[戏说文件系统之ext2【上】](http://blog.chinaunix.net/uid-23069658-id-3468489.html)

虚拟文件系统VFS是对各种文件系统的一个抽象层，抽取其共性，以便对外提供统一管理接口，便于内核对不同种类的文件系统进行管理。那么首先我们得看一下对于一个具体的文件系统，我们该关注重点在哪里。

对于存储设备(以硬盘为例)上的数据，可分为两部分：

用户数据：存储用户实际数据的部分；

管理数据：用于管理这些数据的部分，这部分我们通常叫它元数据(metadata)。

我们今天要讨论的就是这些元数据。这里有个概念首先需要明确一下：块设备。所谓块设备就是以块为基本读写单位的设备，支持缓冲和随机访问。每个文件系统提供的mk2fs.xx工具都支持在构建文件系统时由用户指定块大小，当然用户不指定时会有一个缺省值。在博文“[硬盘的存储原理和内部架构](http://blog.chinaunix.net/uid-23069658-id-3413957.html)”里我们提到过，硬盘的每个扇区512字节，而多个相邻的若干扇区就构成了一个簇，从文件系统的角度看这个簇对应的就是我们这里所说块。用户从上层下发的数据首先被缓存在块设备的缓存里，当写满一个块时数据才会被发给硬盘驱动程序将数据最终写到存储介质上。如果想将设备缓存中数据立即写到存储介质上可以通过sync命令来完成。上一篇博文里我们也说，块越大存储性能越好，但浪费比较严重；块越小空间利用率较高，但性能相对较低。如果你不是专业的“骨灰级”玩儿家，在存储设备上构建文件系统时，块大小就用默认值。通过命令“tune2fs -l /dev/sda1”可以查看该存储设备上文件系统所使用的块大小：

点击(此处)折叠或打开

1. [root@localhost ~]# tune2fs -l /dev/sda1
2. tune2fs 1.39 (29-May-2006)
3. Filesystem volume name: /boot
4. Last mounted on: <not available>
5. Filesystem UUID: 6ade5e49-ddab-4bf1-9a45-a0a742995775
6. Filesystem magic number: 0xEF53
7. Filesystem revision #: 1 (dynamic)
8. Filesystem features: has\_journal ext\_attr resize\_inode dir\_index filetype needs\_recovery sparse\_super
9. Default mount options: user\_xattr acl
10. Filesystem state: clean
11. Errors behavior: Continue
12. Filesystem OS type: Linux
13. Inode count: 38152
14. Block count: 152584
15. Reserved block count: 7629
16. Free blocks: 130852
17. Free inodes: 38111
18. First block: 1
19. Block size: 1024
20. Fragment size: 1024
21. Reserved GDT blocks: 256
22. Blocks per group: 8192
23. Fragments per group: 8192
24. Inodes per group: 2008
25. Inode blocks per group: 251
26. Filesystem created: Thu Dec 13 00:42:52 2012
27. Last mount time: Tue Nov 20 10:35:28 2012
28. Last write time: Tue Nov 20 10:35:28 2012
29. Mount count: 12
30. Maximum mount count: -1
31. Last checked: Thu Dec 13 00:42:52 2012
32. Check interval: 0 (<none>)
33. Reserved blocks uid: 0 (user root)
34. Reserved blocks gid: 0 (group root)
35. First inode: 11
36. Inode size: 128
37. Journal inode: 8
38. Default directory hash: tea
39. Directory Hash Seed: 72070587-1b60-42de-bd8b-a7b7eb7cbe63
40. Journal backup: inode blocks

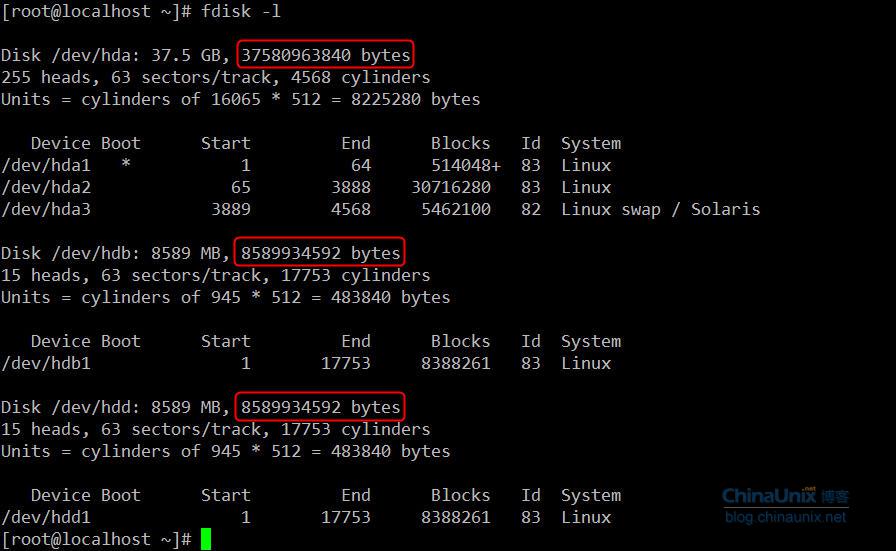
该命令已经暴露了文件系统的很多信息，接下我们将详细分析它们。

下图是我的虚拟机的情况，三块IDE的硬盘。容量分别是：

hda: 37580963840/(1024\*1024\*1024)=35GB

hdb: 8589934592/(1024\*1024\*1024)=8GB

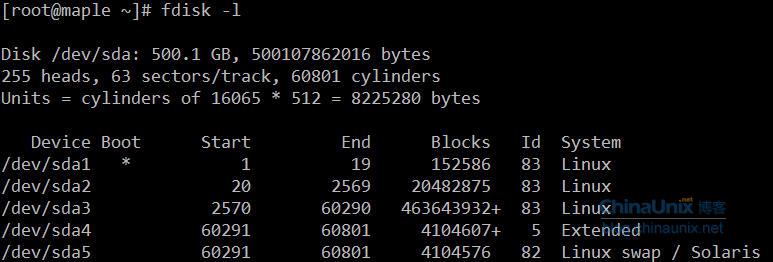
hdd: 8589934592/(1024\*1024\*1024)=8GB

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357743586lfPN.jpg)

如果这是三块实际的物理硬盘的话，厂家所标称的容量就分别是37.5GB、8.5GB和8.5GB。可能有些童鞋觉得虚拟机有点“假”，那么我就来看看实际硬盘到底是个啥样子。

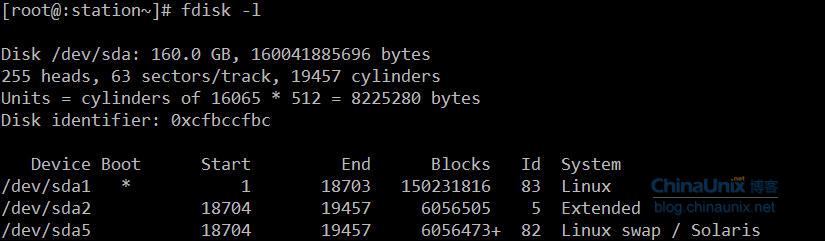
**主角1：西部数据 500G SATA接口 CentOS 5.5**

**实际容量：500107862016B = 465.7GB**

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357743764MMbQ.jpg)

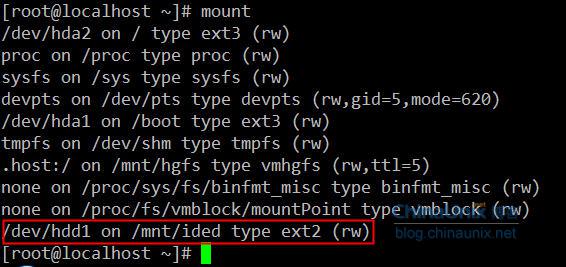
**主角2：希捷 160G SCSI接口 CentOS 5.5**

**实际容量：160041885696B=149GB**

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357743767dXMM.jpg)

大家可以看到，VMware公司的水平还是相当不错的，虚拟硬盘和物理硬盘“根本”看不出差别，毕竟属于云平台基础架构支撑者的风云人物嘛。

以硬盘/dev/hdd1为例，它是我新增的一块新盘，格式化成ext2后，根目录下只有一个lost+found目录，让我们来看一下它的布局情况，以此来开始我们的文件系统之旅。

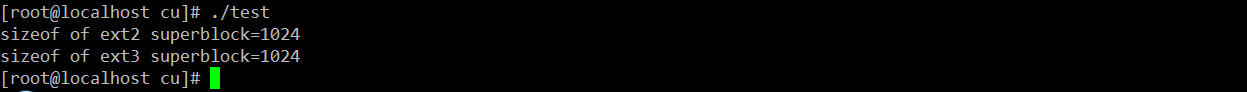
[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744015AGlb.jpg)

对于使用了ext2文件系统的分区来说，superblock的大小为1024字节，其实ext3的superblock也是1024字节。下面的小程序可以证明这一点：

点击(此处)折叠或打开

1. #include <stdio.h>
2. #include <linux/ext2\_fs.h>
3. #include <linux/ext3\_fs.h>
4. int main(int argc,char\*\* argv){
5. printf("sizeof of ext2 superblock=%d\n",sizeof(struct ext2\_super\_block));
6. printf("sizeof of ext3 superblock=%d\n",sizeof(struct ext3\_super\_block));
7. return 0;
8. }

输出结果如下：

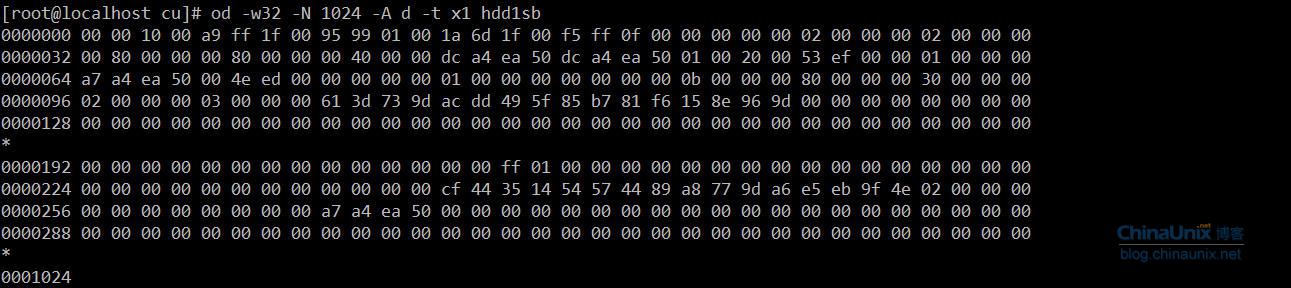
[](http://blog.chinaunix.net/attachment/201301/9/23069658_13577440187Zvj.jpg)

硬盘的第一个字节是从0开始编号，所以第一个字节是byte0，以此类推。/dev/hdd1分区头部的1024个字节(从byte0~byte1023)都用0填充，因为/dev/hdd1不是主引导盘。superblock是从byte1024开始，占1024B存储空间。我们用dd命令把superblock的信息提取出来：

点击(此处)折叠或打开

1. dd if=/dev/hdd1 of=./hdd1sb bs=1024 skip=1 count=1

上述命令将从/dev/hdd1分区的byte1024处开始，提取1024个字节的数据存储到当前目录下的hdd1sb文件里，该文件里就存储了我们superblock的所有信息，如下：

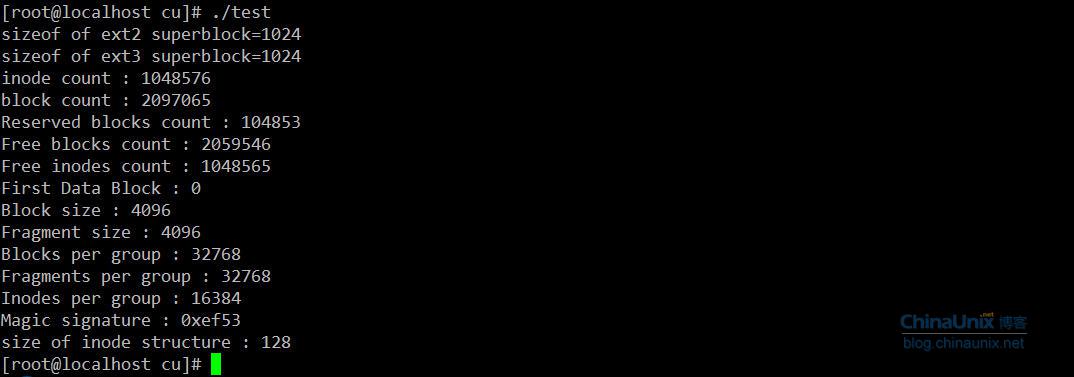
[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744187wPMt.jpg)

上面的程序稍加改造，我们就可以以更直观的方式看到superblock的输出了：

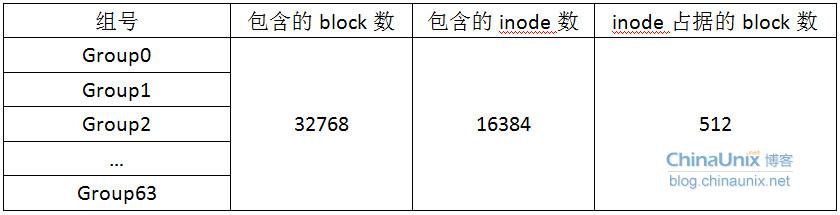
点击(此处)折叠或打开

1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <unistd.h>
6. #include <string.h>
7. #include <linux/ext2\_fs.h>
8. #include <linux/ext3\_fs.h>
9. int main(int argc,char\*\* argv){
10. printf("sizeof of ext2 superblock=%d\n",sizeof(struct ext2\_super\_block));
11. printf("sizeof of ext3 superblock=%d\n",sizeof(struct ext3\_super\_block));
12. char buf[1024] = {0};
13. int fd = -1;
14. struct ext2\_super\_block hdd1sb;
15. memset(&hdd1sb,0,1024);
16. if(-1 == (fd=open("./hdd1sb",O\_RDONLY,0777))){
17. printf("open file error!\n");
18. return 1;
19. }
20. if(-1 == read(fd,buf,1024)){
21. printf("read error!\n");
22. close(fd);
23. return 1;
24. }
25. memcpy((char\*)&hdd1sb,buf,1024);
26. printf("inode count : %ld\n",hdd1sb.s\_inodes\_count);
27. printf("block count : %ld\n",hdd1sb.s\_blocks\_count);
28. printf("Reserved blocks count : %ld\n",hdd1sb.s\_r\_blocks\_count);
29. printf("Free blocks count : %ld\n",hdd1sb.s\_free\_blocks\_count);
30. printf("Free inodes count : %ld\n",hdd1sb.s\_free\_inodes\_count);
31. printf("First Data Block : %ld\n",hdd1sb.s\_first\_data\_block);
32. printf("Block size : %ld\n",1<<(hdd1sb.s\_log\_block\_size+10));
33. printf("Fragment size : %ld\n",1<<(hdd1sb.s\_log\_frag\_size+10));
34. printf("Blocks per group : %ld\n",hdd1sb.s\_blocks\_per\_group);
35. printf("Fragments per group : %ld\n",hdd1sb.s\_frags\_per\_group);
36. printf("Inodes per group : %ld\n",hdd1sb.s\_inodes\_per\_group);
37. printf("Magic signature : 0x%x\n",hdd1sb.s\_magic);
38. printf("size of inode structure : %d\n",hdd1sb.s\_inode\_size);
39. close(fd);
40. return 0;
41. }

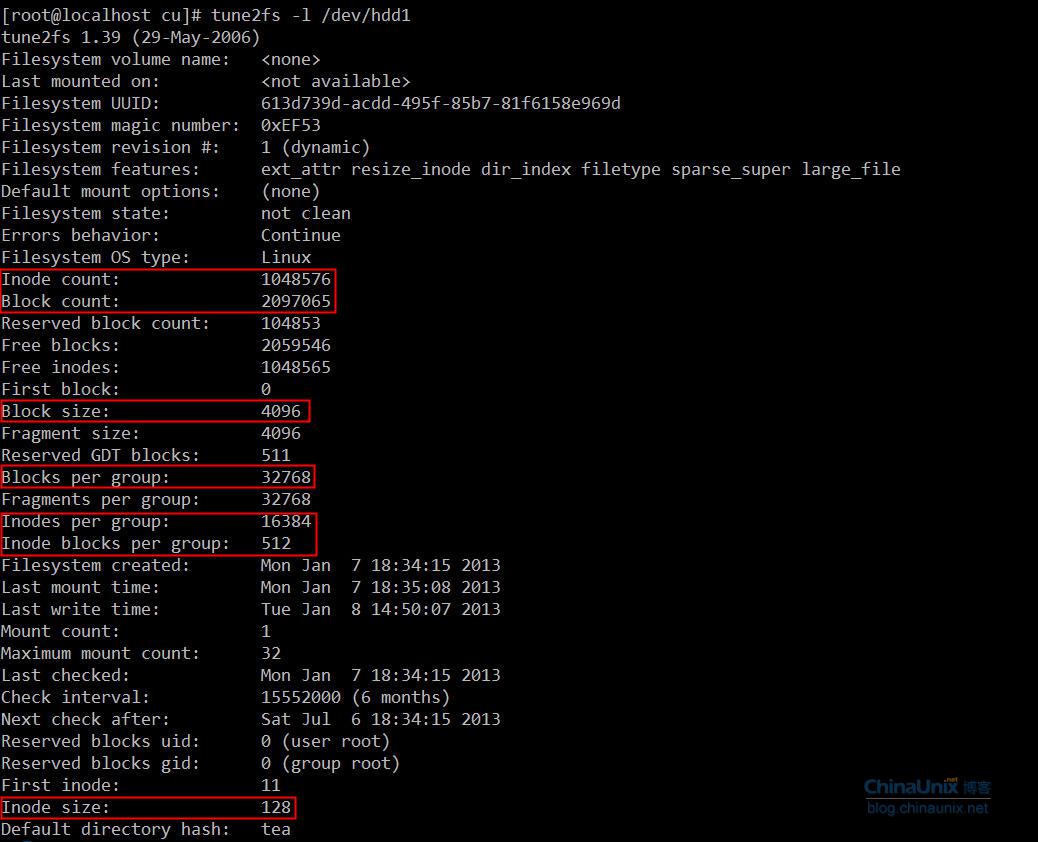
打印结果如下：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_13577441908z3B.jpg)

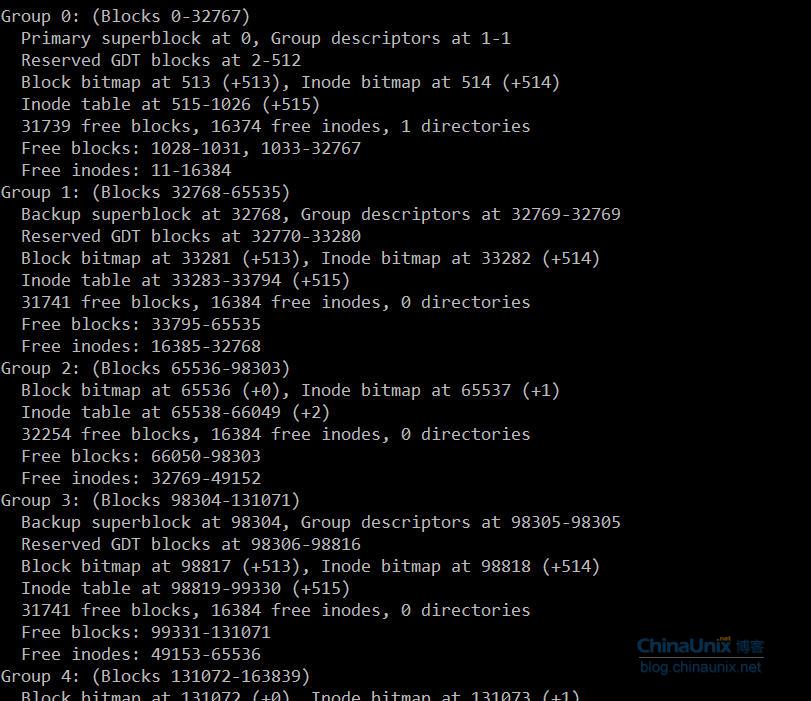
对于ext2/ext3文件系统来说数字签名Magic signature都是0xef53，如果不是那么它一定不是ext2/ext3文件系统。这里我们可以看到，我们的/dev/hdd1确实是ext2文件系统类型。hdd1中一共包含1048576个inode节点(inode编号从1开始)，每个inode节点大小为128字节，所有inode消耗的存储空间是1048576×128=128MB；总共包含2097065个block，每个block大小为4096字节，每32768个block组成一个group，所以一共有2097065/32768=63.99，即64个group(group编号从0开始，即Group0～Group63)。 所以整个/dev/hdd1被划分成了64个group，详情如下：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744369C5DG.jpg)

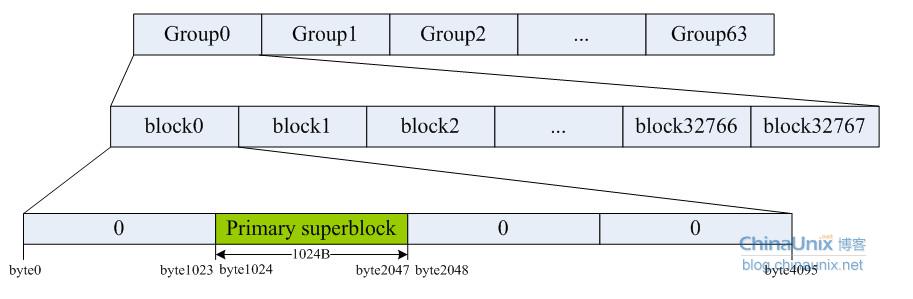
用命令tune2fs可以验证我们之前的分析：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744401Q9mR.jpg)

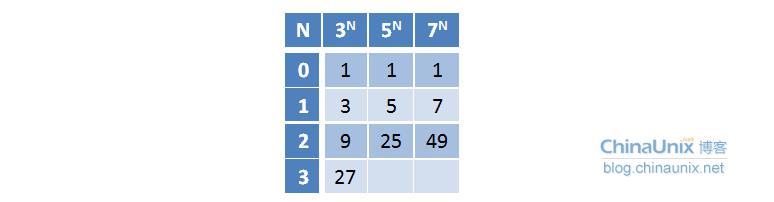
再通过命令dumpe2fs /dev/hdd1的输出，可以得到我们关注如下部分：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744406Mvto.jpg)

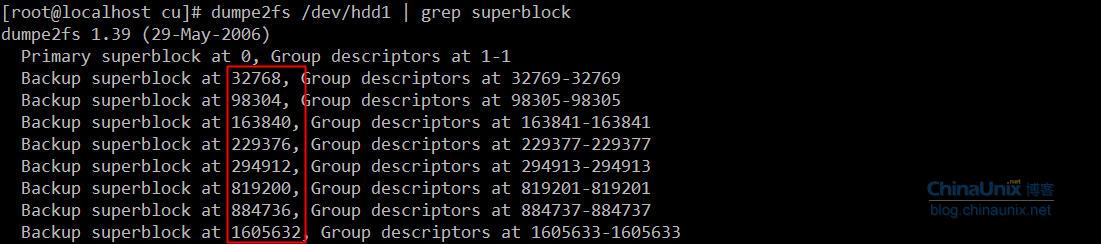
接下来以Group0为例，主superblock在Group0的block0里，根据前面的分析，我们可以画出主superblock在block0中的位置如下：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744525tOTv.jpg)

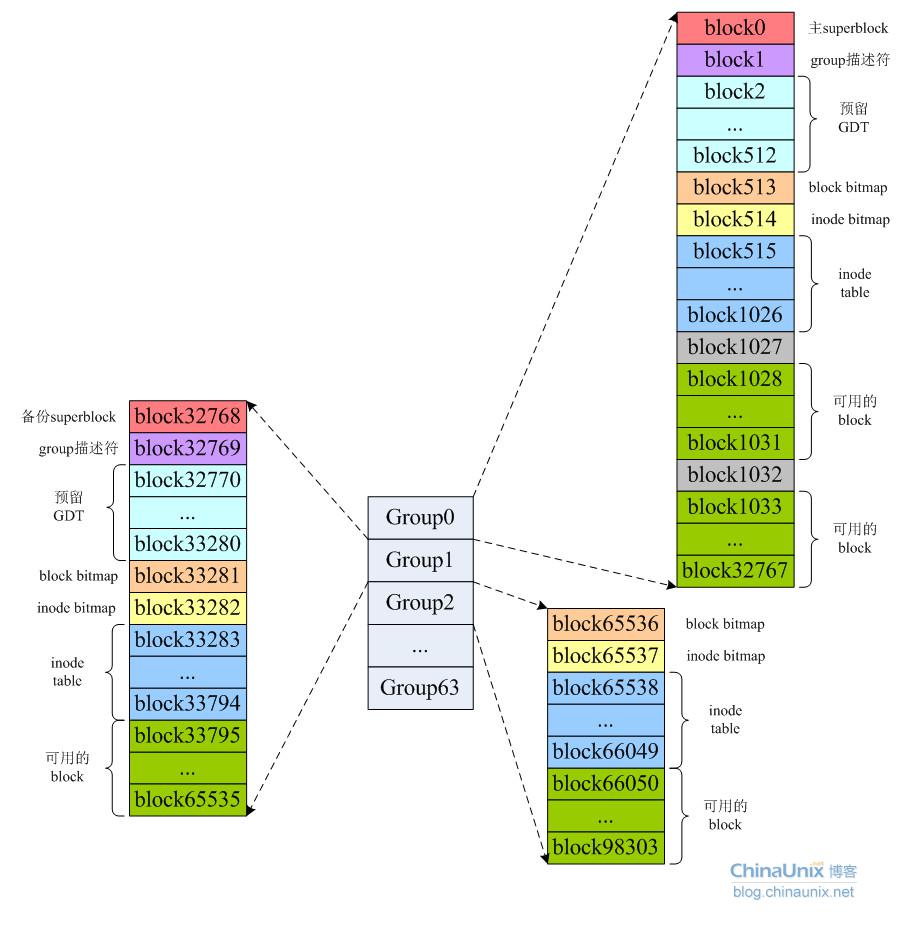
因为superblock是如此之重要，一旦它出错你的整个系统就玩儿完了，所以文件系统中会存在磁盘的多个不同位置会存在主superblock的备份副本，一旦系统出问题后还可以通过备份的superblock对文件系统进行修复。第一版ext2文件系统的实现里，每个Group里都存在一份superblock的副本，然而这样做的负面效果也是相当明显，那就是严重降低了磁盘的空间利用率。所以在后续ext2的实现代码中，选择用于备份superblock的Group组号的原则是3N、5N、7N其中N=0,1,2,3…。根据这个公式我们来计算一下/dev/hdd1中备份有supeblock的Group号：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744586IMD2.jpg)

也就是说Group1、3、5、7、9、25、27、49里都保存有superblock的拷贝，如下：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357744411KGbg.jpg)

用block号分别除以32768就得到了备份superblock的Group号，和我们在上面看到的结果一致。我们来看一下/dev/hdd1中Group和block的关系：

[](http://blog.chinaunix.net/attachment/201301/9/23069658_1357745026Y4l0.jpg)

从上图中我们可以清晰地看出在使用了ext2文件系统的分区上，包含有主superblock的Group、备份superblock的Group以及没有备份superblock的Group的布局情况。存储了superblock的Group中有一个组描述符(Group descriptors)紧跟在superblock所在的block后面，占一个block大小；同时还有个Reserved GDT跟在组描述符的后面。

Reserved GDT的存在主要是支持ext2文件系统的resize功能，它有自己的inode和data block，这样一来如果文件系统动态增大，Reserved GDT就正好可以腾出一部分空间让Group descriptor向下扩展。

今天我们来认识一下superblock，inode，block，group，group descriptor，block bitmap，inode table这些家伙。为什么在上一篇博文开篇没详细介绍这些概念呢，因为任何关于文件系统的文章还是书籍一开始都是先说概念、说理论，让人一直有种雾里看花的感觉。纸上得来终觉浅，事必躬亲才印象深，所以我们以一块实际硬盘为例(当然是虚拟出来的，呵呵)来向大家展示了一下文件系统在存储介质上的布局情况，先让大家对其有个比较直观的认识，然后再逐一对它们进行解释和说明，印象会更深刻些。咱不否认理论的重要性，秉承着“理论指导实践，以实践加深对理论的认识”的宗旨来一步一步入文件系统的乐园。

**superblock：**

这个东西确实很重要，前面我们已经见识过。为此，文件系统还特意精挑细选的找了N多后备Group，在这些Group中都存有superblock的副本，你就知道它有多重要了。说白了，superblock 的作用就是记录文件系统的类型、block大小、block总数、inode大小、inode总数、group的总数等等。

**group descriptors：**

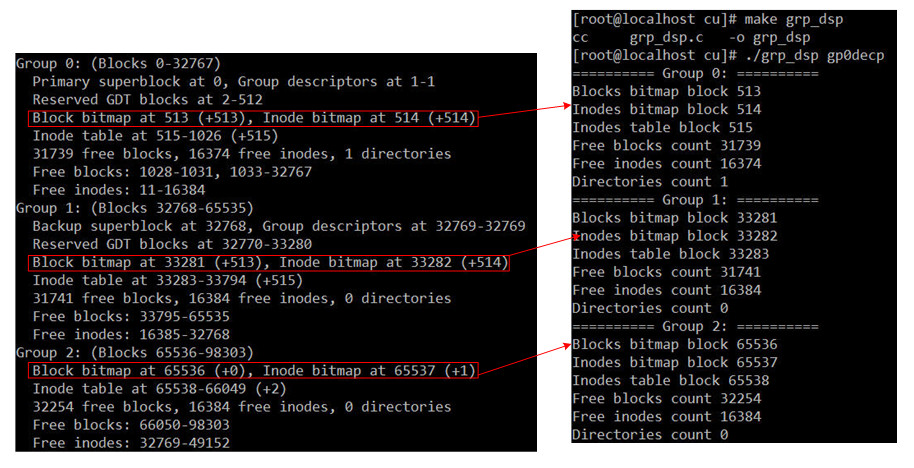
千万不要以为这就是一个组描述符，看到descriptor后面加了个s就知道这是N多描述符的集合。确实，这是文件系统中所有group的描述符所构成的一个数组，它的结构定义在include/linux/ext2\_fs.h中：

|  |
| --- |
| /\*  \* Structure of a blocks group descriptor  \*/  struct ext2\_group\_desc  {  \_\_le32 bg\_block\_bitmap; /\* group中block bitmap所在的第一个block号 \*/  \_\_le32 bg\_inode\_bitmap; /\* group中inode bitmap 所在的第一个block号 \*/  \_\_le32 bg\_inode\_table; /\* group中inodes table 所在的第一个block号 \*/  \_\_le16 bg\_free\_blocks\_count; /\* group中空闲的block总数 \*/  \_\_le16 bg\_free\_inodes\_count; /\* group中空闲的inode总数\*/  \_\_le16 bg\_used\_dirs\_count; /\* 目录数 \*/  \_\_le16 bg\_pad;  \_\_le32 bg\_reserved[3];  }; |

下面的程序可以帮助了解一下/dev/hdd1中所有group的descriptor的详情：

|  |
| --- |
| [root@localhost cu]# cat grp\_dsp.c  #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <string.h>  #include <linux/ext2\_fs.h>  #define B\_LEN 32 //一个struct ext2\_group\_desc{}占固定32字节  int main(int argc,char\*\* argv){  char buf[B\_LEN] = {0};  int i=0,fd = -1;  struct ext2\_group\_desc gd;  memset(&gd,0,B\_LEN);  if(-1 == (fd=open(argv[1],O\_RDONLY,0777))){  printf("open file error!\n");  return 1;  }  while(i<64){ //因为我已经知道了/dev/hdd1中只有64个group  if(-1 == read(fd,buf,B\_LEN)){  printf("read error!\n");  close(fd);  return 1;  }  memcpy((char\*)&gd,buf,B\_LEN);  printf("========== Group %d: ==========\n",i);  printf("Blocks bitmap block %ld \n",gd.bg\_block\_bitmap);  printf("Inodes bitmap block %ld \n",gd.bg\_inode\_bitmap);  printf("Inodes table block %ld \n",gd.bg\_inode\_table);  printf("Free blocks count %d \n",gd.bg\_free\_blocks\_count);  printf("Free inodes count %d \n",gd.bg\_free\_inodes\_count);  printf("Directories count %d \n",gd.bg\_used\_dirs\_count);  memset(buf,0,B\_LEN);  i++;  }  close(fd);  return 0;  } |

运行结果和dumpe2fs /dev/hdd1的输出对比如下：



其中，文件gp0decp是由命令“**dd if=/dev/hdd1 of=./gp0decp bs=4096 skip=1 count=1**”生成。每个group descriptor里记录了该group中的inode table的起始block号，因为inode table有可能会占用连续的多个block；空闲的block、inode数等等。

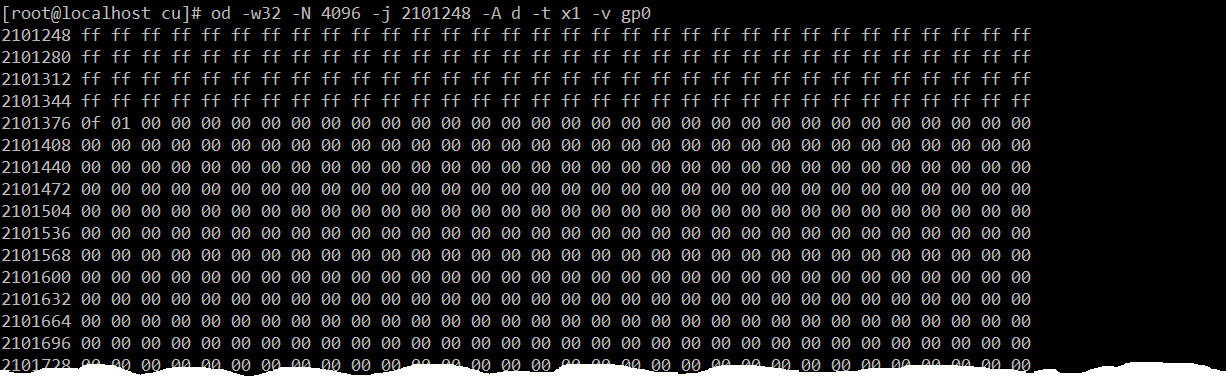
**block bitmap:**

在文件系统中每个对象都有一个对应的inode节点(这句话有些不太准确，因为符号链接和它的目标文件共用一个inode)，里存储了一个对象(文件或目录)的信息有权限、所占字节数、创建时间、修改时间、链接数、属主ID、组ID，如果是文件的话还会包含文件内容占用的block总数以及block号。inode是从1编号，这一点不同于block，需要格外注意。另外，/dev/hdd1是新挂载的硬盘，格式化成ext2后并没有任何数据，只有一个lost+found目录。接下来我们用命令“**dd if=/dev/hdd1 of=./gp0 bs=4096 count=32768**”将Group0里的所有数据提取出来。

前面已经了解了Group0的一些基本信息如下：

|  |
| --- |
| Group 0: (Blocks 0-32767)  Primary superblock at 0, Group descriptors at 1-1  Reserved GDT blocks at 2-512  Block bitmap at 513 (+513), Inode bitmap at 514 (+514)  Inode table at 515-1026 (+515)  31739 free blocks, 16374 free inodes, 1 directories #包含一个目录  Free blocks: 1028-1031, 1033-32767 #一共有31739个空闲block  Free inodes: 11-16384 #一共有16374个空闲inode |

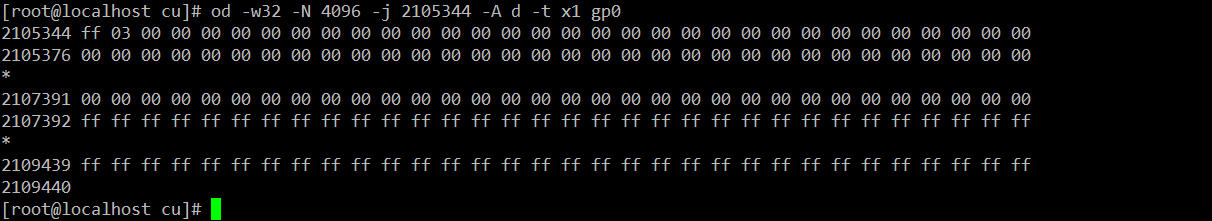
一个block bitmap占用一个block大小，而block bitmap中每个bit表示一个对应block的占用情况，0表示对应的block为空，为1表示相应的block中存有数据。在/dev/hdd1中，一个group里最多只能包含8×4096=32768个block，这一点我们已经清楚了。接下来我们来看一下Group0的block bitmap，如下：



发现block bitmap的前128字节和第129字节的低4位都为1，说明发现Group0中前128×8+4=1028个block，即block0~block1027都已被使用了。第129字节的高4位为0，表示block1028~block1031四个block是空闲的；第130字节的最低位是1，说明block1032被占用了；从block1033～block32767的block bitmap都是0，所以这些block都是空闲的，和上表输出的结果一致。

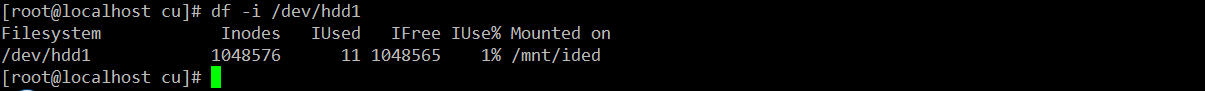
**inode bitmap:**

和block bitmap类似，每个比特表示相应的inode是否被使用。Group0的inode bitmap如下：



/dev/hdd1里inode总数为1048576，要被均分到64个Group里，所以每个Group中都包含了16384个inode。要表示每个Group中16384个inode，inode bitmap总共需要使用2048(16384/8)字节。inode bitmap本身就占据了一个block，所以它只用到了该block中的前2048个字节，剩下的2048字节都被填充成1，如上图所示。

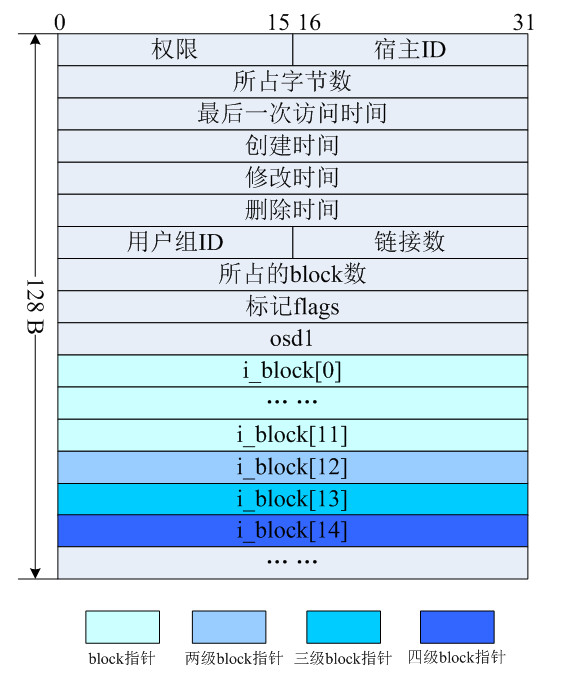
我们可以看到Group0中的inode bitmap前两个字节分别是ff和03，说明Group0里的前11个inode已经被使用了。其中前10个inode被ext2预留起来，第11个inode就是lost+found目录，如下：



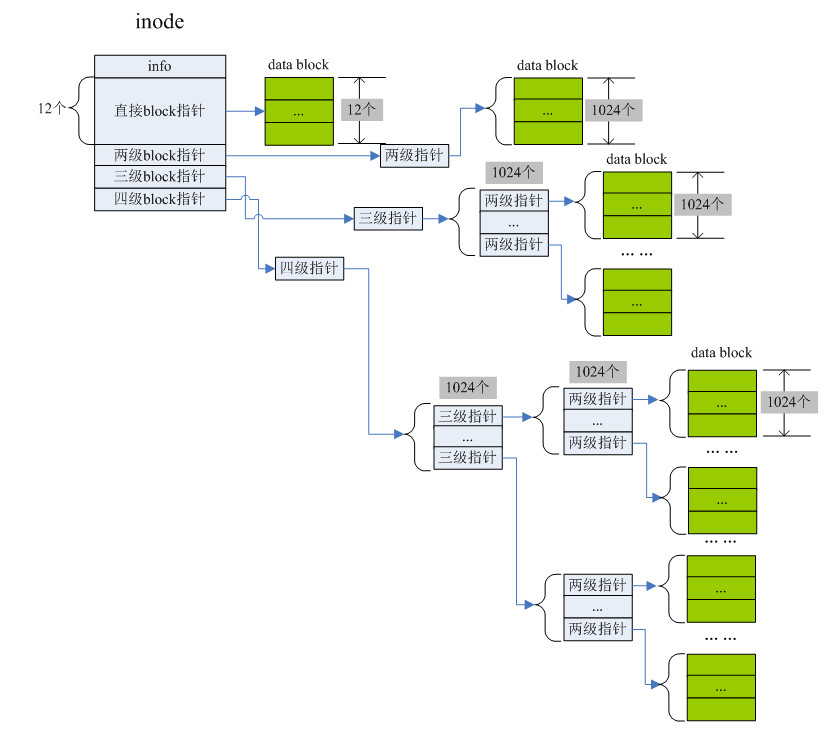
**inode table:**

那么每个Group中的所有inode到底存放在哪里呢？答案就是inode table。它是每个Group中所有inode的聚合地。

因为一个inode占128字节，所以每个Group里的所有inode共占16384×128=2097152字节，总共消耗了512个block。Group的group descriptor里记录了inode table的所占block的起始号，所以就可以唯一确定每个Group里inode table所在的block的起始号和结束号了。inode的结构如下：



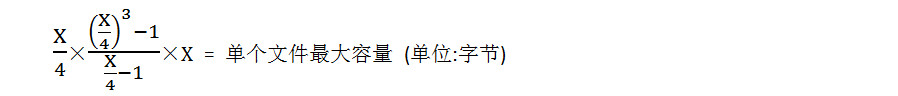
这里我们主要关注的其中的数据block指针部分。前12个block指针直接指向了存有数据的block号；第13个block指针所指向的block中存储的并不是数据而是由其他block号，这些block号所对应的block里存储的才是真正的数据，即所谓的两级block指针；第14个block为三级block指针；第15个block为四级block指针。最后效果图如下：



一个block为4096字节，每个块指针4字节，所以一个block里最多可以容纳4096/4=1024个block指针，我们可以计算出一个inode最大能表示的单个文件的最大容量如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 直接block指针(字节) | 两级block指针(字节) | 三级block指针(字节) | 四级block指针(字节) | 单个文件的最大容量(字节) |
| 12×409 | 4096/4×4096 | 40962/4×4096 | 40963/4×4096 | 4TB |

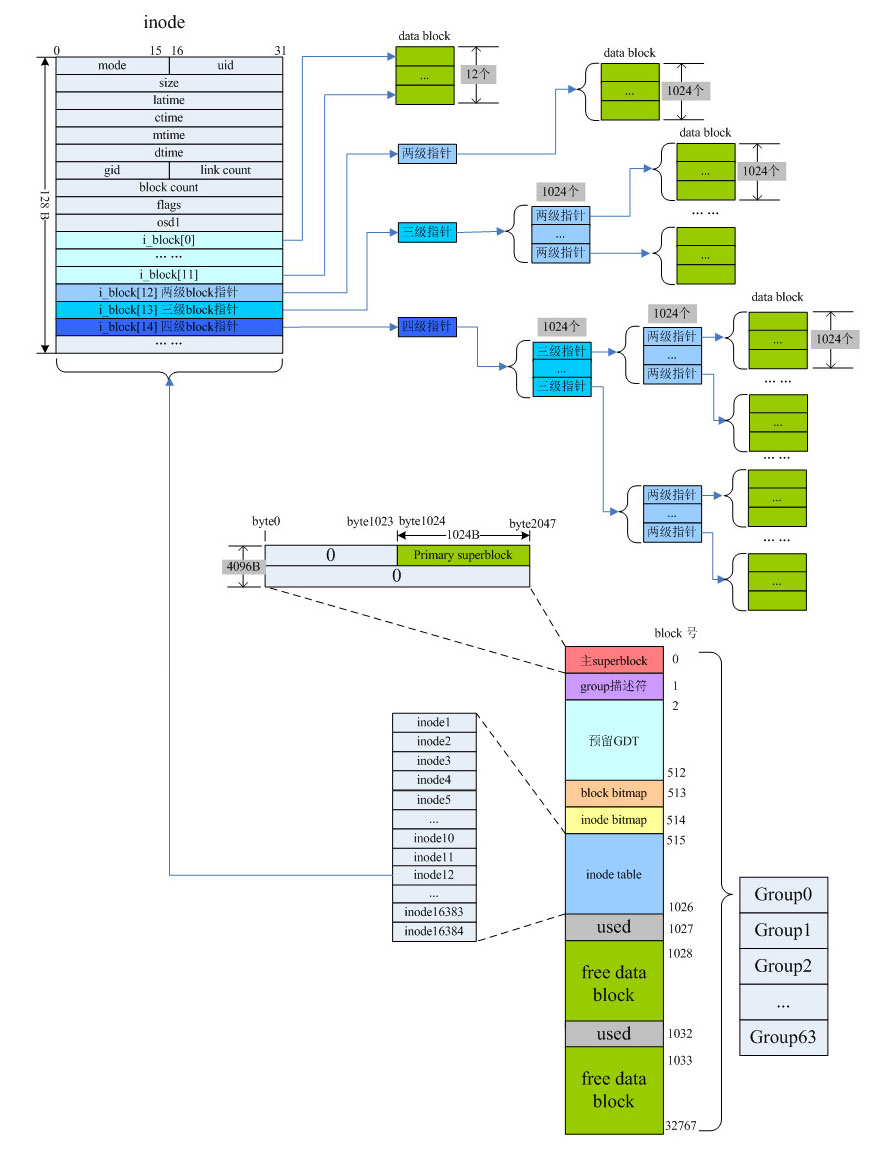
所以，我们可以得出不同block大小，对单个文件最大容量的影响。假设block大小为X字节，则:



如下表所示：

|  |  |
| --- | --- |
| block大小(字节) | 单个文件容量(字节) |
| 1024 | 17247240192字节(16GB) |
| 2048 | 275415826432字节(256GB) |
| 4096 | 4402345672704字节(4TB) |

最后来一张全家福：



完。