3.4.5. inode 相关问题

3.4.5.1. inode table

这一节将要分享一下,当一个文件或者目录被创建的时候，在 glusterfs 中 inode_t 结构会经历哪些过程，而为什么要关注这点呢？因为在文件系统中，因为文件句柄或者 inode 等导致的异常很容易出现严重的 bug，而 glusterfs 的历史版本中，在 7.9 以前关于 open-behind 参数有一个重大的 bug,就是因为统计文件引用出错，文件引用数据为 0，但是实际上文件正在被使用，而导致挂载进程被清理的异常。当然，关于这个问题，会在后面分享一下本人在生产环境使用中遇到的一些版本 bug，希望能引起注意。同时，了解这些内容，对于以后分析内存泄露等问题，也有非常大的帮助。

首先这里先要介绍一下 inode table，所有的 inode 被进程创建之后，都要关联到 inode-table 中，而在一般的顶级的 xlators 功能里面，例如 fuse/server/NFS/gfapi 等不同功能模块中，都会创建一个 inode tables 给它的子集,用于管理 inode，只要进程还存活的，那么 inode table 就一直存在。下面就先简单来了解一下 glusterfs 中的 inode_table 结构吧

```
1.  struct _inode_table {
2.      pthread_mutex_t lock;
3.      size_t dentry_hashsize;
4.      size_t inode_hashsize;
5.      char *name;
6.      inode_t *root;
7.      //调用 xlator 进行清除
8.      xlator_t *xl;
```

```
9.      // lru 缓存最大值
10.     uint32_t lru_limit;
11.     // inode hash 表
12.     struct list_head *inode_hash;
13.     // dentry hash 表
14.     struct list_head *name_hash;
15.     // 活跃的 inode 信息记录表
16.     struct list_head active;
17.     // 记录活跃的 inode 数量
18.     uint32_t active_size;
19.     struct list_head lru;
20.
21.     uint32_t lru_size;
22.     struct list_head purge;
23.     uint32_t purge_size;
24.
25.      //下面是和内存池有关的
26.     struct mem_pool *inode_pool;
27.     struct mem_pool *dentry_pool;
28.     struct mem_pool *fd_mem_pool;
29.     int ctxcount;
30.
31.     //这里是和 inode invalidation 操作有关的
32.     int32_t (*invalidator_fn)(xlator_t *, inode_t *);
33.     xlator_t *invalidator_xl;
34.     struct list_head invalidate;
35.     uint32_t invalidate_size;
36.
37.
38.     //用于标识清理 inode table 操作 是否启动
39.     gf_boolean_t cleanup_started;
40. };
```

通过这里的结构成员信息，可以知道主要就是记录活跃的 inode 信息，还有将要销毁清理的 inode 信息等内容。注意，在关于 inode 这里，主要是三

个重要的东西，分别是 nlookup,ref 和 invalidation。

那么这里 nlookup 到底是什么呢？这个在前面的 inode_t 结构介绍中出现过，nlookup 维护内核查找 inode 的次数。当 fuse 内核认为不再需要 inode 时，将会调用 FUSE_FORGET 或者 FUSE_BATCH_FORGET 进行处理。

这里 FUSE_FORGET 是定义在 fuse-bridge.c 文件中的 fuse_handler_t operations ,这些是内核 fuse 和 glusterfs 之间接口的一部分。简单地来理解，可以理解成为，当内核要清理这个 inode 的时候的回调函数，类似 unref 在 gluster 中的实现一样。这里又是怎么理解呢？其实对于 glusterfs 来说 ,因为是一个自定义的 fuse ,因为可能 inode 会被接口调用释放掉，例如文件被删掉这样的，这时候就是系统自行触发的，但是也有一种可能，就类似底层操作系统释放掉了这个 inode，这时候就要通知到上层的 glusterfs 了，这就是 forget 这个功能存在的意义了，当然这部分内容比较绕，需要仔细了解一下，后面也会不断讲解分析。

那么这里 inode_table 是如何创建的呢？这里代码在 libglusterfs/src/inode.c 中的 inode_table_new 函数，这里会调用 inode_table_with_invalidator 函数，然后调用 __inode_table_init_root,下面先来看看代码的一些内容。

```
1.  inode_table_t *
2.  inode_table_with_invalidator(uint32_t lru_limit, xlator_t
    *xl,
3.                               int32_t (*invalidator_fn)(xl
   ator_t *, inode_t *),
4.                               xlator_t *invalidator_xl, ui
   nt32_t dentry_hashsize,
```

```
5.                                      uint32_t inode_hashsize)
6.  {
7.      inode_table_t *new = NULL;
8.      uint32_t mem_pool_size = lru_limit;
9.      int ret = -1;
10.     int i = 0;
11.
12.     new = (void *)GF_CALLOC(1, sizeof(*new), gf_common_mt
    _inode_table_t);
13.     if (!new)
14.         return NULL;
15.
16.     new->xl = xl;
17.     new->ctxcount = xl->graph->xl_count + 1;
18.
19.     new->lru_limit = lru_limit;
20.     new->invalidator_fn = invalidator_fn;
21.     new->invalidator_xl = invalidator_xl;
22.
23.     if (dentry_hashsize == 0) {
24.         /* Prime number for uniform distribution */
25.         new->dentry_hashsize = 14057;
26.     } else {
27.         new->dentry_hashsize = dentry_hashsize;
28.     }
29.
30.     if (inode_hashsize == 0) {
31.
32.         new->inode_hashsize = 65536;
33.     } else {
34.         new->inode_hashsize = inode_hashsize;
35.     }
36.
37.
38.     if (!mem_pool_size || (mem_pool_size > DEFAULT_INODE_
    MEMPOOL_ENTRIES))
39.         mem_pool_size = DEFAULT_INODE_MEMPOOL_ENTRIES;
40.
41.     new->inode_pool = mem_pool_new(inode_t, mem_pool_size)
    ;
42.     if (!new->inode_pool)
43.         goto out;
44.
```

```
45.      new->dentry_pool = mem_pool_new(dentry_t, mem_pool_si
   ze);
46.      if (!new->dentry_pool)
47.          goto out;
48.
49.      new->inode_hash = (void *)GF_CALLOC(
50.           new->inode_hashsize, sizeof(struct list_head), gf
   _common_mt_list_head);
51.      if (!new->inode_hash)
52.          goto out;
53.
54.      new->name_hash = (void *)GF_CALLOC(
55.           new->dentry_hashsize, sizeof(struct list_head), g
   f_common_mt_list_head);
56.      if (!new->name_hash)
57.          goto out;
58.
59.
60.      new->fd_mem_pool = mem_pool_new(fd_t, 1024);
61.
62.      if (!new->fd_mem_pool)
63.          goto out;
64.
65.      for (i = 0; i < new->inode_hashsize; i++) {
66.          INIT_LIST_HEAD(&new->inode_hash[i]);
67.      }
68.
69.      for (i = 0; i < new->dentry_hashsize; i++) {
70.          INIT_LIST_HEAD(&new->name_hash[i]);
71.      }
72.
73.      INIT_LIST_HEAD(&new->active);
74.      INIT_LIST_HEAD(&new->lru);
75.      INIT_LIST_HEAD(&new->purge);
76.      INIT_LIST_HEAD(&new->invalidate);
77.
78.      ret = gf_asprintf(&new->name, "%s/inode", xl->name);

79.      if (-1 == ret) {
80.
81.          ;
82.      }
83.
84.      new->cleanup_started = _gf_false;
```

```
85.     /* inode-table 初始化的主要操作函数*/
86.     __inode_table_init_root(new);
87.
88.     pthread_mutex_init(&new->lock, NULL);
89.
90.     ret = 0;
91. out:
92.     if (ret) {
93.         if (new) {
94.             GF_FREE(new->inode_hash);
95.             GF_FREE(new->name_hash);
96.             if (new->dentry_pool)
97.                 mem_pool_destroy(new->dentry_pool);
98.             if (new->inode_pool)
99.                 mem_pool_destroy(new->inode_pool);
100.            GF_FREE(new);
101.            new = NULL;
102.        }
103.    }
104.
105.    return new;
106.}
```

因为 inode_table_new 函数中就一行调用，因此这里就不展示了，而这个函数中，可以看到前面都是初始化的内容，包括对 mem pool 的初始化，还有 inode hash 表等信息，这些信息也都是在 _inode_table 中的。

接着下面来看看__inode_table_init_root 函数的内容。

```
1.  static void
2.  __inode_table_init_root(inode_table_t *table)
3.  {
4.      inode_t *root = NULL;
5.      struct iatt iatt = {
6.          0,
7.      };
8.
9.      if (!table)
10.         return;
11.
```

```
12.     root = inode_create(table);
13.
14.     list_add(&root->list, &table->lru);
15.     table->lru_size++;
16.     root->in_lru_list = _gf_true;
17.
18.     iatt.ia_gfid[15] = 1;
19.     iatt.ia_ino = 1;
20.     iatt.ia_type = IA_IFDIR;
21.
22.     __inode_link(root, NULL, NULL, &iatt, 0);
23.     table->root = root;
24. }
```

这里就有两个关键的函数了，一个是 inode_create，另外一个是__inode_link。那么当这里初始化完成了 inode table 之后，任何要对文件或者目录进行的 fop，那么在这之前 ,其实都要调用 lookup 操作找到这个文件或者目录的 inode 对象。而 lookup 的操作会创建一个 inode_t 对象结构(这个结构在之前已经讨论过了)，接着会把这个 inode_t 对象关联到 inode table 中，并且在 active list 里面记录起来，也就是认为当前的 inode_t 对象是活跃的。

3.4.5.2. inode link

那么这里说到 lookup 调用，从客户端调用的请求，都要走 fuse 的，而这里的内容在 fuse-bridge.c 中，不管是 fuse_newentry_cbk,fuse_entry_cbk 和 fuse_lookup_cbk，最终这里都会调用 inode_link 函数，也就是有以下这样的函数调用，以下代码在 fuse-bridge.c 中。

```
1.  linked_inode = inode_link(inode, state->loc.parent, state
   ->loc.name,
2.                                buf);
```

那么接着来就是看看这个 inode_link 函数的实现是怎样的，该函数在 inode.c 中，下面可以进入查看一下。

```
1.  inode_t *
2.  inode_link(inode_t *inode, inode_t *parent, const char *n
    ame, struct iatt *iatt)
3.  {
4.
5.      ...
6.      table = inode->table;
7.
8.      //第一步,这里找到 dentry hash 的内容.
9.      if (parent && name) {
10.         hash = hash_dentry(parent, name, table->dentry_ha
    shsize);
11.     }
12.
13.    ...
14. }
```

这里的第一步，就是找到 dentry 的 hash 内容，对于 inode_t 结构来说，知道了 parent 和 name，那么就可以通过这个函数找到 dentry 信息了。

```
1.  static int
2.  hash_dentry(inode_t *parent, const char *name, int mod)
3.  {
4.      int hash = 0;
5.      int ret = 0;
6.
7.      hash = *name;
8.      if (hash) {
9.          for (name += 1; *name != '\0'; name++) {
10.             hash = (hash << 5) - hash + *name;
11.         }
12.     }
13.     ret = (hash + (unsigned long)parent) % mod;
14.
15.     return ret;
16. }
```

这里的内容就和前面提到的 gfid 的 hash 计算有点类似了 ,而且代码也比较易懂。

那么执行完这一步之后，下面才是真正地调用执行关联的函数，代码如下所示。

```
1.  inode_t *
2.  inode_link(inode_t *inode, inode_t *parent, const char *n
    ame, struct iatt *iatt)
3.  {
4.
5.       ....
6.
7.      //第二步，真正调用关联的处理函数
8.      pthread_mutex_lock(&table->lock);
9.      {
10.          linked_inode = __inode_link(inode, parent, name,
    iatt, hash);
11.         if (linked_inode)
12.             __inode_ref(linked_inode, false);
13.     }
14.     pthread_mutex_unlock(&table->lock);
15.
16.     ....
17.
18. }
```

那么这里调用的__inode__linke,也是前面的初始化 inode table 时会调用的，下面就进入该函数查看一下内容。

```
1.  static inode_t *
2.  __inode_link(inode_t *inode, inode_t *parent, const char
    *name,
3.               struct iatt *iatt, const int dhash)
4.  {
```

```
5.      dentry_t *dentry = NULL;
6.      dentry_t *old_dentry = NULL;
7.      inode_t *old_inode = NULL;
8.      inode_table_t *table = NULL;
9.      inode_t *link_inode = NULL;
10.     char link_uuid_str[64] = {0}, parent_uuid_str[64] = {
    0};
11.
12.     table = inode->table;
13.
14.     //这里首先当 parent 存在的时候，inode 的情况
15.     if (parent) {
16.         //这里是安全检查，为了避免 inode 在不同的 inode table
    中，那样会造成一些错误，并且很难排查
17.         if (inode->table != parent->table) {
18.             errno = EINVAL;
19.             GF_ASSERT(!"link attempted b/w inodes of diff
     table");
20.         }
21.
22.         if (parent->ia_type != IA_IFDIR) {
23.             errno = EINVAL;
24.             GF_ASSERT(!"link attempted on non-directory p
    arent");
25.             return NULL;
26.         }
27.
28.         if (!name || strlen(name) == 0) {
29.             errno = EINVAL;
30.             GF_ASSERT (!"link attempted with no basename
    on "
31.                                     "parent");
32.             return NULL;
33.         }
34.     }
35.
36.     link_inode = inode;
37.
38.     if (!__is_inode_hashed(inode)) {
39.         if (!iatt) {
40.             errno = EINVAL;
```

```
41.             return NULL;
42.         }
43.
44.         if (gf_uuid_is_null(iatt->ia_gfid)) {
45.             errno = EINVAL;
46.             return NULL;
47.         }
48.
49.         //这里是得到 gfid hash 值
50.          int ihash = hash_gfid(iatt->ia_gfid, table->inode
    _hashsize);
51.
52.         //这里发现是否已经存在 inode 对象
53.          old_inode = __inode_find(table, iatt->ia_gfid, ih
    ash);
54.
55.         if (old_inode) {
56.             link_inode = old_inode;
57.         } else {
58.             gf_uuid_copy(inode->gfid, iatt->ia_gfid);
59.             inode->ia_type = iatt->ia_type;
60.             __inode_hash(inode, ihash);
61.         }
62.     } else {
63.         old_inode = inode;
64.     }
65.
66.     if (name && (!strcmp(name, ".") || !strcmp(name, "..")
    )) {
67.         return link_inode;
68.     }
69.
70.     //下面是和  dentry 相关的内容
71.     if (parent) {
72.          old_dentry = __dentry_grep(table, parent, name, d
    hash);
73.
74.         if (!old_dentry || old_dentry->inode != link_inod
    e) {
75.              dentry = dentry_create(link_inode, parent, na
    me);
```

```
76.             if (!dentry) {
77.                 gf_msg_callingfn(THIS->name, GF_LOG_ERROR,
    0,
78.                                  LG_MSG_DENTRY_CREATE_FAI
   LED,
79.                                  "dentry create failed on
    "
80.                                  "inode %s with parent %s
   ",
81.                                  uuid_utoa_r(link_inode->
   gfid, link_uuid_str),
82.                                 uuid_utoa_r(parent->gfid,
    parent_uuid_str));
83.                 errno = ENOMEM;
84.                 return NULL;
85.             }
86.
87.
88.              dentry->parent = __inode_ref(parent, false);

89.              list_add(&dentry->inode_list, &link_inode->de
   ntry_list);
90.
91.              if (old_inode && __is_dentry_cyclic(dentry))
   {
92.                 errno = ELOOP;
93.                 dentry_destroy(__dentry_unset(dentry));
94.                 return NULL;
95.             }
96.             __dentry_hash(dentry, dhash);
97.
98.             if (old_dentry)
99.                 dentry_destroy(__dentry_unset(old_dentry))
   ;
100.         }
101.     }
102.
103.     return link_inode;
104.}
```

这里主要就是分为两大部分的内容，是 inode 和 dentry 对象的获取，这里的代码主要是做一些安全检查，例如查看是否已经存在了该对象，或者新创建该

对象等。

那么这里调用创建了 inode 对象以后,根据回到前面的代码中可以知道，这里还会调用 __inode_ref 的函数 ,这个函数的作用就是对 inode active 的数量进行一些处理，并且这里还有一个值得注意的点，就是关于 root inode 的，见下面代码所示。

```
1.  static inode_t *
2.  __inode_ref(inode_t *inode, bool is_invalidate)
3.  {
4.      int index = 0;
5.      xlator_t *this = NULL;
6.
7.      if (!inode)
8.          return NULL;
9.
10.     this = THIS;
11.
12.     //这里对 root gfid 不做处理，永远保持 1,避免被处理.
13.     if (__is_root_gfid(inode->gfid) && inode->ref)
14.         return inode;
15.
16.     if (!inode->ref) {
17.         if (inode->in_invalidate_list) {
18.             inode->in_invalidate_list = false;
19.             inode->table->invalidate_size--;
20.         } else {
21.             GF_ASSERT(inode->table->lru_size > 0);
22.             GF_ASSERT(inode->in_lru_list);
23.             inode->table->lru_size--;
24.             inode->in_lru_list = _gf_false;
25.         }
26.         if (is_invalidate) {
27.             inode->in_invalidate_list = true;
28.             inode->table->invalidate_size++;
29.              list_move_tail(&inode->list, &inode->table->i
    nvalidate);
30.         } else {
31.             __inode_activate(inode);
```

```
32.             }
33.         }
34.
35.         //这里就是 inode 的 ref 引用加 1
36.         inode->ref++;
37.
38.         index = __inode_get_xl_index(inode, this);
39.         if (index >= 0) {
40.             inode->_ctx[index].xl_key = this;
41.             inode->_ctx[index].ref++;
42.         }
43.
44.         return inode;
45. }
```

这里因为 root inode 要永远保持活跃的，因此单独给这部分做了判断，那么这里在什么时候 ,把创建的这个新的 inode 移到 active 表中呢？就是这个函数中调用的__inode_activate 的功能了，可以进入查看一下。

```
1.  static void
2.  __inode_activate(inode_t *inode)
3.  {
4.      list_move(&inode->list, &inode->table->active);
5.      inode->table->active_size++;
6.  }
```

这里的代码内容很直观，就是把该 inode 移到 active 中，并且增加 active_size 的数量。

3.4.5.3. inode unref

前面提到了创建一个 inode 的过程，那么该如何释放呢？这里的释放有两种 ,一种是 glusterfs 中调用 inode unref 的方式 ,也就是当要删除文件的时候，

那么这个 inode 也要主动释放掉，这里就是使用 inode_unref 函数进行处理，下面一起来了解一下。

```
1.  inode_t *
2.  inode_unref(inode_t *inode)
3.  {
4.      inode_table_t *table = NULL;
5.
6.      if (!inode)
7.          return NULL;
8.
9.      table = inode->table;
10.
11.     pthread_mutex_lock(&table->lock);
12.     {
13.         inode = __inode_unref(inode, false);
14.     }
15.     pthread_mutex_unlock(&table->lock);
16.
17.     inode_table_prune(table);
18.
19.     return inode;
20. }
```

从这里可以知道，真正的处理函数是__inode_unref，下面继续进入查看一下。

```
1.  static inode_t *
2.  __inode_unref(inode_t *inode, bool clear)
3.  {
4.      int index = 0;
5.      xlator_t *this = NULL;
6.      uint64_t nlookup = 0;
7.
8.      // 根目录下永远保持 active 状态.
9.      if (__is_root_gfid(inode->gfid))
10.         return inode;
11.
12.     if (inode->table->cleanup_started && !inode->ref)
```

```
13.
14.         return inode;
15.
16.     this = THIS;
17.
18.
19.     if (clear && inode->in_invalidate_list) {
20.         inode->in_invalidate_list = false;
21.         inode->table->invalidate_size--;
22.         __inode_activate(inode);
23.     }
24.     GF_ASSERT(inode->ref);
25.
26.     --inode->ref;
27.
28.     //这里和 inode_ref 刚好相反，就是对数量减 1
29.     index = __inode_get_xl_index(inode, this);
30.     if (index >= 0) {
31.         inode->_ctx[index].xl_key = this;
32.         inode->_ctx[index].ref--;
33.     }
34.
35.     if (!inode->ref && !inode->in_invalidate_list) {
36.         inode->table->active_size--;
37.
38.         //这里根据 nlookup 的数量判断是否需要销毁.
39.         nlookup = GF_ATOMIC_GET(inode->nlookup);
40.         if (nlookup)
41.             __inode_passivate(inode);
42.         else
43.             __inode_retire(inode);
44.     }
45.
46.     return inode;
47. }
```

这里首先还是判断，如果是根目录的话，是要永远保持 active 状态的，然后这里判断是否要销毁该 inode，要考虑 ref, invalidation 和 nlookup 三者的数量关系的，然后根据不同的状态来调用函数。简单来说，这里有可能会出现，

glusterfs 中想取消引用，但是操作系统还在引用这个 inode，或者两者都想取消等情况的出现，表现出来就是三者的数值之间的变化，加上调用函数的不同，最终处理的方式也不一样。关于这部分三者关系的内容，后面还会进一步总结一下。

3.4.5.4. inode purne

前面提到了 inode 的创建，取消引用，那么下面了解一下销毁删除部分。当一个 dentry 被删除，或者 LRU 达到最大限制的时候，这个 inode 就会被移除 inode-table 中，放进 purge 销毁队列中。另外在销毁之前，这里还要对该 inode 进行 unhash，避免以后有冲突，下面进行简单了解一下。

```
1.  static int
2.  inode_table_prune(inode_table_t *table)
3.  {
4.      int ret = 0;
5.      int ret1 = 0;
6.      struct list_head purge = {
7.          0,
8.      };
9.      inode_t *del = NULL;
10.     inode_t *tmp = NULL;
11.     inode_t *entry = NULL;
12.     uint64_t nlookup = 0;
13.     int64_t lru_size = 0;
14.
15.     if (!table)
16.         return -1;
17.
18.     INIT_LIST_HEAD(&purge);
19.
20.     pthread_mutex_lock(&table->lock);
21.     {
22.         if (!table->lru_limit)
23.             goto purge_list;
```

```
24.
25.         lru_size = table->lru_size;
26.         while (lru_size > (table->lru_limit)) {
27.             if (list_empty(&table->lru)) {
28.                 GF_ASSERT(0);
29.                  gf_msg_callingfn(THIS->name, GF_LOG_WARNI
   NG, 0,
30.                                   LG_MSG_INVALID_INODE_LIS
   T,
31.                                   "Empty inode lru list fo
   und"
32.                                   " but with (%d) lru_size
   ",
33.                                  table->lru_size);
34.                 break;
35.             }
36.
37.             lru_size--;
38.              entry = list_entry(table->lru.next, inode_t,
   list);
39.             GF_ASSERT(entry->in_lru_list);
40.
41.             ...
42.
43.             table->lru_size--;
44.             entry->in_lru_list = _gf_false;
45.             //这里就是做 unhash 操作的入口
46.             __inode_retire(entry);
47.             ret++;
48.         }
49.
50.     purge_list:
51.         list_splice_init(&table->purge, &purge);
52.         table->purge_size = 0;
53.     }
54.     pthread_mutex_unlock(&table->lock);
55.
56.     .....
57.
58.      //这里是真正执行 pruge list 销毁操作的
59.     list_for_each_entry_safe(del, tmp, &purge, list)
60.     {
```

```
61.         list_del_init(&del->list);
62.         inode_forget_atomic(del, 0);
63.         __inode_destroy(del);
64.     }
65.
66.     return ret;
67. }
```

在这个函数里面，当 lru 的容量大于限制时，这里会进行一些清理操作，而清理前的 unhash 操作，就是__inode_retire 函数进行的，这里进入了解一下。

```
1.  static void
2.  __inode_retire(inode_t *inode)
3.  {
4.      dentry_t *dentry = NULL;
5.      dentry_t *t = NULL;
6.
7.      //把节点移到 purge list 上
8.      list_move_tail(&inode->list, &inode->table->purge);
9.      inode->table->purge_size++;
10.
11.     //进行 unhash 操作
12.     __inode_unhash(inode);
13.
14.      list_for_each_entry_safe(dentry, t, &inode->dentry_li
   st, inode_list)
15.     {
16.         dentry_destroy(__dentry_unset(dentry));
17.     }
18. }
```

这里再进入一下，查看一下该函数的实现，这里__inde_unhash 会调用 list_del_init 函数，该函数实现如下所示。

```
1.  static inline void
2.  list_del_init(struct list_head *old)
3.  {
4.      old->prev->next = old->next;
```

```
5.      old->next->prev = old->prev;
6.
7.      old->next = old;
8.      old->prev = old;
9.  }
```

这里其实就是把 inode 在链表中断开，同时自己指向自己，不然该 inode 可以再次指向链表上。

那么这里回到前面的 inode_table_purne 函数，这里在最后 list_for_each_entry_safe 里面，也有调用 list_del_init 函数的。另外这里后面调用的__inode_destroy,在里面会调用__inode_ctx_free，这里会调用 forget,这个在前面提到其作用，这里也是为了避免操作系统还在引用该 inode 会导致出错。

对于 fuse 来说，正常来讲因为 lru 是可以放无限多的数据的，因为 fuse kernel 可以使用 forget call 来删除一些 inode,而 forget 的作用 ,在前面提到过,可以理解为操作系统层面要调用一个类似 unref 的操作一样，会通知上层的 glusterfs。但是因为 glusterfs 是一个用户进程 ,因此当 lru 中有大量的 inode 节点信息 ,但是内核没有快速地回调 forget 通知 glusterfs ,这里就会导致 OOM 的出现了，在最新的实现中，lru 目前被限制在 65536 个数据大小(具体实际大小根据不同的操作系统和设置可能会有区别)。而这里 fuse 内核也会去做做一些 invalidation 操作，也就是把 inode 信息失效，这样来腾出一些空间来。前面在介绍 inode table 的时候，也提到过，关于 inode 其实重要的是三个事情，就是 ref,nlookup 和 invalication 的数值，因为这里就代表着应用与操作系统之间的引用状态关系，下面就给出一个简单总结吧。

```
 refs nlookups inv_sent    op
     1      0        0      unref  -> refs = 0, active--->destroy
     1      1        0      unref  -> refs = 0, active--->lru
     1      1        0     forget  -> nlookups = 0, active-->active
     0      1        0       ref   -> refs = 1, lru--->active
     0      1        1       ref   -> refs = 1, inv_sent = 0, invalidate--->active
     0      1        0    overflow -> refs = 1, inv_sent = 1, lru--->invalidate
     1      1        1      unref  -> refs = 0, invalidate-->invalidate
     1      1        1     forget  -> nlookups = 0, inv_sent = 0, invalidate--->active
```

这里的数值关系与对应的操作，这里的 inv_sent 是一个标记，就是用于判断当前的 inode 是否已经在缓存中失效，这里会和 inode.c 的函数 inode_invalidate 有关，但是 inode 失效并不意味着该 inode 不能读写，除非是调用 forget 和 unref，确定了当前 inode 在操作系统中没的缓存没有引用，同时 glusterfs 中也不再需要了，那么这里就会有 destroy 操作了，也就是 purne 函数。

这里讲了很多关于 inode 部分的内容，核心关键就是希望大家对 inode 问题有一个简单的认识，对于 inode 来说，因为不仅仅涉及到 glusterfs 会使用，还有系统的缓存和内核引用等问题，因此该对象的回收处理，不能单纯直接释放销毁，否则会引起一些异常的，这点其实除了 inode 以外，对于文件句柄 fd 来说也是有异曲同工之处的，因此关于 fuse 的内容，未来大家开发的时候，或许可以借鉴一下这里的思想。