# Ext2 file system disk layout analysis

李磊

目录

[Ext2 file system disk layout analysis 1](#_Toc387826970)

[1 Summery 2](#_Toc387826971)

[2 Super Block 3](#_Toc387826972)

[2.1 Super Block成员分析 3](#_Toc387826973)

[2.2 ext2 Super Block的解析 5](#_Toc387826974)

[3 Block Group Descriptor 7](#_Toc387826975)

[3.1 Block Group Descriptor成员分析 7](#_Toc387826976)

[3.2 Block Group Descriptor 的解析 8](#_Toc387826977)

[4 Block Bitmap 10](#_Toc387826978)

[5 Inode Bitmap 11](#_Toc387826979)

[6 Inode Table 11](#_Toc387826980)

[6.1 Inode 成员分析 11](#_Toc387826981)

[7 Locating an Inode 15](#_Toc387826982)

[8 Directory Structure 15](#_Toc387826983)

[8.1 Linked List Directory 15](#_Toc387826984)

[8.2 Analysis a root directory 16](#_Toc387826985)

[8.3 Indexed Directory Format 22](#_Toc387826986)

## 1 Summery

The Second Extended File System(ext2)文件系统是Linux系统中的标准文件系统，是通过对Minix的文件系统进行扩展而得到的。以下是Ext2文件系统的基本概念:

Block:

被格式化成Ext2文件系统的分区，磁盘，块设备，文件都被分成一个一个的块，这样的一个块就称作Block。注意这里的Block和某些器件的Block不是同一概念，比如MMC Flash, 物理上一个Block为512Byte, 这和文件系统中提及的Block是不同的

Block group:

在Ext2文件系统中， 多个Block组成一个Block group, 为的就是减少磁盘碎片。

Inode:

Inode结构中包含了指向数据块的指针，用于寻找数据块

Super Block:

Super Block中包含了文件系统中所有的配置信息，这些配置信息包含Inode 总数，Block总数，可用Inode总数，可用Block总数等。

Ext2文件系统的磁盘布局如图1-1所示

块组n

…..

块组49

……

块组0

……

普通块

第0块

引导块

6

5

4

6

5

4

2

3

1

7

7

1块 1块 n块 n块

1块 n块 n块 1块 1块 n块 n块

备份块

6

5

4

3

2

1

7

1块 n块 n块 1块 1块 n块 n块

1：超级块 2：组描述符表 3：reverse GDT 4：数据块位图 5：inode位图 6：inode表 7：数据块

图1-1 Ext2文件系统的磁盘布局

## 2 Super Block

### 2.1 Super Block成员分析

在linux kernel 2.6.18中， Super Block的结构如List 2-1

struct ext2\_super\_block {

\_\_le32 s\_inodes\_count; /\* Inodes count \*/

\_\_le32 s\_blocks\_count; /\* Blocks count \*/

\_\_le32 s\_r\_blocks\_count; /\* Reserved blocks count \*/

\_\_le32 s\_free\_blocks\_count; /\* Free blocks count \*/

\_\_le32 s\_free\_inodes\_count; /\* Free inodes count \*/

\_\_le32 s\_first\_data\_block; /\* First Data Block \*/

\_\_le32 s\_log\_block\_size; /\* Block size \*/

\_\_le32 s\_log\_frag\_size; /\* Fragment size \*/

\_\_le32 s\_blocks\_per\_group; /\* # Blocks per group \*/

\_\_le32 s\_frags\_per\_group; /\* # Fragments per group \*/

\_\_le32 s\_inodes\_per\_group; /\* # Inodes per group \*/

\_\_le32 s\_mtime; /\* Mount time \*/

\_\_le32 s\_wtime; /\* Write time \*/

\_\_le16 s\_mnt\_count; /\* Mount count \*/

\_\_le16 s\_max\_mnt\_count; /\* Maximal mount count \*/

\_\_le16 s\_magic; /\* Magic signature \*/

\_\_le16 s\_state; /\* File system state \*/

\_\_le16 s\_errors; /\* Behaviour when detecting errors \*/

\_\_le16 s\_minor\_rev\_level; /\* minor revision level \*/

\_\_le32 s\_lastcheck; /\* time of last check \*/

\_\_le32 s\_checkinterval; /\* max. time between checks \*/

\_\_le32 s\_creator\_os; /\* OS \*/

\_\_le32 s\_rev\_level; /\* Revision level \*/

\_\_le16 s\_def\_resuid; /\* Default uid for reserved blocks \*/

\_\_le16 s\_def\_resgid; /\* Default gid for reserved blocks \*/

\_\_le32 s\_first\_ino; /\* First non-reserved inode \*/

\_\_le16 s\_inode\_size; /\* size of inode structure \*/

\_\_le16 s\_block\_group\_nr; /\* block group # of this superblock \*/

\_\_le32 s\_feature\_compat; /\* compatible feature set \*/

\_\_le32 s\_feature\_incompat; /\* incompatible feature set \*/

\_\_le32 s\_feature\_ro\_compat; /\* readonly-compatible feature set \*/

\_\_u8 s\_uuid[16]; /\* 128-bit uuid for volume \*/

char s\_volume\_name[16]; /\* volume name \*/

char s\_last\_mounted[64]; /\* directory where last mounted \*/

\_\_le32 s\_algorithm\_usage\_bitmap; /\* For compression \*/

\_\_u8 s\_prealloc\_blocks; /\* Nr of blocks to try to preallocate\*/

\_\_u8 s\_prealloc\_dir\_blocks; /\* Nr to preallocate for dirs \*/

\_\_u16 s\_padding1;

\_\_u8 s\_journal\_uuid[16]; /\* uuid of journal superblock \*/

\_\_u32 s\_journal\_inum; /\* inode number of journal file \*/

\_\_u32 s\_journal\_dev; /\* device number of journal file \*/

\_\_u32 s\_last\_orphan; /\* start of list of inodes to delete \*/

\_\_u32 s\_hash\_seed[4]; /\* HTREE hash seed \*/

\_\_u8 s\_def\_hash\_version; /\* Default hash version to use \*/

\_\_u8 s\_reserved\_char\_pad;

\_\_u16 s\_reserved\_word\_pad;

\_\_le32 s\_default\_mount\_opts;

\_\_le32 s\_first\_meta\_bg; /\* First metablock block group \*/

\_\_u32 s\_reserved[190]; /\* Padding to the end of the block \*/

};

List 2-1

下面将对以上的重要成员做出说明：

s\_inodes\_count：Ext2文件系统中inode的总数

s\_blocks\_count：Ext2文件系统中block的总数

s\_r\_blocks\_count：为超级用户预留的block数量，一般为block总数的%5

s\_free\_blocks\_count：空余的block数量，这里包含了s\_r\_blocks\_count

s\_free\_inodes\_count：空余的inode数量

s\_first\_data\_block : 由于superblock的位置在开始的1KB处，当block size为1KB时，此值为1，当b block size大于1KB时，此值为0

s\_log\_block\_size：block的大小可由公式block size = 1024 << s\_log\_block\_size;得出

s\_blocks\_per\_group：一个block group有多少block, 此值可由 block size \* 8的出，文件系统中的最后一个组的大小可能比其他组小，也就是block 个数小于其他组。

s\_inodes\_per\_group : 一个block group有多少inode，所有的组中的inode个数都是一样的。

s\_mtime : 最后一次挂载时间，此值是从1970年1月1日0时0分0秒开始到最后一次挂载的时间差值，以秒为单位

s\_wtime：最后一次写入的时间

s\_mnt\_count：被挂载的次数

s\_max\_mnt\_count：最大被挂载的次数

s\_magic：magic number, Ext2为0Xef53

s\_state : file system的状态，一般有2中值，EXT2\_VALID\_FS表示unmounted cleanly, EXT2\_ERROR\_FS表示有错误，如果有错误就需要fsck

s\_errors : 如果出现错误应该进行的操作，

Constant Name Value Description

EXT2\_ERRORS\_CONTINUE 1 continue as if nothing happen

EXT2\_ERRORS\_RO 2 remount read-only

EXT2\_ERRORS\_PANIC 3 cause a kernel panic

s\_creator\_os : 此文件系统由哪种操作系统创建， 0表示linux

s\_def\_resuid : 保留的block可由此值的UID用户使用

s\_def\_resgid : 保留的block可由此值的GID用户使用

s\_first\_ino : 能够被常规文件使用的第一个inode, 在ext2 revision 0中，此值固定，为11，在ext2 revision 1中，此值可以为任意值

s\_inode\_size：inode的大小

s\_block\_group\_nr：此superblock位于哪个block group

s\_feature\_compat : 兼容特性， 不管系统是否支持此特性，挂载此文件系统都不会有破坏文件系统的风险

s\_feature\_incompat：不兼容特性，如果系统不支持此特性，就应该拒绝挂载此文件系统

s\_feature\_ro\_compat：只读兼容特性，如果系统不支持此特性，此文件系统被挂载时属性只能为只读

s\_uuid：16Byte的uuid

s\_volume\_name : 基本无用

s\_last\_mounted：基本无用

s\_prealloc\_blocks：预分配的块数

s\_prealloc\_dir\_blocks：为目录预分配的块数，只有EXT2\_COMPAT\_PREALLOC被置才起效

### 2.2 ext2 Super Block的解析

首先来做个20 MB的空文件

dd if=/dev/zero of=ext2.img bs=20MB count=1

接着将其格式化成EXT2文件系统格式

mkfs.ext2 ext2.img

图1-2是通过WINHEX显示ext2.img中的16进制

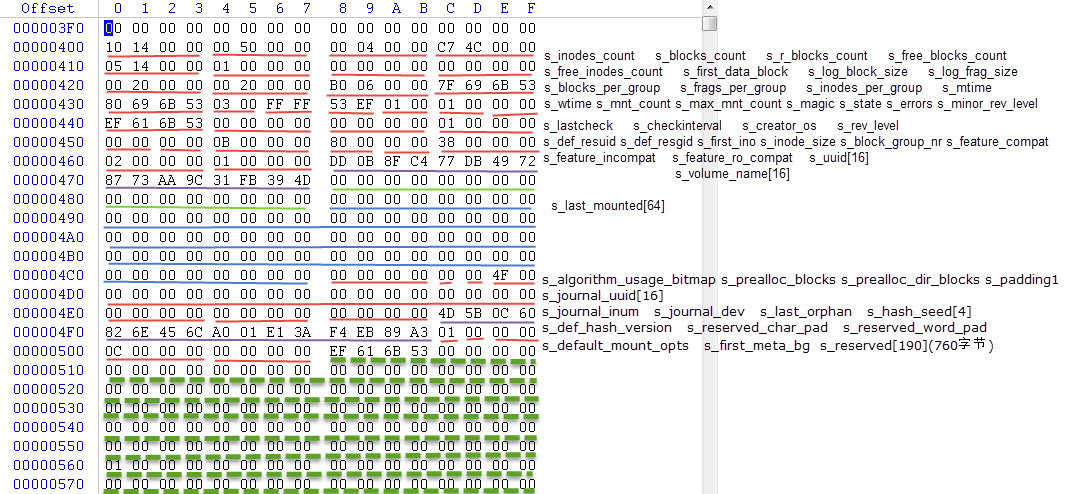


图2-2 ext2.img中的16进制

Super block各成员的值如List 2-2

s\_inodes\_count = 0x1410;

s\_blocks\_count = 0x5000;

s\_r\_blocks\_count = 0x400;

s\_free\_blocks\_count = 0x4cc7;

s\_free\_inodes\_count = 0x1405;

s\_first\_data\_block = 1;

s\_log\_block\_size = 0;

s\_log\_frag\_size = 0;

s\_blocks\_per\_group = 0x2000;

s\_frags\_per\_group = 0x2000;

s\_inodes\_per\_group = 0x6b0;

s\_mtime = 0x536b697f;

s\_wtime = 0x536b6980;

s\_mnt\_count = 0x3;

s\_max\_mnt\_count= 0xff;

s\_magic = 0xef53;

s\_state = 0x1;

s\_errors 0x1;

s\_minor\_rev\_level = 0x0;

s\_lastcheck = 0x536b61ef;

s\_checkinterval = 0x0;

s\_creator\_os = 0x0;

s\_rev\_level = 0x1;

s\_def\_resuid = 0;

s\_def\_resgid = 0;

s\_first\_ino = 0xb;

s\_inode\_size = 0x80;

s\_block\_group\_nr = 0x0;

s\_feature\_compat = 0x38;

s\_feature\_incompat = 0x2;

s\_feature\_ro\_compat = 0x1;

s\_uuid[16] = "dd0b8fc477db49728773aa9c31fb39";

s\_volume\_name[16] = 0x0;

s\_last\_mounted[64] = 0;

s\_algorithm\_usage\_bitmap = 0x0;

s\_prealloc\_blocks = 0x0;

s\_prealloc\_dir\_blocks = 0x0;

s\_padding1 = 0x4f;

s\_journal\_uuid[16] = 0x0;

s\_journal\_inum = 0x0;

s\_journal\_dev = 0x0;

s\_last\_orphan = 0x0;

s\_hash\_seed[4] = 0x600c5b4d, 0x6c456e82, 0x3ae101a0, 0x3a89ebf4;

s\_def\_hash\_version = 0x1;

s\_reserved\_char\_pad = 0x0;

s\_reserved\_word\_pad = 0x0;

s\_default\_mount\_opts = 0x0;

s\_first\_meta\_bg = 0x0;

s\_reserved[190];

List 2-2

其中，比较重要的有

s\_log\_block\_size = 0; 这表示block size为1KB

s\_inode\_size = 0x80; 表示inode size为128Bye

s\_blocks\_per\_group = 0x2000; 这个可以通过block size计算得出，block size为1KB, 一个group则有8K个block

s\_blocks\_count = 0x5000; 由于block size为1KB, 整个文件大小为20MB, 故，block共有20480个

s\_r\_blocks\_count = 0x400; 这个值默认是block count的%5，也就是1024

s\_inodes\_count = 0x1410;在ext2文件系统为8MB的倍数时， 每4K个block分配一个inode,如果不是8MB的倍数时，需要进一步究

s\_free\_inodes\_count = 0x1405; 一个空的文件系统中默认会有个lost+found目录，占用一个inode,

此成员的值可通过s\_inodes\_count - (s\_first\_ino - 1) – 1得到

s\_inodes\_per\_group = 0x6b0; 每个组的inode个数一样，这个值可由s\_inodes\_count / group个数得到

## 3 Block Group Descriptor

Block group指定了inode bitmap, block bitmap, inode table的位置，以及block, inode 的空闲情况。它的位置紧跟Super Block。Block group descriptor可能占用多个block, 这是根据group数量而定。

### 3.1 Block Group Descriptor成员分析

struct ext2\_group\_desc

{

\_\_le32 bg\_block\_bitmap; /\* Blocks bitmap block \*/

\_\_le32 bg\_inode\_bitmap; /\* Inodes bitmap block \*/

\_\_le32 bg\_inode\_table; /\* Inodes table block \*/

\_\_le16 bg\_free\_blocks\_count; /\* Free blocks count \*/

\_\_le16 bg\_free\_inodes\_count; /\* Free inodes count \*/

\_\_le16 bg\_used\_dirs\_count; /\* Directories count \*/

\_\_le16 bg\_pad;

\_\_le32 bg\_reserved[3];

}

List 3-1

下面将对各成员进行说明：

bg\_block\_bitmap： 组内block bitmap的位置，也就是所在block的number, block bitmap占一个block

bg\_inode\_bitmap : 组内inode bitmap的位置,也就是所在block的number, inode bitmap占一个block

bg\_inode\_table : 组内inode table的位置

bg\_free\_blocks\_count : 组内有多少空闲block

bg\_free\_inodes\_count : 组内有多少空闲inode

bg\_used\_dirs\_count : 组内有多少inode分配给目录使用

bg\_pad：用于4字节对齐

bg\_reserved：保留

### 3.2 Block Group Descriptor 的解析

通过dump查看Block Group Descriptor

[rocklee@localhost ~]$ dumpe2fs ext2.img

dumpe2fs 1.39 (29-May-2006)

Filesystem volume name: <none>

Last mounted on: <not available>

Filesystem UUID: dd0b8fc4-77db-4972-8773-aa9c31fb394d

Filesystem magic number: 0xEF53

Filesystem revision #: 1 (dynamic)

Filesystem features: ext\_attr resize\_inode dir\_index filetype sparse\_super

Default mount options: user\_xattr acl

Filesystem state: clean

Errors behavior: Continue

Filesystem OS type: Linux

Inode count: 5136

Block count: 20480

Reserved block count: 1024

Free blocks: 19655

Free inodes: 5125

First block: 1

Block size: 1024

Fragment size: 1024

Reserved GDT blocks: 79

Blocks per group: 8192

Fragments per group: 8192

Inodes per group: 1712

Inode blocks per group: 214

Filesystem created: Thu May 8 18:52:31 2014

Last mount time: Thu May 8 19:24:47 2014

Last write time: Thu May 8 19:24:48 2014

Mount count: 3

Maximum mount count: -1

Last checked: Thu May 8 18:52:31 2014

Check interval: 0 (<none>)

Reserved blocks uid: 0 (user root)

Reserved blocks gid: 0 (group root)

First inode: 11

Inode size: 128

Default directory hash: half\_md4

Directory Hash Seed: 4d5b0c60-826e-456c-a001-e13af4eb89a3

Group 0: (Blocks 1-8192)

Primary superblock at 1, Group descriptors at 2-2

Reserved GDT blocks at 3-81

Block bitmap at 82 (+81), Inode bitmap at 83 (+82)

Inode table at 84-297 (+83)

7881 free blocks, 1701 free inodes, 2 directories

Free blocks: 312-8192

Free inodes: 12-1712

Group 1: (Blocks 8193-16384)

Backup superblock at 8193, Group descriptors at 8194-8194

Reserved GDT blocks at 8195-8273

Block bitmap at 8274 (+81), Inode bitmap at 8275 (+82)

Inode table at 8276-8489 (+83)

7895 free blocks, 1712 free inodes, 0 directories

Free blocks: 8490-16384

Free inodes: 1713-3424

Group 2: (Blocks 16385-20479)

Block bitmap at 16385 (+0), Inode bitmap at 16386 (+1)

Inode table at 16387-16600 (+2)

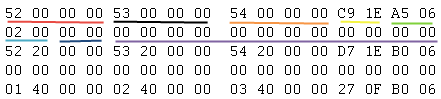
3879 free blocks, 1712 free inodes, 0 directories

Free blocks: 16601-20479

Free inodes: 3425-5136

List 3-2

在ext2 revision 0中，每个Group 均有Super Block和Group Descriptor的备份。上述信息可以看出目前的ext2的revision为1，Super Block 不是每个group都有的，只有group number为0,1及3,5,7次幂的group才有。下面是WINHEX dump出来的16进制数据，由于boot section占一个1KB, Super Block占一个block, 也就是1KB, 故 group 0的GDT从0x800开始。



下面是Group 0的Block Group Descriptor各成员的值

struct ext2\_group\_desc

{

\_\_le32 bg\_block\_bitmap = 0x52

\_\_le32 bg\_inode\_bitmap = 0x53

\_\_le32 bg\_inode\_table = 0x54

\_\_le16 bg\_free\_blocks\_count = 0x1ec9

\_\_le16 bg\_free\_inodes\_count = 0x06a5

\_\_le16 bg\_used\_dirs\_count = 0x2

\_\_le16 bg\_pad;

\_\_le32 bg\_reserved[3];

};

List 3-3

List 3-2中显示Group 0中包含了两个目录，实际上是3个，分别是“.”, “..”及”lost + found”, 其中的“.”, “..”共用第二个inode,”lost+found”占用了第11个inode, 故free inode从第12个开始 。

笔者还发现一个规律，当文件系统的大小为8MB的整数倍，基本上是每4KB分配一个inode。当文件系统的大小不为8MB的整数倍时，每个GROUP的inode数目为8，GROUP的个数，文件系统的大小/4K这3者的最小公倍数，此推断有待进一步研究。

在List 3-2及Super Block的s\_inodes\_per\_group中可以看出，每个Block Group都有1712个inode, inode的个数 \* inode的大小刚好和block size对齐。bg\_inode\_table = 84，而inode table共占用1712 / (1KB / 128) = 124个block, 故Inode table at 84-297 。但为什么Free block不是从block 298开始的，需要进一步研究。

除了Group 0, 其他Group的free block均紧跟Inode table。

## 4 Block Bitmap

Block 位图， 用于标记Block的使用情况，占用一个Block。位图的每一位表示了当前Group内block是否被使用，1表示对应的block在使用中，0表示对应的block未使用。Block Bitmap的位置可由相应的block group descriptor中的bg\_block\_bitmap得到,这里是82 \* 1KB = 82KB。

以Group 0为例，List 3-2中可以看出Free blocks: 312-8192，从图4-1可以看出最后一个使用中的 Block为8 \* 32 + 8 \* 7 - 1= 311，这和List 3-2中的信息完全吻合。但是要注意的是图4-1中标红的部分7F, 对应的2进制也就是0111 1111， 其中为0的那一位是最高位，也就Block 312空闲。

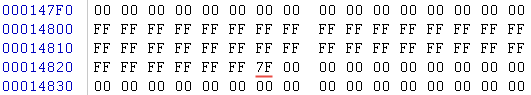
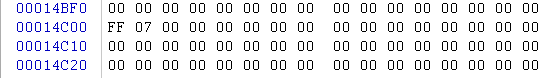


图4-1

## 5 Inode Bitmap

和Block Bitmap相似，只是Inode Bitmap 用于标记Inode的使用情况。Inode Bitmap的位置可由相应的group descriptor 中的bg\_inode\_bitmap得到， Group 0 的Inode Bitmap如图5-1所示。

图5-1

可以看出此Group 中前11个inode被使用。

## 6 Inode Table

Inode Table 用于记录每一个目录，文件，符号链接，或者其他一些特殊的文件；这些文件的位置，大小，类型以及访问权限都保存在inode中。文件名只会保存在目录中，而不会保存在inode里。

Inode编号从1开始，在revision 0中 Inode table中保留前11个inode，在revision 1及后续版本中保留inode的个数是根据Super Block的s\_first\_ino得到的。保留的inode做如下用途：

Constant Name Value Description

EXT2\_BAD\_INO 1 bad blocks inode

EXT2\_ROOT\_INO 2 root directory inode

EXT2\_ACL\_IDX\_INO 3 ACL index inode (deprecated?)

EXT2\_ACL\_DATA\_INO 4 ACL data inode (deprecated?)

EXT2\_BOOT\_LOADER\_INO 5 bootoader inode

EXT2\_UNDEL\_DIR\_INO 6 undelete directory inode

### 6.1 Inode 成员分析

struct ext2\_inode {

\_\_le16 i\_mode; /\* File mode \*/

\_\_le16 i\_uid; /\* Low 16 bits of Owner Uid \*/

\_\_le32 i\_size; /\* Size in bytes \*/

\_\_le32 i\_atime; /\* Access time \*/

\_\_le32 i\_ctime; /\* Creation time \*/

\_\_le32 i\_mtime; /\* Modification time \*/

\_\_le32 i\_dtime; /\* Deletion Time \*/

\_\_le16 i\_gid; /\* Low 16 bits of Group Id \*/

\_\_le16 i\_links\_count; /\* Links count \*/

\_\_le32 i\_blocks; /\* Blocks count \*/

\_\_le32 i\_flags; /\* File flags \*/

union {

struct {

\_\_le32 l\_i\_reserved1;

} linux1;

struct {

\_\_le32 h\_i\_translator;

} hurd1;

struct {

\_\_le32 m\_i\_reserved1;

} masix1;

} osd1; /\* OS dependent 1 \*/

\_\_le32 i\_block[EXT2\_N\_BLOCKS];/\* Pointers to blocks \*/

\_\_le32 i\_generation; /\* File version (for NFS) \*/

\_\_le32 i\_file\_acl; /\* File ACL \*/

\_\_le32 i\_dir\_acl; /\* Directory ACL \*/

\_\_le32 i\_faddr; /\* Fragment address \*/

union {

struct {

\_\_u8 l\_i\_frag; /\* Fragment number \*/

\_\_u8 l\_i\_fsize; /\* Fragment size \*/

\_\_u16 i\_pad1;

\_\_le16 l\_i\_uid\_high; /\* these 2 fields \*/

\_\_le16 l\_i\_gid\_high; /\* were reserved2[0] \*/

\_\_u32 l\_i\_reserved2;

} linux2;

struct {

\_\_u8 h\_i\_frag; /\* Fragment number \*/

\_\_u8 h\_i\_fsize; /\* Fragment size \*/

\_\_le16 h\_i\_mode\_high;

\_\_le16 h\_i\_uid\_high;

\_\_le16 h\_i\_gid\_high;

\_\_le32 h\_i\_author;

} hurd2;

struct {

\_\_u8 m\_i\_frag; /\* Fragment number \*/

\_\_u8 m\_i\_fsize; /\* Fragment size \*/

\_\_u16 m\_pad1;

\_\_u32 m\_i\_reserved2[2];

} masix2;

} osd2; /\* OS dependent 2 \*/

};

List 6-1

i\_mode : 描述了文件类型及访问权限，List 6-2 为可能的值，这些值可以任意组合。

Constant Value Description

-- file format --

EXT2\_S\_IFSOCK 0xC000 socket

EXT2\_S\_IFLNK 0xA000 symbolic link

EXT2\_S\_IFREG 0x8000 regular file

EXT2\_S\_IFBLK 0x6000 block device

EXT2\_S\_IFDIR 0x4000 directory

EXT2\_S\_IFCHR 0x2000 character device

EXT2\_S\_IFIFO 0x1000 fifo

-- process execution user/group override --

EXT2\_S\_ISUID 0x0800 Set process User ID

EXT2\_S\_ISGID 0x0400 Set process Group ID

EXT2\_S\_ISVTX 0x0200 sticky bit

-- access rights --

EXT2\_S\_IRUSR 0x0100 user read

EXT2\_S\_IWUSR 0x0080 user write

EXT2\_S\_IXUSR 0x0040 user execute

EXT2\_S\_IRGRP 0x0020 group read

EXT2\_S\_IWGRP 0x0010 group write

EXT2\_S\_IXGRP 0x0008 group execute

EXT2\_S\_IROTH 0x0004 others read

EXT2\_S\_IWOTH 0x0002 others write

EXT2\_S\_IXOTH 0x0001 others execute

List 6-2

i\_uid : 文件的所有者的低16位

i\_size : 在ext2 revision 0中，此值表示文件占用的字节数。在revision 1及更高版本的ext2文件系统，此值代表普通文件的大小的低32位，高32位保存在i\_dir\_acl中。

i\_atime : inode的最近访问时间

i\_ctime : inode的创建时间

i\_mtime : inode的最后修改时间

i\_dtime : inode被删除的时间

i\_gid : group id的低16位

i\_links\_count : 文件被连接的数目，当此文件有一个硬链接时，此值将增加1。当此值为0时，表示当前inode没有指向数据块

i\_blocks : 此成员表示此inode指向的数据占有的block数目，此block不是文件系统的block, 而是512字节为单位的block

i\_flag : 此成员表示以何种方式访问此inode的数据。List 6-3有详细说明

Constant Name Value Description

EXT2\_SECRM\_FL 0x00000001 secure deletion

EXT2\_UNRM\_FL 0x00000002 record for undelete

EXT2\_COMPR\_FL 0x00000004 compressed file

EXT2\_SYNC\_FL 0x00000008 synchronous updates

EXT2\_IMMUTABLE\_FL 0x00000010 immutable file

EXT2\_APPEND\_FL 0x00000020 append only

EXT2\_NODUMP\_FL 0x00000040 do not dump/delete file

EXT2\_NOATIME\_FL 0x00000080 do not update .i\_atime

-- Reserved for compression usage --

EXT2\_DIRTY\_FL 0x00000100 Dirty (modified)

EXT2\_COMPRBLK\_FL 0x00000200 compressed blocks

EXT2\_NOCOMPR\_FL 0x00000400 access raw compressed data

EXT2\_ECOMPR\_FL 0x00000800 compression error

-- End of compression flags --

EXT2\_BTREE\_FL 0x00001000 b-tree format directory

EXT2\_INDEX\_FL 0x00001000 hash indexed directory

EXT2\_IMAGIC\_FL 0x00002000 AFS directory

EXT3\_JOURNAL\_DATA\_FL 0x00004000 journal file data

EXT2\_RESERVED\_FL 0x80000000 reserved for ext2 library

List 6-3

osd1 : os dependant, 操作系统相关。Os可以为Hurd/Linux/Masix

i\_block:数据块索引，每个inode内保存了15个数据块索引, 其中12个是直接的索引，1个间接索引，1个2级间接索引，1个3级间接索引。间接索引中保存的是一个包含直接索引的block的索引。一个索引占4字节，如果Block size 为1KB, 一个Block可以包含256个索引，故一个文件的占用的block中的13 – 256号会通过间接索引得到。同理，2级间接索引中保存的是包含1级间接索引的block的索引，3级间接 索引中保存的是包含2级间接索引的block的索引。如果i\_block为0，表示此inode没有关联任何数据块。图6-1表示了inode的i\_block和数据块之间的关系

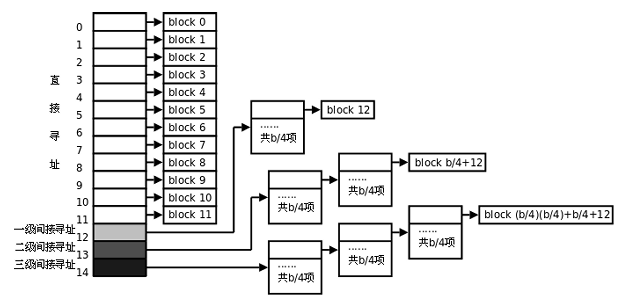


图 6-1

i\_generation：文件version, 给NFS使用。

i\_file\_acl : 包含文件acl属性的block number, 在ext2 文件系统的revision 0中，此值为0。

i\_dir\_acl：ext2文件系统revision 0中，此值为0，在revision 1, 此成员保存了64位文件大小的高32位。如果此inode表示的文件不是一个普通文件，Linux会将i\_dir\_acl设为0，理论上来说此成员用于记录包含目录acl属性的block number。

i\_faddr：此成员基本无用，fragment目前还没使用。

osd2：目前我们只关系linux，其中的l\_i\_uid\_high表示uid的高16位，l\_i\_gid\_high表示gid的高16位。

## 7 Locating an Inode

与Block一样，inode也是被编号了的，Inode number 是inode在inode table中的索引，此索引从1开始。Inode table的大小在格式化的时候就已经确定下来了，每个block group中inode table的大小相等。以下公式可以计算出inode在哪个block group中，其中的s\_inodes\_per\_group保存在super block中。

block group = (inode - 1) / s\_inodes\_per\_group

组内的inode index可以通过一下公式计算得出 , 组内的inode index从0开始

local inode index = (inode - 1) % s\_inodes\_per\_group

## 8 Directory Structure

目录保存在数据块中。inode的i\_mode如果为EXT2\_S\_IFDIR，则表示此inode指向的block为一个目录。Inode Table中的第二个Inode通常指向文件系统的根目录。

在ext2 文件系统的revision 0中，目录只能被链式存储，在revision 1及后续版本中引入了index directory的概念。Index directory向后兼容，兼容链式存储的目录。

### 8.1 Linked List Directory

在kernel 2.6.18中，ext2 文件系统的目录项有两种结构，如List 8-1所示

struct ext2\_dir\_entry {

\_\_le32 inode; /\* Inode number \*/

\_\_le16 rec\_len; /\* Directory entry length \*/

\_\_le16 name\_len; /\* Name length \*/

char name[EXT2\_NAME\_LEN]; /\* File name \*/

};

struct ext2\_dir\_entry\_2 {

\_\_le32 inode; /\* Inode number \*/

\_\_le16 rec\_len; /\* Directory entry length \*/

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

\_\_u8 file\_type;

char name[EXT2\_NAME\_LEN]; /\* File name \*/

};

List 8-1

结构ext2\_dir\_entry一般出现在ext2 revision 0中, 其中的的name\_len占用2个字节，结构中没有file\_type。但是大部分程序在处理文件名时，都要求文件名必须小于255，所以2个字节的name\_len就浪费了一个字节的大小，故在ext2 revision 0.5及以后的版本中目录项就采用struct ext2\_dir\_entry\_2，file\_name缩减到1个字节，新增file\_type成员，用于表示文件类型。下面分析struct ext2\_dir\_entry\_2成员。

inode : 目录中文件（或目录）的inode

rec\_len : record length, 即目录项的长度，此长度表示从当前的目录项的起始到下个目录项起始之间的长度。需要注意的是，目录项的长度必须以4字节对齐，并且一个目录项不能分别位于多个block上。如果在一个block上的剩余空间没办法放下一个目录项，这个目录项必须放到下一个block上。并且前一个目录项的 rec\_len需要做相应的调整。如果一个目录中的一个文件被删除了，前一个目录项必须修改rec\_len, 确保下个目录项是有有效的，如果不存在下个目录项，rec\_len必须调整到当前block的结束位置。如果此目录中的第一个文件被删除了，就需要创建一个空的目录项，调整rec\_len，保证下个目录项有效，同样，如果不存在下个目录项，rec\_len必须调整到当前block的结束位置。

name\_len : 文件名的长度，如果一个文件被重命名，并且文件名增加，超出了当前目录项容纳范围。这时候就需要重新分配目录项。新分配的目录项有可能会被分配到另一个block中。

file\_type ： 文件类型，此值必须和inode 中记录的文件类型一致，List 8-1中表示了各值的含义

Constant Name Value Description

EXT2\_FT\_UNKNOWN 0 Unknown File Type

EXT2\_FT\_REG\_FILE 1 Regular File

EXT2\_FT\_DIR 2 Directory File

EXT2\_FT\_CHRDEV 3 Character Device

EXT2\_FT\_BLKDEV 4 Block Device

EXT2\_FT\_FIFO 5 Buffer File

EXT2\_FT\_SOCK 6 Socket File

EXT2\_FT\_SYMLINK 7 Symbolic Link

List 8-1

name : 文件（或者目录）名称

### 8.2 Analysis a root directory

下面做个试验来分析root dir, 在一个ext2文件系统创建了4个文件，分别是after.c, superblock.c, vim-7.4.tar.bz2, wget-log。

[root@localhost rocklee]# mount -o loop ext2.img temp/

[root@localhost rocklee]# cd temp/

[root@localhost temp]# ll -ai

total 9677

2 drwxr-xr-x 3 root root 1024 May 13 17:53 .

28116033 drwxrwxrwx 14 rocklee rocklee 4096 May 12 19:08 ..

13 -rw-r--r-- 1 root root 1618 May 13 17:53 after.c

11 drwx------ 2 root root 12288 May 8 18:52 lost+found

14 -rw-r--r-- 1 root root 3216 May 13 17:53 superblock.c

12 -rwxr--r-- 1 root root 9843297 May 13 17:53 vim-7.4.tar.bz2

15 -rw-r--r-- 1 root root 1211 May 13 17:53 wget-log

List 8-2

接下来用debugfs,查看文件系统的根目录， 如List 8-3

[rocklee@localhost ~]$ debugfs ext2.img

debugfs 1.39 (29-May-2006)

debugfs: stat /

Inode: 2 Type: directory Mode: 0755 Flags: 0x0 Generation: 0

User: 0 Group: 0 Size: 1024

File ACL: 0 Directory ACL: 0

Links: 3 Blockcount: 2

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x5371eb7f -- Tue May 13 17:53:03 2014

atime: 0x5371eb53 -- Tue May 13 17:52:19 2014

mtime: 0x5371eb7f -- Tue May 13 17:53:03 2014

BLOCKS:

(0):298

TOTAL: 1

List 8-3

以上信息和Winhex中的信息对比

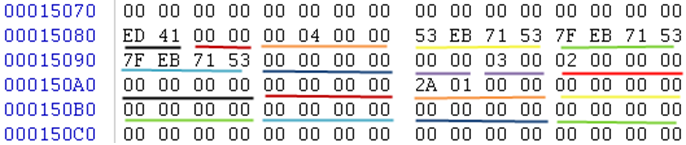


图8-4

参考图8-1对inode各个成员赋值，得:

i\_mode = 0x41ed, 这个值是EXT2\_S\_IFDIR | EXT2\_S\_IRUSR | EXT2\_S\_IWUSR | EXT2\_S\_IXUSR | EXT2\_S\_IRGRP | EXT2\_S\_IXGRP | EXT2\_S\_IROTH | EXT2\_S\_IXOTH, 也就是表示这是一个目录，其权限是755

i\_uid = 0， 由root创建

i\_size = 0x400，大小为1024

i\_atime = 0x5371eb53

i\_ctime = 0x5371eb7f

i\_mtime = 0x5371eb7f

i\_dtime = 0

i\_gid = 0，创建此目录的所有者的group id为0

i\_links\_count = 3, Links为3表示根目录有三个硬链接，分别是根目录下的.和..，以及lost+found子目录下的..。注意，虽然我们通常用/表示根目录，但是并没有名为/的硬链接，事实上，/是路径分隔符，不能在文件名中出现

i\_blocks = 2, block占用2个物理块，也就是1KB, 即一个逻辑块

i\_flags = 0

osd1 = 0

i\_block[0] = 0x12a， 第298个块，它在文件系统中的位置是298 \* 0x400 = 0x4a800

i\_block[1] = 0

以上结果和List 8-3导出来的一致。

接下来我们找到目录的数据块进行分析

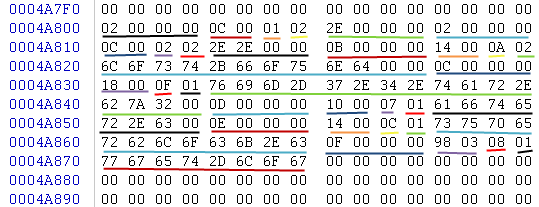


图8-5

第一个目录项：

Inode = 2

rec\_len = 12

name\_len = 1

file\_type = 0x2, 是个目录

name = 0x2e, 也就是“.”

第二个目录项：

Inode = 2

rec\_len = 12

name\_len = 2

file\_type = 0x2, 是个目录

name = 0x2e2e, 也就是“..”

第三个目录项：

Inode = 11

rec\_len = 20

name\_len = 10

file\_type = 0x2, 是个目录

name =“lost+found”

第四个目录项：

Inode = 12

rec\_len = 24

name\_len = 15

file\_type = 0x1, 是个普通文件

name =“vim-7.4.tar.bz2”

第五个目录项：

Inode = 13

rec\_len = 16

name\_len = 7

file\_type = 0x1, 是个普通文件

name =“after.c”

第六个目录项：

Inode = 14

rec\_len = 20

name\_len = 12

file\_type = 0x1, 是个普通文件

name =“superblock.c”

第七个目录项：

Inode = 15

rec\_len = 0x398

name\_len = 8

file\_type = 0x1, 是个普通文件

name =“wget-log”

注意由于第七个目录项是最后一个目录项，故其rec\_len为一个block的大小即1024 – 前6个目录项的rec\_len的总和

接下尝试着分析superblock.c这个文件是如何获取的， 根据List 3-2， Inode table at 84-297 (+83)，而Inode = 14，inode number从1开始，故此inode的地址为84 \* 1024 + （14 - 1） \* 128 = 0x15680, 此文件的inode如图8-6

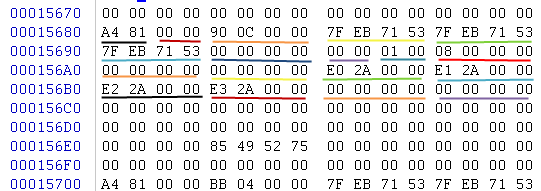


图8-6

i\_mode = 0x81a4, 表示这是一个权限为644的文件

i\_uid = 0， 由root创建

i\_size = 0xc90，大小为3126

i\_atime = 0x5371eb7f

i\_ctime = 0x5371eb7f

i\_mtime = 0x5371eb7f

i\_dtime = 0

i\_gid = 0，创建此目录的所有者的group id为0

i\_links\_count = 1

i\_blocks = 8, block占用8个物理块，也就是4KB, 即4个逻辑块

i\_flags = 0

osd1 = 0

i\_block[0] = 2ae0， 第10976个块，它在文件系统中的位置是10976 \* 0x400 = 0xab8000

i\_block[1] = 2ae1

i\_block[2] = 2ae2

i\_block[3] = 2ae3

找到data block, 实际数据如图8-7

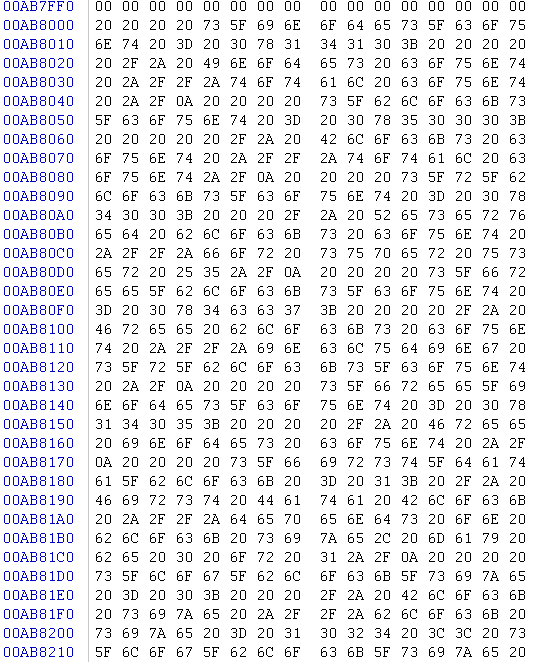


图8-7

实际字符串如图8-8



图8-8

### 8.3 Indexed Directory Format

略