

**<<Linux Ext2文件系统>>**

**实现方案改进**



项目成员

SY1506418 丁贵强

SY1506417 王涵仲

SY1521101 曹卫青

SY1506407 王新晨

北京航空航天大学

2016-4-20

## **0我们组具体实现什么**

通过近2周的源码分析，我们组发现，改进是比较困难的，就目前而言也是不现实的。所以我们组接下来的工作安排如下：

1. 能够大致了解VFS的通用文件模型；
2. 了解VFS中4种对象的数据结构，主要字段的含义，以及针对本字段都有哪些常见操作。
3. 在EXT2文件系统中，找到具体的的实现。
4. 实现上层用户常见的一些命令（其实就是应用程序，只不过是命令行执行而已）：像mkdir，rmdir等等。

1、2基本完成了。

## 1 **项目当前进展**

## 1.1 VFS的文件模型

VFS的关键是，根据不同的文件系统抽象出了一个通用的文件模型。每个特定的文件系统都要把物理操作与通用文件模型对应起来。通用文件模型由4种数据对象组成。

### 1.1.1文件对象

文件对象存储一个打开的文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在。文件对象用struct file结构描述：

struct file{

struct list\_head f\_list;

struct dentry \* f\_dentry;/\* 指向与文件对象关联的dentry对象 \*/

struct vfsmount\* f\_vfsmnt;

struct file\_operations\* f\_op; /\* 文件对象的操作集合 \*/

atomic\_t f\_count; /\* 引用计数 \*/

unsigned int f\_flags; /\* 使用open()时设定的标志 \*/

mode\_t f\_mode; /\* 进程访问模式 \*/

int f\_error;

loff\_t f\_pos; /\* 对文件读写操作的当前位置　\*/

........

};

### 1.1.2 inode对象

inode对象存储某个文件的管理信息，通常对应磁盘文件系统的文件控制块。文件在文件系统内有唯一的inode号。struct inode结构的部分成员如下所述：

struct inode{

struct hlist\_node i\_hash; /\* 用于hash表的指针 \*/

struct list\_head i\_list; /\* 根据inode状态链入相关链表 \*/

struct list\_head i\_sb\_list; /\* 链入超级快的inode链表 \*/

struct list\_head i\_dentry; /\* inode的dentry对象链表 \*/

unsigned long i\_ino; /\* inode号 \*/

atomic\_t i\_count; /\* 使用计数 \*/

umode\_t i\_mode; /\* 表示文件类型及权限 \*/

unsigned int i\_nlink; /\* 硬链接数 \*/

uid\_t i\_uid; /\* 文件拥有者用户id \*/

gid\_t i\_gid; /\* 用户所在组的id \*/

kdev\_t i\_rdev; /\* 所在设备的设备号 \*/??? // dev\_t

loff\_t i\_size; /\* 以字节为单位的文件大小 \*/

struct timespec i\_atime; /\* 最后访问时间 \*/

struct timespec i\_mtime; /\* 最后修改(modify)时间 \*/

struct timespec i\_ctime; /\* 文件创建时间 \*/

.......

struct inode\_operations \*i\_op; / \* inode对象操作集合 \*/

struct file\_operations \*i\_fop; / \* 文件对象对象操作集合 \*/

struct super\_block \*i\_sb; / \* 所属的超级快 \*/

struct file\_lock \*i\_flock

struct address\_space \*i\_mapping;

.............

};

### 1.1.3 dentry对象

inode对象主要是描述目录项及相关联的inode信息。struct dentry 结构描述如下：

struct dentry {

.....

struct inode \* d\_inode; / \* 该dentry对象所属的inode \*/

struct dentry \*d\_parent; /\* 父目录 \*/

struct qstr d\_name; /\* 文件名 及附属信息 \*/

.....

struct dentry\_operations \*d\_op;

....

};

### 1.1.4超级块对象

struct super\_block{

struct list\_head s\_list; / \* 将所有的超级块链接起来 \*/

kdev\_t \*s\_dev; /\* 所在设备号 \*/

unsigned long s\_blocksize; /\* 该文件系统磁盘块的大小 \*/

.....

struct file\_system\_type \*s\_type; /\* 所属的文件系统类型 \*/

struct super\_operations \*s\_op;

....

};

图1描述了各种对象之间的关系。inode对象对应于一个文件， 该文件一般在磁盘上，inode对象与文件之间的关系是一对一的。而一个真正的文件可能有多个文件名，比如硬链接（Hard Link）,因此dentry对象与inode对象之间是多对一的关系。不同的进程可能打开同一个文件，但却不能用同一个文件对象来描述。因为操作标志、文件的操作位置等均可能不同。因而文件对象与dentry对象之间是多对一的关系。

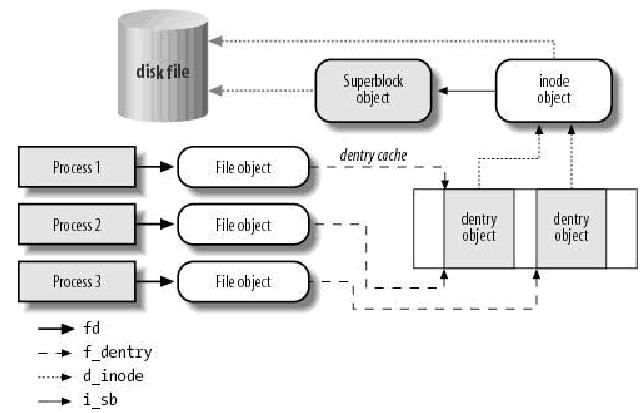


图1

为了加快文件的查找速度，内核引入了缓冲机制。最近使用的dentry对象放在dentry cache(简称dcache)中。最近使用的inode对象放在inode cache中。在cache中的对象又以Hash表的形式将它们组织起来。

## 1.2 文件系统的注册与安装

### 1.2.1 文件系统的注册

每一种文件系统使用前必须先注册一个file\_system\_type对象，其成员如下：

struct file\_system\_type{

const char \*name; /\* 文件系统类型名 \*/

struct super\_block \*(\*get\_sb)(struct file\_system\_type \*,int,const char \*,void \*);

struct file\_system\_type \*next;

......

}；

get\_sb是一个函数指针，负责读取文件系统的超级块对象。每个文件系统都有自己的实现函数。如EXT2文件系统的具体项是ext2\_get\_sb().所有的文件系统类型通过next成员链接起来。

### 1.2.2 文件系统的安装

用户经常使用mount命令把一个分区安装到某个目录下。内核与之对应的系统调用是sys\_mount()。使用该函数一个重要步骤是：根据文件系统类型得到相应的file\_system\_type对象，再取得该分区的超级块对象。

### 1.3 各种对象的操作接口

### 1.3.1面向对象的接口

前面已经提到，超级块对象、inode对象、文件对象、dentry对象构成了VFS框架。但它们又是如何与底层的物理文件系统打交道的呢？又是如何区分不同的文件系统的呢？Linux采用面向对象的思想，每个对象都有一套函数操作集合，不同文件系统的对象其操作也就不同。VFS只提供接口（操作种类集合）。下面以超级块对象为例来说明。

超级块对象有一个struct super\_operations \*s\_op成员，struct super\_operations定义如下：

struct super\_operations{

struct inode \*(\*alloc\_inode)(struct super\_block \*sb);

void (\* destroy\_inode)(struct inode \*);

void (\* read\_inode)(struct inode \*);

void (\* dirty\_inode)(struct inode \*);

int(\* write\_inode)(struct inode \*);

void (\* put\_inode)(struct inode \*);

void (\* drop\_inode)(struct inode \*);

void (\* delete\_inode)(struct inode \*);

void (\* put\_super)(struct super\_block\*);

void (\* write\_super)(struct super\_block\*);

.......

};

当第一次安装（mount）一个EXT2分区时会调用ext2\_get\_sb(),该函数有一条语句”sb->s\_op = &ext2\_sops”。ext2\_sops变量的成员如下：

static struct super\_operations ext2\_sops ={

.alloc\_inode = ext2\_alloc\_inode,

.destroy\_inode = ext2\_destroy\_inode,

.write\_inode = ext2\_write\_inode,

.read\_inode = ext2\_read\_inode,

.delete\_inode = ext2\_delete\_inode,

.put\_super= ext2\_put\_super,

.write\_super = ext2\_write\_super,

.........

};

read\_inode等函数指针初始化ext2\_read\_inode（）之类特定于EXT2文件系统的函数入口，而当第一次安装（mount）一个FAT分区时会调用fat\_read\_super(),该函数有一条语句“sb->s\_op = &fat\_sops”。fat\_sops变量的成员值如下：

static struct super\_operations fat\_sops = {

.alloc\_inode = fat\_alloc\_inode,

.destroy\_inode = fat\_destroy\_inode,

.write\_inode = fat\_write\_inode,

.delete\_inode = fat\_delete\_inode,

.put\_super= fat\_put\_super,

.......

};

从面向对象的角度来描述上面的概念，可以认为super\_operations就是一个抽象类，它只提供接口但并没有实现这些接口。而ext2\_sops,fat\_sops则是具体类，实现相应的接口。与超级块对象类似，每个inode对象包含一个struct inode\_operatons \*i\_op,每个文件对象包含一个struct file\_operations \*f\_op,每个dentry对象包含一个struct dentry\_operatons \*d\_op。每个对象的具体操作取决于该对象所属的文件系统类型。下面介绍各个主要接口。

### 1.3.2 struct super\_operations

alloc\_inode ----------分配一个inode对象。

read\_inode ----------该函数从已mount的文件系统读入一个inode信息。该inode对象的inode号事先已被初始化。

dirty\_inode -----------但inode标记为脏时调用。

write\_inode ----------把inode写入磁盘时使用该函数。

put\_inode ----------从inode cache 移去inode对象时调用该函数。

destroy\_inode ----------注销一个inode对象。

drop\_inode ---------当inode->i\_count变为0时调用。

delete\_inode ----------删除inode时调用。

put\_super----------卸载时，VFS释放超级块时调用。

write\_super ----------希望把VFS超级块写入磁盘时调用。

### 1.3.3 struct inode\_operations

struct inode\_operations{

int (\*create)(struct inode \*,struct dentry \*,int,struct nameidata \*);

struct dentry \* (\*lookup)(struct inode \*,struct dentry \*,struct nameidata \*);

int (\*link)(struct dentry \*,struct inode \*,struct dentry \*);

int (\*unlink)(struct inode \*,struct dentry \*);

int (\*symlink)(struct inode \*,struct dentry \*,const char \*);

int (\*mkdir)(struct inode \*,struct dentry \*,int);

int (\*rmdir)(struct inode \*,struct dentry \*);

.......

};

注意目录和文件在VFS中都用inode对象表示。而目录项则由dentry对象表示。下面介绍前2个函数，以便有助于对整体有所了解。

1. create -----第1个参数必须是目录型文件对应的inode.create()在该目录下创建一个文件，文件名事先被置于第2个参数中。第3个参数是文件的创建类型。
2. lookup-----第一个参数必须是目录型文件对应的inode,lookup()在该目录下查找一个文件，文件名事先被置入第2个参数dentry中。查找到的信息填入dentry中并返回。

### 1.3.4 struct file\_operations

struct file\_operations{

struct module \*owner;

loff\_t (\*llseek)(struct file \*,loff\_t,int );

ssize\_t(\*read)(struct file \*,char \_\_user \*,size\_t,loff\_t);

ssize\_t(\*aio\_ead)(struct kiocb\*,char \_\_user \*,size\_t,loff\_t);

ssize\_t(\*write)(struct file \*,const char \_\_user \*,size\_t,loff\_t \*);

ssize\_t(\*aio\_write)(struct kiocb\*,char \_\_user \*,size\_t,loff\_t);

int (\*readdir)(struct file \*,void \*, filldir\_t);

......

};

这些操作与系统调用接口非常类似。

### 1.3.5 struct dentry\_operations

struct dentry\_operations{

int (\*d\_revalidate)(struct dentry \*,struct nameidata \*);

int (\*d\_hash)(struct dentry \*,struct qstr\*);

int (\*d\_compare)(struct dentry \*,struct qstr\*,struct qstr\*);

};

1. d\_revalidate -----检查dentry cache中的dentry是否最新的。
2. d\_hash-------计算文件名的Hash值，第一个参数是父目录，第2个参数包含了预设的文件名。
3. d\_compare------比较一个目录下的2个文件名是否相等。

### 1.4 EXT2文件系统

EXT2文件系统是Linux的原生文件系统之一，它支持传统UNIX文件的语义及一些高级特性，在性能和健壮性方面都表现不错。

### 1.4.1EXT2在磁盘上的物理布局

EXT2分区的第一个磁盘块用于 引导，其余的部分被分成组，见图2。所有的组大小

组n

引导块

......

组0

超级块

inode表

数据块

inode位图

数据块位图

组描述符

一个块 n个块 一个块 一个块 n个块 n个块

图2 EXT2分区的物理布局

相同且顺序存放，所以由组的序号可以确定组在磁盘上的位置。

每个组由如下6个部分组成：（1）文件系统的超级块；（2）所有组的描述符；（3）数据块的位图；（4）inode位图；（5）inode表；（6）数据块。

早期版本的 EXT2文件系统每个组都有一份文件系统超级块和所有组描述信息的副本，正常情况下，内核只使用第0组的超级块和所有组的描述信息，并在合适时机让各个组的副本一致。当组0的副本遭到破坏时，便可根据其他组的副本恢复，从而加强文件系统的可靠性。当文件系统很大时，将浪费大量空间备份超级块和组描述符，EXT2文件系统随后的版本支持稀疏超级（Sparse Super）特性，当组号是0、1或3、5、7三个数的幂时，该组才需要备份超级块和组描述符。

而块的位图、inode表、inode位图、数据块则专属于组。块位图占用一个块，位图的每一位顺序对应组中的一个块，0表示块可用，1表示已被用。如果块的大小为1KB，则每个组的大小为8196块。所有的文件及目录都需要用inode结构表示。inode结构大小固定，依次存放在inode表中。inode位图用来表示对应的inode表的空间是否已被占用。

### 1.4.2主要的数据结构及基本操作

1. 超级块

stuct ext2\_super\_block{ // le: little endian

\_\_le32 s\_inodes\_count; /\* inode的总数 \*/

\_\_le32 s\_blocks\_count; /\* 块的总数 \*/

\_\_le32 s\_r\_blocks\_count; /\* 保留块的总数 \*/

\_\_le32 s\_free\_blocks\_count; /\* 空闲块的总数 \*/

\_\_le32 s\_free\_inodes\_count; /\* 空闲inode的总数 \*/

\_\_le32 s\_first\_data\_block; /\*第一个数据块 \*/

\_\_le32 s\_log\_block\_size; /\*块的大小 \*/

\_\_le32 s\_log\_frag\_size; /\* 碎片的大小 \*/

\_\_le32 s\_blocks\_per\_group; /\* 每组的块数 \*/

\_\_le32 s\_frags\_per\_group; /\* 每组的碎片数 \*/

\_\_le32 s\_inodes\_per\_group; /\* 每组的inode数 \*/

\_\_le32 s\_mtime; /\* 安装的时间 \*/

\_\_le32 s\_wtime; /\* 写的时间 \*/

\_\_le32 s\_mnt\_count; /\* 安装的次数 \*/

...........

};

1. 组描述符

每个组都有自己的描述符，内核用结构ext2\_group\_desc描述。下面列出它的主要成员：

struct ext2\_group\_desc{

\_\_le32 bg\_block\_bitmap; /\* 本组数据块位图所在的块号 \*/

\_\_le32 bg\_inode\_bitmap; /\* 本组inode位图所在的块号 \*/

\_\_le32 bg\_inode\_table; /\* 本组inode表的起始块号 \*/

\_\_le16 bg\_free\_blocks\_count; /\* 组中空闲块的数目 \*/

\_\_le16 bg\_free\_inodes\_count; /\* 组中空闲inode的数目 \*/

\_\_le16 bg\_used\_dirs\_count; /\* 组中目录的数目 \*/

\_\_le16 bg\_pad; /\* 未用 \*/

\_\_le32 bg\_reserved[3]; /\* 未用 \*/

};

1. inode

inode就是EXT2文件系统的文件控制块，EXT2文件系统所有的inode大小均为128B,每个inode在inode表存的位置就是该inode隐含的inode号。内核用结构ext2\_inode描述inode, 主要成员如下：

struct ext2\_inode{

\_\_le16 i\_mode;

\_\_le16 i\_uid; /\*拥有者的用户ID \*/

\_\_le32 i\_size; /\* 文件大小 \*/

\_\_le32 i\_atime; /\*最近一次访问时间 \*/

\_\_le32 i\_ctime; /\* 创建时间 \*/

\_\_le32 i\_mtime; /\*最近一次修改时间 \*/

\_\_le32 i\_dtime; /\*最近一次不用时间 \*/

\_\_le16 i\_gid; /\* 文件的组id\*/

\_\_le16 i\_ links\_count; /\* 链接数 \*/

\_\_le32 i\_blocks; /\*分配给该文件的磁盘块的数目 \*/

\_\_le32 i\_flag; /\* 文件标志\*/

......

\_\_le32 i\_block[EXT2\_N\_BLOCKS]; /\* 指向磁盘块的指针 \*/

......

};

i\_mode包含了文件的类型信息和文件的访问权限。文件的类型有常规文件、目录、字符设备、块设备等。

i\_blocks[EXT2\_N\_BLOCKS] 一般用于放置文件的数据所在的磁盘块编号，EXT2\_N\_BLOCKS的默认值是15。

i\_blocks[15]数组主要是支持常规文件，因为数据在磁盘上并不一定连续，需要保存各个磁盘块号。它的前12项可看成一级指针，直接存放文件数据所在的磁盘块号。数组的第13项是一个二级指针，指向的磁盘块并不包含文件的数据，而是一系列的一级指针，这些一级指针才用来指向磁盘块。假设磁盘块的大小是1KB，每个磁盘块号占4字节，则第13项可表达文件的大小是256KB。而数据的第14、15项分别是三级指针和四级指针，访问数据方式可由第13项类推。这种方法保证了对大量的小文件访问效率高，同时又支持大文件。

如图3.

i\_block数组 间接块

0

1

间接块

二级间接块

11

10

12

13

14

图3 EXT2文件数据块组织示意图

设备文件用ext2\_inode就足以包含所有信息，不需要另外的数据块，而常规文件、目录有所不同，一般情况下总需要额外的磁盘块存储数据。

目录的数据块包含了所有属于这个目录的文件信息。他的数据项类型是ext2\_dir\_entry\_2结构，用来描述这个目录的文件。原型声明如下：

struct ext2\_dir\_entry\_2{

\_\_le32 inode; /\* inode号\*/

\_\_le32 rec\_len; /\* 本项所占的长度\*/

\_\_u8 name\_len; /\* 文件名的长度 \*/

char name[EXT2\_NAME\_LEN]; / \* 文件名 \*/

};

目录的各项在数据块中依次放置，可以看成一个变长数组，每一项的起始地址偏移rec\_len后即是下一项的起始地址。当删除一项时，被删除的rec\_len加入前一项的rec\_len即可。如果要增加一项，则顺序查找已有项，如果某项的剩余空间足以容纳新项，则利用该空间放置，找不到时需要分配新块。

1. 一些操作

如果知道文件的inode号，如何找到该文件信息呢？因为每组磁盘块的inode数目固定。所以很容易计算出该文件属于哪个组并且得到inode在组中inode表的下标。继而可得到ext2\_inode信息。

再介绍如何查找一个文件，以/root/temp.c为例。由于根目录的inode号总为2，因此可以得到根目录的ext2\_inode信息，再从i\_block指向的数据块查找是否有ext2\_dir\_entry\_2项的名字等于root,如果有，就能得到了root目录的inode号。重复上述过程，则可以判定是否存在/root/temp.c。

最后，介绍如何读取文件某个位置的数据。由于给的的位置是相对于而不是相对于磁盘的，因此需要根据该位置计算出它在i\_block中的下标，这样才能得到在磁盘上的位置。当然，因为多级指针的介入而稍显复杂。

### 1.4.3磁盘块的分配和释放

磁盘块的释放主要工作是修改块位图和涉及块的统计变量。

磁盘块的分配比较复杂。EXT2采用了预分配策略，当申请一个块时，系统尽量预分配8个连续块，这样下一次申请时，分配算法先查找是否存在预分配块，如果有，分配就可。如果不行，则试图在附件32个块的范围内分配。若还不行，则在本组内向前找8个连续空闲的块。若还不满足，则任何空间的块均可以被分配。若还不满足，则到其他的组中去寻找。这样就能有效地减少文件的访问时间。

### **1.5主要文件系统的系统调用处理流程**

文件系统的操作接口open(),read(),write()等看似简单，但内部实现比较复杂，目前没有深入下去。VFS一般都将上层的系统调用变成sys\_\*\*\*\*（）开头的函数等等。