# Linux根文件系统的加载

## VFS的注册vfs\_caches\_init()

asmlinkage void \_\_init **start\_kernel**(void)

{

vfs\_caches\_init\_early();

vfs\_caches\_init(totalram\_pages); //VFS的注册: fs/dcache.c

...

rest\_init();

}

void \_\_init **vfs\_caches\_init**(unsigned long mempages)

{

...

dcache\_init();

inode\_init();

files\_init(mempages);

mnt\_init(); //创建一个虚拟的rootfs，即内存文件系统，后面还会指向真实的文件系统: fs/namespace.c

bdev\_cache\_init();

chrdev\_init();

}

void \_\_init **mnt\_init**(void)

{

err = sysfs\_init();

fs\_kobj = kobject\_create\_and\_add("fs", NULL);

init\_rootfs(); //创建虚拟根文件系统；

init\_mount\_tree(); //注册根文件系统。

}

static void \_\_init **init\_mount\_tree**(void)

{

struct vfsmount \*mnt;

struct mnt\_namespace \*ns;

struct path root;

mnt = do\_kern\_mount("rootfs", 0, "rootfs", NULL); //创建虚拟文件系统

ns = create\_mnt\_ns(mnt);

init\_task.nsproxy->mnt\_ns = ns;

get\_mnt\_ns(ns);

root.mnt = mnt;

root.dentry = mnt->mnt\_root;

set\_fs\_pwd(current->fs, &root);

set\_fs\_root(current->fs, &root); //将当前的文件系统配置为根文件系统。

}

## VFS的挂载rest\_init()--kernel\_init()

asmlinkage void \_\_init **start\_kernel**(void)

{

vfs\_caches\_init\_early();

vfs\_caches\_init(totalram\_pages); //VFS的注册: fs/dcache.c

...

rest\_init();

}

static noinline void \_\_init\_refok **rest\_init**(void)

{

kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND);

pid = kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES);

complete(&kthreadd\_done);

init\_idle\_bootup\_task(current);

schedule\_preempt\_disabled();

cpu\_idle();

}

static int \_\_init **kernel\_init**(void \* unused)

{

wait\_for\_completion(&kthreadd\_done);

//调用所有模块的初始化函数，包括initramfs的初始化函数populate\_rootfs

do\_basic\_setup();

//ramdisk\_execute\_command值通过“rdinit=”指定，如果未指定，则采用默认的值/init。

if (!ramdisk\_execute\_command)

ramdisk\_execute\_command = "/init";

//检查根文件系统中是否存在文件ramdisk\_execute\_command，如果存在的话则执行init\_post()，否则执行prepare\_namespace()挂载根文件系统。

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0) {

ramdisk\_execute\_command = NULL;

prepare\_namespace();

}

init\_post();

}

static void \_\_init **do\_basic\_setup**(void)

{

cpuset\_init\_smp();

usermodehelper\_init();

shmem\_init();

driver\_init();

init\_irq\_proc();

do\_ctors();

usermodehelper\_enable();

do\_initcalls();

}

//函数do\_basic\_setup()调用所有模块的初始化函数，包括initramfs的初始化函数populate\_rootfs。这部分代码在init/initramfs.c下面

//需要特别指出的是initramfs.c模块的入口函数populate\_rootfs()是否执行取决于Kernel的编译选项。参照linux/init目录下的makefile文件

rootfs\_initcall(populate\_rootfs);

**CONFIG\_BLK\_DEV\_INITRD=y**

**CONFIG\_GENERIC\_CALIBRATE\_DELAY=y**

**CONFIG\_BLK\_DEV\_RAM=y**

**# CONFIG\_BLK\_DEV\_MD is not set**

=========================================

# Makefile for the linux kernel.

obj-y := main.o version.o mounts.o

ifneq ($(CONFIG\_BLK\_DEV\_INITRD),y)

obj-y += noinitramfs.o

else

obj-$(CONFIG\_BLK\_DEV\_INITRD) += initramfs.o

endif

obj-$(CONFIG\_GENERIC\_CALIBRATE\_DELAY) += calibrate.o

mounts-y := do\_mounts.o

mounts-$(CONFIG\_BLK\_DEV\_RAM) += do\_mounts\_rd.o

mounts-$(CONFIG\_BLK\_DEV\_INITRD) += do\_mounts\_initrd.o

mounts-$(CONFIG\_BLK\_DEV\_MD) += do\_mounts\_md.o

static int \_\_init **populate\_rootfs**(void)

{

**//解压三种格式的根文件系统**

* //如果\_\_initramfs\_size不为0，就说明这是**和内核文件集成在一起的cpio的intrd**

char \*err = unpack\_to\_rootfs(\_\_initramfs\_start, \_\_initramfs\_size);

//如果initrd\_start不为0，说明这是**由bootloader加载的initrd**，那么需要进一步判断是**cpio格式的initrd**，还是**老式块设备的initrd**

if (initrd\_start) {

#ifdef CONFIG\_BLK\_DEV\_RAM

* //首先判断是不是cpio格式的initrd，也就是这里说的initramfs, 是，则解压

err = unpack\_to\_rootfs((char \*)initrd\_start, initrd\_end - initrd\_start);

if (!err) {

free\_initrd();

return 0;

}

* //如果执行到这里，说明这是旧的块设备格式的initrd。

else {

clean\_rootfs();

unpack\_to\_rootfs(\_\_initramfs\_start, \_\_initramfs\_size);

}

//如果不是CPIO-Initrd，则就是Image-Initrd，将其内容保存到文件/initrd.image中。在根文件系统中创建文件/initrd.image

fd = sys\_open((const char \_\_user \_\_force \*) "/initrd.image", O\_WRONLY|O\_CREAT, 0700);

if (fd >= 0) {

//这里是对Image-Initrd提供支持的，将内存中的initrd赋值到initrd.image中，以释放内存空间

sys\_write(fd, (char \*)initrd\_start, initrd\_end - initrd\_start);

sys\_close(fd);

free\_initrd(); ////释放Initrd所占用的内存空间。

}

#else

err = unpack\_to\_rootfs((char \*)initrd\_start, initrd\_end - initrd\_start);

free\_initrd();

#endif

}

}

### 根文件系统的挂载kernel\_init()--prepare\_namespace()

static int \_\_init **kernel\_init**(void \* unused)

{

wait\_for\_completion(&kthreadd\_done);

//调用所有模块的初始化函数，包括initramfs的初始化函数populate\_rootfs

do\_basic\_setup();

//ramdisk\_execute\_command值通过“rdinit=”指定，如果未指定，则采用默认的值/init。

if (!ramdisk\_execute\_command)

ramdisk\_execute\_command = "/init";

//检查根文件系统中是否存在文件ramdisk\_execute\_command，如果存在的话则执行init\_post()，**否则执行prepare\_namespace()挂载根文件系统**。

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0) {

ramdisk\_execute\_command = NULL;

**prepare\_namespace();** //**挂载根文件系统:** init/do\_mounts.c

}

init\_post();

}

从上面的VFS代码分析中知道，对于Image-Initrd或者VFS（即InitRamfs或者CPIO-Initrd）中不存在文件ramdisk\_execute\_command的情况，则执行prepare\_namespace()。

void \_\_init **prepare\_namespace**(void)

{

//对于将根文件系统存放到USB或者SCSI设备上的情况，Kernel需要等待这些耗费时间比较久的设备驱动加载完毕，所以这里存在一个Delay

if (root\_delay) ssleep(root\_delay);

//等待根文件系统所在的设备探测函数的完成

wait\_for\_device\_probe();

//saved\_root\_name存放的是Kernel参数root=所指定的设备文件

if (saved\_root\_name[0]) {

root\_device\_name = saved\_root\_name;

if (!strncmp(root\_device\_name, "mtd", 3) || !strncmp(root\_device\_name, "ubi", 3)) {

//将saved\_root\_nam指定的设备进行加载。

//如CONFIG\_CMDLINE="console=ttyS0,115200 mem=108M rdinit=/linuxrc root=/dev/mtdblock2", 实际上就是加载/dev/mtdblock2。

mount\_block\_root(root\_device\_name, root\_mountflags);

goto out;

}

//参数ROOT\_DEV存放设备节点号。

ROOT\_DEV = name\_to\_dev\_t(root\_device\_name);

}

//挂载initrd

if (initrd\_load()) goto out; //init/do\_mounts\_initrd.c

//如果指定mount\_initrd为true，即没有指定在函数initrd\_load中mount的话，则在这里重新realfs的mount操作

mount\_root();

out:

devtmpfs\_mount("dev");

//将挂载点从当前目录（实际当前的目录在mount\_root中或者在mount\_block\_root中指定）移到根目录。对于上面的command line的话，当前的目录就是/dev/mtdblock2

sys\_mount(".", "/", NULL, MS\_MOVE, NULL);

//将当前目录当作系统的根目录，至此虚拟系统根目录文件系统切换到了实际的根目录文件系统

sys\_chroot((const char \_\_user \_\_force \*)".");

}

int \_\_init **initrd\_load**(void)

{

//可以通过Kernel的参数“noinitrd“来配置mount\_initrd的值，默认为1

if (mount\_initrd) {

//创建一个Root\_RAM0的设备节点/dev/ram

create\_dev("/dev/ram", Root\_RAM0);

//如果根文件设备号不是Root\_RAM0则程序就会执行以下代码，例如上面指定的/dev/mtdblock2设备节点肯定就不是Root\_RAM0

if (rd\_load\_image("/initrd.image") && ROOT\_DEV != Root\_RAM0) {

sys\_unlink("/initrd.image");

//执行Initrd中的linuxrc文件，并且将realfs的根目录设置为当前目录。这些操作只对image-cpio的情况下才会去执行

handle\_initrd();

return 1;

}

}

sys\_unlink("/initrd.image");

return 0;

}

static void \_\_init **handle\_initrd**(void)

{

real\_root\_dev = new\_encode\_dev(ROOT\_DEV); //全局变量，用来保存realfs的设备号

create\_dev("/dev/root.old", Root\_RAM0);

//将realfs加载到VFS的/root下

mount\_block\_root("/dev/root.old", root\_mountflags & ~MS\_RDONLY);

sys\_mkdir("/old", 0700);

root\_fd = sys\_open("/", 0, 0); old\_fd = sys\_open("/old", 0, 0);

//切换到initrd文件系统，前面已经挂载initrd到VFS的root目录下

//将/root目录挂在为/根目录

sys\_chdir("/root");sys\_mount(".", "/", NULL, MS\_MOVE, NULL); sys\_chroot(".");

//执行/根目录中的linuxrc，并等待执行结束

pid = **kernel\_thread(do\_linuxrc, "/linuxrc", SIGCHLD);**

sys\_fchdir(old\_fd);sys\_mount("/", ".", NULL, MS\_MOVE, NULL);

sys\_fchdir(root\_fd);sys\_chroot("."); //将/根目录挂在为"."当前目录

sys\_close(old\_fd);sys\_close(root\_fd);

//如果real\_root\_dev直接配置为Root\_RAM0，也即直接使用直接使用initrd作为realfs，改变当前目录到initrd中，并直接返回

if (new\_decode\_dev(real\_root\_dev) == Root\_RAM0) {

sys\_chdir("/old"); return;

}

ROOT\_DEV = new\_decode\_dev(real\_root\_dev);

//将realfs挂载到VFS的/root目录下，并将当前的目录配置为VFS的/root

mount\_root();

//将/old目录挂在为"/root/initrd"目录

error = sys\_mount("/old", "/root/initrd", NULL, MS\_MOVE, NULL);

if (!error)

printk("okay\n");

else {

int fd = sys\_open("/dev/root.old", O\_RDWR, 0);

sys\_umount("/old", MNT\_DETACH);

if (fd < 0) { error = fd;} else { error = sys\_ioctl(fd, BLKFLSBUF, 0); sys\_close(fd); }

}

}

### 真实根文件系统挂载后的操作kernel\_init()--init\_post()

在该函数的最后，以此会去搜索文件并执行ramdisk\_execute\_command、execute\_command、/sbin/init、/etc/init、/bin/init和/bin/sh，如果发现这些文件均不存在的话，则通过panic输出错误命令，并将当前的系统Halt在那里

static int \_\_init **kernel\_init**(void \* unused)

{

wait\_for\_completion(&kthreadd\_done);

//调用所有模块的初始化函数，包括initramfs的初始化函数populate\_rootfs

do\_basic\_setup();

//ramdisk\_execute\_command值通过“rdinit=”指定，如果未指定，则采用默认的值/init。

if (!ramdisk\_execute\_command)

ramdisk\_execute\_command = "/init";

//检查根文件系统中是否存在文件ramdisk\_execute\_command，如果存在的话则执行init\_post()，**否则执行prepare\_namespace()挂载根文件系统**。

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0) {

ramdisk\_execute\_command = NULL;

**prepare\_namespace();** //**挂载根文件系统:** init/do\_mounts.c

}

**init\_post();**

}

static noinline int **init\_post**(void)

{

if (ramdisk\_execute\_command) {

run\_init\_process(ramdisk\_execute\_command); //"/init"

}

if (execute\_command) {

run\_init\_process(execute\_command);

}

run\_init\_process("/sbin/init");

run\_init\_process("/etc/init");

run\_init\_process("/bin/init");

run\_init\_process("/bin/sh");

panic("No init found. Try passing init= option to kernel. "

"See Linux Documentation/init.txt for guidance.");

}

# Initramfs文件系统

## initrd与initramfs的区别

### initrd与initramfs产生的背景

boot loader装入kernel, 然后kernel需要执行/sbin/init, 读取这个文件就必须先mount根文件系统, 早期是通过启动时的**“root=”** 参数告诉内核根文件系统在哪个设备上, 随着硬件和技术的发展,现在根文件系统可能位于一个网络存储如NFS上, 可能由于RAID而散布于多个设备上, 可能位于一个加密设备上需要提供用户名和密码,这时**root=**参数就显得不够了. 为了应付这种局面, 先后出现两种机制来作为boot loader装载kernel到真正的/sbin/init执行这个启动过程的桥梁: initrd和initramfs。

initrd和initramfs两者有类似的地方, 比如都是由内核执行其上的某个程序(**initrd是/linuxrc, initramfs是/init**),由这个程序决定加载什么驱动以及如何装载根文件系统. 下面一点笔记总结

### initrd和initramfs的优缺点.

#### initrd:

ramdisk (如 initrd) 是 基于ram的块设备，ram disk是一个基于ram的块设备，因此它占据了一块固定的内存，而且事先要使用特定的工具比如mke2fs格式化，还需要一个文件系统驱动来读写其上的文件。如果这个disk上的空间没有用完，这些未用的内存就浪费掉了，并且这个disk的空间固定导致容量有限，要想装入更多的文件就需要重新格式化。由于Linux的块设备缓冲特性, ram disk上的数据被拷贝到page cache(对于文件数据)和dentry cache(对于目录项), 这个也导致内存浪费.

#### initramfs:

最初的想法是Linus提出的: 把cache当作文件系统装载. 他在一个叫ramfs的cache实现上加了一层很薄的封装, 其它内核开发人员编写了一个改进版tmpfs, 这个文件系统上的数据可以写出到交换分区, 而且可以设定一个tmpfs装载点的最大尺寸以免耗尽内存. initramfs就是tmpfs的一个应用.

**优点:**

(1)tmpfs随着其中数据的增减自动增减容量.

(2)在tmpfs和page cache/dentry cache之间没有重复数据.

(3)tmpfs重复利用了Linux caching的代码, 因此几乎没有增加内核 尺寸, 而caching的代码已经经过良好测试, 所以tmpfs的代码质量也有保证.

(4)不需要额外的文件系统驱动.

另外, initrd机制被设计为旧的"root="机制的前端, 而非其替代物,它假设真正的根设备是一个块设备, 而且也假设了自己不是真正的根设备,这样不便将NFS等作为根文件系统, 最后/linuxrc不是以PID=1执行的, 因为1这个进程ID是给/sbin/init保留的. initrd机制找到真正的根设备后将其设备号写入/proc/sys/kernel/real-root-dev, 然后控制转移到内核由其装载根文件系统并启动/sbin/init.

initramfs则去掉了上述假设, 而且/init以PID=1执行, 由init装载根文件系统并用exec转到真正的/sbin/init, 这样也导致一个更为干净漂亮的设计.