# EXT4 文件系统分析

## EXT4文件系统Feature

dd if=/dev/zero of=64M.img bs=1024 count=65536

mkfs.ext4 -O ^extent,^has\_journal -L data -v 64M.img

#将image格式化,去除extent & journal属性

root@PATEO-HYG:/work1/e2fs/e2fsprogs-1.42.2/toolbox/sbin# dumpe2fs 64M.img | more

dumpe2fs 1.41.12 (17-May-2010)

Filesystem volume name: data #-L参数指定

Last mounted on: <not available>

Filesystem UUID: f8282a9a-b6ed-4351-8d75-a30c1063569f

Filesystem magic number: 0xEF53

Filesystem revision #: 1 (dynamic)

Filesystem features: ext\_attr resize\_inode dir\_index filetype flex\_bg sparse\_super huge\_file uninit\_bg dir\_nlink extra\_isize #使用dump命令查看文件系统的属性

Filesystem flags: signed\_directory\_hash

static struct feature feature\_list[] = {

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_DIR\_PREALLOC,

"dir\_prealloc" },

{ E2P\_FEATURE\_COMPAT, EXT3\_FEATURE\_COMPAT\_HAS\_JOURNAL,

"has\_journal" }, **#EXT4的日志属性**

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_IMAGIC\_INODES,

"imagic\_inodes" },

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_EXT\_ATTR,

"ext\_attr" },

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_DIR\_INDEX,

"dir\_index" },

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_RESIZE\_INODE,

"resize\_inode" },**#文件系统动态增长Reserved GDT blocks**

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_LAZY\_BG,

"lazy\_bg" },

{ E2P\_FEATURE\_COMPAT, EXT2\_FEATURE\_COMPAT\_EXCLUDE\_BITMAP,

"snapshot\_bitmap" },

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT2\_FEATURE\_RO\_COMPAT\_SPARSE\_SUPER,

"sparse\_super" },

**#减少super &块组描述符的备份，只有1,3,5,7,9,25,27…的块组才备份这些信息**

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT2\_FEATURE\_RO\_COMPAT\_LARGE\_FILE,

"large\_file" },

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_HUGE\_FILE,

"huge\_file" },

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_GDT\_CSUM,

"uninit\_bg" },

**#在格式化image时，不用全部初始化Group desc，可以节省格式化的时间**

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_GDT\_CSUM,

"uninit\_groups" },

**#在格式化image时，不用全部初始化Group desc，可以节省格式化的时间**

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_DIR\_NLINK,

"dir\_nlink" },

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_EXTRA\_ISIZE,

"extra\_isize" },

**#格外的inode数据信息，当inode数据结构为256B时使用，struct ext2\_inode\_large**

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_QUOTA,

"quota" },

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_BIGALLOC,

"bigalloc"},

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_METADATA\_CSUM,

"metadata\_csum"},

{ E2P\_FEATURE\_RO\_INCOMPAT, EXT4\_FEATURE\_RO\_COMPAT\_REPLICA,

"replica" },

{ E2P\_FEATURE\_INCOMPAT, EXT2\_FEATURE\_INCOMPAT\_COMPRESSION,

"compression" },

{ E2P\_FEATURE\_INCOMPAT, EXT2\_FEATURE\_INCOMPAT\_FILETYPE,

"filetype" },

**#目录中包含文件属性**

**struct ext2\_dir\_entry\_2 {**

**\_\_u32 inode; /\* Inode number \*/**

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

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

***\_\_u8 file\_type;***

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

**};**

{ E2P\_FEATURE\_INCOMPAT, EXT3\_FEATURE\_INCOMPAT\_RECOVER,

"needs\_recovery" },

{ E2P\_FEATURE\_INCOMPAT, EXT3\_FEATURE\_INCOMPAT\_JOURNAL\_DEV,

"journal\_dev" },

{ E2P\_FEATURE\_INCOMPAT, EXT3\_FEATURE\_INCOMPAT\_EXTENTS,

"extent" },**#ext4相比EXT3增加的extent属性**

{ E2P\_FEATURE\_INCOMPAT, EXT3\_FEATURE\_INCOMPAT\_EXTENTS,

"extents" },

{ E2P\_FEATURE\_INCOMPAT, EXT2\_FEATURE\_INCOMPAT\_META\_BG,

"meta\_bg" },

{ E2P\_FEATURE\_INCOMPAT, EXT4\_FEATURE\_INCOMPAT\_64BIT,

"64bit" }, **#块组描述符的大小为64B，EXT2 & EXT3为32B**

{ E2P\_FEATURE\_INCOMPAT, EXT4\_FEATURE\_INCOMPAT\_MMP,

"mmp" },

{ E2P\_FEATURE\_INCOMPAT, EXT4\_FEATURE\_INCOMPAT\_FLEX\_BG,

"flex\_bg"},**#Group meta-data（块位图 & inode位图可以放在image的任何地方，不用某个块组的位图放在该块组中）**

{ 0, 0, 0 },

};

该函数判断group\_block块组是否存在备份信息

int ext2fs\_bg\_has\_super(ext2\_filsys fs, int group\_block)

{

if (!(fs->super->s\_feature\_ro\_compat &

EXT2\_FEATURE\_RO\_COMPAT\_SPARSE\_SUPER))

return 1;

if (test\_root(group\_block, 3) || (test\_root(group\_block, 5)) ||

test\_root(group\_block, 7))

return 1;

return 0;

}

### 1.2 Improving ext4 feature

**Improving ext4: bigalloc, inline data, and metadata checksums**

　　http://lwn.net/Articles/469805/

**Bigalloc** 硬盘的空间越来越大，但传统的文件系统的块大小一直是4KB，这导致文件系统需要维护大量的metadata，但简单地增大块尺寸也不行——内存管理会复杂不少。但是， 文件系统自身仍然可以在底层做些变化：Bigalloc的工作机制引入了“block cluster”的概念：ext4不再分配单独的块，而是整组的分配块（即block cluster）。而这个4KB块和block cluster的映射过程完全在ext4内完成，对于内存管理子系统完全是透明的。

**Inline data** ext4的inode结构，默认长度是256字节，但其实这里面还有不少空间是空闲的。一般来说，这些空闲空间中保存了扩展属性，例如SELinux labels等。但如果不使用这些特性，这些空间就被浪费掉了。考虑到系统上有不少小文件存在，这部分空间完全可以保存数据。测试的结果表明：内核代码树节省了1%的空间，/usr目录树下节省了3%。因为这些功能需要fsck等用户空间的工具支持。到3.4内核发布之后才能在upstream内核中使用这个功能。

**metadata checksum** 一直以来，ext4缺少校验和的功能，而RAID和btrfs都具备针对数据、元数据的校验和的功能。ext4终于也可以提供这项功能了，但只针对元数据，例如superblocks, bitmaps, inodes, directory indexes, extent trees等。目前也没有计划提供针对数据的校验和功能。

特别值得一提的是，Bigalloc和inline data这两个功能，都是taobao内核团队主推的

## EXT4读文件的步骤

下面该函数展示了如何中EXT4中读取根目录下的hello.c文件的内存并打印出来

/\* try to read /hello.c ,and print content \*/

void do\_a(int argc, char \*argv[])

{

struct ext2\_super\_block \*super;

int gd\_num = 0;

int retval = 0;

super = current\_fs->super; **//得到EXT4文件系统的super block信息**

gd\_num = (int)((super->s\_blocks\_count + 1024 - 1) /

super->s\_blocks\_per\_group); **//计算文件系统的块组数**

printf("super->s\_blocks\_count:%d"

"\nsuper->s\_blocks\_per\_group:%d\n"

"gd\_num:%d\n"

"EXT2\_DESC\_PER\_BLOCK:%d\n",

super->s\_blocks\_count,

super->s\_blocks\_per\_group,

gd\_num,

EXT2\_DESC\_PER\_BLOCK(super)); //\*\*\*compute group desc size(32B/64B)

**//读取文件系统的块组描述符,注意块组描述符的大小,没有设置64bit feature时为32B，//兼容EXT2 & EXT4**

//1. Read Group desc, according to gd\_num:

//\_\_u16 s\_desc\_size; /\* Group desc. size: INCOMPAT\_64BIT \*/

char \*buf = malloc(1024);

struct ext2\_group\_desc \*gdt, \*gdp;

gdp = gdt = malloc(gd\_num \* sizeof(struct ext2\_group\_desc));

retval = io\_channel\_read\_blk64(current\_fs->io, 2, 1, buf);

printf("\ngroup descrip info:%d\nGroup 0:\n", sizeof(struct ext2\_group\_desc));

memcpy((char \*)gdt, buf, 32);

printf("bg\_block\_bitmap block loc:%d\n", gdt->bg\_block\_bitmap);

printf("bg\_inode\_bitmap block loc:%d\n", gdt->bg\_inode\_bitmap);

printf("bg\_inode\_table block loc:%d, inode\_number per GD:%d\n",

gdt->bg\_inode\_table, super->s\_inodes\_count / gd\_num);

printf("bg\_free\_blocks\_count:%d\n", gdt->bg\_free\_blocks\_count);

printf("Check sum:0x%x\n", gdt->bg\_checksum);

printf("bg\_block\_bitmap\_csum\_lo sum:0x%x\n", gdt->bg\_block\_bitmap\_csum\_lo);

printf("bg\_inode\_bitmap\_csum\_lo sum:0x%x\n", gdt->bg\_inode\_bitmap\_csum\_lo);

gdt++;

if (super->s\_desc\_size != 64)

memcpy((char \*)gdt, buf + 32, 32);

else

memcpy((char \*)gdt, buf + 64, 32);

printf("\n\nGroup 1:\n");

printf("bg\_block\_bitmap block loc:%d\n", gdt->bg\_block\_bitmap);

printf("bg\_inode\_bitmap block loc:%d\n", gdt->bg\_inode\_bitmap);

printf("bg\_inode\_table block loc:%d, inode\_number per GD:%d\n",

gdt->bg\_inode\_table, super->s\_inodes\_count / gd\_num);

printf("bg\_free\_blocks\_count:%d\n", gdt->bg\_free\_blocks\_count);

printf("Check sum:0x%x\n", gdt->bg\_checksum);

**//根据块组描述符的信息得到inode table的block位置**

//2. Read inode table,root inode number is EXT2\_ROOT\_INO(2)

printf("\n\nbg\_inode\_table block loc:%d\n", gdp->bg\_inode\_table);

retval = io\_channel\_read\_blk64(current\_fs->io, gdp->bg\_inode\_table, 1, buf);

struct ext2\_inode \*inp;

**//得到ROOT的inode数据结构**

//\*\*\*we need get inode struct size(128B/256B)

inp = (struct ext2\_inode \*)(buf + super->s\_inode\_size \* (EXT2\_ROOT\_INO - 1));

//dump\_inode(2, inp); /\* dump root inode information \*/

printf("root block location:%d\n", inp->i\_block[0]);

**//读取root的目录信息**

retval = io\_channel\_read\_blk64(current\_fs->io, inp->i\_block[0], 1, buf);

//format\_print(buf, 1024, 16);

//3. find hello.c file in root dir

//list files under root

struct ext2\_dir\_entry\_t \*dirp; **//filetype属性，因此使用该数据结构**

int offset = 0;

char \*charp = buf;

while(1) {

if (offset > 1024)

break;

dirp = (struct ext2\_dir\_entry\_t \*)charp;

if (dirp->name\_len > 0) {

printf("file name:%15s ino:%5d\n", dirp->name, dirp->inode);

if (strncmp(dirp->name, "hello.c", dirp->name\_len) == 0) {

printf("We find hello.c haha!\n");

break;

}

}

charp = charp + dirp->rec\_len;

}

**//查找到hello.c文件的ino号**

int ino = dirp->inode;

int block = (ino - 1) / (1024 / super->s\_inode\_size);

//4. find hello.c 's inode struct

retval = io\_channel\_read\_blk64(current\_fs->io, gdp->bg\_inode\_table + block, 1, buf);

struct ext2\_inode inode\_struct;

inode\_struct = \*(struct ext2\_inode \*)(buf + super->s\_inode\_size \* ((ino - 1) % 8));

printf("\n\nhello.c size:%d\n", inode\_struct.i\_size);

//5. read data

**//读取文件**

retval = io\_channel\_read\_blk64(current\_fs->io, inode\_struct.i\_block[0], 1, buf);

format\_print(buf, inode\_struct.i\_size, 16);

free(buf);

return;

}

root@PATEO-HYG:/work1/e2fs/e2fsprogs-1.42.2/toolbox/sbin# ./debugfs 16M.img

debugfs 1.42.2 (27-Mar-2012)

debugfs: a

super->s\_blocks\_count:16384

super->s\_blocks\_per\_group:8192

gd\_num:2

EXT2\_DESC\_PER\_BLOCK:32

group descrip info:32

Group 0:

bg\_block\_bitmap block loc:66

bg\_inode\_bitmap block loc:82

bg\_inode\_table block loc:98, inode\_number per GD:2048

bg\_free\_blocks\_count:7609

Check sum:0xfdfa

bg\_block\_bitmap\_csum\_lo sum:0x0

bg\_inode\_bitmap\_csum\_lo sum:0x0

Group 1:

bg\_block\_bitmap block loc:67 #**使用flex\_gb属性，位图在Group0中**

bg\_inode\_bitmap block loc:83

bg\_inode\_table block loc:354, inode\_number per GD:2048

bg\_free\_blocks\_count:7096

Check sum:0x8166

bg\_inode\_table block loc:98

root block location:68

file name: . ino: 2

file name: .. ino: 2

file name: hello.cund ino: 11

We find hello.c haha!

hello.c size:20

0x00000000 - 0x00000010: 49 20 6c 6f 76 65 20 79 6f 75 20 6c 73 21 20 68 : I love you ls!

0x00000010 - 0x00000020: 61 68 61 0a