**Android 启动过程详解**

Android从Linux系统启动有4个步骤；

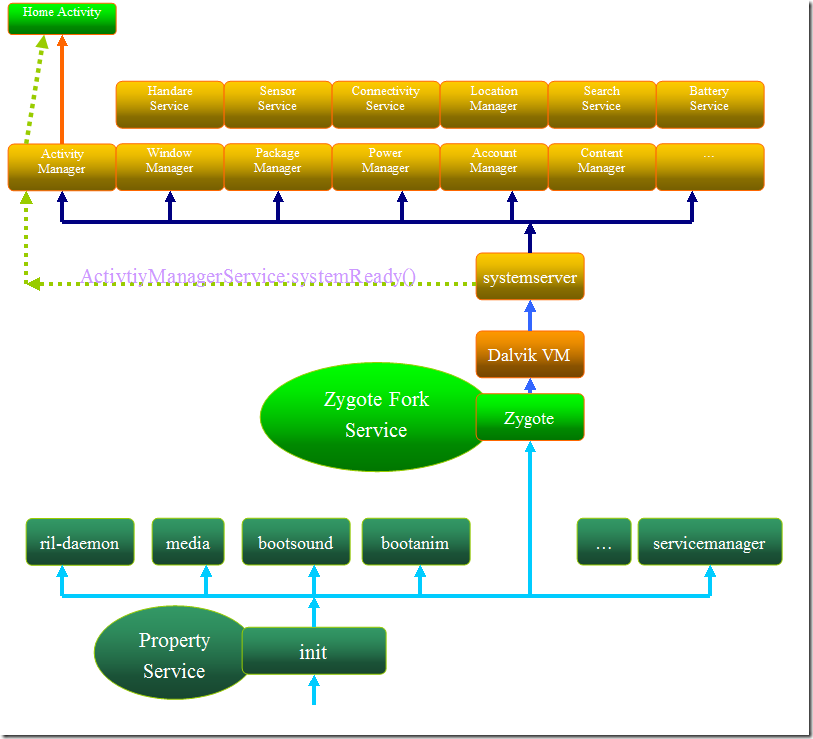
(1) init进程启动

(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：

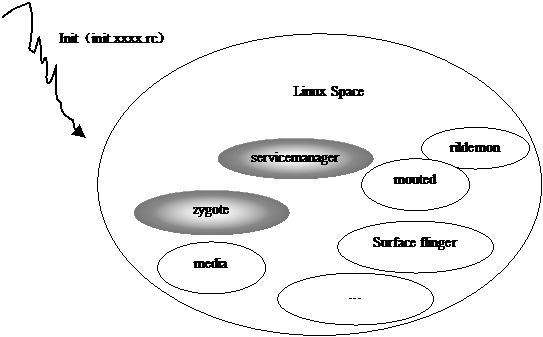


**第一步：initial进程(system/core/init)**

     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。

**1.1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.

**1.2 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。

**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。

(1) main(…)@frameworks/base/cmds/app\_main.cpp

* 建立Java Runtime
* runtime.start("com.android.internal.os.ZygoteInit",startSystemServer);

(2) runtime.start@AndroidRuntime.cpp

* 建立虚拟机
* 运行：com.android.internal.os.ZygoteInit：main函数。

（3）main()@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。

**第三步 System Server**

[startSystemServer@com.android.internal.os.ZygoteInit在Zygote](mailto:startSystemServer@com.android.internal.os.ZygoteInit%E5%9C%A8Zygote)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

**第三步 Home启动**

[在ServerThread@SystemServer.java](mailto:%E5%9C%A8ServerThread@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。

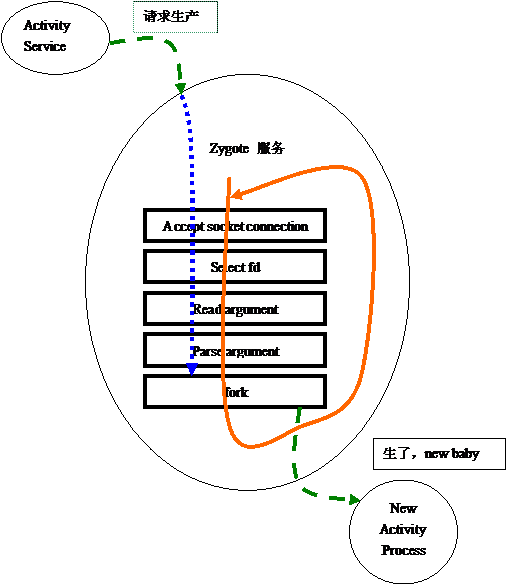
startActivityLocked（。。。）CATEGORY\_HOME

**Zygote Service**

在本章我们会接触到这两个单词：

* Zygote [生物] 受精卵, 接合子, 接合体
* Spawn：产卵

通过这两个单词，我们就可以大体知道Zygote是干什么的了，就是叫老母鸡下蛋。通过“Zygote”产出不同的子“Zygote”。从大的架构上讲，Zygote是一个简单的典型C/S结构。其他进程作为一个客服端向Zygote发出”孵化”请求，Zygote接收到命令就“孵化”出一个Activity进程来。



Zygote系统代码组成及其调用结构：

* Zygote.java

提供访问Dalvik “zygote”的接口。主要是包装Linux系统的Fork，以建立一个新的VM实例进程。

* ZygoteConnection.java

Zygote的套接口连接管理及其参数解析。其他Actvitiy建立进程请求是通过套接口发送命令参数给Zygote。

* ZygoteInit.java

Zygote的main函数入口。

Zygote系统代码层次调用

 main()

startSystemServer()…

runSelectLoopMode()

Accept socket connection

Conntecion.RunOnce()

Read argument

folkAndSpecialize

folkAndSpecialize使用Native函数Dalvik\_dalvik\_system\_Zygote\_forkAndSpecialize

//native 的获取

 dalvik/vm/native

//dalvik\_system\_Zygote.c

  const DalvikNativeMethod dvm\_dalvik\_system\_Zygote[] = {

    { "fork",            "()I",

        Dalvik\_dalvik\_system\_Zygote\_fork },

    { "forkAndSpecialize",            "(II[II[[I)I",

        Dalvik\_dalvik\_system\_Zygote\_forkAndSpecialize },

    { "forkSystemServer",            "(II[II[[I)I",

        Dalvik\_dalvik\_system\_Zygote\_forkSystemServer },

    { NULL, NULL, NULL },

};

在这里我们就有了Zygote服务的全貌理解，也在Code中印证了。【应[yk\_hu0621](http://hi.csdn.net/yk_hu0621)修正】{由于Android中没有具体应用程序的入口，都是通过启动Actvity来启动相关的Android应用，而这个 Android应用则对应着Linux进程，Activity便Host在这个应用程序上。}

{原文：Activity在本质上是个什么东西，就是一个Linux进程}

从分析中我们可以看到，Android使用了Linux的fork机制。在Linux中Fork是很高效的。

一个Android的应用实际上一个Linux进程，所谓进程具备下面几个要素，

a.要有一段程序供该进程运行，程序是可以被多个进程共享的。

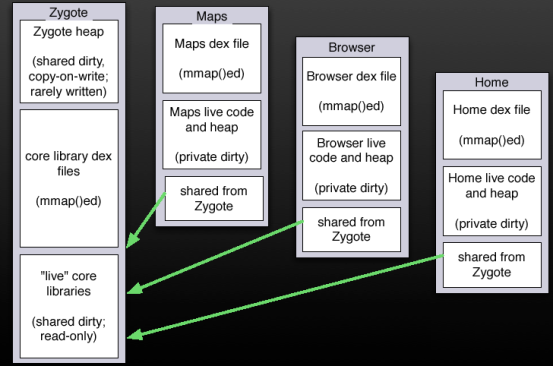
b.进程专用的系统堆栈空间。

c.进程控制块，在linux中具体实现是task\_struct

d.有独立的存储空间。

fork 创造的子进程复制了父亲进程的资源，包括内存的内容task\_struct内容，在复制过程中，子进程复制了父进程的task\_struct，系统堆栈空间和页面表，而当子进程改变了父进程的变量时候，会通过copy\_on\_write的手段为所涉及的页面建立一个新的副本。所以只有子进程有改变变量时，子进程才新建了一个页面复制原来页面的内容，基本资源的复制是必须的，整体看上去就像是父进程的独立存储空间也复制了一遍。

再看看下面Google在讲解Dalvik虚拟机的图片，我们就大体有了Android系统中Actvitiy的实际映射状态有了基本的认识。



./system/core/init/\*

>init.c main

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

snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);

init\_parse\_config\_file(tmp);

>init\_parser.c init\_parse\_config\_file

>init\_parser.c parse\_config

>init\_parser.c parse\_new\_section

>init\_parser.c parse\_service (K\_service: add to service\_list)

list\_add\_tail(&service\_list, &svc->slist);

>init\_parser.c parse\_line\_service (add refs to svc)

>init\_parser.c parse\_action (K\_on: add to action\_list)

list\_add\_tail(&action\_list, &act->alist);

>init\_parser.c parse\_line\_action (add cmds to act)

list\_add\_tail(&act->commands, &cmd->clist);