1. 第一个开机画面的显示过程

Fbmem.c （kernel /drivers/video/fbdev/core/fbmem.c）

1. proc\_create("fb", 0, NULL, &fb\_proc\_fops);   // /proc目录下创建了一个fb文件
2. register\_chrdev(FB\_MAJOR,"fb",&fb\_fops) //来注册了一个名称为fb的字符设备，
3. class\_create(THIS\_MODULE, "graphics"); // sys/class目录下创建了graphics目录
4. EXPORT\_SYMBOL(register\_framebuffer);   // 导出一个函数register\_framebuffer

register\_framebuffer通过调用函数fb\_notifier\_call\_chain来通知帧缓冲区控制台（fbcon），有一个新的帧缓冲区设备被注册到内核中来了。

fbcon.c kernel /drivers/video/console/fbcon.c

1. device\_create(fb\_class, NULL, MKDEV(0, 0), NULL,   "fbcon");//创建一个类别为graphics的设备fbcon
2. fb\_register\_client(&fbcon\_event\_notifier);   //由fbcon\_event\_notify来实现的监听帧缓冲区硬件设备的注册事件

fbcon\_event\_notify

1. ret = fbcon\_fb\_registered(info);  //由fbcon\_event\_notify来实现的监听帧缓冲区硬件设备的注册事件

fbcon\_fb\_registered

1. fbcon\_select\_primary(info);  //检查当前注册的帧缓冲区硬件设备是否是一个主帧缓冲区硬件设备
2. fbcon\_takeover(1) //初始化系统所使用的控制台, 参数为1，表示要显示第一个开机画面。

fbcon\_takeover: 作用就是向系统注册一系列回调函数，以便系统可以通过这些回调函数来操作当前所使用的控制台。这些回调函数使用结构体consw来描述，由fb\_con来指定。

fb\_con

1. .con\_init               = fbcon\_init,

fbcon\_init,

1. fbcon\_prepare\_logo(vc, info, cols, rows, new\_cols, new\_rows);  //调用该函数来准备要显示的第一个开机画面的内容

fbcon\_prepare\_logo调用函数fb\_prepare\_logo来准备的

fb\_prepare\_logo：

1. fb\_get\_color\_depth(&info->var, &info->fix); //得到参数info所描述的帧缓冲区硬件设备的颜色深度depth
2. fb\_logo.logo = fb\_find\_logo(depth); //调用fb\_find\_logo来获得要显示的第一个开机画面的内容，并且保存在全局变量fb\_logo的成员变量logo中

fbcon\_switch

1. .con\_switch             = fbcon\_switch,
2. fb\_show\_logo(info, ops->rotate);  //来显示第一个开机画面

fb\_show\_logo （kernel /drivers/video/fbdev/core/fbmem.c）

1. y = fb\_show\_logo\_line(info, rotate, fb\_logo.logo, 0,    num\_online\_cpus());  // 调用fb\_show\_logo\_line来显示第一个开机画面

fb\_show\_logo\_line     （kernel /drivers/video/fbdev/core/fbmem.c）

1. 参数logo指向了前面所准备的第一个开机画面的内容。这个函数首先根据参数logo的内容来构造一个fb\_image结构体image，用来描述最终要显示的第一个开机画面。
2. 最后就调用函数fb\_do\_show\_logo来真正执行渲染第一个开机画面的操作。

fb\_do\_show\_logo （kernel /drivers/video/fbdev/core/fbmem.c）

1. fb\_do\_show\_logo(struct fb\_info \*info, struct fb\_image \*image,  int rotate, unsigned int num)
2. 参数rotate用来描述屏幕的当前旋转方向
3. 参数info用来描述要渲染的帧缓冲区硬件设备，它的成员变量fbops指向了一系列回调函数，用来操作帧缓冲区硬件设备，其中，回调函数fb\_imageblit就是用来在指定的帧缓冲区硬件设备渲染指定的图像的。

 Android系统的第一个开机画面其实是Linux内核的启动画面。在默认情况下，这个画面是不会出现的。除非我们在编译内核的时候，启用以下两个编译选项：

*CONFIG\_FRAMEBUFFER\_CONSOLE：*表示内核**支持帧缓冲区控制台**

*CONFIG\_LOGO：*表示内核在启动的过程中，需要显示LOGO

一、 帧缓冲区硬件设备在内核中有一个对应的驱动程序模块fbmem（kernel /drivers/video/fbdev/core/fbmem.c），

1. 初始化函数首先调用函数proc\_create在/proc目录下创建了一个fb文件，
2. 接着又调用函数register\_chrdev来注册了一个名称为fb的字符设备，
3. 最后调用函数class\_create在/sys/class目录下创建了一个graphics目录，用来描述内核的图形系统。
4. 还会导出一个函数register\_framebuffer： 在内核的启动过程会被调用，以便用来执行注册帧缓冲区硬件设备的操作

由于系统中可能会存在多个帧缓冲区硬件设备，因此fbmem模块使用一个数组registered\_fb保存所有已经注册了的帧缓冲区硬件设备，其中，每一个帧缓冲区硬件都是使用一个结构体fb\_info来描述的。每一个硬件设备都有一个主设备号FB\_MAJOR（29）和从设备号fb<minor>。例如，第一个被注册的帧缓冲区硬件设备在/dev/graphics目录下都有一个对应的设备文件fb0。用户空间的应用程序通过这个设备文件就可以操作帧缓冲区硬件设备了，即将要显示的画面渲染到帧缓冲区硬件设备上去。

最后，register\_framebuffer通过调用函数fb\_notifier\_call\_chain来通知帧缓冲区控制台（fbcon），有一个新的帧缓冲区设备被注册到内核中来了。

二、帧缓冲区控制台在内核中对应的驱动程序模块为fbcon，它实现在文件kernel /drivers/video/console/fbcon.c中

1、调用函数device\_create来创建一个类别为graphics的设备fbcon之外

2、调用函数fb\_register\_client来监听帧缓冲区硬件设备的注册事件，这由函数fbcon\_event\_notify来实现。

3、 帧缓冲区硬件设备的注册事件最终是由函数fbcon\_fb\_registered来处理

 fbcon\_fb\_registered（）分析：为了简单起见，我们假设系统只有一个帧缓冲区硬件设备，这样当它被注册的时候，全局变量info\_idx的值就会等于-1。当函数fbcon\_fb\_registered在全局数组con2fb\_map\_boot中发现有一个控制台的编号与这个帧缓冲区硬件设备的编号idx对应时，接下来就会调用函数fbcon\_takeover来设置系统所使用的控制台。

fbcon\_takeover的分析：

1. 全局变量logo\_shown的初始值为FBCON\_LOGO\_CANSHOW，表示可以显示第一个开机画面。但是当参数show\_logo的值等于0的时候，不显示开机动画。
2. 函数take\_over\_console用来初始化系统当前所使用的控制台。如果它的返回值不等于0，那么就表示初始化失败。实际上就是向系统注册一系列回调函数，以便系统可以通过这些回调函数来操作当前所使用的控制台。这些回调函数使用结构体consw来描述。这里所注册的结构体consw是由全局变量fb\_con来指定的，

结构体consw中需关注的函数

2.1 fbcon\_init：系统就是通过来初始化，在过程中会决定是否需要准备第一个开机画面的内容

2.1.1、当前正在初始化的控制台使用参数vc来描述，而它的成员变量vc\_num用来描述当前正在初始化的控制台的编号。通过这个编号之后，就可以在全局数组con2fb\_map中找到对应的帧缓冲区硬件设备编号。有了帧缓冲区硬件设备编号之后，就可以在另外一个全局数组中registered\_fb中找到一个fb\_info结构体info，用来描述与当前正在初始化的控制台所对应的帧缓冲区硬件设备。

2.1.2、 变量logo开始的时候被设置为1，表示需要显示第一个开机画面，但是在以下三种情况下，它的值会被设置为0，表示不需要显示开机画面：

        A. 参数vc和变量svc指向的不是同一个vc\_data结构体，即当前正在初始化的控制台不是系统当前可见的控制台。

        B. 全局变量logo\_shown的值等于FBCON\_LOGO\_DONTSHOW，即系统不需要显示第一个开机画面。

        C. 与当前正在初始化的控制台所对应的帧缓冲区硬件设备的显示方式被设置为文本方式，即info->fix.type的值等于FB\_TYPE\_TEXT。

2.1.3  当变量logo= 1时，接下来就会调用函数fbcon\_prepare\_logo来准备要显示的第一个开机画面的内容。函数fbcon\_prepare\_logo中，第一个开机画面的内容是通过调用函数fb\_prepare\_logo（kernel/goldfish/drivers/video/fbmem.c中）来准备的。

2.1.3.1、fbcon\_prepare\_logo函数中，如果要显示的第一个开机画面所占用的控制台行数小于等于参数vc所描述的控制台的最大行数，并且全局变量logo\_show的值不等于FBCON\_LOGO\_DONTSHOW，那么就说明前面所提到的第一个开机画面可以显示在控制台中。这时候全局变量logo\_show的值就会被设置为FBCON\_LOGO\_DRAW，表示第一个开机画面处于等待渲染的状态。

2.1.3.2、fb\_prepare\_logo 这个函数首先得到参数info所描述的帧缓冲区硬件设备的颜色深度depth，接着再调用函数fb\_find\_logo来获得要显示的第一个开机画面的内容，并且保存在全局变量fb\_logo的成员变量logo中。

2.1.3.2.1、fb\_find\_logo（kernel/goldfish/drivers/video/logo/logo.c），

 文件开始声明的一系列linux\_logo结构体变量分别用来保存在kernel/goldfish/drivers/video/logo目录下的一系列ppm或者pbm文件的内容的。这些ppm或者pbm文件都是用来描述第一个开机画面的。

全局变量nologo是一个类型为布尔变量的模块参数，它的默认值等于0，表示要显示第一个开机画面。在这种情况下，函数fb\_find\_logo就会根据参数depth的值以及不同的编译选项来选择第一个开机画面的内容，并且保存在变量logo中返回给调用者。

这一步执行完成之后，第一个开机画面的内容就保存在模块fbmem的全局变量fb\_logo的成员变量logo中了。这时候控制台的初始化过程也结束了，接下来系统就会执行切换控制台的操作。前面提到，当系统执行切换控制台的操作的时候，模块fbcon中的函数fbcon\_switch就会被调用。在调用的过程中，就会执行显示第一个开机画面的操作。

2.2和fbcon\_switch：系统就是通过它来切换控制台，在过程中，会决定是否需要显示第一个开机画面的内容。

2.2.1、由于前面在准备第一个开机画面的内容的时候，全局变量logo\_show的值被设置为FBCON\_LOGO\_DRAW，因此，接下来就会调用函数fb\_show\_logo来显示第一个开机画面。在显示之前，这个函数会将全局变量logo\_shown的值设置为fg\_console，后者表示系统当前可见的控制台的编号。

2.2.1.1、  函数fb\_show\_logo（kernel /drivers/video/fbdev/core/fbmem.c中）

也调用另外一个函数fb\_show\_logo\_line（fbmem.c中）来进一步执行渲染第一个开机画面的操作。

2.2.1.2、函数 fb\_show\_logo\_line分析：参数logo指向了前面所准备的第一个开机画面的内容。这个函数首先根据参数logo的内容来构造一个fb\_image结构体image，用来描述最终要显示的第一个开机画面。最后就调用函数fb\_do\_show\_logo（fbmem.c中）来真正执行渲染第一个开机画面的操作。

2.2.1.3、函数fb\_do\_show\_logo分析：

1. 参数rotate用来描述屏幕的当前旋转方向。屏幕旋转方向不同，第一个开机画面的渲染方式也有所不同。

2. 参数info用来描述要渲染的帧缓冲区硬件设备，它的成员变量fbops指向了一系列回调函数，用来操作帧缓冲区硬件设备，其中，回调函数fb\_imageblit就是用来在指定的帧缓冲区硬件设备渲染指定的图像的。

       至此，第一个开机画面的显示过程就分析完成了。

1. 第二个开机画面的显示过程

Init->main->queue\_builtin\_action->console\_init\_action->load\_565rle\_image

load\_565rle\_image

1. vt\_set\_mode // 显示方式设置为图形方式
2. fd = open(fn, O\_RDONLY)  //来打开设备文件O\_RDONLY= /initlogo.rle
3. fstat(fd, &s) //来获得initlogo.rle的大小
4. mmap(0, s.st\_size, PROT\_READ, MAP\_SHARED, fd, 0);  //把文件/initlogo.rle映射到init进程的地址空间来了，以便可以读取它的内容
5. fb\_open(&fb) // 打开设备文件/dev/graphics/fb0
6. while循环 //得到了文件initlogo.rle和帧缓冲区硬件设备在init进程中的虚拟访问地址以及大小后，将文件initlogo.rle的内容写入到帧缓冲区硬件设备中去，以便可以将第二个开机画面显示出来
7. munmap(data, s.st\_size);  //将从文件initlogo.rle中读取出来的颜色值写入到帧缓冲区硬件设备中去，
8. fb\_update(&fb);  //更新屏幕上的第二个开机画面
9. fb\_close(&fb);  //注销文件/dev/graphics/fb0在init进程中的映射以及关闭文件/dev/graphics/fb0
10. close(fd);   //来关闭文件/initlogo.rle
11. unlink(fn); //删除目标设备上的/initlogo.rle文件，不是删除ramdisk映像中的 initlogo.rle文件

2.1、第二个开机画面是在init（system/core/init/init.c）进程启动的过程中显示的，因此，我们就从init进程的入口函数main开始分析第二个开机画面的显示过程

Main函数分析：函数一开始就首先判断参数argv[0]的值是否等于“ueventd”，如果是的话，那么就以ueventd\_main（system/core/init/ueventd.c）函数来作入口函数。可执行文件/sbin/ueventd是可执行文件/init的一个符号链接文件，即应用程序ueventd和init运行的是同一个可执行文件。内核启动完成之后，即init进程会首先被启动。init进程在启动的过程中，会对启动脚本/init.rc进行解析。在启动脚本/init.rc中，配置了一个ueventd进程，它对应的可执行文件为/sbin/ueventd，即ueventd进程加载的可执行文件也为/init。因此，通过判断参数argv[0]的值，就可以知道当前正在启动的是init进程还是ueventd进程。

ueventd进程是作什么用的呢？用来处理uevent事件的，即用来管理系统设备的。ueventd进程会通过一个socket接口来和内核通信，以便可以监控系统设备事件。例如：我们调用device\_create函数来创建了名为 “hello”的字符设备，这时候内核就会向前面提到的socket发送一个设备增加事件。ueventd进程通过这个socket获得了这个设备增加事件之后，就会/dev目录下创建一个名称为“hello”的设备文件。这样用户空间的应用程序就可以通过设备文件/dev/hello来和驱动程序hello进行通信了。

2.2、main调用另外一个函数queue\_builtin\_action(实现在system/core/init/init\_parser.c中)来向init进程中的一个待执行action队列增加了一个名称等于“console\_init”的action。这个action对应的执行函数为console\_init\_action，它就是用来显示第二个开机画面的。

函数queue\_builtin\_action分析：

action\_list列表用来保存从启动脚本/init.rc解析得到的一系列action，以及一系列内建的action。当这些action需要执行的时候，它们就会被添加到action\_queue列表中去，以便init进程可以执行它们。

2.3、回到init进程的入口函数main中，最后init进程会进入到一个无限循环中去。在这个无限循环中，init进程会做以下五个事情：

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。

注意，由于后面三个事件都是可以通过文件描述符来描述的，因此，init进程的入口函数main使用poll机制来同时轮询它们，以便可以提高效率。

2.2.1、函数console\_init\_action的分析：这个函数主要做了两件事件

1. 初始化控制台。init进程在启动的时候，会解析内核的启动参数（保存在文件/proc/cmdline中）。如果发现内核的启动参数中包含有了一个名称为“androidboot.console”的属性，那么就会将这个属性的值保存在字符数组console中。这样我们就可以通过设备文件/dev/<console>来访问系统的控制台。如果内核的启动参数没有包含名称为“androidboot.console”的属性，那么默认就通过设备文件/dev/console来访问系统的控制台。如果能够成功地打开设备文件/dev/<console>或者/dev/console，那么就说明系统支持访问控制台，因此，全局变量have\_console的就会被设置为1。

2. 显示第二个开机画面。通过调用函数load\_565rle\_image（实现在文件system/core/init/logo.c中）来实现的。在调用函数load\_565rle\_image的时候，指定的开机画面文件为INIT\_IMAGE\_FILE（是一个宏，定义在system/core/init/init.h中）

#define INIT\_IMAGE\_FILE "/initlogo.rle"

即第二个开机画面的内容是由文件/initlogo.rle来指定的。如果文件/initlogo.rle不存在，或者在显示它的过程中出现异常，那么函数load\_565rle\_image的返回值就会等于-1，这时候函数console\_init\_action就以文本的方式来显示第二个开机画面，即向编号为0的控制台（/dev/tty0）输出“ANDROID”这7个字符。

2.2.1.2.1、load\_565rle\_image分析

1.   函数首先调用函数vt\_set\_mode来将控制台的显示方式设置为图形方式。vt\_set\_mode分析：首先打开控制台设备文件/dev/tty0，接着再通过IO控制命令KDSETMODE来将控制台的显示方式设置为文本方式或者图形方式，取决于参数graphics的值。从前面的调用过程可以知道，参数graphics的值等于1，因此，这里是将控制台的显示方式设备为图形方式。

2. 接着调用函数open打开这个文件，并且将获得的文件描述符保存在变量fd中，接着再调用函数fstat来获得这个文件的大小。有了这些信息之后，函数load\_565rle\_image就可以调用函数mmap来把文件/initlogo.rle映射到init进程的地址空间来了，以便可以读取它的内容。

3、将文件/initlogo.rle映射到init进程的地址空间之后，接下来再调用函数fb\_open来打开设备文件/dev/graphics/fb0。前面在介绍第一个开机画面的显示过程中提到，设备文件/dev/graphics/fb0是用来访问系统的帧缓冲区硬件设备的，因此，打开了设备文件/dev/graphics/fb0之后，我们就可以将文件/initlogo.rle的内容输出到帧缓冲区硬件设备中去了。

3.1、fb\_open分析：

3.1.1、  打开了设备文件/dev/graphics/fb0之后，

接着再分别通过IO控制命令FBIOGET\_FSCREENINFO和FBIOGET\_VSCREENINFO来获得帧缓冲硬件设备的固定信息和可变信息：

1. 固定信息使用一个fb\_fix\_screeninfo结构体来描述，它保存的是帧缓冲区硬件设备固有的特性，这些特性在帧缓冲区硬件设备被初始化了之后，就不会发生改变，例如屏幕大小以及物理地址等信息。
2. 可变信息使用一个fb\_var\_screeninfo结构体来描述，它保存的是帧缓冲区硬件设备可变的特性，这些特性在系统运行的期间是可以改变的，例如屏幕所使用的分辨率、颜色深度以及颜色格式等。

3.1.2、fb\_open还会将设备文件/dev/graphics/fb0的内容映射到init进程的地址空间来，这样init进程就可以通过映射得到的虚拟地址来访问帧缓冲区硬件设备的内容了。

4、load\_565rle\_image接下来分别使用宏fb\_width和fb\_height来获得屏幕所使用的的分辨率，即屏幕的宽度和高度。宏fb\_width和fb\_height的定义如下所示：

#define fb\_width(fb) ((fb)->vi.xres)

#define fb\_height(fb) ((fb)->vi.yres)

现在我们分别得到了文件initlogo.rle和帧缓冲区硬件设备在init进程中的虚拟访问地址以及大小，这样我们就可以将文件initlogo.rle的内容写入到帧缓冲区硬件设备中去，以便可以将第二个开机画面显示出来，这是通过函数load\_565rle\_image中的while循环来实现的。

5、load\_565rle\_image通过调用函数android\_memset16来将从文件initlogo.rle中读取出来的颜色值写入到帧缓冲区硬件设备中去。

6、将文件/initlogo.rle的内容写入到帧缓冲区硬件设备去之后，第二个开机画面就可以显示出来了。接下来函数load\_565rle\_image就会调用函数munmap来注销文件/initlogo.rle在init进程中的映射，并且调用函数close来关闭文件/initlogo.rle。关闭了文件/initlogo.rle之后，还会调用函数unlink来删除目标设备上的/initlogo.rle文件。注意，这只是删除了目标设备上的/initlogo.rle文件，而不是删除ramdisk映像中的initlogo.rle文件，因此，每次关机启动之后，系统都会重新将ramdisk映像中的initlogo.rle文件安装到目标设备上的根目录来，这样就可以在每次开机的时候都能将它显示出来。

7、调用fb\_close函数来注销文件/dev/graphics/fb0在init进程中的映射以及关闭文件/dev/graphics/fb0

8、在调用fb\_close函数之前，函数load\_565rle\_image还会调用另外一个函数fb\_update来更新屏幕上的第二个开机画面，

在结构体fb\_var\_screeninfo中，除了使用成员变量xres和yres来描述屏幕所使用的分辨率之外，还使用成员变量xres\_virtual和yres\_virtual来描述屏幕所使用的虚拟分辨率(大于可视分辨率xres和yres)。

 帧缓冲区的大小是由虚拟分辨率决定的，我们就可以在帧缓冲中写入比屏幕大小还要多的像素值，多出来的这个部分像素值就可以用作双缓冲

1. 第三个开机画面的显示过程

第三个开机画面是由应用程序bootanimation来负责显示的。

1、init进程在启动的时候，不会主动将应用程序bootanimation启动起来。

2、当SurfaceFlinger服务启动的时候，它会通过修改系统属性ctl.start的值来通知init进程启动应用程序bootanimation，以便可以显示第三个开机画面，

3、而当System进程将系统中的关键服务都启动起来之后，ActivityManagerService服务就会通知SurfaceFlinger服务来修改系统属性ctl.stop的值，以便可以通知init进程停止执行应用程序bootanimation，即停止显示第三个开机画面

3.1第三个开机画面的显示过程

总体过程：

内核->init --> ServiceManager->Zygote进程->System进程->SurfaceFlinger::函数instantiate->启动SurfaceFlinger服务

->创建一个SurfaceFlinger实例->调用SurfaceFlinger::init()初始化->注册到Service Manager中-> SurfaceFlinger::run()调用过程中启动第三个开机画面。

与之前不同，在android4.4中systemserver进程由init直接根据init.rc配置启动了surfaceflinger进程。

init.rc

1. service surfaceflinger /system/bin/surfaceflinger
2. **class** main
3. user system
4. group graphics drmrpc
5. onrestart restart zygote

surfaceflinger进程的main函数，new一个SurfaceFlinger实例，跟着调用SurfaceFlinger类的init()函数完成后面工作，然后向service manager注册服务

frameworks/native/services/surfaceflinger/main\_surfaceflinger.cpp

1. **int** main(**int** argc, **char**\*\* argv) {
2. ....
4. // instantiate surfaceflinger
5. sp<SurfaceFlinger> flinger = **new** SurfaceFlinger();//新实例
7. ...
9. // initialize before clients can connect
10. flinger->init();//调用surfaceflinger的初始化函数
12. // publish surface flinger
13. sp<IServiceManager> sm(defaultServiceManager());
14. sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, **false**);//向service manager注册服务
16. // run in this thread
17. flinger->run();//开跑
19. **return** 0;
20. }

这个init()函数与4.3系统的ReadyToRun()函数的功能是一样的，内容上有些不同，但大体也分成三个部分，第一硬件初始化，第二建立消息线程，第三启动开机动画。

1. **void** SurfaceFlinger::init() {
2. ALOGI(  "SurfaceFlinger's main thread ready to run. "
3. "Initializing graphics H/W...");

6. .....
8. // initialize our non-virtual displays
9. **for** (**size\_t** i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {
10. DisplayDevice::DisplayType type((DisplayDevice::DisplayType)i);
11. // set-up the displays that are already connected
12. **if** (mHwc->isConnected(i) || type==DisplayDevice::DISPLAY\_PRIMARY) {
13. // All non-virtual displays are currently considered secure.
14. **bool** isSecure = **true**;
15. createBuiltinDisplayLocked(type);
16. wp<IBinder> token = mBuiltinDisplays[i];
18. sp<BufferQueue> bq = **new** BufferQueue(**new** GraphicBufferAlloc());
19. sp<FramebufferSurface> fbs = **new** FramebufferSurface(\*mHwc, i, bq);
20. sp<DisplayDevice> hw = **new** DisplayDevice(**this**,
21. type, allocateHwcDisplayId(type), isSecure, token,
22. fbs, bq,
23. mEGLConfig);
24. **if** (i > DisplayDevice::DISPLAY\_PRIMARY) {
25. // FIXME: currently we don't get blank/unblank requests
26. // for displays other than the main display, so we always
27. // assume a connected display is unblanked.
28. ALOGD("marking display %d as acquired/unblanked", i);
29. hw->acquireScreen();
30. }
31. mDisplays.add(token, hw);
32. }
33. }
35. ...
36. // start the EventThread
37. ...
38. mSFEventThread = **new** EventThread(sfVsyncSrc);
39. mEventQueue.setEventThread(mSFEventThread);
41. mEventControlThread = **new** EventControlThread(**this**);
42. mEventControlThread->run("EventControl", PRIORITY\_URGENT\_DISPLAY);
44. ...
46. // set initial conditions (e.g. unblank default device)
47. initializeDisplays();
49. // start boot animation
50. startBootAnim();
51. }

最后调用run()函数，

1. **void** SurfaceFlinger::run() {
2. **do** {
3. waitForEvent();
4. } **while** (**true**);
5. }

surfaceflinger 启动后

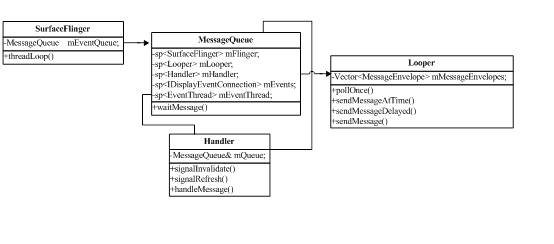
其中waitForEvent()是SurfaceFlinger中的成员函数，它进一步调用mEventQueue.waitMessage()

1. void SurfaceFlinger::waitForEvent() {
2. mEventQueue.waitMessage();
3. }

waitMessage()

1. void MessageQueue::waitMessage() {
2. do {
3. IPCThreadState::self()->flushCommands();
4. int32\_t ret =mLooper->pollOnce(-1);
5. switch (ret) {
6. caseALOOPER\_POLL\_WAKE:
7. caseALOOPER\_POLL\_CALLBACK:
8. continue;
9. caseALOOPER\_POLL\_ERROR:
10. ALOGE("ALOOPER\_POLL\_ERROR");
11. caseALOOPER\_POLL\_TIMEOUT:
12. // timeout(should not happen)
13. continue;
14. default:
15. // should nothappen
16. ALOGE("Looper::pollOnce() returned unknown status %d", ret);
17. continue;
18. }
19. } while (true);
20. }

下面这句将在内部调用MessageQueue::mHandler来处理消息：



mLooper->pollOnce(-1);pollOnce函数使用了一个死循环，它不断地取消息进行处理，

/\*frameworks/native/services/surfaceflinger/MessageQueue.cpp\*/

1. void MessageQueue::Handler::handleMessage(const Message&message) {
2. switch (message.what) {
3. case INVALIDATE:
4. android\_atomic\_and(~eventMaskInvalidate, &mEventMask);
5. mQueue.mFlinger->onMessageReceived(message.what);
6. break;
7. case REFRESH:
8. android\_atomic\_and(~eventMaskRefresh, &mEventMask);
9. mQueue.mFlinger->onMessageReceived(message.what);
10. break;
11. }
12. }

如上述代码，mHandler当收到INVALIDATE和REFRESH请求时，进一步回调了SurfaceFlinger中的onMessageReceived。

1. void SurfaceFlinger::onMessageReceived(int32\_t what) {
2. ATRACE\_CALL();
3. switch (what) {
4. case MessageQueue::TRANSACTION: {
5. handleMessageTransaction();
6. break;
7. }
8. case MessageQueue::INVALIDATE: {
9. bool refreshNeeded = handleMessageTransaction();
10. refreshNeeded |= handleMessageInvalidate();
11. refreshNeeded |= mRepaintEverything;
12. if (refreshNeeded) {
13. // Signal a refresh if a transaction modified the window state,
14. // a new buffer was latched, or if HWC has requested a full
15. // repaint
16. signalRefresh();
17. }
18. break;
19. }
20. case MessageQueue::REFRESH: {
21. handleMessageRefresh();
22. break;
23. }
24. }

```

根据情况调用handleMessageRefresh()

1. void SurfaceFlinger::handleMessageRefresh() {
2. ATRACE\_CALL();
3. preComposition();
4. rebuildLayerStacks();
5. setUpHWComposer();
6. doDebugFlashRegions();
7. doComposition();
8. postComposition();
9. }