**从init进程fork出Zygote的过程详解**

**目录**

原文地址：https://wenku.baidu.com/u/%E4%B8%80%E5%89%91%E4%B8%A4%E5%BF%98?from=wenku

[1. 概述 3](#_Toc10704)

[2 系统启动基本原理 4](#_Toc8084)

[2.1 内核空间 4](#_Toc4166)

[2.2 应用空间 4](#_Toc5734)

[2.3 两个空间需要通讯 4](#_Toc20689)

[2.4 zygote是所有java进程的父进程 5](#_Toc31453)

[3  Init进程解析 5](#_Toc29126)

[3.1 /system/core/init/init.c分析 5](#_Toc27408)

[3.2 核心任务一: 各种初始化 7](#_Toc28044)

[3.3 核心任务二: 解析和执行init.rc文件 8](#_Toc16827)

[3.4 核心任务三: 执行command，监控service重启和socket请求 13](#_Toc32311)

[3.5 疑问：传说中的zygote进程启动在 哪里？ 14](#_Toc31192)

[4 认识和解析init.rc的内容 14](#_Toc7976)

[4.1认识AIL语言写成的init.rc 14](#_Toc9293)

[4.1.1 行(line)的概念 15](#_Toc15503)

[4.1.2 段(section)的概念 15](#_Toc10216)

[4.1.3 动作(action)和服务(service)的概念 16](#_Toc2038)

[4.1.3.1目前init.rc内已经写入的主要的action和service 16](#_Toc1582)

[4.1.3.2 Actions和command的语法格式 17](#_Toc25162)

[4.1.3.3服务(services)和选项(option)语法 21](#_Toc29657)

[4.2 解析init.rc过程 23](#_Toc3831)

[4.2.1 init\_parse\_config\_file和parse\_config 23](#_Toc18925)

[4.2.2 next\_token 25](#_Toc16655)

[4.2.3 lookup\_keyword、kw\_is和keywords.h 29](#_Toc15564)

[4.2.4 parse\_new\_section 34](#_Toc23198)

[5 zygote的启动 36](#_Toc28861)

[5.1 zygote在init.rc中的启动流程 36](#_Toc14224)

[5.1.1 zygote在init.rc中配置为class main 36](#_Toc32694)

[5.1.2 on boot这个action会执行class\_start main 36](#_Toc26838)

[5.1.3 class\_start main对应执行函数do\_class\_start 37](#_Toc25197)

[5.2 do\_class\_start真正启动zygote 38](#_Toc10674)

[5.2.1 do\_class\_start分析： 38](#_Toc32507)

[5.2.2 service\_for\_each\_class分析 38](#_Toc26695)

[5.2.3 service\_start\_if\_not\_disabled分析 39](#_Toc29082)

[5.2.4 service\_start分析 39](#_Toc17390)

[6 zygote进程分析 43](#_Toc2111)

[6.1 分析app\_process改名为zygote及其main函数 流程 43](#_Toc14037)

[6.2 AndroidRuntime启动dalvik虚拟机和ZygoteInit类 44](#_Toc28707)

[6.3 分析ZygoteInit 类 46](#_Toc3574)

[6.3.1 注册socket --registerZygoteSocket分析 47](#_Toc19576)

[6.3.2 加载资源-preload 48](#_Toc10602)

[6.3.3 启动systemserver 48](#_Toc26589)

[6.3.4 启动消息循环：runSelectLoop 51](#_Toc4252)

[6.3.5 zygote最终进化为所有app进程的父进程 52](#_Toc32328)

[7 总结: 54](#_Toc246)

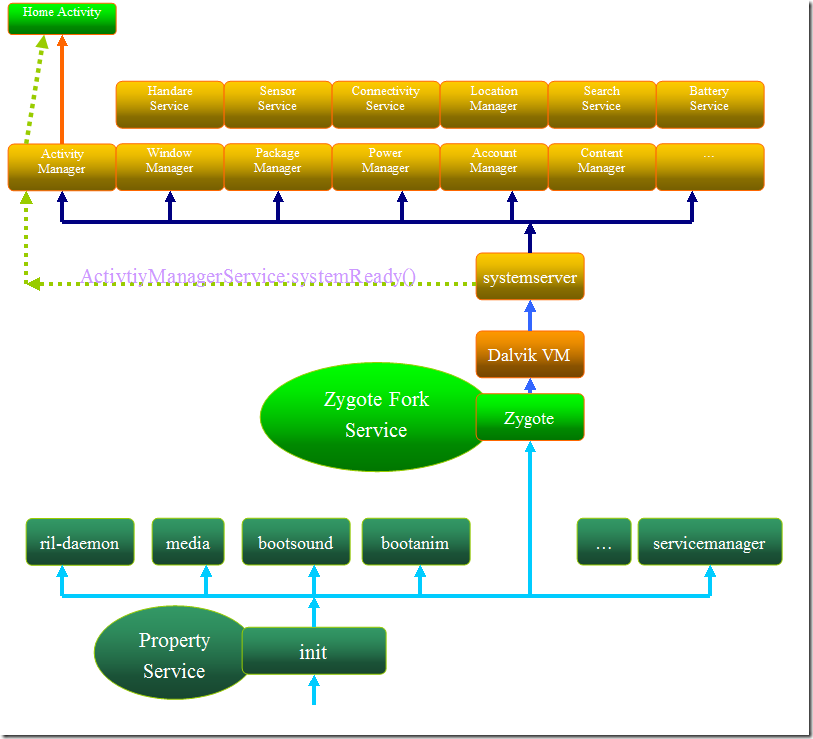
# 1. 概述

Linux内核启动之后,执行第一个进程 Init，并最终到达systemserver，systemserver再往上启动framework层相关服务和进程，启动launcher，完成整个开机流程。 init进程之前和systemserver启动之后的动作，本文不做分析，专著分析从init到systemserver之间的流程。

本文的核心：

* init进程工作内容分
* init.rc内容和执行分析
* zygote启动和内容分析
* systemserver启动和内容简析

android系统启动流程图:



# 2 系统启动基本原理

关键词：内核空间和用户空间

（关于内核空间和应用空间的详细描述（可以参考[《Android深度探索（卷1）：HAL与驱动开发》](http://product.dangdang.com/main/product.aspx?product_id=23043311&_ddclickunion=P-263982|ad_type=0|sys_id=1#dd_refer=http%3A%2F%2Fblog.csdn.net%2Fnokiaguy)）

## 2.1 内核空间

     Android本质上就是一个基于Linux内核的操作系统+庞杂的上层应用。一个完整的Linux系统首先会将一个Linux内核装载到内存，也就是编译Linux内核源代码生成的bzImage文件，对于为Android优化的Linux内核源代码会生成zImage文件。该文件就是Linux内核的二进制版本，zImage在内核空间运行。内核空间简而言之就是内核kernel的非用户user能控制的运行空间。

## 2.2 应用空间

而我们平常使用的软件都是在应用空间运行，这部分空间，用户可以自己分配进程所需内存和其他资源。

## 2.3 两个空间需要通讯

内核空间和应用空间是不能直接通过内存地址级别访问的，所以就需要建立某种通讯机制。目前Linux有很多通讯机制可以在用户空间和内核空间之间交互，例如设备驱动文件（位于/dev目录中）、内存文件（/proc、/sys目录等）。了解Linux的同学都应该知道Linux的重要特征之一就是一切都是以文件的形式存在的，例如，一个设备通常与一个或多个设备文件对应。这些与内核空间交互的文件都在用户空间。

注意：/dev,/proc,/sys这些设备文件路径，在系统编译完的img文件中是不存在的，所以原生的系统是没有他们的。

在Linux内核装载完，需要首先建立这些设备文件所在的目录，而完成这些工作的程序就是本文要介绍的内容之一：init进程。

init进程，它是一个由内核启动的用户级进程。内核自行启动之后，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，执行了：文件夹建立，挂载，rc文件解析，属性设置，启动服务，socket监听，启动zygote进程和servicemanager进程等。

## 2.4 zygote是所有java进程的父进程

　　zygote进程被init启动起来，再往上启动systemserver，并随时等待其他java进程请求，负责java进程的真正的创建和入口函数的执行。

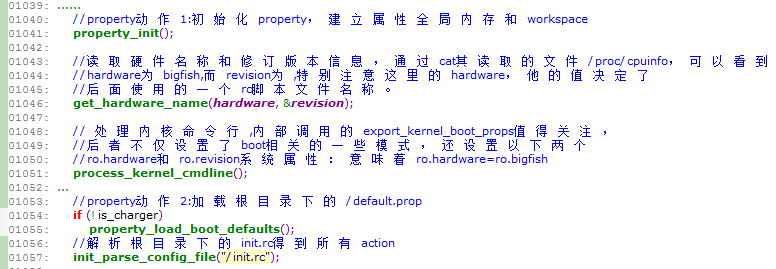
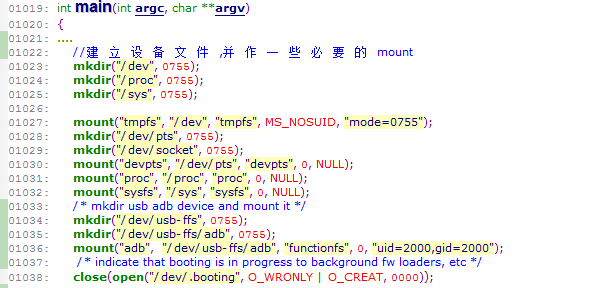
所以我们很有必要搞清楚，zygote到底是怎么启动的以及他自身的工作原理。

# 3  Init进程解析

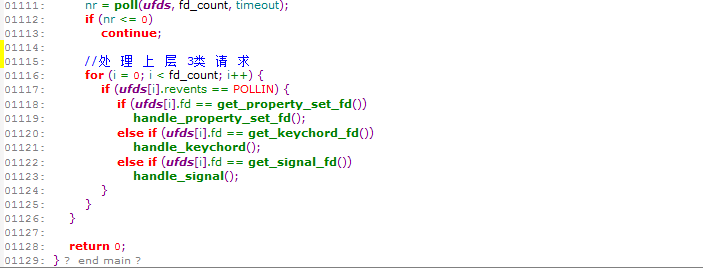
## 3.1/system/core/init/init.c分析



其中init.c是init的主文件，由于init是命令行程序，所以分析init.c首先应从main函数开始，现在好到main函数，代码如下：



注意：下面这段截图的代码本来是在for循环中，但是为了更直观地看到for循环所做的工作，特意把这段代码提出来.



我们可以看到main函数的逻辑并不复杂，但是信息量很大,通常只需要了解init的主线即可。main函数的主要任务包括以下3个部分：

* 初始化（包括建立/dev、/proc等目录、初始化属性）
* 解析和执行init.rc（以及init.rc加载的其他rc文件）文件中的各commond。
* 使用for无限循环执行rc文件的command，监控service进程的重启，响应系统属性等上层修改请求。

## 3.2 核心任务一: 各种初始化

设备文件目录的创建很直观，就不做分析了，这里主要分析下property进程初始化。

从上面的main函数分析中，大家可以看到依次调用了4个property相关的函数

* 初始化property全局内存和工作空间



* 加载根目录下的default.prop文件的全部属性

#define PROP\_PATH\_RAMDISK\_DEFAULT "/default.prop"



* 加载其余4个属性文件的全部属性，并启动socket监听

#define PROP\_PATH\_SYSTEM\_BUILD "/system/build.prop"

#define PROP\_PATH\_SYSTEM\_DEFAULT "/system/default.prop"

#define PROP\_PATH\_LOCAL\_OVERRIDE "/data/local.prop"

#define PERSISTENT\_PROPERTY\_DIR "/data/property"



* 将init.rc这里rc脚本文件解析出来的所有和property相关的action动作加入执行队列



这个任务先简单介绍到这里，这不是本文的主旨，要详细了解系统属性property的相关知识，可以参考另外一篇笔记《android系统属性详解》。

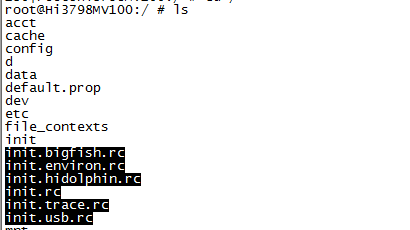
## 3.3 核心任务二: 解析和执行init.rc文件

      这里的配置文件主要指init.rc。我们所看到的init.rc实际上仅仅是kernel加载到内存中的镜像(真实的init.rc实际被打包到boot.image中去了)。要想修改rc的内容，只有修改源文件，并重新编译和烧写。

init.rc代码路径 :

\system\core\rootdir\init.rc

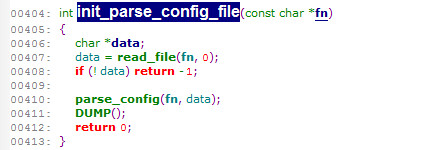
终端根目录下ls可以看到有以下相关rc文件：



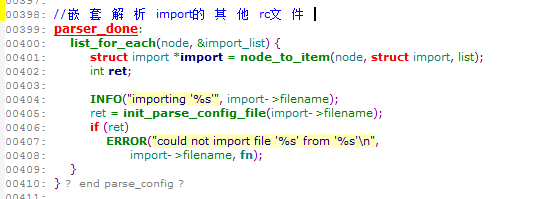
现在回到main函数，启动解析init.rc的代码是:



进入\system\core\init\ init\_parser.c 下的init\_parse\_config\_file函数：



进入\system\core\init\ init\_parser.c 下的parse\_config函数：



注意：

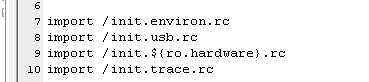
* 解析rc文件中所有的action,token,ketwork,section,service等:

一直跟踪parse\_new\_section，会发现最后数据存在servicelist，actionlist等链表中去了，后续执行这些解析出来的数据的时候，正是从servicelist，actionlist中取出来，加入到执行队列queue中.。至于rc文件到底是怎么解析的，其中有什么关键内容，先放到后面。

* 嵌套调用init\_parse\_config\_file:

继续解析完init.rc用import导入的所有其他rc.

看看init.rc的开头是如何导入其他文件的：



其中的import /init.${ro.hardware}.rc，显然需要知道ro. hardware到底是什么值。

前面分析init.c的main函数的时候，我们注意到下面2个函数：

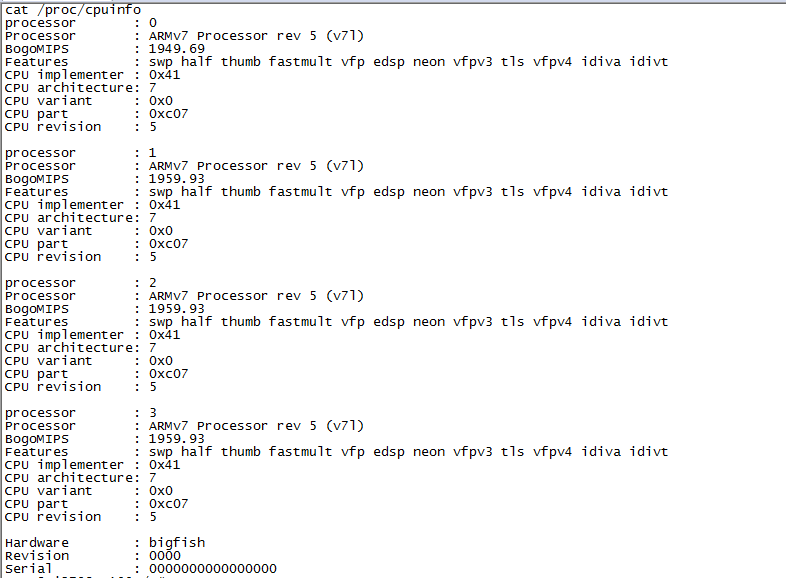


* 进入\system\core\init\util.c的get\_hardware\_name：

该函数的源代码如下:



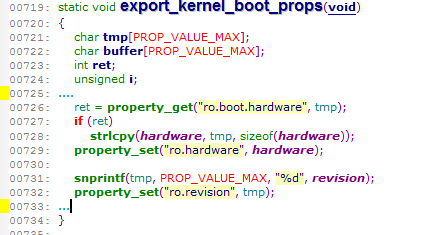
我们在终端cat /proc/cpuinfo获得如下信息



前面是4个CPU信息(4核处理器),最后明确看到Hardware的值是bigfish.

回到\system\core\init\init.c的 process\_kernel\_cmdline：

* process\_kernel\_cmdline中调用了export\_kernel\_boot\_props函数，这个函数设置了， ro. hardware这个属性的值：



ro.boot.hardware在系统中没有设值，所以hardware这个字符串保持为bigfish,导致：



所以import /init.${ro.hardware}.rc就是import /init. bigfish.rc，而这个文件在终端的根目录下，和init.rc同级.

* 接下来，解析得到的所有action就开始加入到执行队列:

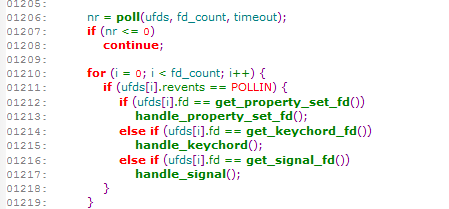


 并在\system\core\init\init.c的main的for循环中得以执行。



## 3.4 核心任务三: 执行command，监控service重启和socket请求

\system\core\init\init.c的main：



poll这个api其实就是检测传入的socket的可读，可写，异常等状态；返回POLLIN表示该socket可读，此时就分别调调三种类型任务对应的handle接口进行任务处理。

这三个socket分别是在各自的模块初始化函数（init）中创建的：



## 3.5 疑问：传说中的zygote进程启动在 哪里？

通过上面对\system\core\rootdir\init.c下面main函数及其执行的相关核心任务的解析，init进程到底干了些什么事，已经基本清楚了，但是我们似乎没发现传说中的zygote进程在哪里启动?

main函数的任务中，唯一确实看到他到底是内容，到底在做什么的就是init.rc的内容，整理我们还没有详细分析过，而zygote和servicemanager等的启动其实就藏在里面，我们这里只关注zygote。

# 4 认识和解析init.rc的内容

从上面可以看出，init.rc文件这个文件到底解析出来包含了哪些任务，是系统启动的根本所在，因为抛开这个部分，\system\core\init\init.c的main函数都没看到其他往上启动相关进程的地方。

源代码: \system\core\rootdir\ init.rc

## 4.1认识AIL语言写成的init.rc

init.rc 文件并不是普通的配置文件，而是由一种被称为“Android初始化语言”

（Android Init Language，这里简称为AIL）的脚本写成的文件。在了解init如何解析init.rc文件之前，先了解AIL非常必要。

AIL由如下4部分组成。

1.  动作（Actions）

2.  命令（Commands）

3．服务（Services）

4.  选项（Options）

5. 导入 (import)

其中action,service,import领衔的语句统称section；另外还有tokens和line的概念。

他们之间的关系：

section>action/services>line> tokens>： command

|Options

### 4.1.1 行(line)的概念

这4部分都是面向行的代码，也就是说：

1）用回车换行符作为每一条语句的分隔符。

2）每一行的代码由多个符号（Tokens）表示。

3）双引号可以将多个由空格分隔的Tokens合成一个Tokens。

4）如果一行写不下，可以在行尾加上反斜杠，来连接下一行。也就是 说，可以用反斜杠将多行代码连接成一行代码。

     5）AIL的注释与很多Shell脚本一样，以#开头。

### 4.1.2 段(section)的概念

     AIL在编写时需要分成多个部分（Section），而每一部分的开头需要指定Actions或Services或import。

而所有的Commands和Options只能属于最近定义的Section。如果Commands和 Options在第一个Section之前被定义，它们将被忽略。

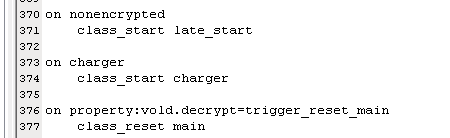
Actions和Services还需要有自己的名称，且必须唯一。如果有两个或多个Action或Service拥有同样的名称，那么init在执行它们时将抛出错误，并忽略这些Action和Service。

以init.rc为例，section包含下面三种类型的段内容：

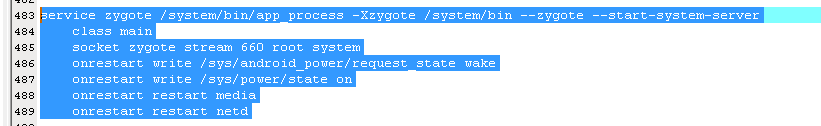
import:



action:



services:



### 4.1.3 动作(action)和服务(service)的概念

#### 4.1.3.1目前init.rc内已经写入的主要的action和service



#### 4.1.3.2 Actions和command的语法格式

on <trigger>

<command>

<command>

<command>

      也就是说Actions是以关键字on开头的，on后面跟的字符串叫触发器trigger，接下来是若干命令。例如，下面就是一个标准的Action。

on boot

ifup lo

hostname localhost

domainname localdomain

其中boot是触发器，下面三行是执行命令command.

* 目前init.rc支持的触发器类型:
* 1.  Boot
* 2. init/post-fs/post-fs-data
* 3.  <name>=<value>

   当属性<name>被设置成<value>时被触发。例如，

on property:vold.decrypt=trigger\_reset\_main class\_reset main

* 4.  device-added-<path>

    当设备节点被添加时触发

* 5.  device-removed-<path>

   当设备节点被移除时添加

* 6. service-exited-<name>

   会在一个特定的服务退出时触发

* 目前支持的command

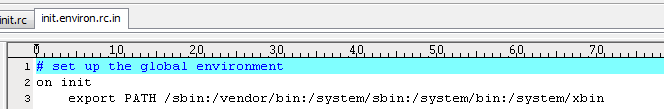
Actions后需要跟若干个命令，这些命令如下：

* 1.  exec <path> [<argument> ]\*

创建和执行一个程序（<path>）。在程序完全执行前，init将会阻塞。由于它不是内置命令，应尽量避免使用exec ，它可能会引起init执行超时。

* 2.  export <name><value>

在全局环境中将 <name>变量的值设为<value>。



* 3.  ifup <interface>

   启动网络接口

* 4.  import <filename>

   指定要解析的其他配置文件。常被用于当前配置文件的扩展

* 5.  hostname <name>

   设置主机名

* 6.  chdir <directory>

   改变工作目录

* 7.  chmod <octal-mode><path>

   改变文件的访问权限

* 8.  chown <owner><group><path>

   更改文件的所有者和组

* 9.  chroot <directory>

  改变处理根目录

* 10.  class\_start<serviceclass>

   启动所有指定服务类下的未运行服务。

* 11  class\_stop<serviceclass>

  停止指定服务类下的所有已运行的服务。

* 12.  domainname <name>

   设置域名

* 13.  insmod <path>

   加载<path>指定的驱动模块

* 14.  mkdir <path> [mode][owner] [group]

创建一个目录<path> ，可以选择性地指定mode、owner以及group。如果没有指定，默认的权限为755，并属于root用户和 root组。

* 15. mount <type><device><dir> [<mountoption> ]\*

   试图在目录<dir>挂载指定的设备。<device> 可以是mtd@name的形式指定一个mtd块设备。<mountoption>包括 "ro"、"rw"、"re

* 16.  setkey

   保留，暂时未用

* 17.  setprop <name><value>

   将系统属性<name>的值设为<value>。

* 18. setrlimit <resource><cur><max>

   设置<resource>的rlimit （资源限制）

* 19.  start <service>

   启动指定服务（如果此服务还未运行）。



* 20．stop<service>

   停止指定服务（如果此服务在运行中）。



* 21. symlink <target><path>

   创建一个指向<path>的软连接<target>。

* 22. sysclktz <mins\_west\_of\_gmt>

   设置系统时钟基准（0代表时钟滴答以格林威治平均时（GMT）为准）

* 23.  trigger <event>

  触发一个事件。用于Action排队

* 24.  wait <path> [<timeout> ]

等待一个文件是否存在，当文件存在时立即返回，或到<timeout>指定的超时时间后返回，如果不指定<timeout>，默认超时时间是5秒。

* 25. write <path><string> [ <string> ]\*

向<path>指定的文件写入一个或多个字符串。

#### 4.1.3.3服务(services)和选项(option)语法

Services （服务）是一个程序，他在初始化时启动，并在退出时重启（可选）。

Services的语法格式如下：

service <name><pathname> [ <argument> ]\*

<option>

<option>

例如，下面是一个标准的Service用法

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm



Services的选项是服务的修饰符，可以影响服务如何以及怎样运行。

* 目前支持的选项(option)如下：
* 1.  critical

表明这是一个非常重要的服务。如果该服务4分钟内退出大于4次，系统将会重启并进入 Recovery （恢复）模式。

* 2. disabled

 表明这个服务不会和其他和他处于同一trigger （触发器）下的服务自动启动。该服务必须被明确的按名启动。

* 3.  setenv <name><value>

在进程启动时将环境变量<name>设置为<value>。

* 4.  socket <name><type><perm> [ <user> [ <group> ] ]

   Create a unix domain socketnamed /dev/socket/<name> and pass

  its fd to the launchedprocess.  <type> must be"dgram", "stream" or "seqpacket".

   User and group default to0.

创建一个unix域的名为/dev/socket/<name> 的套接字，并传递它的文件描述符给已启动的进程。<type> 必须是 "dgram","stream" 或"seqpacket"。用户和组默认是0。

* 5.  user <username>

在启动这个服务前改变该服务的用户名。此时默认为 root。

* 6.  group <groupname> [<groupname> ]\*

在启动这个服务前改变该服务的组名。除了（必需的）第一个组名，附加的组名通常被用于设置进程的补充组（通过setgroups函数），档案默认是root。

* 7.  oneshot

   服务退出时不重启。

* 8.  class <name>

   指定一个服务类。所有同一类的服务可以同时启动和停止。如果不通过class选项指定一个类，则默认为"default"类服务。

* 9. onrestart

    当服务重启，执行一个命令（下详）

## 4.2 解析init.rc过程

     前面分析init.c的main函数的时候，粗略分析了下init.rc的解析过程，这里再回到init\_parse\_config\_file函数并从这里开始，分析详细的解析过程：

### 4.2.1 init\_parse\_config\_file和parse\_config

函数的实现在\system\core\init\ init\_parser.c下



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

这个方法主要负责初始化和分析init.rc文件。init\_parse\_config\_file函数在init\_parser.c文件中实现，代码如下：

int init\_parse\_config\_file(const char \*fn)

{

char \*data;

data = read\_file(fn, 0);

if (!data) return -1;

/\* 实际分析init.rc文件的代码 \*/

parse\_config(fn, data);

DUMP();

return 0;

}

      init\_parse\_config\_file方法开始调用了read\_file函数打开了/init.rc文件，并返回了文件的内容（char\*类 型），然后最核心的函数是parse\_config。该函数也在init\_parser.c文件中实现，代码如下：

static void parse\_config(const char \*fn, char \*s){

struct parse\_state state;

struct listnode import\_list;

struct listnode \*node;

char \*args[INIT\_PARSER\_MAXARGS];

int nargs;

nargs = 0;

state.filename = fn;

state.line = 0;

state.ptr = s;

state.nexttoken = 0;

state.parse\_line = parse\_line\_no\_op;

list\_init(&import\_list);

state.priv = &import\_list;

/\* 开始获取每一个token，然后分析这些token，每一个token就是有空格、字表符和回车符分隔的字符串,简单说就是找出全部我们能看到的一个个的字符串。\*/

for (;;) {

/\* next\_token函数相当于词法分析器 \*/

switch (next\_token(&state)) {

case T\_EOF: /\* init.rc文件分析完毕 \*/

state.parse\_line(&state, 0, 0);

goto parser\_done;

case T\_NEWLINE: /\* 分析每一行的命令 \*/

/\* 下面的代码相当于语法分析器 \*/

state.line++;

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;

}

break;

case T\_TEXT: /\* 处理每一个token \*/

if (nargs < INIT\_PARSER\_MAXARGS) {

args[nargs++] = state.text;

}

break;

}

}

parser\_done:

/\* 最后处理由import导入的初始化文件 \*/

list\_for\_each(node, &import\_list) {

struct import \*import = node\_to\_item(node, struct import, list);

int ret;

INFO("importing '%s'", import->filename);

/\* 递归调用 \*/

ret = init\_parse\_config\_file(import->filename);

if (ret)

ERROR("could not import file '%s' from '%s'\n",

import->filename, fn);

}

}

parse\_config方法的代码就比较复杂了，现在先说说该方法的基本处理流程。首先会调用  list\_init(&import\_list)初始化一个链表，该链表是用于存储通过import语句导入的初始化文件名。然后开始开始在 for循环中分析init.rc文件中的每一行代码。最后将init.rc文件分析完后，就会进入parser\_done部分，并递归调用 init\_parse\_config\_file方法分析通过import导入的初始化文件。

      通过分析parse\_config方法的原理，感觉也并不是很复杂。不过分析parse\_config方法的具体代码，还需要点编译原理的知识（只是概念 上的就可以）。在for循环中调用了一个next\_token方法不断从init.rc文件中获取token。这里的token，就是一种编程语言的最小 单元，也就是不可再分。例如，对于传统的编程语言，if、then等关键字、变量名等标识符都属于一个token。而对于init.rc文件来 说，import、on、以及触发器的参数值，都属于一个token。

### 4.2.2 next\_token

函数的实现在\system\core\init\ parser.c下。



     一个完整的编译器（或解析器）最开始需要进行词法和语法分析，词法分析就是在源代码文件中挑出一个个的Token，也就是说，词法分析器的返回值是 Token，而语法分析器的输入就是词法分析器的输出。也就是说，语法分析器需要分析一个个的token，而不是一个个的字符。由于init解析语言很简 单，所以就将词法和语法分析器放到了一起。词法分析器就是next\_token函数，而语法分析器就是T\_NEWLINE分支中的代码。这些就清楚多了。 现在先看看next\_token函数（在parser.c文件中实现）是如何获取每一个token的。

int next\_token(struct parse\_state \*state)

{

char \*x = state->ptr;

char \*s;

if (state->nexttoken) {

int t = state->nexttoken;

state->nexttoken = 0;

return t;

}

/\* 在这里开始一个字符一个字符地分析 \*/

for (;;) {

switch (\*x) {

case 0:

state->ptr = x;

return T\_EOF;

case '\n':

x++;

state->ptr = x;

return T\_NEWLINE;

case ' ':

case '\t':

case '\r':

x++;

continue;

case '#':

while (\*x && (\*x != '\n')) x++;

if (\*x == '\n') {

state->ptr = x+1;

return T\_NEWLINE;

} else {

state->ptr = x;

return T\_EOF;

}

default:

goto text;

}

}

textdone:

state->ptr = x;

\*s = 0;

return T\_TEXT;

text:

state->text = s = x;

textresume:

for (;;) {

switch (\*x) {

case 0:

goto textdone;

case ' ':

case '\t':

case '\r':

x++;

goto textdone;

case '\n':

state->nexttoken = T\_NEWLINE;

x++;

goto textdone;

case '"':

x++;

for (;;) {

switch (\*x) {

case 0:

/\* unterminated quoted thing \*/

state->ptr = x;

return T\_EOF;

case '"':

x++;

goto textresume;

default:

\*s++ = \*x++;

}

}

break;

case '\\':

x++;

switch (\*x) {

case 0:

goto textdone;

case 'n':

\*s++ = '\n';

break;

case 'r':

\*s++ = '\r';

break;

case 't':

\*s++ = '\t';

break;

case '\\':

\*s++ = '\\';

break;

case '\r':

/\* \ <cr><lf> -> line continuation \*/

if (x[1] != '\n') {

x++;

continue;

}

case '\n':

/\* \ <lf> -> line continuation \*/

state->line++;

x++;

/\* eat any extra whitespace \*/

while((\*x == ' ') || (\*x == '\t')) x++;

continue;

default:

/\* unknown escape -- just copy \*/

\*s++ = \*x++;

}

continue;

default:

\*s++ = \*x++;

}

}

return T\_EOF;

}

next\_token函数的代码还是很多的，不过原理到很简单。就是逐一读取init.rc文件（还有import导入的初始化文件）的字符，并将 由空格、“/t”和“/r”分隔的字符串挑出来，并通过state->text返回。如果返回了正常的token，next\_token函数就返回 T\_TEXT。如果一行结束，就返回T\_NEWLINE，如果init.rc文件的内容已读取完，就返回T\_EOF。当返回T\_NEWLINE时，开始语 法分析（由于init初始化语言是基于行的，所以语言分析实际上就是分析init.rc文件的每一行，只是这些行已经被分解成一个个token了）。感兴 趣的读者可以详细分析一下next\_token函数的代码，尽管代码很多，但并不复杂。而且还很有意思。

### 4.2.3 lookup\_keyword、kw\_is和keywords.h

函数的实现在\system\core\init\ init\_parser.c下

      现在回到parse\_config函数，先看一下T\_TEXT分支。该分支将获得的每一行的token都存储在args数组中。现在来看 T\_NEWLINE分支。该分支的代码涉及到一个state.parse\_line函数指针，该函数指针指向的函数负责具体的分析工作。但我们发现，一看 是该函数指针指向了一个空函数parse\_line\_no\_op，实际上，一开始该函数指针什么都不做，只是为了使该函数一开始不至于为null，否则调 用出错。

 现在来回顾一下T\_NEWLINE分支的完整代码。

case T\_NEWLINE:

state.line++;

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;

}

break;

在上面的代码中首先调用了lookup\_keyword方法搜索关键字。该方法的作用是判断当前行是否合法，也就是根据Init初始化语言预定义的关键字 查询，如果未查到，返回K\_UNKNOWN。lookup\_keyword方法在init\_parser.c文件中实现，代码如下：

int lookup\_keyword(const char \*s)

{

switch (\*s++) {

case 'c':

if (!strcmp(s, "opy")) return K\_copy;

if (!strcmp(s, "apability")) return K\_capability;

if (!strcmp(s, "hdir")) return K\_chdir;

if (!strcmp(s, "hroot")) return K\_chroot;

if (!strcmp(s, "lass")) return K\_class;

if (!strcmp(s, "lass\_start")) return K\_class\_start;

if (!strcmp(s, "lass\_stop")) return K\_class\_stop;

if (!strcmp(s, "lass\_reset")) return K\_class\_reset;

if (!strcmp(s, "onsole")) return K\_console;

if (!strcmp(s, "hown")) return K\_chown;

if (!strcmp(s, "hmod")) return K\_chmod;

if (!strcmp(s, "ritical")) return K\_critical;

break;

case 'd':

if (!strcmp(s, "isabled")) return K\_disabled;

if (!strcmp(s, "omainname")) return K\_domainname;

break;

… …

case 'o':

if (!strcmp(s, "n")) return K\_on;

if (!strcmp(s, "neshot")) return K\_oneshot;

if (!strcmp(s, "nrestart")) return K\_onrestart;

break;

case 'r':

if (!strcmp(s, "estart")) return K\_restart;

if (!strcmp(s, "estorecon")) return K\_restorecon;

if (!strcmp(s, "mdir")) return K\_rmdir;

if (!strcmp(s, "m")) return K\_rm;

break;

case 's':

if (!strcmp(s, "eclabel")) return K\_seclabel;

if (!strcmp(s, "ervice")) return K\_service;

if (!strcmp(s, "etcon")) return K\_setcon;

if (!strcmp(s, "etenforce")) return K\_setenforce;

if (!strcmp(s, "etenv")) return K\_setenv;

if (!strcmp(s, "etkey")) return K\_setkey;

if (!strcmp(s, "etprop")) return K\_setprop;

if (!strcmp(s, "etrlimit")) return K\_setrlimit;

if (!strcmp(s, "etsebool")) return K\_setsebool;

if (!strcmp(s, "ocket")) return K\_socket;

if (!strcmp(s, "tart")) return K\_start;

if (!strcmp(s, "top")) return K\_stop;

if (!strcmp(s, "ymlink")) return K\_symlink;

if (!strcmp(s, "ysclktz")) return K\_sysclktz;

break;

case 't':

if (!strcmp(s, "rigger")) return K\_trigger;

break;

case 'u':

if (!strcmp(s, "ser")) return K\_user;

break;

case 'w':

if (!strcmp(s, "rite")) return K\_write;

if (!strcmp(s, "ait")) return K\_wait;

break;

}

return K\_UNKNOWN;

}

lookup\_keyword方法按26个字母顺序（关键字首字母）进行处理。

现在回到parse\_config方法的T\_NEWLIEN分支，接下来调用了kw\_is宏具体判断当前行是否合法，该宏以及SECTION宏的定义如下。根据这些代码。明显是keyword\_info数组中的某个元素的flags成员变量的值取最后一位。

#define SECTION 0x01

#define kw\_is(kw, type) (keyword\_info[kw].flags & (type))

现在问题又转到keyword\_info数组了。该数组也在init\_parser.c文件中定义，代码如下:

#include "keywords.h"

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {

const char \*name;

int (\*func)(int nargs, char \*\*args);

unsigned char nargs;

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT] = {

[ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },

#include "keywords.h"

};

      从表面上看，keyword\_info数组是一个struct数组，但本质上，是一个map。为每一个数组元素设置了一个key，例如，数组元素{ "unknown", 0, 0,0 }的key是K\_UNKNOWN，而#include “keywords.h”大有玄机。上面的代码中引用了两次keywords.h文件，现在可以看一下keywords.h文件的代码。

#ifndef KEYWORD

int do\_chroot(int nargs, char \*\*args);

… …

int do\_export(int nargs, char \*\*args);

int do\_hostname(int nargs, char \*\*args);

int do\_rmdir(int nargs, char \*\*args);

int do\_loglevel(int nargs, char \*\*args);

int do\_load\_persist\_props(int nargs, char \*\*args);

int do\_wait(int nargs, char \*\*args);

#define \_\_MAKE\_KEYWORD\_ENUM\_\_

/\*

"K\_chdir", ENUM

\*/

#define KEYWORD(symbol, flags, nargs, func) K\_##symbol,

enum {

K\_UNKNOWN,

#endif

KEYWORD(capability, OPTION, 0, 0)

KEYWORD(chdir, COMMAND, 1, do\_chdir)

KEYWORD(chroot, COMMAND, 1, do\_chroot)

KEYWORD(class, OPTION, 0, 0)

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

KEYWORD(class\_stop, COMMAND, 1, do\_class\_stop)

KEYWORD(class\_reset, COMMAND, 1, do\_class\_reset)

KEYWORD(console, OPTION, 0, 0)

… …

KEYWORD(critical, OPTION, 0, 0)

KEYWORD(load\_persist\_props, COMMAND, 0, do\_load\_persist\_props)

KEYWORD(ioprio, OPTION, 0, 0)

#ifdef \_\_MAKE\_KEYWORD\_ENUM\_\_

KEYWORD\_COUNT,

};

#undef \_\_MAKE\_KEYWORD\_ENUM\_\_

#undef KEYWORD

#endif

      从keywords.h文件的代码可以看出，如果未定义KEYWORD宏，则在keywords.h文件中定义一个KEYWORD宏，以及一个枚举类型， 其中K\_##symbol的##表示连接的意思。而这个KEYWORD宏只用了第一个参数（symbol）。例 如，KEYWORD(chdir,       COMMAND, 1, do\_chdir)就会生成K\_chdir。

     而在keyword\_info结构体数组中再次导入keywords.h文件，这是KEYWORD宏已经在init\_parser.c文件中重新定义，所以第一次导入keywords.h文件使用的是如下的宏。

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

    这下就明白了，如果不使用keywords.h文件，直接将所有的代码都写到init\_parser.c文件中，就会有下面的代码。

int do\_chroot(int nargs, char \*\*args);

… …

enum

{

K\_UNKNOWN,

K\_ capability,

K\_ chdir,

… …

}

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {

const char \*name;

int (\*func)(int nargs, char \*\*args);

unsigned char nargs;

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT] = {

[ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },

[K\_ capability] = {" capability ", 0, 1, OPTION },

[K\_ chdir] = {"chdir", do\_chdir ,2, COMMAND},

… …

#include "keywords.h"

};

  可能我们还记着lookup\_keyword方法，该方法的返回值就是keyword\_info数组的key。

      在keywords.h前面定义的函数指针都是处理init.rc文件中service、action和command的。现在就剩下一个问题了，在哪里 为这些函数指针赋值呢，也就是说，具体处理每个部分的函数在哪里呢。现在回到前面的语法分析部分。如果当前行合法，则会执行 parse\_new\_section函数（在init\_parser.c文件中实现），该函数将为service和action设置处理这两部分的函数。

### 4.2.4 parse\_new\_section

parse\_new\_section函数的代码如下：

void parse\_new\_section(struct parse\_state \*state, int kw,

int nargs, char \*\*args)

{

printf("[ %s %s ]\n", args[0],

nargs > 1 ? args[1] : "");

switch(kw) {

case K\_service: // 处理service

state->context = parse\_service(state, nargs, args);

if (state->context) {

state->parse\_line = parse\_line\_service;

return;

}

break;

case K\_on: // 处理action

state->context = parse\_action(state, nargs, args);

if (state->context) {

state->parse\_line = parse\_line\_action;

return;

}

break;

case K\_import: // 单独处理import导入的初始化文件。

parse\_import(state, nargs, args);

break;

}

state->parse\_line = parse\_line\_no\_op;

}

现在看一下处理service的函数（parse\_line\_service）。

static void parse\_line\_service(struct parse\_state \*state, int nargs, char \*\*args)

{

struct service \*svc = state->context;

struct command \*cmd;

int i, kw, kw\_nargs;

if (nargs == 0) {

return;

}

svc->ioprio\_class = IoSchedClass\_NONE;

kw = lookup\_keyword(args[0]);

// 下面处理每一个option

switch (kw) {

case K\_capability:

break;

… …

case K\_group:

if (nargs < 2) {

parse\_error(state, "group option requires a group id\n");

} else if (nargs > NR\_SVC\_SUPP\_GIDS + 2) {

parse\_error(state, "group option accepts at most %d supp.

groups\n", NR\_SVC\_SUPP\_GIDS);

} else {

int n;

svc->gid = decode\_uid(args[1]);

for (n = 2; n < nargs; n++) {

svc->supp\_gids[n-2] = decode\_uid(args[n]);

}

svc->nr\_supp\_gids = n - 2;

}

break;

case K\_keycodes:

if (nargs < 2) {

parse\_error(state, "keycodes option requires atleast

one keycode\n");

} else {

svc->keycodes = malloc((nargs - 1) \* sizeof(svc->keycodes[0]));

if (!svc->keycodes) {

parse\_error(state, "could not allocate keycodes\n");

} else {

svc->nkeycodes = nargs - 1;

for (i = 1; i < nargs; i++) {

svc->keycodes[i - 1] = atoi(args[i]);

}

}

}

break;

… …

}

……

}

 Action的处理方式与service类似，读者可以自行查看相应的函数代码。现在一切都清楚了。处理service的函数是 parse\_line\_service，处理action的函数是parse\_line\_action。而前面的state.parse\_line根据当 前是service还是action，指向这两个处理函数中的一个，并执行相应的函数处理actioncommand和serviceoption。

    综合上述，实际上分析init.rc文件的过程就是通过一系列地处理，最终转换为通过parse\_line\_service或parse\_line\_action函数分析Init.rc文件中每一行的行为。

# 5 zygote的启动

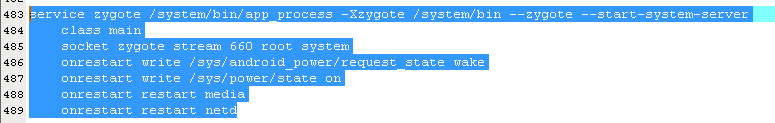
经过上面对/system/core/init/init.c下main函数逻辑流程的分析以及/system/core/rootdir/下面init.rc的解析，我们可以看到：

解析init.rc将所有要执行的action放到了action\_list，将所有要执行的service放到了service\_list。

main函数再将这些action加入到队列action\_queue，在for循环中通过execute\_one\_command进行调用，按先进先出顺序执行每一个section.

## 5.1 zygote在init.rc中的启动流程

### 5.1.1 zygote在init.rc中配置为class main

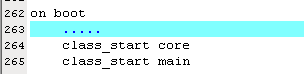


这是一个services的section，意味着他在在rc解析完成后，他被加入到sevicelist队列中，需要通过某个action的执行把他拉起来，并且他的classname是main，从前面的AIL语法知道，只要系统执行main类别的class，则所有class被标记为main的都会被执行。

### 5.1.2 on boot这个action会执行class\_start main

init.rc中有个action是on boot：

在该action下省略其他我们不关心的部分，可以看到：



解析完init.rc后，该action被main函数加入到执行队列

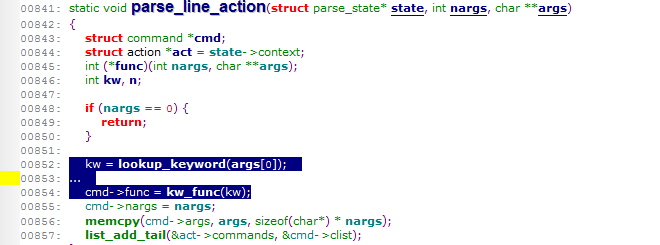


这两个command表示先后执行所有classname为core和main的service。

### 5.1.3 class\_start main对应执行函数do\_class\_start

解析action的流程是:

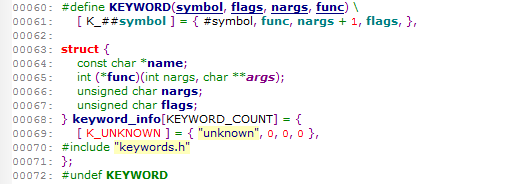
init\_parse\_config\_file-》parse\_config-》parse\_new\_section-》parse\_action-》parse\_line\_action:



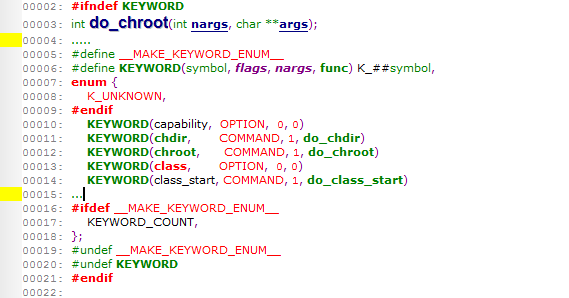
在**class\_start main**这一行上，显然args[0]就是**class\_start，**lookup\_keyword找到其对应的keyword是K\_class\_start，kw\_func继续查询到K\_class\_start对应的执行函数：



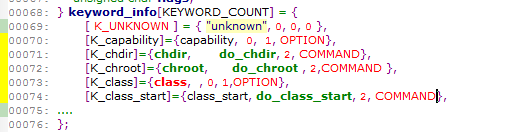
keyword\_info是一个结构数组:



看下这里包含的keywords.h头文件



这两个结合起来，显然keywords.h中的#ifndef KEYWORD和#ifdef \_\_MAKE\_KEYWORD\_ENUM\_\_这两段代码都会等于不存在，keyword\_info数组的初始化变成了：



所以会返回do\_class\_start这个函数。



所以init.c下面main函数的for循环对execute\_one\_command的执行，就会调用到

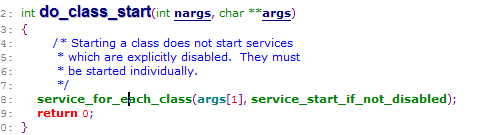
**do\_class\_start**。

现在我们只需要搞清楚do\_class\_start函数是如何执行zygote这个service的就可以了。

## 5.2 do\_class\_start真正启动zygote

### 5.2.1 do\_class\_start分析：

函数的代码实现在\system\core\init\builtins.c下:



注意init.rc中，

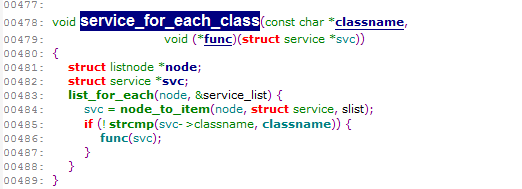


这一line被解析 的时候，args[0]是class\_start,则args[1]是main.

顾名思义，对所有class名称是main的service执行函数service\_start\_if\_not\_disabled。

### 5.2.2 service\_for\_each\_class分析

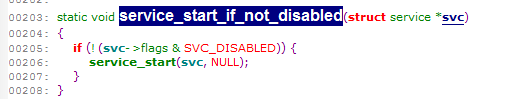
函数的代码实现在\system\core\init\init\_parser.c:



该函数就是查找service\_list下所有classname为main的service，并执行func

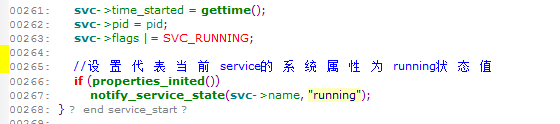
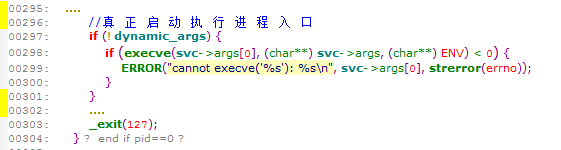
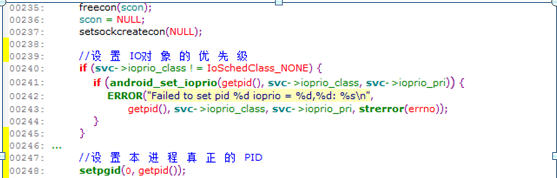
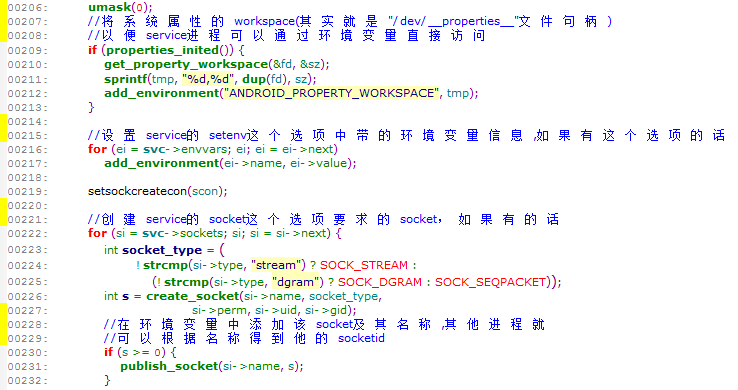
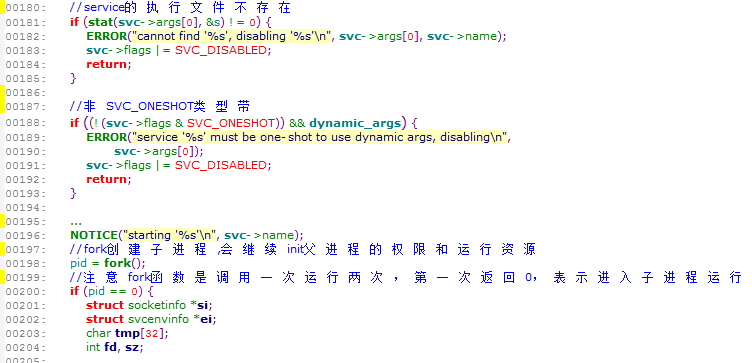
### 5.2.3 service\_start\_if\_not\_disabled分析

函数代码实现在\system\core\init\builtins.c：



### 5.2.4 service\_start分析

函数代码实现在\system\core\init\init.c分析service启动过程:



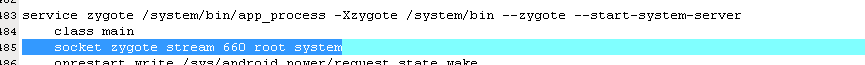
以上过程有以下几个知识点可以关注下::

1)fork函数的执行是调用一次，运行两次，正常情况下第一次返回PID为0，表示当前进入子进程运行；第二次返回真是的子进程PID。

关于fork的详细知识，请参考《android系统之linux内核fork函数说明.docx》

2 )service进程可以配置和创建socket，意味着这可能是其他进程和他通讯的途径

如我们这里要分析的对象zygote：



这里就配置了

选项：

socket zygote stream 660 root system

解读为：

name: zygote

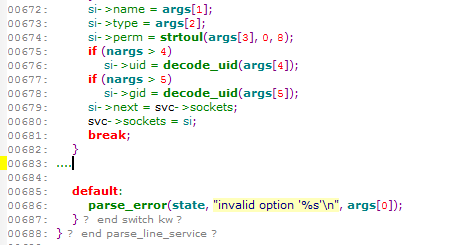
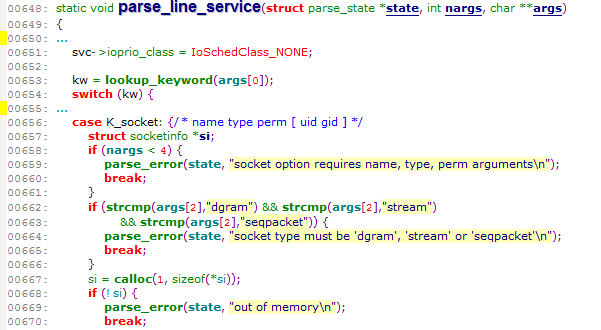
type: stream

permission: 660

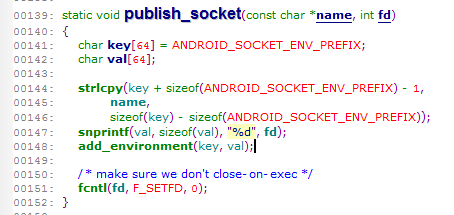
userid:root

groupid: system

,对照\system\core\init\init\_parser.c下面的parse\_line\_service就十分清晰了：



下面看下publish\_socket是如何注册socket到环境变量的：



通过这个流程，环境变量中就多了一个名为zygote的socket.

3) 特别注意execve：



这里的svc->args[0]对应的是



中的/system/bin/app\_process，这就和我们在终端crt上执行

. /system/bin/app\_process一回事，这个进程执行文件就真正开始启动运行了，到这里，zogote进程就被创建起来了,并进入其命令行入口函数main.

# 6 zygote进程分析

上面看启动的进程bin文件是/system/bin/app\_process.

对应的文件在frameworks/base/cmds/app\_process/app\_main.cpp

其他相关文件:

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

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

注意，我们一直都说的是zygote进程，怎么这里启动的进程名字叫app\_process，那么这二者之间一定有某种关联。我们先顺藤摸瓜，分析app\_process的执行。

## 6.1 分析app\_process改名为zygote及其main函数 流程

进入frameworks/base/cmds/app\_process/app\_main.cpp

app\_process目录编译后是一个进程的执行文件:app\_process

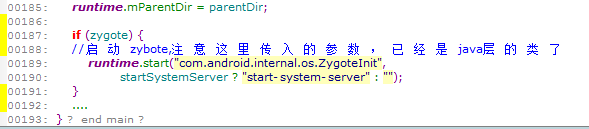
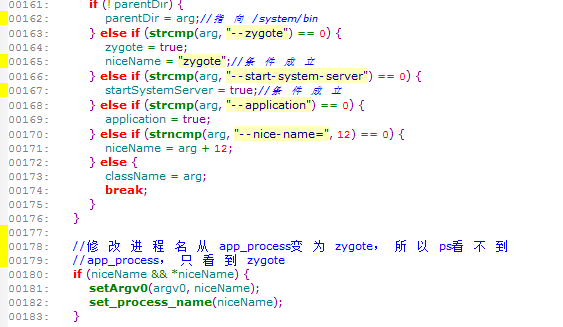
该进程被拉起来后,第一个被执行的是main入口:

从前面的分析可知,传入这个函数的参数列表是:



中的选中部分.

args[0]即/system/bin/app\_process,下面分析main函数:

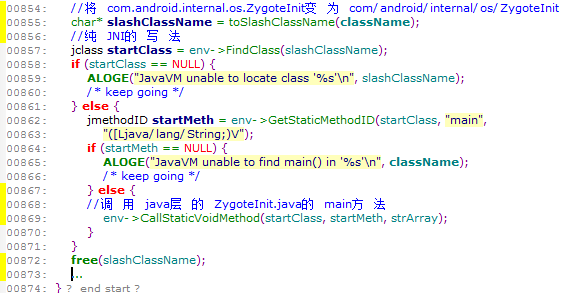


注意这里以AppRuntime.start函数拉起ZygoteInit类，而AppRuntime是AndroidRuntime的子类。



## 6.2 AndroidRuntime启动dalvik虚拟机和ZygoteInit类

既然上面的流程最后调用的是AppRuntime.start，自然我们要进入\frameworks\base\core\jni\AndroidRuntime.cpp，接着分析这个start函数到底做了什么.



这个函数做了2个主要工作：

1)启动了dalvik虚拟机，并注册了所有必须的系统层jni.,取得了全局JNIEnv.

2)以JNIEnv，通过JNI语法直接调用了com.android.internal.os.ZygoteInit 类的main函数，并且特别注意其一并把start-system-server这个参数传进去了。

所以

## 6.3 分析ZygoteInit 类

源代码路径:

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

分析下main函数到底干了啥：



可以看到有4个重要的事情，分别是:

1)注册socket.

2)加载资源.

3)启动systemserver.

4)启动消息循环，响应上层app进程请求.

那么这4个事情到底都是做什么事情，下面进一步分析

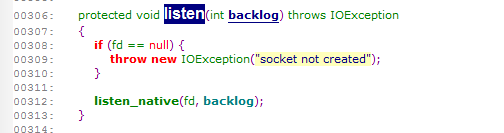
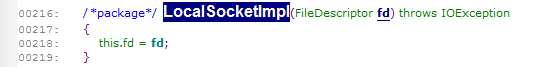
### 6.3.1 注册socket --registerZygoteSocket分析



第二段代码LocalServerSocket本质上就是启动调用jni层的listen而已，只不过拿socketfd对应的文件描述符来转了一次，并不重新创建socket，这里也清晰表明，这个以文件描述符注册的socket就是我们创建的zygote socket。

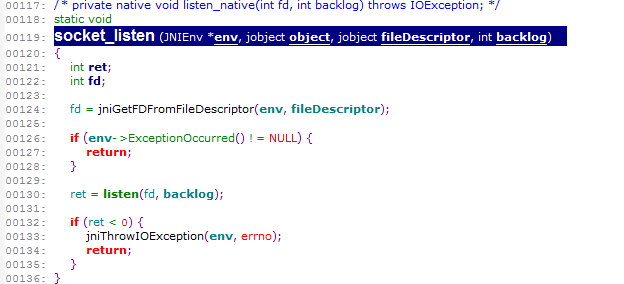
LocalServerSocket流程见下：

\frameworks\base\core\java\android\net\LocalServerSocket.java



最后的listen\_native是一个jni层函数,实现在

\frameworks\base\core\jni\android\_net\_LocalSocketImpl.cpp



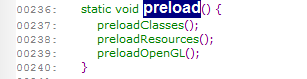
可以看到JNI层对应的接口就是把文件描述符还原成socket的id值，并调用listen.

所以这个LocalServerSocket.java文件仅仅是和C++层socket的一个连接站，registerZygoteSocket也仅仅是把zygotesocket在java层换成用socket的文件描述符再注册一次（登记到LocalServerSocket），并启动listen监听。

可见：

**这个socket变成为zygote进程的服务端，那么他的监听对象自然就是客户端，实际上的他的对象就是上层app创建进程的请求。**

### 6.3.2 加载资源-preload



这里是把加载耗时时间长的资源放在这个开机流程上进行预加载，使后面程序运行的时候不用再临时加载这些资源。

preloadClasses:

android的标准是单次load超过1.25ms的class都会进入preloadClasses的行列，编译时由\frameworks\base\tools\preload\工具检查class生成\frameworks\base\下的preloaded-classes文件。

而preloadClasses函数正是解析preloaded-classes来预加载class。

preloadResources：

预加载系统预制的各种资源文件，如xml,图片等。

preloadOpenGL：

看名字和opengl绘图处理有关，看不懂，没研究

### 6.3.3启动systemserver

这个函数的调用，分化出systemserver子进程，并由此启动进一步luancher，这个步骤正是我们追究的启动流程的最后一站:



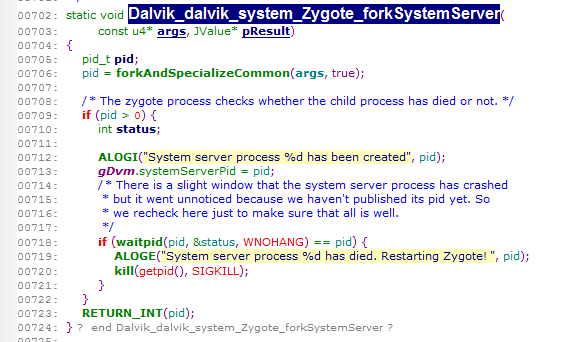
可以看出这个流程实际和zygote从init中分离出来一样的，先fork创建进程，后执行进程入口函数.

1）Zygote.forkSystemServer

再看Zygote.forkSystemServer，代码在/libcore/dalvik/src/main/java/dalvik/system/Zygote.java下，调用native函数nativeForkSystemServer

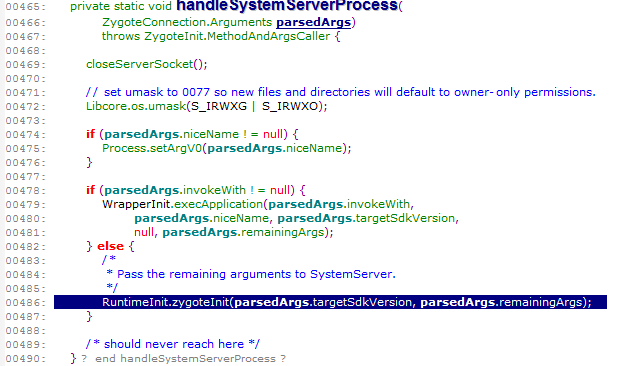


nativeForkSystemServer的jni实现在\dalvik\vm\native\dalvik\_system\_Zygote.cpp下的Dalvik\_dalvik\_system\_Zygote\_forkSystemServer：



可见fork相关接口最后都调用linux的fork这个api，本质都一样的。

2）handleSystemServerProcess

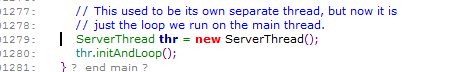


这个函数比较复杂，里面调用RuntimeInit.zygoteInit，绕了一大圈，总之最后的目的也就是调用systemserver类的main入口，大家有兴趣的可以去看看，实在很绕。

3）systemserver

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

进入到systemserver的main入口后，会执行



而initAndLoop调用ActivityManagerService的systemReady最终开始启动launcher



systemserver最终启动了launcher.

关于从systemserver开始到启动launcher结束的详细过程，可以参考另一篇笔记《android系统launcher启动流程.docx》

到这里，我们本文要最终的kernel之后的init进程到launcher启动就全部结束了。

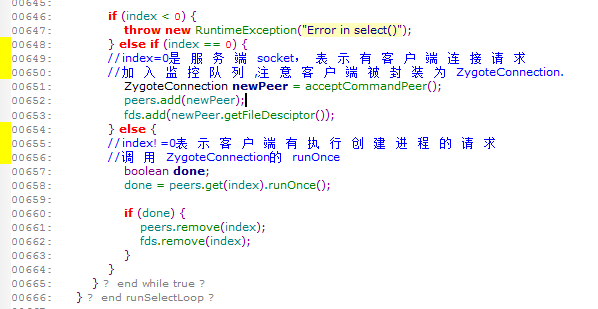
后面的内容是继续分析下zygote进程如何响应上层应用请求建立子进程.

### 6.3.4启动消息循环：runSelectLoop

进入zygote进程的的消息循环。等待上层app请求。

前面registerZygoteSocket将C++层创建的zygote这个socket处理为服务端，启动listen，那么处理accept的地方在哪里？就在这个runSelectLoop中。

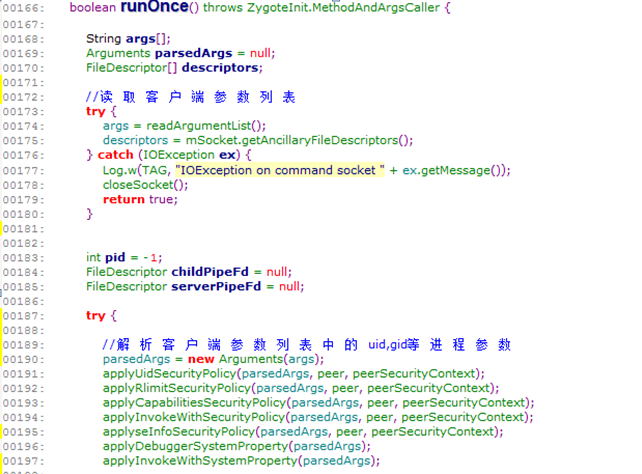
这个函数本身是个死循环，一直检测上层应用是否有链接该socket的请求；上层应用对该socket的链接请求，实际都是请求zygote进程分化出上层应用的子进程。



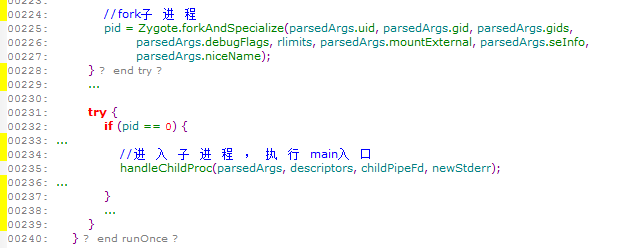
可见，该函数就一直在那循环检查上层连接请求，客户连接用ZygoteConnection封装；有连接后，监控该客户端socket的执行请求，有请求后执行ZygoteConnection.runonce建立客户端子进程。

### 6.3.5 zygote最终进化为所有app进程的父进程

继续进入\frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java



正是由于这个runonce函数的执行，zygote完成响应上层app客户端的子进程创建。



其中Zygote .forkAndSpecialize完成子进程创建，handleChildProc启动执行子进程入口.

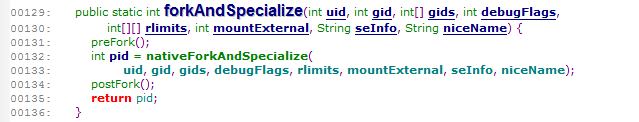
注意：

这里和前面的zygote启动systemserver一样，进入了zygote.java.

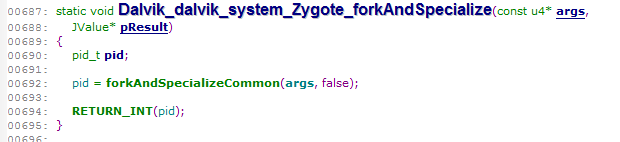
下面也简单分析子进程的创建和执行，可以看到和systemserver非常类似:

Zygote.forkAndSpecialize的代码在

/libcore/dalvik/src/main/java/dalvik/system/Zygote.java下，forkAndSpecialize 调用native函数nativeForkAndSpecialize



nativeForkAndSpecialize 的jni实现在\dalvik\vm\native\dalvik\_system\_Zygote.cpp里面，这里就和systemserver是调用同一个fork接口了.



到此完成对app子进程的创建。

最后再看下handleChildProc如何调用入口函数



请注意，这里写得很清楚，执行的入口是app层传下来的是参数列表的第一个参数：className = parsedArgs.remainingArgs[0];

意味着上层请求创建子进程的时候，第一个参数是一个带包路径的完整的类名。

# 7 总结:

1)学习掌握了从init到 zygote到systemsever的启动流程，如果后续考虑做一些进程启动优化（当然假设有这个需要和能力的话），可以不用再从头梳理流程，可以着重分析需要优化的节点.

2)学习掌握了AIL语言脚本编写方法

3)附带看到了系统属性的加载过程，再梳理下该分支的知识，会有意想不到的收获.

4)认识到系从上电到显示launcher分3个阶段:

sboot到kernel的linux阶段启动阶段

从init到systemserver的真正属于android系统层的启动阶段

从systemserver到launcher的android app层启动阶段.

把前后2个阶段再梳理学习一下，则整个系统的启动都变得清晰了.

5)对系统启动流程从一无所知变得逐渐清晰，也增强可代码阅读能力。

6)学习到进程相关的知识，把fork相关的东西再学习梳理一下，会很收益，这是以点带面的学习方法。