[UEFI Practice] EXT2 Read Driver

摘要 This article analyzes in detail the composition of the disk in the EXT2 file system, including key components such as the boot block, block group, super block, group descriptor, and how to locate and read file data through these structures, involving block size, file system structure, Inode management, and data block addressing algorithm.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

General Description

From the perspective of the EXT2 file system, the disk - under BIOS , it should not be considered a disk, but a disk partition `Block IO` represented by it - is composed as follows:

Boot Block	Block Group 0	Block Group 1	...	Block group N

There are two different meanings of blocks:

1. The hardware represents the blocks of the disk itself, and its size can be `BlockIo->Media->BlockSize` obtained by;
2. There is a software block concept in the EXT2 file system: The EXT2 file system is a block-based file system that divides the hard disk into several blocks, each of which has the same length, and manages metadata and file systems by blocks.

EXT2's (software) blocks are affected by the underlying disk's block size. The possible values are 1024 bytes, 2048 bytes, and 4096 bytes, which do not correspond to the disk size. If the disk block size is 512 bytes, then EXT2's block size will usually be 1024 bytes. Of course, this is not mandatory. When building an EXT2 file system, you can modify the software block size according to the actual situation. After all, unlike the hardware-related value of the disk block size, EXT2's block size is a software concept. If there is no special explanation later, it refers to the EXT2 software block . The EXT2 block size also affects the maximum file length in the file system:

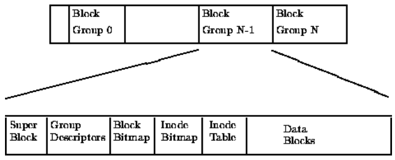
Block size	Maximum file length
1024	16GiB
2048	256GiB
4096	2TiB

In addition, a block (**1024 bytes (1KB) in subsequent implementations**) will store data for at most one file. If the data of a file is larger than a block, the blocks occupied by the file data will be rounded up .

The boot block is part of the disk partition structure, while **block group 0 to block group N** occupy the rest of the partition. They are the focus of this study. The size of the block group can also be set, usually 8092 blocks constitute a block group. The distribution of block groups is as follows:

Super Block	Group Descriptor	Data block bitmap	Inode bitmap	Inode table	Data Block

In general, `Block IO` the structure of a disk partition is shown in the following figure:

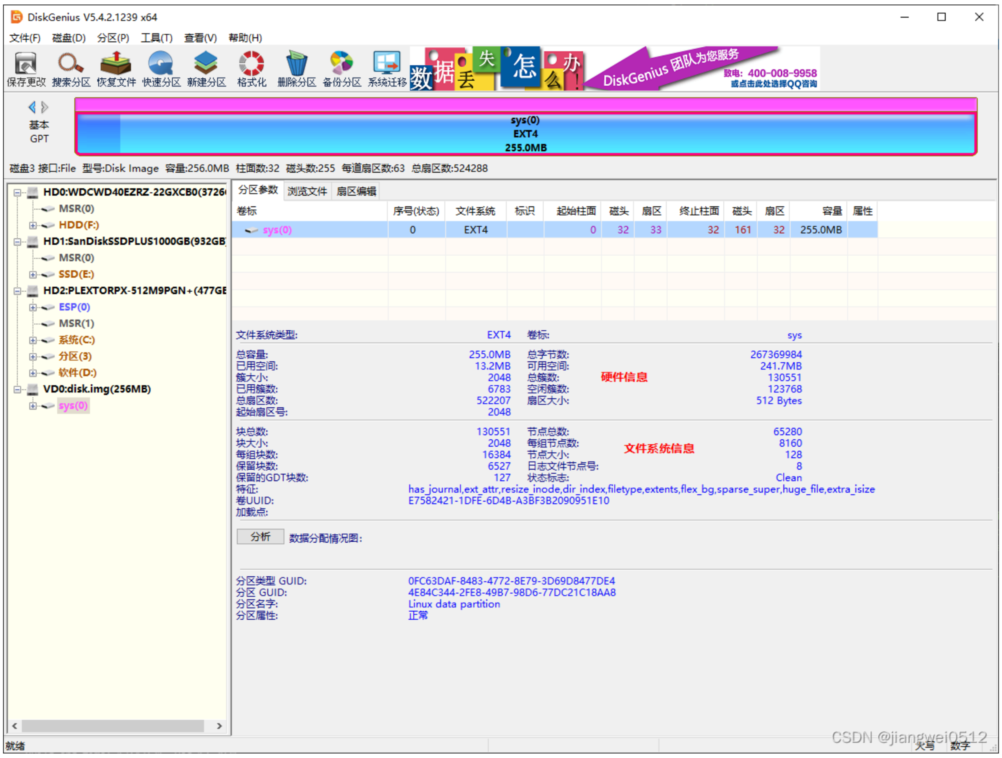


The functions of each structure in the block group are as follows:

- **Superblock** : The core structure used to store the metadata of the file system. Theoretically, superblocks exist in all block groups, but only the superblock in the first block group is used. The reason for this redundant design is to prevent the file system from crashing when a superblock is damaged. The superblock is usually located in the second block of the partition (because the first block is the boot block).
- **Group descriptor** : Each block group has a set of group descriptors, which follow the superblock. The information stored in it reflects the content of each block group in the file system, so it is not only related to the data blocks of the current block group, but also to the data blocks and Inode blocks of other block groups. The occupied blocks are related to the block size, the number of block groups and the length of the group descriptor.
- **Data block bitmap** and **Inode bitmap** : used to store a long string of bits, each bit corresponds to a data block or Inode, used to indicate whether the corresponding data block or Inode is free. Each occupies 1 block.
- **Inode table** : Inode is used to store metadata related to each file and directory in the file system. Each file or directory has one and only one corresponding Inode , which contains metadata (such as access permissions, last modification log, etc., but not the file name) and a pointer to the file data; it occupies several blocks.
- **Data blocks** : Contains useful data of files in the file system and occupies the remaining blocks.

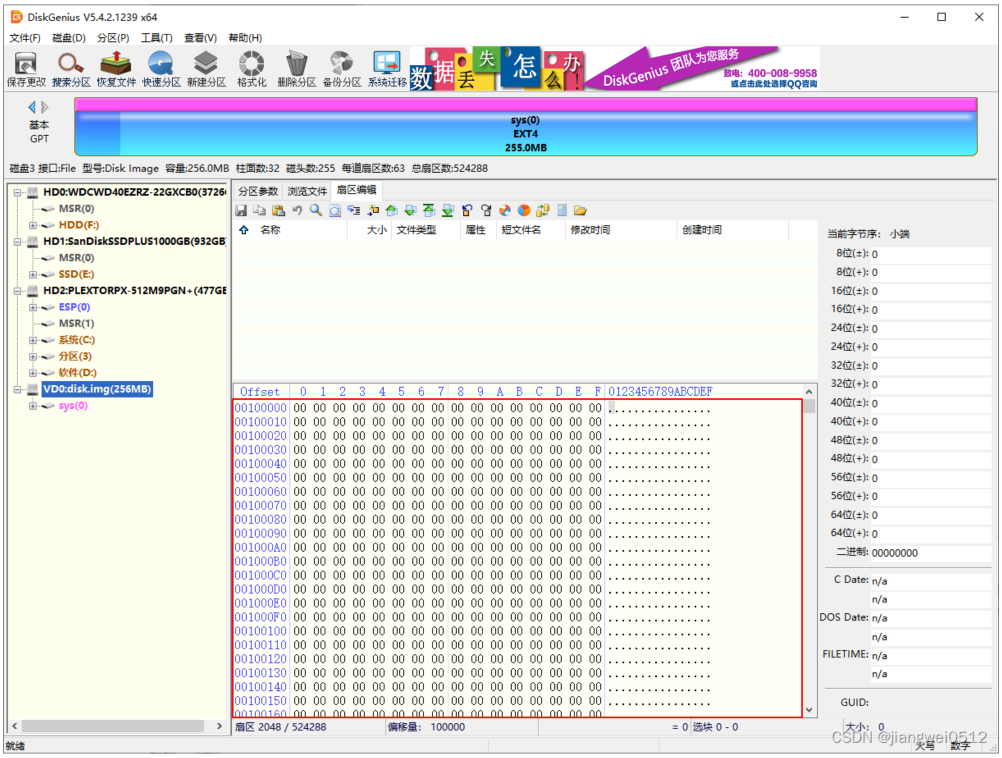
Creating a Test Disk

With the above data structure information, you can write code to read the information of the EXT2 file system, but first you need to create an EXT2 file system, which can be done using DiskGenius:

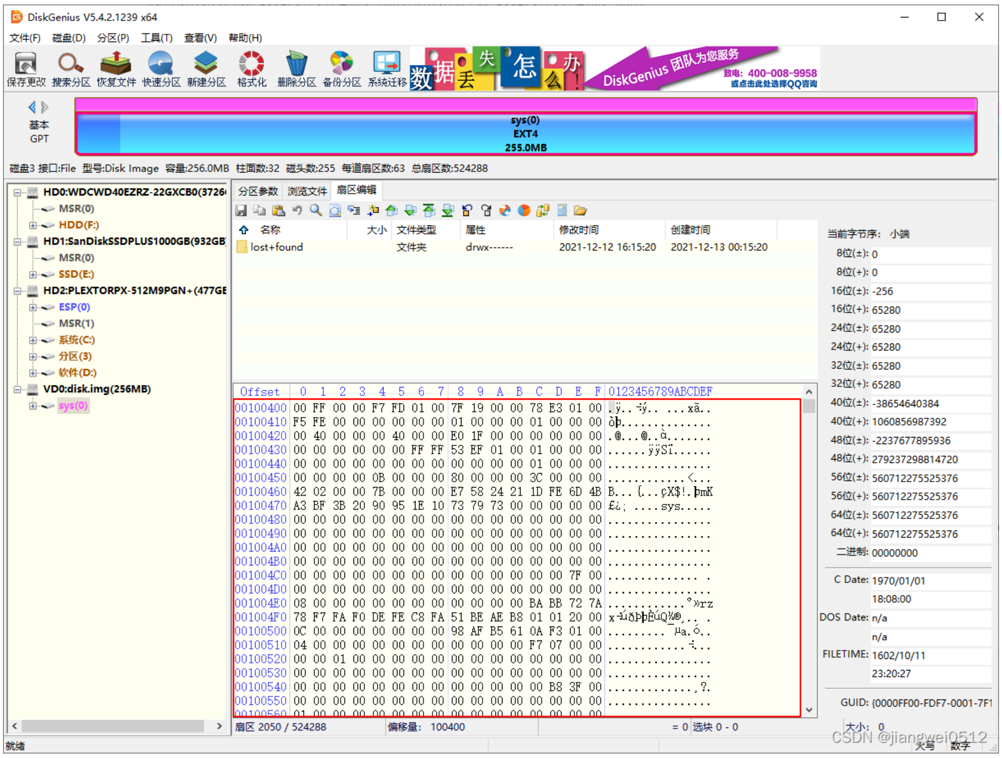


Based on the detailed information of the file system in the figure above and the basics of BIOS, the following information can be determined:

- After the BIOS is started, the disk will be installed `Block IO`, and the partitions on the disk will also be installed `Block IO`. When writing code, you need to pay attention to the partition `Block IO`. Its relative position to the starting position of the disk can be determined by the "starting sector number" to be 2048, plus the "sector size" is 512 bytes, so the `Block IO` storage space described by the partition is offset by 2048 * 512 = 1048576 = 0x100000 bytes for the hard disk. You can view the information of this location from DiskGenius:



- The first block at the beginning of the partition storage space is the boot block, which is temporarily 0 as shown in the figure above. For UEFI BIOS, this content is not required.
- The boot partition occupies one block, and after that is the block group. The first block in the block group is the super block. The size of the boot block is one block (1024 bytes), so the starting position of the super block is 0x100400:



These two addresses can be represented by macros:


```c
#define BBSIZE      1024
#define SBSIZE      1024
#define BBOFF       ((UINTN)(0))
#define SBOFF       ((UINTN)(BBOFF + BBSIZE))
```

Therefore, in order to determine `Block IO` whether there is an EXT2 file system in a certain location, the first step is to obtain 1024 bytes of data from the above location and determine whether it is a super block.

However, DiskGenius cannot write files, so if you want to store files in it, you need to use Linux to create an EXT2 version of the disk.img file. The specific steps are as follows:


```bash
# dd if=/dev/zero of=disk.img count=200 bs=1M
200+0 records in
200+0 records out
209715200 bytes (210 MB, 200 MiB) copied, 0.113229 s, 1.9 GB/s
# mkfs.ext4 disk.img
mke2fs 1.44.1 (24-Mar-2018)
Discarding device blocks: done
Creating filesystem with 204800 1k blocks and 51200 inodes
Filesystem UUID: 513c5206-7726-4dba-8bb1-fbf7b68c449c
Superblock backups stored on blocks:
    8193, 24577, 40961, 57345, 73729

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
# mkdir loop
# mount -o loop disk.img loop/
# cd loop
# ls
lost+found
# mkdir tmp1
# mkdir tmp2
# echo hello1 >> tmp1.log
# echo hello2 >> tmp2.log
# cd ..
# umount loop
```


Here, a 200MB virtual disk disk.img is created, formatted as EXT4 file system, and two directories tmp1 and tmp2, and two files tmp1.log and tmp2.log are added.

Super Block

A normal superblock contains the following data, starting with what EXT2 needs to support:

Starting	Finish	size	describe
0	3	4	Total number of inodes in file system.
4	7	4	Total number of blocks in file system.
8	11	4	Number of reserved blocks.
12	15	4	Total number of unallocated blocks.
16	19	4	Total number of unallocated inodes.
20	twenty three	4	Block number of the block containing the superblock. This is 1 on 1024 byte block size filesystems, and 0 for all others.
twenty four	27	4	*log* 2 (block size) - 10 (In other words, the number to shift 1,024 to the left by to obtain the block size).
28	31	4	*log* 2 (fragment size) - 10 (In other words, the number to shift 1,024 to the left by to obtain the fragment size).
32	35	4	Number of blocks in each block group.
36	39	4	Number of fragments in each block group.
40	43	4	Number of inodes in each block group.

Starting	Finish	size	describe
44	47	4	Last mount time (in POSIX time).
48	51	4	Last written time (in POSIX time).
52	53	2	Number of times the volume has been mounted since its last consistency check (fsck).
54	55	2	Number of mounts allowed before a consistency check (fsck) must be done.
56	57	2	Magic signature (0xef53), used to help confirm the presence of Ext4 on a volume.
58	59	2	File system state.
60	61	2	What to do when an error is detected.
62	63	2	Minor portion of version (combine with Major portion below to construct full version field).
64	67	4	POSIX time of last consistency check (fsck).
68	71	4	Interval (in POSIX time) between forced consistency checks (fsck).
72	75	4	Operating system ID from which the filesystem on this volume was created (see below).
76	79	4	Major portion of version (combine with Minor portion above to construct full version field).
80	81	2	User ID that can use reserved blocks.
82	83	2	Group ID that can use reserved blocks.

Secondly, if you want to support EXT4 dynamic super blocks, there are additional contents:

Starting	Finish	size	describe
84	87	4	First non-reserved inode in file system.
88	89	2	Size of each inode structure in bytes.
90	91	2	Block group that this superblock is part of for backup copies.
92	95	4	Optional features present.
96	99	4	Required features present.
100	103	4	Features that if not supported the volume must be mounted read-only.
104	119	16	File system UUID.
120	135	16	Volume name.
136	199	64	Path Volume was last mounted to.
200	203	4	Compression algorithm used.
204	204	1	Amount of blocks to preallocate for files
205	205	1	Amount of blocks to preallocate for directories.
206	207	2	Amount of reserved GDT entries for filesystem expansion.
208	223	16	Journal UUID.
224	227	4	Journal Inode.
228	231	4	Journal Device number.
232	235	4	Head of orphan inode list.
236	251	16	HTREE hash seed in an array of 32 bit integers.
252	252	1	Hash algorithm to use for directories.
253	253	1	Journal blocks field contains a copy of the inode's block array and size.
254	255	2	Size of group descriptors in bytes, for 64 bit mode.
256	259	4	Mount options.
260	263	4	First metablock block group, if enabled.
264	267	4	Filesystem Creation Time.
268	335	68	Journal Inode Backup in an array of 32 bit integers.

In addition, if 64-bit support is enabled, the content of the super block will be more. The structure data occupies a maximum of 1024 bytes. Generally, the size of the super block is directly set to 1024 bytes in the code. The following data:

Starting	Finish	size	describe
336	339	4	High 32-bits of the total number of blocks.
340	343	4	High 32-bits of the total number of reserved blocks.
344	347	4	High 32-bits of the total number of unallocated blocks.
348	349	2	Minimum inode size.
350	351	2	Minimum inode reservation size.
352	355	4	Misc flags, such as sign of directory hash or development status.
356	357	2	Amount logical blocks read or written per disk in a RAID array.
358	359	2	Amount of seconds to wait in Multi-mount prevention checking.
360	367	8	Block to multi-mount prevent.
368	371	4	Amount of blocks to read or write before returning to the current disk in a RAID array. Amount of disks * stride.
372	372	1	*log* 2 (groups per flex) - 10. (In other words, the number to shift 1,024 to the left by to obtain the groups per flex block group)
373	373	1	Metadata checksum algorithm used. Linux only supports crc32.
374	374	1	Encryption version level.
375	375	1	Reserved padding.
376	383	8	Amount of kilobytes written over the filesystem's lifetime.
384	387	4	Inode number of the active snapshot.
388	391	4	Sequential ID of active snapshot.

Starting	Finish	size	describe
392	399	8	Number of blocks reserved for active snapshot.
400	403	4	Inode number of the head of the disk snapshot list.
404	407	4	Amount of errors detected.
408	411	4	First time an error occurred in POSIX time.
412	415	4	Inode number in the first error.
416	423	8	Block number in the first error.
424	455	32	Function where the first error occurred.
456	459	4	Line number where the first error occurred.
460	463	4	Most recent time an error occurred in POSIX time.
464	467	4	Inode number in the last error.
468	475	8	Block number in the last error.
476	507	32	Function where the most recent error occurred.
508	511	4	Line number where the most recent error occurred.
512	575	64	Mount options. (C-style string: characters terminated by a 0 byte)
576	579	4	Inode number for user quota file.
580	583	4	Inode number for group quota file.
584	587	4	Overhead blocks/clusters in filesystem. Zero means the kernel calculates it at runtime.
588	595	8	Block groups with backup Superblocks, if the sparse superblock flag is set.
596	599	4	Encryption algorithms used, as an array of unsigned char.
600	615	16	Salt for the `string2key` algorithm.
616	619	4	Inode number of the lost+found directory.
620	623	4	Inode number of the project quota tracker.
624	627	4	Checksum of the UUID, used for the checksum seed. (crc32c(~0, UUID))
628	628	1	High 8-bits of the last written time field.
629	629	1	High 8-bits of the last mount time field.
630	630	1	High 8-bits of the Filesystem creation time field.
631	631	1	High 8-bits of the last consistency check time field.
632	632	1	High 8-bits of the first time an error occurred time field.
633	633	1	High 8-bits of the latest time an error occurred time field.
634	634	1	Error code of the first error.
635	635	1	Error code of the latest error.
636	637	2	Filename charset encoding.
638	639	2	Filename charset encoding flags.
640	1019	380	Padding.
1020	1023	4	Checksum of the superblock.

However, we will not discuss the 64-bit feature here, so the corresponding super block structure code is as follows:

c AI generated projects 登录复制 run

```c
//
// EXT2文件系统中的超级块，大小是1024个字节
//
typedef struct {
  UINT32   Ext2FsInodeCount;        //   0: Inode数据
  UINT32   Ext2FsBlockCount;        //   4: 块数据
  UINT32   Ext2FsRsvdBlockCount;    //   8: 已分配块的数据
  UINT32   Ext2FsFreeBlockCount;    //  12: 空闲块数目
  UINT32   Ext2FsFreeInodeCount;    //  16: 空闲Inode数据
  UINT32   Ext2FsFirstDataBlock;    //  20: 第一个数据块
  UINT32   Ext2FsLogBlockSize;      //  24: 块长度
  UINT32   Ext2FsFragmentSize;      //  28: 碎片长度
  UINT32   Ext2FsBlocksPerGroup;    //  32: 每个块组包含的块数
  UINT32   Ext2FsFragsPerGroup;     //  36: 每个块组包含的碎片
  UINT32   Ext2FsInodesPerGroup;    //  40: 每个块组的Inode数据
  UINT32   Ext2FsMountTime;         //  44: 装载时间
  UINT32   Ext2FsWriteTime;         //  48: 写入时间
  UINT16   Ext2FsMountCount;        //  52: 装载计数
  UINT16   Ext2FsMaxMountCount;     //  54: 最大装载计数
  UINT16   Ext2FsMagic;             //  56: 魔数，标记文件系统类型
  UINT16   Ext2FsState;             //  58: 文件系统状态
  UINT16   Ext2FsBehavior;          //  60: 检测到错误时的行为
  UINT16   Ext2FsMinorRev;          //  62: 副修订号
  UINT32   Ext2FsLastFsck;          //  64: 上一次检查的时间
  UINT32   Ext2FsFsckInterval;      //  68: 两次检查允许间隔的最长时间
  UINT32   Ext2FsCreator;           //  72: 创建文件系统的操作系统
  UINT32   Ext2FsRev;               //  76: 修订号
  UINT16   Ext2FsRsvdUid;           //  80: 能够使用保留块的默认UID
  UINT16   Ext2FsRsvdGid;           //  82: 能够使用保留块的默认GID
  //
  // 修订号大于等于1的版本才有下述的数据
  //
  UINT32   Ext2FsFirstInode;        //  84: 第一个非保留的Inode
  UINT16   Ext2FsInodeSize;         //  88: Inode结构的长度
  UINT16   Ext2FsBlockGrpNum;       //  90: 当前超级块所在的块组编号
  UINT32   Ext2FsFeaturesCompat;    //  92: 兼容特性集
  UINT32   Ext2FsFeaturesIncompat;  //  96: 不兼容特性集
  UINT32   Ext2FsFeaturesROCompat;  // 100: 只读兼容特性集
  UINT8    Ext2FsUuid[16];          // 104: 卷的128位UID
  CHAR8    Ext2FsVolumeName[16];    // 120: 卷名
  CHAR8    Ext2FsFSMnt[64];         // 136: 上一次装载的目录
  UINT32   Ext2FsAlgorithm;         // 200: 用于压缩
  UINT8    Ext2FsPreAlloc;          // 204: 试图预分配的块数
  UINT8    Ext2FsDirPreAlloc;       // 205: 试图为目录预分配的块数
```

```c
  UINT16   Ext2FsRsvdGDBlock;       // 206: 为块描述符保留的块
  UINT32   Rsvd2[11];               // 208: 保留
  UINT16   Rsvd3;                   // 252: 保留
  UINT16   Ext2FsGDSize;            // 254: 开启64位模式时组描述符的大小（字节为单位）
  UINT32   Rsvd4[192];              // 256: 保留
} EXT2FS;
```

收起 ^

What is obtained under BIOS `Block IO` may be the disk itself or the partition. For the super block, it is usually on the partition and occupies 1 physical block or 2 physical blocks (because the physical block of the disk may only have 512 bytes, and the super block can be up to 1024 bytes), so the code for obtaining the super block is as follows:

c AI generated projects 登录复制 run

```c
//
// 从磁盘读取超级块，读取方式需要根据硬盘的物理块来确定
// 如果物理块大小是512字节，则指定Buffer大小是1024，即1个超级块的大小，然后读第2个物理块开始的2个物理块
// 如果物理块大小是4096字节，则指定Buffer大小是4096，直接读第1个物理块即可
//
BufferSize = (BlockSize > SBSIZE) ? BlockSize : SBSIZE;
Buffer = AllocatePool (BufferSize);
if (NULL == Buffer) {
  Status = EFI_OUT_OF_RESOURCES;
  goto DONE;
}
Status = MediaReadBlocks (
          Volume->BlockIo,
          SBOFF / BlockSize,
          BufferSize,
          Buffer
          );
if (EFI_ERROR (Status)) {
  DEBUG ((EFI_D_ERROR, "%a MediaReadBlocks failed. - %r\n", __FUNCTION__, Status));
  goto DONE;
}
//
// 读取数据之后需要根据读的Buffer来确定超级块的位置
// 如果读取2个物理块，则Offset是0，读到的数据就是超级块
// 如果读取1个物理块，则这个物理块的offset为1024字节的偏移位置就是超级块的开始
//
SbOffset = (SBOFF < BlockSize) ? SBOFF : 0;
Ext2Fs = (EXT2FS *)(&Buffer[SbOffset]);
```

收起 ^

After obtaining the super block data, you need to determine whether it is a valid super block. The judgment code is:

c AI generated projects 登录复制 run

```c
//
// 超级块首先需要判断魔术字是否正确
//
if (E2FS_MAGIC != Ext2Fs->Ext2FsMagic) {
  Status = EFI_NOT_FOUND;
  goto DONE;
}
//
// 其次需要判断版本信息以及相关的配套数据是否正确
// 1. 如果EXT2的版本是1，则第一个可用Inode是从Inode 11开始的（注意Inode的Index从1开始计数，而不是0）
// 2. 文件系统中的Inode大小必须是固定的128个字节或者256个字节
// 3. 兼容性判断
//
if ((Ext2Fs->Ext2FsRev > E2FS_REV1) ||
    ((E2FS_REV1 == Ext2Fs->Ext2FsRev)  &&
     ((EXT2_FIRSTINO != Ext2Fs->Ext2FsFirstInode) ||
      ((128 != Ext2Fs->Ext2FsInodeSize) && (256 != Ext2Fs->Ext2FsInodeSize)) ||
      (Ext2Fs->Ext2FsFeaturesIncompat & ~EXT2F_INCOMPAT_SUPP)))) {
  Status = EFI_NOT_FOUND;
  goto DONE;
}
```

收起 ^

Group Descriptor

The group descriptor is located immediately after the super block, and its structure is described as follows:

Starting	Finish	size	describe
0	3	4	Low 32bits of block address of block usage bitmap.
4	7	4	Low 32bits of block address of inode usage bitmap.
8	11	4	Low 32bits of starting block address of inode table.
12	13	2	Low 16bits of number of unallocated blocks in group.
14	15	2	Low 16bits of number of unallocated inodes in group.
16	17	2	Low 16bits of number of directories in group.
18	19	2	Block group features present.
20	twenty three	4	Low 32-bits of block address of snapshot exclude bitmap.
twenty four	25	2	Low 16-bits of Checksum of the block usage bitmap.
26	27	2	Low 16-bits of Checksum of the inode usage bitmap.
28	29	2	Low 16-bits of amount of free inodes. This allows us to optimize inode searching.
30	31	2	Checksum of the block group, CRC16 (UUID+group+desc).

If 64-bit support is not enabled, ignore the above `Low 32bits` , because there is no higher bit; if 64-bit support is enabled, there are additional contents:

Starting	Finish	size	describe
32	35	4	High 32-bits of block address of block usage bitmap.
36	39	4	High 32-bits of block address of inode usage bitmap.
40	43	4	High 32-bits of starting block address of inode table.
44	45	2	High 16-bits of number of unallocated blocks in group.
46	47	2	High 16-bits of number of unallocated inodes in group.
48	49	2	High 16-bits of number of directories in group.

Starting	Finish	size	describe
50	51	2	High 16-bits of amount of free inodes.
52	55	4	High 32-bits of block address of snapshot exclude bitmap.
56	57	2	High 16-bits of checksum of the block usage bitmap.
58	59	2	High 16-bits of checksum of the inode usage bitmap.
60	63	4	Reserved as of Linux 5.9rc3.

If you don't care about 64-bit support, the corresponding code is as follows:


```c
//
// EXT2文件系统块描述符
//
typedef struct {
  UINT32   Ext2BGDBlockBitmap;      // 块位图块所在位置
  UINT32   Ext2BGDInodeBitmap;      // Inode位图块所在位置
  UINT32   Ext2BGDInodeTables;      // Inode表块所在位置
  UINT16   Ext2BGDFreeBlocks;       // 空闲块数据
  UINT16   Ext2BGDFreeInodes;       // 空闲Inode数据
  UINT16   Ext2BGDNumDir;           // 目录数据
  UINT16   Rsvd;                    // 保留
  UINT32   Rsvd2[5];                // 保留
  UINT32   Ext2BGDInodeTablesHi;    // 如果支持64位模式，则表示Inode表块所在位置的高32位
  UINT32   Rsvd3[5];                // 保留
} EXT2GD;
```


It should be noted that there is not just one block group descriptor, but a set, so there are multiple structures above. In fact, each block group contains the group descriptors of **all** block groups in the file system. Therefore, from each block group, the following information of all other block groups in the system can be determined:

- Block and Inode bitmap locations
- Location of the Inode table
- Free blocks and Inode data

The number of group descriptors is calculated using the following function (disk size is 200M):


```c
#define HOWMANY(x, y)               (((x)+((y)-1))/(y))

FileSystem->Ext2FsNumCylinder =
    HOWMANY ((FileSystem->Ext2Fs.Ext2FsBlockCount - FileSystem->Ext2Fs.Ext2FsFirstDataBlock),
             FileSystem->Ext2Fs.Ext2FsBlocksPerGroup);
```

`Ext2FsBlockCount` It indicates the number of blocks. Since a block is 1024 bytes (1KB), the corresponding value for a 200M disk is `204800` ; `Ext2FsFirstDataBlock` it indicates the data of the EXT2 file system. Therefore, the super block is the first block, so the value here is `1` ; `Ext2FsBlocksPerGroup` it indicates the number of blocks in a block group. This is fixed when the EXT2 file system is created. The default value is `8192` . The final calculation result is (((x)+((y)-1))/(y)) = (((204800 - 1)+((8192)-1))/(8192)) = 25. From the calculation method, the number of group descriptors is mainly related to the disk size. Finally, it should be noted that the name of the number of group descriptors is `Ext2FsNumCylinder` , which involves the concept of Cylinder, which only appears in ordinary hard disks. It is probably for hard disks that use magnetic heads to address, the design of group descriptors is conducive to faster addressing, and the above calculation method is also for this purpose.

The following macros can be used to calculate the block group where an Inode is located and its position in the block group:


```c
//
// 计算Inode位置所在块组
// 计算Inode在块组中的Index
//
#define INOTOCG(fs, x)              (((x) - 1) / (fs)->Ext2Fs.Ext2FsInodesPerGroup)
#define INODETOFSBO(fs, x)          ((((x) - 1) % (fs)->Ext2Fs.Ext2FsInodesPerGroup) % (fs)->Ext2FsInodesPerBlock)
```

Inode

The location of the Inode table is specified in the group descriptor, from which all Inodes can be found. Its structure is described as follows:

Starting	Finish	size	describe
0	1	2	Type and Permissions (see below)
2	3	2	User ID
4	7	4	Lower 32 bits of size in bytes
8	11	4	Last Access Time (in POSIX time)
12	15	4	Creation Time (in POSIX time)
16	19	4	Last Modification time (in POSIX time)
20	twenty three	4	Deletion time (in POSIX time)
twenty four	25	2	Group ID
26	27	2	Count of hard links (directory entries) to this inode. When this reaches 0, the data blocks are marked as unallocated.
28	31	4	Count of disk sectors (not Ext2 blocks) in use by this inode, not counting the actual inode structure nor directory entries linking to the inode.
32	35	4	Flags (see below)
36	39	4	Operating System Specific value #1
40	43	4	Direct Block Pointer 0
44	47	4	Direct Block Pointer 1
48	51	4	Direct Block Pointer 2
52	55	4	Direct Block Pointer 3
56	59	4	Direct Block Pointer 4
60	63	4	Direct Block Pointer 5
64	67	4	Direct Block Pointer 6
68	71	4	Direct Block Pointer 7
72	75	4	Direct Block Pointer 8

Starting	Finish	size	describe
76	79	4	Direct Block Pointer 9
80	83	4	Direct Block Pointer 10
84	87	4	Direct Block Pointer 11
88	91	4	Singly Indirect Block Pointer (Points to a block that is a list of block pointers to data)
92	95	4	Doubly Indirect Block Pointer (Points to a block that is a list of block pointers to Singly Indirect Blocks)
96	99	4	Triply Indirect Block Pointer (Points to a block that is a list of block pointers to Doubly Indirect Blocks)
100	103	4	Generation number (Primarily used for NFS)
104	107	4	In Ext2 version 0, this field is reserved. In version >= 1, Extended attribute block (File ACL).
108	111	4	In Ext2 version 0, this field is reserved. In version >= 1, Upper 32 bits of file size (if feature bit set) if it's a file, Directory ACL if it's a directory
112	115	4	Block address of fragment
116	127	12	Operating System Specific Value #2

It can be seen that some members are different according to different operating systems, the corresponding codes are:

c AI generated projects 登录复制 run

```c
typedef struct {
  UINT16    Ext2DInodeMode;                    //   0: 文件模式，保存了访问权限和文件类型（目录，设备文件等）
  UINT16    Ext2DInodeUid;                     //   2: 所有者UID的低16位
  UINT32    Ext2DInodeSize;                    //   4: 文件长度，以字节为单位
  UINT32    Ext2DInodeAcessTime;               //   8: 上一次访问文件的时间戳
  UINT32    Ext2DInodeCreatTime;               //  12: 上一次修改Inode的时间戳
  UINT32    Ext2DInodeModificationTime;        //  16: 上一次修改文件的时间戳
  UINT32    Ext2DInodeDeletionTime;            //  20: 文件删除的时间戳
  UINT16    Ext2DInodeGid;                     //  24: 组ID的低16位
  UINT16    Ext2DInodeLinkCount;               //  26: 指向Inode的硬链接的数目
  UINT32    Ext2DInodeBlockCount;              //  28: 文件长度，以块为单位
  UINT32    Ext2DInodeStatusFlags;             //  32: 文件标志
  UINT32    Ext2DInodeLinuxRsvd1;              //  36: 保留
  UINT32    Ext2DInodeBlocks[NDADDR + NIADDR]; //  40: 块指针（块号），指向文件数据块的指针保存在这里，
                                               //      默认情况下数组元素个数是12+3，前12个寻址直接块，
                                               //      后3个用于实现简单、二次和三次间接
  UINT32    Ext2DInodeGen;                     // 100: 文件版本，用户NFS
  UINT32    Ext2DInodeFileAcl;                 // 104: 文件ACL（访问控制表）
  UINT32    Ext2DInodeDirAcl;                  // 108: 目录ACL（访问控制表）
  UINT32    Ext2DInodeFragmentAddr;            // 112: 碎片地址（未使用）
  UINT8     Ext2DInodeFragmentNum;             // 116: 碎片编号（未使用）
  UINT8     Ext2DInodeFragmentSize;            // 117: 锁片长度（未使用）
  UINT16    Ext2DInodeLinuxRsvd2;              // 118: 保留
  UINT16    Ext2DInodeUidHigh;                 // 120: 所有者UID的高16位
  UINT16    Ext2DInodeGidHigh;                 // 122: 组ID的高16位
  UINT32    Ext2DInodeLinuxRsvd3;              // 124: 保留
} EXTFS_DINODE;
```

收起 ^

Possible values of file type (BIT12 ~ BIT15):

Type value in hex	Type Description
0x1000	FIFO
0x2000	Character device
0x4000	Directory
0x6000	Block device
0x8000	Regular file
0xA000	Symbolic link
0xC000	Unix socket

Possible values of access rights (BIT0 ~ BIT11):

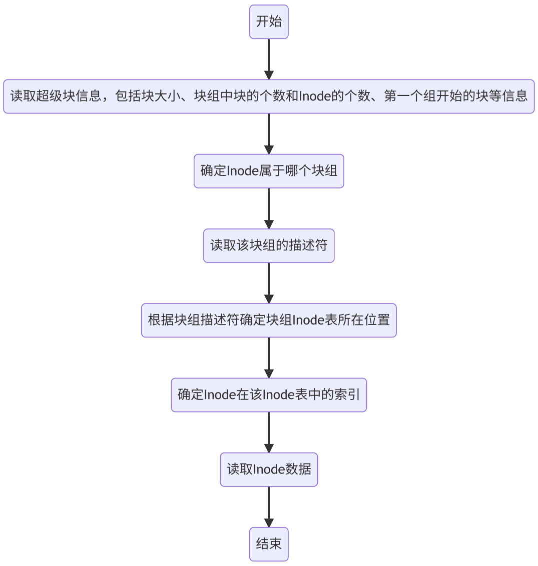
Permission value in hex	Permission value in octal	Permission Description
0x001	00001	Other—execute permission
0x002	00002	Other—write permission
0x004	00004	Other—read permission
0x008	00010	Group—execute permission
0x010	00020	Group—write permission
0x020	00040	Group—read permission
0x040	00100	User—execute permission
0x080	00200	User—write permission
0x100	00400	User—read permission
0x200	01000	Sticky Bit
0x400	02000	Set group ID
0x800	04000	Set user ID

The size of the inode varies depending on the version of EXT2. The specific calculation method is as follows:

c AI generated projects 登录复制 run

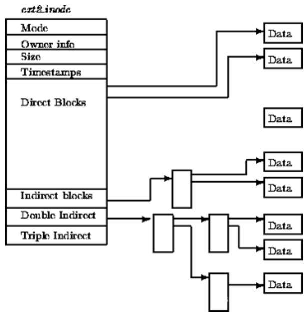
```c
//
// 计算INODE结构体的大小，跟EXT2版本有关：
//   EXT2版本0中Inode的大小是128个字节；
//   EXT2版本1开始，Inode的大小通过超级块中的成员指定；
//
#define EXT2_REV0_DINODE_SIZE       sizeof(EXTFS_DINODE)
#define EXT2_DINODE_SIZE(fs)        ((fs)->Ext2Fs.Ext2FsRev > E2FS_REV0 ?    \
                                     (fs)->Ext2Fs.Ext2FsInodeSize :    \
                                     EXT2_REV0_DINODE_SIZE)
```

The process of reading Inode is as follows:



The file metadata stored in the Inode structure can be associated with the file content located in the disk data block part. The association between the two is established by storing the address of the data block in the Inode. It should be noted that the data blocks are not necessarily continuous .

The address of the data block is stored in `Ext2DInodeBlocks` , but `Ext2DInodeBlocks` there are only 15 32-bit data in total, and the size of a data block may be only 1024 bytes, so if you specify it directly, you will find that the data is obviously not enough. Here, direct and indirect (there are several levels of indirect) methods are used to address the data block, as shown in the following figure:



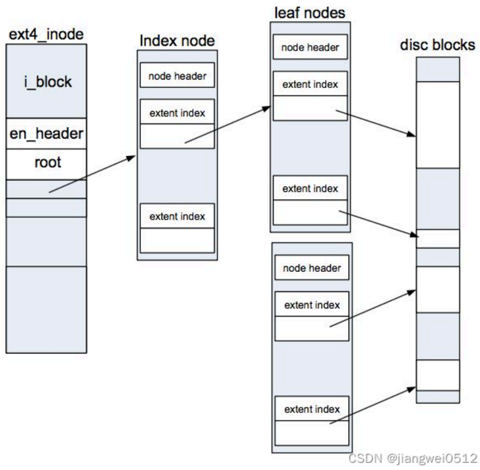
Direct addressing is only used for small files; if the file is larger, indirect addressing will be used. At this time, the file system allocates a data block on the disk, does not store the file, and is specifically used to store the block number. In this case, assuming that a block size is 1024 bytes, and 32 bits represent a data block address, 256 pointers can be stored, corresponding to 256 database blocks, and the data size that can be stored is 256x1024=256K. If it is double indirect, the data size that can be stored is 256x256x1024=65536K. If it is triple indirect, the data size that can be stored is 256x256x256x1024=16777216KB=16G, that is, the maximum supported size for a single file is 16GB.

There is no need to introduce direct addressing. There are two ways of indirect addressing, one is EXT2 itself, and the other is EXT4 extension. Here we mainly introduce EXT4 extension.

The EXT4 extended version uses a new structure to represent it, called the EXT4 extended index tree. At this time, `Ext2DInodeBlocks` the (60 bytes) in the Inode structure is no longer an array, but represents an index tree structure (also 60 bytes in size):


```c
#define EXT4_MAX_HEADER_EXTENT_ENTRIES   4
#define EXT4_EXTENT_HEADER_MAGIC         0xF30A
//
// 一个间接块可以表示的数据块，因为数据块寻址地址的大小是UINT32，就是块大小除以UINT32的大小
//
#define NINDIR(fs)                  ((fs)->Ext2FsBlockSize / sizeof(UINT32))
//
// EXT4扩展版的间接寻址
//
typedef struct {
  UINT16    EhMagic;      // 魔数0xF30A
  UINT16    EhEntries;    // 当前节点中有效entry的数目
  UINT16    EhMax;        // 当前节点中entry的最大数目
  UINT16    EhDepth;      // 当前节点在树中的深度
  UINT32    EhGen;        // 索引树版本
} EXT4_EXTENT_HEADER;

typedef struct {
  UINT32    EiBlk;        // 当前节点块的索引
  UINT32    EiLeafLo;     // 物理块指针低位
  UINT16    EiLeafHi;     // 物理块指针高位
  UINT16    EiUnused;
} EXT4_EXTENT_INDEX;

typedef struct {
  UINT32    Eblk;         // 当前节点的第一个块位置
  UINT16    Elen;         // 块数目
  UINT16    EstartHi;     // 物理块指针高位
  UINT32    EstartLo;     // 物理块指针低位
} EXT4_EXTENT;
//
// 跟Inode中的Ext2DInodeBlocks大小一样，都是64个字节
//
typedef struct {
  EXT4_EXTENT_HEADER Eheader;
  union {
    EXT4_EXTENT_INDEX  Eindex[EXT4_MAX_HEADER_EXTENT_ENTRIES];
    EXT4_EXTENT        Extent[EXT4_MAX_HEADER_EXTENT_ENTRIES];
  } Enodes;
} EXT4_EXTENT_TABLE;
```


The entire indexing process is shown in the following figure (picture from "Addition of Ext4 Extent and Ext3 HTree DIR Read-Only Support in NetBSD"):



The loop code for the index tree is roughly as follows:

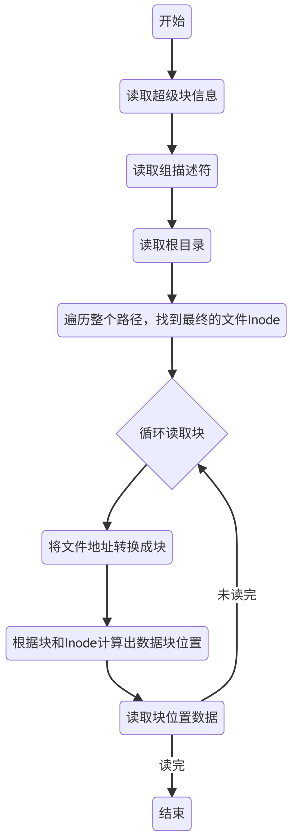
c   AI generated projects   登录复制   run

```c
while (Etable->Eheader.EhDepth > 0) {
  ExtIndex = NULL;
  for (Index=1; Index < Etable->Eheader.EhEntries; Index++) {
    ExtIndex = &(Etable->Enodes.Eindex[Index]);
    if (((UINT32) FileBlock) < ExtIndex->EiBlk) {
      ExtIndex = &(Etable->Enodes.Eindex[Index-1]);
      break;
    }
    ExtIndex = NULL;
  }

  if (NULL != ExtIndex) {
    //
    // 目前还不支持48位
    //
    ASSERT (ExtIndex->EiLeafHi == 0);
    NextLevelNode = ExtIndex->EiLeafLo;

    //
    // 继续深入读取下一个间接块
    //
    Status = MediaReadBlocks (
               File->BlockIo,
               FSBTODB (Fp->FsPtr, (DADDRESS) NextLevelNode),
               FileSystem->Ext2FsBlockSize,
               Buf
               );
    if (EFI_ERROR (Status)) {
      DEBUG ((EFI_D_ERROR, "%a MediaReadBlocks failed. - %r\n", __FUNCTION__, Status));
      return Status;
    }

    Etable = (EXT4_EXTENT_TABLE*) Buf;
    if (Etable->Eheader.EhMagic != EXT4_EXTENT_HEADER_MAGIC) {
        DEBUG ((DEBUG_ERROR, "EXT4 extent header magic mismatch 0x%X!\n", Etable->Eheader.EhMagic));
        return EFI_DEVICE_ERROR;
    }
  } else {
    DEBUG ((DEBUG_ERROR, "Could not find FileBlock #%d in the index extent data!\n", FileBlock));
    return EFI_NO_MAPPING;
  }
}
```

收起

File Reading

One file corresponds to one Inode, and Inode corresponds to data. However, according to the previous introduction, there are two ways to correspond between Inode and data: direct and indirect. Therefore, in order to read the entire file, you need to cut the file into blocks, and then each block corresponds to a file index. Through this file block index combined with the addressing data in the Inode, you can finally find the corresponding data block and read the entire data block. This cycle continues until the entire file is read out.



Code Sample

The corresponding code implementation has been uploaded to https://gitee.com/jiangwei0512/edk2-beni . The command execution results are as follows: