**Linux2.6 内核的 Initrd 机制解析**

Linux 的 initrd 技术是一个非常普遍使用的机制，linux2.6 内核的 initrd 的文件格式由原来的文件系统镜像文件转变成了 cpio 格式，变化不仅反映在文件格式上， linux 内核对这两种格式的 initrd 的处理有着截然的不同。本文首先介绍了什么是 initrd 技术，然后分别介绍了 Linux2.4 内核和 2.6 内核的 initrd 的处理流程。最后通过对 Linux2.6 内核的 initrd 处理部分代码的分析，使读者可以对 initrd 技术有一个全面的认识。为了更好的阅读本文，要求读者对 Linux 的 VFS 以及 initrd 有一个初步的了解。

**1．什么是 Initrd**

initrd 的英文含义是 boot loader initialized RAM disk，就是由 boot loader 初始化的内存盘。在 linux内核启动前， boot loader 会将存储介质中的 initrd 文件加载到内存，内核启动时会在访问真正的根文件系统前先访问该内存中的 initrd 文件系统。在 boot loader 配置了 initrd 的情况下，内核启动被分成了两个阶段，第一阶段先执行 initrd 文件系统中的"某个文件"，完成加载驱动模块等任务，第二阶段才会执行真正的根文件系统中的 /sbin/init 进程。这里提到的"某个文件"，Linux2.6 内核会同以前版本内核的不同，所以这里暂时使用了"某个文件"这个称呼，后面会详细讲到。第一阶段启动的目的是为第二阶段的启动扫清一切障爱，最主要的是加载根文件系统存储介质的驱动模块。我们知道根文件系统可以存储在包括IDE、SCSI、USB在内的多种介质上，如果将这些设备的驱动都编译进内核，可以想象内核会多么庞大、臃肿。

Initrd 的用途主要有以下四种：

1. linux 发行版的必备部件

linux 发行版必须适应各种不同的硬件架构，将所有的驱动编译进内核是不现实的，initrd 技术是解决该问题的关键技术。Linux 发行版在内核中只编译了基本的硬件驱动，在安装过程中通过检测系统硬件，生成包含安装系统硬件驱动的 initrd，无非是一种即可行又灵活的解决方案。

2. livecd 的必备部件

同 linux 发行版相比，livecd 可能会面对更加复杂的硬件环境，所以也必须使用 initrd。

3. 制作 Linux usb 启动盘必须使用 initrd

usb 设备是启动比较慢的设备，从驱动加载到设备真正可用大概需要几秒钟时间。如果将 usb 驱动编译进内核，内核通常不能成功访问 usb 设备中的文件系统。因为在内核访问 usb 设备时， usb 设备通常没有初始化完毕。所以常规的做法是，在 initrd 中加载 usb 驱动，然后休眠几秒中，等待 usb设备初始化完毕后再挂载 usb 设备中的文件系统。

4. 在 linuxrc 脚本中可以很方便地启用个性化 bootsplash。

**2．Linux2.4内核对 Initrd 的处理流程**

为了使读者清晰的了解Linux2.6内核initrd机制的变化，在重点介绍Linux2.6内核initrd之前，先对linux2.4内核的initrd进行一个简单的介绍。Linux2.4内核的initrd的格式是文件系统镜像文件，本文将其称为image-initrd，以区别后面介绍的linux2.6内核的cpio格式的initrd。 linux2.4内核对initrd的处理流程如下：

1. boot loader把内核以及/dev/initrd的内容加载到内存，/dev/initrd是由boot loader初始化的设备，存储着initrd。

2. 在内核初始化过程中，内核把 /dev/initrd 设备的内容解压缩并拷贝到 /dev/ram0 设备上。

3. 内核以可读写的方式把 /dev/ram0 设备挂载为原始的根文件系统。

4. 如果 /dev/ram0 被指定为真正的根文件系统，那么内核跳至最后一步正常启动。

5. 执行 initrd 上的 /linuxrc 文件，linuxrc 通常是一个脚本文件，负责加载内核访问根文件系统必须的驱动， 以及加载根文件系统。

6. /linuxrc 执行完毕，真正的根文件系统被挂载。

7. 如果真正的根文件系统存在 /initrd 目录，那么 /dev/ram0 将从 / 移动到 /initrd。否则如果 /initrd 目录不存在， /dev/ram0 将被卸载。

8. 在真正的根文件系统上进行正常启动过程 ，执行 /sbin/init。 linux2.4 内核的 initrd 的执行是作为内核启动的一个中间阶段，也就是说 initrd 的 /linuxrc 执行以后，内核会继续执行初始化代码，我们后面会看到这是 linux2.4 内核同 2.6 内核的 initrd 处理流程的一个显著区别。

**3．Linux2.6 内核对 Initrd 的处理流程**

linux2.6 内核支持两种格式的 initrd，一种是前面第 3 部分介绍的 linux2.4 内核那种传统格式的文件系统镜像－image-initrd，它的制作方法同 Linux2.4 内核的 initrd 一样，其核心文件就是 /linuxrc。另外一种格式的 initrd 是 cpio 格式的，这种格式的 initrd 从 linux2.5 起开始引入，使用 cpio 工具生成，其核心文件不再是 /linuxrc，而是 /init，本文将这种 initrd 称为 cpio-initrd。尽管 linux2.6 内核对 cpio-initrd和 image-initrd 这两种格式的 initrd 均支持，但对其处理流程有着显著的区别，下面分别介绍 linux2.6 内核对这两种 initrd 的处理流程。

cpio-initrd 的处理流程

1． boot loader 把内核以及 initrd 文件加载到内存的特定位置。

2． 内核判断initrd的文件格式，如果是cpio格式。

3． 将initrd的内容释放到rootfs中。

4． 执行initrd中的/init文件，执行到这一点，内核的工作全部结束，完全交给/init文件处理。

image-initrd的处理流程

1． boot loader把内核以及initrd文件加载到内存的特定位置。

2． 内核判断initrd的文件格式，如果不是cpio格式，将其作为image-initrd处理。

3． 内核将initrd的内容保存在rootfs下的/initrd.image文件中。

4． 内核将/initrd.image的内容读入/dev/ram0设备中，也就是读入了一个内存盘中。

5． 接着内核以可读写的方式把/dev/ram0设备挂载为原始的根文件系统。

6． 如果/dev/ram0被指定为真正的根文件系统，那么内核跳至最后一步正常启动。

7． 执行initrd上的/linuxrc文件，linuxrc通常是一个脚本文件，负责加载内核访问根文件系统必须的驱动以及加载根文件系统。

8． ***/linuxrc***执行完毕，常规根文件系统被挂载

9． 如果常规根文件系统存在***/initrd***目录，那么***/dev/ram0***将从***/***移动到***/initrd***。否则如果**/initrd**目录不存在， **/dev/ram0**将被卸载。

10． 在常规根文件系统上进行正常启动过程，执行**/sbin/init**。

通过上面的流程介绍可知，Linux2.6内核对image-initrd的处理流程同linux2.4内核相比并没有显著的变化， cpio-initrd的处理流程相比于image-initrd的处理流程却有很大的区别，流程非常简单，在后面的源代码分析中，读者更能体会到处理的简捷。

4．cpio-initrd同image-initrd的区别与优势

没有找到正式的关于cpio-initrd同image-initrd对比的文献，根据笔者的使用体验以及内核代码的分析，总结出如下三方面的区别，这些区别也正是cpio-initrd的优势所在：

**cpio-initrd的制作方法更加简单**

cpio-initrd的制作非常简单，通过两个命令就可以完成整个制作过程

#假设当前目录位于准备好的initrd文件系统的根目录下

bash# find . | cpio -c -o > ../initrd.img

bash# gzip ../initrd.img

而传统initrd的制作过程比较繁琐，需要如下六个步骤

#假设当前目录位于准备好的initrd文件系统的根目录下

bash# dd if=/dev/zero of=../initrd.img bs=512k count=5

bash# mkfs.ext2 -F -m0 ../initrd.img

bash# mount -t ext2 -o loop ../initrd.img /mnt

bash# cp -r \* /mnt

bash# umount /mnt

bash# gzip -9 ../initrd.img

本文不对上面命令的含义作细节的解释，因为本文主要介绍的是linux内核对initrd的处理，对上面命令不理解的读者可以参考相关文档。

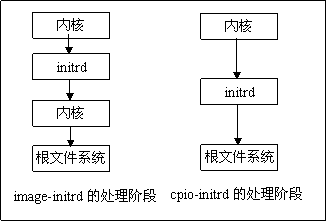
**cpio-initrd的内核处理流程更加简化**

通过上面initrd处理流程的介绍，cpio-initrd的处理流程显得格外简单，通过对比可知cpio-initrd的处理流程在如下两个方面得到了简化：

1． cpio-initrd并没有使用额外的ramdisk,而是将其内容输入到rootfs中，其实rootfs本身也是一个基于内存的文件系统。这样就省掉了ramdisk的挂载、卸载等步骤。

2． cpio-initrd启动完**/init**进程，内核的任务就结束了，剩下的工作完全交给**/init**处理；而对于image-initrd，内核在执行完**/linuxrc**进程后，还要进行一些收尾工作，并且要负责执行真正的根文件系统的**/sbin/init**。通过图1可以更加清晰的看出处理流程的区别：

图1内核对cpio-initrd和image-initrd处理流程示意图



cpio-initrd的职责更加重要

如图1所示，cpio-initrd不再象image-initrd那样作为linux内核启动的一个中间步骤，而是作为内核启动的终点，内核将控制权交给cpio-initrd的/init文件后，内核的任务就结束了，所以在/init文件中，我们可以做更多的工作，而不比担心同内核后续处理的衔接问题。当然目前linux发行版的cpio-initrd的/init文件的内容还没有本质的改变，但是相信initrd职责的增加一定是一个趋势。

5．linux2.6内核initrd处理的源代码分析

上面简要介绍了Linux2.4内核和2.6内核的initrd的处理流程，为了使读者对于Linux2.6内核的initrd的处理有一个更加深入的认识，下面将对Linuxe2.6内核初始化部分同initrd密切相关的代码给予一个比较细致的分析，为了讲述方便，进一步明确几个代码分析中使用的概念：

rootfs: 一个基于内存的文件系统，是linux在初始化时加载的第一个文件系统,关于它的进一步介绍可以参考文献[4]。

initramfs: initramfs同本文的主题关系不是很大，但是代码中涉及到了initramfs，为了更好的理解代码，这里对其进行简单的介绍。Initramfs是在 kernel 2.5中引入的技术，实际上它的含义就是：在内核镜像中附加一个cpio包，这个cpio包中包含了一个小型的文件系统，当内核启动时，内核将这个cpio包解开，并且将其中包含的文件系统释放到rootfs中，内核中的一部分初始化代码会放到这个文件系统中，作为用户层进程来执行。这样带来的明显的好处是精简了内核的初始化代码，而且使得内核的初始化过程更容易定制。Linux 2.6.12内核的 initramfs还没有什么实质性的东西，一个包含完整功能的initramfs的实现可能还需要一个缓慢的过程。对于initramfs的进一步了解可以参考文献[1][2][3]。

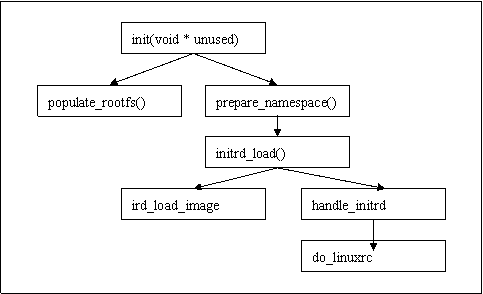
cpio-initrd: 前面已经定义过，指linux2.6内核使用的cpio格式的initrd。

image-initrd: 前面已经定义过，专指传统的文件镜像格式的initrd。

realfs: 用户最终使用的真正的文件系统。

内核的初始化代码位于 init/main.c 中的 static int init(void \* unused)函数中。同initrd的处理相关部分函数调用层次如下图，笔者按照这个层次对每一个函数都给予了比较详细的分析，为了更好的说明，下面列出的代码中删除了同本文主题不相关的部分：

图2 initrd相关代码的调用层次关系图



init函数是内核所有初始化代码的入口，代码如下，其中只保留了同initrd相关部分的代码。

static int init(void \* unused){

[1] populate\_rootfs();

[2] if (sys\_access((const char \_\_user \*) "/init", 0) == 0)

execute\_command = "/init";

else

prepare\_namespace();

[3] if (sys\_open((const char \_\_user \*) "/dev/console", O\_RDWR, 0) < 0)

printk(KERN\_WARNING "Warning: unable to open an initial console.\n");

(void) sys\_dup(0);

(void) sys\_dup(0);

[4] 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.");

}

代码[1]：populate\_rootfs函数负责加载initramfs和cpio-initrd，对于populate\_rootfs函数的细节后面会讲到。

代码[2]：如果rootfs的根目录下中包含/init进程，则赋予execute\_command,在init函数的末尾会被执行。否则执行prepare\_namespace函数，initrd是在该函数中被加载的。

代码[3]：将控制台设置为标准输入，后续的两个sys\_dup(0),则复制标准输入为标准输出和标准错误输出。

代码[4]：如果rootfs中存在init进程，就将后续的处理工作交给该init进程。其实这段代码的含义是如果加载了cpio-initrd则交给cpio-initrd中的/init处理，否则会执行realfs中的init。读者可能会问：如果加载了cpio-initrd, 那么realfs中的init进程不是没有机会运行了吗？确实，如果加载了cpio-initrd,那么内核就不负责执行realfs的init进程了，而是将这个执行任务交给了cpio-initrd的init进程。解开fedora core4的initrd文件，会发现根目录的下的init文件是一个脚本，在该脚本的最后一行有这样一段代码：

………..

switchroot --movedev /sysroot

就是switchroot语句负责加载realfs,以及执行realfs的init进程。

对cpio-initrd的处理

对cpio-initrd的处理位于populate\_rootfs函数中。

void \_\_init populate\_rootfs(void){

[1] char \*err = unpack\_to\_rootfs(\_\_initramfs\_start,

\_\_initramfs\_end - \_\_initramfs\_start, 0);

[2] if (initrd\_start) {

[3] err = unpack\_to\_rootfs((char \*)initrd\_start,

initrd\_end - initrd\_start, 1);

[4] if (!err) {

printk(" it is\n");

unpack\_to\_rootfs((char \*)initrd\_start,

initrd\_end - initrd\_start, 0);

free\_initrd\_mem(initrd\_start, initrd\_end);

return;

}

[5] fd = sys\_open("/initrd.image", O\_WRONLY|O\_CREAT, 700);

if (fd >= 0) {

sys\_write(fd, (char \*)initrd\_start,

initrd\_end - initrd\_start);

sys\_close(fd);

free\_initrd\_mem(initrd\_start, initrd\_end);

}

}

代码[1]：加载initramfs， initramfs位于地址\_\_initramfs\_start处，是内核在编译过程中生成的，initramfs的是作为内核的一部分而存在的，不是 boot loader加载的。前面提到了现在initramfs没有任何实质内容。

代码[2]：判断是否加载了initrd。无论哪种格式的initrd，都会被boot loader加载到地址initrd\_start处。

代码[3]：判断加载的是不是cpio-initrd。实际上 unpack\_to\_rootfs有两个功能一个是释放cpio包，另一个就是判断是不是cpio包， 这是通过最后一个参数来区分的， 0：释放 1：查看。

代码[4]：如果是cpio-initrd则将其内容释放出来到rootfs中。

代码[5]：如果不是cpio-initrd,则认为是一个image-initrd，将其内容保存到/initrd.image中。在后面的image-initrd的处理代码中会读取/initrd.image。

对image-initrd的处理 在prepare\_namespace函数里，包含了对image-initrd进行处理的代码，相关代码如下：

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

[1] if (initrd\_load())

goto out;

out:

umount\_devfs("/dev");

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

sys\_chroot(".");

security\_sb\_post\_mountroot();

mount\_devfs\_fs ();

}

代码[1]：执行initrd\_load函数，将initrd载入，如果载入成功的话initrd\_load函数会将realfs的根设置为当前目录。

代码[2]：将当前目录即realfs的根mount为Linux VFS的根。initrd\_load函数执行完后，将真正的文件系统的根设置为当前目录。

initrd\_load函数负责载入image-initrd，代码如下：

int \_\_init initrd\_load(void)

{

[1] if (mount\_initrd) {

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

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

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

handle\_initrd();

return 1;

}

}

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

return 0;

}

代码[1]：如果加载initrd则建立一个ram0设备 /dev/ram。

代码[2]：/initrd.image文件保存的就是image-initrd，rd\_load\_image函数执行具体的加载操作，将image-initrd的文件内容释放到ram0里。判断ROOT\_DEV!=Root\_RAM0的含义是，如果你在grub或者lilo里配置了 root=/dev/ram0 ,则实际上真正的根设备就是initrd了，所以就不把它作为initrd处理 ，而是作为realfs处理。

handle\_initrd()函数负责对initrd进行具体的处理，代码如下：

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

[1] real\_root\_dev = new\_encode\_dev(ROOT\_DEV);

[2] create\_dev("/dev/root.old", Root\_RAM0, NULL);

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

[3] sys\_mkdir("/old", 0700);

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

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

/\* move initrd over / and chdir/chroot in initrd root \*/

[4] sys\_chdir("/root");

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

sys\_chroot(".");

mount\_devfs\_fs ();

[5] pid = kernel\_thread(do\_linuxrc, "/linuxrc", SIGCHLD);

if (pid > 0) {

while (pid != sys\_wait4(-1, &i, 0, NULL))

yield();

}

/\* move initrd to rootfs' /old \*/

sys\_fchdir(old\_fd);

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

/\* switch root and cwd back to / of rootfs \*/

[6] sys\_fchdir(root\_fd);

sys\_chroot(".");

sys\_close(old\_fd);

sys\_close(root\_fd);

umount\_devfs("/old/dev");

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

sys\_chdir("/old");

return;

}

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

mount\_root();

[9] printk(KERN\_NOTICE "Trying to move old root to /initrd ... ");

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);

printk("failed\n");

printk(KERN\_NOTICE "Unmounting old root\n");

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

printk(KERN\_NOTICE "Trying to free ramdisk memory ... ");

if (fd < 0) {

error = fd;

} else {

error = sys\_ioctl(fd, BLKFLSBUF, 0);

sys\_close(fd);

}

printk(!error ? "okay\n" : "failed\n");

}

handle\_initrd函数的主要功能是执行initrd的linuxrc文件，并且将realfs的根目录设置为当前目录。

代码[1]：real\_root\_dev，是一个全局变量保存的是realfs的设备号。

代码[2]：调用mount\_block\_root函数将initrd文件系统挂载到了VFS的/root下。

代码[3]：提取rootfs的根的文件描述符并将其保存到root\_fd。它的作用就是为了在chroot到initrd的文件系统，处理完initrd之后要，还能够返回rootfs。返回的代码参考代码[7]。

代码[4]：chroot进入initrd的文件系统。前面initrd已挂载到了rootfs的/root目录。

代码[5]：执行initrd的linuxrc文件，等待其结束。

代码[6]：initrd处理完之后，重新chroot进入rootfs。

代码[7]：如果real\_root\_dev在 linuxrc中重新设成Root\_RAM0，则initrd就是最终的realfs了，改变当前目录到initrd中，不作后续处理直接返回。

代码[8]：在linuxrc执行完后，realfs设备已经确定，调用mount\_root函数将realfs挂载到root\_fs的 /root目录下，并将当前目录设置为/root。

代码[9]：后面的代码主要是做一些收尾的工作，将initrd的内存盘释放。

到此代码分析完毕。

**6．结束语**

通过本文前半部分对cpio-initrd和imag-initrd的阐述与对比以及后半部分的代码分析，我相信读者对Linux 2.6内核的initrd技术有了一个较为全面的了解。在本文的最后，给出两点最重要的结论：

1． 尽管Linux2.6既支持cpio-initrd，也支持image-initrd，但是cpio-initrd有着更大的优势，在使用中我们应该优先考虑使用cpio格式的initrd。

2． cpio-initrd相对于image-initrd承担了更多的初始化责任，这种变化也可以看作是内核代码的用户层化的一种体现，我们在其它的诸如FUSE等项目中也看到了将内核功能扩展到用户层实现的尝试。精简内核代码，将部分功能移植到用户层必然是linux内核发展的一个趋势。

**参考资料**

从下面三篇文章中，可以获得更多的关于initramfs的知识：

[1]http://tree.celinuxforum.org/pubwiki/moin.cgi/EarlyUserSpace

[2]http://lwn.net/Articles/14776/

[3]http://www.ussg.iu.edu/hypermail/linux/kernel/0211.0/0341.html

从下面这篇文章中读者可以了解到关于linux VSF、rootfs的相关知识：

[4] http://www.ibm.com/developerworks/cn/linux/l-vfs/

下面是一些initrd的参考资料：

[5] http://www.die.net/doc/linux/man/man4/initrd.4.html