哈希树详解

目录条目的线性数组对性能影响较大，因此 ext3 增加了一项新功能，以提供更快（但特殊）的平衡树，该树以目录条目名称的哈希值为关键。

通常情况下，Ext4 文件系统中的目录最初采用无索引方式存储目录条目。在这种无索引模式下，目录条目按顺序存储在目录的单个数据块中。当目录条目的数量包含在单个数据块中时，这种方法是有效的。
但是，一旦目录条目的数量超过了单个数据块的容量，目录就可以转换为索引树结构，即散列树。启用哈希树功能后，这种转换就会发生。

如果在 inode 中设置了 EXT4_INDEX_FL (0x1000) 标志，该目录就会使用散列 btree（htree）来组织和查找目录条目。为实现与 ext2 的向后只读兼容性，该树实际上隐藏在目录文件中，伪装成 "空 "目录数据块！

哈希树结构包括根节点、中间节点和叶节点。

如果目录没有索引，其数据块将存储 dirents（目录条目）。但是，如果目录有索引（即使用哈希树），前几个数据块将存储与哈希树结构相关的信息，而其余数据块将存储 dirents。在哈希树的情况下，第 1 个数据块是哈希树根的主机，由`struct dx_root`表示，其后是`struct dx_entry`实例。这些`dx_entry`结构提供哈希树层次结构中下一级数据块的地址。

前面说过，线性目录条目表的末尾有一个指向 inode 0 的条目；这（滥用）是为了欺骗旧的线性扫描算法，让它以为目录块的其余部分是空的，从而继续前进。

树根始终位于目录的第一个数据块中。根据 ext2 的习惯，". "和".. "条目必须出现在第一个数据块的开头，因此它们会以两个`struct ext4_dir_entry_2`的形式放在这里，而不会存储在树中。根节点的其余部分包含有关树的元数据，最后是一个 hash-> block 映射，用于查找 htree 中低一层的节点。

如果`dx_root.info.indirect_levels`非零，那么 htree 有两层；根节点映射指向的数据块是一个内部节点，由一个次要哈希值索引。该树的内部节点包含一个清零的`struct ext4_dir_entry_2`，然后是一个 minor_hash->block 映射，用于查找叶节点。

叶节点包含一个由所有`struct ext4_dir_entry_2`组成的线性数组；所有这些条目（大概）的哈希值相同。如果出现溢出，这些条目就会溢出到下一个叶节点，并设置进入下一个叶节点的哈希值（在内部节点映射中）的最小有效位。

要以 htree 的形式遍历目录，代码会计算所需文件名的哈希值，并用它来查找相应的块编号。如果树是扁平的，则块是一个可搜索的线性目录条目数组；否则，将计算文件名的次要哈希值，并根据第二个块找到相应的第三个块编号。第三个块编号将是一个线性目录条目数组。

要以线性数组的形式遍历目录（如旧代码），代码只需读取目录中的每个数据块即可。用于 htree 的数据块看起来没有条目（除了". "和".."），因此只有叶节点看起来有任何有趣的内容。

htree 的根节点位于`结构体 dx_root`中，它是一个数据块的全长(占据一整个数据块)

Offset	Type	Name	Description
0x0	__le32	dot.inode	inode number of this directory.
0x4	__le16	dot.rec_len	Length of this record, 12.

Offset	Type	Name	Description
0x6	u8	dot.name_len	Length of the name, 1.
0x7	u8	dot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).
0x8	char	dot.name[4]	".000"
0xC	__le32	dotdot.inode	inode number of parent directory.
0x10	__le16	dotdot.rec_len	block_size - 12. The record length is long enough to cover all htree data.
0x12	u8	dotdot.name_len	Length of the name, 2.
0x13	u8	dotdot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).
0x14	char	dotdot_name[4]	"..00"
0x18	__le32	struct dx_root_info.reserved_zero	Zero.
0x1C	u8	struct dx_root_info.hash_version	Hash type, see dirhash table below.
0x1D	u8	struct dx_root_info.info_length	Length of the tree information, 0x8.
0x1E	u8	struct dx_root_info.indirect_levels	Depth of the htree. Cannot be larger than 3 if the INCOMPAT_LARGEDIR feature is set; cannot be larger than 2 otherwise.
0x1F	u8	struct dx_root_info.unused_flags	
0x20	__le16	limit	Maximum number of dx_entries that can follow this header, plus 1 for the header itself.
0x22	__le16	count	Actual number of dx_entries that follow this header, plus 1 for the header itself.
0x24	__le32	block	The block number (within the directory file) that goes with hash=0.
0x28	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

The directory hash is one of the following values:

Value	Description
0x0	Legacy.
0x1	Half MD4.
0x2	Tea.
0x3	Legacy, unsigned.
0x4	Half MD4, unsigned.
0x5	Tea, unsigned.
0x6	Siphash.

Interior nodes of an htree are recorded as `struct dx_node`, which is also the full length of a data block:(占据一个block)

Offset	Type	Name	Description
0x0	__le32	fake.inode	Zero, to make it look like this entry is not in use.
0x4	__le16	fake.rec_len	The size of the block, in order to hide all of the dx_node data.
0x6	u8	name_len	Zero. There is no name for this “unused” directory entry.
0x7	u8	file_type	Zero. There is no file type for this “unused” directory entry.
0x8	__le16	limit	Maximum number of dx_entries that can follow this header, plus 1 for the header itself.
0xA	__le16	count	Actual number of dx_entries that follow this header, plus 1 for the header itself.
0xE	__le32	block	与该数据块最低哈希值对应的数据块编号（在目录文件中）。该值存储在父块中。
0x12	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

`结构 dx_root` 和 `结构 dx_node` 中的哈希映射都记录为 `结构 dx_entry`，长度为 8 字节：

Offset	Type	Name	Description
0x0	__le32	hash	Hash code.
0x4	__le32	block	htree 中下一个节点的块编号（目录文件中的块0，1，2.....，而非文件系统块）。

对于dx_root所在的块中：

前 36 个字节由 `dx_root` 使用，在 `dx_root` 之后有 28 个 `dx_entrys`。我们可以看到，考虑到 28 个 `dx_entrys` 和 `dx_root` 本身，`dx_root` 中的有效条目数为 29 个。所提供的 hexdump 中突出显示了前三个 `dx_entrys` 条目。此外，树的深度显示为 0，表示它是哈希树结构中的叶子层。

在 `dx_entry` 中，前 4 个字节表示散列值，后 4 个字节表示逻辑块编号。该逻辑块对应一个数据块，用于存储哈希值介于当前 `dx_entry` 的哈希值和哈希树结构中下一个 `dx_entry` 的哈希值之间的目录条目（dirents）。

下表列出了前 3 个 `dx_entry` 的哈希值和逻辑块：

Hash value	Logical block number
Zero Hash	0x01
0x082e0162	0x12
0x103050d2	0x09
0x17358dcc	0x18

举个例子，我们要添加一个哈希值为 0x092e0111 的 dirent。通过将哈希值与上表中的哈希值进行比较，我们发现哈希值大于 0x082e0162，小于 0x103050d2。因此，目录条目将被插入逻辑块 12，该逻辑块对应的数据块存储了哈希值介于 0x082e0162 和 0x103050d2 之间的目录。

同样，如果我们要搜索一个哈希值大于 0 但小于 0x082e0162 的 dirent，则需要搜索块 1 来查找目标目录条目。另一方面，如果目标 dirent 的哈希值大于 0x103050d2 且小于 0x17358dcc，我们就需要搜索块 9 来找到所需的 dirent。通过这一过程，我们可以根据哈希树结构中的哈希值找到目录条目。

目录项　dirent structure which is stored by the directory in its data block:

```
struct ext4_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT4_NAME_LEN];    /* File name */
};
```

当深度大于 0 时：

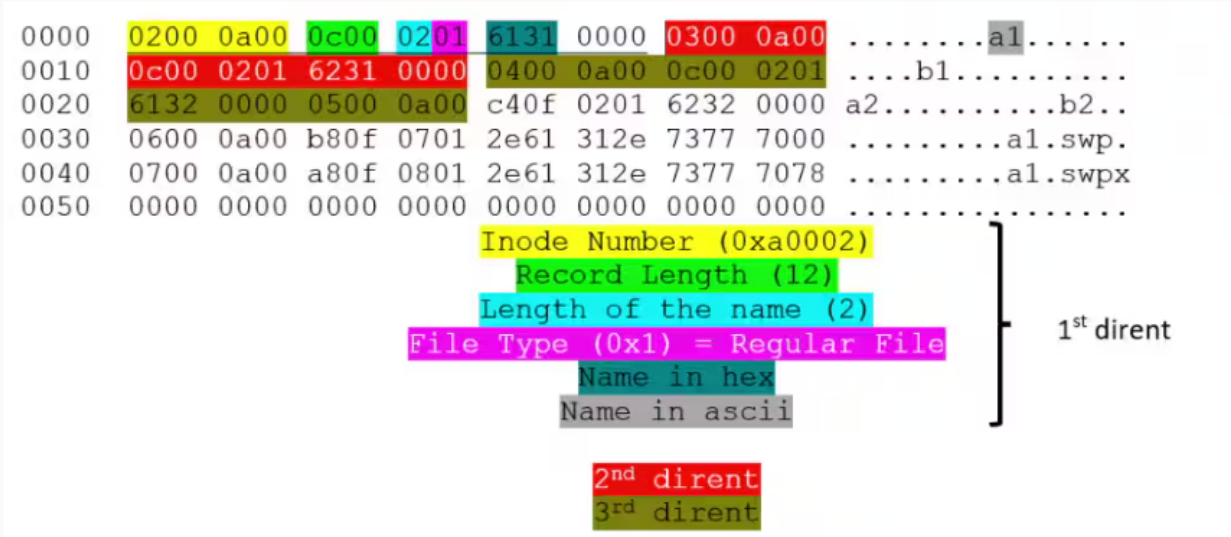
在哈希树深度为 2 的情况下，在叶子块（存储目录条目）和`dx_root`块之间会有两层中间块。这些中间块专门存储`dx_entry`块。

在哈希树中搜索文件时，我们只检查每层的一个块，直到到达叶子块。在叶子区块，我们执行线性搜索，以找到所需的 dirent。与在目录的所有块中进行线性搜索的无索引目录相比，这种分层搜索过程大大提高了效率。

不同的哈希算法：

Value	Description
0x0	Legacy.
0x1	Half MD4.
0x2	Tea.
0x3	Legacy, unsigned.
0x4	Half MD4, unsigned.
0x5	Tea, unsigned.
0x6	Siphash.

Below is the hexdump of one of the leaf blocks in the hash tree:



`a1`、`b1`、`a2`、`b2`等是目录中的文件，因此它们的地址都存储在目录的数据块中。

让我们仔细看看目录中文件 a1 的 dirent。dirent 的结构如下：

1. 前四个字节表示文件的inode number。
2. 接下来的两个字节表示 dirent 的记录长度。
3. 随后的字节表示file name的长度。
4. 接下来是文件类型，提供文件类型的相关信息（如普通文件、目录、符号链接等）。

5. 最后是文件的实际名称。

最后一个 dirent 的`rec_len`将跨到块的末尾。

通过这种结构，我们可以从存储在目录数据块中的 dirent 中获取文件的重要详细信息，如 inode 编号、长度、类型和名称。

下表列出了文件类型：

Value	File type
0x0	Unknown
0x1	Regular file
0x2	Directory
0x3	Character device file
0x4	Block device file
0x5	FIFO
0x6	Socket
0x7	Symbolic link

哈希树的建树过程

随着目录中 dirents 数量的增加，用于存储这些 dirents 的树的大小和深度也会增加。

起初，目录处于无索引状态，dirents存储在单个块中。但是，如果 dirents 的数量超过了单个块的容量，目录就会转换为索引树结构。在转换过程中，会创建两个新块。其中一个块将成为根块，并使用`dx_root`结构进行初始化。然后按升序对 dirents 进行排序，前半部分 dirents 保留在原始块中，而剩余的 dirents 则移到另一个新块中。两个`dx_entry`结构被添加到`dx_root`，以保持目录的索引状态。

假设目录最初未编入索引，包含 40 个 dirents，目录的数据块已完全满载。当添加第 41 个 dirent 时，目录将转换为索引结构。

在转换过程中，会创建两个新块。其中一个块成为根块，40 个字段被排序。前 20 个 dirents 保留在原始块中，而其他 20 个 dirents 则被移到新块中。新区块的`dx_entry`的哈希值等于该区块中第一个 dirent 的哈希值。另一方面，初始数据块的`dx_entry`的哈希值为 0。

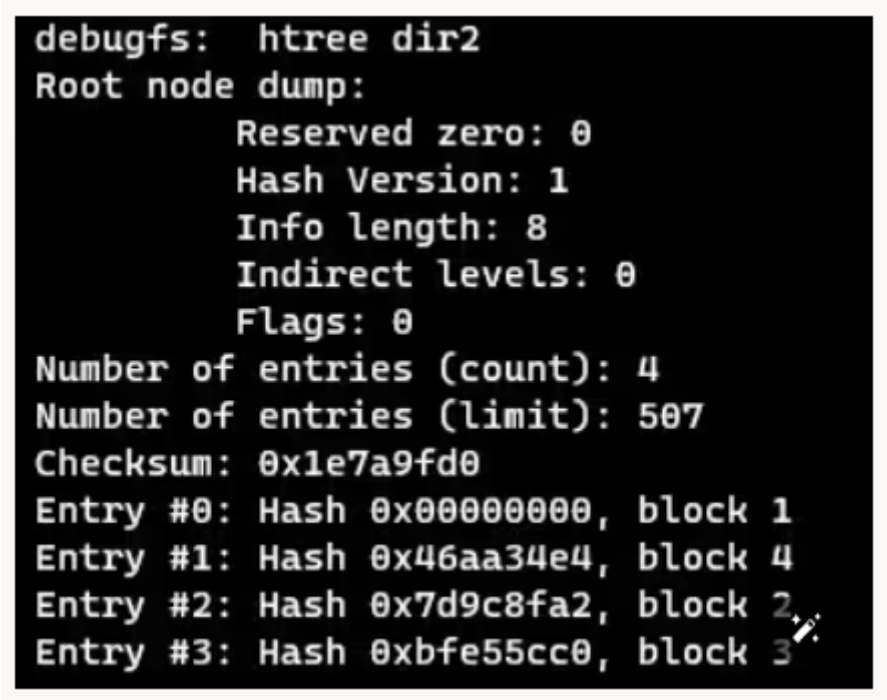
经过这样的重组，目录的索引结构中就可以容纳新增的第 41 个 dirent 了。

将目录转换为索引结构后，如果有任何叶块被 dirents 填满，就会创建一个新块来容纳新增的 dirents。已填满的叶块中的 dirents 将在已填满的块和新块之间重新平均分配。

这种重新分配可确保 dirents 在叶子区块之间均匀分布，防止任何一个区块的 dirents 超载。通过创建新块并重新分配字典，目录的索引结构可以有效地存储和管理更多字典，同时保持块的均衡使用。

举例说明 下图描述了目录哈希树的初始配置。此时，散列树的深度为 0，由三个叶块（1、2 和 3）组成。

现在，我们考虑在同一目录下创建 20 个新文件的情况。添加这些新文件后，哈希树配置更新如下：



哈希树的深度保持不变，表明不需要增加层级。不过，在哈希树中的区块 1 和区块 2 之间创建并插入了一个新的叶子区块（区块 4）。这是必要的，因为一些新添加的条目完全填满了块 1。为了容纳新增的字段，又创建了一个新块（块 4），并在块 1 和块 4 之间均匀地重新分配字段，以保持哈希树结构的平衡。

碰撞

在发生哈希值碰撞时，需要注意的是，哈希值主要用于定位特定哈希值范围内的字典块。如果发生碰撞，且多个 dirents 共享相同的哈希值，它们将被一起存储在同一区块中。一旦找到目标数据块，就会在该数据块内进行线性搜索，以找到特定的 dirent。

在添加新的 dirent 时，如果该 dirent 与完全填满的区块中的 dirent 发生哈希碰撞，则会创建一个新的区块。这些 dirent 将根据哈希值在两个区块之间重新平均分配。不过，值得注意的是，具有相同哈希值的 dirent 仍会保留在同一区块中。这就确保了尽管进行了重新分配，但具有碰撞哈希值的数据字段仍会保留在同一数据块中（按照hash值升序排列的），以便在线性搜索操作中进行高效检索。

总结：哈希树方案有助于快速高效地搜索、添加和删除目录条目（dirents）。这一功能对于容纳大量文件的目录尤为重要，因为它能最大限度地减少线性搜索的需要，确保快速访问所需的目录。

rmdir只需要将d_entry删除即可，树的结构不用改变

删除

由于目录项的长度不固定，可以直接合并 rec_len 目录项长度

```
struct ext4_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT4_NAME_LEN];    /* File name */
};
```

通过合并目录项可以优化文件系统的布局和性能，而不是为了增加空间。这种优化可以提高文件系统的整体效率

```
//将该目录项与前一个目录项合并，以确保文件系统中没有未使用的空间。
int ext4_generic_delete_entry(struct inode *dir,
                  struct ext4_dir_entry_2 *de_del,
                  struct buffer_head *bh,
                  void *entry_buf,
                  int buf_size,
                  int csum_size)
{
    struct ext4_dir_entry_2 *de, *pde;//pde跟踪前一个目录项
    unsigned int blocksize = dir->i_sb->s_blocksize;
    int i;

    i = 0;
    pde = NULL;
    de = (struct ext4_dir_entry_2 *)entry_buf;//d_entry的起始位置
    while (i < buf_size - csum_size) {
        if (ext4_check_dir_entry(dir, NULL, de, bh,
                     entry_buf, buf_size, i))
            return -EFSCORRUPTED;
        if (de == de_del)  {
            if (pde) {
                //则将前一个目录项的记录长度（rec_len）增加当前目录项的记录长度，以实现合并
                pde->rec_len = ext4_rec_len_to_disk(
```

```
                    ext4_rec_len_from_disk(pde->rec_len,
                                blocksize) +
                    ext4_rec_len_from_disk(de->rec_len,
                                blocksize),
                    blocksize);

                /* wipe entire dir_entry */
                //rec_len代表长度
                memset(de, 0, ext4_rec_len_from_disk(de->rec_len,
                                blocksize));
            } else {
                /* wipe dir_entry excluding the rec_len field */
                de->inode = 0;
                memset(&de->name_len, 0,
                    ext4_rec_len_from_disk(de->rec_len,
                                blocksize) -
                    offsetof(struct ext4_dir_entry_2,
                                name_len));
            }

            inode_inc_iversion(dir);
            return 0;
        }
        i += ext4_rec_len_from_disk(de->rec_len, blocksize);
        pde = de;
        de = ext4_next_entry(de, blocksize);
    }
    return -ENOENT;
}
```

目录项的长度实际上不固定

```
static inline struct ext4_dir_entry_2 *
ext4_next_entry(struct ext4_dir_entry_2 *p, unsigned long blocksize)
{
    return (struct ext4_dir_entry_2 *)((char *)p +
        ext4_rec_len_from_disk(p->rec_len, blocksize));
}
```