# Android 系统分析

修改记录

|  |  |  |  |
| --- | --- | --- | --- |
| *版本* | *说明* | *日期* | *作者* |
| *V0.01* | *初始版本。* | *10年9月7日* | *徐申龙* |
| V0.02 | 初始化流程 | 11年10月13日 | 徐申龙 |
|  |  |  |  |
|  |  |  |  |

目录

[Android 启动流程 1](#_Toc273458953)

[两个重要的service： 3](#_Toc273458954)

[Android的键盘事件处理 4](#_Toc273458955)

[View类简析 7](#_Toc273458956)

[参考资料 10](#_Toc273458957)

### Android 启动流程及架构（服务）

**概述**

在kernle引导完成后，加载init进程，init是一个由内核启动的用户级进程，负责建立基本的系统服务。 Init启动Zygote服务，由于此时Android运行环境还没初始化，所以系统 通过app\_process进程来加载android Runtime，启动Zygote服务。Zygote是第一个android进程，所有android的应用都是通过zygote fork出来的子进程。相比execv将子进程空间完全替换，fork采用COW，开销要小，而且子进程保留了VM所需的启动信息比如JNI 信息。

Zygote fork进程 SystemServer。SystemServer建立Android系统服务，加载android\_servers库，调用init1（其JNI函数android\_server\_SystemServer\_init1 调用system\_init）建立Native 服务，如SurfaceFlinger, AudioFlinger；init1最后调用init2， init2启动各种 Android 服务（在其创建的ServerThread对象里），如 ActivityManangerService、WindowManagerService。 ActivityManagerServer.systemReady 启动第一个应用程序，

resumeTopActivityLocked -> startHomeActivityLocked ，启动Intent.CATEGORY\_HOME。

注意到system\_server 也调用了system\_init，也可以启动服务，但未被采用。可能因为system\_init 要调用JNI 函数启动SystemServer.init2，而在system\_server中启动SystemServer.init2时可能 Zygote 未启动Dalvik VM，此方法跟 Zygote 会有同步问题。

**进程分析**

我们用toolbox的ps命令来分析一下系统的进程：

USER PID PPID VSIZE RSS WCHAN PC NAME

root 1 0 320 180 c00c320c 000086cc S /init

root 2 0 0 0 c0069660 00000000 S kthreadd

root 3 2 0 0 c00590f0 00000000 S ksoftirqd/0

... ...

root 43 2 0 0 c0065ca8 00000000 S binder

... ...

root 50 1 296 152 c00c320c 000086cc S /sbin/ueventd

... ...

system 55 1 816 252 c02b8a80 afd0b6fc S /system/bin/servicemanager

root 56 1 5720 1196 ffffffff afd0bdac S /system/bin/vold

root 57 1 3856 564 ffffffff afd0bdac S /system/bin/netd

root 58 1 676 256 c02e7834 afd0c0cc S /system/bin/debuggerd

root 59 1 1280 572 c006d9f4 afd0bdac S /system/bin/rild

root 60 1 83320 28224 c00c320c afd0b844 S zygote

media 61 1 23056 4700 ffffffff afd0b6fc S /system/bin/mediaserver

bluetooth 62 1 1264 696 c00c320c afd0c59c S /system/bin/dbus-daemon

root 63 1 824 308 c037c6f0 afd0b45c S /system/bin/installd

keystore 64 1 1752 448 c02e7834 afd0c0cc S /system/bin/keystore

root 66 1 740 320 c0056978 afd0c3ac S /system/bin/sh

root 67 1 3404 160 ffffffff 00008294 S /sbin/adbd

system 78 60 179412 59184 ffffffff afd0b6fc S system\_server

system 129 60 103412 25072 ffffffff afd0c51c S com.android.systemui

app\_25 137 60 113412 35356 ffffffff afd0c51c S com.android.wallpaper

app\_12 138 60 99508 20820 ffffffff afd0c51c S com.android.inputmethod.latin

app\_23 157 60 121080 45292 ffffffff afd0c51c S com.android.launcher

app\_1 185 60 95480 18648 ffffffff afd0c51c S android.process.acore

system 214 60 103108 24680 ffffffff afd0c51c S com.android.settings

app\_8 225 60 96324 19328 ffffffff afd0c51c S android.process.media

... ...

root 357 66 904 308 00000000 afd0b45c R ps

PID是进程号， PPID是父进程号， Name是进程的名称，一般是对应的程序的文件路径。为了简单起见，我们删除了一些PPID为2和66的进程，这不影响我们的讨论。

进程的关系

PPID是进程间父子关系的体现，子进程是父进程调用系统函数fork()生成的。上面PPID只有0，1，2，60，66几种。

PPID为0的进程是内核启动的。init是第一个用户进程，放在ramdisk分区里，这个分区也是启动时的临时用户文件系统； kthreadd是内核线程管理的进程，kthreadd启动的进程（即PPID为2的进程）中有个binder，是Android IPC的基础，其他暂不关注。

PPID为1的进程就是init启动的，这是分析Android启动的重点。init根据/init.rc的内容来启动这些进程，包括55号进程/system/bin/servicemanager，60号进程zygote。

PPID为60的进程是zygote启动的。system\_server 的PPID 就是60，其他的Android应用如157，214号进程都是通过Zygote启动。

PPID 为66的是进程ps，就是我们从shell输入显示系统进程信息的命令，66号进程是sh。

解释了进程父子关系，还有个问题就是进程对应的程序。NAME是路径名很好理解，比如/system/bin/servicemanager、/sbin/adbd，但是system\_server 是/system/bin/ system\_server吗？zygote是哪个程序呢？

其实，ps 也是分析系统内存文件系统proc来获取这些信息的，我们可以通过 proc获得进程的启动命令：

# cat /proc/1/cmdline

/init

# cat /proc/60/cmdline

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

这里对cat解释一下，cmdline中用’\0’ （ASCII值0）代替了空格（ASCII值为32），前者是不可打印的字符， 如果把 cmdline 用adb 从设备pull到主机，用vi查看。

Zygote 其实就是app\_process，init.rc定义了service zygote，init启动zygote服务：

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

app\_main.cpp 函数main中使用setArgv0(argv0, "zygote"); 把argv[0]改成zygote，如果在此调用之前打印argv[0]，显示的是 “/system/bin/app\_process”。

zygote 获取AndroidRuntime来执行Java类 com.android.internal.os.ZygoteInit，传给的参数是true（因为有选项"--start-system-server"），ZygoteInit类有main方法，其主要是预加载类和资源(调用方法preloadClasses(), preloadResources())，然后就调用方法startSystemServer()，在startSystemServer()里，args数组变量有两个参数"com.android.server.SystemServer" 及 "--nice-name=system\_server"，传给Zygote.forkSystemServer，此方法是fork的包装。

app\_process的使用说明：

app\_process [java-options] cmd-dir start-class-name [options]

"com.android.server.SystemServer" 是放在最后，就是app\_process启动后要执行的Java类（当然这个类有main函数），而"--nice-name=system\_server"是 java-options，就是给Java虚拟机Dalvik的启动选项。"--nice-name=”选项由com.android.internal.os. RuntimeInit类（即AndroidRuntime，其JNI及log tag 以此表示）方法zygoteInit()解析为进程名， 调用Process.setArgV0(niceName);设置。

所以，system\_server 不是/system/bin/ system\_server的进程，而是Zygote fork出来的的app\_process进程--zygote就是一个app\_process进程。

“android.process.media”这些又是怎么来的呢？ com.android.server.am.ActivityManagerService类方法startProcessLocked (ProcessRecord, String, String) 调用 android.os.Process方法start("android.app.ActivityThread", ... ..)， 后者调用本类方法startViaZygote(), 设置了"--nice-name=”选项，从而设置了程序名（而不是用默认的app\_process）。

我们参考一下am，此命令可以启动Android activity和service，其实是一个脚本：

base=/system

export CLASSPATH=$base/framework/am.jar

exec app\_process $base/bin com.android.commands.am.Am "$@"

这个也是在app\_process执行Java类。Am要运行，必然有个入口，即main()方法。run()方法调用ActivityManagerNative.getDefault();获取IActivityManager的对象，然后，使用此对象的方法来启动service或activity。android.app.ActivityManagerNative是个抽象类，并且是隐藏（@hide），而getDefault()是个静态方法, 所以此方法可以直接调用，其实现如下：

static public IActivityManager getDefault() {

... ...

IBinder b = ServiceManager.getService("activity");

gDefault = asInterface(b);

return gDefault;

}

那么，“activity”服务是哪个类呢？就是com.android.server.am.ActivityManagerService，所以，Activity及Service的启动是通过ActivityManagerService 实现的。

从Am类看，IActivityManager 接口的startActivityAndWait或startActivity方法启动Activity， startService方法启动Service，broadcastIntent方法发送Broadcast。

startActivity方法更多的是startActivity(Intent),由android.app.ContextImpl类提供的，他通过android.app.Instrumentation.execStartActivity()间接调用ActivityManagerNative.getDefault() .startActivity()。android.app.Activity. startActivityIfNeeded() 也调用了ActivityManagerNative.getDefault() .startActivity()， 不过此方法没什么地方被调用。 而com.android.internal.policy.impl. PhoneWindowManager.goHome调用ActivityManagerNative.getDefault() .startActivity()是自用的。android.app.ContextImpl 各个sendBroadcast方法、以及startService、bindService均是直接调用了ActivityManagerNative.getDefault()。

ActivityManagerService 的startActivityAndWait 会到 ActivityStack的startActivityMayWait-> startActivityLocked->resumeTopActivityLocked ->[ensureActivitiesVisibleLocked]-> startSpecificActivityLocked再回到ActivityManagerService 的方法startProcessLocked,进而调到了Process.start

DVM虚拟机的启动流程。

frameworks/base/cmds/app\_process/app\_main.cpp的函数main得到AndroidRuntime的实例，并调用其start成员函数。AndroidRuntime::start，它主要做了三件事情，一是调用函数startVM启动虚拟机，二是调用函数startReg注册JNI方法，三是调用了com.android.internal.os.ZygoteInit类的入口方法main。

startViaZygote 好像是通过socket与Zygote通讯，让Zygote fork新的进程。让我们梳理一下ZygoteInit的main()方法。app\_process的第一个进程就是Zygote,此进程启动Dalvik虚拟机，让虚拟机执行的第一个类就是ZygoteInit，入口就是方法main(), 大致流程如下：

1.registerZygoteSocket();

2.preloadClasses();/preloadResources();

3. startSystemServer();

4.runSelectLoopMode();

第一步，是通过环境变量"ANDROID\_SOCKET\_zygote"获得zygote Socket，并以此创建LocalServerSocket对象sServerSocket.

第二步，预加载类和资源。

第三步，启动system\_server，system\_server执行RuntimeInit，启动ActivityManagerService等服务。system\_server进程不需要监听zygote socket，closeServerSocket();就是关掉他的。

第四步，进入一个无限循环，在前面创建的socket接口上等待（使用select系统调用）ActivityManagerService请求创建新的应用程序进程。具体是调用ZygoteConnection. runOnce()，后者再调用Zygote.forkAndSpecialize方法fork新进程。

回头看一下客户端，流程是startViaZygote-> zygoteSendAndGetPid-> openZygoteSocketIfNeeded, 在openZygoteSocketIfNeeded 里new了LocalSocket对象，但是socket的名字只是用了"zygote"。

init在执行init.rc脚本中时，如果service定义了socket，就在/dev/socket下创建socket， 如zygote服务下就有命令：

socket zygote stream 666

具体实现在函数service\_start，他调用函数create\_socket在ANDROID\_SOCKET\_DIR（值为/dev/socket）创建socket，得到的文件描述符通过publish\_socket函数写入到环境变量中去。zygote socket 变量名是"ANDROID\_SOCKET\_zygote"，其值为”/dev/socket/zygote”.

包android.net中，LocalServerSocket、LocalSocket分别是服务器端和客户端的类，两者使用LocalSocketAddress做地址，以LocalSocketImpl类作为连接本地库的接口，对应的JNI文件是frameworks/base/core/jni/ android\_net\_LocalSocketImpl.cpp， JNI通过libcutils库提供的API才使用了系统函数socket、connect等等，其中还提供了函数socket\_make\_sockaddr\_un，用来解析地址LocalSocketAddress。

LocalServerSocket使用的LocalSocketAddress的命名空间是LocalSocketAddress.Namespace.ABSTRACT，值是”/dev/socket/zygote”，Process方法openZygoteSocketIfNeeded命名空间是LocalSocketAddress.Namespace.RESERVED，值是“zygote”，两种地址经过socket\_make\_sockaddr\_un处理生成数据struct sockaddr\_un，类型转换成struct sockaddr供bind和connect使用。有关这两个地址查看手册man 7 unix。

init在启动zygote进程时创建了zygote socket，但是这个socket并未使用，只是为了导出一个环境变量（这仅仅是个推论，有待验证）。zygote进程使用LocalServerSocket，创建socket，绑定地址（bind）并监听（listen）zygote socket；而system\_server的服务ActivityManagerService使用LocalSocket连接zygote socket。LocalServerSocket、LocalSocket均使用LocalSocketImpl的方法create，对应的JNI函数是socket\_create，当然这两个socket的地址是一致的，虽然描述符不一样。

android包（包含以”android.”开头的类）代码放在frameworks/base/core/java/。

server包（包含以” com.android.server”开头的类）源码在frameworks/base/services/java。

init程序的代码在system/core/init/，service\_start在init.c， create\_socket在util.c实现。

libcutils库代码在 system/core/libcutils/，socket\_local\_server.c、socket\_local\_client.c、socket\_local.h，socket\_make\_sockaddr\_un在socket\_local\_client.c实现，文档在socket\_local.h， 还涉及了文件system/core/include/cutils/sockets.h。socket\_local\_server.c、socket\_local\_client.c也有创建套接字函数socket\_local\_client、socket\_local\_server，即这两函数调用了socket，但是zygote socket是由init创建的，而且android\_net\_LocalSocketImpl.cpp并没有调用到他们。

我们再来分析一下，system\_server里面的内容。

我们有个疑问，就是进程号不全是连续的，比如从78跳到了129，而我们并没有省略其中的内容？。ps选项”-t”可以帮助我们解释这个疑问，t就是thread的意思，打印进程的线程，加。进程名最长15字符，prctl函数的限制， system/core/libcutils/process\_name.c封装， frameworks/base/libs/binder/ProcessState.cpp， frameworks/base/cmds/app\_process/app\_main.cpp均匀调用。

system 78 60 179412 59384 ffffffff afd0b6fc S system\_server

system 79 78 179412 59384 c007768c afd0c738 S HeapWorker

system 80 78 179412 59384 c007768c afd0c738 S GC

system 81 78 179412 59384 c006139c afd0bfec S Signal Catcher

system 82 78 179412 59384 c037c6f0 afd0c22c S JDWP

system 83 78 179412 59384 c007768c afd0c738 S Compiler

system 84 78 179412 59384 c02b8a80 afd0b6fc S Binder Thread #

system 85 78 179412 59384 c02b8a80 afd0b6fc S Binder Thread #

system 86 78 179412 59384 c007768c afd0c738 S SurfaceFlinger

system 87 78 179412 59384 c0082be8 afd0b45c S DisplayEventThr

system 89 78 179412 59384 c00c320c afd0c59c S SensorService

system 90 78 179412 59384 c00e6438 afd0c51c S er.ServerThread

system 93 78 179412 59384 c00e6438 afd0c51c S ActivityManager

system 95 78 179412 59384 c007768c afd0c738 S ProcessStats

system 98 78 179412 59384 c00e6438 afd0c51c S PackageManager

system 99 78 179412 59384 c00e51f0 afd0b45c S FileObserver

system 102 78 179412 59384 c00e6438 afd0c51c S AccountManagerS

system 104 78 179412 59384 c00e6438 afd0c51c S SyncHandlerThre

system 106 78 179412 59384 c00c320c afd0c59c S UEventObserver

system 107 78 179412 59384 c00e6438 afd0c51c S ScreenOffThread

system 108 78 179412 59384 c00e6438 afd0c51c S PowerManagerSer

system 109 78 179412 59384 c02475c4 afd0b6fc S AlarmManager

system 110 78 179412 59384 c00e6438 afd0c51c S WindowManager

system 111 78 179412 59384 c00e6438 afd0c51c S WindowManagerPo

system 112 78 179412 59384 c00e6438 afd0c51c S InputDispatcher

.......

toolbox的ps其他的选项查看源码system/core/toolbox/ps.c

logcat

ZygoteConnection. runOnce()调用Zygote.forkAndSpecialize后面加一个log。

pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids, parsedArgs.debugFlags, rlimits);

Log.e(TAG, "forkAndSpecialize with args "+ parsedArgs);

com.android.server.am.ActivityManagerService.startProcessLocked(ProcessRecord, String, String)

new Exception().printStackTrace();

int pid = Process.start("android.app.ActivityThread",

mSimpleProcessManagement ? app.processName : null, uid, uid,

gids, debugFlags, null);

主要括号中的数字代表了进程号。

服务分Native service 和 Android service， Native service 是C/C++写的库，Android service运行在DVM上的Java代码， 他们依赖Binder，当然也依赖servicemanager。其实，还有有些服务，是守护进程，如rild， netd， adbd， debuggerd， installd， installd， vold， 至少他们在init.rc里都是用service标识，我想他们是不需要 servicemanager 进程支持的。

启动过程涉及的源码，路径均是Android源码的根目录的相对路径，其中

$(CORE) = frameworks/base/core

$(SERVICES)= framework/base /services

$(CMDS)= frameworks/base/cmds

**init 可执行程序**，系统初始化进程，放在ramdisk中，配置文件init.rc、init.$(PRODUCT).rc， 位于system/core/init

**app\_process可执行程序**： $(CMDS)/app\_process， 他的第一个进程是Zygote。

**runtime可执行程序：** 模拟器才编译的，但在模拟器上没看到此程序，设备上没有的。

**servicemanager可执行程序**：Binder IPC的桥梁，Android 服务的总管家，服务要向其注册，客户要通过它才能获得服务。

**service可执行程序：**$(CMD)/service， 显示服务列表，向服务发送数据。

**system\_server可执行程序**：$(CMDS)/system\_server，主要就是调用libsystem\_server库的system\_init函数。注意到，com.android.server.SystemServer 类的方法init1实际调用的也是system\_init，而system\_init是系统的初始化入口，难道需要调用两次吗？ 实际上，system\_server并未被使用，与runtime一样。系统中的system\_server是app\_process的进程。

**libsystem\_server共享库（so）**：$(CMDS)/system\_server/library，

主要就是函数system\_init，是Android系统的入口，她初始化SurfaceFlinger、SensorService、 AudioFlinger、 MediaPlayerService、CameraService、 AudioPolicyService等Native service，然后跳到com.android.server包类SystemServer的方法init2，init2负责Android service的初始化。

**Zygote**： 代码libcore/dalvik/src/main/java/dalvik/system/Zygote.java， 是对fork()系统调用的封装。

system/core/libcutils/zygote.c

**libdvm**： dalvik/vim/Init.c dalvik/vm/native/dalvik\_system\_Zygote.c -> Dalvik\_dalvik\_system\_Zygote\_fork, Dalvik\_dalvik\_system\_Zygote\_forkSystemServer

**ZygoteInit**：$(CORE)/java/com/android/internal/os

ZygoteInit.java, RuntimeInit.java

**libandroid\_runtime 共享库（so）**，位于 $(CORE)/jni/

com\_android\_internal\_os\_ZygoteInit.cpp 是 ZygoteInit.java的JNI,

AndroidRuntime.cpp 是 RuntimeInit.java的JNI

**services Java库，**位于$(SERVICES)/java/，是com.android.server包，包括

SystemServer.java，WifiService.java， WallpaperManagerService.java， PackageManagerService.java ，ActivityManagerService.java等等服务。

SystemServer.java有两个类： SystemServer、ServerThread ，其中SystemServer是启动Android服务的进程，有main函数， android\_servers 加载android\_servers共享库，这是为了调用init1，init1完成Native service加载，ServerThread 是做Android service的加载，init2方法中实例化ServerThread对象。但是在SystemServer.java中看不到init1到init2的跳转，实际，libandroid\_servers库把init1指向了libsystem\_server 库中的函数system\_init， system\_init在最后会通过AndroidRuntime 调用Java的静态方法init2。

ActivityManagerService.java里提供了 "activity"（ActivityManagerService）、"meminfo"（MemBinder），"cpuinfo"（CpuBinder），"permission"（PermissionController）等服务。

**libandroid\_servers共享库（so）**：位于$(SERVICES)/jni， 是services Java库的JNI支持库，因为一些services需要特定的硬件支持。

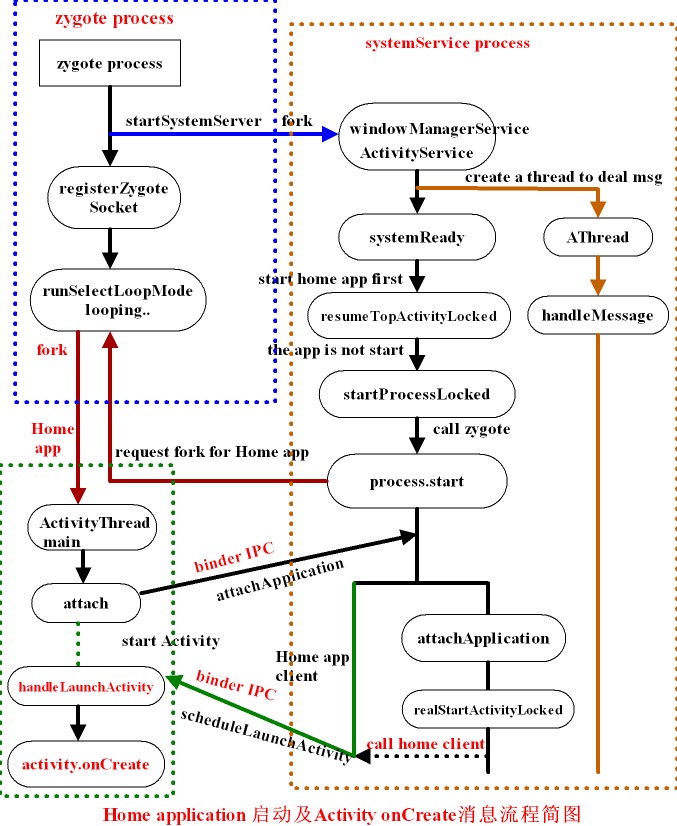
包含文件com\_android\_server\_SystemServer.cpp,

com\_android\_server\_PowerManagerService.cpp,

com\_android\_server\_UsbService.cpp,

com\_android\_server\_AlarmManagerService.cpp 等等。

com\_android\_server\_SystemServer.cpp的方法android\_server\_SystemServer\_init1 调用 函数libsystem\_server 库中的函数system\_init， 对应Java com.android.server.SystemServer类的方法 Init1，这个类是启动本地服务，如SurfaceFlinger、AudioFlinger等等。



### 两个重要的service：

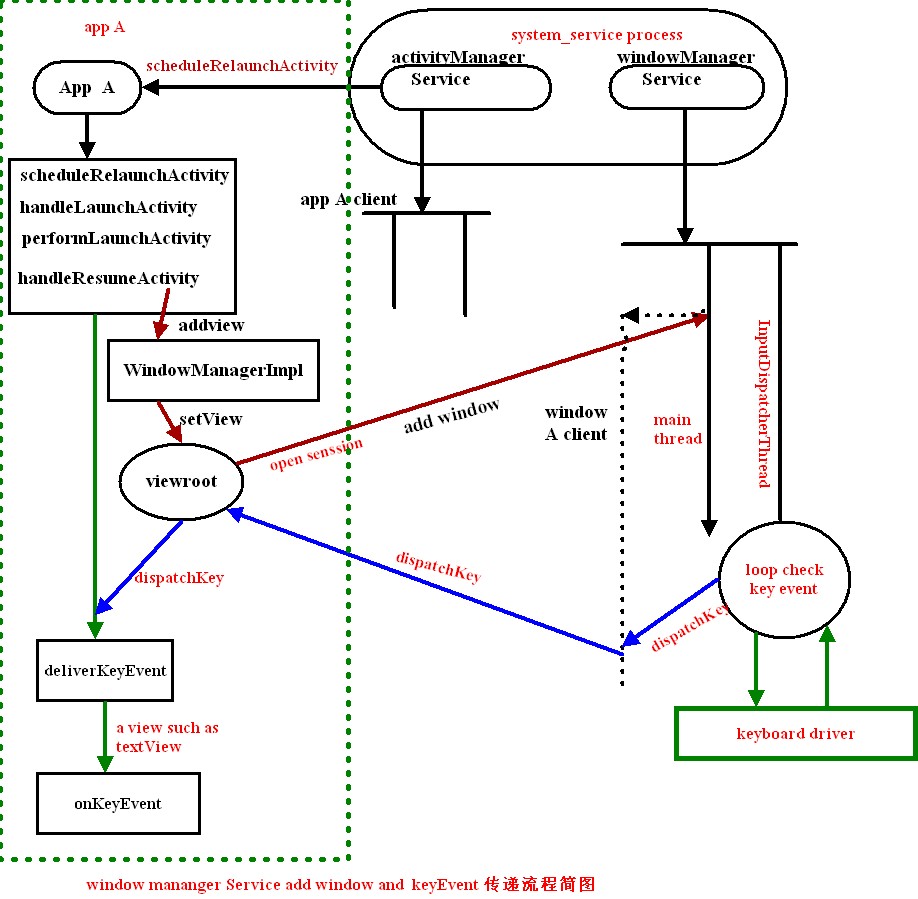
上面提到system service进程起来的过程过，启动了很多的系统service,这中间有两个service非常重要一个是ActivitManangerService,另一个是windowManagerService.

AcvivityManagerService:

主要管理Activity, service的加载，卸载，应用的创建等，当在一个应用程序中调用StartActivity时，应用就通过IPC机制向System Service请求启动新的Activity，System Service首先会查找，这个Activity是否已经被创建，如果没有,查找这个Activity所在的应用程序是否已经起来，如果没有就先启动应用。然后向此应用发送launchActivity消息创建Activity。

WindowManagerService：

     主要管理 窗体的加载，事件的分发。启动每个Activity系统都会创建一个viewRoot用于保存所有的view,通过viewRoot将窗体加载到 WindowManagerService并创建窗体的客户端，当服务程序接收到按键事件后，就会查找当前窗体是否是焦点控件，如果是的话就会将窗体事件 通过客户端IPC传递到相应的应用程序，如果在焦点View中有注册接收KeyEvent的消息则事件就在该view中得到处理。



### Android的键盘事件处理

1.     基本流程

1)         内核处理按键，通过设备文件的方式提供给framework层；

2)         framework层的KeyInputQueue.java启动线程从设备文件中读出键码，然后把读出的键码按kl文件转成相应键值(JNI调用EventHub.cpp)，最后写入事件队列；

3)         framework层的WindowManagerService.java启动线程从事件队列中读出键值，然后根据当前focus分发给相应窗口

4)         UI通过KeyCharacterMap.java处理kcm规则将用户基本按键与功能键(Shift, Alt等)组合，得出最终按键

2． 逐步分析

输入设备驱动程序对用户空间应用程序提供一些设备文件，这些设备文件放在/dev/input里面。EventHub（frameworks/base/libs/ui）是事件中转站，处理诸如按键输入设备（如Keyboard、TouchScreen、TraceBall）、vsync等事件。EventHub使用openPlatformInput( scan\_dir -> open\_device -> open）打开/dev/input输入设备文件，使用getEvent获取事件，getEvent是对设备文件描述符使用poll操作以获取驱动层发生的事件。

如果输入设备是键盘，读取键盘映射文件/system/usr/keylayout/\*.kl，让设备KeyLayoutMap 对象加载它（device->layoutMap->load(keylayoutFilename)），解析映射关系，并保存m\_keys （KeyedVector<int32\_t,Key>） 中。发生键盘事件实调用KeyLayoutMap::map函数按照映射文件转换成相应的键值。将键盘扫描码和键码进行转换 后送给KeyInputQueue。

com\_android\_server\_KeyInputQueue.cpp (frameworks\base\core\jni\server) 对 EventHub 接口的封装，并作为KeyinputQueue JNI接口向上层提供事件，如KeyinputQueue 本地方法 readEvent就是被注册为函数 android\_server\_KeyInputQueue\_readEvent 。

KeyInputQueue创建名为 ”InputDeviceReader” 线程，并在无限循环中调用 readEvent 读取事件，分析事件类型以及值，从而确定事件对应设备状态并更新设备的描述结构InputDevice。 但 KeyInputQueue 是个抽象类，被线程”InputDeviceReader”调用的方法preprocessEvent就没有实现。

WindowmanagerService内部类KeyQ扩展了KeyInputQueue，实现方法preprocessEvent；

同时实现线程类InputDispatcherThread处理事件。WindowmanagerService构造方法创建这两个线程类对象并启动线程。InputDispatcherThread. process 从KeyQ 中获取事件（QueuedEvent ev = mQueue.getEvent（）），分发（dispatchKey/dispatchPointer/dispatchTrackball

），等侍下一个事件（mQueue.recycleEvent）。

3.     相关代码

Kernel

1)drivers/input/keyboard 目录

2)drivers/input/touchscreen 目录  
3)include/linux/input.h

打开键盘设备的时候通过ioctl 获得设备名称（命令字 为EVIOCGNAME）等信息。

Android Framework

1) frameworks/base/core/java/android/view/KeyEvent.java     (按键事件定义)

2) frameworks/base/services/java/com/android/server/KeyInputQueue.java    (事件读取线程)

3) frameworks/base/services/java/com/android/server/WindowManagerService.java（事件分发线程）

4) frameworks/base/core/java/android/view/KeyCharacterMap.java   (功能键转换kcm)

5) frameworks/base/core/jni/server/com\_android\_server\_KeyInputQueue.cpp (KeyInputQueue JNI)

6) frameworks/base/libs/ui/EventHub.cpp   (键码与键值转换)

7) frameworks/base/libs/ui/KeyLayoutMap.cpp

配置文件

通常更换一种新的硬件，可能其键盘布局及键码与标准版本不同，不用更改代码，只要修改以下配置文件即可（如果增加新的未定义功能的按键，则需要修改代码）：

xxx.kl - 硬件全键盘的键码与键值的对应规则文件（如0x21对应A）

xxx.kcm - 硬件全键盘的键值对应表（如按下Alt, Shift时按键对应的键值）

两个文件源码存放位置

development/emulator/keymaps  (1.6版本模拟器使用)

sdk/emulator/keymaps   (2.2版本模拟器使用)

vendor/xxx/xxx       (特定硬件专用kl)

Android 运行环境

根据EventHub.cpp 函数 open\_device 代码片断：

// find the .kl file we need for this device

const char\* root = getenv("ANDROID\_ROOT");

snprintf(keylayoutFilename, sizeof(keylayoutFilename),

"%s/usr/keylayout/%s.kl", root, tmpfn);

bool defaultKeymap = false;

ANDROID\_ROOT 为环境变量，在android的命令模式下通过 printenv 可以知道它为： system

所以文件放在目录 /system/usr/keylayout。

UEventObserver 抽象类，提供接口供 BatteryService、HeadsetObserver、DockObserver 等使用。在SystemServer 启动为服务。

Event 设备在用户空间使用read、ioctl、poll等文件接口操作，read读取输入事件，ioctl获取设备信息，poll（轮询）进行用户空间阻塞，当内核有按键等中断时，内核在中断服务例程唤醒阻塞，这样用户空间可以从poll调用中返回。

### View类简析

invalidate: 更新View

setFocusable: 是否可以获取焦点，如静态文本设false

setEnabled: View使能，View去使能一般灰显，对按键不响应

setVisibility: View是否可见

setFadeColor

ViewAnimator ViewConfiguration

#### View 各种状态解析

|  |  |
| --- | --- |
| pressed | 是否被按住了 |
| enable/disable | 也可以称作 active / inactive，如果处于不可用 (disable)状态， View 一般是灰显，并且不响应用户事件。 |
| focus | 是否获得焦点， 在事件分派的流程， 上一级会把事件首先交给下一级的焦点View处理。 |
| select | 适用于 AdapterView 及其派生类 ListView、 GridView， 表示其选中的 Item 项。 |
| Window Focus | 表示 View 所在的窗口 (window) 是否是焦点视窗 |

系统的窗口管理器把窗口放到栈里面管理，只有一个窗口为当前窗口，即焦点窗口。每个窗口中，只有一个View为焦点的，而AdapterView 只有一个 Item 被选中。由于保存了每一级的焦点，所以当一个窗口失去焦点窗口属性时，不会失去其中的焦点 View; 当 AdapterView 失去焦点时，不会丢失选中的 Item。

android.util.StateSet 是管理 View 各种状态组合的类。

onStateChange 跟 View 无关， 而是android.graphics包中Drawable及相关类响应状态改变的接口。

onCreateDrawableState

getDrawableState

诸如 WINDOW\_FOCUSED\_STATE\_SET 等 XXX\_XXX\_STATE\_SET

这些属性（attr）在文件framework/base/core/res/res/valuse/attrs.xml

各种state 在 DrawableStates styleable下StateListDrawable

ViewTreeObserver

selector 标记是用于描述 View 各种状态的不同显示效果，如背景色、前景色。

StateListDrawable

#### View Layout（排版/布局）及Draw （绘制/渲染）

dispatchDraw 分派绘制，即告诉容器中的子 View绘制， draw 渲染View。

requestLayout 的效果等同于 invalidate，即更新 View中“脏”的区域，就是局部更新。她不会去做绘制工作，而是像 View 中的各个子 View发送更新请求。当然，最后也会发一个绘制消息，这样做是为了把多个更新合并，避免频繁绘制，（有些平台，还有只更新显存而不立即刷屏、定时刷屏等方法减少刷屏次数）。

request 调用 ViewParent，利用 ViewParet 的递归，把事件传递到ViewRoot.requestLayout，后者调用 scheduleTraversals， scheduleTraversals把消息 DO\_TRAVERSAL，放到消息队列中。 ViewRoot.handleMessage 在收到DO\_TRAVERSAL 消息后，调用 ViewRoot 私有方法 performTraversals。只有 DO\_TRAVERSAL 消息能驱动 performTraversals 工作， 这个方法是遍历整个View 做 Layout，因此开销比较大，她在做完 Layout 后，调用方法 draw 绘制 View。

Layout.java 是用于 text （文本） 的排版，而不是 View 的 Layout，后者由ViewGroup 及其派生类 LinearLayout、 FrameLayout、 RelativeLayout、AbsoluteLayout 等负责。

ViewGroup负责把多个View 按一定的布局组织起来，绘制和响应交互事件等。 这是

View 布局 和 容器的基类。

onDraw与draw不同，这是个protected方法，后者是public方法，它只是让View及其子类内部作一些特殊的绘制，而不开放给用户（client）。

onLayout（在接口ViewGroup 定义的方法） layout 调用

# 原创  Android Layout机制研究 [收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

Layout对于迅速的搭建界面和提高界面在不同分辨率的屏幕上的适应性具有很大的作用。这里简要介绍Android的Layout和研究一下它的实现。

Android有4种Layout：FrameLayout，LinearLayout，TableLayout，RelativeLayout。

放入Layout中进行排布的View的XML属性：  
4种Layout中Item所共有的XML属性：  
(1)layout\_width  
(2)layout\_height  
(3)layout\_marginLeft  
(4)layout\_marginTop  
(5)layout\_marginRight  
(6)layout\_marginBottom  
(7)layout\_gravity

FrameLayout中的Item就具有这些属性。  
对于LinearLayout还会有  
(8)layout\_weight

TableLayout的行TableRow是一个横向的（horizontal）的LinearLayout。

RelativeLayout有16个align相关的XML属性。  
(9)layout\_above  
(10)layout\_alignBaseline  
(11)layout\_alignBottom  
(12)layout\_alignLeft  
(13)layout\_alignParentBottom  
(14)layout\_alignParentLeft  
(15)layout\_alignParentRight  
(16)layout\_alignParentTop  
(17)layout\_alignRight  
(18)layout\_alignTop  
(19)layout\_below  
(20)layout\_centerHorizontal  
(21)layout\_centerInParent  
(22)layout\_centerVertical  
(23)layout\_toLeftOf  
(24)layout\_toRightOf

(1)和(2)用来确定放入Layout中的View的宽度和高度：它们的可能取值为fill\_parent，wrap\_content或者固定的像素值。  
(3)(4)(5)(6)是放入Layout中的View期望它能够和Layout的边界或者其他View之间能够相距一段距离。  
(7)用来确定View在Layout中的停靠位置。  
(8)用于在LinearLayout中把所有子View排布之后的剩余空间按照它们的layout\_weight分配给各个拥有这个属性的View。  
(9)到(24)用来确定RelativeLayout中的View相对于Layout或者Layout中的其他View的位置。

根据Android的文档，Android会对Layou和View嵌套组成的这棵树进行2次遍历，一次是measure调用，用来确定Layout或者View的大小；一次是layout调用，用来确定Layout或者view的位置。当然后来我自己的山寨实现把这2次调用合并到了一起。那就是Layout在排布之前都对自己进行measure一次，然后对View递归调用Layout方法。这样子的大小肯定是确定了的。然后用确定了的大小来使用gravity或者align属性来定位，使用margin来调整位置。伪代码如下所示：

long layout(parent\_leftPos, parent\_topPos, parent\_width, parent\_height)

{

      measure(parent\_width, parent\_height);

      while (!list.empty())

      {

             horizontalLeft = horizontalLeft - pChild->leftMargin - pChild->rightMargin;

             verticalLeft = verticalLeft - pChild->topMargin - pChild->bottomMargin;

             pChild->layout(curLeftPos, curTopPos, horizontalLeft, verticalLeft);

             // use gravity or align

             // use margin

             // scale layout size if this is a layout and it's layout\_width or layout\_height is wrap\_content

      }

      // use weight.

}

如果自己来实现这套机制的话，实现的难点就是根据用户对View或者Layout指定的layout\_width，layout\_height的三种值来确定View或者Layout的大小。

对于View来说这个比较容易。  
(1)它下面没有嵌套的Layout的时候：比如如果它是一个标准控件Button，它的layout\_width是WRAP\_CONTENT或者像素值的话，那么就是确定的了。如果layout\_width是FILL\_PARENT，它所在的Layout如果是FILL\_PARENT，也是确定了的。如果是WRAP\_CONTENT，那么这个是一个属性冲突，可以依照Layout的属性来进行处理。  
(2)它下面嵌套有Layout的时候，那么这个就涉及到了Layout的大小确定了，这转化为了Layout大小确定问题。

对于Layout来说比较复杂。  
主要Layout之间可以相互嵌套。如果父Layout是WRAP\_CONTENT，子Layout是FILL\_CONTENT，而子Layout的content则是它下面的各个View排布完成之后才可以确定的，所以View的冲突处理方法不能直接使用。也就是说measure的调用过程中，如果不对View进行排布的话，就不能得到Layout的content大小。那么既然不排布不知道，那就在measure的时候把WRAP\_CONTENT属性保留到layout调用的时候再确定吧。

在实现Layout的measure方法的时候，先要尽可能（因为WRAP\_CONTENT的原因）确定自己本身的measuredWidth和measuredHeight，然后传给各个子View或者Layout进行measure调用。在实现Layout的layout方法的时候，则要先对各个子View或者Layout进行layout调用，然后才是对Layout本身根据Layout本身的性质和Layout中各个View的属性来进行排布。这样做的目的是先排布子节点，获得子节点的大小（逐层递归调用下去，最后达到View子节点的时候就全部确定了，然后逐层递归返回，上层节点也都确定了measuredWidth和measuredHeight）。

最后有一点要说明的就是Layout在排布它的子View或者Layout的时候，由于自己的大小还没有确定，所以有些View的layout属性即使设置了，也是要忽略的。比如Android文档上提到：如果你对RelativeLayout的layout\_width设置为WRAP\_CONTENT，而对它下面的一个View设置了属性layout\_alignParentRight为true，这就是冲突的情况。所以用户在使用这些layout属性之前，首先要确保它们逻辑上都是行的通的，然后才能期望Layout机制能够正确的起作用。

当然Android的真实实现过程当然比上面的描述的要复杂的多，但是通过Eclipse上Android插件反复的试验，也可以看出Android的实现也并非是完美无瑕的，它也有赖于用户的正确使用。逻辑不通的，相互冲突的layout属性也会使它的行为难以预测。

#### View 交互接口

回调函数是一种常见的模块交互方式。

|  |  |  |
| --- | --- | --- |
| View提供的设置方法 | Listener 接口 | 接口提供的方法（回调函数） |
| setOnClickListener | OnClickListener | onClick |
| setOnLongClickListener | OnLongClickListener | onLongClick |
| setOnFocusChangeListener | OnFocusChangeListener | onFocusChange |
| setOnKeyListener | OnKeyListener | onKey |
| setOnTouchListener | OnTouchListener | onTouch |
| setOnCreateContextMenuListener | OnCreateContextMenuListener | onCreateContextMenu |

dispatchTouchEvent 先通过 OnTouchListener 的 onTouch让监听者处理这事件，如果她不处理， 调用 View 的 onTouchEvent， 即默认处理。

KeyEvent.Callback

Android操作系统中，对于事件的处理是一个非常基础而且重要的操作。许多功能的实现都需要对相关事件进行触发才能达到自己的目的。比如Android事件侦听器是视图View类的接口，包含一个单独的回调方法。这些方法将在视图中注册的侦听器被用户界面操作触发时由Android框架调用。下面这些回调方法被包含在Android事件侦听器接口中：

onClick()

包含于View.OnClickListener。当用户触摸这个item（在触摸模式下），或者通过浏览键或跟踪球聚焦在这个item上，然后按下“确认”键或者按下跟踪球时被调用。

onLongClick()

包含于View.OnLongClickListener。当用户触摸并控制住这个item（在触摸模式下），或者通过浏览键或跟踪球聚焦在这个item上，然后保持按下“确认”键或者按下跟踪球（一秒钟）时被调用。

onFocusChange()

包含于View.OnFocusChangeListener。当用户使用浏览键或跟踪球浏览进入或离开这个item时被调用。

onKey()

包含于View.OnKeyListener。当用户聚焦在这个item上并按下或释放设备上的一个按键时被调用。

onTouch()

包含于View.OnTouchListener。当用户执行的动作被当做一个触摸事件时被调用，包括按下，释放，或者屏幕上任何的移动手势（在这个item的边界内）。

onCreateContextMenu()

包含于View.OnCreateContextMenuListener。当正在创建一个上下文菜单的时候被调用（作为持续的“长点击”动作的结果）。参阅创建菜单Creating Menus章节以获取更多信息。

这些方法是它们相应接口的唯一“住户”。要定义这些方法并处理你的事件，在你的活动中实现这个嵌套接口或定义它为一个匿名类。然后，传递你的实现的一个实例给各自的View.set...Listener() 方法。（比如，调用setOnClickListener()并传递给它你的OnClickListener实现。)

下面的例子说明了如何为一个按钮注册一个点击侦听器：

1. // Create an anonymous implementation of OnClickListener
2. private OnClickListener mCorkyListener = new OnClickListener() {
3. public void onClick(View v) {
4. // do something when the button is clicked
5. }
6. };
7. protected void onCreate(Bundle savedValues) {
8. ...
9. // Capture our button from layout
10. Button button = (Button)findViewById(R.id.corky);
11. // Register the onClick listener with the implementation above
12. button.setOnClickListener(mCorkyListener);
13. ...
14. }

你可能会发现把OnClickListener作为活动的一部分来实现会便利的多。这将避免额外的类加载和对象分配。比如：

1. public class ExampleActivity extends Activity implements OnClickListener {
2. protected void onCreate(Bundle savedValues) {
3. ...
4. Button button = (Button)findViewById(R.id.corky);
5. button.setOnClickListener(this);
6. }
7. // Implement the OnClickListener callback
8. public void onClick(View v) {
9. // do something when the button is clicked
10. }
11. ...
12. }

注意上面例子中的onClick()回调没有返回值，但是一些其它Android事件侦听器必须返回一个布尔值。原因和事件相关。对于其中一些，原因如下：

· onLongClick() – 返回一个布尔值来指示你是否已经消费了这个事件而不应该再进一步处理它。也就是说，返回true 表示你已经处理了这个事件而且到此为止；返回false 表示你还没有处理它和/或这个事件应该继续交给其他on-click侦听器。

· onKey() –返回一个布尔值来指示你是否已经消费了这个事件而不应该再进一步处理它。也就是说，返回true 表示你已经处理了这个事件而且到此为止；返回false 表示你还没有处理它和/或这个事件应该继续交给其他on-key侦听器。

· onTouch() - 返回一个布尔值来指示你的侦听器是否已经消费了这个事件。重要的是这个事件可以有多个彼此跟随的动作。因此，如果当接收到向下动作事件时你返回false，那表明你还没有消费这个事件而且对后续动作也不感兴趣。那么，你将不会被该事件中的其他动作调用，比如手势或最后出现向上动作事件。

记住按键事件总是递交给当前焦点所在的视图。它们从视图层次的顶层开始被分发，然后依次向下，直到到达恰当的目标。如果你的视图（或者一个子视图）当前拥有焦点，那么你可以看到事件经由dispatchKeyEvent()方法分发。除了从你的视图截获按键事件，还有一个可选方案，你还可以在你的活动中使用onKeyDown() and onKeyUp()来接收所有的事件。

注意: Android 将首先调用事件处理器，其次是类定义中合适的缺省处理器。这样，从这些事情侦听器中返回true 将停止事件向其它Android事件侦听器传播并且也会阻塞视图中的缺事件处理器的回调函数。因此当你返回true时确认你希望终止这个事件。

#### ListView



AdapterView 虽然有很多的 Item， 但是这些 Item 其实都是由同一个 View绘制出来的，只是位置和数据不同罢了。 AdapterView 的addView、 removeView、 removeViewAt、removeAllViews 等方式均直接抛出异常，就在说明这个特点。

### 参考资料

1, AlbertChen, Android 应用初始化及窗体事件的分发

http://hi.baidu.com/albertchen521/blog/item/47dd3962482323670c33fa81.html

2，李先静， Android输入事件流程

<http://blog.csdn.net/absurd/archive/2009/05/17/4195363.aspx>

3， Android面面观——Android事件处理

<http://hi.baidu.com/zhouhanqing/blog/item/79ebf6dfac094551cdbf1a1c.html>

<http://hi.baidu.com/zhouhanqing/blog/item/9d18a3cc16020a580eb3451c.html>

4，  Android键盘系统浅析

    http://www.armfans.net/archiver/tid-2671.html

    http://www.armfans.net/viewthread.php?tid=2671

5， Android Layout机制研究

<http://blog.csdn.net/archimedes_zht/archive/2010/02/28/5333120.aspx>

6, Android Display System --- Surface Flinger

<http://blog.csdn.net/yili_xie/archive/2009/11/12/4803527.aspx>

7, Android核心分析

Android核心分析（23）-----Andoird GDI之基本原理及其总体框架

<http://blog.csdn.net/maxleng/archive/2010/06/13/5670052.aspx>

Android核心分析（24)-----Android GDI之显示缓冲管理

<http://blog.csdn.net/maxleng/archive/2010/06/14/5670701.aspx>

Android核心分析（25）------Android GDI之共享缓冲区机制

<http://blog.csdn.net/maxleng/archive/2010/06/14/5670970.aspx>

Android核心分析（26）-----Android GDI之SurfaceFlinger

<http://blog.csdn.net/maxleng/archive/2010/06/14/5671271.aspx>

Android核心分析（27）-----Android GDI 之SurfaceFlinger之动态结构示意图

<http://blog.csdn.net/maxleng/archive/2010/06/14/5671375.aspx>