

**<<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

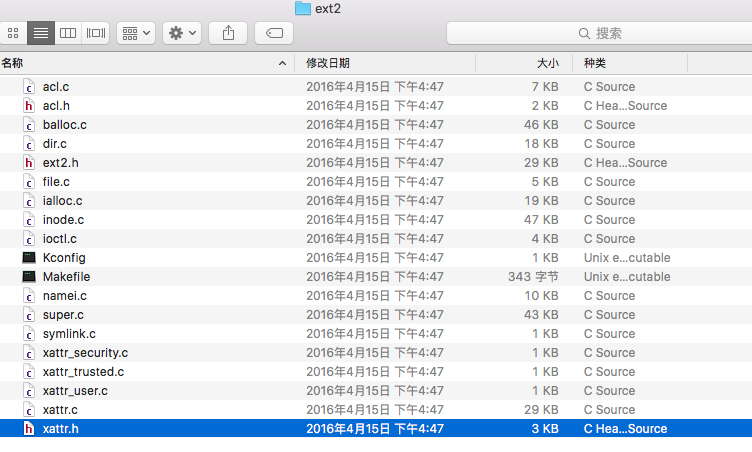
2016-4-20

## **0本周工作内容**

1. 在EXT2文件系统中，找到具体的的实现。例如：列出常见函数，分析功能。
2. 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 **项目当前进展**

## 1.1 EXT2代码目录结构



下面简要分析这些文件：

xattr.c 和 xattr.h

On-disk format of extended attributes for the ext2 filesystem.

/\*

\* ext2\_xattr\_get()

\*

\* Copy an extended attribute into the buffer

\* provided, or compute the buffer size required.

\* Buffer is NULL to compute the size of the buffer required.

\*

\* Returns a negative error number on failure, or the number of bytes

\* used / required on success.

\*/

int

ext2\_xattr\_get(struct inode \*inode, int name\_index, const char \*name,void \*buffer, size\_t buffer\_size);

/\*

\* ext2\_xattr\_list()：获取属性名字列表

\*

\* Copy a list of attribute names into the buffer

\* provided, or compute the buffer size required.

\* Buffer is NULL to compute the size of the buffer required.

\*

\* Returns a negative error number on failure, or the number of bytes used / required on success.

\*/

static int

ext2\_xattr\_list(struct dentry \*dentry, char \*buffer, size\_t buffer\_size);

/\*

\* Inode operation listxattr()

\*

\* d\_inode(dentry)->i\_mutex: don't care

\*/

ssize\_t

ext2\_listxattr(struct dentry \*dentry, char \*buffer, size\_t size);

/\*

\* If the EXT2\_FEATURE\_COMPAT\_EXT\_ATTR feature of this file system is not set, set it.

\*/

static void ext2\_xattr\_update\_super\_block(struct super\_block \*sb)

/\*

\* ext2\_xattr\_set()

\*

\* Create, replace or remove an extended attribute for this inode. Value is NULL to remove an existing extended attribute, and non-NULL to either replace an existing extended attribute, or create a new extended attribute. The flags XATTR\_REPLACE and XATTR\_CREATE

\* specify that an extended attribute must exist and must not exist

\* previous to the call, respectively.

\*

\* Returns 0, or a negative error number on failure.

\*/

int

ext2\_xattr\_set(struct inode \*inode, int name\_index, const char \*name,const void \*value, size\_t value\_len, int flags)

/\*

\* Second half of ext2\_xattr\_set(): Update the file system.

\*/

static int

ext2\_xattr\_set2(struct inode \*inode, struct buffer\_head \*old\_bh,

struct ext2\_xattr\_header \*header);

/\*

\* ext2\_xattr\_delete\_inode()

\*

\* Free extended attribute resources associated with this inode. This

\* is called immediately before an inode is freed.

\*/

void

ext2\_xattr\_delete\_inode(struct inode \*inode);

/\*

\* ext2\_xattr\_put\_super()

\*

\* This is called when a file system is unmounted.

\*/

void

ext2\_xattr\_put\_super(struct super\_block \*sb);

/\*

\* ext2\_xattr\_cache\_insert()

\*

\* Create a new entry in the extended attribute cache, and insert

\* it unless such an entry is already in the cache.

\*

\* Returns 0, or a negative error number on failure.

\*/

static int

ext2\_xattr\_cache\_insert(struct buffer\_head \*bh);

/\*

\* ext2\_xattr\_cmp()

\*

\* Compare two extended attribute blocks for equality.

\*

\* Returns 0 if the blocks are equal, 1 if they differ, and

\* a negative error number on errors.

\*/

static int

ext2\_xattr\_cmp(struct ext2\_xattr\_header \*header1,

struct ext2\_xattr\_header \*header2)

/\*

\* ext2\_xattr\_cache\_find()

\*

\* Find an identical extended attribute block.

\*

\* Returns a locked buffer head to the block found, or NULL if such

\* a block was not found or an error occurred.

\*/

static struct buffer\_head \*

ext2\_xattr\_cache\_find(struct inode \*inode, struct ext2\_xattr\_header \*header)

/\*

\* ext2\_xattr\_hash\_entry()

\*

\* Compute the hash of an extended attribute.

\*/

static inline void ext2\_xattr\_hash\_entry(struct ext2\_xattr\_header \*header,struct ext2\_xattr\_entry \*entry)

/\*

\* ext2\_xattr\_rehash()

\*

\* Re-compute the extended attribute hash value after an entry has changed.

\*/

static void ext2\_xattr\_rehash(struct ext2\_xattr\_header \*header,

struct ext2\_xattr\_entry \*entry)；

**xattr\_user.c:**

Handler for extended user attributes.

xattr\_trusted.c:

Handler for trusted extended attributes.

const struct xattr\_handler ext2\_xattr\_trusted\_handler = {

.prefix = XATTR\_TRUSTED\_PREFIX,

.list = ext2\_xattr\_trusted\_list,

.get = ext2\_xattr\_trusted\_get,

.set = ext2\_xattr\_trusted\_set,

};

**xattr\_security.c:**

Handler for storing security labels as extended attributes.

**symlink.c:**

Only fast symlinks left here - the rest is done by generic code.

ext2 symlink handling code

const struct inode\_operations ext2\_symlink\_inode\_operations = {

.readlink = generic\_readlink,

.get\_link = page\_get\_link,

.setattr = ext2\_setattr,

#ifdef CONFIG\_EXT2\_FS\_XATTR

.setxattr = generic\_setxattr,

.getxattr = generic\_getxattr,

.listxattr = ext2\_listxattr,

.removexattr = generic\_removexattr,

#endif

};

const struct inode\_operations ext2\_fast\_symlink\_inode\_operations = {

.readlink = generic\_readlink,

.get\_link = simple\_get\_link,

.setattr = ext2\_setattr,

#ifdef CONFIG\_EXT2\_FS\_XATTR

.setxattr = generic\_setxattr,

.getxattr = generic\_getxattr,

.listxattr = ext2\_listxattr,

.removexattr = generic\_removexattr,

#endif

};

**super.c:**

Big-endian to little-endian byte-swapping/bitmaps

**namei.c:**

rewrite to pagecache.页面缓存

\* Stuff here is basically a glue between the VFS and generic UNIXish

\* filesystem that keeps everything in pagecache. All knowledge of the

\* directory layout is in fs/ext2/dir.c - it turned out to be easily separatable

\* and it's easier to debug that way. In principle we might want to

\* generalize that a bit and turn it into a library. Or not.

\*

\* The only non-static object here is ext2\_dir\_inode\_operations.

\*

\* TODO: get rid of kmap() use, add readahead.

**Makefile:**

#

**Makefile for the linux ext2-filesystem routines.**

#

obj-$(CONFIG\_EXT2\_FS) += ext2.o

ext2-y := balloc.o dir.o file.o ialloc.o inode.o \ ioctl.o namei.o super.o symlink.o

ext2-$(CONFIG\_EXT2\_FS\_XATTR) += xattr.o xattr\_user.o xattr\_trusted.o ext2-$(CONFIG\_EXT2\_FS\_POSIX\_ACL) += acl.o ext2-$(CONFIG\_EXT2\_FS\_SECURITY) += xattr\_security.o

编译产生的目标文件（14个）：

acl.o,ballo.o,dir.o,file.o,ialloc.o,inode.o,ioctl.o,namei.o,super.o,symlink.o,xattr\_security.o,xattr\_trusted.o,xattr\_user.o,xattr.o.

**Kconfig:**

配置文件

**ioctl.c**：

设备驱动程序中对设备的I/O通道进行管理的函数。

long ext2\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)；

long ext2\_compat\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)；

**inode.c：**

/\*

\* Test whether an inode is a fast symlink.

\*/

static inline int ext2\_inode\_is\_fast\_symlink(struct inode \*inode)；

/\*

\* Called at the last iput() if i\_nlink is zero.

\*/

void ext2\_evict\_inode(struct inode \* inode)；

/\*

\* Allocation strategy is simple: if we have to allocate something, we will

\* have to go the whole way to leaf. So let's do it before attaching anything

\* to tree, set linkage between the newborn blocks, write them if sync is

\* required, recheck the path, free and repeat if check fails, otherwise

\* set the last missing link (that will protect us from any truncate-generated

\* removals - all blocks on the path are immune now) and possibly force the

\* write on the parent block.

\* That has a nice additional property: no special recovery from the failed

\* allocations is needed - we simply release blocks and do not touch anything

\* reachable from inode.

\*

\* `handle' can be NULL if create == 0.

\*

\* return > 0, # of blocks mapped or allocated.

\* return = 0, if plain lookup failed.

\* return < 0, error case.

\*/

static int ext2\_get\_blocks(struct inode \*inode,

sector\_t iblock, unsigned long maxblocks,

struct buffer\_head \*bh\_result,

int create)；

const struct address\_space\_operations ext2\_aops = {

.readpage = ext2\_readpage,

.readpages = ext2\_readpages,

.writepage = ext2\_writepage,

.write\_begin = ext2\_write\_begin,

.write\_end = ext2\_write\_end,

.bmap = ext2\_bmap,

.direct\_IO = ext2\_direct\_IO,

.writepages = ext2\_writepages,

.migratepage = buffer\_migrate\_page,

.is\_partially\_uptodate = block\_is\_partially\_uptodate,

.error\_remove\_page = generic\_error\_remove\_page,

};

const struct address\_space\_operations ext2\_nobh\_aops = {

.readpage = ext2\_readpage,

.readpages = ext2\_readpages,

.writepage = ext2\_nobh\_writepage,

.write\_begin = ext2\_nobh\_write\_begin,

.write\_end = nobh\_write\_end,

.bmap = ext2\_bmap,

.direct\_IO = ext2\_direct\_IO,

.writepages = ext2\_writepages,

.migratepage = buffer\_migrate\_page,

.error\_remove\_page = generic\_error\_remove\_page,

};

**ialloc.c:**

inode创建和销毁的一些函数。

/\*

\* ialloc.c contains the inodes allocation and deallocation routines

\*/

/\*

\* The free inodes are managed by bitmaps. A file system contains several blocks groups. Each group contains 1 bitmap block for blocks, 1 bitmap block for inodes, N blocks for the inode table and data blocks.

\*

\* The file system contains group descriptors which are located after the super block. Each descriptor contains the number of the bitmap block and the free blocks count in the block.

\*/

/\*

\* Read the inode allocation bitmap for a given block\_group, reading

\* into the specified slot in the superblock's bitmap cache.

\*

\* Return buffer\_head of bitmap on success or NULL.

\*/

static struct buffer\_head \*

read\_inode\_bitmap(struct super\_block \* sb, unsigned long block\_group);

static void ext2\_release\_inode(struct super\_block \*sb, int group, int dir);

static void ext2\_preread\_i node(struct inode \*inode);

static int find\_group\_dir(struct super\_block \*sb, struct inode \*parent);

**file.c：**

ext2 fs regular file handling primitives

static int ext2\_dax\_fault(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf)；

static const struct vm\_operations\_struct ext2\_dax\_vm\_ops = {

.fault = ext2\_dax\_fault,

.pmd\_fault = ext2\_dax\_pmd\_fault,

.page\_mkwrite = ext2\_dax\_fault,

.pfn\_mkwrite = ext2\_dax\_pfn\_mkwrite,

};

static int ext2\_file\_mmap(struct file \*file, struct vm\_area\_struct \*vma)

{

if (!IS\_DAX(file\_inode(file)))

return generic\_file\_mmap(file, vma);

file\_accessed(file);

vma->vm\_ops = &ext2\_dax\_vm\_ops;

vma->vm\_flags |= VM\_MIXEDMAP | VM\_HUGEPAGE;

return 0;

}

/\*

\* We have mostly NULL's here: the current defaults are ok for

\* the ext2 filesystem.

\*/

const struct file\_operations ext2\_file\_operations = {

.llseek = generic\_file\_llseek,

.read\_iter = generic\_file\_read\_iter,

.write\_iter = generic\_file\_write\_iter,

.unlocked\_ioctl = ext2\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = ext2\_compat\_ioctl,

#endif

.mmap = ext2\_file\_mmap,

.open = dquot\_file\_open,

.release = ext2\_release\_file,

.fsync = ext2\_fsync,

.splice\_read = generic\_file\_splice\_read,

.splice\_write = iter\_file\_splice\_write,

};

const struct inode\_operations ext2\_file\_inode\_operations = {

#ifdef CONFIG\_EXT2\_FS\_XATTR

.setxattr = generic\_setxattr,

.getxattr = generic\_getxattr,

.listxattr = ext2\_listxattr,

.removexattr = generic\_removexattr,

#endif

.setattr = ext2\_setattr,

.get\_acl = ext2\_get\_acl,

.set\_acl = ext2\_set\_acl,

.fiemap = ext2\_fiemap,

};

**ext2.h：**

列举出ext2里面其他源文件（14个）用到的类型定义，一些宏定义，特殊inode号（坏的块inode号1，root inode号2），最大最小块的大小，页的大小，等等。

/\*

\* second extended-fs super-block data in memory

\*/

struct ext2\_sb\_info {

unsigned long s\_frag\_size; /\* Size of a fragment in bytes \*/

unsigned long s\_frags\_per\_block;/\* Number of fragments per block \*/

unsigned long s\_inodes\_per\_block;/\* Number of inodes per block \*/

unsigned long s\_frags\_per\_group;/\* Number of fragments in a group \*/

unsigned long s\_blocks\_per\_group;/\* Number of blocks in a group \*/

unsigned long s\_inodes\_per\_group;/\* Number of inodes in a group \*/

unsigned long s\_itb\_per\_group; /\* Number of inode table blocks per group \*/

unsigned long s\_gdb\_count; /\* Number of group descriptor blocks \*/

unsigned long s\_desc\_per\_block; /\* Number of group descriptors per block \*/

unsigned long s\_groups\_count; /\* Number of groups in the fs \*/

unsigned long s\_overhead\_last; /\* Last calculated overhead \*/

unsigned long s\_blocks\_last; /\* Last seen block count \*/

struct buffer\_head \* s\_sbh; /\* Buffer containing the super block \*/

struct ext2\_super\_block \* s\_es; /\* Pointer to the super block in the buffer \*/

struct buffer\_head \*\* s\_group\_desc;

unsigned long s\_mount\_opt;

unsigned long s\_sb\_block;

kuid\_t s\_resuid;

kgid\_t s\_resgid;

unsigned short s\_mount\_state;

unsigned short s\_pad;

int s\_addr\_per\_block\_bits;

int s\_desc\_per\_block\_bits;

int s\_inode\_size;

int s\_first\_ino;

spinlock\_t s\_next\_gen\_lock;

u32 s\_next\_generation;

unsigned long s\_dir\_count;

u8 \*s\_debts;

struct percpu\_counter s\_freeblocks\_counter;

struct percpu\_counter s\_freeinodes\_counter;

struct percpu\_counter s\_dirs\_counter;

struct blockgroup\_lock \*s\_blockgroup\_lock;

/\* root of the per fs reservation window tree \*/

spinlock\_t s\_rsv\_window\_lock;

struct rb\_root s\_rsv\_window\_root;

struct ext2\_reserve\_window\_node s\_rsv\_window\_head;

/\*

\* s\_lock protects against concurrent modifications of s\_mount\_state,

\* s\_blocks\_last, s\_overhead\_last and the content of superblock's

\* buffer pointed to by sbi->s\_es.

\*

\* Note: It is used in ext2\_show\_options() to provide a consistent view

\* of the mount options.

\*/

spinlock\_t s\_lock;

};

**dir.c：**

目录操作有关函数，goto等

ext2 directory handling functions

static void ext2\_check\_page(struct page \*page, int quiet)

static struct page \* ext2\_get\_page(struct inode \*dir, unsigned long n,int quiet);

/\*

\* routine to check that the specified directory is empty (for rmdir)

\*/

int ext2\_empty\_dir (struct inode \* inode);

#define S\_SHIFT 12

static unsigned char ext2\_type\_by\_mode[S\_IFMT >> S\_SHIFT] = {

[S\_IFREG >> S\_SHIFT] = EXT2\_FT\_REG\_FILE,// 正常文件

[S\_IFDIR >> S\_SHIFT] = EXT2\_FT\_DIR,// 目录

[S\_IFCHR >> S\_SHIFT] = EXT2\_FT\_CHRDEV,// 字符设备

[S\_IFBLK >> S\_SHIFT] = EXT2\_FT\_BLKDEV,// 块设备

[S\_IFIFO >> S\_SHIFT] = EXT2\_FT\_FIFO,// 管道

[S\_IFSOCK >> S\_SHIFT] = EXT2\_FT\_SOCK,// sock

[S\_IFLNK >> S\_SHIFT] = EXT2\_FT\_SYMLINK,// 符号链接文件

}; 7种

**balloc.c：**

块的分配和销毁函数

balloc.c contains the blocks allocation and deallocation routines

**acl.c和acl.h：**

访问控制列表

/\*\*

ext2\_acl\_entry: ext2文件系统的acl结构体，遵循posix标准，和POSIX标准的一样代表某一个具体的访问控制规则

e\_tag：表示实体标识，共有 6 种 e\_tag ，分别是 ACL\_USER\_OBJ, ACL\_USER, ACL\_GROUP\_OBJ, ACL\_GROUP, ACL\_MASK, ACL\_OTHER

分别代表文件主，文件其他用户，文件组主，文件其他组，文件掩码，其他人。

e\_perm: 表示权限。

e\_id: 标识符，只有e\_tag为ACL\_USER\_OBJ和ACL\_GROUP\_OBJ时有效

\*\*/

typedef struct {

\_\_le16 e\_tag;

\_\_le16 e\_perm;

\_\_le32 e\_id;

} ext2\_acl\_entry;

/\*\*ext2文件系统的简短的结构体，和posix标准的区别是没有了e\_id字段\*\*/

typedef struct {

\_\_le16 e\_tag;

\_\_le16 e\_perm;

} ext2\_acl\_entry\_short;

/\*\*ext2的头部，仅仅有一个版本号\*\*/

typedef struct {

\_\_le32 a\_version;

} ext2\_acl\_header;

/\*\*函数功能：读取保存在物理磁盘上的ACL，转换为内存形式的ACL

参数： value：ACL的首地址

size: ACL的大小

返回值：指向内存形式的ACL的指针

\*\*/

static struct posix\_acl \*

ext2\_acl\_from\_disk(const void \*value, size\_t size)；

/\*\*

函数功能：将内存形式的ACL转换为文件系统的acl结构体形式

参数： acl：指向内存形式的ACL的地址

size: 指针形式返回posix\_acl的大小

返回值：void\*，指向ext2文件系统的acl结构体，或错误类型

\*\*/

static void \*

ext2\_acl\_to\_disk(const struct posix\_acl \*acl, size\_t \*size)；

/\*\*

函数功能：从索引节点获取内存形式的acl，同时增加引用计数

参数： inode：索引节点

type: 标识从哪里读取，如果是ACL\_TYPE\_ACCESS标识从ext2\_inode\_info->i\_acl，如果是ACL\_TYPE\_DEFAULT标识从ext2\_inode\_info->i\_default\_acl读取

返回值：返回指向内存形式的ACL的指针

\*\*/

struct posix\_acl \*

ext2\_get\_acl(struct inode \*inode, int type)；

/\*\*

函数功能：用内存形式的acl设置索引节点的acl

参数： inode：索引节点

acl: 指向内存形式的ACL

type: 标识设置哪一个acl

返回值：如果错误，返回错误码

\*\*/

int

ext2\_set\_acl(struct inode \*inode, struct posix\_acl \*acl, int type)

### 1.2我们将要实现的命令