# Todo

**CountryDetectorService**

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

# 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。

## 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 |

## 启动流程

### 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");  } |

#### 设置系统的的语言

/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);

### 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

#### 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);

#### startCoreServices

/\* 启动一些关键服务\*

//启动LightsService

//启动BatteryService

//启动UsageStatsService

//启动WebViewUpdateService

#### startOtherServices

//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.

#### 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);

}

### AMS.systemReady

|  |
| --- |
| public final class ActivityManagerService extends ActivityManagerNative  implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {    public void systemReady(final Runnable goingCallback) {  ... //update相关  mSystemReady = true;    //杀掉所有非persistent进程  removeProcessLocked(proc, true, false, "system update done");  mProcessesReady = true;  goingCallback.run(); //[见小节1.6.2]    addAppLocked(info, false, null); //启动所有的persistent进程  mBooting = true;    //启动home  **startHomeActivityLocked**(mCurrentUserId, "systemReady");  //恢复栈顶的Activity  mStackSupervisor.resumeTopActivitiesLocked();  }  } |

System\_server主线程的启动工作,总算完成, 进入Looper.loop()状态,等待其他线程通过handler发送消息再处理.

## 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.

### AMS.systemReady

|  |
| --- |
| public final class ActivityManagerService extends ActivityManagerNative  implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {    public void systemReady(final Runnable goingCallback) {  ... //update相关  mSystemReady = true;    //杀掉所有非persistent进程  removeProcessLocked(proc, true, false, "system update done");  mProcessesReady = true;  goingCallback.run(); //[见小节1.6.2]    addAppLocked(info, false, null); //启动所有的persistent进程  mBooting = true;    //启动home  startHomeActivityLocked(mCurrentUserId, "systemReady");  //恢复栈顶的Activity  mStackSupervisor.resumeTopActivitiesLocked();  }  } |

System\_server主线程的启动工作,总算完成, 进入Looper.loop()状态,等待其他线程通过handler发送消息再处理.

## 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);

}

}

# JNI层

## JNI的架构和实现方式

# 栈管理

我们知道,a ctivity 在 AMS 中的形式是 ActivityRecord,task 在 AMS 中的形式为TaskRecord,进程在 AMS 中的管理形式为 ProcessRecord。

**Android系统界面用户体验组织（TASKS），**，用户体验到的直观界面都是由很多Tasks组成的。比如，我们从最近任务栏，就可以看到很多TASKS。用户可以随意的切换到其中的一个TASK。也可以按HOME键退出一个TASK。TASK的组织形式是ActivityStack，一个ActivityStack是由好多Activity组成的堆栈。从Android系统设计者的角度来看，一个Task定义了一组行为，而这组行为是由跨多个应用程序的多个Activity组织而构成的。这个打破了以往由应用程序（进程）来定义的资源边界。一个Task就是一个场景的实现。从此构建系统用户体验和行为的边界在于TASK，而非应用程序或者进程。

## 概念

### Task

Task 是activities的集合，通过activity stack来管理，依靠先进后出队列来实现;每个task中都至少有一个activity，新实例出来的activity置于栈顶，Task可以被切换到后台

### Activity Stack

如上所诉，Activity承担了大量的显示和交互工作，从某种角度上将，我们看见的应用程序就是许多个Activity的组合。为了让这许多 Activity协同工作而不至于产生混乱，Android平台设计了一种堆栈机制用于管理Activity，其遵循先进后出的原则，系统总是显示位于栈顶的Activity，从逻辑上将，位于栈顶的Activity也就是最后打开的Activity，这也是符合逻辑的。

在操作应用程序时，每次启动新的Activity，都会将此压入Activity Stack，当用户执行返回操作时，移除Activity Stack顶上的Activity，这样就实现了返回上一个Activty的功能。直到用户一直返回到Home Screen，这时候可以理解为移除了Activity Stack所有的Activity，这个Activity Stack不再存在，应用程序也结束了运行。

可以通过 adb shell dumpsys 查看 TASKS的ActivityStacks

### task的taskAffinity

 任务归属，taskAffinity 这个属性主要是决定持有每个activity属于哪个task。 默认情况下，同一个包中的activity共享同一个affinity（任务共用性）。

### **task的launchMode**

 standard(default)：standard，标准的Activity是可以随意插入到TASK中去的一个组织结构，可以去TaskA，也可以去TaskB，也可以去TaskC，直接并无任何的联系

 singleTop，如果在任务的栈顶正好存在该Activity的实例，就重用该实例，否则就创建新的实例并放入栈顶。

 singleTask，如果在栈中已经有该Activity的实例，就重用该实例(会调用实例的onNewIntent())。重用时，会让该实例回到栈顶，因此在它上面的实例将会被移除栈。如果栈中不存在该实例，将会创建新的实例放入栈中。

 singleInstance

## 源码分析

### 主要涉及4个类

**1) ActivityRecord：activity记录的基本数据结构，**存在历史栈的一个实例，代表一个Activity。

**2) TaskRecord：**Activity栈，内部维护一个ArrayList<ActivityRecord>

**3) ActivityStack：**并不是一个Activity栈，真正意义上的Activity栈是TaskRecord，这个类是负责管理各个Activity栈，内部维护一个ArrayList<TaskRecord>

**4) ActivityStackSupervisor：**内部持有一个ActivityStack，而ActivityStack内部也持有ActivityStackSupervisor，相当于ActivityStack的辅助管理类

#### **ActivityRecord**

/frameworks/base/services/core/java/com/android/server/am/ActivityRecord.java

**void** **[dump](http://androidxref.com/5.1.0_r1/s?refs=dump&project=frameworks)**([PrintWriter](http://androidxref.com/5.1.0_r1/s?defs=PrintWriter&project=frameworks) **[pw](http://androidxref.com/5.1.0_r1/s?refs=pw&project=frameworks)**, [String](http://androidxref.com/5.1.0_r1/s?defs=String&project=frameworks) **[prefix](http://androidxref.com/5.1.0_r1/s?refs=prefix&project=frameworks)**) 定义的方法用于dumpsys activity对于2.2.1.1. ActivityRecord的打印，

final class ActivityRecord {

TaskRecord **task**; // the task this is in.

final IApplicationToken.Stub appToken;

final int userId;

int theme;

int launchMode;

...

}

成员变量task表示自己所在的TaskRecord，这样要找到自己所在的TaskRecord就不必遍历查找了。

#### TaskRecord

final class TaskRecord {

/\*\* List of all activities in the task arranged in history order \*/

final ArrayList<ActivityRecord> mActivities;

/\*\* Current stack \*/

ActivityStack stack;

}

同样的道理，成员变量stack表示自己所在的ActivityStack

#### ActivityStack

final class ActivityStack {

private ArrayList<TaskRecord> **mTaskHistory** = new ArrayList<>();

/\*\* Run all ActivityStacks through this \*/

final ActivityStackSupervisor mStackSupervisor;

ActivityStack(ActivityStackSupervisor.ActivityContainer activityContainer, RecentTasks recentTasks) {

mStackSupervisor = activityContainer.getOuter();

...

}

}

#### ActivityStackSupervisor

public final class ActivityStackSupervisor {

private ActivityStack mFocusedStack;

}

### 场景分析

#### 二、场景解析

##### 1、从桌面第一次启动App

startActivityLocked里构造一个ActivityRecord  
新建一个TaskRecord，并存入mTaskHistory  
ActivityRecord存入mActivities



final int startActivityUncheckedLocked(...) {

final int startActivityUncheckedLocked(...) {

if (reuseTask == null) {

r.setTask(targetStack.createTaskRecord(...);

...

targetStack.startActivityLocked(r, newTask, doResume, keepCurTransition, options);

...

}

}

1) TaskRecord存入mTaskHistory

TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

boolean toTop) {

TaskRecord task = new TaskRecord(mService, taskId, info, intent, voiceSession,

voiceInteractor);

addTask(task, toTop, false);

return task;

}

void addTask(final TaskRecord task, final boolean toTop, boolean moving) {

task.stack = this;

if (toTop) {

insertTaskAtTop(task, null);

} else {

mTaskHistory.add(0, task);

updateTaskMovement(task, false);

}

...

}

private void insertTaskAtTop(TaskRecord task, ActivityRecord newActivity) {

...

mTaskHistory.add(taskNdx, task);

updateTaskMovement(task, true);

}

2) ActivityRecord存入mActivities



final void startActivityLocked(ActivityRecord r, boolean newTask, ...) {

...

task = mTaskHistory.get(taskNdx);

...

task.addActivityToTop(r);

}

void addActivityToTop(ActivityRecord r) {

addActivityAtIndex(mActivities.size(), r);

}

void addActivityAtIndex(int index, ActivityRecord r) {

...

mActivities.add(index, r);

...

}

##### 2、App启动一个Activity

会不会新建一个TaskRecord取决于launchMode，默认的standard模式不会创建新的TaskRecord  
构造一个ActivityRecord存入mActivities，与上面第二步一样

##### 3、回退



/\*\* @return true if this was the last activity in the task \*/

boolean removeActivity(ActivityRecord r) {

mActivities.remove(r);

...

if (mActivities.isEmpty()) {

return !mReuseTask;

}

...

return false;

}

#### 4、再次回退，回到桌面

从mActivities移除当前ActivityRecord与上面一样，只是当mActivities为空时，会触发mTaskHistory移除当前TaskRecord，如果mTaskHistory为空，则切换到桌面，给mStackSupervisor.mFocusedStack重新赋值

private void removeActivityFromHistoryLocked(ActivityRecord r, String reason) {

...

final TaskRecord task = r.task;

if (task != null && task.removeActivity(r)) {

if (mStackSupervisor.isFrontStack(this) && task == topTask() &&

task.isOverHomeStack()) {

mStackSupervisor.moveHomeStackTaskToTop(task.getTaskToReturnTo(), reason);

}

removeTask(task, reason);

}

}

void removeTask(TaskRecord task, String reason, boolean notMoving) {

...

mTaskHistory.remove(task);

...

if (mTaskHistory.isEmpty()) {

final boolean notHomeStack = !isHomeStack();

if (isOnHomeDisplay()) {

String myReason = reason + " leftTaskHistoryEmpty";

if (mFullscreen || !adjustFocusToNextVisibleStackLocked(null, myReason)) {

mStackSupervisor.moveHomeStack(notHomeStack, myReason);

}

}

...

}

...

task.stack = null;

}

如果不是从Activity调用startActivity，那么目标Activity就不知道自己该属于哪个TaskRecord，所以得指定FLAG\_ACTIVITY\_NEW\_TASK，就会新建一个TaskRecord

作者：风风风筝  
链接：http://www.jianshu.com/p/dbea4645cbc7  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## REF

[Activity（三）栈管理](http://www.jianshu.com/p/dbea4645cbc7)

# startService的原理分析

http://blog.csdn.net/luoshengyang/article/details/6677029

# WMS

frameworks/base/services/java/com/android/server/WindowManagerService.java

WMS只负责管理View的z-order，也就是管理当前那个View在最上层显示，并不管理绘制。

关键字：堆栈，

与Activity类似，Android系统中的窗口也是以**堆栈**的形式组织在WMS中的，其中，Z轴位置较低的窗口位于Z轴位置较高的窗口的下面（前台Z最大）。在本文中，我们就详细分析WMS是如何以堆栈的形式来组织窗口的。

应用程序进程中的每一个Activity组件在AMS中都对应有一个ActivityRecord对象。AMS的每一个ActivityRecord对象AMS中都对应有一个AppWindowToken对象。输入法窗口类似，在壁纸管理服务WallpaperManagerService中，每一个壁纸窗口都对应有一个Binder对象，这个Binder对象在WMS也对应有一个WindowToken对象。

在WMS中，无论是AppWindowToken对象，还是WindowToken对象，它们都是用来描述一组有着相同令牌的窗口的，每一个窗口都是通过一个WindowState对象来描述的。例如，一个Activity组件窗口可能有一个启动窗口（Starting Window），还有若干个子窗口，那么这些窗口就会组成一组，并且都是以Activity组件在AMS中所对应的AppWindowToken对象为令牌的。从抽象的角度来看，就是在AMS中，每一个令牌（AppWindowToken或者WindowToken）都是用来描述一组窗口（WindowState）的，并且每一个窗口的子窗口也是与它同属于一个组，即都有着相同的令牌。

上述的窗口组织方式如图1所示：



图1 窗口在WindowManagerService服务中的组织方式

其中，Activity Stack是在ActivityManagerService服务中创建的，Token List和Window Stack是在WindowManagerService中创建的，而Binder for IM和Binder for WP分别是在InputMethodManagerService服务和WallpaperManagerService服务中创建的，用来描述一个输入法窗口和一个壁纸窗口。

图1中的对象的对应关系如下所示：

1. ActivityRecord-J对应于AppWindowToken-J，后者描述的一组窗口是{WindowState-A, WindowState-B, WindowState-B-1}，其中， WindowState-B-1是WindowState-B的子窗口。

2. ActivityRecord-K对应于AppWindowToken-K，后者描述的一组窗口是{WindowState-C, WindowState-C-1, WindowState-D, WindowState-D-1}，其中， WindowState-C-1是WindowState-C的子窗口，WindowState-D-1是WindowState-D的子窗口。

3. ActivityRecord-N对应于AppWindowToken-N，后者描述的一组窗口是{WindowState-E}，其中， WindowState-E是系统当前激活的Activity窗口。

4. Binder for IM对应于WindowToken-I，后者描述的一组窗口是{WindowState-I}，其中， WindowState-I是WindowState-E的输入法窗口。

5. Binder for WP对应于WindowToken-W，后者描述的一组窗口是{WindowState-W}，其中， WindowState-W是WindowState-E的壁纸窗口。

从图1还可以知道，Window Stack中的WindowState是按照它们所描述的窗口的Z轴位置从低到高排列的。

以上就是WindowManagerService服务组织系统中的窗口的抽象模型，接下来我们将分析AppWindowToken、WindowToken和WindowState的一些增加、移动和删除等操作，以便可以对这个抽象模型有一个更深刻的认识。

## 增加AppWindowToken

从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，一个Activity组件在启动的过程中，ActivityManagerService服务会调用调用WindowManagerService类的成员函数addAppToken来为它增加一个AppWindowToken，如下所示：

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. /\*\*
6. \* Mapping from a token IBinder to a WindowToken object.
7. \*/
8. **final** HashMap<IBinder, WindowToken> mTokenMap =
9. **new** HashMap<IBinder, WindowToken>();
11. /\*\*
12. \* The same tokens as mTokenMap, stored in a list for efficient iteration
13. \* over them.
14. \*/
15. **final** ArrayList<WindowToken> mTokenList = **new** ArrayList<WindowToken>();
16. ......
18. /\*\*
19. \* Z-ordered (bottom-most first) list of all application tokens, for
20. \* controlling the ordering of windows in different applications.  This
21. \* contains WindowToken objects.
22. \*/
23. **final** ArrayList<AppWindowToken> mAppTokens = **new** ArrayList<AppWindowToken>();
24. ......
26. **public** **void** addAppToken(**int** addPos, IApplicationToken token,
27. **int** groupId, **int** requestedOrientation, **boolean** fullscreen) {
28. ......
30. **synchronized**(mWindowMap) {
31. AppWindowToken wtoken = findAppWindowToken(token.asBinder());
32. **if** (wtoken != ) {
33. ......
34. **return**;
35. }
36. wtoken = **new** AppWindowToken(token);
37. ......
38. mAppTokens.add(addPos, wtoken);
39. ......
40. mTokenMap.put(token.asBinder(), wtoken);
41. mTokenList.add(wtoken);
43. ......
44. }
45. }
47. ......
48. }

WMS类有三个成员变量mTokenMap、mTokenList和mAppTokens，它们都是用来描述系统中的窗口的。

成员变量mTokenMap指向的是一个HashMap，它里面保存的是一系列的WindowToken对象，每一个WindowToken对象都是用来描述一个窗口的，并且是以描述这些窗口的一个Binder对象的IBinder接口为键值的。例如，对于Activity组件类型的窗口来说，它们分别是以用来描述它们的一个ActivityRecord对象的IBinder接口保存在成员变量mTokenMap所指向的一个HashMap中的。

成员变量mTokenList指向的是一个ArrayList，它里面保存的也是一系列WindowToken对象，这些WindowToken对象与保存在成员变量mTokenMap所指向的一个HashMap中的WindowToken对象是一样的。成员变量mTokenMap和成员变量mTokenList的区别就在于，前者在给定一个IBinder接口的情况下，可以迅速指出是否存在一个对应的WindowToken对象，而后者可以迅速**遍**历WindowManagerService服务中的WindowToken对象。

成员变量mAppTokens指向的也是一个ArrayList，不过它里面保存的是一系列**AppWindowToken**对象，每一个AppWindowToken对象都是用来描述一个**Activity**组件窗口的，而这些AppWindowToken对象是以它们描述的窗口的Z轴坐标由小到大保存在这个ArrayList中的，这样我们就可以通过这个ArrayList来从上到下或者从下到上地遍历系统中的**所有Activity**组件窗口。由于这些AppWindowToken对象所描述的Activity组件窗口也是一个窗口，并且AppWindowToken类是从WindowToken继承下来的，因此，**这些AppWindowToken对象还会同时被保存在成员变量mTokenMap所指向的一个HashMap和成员变量mTokenList所指向的一个ArrayList中。**

理解了WMS类的这三个成员变量的含义之后，它的成员函数addAppToken的实现就好理解了，其中，参数token指向的便是用来描述正在启动的Activity组件所对应的一个ActivityRecord对象，而参数addPos用来描述该Activity组件在堆栈中的位置，这个位置同时也是接下来要创建的AppWindowToken对象在WMS类的mTokenList所描述的一个ArrayList中的位置。

WMS类的成员函数addAppToken首先调用另外一个成员函数findAppWindowToken来在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个AppWindowToken。如果已经存在的话，那么WindowManagerService类的成员函数addAppToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个AppWindowToken对象，并且会将该AppWindowToken对象分别保存在WMS的成员变量mTokenMap、mTokenList和mAppTokens中。

## 删除AppWindowToken

删除AppWindowToken是通过调用WindowManagerService类的成员函数removeAppTokensLocked来实现的，如下所示：

1. **private** **void** removeAppTokensLocked(List<IBinder> tokens) {
2. // XXX This should be done more efficiently!
3. // (take advantage of the fact that both lists should be
4. // ordered in the same way.)
5. **int** N = tokens.size();
6. **for** (**int** i=0; i<N; i++) {
7. IBinder token = tokens.get(i);
8. **final** AppWindowToken wtoken = findAppWindowToken(token);
9. **if** (!mAppTokens.remove(wtoken)) {
10. ......
11. i--;
12. N--;
13. }
14. }
15. }

WindowManagerService类的成员函数removeAppTokensLocked可以同时删除一组AppWindowToken对象。

参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要删除的。WindowManagerService类的成员函数removeAppTokensLocked通过一个for循环来依次调用另外一个成员函数findAppWindowToken，以便可以找到保存在列表tokens中的每一个IBinder接口所对应的AppWindowToken对象，然后将该AppWindowToken对象从WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList中删除。

注意，WindowManagerService类的成员函数removeAppTokensLocked是在内部使用的，它只是把一个AppWindowToken对象从成员变量mAppTokens中删除，而没有从另外两个成员变量mTokenMap和mTokenList中删除。

## 移动AppWindowToken至指定位置

移动AppWindowToken至指定位置是通过调用WMS类的成员函数moveAppToken来实现的，如下所示：

1. **public** **void** moveAppToken(**int** index, IBinder token) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **synchronized**(mWindowMap) {
8. ......
9. **final** AppWindowToken wtoken = findAppWindowToken(token);
10. **if** (wtoken ==  || !mAppTokens.remove(wtoken)) {
11. ......
12. **return**;
13. }
14. mAppTokens.add(index, wtoken);
15. ......
17. **final** **long** origId = Binder.clearCallingIdentity();
18. ......
19. **if** (tmpRemoveAppWindowsLocked(wtoken)) {
20. ......
21. reAddAppWindowsLocked(findWindowOffsetLocked(index), wtoken);
22. ......
23. updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES);
24. mLayoutNeeded = **true**;
25. performLayoutAndPlaceSurfacesLocked();
26. }
27. Binder.restoreCallingIdentity(origId);
28. }
29. }

参数token描述的是要移动的AppWindowToken对象所对应的一个IBinder接口，而参数index描述的是该AppWindowToken对象要移动到的位置。注意，移动一个AppWindowToken对象到指定的位置是需要android.Manifest.permission.MANAGE\_APP\_TOKENS权限的。

WMS类的成员函数moveAppToken首先找到与参数token所对应的AppWindowToken对象，并且将该AppWindowToken对象从WMS类的成员变量mAppTokens所描述的一个ArrayList中移除，这样做的目的是为了接下来可以将该AppWindowToken对象移动至该ArrayList中的指定位置上，即参数index所描述的位置上。

注意，上述操作只是将参数token所对应的AppWindowToken对象移动到了WMS类的成员变量mAppTokens所描述的一个ArrayList的指定位置上，接下来还需要同时将与该A**ppWindowToken对象所对应的WindowState**对象移动至WMS服务内部的一个WindowState堆栈合适位置上去。

**移动对应的WindowState**对象的操作同样也是分两步执行的：第一步先调用WMS类的成员函数tmpRemoveAppWindowsLocked来将这些WindowState对象从**原来的WindowState堆栈位置移除**；第二步再调用WMS类的成员函数reAddAppWindowsLocked来将这些WindowState对象**插入到WindowState堆栈的合适位置去**。

对应的WindowState对象被移动到的合适位置是通过调用WMS类的成员函数findWindowOffsetLocked来获得的，它的实现如下所示：

1. 参考/\*\*
2. \* Z-ordered (bottom-most first) list of all Window objects.
3. \*/
4. **final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();
5. ......
7. **private** **int** findWindowOffsetLocked(**int** tokenPos) {
8. **final** **int** NW = mWindows.size();
10. **if** (tokenPos >= mAppTokens.size()) {
11. **int** i = NW;
12. **while** (i > 0) {
13. i--;
14. WindowState win = mWindows.get(i);
15. **if** (win.getAppToken() != ) {
16. **return** i+1;
17. }
18. }
19. }
21. **while** (tokenPos > 0) {
22. // Find the first app token below the new position that has
23. // a window displayed.
24. **final** AppWindowToken wtoken = mAppTokens.get(tokenPos-1);
25. ......
26. **if** (wtoken.sendingToBottom) {
27. ......
28. tokenPos--;
29. **continue**;
30. }
31. **int** i = wtoken.windows.size();
32. **while** (i > 0) {
33. i--;
34. WindowState win = wtoken.windows.get(i);
35. **int** j = win.mChildWindows.size();
36. **while** (j > 0) {
37. j--;
38. WindowState cwin = win.mChildWindows.get(j);
39. **if** (cwin.mSubLayer >= 0) {
40. **for** (**int** pos=NW-1; pos>=0; pos--) {
41. **if** (mWindows.get(pos) == cwin) {
42. ......
43. **return** pos+1;
44. }
45. }
46. }
47. }
48. **for** (**int** pos=NW-1; pos>=0; pos--) {
49. **if** (mWindows.get(pos) == win) {
50. ......
51. **return** pos+1;
52. }
53. }
54. }
55. tokenPos--;
56. }
58. **return** 0;
59. }

参数tokenPos描述的是一个AppWindowToken对象在WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的位置，WindowManagerService类的成员函数findWindowOffsetLocked的目标就要找到与该AppWindowToken对象所对应的WindowState对象在WindowManagerService服务内部的一个WindowState堆栈的起始偏移位置。有了这个起始偏移位置之后，我们就可以将对应的所有WindowState对象有序地插入到该WindowState堆栈中去。WindowManagerService服务内部的WindowState堆栈是通过WindowManagerService类的成员变量mWindows来描述的。接下来我们就分两种情况来分析这个起始偏移位置的计算过程。

第一种情况是参数tokenPos的值大于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。这是一种**异常**情况，一般来说，参数tokenPos是指向mAppTokens列表的某一个位置的，不过这时候意味着它所描述的AppWindowToken对象的Z轴位置要大于mAppTokens列表的最上面的一个AppWindowToken对象的Z轴位置的。这也就是说，与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象的要位于与mAppTokens列表的最上面的一个AppWindowToken对象所对应的任一个WindoState对象的上面。因此，就需要找到与mAppTokens列表的最上面的一个AppWindowToken对象所对应的**Z轴位置最大的一个WindoState**对象在WindowState堆栈中的位置i，然后就可以知道与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置为i+1。

如何找到mAppTokens列表的最上面的一个AppWindowToken对象所对应的Z轴位置最大的一个WindoState对象在WindowState堆栈中的位置i呢？从图1可以可得到一个结论：WMS内部中的所有WindowState对象都是**按照Z轴从位置从小到大排列在WindowState堆栈中的**，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。**因此，我们只要从WindowState堆栈的顶端开始往下遍历，找到这样的一个WindowState对象，它是属于一个AppWindowToken对象的，即它的成员函数getAppToken的返回值不等于null**，那么它在WindowState堆栈中的位置就是我们要找到的位置i。有了这个位置i之后，将它的值加上1，就可以得到参数t所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

第二种情况是参数tokenPos的值小于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。根据前面得到的推论，我们只要在mAppTokens列表中找到一个AppWindowToken对象，它满足以下三个条件：

A. 它在mAppTokens列表中的位置小于tokenPos；

B. 它在WindowState堆栈中对应有WindowState对象；

C. 它不是将要置于WindowState堆栈的底部。

如果一个AppWindowToken对象在WindowState堆栈中对应有WindowState对象，那么这些WindowState对象也会同时按照Z轴从小到大的顺序保存它的成员变量windows所描述的一个ArrayList中，这意味着如果一个AppWindowToken对象满足条件B，那么它的成员变量windows所描述的一个ArrayList的大小就大于0。

如果一个AppWindowToken对象不是将要置于WindowState堆栈的底部，那么它的成员变量sendingToBottom的值就不等于true，这也意味这个AppWindowToken对象满足条件C。

如果能找到满足上述条件的一个AppWindowToken对象wtoken，那么我们只要找到与它所对应的Z轴位置最大的WindowState对象在WindowManagerService服务内部的WindowState堆栈中的位置i，那么将它的值加1，就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

那么如何找到与这个AppWindowToken对象wtoken对应的Z轴位置最大的WindowState对象在WMS服务内部的WindowState堆栈中的位置i呢？从前面的图1可以知道，一个AppWindowToken对象所对应的WindowState对象可以划分为两种类型：第一种类型是父窗口类型的；第二种是子窗口类型的。如果一个WindowState对象所描述的窗口是父窗口，那么它的子窗口就保存在它的成员变量mChildWindows所描述的一个ArrayList中，并且这些子窗口是按照Z轴位置从小到大的顺序排列的，同时，该WindowState对象也会保存在与它所对应的一个AppWindowToken对象的成员变量windows所描述的一个ArrayList中。

有了上述结论，并且假设存在一个能够满足上述三个条件的AppWindowToken对象wtoken，那么就可以从上到下遍历保存在它的成员变量windows所描述的一个ArrayList中的每一个WindowState对象win：

I. 如果WindowState对象win所描述的一个窗口具有子窗口，那么就继续从上到下遍历这些子窗口，即从上到下遍历WindowState对象win的成员变量mChildWindows所描述的一个ArrayList。如果能找到一个WindowState对象cwin，它的成员变量mSubLayer的值大于等于0，那么该WindowState对象cwin在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。注意，如果WindowState对象cwin的成员变量mSubLayer的值小于0，那么它虽然是一个子窗口，但是它却是位于父窗口的后面的，即它的Z轴位置是小于父窗口的Z轴位置的。

II. 如果WindowState对象win所描述的一个窗口不具有子窗口，即它的成员变量mChildWindows所描述的一个ArrayList的大小等于0，那么它在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。

得到了位置i之后，将它的值加1，那么就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

回到WindowManagerService类的成员函数moveAppToken中，调整好参数token所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈中的位置之后，即调用了成员函数reAddAppWindowsLocked之后，这时候系统中的窗口的布局就会发生了变化，即系统中的窗口的Z轴位置关系发生了变化，那么接下来就需要调用成员函数updateFocusedWindowLocked来重新计算系统中的窗口的Z轴位置，并且调用成员函数performLayoutAndPlaceSurfacesLocked来重新布局系统中的窗口。

## 移动AppWindowToken至顶端

移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToTop来实现的，如下所示：

1. **public** **void** moveAppTokensToTop(List<IBinder> tokens) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppTokensToTop()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. removeAppTokensLocked(tokens);
10. **final** **int** N = tokens.size();
11. **for** (**int** i=0; i<N; i++) {
12. AppWindowToken wt = findAppWindowToken(tokens.get(i));
13. **if** (wt != ) {
14. mAppTokens.add(wt);
15. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
16. mToTopApps.remove(wt);
17. mToBottomApps.remove(wt);
18. mToTopApps.add(wt);
19. wt.sendingToBottom = **false**;
20. wt.sendingToTop = **true**;
21. }
22. }
23. }
25. **if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {
26. moveAppWindowsLocked(tokens, mAppTokens.size());
27. }
28. }
29. Binder.restoreCallingIdentity(origId);
30. }

WindowManagerService类的成员函数moveAppTokensToTop可以同时将一组AppWindowToken移至顶端，同时需要调用者具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要移至顶端的。在将保存在参数tokens中的IBinder接口所对应的AppWindowToken对象移至顶端之前，WindowManagerService类的成员函数首先会调用前面所描述的成员函数removeAppTokensLocked来删除这些AppWindowToken对象，然后再依次将它们添加到WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的末尾去。

注意，WindowManagerService类的成员变量mNextAppTransition用来描述系统当前是否正在切换Activity窗口。如果是的话，那么它的值就不等于WindowManagerPolicy.TRANSIT\_UNSET，这时候就需要：

A. 将所有要移至顶端的AppWindowToken对象都保存在WindowManagerService类的另外一个成员变量mToTopApps所描述的一个ArrayList中去，并且将这些AppWindowToken对象的成员变量sendingToTop的值设置为true。

B. 将所有要移至顶端的AppWindowToken对象所对应WindowState对象都移至WindowManagerService服务内部的一个WindowState堆栈的顶端去，这是通过调用另外一个成员函数moveAppWindowsLocked来实现的。

执行完成上述两个操作之后，与要移至顶端的AppWindowToken对象所对应的窗口就会位于窗口堆栈的最上面了。

## 移动AppWindowToken至底端

移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToBottom来实现的，如下所示：

1. **public** **void** moveAppTokensToBottom(List<IBinder> tokens) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppTokensToBottom()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. removeAppTokensLocked(tokens);
10. **final** **int** N = tokens.size();
11. **int** pos = 0;
12. **for** (**int** i=0; i<N; i++) {
13. AppWindowToken wt = findAppWindowToken(tokens.get(i));
14. **if** (wt != ) {
15. mAppTokens.add(pos, wt);
16. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
17. mToTopApps.remove(wt);
18. mToBottomApps.remove(wt);
19. mToBottomApps.add(i, wt);
20. wt.sendingToTop = **false**;
21. wt.sendingToBottom = **true**;
22. }
23. pos++;
24. }
25. }
27. **if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {
28. moveAppWindowsLocked(tokens, 0);
29. }
30. }
31. Binder.restoreCallingIdentity(origId);
32. }

WindowManagerService类的成员函数moveAppTokensToBottom可以同时将一组AppWindowToken移至底端。将一组AppWindowToken移至底端与将一组AppWindowToken移至顶端的实现是类似的，只不过是移动的方向相反而已。因此，WindowManagerService类的成员函数moveAppTokensToBottom的实现可以参考前面所分析的成员函数moveAppTokensToTop的实现，这里不再详述。

## 增加WindowToken

从图1可以知道，如果一个WindowState对象不是与一个AppWindowToken对象对应的，那么它就必须要与一个WindowToken对象对应。例如，用来描述输入法窗口和壁纸窗口的WindowState对象对应的就是WindowToken对象，而不是AppWindowToken对象，因为它们不是Activity类型的窗口。

输入法窗口和壁纸窗口分别是由输入法管理服务InputMethodManagerService和壁纸管理服务WallpaperManagerService调用WindowManagerService类的成员函数addWindowToken来增加对应的WindowToken对象的，如下所示：

1. **public** **void** addWindowToken(IBinder token, **int** type) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "addWindowToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **synchronized**(mWindowMap) {
8. WindowToken wtoken = mTokenMap.get(token);
9. **if** (wtoken != ) {
10. Slog.w(TAG, "Attempted to add existing input method token: " + token);
11. **return**;
12. }
13. wtoken = **new** WindowToken(token, type, **true**);
14. mTokenMap.put(token, wtoken);
15. mTokenList.add(wtoken);
16. **if** (type == TYPE\_WALLPAPER) {
17. mWallpaperTokens.add(wtoken);
18. }
19. }
20. }

调用WindowManagerService类的成员函数addWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

对于输入法窗口和壁纸窗口来说，参数token指向的是与它们所关联的一个Binder对象的IBinder接口，而参数type描述的是要在WindowManagerService服务内部增加WindowToken对象的窗口的类型。

WindowManagerService类的成员函数addWindowToken首先检查在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个WindowToken对象与参数token对应。如果已经存在的话，那么WindowManagerService类的成员函数addWindowToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个WindowToken对象，并且会将该WindowToken对象分别保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

这里有两个地方需要注意：

A. 由于这里增加的是WindowToken对象，而不是AppWindowToken对象，因此，与增加AppWindowToken不同，这里不需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mAppTokens中。

B. 如果参数type的值等于TYPE\_WALLPAPER，那么就意味着新创建的WindowToken对象是用来描述壁纸窗口的，这时候还需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中，以方便管理壁纸窗口。

对于非输入法窗口、非壁纸窗口以及非Activity窗口来说，它们所对应的WindowToken对象是在它们增加到WindowManagerService服务的时候创建的。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，接下来我们就主要分析与创建WindowToken相关的逻辑，如下所示：

1. **public** **int** addWindow(Session session, IWindow client,
2. WindowManager.LayoutParams attrs, **int** viewVisibility,
3. Rect outContentInsets, InputChannel outInputChannel) {
4. ......
6. **synchronized**(mWindowMap) {
7. ......
9. **boolean** addToken = **false**;
10. WindowToken token = mTokenMap.get(attrs.token);
11. **if** (token == ) {
12. **if** (attrs.type >= FIRST\_APPLICATION\_WINDOW
13. && attrs.type <= LAST\_APPLICATION\_WINDOW) {
14. ......
15. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
16. }
17. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
18. ......
19. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
20. }
21. **if** (attrs.type == TYPE\_WALLPAPER) {
22. ......
23. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
24. }
25. token = **new** WindowToken(attrs.token, -1, **false**);
26. addToken = **true**;
27. }
29. ......
31. **if** (addToken) {
32. mTokenMap.put(attrs.token, token);
33. mTokenList.add(token);
34. }
36. ......
37. }
39. ......
40. }

如果参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口在WindowManagerService类的成员变量mTokenMap所描述的一个HashMap中没有一个对应的WindowToken对象，并且该WindowManager.LayoutParams对象的成员变量type的值不等于TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER，以及不在FIRST\_APPLICATION\_WINDOW和LAST\_APPLICATION\_WINDOW，那么就意味着这时候要增加的窗口就既不是输入法窗口，也不是壁纸窗口和Activity窗口，因此，就需要以参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口为参数来创建一个WindowToken对象，并且将该WindowToken对象保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

## 删除WindowToken

删除WindowToken是通过调用WindowManagerService类的成员函数removeWindowToken来实现的，如下所示：

1. **public** **void** removeWindowToken(IBinder token) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "removeWindowToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. WindowToken wtoken = mTokenMap.remove(token);
10. mTokenList.remove(wtoken);
11. **if** (wtoken != ) {
12. **boolean** delayed = **false**;
13. **if** (!wtoken.hidden) {
14. wtoken.hidden = **true**;
16. **final** **int** N = wtoken.windows.size();
17. **boolean** changed = **false**;
19. **for** (**int** i=0; i<N; i++) {
20. WindowState win = wtoken.windows.get(i);
22. **if** (win.isAnimating()) {
23. delayed = **true**;
24. }
26. **if** (win.isVisibleNow()) {
27. applyAnimationLocked(win,
28. WindowManagerPolicy.TRANSIT\_EXIT, **false**);
29. changed = **true**;
30. }
31. }
33. **if** (changed) {
34. mLayoutNeeded = **true**;
35. performLayoutAndPlaceSurfacesLocked();
36. updateFocusedWindowLocked(UPDATE\_FOCUS\_NORMAL);
37. }
39. **if** (delayed) {
40. mExitingTokens.add(wtoken);
41. } **else** **if** (wtoken.windowType == TYPE\_WALLPAPER) {
42. mWallpaperTokens.remove(wtoken);
43. }
44. }
46. ......
47. } **else** {
48. Slog.w(TAG, "Attempted to remove non-existing token: " + token);
49. }
50. }
51. Binder.restoreCallingIdentity(origId);
52. }

调用WindowManagerService类的成员函数removeWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

WindowManagerService类的成员函数removeWindowToken首先找到与参数token所描述的Binder接口所对应的WindowToken对象，接着再将该WindowToken对象从WindowManagerService类的成员变量mTokenMap和mTokenList中删除。

删除了一个WindowToken对象之后，如果该WindowToken对象不是处于不可见的状态，即它的成员变量hidden的值不等于false，那么就意味着它所描述窗口口也有可能是可见的，那么WindowManagerService类的成员函数removeWindowToken就需要作以下两个检查：

A. 如果该WindowToken对象所描述的窗口的其中一个处于动画显示过程，即用来描述该窗口的一个WindowState对象的成员函数isAnimating的返回值等于true，那么就需要该WindowToken对象的状态设置为正在退出状态，即将它保存在WindowManagerService类的成员变量mExitingTokens所描述的一个ArrayList中。

B. 如果该WindowToken对象所描述的窗口是可见的，即用来描述该窗口的一个WindowState对象的成员函数isVisibleNow的返回值等于true，那么就需要调用WindowManagerService类的成员函数applyAnimationLocked来给它应用一个退出动画，该退出动画是通过调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来实现的。当一个窗口退出了之后，系统当前获得焦点的窗口可能会发生变化，这时候就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来重新调整系统当前获得焦点的窗口。

注意，如果正在删除的WindowToken对象是用来描述壁纸窗口的，那么还需要将该WindowToken对象从WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中删除。

* 1. 增加WindowState

从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，如下所示：

1. /\*\*
2. \* Mapping from an IWindow IBinder to the server's Window object.
3. \* This is also used as the lock for all of our state.
4. \*/
5. **final** HashMap<IBinder, WindowState> mWindowMap = **new** HashMap<IBinder, WindowState>();
6. ......
8. /\*\*
9. \* Z-ordered (bottom-most first) list of all Window objects.
10. \*/
11. **final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();
12. ......
14. **public** **int** addWindow(Session session, IWindow client,
15. WindowManager.LayoutParams attrs, **int** viewVisibility,
16. Rect outContentInsets, InputChannel outInputChannel) {
17. ......
19. WindowState win = ;
21. **synchronized**(mWindowMap) {
22. ......
24. win = **new** WindowState(session, client, token,
25. attachedWindow, attrs, viewVisibility);
26. ......
28. mWindowMap.put(client.asBinder(), win);
29. ......
31. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
32. mInputMethodWindow = win;
33. addInputMethodWindowToListLocked(win);
34. ......
35. } **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {
36. mInputMethodDialogs.add(win);
37. addWindowToListInOrderLocked(win, **true**);
38. adjustInputMethodDialogsLocked();
39. ......
40. } **else** {
41. addWindowToListInOrderLocked(win, **true**);
42. **if** (attrs.type == TYPE\_WALLPAPER) {
43. .......
44. adjustWallpaperWindowsLocked();
45. } **else** **if** ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {
46. adjustWallpaperWindowsLocked();
47. }
48. }
50. ......
51. }
53. ......
54. }

WindowManagerService类有两个成员变量mWindowMap和mWindows是用来保存系统中的WindowState对象。其中，成员变量mWindowMap指向的是一个HashMap，它的关键字是一个IBinder接口，一般这个IBinder接口指向的是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述一个窗口；成员变量mWindows指向的是一个ArrayList，保存在它里面的WindowState对象是按照其Z轴位置从小到大的顺序排列的。成员变量mWindowMap和mWindows的区别在于，前者给在定一个IBinder接口的情况下，可以快速找到与对应的WindowState对象，而后者用来从上到下或者下到上遍历系统的WindowState对象。由于系统中的WindowState对象是按照其Z轴位置从小到大的顺序排列在成员变量mWindows中的，因此，成员变量mWindows所指向的ArrayList就是我们在前面图1中所说的Window Stack。

理解了WindowManagerService类有两个成员变量mWindowMap和mWindows的作用之后，WindowManagerService类的成员函数addWindow增加一个WindowState对象的过程就容易理解了。

参数client是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述要增加到WindowManagerService服务中的一个窗口。WindowManagerService类的成员函数addWindow首先创建一个WindowState对象win，接着再以参数client所描述的一个Binder代理对象的IBinder接口为关键字，将WindowState对象win保存在WindowManagerService类的成员变量mWindowMap中，最后还会根据要增加到WindowManagerService服务中的窗口的类型来调用不同的成员函数将WindowState对象win增加到WindowManagerService类的成员变量mWindows中：

A. 如果要增加的是输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么就会调用成员函数addInputMethodWindowToListLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow中。

B. 如果要增加的是输入法对话框，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD\_DIALOG，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodDialogs中，以及调用成员函数adjustInputMethodDialogsLocked来调整刚才所添加的输入法窗口在窗口堆栈中的位置，使得它位于系统当前需要输入法窗口的窗口的上面。

C. 如果要增加的是壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会调用成员函数adjustWallpaperWindowsLocked来调整刚才所添加的壁纸窗口在窗口堆栈中的位置，使得它位于系统当前需要壁纸窗口的窗口的下面。

D . 如果要增加的既不是输入法窗口，也不是输入法对话框和壁纸窗口，那么就只会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，但是如果要增加的窗口需要显示壁纸，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SHOW\_WALLPAPER位等于1，那么还会继续调用成员函数adjustWallpaperWindowsLocked来调整系统中的壁纸窗口在窗口堆栈中的位置，使得它位于刚才所添加的窗口的下面。

在后面的两篇文章中，我们再详细分析WindowManagerService类的成员函数addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和adjustWallpaperWindowsLocked的实现，其中，前两者是与输入法窗口相关的，而后者是与壁纸窗口相关的。本文主要关注WindowManagerService类的成员函数addWindowToListInOrderLocked的实现，它会将一个指定的WindowState对象增加到窗口堆栈中的合适位置上去。

## 增加WindowState到窗口堆栈

从前面的分析可以知道，将一个WindowState对象增加到WMS服务内部中的窗口堆栈，即WMS类的成员变量mWindows，是通过调用WMS类的成员函数addWindowToListInOrderLocked来实现的。

WMS类的成员函数addWindowToListInOrderLocked的实现比较复杂，我们先列出它的框架，然后再详细分析它的实现，如下所示：

1. **private** **void** addWindowToListInOrderLocked(WindowState win, **boolean** addToToken) {
2. **final** IWindow client = win.mClient;
3. **final** WindowToken token = win.mToken;
4. **final** ArrayList<WindowState> localmWindows = mWindows;
6. **final** **int** N = localmWindows.size();
7. **final** WindowState attached = win.mAttachedWindow;
8. **int** i;
9. **if** (attached == ) {
10. //CASE 1：要增加的窗口win没有附加在其它窗口上
11. **int** tokenWindowsPos = token.windows.size();
12. **if** (token.appWindowToken != ) {
13. //CASE 1.1：要增加的窗口win是一个Activity窗口
14. **int** index = tokenWindowsPos-1;
15. **if** (index >= 0) {
16. //CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口
17. ......
18. } **else** {
19. //CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口
20. ......
21. }
22. } **else** {
23. //CASE 1.2：要增加的窗口win不是一个Activity窗口
24. ......
25. }
26. **if** (addToToken) {
27. token.windows.add(tokenWindowsPos, win);
28. }
29. } **else** {
30. //CASE 2：要增加的窗口win附加在窗口attached上
31. ......
32. }
34. **if** (win.mAppToken !=  && addToToken) {
35. win.mAppToken.allAppWindows.add(win);
36. }
37. }
39. ......

我们首先分析一下WMS类的成员函数addWindowToListInOrderLocked的几个本地变量的含义：

A. token。本地变量token指向的是参数win所描述的一个WindowState对象的成员变量mToken所指向一个WindowToken对象，这个WindowToken对象用来描述WindowState对象win所对应的窗口令牌。

B. localmWindows。本地变量localmWindows指向的是WMS类的成员变量mWindows所描述的一个ArrayList，即一个窗口堆栈，WMS类的成员函数addWindowToListInOrderLocked的目标就是要将参数win所描述的一个WindowState对象增加到该窗口堆栈的合适位置上去。

C. attached。本地变量attached指向的是参数win所描述的一个WindowState对象的成员变量mAttachedWindow 所指向的一个WindowState对象，如果它的值不等于null，那么就意味参数win所描述的窗口要附加在本地变量attached所描述的窗口上。

D. tokenWindowsPos。本地变量tokenWindowsPos用来描述与窗口令牌token所对应的窗口的数量。

E. token.appWindowToken。从前面Android应用程序窗口（Activity）与WMS服务的连接过程分析一文可以知道，如果一个WindowToken对象的成员变量appWindowToken的值不等于null，那么就意味着该WindowToken对象的实际类型为是AppWindowToken，即它所描述的是一个Activity窗口令牌，这种类型的令牌的特点是在ActivityManagerService服务的Activity组件堆栈中对应有一个ActivityRecord对象，如图1所示。

F. index。本地变量index的值等于tokenWindowsPos-1，如果它的值大于等于0，那么就意味着窗口令牌tokent已经存在其它窗口，否则的话，就意味着窗口令牌tokent尚未存在任何窗口。

从这些本地变量的含义，我们就可以分情况来将参数win所描述的一个WindowState对象增加到WMS服务内部的窗口堆栈的合适位置上去：

CASE 1：要增加的窗口win没有附加在其它窗口上

----CASE 1.1：要增加的窗口win是一个Activity窗口

----CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口。这时候意味着窗口win需要保存在其它已经存在的窗口的附近，因此，我们只要找到这些已经存在的窗口在窗口堆栈中的位置，那么再根据其它属性，就可以将窗口win保存在已经存在的窗口的上面或者下面。

----CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口。虽然这时候窗口win在窗口堆栈中没有位置可以参考，但是它毕竟是一个Activity窗口，我们可以通过与它所对应的AppWindowToken对象在App Token List（即WMS类的成员变量mAppTokens所描述的一个ArrayList）中的位置来获得它窗口堆栈中的位置。回忆我们在前面第3节分析移动AppWindowToken至指定位置的操作时得到的结论：WMS服务内部中的所有WindowState对象都是按照Z轴从位置从小到大排列在WindowState堆栈中的，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。因此，我们只要找到用来描述窗口win的一个AppWindowToken对象（token.appWindowToken）的上一个或者下一个AppWindowToken对象所对应的窗口在窗口堆栈中的位置，那么就可以这个位置为参考，得到窗口win在窗口堆栈中的位置。

----CASE 1.2：要增加的窗口win不是一个Activity窗口。这时候既然要增加的窗口也没有附加在其它窗口上，那么就意味着要增加的窗口win在窗口堆栈中没有位置可以参考，因此，我们就需要根据它的Z轴位置来决定它在窗口堆栈的位置。

CASE 2：要增加的窗口win附加在窗口attached上。这时候就意味着要增加的窗口win要保存在窗口attached的上面，即窗口在窗口堆栈的位置要以窗口attached在窗口堆栈的位置为参考。

从上面的分析就可以知道，CASE 1.1.1、CASE 1.1.2和CASE 2都有一个共同特点，即要增加的窗口win在窗口堆栈的位置有一个参考值，而在CASE 1.2中，要增加的窗口win在窗口堆栈的位置没有参考值，需要通过其Z轴位置来确定。

在分析上述四种情况之前， 我们还需要再说明一下WMS类的成员函数addWindowToListInOrderLocked的参数addToToken的含义。参数addToToken是一个布尔变量，如果它的值等于true，那么就说明需要将参数win所描述的一个WindowState对象添加用来描述它的窗口令牌token的成员变量windows所描述的一个ArrayList中去。注意，窗口令牌token的成员变量windows所描述的一个ArrayList里面所保存的WindowState对象是按照Z轴位置从小到大的顺序来排列的，因此，在将WindowState对象win保存到这个ArrayList之前，首先要按照它的Z轴位置计算得到它在这个ArrayList中的位置tokenWindowsPos。另一方面，在参数addToToken的值等于true，并且参数win所描述的是一个Activity窗口，即它的成员变量mAppToken不等于null的情况下，还需要将参数win所描述的一个WindowState对象保存在用来描述它的窗口令牌，即一个AppWindowToken对象成员变量allAppWindows所描述的一个ArrayList中去，以便可以知道一个AppWindowToken对象对应的Activity窗口都有哪些。

接下来，我们就分别分析这四种情况是如何将窗口win增加窗口堆栈中去的。

CASE 1.1.1对应的代码为：

1. **if** (win.mAttrs.type == TYPE\_BASE\_APPLICATION) {
2. // Base windows go behind everything else.
3. placeWindowBefore(token.windows.get(0), win);
4. tokenWindowsPos = 0;
5. } **else** {
6. AppWindowToken atoken = win.mAppToken;
7. **if** (atoken !=  &&
8. token.windows.get(index) == atoken.startingWindow) {
9. placeWindowBefore(token.windows.get(index), win);
10. tokenWindowsPos--;
11. } **else** {
12. **int** newIdx =  findIdxBasedOnAppTokens(win);
13. **if**(newIdx != -1) {
14. //there is a window above this one associated with the same
15. //apptoken note that the window could be a floating window
16. //that was created later or a window at the top of the list of
17. //windows associated with this token.
18. ......
19. localmWindows.add(newIdx+1, win);
20. mWindowsChanged = **true**;
21. }
22. }
23. }

这段代码又分为三种情况来将参数win所描述的一个WindowState对象添加到窗口堆栈中：

A. 参数win描述的窗口的类型为TYPE\_BASE\_APPLICATION。在一个令牌对应的所有窗口中，类型为TYPE\_BASE\_APPLICATION的窗口位于其它类型的窗口的下面。因此，这段代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌token的窗口列表的第0个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值会被设置为0，表示参数win所描述的一个WindowState对象要保存窗口令牌token的窗口列表的第0个位置上。

B. 参数win描述的一个WindowState对象的成员变量mAppToken的值不等于null，这意味着参数win描述的是一个Activity窗口，这时候如果窗口令牌atoken（与token描述的是同一个窗口令牌）的窗口列表的第index个位置（即最上面的一个位置） 的WindowState对象描述的是一个Activity启动窗口，即与窗口令牌atoken的成员变量startingWindow描述的是同一个窗口，那么就说明窗口令牌atoken的窗口列表的第index个位置的WindowState对象描述的是窗口win的启动窗口。由于一个窗口的启动窗口总是位于它的上面，因此，这段代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌atoken的窗口列表的第index个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值减少1，即相当于是等于index，表示参数win所描述的一个WindowState对象要插入在窗口令牌token的窗口列表的第index个位置上。

C. 参数win所描述的窗口的类型既不是TYPE\_BASE\_APPLICATION，而且它也没有启动窗口，那么这时候就需要将它保存在窗口令牌token的窗口列表的最上面一个窗口的上面。窗口令牌token的窗口列表的最上面一个窗口在窗口堆栈中的位置newIdx是通过调用WMS类的成员函数findIdxBaseOnAppTokens来获得的，这时候参数win所描述的一个WindowState对象就应该保存在窗口堆栈，即变量localmWindows所描述的一个ArrayList的第newIdx+1个位置上。

CASE 1.1.2对应的代码为：

1. // Figure out where the window should go, based on the
2. // order of applications.
3. **final** **int** NA = mAppTokens.size();
4. WindowState pos = ;
5. **for** (i=NA-1; i>=0; i--) {
6. AppWindowToken t = mAppTokens.get(i);
7. **if** (t == token) {
8. i--;
9. **break**;
10. }
12. // We haven't reached the token yet; if this token
13. // is not going to the bottom and has windows, we can
14. // use it as an anchor for when we do reach the token.
15. **if** (!t.sendingToBottom && t.windows.size() > 0) {
16. pos = t.windows.get(0);
17. }
18. }
19. // We now know the index into the apps.  If we found
20. // an app window above, that gives us the position; else
21. // we need to look some more.
22. **if** (pos != ) {
23. // Move behind any windows attached to this one.
24. WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());
25. **if** (atoken != ) {
26. **final** **int** NC = atoken.windows.size();
27. **if** (NC > 0) {
28. WindowState bottom = atoken.windows.get(0);
29. **if** (bottom.mSubLayer < 0) {
30. pos = bottom;
31. }
32. }
33. }
34. placeWindowBefore(pos, win);
35. } **else** {
36. // Continue looking down until we find the first
37. // token that has windows.
38. **while** (i >= 0) {
39. AppWindowToken t = mAppTokens.get(i);
40. **final** **int** NW = t.windows.size();
41. **if** (NW > 0) {
42. pos = t.windows.get(NW-1);
43. **break**;
44. }
45. i--;
46. }
47. **if** (pos != ) {
48. // Move in front of any windows attached to this
49. // one.
50. WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());
51. **if** (atoken != ) {
52. **final** **int** NC = atoken.windows.size();
53. **if** (NC > 0) {
54. WindowState top = atoken.windows.get(NC-1);
55. **if** (top.mSubLayer >= 0) {
56. pos = top;
57. }
58. }
59. }
60. placeWindowAfter(pos, win);
61. placeWindowAfter(pos, win);
62. } **else** {
63. // Just search for the start of this layer.
64. **final** **int** myLayer = win.mBaseLayer;
65. **for** (i=0; i<N; i++) {
66. WindowState w = localmWindows.get(i);
67. **if** (w.mBaseLayer > myLayer) {
68. **break**;
69. }
70. }
71. ......
72. localmWindows.add(i, win);
73. mWindowsChanged = **true**;
74. }
75. }

这段代码要能冠军WMS服务内部的一个AppWindowToken列表mAppTokens来在窗口堆栈中找到一个参数位置来保存参数win所描述的一个WindowState对象。

最上面的一个for循环执行完成之后，我们假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图2所示：



图2 窗口win位于窗口C的下面

这时候位于令牌token上面的令牌在窗口堆栈中对应有WindowState对象。注意，这时候第i+2个令牌在窗口堆栈中不对应有WindowState对象，而第i+3个令牌在窗口堆栈中对应有C和D两个WindowState对象，并且这两个WindowState对象所描述的窗口都不是即将要切换到窗口堆栈的底部的。由于第i+3个令牌位于令牌token的上面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i+3个令牌所对应的Z轴位置最小的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象C在窗口堆栈中的位置为参考，而WindowState对象C也正好是变量pos所指向的WindowState对象。

接下来，上述代码会继续检查WindowState对象C是否附加有SubLayer值小于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最小的那个WindowState对象，这是因为该WindowState对象是在WindowState对象C的最下面的，并且它与WindowState对象C是同属一个令牌的。最后，上述代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的下面。

假设最上面的一个for循环执行完成之后，变量pos的值等于null，那么就说明位于令牌token上面的令牌在窗口堆栈中都没有对应有WindowState对象，或者说它们所对应的WindowState对象都是即将要切换到窗口堆栈的底部去的，这时候就需要通过位于令牌token上面的令牌来在窗口堆栈中找到一个参考位置来保存参数win所描述的WindowState对象，这是通过中间的while循环来实现的。

中间的while循环执行完成之后，假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图3所示：



图3 窗口win位于窗口D的上面

这时候位于令牌token上面的令牌在窗口堆栈中没有对应有WindowState对象。注意，这时候第i-1个令牌在窗口堆栈中不对应有WindowState对象，而第i-2个令牌在窗口堆栈中对应有C和D两个WindowState对象。由于第i-2个令牌位于令牌token的下面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i-2个令牌所对应的Z轴位置最大的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象D在窗口堆栈中的位置为参考，而WindowState对象D也正好是变量pos所指向的WindowState对象。

接下来，上述代码会继续检查WindowState对象D是否附加有SubLayer值大于等于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最大的那个WindowState对象，这是因为该WindowState对象是在WindowState对象D的最上面的，并且它与WindowState对象D是同属一个令牌的。最后，上述代码就会调用WMS类的成员函数placeWindowAfter来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的上面。

假设中间的while循环执行完成之后，变量pos的值等于null，这时候就说明在窗口堆栈中实在是找不到参考位置来保存参数win所描述的WindowState对象了，因此，就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如最下面的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，最下面的for循环只要从下到上找到一个Z轴位置比参数win所描述的WindowState对象的Z轴位置大的一个WindowState对象在窗口堆栈中的位置i，那么就可以将参数win所描述的WindowState对象插入在窗口堆栈的第i个位置上了。

CASE 1.2对应的代码为：

1. // Figure out where window should go, based on layer.
2. **final** **int** myLayer = win.mBaseLayer;
3. **for** (i=N-1; i>=0; i--) {
4. **if** (localmWindows.get(i).mBaseLayer <= myLayer) {
5. i++;
6. **break**;
7. }
8. }
9. **if** (i < 0) i = 0;
10. ......
11. localmWindows.add(i, win);
12. mWindowsChanged = **true**;

由于这时候在窗口堆栈中是没有参考位置来保存参数win所描述的WindowState对象的，因此，这段代码就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如这段代码中的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，这段代码中的for循环只要从上到下找到一个WindowState对象，它的Z轴位置小于或者等于参数win所描述的WindowState对象的Z轴位置，那么该WindowState对象在窗口堆栈中的位置i就可以用插入参数win所描述的WindowState对象了。

CASE 2对应的代码为：

1. // Figure out this window's ordering relative to the window
2. // it is attached to.
3. **final** **int** NA = token.windows.size();
4. **final** **int** sublayer = win.mSubLayer;
5. **int** largestSublayer = Integer.MIN\_VALUE;
6. WindowState windowWithLargestSublayer = ;
7. **for** (i=0; i<NA; i++) {
8. WindowState w = token.windows.get(i);
9. **final** **int** wSublayer = w.mSubLayer;
10. **if** (wSublayer >= largestSublayer) {
11. largestSublayer = wSublayer;
12. windowWithLargestSublayer = w;
13. }
14. **if** (sublayer < 0) {
15. // For negative sublayers, we go below all windows
16. // in the same sublayer.
17. **if** (wSublayer >= sublayer) {
18. **if** (addToToken) {
19. token.windows.add(i, win);
20. }
21. placeWindowBefore(
22. wSublayer >= 0 ? attached : w, win);
23. **break**;
24. }
25. } **else** {
26. // For positive sublayers, we go above all windows
27. // in the same sublayer.
28. **if** (wSublayer > sublayer) {
29. **if** (addToToken) {
30. token.windows.add(i, win);
31. }
32. placeWindowBefore(w, win);
33. **break**;
34. }
35. }
36. }
37. **if** (i >= NA) {
38. **if** (addToToken) {
39. token.windows.add(win);
40. }
41. **if** (sublayer < 0) {
42. placeWindowBefore(attached, win);
43. } **else** {
44. placeWindowAfter(largestSublayer >= 0
45. ? windowWithLargestSublayer
46. : attached,
47. win);
48. }
49. }

这段代码要将参数win所描述的WindowState对象附加在变量attached所描述的WindowState对象的上面或者下面，取决于它的成员变量mSubLayer的值是大于0还是小于0。我们分四种情况来考虑。

第一种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于等于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图4和图5所示：



图4 窗口win插入到窗口B的下面



图5 窗口win插入在窗口attached的下面

在图4和图5中，WindowState对象A和B均是附加在WindowState对象attached中。

在图4中，WindowState对象A和B的成员变量mSubLayer的值均小于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

在图5中，WindowState对象A和B的成员变量mSubLayer的值均大于0，由于WindowState对象win的成员变量mSubLayer的值小于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

第二种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图6所示：



图6 窗口win插入在窗口B的下面

在图6中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B的成员变量mSubLayer的值均大于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

第三种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图7所示：



图7 窗口win插入在窗口attached的下面

        在图7中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B以及win的成员变量mSubLayer的值均小于0，但是WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

        第四种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于等于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图8和图9所示：



图8 窗口win插入在窗口B的上面



图9 窗口win插入在窗口attached的上面

图9 窗口win插入在窗口attached的上面

        在图8和图9中，WindowState对象A和B均是附加在WindowState对象attached中。

        在图8中，WindowState对象A和B的成员变量mSubLayer的值均大于0，并且WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的上面，这是通过调用WMS类的成员函数placeWindowAfter来实现的。

        在图9中，WindowState对象A和B的成员变量mSubLayer的值均小于等于0，而WindowState对象win的成员变量mSubLayer的值大于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的上面，这是通过调用WMS类的成员函数placeWindowAfter来实现的。

         注意，在这四种情况中，如果参数addToToken的值等于true，那么都需要将参数win所描述的WindowState对象增加到与它所对应的窗口令牌token的窗口列表windows中去。

         10. 删除WindowState

          删除WindowState是通过调用WMS类的成员函数tmpRemoveWindowLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **int** tmpRemoveWindowLocked(**int** interestingPos, WindowState win) {
6. **int** wpos = mWindows.indexOf(win);
7. **if** (wpos >= 0) {
8. **if** (wpos < interestingPos) interestingPos--;
9. ......
10. mWindows.remove(wpos);
11. mWindowsChanged = **true**;
12. **int** NC = win.mChildWindows.size();
13. **while** (NC > 0) {
14. NC--;
15. WindowState cw = win.mChildWindows.get(NC);
16. **int** cpos = mWindows.indexOf(cw);
17. **if** (cpos >= 0) {
18. **if** (cpos < interestingPos) interestingPos--;
19. ......
20. mWindows.remove(cpos);
21. }
22. }
23. }
24. **return** interestingPos;
25. }
27. ......
28. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        WMS类的成员函数tmpRemoveWindowLocked将参数win所描述的窗口及其子窗口从WMS服务内部的窗口堆栈中删除，即从 WMS类的成员变量mWindows所描述的一个ArrayList中删除。

        如果每一个被删除的窗口在窗口堆栈中的位置比参数interestingPos的值小，那么WMS类的成员函数tmpRemoveWindowLocked还会将参数interestingPos的值减少1，这相当于是计算当删除参数win所描述的窗口及其子窗口之后，原来位于窗口堆栈中第interestingPos个位置的窗口现在位于窗口堆栈的位置，这个位置最终会作为WMS类的成员函数tmpRemoveWindowLocked的返回值。

       11. 在指定位置增加WindowState

       在指定位置增加WindowState是通过调用WMS类的成员函数reAddWindowLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **int** reAddWindowLocked(**int** index, WindowState win) {
6. **final** **int** NCW = win.mChildWindows.size();
7. **boolean** added = **false**;
8. **for** (**int** j=0; j<NCW; j++) {
9. WindowState cwin = win.mChildWindows.get(j);
10. **if** (!added && cwin.mSubLayer >= 0) {
11. ......
12. mWindows.add(index, win);
13. index++;
14. added = **true**;
15. }
16. ......
17. mWindows.add(index, cwin);
18. index++;
19. }
20. **if** (!added) {
21. ......
22. mWindows.add(index, win);
23. index++;
24. }
25. mWindowsChanged = **true**;
26. **return** index;
27. }
29. ......
30. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        参数win描述的即为要增加的WindowState对象，而参数index描述的即为要将参数win所描述的WindowState对象及其子WindowState对象要增加到窗口堆栈中的起始位置。

       由于参数win所描述的WindowState对象的子WindowState对象的成员变量mSubLayer的值可能会小于0，也可能大于0。大于0的子WindowState对象位于参数win所描述的WindowState对象的上面，而小于0的子WindowState对象位于参数win所描述的WindowState对象的下面。因此，WMS类的成员函数reAddWindowLocked先增加那些小于0的子WindowState对象，接着再增加参数win所描述的WindowState对象，最后增加那些大于0的子WindowState对象。

        假设WMS类的成员函数reAddWindowLocked一共在窗口堆栈中增加了N个WindowState对象，那么它的返回值就等于index + N，这样调用者就可以知道参数win所描述的WindowState对象及其子WindowState对象在窗口堆栈中的最高位置是多少。

        基于第9、第10和第11这三操作，可以组合成很多其它的WindowState操作，如接下来的第12、第13、第14和第15个操作所示。

        12. 将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中

         将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中是通过调用WMS类的成员函数reAddWindowToListInOrderLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** reAddWindowToListInOrderLocked(WindowState win) {
6. addWindowToListInOrderLocked(win, **false**);
7. // This is a hack to get all of the child windows added as well
8. // at the right position.  Child windows should be rare and
9. // this case should be rare, so it shouldn't be that big a deal.
10. **int** wpos = mWindows.indexOf(win);
11. **if** (wpos >= 0) {
12. ......
13. mWindows.remove(wpos);
14. mWindowsChanged = **true**;
15. reAddWindowLocked(wpos, win);
16. }
17. }
19. ......
20. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        为了得到参数win所描述的WindowState对象的子WindowState对象在窗口堆栈中的起始位置，WMS类的成员函数reAddWindowToListInOrderLocked首先将参数win所描述的WindowState对象增加到窗口堆栈中，这是通过调用前面所分析的成员函数addWindowToListInOrderLocked来实现的，目的是为了获得它在窗口堆栈的位置。有了这个位置之后，WMS类的成员函数reAddWindowToListInOrderLocked就可以调用前面所分析的成员函数reAddWindowLocked来将WindowState对象及其所有子WindowState对象增加到窗口堆栈中去了，不过在调用之前，要先将参数win所描述的WindowState对象从窗口中堆栈删除。

        13. 将一个WindowToken对象对应的所有WindowState对象及其子WindowState对象增加到窗口堆栈的指定位置上

         将一个WindowToken对象对应的所有WindowState对象都增加到窗口堆栈中是通过调用WMS类的成员函数reAddAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **int** reAddAppWindowsLocked(**int** index, WindowToken token) {
6. **final** **int** NW = token.windows.size();
7. **for** (**int** i=0; i<NW; i++) {
8. index = reAddWindowLocked(index, token.windows.get(i));
9. }
10. **return** index;
11. }
13. ......
14. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        与参数token所描述的WindowToken对象所对应的WindowState对象保存在它的成员变量windows所描述的一个ArrayList中。通过遍历这个ArrayList，就可以将与参数token所描述的WindowToken对象所对应的WindowState对象及其子WindowState对象都增加到窗口堆栈的指定的起始位置上去，这是通过调用前面所分析的成员函数reAddWindowLocked来实现的。

        参数index描述的便是最初指定的起始位置，每一次调用WMS类的成员函数reAddWindowLocked之后，它的值都便会被更新为下一个WindowState对象及其子WindowState对象要增加到窗口堆栈中的位置。

        最后，WMS类的成员函数reAddAppWindowsLocked将与参数token所描述的WindowToken对象所对应的WindowState对象在窗口堆栈中的最高位置加1后的得到结果返回给调用者。

       14. 将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

        将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上是通过调用WMS类的成员函数moveAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** moveAppWindowsLocked(AppWindowToken wtoken, **int** tokenPos,
6. **boolean** updateFocusAndLayout) {
7. // First remove all of the windows from the list.
8. tmpRemoveAppWindowsLocked(wtoken);
10. // Where to start adding?
11. **int** pos = findWindowOffsetLocked(tokenPos);
13. // And now add them back at the correct place.
14. pos = reAddAppWindowsLocked(pos, wtoken);
16. **if** (updateFocusAndLayout) {
17. **if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {
18. assignLayersLocked();
19. }
20. mLayoutNeeded = **true**;
21. performLayoutAndPlaceSurfacesLocked();
22. }
23. }
25. ......
26. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        参数wtoken描述的是要移动其所对应的WindowState对象的一个AppWindowToken对象，而参数tokenPos描述的是该AppWindowToken对象在WMS服务内部的AppWindowToken列表中的新位置。

        WMS类的成员函数moveAppWindowsLocked首先调用前面所分析的成员函数tmpRemoveAppWindowsLocked来移除所有与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象，接着再调用也是前面所分析的成员函数findWindowOffsetLocked来获得与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象在窗口堆栈中的起始位置。有了这个起始位置之后，就可以也是前面所分析的成员函数reAddAppWindowsLocked来将与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈上去了。

        最后，如果参数updateFocusAndLayout的值等于true，那么WMS类的成员函数moveAppWindowsLocked还会更新系统当前获得焦点的窗口，以及重新计算系统中的所有窗口的Z轴位置以及重新布局系统中的所有窗口，这三个操作分别是通过调用WMS类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来实现的。

        15. 将一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

         将一组AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈的指定位置上是通过调用WMS类的另外一个版本的成员函数moveAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** moveAppWindowsLocked(List<IBinder> tokens, **int** tokenPos) {
6. // First remove all of the windows from the list.
7. **final** **int** N = tokens.size();
8. **int** i;
9. **for** (i=0; i<N; i++) {
10. WindowToken token = mTokenMap.get(tokens.get(i));
11. **if** (token != ) {
12. tmpRemoveAppWindowsLocked(token);
13. }
14. }
16. // Where to start adding?
17. **int** pos = findWindowOffsetLocked(tokenPos);
19. // And now add them back at the correct place.
20. **for** (i=0; i<N; i++) {
21. WindowToken token = mTokenMap.get(tokens.get(i));
22. **if** (token != ) {
23. pos = reAddAppWindowsLocked(pos, token);
24. }
25. }
27. **if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {
28. assignLayersLocked();
29. }
30. mLayoutNeeded = **true**;
31. performLayoutAndPlaceSurfacesLocked();
33. //dump();
34. }
36. ......
37. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        这个操作与前面分析的第14个操作是类似，区别只在于前者是批量地移动一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象，而后者是只移动一个AppWindowToken对象所对应的WindowState对象及其子WindowState对象，此外，前者总是会调用WMS类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来更新系统当前获得焦点的窗口、以及重新计算每一个窗口的Z轴位置，并且对这些窗口进行重新布局。

        至此，我们就分析完成WMS服务组织系统中的窗口的方式了。从分析的过程中，可以得到以下结论：

        1. WMS服务维护有一个AppWindowToken堆栈和一个WindowState堆栈，它们与ActivityManagerService服务维护的Actvity堆栈是有关相同的Z轴位置关系的。

        2. ActivityManagerService服务中的每一个ActivityRecord对象在WMS服务中都对应有一个AppWindowToken对象，而WMS服务中的每一个AppWindowToken对象都对应有一组WindowState对象。

        3. 在WindowState堆栈中，AppWindowToken堆栈中的第i+1个AppWindowToken对象所对应的WindowState对象都位于第i个AppWindowToken对象所对应的WindowState对象的上面。

        4. 一个WindowState对象可以附加在另外一个WindowState对象上面，此外，一个WindowState对象还可以有子WindowState对象，它们都是与同一个AppWindowToken对象或者WindowToken对象所对应的。

        5. WMS服务有两个特殊的WindowToken，它们分别用来描述系统中的输入法窗口令牌和壁纸窗口令牌，其中，输入法窗口位于需要输入法的窗口的上面，而壁纸窗口位于需要壁纸的窗口的下面。

        最后，我们可以将WMS服务中的AppWindowToken理解成一个Activity组件令牌，而将它所对应的WindowState对象理解成一个Activity窗口。有了这些概念之后，就为学习WMS服务的各种实现打下坚实的基础。在接下来的两篇文章中，我们就会在本文的基础上，继续分析WMS服务是如何管理系统中的输入法窗口和壁纸窗口的，敬请关注！

## 实例分析：显示一个系统dlg流

new WindowState()

## REF

[Android窗口管理服务WMS对窗口的组织方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)

<http://www.jianshu.com/p/40776c123adb>

<http://gityuan.com/2017/01/08/windowmanger/>

# NoticationManagerService

## 如何设置静音模式

NotificationManager.java

在setNotificationPolicy（...）方法内部，通过AIDL方式获得了一个service，该service的具体实现是在NotificationManagerService.java中。由service调用setNotificationPolicy（...）

### NoticationManagerService. setNotificationPolicy

IBinder的实现类中，被调用的方法setNotificationPolicy（...）里面，可以看到代码mZenModeHelper.setNotificationPolicy(policy)，一步步深入。

### ZenModeHelper. setNotificationPolicy

public void setNotificationPolicy(Policy policy) {

if (policy == null || mConfig == null) return;

synchronized (mConfig) {

final ZenModeConfig newConfig = mConfig.copy();

newConfig.applyNotificationPolicy(policy);

setConfigLocked(newConfig, "setNotificationPolicy");

}

}

获得了一个ZenModeCogfig的对象，并调用applyNotificationPolicy方法。

### ZenModeConfig.applyNotificationPolicy(Policy policy)

enModeConfig.java

applyNotificationPolicy(Policy policy)，该方法主要是对Policy进行解析，将解析出来的值赋给各个变量。然后在回到ZenModeHelper.java。

### ZenModeHelper.setConfigLocked

ZenModeHelper.java

setConfigLocked(newConfig, "setNotificationPolicy")方法中先对配置文件进行空判断和有效判断。然后判断配置文件相对于原始是否有变得，如果有，调用dispatchOnConfigChanged()，该方法内部是一个回调。

private void dispatchOnConfigChanged() {

for (Callback callback : mCallbacks) {

callback.onConfigChanged();

}

}

所以我们再次回到NotificationManagerService.java

### NotificationManagerService.onConfigChanged

找到onConfigChanged()，里面调用了savePolicyFile()，该方法通过handle发送消息，处理该消息时调用handleSavePolicyFile()，该方法内部调用writePolicyXml()，该方法通过ZenModeHelper对象再继续调用writeXml(...)；于是再次回到ZenModeHelper.java

### ZenModeHelper. writeXml

找到writeXml(...)方法，里面遍历每一个ZenModeConfig对象，逐个调用ZenModeConfig对象的writeXml(...)方法。再次回到ZenModeConfig.java。

### ZenModeConfig.java

恭喜你，终于到头了。原来真正写数据的地方在这里。系统将之前解析出来的数据信息，也就是用户设定的各种勿扰的配置信息写在xml里面。好了，到此为止，另外如何使这些数据生效，请等待笔者的下一篇文章吧！

### REF

<https://blog.csdn.net/firedancer0089/article/details/78224249>

<https://blog.csdn.net/w690333243/article/details/77942222>

注明出处。

# 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

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

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

## 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)

# 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

# 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)

# 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