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

# 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

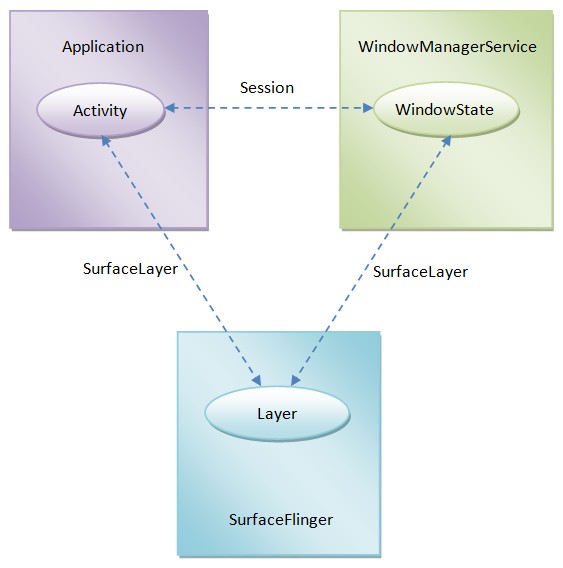
--- > SurfaceFlinger::onFirstRef

--- > mEventQueue.init(this)

--- > flinger->run

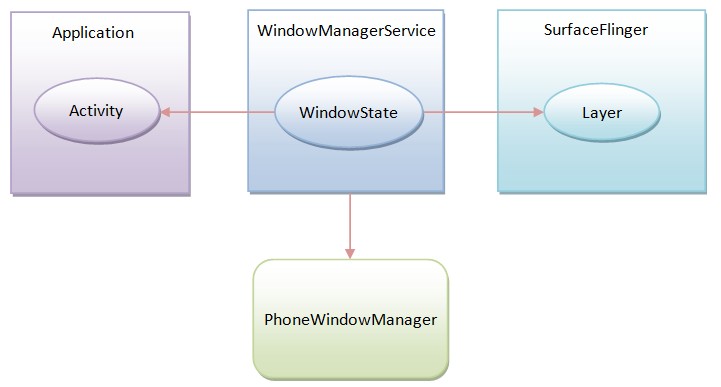
--- > waitForEvent 在mEventQueue上等待事件（epoll）

# Activity应用程序窗口实现

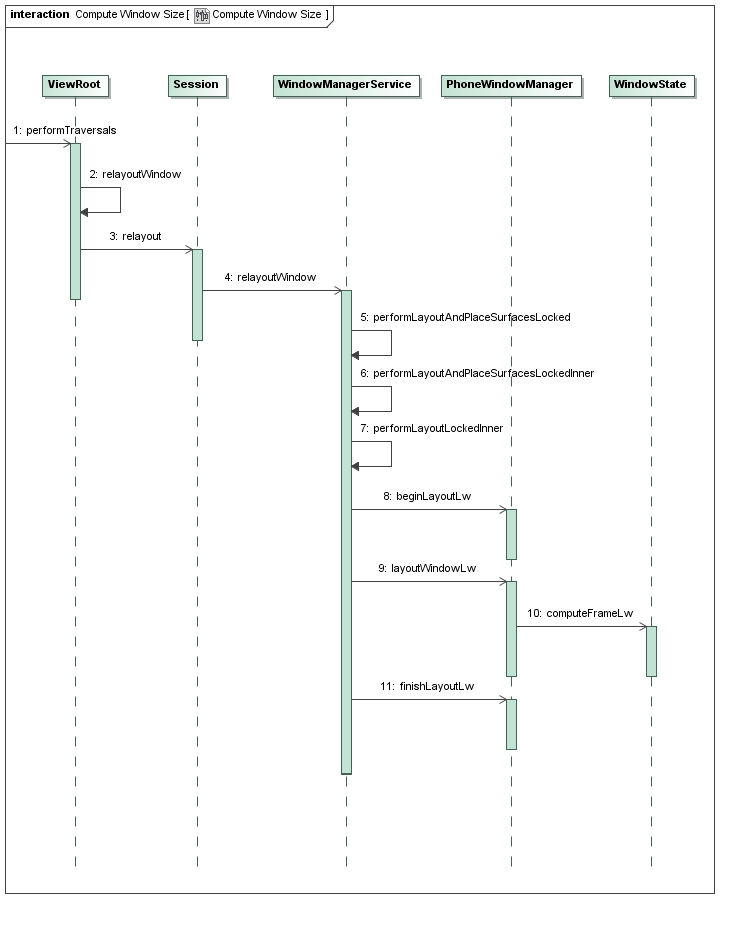


# WindowManagerService窗口管理机制

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()