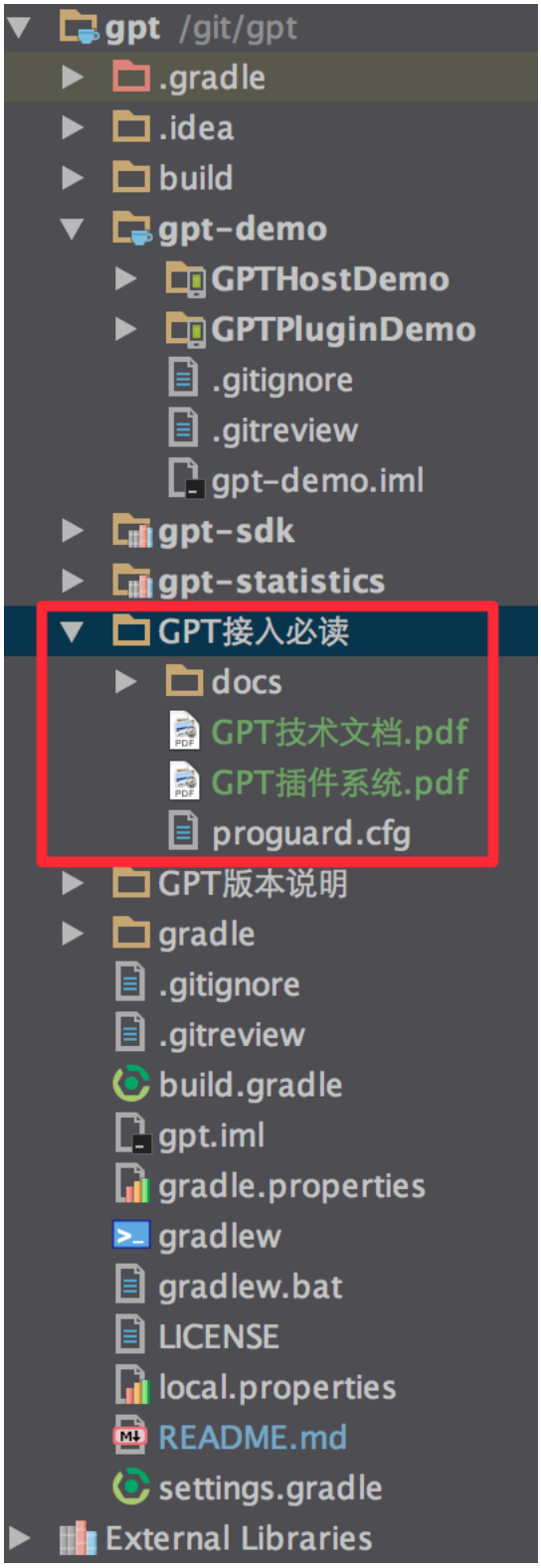


- 插件如需获取宿主的相关方法可参看"gpt-sdk"中"com.baidu.android.gporter.hostapi"下的"HostUtil"相关类和接口方法说明。



- 主要API相关类和接口方法说明也可用浏览器直接参看"GPT接入必读"中 docs 文件中的"index"和"index-all"等文档说明或代码注释。

- 更多详细文档可参考"GPT 接入必读"和代码注释说明。


gpt /git/gpt
.gradle
.idea
build
gpt-demo
GPTHostDemo
GPTPluginDemo
.gitignore
.gitreview
gpt-demo.iml
gpt-sdk
gpt-statistics
GPT接入必读
docs
GPT技术文档.pdf
GPT插件系统.pdf
proguard.cfg
GPT版本说明
gradle
.gitignore
.gitreview
build.gradle
gpt.iml
gradle.properties
gradlew
gradlew.bat
LICENSE
local.properties
README.md
settings.gradle
External Libraries

五．插件主要限制

- overridePendingTransition 建议使用系统或宿主的

原因：此方法仅传给 AMS 两个 int 值，且由 AMS 层进行资源解析并实现动画效果，根本到不了客户端。

方案：支持系统或宿主动画资源，可将动画资源“预埋在主程序”并利用 public.xml 确保其 ID 固定，通过主程序动画 ID 传递给系统，实现相应效果。

- Notification 的资源建议透传或使用系统、宿主的

原因：RemoteViews 性质决定。

方案：支持图片资源以及文案等通过 Drawable 或 String 透传。

方案：也可在主程序添加相关资源，并共用主程序或系统资源。

- 插件不要静态写死数据路径

原因：插件数据路径和权限是依赖于宿主而生成的。

方案：使用标准方法 context.getFilesDir()或 getExternalFilesDir()等。

- 插件权限需宿主内声明

原因：权限是系统安装宿主程序时读取其 AndroidManifest.xml 中权限部分，并放入系统的 package.xml 中，除系统核心应用和 Root 外，外界无法修改。

此外：抛开技术，也建议插件权限走“宿主申请审核控制”，以保证宿主对插件权限的安全控制。

方案：插件接入联调时检查权限并在宿主中声明即可。

- 由于插件安装过程需要进行 dex 重新生成，需要较大的内存，所以插件 Manifest 中的 Component 建议不要过多，否则有可能出现 OOM。

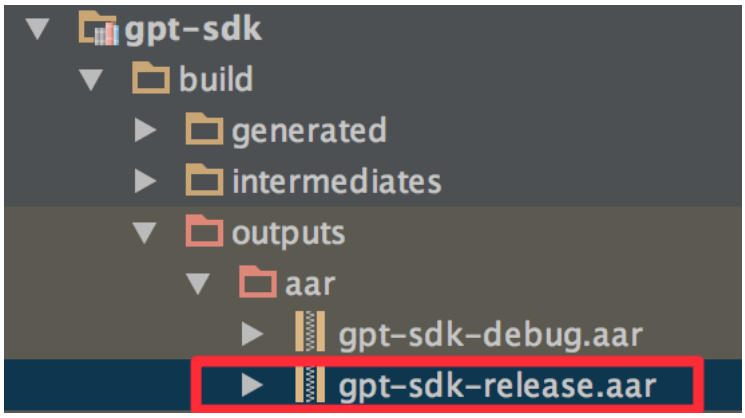
- 多个插件运行在同一个进程时，需要注意内存占用问题。

- GPT 会把静态 Broadcast 转为动态方式进行支持。

- AccountManager、GMS 接口、Class.getInputStream()等暂不支持。

六．Host 接入 GPT

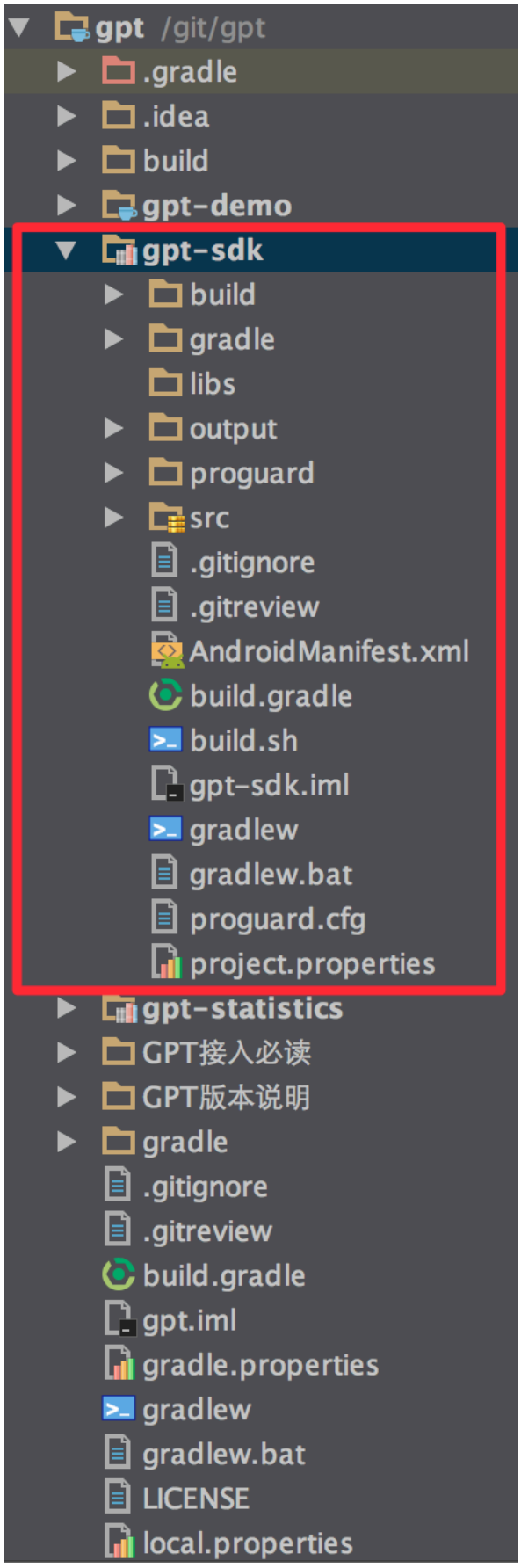
- 方法一:直接运行"GPTHostDemo"工程,并把"gpt-sdk"->"build"->"outputs"->"aar"路径下的"gpt-sdk-release.aar"加到对应项目工程里并添加对应依赖。



- 方法二:直接把"gpt-sdk"的 Module 引入到对应宿主工程模块中,并在对应项目的 build.gradle 文件中，添加如下依赖。

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // TODO 非源码引入Release版AAR方法,可查看"GPT接入必读"等相关说明。
    compile project(':gpt-sdk')
    // TODO GPT统计上报依赖,不影响GPT插件主体功能;外部使用可注掉,
    // 或参照HostApplication对应方法实现并设置GPT对应统计数据接口自定义处理上报策略即可。
    // compile project(':gpt-statistics')
}
```

- 方法三:根据实际产品需求直接拷贝"gpt-sdk"的工程源码到对应项目工程中。


gpt /git/gpt
.gradle
.idea
build
gpt-demo
gpt-sdk
build
gradle
libs
output
proguard
src
.gitignore
.gitreview
AndroidManifest.xml
build.gradle
build.sh
gpt-sdk.iml
gradlew
gradlew.bat
proguard.cfg
project.properties
gpt-statistics
GPT接入必读
GPT版本说明
gradle
.gitignore
.gitreview
build.gradle
gpt.iml
gradle.properties
gradlew
gradlew.bat
LICENSE
local.properties

七．插件校验

- 插件校验和 Android 系统比较相似，默认采用签名一致的校验方式;主程序需添加对应实现类声明并可自定义控制校验过程如下所示。

- 主程序写一个类继承自 com.baidu.android.gporter.pm.ISignatureVerify 并实现对应 checkSignature 方法,效验成功返回 true 后才能进行后续安装过程。

```
package com.baidu.gpt.hostdemo;

import android.content.pm.Signature;
import android.util.Log;
import com.baidu.android.gporter.pm.ISignatureVerify;

/**
 * SignatureVerifier
 * 插件的签名校验宿主自定义策略实现类
 * TODO 策略实现后同时在宿主 AndroidManifest.xml 中的 application 标签中添加如下对应 meta-data,其中 name 不变、value 为本类全路径名即可(可直接参考本 Demo 对应添加方法)
 * <meta-data android:name="com.baidu.android.gporter.signatureverify.class"
android:value="com.baidu.gpt.hostdemo.SignatureVerifier" />
 *
 * @author liuhaitao
 * @since 2014-07-16
 */
public class SignatureVerifier implements ISignatureVerify {
    /**
     * DEBUG 开关
     */
    public static final boolean DEBUG = true & MainActivity.DEBUG;

    /**
     * TAG
     */
    public static final String TAG = "SignatureVerifier";
```

```
    @Override
    public boolean checkSignature(String packageName, boolean isReplace, Signature[] signatures, Signature[]
newSignatures) {
        if (DEBUG) {
            Log.d(TAG, "checkSignature(String packageName, boolean isReplace, Signature[] signatures, Signature[]
newSignatures):\n"
                    + "packageName=" + packageName + "; isReplace=" + isReplace + "; signatures=" + signatures + ";
newSignatures=" + newSignatures);
        }
        // 自定义签名策略验证合法时需要返回 true,否则返回 false。
        return true;
    }

}
```

- 然后在主程序的 AndroidManifest 中的 application 标签中添加如下配置声明使用对应的校验类即可。

```
<application
    android:name="com.baidu.gpt.hostdemo.HostApplication"
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="GPTHostDemo"
    android:theme="@style/AppTheme">

    <!--插件签名校验宿主自定义策略实现类,具体说明可参考SignatureVerifier类里的详细注释。-->
    <meta-data
        android:name="com.baidu.android.gporter.signatureverify.class"
        android:value="com.baidu.gpt.hostdemo.SignatureVerifier" />
```

八. 混淆配置

- Host 的混淆 proguard 配置可参考"GPT 接入必读"中的 proguard.cfg,为方便定位问题代码行号等,也可选择保持混淆后的行号或不混淆。

九．安装插件

- 如下所示直接传入对应插件 APK 的文件路径即可,安装成功失败会有广播通知,具体可参考"GPTHostDemo"并查看"com.baidu.android.gporter.pm.GPTPackageManager"类的相关方法和参数说明。

```
// "扫描加载插件"显示点击处理
View scanLoadPlugin = mHeaderView.findViewById(R.id.scan_load_plugin);
scanLoadPlugin.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        if (isInstalling) {
            return;
        }

        isInstalling = true;

        File pluginScanLoadDir = new File(Environment.getExternalStorageDirectory(), PLUGIN_SCAN_PATH);
        if (DEBUG) {
            Log.d(TAG, "onCreate(Bundle savedInstanceState): pluginScanLoadDir="
                    + pluginScanLoadDir.getAbsolutePath());
        }

        new AsyncTask<File, Integer, Void>() {

            @Override
            protected Void doInBackground(File... params) {

                if (params[0] != null && params[0].isDirectory() && params[0].exists()) {
                    String[] files = params[0].list();
                    if (files != null) {
                        String filePath = "";
                        for (String file : files) {
                            filePath = params[0].getAbsolutePath() + File.separator + file;
                            GPTPackageManager.getInstance(getApplicationContext()).installApkFile(filePath);
                            if (DEBUG) {
                                Log.d(TAG, "AsyncTask doInBackground(File... params): "
                                        +
"GPTPackageManager.getInstance(getApplicationContext()).installApkFile(filePath): "
```

```
                                                + "\n filePath=" + filePath);
                                    }
                                }
                            }
                        }

                        return null;
                    }

                    protected void onPostExecute(Void result) {
                        isInstalling = false;
                    }
                }.execute(pluginScanLoadDir);

            }
});
```

十．启动插件

- 支持插件包名和插件 Intent 组件,以及插件加载动画自定义和静默加载等多种不同形式的插件启动启动方法,具体可查看插件调用"com.baidu.android.gporter.api.TargetActivator"类的相关方法和参数说明。

```
@Override
public void onListItemClick(ListView l, final View v, int position, long id) {
    Map<String, Object> item = data.get(position);
    String packageName = (String) item.get("packagename");
    Intent intent = new Intent();
    intent.setComponent(new ComponentName(packageName, ""));

    // 启动插件的方法
    TargetActivator.loadTargetAndRun(getActivity(), intent);
}
```

```
/**
 * 加载并启动插件。
 *
```

```
     * @param context host 的 Activity
     * @param intent    目标 intent，可以为 activity， service， broadcast
     */
    public static void loadTargetAndRun(final Context context, final Intent intent) {
        loadTargetAndRun(context, intent, false);
    }

    /**
     * 加载并启动插件
     *
     * @param context host 的 Activity
     * @param intent    目标 Intent
     * @param creator loading 界面创建器
     */
    public static void loadTargetAndRun(final Context context, final Intent intent, ILoadingViewCreator creator) {
        ProxyEnvironment.putLoadingViewCreator(intent.getComponent().getPackageName(), creator);
        loadTargetAndRun(context, intent, false);
    }

    /**
     * 加载并启动插件
     *
     * @param context     host 的 Activity
     * @param intent      目标 Intent
     * @param isSilence  是否是静默加载插件
     */
    public static void loadTargetAndRun(final Context context, final Intent intent, boolean isSilence) {
        Context hostContext = Util.getHostContext(context); // 有可能从插件中调过来的，这时候获取到 host context
        ProxyEnvironment.enterProxy(hostContext, intent, isSilence, false);
    }

    /**
     * 加载并启动插件
     *
     * @param context           host 的 Activity
     * @param componentName  目标 Component
     */
    public static void loadTargetAndRun(final Context context, final ComponentName componentName) {
        Intent intent = new Intent();
        intent.setComponent(componentName);
        loadTargetAndRun(context, intent);
    }

    /**
```

```
     * 加载并启动插件
     *
     * @param context           host 的 Activity
     * @param componentName  目标 Component
     * @param creator           loading 界面创建器
     */
    public static void loadTargetAndRun(final Context context, final ComponentName componentName,
                                         ILoadingViewCreator creator) {
        ProxyEnvironment.putLoadingViewCreator(componentName.getPackageName(), creator);
        loadTargetAndRun(context, componentName);
    }

    /**
     * 加载并启动插件，  启动插件的默认 launcher activity
     *
     * @param context        host 的 application context
     * @param packageName  插件包名
     */
    public static void loadTargetAndRun(final Context context, String packageName) {
        loadTargetAndRun(context, new ComponentName(packageName, ""));
    }

    /**
     * 加载并启动插件
     *
     * @param context        host 的 application context
     * @param packageName  插件包名
     * @param creator        插件 loading 界面的创建器
     */
    public static void loadTargetAndRun(final Context context, String packageName, ILoadingViewCreator creator) {
        ProxyEnvironment.putLoadingViewCreator(packageName, creator);
        loadTargetAndRun(context, new ComponentName(packageName, ""));
    }

    /**
     * 静默加载插件，异步加载
     *
     * @param context        application Context
     * @param packageName  插件包名
     */
    public static void loadTarget(final Context context, String packageName) {
        Intent intent = new Intent();
        intent.setComponent(new ComponentName(packageName,
ProxyEnvironment.LOADTARGET_STUB_TARGET_CLASS));
```

```
        loadTargetAndRun(context, intent, true);
    }

    /**
     * 静默加载插件，异步加载，可以设置 callback
     *
     * @param context        application Context
     * @param packageName  插件包名
     * @param callback       加载成功的回调
     */
    public static void loadTarget(final Context context, final String packageName,
                                  final ITargetLoadedCallBack callback) {

        Context hostContext = Util.getHostContext(context); // 有可能从插件中调过来的，这时候获取到 host context

        // 插件已经加载
        if (ProxyEnvironment.isEnterProxy(packageName)) {
            if (callback != null) {
                callback.onTargetLoaded(packageName, true);
            }
            return;
        }

        if (callback == null) {
            loadTarget(hostContext, packageName);
            return;
        }
        BroadcastReceiver recv = new BroadcastReceiver() {
            public void onReceive(Context ctx, Intent intent) {

                String curPkg = intent.getStringExtra(ProxyEnvironment.EXTRA_TARGET_PACKAGNAME);

                if (ProxyEnvironment.ACTION_TARGET_LOADED.equals(intent.getAction())
                        && TextUtils.equals(packageName, curPkg)) {
                    boolean isSucc = intent.getBooleanExtra(ProxyEnvironment.EXTRA_TARGET_LOADED_RESULT,
false);
                    callback.onTargetLoaded(packageName, isSucc);
                    try {
                        ctx.unregisterReceiver(this);
                    } catch (RuntimeException e) {
                        // 某些 2.3 手机上会 crash，暂时先捕获一下
                        if (DEBUG) {
                            e.printStackTrace();
                        }
```

```
                }
            }
        }
    };
    IntentFilter filter = new IntentFilter();
    filter.addAction(ProxyEnvironment.ACTION_TARGET_LOADED);
    hostContext.getApplicationContext().registerReceiver(recv, filter);

    Intent intent = new Intent();
    intent.setAction(ProxyEnvironment.ACTION_TARGET_LOADED);
    intent.setComponent(new ComponentName(packageName, recv.getClass().getName()));
    ProxyEnvironment.enterProxy(hostContext, intent, true, false);
}
```

十一．插件通信

- 大部分插件通信都可以通过标准方法和指定插件包名,组件类名的 Intent 参数完成。例如:

```
/**
 * 打开换机精灵插件通讯录整理页面
 */
public static void openHuanjiTidyShowActivity(Context context) {
    Map<String, PlugInAppInfo> pluginAppMap = PluginAppManager.getInstance(context).getPlugAppMap();
    if (pluginAppMap == null || !pluginAppMap.containsKey("com.cx.huanjisdk")) {
        return;
```

```
        }
        PlugInAppInfo appInfo = pluginAppMap.get("com.cx.huanjisdk");
        Intent intent = new Intent();
        ComponentName localComponentName = new ComponentName("com.cx.huanjisdk", "com.cx.huanjisdk.tidy.contacts.TidyShowActivity);
        intent.setComponent(localComponentName);
        intent.setAction("android.intent.action.MAIN");
        PluginAppManager.getInstance(context).launchApp(appInfo, intent.toURI());
    }
```

- startActivityForResult 等特殊需求使用前可先调用 TargetActivator.remapActivityIntent(mContext, intent)。例如:

```
    TargetActivator.loadTarget(context, appInfo.getPkgName(), new ITargetLoadedCallBack() {

        @Override
        public void onTargetLoaded(String packageName, boolean isSucc) {
```

```
                try {
                    if (isSucc) {
                        if (DEBUG) {
                            Log.d(TAG, packageName + " is l
oaded");
                        }
                        TargetActivator.remapActivityIntent(mCo
ntext, intent);
                        ((Activity) context).startActivityForResul
t(intent, requestCode);
                    } else {
                        Toast.makeText(mContext, R.string.plugi
n_load_fail, Toast.LENGTH_LONG).show();
                    }
                } catch (Exception e) {
                    Toast.makeText(mContext, R.string.plugin_lo
ad_fail, Toast.LENGTH_LONG).show();
                }

            }
        });
```

- 对于和宿主同进程的插件可以通过简单引入公开功能接口的 Jar 或第三方公共 Jar 引入保证插件编写通过,而在实际运行时排除对应 Jar 包来共享宿主公共库代码或匹配宿主实际功能实现。

- 简单的跨进程通信也可以通过 BroadcastReceiver 进行 Action 自定义协议处理,更可以通过宿主直接调用插件对应包名和组件类名直接启动对应组件。

- 插件独立进程和主进程的调用可参考接口 com.baidu.android.gporter.rmi.Nameing 并根据实际产品需求 Log 兼容测试，该接口是基于 aidl 方式通信。

- 以换机助手为例:插件定义需要开放给 HOST 主程序调用的接口，定义 AIDL 文件，放到相应 src 目录下，会在 gen 的相应目录生成对应的.java 文件，例如：PhoneCheckedRemote.aidl

```
package  com.cx.tools.remote;

import  com.cx.tools.remote.IPhoneCheckedCallback;

interface  PhoneCheckedRemote{
```

```
    void  startChecked();
    void  registerCallback(IPhoneCheckedCallback  callBack);
    void  unRgisterCallback(IPhoneCheckedCallback  callBack);
}
```

- 插件定义需要回调 HOST 主程序的接口，定义 AIDL 文件，例如：IPhoneCheckedCallback.aidl

```
package  com.cx.tools.remote;

interface  IPhoneCheckedCallback{
    void  notifyToUI(int  score);
}
```

- 插件实现 GPT 的 Remote 接口，HOST 主程序才能获得 IBinder；Binder 实现例如 PhoneCheckedRemote.Stub。

如果从 HOST 主程序传入了回调接口，则可以使用 RemoteCallbackList 回调 HOST 主程序传入的回调接口 PhoneCheckedRemoteImpl.java。

```
package  com.cx.tools.remote;

import  android.os.IBinder;
```

```
import android.os.RemoteCallbackList;
import android.os.RemoteException;

import com.baidu.android.gporter.rmi.Remote;
import com.cx.base.CXApplication;
import com.cx.tools.check.IPhoneInfoListener;
import com.cx.tools.check.PhoneInfoChecked;
public class PhoneCheckedRemoteImpl implements Remote, IPhoneInfoListener {
    private RemoteCallbackList<IPhoneCheckedCallback> callbackList = new RemoteCallbackList<IPhoneCheckedCallback>();

    public PhoneCheckedRemoteImpl() {

    }

    @Override
    public IBinder getIBinder() {
        return mBinder;
    }
```

```
        private final PhoneCheckedRemote.Stub mBinder = new
PhoneCheckedRemote.Stub() {

            @Override
            public void unRgisterCallback(IPhoneCheckedCallback
 callBack) throws RemoteException {
                if (callBack != null) {
                    callbackList.unregister(callBack);
                }
            }

            @Override
            public void registerCallback(IPhoneCheckedCallback c
allBack) throws RemoteException {
                if (callBack != null) {
                    callbackList.register(callBack);
                }
            }
```

```
        @Override
        public void startChecked() throws RemoteException {
            new PhoneInfoChecked(CXApplication.mAppContext, PhoneCheckedRemoteImpl.this).startChecked();
        }
    };

    @Override
    public void notifyToUI(int score) {
        int count = callbackList.beginBroadcast();
        for (int i = 0; i < count; i++) {
            try {
                callbackList.getBroadcastItem(i).notifyToUI(score);
            } catch (Exception e) {
                e.printStackTrace();
            }

        }
        callbackList.finishBroadcast();
```

```
        }
    }
```

- HOST 主程序将插件定义需要开放出来的接口 AIDL 文件和插件定义需要回调 HOST 主程序的接口 AIDL 文件，放到相应 src 目录下，会在 gen 的相应目录生成对应的.java 文件，例如 PhoneCheckedRemote.aidl 和 IPhoneCheckedCallback.aidl

- HOST 主程序实现插件定义的需要回调 HOST 主程序的接口，例如

```
    public IPhoneCheckedCallbackImpl mIPhoneCheckedCallback = new IPhoneCheckedCallbackImpl();

    public class IPhoneCheckedCallbackImpl extends IPhoneCheckedCallback.Stub {

        @Override
        public void notifyToUI(int score) throws RemoteException {
            Log.d(TAG, "IPhoneCheckedCallbackImpl:notifyToUI:s
```

core=" + score);

}

}

- HOST 主程序 LOAD 插件并调用插件接口，传入需要插件回调的接口，例如

TargetActivator.loadTarget(mContext, "com.cx.huanjisdk", new ITargetLoadedCallBack() {

@Override

public void onTargetLoaded(String packageName) {

System.out.println(packageName + "is onTargetLoaded.");

IBinder binderPhoneCheck = Naming.lookupPlugin("com.cx.huanjisdk",

"com.cx.tools.remote.PhoneCheckedRemoteImpl");

PhoneCheckedRemote clientPhoneCheck = P

```
honeCheckedRemote.Stub.asInterface(binderPhoneCheck);


                    if (clientPhoneCheck != null) {
                        try {
                            clientPhoneCheck.registerCallback
(mIPhoneCheckedCallback);

                            clientPhoneCheck.startChecked();
                        } catch (RemoteException e) {
                            e.printStackTrace();
                        }
                    }

                }
            });
```

十二．常见问题

1.宿主和插件在 64 位设备上加载 so 问题

- Android 的 64 位和 32 位运行分析可参考下述相关文章:

http://blog.csdn.net/dffd001/article/details/79265028 或

https://www.jianshu.com/p/393f806f1348

- 在 64 位设备上，对于插件系统有一定的影响，主要是安装和加载。

- 注意:为了有效识别宿主和插件,宿主工程需要在"libs"目录下包含至少 1 个对应设备类型的 so 文件。

- 插件无法安装，插件系统报 cpuabi 不一致无法安装也是由于上述原因导致，比如 Host 没有 so，插件只有 armabi 32 位的 so，此时如果运行在 64 位设备上，则插件无法安装。

- 64 位设备上运行策略如下:

- 如果 APK 存在 lib/arm64-v8a,也存在 lib/armabi，则系统运行主程序是则按照 64 位程序运行;

因为主程序存在 64 位代码则此时加载插件也需要 64 位代码，插件中必须包含 lib/arm64-v8a 的 so，否则无法安装也无法运行。

- 如果 APK 中没有 so 目录，则系统按照默认配置 64 位加载主程序，此时按照上一条原则插件必须也是 64 位的。

- 如果 APK 存在 lib/armabi 目录的 so，则系统以 32 位兼容方式加载主程序，此时运行插件也跟主程序一样以 32 位兼容方式运行，所以此时插件中必须包含 armabi 32 位 so 目录。

- Host 的混淆 proguard 配置可参考"GPT 接入必读"中的 proguard.cfg,为方便定位问题代码行号等,也可选择保持混淆后的行号或不混淆。