

前言

为什么写这本书？

对于这个问题，萌生这种想法是从 2020 年下半年，因为转变方向开始接触 `glusterfs` 开始的。对于 `glusterfs`，一开始在网上搜了很多资料，但是大部分资料的内容都是残缺不全的，并且没有一本专业系统的技术书来专门讲述 `glusterfs`。而 `glusterfs` 作为世界上与 `ceph`、`hdfs` 齐名的分布式存储系统，但是资料却完全无法和这两者相比，这是不应该的。因此在 2021 年五月下旬的时候，和朋友一起聊天时，激起了想专门写一本 `glusterfs` 书的想法，抱着尝试的心态，通过阅读源码资料，还有国内外的一些信息，慢慢不断地整合学习，才有了这本书的诞生。当然这本书作为本人工作生涯的第一本书，写下这本书的时候，恰好也是工作第三年了，对于程序员的职业生涯来说，这算是迈向了一个新的阶段了，给自己一个新的礼物吧。

读者对象

这本书因为是希望对 `glusterfs` 有一个相对全面系统的认识，因此在部分章节会有大量的源码阅读内容，而这本书内容，对于理解每一个系统工具都是必经之路，不同基础的读者阅读时可能会感受不一样，如果在阅读过程中，遇到部分细节没有弄清楚或者一时半会不知道的时候，建议先宏观了解每个章节的宏观思想，然后后面有恰当的机会再回头重新阅读，那样或许会有新的感受和发现。

另外对于本书的读者对象，本书适合以下人群：

1. 想要深入了解 `glusterfs` 的，但是之前只是大概理解一些运维操作，对原理不熟悉的。
2. 有一定 `glusterfs` 运维基础经验，并且对 `Linux` 系统有使用经验，对 `VFS` 这些文件系统概念有简单认识的人。
3. 对分布式系统感兴趣的互联网工程师和架构师。

如何阅读这本书

本书主要分为五大部分：

简单使用篇(第一章)：这一部分的内容，主要是讲解在部署好了 `glusterfs` 系统之后，如何简单地创建并且使用不同类型的 `volume`，还有讲解了一下仲裁节点的特点，和集群维护中的日志文件位置等。

基础概念篇(第二章)：这一部分的内容主要是讲解一下 `Linux` 文件系统的一些基础概念基础，对 `VFS` 的介绍等，为后面的章节部分内容做好一些基础概念的准备。

原理篇(第三章)：这一部分的内容，主要就是讲解了 `glusterfs` 的核心理念，包括 `gfid` 和 `posix` 接口，线程模型等，这部分内容主要以源码分析为主，阅读难度较大，对于具体细节的内容如暂时无法理解的，建议先宏观了解每个章节的思想和内容，后续再考虑深入每个章节的部分内容。

性能特性篇(第四和五章)：这一部分的内容，主要讲解 `glusterfs` 的特性和性能参数相关的，包括扩缩容和容量限制等。

运维篇(第六章)：这一章主要分享了在日常运维使用过程中遇到的生产环境版本的 `bug` 问题，还有目前和未来 `glusterfs` 的一些优化内容与项目发展计划等。

勘误和支持

由于本人水平有限，写这本书时是本人工作第三年同时也是第一次写书，因此如有错误和不当之处，敬请指出，或者阅读时有疑惑不解时，可以联系本人联系，github 地址为：github.com/httaotao,谢谢。



1. 第一章让集群先跑起来

1.1. 我们为什么要用 glusterfs

在这里我想简单介绍一下，什么是 glusterfs,在官方的介绍里面,这里提到 GlusterFS 是可扩展的网络文件系统，适用于数据密集型任务，例如云存储和媒体流。以上这段话是从 glusterfs 官方网站^[1] 上面获取的信息。而目前对于 glusterfs 而言，个人觉得最大的特点是无中心架构的分布式文件系统.其中和我们熟悉的 master/slave 架构的系统有着非常不同的地方。而我们这里将要介绍的，也是基于这个特点下，去逐渐了解 glusterfs 的一些特性与使用方式。

在了解一下新的工具，尤其是分布式系统的时候，总会有一些相似的问题，为什么我们要使用这个工具？glusterfs 能够带给我们什么？如果你还没有真正投入生产环境使用，在做调研测试阶段的话，那么我们又该如何去熟悉一些新的文件系统呢？关于这个问题，我希望在最后的时刻才回答，因为当我们真正去深入了一些不同的系统模块之后，未来在面对新的系统架构工具，我们才会有一些明确地目标和方式去进行对比。同时基于这些内容，我们才会有更多的思考与抉择。

相反，如果在内容的最开始就讲解这些问题，那么可能大家会比较迷茫与无法理解，或者说没有一个明确的印象。当然如果你已经在生产环境中有了比较多的 glusterfs 的使用经验，那么也可以直接去阅读感兴趣的内容，而不需从部署与一些特性开始关注。

这里先约定一下,后面所有的讲解内容,如果没有特别指明,那么默认就是本人写该资料时官方最新的版本,也就是 9.2。同时关于实验环境,这里建议至少创建四台虚拟机进行测试使用,三台虚拟机是一个小集群,然后一台客户端机器。在部分场景下,可能需要更多的虚拟机,操作系统版本等则没有严格要求,而本人的测试环境测试 centos7 的系统,使用 ubuntu 或者其他 linux 系统皆可。

另外这里为了保持一下风格,或者说不显得那么怪异,有一些专业名词将保持使用英文名词,这样可能在未来大家接触官方的英文文档,或者在 github 上面提 issue 的时候,并不会感到比较陌生和困惑,同时可能未来也会有更多关于 glusterfs 的书籍,那么阅读起来的话,会更有统一性和流畅性。

1.2. 先让 glusterfs 跑起来

1.2.1. 简单概念介绍

在讲解 glusterfs 的 volume 和 brick 之前，这里为了方便大家理解，需要先简单介绍一些概念，主要是关于 glusterfs 中的常见的。当然，有一些概念可能会放到后面讲到的时候再介绍，避免一下子讲解太多陌生概念导致的困惑与迷茫。

这里 glusterfs 的 brick 可以简单理解为数据存储在节点上面的目录及其管理的进程，而卷，也就是 volume 就是一份完整的数据存储在不同的 brick 上面的集合，这里会有关于设置这些存储时候的一些性能参数或者其他指标等。如果这段话不太理解也没关系，相信在后面创建 volume 之后，大家会有较为直观的感受与理解。

glusterfs 目前官方支持五种不同类型的卷。其中生产环境里面用的比较多的是分布式复制卷和分布式冗余卷。下面将会简单介绍一下不同类型的卷的使用特点与方式。当然在讲解 volume 之前，希望分享一下关于 glusterfs 的一个特点，通信的全互联架构，因为 glusterfs 是没有主节点，也就是通常大家常见的 master 节点那样的概念，因此每一个客户端都会和不同的节点进行通信，具体的可以看看下图所示。

1.2.2. 一些有趣的 volume

1.2.2.1. 复制卷

官方支持复制卷和分布式复制卷，而生产环境中常用的是分布式复制卷，首先这里引用一些官方的图片，来理解一下什么是分布式复制卷。

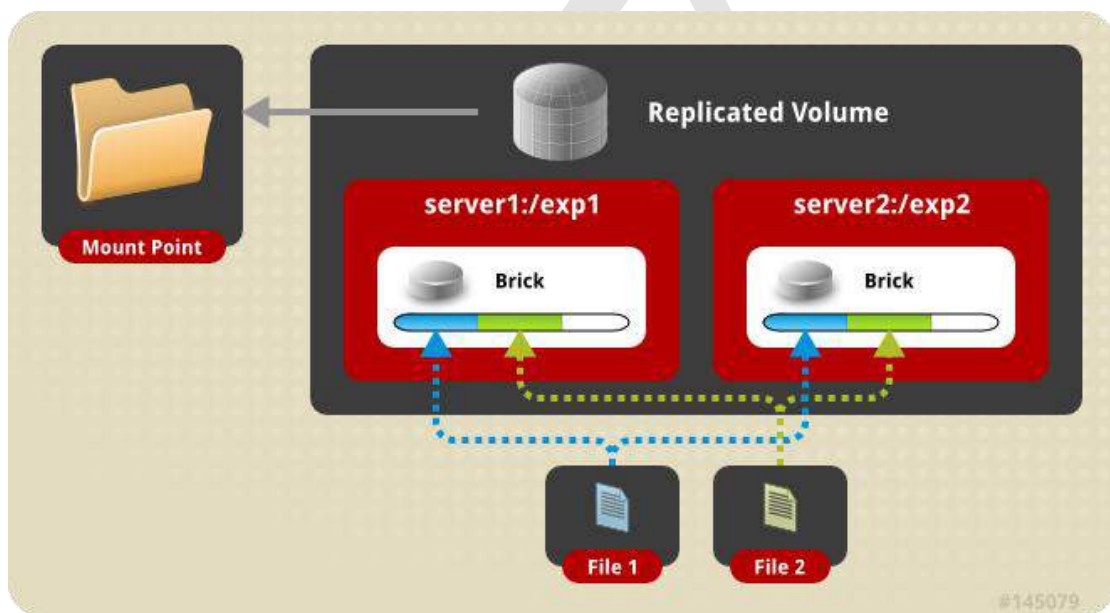


图 1.1

这里图片给出的是一个 2 个副本的复制卷，生产环境中通常使用 3 个副本的复制卷，创建的命令如下所示。

1. [root@gfs01 ~]# gluster volume create test-replica replica 3 192.168.0.{110,111,112}:/glusterfs/test-replica
2. volume create: test-replica: failed: The brick 192.168.0.110:/glusterfs/test-replica is being created in the root partition. It is recommended that you don't use the system's root partition for storage backend. Or use 'force' at the end of the command if you want to override this behavior.
3. [root@gfs01 ~]# gluster volume create test-replica replica 3 192.168.0.{110,111,112}:/glusterfs/test-replica force

4. volume create: test-replica: success: please start the volume to access data
5. [root@gfs01 ~]# gluster volume start test-replica
6. volume start: test-replica: success

这里因为 glusterfs 官方推荐不使用 root 进行操作的, 或者使用 force 命令强制创建, 那么这里作为测试使用可以不用考虑, 但生产环境安装部署的时候需要注意该问题。这里使用命令创建了一个名为 test-replica 的三副本的分布式复制卷, 那么这里创建之后并且启动了。那么该如何知道该卷的状态是否正常呢? 可以查看状态和挂载使用。

1. # gluster volume status test-replica
2. Status of volume: test-replica
3. Gluster process TCP Port RDMA Port Online Pid
4. -----
5. Brick 192.168.0.110:/glusterfs/test-replica 49153 0 Y 16485
6. Brick 192.168.0.111:/glusterfs/test-replica 49152 0 Y 16368
7. Brick 192.168.0.112:/glusterfs/test-replica 49153 0 Y 16422
8. Self-heal Daemon on localhost N/A N/A Y 16502
9. Self-heal Daemon on gfs03 N/A N/A Y 16439
10. Self-heal Daemon on gfs02 N/A N/A Y 16385
- 11.
12. Task Status of Volume test-replica
13. -----
14. There are no active volume tasks

这里重点需要关注两个地方, 分别是 online 和 pid, 这里如果每个节点上关于该 volume 的存储目录管理进程异常, 也就是 brick 异常的话, 那么这里就无法获取到对应的端口号的。那么这里还可以使用命令查看一下每个节点的 brick 进程, 结果如下所示。


```

1. # ps -ef |grep 16485
2. root 16485 1 0 12:27 ? 00:00:00 /usr/sbin/glusterfsd -s 192.
168.0.110 --volfile-id test-replica.192.168.0.110.glusterfs-test-replica -
p /var/run/gluster/vols/test-replica/192.168.0.110-glusterfs-test-replica.
pid -S /var/run/gluster/d403ef3161bd809c.socket --brick-name /glusterf
s/test-replica -l /var/log/glusterfs/bricks/glusterfs-test-replica.log --xlato
r-option *-posix.glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b
54 --process-name brick --brick-port 49153 --xlator-option test-replica-s
erver.listen-port=49153

```

这一段信息中,指定的日志路径默认都是在/var/log/glusterfs 下的,而 brick 的日志则在/var/log/glusterfs/bricks 中,同时如果生产环境中遇到某个 brick 的进程异常,遇到需要手动启动 brick 进程的时候,可以使用上面的输出结果来手动执行,当然一般也很少会遇到这样的情况。

这里想先强调一下,对于复制卷,不建议使用 2 副本的复制,因为这样会在 brick 异常之后,比较容易出现脑裂的现象,关于这部分的内容,会在后面详细讲解一下。

那么这里该使用呢?可以在其他节点上,只要安装了 glusterfs client 的话,那直接使用 mount 挂载即可。

```

1. [root@gfs03 ~]# gluster volume info test-replica
2.
3. Volume Name: test-replica
4. Type: Replicate
5. Volume ID: 66fed900-29f5-40f5-8add-7e4e365ab29a
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 192.168.0.110:/glusterfs/test-replica

```

```
12. Brick2: 192.168.0.111:/glusterfs/test-replica
13. Brick3: 192.168.0.112:/glusterfs/test-replica
14. Options Reconfigured:
15. performance.client-io-threads: off
16. nfs.disable: on
17. transport.address-family: inet
18. storage.fips-mode-rchecksum: on
19. cluster.granular-entry-heal: on
20.
21.
22. [root@gfs03 ~]# mount -t glusterfs 192.168.0.110:test-replica /mnt/test-replica
23. [root@gfs03 ~]# ls /mnt/test-replica/
24. [root@gfs03 ~]# touch /mnt/test-replica/a.txt
25. [root@gfs03 ~]# date >> /mnt/test-replica/a.txt
26. ...
27. [root@gfs03 ~]# md5sum /mnt/test-replica/a.txt
28. 9eb6c6683f293a1f62d38a4ed94b17c8 /mnt/test-replica/a.txt
29. [root@gfs03 ~]# md5sum /glusterfs/test-replica/a.txt
30. 9eb6c6683f293a1f62d38a4ed94b17c8 /glusterfs/test-replica/a.txt
31.
32.
33. [root@gfs02 ~]# md5sum /glusterfs/test-replica/a.txt
34. 9eb6c6683f293a1f62d38a4ed94b17c8 /glusterfs/test-replica/a.txt
35.
36. [root@gfs01 ~]# md5sum /glusterfs/test-replica/a.txt
37. 9eb6c6683f293a1f62d38a4ed94b17c8 /glusterfs/test-replica/a.txt
```

这里选择的 ip 节点可以是集群中的任意一个节点, 哪怕 brick 并不在该节点上面, 而这也正是 glusterfs 无中心架构的一个特点. 因为通信方式采用的是全互联的模式, 因此元数据信息保持一致的情况下, 那么这里是可以选择任意节点进行挂载的。另外这里可以看到, 复制卷的特点就是数据都是完整的一份的, 然而冗余卷则是不一样的, 可以进行对比一下。

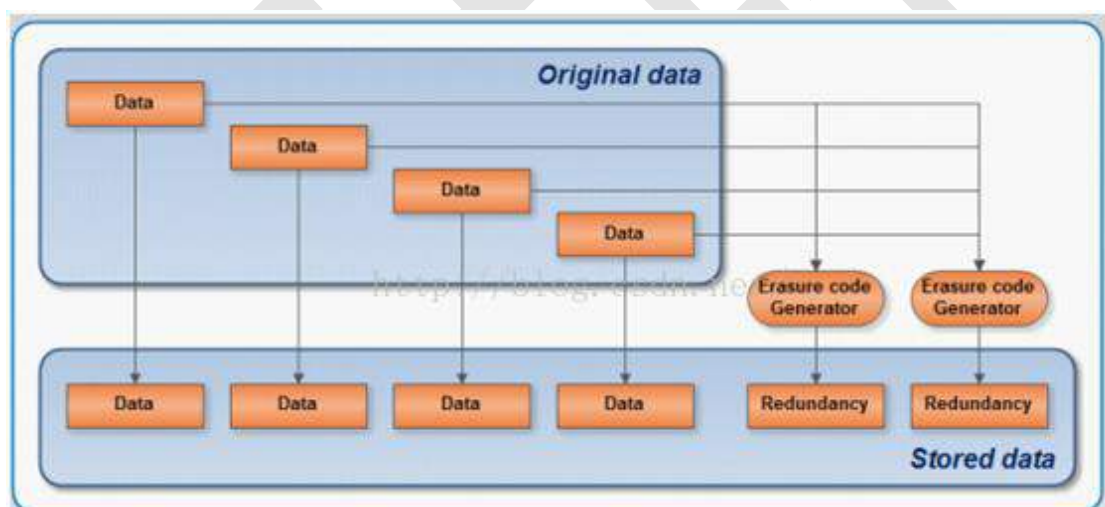
当然在生产环境中, 也不建议把挂载的 ip 都集中在其中的一个节点上, 否则所有的请求流量都会经过该节点, 会加重节点的负载。而挂载以后, 这里的使

用就和本地目录类似了，也和 NFS 的使用类似。最后如果想解除挂载，那么直接使用 `umount` 目录即可。

```
1. [root@gfs03 ~]# umount /mnt/test-replica/
```

1.2.2.2. 冗余卷

什么是冗余卷，这里与复制卷不一样的地方就是，数据到底以怎样的形式存放在节点上，像常见的 hdfs 是 3 副本机制的。而随着数据量越来越多，如果未来对于所有的数据都是 3 副本，那么占用的空间就会比较多了，因此为了节约空间，就有了冗余码的出现，通过下面的图来感受一下。



从图中可以很直观地感受到，所谓的冗余码，就是把一个完整的数据切割成多份，然后每一份计算得到一部分冗余码，并且存储在节点中，这里的存储空间会比完成的一份数据多一些，但是会远远比 3 副本所占用的空间小。同时因为有了冗余码，因此即使在损坏一定的比例数据下，数据的完整性都能得到保证。

那么这里的数据是如何存储地，冗余码如何计算得到的？通过计算得到这些冗余码等问题，这些将放在后面详细讲解，因为这部分内容涉及到比较多的数学内容，如果感兴趣，也可以先自行简单了解一下里德-所罗门码(Reed-solomon codes)。同时目前工业界也比较关注，未来到底是复制卷还是冗余卷，该如何使用，优缺点如何，都是非常值得思考与关注的问题。

这里我们先了解一下如何创建分布式冗余卷，并且这里我们挂载创建一个文件，写入一些数据，查看一下到底分布式冗余卷的数据是如何分布的。

```
1. [root@gfs01 ~]# gluster volume create test-disperse disperse 3 192.168.0.{110,111,112}:/glusterfs/test-disperse force
2. volume create: test-disperse: success: please start the volume to access data
3.
4. [root@gfs01 ~]# gluster volume start test-disperse
5. volume start: test-disperse: success
6.
7. [root@gfs01 ~]# gluster volume info test-disperse
8.
9. Volume Name: test-disperse
10. Type: Disperse
11. Volume ID: 0390d729-b6d8-4edd-bb72-bd28c3ec7472
12. Status: Started
13. Snapshot Count: 0
14. Number of Bricks: 1 x (2 + 1) = 3
15. Transport-type: tcp
16. Bricks:
17. Brick1: 192.168.0.110:/glusterfs/test-disperse
18. Brick2: 192.168.0.111:/glusterfs/test-disperse
19. Brick3: 192.168.0.112:/glusterfs/test-disperse
20. Options Reconfigured:
21. storage.fips-mode-rchecksum: on
22. transport.address-family: inet
23. nfs.disable: on
24.
25.
26. [root@gfs01 ~]# mount -t glusterfs 192.168.0.112:test-disperse /mnt/test-disperse
27. [root@gfs01 ~]# touch /mnt/test-disperse/1.txt
28. [root@gfs01 ~]# date >> /mnt/test-disperse/1.txt
```

```

29. [root@gfs01 ~]# date >> /mnt/test-disperse/1.txt
30. ...
31. [root@gfs01 ~]# cat /mnt/test-disperse/1.txt
32. Wed May 26 11:42:49 EDT 2021
33. Wed May 26 11:42:49 EDT 2021
34. Wed May 26 11:42:50 EDT 2021
35. Wed May 26 11:42:51 EDT 2021
36. Wed May 26 11:42:51 EDT 2021
37.
38. [root@gfs01 ~]# md5sum /mnt/test-disperse/1.txt
39. 95af6ad88c70c127faee8d84e3eb8d8f /mnt/test-disperse/1.txt
40. [root@gfs01 ~]# md5sum /glusterfs/test-disperse/1.txt
41. 0215b19719d86fd12a973926ec0c0854 /glusterfs/test-disperse/1.txt
42.
43.
44. [root@gfs02 ~]# md5sum /glusterfs/test-disperse/1.txt
45. cdcf65a53fef2a8066bd96d6324dab85 /glusterfs/test-disperse/1.txt
46.
47. [root@gfs03 ~]# md5sum /glusterfs/test-disperse/1.txt
48. d76f9fc89d264d19641ec552f600d46a /glusterfs/test-disperse/1.txt

```

我们查看一下每一个 brick 下面的数据情况可以发现文件的 md5 都不一样的，那么这里并不是和完整的挂载目录文件，同时如果感兴趣，这里可以查看一下几个 brick 下面的数据分布情况。

1.2.2.3. force 的作用

前面提到了两种常见的 volume,那么对于 volume 的操作有 status,start 和 info 这些常见的，而其中对于 start 和 stop 都有一个可选项 force，这个的作用其实是什么呢？这里我们可以简单测试一下。

```

1. [root@gfs01 ~]# gluster volume start
2.
3. Usage:
4. volume start <VOLNAME> [force]

```

```

5.
6. [root@gfs01 ~]# gluster volume stop
7.
8. Usage:
9. volume stop <VOLNAME> [force]

```

下面进行简单的测试，方法就是 kill 掉其中一个 brick 的目录，然后重新启动查看一下作用。

```

1. Status of volume: test-replica
2. Gluster process          TCP Port RDMA Port Online Pid
3. -----
4. Brick 192.168.0.110:/glusterfs/test-replica 49153    0        Y    177
5. 6
6. Brick 192.168.0.111:/glusterfs/test-replica 49153    0        Y    162
7. 0
8. Brick 192.168.0.112:/glusterfs/test-replica 49155    0        Y    152
9. 1
10. Self-heal Daemon on localhost      N/A      N/A      Y    1160
11. Self-heal Daemon on gfs02          N/A      N/A      Y    1426
12. Self-heal Daemon on gfs03          N/A      N/A      N    N/A
13.
14. Task Status of Volume test-replica
15. -----
16. There are no active volume tasks
17.
18. //这里删掉了其中一个 brick 目录
19. [root@gfs01 ~]# rm -fr /glusterfs/test-replica
20. [root@gfs01 ~]# gluster volume start test-replica
21. volume start: test-replica: failed: Failed to find brick directory /gluster
22. fs/test-replica for volume test-replica. Reason : No such file or director
23. y
24.
25. //前面无法正常启动了.使用 force 强制启动.
26. [root@gfs01 ~]# gluster volume start test-replica force
27. volume start: test-replica: success
28. [root@gfs01 ~]# gluster volume status test-replica
29. Status of volume: test-replica
30. Gluster process          TCP Port RDMA Port Online Pid

```

26.	-----				
27.	Brick 192.168.0.110:/glusterfs/test-replica	N/A	N/A	N	N/A
28.	Brick 192.168.0.111:/glusterfs/test-replica	49154	0	Y	1674
29.	Brick 192.168.0.112:/glusterfs/test-replica	49154	0	Y	1573
30.	Self-heal Daemon on localhost	N/A	N/A	Y	1160
31.	Self-heal Daemon on gfs03	N/A	N/A	N	N/A
32.	Self-heal Daemon on gfs02	N/A	N/A	Y	1426
33.					
34.	Task Status of Volume test-replica				
35.	-----				
36.	There are no active volume tasks				

从这里可以看到，其中所谓的 **force** 就是在一些 **brick** 坏掉异常的时候，能够强制启动，对于 3 副本的分布式复制卷来说，正常是只要超过 2 个 **brick** 正常就可以启动的，但是默认是不行，需要使用 **force** 命令。另外这里如果感兴趣，读者可以自行测试一下，前面删掉的数据目录，如果再次重新创建，能否继续生效呢？另外这里 **force** 的作用可以从代码中找到答案。

```

1. // xlator->mgmt->glusterd->src->glusterd-volume-ops.c
2. //函数名: glusterd_op_stage_start_volume
3.
4. ret = gf_lstat_dir(brickinfo->path, NULL);
5.     if (ret && (flags & GF_CLI_FLAG_OP_FORCE)) {
6.         continue;
7.     } else if (ret) {
8.         len = snprintf(msg, sizeof(msg),
9.             "Failed to find "
10.            "brick directory %s for volume %s. "
11.            "Reason : %s",
12.            brickinfo->path, volname, strerror(errno));
13.         if (len < 0) {
14.             strcpy(msg, "<error>");
15.         }
16.         goto out;
17.     }

```

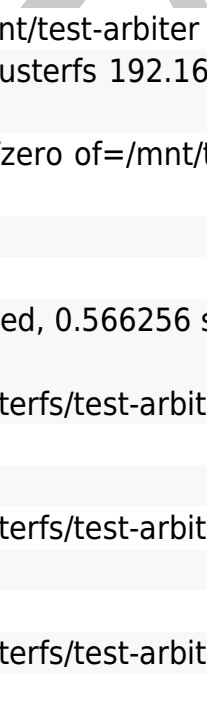
1.2.3. 仲裁节点 arbiter

1.2.3.1. arbiter 到底是什么

在分布式复制卷这里,还有一种特殊的类型,就是仲裁节点,首先向说明一下,为什么需要考虑仲裁节点 arbiter 呢? 最大的原因就是减少脑裂的发生.(所谓的脑裂现象其实副本中的元数据信息在机器产生故障或者反复宕机后等复杂环境下,元数据信息不一致产生的,在分布式系统中属于非常致命且危险的现象)。为了进一步感受一下仲裁节点的作用,下面先简单创建一个进行测试。

1. [root@gfs03 ~]# gluster volume create test-arbiter replica 2 arbiter 1 192.168.0.{110,111,112}:/glusterfs/test-arbiter force
2. volume create: test-arbiter: success: please start the volume to access data
3. [root@gfs03 ~]# gluster volume start test-arbiter
4. volume start: test-arbiter: success
5. [root@gfs03 ~]# gluster volume info test-arbiter
- 6.
7. Volume Name: test-arbiter
8. Type: Replicate
9. Volume ID: c0f1d50a-0060-456f-a82e-64fb8b99c020
10. Status: Started
11. Snapshot Count: 0
12. Number of Bricks: 1 x (2 + 1) = 3
13. Transport-type: tcp
14. Bricks:
15. Brick1: 192.168.0.110:/glusterfs/test-arbiter
16. Brick2: 192.168.0.111:/glusterfs/test-arbiter
17. Brick3: 192.168.0.112:/glusterfs/test-arbiter (arbiter)
18. Options Reconfigured:
19. cluster.granular-entry-heal: on
20. storage.fips-mode-rchecksum: on
21. transport.address-family: inet
22. nfs.disable: on
23. performance.client-io-threads: off

这里注意一点, 为了保留一下原来的写法, 可能会见到创建文件时写 replica 3 arbiter 1 也是一样的, 对于 3 副本的仲裁节点来说, 默认都是最后一个 brick 作为仲裁节点 arbiter, 而如果是 2*3 的 volume, 则是每一组的最后一个, 也是第三个 brick 作为 arbiter。同时, 作为仲裁节点, 那么该 brick 的最大的作用就是对数据元信息的保存, 并且不保存实际的数据。



```
1. [root@gfs03 ~]# mkdir -p /mnt/test-arbiter
2. [root@gfs03 ~]# mount -t glusterfs 192.168.0.110:test-arbiter /mnt/test-arbiter
3. [root@gfs03 ~]# dd if=/dev/zero of=/mnt/test-arbiter/1.txt bs=64k count=1000
4. 1000+0 records in
5. 1000+0 records out
6. 65536000 bytes (66 MB) copied, 0.566256 s, 116 MB/s
7.
8. [root@gfs03 ~]# du -sh /glusterfs/test-arbiter/
9. 16K    /glusterfs/test-arbiter/
10.
11. [root@gfs02 ~]# du -sh /glusterfs/test-arbiter/
12. 63M    /glusterfs/test-arbiter/
13.
14. [root@gfs01 ~]# du -sh /glusterfs/test-arbiter/
15. 63M    /glusterfs/test-arbiter/
```

对于仲裁节点来说, 因为不直接存放数据, 因此这里的磁盘使用量会比其他数据节点少很多, 那么关于这个 brick 的目录容量大小估计, 这里最好还是根据官方的建议, 是复制副本中文件数的 4KB 倍. 当然这里的估计值还取决于底层文件系统为给定磁盘大小分配的 inode 空间。

对于大小为 1TB 到 50TB 的卷，XFS 中的 maxpct 值仅为 5%。如果你想存储 3 亿个文件，4kx300m 给我们 1.2TB。其中 5% 的容量在 60GB 左右。假设建议的 inode 大小为 512 字节，那么我们只能存储 $60\text{GB}/512 \approx 1.2$ 亿个文件。因此，在格式化大于 1TB 的 XFS 磁盘时，最好选择更高的 maxpct 值（例如 25%）。这里 maxpct 是 xfs 中关于设置 inode 的一个参数，感兴趣的可以自行查阅。

那么关于 arbiter 的性能问题,还做了一个简单的测试,关于 arbiter 和非 arbiter 的 volume 性能.当然这里的测试只是为了测试出相对大小的差异.如果想真正测试出极限情况，建议在测试集群中进一步测试压测。下面先列出两个用于测试的 volume。

1. Volume Name: test-replica
2. Type: Replicate
3. Volume ID: 4568c063-5b75-4304-98f6-21f3955cc138
4. Status: Started
5. Snapshot Count: 0
6. Number of Bricks: $1 \times 3 = 3$
7. Transport-type: tcp
8. Bricks:
9. Brick1: 192.168.0.110:/glusterfs/test-replica
10. Brick2: 192.168.0.111:/glusterfs/test-replica
11. Brick3: 192.168.0.112:/glusterfs/test-replica
12. Options Reconfigured:
13. performance.client-io-threads: off
14. nfs.disable: on
15. transport.address-family: inet
16. storage.fips-mode-rchecksum: on
17. cluster.granular-entry-heal: on
- 18.
- 19.
20. Volume Name: test-arbiter

21. Type: Replicate
22. Volume ID: c0f1d50a-0060-456f-a82e-64fb8b99c020
23. Status: Started
24. Snapshot Count: 0
25. Number of Bricks: $1 \times (2 + 1) = 3$
26. Transport-type: tcp
27. Bricks:
28. Brick1: 192.168.0.110:/glusterfs/test-arbiter
29. Brick2: 192.168.0.111:/glusterfs/test-arbiter
30. Brick3: 192.168.0.112:/glusterfs/test-arbiter (arbiter)
31. Options Reconfigured:
32. performance.client-io-threads: off
33. nfs.disable: on
34. transport.address-family: inet
35. storage.fips-mode-rchecksum: on
36. cluster.granular-entry-heal: on

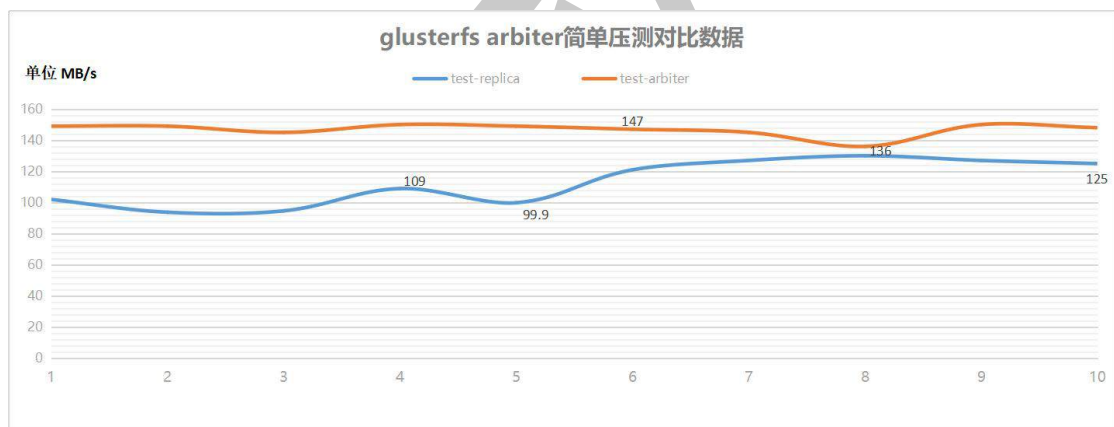
下面是测试的脚本,这里的测试是在一台客户端的虚拟机中挂载两个 volume,然后执行脚本进行测试的.测试的命令则是使用了常见的 dd 命令.每次创建一个 655MB 的文件,然后查看创建的速度。

```

1.  #!/bin/bash
2.
3.  #path=$1
4.
5.
6.  test(){
7.      sudo echo -e "\n path is $1"
8.      for((i=1;i<=10;i++));
9.      do
10.         sudo dd if=/dev/zero of=$1/$i.txt bs=64k count=10000
11.         sudo echo -e "\n"
12.         sudo sleep 2
13.     done
14. }
15.
16. test /mnt/test-replica
17. test /mnt/test-arbiter

```

最后给出测试结果，这里的测试结果表明,arbiter 的 volume 在较大文件的写入时，速度会比常用的 3 副本还要快一些，当然这是因为 arbiter 的 brick 并不承担保存源数据。因此这里的测试可以作为一个业务场景的参考。



这里的测试结果是在 centos7.9 虚拟机上面测试的,而在真实的生产环境中，除了测试这种较大的文件以外，还应该测试一些小文件的读写情况，可以使用以下脚本进行测试。该 脚本就是随机生成一万个小文件进行写入测试。

```
1.  #!/bin/bash
2.
3.  test(){
4.      sudo echo -e "\n\n path is: $1"
5.      #随机数
6.      sudo date
```

```

7.   for i in $(seq 1 10000)
8.   do
9.       num=`sudo echo [$RANDOM%100+1]`
10.      sudo dd if=/dev/zero of=$1/$i.txt bs=6k count=$num > /dev/nu
      || 2>&1
11.      done
12.      sudo date
13.  }
14.
15.  test /mnt/test-arbiter/
16.  test /mnt/test-replica/

```

最后关于 **arbiter**,会有客户端和服务端两个区别,同时每一种还有对应的参数进行选择,感兴趣的建议在测试集群中进行相应的测试。

1.2.3.2. **arbiter** 的客户端与服务端区别

关于仲裁节点,官方提供了两种不同的模式,一种是客户端,另外一种和服务端,对于两者的区别,下面先简单介绍一下。

仲裁节点类型	服务端	客户端
适用 volume	所有卷	复制卷和分布式复制卷
重要参数	server-quorum-type cluster.server-quorum-ratio	cluster.quorum-type cluster.quorum-count
特点	1. 在 2 副本的 2 个节点的集群上,设置 ratio 超过 51%是不会生效的,没法防止脑裂。 2. 如果不满足多数 bricks 活跃,那么服务端会 kill 掉 bricks 进程,让 volume 无法读取。对防止脑裂的效果会比客户端模式更有效。	1. 当 quorum-type 为 fixed 时,只有当活跃的 bricks 数量超过 quorum-count 时才允许写入。 2. 当 quorum-type 为 auto 时,只有超过半数活跃的 bricks 才能写入。 3. 当活跃的 bricks 恰好为一半时,第一个 bricks 必须活跃。

1.2.4. 集群的相关日志文件

这里主要想简单讲解一下一些常见的默认的集群日志文件目录存放地方,这样对于以后排查日志的时候会方便快速地找到对应的内容进行查看。

glusterfs 集群的默认日志路径都是在/var/log/glusterfs 下面的,其中这里分为几大类的日志。

日志路径	作用
/var/log/glusterfs/glusterd.log	glusterd 进程日志,该进程是管理每个节点上服务的主进程.
/var/log/glusterfs/cli.log	glusterfs client 在 server 端的日志
/var/log/glusterfs/cmd_history.log	集群执行相关命令的结果历史记录日志,包括执行 status 等.
/var/log/glusterfs/bricks/<path extraction of brick path>.log	每个 brick 在该节点的相关日志
/var/log/glusterfs/VOLNAME-rebalance.log	volume 进程 rebalance 重平衡操作的日志,当 volume 进行扩容或者缩容时会触发 rebalance.
/var/log/glusterfs/glustershd.log /var/log/glusterfs/gfsheal-VOLNAME.log	集群 volume 自愈时产生的信息日志文件,当节点宕机后重新加入集群,数据需要校验恢复,就会触发 heal 功能.
/var/log/glusterfs/quotad.log /var/log/glusterfs/quota-crawl.log /var/log/glusterfs/quota-mount- VOLNAME.log	quota 功能相关日志,就是对 volume 的数据容量大小进行限制的

/var/log/glusterfs/nfs.log	glusterfs nfs 功能日志
/var/log/glusterfs/geo-replication	和不同 glusterfs 集群之间进行数据迁移有关的功能日志

这里实际上还有其他不同的日志，当然有些可能会比较少用到，如果有需要的话，可以到官方文档中进行查看 glusterfs 不同日志文件[2]。另外这里除了和集群有关的日志文件以外，如果是使用了 k8s 集群的话，还要关注以下几个目录下的文件，注意这里是在 k8s 集群节点上的日志文件，并不是在 glusterfs 集群上的。

日志路径	作用
/var/lib/kubelet/plugins/kubernetes.io/glusterfs	这里是 k8s 下的不同存储下的 pvc 日志,有时客户端报错的日志可以在这里查看
/var/log/syslog	k8s 的日志文件信息默认输出到系统日志中

最后,如果想调整一下 volume 的日志文件级别，可以修改以下两个参数，设置为 DEBUG 级别的话，会有更加详细的日志输出信息。

参数	作用
----	----

diagnostics.brick-log-level	控制 brick 端的日志级别,影响 glusterd 和 brick 日志输出
diagnostics.client-log-level	控制 client 日志输出级别

章节语:

在最开始,我们尝试着在部署了一下 glusterfs 集群之后,简单地创建和体验了一些不同类型的 volume 特点,去简单理解一下 glusterfs 的使用,而仲裁节点的出现,也是一个非常好的亮点,同时还去关心了一下集群运维中比较重要的日志文件内容。

附录引用:

- [1] <https://gluster.readthedocs.io/en/latest/>
- [2] <https://gluster.readthedocs.io/en/latest/Administrator-Guide/Logging>

2. 第二章 文件系统的那些事

2.1.1. 文件系统的层次结构

生产环境的服务器中使用的基本上是 linux 操作系统，而操作系统中会涉及到进程模块，内存管理和文件系统等各大模块，那么对于一个正常的文件系统，如常见的 ext4 和 xfs 这些，一个普通的读写会经历哪些层次呢？这些都是非常值得关心的内容，下面请先看一张图。

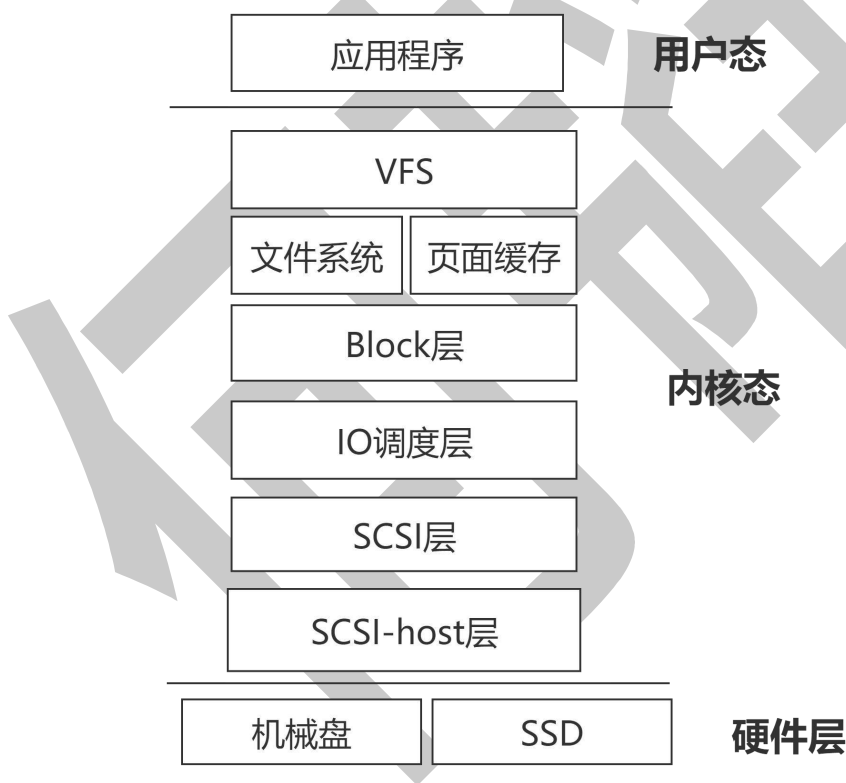


图 2.1-1 linux 文件系层次图

对于一个正常的应用服务，如果要读写本地磁盘数据，会向本地的文件系统发起请求，例如使用 mkdir 或者 touch 这样的 shell 命令进行操作，那么首先会经过一个叫 VFS 层，这个其实是 linux 操作系统中对于文件系统的一个规划，也就是说，不管是自定义的操作系

统，还有一些 ext4 这些，都需要遵守的一些规范，而 VFS 中最核心的元素就是 inode,dentry,superblock 和 file 四个元素。

inode 是负责记录每个文件的一些元数据信息的(在 linux 系统中，一切皆文件，不管是目录还是 devices 设备，都会封装看成一个文件一样调用)。dentry 对象则是用于记录文件的结构关系的，也就是不同目录的上下级层级结构关系和树状关系等。superblock 就是超级块，用于管理整个操作系统中 inode 资源整体使用情况等，如果把文件系统比作一个图书馆，那么图书馆里面的每一本书就是 block，而书的分类和标签信息这些就是 inode,superblock 就是统计整理整个图书馆的资源情况的。file 对象就是用于记录一下每一本书的租借情况，对于文件来说，这个文件是否被打开过，是否产生了文件句柄 fd(所谓的文件句柄可以理解为，在 file 对象和打开的进程的数据结构中做了一些标记)等。

定义了 VFS 的规范之后，所有的文件系统的都要使用这四个元素进行管理文件系统，而每一个文件系统，在注册到操作系统里面时，还需要定义该文件系统的 fileOptions(这里可以理解为，每个文件系统的一些操作如何定义与具体实现，例如打开一个文件时，是否使用缓存，或者实现一些特殊的操作定义，在 glusterfs 中也有该实现，叫做 FOP)。那么有了这些 FOP 之后，一个正常的读写请求就会到达该文件系统，根据调用的函数不同，这里就会区分到底请求是否使用缓存了，也就是是否使用操作系统的缓存，而使用这个缓存的优势与缺点也是非常明显的，如果想达到快速读取，那么如果缓存中有该数据，就不需要从磁盘中进行读取了，但是对于写入来说，尤其是数据库的一些写入，因为很多在应用的内存里面已经做了一次缓存，因此并不需要再次做缓存，而且数据库是对于写入 IO 是比较敏感的服务，为了加快写入速度，通常会不使用操作系统的缓存的。

不管是否使用缓存，对于一个读写请求来说，最后都是要到达 block 层的，而这一层的出现，其实就是为了封装所有的读写请求为一个 bio 对象。而为什么需要 block 层，一

个很重要的原因就是，操作系统上面，可能会有多个不同的文件系统，不管任何文件系统最后想要对磁盘进行读写之前，都要进行统一的格式封装，而这就是 block 层的 bio 对象所需要做的事情了，同时在这里，对于读写请求来说，还有考虑能否合并请求，如果两个请求的写入磁盘位置是前后关联的，那么这里就可以进行合并操作了，如果请求可以进行合并的话，那么封装后的 bio 对象就会传递给调度队列了，接下来就是常见的调度算法进行操作了，有 Deadline 和 CFQ 等常见的算法。

当然对于 glusterfs 来说，也实现了一套 fuse，GlusterFS 是一个用户空间文件系统。GlusterFS 开发人员选择这种方法是为了避免在 Linux 内核中使用模块。下图将简单展示一下 glusterfs 的 fuse，结合 linux 文件系统的内容，其实两者是有非常多的共同点的。

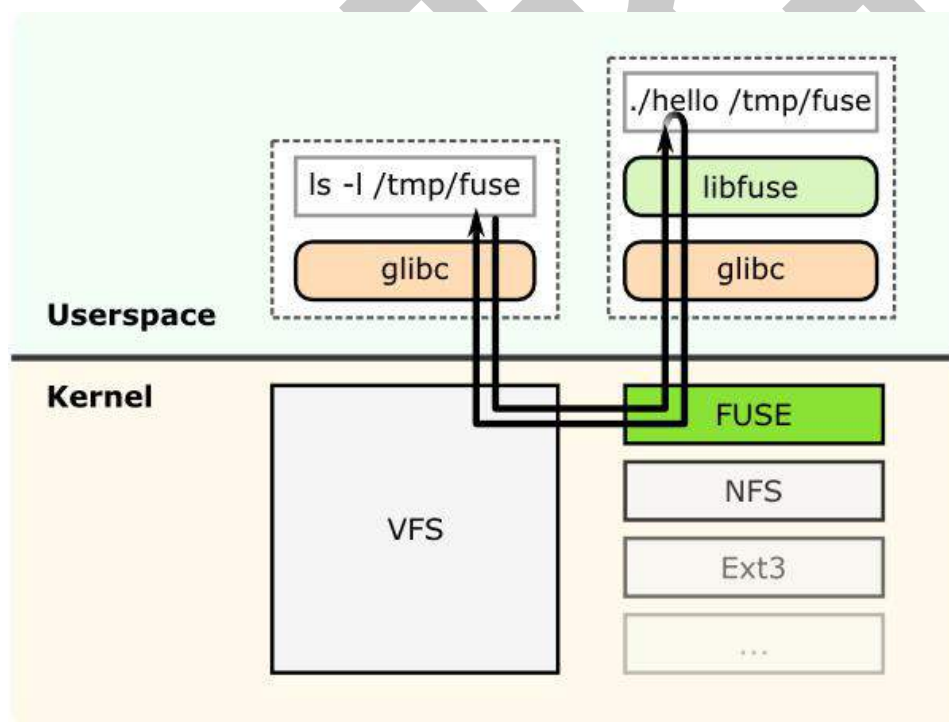


图 2.1-2 glusterfs fuse 架构图

接下来，将结合代码，具体介绍一下上面提到的内容，该部分内容是为了让大家更好地理解一些概念，如果已经熟悉该部分内容，可以跳过阅读。

2.1.2. 有趣的 VFS 和文件系统

VFS 在 linux 的出现 ,不得不惊叹是一个神来之笔 ,下面先来看一下内核代码中的 ext4 文件系统与 VFS 关系图。

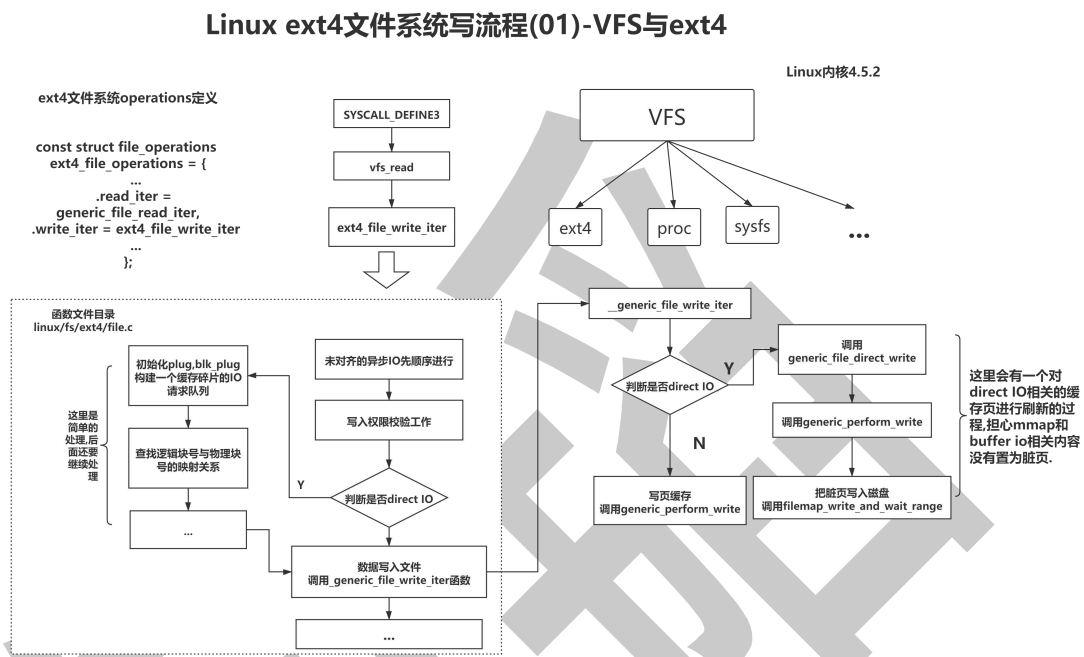


图 2.1.1-1 vfs 与 ext4 代码关系图

首先图中的一个读请求,是经过了 `vfs_read` 然后调用了 `ext4_file_write_iter`, 而后者则是 `ext4` 文件系统注册在操作系统中的 FOP,这里在调用的时候,会根据文件系统注册的函数进行回调,然后该函数就是真正地进入了文件系统的操作逻辑了,这里也就是有了两个分支,就是前面提到的是否使用操作系统内核缓存的问题了。下面为了让大家更好地理解一下 VFS 的元素,将具体介绍一下内容。

2.1.2.1. VFS inode

为了更好地理解一下 inode 信息,下面给出一个普通文件的 `stat` 命令结果,该命令可

以列出文件的一些属性信息。

```
1. $ sudo ls -li test.sh
2. 1328 test.sh
3.
4.
5. $ sudo stat test.sh
6. File: test.sh
7. Size: 614      Blocks: 19      IO Block: 1024   regular file
8. Device: 74h/116d Inode: 1328      Links: 1
9. Access: (0644/-rw-r--r--)  Uid: (  0/   root)   Gid: (  0/   root)
10. Access: 2020-11-07 03:21:30.777590296 +0000
11. Modify: 2020-11-07 03:21:26.485583231 +0000
12. Change: 2020-11-07 03:21:26.493583244 +0000
13. Birth: -
```

这里 inode 号为 1328,这里还要留意一下 change,modify 和 access 三个时间,这里也就是 ctime,mtime 和 atime,在 glusterfs 中的文件,也是利用了这三个信息进行一些文件更新校验的工作。另外这里的 size,也是在 glusterfs 中使用的一个很重要的信息。

change: 也就是 ctime,最后一次改变文件或目录(属性)的时间,例如执行 chmod, chown 等命令。

modity: 也就是 mtime,是最后一次修改文件或目录(内容)的时间。

access: 也就是 atime,是最后一次访问文件或目录的时间。

```
1. $ sudo stat dir
2. File: dir
3. Size: 2      Blocks: 1      IO Block: 131072 directory
4. Device: 74h/116d Inode: 131881    Links: 2
5. Access: (0755/drwxr-xr-x)  Uid: (  0/   root)   Gid: (  0/   root)
6. Access: 2021-06-03 03:21:17.577886842 +0000
7. Modify: 2021-06-03 03:21:17.577886842 +0000
8. Change: 2021-06-03 03:21:17.577886842 +0000
9. Birth: -
```

与文件的不同,目录的 stat 信息中的 Links 注意是 2 的,其他信息则相似。当然 inode 中的信息远远不只这么少,还包括了其他很多的信息,例如包括了用户和所属组的权限,关

联的 superblock 等信息。具体想了解的可以看看 inode 的数据结构，该结构体定义在 <linux/fs.h> 中。

```
1.  struct inode
2.  {
3.      /* 哈希表 */
4.      struct hlist_node    i_hash;
5.      /* 索引节点链表 */
6.      struct list_head     i_list;
7.      /* 目录项链表 */
8.      struct list_head     i_dentry;
9.      ...
10.     uid_t                 i_uid;
11.     gid_t                 i_gid;
12.     ...
13.     struct timespec       i_atime;
14.     struct timespec       i_mtime;
15.     struct timespec       i_ctime;
16.     ...
17.     /* 索引节点操作表 */
18.     struct inode_operations *i_op;
19.     /* 默认的索引节点操作 */
20.     struct file_operations *i_fop;
21.     /* 相关的超级块 */
22.     struct super_block     *i_sb;
23.     /* 文件锁链表 */
24.     struct file_lock       *i_flock;
25.     /* 相关的地址映射 */
26.     struct address_space   *i_mapping;
27.     ...
28. }
```

2.1.2.2. VFS superblock

对于超级块来说,可以使用 df -i 命令来简单查看一下操作系统层面的 inode 使用情况。

```
1.  $ sudo df -i
2.  Filesystem      Inodes   IUsed   IFree IUse% Mounted on
3.  udev             2023350    620  2022730    1% /dev
```

4.	tmpfs	2029590	1379	2028211	1% /run
5.	/dev/nvme0n1p2	30498816	1481554	29017262	5% /
6.	tmpfs	2029590	3222	2026368	1% /dev/shm
7.	tmpfs	2029590	7	2029583	1% /run/lock
8.	tmpfs	2029590	18	2029572	1% /sys/fs/cgroup

这里使用 `df -i` 命令可以直观地看到目前系统中使用的 inode 情况 ,同时当 inode 耗尽的时候 ,也是一个非常要注意的问题。这些在日常的操作系统维护中都是需要监控的内容。

另外下面给出 superblock 的对象结构。

```

1.  struct super_block {
2.      ...
3.      // 关联设备
4.      dev_t    s_dev;
5.      // 块大小
6.      unsigned long s_blocksize;
7.      ...
8.      // 文件系统类型,常见的有 ext,xfs 等
9.      struct file_system_type *s_type;
10.     // 超级块的操作函数
11.     const struct super_operations *s_op;
12.     ...
13.     // 该 super_block 中所有 inode 的 i_sb_list 成员的双向链表
14.     struct list_head s_inodes;
15.     ...
16.
17. }
```

2.1.2.3. VFS struct file

`struct file` 结构体定义在 `include/linux/fs.h` 中 ,其中每一个文件结构体表示被进程打开的一个文件 ,而系统中每个被打开的文件 ,都可以在内核空间中找到关联的一个 `struct file` 对象。这个对象由内核在打开文件时创建 ,并且传递给在文件上进程操作的函数。在文件的所有实例都关闭后 ,内核释放这个数据结构。下面先简单感受一下在操作系统中 ,文件打开后的变化。

```

1. [root@gfsclient01 ~]# lsof |grep test
2.  COMMAND    PID  TID   USER  FD      TYPE          DEVICE  SIZE/OFF      NODE NAME
3.  ....
4.  vim          1827      root   4u     REG          253,0      12288  100663370 /root/.test.sh.
    swp
5.  /root/.test.sh.swp
6. [root@gfsclient01 ~]# ps -ef |grep 1677
7. root        1827  1529   0 10:09 pts/0    00:00:00 vim test.sh

```

在 linux 中，打开了一个文件之后，其实并不是真正意义上直接对原文件进行修改的，而是会把原文件复制一份加上.swp 之后，对应的所有的操作都会在该临时的缓存文件中，而一旦修改完成进行合并的时候，才会替换到原文件。当然这里会有常见的文件冲突，就是当文件被打开之后，如果再次打开该文件，操作系统会检测到有临时缓存文件存在，那么就证明可能是有修改没完成，或者之前的修改中断了没有合并，这样会导致冲突，但是可以进行替换操作等恢复。

下面给出一种使用文件句柄来恢复数据的方式。当该文件有被进程引用 fd，那么就有机会恢复原来的数据。

```

1. [root@gfsclient01 ~]# lsof |grep test_file.txt
2.  tail        2165      root   3r     REG          253,0        10  100730896 /root/test_file.
    txt
3. [root@gfsclient01 ~]# ls -l /proc/2165/fd/3
4. lr-x-----. 1 root root 64 Jun  3 10:20 /proc/2165/fd/3 -> /root/test_file.txt
5. [root@gfsclient01 ~]# cat /proc/2165/fd/3
6. #123
7.
8. 321
9. [root@gfsclient01 ~]# rm -fr test_file.txt
10. [root@gfsclient01 ~]# cat /proc/2165/fd/3 > abc.txt
11. [root@gfsclient01 ~]# cat abc.txt
12. #123
13.
14. 321

```


这里首先创建了一个名字为 test_file.txt 的文件，然后在一个终端使用 tail -f 来阅读该文件，这样可以保证文件一直在被打开的状态，同时 fd 中也可以看到是 read 状态。接着通过 lsof 找到该文件对应的引用进程，在 proc 下可以找到该文件的缓存内容，这样就可以进行数据恢复了。

同样的，为了进一步了解 struct file 的数据结构，该结构定义在内核代码中的 include/linux/fs.h 中，下面给出部分代码内容。

```
1.  struct file {
2.      ...
3.      struct path      f_path;
4.      struct inode      *f_inode;
5.      //和文件关联的操作.
6.      const struct file_operations  *f_op;
7.      spinlock_t      f_lock;
8.      atomic_long_t      f_count;
9.      //文件标志,有 O_RDONLY 和 O_SYNC 等.
10.     unsigned int      f_flags;
11.     //文件模式
12.     fmode_t      f_mode;
13.     struct mutex      f_pos_lock;
14.     //当前读写位置
15.     loff_t      f_pos;
16.     struct fown_struct f_owner;
17.     const struct cred      *f_cred;
18.     struct file_ra_state  f_ra;
19.     ...
20. }
```

2.1.2.4. VFS dentry

dentry，也就是目录项，是多个文件或者目录的链接，通过这个链接可以找寻到目录之下的文件或者是目录项。dentry 在文件系统里是极其重要的一个概念，dentry 结构体

在 linux 内核里也是用处广泛，这个结构体定义在 include/linux/dcache.h 里。

```
1.  struct dentry {
2.      //目录项标志
3.      unsigned int d_flags;
4.      seqcount_t d_seq;
5.      //散列表表项的指针
6.      struct hlist_bl_node d_hash;
7.      //父目录的目录项
8.      struct dentry *d_parent;
9.      struct qstr d_name;
10.     // 与文件名关联的索引节点
11.     struct inode *d_inode;
12.     unsigned char d_iname[DNAME_INLINE_LEN];
13.
14.
15.     struct lockref d_lockref;
16.     //dentry 的操作函数
17.     const struct dentry_operations *d_op;
18.     //文件的超级块对象
19.     struct super_block *d_sb;
20.     unsigned long d_time;
21.     void *d_fsdata;
22.
23.     struct list_head d_lru;
24.     //父列表的子级
25.     struct list_head d_child;
26.     struct list_head d_subdirs;
27.
28.     union {
29.         struct hlist_node d_alias;
30.         struct rcu_head d_rcu;
31.     } d_u;
32. };
```

为了观察操作系统中目录的关系结构，可以使用 tree 命令进行查看。

```
1.  [root@gfscclient01 ~]# tree /tmp/
2.  /tmp/
3.  ├── ks-script-YbuVG1
4.  └── systemd-private-4556ac58b212443eb846ec7ab9a7f059-chronyd.service-soC2Da
```

```

5. |   └─ tmp
6. └─ yum.log
7.   └─ yum_save_tx.2021-06-03.09-45.cn30by.yumtx
8.
9. 2 directories, 3 files

```

2.1.2.5. 文件对象

有了前面的四个核心基本元素，那么如果要打开一个文件，就必须要使用文件对象了，它的作用是描述进程和文件交互的关系，说白了就是要知道哪个进程在打开这个文件，并且打开读取的文件位置等信息。文件的结构定义如下所示。

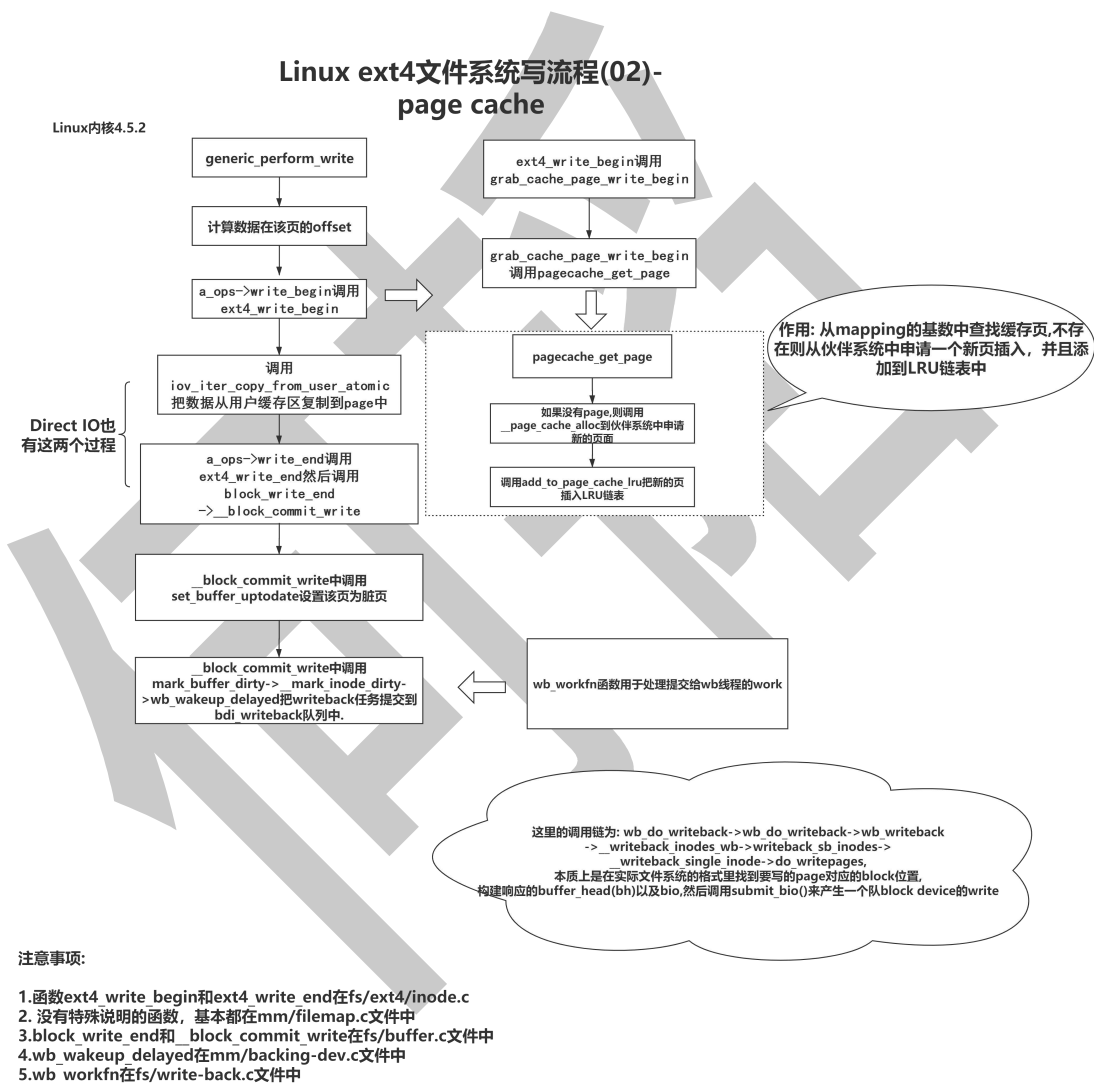
```

1. struct file {
2.     //指向文件对应的 dentry 结构
3.     struct dentry                *f_dentry;
4.     //指向文件所属于的文件系统的 vfsmount 对象,和挂载 有关
5.     struct vfsmount              *f_vfsmnt;
6.     //指向文件的 file operation
7.     const struct file_operations *f_op;
8.     ...
9.     //记录进程对文件操作的位置。对文件读取前 n 字节，那么其指向 n+1 字节位置
10.    loff_t                        f_pos;
11.
12.    struct fown_struct             f_owner;
13.    //表示文件的用户的 uid 和 gid
14.    unsigned int                   f_uid;
15.    unsigned int                   f_gid;
16.    //用于记录文件预读的设置
17.    struct file_ra_state           f_ra;
18.
19.    //这个结构是指向一个封装文件的读写缓存页面的结构
20.    struct address_space           *f_mapping;
21.
22. }

```

2.1.3. page cahe 的作用

回到前面提的内容，对于一个读写请求来说，经过前面的 VFS 和文件系统的调用，那么下面就会进入到内核调用中，而如果要使用内核缓存，这里就有了 page cache 的存在目的了。下面先来简单看一张代码调用逻辑图，理解一下其中的思路。



对于一个读写请求，如果这里发现在缓存中没有找到对应的数据，那么就需要申请 page 缓存页了，而这里会涉及到 linux 操作系统的内存管理模块伙伴系统和 mapp 相关的内容。而所谓的伙伴系统，如果熟悉 java jvm 的话，其实这里的思想是类似的。伙伴系

统是一个结合了 2 的方幂个分配器和空闲缓冲区合并技术的内存分配方案，其基本思想很简单。内存被分成含有很多页面的大块，每一块都是 2 个页面大小的方幂。如果找不到想要的块，一个大块会被分成两部分，这两部分彼此就成为伙伴。其中一半被用来分配，而另一半则空闲。这些块在以后分配的过程中会继续被二分直至产生一个所需大小的块。当一个块被最终释放时，其伙伴将被检测出来，如果伙伴也空闲则合并两者。

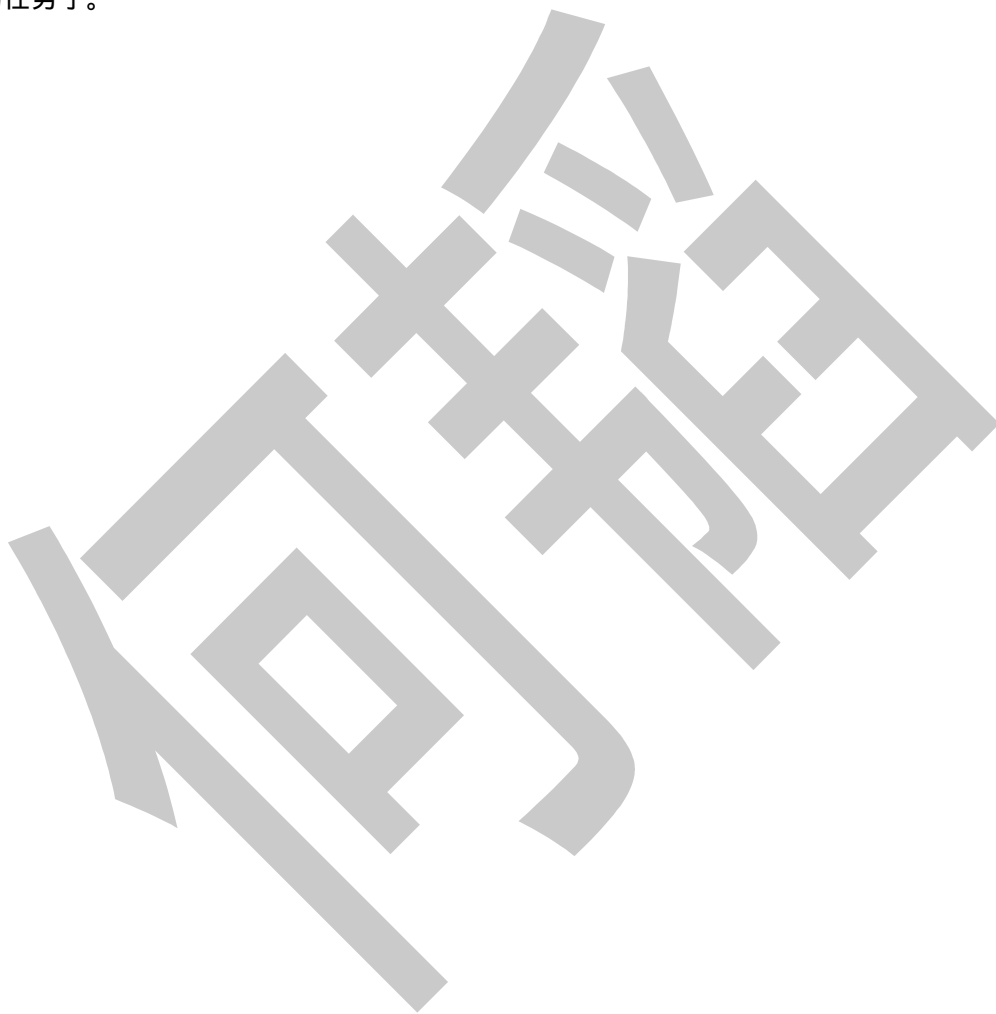
那么不管内存是如何管理的，申请出来的页面，最后会被存放到一个叫做 LRU 链表进行管理。在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是 Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。

而内核的内存管理部分，会有三个概念，分别是 node, zone 和 page, 简单理解，page 就是一个数据页，zone 是一个区域分组，CPU 被划分为多个节点(node)，内存则被分簇，每个 CPU 对应一个本地物理内存，即一个 CPU-node 对应一个内存簇 bank，即每个内存簇被认为是一个节点。

有了这些缓存的数据页之后，那么数据就会从用户缓存区复制到 page 当中，这里就会

涉及到内核调用，同时为了进一步优化，这里会有零拷贝的技术出现了。当然因为这里并不打算深入具体去了解每一部分的内容，因此如果感兴趣的话，可以自行查阅相关资料。

而当数据存放到内核缓存之后，那么写入的请求会把请求封装成一个 bio 对象，提交到 bdi_writeback 队列中，而这里会有一个 writeback 机制，也就是回写机制，下面就是 block 层的任务了。



2.1.4. Fuse block

为了更好地简单理解读写请求后面要做的事情，可以先看看下面这张图。

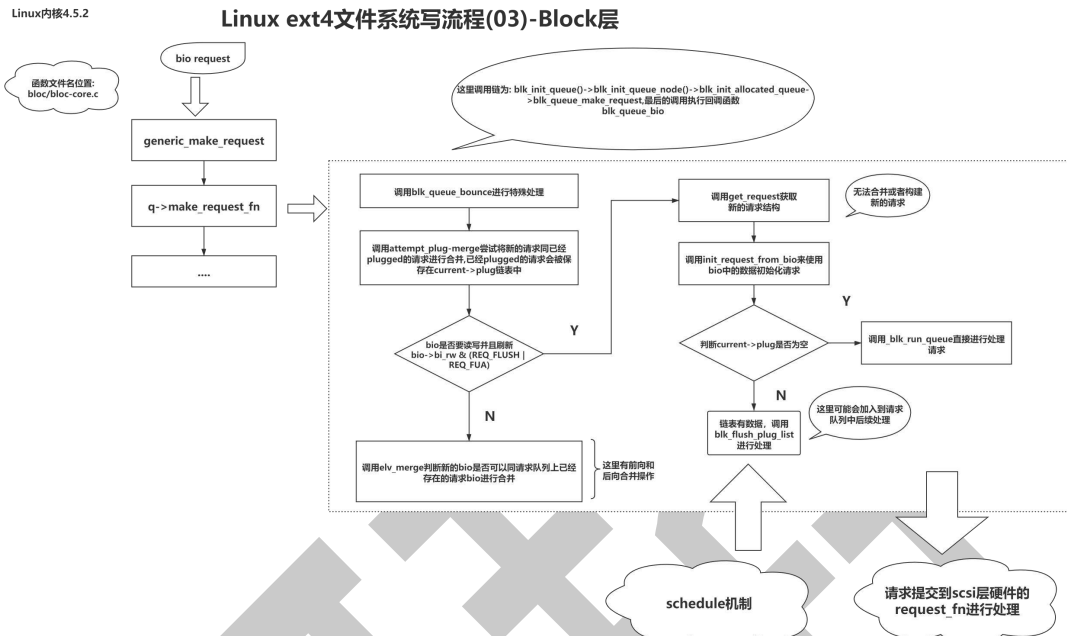


图 2.1.3-1 fuse block 层原理

对于一个 IO 请求，除了前面提到的把数据写入到缓存以外，其实还有一个叫直接 IO 的内容，那么关于 IO 我们常常还有听到同步与异步 IO 的内容，而异步 IO 则只能使用直接 IO 来实现的。另外对于一个 bio 请求来说，这里还会进行前向与后向的合并，之后会放入到调度队列里面，等待调度算法来进行调度处理。

最后，在经历了一系列的原理的理解之后，我们使用 strace 命令来了解一下，在 linux 系统中创建一个文件时使用的函数调用吧。

```
1. # strace touch 1.txt
2. execve("/usr/bin/touch", ["touch", "1.txt"], 0x7ffe4906edc8 /* 24 vars */) = 0
3. brk(NULL) = 0x13ef000
4. mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b1044000
```

```

5.  access("/etc/ld.so.preload", R_OK)          = -1 ENOENT (No such file or directory)
6.  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7.  fstat(3, {st_mode=S_IFREG|0644, st_size=25560, ...}) = 0

8.  mmap(NULL, 25560, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5b103d000
9.  close(3)                                     = 0
10. open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
11. read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0"..., 832) = 832
12. fstat(3, {st_mode=S_IFREG|0755, st_size=2156352, ...}) = 0
13. mmap(NULL, 3985920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff5b0a56000
14. mprotect(0x7ff5b0c1a000, 2093056, PROT_NONE) = 0
15. mmap(0x7ff5b0e19000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c3000) = 0x7ff5b0e19000
16. mmap(0x7ff5b0e1f000, 16896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff5b0e1f000
17. close(3)                                     = 0
18. mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103c000
19. mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103a000
20. arch_prctl(ARCH_SET_FS, 0x7ff5b103a740) = 0
21. mprotect(0x7ff5b0e19000, 16384, PROT_READ) = 0
22. mprotect(0x60d000, 4096, PROT_READ) = 0
23. mprotect(0x7ff5b1045000, 4096, PROT_READ) = 0
24. munmap(0x7ff5b103d000, 25560) = 0
25. brk(NULL) = 0x13ef000
26. brk(0x1410000) = 0x1410000
27. brk(NULL) = 0x1410000
28. open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
29. fstat(3, {st_mode=S_IFREG|0644, st_size=106176928, ...}) = 0
30. mmap(NULL, 106176928, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5aa513000
31. close(3) = 0
32. open("1.txt", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
33. dup2(3, 0) = 0

```


34. close(3)	= 0
35. utimensat(0, NULL, NULL, 0)	= 0
36. close(0)	= 0
37. close(1)	= 0
38. close(2)	= 0
39. exit_group(0)	= ?
40. +++ exited with 0 +++	

这里有常见的 mmap 和 brk 是和内存管理分配有关的函数，open 是文件打开的函数等，其中 open 函数中的参数 O_CREAT 是创建并打开一个新文件,关于这些不同的函数的作用与参数意义，可以根据需要时去查阅接口文档。

章节语:

这一章我们主要是理解了一些 linux 文件系统概念，一个正常的读写请求所经过的层次结构，而这里面涉及到内存管理，进程调度等模块，本章也只是非常粗糙简略地讲解了 VFS 和缓存的一些内容，而这个章节的内容，也是为了方便理解后续 glusterfs fuse 中的一些参数做准备的，因为 glusterfs fuse 的底层也是很多我们所熟悉的系统调用，而且 glusterfs fuse 是一个用户文件系统。

3. 第三章 glusterfs 的核心概念

从本章开始，有了前面的关于 linux 的一些文件系统的简单认识，那么从这里开始真正去认识一下 glusterfs 这种无中心架构的特点，因为无中心架构，那么必然在元数据存储与节点之间通信，数据恢复等方面会与常见的架构有着非常大的不同，而这也是 glusterfs 的魅力之处，下面开始将一点点地深入去理解 glusterfs 是如何运作的。另外这里先做个简单的约定，下面的代码内容如无特殊说明，则基本是在虚拟机环境中进行的实验操作，对于集群节点则是以 gfs01,gfs02 这样来命令规范的，而客户端机器则是 gfsclient01 这样的，因此阅读代码的时候，可以进行区别开来。

关于实验的虚拟机的 ip 与 hostname 对应关系如下所示..

```
1. # cat /etc/hosts
2. ...
3. 192.168.0.110 gfs01
4. 192.168.0.111 gfs02
5. 192.168.0.112 gfs03
```

3.1. 有趣的扩展属性 gfid

3.1.1. 文件的 gfid 属性

因为 glusterfs 是无中心架构的，这点在前面的内容中也多次提到过，当然可能会有困惑，那么到底这个文件是存放到哪里的，由什么来决定呢？对于 glusterfs 来说，这里巧妙地利用了 linux 文件的扩展属性，而一个文件除了有基本的元数据以外，还支持扩展属性的，而其中有一个叫做 gfid 的扩展属性是

关键的作用，下面来感受一下。

```
1. # gluster volume info test-replica
2.
3. Volume Name: test-replica
4. Type: Replicate
5. Volume ID: 4568c063-5b75-4304-98f6-21f3955cc138
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 192.168.0.110:/glusterfs/test-replica
12. Brick2: 192.168.0.111:/glusterfs/test-replica
13. Brick3: 192.168.0.112:/glusterfs/test-replica
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
```

首先这里有一个 3 副本的复制卷，然后在一个客户端虚拟机上面进行挂载，接着查看一下扩展属性。

```
1. [root@gfsclient01 ~]# mount -t glusterfs -o aux-gfid-mount 192.168.0.110:test-replica /mnt/test-replica
2. [root@gfsclient01 ~]# cd /mnt/test-replica/
3. [root@gfsclient01 test-replica]# getfattr -n glusterfs.gfid.string a.txt
4. # file: a.txt
5. glusterfs.gfid.string="71b9fb49-53ae-42f8-bb1b-b53af17521fc"
```

从这里可以看到 glusterfs 的文件是有一个叫做 gfid 的扩展属性的，那么这里还可以到 brick 所在的目录下面进行查看一下信息。

```
1. [root@gfs01 test-replica]# getfattr -d -m . -e hex a.txt
2. # file: a.txt
3. trusted.gfid=0x71b9fb4953ae42f8bb1bb53af17521fc
```


1. [root@gfs01 test-replica]# ls -i .glusterfs/71/b9/71b9fb49-53ae-42f8-bb1b-b53af17521fc
2. 68224979 .glusterfs/71/b9/71b9fb49-53ae-42f8-bb1b-b53af17521fc
3. [root@gfs01 test-replica]# ls -i a.txt
4. 68224979 a.txt

从这里就可以知道 ,在这个.glusterfs 隐藏目录下面 ,有一个该文件的硬链接 ,而其中 gfid 的前四位 ,每两位就是隐藏目录下面的目录名称。因此这里就是非常重要的地方了 ,因此对于 glusterfs 来说 ,所有的操作 ,对文件的增删改查 ,都是需要先找到该文件的 ,而寻找该文件 ,也就是 lookup 操作 ,都是基于 inode 的 ,而不是直接去找 path 的。

3.1.2. 目录的 gfid 属性

前面看了文件的 gfid 属性，那么下面来看看目录的该属性是否有不一样的地方。首先还是在客户端挂载，然后创建一个目录，然后查看一下属性。

```
1. [root@gfsclient01 test-replica]# getfattr -n glusterfs.gfid.string dir01
2. # file: dir01
3. glusterfs.gfid.string="a0e974d1-0902-40ba-ad21-7436f8bf31cf"
4.
5.
6.
7. [root@gfs03 test-replica]# ls -l
8. total 0
9. -rw-r--r-- 2 root root 0 Jun 4 10:46 a.txt
10. drwxr-xr-x 2 root root 19 Jun 4 11:20 dir01
11. drwxr-xr-x 2 root root 6 Jun 4 11:20 dir02
12. [root@gfs03 test-replica]# getfattr -d -m . -e hex dir01
13. # file: dir01
14. trusted.gfid=0xa0e974d1090240baad217436f8bf31cf
15. trusted.glusterfs.dht=0x00000000000000000000000000000000ffffffff
```



```

10. trusted.glusterfs.mdata=0x010000000000000000000000000060bc7
    b0b0000000260310640000000060bc7b0b00000000260310640000000
    060bc7b0b0000000026031064
11.
12. [root@gfs03 test-replica]# ls -l .glusterfs/a1/06/a1064e3
    f-a66e-4663-a69a-112ccad4eaaa
13. lrwxrwxrwx 1 root root 55 Jun  6 03:36 .glusterfs/a1/06/a
    1064e3f-a66e-4663-a69a-112ccad4eaaa -> ../../a0/e9/a0e974d
    1-0902-40ba-ad21-7436f8bf31cf/dir001
14.
15. [root@gfs03 test-replica]# ls .glusterfs/a0/e9/a0e974d1-0
    902-40ba-ad21-7436f8bf31cf -l
16. lrwxrwxrwx 1 root root 54 Jun  4 11:20 .glusterfs/a0/e9/a
    0e974d1-0902-40ba-ad21-7436f8bf31cf -> ../../00/00/0000000
    0-0000-0000-0000-000000000001/dir01

```

从这里的信息显示，很明显，不管是创建文件还是目录，这里会在隐藏目录下面创建一个链接文件，而这里的 dir001 因为是在目录 dir01 下面的，所以隐藏目录下的上一层路径就是父目录的 gfid，然后再通过父目录的 gfid 就可以找到父目录的路径了。接着观察一下那个很多个 0 最后是 1 的目录隐藏路径下的内容吧。

```

1. [root@gfs03 test-replica]# ls -l .glusterfs/00/00/0000000
    0-0000-0000-0000-000000000001/
2. total 0
3. -rw-r--r-- 2 root root 0 Jun  4 10:46 a.txt
4. drwxr-xr-x 3 root root 33 Jun  6 03:36 dir01
5. drwxr-xr-x 2 root root 6 Jun  4 11:20 dir02
6. drwxr-xr-x 2 root root 6 Jun  6 03:43 dir03
7.
8. [root@gfs03 test-replica]# ls -l /glusterfs/test-replica/
9. total 0
10. -rw-r--r-- 2 root root 0 Jun  4 10:46 a.txt
11. drwxr-xr-x 3 root root 33 Jun  6 03:36 dir01
12. drwxr-xr-x 2 root root 6 Jun  4 11:20 dir02
13. drwxr-xr-x 2 root root 6 Jun  6 03:43 dir03

```

通过这里的一些例子的观察，当然这里只是两层目录，如果多层目录下面，

大家可以测试一下效果会是怎样的。

最后这里简单总结一下 gfid 这个扩展属性的特点：

1. 在复制卷中，每一个 brick 的文件和目录的 gfid 都是相同的。
2. 创建文件和目录，都会在隐藏目录下面.glusterfs 创建对应的链接文件
3. gfid 属性的这一串字符里面，前四位是用于标识隐藏目录名称的。其中每两位是一个隐藏目录下的目录名称。
4. 隐藏目录中，对文件创建是硬链接，目录是软连接。

那么最后了解一下，这个 gfid 到底是怎么来的呢？这里可以在代码中找到相关的内容，libglusterfs/src/inode.c 中有一个 hash_gfid 的函数。

```
1. static int
2. hash_gfid(uuid_t uuid, int mod)
3. {
4.     return ((uuid[15] + (uuid[14] << 8)) % mod);
5. }
```


3.2. 数据内存模型

对于 glusterfs 来说，是使用了一些扩展属性来维护元数据信息的，那么根据前面讲的 gfid 这个扩展属性，通过隐藏目录拿到了路径和名称这些信息，那如果要打开一个文件，对于 glusterfs 来说是如何做到的呢？这个就涉及到数据在内存中的模型了，了解这部分内容，可以进一步理解一下，到底什么是 fuse，glusterfs 的 fuse 和一般的文件系统区别在哪里。这里为了区别 glusterfs 定义的数据结构与操作系统中常见的元素的区别，如 inode，那么在 glusterfs 中使用 inode_t 来指代其定义的结构，这个称呼也是从官方中继承下来的(虽然代码中命名并没有这样写，但是看到官方的资料是这样提到的，因此为了统一习惯也就这样称呼吧)。

3.2.1. loc_t 结构

对于一个文件系统来说，正常要找到一个文件，可以通过目录一层层地寻找，而对于 glusterfs 来说，则是通过隐藏目录 glusterfs 下的 gfid 来找到对应的路径，然后找到文件的 inode 信息，而有了 inode 这些信息，才能够进一步对文件进行读写操作。因此对于从文件路径再到找到文件的 inode，这里需要一个叫做 lookup 的文件操作(也就是 file operation,简称 fop,这个在后面会多次提到，可以简单理解为文件系统封装的一些操作逻辑，这里不理解没关系，后面看多了会慢慢理解的)，这个 fop 解决的就是知道一个文件路径如何找到该文件的 Inode 问题。当然为了解决这个问题，就不得不提到 glusterfs 的一些数据内存模型了，首先要讲到的就是 loc_t 结构，这个结构中就是记录了关键的一些数据

结构信息，下面一起来了解一下。

```
1. struct _loc {
2.     const char *path;
3.     const char *name;
4.     inode_t *inode;
5.     inode_t *parent;
6.     uuid_t gfid;
7.     uuid_t pargfid;
8. };
```

这个数据结构内容在源码中 libglusterfs/src/glusterfs/xlator.h 文件里面，这里面记录的数据结构信息也非常直接易懂，主要就是当前的路径，文件名称，当前文件和父目录的 inode,gfid 信息。那么通过路径和名称等等找到 inode 信息之后，就可以真正找到这个文件了。

3.2.2. inode_t 结构

那么前面了解了 loc_t 结构之后，看到 inode_t 也是一个结构，那么这里 inode_t 到底记录了什么呢？这个结构的作用又是什么？下面也来简单了解一下。

```
1. struct _inode {
2.     //把活跃的 inode 信息记录在内存的一张表
3.     inode_table_t *table;
4.     uuid_t gfid;
5.     gf_lock_t lock;
6.     gf_atomic_t nlookup;
7.     //文件句柄 fd 的统计数量,也就是文件被打开的累计次数
8.     uint32_t fd_count;
9.     //活跃的 fd 打开统计次数
```

```

10.     uint32_t active_fd_count;
11.     //这个 Inode 被引用的统计数量
12.     uint32_t ref;
13.     //文件类型
14.     ia_type_t ia_type;
15.     //记录打开了这个文件的 fd 的 list 列表
16.     struct list_head fd_list;
17.     //此 inode 的目录项列表
18.     struct list_head dentry_list;
19.     struct list_head hash;
20.     //和 lru 相关的
21.     struct list_head list;
22.
23.     struct _inode_ctx *_ctx;
24.     //如果 inode 是不活跃的则设置这个属性
25.     bool in_invalidate_list;
26.     bool invalidate_sent;
27.     //记录 inode 是否在 lru 列表里面
28.     bool in_lru_list;
29. };

```

那么从这里也可以简单了解到，inode_t 这个结构里面，记录的主要还是和操作系统里面有关的信息比较多，主要就是该 Inode 信息是否活跃，文件类型等信息。另外通过这个 inode_t 的结构，还记录和当前这个文件被打开的文件句柄(简称 fd)相关的信息,通过这些信息，就可以比较好地在追踪文件是否还被引用或者文件是否已经关闭等等。

当然从这两个简单的数据结构中，同时对比前面提到的 vfs 里面的一些数据结构 inode 等，所谓的 fuse，就可以简单理解为对一些操作系统的内容进行自定义封装的一套文件系统接口。而 glusterfs 则也是封装了一套 glusterfs fuse

接口的，这样做的优缺点也是非常明显的，优点是可以自定义做很多特定的需求与想法，缺点就是实现难度比较大，如团队技术水平不足时，或者代码设计出现问题时，很容易造成一些额外的重要 bug，例如 glusterfs 的 fuse 则在 7.7 之前的版本，有一个文件句柄泄露的重大 bug 出现了，会导致在 mount 的时候掉线，当然关于 glusterfs 的一些版本问题等，后面也会再次提到。

3.2.3. dentry_t 结构

这个结构其实也是 vfs 中的四个基本元素中 dentry 对应的，这里 glusterfs 对其的结构定义代码在 libglusterfs/src/glusterfs/inode.h 文件中。该结构记录的信息也比较简单，而 dentry_t 和 inode_t 结构，是做打开和修改文件等操作之前，需要获取的两个数据结构的信息，而如何获取，就是后面提到的 lookup 接口实现的功能。

```
1. struct _dentry {
2.     //记录 dentry 下的 inode 列表
3.     struct list_head inode_list;
4.     // 指向 hash 表
5.     struct list_head hash;
6.     // 当前目录的 inode
7.     inode_t *inode;
8.     // 当前目录的名称
9.     char *name;
10.    // 父目录的 inode
11.    inode_t *parent;
12.};
```

3.2.4. fd_t 结构

有了前面的 `inode_t` 和 `dentry_t` 结构，在 `glusterfs` 中就可以调用打开文件的相关操作了(`open` fop, 这个会在后面提到), 这里 `fd_t` 结构的代码文件在 `libglusterfs/src/glusterfs/fd.h` 中。

```
1.  struct _fd {
2.      uint64_t pid;
3.      //系统调用 open 中的 flag, 如 O_CREAT 表示不存在则创建等
4.      int32_t flags;
5.      //fd 引用统计
6.      gf_atomic_t refcount;
7.      struct list_head inode_list;
8.      //打开的文件所对应的 inode
9.      struct _inode *inode;
10.     ...
11. };
```

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,
19.             dfd);
19.         //获取 stat 信息
20.         ret = sys_fstatat(dfd, gfid_path, &statbuf, 0)
21.         ;
22.         if (ret == 0 && ((statbuf.st_mode & S_IFMT) =
23.             = S_IFDIR))
24.             goto parent;
25.         ret = sys_fstatat(dfd, gfid_path, &statbuf, A
26.             T_SYMLINK_NOFOLLOW);
27.         if (ret == 0 && statbuf.st_nlink == 1) {
28.             gf_msg(this->name, GF_LOG_WARNING, op_errn
29.                 no,
30.                 P_MSG_HANDLE_DELETE,
31.                 "Found stale gfid "
32.                 "handle %s, removing it.",
33.                 gfid_path);
34.             posix_handle_unset(this, gfid, NULL);
35.         }
36.     }
37.     goto parent;
38. }
39. ...
40. }

```

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。

lookup(loc_t,xdata) —————> (gfid,stat) —————> inode_t

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 *loc,
3.             int32_t flags,
4.             fd_t *fd, dict_t *xdata)
5.  {
6.      ....
7.      //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
8.      _fd = sys_open(real_path, flags, priv->force_create_mode);
9.      if (_fd == -1) {
10.         op_ret = -1;
11.         op_errno = errno;
12.         gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_FILE_OP_FAILED,
13.                "open on gfid-handle %s (path: %s), flags: %d", real_path,
14.                loc->path, flags);
15.         goto out;
16.     }
```

```

17.    //修改 ctime 时间
18.    posix_set_ctime(frame, this, real_path, -1, loc->inode, &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, &preop, 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_PATH_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_opensat(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 = opensat(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 *loc, 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
10.               IR_FAILED,
11.                  "mkdir of %s failed", real_path);
12.          goto out;
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
22.                   WN_FAILED,
23.                      "chown on %s failed", real_path);
24.              goto out;
25.          }
26.          #endif
27.          op_ret = posix_acl_xattr_set(this, real_path, xdata);
28.          if (op_ret) {
29.              gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_ACL
30.                   _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,

```

```

61.             _gf_false);
62.     if (op_ret == -1) {
63.         op_errno = errno;
64.         gf_msg(this->name, GF_LOG_ERROR, errno, P_MSG_LST
        AT_FAILED,
65.             "post-operation lstat on parent of %s fail
        ed", real_path);
66.         goto out;
67.     }
68.
69.     //这里是修改父目录的 ctime
70.     posix_set_parent_ctime(frame, this, par_path, -1, loc
        ->parent, &postparent);
71.
72.     op_ret = 0;
73.     ...
74. }

```

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 `gid`，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。

```

1. int
2. 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_CREAT 的，也就是说当文件不存在的时候会创建的，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

```

9. Self-heal Daemon on gfs01	N/A	N/A
Y 1438		
10.		
11. Task Status of Volume test-replica		
12. -----		

13. 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。

```

1. [root@gfs01 ~]# ls /proc/1427/fd -l
2. total 0
3. lr-x----- 1 root root 64 Jun  9 10:56 0 -> /dev/null
4. l-wx----- 1 root root 64 Jun  9 10:56 1 -> /dev/null
5. ...
6. l-wx----- 1 root root 64 Jun  9 10:57 273 -> /glusterfs/
   test-replica/.glusterfs/0d/0f/0d0fc592-afe8-441c-a736-6146
   491ae571
7. ....

```

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。

```

1. [root@gfsclient01 ~]# getfattr -n glusterfs.gfid.string
   /mnt/test-replica/210609_02.txt
2. getfattr: Removing leading '/' from absolute path names
3. # file: mnt/test-replica/210609_02.txt
4. 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。



3.4. 一些重要的概念与进程

前面提到了一些 glusterfs 的数据内存模型，还有一些 posix 接口的实现，那么 glusterfs 当中其中还有一些其他比较重要的概念和进程，因为这些概念有些是代码架构设计层面的，有些是和配置文件有关的，有些进程则是 glusterfs 集群中非常重要的进程，而这些内容的关联性比较紧密，因此打算把这几样东西一起介绍一下。同时有些概念因为涉及到代码规范设计模式那样的，可能理解起来不太好一下子理解，但本书的目的也并没有计划把 glusterfs 的源码理解个通透，在源码解读方面，本书的定位还是希望保持入门带领大家认识一下 glusterfs（其实就是本人水平有限呀，很多细节和内容理解起来也没有很好掌握~）。

3.4.1. 代码模块功能 xlator

从前面的一些代码路径上面，大家可能会留意到这个名词 xlator，当然在官方文档中，还有时会看到 translators 这个名词，这两个其实都是一样的。而且 xlator 还是一个文件路径的开头，例如前面提到的文件路径 `xlators/storage/posix/src/posix-entry-ops.c`，从这里可以看到，xlator 的作用应该是起到一个很重要的划分项目不同功能的作用。是的，一个项目里面（不管是使用 java 或者 c，又或者其他 go 等开发语言开发的项目），项目功能都会有很多层划分，而在 glusterfs 当中就使用了这种叫 xlator 的架构，这是一种模块化、堆栈式的架构，可通过灵活的配置支持高度定制化的应用环境，比如大文件存储、海量小文件存储、云存储、多传输协议应用等。每个功能以模块形式实现，然后以积木方式进行简单的组合，即可实现复杂的功能。

下面可以来看看 xlator_api 的定义。

```
1. xlator_api_t xlator_api = {
2.     .init = init,
3.     .fini = fini,
4.     .notify = notify,
5.     .reconfigure = reconfigure,
6.     .mem_acct_init = mem_acct_init,
7.     .op_version = {1},
8.     //这里和 statedump 的内容有关
9.     .dumpops = &dumpops,
10.    //定义的 file operation
11.    .fops = &fops,
12.    // 和回调函数有关
13.    .cbks = &cbks,
14.    .options = options,
15.    .identifier = "replicate",
16.    .category = GF_MAINTAINED,
17. };
```

这是 afr 功能的 xlator 定义,afr 是 glusterfs 使用的一个数据恢复机制,关于这部分内容后面会讲。而该内容是在文件 xlator/cluster/afr/src/afr.c 中定义的。当然这里还可以拿其他模块的进行对比,例如 ec 模块的,这里路径在 xlator/cluster/ec/src/ec.c 中。

当然这里还有一个 xlator 的定义,该内容是在 libglusterfs/src/glusterfs/xlator.h 中,定义如下所示。

```
1. struct _xlator {
2.
3.     char *name;
4.     char *type;
5.     char *instance_name;
6.
7.     //指向下一个 xlator
```

```

8.     xlator_t *next;
9.     //指向上一个 xlatro
10.    xlator_t *prev;
11.    xlator_list_t *parents;
12.    xlator_list_t *children;
13.    dict_t *options;
14.
15.    //和 linux 的 dl 库有关的
16.    void *dlhandle;
17.    struct xlator_fops *fops;
18.    struct xlator_cbks *cbks;
19.    struct xlator_dumpops *dumpops;
20.    struct list_head volume_options;
21.    .....
22.    //用于记录 xlator 的一些特殊的信息
23.    void *private;
24.    ....
25. }

```

这里有一个信息叫做 private 的，这个在每个 xlator 中还有定义，下面来简单了解一下。

```

1.  typedef struct _afr_private {
2.      ...
3.      //用于记录一些可能影响决策一致性的因素
4.      uint32_t event_generation;
5.      char vol_uuid[UUID_SIZE + 1];
6.
7.      ...
8.      gf_boolean_t esh_granular;
9.      ...
10. }
11.
12. ...
13.
14. typedef struct afr_granular_esh_args {
15.     fd_t *heal_fd;
16.     xlator_t *xl;

```

```

17.     call_frame_t *frame;
18.     gf_boolean_t mismatch;
19.
20. } afr_granular_esh_args_t;

```

这里有一个叫 `esh_granular` 的变量，这个变量记录的是 `afr` 这个模块中和 `granular` 有关的功能。而在 `atf.c` 中就有相关的 `options` 参数选项，例如 `granular-entry-heal`，当然这里在 `afr.c` 中还有和该相关相关的一些设置代码，如下所示。

```

1.  int32_t
2.  init(xlator_t *this)
3.  {
4.      ...
5.      priv->granular_locks = (strcmp(locking_scheme, "gran
   ular") == 0);
6.      GF_OPTION_INIT("full-lock", priv->full_lock, bool, ou
   t);
7.      GF_OPTION_INIT("granular-entry-heal", priv->esh_granu
   lar, bool, out);
8.      ...
9.  }

```

这个函数是一个初始化的函数，这里就有一些和参数 `granular` 相关的设置了。

另外那么这里每个模块都有一些 `fops` 的，如果要重新实现的话，这里的定义同样是在该文件中的，如下所示。

```

1.  struct xlator_fops fops = {
2.      .lookup = afr_lookup,
3.      .lk = afr_lk,
4.      .flush = afr_flush,
5.      .statfs = afr_statfs,
6.      ...
7.
8.      /* inode read */

```

```

9.      .access = afr_access,
10.     .stat = afr_stat,
11.     ...
12.
13.     /* inode write */
14.     .writev = afr_writev,
15.     .truncate = afr_truncate,
16.     .ftruncate = afr_ftruncate,
17.     ...
18.
19.     /*inode open*/
20.     .opendir = afr_opendir,
21.     .open = afr_open,
22.
23.     /* dir read */
24.     .readdir = afr_readdir,
25.     .readdirp = afr_readdirp,
26.
27.     /* dir write */
28.     .create = afr_create,
29.     .mknod = afr_mknod,
30.     ...
31. };

```

但是这里并不是所有的 fop 都会重新定义实现的，那么这里就会使用 default 的内容了。那么这里 default fop 在哪里呢？就是在 libglusterfs/src/default-tmpl.c

```

1.  struct xlator_fops _default_fops = {
2.      .create = default_create,
3.      .open = default_open,
4.      .stat = default_stat,
5.      .readlink = default_readlink,
6.      .mknod = default_mknod,
7.      .mkdir = default_mkdir,
8.      .unlink = default_unlink,
9.      .rmdir = default_rmdir,
10.     ...
11.
12. }

```

这里就是定义的 default 的 fop 了，如果感兴趣的，可以对比一下 default

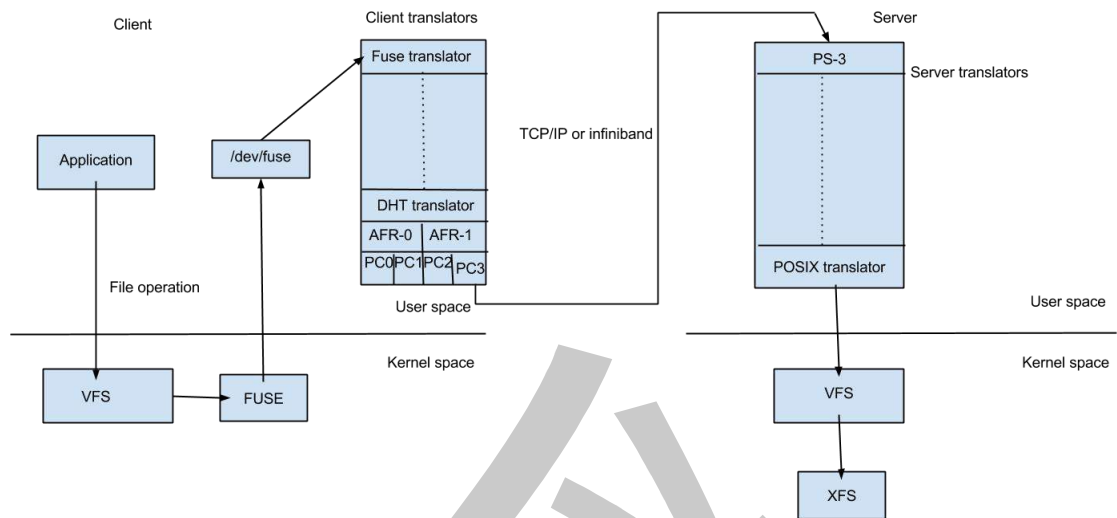
的内容与模块实现的不同。

当然对于 xlator 这里的定义的数据成员的作用，如果有些不明白的，建议暂时不需要全部深入理解，在后面的内容中，其中部分内容会详细讲解一下。另外在 afr.c 文件中，这里还可以看到很多参数的定义，因为不同的模块，涉及的参数不同，在 afr.c 中可以看到这个参数。

```
1.  {
2.      .key = {"quorum-count"},
3.      .type = GF_OPTION_TYPE_INT,
4.      .min = 1,
5.      .max = INT_MAX,
6.      .default_value = 0,
7.      .op_version = {1},
8.      .flags = OPT_FLAG_CLIENT_OPT | OPT_FLAG_SETTABLE
9.      | OPT_FLAG_DOC,
10.     .tags = {"replicate"},
11.     /*.option = quorum-count*/
12.     /*.validate_fn = validate_quorum_count*/
13.     .description = "If quorum-type is \"fixed\" only
14.         allow writes if "
15.         "this many bricks are present. Other quorum types "
16.         "will OVERWRITE this value.",
17. }
```

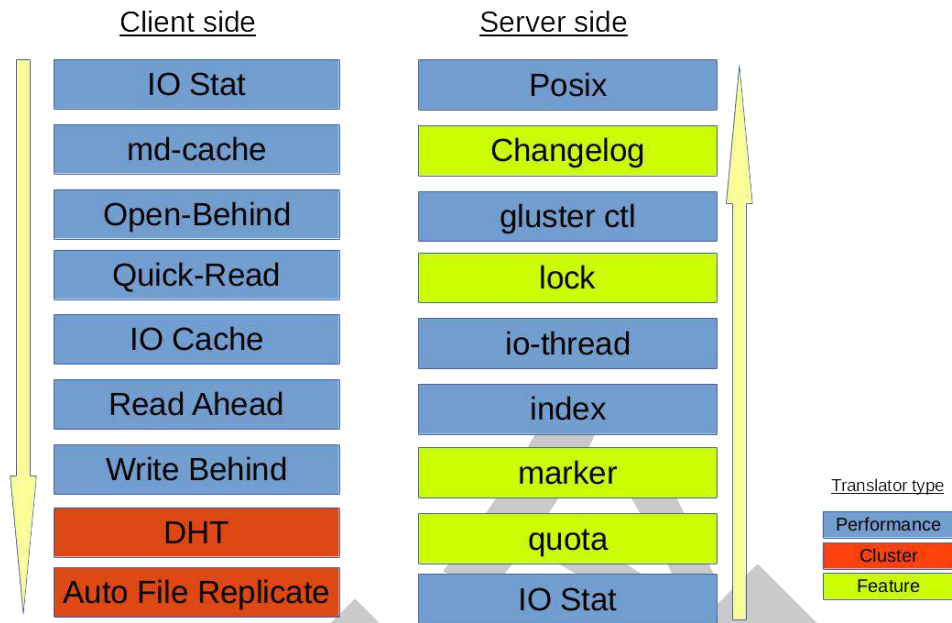
这个参数的作用就是前面仲裁节点中提到的，因此未来如果遇到一些参数，想要知道这个参数到底在哪个模块中发挥作用，就可以使用这种方式在代码中找到对应的位置进行判断了。

那么这里一个挂载从客户端到服务端，会经过哪些 xlator 呢？可以见下图所示。



在客户端挂载一个 volume 的时候,客户端的 glusterfs 进程会与服务端的 glusterd 进程沟通,服务端的 glusterd 进程会发送一个配置文件叫(vol file),里面包含了一系列客户端模块(xlator)和其他信息的内容(例如 volume 每个 brick 的信息)。通过这个文件,客户端 glusterfs 进程就可以与 volume 的每一个 brick 的 glusterfsd 进程进行沟通了,然后就可以进行其他的操作了。

当然这里的 vol file 文件包含很多内容,这个内容就是 graph,简单理解就是不同的 xlator 模块的组合,也就是图中的 client translators 和 server translators 的内容,那么这里会有一些什么模块呢?可以见下图所示。



这里有很多不同的 translator(也叫 xlator), 如 read ahead 就是文件预读相关的功能, io cache 就是 IO 缓存相关的, dht 则是 glusterfs 中的一个重要的哈希算法, changlog 是一个用于记录文件操作与恢复的, quota 则是给 volume 进行容量限制的。这里其中有些功能, 会在后面介绍。

那么最后关于 xlator ,这里简单总结以下 glusterfs 中使用这种方式来组织代码层次结构的一些好处优点吧。

1. 代码松耦合, 这一点在日常开发中比较容易理解。
2. xlator 在不同环境下具备很好的可重用性与可插拔性, 也就是增加新的 xlator 结构比较容易。
3. 可以通过 volfile 文件来实现 graph, 这样不同层次结构之间的关系比较明确和清晰。graph 就是不同 xlator 的组合起来, 一个 volume 中会有多个不同的 xlator 组成的, 其中的关系顺序就是 graph。volfile 可以先简单理解为和 xml 一样的有一定规范的配置文件。

3.4.2. glusterfs 的几个重要进程

这里想简单分享一下关于 glusterfs 中的几个重要的进程，这一小节的内容也与前面的关系不大，可能日常运维中有些大家也见过了，下面列举几个重要的常见的进程。

1) gluster : 是 cli 命令执行工具，主要功能是解析命令行参数，然后把命令发送给 glusterd 模块执行。

2) glusterd: 是一个管理模块，处理 gluster 发过来的命令，处理集群管理、存储池管理、brick 管理、负载均衡、快照管理等。集群信息、存储池信息和快照信息等都是以配置文件的形式存放在服务器中，当客户端挂载存储时，glusterd 会把存储池的配置文件发送给客户端。

3) glusterfsd : 是服务端模块，存储池中的每个 brick 都会启动一个 glusterfsd 进程。此模块主要是处理客户端的读写请求，从关联的 brick 所在磁盘中读写数据，然后返回给客户端。

4) glusterfs : 是客户端模块，负责通过 mount 挂载集群中某台服务器的存储池，以目录的形式呈现给用户。当用户从此目录读写数据时，客户端根据从 glusterd 模块获取的存储池的配置文件信息，通过 DHT 算法计算文件所在服务器的 brick 位置，然后通过 Infiniband RDMA 或 Tcp/Ip 方式把数据发送给 brick，等 brick 处理完，给用户返回结果。存储池的副本、条带、hash、EC 等逻辑都在客户端处理。

除了这几个核心最基本的进程以外，当然这里还有关于 heal 的 shd 进程，还有和 rebalance(重平衡，也就是当一个 volume 扩容，添加 brick 之后，数据需要重新平衡)相关的进程等等，关于这些进程，在后面提到不同功能的时候

再进行分享。下面先来再感受一下这几个核心进程的关系。

```
1. [root@gfs03 ~]# systemctl status glusterd
2. • glusterd.service - GlusterFS, a clustered file-system s
   erver
3.    Loaded: loaded (/usr/lib/systemd/system/glusterd.servi
   ce; enabled; vendor preset: disabled)
4.    Active: active (running) since Sun 2021-06-13 05:28:42
   EDT; 28min ago
5.    Docs: man:glusterd(8)
6.    Process: 897 ExecStart=/usr/sbin/glusterd -p /var/run/g
   lusterd.pid --log-level $LOG_LEVEL $GLUSTERD_OPTIONS (code
   =exited, status=0/SUCCESS)
7.    Main PID: 902 (glusterd)
8.    CGroup: /system.slice/glusterd.service
9.           └─ 902 /usr/sbin/glusterd -p /var/run/glusterd.
   pid --log-level INFO
10.            └─ 955 /usr/sbin/glusterfsd -s 192.168.0.112 -
   -volfile-id test-arbiter.192.168.0.112.glusterfs-test-arbi
   ter -p /var/run/gluster/vols/test-arbiter/192.168...
11.            └─ 974 /usr/sbin/glusterfsd -s 192.168.0.112 -
   -volfile-id test-disperse.192.168.0.112.glusterfs-test-dis
   perse -p /var/run/gluster/vols/test-disperse/192....
12.            └─ 999 /usr/sbin/glusterfsd -s 192.168.0.112 -
   -volfile-id test-replica.192.168.0.112.glusterfs-test-repl
   ica -p /var/run/gluster/vols/test-replica/192.168...
13.            └─1026 /usr/sbin/glusterfs -s localhost --volf
   ile-id shd/test-arbiter -p /var/run/gluster/shd/test-arbit
   er/test-arbiter-shd.pid -l /var/log/glusterfs/glu...
14.
15. Jun 13 05:28:42 gfs03 systemd[1]: Starting GlusterFS, a c
   lustered file-system server...
16. Jun 13 05:28:42 gfs03 systemd[1]: Started GlusterFS, a cl
   ustered file-system server.
```

从这里的进程关系中就比较好理解 glusterfsd 和 glusterd 的关系了 ,另外

这里还可以从/usr/sbin/中看出两者的关系。

```
1. [root@gfs01 ~]# ls -l /usr/sbin/gluster*
2. -rwxr-xr-x. 1 root root 453864 May 10 19:13 /usr/sbin/glu
   ster
3. lrwxrwxrwx. 1 root root      10 May 24 12:45 /usr/sbin/glu
   sterd -> glusterfsd
```

```

4.  lrwxrwxrwx. 1 root root      10 May 24 12:45 /usr/sbin/glu
sterfs -> glusterfsd
5.  -rwxr-xr-x. 1 root root 251384 May 10 19:13 /usr/sbin/glu
sterfsd
6.  -rwxr-xr-x. 1 root root  11232 May 10 19:13 /usr/sbin/glu
ster-setgid2path

```

从这里可以知道，其实 glusterfs 和 glusterd 都是来自 glusterfsd 的。当然这里就可以使用这些命令来手动启动一些进程了，下面进行一下小实验，当然在做这个实验之前，先看看测试的 volume 情况。

```

1.  [root@gfs02 ~]# gluster volume info test-replica
2.
3.  Volume Name: test-replica
4.  Type: Replicate
5.  Volume ID: 4568c063-5b75-4304-98f6-21f3955cc138
6.  Status: Started
7.  Snapshot Count: 0
8.  Number of Bricks: 1 x 3 = 3
9.  Transport-type: tcp
10. Bricks:
11. Brick1: 192.168.0.110:/glusterfs/test-replica
12. Brick2: 192.168.0.111:/glusterfs/test-replica
13. Brick3: 192.168.0.112:/glusterfs/test-replica
14. Options Reconfigured:
15. diagnostics.client-log-level: DEBUG
16. diagnostics.brick-log-level: DEBUG
17. cluster.granular-entry-heal: on
18. storage.fips-mode-rchecksum: on
19. transport.address-family: inet
20. nfs.disable: on
21. performance.client-io-threads: off

```

这里是一个很简单的复制卷，同时有三个 brick，下面进行测试。

```

1.  [root@gfs01 ~]# ps aux |grep glusterfs
2.  root      1402  0.0  0.3 875904 14328 ?        Ssl  05:28
    0:00 /usr/sbin/glusterfsd -s 192.168.0.110 --volfile-id
    test-arbiter.192.168.0.110.glusterfs-test-arbiter -p /var
    /run/gluster/vols/test-arbiter/192.168.0.110-glusterfs-tes
    t-arbiter.pid -S /var/run/gluster/fdd86b6311f78aa9.socket
    --brick-name /glusterfs/test-arbiter -l /var/log/glusterfs

```

```

/bricks/glusterfs-test-arbiter.log --xlator-option *-posix.
glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b54 --proce
ss-name brick --brick-port 49152 --xlator-option test-arbi
ter-server.listen-port=49152
3.  root          1413  0.0  0.3 875904 15324 ?          Ssl  05:28
    0:00 /usr/sbin/glusterfsd -s 192.168.0.110 --volfile-id
    test-disperse.192.168.0.110.glusterfs-test-disperse -p /v
    ar/run/gluster/vols/test-disperse/192.168.0.110-glusterfs-
    test-disperse.pid -S /var/run/gluster/fa9f90c834b1bc98.soc
    ket --brick-name /glusterfs/test-disperse -l /var/log/glus
    terfs/bricks/glusterfs-test-disperse.log --xlator-option *
    -posix.glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b54
    --process-name brick --brick-port 49153 --xlator-option te
    st-disperse-server.listen-port=49153
4.  .....
5.
6.
7.  [root@gfs01 ~]# ps aux |grep glusterd
8.  root          904  0.0  0.2 524836  9452 ?          Ssl  05:28
    0:00 /usr/sbin/glusterd -p /var/run/glusterd.pid --log-
    level INFO
9.  root          1402  0.0  0.3 875904 14328 ?          Ssl  05:28
    0:00 /usr/sbin/glusterfsd -s 192.168.0.110 --volfile-id
    test-arbiter.192.168.0.110.glusterfs-test-arbiter -p /var
    /run/gluster/vols/test-arbiter/192.168.0.110-glusterfs-tes
    t-arbiter.pid -S /var/run/gluster/fdd86b6311f78aa9.socket
    --brick-name /glusterfs/test-arbiter -l /var/log/glusterfs
    /bricks/glusterfs-test-arbiter.log --xlator-option *-posix.
    glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b54 --proce
    ss-name brick --brick-port 49152 --xlator-option test-arbi
    ter-server.listen-port=49152
10. root          1413  0.0  0.3 875904 15324 ?          Ssl  05:28
    0:00 /usr/sbin/glusterfsd -s 192.168.0.110 --volfile-id
    test-disperse.192.168.0.110.glusterfs-test-disperse -p /v
    ar/run/gluster/vols/test-disperse/192.168.0.110-glusterfs-
    test-disperse.pid -S /var/run/gluster/fa9f90c834b1bc98.soc
    ket --brick-name /glusterfs/test-disperse -l /var/log/glus
    terfs/bricks/glusterfs-test-disperse.log --xlator-option *
    -posix.glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b54
    --process-name brick --brick-port 49153 --xlator-option te
    st-disperse-server.listen-port=49153
11. .....
12.
13.

```

14. //下面删掉所有相关的进程

```
15. [root@gfs01 ~]# killall glusterd glusterfs glusterfsd
```

```
16. [root@gfs01 ~]# ps aux |grep glusterd
```

```
17. root      1964  0.0  0.0 112812   976 pts/0    R+   06:30
    0:00 grep --color=auto glusterd
```

```
18. [root@gfs01 ~]# ps aux |grep glusterfs
```

```
19. root      1966  0.0  0.0 112812   976 pts/0    R+   06:30
    0:00 grep --color=auto glusterfs
```

这里首先使用 killall 干掉所有相关的进程，主要就是 glusterd 和 glusterfsd。然后下面启动 glusterfs 进程，如下所示。

```
1. [root@gfs01 ~]# glusterfs -f /var/lib/glusterd/vols/test-
   replica/trusted-test-replica.tcp-fuse.vol /mnt/test-replic
   a
```

```
2. [root@gfs01 ~]# df -h /mnt/test-replica
```

```
3. Filesystem
```

```
      Size  Used Avail Use% Mounted on
```

```
4. /var/lib/glusterd/vols/test-replica/trusted-test-replica.
   tcp-fuse.vol  50G  8.7G  42G  18% /mnt/test-replica
```

```
5. [root@gfs01 ~]# ps aux |grep glusterfs
```

```
6. root      1984  0.3  0.2 629228  8304 ?        Ssl  06:48
    0:02 glusterfs -f /var/lib/glusterd/vols/test-replica/t
   rusted-test-replica.tcp-fuse.vol /mnt/test-replica
```

```
7.
```

8. //这里在其他节点上查看 volume status

```
9. [root@gfs02 ~]# gluster volume status test-replica
```

```
10. Status of volume: test-replica
```

```
11. Gluster process                                TCP Port  RDM
    A Port  Online  Pid
```

```
12. -----
    -----
```

```
13. Brick 192.168.0.111:/glusterfs/test-replica 49154      0
    Y      1423
```

```
14. Brick 192.168.0.112:/glusterfs/test-replica 49154      0
    Y      999
```

```
15. Self-heal Daemon on localhost                N/A        N/A
    Y      1436
```

```
16. Self-heal Daemon on gfs03                    N/A        N/A
    Y      1026
```



```

17.
18. Task Status of Volume test-replica
19. -----
    -----
20. There are no active volume tasks

```

这里可以看到，虽然挂载的进程启动了，但是 brick 相关进程是没有的，同时在其他节点上，使用 volume status 也是看不到相关信息的。另外为了测试到底这个挂载进行是否生效的，可以在挂载目录下创建一个文件，然后在其他节点下的 brick 目录查看这个文件内容，这里的测试结果如下所示。

```

1. [root@gfs01 ~]# touch /mnt/test-replica/210613.txt
2. [root@gfs01 ~]# date >> /mnt/test-replica/210613.txt
3.
4. [root@gfs02 ~]# cat /glusterfs/test-replica/210613.txt
5. Sun Jun 13 07:08:14 EDT 2021

```

另外这里如果想手动启动 glusterfsd 进程，可以根据 brick 中的日志内容，找到之前启动的一些类似的命令进行重启，但是往往这里还要先判断端口是否存在占用的情况。这里的日志类似如下所示。

```

1. [2021-05-25 16:11:42.442930 +0000] I [MSGID: 100030] [glusterfsd.c:2687:main] 0-/usr/sbin/glusterfsd: Started running version [{arg=/usr/sbin/glusterfsd}, {version=9.2}, {cmdlinestr=/usr/sbin/glusterfsd -s 192.168.0.110 --volfile-id test-replica.192.168.0.110.glusterfs-test-replica -p /var/run/gluster/vols/test-replica/192.168.0.110-glusterfs-test-replica.pid -S /var/run/gluster/d403ef3161bd809c.socket --brick-name /glusterfs/test-replica -l /var/log/glusterfs/bricks/glusterfs-test-replica.log --xlator-option *-posix.glusterd-uuid=99827066-72c7-484f-b1e8-0a9c4bc46b54 --process-name brick --brick-port 49152 --xlator-option test-replica-server.listen-port=49152}]

```

3.4.3. graph 和 volfile

这里分享一下什么是 volfile 和 graph,其实在前面的一节内容中,这里也有使用到 volfile 文件,就是在/var/lib/glusterd/vols 下面的文件中,如下所示。

```
1. [root@gfs01 ~]# ls /var/lib/glusterd/vols/test-replica/  
2. bricks                quota.conf  
                           test-replica.gfproxyd.vol  
3. cksum                 snapd.info  
                           test-replica-shd.vol  
4. info                 test-replica.192.168.0.110.glusterfs-test-replica.vol  
                           test-replica.tcp-fuse.vol  
5. node_state.info      test-replica.192.168.0.111.glusterfs-test-replica.vol  
                           trusted-test-replica.tcp-fuse.vol  
6. quota.cksum          test-replica.192.168.0.112.glusterfs-test-replica.vol  
                           trusted-test-replica.tcp-gfproxy-fuse.vol  
7.  
8. [root@gfs01 ~]# ls /var/lib/glusterd/vols/test-replica/test-replica.192.168.0.110.glusterfs-test-replica.vol  
9. /var/lib/glusterd/vols/test-replica/test-replica.192.168.0.110.glusterfs-test-replica.vol
```

这里的 vol 文件就是一个 volfile 了,那么这里为什么 glusterfs 官方不使用 xml,json 或者 yaml 文件呢?因为 glusterfs 是项目在 2005 就开始了,那时候现在主流的配置文件格式这些还没有,或者还不是很出名,因此当时就选择 volfile 了,也一直保留了下来。另外因为这个配置文件的格式问题,也不算是集群的一些影响到性能或者代码层次结构的,因此虽然没有用到现在主流的方式,但是并不影响真正使用。

那么这里 volfile 中记录的其实就是 graph,也就是不同的 xlator 功能结构的组织结构顺序,下面可以简单看看这个文件来感受一下。

```
1. [root@gfs01 ~]# cat /var/lib/glusterd/vols/test-replica/test-replica.192.168.0.110.glusterfs-test-replica.vol
```

```

2. volume test-replica-posix
3.     type storage/posix
4.     option shared-brick-count 1
5.     option fips-mode-rchecksum on
6.     option volume-id 4568c063-5b75-4304-98f6-21f3955cc138

7.     option directory /glusterfs/test-replica
8. end-volume
9.
10. volume test-replica-trash
11.     type features/trash
12.     option trash-internal-op off
13.     option brick-path /glusterfs/test-replica
14.     option trash-dir .trashcan
15.     subvolumes test-replica-posix
16. end-volume
17.
18. ....
19.
20. volume /glusterfs/test-replica
21.     type debug/io-stats
22.     option global-threading off
23.     option count-fop-hits off
24.     option latency-measurement off
25.     option threads 16
26.     option log-level DEBUG
27.     option volume-id 4568c063-5b75-4304-98f6-21f3955cc138

28.     option unique-id /glusterfs/test-replica
29.     subvolumes test-replica-quota
30. end-volume
31.
32. volume test-replica-server
33.     type protocol/server
34.     option transport.listen-backlog 1024
35.     option transport.socket.keepalive-count 9
36.     option transport.socket.keepalive-interval 2
37.     option transport.socket.keepalive-time 20
38.     option transport.socket.ssl-enabled off
39.     option transport.socket.keepalive 1
40.     option auth.addr./glusterfs/test-replica.allow *
41.     option auth-path /glusterfs/test-replica
42.     option auth.login.fad65c15-ff26-4465-914a-eb83bb332fff
        f.password b2784284-715c-4e46-a1ca-97b86eb7fa1a

```

```
43.      option auth.login./glusterfs/test-replica.allow fad65
        c15-ff26-4465-914a-eb83bb332fff
44.      option transport.address-family inet
45.      option transport-type tcp
46.      subvolumes /glusterfs/test-replica
47. end-volume
```

这里有几个地方值得注意的，首先是这个文件的格式，每一个 xlator 层次都是用 volume 然后 end-volume 来结束的，这样看起来和 xml 中的格式有点类似。接着这里还要留意一下 subvolumes 的内容，subvolumes 是什么内容呢？简单理解就是记录下一个 xlator 是哪个，在 volfile 文件中，这里要注意到，文件的顺序是从下往上的，这一点就可以从 subvolumes 的内容可以看出来。

那么这里解析 volfile 文件的代码又是在哪里呢？这是在 libglusterfs/src/graph.ch 和 graph.y 里面，这里 graph.c 中有一个函数 glusterfs_process_svc_attach_volfp，该函数会调用 glusterfs_graph_construct，这就是和构造 volfile 文件有关，另外这里可能不同版本之间的函数名称会有区别，阅读代码的时候请留意。

在 graph.y 文件中有 token 结构，这里就是和定义 volfile 文件结构有关的，同时里面也有 volume_type, new_volume 和 volume_option 等函数，这些也是相关的一些函数，感兴趣的可以自行详细了解。

还有在 volfile 中，如前面的 volume 有 type 有 protocol/server，这个文件又是在哪里的呢？可以简单找一下。

```
1. [root@gfs03 ~]# ls /usr/lib64/glusterfs/9.2/xlator/protocol/
2. client.so  server.so
```

请注意这里不同的操作系统版本，可能存放的文件位置不同。

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 *name, 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_hashsize);
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 *name, 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,
78.                 0,
79.                 LG_MSG_DENTRY_CREATE_FAI
80.                 LED,
81.                 "dentry create failed on
82.                 ",
83.                 "inode %s with parent %s",
84.                 uuid_utoa_r(link_inode->
85.                 gfid, link_uuid_str),
86.                 uuid_utoa_r(parent->gfid,
87.                 parent_uuid_str));
88.             errno = ENOMEM;
89.             return NULL;
90.         }
91.
92.         dentry->parent = __inode_ref(parent, false);
93.
94.         list_add(&dentry->inode_list, &link_inode->de
95.         ntry_list);
96.
97.         if (old_inode && __is_dentry_cyclic(dentry))
98.         {
99.             errno = ELOOP;
100.            dentry_destroy(__dentry_unset(dentry));
101.            return NULL;
102.        }
103.        __dentry_hash(dentry, dhash);
104.
105.        if (old_dentry)
106.            dentry_destroy(__dentry_unset(old_dentry));
107.        ;
108.    }
109.
110.    return link_inode;
111.}

```

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->invalidate);
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 prune

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_list, 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 __inode_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 `invalidation` 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.					
2.	1	0	0	unref	-> refs = 0, active--->destroy
3.	1	1	0	unref	-> refs = 0, active--->lru
4.	1	1	0	forget	-> nlookups = 0, active--->active
5.	0	1	0	ref	-> refs = 1, lru--->active
6.	0	1	1	ref	-> refs = 1, inv_sent = 0, invalidate--->active
7.	0	1	0	overflow	-> refs = 1, inv_sent = 1, lru--->invalidate
8.	1	1	1	unref	-> refs = 0, invalidate--->invalidate
9.	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 `purge` 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。

3.5. 内存跟踪调用

前面提到了关于 inode 和一些接口的事情,那么这一节主要想了解一下关于内存相关的,因为内存泄露的问题,在系统开发中会时不时地出现,尤其是因为 glusterfs 使用 C 语言进行开发的,并没有 java 和 python 这些语言天然自带的内存回收管理机制。

3.5.1. mem_acct

在前面介绍 xlator 的内容中,提到了关于_xlator 结构,这个就是每个功能模块的数据结构定义,在这个结构里面,有一个和该模块的内存相关的变量,如下所示。

```
1. struct _xlator {
2.     ...
3.     int32_t (*mem_acct_init)(xlator_t *this);
4.     struct mem_acct *mem_acct;
5.     ...
6. }
```

在这个结构里面的 mem_acc_init 是一个函数,在 xlator 初始化的时候和调用的时候,会记录该 xlator 的一些内存相关的信息,那么这里还有一个相关的数据结构,就是 men_acct 了,这个结构定义在 mem-pool.h 文件中,这里的定义如下所示。

```
1. struct mem_acct_rec {
2.     const char *typestr;
3.     uint64_t size;
4.     uint64_t max_size;
5.     uint64_t total_allocs;
6.     uint32_t num_allocs;
7.     uint32_t max_num_allocs;
```

```

8.     gf_lock_t lock;
9.     ...
10. };
11.
12. struct mem_acct {
13.     uint32_t num_types;
14.     gf_atomic_t refcnt;
15.     struct mem_acct_rec rec[0];
16. };

```

从上面的内容中可以看到，这里记录了总的内存容量，最大可申请的容量等变量，那么下面就来简单了解一下，每个 xlator 初始化的时候的函数调用。

3.5.2. xlator_init

每一个 xlator 在初始化的时候，需要调用 xlator_init 函数，这个函数定义在 xlator.c 中，在这个函数中，就会对该 xlator 的内存做一些初始化的工作，下面来简单了解一下。

```

1. int
2. xlator_init(xlator_t *xl)
3. {
4.     ...
5.     if (xl->mem_acct_init)
6.         xl->mem_acct_init(xl);
7.     ...
8. }

```

这里就是会调用 xlator 的 mem_acct_init 函数，那么这里和前面提到的 xlator_api 中的 mem_acct_init 就对应上了，这里会关联每个 xlator 的对应的函数，例如对于 afr 来说，这里就是 afr.c 中的 mem_acct_init 函数了，如下所示。

```

1. int32_t

```

```

2. mem_acct_init(xlator_t *this)
3. {
4.     int ret = -1;
5.
6.     if (!this)
7.         return ret;
8.
9.     ret = xlator_mem_acct_init(this, gf_afr_mt_end + 1);
10.
11.    if (ret != 0) {
12.        return ret;
13.    }
14.
15.    return ret;
16. }

```

那么这里调用的 `xlator_mem_acct_init` 中的第二个参数 `gf_afr_mt_end` 又是什么呢？这个是每个 `xlator` 定义的 `mem types`，因为对于不同的模块来说，需要关注的内存的类型是不同的，例如 `afr` 模块，这里需要考虑定义不同的内存类型的，这里的定义在 `afr-mem-types.h` 文件中，那么接着这里进入到 `xlator_mem_acct_init` 函数中查看一下实现。

```

1. int
2. xlator_mem_acct_init(xlator_t *xl, int num_types)
3. {
4.     int i = 0;
5.     int ret = 0;
6.     ...
7.     xl->mem_acct = MALLOC(sizeof(struct mem_acct) +
8.                             sizeof(struct mem_acct_rec) * n
9.                             um_types);
10.    ...
11.    xl->mem_acct->num_types = num_types;
12.    GF_ATOMIC_INIT(xl->mem_acct->refcnt, 1);
13.
14.    for (i = 0; i < num_types; i++) {

```

```

15.         memset(&xl->mem_acct->rec[i], 0, sizeof(struct me
    m_acct_rec));
16.         ret = LOCK_INIT(&(xl->mem_acct->rec[i].lock));
17.         if (ret) {
18.             fprintf(stderr, "Unable to lock..errno : %d",
                errno);
19.         }
20. #ifdef DEBUG
21.         INIT_LIST_HEAD(&(xl->mem_acct->rec[i].obj_list));
22. #endif
23.     }
24.
25.     return 0;
26. }

```

那么这里会做一些安全检查，并且调用 MALLOC,这个是一个宏定义，最后还是会调用 malloc 系统函数，然后申请一些内容，接着就是会进行 memset,也就是是一些简单的信息设置。

那么做完了这些，对于一个 xlator 来说，内存初始化部分基本算完成了，而在系统运行中，这里要申请内存的话，就要使用到另外一个东西，叫做 GF_CALLOC,在下一节中将会讲解。

3.5.3. GF_CALLOC

这里 GF_CALLOC 是一个宏定义，这里的定义在 mem-pool.h 文件中，定义如下所示。

```

1.  #define GF_CALLOC(nmemb, size, type) __gf_malloc(nmemb, s
    ize, type, #type)
2.
3.  #define GF_MALLOC(size, type) __gf_malloc(size, type, #ty
    pe)
4.
5.  #define GF_REALLOC(ptr, size) __gf_realloc(ptr, size)

```

```
6.  
7.  #define GF_FREE(free_ptr) __gf_free(free_ptr)
```

这里 MALLOC 和 CALLOC 其实是类似的，只是这里的参数不一样，多了一个 nmemb,这个 是调用系统函数 calloc 的时候的参数，表示元素的个数的。

```
1.  server = GF_CALLOC(1, sizeof(server_cmdline_t),gf_common_  
    mt_server_cmdline_t);
```

那么这里调用的方式，随便中代码中找到的一个，从这里可以看到，就是传入了想要创建申请内存的元素个数为 1，然后大为 sizeof 判断的，最后的就是 type，也就是申请的类型。

那么这里 GF_CALLOC 这里的定义的一个函数 __gf_malloc，那么这里就可以看看该函数的实现。

```
1.  void *  
2.  __gf_malloc(size_t nmemb, size_t size, uint32_t type, const  
    char *typestr)  
3.  {  
4.      size_t tot_size = 0;  
5.      size_t req_size = 0;  
6.      void *ptr = NULL;  
7.      xlator_t *xl = NULL;  
8.  
9.      if (!gf_mem_acct_enabled())  
10.         return CALLOC(nmemb, size);  
11.  
12.     xl = THIS;  
13.  
14.     req_size = nmemb * size;  
15.     tot_size = req_size + GF_MEM_HEADER_SIZE + GF_MEM_TRA  
        ILER_SIZE;  
16.  
17.     ptr = calloc(1, tot_size);  
18.  
19.     if (!ptr) {  
20.         gf_msg_nomem("", GF_LOG_ALERT, tot_size);  
21.         return NULL;
```

```

22.     }
23.
24.     return gf_mem_set_acct_info(xl->mem_acct, ptr, req_si
    ze, type, typestr);
25. }

```

那么这里可以看到，最终就是调用了系统函数 `calloc` 进行申请内存，而 `return` 之前，还会调用 `gf_mem_set_acct_info` 函数，这个函数就是用来设置一下内存相关的一些信息的，这里就和前面提到的 `mem_acct` 的结构这些有关联了。

```

1.  static void *
2.  gf_mem_set_acct_info(struct mem_acct *mem_acct, struct me
    m_header *header,
3.                      size_t size, uint32_t type, const ch
    ar *typestr)
4.  {
5.      struct mem_acct_rec *rec = NULL;
6.      bool new_ref = false;
7.
8.      if (mem_acct != NULL) {
9.          GF_ASSERT(type <= mem_acct->num_types);
10.
11.          rec = &mem_acct->rec[type];
12.          LOCK(&rec->lock);
13.          {
14.              if (!rec->typestr) {
15.                  rec->typestr = typestr;
16.              }
17.              rec->size += size;
18.              new_ref = (rec->num_allocs == 0);
19.              rec->num_allocs++;
20.              rec->total_allocs++;
21.              rec->max_size = max(rec->max_size, rec->size);
22.
23.              rec->max_num_allocs = max(rec->max_num_allocs,
    rec->num_allocs);
24.              ..
25.          }

```

```

26.         UNLOCK(&rec->lock);
27.
28.         ...
29.     }
30.
31.     header->type = type;
32.     header->mem_acct = mem_acct;
33.     header->magic = GF_MEM_HEADER_MAGIC;
34.
35.     return gf_mem_header_prepare(header, size);
36. }

```

这里可以看到，每次申请了内存之后，num_allocs 和 total_allocs 这些都会自增，另外这里还可以看到有另外一个结构，叫做 header，这个结构是哪里来的呢，就是调用了系统函数 calloc 之后返回值 ptr，而 ptr 则是一个指针，这个指针指向的是分配内存的地址，而在释放内存的时候，也需要用到这个 ptr 的，否则是无法找到之前申请的内存的，而 glusterfs 中则是使用了 mem_header 这个结构来进行记录。另外处理完成了这个函数之后，可以看到最后还会调用 gf_mem_header_prepare 这个函数。

3.5.4. GF_FREE

那么这里有申请内存的，就肯定会有释放内存，而释放内存就是 GF_FREE，这里的定义比较简单，另外这里实际调用的函数是 __gf_free，下面来简单了解一下。

```

1. void
2. __gf_free(void *free_ptr)
3. {
4.     void *ptr = NULL;
5.     struct mem_acct *mem_acct;
6.     struct mem_header *header = NULL;
7.     bool last_ref = false;
8.

```



```

9.      if (!gf_mem_acct_enabled()) {
10.          FREE(free_ptr);
11.          return;
12.      }
13.
14.      if (!free_ptr)
15.          return;
16.
17.      ...
18.
19.      LOCK(&mem_acct->rec[header->type].lock);
20.      {
21.          mem_acct->rec[header->type].size -= header->size;
22.
23.          mem_acct->rec[header->type].num_allocs--;
24.
25.          if (!mem_acct->rec[header->type].num_allocs) {
26.              last_ref = true;
27.              mem_acct->rec[header->type].typestr = NULL;
28.          }
29.      }
30.      #ifdef DEBUG
31.          list_del(&header->acct_list);
32.      #endif
33.
34.      UNLOCK(&mem_acct->rec[header->type].lock);
35.
36.      if (last_ref) {
37.          xlator_mem_acct_unref(mem_acct);
38.      }
39.
40.      free:
41.      #ifdef DEBUG
42.          __gf_mem_invalidate(ptr);
43.      #endif
44.      FREE(ptr);
45.  }

```

那么这里释放的函数是 FREE, 并且可以和前面申请时的函数 gf_mem_set_acct_info 进行对比, 这里可以看到就是做相反的操作, 把记录的申请数减少 1 等。

3.5.5. statedump

提到了可以内存相关的东西，那么这里接着就不得不讲到 statedump 了，这个是 glusterfs 提供的一个命令，就是用来获取当前 volume 的内存状态情况的，对于内存是否存在的泄露等提供非常大的帮助，那么下面先简单来感受一下这个命令的作用。

```
1. root@gfs01:~# gluster --print-statedumpdir
2. /var/run/gluster
```

这里首先要知道 statedump 的报告输出目录在哪里，然后这里有两种方式获取 volume 的 statedump 状态信息，一种是找到每个 volume 对应节点的 brick 的 pid，可以通过 status 命令知道，然后使用 kill -USR1 <pid> 的方式，获取当前节点的 volume brick 的 statedump，而如果想获取该 volume 的全部 brick 的，则可以使用下面的命令。

```
1. root@gfs01:~# gluster volume statedump test-disperse
2. volume statedump: success
3.
4. root@gfs01:~# ls /var/run/gluster
5. ...
6. glusterfs-test-disperse.132218.dump.1624803759
```

那么这里关于 statedump 的内容有都有哪些内容呢？下面可以来看看。

```
1. root@gfs01:~# cat /var/run/gluster/glusterfs-test-dispers
   e.132218.dump.1624803759
2. DUMP-START-TIME: 2021-06-27 14:22:39.732433 +0000
3.
4. [mallinfo]
5. mallinfo_arena=11046912
6. mallinfo_ordblks=431
```

```

7. mallinfo_smblocks=23
8. mallinfo_hblocks=8
9. mallinfo_hblkhd=2105344
10. mallinfo_usmblocks=0
11. mallinfo_fsmblocks=2080
12. mallinfo_uordblks=1596384
13. mallinfo_fordblks=9450528
14. mallinfo_keeppcost=54368
15. ...

```

首先这里的文件名称是有规范的，132218 就是当前 volume brick 对应的 pid 了，然后 mallinfo 的内容，这里是通过调用系统函数 mallinfo 获取的，这里 mallinfo_uordblks 表示进程总的分配的空间，单位是 bytes，而 mallinfo_hblocks 则表示 mmaped 区域的数量，关于这部分的参数的详细解释，其实是和 mallinfo 这个系统函数对应的，可以使用命令 `man mallinfo` 查看一下。

```

1. [global.glusterfs - Memory usage]
2. num_types=125
3.
4. [global.glusterfs - usage-type gf_common_mt_dnscache6 mem
   usage]
5. size=16
6. num_allocs=1
7. max_size=16
8. max_num_allocs=1
9. total_allocs=1
10.
11. [global.glusterfs - usage-type gf_common_mt_event_pool me
   musage]
12. size=20696
13. num_allocs=3
14. max_size=20696
15. max_num_allocs=3
16. total_allocs=3
17. ....

```

这里第二部分是关于 glusterfs 这个模块的内存申请情况，这里下面会有不

同的数据类型 data type 的内存使用情况，这里可以多留意一下 num_allocs 的数值，如果这个数值一直在增加，那么可能会存在内存泄露的情况。

```
1. [mempool]
2. -----
3. pool-name=struct saved_frame
4. xlator-name=test-disperse-quota
5. active-count=0
6. sizeof-type=88
7. padded-sizeof=128
8. size=0
9. shared-pool=0x7f7c127c4060
10. -----
11. ...
12. pool-name=inode_t
13. xlator-name=test-disperse-server
14. active-count=2
15. sizeof-type=168
16. padded-sizeof=256
17. size=512
18. shared-pool=0x7f7c127c4088
19. ...
```

第三部分就是关于内存池的，内存池是一种旨在减少数据类型分配数量的优化。通过为一个数据类型创建一个包含 1024 个元素的内存池，只有当池中的 1024 个元素都处于活动状态时，才会使用 calloc 之类的系统调用从堆中分配该类型的新元素。这里的作用其实和大家常见的线程池是类似的。

那么这里可以看到每一个 pool-name 都是一部分，像 inode_t 这里，下面的 active-count 不为 0，就是表示目前有 2 个正在活跃或者使用的 inode_t 对象。

另外这里还有其他很多不同的内存信息，如 iobuf, fuse operation 等，具体地可以自行阅读，根据需要来查找排查问题等。

3.5.6. io thread

这里提到了内存相关的，那么 io thread,也就是 io 线程的问题也是需要关注的，这里涉及到 fop 操作的优先级，还有线程池等问题，这些都是值得关注的，下面来简单了解一下。

```
1. struct iot_conf {
2.     pthread_mutex_t mutex;
3.     pthread_cond_t cond;
4.
5.     //最大的线程池可运行线程数量
6.     int32_t max_count;
7.     //现在实际正在运行的线程的数量
8.     int32_t curr_count;
9.     int32_t sleep_count;
10.
11.     ...
12.     //线程池的一个等待时间,如果超过这个时间没有 fop 进行处理，线程
        池会保持最小线程数运行
13.     int32_t idle_time;
14.     ...
15.
16.     //是否开启 least-priority,这个参数的作用是区分客户端内部的 fop
        优先级这些的
17.     gf_boolean_t least_priority;
18.     ...
19.};
```

对于 io 线程，这里首先需要简单了解一下 iot_conf 数据结构，这个结构定义的就是线程的 conf,也包括线程池的一些配置等等，在初始化的时候，这里需把一些参数进行设置然后运行。下面来简单了解一下 io thread 初始化的函

数。

```
1. int
2. init(xlator_t *this)
3. {
4.     iot_conf_t *conf = NULL;
5.     int ret = -1;
6.     int i = 0;
7.
8.     //下面进行一些安全检查
9.     if (!this->children || this->children->next) {
10.         gf_msg("io-threads", GF_LOG_ERROR, 0,
11.             IO_THREADS_MSG_XLATOR_CHILD_MISCONFIGURED,
12.             NULL);
13.         goto out;
14.     }
15.
16.     //下面对一些参数配置进行初始化设置
17.     GF_OPTION_INIT("thread-count", conf->max_count, int32, out);
18.     ...
19.
20.     if (!this->pass_through) {
21.         //这里就是对线程池进行调整线程,对于初始化,就是创建线程
22.         ret = iot_workers_scale(conf);
23.         ...
24.     }
25.
26.     this->private = conf;
27.
28.     conf->watchdog_secs = 0;
29.     GF_OPTION_INIT("watchdog-secs", conf->watchdog_secs, int32, out);
30.     if (conf->watchdog_secs > 0) {
31.         start_iot_watchdog(this);
32.     }
33.
34.     ret = 0;
35.out:
36.     if (ret)
```

```

37.     GF_FREE(conf);
38.
39.     return ret;
40. }

```

上面的代码是在 `xlator/performance/io-threads/io-threads.c` 这个文件中,这里主要就是初始化设置一些参数,然后就进行创建一些线程了。

那么每一个 `xlator` 都会有对应的 `xlator_api` 结构,在这里有一个 `fops` 的参数,对应的就是当前 `xlator` 的 `xlator_fops`, 对于 `io threads` 模块来说,这里有很多的 `fop`,其中包括 `iot_open`,`iot_create`,`iot_writev` 等,而观察这些 `fop` 的实现,可以知道这里大部分调用的都是 `IOT_FOP`, 如下所示。

```

1. int
2. iot_open(call_frame_t *frame, xlator_t *this, loc_t *loc, int32_t
   flags,
3.         fd_t *fd, dict_t *xdata)
4. {
5.     IOT_FOP(open, frame, this, loc, flags, fd, xdata);
6.     return 0;
7. }
8.
9. int
10. iot_create(call_frame_t *frame, xlator_t *this, loc_t *loc, int32_t
   flags,
11.            mode_t mode, mode_t umask, fd_t *fd, dict_t *xdata)
12. {
13.     IOT_FOP(create, frame, this, loc, flags, mode, umask, fd, xdata);
14.     return 0;
15. }

```

这里调用的时候可以看到, `IOT_FOP` 的第一个参数是用来区分不同的 `fop` 的,下面再进入其中了解一下。

```

1. #define
   IOT_FOP(name,frame,this,args...)

```

```

2. do{
3.   call_stub_t *__stub = NULL;
4.   int __ret = -1;
5.
6.   __stub = fop_##name##_stub(frame, default_##name##_r
       esume, args);
7.   if(!__stub){
8.       __ret = -ENOMEM;
9.       goto out;
10.  }
11.
12.  __ret = iot_schedule(frame, this, __stub);
13.
14.out:
15.  if (__ret < 0) {
16.      default_##name##_failure_cbk(frame, -__ret);
17.      if(__stub!=NULL){
18.          call_stub_destroy(__stub);
19.      }
20.  }
21.} while (0)

```

这里有一个 `fop_##name##_stub` 的东西，这个就是根据传入的 `fop` 进行动态替换的，如传入的 `name` 是 `writev`，那么这里调用的函数就是 `fop_writev_stub`，这个函数的定义在 `libglusterfs/src/call-stub.c` 中定义。另外 `stub` 又是什么呢？这个可以理解为封装的一个数据结构，封装了一些 `fop` 的，感兴趣的可以查看 `call-stub.h` 文件中的定义。

那么这里回到 `IOT_FOP` 中，这里下面就是进入到 `iot_schedule` 这个函数中了，这个函数主要是弄一些 `fop` 的优先级的，因为对于不同的 `fop` 操作，优先级是有区别的，设置优先级的原因是，对于一些客户端的查询响应希望能够快一点，那样能提高客户端的体验。

在 `iot_schedule` 函数中，这里后面会执行 `do_iot_schedule`，从这个

函数开始，就是处理和线程有关的事情了，下面进入简单了解一下。

```
1. int
2. do_iot_schedule(iot_conf_t *conf, call_stub_t *stub, int pri)
3. {
4.     int ret = 0;
5.
6.     pthread_mutex_lock(&conf->mutex);
7.     {
8.         __iot_enqueue(conf, stub, pri);
9.
10.        pthread_cond_signal(&conf->cond);
11.
12.        ret = __iot_workers_scale(conf);
13.    }
14.    pthread_mutex_unlock(&conf->mutex);
15.
16.    return ret;
17.}
```

这里的__iot_enqueue 函数就是通过配置 conf,还有封装的 stub 和优先级 pri 来设置队列情况了，这里每个 xlator 模块都有一个队列的，然后会把任务根据优先级插入到队列中。接着下面的__iot_workers_scale 就根据配置和实际的任务情况，来调整线程池的线程数量了。下面继续进入到__iot_workers_scale 中了解一下。

```
1. int
2. __iot_workers_scale(iot_conf_t *conf)
3. {
4.     int scale = 0;
5.     int diff = 0;
6.     pthread_t thread;
7.     int ret = 0;
8.     int i = 0;
9.
10.    for (i = 0; i < GF_FOP_PRI_MAX; i++)
11.        scale += min(conf->queue_sizes[i], conf->ac_iot_limit[i]);
```

```

12.
13.  if (scale < IOT_MIN_THREADS)
14.      scale = IOT_MIN_THREADS;
15.
16.  if (scale > conf->max_count)
17.      scale = conf->max_count;
18.
19.  if (conf->curr_count < scale) {
20.      diff = scale - conf->curr_count;
21.  }
22.
23.  while (diff) {
24.      diff--;
25.
26.      ret = gf_thread_create(&thread, &conf->w_attr, iot_worker,
27.                             conf,
28.                             "iotwr%03hx", conf->curr_count & 0x3ff);
29.      if (ret == 0) {
30.          pthread_detach(thread);
31.          conf->curr_count++;
32.          gf_msg_debug(conf->this->name, 0,
33.                       "scaled threads to %d (queue_size=%d/%d)",
34.                       conf->curr_count, conf->queue_size, scale);
35.      } else {
36.          break;
37.      }
38.
39.  return diff;
40. }

```

这里可以看到首先是判断要调整的线程数据和不同，然后进入 while 循环进行调整，在这里调用的是 gf_thread_create 函数，这个函数里面有一个参数 iot_worker, iot_worker 是每个 io-thread 执行的函数。这是一个线程池实现。大致是这样的工作：

1. 每个线程从第一个客户端的优先级队列的 fop 列表中取出一个 fop。

2. 在使 fop 出列后，下一个客户端会到达客户端列表的头部，以便下一个线程将从该客户端队列中选择 fop。
3. 它执行 fop 并转到第 1 步
4. 如果在任何时候队列中没有 fops，则线程等待 idle_time（默认配置为 120 秒），如果在此期间没有任何操作，只要最小线程数仍然存在，线程就会终止跑步。
5. 最低优先级用于内部客户端，如自我修复守护程序和重新平衡。只有当没有来自用户的 I/O 时，才会接收来自这些内部客户端的 I/O。

所以从这里就不难理解，为什么有时候设置关于 threads 的参数，例如 config.client-threads 和 config.brick-threads 这些参数，并没有看到明显的效果，因为这里可能当前的 IO 任务不多的时候，并不会扩大线程池的情况。

3.6. index 模块

在 glusterfs 中，index 也是一个很重要的模块，对于一个文件要进行修改或者更新的时候，因为 glusterfs 的无中心架构，因此并没有一个元数据中心来记录当前哪个文件被更改，而一旦出现异常需要进行处理，对异常文件的寻找和修复，那么就非常麻烦了。因此在很早以前，glusterfs 关于文件异常修复，需要扫描全部文件，那样的做法是不可行的，而为了解决这个问题，就引入了 heal 的概念，也就是自愈，而想要实现该功能，index 就是一个非常重要的内容了。

3.6.1. dirty 目录

为了了解这方面的内容，先简单做个试验感受一下。

```
1. root@gfs01:~# gluster volume info test-index
2.
3. Volume Name: test-index
4. Type: Replicate
5. Volume ID: 58675259-84eb-4e85-8a46-9d9a9bfb48f6
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-index
12. Brick2: 10.0.12.9:/glusterfs/test-index
13. Brick3: 10.0.12.12:/glusterfs/test-index
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
20.
21.
```

```

22.
23. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indices/
24. /glusterfs/test-index/.glusterfs/indices/
25. |— dirty
26. |— entry-changes
27. |— xattrop

```

在创建 volume 之后，这里会有一个隐藏目录 .glusterfs，在前面讲 gfid 的时候就提到了，那么该目录下还有一个特殊的重要目录 indices，这下面有三个目录，暂时都是没有内容的，那么这里到底有什么作用呢？下面继续测试一下。

```

1. root@gfs01:~# df -h /mnt/test-index/
2. Filesystem      Size  Used Avail Use% Mounted on
3. 10.0.12.2:test-index 40G  8.1G   30G  22% /mnt/test-index
4.
5.
6. root@gfs01:~# dd if=/dev/zero of=/mnt/test-index/1.txt bs=1k count=100000
7. 100000+0 records in
8. 100000+0 records out
9. 102400000 bytes (102 MB, 98 MiB) copied, 14.1931 s, 7.2 MB/s
10.
11. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indices/
12. /glusterfs/test-index/.glusterfs/indices/
13. |— dirty
14. |   |— a9f2710b-ea25-471b-bb3b-bd799e76fa1f
15. |   |— dirty-495a4ce0-8117-4bac-992e-5c9bb69260b9
16. |— entry-changes
17. |— xattrop
18.
19. 3 directories, 2 files

```

这里挂载 volume 之后，然后使用 dd 命令创建文件，这里接着在 brick 目录下查看该隐藏文件，可以看到 dirty 目录下这里多出来了两个文件，其中一个文件是没有带上前缀 dirty 的，那么这个文件名这一串其实是什么呢？这个其实就

是 brick 目录下实际文件的 gfid 编号，而且这两个文件的 inode 是一样的。

```
1. root@gfs02:~# ls -i /glusterfs/test-index/.glusterfs/indices/dirty/a9f2710b-ea25-471b-bb3b-bd799e76fa1f
2. 2101879 /glusterfs/test-index/.glusterfs/indices/dirty/a9f2710b-ea25-471b-bb3b-bd799e76fa1f
3. root@gfs02:~# ls -i /glusterfs/test-index/1.txt
4. 2101878 /glusterfs/test-index/1.txt
5.
6. root@gfs02:~# getfattr -d -m . -e hex /glusterfs/test-index/1.txt
7. getfattr: Removing leading '/' from absolute path names
8. # file: glusterfs/test-index/1.txt
9. trusted.afr.dirty=0x000000010000000000000000
10. trusted.gfid=0xa9f2710bea25471bbb3bbd799e76fa1f
11. trusted.gfid2path.46104af11fea85fe=0x30303030303030302d303030302d30303030302d30303030303030303030312f312e747874
12. trusted.glusterfs.mdata=0x01000000000000000000000000000060e02244000000030272144000000060e022440000000302721440000000060e01d4a00000000952863e
```

可以看到这里其实就是一个硬链接了，然后当写入完成的时候，该文件也会被删除的，就只剩下以 dirty 为前缀开头的文件了。

所以从这个简单的小实验中可以看到，dirty 这个文件就是用来记录正在改变的文件的。

3.6.2. xattrop 目录

如果这里在创建过程中出现异常呢？例如 brick 进程被杀死了，这里会有怎样的表现呢？下面继续来实验一下。另外这里为了测试的准确性，再次重新创建了该 volume，因此会和上面的实验部分内容有点不一样，但是不影响测试情况。

```
1. root@gfs02:~# ps -ef |grep 2617591
```

```

2. root      2617591      1  0 16:59 ?          00:00:00 /usr/
   sbin/glusterfsd -s 10.0.12.2 --volfile-id test-index.10.0.
   12.2.glusterfs-test-index -p /var/run/gluster/vols/test-in
   dex/10.0.12.2-glusterfs-test-index.pid -S /var/run/gluster
   /21ffe8c1aae2d06c.socket --brick-name /glusterfs/test-inde
   x -l /var/log/glusterfs/bricks/glusterfs-test-index.log --
   xlator-option *-posix.glusterd-uuid=69da12d3-7d23-408b-add
   3-e20e60ae725f --process-name brick --brick-port 49164 --x
   lator-option test-index-server.listen-port=49164
3.
4.
5. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indi
   ces/
6. /glusterfs/test-index/.glusterfs/indices/
7. |— dirty
8. |   |— dirty-1bf6eb50-1087-42a2-a430-a04b4c9efda8
9. |   |— e9759499-e824-4a99-8265-a63894c386a2
10. |— entry-changes
11. |— xattrop
12.
13. 3 directories, 2 files
14. root@gfs02:~# kill -9 2617591
15. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indi
   ces/
16. /glusterfs/test-index/.glusterfs/indices/
17. |— dirty
18. |   |— dirty-1bf6eb50-1087-42a2-a430-a04b4c9efda8
19. |   |— e9759499-e824-4a99-8265-a63894c386a2
20. |— entry-changes
21. |— xattrop
22.
23. 3 directories, 2 files

```

这里首先找到其中一个 brick 的 pid ,然后在创建文件的过程中杀掉该进程 ,接着再查看 indices 的目录结构 ,等到任务完成的时候 ,这里再次查看其它健康的 brick 的该目录情况。

```

1. root@gfs03:~# tree /glusterfs/test-index/.glusterfs/indi
   ces/
2. /glusterfs/test-index/.glusterfs/indices/

```

```

3.  |— dirty
4.  |   └─ dirty-c9493c55-3b96-4b13-a93f-dc05409ee34e
5.  |— entry-changes
6.  |— xattrop
7.  |   └─ e9759499-e824-4a99-8265-a63894c386a2
8.  |   └─ xattrop-c9493c55-3b96-4b13-a93f-dc05409ee34e
9.
10. 3 directories, 3 files
11.
12. root@gfs01:~# tree /glusterfs/test-index/.glusterfs/indices/
13. /glusterfs/test-index/.glusterfs/indices/
14. |— dirty
15. |   └─ dirty-027d5224-8211-483e-8bf1-8b088222f6f8
16. |— entry-changes
17. |— xattrop
18. |   └─ e9759499-e824-4a99-8265-a63894c386a2
19. |   └─ xattrop-027d5224-8211-483e-8bf1-8b088222f6f8
20.
21. 3 directories, 3 files

```

从这里可以看到，其它的两个节点，在 dirty 目录下面是没有了以 gfid 开头的文件，但是在 xattrop 中是存在该文件的，也就是说，这里当这里有 brick 没有完成请求的时候，需要对文件进行异常修复处理时，那么这里就会在 xattrop 目录下保留该文件了，这样方便记录知道哪些文件需要进行修复自愈。

```

1. root@gfs01:~# gluster volume heal test-index info summary
2. Brick 10.0.12.2:/glusterfs/test-index
3. Status: Transport endpoint is not connected
4. Total Number of entries: -
5. Number of entries in heal pending: -
6. Number of entries in split-brain: -
7. Number of entries possibly healing: -
8.
9. Brick 10.0.12.9:/glusterfs/test-index
10. Status: Connected
11. Total Number of entries: 1
12. Number of entries in heal pending: 1
13. Number of entries in split-brain: 0

```



```

14. Number of entries possibly healing: 0
15.
16. Brick 10.0.12.12:/glusterfs/test-index
17. Status: Connected
18. Total Number of entries: 1
19. Number of entries in heal pending: 1
20. Number of entries in split-brain: 0
21. Number of entries possibly healing: 0
22.
23. root@gfs01:~# gluster volume heal test-index info
24. Brick 10.0.12.2:/glusterfs/test-index
25. Status: Transport endpoint is not connected
26. Number of entries: -
27.
28. Brick 10.0.12.9:/glusterfs/test-index
29. /1.txt
30. Status: Connected
31. Number of entries: 1
32.
33. Brick 10.0.12.12:/glusterfs/test-index
34. /1.txt
35. Status: Connected
36. Number of entries: 1

```

这里也可以从这里知道一些具体的信息情况,heal info 中显示了当前有一个 brick 是无法连接上的,这里是需要留意的。那么这里要怎么解决这个问题呢? 可以再次重启 brick 的进程即可,这里使用命令 start force 即可,然后启动之后,可以观察一下异常 brick 所在节点的 heal 日志,也就是 glustershd.log 日志,下面简单查看一下日志的情况。

```

1. root@gfs02:~# cat /var/log/glusterfs/glustershd.log
2. ...
3. //这里是 brick 进程被杀死时的日志
4. [2021-07-03 09:28:49.541314 +0000] D [MSGID: 0] [client.c:
   2231:client_rpc_notify] 16-test-index-client-0: disconnect
   ed from test-index-client-0. Client process will keep tryi
   ng to connect to glusterd until brick's port is available

```

5. [2021-07-03 09:28:51.550508 +0000] I [MSGID: 100041] [glusterfsd-mgmt.c:1034:glusterfs_handle_svc_attach] 0-glusterfs: received attach request for volfile [{volfile-id=shd/test-index}]
6. [2021-07-03 09:28:51.550550 +0000] I [MSGID: 100040] [glusterfsd-mgmt.c:108:mgmt_process_volfile] 0-glusterfs: No change in volfile, countinuing []
7. ...
- 8.
- 9.
10. //下面是强制启动 brick 进程，重新连接的日志
11. [2021-07-03 09:28:52.544202 +0000] I [MSGID: 114046] [client-handshake.c:855:client_setvolume_cbk] 16-test-index-client-0: Connected, attached to remote volume [{conn-name=test-index-client-0}, {remote_subvol=/glusterfs/test-index}]
12. [2021-07-03 09:28:52.544216 +0000] D [MSGID: 0] [client-handshake.c:675:client_post_handshake] 16-test-index-client-0: No fds to open - notifying all parents child up
13. [2021-07-03 09:28:52.544230 +0000] D [MSGID: 0] [afr-common.c:5977:afr_get_halo_latency] 16-test-index-replicate-0: Using halo latency 99999
14. [2021-07-03 09:28:52.544494 +0000] D [rpc-clnt-ping.c:93:rpc_clnt_remove_ping_timer_locked] (--> /lib/x86_64-linux-gnu/libglusterfs.so.0(_gf_log_callingfn+0x199)[0x7f45431b3209] (--> /lib/x86_64-linux-gnu/libgfrpc.so.0(+0x13bd2)[0x7f454315ebd2] (--> /lib/x86_64-linux-gnu/libgfrpc.so.0(+0x14401)[0x7f454315f401] (--> /lib/x86_64-linux-gnu/libgfrpc.so.0(rpc_clnt_submit+0x30f)[0x7f454315b5df] (--> /usr/lib/x86_64-linux-gnu/glusterfs/9.2/xlator/protocol/client.so(+0x16396)[0x7f453d9c7396]))))) 16-: 10.0.12.12:49163: ping timer event already removed
15. ...
16. //这里开始进行 heal 的信息获取
17. [2021-07-03 09:28:52.544964 +0000] D [MSGID: 0] [client-rpc-fops_v2.c:916:client4_0_getxattr_cbk] 16-test-index-client-2: resetting op_ret to 0 from 59
18. [2021-07-03 09:28:52.544957 +0000] D [rpc-clnt-ping.c:194:rpc_clnt_ping_cbk] 16-test-index-client-2: Ping latency is 0ms

```

19. [2021-07-03 09:28:52.545098 +0000] D [MSGID: 0] [syncop-
    tils.c:538:syncop_is_subvol_local] 16-test-index-client-2:
    subvol test-index-client-2 is not local
20. [2021-07-03 09:28:52.545193 +0000] D [MSGID: 0] [client-r
    pc-fops_v2.c:916:client4_0_getxattr_cbk] 16-test-index-cli
    ent-1: resetting op_ret to 0 from 59
21. [2021-07-03 09:28:52.545262 +0000] D [MSGID: 0] [syncop-
    tils.c:538:syncop_is_subvol_local] 16-test-index-client-1:
    subvol test-index-client-1 is not local
22. [2021-07-03 09:28:52.545610 +0000] D [rpc-clnt-ping.c:194:
    rpc_clnt_ping_cbk] 16-test-index-client-1: Ping latency is
    0ms
23. [2021-07-03 09:28:52.545668 +0000] D [MSGID: 0] [client-r
    pc-fops_v2.c:916:client4_0_getxattr_cbk] 16-test-index-cli
    ent-0: resetting op_ret to 0 from 59
24. [2021-07-03 09:28:52.545708 +0000] D [MSGID: 0] [syncop-
    tils.c:538:syncop_is_subvol_local] 16-test-index-client-0:
    subvol test-index-client-0 is local
25. [2021-07-03 09:28:52.545739 +0000] D [MSGID: 0] [afr-self
    -heald.c:1053:afr_shd_index_healer] 16-test-index-replicat
    e-0: starting index sweep on subvol test-index-client-0
26. ..
27. //这里开始进行自愈异常文件
28. [2021-07-03 09:28:52.547566 +0000] D [MSGID: 0] [afr-self
    -heald.c:415:afr_shd_index_heal] 16-test-index-replicate-0:
    got entry: e9759499-e824-4a99-8265-a63894c386a2 from test
    -index-client-0
29. [2021-07-03 09:28:52.547648 +0000] D [MSGID: 0] [stack.h:
    509:copy_frame] 16-stack: groups is null (ngrps: 0) [Inval
    id argument]
30. [2021-07-03 09:28:52.547978 +0000] D [MSGID: 0] [client-r
    pc-fops_v2.c:916:client4_0_getxattr_cbk] 16-test-index-cli
    ent-0: resetting op_ret to 0 from 43
31. [2021-07-03 09:28:52.548508 +0000] D [MSGID: 0] [afr-self
    -heal-common.c:2396:afr_selfheal_unlocked_inspect] 16-test
    -index-replicate-0: SIZE mismatch 77826048 vs 102400000 on
    test-index-client-1 for gfid:e9759499-e824-4a99-8265-a638
    94c386a2
32. [2021-07-03 09:28:52.548582 +0000] D [MSGID: 0] [afr-self
    -heal-common.c:2396:afr_selfheal_unlocked_inspect] 16-test
    -index-replicate-0: SIZE mismatch 77826048 vs 102400000 on
    test-index-client-2 for gfid:e9759499-e824-4a99-8265-a638
    94c386a2

```

```

33. //留意这里显示的,是 data-heal 出现问题了.
34. [2021-07-03 09:28:52.548887 +0000] D [MSGID: 0] [afr-self
    -heal-common.c:2560:afr_selfheal_do] 16-test-index-replica
    te-0: heals needed for e9759499-e824-4a99-8265-a63894c386a
    2: [entry-heal=0, metadata-heal=0, data-heal=1]
35.
36. ....
37. //这里看到文件自愈成功了
38. [2021-07-03 09:28:53.275487 +0000] D [MSGID: 0] [client-r
    pc-fops_v2.c:1536:client4_0_xattrop_cbk] 16-test-index-cli
    ent-2: resetting op_ret to 0 from 0
39. [2021-07-03 09:28:53.275860 +0000] I [MSGID: 108026] [afr
    -self-heal-common.c:1742:afr_log_selfheal] 16-test-index-r
    eplicate-0: Completed data selfheal on e9759499-e824-4a99-
    8265-a63894c386a2. sources=[1] 2 sinks=0

```

通过这里对日志的追踪分析,可以看到当重新连接之后,会获取其他正常节点的 heal 信息,其实就是 xattrop 目录的信息,确定这里要修复的异常文件。接着这里就开始进行修复了,同时这里可以留意到日志中还会判断该异常文件是出现了 data-heal 异常,也就是说这里是数据出现问题了,而不是文件的元信息出现问题了,这一点在 AFR 中也是非常重要的。那么这里自愈结束之后,也会把 xattrop 目录下的文件删除的。

3.6.3. entry-changes 目录

这里隐藏目录 indices 下面还有一个特殊的目录 entry-changes,那么这个目录的作用是什么呢?其实这个目录是用来记录要修复的文件目录的结构情况的,也就是哪个文件属于哪个目录下面,这里做一个小实验来感受一下。

```

1. root@gfs01:~# gluster volume status test-index
2. Status of volume: test-index

```

```

3. Gluster process                                TCP Port  RDM
   A Port  Online  Pid
4. -----
5. Brick 10.0.12.2:/glusterfs/test-index          49164      0
   Y      2635161
6. Brick 10.0.12.9:/glusterfs/test-index          49165      0
   Y      2776280
7. Brick 10.0.12.12:/glusterfs/test-index         49163      0
   Y      2665705
8. Self-heal Daemon on localhost                  N/A        N/A
   Y      128017
9. Self-heal Daemon on gfs02                      N/A        N/A
   Y      109807
10. Self-heal Daemon on 10.0.12.3                 N/A        N/A
   Y      15298
11. Self-heal Daemon on 10.0.12.7                 N/A        N/A
   Y      15526
12. Self-heal Daemon on 10.0.12.9                 N/A        N/A
   Y      2515251
13.
14. Task Status of Volume test-index
15. -----
16. There are no active volume tasks
17.
18. root@gfs01:~# hostname -i
19. 10.0.12.12
20.
21. root@gfs01:~# kill -9 2665705
22.
23. root@gfs01:~# cd /mnt/test-index/
24.
25. root@gfs01:/mnt/test-index# touch 1.txt
26.
27. root@gfs01:/mnt/test-index# mkdir -p a
28.
29. root@gfs01:/mnt/test-index# mkdir -p b
30.
31. root@gfs01:/mnt/test-index# touch a/a-01.txt
32.
33. root@gfs01:/mnt/test-index# touch a/a-2.txt
34.
35. root@gfs01:/mnt/test-index# touch b/b-2.txt

```

```

36.
37. root@gfs01:/mnt/test-index# touch b/b-1.txt

```

这里首先再次重新创建一个 volume，在删除旧的之前，要先把 brick 目录删掉，否则隐藏文件保留的话，是无法在原来的位置上重新创建的并且挂载。接着这里找到其中一个节点的 brick 的 pid，然后 kill 掉进入到挂载目录进行创建文件，这时候当前被 kill 掉节点下的隐藏目录是没有任何文件的，因为进程已经被 kill 掉了。

```

1. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indices/
2. /glusterfs/test-index/.glusterfs/indices/
3. |— dirty
4. |   |— dirty-db686567-3b6f-4391-b7e7-3d0c1b20fe7b
5. |— entry-changes
6. |   |— 00000000-0000-0000-0000-000000000001
7. |       |— 1.txt
8. |       |— a
9. |       |— b
10. |   |— 6eb86739-a0f7-4d20-91c8-af2030a6a712
11. |       |— b-1.txt
12. |       |— b-2.txt
13. |   |— ba93fe3a-c92d-44d6-a81e-a224fe27d1c1
14. |       |— a-01.txt
15. |       |— a-2.txt
16. |   |— entry-changes-db686567-3b6f-4391-b7e7-3d0c1b20fe7
    b
17. |— xattrop
18. |   |— 00000000-0000-0000-0000-000000000001
19. |   |— 15644cf1-83c1-4807-9b70-d43198d7abc9
20. |   |— 2022bfa3-96cd-43c2-b9b8-e28425a13cca
21. |   |— 6eb86739-a0f7-4d20-91c8-af2030a6a712
22. |   |— 8740d0ae-63c7-4d6c-a8f1-5a07490e8865
23. |   |— 91cec102-72b7-424d-a936-0bf1e2683cc3
24. |   |— ba93fe3a-c92d-44d6-a81e-a224fe27d1c1
25. |   |— ecfe29ff-34bc-47c3-a03f-303443517c5b
26. |   |— xattrop-db686567-3b6f-4391-b7e7-3d0c1b20fe7b
27.
28. 6 directories, 18 files
29.

```

```

30.
31. root@gfs03:~# tree /glusterfs/test-index/.glusterfs/indices/
32. /glusterfs/test-index/.glusterfs/indices/
33. |— dirty
34. |   └─ dirty-9f366381-bb65-4a25-8bf5-dff349508ffc
35. |— entry-changes
36. |   └─ 00000000-0000-0000-0000-000000000001
37. |       └─ 1.txt
38. |       └─ a
39. |       └─ b
40. |   └─ 6eb86739-a0f7-4d20-91c8-af2030a6a712
41. |       └─ b-1.txt
42. |       └─ b-2.txt
43. |   └─ ba93fe3a-c92d-44d6-a81e-a224fe27d1c1
44. |       └─ a-01.txt
45. |       └─ a-2.txt
46. |   └─ entry-changes-9f366381-bb65-4a25-8bf5-dff349508ffc
47. └─ c
48.     └─ xattrop
49.         └─ 00000000-0000-0000-0000-000000000001
50.         └─ 15644cf1-83c1-4807-9b70-d43198d7abc9
51.         └─ 2022bfa3-96cd-43c2-b9b8-e28425a13cca
52.         └─ 6eb86739-a0f7-4d20-91c8-af2030a6a712
53.         └─ 8740d0ae-63c7-4d6c-a8f1-5a07490e8865
54.         └─ 91cec102-72b7-424d-a936-0bf1e2683cc3
55.         └─ ba93fe3a-c92d-44d6-a81e-a224fe27d1c1
56.         └─ ecfe29ff-34bc-47c3-a03f-303443517c5b
57.         └─ xattrop-9f366381-bb65-4a25-8bf5-dff349508ffc
58. 6 directories, 18 files

```

这里通过其他两个正常的目录可以看到这里 entry-changes 目录下其实记录的就是文件和目录的结构情况，也就是要进行自愈的文件结构，这里 1.txt 是在 brick 的根目录下创建的，因此这里的上层目录 gfid 就是最后为 1，其他全为 0 的。

接着这里还是再次强制启动 brick 的进程，也就是使用命令 start force，然后这里的目录就会恢复了。

```

1. root@gfs02:~# tree /glusterfs/test-index/.glusterfs/indices/
2. /glusterfs/test-index/.glusterfs/indices/
3. |— dirty
4. |   └─ dirty-db686567-3b6f-4391-b7e7-3d0c1b20fe7b
5. |— entry-changes
6. |   └─ entry-changes-db686567-3b6f-4391-b7e7-3d0c1b20fe7b
7. └─ xattrop
8.     └─ xattrop-db686567-3b6f-4391-b7e7-3d0c1b20fe7b
9.
10. 3 directories, 3 files

```

注意如果这里 entry-changes 的目录还是没有恢复的话，那么考虑是否 heal 进程有异常情况出现了，这里可以根据日志来排查一下，或者直接对 glusterd 进程进行重启，这里 heal 进程其实是由 glusterd 创建的。

通过这三个目录的实验，那么对于 AFR 来说，有了这里的内容，才好对文件进行更多的处理。

3.6.4. heal 连接数

这里有一个地方是非常值得注意的，那就是 heal 进程与所有的 bricks 都会相连的，使用 status clients 可以看到。哪怕这个 volume 的 brick 不是当前节点的，那么也会连接到 heal 进程上面，也就是说，每一个集群节点默认都有一个 glustershd 进程，然后这个进程会和所有的 bricks 进行连接，下面来感受一下。

```

1. root@gfs01:~# gluster volume info test-shd-num
2.
3. Volume Name: test-shd-num
4. Type: Replicate
5. Volume ID: 7b40457b-846a-4ace-90a6-3143fdc11782

```



```

6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-shd-num
12. Brick2: 10.0.12.9:/glusterfs/test-shd-num
13. Brick3: 10.0.12.12:/glusterfs/test-shd-num
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
20.
21.
22.
23. root@gfs01:~# gluster volume status test-shd-num clients
24. Client connections for volume test-shd-num
25. -----
26. Brick : 10.0.12.2:/glusterfs/test-shd-num
27. Clients connected : 5
28. Hostname      BytesRead  BytesWritten  OpVersion
29. -----
30. 10.0.12.9:49075 1480        1460          90000
31. 10.0.12.2:48912 4972        5356          90000
32. 10.0.12.12:49058 1868        2264          90000
33. 10.0.12.3:49071 1252        968           90000
34. 10.0.12.7:49062 1252        968           90000
35. -----
36. Brick : 10.0.12.9:/glusterfs/test-shd-num
37. Clients connected : 5
38. Hostname      BytesRead  BytesWritten  OpVersion
39. -----
40. 10.0.12.9:49088 4732        5012          90000
41. 10.0.12.2:48937 1720        1804          90000
42. 10.0.12.12:49054 1716        1804          90000
43. 10.0.12.3:49070 1252        968           90000
44. 10.0.12.7:49061 1252        968           90000
45. -----
46. Brick : 10.0.12.12:/glusterfs/test-shd-num
47. Clients connected : 5
48. Hostname      BytesRead  BytesWritten  OpVersion
49. -----

```

50.10.0.12.12:49146	8372	9368	90000
51.10.0.12.3:49075	1252	968	90000
52.10.0.12.7:49074	1252	968	90000
53.10.0.12.9:49079	1484	1460	90000
54.10.0.12.2:48936	1720	1804	90000
55.-----			

从上面可以看出，这里 volume 的 brick 只是 2,9,12 三个节点，但是创建之后这里还有其他节点的连接，然后查看一下就可以知道是和 glustershd 的进程连接，那么这里根据官方的说法，这是一个待优化的项目，另外这里当一个节点有多个 glustershd 进程的时候，并不会增加连接数。

3.6.5. AFR

那么有了前面提到的三个目录，就可以准确知道哪些文件正在修改，并且哪些文件需要修复了，而修复和自愈，在 glusterfs 中有一个专门的算法叫做 AFR，不过在讲解 AFR 之前这里需要先了解一下对于一个写操作流程到底是怎样的。而在了解写操作，也就是涉及写入的 fop，我们需要区分一下 glusterfs 中常用的 fop，到底哪些 fop 涉及修改文件，哪些涉及读取的。

fop 类型	fop	备注
写入	afr_create afr_mknodafr_mkdir afr_link afr_symlink afr_rename afr_unlink afr_rmdir afr_do_writev afr_truncate	这里的 fop 都是会修改文件的。

	afr_ftruncate afr_setattr afr_fsetattr afr_setxattr afr_fsetxattr afr_removexattr afr_fremovexattr afr_fallocate afr_discard afr_zerofill afr_xattrop afr_fxattrop afr_fsync	
读取	afr_readdir afr_access afr_stat afr_fstat afr_readlink afr_getxattr afr_fgetxattr afr_readv afr_seek	这里的 fop 只是涉及到读取，不会涉及修改文件。

那么在知道了 fop 的一些分类之后,对于写入操作的 fop 流程是怎样的呢?

这里其实是分为五个步骤的:

- 1) 锁定阶段: 在所有 brick 上锁定正在修改的文件,以便其他客户端的 AFR 在尝试同时修改同一文件时被阻止。
- 2) 预操作阶段: 在所有参与的 brick 上将 xattr 扩展属性 (trusted.afr.dirty) 增加 1,作为即将发生的 FOP 的指示(在下一阶段),也就是记录正在操作。
- 3) fop 操作阶段: 这里 volume 的 bricks 执行 fop 操作。
- 4) 完成阶段: fop 执行完成之后,在执行成功的 brick 上第二步中的扩展属性增

加的数值减少 1 , 正常情况下 , 这时候扩展属性应该全为 0,证明文件执行成功了。

- 5) 解锁阶段: 释放在阶段 1 中获取的锁。任何竞争客户端现在都可以继续进行自己的写入事务。

那么通过这几个步骤可以知道 , 一旦文件写入异常 , 也就是在扩展属性上面是可以看到的 , 下面做一个小的测试来感受一下。

```
1. root@gfs01:/mnt/test-afr# gluster volume info test-afr
2.
3. Volume Name: test-afr
4. Type: Replicate
5. Volume ID: 6637cea5-2ead-4c3f-a319-9ac7e32d3fd1
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-afr
12. Brick2: 10.0.12.9:/glusterfs/test-afr
13. Brick3: 10.0.12.12:/glusterfs/test-afr
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
```

这里先创建一个 volume 然后挂载 , 接着创建一个文件 1.txt , 接着 kill 掉其中一个 brick 进程 , 再对 1.txt 文件使用进行修改 , 这里使用命令 `date > 1.txt` 导入时间 , 这时候对比一下正常和非正常的 brick 下的 1.txt 文件属性

```
1. //下面是正常 brick
2. root@gfs02:~# getfattr -d -m . -e hex /glusterfs/test-afr/1.txt
3. getfattr: Removing leading '/' from absolute path names
4. # file: glusterfs/test-afr/1.txt
```

```

5. trusted.afr.dirty=0x000000000000000000000000
6. trusted.afr.test-afr-client-1=0x000000020000000000000000
7. trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
8. trusted.gfid2path.46104af11fea85fe=0x30303030303030302
d303030302d303030302d303030302d3030303030303030
3030312f312e747874
9. trusted.glusterfs.mdata=0x01000000000000000000000006
0e54082000000001b967e1d0000000060e54082000000001b9
67e1d0000000060e5405b000000003b3d42a9
10.
11.root@gfs01:/mnt/test-afr# getfattr -d -m . -e hex /glusterfs/tes
t-afr/1.txt
12.getfattr: Removing leading '/' from absolute path names
13.# file: glusterfs/test-afr/1.txt
14.trusted.afr.dirty=0x000000000000000000000000
15.trusted.afr.test-afr-client-1=0x000000020000000000000000
16.trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
17.trusted.gfid2path.46104af11fea85fe=0x30303030303030302
d303030302d303030302d303030302d3030303030303030
3030312f312e747874
18.trusted.glusterfs.mdata=0x01000000000000000000000006
0e54082000000001b967e1d0000000060e54082000000001b9
67e1d0000000060e5405b000000003b3d42a9
19.
20.//这里是异常的 brick
21.root@gfs03:~# getfattr -d -m . -e hex /glusterfs/test-afr/1.txt
22.getfattr: Removing leading '/' from absolute path names
23.# file: glusterfs/test-afr/1.txt
24.trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
25.trusted.gfid2path.46104af11fea85fe=0x30303030303030302
d303030302d303030302d303030302d3030303030303030
3030312f312e747874
26.trusted.glusterfs.mdata=0x01000000000000000000000006
0e5405b000000003b3d42a90000000060e5405b000000003b3
d42a90000000060e5405b000000003b3d42a9

```

从这里可以看到，首先异常的 brick 这里没有显示扩展属性 trusted.afr.dirty，接着正常的 brick 的文件扩展属性，这里的扩展属性 trusted.afr.test-afr-client-1，这里表示该 volume 名为 test-afr 的第二个 brick 出现异常了，接着可以查看一下 heal info 信息来核实一下。

```

1. root@gfs02:~# gluster volume heal test-afr info
2. Brick 10.0.12.2:/glusterfs/test-afr
3. /1.txt
4. Status: Connected
5. Number of entries: 1
6.
7. Brick 10.0.12.9:/glusterfs/test-afr
8. Status: Transport endpoint is not connected
9. Number of entries: -
10.
11. Brick 10.0.12.12:/glusterfs/test-afr
12. /1.txt
13. Status: Connected
14. Number of entries: 1

```

这里就指出了其中一个 brick 是没有连接上的，那么这里如果强制启动的话，再使用 heal 命令则可以修复该问题了。

那么下面再来看另外一种测试。

```

1. root@gfs01:/mnt/test-afr# dd if=/dev/zero of=2.txt bs=1k count=1000000
2.
3. root@gfs03:~# gluster volume status test-afr
4. Status of volume: test-afr
5. Gluster process          TCP Port  RDMA Port  Online
   Pid
6. -----
7. Brick 10.0.12.2:/glusterfs/test-afr    49167    0          Y      3
   616131
8. Brick 10.0.12.9:/glusterfs/test-afr    49169    0          Y      3
   764494
9. Brick 10.0.12.12:/glusterfs/test-afr    49166    0          Y
   3676444
10. Self-heal Daemon on localhost          N/A      N/A        Y
   2515251
11. Self-heal Daemon on 10.0.12.3          N/A      N/A        Y
   15298
12. Self-heal Daemon on 10.0.12.7          N/A      N/A        Y
   15526

```



```

19.trusted.gfid=0x11a0da6539874e87980399ab64552333
20.trusted.gfid2path.8954a07f86db8340=0x3030303030303030
   2d303030302d303030302d303030302d3030303030303030
   03030312f322e747874
21.trusted.glusterfs.mdata=0x01000000000000000000000000000006
   0e54523000000001006ff9a0000000060e5452300000000100
   6ff9a00000000060e544e40000000034ad3322
22.
23.root@gfs02:~# du -sh /glusterfs/test-afr/2.txt
24.346M   /glusterfs/test-afr/2.txt
25.
26.
27.//下面是异常 brick 的
28.root@gfs03:~# du -sh /glusterfs/test-afr/2.txt
29.67M /glusterfs/test-afr/2.txt
30.
31.root@gfs03:~# getfattr -d -m . -e hex /glusterfs/test-afr/2.txt
32.getfattr: Removing leading '/' from absolute path names
33.# file: glusterfs/test-afr/2.txt
34.trusted.afr.dirty=0x00000001000000000000000000000000
35.trusted.gfid=0x11a0da6539874e87980399ab64552333
36.trusted.gfid2path.8954a07f86db8340=0x3030303030303030
   2d303030302d303030302d303030302d3030303030303030
   03030312f322e747874
37.trusted.glusterfs.mdata=0x01000000000000000000000000000006
   0e544f0000000000170e52b00000000060e544f000000000170
   e52b000000000060e544e40000000034ad3322

```

从这里上面的文件扩展属性可以留意到 trusted.afr.dirty 中，异常的 brick 中间有一个数字为 1 的，而正常的 brick 都是 0 的，这个也就是和上面提到的五个步骤呼应的，这里因为异常的 brick 的操作还没完成但是被异常销毁了，因此并没有执行到第四步，所以还会保留该数值的。那么接着我们再来看看 indices 目录的情况是如何的。

```

1. //下面是正常 brick
2. root@gfs03:~# tree /glusterfs/test-afr/.glusterfs/indices/
3. /glusterfs/test-afr/.glusterfs/indices/

```



```

4. |— dirty
5. |   |— 11a0da65-3987-4e87-9803-99ab64552333
6. |   |— dirty-b14af3d5-3645-4530-bf53-837807620b2f
7. |— entry-changes
8. |— xattrop
9.
10.3 directories, 2 files
11.
12.
13.root@gfs02:~# tree /glusterfs/test-afr/.glusterfs/indices/
14./glusterfs/test-afr/.glusterfs/indices/
15. |— dirty
16. |   |— dirty-87eb2ed4-e5da-48f1-8367-9b00177ff9df
17. |— entry-changes
18. |— xattrop
19. |   |— 11a0da65-3987-4e87-9803-99ab64552333
20. |   |— xattrop-87eb2ed4-e5da-48f1-8367-9b00177ff9df
21.
22.3 directories, 3 files
23.
24.
25.//下面是异常 brick 的
26.root@gfs03:~# tree /glusterfs/test-afr/.glusterfs/indices/
27./glusterfs/test-afr/.glusterfs/indices/
28. |— dirty
29. |   |— 11a0da65-3987-4e87-9803-99ab64552333
30. |   |— dirty-b14af3d5-3645-4530-bf53-837807620b2f
31. |— entry-changes
32. |— xattrop
33.
34.3 directories, 2 files

```

下面来关注了解一下 trusted.afr.dirty 这个文件扩展属性的数字规律。这里的数字是前 12 位和文件异常有关的，每 4 位作为一个分类，其中前 4 位是和数据有关的，接着 4 位是和元数据有关的，最后 4 位和 entry 目录有关，也就是说，可能在操作文件的过程中异常也是分为三种异常的，有些异常是修改了文件的属性，但是并没有修改数据，而有些是修改了数据的等等，因此这里修复。

那么这里什么时候发生自愈修复呢？一方面每个节点默认是有一个 glustershd 进程的，这个进程在一段时间内检查是否需要修复的，当然也可以触发命令进行修复，执行如下命令。

1. root@gfs02:~# gluster volume heal test-afr
2. Launching heal operation to perform index self heal on volume test-afr has been unsuccessful:
3. Glusterd Syncop Mgmt brick op 'Heal' failed. Please check glustershd log file **for** details.

那么提到了自愈的问题，脑裂也是可能出现的，所谓的脑裂，如一个 3 副本的复制卷，这里其中超过半数的文件属性不一致或者出现异常的时候，就会有脑裂出现了，而这时候，通过正常的 heal 可能无法修复问题，那么这时候使用 heal info 命令，可以看到有些文件会提示 Is in split-brain，也就是存在脑裂风险。那么这里修复的方式有多种，其中可以以最新的时间作为修复，也就是使用如下命令。

1. sudo gluster volume heal {VOLUME-NAME} split-brain latest-mtime {**FILE-NAME**}

这里最后的 FILE-NAME 就是 heal info 命令中提示存在脑裂的文件名称，然后执行可以修复，当然这里还有可以根据 bigger-file 最大的文件，source-brick 自行选择恰当的 brick 来修复等。

章节语:

这一章的内容很多,从 gfid 到数据内存模型再到 AFR 这些,涉及到 glusterfs 很多核心的概念和模块,而不同的模块作用不同,在阅读理解的时候,如果对于部分模块的细节不是很理解,可以考虑先放下,先宏观理解一下 glusterfs 的整体架构和其中部分核心内容,接着再慢慢去理解不同的细节。



4. 第四章 glusterfs 的特性

从这一章开始，主要分享一些 glusterfs 的特性，例如限制 volume 容量的 quota,快照 snapshot 和 add-brick 的 rebalance 等，这些功能都有着不同的特点与应用场景，下面将进行详细地讲解一下。

4.1. quota 容量限制

4.1.1. 开启 quota 功能

Glusterfs 中的 quota 功能其实就是在对 volume 进行容量限制，使用起来也比较简单，下面来简单尝试一下。

```
1. [root@gfs03 ~]# gluster volume quota test-quota enable
2. volume quota : success
3.
4. [root@gfs03 ~]# gluster volume quota test-quota limit-us
   age / 1GB
5. volume quota : success
6.
7. [root@gfs03 ~]# gluster volume quota test-quota list /
8. Path      Hard-limit  Soft-limit    Used    Available  So
   ft-limit exceeded? Hard-limit exceeded?
9. -----
10. /          1.0GB      80%(819.2MB)  0Bytes  1.0GB
    No                      No
11.
12. [root@gfs03 ~]# gluster volume info test-quota
13.
14. Volume Name: test-quota
15. Type: Replicate
16. Volume ID: cfc66790-7e2f-45e1-b188-4f6a9a3bb210
17. Status: Started
18. Snapshot Count: 0
```

19. Number of Bricks: 1 x 3 = 3
20. Transport-type: tcp
21. Bricks:
22. Brick1: 192.168.0.110:/glusterfs/test-quota
23. Brick2: 192.168.0.111:/glusterfs/test-quota
24. Brick3: 192.168.0.112:/glusterfs/test-quota
25. Options Reconfigured:
26. features.quota-deem-statfs: on
27. features.inode-quota: on
28. features.quota: on
29. cluster.granular-entry-heal: on
30. storage.fips-mode-rchecksum: on
31. transport.address-family: inet
32. nfs.disable: on
33. performance.client-io-threads: off

这里开启了容量限制功能，并且对 volume 的根目录大小限制为 1GB，那么这里查看 volume info 信息的时候，可以留意到多了两个特性，就是和 quota 有关的。

那么下面尝试写入数据进行测试。

1. [root@gfsclient01 ~]# mount -t glusterfs 192.168.0.110:test-quota /mnt/test-quota
- 2.
3. [root@gfsclient01 ~]# cp -rp glusterfs-9.2.zip /mnt/test-quota/
- 4.
5. [root@gfsclient01 ~]# du -sh /mnt/test-quota/
6. 4.9M /mnt/test-quota/
- 7.
8. [root@gfs03 ~]# gluster volume quota test-quota list /
9. Path Hard-limit Soft-limit Used Available Soft-limit exceeded? Hard-limit exceeded?
10. -----
11. / 1.0GB 80%(819.2MB) 4.8MB 1019.2MB

Path	Hard-limit	Soft-limit	Used	Available	Soft-limit exceeded?	Hard-limit exceeded?
/	1.0GB	80%(819.2MB)	4.8MB	1019.2MB	No	No

这里导入了一个比较小的文件，然后再次使用 list 的时候发现使用量已经改

变了。

4.1.2. quota 对不同 volume 的使用

那么这里前面测试了一下开启 quota 功能的使用，那么这里对于复制卷，到底 quota 的容量是总容量呢？还是每个 brick 的容量？而对于冗余卷来说，因为这里会对数据进行拆分，又是怎样的呢？下面来做一个简单的测试对比。

```
1. root@gfs01:~# gluster volume info test-replica
2.
3. Volume Name: test-replica
4. Type: Replicate
5. Volume ID: d2614e89-9aba-46f6-bf04-984782ac6d6f
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-replica
12. Brick2: 10.0.12.9:/glusterfs/test-replica
13. Brick3: 10.0.12.12:/glusterfs/test-replica
14. Options Reconfigured:
15. features.quota-deem-statfs: on
16. features.inode-quota: on
17. features.quota: on
18. cluster.granular-entry-heal: on
19. storage.fips-mode-rchecksum: on
20. transport.address-family: inet
21. nfs.disable: on
22. performance.client-io-threads: off
```

这里有一个简单的 test-replica 的 3 副本的 volume 然后下面开启 quota 并且限制容量为 2GB,并且挂载之后导入一个镜像查看一下效果。

```
1. root@gfs01:~# gluster volume quota test-replica list /
2. Path      Hard-limit  Soft-limit  Used  Available  Soft-l
   imit exceeded? Hard-limit exceeded?
```

```

3. -----
4. /          2.0GB      80%(1.6GB)    0Bytes    2.0GB
   No                      No
5.
6. root@gfs01:~# ls /mnt/test-replica/
7.
8. root@gfs01:~# cp CentOS-8.3.2011-x86_64-minimal.iso /mnt
   /test-replica/
9.
10. root@gfs01:~# du -sh /glusterfs/test-replica
11. 1.8G      /glusterfs/test-replica
12.
13. root@gfs01:~# du -sh /mnt/test-replica/
14. 1.8G      /mnt/test-replica/
15.
16. root@gfs01:~# gluster volume quota test-replica list /
17. Path          Hard-limit  Soft-limit    Used  Available  Sof
   t-limit exceeded? Hard-limit exceeded?
18. -----
19. /          2.0GB      80%(1.6GB)    1.7GB 268.0MB
   Yes                      No

```

这里可以看到 brick 中的容量大小是和 quota 大小一样的,说明这里 quota 对复制卷的限制的容量大小就是每一个 brick 的大小,下面再看一下其他两个 brick 其实也是一样的。那么这里对于 disperse 冗余卷呢?下面也简单测试一下。

```

1. root@gfs01:~# gluster volume info test-disperse
2.
3. Volume Name: test-disperse
4. Type: Disperse
5. Volume ID: 529c1b28-d49b-4058-a8ef-cc43d265061a
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x (2 + 1) = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-disperse

```

```

12. Brick2: 10.0.12.9:/glusterfs/test-disperse
13. Brick3: 10.0.12.12:/glusterfs/test-disperse
14. Options Reconfigured:
15. features.quota-deem-statfs: on
16. features.inode-quota: on
17. features.quota: on
18. storage.fips-mode-rchecksum: on
19. transport.address-family: inet
20. nfs.disable: on
21.
22. root@gfs01:~# gluster volume quota test-disperse list /
23. Path      Hard-limit  Soft-limit    Used  Available  Soft-
    limit exceeded? Hard-limit exceeded?
24. -----
    -----
25. /          N/A        N/A          N/A    N/A
    N/A          N/A
26.
27. root@gfs01:~# cp CentOS-8.3.2011-x86_64-minimal.iso /mnt
    /test-disperse/
28.
29. root@gfs01:~# gluster volume quota test-disperse list /
30. Path      Hard-limit  Soft-limit    Used  Available
    Soft-limit exceeded? Hard-limit exceeded?
31. -----
    -----
32. /          2.0GB      80%(1.6GB)    1.7GB 268.0MB
    Yes          No
33.
34. root@gfs01:~# du -sh /mnt/test-disperse/
35. 1.8G      /mnt/test-disperse/
36.
37. root@gfs01:~# du -sh /glusterfs/test-disperse/
38. 892M      /glusterfs/test-disperse/

```

从这里可以看到，disperse 冗余卷的数据分布是不一样的，这里会对文件进行拆分，那么这里的 quota 大小，其实就是数据的完整大小的，而不是单个 brick 的大小了，这里就要注意区别了，下面再看看其他两个 brick 的大小情况。

```

1. root@gfs02:~# du -sh /glusterfs/test-disperse/
2. 892M      /glusterfs/test-disperse/

```



```

3.
4. root@gfs03:~# du -sh /glusterfs/test-disperse/
5. 892M    /glusterfs/test-disperse/

```

所以这里对于不同类型的 volume 使用 quota 大小是有区别的，那么这里对于分布式复制卷呢？会不会又不一样，大家可以自行测试一下。

4.1.3. quota 真的能限制大小吗？

那么下面尝试一下将一个超过容量限制的文件存放在 volume 里面，看看会有怎样的效果。

```

1. [root@gfsclient01 ~]# du -sh ubuntu-18.04.3-desktop-amd64.
   iso
2. 2.0G    ubuntu-18.04.3-desktop-amd64.iso
3.
4. [root@gfsclient01 ~]# cp ubuntu-18.04.3-desktop-amd64.iso
   /mnt/test-quota/
5. cp: error writing '/mnt/test-quota/ubuntu-18.04.3-desktop
   -amd64.iso': Disk quota exceeded
6. cp: failed to extend '/mnt/test-quota/ubuntu-18.04.3-desk
   top-amd64.iso': Disk quota exceeded
7. cp: failed to close '/mnt/test-quota/ubuntu-18.04.3-deskt
   op-amd64.iso': Disk quota exceeded
8.
9.
10. [root@gfs03 ~]# gluster volume quota test-quota list /
11. Path          Hard-limit  Soft-limit    Used  Available  S
   oft-limit exceeded? Hard-limit exceeded?
12. -----
13. /              1.0GB      80%(819.2MB)  1.1GB  0Bytes
                        Yes              Yes

```

这里在客户端放入一个 2G 大小的镜像文件，而 volume 容量才 1GB，并且已经使用一些空间了，因此这里放入的文件最终是失败的，会提示 Disk quota

exceeded，也就是超过容量限制了，而这时候再次查看 volume 的容量使用，会发现已经超过了容量限制了。因此在使用 quota 的时候，要对这一点比较注意，并不是使用 quota 就一定可以绝对限制容量大小的。

那么这时候再来看 brick 的日志，会发现很多报错了。

```
1. [root@gfs01 bricks]# tail -n 10 -f glusterfs-test-quota.log
2. ....
3. [2021-06-15 14:54:03.263585 +0000] W [socket.c:767:__socket_rvw] 0-tcp.test-quota-server: readv on 192.168.0.112:49114 failed (No data available)
4. [2021-06-15 14:54:03.263628 +0000] I [MSGID: 115036] [server.c:500:server_rpc_notify] 0-test-quota-server: disconnecting connection [{client-uid=CTX_ID:b15f6358-7988-40f9-bf47-13f00016774a-GRAPH_ID:0-PID:2005-HOST:gfs03-PC_NAME:test-quota-client-0-RECON_NO:-0}]
5. [2021-06-15 14:54:03.263740 +0000] I [MSGID: 101055] [client_t.c:397:gf_client_unref] 0-test-quota-server: Shutting down connection CTX_ID:b15f6358-7988-40f9-bf47-13f00016774a-GRAPH_ID:0-PID:2005-HOST:gfs03-PC_NAME:test-quota-client-0-RECON_NO:-0
6. [2021-06-15 14:54:04.129323 +0000] E [MSGID: 115067] [server-rpc-fops_v2.c:1324:server4_writev_cbk] 0-test-quota-server: WRITE info [{frame=13323}, {WRITEV_fd_no=0}, {uuid_upto=26d275a0-510b-47c5-aa8b-c93fcbd8f1a8}, {client=CTX_ID:8616da2c-1d9b-4952-809d-ecdafda5fe7d-GRAPH_ID:0-PID:1616-HOST:gfsclient01-PC_NAME:test-quota-client-0-RECON_NO:-0}, {error-xlator=test-quota-quota}, {errno=122}, {error=Disk quota exceeded}]
7. ....
```

4.1.4. quota 监控

那么这里如果想要监控容量的话，操作系统中如果分区的容量不是独占的，那么是无法正常监控的，如下所示。

```
1. [root@gfs03 ~]# df -h /glusterfs/test-quota
```

2.	Filesystem	Size	Used	Avail	Use%	Mounted on
3.	/dev/mapper/centos-root	50G	3.8G	47G	8%	/
4.						
5.	[root@gfs03 ~]# gluster volume get test-quota quota-deem-statfs					
6.	Option	Value				
7.	-----	-----				
8.	features.quota-deem-statfs	on				

这里因为系统中的使用 `df -h` 的结果是不准确的，因此这里可以考虑使用一些脚本的方式进行监控。

这里要实现的步骤如下所示：

1. 遍历所有的 volume, 然后判断是否开启 quota 功能，如果开启则跳转 2，否则下一个 volume，遍历完成所有 volume 之后，进入步骤 5。
2. 这里还要进一步检查，quota list 中的数据是否为 Null，如果是则进行下一个 volume 遍历，回到步骤 1，否则进入下一个步骤
3. 获取 quota list 中的信息，这里获取 Hard-limit 和 Used 下的大小，然后进行数值单位归一化处理，也就是防止 TB 与 MB 的单位的数值进行直接比较，然后进入下一步。
4. 判断计算使用率是否超过阈值，如果超过则累计告警信息，否则回到步骤 1。
5. 遍历结束后，判断告警信息中是否为空，如果不为空则证明有 volume 要超过阈值要进行告警，输出调用告警函数。

这里的步骤比较简单，不过主要是一些细节可能需要注意一下，例如过滤获取 quota 信息中的单位，要对单位进行归一化处理。下面给出一个简单的监控

告警脚本。

```
1.  #!/bin/bash
2.
3.  volAterStr=""
4.  #这两个变量用于检查数据容量是否有不一致要进行转换的
5.  volQuotaTotalRatio=1
6.  volQuotaUsedRatio=1
7.  #这个变量用于获取当前 vol 的 quota 信息
8.  volQuotaInfo=""
9.  #告警阈值
10. volLimitAlter=0.7
11.
12.
13.
14. #这个函数用于对 quota 中的单位显示进行归一化
15. #方便进行统一比较数值大小。
16. function volSizeChange(){
17.     unix=$1
18.     volRatio=1
19.
20.     if [ $unix == "GB" ]
21.     then
22.         volRatio=1024
23.     elif [ $unix == "TB" ]
24.     then
25.         volRatio=1048576
26.     elif [ $unix == "MB" ]
27.     then
28.         volRatio=1
29.     elif [ $unix == "KB" ]
30.     then
31.         volRatio=0.001
32.     fi
33.
34.     param=$2
35.     if [ $param == "total" ]
36.     then
37.         volQuotaTotalRatio=$volRatio
```

```

38.     else
39.         volQuotaUsedRatio=$volRatio
40.     fi
41.
42. }
43.
44.
45. #这个函数用于获取 quota 信息的
46. #有可能出现 quota command failed 报错
47. #入参是 volume 名称
48. volGetQuota(){
49.     volQuotaInfo=" "
50.     num=1
51.     volume=$1
52.
53.     while :
54.     do
55.         volQuotaInfo=`sudo gluster volume quota $volume list
56.         /`
57.         checkInfo=`sudo echo -e "$volQuotaInfo\n" |grep "Har
58.         d-limit"`
59.
60.         if [ $num -gt 10 ]
61.         then
62.             quotaError=`sudo echo -e "\n 当前
63.             volume: $volume 尝试获取 quota 超过次数失败!!!!\n"`
64.             volAlterStr=${volAlterStr}${quotaError}
65.             return 1
66.         elif [ -z "$checkInfo" ]
67.         then
68.             volQuotaInfo=""
69.             sudo echo -e "\n $volume 尝试 $num 次获取 quota 信
70.             息...."
71.         else
72.             return 0
73.         fi

```

```

71.
72.     num=`expr $num + 1`
73.
74.     done
75.
76. }
77.
78.
79. #获得 vol 的 quota 容量信息进行比较的。
80. #如果超过比例的话，那么这里将会累计告警信息。
81. #待全部 vol 遍历完成再统一发送告警,避免过于频繁发送。
82. volQuota(){
83.
84.     volume=$1
85.
86.     volGetQuota $volume
87.     quota=$volQuotaInfo
88.
89.     #这里要对计算单位进行归一化处理
90.     volQuotaTotalRatio=1
91.     quotaTotal=`sudo echo -e "$quota\n" | grep "80%" |
    awk -F ' ' '{print $2}' | tr -cd "[0-9.]"`
92.     quotaTotalUnix=`sudo echo -e "$quota\n" | grep "80%
    %" |awk -F ' ' '{print $2}' | tr -d "0-9."`
93.     volSizeChange $quotaTotalUnix "total" $volume
94.     quotaTotalNew=`awk -v x=$quotaTotal -v y=$volQuotaT
    otalRatio 'BEGIN{printf "%.2f\n",x*y}'`
95.
96.
97.     volQuotaUsedRatio=1
98.     quotaUsed=`sudo echo -e "$quota\n" | grep "80%" |
    awk -F ' ' '{print $4}' | tr -cd "[0-9.]"`
99.     quotaUsedUnix=`sudo echo -e "$quota\n" | grep "80%
    " |awk -F ' ' '{print $4}' | tr -d "0-9."`
100.    volSizeChange $quotaUsedUnix "used" $volume
101.    quotaUsedNew=`awk -v x=$quotaUsed -v y=$volQuotaUse
    dRatio 'BEGIN{printf "%.2f\n",x*y}'`
102.
103.    #判断是否为空或者 null

```

```

104.     if [ -z "$quotaTotal" ]
105.     then

106.         sudo echo -e "$1 volume 已经开启 quota 功能,但是总
           容量为 Null "

107.     elif [ -z "$quotaUsed" ]
108.     then

109.         sudo echo -e "$1 volume 已经开启 quota 功能,但是使
           用量为 Null "

110.     else
111.         percent=`awk 'BEGIN{printf "%.2f\n",'$quotaUsed
           New '/'$quotaTotalNew'}'`

112.         #如果 num1>num2,则为 1,否则为 0
113.         result=`awk -v num1=$volLimitAlter -v num2=$pe
           rcent 'BEGIN{print(num1>num2)?"1":"0"}'`
114.         if [ $result -eq 0 ]
115.         then

116.             #告警信息累计

117.             quotaTotalWithSize=`sudo echo -e "$quota\n"
           | grep "80%" |awk -F ' ' '{print $2}'`
118.             quotaUsedWithSize=`sudo echo -e "$quota\n"
           | grep "80%" |awk -F ' ' '{print $4}'`
119.             volumeAlterStr="\n\nvolume: $1\n  volQuotaTo
           tal: $quotaTotalWithSize\n  volQuotaUsedWithSize: $quotaU
           sedWithSize\n  volUsedPercent: $percent\n"

120.             #这里拼接告警信息
121.             volAlterStr=${volAlterStr}${volumeAlterStr}

122.         fi
123.     fi
124.
125.}
126.
127.

128.#遍历 vol, 获取每个 vol 的 quota 信息
129.volQuery(){
130.    volList=`sudo gluster volume list`

```

```

131.  for vol in $volList
132.  do

133.      #检查 vol quota 功能是否开启

134.      #这里可以通过检查 vol 的 info 信息中是否会 quota 相关特性,并
      且该特性是 on 的

135.      features_quota=`sudo gluster volume info $vol |g
      rep "features.quota" | awk -F ' ' '{print $NF}' |grep "on"
      `

136.      if [ -z "$features_quota" ]
137.      then
138.          sudo echo -e "\n$vol quota disable"
139.          continue
140.      else

141.          #开启 quota 的 vol 将进行检查容量使用情况

142.          volQuota "$vol"
143.      fi
144.  done
145.
146.

147.  #这里遍历完所有的 volume

148.  #如果累计的告警信息不为空,那么证明有 vol 要进行告警

149.  if [ -n "$volAlterStr" ]
150.  then
151.      sudo echo -e "\n  alter info: $volAlterStr  \n"
152.  fi
153.}
154.
155.
156.
157.volQuery

```

这里脚本执行的结果如下所示。

```

1.  [root@gfs03 ~]# bash gfs-quota.sh
2.
3.  test-arbiter quota disable
4.

```



```
5. test-disperse quota disable
6.
7. test-replica quota disable
8.
9. alter info:
10.
11. volume: test-quota
12. VolQuotaTotal: 1.0GB
13. volQuotaUsedWithSize: 1.1GB
14. volUsedPercent: 1.10
```

对于 quota 的监控，因为如果 volume 不是使用 lvm2 创建的话，那么文件系统中显示的容量大小就是操作系统全部的容量集合，因此没有办法准确地显示容量限制大小，所以可以考虑使用命令获取的方式进行监控。

当然这里还有一种操作方式，就是操作系统层面对分区做好容量限制范围，那样每个 volume 的 brick 对应一个分区，但是这种操作相对麻烦一点，可以根据业务场景来进行操作。

4.2. 偷懒的快照 snapshot

对于一个分布式文件系统来说，快照是一个比较重要的功能，而快照这个概念，在很多地方也能遇到，例如安装虚拟机用的 virtualbox 或者 vmware 这些软件，也可以给虚拟机做快照，但是具体的实现可能各不相同，那么 glusterfs 中的快照功能其实是有点“偷懒”的，下面就来了解一下。

4.2.1. lvm2 原理

首先 glusterfs 的快照需要 lvm2 功能的支持，也就是说，并不是一个普通的操作系统就可以进行创建快照的，下面来了解一下。

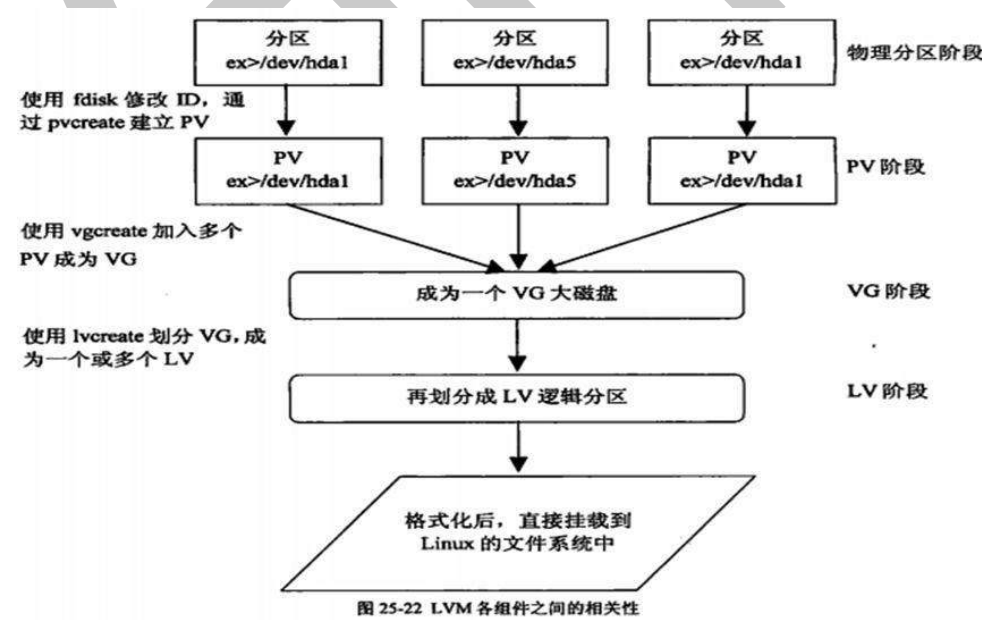
```
1. [root@gfs02 ~]# gluster volume info test-snapshot
2.
3. Volume Name: test-snapshot
4. Type: Replicate
5. Volume ID: 09c5d8b7-3c79-487e-8dc5-48b15db809ec
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 192.168.0.110:/glusterfs/test-snapshot
12. Brick2: 192.168.0.111:/glusterfs/test-snapshot
13. Brick3: 192.168.0.112:/glusterfs/test-snapshot
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
20.
21.
22. [root@gfs02 ~]# gluster snapshot create test-snapshot-210
    616 test-snapshot no-timestamp
```

23. snapshot create: failed: Snapshot is supported only for thin provisioned LV. Ensure that all bricks of test-snapshot are thinly provisioned LV.
24. Snapshot command failed

这里提示快照是需要 lv 功能支持的,那么也就是 brick 的目录也是要 lvm2 格式的,而 lvm2 是 Linux 中一个功能,而 linux 中的 lvm2 主要是以下几点组成的。

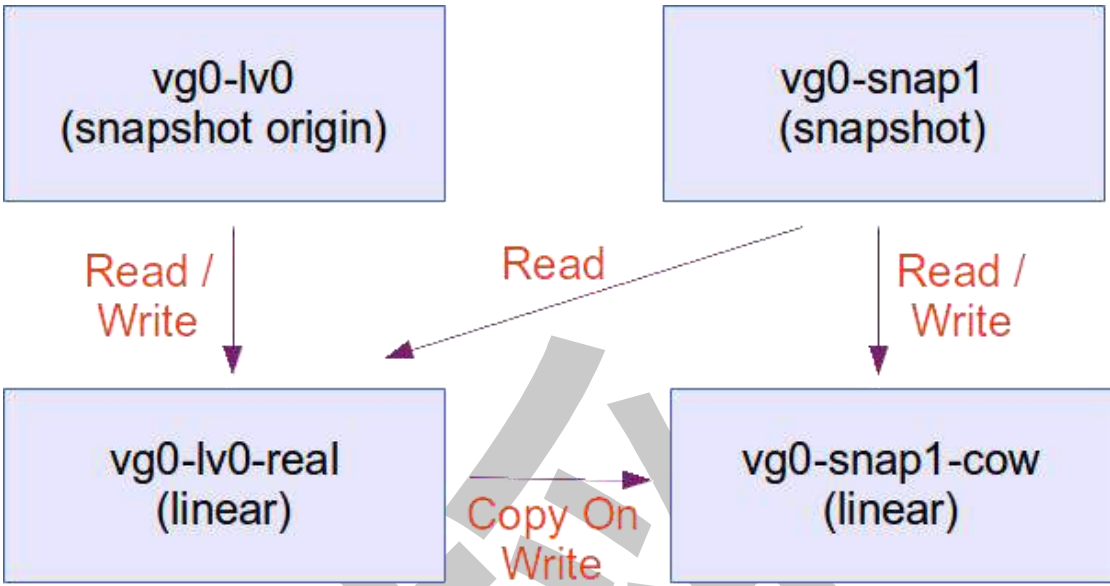
1. 将设备指定为物理卷
2. 用一个或者多个物理卷来创建一个卷组
3. 物理卷是用固定大小的物理区域 (Physical Extent, PE) 来定义的
4. 在物理卷上创建的逻辑卷是由物理区域 (PE) 组成
5. 可以在逻辑卷上创建文件系统

为了更好地理解一下 lvm 中的 pv,vg 和 lv 概念,可以用下面的图来理解。



那么这里 vg 和 lv 这些是如何组织的呢? 可以通过下图来理解一

下。



这里其实就是当创建一个快照之后，会创建一个新的目录位置，而新的数据则会指向到新的目录位置，但是读取旧数据的时候，还是从原来旧的目录进行读取。

4.2.2. lvm2 命令的使用

为了进一步理解这几个之间的关系，下面在客户端使用命令创建并且挂载 lvm2 目录。

```
1. [root@gfsclient01 ~]# lsblk
2. NAME                                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
3. sda                                  8:0      0  100G  0 disk
4. └─sda1                              8:1      0    1G  0 part /boot
5. └─sda2                              8:2      0   99G  0 part
```

```

6.    |centos-root 253:0    0   50G  0 lvm  /
7.    |centos-swap 253:1    0   3.9G  0 lvm  [SWAP]
8.    |centos-home 253:2    0  45.1G  0 lvm  /home
9.   sdb              8:16    0    5G  0 disk
10.  sr0              11:0     1 1024M  0 rom
11.
12.

13. #创建 pv

14. [root@gfsclient01 ~]# pvcreate  --qq --metadatasize=512M
    --dataalignment=256K /dev/sdb
15.
16. [root@gfsclient01 ~]# pvcreate  --qq --metadatasize=512M
    --dataalignment=256K /dev/sdb
17. pvcreate: unrecognized option '--qq'
18.   Error during parsing of command line.
19.
20. [root@gfsclient01 ~]# pvcreate  -qq --metadatasize=512M -
    -dataalignment=256K /dev/sdb
21.
22. [root@gfsclient01 ~]# lsblk
23. NAME                                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
24. sda                                8:0      0  100G  0 disk
25. └─sda1                            8:1      0    1G  0 part /boot
26. └─sda2                            8:2      0   99G  0 part
27.    |centos-root 253:0    0   50G  0 lvm  /
28.    |centos-swap 253:1    0   3.9G  0 lvm  [SWAP]
29.    |centos-home 253:2    0  45.1G  0 lvm  /home
30. sdb                                8:16     0    5G  0 disk
31. sr0                                11:0     1 1024M  0 rom
32.

33. [root@gfsclient01 ~]# pvcreate  -qq --metadatasize=512M -
    -dataalignment=256K /dev/sdb
34.
35. [root@gfsclient01 ~]# pvs
36.  PV          VG      Fmt  Attr  PSize   PFree
37.  /dev/sda2  centos  lvm2 a--   <99.00g 4.00m
38.  /dev/sdb                   lvm2 ---    5.00g 5.00g
39.

40. #创建 vg 信息

41. [root@gfsclient01 ~]# vgcreate -qq --physicalextentsize=4
    M --autobackup=y lvm-sdb-vg /dev/sdb
42.

```

```

43. [root@gfsclient01 ~]# vgs
44.   VG          #PV #LV #SN Attr   VSize   VFree
45.   centos        1   3   0 wz--n- <99.00g  4.00m
46.   lvm-sdb-vg    1   0   0 wz--n- <4.50g <4.50g
47. [root@gfsclient01 ~]# pvs -o pv_name,pv_uuid,vg_name --re
    portformat=json /dev/sdb
48.   {
49.     "report": [
50.       {
51.         "pv": [
52.           {"pv_name":"/dev/sdb", "pv_uuid":"sl0tKG-SL4x-VgvT-S71I-VpmH-mk1D-LmMEBj", "vg_name":"lvm-sdb-vg"}
53.         ]
54.       }
55.     ]
56.   }
57.
58. [root@gfsclient01 ~]# udevadm info --query=symlink --name
    =/dev/sdb
59. disk/by-id/ata-VBOX_HARDDISK_VB4923bed4-cb43f4fd disk/by-
    id/lvm-pv-uuid-sl0tKG-SL4x-VgvT-S71I-VpmH-mk1D-LmMEBj disk
    /by-path                                     /pci-000
    0:00:0d.0-ata-2.0
60.
61. [root@gfsclient01 ~]# vgdisplay -c lvm-sdb-vg
62.   lvm-sdb-vg:r/w:772:-1:0:0:0:-1:0:1:1:4714496:4096:1151:
    0:1151:t19Pps-GeL4-lzWv-msMM-MgFF-oqhD-iGgXqd
63.
64. #创建 lv 信息
65. [root@gfsclient01 ~]# lvcreate -qq --autobackup=y --pool
    metadatasize 8192K --chunksize 256K --size 1048576K --thin
    lvm-sdb                                     -vg/lvm-
    sdb-tp --virtualsize 1048576K --name lvm-sdb-lvname
66.
67. [root@gfsclient01 ~]# lvs
68.   LV          VG          Attr      LSize   Pool
    Origin Data%  Meta%   Move Log Cpy%Sync Convert
69.   home          centos      -wi-ao---- <45.12g
70.   root          centos      -wi-ao---- 50.00g
71.   swap          centos      -wi-ao---- <3.88g
72.   lvm-sdb-lvname lvm-sdb-vg Vwi-a-tz-- 1.00g lvm-sdb-tp
    0.00

```

```

73.   lvm-sdb-tp      lvm-sdb-vg twi-aotz--    1.00g
      0.00    10.45
74.
75. [root@gfsclient01 ~]# ls -l /dev/mapper/
76. total 0
77. lrwxrwxrwx. 1 root root      7 Jun 15 12:22 centos-home
   -> ../dm-2
78. lrwxrwxrwx. 1 root root      7 Jun 15 12:22 centos-root
   -> ../dm-0
79. lrwxrwxrwx. 1 root root      7 Jun 15 12:22 centos-swap
   -> ../dm-1
80. crw----- . 1 root root 10, 236 Jun 15 12:22 control
81. lrwxrwxrwx. 1 root root      7 Jun 15 12:31 lvm--sdb--vg
   -lvm--sdb--lvname -> ../dm-7
82. lrwxrwxrwx. 1 root root      7 Jun 15 12:31 lvm--sdb--vg
   -lvm--sdb--tp -> ../dm-6
83. lrwxrwxrwx. 1 root root      7 Jun 15 12:31 lvm--sdb--vg
   -lvm--sdb--tp_tdata -> ../dm-4
84. lrwxrwxrwx. 1 root root      7 Jun 15 12:31 lvm--sdb--vg
   -lvm--sdb--tp_tmeta -> ../dm-3
85. lrwxrwxrwx. 1 root root      7 Jun 15 12:31 lvm--sdb--vg
   -lvm--sdb--tp-tpool -> ../dm-5
86.
87. #对 lv 进行格式化为 xfs 文件系统。
88. [root@gfsclient01 ~]# mkfs.xfs -i size=512 -n size=8192 /
   dev/mapper/lvm--sdb--vg-lvm--sdb--lvname
89. Discarding blocks...Done.
90. meta-data=/dev/mapper/lvm--sdb--vg-lvm--sdb--lvname isize
   =512    agcount=8, agsize=32768 blks
91.          =                sectsz=512    attr=2, pro
   jid32bit=1
92.          =                crc=1         finobt=0, s
   parse=0
93. data      =                bsize=4096    blocks=2621
   44, imaxpct=25
94.          =                sunit=64     swidth=64 b
   lks
95. naming    =version 2       bsize=8192    ascii-ci=0
   ftype=1
96. log       =internal log    bsize=4096    blocks=2560,
   version=2
97.          =                sectsz=512    sunit=64 bl
   ks, lazy-count=1

```

```

98. realtime =none                      extsz=4096   blocks=0, r
    textents=0
99.
100.#创建目录进行挂载
101.[root@gfsclient01 ~]# mkdir -p /var/lib/gfs/lvm-sdb-vg/lv
    m-sdb-lvname
102.
103.[root@gfsclient01 ~]# mount -o rw,inode64,noatime,nouuid,
    noauto,x-systemd.automount /dev/mapper/lvm--sdb--vg-lvm--s
    db--lvna                                me /var/
    lib/gfs/lvm-sdb-vg/lvm-sdb-lvname
104.
105.[root@gfsclient01 ~]# df -h /var/lib/gfs/lvm-sdb-vg/lvm-s
    db-lvname
106.


| Filesystem                                | Size  | Used | Avail |
|-------------------------------------------|-------|------|-------|
| /dev/mapper/lvm--sdb--vg-lvm--sdb--lvname | 1014M | 33M  | 982M  |


107.
108.
109.[root@gfsclient01 ~]# lsblk
110.


| NAME                            | MAJ:MIN | RM | SIZE  | RO | T |
|---------------------------------|---------|----|-------|----|---|
| sda                             | 8:0     | 0  | 100G  | 0  | d |
| sda1                            | 8:1     | 0  | 1G    | 0  | p |
| sda2                            | 8:2     | 0  | 99G   | 0  | p |
| centos-root                     | 253:0   | 0  | 50G   | 0  | l |
| centos-swap                     | 253:1   | 0  | 3.9G  | 0  | l |
| centos-home                     | 253:2   | 0  | 45.1G | 0  | l |
| sdb                             | 8:16    | 0  | 5G    | 0  | d |
| lvm--sdb--vg-lvm--sdb--tp_tmeta | 253:3   | 0  | 8M    | 0  | l |
| lvm--sdb--vg-lvm--sdb--tp-tpool | 253:5   | 0  | 1G    | 0  | l |
| lvm--sdb--vg-lvm--sdb--tp       | 253:6   | 0  | 1G    | 1  | l |


111.
112.
113.
114.
115.
116.
117.
118.
119.
120.

```



```

121.|    └─lvm--sdb--vg-lvm--sdb--lvname 253:7      0      1G  0 1
    vm  /var/lib/gfs/lvm-sdb-vg/lvm-sdb-lvname
122.└─lvm--sdb--vg-lvm--sdb--tp_tdata  253:4      0      1G  0 1
    vm
123.    └─lvm--sdb--vg-lvm--sdb--tp-tpool 253:5      0      1G  0 1
    vm
124.    └─lvm--sdb--vg-lvm--sdb--tp      253:6      0      1G  1 1
    vm
125.    └─lvm--sdb--vg-lvm--sdb--lvname 253:7      0      1G  0 1
    vm  /var/lib/gfs/lvm-sdb-vg/lvm-sdb-lvname
126.sr0

```

这里通过 pv,vg 和 lv 操作之后,再对 mapper 进行格式化为常用的文件系统,然后就可以挂载使用了。另外注意,这里建议取的名字使用下划线,因为 lvm2 的 lvcreate 这些命令中使用横线的话,识别出现会和命名的有点区别,而使用下划线可以避免命令的统一。

同时这里还可以留意在对块设备进行操作前 lsblk 的输出,对比最后挂载目录之后,这里可以发现对应的 lv,vg 和 pv,还有和块设备的关联关系了。

这里也可以留意到,一个块设备进行格式化之后,这里需要一定空间进行保留元数据信息的,这里是有要求的,那么这个大小一般要求多少呢?通常保留不超过 16G 的大小。

下面给出一个脚本来进行创建。

```

1. [root@gfsclient01 ~]# cat lvm2.sh
2.
3. size="50G"
4.
5.
6. sizenum=`echo $size | tr -cd "[0-9]"`
7.
8. vsize=`awk -v x=$sizenum -v y=0.9 'BEGIN{printf "%.2f\n",
   x*y}'`"G"
9.
10. mountpath="/glusterfs/mounts"

```

```

11.
12. devices="/dev/sdb"
13. vgName="lv_sdb_vg_210616"
14.
15. sudo pvcreate -qq --metadatasize=512M --dataalignment=256
    K $devices
16.
17. sudo vgcreate -qq --physicalextentsize=4M --autobackup=y
    $vgName $devices
18.
19. #pvs show info
20. sudo pvs -o pv_name,pv_uuid,vg_name --reportformat=json $
    devices
21.
22. #udevadm
23. sudo udevadm info --query=symlink --name=$devices
24.
25. #vgdisplay -c
26. sudo vgdisplay -c $vgName
27.
28. #这里--thinpool 后面第一个参数是 thin 的名称,第二个是 vg 的名
    称

29. #-Zn 是为了 consider disabling zeroing,避免警告.
30. lvcreate -L $vsize --thinpool thinpool_$vgName $vgName -
    Zn
31.
32. #这里--thin -n后面的参数,第一个是thin volume名称,后面斜杠前
    是vg名称,斜杠后是thinpool名称

33. lvcreate -V $vsize --thin -n lvuser_$vgName $vgName/thinp
    ool_$vgName
34.
35. #format xfs
36. sudo mkfs.xfs -i size=512 -n size=4096 -s size=4096 -K /
    dev/mapper/$vgName-lvuser_$vgName
37.
38. #write into fstab
39. sudo mkdir -p $mountpath/$vgName/lvuser_$vgName/brick
40.

```

```

41. sudo mount -o rw,inode64,noatime,nouuid,noauto,x-systemd.
    automount /dev/mapper/$vgName-lvuser_$vgName $mountpath/
    $vgName/lvuser_$vgName/brick
42.
43. sudo awk "BEGIN {print \"/dev/mapper/$vgName-lvuser_$vgName
    $mountpath/$vgName/lvuser_$vgName/brick xfs rw,inode6
    4,noatime,nouuid,noauto,x-systemd.automount 1 2 \" >> \"/
    etc/fstab\"}"
44.
45. sudo df -h $mountpath/$vgName/lvuser_$vgName/brick

```

这里和前面的区别是 lvcreate 创建的时候，考虑了块设备的容量大小进行设置，并且改用 lvcreate -V 参数，这样可以把除了保留的空间以外都使用上，最后还会把这个挂载信息写入到/etc/fstab 文件中。

下面是其中一个节点的执行结果。

```

1. [root@gfs03 ~]# bash lvm2.sh
2. {
3.     "report": [
4.         {
5.             "pv": [
6.                 {"pv_name":"/dev/sdb", "pv_uuid":"EbzPs
F-Qqkl-FMy7-11EZ-clld-dAyl-uolZbi", "vg_name":"lv_sdb_vg_2
10616"}
7.             ]
8.         }
9.     ]
10. }
11. disk/by-id/ata-VBOX_HARDDISK_VBf2804901-94951830 disk/by-
    id/lvm-pv-uuid-EbzPsF-Qqkl-FMy7-11EZ-clld-dAyl-uolZbi disk
    /by-path/pci-0000:00:0d.0-ata-2.0
12.   lv_sdb_vg_210616:r/w:772:-1:0:0:0:-1:0:1:1:51900416:409
    6:12671:0:12671:D32P11-Ns7P-dfV4-2Nj0-Nwuh-04SE-fXMtYs
13.   Thin pool volume with chunk size 64.00 KiB can address
    at most 15.81 TiB of data.
14.   Logical volume "thinpool_lv_sdb_vg_210616" created.
15.   Logical volume "lvuser_lv_sdb_vg_210616" created.
16. meta-data=/dev/mapper/lv_sdb_vg_210616-lvuser_lv_sdb_vg_2
    10616 isize=512    agcount=16, agsize=737280 blks

```

```

17.          =                      sectsz=4096  attr=2, pro
   jid32bit=1
18.          =                      crc=1        finobt=0, s
   parse=0
19. data      =                      bsize=4096    blocks=1179
   6480, imaxpct=25
20.          =                      sunit=16      swidth=16 b
   lks
21. naming    =version 2            bsize=4096    ascii-ci=0
   ftype=1
22. log       =internal log         bsize=4096    blocks=5760,
   version=2
23.          =                      sectsz=4096    sunit=1 blk
   s, lazy-count=1
24. realtime  =none                 extsz=4096    blocks=0, r
   textents=0
25. Filesystem                               Siz
   e Used Avail Use% Mounted on
26. /dev/mapper/lv_sdb_vg_210616-lvuser_lv_sdb_vg_210616 45
   G  33M  45G   1% /glusterfs/mounts/lv_sdb_vg_210616/lvus
   er_lv_sdb_vg_210616/brick

```

那么这里还可以检查一下 vgs 和 lvs 的信息，看看和之前的操作的区别。

```

1. [root@gfs03 ~]# lvs
2.   LV                               VG                               Attr      L
   Size  Pool                               Origin Data%  Meta%  Move
   Log Cpy%Sync Convert
3.   home                               centos                               -wi-ao---- <
   45.12g
4.   root                               centos                               -wi-ao----
   50.00g
5.   swap                               centos                               -wi-ao----
   <3.88g
6.   lvuser_lv_sdb_vg_210616  lv_sdb_vg_210616 Vwi-aot---
   45.00g thinpool_lv_sdb_vg_210616 0.05
7.   thinpool_lv_sdb_vg_210616 lv_sdb_vg_210616 twi-aot---
   45.00g 0.05 10.46
8.
9. [root@gfs03 ~]# vgs
10.  VG                               #PV #LV #SN Attr   VSize   VFree
11.  centos                           1   3   0 wz--n- <99.00g 4.00m
12.  lv_sdb_vg_210616                 1   2   0 wz--n- <49.50g 4.40g

```

可以看到这里保留了 10%的空间作为元数据保存空间的。

4.2.3. snapshot 创建

同理，下面对其他几个节点进行相同操作，并且创建一个 volume 然后创建快照，结果如下所示。

```
1. [root@gfs01 ~]# gluster volume create test-snapshot-lvm2
   replica 3 192.168.0.{110,111,112}:/glusterfs/mounts/lv_sdb
   _vg_210616/lvuser_lv_sdb_vg_210616/brick/vol
2. volume create: test-snapshot-lvm2: success: please start
   the volume to access data
3.
4. [root@gfs01 ~]# gluster volume start test-snapshot-lvm2
5. volume start: test-snapshot-lvm2: success
6.
7. [root@gfs01 ~]# gluster snapshot create test-snapshot-lvm
   2-210616 test-snapshot-lvm2 no-timestamp
8. snapshot create: success: Snap test-snapshot-lvm2-210616
   created successfully
9.
10. [root@gfs01 ~]# gluster snapshot list
11. test-snapshot-lvm2-210616
12.
13. [root@gfs01 ~]# gluster snapshot info
14. Snapshot                               : test-snapshot-lvm2-210616
15. Snap UUID                               : 7e7c2dbe-62bd-4253-b1e6-0f326
    600d117
16. Created                               : 2021-06-15 17:51:41 +0000
17. Snap Volumes:
18.
19.      Snap Volume Name                   : 1de228ed2d248e9bee60
    a5985fdf821
20.      Origin Volume name                 : test-snapshot-lvm2
21.      Snaps taken for test-snapshot-lvm2 : 1
22.      Snaps available for test-snapshot-lvm2 : 255
23.      Status                             : Stopped
```

这里创建了一个名为 test-snapshot-lvm2-210616 的快照，参数 no-timestamp 的作用是不添加随机字符串，否则默认快照名称是添加后缀的，这样不方便管理。

另外这里可以看到快照的默认状态是 stopped 的，但是并不代表快照没有生效，只是该快照的进程并没有激活，因为都是依赖 linux lvm2 的快照，因此这里没有办法看到快照的一些数据信息。

在快照中有两个比较重要的参数，分别是 snap-max-hard-limit 和 snap-max-soft-limit，这两个就是决定了该 volume 的最大快照数量，还有达到多少比例之后，会把之前的快照删掉，因为快照的空间也不是无尽的，因为保留过多的快照数量会非常占用空间的，而实际参数的使用效果，大家可以自行测试一下。

下面可以进行测试一下快照的回滚功能。

4.2.4. 快照回滚

下面进行快照的回滚测试，操作如下所示。

1. //这里先创建一个文件
2. [root@gfs03 ~]# ls /mnt/test-snapshot-lvm2/
3. 1.txt
- 4.
- 5.
6. [root@gfs03 ~]# gluster snapshot restore test-snapshot-lvm2-210616
7. Restore operation will replace the original volume with the snapshotted volume. Do you still want to **continue**? (y/n)
y

```

8. snapshot restore: failed: Volume (test-snapshot-lvm2) has
   been started. Volume needs to be stopped before restoring
   a snapshot.
9. Snapshot command failed
10.
11. //激活快照
12. [root@gfs03 ~]# gluster snapshot activate test-snapshot-
   lvm2-210616
13. Snapshot activate: test-snapshot-lvm2-210616: Snap activa
   ted successfully
14.
15. [root@gfs03 ~]# gluster snapshot status test-snapshot-l
   vm2-210616
16.
17. Snap Name : test-snapshot-lvm2-210616
18. Snap UUID : 7e7c2dbe-62bd-4253-b1e6-0f326600d117
19.
20.      Brick Path      : 192.168.0.110:/run/gluster/
   snaps/1de2288ed2d248e9bee60a5985fdf821/brick1/vol
21.      Volume Group    : lv_sdb_vg_210616
22.      Brick Running   : Yes
23.      Brick PID       : 10189
24.      Data Percentage : 0.05
25.      LV Size         : 45.00g
26.
27.
28.      Brick Path      : 192.168.0.111:/run/gluster/
   snaps/1de2288ed2d248e9bee60a5985fdf821/brick2/vol
29.      Volume Group    : lv_sdb_vg_210616
30.      Brick Running   : Yes
31.      Brick PID       : 9775
32.      Data Percentage : 0.05
33.      LV Size         : 45.00g
34.
35.
36.      Brick Path      : 192.168.0.112:/run/gluster/
   snaps/1de2288ed2d248e9bee60a5985fdf821/brick3/vol
37.      Volume Group    : lv_sdb_vg_210616
38.      Brick Running   : Yes
39.      Brick PID       : 10012
40.      Data Percentage : 0.05
41.      LV Size         : 45.00g

```

这里首先挂载该 volume，然后创建一个文件，因为无法直接 restore 回滚

快照的，需要先激活该快照，接着可以使用 `snapshot status` 看到快照的一些信息，这里包括一家使用的数据比例，brick 的 pid 和路径等信息,这里的路径和 volume brick 是不一样的，要留意一下下。

```
1. [root@gfs03 ~]# gluster volume stop test-snapshot-lvm2
2. Stopping volume will make its data inaccessible. Do you want to continue? (y/n) y
3. volume stop: test-snapshot-lvm2: success
4.
5. [root@gfs03 ~]# gluster snapshot restore test-snapshot-lvm2-210616
6. Restore operation will replace the original volume with the snapshotted volume. Do you still want to continue? (y/n) y
7. Snapshot restore: test-snapshot-lvm2-210616: Snap restored successfully
8.
9. [root@gfs03 ~]# umount /mnt/test-snapshot-lvm2
10.
11. [root@gfs03 ~]# gluster volume start test-snapshot-lvm2
12. volume start: test-snapshot-lvm2: success
13.
14. [root@gfs03 ~]# mount -t glusterfs 192.168.0.110:test-snapshot-lvm2 /mnt/test-snapshot-lvm2
15.
16. [root@gfs03 ~]# ls /mnt/test-snapshot-lvm2/
```

接着这里想要回滚快照的话，需要先停止该 volume，接着进行快照 restore，然后再次挂载的时候，就发现文件已经不存在了。这里还要注意一点，一旦快照被回滚了，那么这里就无法再次看到该快照了。

4.2.5. 删除快照

这里删除快照的方式比较简单，直接使用命令 `delete` 即可。

1. [root@gfs01 ~]# gluster snapshot **delete** test-snapshot-lvm2-21061701
2. Deleting snap will erase all the information about the snap. Do you still want to **continue**? (y/n) y
3. snapshot **delete**: test-snapshot-lvm2-21061701: snap removed successfully

但是如果还存在快照的时候,是无法直接删除 volume 的,会提示以下报错。

1. [root@gfs01 ~]# gluster volume **delete** test-snapshot-lvm2
2. Deleting volume will erase all information about the volume. Do you want to **continue**? (y/n) y
3. volume **delete**: test-snapshot-lvm2: failed: Cannot **delete** Volume test-snapshot-lvm2 ,as it has 1 snapshots. To **delete** the volume, first **delete** all the snapshots under it.

那么这里在使用快照的时候,就要留意一个问题了,当快照无法删除时,就会影响到 volume 的删除了,而无法删除 volume,又会影响到剔除节点等功能。那这里什么情况下会导致快照无法删除呢?就是当本地的 lvm2 出现问题的时候,例如 brick 所在的块设备,因为磁盘容量爆了,导致出现 io error 的时候,这个在生产环境中遇到过,导致快照异常了。

4.3. 防止误删数据的 trash

glusterfs 中提供了一种叫 trash 的功能，在介绍当中，提到如果当数据在重平衡 rebalance 或者文件自愈的时候进行使用，当然，这种功能的出现，个人觉得，更多的用途，似乎可以在一些重要的数据 volume 防止误删的情况下使用，有点类似快照了。这个功能的作用，就是开启之后，默认会在 brick 根目录下面多出来一个 .trashcan 目录，而这个目录下面就是删除或者移动操作的文件了，下面可以简单演示一下效果。

```
1. [root@gfs03 ~]# gluster volume info test-trash
2.
3. Volume Name: test-trash
4. Type: Replicate
5. Volume ID: 4ac49896-08db-4b8c-a893-48b4bd97f492
6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/trash
12. Brick2: 10.0.12.9:/glusterfs/trash
13. Brick3: 10.0.12.12:/glusterfs/trash
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
20.
21. [root@gfs03 ~]# gluster volume get test-trash features.trash
22. Option                                Value
23. -----                                -----
```

首先这里创建了一个简单的 3 副本的复制卷，然后默认 trash 这个功能是关闭的。

```
1. [root@gfs03 ~]# mkdir -p /mnt/test-trash
2.
3. [root@gfs03 ~]# mount -t glusterfs 10.0.12.2:test-trash /mnt/test-trash
```

```

4.
5. [root@gfs03 ~]# gluster volume set test-trash features.trash on
6. volume set: success
7.
8. [root@gfs03 ~]# touch /mnt/test-trash/1.txt
9.
10. [root@gfs03 ~]# date > /mnt/test-trash/1.txt
11.
12. [root@gfs03 ~]# cat /mnt/test-trash/1.txt
13. Thu Jun 21 21:30:49 CST 2018
14.
15. //删除该文件进行测试
16. [root@gfs03 ~]# rm -fr /mnt/test-trash/1.txt
17.
18. [root@gfs03 ~]# ls -la /glusterfs/trash/
19. total 28
20. drwxr-xr-x  5 root root 4096 Jun 21 21:30 .
21. drwxr-xr-x  5 root root 4096 Jun 21 21:29 ..
22. drw----- 262 root root 4096 Jun 21 21:29 .glusterfs
23. drwxr-xr-x  2 root root 4096 Jun 21 21:29 .glusterfs-anonymous-inode-4ac49896-08db-4b8c-a893-48b4
    bd97f492
24. drwxr-xr-x  2 root root 4096 Jun 21 21:30 .trashcan
25.
26. [root@gfs03 ~]# ls -la /glusterfs/trash/.trashcan/
27. total 24
28. drwxr-xr-x 2 root root 4096 Jun 21 21:30 .
29. drwxr-xr-x 5 root root 4096 Jun 21 21:30 ..
30. -rw-r--r-- 2 root root  29 Jun 21 21:30 1.txt_2018-06-21_133056_+0000
31.
32. //在 trashcan 目录下能够找到该文件
33. [root@gfs03 ~]# cat /glusterfs/trash/.trashcan/1.txt_2018-06-21_133056_+0000
34. Thu Jun 21 21:30:49 CST 2018
35.
36. //这里可以看到挂载目录是没有该文件的
37. [root@gfs03 ~]# ls /mnt/test-trash/

```

从这里可以看到，其实就是删除之后，文件转移到了隐藏目录下面的，那么这里还有一些相关的参数，包括 features.trash-max-filesize 和 features.trash-eliminate-path，这两个参数，主要就是控制最大的文件大小，超过该大小的文件不进行转移，还有控制哪些目录下的文件会进行转移等。

4.4. 简洁的客户端 coreutils

在 glusterfs 中，这里提供了一个叫 gfccli 的工具，简单来理解，这可以理解为一个客户端工具，只是能提供一些非常简单的小功能，例如 cat, cp, flock, ls, mkdir, rm 和 stat 等功能，而使用这个功能，还需要先提前安装 glusterfs-coreutils 包，下面来简单演示一下。

```
1. [root@gfs03 ~]# gfc
2. gfc cat    gfc clear  gfc cli    gfc cp
3.
4. [root@gfs03 ~]# gfc cli
5. //连接 volume
6. gfc cli> connect glfs://localhost/test-afr
7. gfc cli (localhost/test-afr)> ls
8. test-dir.sh test.sh c.txt
9. gfc cli (localhost/test-afr)> rm -fr c.txt
10. rm: failed to remove `c.txt': Not a directory
11. //删除文件
12. gfc cli (localhost/test-afr)> rm c.txt
13. gfc cli (localhost/test-afr)> ls
14. test-dir.sh test.sh
15. //使用 cat 命令查看文件
16. gfc cli (localhost/test-afr)> cat test.sh
17. #!/bin/bash
18.
19.
20. hostNameStr=`sudo hostname -i`
21.
22. for i in $(seq 1 10000)
23. do
24.     dateStr=`sudo date`
25.     sudo echo -e $dateStr > 1.txt
26.     sudo echo -e $dateStr > /tmp/$hostNameStr
27. done
28.
29. //这里使用 stat 命令
```

```

30. gfccli (localhost/test-afr)> stat test.sh
31.   File: `test.sh'
32.   Size: 181          Blocks: 1          IO Block: 4096   r
   egular file
33. Device: b8273e59h/3089579609d   Inode: 116561552966268014
   72 Links: 1
34. Access: (0644/-rw-r--r--)  Uid: (   0/   root)   Gid: (
   0/   root)
35. Access: 2021-06-17 14:02:39.782872499 +0800
36. Modify: 2021-06-17 14:02:39.784426977 +0800
37. Change: 2021-06-17 14:02:39.791984784 +0800
38. gfccli (localhost/test-afr)> quit

```

4.5. 强大的 webhook event

在 glusterfs 当中,当创建一个 volume 或者做一些操作的时候,想需要知道这些时间 event 的话,glusterfs 官方提供了一个叫 glustereventsds 的服务,需要先启动,下面进行简单的演示。

```

1. root@ubuntu20:~# gluster-eventsapi status
2. Webhooks:
3. http://10.0.12.9:9000/listen
4.
5. +-----+-----+-----+
6. |  NODE   | NODE STATUS | GLUSTEREVENTSD STATUS |
7. +-----+-----+-----+
8. | localhost |          UP |                OK |
9. +-----+-----+-----+

```

这里可以先看看是否有注册的 webhook URL,这里因为之前已经调用了因此这里会显示,当然这里也可以进行删除。

```

1. root@ubuntu20:~# gluster-eventsapi webhook-del http://10.
   0.12.9:9000/listen
2. +-----+-----+-----+
3. |  NODE   | NODE STATUS | SYNC STATUS |
4. +-----+-----+-----+
5. | localhost |          UP |                OK |
6. +-----+-----+-----+

```

```

7.
8. root@ubuntu20:~# gluster-eventsapi status
9. Webhooks: None
10.
11. +-----+-----+-----+
12. |   NODE   | NODE STATUS | GLUSTEREVENTSD STATUS |
13. +-----+-----+-----+
14. | localhost |           UP |                   OK |
15. +-----+-----+-----+

```

这里取消掉了该 webhook url 了，下面重新弄一下，并且启动一个测试代码进行监听测试。

```

1. root@ubuntu20:~# gluster-eventsapi webhook-add http://10.
  0.12.9:9000/listen
2. ...
3.
4. root@ubuntu20:~# cat webhook.py
5. from flask import Flask, request
6.
7. app = Flask(__name__)
8.
9. @app.route("/listen", methods=["POST"])
10. def events_listener():
11.     gluster_event = request.json
12.     if gluster_event is None:
13.         # No event to process, may be test call
14.         return "OK"
15.
16.     # Process gluster_event
17.     # {
18.     #   "nodeid": NODEID,
19.     #   "ts": EVENT_TIMESTAMP,
20.     #   "event": EVENT_TYPE,
21.     #   "message": EVENT_DATA
22.     # }
23.     print (gluster_event)
24.     return "OK"
25.
26. app.run(host="0.0.0.0", port=9000)

```

那么这里有一段 python 代码,这里就是使用了 flask 框架 ,这里的作用就是

监听了该端口，然后使用 python 进行执行，注意这里需要先安装 python-pip，然后使用 pip 进行安装 flask 包，然后执行。

```
1. root@ubuntu20:~# gluster volume create test-event 10.0.12.9:/glusterfs/test-event force
2. volume create: test-event: success: please start the volume to access data
3.
4. ....
```

那么当程序运行的时候，然后创建了一个 volume 的时候，在程序中可以看到下面的信息输出，这里的输出是有规范的，同时这里有很多 events，其中包括 snapshot,quota,brick 和 volume 等 events 的监听，具体的可以查看官方文档^[1]。

```
1. root@ubuntu20:~# python webhook.py
2. * Serving Flask app 'webhook' (lazy loading)
3. * Environment: production
4.   WARNING: This is a development server. Do not use it in a production deployment.
5.   Use a production WSGI server instead.
6. * Debug mode: off
7. * Running on all addresses.
8.   WARNING: This is a development server. Do not use it in a production deployment.
9. * Running on http://10.0.12.9:9000/ (Press CTRL+C to quit)
10.
11. {'event': 'VOLUME_CREATE', 'message': {'bricks': ' 10.0.12.9:/glusterfs/test-event', 'name': 'test-event'}, 'nodeid': '3ed7d443-1312-4098-bd50-2a9c98ada9fd', 'ts': 1624356529}
12. 10.0.12.9 - - [22/Jun/2021 18:08:49] "POST /listen HTTP/1.1" 200 -
```

附录引用：

[1] <https://gluster.readthedocs.io/en/latest/Administrator-Guide/Events-APIs/>

4.6. 麻烦的扩缩容

在 glusterfs 当中，如果 glusterfs 的 volume 是使用 heketi 创建的，那么就无法正常地使用 quota 进行容量限制了，因为底层使用 lvm2 的时候，如果要扩缩容，有两种办法，一种是对底层的 vg 和 lv 进行处理，但是这种方法风险比较高，一旦操作不慎，会直接影响原来的数据，而且底层的 vg 和 lv 容量改变，信息还需要同步到上层的应用中，会比较麻烦；第二种方法，就是使用 glusterfs 的 add-brick 功能，也就是增加 brick 的方式来进行扩缩容处理，下面来了解一下。

4.6.1. add-brick 操作

```
1. root@gfs01:~# gluster volume info test-event
2.
3. Volume Name: test-event
4. Type: Distribute
5. Volume ID: 4ff63ab9-561a-4c32-a4b9-60c5bda0c6e9
6. Status: Created
7. Snapshot Count: 0
8. Number of Bricks: 1
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.9:/glusterfs/test-event
12. Options Reconfigured:
13. nfs.disable: on
14. transport.address-family: inet
15. storage.fips-mode-rchecksum: on
16.
17.
18. root@gfs01:~# gluster volume add-brick test-event 10.0.12.
    2:/glusterfs/test-event force
19. volume add-brick: success
20.
21. root@gfs01:~# gluster volume info test-event
22.
```



```
23. Volume Name: test-event
24. Type: Distribute
25. Volume ID: 4ff63ab9-561a-4c32-a4b9-60c5bda0c6e9
26. Status: Created
27. Snapshot Count: 0
28. Number of Bricks: 2
29. Transport-type: tcp
30. Bricks:
31. Brick1: 10.0.12.9:/glusterfs/test-event
32. Brick2: 10.0.12.2:/glusterfs/test-event
33. Options Reconfigured:
34. nfs.disable: on
35. transport.address-family: inet
36. storage.fips-mode-rchecksum: on
```

这里有一个单 brick 的卷 test-event,然后使用了 add-brick 进行扩容,那么这里扩容之后要注意,数据是否要进行 rebalance,因为对于数据的分布如果不进行重平衡,那么可能后续会造成数据倾斜,也就是部分 brick 的数据很多,但是新添加的 brick 可能数据很少的现象,而关于 rebalance 的问题,后面还会继续了解。

4.6.2. remove-brick 操作

有了扩容的操作,那么一般自然也会有减少 brick 的,而这个操作可能在实际的生产环境中会使用频率比较少,但是这里也有一些值得注意的问题,下面来了解一下。

```
1. root@gfs01:~# mount -t glusterfs 10.0.12.9:test-event /mnt/test-event
2.
3. root@gfs01:~# ls /mnt/test-event/
4.
5. root@gfs01:~# cd /mnt/test-event/
6.
```

```

7. root@gfs01:~# cat /mnt/test-event/smallFile.sh
8. #!/bin/bash
9.
10. for((i=1;i<=10000;i++));
11. do
12.     touch $i.txt
13.     date > $i.txt
14.     #echo $(expr $i \* 3 + 1);
15. done
16.
17. root@gfs01:~# ls /mnt/test-event/ |wc -l
18. 10001

```

为了测试缩容的效果，这里弄了一万个小文件，然后再执行 remove-brick，在执行之前，这里再核实一下不同的 brick 的文件数量分布情况。

```

1. root@gfs03:~# ls /glusterfs/test-event/ |wc -l
2. 4989
3.
4. root@gfs03:~# hostname -i
5. 10.0.12.9
6.
7. root@gfs02:~# hostname -i
8. 10.0.12.2
9.
10. root@gfs02:~# ls /glusterfs/test-event/ |wc -l
11. 5012

```

这里可以看到，在该 volume 中，文件不是和复制卷一样的，这里也没有绝对平均地分配文件数量，那么下面进行 remove-brick 操作尝试。

```

1. root@gfs01:/mnt/test-event# gluster volume remove-brick test-event 10.0.12.9:/glusterfs/test-event force
2. Remove-brick force will not migrate files from the removed bricks, so they will no longer be available on the volume.
3. Do you want to continue? (y/n) y
4. volume remove-brick commit force: success
5.

```

```

6. root@gfs01:/mnt/test-event# gluster volume info test-event
7.
8. Volume Name: test-event
9. Type: Distribute
10. Volume ID: 4ff63ab9-561a-4c32-a4b9-60c5bda0c6e9
11. Status: Started
12. Snapshot Count: 0
13. Number of Bricks: 1
14. Transport-type: tcp
15. Bricks:
16. Brick1: 10.0.12.2:/glusterfs/test-event
17. Options Reconfigured:
18. performance.client-io-threads: on
19. nfs.disable: on
20. transport.address-family: inet
21. storage.fips-mode-rchecksum: on
22.
23. root@gfs01:/mnt/test-event# gluster volume rebalance test-event status
24. volume rebalance: test-event: failed: Volume test-event is not a distribute volume or contains only 1 brick.
25. Not performing rebalance

```

这里可以看到，因为只有一个 brick，那么是无法进行 rebalance 的，同时这里如果只剩下了最后一个 brick，会有数据丢失的风险的，因此要特别注意。

4.6.3. replace-brick 操作

这里的除了增加和删除之外，其实还有一个叫 replace-brick 的操作，这里在什么时候需要使用该操作呢？主要是在当前节点有问题的时候，想转移这个 brick 的数据到另外一个节点上，那么下面可以了解一下。

```

1. root@gfs01:~# gluster volume info test-replica
2.
3. Volume Name: test-replica
4. Type: Replicate
5. Volume ID: d2614e89-9aba-46f6-bf04-984782ac6d6f

```

```

6. Status: Started
7. Snapshot Count: 0
8. Number of Bricks: 1 x 3 = 3
9. Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-replica
12. Brick2: 10.0.12.9:/glusterfs/test-replica
13. Brick3: 10.0.12.12:/glusterfs/test-replica
14. Options Reconfigured:
15. features.quota-deem-statfs: on
16. features.inode-quota: on
17. features.quota: on
18. cluster.granular-entry-heal: on
19. storage.fips-mode-rchecksum: on
20. transport.address-family: inet
21. nfs.disable: on
22. performance.client-io-threads: off
23.
24.
25.
26. root@gfs01:~# gluster volume replace-brick test-replica 1
    0.0.12.2:/glusterfs/test-replica 10.0.12.2:/glusterfs/test
    -replica-new commit force
27. volume replace-brick: success: replace-brick commit force
    operation successful
28.
29.
30. root@gfs01:~# gluster volume info test-replica
31.
32. Volume Name: test-replica
33. Type: Replicate
34. Volume ID: d2614e89-9aba-46f6-bf04-984782ac6d6f
35. Status: Started
36. Snapshot Count: 0
37. Number of Bricks: 1 x 3 = 3
38. Transport-type: tcp
39. Bricks:
40. Brick1: 10.0.12.2:/glusterfs/test-replica-new
41. Brick2: 10.0.12.9:/glusterfs/test-replica
42. Brick3: 10.0.12.12:/glusterfs/test-replica
43. Options Reconfigured:
44. features.quota-deem-statfs: on
45. features.inode-quota: on
46. features.quota: on

```

```
47. cluster.granular-entry-heal: on
48. storage.fips-mode-rchecksum: on
49. transport.address-family: inet
50. nfs.disable: on
51. performance.client-io-threads: off
```

那么注意，这里因为使用 replace-brick 之后，因为不是分布式卷，因此是无法进行 rebalance 的，下面可以看到内容。

```
1. root@gfs01:~# gluster volume rebalance test-replica status
2. volume rebalance: test-replica: failed: Volume test-replica is not a distributed volume or contains only 1 brick.
3. Not performing rebalance
```

那么这里还可以留意一下是否数据已经转移了。

```
1. root@gfs02:~# ls -l /glusterfs/test-replica
2. total 1822728
3. -rw-r--r-- 2 root root 1866465280 Jun 23 20:51 CentOS-8.3.2011-x86_64-minimal.iso
4.
5. root@gfs02:~# ls -l /glusterfs/test-replica-new/
6. total 1822728
7. -rw-r--r-- 2 root root 1866465280 Jun 23 20:51 CentOS-8.3.2011-x86_64-minimal.iso
8.
9. root@gfs02:~# md5sum /glusterfs/test-replica/CentOS-8.3.2011-x86_64-minimal.iso
10. 8934d42a86d8589342ac9bdfec82d6b4 /glusterfs/test-replica/CentOS-8.3.2011-x86_64-minimal.iso
11.
12. root@gfs02:~# md5sum /glusterfs/test-replica-new/CentOS-8.3.2011-x86_64-minimal.iso
13. 8934d42a86d8589342ac9bdfec82d6b4 /glusterfs/test-replica-new/CentOS-8.3.2011-x86_64-minimal.iso
14.
15. root@gfs02:~# hostname -i
16. 10.0.12.2
```

4.6.4. rebalance 很重要

那么提到了扩缩容的问题，就不得不说一下 rebalance 重平衡了，所谓的重平衡，这里就是对数据的分布进行重新的均衡，这样的话可以保证数据不会出现明显的数据倾斜情况，下面可以进行测试一下。

```
1. root@gfs01:~# gluster volume create rebalance-test replica 3
   10.0.12.{2,9,12}:/glusterfs/rebalance-01 force
2. volume create: rebalance-test: success: please start the volume
   to access data
3.
4. root@gfs01:~# gluster volume start rebalance-test
5. volume start: rebalance-test: success
6.
7. root@gfs01:~# mkdir -p /mnt/rebalance-test
8.
9. root@gfs01:~# mount -t glusterfs 10.0.12.2:rebalance-test /mnt/
   rebalance-test
10.
11.root@gfs01:~# cp /mnt/test-rebalance/test.sh /mnt/rebalance-test/
12.
13.root@gfs01:~# cd /mnt/rebalance-test/
14.
15.root@gfs01:/mnt/rebalance-test# ls
16.test.sh
17.
18.root@gfs01:/mnt/rebalance-test# cat test.sh
19.#!/bin/bash
20.for ((i=1; i<=10000; i++))
21.do
22.  sudo touch $i.txt
23.  sudo date > $i.txt
24.done
25.
26.root@gfs01:/mnt/rebalance-test# bash test.sh
27.
28.root@gfs01:/mnt/rebalance-test# ls -l |wc -l
29.10002
```

这里首先创建一个 3 副本的复制卷，然后使用脚本创建了 1 万个小文件，接着下面就是使用 add-brick 进行增加 brick，然后进行 rebalance 操作。

```
1. root@gfs01:/mnt/rebalance-test# gluster volume add-brick re
   balance-test 10.0.12.{2,9,12}:/glusterfs/rebalance-02 force
2. volume add-brick: success
3.
4. root@gfs01:/mnt/rebalance-test# gluster volume rebalance re
   balance-test start
5. volume rebalance: rebalance-test: success: Rebalance on reba
   lance-test has been started successfully. Use rebalance status
   command to check status of the rebalance process.
6. ID: 39101691-e7d7-4e51-847a-4fc2845eea3a
7.
8. root@gfs01:/mnt/rebalance-test# gluster volume rebalance re
   balance-test status
9.  Node Rebalanced-files size  scanned failures skipped  status
   run time in h:m:s
10. -----
11. 10.0.12.9 21 672Bytes 506 0 0 in progress 0:00:0
   4
12. gfs02 22 704Bytes 504 0 0 in progress 0:00:04
13. localhost 8 576Bytes 503 0 0 in progress 0:00:04
14. The estimated time for rebalance to complete will be unavaila
   ble for the first 10 minutes.
15. volume rebalance: rebalance-test: success
```

从这里可以看到,重平衡已经启动了，那么这里数据最后的分布会变成怎样呢？

```
1. root@gfs02:~# ls -l /glusterfs/rebalance-01/ | wc -l
2. 4990
3. root@gfs02:~# ls -l /glusterfs/rebalance-02/ | wc -l
4. 5013
```

另外对于 volume 的 rebalance 的操作，这里是有两种情况的，一种是像上面的把数据全部重新平衡，包括元数据，还有一种是单纯平衡元数据的，这里

可以根据业务场景的不同进行选择。

对于 rebalance 这里，还有一点需要注意的是，force 选项并不是强制的，加上 force 的话会考虑 brick 的数据分布情况，也就是说，如果不加上 force 的话，那么 add-brick 之后有可能数据并不会一定进行迁移和平衡的，这一点可以查看官方 issue 编号为 2571 的内容，本人在做测试的时候曾经遇到过。



4.7. 性能管家 profile 和 top

对于性能监控,glusterfs 是提供了一些工具的,主要是三个命令 profile、top 和 statedump,这三者面向是不一样的,其中 statedump 主要是排查一些文件句柄泄漏这样的问题,并且可以获取到申请的系统内存等信息。而 profile 和 top 命令,则是通用的性能检测工具,其中 profile 命令默认是不打开的,需要手动启动该命令,适合用在排查当前的 volume 运行情况,哪些 fop 在运行并且延迟怎样; top 命令则适合查看某些文件的句柄连接和打开情况等。下面先来感受一下 profile 命令的使用。

```
1. root@gfs03:~# gluster volume profile test-shd-num info
2. Profile on Volume test-shd-num is not started
3.
4. root@gfs03:~# gluster volume profile test-shd-num start
5. Starting volume profile on test-shd-num has been successful
```

这里需要手动打开 profile 命令,然后下面挂载 volume,使用脚本创建大量的文件。

```
1. root@gfs01:/mnt/test-shd-num# cat test.sh
2. #!/bin/bash
3.
4.
5. for((i=1;i<=10000;i++));
6. do
7.     sudo touch $i.txt
8.     sudo date > $i.txt
9. done
```

这里默认创建一万个小文件,并且文件写入时间,接着执行脚本,并且使用 profile 命令来查看情况。

```
1. root@gfs03:~# gluster volume profile test-shd-num info
2. Brick: 10.0.12.9:/glusterfs/test-shd-num
```

```

3. -----
4. Cumulative Stats:
5.   Block Size:      32b+      64b+      4096b+
6.   No. of Reads:    0          0          0
7.   No. of Writes:   5838      1          1
8.
9.   %-latency Avg-latency Min-Latency Max-Latency No.of
      calls   Fop
10.  -----
11.   0.00   0.00 us   0.00 us   0.00 us   5680   FORGET
12.   0.00   0.00 us   0.00 us   0.00 us   11684  RELEASE
13.   0.00   0.00 us   0.00 us   0.00 us   82     RELEASEDIR
14.   0.03   34.08 us  28.46 us  39.69 us   2     GETXATTR
15.   0.04   45.86 us  25.16 us  66.56 us   2     OPENDIR
16.   2.48   15.42 us  9.22 us   68.12 us  328    ENTRYLK
17.   3.46   21.50 us  9.36 us   131.09 us 328    FINODELK
18.   4.52   56.15 us  38.82 us  243.58 us 164    SETATTR
19.   4.84   60.14 us  47.06 us  155.08 us 164    TRUNCAT
    E
20.   5.33   16.57 us  6.63 us   210.16 us 656    FLUSH
21.   5.38   66.90 us  49.67 us   172.31 us 164    WRITE
22.   5.93   18.44 us  9.63 us   1674.93 us 656    INODELK
23.   6.68   83.05 us  46.11 us   166.93 us 164    OPEN
24.   11.87  73.79 us  35.11 us   287.36 us 328    XATTROP
25.   14.59  90.70 us  29.52 us   7414.06 us 328    FXATTRO
    P
26.   14.75  183.39 us  127.94 us   745.42 us 164    CREAT
    E
27.   20.10  62.03 us  20.71 us   884.99 us 661    LOOKUP
28.
29.   Duration: 5141 seconds
30.   Data Read: 0 bytes
31. Data Written: 191004 bytes
32.
33. Interval 1 Stats:
34.   Block Size:      32b+
35.   No. of Reads:      0
36. No. of Writes:     164

```

```

37. %-latency Avg-latency Min-Latency Max-Latency No.of calls
    Fop
38. -----
39.  0.00  0.00 us  0.00 us  0.00 us  328  RELEASE
40.  0.00  0.00 us  0.00 us  0.00 us   2  RELEA
41.  0.03  34.08 us  28.46 us  39.69 us   2  GETXATTR
42.  0.04  45.86 us  25.16 us  66.56 us   2  OPENDIR
43.  2.48  15.42 us  9.22 us  68.12 us  328  ENTRYLK
44.  3.46  21.50 us  9.36 us  131.09 us  328  FINODELK

45.  4.52  56.15 us  38.82 us  243.58 us  164  SETATTR

46.  4.84  60.14 us  47.06 us  155.08 us  164  TRUNCAT
    E
47.  5.33  16.57 us  6.63 us  210.16 us  656  FLUSH
48.  5.38  66.90 us  49.67 us  172.31 us  164  WRITE
49.  5.93  18.44 us  9.63 us  1674.93 us  656  INODEL
    K
50.  6.68  83.05 us  46.11 us  166.93 us  164  OPEN
51.  11.87  73.79 us  35.11 us  287.36 us  328  XATTRO
    P
52.  14.59  90.70 us  29.52 us  7414.06 us  328  FXATTR
    OP
53.  14.75  183.39 us  127.94 us  745.42 us  164  CREAT
    E
54.  20.10  62.03 us  20.71 us  884.99 us  661  LOOKUP

55.
56.  Duration: 22 seconds
57.  Data Read: 0 bytes
58. Data Written: 5248 bytes
59.
60. Brick: 10.0.12.2:/glusterfs/test-shd-num
61. -----
62. Cumulative Stats:
63.  Block Size:  32b+  64b+  4096b+
64. No.of Reads:    0    0    0
65. No.of Writes:  5838    1    1
66. %-latency Avg-latency Min-Latency Max-Latency No.of calls
    Fop
67. -----
68.  0.00  0.00 us  0.00 us  0.00 us  5680  FORGET
69.  0.00  0.00 us  0.00 us  0.00 us  11684  RELEASE
70.  0.00  0.00 us  0.00 us  0.00 us   82  RELEA

```

71.	0.03	32.32 us	27.99 us	36.64 us	2	GETXATTR
72.	0.04	41.98 us	26.41 us	57.56 us	2	OPENDIR
73.	2.19	14.77 us	8.53 us	181.93 us	328	ENTRYLK
74.	3.66	24.66 us	9.71 us	796.68 us	328	FINODELK
75.	4.10	13.82 us	8.88 us	78.91 us	656	INODELK
76.	4.26	57.36 us	38.09 us	136.71 us	164	SETATTR
77.	4.84	65.21 us	45.30 us	367.69 us	164	TRUNCATE
78.	5.07	68.35 us	50.81 us	212.95 us	164	WRITE
79.	5.83	78.50 us	44.37 us	144.30 us	164	OPEN
80.	6.43	21.64 us	6.32 us	1814.42 us	656	FLUSH
81.	8.39	56.54 us	28.91 us	167.37 us	328	FXATTROP
82.	11.80	79.46 us	35.23 us	1428.97 us	328	XATTROP
83.	18.47	61.74 us	20.05 us	1442.66 us	661	LOOKUP
84.	24.89	335.33 us	148.63 us	3005.57 us	164	CREATE
85.						
86.	Duration: 5141 seconds					
87.	Data Read: 0 bytes					
88.	Data Written: 191004 bytes					
89.						
90.	Interval 1 Stats:					
91.	Block Size:		32b+			
92.	No. of Reads:		0			
93.	No. of Writes:		164			
94.	%-latency	Avg-latency	Min-Latency	Max-Latency	No.of calls	
95.	-----	-----	-----	-----	-----	
96.	0.00	0.00 us	0.00 us	0.00 us	328	RELEASE
97.	0.00	0.00 us	0.00 us	0.00 us	2	RELEASEDIR
98.	0.03	32.32 us	27.99 us	36.64 us	2	GETXATTR
99.	0.04	41.98 us	26.41 us	57.56 us	2	OPENDIR
100.	2.19	14.77 us	8.53 us	181.93 us	328	ENTRYLK
101.	3.66	24.66 us	9.71 us	796.68 us	328	FINODELK

102.	LK	4.10	13.82 us	8.88 us	78.91 us	656	INODE
103.	TTR	4.26	57.36 us	38.09 us	136.71 us	164	SETA
104.	CATE	4.84	65.21 us	45.30 us	367.69 us	164	TRUN
105.	TE	5.07	68.35 us	50.81 us	212.95 us	164	WRI
106.	N	5.83	78.50 us	44.37 us	144.30 us	164	OPE
107.	SH	6.43	21.64 us	6.32 us	1814.42 us	656	FLU
108.	TROP	8.39	56.54 us	28.91 us	167.37 us	328	FXAT
109.	TROP	11.80	79.46 us	35.23 us	1428.97 us	328	XAT
110.	OKUP	18.47	61.74 us	20.05 us	1442.66 us	661	LO
111.	REATE	24.89	335.33 us	148.63 us	3005.57 us	164	C
112.							
113.	Duration: 22 seconds						
114.	Data Read: 0 bytes						
115.	Data Written: 5248 bytes						
116.							
117.	Brick: 10.0.12.12:/glusterfs/test-shd-num						
118.	-----						
119.	Cumulative Stats:						
120.	Block Size:		32b+	64b+	4096b+		
121.	No. of Reads:		0	2	0		
122.	No. of Writes:		5838	1	1		
123.							
124.	%-latency lls Fop		Avg-latency	Min-Latency	Max-Latency	No.of ca	
125.	-----	-----	-----	-----	-----	----	
126.	0.00	0.00 us	0.00 us	0.00 us	5680	FORG	
127.	0.00	0.00 us	0.00 us	0.00 us	11684	RELEA	
128.	0.00	0.00 us	0.00 us	0.00 us	82	RELEASE	
129.	0.12	526.14 us	28.92 us	1023.36 us	2	OPEN	
	DIR						

130.	0.30	1273.69 us	46.65 us	2500.74 us	2	GETX ATTR
131.	1.50	39.03 us	9.81 us	2227.32 us	328	ENTR YLK
132.	1.81	94.05 us	44.32 us	1424.93 us	164	SET ATTR
133.	2.45	126.94 us	56.05 us	3559.47 us	164	TRU NCATE
134.	2.57	133.25 us	53.76 us	4854.92 us	164	WR ITE
135.	2.58	66.91 us	12.21 us	1911.90 us	328	FINO DELK
136.	2.78	36.02 us	7.80 us	1778.86 us	656	FLU SH
137.	2.85	36.95 us	10.73 us	3242.02 us	656	INO DELK
138.	3.19	165.32 us	68.88 us	1437.64 us	164	O PEN
139.	3.73	96.79 us	33.39 us	1454.73 us	328	FXAT TROP
140.	4.74	10077.98 us	15.21 us	40048.82 us	4	REA DDIRP
141.	5.82	151.06 us	39.16 us	3738.27 us	328	XAT TROP
142.	23.35	300.52 us	24.16 us	15041.37 us	661	L OOKUP
143.	42.20	2188.84 us	189.06 us	20331.92 us	164	CREATE
144.						
145.	Duration: 5141 seconds					
146.	Data Read: 184 bytes					
147.	Data Written: 191004 bytes					
148.						
149.	Interval 1 Stats:					
150.	Block Size: 32b+					
151.	No. of Reads: 0					
152.	No. of Writes: 164					
153.	%-latency lls	Avg-latency Fop	Min-Latency	Max-Latency	No.of ca	
154.	-----	-----	-----	-----	----	
155.	0.00	0.00 us	0.00 us	0.00 us	328	RELEAS E
156.	0.00	0.00 us	0.00 us	0.00 us	2	RELEASE DIR

157.	0.12	526.14 us	28.92 us	1023.36 us	2	OPEN DIR
158.	0.30	1273.69 us	46.65 us	2500.74 us	2	GETXA TTR
159.	1.50	39.03 us	9.81 us	2227.32 us	328	ENTR YLK
160.	1.81	94.05 us	44.32 us	1424.93 us	164	SETA TTR
161.	2.45	126.94 us	56.05 us	3559.47 us	164	TRU NCATE
162.	2.57	133.25 us	53.76 us	4854.92 us	164	WR ITE
163.	2.58	66.91 us	12.21 us	1911.90 us	328	FINO DELK
164.	2.78	36.02 us	7.80 us	1778.86 us	656	FLU SH
165.	2.85	36.95 us	10.73 us	3242.02 us	656	INO DELK
166.	3.19	165.32 us	68.88 us	1437.64 us	164	O PEN
167.	3.73	96.79 us	33.39 us	1454.73 us	328	FXAT TROP
168.	4.74	10077.98 us	15.21 us	40048.82 us	4	RE ADDIRP
169.	5.82	151.06 us	39.16 us	3738.27 us	328	XAT TROP
170.	23.35	300.52 us	24.16 us	15041.37 us	661	L OOKUP
171.	42.20	2188.84 us	189.06 us	20331.92 us	164	CREATE
172.						
173.	Duration: 22 seconds					
174.	Data Read: 0 bytes					
175.	Data Written: 5248 bytes					

这里在写入文件的过程中,可以看到 REaddir 和 CREATE 这两个操作耗时是比较多,其中 CREATE 是创建文件需要申请空间,而 REaddir 操作慢,则是因为对于 glusterfs 来说,这里并没有对元数据信息缓存,因此在有很多文件的时候,哪怕执行 ls 耗费的时间也会比较多,因此关于这部分内容,未来也是一个很重要的优化内容。

那么下面也可以使用 top 命令来查看一下 volume 的情况。

```
1. root@gfs03:~# gluster volume top test-shd-num write list-c
   nt 5
2. Brick: 10.0.12.9:/glusterfs/test-shd-num
3. Count    filename
4. =====
5. 1        /773.txt
6. 1        /772.txt
7. 1        /771.txt
8. 1        /770.txt
9. 1        /769.txt
10. Brick: 10.0.12.2:/glusterfs/test-shd-num
11. Count    filename
12. =====
13. 1        /773.txt
14. 1        /772.txt
15. 1        /771.txt
16. 1        /770.txt
17. 1        /769.txt
18. Brick: 10.0.12.12:/glusterfs/test-shd-num
19. Count    filename
20. =====
21. 1        /773.txt
22. 1        /772.txt
23. 1        /771.txt
24. 1        /770.txt
25. 1        /769.txt
26.
27.
28. root@gfs03:~# gluster volume top test-shd-num read list-c
   nt 5
29. Brick: 10.0.12.9:/glusterfs/test-shd-num
30. Brick: 10.0.12.2:/glusterfs/test-shd-num
31. Brick: 10.0.12.12:/glusterfs/test-shd-num
32. Count    filename
33. =====
34. 1        /test.sh
35. 1        /test.sh
```

这里在运行的时候可以看到当前的 volume 读写文件的情况，这里因为只是创建文件然后就关闭，因此这里显示的都是 1，如果想测试，可以自行做

一些其他的。

章节语:

这一章主要关注了 glusterfs 中的一些特性和工具的使用，从快照到 quota 容量限制，还有扩缩容等，这些内容在日常系统使用中会比较频繁遇到，因此有必要平时多测试和进行一些破坏性测试，例如对于快照，可以手动把 lvm 快照删掉或者移除，导致 glusterfs 快照丢失异常等。多进行这样的测试，以便在遇到生产环境问题时能更好地排查问题，平时项目维护的时候规避部分问题等。

5. 第五章 参数与性能优化

5.1. 那些有趣的参数

这里 glusterfs 中提供了非常多的参数，其中不同的参数作用是不同的，而下面将分享一下，关于本人在使用过程中了解和遇到过的使用的一些参数。当然这里要查询不同的参数的作用，想了解参数的用法，可以使用命令 `gluster volume set help` 输出内容。

5.1.1. rebalance 相关

这里在 rebalance 中有一些比较重要的参数需要关注一下，这个参数就是 `rebal-throttle`，这个参数就是控制 rebalance 的时候在节点并行处理的文件数量，这里受限于节点的 cpu 数量的，计算的公式是 $[(\$(processing\ units) - 4) / 2], 4]$ ，在进行 rebalance 的时候，适当提高该参数，可以加快处理的速度。

5.1.2. 服务和性能相关

这里有很多关于读写性能的优化参数，而优化读写的速度，可以优化的地方也是比较多的，下面分享部分常见的。

1. open-behind

这个参数是用于提高文件打开速度的，和匿名 inode 那些有关，因为对于很多文件需要频繁读写打开的时候，每次的文件句柄地创建申请都会比较麻烦，尤其是一些临时文件，因此把该参数设置为 `on`，可以优化文件的打开参数。

然而这个参数，因为在历史版本中，曾经出现过一个比较大的 bug，该 bug 在 7.x 版

本中出现，表现的现象主要是在 k8s 中挂载 volume 的时候，会突然间挂载掉线，但是 volume 是正常的。原因是因为使用了匿名的 fd，在以前的实现方式中，并没有在 fd_t 的结构中添加引用，因此有可能会导导致还在使用的文件句柄被杀死。该问题可以通过关闭该参数解决，如果想了解具体的情况，可以看官方 github 上 issues 为 1225 的内容，那里具体讨论了该情况。

2. config.brick-threads 和 config.client-threads

这个参数是需要设置 config.global-threading 为 on 开启之后才会生效的，该参数的作用就是控制每个 brick 和 clients 最大的线程数量，但是这里设置了，并不代表立马生效的，因为具体的情况，glusterfs 会根据目前的负载来进行处理的。如果当前默认在 120 秒内没有新的请求过来，当前的 xlator 的线程池就会维护最小的数量运行。

3. ctime

这个参数默认是开启的，在生产环境中，我们曾经尝试关闭过该参数，但是发现对于一些依赖于此类时间戳属性的应用程序会中断，如 ES 在检测到 ctime 的差异（如果统计信息来自不同的块）时，将会出现“file changed as we read it”和“Underlying file changed by an external force”的警告，因为不一定总是从副本集的同一块中返回时间属性，而目前该参数发现会对 jira 日志，confluence、elasticsearch 服务有影响。

4. read-hash-mode

因为 glusterfs 的无中心架构，对于每一个节点都是平等地对待，因此这里如何避免解决读写热节点则天然不具备优势，这里不像 k8s 集群那样，可以设置节点标签或者其他的方式，但是对于读取数据的时候，该参数可以做到一定程度避免读取热节点的数据，这个参数有 5 种情况可以选择，1 是默认值，就是根据 hash 来读取其中一个 brick，而 3 则可以找到具有较少没有完成的读请求的 brick 来进行操作，也就是这里会考虑读请求队列的正

在运行的任务情况。而 4 则是考虑网络延迟的，5 则是介于 3 和 4 之间的一个方法。

5. 数据库相关

对于数据库，其实是不推荐使用 glusterfs 作为存储的，因为 glusterfs 相比起 ceph 和 openEBS 这些存储方案，其独特的架构方式和元数据管理方式，会让 k8s+mysql+glusterfs 的组合，使用效率会显得非常低。当然，对于数据库使用 glusterfs 作为存储的话，因为数据库本身是有应用缓存的，因此这里需要进行一些参数优化，主要就是避免使用 io cache 那些，尽量使用直接 IO 那样的方式，加快读写速度，因此下面有一些参数是可以进行参数设置的。

参数名称	参数值
performance.stat-prefetch	off
performance.read-ahead	
performance.write-behind	
performance.readdir-ahead	
performance.io-cache	
performance.quick-read	
performance.open-behind	on
performance.strict-odirect	

那么这里对于一些参数，如 write-behind，如果写入请求很小，它还会合并连续写入以形成更大的缓冲区，以便它可以一次性发送所有数据。所以 write-behind 的目的是减少 fuse 写路径中网络写的次数。

6. min-free-disk

这个参数是一个比例,brick 对应的节点的磁盘剩余空间比例,默认是 10%。如果是多个 subvolume ,其中某个 subvolume 如果空间达到这个阈值,然后在 gluster 的 client 创建文件夹 那么新的文件夹下这个 subvolume 对应的 dht 的 hash 值就是 0,所有以后在这个文件夹下创建的文件都不会落到这个 subvolume。这个参数是一个优化的参数,但是同时这样的优化也会带来一定的麻烦,那就是当空间快满了以后,调用链会变长了,会导致创建文件变慢,系统表现地比较卡了。

5.1.3. 日志相关

在日常运维过程中,有时候会遇到一些问题,那么正常的日志输出信息是不够的,因此需要 debug 一些日志信息的时候,就需要用到下面两个参数了。

参数	作用
diagnostics.brick-log-level	控制 brick 端的日志级别,影响 glusterd 和 brick 日志输出
diagnostics.client-log-level	控制 client 日志输出级别

5.1.4. heal 相关

关于 heal 自愈,属于一个低优先级的功能,同时默认中每个节点都是一个 heal 进程的,因此这里如果想加快 heal 自愈的速度,那么下面有一些参数可以进行调整。

1. cluster.shd-max-threads 和 cluster.shd-wait-qlength

这两个参数的作用就是设置最大的 glustershd 的进程数量,也就是每个节点的 heal

进程数据，还有每个进程处理的队列长度。另外这里还可以手动进行增加 shd 进程，这里的命令类似如下所示。

```
1. root@gfs01:~# /usr/sbin/glusterfs -s localhost --volfile-id gluster/glustershd -p /var/run/gluster/glustershd/<glustershd-pidfile> -l /var/log/glusterfs/<glustershd-logfile> -S /var/run/gluster/<glustershd-socketid>.socket --xlator-option *replicate*.node-uuid=<gluster-node-uuid>
```

下面的这几个内容是需要进行替换的

- ① <glustershd-pidfile>：新的存放 pid 信息的文件路径
- ② <glustershd-logfile>：新的保存日志的地方
- ③ <glustershd-socketid>：socket 文件,这里 id 是可以使用 uuidgen 命令获得
- ④ <gluster-node-uuid>：节点的 uuid 信息，这里可以通过 peer status 得到

那么实际执行类似下面这样的，然后这里再使用 ps -ef |grep glustershd 就可以看到多出来一个进程了。

```
1. root@gfs01:~# /usr/sbin/glusterfs -s localhost --volfile-id shd/test-replica -p /var/run/gluster/shd/test-replica/test-replica-shd-01.pid -l /var/log/glusterfs/glustershd-01.log -S /var/run/gluster/53fc946f9b1443bf.socket --xlator-option *replicate*.node-uuid=f1c72979-3502-41fb-83f2-45f9d2e47cc2 --process-name glustershd-01 --client-pid=-6
```

另外这里注意的是，进程数和链接数是无关的，也就是说当这里的进程增加之后，新的进程并不会再和所有的 bricks 相连接。

2. cluster.data-self-heal-algorithm

这是选择 heal 的算法，有两个选项，分别是 full 和 diff,区别就是是否全量校验，默认情况下是 full,而使用 diff 的话则可以加快自愈的速度，因为会减少一部分检查。

5.2. linux 系统优化

对于 linux 系统, Glusterfs 因为基于 fuse 开发的, 而 IO 和文件系统的一些参数配置优化, 有时候可以明显提高系统性能, 因此下面来简单了解一下。

1) vm.swappiness

这个参数是控制如何使用 swap 分区的, 对于很多分布式系统, 大多数建议是直接禁用 swap 分区, 避免因为使用该分区而导致性能下降, 因此建议对于该参数可以考虑设置为 0。

2) vfs_cache_pressure

该文件表示内核回收用于 directory 和 inode cache 内存的倾向; 缺省值 100 表示内核将根据 pagecache 和 swapcache, 把 directory 和 inode cache 保持在一个合理的百分比; 降低该值低于 100, 将导致内核倾向于保留 directory 和 inode cache; 增加该值超过 100, 将导致内核倾向于回收 directory 和 inode cache。因此这里建议可以考虑将该值设置大于 100。

3) vm.dirty_background_ratio 和 vm.dirty_ratio

两者中的第一个 (vm.dirty_background_ratio) 定义了后台将页面刷新到磁盘之前可能变脏的内存百分比。在达到此百分比之前, 不会将页面刷新到磁盘。但是, 当刷新开始时, 它会在后台完成, 而不会中断前台的任何正在运行的进程。

现在, 两个参数中的第二个 (vm.dirty_ratio) 定义了强制刷新开始之前脏页可以占用的内存百分比。如果脏页的百分比达到这个阈值, 那么所有进程都变得同步, 并且在它们请求的 io 操作实际执行并且数据在磁盘上之前不允许它们继续。在高性能 I/O 机器的情况

下，这会导致问题，因为数据缓存被切断并且所有执行 I/O 的进程都被阻塞以等待 I/O。

这将导致大量挂起进程，从而导致高负载，从而导致系统不稳定和糟糕的性能。

4) scheduler

这个参数是/sys/block/{DEVICE-NAME}/queue/scheduler 的配置，这里设置的是 IO 调度算法，操作系统中大部分默认的是 none。

```
1. $ sudo cat /etc/issue
2. Ubuntu 18.04.5 LTS \n \l
3.
4. $ sudo cat /sys/block/sda/queue/scheduler
5. [mq-deadline] none
```

这里默认的调度是 none,那么这里可以考虑 deadline 算法，该算法主要考虑调度任务饥饿问题，可以根据业务和 IO 情况来进行调整。

5) nr_requests 和 queue_depth

nr_requests 是请求的 IO 调度队列大小,queue_depth 是请求在磁盘设备上的队列深度,I/O 调度器中的最大 I/O 操作数是 nr_requests * 2。操作系统中 nr_requests 参数，可以提高系统的吞吐量，似乎越大越好，但是该请求队列的也不能过大，因为这样会消耗大量的内存空间。该值的调整需要综合多处因素。

章节语:

这一章主要关注了 glusterfs 的一些参数和 linux 中的系统优化配置，对于这两部分的内容，因为涉及内容比较多，需要经过大量测试和了解原理后建议才在生产环境中使用。

6. 第六章 运维之路

6.1. 难受的运维经历

对于 glusterfs 的使用,在生产环境中使用的版本是 7.5,而到了写下这段话的时候,目前最新的版本已经是 9.2 了,短短的一两年时间内,glusterfs 的版本迭代非常的迅速,同时也有了很大的优化改进,而因为生产环境的追求稳定,加上当时对 glusterfs 的熟悉程度不够,还有使用场景的特殊性,遇到了很多惨痛的生产环境问题和棘手的需求,下面分享其中一些故事和事故,希望能够给大家在日常运维使用过程中规避一些问题。同时如果你是准备使用 glusterfs,那么能够很好地规划与设计集群的使用。

1) 挂载突然掉线

这个问题是当时生产环境在使用 k8s+glusterfs+heketi 的时候遇到的,时不时会出现挂载掉线的问题,挂载点提示 transport endpoint is not connected,而容器 pod 也并不会因为这个而重启,并且已经无法正常读写数据了,但是这时候 volume 状态是正常的。

后来通过排查日志和官方的 issue 发现,这里是因为存在了一个版本 bug 导致的,这里出现问题的原因是以前实现的时候,文件句柄在使用时并没有添加引用到 fd_t 这个结构中,导致可能还在被使用的连接被销毁。这个问题如果感兴趣的话,可以查看 issue 为 1225 的文档^[1],这里给出了相关的内容。

这个问题在 7.7 以前的版本中会存在,对于旧版本的集群,可以通过关闭 open-behind 这个参数来进行规避。

2) brick 进程突然收到关闭信号

这个问题是一个很有趣的问题,当时是计划打算把 minio 放到 k8s 中运行,然后底层

存储使用 glusterfs 的，而在测试过程中，突然发现有時候 volume 的 brick 进程会异常，然后通过排查日志发现，在服务端会收到客户端发送的日志，其中关键是有段信息中包含 Shutting down connection,显示这个是从客户端发送过来的。后来通过排查客户端的日志，包括查看了 pvc 的日志(日志的位置则是在 k8s 节点中的，前面提到过)，但是并没有发现发送相关信息。接着通过去咨询官方才发现，这是一个遗留问题^[2]，同时这个问题在比较早之前 minio 官方的 issue^[3]中也曾经有其他使用者提到过,然而目前并没有找到具体的场景来准确复现，因此属于一个长期遗留的问题。而 glusterfs 官方为了进一步排查这些问题，在最新的 9.x 版本中，添加了 core dump 相关的内容^[4]，方便进一步排查。

因此后续中就不再考虑把 minio 放在 k8s 中使用了，而其他的应用服务，也暂时没有遇到过该问题。

3) 应用服务对 glusterfs 的不支持

在生产环境中，使用的架构是 k8s+glusterfs+zfs 的模式，而很多服务也是部署在 k8s 上面，在使用一些服务的过程中，调研发现部分服务官方是明确提出不太建议使用 glusterfs 作为存储后端的，这里 nexus 官方的文档中也有提到这个问题^[5]。因此在日常运维使用过程中，对于服务组件的要求，也是需要调研清楚的。

4) 集群节点 peer 状态异常

在一次日常运维集群中，因为集群的部分节点被异常重启，接着检查发现，其中有一个节点 A 出现异常了，使用 peer status 的时候发现，对另外一个节点 B 的连接状态显示为 peer is connected and accepted，后来检查日志，发现这里是 peer 状态元数据信息不一致，异常的节点 A 的文件/var/lib/glusterd/peers，记录的 B 节点的文件中的 state 和其他节点是不一样的。接着通过咨询官方发现^[6]，这里是当前开启了 quota 功能之后，

7.8 以前的版本存在 checksum 异常的 bug，后续通过修改这个异常问题，同时检查 volume 元数据是否一致，重启该异常节点的 glusterd 进程解决的。

5) glusterfsd 进程连接所有的 brick 进程

这里其实属于一个优化内容，因为以前的设计和实现，目前对于每个节点上的 glustershd 进程，都会去连接所有 brick，哪怕当前 brick 不是该节点的，但是对于该 glustershd 进程来说，因为 brick 不在该节点上，因此却又不做自愈等操作，属于一个多余的连接情况，这一点目前也提了一个 issue 给官方^[7]，后续待官方修复该问题。

6) zfs+glusterfs 的使用在大量文件下 heal 速度很慢

在生产环境中，这里使用的是 zfs+glusterfs+k8s 的架构，而当 volume 数量逐渐增加，并且有些 volume 存放了一些小文件之后，可以明显感受到 heal 的速度非常慢，而官方也有使用者提出如果是 zfs+glusterfs，使用 xfs 的文件格式的话，这个问题会比较明显，而目前来说，关于 zfs 文件系统对于小文件的支持，也不是特别理想。当然对于小文件的支持问题，这也是目前工业界的一个难题。

7) 节点负载过高影响集群稳定性

这个问题的出现很有意思，生产环境的物理节点其中有一台出现了很高的负载，64 核的物理机，top 命令显示的 1 分钟和 5 分钟平均负载高达三百多，直接导致了这台机器无法远程登录，也无法进行任何的命令操作，但是这时候还没有导致机器宕机，也就是说其他 glusterfs 集群节点使用 peer status 显示还是正常的，但是这时候，如果 volume 需要同步信息的话，是需要所有节点都有响应的，会导致集群中其他的 volume 服务会受到严重的影响。

对于这个问题，根本原因就是因为在 glusterfs 的无中心架构设计，所有的 volume 信息是每个节点都需要同步的，所以如果一旦出现某个节点负载很高响应很慢，这样对整个集群的影响是非常大的，这一点需要特别留意的。

6.2. 周边生态项目

在使用一个工具的时候，除了当前工具的功能与特性以外，还要考虑工具的周边生态项目，如果周边生态项目的活跃度，也会直接影响到当前项目的使用量与便捷性的，而对于 glusterfs 来说，这里也有一些相关的生态项目，下面简单分享一些使用和遇到的项目。

1) heketi 和 kadalu

在使用 k8s+glusterfs 的架构中,k8s 管理 glusterfs 的 volume 工具，可以使用 heketi 来进行管理,而 heketi 的原理，其实是对块设备进行格式化，做成 lvm2，弄成 vg，而有 pod 要申请 volume 的时候，则从 vg 中创建 lv，然后格式化成 xfs 文件系统格式，接着挂载到系统的目录上面使用，对于这种方式，优缺点是非常明显的。优点是使用这方式创建的 volume,非常方便进行容量监控，因为底层的 vg 信息创建出来的 lv 已经限制了容量，在挂载之后，可以直接使用 exporter 进行检测。同时使用这种方式进行创建的 volume,可以很方便地使用快照功能。

当然这样做的缺点也是显而易见的，首先如果一旦 volume 容量不足，因为底层是 lvm2 的，因此 heketi 的扩容是再次从 vg 中创建一个新的 lv 出来，并且再次格式化挂载，对原来的 volume 进行 add-brick 操作，但是对于 volume 来说则需进行数据的 rebalance 操作，而 rebalance 操作则是一个不确定性非常大的操作，因为对于

容量很大的 volume,或者小文件非常多的 volume,数据的重平衡rebalance操作时间是没有办法准确预估的。曾经在测试环境中对 nexus 进行扩容操作,两百多 G 的数据重平衡花费了两个多小时,当然这个时间可以通过设置一些参数来减少,但是仍然不是一个短时间内可以完成的操作。另外对于重平衡后的 volume,没有办法完全做到数据的平均分布的,尤其是对于一些数据大小并不是一致的时候,有出现数据倾斜的风险。

其次对于使用块设备进行格式化为 lvm2 的话,这里如果多个服务都是部署在当前的块设备上面,那么对于节点的读写负载是比较高的,同样会影响到其他服务的使用,而 heketi 为了解决这个问题,有一个 tag 标签的功能,可以对不同的块设备在格式化之后打上 tag 标签,然后 storageclass 可以指定对应的标签的块设备来进行创建 volume,实际上就是一个分组的功能,因此这里也需要额外去规划好块设备的分组问题。

另外对于 heketi 来说,因为目前官方已经暂时停止更新了,从最新的 10.3 版本以后,基本上没有了更新,目前官方也不再推荐使用该工具了,而是推荐使用 kadal 这个相对轻量级一点的工具。相比起 heketi,kadal 属于一个非常新的工具,在写下目前这段话的时候,目前也并没有 1.x 版本的出现,kadal 目前最新的是 0.8.3 版本,因此建议可以继续跟进了解。而 kadal 的使用其实是放弃了块设备进行格式化为 lvm2 的方式,采用了类似直接使用命令 create volume 的方式,还有手动创建之后,挂载到 pod 等方式,会更加直接简单方便管理。

2) ganesha

Nfs-ganesha 是 NFS v3、4.0、4.1 和 4.2 的用户模式文件服务器,考虑使用这个项目的原因为,是出于对 volume 的安全管理,因为对于 glusterfs 集群来说,只要知道了 volume 和任意一个节点的 ip,那么就可以随意挂载并且使用了,只要防火

墙允许，因此对于生产环境来说，如果一个 volume 是一个团队使用的话，那么权限管理就比较混乱和麻烦了，而使用 ganesha 项目则可以隐藏 volume 信息。当然因为 ganesha 项目是基于 NFS 的，而 NFS 对于读写性能的损耗也是比较大的，因此如果生产环境使用的话，建议多进行压测。

6.3. 未来还能做什么

Glusterfs 作为无中心架构的分布式文件系统代表，目前来说经历过很多版本的迭代发展，在性能和架构方面有了很大的改变，然而需要优化和改进的地方其实也是很多的，其中一点就是对于很多小文件的读取问题，在测试环境中经过测试，volume 创建了上万个文件以后，执行 ls 等命令都会比较慢，这是因为目前对于文件的元数据信息缓存，并没有做的很好，而国内的 TaoCloud 公司团队，通过给 server 端加入了一个 dcache 层^[9]，仅将目录结构信息保存于内存当中，而将元数据保存到嵌入式 KV 数据库 levelDB 当中。这种架构方案的实现，可以大幅度提高文件元数据的读取速度，关于这个架构方案的实现，建议可以具体阅读一下 TaoCloud 团队的技术博客文档。

那么除了这个以外，glusterfs 官方提到了一个 RIO 的技术实现方案^[10]，这个方案的核心就是考虑把 volume 的元数据管理和真实数据信息两部分拆开，因为有些 fop 操作是只需要访问元数据的，有些则是两者都访问等，而拆开之后，对于元数据管理部分，可以考虑引入其他的工具进行管理和缓存了，这样未来对于扩展 glusterfs 的架构层次也有了更大的空间，另外关于这个方案的实现思路，可以查看文档^[11]。

章节语:

这一章主要关于了 glusterfs 运维过程中遇到的一些问题，和 glusterfs 配合使用的一些相关生态项目，还有未来 glusterfs 的一些架构发展等，对于 glusterfs，目前来说不太建议将数据库、confluence 和 gitlab 这些作为后端存储使用，而 glusterfs 更加适合那种冷存储数据系统，或者是很大的文件存储系统使用。同时作为运维和开发团队来说，如果使用 glusterfs，建议需要考虑团队技术熟悉程度，否则无中心架构的分布式存储系统，特点与常见的 hdfs 那些有着很多不同的地方，系统规划和设计方面要考虑的内容也会不一样。

附录引用:

- [1] <https://github.com/gluster/glusterfs/issues/1225>
- [2] <https://github.com/gluster/glusterfs/issues/2473>
- [3] <https://github.com/minio/minio/issues/4993>
- [4] <https://github.com/gluster/glusterfs/issues/1810>
- [5] <https://help.sonatype.com/repomanager3/installation/system-requirements#SystemRequirements-FileSystemsToAvoid>
- [6] <https://github.com/gluster/glusterfs/issues/2498>
- [7] <https://github.com/gluster/glusterfs/issues/2594>
- [8] <https://github.com/gluster/glusterfs/issues/11764>
- [9] <https://mp.weixin.qq.com/s/qaRGjUtjG1JUsz43ktlwTw>
- [10] <https://github.com/gluster/glusterfs/issues/243>
- [11] https://docs.google.com/document/d/1KEwVtSNvDhs4qb63gWx2ulCp5Gjige77NGJk4p_Ms4Q/edit#
- [12]