### 1 linux**内核启动之后便有了**init**进程**

2 init**进程根据**init.rc**文件进行各种配置并**fork**出**Zygote**进程，并且启动**ServiceManager**服务**

3 Zygote**建立起**app\_runtime **以及虚拟机之后，**fork**出**SystemServer**进程，并建立**socket**监听，等待**AMS**请求创建新的应用程序进程。**

4 SystemServer**启动各种服务包括**PMS**，**AMS**，**WMS**等等并将他们加入到**ServiceManager**管理**

##### 1.3.1**，**init**进程启动**

        源码：system/core/init/\*  
        配置文件：system/rootdir/init.rc  
        说明：init是一个由内核启动的用户级进程，它按照init.rc中的设置执行：启动服务（这里的服务指linux底层服务，如adbd提供adb支持，vold提供SD卡挂载等），执行命令和按其中的配置语句执行相应功能。

##### 1.3.2，zygote服务启动

       源码：frameworks/base/cmds/app\_main.cpp等。  
       说明：zygote是一个在init.rc中被指定启动的服务，该服务对应的命令是/system/bin/app\_process。  
       作用：建立[**Java**](http://lib.csdn.net/base/java)Runtime，建立虚拟机；建立Socket接收ActivityManangerService的请求，用于Fork应用程序；启动System Server。

##### 1.3.3，systemserver服务启动

       源码：frameworks/base/services/[**Java**](http://lib.csdn.net/base/javase)/com/android/server/SystemServer.java  
       说明：被zygote启动，通过System Manager管理android的服务（这里的服务指frameworks/base/services下的服务，如卫星定位服务，剪切板服务等）。

##### 1.3.4，launcher桌面启动

       源码：ActivityManagerService.java为入口，packages/apps/launcher\*实现。  
       说明：系统启动成功后SystemServer使用xxx.systemReady()通知各个服务，系统已经就绪，桌面程序Home就是在ActivityManagerService.systemReady()通知的过程中建立的，最终调用startHomeActivityLocked()启launcher。

##### 1.3.5，lockscreen启动

       源码：frameworks/policies/base/phone/com/android/internal/policy/impl/\*lock\*  
       说明：系统启动成功后SystemServer调用wm.systemReady()通知WindowManagerService，进而调用PhoneWindowManager，最终通过LockPatternKeyguardView显示解锁界面，跟踪代码可以看到解锁界面并不是一个Activity，这是只是向特定层上绘图，其代码了存放在特殊的位置。

##### 1.3.6，othersapp启动

       源码：frameworks/base/services/java/com/android/server/am/ActivityManagerService.java  
       说明：系统启动成功后SystemServer调用ActivityManagerNative.getDefault().systemReady()通知ActivityManager启动成功，ActivityManager会通过置变量mBooting，通知它的另一线程，该线程会发送广播android.intent.action.BOOT\_COMPLETED以告知已注册的第三方程序在开机时自动启动。

### **首先看一下Android系统的体系结构，相信大家都不陌生**



### **1.首先Bootloader引导程序启动完Linux内核后，会加载各种驱动和数据结构，当有了驱动以后，开始启动Android系统，同时会加载用户级别的第一个进程init(system\core\init.c),该进程会首先加载一个init.rc配置文件，代码如下**

int main(int argc, char \*\*argv)

{

// 创建文件夹 挂载

mount("tmpfs", "/dev", "tmpfs", 0, "mode=0755");

mkdir("/dev/pts", 0755);

// 打开日志

log\_init();

INFO("reading config file\n");

// 加载init.rc配置文件

init\_parse\_config\_file("/init.rc");

}

### **2.init.rc配置文件会进行很多的配置，创建很多的文件夹及文件，然后初始化一些Android驱动器，之后该配置文件最重要的一个任务就是启动一个Zygote(孵化器)进程，此进程是Android系统的一个母进程，用来启动Android的其他服务进程，代码：**

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

onrestart restart media

onrestart restart netd

### **3. Zygote会执行一个app\_process可执行文件，在这个文件中首先添加了Android运行时环境，在Android运行时中调用了ZygoteInit.java，这就从c++代码跳到了java代码。**

int main(int argc, const char\* const argv[])

{

...

// Android运行时环境

AppRuntime runtime;

...

// Next arg is startup classname or "--zygote"

if (i < argc) {

arg = argv[i++];

if (0 == strcmp("--zygote", arg)) {

bool startSystemServer = (i < argc) ?

strcmp(argv[i], "--start-system-server") == 0 : false;

setArgv0(argv0, "zygote");

set\_process\_name("zygote");

// 启动java代码

runtime.start("com.android.internal.os.ZygoteInit",

...

}

### **4.在ZytofeInit.java代码中首先设置了Java虚拟机的堆内存空间，然后启动一个类加载器加载Android启动依赖的类比如Activity等四大组件，dialog等UI的类，然后分出一个子进程启动SystemServer系统服务**

public static void main(String argv[]) {

try {

VMRuntime.getRuntime().setMinimumHeapSize(5 \* 1024 \* 1024);

// 加载Android依赖的类

preloadClasses();

//cacheRegisterMaps();

preloadResources();

...

if (argv[1].equals("true")) {

// 启动系统服务

startSystemServer();

} else if (!argv[1].equals("false")) {

...

}

private static boolean startSystemServer()

...

args = new String[] {

"--setuid=1000",

"--setgid=1000",

"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003,3006",

"--capabilities=130104352,130104352",

"--rlimit=8,",

"--runtime-init",

"--nice-name=system\_server",

"com.android.server.SystemServer",

...

/\* Request to fork the system server process \*/

// 母进程开始分叉服务 启动SystemServer

pid = Zygote.forkSystemServer(

parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids, debugFlags, rlimits,

parsedArgs.permittedCapabilities,

parsedArgs.effectiveCapabilities);

...

}

### **5.在SystemServer.java代码中有两个方法init1()启动Native世界，init2()启动Android的Framework世界**

public static void main(String[] args) {

...

// 加载jni库

System.loadLibrary("android\_servers");

// 调用native方法,该方法启动Native世界

init1(args);

}

native public static void init1(String[] args);

### **6.SystemServer首先调用init1()方法加载JNI库，启动Native世界。init1通过System.loadLibrary("android-servers")加载一个类库文件，其对应的源码文件为com\_android\_server\_SystemServer.cpp 其C++代码如下，在该类库中转调了system\_init()方法**

// 类似java的抽象方法

extern "C" int system\_init();

static void android\_server\_SystemServer\_init1(JNIEnv\* env, jobject clazz)

{

// 转调

system\_init();

}

/\*

\* JNI registration.

\*/

static JNINativeMethod gMethods[] = {

/\* name, signature, funcPtr \*/

// 函数指针 把init1方法映射到android\_server\_SystemServer\_init1

{ "init1", "([Ljava/lang/String;)V", (void\*) android\_server\_SystemServer\_init1 },

};

### **7.System\_init方法在System\_init.cpp中实现，它首先启动系统的硬件服务，比如Audio、Camera等，启动完硬件服务后它又通过Android运行时环境调用了SystemServer中的init2()方法，init2()方法启动Framework世界，代码如下：**

extern "C" status\_t system\_init()

{

...

// 启动硬件的服务

if (strcmp(propBuf, "1") == 0) {

// Start the SurfaceFlinger

SurfaceFlinger::instantiate();

}

AndroidRuntime\* runtime = AndroidRuntime::getRuntime();

LOGI("System server: starting Android services.\n");

// 启动完硬件服务后，又回到Systemserver的init2方法

runtime->callStatic("com/android/server/SystemServer", "init2");

...

}

### **8.SystemServer的init2方法，init2()方法启动Android的Framework层，启动Android世界。在该方法中启动了一个ServerThread线程，其代码如下：**

public static final void init2() {

Slog.i(TAG, "Entered the Android system server!");

Thread thr = new ServerThread();

thr.setName("android.server.ServerThread");

thr.start();

}

### **9.init2()启动的线程ServerThread中的的run方法中通过一个Looper开启了Android中的各种服务比如LightService，PowerManagerService，BatteryService，WindowManagerService等，并将服务添加到ServiceManager中去管理，启动完各种服务后，调用ActivityManagerService.systemReady方法：**

public void run() {

...

// 开启Android各种服务并且添加到ServiceManager去管理

Slog.i(TAG, "Device Policy");

devicePolicy = new DevicePolicyManagerService(context);

ServiceManager.addService(Context.DEVICE\_POLICY\_SERVICE, ottle =

...

// We now tell the activity manager it is okay to run third party

// code. It will call back into us once it has gotten to the state

// where third party code can really run (but before it has actually

// started launching the initial applications), for us to complete our

// initialization.

// 各种服务开启后调用ActivityManagerService.systemReady

((ActivityManagerService)ActivityManagerNative.getDefault())

.systemReady(new Runnable() {

public void run() {

Slog.i(TAG, "Making services ready");

### **10.在ActivityManagerService的systemReady方法中打开Android系统的第一个Activity**

public void systemReady(final Runnable goingCallback) {

...

// 打开第一个Activity

mMainStack.resumeTopActivityLocked(null);

}

}

### **11.ActivityStack的resumeTopActivityLocked方法启动home界面**

final boolean resumeTopActivityLocked(ActivityRecord prev) {

// Find the first activity that is not finishing.

// 没有已经打开的Activity, next为 null

ActivityRecord next = topRunningActivityLocked(null);

// Remember how we'll process this pause/resume situation, and ensure

// that the state is reset however we wind up proceeding.

final boolean userLeaving = mUserLeaving;

mUserLeaving = false;

if (next == null) {

// There are no more activities! Let's just start up the

// Launcher...

if (mMainStack) {

// 启动lucher应用的锁屏界面

return mService.startHomeActivityLocked();

}

}

### **12.打开了Luncher应用的Home界面之后，到此Android系统启动完成了。**

### **可以把整个过程总结为如下这张图**