第二十一期：Android 启动流程分析

eoe 特刊

Android 启动流程分析

android eoe 开发者门户

eoe 优亿市场 Android应用发布与分享平台

目录

【序言】

hxdanya——胡晓崇

绝对的手机控，从黑莓，到 Window Mobile，到 Webos，再到 Android。一路“喜新厌旧”过来。当年是黑莓中文输入法开发人员的绝对崇拜者，一个输入法改变了一个用户群。

于是乎，开始幻想自己的软件能“入侵”众人的手机。对于 Android，我还是一只刚满 6 个月大的菜鸟，相信自己可以飞的很高。

---------采访者：果子狸

【Android 启动流程分析理论介绍】

1.1 Android 启动过程——BY hxdanya

1.1.1 第一步：系统引导 bootloader(bootable/bootloader/*)

加电后，CPU 先执行 bootloader 程序,正常启动系统,加载 boot.img，boot.img 中包含内核。

1.1.2 第二步：内核 kernel(kernel/*)

由 bootloader 加载 kernel，kernel 经自解压、初始化、载入 built-in 驱动程序,完成启动。kernel 启动后会创建若干内核线程(kernel thread)，之后装入并执行程序/sbin/init/，载入 init process，切换至 user-space。

1.1.3 第三步：init 进程启动(system/core/init)

Android 从 Linux 系统启动有 4 个步骤：

1)init 进程启动

2)Native 服务启动

3)System Server,Android 服务启动

4)HOME 启动

Init 进程,是一个由内核启动的用户级进程。内核自行启动(已经被载入内存，开始运行，并已初始化所有设备驱动程序和数据结构等)之后，就通过启动一个用户级程序 init 的方式，完成引导进程。Init 始终是第一个进程。

Init 进程启动后，根据 init.rc 和 init.xxx.rc 脚本文件建立个几个基本服务：

ServiceManager

Zygote

Vold

。。。

最后 init 并不退出，而是担当 property service 的功能。

服务启动机制

(1) 打开.rc 文件，解析文件内容。(system/core/init/init.c)

将 service 信息放置到 service_list 中 (system/core/init/init_parser.c)

(2) restart_service_if_needed(...) (system/core/init/init.c)

service_start(...)

execve(...).建立 service 进程。

具体在 init.c 中，会一步步完成以下工作：

初始化 log 系统

解析/init.rc 和/init.%hardware%.rc 文件。

执行 "early-init "action in the two files parsed in step 2

设备初始化。如：在/dev 下创建所有设备节点，下载 firmwares.

初始化属性服务器.

执行"init" action in the two files parsed in step 2.

开启属性服务

执行"early-boot" 和 "boot" action in the two files parsed in step 2.

执行 Execute property action in the two files parsed in step 2

进入一个无限循环 to wait for device/property set /child process exit events.

1.1.4 第四步：Servicemanager

Servicemanager 属于 Native Service。在执行 init.rc 时就会启动，与 Zygote 一样。在 init.rc 脚本中 Servicemanager 的描述：

service servicemanager /system/bin/servicemanager

所以 Servicemanager 从 main(..)开始(frameworks/base/cmd/servicemanager/service_manager.c)

```
int main(int argc, char **argv)
{
    bs = binder_open(128*1024);//打开/dev/binder 设备，并在内存中映射 128k 的空间
    if (binder_become_context_manager(bs)) {
        return -1;
    }//通知 Binder 设备，把自己变成 context_manager
    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);//进入循环，不停的去读 Binder 设备，看是否有对 service 的请求，如果有就会调用 svcmar_handler 函数回调处理请求
    return 0;
}
```

在 Servicemanager 中的 svcmar_handler 负责处理 Android 中所有有关 service 的请求响应

```
int svcmgr_handler(...,...struct binder_io *reply)
{
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
    ....
    case SVC_MGR_ADD_SERVICE:
    ....
    case SVC_MGR_LIST_SERVICES:
    ....
}
```

如在第六步 SystemServer 中注册各种 Android 中的 service 时，调用的

ServiceManager.addService("entropy", new EntropyService())

即最终调用该 Servicemanager 中 SVC_MGR_ADD_SERVICE，执行

do_add_service(bs, s, len, ptr, txn->sender_euid)

接口，由 Servicemanager 负责注册各项服务。注册后，会将该 service 加入 svcList 中。svcList 中存了各个注册过的 service 的 name 和 handler。

当接收到获取 service 的请求时，则会执行

SVC_MGR_CHECK_SERVICE：

```
do_find_service(bs, s, len);
...
bio_put_ref(reply, ptr);
```

Servicemanager 会去查找该 service，如果存在，则把返回数据写入 reply，返回给客户。

1.1.5 第五步： Zygote

Servicemanager 和 Zygote 进程奠定了 Android 的基础。Zygote 这个进程起来才会立起真正的 Android 运行空间,初始化建立的 Service 都是 Native service.在.rc 脚本文件中 zygote 的描述：

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
所以 Zygote 从 main(...)开始 (frameworks/base/cmds/app_main.cpp)
main(...) (frameworks/base/cmds/app_main.cpp)
建立 Java Runtime
runtime.start("com.android.internal.os.ZygoteInit",startSystemServer);/
//AppRuntime runtime AppRuntime 为 AndroidRuntime 子类。实际呼叫 AndroidRuntime::start(...)
AndroidRuntime::start(...) (frameworks/base/core/jni/AndroidRuntime.cpp)
该函数中：
1.创建 Dalvik Java 虚拟机,JNI_CreateJavaVM(...);
2.注册 Android Runtime 中的 JNI 接口给虚拟机；
3.呼叫 Java 类 com.android.internal.os.ZygoteInit 中的 main 函数；

main(....) (frameworks/base/core/java/com/android/internal/os/ZygoteInit.java)
该函数中：
1;registerZygoteSocket();//登记 Listen 端口,用来接受请求；
2.preloadClasses();
preloadResources();//加载 preloaded class、resources 用来加快启动速度,文件清单在 framework.jar 中的 preloaded-class,framework-res.apk 中的 res 中；
startSystemServer();//启动 System Server;
进入 Zygote 服务框架，经过以上步骤 Zygote 就建立完成，利用 Socket 通信，接受 ActivityManagerService 的请求，fork 应用程序。

1.1.6 第六步 System Server

(1)startSystemServer()
在该函数中：
1.Zygote.forkSystemServer(...)//fork 出独立的进程，名称为"system-server".Android 的所有服务循环框架都建立在 SystemServer(SystemServer.java)上。
2.handleSystemServerProcess(...)
RuntimeInit.zygoteInit(...);
zygoteInitNative();//最终呼叫的是 AppRuntime 的 OnZygoteInit 函数
invokeStaticMain(...);//呼叫 com.android.server.SystemServer 类的 main 函数。

main(...) (frameworks/base/services/java/com/android/server/SystemServer.java)
在该函数中：
System.loadLibrary("android_servers")//首先加载 android_server 共享库。源码位于/frameworks/base/service/jni,该库中定义 JNI_onload 函数,Dalvik 在加载 libandroid_server.so 时，会呼叫该函数，该函数将 android server 注册到 Java 虚拟机中，如：HardwareService,AlarmService 等；
init1(args);//呼叫 libandroid_server.so 中注册的 native 函数 init1。
android_server_SystemServer_init1(...)//位于 /frameworks/base/services/jni/com_android_server_SystemServer.cpp 中
System_init() //最终呼叫该函数

System_init() (/frameworks/base/cmds/system_server/library/System_init.cpp)
该函数中：

SurfaceFlinger::instantiate()
sm->addService(String16(SERVICE::getServiceName()), new SERVICE());
...//将 SurfaceFilinger/AudioFlinger 等组件注册到 ServiceManager 中
runtime->callStatic("com/android/server/SystemServer", "init2")//呼叫 SystemServer 的 init2 函数。
ProcessState::self()->startThreadPool();
IPCThreadState::self()->joinThreadPool();//组成了循环闭合管理框架。

(4)init2() (frameworks/base/services/java/com/android/server/SystemServer.java)
在该函数中：
thr = new ServerThread();//创建了 ServerThead
thr.start()
run{
Looper.prepare();
...
ServiceManager.addService("entropy", new EntropyService())//将服务注册到 ServiceManager
...
Looper.loop();
}//在该 thread 中开始注册各种 service 到 service manager 中。如：PowerManagerservice、ActivityManagerService 等等。

1.1.7 第七步 HOME 启动

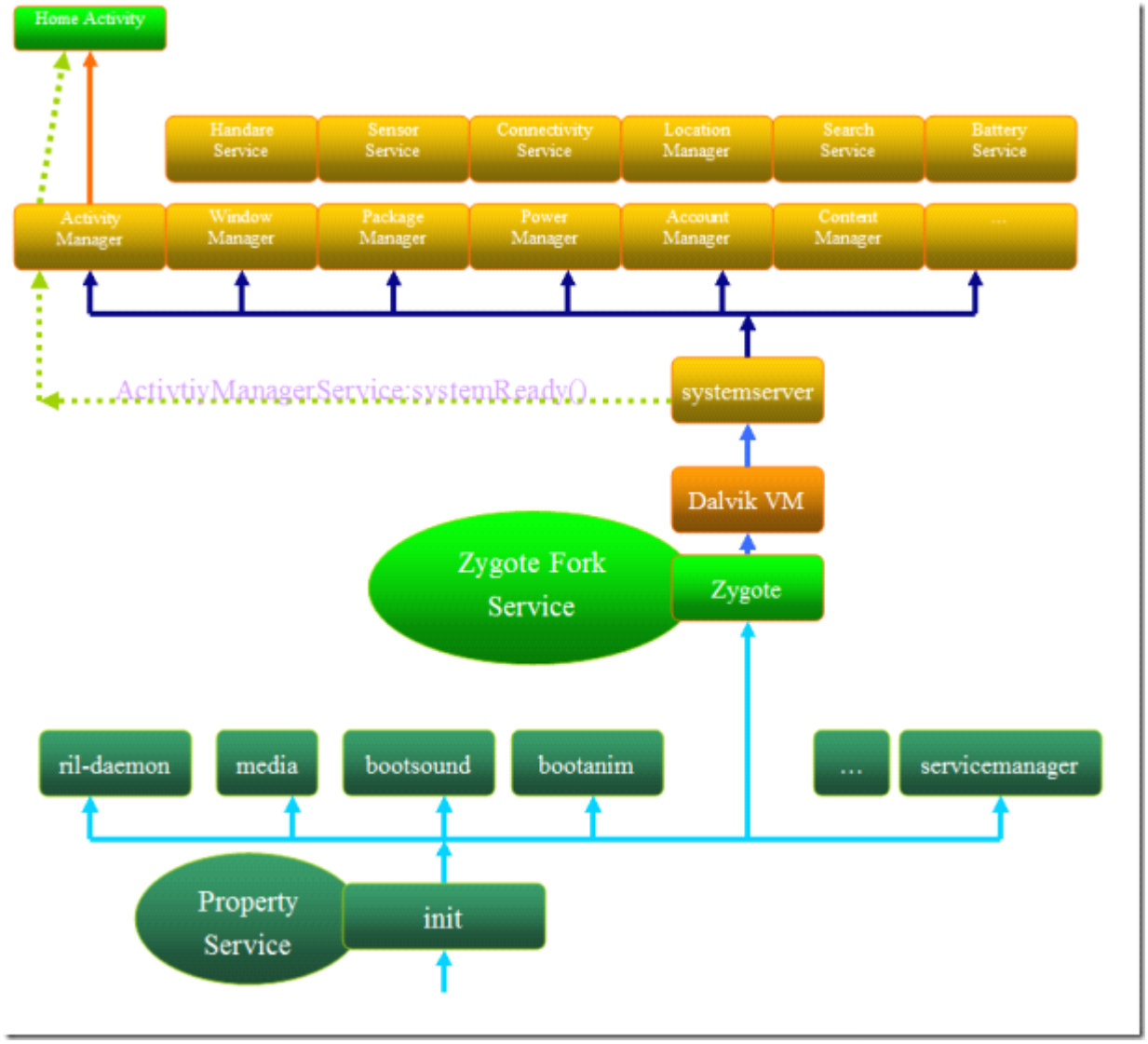
在 ServetThread 中的后半段，在系统启动完所有的 Android 服务后，做了如下动作：
run {
...
f (devicePolicy != null) {
devicePolicy.systemReady();
}
...
}//使用 XXX.systemReady()通知各个服务，系统已经准备就绪。
2.对于 ActivityManagerService
(ActivityManagerService)ActivityManagerNative.getDefault())
.systemReady(new Runnable() {
Public void run (){
startSystemUi(contextF);
appWidgetF.systemReady;
wallpaperF.systemReady();
...
} }//Widget,Wallpaper 等 ready 通知。
Home 就是在 ActivityManagerService.systemReady()通知过程中建立。
public void systemReady(final Runnable goingCallback) {
{
....
mMainStack.resumeTopActivityLocked(null);//mMainStack 为 ActivityStack 实例
}// ActivityManagerService.java

resumeTopActivityLocked(ActivityRecord prev){
return mService.startHomeActivityLocked();
}//当栈中没有任何 Activity 记录时，判断为系统处于刚开机状态，需要 start 第一个 Activity 为 Home，则返回调用 ActivityManagerService 中的 startHomeActivityLocked
接口函数，
startHomeActivityLocked(){
Intent intent = new Intent(
mTopAction,
mTopData != null ? Uri.parse(mTopData) : null);//新建一个 Intent,其中 mTopAction=Intent.ACTION_MAIN，mTopData 的值为 null.
intent.addCategory(Intent.CATEGORY_HOME);//设置 Category 为 CATEGORY_HOME。
ActivityInfo aInfo =
intent.resolveActivityInfo(mContext.getPackageManager(),
STOCK_PM_FLAGS);//根据改 intent 中的 Action 和 Cagegory 的信息，寻找出适合启动的 Activity 的信息。结果会返回 Launcher 的所在的包名，以及类名等信息。
mMainStack.startActivityLocked(null, intent, null, null, 0, aInfo,
null, null, 0, 0, 0, false, false, null);
}
startActivityLocked(...){
...
ActivityRecord r = new ActivityRecord(, , , callingUid,
intent, , aInfo, ,, , , componentSpecified);
//根据传入的 Intent 以及 Activity 的 Info 实例化一个 ActivityRecord 对象，在初始化过程中会去获得该 Activity 的各种信息，一个 ActicityRecord 对象中包含了该 Activity 的各种信息,是 History Stack 的入口。
...
return startActivityUncheckedLocked(r, sourceRecord,
grantedUriPermissions, grantedMode, onlyIfNeeded, true);//之后将该 ActivityRecord 传入该接口函数。
}
startActivityUncheckedLocked(ActivityRecord r ,...){
...
startActivityLocked(r, newTask, doResume, keepCurTransition);
}
startActivityLocked(ActivityRecord r, ...,){
...
mHistory.add(addPos, r);//将该 ActivityRecord,该 ActivityRecord 即代表了 Home Activity，添加到堆栈。
...
resumeTopActivityLocked(null);//重新又调用该接口函数，启动处于栈顶的 Activity。
}
resumeTopActivityLocked(ActivityRecord prev){
ActivityRecord next = topRunningActivityLocked(null);//此时改 next 已经不为空了，因为已经有 Home Activity 加入了历史堆栈，因此该 next 即为 Home Activity 的 ActivityRecord 实例。
...
mService.mWindowManager.setAppVisibility(next, true);//设置 Home Activity 为 Visible

```
...
startSpecificActivityLocked(next, true, false);//调用该接口函数,启动指定 Activity
return true;
..
}
startSpecificActivityLocked(ActivityRecord r,...){
...
realStartActivityLocked(r, app, andResume, checkConfig);//该接口函数会实例化一个 Launcher Activity。
...
mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
                    "activity", r.intent.getComponent(), false);//该接口函数会启动 Launcher Activity 的进程。
}
Activity a = performLaunchActivity(r, customIntent);//此时将会根据传入的 r 中的信息，实例化一个 Activity
startProcessLocked(ProcessRecord app,
                String hostingType, String hostingNameStr){
...
int pid = Process.start("android.app.ActivityThread",
                        mSimpleProcessManagement ? app.processName : null, uid, uid,
                        gids, debugFlags, null);//根据 app 指定的 uid 等信息，启动对应的进程。
...
}
}//至此 Launcher 进程便启动完成,即完成了 HOME 的启动。
```

1.2 Android 启动过程详解的补充

Android 从 Linux 系统启动有 4 个步骤；
(1) init 进程启动
(2) Native 服务启动
(3) System Server，Android 服务启动
(4) Home 启动
总体启动框架图如：

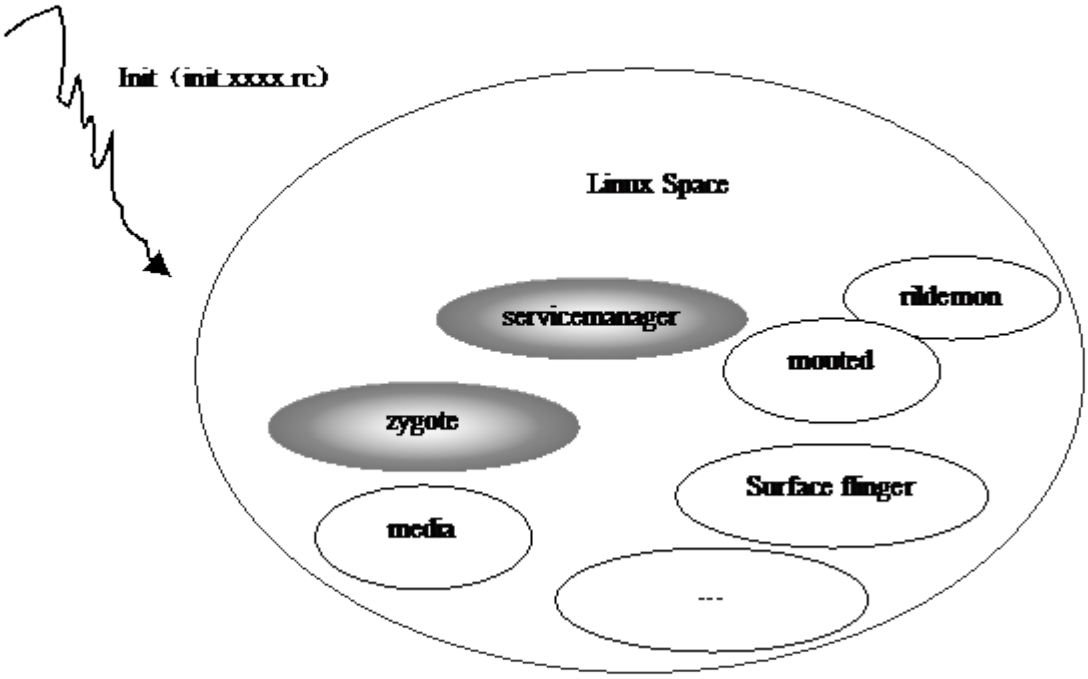


1.2.1 第一步：initial 进程(system/core/init)

init 进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序 init 的方式，完成引导进程。init 始终是第一个进程.

Init.rc

Init.marvell.rc



Init 进程一起来就根据 init.rc 和 init.xxx.rc 脚本文件建立了几个基本的服务：

servicemanamger

zygote

。。。

最后 Init 并不退出，而是担当起 property service 的功能。

①脚本文件

init@System/Core/Init

Init.c： parse_config_file(Init.rc)

@parse_config_file(Init.marvel.rc)

解析脚本文件：Init.rc 和 Init.xxxx.rc(硬件平台相关)

Init.rc 是 Android 自己规定的初始化脚本(Android Init Language, System/Core/Init/readme.txt)

该脚本包含四个类型的声明：

Actions

Commands

Services

Options.

②服务启动机制

我们来看看 Init 是这样解析.rc 文件开启服务的。

（1）打开.rc 文件，解析文件内容@ system/core/init/init.c

将 service 信息放置到 service_list 中。@ system/core/init parser.c

（2）restart_service()@ system/core/init/init.c

service_start

execve(…).建立 service 进程。

1.2.2 第二步 Zygote

Servicemanager 和 zygote 进程就奠定了 Android 的基础。Zygote 这个进程起来才会建立起真正的 Android 运行空间，初始化建立的 Service 都是 Navtive service.在.rc 脚本文件中 zygote 的描述：

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

所以 Zygote 从 main(…)@frameworks/base/cmds/app_main.cpp 开始。

①main(…)@frameworks/base/cmds/app_main.cpp

建立 Java Runtime

runtime.start("com.android.internal.os.ZygoteInit", startSystemServer);

②runtime.start@AndroidRuntime.cpp

建立虚拟机

运行：com.android.internal.os.ZygoteInit：main 函数。

③main()@com.android.internal.os.ZygoteInit//正真的 Zygote。

registerZygoteSocket();//登记 Listen 端口

startSystemServer();

进入 Zygote 服务框架。

经过这几个步骤，Zygote 就建立好了，利用 Socket 通讯，接收 ActivityManangerService 的请求，Fork 应用程序。

1.2.3 第三步 System Server

startSystemServer@com.android.internal.os.ZygoteInit 在 Zygote 上 fork 了一个进程：com.android.server.SystemServer.于是 SystemServer@(SystemServer.java）就建立了。Android 的所有服务循环框架都是建立 SystemServer@(SystemServer.java）上。在 SystemServer.java 中看不到循环结构，只是可以看到建立了 init2 的实现函数，建立了一大堆服务，并 AddService 到 service Manager。

```
main() @ com/android/server/SystemServer
{
init1();
}
```

Init1()是在 Native 空间实现的（com_andoird_server_systemServer.cpp）。我们一看这个函数就知道了，init1->system_init() @System_init.cpp

在 system_init()我们看到了循环闭合管理框架。

```
{
Call "com/android/server/SystemServer", "init2"
…..
ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

init2()@SystemServer.java 中建立了 Android 中所有要用到的服务。

这个 init2（）建立了一个线程，来 New Service 和 AddService 来建立服务

1.2.4 第四步 Home 启动

在 ServerThread@SystemServer.java 后半段，我们可以看到系统在启动完所有的 Android 服务后，做了这样一些动作：

①使用 xxx.systemReady()通知各个服务，系统已经就绪。

②特别对于 ActivityManagerService.systemReady(回调)

Widget.wallpaper,imm(输入法)等 ready 通知。

Home 就是在 ActivityManagerService.systemReady() 通知的过程中建立的。下面是 ActivityManagerService.systemReady()的伪代码：

systemReady()@ActivityManagerService.java

resumeTopActivityLocked()

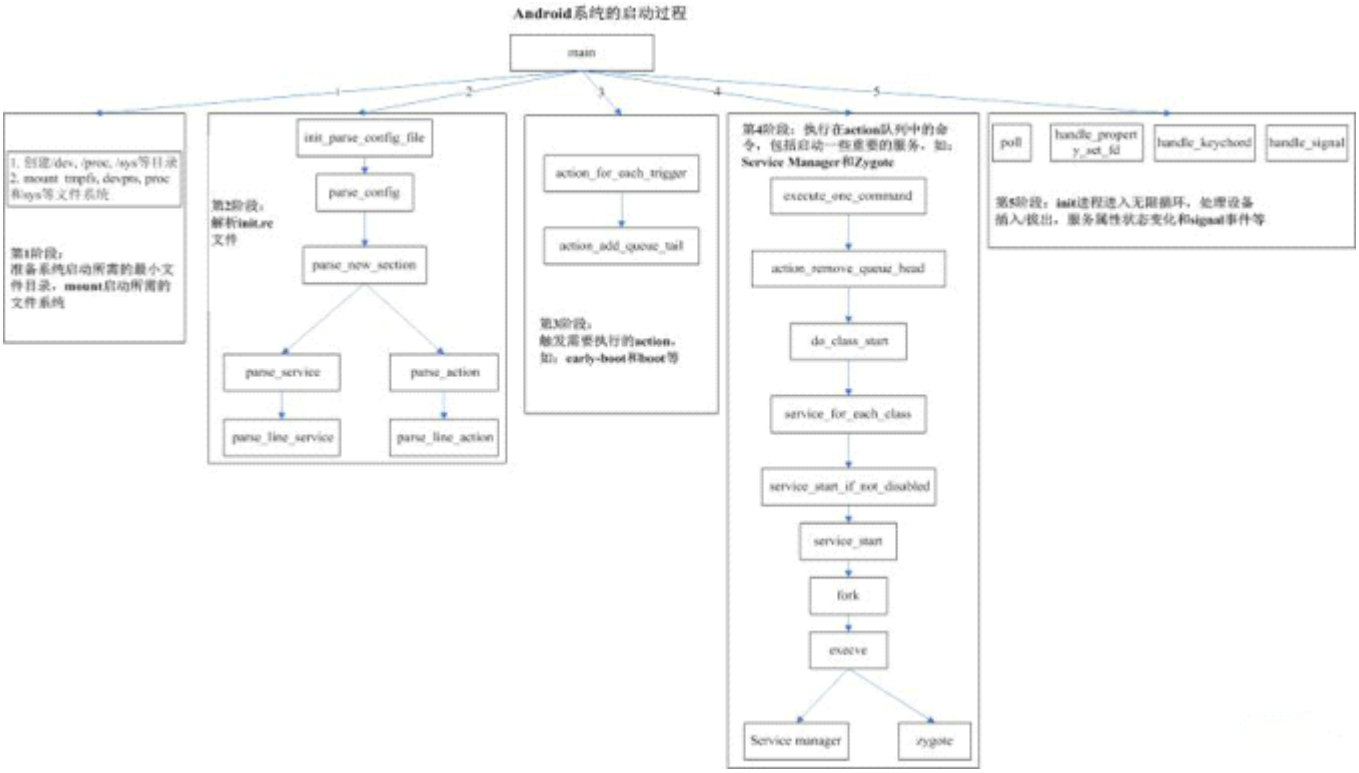
startHomeActivityLocked();//如果是第一个则启动 HomeActivity。

startActivityLocked（。。。）CATEGORY_HOME

1.3 Android 系统启动过程剖析

1.3.1 系统启动流程简介

在 Linux 内核启动后，init 1(1 号进程)将作为第一个用户空间(Linux 虚拟内存的大小为 232(在 32 位的 x86 机器上)，内核将这 4G 字节的空间分为两部分。最高的 1G 字节供内核使用，称为“内核空间”。而较低的 3G 字节供各个进程使用，称为“用户空间”。)的进程来启动 Android 系统，该启动流程可以分为如下 5 个阶段，如下图：



(1) 启动准备：该阶段包括创建文件系统的基本目录、打开基本输入、输出设备，初始化日志功能等;

(2) 解析 init.rc 文件：该阶段对 init.rc 脚本文件进行解析，主要对 Service(服务)和 Action(动作)进行解析。其中，Service 由命令(Command)和一系列服务的附加内容(Option，选项)组成，如：“service vold /system/bin/vold”为一个 Service，而“socket vold stream 0660 root mount“则为配合该服务使用的 Option;Action 则由一系列的命令组成，如：“on init mkdir /system”为系统初始化时建立系统文件夹的 Action;

(3) 触发需要执行的 action：Action 需要在 Triggers(触发条件)中调用，本阶段对需要执行的 Action 进行触发，并根据触发条件将需要执行的 Action 放入 Action 队列;

(4) 执行在 action 队列中的命令：对上一阶段触发的 Action 以及 Service 进行执行。并在此过程中，派生了 Zygote 和 Service Manager 两个非常重要的进程;

(5) 循环处理事件：init 进程进入无限循环，处理设备插入/拔出，服务属性状态变化和 signal 事件等。

1.3.2 源码分析结果

对 android-2.3.3_r1 版本中的如下源码文件进行分析：

(1) init.c：路径为 system/core/init/init.c

(2) init_parser.c：路径为 system/core/init/ init_parser.c

(3) builtins.c：路径为 system/core/init/ builtins.c

(4) property_service.c：路径为 system/core/init/property_service.c
(5) keycords.c：路径为 system/core/init/keycords.c
(6) signal_handler.c：路径为 system/core/init/signal_handler.c

总结得出系统启动流程对应的源代码文件及函数如下(注：以下函数间的顺序执行关系使用“>”表示;函数间的调用执行关系使用“à”表示)

第一阶段(启动准备)

具体的函数执行过程如下：

mkdir > mount > open_devnull_stdio > log_init

第二阶段(解析 init.rc 文件)

具体的函数调用过程如下：

init_parse_config_fileàparse_configà parse_new_sectionàparse_service (或 者 parse_action)-> parse_line_service(或者 parce_line_action)

第三阶段(触发需要执行的 action)

具体的调用过程如下：

action_for_each_trigger("boot", action_add_queue_tail);à action_add_queue_tail (class_start default) à action_remove_queue_head à do_class_start

第四阶段(执行在 action 队列中的命令)

具体的调用过程如下：

execute_one_commandà action_remove_queue_head àdo_class_start àservice_for_each_classà service_start_if_not_disabledà service_start

第五阶段(循环处理)

具体的循环处理过程如下：

for (; ;) {
poll > handle_property_set_fd > handle_keychord > handle_signal
}

主要函数介绍

函数名	所在文件	功能概述
main	system/core/init/init.c	1号进程 init 的入口函数。主要分析 init.rc 配置文件，执行基本的 action 和启动必备的 native service，然后进入一个 infinite loop 处理来自 property, signal 的 event
mkdir	system/core/init/init.c	建立文件系统的基本目录
mount	system/core/init/init.c	装载文件系统
open_devnull_stdio	system/core/init/init.c	打开基本输入、输出设备
log_init	system/core/init/init.c	初始化日志功能
init_parse_config_file	system/core/init/ init_parser.c	读取 init.rc 文件内容到内存数据区
parse_config	system/core/init/ init_parser.c	识别 init.rc 文件中的 Section（service and action series ）和 Text
parse_new_section	system/core/init/ init_parser.c	识别 section 类别

parse_service	system/core/init/ init_parser.c	对 service section 第一行进行分析
parse_line_service	system/core/init/ init_parser.c	对 service section 的 option 选项进行分析
parse_action	system/core/init/ init_parser.c	对 action section 第一行进行分析
parse_line_action	system/core/init/ init_parser.c	对 action section 的每一行独立的命令进行分析
action_for_each_trigger	system/core/init/ init_parser.c	触发某个 action 的执行
action_add_queue_tail	system/core/init/ init_parser.c	将某个 action 的从 action_list 加到 action_queue
execute_one_command	system/core/init/init.c	执行当前 action 的一个 command
action_remove_queue_head	system/core/init/ init_parser.c	从 action_queue 链表上移除头结点(action)
do_class_start	system/core/init/ builtins.c	class_start default 对应的入口函数，主要用于启动 native service
service_for_each_class	system/core/init/ init_parser.c	遍历 service_list 链表上的所有结点
service_start_if_not_disabled	system/core/init/ builtins.c	判断 service 的 flag 是否 disabled，如果不是，则调用相关函数，准备启动 service
service_start	system/core/init/init.c	启动 service 的主要入口函数，设置 service 数据结构的相关数据结构后，调用 fork 创建一个新的进行，然后调用 execve 执行新的 service
fork	Lib function(ulibc)	进程创建函数
execve	Lib function(ulibc)	调用执行新的 service
poll	Lib function(ulibc)	查询 property_set_fd，signal_fd 和 keychord_fd 文件句柄是否有服务请求
handle_property_set_fd	system/core/init/property_service.c	处理系统属性服务请求，如：service, wlan 和 dhcp 等等
handle_keychord	system/core/init/keycords.c	处理注册在 service structure 上的 keychord，通常是启动 service
handle_signal	system/core/init/signal_handler.c	处理 SIGCHLD signal

1.4 Android init 启动过程分析

分析 android 的启动过程，从内核之上，我们首先应该从文件系统的 init 开始，因为 init 是内核进入文件系统后第一个运行的程序，通常我们可以在 linux 的命令行中指定内核第一个调用谁，如果没指定那么内核将会到 /sbin/, /bin/ 等目录下查找默认的 init，如果没有找到那么就报告出错。

下面是曾经用过的几种开发板的命令行参数：

S3C2410 启动参数：

noinitrd root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs ip=192.168.2.188:192.168.2.56:192.168.2.56:255.255.255.0::eth0:on console=ttySAC0

S3C2440 启动参数：

setenv bootargs console=ttySAC0 root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs ip=192.168.2.175:192.168.2.56:192.168.2.201:255.255.255.0::eth0:on mem=64M init=/init

marvell 310 启动参数：

boot root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs,rsize=1024,wsize=1024 ip=192.168.2.176:192.168.2.201:192.168.2.201:255.255.255.0::eth0:-On console=ttyS2,115200 mem=64M init=/init

init 的源代码在文件：./system/core/init/init.c 中，init 会一步步完成下面的任务：

1.初始化 log 系统

2.解析/init.rc 和/init.%hardware%.rc 文件

3. 执行 early-init action in the two files parsed in step 2.

4. 设备初始化，例如：在 /dev 下面创建所有设备节点，下载 firmwares.

5. 初始化属性服务器，Actually the property system is working as a share memory. Logically it looks like a registry under Windows system.

6. 执行 init action in the two files parsed in step 2.

7. 开启 属性服务。

8. 执行 early-boot and boot actions in the two files parsed in step 2.

9. 执行 Execute property action in the two files parsed in step 2.

10. 进入一个无限循环 to wait for device/property set/child process exit events.例如，如果 SD 卡被插入，init 会收到一个设备插入事件，它会为这个设备创建节点。系统中比较重要的进程都是由 init 来 fork 的，所以如果他们他谁崩溃了，那么 init 将会收到一个 SIGCHLD 信号，把这个信号转化为子进程退出事件， 所以在 loop 中，init 会操作进程退出事件并且执行 *.rc 文件中定义的命令。

例如，在 init.rc 中，因为有：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
```

所以，如果 zygote 因为启动某些服务导致异常退出后，init 将会重新去启动它。

```
int main(int argc, char **argv)
{
    ...
    //需要在后面的程序中看打印信息的话，需要屏蔽 open_devnull_stdio()函数
    open_devnull_stdio();
    ...
```

```
//初始化 log 系统
log_init();
//解析/init.rc 和/init.%hardware%.rc 文件
parse_config_file("/init.rc");
...
snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
parse_config_file(tmp);
...
//执行  early-init action in the two files parsed in step 2.
action_for_each_trigger("early-init", action_add_queue_tail);
drain_action_queue();
...
/* execute all the boot actions to get us started */
/*  执行  init action in  the two files parsed in step 2 */
action_for_each_trigger("init", action_add_queue_tail);
drain_action_queue();
...
/*  执行  early-boot and boot actions in the two files parsed in step 2 */
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
drain_action_queue();

/* run all property triggers based on current state of the properties */
queue_all_property_triggers();
drain_action_queue();

/* enable property triggers */
property_triggers_enabled = 1;
...
for(;;) {
    int nr, timeout = -1;
...
    drain_action_queue();
    restart_processes();

    if (process_needs_restart) {
        timeout = (process_needs_restart - gettime()) * 1000;
        if (timeout < 0)
            timeout = 0;
    }
 ...
    nr = poll(ufds, 3, timeout);
    if (nr <= 0)
        continue;
```

```
        if (ufds[2].revents == POLLIN) {
            /* we got a SIGCHLD - reap and restart as needed */
            read(signal_recv_fd, tmp, sizeof(tmp));
            while (!wait_for_one_process(0))
                ;
            continue;
        }

        if (ufds[0].revents == POLLIN)
            handle_device_fd(device_fd);

        if (ufds[1].revents == POLLIN)
    {
            handle_property_set_fd(property_set_fd);
    }
    }

    return 0;
}
```

解析 init.rc 脚本

名词解释：

Android 初始化語言由四大类声明组成：行为类(Actions),命令类(Commands)，服务类(Services),选项类(Options).

初始化语言以行为单位，由以空格间隔的语言符号組成。C 风格的反斜杠转义符可以用来插入空白到语言符号。双引号也可以用来防止文本被空格分成多个语言符号。当反斜杠在行末时，作为换行符。

* 以#开始(前面允许空格)的行为注释。

* Actions 和 Services 隐含声明一个新的段落。所有该段落下 Commands 或 Options 的声明属于该段落。第一段落前的 Commands 或 Options 被忽略。

* Actions 和 Services 拥有唯一的命名。在他们之后声明相同命名的类将被当作错误并忽略。

Actions 是一系列命令的命名。Actions 拥有一个触发器(trigger)用来决定 action 何時执行。当一个 action 在符合触发条件被执行时，如果它还没被加入到待执行队列中的话，则加入到队列最后。

队列中的 action 依次执行，action 中的命令也依次执行。Init 在执行命令的中间处理其他活动(设备创建/销毁,property 设置，进程重启)。

Actions 的表现形式:

```
on <trigger>
   <command>
   <command>
   <command>
```

重要的数据结构

两个列表，一个队列。

```
static list_declare(service_list);
```

```
static list_declare(action_list);
static list_declare(action_queue);
```

*.rc 脚本中所有 service 关键字定义的服务将会添加到 service_list 列表中。

*.rc 脚本中所有 on 关键开头的项将会被会添加到 action_list 列表中。

每个 action 列表项都有一个列表，此列表用来保存该段落下的 Commands

脚本解析过程：

```
parse_config_file("/init.rc")
int parse_config_file(const char *fn)
{
char *data;
data = read_file(fn, 0);
if (!data) return -1;

parse_config(fn, data);
DUMP();
return 0;
}
static void parse_config(const char *fn, char *s)
{
...
case T_NEWLINE:
    if (nargs) {
        int kw = lookup_keyword(args[0]);
        if (kw_is(kw, SECTION)) {
            state.parse_line(&state, 0, 0);
            parse_new_section(&state, kw, nargs, args);
        } else {
            state.parse_line(&state, nargs, args);
        }
        nargs = 0;
    }
...
}
```

parse_config 会逐行对脚本进行解析，如果关键字类型为 SECTION ，那么将会执行 parse_new_section()

类型为 SECTION 的关键字有： on 和 sevice

关键字类型定义在 Parser.c (system/core/init) 文件中

Parser.c (system/core/init)

```
#define SECTION 0x01
#define COMMAND 0x02
#define OPTION  0x04
```

parse_new_section()中再分别对 service 或者 on 关键字开头的内容进行解析。

```
    ...
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    }
    ...
```

对 on 关键字开头的内容进行解析

```
    static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    ...
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);
    ...
}
```

对 service 关键字开头的内容进行解析

```
    static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program/n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'/n", args[1]);
        return 0;
    }
    //如果服务已经存在 service_list 列表中将会被忽略
    svc = service_find_by_name(args[1]);
    if (svc) {
```

```
            parse_error(state, "ignored duplicate definition of service '%s'/n", args[1]);
            return 0;
        }

        nargs -= 2;
        svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
        if (!svc) {
            parse_error(state, "out of memory/n");
            return 0;
        }
        svc->name = args[1];
        svc->classname = "default";
        memcpy(svc->args, args + 2, sizeof(char*) * nargs);
        svc->args[nargs] = 0;
        svc->nargs = nargs;
        svc->onrestart.name = "onrestart";
        list_init(&svc->onrestart.commands);
        //添加该服务到 service_list 列表
        list_add_tail(&service_list, &svc->slist);
        return svc;
}
```

服务的表现形式:

```
service <name> <pathname> [ <argument> ]*
<option>
<option>
...
```

申请一个 service 结构体,然后挂接到 service_list 链表上,name 为服务的名称 pathname 为执行的命令 argument 为命令的参数。之后的 option 用来控制这个 service 结构体的属性,parse_line_service 会对 service 关键字后的内容进行解析并填充到 service 结构中 ， 当遇到下一个 service 或者 on 关键字的时候此 service 选项解析结束。

例如：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
```

服务名称为： zygote

启动该服务执行的命令： /system/bin/app_process

命令的参数： -Xzygote /system/bin --zygote --start-system-server

socket zygote stream 666： 创建一个名为：/dev/socket/zygote 的 socket ， 类型为：stream

当*.rc 文件解析完成以后：

action_list 列表项目如下：

on init

on boot

on property:ro.kernel.qemu=1
on property:persist.service.adb.enable=1
on property:persist.service.adb.enable=0
init.marvell.rc 文件
on early-init
on init
on early-boot
on boot

service_list 列表中的项有：
service console
service adbd
service servicemanager
service mountd
service debuggerd
service ril-daemon
service zygote
service media
service bootsound
service dbus
service hcid
service hfag
service hsag
service installd
service flash_recovery

状态服务器相关：
在 init.c 的 main 函数中启动状态服务器。
property_set_fd = start_property_service();
状态读取函数：
Property_service.c (system/core/init)
const char* property_get(const char *name)
Properties.c (system/core/libcutils)
int property_get(const char *key, char *value, const char *default_value)
状态设置函数：
Property_service.c (system/core/init)
int property_set(const char *name, const char *value)
Properties.c (system/core/libcutils)
int property_set(const char *key, const char *value)

在终端模式下我们可以通过执行命令 setprop <key> <value>
setprop 工具源代码所在文件： Setprop.c (system/core/toolbox)
Getprop.c (system/core/toolbox): property_get(argv[1], value, default_value);

Property_service.c (system/core/init)
中定义的状态读取和设置函数仅供 init 进程调用，

handle_property_set_fd(property_set_fd);
  property_set() //Property_service.c (system/core/init)
    property_changed(name, value) //Init.c (system/core/init)
      queue_property_triggers(name, value)
      drain_action_queue()
只要属性一改变就会被触发，然后执行相应的命令：
例如：
在 init.rc 文件中有
on property:persist.service.adb.enable=1
  start adbd
on property:persist.service.adb.enable=0
  stop adbd
所以如果在终端下输入：
setprop property:persist.service.adb.enable 1 或者 0
那么将会开启或者关闭 adbd 程序。
执行 action_list 中的命令：
从 action_list 中取出 act->name 为 early-init 的列表项，再调用 action_add_queue_tail(act)将其插入到队列 action_queue 尾部。drain_action_queue() 从 action_list 队列中取出队列项 ，然后执行 act->commands 列表中的所有命令。
所以从 ./system/core/init/init.c mian()函数的程序片段：

```
action_for_each_trigger("early-init", action_add_queue_tail);
drain_action_queue();
action_for_each_trigger("init", action_add_queue_tail);
drain_action_queue();
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
drain_action_queue();
/* run all property triggers based on current state of the properties */
queue_all_property_triggers();
drain_action_queue();
```

可以看出，在解析完 init.rc init.marvell.rc 文件后，action 命令执行顺序为：
执行 act->name 为 early-init，act->commands 列表中的所有命令
执行 act->name 为 init，act->commands 列表中的所有命令
执行 act->name 为 early-boot，act->commands 列表中的所有命令
执行 act->name 为 boot，act->commands 列表中的所有命令

关键的几个命令：
class_start default 启动所有 service 关键字定义的服务。
class_start 在 act->name 为 boot 的 act->commands 列表中，所以当 class_start 被触发后，实际上调用的是函数 do_class_start（）

```
int do_class_start(int nargs, char **args)
{
        /* Starting a class does not start services
         * which are explicitly disabled.  They must
         * be started individually.
         */
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}
void service_for_each_class(const char *classname,
                            void (*func)(struct service *svc))
{
    struct listnode *node;
    struct service *svc;
    list_for_each(node, &service_list) {
        svc = node_to_item(node, struct service, slist);
        if (!strcmp(svc->classname, classname)) {
            func(svc);
        }
    }
}
```

因为在调用 parse_service() 添加服务列表的时候，所有服务 svc->classname 默认取值："default"，所以 service_list 中的所有服务将会被执行。

Zygote 服务概论：

Zygote 是 android 系统中最重要的一个服务，它将一步一步完成下面的任务：

start Android Java Runtime and start system server. It's the most important service. The source is in device/servers/app.

1. 创建 JAVA 虚拟机
2. 为 JAVA 虚拟机注册 android 本地函数
3. 调用 com.android.internal.os.ZygoteInit 类中的 main 函数，android/com/android/internal/os/ZygoteInit.java.
   a) 装载 ZygoteInit 类
   b) 注册 zygote socket
   c) 装载 preload classes(the default file is device/java/android/preloaded-classes)
   d) 装载 Load preload 资源
   e) 调用 Zygote::forkSystemServer (定义在./dalvik/vm/InternalNative.c)来 fork 一个新的进程，在新进程中调用 com.android.server.SystemServer 的 main 函数。
   a) 装载 libandroid_servers.so 库
   b) 调用 JNI native init1 函数 （device/libs/android_servers/com_android_server_SystemServers）

Load libandroid_servers.so

Call JNI native init1 function implemented in device/libs/android_servers/com_android_server_SystemServers. It only calls system_init implemented in device/servers/system/library/system_init.cpp.

If running on simulator, instantiate AudioFlinger, MediaPlayerService and CameraService here.

Call init2 function in JAVA class named com.android.server.SystemServer, whose source is in device/java/services/com/android/server. This function is very critical for Android because it start all of Android JAVA services.
If not running on simulator, call IPCThreadState::self()->joinThreadPool() to enter into service dispatcher.

SystemServer::init2 将会启动一个新的线程来启动下面的所有 JAVA 服务：
Core 服务:

1. Starting Power Manager(电源管理）
2. Creating Activity Manager（活动服务）
3. Starting Telephony Registry（电话注册服务）
4. Starting Package Manager（包管理器）
5. Set Activity Manager Service as System Process
6. Starting Context Manager
7. Starting System Context Providers
8. Starting Battery Service（电池服务）
9. Starting Alarm Manager（闹钟服务）
10. Starting Sensor Service
11. Starting Window Manager（启动窗口管理器）
12. Starting Bluetooth Service（蓝牙服务）
13. Starting Mount Service

其他 services：

1. Starting Status Bar Service（状态服务）
2. Starting Hardware Service（硬件服务）
3. Starting NetStat Service（网络状态服务）
4. Starting Connectivity Service
5. Starting Notification Manager
6. Starting DeviceStorageMonitor Service
7. Starting Location Manager
8. Starting Search Service（查询服务）
9. Starting Clipboard Service
10. Starting Checkin Service
11. Starting Wallpaper Service
12. Starting Audio Service
13. Starting HeadsetObserver
14. Starting AdbSettingsObserver

最后 SystemServer::init2 将会调用 ActivityManagerService.systemReady 通过发送 Intent.CATEGORY_HOME intent 来启动第一个 activity.还有另外一种启动 system server 的方法是：通过名为 system_server 的程序（源代码：device/servers/system/system_main.cpp）它也是通过调用 system_init 来启动 system services，这时候就有个问题：为什么 android 有两种方式启动 system services?

我的猜想是：
My guess is that directly start system_server may have synchronous problem with zygote because system_server

will call JNI to start SystemServer::init2, while at that time zygote may not start JAVA VM yet. So Android uses another method. After zynote is initialized, fork a new process to start system services.

Zygote 服务启动的详细过程：

通过启动服务列表的 app_process 进程，实际上进入的是

App_main.cpp (frameworks/base/cmds/app_process)

main()

根据 init.rc 中的 --zygote --start-system-server

分别调用的是

runtime.start("com.android.internal.os.ZygoteInit",startSystemServer);

或者

runtime.start();

start()函数在 AndroidRuntime.cpp (frameworks/base/core/jni)文件中

从打印信息：

D/AndroidRuntime(56): >>>>>>>>>>>>>> AndroidRuntime START <<<<<<<<<<<<<<

也可以看出调用的是：

AndroidRuntime::start(const char* className, const bool startSystemServer)

JNI_CreateJavaVM()

startReg()

LOGD("--- registering native functions ---/n");

startMeth = env->GetStaticMethodID(startClass, "main","([Ljava/lang/String;)V");

从上面的调用可以看出一类引用的过程都是从 main 方法,所以接着调用了 ZygoteInit 类的 main 方法

main 方法主要完成：

1.Register zygote socket, Registers a server socket for zygote command connections

2.Load preload classes(the default file is device/java/android/preloaded-classes).

3.Load preload resources, Load in commonly used resources, so they can be shared across processes.

4.Start SystemServer, Prepare the arguments and fork for the system server process.

具体执行过程如下：

ZygoteInit.java (frameworks/base/core/java/com/android/internal/os)中的 mian

main()

registerZygoteSocket()

preloadClasses()

loadLibrary()

Log.i(TAG, "Preloading classes...");

Runtime.loadLibrary

Dalvik_java_lang_Runtime_nativeLoad()

dvmLoadNativeCode()

LOGD("Trying to load lib %s %p/n", pathName, classLoader)

System.loadLibrary("media_jni");

preloadResources();

startSystemServer()

Zygote.forkSystemServer(parsedArgs.uid, parsedArgs.gid,parsedArgs.gids, debugFlags, null);

//Zygote.java (dalvik/libcore/dalvik/src/main/java/dalvik/system)

forkSystemServer()

forkAndSpecialize() //Zygote.java (dalvik/libcore/dalvik/src/main/java/dalvik/system)

```
            Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c (dalvik/vm/native)
              Dalvik_dalvik_system_Zygote_forkAndSpecialize()
                setSignalHandler()
                fork()
        handleSystemServerProcess() //handleChildProc(parsedArgs, descriptors, newStderr);
          closeServerSocket();
        RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
          zygoteInit()              //RuntimeInit.java (frameworks/base/core/java/com/android/internal/os)
            zygoteInitNative()
              invokeStaticMain()
                System.loadLibrary("android_servers");
                //com.android.server.SystemServer startSystemServer()  函数中
                m = cl.getMethod("main", new Class[] { String[].class });
                  //  执  行  的  是  SystemServer  类  的  main  函  数        SystemServer.java
(frameworks/base/services/java/com/android/server)
                  init1() //SystemServer.java (frameworks/base/services/java/com/android/server)
                    //init1()实际上是调用 android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
                    //com_android_server_SystemServer.cpp (frameworks/base/services/jni)
                  android_server_SystemServer_init1()//JNI  调用
                    system_init() //System_init.cpp (frameworks/base/cmds/system_server/library)
                      // Start the SurfaceFlinger
                      SurfaceFlinger::instantiate();
                      //Start the AudioFlinger media playback    camera service
                        AudioFlinger::instantiate();
                        MediaPlayerService::instantiate();
                        CameraService::instantiate();
                        //调用  SystemServer  类的 init2
                        runtime->callStatic("com/android/server/SystemServer", "init2");
                        init2()//SystemServer.java (frameworks/base/services/java/com/android/server)
                        ServerThread()
                          run()//在 run 中启动电源管理,蓝牙,等核心服务以及状态,查找等其他服务

((ActivityManagerService)ServiceManager.getService("activity")).setWindowManager(wm);
                            ...
                            ActivityManagerNative.getDefault().systemReady();
  runSelectLoopMode();
    done = peers.get(index).runOnce();
      forkAndSpecialize()     //Zygote.java (dalvik/libcore/dalvik/src/main/java/dalvik/system)
        Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c (dalvik/vm/native)
          forkAndSpecializeCommon()
            setSignalHandler()
          RETURN_INT(pid);
  closeServerSocket();
    主進程 runSelectLoopMode()
```

5.Runs the zygote process's select loop runSelectLoopMode(), Accepts new connections as they happen, and reads commands from connections one spawn-request's worth at a time.
如果运行正常，则 zygote 进程会在 runSelectLoopMode()中循环：

zygote 被 siganl(11)终止
在 dalvik_system_Zygote.c (dalvik/vm/native)
的 static void sigchldHandler(int s) 函数中打印：
"Process %d terminated by signal (%d)/n",
"Exit zygote because system server (%d) has terminated/n",

startSystemServer() ZygoteInit.java (frameworks/base/core/java/com/android/internal/os)
SystemServer 的 mian()函数会调用
SystemServer.java (frameworks/base/services/java/com/android/server)中的 init1()函数。
init1()实际执行的是 com_android_server_SystemServer.cpp (frameworks/base/services/jni)
中的 android_server_SystemServer_init1()。
android_server_SystemServer_init1()调用的是
System_init.cpp (frameworks/base/cmds/system_server/library) 中的 system_init()函数
system_init()函数定义如下：

```
extern "C" status_t system_init()
{
    ...
    sp<IServiceManager> sm = defaultServiceManager();
    ...
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        //读取属性服务器,开启启动 SurfaceFlinger 服务
        //接着会开始显示机器人图标
        //BootAnimation.cpp (frameworks/base/libs/surfaceflinger):status_t BootAnimation::readyToRun()
        SurfaceFlinger::instantiate();

    }
    //在模拟器上 audioflinger 等几个服务与设备上的启动过程不一样，所以
    //我们在这里启动他们。

    if (!proc->supportsProcesses()) {
        //启动 AudioFlinger，media playback service，camera service 服务
        AudioFlinger::instantiate();
        MediaPlayerService::instantiate();
        CameraService::instantiate();
    }
    //现在开始运行 the Android runtime ，我们这样做的目的是因为必须在 core system services
    //起来以后才能 Android runtime initialization，其他服务在调用他们自己的 main()时，都会
    //调用 Android runtime
    //before calling the init function.
```

```
    LOGI("System server: starting Android runtime./n");
    AndroidRuntime* runtime = AndroidRuntime::getRuntime();
    LOGI("System server: starting Android services./n");
    //调用 SystemServer.java (frameworks/base/services/java/com/android/server)
    //中的 init2 函数
    runtime->callStatic("com/android/server/SystemServer", "init2");

    // If running in our own process, just go into the thread
    // pool.    Otherwise, call the initialization finished
    // func to let this process continue its initilization.
    if (proc->supportsProcesses()) {
        LOGI("System server: entering thread pool./n");
        ProcessState::self()->startThreadPool();
        IPCThreadState::self()->joinThreadPool();
        LOGI("System server: exiting thread pool./n");
    }
    return NO_ERROR;
}
```

System server: entering thread pool 表明已经进入服务线程 ServerThread
在 ServerThread 类的 run 服务中开启核心服务：

```
    @Override
    public void run() {
        EventLog.writeEvent(LOG_BOOT_PROGRESS_SYSTEM_RUN,
            SystemClock.uptimeMillis());

        ActivityManagerService.prepareTraceFile(false);        // create dir

        Looper.prepare();
    //设置线程的优先级
        android.os.Process.setThreadPriority(
                android.os.Process.THREAD_PRIORITY_FOREGROUND);
    ...
        //关键（核心）服务
        try {
            Log.i(TAG, "Starting Power Manager.");
            Log.i(TAG, "Starting    activity Manager.");
            Log.i(TAG, "Starting telephony registry");
            Log.i(TAG, "Starting Package Manager.");
            Log.i(TAG, "    tarting Content Manager.");
            Log.i(TAG, "Starting System Content Providers.");
            Log.i(TAG, "Starting Battery Service.");
            Log.i(TAG, "Starting Alarm Manager.");
```

```
        Log.i(TAG, "Starting Sensor Service.");
        Log.i(TAG, "Starting Window Manager.");
        Log.i(TAG, "Starting Bluetooth Service.");
        //如果是模拟器，那么跳过蓝牙服务。
        // Skip Bluetooth if we have an emulator kernel
    //其他的服务
        Log.i(TAG, "Starting Status Bar Service.");
        Log.i(TAG, "Starting Clipboard Service.");
        Log.i(TAG, "Starting Input Method Service.");
        Log.i(TAG, "Starting Hardware Service.");
        Log.i(TAG, "Starting NetStat Service.");
        Log.i(TAG, "Starting Connectivity Service.");
        Log.i(TAG, "Starting Notification Manager.");
        // MountService must start after NotificationManagerService
        Log.i(TAG, "Starting Mount Service.");
    Log.i(TAG, "Starting DeviceStorageMonitor service");
        Log.i(TAG, "Starting Location Manager.");
        Log.i(TAG, "Starting Search Service.");
        ...
        if (INCLUDE_DEMO) {
            Log.i(TAG, "Installing demo data...");
            (new DemoThread(context)).start();
        }
        try {
            Log.i(TAG, "Starting Checkin Service.");
            Intent intent = new Intent().setComponent(new ComponentName(
                    "com.google.android.server.checkin",
                    "com.google.android.server.checkin.CheckinService"));
            if (context.startService(intent) == null) {
                Log.w(TAG, "Using fallback Checkin Service.");
                ServiceManager.addService("checkin", new FallbackCheckinService(context));
            }
        } catch (Throwable e) {
            Log.e(TAG, "Failure starting Checkin Service", e);
        }


        Log.i(TAG, "Starting Wallpaper Service");
    Log.i(TAG, "Starting Audio Service");
        Log.i(TAG, "Starting HeadsetObserver");
        Log.i(TAG, "Starting AppWidget Service");
    ...
        try {
            com.android.server.status.StatusBarPolicy.installIcons(context, statusBar);
```

```
            } catch (Throwable e) {
                Log.e(TAG, "Failure installing status bar icons", e);
            }
        }

        // make sure the ADB_ENABLED setting value matches the secure property value
        Settings.Secure.putInt(mContentResolver, Settings.Secure.ADB_ENABLED,
                "1".equals(SystemProperties.get("persist.service.adb.enable")) ? 1 : 0);

        // register observer to listen for settings changes
        mContentResolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.ADB_ENABLED),
                false, new AdbSettingsObserver());

        // It is now time to start up the app processes...
        boolean safeMode = wm.detectSafeMode();
        if (statusBar != null) {
            statusBar.systemReady();
        }
        if (imm != null) {
            imm.systemReady();
        }
        wm.systemReady();
        power.systemReady();
        try {
            pm.systemReady();
        } catch (RemoteException e) {
        }
        if (appWidget != null) {
            appWidget.systemReady(safeMode);
        }

        // After making the following code, third party code may be running...
        try {
            ActivityManagerNative.getDefault().systemReady();
        } catch (RemoteException e) {
        }

        Watchdog.getInstance().start();

        Looper.loop();
        Log.d(TAG, "System ServerThread is exiting!");
    }
```

```
startActivity()
  mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);


ActivityManagerService.java 3136p (frameworks/base/services/java/com/android/server/am)
  startActivity()
    startActivityLocked() //3184
    int res = startActivityLocked(caller, intent, resolvedType,grantedUriPermissions, grantedMode, aInfo,
                        resultTo, resultWho, requestCode, -1, -1,
                        onlyIfNeeded, componentSpecified);


public abstract class ActivityManagerNative extends Binder implements IActivityManager
ActivityManagerService.java 1071p  (frameworks/base/services/java/com/android/server/am)
ActivityManagerService.main()
  //ActivityManagerService.java 7375p (frameworks/base/services/java/com/android/server/am)
  m.startRunning(null, null, null, null);
    //ActivityManagerService.java 7421p (frameworks/base/services/java/com/android/server/am)
    systemReady();

ActivityManagerService.java 3136p (frameworks/base/services/java/com/android/server/am)
  startActivity(IApplicationThread caller,Intent intent,...)
    int startActivityLocked(caller, intent,...)      //3184L 定义：2691L
      void startActivityLocked()          //3132L 定义：2445L
        resumeTopActivityLocked(null); //2562p 定义：2176L
        if(next=NULL)
        {
          intent.addCategory(Intent.CATEGORY_HOME);
          startActivityLocked(null, intent, null, null, 0, aInfo,null, null, 0, 0, 0, false, false);
        }
        else
        {
          startSpecificActivityLocked(next, true, false); //2439L 定义：1628L
            realStartActivityLocked() //1640L      定义：1524L
            //1651L 定义：1654L
            startProcessLocked(r.processName, r.info.applicationInfo, true, 0,"activity", r.intent.getComponent());
              //1717L 定义：1721L
              startProcessLocked(app, hostingType, hostingNameStr);
                //1768L  定义：Process.java 222L(frameworks/base/core/java/android/os)
                int pid = Process.start("android.app.ActivityThread",...)
                  startViaZygote(processClass, niceName, uid, gid, gids,debugFlags, zygoteArgs);
                    pid = zygoteSendArgsAndGetPid(argsForZygote);
                      sZygoteWriter.write(Integer.toString(args.size()));
```

```
        }

  runSelectLoopMode();
    done = peers.get(index).runOnce();
      forkAndSpecialize()     //Zygote.java (dalvik/libcore/dalvik/src/main/java/dalvik/system)
        Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c (dalvik/vm/native)
          forkAndSpecializeCommon()
            setSignalHandler()
          RETURN_INT(pid);


        ActivityThread main()
          ActivityThread attach() //ActivityThread.java 3870p (frameworks/base/core/java/android/app)
            mgr.attachApplication(mAppThread)
            //ActivityManagerService.java 4677p (frameworks/base/services/java/com/android/server/am)
            attachApplication()
              //ActivityManagerService.java 4677p (frameworks/base/services/java/com/android/server/am)
              attachApplicationLocked()
                if (realStartActivityLocked(hr, app, true, true)) //ActivityManagerService.java 4609p
(frameworks/base/services/java/com/android/server/am)
                realStartActivityLocked()
                  //ActivityManagerService.java (frameworks/base/services/java/com/android/server/am)
                  app.thread.scheduleLaunchActivity(new Intent(r.intent), r,r.info, r.icicle, results,
newIntents, !andResume,isNextTransitionForward());
                  scheduleLaunchActivity()
                    queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
                      ActivityThread.H.handleMessage()
                        handleLaunchActivity()      //ActivityThread.java (frameworks/base/core/java/android/app)
                          performLaunchActivity() //ActivityThread.java (frameworks/base/core/java/android/app)
                            activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);


init 守护进程：
//andriod init 函数启动过程分析：
在 main 循环中会重复调用
drain_action_queue();
restart_processes();
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
通过循环检测服务列表 service_list 中每个服务的 svc->flags 标记，如果为 SVC_RESTARTING，
```

那么在满足条件的情况下调用：restart_service_if_needed
通过 service_start 来再次启动该服务。

ActivityManagerService.main

I/SystemServer(45): Starting Power Manager.
I/ServiceManager(26): service 'SurfaceFlinger' died
D/Zygote (30): Process 45 terminated by signal (11)
I/Zygote (30): Exit zygote because system server (45) has terminated
通过错误信息发现程序在调用 SurfaceFlinger 服务的时候被中止。
Service_manager.c (frameworks/base/cmds/servicemanager):
LOGI("service '%s' died/n", str8(si->name));
Binder.c (frameworks/base/cmds/servicemanager):
death->func(bs, death->ptr);

Binder.c (kernel/drivers/misc)中的函数
binder_thread_read()
struct binder_work *w;
switch (w->type)
为 BINDER_WORK_DEAD_BINDER 的时候
binder_parse()中
当 cmd 为 BR_DEAD_BINDER 的时候
执行 death->func(bs, death->ptr)
因为函数
int do_add_service(struct binder_state *bs,
uint16_t *s, unsigned len,
void *ptr, unsigned uid)
的 si->death.func = svcinfo_death;
所以 death->func(bs, death->ptr) 实际上执行的是
svcinfo_death()//Service_manager.c (frameworks/base/cmds/servicemanager)
所以会打印出：service 'SurfaceFlinger' died
I/ServiceManager(26): service 'SurfaceFlinger' died

Thread::run
_threadLoop() // Threads.cpp (frameworks/base/libs/utils)
status_t SurfaceFlinger::readyToRun()
mBootAnimation = new BootAnimation(this);

1.5 Android 启动- init 守护进程分析

在 Android 系统启动时，内核引导参数上一般都会设置“init=/init”,这样的话，如果内核成功挂载了这个文件系统之后，首先运行的就是这个根目录下的 init 程序。这个程序所了什么呢？ 我们只有 RFSC（Readthe Fucking Source code）!!

init 程序源码在 Android 官方源码的 system/core/init 中，main 在 init.c 里。我们的分析就从 main 开始。

init：

（1）安装 SIGCHLD 信号。（如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。因此需要对 SIGCHLD 信号做出处理，回收僵尸进程的资源，避免造成不必要的资源浪费。

（2）对 umask 进行清零。

何为 umask，请看 http://www.szstudy.cn/showArticle/53978.shtml

（3）为 rootfs 建立必要的文件夹，并挂载适当的分区。

/dev （tmpfs）

/dev/pts (devpts)

/dev/socket

/proc (proc)

/sys (sysfs)

（4）创建/dev/null 和/dev/kmsg 节点。

（5）解析/init.rc，将所有服务和操作信息加入链表。

（6）从/proc/cmdline 中提取信息内核启动参数,并保存到全局变量。

（7）先从上一步获得的全局变量中获取信息硬件信息和版本号，如果没有则从/proc/cpuinfo 中提取,并保存到全局变量。

（8）根据硬件信息选择一个/init.(硬件).rc，并解析，将服务和操作信息加入链表。

在 G1 的 ramdisk 根目录下有两个/init.(硬件).rc：init.goldfish.rc 和 init.trout.rc，init 程序会根据上一步获得的硬件信息选择一个解析。

（9）执行链表中带有“early-init”触发的的命令。

（10）遍历/sys 文件夹，是内核产生设备添加事件（为了自动产生设备节点)。

（11）初始化属性系统，并导入初始化属性文件。

（12）从属性系统中得到 ro.debuggable，若为 1，则初始化 keychord 监听。

（13）打开 console,如果 cmdline 中没有指定 console 则打开默认的/dev/console。

（14）读取/initlogo.rle（一张 565 rle 压缩的位图），如果成功则在/dev/graphics/fb0 显示 Logo,如果失败则将/dev/tty0 设为 TEXT 模式并打开/dev/tty0,输出文本“ANDROID”字样。

（15）判断 cmdline 中的参数，并设置属性系统中的参数:

1、 如果 bootmode 为

- factory,设置 ro.factorytest 值为 1
- factory2,设置 ro.factorytest 值为 2
- 其他的设 ro.factorytest 值为 0

2、如果有 serialno 参数，则设置 ro.serialno，否则为""

3、如果有 bootmod 参数，则设置 ro.bootmod，否则为"unknown"

4、如果有 baseband 参数，则设置 ro.baseband，否则为"unknown"

5、如果有 carrier 参数，则设置 ro.carrier，否则为"unknown"

6、如果有 bootloader 参数，则设置 ro.bootloader，否则为"unknown"

7、通过全局变量（前面从/proc/cpuinfo 中提取的）设置 ro.hardware 和 ro.version。

（16）执行所有触发标识为 init 的 action。

（17）开始 property 服务，读取一些 property 文件，这一动作必须在前面那些 ro.foo 设置后做，以便 /data/local.prop 不能干预到他们。

- /system/build.prop
- /system/default.prop
- /data/local.prop
- 在读取默认的 property 后读取 presistent propertie，在/data/property 中

（18）为 sigchld handler 创建信号机制。

（19）确认所有初始化工作完成：

device_fd(device init 完成)

property_set_fd(property server start 完成)

signal_recv_fd (信号机制建立)

（20） 执行所有触发标识为 early-boot 的 action

（21） 执行所有触发标识为 boot 的 action

（22）基于当前 property 状态，执行所有触发标识为 property 的 action

（23）注册轮询事件:

- device_fd
- property_set_fd
-signal_recv_fd
-如果有 keychord，则注册 keychord_fd

（24）如果支持 BOOTCHART,则初始化 BOOTCHART

（25）进入主进程循环:

- 重置轮询事件的接受状态，revents 为 0
- 查询 action 队列，并执行。
- 重启需要重启的服务
- 轮询注册的事件
  - 如果 signal_recv_fd 的 revents 为 POLLIN，则得到一个信号，获取并处理
  - 如果 device_fd 的 revents 为 POLLIN,调用 handle_device_fd
  - 如果 property_fd 的 revents 为 POLLIN,调用 handle_property_set_fd
  - 如果 keychord_fd 的 revents 为 POLLIN,调用 handle_keychord

1.6 Android 启动- init 启动后的一些步骤

对于关注 Android 底层的朋友来说，其具体的启动过程应该是比较吸引我们的。但是很多启动文件什么的，都得 adb push 到 host 上来看，挺不方便的，都怪 Android 自带的 Toolbox 太简略了。所以在深入了解 Android 的启动流程之前，我们来把 Busybox 安装到 Android 上去，这样，就有很多工具供我们使用了。

首先去 busybox 主页 下载最新版本的源代码，然后用 arm 的交叉编译器编译出 busybox 的可执行程序，编译的时候需要注意一些设置选项，例如

Build Options --->

Build BusyBox as a static binary (no shared libs) 这个要选上，因上这样子编译出来的 busyBox 才是可以独立运行的。

| Do you want to build BusyBox with a Cross Compiler? | |

| |(/HOME/toolchains/gcc-4.0.2-glibc-2.3.5/arm-9tdmi-linux-gnu/bin/arm-9tdmi-linux-gnu| 这是交叉编译器的路径，要根据具体的情况来设置。

Installation Options --->

Don't use /usr

这样子编译出来的 busybox 才不会安装到你主机的/usr 目录下。一定要选上。

busybox 的功能选项根据需要自选,但是不要太贪心.

OK，这里就不纠缠于编译 busybox 的东西了，网上资料无数。接下来，我们把 busybox 安装到模拟器上去。先在模拟器上随便建一个 busybox 的文件夹，然后进入 busybox 可执行文件目录，使用命令

adb push busybox.asc /data/busybox/busybox

然后进入 adb shell，chmod 777 ./busybox，就可以直接使用了。但现在还是不方便，总不能每用一个命令就输一次 busybox 吧？所以，我们可以先用./busybox --install 将程序都安装到当前目录下，然后把当前目录添加到 PATH 变量中即可。暂时使用 export 来添加吧，如果想永久添加，往下看。

好了，准备工作完成，开始研究的工作了。既然是研究启动过程，那当然是先看看 init.rc 文件。去 etc 目录打开它，分析一下内容，首先是对 env 的定义，也就是全局环境变量的定义，接下来的建立和初始化里面的内容目前还不清楚什么意思，紧接着就是系统启动时运行的初始进程信息，这个比较有意思，包括了 usbd-config 和 qemu，qemu 自不用说，而 usbd-config 作为初始启动的进程，应该就是和上一篇文章猜的一样，用来调试或者 usb 通信的。往下看，是在初始启动进程完成之后开始启动的服务进程，这些进程如果因故退出，会自动重启。这里面包括了 console 控制台，adbd 监护进程，usbd 监护进程，debuggerd 监护进程等.除去这些守护进程，能引起我们注意的，是 runtime 和 zygote。这两个进程似乎掌管着其他进程以及应用程序的启动。

现在，来让我们做一个实验吧，将自动调用的启动过程变成手动，看看启动流程具体是什么样的。想达到这个目的，首先就是要修改 init.rc 文件，当然不是在模拟器的 console 中改，一是不能改，二是你改了也没用，下次加载就会给你覆盖了。所以，我们要从原始镜像 ramdisk.img 入手了。从 2.6 标准 Linux 内核开始，initrd.img 都采用 cpio 压缩，猜测 ramdisk.img 也一样，需要使用 gunzip 解压缩，然后再使用 cpio 解包。好，进入 tools/lib/images 目录下，先用 file 命令看看 ramdisk.img 的类型，没错，系统提示

ramdisk.img: gzip compressed data, from Unix

很好，然后将 ramdisk.img 复制一份到任何其他目录下，将其名称改为 ramdisk.img.gz，并使用命令

gunzip ramdisk.img.gz

然后新建一个文件夹，叫 ramdisk 吧，进入，输入命令

cpio -i -F ../ramdisk.img

这下，你就能看见并操作 ramdisk 里面的内容了。当然你也可以直接在外面进行操作，但是还是建议把 cpio

解压缩出来的内容全部集中在一个文件夹里面，因为一会我们还要将其压缩成新的 ramdisk.img。

OK，现在开始修改步骤吧。用任何一款编辑器打开 init.rc，首先在 PATH 那里加上你的 Busybox 安装路径，然后注释内容，我们要手工启动他们。

```
#       zygote {
#               exec /system/bin/app_process
#               args {
#                       0 -Xzygote
#                       1 /system/bin
#                       2 --zygote
#               }
#               autostart 1
#       }

#       runtime {
#               exec /system/bin/runtime
#               autostart 1
#       }
```

在这里需要注意，不要同时把两者都注释了，注释某一个，再试验手工启动它，如果两者同时注释我这里有问题，无法启动。

好，接下来，使用下列命令重新打包成镜像

cpio -i -t -F ../ramdisk.img > list

cpio -o -H newc -O lk.img < list

当前目录下生成的 lk.img 就是我们的新镜像了。使用自己的镜像启动 emulator；

emulator -console -ramdisk lk.img

如果我们注释的是 zygote，那么在#后输入

app_process -Xzygote /system/bin --zygote

手工启动，命令行中输出的信息是

Prepping:
/system/app/AlarmProvider.apk:/system/app/Browser.apk:/system/app/Calendar.apk:/system/app/Camera.apk:/system/app/Contacts.apk:
/system/app/Development.apk:/system/app/GDataFeedsProvider.apk:/system/app/Gmail.apk:/system/app/GmailProvider.apk:/system/app/GoogleApps.apk:
/system/app/GoogleAppsProvider.apk:/system/app/Home.apk:/system/app/ImProvider.apk:/system/app/Maps.apk:/system/app/MediaPickerActivity.apk:
/system/app/MediaProvider.apk:/system/app/Phone.apk:/system/app/PimProvider.apk:/system/app/ApiDemos.apk:/system/app/SettingsProvider.apk:
/system/app/Sms.apk:/system/app/SyncProvider.apk:/system/app/TelephonyProvider.apk:/system/app/XmppService.apk:/system/app/YouTube.apk
File not found: /system/app/AlarmProvider.apk
File not found: /system/app/Calendar.apk
File not found: /system/app/Camera.apk
File not found: /system/app/GDataFeedsProvider.apk

File not found: /system/app/Gmail.apk
File not found: /system/app/GmailProvider.apk
File not found: /system/app/MediaPickerActivity.apk
File not found: /system/app/PimProvider.apk
File not found: /system/app/ApiDemos.apk
File not found: /system/app/Sms.apk
File not found: /system/app/SyncProvider.apk
File not found: /system/app/YouTube.apk
Prep complete

嘿嘿，从 File not found 的信息中可以看到一些 Google 可能会即将推出的应用，比如 Gmail 什么的。如果我们注释的是 runtime，那么输出信息是：

+++ post-zygote

1.7 Android 启动- build 过程

首先下载下 android 源码并编译，网上的资料特别多。按照网站上的步骤，将 android 内核编译成功，如果不出意外的话，在 out/target/product/generic 目录下会生成三个文件，分别是 ramdisk.img、sytem.img、userdata.img，这三个文件到底有什么用呢？下面开始分析一下。

首先在 linux 终端下使用命令 file ramdisk.img，打印出如下字符 ramdisk.img: gzip compressed data, from Unix，可以看出，它是一个 gzip 压缩的格式，下面对其进行解压，使用 fedora 自带的工具进行解压，或者使用 gunzip 进行解压（可能需要将扩展名改为.gz），可以看到解压出一个新的 ramdisk.img,这个 ramdisk.img 是使用 cpio 压缩的，可以使用 cpio 命令对其进行解压,cpio –i –F ramdisk.img，解压后可以看到生成了一些文件夹和文件。看到这些文件就会明白，它和 root 目录下的内容完全一样。说明了 ramdisk.img 其实是对 root 目录的打包和压缩。

下面分析 system.img 的来源。在 build/core/Makefile 里的 629 行，可以看到这么一段文字

```
# The installed image, which may be optimized or unoptimized.
#
INSTALLED_SYSTEMIMAGE := $(PRODUCT_OUT)/system.img
```

从这里可以看出，系统应该会在$(PRODUCT_OUT)目录下生成 system.img

再继续往下看，在 662 行有一个 copy-file-to-target，这实现了将 system.img 从一个中间目录复制到/generic 目录。BUILD_SYSTEM 的定义在 636 行。

这里的 system.img 不是/generic 目录下面我们看到的那个 system.img，而是另一个中间目录下的，但是是同一个文件。一开始看到的复制就是把 out /target/product/generic/obj/PACKAGING/systemimage_unopt_intermediates 目录下面的 system.img 复制到/generic 目录下。

现在，知道了 system.img 的来历，然后要分析它是一个什么东西，里面包含什么？？

Makefile line624

```
$(BUILT_SYSTEMIMAGE_UNOPT): $(INTERNAL_SYSTEMIMAGE_FILES) $(INTERNAL_MKUSERFS)
$(call build-systemimage-target,$@)
```

这里调用了 build-systemimg-target Makefile line605

```
ifeq ($(TARGET_USERIMAGES_USE_EXT2),true)
## generate an ext2 image
# $(1): output file
define build-systemimage-target
@echo "Target system fs image: $(1)"
$(call build-userimage-ext2-target,$(TARGET_OUT),$(1),system,)
endef
else # TARGET_USERIMAGES_USE_EXT2 != true
## generate a yaffs2 image
# $(1): output file
define build-systemimage-target
@echo "Target system fs image: $(1)"
@mkdir -p $(dir $(1))
＊$(hide) $(MKYAFFS2) -f $(TARGET_OUT) $(1)＊
endef
endif # TARGET_USERIMAGES_USE_EXT2
```

找不到TARGET_USERIMAGES_USE_EXT2 的定义!!不过从上面的分析可以推断出应该是yaffs2 文件系统。

其中 MKYAFFS2：（core/config.mk line161）

MKYAFFS2 := $(HOST_OUT_EXECUTABLES)/mkyaffs2image$(HOST_EXECUTABLE_SUFFIX)

定义 MKYAFFS2 是目录/media/disk/mydroid /out/host/linux-x86/bin 下的一个可执行文件 mkyaffs2image，运行这个程序可得到如下信息：

lzj@lzj-laptop:/media/disk/mydroid/out/host/linux-x86/bin$ ./mkyaffs2image
mkyaffs2image: image building tool for YAFFS2 built Nov 13 2009
usage: mkyaffs2image [-f] dir image_file [convert]
-f fix file stat (mods, user, group) for device
dir the directory tree to be converted
image_file the output file to hold the image
'convert' produce a big-endian image from a little-endian machine

得知这个程序可以生成 yaffs2 的文件系统映像。并且也清楚了上面＊$(hide) $(MKYAFFS2) -f $(TARGET_OUT) $(1)＊的功能，把 TARGET_OUT 目录转变成 yaffs2 格式并输出成/media/disk/mydroid/out/target /product/generic/obj/PACKAGING/systemimage_unopt_intermediates /system.img(也就是我们最终在/generic 目录下看到的那个 system.img)。

到现在已经差不多知道 system.img 的产生过程，要弄清楚 system.img 里面的内容，就要分析 TARGET_OUT 目录的内容了。（想用 mount 把 system.img 挂载到 linux 下面看看里面什么东西，却不支持 yaffs 和 yaffs2 文件系统!!!）

下一步：分析 TARGET_OUT 在 build/core/envsetup.sh 文件（line205）中找到了 TARGET_OUT 的定义：

TARGET_OUT := $(PRODUCT_OUT)/system

也就是/media/disk/mydroid/out/target /product/generic 目录下的 system 目录。

lzj@lzj-laptop:/media/disk/mydroid/out/target/product/generic/system$ tree -L 1
.
|-- app
|-- bin
|-- build.prop
|-- etc
|-- fonts
|-- framework
|-- lib
|-- usr
`-- xbin

现在一切都明白了，我们最终看到的 system.img 文件是该目录下的 system 目录的一个映像，类似于 linux 的根文件系统的映像，放着 android 的应用程序，配置文件，字体等。

Userdata.img 来来自于 data 目录，默认里面是没有文件的。

【Android 启动流程分析一些案例】

2.1 Android 启动界面实现

启动界面的主要功能就是显示一幅启动图像,后台进行系统初始化. 如果是第一次使用本程序,需要初始化本程序的 sqlite 数据库,建库,建 Table,初始化账目数据. 如果不是第一次使用,就进入登记收支记录界面.

界面效果如图:

界面很简单,一个 imageview 和一个 textview

可是如何是 2 个 view 垂直居中显示,我开始使用 linearlayout 就没法完成垂直和横向居中. 后来使用 RelativeLayout 才搞定了横向居中.

界面的具体 xml 如下:

main.xml

```
< RelativeLayout android:id = "@+id/RelativeLayout01" xmlns:android = "http://schemas.android.com/apk/res/android"
android:layout_gravity = "center_vertical|center_horizontal"
android:layout_height = "wrap_content"
android:layout_width = "wrap_content" >
< ImageView android:id = "@+id/ImageView01"
android:src = "@drawable/logo3"
android:layout_width = "wrap_content"
```

```
android:layout_height = "wrap_content" >

< TextView android:id = "@+id/TextView01"
android:text = "@string/welcome"
android:layout_below = "@id/ImageView01"
android:layout_width = "wrap_content"
android:layout_height = "wrap_content" >
```

在这儿我来使用一个小技巧,就是在程序初始化完成后,让图片淡出,然后显示下一个界面.

开始我准备使用一个 timer 来更新图片的 alpha 值,后来程序抛出异常 Only the original thread that created a view hierarchy can touch its views.

这才发现 android 的 ui 控件是线程安全的.这里需要我们在主线程外,再开一个线程更新界面上的图片.可以使用 imageview.invalidate

关于如何另开一个线程更新界面的相关代码如下.

```
//给主线程发送消息更新 imageview
mHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
super .handleMessage(msg);
imageview.setAlpha(alpha);
imageview.invalidate();
}
};
new Thread( new Runnable() {
public void run() {
while (b < 2 ) {
try {
//延时 2 秒后,每 50 毫秒更新一次 imageview
if (b == 0 ) {
Thread.sleep( 2000 );
b = 1 ;
} else {
Thread.sleep( 50 );
}
updateApp();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}).start();
public void updateApp() {
alpha -= 5 ; //每次减少 alpha 5
```

```
if (alpha <= 0 ) {
b = 2 ;
Intent in = new Intent( this , com.cola.ui.Frm_Addbills. class );
startActivity(in); //启动下个界面
}
mHandler.sendMessage(mHandler.obtainMessage());
}
```

通过这段代码,我们能够理解 android 里面如何对 ui 视图进行更新.

下面我们来看看 sqlite 的使用.如何初始化程序.

附 ColaBox.java:

```
package com.cola.ui;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.view.KeyEvent;
import android.widget.ImageView;
import android.widget.TextView;
public class ColaBox extends Activity {
private Handler mHandler = new Handler();
ImageView imageview;
TextView textview;
int alpha = 255 ;
int b = 0 ;
public void onCreate(Bundle savedInstanceState) {
super .onCreate(savedInstanceState);
setContentView(R.layout.main);
imageview = (ImageView) this .findViewById(R.id.ImageView01);
textview = (TextView) this .findViewById(R.id.TextView01);
Log.v( "ColaBox" , "ColaBox start ..." );
imageview.setAlpha(alpha);
new Thread( new Runnable() {
public void run() {
initApp(); //初始化程序
while (b < 2 ) {
try {
if (b == 0 ) {
Thread.sleep( 2000 );
b = 1 ;
} else {
Thread.sleep( 50 );
}
```

```
updateApp();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}).start();
mHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
super .handleMessage(msg);
imageview.setAlpha(alpha);
imageview.invalidate();
}
};
}
public void updateApp() {
alpha -= 5 ;
if (alpha <= 0 ) {
b = 2 ;
Intent in = new Intent( this , com.cola.ui.Frm_Addbills. class );
startActivity(in);
}
mHandler.sendMessage(mHandler.obtainMessage());
}
public void initApp(){
}
}
```

2.2 创建 Android 启动界面

每个 Android 应用启动之后都会出现一个 Splash 启动界面，显示产品的 LOGO、公司的 LOGO 或者开发者信息。如果应用程序启动时间比较长，那么启动界面就是一个很好的东西，可以让用户耐心等待这段枯燥的时间。

①制作 Splash 界面

突出产品 LOGO，产品名称，产品主要特色；注明产品的版本信息；注明公司信息或者开发者信息；背景图片，亦可以用背景颜色代替；

②除了等待还能做点什么

大多数的 Splash 界面都是会等待一定时间，然后切换到下一个界面；其实，在这段时间里，可以对系统状况进行检测，比如网络是否通，电源是否充足；或者，预先加载相关数据；为了能让启动界面展现时间固定，需要计算执行以上预处理任务所花费的时间，那么：启动界面 SLEEP 的时间=固定时间-预处理任务时间

源码示例（以 Wordpress 的 Android 客户端为例）

AndroidMenifest.xml

```
<activity android:icon="@drawable/app_icon"
 android:screenOrientation="portrait"
 android:name=".splashScreen"
 android:theme="@android:style/Theme.NoTitleBar">
 <intent-filter>
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
 </activity>
```

splashScreen.java

```
package org.wordpress.android;
import android.app.Activity;import android.content.Intent;import
android.content.pm.PackageInfo;import android.content.pm.PackageManager;import
android.content.pm.PackageManager.NameNotFoundException;import
android.graphics.PixelFormat;import android.os.Bundle;import android.os.Handler;import
android.view.WindowManager;import android.widget.TextView;

public class splashScreen extends Activity {
 /**
 * Called when the activity is first created.
 */

 @Override
 public void onCreate(Bundle icicle) {
 super.onCreate(icicle);
 getWindow().setFormat(PixelFormat.RGBA_8888);
 getWindow().addFlags(WindowManager.LayoutParams.FLAG_DITHER);

 setContentView(R.layout.splashscreen);
```

```
//Display the current version number PackageManager pm = getPackageManager();
try {
PackageInfc pi = pm.getPackageInfo("org.wordpress.android", 0);
TextView versionNumber = (TextView) findViewById(R.id.versionNumber);
versionNumber.setText("Version " + pi.versionName);
} catch (NameNotFoundException e) {
e.printStackTrace();
}

new Handler().postDelayed(new Runnable() {
public void run() {
/* Create an Intent that will start the Main WordPress Activity. */
Intent mainIntent = new Intent(splashScreen.this, wpAndroid.class);
splashScreen.this.startActivity(mainIntent);
splashScreen.this.finish();
}
}, 2900); //2900 for release
}
}
```

splashscreen.xml

```
<!--
android:gravity是对元素本身说的，元素本身的文本显示在什么地方靠着换个属性设置，不过不设置默认是在左侧的。

android:layout_gravity是相对与它的父元素说的，说明元素显示在父元素的什么位置--><LinearLayout
android:id="@+id/LinearLayout01"
 android:layout_width="fill_parent"
 android:layout_height="fill_parent"
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:gravity="center|center"
 android:background="@drawable/home_gradient"
 android:orientation="vertical">
 <!--
 android:scaleType是控制图片如何resized/moved来匹对ImageView的size
 CENTER_INSIDE / centerInside 将图片的内容完整居中显示，通过按比例缩小或原来的size使得图片长/宽等于或小于View的长/宽
 -->
 <ImageView android:layout_marginTop="-60dip"
 android:paddingLeft="20dip"
 android:paddingRight="20dip"
 android:scaleType="centerInside"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:id="@+id/wordpress_logo"
```

```
android:src="@drawable/wordpress_home">
</ImageView>
<!--
android:typeface 字体风格
-->
<TextView android:text="@+id/TextView01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="20dip"
android:typeface="serif"
android:shadowDx="0"
android:shadowDy="2"
android:shadowRadius="1"
android:shadowColor="#FFFFFF"
android:textColor="#444444"
android:textSize="20dip"
android:id="@+id/versionNumber"
android:gravity="bottom">
</TextView></LinearLayout>
```

2.3 制作 Android 启动界面

作为一个准程序猿，特别是写 Java 代码的程序猿，怎么说也要写一篇开发经验吧。于是立马下载了 Android SDK，啃了三天的 Android Development Guide，为以前做的一个 Web 小应用开发了 Android 版本（当然只实现了最基本的功能，相当于一个 demo ）。

本来打算写一篇文章来整体介绍这个应用的开发过程的，不过其复杂程度其实已经超过了一篇中短篇博文的信息负载能力了。所以不如分开来写，第一篇就挑最开始的这个部分，也是相当简单的一个部分来写吧——制作一个启动界面（ Splash screen ）。

先上效果图吧。

上图就是启动界面了，停留一段时间后，自动进入主界面：

好了，效果就是这么简单。下面具体来说怎样实现。

有用户界面，自然离不开 Activity。有 Activity，自然离不开 Layout。我们使用一个 XML 文件来声明一个 RelativeLayout,这个 XML 文件除了能创建 Activity 的组件之外,还能对组件的样式进行定制(有点像 CSS 的赶脚)。我们把这个文件命名为 welcome.xml，其内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:weightSum="1"
    android:background="#FFFFFF">
        <ImageView android:layout_height="wrap_content"
         android:layout_width="match_parent"
         android:src="@drawable/logo"
         android:id="@+id/imageView1"
         android:layout_alignParentTop="true"
         android:layout_alignParentLeft="true"
         android:layout_marginTop="64dp">
        </ImageView>
        <TextView android:text="@string/xnuol"
         android:layout_height="wrap_content"
         android:textAppearance="?android:attr/textAppearanceMedium"
         android:layout_width="wrap_content"
         android:textColor="#777777"
         android:id="@+id/textView1"
         android:layout_alignParentBottom="true"
         android:layout_centerHorizontal="true"
         android:layout_marginBottom="34dp">
        </TextView>
        <TextView android:text="@string/ver"
         android:layout_height="wrap_content"
         android:textAppearance="?android:attr/textAppearanceSmall"
         android:layout_width="wrap_content"
         android:id="@+id/textView2"
         android:layout_alignParentBottom="true"
         android:layout_centerHorizontal="true"
         android:layout_marginBottom="14dp">
        </TextView>
</RelativeLayout>
```

可以看到，XML 中声明了一个 RelativeLayout、一个 ImageView 和两个 TextView。其中 ImageView 就是用来展现 Logo 的组件， 两个 TextView 用来展示版权和版本信息。有 Layout 的 XML，我们就可以创建一个 Activity 类了。把这个类命名为 WelcomeActivity，它继承 *android.app.Activity*，完整内容如下：

```java
package com.psjay.campus_clock;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
/**
 * 校园时钟欢迎界面
 * @author PSJay
 *
 */
public class WelcomeActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.welcome);

        Thread thread = new Thread() {
            @Override
            public void run() {
                int waitingTime = 3000; // ms
                try {
                    while(waitingTime > 0) {
                            sleep(100);
                            waitingTime -= 100;    // 100ms per time
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    finish();
                    Intent intent = new Intent(WelcomeActivity.this, MainActivity.class);
                    startActivity(intent);   // enter the main activity finally
                }
            }
        };

        thread.start();
    }
}
```

重点就是 onCreate() 这个方法中的那个匿名内部类线程了。waitingTime 变量表示需要等待的时间，单位是毫秒。循环中线程让自己休眠，最后结束 Activity ，创建一个 Intent 对象，调用 startActivity() 方法跳转到主界面，这样，就完整了启动界面的整个流程了，so easy。

2.4 Android 开机动画过程

Android 开机会出现 3 个画面：

①Linux 系统启动，出现 Linux 小企鹅画面(reboot)(Android 1.5 及以上版本已经取消加载图片)；

②Android 平台启动初始化，出现"A N D R I O D"文字字样画面；

③Android 平台图形系统启动，出现含闪动的 ANDROID 字样的动画图片(start)。

2.4.1 开机图片(Linux 小企鹅) (Android 1.5 及以上版本已经取消加载图片)；

Linux Kernel 引导启动后，加载该图片。

logo.c 中定义 nologo,在 fb_find_logo(int depth)函数中根据 nologo 的值判断是否需要加载相应图片。

代码如下：

```
static int nologo;
module_param(nologo, bool, 0);
MODULE_PARM_DESC(nologo, "Disables startup logo");
/* logo's are marked __initdata. Use __init_refok to tell
* modpost that it is intended that this function uses data
* marked __initdata.
*/
const struct linux_logo * __init_refok fb_find_logo(int depth)
{
const struct linux_logo *logo = NULL;
if (nologo)
return NULL;
        ……
}
```

相关代码：

/kernel/drivers/video/fbmem.c

/kernel/drivers/video/logo/logo.c

/kernel/drivers/video/logo/Kconfig

/kernel/include/linux/linux_logo.h

2.4.2 开机文字("A N D R I O D")

Android 系统启动后，init.c 中 main()调用 load_565rle_image()函数读取/initlogo.rle（一张 565 rle 压缩的位图），如果读取成功，则在/dev/graphics/fb0 显示 Logo 图片；如果读取失败，则将/dev/tty0 设为 TEXT 模式， 并打开/dev/tty0，输出文本“A N D R I O D”字样。

定义加载图片文件名称

```
#define INIT_IMAGE_FILE "/initlogo.rle"
int load_565rle_image( char *file_name );
#endif
```

init.c 中 main()加载/initlogo.rle 文件。

```
if( load_565rle_image(INIT_IMAGE_FILE) ) {//加载 initlogo.rle 文件
    fd = open("/dev/tty0", O_WRONLY);//将/dev/tty0 设为 text 模式
    if (fd >= 0) {
```

```
        const char *msg;
            msg = "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"    // console is 40 cols x 30 lines
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "                    A N D R O I D ";
        write(fd, msg, strlen(msg));
        close(fd);
    }
}
```

相关代码：

/system/core/init/init.c

/system/core/init/init.h

/system/core/init/init.rc

/system/core/init/logo.c

*.rle 文件的制作步骤:

a. 使用 GIMP 或者 Advanced Batch Converter 软件，将图象转换为 RAW 格式；

b. 使用 android 自带的 rgb2565 工具，将 RAW 格式文件转换为 RLE 格式(如：rgb2565 -rle < initlogo.raw > initlogo.rle)。

2.4.3 开机动画(闪动的 ANDROID 字样的动画图片)

①Android 1.5 版本：Android 的系统登录动画类似于 Windows 系统的滚动条，是由前景和背景两张 PNG 图片组成，这两张图片存在于手机或模拟器 /system/framework /framework-res.apk 文件当中，对应原文件位于 /frameworks/base/core/res/assets/images/。前 景图片（android-logo-mask.png）上的 Android 文字部分镂空，背景图片（android-logo-shine.png）则是 简单的纹理。系统登录时，前景图片在最上层显示，程序代码（BootAnimation.android()）控制背景图片连续滚动，透过前景图片文字镂 空部分滚动显示背景纹理，从而实现动画效果。

相关代码：

/frameworks/base/libs/surfaceflinger/BootAnimation.h

/frameworks/base/libs/surfaceflinger/BootAnimation.cpp

/frameworks/base/core/res/assets/images/android-logo-mask.png Android 默认的前景图片，文字部分镂空，大小 256×64

/frameworks/base/core/res/assets/images/android-logo-shine.png Android 默认的背景图片，有动感效果，大小 512×64

②Android 1.6 及以上版本：

i nit.c 解析 init.rc（其中定义服务："service bootanim /system/bin/bootanimation"），bootanim 服务由 SurfaceFlinger.readyToRun()（property_set("ctl.start", "bootanim");）执行开机动画、bootFinished()（property_set("ctl.stop", "bootanim");）执行停止开机动画。

BootAnimation.h 和 BootAnimation.cpp 文件放到了/frameworks/base/cmds /bootanimation 目录下了，增加了一个入口文件 bootanimation_main.cpp。Android.mk 文件中可以看到，将开机 动画从原来的 SurfaceFlinger 里提取出来了，生成可执行文件：bootanimation。Android.mk 代码如下：

```
//==============Android.mk=======================
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
    bootanimation_main.cpp \
    BootAnimation.cpp
# need "-lrt" on Linux simulator to pick up clock_gettime
ifeq ($(TARGET_SIMULATOR),true)
    ifeq ($(HOST_OS),linux)
        LOCAL_LDLIBS += -lrt
    endif
endif
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libui \
    libcorecg \
    libsgl \
    libEGL \
    libGLESv1_CM \
    libmedia
LOCAL_C_INCLUDES := \
    $(call include-path-for, corecg graphics)
LOCAL_MODULE:= bootanimation
include $(BUILD_EXECUTABLE)
//===============================================
```

（1）adb shell 后，可以直接运行"bootanimation"来重新看开机动画，它会一直处于动画状态，而不会停止。

（2）adb shell 后，命令"setprop ctl.start bootanim"执行开机动画；命令"getprop ctl.start bootanim"停止开机动画。

这两句命令分别对应 SurfaceFlinger.cpp 的两句语 句：property_set("ctl.start", "bootanim");和 property_set("ctl.stop", "bootanim");

相关文件：

/frameworks/base/cmds/bootanimation/BootAnimation.h

/frameworks/base/cmds/bootanimation/BootAnimation.cpp

/frameworks/base/cmds/bootanimation/bootanimation_main.cpp

/system/core/init/init.c

/system/core/rootdir/init.rc

【其它相关材料】

3.1 以 HTC Wildfire 为例讲解 Android 的几种启动模式

据我对 Android 的理解，Android 系统大致支持这几种启动模式：Normal, Bootloader, Fastboot, Recovery, Factory(其中 Bootloader 和 Fastboot 也可以归为一类)，这些命名是我自己根据实际情况起的名字。这里先简单介绍一下这几种模式：

1. Normal：即正常启动到 OS 的模式，对于不同的 Android 设备来说，一般都是按下电源键开机即可。

2. Bootloader：即启动到 Bootloader。对于 Linux 来说，目前最常见的 Bootloader 是 GRUB，对 Android OS 来说，常见的 Bootloader 是 uBoot，不过不同的公司会使用自己的 Bootloader，比如 HTC 的 Bootloader 是 HBoot。另外，Bootloader 一般会分为两阶段，IPL(Initial Program Loader)和 SPL(Second Program Loader)。

3. Fastboot：Fastboot 是 SPL 的一项特殊功能，即可以通过 USB 将 Android 设备与 Windows/Linux 主机连接起来，然后通过 Android SDK 的 tools 目录下的工具 Fastboot 来对 Android 设备进行一些操作，比如清除 userdata 分区(挂载点为/data)，cache 分区(挂载点为/data)，甚至是 boot 分区和 system 分区(挂载点为/system)，flash update.zip 更新文件和一些.img 文件。不过一般来说，即使支持 Fastboot 的 Android 设备，Fastboot 默认也是被锁住的。

4. Recovery：即启动到恢复模式，进入 Recovery 模式可以进行恢复出厂设置(所作操作为 format /data 和/cache 两个分区)，执行更新 update.zip(一般可以更新 Bootloader/boot/Recovery/system 这四种 image)

5. Factory：即进入工厂恢复模式，该模式一般用于重烧整个 ROM 上的 image。

我自己的手机是 HTC Wildfire(野火)，所以下面我会以 HTC Wildfire 为例来讲解这几种启动模式，下面先看一下 HTC Wildfire 的相关实体键：

HTC Wildfire 机身上有 4 个键，分别为电源键，音量调大键， 音量调小键以及光学键(也有人称轨迹球，拍照键)。

下面讲解 HTC Wildfire 如何进入不同启动模式(其它 Android 设备请自行摸索或者 Google)：

1. Normal:

HTC Wildfire 按一下电源键即可正常开机。

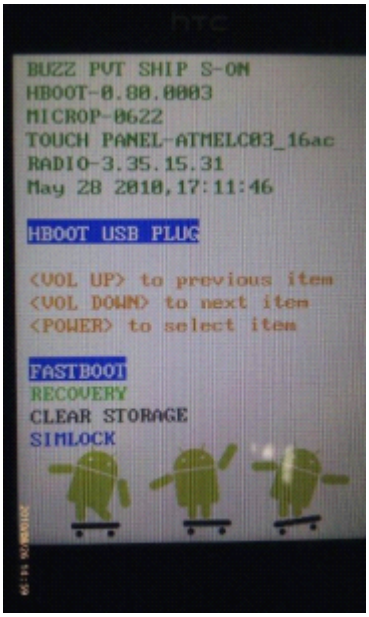
这里也提一下重启的方法：Wildfire 同时按下电源键+音量调小键+光学键可以重启。

另外，在 Windows/Linux 下，也可以借助 Android SDK 的 tools 目录下的工具 adb 来操作让 Android 设备重启，具体命令为”adb reboot”。

2. Bootloader:

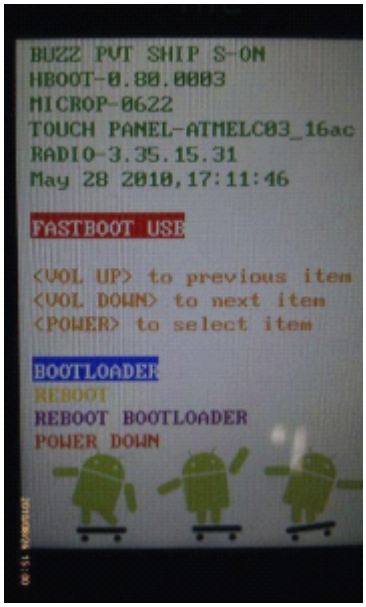
HTC Wildfire 同时按下电源键+音量调小键开机即可进入 Bootloader。下图为 HTC Wildfire 进入 Bootloader 后的画面，在该模式下，电源键相当于选择键，音量调大键相当于向上选择，音量调小键相当于向下选择。

另外，在 Windows/Linux 下，也可以借助 Android SDK 的 tools 目录下的工具 adb 来操作让 Android 设备进入 Bootloader，具体命令为”adb reboot Bootloader”。但是，HTC Wildfire 实际进入的是 Fastboot。



3. Fastboot:

HTC Wildfire 同时按下电源键+光学键开机即可进入 Fastboot。下图为 HTC Wildfire 进入 Fastboot 后的画面，在该模式下，电源键相当于选择键，音量调大键相当于向上选择，音量调小键相当于向下选择。第一行的”S-ON”代表 security-on，也就是 Fastboot 被锁住了。



4. Recovery:

HTC Wildfire 进入 Bootloader 后，按音量调小键向下选择”Recovery”，然后按下电源键即可进入 Recovery 模式。下图为 HTC Wildfire 进入 Recovery 模式的画面(已经 root 过，不是 Wildfire 进入 Recovery 模式的默认画面了)，在该模式下，电源键相当于返回键，音量调大键相当于向上选择，音量调小键相当于向下选择，光学键相当于确认。

另外，在 Windows/Linux 下，也可以借助 Android SDK 的 tools 目录下的工具 adb 来操作让 Android 设备进入 Bootloader，具体命令为”adb reboot Recovery”。

再另外，在 Recovery 模式下，ClockworkMod 的 Recovery image 可以使用 adb，应该是该 image 内置了 adbd

5. Factory:

HTC Wildfire 同时按下电源键+音量调大键即可进入工厂模式。进入该模式后，初始是黑屏的。

Ps:Wildfire 的这种模式是否就是工厂模式，目前我还有得到确认。不过有一个很有意思的实验，将 HTC

Wildfire 通过 USB 数据线跟 Linux 系统连接起来，然后在终端下使用”lsusb”命令来查看系统所认到的设备，USB 设备都会有对应的 Vendor ID 和 Device ID，分别指示设备制造商和设备 ID。”lsusb”的结果就会显示这两组 ID。将 HTC Wildfire 分别开启到这几种不同的模式，再来通过”lsusb”命令来查看这两组 ID 时，我们会看到以下结果：

a. Normal: 0BB4:0C8B
b. Bootloader: 0BB4:0C94
c. Fastboot: 0BB4:0FFF
d. Recovery: 0BB4:0C8B
e. Factory: 05c6:9002

可以看到，前四种模式下的 Vendor ID 都是 0BB4，而第五种模式下确实 05c6，这很有趣。事实上，0BB4 是 HTC 的 USB 设备 Vendor ID，而 05c6 则是 Qualcomm 的 USB 设备 Vendor ID(可以从文末我给的链接去证实)，而 HTC Wildfire 采用的就是 Qualcomm 的芯片。所以根据硬件 ID 信息我推测这第五种模式应该是工厂模式，用于重烧整个 ROM 上的 image。

3.2 Android 启动过程代码分析

开机过程中无线模块的初始化过程;如果 sim 卡锁开启，或者 pin 被锁住的时候，会要求输入 pin 或者 puk，但是这个解锁动作必须在系统初始化完成以后才能进行。(图形系统都还没有初始化怎么输入密码阿?)当系统初始化完成以后会调用 wm.systemReady()来通知大家。这时候该做什么就做什么。

开机过程中无线模块的初始化过程：

rild 调用参考实现 Reference-ril.c (hardware\ril\reference-ril) 中的函数：

```
const RIL_RadioFunctions *RIL_Init(const struct RIL_Env *env, int argc, char **argv)
ret = pthread_create(&s_tid_mainloop, &attr, mainLoop, NULL);
static void *mainLoop(void *param)
ret = at_open(fd, onUnsolicited);
RIL_requestTimedCallback(initializeCallback, NULL, &TIMEVAL_0);
```

在 initializeCallback 函数中对猫进行了初始化。

```
static void initializeCallback(void *param)
{
ATResponse *p_response = NULL;
int err;
setRadioState (RADIO_STATE_OFF);
at_handshake();
/* note: we don't check errors here. Everything important will
be handled in onATTimeout and onATReaderClosed */
/* atchannel is tolerant of echo but it must */
/* have verbose result codes */
at_send_command("ATE0Q0V1", NULL);
/* No auto-answer */
at_send_command("ATS0=0", NULL);
/* Extended errors */
at_send_command("AT+CMEE=1", NULL);
/* Network registration events */
err = at_send_command("AT+CREG=2", &p_response);
/* some handsets -- in tethered mode -- don't support CREG=2 */
if (err < 0 || p_response->success == 0) {
at_send_command("AT+CREG=1", NULL);
}
at_response_free(p_response);
/* GPRS registration events */
at_send_command("AT+CGREG=1", NULL);
/* Call Waiting notifications */
at_send_command("AT+CCWA=1", NULL);
/* Alternating voice/data off */
at_send_command("AT+CMOD=0", NULL);
/* Not muted */
at_send_command("AT+CMUT=0", NULL);
```

```
/* +CSSU unsolicited supp service notifications */
at_send_command("AT+CSSN=0,1", NULL);
/* no connected line identification */
at_send_command("AT+COLP=0", NULL);
/* HEX character set */
at_send_command("AT+CSCS=\"HEX\"", NULL);
/* USSD unsolicited */
at_send_command("AT+CUSD=1", NULL);
/* Enable +CGEV GPRS event notifications, but don't buffer */
at_send_command("AT+CGEREP=1,0", NULL);
/* SMS PDU mode */
at_send_command("AT+CMGF=0", NULL);
#ifdef USE_TI_COMMANDS
at_send_command("AT%CPI=3", NULL);
/* TI specific -- notifications when SMS is ready (currently ignored) */
at_send_command("AT%CSTAT=1", NULL);
#endif /* USE_TI_COMMANDS */
/* assume radio is off on error */
if (isRadioOn() > 0) {
setRadioState (RADIO_STATE_SIM_NOT_READY);
}
}
```

默认状况下假设射频模块是好的，

通过 setRadioState (RADIO_STATE_SIM_NOT_READY) 来触发对无线模块的初始化。

通过 static void onRadioPowerOn() 对无线模块初始化。

首先通过 pollSIMState(NULL); 轮询 sim 卡状态 。

```
static void pollSIMState (void *param)
{
ATResponse *p_response;
int ret;
if (sState != RADIO_STATE_SIM_NOT_READY) {
// no longer valid to poll
return;
}
switch(getSIMStatus()) {
case RIL_SIM_ABSENT:
case RIL_SIM_PIN:
case RIL_SIM_PUK:
case RIL_SIM_NETWORK_PERSONALIZATION:
default:
setRadioState(RADIO_STATE_SIM_LOCKED_OR_ABSENT);
return;
case RIL_SIM_NOT_READY:
RIL_requestTimedCallback (pollSIMState, NULL, &TIMEVAL_SIMPOLL);
```

return;
case RIL_SIM_READY:
setRadioState(RADIO_STATE_SIM_READY);
return;
}
}
读取 sim 卡状态的函数是：getSIMStatus()
err = at_send_command_singleline("AT+CPIN?", "+CPIN:", &p_response);
它向猫发送了 at 命令 AT+CPIN? 来查询无线模块的状态，如果无线模块还没有就绪，那么他隔 1 秒钟继续调用
sim 卡状态轮询函数 pollSIMState，直到获得 sim 卡状态。
当 sim 卡状态为就绪，那么通过 setRadioState(RADIO_STATE_SIM_READY) 设置变量 sState 为：RADIO_STATE_SIM_READY，这时候会调用函数 static void onSIMReady()来进一步初始化无线模块。
发送的 at 命令有：
at_send_command_singleline("AT+CSMS=1", "+CSMS:", NULL);
at_send_command("AT+CNMI=1,2,2,1,1", NULL);
如果 sim 卡锁开启，或者 pin 被锁住的时候，会要求输入 pin 或者 puk，但是这个解锁动作必须在系统初始化完成以后才能
进行。(图形系统都还没有初始化怎么输入密码阿?)当系统初始化完成以后会调用 wm.systemReady()来通知大家。
这时候该做什么就做什么。
wm.systemReady()的调用会触发解锁界面。具体流程如下：
因为有： WindowManagerService wm = null;所以 wm.systemReady()
调用的是 WindowManagerService 中的函数：
public void systemReady() {
mPolicy.systemReady();
}
WindowManagerService 中有：
final WindowManagerPolicy mPolicy = PolicyManager.makeNewWindowManager();
PolicyManager.makeNewWindowManager 调用的是文件 PolicyManager.java 中的函数：
public static WindowManagerPolicy makeNewWindowManager() {
return sPolicy.makeNewWindowManager();
}
sPolicy.makeNewWindowManager 调用的是文件 Policy.java 中的函数：
public PhoneWindowManager makeNewWindowManager() {
return new PhoneWindowManager();
}
因为 PhoneWindowManager 继承自 WindowManagerPolicy
所以 mPolicy.systemReady() 最终调用的是文件 PhoneWindowManager.java 中的函数：
public void systemReady()
mKeyguardMediator.onSystemReady();
doKeyguard();
showLocked();
Message msg = mHandler.obtainMessage(SHOW);

mHandler.sendMessage(msg);
发送 SHOW 的消息。
文件 KeyguardViewMediator.java 中的消息处理函数：
public void handleMessage(Message msg) 对 SHOW 消息进行了处理。
如果 msg.what 等于 SHOW 那么执行：
handleShow();
private void handleShow()
...
mCallback.onKeyguardShow();
mKeyguardViewManager.show();
mShowing = true;
mKeyguardViewManager.show() 调用的是文件 KeyguardViewManager.java 中的函数：
public synchronized void show()
...
mKeyguardView = mKeyguardViewProperties.createKeyguardView(mContext, mUpdateMonitor, this);
...
mKeyguardViewProperties.createKeyguardView 调用的是文件 LockPatternKeyguardViewProperties.java 中的函数：
public KeyguardViewBase createKeyguardView(Context context,
KeyguardUpdateMonitor updateMonitor,
KeyguardWindowController controller) {
return new LockPatternKeyguardView(context, updateMonitor,
mLockPatternUtils, controller);
}
new LockPatternKeyguardView 调用了类 LockPatternKeyguardView 的构造函数：
public LockPatternKeyguardView(
Context context,
KeyguardUpdateMonitor updateMonitor,
LockPatternUtils lockPatternUtils,
KeyguardWindowController controller)
...
mLockScreen = createLockScreen();
addView(mLockScreen);
final UnlockMode unlockMode = getUnlockMode();
mUnlockScreen = createUnlockScreenFor(unlockMode);
mUnlockScreenMode = unlockMode;
addView(mUnlockScreen);
updateScreen(mMode);
执行上面的程序然后弹出解锁界面，getUnlockMode 获得锁类型，通常有三种：
enum UnlockMode {
Pattern, //图案锁
SimPin, //输入 pin 或者 puk
Account //账号锁
}

通过上面的过程我们可以知道，在系统初始化阶段启动 rild 的时候，rild 与猫进行了通信，并对猫进行初始化。

保存了网络的一系列状态。

=========

待机状态下，飞行模式切换流程分析：

飞行模式切换比较复杂，它状态改变时涉及到极大模块状态切换：

GSM 模块，蓝牙模块，wifi 模块。

飞行模式的 enabler 层会发送广播消息：ACTION_AIRPLANE_MODE_CHANGED

```
private void setAirplaneModeOn(boolean enabling) {
mCheckBoxPref.setEnabled(false);
mCheckBoxPref.setSummary(enabling ? R.string.airplane_mode_turning_on
: R.string.airplane_mode_turning_off);
// Change the system setting
Settings.System.putInt(mContext.getContentResolver(), Settings.System.AIRPLANE_MODE_ON,
enabling ? 1 : 0);
// Post the intent
Intent intent = new Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);
intent.putExtra("state", enabling);
mContext.sendBroadcast(intent);
}
```

因为 GSM ，蓝牙，wifi 模块分别注册了对 ACTION_AIRPLANE_MODE_CHANGED 消息的监测，所以收到

该消息后，模块会进行切换。

BluetoothDeviceService.java

开启蓝牙：enable(false);

关闭蓝牙：disable(false);

PhoneApp.java (packages\apps\phone\src\com\android\phone)

设置 GSM 模块状态 phone.setRadioPower(enabled);

WifiService.java

设置 wifi 状态 setWifiEnabledBlocking(wifiEnabled, false, Process.myUid());

===

GSM 模块切换过程分析：

phone.setRadioPower(enabled)调用的是：

文件 GSMPhone.java 中的的函数：

```
public void setRadioPower(boolean power)
mSST.setRadioPower(power);
```

因为有 ServiceStateTracker mSST;

mSST.setRadioPower 调用的是文件 ServiceStateTracker.java 中的函数：

```
public void setRadioPower(boolean power)
mDesiredPowerState = power;
setPowerStateToDesired();
cm.setRadioPower(true, null);
```

或者

```
cm.setRadioPower(false, null);
```

因为有：
CommandsInterface cm;
public final class RIL extends BaseCommands implements CommandsInterface
所以 cm.setRadioPower 调用的是文件 RIL.java 中的函数：
public void setRadioPower(boolean on, Message result)
RILRequest rr = RILRequest.obtain(RIL_REQUEST_RADIO_POWER, result);
rr.mp.writeInt(1);
...
send(rr)
通过 send 向 rild 发送 RIL_REQUEST_RADIO_POWER 请求来开启或者关闭 GSM 模块。
rild 数据接收流程：
收到 RIL_REQUEST_RADIO_POWER 执行：
requestRadioPower(data, datalen, t);
然后根据条件往无线模块发送模块开启和关闭请求
主要的 at 命令有：
err = at_send_command("AT+CFUN=0", &p_response);
err = at_send_command("AT+CFUN=1", &p_response);
===
蓝牙模块切换过程分析：
enable(false);
蓝牙开启调用文件 BluetoothDeviceService.java 中的函数：
public synchronized boolean enable(boolean saveSetting)
setBluetoothState(BluetoothDevice.BLUETOOTH_STATE_TURNING_ON);
mEnableThread = new EnableThread(saveSetting);
mEnableThread.start();
---
disable(false)
蓝牙关闭调用文件 中的函数：
public synchronized boolean disable(boolean saveSetting)
setBluetoothState(BluetoothDevice.BLUETOOTH_STATE_TURNING_OFF);
===
wifi 模块切换过程分析：
广播 wifi 状态改变的消息：WIFI_STATE_CHANGED_ACTION
setWifiEnabledState(enable ? WIFI_STATE_ENABLING : WIFI_STATE_DISABLING, uid);
更新 wifi 状态：
private void updateWifiState()
如果需要使能开启 wifi 那么会发送：
sendEnableMessage(true, false, mLastEnableUid);
sendStartMessage(strongestLockMode == WifiManager.WIFI_MODE_SCAN_ONLY);
mWifiHandler.sendEmptyMessage(MESSAGE_STOP_WIFI);
消息循环中处理命令消息：
public void handleMessage(Message msg)
如果使能 wifi：setWifiEnabledBlocking(true, msg.arg1 == 1, msg.arg2);
开启 wifi： mWifiStateTracker.setScanOnlyMode(msg.arg1 != 0);

setWifiEnabledBlocking(false, msg.arg1 == 1, msg.arg2);
断开 mWifiStateTracker.disconnectAndStop();
开启过程步骤：
1> 装载 wifi 驱动： WifiNative.loadDriver()
2> 启动后退 daemo supplicant： WifiNative.startSupplicant()
关闭过程步骤：
1> 停止后退 daemo supplicant：WifiNative.stopSupplicant()
2> 卸载 wifi 驱动: WifiNative.unloadDriver()
如果 wifi 状态默认为开启那么 WifiService 服务的构造函数：
WifiService(Context context, WifiStateTracker tracker)
boolean wifiEnabled = getPersistedWifiEnabled();
setWifiEnabledBlocking(wifiEnabled, false, Process.myUid());
会开启 wifi 模块。

3.3 Android Activity 的启动方式

在 android 里，有 4 种 activity 的启动模式，分别为：

“standard” (默认)

“singleTop”

“singleTask”

“singleInstance”

它们主要有如下不同：

3.3.1 如何决定所属 task

“standard”和”singleTop”的 activity 的目标 task，和收到的 Intent 的发送者在同一个 task 内，除非 intent 包括参数 FLAG_ACTIVITY_NEW_TASK。

如果提供了 FLAG_ACTIVITY_NEW_TASK 参数，会启动到别的 task 里。

“singleTask”和”singleInstance”总是把 activity 作为一个 task 的根元素，他们不会被启动到一个其他 task 里。

3.3.2 是否允许多个实例

“standard”和”singleTop”可以被实例化多次，并且存在于不同的 task 中，且一个 task 可以包括一个 activity 的多个实例；

“singleTask”和”singleInstance”则限制只生成一个实例，并且是 task 的根元素。

singleTop 要求如果创建 intent 的时候栈顶已经有要创建 的 Activity 的实例，则将 intent 发送给该实例，而不发送给新的实例。

3.3.3 是否允许其它 activity 存在于本 task 内

“singleInstance”独占一个 task，其它 activity 不能存在那个 task 里；如果它启动了一个新的 activity，不管新的 activity 的 launch mode 如何，新的 activity 都将会到别的 task 里运行（如同加了 FLAG_ACTIVITY_NEW_TASK 参数）。

而另外三种模式，则可以和其它 activity 共存。

3.3.4 是否每次都生成新实例

“standard”对于没一个启动 Intent 都会生成一个 activity 的新实例；

“singleTop”的 activity 如果在 task 的栈顶的话，则不生成新的该 activity 的实例，直接使用栈顶的实例，否则，生成该 activity 的实例。

比如现在 task 栈元素为 A-B-C-D（D 在栈顶），这时候给 D 发一个启动 intent，如果 D 是 “standard”的，则生成 D 的一个新实例，栈变为 A－B－C－D－D。

如果 D 是 singleTop 的话，则不会生产 D 的新实例，栈状态仍为 A-B-C-D

如果这时候给 B 发 Intent 的话，不管 B 的 launchmode 是”standard” 还是 “singleTop” ，都会生成 B 的新实例，栈状态变为 A-B-C-D-B。

“singleInstance”是其所在栈的唯一 activity，它会每次都被重用。

“singleTask”如果在栈顶，则接受 intent，否则，该 intent 会被丢弃，但是该 task 仍会回到前台。

当已经存在的 activity 实例处理新的 intent 时候，会调用 onNewIntent()方法

如果收到 intent 生成一个 activity 实例，那么用户可以通过 back 键回到上一个状态；如果是已经存在的一个 activity 来处理这个 intent 的话，用户不能通过按 back 键返回到这之前的状态。

3.4 Android arm linux kernel 启动流程

虽然这里的 Arm Linux kernel 前面加上了 Android，但实际上还是和普遍 Arm linux kernel 启动的过程一样的，这里只是结合一下 Android 的 Makefile，讲一下 bootimage 生成的一个过程。这篇文档主要描述 bootimage 的构造，以及 kernel 真正执行前的解压过程。

在了解这些之前我们首先需要了解几个名词，这些名词定义在/Documentation/arm/Porting 里面，这里首先提到其中的几个，其余几个会在后面 kernel 的执行过程中讲述：

①ZTEXTADDR　boot.img 运行时候 zImage 的起始地址，即 kernel 解压代码的地址。这里没有虚拟地址的概念，因为没有开启 MMU，所以这个地址是物理内存的地址。解压代码不一定需要载入 RAM 才能运行，在 FLASH 或者其他可寻址的媒体上都可以运行。

②ZBSSADDR　解压代码的 BSS 段的地址，这里也是物理地址。

③ZRELADDR　这个是 kernel 解压以后存放的内存物理地址，解压代码执行完成以后会跳到这个地址执行 kernel 的启动，这个地址和后面 kernel 运行时候的虚拟地址满足：__virt_to_phys(TEXTADDR) = ZRELADDR。

④INITRD_PHYS　Initial Ram Disk 存放在内存中的物理地址，这里就是我们的 ramdisk.img。

⑤INITRD_VIRT　Initial Ram Disk 运行时候虚拟地址。

⑥PARAMS_PHYS　内核启动的初始化参数在内存上的物理地址。

下面我们首先来看看 boot.img 的构造，了解其中的内容对我们了解 kernel 的启动过程是很有帮助的。首先来看看 Makefile 是如何产生我们的 boot.img 的：

out/host/linux-x86/bin/mkbootimg-msm7627_ffa --kernel out/target/product/msm7627_ffa/kernel --ramdisk out/target/product/msm7627_ffa/ramdisk.img --cmdline "mem=203M console=ttyMSM2,115200n8 androidboot.hardware=qcom" --output out/target/product/msm7627_ffa/boot.img

根据上面的命令我们可以首先看看 mkbootimg-msm7627ffa 这个工具的源文件：system/core/mkbootimg.c。看完之后我们就能很清晰地看到 boot.img 的内部构造，它是由 boot header /kernel　/ramdisk /second stage 构成的，其中前 3 项是必须的，最后一项是可选的。

java 代码：/*
** +-----------------+
** | boot header | 1 page
** +-----------------+
** | kernel | n pages
** +-----------------+
** | ramdisk | m pages
** +-----------------+
** | second stage | o pages
** +-----------------+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
**
** 0. all entities are page_size aligned in flash
** 1. kernel and ramdisk are required (size != 0)

```
** 2. second is optional (second_size == 0 -> no second)
** 3. load each element (kernel, ramdisk, second) at
** the specified physical address (kernel_addr, etc)
** 4. prepare tags at tag_addr. kernel_args[] is
** appended to the kernel commandline in the tags.
** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
** 6. if second_size != 0: jump to second_addr
** else: jump to kernel_addr
*/
```

关于 boot header 这个数据结构我们需要重点注意，在这里我们关注其中几个比较重要的值，这些值定义在 boot/boardconfig.h 里面，不同的芯片对应 vendor 下不同的 boardconfig，在这里我们的值分别是（分别是 kernel/ramdis/tags 载入 ram 的物理地址）：

java 代码：

```
#define PHYSICAL_DRAM_BASE 0x00200000
#define KERNEL_ADDR (PHYSICAL_DRAM_BASE + 0x00008000)
#define RAMDISK_ADDR (PHYSICAL_DRAM_BASE + 0x01000000)
#define TAGS_ADDR (PHYSICAL_DRAM_BASE + 0x00000100)
#define NEWTAGS_ADDR (PHYSICAL_DRAM_BASE + 0x00004000)
```

上面这些值分别和我们开篇时候提到的那几个名词相对应，比如 kernel_addr 就是 ZTEXTADDR，RAMDISK_ADDR 就是 INITRD_PHYS,而 TAGS_ADDR 就是 PARAMS_PHYS。bootloader 会从 boot.img 的分区中将 kernel 和 ramdisk 分别读入 RAM 上面定义的地址中，然后就会跳到 ZTEXTADDR 开始执行。

基本了解 boot.img 的内容之后我们来分别看看里面的 ramdisk.img 和 kernel 又是如何产生的，以及其包含的内容。从简单的说起，我们先看看 ramdisk.img，这里首先要强调一下这个 ramdisk.img 在 arm linux 中的作用。它在 kernel 启动过程中充当着第一阶段的文件系统，是一个 CPIO 格式打成的包。通俗上来讲他就是我们将生成的 root 目录，用 CPIO 方式进行了打包，然后在 kernel 启动过程中会被 mount 作为文件系统，当 kernel 启动完成以后会执行 init，然后将 system.img 再 mount 进来作为 Android 的文件系统。

在这里稍微解释下这个 mount 的概念，所谓 mount 实际上就是告诉 linux 虚拟文件系统它的根目录在哪，就是说我这个虚拟文件系统需要操作的那块区域在哪，比如说 ramdisk 实际上是我们在内存中的一块区域，把它作为文件系统的意思实际上就是告诉虚拟文件系统你的根目录就在我这里，我的起始地址赋给你，你以后就能对我进行操作了。实际上我们也可以使用 rom 上的一块区域作为根文件系统，但是 rom 相对 ram 慢，所以这里使用 ramdisk。然后我们在把 system.img mount 到 ramdisk 的 system 目录，实际上就是将 system.img 的地址给了虚拟文件系统，然后虚拟文件系统访问 system 目录的时候会重新定位到对 system.img 的访问。

【附录】

4.1 提交 BUG

亲爱的开发者，如果您发现文档中有不妥的地方，请发邮件至 eoeandroid@eoemobile.com 进行反馈，我们会定期更新、发布更新后的版本。感谢您对 eoe 的支持！我们的进步离不开大家的互勉！

4.2 关于 eoeAndroid

eoeAndroid 是国内成立最早，规模最大的 Android 开发者社区，拥有海量的 Android 学习资料。分享、互助的氛围，让 Android 开发者迅速成长，逐步从懵懂到了解，从入门到开发出属于自己的应用，eoeAndroid 为广大 Android 开发者奠定坚实的技术基础。从初级到高级，从环境搭建到底层架构，eoeAndroid 社区为开发者精挑细选了丰富的学习资料和实例代码。让 Android 开发者在社区中迅速的成长，并在此基础上开发出更多优秀的 Android 应用。

北京易联致远无线技术有限公司

责任编辑：果子狸

美术支持：金明根　　**技术支持：**gaotong

特别感谢：&麦#兜

中国最大的Android 开发者社区：www.eoeandroid.com

中国本土的 Android 软件下载平台：www.eoemarket.com