# Todo

FallbackHome原理

开机整个过程，越详细越好

<https://blog.csdn.net/lilian0118/article/details/23168531>

dlg堵塞的原因，默认类型是啥

Window w = **mDialog**.getWindow();  
w.setType(WindowManager.LayoutParams.***TYPE\_SYSTEM\_ALERT***);  
**mDialog**.show();

跟UID系统签名没有关的，不必等于1000

需要依赖系统目录的

Application\_directBootAware作用域不是最大

Intent in = **new** Intent();  
in.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***);  
*//class 为何需要全名啊*in.setClassName(**"com.android.settings"**, **"com.android.settings.Settings"**);  
startActivity(in);

Intent intent = new Intent(Intent.ACTION\_MASTER\_CLEAR);

intent.addFlags(Intent.FLAG\_RECEIVER\_FOREGROUND);

intent.putExtra(Intent.EXTRA\_REASON, "MasterClearConfirm");

intent.putExtra(Intent.EXTRA\_WIPE\_EXTERNAL\_STORAGE, mEraseSdCard);

MasterClearReceiver

StorageManager.wipeAdoptableDisks

MountService.partitionPublic

开机优化

<https://blog.csdn.net/freshui/article/details/53700771>

<https://blog.csdn.net/wd229047557/article/details/80773797>

https://blog.csdn.net/zhangbijun1230/article/details/80562616

<https://blog.csdn.net/freshui/article/details/53700771>

130|rm500:/ # service list

通过服务来裁剪吧

---------------------

-----------------------------------

<http://www.52im.net/thread-125-1-1.html>

ShutdownThread 关机动画，也可以用那个C++效率高？有什么好处呢？？明显不搭边啊

config\_autoBrightness可以加速调节？？？

SystemService.stop(“bootanimal”)

ZygoteInit

如何优化，怎么样的类，是重的类：Typeface, PhoneNumberUtil

不觉得呢

？

Preloaded-classes0

Class.*forName*(**""**,**true**, **null**);

D:\BaiduYunDownload\android-7.1.1\_r1\android-7.1.1\_r1\frameworks\base\preloaded-classes

把自己的手机优化好一点

Android7.0(Android N)适配教程，心得

https://blog.csdn.net/chuyouyinghe/article/details/52838678

## Hook

# [App 中不需要root使用 hook](https://confluence.djicorp.com/pages/viewpage.action?pageId=66594015)

xhook: <https://github.com/iqiyi/xHook>

**github上星标已经在700多个了，关注度还是蛮高的。**

抄录自 readMe

# 编译

你需要 google NDK 来编译 xhook。  
<https://developer.android.com/ndk/downloads/index.html>

最新版本的 xhook 在开发和调试中使用的 NDK 版本是 r16b。

* 编译动态库 ([libxhook.so](http://libxhook.so/) 和其他的用于测试的动态库)

./build\_libs.sh

* 把动态库安装到 Demo 工程的 libs 目录中

./install\_libs.sh

* 清除动态库

./clean\_libs.sh

# Demo

cd ./xhookwrapper/

./gradlew assembleDebug

adb install ./app/build/outputs/apk/debug/app-debug.apk

### blkio子系统

Android 开机速度优化

https://www.jianshu.com/p/007a87a26bf0

*/\*\*  
 \* Writes trace events to the system trace buffer. These trace events can be  
 \* collected and visualized using the Systrace tool.  
 \*  
 \* <p>This tracing mechanism is independent of the method tracing mechanism  
 \* offered by {****@link*** *Debug#startMethodTracing}. In particular, it enables  
 \* tracing of events that occur across multiple processes.  
 \* <p>For information about using the Systrace tool, read <a  
 \* href="{****@docRoot****}tools/debugging/systrace.html">Analyzing Display and Performance  
 \* with Systrace</a>.  
 \*/***public final class** Trace {

和埋点之间的关系

设计模式与android系统

https://www.jianshu.com/p/4f1a06675039

traceBeginAndSlog与买点关系

如何设计framework-res.Apk.关联到的呢，我也写一个类似的app呢

Environment.setUserRequired(**true**);为何需要这个限制呢，怎么解除了，安全性怎么保障

多用户管理User，分析系统设置什么时候不可用啊

从 onPlaySoundEffect来看，manager和service关系，定制放在哪个地方比较好呢

UsbDeviceManager

StorageManager

* Aidl在服务中的设计
* shilifenxi https://blog.csdn.net/myfriend0/article/details/55210074

haveInputDeviceWithVibrator震动配置

https://blog.csdn.net/u010164190/article/details/51866419

Android Context getSystemService分析

<https://blog.csdn.net/xyh269/article/details/52789737>

private void createSystemContext() {

ActivityThread activityThread = ActivityThread.systemMain();

mSystemContext = activityThread.getSystemContext();

mSystemContext.setTheme(DEFAULT\_SYSTEM\_THEME);

}

https://blog.csdn.net/qq\_37475168/article/details/80909102

**CountryDetectorService**

AccessibilityManager：https://www.jianshu.com/p/4cd8c109cdfb

services\core\java\com\android\server\firewall

Watchdog

NetworkTimeUpdateService 网络时间更新服务

CommonTimeManagementService 时间管理服务

UsageStatsService: 这是一个Android私有service,主要作用是收集用户使用每一个APP的频率、使买点更新？

dji.json 网络配置

Chaechrevorey:MISUC 为何不会擦除，LAST\_LOG

Ps等Android命令研究

和rk对比改动点，学习

和其他人kk比较改动点，学习之，继承经验

关闭开机启动服务

SystemService.stop(“booanmin”)

获取包名

WindowState.mAttrs.packageName

不管类有多大，只要遵循一定的分析方法，总能逐步的将这个类理解下来。 笔者认为，要分析一个类只要遵循以下几个步骤即可：

理解该类的主要作用

分析类中的主要字段

理解类的构造方法以及初始化过程

理解类中的主要业务逻辑方法

分析类中的其他成员：例如内部类

分析与这个类紧密相关的其他类

# 回调设计

IAccountManagerResponse chooseResponse =  
 **new** IAccountManagerResponse.Stub() {  
 @Override  
 **public void** onResult(Bundle value) **throws** RemoteException {  
 Account account = **new** Account(  
 value.getString(KEY\_ACCOUNT\_NAME),  
 value.getString(KEY\_ACCOUNT\_TYPE),  
 value.getString(KEY\_ACCOUNT\_ACCESS\_ID));  
 mFuture = getAuthToken(account, mAuthTokenType,  
 mLoginOptions, mActivity, mMyCallback,  
 mHandler);  
 }  
  
 @Override  
 **public void** onError(**int** errorCode, String errorMessage)  
 **throws** RemoteException {  
 mResponse.onError(errorCode, errorMessage);  
 }  
};

# SystemServer

Android体系架构中四种意义上服务:Native服务、Android服务、Init空间的服务、应用层空间的服务。这里的system service属于android服务这一块。

浏览一下Android的system service

Adb shell Service list

从结果看来Android后台有很多的system service,他们是分散在不同进程中的线程实体。

从架构来说，实现了资源文件和代码的解耦：所有的资源文件是framework-res.Apk.提供的，我们的资源包加载也可以这么做啊，包括资源预加载！！！先把首页的加载上去吧。。

## 基本概念

### SystemServer

是Android系统的一个核心进程,它是由zygote进程创建的,因此在android的启动过程中位于zygote之后。android的所有服务循环都是建立在 SystemServer之上的。在SystemServer中,将可以看到它建立了android中的大部分服务,并通过ServerManager的add\_service方法把这些服务加入到了ServiceManager的svclist中。从而完成ServcieManager对服务的管理。

### Service Manager

Service manager是管理以上services的一个进程,可以在adb shell中运行ps看看进程列表就知道了。

源代码位于:frameworks/base/cmds/servicemanager

执行方式:

他是用c和c++语言编写的natvie可以执行文件。在Android中称之为EXECUTABLE,这个名称很重要因为Android.mk文件中用这个名字来确定他是可以执行的二进制文件。

### 基本框架

SystemServer的main()函数首先调用的是init1()函数,这是一个native函数,内部会进行一些与Dalvik虚拟机相关的初始化工作。该函数执行完毕后,其内部会调用Java端的init2()函数,这就是为什么Java源码中没有引用init2()的地方,主要的系统服务都是在init2()函数中完成的。

该函数首先创建了一个ServerThread对象,该对象是一个线程,然后直接运行该线程,如以下代码所示:

public static final void init2() { Slog.i(TAG, "Entered the Android system server!");

Thread thr = new ServerThread();

thr.setName("android.server.ServerThread");

thr.start(); }

基本上每个服务都有对应的Java类,从编码规范的角度来看,启动这些服务的模式可归类为三种,如图9-3所示。于是,从ServerThread的run()方法内部开始真正启动各种服务线程。



模式二是指服务类会提供一个getInstance()方法,通过该方法获取该服务对象,这样的好处是保证系统中仅包含一个该服务对象。模式一是指直接使用构造函数构造一个服务,由于大多数服务都对应一个线程,因此,在构造函数内部就会创建一个线程并自动运行。

模式三是指从服务类的main()函数中开始执行。

无论以上何种模式,当创建了服务对象后,有时可能还需要调用该服务类的init()或者systemReady()函数以完成该对象的启动,当然这 些都是服务类内部自定义的。为了区分以上启动的不同,以下采用一种新的方式描述该启动过程。比如当一个服务对象是通过模式一创建,并调用init()完成 该服务的启动,我们就用模式1.2表示;如果构造函数返回后就已经启动,而无须任何其他调用,即什么都不做(nothing),我们就用模式1.1表示。

### 启动服务列表

表9-2列出了SystemServer中所启动的所有服务,以及这些服务的启动模式。

表9-2  SystemServer中启动服务列表

|  |  |  |
| --- | --- | --- |
| 服务类名称 | 作用描述 | 启动模式 |
| EntropyService | 提供伪随机数 | 1.0 |
| PowerManagerService | 电源管理服务 | 1.2/3 |
| ActivityManagerService | 最核心的服务之一,管理 Activity | 自定义 |
| TelephonyRegistry | 通过该服务注册电话模块的事件响应,比如重启、关闭、启动等 | 1.0 |
| PackageManagerService | 程序包管理服务 | 3.3 |
| AccountManagerService | 账户管理服务,是指联系人账户,而不是 Linux 系统的账户 | 1.0 |
| ContentService | ContentProvider 服务,提供跨进程数据交换 | 3.0 |
| BatteryService | 电池管理服务 | 1.0 |
| LightsService | 自然光强度感应传感器服务  // 管理发光二极管和显示背光，我们需要它来显示显示器 | 1.0 |
| VibratorService | 震动器服务 | 1.0 |
| AlarmManagerService | 定时器管理服务,提供定时提醒服务 | 1.0 |
| WindowManagerService | Framework 最核心的服务之一,负责窗口管理 | 3.3 |
| BluetoothService | 蓝牙服务 | 1.0 + |
| DevicePolicyManagerService | 提供一些系统级别的设置及属性 | 1.3 |
| StatusBarManagerService | 状态栏管理服务 | 1.3 |
| ClipboardService | 系统剪切板服务 | 1.0 |
| InputMethodManagerService | 输入法管理服务 | 1.0 |
| NetStatService | 网络状态服务 | 1.0 |
| NetworkManagementService | 网络管理服务 | NMS.create() |
| ConnectivityService | 网络连接管理服务 | 2.3 |
| ThrottleService | 暂不清楚其作用 | 1.3 |

(续表)

|  |  |  |
| --- | --- | --- |
| 服务类名称 | 作用描述 | 启动模式 |
| AccessibilityManagerService | 辅助管理程序截获所有的用户输入,并根据这  些输入给用户一些额外的反馈,起到辅助的效果 | 1.0 |
| MountService | 挂载服务,可通过该服务调用 Linux 层面的 mount 程序 | 1.0 |
| NotificationManagerService | 通知栏管理服务, Android 中的通知栏和状  态栏在一起,只是界面上前者在左边,后者在右边 | 1.3 |
| DeviceStorageMonitorService | 磁盘空间状态检测服务 | 1.0 |
| LocationManagerService | 地理位置服务 | 1.3 |
| SearchManagerService | 搜索管理服务 | 1.0 |
| DropBoxManagerService | 通过该服务访问 Linux 层面的 Dropbox 程序 | 1.0 |
| WallpaperManagerService | 墙纸管理服务,墙纸不等同于桌面背景,  在 View 系统内部,墙纸可以作为任何窗口的背景 | 1.3 |
| AudioService | 音频管理服务 | 1.0 |
| BackupManagerService | 系统备份服务 | 1.0 |
| AppWidgetService | Widget 服务 | 1.3 |
| RecognitionManagerService | 身份识别服务 | 1.3 |
| DiskStatsService | 磁盘统计服务 | 1.0 |

这些服务在又具体定义

@StringDef({

POWER\_SERVICE,

WINDOW\_SERVICE,

LAYOUT\_INFLATER\_SERVICE,

ACCOUNT\_SERVICE,

ACTIVITY\_SERVICE,

ALARM\_SERVICE,

NOTIFICATION\_SERVICE,

ACCESSIBILITY\_SERVICE,

CAPTIONING\_SERVICE,

KEYGUARD\_SERVICE,

LOCATION\_SERVICE,

//@hide: COUNTRY\_DETECTOR,

SEARCH\_SERVICE,

SENSOR\_SERVICE,

STORAGE\_SERVICE,

WALLPAPER\_SERVICE,

VIBRATOR\_SERVICE,

//@hide: STATUS\_BAR\_SERVICE,

CONNECTIVITY\_SERVICE,

//@hide: UPDATE\_LOCK\_SERVICE,

//@hide: NETWORKMANAGEMENT\_SERVICE,

NETWORK\_STATS\_SERVICE,

//@hide: NETWORK\_POLICY\_SERVICE,

WIFI\_SERVICE,

WIFI\_NAN\_SERVICE,

WIFI\_P2P\_SERVICE,

WIFI\_SCANNING\_SERVICE,

//@hide: WIFI\_RTT\_SERVICE,

//@hide: ETHERNET\_SERVICE,

WIFI\_RTT\_SERVICE,

NSD\_SERVICE,

AUDIO\_SERVICE,

FINGERPRINT\_SERVICE,

MEDIA\_ROUTER\_SERVICE,

TELEPHONY\_SERVICE,

TELEPHONY\_SUBSCRIPTION\_SERVICE,

CARRIER\_CONFIG\_SERVICE,

TELECOM\_SERVICE,

CLIPBOARD\_SERVICE,

INPUT\_METHOD\_SERVICE,

TEXT\_SERVICES\_MANAGER\_SERVICE,

APPWIDGET\_SERVICE,

//@hide: VOICE\_INTERACTION\_MANAGER\_SERVICE,

//@hide: BACKUP\_SERVICE,

DROPBOX\_SERVICE,

//@hide: DEVICE\_IDLE\_CONTROLLER,

DEVICE\_POLICY\_SERVICE,

UI\_MODE\_SERVICE,

DOWNLOAD\_SERVICE,

NFC\_SERVICE,

BLUETOOTH\_SERVICE,

//@hide: SIP\_SERVICE,

USB\_SERVICE,

LAUNCHER\_APPS\_SERVICE,

//@hide: SERIAL\_SERVICE,

//@hide: HDMI\_CONTROL\_SERVICE,

INPUT\_SERVICE,

DISPLAY\_SERVICE,

USER\_SERVICE,

RESTRICTIONS\_SERVICE,

APP\_OPS\_SERVICE,

CAMERA\_SERVICE,

PRINT\_SERVICE,

CONSUMER\_IR\_SERVICE,

//@hide: TRUST\_SERVICE,

TV\_INPUT\_SERVICE,

//@hide: NETWORK\_SCORE\_SERVICE,

USAGE\_STATS\_SERVICE,

MEDIA\_SESSION\_SERVICE,

BATTERY\_SERVICE,

JOB\_SCHEDULER\_SERVICE,

//@hide: PERSISTENT\_DATA\_BLOCK\_SERVICE,

MEDIA\_PROJECTION\_SERVICE,

MIDI\_SERVICE,

RADIO\_SERVICE,

HARDWARE\_PROPERTIES\_SERVICE,

//@hide: SOUND\_TRIGGER\_SERVICE,

SHORTCUT\_SERVICE,

//@hide: CONTEXTHUB\_SERVICE,

})

@Retention(RetentionPolicy.SOURCE)

public @interface ServiceName {}

public abstract Object getSystemService(@ServiceName @NonNull String name);

## 启动流程

### 时序



### SystemServer.main

SystemServer.java

|  |
| --- |
| public final class SystemServer {  ...  public static void main(String[] args) {  //先初始化SystemServer对象，再调用对象的run()方法  new SystemServer().run();  }  } |

### SystemServer.run

SystemServer.java

|  |
| --- |
| private void run() {  // Prepare the main looper thread (this thread). 初始化looper，是不是没想到looper会被用在此处？  android.os.Process.setThreadPriority(  android.os.Process.THREAD\_PRIORITY\_FOREGROUND);  android.os.Process.setCanSelfBackground(false);  Looper.prepareMainLooper()  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }  ...  if (!SystemProperties.get("persist.sys.language").isEmpty()) {  设置系统的的语言  final String languageTag = Locale.getDefault().toLanguageTag();  SystemProperties.set("persist.sys.locale", languageTag);  SystemProperties.set("persist.sys.language", "");  SystemProperties.set("persist.sys.country", "");  SystemProperties.set("persist.sys.localevar", "");  }  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());  Looper.prepareMainLooper();// 准备主线程looper  //加载android\_servers.so库，该库包含的源码在frameworks/base/services/目录下  System.loadLibrary("android\_servers");  //检测上次关机过程是否失败，该方法可能不会返回[见小节3.6.1]  performPendingShutdown();  createSystemContext(); //初始化系统上下文  //创建系统服务管理  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  //启动各种系统服务[见小节3.7]  try {  startBootstrapServices(); // 启动引导服务  startCoreServices(); // 启动核心服务  startOtherServices(); // 启动其他服务[见小节4.6]  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  }  //一直循环执行  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

#### 准备SystemServer运行环境

// 准备SystemServer运行环境:设置线程优先级，创建主线层Looper，ActivityThread和SystemContext

#### 设置时间

Sda

*// If a device's clock is before 1970 (before 0), a lot of  
// APIs crash dealing with negative numbers, notably  
// java.io.File#setLastModified, so instead we fake it and  
// hope that time from cell towers or NTP fixes it shortly.***if** (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  
 Slog.w(TAG, **"System clock is before 1970; setting to 1970."**);  
 SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  
}

如何修改Android系统默认时间

https://blog.csdn.net/ljx646566715/article/details/81189914

#### Shiqi

SystemProperties.set(**"persist.sys.timezone"**, **"GMT"**);

#### 设置系统的语言

/1、设置系统的的语言

if (!SystemProperties.get("persist.sys.language").isEmpty()) {

final String languageTag = Locale.getDefault().toLanguageTag();

SystemProperties.set("persist.sys.locale", languageTag);

SystemProperties.set("persist.sys.language", "");

SystemProperties.set("persist.sys.country", "");

SystemProperties.set("persist.sys.localevar", "");

}

这里通过SystemProperties类去读取系统的属性。这个语句的逻辑也比较简单，主要是去设置系统语言的环境。这些属性最后都会被init进程在它们对应的动作列表或者是服务列表中检测到，并且调用相对应的函数去执行。

#### 进程性能统计

系统中很多进程都需要通过SamplingProfilerIntegration去统计性能。

// 2、Enable the sampling profiler.进程性能统计

if (SamplingProfilerIntegration.isEnabled()) {

SamplingProfilerIntegration.start();

mProfilerSnapshotTimer = new Timer();

mProfilerSnapshotTimer.schedule(new TimerTask() {

@Override

public void run() {

SamplingProfilerIntegration.writeSnapshot("system\_server", null);

}

}, SNAPSHOT\_INTERVAL, SNAPSHOT\_INTERVAL);

}

#### 置虚拟机运行内存

//3、 Mmmmmm... more memory!设置虚拟机运行内存

VMRuntime.getRuntime().clearGrowthLimit();

VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);

Build.ensureFingerprintProperty();

Environment.setUserRequired(true);

Bundle.setShouldDefuse(true);

#### 设置环境变量

*// Within the system server, it is an error to access Environment paths without  
// explicitly specifying a user.*Environment.setUserRequired(**true**);

### 服务启动

这三类：boot服务、core服务、其他服务.启动critical级别的服务，他们有复杂的依赖关系，顺序不能乱,自己想加服

#### 载入nativie 服务库

*// Initialize native services.*System.loadLibrary(**"android\_servers"**);

手动加载的啊，也没有什么神奇的

#### createSystemContext()

系统Context初始化过程

**private void** createSystemContext() {  
 ActivityThread activityThread = ActivityThread.systemMain();  
 mSystemContext = activityThread.getSystemContext();  
 mSystemContext.setTheme(DEFAULT\_SYSTEM\_THEME);  
  
 **final** Context systemUiContext = activityThread.getSystemUiContext();  
 systemUiContext.setTheme(DEFAULT\_SYSTEM\_THEME);  
}

android.ddm.DdmHandleAppName.setAppName("system\_process",

UserHandle.myUserId());

**系统APK(framework-res.apk)对应的LoadedApk初始化**

/\*\*

\* Create information about the system package.

\* system package : framework-res.apk,packagename为android

\* Must call {@link #installSystemApplicationInfo} later.

\*/

LoadedApk(ActivityThread activityThread) {

mActivityThread = activityThread;

mApplicationInfo = new ApplicationInfo();

**// packageName为"android",这个APK为framework-res.apk**

**mApplicationInfo.packageName = "android";**

**mPackageName = "android";**

mAppDir = null;

mResDir = null;

mSplitAppDirs = null;

mSplitResDirs = null;

mOverlayDirs = null;

mSharedLibraries = null;

mDataDir = null;

mDataDirFile = null;

mLibDir = null;

mBaseClassLoader = null;

mSecurityViolation = false;

mIncludeCode = true;

mRegisterPackage = false;

mClassLoader = ClassLoader.getSystemClassLoader();

mResources = Resources.getSystem();

}

SystemServer.createSystemContext()分析到这里就基本结束了，从这里可以看到，SystemServer.createSystemContext()主要是创建了ActivityThread，获取了Context，这个Context是系统进程（framework-res.apk）运行的环境。另外，从代码中也可以看到:  
1)ActivityThread中用了一个集合来保存Application，说明一个进程里面可以包含多个Application;  
具体实现：在APK的AndroidManifext.xml中配置<manifest>标签的android:sharedUserId和配置<application>标签的android:process相同即可，这样配置之后，这些APK就能够运行在同一个进程，实现数据共享；  
2)Application是在LoadedApk中通过Intrumentation创建出来的(LoadedApk.makeApplication()-->Instrumentation.newApplication()-->Application.attach())，Application通过attach()方法持有LoadedApk对象，Application和LoadedApk是一一对应的关系。

<https://www.jianshu.com/p/7d4b605f5060>

这里的context是系统的，没有注册服务缓存。因此，获取服务需要：

mWifiManager = IWifiManager.Stub.asInterface(

ServiceManager.getService(Context.WIFI\_SERVICE));

而不是：

// WifiManager wifiManager = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

#### startBootstrapServices

|  |
| --- |
| public final class SystemServer {  private void startBootstrapServices() {  ...  //phase100  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);  ...  //启动Installer    //启动ActivityManagerService    //启动PowerManagerService    //启动DisplayManagerService    //启动PackageManagerService    }  } |

//Installer类，该类是系统安装apk时的一个服务类，该类是系统安装apk时的一个服务类

Installer installer = mSystemServiceManager.startService(Installer.class);

##### ActivityMS

Sda

*// Activity manager runs the show.*traceBeginAndSlog(**"StartActivityManager"**);  
mActivityManagerService = mSystemServiceManager.startService(  
 ActivityManagerService.Lifecycle.**class**).getService();  
mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  
mActivityManagerService.setInstaller(installer);  
traceEnd();

##### setSystemProcess

**public void** setSystemProcess() {  
 **try** {  
 ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(  
 **"android"**, STOCK\_PM\_FLAGS | MATCH\_SYSTEM\_ONLY);  
 mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());  
  
 **synchronized** (**this**) {  
 ProcessRecord app = newProcessRecordLocked(info, info.processName, **false**, 0);  
 app.persistent = **true**;  
 app.pid = MY\_PID;  
 app.maxAdj = ProcessList.SYSTEM\_ADJ;  
 app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);  
 **synchronized** (mPidsSelfLocked) {  
 mPidsSelfLocked.put(app.pid, app);  
 }  
 updateLruProcessLocked(app, **false**, **null**);  
 updateOomAdjLocked();  
 }  
 } **catch** (PackageManager.NameNotFoundException e) {  
 **throw new** RuntimeException(  
 **"Unable to find android system package"**, e);  
 }  
}

但system\_server的ADJ并非等于-900，而是-800？是由于startPersistentApps()过程直接把其adj重新被设置为-800，这算是一个小BUG，但 其实目前来说对于ADJ<0的进程，LMK不会杀，两者没有什么区别。

#### startCoreServices

/\* 启动一些关键服务\*

//启动LightsService

//启动BatteryService

//启动UsageStatsService

//启动WebViewUpdateService

#### startOtherServices

*// For debugging RescueParty***if** (Build.IS\_DEBUGGABLE && SystemProperties.getBoolean(**"debug.crash\_system"**, **false**)) {  
 **throw new** RuntimeException();  
}

//启动AccountManagerService 帐号管理

//启动ContentService 内容管理

//启动SystemProviders

//启动VibratorService 震动

//启动ConsumerIrService 远程控制周边设备

//启动AlarmManagerService 闹钟

//启动Watchdog

//启动WindowManagerService 窗口管理

//启动InputMethodManagerService 输入法

//截获用户输入，给一些额外反馈，view的点击、焦点事件分发

//启动AccessibilityManagerService

//启动MountService 磁盘加载服务

//启动LockSettingsService 锁屏、手势

//启动PersistentDataBlockService

//启动DevicePolicyManagerService //保证和API8兼容

//启动StatusBarManagerService 状态栏

//启动ClipboardService 剪切板

//启动NetworkManagementService 网络管理

//启动NetworkScoreService

//启动NetworkStatsService 网络统计

//启动NetworkPolicyManagerService 维护网络使用策略

//启动WifiService WIFI服务

//启动ConnectivityService 网络连接状态

//启动网络发现服务

//启动TextServicesManagerService 文本服务，如文本检查

//启动UpdateLockService

//启动LocationManagerService 位置

//启动**CountryDetectorService** 国家检测

//启动SearchManagerService 搜索

//启动DropBoxManagerService

//启动WallpaperManagerService 墙纸

//启动AudioService 音频

//启动USB服务

//WiredAccessoryManager 有线接入，耳机之类的

//启动SerialService 不知道是啥

//启动TwilightService 指明用户当前位置是否为晚上，配合调整夜间模式

//启动UiModeManagerService 界面模式

//启动JobSchedulerService 任务调度

//启动BackupManagerService 备份

//启动APPWIDGET\_SERVICE 窗口小部件管理

//启动VOICE\_RECOGNITION\_MANAGER\_SERVICE 声音重置服务

//启动DiskStatsService 硬盘统计服务

//启动NetworkTimeUpdateService 网络时间更新服务

//启动CommonTimeManagementService 时间管理服务

//启动DreamManagerService 屏幕保护

//负责将预加载的bitmap组装成纹理贴图，生成的纹理贴图可以被用来跨进程使用，以减少内存

//启动AssetAtlasService

//启动打印服务

//RestrictionsManagerService ？

//MediaSessionService ？

//HdmiControlService HDMI服务

//TvInputManagerService TV输入管理

//MediaRouterService ？

//TrustManagerService 信任管理

//FingerprintService 指纹

//BackgroundDexOptService ？

//LauncherAppsService

//MediaProjectionManagerService

//如果“安全” ，禁用JIT编译模式，不安全则开启JIT 何为安全？？？？

// VMRuntime.getRuntime().disableJitCompilation()

//MmsServiceBroker ？

//接下来启动各种APP，省略……

/\*

启动各种各样的“杂项”服务

有一个关键标志变量 disableNonCoreServices 是否把非核心的服务禁用——————如果是，下面很多服务就不会启动了

Starts a miscellaneous grab bag of stuff that has yet to be refactored

\* and organized.

private void startOtherServices() {

...

//phase480 和phase500

mSystemServiceManager.startBootPhase(SystemService.PHASE\_LOCK\_SETTINGS\_READY);

mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);

...

//[见小节4.7]

mActivityManagerService.systemReady(new Runnable() {

@Override

public void run() {

//phase550

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);

...

//phase600

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_THIRD\_PARTY\_APPS\_CAN\_START);

}

}

}

* start: 创建AMS, PMS, LightsService, DMS.
* phase100: 进入Phase100, 创建PKMS, WMS, IMS, DBMS, LockSettingsService, JobSchedulerService, MmsService等服务;
* phase480 && 500: 进入Phase480, 调用WMS, PMS, PKMS, DisplayManagerService这4个服务的systemReady();
* Phase550: 进入phase550, 执行AMS.systemReady(), 启动SystemUI, WebViewFactory, Watchdog.
* Phase600: 进入phase600, 执行AMS.systemReady(), 执行各种服务的systemRunning().
* Phase1000: 进入1000, 执行finishBooting, 启动启动on-hold进程.

##### WallpaperManagerService

config\_enableWallpaperService的config值

*<!-- Component name of the built in wallpaper used to display bitmap wallpapers. This must not be null. -->*<**string name="image\_wallpaper\_component" translatable="false"**>com.android.systemui/com.android.systemui.ImageWallpaper</**string**>

mImageWallpaper = ComponentName.unflattenFromString(  
 context.getResources().getString(R.string.image\_wallpaper\_component));

##### startSystemUi

//启动系统UI

static final void startSystemUi(Context context) {

Intent intent = new Intent();

intent.setComponent(new ComponentName("com.android.systemui",

"com.android.systemui.SystemUIService"));

intent.addFlags(Intent.FLAG\_DEBUG\_TRIAGED\_MISSING);

//Slog.d(TAG, "Starting service: " + intent);

context.startServiceAsUser(intent, UserHandle.SYSTEM);

}

## App如何启动

对于普通的app进程,跟system\_server进程的启动过来有些类似.不同的是app进程是向发消息给system\_server进程, 由system\_server向zygote发出创建进程的请求.

[理解Android进程创建流程](http://gityuan.com/2016/03/26/app-process-create/), 可知进程创建后 接下来会进入ActivityThread.main()过程。

### ActivityThread.main

|  |
| --- |
| public static void main(String[] args) {  ...  Environment.initForCurrentUser();  ...  Process.setArgV0("<pre-initialized>");  //创建主线程looper  Looper.prepareMainLooper();  ActivityThread thread = new ActivityThread();  thread.attach(false); //attach到系统进程  if (sMainThreadHandler == null) {  sMainThreadHandler = thread.getHandler();  }    //主线程进入循环状态  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

app进程的主线程调用栈的栈底如下:

|  |
| --- |
| ...  at android.app.ActivityThread.main(ActivityThread.java:5442)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

跟前面介绍的system\_server进程调用栈对比:

|  |
| --- |
| at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

### main

SystemServer.java

|  |
| --- |
| public final class SystemServer {  ...  public static void main(String[] args) {  //先初始化SystemServer对象，再调用对象的run()方法  new SystemServer().run();  }  } |

### SystemServer.run

SystemServer.java

|  |
| --- |
| private void run() {  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }  ...  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());  Looper.prepareMainLooper();// 准备主线程looper  //加载android\_servers.so库，该库包含的源码在frameworks/base/services/目录下  System.loadLibrary("android\_servers");  //检测上次关机过程是否失败，该方法可能不会返回[见小节3.6.1]  performPendingShutdown();  createSystemContext(); //初始化系统上下文  //创建系统服务管理  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  //启动各种系统服务[见小节3.7]  try {  startBootstrapServices(); // 启动引导服务  startCoreServices(); // 启动核心服务  startOtherServices(); // 启动其他服务[见小节4.6]  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  }  //一直循环执行  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

### 服务启动

|  |
| --- |
| public final class SystemServer {  private void startBootstrapServices() {  ...  //phase100  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);  ...  }  private void startOtherServices() {  ...  //phase480 和phase500  mSystemServiceManager.startBootPhase(SystemService.PHASE\_LOCK\_SETTINGS\_READY);  mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);  ...  //[见小节4.7]  mActivityManagerService.systemReady(new Runnable() {  @Override  public void run() {  //phase550  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);  ...  //phase600  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_THIRD\_PARTY\_APPS\_CAN\_START);  }  }  }  } |

* start: 创建AMS, PMS, LightsService, DMS.
* phase100: 进入Phase100, 创建PKMS, WMS, IMS, DBMS, LockSettingsService, JobSchedulerService, MmsService等服务;
* phase480 && 500: 进入Phase480, 调用WMS, PMS, PKMS, DisplayManagerService这4个服务的systemReady();
* Phase550: 进入phase550, 执行AMS.systemReady(), 启动SystemUI, WebViewFactory, Watchdog.
* Phase600: 进入phase600, 执行AMS.systemReady(), 执行各种服务的systemRunning().
* Phase1000: 进入1000, 执行finishBooting, 启动启动on-hold进程.

## App

对于普通的app进程,跟system\_server进程的启动过来有些类似.不同的是app进程是向发消息给system\_server进程, 由system\_server向zygote发出创建进程的请求.

[理解Android进程创建流程](http://gityuan.com/2016/03/26/app-process-create/), 可知进程创建后 接下来会进入ActivityThread.main()过程。

### ActivityThread.main

|  |
| --- |
| public static void main(String[] args) {  ...  Environment.initForCurrentUser();  ...  Process.setArgV0("<pre-initialized>");  //创建主线程looper  Looper.prepareMainLooper();  ActivityThread thread = new ActivityThread();  thread.attach(false); //attach到系统进程  if (sMainThreadHandler == null) {  sMainThreadHandler = thread.getHandler();  }    //主线程进入循环状态  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

app进程的主线程调用栈的栈底如下:

|  |
| --- |
| ...  at android.app.ActivityThread.main(ActivityThread.java:5442)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

跟前面介绍的system\_server进程调用栈对比:

|  |
| --- |
| at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

## startService

/\*\*

\* Starts a service by class name.

\*

\* @return The service instance.

\*/

@SuppressWarnings("unchecked")

public SystemService startService(String className) {

final Class<SystemService> serviceClass;

try {

serviceClass = (Class<SystemService>)Class.forName(className);

} catch (ClassNotFoundException ex) {

Slog.i(TAG, "Starting " + className);

throw new RuntimeException("Failed to create service " + className

+ ": service class not found, usually indicates that the caller should "

+ "have called PackageManager.hasSystemFeature() to check whether the "

+ "feature is available on this device before trying to start the "

+ "services that implement it", ex);

}

return startService(serviceClass);

}

/\*\*

\* Creates and starts a system service. The class must be a subclass of

\* {@link com.android.server.SystemService}.

\*

\* @param serviceClass A Java class that implements the SystemService interface.

\* @return The service instance, never null.

\* @throws RuntimeException if the service fails to start.

\*/

@SuppressWarnings("unchecked")

public <T extends SystemService> T startService(Class<T> serviceClass) {

try {

final String name = serviceClass.getName();

Slog.i(TAG, "Starting " + name);

Trace.traceBegin(Trace.TRACE\_TAG\_SYSTEM\_SERVER, "StartService " + name);

// Create the service.

if (!SystemService.class.isAssignableFrom(serviceClass)) {

throw new RuntimeException("Failed to create " + name

+ ": service must extend " + SystemService.class.getName());

}

final T service;

try {

Constructor<T> constructor = serviceClass.getConstructor(Context.class);

service = constructor.newInstance(mContext);

} catch (InstantiationException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service could not be instantiated", ex);

} catch (IllegalAccessException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (NoSuchMethodException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (InvocationTargetException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service constructor threw an exception", ex);

}

// Register it.

mServices.add(service);

// Start it.

try {

service.onStart();

} catch (RuntimeException ex) {

throw new RuntimeException("Failed to start service " + name

+ ": onStart threw an exception", ex);

}

return service;

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);

}

}

# 服务list

Sd

EntropyService：熵（shang）服务，用于产生随机数

PowerManagerService：电源管理服务

ActivityManagerService：最核心服务之一，Activity管理服务

TelephonyRegistry：电话服务，电话底层通知服务

PackageManagerService：程序包管理服务

AccountManagerService：联系人帐户管理服务

ContentService：内容提供器的服务，提供跨进程数据交换

LightsService：光感应传感器服务

BatteryService：电池服务，当电量不足时发广播

VibratorService：震动器服务

AlarmManagerService：闹钟服务

WindowManagerService：窗口管理服务

BluetoothService：蓝牙服务

InputMethodManagerService：输入法服务，打开关闭输入法

AccessibilityManagerService：辅助管理程序截获所有的用户输入，并根据这些输入给用户一些额外的反馈，起到辅助的效果，View的点击、焦点等事件分发管理服务

DevicePolicyManagerService：提供一些系统级别的设置及属性

StatusBarManagerService：状态栏管理服务

ClipboardService：粘贴板服务

NetworkManagementService：手机网络管理服务

TextServicesManagerService：

NetworkStatsService：手机网络状态服务

NetworkPolicyManagerService：

WifiP2pService：Wifi点对点直联服务

WifiService：WIFI服务

ConnectivityService：网络连接状态服务

ThrottleService：modem节流阀控制服务

MountService：磁盘加载服务，通常也mountd和vold服务结合

NotificationManagerService：通知管理服务，通常和StatusBarManagerService

DeviceStorageMonitorService：存储设备容量监听服务

LocationManagerService：位置管理服务

CountryDetectorService：检查当前用户所在的国家

SearchManagerService：搜索管理服务

DropBoxManagerService：系统日志文件管理服务（大部分程序错误信息）

WallpaperManagerService：壁纸管理服务

AudioService：AudioFlinger上层的封装的音量控制管理服务

UsbService：USB Host和device管理服务

UiModeManagerService：UI模式管理服务，监听车载、座机等场合下UI的变化

BackupManagerService：备份服务

AppWidgetService：应用桌面部件服务

RecognitionManagerService：身份识别服务

DiskStatsService：磁盘统计服务

SamplingProfilerService：性能统计服务

NetworkTimeUpdateService：网络时间更新服务

# SystemServiceRegistry

SystemServiceRegistry，这个类主要是用来缓存、注册、获取系统服务的。因为对它的缓存机制比较感兴趣，所以就研究下它的源码并记录下来

A:SystemServiceRegistry的static中注册map

registerService(Context.V1\_SERVICE, V1Manager.class,

new CachedServiceFetcher<V1Manager>() {

@Override

public V1Manager createService(ContextImpl ctx) {

IBinder b = ServiceManager.getService(Context.V1\_SERVICE);

return new V1Manager(ctx, IV1Manager.Stub.asInterface(b));

public static Object getSystemService(ContextImpl ctx, String name) {

ServiceFetcher<?> fetcher = SYSTEM\_SERVICE\_FETCHERS.get(name);

return fetcher != null ? fetcher.getService(ctx) : null;

}

## SystemServiceRegistry刚被导入内存时的分析

    先来看一段源码：

package android.app;

final class SystemServiceRegistry {

//用来保存所有Service的名字

private static final HashMap<Class<?>, String> SYSTEM\_SERVICE\_NAMES =

new HashMap<Class<?>, String>();

//用来保存所有的ServiceFetcher，后面会拿出ServiceFetcher源码来说

//现在只要知道它是从缓存中拿Service，缓存中没有就创建Service就可以了

private static final HashMap<String, ServiceFetcher<?>> SYSTEM\_SERVICE\_FETCHERS =

new HashMap<String, ServiceFetcher<?>>();

//记录缓存的大小

private static int sServiceCacheSize;

//注意构造方法是private的

private SystemServiceRegistry() { }

static{

//注册服务代码

registerService(Context.ACCESSIBILITY\_SERVICE, AccessibilityManager.class,

new CachedServiceFetcher<AccessibilityManager>() {

@Override

public AccessibilityManager createService(ContextImpl ctx) {

return AccessibilityManager.getInstance(ctx);

}});

//注册服务代码

registerService(Context.CAPTIONING\_SERVICE, CaptioningManager.class,

new CachedServiceFetcher<CaptioningManager>() {

@Override

public CaptioningManager createService(ContextImpl ctx) {

return new CaptioningManager(ctx);

}});

...//大量类似的注册服务代码

}

}

    在SystemServiceRegistry类刚被导入内存时，这些静态字段就会被初始化、静态字段就会被执行。SYSTEM\_SERVICE\_NAMES、SYSTEM\_SERVICE\_FETCHERS这两个字段不仅是静态的，还是final的，这说明这俩字段被初始化之后，对象就不能更改了但可以增删元素（因为是HashMap）。这两个字段，分别用来存储Service名字和获取Service实例的Fetcher对象的。

    SystemServiceRegistry的构造方法是私有的，说明它不能在类外创建对象，看一下该构造方法在类内的调用，发现也没有调用，这说明SystemServiceRegistry是不允许创建对象的。再看一下该类的其它方法和字段，发现都是静态的，这就可以确定SystemServiceRegistry类是被当做Util来使用的，并且不允许创建该类的实例。

    static代码块中大量调用了registerService方法，那就结合registerService方法的调用和声明来看一下：

...

static{

...

//registerService的调用

registerService(Context.WINDOW\_SERVICE, WindowManager.class,

new CachedServiceFetcher<WindowManager>() {

@Override

public WindowManager createService(ContextImpl ctx) {

return new WindowManagerImpl(ctx);

}});

...

}

...

//registerService的声明

private static <T> void registerService(String serviceName, Class<T> serviceClass,

ServiceFetcher<T> serviceFetcher) {

SYSTEM\_SERVICE\_NAMES.put(serviceClass, serviceName);

SYSTEM\_SERVICE\_FETCHERS.put(serviceName, serviceFetcher);

}

...

    先看一下registerService方法声明的参数：

| **参数** | **含义** |
| --- | --- |
| serviceName | service的名字，比如Context.WINDOW\_SERVICE |
| serviceClass | 该Service对象的Class类，比如WindowManager.class |
| serviceFetcher | ServiceFetcher对象 |

    registerService方法内的逻辑很简单，就是以serviceClass为key将serviceName存储到SYSTEM\_SERVICE\_NAMES这个HashMap中，以serviceName为key将ServiceFetcher对象对象存储到SYSTEM\_SERVICE\_FETCHERS这个HaspMap中。

    下面看一下，ServiceFetcher的声明及其实现类CachedServiceFetcher。如上述代码所示，registerService方法被调用时传入的第三个参数即为CachedServiceFetcher的对象。

package android.app;

final class SystemServiceRegistry {

...

private static int sServiceCacheSize;

...

static abstract interface ServiceFetcher<T> {

T getService(ContextImpl ctx);

}

static abstract class CachedServiceFetcher<T> implements ServiceFetcher<T> {

private final int mCacheIndex;

public CachedServiceFetcher() {

mCacheIndex = sServiceCacheSize++;

}

@Override

@SuppressWarnings("unchecked")

public final T getService(ContextImpl ctx) {

final Object[] cache = ctx.mServiceCache;

synchronized (cache) {

// Fetch or create the service.

Object service = cache[mCacheIndex];

if (service == null) {

service = createService(ctx);

cache[mCacheIndex] = service;

}

return (T)service;

}

}

public abstract T createService(ContextImpl ctx);

}

...

}

    从代码结构来看，ServiceFetcher为SystemServiceRegistry类的静态内部接口，定义了getService方法；CachedServiceFetcher为SystemServiceRegistry类的抽象静态内部类，实现ServiceFetcher接口并实现其定义方法getService方法。

    另外，CachedServiceFetcher的构造方法被调用时会将此时SystemServiceRegistry类中静态成员变量sServiceCacheSize的值赋给其final的成员变量mCacheIndex，然后sServiceCacheSize加1。结合上述static代码块中大量调用registerService方法时大量new CachedServiceFetcher对象作为第三个参数可知，SystemServiceRegistry类中静态成员变量sServiceCacheSize其实就是用来记录static代码块中注册了多少个服务的。同时还保证了每一个注册的服务都有一个唯一的下标mCacheIndex（与其他注册服务的下标不同）记录在ServiceFetcher对象内。

说一个小知识点：类的成员final非静态变量如果未初始化，那其实是可以在该类的构造方法初始化的。比如说，这里的mCacheIndex。

    SystemServiceRegistry类被导入内存时，能执行到的差不多就是如上所说了：静态成员变量的初始化，静态代码块以及静态代码块中执行到的registerService方法和给registerService传递参数时执行到的CachedServiceFetcher类的构造方法。

    SystemServiceRegistry类一般只会导入内存一次，然后就留在内存里了。这样就保证了static代码块只会执行一次，注册各个服务也只各执行一次，获取服务的ServiceFetcher对象也是一个服务对应一个。

## 2. 使用SystemServiceRegistry提供的方法创建缓存并获取服务

    通过上述对registerService方法功能的分析可知，SystemServiceRegistry存储下来的只是service名字和获取该service的ServiceFetcher对象，并没有直接存储该Service实现类的对象。那SystemServiceRegistry的缓存是在什么时候创建的？Service实现类的对象又是什么创建并存入到缓存的呢？

## 1. 创建缓存

    SystemServiceRegistry类的方法并不多，public修饰可被外界调用的也只有3个而已：createServiceCache、getSystemService、getSystemServiceName。根据方法名就很快找到了创建缓存的方法createServiceCache，下面具体看一下这个方法的声明：

public static Object[] createServiceCache() {

return new Object[sServiceCacheSize];

}

    超级简单的一个方法，就创建了一个Object数组而已。上面已经解释了sServiceCacheSize的作用是记录注册的所有服务的个数，而此处使用sServiceCacheSize来作为缓存数组的长度。

    关于创建缓存还有一点要说的，查看createServiceCache方法被调用的地方，发现它只在初始化ContextImpl的一个成员变量时被调用了,其它地方没调用：

package android.app;

class ContextImpl extends Context {

...

final Object[] mServiceCache = SystemServiceRegistry.createServiceCache();

...

}

    从这里就可以看出，SystemServiceRegistry类中的缓存对象并没有设置到SystemServiceRegistry类中，而是存储到了另一个类ContextImpl的成员变量上了。这个跟ThreadLocal中的静态内部类ThreadLocalMap的实例对象并没有设置在ThreadLocal类中，而是设置在了Thread类的成员变量上多多少少有些相似。

## 2. 获取Service实例

    上面说了，SystemServiceRegistry中可被外界调用的public方法很少，只有3个。上面说了用来创建缓存数组的createServiceCache方法，下面说一下用来获取Service实例的getSystemService方法：

public static Object getSystemService(ContextImpl ctx, String name) {

ServiceFetcher<?> fetcher = SYSTEM\_SERVICE\_FETCHERS.get(name);

return fetcher != null ? fetcher.getService(ctx) : null;

}

    也是超级简单的一个方法，从SystemServiceRegistry的静态且final的hashmap（SYSTEM\_SERVICE\_FETCHERS）中取出ServiceFetcher对象（ServiceFetcher对象在SystemServiceRegistry类被导入内存执行static代码块时已被初始化），然后调用其getService方法来获取Service实例。下面来看一下ServiceFetcher的实现类CachedServiceFetcher中的getService方法（CachedServiceFetcher类上面已经贴过，这里只贴出其方法getService）：

@Override

@SuppressWarnings("unchecked")

public final T getService(ContextImpl ctx) {

//从ContextImpl对象中获得缓存数组

final Object[] cache = ctx.mServiceCache;

//某个线程正在获取Service，其他获取Service线程在这里阻塞

synchronized (cache) {

//一个下标对应一个Service，从缓存数组中获取该下标下的Service

Object service = cache[mCacheIndex];

//如果缓存数组中还没有Service就要去创建，创建好后保存到缓存数组的相应下标下

if (service == null) {

service = createService(ctx);

cache[mCacheIndex] = service;

}

//返回该服务

return (T)service;

}

}

public abstract T createService(ContextImpl ctx);

    这个getService的逻辑基本就是缓存数组中获取不到该服务就先去创建该服务，创建好后将该服务存储到缓存数组中并返回该服务。下面看一下CachedServiceFetcher类中的createService方法：

static abstract class CachedServiceFetcher<T> implements ServiceFetcher<T> {

...

public abstract T createService(ContextImpl ctx);

}

    该方法是抽象的，那该方法的具体实现在哪儿呢？其实在上述的static代码中new CachedServiceFetcher对象时在后面的大括号中已经实现了createService方法。比如静态代码块中注册窗口服务时是这样的：

registerService(Context.WINDOW\_SERVICE, WindowManager.class,

new CachedServiceFetcher<WindowManager>() {

@Override

public WindowManager createService(ContextImpl ctx) {

return new WindowManagerImpl(ctx);

}});

    到此已经通过获取了Service实例。

## 总结

    这里作一个总结，以便对SystemServiceRegistry有一个整体的印象：

* SystemServiceRegistry类中ServiceFetcher对象是复用的，不同的ContextImpl对象获取相同的Service实例使用的同一个ServiceFetcher对象；
* 对Service缓存的数组是记录在ContextImpl中的，不同的ContextImpl对象缓存数组不同。
* Service缓存数组初始大小即为要注册的所有服务的个数，但数组每个位置上的Service实例均为空。只有在调用SystemServiceRegistry的getSystemService方法时发现没这个Service实例才会去创建这个Service实例，并存储到缓存数组中，保证再次调用getSystemService方法时返回缓存数组中的该实例；
* SystemServiceRegistry这种缓存机制的好处在于，不同ContextImpl对象使用不同的缓存数组，但不同的缓存数组获取相同的Service实例可以复用同一个ServiceFetcher对象；只在需要获取Service实例时才会去创建该Service实例，并保证以后需要时返回缓存中的该实例。

# System server里创建常见的几个thread

## UiThread.java-"android.ui"

C:\k\android-8.0.0\_r1\frameworks\base\services\core\java\com\android\server\UiThread.java

**super**(**"android.ui"**, Process.THREAD\_PRIORITY\_FOREGROUND, **false** */\*allowIo\*/*);

主要处理：

1. AMS UiHandler里show各种msg

2. DisplayManagerService里的overlay相关msg

3. PointerEventDispatcher inputevent相关

4. VoiceInteractionManagerService Voice交互

5. WindowManagerPolicy init操作

## BackgroundThread-"android.bg"

**super**(**"android.bg"**, android.os.Process.THREAD\_PRIORITY\_BACKGROUND);

主要处理这两件事情：＠mBgHandler.handleMessage()   
CHECK\_INTERNET\_PERMISSION\_MSG   
COLLECT\_PSS\_BG\_MSG

## FgThread-"android.fg"

super("android.fg", android.os.Process.THREAD\_PRIORITY\_DEFAULT, true /\*allowIo\*/);

主要用于：

1. AccountManagerService

2. BatteryStatsService

3. DreamManagerService

4. MountService

5. NetworkManagementService

6. PackageManagerService

7. usb相关（debug, device, portmanager）

8. WindowManagerService(screenshotApplicationsInner)

---------------------

作者：head\_first545

来源：CSDN

原文：https://blog.csdn.net/songyinzhong/article/details/52275031

版权声明：本文为博主原创文章，转载请附上博文链接！

## IoThread

**private** IoThread() {  
 **super**(**"android.io"**, android.os.Process.THREAD\_PRIORITY\_DEFAULT, **true** */\*allowIo\*/*);  
}

主要有用于：

1.BluetoothManagerService 相关操作

2.MountService里的obb操作

3.Tethering 网络共享（usb ／wifi／mobile？）

4.TvInputManagerService tv里channel session相关

## DisplayThread- android.display

**super**(**"android.display"**, Process.THREAD\_PRIORITY\_DISPLAY + 1, **false** */\*allowIo\*/*);

主要用于：

1. DisplayManagerService（display adapter,viewport ,event…）

2. InputManagerService (keyboard , input device …)

3. WindowManagerService 实例的创建

## "main thread"

mHandlerCheckers.add(**new** HandlerChecker(**new** Handler(Looper.getMainLooper()),  
 **"main thread"**, DEFAULT\_TIMEOUT));

# ServiceWatcher

<**bool name="config\_enableNetworkLocationOverlay" translatable="false"**>true</**bool**>  
*<!-- Package name providing network location support. Used only when  
 config\_enableNetworkLocationOverlay is false. -->*<**string name="config\_networkLocationProviderPackageName" translatable="false"**>@null</**string**>

<**string-array name="config\_locationProviderPackageNames" translatable="false"**>  
 *<!-- The standard AOSP fused location provider -->* <**item**>com.android.location.fused</**item**>  
</**string-array**>

服务替换着

LocationProviderProxy networkProvider = LocationProviderProxy.createAndBind(  
 mContext,  
 LocationManager.NETWORK\_PROVIDER,  
 NETWORK\_LOCATION\_SERVICE\_ACTION,  
 com.android.internal.R.bool.config\_enableNetworkLocationOverlay,  
 com.android.internal.R.string.config\_networkLocationProviderPackageName,  
 com.android.internal.R.array.**config\_locationProviderPackageNames**,  
 mLocationHandler);

**public** ServiceWatcher(Context context, String logTag, String action,  
 **int overlaySwitchResId**, **int** defaultServicePackageNameResId,  
 **int** initialPackageNamesResId, Runnable newServiceWork,  
 Handler handler) {

*// Whether to enable service overlay.***boolean** enableOverlay = resources.getBoolean(overlaySwitchResId);  
ArrayList<String> initialPackageNames = **new** ArrayList<String>();  
**if** (enableOverlay) {  
 *// A list of package names used to create the signatures.* String[] pkgs = resources.getStringArray(initialPackageNamesResId);  
 **if** (pkgs != **null**) initialPackageNames.addAll(Arrays.asList(pkgs));  
 mServicePackageName = **null**;  
 **if** (D) Log.d(mTag, **"Overlay enabled, packages="** + Arrays.toString(pkgs));  
} **else** {  
 *// The default package name that is searched for service implementation when overlay is  
 // disabled.* String servicePackageName = resources.getString(defaultServicePackageNameResId);  
 **if** (servicePackageName != **null**) initialPackageNames.add(servicePackageName);  
 mServicePackageName = servicePackageName;  
 **if** (D) Log.d(mTag, **"Overlay disabled, default package="** + servicePackageName);  
}  
mSignatureSets = getSignatureSets(context, initialPackageNames);

# android软硬件工作原理-振动系统

和大家分享自己对Android系统的一点认识：以马达为代表，来考究“**Android是如何一步步工作的。它从硬件设计，到Linux驱动，再到HAL，再到JNI，再到Framework，最后到被应用调用，这整套的流程到底是怎么样的！**

## 基本架构

Fsd

安卓总体架构是在 Linux内核基础上，增加硬件抽象层（HAL），运行库，java虚拟机，程序框架等组成的，具体如下图



安卓的应用程序是从application framework层架构上建立的。所有APK应用程序都是通过framework层来运行的。application framework是google写好的，除非自己深度定制，一般是不会更改这个层的。对于驱动开发来讲，我们要做的就是让framework层能认识并操作我们的硬件设备就OK了。因此我们关心主要有3个层面： linux Kernel层 HAL层 JNI层

1.       linuxKernel：是google在linux内核基础上，专门为移动设备优化后的内核，增加修改一些东西，但修改的不多，对于内核驱动来讲，基本没有修改，做过linux驱动开发的人应该很容易理解。

2.       HAL，硬件抽象层：简单来说，就是对Linux 内核驱动程序的封装，向上提供接口，屏蔽低层的实现细节。也就是说，把对硬件的支持分成了两层，一层放在用户空间（User Space），一层放在内核空间（Kernel Space），其中，硬件抽象层运行在用户空间。用户空间不属于内核，不必遵守GPL协议，各个厂商可以把与自己硬件设备相关，具有商业机密的一些代码放在HAL层。

3.       JNI层：提供java和底层C、C++的动态链接库的接口。我理解的是JNI就是一个代理，可以把C和C++生成的接口函数翻译成Java可用，提供给framework层。

## 震动服务

VibratorService即安卓震动服务，是Android系统中一个和硬件相关的服务，管理和驱动着设备的振动器。在Android手持设备，如手机，平板等，振动器是不可或缺的硬件设备，在给用户震动反馈的用户交互中发挥了举足轻重的作用。 既然是和硬件相关的服务，那么以Android系统的架构模式，VibratorService将对应以下架构



public class VibratorService extends IVibratorService.Stub implements InputManager.InputDeviceListener { }

接上代码： public class VibratorService extends IVibratorService.Stub implements InputManager.InputDeviceListener { }

上述代码可以看到，VibratorService继承了IVibratorService.Stub，也就是说，VibratorService直接支持Android IPC Binder通信，那么上层APP就可以通过AIDL使用VibratorService，驱动设备上的振动器，从而给用户震动的反馈。

VibratorService提供哪些接口可以给上层使用呢？看看接口IVibratorService：

public interface IVibratorService extends android.os.IInterface{

public boolean hasVibrator();

public void vibrate();

public void vibratePattern();

public void cancelVibrate(); }

上述代码可以看到，VibratorService提供上述四个接口给上层调用，hasVibrator()判断是当前设备否有支持振动器，然后通过vibrate()或vibratePattern()驱动震动器发起震动，前者只需要关心参数milliseconds，即震动的时间，时间结束后，震动停止，后者需要关心的参数有pattern和repeat，pattern是long型的数组，保存的是每次震动持续的时间，即milliseconds，repeat固然就是重复的次数。所以vibrate()是一次性振动器，vibratePattern()是重复多次震动。cancelVibrate()固然是取消震动了 ---------------------

### APP请求震动服务

以APP请求震动服务为例，看看这个过程，首先是APP调用Vibrator.vibrate()，然后IPC通信到VibratorService到：

马达的应用如何调用到马达服务的

接下来，我们分析一下如何获取马达服务的：即 *mVibrator= (Vibrator) getSystemService(VIBRATOR\_SERVICE)* 的工作原理。

**1. Context.java中的getSystemService()**

getSystemService()定义在frameworks/base/core/java/android/content/Context.java中，源码如下：

public abstract Object getSystemService(String name);

Context.java中的getSystemService() 是个抽象方法，它的实现在ContextImpl.java中。

**2. ContextImpl.java中的getSystemService()**

frameworks/base/core/java/android/app/ContextImpl.java中的 getSystemService() 源码如下：

1 @Override

2 public Object getSystemService(String name) {

3 ServiceFetcher fetcher = SYSTEM\_SERVICE\_MAP.get(name);

4 return fetcher == null ? null : fetcher.getService(this);

5 }

**3. ContextImpl.java中的SYSTEM\_SERVICE\_MAP**

SYSTEM\_SERVICE\_MAP是一个HashMap对象，它的相关代码如下：

复制代码

1 private static final HashMap<String, ServiceFetcher> SYSTEM\_SERVICE\_MAP =

2 new HashMap<String, ServiceFetcher>();

3

4 SYSTEM\_SERVICE\_MAP的初始化，是在ContextImpl.java通过static静态模块完成的。源码如下：

5 static {

6

7 ...

8

9 // 注册“传感器服务”

10 registerService(SENSOR\_SERVICE, new ServiceFetcher() {

11 public Object createService(ContextImpl ctx) {

12 return new SystemSensorManager(ctx.mMainThread.getHandler().getLooper());

13 }});

14

15 // 注册其它服务 ...

16

17 // 注册马达服务

18 registerService(VIBRATOR\_SERVICE, new ServiceFetcher() {

19 public Object createService(ContextImpl ctx) {

20 return new SystemVibrator();

21 }});

22

23 ...

24 }

复制代码

**说明**：在上面的static静态模块中，会通过registerService()注册一系列的服务，包括马达服务。注册服务是通过registerService()实现的，下面我们看看registerService()的定义。

复制代码

1 private static int sNextPerContextServiceCacheIndex = 0;

2 private static void registerService(String serviceName, ServiceFetcher fetcher) {

3 if (!(fetcher instanceof StaticServiceFetcher)) {

4 fetcher.mContextCacheIndex = sNextPerContextServiceCacheIndex++;

5 }

6 SYSTEM\_SERVICE\_MAP.put(serviceName, fetcher);

7 }

复制代码

    从中，我们知道，在registerService()中，会通过 SYSTEM\_SERVICE\_MAP.put(serviceName, fetcher) 将serviceName和fetcher添加到哈希表SYSTEM\_SERVICE\_MAP中。  
    对马达服务而言，添加到哈希表SYSTEM\_SERVICE\_MAP中的key-value中的**key是VIBRATOR\_SERVICE**，**value则是ServiceFetcher对象**；而且该匿名ServiceFetcher对象的createService()方法会“通过new SystemVibrator()”返回SystemVibrator对象。而SystemVibrator我们在前面已经介绍过了，它是马达服务对外提供接口的类。

OK，接着往下看。

**3. ContextImpl.java中的fetcher.getService(this)**

1 public Object getSystemService(String name) {

2 ServiceFetcher fetcher = SYSTEM\_SERVICE\_MAP.get(name);

3 return fetcher == null ? null : fetcher.getService(this);

4 }

我们已经知道SYSTEM\_SERVICE\_MAP是哈希表，通过SYSTEM\_SERVICE\_MAP.get(name)返回的是ServiceFetcher对象。  
由于fetcher不为null，所以，getSystemService()会返回fetcher.getService(this)。我们看看ServiceFetcher中getService()源码：

复制代码

1 static class ServiceFetcher {

2 int mContextCacheIndex = -1;

3

4 public Object getService(ContextImpl ctx) {

5 ArrayList<Object> cache = ctx.mServiceCache;

6 Object service;

7 synchronized (cache) {

8 if (cache.size() == 0) {

9

10 // “服务对象”缓冲

11 for (int i = 0; i < sNextPerContextServiceCacheIndex; i++) {

12 cache.add(null);

13 }

14 } else {

15 service = cache.get(mContextCacheIndex);

16 if (service != null) {

17 return service;

18 }

19 }

20 service = createService(ctx);

21 cache.set(mContextCacheIndex, service);

22 return service;

23 }

24 }

25

26 public Object createService(ContextImpl ctx) {

27 throw new RuntimeException("Not implemented");

28 }

29 }

复制代码

从中，我们发现，getService()实际上返回的是“通过createService(ctx)创建的service对象”。  
而在registerService()注册马达服务时，我们匿名实现了createService()方法：它实际上是通过 new SystemVibrator() 返回SystemVibrator对象。

至此，我们知道：**getSystemService(VIBRATOR\_SERVICE) 返回的是 SystemVibrator对象！**SystemVibrator前面已经分析过，这里就不再说明了

### Java应用层

应用层操作马达，是通过马达服务进行操作的。而马达服务是通过aidl实现的，aidl是Android进程间的通信方式。关于aidl的更多说明可以参考“[**Android Service总结06 之AIDL**](http://www.cnblogs.com/skywang12345/p/3165544.html)”。

马达服务涉及的主要文件如下：

1 frameworks/base/services/java/com/android/server/SystemServer.java

2 frameworks/base/services/java/com/android/server/VibratorService.java

3 frameworks/base/core/java/android/os/IVibratorService.aidl

4 frameworks/base/core/java/android/os/Vibrator.java

5 frameworks/base/core/java/android/os/SystemVibrator.java

下面，对这几个文件的功能进行简要说明。

**文件1: SystemServer.java**  
           它是系统服务，作用是**启动、管理系统服务**，包括“马达服务、Wifi服务、Activity管理服务”等等。  
           SystemServer是通过Zygote启动的，而Zygote又是在init中启动的，init则是kernel加载完毕之后启动的第一个进程。在这里，我们只需要知道“SystemServer是用来**启动/管理马达服务**即可。”

**文件2: IVibratorService.aidl**  
           它是马达服务对应的aidl配置文件。我们在aidl中定义了其它进程可以访问的外部接口；然后再通过VibratorService.java实现这些接口。

**文件3: VibratorService.java**  
           它是马达服务对应的aidl接口的实现程序。它实现IVibratorService.aidl的接口，从而实现马达服务；它的函数接口，是通过调用JNI层对应的马达控制函数来实现的。

**文件4: Vibrator.java**  
           它是马达服务开放给应用层的调用类。理论上讲，我们完全可以通过aidl直接调用马达服务，而不需要Vibrator.java类。但是！既然它存在，就肯定有它的理由。事实的确如此，Google之所以这么做。有以下几个原因：  
           **第一，提供统一而且方便的服务调用方式。**这里的**“统一”**，是指和所有其它的系统服务一样，我们调用服务时，需先通过getSystemService()获取服务，然后再调用服务的函数接口。这里的**“方便”**，是指若我们直接通过aidl调用，操作比较繁琐(若你用过aidl就会知道，需要先实现ServiceConnection接口以获取IBinder对象，然后再通过IBinder对象调用aidl的接口)； 而Vibrator.java封装之后的接口，将许多细节都隐藏了，非常便于应用者调用！  
          **第二，基于安全的考虑。**Vibrator.java封装隐藏了许多细节，而这些都是应用开发者不必要知道的。  
          **第三，Vibrator是抽象类。**它便于我们支持不同类型的马达：包括“将马达直接映射到文件”以及“将马达注册到输入子系统”中。

**文件5: SystemVibrator.java**  
         它是Vibrator.java的子类，实现了马达的服务接口。

--------------------- 本文来自 慢慢的燃烧 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/u010164190/article/details/51866419?utm_source=copy>

public class VibratorService extends IVibratorService.Stub

implements InputManager.InputDeviceListener {

@Override // Binder call

public void vibrate(int uid, String opPkg, long milliseconds, int usageHint,

IBinder token) {

if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)

!= PackageManager.PERMISSION\_GRANTED) {

throw new SecurityException("Requires VIBRATE permission");

}

Vibration vib = new Vibration(token, milliseconds, usageHint, uid, opPkg);

try {

synchronized (mVibrations) {

......

startVibrationLocked(vib);

}

} finally {

Binder.restoreCallingIdentity(ident);

}

}

}

看上述代码，首先是通过checkCallingOrSelfPermission()检测调用者APP是否已经有android.Manifest.permission.VIBRATE这个权限，所以APP要使用震动服务，需要在AndroidManifest.xml中声明权限android.Manifest.permission.VIBRATE的值android.permission.VIBRATE权限，权限保护等级为normal，即安装时授权。

随后，把token, milliseconds, usageHint, uid, opPkg封装到Vibration对象中，那么就把震动更好地抽象成一次震动对象，通过startVibrationLocked继续传输：

private void startVibrationLocked(final Vibration vib) {

......

if (vib.mTimeout != 0) {

doVibratorOn(vib.mTimeout, vib.mUid, vib.mUsageHint);

mH.postDelayed(mVibrationRunnable, vib.mTimeout);

} else {

// mThread better be null here. doCancelVibrate should always be

// called before startNextVibrationLocked or startVibrationLocked.

mThread = new VibrateThread(vib);

mThread.start();

}

}

这里有一个分支，判断依据是vib.mTimeout，vib.mTimeout什么时候等于0，什么时候不等于0呢？回顾Vibration的初始化过程：

public class VibratorService extends IVibratorService.Stub

implements InputManager.InputDeviceListener {

@Override // Binder call

public void vibrate(int uid, String opPkg, long milliseconds, int usageHint,

IBinder token) {

......

Vibration vib = new Vibration(token, milliseconds, usageHint, uid, opPkg);

......

}

}

private class Vibration implements IBinder.DeathRecipient {

private final IBinder mToken;

private final long mTimeout;

......

Vibration(IBinder token, long millis, int usageHint, int uid, String opPkg) {

this(token, millis, null, 0, usageHint, uid, opPkg);

}

private Vibration(IBinder token, long millis, long[] pattern,

int repeat, int usageHint, int uid, String opPkg) {

mToken = token;

mTimeout = millis;

......

}

在vibrate()这个方法中可以看到，实例化Vibration对象时，传入了milliseconds，在Vibration的构造方法中，把milliseconds赋值给mTimeout，也就是说mTimeout保存的是震动持续的时间。前文提到，启动震动有两个接口，如下：

public class SystemVibrator extends Vibrator {

......

public void vibrate(..... long milliseconds .....) {}

public void vibrate(..... long[] pattern, int repeat .....) {

}

一个是带milliseconds的作为参数的，另外一个是不带milliseconds参数的，前文有说明，前者是一次性震动，后者是重复多次震动，那么，也就是在下面的代码中：

private void startVibrationLocked(final Vibration vib) {

......

if (vib.mTimeout != 0) {

doVibratorOn(vib.mTimeout, vib.mUid, vib.mUsageHint);

mH.postDelayed(mVibrationRunnable, vib.mTimeout);

} else {

// mThread better be null here. doCancelVibrate should always be

// called before startNextVibrationLocked or startVibrationLocked.

mThread = new VibrateThread(vib);

mThread.start();

}

}

一次性震动走的是vib.mTimeout != 0的情况，重复震动的走的是!(vib.mTimeout != 0)的情况，那么这里的代码就很符合这个实际情况了，vib.mTimeout != 0时，直接调用doVibratorOn()启动震动，而重复震动时，通过VibrateThread线程实现重复震动的功能，这里就不在赘述这个重复的实现过程了。

接着追踪doVibratorOn()方法：

private void doVibratorOn(long millis, int uid, int usageHint) {

synchronized (mInputDeviceVibrators) {

......

if (vibratorCount != 0) {

......

for (int i = 0; i < vibratorCount; i++) {

mInputDeviceVibrators.get(i).vibrate(millis, attributes);

}

} else {

vibratorOn(millis);

}

}

}

这里会考虑多个振动器的情况，本文默认只有一个，则直接调用vibratorOn(millis)这个方法：

public class VibratorService extends IVibratorService.Stub

implements InputManager.InputDeviceListener {

native static void vibratorOn(long milliseconds);

}

如上述代码，vibratorOn()是一个native方法，也就通过JNI，调用到Native framework了，传递的参数只有一个，就是mills，震动持续的时间。

### JNI框架层

JNI(Java Native Interface)，中文是“Java本地接口”。

JNI是Java中一种技术，它存在的意义，是保证本地代码(C/C++代码)能在任何Java虚拟机下工作。简单点说，Java通过JNI接口，能够调用到C/C++代码。 关于“JNI的更多内容”，请参考“Android

JNI和NDK学习系列文章

通过 jniRegisterNativeMethods()，我们将method\_table中的方法注册到 com.android.server.VibratorService.java 中

以vibratorOff()来说，我们在VibratorService.java中调用vibratorOff()；实际上会调用到com\_android\_server\_VibratorService.cpp中的vibratorOff()函数；进一步会调用到vibrator\_off()函数，而vibrator\_off()是我们在 “HAL层的vibrator.c中的接口”。

VibratorService的JNI实现是frameworks/base/services/core/jni/com\_android\_server\_VibratorService.cpp中，继续看vibratorOn()这个方法：

static void vibratorOn(JNIEnv\* /\* env \*/, jobject /\* clazz \*/, jlong timeout\_ms)

{

if (gVibraDevice) {

int err = gVibraDevice->vibrator\_on(gVibraDevice, timeout\_ms);

......

} else {

ALOGW("Tried to vibrate but there is no vibrator device.");

}

}

1

#### **JNI如何和HAL关联方式**

我们先搞清楚：**JNI如何和HAL层代码关联起来的。**即com\_android\_server\_VibratorService.cpp是如何调用到vibrator.c中的代码的。  
实际上道理很简单，我们先将vibrator.c封装成.so库；然后在com\_android\_server\_VibratorService.cpp中导入该库，就可以调用vibrator.c的接口了。下面，看看Android中具体是如何做到的。

vibrator.c封装到libhardware\_legacy.so中的步骤

在hardware/libhardware\_legacy/vibrator/Android.mk中，会将vibrator.c添加到 LOCAL\_SRC\_FILES 变量中。  
hardware/libhardware\_legacy/vibrator/Android.mk源码如下：

LOCAL\_SRC\_FILES += vibrator/vibrator.c

在hardware/libhardware/Android.mk中，它会调用子目录的Android.mk并将它们导入当前的Android.mk中。  
hardware/libhardware/Android.mk源码如下：

复制代码

legacy\_modules := power uevent vibrator wifi qemu qemu\_tracing

SAVE\_MAKEFILES := $(call all-named-subdir-makefiles,$(modules))

LEGACY\_AUDIO\_MAKEFILES := $(call all-named-subdir-makefiles,audio)

include $(SAVE\_MAKEFILES)

...

LOCAL\_MODULE:= libhardware

include $(BUILD\_SHARED\_LIBRARY)

复制代码

在“我们编译Android系统”或“通过 mmm hardware/libhardwar进行模块编译”的时候，就会生成库libhardware.so；而且vibrator.c被包含在该库中。

**(02) 在 com\_android\_server\_VibratorService.cpp 对应的Android.mk中，会导入libhardware\_legacy.so。**  
com\_android\_server\_VibratorService.cpp 对应的frameworks/base/services/jni/Android.mk 的源码如下：

复制代码

LOCAL\_SRC\_FILES:= \

com\_android\_server\_VibratorService.cpp \

...

LOCAL\_SHARED\_LIBRARIES := \

libhardware \

...

LOCAL\_MODULE:= libandroid\_servers

include $(BUILD\_SHARED\_LIBRARY)

### HAL硬件抽象层

Android 封装了对底层驱动的调用，成为硬件抽象层。

HAL (Hardware Abstraction Layer), 又称为“硬件抽象层”。在Linux驱动中，我们已经将马达设为映射为文件了；而该HAL层的存在的意义，就是**“对设备文件进行操作，从而相当于硬件进行操作**”。HAL层的作用，**一是操作硬件设备**，二是**操作接口封装，外界能方便的使用HAL提供的接口直接操作硬件设备。**

这里有个对象gVibraDevice，gVibraDevice实际代表的就是振动器，有振动器，才会实例化gVibraDevice。gVibraDevice调用了vibrator\_on函数。先看gVibraDevice的定义，在hardware/libhardware/include/hardware/vibrator.h中：

typedef struct vibrator\_device {

struct hw\_device\_t common;

/\*\* Turn on vibrator

\*/

int (\*vibrator\_on)(struct vibrator\_device\* vibradev, unsigned int timeout\_ms);

/\*\* Turn off vibrator

\*/

int (\*vibrator\_off)(struct vibrator\_device\* vibradev);

} vibrator\_device\_t;

1

vibrator.h中定义了vibrator\_device\_t，以及它的函数vibrator\_on和vibrator\_off，再继续看调用gVibraDevice->vibrator\_on(gVibraDevice, timeout\_ms)如何发起震动。

vibrator\_on的具体实现是在hardware/libhardware/modules/vibrator/vibrator.c 中，先看如下代码：

static int vibra\_open(const hw\_module\_t\* module, const char\* id \_\_unused,

hw\_device\_t\*\* device \_\_unused) {

......

vibrator\_device\_t \*vibradev = calloc(1, sizeof(vibrator\_device\_t));

......

vibradev->vibrator\_on = vibra\_on;

vibradev->vibrator\_off = vibra\_off;

\*device = (hw\_device\_t \*) vibradev;

return 0;

}

上述C语言代码中，vibra\_open函数中在初始化震动器是就被调用了，vibra\_open在vibrator\_on的前面，在本文就不再赘述这个过程。通过上述代码可知，gVibraDevice->vibrator\_on(gVibraDevice, timeout\_ms)实际是调用了函数vibra\_on，继续看vibra\_on的实现：

static int vibra\_on(vibrator\_device\_t\* vibradev \_\_unused, unsigned int timeout\_ms)

{

/\* constant on, up to maximum allowed time \*/

return sendit(timeout\_ms);

}

---------------------

直接调用了函数sendit(timeout\_ms)，往下看：

static int sendit(unsigned int timeout\_ms)

{

int to\_write, written, ret, fd;

char value[20]; /\* large enough for millions of years \*/

fd = TEMP\_FAILURE\_RETRY(open(THE\_DEVICE, O\_RDWR));

to\_write = snprintf(value, sizeof(value), "%u\n", timeout\_ms);

written = TEMP\_FAILURE\_RETRY(write(fd, value, to\_write));

......

return ret;

}

**static const char** THE\_DEVICE[] = **"/sys/class/timed\_output/vibrator/enable"**;

在这里，打开设备，写入数据。到此，本文就不再往下继续分析这个过程了，

### 内核层-驱动层

就一个设备描述文件来搞的，如此简单

跟平台有关系

MTK6573

驱动实现移植 以MTK 6573平台为例  ./mediatek/platform/mt6573/kernel/drivers/vibrator/vibrator.c 操作设备

首先打开手机调试，连接USB，执行adb shell，进入/sys/devices/timed\_output/vibrator/ 执行

 echo "10000" > enable 发现手机在震动

# echo "10000" > enable

echo "10000"  > enable  10000 enable

执行 cat enable 可以查看当前震动时间剩余数：

# cat enable  cat enable  0

https://blog.csdn.net/jzjhome/article/details/71173144

但是我们的驱动在另一块芯片实现的

因此

### 驱动代码

#### 硬件设计

马达的震动原理很简单，给马达通电，马达就能震动。至于马达是如何工作，如何将电能转化为机械能，这不是我们关心的重点。但是，我们要需要了解如何控制马达的通电。在硬件上，我们是通过一个IO口(GPIO)去控制；对于马达而言，我们可以将IO理解为一个开关。当开关合上时，马达震动；开关断开，马达停止震动。

GPIO(General Purpose Input Output)，称为通用输入/输出。它可以被配置为中断、输入、输出等类型，从而对各个IO进行控制。对于马达而已，GPIO就相当于一个开关。下面看看硬件原理图中的马达部分，如下图：



注：上面原理图对应CPU是“三星A8”。不同平台的马达，马达的接法和GPIO都不一样；但原理都是类似的。

原理图中红线标注部分的含义：GPH3\_3是马达的GPIO。三星A8中有很多组GPIO，而马达对应和GPH3\_3连接。

#### 马达的驱动代码

知道马达的硬件设计之后，我们就可以进行Linux Driver开发工作，也就是编写马达的驱动。Linux的一个非常重要的特点，**一切都是文件**！而**我们进行Linux Driver开发的目的，就是将硬件设备映射成一个文件；然后，我们可以通过操作文件，来操作对应的硬件设备。**

OK！理解了驱动的作用和原理之后，我们接下来开发讲解马达的驱动开发。

我们知道，马达是通过GPIO去控制；接下来，我们就是找到马达对应的GPIO信息，然后控制该GPIO即可。

    通过马达的原理图，我们知道马达和GPH3\_3相连接。我们查阅“三星A8 的Datasheet”，查找GPH3\_3的相关信息。

*所谓Datasheet，就是CPU芯片的数据手册。*

*上面记载了CPU的功能特性和操作方式等信息。任何一个厂家在发布它的芯片时，都会提供对应的Datasheet给它的客户；客户根据Datasheet上面所描述的CPU的特性，就可以进行相关的开发(当然，实际开发中可能还需要芯片厂商的支持)。例如，国内手机都是采用MTK平台，对于MTK方案开发商来说，它要开发MTK6577的产品。那么首先，MTK原厂会提供一份MTK6577的BSP包，BSP包中包括了MTK6577的Datasheet，也就是该芯片的数据手册。方案开发商有任何关于MTK6577的问题，都可以查阅该Datasheet。*

三星A8的Datasheet中，关于GPH3\_3的信息如下：



**说明**：

(01) GPH3\_3对应CPU中的寄存器是GPH3CON[3]。

(02) [15:12] 表示寄存器的第12~15位，一个寄存器共32 bits。而第三列的 0000, 0001, 0010, 0011, 1111表示“寄存器取不同值的时候，该GPIO的功能”。

例如， 0000表示将该GPIO作为输入，0001表示将GPIO作为输出，1111表示将该GPIO作为中断。

    前面，我们已经说过，操作马达就是相当与将它作为一个开关操作。因此，我们需要将马达的GPIO设为“输入”类型；然后输入1，相当于开启马达；输入0，则是关闭马达！

下面，我们需要做的就是在Driver中将GPH3\_3(也就是GPH3CON[3])映射为一个文件节点，并将它配置为“输入”类型，即将GPH3CON[3]的寄存器值设为0000。

--------------------- 本文来自 慢慢的燃烧 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u010164190/article/details/51866419?utm\_source=copy

--------------------- 本文来自 慢慢的燃烧 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u010164190/article/details/51866419?utm\_source=copy

### 参考

Android System Server大纲之VibratorService

https://blog.csdn.net/myfriend0/article/details/55210074

s://blog.csdn.net/angle\_birds/article/details/16801533?utm\_source=copy

[Android之 震动（Vibrator）如何贯通Android系统 (从硬件设计 --> 驱动 --> HAL --> JNI --> Framework --> Application)](https://blog.csdn.net/u010164190/article/details/51866419?utm_source=copy)

**震动**

**外设架构，实践一把roid 6.0 如何添加完整的系统服务(app-framework-kernel)**

**https://blog.csdn.net/A8316124/article/details/78560507**

**https://blog.csdn.net/maochengtao/article/details/46849451**

# 服务设计

参考LocationManager

## TODO

PackFilter必须实施

## DOING

APP和applink 桥接好，是否需要

## 设计模式

WifiManager/USBManager/SensorManager/LocationManager

Brocast：太复杂了，我们需要一个简单化的

就是把之前的3288的app上面的binder客户端和服务端都封装一下就好了

我们不能一个协议就占用一个binder，全局共用就好了

base/services/core/java/com/android/server/CountryDetectorService.java

这个就非常适合啊啊啊，又简单，嘻嘻

Djilink->applink->actvivityLink->viewLink

## APP层

App框架：midware版本，sdk版本

Application级别，再做一次ActivityLink

Pack对象池实现？？

## FW-API层

## FW-java层

对象池技术下沉到这个地方了

### 序列化方案

大部分开发者会通过实现类的Serializable接口来实现序列化，Serializable接口是来自Java中的序列化接口，代码如下。

但是这种实现方法在性能方面不理想，Serializable在序列化时会产生大量的临时变量，在序列化的过程中会消耗更多的内存，从而引起频繁的GC。

Android也提供了一个序列化接口Parcelable。但Parcelable有个明显的缺点，不能使用在要将数据存储在磁盘上的情况，如永久性保存对象、保存对象的字节序列到本地文件中，因为Parcel本质上是为了更好地实现对象在IPC间传递，并不是一个通用的序列化机制，改变Parcel中任何数据的底层实现都可能导致之前的数据不可读取，所以此时还是建议使用Serializable。

可以总结出Serializable的性能较差，而Parcelable又有使用限制。

因此我们采用Parcel来做进程间通讯的序列化工具

## FW-native层

## 运行模块

Framework.jar

Service.jar

监控命令，dump状态，完善功能

## TODO

\frameworks\opt\net\wifi\service\Android.mk

测试用例

类似input子系统一样的，view的事件分发机制？？？

FindObj窗口？？？后台也需要执行的

类似广播的办法呢，回调onReview 方法，send广播，其实是一样的，也分前台广播（优先考虑这个吧）

和后台广播的

后台问题，还是解决不了的。

哎，搞了半天，直接用广播不就ok了？？？？

都需要上下文，是不是不太好呢

<https://www.jianshu.com/p/f057c460c77e>

eventbus设计得好，跨进程通讯呢，

模仿这种注解功能

BroadcastQueue，packQueue

### WifiService

## 参考

https://blog.csdn.net/sgzy001/article/details/54344739

# JNI层

源码目录：frameworks/base/services/core/jni/Android.mk

编译脚本：frameworks/base/services/Android.mk

定义了所有的jni编译成为libandroid\_servers.so

## 如何加载

系统：build/make/target/product/base.mk定义模块依赖

*// Initialize native services.*System.loadLibrary(**"android\_servers"**);

### onload

frameworks/base/services/core/jni/onload.cpp

#include **"JNIHelp.h"**#include **"jni.h"**#include **"utils/Log.h"**#include **"utils/misc.h"**

**namespace** android {

**int** register\_android\_server\_InputManager(JNIEnv\* env);

};  
  
**using namespace** android;  
  
**extern "C"** jint JNI\_OnLoad(JavaVM\* vm, **void**\* */\* reserved \*/*)  
{  
 JNIEnv\* env = NULL;  
 jint result = -1;  
  
 **if** (vm->GetEnv((**void**\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) {  
 ALOGE(**"GetEnv failed!"**);  
 **return** result;  
 }  
 ALOG\_ASSERT(env, **"Could not retrieve the env!"**);

register\_android\_server\_InputManager(env);

**return** JNI\_VERSION\_1\_4;  
}

### com\_android\_server\_input\_InputManagerService.cpp

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

#### static 部分

##### gInputManagerMethods

**static const** JNINativeMethod gInputManagerMethods[] = {  
 */\* name, signature, funcPtr \*/* { **"nativeInit"**,  
 **"(Lcom/android/server/input/InputManagerService;Landroid/content/Context;Landroid/os/MessageQueue;)J"**,  
 (**void**\*) nativeInit },  
 { **"nativeStart"**, **"(J)V"**,  
 (**void**\*) nativeStart },

##### register\_android\_server\_InputManager

**int** register\_android\_server\_InputManager(JNIEnv\* env) {  
 **int** res = jniRegisterNativeMethods(env, **"com/android/server/input/InputManagerService"**,  
 gInputManagerMethods, NELEM(gInputManagerMethods));  
 (**void**) res; *// Faked use when LOG\_NDEBUG.* LOG\_FATAL\_IF(res < 0, **"Unable to register native methods."**);

*// Callbacks*jclass clazz;  
FIND\_CLASS(clazz, **"com/android/server/input/InputManagerService"**);

GET\_METHOD\_ID(gServiceClassInfo.notifyANR, clazz,  
 **"notifyANR"**,  
 **"(Lcom/android/server/input/InputApplicationHandle;Lcom/android/server/input/InputWindowHandle;Ljava/lang/String;)J"**);

**return** 0;  
}

#### JNI调用

##### ptr = static jlong nativeInit

**static** jlong nativeInit(JNIEnv\* env, jclass */\* clazz \*/*,  
 jobject serviceObj, jobject contextObj, jobject messageQueueObj) {  
 NativeInputManager\* im = **new** NativeInputManager(contextObj, serviceObj,  
 messageQueue->getLooper());  
 im->incStrong(0);  
 **return reinterpret\_cast**<jlong>(im);  
}

##### nativeStart

**static void** nativeStart(JNIEnv\* env, jclass */\* clazz \*/*, jlong ptr) {  
 NativeInputManager\* im = **reinterpret\_cast**<NativeInputManager\*>(ptr);  
  
 status\_t result = im->getInputManager()->start();  
 **if** (result) {  
 jniThrowRuntimeException(env, **"Input manager could not be started."**);  
 }  
}

##### start

##### NativeInputManager::notifyANR

nsecs\_t NativeInputManager::notifyANR(**const** sp<InputApplicationHandle>& inputApplicationHandle,  
 **const** sp<InputWindowHandle>& inputWindowHandle, **const** String8& reason) {  
  
 jlong newTimeout = env->CallLongMethod(mServiceObj,  
 gServiceClassInfo.notifyANR, inputApplicationHandleObj, inputWindowHandleObj,  
 reasonObj);  
   
 **return** newTimeout;  
}

## JNI的架构和实现方式

# 业务分析

## protected-broadcast

### ****引****言

注：本文中提及的“广播（Broadcast）”，“广播事件”和“Action”的意思大致相同。发送广播（sendBroadcast）也是发送一个指定的action给BroadcastReceiver。在本文中不严格区分“广播”和“Action”，除非有地方特别说明。

#### 概念

普通广播：任何app都可以发送此action的广播给MyReceiver1。场景：普通app(如果一个action是不受protected-broadcast保护，并且使用此action的<receiver>组件[称之为MyReceiver1]没有system或者signature权限保护的话)

protected-broadcast：保护广播事件（Action）不被滥用的。这是android 系统应用基本的安全要求。 可以不用声明system或者signature权限去保护receiver。当然，从安全的角度来说，给每一个组件设置适当的system或者signature权限是更好的。

frameworks\base\core\res中的AndroidManifest.xml文件中看到过这样的代码：

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

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

<protected-broadcast android:name="android.intent.action.USER\_PRESENT" />

<protected-broadcast android:name="android.intent.action.TIME\_SET" />

<protected-broadcast android:name="android.intent.action.TIME\_TICK" />

<protected-broadcast android:name="android.intent.action.TIMEZONE\_CHANGED" />

<protected-broadcast android:name="android.intent.action.DATE\_CHANGED" />

<protected-broadcast android:name="android.intent.action.PRE\_BOOT\_COMPLETED" />

<protected-broadcast android:name="android.intent.action.BOOT\_COMPLETED" />

<protected-broadcast android:name="android.intent.action.PACKAGE\_INSTALL" />

<protected-broadcast android:name="android.intent.action.PACKAGE\_ADDED" />

(略)

#### 针对protected-broadcast做一些细节方面的介绍

Sdf

* （1）protected-broadcast的适用范围：哪些应用可以使用这个特性？有系统签名的非系统app，能够使用这个特性吗？
* （2）Android 系统是如何阻止第三方app发送被 protected-broadcast 保护的广播（Action）的？
* （3）protected-broadcast 是对Activity（startActivity）、Receiver（sendBroadcast）、Service（startService）都有效吗？
* （4）如何判断一个action是否是受保护的？

### 源码解析

#### AndroidManifest解析：适用范围

frameworks\base\core\java\android\content\pm

PackageParser.java中定义的常量TAG\_PROTECTED\_BROADCAST：

private static final String TAG\_PROTECTED\_BROADCAST = "protected-broadcast";

PackageParser的内部类Package中的变量protectedBroadcasts：

public ArrayList<String> protectedBroadcasts;

解析 apk 中的 AndroidManifest.xml 的代码在 PackageParser.java 中的parseBaseApkCommon()。

private Package parseBaseApkCommon(Package pkg, Set<String> acceptedTags, Resources res,

XmlResourceParser parser, int flags, String[] outError) throws XmlPullParserException,

IOException {

//（略）

} else if (tagName.equals(TAG\_PROTECTED\_BROADCAST)) {

// 解析<protected-broadcast>标签

// 获取 AndroidManifestProtectedBroadcast 的属性值

// 这里 res 是 Resources res， sa 是 TypedArray sa。

sa = res.obtainAttributes(parser,

com.android.internal.R.styleable.AndroidManifestProtectedBroadcast);

// getNonResourceString()确保protected broadcast 的name是从xml中得到的，而不能是引用strings.xml中的字符串。

// protected broadcast 的name在编译之后不能改变，除非重新编译。

// AndroidManifestProtectedBroadcast\_name是属性值的索引，其值为0。

// name 为获取到的受保护的action名。

String name = sa.getNonResourceString(

com.android.internal.R.styleable.AndroidManifestProtectedBroadcast\_name);

sa.recycle();

if (name != null && (flags & PARSE\_IS\_SYSTEM) != 0) {

// PARSE\_IS\_SYSTEM表明只在解析系统app时，protected broadcast才会有效

// pkg 是内部类 Package 的实例

if (pkg.protectedBroadcasts == null) {

pkg.protectedBroadcasts = new ArrayList<String>();

}

if (!pkg.protectedBroadcasts.contains(name)) {

// 所有的受保护的action都放到protectedBroadcasts

pkg.protectedBroadcasts.add(name.intern());

}

}

// 此标签<protected-broadcast>解析完毕，跳到END\_TAG：'/>' 或者 '</protected-broadcast>'

XmlUtils.skipCurrentTag(parser);

}

略

}

从(flags & PARSE\_IS\_SYSTEM) != 0这个判断可知，**可以使用<protected-broadcast>标签的app只能是系统app**。**如果不是系统app，那么此标签将被忽略**。

**哪些app是系统app呢？**   
有两类：   
\* 一类是uid为 android.uid.system，android.uid.phone，android.uid.log，android.uid.nfc，android.uid.bluetooth，android.uid.shell的app是系统app。

* 另一类是指定目录中的app是系统app，“指定目录”包括/vendor/overlay，/system/framework，/system/priv-app，/system/app，/vendor/app，/oem/app。

解析这两类apk的时候，用到了PARSE\_IS\_SYSTEM标志位

如果安装一个系统签名app，**新报名**，就不会用到PARSE\_IS\_SYSTEM标志来解析apk，所以**在这种情况下，有系统签名但不是系统app，不能使用<protected-broadcast>特性**。

#### ****阻止第三方app发送****protected-broadcast****广播（Action）****

分两个部分：

* 第一部分：受保护的广播是如何保存到PackageManagerService（简称为PMS）的，即protected-broadcast从系统app到PMS。
* 第二部分：什么时候检查广播（action）是不是受保护的。

##### mProtectedBroadcasts

* scanPackageDirtyLI() {
* if (pkg.protectedBroadcasts != null) {
* N = pkg.protectedBroadcasts.size();
* for (i=0; i<N; i++) {
* mProtectedBroadcasts.add(pkg.protectedBroadcasts.get(i));
* }
* }
* }

<protected-broadcast>保存到PMS后，PMS提供了一个接口isProtectedBroadcast()供其他应用调用，目前只是AMS在调用。

从下面的代码中可以看到，mProtectedBroadcasts中的action是受保护的，除此之外，某些action名字是受保护的或者说是被禁止的，第三方app不能乱发，例如，以android.net.netmon.lingerExpired开头的action。

@Override

public boolean isProtectedBroadcast(String actionName) {

synchronized (mPackages) {

if (mProtectedBroadcasts.contains(actionName)) {

return true;

} else if (actionName != null) {

// // TODO: remove these terrible hacks

if (actionName.startsWith("android.net.netmon.lingerExpired")

|| actionName.startsWith("com.android.server.sip.SipWakeupTimer")

|| actionName.startsWith("com.android.internal.telephony.data-reconnect")

|| actionName.startsWith("android.net.netmon.launchCaptivePortalApp")) {

return true;

}

}

}

return false;

}

##### broadcastIntentLocked

ActivityManagerService

-> App中调用sendBroadcast()

-> sendBroadcast() @ ContextImpl

-> broadcastIntent() @ ActivityManagerService

-> broadcastIntentLocked() {

final String action = intent.getAction();

isProtectedBroadcast = AppGlobals.getPackageManager().isProtectedBroadcast(action);

if (!isCallerSystem) {// 如果调用者不是系统app，即调用sendBroadcast()的不是系统app

if (isProtectedBroadcast) {//action是被保护的广播，那么将抛出异常

String msg = "Permission Denial: not allowed to send broadcast "

+ action + " from pid="

+ callingPid + ", uid=" + callingUid;

throw new SecurityException(msg);

}

}

由上面可知，如果调用sendBroadcast()的不是系统app，并且广播action是受保护的，那么将抛出SecurityException异常。

### protected-broadcast****都有效吗？****

**是对Activity（startActivity）、Receiver（sendBroadcast）、Service（startService）**

从上面的分析来看，不是都有效。   
而且从名字**protected-broadcast**直观的来看（猜），也是**只保护广播action的，不保护 startActivity 和 startService 的action。**   
都是action，待遇怎么差那么多呢？

### ****如何为系统app增加一个广播action****

加这个字段就好<protected-broadcast android:name="isentek.intent.action.UPDATE\_DATA" />

作为系统app

### ****REF****

[protected-broadcast 的一些细节](ttps://blog.csdn.net/u013553529/article/details/78409382)

01-08 07:24:57.022 1572-1572/com.android.settings W/ContextImpl: Calling a method in the system process without a qualified user: android.app.ContextImpl.sendBroadcast:878 android.content.ContextWrapper.sendBroadcast:421 com.isentek.calibrationservice.CalibrationService.onSensorChanged:663 android.hardware.SystemSensorManager$SensorEventQueue.dispatchSensorEvent:699 android.os.MessageQueue.nativePollOnce:-2

## 系统应用是如何解析出来的

**哪些app是系统app呢？**   
有两类：   
\* 一类是uid为 android.uid.system，android.uid.phone，android.uid.log，android.uid.nfc，android.uid.bluetooth，android.uid.shell的app是系统app。

* 另一类是指定目录中的app是系统app，“指定目录”包括/vendor/overlay，/system/framework，/system/priv-app，/system/app，/vendor/app，/oem/app。

解析这两类apk的时候，用到了PARSE\_IS\_SYSTEM标志位

## Calling a method in the system process without a qualified

public void sendBroadcast(Intent intent) {

warnIfCallingFromSystemProcess();

private void warnIfCallingFromSystemProcess() {

if (Process.myUid() == Process.SYSTEM\_UID) {

Slog.w(TAG, "Calling a method in the system process without a qualified user: "

+ Debug.getCallers(5));

}

}

# android 通信

Application-四大组件通信

## Activity通信

隐式调用，有利于降低发送者和接收者之间的耦合，它一般用在没有明确指出目标组件名称的前提下

### Intent Filter match过程

#### 使用规范

隐式 Intent

Intent intent = new Intent();

intent.setAction("com.wooyun.test"); startActivity(intent);

* a. action匹配规则：要求intent中的action 存在 且 必须和过滤规则中的其中一个相同 区分大小写；
* b. category匹配规则：系统会默认加上一个android.intent.category.DEAFAULT，所以intent中可以不存在category，但如果存在就必须匹配其中一个；
* c. data匹配规则：data由两部分组成，mimeType和URI，要求和action相似。如果没有指定URI，URI但默认值为content和file（schema）

### 源码分析

https://www.yuanmas.com/info/9ezZpgB0y6.html

主线流程：先match action, 再match data, 最后match category



# 出厂设置

https://blog.csdn.net/wdaming1986/article/details/11988531

<http://blog.csdn.net/mirkerson/article/details/24322207>

<http://blog.csdn.net/woshing123456/article/details/44524051>

## ACTION\_MASTER\_CLEAR

Intent intent = **new** Intent(Intent.ACTION\_MASTER\_CLEAR);  
intent.addFlags(Intent.***FLAG\_RECEIVER\_FOREGROUND***);  
intent.putExtra(Intent.EXTRA\_REASON, **"MasterClearConfirm"**);  
intent.putExtra(Intent.EXTRA\_WIPE\_EXTERNAL\_STORAGE, **mEraseSdCard**);  
getActivity().sendBroadcast(intent);

**public static final** String ***ACTION\_MASTER\_CLEAR*** = **"android.intent.action.MASTER\_CLEAR"**;

**final** PersistentDataBlockManager pdbManager = (PersistentDataBlockManager)  
 getActivity().getSystemService(Context.PERSISTENT\_DATA\_BLOCK\_SERVICE);

## MasterClearReceiver

<**receiver android:name="com.android.server.MasterClearReceiver"  
 android:permission="android.permission.MASTER\_CLEAR"**>  
 <**intent-filter  
 android:priority="100"** >  
 *<!-- For Checkin, Settings, etc.: action=FACTORY\_RESET -->* <**action android:name="android.intent.action.FACTORY\_RESET"** />  
 *<!-- As above until all the references to the deprecated MASTER\_CLEAR get updated to  
 FACTORY\_RESET. -->* <**action android:name="android.intent.action.MASTER\_CLEAR"** />  
  
 *<!-- MCS always uses REMOTE\_INTENT: category=MASTER\_CLEAR -->* <**action android:name="com.google.android.c2dm.intent.RECEIVE"** />  
 <**category android:name="android.intent.category.MASTER\_CLEAR"** />  
 </**intent-filter**>  
</**receiver**>

**final boolean** shutdown = intent.getBooleanExtra(**"shutdown"**, **false**);  
**final** String reason = intent.getStringExtra(Intent.EXTRA\_REASON);  
**final boolean** wipeExternalStorage = intent.getBooleanExtra(  
 Intent.EXTRA\_WIPE\_EXTERNAL\_STORAGE, **false**);  
**final boolean** forceWipe = intent.getBooleanExtra(Intent.EXTRA\_FORCE\_MASTER\_CLEAR, **false**)  
 || intent.getBooleanExtra(Intent.EXTRA\_FORCE\_FACTORY\_RESET, **false**);  
  
Slog.w(TAG, **"!!! FACTORY RESET !!!"**);  
*// The reboot call is blocking, so we need to do it on another thread.*Thread thr = **new** Thread(**"Reboot"**) {  
 @Override  
 **public void** run() {  
 **try** {  
 RecoverySystem.rebootWipeUserData(context, shutdown, reason, forceWipe);  
 Log.wtf(TAG, **"Still running after master clear?!"**);  
 } **catch** (IOException e) {  
 Slog.e(TAG, **"Can't perform master clear/factory reset"**, e);  
 } **catch** (SecurityException e) {  
 Slog.e(TAG, **"Can't perform master clear/factory reset"**, e);  
 }  
 }  
};  
  
**if** (wipeExternalStorage) {  
 *// thr will be started at the end of this task.* **new** WipeAdoptableDisksTask(context, thr).execute();  
} **else** {  
 thr.start();  
}

### RecoverySystem.rebootWipeUserData

**public static void** rebootWipeUserData(Context context, **boolean** shutdown, String reason,  
 **boolean** force) **throws** IOException {

#### sendOrderedBroadcastAsUser

**final** ConditionVariable condition = **new** ConditionVariable();  
  
Intent intent = **new** Intent(**"android.intent.action.MASTER\_CLEAR\_NOTIFICATION"**);  
intent.addFlags(Intent.FLAG\_RECEIVER\_FOREGROUND  
 | Intent.FLAG\_RECEIVER\_INCLUDE\_BACKGROUND);  
context.sendOrderedBroadcastAsUser(intent, UserHandle.SYSTEM,  
 android.Manifest.permission.MASTER\_CLEAR,  
 **new** BroadcastReceiver() {  
 @Override  
 **public void** onReceive(Context context, Intent intent) {  
 condition.open();  
 }  
 }, **null**, 0, **null**, **null**);  
  
*// Block until the ordered broadcast has completed.*condition.block();

#### bootCommand

组装命令

String shutdownArg = **null**;  
**if** (shutdown) {  
 shutdownArg = **"--shutdown\_after"**;  
}  
  
String reasonArg = **null**;  
**if** (!TextUtils.isEmpty(reason)) {  
 reasonArg = **"--reason="** + sanitizeArg(reason);  
}  
  
**final** String localeArg = **"--locale="** + Locale.getDefault().toLanguageTag() ;  
bootCommand(context, shutdownArg, **"--wipe\_data"**, reasonArg, localeArg);

## data/dalvik-cache

dalvik-cache名词解释： 在系统data/dalvik-cache文件夹里有很多安装卸载文件（优化过的字节码），这些文件是当你安装好一个应用程序后，系统会自动生成的一个优化过的字节码文件，但是当你频繁安装卸载某些应用软件后可能对应字节码文件不会同时删除，也即是残留的垃圾，这时就需要用到缓存清理助手来帮助增加您手机的可用空间^0^ 特别提示： 清理Dalvik缓存时，需要Root权限 。

安卓手机程序越装越多，系统提示内存空间不足了吧？卸载一些程序也不会增加多少可用空间

系统缓存文件和卸载定制程序留下来的无用垃圾，可以放心全部删除，系统所需文件重启后能自动生成的。删除后重启手机的时候，时间有点久，大概2－3分钟，期间有段时间为黑屏状态，应用不能正常打开，不要担心，一会系统就加载完成，启动加载期间，不要有任何操作，要耐心等待......系统启动加载完毕，一切正常。

至此，宣布成功！

[](http://images.cnitblog.com/blog/363476/201402/082037504515476.jpg)

[](http://images.cnitblog.com/blog/363476/201402/082037518684475.png)

# BackupManagerService

https://blog.csdn.net/GooHong/article/details/8026045

其服务占到0.6秒～0.8秒

 Android2.2以后的备份服务功能可以允许用户备份应用数据到云存储中，即当应用执行了工厂服务或转换到一个新的平台上时,如果备份的应用需要重新安装,系统就自动恢复原先备份的数据到本地。

 GOOGLE目前为Android框架系统提供了一个云存储服务BackupTransportService和一个GOOGLE IBackupTransport接口对象，不过要使用这个云存储，需要首先向该服务登记你要备份的应用，获得一个BackupService Key，并包括在你要备份的应用的manifest工程文件中。当然你也可以实现你自己的云存储服务和一个备份IBackupTransport对象。Android框架系统也提供了一个供测试的本地IBackupTransport接口对象。

使用场景:**比如设备factory reset之后,app的设置和数据会由云端恢复**; 比如重新安装app后,可以从云端恢复之前的数据.

使用前提:App需在Manifest中指定BackAgent,并且需要由设备制造商或者服务提供商提供云存储服务;或者app也可以使用Google提供的Backup Service,但需要app在Google Service上注册并获得一个唯一的key,将key注册进manifest,作为服务使用权限的凭证.

备份已经做完了 现在来看看结果

1. bash-4.0# cd /data/backup/

# 初探boot.art与boot.oat

boot.art与boot.oat与其说是ART虚拟机的两种执行格式，不如说他俩就是ART虚拟机的一部分！！！ART离开了这两个文件，也就无法启动了。

boot.art是一个img文件，而boot.oat文件可以将其理解为ART虚拟机的启动类。

这两个文件是dex2oat命令将Android系统必须的的jar包编译生成的，这两个文件相互联系，缺一不可，boot.art这个img文件直接被映射到ART虚拟机的堆空间中，包含了boot.oat中的某些对象实例以及函数地址。

## 产生过程

删除/data/dalvik-cache/arm的boot.art和boot.oat，

Reboot

adb logcat | grep dex2oat

观察生成日志

root@cs500c:/data/dalvik-cache/arm # rm system@framework@boot.oat

root@cs500c:/data/dalvik-cache/arm # reboot

shell@cs500c:logcat | grep dex2oat

|  |
| --- |
| I/dex2oat ( 379): /system/bin/dex2oat --image=/data/dalvik-cache/arm/system@framework@boot.art --dex-file=/system/framework/core-libart.jar --dex-file=/system/  framework/conscrypt.jar --dex-file=/system/framework/okhttp.jar --dex-file=/system/framework/core-junit.jar --dex-file=/system/framework/bouncycastle.jar --dex-  file=/system/framework/ext.jar --dex-file=/system/framework/framework.jar --dex-file=/system/framework/telephony-common.jar --dex-file=/system/framework/voip-co  mmon.jar --dex-file=/system/framework/ims-common.jar --dex-file=/system/framework/mms-common.jar --dex-file=/system/framework/android.policy.jar --dex-file=/sys  tem/framework/apache-xml.jar --oat-file=/data/dalvik-cache/arm/system@framework@boot.oat --instruction-set=arm --instruction-set-features=div --base=0x708ea000  --runtime-arg -Xms64m --runtime-arg -Xmx64m --image-classes=/system/etc/preloaded-classes |

boot.oat和boot.art文件依赖的dalvik的dex来自于BOOTCLASSPATH中指定的jar包。

|  |
| --- |
| shell@cs500c:/ $ echo ${BOOTCLASSPATH}  /system/framework/core-libart.jar:/system/framework/conscrypt.jar:/system/framework/okhttp.jar:/system/framework/core-junit.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/**framework**.jar:/system/framework/telephony-common.jar:/system/framework/voip-common.jar:/system/framework/ims-common.jar:/system/framework/mms-common.jar:/system/framework/android.policy.jar:/system/framework/apache-xml.jar |

[初探boot.art与boot.oat](http://www.iloveandroid.net/2015/12/19/AndroidART-2/)

# ROOT

# FactoryTest

Sd

<https://blog.csdn.net/thl789/article/details/8053574>

本文简析Android内置的Factory Test（有测试模式/工厂模式/工程模式等叫法）框架

Android系统中在com.android.server.SystemServer(loc:frameworks/base/services/java/)中对FactoryTest的级别做了定义：

public static final int FACTORY\_TEST\_OFF =0;

public static final int FACTORY\_TEST\_LOW\_LEVEL = 1;

public static final int FACTORY\_TEST\_HIGH\_LEVEL= 2;

  FACTORY\_TEST\_OFF是正常模式；

  FACTORY\_TEST\_LOW\_LEVEL是工程模式做在较低的级别，也就是运行工程模式的时候，很多的service并不需要启动；

  FACTORY\_TEST\_HIGH\_LEVEL是工程模式做在较高的级别，也就是运行工程模式的时候，基本的Android运行环境是同正常模式是相同的。在原生的实现中，与正常模式相比除了ActivityManagerService外并没有太多不同。

# Direct Boot

Android 7.0新增了DirectBoot功能，AOSP中为实现该功能修改了开机代码流程，并且这部分流程并未根据设备是否支持DirectBoot做区分，只是流程上做了兼容，确保不支持DirectBoot的设备在这套流程下也能正常开机。在这套流程下，用户解锁后才可进入非directBootAware应用，包括Launcher。

当手机已经通电开机但是用户并有解锁锁屏的时候，Android N运行于一个安全的模式，也就是Dierect Boot模式。

​ 为了支持Dierect Boot模式，系统提供了两个存储数据的地方：

1.Credential encrypted storage，默认存储数据的地方，仅在用户解锁手机后可用。

2.Device encrypted storage，主要对应的就是Direct Boot使用的存储空间。在Direct Boot模式下和用户解锁手机后都可以使用的存储空间。

相关文件如下：

drwxrwxr-x 15 system system 3488 2019-01-22 11:16 system

drwxrwx--- 3 system system 3488 2019-01-21 14:23 system\_ce

drwxrwx--- 3 system system 3488 2019-01-21 13:22 system\_de

drwx--x--x 2 system system 3488 2019-01-21 13:22 user

drwx--x--x 3 system system 3488 2019-01-21 13:22 user\_de

drwxrwx--t 32 system misc 3488 2019-01-21 13:22 misc

drwxrwx--t 2 system misc 3488 2019-01-21 13:22 misc\_ce

drwxrwx--t 3 system misc 3488 2019-01-21 13:22 misc\_de

​ 系统把部分系统数据和已经注册了相关权限的Apps的数据保存在device-encrypted store 。其他的数据默认保存到credential-encrypted store。当手机开机，首先进入一个Dierect Boot的模式，在这个模式下只可以访问device-encrypted store下的数据，无法访问credential-encrypted store下的数据。当用户解锁后就都可以访问了。

​ 一般情况下，应用是无法在Direct Boot模式下运行的如果需要某个app能够在Direct Boot模式下运行，需要注册相关APP的组件。通常需要在这个模式下运行的app：1.计划通知的应用，例如Clock2.重要的用户通知的应用，例如sms3.提供无障碍服务的应用，例如Talkback

​ 应用组件申请在Direct Boot模式下运行：在AndroidManinfest.xml中设置 android:directBootAware="true"。

​ 应用访问device encrypted storage：创建Context.createDeviceEncryptedStorageContext().然后通过这个Context来使用device encrypted storage 的存储空间。

应用获取解锁的通知：

监听广播ACTION\_USER\_UNLOCKED 。

或者接收ACTION\_BOOT\_COMPLETED ，这个广播的意思是手机开机并且用户解锁。

也可调用UserManager.isUserUnlocked()方法来查询。

应用迁移已经存在的数据：

Context.migrateSharedPreferencesFrom()

Context.migrateDatabaseFrom()

两种方法在credential encrypted storage 和device encrypted storage存储空间之间去迁移preference 和database的数据.

## 实例分析：启动FallbackHome流程

在分析7.0过程中发现在启动Launcher之前会先启动一个FallbackHome，之后才会启动Launcher，通过调查发现FallbackHome属于Settings中的一个activity，Settings的android:directBootAware为true，并且FallbackHome在category中配置了Home属性，而Launcher的android:directBootAware为false，所有只有FallbackHome可以在direct boot模式下启动。

## enableScreenIfNeededLocked

**void** enableScreenIfNeededLocked() {  
 **if** (**mDisplayEnabled**) {  
 **return**;  
 }  
 **if** (!**mSystemBooted** && !**mShowingBootMessages**) {  
 **return**;  
 }  
 **mH**.sendEmptyMessage(H.***ENABLE\_SCREEN***);  
}

**case *ENABLE\_SCREEN***: {  
 performEnableScreen();  
 **break**;  
}

## performEnableScreen

**private void** performEnableScreen() {  
 **synchronized**(**mWindowMap**) {  
 **if** (**mDisplayEnabled**) {  
 **return**;// //如果设备已经enabled，返回  
 }  
 **if** (!**mSystemBooted** && !**mShowingBootMessages**) {  
 **return**;// 如果不是系统启动，并且没有启动信息，返回  
 }  
  
 **if** (!**mShowingBootMessages** && !**mPolicy**.canDismissBootAnimation()) {  
 **return**;  
 }  
////如果不是强制设备enable，并且开机动画还没有结束，返回  
 *// Don't enable the screen until all existing windows have been drawn.* **if** (!**mForceDisplayEnabled** *//* ***TODO(multidisplay): Expand to all displays?*** && getDefaultDisplayContentLocked().checkWaitingForWindows()) {  
 **return**;  
 }  
  
 **if** (!**mBootAnimationStopped**) {  
 **try** {  
 IBinder surfaceFlinger = ServiceManager.*getService*(**"SurfaceFlinger"**);  
 **if** (surfaceFlinger != **null**) {  
 Slog.*i*(***TAG\_WM***, **"\*\*\*\*\*\*\* TELLING SURFACE FLINGER WE ARE BOOTED!"**);  
 Parcel data = Parcel.*obtain*();  
 data.writeInterfaceToken(**"android.ui.ISurfaceComposer"**);  
 surfaceFlinger.transact(IBinder.***FIRST\_CALL\_TRANSACTION***, *// BOOT\_FINISHED* data, **null**, 0);  
 data.recycle();  
 }  
 }  
 **mBootAnimationStopped** = **true**;  
 }  
  
 **if** (!**mForceDisplayEnabled** && !checkBootAnimationCompleteLocked()) {  
 **if** (***DEBUG\_BOOT***) Slog.*i*(***TAG\_WM***, **"performEnableScreen: Waiting for anim complete"**);  
 **return**;  
 }  
  
 **mDisplayEnabled** = **true**;  
  
 *// Enable input dispatch.* **mInputMonitor**.setEventDispatchingLw(**mEventDispatchingEnabled**);  
 }  
//通知ActivityManagerService开机动画完成  
 **try** {  
 **mActivityManager**.bootAnimationComplete();  
 } **catch** (RemoteException e) {  
 }  
//通知ActivityManagerService Screen可以enable  
 **mPolicy**.enableScreenAfterBoot();  
  
 *// Make sure the last requested orientation has been applied.* updateRotationUnchecked(**false**, **false**);  
}

在开机将近尾声时WindowManagerService会调用enableScreenIfNeededLocked函数来判断是否将Screen enable。通过Handler发送ENABLE\_SCREEN消息到主线程

### checkWaitingForWindows

**boolean** checkWaitingForWindows() {  
  
 **mHaveBootMsg** = **false**; //是否有启动message   
 **mHaveApp** = **false**; //是否有APP  
 **mHaveWallpaper** = **false**; //是否有Wallpaper  
 **mHaveKeyguard** = **true**; //是否有Keyguar  
  
 **final** WindowState visibleWindow = getWindow(w -> {  
 **if** (w.isVisibleLw() && !w.**mObscured** && !w.isDrawnLw()) {  
 **return true**;  
 }  
 **if** (w.isDrawnLw()) {  
 **if** (w.**mAttrs**.**type** == ***TYPE\_BOOT\_PROGRESS***) {  
 **mHaveBootMsg** = **true**;  
 } **else if** (w.**mAttrs**.**type** == ***TYPE\_APPLICATION*** || w.**mAttrs**.**type** == ***TYPE\_DRAWN\_APPLICATION***) {  
 **mHaveApp** = **true**;  
 } **else if** (w.**mAttrs**.**type** == ***TYPE\_WALLPAPER***) {  
 **mHaveWallpaper** = **true**;  
 } **else if** (w.**mAttrs**.**type** == ***TYPE\_STATUS\_BAR***) {  
 **mHaveKeyguard** = **mService**.**mPolicy**.isKeyguardDrawnLw();  
 }  
 }  
 **return false**;  
 });  
  
 **if** (visibleWindow != **null**) {  
 *// We have a visible window.* **return true**;  
 }  
  
 *// if the wallpaper service is disabled on the device, we're never going to have  
 // wallpaper, don't bother waiting for it* **boolean** wallpaperEnabled = **mService**.**mContext**.getResources().getBoolean(  
 com.android.internal.R.bool.***config\_enableWallpaperService***)  
 && !**mService**.**mOnlyCore**;  
  
 *// If we are turning on the screen to show the boot message, don't do it until the boot  
 // message is actually displayed.* **if** (!**mService**.**mSystemBooted** && !**mHaveBootMsg**) {  
 **return true**;  
 }  
  
 *// If we are turning on the screen after the boot is completed normally, don't do so until  
 // we have the application and wallpaper.* **if** (**mService**.**mSystemBooted** && ((!**mHaveApp** && !**mHaveKeyguard**) || (wallpaperEnabled && !**mHaveWallpaper**))) {  
 **return true**;  
 }  
  
 **return false**;  
}

#### 获取所有窗口

WindowState getWindow(Predicate<WindowState> callback) {  
 **for** (**int** i = **mChildren**.size() - 1; i >= 0; --i) {  
 **final** WindowState w = **mChildren**.get(i).getWindow(callback);  
 **if** (w != **null**) {  
 **return** w;  
 }  
 }  
  
 **return null**;  
}

## AMS. *UNLOCK*

调用UserController的finishUserUnlocked函数来发送ACTION\_USER\_UNLOCKED广播

**case *SYSTEM\_USER\_UNLOCK\_MSG***: {  
 **final int** userId = msg.**arg1**;  
 **mSystemServiceManager**.unlockUser(userId);  
 **synchronized** (ActivityManagerService.**this**) {  
 **mRecentTasks**.loadUserRecentsLocked(userId);  
 }  
 **if** (userId == UserHandle.***USER\_SYSTEM***) {  
 startPersistentApps(PackageManager.***MATCH\_DIRECT\_BOOT\_UNAWARE***);  
 }  
 installEncryptionUnawareProviders(userId);  
 **mUserController**.finishUserUnlocked((UserState) msg.**obj**);  
 **break**;

## UserController. finishUserUnlocked发送广播

**void** finishUserUnlocked(**final** UserState uss) {

*// Dispatch unlocked to external apps***final** Intent unlockedIntent = **new** Intent(Intent.***ACTION\_USER\_UNLOCKED***);  
unlockedIntent.putExtra(Intent.***EXTRA\_USER\_HANDLE***, userId);  
unlockedIntent.addFlags(  
 Intent.***FLAG\_RECEIVER\_REGISTERED\_ONLY*** | Intent.***FLAG\_RECEIVER\_FOREGROUND***);  
**mInjector**.broadcastIntentLocked(unlockedIntent, **null**, **null**, 0, **null**,  
 **null**, **null**, AppOpsManager.***OP\_NONE***, **null**, **false**, **false**, ***MY\_PID***, ***SYSTEM\_UID***,  
 userId);

## 小结

Android 7.0新增了DirectBoot功能，AOSP中为实现该功能修改了开机代码流程，并且这部分流程并未根据设备是否支持DirectBoot做区分，只是流程上做了兼容，确保不支持DirectBoot的设备在这套流程下也能正常开机。

在这套流程下，用户解锁后才可进入非directBootAware应用，包括Launcher。

com.android.settings/.FallbackHome中判断用户解锁状态，已解锁才会Finish掉去启动Launcher，未解锁就

等待ACTION\_USER\_UNLOCKED广播后再去启动Launcher。非DirectBoot模式下耗时4s就是在等待  
finishBooting后的系统广播ACTION\_USER\_UNLOCKED。

目前已从APP和PackageManagerService的角度尝试修改，在开机流程中绕过FallbackHome，但验证失败：

### 绕过FallbackHome

1）去除FallbackHome的android.intent.category.Home属性会导致停留在开机动画之后的界面。因为此时仍旧处于未解锁状态，且Launcher非directBootAware应用，PMS中的限制导致此时无法启动Launcher；

2）修改FallbackHome和Launcher的优先级仍旧先启动FallbackHome；

3）将Launcher标记为directBootAware应用会导致开机后Launcher crash。因为Launcher中的widget仍旧是非directBootAware的，此时仍旧无法启动，除非将widget相关的APP都标记为directBootAware；

4）PMS依赖手机当前的状态，需要user解锁才能正常查询。如果强制修改，不考虑DirectBoot和当前启动状态即使当前user未解锁，依然可以查询符合条件的component，修改后会有无法开机的现象。因为Launcher不是directBootAware的，当前手机user尚未解锁，涉及存储相关的解锁也未进行。  
开机绕过FallbackHome涉及的修改面很多，并非通过修改APP或PMS可以实现，还涉及存储区域解锁以及用户状态和ACTION\_USER\_UNLOCKED广播的修改，对AOSP开机流程改动较大，暂时尚未有较好的优化方案

## 其他app怎么做的

### DirectBoot

**public class** BootBroadcastReceiver **extends** BroadcastReceiver {  
  
 **private static final** String TAG = **"BootBroadcastReceiver"**;  
  
 @Override  
 **public void** onReceive(Context context, Intent intent) {  
 **boolean** bootCompleted;  
 String action = intent.getAction();  
 Log.i(TAG, **"Received action: "** + action + **", user unlocked: "** + UserManagerCompat  
 .isUserUnlocked(context));  
 **if** (BuildCompat.isAtLeastN()) {  
 bootCompleted = Intent.ACTION\_LOCKED\_BOOT\_COMPLETED.equals(action);  
 }  
 **if** (!bootCompleted) {  
 **return**;  
 }  
 AlarmUtil util = **new** AlarmUtil(context);  
 AlarmStorage alarmStorage = **new** AlarmStorage(context);  
 **for** (Alarm alarm : alarmStorage.getAlarms()) {  
 util.scheduleAlarm(alarm);  
 }  
 }  
}

**public void** scheduleAlarm(Alarm alarm) {  
 Intent intent = **new** Intent(mContext, AlarmIntentService.**class**);  
 intent.putExtra(AlarmIntentService.ALARM\_KEY, alarm);

}

developers\samples\android\security\DirectBoot\Application\src\main\AndroidManifest.xml

<**receiver android:name=".BootBroadcastReceiver"  
 android:exported="false"  
 android:directBootAware="true"**>  
 *<!-- Listening the BOOT\_COMPLETED action for legacy pre-N devices -->* <**intent-filter**>  
 <**action android:name="android.intent.action.LOCKED\_BOOT\_COMPLETED"** />  
 <**action android:name="android.intent.action.BOOT\_COMPLETED"** />  
 </**intent-filter**>  
</**receiver**>  
  
<**service android:name=".alarms.AlarmIntentService"  
 android:exported="false"  
 android:directBootAware="true"**/>

## 参考

DirectBoot功能介绍

https://www.jianshu.com/p/c9534e487f15

# QA

1. adb install –r A.apk 执行原理？tmp目录作用，INSTALL\_FAILED\_ILLEGITIMATE\_APK各种安装报错原因，cnt的作用，为啥可以。
2. 系统目录结构是啥
3. App和系统签名的原理是什么？
4. 在sws下正常运行as，执行一次系统签名呢

framework如何更新呢？

pms包管理服务分析-PackageManagerService构造函数和包扫描过程

恢复出厂究竟干了啥

rwxrwxrwx root root 2017-09-15 10:10 bugreports -> /data/data/com.android.shell/files/bugreports

限制第三方cnt原理

updated-package 是否是判断已经升级的标志，在settings界面的时候

SystemServer工作原理，Zygote如何启动的？

alreadyDexOpted.add(frameworkDir.getPath() + "/framework-res.apk");总是失败的原因

adb install执行原理

权限方式来控制第三方安装

最好是在拷贝文件之前就处理下

机子都能root了，其实就没有必要处理版本问题了

# 参考

[Android系统应用的开发和测试](http://www.epubit.com.cn/book/onlinechapter/30644)

[Android 源代码目录结构1 - bionic](http://blog.csdn.net/kickxxx/article/details/6927272)

[Android init.rc文件解析过程详解(一)](http://blog.csdn.net/mk1111/article/details/16357327)

[Android的权限机制之—— “沙箱”机制sharedUserId和签名](http://dengzhangtao.iteye.com/blog/1989065)

**android笔记之SystemServiceRegistry**

**https://www.jianshu.com/p/db3ced6c933b**

**服务怎么添加**

**https://blog.csdn.net/csdn1126274345/article/details/79549124**

# Task

Settings.Global

Wifi热点的ip的查询

Root和非root区别是什么？是否存在root也无法修改的build.prop

书籍: 2017年06月深入理解Android内核设计思想 第2版（上下册）

2017年04月 深入浅出Android源代码：基于Android 6.0和实际开发案例剖析

~~2015年11月 Android系统优化从入门到精通~~

2016年10月 深入解析Android虚拟机

2015年07月深入理解Android系统

2015年06月 构建嵌入式Android系统

深入理解android 卷1 2 3

[RK3399][Android7.1] 调试笔记: <http://blog.csdn.net/kris_fei/article/category/7318364>

Android 安全框架 -- 总概http://blog.csdn.net/blue\_rush/article/details/55045546

https://security.tencent.com/index.php/blog/msg/38