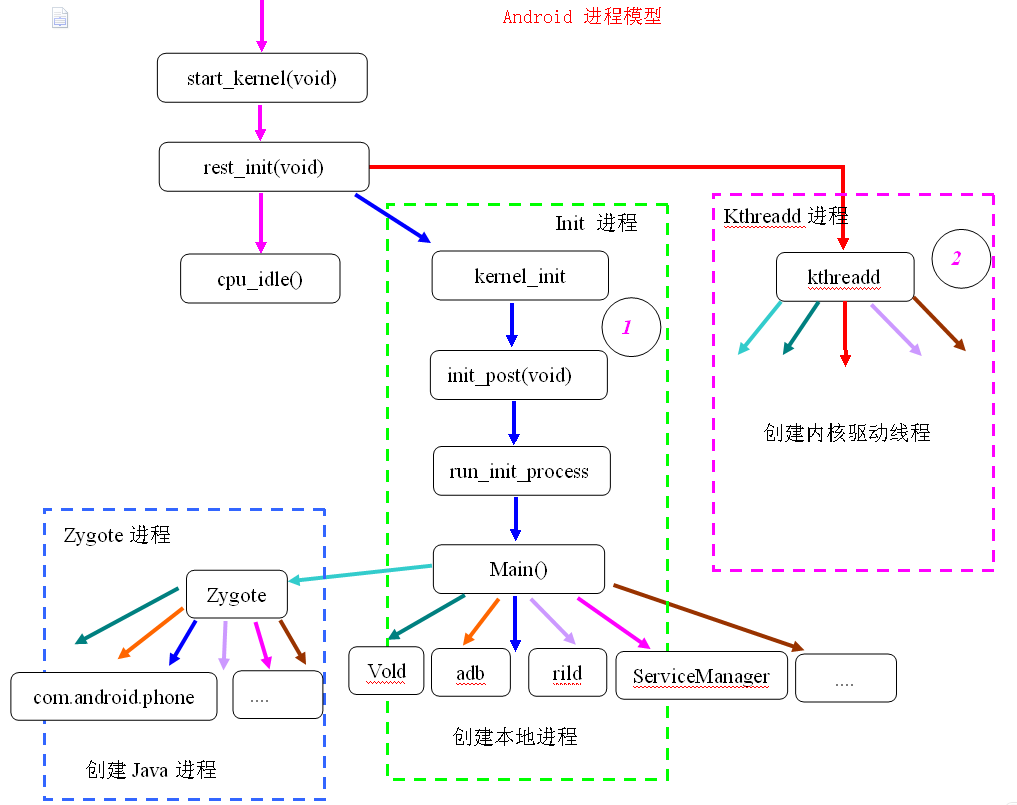
**Init 进程源码分析**

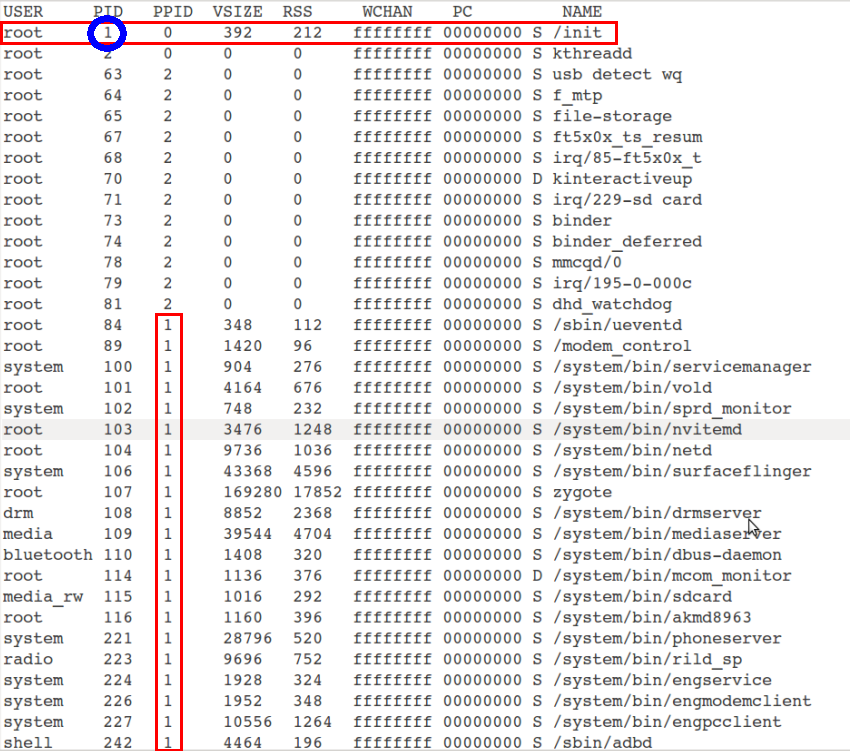
基于Linux内核的android系统，在内核启动完成后将创建一个Init用户进程，实现了内核空间到用户空间的转变。在[Android 启动过程介绍](http://blog.csdn.net/yangwen123/article/details/8023654" \t "_blank)一文中介绍了Android系统的各个启动阶段，init进程启动后会读取init.rc配置文件，通过fork系统调用启动 init.rc文件中配置的各个Service进程。init进程首先启动启动android的服务大管家ServiceManager服务，然后启动 Zygote进程。Zygote进程的启动开创了Java世界，无论是SystemServer进程还是android的应用进程都是Zygote的子进 程，[Zygote进程启动过程的源代码分析](http://blog.csdn.net/yangwen123/article/details/8059995" \t "_blank)一文中详细介绍了Zygote进程的启动过程，[System Server进程启动过程源码分析](http://blog.csdn.net/yangwen123/article/details/8060156" \t "_blank)则详细介绍了在Zygote进程启动完成后创建的第一个进程SystemServer进程的启动过 程，SystemServer进程的启动包括两个阶段，在第一阶段主要是启动C++相关的本地服务，如SurfaceFlinger等，在第二阶段通过在 ServerThread线程中启动android的各大关键Java服务。[Zygote孵化应用进程过程的源码分析](http://blog.csdn.net/yangwen123/article/details/8080924" \t "_blank)一 文中详细介绍了Zygote进程创建android应用进程的过程，当用户点击Luncher上的应用图标时，Luncher进程通过socket向 Zygote进程发送进程创建请求，Zygote进程接受客户端的请求后，通过fork系统调用为应用程序创建相应的进程。本文则介绍android用户 进程的始祖Init进程，Init进程是Linux系统中用户空间的第一个进程，负责创建系统中的关键进程，同时提供属性服务来管理系统属性。

**Android进程模型**

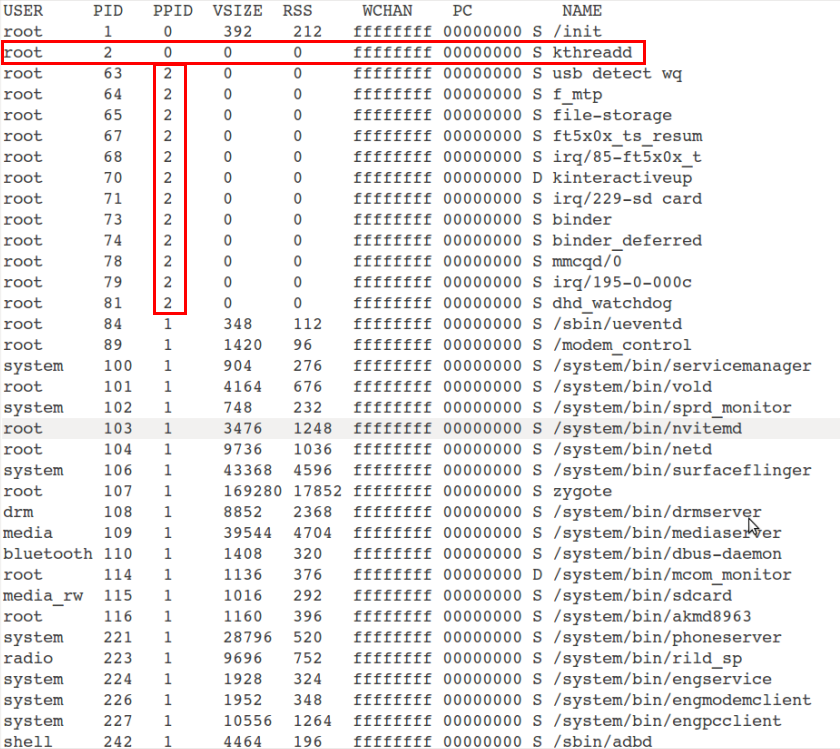
Linux通过调用start\_kernel函数来启动内核，当内核启动模块启动完成后，将启动用户空间的第一个进程——Init进程，下图为Android系统的进程模型图：



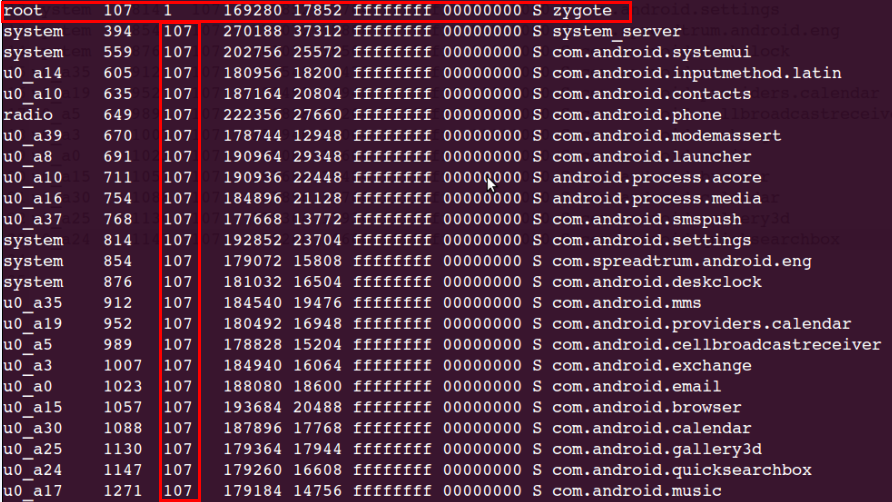
从上图可以看出，Linux内核在启动过程中，创建一个名为Kthreadd的内核进程，PID=2，用于创建内核空间的其他进程；同时创建第一个 用户空间Init进程，该进程PID = 1，用于启动一些本地进程，比如Zygote进程，而Zygote进程也是一个专门用于孵化Java进程的本地进程，上图清晰地描述了整个Android 系统的进程模型，为了证明以上进程模型的正确性，可以通过ps命令来查看进程的PID级PPID，下图显示了Init进程的PID为1，其他的本地进程的 PPID都是1，说明它们的父进程都是Init进程，都是由Init进程启动的。



下图显示kthreadd进程的PID=2，有一部分内核进程如binder、dhd\_watchdog等进程的PPID=2，说明这些进程都是由kthreadd进程创建：



上图中显示zygote进程PID=107，下图显示了zygote进程创建的子进程，从图中可以看到，zygote进程创建的都是Java进程，证明了zygote进程开创了Android系统的Java世界。



上面介绍了Android系统的进程模型设计，接下来将详细分析Init进程。

**Init进程源码分析**

上节介绍了Init进程在Linux内核启动时被创建的，那它是如何启动的呢？

**Init进程启动分析**

在Linux内核启动过程中，将调用Start\_kernel来初始化配置：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. asmlinkage void \_\_init start\_kernel(void)
2. {
3. .............. //执行初始化工作
4. rest\_init();
5. }

start\_kernel函数调用一些初始化函数完成初始化工作后，调用rest\_init()函数来创建新的进程：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

在rest\_init函数里完成两个新进程的创建：Init进程和kthreadd进程，因为Init进程创建在先，所以其PID=1而kthreadd的PID=2，本文只对Init进程进行详细分析，如果读者对kthreadd进行感兴趣，可自行分析。

kernel\_thread函数仅仅调用了fork系统调用来创建新的进程，创建的子进程和父进程都执行在fork函数调用之后的代码，子进程是父进程的一个拷贝。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

在kernel\_init函数中调用\_\_initcall\_start到\_\_initcall\_end之间保存的函数进行驱动模块初始化，然后直接调用init\_post()函数进入用户空间，执行Init 进程代码。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static noinline int init\_post(void)
2. {
4. async\_synchronize\_full();
5. free\_initmem();
6. mark\_rodata\_ro();
7. system\_state = SYSTEM\_RUNNING;
8. numa\_default\_policy();
10. current->signal->flags |= SIGNAL\_UNKILLABLE;
11. //如果ramdisk\_execute\_command不为空，ramdisk\_execute\_command下的Init程序
12. if (ramdisk\_execute\_command) {
13. run\_init\_process(ramdisk\_execute\_command);
14. printk(KERN\_WARNING "Failed to execute %s\n",ramdisk\_execute\_command);
15. }
16. //如果execute\_command不为空，execute\_command下的Init程序
17. if (execute\_command) {
18. run\_init\_process(execute\_command);
19. printk(KERN\_WARNING "Failed to execute %s.  Attempting ""defaults...\n", execute\_command);
20. }
21. //如果以上路径下都没有init程序，就从/sbin、/etc、/bin三个路径下寻找init程序，同时启动一个sh进程
22. run\_init\_process("/sbin/init");
23. run\_init\_process("/etc/init");
24. run\_init\_process("/bin/init");
25. run\_init\_process("/bin/sh");
26. //如果以上路径都没有找到init程序，调用内核panic
27. panic("No init found.  Try passing init= option to kernel. "
28. "See Linux Documentation/init.txt for guidance.");
29. }

当 根文件系统顶层目录中不存在init进程，或未指定启动选项"init="时，内核会到/sbin、/etc、/bin目录下查找init文件。如果在这 些目录中仍未找到init文件，内核就会中止执行init进程，并引发Kernel Panic。run\_init\_process函数通过系统调用do\_execve从内核空间跳转到用户空间，并且执行用户空间的Init程序的入口函 数。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void run\_init\_process(const char \*init\_filename)
2. {
3. argv\_init[0] = init\_filename;
4. kernel\_execve(init\_filename, argv\_init, envp\_init);
5. }

这里就介绍完了内核启动流程，run\_init\_process函数的将执行Init程序的入口函数，Init的入口函数位于/system/core/init/init.c

**Init进程源码分析**

Android的init进程主要功能：  
1)、分析init.rc启动脚本文件，根据文件内容执行相应的功能；  
2)、当一些关键进程死亡时，重启该进程；  
3)、提供Android系统的属性服务；

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

**屏蔽标准的输入输出**

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void open\_devnull\_stdio(void)
2. {
3. int fd;
4. //创建一个字符专用文件/dev/\_\_null\_\_
5. static const char \*name = "/dev/\_\_null\_\_";
6. if (mknod(name, S\_IFCHR | 0600, (1 << 8) | 3) == 0) {
7. //获取/dev/\_\_null\_\_的文件描述符，并输出该文件
8. fd = open(name, O\_RDWR);
9. unlink(name);
10. if (fd >= 0) {
11. //将与进程相关的标准输入(0),标准输出(1),标准错误输出(2)，均定向到NULL设备
12. dup2(fd, 0);
13. dup2(fd, 1);
14. dup2(fd, 2);
15. if (fd > 2) {
16. close(fd);
17. }
18. return;
19. }
20. }
22. exit(1);
23. }

将标准输入输出，错误输出重定向到/dev/\_null\_设备中

**初始化内核log系统**

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void klog\_init(void)
2. {
3. static const char \*name = "/dev/\_\_kmsg\_\_";
4. //创建/dev/\_\_kmsg\_\_设备节点
5. if (mknod(name, S\_IFCHR | 0600, (1 << 8) | 11) == 0) {
6. klog\_fd = open(name, O\_WRONLY);
7. //当进程在进行exec系统调用时，要确保log\_fd是关闭的
8. fcntl(klog\_fd, F\_SETFD, FD\_CLOEXEC);
9. unlink(name);
10. }
11. }

**属性存储空间初始化**

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void property\_init(void)
2. {
3. init\_property\_area();
4. }

关于Android的属性系统，请查看[Android 系统属性SystemProperty分析](http://blog.csdn.net/yangwen123/article/details/8936555" \t "_blank)一文，在这篇文章中详细分析了Android的属性系统。

**读取机器硬件名称**

从/proc/cpuinfo中获取“Hardware”字段信息写入；“Reversion” 字段信息写入

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void get\_hardware\_name(char \*hardware, unsigned int \*revision)
2. {
3. char data[1024];
4. int fd, n;
5. char \*x, \*hw, \*rev;
7. if (hardware[0])
8. return;
9. //打开/proc/cpuinfo文件
10. fd = open("/proc/cpuinfo", O\_RDONLY);
11. if (fd < 0) return;
12. //读取/proc/cpuinfo文件内容
13. n = read(fd, data, 1023);
14. close(fd);
15. if (n < 0) return;
16. data[n] = 0;
17. hw = strstr(data, "\nHardware");
18. rev = strstr(data, "\nRevision");
19. if (hw) {
20. x = strstr(hw, ": ");
21. if (x) {
22. x += 2;
23. n = 0;
24. while (\*x && \*x != '\n') {
25. if (!isspace(\*x))
26. hardware[n++] = tolower(\*x);
27. x++;
28. if (n == 31) break;
29. }
30. hardware[n] = 0;
31. }
32. }
33. if (rev) {
34. x = strstr(rev, ": ");
35. if (x) {
36. \*revision = strtoul(x + 2, 0, 16);
37. }
38. }
39. }

get\_hardware\_name函数从/proc/cpuinfo文件中读取硬件名称等信息，/proc/cpuinfo文件内容如下：

**[plain]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. Processor   : ARMv7 Processor rev 1 (v7l)
2. BogoMIPS    : 1024.00
3. Features    : swp half thumb fastmult vfp edsp thumbee neon vfpv3
4. CPU implementer : 0x41
5. CPU architecture: 7
6. CPU variant : 0x0
7. CPU part    : 0xc05
8. CPU revision    : 1
9. Hardware    : sc7710g
10. Revision    : 0000
11. Serial      : 0000000000000000

**设置命令行参数属性**

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void process\_kernel\_cmdline(void)
2. {
4. chmod("/proc/cmdline", 0440);

7. import\_kernel\_cmdline(0, import\_kernel\_nv);
8. if (qemu[0])
9. import\_kernel\_cmdline(1, import\_kernel\_nv);

12. export\_kernel\_boot\_props();
13. }

process\_kernel\_cmdline 函数首先修改/proc/cmdline文件权限，然后调用import\_kernel\_cmdline函数来读取/proc/cmdline文件的内 容，并查找格式为： = 的字串，调用import\_kernel\_nv函数来设置属性。函数export\_kernel\_boot\_props()用于设置内核启动时需要的属 性。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void import\_kernel\_cmdline(int in\_qemu,void (\*import\_kernel\_nv)(char \*name, int in\_qemu))
2. {
3. char cmdline[1024];
4. char \*ptr;
5. int fd;
6. //打开并读取/proc/cmdline文件
7. fd = open("/proc/cmdline", O\_RDONLY);
8. if (fd >= 0) {
9. int n = read(fd, cmdline, 1023);
10. if (n < 0) n = 0;
12. if (n > 0 && cmdline[n-1] == '\n') n--;
13. cmdline[n] = 0;
14. close(fd);
15. } else {
16. cmdline[0] = 0;
17. }
19. ptr = cmdline;
20. while (ptr && \*ptr) {
21. char \*x = strchr(ptr, ' ');
22. if (x != 0) \*x++ = 0;
23. //回调import\_kernel\_nv函数，in\_qemu =0
24. import\_kernel\_nv(ptr, in\_qemu);
25. ptr = x;
26. }
27. }

/proc/cmdline文件内容如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void import\_kernel\_nv(char \*name, int for\_emulator)
2. {
3. char \*value = strchr(name, '=');
4. int name\_len = strlen(name);
5. if (value == 0) return;
6. \*value++ = 0;
7. if (name\_len == 0) return;
9. #ifdef HAVE\_SELINUX
10. if (!strcmp(name,"enforcing")) {
11. selinux\_enforcing = atoi(value);
12. } else if (!strcmp(name,"selinux")) {
13. selinux\_enabled = atoi(value);
14. }
15. #endif
16. //判断是否为模拟器
17. if (for\_emulator) {
19. char buff[PROP\_NAME\_MAX];
20. int len = snprintf( buff, sizeof(buff), "ro.kernel.%s", name );
21. if (len < (int)sizeof(buff))
22. property\_set( buff, value );
23. return;
24. }
25. //如果/proc/cmdline文件中有qemu关键字
26. if (!strcmp(name,"qemu")) {
27. strlcpy(qemu, value, sizeof(qemu));
28. //如果/proc/cmdline文件中有以androidboot.开头的关键字
29. } else if (!strncmp(name, "androidboot.", 12) && name\_len > 12) {
30. const char \*boot\_prop\_name = name + 12;
31. char prop[PROP\_NAME\_MAX];
32. int cnt;
33. //格式化为ro.boot.xx 属性
34. cnt = snprintf(prop, sizeof(prop), "ro.boot.%s", boot\_prop\_name);
35. if (cnt < PROP\_NAME\_MAX)
36. property\_set(prop, value);
37. }
38. }

最后调用函数export\_kernel\_boot\_props设置内核启动属性

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void export\_kernel\_boot\_props(void)
2. {
3. char tmp[PROP\_VALUE\_MAX];
4. const char \*pval;
5. unsigned i;
6. //属性表
7. struct {
8. const char \*src\_prop;
9. const char \*dest\_prop;
10. const char \*def\_val;
11. } prop\_map[] = {
12. { "ro.boot.serialno", "ro.serialno", "", },
13. { "ro.boot.mode", "ro.bootmode", "unknown", },
14. { "ro.boot.baseband", "ro.baseband", "unknown", },
15. { "ro.boot.bootloader", "ro.bootloader", "unknown", },
16. };
17. //循环读取ro.boot.xxx属性值，并设置ro.xxx属性
18. for (i = 0; i < ARRAY\_SIZE(prop\_map); i++) {
19. pval = property\_get(prop\_map[i].src\_prop);
20. property\_set(prop\_map[i].dest\_prop, pval ?: prop\_map[i].def\_val);
21. }
22. //读取ro.boot.console属性值
23. pval = property\_get("ro.boot.console");
24. if (pval)
25. strlcpy(console, pval, sizeof(console));
26. //读取ro.bootmode属性值
27. strlcpy(bootmode, property\_get("ro.bootmode"), sizeof(bootmode));
28. //读取ro.boot.hardware属性值
29. pval = property\_get("ro.boot.hardware");
30. if (pval)
31. strlcpy(hardware, pval, sizeof(hardware));
32. //设置ro.hardware属性
33. property\_set("ro.hardware", hardware);
34. //设置ro.revision属性
35. snprintf(tmp, PROP\_VALUE\_MAX, "%d", revision);
36. property\_set("ro.revision", tmp);
37. //设置ro.factorytest属性
38. if (!strcmp(bootmode,"factory"))
39. property\_set("ro.factorytest", "1");
40. else if (!strcmp(bootmode,"factory2"))
41. property\_set("ro.factorytest", "2");
42. else
43. property\_set("ro.factorytest", "0");
44. }

**init.rc 文件解析**

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. init\_parse\_config\_file(const char \*fn)
2. {
3. char \*data;
4. //读取/init.rc文件内容
5. data = read\_file(fn, 0);
6. if (!data) return -1;
7. //解析读取到的文件内容
8. parse\_config(fn, data);
9. DUMP();
10. return 0;
11. }

函数首先调用read\_file函数将init.rc文件的内容读取保存到data中，在调用parse\_config对其进行解析

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void \*read\_file(const char \*fn, unsigned \*\_sz)
2. {
3. char \*data;
4. int sz;
5. int fd;
6. struct stat sb;
7. data = 0;
8. //打开/init.rc文件
9. fd = open(fn, O\_RDONLY);
10. if(fd < 0) return 0;
12. // for security reasons, disallow world-writable
13. // or group-writable files
14. if (fstat(fd, &sb) < 0) {
15. ERROR("fstat failed for '%s'\n", fn);
16. goto oops;
17. }
18. if ((sb.st\_mode & (S\_IWGRP | S\_IWOTH)) != 0) {
19. ERROR("skipping insecure file '%s'\n", fn);
20. goto oops;
21. }
22. //将文件指针移到文件尾部，得到文件内容长度
23. sz = lseek(fd, 0, SEEK\_END);
24. if(sz < 0) goto oops;
26. if(lseek(fd, 0, SEEK\_SET) != 0) goto oops;
27. //分配buffer
28. data = (char\*) malloc(sz + 2);
29. if(data == 0) goto oops;
30. //读取文件
31. if(read(fd, data, sz) != sz) goto oops;
32. close(fd);
33. data[sz] = '\n';
34. data[sz+1] = 0;
35. if(\_sz) \*\_sz = sz;
36. return data;
37. oops:
38. close(fd);
39. if(data != 0) free(data);
40. return 0;
41. }

**init.rc文件语法介绍**

在Android根文件系统下存在多个.rc文件，该文件为Android启动配置脚本文件，文件内容如下：

**[plain]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. service keystore /system/bin/keystore /data/misc/keystore
2. class main
3. user keystore
4. group keystore drmrpc
5. socket keystore stream 666

init.rc 是一个可配置的初始化文件,通常定制厂商可以配置额外的初始化配置，如果关键字中有空格，处理方法类似于C语言，使用/表示转义，使用“”防止关键字被断 开，另外注意/在末尾表示换行，由 # （前面允许有空格）开始的行都是注释行。init.rc包含4种状态类别：Actions/Commands/Services/Options。当声明 一个service或者action的时候，它将隐式声明一个section，它之后跟随的command或者option都将属于这个 section，action和service不能重名，否则忽略为error。

**Action**

actions就是在某种条件下触发一系列的命令，通常有一个trigger，形式如：    
  
on  
     

trigger主要包括：

boot 当/init.conf加载完毕时  
= 当被设置为时  
device-added- 设备被添加时  
device-removed- 设备被移除时  
service-exited- 服务退出时

**Service**

service就是要启动的本地服务进程

service [ ]\*

**Option**

option是service的修饰词，由它来指定何时并且如何启动Services程序，主要包括：  
     critical  表示如果服务在4分钟内存在多于4次，则系统重启到recovery mode  
     disabled   表示服务不会自动启动，需要手动调用名字启动  
     setEnv   设置启动环境变量  
     socket [ []] 开启一个unix域的socket，名字为/dev/socket/ , 只能是dgram或者stream,和默认为0  
     user 表示将用户切换为,用户名已经定义好了，只能是system/root  
     group   表示将组切换为  
     oneshot 表示这个service只启动一次  
     class 指定一个要启动的类，这个类中如果有多个service，将会被同时启动。默认的class将会是“default”  
     onrestart  在重启时执行一条命令

**Command**

comand主要包括：

 exec [ ]\*执行一个指定的程序  
 export 设置一个全局变量  
 ifup 使网络接口连接  
 import 引入其他的配置文件  
 hostname 设置主机名  
 chdir 切换工作目录  
 chmod 设置访问权限  
 chown 设置用户和组  
 chroot 设置根目录  
 class\_start 启动类中的service  
 class\_stop 停止类中的service  
 domainname 设置域名  
 insmod 安装模块  
 mkdir [mode] [owner] [group] 创建一个目录，并可以指定权限，用户和组  
 mount

[ ]\* 加载指定设备到目录下 包括"ro", "rw", "remount", "noatime"  
 setprop 设置系统属性  
 setrlimit 设置资源访问权限  
 start 开启服务  
 stop 停止服务  
 symlink 创建一个动态链接  
 sysclktz 设置系统时钟  
 trigger 触发事件  
 write [ ]\* 向路径的文件写入多个

**Properties（属性）**

Init更新一些系统属性以提供对正在发生的事件的监控能力:  
       init.action 此属性值为正在被执行的action的名字，如果没有则为""。  
       init.command  此属性值为正在被执行的command的名字，如果没有则为""。  
       init.svc. 名为的service的状态("stopped"（停止）, "running"（运行）, "restarting"（重启）)

在默认情况下，程序在被init执行时会将标准输出和标准错误都重定向到/dev/null（丢弃）。若你想要获得调试信息，你可以通过 Andoird系统中的logwrapper程序执行你的程序。它会将标准输出/标准错误都重定向到Android日志系统(通过logcat访问)。  
例如：  
    service akmd /system/bin/logwrapper /sbin/akmd

**init.rc解析过程**

1. 扫描init.rc中的token  
    找到其中的 文件结束EOF/文本TEXT/新行NEWLINE，其中的空格‘ ’、‘\t’、‘\r’会被忽略，#开头的行也被忽略掉；而对于TEXT，空格‘ ’、‘\t’、‘\r’、‘\n’都是TEXT的结束标志。  
2. 对每一个TEXT token，都加入到args[]数组中  
3. 当遇到新一行（‘\n’）的时候，用args[0]通过lookup\_keyword()检索匹配关键字；

   1) 对Section（on和service），调用parse\_new\_section() 解析：  
     - 对on section，调用parse\_action()，并设置解析函数parse\_line为parse\_line\_action()  
     - 对service section，调用parse\_service()，并设置解析函数parse\_line为parse\_line\_service()  
   2) 对其他关键字的行（非on或service开头的地方，也就是没有切换section）调用parse\_line()  
     - 对于on section内的命令行，调用parse\_line\_action()解析；  
     - 对于service section内的命令行，调用parse\_line\_service()解析。

Token的定义

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. #define T\_EOF 0
2. #define T\_TEXT 1
3. #define T\_NEWLINE 2

 解 析过程中的双向循环链表的使用，android用到了一个非常巧妙的链表实现方法，一般情况下如果链表的节点是一个单独的数据结构的话，那么针对不同的数 据结构，都需要定义不同链表操作。而在初始化过程中使用到的链表则解决了这个问题，它将链表的节点定义为了一个非常精简的结构，只包含前向和后向指针，那 么在定义不同的数据结构时，只需要将链表节点嵌入到数据结构中即可。链表节点定义如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct listnode
2. {
3. struct listnode \*next;
4. struct listnode \*prev;
5. };

对于Action数据结构为例：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct action {
3. struct listnode alist;
5. struct listnode qlist;
7. struct listnode tlist;
9. unsigned hash;
10. const char \*name;
12. struct listnode commands;
13. struct command \*current;
14. };

这 样的话，所有的链表的基本操作，例如插入，删除等只会针对listnode进行操作，而不是针对特定的数据结构，链表的实现得到了统一，即精简了代码，又 提高了效率。 但是这样的链表实现，存在一个问题，链表节点listnode中只有前向和后向指针，并且前向和后向指针均指向listnode，那么我们通过什么方式来 访问数据结构action的内容呢？我们使用offsetof宏来计算链表节点在数据结构中的偏移量，从而计算数据结构实例的地址。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. #define node\_to\_item(node, container, member) \
2. (container \*) (((char\*) (node)) - offsetof(container, member))

这种链表的优点：(1)所有链表基本操作都是基于listnode指针的，因此添加类型时，不需要重复写链表基本操作函数(2)一个container数据结构可以含有多个listnode成员，这样就可以同时挂到多个不同的链表中。

Service数据结构定义：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct service {
3. struct listnode slist;
4. const char \*name;
5. const char \*classname;
6. unsigned flags;
7. pid\_t pid;
8. time\_t time\_started;
9. time\_t time\_crashed;
10. int nr\_crashed;
12. uid\_t uid;
13. gid\_t gid;
14. gid\_t supp\_gids[NR\_SVC\_SUPP\_GIDS];
15. size\_t nr\_supp\_gids;
17. #ifdef HAVE\_SELINUX
18. char \*seclabel;
19. #endif
20. struct socketinfo \*sockets;
21. struct svcenvinfo \*envvars;
22. struct action onrestart;
24. int \*keycodes;
25. int nkeycodes;
26. int keychord\_id;
27. int ioprio\_class;
28. int ioprio\_pri;
29. int nargs;
31. char \*args[1];
32. };

对于某些Service可能采用Socket来实现进程间通信，因此该Service需要创建多个socket，比如：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. service wril-daemon /system/bin/rild\_sp -l /system/lib/libreference-ril\_sp.so -m w -n 0
2. class core
3. socket rild stream 660 root radio
4. socket rild-debug stream 660 radio system
5. disabled
6. user root
7. group radio cache inet misc audio sdcard\_rw log

该service需要创建rild 和rild-debug socket，这些socket的信息在解析init.rc文件时保存在Service的成员变量sockets链表中。socketinfo 数据结构定义如下:

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct socketinfo {
2. struct socketinfo \*next;
3. const char \*name;
4. const char \*type;
5. uid\_t uid;
6. gid\_t gid;
7. int perm;
8. };

某些Service在运行时需要设置环境变量，这些环境变量被保存在Service的成员变量envvars链表中,svcenvinfo 数据结构定义如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct svcenvinfo {
2. struct svcenvinfo \*next;
3. const char \*name;
4. const char \*value;
5. };

在每个Action或Service下可能需要执行多个Command，关于command数据结构定义如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

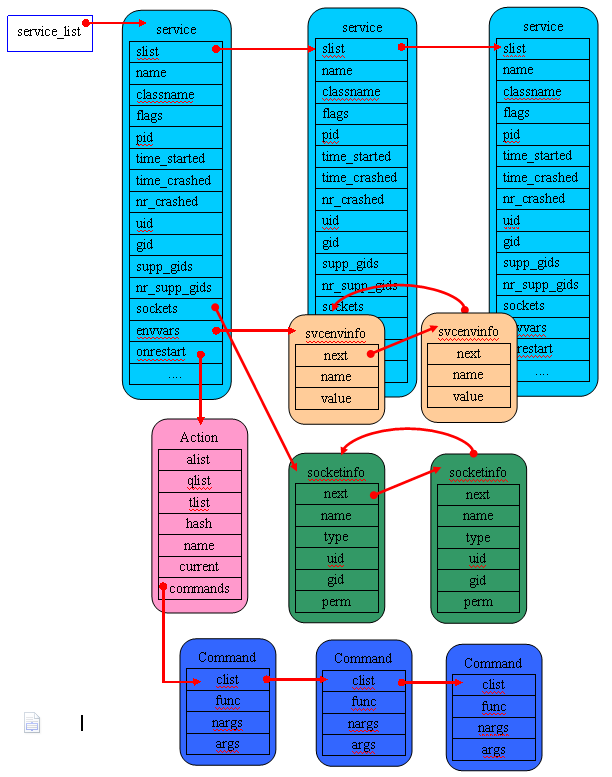
1. struct command
2. {
4. struct listnode clist;
6. int (\*func)(int nargs, char \*\*args);
7. int nargs;
8. char \*args[1];
9. };

在Init进程中分别使用了3个链表来存储init.rc文件中的Action和Service：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

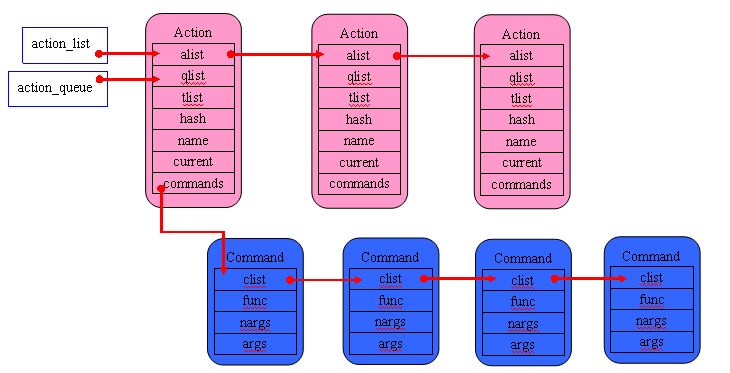
1. static list\_declare(service\_list);
2. static list\_declare(action\_list);
3. static list\_declare(action\_queue);

service\_list链表用于保存init.rc文件中的Service配置信息，service\_list链表的存储如下图所示：



service\_list 链表保存init.rc文件中的所有service，每个service下的所有socket信息保存在该service的成员变量sockets链表 中，当该service重启时，需要重启某些服务，对于重启某些服务的命令以Action的形式保存在Service的成员变量onrestart链表 中，而真正执行的命令却存放在该Action下的commands链表里。

action\_list用于保存init.rc文件中的所有以on开头的section，action\_list链表的存储如下图所示：



从上图可以看出action\_queue和action\_list都是用来保存所有的Action，它们之间的区别是action\_list用于保 存从init.rc中解析出来的所有Action,而action\_queue却是用于保存待执行的Action，action\_queue是一个待执行 队列。

在system\core\init\keywords.h文件中定义了解析关键字，其内容如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

宏KEYWORD并未定义，因此将定义宏\_\_MAKE\_KEYWORD\_ENUM\_\_ 及KEYWORD，KEYWORD宏定义如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

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

同时定义了枚举：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

该枚举的定义主要是为每个命令指定对应的序号。在keywords.h文件最后取消了宏\_\_MAKE\_KEYWORD\_ENUM\_\_ 及KEYWORD的定义，在system\core\init\init\_parser.c文件中又重定义了KEYWORD宏：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. #define KEYWORD(symbol, flags, nargs, func) \
2. [ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

该宏的定义是为了给接下来定义的keyword\_info这个关键字信息数组的赋值，keyword\_info定义如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. struct {
2. const char \*name;
3. int (\*func)(int nargs, char \*\*args);
4. unsigned char nargs;
5. unsigned char flags;
6. } keyword\_info[KEYWORD\_COUNT] = {
7. [ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },
8. #include "keywords.h"
9. };

keyword\_info数组元素是keywords.h文件中的内容，因为此时KEYWORD宏已经被定义了同时\_\_MAKE\_KEYWORD\_ENUM\_\_被取消定义，因此keywords.h文件内容此时变为：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. KEYWORD(capability,  OPTION,  0, 0)
2. KEYWORD(chdir,       COMMAND, 1, do\_chdir)

使用上述KEYWORD宏展开得到keyword\_info数组内容如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. [ K\_capability      ] = { capability,   0,              1,  OPTION, },
2. [ K\_class           ] = { class,        0,              1,  OPTION, },
3. [ K\_console         ] = { console,      0,              1,  OPTION, },
4. [ K\_critical        ] = { critical,     0,              1,  OPTION, },
5. [ K\_group           ] = { group,        0,              1,  OPTION, },
6. [ K\_disabled        ] = { disabled,     0,              1,  OPTION, },

了解了这些内容之后，我们开始分析init.rc文件的真正解析过程：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

3. Action解析

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void \*parse\_action(struct parse\_state \*state, int nargs, char \*\*args)
2. {
3. struct action \*act;
4. if (nargs < 2) {
5. parse\_error(state, "actions must have a trigger\n");
6. return 0;
7. }
8. if (nargs > 2) {
9. parse\_error(state, "actions may not have extra parameters\n");
10. return 0;
11. }
12. //创建一个action
13. act = calloc(1, sizeof(\*act));
14. act->name = args[1];
15. list\_init(&act->commands);
16. //添加到action\_list链表中
17. list\_add\_tail(&action\_list, &act->alist);
18. return act;
19. }

解析到新的on section调用parse\_action()时，申请了struct action \*act，设置：

     1) act->name为on section的名字（比如boot/fs/）；

     2) 初始化list act->*commands*；

     3) 把act->*alist*加入到**action\_list**的列尾

这样，action创建并加入到了**action\_list**中。

4.Action 命令解析

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void parse\_line\_action(struct parse\_state\* state, int nargs, char \*\*args)
2. {
3. struct command \*cmd;
4. //获取解析得到的action
5. struct action \*act = state->context;
6. int (\*func)(int nargs, char \*\*args);
7. int kw, n;
8. if (nargs == 0) {
9. return;
10. }
11. //查找关键字类型
12. kw = lookup\_keyword(args[0]);
13. //如果不是COMMAND类型，跳出不处理
14. if (!kw\_is(kw, COMMAND)) {
15. parse\_error(state, "invalid command '%s'\n", args[0]);
16. return;
17. }
18. //得到命令参数个数，验证参数个数的合法性
19. n = kw\_nargs(kw);
20. if (nargs < n) {
21. parse\_error(state, "%s requires %d %s\n", args[0], n - 1,
22. n > 2 ? "arguments" : "argument");
23. return;
24. }
25. //创建命令command
26. cmd = malloc(sizeof(\*cmd) + sizeof(char\*) \* nargs);
27. cmd->func = kw\_func(kw);
28. cmd->nargs = nargs;
29. memcpy(cmd->args, args, sizeof(char\*) \* nargs);
30. //将command添加到act->commands链表中
31. list\_add\_tail(&act->commands, &cmd->clist);
32. }

action里的command的解析  
    对on section内action里的command，调用parse\_line\_action()  
     1) 查找关键字，核对是否是COMMAND，参数数目是否正确  
     2) 申请struct command \*cmd  
       - cmd->func从keyword表中获取；  
       - 设置参数个数给cmd->nargs，拷贝参数给cmd->args；  
       - 把cmd->clist加入到act->commands的列尾

这样，command加入到了action中。

5. import 命令解析

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void parse\_import(struct parse\_state \*state, int nargs, char \*\*args)
2. {
3. struct listnode \*import\_list = state->priv;
4. struct import \*import;
5. char conf\_file[PATH\_MAX];
6. int ret;
7. //参数个数判断
8. if (nargs != 2) {
9. ERROR("single argument needed for import\n");
10. return;
11. }
13. ret = expand\_props(conf\_file, args[1], sizeof(conf\_file));
14. if (ret) {
15. ERROR("error while handling import on line '%d' in '%s'\n",
16. state->line, state->filename);
17. return;
18. }
19. //创建一个import
20. import = calloc(1, sizeof(struct import));
21. //设置import文件名称
22. import->filename = strdup(conf\_file);
23. //添加到import->list链表中
24. list\_add\_tail(import\_list, &import->list);
25. INFO("found import '%s', adding to import list", import->filename);
26. }

当init.rc文件解析完成后，将从import\_list链表中取出通过关键字import导入的其他rc文件，并调用init\_parse\_config\_file函数进行解析：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. list\_for\_each(node, &import\_list) {
2. struct import \*import = node\_to\_item(node, struct import, list);
3. int ret;
5. INFO("importing '%s'", import->filename);
6. ret = init\_parse\_config\_file(import->filename);
7. if (ret)
8. ERROR("could not import file '%s' from '%s'\n",
9. import->filename, fn);
10. }

到此init.rc文件就解析完成，文件内容全部存储在service\_list和action\_list链表中。

**添加Action到待执行队列**

 当解析完所有的init.rc内容之后，在执行这些action之前，需要按顺序将其置于一个待执行队列中

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void action\_for\_each\_trigger(const char \*trigger,void (\*func)(struct action \*act))
2. {
3. struct listnode \*node;
4. struct action \*act;
5. //遍历action\_list链表，根据名字查找相关的action
6. list\_for\_each(node, &action\_list) {
7. act = node\_to\_item(node, struct action, alist);
8. if (!strcmp(act->name, trigger)) {
9. //回调action\_add\_queue\_tail函数
10. func(act);
11. }
12. }
13. }

从action\_list链表中查询指定名称的action，并调用函数action\_add\_queue\_tail将其添加到待执行队列action\_queue中。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void action\_add\_queue\_tail(struct action \*act)
2. {
3. list\_add\_tail(&action\_queue, &act->qlist);
4. }

action\_for\_each\_trigger()把队列action\_list里所匹配的action，追加到action\_queue的队尾

**构建新的Action**

还有一些没有在init.rc中定义的action，相比init.rc，这些action的共同点是没有参数，对于这类action，通过queue\_builtin\_action()函数来构建

queue\_builtin\_action()把执行的函数组成command，创建action，挂在**action\_list**上，并追加到**action\_queue**的队尾。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

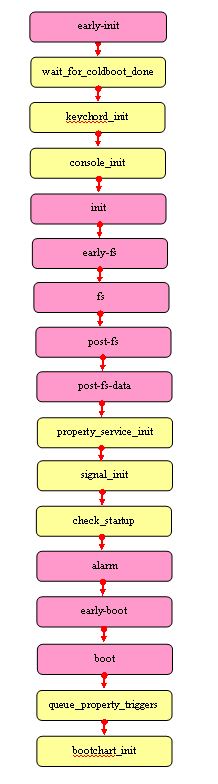
1. void queue\_builtin\_action(int (\*func)(int nargs, char \*\*args), char \*name)
2. {
3. struct action \*act;
4. struct command \*cmd;
5. //创建一个Action
6. act = calloc(1, sizeof(\*act));
7. act->name = name;
8. list\_init(&act->commands);
9. //为该Action创建一个command
10. cmd = calloc(1, sizeof(\*cmd));
11. cmd->func = func;
12. cmd->args[0] = name;
13. //将该command添加到Action的commands链表中
14. list\_add\_tail(&act->commands, &cmd->clist);
15. //将该Action添加到action\_list链表中
16. list\_add\_tail(&action\_list, &act->alist);
17. //将该Action添加到待执行队列action\_queue中
18. action\_add\_queue\_tail(act);
19. }

**添加Action到待执行队列**

init 进程通过action\_for\_each\_trigger 和queue\_builtin\_action 函数向待执行队列action\_queue依次添加了以下Action：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. action\_for\_each\_trigger("early-init", action\_add\_queue\_tail);
2. queue\_builtin\_action(wait\_for\_coldboot\_done\_action, "wait\_for\_coldboot\_done");
3. queue\_builtin\_action(keychord\_init\_action, "keychord\_init");
4. queue\_builtin\_action(console\_init\_action, "console\_init");
6. action\_for\_each\_trigger("init", action\_add\_queue\_tail);
8. action\_for\_each\_trigger("early-fs", action\_add\_queue\_tail);
9. action\_for\_each\_trigger("fs", action\_add\_queue\_tail);
10. action\_for\_each\_trigger("post-fs", action\_add\_queue\_tail);
11. if (!is\_charger) {
12. //action\_for\_each\_trigger("post-fs", action\_add\_queue\_tail);
13. action\_for\_each\_trigger("post-fs-data", action\_add\_queue\_tail);
14. }
15. queue\_builtin\_action(property\_service\_init\_action, "property\_service\_init");
16. queue\_builtin\_action(signal\_init\_action, "signal\_init");
17. queue\_builtin\_action(check\_startup\_action, "check\_startup");
18. if (!strcmp(bootmode, "alarm")) {
19. action\_for\_each\_trigger("alarm", action\_add\_queue\_tail);
20. }
21. if (is\_charger) {
22. action\_for\_each\_trigger("charger", action\_add\_queue\_tail);
23. } else {
24. action\_for\_each\_trigger("early-boot", action\_add\_queue\_tail);
25. action\_for\_each\_trigger("boot", action\_add\_queue\_tail);
26. }
28. queue\_builtin\_action(queue\_property\_triggers\_action, "queue\_property\_triggers");
29. #if BOOTCHART
30. queue\_builtin\_action(bootchart\_init\_action, "bootchart\_init");
31. #endif



early-init  
    查看init.rc中的相应字符段为start ueventd  
    这个action主要目的是通过early-init启动ueventd服务，这个服务负责uevent(user space event)的处理，uevent是内核向用户空间发出的一个时间通知，使应用程序能够有机会对该event做出反应。

wait\_for\_coldboot\_done  
    android 冷过程结束后会生成dev/.coldboot\_done文件，wait\_for\_coldboot\_done这个action会等待dev /.coldboot\_done文件的生成，等待时长为5s。当然这个action不会阻塞android的冷启动过程，它会每查询一次就会休眠 0.1s，直到冷启动结束。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int wait\_for\_coldboot\_done\_action(int nargs, char \*\*args)
2. {
3. int ret;
4. INFO("wait for %s\n", coldboot\_done);
5. //  /dev/.coldboot\_done
6. //#define COMMAND\_RETRY\_TIMEOUT 5
7. ret = wait\_for\_file(coldboot\_done, COMMAND\_RETRY\_TIMEOUT);
8. if (ret)
9. ERROR("Timed out waiting for %s\n", coldboot\_done);
10. return ret;
11. }

keychord\_init

keychord是组合按键，Android暂时还不支持keychord机制，keychord机制就是在init.rc文件中为每个服务配置组合键，在服务解析时为指定服务设置相应的键码值。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int keychord\_init\_action(int nargs, char \*\*args)
2. {
3. keychord\_init();
4. return 0;
5. }

调用keychord\_init函数来初始化组合键机制。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void keychord\_init()
2. {
3. int fd, ret;
4. //遍历service\_list链表，为每个service分配keychord\_id
5. service\_for\_each(add\_service\_keycodes);

8. if (!keychords)
9. return;
10. //打开/dev/keychord设备文件
11. fd = open("/dev/keychord", O\_RDWR);
12. if (fd < 0) {
13. ERROR("could not open /dev/keychord\n");
14. return;
15. }
16. //设置设备属性
17. fcntl(fd, F\_SETFD, FD\_CLOEXEC);
18. //将keychords数组内容写入设备文件中
19. ret = write(fd, keychords, keychords\_length);
20. if (ret != keychords\_length) {
21. ERROR("could not configure /dev/keychord %d (%d)\n", ret, errno);
22. close(fd);
23. fd = -1;
24. }
25. free(keychords);
26. keychords = 0;
27. keychord\_fd = fd;
28. }

console\_init  
    1.如果/proc/cmdline指定了控制台终端，那么优先使用这个控制台，如果没有指定，那么将使用默认控制台终端/dev/console。  
    2.加载开机图片,参考load\_565rle\_image函数  
    a,通过ioctl函数修改dev/tty0(即终端控制台)为图像显示模式；  
    b,尝试打开/initlogo.rle,如果失败，那么将dev/tty0恢复为文本显示模式，则开机时显示"ANDROID"文字；  
    c,如果打开/initlogo.rle成功，那么init将会打开Framebuffer；

    d,将initlogo.rle数据写到Framebuffer中。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int console\_init\_action(int nargs, char \*\*args)
2. {
3. int fd;
4. char tmp[PROP\_VALUE\_MAX];
6. if (console[0]) {
7. snprintf(tmp, sizeof(tmp), "/dev/%s", console);
8. console\_name = strdup(tmp);
9. }
11. fd = open(console\_name, O\_RDWR);
12. if (fd >= 0)
13. have\_console = 1;
14. close(fd);
15. //加载开机图片
16. if( load\_565rle\_image(INIT\_IMAGE\_FILE) ) {
17. fd = open("/dev/tty0", O\_WRONLY);
18. if (fd >= 0) {
19. const char \*msg;
20. msg = "\n"
21. "\n"
22. "\n"
23. "\n"
24. "\n"
25. "\n"
26. "\n"  // console is 40 cols x 30 lines
27. "\n"
28. "\n"
29. "\n"
30. "\n"
31. "\n"
32. "\n"
33. "\n"
34. "             A N D R O I D ";
35. write(fd, msg, strlen(msg));
36. close(fd);
37. }
38. }
39. return 0;
40. }

load\_565rle\_image()函数将加载由参数传递过来的图像文件，而后将该文件显示在LCD屏幕上。

property\_service\_init

       读取属性文件，并设置相关属性。关于Android属性系统，请查看[Android 系统属性SystemProperty分析](http://blog.csdn.net/yangwen123/article/details/8936555" \t "_blank)

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int property\_service\_init\_action(int nargs, char \*\*args)
2. {
4. start\_property\_service();
5. return 0;
6. }

signal\_init  
        
    创建套接字对，以便init进程在收到子进程终止的SIGCHLD信号时调用相应的handler

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int signal\_init\_action(int nargs, char \*\*args)
2. {
3. signal\_init();
4. return 0;
5. }

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void signal\_init(void)
2. {
3. int s[2];
4. struct sigaction act;
5. act.sa\_handler = sigchld\_handler; //设置handler回调函数
6. act.sa\_flags = SA\_NOCLDSTOP;
7. act.sa\_mask = 0;
8. act.sa\_restorer = NULL;
9. sigaction(SIGCHLD, &act, 0); //安装信号处理器

12. if (socketpair(AF\_UNIX, SOCK\_STREAM, 0, s) == 0) {
13. signal\_fd = s[0];
14. signal\_recv\_fd = s[1];
15. fcntl(s[0], F\_SETFD, FD\_CLOEXEC);
16. fcntl(s[0], F\_SETFL, O\_NONBLOCK);
17. fcntl(s[1], F\_SETFD, FD\_CLOEXEC);
18. fcntl(s[1], F\_SETFL, O\_NONBLOCK);
19. }
20. handle\_signal();
21. }

init 进程定义了handler，用于处理子进程的终止，当子进程死亡时将向父进程发送SIGCHLD信号，为了调用相关handler，init进程会通过 socket连接SIGCHLD信号的handler，socketpair()函数会创建一对已经连接的套接字，事件处理handler会监视 signal\_recv\_fd的值，当其值为1时，init进程就会调用子进程停止处理函数handler。

check\_startup

    检查属性socket句柄及信号句柄是否安装成功

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int check\_startup\_action(int nargs, char \*\*args)
2. {
4. if ((get\_property\_set\_fd() < 0) ||(get\_signal\_fd() < 0)) {
5. ERROR("init startup failure\n");
6. exit(1);
7. }
9. unlink("/dev/.booting");
10. return 0;
11. }

queue\_property\_triggers  
   根据当前属性值来触发该属性对应的动作

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int queue\_property\_triggers\_action(int nargs, char \*\*args)
2. {
3. queue\_all\_property\_triggers();
5. property\_triggers\_enabled = 1;
6. return 0;
7. }

调用queue\_all\_property\_triggers()函数来检查init.rc文件中配置的属性触发条件是否满足，如果满足，则将该Action添加到待执行队列中：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void queue\_all\_property\_triggers()
2. {
3. struct listnode \*node;
4. struct action \*act;
5. //遍历action\_list链表
6. list\_for\_each(node, &action\_list) {
7. //取得每个节点下对应的action
8. act = node\_to\_item(node, struct action, alist);
9. //如果该action的名字以property开头
10. if (!strncmp(act->name, "property:", strlen("property:"))) {
11. //读取该属性的名称
12. const char\* name = act->name + strlen("property:");
13. //读取该属性的值
14. const char\* equals = strchr(name, '=');
15. if (equals) {
16. char prop\_name[PROP\_NAME\_MAX + 1];
17. const char\* value;
18. int length = equals - name;
19. if (length > PROP\_NAME\_MAX) {
20. ERROR("property name too long in trigger %s", act->name);
21. } else {
22. memcpy(prop\_name, name, length);
23. prop\_name[length] = 0;

26. value = property\_get(prop\_name);
27. //如果属性系统中的值等于init.rc文件中设置的触发值
28. if (value && (!strcmp(equals + 1, value) ||!strcmp(equals + 1, "\*"))) {
29. //将该Action添加到待执行队列action\_queue中
30. action\_add\_queue\_tail(act);
31. }
32. }
33. }
34. }
35. }
36. }

Action - boot

  在boot动作中启动所有的service服务，启动命令如下：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. class\_start core
2. class\_start main

我们可以在service配置中通过关键字class 将service分为不同的类别，从而可以通过class\_start 或class\_stop 来启动或停止某一类型的service，如下将adbd服务设置为core类型的服务：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. service adbd /sbin/adbd
2. class core
3. disabled

class\_start core 表示启动所有类型为core的服务：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. int do\_class\_start(int nargs, char \*\*args)
2. {
4. service\_for\_each\_class(args[1], service\_start\_if\_not\_disabled);
5. return 0;
6. }

函数中args[1]指定该服务所属类型，service\_start\_if\_not\_disabled是启动服务的回调函数；

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void service\_for\_each\_class(const char \*classname,
2. void (\*func)(struct service \*svc))
3. {
4. struct listnode \*node;
5. struct service \*svc;
6. list\_for\_each(node, &service\_list) {
7. svc = node\_to\_item(node, struct service, slist);
8. if (!strcmp(svc->classname, classname)) {
9. func(svc);
10. }
11. }
12. }

service\_for\_each\_class函数通过遍历service\_list服务链表来查找指定类型名称的服务，并调用函数service\_start\_if\_not\_disabled来启动服务。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void service\_start\_if\_not\_disabled(struct service \*svc)
2. {
3. if (!(svc->flags & SVC\_DISABLED)) {
4. service\_start(svc, NULL);
5. }
6. }

函数service\_start\_if\_not\_disabled()首先判断该服务的标志位是否设置成了SVC\_DISABLED，SVC\_DISABLED标志着服务不能在开始时启动，如果服务没有设置此标志位，则启动该服务，

service\_start()函数比较复杂，这里就不详细分析，service\_start()函数主要完成以下工作：

1）设置服务标志位

2）调用fork()系统调用创建新的进程；

3）获取属性匿名存储空间句柄，并添加为服务配置的环境变量；

4）创建服务配置的socket，调用publish\_socket函数将创建的socket句柄添加到环境变量中；该环境变量为：ANDROID\_SOCKET\_XXX = fd

5）为新进程打开控制台，并设置新进程的PID,GID等；

6）调用execve()系统调用执行新进程运行的程序；

7）设置服务运行状态属性；该属性为：init.svc.XXX = running

**Init进程循环执行**

当将以上Action添加到待执行队列中后，init进程将进入无限循环中执行，循环过程中主要完成以下工作：

A. 调用函数execute\_one\_command来检查action\_queue列表是否为空。如果不为空的话，那么init进程就会将保存在列表头中的 action移除，并且执行这个被移除的action。由于前面我们将一个名称为“console\_init”的action添加到了 action\_queue列表中，因此，在这个无限循环中，这个action就会被执行，即函数console\_init\_action会被调用。  
B. 调用函数restart\_processes来检查系统中是否有进程需要重启。在启动脚本/init.rc中，我们可以指定一个进程在退出之后会自动重新 启动。在这种情况下，函数restart\_processes就会检查是否存在需要重新启动的进程，如果存在的话，那么就会将它重新启动起来。  
C. 处理系统属性变化事件。当我们调用函数property\_set来改变一个系统属性值时，系统就会通过一个socket（通过调用函数 get\_property\_set\_fd可以获得它的文件描述符）来向init进程发送一个属性值改变事件通知。init进程接收到这个属性值改变事件之 后，就会调用函数handle\_property\_set\_fd来进行相应的处理。后面在分析第三个开机画面的显示过程时，我们就会看 到，SurfaceFlinger服务就是通过修改“ctl.start”和“ctl.stop”属性值来启动和停止第三个开机画面的。  
D. 处理一种称为“chorded keyboard”的键盘输入事件。这种类型为chorded keyboard”的键盘设备通过不同的铵键组合来描述不同的命令或者操作，它对应的设备文件为/dev/keychord。我们可以通过调用函数 get\_keychord\_fd来获得这个设备的文件描述符，以便可以监控它的输入事件，并且调用函数handle\_keychord来对这些输入事件进 行处理。  
E. 回收僵尸进程。我们知道，在Linux内核中，如果父进程不等待子进程结束就退出，那么当子进程结束的时候，就会变成一个僵尸进程，从而占用系统的资源。 为了回收这些僵尸进程，init进程会安装一个SIGCHLD信号接收器。当那些父进程已经退出了的子进程退出的时候，内核就会发出一个SIGCHLD信 号给init进程。init进程可以通过一个socket（通过调用函数get\_signal\_fd可以获得它的文件描述符）来将接收到的SIGCHLD 信号读取回来，并且调用函数handle\_signal来对接收到的SIGCHLD信号进行处理，即回收那些已经变成了僵尸的子进程。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. for(;;) {
2. int nr, i, timeout = -1;
3. execute\_one\_command();①
4. restart\_processes();②
5. if (!property\_set\_fd\_init && get\_property\_set\_fd() > 0) {③
6. ufds[fd\_count].fd = get\_property\_set\_fd();
7. ufds[fd\_count].events = POLLIN;
8. ufds[fd\_count].revents = 0;
9. fd\_count++;
10. property\_set\_fd\_init = 1;
11. }
12. if (!signal\_fd\_init && get\_signal\_fd() > 0) {
13. ufds[fd\_count].fd = get\_signal\_fd();
14. ufds[fd\_count].events = POLLIN;
15. ufds[fd\_count].revents = 0;
16. fd\_count++;
17. signal\_fd\_init = 1;
18. }
19. if (!keychord\_fd\_init && get\_keychord\_fd() > 0) {
20. ufds[fd\_count].fd = get\_keychord\_fd();
21. ufds[fd\_count].events = POLLIN;
22. ufds[fd\_count].revents = 0;
23. fd\_count++;
24. keychord\_fd\_init = 1;
25. }
27. if (process\_needs\_restart) {④
28. timeout = (process\_needs\_restart - gettime()) \* 1000;
29. if (timeout < 0)
30. timeout = 0;
31. }
33. if (!action\_queue\_empty() || cur\_action)
34. timeout = 0;
36. #if BOOTCHART
37. if (bootchart\_count > 0) {
38. if (timeout < 0 || timeout > BOOTCHART\_POLLING\_MS)
39. timeout = BOOTCHART\_POLLING\_MS;
40. if (bootchart\_step() < 0 || --bootchart\_count == 0) {
41. bootchart\_finish();
42. bootchart\_count = 0;
43. }
44. }
45. #endif
47. nr = poll(ufds, fd\_count, timeout);⑤
48. if (nr <= 0)
49. continue;
51. for (i = 0; i < fd\_count; i++) {
52. if (ufds[i].revents == POLLIN) {
53. if (ufds[i].fd == get\_property\_set\_fd())
54. handle\_property\_set\_fd();⑥
55. else if (ufds[i].fd == get\_keychord\_fd())
56. handle\_keychord();⑦
57. else if (ufds[i].fd == get\_signal\_fd())
58. handle\_signal();⑧
59. }
60. }
61. }

1).execute\_one\_command(void)

从待执行队列action\_queue中取出一个Action来执行，并且将已经执行完的Action从action\_queue队列中移除。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void execute\_one\_command(void)
2. {
3. int ret;
5. if (!cur\_action || !cur\_command || is\_last\_command(cur\_action, cur\_command)) {
6. cur\_action = action\_remove\_queue\_head();
7. cur\_command = NULL;
8. if (!cur\_action)
9. return;
10. INFO("processing action %p (%s)\n", cur\_action, cur\_action->name);
11. cur\_command = get\_first\_command(cur\_action);
12. } else {
13. cur\_command = get\_next\_command(cur\_action, cur\_command);
14. }
16. if (!cur\_command)
17. return;
19. ret = cur\_command->func(cur\_command->nargs, cur\_command->args);
20. INFO("command '%s' r=%d\n", cur\_command->args[0], ret);
21. }

      1) 从action\_queue取下struct action \*act赋给cur\_action；  
  
      2) 从cur\_action获得struct command \*赋给cur\_command；  
  
      3) 执行cur\_command->func(cur\_command->nargs, cur\_command->args)

2).服务重启

当内存不足时，Android系统会自动杀死一下进程来释放空间，所以当某些重要的服务被杀，同时该服务进程并未设置为oneshot，则必须重新启动该服务进程。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void restart\_processes()
2. {
3. process\_needs\_restart = 0;
4. service\_for\_each\_flags(SVC\_RESTARTING,restart\_service\_if\_needed);
5. }

调 用函数service\_for\_each\_flags来循环遍历服务链表，查找标志位为SVC\_RESTARTING的服务，当该服务进程死亡 时，init进程监控到进程死亡事件，在处理该事件的时候会为该服务进程设置SVC\_RESTARTING标志位，并调用 restart\_service\_if\_needed函数重启服务

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void service\_for\_each\_flags(unsigned matchflags,
2. void (\*func)(struct service \*svc))
3. {
4. struct listnode \*node;
5. struct service \*svc;
6. list\_for\_each(node, &service\_list) {
7. svc = node\_to\_item(node, struct service, slist);
8. if (svc->flags & matchflags) {
9. func(svc);
10. }
11. }
12. }

从服务链表中查找具有相同标志位的服务，并调用回调函数进行处理，对于具有SVC\_RESTARTING标志的服务，说明该服务需要重启，

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void restart\_service\_if\_needed(struct service \*svc)
2. {
3. time\_t next\_start\_time = svc->time\_started + 5;
5. if (next\_start\_time <= gettime()) {
6. svc->flags &= (~SVC\_RESTARTING);
7. service\_start(svc, NULL);
8. return;
9. }
11. if ((next\_start\_time < process\_needs\_restart) ||
12. (process\_needs\_restart == 0)) {
13. process\_needs\_restart = next\_start\_time;
14. }
15. }

当当前时间大于服务启动时间时，清楚服务重启标志并启动该服务，service\_start()函数已经在前面简单介绍过了。那服务重启标志位是在哪里设置的呢？在接下来介绍的init进程处理子进程死亡信号SIGCHLD时会进行详细介绍。

3.设置句柄池

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. if (!property\_set\_fd\_init && get\_property\_set\_fd() > 0) {
2. ufds[fd\_count].fd = get\_property\_set\_fd();
3. ufds[fd\_count].events = POLLIN;
4. ufds[fd\_count].revents = 0;
5. fd\_count++;
6. property\_set\_fd\_init = 1;
7. }
8. if (!signal\_fd\_init && get\_signal\_fd() > 0) {
9. ufds[fd\_count].fd = get\_signal\_fd();
10. ufds[fd\_count].events = POLLIN;
11. ufds[fd\_count].revents = 0;
12. fd\_count++;
13. signal\_fd\_init = 1;
14. }
15. if (!keychord\_fd\_init && get\_keychord\_fd() > 0) {
16. ufds[fd\_count].fd = get\_keychord\_fd();
17. ufds[fd\_count].events = POLLIN;
18. ufds[fd\_count].revents = 0;
19. fd\_count++;
20. keychord\_fd\_init = 1;
21. }

get\_property\_set\_fd() 函数用于得到属性socket设备/dev/socket/property\_service的句柄property\_set\_fd，并添加到句柄次 ufds中；property\_set\_fd\_init标志位的设置是为了在下一次循环中不在执行这部分代码，从而避免了重复添加句柄的工作。

get\_signal\_fd()函数用于得到安装信号处理时创建的socket对的接收端句柄signal\_recv\_fd；signal\_fd\_init和property\_set\_fd\_init的作用相同；

get\_keychord\_fd()函数用于得到设备/dev/keychord的句柄keychord\_fd，keychord\_fd\_init和property\_set\_fd\_init的作用相同；

4.计算超时时间

系统调用poll在监控句柄池时，如果超时时间到了或者有事件发生时，才会返回，如果超时时间被设置为-1时，只有事件发生才会返回。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. if (process\_needs\_restart) {
2. timeout = (process\_needs\_restart - gettime()) \* 1000;
3. if (timeout < 0)
4. timeout = 0;
5. }
7. if (!action\_queue\_empty() || cur\_action)
8. timeout = 0;

如果待执行队列不为空，并且当前Action也不为空，这设置timeout为0，这样poll就不会阻塞，init进程就可以循环执行队列action\_queue中的Action

5.事件监控

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. nr = poll(ufds, fd\_count, timeout);
2. if (nr <= 0)
3. continue;

如果被监控的句柄池中的句柄没有事件发生，但超时时间已到，则返回-1，此时代码不往下执行，而是继续循环执行队列action\_queue中的Action，及重启必要的服务。

6.事件处理

当监控的句柄池中的句柄发生了某些事件时，返回事件发生对应的句柄，从而进入该句柄对应的事件处理函数中。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. for (i = 0; i < fd\_count; i++) {
2. if (ufds[i].revents == POLLIN) {
3. if (ufds[i].fd == get\_property\_set\_fd())
4. handle\_property\_set\_fd();
5. else if (ufds[i].fd == get\_keychord\_fd())
6. handle\_keychord();
7. else if (ufds[i].fd == get\_signal\_fd())
8. handle\_signal();
9. }
10. }

这里有三类事件:

1.属性设置事件；

2.键盘组合事件；

3.子进程死亡信号事件；

对于属性设置事件处理handle\_property\_set\_fd()，请查看[Android 系统属性SystemProperty分析](http://blog.csdn.net/yangwen123/article/details/8936555" \t "_blank)。由于Android系统暂时未使用keychord机制，因此这里不详细介绍。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void handle\_keychord()
2. {
3. struct service \*svc;
4. const char\* debuggable;
5. const char\* adb\_enabled;
6. int ret;
7. \_\_u16 id;
9. // only handle keychords if ro.debuggable is set or adb is enabled.
10. // the logic here is that bugreports should be enabled in userdebug or eng builds
11. // and on user builds for users that are developers.
12. debuggable = property\_get("ro.debuggable");
13. adb\_enabled = property\_get("init.svc.adbd");
14. ret = read(keychord\_fd, &id, sizeof(id));
15. if (ret != sizeof(id)) {
16. ERROR("could not read keychord id\n");
17. return;
18. }
19. //只有在调试模式下才使用
20. if ((debuggable && !strcmp(debuggable, "1")) ||
21. (adb\_enabled && !strcmp(adb\_enabled, "running"))) {
22. svc = service\_find\_by\_keychord(id); //根据keychord\_id查找指定的服务
23. if (svc) {
24. INFO("starting service %s from keychord\n", svc->name); //通过发送组合键消息来启动某些服务
25. service\_start(svc, NULL);
26. } else {
27. ERROR("service for keychord %d not found\n", id);
28. }
29. }
30. }

keychord机制就是为服务配置指定的组合键，可以通过该组合键来启动对应的服务。

当init进程的某个子进程终止时，会对系统的运行产生影响，因此init进程需要重新启动他们。当init的子进程意外终止时，会向父进程 init进程传递SIGCHLD信号，init进程接收到该信号时，预先安装的handler将被调用，将SIGCHLD信号的编号写入socket对的 一端，在socket另一端通过poll系统调用监控到事件的发生，将调用子进程死亡事件处理函数。

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif

当init子进程终止时，init进程会接收到SIGCHLD信号，前面已经介绍了init进程首先安装了信号处理器，因此当接收到SIGCHLD信号时，init进程会调用与该信号相对应的处理函数sigchld\_handler：

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static void sigchld\_handler(int s)
2. {
3. write(signal\_fd, &s, 1);
4. }

参 数s用来接收SIGCHLD信号的编号，该函数仅仅将信号编号写入socket对的一端signal\_fd中，由于signal\_fd与 signal\_recv\_fd是一对已连接的socket，因此当向signal\_fd写入信号编号时，信号编号被传递到接收端 signal\_recv\_fd中，由于signal\_recv\_fd被添加到了监控句柄池中并被注册到了poll系统调用中，因此信号编号的写入将触发 poll函数返回并调用handle\_signal()函数来处理信号事件。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. void handle\_signal(void)
2. {
3. char tmp[32];
4. //读取socket接收端的数据
6. read(signal\_recv\_fd, tmp, sizeof(tmp));
7. while (!wait\_for\_one\_process(0))
8. ;
9. }

该 函数首先从signal\_recv\_fd读取发送过来的信号编号，表示该事件得到处理，避免重复处理该信号事件，然后循环调用 wait\_for\_one\_process函数，直到wait\_for\_one\_process函数返回非0,wait\_for\_one\_process 函数在产生SIGCHLD信号的进程服务列表中，检查进程的设置选项，若选项没有配置oneshot(SVC\_ONE\_SHOT)则设置重启选项 (SVC\_RESTARTING),oneshot选项定义在init.rc文件的service部分中，若进程带有oneshot选项，进程终止时不会 被重启。

**[cpp]** [view plain](http://blog.csdn.net/yangwen123/article/details/9029959" \o "view plain)[copy](http://blog.csdn.net/yangwen123/article/details/9029959" \o "copy)

1. static int wait\_for\_one\_process(int block)
2. {   //block = 0 -->false
3. pid\_t pid;
4. int status;
5. struct service \*svc;
6. struct socketinfo \*si;
7. time\_t now;
8. struct listnode \*node;
9. struct command \*cmd;
11. while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 && errno == EINTR );
12. //正常情况下返回的死亡进程pid大于0，因此wait\_for\_one\_process的返回值正常情况下为0
13. if (pid <= 0) return -1;
14. INFO("waitpid returned pid %d, status = x\n", pid, status);
15. //用于根据pid值在服务链表中查找对应的服务
16. svc = service\_find\_by\_pid(pid);
17. if (!svc) {
18. ERROR("untracked pid %d exited\n", pid);
19. return 0;
20. }
22. NOTICE("process '%s', pid %d exited\n", svc->name, pid);
24. if (!(svc->flags & SVC\_ONESHOT)) {
25. kill(-pid, SIGKILL);
26. NOTICE("process '%s' killing any children in process group\n", svc->name);
27. }

30. for (si = svc->sockets; si; si = si->next) {
31. char tmp[128];
32. snprintf(tmp, sizeof(tmp), ANDROID\_SOCKET\_DIR"/%s", si->name);
33. unlink(tmp);
34. }
35. //设置服务的pid为0 ，并清除SVC\_RUNNING标志
36. svc->pid = 0;
37. svc->flags &= (~SVC\_RUNNING);

40. if (svc->flags & SVC\_ONESHOT) {
41. svc->flags |= SVC\_DISABLED;
42. }

45. if (svc->flags & (SVC\_DISABLED | SVC\_RESET) )  {
46. //设置进程运行状态属性值为stopped
47. notify\_service\_state(svc->name, "stopped");
48. return 0;
49. }
51. now = gettime();
52. //如果死亡的服务进程是系统关键进程，则直接重启手机
53. if (svc->flags & SVC\_CRITICAL) {
54. if (svc->time\_crashed + CRITICAL\_CRASH\_WINDOW >= now) {
55. if (++svc->nr\_crashed > CRITICAL\_CRASH\_THRESHOLD) {
56. ERROR("critical process '%s' exited %d times in %d minutes; "
57. "rebooting into recovery mode\n", svc->name,
58. CRITICAL\_CRASH\_THRESHOLD, CRITICAL\_CRASH\_WINDOW / 60);
59. //手机重启
60. android\_reboot(ANDROID\_RB\_RESTART2, 0, "recovery");
61. return 0;
62. }
63. } else {
64. svc->time\_crashed = now;
65. svc->nr\_crashed = 1;
66. }
67. }
68. //设置服务进程标志SVC\_RESTARTING，在restart\_processes()函数中会重启持有SVC\_RESTARTING
69. svc->flags |= SVC\_RESTARTING;

72. list\_for\_each(node, &svc->onrestart.commands) {
73. cmd = node\_to\_item(node, struct command, clist);
74. cmd->func(cmd->nargs, cmd->args);
75. }
76. //设置进程运行状态属性值为stopped
77. notify\_service\_state(svc->name, "restarting");
78. return 0;
79. }

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif