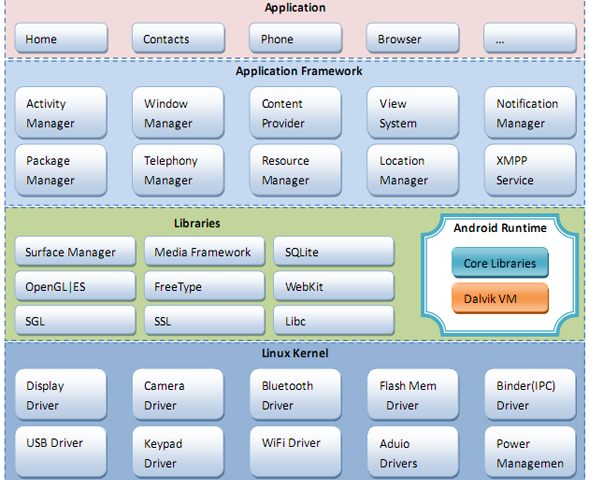
[Android 启动流程](http://blog.csdn.net/kevinx_xu/article/details/20444315)

2014-03-04 09:32 594人阅读 [评论](http://blog.csdn.net/kevinx_xu/article/details/20444315#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/kevinx_xu/article/details/20444315#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

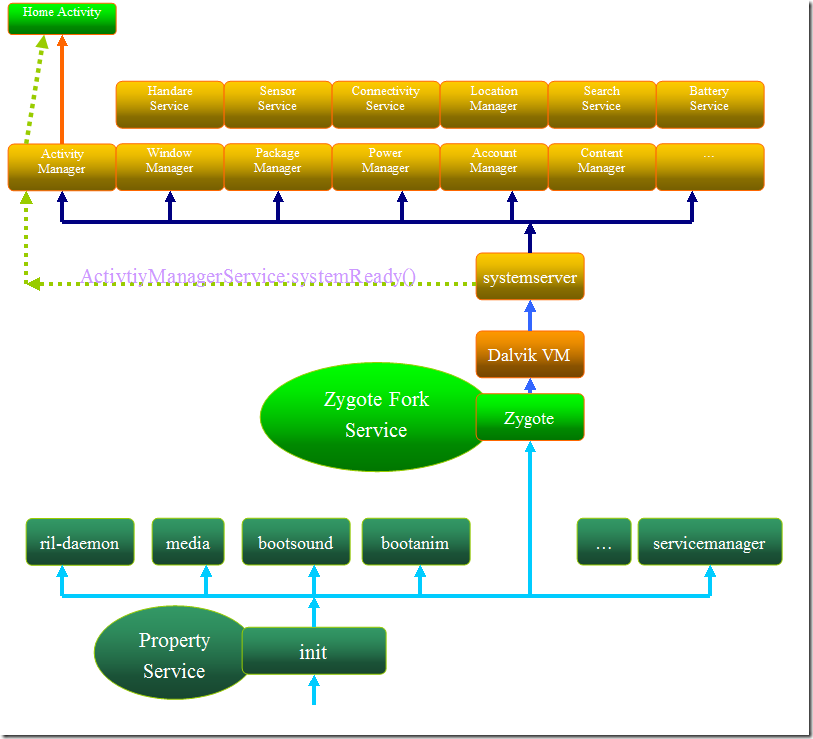
Android（29） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

首先**[Android](http://lib.csdn.net/base/15" \o "undefined" \t "_blank)**框架架构图：（来自网上，我觉得这张图看起来很清晰）



Linux内核启动之后就到Android Init进程，进而启动Android相关的服务和应用。

启动的过程如下图所示：（图片来自网上，后面有地址）



　　下面将从Android4.0源码中，和网络达人对此的总结中，对此过程加以学习了解和总结，

以下学习过程中代码片段中均有省略不完整，请参照源码。

**一 Init进程的启动**

　　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。

　　启动过程就是代码init.c中main函数执行过程：system\core\init\init.c

在函数中执行了：文件夹建立，挂载，rc文件解析，属性设置，启动服务，执行动作，socket监听……

下面看两个重要的过程：rc文件解析和服务启动。

**1 rc文件解析**

　　.rc文件是Android使用的初始化脚本文件 （System/Core/Init/readme.txt中有描述：

four broad classes of statements which are **Actions**, **Commands**, **Services**, and **Options**.）

　　其中Command 就是系统支持的一系列命令，如：export，hostname，mkdir，mount，等等，其中一部分是 linux 命令，

还有一些是 android 添加的，如：class\_start <serviceclass>： 启动服务，class\_stop <serviceclass>：关闭服务，等等。

　　其中Options是针对 Service 的选项的。

系统初始化要触发的动作和要启动的服务及其各自属性都在rc脚本文件中定义。 具体看一下启动脚本：\system\core\rootdir\init.rc

       在解析rc脚本文件时，将相应的类型放入各自的List中：

　　\system\core\init\Init\_parser.c  ：init\_parse\_config\_file( )存入到

　　action\_queue、   action\_list、 service\_list中，解析过程可以看一下parse\_config函数，类似状态机形式挺有意思。

　　这其中包含了服务：adbd、servicemanager、vold、ril-daemon、debuggerd、surfaceflinger、zygote、media……

**2 服务启动**

       文件解析完成之后将service放入到service\_list中。

文件解析完成之后将service放入到service\_list中。

 　　\system\core\init\builtins.c

       Service的启动是在do\_class\_start函数中完成：

int do\_class\_start(int nargs, char \*\*args)

{

service\_for\_each\_class(args[1], service\_start\_if\_not\_disabled);

return 0;

}

遍历所有名称为classname，状态不为SVC\_DISABLED的Service启动

复制代码

void service\_for\_each\_class(const char \*classname,

void (\*func)(struct service \*svc))

{

……

}

static void service\_start\_if\_not\_disabled(struct service \*svc)

{

if (!(svc->flags & SVC\_DISABLED)) {

service\_start(svc, NULL);

}

}

复制代码

do\_class\_start对应的命令：

**KEYWORD(class\_start, COMMAND, 1, do\_class\_start)**

init.rc文件中搜索**class\_start**：class\_start main 、class\_start core、……

　　main、core即为do\_class\_start参数classname

init.rc文件中Service class名称都是main：

       service drm /system/bin/drmserver

　　　　class **main**

　　service surfaceflinger /system/bin/surfaceflinger

    　　　class **main**

于是就能够通过main名称遍历到所有的Service，将其启动。

do\_class\_start调用：

       init.rc中

　　　　on boot　　//action

　　　　　　class\_start core　　　　//执行command 对应 do\_class\_start

    　　　　  class\_start main

Init进程main函数中：

system/core/init/init.c中：

复制代码

int main(){

　　　　　//挂在文件

       //解析配置文件：init.rc……

       //初始化化action queue

　　　　 ……

for(;;){

execute\_one\_command();

restart\_processes();

for (i = 0; i < fd\_count; i++) {

if (ufds[i].revents == POLLIN) {

if (ufds[i].fd == get\_property\_set\_fd())

handle\_property\_set\_fd();

else if (ufds[i].fd == get\_keychord\_fd())

handle\_keychord();

else if (ufds[i].fd == get\_signal\_fd())

handle\_signal();

}

}

}

}

复制代码

　　循环调用**service\_start**，将状态SVC\_RESTARTING启动， 将启动后的service状态设置为SVC\_RUNNING。

　　pid=fork();

　　execve();

　　在消息循环中：Init进程执行了Android的Command，启动了Android的NativeService，监听Service的变化需求，Signal处理。

Init进程是作为属性服务（Property service），维护这些NativeService。

**二 ServiceManager启动**

       在.rc脚本文件中zygote的描述：

复制代码

service servicemanager /system/bin/servicemanager

　　class core

　　user system

　　group system

　　critical

　　onrestart restart zygote

　　onrestart restart media

　　onrestart restart surfaceflinger

　　onrestart restart drm

复制代码

       ServiceManager用来管理系统中所有的binder service，不管是本地的c++实现的还是java语言实现的都需要

这个进程来统一管理，最主要的管理就是，注册添加服务，获取服务。所有的Service使用前都必须先在servicemanager中进行注册。

**do\_find\_service**( )

**do\_add\_service**( )

　　svcmgr\_handler( )

　　代码位置：frameworks\base\cmds\servicemanager\Service\_manager.c

**三 Zygote进程的启动**

　　Zygote这个进程是非常重要的一个进程，Zygote进程的建立是真正的Android运行空间，初始化建立的Service都是Navtive service.

**（1） 在.rc脚本文件中zygote的描述**：

复制代码

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

　　class main

　　socket zygote stream 666

　　onrestart write /sys/android\_power/request\_state wake

　　onrestart write /sys/power/state on

　　onrestart restart media

　　onrestart restart netd

参数：--zygote --start-system-server

复制代码

代码位置：frameworks/base/cmds/app\_process/app\_main.cpp

       上面的参数在这里就会用上，决定是否要启动和启动那些进程。

复制代码

int main( ){

AppRuntime runtime;

if (zygote) {

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

**startSystemServer ? "start-system-server" : "");**

}

}

class AppRuntime : public AndroidRuntime{}；

复制代码

**（2） 接着到了AndroidRuntime类中：**

frameworks\base\core\**jni**\AndroidRuntime.cpp

复制代码

void start(const char\* className, const char\* options){

// start the virtual machine Java在虚拟机中运行的

JNIEnv\* env;

if (startVm(&mJavaVM, &env) != 0) {

return;

}

//向刚刚新建的虚拟机注册JNI本地接口

if (startReg(env) < 0) {

return;

}

　　　　// jni 调用 java 方法，获取对应类的静态main方法

　　　　jmethodID startMeth = env->GetStaticMethodID(startClass,

　　"main","([Ljava/lang/String;)V");

// jni调用 java方法，调用到ZygoteInit类的main函数

jclass startClass = env->FindClass(className);

env->CallStaticVoidMethod(startClass, startMeth, strArray);

}

复制代码

**到了ZygoteInit.java中的静态main函数中，从C++ ——》JAVA**

**（3）ZygoteInit**

       真正Zygote进程:

              frameworks\base\core\java\com\android\internal\os\ZygoteInit.java

复制代码

public static void main(String argv[]) {

//Registers a server socket for zygote command connections

registerZygoteSocket();

//Loads and initializes commonly used classes and

//used resources that can be shared across processes

preload();

// Do an initial gc to clean up after startup

gc();

if (argv[1].equals("start-system-server")) {

startSystemServer();

}

/\*\*

\* Runs the zygote process's select loop. Accepts new connections as

\* they happen, and reads commands from connections one spawn-request's

\* worth at a time.

\*/

runSelectLoopMode(); //loop中

/\*\*

\* Close and clean up zygote sockets. Called on shutdown and on the

\* child's exit path.

\*/

closeServerSocket();

}

复制代码

**Zygote就建立好了，利用Socket通讯，接收请求，Fork应用程序进程，进入Zygote进程服务框架中。**

**四 SystemServer启动**

（1）在Zygote进程进入循环之前，调用了startSystemServer( );

复制代码

private static boolean startSystemServer(){

/\* Request to fork the system server process 孵化新的进程 \*/

　　　　ZygoteConnection.Arguments parsedArgs = null;

pid = Zygote.forkSystemServer(

parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids,

parsedArgs.debugFlags,

null,

parsedArgs.permittedCapabilities,

parsedArgs.effectiveCapabilities);

/\* For child process 对新的子进程设置 \*/

if (pid == 0) {

handleSystemServerProcess(parsedArgs);

}

}

void handleSystemServerProcess(parsedArgs){

closeServerSocket();

//"system\_server"

Process.setArgV0(parsedArgs.niceName);

//Pass the remaining arguments to SystemServer.

　　　　RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,

　　　　　　parsedArgs.remainingArgs);

/\* should never reach here \*/

}

复制代码

**（2）RuntimeInit中：**

       frameworks\base\core\java\com\android\internal\os\RuntimeInit.java

复制代码

//The main function called when started through the zygote process.

void zygoteInit(int targetSdkVersion, String[] argv){

applicationInit(targetSdkVersion, argv);

}

void applicationInit(int targetSdkVersion, String[] argv){

// Remaining arguments are passed to the start class's static main

invokeStaticMain(args.startClass, args.startArgs);

}

void invokeStaticMain(String className, String[] argv){

Class<?> cl;

cl = Class.forName(className);

//获取SystemServer的main方法，抛出MethodAndArgsCaller异常

Method m;

m = cl.getMethod("main", new Class[] { String[].class });

int modifiers = m.getModifiers();

throw new ZygoteInit.MethodAndArgsCaller(m, argv);

}

复制代码

**（3）**从**startSystemServer**开始执行并没有去调用SystemServer的任何方法，

　　　　只是通过反射获取了main方法，付给了MethodAndArgsCaller，并抛出了MethodAndArgsCaller异常。

　　　　此异常是在哪里处理的呢？

       回到startSystemServer( )函数的调用处：

       在ZygoteInit的main函数中：

复制代码

public static void main(String argv[]) {

try {

……

if (argv[1].equals("start-system-server")) {

startSystemServer(); //这里如果抛出异常，跳过下面流程

}

　　　　　　　　runSelectLoopMode(); //loop中

……

} catch (MethodAndArgsCaller caller) {

caller.run(); //处理的异常

}

}

复制代码

　　如果startSystemServer抛出了异常，跳过执行ZygoteInit进程的循环，这是怎么回事呢？

　　在startSystemServer中异常是由handleSystemServerProcess抛出，而

　　　　　　pid = Zygote.**forkSystemServer**( )

　　　　　　/\* For child process 仅对新的子进程设置 \*/

　　　　　　if (pid == 0) {

**handleSystemServerProcess**(parsedArgs);

　　　　　　}

　　　　　　// Zygote.**forkSystemServer**根据参数fork 出一个子进程，若成功调用，则返回两次：

　　　　一次返回的是 zygote 进程的 pid ，值大于0；一次返回的是子进程 pid，值等于0否则，出错返回-1；

**caller**.**run**();

　　　　MethodAndArgsCaller run函数：调用前面所提到的

　　　　//SystemServer main方法

　　　　m = cl.getMethod("main", new Class[] { String[].class });

　　　　启动了进程SystemServer。

**（4）SystemServer的执行 init1( )**

              //frameworks\base\services\java\com\android\server\SystemServer.java

复制代码

public static void main(String[] args) {

　　System.loadLibrary("android\_servers");

　　/\*

　　\* This method is called from Zygote to initialize the system.

　　\* This will cause the native services (SurfaceFlinger, AudioFlinger, etc..)

　　\* to be started. After that it will call back

　　\* up into init2() to start the Android services.

　　 \*/

　　 init1(args); //native 完了回调init2( )

　　}

//init1:

　　frameworks/base/services/jni/com\_android\_server\_SystemServer.cpp:: android\_server\_SystemServer\_init1( )

　　中调用：system\_init

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

{

sp<ProcessState> proc(ProcessState::self());

sp<IServiceManager> sm = defaultServiceManager();

//启动SurfaceFlinger 和传感器

property\_get("system\_init.startsurfaceflinger", propBuf, "1");

SurfaceFlinger::instantiate();

property\_get("system\_init.startsensorservice", propBuf, "1");

SensorService::instantiate();

// And now start the Android runtime. We have to do this bit

// of nastiness because the Android runtime initialization requires

// some of the core system services to already be started.

　　　　// All other servers should just start the Android runtime at

// the beginning of their processes's main(), before calling

// the init function.

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

//回调 com.android.server.SystemServer init2 方法

JNIEnv\* env = runtime->getJNIEnv();

jclass clazz = env->FindClass("com/android/server/SystemServer");

jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");

env->CallStaticVoidMethod(clazz, methodId);

//启动线程池 做为binder 服务

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

return NO\_ERROR;

}

复制代码

**ProcessState：**

　　每个进程在使用binder 机制通信时，均需要维护一个ProcessState 实例来描述当前进程在binder 通信时的binder 状态。

　　ProcessState 有如下2 个主要功能：

　　1. 创建一个thread, 该线程负责与内核中的binder 模块进行通信，称该线程为Pool thread ；

　　2. 为指定的handle 创建一个BpBinder 对象，并管理该进程中所有的BpBinder 对象。

**Pool thread：**

　　在Binder IPC 中，所有进程均会启动一个thread 来负责与BD 来直接通信，也就是不停的读写BD ，

　　这个线程的实现主体是一个IPCThreadState 对象，下面会介绍这个类型。

　　下面是Pool thread 的启动方式：

　　ProcessState::self()->**startThreadPool**();

**IPCThreadState ：**

　　IPCThreadState 也是以单例模式设计的。由于每个进程只维护了一个ProcessState 实例，同时ProcessState 只启动一个Pool thread ，

也就是说每一个进程只会启动一个Pool thread ，因此每个进程则只需要一个IPCThreadState 即可。

Pool thread 的实际内容则为：

IPCThreadState::self()->**joinThreadPool**();

**（5）SystemServer的执行 init2( )**

复制代码

public static final void init2() {

　　　　//建立线程来处理

Thread thr = new ServerThread();

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

thr.start();

}

**//看看线程ServerThread里面都做了什么事情？**

public void run() {

addBootEvent(new String("Android:SysServerInit\_START"));

Looper.prepare();

android.os.Process.setThreadPriority(

android.os.Process.THREAD\_PRIORITY\_FOREGROUND);

//初始化服务，创建各种服务实例，如：电源、网络、Wifi、蓝牙，USB等，

　　//初始化完成以后加入到 ServiceManager中，

//事我们用 Context.getSystemService (String name) 才获取到相应的服务

PowerManagerService power = null;

NetworkManagementService networkManagement = null;

WifiP2pService wifiP2p = null;

WindowManagerService wm = null;

BluetoothService bluetooth = null;

UsbService usb = null;

NotificationManagerService notification = null;

StatusBarManagerService statusBar = null;

……

power = new PowerManagerService();

ServiceManager.addService(Context.POWER\_SERVICE, power);

……

// ActivityManagerService作为ApplicationFramework最重要的服务

ActivityManagerService.setSystemProcess();

ActivityManagerService.installSystemProviders();

ActivityManagerService.self().setWindowManager(wm);

　　// 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.

　　//系统服务初始化准备就绪，通知各个模块

ActivityManagerService.self().systemReady(new Runnable() {

public void run() {

startSystemUi(contextF);

batteryF.systemReady();

networkManagementF.systemReady();

usbF.systemReady();

……

// It is now okay to let the various system services start their

// third party code...

appWidgetF.systemReady(safeMode);

wallpaperF.systemReady();

}

});

//

//BOOTPROF

addBootEvent(new String("Android:SysServerInit\_END"));

Looper.loop();

}

复制代码

 　　到这里系统ApplicationFramework层的XxxServiceManager准备就绪，可以开始跑上层应用了，我们的第一个上层应用HomeLauncher。

　　HomeActivity又是如何启动的呢？

　　Activity的启动必然和ActivityManagerService有关，我们需要去看看

　　ActivityManagerService.systemReady( )中都干了些什么。

**五 Home界面启动**

复制代码

public void systemReady(final Runnable goingCallback) {

　　　　……

　　　　//ready callback

if (goingCallback != null)

goingCallback.run();

synchronized (this) {

// Start up initial activity.

// ActivityStack mMainStack;

mMainStack.resumeTopActivityLocked(null);

}

……

}

final boolean resumeTopActivityLocked(ActivityRecord prev) {

　　// Find the first activity that is not finishing.

　　ActivityRecord next = topRunningActivityLocked(null);

　　if (next == null) {

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

　　　　// Launcher...

　　　　if (mMainStack) {

　　　　　　//ActivityManagerService mService;

　　　　　　return mService.startHomeActivityLocked();

　　　　}

　　}

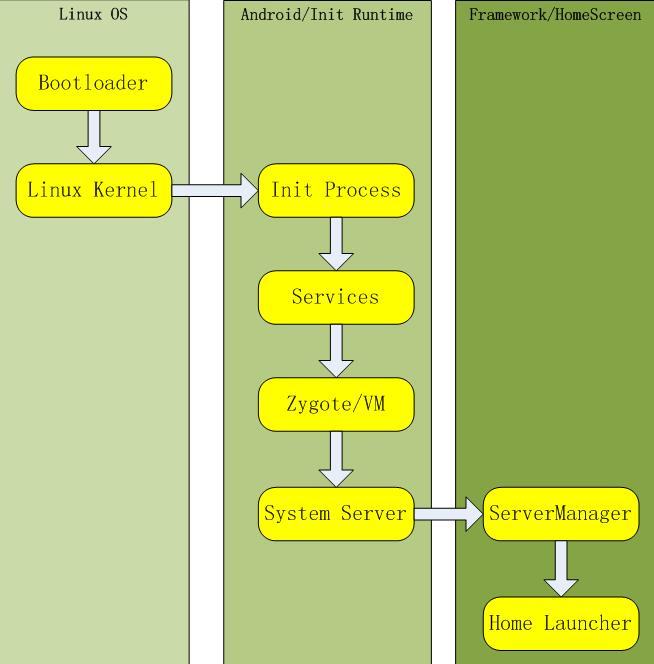
　　……

}

复制代码

       然后就启动了Home界面，完成了整个Android启动流程。

      整个过程如下：



参考文档：

<http://blog.csdn.net/maxleng/article/details/5508372>

<http://www.cnblogs.com/linucos/archive/2012/05/22/2513760.html#commentform>

<http://www.cnblogs.com/idiottiger/archive/2012/05/25/2516295.html>