# Android系统启动

## 概述

Android系统底层基于Linux Kernel, 当Kernel启动过程会创建init进程, 该进程是uoyou用户空间的鼻祖, init进程会启动servicemanager(binder服务管家), Zygote进程(Java进程的鼻祖). Zygote进程会创建 system\_server进程以及各种app进程.



## init

[init](http://gityuan.com/2016/02/05/android-init/)是Linux系统中用户空间的第一个进程(pid=1), Kerner启动后会调用/system/core/init/Init.cpp的main()方法.

root@zs600b:/etc/wifi # ps | grep init

root 1 0 748 392 c01191c8 00035888 S /init

### Init.main

|  |
| --- |
| int main(int argc, char\*\* argv) {  ...  klog\_init(); //初始化kernel log  property\_init(); //创建一块共享的内存空间，用于属性服务  signal\_handler\_init(); //初始化子进程退出的信号处理过程  property\_load\_boot\_defaults(); //加载/default.prop文件  start\_property\_service(); //启动属性服务器(通过socket通信)  init\_parse\_config\_file("/init.rc"); //解析init.rc文件  //执行rc文件中触发器为 on early-init的语句  action\_for\_each\_trigger("early-init", action\_add\_queue\_tail);  //执行rc文件中触发器为 on init的语句  action\_for\_each\_trigger("init", action\_add\_queue\_tail);  //执行rc文件中触发器为 on late-init的语句  action\_for\_each\_trigger("late-init", action\_add\_queue\_tail);  while (true) {  if (!waiting\_for\_exec) {  execute\_one\_command();  restart\_processes();  }  int timeout = -1;  if (process\_needs\_restart) {  timeout = (process\_needs\_restart - gettime()) \* 1000;  if (timeout < 0)  timeout = 0;  }  if (!action\_queue\_empty() || cur\_action) {  timeout = 0;  }  epoll\_event ev;  //循环 等待事件发生  int nr = TEMP\_FAILURE\_RETRY(epoll\_wait(epoll\_fd, &ev, 1, timeout));  if (nr == -1) {  ERROR("epoll\_wait failed: %s\n", strerror(errno));  } else if (nr == 1) {  ((void (\*)()) ev.data.ptr)();  }  }  return 0;  } |

init进程的主要功能点:

* 分析和运行所有的init.rc文件;
* 生成设备驱动节点; （通过rc文件创建）
* 处理子进程的终止(signal方式);
* 提供属性服务property service。

### 启动Zygote

当init解析到下面这条语句,便会启动Zygote进程

|  |
| --- |
| service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server  class main //伴随着main class的启动而启动  socket zygote stream 660 root system //创建socket  onrestart write /sys/android\_power/request\_state wake  onrestart write /sys/power/state on  onrestart restart media //当zygote重启时,则会重启media  onrestart restart netd // 当zygote重启时,则会重启netd  } |

当init子进程(Zygote)退出时，会产生SIGCHLD信号，并发送给init进程，通过socket套接字传递数据，调用到wait\_for\_one\_process()方法，根据是否是oneshot，来决定是重启子进程，还是放弃启动。由于缺省模式oneshot=false,因此**Zygote一旦被杀便会再次由init进程拉起**.



接下来,便是进入了Zygote进程.

## Zygote

当[Zygote](http://gityuan.com/2016/02/13/android-zygote/)进程启动后, 便会执行到frameworks/base/cmds/app\_process/App\_main.cpp文件的main()方法. 整个调用流程:

|  |
| --- |
| App\_main.main  AR.start  AR.startVm  AR.startReg  ZygoteInit.main (首次进入Java世界)  registerZygoteSocket  preload  startSystemServer  runSelectLoop |

### App\_main.main

|  |
| --- |
| int main(int argc, char\* const argv[])  {  AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));  while (i < argc) {  ...//参数解析  }  //设置进程名  if (!niceName.isEmpty()) {  runtime.setArgv0(niceName.string());  set\_process\_name(niceName.string());  }  if (zygote) {  // 启动AppRuntime  runtime.start("com.android.internal.os.ZygoteInit", args, zygote);  }  ...  } |

### AndroidRuntime::start

|  |
| --- |
| void AndroidRuntime::start(const char\* className, const Vector<String8>& options)  {  ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",  className != NULL ? className : "(unknown)");  ...  // 虚拟机创建  if (startVm(&mJavaVM, &env, zygote) != 0) {  return;  }  onVmCreated(env);  // JNI方法注册  if (startReg(env) < 0) {  return;  }  ...  // 调用ZygoteInit.main()方法[见小节3.3]  env->CallStaticVoidMethod(startClass, startMeth, strArray); |

### ZygoteInit.main

|  |
| --- |
| public static void main(String argv[]) {  try {  ...  registerZygoteSocket(socketName); //为Zygote注册socket  preload(); // 预加载类和资源[见小节3.4]  ...  if (startSystemServer) {  startSystemServer(abiList, socketName);//启动system\_server【见小节3.6】  }  Log.i(TAG, "Accepting command socket connections");  runSelectLoop(abiList); //进入循环模式[见小节3.5]  ...  } catch (MethodAndArgsCaller caller) {  caller.run(); //启动system\_server中会讲到。  }  ...  } |

### ZygoteInit.preload

|  |
| --- |
| static void preload() {  Log.d(TAG, "begin preload");  preloadClasses();  preloadResources();  preloadOpenGL();  preloadSharedLibraries();  WebViewFactory.prepareWebViewInZygote();  Log.d(TAG, "end preload");  } |

### ZygoteInit.runSelectLoop

|  |
| --- |
| private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {  ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();  ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();    //sServerSocket是socket通信中的服务端，即zygote进程  fds.add(sServerSocket.getFileDescriptor());  peers.add(null);  while (true) {  StructPollfd[] pollFds = new StructPollfd[fds.size()];  for (int i = 0; i < pollFds.length; ++i) {  pollFds[i] = new StructPollfd();  pollFds[i].fd = fds.get(i);  pollFds[i].events = (short) POLLIN;  }  ...  Os.poll(pollFds, -1);  for (int i = pollFds.length - 1; i >= 0; --i) {  //采用I/O多路复用机制，当客户端发出连接请求或者数据处理请求时，跳过continue，执行后面的代码  if ((pollFds[i].revents & POLLIN) == 0) {  continue;  }  if (i == 0) {  //创建客户端连接  ZygoteConnection newPeer = acceptCommandPeer(abiList);  peers.add(newPeer);  fds.add(newPeer.getFileDesciptor());  } else {  //处理客户端数据事务  boolean done = peers.get(i).runOnce();  if (done) {  peers.remove(i);  fds.remove(i);  }  }  }  }  } |

### ZygoteInit.startSystemServer

|  |
| --- |
| private static boolean startSystemServer(String abiList, String socketName)  throws MethodAndArgsCaller, RuntimeException {  ...  // fork子进程system\_server  pid = Zygote.forkSystemServer(  parsedArgs.uid, parsedArgs.gid,  parsedArgs.gids,  parsedArgs.debugFlags,  null,  parsedArgs.permittedCapabilities,  parsedArgs.effectiveCapabilities);  ...  if (pid == 0) {  if (hasSecondZygote(abiList)) {  waitForSecondaryZygote(socketName);  }  //进入system\_server进程[见小节4.1]  handleSystemServerProcess(parsedArgs);  }  return true;  } |

Zygote进程创建Java虚拟机,并注册JNI方法, 真正成为Java进程的母体,用于孵化Java进程. 在创建完[小节4.1]system\_server进程后,zygote功成身退，调用runSelectLoop()，随时待命，当接收到请求创建新进程请求时立即唤醒并执行相应工作。

<http://gityuan.com/2016/02/01/android-booting/>

## system\_server

Zygote通过fork后创建system\_server进程。

### handleSystemServerProcess

|  |
| --- |
| private static void handleSystemServerProcess(  ZygoteConnection.Arguments parsedArgs)  throws ZygoteInit.MethodAndArgsCaller {  ...  if (parsedArgs.niceName != null) {  //设置当前进程名为"system\_server"  Process.setArgV0(parsedArgs.niceName);  }  final String systemServerClasspath = Os.getenv("SYSTEMSERVERCLASSPATH");  if (systemServerClasspath != null) {  //执行dex优化操作,比如services.jar  performSystemServerDexOpt(systemServerClasspath);  }  if (parsedArgs.invokeWith != null) {  ...  } else {  ClassLoader cl = null;  if (systemServerClasspath != null) {  cl = new PathClassLoader(systemServerClasspath, ClassLoader.getSystemClassLoader());  Thread.currentThread().setContextClassLoader(cl);  }  //[见小节4.2]  RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);  }  } |

system\_server进程创建PathClassLoader类加载器.

### RuntimeInit.zygoteInit

|  |
| --- |
| public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)  throws ZygoteInit.MethodAndArgsCaller {  Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "RuntimeInit");  redirectLogStreams(); //重定向log输出  commonInit(); // 通用的一些初始化  nativeZygoteInit(); // zygote初始化  applicationInit(targetSdkVersion, argv, classLoader); // [见小节3.4]  } |

nativeZygoteInit()方法经过层层调用,会进入app\_main.cpp中的onZygoteInit()方法, Binder线程池的创建也是在这个过程,如下:

|  |
| --- |
| virtual void onZygoteInit()  {  sp<ProcessState> proc = ProcessState::self();  proc->startThreadPool(); //启动新binder线程  } |

applicationInit()方法经过层层调用,会抛出异常ZygoteInit.MethodAndArgsCaller(m, argv), ZygoteInit.main() 会捕捉该异常, 见下文.

### ZygoteInit.main

ZygoteInit.java

|  |
| --- |
| public static void main(String argv[]) {  try {  startSystemServer(abiList, socketName); //抛出MethodAndArgsCaller异常  ....  } catch (MethodAndArgsCaller caller) {  caller.run(); //此处通过反射,会调用SystemServer.main()方法 [见小节4.4]  } catch (RuntimeException ex) {  ...  }  } |

采用抛出异常的方式,用于栈帧清空,提供利用率, 以至于现在大家看到的每个Java进程的调用栈如下:

|  |
| --- |
| ...  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) |

### 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() {  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) |

**六. 实战分析**

以下列举启动部分重要进程以及关键节点会打印出的log

/system/bin/vold: 383

/system/bin/lmkd: 432

/system/bin/surfaceflinger: 434

/system/bin/debuggerd64: 537

/system/bin/mediaserver: 540

/system/bin/installd: 541

/system/vendor/bin/thermal-engine: 552

zygote64: 557

zygote: 558

system\_server: 1274

**1. before zygote**

//启动vold, 再列举当前系统所支持的文件系统. 执行到system/vold/main.cpp的main()

11-23 14:36:47.474 383 383 I vold : Vold 3.0 (the awakening) firing up

11-23 14:36:47.475 383 383 V vold : Detected support for: ext4 vfat

//使用内核的lmk策略

11-23 14:36:47.927 432 432 I lowmemorykiller: Using in-kernel low memory killer interface

//启动SurfaceFlinger

11-23 14:36:48.041 434 434 I SurfaceFlinger: SurfaceFlinger is starting

11-23 14:36:48.042 434 434 I SurfaceFlinger: SurfaceFlinger's main thread ready to run. Initializing graphics H/W...

// 开机动画

11-23 14:36:48.583 508 508 I BootAnimation: bootanimation launching ...

// debuggerd

11-23 14:36:50.306 537 537 I : debuggerd: starting

// installd启动

11-23 14:36:50.311 541 541 I installd: installd firing up

// thermal守护进程

11-23 14:36:50.369 552 552 I ThermalEngine: Thermal daemon started

**2. zygote**

// Zygote64进程(Zygote): AndroidRuntime::start

11-23 14:36:51.260 557 557 D AndroidRuntime: >>>>>> START com.android.internal.os.ZygoteInit uid 0 <<<<<<

// Zygote64进程: AndroidRuntime::startVm

11-23 14:36:51.304 557 557 D AndroidRuntime: CheckJNI is OFF

// 执行ZygoteInit.preload()

11-23 14:36:52.134 557 557 D Zygote : begin preload

// 执行ZygoteInit.preloadClasses(), 预加载3860个classes, 花费时长746ms

11-23 14:36:52.134 557 557 I Zygote : Preloading classes...

11-23 14:36:52.881 557 557 I Zygote : ...preloaded 3860 classes in 746ms.

// 执行ZygoteInit.preloadClasses(), 预加载86组资源, 花费时长179ms

11-23 14:36:53.114 557 557 I Zygote : Preloading resources...

11-23 14:36:53.293 557 557 I Zygote : ...preloaded 86 resources in 179ms.

// 执行ZygoteInit.preloadSharedLibraries()

11-23 14:36:53.494 557 557 I Zygote : Preloading shared libraries...

11-23 14:36:53.503 557 557 D Zygote : end preload

// 执行com\_android\_internal\_os\_Zygote\_nativeForkSystemServer(),成功fork出system\_server进程

11-23 14:36:53.544 557 557 I Zygote : System server process 1274 has been created

// Zygote开始进入runSelectLoop()

11-23 14:36:53.546 557 557 I Zygote : Accepting command socket connections

**3. system\_server**

//进入system\_server, 建立跟Zygote进程的socket通道

11-23 14:36:53.586 1274 1274 I Zygote : Process: zygote socket opened, supported ABIS: armeabi-v7a,armeabi

// 执行SystemServer.run()

11-23 14:36:53.618 1274 1274 I SystemServer: Entered the Android system server! <===> boot\_progress\_system\_run

// 等待installd准备就绪

11-23 14:36:53.707 1274 1274 I Installer: Waiting for installd to be ready.

//服务启动

11-23 14:36:53.732 1274 1274 I ActivityManager: Memory class: 192

//phase100

11-23 14:36:53.883 1274 1274 I SystemServiceManager: Starting phase 100

11-23 14:36:53.902 1274 1274 I SystemServer: Package Manager

11-23 14:37:03.816 1274 1274 I SystemServer: User Service

...

11-23 14:37:03.940 1274 1274 I SystemServer: Init Watchdog

11-23 14:37:03.941 1274 1274 I SystemServer: Input Manager

11-23 14:37:03.946 1274 1274 I SystemServer: Window Manager

...

11-23 14:37:04.081 1274 1274 I SystemServiceManager: Starting com.android.server.MountService$Lifecycle

11-23 14:37:04.088 1274 2717 D MountService: Thinking about reset, mSystemReady=false, mDaemonConnected=true

11-23 14:37:04.088 1274 1274 I SystemServiceManager: Starting com.android.server.UiModeManagerService

11-23 14:37:04.520 1274 1274 I SystemServer: NetworkTimeUpdateService

//phase480 && 500

11-23 14:37:05.056 1274 1274 I SystemServiceManager: Starting phase 480

11-23 14:37:05.061 1274 1274 I SystemServiceManager: Starting phase 500

11-23 14:37:05.231 1274 1274 I ActivityManager: System now ready <==> boot\_progress\_ams\_ready

11-23 14:37:05.234 1274 1274 I SystemServer: Making services ready

11-23 14:37:05.243 1274 1274 I SystemServer: WebViewFactory preparation

//phase550

11-23 14:37:05.234 1274 1274 I SystemServiceManager: Starting phase 550

11-23 14:37:05.237 1274 1288 I ActivityManager: Force stopping com.android.providers.media appid=10010 user=-1: vold reset

//Phase600

11-23 14:37:06.066 1274 1274 I SystemServiceManager: Starting phase 600

11-23 14:37:06.236 1274 1274 D MountService: onStartUser 0

**4. logcat小技巧**

通过adb bugreport抓取log信息.先看zygote是否起来, 再看system\_server主线程的运行情况,再看ActivityManager情况

adb logcat -s Zygote

adb logcat -s SystemServer

adb logcat -s SystemServiceManager

adb logcat | grep "1359 1359" //system\_server情况

adb logcat -s ActivityManager

现场调试命令

1. cat proc/[pid]/stack ==> 查看kernel调用栈
2. debuggerd -b [pid] ==> 也不可以不带参数-b, 则直接输出到/data/tombstones/目录
3. kill -3 [pid] ==> 生成/data/anr/traces.txt文件
4. lsof [pid] ==> 查看进程所打开的文件

**七. 总结**

各大核心进程启动后，都会进入各种对象所相应的main()方法，如下

| **进程** | **主方法** |
| --- | --- |
| init进程 | Init.main() |
| zygote进程 | ZygoteInit.main() |
| app\_process进程 | RuntimeInit.main |
| system\_server进程 | SystemServer.main() |
| app进程 | ActivityThread.main() |

注意其中app\_process进程是指通过/system/bin/app\_process来启动的进程，且后面跟的参数不带–zygote，即并非启动zygote进程。 比如常见的有通过adb shell方式来执行am,pm等命令，便是这种方式。

关于重要进程重启的过程，会触发哪些关联进程重启名单：

* zygote：触发media、netd以及子进程(包括system\_server进程)重启；
* system\_server: 触发zygote重启;
* surfaceflinger：触发zygote重启;
* servicemanager: 触发zygote、healthd、media、surfaceflinger、drm重启

所以，surfaceflinger,servicemanager,zygote自身以及system\_server进程被杀都会触发Zygote重启。

## REF

[Android系统启动-概述](http://gityuan.com/2016/02/01/android-booting/)

# AMS

在android系统中，应用程序是由Activity组成的，因此，应用程序的启动过程实际上就是应用程序中的默认Activity的启动过程，本文将详细分析应用程序框架层的源代码，了解Android应用程序的启动过程。在手机屏幕中点击应用程序图标开始——到应用MainActivity展示出来结束。

下面详细分析每一步是如何实现的。

## Luancher进程

### Step 1. Launcher.startActivitySafely

1. **void** startActivitySafely(Intent intent, Object tag) {
2. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
3. try {
4. startActivity(intent);
5. } catch (ActivityNotFoundException e) {
6. ......
7. } catch (SecurityException e) {
8. ......
9. }
10. }

MainActivity，这里是AndroidManifest.xml文件中配置的：

1. **<activity** android:name=".MainActivity"
2. android:label="@string/app\_name"**>**
3. **<intent-filter>**
4. **<action** android:name="android.intent.action.MAIN" **/>**
5. **<category** android:name="android.intent.category.LAUNCHER" **/>**
6. **</intent-filter>**
7. **</activity>**

因此，这里的intent包含的信息为：action = "android.intent.action.Main"，category="android.intent.category.LAUNCHER", cmp="shy.luo.activity/.MainActivity"，表示它要启动的Activity为shy.luo.activity.MainActivity。Intent.FLAG\_ACTIVITY\_NEW\_TASK表示要在一个新的Task中启动这个Activity，注意，Task是Android系统中的概念，它不同于进程Process的概念。简单地说，一个Task是一系列Activity的集合，这个集合是以堆栈的形式来组织的，遵循后进先出的原则。一个task可以包含多个process。





在Android系统中，应用程序是由Launcher启动起来的，其实，Launcher本身也是一个应用程序，其它的应用程序安装后，就会Launcher的界面上出现一个相应的图标，点击这个图标时，Launcher就会对应的应用程序启动起来。

### Step 2. Activity.startActivity

Launcher的父类Activity实现了startActivity函数，因此，这里就调用了Activity.startActivity函数。 这个函数实现很简单，它调用startActivityForResult来进一步处理，第二个参数传入-1表示不需要这个Actvity结束后的返回结果。

1. @Override
2. **public** **void** startActivity(Intent intent) {
3. startActivityForResult(intent, -1);
4. }

### Step 3. Activity.startActivityForResult

1. **public** **void** startActivityForResult(Intent **intent**, **int** requestCode) {
2. **if** (mParent == **null**) {
3. Instrumentation.ActivityResult ar =
4. mInstrumentation.execStartActivity(
5. **this**, mMainThread.getApplicationThread(), mToken, **this**,
6. **intent**, requestCode);
7. ......
8. } **else** {
9. ......
10. }

这里的mInstrumentation是Activity类的成员变量，它的类型是Intrumentation，定义在frameworks/base/core/java/android/app/Instrumentation.java文件中，它用来监控应用程序和系统的交互。

这里的mMainThread也是Activity类的成员变量，它的类型是ActivityThread，它代表的是应用程序的主线程，这里通过mMainThread.getApplicationThread获得它里面的ApplicationThread成员变量，它是一个Binder对象，后面我们会看到，ActivityManagerService会使用它来和ActivityThread来进行进程间通信。这里我们需注意的是，**这里的mMainThread代表的是Launcher应用程序运行的进程。**

这里的mToken也是Activity类的成员变量，它是一个Binder对象的远程接口。

### Step 4. Instrumentation.execStartActivity

在frameworks/base/core/java/android/app/Instrumentation.java文件中：

1. **public** **class** Instrumentation {
2. **public** ActivityResult execStartActivity(
3. Context who, IBinder contextThread, IBinder token, Activity target,
4. Intent intent, **int** requestCode) {
5. IApplicationThread whoThread = (IApplicationThread) contextThread;
6. **if** (mActivityMonitors != **null**) {
7. ......
8. }
9. **try** {
10. **int** result = ActivityManagerNative.getDefault()
11. .**startActivity**(whoThread, intent,
12. intent.**resolveTypeIfNeeded**(who.getContentResolver()),
13. **null**, 0, token, target != **null** ? target.mEmbeddedID : **null**,
14. requestCode, **false**, **false**);
15. ......
16. } **catch** (RemoteException e) {
17. }
18. **return** **null**;
19. }
21. }

这里的ActivityManagerNative.getDefault返回**ActivityManagerService**的远程接口，即**ActivityManagerProxy**接口，具体可以参考Android系统在新进程中启动自定义服务过程（startService）的原理分析一文。

这里的intent.resolveTypeIfNeeded返回这个intent的MIME类型，在这个例子中，没有AndroidManifest.xml设置MainActivity的MIME类型，因此，这里返回null。

这里的target不为null，但是target.mEmbddedID为null，我们不用关注。

### Step 5. ActivityManagerProxy.startActivity

定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **int** startActivity(IApplicationThread caller, Intent intent,
4. String resolvedType, Uri[] grantedUriPermissions, **int** grantedMode,
5. IBinder resultTo, String resultWho,
6. **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) **throws** RemoteException {
8. Parcel data = Parcel.obtain();
9. Parcel reply = Parcel.obtain();
10. data.writeInterfaceToken(IActivityManager.descriptor);
11. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
12. intent.writeToParcel(data, 0);
13. data.writeString(resolvedType);  **//null**
14. data.writeTypedArray(grantedUriPermissions, 0);   **//null**
15. data.writeInt(grantedMode);
16. data.writeStrongBinder(resultTo);
17. data.writeString(resultWho);
18. data.writeInt(requestCode);  //-1
19. data.writeInt(onlyIfNeeded ? 1 : 0);  //false
20. data.writeInt(debug ? 1 : 0);  //false
21. **mRemote**.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);
22. reply.readException();
23. **int** result = reply.readInt();
24. reply.recycle();
25. data.recycle();
26. **return** result;
27. }
28. }

这里的参数比较多，我们先整理一下。从上面的调用可以知道，这里的参数resolvedType、grantedUriPermissions和resultWho均为null；参数caller为ApplicationThread类型的Binder实体；参数resultTo为一个Binder实体的远程接口，我们先不关注它；参数grantedMode为0，我们也先不关注它；参数requestCode为-1；参数onlyIfNeeded和debug均空false。**mRemote开始向AMS进程间通信，发送**START\_ACTIVITY\_TRANSACTION消息。

## AMS进程

上述都在launer中处理的，接下是AMS响应START\_ACTIVITY\_TRANSACTIO的进程间通信

### Step 6. ActivityManagerService.startActivity

上一步Step 5通过Binder驱动程序就进入到ActivityManagerService的startActivity函数来了，它定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **int** startActivity(IApplicationThread caller,
4. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
5. **int** grantedMode, IBinder resultTo,
6. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) {
8. **return** **mMainStack**.startActivityMayWait(caller, intent, resolvedType,
9. grantedUriPermissions, grantedMode, resultTo, resultWho,
10. requestCode, onlyIfNeeded, debug, **null**, **null**);
11. }
12. }

这里只是简单地将操作转发给成员变量mMainStack的startActivityMayWait函数，这里的mMainStack的类型为ActivityStack。

### ActivityStack.startActivityMayWait

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityMayWait(IApplicationThread caller,
3. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
4. **int** grantedMode, IBinder resultTo,
5. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
6. **boolean** debug, WaitResult outResult, Configuration config) {
7. **boolean** componentSpecified = intent.getComponent() != **null**;
8. // Don't modify the client's object!
9. intent = **new** Intent(intent);
10. // Collect information about the target of the Intent.
11. ActivityInfo aInfo;
12. **try** {
13. ResolveInfo rInfo =
14. AppGlobals.getPackageManager().resolveIntent(
15. intent, resolvedType,
16. PackageManager.MATCH\_DEFAULT\_ONLY
17. | ActivityManagerService.STOCK\_PM\_FLAGS);
18. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
19. } **catch** (RemoteException e) {
20. ......
21. }
23. **if** (aInfo != **null**) {
24. // Store the found target back into the intent, because now that
25. // we have it we never want to do this again.  For example, if the
26. // user navigates back to this point in the history, we should
27. // always restart the exact same activity.
28. intent.setComponent(**new** ComponentName(
29. aInfo.applicationInfo.packageName, aInfo.name));
30. ......
31. }
33. **synchronized** (mService) {
34. **int** callingPid;
35. **int** callingUid;
36. **if** (caller == **null**) {
37. ......
38. } **else** {
39. callingPid = callingUid = -1;
40. }
42. mConfigWillChange = config != **null**
43. && mService.mConfiguration.diff(config) != 0;
44. **int** res = **startActivityLocked**(caller, intent, resolvedType,
45. grantedUriPermissions, grantedMode, aInfo,
46. resultTo, resultWho, requestCode, callingPid, callingUid,
47. onlyIfNeeded, componentSpecified);
48. **return** res;
49. }

注意，从Step 6传下来的参数outResult和config均为null，此外，表达式(aInfo.applicationInfo.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0为false，因此，这里忽略了无关代码。 下面语句对参数intent的内容进行解析，得到MainActivity的相关信息，保存在aInfo变量中：

1. ActivityInfo aInfo;
2. **try** {
3. ResolveInfo rInfo =   AppGlobals.getPackageManager().resolveIntent(  intent, resolvedType,
4. PackageManager.MATCH\_DEFAULT\_ONLY
5. | ActivityManagerService.STOCK\_PM\_FLAGS);
6. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
7. } **catch** (RemoteException e) {
8. ......
9. }

解析之后，得到的aInfo.applicationInfo.packageName的值为"k.demo.asys"，aInfo.name的值为"k.demo.asys.activity.MainActivity"，这是在这个实例的配置文件AndroidManifest.xml里面配置的。此外，函数开始的地方调用intent.getComponent()函数的返回值不为null，因此，这里的componentSpecified变量为true。

接下去就调用startActivityLocked进一步处理了。

### Step 8. ActivityStack.startActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityLocked(IApplicationThread caller,
3. Intent intent, String resolvedType,
4. Uri[] grantedUriPermissions,
5. **int** grantedMode, ActivityInfo aInfo, IBinder resultTo,
6. String resultWho, **int** requestCode,
7. **int** callingPid, **int** callingUid, **boolean** onlyIfNeeded,
8. **boolean** componentSpecified) {
9. **int** err = START\_SUCCESS;
11. ProcessRecord callerApp = **null**;
12. **if** (caller != **null**) {
13. **callerApp** = mService.getRecordForAppLocked(caller);
14. **if** (callerApp != **null**) {
15. callingPid = callerApp.pid;
16. callingUid = callerApp.info.uid;
17. }
18. }
19. ActivityRecord sourceRecord = **null**;
20. ActivityRecord resultRecord = **null**;
21. **if** (resultTo != **null**) {
22. **int** index = indexOfTokenLocked(resultTo);
23. **if** (index >= 0) {
24. sourceRecord = (ActivityRecord)mHistory.get(index);
25. **if** (requestCode >= 0 && !sourceRecord.finishing) {
26. ......
27. }
28. }
29. }
31. **int** launchFlags = intent.getFlags();
33. **if** ((launchFlags&Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0
34. && sourceRecord != **null**) {
35. ......
36. }
38. **if** (err == START\_SUCCESS && intent.getComponent() == **null**) {
39. ......
40. }
42. **if** (err == START\_SUCCESS && aInfo == **null**) {
43. ......
44. }
46. **if** (err != START\_SUCCESS) {
47. ......
48. }
49. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
50. intent, resolvedType, aInfo, mService.mConfiguration,
51. resultRecord, resultWho, requestCode, componentSpecified);
52. **return** **startActivityUncheckedLocked**(r, sourceRecord,
53. grantedUriPermissions, grantedMode, onlyIfNeeded, **true**);
54. }
55. }

从传进来的参数caller得到调用者的进程信息，并保存在callerApp变量中，这里就是Launcher应用程序的进程信息了。前面说过，参数resultTo是Launcher这个Activity里面的一个Binder对象，通过它可以获得Launcher这个Activity的相关信息，保存在sourceRecord变量中。再接下来，创建即将要启动的Activity的相关信息，并保存在r变量中：

1. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
2. intent, resolvedType, aInfo, mService.mConfiguration,
3. resultRecord, resultWho, requestCode, componentSpecified);

接着调用startActivityUncheckedLocked函数进行下一步操作。

### Step 9. ActivityStack.startActivityUncheckedLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityUncheckedLocked(ActivityRecord r,
3. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
4. **int** grantedMode, **boolean** onlyIfNeeded, **boolean** doResume) {
5. **final** Intent intent = r.intent;
6. **final** **int** callingUid = r.launchedFromUid;
7. **int** launchFlags = intent.getFlags();
8. // We'll invoke onUserLeaving before onPause only if the launching
9. // activity did not explicitly state that this is an automated launch.
10. mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;
11. ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)
12. != 0 ? r : **null**;
13. // If the onlyIfNeeded flag is set, then we can do this if the activity
14. // being launched is the same as the one making the call...  or, as
15. // a special case, if we do not know the caller then we count the
16. // current top activity as the caller.
17. **if** (onlyIfNeeded) {
18. ......
19. }
21. **if** (sourceRecord == **null**) {
22. ......
23. } **else** **if** (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
24. ......
25. } **else** **if** (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
26. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {
27. ......
28. }
30. **if** (r.resultTo != **null** && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
31. ......
32. }
34. **boolean** addingToTask = **false**;
35. **if** (((**launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK)** != 0 &&
36. (launchFlags&Intent.**FLAG\_ACTIVITY\_MULTIPLE\_TASK**) == 0)
37. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
38. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
39. // If bring to front is requested, and no result is requested, and
40. // we can find a task that was started with this same
41. // component, then instead of launching bring that one to the front.
42. **if** (r.resultTo == **null**) {
43. // See if there is a task to bring to the front.  If this is
44. // a SINGLE\_INSTANCE activity, there can be one and only one
45. // instance of it in the history, and it is always in its own
46. // unique task, so we do a special search.
47. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
48. ? findTaskLocked(intent, r.info)
49. : findActivityLocked(intent, r.info);
50. **if** (taskTop != **null**) {
51. ......
52. }
53. }
54. }
55. **if** (r.packageName != **null**) {
56. // If the activity being launched is the same as the one currently
57. // at the top, then we need to check if it should only be launched
58. // once.
59. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
60. **if** (top != **null** && r.resultTo == **null**) {
61. **if** (top.realActivity.equals(r.realActivity)) {
62. ......
63. }
64. }
66. }
68. **boolean** newTask = **false**;
70. // Should this be considered a new task?
71. **if** (r.resultTo == **null** && !addingToTask
72. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
73. // todo: should do better management of integers.
74. mService.mCurTask++;
75. **if** (mService.mCurTask <= 0) {
76. mService.mCurTask = 1;
77. }
78. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
79. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
80. ......
81. newTask = **true**;
82. **if** (mMainStack) {
83. mService.addRecentTaskLocked(r.task);
84. }
85. }
86. **startActivityLocked**(r, newTask, doResume);
87. **return** START\_SUCCESS;
88. }
89. }

函数首先获得intent的标志值，保存在launchFlags变量中。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION没有置位，因此 ，成员变量mUserLeaving的值为true。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP也没有置位，因此，变量notTop的值为null。

由于在这个例子的AndroidManifest.xml文件中，MainActivity没有配置launchMode属值，因此，这里的r.launchMode为默认值0，表示以标准（Standard，或者称为ActivityInfo.LAUNCH\_MULTIPLE）的方式来启动这个Activity。Activity的启动方式有四种，其余三种分别是ActivityInfo.LAUNCH\_SINGLE\_INSTANCE、ActivityInfo.LAUNCH\_SINGLE\_TASK和ActivityInfo.LAUNCH\_SINGLE\_TOP，具体可以参考官方网站<http://developer.android.com/reference/android/content/pm/ActivityInfo.html>。

传进来的参数r.resultTo为null，表示Launcher不需要等这个即将要启动的MainActivity的执行结果。

由于这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NEW\_TASK被置位，而且Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK没有置位，因此，下面的if语句会被执行：

1. **if** (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
2. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
3. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
4. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
5. // If bring to front is requested, and no result is requested, and
6. // we can find a task that was started with this same
7. // component, then instead of launching bring that one to the front.
8. **if** (r.resultTo == **null**) {
9. // See if there is a task to bring to the front.  If this is
10. // a SINGLE\_INSTANCE activity, there can be one and only one
11. // instance of it in the history, and it is always in its own
12. // unique task, so we do a special search.
13. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
14. ? findTaskLocked(intent, r.info)
15. : findActivityLocked(intent, r.info);
16. **if** (taskTop != **null**) {
17. ......
18. }
19. }
20. }

这段代码的逻辑是查看一下，当前有没有Task可以用来执行这个Activity。由于r.launchMode的值不为ActivityInfo.LAUNCH\_SINGLE\_INSTANCE，因此，它通过findTaskLocked函数来查找存不存这样的Task，这里返回的结果是null，即taskTop为null，因此，需要创建一个新的Task来启动这个Activity。

接着往下看：

1. **if** (r.packageName != **null**) {
2. // If the activity being launched is the same as the one currently
3. // at the top, then we need to check if it should only be launched
4. // once.
5. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
6. **if** (top != **null** && r.resultTo == **null**) {
7. **if** (top.realActivity.equals(r.realActivity)) {
8. ......
9. }
10. }
12. }

这段代码的逻辑是看一下，当前在堆栈顶端的Activity是否就是即将要启动的Activity，有些情况下，如果即将要启动的Activity就在堆栈的顶端，那么，就不会重新启动这个Activity的别一个实例了，具体可以参考官方网站http://developer.android.com/reference/android/content/pm/ActivityInfo.html。现在处理堆栈顶端的Activity是Launcher，与我们即将要启动的MainActivity不是同一个Activity，因此，这里不用进一步处理上述介绍的情况。

执行到这里，我们知道，要在一个新的Task里面来启动这个Activity了，于是新创建一个Task：

1. **if** (r.resultTo == **null** && !addingToTask
2. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
3. // todo: should do better management of integers.
4. mService.mCurTask++;
5. **if** (mService.mCurTask <= 0) {
6. mService.mCurTask = 1;
7. }
8. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
9. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
10. ......
11. newTask = **true**;
12. **if** (mMainStack) {
13. mService.addRecentTaskLocked(r.task);
14. }
16. }

新建的Task保存在r.task域中，同时，添加到mService中去，这里的mService就是ActivityManagerService了。

最后就进入startActivityLocked(r, newTask, doResume)进一步处理了。这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **private** **final** **void** **startActivityLocked**(ActivityRecord r, **boolean** newTask,
3. **boolean** doResume) {
4. **final** **int** NH = mHistory.size();
5. **int** addPos = -1;
6. **if** (!newTask) {
7. ......
8. }
9. // Place a new activity at top of stack, so it is next to interact
10. // with the user.
11. **if** (addPos < 0) {
12. addPos = NH;
13. }
14. // If we are not placing the new activity frontmost, we do not want
15. // to deliver the onUserLeaving callback to the actual frontmost
16. // activity
17. **if** (addPos < NH) {
18. ......
19. }
21. // Slot the activity into the history stack and proceed
22. mHistory.add(addPos, r);
23. r.inHistory = **true**;
24. r.frontOfTask = newTask;
25. r.task.numActivities++;
26. **if** (NH > 0) {
27. // We want to show the starting preview window if we are
28. // switching to a new task, or the next activity's process is
29. // not currently running.
30. ......
31. } **else** {
32. // If this is the first activity, don't do any fancy animations,
33. // because there is nothing for it to animate on top of.
34. ......
35. }
36. **if** (doResume) {
37. resumeTopActivityLocked(**null**);
38. }
39. }
40. }

这里的NH表示当前系统中历史任务的个数，这里肯定是大于0，因为Launcher已经跑起来了。当NH>0时，并且现在要切换新任务时，要做一些任务切的界面操作，这段代码我们就不看了，这里不会影响到下面启Activity的过程，有兴趣的读取可以自己研究一下。

这里传进来的参数doResume为true，于是调用resumeTopActivityLocked进一步操作。

### Step 10. ActivityStack.resumeTopActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. /\*\*
3. \* Ensure that the top activity in the stack is resumed.
4. \*
5. \* @param prev The previously resumed activity, for when in the process
6. \* of pausing; can be null to call from elsewhere.
7. \*
8. \* @return Returns true if something is being resumed, or false if
9. \* nothing happened.
10. \*/
11. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
12. // Find the first activity that is not finishing.
13. ActivityRecord next = topRunningActivityLocked(**null**);
15. // Remember how we'll process this pause/resume situation, and ensure
16. // that the state is reset however we wind up proceeding.
17. **final** **boolean** userLeaving = mUserLeaving;
18. mUserLeaving = **false**;
20. **if** (next == **null**) {
21. ......
22. }
24. next.delayedResume = **false**;
26. // If the top activity is the resumed one, nothing to do.
27. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
28. ......
29. }
31. // If we are sleeping, and there is no resumed activity, and the top
32. // activity is paused, well that is the state we want.
33. **if** ((mService.mSleeping || mService.mShuttingDown)
34. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
35. ......
36. }
38. ......
40. // If we are currently pausing an activity, then don't do anything
41. // until that is done.
42. **if** (mPausingActivity != **null**) {
43. ......
44. }
46. ......
48. // We need to start pausing the current activity so the top one
49. // can be resumed...
50. **if** (mResumedActivity != **null**) {
51. ......
52. startPausingLocked(userLeaving, **false**);
53. **return** **true**;
54. }
56. ......
57. }
59. ......
61. }

函数先通过调用topRunningActivityLocked函数获得堆栈顶端的Activity，这里就是**MainActivity**了，这是在上面的Step 9设置好的，保存在next变量中。

接下来把mUserLeaving的保存在本地变量userLeaving中，然后重新设置为false，在上面的Step 9中，mUserLeaving的值为true，因此，这里的userLeaving为true。

这里的mResumedActivity为Launcher，因为Launcher是当前正被执行的Activity。

当我们处理休眠状态时，mLastPausedActivity保存堆栈顶端的Activity，因为当前不是休眠状态，所以mLastPausedActivity为null。

有了这些信息之后，下面的语句就容易理解了：

1. // If the top activity is the resumed one, nothing to do.
2. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
3. ......
4. }
6. // If we are sleeping, and there is no resumed activity, and the top
7. // activity is paused, well that is the state we want.
8. **if** ((mService.mSleeping || mService.mShuttingDown)
9. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
10. ......
11. }

就可以了；否则再看一下系统当前是否休眠状态，如果是的话，再看看要启动的Activity是否就是当前处于堆栈顶端的Activity，如果是的话，也是什么都不用做。

上面两个条件都不满足，因此，在继续往下执行之前，首先要把当处于Resumed状态的Activity推入Paused状态，然后才可以启动新的Activity。但是在将当前这个Resumed状态的Activity推入Paused状态之前，首先要看一下当前是否有Activity正在进入Pausing状态，如果有的话，当前这个Resumed状态的Activity就要稍后才能进入Paused状态了，这样就保证了所有需要进入Paused状态的Activity串行处理。

这里没有处于Pausing状态的Activity，即mPausingActivity为null，而且mResumedActivity也不为null，于是就调用startPausingLocked函数把Launcher推入Paused状态去了。

### Step 11. ActivityStack.startPausingLocked

1. **public** **class** ActivityStack {
3. ......
5. **private** **final** **void** startPausingLocked(**boolean** userLeaving, **boolean** uiSleeping) {
6. **if** (mPausingActivity != **null**) {
7. ......
8. }
9. ActivityRecord prev = mResumedActivity;
10. **if** (prev == **null**) {
11. ......
12. }
13. ......
14. mResumedActivity = **null**;
15. mPausingActivity = prev;
16. mLastPausedActivity = prev;
17. prev.state = ActivityState.PAUSING;
18. ......
20. **if** (prev.app != **null** && prev.app.thread != **null**) {
21. ......
22. **try** {
23. ......
24. prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
25. prev.configChangeFlags);
26. ......
27. } **catch** (Exception e) {
28. ......
29. }
30. } **else** {
31. ......
32. }
34. ......
36. }
38. ......
40. }

函数首先把mResumedActivity保存在本地变量prev中。在上一步Step 10中，说到mResumedActivity就是Launcher，因此，这里把Launcher进程中的ApplicationThread对象取出来，通过它来通知Launcher这个Activity它要进入Paused状态了。当然，这里的prev.app.thread是一个ApplicationThread对象的远程接口，通过调用这个远程接口的schedulePauseActivity来通知Launcher进入Paused状态。

参数prev.finishing表示prev所代表的Activity是否正在等待结束的Activity列表中，由于Laucher这个Activity还没结束，所以这里为false；参数prev.configChangeFlags表示哪些config发生了变化，这里我们不关心它的值。

### Step 12. ApplicationThreadProxy.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
3. ......
5. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
6. **boolean** userLeaving, **int** configChanges) **throws** RemoteException {
7. Parcel data = Parcel.obtain();
8. data.writeInterfaceToken(IApplicationThread.descriptor);
9. data.writeStrongBinder(token);
10. data.writeInt(finished ? 1 : 0);
11. data.writeInt(userLeaving ? 1 :0);
12. data.writeInt(configChanges);
13. mRemote.transact(SCHEDULE\_PAUSE\_ACTIVITY\_TRANSACTION, data, **null**,
14. IBinder.FLAG\_ONEWAY);
15. data.recycle();
16. }
18. ......
20. }

这个函数通过Binder进程间通信机制进入到ApplicationThread.schedulePauseActivity函数中。

## L进程

### Step 13. ApplicationThread.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中，它是ActivityThread的内部类：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
3. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
4. **boolean** userLeaving, **int** configChanges) {
5. queueOrSendMessage(
6. finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,
7. token,
8. (userLeaving ? 1 : 0),
9. configChanges);
10. }
11. }
12. }

这里调用的函数queueOrSendMessage是ActivityThread类的成员函数。

上面说到，这里的finished值为false，因此，queueOrSendMessage的第一个参数值为H.PAUSE\_ACTIVITY，表示要暂停token所代表的Activity，即Launcher。

### Step 14. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1) {
3. queueOrSendMessage(what, obj, arg1, 0);
4. }
5. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
6. **synchronized** (**this**) {
7. ......
8. Message msg = Message.obtain();
9. msg.what = what;
10. msg.obj = obj;
11. msg.arg1 = arg1;
12. msg.arg2 = arg2;
13. mH.sendMessage(msg);
14. }
15. }

这里首先将相关信息组装成一个msg，然后通过mH成员变量发送出去，mH的类型是H，继承于Handler类，是ActivityThread的内部类，因此，这个消息最后由H.handleMessage来处理。

### Step 15. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** H **extends** Handler {
3. **public** **void** handleMessage(Message msg) {
4. **switch** (msg.what) {
5. **case** PAUSE\_ACTIVITY:
6. handlePauseActivity((IBinder)msg.obj, **false**, msg.arg1 != 0, msg.arg2);
7. maybeSnapshot();
8. **break**;
9. }

这里调用ActivityThread.handlePauseActivity进一步操作，msg.obj是一个ActivityRecord对象的引用，它代表的是Launcher这个Activity。

### Step 16. ActivityThread.handlePauseActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handlePauseActivity(IBinder token, **boolean** finished,
3. **boolean** userLeaving, **int** configChanges) {
4. ActivityClientRecord r = mActivities.get(token);
5. **if** (r != **null**) {
6. //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
7. **if** (userLeaving) {
8. performUserLeavingActivity(r);
9. }
10. r.activity.mConfigChangeFlags |= configChanges;
11. Bundle state = **performPauseActivity**(token, finished, **true**);
12. // Make sure any pending writes are now committed.
13. QueuedWork.waitToFinish();
14. // Tell the activity manager we have paused.
15. **try** {
16. ActivityManagerNative.getDefault().**activityPaused**(token, state);
17. } **catch** (RemoteException ex) {
18. }
19. }
20. }
21. }

函数首先将Binder引用token转换成ActivityRecord的远程接口ActivityClientRecord，然后做了三个事情：1. 如果userLeaving为true，则通过调用performUserLeavingActivity函数来调用Activity.onUserLeaveHint通知Activity，用户要离开它了；2. 调用**performPauseActivity函数来调用Activity.onPause函数**，我们知道，在Activity的生命周期中，当它要让位于其它的Activity时，系统就会调用它的onPause函数；3. 它通知ActivityManagerService，这个Activity已经进入Paused状态了，ActivityManagerService现在可以完成未竟的事情，即启动MainActivity了。

### Step 17. ActivityManagerProxy.activityPaused

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. ......
5. **public** **void** activityPaused(IBinder token, Bundle state) **throws** RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(token);
11. data.writeBundle(state);
12. mRemote.transact(ACTIVITY\_PAUSED\_TRANSACTION, data, reply, 0);
13. reply.readException();
14. data.recycle();
15. reply.recycle();
16. }
18. ......
20. }

这里通过Binder进程间通信机制就进入到ActivityManagerService.activityPaused函数中去了。

## AMS

### Step 18. ActivityManagerService.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **void** activityPaused(IBinder token, Bundle icicle) {
4. **final** **long** origId = Binder.clearCallingIdentity();
5. mMainStack.activityPaused(token, icicle, **false**);
6. }
7. }

这里，又再次进入到ActivityStack类中，执行activityPaused函数。

### Step 19. ActivityStack.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **void** activityPaused(IBinder token, Bundle icicle, **boolean** timeout) {
3. ActivityRecord r = **null**;
4. **synchronized** (mService) {
5. **int** index = indexOfTokenLocked(token);
6. **if** (index >= 0) {
7. r = (ActivityRecord)mHistory.get(index);
8. **if** (!timeout) {
9. r.icicle = icicle;
10. r.haveState = **true**;
11. }
12. mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);
13. **if** (mPausingActivity == r) {
14. r.state = ActivityState.PAUSED;
15. completePauseLocked();
16. } **else** {
17. ......
18. }
19. }
20. }
21. }
22. }

这里通过参数token在mHistory列表中得到ActivityRecord，从上面我们知道，这个ActivityRecord代表的是Launcher这个Activity，而我们在Step 11中，把Launcher这个Activity的信息保存在mPausingActivity中，因此，这里mPausingActivity等于r，于是，执行completePauseLocked操作。

### Step 20. ActivityStack.completePauseLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** completePauseLocked() {
3. ActivityRecord prev = mPausingActivity;
4. **if** (prev != **null**) {
5. mPausingActivity = **null**;
6. }
7. **if** (!mService.mSleeping && !mService.mShuttingDown) {
8. resumeTopActivityLocked(prev);
9. } **else** {
10. }
11. }
12. }

函数首先把mPausingActivity变量清空，因为现在不需要它了，然后调用resumeTopActivityLokced进一步操作，它传入的参数即为代表Launcher这个Activity的ActivityRecord。

### Step 21. ActivityStack.resumeTopActivityLokced

1. **public** **class** ActivityStack {
2. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
3. // Find the first activity that is not finishing.
4. ActivityRecord next = topRunningActivityLocked(**null**);
6. // Remember how we'll process this pause/resume situation, and ensure
7. // that the state is reset however we wind up proceeding.
8. **final** **boolean** userLeaving = mUserLeaving;
9. mUserLeaving = **false**;
10. next.delayedResume = **false**;
12. // If the top activity is the resumed one, nothing to do.
13. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
14. ......
15. **return** **false**;
16. }
18. // If we are sleeping, and there is no resumed activity, and the top
19. // activity is paused, well that is the state we want.
20. **if** ((mService.mSleeping || mService.mShuttingDown)
21. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
22. ......
23. **return** **false**;
24. }
26. .......

29. // We need to start pausing the current activity so the top one
30. // can be resumed...
31. **if** (mResumedActivity != **null**) {
32. ......
33. **return** **true**;
34. }
35. **if** (next.app != **null** && next.app.thread != **null**) {
36. } **else** {
37. startSpecificActivityLocked(next, **true**, **true**);
38. }
40. **return** **true**;
41. }
42. }

通过上面的Step 9，我们知道，**当前在堆栈顶端的Activity为我们即将要启动的MainActivity**，这里通过调用topRunningActivityLocked将它取回来，保存在next变量中。之前最后一个Resumed状态的Activity，即Launcher，到了这里已经处于Paused状态了，因此，mResumedActivity为null。最后一个处于Paused状态的Activity为Launcher，因此，这里的mLastPausedActivity就为Launcher。前面我们为MainActivity创建了ActivityRecord后，它的app域一直保持为null。有了这些信息后，上面这段代码就容易理解了，它最终调用startSpecificActivityLocked进行下一步操作。

### Step 22. ActivityStack.startSpecificActivityLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** startSpecificActivityLocked(ActivityRecord r,
3. **boolean** andResume, **boolean** checkConfig) {
4. // Is this activity's application already running?
5. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
6. r.info.applicationInfo.uid);
7. **if** (app != **null** && app.thread != **null**) {
8. **try** {
9. realStartActivityLocked(r, app, andResume, checkConfig);
10. **return**;
11. } **catch** (RemoteException e) {
12. ......
13. }
14. }
16. mService.startProcessLocked(r.processName, r.info.applicationInfo, **true**, 0,
17. "activity", r.intent.getComponent(), **false**);
18. }
19. }

注意，这里由于是第一次启动应用程序的Activity，所以下面语句：

ProcessRecord app = mService.getProcessRecordLocked(r.processName, r.info.applicationInfo.uid); 取回来的app为null。

在Activity应用程序中的AndroidManifest.xml配置文件中，我们没有指定Application标签的process属性，系统就会默认使用package的名称，这里就是"k.demo.system"了。每一个应用程序都有自己的uid，因此，这里uid + process的组合就可以为每一个应用程序创建一个ProcessRecord。当然，我们可以配置两个应用程序具有相同的uid和package，或者在AndroidManifest.xml配置文件的application标签或者activity标签中显式指定相同的process属性值，这样，不同的应用程序也可以在同一个进程中启动。

函数最终执行ActivityManagerService.startProcessLocked函数进行下一步操作。

### Step 23. ActivityManagerService.startProcessLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **final** ProcessRecord startProcessLocked(String processName,
4. ApplicationInfo info, **boolean** knownToBeDead, **int** intentFlags,
5. String hostingType, ComponentName hostingName, **boolean** allowWhileBooting) {
6. ProcessRecord app = getProcessRecordLocked(processName, info.uid);
7. String hostingNameStr = hostingName != **null**
8. ? hostingName.flattenToShortString() : **null**;
9. **if** (app == **null**) {
10. app = **new** ProcessRecordLocked(**null**, info, processName);
11. mProcessNames.put(processName, info.uid, app);
12. } **else** {
13. // If this is a new package in the process, add the package to the list
14. app.addPackage(info.packageName);
15. }
16. startProcessLocked(app, hostingType, hostingNameStr);
17. **return** (app.pid != 0) ? app : **null**;
18. }
19. }

这里再次检查是否已经有以process + uid命名的进程存在，在我们这个情景中，返回值app为null，因此，后面会创建一个ProcessRecord，并存保存在成员变量mProcessNames中，最后，调用另一个startProcessLocked函数进一步操作：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **private** **final** **void** startProcessLocked(ProcessRecord app,
4. String hostingType, String hostingNameStr) {
5. **try** {
6. **int** uid = app.info.uid;
7. **int**[] gids = **null**;
8. **try** {
9. gids = mContext.getPackageManager().getPackageGids(
10. app.info.packageName);
11. } **catch** (PackageManager.NameNotFoundException e) {
12. }
13. **int** debugFlags = 0;
14. **int** pid = Process.start("android.app.ActivityThread",
15. mSimpleProcessManagement ? app.processName : **null**, uid, uid,
16. gids, debugFlags, **null**);
17. } **catch** (RuntimeException e) {
18. }
19. }
20. }

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

### Step 24. ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. mgr.attachApplication(mAppThread);
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了ActivityManagerService的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的。

### Step 25. ActivityManagerProxy.attachApplication

这个函数定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **void** attachApplication(IApplicationThread app) **throws** RemoteException
4. {
5. Parcel data = Parcel.obtain();
6. Parcel reply = Parcel.obtain();
7. data.writeInterfaceToken(IActivityManager.descriptor);
8. data.writeStrongBinder(app.asBinder());
9. mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);
10. reply.readException();
11. data.recycle();
12. reply.recycle();
13. }
14. }

这里通过Binder驱动程序，最后进入ActivityManagerService的attachApplication函数中。

## AMS

### Step 26. ActivityManagerService.attachApplication

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **void** attachApplication(IApplicationThread thread) {
4. **synchronized** (**this**) {
5. **int** callingPid = Binder.getCallingPid();
6. **final** **long** origId = Binder.clearCallingIdentity();
7. attachApplicationLocked(thread, callingPid);
8. Binder.restoreCallingIdentity(origId);
9. }
10. }
11. }

这里将操作转发给attachApplicationLocked函数。

### Step 27. ActivityManagerService.attachApplicationLocked

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. **private** **final** **boolean** attachApplicationLocked(IApplicationThread thread,
7. **int** pid) {
8. // Find the application record that is being attached...  either via
9. // the pid if we are running in multiple processes, or just pull the
10. // next app record if we are emulating process with anonymous threads.
11. ProcessRecord app;
12. **if** (pid != MY\_PID && pid >= 0) {
13. **synchronized** (mPidsSelfLocked) {
14. app = mPidsSelfLocked.get(pid);
15. }
16. } **else** **if** (mStartingProcesses.size() > 0) {
17. ......
18. } **else** {
19. ......
20. }
22. **if** (app == **null**) {
23. ......
24. **return** **false**;
25. }
27. ......
29. String processName = app.processName;
30. **try** {
31. thread.asBinder().linkToDeath(**new** AppDeathRecipient(
32. app, pid, thread), 0);
33. } **catch** (RemoteException e) {
34. ......
35. **return** **false**;
36. }
38. ......
40. app.thread = thread;
41. app.curAdj = app.setAdj = -100;
42. app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;
43. app.setSchedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;
44. app.forcingToForeground = **null**;
45. app.foregroundServices = **false**;
46. app.debugging = **false**;
48. ......
50. **boolean** normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
52. ......
54. **boolean** badApp = **false**;
55. **boolean** didSomething = **false**;
57. // See if the top visible activity is waiting to run in this process...
58. ActivityRecord hr = mMainStack.topRunningActivityLocked(**null**);
59. **if** (hr != **null** && normalMode) {
60. **if** (hr.app == **null** && app.info.uid == hr.info.applicationInfo.uid
61. && processName.equals(hr.processName)) {
62. **try** {
63. **if** (mMainStack.realStartActivityLocked(hr, app, **true**, **true**)) {
64. didSomething = **true**;
65. }
66. } **catch** (Exception e) {
67. ......
68. }
69. } **else** {
70. ......
71. }
72. }
74. ......
76. **return** **true**;
77. }
79. ......
81. }

在前面的Step 23中，已经创建了一个ProcessRecord，这里首先通过pid将它取回来，放在app变量中，然后对app的其它成员进行初始化，最后调用mMainStack.realStartActivityLocked执行真正的Activity启动操作。这里要启动的Activity通过调用mMainStack.topRunningActivityLocked(null)从堆栈顶端取回来，这时候在堆栈顶端的Activity就是MainActivity了。

### Step 28. ActivityStack.realStartActivityLocked

1. **public** **class** ActivityStack {
3. ......
5. **final** **boolean** realStartActivityLocked(ActivityRecord r,
6. ProcessRecord app, **boolean** andResume, **boolean** checkConfig)
7. **throws** RemoteException {
9. ......
11. r.app = app;
13. ......
15. **int** idx = app.activities.indexOf(r);
16. **if** (idx < 0) {
17. app.activities.add(r);
18. }
20. ......
22. **try** {
23. ......
25. List<ResultInfo> results = **null**;
26. List<Intent> newIntents = **null**;
27. **if** (andResume) {
28. results = r.results;
29. newIntents = r.newIntents;
30. }
32. ......
34. app.thread.scheduleLaunchActivity(**new** Intent(r.intent), r,
35. System.identityHashCode(r),
36. r.info, r.icicle, results, newIntents, !andResume,
37. mService.isNextTransitionForward());
39. ......
41. } **catch** (RemoteException e) {
42. ......
43. }
45. ......
47. **return** **true**;
48. }
50. ......
52. }

这里最终通过app.thread进入到ApplicationThreadProxy的scheduleLaunchActivity函数中，注意，这里的第二个参数r，是一个ActivityRecord类型的Binder对象，用来作来这个Activity的token值。

### Step 29. ApplicationThreadProxy.scheduleLaunchActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
2. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
3. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
4. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward)
5. **throws** RemoteException {
6. Parcel data = Parcel.obtain();
7. data.writeInterfaceToken(IApplicationThread.descriptor);
8. intent.writeToParcel(data, 0);
9. data.writeStrongBinder(token);
10. data.writeInt(ident);
11. info.writeToParcel(data, 0);
12. data.writeBundle(state);
13. data.writeTypedList(pendingResults);
14. data.writeTypedList(pendingNewIntents);
15. data.writeInt(notResumed ? 1 : 0);
16. data.writeInt(isForward ? 1 : 0);
17. mRemote.transact(SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION, data, **null**,
18. IBinder.FLAG\_ONEWAY);
19. data.recycle();
20. }
21. }

这个函数最终通过Binder驱动程序进入到ApplicationThread的scheduleLaunchActivity函数中。

### Step 30. ApplicationThread.scheduleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // we use token to identify this activity without having to send the
10. // activity itself back to the activity manager. (matters more with ipc)
11. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
12. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
13. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward) {
14. ActivityClientRecord r = **new** ActivityClientRecord();
16. r.token = token;
17. r.ident = ident;
18. r.intent = intent;
19. r.activityInfo = info;
20. r.state = state;
22. r.pendingResults = pendingResults;
23. r.pendingIntents = pendingNewIntents;
25. r.startsNotResumed = notResumed;
26. r.isForward = isForward;
28. queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);
29. }
31. ......
33. }
35. ......
36. }

函数首先创建一个ActivityClientRecord实例，并且初始化它的成员变量，然后调用ActivityThread类的queueOrSendMessage函数进一步处理。

### Step 31. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // if the thread hasn't started yet, we don't have the handler, so just
10. // save the messages until we're ready.
11. **private** **final** **void** queueOrSendMessage(**int** what, Object obj) {
12. queueOrSendMessage(what, obj, 0, 0);
13. }
15. ......
17. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
18. **synchronized** (**this**) {
19. ......
20. Message msg = Message.obtain();
21. msg.what = what;
22. msg.obj = obj;
23. msg.arg1 = arg1;
24. msg.arg2 = arg2;
25. mH.sendMessage(msg);
26. }
27. }
29. ......
31. }
33. ......
34. }

函数把消息内容放在msg中，然后通过mH把消息分发出去，这里的成员变量mH我们在前面已经见过，消息分发出去后，最后会调用H类的handleMessage函数。

### Step 32. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. **case** LAUNCH\_ACTIVITY: {
13. ActivityClientRecord r = (ActivityClientRecord)msg.obj;
15. r.packageInfo = getPackageInfoNoCheck(
16. r.activityInfo.applicationInfo);
17. handleLaunchActivity(r, **null**);
18. } **break**;
19. ......
20. }
22. ......
24. }
26. ......
27. }

这里最后调用ActivityThread类的handleLaunchActivity函数进一步处理。

### Step 33. ActivityThread.handleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **void** handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
6. ......
8. Activity a = performLaunchActivity(r, customIntent);
10. **if** (a != **null**) {
11. r.createdConfig = **new** Configuration(mConfiguration);
12. Bundle oldState = r.state;
13. handleResumeActivity(r.token, **false**, r.isForward);
15. ......
16. } **else** {
17. ......
18. }
19. }
21. ......
22. }

这里首先调用performLaunchActivity函数来加载这个Activity类，即shy.luo.activity.MainActivity，然后调用它的onCreate函数，最后回到handleLaunchActivity函数时，再调用handleResumeActivity函数来使这个Activity进入Resumed状态，即会调用这个Activity的onResume函数，这是遵循Activity的生命周期的。

### Step 34. ActivityThread.performLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
7. ActivityInfo aInfo = r.activityInfo;
8. **if** (r.packageInfo == **null**) {
9. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
10. Context.CONTEXT\_INCLUDE\_CODE);
11. }
13. ComponentName component = r.intent.getComponent();
14. **if** (component == **null**) {
15. component = r.intent.resolveActivity(
16. mInitialApplication.getPackageManager());
17. r.intent.setComponent(component);
18. }
20. **if** (r.activityInfo.targetActivity != **null**) {
21. component = **new** ComponentName(r.activityInfo.packageName,
22. r.activityInfo.targetActivity);
23. }
25. Activity activity = **null**;
26. **try** {
27. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
28. activity = mInstrumentation.newActivity(
29. cl, component.getClassName(), r.intent);
30. r.intent.setExtrasClassLoader(cl);
31. **if** (r.state != **null**) {
32. r.state.setClassLoader(cl);
33. }
34. } **catch** (Exception e) {
35. ......
36. }
38. **try** {
39. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);
41. ......
43. **if** (activity != **null**) {
44. ContextImpl appContext = **new** ContextImpl();
45. appContext.init(r.packageInfo, r.token, **this**);
46. appContext.setOuterContext(activity);
47. CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
48. Configuration config = **new** Configuration(mConfiguration);
49. ......
50. activity.attach(appContext, **this**, getInstrumentation(), r.token,
51. r.ident, app, r.intent, r.activityInfo, title, r.parent,
52. r.embeddedID, r.lastNonConfigurationInstance,
53. r.lastNonConfigurationChildInstances, config);
55. **if** (customIntent != **null**) {
56. activity.mIntent = customIntent;
57. }
58. r.lastNonConfigurationInstance = **null**;
59. r.lastNonConfigurationChildInstances = **null**;
60. activity.mStartedActivity = **false**;
61. **int** theme = r.activityInfo.getThemeResource();
62. **if** (theme != 0) {
63. activity.setTheme(theme);
64. }
66. activity.mCalled = **false**;
67. mInstrumentation.callActivityOnCreate(activity, r.state);
68. ......
69. r.activity = activity;
70. r.stopped = **true**;
71. **if** (!r.activity.mFinished) {
72. activity.performStart();
73. r.stopped = **false**;
74. }
75. **if** (!r.activity.mFinished) {
76. **if** (r.state != **null**) {
77. mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
78. }
79. }
80. **if** (!r.activity.mFinished) {
81. activity.mCalled = **false**;
82. mInstrumentation.callActivityOnPostCreate(activity, r.state);
83. **if** (!activity.mCalled) {
84. **throw** **new** SuperNotCalledException(
85. "Activity " + r.intent.getComponent().toShortString() +
86. " did not call through to super.onPostCreate()");
87. }
88. }
89. }
90. r.paused = **true**;
92. mActivities.put(r.token, r);
94. } **catch** (SuperNotCalledException e) {
95. ......
97. } **catch** (Exception e) {
98. ......
99. }
101. **return** activity;
102. }
104. ......
105. }

函数前面是收集要启动的Activity的相关信息，主要package和component信息：

1. ActivityInfo aInfo = r.activityInfo;
2. **if** (r.packageInfo == **null**) {
3. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
4. Context.CONTEXT\_INCLUDE\_CODE);
5. }
7. ComponentName component = r.intent.getComponent();
8. **if** (component == **null**) {
9. component = r.intent.resolveActivity(
10. mInitialApplication.getPackageManager());
11. r.intent.setComponent(component);
12. }
14. **if** (r.activityInfo.targetActivity != **null**) {
15. component = **new** ComponentName(r.activityInfo.packageName,
16. r.

然后通过ClassLoader将shy.luo.activity.MainActivity类加载进来：

1. Activity activity = **null**;
2. **try** {
3. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
4. activity = mInstrumentation.newActivity(
5. cl, component.getClassName(), r.intent);
6. r.intent.setExtrasClassLoader(cl);
7. **if** (r.state != **null**) {
8. r.state.setClassLoader(cl);
9. }
10. } **catch** (Exception e) {
11. ......
12. }

接下来是创建Application对象，这是根据AndroidManifest.xml配置文件中的Application标签的信息来创建的：

1. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);

后面的代码主要创建Activity的上下文信息，并通过attach方法将这些上下文信息设置到MainActivity中去：

1. activity.attach(appContext, **this**, getInstrumentation(), r.token,
2. r.ident, app, r.intent, r.activityInfo, title, r.parent,
3. r.embeddedID, r.lastNonConfigurationInstance,
4. r.lastNonConfigurationChildInstances, config);

最后还要调用MainActivity的onCreate函数：

Instrumentation.callActivityOnCreate(activity, r.state);

这里不是直接调用MainActivity的onCreate函数，而是通过mInstrumentation的callActivityOnCreate函数来间接调用，前面我们说过，mInstrumentation在这里的作用是监控Activity与系统的交互操作，相当于是系统运行日志。

### Step 35. MainActivity.onCreate

1. **public** **class** MainActivity **extends** Activity  **implements** OnClickListener {
3. ......
5. @Override
6. **public** **void** onCreate(Bundle savedInstanceState) {
7. ......
9. Log.i(LOG\_TAG, "Main Activity Created.");
10. }
12. ......

## 小结

这样，MainActivity就启动起来了，整个应用程序也启动起来了。

       整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知ActivityManagerService，它要启动一个Activity；

       二. Step 12 - Step 16：ActivityManagerService通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给ActivityManagerService，以便以后ActivityManagerService能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：ActivityManagerService通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

       这里不少地方涉及到了Binder进程间通信机制，相关资料请参考[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)一文。

       这样，应用程序的启动过程就介绍完了，它实质上是启动应用程序的默认Activity，在下一篇文章中，我们将介绍在应用程序内部启动另一个Activity的过程，即新的Activity与启动它的Activity将会在同一个进程（Process）和任务（Task）运行，敬请关注。

## 参考

[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)

[**Android深入四大组件（一）应用程序启动过程**](http://blog.csdn.net/itachi85/article/details/69388942)

<http://blog.csdn.net/itachi85/article/details/69388942>

# 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

# apk安装过程/PMS

## 概述

众所周知，Android应用最终是打包成.apk格式（其实就是一个压缩包），然后安装至手机并运行的。**APK即Android Pac**k**age的缩写。**

Android系统在启动的过程中，会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，完成应用程序的安装过程。

**应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了**。

### 四种安装方式

一般而言，Android应用安装有如下四种方式：

* 系统应用安装：开机时加载指定系统路径下的APK，没有安装界面；
* 原生maket安装：通过各种market应用完成，没有安装界面；
* ADB工具安装：即通过Android的SDK开发tools里面的adb.exe程序安装，没有安装界面；
* 第三方应用安装：比如第三方market、文件浏览器（通过SD卡里的APK文件安装(比如双击APK文件触发)）、app自升级。有安装界面，系统默认已经安装了一个安装卸载应用的程序，即由packageinstaller.apk应用处理安装及卸载过程的界面。

安装过程：复制APK安装包到data/app目录下，解压并扫描安装包，把dex文件(Dalvik字节码)保存到**dalvik-cache**目录，并data/data目录下创建对应的应用数据目录。

卸载过程：删除安装过程中在上述三个目录下创建的文件及目录。

### 应用安装涉及到的目录

/system/app ：系统自带的应用程序，获得adb root权限才能删除

/data/app ：用户程序安装的目录。安装时把apk文件复制到此目录

/data/data ：存放应用程序的数据

/data/dalvik-cache：将apk中的dex文件安装到dalvik-cache目录下(dex文件是dalvik虚拟机的可执行文件,当然，ART–Android Runtime的可执行文件格式为oat，启用ART时，系统会执行dex文件转换至oat文件)

/data/system/packages.xml：通过它可以看到系统安装的所有软件包，以及软件包的信息.类似于Windows**的注册表**，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。

记录系统中所有已经安装的应用信息，包括基本信息，签名和权限。

/system/etc/permissions/

### 安装概述

* 拷贝apk文件到指定目录

在Android系统中，apk安装文件是会被保存起来的，默认情况下，用户安装的apk首先会被拷贝到 /data/app 目录下。/data/app目录是用户有权限访问的目录，在安装apk的时候会自动选择该目录存放用户安装的文件，而系统出厂的apk文件则被放到了 /system 分区下,包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有Root权限的用户才能访问，这也就是为什么在**没有Root手机之前，我们无法删除系统出厂的app的原因了。**

|  |
| --- |
| **root@csb:/data # ls -al | grep app**  **drwxrwx--x system system 2017-09-16 09:38 app** |

* 解压apk，拷贝文件，创建应用的数据目录

为了加快app的启动速度，apk在安装的时候，会首先将app的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml文件、cache、二进制的so动态库等等。

* 解析apk的AndroidManifinest.xml文件

Android系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何apk文件，都会更新这个文件。这个文件位于如下目录：/data/system/packages.xml。系统在安装apk的过程中，会解析apk的AndroidManifinest.xml文件，提取出这个apk的重要信息写入到packages.xml文件中，这些信息包括：权限、应用包名、APK的安装位置、版本、userID等等。

由此，我们就知道了为啥一些**应用市场**和软件管理类的app能够很清楚地知道当前手机所安装的所有的app，以及这些**app的详细信息**了。

另外一件事就是Linux的用户Id和用户组Id，以便他可以获得合适的运行权限。以上这些都是由PackageServiceManager完成的，下面我们会重点介绍PackageServiceManager。

* 显示快捷方式

这些应用程序只是相当于在PackageManagerService服务注册好了，如果我们想要在Android桌面上看到这些应用程序，还需要有一个Home应用程序，负责从PackageManagerService服务中把这些安装好的应用程序取出来，并以友好的方式在桌面上展现出来，例如以快捷图标的形式。在Android系统中，负责把系统中已经安装的应用程序在桌面中展现出来的Home应用程序就是Launcher了

## 安装调用概述PMS

应用程序管理服务PackageManagerService是系统启动的时候由SystemServer组件启动的，启后它就会执行应用程序安装的过程.

### 系统应用安装方式



**1.PackageManagerService.main()初始化注册**

将PackageManagerService服务初始化并注册到ServiceManager里面进行管理。

**2.建立java层的installer与c层的installd的socket联接**

建立Java层的installer与c层的installd的socket联接，使得在上层的install,remove,dexopt等功能最终由installd在底层实现；

**3.建立PackageHandler消息循环**

建立PackageHandler消息循环，用于处理外部的apk安装请求消息，如adb install,packageinstaller安装apk时会发送消息；

典型的比如INIT\_COPY和MCS\_BOUND等，在通过网络下载时候会调用。

**4. 成员变量readLp（）恢复上一次的安装信息**

由于Android每次启动的时候都需要安装一次信息，但是有些信息是保持不变的，例如Linux用户组Id，PackageManagerService 每次安装程序之后，都会把这些程序的信息保存下来，以便下次使用， 恢复上一次程序的安装信息是通过PackageManagerService 的成员变量mSetting的readLP()来实现的，恢复信息之后就开始扫描和安装app了。

检查/data/system/packages.xml是否存在，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通过apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。

**5.jar的detopt优化**

检查**BootClassPath，mSharedLibraries**及/system/framework下的jar是否需要dexopt，需要的则通过dexopt进行优化；

**6.scanDirLI函数扫描特定目录的apk文件解析**

启动AppDirObserver线程监测**/system/framework,/system/app,/data/app,/data/app-privat**e目录的事件,主要监听add和remove事件。对于目录监听底层通过inotify机制实现，inotify 是一种文件系统的变化通知机制，如文件增加、删除等事件可以立刻让用户态得知,它为用户态监视文件系统的变化提供了强大的支持。当有add event时调用scanPackageLI(File , int , int)处理；当有remove event时调用removePackageLI()处理;

调用installer.install()进行安装工作,检查apk里的dex文件是否需要再优化,如果需要优化则通过辅助工具dexopt进行优化处理；将解析出的componet添加到pkg的对应列表里；对apk进行签名和证书校验,进行完整性验证。



**7.updatePermmisonLp函数分配权限**

这个函数为申请了特定资源访问权限的app，分配相应的用户组ID.

**8.writeLP()函数保存安装信息**

mSetting的writeLP（）将所获得应用程序的安装信息，保存在一个本地的配置文件中。以便下次安装的时候，将应用的信息回复过来。

### 网络下载应用安装



### cmd工具安装

包含pm(AS编译就是用的这个)和adb install（程序安装）

$ adb push \app\build\outputs\apk\app-debug.apk /data/local/tmp/system.launcher

$ adb shell pm install -r "/data/local/tmp/system.launcher"

pkg: /data/local/tmp/system.launcher

Failure [INSTALL\_FAILED\_VERSION\_DOWNGRADE]

$ adb shell pm uninstall dji.system.launcher

DELETE\_FAILED\_INTERNAL\_ERROR

Error while Installing APK

其入口函数源文件为pm.java

(源文件路径：android\frameworks\base\cmds\pm\src\com\android\commands\pm\pm.java)

其中\system\framework\pm.jar 包管理库

包管理脚本 \system\bin\pm 解析

showUsage就是使用方法

|  |
| --- |
| private static void showUsage() {  System.err.println("usage: pm [list|path|install|uninstall]");  System.err.println(" pm list packages [-f]");  System.err.println(" pm list permission-groups");  System.err.println(" pm list permissions [-g] [-f] [-d] [-u] [GROUP]");  System.err.println(" pm list instrumentation [-f] [TARGET-PACKAGE]");  System.err.println(" pm list features");  System.err.println(" pm path PACKAGE");  System.err.println(" pm install [-l] [-r] [-t] [-i INSTALLER\_PACKAGE\_NAME] [-s] [-f] PATH");  System.err.println(" pm uninstall [-k] PACKAGE");  System.err.println(" pm enable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm disable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm setInstallLocation [0/auto] [1/internal] [2/external]");  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*省略\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  } |

安装时候会调用 runInstall（）方法

private void runInstall() {

int installFlags = 0;

String installerPackageName = null;

String opt;

while ((opt=nextOption()) != null) {

if (opt.equals("-l")) {

installFlags |= PackageManager.INSTALL\_FORWARD\_LOCK;

} else **if (opt.equals("-r")) {**

**installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;**

} else if (opt.equals("-i")) {

installerPackageName = nextOptionData();

if (installerPackageName == null) {

System.err.println("Error: no value specified for -i");

showUsage();

return;

}

} else if (opt.equals("-t")) {

installFlags |= PackageManager.INSTALL\_ALLOW\_TEST;

} else if (opt.equals("-s")) {

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_EXTERNAL;

} else if (opt.equals("-f")) {

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_INTERNAL;

} else {

System.err.println("Error: Unknown option: " + opt);

showUsage();

return;

}

}

String apkFilePath = nextArg();

System.err.println("\tpkg: " + apkFilePath);

if (apkFilePath == null) {

System.err.println("Error: no package specified");

showUsage();

return;

}

PackageInstallObserver obs = new PackageInstallObserver();

try {

mPm.installPackage(Uri.fromFile(new File(apkFilePath)), obs, installFlags,

installerPackageName);

synchronized (obs) {

while (!obs.finished) {

try {

obs.wait();

} catch (InterruptedException e) {

}

}

if (obs.result == PackageManager.INSTALL\_SUCCEEDED) {

System.out.println("Success");

} else {

System.err.println("Failure ["

+ installFailureToString(obs.result)

+ "]");

}

}

} catch (RemoteException e) {

System.err.println(e.toString());

System.err.println(PM\_NOT\_RUNNING\_ERR);

}

}

其中的 因为有

mPm = IpackageManager.Stub.asInterface(ServiceManager.getService("package"));

Stub是接口IPackageManage的静态抽象类，asInterface是返回IPackageManager代理的静态方法。因为class PackageManagerService extends IPackageManager.Stub所以mPm.installPackage 调用

/\* Called when a downloaded package installation has been confirmed by the user \*/

public void installPackage(

final Uri packageURI, final IPackageInstallObserver observer, final int flags,final String installerPackageName)

这样就是从网络下载安装的入口了。



### 第三方应用安装



## 源码分析

### 配置文件package.xml

/data/system/packages.xml通过它可以看到系统安装的所有软件包，以及软件包的信息

系统自带的软件能升级吗（即安装在系统分区system中的包，如电话，短信），可以升级，如果升级/system/app目录中的包，PackageManagerServer.java对此情况进行处理，被升级的包出现package.xml的**updated-package**字段中，新的包信息会写在package字段中，卸载新包后，原包会恢复到package字段中。启动时新的包会优先地被启动

|  |
| --- |
| **<updated-package** name="com.android.providers.settings" codePath="/system/priv-app/SettingsProvider" ft="15e3b5e10b0" it="15e2e387ad8" ut="15e3b5e10b0" version="22" nativeLibraryPath="/system/priv-app/SettingsProvider/lib" primaryCpuAbi="armeabi-v7a" sharedUserId="1000">  <perms />  **</updated-package>** |

### PMS启动过程

在SystemServer的startBootstrapServices方法中获得启动pms，通过pms的main方法获得其实例。

[/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java]

**[java]** [view plain](http://blog.csdn.net/lgglkk/article/details/54579396) [copy](http://blog.csdn.net/lgglkk/article/details/54579396)

[print?](http://blog.csdn.net/lgglkk/article/details/54579396)

1. **public** **static** PackageManagerService main(Context context, Installer installer,
2. **boolean** factoryTest, **boolean** onlyCore) {
3. PackageManagerService m = **new** PackageManagerService(context, installer, factoryTest, onlyCore);
4. ServiceManager.addService("package", m);
5. **return** m;
6. }

Main方法比较简单，就是实例化了一个pms对象，然后将服务对象注册到ServiceManager中，服务名字为”package”，通过命令adb shell service list列出系统所有注册服务中，可以找到package服务。

C:\Users\key.guan>adb shell service list | findstr package

80 package: [android.content.pm.IPackageManager]

注意：pms比ams晚启动，但比ams提前SystemReady。pms的启动到ready的大致流程如（图1）所示。



Android系统在启动的过程中，启动一个应用程序管理服务PackageManagerService负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息。

应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了。

### PackageManagerService构造方法 ①

[/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java]

1. **public** PackageManagerService(Context context, Installer installer, **boolean** factoryTest, **boolean** onlyCore) {
2. EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_START, SystemClock.uptimeMillis());
3. mContext = context;
4. mFactoryTest = factoryTest;
5. mOnlyCore = onlyCore;
6. mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
7. mMetrics = **new** DisplayMetrics();
8. mSettings = **new** Settings(mPackages);
9. mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM\_UID,
10. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
11. mSettings.addSharedUserLPw("android.uid.phone", RADIO\_UID,
12. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
13. mSettings.addSharedUserLPw("android.uid.log", LOG\_UID,
14. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
15. mSettings.addSharedUserLPw("android.uid.nfc", NFC\_UID,
16. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
17. mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH\_UID,
18. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
19. mSettings.addSharedUserLPw("android.uid.shell", SHELL\_UID,
20. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);

构造方法内部首先获得一个系统级上下文mContext，并在实例化mSettings后，添加system, radio, log, nfc, bluetooth, shell  6种SharedUserSettings到mSettings。

Settings这个类包含所有安装后的apk信息，里面保存了一个mPackages映射表，根据apk包名映射对应的apk包信息，比如permissions权限信息 ，name, codePath, mSharedLibraries, restrictions, userid, version等等，这些信息将保存到一个名为 packages的XML文件中，pms服务启动时，如果packages.xml文件存在，那么会先读里面的内容初始化Settings实例，随后packages.xml文件里面的内容会随着apk安装信息的更新而更新。

Settings类结构如图（2）所示



#### ****Settings的构造方法****

Settings类的构造方法如下，主要创建data/system目录下的多个配置文件，例如packages.xml。

[/frameworks/base/services/core/java/com/android/server/pm/Settings.java]

1. Settings(Context context) {
2. **this**(context, Environment.getDataDirectory());
3. }
5. Settings(Context context, File dataDir) {
6. mSystemDir = **new** File(dataDir, "system");
7. mSystemDir.mkdirs();
8. FileUtils.setPermissions(mSystemDir.toString(),
9. FileUtils.S\_IRWXU|FileUtils.S\_IRWXG
10. |FileUtils.S\_IROTH|FileUtils.S\_IXOTH,
11. -1, -1);
12. mSettingsFilename = **new** File(mSystemDir, "packages.xml");
13. mBackupSettingsFilename = **new** File(mSystemDir, "packages-backup.xml");
14. mPackageListFilename = **new** File(mSystemDir, "packages.list");
15. FileUtils.setPermissions(mPackageListFilename, 0640, SYSTEM\_UID, PACKAGE\_INFO\_GID);
17. // Deprecated: Needed for migration
18. mStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped.xml");
19. mBackupStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped-backup.xml");
20. }

//创建data/system目录

//创建data/system/packages.xml文件

//创建data/system/pacakges-backup.xml文件

//创建data/system/packages.list文件

//创建data/system/packages-stopped.xml文件

//创建data/system/packages-stopped-backup.xml文件

#### ****创建并添加SharedUserSetting****

 Settings实例化后，调用Settings的addSharedUserLPw方法添加6个系统的sharedUser，保存在Settings的mSharedUsers数组中。下图是SharedUserSettings的类结构，其中SettingBase是SharedUserSetting的基类，基类中包含pkgFlags/pkgPrivateFlags/PermissionsState，另外SettingBase也是PackageSetting的基类。



### ****实例化SystemConfig并获得gids, systempermissions, features****

## 版本控制原理

pm install

## 错误码分析

### INSTALL FAILED CONFLICTING PROVIDER

#### 问题原因

在Android中authority要求必须是唯一的：不同applicationId&&同authority报错，同一个applicationId&&不同authority不会报错。

比如你在定义一个provider时需要为它指定一个唯一的authority。如果你在安装一个带有provider的应用时，系统会检查当前已安装应用的authority是否和你要安装应用的authority相同，如果相同则会弹出上述警告，并且安装失败。

比如2个应用都是fileprovider，**com.facebook.app.FacebookContentProvider**

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="fileprovider"

android:grantUriPermissions="true"

android:exported="false">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/file\_paths" />

</provider>

<**provider  
 android:name="com.facebook.FacebookContentProvider"  
 android:authorities="com.facebook.app.FacebookContentProvider"  
 android:exported="true"  
 tools:replace="android:authorities"** />

Log查看

|  |
| --- |
| 09-17 11:01:20.288 W/PackageManager( 488): Package couldn't be installed in /data/app/kapp4.v4-2  09-17 11:01:20.288 W/PackageManager( 488): com.android.server.pm.PackageManagerException: Can't install because provider name com.facebook.app.FacebookContentProvider111111111（in kapp4.v4) is already used by kapp3  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.scanPackageDirtyLI(PackageManagerService.java:5872)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.scanPackageLI(PackageManagerService.java:5564)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.replaceSystemPackageLI(PackageManagerService.java:10755)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.replacePackageLI(PackageManagerService.java:10581)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.installPackageLI(PackageManagerService.java:11045)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService.access$2700(PackageManagerService.java:249)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PackageManagerService$7.run(PackageManagerService.java:8953) |

#### 解决方案

在定义provider是，使用软编码的形式，如下：

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="${applicationId}.fileprovider"

android:grantUriPermissions="true"

android:exported="false">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/file\_paths" />

</provider>

上述代码中通过${applicationId}.fileprovider的形式来指定provider的authorities，所以该provider的authorities会根据applicationId的不同而不同，从而避免了authorities的冲突问题。

<**provider  
 android:name="com.facebook.FacebookContentProvider"  
 android:authorities="com.facebook.app.FacebookContentProvider${facebookId}"  
 android:exported="true"  
 tools:replace="android:authorities"** />

INSTALL\_FAILED\_CONFLICTING\_PROVIDER问题完美解决方案 <http://www.jianshu.com/p/ad8c066e9166>

#### 错误方法

删除了phone.debug.go也没用？

-d有无也没有用

cnt=1000 可用？

pm list packages | grep dji.go.v4 none

root@cs500c:/system/priv-app/DJI-GO4 # ls -al

-rw-r--r-- root root 135979984 2017-09-15 09:59 DJI-GO4.apk

<package name="dji.go.v4" codePath="/system/priv-app/DJI-GO4" version="25701"

clear data reboot 恢复称为cs版本

#### 源码分析

1. **if** ((scanFlags & SCAN\_NEW\_INSTALL) != 0) {
2. **for** (**int** i=0; i<pkg.providers.size(); i++) {
3. PackageParser.Provider p = pkg.providers.get(i);
4. **if** (p.info.authority != **null**) {
5. String names[] = p.info.authority.split(";");
6. **for** (**int** j = 0; j < names.length; j++) {
7. if (mProvidersByAuthority.containsKey(names[j])) {
8. PackageParser.Provider other = mProvidersByAuthority.get(names[j]);
9. **final** String otherPackageName = ((other != **null** && other.getComponentName()
10. != **null**) ? other.getComponentName().getPackageName() : "?");
11. **throw** **new** PackageManagerException(**INSTALL\_FAILED\_CONFLICTING\_PROVIDER**,
12. "Can't install because provider name " + names[j] +
13. " (in package " + pkg.applicationInfo.packageName
14. + ") is already used by " + otherPackageName);
15. }
16. }
17. }
18. }
19. }

## 总结

1.安装和卸载都是通过PackageManager，实质上是实现了PackageManager的远程服务PackageManagerService来完成具体的操作，所有细节和逻辑均可以在PackageManagerService中跟踪查看；

2.所有安装方式殊途同归，最终就回到PackageManagerService中，然后调用底层本地代码的installd来完成。

3.再看apk 的安装过程。回个我们再看apk的安装过程，主要分为如下几部

* 拷贝apk文件到指定目录
* 解压apk，拷贝文件，创建应用的数据目录
* 解析apk的AndroidManifinest.xml文件
* 向Launcher应用申请添加创建快捷方式

各个系统的版本如下:



## Ref

[深度探究apk安装过程](http://www.sohu.com/a/132599459_465908)

[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)

[pms包管理服务分析-初步理解](http://blog.csdn.net/lgglkk/article/details/54579396)

[pms包管理服务分析-PackageManagerService构造函数和包扫描过程](http://blog.csdn.net/lgglkk/article/details/54839662)

// TODO word

# pms

不论是cmd安装，还是预装market安装，还是ui安装，最终都会调用到installPackage这个方法入口，本节单独讨论系统是如何执行这一个过程的

## 文件拷贝阶段

### installPackage

installPackage方法只是用当前用户安装应用，最后也会调用installPackageAsUser  
//TODO 用户的概念？一个进程就是一个用户？

@Override

public void installPackage(String originPath, IPackageInstallObserver2 observer,int installFlags, String installerPackageName, VerificationParams verificationParams,String packageAbiOverride) {

installPackageAsUser(originPath, observer, installFlags,installerPackageName, verificationParams,packageAbiOverride, UserHandle.getCallingUserId());

}

### installPackageAsUser

installPackageAsUser先检查调用进程是否有安装应用的权限，[再检查调用进程所属的用户是否有权限安装应用](file:///F:\key.guan\kgszgt\asys\%E4%B8%BA%E4%BD%95%E9%9C%80%E8%A6%81%E6%A3%80%E6%9F%A5%E7%94%A8%E6%88%B7?)，最后检查指定的用户是否被限制安装应用。如果参数installFlags带有INSTALL\_ALL\_USERS，则该应用将给系统中所有用户安装，否则只给指定用户安装。安装应用实践比较长，因此不可能在一个函数中完成。上面函数把数据保存在installParams然后发送了INIT\_COPY消息。通过PackageHandler的实例mHandler.sendMessage（msg）把信息发给继承Handler的类HandleMessage()方法会自动调用Packagemanager的安装方法installPackage（），发送消息时会传递一个InstallParams参数，InstallParams是继承自HandlerParams抽象类的，用来记录安装应用的参数。

@Override

public void installPackageAsUser(String originPath, IPackageInstallObserver2 observer,

int installFlags, String installerPackageName, VerificationParams verificationParams,

String packageAbiOverride, int userId) {

//检查调用进程的权限,比如PackageInstaller.apk这个系统应用就必须申请这个权限

mContext.enforceCallingOrSelfPermission(android.Manifest.permission.INSTALL\_PACKAGES, null);

//检查调用进程的用户是否有权限安装应用

final int callingUid = Binder.getCallingUid();

enforceCrossUserPermission(callingUid, userId, true, true, "installPackageAsUser");

//检查指定的用户是否被限制安装应用

// TODO DISALLOW\_INSTALL\_APPS 是安装黑名单

if (isUserRestricted(userId, UserManager.DISALLOW\_INSTALL\_APPS)) {

try {

if (observer != null) {

observer.onPackageInstalled("", INSTALL\_FAILED\_USER\_RESTRICTED, null, null);

}

} catch (RemoteException re) {

}

return;

}

//adb INSTALL\_FAILED\_USER\_RESTRICTED

if ((callingUid == Process.SHELL\_UID) || (callingUid == Process.ROOT\_UID)) {

installFlags |= PackageManager.INSTALL\_FROM\_ADB;

} else {

// Caller holds INSTALL\_PACKAGES permission, so we're less strict

// about installerPackageName.

installFlags &= ~PackageManager.INSTALL\_FROM\_ADB;

installFlags &= ~PackageManager.INSTALL\_ALL\_USERS;

}

//给所有用户安装

UserHandle user;

if ((installFlags & PackageManager.INSTALL\_ALL\_USERS) != 0) {

user = UserHandle.ALL;

} else {

user = new UserHandle(userId);

}

verificationParams.setInstallerUid(callingUid);

final File originFile = new File(originPath);

final OriginInfo origin = OriginInfo.fromUntrustedFile(originFile);

//保存参数到InstallParamsm,发送消息

final Message msg = mHandler.obtainMessage(INIT\_COPY);

msg.obj = new InstallParams(origin, observer, installFlags,

installerPackageName, verificationParams, user, packageAbiOverride);

mHandler.sendMessage(msg);

}

### doHandleMessage-INIT\_COPY

void doHandleMessage(Message msg) {

switch (msg.what) {

case INIT\_COPY: {

HandlerParams params = (HandlerParams) msg.obj;

int idx = mPendingInstalls.size();

if (DEBUG\_INSTALL) Slog.i(TAG, "init\_copy idx=" + idx + ": " + params);

// If a bind was already initiated we dont really

// need to do anything. The pending install

// will be processed later on.

if (!mBound) {

// If this is the only one pending we might

// have to bind to the service again.

if (!connectToService()) {//绑定DefaultContainerService

Slog.e(TAG, "Failed to bind to media container service");

params.serviceError();

return;

} else {//连接成功把安装信息保存到mPendingInstalls

// Once we bind to the service, the first

// pending request will be processed.

mPendingInstalls.add(idx, params);

}

} else {//如果已经绑定好了

mPendingInstalls.add(idx, params);

// Already bound to the service. Just make

// sure we trigger off processing the first request.

if (idx == 0) {

mHandler.sendEmptyMessage(MCS\_BOUND);

}

}

break;

}

INIT\_COPY消息的处理将绑定DefaultContainerService，因为这是一个异步的过程，要等待绑定的结果通过onServiceConnected返回，所以这里的安装参数放到了mPendingInstalls列表中。如果这个Service以前就绑定好了，现在就不需要再绑定，安装信息也会先放到mPendingInstalls。如果有多个安装请求同时到达，这里通过mPendingInstalls列表对他们进行排队。如果列表中只有一项，说明没有更多的安装请求，因此这种情况下回立即发出MCS\_BOUND消息。而onServiceConnected方法同样是发出MCS\_BOUND消息：

class DefaultContainerConnection implements ServiceConnection {

public void onServiceConnected(ComponentName name, IBinder service) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG, "onServiceConnected");

IMediaContainerService imcs =

IMediaContainerService.Stub.asInterface(service);

mHandler.sendMessage(mHandler.obtainMessage(MCS\_BOUND, imcs));

}

public void onServiceDisconnected(ComponentName name) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG, "onServiceDisconnected");

}

};

看下MCS\_BOUND的消息处理

case MCS\_BOUND: {

if (DEBUG\_INSTALL) Slog.i(TAG, "mcs\_bound");

if (msg.obj != null) {

mContainerService = (IMediaContainerService) msg.obj;

}

if (mContainerService == null) {//没有连接成功

// Something seriously wrong. Bail out

Slog.e(TAG, "Cannot bind to media container service");

for (HandlerParams params : mPendingInstalls) {

// Indicate service bind error

params.serviceError();//通知出错了

}

mPendingInstalls.clear();

} else if (mPendingInstalls.size() > 0) {

HandlerParams params = mPendingInstalls.get(0);

if (params != null) {

if (params.startCopy()) {//执行安装

// We are done... look for more work or to

// go idle.

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Checking for more work or unbind...");

// Delete pending install

if (mPendingInstalls.size() > 0) {

mPendingInstalls.remove(0);//工作完成，删除第一项

}

if (mPendingInstalls.size() == 0) {//如果没有安装消息了，延时发送10秒MCS\_UNBIND消息

if (mBound) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Posting delayed MCS\_UNBIND");

removeMessages(MCS\_UNBIND);

Message ubmsg = obtainMessage(MCS\_UNBIND);

// Unbind after a little delay, to avoid

// continual thrashing.

sendMessageDelayed(ubmsg, 10000);

}

} else {

// There are more pending requests in queue.

// Just post MCS\_BOUND message to trigger processing

// of next pending install.

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Posting MCS\_BOUND for next work");

mHandler.sendEmptyMessage(MCS\_BOUND);//还有消息继续发送MCS\_BOUND消息

}

}

}

} else {

// Should never happen ideally.

Slog.w(TAG, "Empty queue");

}

break;

}

如果结束了我们看看MCS\_UNBIND消息的处理

case MCS\_UNBIND: {

// If there is no actual work left, then time to unbind.

if (DEBUG\_INSTALL) Slog.i(TAG, "mcs\_unbind");

if (mPendingInstalls.size() == 0 && mPendingVerification.size() == 0) {

if (mBound) {

if (DEBUG\_INSTALL) Slog.i(TAG, "calling disconnectService()");

disconnectService();//断开连接

}

} else if (mPendingInstalls.size() > 0) {

// There are more pending requests in queue.

// Just post MCS\_BOUND message to trigger processing

// of next pending install.

mHandler.sendEmptyMessage(MCS\_BOUND);

}

break;

}

MCS\_UNBIND消息的处理，如果处理的时候发现mPendingInstalls又有数据了，还是发送MCS\_BOUND消息继续安装，否则断开和DefaultContainerService的连接，安装结束。这个安装会尝试4次，超过4次就GG了  
下面我们看执行安装的函数startCopy：

final boolean startCopy() {

boolean res;

try {

if (DEBUG\_INSTALL) Slog.i(TAG, "startCopy " + mUser + ": " + this);

if (++mRetries > MAX\_RETRIES) {//重试超过4次退出

Slog.w(TAG, "Failed to invoke remote methods on default container service. Giving up");

mHandler.sendEmptyMessage(MCS\_GIVE\_UP);

handleServiceError();

return false;

} else {

handleStartCopy();

res = true;

}

} catch (RemoteException e) {

if (DEBUG\_INSTALL) Slog.i(TAG, "Posting install MCS\_RECONNECT");

mHandler.sendEmptyMessage(MCS\_RECONNECT);//安装出错，发送重新连接

res = false;

}

handleReturnCode();

return res;

}

### InstallParams.handleStartCopy

handleStartCopy()执行的工作如下：

* 判断安装标志位是否合法
* 判断安装空间是否足够
* 对安装位置的校验
* 判断是否需要对应用进行校验工作
* 如果校验成功，执行InstallArgs.copyApk()
* 如果无需校验，直接执行InstallArgs.copyApk()

handleStartCopy函数先通过DefaultContainerService调用了getMinimallPackageInfo来确定安装位置是否有足够的空间，并在PackageInfoLite对象的recommendedIntallLocation记录错误原因。发现空间不够，会调用installer的freecache方法来释放一部分空间。  
// 首先对安装的标志位进行判断，如果既有内部安装标志，又有外部安装标志，那么就设置  
//PackageManager.INSTALL\_FAILED\_INVALID\_INSTALL\_LOCATION返回值  
再接下来handleStartCopy有很长一段都在处理apk的校验，这个校验过程是通过发送Intent ACTION\_PACKAGE\_NEEDS\_VERIFICATION给系统中所有接受该Intent的应用来完成。如果无需校验，直接调用InstallArgs对象的copyApk方法。

这个方法比较长，分段来看。

ret = PackageManager.INSTALL\_SUCCEEDED

final StorageManager storage = StorageManager.from(mContext);

final long lowThreshold = storage.getStorageLowBytes(

Environment.getDataDirectory());

final long sizeBytes = mContainerService.calculateInstalledSize(

origin.resolvedPath, isForwardLocked(), packageAbiOverride);

if (mInstaller.freeCache(null, sizeBytes + lowThreshold) >= 0) {

pkgLite = mContainerService.getMinimalPackageInfo(origin.resolvedPath,

installFlags, packageAbiOverride);

}

首先，如果需要的空间不够大，就调用Install的freeCache去释放一部分缓存。这里的mContainerService对应的binder服务端实现，在DefaultContainerService中。中间经过复杂（安装位置，pkgLite.recommendedIntallLocation，安装位置的校验，installLocationPoliy策略等）的判断处理之后，创建一个InstallArgs对象，如果前面的判断结果是能安装成功的话ret=PackageManager.INSTALL\_SUCCEEDED，进入分支。  
// TODO installLocationPoliy() 是位置的策略PackageINfoLite

if (ret == PackageManager.INSTALL\_SUCCEEDED) {

/\*

\* ADB installs appear as UserHandle.USER\_ALL, and can only be performed by

\* UserHandle.USER\_OWNER, so use the package verifier for UserHandle.USER\_OWNER.

\*/

int userIdentifier = getUser().getIdentifier();

if (userIdentifier == UserHandle.USER\_ALL

&& ((installFlags & PackageManager.INSTALL\_FROM\_ADB) != 0)) {

userIdentifier = UserHandle.USER\_OWNER;

}

/\*

\* Determine if we have any installed package verifiers. If we

\* do, then we'll defer to them to verify the packages.

\*/

final int requiredUid = mRequiredVerifierPackage == null ? -1

: getPackageUid(mRequiredVerifierPackage, userIdentifier);

if (!origin.existing && requiredUid != -1

&& isVerificationEnabled(userIdentifier, installFlags)) {

final Intent verification = new Intent(

Intent.ACTION\_PACKAGE\_NEEDS\_VERIFICATION);

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

verification.setDataAndType(Uri.fromFile(new File(origin.resolvedPath)),

PACKAGE\_MIME\_TYPE);

verification.addFlags(Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION);

final List<ResolveInfo> receivers = queryIntentReceivers(verification,

PACKAGE\_MIME\_TYPE, PackageManager.GET\_DISABLED\_COMPONENTS,

0 /\* TODO: Which userId? \*/);

if (DEBUG\_VERIFY) {

Slog.d(TAG, "Found " + receivers.size() + " verifiers for intent "

+ verification.toString() + " with " + pkgLite.verifiers.length

+ " optional verifiers");

}

final int verificationId = mPendingVerificationToken++;

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_ID, verificationId);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALLER\_PACKAGE,

installerPackageName);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALL\_FLAGS,

installFlags);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_PACKAGE\_NAME,

pkgLite.packageName);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_VERSION\_CODE,

pkgLite.versionCode);

if (verificationParams != null) {

if (verificationParams.getVerificationURI() != null) {

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_URI,

verificationParams.getVerificationURI());

}

if (verificationParams.getOriginatingURI() != null) {

verification.putExtra(Intent.EXTRA\_ORIGINATING\_URI,

verificationParams.getOriginatingURI());

}

if (verificationParams.getReferrer() != null) {

verification.putExtra(Intent.EXTRA\_REFERRER,

verificationParams.getReferrer());

}

if (verificationParams.getOriginatingUid() >= 0) {

verification.putExtra(Intent.EXTRA\_ORIGINATING\_UID,

verificationParams.getOriginatingUid());

}

if (verificationParams.getInstallerUid() >= 0) {

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALLER\_UID,

verificationParams.getInstallerUid());

}

}

final PackageVerificationState verificationState = new PackageVerificationState(

requiredUid, args);

mPendingVerification.append(verificationId, verificationState);

final List<ComponentName> sufficientVerifiers = matchVerifiers(pkgLite,

receivers, verificationState);

// Apps installed for "all" users use the device owner to verify the app

UserHandle verifierUser = getUser();

if (verifierUser == UserHandle.ALL) {

verifierUser = UserHandle.OWNER;

}

/\*

\* If any sufficient verifiers were listed in the package

\* manifest, attempt to ask them.

\*/

if (sufficientVerifiers != null) {

final int N = sufficientVerifiers.size();

if (N == 0) {

Slog.i(TAG, "Additional verifiers required, but none installed.");

ret = PackageManager.INSTALL\_FAILED\_VERIFICATION\_FAILURE;

} else {

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

final ComponentName verifierComponent = sufficientVerifiers.get(i);

final Intent sufficientIntent = new Intent(verification);

sufficientIntent.setComponent(verifierComponent);

mContext.sendBroadcastAsUser(sufficientIntent, verifierUser);

}

}

}

final ComponentName requiredVerifierComponent = matchComponentForVerifier(

mRequiredVerifierPackage, receivers);

if (ret == PackageManager.INSTALL\_SUCCEEDED

&& mRequiredVerifierPackage != null) {

/\*

\* Send the intent to the required verification agent,

\* but only start the verification timeout after the

\* target BroadcastReceivers have run.

\*/

verification.setComponent(requiredVerifierComponent);

mContext.sendOrderedBroadcastAsUser(verification, verifierUser,

android.Manifest.permission.PACKAGE\_VERIFICATION\_AGENT,

new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

final Message msg = mHandler

.obtainMessage(CHECK\_PENDING\_VERIFICATION);

msg.arg1 = verificationId;

mHandler.sendMessageDelayed(msg, getVerificationTimeout());

}

}, null, 0, null, null);

/\*

\* We don't want the copy to proceed until verification

\* succeeds, so null out this field.

\*/

mArgs = null;

}

} else {

/\*

\* No package verification is enabled, so immediately start

\* the remote call to initiate copy using temporary file.

\*/

ret = args.copyApk(mContainerService, true);

}

}

InstallArgs是个抽象类，一共有三个实现类MoveInstallArgs（针对已有文件的Move）、AsecInstallArgs（针对SD卡）和FileInstallArgs（针对内部存储），会在createInstallArgs()方法中根据不同的参数返回不同的实现类。  
接下来分析FileInstallArgs.copyApk()方法：

### FileInstallArgs.copyApk()

int copyApk(IMediaContainerService imcs, boolean temp) throws RemoteException {

// 已经执行过copy了

if (origin.staged) {

codeFile = origin.file;

resourceFile = origin.file;

return PackageManager.INSTALL\_SUCCEEDED;

}

try {

// 在/data/app/下面生成一个类似vmdl1354353418.tmp的临时文件

final File tempDir = mInstallerService.allocateStageDirLegacy(volumeUuid);

codeFile = tempDir;

resourceFile = tempDir;

} catch (IOException e) {

return PackageManager.INSTALL\_FAILED\_INSUFFICIENT\_STORAGE;

}

// 在imcs.copyPackage()中会调用target.open()，返回一个文件描述符

final IParcelFileDescriptorFactory target = new IParcelFileDescriptorFactory.Stub() {

@Override

public ParcelFileDescriptor open(String name, int mode) throws RemoteException {

if (!FileUtils.isValidExtFilename(name)) {

throw new IllegalArgumentException("Invalid filename: " + name);

}

try {

final File file = new File(codeFile, name);

final FileDescriptor fd = Os.open(file.getAbsolutePath(),

O\_RDWR | O\_CREAT, 0644);

Os.chmod(file.getAbsolutePath(), 0644);

return new ParcelFileDescriptor(fd);

} catch (ErrnoException e) {

throw new RemoteException("Failed to open: " + e.getMessage());

}

}

};

int ret = PackageManager.INSTALL\_SUCCEEDED;

// 调用DefaultContainerService.mBinder.copyPackage()方法复制文件到target.open()方法指定的文件中，也即是上面产生的临时文件

ret = imcs.copyPackage(origin.file.getAbsolutePath(), target);

if (ret != PackageManager.INSTALL\_SUCCEEDED) {

return ret;

}

final File libraryRoot = new File(codeFile, LIB\_DIR\_NAME);

NativeLibraryHelper.Handle handle = null;

try {

handle = NativeLibraryHelper.Handle.create(codeFile);

ret = NativeLibraryHelper.copyNativeBinariesWithOverride(handle, libraryRoot,

abiOverride);

} catch (IOException e) {

ret = PackageManager.INSTALL\_FAILED\_INTERNAL\_ERROR;

} finally {

IoUtils.closeQuietly(handle);

}

return ret;

}

而copyApk方法同样是调用DefaultContainerService的copyPackage将应用的文件复制到/data/app下，如果还有native动态库，也会把包在apk文件中的动态库提取出来。

执行完copyApk后，应用安装到了data/app目录下了。

### InstallParams.handleReturnCode()

在handleStartCopy()执行完之后，文件复制工作阶段的工作已经完成了，接下来会在startCopy()中调用handleReturnCode()->processPendingInstall()来进行应用的解析和装载。

## 解析应用阶段

这个阶段的工作是对安装包进行扫描优化，把应用转换成oat格式，然后装载到内存中去。

### processPendingInstall()

private void processPendingInstall(final InstallArgs args, final int currentStatus) {

// 以异步的方式执行安装，因为安装工作可能持续时间比较长

mHandler.post(new Runnable() {

public void run() {

// 防止重复调用

mHandler.removeCallbacks(this);

PackageInstalledInfo res = new PackageInstalledInfo();

res.returnCode = currentStatus;

res.uid = -1;

res.pkg = null;

res.removedInfo = new PackageRemovedInfo();

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED) {

// 如果前面返回的是执行成功的返回值

args.doPreInstall(res.returnCode);

synchronized (mInstallLock) {

// 开始安装应用，带LI后缀的函数执行时要带mInstallLock锁

installPackageLI(args, res);

}

// 执行doPostInstall()，这里主要分析一下FileInstallArgs.doPostInstall()

// 如果没有安装成功，这里会清除前面生成的临时文件

args.doPostInstall(res.returnCode, res.uid);

}

// 执行备份，在下面的情况下会执行备份：1.安装成功，2.是一个新的安装而不是一个升级的操作，3.新的安装包还没有执行过备份操作

final boolean update = res.removedInfo.removedPackage != null;

final int flags = (res.pkg == null) ? 0 : res.pkg.applicationInfo.flags;

boolean doRestore = !update

&& ((flags & ApplicationInfo.FLAG\_ALLOW\_BACKUP) != 0);

// Set up the post-install work request bookkeeping. This will be used

// and cleaned up by the post-install event handling regardless of whether

// there's a restore pass performed. Token values are >= 1.

int token;

if (mNextInstallToken < 0) mNextInstallToken = 1;

token = mNextInstallToken++;

PostInstallData data = new PostInstallData(args, res);

mRunningInstalls.put(token, data);

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED && doRestore) {

IBackupManager bm = IBackupManager.Stub.asInterface(

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

if (bm != null) {

try {

if (bm.isBackupServiceActive(UserHandle.USER\_OWNER)) {

bm.restoreAtInstall(res.pkg.applicationInfo.packageName, token);

} else {

doRestore = false;

}

} catch (RemoteException e) {

} catch (Exception e) {

doRestore = false;

}

} else {

doRestore = false;

}

}

if (!doRestore) {

// 发送POST\_INSTALL消息

Message msg = mHandler.obtainMessage(POST\_INSTALL, token, 0);

mHandler.sendMessage(msg);

}

}

});

}

processPendingInstall()方法内部是以异步的方式继续执行安装工作的，首先来调用installPackageLI()执行安装工作，然后调用doPostInstall()对前面的工作的返回结果进行处理，如果没有安装成功，执行清除的工作。然后再执行备份操作。  
下面来看一下installPackageLI()方法：

### installPackageLI()

installPackageLI()方法首先解析apk安装包，然后判断当前是否有安装该应用，然后根据不同的情况进行不同的处理，然后进行Dex优化操作。如果是升级安装，调用replacePackageLI()。如果是新安装，调用installNewPackageLI()。这两个方法会在下面详细介绍。

processPendingInstall()方法中执行安装的最后是发送POST\_INSTALL消息，现在来看一下这个消息需要处理的事情：

private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {

final int installFlags = args.installFlags;

final String installerPackageName = args.installerPackageName;

final String volumeUuid = args.volumeUuid;

final File tmpPackageFile = new File(args.getCodePath());

final boolean forwardLocked = ((installFlags & PackageManager.INSTALL\_FORWARD\_LOCK) != 0);

final boolean onExternal = (((installFlags & PackageManager.INSTALL\_EXTERNAL) != 0)

|| (args.volumeUuid != null));

boolean replace = false;

int scanFlags = SCAN\_NEW\_INSTALL | SCAN\_UPDATE\_SIGNATURE;

if (args.move != null) {

scanFlags |= SCAN\_INITIAL;

}

res.returnCode = PackageManager.INSTALL\_SUCCEEDED;

// 创建apk解析器

final int parseFlags = mDefParseFlags | PackageParser.PARSE\_CHATTY

| (forwardLocked ? PackageParser.PARSE\_FORWARD\_LOCK : 0)

| (onExternal ? PackageParser.PARSE\_EXTERNAL\_STORAGE : 0);

PackageParser pp = new PackageParser();

pp.setSeparateProcesses(mSeparateProcesses);

pp.setDisplayMetrics(mMetrics);

final PackageParser.Package pkg;

try {

// 开始解析文件，解析apk的信息存储在PackageParser.Package中

pkg = pp.parsePackage(tmpPackageFile, parseFlags);

} catch (PackageParserException e) {

res.setError("Failed parse during installPackageLI", e);

return;

}

......

// 获取安装包的签名和AndroidManifest摘要

try {

pp.collectCertificates(pkg, parseFlags);

pp.collectManifestDigest(pkg);

} catch (PackageParserException e) {

res.setError("Failed collect during installPackageLI", e);

return;

}

if (args.manifestDigest != null) {

// 与installPackage()方法传递过来的VerificationParams获取的AndroidManifest摘要进行对比

if (!args.manifestDigest.equals(pkg.manifestDigest)) {

res.setError(INSTALL\_FAILED\_PACKAGE\_CHANGED, "Manifest digest changed");

return;

}

} else if (DEBUG\_INSTALL) {...}

// Get rid of all references to package scan path via parser.

pp = null;

String oldCodePath = null;

boolean systemApp = false;

synchronized (mPackages) {

// 判断是否是升级当前已有应用

if ((installFlags & PackageManager.INSTALL\_REPLACE\_EXISTING) != 0) {

String oldName = mSettings.mRenamedPackages.get(pkgName);

if (pkg.mOriginalPackages != null

&& pkg.mOriginalPackages.contains(oldName)

&& mPackages.containsKey(oldName)) {

// 如果当前应用已经被升级过

pkg.setPackageName(oldName);

pkgName = pkg.packageName;

replace = true;

} else if (mPackages.containsKey(pkgName)) {

// 当前应用没有被升级过

replace = true;

}

// 如果已有应用oldTargetSdk大于LOLLIPOP\_MR1(22)，新升级应用小于LOLLIPOP\_MR1，则不允许降级安装

// 因为AndroidM(23)引入了全新的权限管理方式：动态权限管理

if (replace) {

PackageParser.Package oldPackage = mPackages.get(pkgName);

final int oldTargetSdk = oldPackage.applicationInfo.targetSdkVersion;

final int newTargetSdk = pkg.applicationInfo.targetSdkVersion;

if (oldTargetSdk > Build.VERSION\_CODES.LOLLIPOP\_MR1

&& newTargetSdk <= Build.VERSION\_CODES.LOLLIPOP\_MR1) {

...

return;

}

}

}

PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps != null) {

if (shouldCheckUpgradeKeySetLP(ps, scanFlags)) {

// 判断签名是否一致

if (!checkUpgradeKeySetLP(ps, pkg)) {

...

return;

}

} else {

try {

verifySignaturesLP(ps, pkg);

} catch (PackageManagerException e) {

...

return;

}

}

oldCodePath = mSettings.mPackages.get(pkgName).codePathString;

if (ps.pkg != null && ps.pkg.applicationInfo != null) {

// 判断是否是系统应用

systemApp = (ps.pkg.applicationInfo.flags &

// 给origUsers赋值，此变量代表哪些用户以前已经安装过该应用

res.origUsers = ps.queryInstalledUsers(sUserManager.getUserIds(), true);

}

// Check whether the newly-scanned package wants to define an already-defined perm

int N = pkg.permissions.size();

for (int i = N-1; i >= 0; i--) {

PackageParser.Permission perm = pkg.permissions.get(i);

BasePermission bp = mSettings.mPermissions.get(perm.info.name);

if (bp != null) {

// If the defining package is signed with our cert, it's okay. This

// also includes the "updating the same package" case, of course.

// "updating same package" could also involve key-rotation.

final boolean sigsOk;

if (bp.sourcePackage.equals(pkg.packageName)

&& (bp.packageSetting instanceof PackageSetting)

&& (shouldCheckUpgradeKeySetLP((PackageSetting) bp.packageSetting,

scanFlags))) {

sigsOk = checkUpgradeKeySetLP((PackageSetting) bp.packageSetting, pkg);

} else {

sigsOk = compareSignatures(bp.packageSetting.signatures.mSignatures,

pkg.mSignatures) == PackageManager.SIGNATURE\_MATCH;

}

if (!sigsOk) {

// If the owning package is the system itself, we log but allow

// install to proceed; we fail the install on all other permission

// redefinitions.

if (!bp.sourcePackage.equals("android")) {

res.setError(INSTALL\_FAILED\_DUPLICATE\_PERMISSION, "Package "

+ pkg.packageName + " attempting to redeclare permission "

+ perm.info.name + " already owned by " + bp.sourcePackage);

res.origPermission = perm.info.name;

res.origPackage = bp.sourcePackage;

return;

} else {

pkg.permissions.remove(i);

}

}

}

}

}

// 系统应用不允许安装在SDCard上

if (systemApp && onExternal) {

res.setError(INSTALL\_FAILED\_INVALID\_INSTALL\_LOCATION,

"Cannot install updates to system apps on sdcard");

return;

}

// 下面将会进行Dex优化操作

if (args.move != null) {

// 如果是针对已有文件的Move，就不用在进行Dex优化了

scanFlags |= SCAN\_NO\_DEX;

scanFlags |= SCAN\_MOVE;

synchronized (mPackages) {

final PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps == null) {

res.setError(INSTALL\_FAILED\_INTERNAL\_ERROR,

"Missing settings for moved package " + pkgName);

}

pkg.applicationInfo.primaryCpuAbi = ps.primaryCpuAbiString;

pkg.applicationInfo.secondaryCpuAbi = ps.secondaryCpuAbiString;

}

} else if (!forwardLocked && !pkg.applicationInfo.isExternalAsec()) {

// 没有设置了PRIVATE\_FLAG\_FORWARD\_LOCK标志且不是安装在外部SD卡

// 使能 SCAN\_NO\_DEX 标志位，在后面的操作中会跳过 dexopt

scanFlags |= SCAN\_NO\_DEX;

try {

derivePackageAbi(pkg, new File(pkg.codePath), args.abiOverride,

true /\* extract libs \*/);

} catch (PackageManagerException pme) {

res.setError(INSTALL\_FAILED\_INTERNAL\_ERROR, "Error deriving application ABI");

return;

}

// 进行DexOpt操作，会调用install 的dexopt命令，优化后的文件放在 /data/dalvik-cache/ 下面

int result = mPackageDexOptimizer

.performDexOpt(pkg, null /\* instruction sets \*/, false /\* forceDex \*/,

false /\* defer \*/, false /\* inclDependencies \*/,

true /\* boot complete \*/);

if (result == PackageDexOptimizer.DEX\_OPT\_FAILED) {

res.setError(INSTALL\_FAILED\_DEXOPT, "Dexopt failed for " + pkg.codePath);

return;

}

}

// 重命名/data/app/下面应用的目录名字，调用getNextCodePath()来获取目录名称，类似com.android.browser-1

if (!args.doRename(res.returnCode, pkg, oldCodePath)) {

res.setError(INSTALL\_FAILED\_INSUFFICIENT\_STORAGE, "Failed rename");

return;

}

startIntentFilterVerifications(args.user.getIdentifier(), replace, pkg);

if (replace) {

// 如果是安装升级包，调用replacePackageLI

replacePackageLI(pkg, parseFlags, scanFlags | SCAN\_REPLACING, args.user,

installerPackageName, volumeUuid, res);

} else {

// 如果安装的新应用，调用installNewPackageLI

installNewPackageLI(pkg, parseFlags, scanFlags | SCAN\_DELETE\_DATA\_ON\_FAILURES,

args.user, installerPackageName, volumeUuid, res);

}

synchronized (mPackages) {

final PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps != null) {

// 安装完成后，给newUsers赋值，此变量代表哪些用户刚刚安装过该应用

res.newUsers = ps.queryInstalledUsers(sUserManager.getUserIds(), true);

}

}

}

processPendingInstall()来进行应用的解析和装载

installPackageLI()方法首先解析apk安装包，然后判断当前是否有安装该应用，然后根据不同的情况进行不同的处理，然后进行Dex优化操作。如果是升级安装，调用replacePackageLI()。如果是新安装，调用installNewPackageLI()。这两个方法会在下面详细介绍。

processPendingInstall()方法中执行安装的最后是发送POST\_INSTALL消息，现在来看一下这个消息需要处理的事情：

### doHandleMessage- POST\_INSTALL

case POST\_INSTALL: {

//从正在安装队列中将当前正在安装的任务删除

PostInstallData data = mRunningInstalls.get(msg.arg1);

mRunningInstalls.delete(msg.arg1);

boolean deleteOld = false;

if (data != null) {

InstallArgs args = data.args;

PackageInstalledInfo res = data.res;

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED) {

final String packageName = res.pkg.applicationInfo.packageName;

res.removedInfo.sendBroadcast(false, true, false);

Bundle extras = new Bundle(1);

extras.putInt(Intent.EXTRA\_UID, res.uid);

// 现在已经成功的安装了应用，在发送广播之前先授予一些必要的权限

// 这些权限在 installPackageAsUser 中创建 InstallParams 时传递的，为null

if ((args.installFlags

& PackageManager.INSTALL\_GRANT\_RUNTIME\_PERMISSIONS) != 0) {

grantRequestedRuntimePermissions(res.pkg, args.user.getIdentifier(),

args.installGrantPermissions);

}

// 看一下当前应用对于哪些用户是第一次安装，哪些用户是升级安装

int[] firstUsers;

int[] updateUsers = new int[0];

if (res.origUsers == null || res.origUsers.length == 0) {

// 所有用户都是第一次安装

firstUsers = res.newUsers;

} else {

firstUsers = new int[0];

// 这里再从刚刚已经安装该包的用户中选出哪些是以前已经安装过该包的用户

for (int i=0; i<res.newUsers.length; i++) {

int user = res.newUsers[i];

boolean isNew = true;

for (int j=0; j<res.origUsers.length; j++) {

if (res.origUsers[j] == user) {

// 找到以前安装过该包的用户

isNew = false;

break;

}

}

if (isNew) {

int[] newFirst = new int[firstUsers.length+1];

System.arraycopy(firstUsers, 0, newFirst, 0,

firstUsers.length);

newFirst[firstUsers.length] = user;

firstUsers = newFirst;

} else {

int[] newUpdate = new int[updateUsers.length+1];

System.arraycopy(updateUsers, 0, newUpdate, 0,

updateUsers.length);

newUpdate[updateUsers.length] = user;

updateUsers = newUpdate;

}

}

}

//为新安装用户发送广播ACTION\_PACKAGE\_ADDED

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_ADDED,

packageName, extras, null, null, firstUsers);

final boolean update = res.removedInfo.removedPackage != null;

if (update) {

extras.putBoolean(Intent.EXTRA\_REPLACING, true);

}

//为升级安装用户发送广播ACTION\_PACKAGE\_ADDED

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_ADDED,

packageName, extras, null, null, updateUsers);

if (update) {

// 如果是升级安装，还会发送ACTION\_PACKAGE\_REPLACED和ACTION\_MY\_PACKAGE\_REPLACED广播

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_REPLACED,

packageName, extras, null, null, updateUsers);

sendPackageBroadcast(Intent.ACTION\_MY\_PACKAGE\_REPLACED,

null, null, packageName, null, updateUsers);

// 判断该包是否是设置了PRIVATE\_FLAG\_FORWARD\_LOCK标志或者是安装在外部SD卡

if (res.pkg.isForwardLocked() || isExternal(res.pkg)) {

int[] uidArray = new int[] { res.pkg.applicationInfo.uid };

ArrayList<String> pkgList = new ArrayList<String>(1);

pkgList.add(packageName);

sendResourcesChangedBroadcast(true, true,

pkgList,uidArray, null);

}

}

if (res.removedInfo.args != null) {

// 删除被替换应用的资源目录标记位

deleteOld = true;

}

// 针对Browser的一些处理

if (firstUsers.length > 0) {

if (packageIsBrowser(packageName, firstUsers[0])) {

synchronized (mPackages) {

for (int userId : firstUsers) {

mSettings.setDefaultBrowserPackageNameLPw(null, userId);

}

}

}

}

...

}

// 执行一次GC操作

Runtime.getRuntime().gc();

// 执行删除操作

if (deleteOld) {

synchronized (mInstallLock) {

res.removedInfo.args.doPostDeleteLI(true);

}

}

if (args.observer != null) {

try {

// 调用回调函数通知安装者此次安装的结果

Bundle extras = extrasForInstallResult(res);

args.observer.onPackageInstalled(res.name, res.returnCode,

res.returnMsg, extras);

} catch (RemoteException e) {...}

}

} else {...}

} break;

对POST\_INSTALL消息消息的处理主要就是一些权限处理、发送广播、通知相关应用处理安装结果，然后调用回调函数onPackageInstalled()，这个回调函数是调用installPackage()方法时作为参数传递进来的。

### 总结一下解析应用阶段的工作：

1. 解析apk信息
2. dexopt操作
3. 更新权限信息
4. 完成安装,发送Intent.ACTION\_PACKAGE\_ADDED广播

### 其他相关方法分析

#### getNextCodePath

|  |
| --- |
|  |

类似com.android.browser-1

#### replacePackageLI()

## 其他方法

### PackageParser

parseActivity

## ref

## 装载应用

## ref

<http://www.heqiangfly.com/2016/05/12/android-source-code-analysis-package-manager-installation/>

[https://guolei1130.github.io/2017/01/04/Android应用程序是如何安装的/](https://guolei1130.github.io/2017/01/04/Android%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E6%98%AF%E5%A6%82%E4%BD%95%E5%AE%89%E8%A3%85%E7%9A%84/)  
[Android PackageManager相关源码分析之安装应用](http://www.heqiangfly.com/2016/05/12/android-source-code-analysis-package-manager-installation/)  
[PackageManagerService(Android5.1)深入分析（四）安装应用](http://www.aichengxu.com/android/2506357.htm)  
[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)  
<http://www.jianshu.com/p/21412a697eb0>  
<http://solart.cc/2016/10/30/install_apk/>  
一次测试的信息  
adb logcat -b system

10-03 17:39:21.892 I/PackageManager(20739): init\_copy idx=0: InstallParams{3c107196 file=/data/local/tmp/k.art.debug cid=null}

10-03 17:39:21.896 I/PackageManager(20739): mcs\_bound

10-03 17:39:21.896 I/PackageManager(20739): startCopy UserHandle{-1}: InstallParams{3c107196 file=/data/local/tmp/k.art.debug cid=null}

10-03 17:39:21.923 D/PackageManager(20739): installPackageLI: path=/data/app/vmdl828827845.tmp

10-03 17:39:22.027 D/PackageManager(20739): manifestDigest was not present, but parser got: ManifestDigest {mDigest=fe,da,41,e8,49,d6,cd,e5,10,16,26,df,83,1c,24

,cf,eb,1f,7a,fb,be,27,9f,2d,94,92,9c,ce,f2,6d,78,a1,}

10-03 17:39:22.027 W/PackageManager(20739): Package k.art.debug attempting to redeclare system permission android.permission.WRITE\_SETTINGS; ignoring new declar

ation

10-03 17:39:22.027 D/PackageManager(20739): Renaming /data/app/vmdl828827845.tmp to /data/app/k.art.debug-1

10-03 17:39:22.028 D/PackageManager(20739): installNewPackageLI: Package{5c4549c k.art.debug}

10-03 17:39:22.043 I/PackageManager(20739): Linking native library dir for /data/app/k.art.debug-1

10-03 17:39:22.043 D/PackageManager(20739): Resolved nativeLibraryRoot for k.art.debug to root=/data/app/k.art.debug-1/lib, isa=true

10-03 17:39:23.427 D/PackageManager(20739): New package installed in /data/app/k.art.debug-1

10-03 17:39:23.467 V/PackageManager(20739): BM finishing package install for 4

10-03 17:39:23.468 D/PackageManager(20739): Sending to user 0: act=android.intent.action.PACKAGE\_ADDED dat=package:k.art.debug flg=0x4000000 Bundle[{android.int

ent.extra.UID=10047, android.intent.extra.user\_handle=0}]

10-03 17:39:23.468 D/PackageManager(20739): java.lang.RuntimeException: here

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PackageManagerService.sendPackageBroadcast(PackageManagerService.java:8321)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PackageManagerService$PackageHandler.doHandleMessage(PackageManagerService.java:1066)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PackageManagerService$PackageHandler.handleMessage(PackageManagerService.java:824)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.Handler.dispatchMessage(Handler.java:102)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.Looper.loop(Looper.java:135)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.HandlerThread.run(HandlerThread.java:61)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.ServiceThread.run(ServiceThread.java:46)

10-03 17:39:31.964 I/PackageManager(20739): mcs\_unbind

10-03 17:39:31.965 I/PackageManager(20739): calling disconnectService()

## MyS-自定义服务

* <http://www.cnblogs.com/welhzh/p/5509125.html>
* <http://blog.csdn.net/jinliang_890905/article/details/7320234>
* <http://view.inews.qq.com/a/20160930G078QZ00?refer=share_recomnews>

⬆︎

⟳︎

§

* [framework debug 技巧](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#framework-debug-%E6%8A%80%E5%B7%A7)
  + [安装](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E5%AE%89%E8%A3%85)
  + [Java调试](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#java%E8%B0%83%E8%AF%95)
    - [打断点](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E6%89%93%E6%96%AD%E7%82%B9)
    - [跟踪代码](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E8%B7%9F%E8%B8%AA%E4%BB%A3%E7%A0%81)
    - [打印日志](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E6%89%93%E5%8D%B0%E6%97%A5%E5%BF%97)
  + [native调试](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#native%E8%B0%83%E8%AF%95)
  + [ref](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#ref)
* [pms](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#pms)
  + [文件拷贝阶段](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E6%96%87%E4%BB%B6%E6%8B%B7%E8%B4%9D%E9%98%B6%E6%AE%B5)
    - [installPackage](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#installpackage)
    - [installPackageAsUser](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#installpackageasuser)
    - [doHandleMessage-INIT\_COPY](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#dohandlemessage-init_copy)
    - [InstallParams.handleStartCopy](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#installparamshandlestartcopy)
    - [FileInstallArgs.copyApk()](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#fileinstallargscopyapk)
    - [InstallParams.handleReturnCode()](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#installparamshandlereturncode)
  + [解析应用阶段](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E8%A7%A3%E6%9E%90%E5%BA%94%E7%94%A8%E9%98%B6%E6%AE%B5)
    - [processPendingInstall()](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#processpendinginstall)
    - [installPackageLI()](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#installpackageli)
  + [装载应用](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#%E8%A3%85%E8%BD%BD%E5%BA%94%E7%94%A8)
  + [ref](file:///C:\Users\KEY~1.GUA\AppData\Local\Temp\mpe_preview117912-94856-y0tmon.8azk8lrf6r.html#ref-1)

# 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/

# framework debug 技巧

## 安装

拷贝文件到system/framework/;  
stop;  
start重启system\_server进程

## Java调试

其实整个调试过程非常简单：

* 打断点
* 跟踪代码（Step in/out/over等等）

### 打断点

在正确的进程的合适位置打断点:区别于普通app的进程，framwork的所在进程需要进行源码分析，比如ams是运行在system\_server中的，并且这些进程只能在root的机子调试（模拟器或者root真机）

### 跟踪代码

### 打印日志

-系统一般都会有debug标志，我们修改源码中的标志，打印就好，比如

修改后

-- Log.e("tag",logStr);

## native调试

## ref

* [如何调试Android Framework？](http://weishu.me/2016/05/30/how-to-debug-android-framework/)
* [Android Studio如何调试Framework层的代码？](https://www.zhihu.com/question/37606394)
* [Markdown 编辑器推荐](https://github.com/wizardforcel/markdown-simple-world/blob/master/1.md)

# 属性系统build.prop

## 概述

### build.prop是什么？

在Android中.prop类型的文件是一个属性文件，记录一些系统设置。.prop文件类似/etc中的文件。更简单的来说，就相当于Windows中的系统注册表。所以，build.prop中记录了一些系统的设置属性。这些属性包括系统初始或固定的一些参数属性、功能的开放等。通过调整/增加这些参数可以达到较调系统性能偏重点和附加功能开启的作用。

### build.prop是怎样生成的？

首先需要确定build.prop在编译过程中是由哪个Makefile、.mk或编译脚本控制的。熟悉Android编译过程的可能很容易定位到./build/core/Makefile这个文件。如果不熟悉Android编译过程可以通过查看Android整编的log。在./build/core/Makefile中，可以看到：

# build.prop

INSTALLED\_BUILD\_PROP\_TARGET := $(TARGET\_OUT)/build.prop

ALL\_DEFAULT\_INSTALLED\_MODULES += $(INSTALLED\_BUILD\_PROP\_TARGET)

ADDITIONAL\_BUILD\_PROPERTIES := \

$(call collapse-pairs, $(ADDITIONAL\_BUILD\_PROPERTIES))

ADDITIONAL\_BUILD\_PROPERTIES := $(call uniq-pairs-by-first-component, \

$(ADDITIONAL\_BUILD\_PROPERTIES),=)

ifdef TARGET\_SYSTEM\_PROP

system\_prop\_file := $(TARGET\_SYSTEM\_PROP)

else

system\_prop\_file := $(wildcard $(TARGET\_DEVICE\_DIR)/system.prop)

endif

通过Makefile文件，可以得到，build.prop的来源有：

* ./build/tools/buildinfo.sh：将buildinfo.sh的内容，直接echo到build.prop中
* system.prop：把默认的system.prop内容追加到build.prop中。如，./device/jrdcom/Alto5TF/system.prop
* ADDITIONAL\_BUILD\_PROPERTIES：把ADDITIONAL\_BUILD\_PROPERTIES中的属性追加到build.prop
* ODM/OED厂商的定制：一些厂商一般都会有自己的一些额外定制

### build.prop读/写

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Code-java | 主要通过SystemProperties中提供的get相关的方法；者直接用类**Build**.java。 | get(String key, String def)  **Build.XXX** | frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java调用jni  \frameworks\base\core\jni\android\_os\_SystemProperties.cpp |  |
| C/C++层 |  | property\_get("xxxx", value, "0"); |  |  |
| Adb命令 |  | adb shell getprop xxxx | adb shell setprop xxxx  临时修改，不重启系统 |  |
| 文件修改 | /system/build.prop | Pull/push要修改build.prop权限。可以改为644，重启生效。 |  |  |
|  |  |  |  |  |

### 特殊生命周期

如果属性名称以“ro.”开头，那么这个属性被视为只读属性。一旦设置，属性值不能改变。

如果属性名称以“persist.”开头，当设置这个属性时，其值也将写入**/data/property、。**

## 参数详解

build.prop中参数包括

1. Dalvik虚拟机相关参数
2. 系统版本、定义相关参数、
3. 基本性能相关参数、
4. 基本耗电相关参数、
5. 扩展性能较调及附加功能开启

### 类型分类

属性值中ro开头的表示read-only，这种属性值一旦被赋值一次之后，都不能被改写；

而以persist开头的属性值表示持久化的数据，会将其保存到/data/property目录下。

其他类型的属性除了位于属性文件中的属性值之外，手机一旦关机这些属性值都会丢失，不能持久存在

系统属性根据不同的应用类型，分为不可变型，持久型，网络型，启动和停止服务等。

 　　特别属性：

　　属性名称以“ro.”开头，那么这个属性被视为只读属性。一旦设置，属性值不能改变。

　　属性名称以“persist.”开头，当设置这个属性时，其值也将写入/data/property。

　　属性名称以“net.”开头，当设置这个属性时，“net.change”属性将会自动设置，以加入到最后修改的属性名。（这是很巧妙的。 netresolve模块的使用这个属性来追踪在net.\*属性上的任何变化。）

　　属性“ ctrl.start ”和“ ctrl.stop ”是用来启动和停止服务。每一项服务必须在/init.rc中定义.系统启动时，与init守护进程将解析init.rc和启动属性服务。一旦收到设置“ ctrl.start ”属性的请求，属性服务将使用该属性值作为服务名找到该服务，启动该服务。这项服务的启动结果将会放入“ init.svc.<服务名>“属性中 。客户端应用程序可以轮询那个属性值，以确定结果。

### Dalvik虚拟机相关参数

**dalvik.vm.** **heapstartsize**

本参数控制Dalvik虚拟机在启动一个应用程序之后为其分配的初始堆栈大小，可填写的值为**2m~48m**。例如：dalvik.vm.heapstartsize=8m，就表示应用程序启动后为其分配的初始堆栈大小为8兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程度，和第一次使用应用程序时的流畅程序。这个值越大，系统消耗RAM则越快，但是应用程序打开后的反应也越快。值越小，系统的RAM剩余则越多，但是程序在启动后会很卡。建议值是8m，既可以保持140M左右的RAM，程序的反应速度也会大幅度提高。

**dalvik.vm.heapsize**

本参数控制Dalvik虚拟机给一个应用程序分配的最大堆栈量，可填写的值为12m~48m。例如：dalvik.vm.heapsize=48m，就表示应用程序在任意时刻内可以使用的最大堆栈大小为48兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程序，和程序在运行一段时间后的反应速度。**这个值越大，系统消耗RAM则越快，但是程序会运行的非常稳定**，尤其是游戏和视频程序的内容加载速度可以大幅度提升。值越小，系统的RAM剩余则越多，但是程序会很卡，尤其是游戏在切换场景Loading的时候会花费很多的时间。若应用程序需要使用超过这个值的内存时，将会触发系统的垃圾收集器，系统和程序就会卡顿。建议值是40~40m。

**dalvik.vm.lockprof.threshold**

本参数控制Dalvik虚拟机调试记录程序内部锁资源争夺的阈值，默认值是500。多用于程序的数据统计，对性能较调意义不大。

**dalvik.vm.stack-trace-file**

本参数控制Dalvik虚拟机的堆栈记录调试文件。用于系统调试，一般用户对其调整无意义,默认为/data/anr/traces.txt，这是开发者分析anr的重要文件。

**dalvik.vm.execution-mode**

本参数控制Dalvik虚拟机的程序执行机制。可填写的值有"int:portable"、"int:fast"和"int:jit"。int:portable表示以兼容模式运行(脚本翻译模式)，此模式下程序的兼容性最高，但其执行效率最低(程序优化度依赖于dalvik虚拟机版本)。官方默认此模式。int:fast表示以快速自优化模式运行(脚本翻译和预优化混合)，此模式下程序的兼容性很高，执行效率也比较高。因为此时dalvik虚拟机允许程序使用自己的预定义优化模式和代码(包括C/C  /汇编代码)。推荐使用。**int:jit表示以Just-In-Time**模式运行(JIT模式)，此模式下程序的兼容性最差，**但程序一旦加载后其运行效率最高(与C/C  直接编写的程序效率无异)**，因为在此模式下dalvik虚拟机会预先将Java程序翻译成针对机器平台的本地语言(Native)，同时完全允许代码中的所有预优化和代码，**允许所有不安全的非托管代码，同时不严谨的程序如果运行在JIT模式可能会造成内存泄露**。但要注意，**很多Dalvik虚拟机并不支持此模式**(如官方2.2)。

**dalvik.vm.dexopt-flags**

本参数控制Dalvik虚拟机的程序代码校验和优化。可填写的值有m、v和o。m为标准选项，可以是m=y或m=n，若m=y则启用不安全代码的校验和托管代码的优化，兼容性和安全性最高，推荐使用；

v为校验选项，可与o并存，可以是v=a或v=n。若v=a则表示校验所有代码，v=n则关闭代码的校验

o为优化选项，可与v并存。可以是o=v或o=a。若o=v则表示优化以校验过的代码，o=a则表示优化所有代码。例如：dalvik.vm.dexopt-flags=m=ydalvik.vm.dexopt-flags=v=n,o=v注意，这个参数只会影响到安装APK之后或初次使用APK时生成dex文件时有效。若整个系统(包括应用程序)为odex化，则无意义。

**dalvik.vm.verify-bytecode**

本参数控制Dalvik虚拟机是否验证应用程序的可执行代码。可以与上一个参数配合使用。可填写的值为true和false。其具体意义与dalvik.vm.dexopt-flags的v=n一模一样。但可以与dalvik.vm.dexopt-flags配合使用以取得更好的效果。例如：dalvik.vm.dexopt-flags=v=n,o=vdalvik.vm.verify-bytecode=false这样可以令后来安装的apk文件可以被优化而不被检验。dalvik.vm.checkjni，本参数控制Dalvik虚拟机在调用外部jni链接库的时候是否对其做安全性检验。可填写的值为true和false。此参数会覆盖ro.kernel.android.checkjni。若值为true，会增加程序的兼容性和稳定性，但也会增加其加载和执行的时间。推荐为false。

**dalvik.vm.deadlock-predict**

本参数控制Dalvik虚拟机对程序死锁预测处理。可填写的值有off、warn和err。off表示关闭死锁预测功能(默认设置)。warn表示在继续程序运行的同时只记录该死锁预测(如果为真死锁就会出现程序假死现象，然后等N久出现关闭)。err表示预测到死锁时马上弹出FC。注意：有些Dalvik虚拟机版本并不支持此参数。

超级急速流畅型

dalvik.vm.startheapsize=16m

dalvik.vm.heapsize=48m

dalvik.vm.execution-mode=int:jit

dalvik.vm.dexopt-flags=v=n,o=v

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

常用稳定加流畅型：

dalvik.vm.startheapsize=8m

dalvik.vm.heapsize=40m

dalvik.vm.execution-mode=int:fast

dalvik.vm.dexopt-flags=m=y

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

超级稳定大内存型：

dalvik.vm.startheapsize=4m

dalvik.vm.heapsize=30m

dalvik.vm.execution-mode=int:portable

dalvik.vm.dexopt-flags=v=a,o=v

dalvik.vm.verify-bytecode=true

dalvik.vm.checkjni=true

**(以荣耀一代为例，RAM 512M，以下是像我一样追求速度的童鞋的推荐参数)**  
**dalvik.vm.startheapsize=16m  
dalvik.vm.heapsize=64m（RAM大的童鞋这个参数可以再大一些）  
dalvik.vm.execution-mode=int:jit  
dalvik.vm.dexopt-flags=v=n,o=v  
dalvik.vm.verify-bytecode=false  
dalvik.vm.checkjni=false**

### 系统版本、定义等参数

**ro.build.id**

本参数定义了系统的版本ID。为系统内部使用，OTA时作为粗略版本比较。更改后可避免OTA提示，但可能会引起预装程序(如Blur)的稳定性。

**ro.build.display.id**

本参数定义了设置中显示的系统版本号。主要用于设置中显式出现可读版本，一般用于个性化定制和第三方应用程序对系统版本的判断(如魔趣设置)。更改后可自定义版本显示，但某些第三方应用程序会出现错误(如魔趣设置无法实现机器保修查询)。

**ro.build.version.incremental**

本参数定义了系统的升级字。主要用于系统OTA精确版本比对，同时与ro.build.description和ro.build.fingerprint相匹配。更改后可以免OTA提示(如避免Miui的升级提示和Blur的升级提示)。

#### ro.product.model

本参数定义了机器的型号字符串。主要用于机器型号**显式定义(如系统设置中的手机型号和Blur、Google设置向导中的机型等)**。更改后可自定义手机型号名称，供用户界面显示。

*/\*\* The end-user-visible name for the end product. \*/***public static final** String ***MODEL*** = *getString*(**"ro.product.model"**);

如小米的

C:\Users\key.guan>adb shell

sagit:/ $ getprop | grep product

[ro.boot.product.region]: [cn]

[ro.build.product]: [sagit]

[ro.product.board]: [msm8998]

[ro.product.brand]: [Xiaomi]

[ro.product.cert]: [MCE16]

[ro.product.cpu.abi]: [arm64-v8a]

[ro.product.cpu.abilist]: [arm64-v8a,armeabi-v7a,armeabi]

[ro.product.cpu.abilist32]: [armeabi-v7a,armeabi]

[ro.product.cpu.abilist64]: [arm64-v8a]

[ro.product.cuptsm]: [XIAOMI|ESE|02|01]

[ro.product.device]: [sagit]

[ro.product.first\_api\_level]: [25]

[ro.product.locale]: [zh-CN]

[ro.product.manufacturer]: [Xiaomi]

[**ro.product.model**]: [MI 6]

[ro.product.name]: [sagit]

*/\*\* The name of the overall product. \*/***public static final** String ***PRODUCT*** = *getString*(**"ro.product.name"**);  
  
*/\*\* The name of the industrial design. \*/***public static final** String ***DEVICE*** = *getString*(**"ro.product.device"**);  
  
*/\*\* The name of the underlying board, like "goldfish". \*/***public static final** String ***BOARD*** = *getString*(**"ro.product.board"**);

# ro.build.product is obsolete; use ro.product.device （旧代码ro.build.product，使用代码ro.product.device）

**ro.product.locale.language**

本参数定义了系统的初始(默认)语言。此处注意是语言，如中文是zh，英文是en。更改后改变系统**初次启动**时的语言设置。

**ro.product.locale.region**

本参数定义了系统的初始(默认)区域。此处注意是区域，如中国大陆为CN，台湾为TW，美国为US。更改后改变系统初次启动时的区域设置。

**ro.build.description和ro.build.fingerprint**

ROM的编译综合说明。其中包含了平台硬件、Android版本、源代码分支和标签、OTA详细版本等。其中的OTA部分，例如：umts\_jordan\_china-user 2.3.6 4.5.3-109\_DPP-141323416413release-keys将此数字与ro.build.version.incremental一同更改可避免OTA升级提醒(如Miui和Blur等)。

### 基本性能相关参数

**windowsmgr.max\_events\_per\_sec**

本参数定义了Android系统的窗体事件管理器在单位时间内可以处理的最大事件数量。通过更改本参数可以获得非常明显的丝滑流畅体验。可填写的值范围为“大于0的正整数”，官方默认为60。建议150、200、260、300这几个值。当此值变大时，系统触控平滑度明显提高，但对应的CPU使用率也会升高，最终的结果就是电池续航能力下降。以我个人的经验来说，此值取到240左右时在系统设置中滑动可以得到接近WP7的流畅和平滑度。

**ro.min\_pointer\_dur**

本参数定义了两次触摸之间的最短时间间隔，单位是毫秒。默认值为25，推荐值是10。通过调整此参数可以提高系统触控的灵敏度或稳定度。当此值越大时，触控越稳定。此值越小，触控越灵敏。

**mot.proximity.delay**

本参数定义了手机光纤感应器的抖动消除时间，单位是毫秒。默认值是500，推荐值是250。通过调整此参数可以提高在通话结束后屏幕点亮的速度。当此值越大时，通话结束后屏幕点亮所需要的时间越长，但在通话过程中如果手机意外瞬间离开脸部也不会点亮屏幕，可防止通话过程中的误操作(比方说通话时不小心手机移动了一下，屏幕就会点亮，此时如果脸部触碰到了屏幕就会对通话造成影响)。此值越小，则当手机离开脸部或装入口袋后会立即点亮或关闭屏幕。

**mot.proximity.distance**

本参数定义了手机屏幕上的两个触摸点之间的最短距离，若距离小于此值则认为是一个触摸点，单位是像素。默认值是60，推荐值是100。为什么推荐100呢？因为Defy的屏幕分辨率为480x854，也就是说横向有480个像素点，对应上去也就相当于是横向并排允许4个触摸点，平均一个手指一个点，这样在类似于杀西瓜等游戏中可以提升游戏操作。

**ro.kernel.android.checkjni**

本参数定义了Dalvik虚拟机在执行程序的时候是否要做Jni链接库的检查工作。详细见Dalvik参数属性期。若考虑稳定性可使用true，若需要性能可使用false。注意：此参数会被Dalvik参数覆盖。

**ro.media.enc.jpeg.quality**

本参数定义了JPEG图像编码器所使用的质量因子，可填写的值为1~100，默认为80，推荐为100。想照出更好的照片吗？想让照片的大小轻松上M吗？那就使用100吧。

**debug.sf.hw**

本参数定义了系统是否启用GPU来渲染程序的UI，默认为0，推荐为1。但要注意，如果此值为1，在某些应用程序中可能会出现显示错乱的现象(极少见)。

**persist.sys.use\_dithering**

本参数定义了系统渲染器对图像的缩放是否启用抖动技术。可填写的值为0或1。当开启抖动后，图像的显示(指背景、解锁等的图像，并非图库、相机那些的)会很柔和，但会增加CPU负载，最终导致ROM卡顿。

**persist.sys.purgeable\_assets**

本参数定义了系统是否可以清除暂时不用的数据以释放更多的RAM。可填写的值为0或1。当值为1时，系统会定期清理不用的数据以释放更多的RAM，同时作为代价就是下次启动程序或游戏加载数据会变慢。

**video.accelerate.hw**

本参数定义了系统是否对视频启用**硬件加速**功能。这里的视频指代屏幕上显示的东西，不仅仅是“电影视频”。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

**debug.performance.tuning**

本参数定义了系统是否针对性能做较调。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

ro.HOME\_APP\_ADJ

ro.FOREGROUND\_APP\_ADJ

ro.VISIBLE\_APP\_ADJ

ro.PERCEPTIBLE\_APP\_ADJ

ro.HEAVY\_WEIGHT\_APP\_ADJ

ro.SECONDARY\_SERVER\_ADJ

ro.BACKUP\_APP\_ADJ

ro.HIDDEN\_APP\_MIN\_ADJ

ro.EMPTY\_APP\_ADJ

以上参数可填写的值为整数。这里只给出值的规律

0代表降低进程的优先级且驻留内存，1代表驻留内存，4代表缓存较多的内存，15代表尽量缓存内存。也就是说内存缓存器是按照ADJ从大到小来进行缓存的。大家可根据自系统中自己对各种应用程序的要求进行更改。

以下给出一个经典用例：

ro.FOREGROUND\_APP\_ADJ=0 前台程序驻留内存(不缓存)

ro.VISIBLE\_APP\_ADJ=1         可见的程序驻留内存(不缓存)

ro.PERCEPTIBLE\_APP\_ADJ=2    缓存的RAM多一些

ro.HOME\_APP\_ADJ=3         桌面程序，缓存的RAM稍多一些

ro.HEAVY\_WEIGHT\_APP\_ADJ=4        缓存的RAM再多一些

ro.SECONDARY\_SERVER\_ADJ=5        缓存的RAM再再多一些

ro.BACKUP\_APP\_ADJ=6       缓存的RAM再再再多一些

ro.HIDDEN\_APP\_MIN\_ADJ=7         隐藏的程序，根据程序的类型进行内存管理，最低为缓存的RAM再再再再多一些，最高就是直接缓存内存。

ro.EMPTY\_APP\_ADJ=15         已经退出的程序，直接缓存内存

ro.FOREGROUND\_APP\_MEM

ro.VISIBLE\_APP\_MEM

ro.PERCEPTIBLE\_APP\_MEM

ro.HEAVY\_WEIGHT\_APP\_MEM

ro.SECONDARY\_SERVER\_MEM

ro.BACKUP\_APP\_MEM

ro.HOME\_APP\_MEM

ro.HIDDEN\_APP\_MEM

ro.CONTENT\_PROVIDER\_MEM

ro.EMPTY\_APP\_MEM

以上参数定义了各种类型的应用程序在内存缓冲的大小，单位是页下面给出一个经典用例：

ro.FOREGROUND\_APP\_MEM=1280

ro.VISIBLE\_APP\_MEM=2560

ro.PERCEPTIBLE\_APP\_MEM=3840

ro.HEAVY\_WEIGHT\_APP\_MEM=6400

ro.SECONDARY\_SERVER\_MEM=7680

ro.BACKUP\_APP\_MEM=8960

ro.HOME\_APP\_MEM=5120

ro.HIDDEN\_APP\_MEM=12800

ro.CONTENT\_PROVIDER\_MEM=15360

ro.EMPTY\_APP\_MEM=20480

### 基本耗电相关参数

**wifi.supplicant\_scan\_interval**

本参数定义了Wifi扫描已保存节电的时间间隔。当点亮屏幕或打开Wifi时，系统会不停的扫描环境中是否存在已经保存的Wifi节点，当发现后则进行连接，而这个参数控制了每次扫描的时间间隔。单位是秒。取值范围是正整数。**官方默认为45，推荐180**。

**ro.mot.battmanager.wifictrl**

本参数定义了电源管理模块对Wifi的控制。默认为0。当此值为1时可以明显节电，但有时Wifi会出现不稳定的情况(不是所有ROM都如此)。

**ro.mot.deep.sleep.supported**

本参数定义了是否开启摩托的“休眠”模式。取值为true或false。当值为true时，在电源菜单中会出现“休眠”模式。此模式类似于电脑的睡眠，即将CPU等部件的电源全部关闭，只为**RAM供电**以保存休眠前的系统状态。耗电量比完全关机多一些，但可以做到**瞬间开机**。仅在官方ROM有效。

**pm.sleep\_mode**

本参数定义了系统待机时的睡眠深度，在所有Android系统上有效。取值范围是0~4，对应解释如下。0：强制关闭除RAM之外的所有部件，此状态下最省电。Defy几乎可以纯待机3~4个礼拜。但是此模式与“休眠”类似，一旦进入之后射频也会关闭，手机的2G/3G信号也就断了(语音和数据)。

1：让ARM进入中断触发的待机(超低功耗)模式。与模式0相比，本模式下射频不会关闭，而ARM可以通过软件(闹铃)和硬件(来电)中断来唤醒，因此耗电方面远大于模式0，Defy可以纯待机7天(不安装任何软件)。非常建议使用。  
2：将所有应用程序挂起到后台。与模式1相比，本模式下硬件几乎不参与多少节电，耗电自然比模式1多很多。当应用程序被挂起后，CPU的负载会大幅度降低，从而节电。此模式下Defy纯待机5天。  
3：将CPU的频率和电压降至最低，低到主频只有几十MHz的水平，而此时CPU接受外部中断(通过中断来恢复频率和电压)。与模式2相比，本模式下CPU通过降频和降压参与了节电，因此本模式的耗电比模式2多了一点。Defy纯待机约4~5天。本模式也是官方ROM和官方CM系统的默认值。  
4：CPU接受外部中断。与上述4个模式相比，此模式下几乎不做任何节电，只是关闭了屏幕和按键背光而已。Defy纯待机约2天。将上述5个模式的节电按照星级来分就是，模式0和1为5颗星，模式2和3为3颗星，模式4为1颗星。综上所属就是，模式0和模式1基本一样，是靠完全关闭几乎所有硬件部件来进行节电，省电效果最佳。模式2和模式3是靠调节CPU频率来进行节电。个人强烈推荐采用pm.sleep\_mode=1，即省电又稳定。如果想用模式0但又担心基带射频的同学可以继续往下看，解决办法在下面。

**ro.ril.disable.power.collapse**

本参数定义了是否禁止射频参与电源休眠。取值是0或1。这个参数的使用需要与上一个参数相匹配(我看到很多ROM中的这两个参数都是不匹配的，**最终造成的效果就是点亮屏幕后信号存在问题**)。当本参数为1的时候即射频永远打开，为0的时候根据上一个参数pm.sleep\_mode来判断是否关闭射频。永远打开射频必然费电，但是如果射频关闭，那手机就没信号了。那么当pm.sleep\_mode=0的时候，上面说过，此时待机会关闭几乎所有硬件部件，包括射频。而此时如果ro.ril.disable.power.collapse=1，就会保持射频的开启(即使进入休眠模式也一样)。这样即使待机，手机也有信号。但是又存在这样一个现象，在有些ROM中pm.sleep\_mode=0会带来更多的问题， **如睡死、亮屏后Wifi打不开、蓝牙打不开等**。因此建议同学们可以先尝试一下pm.sleep\_mode=0和ro.ril.disable.power.collapse=1组合使用，看看是否有bug，如果没有那自然使用此种模式，毕竟最省电了(极端省电)。对于稳定与省电兼得，可使用如下组合：pm.sleep\_mode=1ro.ril.disable.power.collapse=0这样射频在pm.sleep\_mode=1下不会被关闭，而进入休眠模式后射频会关闭。

### 参数定制建议

|  |  |  |  |
| --- | --- | --- | --- |
|  | 原生定义 | 现状 | 修改意见 |
| 系统设置-》关于设备显示-》  产品名 | **ro.product.model：**The end-user-visible name for the end product. | 硬编码或者，没有 | 类似公司华为手机，实现属性ro.product.model |
| 打印在产品硬件印刷名：同上 | 同上 | 直接是ma；GL300A | 同上 |
| 上传到服务器产品名字：同上 | 同上 | 同上 | 同上 |
| 对外宣传的名字 | **无** |  | * 旧产品保留，新产品改为，加一个ro. \*\*.product. [**media**](http://dict.cn/media)   在系统设置-》关于设备显示-》增加一个显示 |
| App与系统之间代码标志名字 | **ro.product.device ：***The name of the industrial design.* | 使用**ro.product.name** | 旧产品保留，新产品改为原生定义**ro.product.device** |
|  |  |  |  |

ro.build.product，使用代码ro.product.device

/\*\* \*/

public static final String MODEL = getString("ro.product.model");

*/\*\* The name of the overall product. \*/***public static final** String ***PRODUCT*** = *getString*(**"ro.product.name"**);  
  
*/\*\* The name of the industrial design. \*/***public static final** String ***DEVICE*** = *getString*(**"ro.product.device"**);  
  
*/\*\* The name of the underlying board, like "goldfish". \*/***public static final** String ***BOARD*** = *getString*(**"ro.product.board"**);

## 源码分析

**JAVA**：\frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java：get()—》native\_get

**JIN**:\frameworks\base\core\jni\android\_os\_SystemProperties.cpp: SystemProperties\_getSS-> property\_get

**操作在\bionic\libc\bionic\system\_properties.c中：**

|  |
| --- |
| int \_\_system\_property\_get(const char \*name, char \*value)  {    //数据已经存储在内存中\_\_system\_property\_area\_\_ 等待读取完返回  const prop\_info \*pi = \_\_system\_property\_find(name);  return \_\_system\_property\_read(pi, 0, value);  } |

进程启动后数据已经将系统属性数据读取到相应的**共享内存**中，保存在全局变量\_\_system\_property\_area\_\_，在

### **设置属性**异步socket通信

|  |
| --- |
| int \_\_system\_property\_set(const char \*key, const char \*value)  {     msg.cmd = PROP\_MSG\_SETPROP;     strlcpy(msg.name, key, sizeof msg.name);     strlcpy(msg.value, value, sizeof msg.value);     err = send\_prop\_msg(&msg);  }  static int send\_prop\_msg(prop\_msg \*msg)  {    //sokcet 通信 /dev/socket/property\_service     s = socket(AF\_LOCAL, SOCK\_STREAM, 0);     connect(s, (struct sockaddr \*) &addr, alen)     send(s, msg, sizeof(prop\_msg), 0)     close(s);  } |

通过socket向property\_service发送消息，property\_service运行在哪里呢？

### Property Service创建服务端socket

**init进程启动监听过程中：**，**Property Service 是运行在init守护进程中。**

|  |
| --- |
| \system\core\init\Init.c  int main(int argc, char \*\*argv)  {     //加入到action queue队列  queue\_builtin\_action(property\_service\_init\_action, "property\_service\_init");   for(;;)  //执行action queue队列  //接收通过socket向property service 发送的数据;         nr = poll(ufds, fd\_count, timeout);         ……         handle\_property\_set\_fd();  }  static int property\_service\_init\_action(int nargs, char \*\*args)  {     start\_property\_service();  }  \system\core\init\property\_service.c：  void start\_property\_service(void)  {    //加载属性配置文件      load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_BUILD);     load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_DEFAULT);     load\_properties\_from\_file(PROP\_PATH\_LOCAL\_OVERRIDE);     load\_persistent\_properties();     //创建socket资源 并绑定     fd = create\_socket(PROP\_SERVICE\_NAME, SOCK\_STREAM, 0666, 0, 0);    //监听     listen(fd, 8);  } |



#define PROP\_PATH\_SYSTEM\_BUILD     "/system/build.prop"  
#define PROP\_PATH\_SYSTEM\_DEFAULT   "/system/default.prop"  
#define PROP\_PATH\_LOCAL\_OVERRIDE   "/data/local.prop"

Persist 顾名思义是从持久化文件读写的

### Property Service 监听socket处理

|  |
| --- |
| //\system\core\init\property\_service.c：  void handle\_property\_set\_fd()  {    //等待建立通信     s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)    //获取套接字相关信息 uid gid     getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size);    //接收属性设置请求消息     recv(s, &msg, sizeof(msg), 0);    //处理消息     switch(msg.cmd) {    case PROP\_MSG\_SETPROP:        //通过设置系统属性  处理ctl.开头消息         if(memcmp(msg.name,"ctl.",4) == 0)         {            //权限检测             if (check\_control\_perms(msg.value, cr.uid, cr.gid))             {                 handle\_control\_message((char\*) msg.name + 4, (char\*) msg.value);             }         } else         {            //更改系统属性值             if (check\_perms(msg.name, cr.uid, cr.gid))             {                 property\_set((char\*) msg.name, (char\*) msg.value);             }         }    break;     }     close(s);  } |

### 设置属性

|  |
| --- |
| **int** property\_set(**const char** \*name, **const char** \*value)  {  prop\_info \*pi;  **int** ret;   size\_t namelen = strlen(name);  size\_t valuelen = strlen(value);   **if** (!is\_legal\_property\_name(name, namelen)) **return** -1;  **if** (valuelen >= PROP\_VALUE\_MAX) **return** -1;   pi = (prop\_info\*) \_\_system\_property\_find(name);   **if**(pi != 0) {  */\* ro.\* properties may NEVER be modified once set \*/* **if**(!strncmp(name, **"ro."**, 3)) **return** -1;   \_\_system\_property\_update(pi, value, valuelen);  } **else** {  ret = \_\_system\_property\_add(name, namelen, value, valuelen);  **if** (ret < 0) {  ERROR(**"Failed to set '%s'='%s'\n"**, name, value);  **return** ret;  }  }  */\* If name starts with "net." treat as a DNS property. \*/* **if** (strncmp(**"net."**, name, strlen(**"net."**)) == 0) {  **if** (strcmp(**"net.change"**, name) == 0) {  **return** 0;  }  */\*  \* The 'net.change' property is a special property used track when any  \* 'net.\*' property name is updated. It is \_ONLY\_ updated here. Its value  \* contains the last updated 'net.\*' property.  \*/* property\_set(**"net.change"**, name);  } **else if** (persistent\_properties\_loaded &&  strncmp(**"persist."**, name, strlen(**"persist."**)) == 0) {  */\*  \* Don't write properties to disk until after we have read all default properties  \* to prevent them from being overwritten by default values.  \*/* write\_persistent\_property(name, value);  } **else if** (strcmp(**"selinux.reload\_policy"**, name) == 0 &&  strcmp(**"1"**, value) == 0) {  selinux\_reload\_policy();  }  property\_changed(name, value);  **return** 0;  }   **void** handle\_property\_set\_fd()  {  prop\_msg msg;  **int** s;  **int** r;  **int** res;  struct ucred cr;  struct sockaddr\_un addr;  socklen\_t addr\_size = sizeof(addr);  socklen\_t cr\_size = sizeof(cr);  **char** \* source\_ctx = NULL;  struct pollfd ufds[1];  **const int** timeout\_ms = 2 \* 1000; */\* Default 2 sec timeout for caller to send property. \*/* **int** nr;   **if** ((s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)) < 0) {  **return**;  }   */\* Check socket options here \*/* **if** (getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size) < 0) {  close(s);  ERROR(**"Unable to receive socket options\n"**);  **return**;  }   ufds[0].fd = s;  ufds[0].events = POLLIN;  ufds[0].revents = 0;  nr = TEMP\_FAILURE\_RETRY(poll(ufds, 1, timeout\_ms));  **if** (nr == 0) {  ERROR(**"sys\_prop: timeout waiting for uid=%d to send property message.\n"**, cr.uid);  close(s);  **return**;  } **else if** (nr < 0) {  ERROR(**"sys\_prop: error waiting for uid=%d to send property message. err=%d %s\n"**, cr.uid, errno, strerror(errno));  close(s);  **return**;  }   r = TEMP\_FAILURE\_RETRY(recv(s, &msg, sizeof(msg), MSG\_DONTWAIT));  **if**(r != sizeof(prop\_msg)) {  ERROR(**"sys\_prop: mis-match msg size received: %d expected: %zu errno: %d\n"**,  r, sizeof(prop\_msg), errno);  close(s);  **return**;  }   **switch**(msg.cmd) {  **case** PROP\_MSG\_SETPROP:  msg.name[PROP\_NAME\_MAX-1] = 0;  msg.value[PROP\_VALUE\_MAX-1] = 0;   **if** (!is\_legal\_property\_name(msg.name, strlen(msg.name))) {  ERROR(**"sys\_prop: illegal property name. Got: \"%s\"\n"**, msg.name);  close(s);  **return**;  }   getpeercon(s, &source\_ctx);   **if**(memcmp(msg.name,**"ctl."**,4) == 0) {  *// Keep the old close-socket-early behavior when handling  // ctl.\* properties.* close(s);  **if** (check\_control\_mac\_perms(msg.value, source\_ctx)) {  handle\_control\_message((**char**\*) msg.name + 4, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n"**,  msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);  }  } **else** {  **if** (check\_perms(msg.name, source\_ctx)) {  property\_set((**char**\*) msg.name, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: permission denied uid:%d name:%s\n"**,  cr.uid, msg.name);  }   *// Note: bionic's property client code assumes that the  // property server will not close the socket until \*AFTER\*  // the property is written to memory.* close(s);  }  freecon(source\_ctx);  **break**;  **default**:  close(s);  **break**;  }  } |

**修改系统属性权限表：**

property\_perms[] = {

   { "net.dns",          AID\_RADIO,    0 },

   { "net.",             AID\_SYSTEM,   0 },

   { "dev.",             AID\_SYSTEM,   0 },

   { "runtime.",         AID\_SYSTEM,   0 },

   { "sys.",             AID\_SYSTEM,   0 },

   { "service.",         AID\_SYSTEM,   0 },

   { "persist.sys.",     AID\_SYSTEM,   0 },

   { "persist.service.", AID\_SYSTEM,   0 },

   ……

    { NULL, 0, 0 }

};

一般property启动应该加在init.<your hardware>.rc而不是直接init.rc里。下面是一个init.rc里的例子：

### 开关服务

可以通过设置系统属性 改变服务的执行状态 start/stop：如果想要应用有权限启动/关闭某Native Service：需要具有system/root权限，找到对应应用uid gid，将应用名称加入到control\_perms列表中。

连着前面就是ctr.start和ctr.stop系统属性：用来启动和停止服务的。

|  |
| --- |
| void handle\_control\_message(const char \*msg, const char \*arg)  {    if (!strcmp(msg,"start")) {         msg\_start(arg);     } else if (!strcmp(msg,"stop")) {         msg\_stop(arg);     } else if (!strcmp(msg,"restart")) {         msg\_stop(arg);         msg\_start(arg);     }  }static void msg\_start(const char \*name)  {     service\_start(svc, args);  }void service\_start(struct service \*svc, const char \*dynamic\_args){    //创建进程启动服务     pid = fork();     execve(svc->args[0], (char\*\*) svc->args, (char\*\*) ENV);     //修改服务的系统属性 执行状态     notify\_service\_state(svc->name, "running");  } |

例如：　// start boot animation

　　　　property\_set("ctl.start", "bootanim");

定义开机启动的服务:**启动服务的时候会判断：**

static void service\_start\_if\_not\_disabled(struct service \*svc)

{        //判断是否启动

       if (!(svc->flags & SVC\_DISABLED)) {

           service\_start(svc, NULL);

       }

}

在init.rc中表明服务是否在开机时启动：

service adbd /sbin/adbd

　　　　class core

　　　　disabled //不自动启动

### 共享属性原理

初始化共享内存：属性是存储在共享内存中的,而要在使用共享内存之前呢,又必须要先初始化共享内存.初始化之后,肯定要加载那些存储属性的文件等等.这些都是init进程中完成的.

init.c 的main函数中调用了property\_init函数

init. property\_init（）

init\_property\_area

|  |  |
| --- | --- |
| 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | static int init\_property\_area(void) {  if (property\_area\_inited)  return -1;   if(\_\_system\_property\_area\_init())//会以读写方式打开/dev/\_\_properties\_\_  return -1;   if(init\_workspace(&pa\_workspace, 0))//这里面会以只读方式再次打开/dev/\_\_properties\_\_  return -1;   fcntl(pa\_workspace.fd, F\_SETFD, FD\_CLOEXEC);   property\_area\_inited = 1;//表明共享内存已经被初始化了  return 0; } |

system\_property\_area\_init用来初始化共享内存.

|  |  |
| --- | --- |
| 1 2 3 4 | int \_\_system\_property\_area\_init() {  return map\_prop\_area\_rw(); } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 | static int map\_prop\_area\_rw() {  /\* dev is a tmpfs that we can use to carve a shared workspace  \* out of, so let's do that...  \*/  //proerty\_file是 /dev/\_\_properties\_\_,要注意,这里是以可读可写方式打开的,这个文件描述符是给property service使用的  const int fd = open(property\_filename,  O\_RDWR | O\_CREAT | O\_NOFOLLOW | O\_CLOEXEC | O\_EXCL, 0444);   if (fd < 0) {  if (errno == EACCES) {  /\* for consistency with the case where the process has already  \* mapped the page in and segfaults when trying to write to it  \*/  abort();  }  return -1;  }   // TODO: Is this really required ? Does android run on any kernels that  // don't support O\_CLOEXEC ?  const int ret = fcntl(fd, F\_SETFD, FD\_CLOEXEC);  if (ret < 0) {  close(fd);  return -1;  }   if (ftruncate(fd, PA\_SIZE) < 0) {  close(fd);  return -1;  }   pa\_size = PA\_SIZE;  pa\_data\_size = pa\_size - sizeof(prop\_area);  compat\_mode = false;   //----------------pa\_size大小为128K  void \*const memory\_area = mmap(NULL, pa\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);  if (memory\_area == MAP\_FAILED) {  close(fd);  return -1;  }   prop\_area \*pa = new(memory\_area) prop\_area(PROP\_AREA\_MAGIC, PROP\_AREA\_VERSION);   /\* plug into the lib property services \*/  \_\_system\_property\_area\_\_ = pa;//共享内存的起始地址存储在这个全局变量上   close(fd);  return 0; } |

我们可以看到，在init进程的main()函数里，打开了一个设备文件“/dev/\_\_**properties**”，并把它设定为128KB大小，接着调用mmap()将这块内存映射到init进程空间了。这个内存的首地址被记录在**system\_property\_area**全局变量里，以后每添加或修改一个属性，都会基于这个**system\_property\_area**变量来计算位置。

在来看下面的函数:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | static int init\_workspace(workspace \*w, size\_t size)//size传入的参数值是0 {  void \*data;  int fd = open(PROP\_FILENAME, O\_RDONLY | O\_NOFOLLOW);//以只读的方式再次打开/dev/\_\_properties\_\_,  if (fd < 0)  return -1;   w->size = size;  w->fd = fd;  return 0; } |

打开的句柄记录在pa\_workspace.fd处，以后每当init进程调用service\_start()时，会执行下面的代码

|  |  |
| --- | --- |
| 1 2 3 4 5 | if (properties\_inited()) {  get\_property\_workspace(&fd, &sz);  sprintf(tmp, "%d,%d", dup(fd), sz);  add\_environment("ANDROID\_PROPERTY\_WORKSPACE", tmp);  } |

说白了就是把 pa\_workspace.fd 的句柄记入一个名叫“ ANDROID\_PROPERTY\_WORKSPACE ”的环境变量去,另外size似乎没什么用,一直是0.

|  |  |
| --- | --- |
| 1 2 | root@generic\_x86\_64:/ # echo $ANDROID\_PROPERTY\_WORKSPACE  8,0 |

存入环境变量的作用是其他进程可以很方便拿到文件描述符fd,利用这个fd就可以读取属性值了.

为什么要两次open那个/dev/**properties**文件呢？是这样的：第一次open的句柄，最终是给属性服务自己用的，所以需要有读写权限；而第二次open的句柄，会被记入pa\_workspace.fd，并在合适时机添加进环境变量，**供其他进程使用，因此只能具有读取权限。**

## 属性系统设计

　 Property Service运行在init进程中，开机从属性文件中加载到共享内存中；设置系统属性通过socket与Property Service通信。

　　Property Consumer进程将存储系统属性值的共享内存，加载到当前进程虚拟空间中，实现对系统属性值的读取。

Property Setter进程修改系统属性，通过socket向Property Service发送消息，更改系统属性值。

属性系统设计的关键就是：跨进程共享内存的实现。

[详情看](http://blog.csdn.net/hecong_kit/article/details/46008895)

## 小结

属性初始化的入口点是property\_init ，在system/core/init/property\_service.c中定义。它的主要工作是申请32k共享内存，其中前1k是属性区的头，后面31k可以存247个属性（受前1k头的限制）。property\_init初始化完property以后，加载/default.prop的属性定义。   
  
其它的系统属性（build.prop, local.prop,…）在start\_property\_service中加载。加载完属性服务创建一个socket和其他进程通信（设置或读取属性）。   
  
Init进程poll属性的socket，等待和处理属性请求。如果有请求到来，则调用handle\_property\_set\_fd来处理这个请求。在这个函数里，首先检查请求者的uid/gid看看是否有权限，如果有权限则调property\_service.c中的property\_set函数。   
  
在property\_set函数中，它先查找就没有这个属性，如果找到，更改属性。如果找不到，则添加新属性。更改时还会判断是不是“ro”属性，如果是，则不能更改。如果是persist的话还会写到/data/property/<name>中。   
  
**最后它会调property\_changed，把事件挂到队列里**，如果有人注册这个属性的话（比如init.rc中on property:ro.kernel.qemu=1），最终会调它的会调函数。   
  
property名字长度限制是32字节，值的限制是92字节。不知道是google怎么想的 — 一般都是名字比值长得多！比如[dalvik.vm.heapgrowthlimit]: [48m]

## Ref

[Android 5.1 property属性系统分析上篇](http://www.iloveandroid.net/2015/09/26/Android_property_1/)

# 系统升级

升级方式：SD卡、usb、OTA升级（以OTA升级方式梳理整个升级流程）

在setting中触发升级按钮，检测prop中的服务器路径上的版本是否比本地版本高，如果低于或者等于本地版本，则提示已是最新版本，如果高于本地版本，则提示有新版本是否进行升级，点击立即升级。

bootCommand（）此函数在main system重启之前做准备工作。首先创建/cache/recovery/目录，删除该目录下的command和log（可能不存在）在sqlite数据库中的备份，然后将上一步的arg命令写入到/cache/recovery/command文件中，下一步就真正重启了。

源码路径：frameworks/base/core/java/android/os/RecoverySystem.java

BCB：bootloader control block

arg就是进入recovery模式后，recovery服务需要进行的操作

RecoverySystem类中的**installPackage(Context context,FilepackageFile)**函数根据传进来的包文件获取包的绝对路径filename，然后将其拼成**arg=“--update\_package=”+filename**，它最终会被传到BCB中，然后被传到**bootCommand(context,arg)。**

升级包在此处也被传进来了

mService是SystemUpdateService实例，跟踪发现SystemUpdateService.Java中的rebootAndUpdate函数中新建了一个线程，此此线程调用了**RecoverySystem.installPackage(mContext,mFile)**

按钮监听事件，调用mService.rebootAndUpdate（new  File(mFile)）。此处的file即为升级包

跟踪file来源，mService.getInstallFile() 获取到OTA在线下载的文件路径

注：从Bootloader开始如果没有组合键按下，就从MISC分区读取BCB块的command域，在主系统时已经将“boot-recovery”写入。然后进入recovery模式开始启动，与正常的启动系统类似，也有启动内核，然后启动文件系统，在进入文件系统后会执行/init，init的配置文件就是/init.rc。（bootable/recovery/etc/init.rc）

主要工作：

1. 设置环境变量
2. 建立etc连接
3. 新建目录，备用
4. 挂载/tmp为内存文件系统tmpfs
5. 启动recovery（/sbin/recovery）服务
6. 启动adbd服务（用于调试）

（\*其中最重要的就是启动recovery服务）

源码路径：bionic/libc/unistd/reboot.c

此函数将”recovery”传过去，之后将”boot-recovery”写入到BCB数据块的command域中，重启之后就知道进入recover

进入reboot：

调用pm.reboot(“recovery”)，跟踪发现其实是一个系统调用\_\_reboot(LINUX\_REBOOT\_MAGIC1,LINUX\_REBOOT\_MAGIC2,mode,NULL)，前两个参数代表组合键，mode代表传过来的”recovery”

## Recovery流程

Recovery的内容有：恢复出厂设置、OTA升级、使能/关闭加密文件系统

（下图是recovery整个流程，很详细



1、ui\_init():在recovery的过程中主要用于显示一个背景图片（正在安装或者安装失败）和一个进度条（用于显示进度）,另外还启动了两个线程，一个用于处理进度条的显示（progress\_thread），另一个用于响应用户的按键（input\_thread）。

2、 get\_arg()：这个函数主要做了上图中get\_arg()往右往下直到parse arg/v的工作。对照着流程看。

      ①get\_bootloader\_message()：主要工作是根据分区的文件格式类型（mtd或emmc）从MISC分区中读取BCB数据块到一个临时的变量中。

      ②然后开始判断Recovery服务是否有带命令行的参数（/sbin/recovery，根据现有的逻辑是没有的），若没有就从BCB中读取recovery域。如果读取失败则从/cache/recovery/command中读取然后。这样这个BCB的临时变量中的recovery域就被更新了。在将这个BCB的临时变量写回真实的BCB之前，又更新的这个BCB临时变量的command域为“boot-recovery”。这样做的目的是如果在升级失败（比如升级还未结束就断电了）时，系统在重启之后还会进入Recovery模式，直到升级完成。

      ③在这个BCB临时变量的各个域都更新完成后使用set\_bootloader\_message()写回到真正的BCB块中。

                这个过程可以用一个简单的图来概括，这样更清晰：



  3、parserargc/argv：解析我们获得参数。注册所解析的命令（register\_update\_command），在下面的操作中会根据这一步解析的值进行一步步的判断，然后进行相应的操作。

   4、 if(update\_package)：判断update\_package是否有值，若有就表示需要升级更新包，此时就会调用install\_package()（即图中红色的第二个阶段）。在这一步中将要完成安装实际的升级包。

     5、 if(wipe\_data/wipe\_cache)：这一步判断实际是两步，在源码中是先判断是否擦除data分区（用户数据部分）的，然后再判断是否擦除cache分区。值得注意的是在擦除data分区的时候必须连带擦除cache分区。在只擦除cache分区的情形下可以不擦除data分区。

     6、maybe\_install\_firmware\_update()：如果升级包中包含/radio/hboot firmware的更新，则会调用这个函数。查看源码发现，在注释中（OTA INSTALL）有这一个流程。但是main函数中并没有显示调用这个函数。目前尚未发现到底是在什么地方处理。但是其流程还是向上面的图示一样。即，① 先向BCB中写入“boot-recovery”和“—wipe\_cache”之后将cache分区格式化，然后将firmware image 写入原始的cache分区中。②将命令“update-radio/hboot”和“—wipe\_cache”写入BCB中，然后开始重新安装firmware并刷新firmware。③之后又会进入图示中的末尾，即finish\_recovery()。

     7、 prompt\_and\_wait()：这个函数是在一个判断中被调用的。其意义是如果安装失败（update.zip包错误或验证签名失败），则等待用户的输入处理（如通过组合键reboot等）。

     8、 finish\_recovery()：这是Recovery关闭并进入Main System的必经之路。其大体流程如下：



 ① 将intent（字符串）的内容作为参数传进finish\_recovery中。如果有intent需要告知Main System，则将其写入/cache/recovery/intent中。这个intent的作用尚不知有何用。

       ② 将内存文件系统中的Recovery服务的日志（/tmp/recovery.log）拷贝到cache（/cache/recovery/log）分区中，以便告知重启后的Main System发生过什么。

       ③ 擦除MISC分区中的BCB数据块的内容，以便系统重启后不在进入Recovery模式而是进入更新后的主系统。

        ④ 删除/cache/recovery/command文件。这一步也是很重要的，因为重启后Bootloader会自动检索这个文件，如果未删除的话又会进入Recovery模式。原理在上面已经讲的很清楚了。

9、 reboot()：这是一个系统调用。在这一步Recovery完成其服务重启并进入Main System。这次重启和在主系统中重启进入Recovery模式调用的函数是一样的，但是其方向是不一样的。所以参数也就不一样。查看源码发现，其重启模式是RB\_AUTOBOOT。这是一个系统的宏。

## Install\_package详解

源码路径：/bootable/recovery/install.c



①ensure\_path\_mount()：先判断所传的update.zip包路径所在的分区是否已经挂载。如果没有则先挂载。

②load\_keys()：加载公钥源文件，路径位于/res/keys。这个文件在Recovery镜像的根文件系统中。

 ③verify\_file()：对升级包update.zip包进行签名验证。

 ④mzOpenZipArchive()：打开升级包，并将相关的信息拷贝到一个临时的ZipArchinve变量中。这一步并未对我们的update.zip包解压。

  ⑤try\_update\_binary()：在这个函数中才是对我们的update.zip升级的地方。这个函数一开始先根据我们上一步获得的zip包信息，以及升级包的绝对路径将update\_binary文件拷贝到内存文件系统的/tmp/update\_binary中。以便后面使用。

   ⑥pipe()：创建管道，用于下面的子进程和父进程之间的通信。

   ⑦fork()：创建子进程。其中的子进程主要负责执行binary（execv(binary,args)，即执行我们的安装命令脚本），父进程负责接受子进程发送的命令去更新ui显示（显示当前的进度）。子父进程间通信依靠管道。

   ⑧其中，在创建子进程后，父进程有两个作用。一是通过管道接受子进程发送的命令来更新UI显示。二是等待子进程退出并返回INSTALL SUCCESS。其中子进程在解析执行安装脚本的同时所发送的命令有以下几种：

     progress  <frac> <secs>：根据第二个参数secs（秒）来设置进度条。

     set\_progress  <frac>：直接设置进度条，frac取值在0.0到0.1之间。

     firmware <”hboot”|”radio”><filename>：升级firmware时使用，在API  V3中不再使用。

      ui\_print <string>：在屏幕上显示字符串，即打印更新过程。

      execv(binary,args)的作用就是去执行binary程序，这个程序的实质就是去解析update.zip包中的updater-script脚本中的命令并执行。由此，Recovery服务就进入了实际安装update.zip包的过程。

|  |
| --- |
|  |

# 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)

# PowerManagerService

第5章 深入理解 PowerManagerService

<http://wiki.jikexueyuan.com/project/deep-android-v2/powermanagerservice.html>

# [Android5.1--PowerManagerService电源管理](http://blog.csdn.net/kitty_landon/article/details/47107045)

<http://blog.csdn.net/kitty_landon/article/details/47107045>

# [Android5.1 PowerManagerService深入分析](http://blog.csdn.net/kc58236582/article/details/45537749)

<http://blog.csdn.net/kc58236582/article/details/45537749>

、、、、、、、、、、、、、、、、、、、、、、、、、

[Android5.1AlarmManagerService深入分析（Android4.4补充）](http://blog.csdn.net/kc58236582/article/details/45666709)

<http://blog.csdn.net/kc58236582/article/details/45666709>

# Android之AlarmManagerService源码分析

<http://www.procedurego.com/article/128389.html>

<http://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

<https://github.com/clarkehe/Android/wiki/Android%E4%BC%91%E7%9C%A0%E6%9C%BA%E5%88%B6>

休眠机制

# [AlarmManager 实现闹钟的基本功能](http://www.cnblogs.com/rainly/p/3437627.html)

## Ref

# Path

编译生成的apk路径：out/target/product/ maguro/system/app

签名文件目录：/build/target/product/security

Android系统应用的源码路径：/packages/apps目录或其子目录中

源码目录/package/apps/PackageInstaller

# Framework开 push失败

  替换  
编译后用生成的包替换手机上 /system/framework/ 中相应的包

$ adb push services.jar /system/framework/

4．   使新的包生效

a)          方法 1:  
重启手机后生效

b)         方法 2:  
杀死 system\_server 进程，使系统重新启动，这种方法速度快  
$ ps  
找到 system\_server 对应的 pid, 假设它为 1219  
$ kill 1219

-rw-r--r-- root root 10174464 2017-09-17 13:09 system@framework@boot.art

-rw-r--r-- root root 50889136 2017-09-17 13:09 system@framework@boot.oat

[解决framework.jar push 到/system/framework/中不生效](http://blog.csdn.net/wustli/article/details/52331148)

修改Android framework源码后将framework.jar给push进机子后为什么不起作用

<http://bbs.csdn.net/topics/391963822?page=1>

# [修改替换/system/framework/framework.jar后重启手机为何没有效果？](http://www.cnblogs.com/zhhd/p/5790448.html)

<http://www.cnblogs.com/zhhd/p/5790448.html>

# Android ART模式，不重新烧录system.img的情况下如何修改framework和app？

https://www.zhihu.com/question/40425628

# 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/)

# Task

Settings.Global

Settings源码解读

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

# Android编译原理

## 引言

先看下面几条指令，相信编译过Android源码的人都再熟悉不过的。

source setenv.sh //初始化编译环境，包括后面的lunch和make指令

lunch //指定此次编译的目标设备以及编译类型

make -j12 //开始编译，默认为编译整个系统，其中-j12代表的是编译的job数量为12。

跟大家分享一下在研究上述几条指令含义的过程中，深入了解到的Android Build(编译)系统。

## 编译命令

准备好编译环境后，编译Android源码的第一步是 source build/envsetup.sh，其中source命令就是用于运行shell脚本命令，功能等价于”.”，因此该命令也等价于. build/envsetup.sh。在文件envsetup.sh声明了当前会话终端可用的命令，这里需要注意的是当前会话终端，也就意味着每次新打开一个终端都必须再一次执行这些指令。起初并不理解为什么新开的终端不能直接执行make指令，到这里总算明白了。

#### 2.1 代码编译

| **编译指令** | **解释** |
| --- | --- |
| m | 在源码树的根目录执行编译 |
| mm | 编译当前路径下所有模块，但不包含依赖 |
| mmm [module\_path] | 编译指定路径下所有模块，但不包含依赖 |
| mma | 编译当前路径下所有模块，且包含依赖 |
| mmma [module\_path] | 编译指定路径下所有模块，且包含依赖 |
| make [module\_name] | 无参数，则表示编译整个Android代码 |

下面列举部分模块的编译指令：

| **模块** | **make命令** | **mmm命令** |
| --- | --- | --- |
| init | make init | mmm system/core/init |
| zygote | make app\_process | mmm frameworks/base/cmds/app\_process |
| system\_server | make services | mmm frameworks/base/services |
| java framework | make framework | mmm frameworks/base  如果编译失败：需要再编译一次framework-res  [务必保证系统为debug版本，否则不生效的！！](https://www.zhihu.com/question/40425628) |
| framework资源 | make framework-res | mmm frameworks/base/core/res |
| jni framework | make libandroid\_runtime | mmm frameworks/base/core/jni |
| binder | make libbinder | mmm frameworks/native/libs/binder |

上述mmm命令同样适用于mm/mma/mmma，编译系统采用的是增量编译，只会编译发生变化的目标文件。当需要重新编译所有的相关模块，则需要编译命令后增加参数-B，比如make -B [module\_name]，或者 mm -B [module\_path]。

**Tips:**

* 对于m、mm、mmm、mma、mmma这些命令的实现都是通过make方式来完成的。
* mmm/mm编译的效率很高，而make/mma/mmma编译较缓慢；
* make/mma/mmma编译时会把所有的依赖模块一同编译，但mmm/mm不会;
* 建议：首次编译时采用make/mma/mmma编译；当依赖模块已经编译过的情况，则使用mmm/mm编译。

## DEBUG 技巧

貌似了

## 常见的坑

编译frameworks/base的mk不是递归编译关系，其子模块需要单独mmm，比如services模块，修改了pm之类，必须mmm frameworks/base/services

编译framwork，有些依赖不好处理

## REF

[理解Android编译命令](http://gityuan.com/2016/03/19/android-build/)

# ROOT

# ADB源码分析

## ADB APK安装

## adb install

安装 APK 命令可以用adb install [-lrtsd] <file>或者adb install-multiple [-lrtsdp] <file...>，adb install-multiple表示批量安装。  
参数介绍：

* -l：锁定该程序
* -r：可以覆盖已有的应用，保留数据和缓存文件
* -t：允许测试该应用
* -s：安装在SD卡中
* -d：允许降低版本安装
* -p：部分应用程序安装

# 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)