**Author:** “Bobby Wang [wbo4958@gmail.com](mailto:wbo4958@gmail.com)”

这节主要介绍Android是怎样将system\_server 这个进程deploy 成android的application的运行环境的，其中会涉及到framework-res.apk , SettingsProvider.apk

**一、 framework-res.apk**

该apk是framework的资源文件apk, 里面保存了framework使用到的一些layout, 图片，string值等等，并且声明一些系统级的Activity.

项目地址: frameworks/base/core/res

**Android.mk**

LOCAL\_PACKAGE\_NAME := framework-res

LOCAL\_CERTIFICATE := platform

LOCAL\_EXPORT\_PACKAGE\_RESOURCES := true //为true, 表示允许framework-res.apk里的资源可以被其它app使用

**AndroidManifest.mk**

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="android"

coreApp="true"

android:sharedUserId="android.uid.system" //system权限

android:sharedUserLabel="@string/android\_system\_label">

从manifest里可以看出framework-res.apk的package name 为 “android”, 且要运行在system进程里

<protected-broadcast android:name="android.intent.action.SCREEN\_OFF" />

<protected-broadcast android:name="android.intent.action.SCREEN\_ON" />

接着定义了很多protected-broadcast，表明该广播只能由系统级应用发出。PKMS会解析该属性, 在系统运作起来之后，如果某个不具有系统权限的应用试图发送系统中的“保护性广播”，那么到AMS的broadcastIntentLocked()处就会被拦住，AMS会抛出异常，提示

java.lang.SecurityException: Permission Denial: not allowed to send broadcast android.intent.action.SCREEN\_OFF from pid=3225, uid=10068

<!-- Allows an application to send SMS messages.

<p>Protection level: dangerous

-->

<permission android:name="android.permission.SEND\_SMS"

android:permissionGroup="android.permission-group.SMS"

android:label="@string/permlab\_sendSms"

android:description="@string/permdesc\_sendSms"

android:permissionFlags="costsMoney"

android:protectionLevel="dangerous" />

<!-- Allows an application to receive SMS messages.

<p>Protection level: dangerous

-->

<permission android:name="android.permission.RECEIVE\_SMS"

android:permissionGroup="android.permission-group.SMS"

android:label="@string/permlab\_receiveSms"

android:description="@string/permdesc\_receiveSms"

android:protectionLevel="dangerous"/>

接着定义一些permission与permission-group

<http://developer.android.com/guide/topics/manifest/permission-element.html> 用来作为安全权限限制访问一些特殊的模块或者features或者其它应用程序，其它程序要使用，必须要使用 <use-permission />

接下来看下它的主application

<application android:process="system"

//默认所有的components运行在system进程, 前提是有相同的shared User ID,和相同的certificate (Android.mk里定义了platform)

android:persistent="true" //persistent进程

android:hasCode="false" //没有application的code,

android:label="@string/android\_system\_label"

android:allowClearUserData="false"

android:backupAgent="com.android.server.backup.SystemBackupAgent"

android:killAfterRestore="false"

android:icon="@drawable/ic\_launcher\_android"

android:supportsRtl="true">

//定义一些activity

<activity android:name="com.android.internal.app.ChooserActivity"

android:theme="@style/Theme.DeviceDefault.Resolver"

android:finishOnCloseSystemDialogs="true"

android:excludeFromRecents="true"

android:documentLaunchMode="never"

android:relinquishTaskIdentity="true"

**二、SettingsProvider.apk**

SettingsProvider.apk 是一个ContentProvider, 主要用来提供系统比较公共的Settings的值等，它是运行是system进程中的，因为system进程里面有很多service, 这些service都可能需要访问到SettingsProvider里的值，因此将SettingsProvider.apk跑在system进程中可以避免不必要的跨进程间消耗

**Android.mk**

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := $(call all-subdir-java-files) \

src/com/android/providers/settings/EventLogTags.logtags

LOCAL\_JAVA\_LIBRARIES := telephony-common ims-common

LOCAL\_PACKAGE\_NAME := SettingsProvider

LOCAL\_CERTIFICATE := platform //platform签名

LOCAL\_PRIVILEGED\_MODULE := true //privileged的apk

**AndroidManifest.xml**

<**manifest** xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.android.providers.settings"  
 coreApp="true"  
 android:sharedUserId="android.uid.system"> //system权限  
  
 <**application** android:allowClearUserData="false"  
 android:label="@string/app\_label"  
 android:process="system"

//该application下的所有Component运行在 system进程中  
 android:backupAgent="SettingsBackupAgent"  
 android:killAfterRestore="false"  
 android:icon="@mipmap/ic\_launcher\_settings">  
  
 *<!--* ***todo add: android:neverEncrypt="true"*** *-->* <**provider** android:name="SettingsProvider" android:authorities="settings"  
 android:multiprocess="false"  
 android:exported="true"

//其它application可通过 URL访问该Provider  
 android:singleUser="true"  
 android:initOrder="100" />  
 </**application**>  
</**manifest**>

**三、配置 SystemServer为Android application的环境**

接下来这节主要来介绍如何将SystemServer配置成android应用程序的运行环境

**public static void** main(String[] args) {  
 **new** SystemServer().run();  
}

**private void** run() {

…

*// Initialize the system context.*PART1, createSystemContext();

//初始化系统上下文

*// Set up the Application instance for the system process and get started.*PART2, mActivityManagerService.setSystemProcess();

//通过AMS，将当前进程设置成android的应用程序的环境，安装framework-res.apk,生成进程相关的ProcessRecord

//安装系统Providers

PART3, mActivityManagerService.installSystemProviders();

}

**PART 1** =======================================================

**private void** createSystemContext() {  
 ActivityThread activityThread = ActivityThread.*systemMain*();

//创建与线程相关的AcivityThread, 并且创建系统context  
 mSystemContext = activityThread.getSystemContext();  
 …  
}

**public static** ActivityThread systemMain() {  
 *…*  
 ActivityThread thread = **new** ActivityThread();

//将当前system\_server运行的主线程关联一个AcivityThread.   
 thread.attach(**true**);  
 **return** thread;  
}

**private void** attach(**boolean** system) {  
 *sCurrentActivityThread* = **this**;  
 mSystemThread = system;  
 **if** (!system) {  
 …  
 } **else** {  
 *// Don't set application object here -- if the system crashes,  
 // we can't display an alert, we just want to die die die.* android.ddm.DdmHandleAppName.*setAppName*("system\_process",  
 UserHandle.*myUserId*());

//将Systemserver进程在DDM里设置名称为 system\_process  
 **try** {  
 mInstrumentation = **new** Instrumentation();

//生成一个Instrumentation???  
 ContextImpl context = ContextImpl.*createAppContext*(  
 **this**, getSystemContext().mPackageInfo);

//这个context完全没用，只是临时使用LoadedApk, “android/system”  
 mInitialApplication = context.mPackageInfo.makeApplication(**true**, **null**);

//生成应用程序对应的全局的Application  
 mInitialApplication.onCreate();

//进入Application的生命周期 onCreate()  
 } **catch** (Exception e) {  
 **…**  
 }  
 }

**public** Application makeApplication(**boolean** forceDefaultAppClass,  
 Instrumentation instrumentation) {  
 Application app = **null**;  
  
 String appClass = mApplicationInfo.className;  
 **if** (forceDefaultAppClass || (appClass == **null**)) {  
 appClass = "android.app.Application";

//强制初始化"android.app.Application"  
 }  
  
 **try** {  
 java.lang.ClassLoader cl = getClassLoader();  
 **if** (!mPackageName.equals("android")) {  
 initializeJavaContextClassLoader();  
 }  
 ContextImpl appContext = ContextImpl.*createAppContext*(mActivityThread, **this**);

//这个创建出来的Context是真正的Systemserver的Application里对应的那个Context  
 app = mActivityThread.mInstrumentation.newApplication(  
 cl, appClass, appContext);  
 appContext.setOuterContext(app);

//生成Application, Application是继承于ContextWrapper，类似于proxy模式，ContextImpl与Application通过mOuterContext, mBase互相引用  
 } **catch** (Exception e) {

…

}  
 mActivityThread.mAllApplications.add(app);

//将当前生成的Application加入到mAllApplications里，可以看出，一个线程是可以跑多个apk的，(一个apk对应一个Application)  
 mApplication = app; //用mApplication表示最初始化的Application  
  
 **return** app;  
}

**PART 2 =======================================**

AMS setSystemProcess() //安装framework-res.apk, 生成systemserver对应的ProcessRecord, 并与**ActivityThread**进行绑定

**public void** setSystemProcess() {

ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(  
 "android", ***STOCK\_PM\_FLAGS***);

//注意，这里的mContext是 mSystemContext, 即系统级的上下文， 查找package name为 “android”的ApplicationInfo，从以上可知package name为 “android”的 apk是 framework-res.apk,即framework的资源文件apk, ApplicationInfo是通过解析framework-res.apk里的AndroidManifest.xml获得的

mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());

//开始为ActivityThread 安装 system application相关信息,将framework-res.apk对应的ApplicationInfo安装到LoadedApk中的mApplicationInfo

//为systemserver 主进程开辟一个ProcessRecord来维护进程的相关信息

**synchronized** (**this**) {

//从framework-res.apk里可以知道info.processName为 “system”进程，即framework-res.apk是要跑在system进程中的。  
 ProcessRecord app = newProcessRecordLocked(info, info.processName, **false**, 0);  
 app.persistent = **true**;  
 app.pid = ***MY\_PID***; //为ProcessRecord赋值当前进程ID，即system\_server进程ID  
 app.maxAdj = ProcessList.***SYSTEM\_ADJ***; //这个值跟OOM killer有关，值越小，越不容易被kill来释放内存  
 app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);

//将ProcessRecord与ActivityThread进行关联  
 **synchronized** (mPidsSelfLocked) {  
 mPidsSelfLocked.put(app.pid, app);

//将ProcessRecord放到mPidSelfLocked里统一管理  
 }  
 updateLruProcessLocked(app, **false**, **null**);  
 updateOomAdjLocked(); //更新oom adj, 没看  
}

}

//生成 ProcessRecord对象

**final** ProcessRecord newProcessRecordLocked(ApplicationInfo info, String customProcess,  
 **boolean** isolated, **int** isolatedUid) {  
 String proc = customProcess != **null** ? customProcess : info.processName;  
 **final** ProcessRecord r = **new** ProcessRecord(stats, info, proc, uid);  
 **if** (!mBooted && !mBooting  
 && userId == UserHandle.***USER\_OWNER*** && (info.flags & ***PERSISTENT\_MASK***) == ***PERSISTENT\_MASK***) {  
 r.persistent = **true**; //persistent  
 }  
 addProcessNameLocked(r);  
 **return** r;

}

**PART 3 =======================================**

AMS installSystemProviders() //安装系统级的Providers

**public final void** installSystemProviders() {  
 List<ProviderInfo> providers;  
 **synchronized** (**this**) {  
 ProcessRecord app = mProcessNames.get("system", Process.***SYSTEM\_UID***);

//这里是查找system进程的ProcessRecord，即PART2 生成的  
 providers = generateApplicationProvidersLocked(app);

//根据app.processName “system” 来查看Providers, 在这里是SettingsProvider  
 **if** (providers != **null**) {  
 **for** (**int** i=providers.size()-1; i>=0; i--) {  
 ProviderInfo pi = (ProviderInfo)providers.get(i);  
 **if** ((pi.applicationInfo.flags&ApplicationInfo.***FLAG\_SYSTEM***) == 0) {  
 Slog.*w*(***TAG***, "Not installing system proc provider " + pi.name  
 + ": not system .apk");

//这里只安装系统级的Providers  
 providers.remove(i);  
 }  
 }  
 }  
 }  
 **if** (providers != **null**) {  
 mSystemThread.installSystemProviders(providers);  
 }  
  
 mCoreSettingsObserver = **new** CoreSettingsObserver(**this**);  
  
 *//mUsageStatsService.monitorPackages();*}

//往**ActivityThread**里安装SystemProviders, mInitialApplication即是systemserver进程的Application, 前面有讲

mSystemThread.installSystemProviders(providers); ->

installContentProviders(**mInitialApplication**, providers);

**private void** installContentProviders(  
 Context **context**, List<ProviderInfo> providers) {  
 **final** ArrayList<IActivityManager.ContentProviderHolder> results =  
 **new** ArrayList<IActivityManager.ContentProviderHolder>();  
  
 **for** (ProviderInfo cpi : providers) {  
 **if** (***DEBUG\_PROVIDER***) {  
 StringBuilder buf = **new** StringBuilder(128);  
 buf.append("Pub ");  
 buf.append(cpi.authority);  
 buf.append(": ");  
 buf.append(cpi.name);  
 Log.*i*(***TAG***, buf.toString());  
 }

//具体安装到 ActivityThread里的mProviderMap

IActivityManager.ContentProviderHolder cph = installProvider(context, **null**, cpi, **false /\*noisy\*/**, **true /\*noReleaseNeeded\*/**, **true /\*stable\*/**);  
 **if** (cph != **null**) {  
 cph.noReleaseNeeded = **true**;  
 results.add(cph);  
 }  
 }  
  
 **try** {  
 ActivityManagerNative.*getDefault*().publishContentProviders(  
 getApplicationThread(), results);  
 } **catch** (RemoteException ex) {  
 }  
}

**四、 SystemServer的ActivityThread安装图**

