# #【ARM】制作Linux 文件系统

[TOC]

# #1从零开始由busybox 建立文件系统

## ##1.1文件系统在LINUX 中的结构图



## ##1.2编译好kernel

编译linux kernel的具体操作在之前文章讲过，下面简要提示一下。

**修改编译器PATH，指定寻找路径。**

**cd kernel-2.6.13 (进入内核目录)**

**cp config\_n35 .config**

**make （编译内核，假设已经设置好了）**

**cd arch/arm/boot （进入生成的zImage 目录）**

**mkimage.sh （产生有uboot 可以引导的uImage 内核文件）**

## ##1.3Uboot 状态下显示的环境设置信息

```

setenv bootcmd "tftp 30008000 uimage;bootm"

setenv ipaddr 192.168.1.244

setenv serverip 192.168.1.112

setenv gatewayip 192.168.1.1

setenv bootargs noinitrd root=/dev/nfs rw nfsroot=192.168.1.113:/home/wu-being/linux/my\_nfs ip=192.168.1.244:192.168.1.114:192.168.1.1:255.255.255.0 console=ttySAC0,115200 init=/linuxrc mem=64M

```

我们启用NFS 方式装载文件系统，首先启动PC 上的NFS 服务器

**vi /etc/exports (添加你要装载的文件系统目录)**

**`** **/home/wu-being/linux/my\_nfs 192.168.\*.\*(rw,sync,no\_root\_squash)`**

**/etc/rc.d/init.d/portmap restart**

**/etc/rc.d/init.d/nfs restart**

然后启动内核，查看结果？？

## ##1.4开始busybox 生成文件系统

Busybox 是很多标准Linux 工具的单一可执行实现，它说明了一个问题，

很多标准的linux 工具可以共享很多共同的元素。

- 第1步：从[www.busybox.net](http://www.busybox.net) 下载最新的版本busybox1.18.4.

- 第2步：解压并打补丁

`tar –jxvf busybox-1.18.4.tar.bz2`

```

wu-being@ubuntu:~/linux/bosybox$ l

busybox-1.18.4/ busybox-1.18.4-hush.patch\*

busybox-1.18.4-buildsys.patch\* busybox-1.18.4.tar.bz2\*

busybox-1.18.4-fuser.patch\*

wu-being@ubuntu:~/linux/bosybox$ patch -p0 <./busybox-1.18.4-buildsys.patch

wu-being@ubuntu:~/linux/bosybox$ patch -p0 <./busybox-1.18.4-fuser.patch

wu-being@ubuntu:~/linux/bosybox$ patch -p0 <./busybox-1.18.4-hush.patch

```

- 第3步：配制编译器路径，\*\***注意这个编译器版本要和编译Kernel的编译器版本同样。\*\***

```

wu-being@ubuntu:~/linux/bosybox/busybox-1.18.4$ l /home/wu-being/linux/3.4.1/bin/

arm-linux-addr2line\* arm-linux-gcc\* arm-linux-objcopy\*

arm-linux-ar\* arm-linux-gcc-3.4.1\* arm-linux-objdump\*

arm-linux-as\* arm-linux-gccbug\* arm-linux-ranlib\*

arm-linux-c++\* arm-linux-gcov\* arm-linux-readelf\*

arm-linux-c++filt\* arm-linux-gprof\* arm-linux-size\*

arm-linux-cpp\* arm-linux-ld\* arm-linux-strings\*

arm-linux-g++\* arm-linux-nm\* arm-linux-strip\*

wu-being@ubuntu:~/linux/bosybox/busybox-1.18.4$ export PATH=$PATH:/home/wu-being/linux/3.4.1/bin/

wu-being@ubuntu:~/linux/bosybox/busybox-1.18.4$ arm-linux-gcc（TAB）

arm-linux-gcc arm-linux-gcc-4.4.3

arm-linux-gcc-3.4.1 arm-linux-gccbug

wu-being@ubuntu:~/linux/bosybox/busybox-1.18.4$ arm-linux-gcc

```

- 第4步：进入目录`busybox-1.18.4/`执行命令`make defconfig` (default config)

- 第5步：指定构架和编译工具编译和安装busybox

1.先执行编译命令：`make ARCH=arm CROSS\_COMPILE=arm-linux-`

```

如果没有定义ARPHRD\_INFINIBAND，定义之

**vi networking/interface.c**

#define INFINIABAND\_ALEN 20

#define ARPHRD\_INFINIBAND 20 //++

```

```

如果没有定义IFLA\_OPERSTATE，屏蔽它

**vi networking/libiproute/ipaddress.c**

#ifdef IFLA\_OPERSTATE //++

if (tb[IFLA\_OPERSTATE]) {

static const char operstate\_labels[] ALIGN1 =

"UNKNOWN\0""NOTPRESENT\0""DOWN\0""LOWERLAYERDOWN\0"

"TESTING\0""DORMANT\0""UP\0";

printf("state %s ", nth\_string(operstate\_labels,

\*(uint8\_t\*)RTA\_DATA(tb[IFLA\_OPERSTATE])));

}

#endif //++

```

```

如果编译mkfs\_ext2 工具错误，去掉它。

make menuconfig --> Linux System Utilites --> mkfs\_ext2 (disable) --->

save and quit

```

2.命令编译成功后，执行安装命令：

`make ARCH=arm CROSS\_COMPILE=arm-linux- **install`**

- 第6步：拷贝文件系统到NFS 能输出的目录

mkdir /home/wu-being/linux/my\_nfs

cp -rf \_INSTALL/\* /home/wu-being/linux/my\_nfs

- 第7步：修改nfs 配置

vi /etc/exports and append "/home/wu-being/linux/my\_nfs 192.168.\*.\*(rw,sync,no\_root\_squash)

"

/etc/init.d/nfs-kernel-server restart

- 第8步：下载内核到目标板运行看结果

配制板子文件系统目录，并保存。

```

setenv bootargs noinitrd root=/dev/nfs rw nfsroot=192.168.1.113:/home/wu-being/linux/my\_nfs ip=192.168.1.244:192.168.1.114:192.168.1.1:255.255.255.0 console=ttySAC0,115200 init=/linuxrc mem=64M

saveenv

```

应该会看到文件系统已经Mount 成功，但是现实console 错误，应该是没有这个设备。

```

VFS: Mounted root (nfs filesystem).

mount\_devfs\_fs(): unable to mount devfs, err: -2

Freeing init memory: 144K

Warning: unable to open an initial console.

Kernel panic - not syncing: No init found. Try passing init= option to kernel.

```

- 第9步：建立串口设备

建立dev和lib目录，并在文件系统的dev目录创建主设备号为5次设备号为1的`console`字符设备：`mknod console c 5 1`

- 第10步：下载并启动内核显示如下

```

VFS: Mounted root (nfs filesystem).

Mounted devfs on /dev

Freeing init memory: 172K

Kernel panic - not syncing: No init found. Try passing init= option to kernel.

```

看来init 不存在或不能运行.

- 第11步：指定init

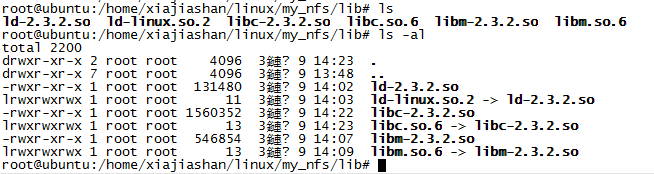
ln -sf bin/busybox init

- 第12步：再启动内核

init 还是不能运行，也许就是busybox 不能运行，用文件形式打开busybox发现是要一些库文件。

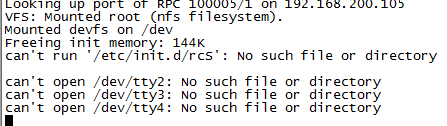
从原来挂载成功的文件系统目录的Lib下拷贝所需库

最后结果



第12步：再启动内核

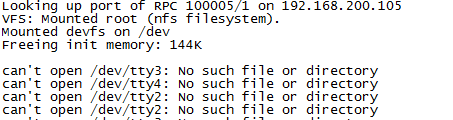
看看效果



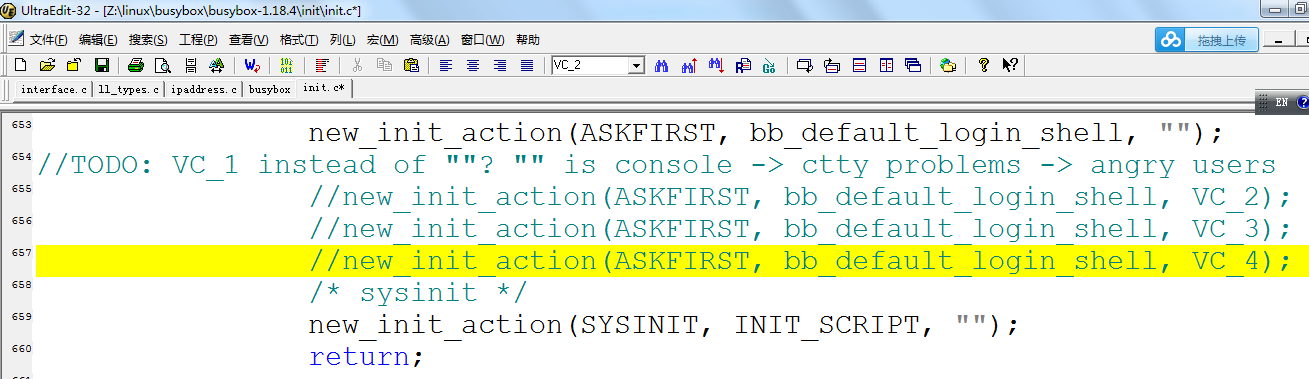
缺失etc下面的文件



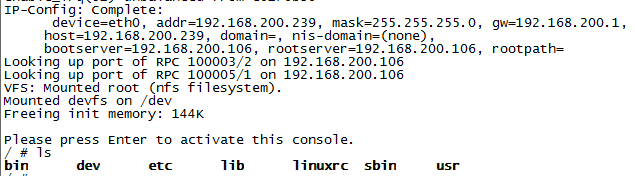
再看



修改busybox源码



重新编译，安装，覆盖



挂载成功。

## ##1.5完善文件系统

第一步： init 会调用/etc/inittab 文件，并根据其做相应的操作。

add file etc/inittab, the init will use it to initialize system

------

::respawn:-/bin/sh

运行内核，显示OK

VFS: Mounted root (nfs filesystem).

Mounted devfs on /dev

Freeing init memory: 172K

/ #

/ # ls

bin dev etc init lib sbin usr

/ #

第二步：可以手工做些设置

mkdir proc

mkdir sys

mount -t proc proc /proc

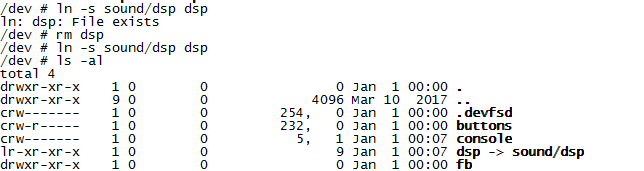
mount -t sysfs sysfs /sys

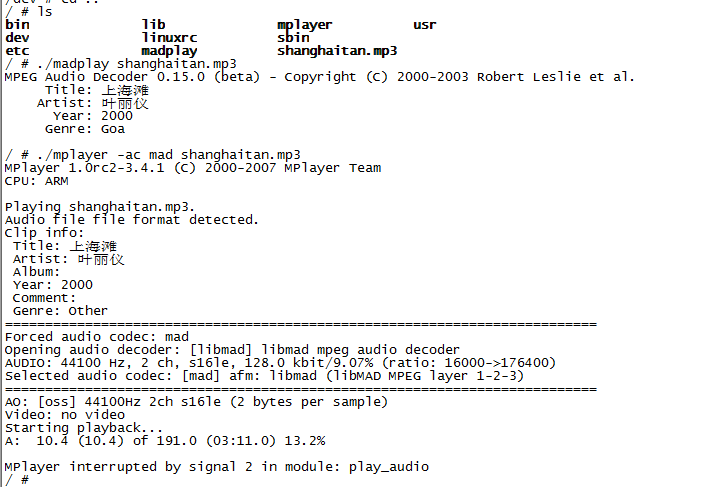
mdev -s

第三步：play MP3

play mp3

3





./mplayer -ac mad yishengyouni.mp3

第四步：添加Login

vi passwd and vi group (create tw files)

adduser root

modify passwd file and change its user id and group id to 0.

passwd root (modify password)

修改inittab 为：

::sysinit:/etc/rcS

::respawn:/bin/login

::shutdown:/bin/umount -a

创造文件rcS

mount -t proc proc /proc

mount -t sysfs sysfs /sys

mdev -s

telnetd &

# #2Linux 文件系统结构

Linux 下文件系统是对复杂系统进行合理抽象的一个经典的例子，它通过一套统一的接口函数对不同的文件进行操作（普通文件，字符设备文件，块设备文件和socket 文件）。

索引节点(inode): 用来存储数据的信息，如大小，属主，归属和读写权限

等。

块(block): 存储数据。

虚拟文件系统VFS(virtual File System)是Linux 内核层实现的一种构架，

为用户空间提供统一的文件操作借口。



逻辑图

## ##2.1ext2 文件系统的总体存储布局



### ###2.1.1Block

文件系统中存储的\*\*最小单位\*\*是\*\*块（Block）\*\* ，一个块究竟多大在格式化时确定的，例如mke2fs 的-b 选项可以设定块大小为1024、2048 或4096 字节。

### ###2.1.2Boot block

而上图中启动块（Boot Block） 的大小是确定的，就是1KB，启动块是由PC 标准规定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使用启动块。启动块BootBlock之后才是ext2 文件系统的开始。

### ###2.1.3Block Group:

ext2 文件系统将整个分区划成n 个同样大小的块，多个块组成块组（Block

Group） ，每个块组都由以下部分组成。

#### Super Block:

超级块 （Super Block)(占用一个块大小)描述整个分区的文件系统信息，例如块大小、文件系统版本号、上次mount的时间等等。超级块在每个块组的开头都有一份拷贝(也就是说每个块组组开头1024 字节为超级快，内容相同，估计是出于安全考量)。占用每个块组的第一个块，共1024 字节，记录着magic(魔数)，block\_size(块大小)，block\_count(块数)，block\_per\_group(每组的块数)等信息。

#### GDT，Group Descriptor Table

块组描述符表（GDT，Group Descriptor Table）（占用一个块大小）每个块组的块组描述符占用32 个字节（见下面），整个分区的块组的块组描述符连成一片放在一个块里面就组成了块组描述符表。每个块组描述符（GroupDescriptor） 存储一个块组的描述信息，例如在这个块组中从哪里开始是inode表，从哪里开始是数据块，空闲的inode 和数据块还有多少个等等。

和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息

是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第0 个块组中的拷贝，当执行e2fsck 检查文件系统一致性时，第0 个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第0 个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

注意，硬盘分区上的所有这些 group descriptors 要能塞在一个 block 里

面。也就是说 groups\_count \* descriptor\_size 必须小于等于 block\_size。

知道了硬盘分区上一共有多少个 block group，我们就可以把这么多个

group descriptors 读出来了。先来看看 group descriptor 是什么样子的。

struct ext3\_group\_desc

{

\_\_u32 bg\_block\_bitmap; /\* block 指针指向 block bitmap \*/

\_\_u32 bg\_inode\_bitmap; /\* block 指针指向 inode bitmap \*/

\_\_u32 bg\_inode\_table; /\* block 指针指向 inodes table \*/

\_\_u16 bg\_free\_blocks\_count; /\* 空闲的 blocks 计数 \*/

\_\_u16 bg\_free\_inodes\_count; /\* 空闲的 inodes 计数 \*/

\_\_u16 bg\_used\_dirs\_count; /\* 目录计数 \*/

\_\_u16 bg\_pad; /\* 可以忽略 \*/

\_\_u32 bg\_reserved[3]; /\* 可以忽略 \*/

};

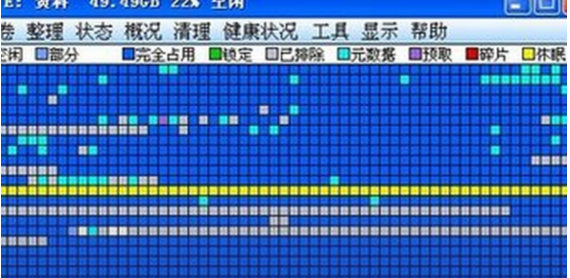
每个 group descriptor 是 32 bytes 那么大。从上面，我们看到了三个

关键的 block 指针，这三个关键的 block 指针，我们已经在前面都提到过了。

#### 块位图 （Block Bitmap）（占用一个块）

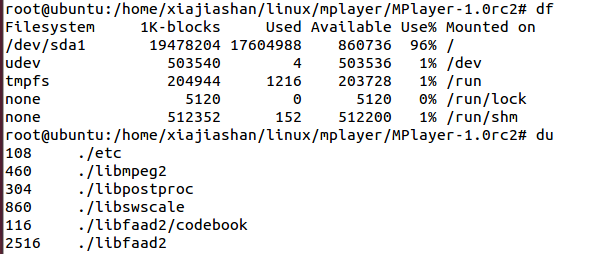
一个块组中的块是这样利用的：数据块存储所有文件的数据，比如某个分

区的块大小是1024 字节，某个文件是2049 字节，那么就需要三个数据块来存，即使第三个块只存了一个字节也需要占用一个整块；超级块、块组描述符表、块位图、inode 位图、inode 表这几部分存储该块组的描述信息。那么如何知道哪些块已经用来存储文件数据或其它描述信息，哪些块仍然空闲可用呢？**块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个bit 代表本块组中的一个块，这个bit 为1 表示该块已用，这个bit 为0 表示该块空闲可用。**



为什么用df 命令统计整个磁盘的已用空间非常快呢？因为只需要查看每

个块组的块位图即可，而不需要搜遍整个分区（所有块组信息通过GDT形成了链表）。相反，用du 命令查看一个较大目录的已用空间就非常慢，因为不可避免地要搜遍整个目录的所有文件。



与此相联系的另一个问题是：

在格式化一个分区时究竟会划出多少个块组呢？主要的限制在于块位图本身必须只占一个块。用mke2fs 格式化时默认块大小是1024 字节，可以用-b 参数指定块大小，现在设块大小指定为b 字节，那么一个块可以有8b 个bit，这样大小的一个块位图就可以表示8b 个块的占用情况，因此一个块组最多可以有 8b 个块，如果整个分区有s 个块，那么就可以有s/(8b)个块组。格式化时可以用-g 参数指定一个块组有多少个块，但是通常不需要手动指定，mke2fs 工具会计算出最优的数值。

#### inode 位图（inode Bitmap）（占用一个块）

和块位图类似，本身占一个块，其中每个bit 表示一个inode 是否空闲可用。

#### inode 表（inode Table）（占用n 个块）

我们知道，一个文件除了数据需要存储之外，一些描述信息也需要存储，

例如文件类型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等，也就是ls -l 命令看到的那些信息，这些信息存在inode 中而不是数据块中。每个文件都有一个inode，一个块组中的所有 inode 组成了inode 表。

inode 表占多少个块在格式化时就要决定并写入块组描述符中，mke2fs 格式化工具的默认策略是一个块组有多少个8KB 就分配多少个inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个8KB 就分配多少个inode，换句话说，如果平均每个文件的大小是8KB，当分区存满的时候inode表会得到比较充分的利用，数据块也不浪费。如果这个分区存的都是很大的文件（比如电影），则数据块用完的时候inode 会有一些浪费，如果这个分区存的都是很小的文件（比如源代码），则有可能数据块还没用完inode 就已经用完了，数据块可能有很大的浪费。如果用户在格式化时能够对这个分区以后要存储的文件大小做一个预测，也可以用mke2fs 的-i 参数手动指定每多少个字节分配一个inode。

#### 数据块（Data Block）（n 个块）

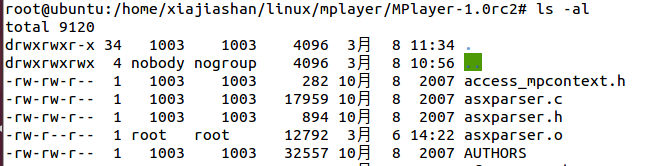
根据不同的文件类型有以下几种情况

对于常规文件，文件的数据存储在数据块中。

对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l 命令看到的其它信息都

保存在该文件的inode 中。注意这个概念：目录也是一种文件，是一种特殊

类型的文件。



对于符号链接，如果目标路径名较短则直接保存在inode 中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

### ###2.1.4我们总结一下

一个格式好的文件系统，分成固定大小的块，块组成块组，块的大小相

同，每个块组的块数也相同；每个块组由超级快（记录块大小，有多少块，每个块组由多少块组成等），块组描述符表（由连贯的块组描述符组成，描述块位图，inode 位图和

inode 表的地址等），块位图（哪个块使用了，哪个块空闲），inode位图（那个inode 占用了，那没有占用），inode 表（文件类型，权限，文件大小，创建/修改/访问时间和文件内容所在的块的地址）和数据块（文件的数据）组成。

一个文件由文件内容和描述信息组成，文件内容保存在数据块中，文件描述信息保存在inode 中。

知道了inode 的数值(ls –i),可以读出文件内容，如何，见下面.前面都准备好了以后，我们现在终于可以开始读取文件了。首先要读的，当然是文件系统的根目录。注意，这里所谓的根目录，是相对于这一个文件系统或者说硬盘分区而言的，它并不一定是整个 Linux 操作系统上的根目录。这里的这个 root 目录存放在一个固定的 inode 中，这就是文件系统上的 inode 2。需要提到 inode 计数同 block 计数一样，也是全局性质的。这里需要特别注意的是，inode 计数是从 1 开始的，而前面我们提到过 block 计数是从 0 开始，这个不同在开发程序的时候要特别留心。\_\_

那么，我们先来看一下得到一个 inode 号数以后，怎样读取这个 inode 中的用户数据。在 super block 中有一个字段 s\_inodes\_per\_group 记载了每个block group 中有多少个 inode 。用我们得到的 inode 号数除以s\_inodes\_per\_group，我们就知道了我们要的这个 inode 是在哪一个 block group 里面，这个除法的余数也告诉我们，我们要的这个 inode 是这个 block group 里面的第几个 inode；然后，我们可以先找到这个 block group 的 group descriptor，从这个 descriptor，我们找到这个 group 的 inode table，再从inode table 找到我们要的第几个 inode，再以后，我们就可以开始读取 inode中的用户数据了。

这个公式是这样的：block\_group = (ino - 1) / s\_inodes\_per\_group。这里 ino 就是我们的 inode 号数。而 offset = (ino - 1) % s\_inodes\_per\_group，这个 offset 就指出了我们要的 inode 是这个 block group 里面的第几个inode。找到这个 inode 之后，我们来具体的看看 inode 是什么样的。

struct ext3\_inode {

\_\_u16 i\_mode; /\* File mode \*/

\_\_u16 i\_uid; /\* Low 16 bits of Owner Uid \*/

\_\_u32 i\_size; /\* 文件大小，单位是 byte \*/

\_\_u32 i\_atime; /\* Access time \*/

\_\_u32 i\_ctime; /\* Creation time \*/

\_\_u32 i\_mtime; /\* Modification time \*/

\_\_u32 i\_dtime; /\* Deletion Time \*/

\_\_u16 i\_gid; /\* Low 16 bits of Group Id \*/

\_\_u16 i\_links\_count; /\* Links count \*/

\_\_u32 i\_blocks; /\* blocks 计数 \*/

\_\_u32 i\_flags; /\* File flags \*/

\_\_u32 l\_i\_reserved1; /\* 可以忽略 \*/

\_\_u32 i\_block[EXT3\_N\_BLOCKS]; /\* 一组 block 指针 \*/

\_\_u32 i\_generation; /\* 可以忽略 \*/

\_\_u32 i\_file\_acl; /\* 可以忽略 \*/

\_\_u32 i\_dir\_acl; /\* 可以忽略 \*/

\_\_u32 i\_faddr; /\* 可以忽略 \*/

\_\_u8 l\_i\_frag; /\* 可以忽略 \*/

\_\_u8 l\_i\_fsize; /\* 可以忽略 \*/

\_\_u16 i\_pad1; /\* 可以忽略 \*/

\_\_u16 l\_i\_uid\_high; /\* 可以忽略 \*/

\_\_u16 l\_i\_gid\_high; /\* 可以忽略 \*/

\_\_u32 l\_i\_reserved2; /\* 可以忽略 \*/

};

我们看到在 inode 里面可以存放 EXT3\_N\_BLOCKS（= 15）这么多个 block 指

针。用户数据就从这些 block 里面获得。15 个 blocks 不一定放得下全部的用

户数据，在这里 ext3 文件系统采取了一种分层的结构。这组 15 个 block 指

针的前 12 个是所谓的 direct blocks，里面直接存放的就是用户数据。第 13

个 block，也就是所谓的 indirect block，里面存放的全部是 block 指针，这

些 block 指针指向的 block 才被用来存放用户数据。第 14 个 block 是所谓

的 double indirect block，里面存放的全是 block 指针，这些 block 指针指

向的 block 也被全部用来存放 block 指针，而这些 block 指针指向的 block，

才被用来存放用户数据。第 15 个 block 是所谓的 triple indirect block，

比上面说的 double indirect block 有多了一层 block 指针。作为练习，读者

可以计算一下，这样的分层结构可以使一个 inode 中最多存放多少字节的用户

数据。

一个 inode 里面实际有多少个 block，这是由 inode 字段 i\_size 再通过

计算得到的。i\_size 记录的是文件或者目录的实际大小，用它的值除以 block

的大小，就可以得出这个 inode 一共占有几个 block。现在我们已经可以读取 inode 的内容了，再往后，我们将要读取文件系统上文件和目录的内容。读取文件的内容，只要把相应的 inode 的内容全部读出来就行了；而目录只是一种固定格式的文件，这个文件按照固定的格式记录了目录中有哪些文件，以及它们的文件名，和 inode 号数等等。

struct ext3\_dir\_entry\_2 {

\_\_u32 inode; /\* Inode 号数 \*/

\_\_u16 rec\_len; /\* Directory entry length \*/

\_\_u8 name\_len; /\* Name length \*/

\_\_u8 file\_type;

char name[EXT3\_NAME\_LEN]; /\* File name \*/

};

上面用到的 EXT3\_NAME\_LEN 是 255。注意，在硬盘分区上的 dir entry 不是固定长度的，每个 dir entry 的长度由上面的 rec\_len 字段记录。

问题：目录和文件名如何保存，如何读取？

（目录只是一种固定格式的文件，可以“vim 目录”，看到这个目录文件的

内容）

## ##2.2VFS 的建立

经过前面的ext2 文件系统格式的介绍，我们起码知道了如何去获取文件系

统的信息和如何读取文件内容。其中也建立了两个很重要的概念super block

超级块和inode 节点信息，然后如果理解dentry（directory entry）概念就更

好了。

关于VFS，看开头的描述和逻辑图。

在2.6 内核中，启动的过程中由vfs 建立的rootfs 根文件系统过程如下：

kernel\_start()

|\_\_\_\_start\_kernel()

|\_\_\_\_vfs\_caches\_init()(fs/dcache.c)

|\_\_\_\_dcache\_init()

|\_\_\_\_inode\_init()

|\_\_\_\_files\_init()

|\_\_\_\_mnt\_init()

| |\_\_\_\_init\_rootfs()

| | |\_\_\_register\_filesystem(

| | &rootfs\_fs\_type)

| |\_\_\_\_init\_mount\_tree()

| |\_\_\_ do\_kern\_mount()

| (建立系统的根”/”)

|\_\_\_\_bdev\_cache\_init()

|\_\_\_\_chrdev\_init()

### ###2.2.1第一步init\_rootfs()

init\_rootfs() 通过调用 register\_filesystem(&rootfs\_fs\_type)

函数来完成 rootfs 文件系统注册的，其中rootfs\_fs\_type 定义如下：

struct file\_system\_type rootfs\_fs\_type = {

name: "rootfs",

read\_super: ramfs\_read\_super,

fs\_flags: FS\_NOMOUNT|FS\_LITTER,

owner: THIS\_MODULE,

}

注册之后的 file\_systems 链表结构如下图所示：

### ###2.2.2第二步而init\_mount\_tree()

第二步而init\_mount\_tree() 是用来建立根“ / ” 目录的， 它会调用

do\_kern\_mount("rootfs", 0, "rootfs", NULL) 来挂载前面已经注册了的

rootfs 文件系统。

过程如下：

1. 调用 alloc\_vfsmnt() 函数在内存里申请了一块该类型的内存空间

（struct vfsmount \*mnt），并初始化其部分成员变量。(mnt : a vfs-internal representation of a mount point，主要记录装载信息)

2) 调用 get\_sb\_nodev（） 函数在内存中分配一个超级块结构 (struct

super\_block) sb，并初始化其部分成员变量，将成员 s\_instances 插入到

rootfs 文件系统类型结构中的 fs\_supers 指向的双向链表中。

(sb:描述文件系统信息，例如块大小、块组数，文件系统版本号、上次mount 的时间等等，参考前面ext2 文件系统的描述。)

3) 通过 rootfs 文件系统中的 read\_super 函数指针调用

ramfs\_read\_super() 函数。还记得当初注册rootfs 文件系统时，其成员

read\_super 指针指向了 ramfs\_read\_super() 函数，参见上图(读取super

block 信息填充其内容，参考前面ext2 的描述)

4) ramfs\_read\_super() 函数调用 ramfs\_get\_inode() 在内存中分配了一

个 inode 结构 (struct inode) inode，并初始化其部分成员变量，其中比较重要的有 i\_op、i\_fop 和 i\_sb。

inode->i\_op = &ramfs\_dir\_inode\_operations;

inode->i\_fop = &dcache\_dir\_ops;

inode->i\_sb = sb;

(一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类

型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等，也就是ls -l 命令看到的那些信息，这些信息存在inode 中而不是数据块中。每个文件都有一个inode，一个块组中的所有 inode 组成了inode 表。)

这使得将来通过文件系统调用对 VFS 发起的文件操作等指令将被 rootfs

文件系统中相应的函数接口所接管。

5) ramfs\_read\_super() 函数在分配和初始化了 inode 结构之后，会调用

d\_alloc\_root() 函数来为 VFS 的目录树建立起关键的根目录 (struct

dentry)dentry，并将 dentry 中的 d\_sb 指针指向 sb，d\_inode 指针指向

inode。(这里得到该文件系统的根，dentry directory entry)

(对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l 命令看到的其它信息都保存在该文件的inode 中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。

例如“vim 目录“可以看到其内容)

6) 将 mnt 中的 mnt\_sb 指针指向 sb，mnt\_root 和 mnt\_mountpoint 指针

指向 dentry，而 mnt\_parent 指针则指向自身。

结果建立了根目录如下：



2.6 内核中，调用过程如下：

init\_mount\_tree()(fs/namespace.c)

|\_\_\_\_do\_kern\_mount("rootfs",0,"rootfs",0)(fs/super.c)

|\_\_\_\_vfs\_kern\_mount("rootfs")(fs/super.c)

|\_\_\_mnt=alloc\_vfsmnt("rootfs")(分配mnt 结构)

|\_\_\_get\_sb(fs\_type,...,ramfs\_fill\_super,mnt)

| (就是调用rootfs\_get\_sb(),get\_sb\_nodev(),

| ramfs\_fill\_super()分配sb 块)

|\_\_\_inode=ramfs\_get\_inode(sb,...)(分配inode)

|\_\_\_root=d\_alloc\_root(inode)

(分配root 结构，填充'/'目录)

1. 问题：在上面的图中，如何建立一个目录，如何装载(mount)一个新的文件系统？

## ##2.3文件系统和进程的联系



# #3Kernel 启动过程和文件系统

Kernel 启动过程中，初始化硬件和各种数据结构，并加载文件系统，进入用户态，运行文件系统中的程序，这个过程简述如下。Exec 函数族(execve 为其中一种)提供了一个在进程中启动另外一个程序执行的方法。它可以根据指定的文件名或目录找到可执行文件，并用它来取代原先调用进程的属性包括数据段，代码段和堆栈段等。在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。这里在执行execve 之前，进程1 还是共享着内核线程0 资源属性的内核线程。执行execve 后(即执行了用户空间的init 程序)，此时，进程1 就拥有了自己的地址空间等属性，成为一个普通进程。

Fork

Init()--->call----->/etc/inittab------> /etc/rcS -------->login ----🡪run shell