Android Source Code Guide

|  |
| --- |
| **Xu Feng** |
| **2014/8/14** |

# Android系统启动过程

<https://www.ibm.com/developerworks/cn/linux/l-linuxboot/>

Linux系统启动

BIOS 固定地址引导程序

--> stage 1 512B加载程序

--> stage2 GRUB

--> 解压内核并运行 Head.s: startup\_32

--> start\_kernel

--> setup\_arch/rest\_init/trap\_init/init\_IRQ

--> kernel\_thread --> init

main.c: init

--> do\_basic\_setup

--> sock\_init/do\_initcalls

init.c: paging\_init

-->pagetable\_init

　 physmem.c: init\_maps

startup\_32

---> decompress\_kernel

---> startup\_32 (/arch/i386/kernel/head\_32.S) <进程 0>

---> i386\_start\_kernel

---> start\_kernel

---> rest\_init

---> kernel\_thread <进程 1>

---> do\_fork

---> kernel\_init

---> sys\_access

---> init\_post

---> run\_init\_process(ramdisk\_execute\_command); = '/init'

---> init.rc / init.xx.rc

---> service adbd /sbin/adbd

---> service servicemanager /system/bin/servicemanager

---> service ril-daemon /system/bin/rild

---> service surfaceflinger /system/bin/surfaceflinger

---> service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

---> run\_init\_process(execute\_command);

---> run\_init\_process("/sbin/init");

---> run\_init\_process("/etc/init");

---> run\_init\_process("/bin/init");

---> run\_init\_process("/bin/sh");

---> cpu\_idle

Main app\_main.c

---> AndroidRuntime::start

---> startVm

---> JNI\_CreateJavaVM

---> dvmCreateJNIEnv (malloc JNIEnvExt and insert it to vm->envList)

---> dvmStartup

---> startReg

---> register\_jni\_procs

---> array[i].mProc(env) = gRegJNI

---> env->CallStaticVoidMethod = ZygoteInit -> Main

ZygoteInit -> Main Zygote启动

---> registerZygoteSocket

---> new LocalServerSocket

---> preload

---> preloadClasses

---> preloadResources

---> preloadOpenGL

---> gc

---> startSystemServer 启动SystemServer

---> Zygote.forkSystemServer

---> nativeForkSystemServer 通过JNI调用native代码

---> Dalvik\_dalvik\_system\_Zygote\_forkSystemServer

---> forkAndSpecializeCommon

---> fork()

---> SystemServer.main

---> nativeInit

---> ServerThread.initAndLoop

---> new DisplayManagerService

---> new TelephonyRegistry

---> new PackageManagerService

---> new ServiceManager

---> new BatteryService

---> new VibratorService

---> new WindowManagerService

---> new WifiService 启动WifiService

---> new ConnectivityService 启动ConnectivityService

---> handleSystemServerProcess

---> RuntimeInit.zygoteInit

---> commonInit

---> nativeZygoteInit

---> applicationInit

---> runSelectLoop

---> closeServerSocket

# Android应用启动过程



应用程序进程启动

ActivityManagerService. startProcessLocked

---> Process.start 将启动进程的指令通过socket发送给zygote进程

---> startViaZygote

ZygoteConnection.runOnce

---> Zygote.forkAndSpecialize

--- > nativeForkAndSpecialize = Dalvik\_dalvik\_system\_Zygote\_forkAndSpecialize

--- > forkAndSpecializeCommon

--- > fork

--- > setSELinuxContext 设置安全上下文

--- > selinux\_android\_setcontext

--- > setcon

--- > setprocattrcon

# Binder机制

参见PPT Introduction of Binder.ppt

# AudioFlinger

* 1. 启动

Main\_mediaserver.cpp -> Main 启动

---> AudioFlinger::instantiate()

---> BinderService. publish

---> sm->addService 将binder对象注册到serviceManager

---> AudioPolicyService::instantiate() 启动AudioPolicyService

--> AudioPolicyService::AudioPolicyService

--> hw\_get\_module

--> audio\_policy\_dev\_open

--> mpAudioPolicyDev->create\_audio\_policy = create\_qcom\_ap

--> create\_legacy\_ap/create\_qcom\_ap

--> createAudioPolicyManager

--> AudioPolicyManager::AudioPolicyManager

--> AudioPolicyManagerBase::AudioPolicyManagerBase

* 1. 功能实现
     1. 类图



* + 1. 音频设备接口类型

static const char \* const audio\_interfaces[] = {

AUDIO\_HARDWARE\_MODULE\_ID\_PRIMARY,

AUDIO\_HARDWARE\_MODULE\_ID\_A2DP,

AUDIO\_HARDWARE\_MODULE\_ID\_USB,

};

* + 1. 音频设备打开

AudioPolicyManagerBase::AudioPolicyManagerBase

---> loadAudioPolicyConfig(AUDIO\_POLICY\_VENDOR\_CONFIG\_FILE) –

加载配置文件"/vendor/etc/audio\_policy.conf"

---> loadHwModules

---> loadHwModule

---> mHwModules.add

---> loadAudioPolicyConfig(AUDIO\_POLICY\_CONFIG\_FILE) –

如果上一步不成功，加载配置文件"/system/etc/audio\_policy.conf"

---> mpClientInterface->loadHwModule(mHwModules[i]->mName)

---> AudioPolicyCompatClient::loadHwModule

---> mServiceOps->load\_hw\_module = aps\_load\_hw\_module

---> af->loadHwModule

---> AudioFlinger::loadHwModule

---> AudioFlinger::loadHwModule\_l

---> load\_audio\_interface

---> load 加载音频设备动态库文件audio.a2dp.default.so/audio.primary.default.so等

---> mAudioHwDevs.add

* 1. 打开Output（输出）通道

AudioPolicyManagerBase::AudioPolicyManagerBase

---> loadAudioPolicyConfig(AUDIO\_POLICY\_VENDOR\_CONFIG\_FILE) –

---> loadAudioPolicyConfig(AUDIO\_POLICY\_CONFIG\_FILE) –

---> mpClientInterface->loadHwModule(mHwModules[i]->mName)

---> AudioPolicyCompatClient::loadHwModule

---> mpClientInterface->openOutput = AudioPolicyCompatClient::openOutput

---> mServiceOps->open\_output\_on\_module = aps\_open\_output\_on\_module

---> af->openOutput

---> AudioFlinger:: findSuitableHwDev\_l

---> hwDevHal->open\_output\_stream = adev\_open\_output\_stream 打开输出流

---> new AudioStreamOut

---> new MixerThread

---> mPlaybackThreads.add 添加播放线程

---> thread->audioConfigChanged\_l

* 1. PlaybackThread线程主循环

PlaybackThread::onFirstRef

---> run

---> createThreadEtc

---> Thread. \_threadLoop

---> self->threadLoop = PlaybackThread::threadLoop

---> threadLoop\_standby = AudioFlinger::MixerThread::threadLoop\_standby

---> PlaybackThread::threadLoop\_standby

---> mOutput->stream->common.standby

---> prepareTracks\_l = MixerThread::prepareTracks\_l

---> track->cblk 取得数据块

---> 计算minFrames

---> 若数据ready，mAudioMixer->setBufferProvider

---> mAudioMixer->enable/mAudioMixer->setParameter 设置音量等

---> 若数据未ready，则将track加入tracksToRemove中

---> threadLoop\_mix = MixerThread::threadLoop\_mix

---> mAudioMixer->process

---> mState.hook = process\_\_validate

---> state->hook = process\_\_OneTrack16BitsStereoNoResampling/

process\_\_genericNoResampling/process\_\_genericResampling

---> threadLoop\_write = AudioFlinger::MixerThread::threadLoop\_write

---> PlaybackThread::threadLoop\_write

---> NBAIO: mNormalSink->write

---> mOutput->stream->write

---> threadLoop\_removeTracks

* 1. 向AudioMixer中添加Track

New AudioTrack

---> AudioTrack. set

---> AudioSystem::getOutput

---> new AudioTrackThread

---> mAudioTrackThread->run

---> AudioTrack.createTrack\_l

---> audioFlinger->createTrack

---> checkPlaybackThread\_l

---> thread->createTrack\_l = PlaybackThread::createTrack\_l

---> new Track

---> mTracks.add

---> new TrackHandle

---> track->getCblk

---> thread->audioConfigChanged\_l

# MultiMedia框架

# Activity框架之实现

# Activity创建过程

ActivityThread.main启动

---> thread.attach

---> ActivityManagerService. attachApplication

---> attachApplicationLocked

---> ActivityStackSupervisor.attachApplicationLocked

---> realStartActivityLocked [ActivityManagerService]

---> scheduleLaunchActivity [回到应用的ActivityThread]

---> handleLaunchActivity

---> performLaunchActivity

---> mInstrumentation.newActivity [Activity is created]

---> createBaseContextForActivity [ContextImpl is created]

---> activity.attach

---> mInstrumentation.callActivityOnCreate [Call onCreate]

---> activity.performStart [Call onStart]

---> handleResumeActivity [Call onResume]

# Activity窗口对象创建过程

接上图

---> activity.attach

---> PolicyManager.makeNewWindow

---> sPolicy.makeNewWindow = Policy. makeNewWindow

---> new PhoneWindow [Create PhoneWindow]

---> mWindow.setCallback

---> mWindow.setSoftInputMode

---> mWindow.setUiOptions

---> mWindow.setWindowManager

---> new WindowManagerImpl

# Activity视图DecorView创建过程

ViewRoot创建

---> scheduleLaunchActivity

---> handleLaunchActivity

---> performLaunchActivity

---> callActivityOnCreate [onCreate]

---> setContentView

---> getWindow().setContentView

---> PhoneWindow.setContentView

---> PhoneWindow .installDecor

---> generateDecor

---> generateLayout

---> handleResumeActivity

---> performResumeActivity

---> r.window.getDecorView

---> wm.addView = WindowManagerImpl.addView

---> WindowManagerGlobal.addView

---> new ViewRootImpl

---> mViews.add(view)

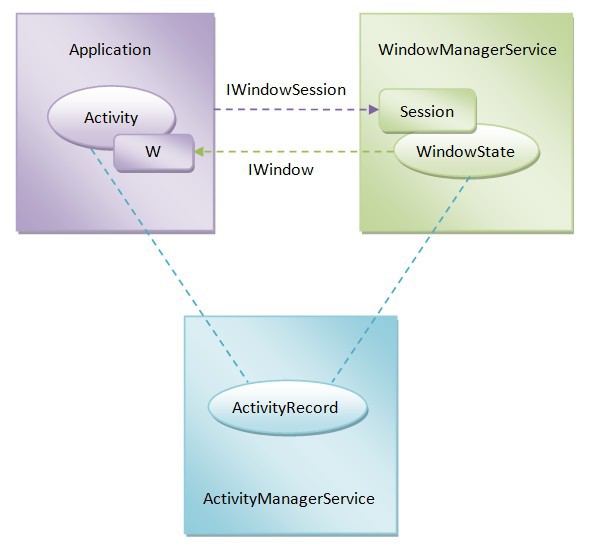
---> mRoots.add(root)

---> mParams.add(wparams)

---> root.setView

# Activity与WindowManagerService连接过程

Activity组件在其窗口对象和视图对象创建完成之后，就会请求与WindowManagerService建立一个连接，即请求WindowManagerService为其增加一个WindowState对象，用来描述它的窗口状态。当一个应用程序进程在启动第一个Activity组件的时候，它便会打开一个到 WindowManagerService服务的连接，这个连接以应用程序进程从WindowManagerService服务处获得一个实现了 IWindowSession接口的Session代理对象来标志。从WindowManagerService服务到Activity组件的连接是以 Activity组件为单位来进行的。在应用程序进程这一侧，每一个Activity组件都关联一个实现了IWindow接口的W对象，这个W对象在 Activity组件的视图对象创建完成之后，就会通过前面所获得一个Session代理对象来传递给WindowManagerService服务，而 WindowManagerService服务接收到这个W对象之后，就会在内部创建一个WindowState对象来描述与该W对象所关联的 Activity组件的窗口状态，并且以后就通过这个W对象来控制对应的Activity组件的窗口状态。



Session创建

---> handleResumeActivity

---> performResumeActivity

---> r.window.getDecorView

---> wm.addView = WindowManagerImpl.addView

---> WindowManagerGlobal.addView

---> new ViewRootImpl

---> WindowManagerGlobal.getWindowSession

---> windowManager.openSession

---> new Session

---> mService.mInputMethodManager.addClient

AppWindowToken对象的创建过程

---> ActivityStack.startActivityLocked

---> insertTaskAtTop [Put current task on the top]

---> mWindowManager.addAppToken = WindowManagerService. addAppToken

---> new AppWindowToken

---> new InputApplicationHandle

---> new AppWindowAnimator

---> task.addAppToken

---> mTokenMap.put

WindowState创建

---> handleResumeActivity

---> performResumeActivity

---> r.window.getDecorView

---> wm.addView = WindowManagerImpl.addView

---> WindowManagerGlobal.addView

---> new ViewRootImpl

---> ViewRootImpl.setView

---> requestLayout

---> mWindowSession.addToDisplay

---> mService.addWindow

---> new WindowState

---> WindowState.attach

---> mSession.windowAddedLocked

---> new SurfaceSession

---> nativeCreate

---> new SurfaceComposerClient

---> onFirstRef

---> sm->createConnection

---> SurfaceFlinger::createConnection

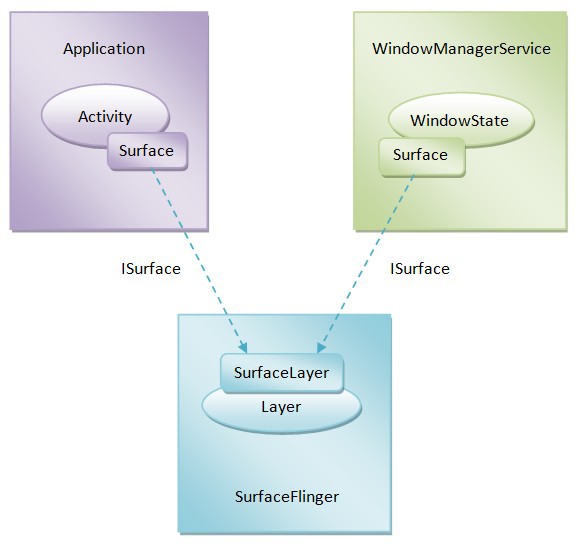
---> new Client

---> mService.mSessions.add

---> mWindowMap.put



# Activity窗口绘制Surface创建过程



Surface的创建

---> handleResumeActivity

---> performResumeActivity

---> r.window.getDecorView

---> wm.addView = WindowManagerImpl.addView

---> WindowManagerGlobal.addView

---> new ViewRootImpl

---> new Surface

---> new CompatibleCanvas

---> ViewRootImpl.setView

---> ViewRootImpl .requestLayout

---> checkThread [只有创建View的线程（UI线程）可以访问]

---> ViewRootImpl. scheduleTraversals

---> TraversalRunnable. run

---> doTraversal

---> performTraversals

---> relayoutWindow

---> mWindowSession.relayout

---> WindowManagerService.relayoutWindow

---> winAnimator.createSurfaceLocked

---> **new SurfaceControl**

---> nativeCreate

---> client->createSurface

---> mClient->createSurface

---> Client. createSurface

---> new MessageCreateLayer

---> flinger->createLayer

---> **SurfaceFlinger .createNormalLayer**

**---> createNormalLayer**

**---> new Layer**

---> new SurfaceControl

---> outSurface.copyFrom

---> nativeCreateFromSurfaceControl

---> SurfaceControl .getSurface

---> new Surface

---> mService.mSessions.add

---> mWindowMap.put

# Activity窗口绘制过程

绘制

---> performTraversals

---> **relayoutWindow** [Create Surface]

---> **performMeasure** [Measure Views]

---> View. measure

---> FrameLayout .onMeasure

---> measureChildWithMargins

---> **performLayout**

---> View.layout

---> setFrame

---> invalidate

---> ViewRootImpl.invalidateChild

---> invalidateChildInParent

---> scheduleTraversals

---> onLayout

---> layoutChildren

---> child.layout [do layout for all children view]

---> **performDraw**

---> ViewRootImpl.draw

---> attachInfo.mHardwareRenderer.draw [如果支持，启用硬件加速绘制]

---> drawSoftware [若不支持应将加速，采用软件绘制]

---> mSurface.lockCanvas

---> nativeLockCanvas

---> surface->lock [获取buffer，并返回Canvas]

---> Surface::connect

---> dequeueBuffer

---> mGraphicBufferProducer->dequeueBuffer

---> backBuffer->lock [返回后端缓冲区地址]

---> DecorView.draw

---> FrameLayout.draw

---> View.draw

---> background.draw

---> onDraw

---> dispatchDraw

---> onDrawScrollBars

---> surface.unlockCanvasAndPost

---> nativeUnlockCanvasAndPost

---> surface->unlockAndPost

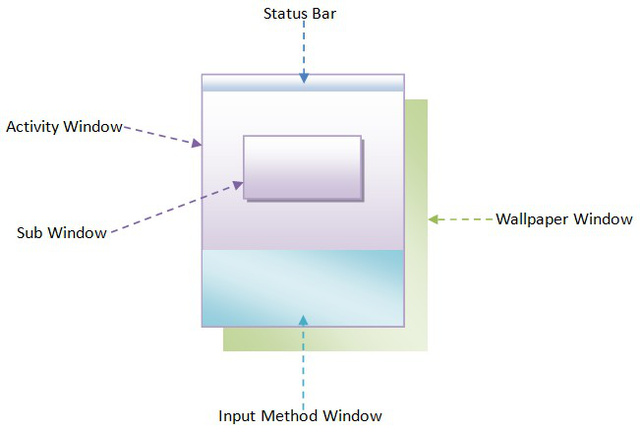
---> mLockedBuffer->unlock

---> queueBuffer

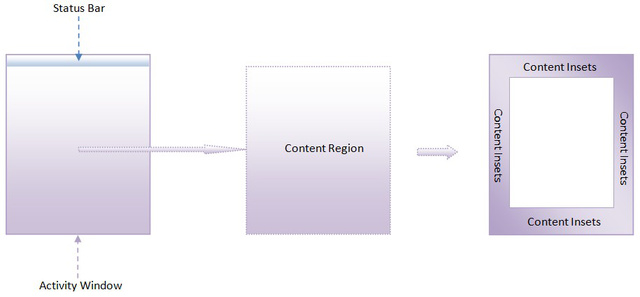
---> mGraphicBufferProducer->queueBuffer

---> output.deflate

# WindowManagerService之实现



# 计算Activity窗口大小



窗口大小计算

---> performTraversals

---> **relayoutWindow** [Create Surface]

---> mWindowSession.relayout

---> WindowManagerService.relayoutWindow [计算窗口大小，通过out参数返回]

---> wm.addView = WindowManagerImpl.addView

---> WindowManagerGlobal.addView

---> new ViewRootImpl

---> ViewRootImpl.setView

---> requestLayout

---> mWindowSession.addToDisplay

---> mService.addWindow

---> new WindowState

---> WindowState.attach

---> mSession.windowAddedLocked

---> new SurfaceSession

---> nativeCreate

---> new SurfaceComposerClient

---> onFirstRef

---> sm->createConnection

---> SurfaceFlinger::createConnection

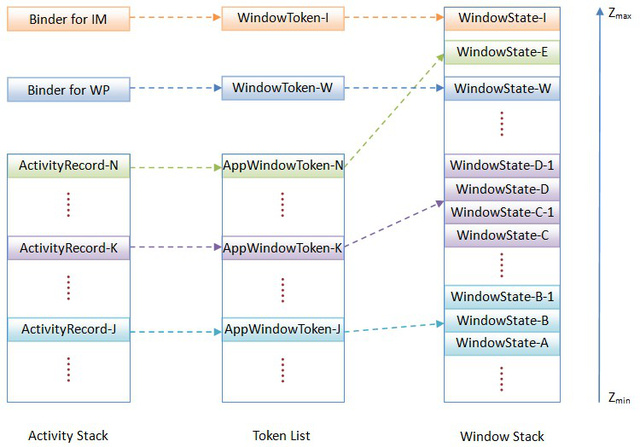
---> new Client

---> mService.mSessions.add

---> mWindowMap.put

# 窗口组织方式及实现

在Window管理服务WindowManagerService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Window管理服务 WindowManagerService中所对应的AppWindowToken对象为令牌的。从抽象的角度来看，就是在Window管理服务 WindowManagerService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。



removeAppToken对象的删除过程

---> ActivityStack.startActivityLocked

---> mTokenMap.remove [从HashMap中按键值删除]

---> mTaskIdToTask.get [找到对应的task]

---> task.removeAppToken [从task的mAppTokens中删除对应的token]

---> mTaskIdToTask.delete [若task中的token为空，则还需删除task]

AppWindowToken对象的创建过程

---> ActivityStack.startActivityLocked

---> insertTaskAtTop [Put current task on the top]

---> mWindowManager.addAppToken = WindowManagerService. addAppToken

---> new AppWindowToken

---> new InputApplicationHandle

---> new AppWindowAnimator

---> task.addAppToken [按照z轴顺序放置]

---> mTokenMap.put [以ActivityRecord的binder对象为键值]

removeAppToken对象的移动过程

---> ActivityStack.startActivityLocked

---> mTokenMap.remove [从HashMap中按键值删除]

---> mTaskIdToTask.get [找到对应的task]

---> task.removeAppToken [从task的mAppTokens中删除对应的token]

---> mTaskIdToTask.delete [若task中的token为空，则还需删除task]

# 输入法窗口管理

* + 1. 添加输入法窗口

addWindow

---> new WindowState

---> addInputMethodWindowToListLocked

---> addWindowToListInOrderLocked

---> moveInputMethodDialogsLocked

---> setUpdateInputWindowsNeededLw

---> moveInputMethodWindowsIfNeededLocked

---> broadcastInputMethodWindowState

# ActivityManagerService之实现

* 1. 启动

SystemServer-> Main systemServer启动

---> thr.initAndLoop

---> ActivityManagerService.main 启动Service

---> thr.start()

---> AThread.run

---> new ActivityManagerService

---> ActivityManagerService.setSystemProcess

---> ServiceManager.addService(Context.ACTIVITY\_SERVICE, m, true) 注册Binder

实现详解blog：

<http://blog.csdn.net/yueliangniao1/article/details/7227165>

ActivityManagerService.startActivity

---> ActivityManagerService.startActivityAsUser

---> ActivityStackSupervisor.startActivityMayWait

---> ActivityStackSupervisor.resolveActivity 从packageManager获取launchMode，permission，screenOrientation等信息

---> ActivityStackSupervisor.startActivityLocked

---> ActivityManagerService.checkPermission 验证是否允许启动activity

---> new ActivityRecord 创建ActivityRecord

---> ActivityStackSupervisor. startActivityUncheckedLocked 判断是否要创建新task或者重用已有的task

---> ActivityStackSupervisor .adjustStackFocus 重用或创建activityStack

---> r.setTask (targetStack.createTaskRecord(…)..) 创建新TaskRecord，并将其与activityRecord关联

* 1. 应用程从Launcher启动过程

Launcher.startActivitySafely

---> Launcher.startActivity

---> Activity. startActivity

---> Activity .startActivityForResult

---> mInstrumentation.execStartActivity

---> ActivityManagerService. startActivity

---> startActivityAsUser

---> ActivityStackSupervisor. startActivityMayWait

---> ActivityStackSupervisor . resolveActivity 从packageManager获取activity信息

---> startActivityLocked

--- > new ActivityRecord 创建新的ActivityRecord

--- > startActivityUncheckedLocked

--- > adjustStackFocus

--- > targetStack.createTaskRecord

--- > r.setTask

--- > startActivityLocked

--- > resumeTopActivitiesLocked

--- > startPausingLocked Pause当前的activity

--- > schedulePauseActivity

--- > sendMessage(H. PAUSE\_ACTIVITY)

--- > ActivityThread.handlePauseActivity

--- > performPauseActivity

--- > onPause

--- > ActivityManagerService .activityPaused

--- > activityPausedLocked

--- > completePauseLocked

--- > resumeTopActivitiesLocked

--- > startSpecificActivityLocked

--- > getProcessRecordLocked

--- > startProcessLocked

--- > Process.start (通过socket发送命令道zygote)

---> startViaZygote

ZygoteConnection.runOnce

---> Zygote.forkAndSpecialize

--- > nativeForkAndSpecialize = Dalvik\_dalvik\_system\_Zygote\_forkAndSpecialize

--- > forkAndSpecializeCommon

--- > fork

--- > handleChildProc

--- > ZygoteInit.invokeStaticMain 查找main函数

--- > ZygoteInit.MethodAndArgsCaller

--- > mMethod.invoke

--- > ActivityThread.main

--- > new ActivityThread();

--- > thread.attach(false)

--- > mgr.attachApplication(mAppThread)

--- > ActivityManagerService .attachApplication

--- > ActivityManagerService .attachApplicationLocked

--- > attachApplicationLocked

--- > realStartActivityLocked

--- > scheduleLaunchActivity

--- > sendMessage(H.LAUNCH\_ACTIVITY, r)

--- > handleLaunchActivity

--- > performLaunchActivity

--- > mInstrumentation.callActivityOnCreate

--- > activity.performStart

--- > mInstrumentation.callActivityOnStart

--- > handleResumeActivity

--- > performResumeActivity

--- > r.activity.performResume

--- > mInstrumentation.callActivityOnResume(this)

* 1. 从应用程序内部启动另一个Activity过程

Activity. startActivity

---> Activity .startActivityForResult

---> mInstrumentation.execStartActivity

---> ActivityManagerService. startActivity

---> startActivityAsUser

---> ActivityStackSupervisor. startActivityMayWait

---> ActivityStackSupervisor . resolveActivity 从packageManager获取activity信息

---> startActivityLocked

--- > new ActivityRecord 创建新的ActivityRecord

--- > startActivityUncheckedLocked

--- > targetStack = sourceTask.stack

--- > targetStack.createTaskRecord

--- > r.setTask(sourceTask, sourceRecord.thumbHolder, false) 重用task

--- > startActivityLocked

--- > resumeTopActivitiesLocked

--- > startPausingLocked Pause当前的activity

--- > schedulePauseActivity

--- > sendMessage(H. PAUSE\_ACTIVITY)

--- > ActivityThread.handlePauseActivity

--- > performPauseActivity

--- > onPause

--- > ActivityManagerService .activityPaused

--- > activityPausedLocked

--- > completePauseLocked

--- > resumeTopActivitiesLocked

--- > startSpecificActivityLocked

--- > realStartActivityLocked

--- > realStartActivityLocked

--- > scheduleLaunchActivity

--- > sendMessage(H.LAUNCH\_ACTIVITY, r)

--- > handleLaunchActivity

--- > performLaunchActivity

--- > mInstrumentation.callActivityOnCreate

--- > activity.performStart

--- > mInstrumentation.callActivityOnStart

--- > handleResumeActivity

--- > performResumeActivity

--- > r.activity.performResume

--- > mInstrumentation.callActivityOnResume(this)

# SEAndroid机制

进程安全上下文的设置

系统预设置的安全context （system.img）

build/core/Makefile

---> generate-userimage-prop-dictionary 生成system\_image\_info.txt

---> echo "selinux\_fc=$(SELINUX\_FC)" >> $(1) 参数SELINUX\_FC = build/external/sepolicy/file\_contexts

---> build\_image （build/tools/releasetools/build\_image.py）

--- > mkuserimg 根据file\_contexts的规则打包img里文件context

虚拟文件系统安装，以selinux安装为例

Sepolicy创建安全策略文件

external/sepolicy/Android.mk

--- > build\_policy 参数genfs\_contexts

Init

--- > selinux\_android\_load\_policy

--- > selinux\_android\_reload\_policy

--- > mmap

--- > security\_load\_policy

--- > 加载安全策略到内核LSM中

应用程序数据文件安全上下文设置

PackageManagerService

--- > SELinuxMMAC.readInstallPolicy 启动service是加载mac\_permissions.xml文件， 将解析的内容保存到sPackageSeinfo和sSigSeinfo两个HashMap中

PackageManagerService. installNewPackageLI

--- > PackageManagerService.scanPackageLI

--- > SELinuxMMAC.assignSeinfoValue 为安装的package分配seinfo

--- > 根据签名和报名查找sPackageSeinfo和sSigSeinfo表

--- > createDataDirsLI

--- > mInstaller.install 创建安装命令

--- > execute 将安装命令发送到installd后台进程完成安装

Installd

--- > do\_install

--- > install

--- > create\_pkg\_path 在/data/data下创建于包名同名的目录

--- > selinux\_android\_setfilecon2 为目录设置安全上下文

--- > seapp\_context\_init 解析seapp\_contexts文件

--- > seapp\_context\_lookup 根据seinfo查找seapp\_contexts文件中对应的item

--- > 对第三方应用将会匹配到 user=\_app domain=untrusted\_app type=app\_data\_file levelFrom=none

--- > setfilecon 设置新的安全上下文

# Davilk/ART虚拟机启动过程

Davilk虚拟机启动过程

Main app\_main.c

---> AndroidRuntime::start

---> startVm

---> JNI\_CreateJavaVM

---> dvmCreateJNIEnv (malloc JNIEnvExt and insert it to vm->envList)

---> dvmStartup

---> startReg

---> register\_jni\_procs

---> array[i].mProc(env) = gRegJNI

---> env->CallStaticVoidMethod = ZygoteInit -> Main

Zygote启动时创建了Davilk虚拟机实例并加载了各种系统库，系统资源以及Java核心库等，由于所有的应用程序进程都要从zygote中fork出来，基于COW（写实拷贝）机制，子进程共享了父进程的虚地址空间，因此省去了子进程创建虚拟机及加载库的开销，可以大大提高应用程序的启动速度。

ART虚拟机启动过程

Main app\_main.c

---> AndroidRuntime::start

---> jni\_invocation.Init(NULL)

--- > 读取系统属性persist.sys.dalvik.vm.lib，得到虚拟机动态库，对ART应为libart.so

--- > dlopen(library, RTLD\_NOW) 加载动态库

---> startVm

---> JNI\_CreateJavaVM

---> JniInvocation::GetJniInvocation().JNI\_CreateJavaVM

--- > JNI\_CreateJavaVM\_

---> startReg

---> register\_jni\_procs

---> array[i].mProc(env) = gRegJNI

---> env->CallStaticVoidMethod = ZygoteInit -> Main

Zygote启动时创建并初始化Davilk虚拟机实例，由于所有的应用程序进程都要从zygote中fork出来，基于COW（写实拷贝）机制，子进程共享了父进程的虚地址空间，因此省去了子进程创建虚拟机的开销。

ART虚拟机字节码转换

PackageManagerService. installNewPackageLI

--- > PackageManagerService.scanPackageLI

--- > SELinuxMMAC.assignSeinfoValue 为安装的package分配seinfo

--- > 根据签名和报名查找sPackageSeinfo和sSigSeinfo表

--- > createDataDirsLI

--- > mInstaller.install 创建安装命令

--- > execute 将安装命令发送到installd后台进程完成安装

--- > performDexOptLI

--- > performDexOptLI

--- > mInstaller.dexopt 请求发送到installd

Installd

--- > do\_dexopt

--- > dexopt

--- > 读取系统属性persist.sys.dalvik.vm.lib，得到虚拟机动态库，对ART应为libart.so

--- > create\_cache\_path 创建/data/dalvik-cache//classes.dex目录

--- > fork 创建子进程

--- > run\_dex2oat 对ART调用此函数编译机器码

--- > execl 调用命令/system/bin/dex2oat完成编译

# Davilk虚拟机进程/线程启动机制

Dalvik虚拟机进程的创建过程

ActivityManagerService. startProcessLocked

---> Process.start 将启动进程的指令通过socket发送给zygote进程

---> startViaZygote

ZygoteConnection.runOnce

---> Zygote.forkAndSpecialize

--- > nativeForkAndSpecialize = Dalvik\_dalvik\_system\_Zygote\_forkAndSpecialize

--- > forkAndSpecializeCommon

--- > fork

--- > dvmInitAfterZygot

--- > dvmGcStartupAfterZygote 进行一次GC

--- > dvmCompilerStartup 启动JIT

Dalvik虚拟机线程的创建过程

Thread.start

---> VMThread.create = Dalvik\_java\_lang\_VMThread\_create

---> dvmCreateInterpThread

--- > allocThread

--- > dvmInitInterpStack

--- > dvmInitInterpreterState

--- > pthread\_create

interpThreadStart 线程入口

---> dvmCreateJNIEnv

--- > dvmChangeThreadPriority

--- > dvmCallMethod 运行JAVA层线程入口函数

--- > dvmInterpret

--- > dvmInterpretPortable

--- > dvmDetachCurrentThread 线程结束，进行清理工作

只执行C/C++代码的Native线程的创建过程

Thread::run

---> androidCreateRawThreadEtc

---> pthread\_create

--- > Thread::\_threadLoop

能同时执行C/C++代码和Java代码的Native线程的创建过程

Thread::run

---> createThreadEtc

---> androidCreateThreadEtc

--- > gCreateThreadFn = javaCreateThreadEtc

--- > androidCreateRawThreadEtc

--- > pthread\_create

--- > AndroidRuntime::javaThreadShell

--- > javaAttachThread

--- > attachThread

--- > dvmAttachCurrentThread

--- > allocThread

--- > dvmCreateJNIEnv

--- > dvmCallMethod 执行JAVA代码

--- > start(userData) = Thread::\_threadLoop

# Davilk运行过程分析

Main app\_main.c

---> AndroidRuntime::start

---> startVm

---> JNI\_CreateJavaVM

---> dvmCreateJNIEnv (malloc JNIEnvExt and insert it to vm->envList)

---> dvmStartup

---> startReg

---> register\_jni\_procs

---> array[i].mProc(env) = gRegJNI

---> env->CallStaticVoidMethod

--- > functions->CallStaticVoidMethodV = gNativeInterface. CallStaticVoidMethodV

--- > dvmCallMethodV (jni.cpp)

--- > dvmIsNativeMethod 判断要调用的方法是否为Native方法

--- > (\*method->nativeFunc) 如果是Native方法则直接调用

Else --- > dvmInterpret 否则则为Java方法

Else --- > dvmInterpret 否则则为Java方法

--- > if kExecutionModeInterpFast, dvmMterpStd 通过解释器解释执行

--- > if kExecutionModeJit, dvmMterpStd

--- > else dvmInterpretPortable, dvmInterpretPortable

# Dalvik垃圾收集实现

* 1. 垃圾收集触发方式
     1. 回收线程自动回收

ZygoteInit -> Main Zygote启动

---> startSystemServer 启动SystemServer

---> Zygote.forkSystemServer

---> nativeForkSystemServer 通过JNI调用native代码

---> Dalvik\_dalvik\_system\_Zygote\_forkSystemServer

---> forkAndSpecializeCommon

---> dvmGcPreZygoteFork

---> fork()

---> SystemServer.main

---> dvmInitAfterZygote

---> dvmGcStartupAfterZygote

---> dvmHeapStartupAfterZygote

---> dvmHeapSourceStartupAfterZygote

---> gcDaemonStartup

---> dvmCreateInternalThread

---> gcDaemonThread

---> dvmSignalCatcherStartup

---> dvmCreateInternalThread

---> signalCatcherThreadStart

---> handleSigUsr1

---> dvmCollectGarbage

---> dvmCollectGarbageInternal

* + 1. 手动触发垃圾回收

**Dalvik\_java\_lang\_Runtime\_gc**

---> dvmCollectGarbage

---> dvmCollectGarbageInternal

* 1. 垃圾收集实现

<http://mysuperbaby.iteye.com/blog/1434423>

gcDaemonThread

---> dvmCollectGarbageInternal

---> dvmSuspendAllThreads (stop the world)

---> dvmHeapBeginMarkStep 创建位图

---> dvmHeapMarkRootSet 对所有根对象进行标记

---> dvmHeapScanMarkedObjects

---> dvmHeapProcessReferences

---> dvmHeapSweepUnmarkedObjects 对未曾标记的对象进行清除操作，也就是删除没有再使用的对象

---> dvmHeapFinishMarkStep 对已经删除的对象进行内存回收

# SurfaceFlinger服务之实现

* 1. SurfaceFlinger服务启动过程

init.rc / init.xx.rc

---> service surfaceflinger /system/bin/surfaceflinger

--- > main main\_surfaceflinger.cpp

--- > sp<ProcessState> ps(ProcessState::self()) 打开binder

--- > new SurfaceFlinger()

--- > flinger->init()

--- > mEventControlThread->run("EventControl", PRIORITY\_URGENT\_DISPLAY);

--- > sm->addService

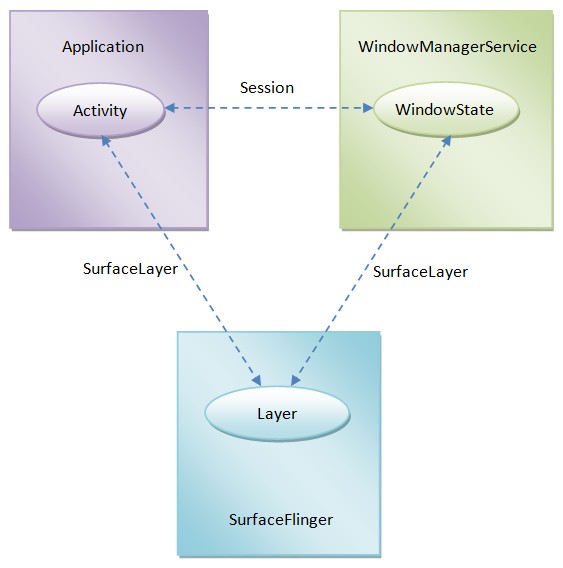
--- > SurfaceFlinger::onFirstRef

--- > mEventQueue.init(this)

--- > flinger->run

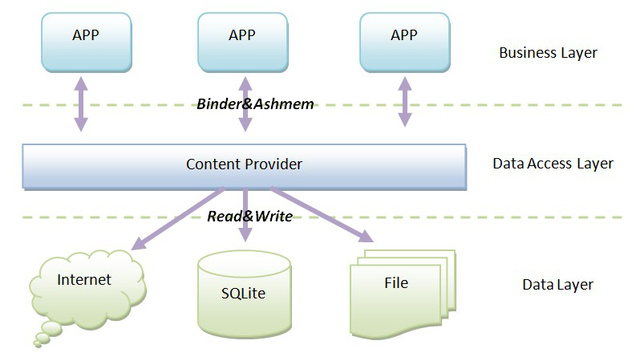
--- > waitForEvent 在mEventQueue上等待事件（epoll）

# Android四大组件之Activity



# Android四大组件之ContentProvider

实现数据共享的组件，为数据的访问提供统一的接口，隐藏数据存储介质的区别。通过Binder和Ashmem提供高效的数据共享。



# Content Provider启动

context.getContentResolver

resolver.acquireProvider = ApplicationContentResolver. acquireProvider

---> mMainThread.acquireProvider

--- > ActivityThread. acquireProvider

--- > ActivityThread .acquireExistingProvider

---> ActivityManagerNative.getDefault().getContentProvider

--- > ActivityManagerService. getContentProvider

--- > ActivityManagerService. getContentProviderImpl

--- > getProcessRecordLocked

--- > proc.pubProviders.put

--- > proc.thread.scheduleInstallProvider

Or --- > startProcessLocked [创建运行contentProvider的目标进程]

---> Process.start

---> ActivityThread.main

---> ActivityThread.attach

---> ActivityManagerNative.getDefault().attachApplication

---> ActivityManagerService. attachApplication

---> attachApplicationLocked

---> generateApplicationProvidersLocked

---> new **ContentProviderRecord**

---> bindApplication

---> handleBindApplication

---> installContentProviders

---> installProvider

---> localProvider.attachInfo

---> ContentProvider.this.onCreate

---> ActivityManagerNative.getDefault().publishContentProviders

---> mLaunchingProviders.add

---> mProviderMap.putProviderByClass

---> mProviderMap.putProviderByName

---> installProvider

# 数据共享机制

ContentResolver.query 发起数据查询

---> acquireUnstableProvider

--- > ContextImpl. acquireUnstableProvider

--- > ActivityThread. acquireProvider

--- > acquireExistingProvider

--- > ContentProvider.query

--- > ContentProviderProxy.query

---> **new BulkCursorToCursorAdaptor**

---> query [调用query接口]

---> new CursorToBulkCursorAdaptor

---> getBulkCursorDescriptor

---> new BulkCursorDescriptor

---> getWindow

---> **new CursorWindow**

---> nativeCreate

---> CursorWindow::create

---> ashmem\_create\_region [创建Ashmem]

---> mmap [内存映射]

---> ashmem\_set\_prot\_region

---> new CursorWindow

---> adaptor.initialize

---> setWindow

# Android四大组件之BroadcastReceiver

# 广播的注册

ContextImpl. registerReceiver

---> ContextImpl. registerReceiverInternal

---> new LoadedApk.ReceiverDispatcher

---> ActivityManagerService.registerReceiver

---> **new ReceiverList**

---> mRegisteredReceivers.put

---> **new BroadcastFilter**

---> mReceiverResolver.addFilter

# 发送接收广播

ContextImpl. sendBroadcast

---> ActivityManagerService. broadcastIntent

---> broadcastIntentLocked

---> mReceiverResolver.queryIntent

---> broadcastQueueForIntent

---> **new BroadcastRecord**

---> enqueueParallelBroadcastLocked

---> handleDispatchPackageBroadcast

---> scheduleBroadcastsLocked

---> processNextBroadcast

---> deliverToRegisteredReceiverLocked

---> performReceiveLocked

---> scheduleRegisteredReceiver

---> performReceive

---> new Args

---> mActivityThread.post

---> handleReceiver

---> receiver.onReceive

# Android四大组件之Service

* **分类**

一般我们认为service分为两类，本地service和远程service。

本地service顾名思义，那就是和当前应用在同一个进程中的service，彼此之间拥有共同的内存区域，所以对于某些数据的共享特别的方便和简单；   
远程service：主要牵扯到不同进程间的service访问。因为android的系统安全的原因导致了我们在不同的进程间无法使用一般的方式共享数据。在这里android为我们提供了一个AIDL工具。（android interface description language）android接口描述语言。在后边我们将会对其进行详细的介绍。

* **Service的生命周期**

和Activity相比，service的生命周期已经简单的不能再简单了，只有onCreate()->onStart()->onDestroy()三个方法。

* **Service的启动/停止**
  + **startService/stopService**

Context. stopService

---> ContextImpl. stopService

--- > ContextImpl. stopServiceCommon

--- > ActivityManagerNative.getDefault().stopService

--- > ActivityManagerService. stopService

--- > ActiveServices. stopServiceLocked

--- > stopServiceLocked

**--->** bringDownServiceIfNeededLocked

---> isServiceNeeded 如果存在bind的connection，则不停止Service

---> bringDownServiceLocked

Context. startService

---> ContextImpl. startService

--- > ContextImpl. startServiceCommon

--- > ActivityManagerNative.getDefault().startService

--- > ActivityManagerService. startService

--- > ActiveServices. startServiceLocked

--- > startServiceInnerLocked

--- > bringUpServiceLocked

--- > realStartServiceLocked

--- > bringUpServiceLocked

---> realStartServiceLocked

* + **bindService/unbindService**

Context. bindService

---> ContextImpl. bindService

--- > ContextImpl. bindServiceCommon

--- > ActivityManagerNative.getDefault().bindService

--- > ActivityManagerService.bindService

--- > ActiveServices.bindServiceLocked

--- > s.connections.get 在service的connections数组里找当前对应的Connection数组

--- > bringUpServiceLocked

--- > realStartServiceLocked

---> app.thread.scheduleCreateService

---> sendMessage(H.CREATE\_SERVICE, s)

---> handleCreateService

---> service.onCreate

---> requestServiceBindingsLocked

---> requestServiceBindingLocked

---> scheduleBindService

---> sendMessage(H.BIND\_SERVICE, s)

---> handleBindService

---> service. onBind

--- > mAm.startProcessLocked

---> startProcessLocked

---> Process.start

---> c.conn.connected 调用onConnect

Context. unbindService

---> ContextImpl. unbindService

--- > ActivityManagerNative.getDefault().unbindService

--- > ActivityManagerService. unbindServiceLocked

--- > ActiveServices. unbindServiceLocked

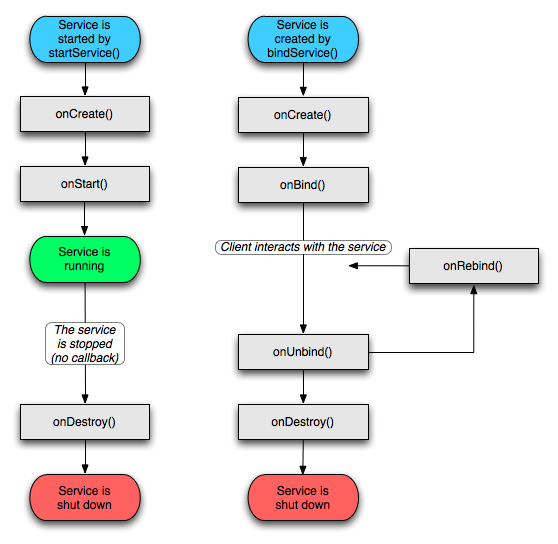
--- > removeConnectionLocked

---> scheduleUnbindService

---> sendMessage(H. UNBIND\_SERVICE, s)

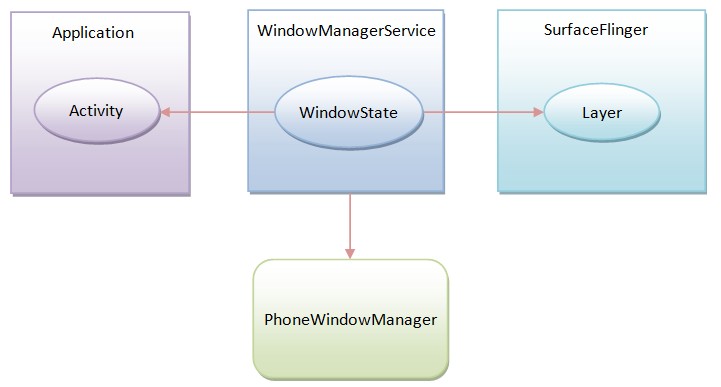
---> handleUnbindService

---> service. onUnbind

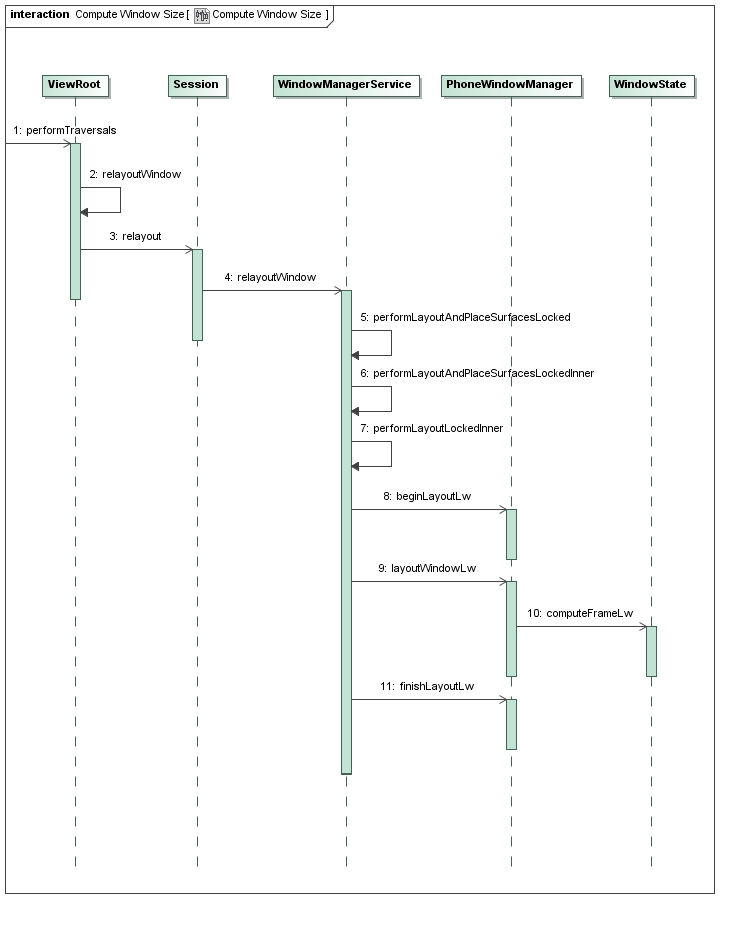


# WindowManagerService窗口管理机制

WindowManagerService服务除了要与Activity窗口所运行在的应用程序进程打交道之外，还需要与SurfaceFlinger服务以及窗口管理策略类PhoneWindowManager交互。



* 1. [窗口大小和位置（X轴和Y轴）的计算过程](http://blog.csdn.net/luoshengyang/article/details/8479101)



performTraversals

---> proc\_create 创建/proc/fb目录

--- > register\_chrdev 创建字符设备

--- > class\_create 在/system/class下创建graphics目录

register\_framebuffer 内核启动调用此函数

--- > do\_register\_framebuffer 注册帧缓冲设备

--- > device\_create 根据主从设备号在/dev/graphics下创建设备

--- > fb\_notifier\_call\_chain 通知控制台新的设备已经注册

--- > fbcon\_event\_notify

--- > fbcon\_fb\_registered

--- > fbcon\_takeover(1) 显示第一开机画面

--- > take\_over\_console

--- > fbcon\_init

--- > fbcon\_prepare\_logo

--- > fb\_prepare\_logo

--- > fb\_find\_logo

--- > fbcon\_switch

--- > fb\_show\_logo

--- > fb\_show\_logo\_line

--- >fb\_do\_show\_logo

* 1. [窗口的组织方式](http://blog.csdn.net/luoshengyang/article/details/8498908)
  2. [输入法窗口的调整过程](http://blog.csdn.net/luoshengyang/article/details/8526644)
  3. [壁纸窗口的调整过程](http://blog.csdn.net/luoshengyang/article/details/8550820)
  4. [窗口Z轴位置的计算和调整过程](http://blog.csdn.net/luoshengyang/article/details/8570428)
  5. [Activity窗口的启动窗口的显示过程](http://blog.csdn.net/luoshengyang/article/details/8577789)
  6. [Activity窗口的切换过程](http://blog.csdn.net/luoshengyang/article/details/8596449)
  7. [Activity窗口的动画显示过程](http://blog.csdn.net/luoshengyang/article/details/8611754)

# Android系统开机画面显示过程

* 1. 第一开机画面显示

内核启动阶段显示第一开机画面

fbmem\_init

---> proc\_create 创建/proc/fb目录

--- > register\_chrdev 创建字符设备

--- > class\_create 在/system/class下创建graphics目录

register\_framebuffer 内核启动调用此函数

--- > do\_register\_framebuffer 注册帧缓冲设备

--- > device\_create 根据主从设备号在/dev/graphics下创建设备

--- > fb\_notifier\_call\_chain 通知控制台新的设备已经注册

--- > fbcon\_event\_notify

--- > fbcon\_fb\_registered

--- > fbcon\_takeover(1) 显示第一开机画面

--- > take\_over\_console

--- > fbcon\_init

--- > fbcon\_prepare\_logo

--- > fb\_prepare\_logo

--- > fb\_find\_logo

--- > fbcon\_switch

--- > fb\_show\_logo

--- > fb\_show\_logo\_line

--- >fb\_do\_show\_logo

* 1. 第二开机画面显示

Init进程启动是显示第二开机画面

Main init.c

---> queue\_builtin\_action(console\_init\_action, "console\_init")

--- > list\_add\_tail(&action\_list, &act->alist)

--- > execute\_one\_command

--- > action\_remove\_queue\_head 从Action\_list获取action

--- > cur\_command->func 调用action方法

--- > console\_init\_action

--- > open(console\_name, O\_RDWR); 打开控制台

--- > load\_565rle\_image 显示第二开机画面

--- > fb\_open 打开FB设备/dev/graphics/fb0

--- > android\_memset16 将开机画面写入FB设备缓冲区

--- > restart\_processes 重启需要启动的服务

* 1. 第三开机画面显示

第三个开机画面是由应用程序bootanimation来负责显示

init.rc / init.xx.rc

---> service surfaceflinger /system/bin/surfaceflinger

--- > main main\_surfaceflinger.cpp

--- > sp<ProcessState> ps(ProcessState::self()) 打开binder

--- > new SurfaceFlinger()

--- > flinger->init()

--- > startBootAnim

--- > property\_set("ctl.start", "bootanim")

Main init.c

--- > handle\_property\_set\_fd 处理property改变

--- > handle\_control\_message 处理控制类型属性更改

--- > msg\_start

--- > service\_start 启动bootanim服务

--- > fork创建进程

--- > execve(svc->args[0], (char\*\*) arg\_ptrs, (char\*\*) ENV)

--- > main （bootanimation\_main.cpp）

--- > new BootAnimation()

--- > BootAnimation::onFirstRef

--- > run("BootAnimation", PRIORITY\_DISPLAY)

--- > BootAnimation::readyToRun

--- > mZip.open(getAnimationFileName()) 打开动画文件

--- > BootAnimation::threadLoop

---> android()/movie()

# 匿名共享内存Ashmem

# 初始化

ashmem\_init

---> kmem\_cache\_create

---> misc\_register

---> register\_shrinker

# 共享内存的创建

New MemoryFile

---> native\_open = android\_os\_MemoryFile\_open

---> ashmem\_create\_region

---> open(“/dev/ashmem”)

---> ashmem\_open

---> kmem\_cache\_zalloc

---> ioctl(ASHMEM\_SET\_NAME)

---> ashmem\_ioctl

---> set\_name

---> ioctl(ASHMEM\_SET\_SIZE)

---> native\_mmap = android\_os\_MemoryFile\_mmap

---> mmap

---> ashmem\_mmap

---> shmem\_file\_setup

---> shmem\_set\_file

# 共享内存的读写

MemoryFile. readBytes

---> native\_read = android\_os\_MemoryFile\_read

---> ashmem\_pin\_region

---> ioctl(ASHMEM\_PIN)

---> ashmem\_ioctl

---> ashmem\_pin\_unpin

---> ashmem\_pin

---> SetByteArrayRegion