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

```
5.  device_create(fb_class, NULL, MKDEV(0, 0), NULL,   "fbcon");//创建一个类别为 graphics 的设备 fbcon
6. 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

```
7.  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

```
8.  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
9.  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

```
10. .con_init                = fbcon_init,
```

fbcon_init,

```
11. 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：

```
12.  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
13. 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

```
14. .con_switch              = fbcon_switch,
```

```
15. 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）

```
16. 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）

```
17. 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。
18. 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）

```
19. fb_do_show_logo(struct fb_info *info, struct fb_image *image,  int rotate, unsigned int num)
20.  参数 rotate 用来描述屏幕的当前旋转方向
21.  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。

2. 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：

一、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。

二、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

3. 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.  ....
3.
4.      // instantiate surfaceflinger
5.      sp<SurfaceFlinger> flinger = new SurfaceFlinger();//新实例
6.
7.  ...
8.
9.      // initialize before clients can connect
10.     flinger->init();//调用 surfaceflinger 的初始化函数
11.
```

```
12.     // publish surface flinger
13.     sp<IServiceManager> sm(defaultServiceManager());
14.     sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, fals
    e);//向 service manager 注册服务
15.
16.     // run in this thread
17.     flinger->run();//开跑
18.
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...");
4.
5.
6.  .....
7.
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];
17.
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);
```

```
                hw->acquireScreen();
            }
            mDisplays.add(token, hw);
        }
    }

...
    // start the EventThread
    ...
    mSFEventThread = new EventThread(sfVsyncSrc);
    mEventQueue.setEventThread(mSFEventThread);

    mEventControlThread = new EventControlThread(this);
    mEventControlThread->run("EventControl", PRIORITY_URGENT_DISPLAY);

...

    // set initial conditions (e.g. unblank default device)
    initializeDisplays();

    // start boot animation
    startBootAnim();
}
```

最后调用 run()函数，

```
void SurfaceFlinger::run() {
    do {
        waitForEvent();
    } while (true);
}
```

surfaceflinger 启动后

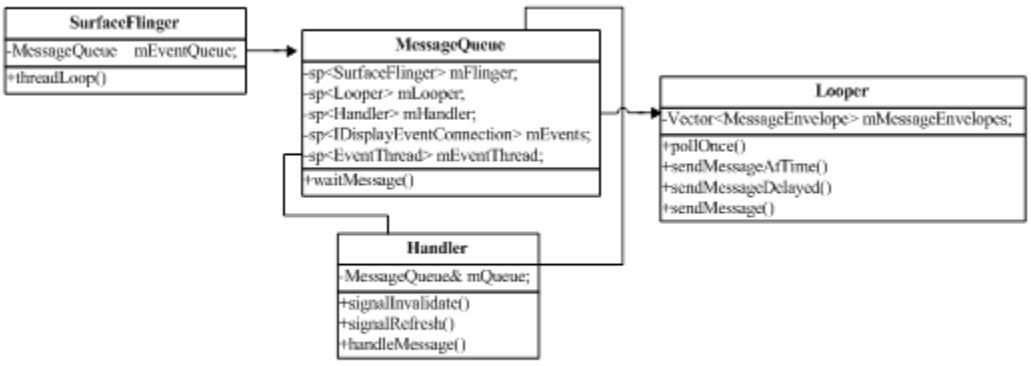
其中 waitForEvent()是 SurfaceFlinger 中的成员函数，它进一步调用 mEventQueue.waitMessage()

```
void SurfaceFlinger::waitForEvent() {
    mEventQueue.waitMessage();
}
```

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.     }
```

```
25.
```

```
```

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.  }
10.
```