3.3. posix 接口的那些事

前面讲了一些简单的数据结构在内存中的模型 ,那么知道了 loc_t 和 inode_t 这两个结构之后，如果要真正获取一个文件 inode 信息，那么这里需要怎么做呢？这就需要用到 posix 接口了，所谓的 posix 接口，是可移植操作系统接口，因为操作系统的差异可能会导致在具体某些功能实现上不同 ,但是接口的出入参和功能确定好，那么调用的时候，调用者就不太需要关心实现细节了。当然，简单来理解，也可以理解为 web 开发中常见的 restful api 接口。

那么 glusterfs 中要获取一个文件的 inode 信息，必然离不开一个基础的 posix 接口了，那就是 posix_lookup,下面来简单了解一下。

3.3.1. posix_lookup

首先这个接口的实现代码在 xlators/storage/posix/src/posix-entry-ops.c 文件里面，下面来看看代码。另外请注意下面提到的 inode 信息一般指代操作系统的 inode，而 inode_t 则指代 glusterfs 的一个数据结构，也就是前面提到的。

```
1.  int32_t
2.  posix_lookup(call_frame_t *frame, xlator_t *this, loc_t *
    loc, dict_t *xdata)
3.  {
4.
5.      ....
6.      if (op_ret == -1) {
7.          if (op_errno != ENOENT) {
8.              gf_msg(this->name, GF_LOG_WARNING, op_errno,
    P_MSG_LSTAT_FAILED,
9.                     "lstat on %s failed", real_path ? real
    _path : "null");
```

```
10.         }
11.         entry_ret = -1;
12.         //这里意味着,loc 结构中只有 inode 和 gfid
13.         if (loc_is_nameless(loc)) {
14.             if (!op_errno)
15.                 op_errno = ESTALE;
16.             loc_gfid(loc, gfid);
17.             //这里是获取到绝对路径
18.              MAKE_HANDLE_ABSPATH_FD(gfid_path, this, gfid,
    dfd);
19.             //获取 stat 信息
20.             ret = sys_fstatat(dfd, gfid_path, &statbuf, 0)
   ;
21.              if (ret == 0 && ((statbuf.st_mode & S_IFMT) =
   = S_IFDIR))
22.                 goto parent;
23.              ret = sys_fstatat(dfd, gfid_path, &statbuf, A
   T_SYMLINK_NOFOLLOW);
24.             if (ret == 0 && statbuf.st_nlink == 1) {
25.                  gf_msg(this->name, GF_LOG_WARNING, op_err
   no,
26.                         P_MSG_HANDLE_DELETE,
27.                         "Found stale gfid "
28.                         "handle %s, removing it.",
29.                         gfid_path);
30.                 posix_handle_unset(this, gfid, NULL);
31.             }
32.         }
33.         goto parent;
34.     }
35.     ...
36.
37. }
```

首先这里要注意一下入参 ,这里入参的 xlator 是 glusterfs 中另外一个很重要的功能模块化的概念，后面会提到。

在该函数中，主要四部分代码构成，前面还有一部分代码是处理 gfid 不存

在的情况的。在这段代码实现中，这里一开始入参中 loc_t *loc 是带有数据的，然后如果这里走到了 loc_is_nameless 的判断中，这里的判断就是当 loc 中只有 inode 和 gfid 信息的时候，需要进行获取一些其他的信息。另外这里还可以看下函数 MAKE_HANDLE_ABSPATH_FD 的实现。

```
1.  #define MAKE_HANDLE_ABSPATH_FD(var, this, gfid, dfd)
                                \
2.      do {
                                \
3.          struct posix_private *__priv = this->private;
                                \
4.          int findex = gfid[0];
                                \
5.          int __len = POSIX_GFID_HASH2_LEN;
                                \
6.          var = alloca(__len);
                                \
7.          snprintf(var, __len, "%02x/%s", gfid[1], uuid_uto
   a(gfid));                    \
8.          dfd = __priv->arrdfd[findex];
                                \
9.      } while (0)
```

这段代码在 xlator/storage/posix/src/posix-handle.h 文件中，这里的 whil(0)就是只执行一次，然后使用宏定义封装函数，这样做的好处是避免编译出错。这里的作用是根据 gfid 的规则，拼接获取文件的绝对路径的。获取到了绝对路径，那么获取就可以获取 stat 信息了。

那么这里回到前面的函数，后面还有跳转到 parent 分支的，这里的思路其实也是类似的。

总的来说，这个函数的作用，总结起来可以如下所示，就是通过 gfid 和目录信息，获取到对应的 stat,然后最终获取到 inode_t 结构里面的其他信息。

3.3.2. posix_open

那么前面有了 inode_t 这个结构的信息之后，如果想打开文件的话，那么就需要用到 posix_open 这个接口了，该接口的代码是在 xlator/storage/posix/src/posix-inode-fd-ops.c，这个函数的作用就是通过 inode_t 和 loc_t,获取到 fd_t 对象信息，然后调用系统调用 open 来打开文件得到文件句柄 fd,并且回填 inode_t 中的相关信息。下面来简单看看函数中的主要内容。

```
1.  int32_t
2.  posix_open(call_frame_t *frame, xlator_t *this, loc_t *lo
    c, int32_t flags,
3.              fd_t *fd, dict_t *xdata)
4.  {
5.          ....
6.           //前面有很多校验代码,这里通过 inode_t 信息获取到真实路
    径之后调用系统调用 open
7.           _fd = sys_open(real_path, flags, priv->force_creat
    e_mode);
8.       if (_fd == -1) {
9.           op_ret = -1;
10.          op_errno = errno;
11.          gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_FIL
    E_OP_FAILED,
12.                 "open on gfid-handle %s (path: %s), flags:
     %d", real_path,
13.                 loc->path, flags);
14.          goto out;
15.      }
16.
```

```
17.     //修改 ctime 时间
18.      posix_set_ctime(frame, this, real_path, -1, loc->inod
   e, &stbuf);
19.
20.     pfd = GF_CALLOC(1, sizeof(*pfd), gf_posix_mt_posix_fd)
   ;
21.     if (!pfd) {
22.         op_errno = errno;
23.         goto out;
24.     }
25.
26.     //这里记录 fd 和 flags 信息
27.     pfd->flags = flags;
28.     pfd->fd = _fd;
29.
30.     if (xdata) {
31.         op_ret = posix_fdstat(this, fd->inode, pfd->fd, &
   preop);
32.         if (op_ret == -1) {
33.             gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG
   _FSTAT_FAILED,
34.                    "pre-operation fstat failed on fd=%p",
    fd);
35.             GF_FREE(pfd);
36.             goto out;
37.         }
38.
39.         //这个函数用于检查和更新扩展属性的请求.
40.        //另外这里还修复文件的一些状态信息，因为可能请求结果会发
   生冲突或失败等
41.         posix_cs_maintenance(this, fd, NULL, &pfd->fd, &p
   reop, NULL, xdata,
42.                              &rsp_xdata, _gf_true);
43.     }
44.
45.     op_ret = fd_ctx_set(fd, this, (uint64_t)(long)pfd);
46.     if (op_ret)
47.         gf_msg(this->name, GF_LOG_WARNING, 0, P_MSG_FD_PA
   TH_SETTING_FAILED,
```

```
48.                 "failed to set the fd context gfid-handle=
    %s path=%s fd=%p",
49.                 real_path, loc->path, fd);
50.
51.     op_ret = 0;
52.         ....
53. }
```

那么这里的关键之一是调用 sys_open，可以进入该函数进一步查看具体实现，代码如下所示。

```
1.  int
2.  sys_openat(int dirfd, const char *pathname, int flags, in
    t mode)
3.  {
4.      int fd;
5.
6.  #ifdef GF_DARWIN_HOST_OS
7.      if (fchdir(dirfd) < 0)
8.          return -1;
9.      fd = open(pathname, flags, mode);
10.
11. #else
12.     fd = openat(dirfd, pathname, flags, mode);
13. #ifdef __FreeBSD__
14.
15.     if ((fd >= 0) && ((flags & O_CREAT) != 0) && ((mode &
    S_ISVTX) != 0)) {
16.         sys_fchmod(fd, mode);
17.
18.     }
19. #endif /* __FreeBSD__ */
20. #endif /* !GF_DARWIN_HOST_OS */
21.
22.     return FS_RET_CHECK(fd, errno);
23. }
```

这一段代码是在 libglusterfs/src/syscall.c 中，在这个文件里面还定义了很多不同的系统调用。而这段代码里面，其实就是定义不同的操作平台的系统调用了。

另外这里为了方便去查阅一些 linux 系统的调用接口，可以使用 man 2 open 这样的命令进行查看,这样可以更加方便地理解一些参数的作用。

上面介绍的 posix 接口都是比较基础的功能 ,这里还有很多不同的 posix 接口，如果在阅读代码的过程中，对部分细节掌握感到困惑的时候，只要对函数代码功能整体比较清晰的话，可以暂时不用细究每一处细节，等到不断学习的时候到后面的时候，或许有些地方会有了新的理解，再重新回头看该处困惑，那么可能就能明白了，尤其是在阅读项目源码的时候，会经常遇到。

3.3.3. gluster fop

那么前面讲了两个比较基础和重要的 posix 接口 ,同时前面的内容中多次提到了 fop,那么这两者之间是否有联系和区别呢？在 glusterfs 中 ,发生在文件上的所有操作都表示为 fop。从文件系统的角度来说，每一个文件系统都需要注册到操作系统里面的，然后自定义操作，例如打开 create,创建目录 mkdir 等操作函数，这些就是 file operations,也就是文件操作函数。而所谓的 posix，则是一套规范，因为很多系统是需要考虑跨平台的，因此有了 posix 规范，而对于真正的代码开发来说，定义是采用了 operations 的，而实现是使用了 posix 接口的， 下面可以简单看看 glusterfs 中的 fop 相关的内容。

```
1.  enum glusterfs_fop_t {
2.      GF_FOP_NULL = 0,
3.      GF_FOP_STAT = 0 + 1,
4.      GF_FOP_READLINK = 0 + 2,
```

```
5.      GF_FOP_MKNOD = 0 + 3,
6.      GF_FOP_MKDIR = 0 + 4,
7.      GF_FOP_UNLINK = 0 + 5,
8.      GF_FOP_RMDIR = 0 + 6,
9.      GF_FOP_SYMLINK = 0 + 7,
10.     GF_FOP_RENAME = 0 + 8,
11.     GF_FOP_LINK = 0 + 9,
12.     GF_FOP_TRUNCATE = 0 + 10,
13.     GF_FOP_OPEN = 0 + 11
14.     ...
15.     GF_FOP_ICREATE = 0 + 56,
16.     GF_FOP_NAMELINK = 0 + 57,
17.     GF_FOP_COPY_FILE_RANGE = 0 + 58,
18.     GF_FOP_MAXVALUE = 0 + 59,
19. };
```

这里的代码是在 libglusterfs/src/glusterfs/glusterfs-fops.h 文件中，这里定义了目前 glusterfs 中的 fop，而每一个 fop 的实现内容，可以见 posix.c 中的内容，该文件在 xlator/storage/posix/src/posix.c 中。

```
1.  struct xlator_fops fops = {
2.      .lookup = posix_lookup,
3.      .stat = posix_stat,
4.      .opendir = posix_opendir,
5.      .readdir = posix_readdir,
6.      .readdirp = posix_readdirp,
7.      .readlink = posix_readlink,
8.      .mknod = posix_mknod,
9.      .mkdir = posix_mkdir,
10.     .unlink = posix_unlink,
11.     .rmdir = posix_rmdir,
12.     .symlink = posix_symlink,
13.     .rename = posix_rename,
14.     ...
15. }
```

简单地来说，就是这里内部定义了一套 operation,但实现采用 posix。另外这里 fop 的执行都需要使用到 loc_t 或者 fd_t 结构。

3.3.4. posix_mkdir

那么这里也简单讲讲关于 posix_mkdir 这个接口吧，对于 mkdir 这个命令大家也不会陌生,那么这里代码如下所示。

```
1.  int
2.  posix_mkdir(call_frame_t *frame, xlator_t *this, loc_t *l
    oc, mode_t mode,
3.              mode_t umask, dict_t *xdata)
4.  {
5.       //系统调用 mkdir
6.       op_ret = sys_mkdir(real_path, mode);
7.      if (op_ret == -1) {
8.          op_errno = errno;
9.           gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_MKD
    IR_FAILED,
10.                 "mkdir of %s failed", real_path);
11.         goto out;
12.     }
13.
14.     entry_created = _gf_true;
15.
16. //下面开始都是设置一些参数的
17. #ifndef HAVE_SET_FSID
18.     op_ret = sys_chown(real_path, frame->root->uid, gid);
19.     if (op_ret == -1) {
20.         op_errno = errno;
21.          gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_CHO
    WN_FAILED,
22.                 "chown on %s failed", real_path);
23.         goto out;
24.     }
25. #endif
26.     op_ret = posix_acl_xattr_set(this, real_path, xdata);
27.     if (op_ret) {
28.          gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_ACL
    _FAILED,
```

```
29.                 "setting ACLs on %s failed ", real_path);
30.     }
31.
32.     op_ret = posix_entry_create_xattr_set(this, loc, real
   _path, xdata);
33.     if (op_ret) {
34.         gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_XAT
   TR_FAILED,
35.                 "setting xattrs on %s failed", real_path);
36.     }
37.
38.     //设置 gfid
39.     op_ret = posix_gfid_set(this, real_path, loc, xdata,
   frame->root->pid,
40.                             &op_errno);
41.     if (op_ret) {
42.         gf_msg(this->name, GF_LOG_ERROR, op_errno, P_MSG_
   GFID_FAILED,
43.                 "setting gfid on %s failed", real_path);
44.         goto out;
45.     } else {
46.         gfid_set = _gf_true;
47.     }
48.
49.     op_ret = posix_pstat(this, loc->inode, NULL, real_pat
   h, &stbuf, _gf_false);
50.     if (op_ret == -1) {
51.         op_errno = errno;
52.         gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_LST
   AT_FAILED,
53.                 "lstat on %s failed", real_path);
54.         goto out;
55.     }
56.
57.     //设置 ctime
58.     posix_set_ctime(frame, this, real_path, -1, loc->inod
   e, &stbuf);
59.
60.     op_ret = posix_pstat(this, loc->parent, loc->pargfid,
    par_path, &postparent,
```

```
                                _gf_false);
    if (op_ret == -1) {
        op_errno = errno;
         gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_LST
AT_FAILED,
                "post-operation lstat on parent of %s fail
ed", real_path);
        goto out;
    }

    //这里是修改父目录的 ctime
     posix_set_parent_ctime(frame, this, par_path, -1, loc
->parent, &postparent);

    op_ret = 0;
    ...
}
```

这个函数中前面的内容主要还是做一些参数缺失逻辑判断，而真正的系统函数调用则是从 sys_mkdir 开始，这里创建之后，那么还要进行参数设置，其中包括设置 gfid,还要设置父目录的 ctime 时间。 那么这里有一个小问题，如果是一个多层目录，使用 mkdir 命令创建之后，那么除了父目录和自身的 ctime 时间会更改之外，父目录的父目录这些还会更改吗？

关于这个小实验，各位可以自行测试实验一下。

3.3.5. posix_create

这里还有一个比较基础的 posix 接口是 posix_create,这个接口的代码也是比较类似和简单易懂的，下面给出函数中重要的代码部分。

```
int
posix_create(call_frame_t *frame, xlator_t *this, loc_t *
loc, int32_t flags,
```

```
3.                 mode_t mode, mode_t umask, fd_t *fd, dict_t
   *xdata)
4.  {
5.        ...
6.       if (!flags) {
7.          _flags = O_CREAT | O_RDWR | O_EXCL;
8.       } else {
9.          _flags = flags | O_CREAT;
10.      }
11.     ...
12.      mode_bit = (priv->create_mask & mode) | priv->force_c
   reate_mode;
13.     mode = posix_override_umask(mode, mode_bit);
14.     _fd = sys_open(real_path, _flags, mode);
15.     ...
16.
17. }
```

这里可以看到 flag 是带有 O_CEART 的，也就是说当文件不存在的时候回创建的，然后调用 sys_open 进行打开，接着后面也是一些属性的设置过程。这里如果感兴趣可以自行查阅代码。

最后可以通过一个小的实验观察一下，glusterfs 在创建文件过程中的特点，这里可以通过/proc 下创建文件的过程看到一些输出内容。

```
1.  Status of volume: test-replica
2.  Gluster process                                 TCP Port  RDM
   A Port  Online  Pid
3.  ------------------------------------------------------------
   ------------------
4.  Brick 192.168.0.110:/glusterfs/test-replica 49154      0
          Y       1427
5.  Brick 192.168.0.111:/glusterfs/test-replica 49154      0
          Y       1340
6.  Brick 192.168.0.112:/glusterfs/test-replica 49154      0
          Y       1291
7.  Self-heal Daemon on localhost                N/A        N/A
          Y       1345
8.  Self-heal Daemon on gfs02                    N/A        N/A
          Y       1351
```

```
Self-heal Daemon on gfs01                   N/A       N/A
        Y       1438

Task Status of Volume test-replica
------------------------------------------------------------
---------------------
There are no active volume tasks
```

首先这里看到一个 volume 的 status 信息，挂载该 volume,接着然后这里可以找到其中一个 brick 的 pid 。 在客户端挂载目录下使用 dd 命令的时候，可以在 brick 端执行命令 ls -l /proc/{PID}/fd，看到如下输出内容。

```
[root@gfs01 ~]# ls /proc/1427/fd -l
total 0
lr-x------ 1 root root 64 Jun  9 10:56 0 -> /dev/null
l-wx------ 1 root root 64 Jun  9 10:56 1 -> /dev/null
...
l-wx------ 1 root root 64 Jun  9 10:57 273 -> /glusterfs/
test-replica/.glusterfs/0d/0f/0d0fc592-afe8-441c-a736-6146
491ae571
....
```

而在客户端挂载目录下查看该文件的 gfid 属性可以看到就是对应的。当然这里第一次的时候，可以看到是一个绝对路径名称，而不是一个隐藏目录下的 gfid 路径，只有当文件存在的时候，这时候再次 dd，则是对同一个文件的打开，那么会使用到 gfid 路径了。

```
[root@gfsclient01 ~]# getfattr -n glusterfs.gfid.string
/mnt/test-replica/210609_02.txt
getfattr: Removing leading '/' from absolute path names
# file: mnt/test-replica/210609_02.txt
glusterfs.gfid.string="0d0fc592-afe8-441c-a736-6146491ae5
71"
```

而这里可以简单观察到在打开和创建文件过程中一些特点。

这里只是简单地分享了一些 glusterfs 中的 posix 接口和 fop 的概念，其中还有很多其他的 posix 接口，也不需要每一个都仔细阅读，当有需要的时候，去找到对应的代码分析一下即可。同时对于一些 fop 操作，这里也会在不同的场景下执行操作会遇到，可以遇到时再分析。