准备工作

- 安装环境：Centos7.8
- 安装步骤
- glusterfs的版本v7.7
- systemtap跟踪
- 修改日志级别

```
//for brick
gluster volume set test_volume diagnostics.brick-log-level TRACE

//for client
gluster volume set test_volume diagnostics.client-log-level TRACE

//for glusterd
// vim /etc/sysconfig/glusterd
## Set custom log file and log level (below are defaults)
#LOG_FILE='/var/log/glusterfs/glusterd.log'
LOG_LEVEL='DEBUG'

systemctl restart glusterd.service

//mount日志 trace
mount.glusterfs ks0:test_volume /mnt/test_volume log_level=trace
```

拓扑结构

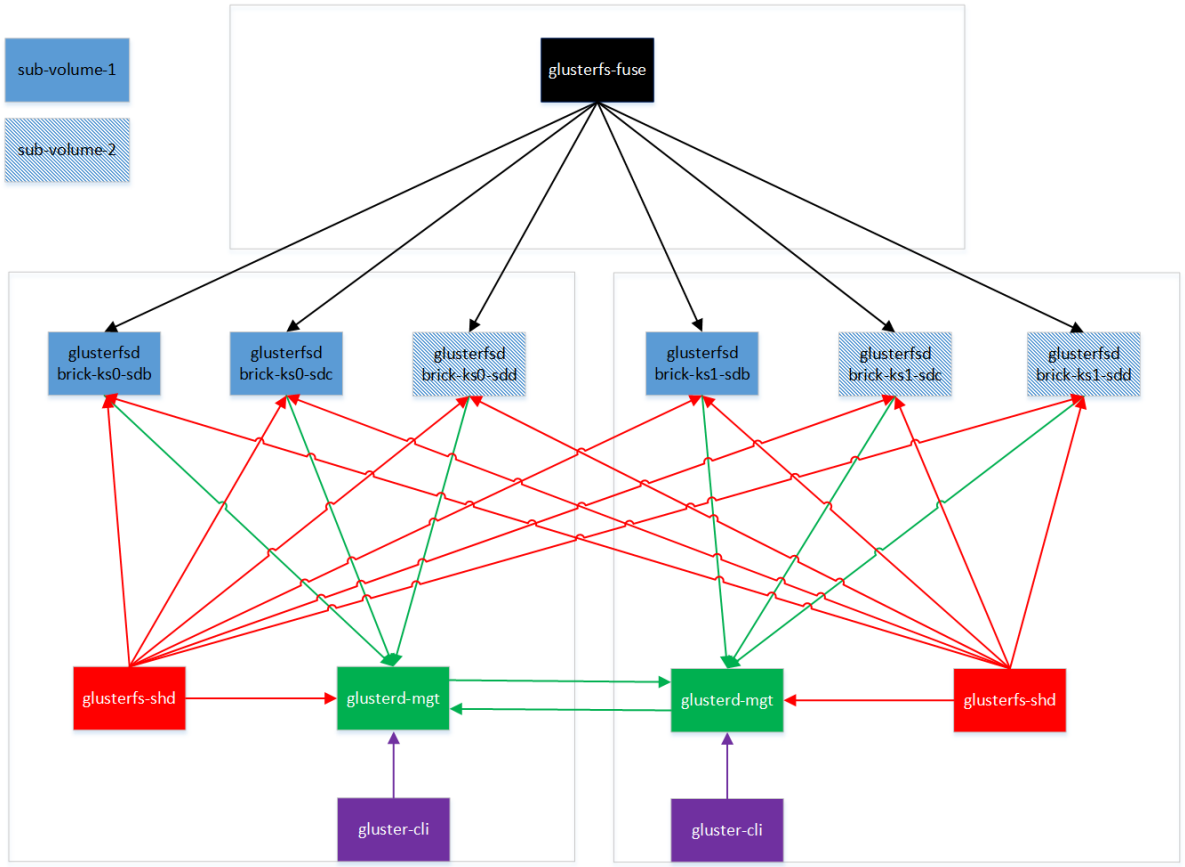
- 集群信息

```
gluster volume create test_volume disperse 3 redundancy 1 ks0:/data_ks0/sdb
ks0:/data_ks0/sdc ks1:/data_ks1/sdb ks0:/data_ks0/sdd ks1:/data_ks1/sdc ks1:/data_ks1/sdd
force

brick的个数必须是disperse的整数倍

[root@ks0 ~]# gluster peer status
Number of Peers: 1

Hostname: ks1
Uuid: 1cb6cb30-f636-4079-82a3-df97b2eb5cab
State: Peer in Cluster (Connected)


[root@ks0 /]# gluster volume status
Status of volume: test_volume
Gluster process                             TCP Port  RDMA Port  Online  Pid
------------------------------------------------------------------------------
Brick ks0:/data_ks0/sdb                     49155     0          Y       64066
Brick ks0:/data_ks0/sdc                     49153     0          Y       90020
```

```
Brick ks1:/data_ks1/sdb                     49152     0          Y       116311
Brick ks0:/data_ks0/sdd                     49152     0          Y       89185
Brick ks1:/data_ks1/sdc                     49153     0          Y       129680
Brick ks1:/data_ks1/sdd                     49156     0          Y       111629
Self-heal Daemon on localhost               N/A       N/A        Y       108469
Self-heal Daemon on ks1                     N/A       N/A        Y       3065

Task Status of Volume test_volume
------------------------------------------------------------------------------
Task                 : Rebalance
ID                   : 4de76df5-f88a-4e66-8c73-9b2ca72c097f
Status               : completed

[root@ks1 ~]# gluster volume info test_volume
Volume Name: test_volume
Type: Distributed-Disperse
Volume ID: 0c88aedc-dc9b-40cd-bbaf-d703f72dc0e5
Status: Started
Snapshot Count: 0
Number of Bricks: 2 x (2 + 1) = 6
Transport-type: tcp
Bricks:
Brick1: ks0:/data_ks0/sdb
Brick2: ks0:/data_ks0/sdc
Brick3: ks1:/data_ks1/sdb
Brick4: ks0:/data_ks0/sdd
Brick5: ks1:/data_ks1/sdc
Brick6: ks1:/data_ks1/sdd
Options Reconfigured:
cluster.disperse-self-heal-daemon: enable
nfs.disable: on
storage.fips-mode-rchecksum: on
transport.address-family: inet
diagnostics.brick-log-level: TRACE
diagnostics.client-log-level: TRACE


[root@ks1 ~]# ps axu | grep gluster | grep -v grep
root       3018  0.0  0.6 526860 11336 ?        Ssl  16:33   0:01 /usr/sbin/glusterd -p
/var/run/glusterd.pid --log-level INFO
root       3065  0.0  0.8 1034264 15512 ?       SLsl 16:33   0:01 /usr/sbin/glusterfs -s
localhost --volfile-id shd/test_volume -p /var/run/gluster/shd/test_volume/test_volume-
shd.pid -l /var/log/glusterfs/glustershd.log -S /var/run/gluster/14453e9a25aec937.socket
--xlator-option *replicate*.node-uuid=1cb6cb30-f636-4079-82a3-df97b2eb5cab --process-name
glustershd --client-pid=-6
root      97161  0.0  1.2 972004 21700 ?        SLsl 00:00   0:20 /usr/sbin/glusterfs --
process-name fuse --volfile-server=ks0 --volfile-id=test_volume /mnt/test_volume
root     111629  0.0  1.5 1378360 27340 ?       Ssl  06:21   0:25 /usr/sbin/glusterfsd -s
ks1 --volfile-id test_volume.ks1.data_ks1-sdd -p /var/run/gluster/vols/test_volume/ks1-
data_ks1-sdd.pid -S /var/run/gluster/0c44f763398723eb.socket --brick-name /data_ks1/sdd -
l /var/log/glusterfs/bricks/data_ks1-sdd.log --xlator-option *-posix.glusterd-
uuid=1cb6cb30-f636-4079-82a3-df97b2eb5cab --process-name brick --brick-port 49156 --
xlator-option test_volume-server.listen-port=49156
root     116311  0.0  1.3 1378360 24664 ?       Ssl  Nov03   2:02 /usr/sbin/glusterfsd -s
ks1 --volfile-id test_volume.ks1.data_ks1-sdb -p /var/run/gluster/vols/test_volume/ks1-
data_ks1-sdb.pid -S /var/run/gluster/6f25a5cc3fc90994.socket --brick-name /data_ks1/sdb -
l /var/log/glusterfs/bricks/data_ks1-sdb.log --xlator-option *-posix.glusterd-
uuid=1cb6cb30-f636-4079-82a3-df97b2eb5cab --process-name brick --brick-port 49152 --
```

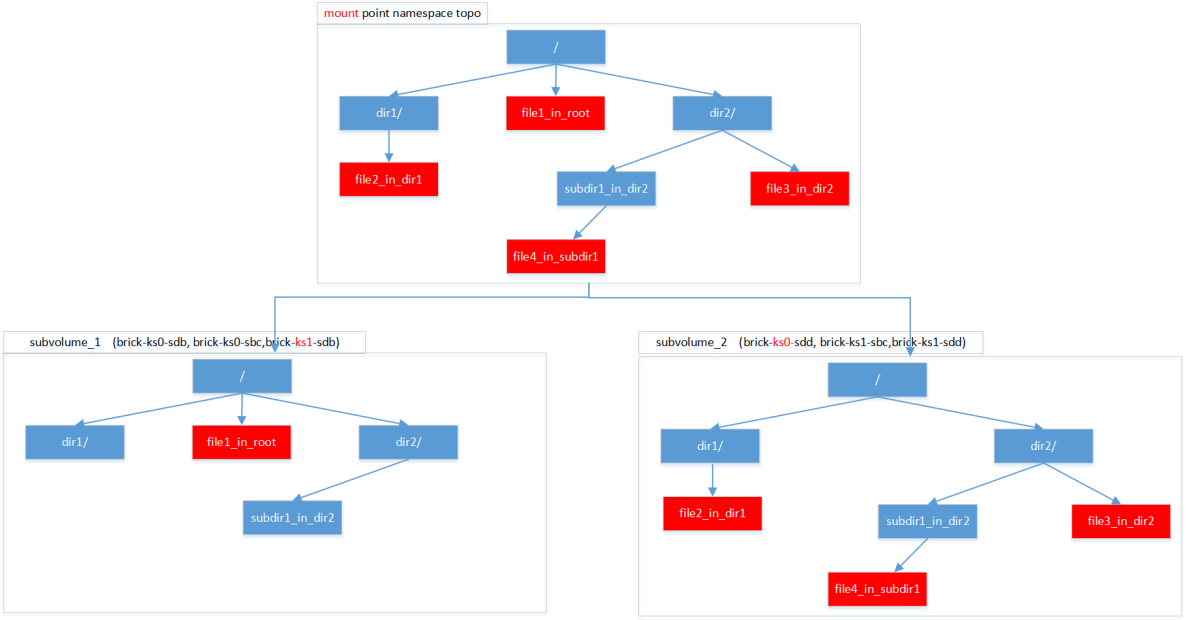
```
xlator-option test_volume-server.listen-port=49152
root     129680  0.0  1.4 1312824 26528 ?       Ssl  15:32   0:08 /usr/sbin/glusterfsd -s
ks1 --volfile-id test_volume.ks1.data_ks1-sdc -p /var/run/gluster/vols/test_volume/ks1-
data_ks1-sdc.pid -S /var/run/gluster/619b4b93af824dec.socket --brick-name /data_ks1/sdc -
l /var/log/glusterfs/bricks/data_ks1-sdc.log --xlator-option *-posix.glusterd-
uuid=1cb6cb30-f636-4079-82a3-df97b2eb5cab --process-name brick --brick-port 49153 --
xlator-option test_volume-server.listen-port=49153
```

- 进程topo



  - glustfsfs-fuse：client，posix语义的接入端，和mount point个数一致
  - glusterfs-shd: 负责数据heal的进程，只有一个
  - glustfsd-brick: server, 操作实际brick，落盘数据，和主机上的brick的个数一致
  - glusterd: glusterfs的管理进程，接受glusterfs-cli的管理和查询请求
  - gluster-cli: 发送cli命令到glusterd，glusterd将命令分发给glustfsd-server或者glustfsfs-fuse-client，或者gluserfs-shd

- 挂载点名字空间topo，brick名字空间topo



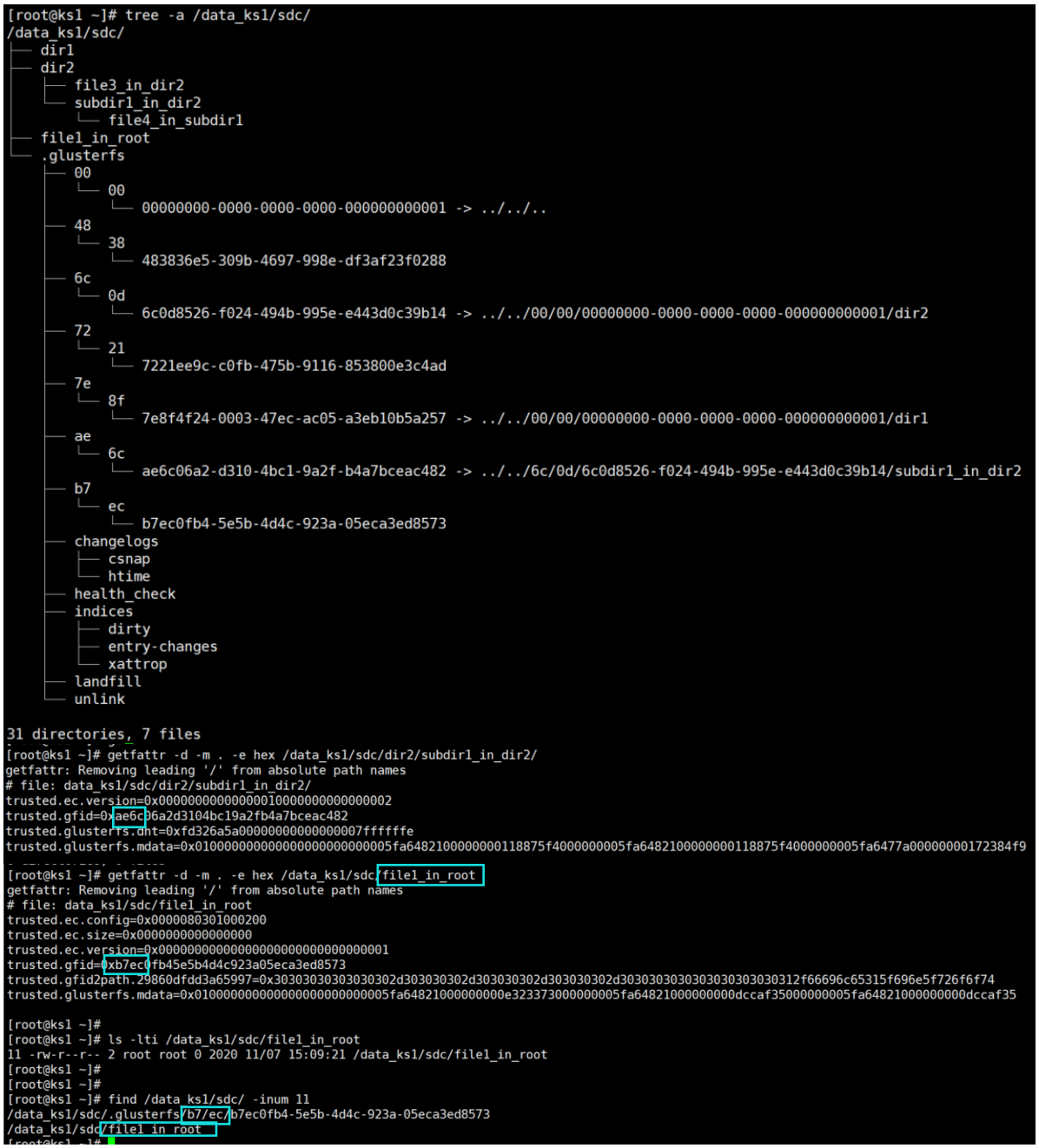
  - dir存在于所有brick下
    - dir创建逻辑：第一步，按全路径计算hash，在匹配的范围的subvolume中创建；第二步在其他subvolume中创建
  - file 文件名，在dht-[start, volume]范围的subvolume中创建
    - dir保存dht-layout
    - rebalance会更新 <rebalance 和 dht>
    - lookup会获取"trusted.glusterfs.dht"设置dir的dht-layout

```
[root@ks0 /]# umount /mnt/test_volume/
[root@ks0 /]# mount.glusterfs ks0:test_volume /mnt/test_volume log_level=trace
[root@ks0 /]# stat /mnt/test_volume/test/test1/test_file_in_test1/test_file_again/file_touch
  File: '/mnt/test_volume/test/test1/test_file_in_test1/test_file_again/file_touch'
  Size: 0               Blocks: 0          IO Block: 131072 regular empty file
Device: 23h/35d Inode: 10176855972081109060  Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2020-11-26 23:31:04.362176707 +0800
Modify: 2020-11-26 23:31:04.362176707 +0800
Change: 2020-11-26 23:31:04.363967999 +0800
 Birth: -
```

```
[root@ks0 ~]# grep -rn "trying regex for" /var/log/glusterfs/mnt-test_volume.log
187:[2020-11-26 15:39:04.133613] T [MSGID: 0] [dht-hashfn.c:86:dht_hash_compute] 0-test_volume-dht: trying regex for test
423:[2020-11-26 15:39:04.140818] T [MSGID: 0] [dht-hashfn.c:86:dht_hash_compute] 0-test_volume-dht: trying regex for test1
648:[2020-11-26 15:39:04.150550] T [MSGID: 0] [dht-hashfn.c:86:dht_hash_compute] 0-test_volume-dht: trying regex for test_file_in_test
1
873:[2020-11-26 15:39:04.157382] T [MSGID: 0] [dht-hashfn.c:86:dht_hash_compute] 0-test_volume-dht: trying regex for test_file_again
1098:[2020-11-26 15:39:04.167708] T [MSGID: 0] [dht-hashfn.c:86:dht_hash_compute] 0-test_volume-dht: trying regex for file_touch
[root@ks0 ~]#
```

```
int dht_hash_compute(xlator_t *this, int type, const char *name, uint32_t *hash_p) 计算hash_volume
xlator_t *dht_layout_search(xlator_t *this, dht_layout_t *layout, const char *name) 获取subvolume
    .....
    for (i = 0; i < layout->cnt; i++) {
         if (layout->list[i].start <= hash && layout->list[i].stop >= hash) {
             subvol = layout->list[i].xlator;
             break;
         }
     }
```

- brick目录文件说明



  - .glusterfs目录
    - 同一个subvolume下所有的brick的目录topo是一样的
    - 以gfid的头4个字符构成构建gfid快速查找目录：头两个字符构成第一级目录，后两个字符构成第二级目录
      - 两级目录下的文件名和dir,file的gfid一样，gfid是在volume中唯一存在的，由glusterfs-client生成
      - 如果是目录：以symlink指向brick下普通目录
      - 如果是文件：以hardlink关联brick下普通文件文件
  - xattr说明：
    - trusted.ec.config

```
8B---->
config.version(1B)|config.algorithm(1B)|gf_word_size(1B)|config.bricks(1B)|c
onfig.redundancy(1B)|config.chunk_size(3B)
```

```
config.version = EC_CONFIG_VERSION;
config.algorithm = EC_CONFIG_ALGORITHM;
config.gf_word_size = EC_GF_BITS;
config.bricks = ec->nodes;
config.redundancy = ec->redundancy;
config.chunk_size = EC_METHOD_CHUNK_SIZE;

#define EC_CONFIG_VERSION 0
#define EC_CONFIG_ALGORITHM 0
```

- trusted.ec.size: 文件实际大小
- trusted.ec.version：前8B是data的版本号，后8B是meta的版本号
- trusted.gfid: 有glusterfs-clinet生成的uuid
- trusted.gfid2path.xxxxx:
  - xxxxxx由"parentgfid/文件名"生成：

```
    snprintf(pgfid_bname, sizeof(pgfid_bname), "%s/%s",
uuid_utoa(pgfid),bname);
    gf_xxh64_wrapper((unsigned char *)pgfid_bname, strlen(pgfid_bname),
GF_XXHSUM64_DEFAULT_SEED, xxh64);
```

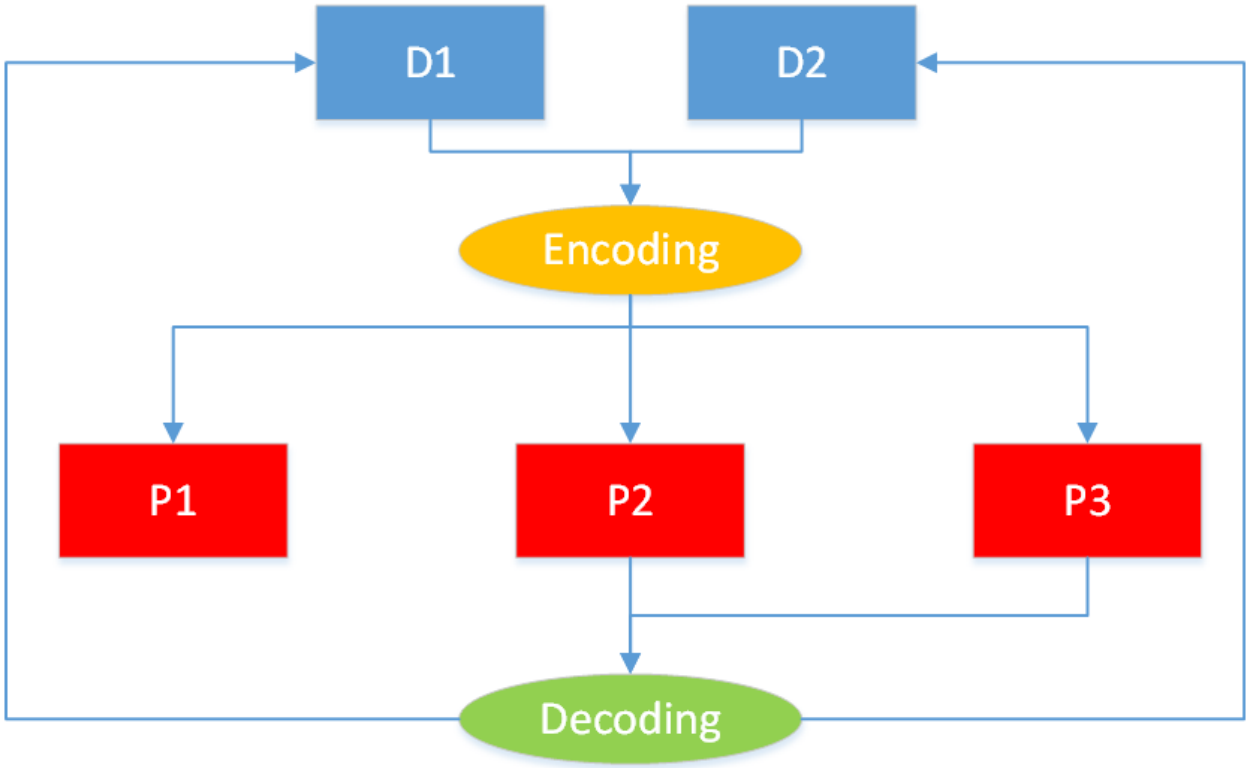
  - value: pgfid/base_name

```
[root@ks1 ~]# getfattr -d -m trusted.gfid2path -e text
/data_ks1/sdd/dir2/file3_in_dir2
getfattr: Removing leading '/' from absolute path names
# file: data_ks1/sdd/dir2/file3_in_dir2
trusted.gfid2path.73d9aeaf4bcc974f="6c0d8526-f024-494b-995e-
e443d0c39b14/file3_in_dir2"
```

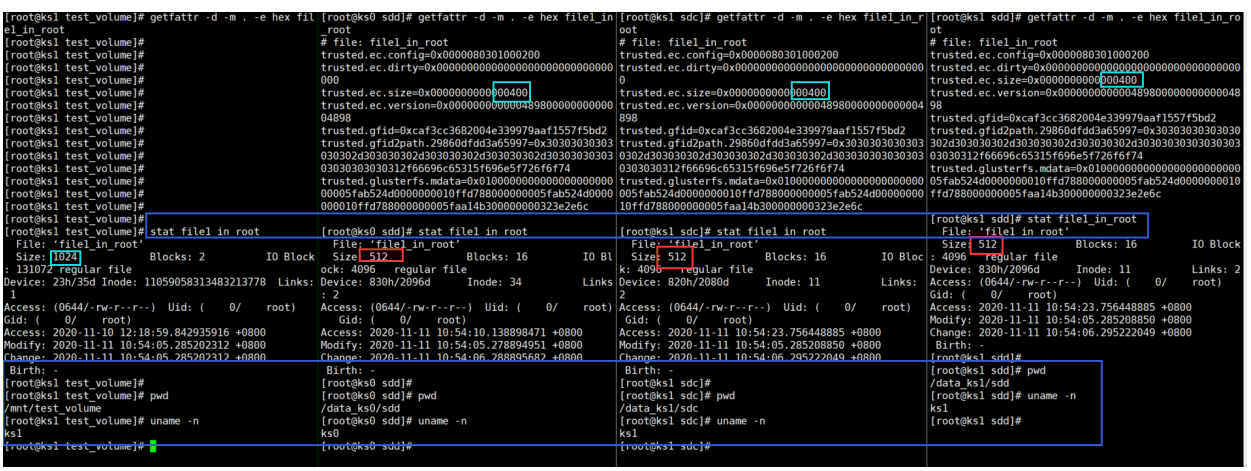
- trusted.glusterfs.mdata：ctime，mtime，atime (更新机制和修改机制不了解)
- trusted.glusterfs.dht：前8B rebalance的版本号，后8B[4B-4B]是layout[start-end]
- trusted.ec.dirty：前8B是data的dity的累计量，后8B是meta的dirty的累计量
  - 修改之前，dirty加1
  - 修改后update_xattr：如果所有的brick是好的，dirty减1，如果有brick无法访问，dirty维持加1

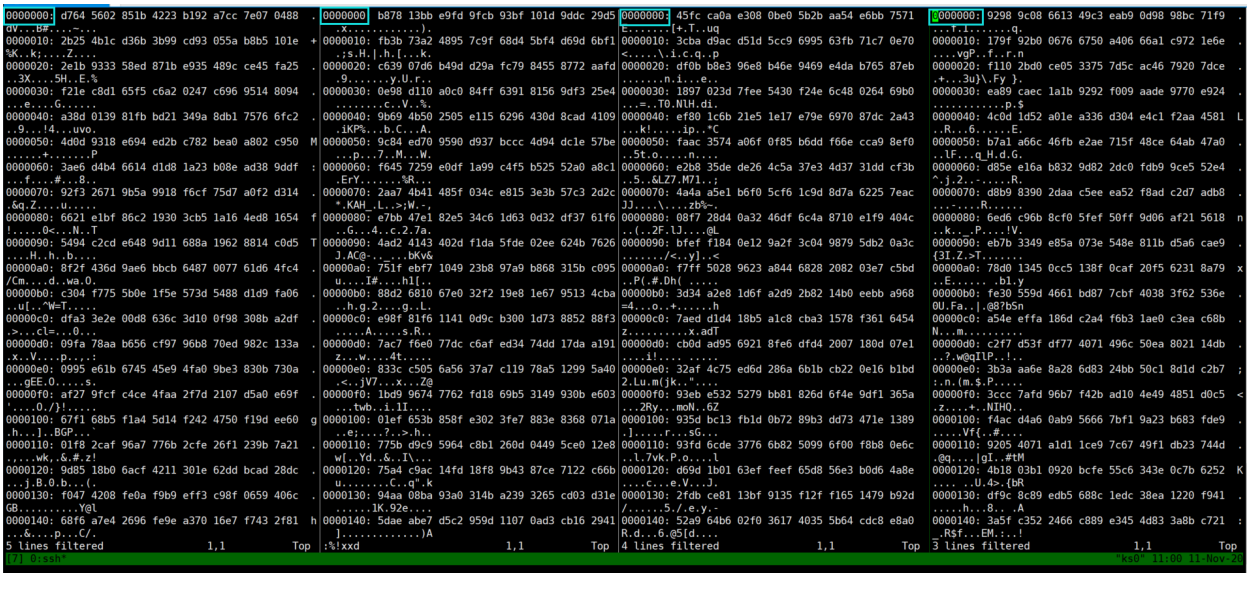
EC 数据块的姿势

由于glusterfs EC使用的非系统编码，存储的数据块都是经过编码后的数据，IO操作都是以条带stripe为单位



brick数据对比：





Fragment size 条块大小512B(EC_METHOD_CHUNK_SIZE)，Total number of fragments(stripe) 一个条带是512*N (N = K + R，一个条块中，K表示能够恢复数据的最小Fragment个数，R最冗余个数)

由于存的数据都是编码后的数据，每次读都需要decode；由于编码方式的原因，所以不会存在读改写，重构写等情况，每次编码都需要整个条带stripe参与计算

eg: 2+1模式

$$\begin{pmatrix}1 & 1\\ 1 & 2\\ 1 & 4\end{pmatrix}*\begin{pmatrix}D1\\ D2\end{pmatrix}=\begin{pmatrix}P1\\ P2\\ P3\end{pmatrix} \qquad => \qquad \begin{pmatrix}D1+D2\\ D1+2D2\\ D1+4D2\end{pmatrix}=\begin{pmatrix}P1\\ P2\\ P3\end{pmatrix}$$

通过P1和P2恢复数据：

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 1 & 2\end{pmatrix}=(P1 \quad P2)$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 1 & 2\end{pmatrix}=(P1 \quad P2)*\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 0 & 3\end{pmatrix}=(P1 \quad P2)*\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}=(P1 \quad P2)*\begin{pmatrix}1 & 0\\ 6 & 6\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}=(P1 \quad P2)*\begin{pmatrix}7 & 6\\ 6 & 6\end{pmatrix}$$

eg :

$$D1=3, D2=7 \qquad code => \qquad P1=4, P2=6$$

$$D1=4*7+6*6=g^2*g^5+g^4*g^4=g^7+g^1=1+2=3$$

$$D2=4*6+6*6=g^2*g^4+g^4*g^4=g^6+g^1=5+2=7$$

通过P2和P3恢复数据

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 2 & 4\end{pmatrix}=(P2 \quad P3)$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 2 & 4\end{pmatrix}=(P2 \quad P3)*\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 1 & 2\end{pmatrix}=(P2 \quad P3)*\begin{pmatrix}1 & 0\\ 0 & 5\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 0 & 3\end{pmatrix}=(P2 \quad P3)*\begin{pmatrix}1 & 0\\ 1 & 5\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}=(P2 \quad P3)*\begin{pmatrix}1 & 0\\ 6 & 3\end{pmatrix}$$

$$(D1 \quad D2)*\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}=(P2 \quad P3)*\begin{pmatrix}7 & 3\\ 6 & 3\end{pmatrix}$$

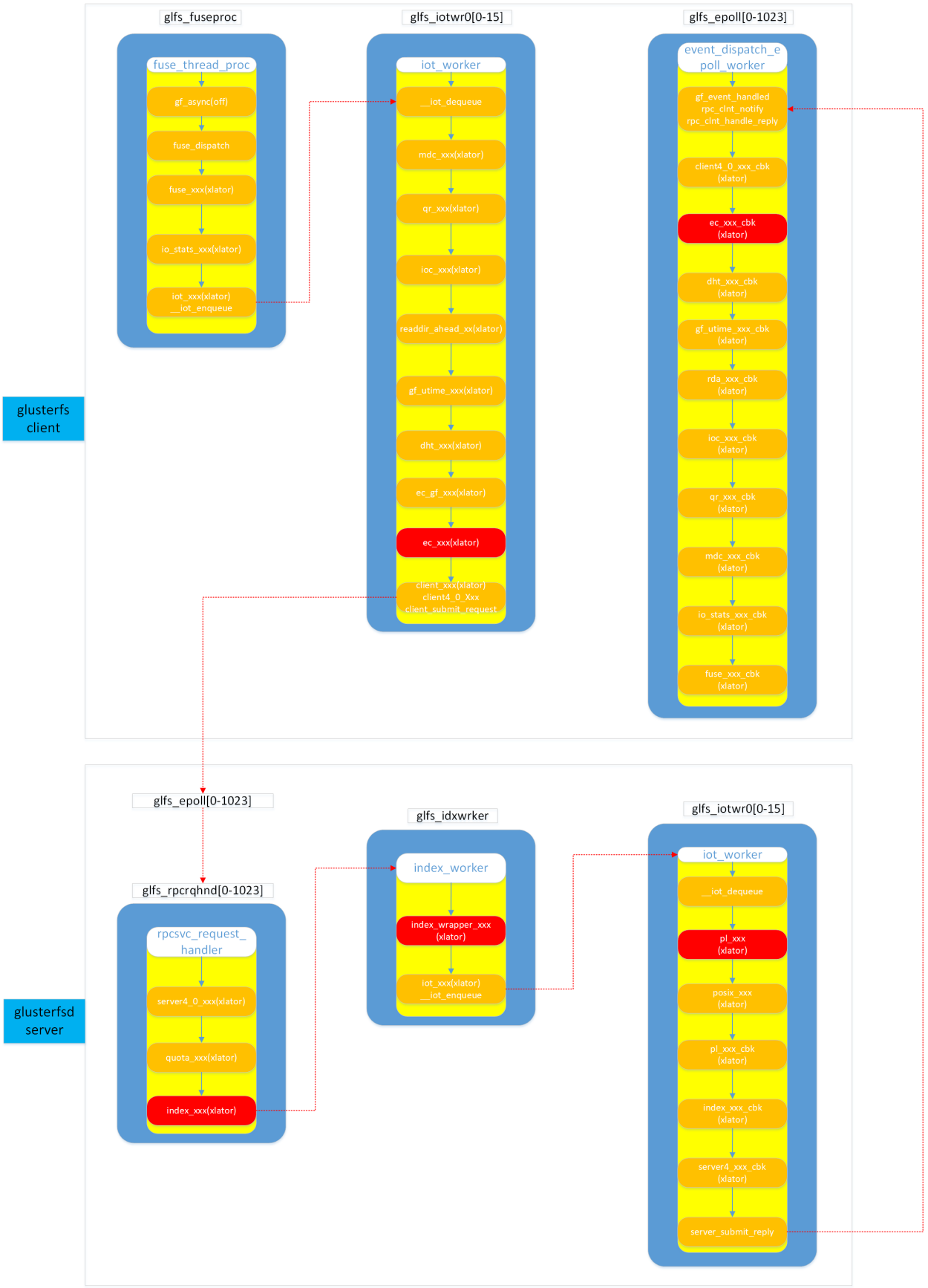
eg :

$$D1=3, D2=7 \qquad code => \qquad P2=6, P3=2$$

$$D1=6*7+2*6=g^4*g^5+g^1*g^4=g^2+g^5=4+7=3$$

$$D2=6*3+2*3=g^4*g^3+g^1*g^3=g^7+g^4=1+6=7$$

线程模型



- glusterfs-client

```
[root@ks1 ~]# cat /proc/`ps axu | grep glusterfs | grep fuse | grep -v grep | awk
'{print $2}'`/task/*/comm | sort
glfs_epoll000
```

```
glfs_epoll001
glfs_fusedlyd
glfs_fusenoti
glfs_fuseproc
glfs_iotwr000
glfs_memsweep
glfs_sigwait
glfs_sproc0
glfs_sproc1
glfs_timer
glusterfs
glusterfs
```

    - 线程信息：
        - glfs_sproc: api syncenv_scale
        - glfs_fuseproc: read /dev/fuse, get fuse_request_msg
        - glfs_iotwr00[0-15]: 核心业务线程，转接业务逻辑
        - glfs_epoll00[0-1023]: 处理 glusterfs-server-reply
    - xlator:
        - fuse: 读写/dev/fuse，获取来自kernel fuse的request，或者将reply写回到/dev/fuse，给kernel_fuse回包
        - io_stats：posix请求统计
        - iot: 来自fuse请求派发到 glfs_iotwrxxx线程处理
        - mdc：meta_cache，默认超时时间是1s，
        - qr: data quick read, 默认超时1s,
        - ioc: 没有超时时间，类似vfs pagecache
        - rda: readdir ahead 不了解
        - gf_utime: 更新xattr的atime, mtime
        - dht: 通过全路径名，计算hash值，定位所属subvolume
        - ec_gf: 阻止通过挂点，通过xattr语义操作内部xattr的
        - ec: ec逻辑
        - client: 封包解包对应posix语义逻辑，send request to server

- glusterfs-server

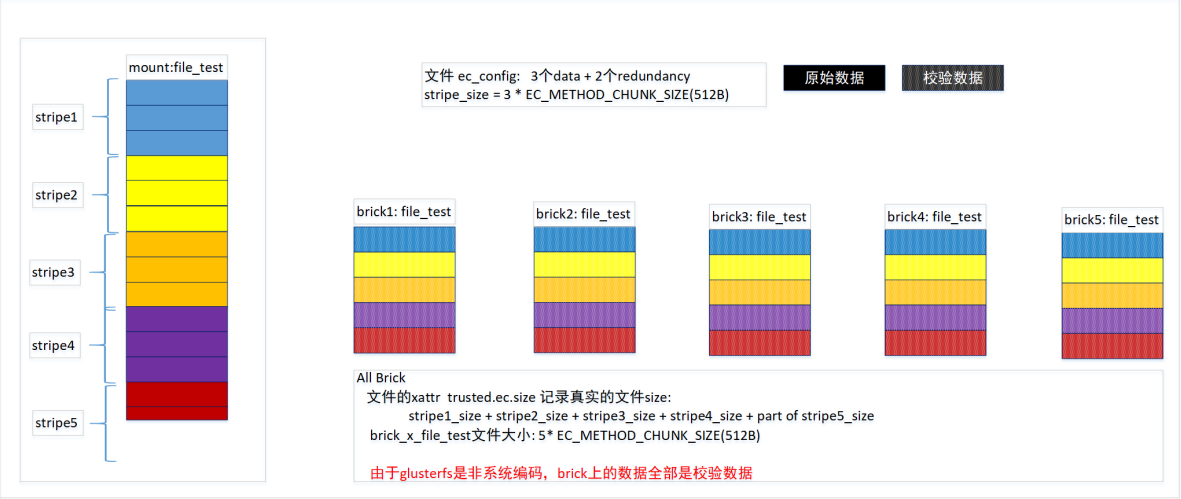
```
[root@ks1 systemtap]# cat /proc/`ps axu | grep glusterfsd | grep sdc | grep -v grep | awk '{print $2}'`/task/*/comm | sort
glfs_clogd000
glfs_clogd001
glfs_clogd002
glfs_clogecon
glfs_epoll000
glfs_epoll001
glfs_idxwrker
glfs_iotwr001
glfs_memsweep
glfs_posixctxja
glfs_posixfsy
```

```
glfs_posixhc
glfs_posix_rese
glfs_rpcrqhnd
glfs_rpcrqhnd
glfs_sigwait
glfs_sproc0
glfs_sproc1
glfs_timer
glusterfsd
glusterfsd
```

  - 线程信息：
    - glfs_rpcrqhnd: 收包处理glusterfs-client-request msg
    - glfs_idxwrker: 处理index xlator逻辑
    - glfs_iotwr00[0-15]: 核心业务线程，转接业务逻辑
  - xlator:
    - iot: 来自派发request msg对应业务逻辑到glfs_iotwrxxx线程
    - quoto：没看，不了解
    - index: 针对修改数据请求，pre_do在brick的brick/.glusterfs/indices/xattrop目录下创建文件，post_do删除创建的文件
    - pl: brick维护的file_lock和entry_lock
    - posix: 数据落地brick

数据结构和写逻辑

- 数据的存储方式:



- 基本数据结构

```
struct _loc {
    const char *path;  //全路径
    const char *name;  //name
    inode_t *inode;    //inode
    inode_t *parent;   //parent
    uuid_t gfid;       //gfid
    uuid_t pargfid;    //parent gfid
```

```
};

struct _inode {
    inode_table_t *table;         //全局inode_table，保存inode缓存
    uuid_t gfid;                  // gfid
    ia_type_t ia_type;            // inode type
    struct list_head fd_list;     // fd list
    struct list_head dentry_list; // dentry list if it is dir
    struct list_head hash;        // link to global itable
    struct _inode_ctx *_ctx;      /* replacement for dict_t *(inode->ctx) */
};

struct _xlator {
    char *name;       // name: mdc_xlator, iot_xlator等等
    char *type;
    xlator_t *next;  //平行结构：比如dht_xlator，每个xlaotr对应一个subvolume，用list链接起来
    xlator_t *prev;  //          还有：client端的 每个brick_client_xlator也是平行list链接起
来
    xlator_list_t *parents;      // parent xlator, eg: ec_xlator parent is ec_gl_xlator
    xlator_list_t *children;     // child xlator, , eg: ec_xlator children is
client4_xlator

    struct xlator_fops *fops;    // xlator的posix的fop
    struct xlator_cbks *cbks;    // xlator的posix的fop的回调
    struct xlator_dumpops *dumpops; // xlator的dump信息，eg:ec_dump_private
};


struct _ec_fop_data {
    int32_t id;         //fop的操作标识：eg writev的是GF_FOP_WRITE
    int32_t refs;
    int32_t state;      //fop的状态机：EC_STATE_INIT，EC_STATE_LOCK，EC_STATE_DISPATCH，
EC_STATE_PREPARE_ANSWER，EC_STATE_REPORT，EC_STATE_LOCK_REUSE，EC_STATE_UNLOCK
                        //             一般是以上几种，不同fop也有自己独有的state，eg：write
的有EC_STATE_DELAYED_START，编码派发
    uint32_t minimum;   // 当请求发送到各个brick成功的数据 大于等于mininum认为操作成功
                        switch (fop->minimum) {
                        case EC_MINIMUM_ALL:
                            fop->minimum = gf_bits_count(fop->mask);
                            if (fop->minimum >= ec->fragments) {
                                break;
                            }
                        case EC_MINIMUM_MIN:
                            fop->minimum = ec->fragments;
                            break;
                        case EC_MINIMUM_ONE:
                            fop->minimum = 1;
                        }
    }
    int32_t expected;//期望成功的brick_index的占位符，基本没有逻辑判断，判断逻辑在
ec_update_good会使用，如果expected为1，不进行heal检查
                     //    这样的fop有：readdir，access，getxattr(来自引用的xattr语义)
    int32_t winds;   //已经发送的brick_index的占位符
    int32_t jobs;    //可以理解成子job的个数，如果ec_write在ec_manger根据状态机流转，进行相应
的操作，
                     //        如果需要加锁 那么需要执行ec_sleep将job数加1，等加锁成功执行回
```

```
调以后job才会减1，执行ec_write剩下的状态流转
                        //          当然加锁前和加锁后的状态机的执行可能已经不在一个线程了
    int32_t error;
    ec_fop_data_t *parent;// parent的fop，比如ec_writev生成fop在装机流转中，需要加锁，加锁会执行ec_inodelk也是生成fop，那么inodelk_fop->parent = ec_writev_fop
    xlator_t *xl;    /* points to EC xlator */

    uintptr_t mask;     //能够被发送clinet的占位符，eg: 要发给三个brick_server，就是0b 111，如果第0个brick无法建立链接，那么mask就是0b 110
    uintptr_t healing;  //brick正在执行healing的占位符，brick被heal的时候，client会把data_version的EC_SELFHEAL_BIT的位，置为1，设置到brick_file的trusted.ec.version中

    uintptr_t remaining;//还未发送的brick index的占位符
    uintptr_t received; //回包的brick index的占位符(包括执行错误的回包占位符)
    uintptr_t good;     //回包成功的brick index占位符

    ec_wind_f wind;       //语义的派发接口，eg： ec_writev流程生成的fop->wind = ec_wind_writev
    ec_handler_f handler; //执行fop的状态机某个状态 需要操作的逻辑一般是 ec_manager_xxxx，eg: ec_manager_writev
    ec_resume_f resume;   //中断的状态流转再次执行，eg: ec_writev执行ec_lock那么ec_writev无法再次前进，
                          //     需要等ec_lock执行回调之后，执行ec_resume_parent然后会执行fop->resume再次进行ec_writev的状态流转
    ec_cbk_t cbks;        // fop执行完毕的回调

    dict_t *xdata;        // 比如fop执行rpc请求过程前，非xattr的key设置进去，rpc编码过程中将key封包(eg: GF_GET_SIZE "get-size"等)
    dict_t *dict;         // 比如fop执行rpc请求过程前，把需要获取的xattr的key设置进去，rpc编码过程中将key封包(eg: trusted.ec.size，trusted.ec.version等)
    struct list_head answer_list;  // 链接所有的_ec_cbk_data
    struct list_head cbk_list;     // 将_ec_cbk_data返回值归类，比如相同返回值的cbk_data组合成一个cbk_data连到cbk_list上，并且一个cbk_data的一定是相同类型最多的
};

struct _ec_cbk_data {
    ec_fop_data_t *fop;           // 指向fop
    struct list_head answer_list; // 所有的cbk都链上fop->answer_list
    struct list_head cbk_list;    // 相同返回值cbk_data链到fop->cbk_list上(一种类型的返回值只会链一个cbk_data，一般是最后返回的那个cbk_data做combine操作，之前链上去的删掉)
    uint32_t idx;                 // brick的index
    int32_t op_ret;               // rpc的返回值
    int32_t op_errno;             // brick上的操作的返回值
    int32_t count;                // 相同返回值的cbk的个数
    uintptr_t mask;               // 相同类型返回的brick的index占位符

    dict_t *xdata;                // 保存非xattr的返回值
    dict_t *dict;                 // 保存xattr的返回值
    inode_t *inode;
};


ec_fop_data_allocate: 分配fop，状态机操作的主要结构
    (1) 确定IO类型: eg GF_FOP_WRITE，GF_FOP_READ等
    (2) 注册rpc派发函数ec_wind_xxxx(fop->wind)，注册状态机操作的主要逻辑ec_manager_xxxx(fop-
```

>handle)，注册ec_xxx的回调

ec_manager(__ec_manager)：状态机流转逻辑的执行fop->handle(fop-state)

ec_check_complete：检查子任务是否完成(fop->job==0)，如果完成，继续进行ec_manager继续状态机的流转，否则终止流转，等子任务完成唤醒状态机的流转
，比如状态机中有派发rpc到all brick，每次派发都会ec_sleep(fop->job++)，然后终止该状态机的继续进行，等到所有的rpc回调完成之后job变为0，继续进行之前的状态机流转

ec_manager_xxxx：状态机执行的主函数，比如ec_manager_writev

ec_lock_prepare_fd(ec_lock_insert)：加锁操作准备工作，生成ec_lock_link_t *link，注意这个锁是在client端的，放到fop->locks[]中

ec_lock:
(1) 获取fop->locks的link_lock
(2) 如果全局锁link->lock (inode->_inode_ctx->lock)加锁成功，
(1)和本client的其他link_lock没有范围冲突，认为加锁成功，link->owner_list add to lock->owners
(2) 和本client的其他link_lock有范围冲突，需要将link->wait_list add to lock->waiting，等被冲突的锁从lock->owners中移除，然后再次尝试加锁lock->waiting中的锁请求
(3) 如果全局锁link->lock (inode->_inode_ctx->lock)还没成功，所有的锁加入的lock->waiting，等待全局锁加锁成功后将wating的中锁在client中尝试加锁
(4) 如果全局加锁成功，获取xattr，stat等信息，如果是写操作设置dirty，写回到brick的file中，同时在brick的.glusterfs/indices/xattrop生成一个硬链接文件，文件名称是这个被写的文件gfid
(5) 执行ec_lock_acquired->ec_lock_wake_shared->ec_lock_resume_shared，查看是否有其他waiting的link_lock能被加锁成功

ec_xxx_cbk:
(1) 发送给brick的rpc的回调
(2) ec_cbk_data_allocate 生成ec_cbk_data_t，链上fop->fop->answer_list
(3) ec_combine(ec_combine_xxxx) 排列组合相同返回值的rpc回调，ec_combine_check和ec_combine_xxxx为判断是否一致的方法，看_ec_cbk_data说明
(4) ec_complete：检查所有的回调是否完成，如果全部的rpc回调都完成，并且有大于等于minimum的回调成功，并且有brick失败，并且fop->parnet==null<不是状态机中的子任务>那么进入调用ec_heal逻辑

ec_write：先加锁，然后如果条带未对齐，需要读也要加锁，为什么没有死锁？？？
(1) 因为write_fop是read_fop的父，父加锁了之后，子不需要加锁

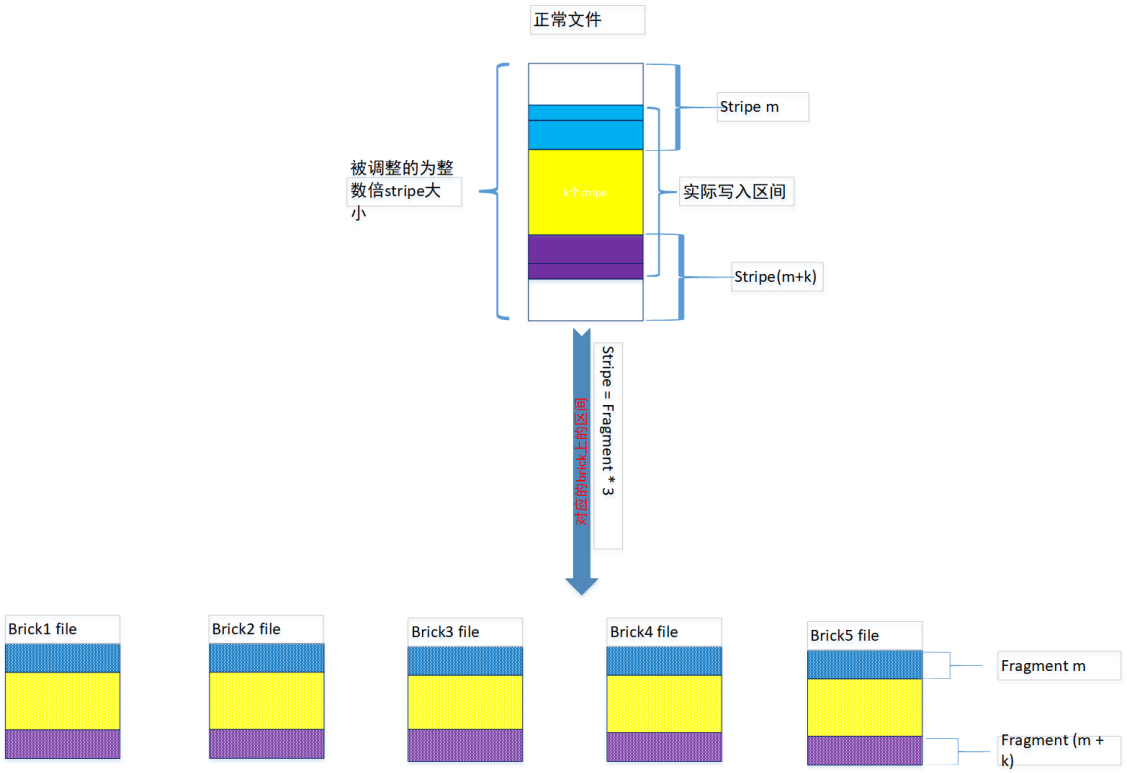
```
static void ec_lock_prepare_inode_internal(ec_fop_data_t *fop, loc_t *loc, uint32_t flags,
                                           loc_t *base, off_t fl_start, uint64_t fl_size)
{
    ec_lock_t *lock = NULL;
    ec_inode_t *ctx;

    // fop->parent不是null
    if ((fop->parent != NULL) || (fop->error != 0) || (loc->inode == NULL)) {
        return;
}
```

(2) 写逻辑发送之前会更新cache stripe，如果失败了怎么办，ec_lock_reuse逻辑会更新stripe逻辑，失效缓存条带

- 写逻辑

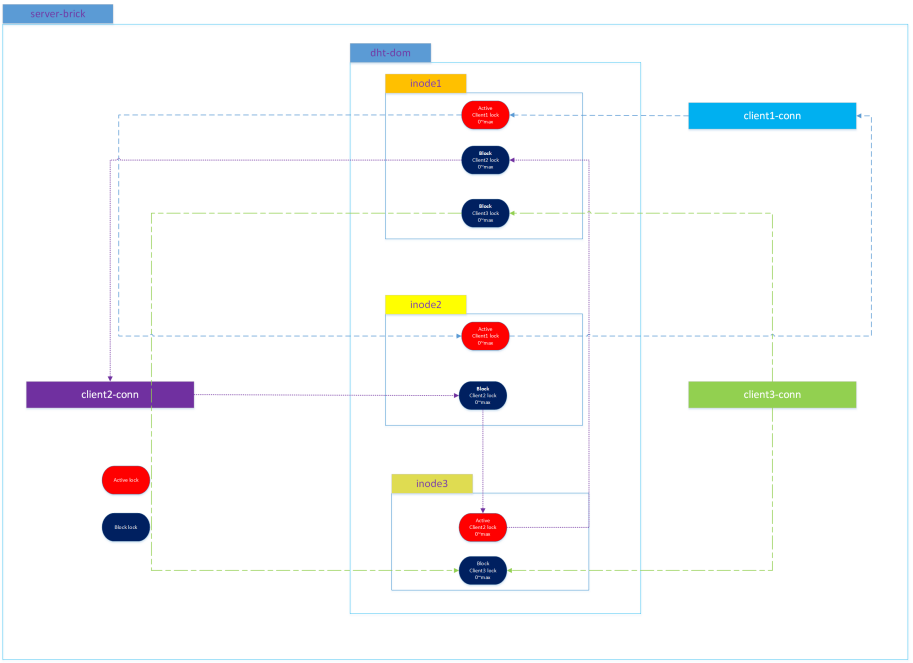
例子的ec_config是3+2模式, 条块为512B, 所以一个stipte是512B*3

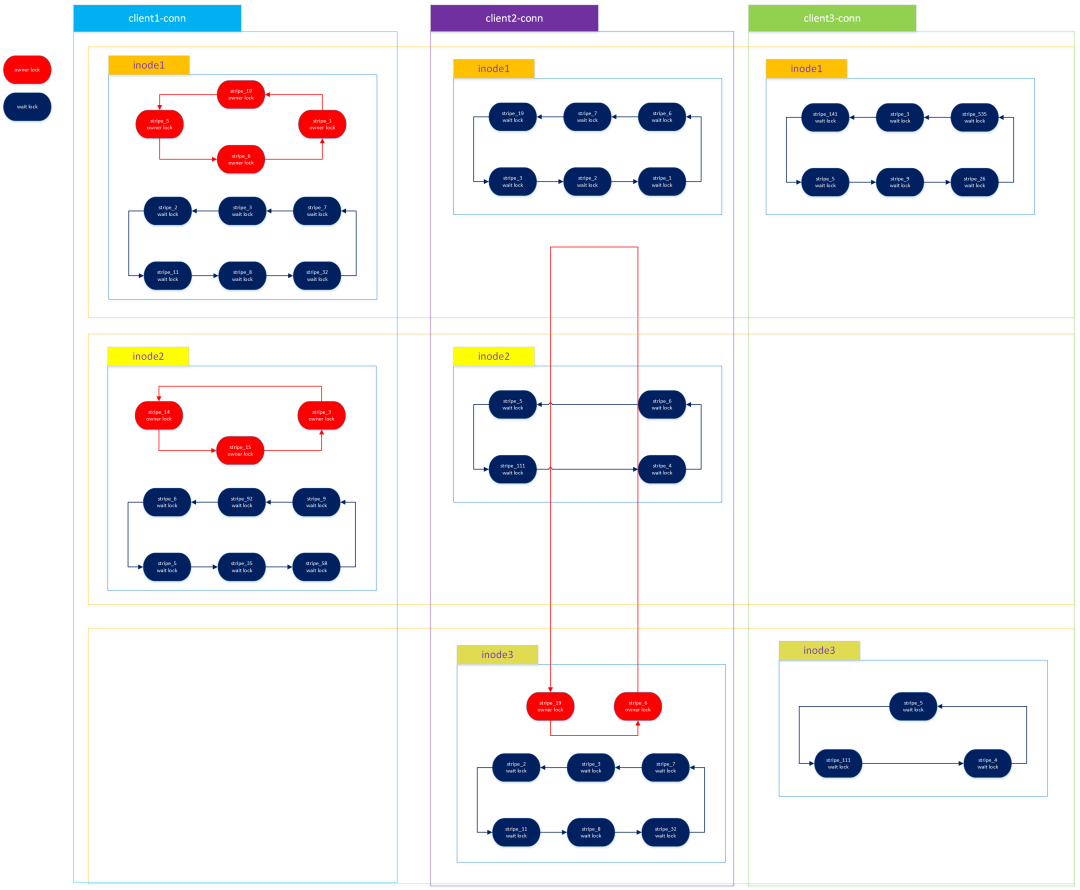
- 状态机：
  - EC_STATE_INIT，EC_STATE_LOCK：
    - offset和size调整，确定加锁范围
    -




```
pwrite:
fl_start:
    fl_start = fop->offset;
    head = ec_adjust_offset_down(ec, &fl_start, _gf_true);
    scale=true, fl_start被调整为brick的文件的起始offset(Fragment整数倍), 比如offset在第8个stripe, 那么file_start被调整为8*Fragment

fl_size:
    fl_size = user_size + head;
    ec_adjust_size_up(ec, &fl_size, _gf_true);
    scale=true, fl_size被调整为在brick的文件需要被写入的file_size(Fragment整数倍)

write_append:
fl_start = 0
fl_size = LONG_MAX
```

    - 加锁： 对[fl_start, fl_size]区间加锁:ec_lock_prepare_fd, ec_lock等待加锁成功，执行EC_STATE_DISPATCH状态机





- 发送client全局锁[0~MAX_LEN]到server: F_SETLK 尝试获取，如果失败再次发送 F_SETLKW 阻塞式的获取

```
ec_lock_acquire(ec_lock_link_t *link) {
    //lk_owner是设置的值是 lock的地址值变化而来，只要全局lock不释放，在client端owner也是唯一的
    set_lk_owner_from_ptr(&lk_owner, lock);

    fop->flock.l_type = flock->l_type;
    fop->flock.l_whence = flock->l_whence;
```

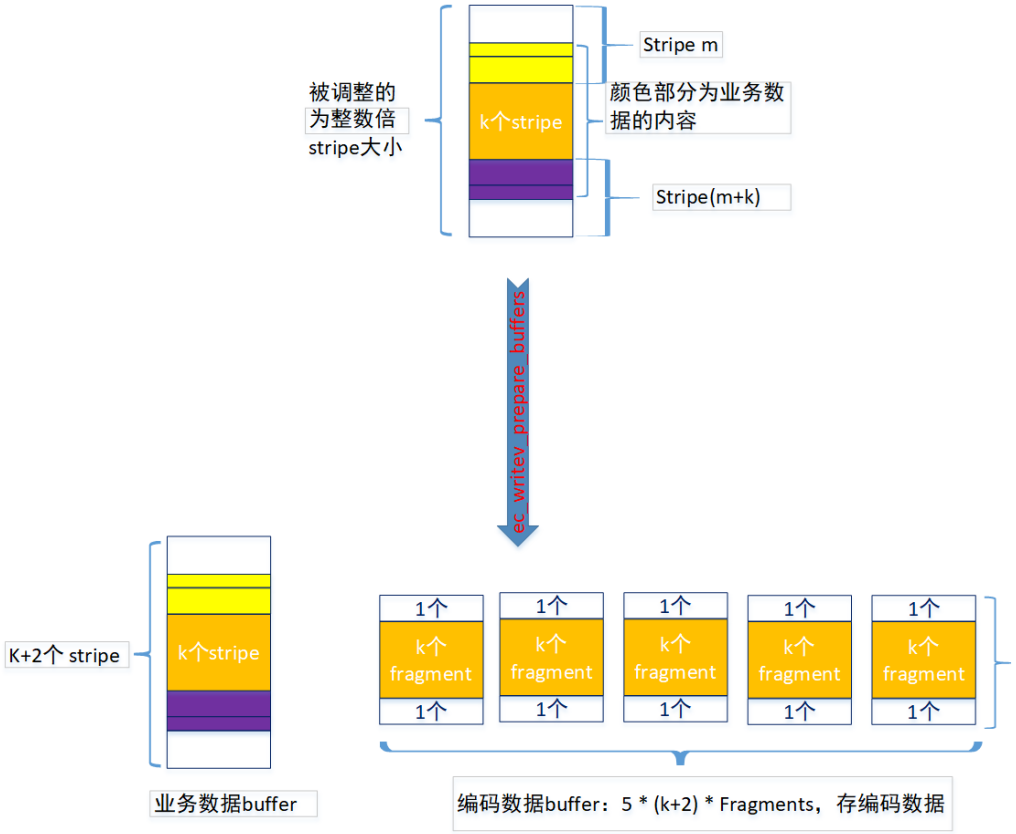
```
    fop->flock.l_start = flock->l_start;(=0)
    fop->flock.l_len = flock->l_len;(=0), //在server端会设置为[0~MAX]锁
    fop->flock.l_pid = flock->l_pid;(=0)
    fop->flock.l_owner.len = flock->l_owner.len;
    if (flock->l_owner.len > 0) {
        memcpy(fop->flock.l_owner.data, flock->l_owner.data, flock-
>l_owner.len);
    }
}
```

- 全局锁获取成功，尝试在客户端加[fl_start, fl_size]的锁，如果有锁冲突，等待冲突的锁解锁的时候，再次尝试加waiting_lock

```
    //全局加锁成功，会尝试加锁link_waiting_lock<冲突等待的锁>，全局锁为获取之前，
所有client的局部所都是link_waiting_lock
    //解锁client局部锁的时候，会加锁link_waiting_lock<冲突等待的锁>
    ec_lock_wake_shared(lock, &list);
    ec_lock_resume_shared(&list);
```

- 发送xattrop到brick：
  - 主要更新xattr的dirty +1，然后server端的index features会在/brick/.glusterfs/indices/xattrop/下生成一个hardlink文件，文件名称是gfid
  - 回包中获取最新xattr信息(ec.dirty, ec.size, ec.version)，获取brick的记录数据的文件的真实size

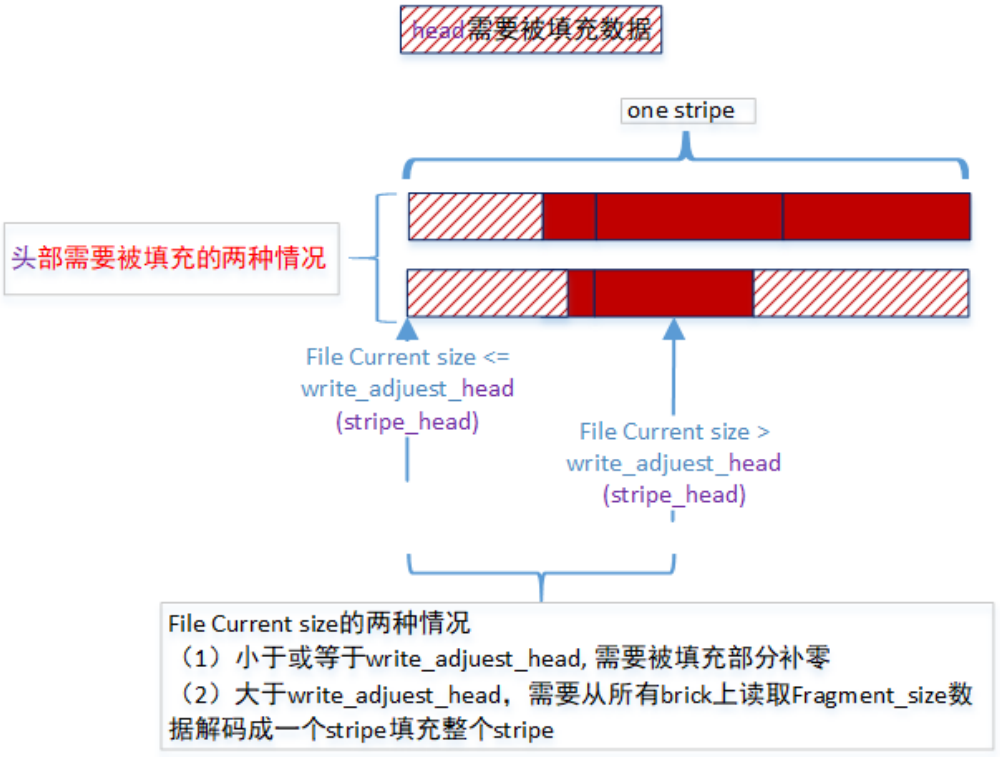
- EC_STATE_DISPATCH: 并没有dispatch还是准备工作 ec_writev_start
  - ec_writev_prepare_buffers 构造业务数据buffer和编码数据buff

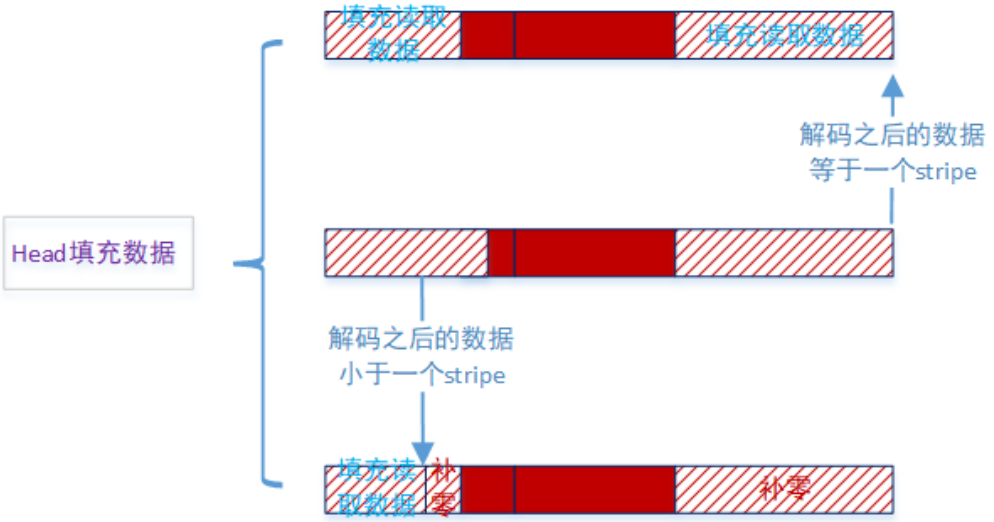


- 业务数据buffer: 如果head或者tail不是按照stripe对齐，或者业务数据的地址值不是64B对齐，都需要申请buff，buff_size按照条带对齐的size，并且buff_addr是64B对齐的，然后拷贝业务数据到申请的buff中，buff_offset为业务数据在条带中位置
- 构造编码数据所需的buff，大小为 stripe_num * Fangment_szie * (数据个数 + 冗余个数)

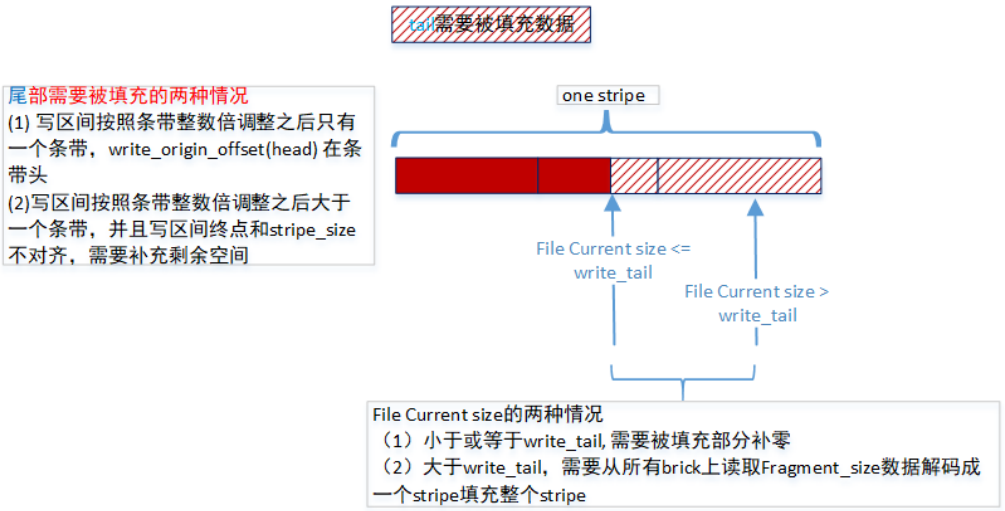
```
fop->vector[0].io_base 业务数据
fop->vector[1].io_base 编码数据

fop->frag_range.first 为操作brick上的文件的起始位置，按照fragment对齐
fop->frag_range.last  为操作brick上的文件的结束位置，按照fragment对齐
```

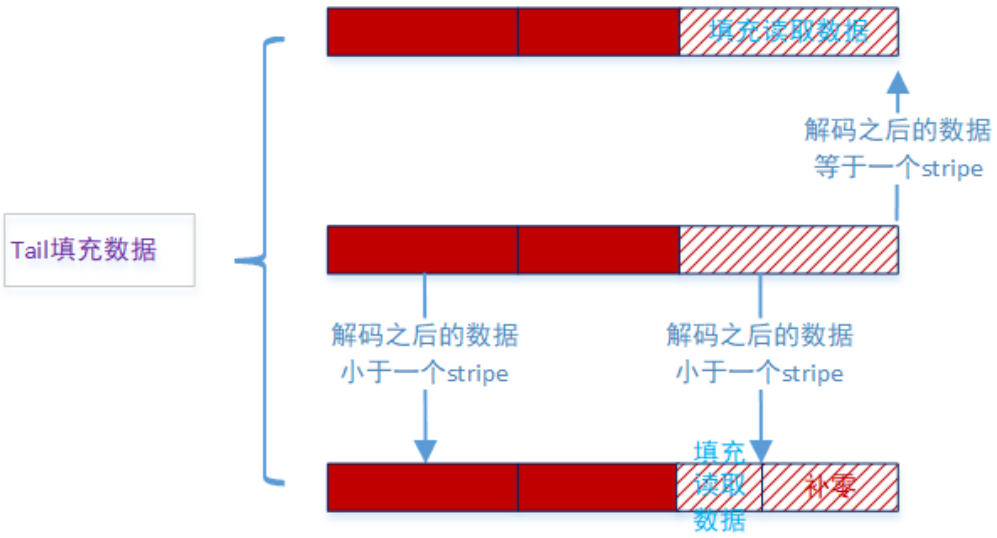
- 补齐数据，同时更新head和tail到stripe_cache中
  - head



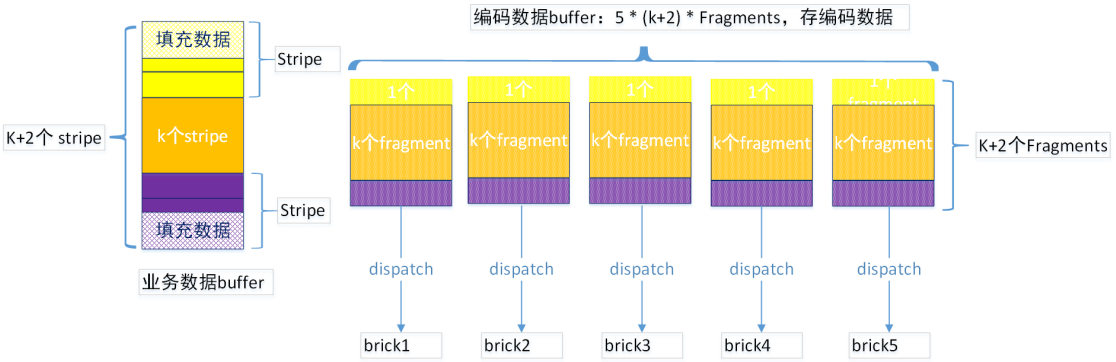


- tail





- EC_STATE_DELAYED_START: 编码后的数据派发到brick




```
//编码业务数据
ec_writev_encode(fop);
//派发给所有brick
ec_dispatch_all(fop);
//调用rpc派发给brick
ec_wind_writev
// rpc回调，op_ret为写入单个brick中大小，必须是fragment_size的整数倍
ec_writev_cbk
```

- EC_STATE_PREPARE_ANSWER：等待所有发送给brick请求返回，更新iatt和file_size
- EC_STATE_REPORT: 执行ec_writev注册的回调default_writev_cbk返回给上一层的xlator
- EC_STATE_LOCK_REUSE: 执行ec_lock_reuse
  - 写请求中获取brick中inode的全局锁count(包含block lock), count大于1，置上立即release的标记位，让其他client不至于加锁饿死
  - ec_update_cached_stripes：更新条带，如果写请求失败，从cache中remove掉命中写区间的所有stripe_cache，如果成功加stripe数据加入到lru cache(ec_writev_start只更新了头部或者尾部的stripe)
  - ec_lock_next_owner:
    - 释放client端的[fl_start, fl_end]的局部锁
    - inode->post_version++(内存变量加1，统一到释放全局锁的逻辑中写入到brick的xattr中)
    - 尝试加锁link_waiting_lock: 之前因为冲突被阻塞的局部锁
- EC_STATE_UNLOCK：给全局锁释放添加定时器中(1s到期)
  - ec_unlock_timer_add：
    - 如果还有局部锁未释放，退出
    - 如果局部锁全部释放：添加定时器(1s)，到期后释放全局锁
    - 如果全局锁需要立即释放(远端lock_count > 1或者ec_shutdown等等)立即释放全局锁
  - 如果全局锁已经被标记为释放，局部锁还有加锁请求，将局部锁加到lock->frozen中，等到全局锁放的回调中，再次尝试加全局锁，如果成功同时加局部锁
- ec_unlock_now：释放全局锁(没有局部锁持有全局锁)
  - 将内存总维护的xatrr信息：trusted.ec.size, trusted.ec.version, trusted.ec.dirty(-1)更新到brick, xattr的更新是累积量更新的在brick更新的，比如

trusted.ec.dirty + (-1)，可以将之前的全局锁成功之后设置的xattr.dirty+1-1变为之前的值

    - 如果写请求在某个brick中失败，trusted.ec.dirty不设置到rpc中，让brick的index模块继续保留在.glusterfs/indices/xattrop/中创建的硬链接文件，让index heal能够快速知道需要被修复的文件(brick起来之后，会事件的方式通知heal-daemon执行修复，或者heal-daemon周期性的修复10分钟)
    - ec_unlock_lock：
        - ec_clear_inode_info：清理全局锁管理的stripe缓存，version，size，dirty等信息
        - 到brick中执行释放全局锁操作

- 其他问题：由于默认配置，更新xattr: trusted.ec.versoin, trusted.ec.size是释放锁的时候一次更新，那么怎么知道那个数据是最新的了？？？

  客户端加锁发送到server：0~max的大锁 成功

  然后客户端获取xattr信息：version, dirty, size等

  然后客户端的写操作只在 client加锁，在内存中维护客户端的并发操作

  每次fuse_xxx的操作 按照xlator的依次执行，会到gf_utime_xxx中进行gl_timespec_get(&frame->root->ctime)操作，设置ctime，然后到ec_xxx中执行具体业务逻辑，然后派发到brick, ctime时间会设置到rpc的head字段中

  如果出现brick执行写操作的过程中，已近连续写了很多数据， 但是但是由于全局锁还没有释放，还不会更新xattr的数据 但是每次brick的写操作会更新xattr的trusted.glusterfs.mdata，这个表示最新的数据

```
client端:

client4_0_xxx
    client_submit_request
        rpc_clnt_submit
            rpc_clnt_record
                rpc_clnt_record_build_record
                    rpc_clnt_fill_request
                        xdr_serialize_glusterfs_auth
                            setup_glusterfs_auth_param_v3
                                   au->flags = frame->root->flags;
                                   au->ctime_sec = frame->root->ctime.tv_sec;
                                   au->ctime_nsec = frame->root->ctime.tv_nsec;

server端:

rpcsvc_request_create
    rpcsvc_authenticate
        auth->authops->authenticate(auth_glusterfs_v3_authenticate)
            rpc_receive_common
                get_frame_from_request
                    {
                        frame->root->uid = req->uid;
```

```
                        frame->root->gid = req->gid;
                        /*client的 执行读写操作的操作的pid*/
                        frame->root->pid = req->pid;
                        frame->root->client = client;
                        frame->root->lk_owner = req->lk_owner;

                        if (priv->server_manage_gids)
                            server_resolve_groups(frame, req);
                        else
                            server_decode_groups(frame, req);
                        trans = req->trans;
                        if (trans) {
                            memcpy(&frame->root->identifier, trans-
>peerinfo.identifier,
                                   sizeof(trans->peerinfo.identifier));
                        }

                        /* more fields, for the clients which are 3.x series this
will be 0 */
                        frame->root->flags = req->flags;

                        //设置时间
                        frame->root->ctime = req->ctime;

                        frame->local = req;

                        state = CALL_STATE(frame);
                        state->client = client;
                    }
```

posix_xxx（需要更新ctime或者mtime的语义，会写入到rpc的flags中传递到 server端进行xattr的trusted.glusterfs.mdata的设置）

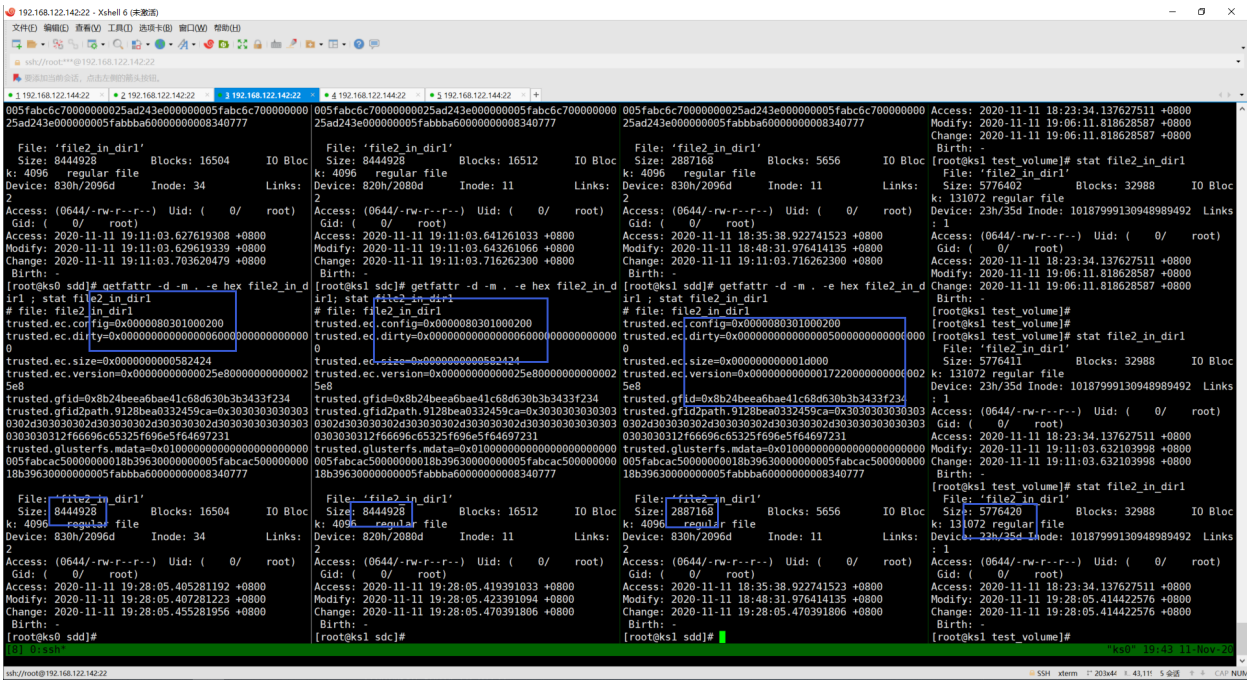
posix_set_ctime

posix_set_mdata_xattr

在修复确定以那些brick作为基准的修复源的时候，会比较mtime，相同的mtime个数大于等于qurom的时候，才能进行修复

```
int ec_heal_metadata_find_direction(ec_t *ec, default_args_cbk_t *replies,
                                    uint64_t *versions, uint64_t *dirty,
                                    unsigned char *sources,
                                    unsigned char *healed_sinks)
{
     if (!are_dicts_equal(replies[i].xdata, replies[j].xdata,
                                     ec_sh_key_match, NULL))
}
```

- 无法自动修复的问题： 操作：比如2+1，(1) dd过程中 kill掉一个brick的进程, (2) 然后kill掉glusterfs-fuse进程



会出现这样一情况：xattr sdd和sdc一样， sdd和其他两个不一样(被kill的brick)

不能修复的原因：虽然存在两个brick一致，但是xattr中记录的trusted.ec.size和brick上的校验数据的文件大小不能满足一定关系：

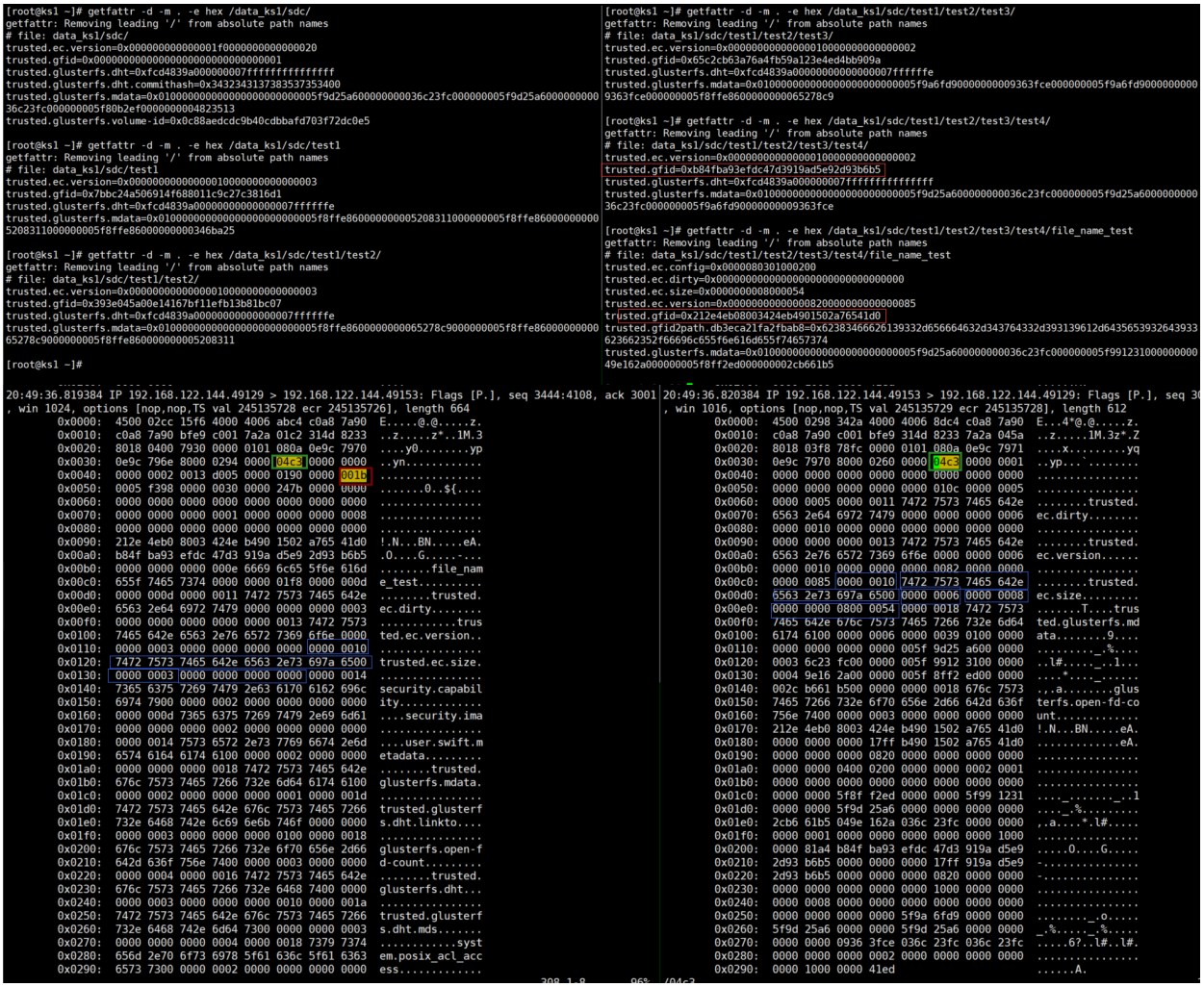
```
ec_heal_data_find_direction(check_ondisksize=true)


    if (check_ondisksize) {
        source_size = size[source];
        //将ec_file_size调整为brick上的文件真实size, brick_file_size = ec_file_size(条带对齐后size) / ec->fragments<quorum>
        ec_adjust_size_up(ec, &source_size, _gf_true);

        for (i = 0; i < ec->nodes; i++) {
            if (sources[i]) {
                if (replies[i].stat.ia_size != source_size) {
                    sources[i] = 0;
                    healed_sinks[i] = 1;
                    max_same_count--;
                } else {
                    source = i;
                }
            }
        }
        if (max_same_count < ec->fragments) {
            ret = -EIO;
            goto out;
        }
    }
```

- echo文件的网络抓包

glusterfs的挂载点在ks1上
要抓的server也在ks1上，抓 /data_ks1/sdc

```
[root@ks1 ~]# lsof /var/log/glusterfs/mnt-test_volume.log
COMMAND      PID USER   FD   TYPE DEVICE SIZE/OFF     NODE NAME
glusterfs 124821 root    5w   REG    8,3 48127101 10232946 /var/log/glusterfs/mnt-
test_volume.log
[root@ks1 ~]#
[root@ks1 ~]# ps axu | grep glusterfs | grep sdc
root     116318  0.0  1.2 1312824 21508 ?       Ssl  16:40   0:08 /usr/sbin/glusterfsd -s
ks1 --volfile-id test_volume.ks1.data_ks1-sdc -p /var/run/gluster/vols/test_volume/ks1-
data_ks1-sdc.pid -S /var/run/gluster/619b4b93af824dec.socket --brick-name /data_ks1/sdc -
l /var/log/glusterfs/bricks/data_ks1-sdc.log --xlator-option *-posix.glusterd-
uuid=1cb6cb30-f636-4079-82a3-df97b2eb5cab --process-name brick --brick-port 49153 --
xlator-option test_volume-server.listen-port=49153
[root@ks1 ~]#
[root@ks1 ~]# netstat -tpn | grep 124821 | grep 49153
tcp        0      0 192.168.122.144:49129   192.168.122.144:49153   ESTABLISHED
124821/glusterfs
tcp        0      0 192.168.122.144:49139   192.168.122.142:49153   ESTABLISHED
124821/glusterfs
[root@ks1 ~]#
[root@ks1 ~]# netstat -tpn | grep 124821 | grep 49153 | grep "122.144"
tcp        0      0 192.168.122.144:49129   192.168.122.144:49153   ESTABLISHED
124821/glusterfs
tcp        0      0 192.168.122.144:49139   192.168.122.142:49153   ESTABLISHED
124821/glusterfs

// ks1 brick:sdc的 glusterfs-clinet.port <----> glusterfs-server.port
netstat -ntp | grep `ps aux | grep glusterfs | grep sdc | awk '{print $29}'` | grep -w
glusterfs | grep -v "122.142" | grep `lsof /var/log/glusterfs/mnt-test_volume.log | awk
'{print $2}'| grep -v PID`

[root@ks1 ~]# tcpdump -i ens34 -Xnpls0 -iany tcp  dst port 49153 and src port 49129 >
/root/requset.txt //抓包request
[root@ks1 ~]# tcpdump -i ens34 -Xnpls0 -iany tcp src port 49153 and dst port 49129 >
/root/reply.txt    //抓包reply

tcpdump -i ens34 -Xnpls0 -iany src host 192.169.122.145 and src port 49142 and dst port
49154


比如这个报文：  client的lookup
client4_0_lookup:  GFS3_OP_LOOKUP就是001b
        ret = client_submit_request(this, &req, frame, conf->fops, GFS3_OP_LOOKUP,
                                    client4_0_lookup_cbk, &cp,
                                    (xdrproc_t)xdr_gfx_lookup_req);
                -> rpc_clnt_submit -> rpc_clnt_record -> rpc_clnt_record_build_header ->
rpc_clnt_record_build_header -> rpc_request_to_xdr

rpc_clnt_prog_t clnt4_0_fop_prog = {
    .progname = "GlusterFS 4.x v1",
    .prognum = GLUSTER_FOP_PROGRAM,         //1298437 -> 0x 0013 d005
    .progver = GLUSTER_FOP_VERSION_v2,      // 400    -> 0x 0190
    .numproc = GLUSTER_FOP_PROCCNT,
    .proctable = clnt4_0_fop_actors,
```

```
    .procnames = clnt4_0_fop_names,
};


20:25:07.685221 IP 192.168.122.144.49139 > 192.168.122.142.49153: Flags [P.], seq
2132:2788, ack 1977, win 331, options [nop,nop,TS val 224788688 ecr 224789064], length
656
        0x0000:  4500 02c4 daab 4000 4006 e718 c0a8 7a90  E.....@.@.....z.
        0x0010:  c0a8 7a8e bff3 c001 7eeb 807a 7951 72e1  ..z.....~..zyQr.
        0x0020:  8018 014b 7926 0000 0101 080a 0d66 00d0  ...Ky&.......f..
        0x0030:  0d66 0248 8000 028c 0000 00be 0000 0000  .f.H............     // 0000
00be是msg的xid
        0x0040:  0000 0002 0013 d005 0000 0190 0000 001b  ................     // 0000
001b就是GFS3_OP_LOOKUP, 0013 d005就是 GLUSTER_FOP_PROGRAM
        0x0050:  0005 f398 0000 0030 0001 ed24 0000 0000  .......0...$....
        0x0060:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0070:  0000 0000 0000 0001 0000 0000 0000 0008  ................
        0x0080:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0090:  65c2 cb63 a76a 4fb5 9a12 3e4e d4bb 909a  e..c.jO...>N....     //parent的
gfid
        0x00a0:  393e 045a 00e1 4167 bf11 efb1 3b81 bc07  9>.Z..Ag....;...     // test3的
gfid
        0x00b0:  0000 0000 0000 0005 7465 7374 3300 0000  ........test3...     //
entry_name
        0x00c0:  0000 01f8 0000 000d 0000 000d 0000 0011  ................

clinet ping包:

struct rpc_clnt_program clnt_ping_prog = {
    .progname = "GF-DUMP",
    .prognum = GLUSTER_DUMP_PROGRAM,
    .progver = GLUSTER_DUMP_VERSION,
    .procnames = clnt_ping_procs,
};

0x 075b b86d = 123,451,501 , GLUSTER_DUMP_PROGRAM 0x 075b b86d
0x 0000 0001 = 1, GLUSTER_DUMP_VERSION
0x 0000 0002 = 2, GF_DUMP_PING

11:18:34.773232 IP 192.168.122.144.49144 > 192.168.122.144.49153: Flags [P.], seq
188:272, ack 1, win
024, options [nop,nop,TS val 300995467 ecr 300981188], length 84
        0x0000:  4500 0088 d02d 4000 4006 f3d0 c0a8 7a90  E....-@.@.....z.
        0x0010:  c0a8 7a90 bff8 c001 5f4a 62ab e642 cbbf  ..z....._Jb..B..
        0x0020:  8018 0400 76ec 0000 0101 080a 11f0 d38b  ....v...........
        0x0030:  11f0 9bc4 8000 0050 0000 02ea 0000 0000  .......P........
        0x0040:  0000 0002 075b b86d 0000 0001 0000 0002  .....[.m........
//GLUSTER_DUMP_PROGRAM, GLUSTER_DUMP_VERSION, GF_DUMP_PING
        0x0050:  0005 f398 0000 0028 0000 0000 0000 0000  .......(........
        0x0060:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0070:  0000 0000 0000 0000 0000 0004 0000 0000  ................
        0x0080:  0000 0000 0000 0000
```



request:

0000 04c3是 xid

0000 001b是 lookup

key: trusted.ec.size.

key_len是 0x 0010

key_str: 0x 7472 7573 7465 642e 6563 2e73 697a 6500

value_type: GF_DATA_TYPE_UINT -> 0x 0003

value: 0x 0000 0000 0000 0000

reply:

0000 04c3是 xid

key: trusted.ec.size.

key_len: 0x 0000 0010

key_str: 0x 7472 7573 7465 642e 6563 2e73 697a 6500

value_type: GF_DATA_TYPE_PTR -> 0x 0006

value_size: 0x 0008

value: 0x 0000 0000 0800 0054

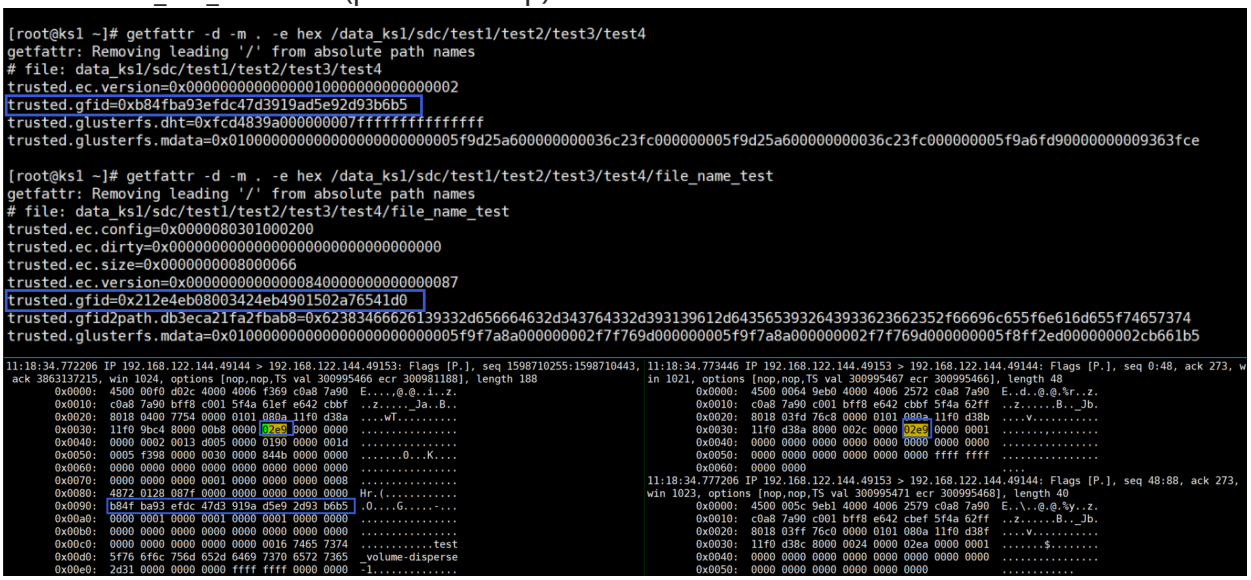
001b: GFS3_OP_LOOKUP (parent lookup)

0021: GFS3_OP_XATTROP (parent **get trusted**.ec.**version**[data **and** meta])

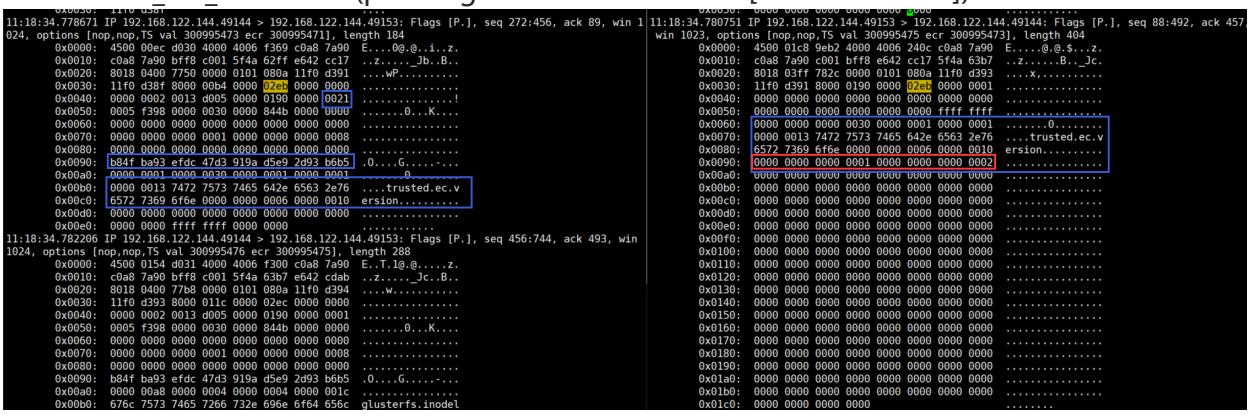
0001: GFS3_OP_STAT (parent stat, **get** glusterfs.inodelk-dom-count -> test_volume-

```
disperse-1)
001b: GFS3_OP_LOOKUP (file lookup, get trusted.ec.<version, dirty, size, mdata,
dht.linkto, dht, open-fd-count>)
000b: GFS3_OP_OPEN (file open get server_fd)
001d: GFS3_OP_INODELK (file lock)
0022: GFS3_OP_FXATTROP (file set xattr, set<add> trusted.ec.<dirty, config, version<未设
置>>)
000c: GFS3_OP_READ (file read all stripe of fragment)
000d: GFS3_OP_WRITE (write file stripe of fragment)
0022: GFS3_OP_FXATTROP (set xattr: set<add> trusted.ec.<size, version>)
000f: GFS3_OP_FLUSH (do nothing, get glusterfs.inodelk-count)
001d: GFS3_OP_INODELK (parent-dir lock)
0022: GFS3_OP_FXATTROP (set xattr: set<add -1> trusted.ec.dirty)
0029: GFS3_OP_RELEASE (file release)
```

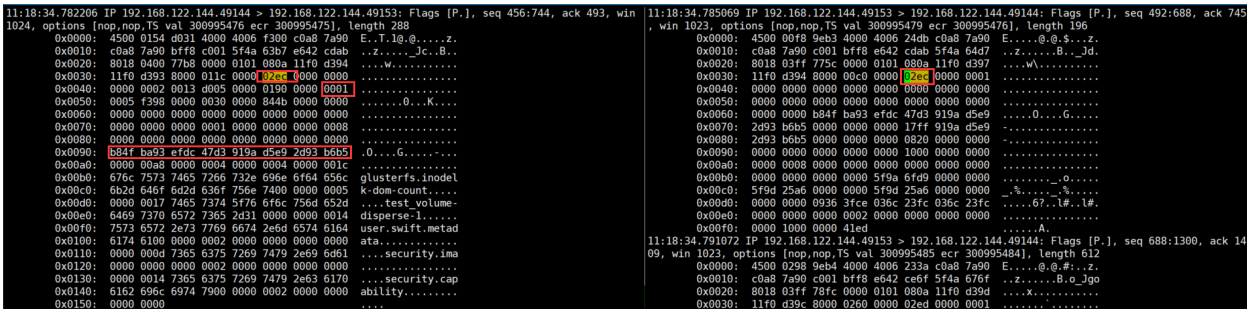
001b: GFS3_OP_LOOKUP (parent lookup)



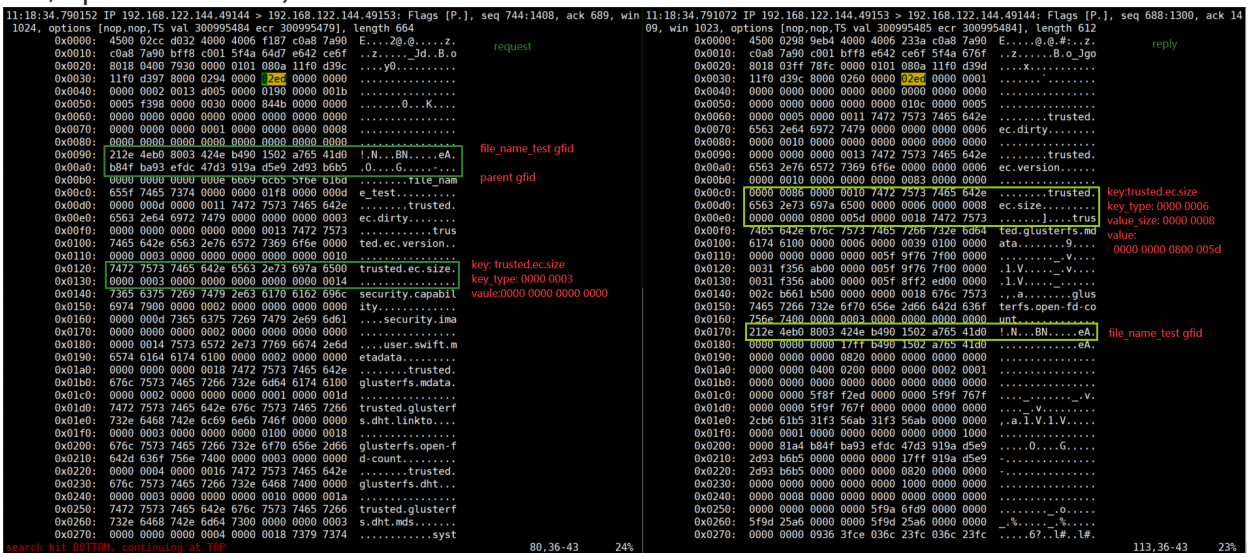
0021: GFS3_OP_XATTROP (parent get trusted.ec.version[data and meta])



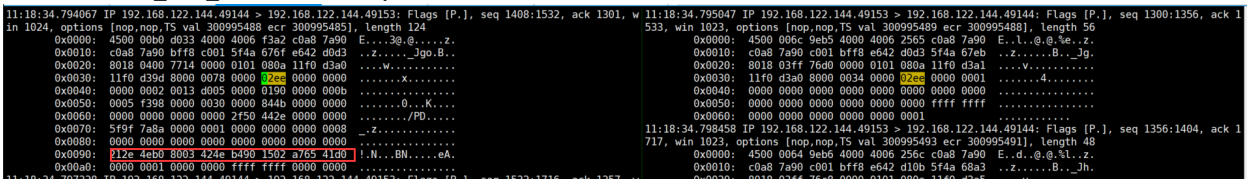
0001: GFS3_OP_STAT (parent statfs, get glusterfs.inodelk-dom-count -> test_volume-disperse-1)



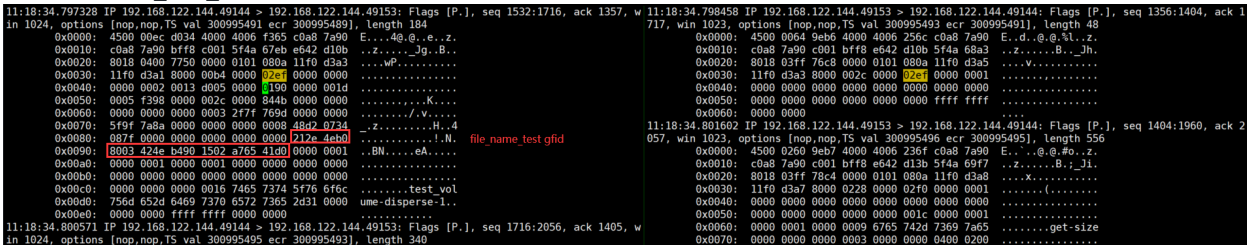
001b: GFS3_OP_LOOKUP (file lookup, get trusted.ec.<version, dirty, size, mdata, dht.linkto, dht, open-fd-count>)



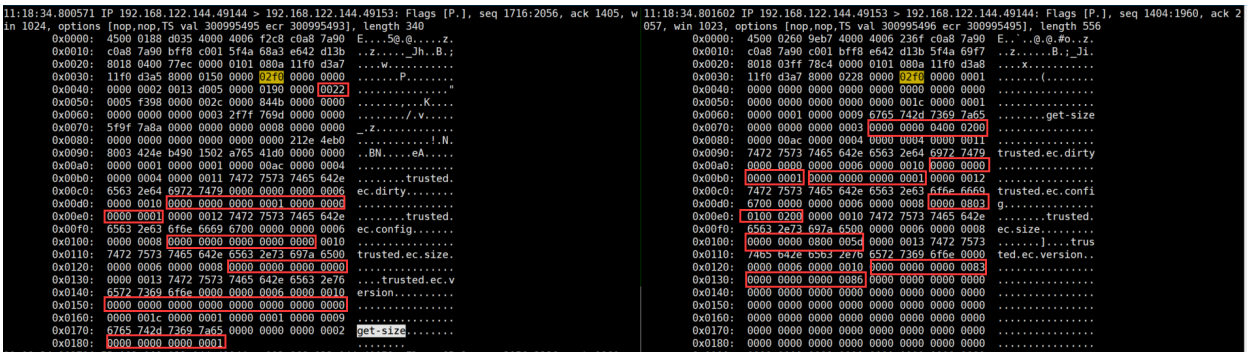
000b: GFS3_OP_OPEN (file open)



001d: GFS3_OP_INODELK (file lock)



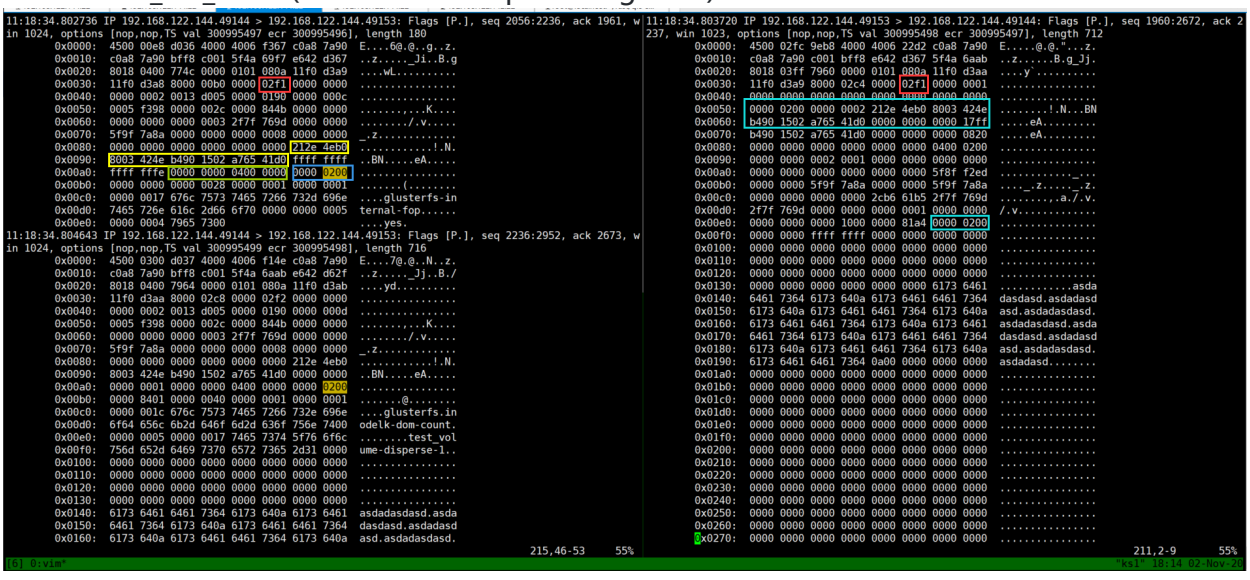
0022: GFS3_OP_FXATTROP (file set xattr, set<add> trusted.ec.<dirty, config, version<未设置>>)



request:
    xid: 02f0
    fop: 0022
    trusted.ec.dirty: add 1 (data, meta)<0x 0000 0000 0000 0001, 0x 0000 0000 0000 0001>
    get_size: **get** ec_file_shared_size

reply:
    xid: 02f0
    get-size: ec_file_shared_size -> 0x 0000 0000 0400 0200
    trusted.ec.dirty: 0x 0000 0000 0000 0001(data), 0x 0000 0000 0000 0001(meta), origin **is** zero
    trusted.ec.version:
        key_len: 0x 0013 是key的len
        key: 0x 7472 7573 7465 642e 6563 2e76 6572 7369 6f6e 0000 是 trusted.ec.version.., 最后一个.是对齐补零
        value_type: 0x 0000 0006 是 value type: GF_DATA_TYPE_PTR (0x06)
        value_size: 0x 0000 0010 是value size 16
        value: 0x 0000 0000 0000 0083 是data version, 0x 0000 0000 0000 0086是meta version

000c: GFS3_OP_READ(file read all stripe of fragment)

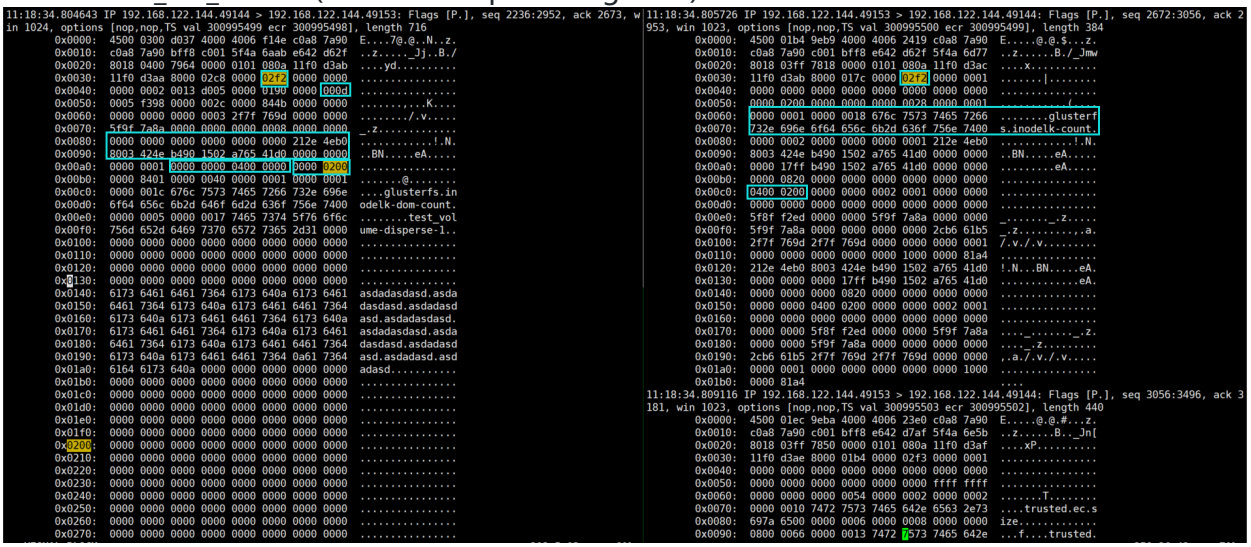


offset: 0000 0000 0400 0000
size: 0000 0200

bool_t

```
xdr_gfx_read_req (XDR *xdrs, gfx_read_req *objp)
{
    register int32_t *buf;

    int i;
     if (!xdr_opaque (xdrs, objp->gfid, 16))
         return FALSE;
     if (!xdr_quad_t (xdrs, &objp->fd))
         return FALSE;
     if (!xdr_u_quad_t (xdrs, &objp->offset))
         return FALSE;
     if (!xdr_u_int (xdrs, &objp->size))
         return FALSE;
     if (!xdr_u_int (xdrs, &objp->flag))
         return FALSE;
     if (!xdr_gfx_dict (xdrs, &objp->xdata))
         return FALSE;
    return TRUE;
}
```

000d: GFS3_OP_WRITE (write file stripe of fragment)



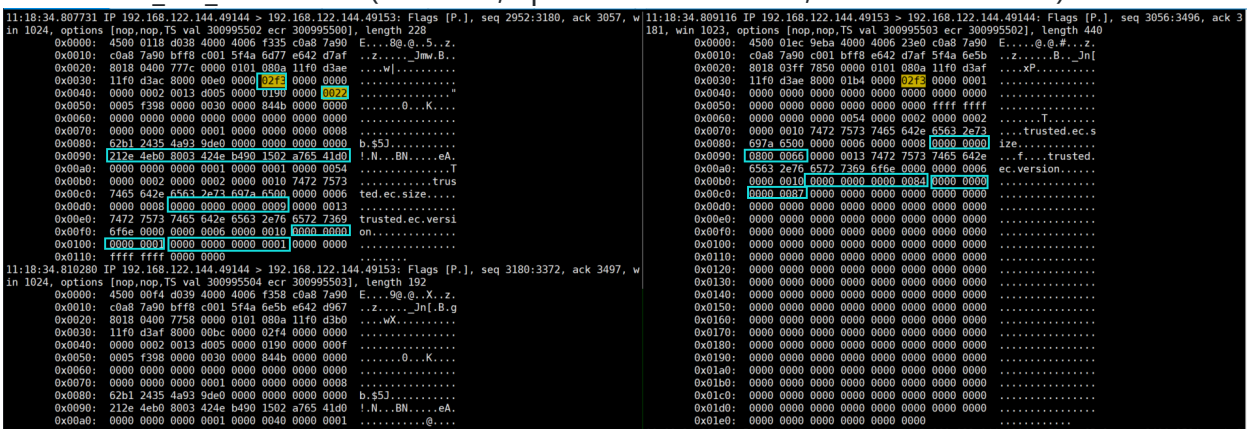

```
request:
    xid: 0x 02f2
    fop: 0x 000d
    gfid: 0x 212e 4eb0 8003 424e b490 1502 a765 41d0
    fd: 0x 0000 0000 0000 0001
    offset: 0x 0000 0000 0400 0000
    size: 0x 0000 0200

reply:
    xid: 0x 02f2
    key: glusterfs.inodelk-count, value:0000 0000 0000 0001
    pre-iattr and post-iattr

client:
bool_t
xdr_gfx_write_req (XDR *xdrs, gfx_write_req *objp)
{
    register int32_t *buf;
```

```
    int i;
     if (!xdr_opaque (xdrs, objp->gfid, 16))
         return FALSE;
     if (!xdr_quad_t (xdrs, &objp->fd))
         return FALSE;
     if (!xdr_u_quad_t (xdrs, &objp->offset))
         return FALSE;
     if (!xdr_u_int (xdrs, &objp->size))
         return FALSE;
     if (!xdr_u_int (xdrs, &objp->flag))
         return FALSE;
     if (!xdr_gfx_dict (xdrs, &objp->xdata))
         return FALSE;
    return TRUE;
}


server:
    rpc_receice_write_msg:

    Line 16252:      0 glfs_rpcrqhnd(21154): -> server4_0_writev, pid:20750, tid:21154
    Line 16373:  10394 glfs_rpcrqhnd(21154):    -> server4_writev_resume, pid:20750,
tid:21154
    Line 16503:  22107 glfs_rpcrqhnd(21154):        -> quota_writev, pid:20750, tid:21154
    Line 16563:  28162 glfs_rpcrqhnd(21154):            -> default_writev, pid:20750,
tid:21154
    Line 16624:  32859 glfs_rpcrqhnd(21154):                -> barrier_writev, pid:20750,
tid:21154
    Line 16685:  36923 glfs_rpcrqhnd(21154):                    -> marker_writev,
pid:20750, tid:21154
    Line 16750:  41573 glfs_rpcrqhnd(21154):                        -> default_writev,
pid:20750, tid:21154
    Line 16811:  44765 glfs_rpcrqhnd(21154):                            -> iot_writev,
pid:20750, tid:21154
    Line 16812:  44814 glfs_rpcrqhnd(21154):                                ->
fop_writev_stub, pid:20750, tid:21154
    Line 16817:  45062 glfs_rpcrqhnd(21154):                                    ->
args_writev_store, pid:20750, tid:21154

    iot dispatch write_msg to glfs_iotwr000:

    Line 16898:    250 glfs_iotwr000(20777): -> default_writev_resume, pid:20750,
tid:20777
    Line 16966:   3677 glfs_iotwr000(20777):     -> up_writev, pid:20750, tid:20777
    Line 17032:   7278 glfs_iotwr000(20777):         -> leases_writev, pid:20750,
tid:20777
    Line 17098:  11126 glfs_iotwr000(20777):             -> ro_writev, pid:20750,
tid:20777
    Line 17160:  14398 glfs_iotwr000(20777):                 -> worm_writev, pid:20750,
tid:20777
    Line 17221:  17599 glfs_iotwr000(20777):                     -> pl_writev, pid:20750,
tid:20777
    Line 17357:  24716 glfs_iotwr000(20777):                         -> default_writev,
pid:20750, tid:20777
    Line 17418:  28243 glfs_iotwr000(20777):                             ->
br_stub_writev, pid:20750, tid:20777
    Line 17483:  32439 glfs_iotwr000(20777):                                 ->
```

```
changelog_writev, pid:20750, tid:20777
    Line 17547:  35937 glfs_iotwr000(20777):                                        ->
default_writev, pid:20750, tid:20777
    Line 17607:  39169 glfs_iotwr000(20777):                                            ->
posix_writev, pid:20750, tid:20777
    Line 17656:  43308 glfs_iotwr000(20777):  (write data)                               ->
__posix_writev, pid:20750, tid:20777
    Line 17657:  43355 glfs_iotwr000(20777):
-> __posix_pwritev, pid:20750, tid:20777
    Line 17659:  43474 glfs_iotwr000(20777):
-> _fill_writev_xdata, pid:20750, tid:20777
    Line 17755:  50421 glfs_iotwr000(20777):
-> changelog_writev_cbk, pid:20750, tid:20777
    Line 17817:  54027 glfs_iotwr000(20777):
-> default_writev_cbk, pid:20750, tid:20777
    Line 17878:  57319 glfs_iotwr000(20777):  (PL_STACK_UNWIND->pl_set_xdata_response)
-> pl_writev_cbk, pid:20750, tid:20777
    Line 17976:  65929 glfs_iotwr000(20777):
-> leases_writev_cbk, pid:20750, tid:20777
    Line 18037:  69277 glfs_iotwr000(20777):
-> up_writev_cbk, pid:20750, tid:20777
    Line 18099:  72551 glfs_iotwr000(20777):
-> default_writev_cbk, pid:20750, tid:20777
    Line 18161:  75920 glfs_iotwr000(20777):
-> marker_writev_cbk, pid:20750, tid:20777
    Line 18296:  84871 glfs_iotwr000(20777):  (server4_post_common_2iatt set iattr)
-> server4_writev_cbk, pid:20750, tid:20777
    Line 18360:  88285 glfs_iotwr000(20777):
-> sys_writev, pid:20750, tid:20777

    pl_set_xdata_response: pl_inodelk_xattr_fill (set glusterfs.inodelk-count)
    server4_post_common_2iatt: set pre-attr and post-attr

    static inline void
gfx_stat_to_iattx(struct gfx_iattx *gf_stat, struct iatt *iatt)
{
    if (!iatt || !gf_stat)
        return;

    memcpy(iatt->ia_gfid, gf_stat->ia_gfid, 16);

    iatt->ia_flags = gf_stat->ia_flags;
    iatt->ia_ino = gf_stat->ia_ino;
    iatt->ia_dev = gf_stat->ia_dev;
    iatt->ia_rdev = gf_stat->ia_rdev;
    iatt->ia_size = gf_stat->ia_size;
    iatt->ia_nlink = gf_stat->ia_nlink;
    iatt->ia_uid = gf_stat->ia_uid;
    iatt->ia_gid = gf_stat->ia_gid;
    iatt->ia_blksize = gf_stat->ia_blksize;
    iatt->ia_blocks = gf_stat->ia_blocks;
    iatt->ia_atime = gf_stat->ia_atime;
    iatt->ia_atime_nsec = gf_stat->ia_atime_nsec;
    iatt->ia_mtime = gf_stat->ia_mtime;
    iatt->ia_mtime_nsec = gf_stat->ia_mtime_nsec;
    iatt->ia_ctime = gf_stat->ia_ctime;
    iatt->ia_ctime_nsec = gf_stat->ia_ctime_nsec;
    iatt->ia_btime = gf_stat->ia_btime;
```

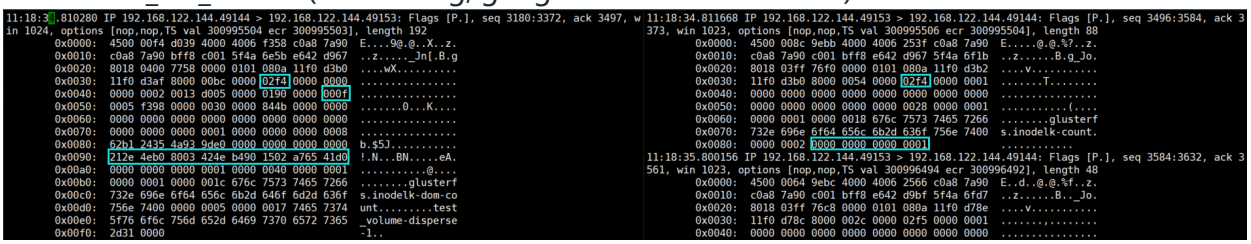
```
    iatt->ia_btime_nsec = gf_stat->ia_btime_nsec;
    iatt->ia_attributes = gf_stat->ia_attributes;
    iatt->ia_attributes_mask = gf_stat->ia_attributes_mask;

    iatt->ia_type = ia_type_from_st_mode(gf_stat->mode);
    iatt->ia_prot = ia_prot_from_st_mode(gf_stat->mode);
}
```

0022: GFS3_OP_FXATTROP (size xattr, update trusted.ec.size, trusted.ec.version)

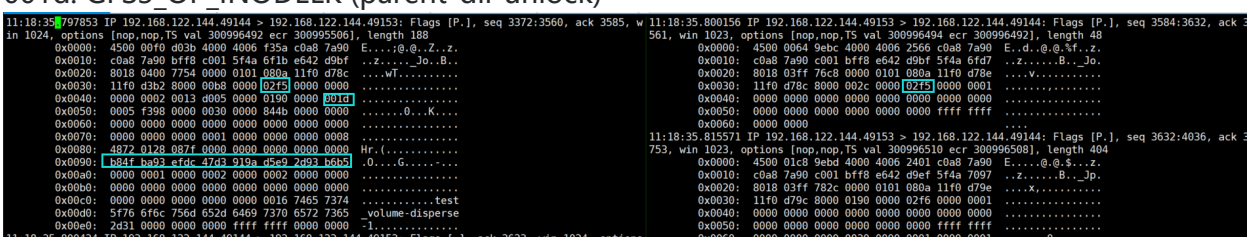



```
request:
    xid; 0x 02f3
    fop: 0x 0022
    trusted.ec.size: add 0x 0000 0000 0000 0009
    trusted.ec.version: data version add 0x 0000 0000 0000 0001, meta version add 0x 0000
0000 0000 0001

reply:
    xid: 0x 02f3
    trusted.ec.size: 0x 0000 0000 0800 0066
    trusted.ec.version: data version 0x 0000 0000 0000 0084, meta version: 0x 0000 0000
0000 0087
```

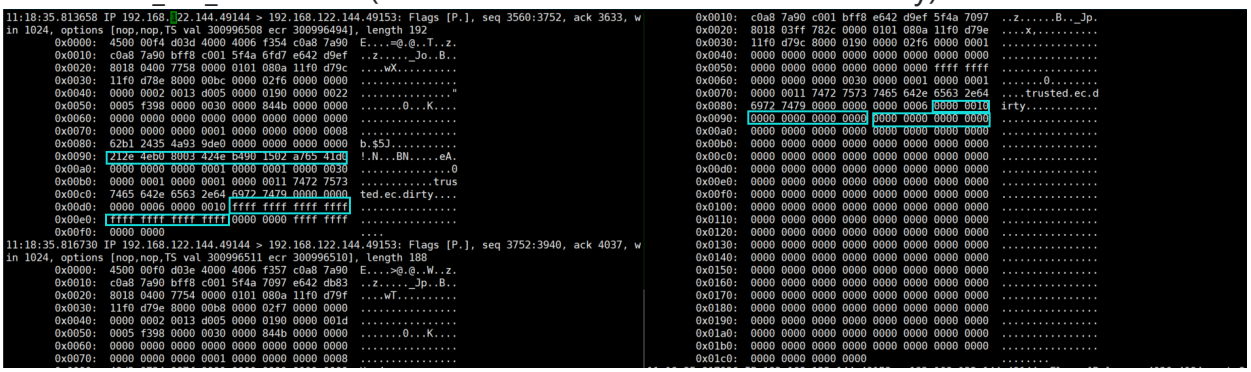
000f: GFS3_OP_FLUSH (do nothing, get glusterfs.inodelk-count)



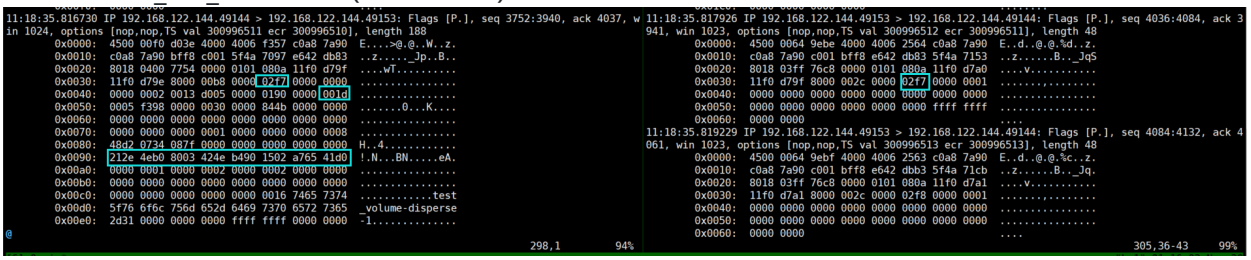
001d: GFS3_OP_INODELK (parent-dir unlock)



0022: GFS3_OP_FXATTROP (file set xattr: set<add -1> trusted.ec.dirty)



001d: GFS3_OP_INODELK (file unlock)



0029: GFS3_OP_RELEASE (file release)

修复逻辑

修复逻辑

未完的工作

- clinet端：详细了解dht和rebalance
- server端：详细了解index模块，changelog features(默认关闭)，bitrot features(默认关闭)