# 简介

**GlusterFS 是一个可扩展的网络文件系统**，采用堆叠式设计，适用于云存储和媒体流等数据密集型任务。它将来自多个服务器的磁盘存储资源聚合到单个全局命名空间中。

**具有以下优势**：

可扩展到数PB 大小

同时处理数以千计的客户端

POSIX兼容

使用相对便宜、货源充足且可替换的商用硬件

可以使用任何支持扩展属性的磁盘文件系统

可使用 NFS 和 SMB 等行业标准协议访问

提供副本、配额、异地副本、快照和比特轮检测

允许针对不同的工作负载进行优化

开源

相关参考链接：

主页：<https://docs.gluster.org/en/latest/>

快速安装： <https://docs.gluster.org/en/latest/Quick-Start-Guide/Quickstart/>

GlusterFS 的配置和管理：<https://docs.gluster.org/en/latest/Administrator-Guide/>

github主页：<https://github.com/gluster/glusterfs/tree/master/doc/>

博客：

<https://blog.csdn.net/u010129347/article/details/43955397>

<https://blog.csdn.net/wangyuling1234567890/article/details/24601417>

<https://www.cnblogs.com/brucewoo/archive/2012/04/26/2472059.html>

<https://www.cnblogs.com/brucewoo/archive/2012/04/28/2475088.html>

<https://www.cnblogs.com/brucewoo/archive/2012/05/07/2489248.html>

<https://www.cnblogs.com/brucewoo/archive/2012/05/08/2490826.html>

<https://blog.csdn.net/qq_38918259/article/details/80476464>

<https://blog.csdn.net/qq_38918259/article/details/80477257?spm=1001.2014.3001.5501>

<https://blog.csdn.net/u010129347/article/details/43955397>

<https://blog.csdn.net/liuhong1123/article/details/8118174>

<https://www.cnblogs.com/smarty/p/4040605.html>

# 安装实践

## 概念名词

trusted pool：信任池，包含gfs集群中所有主机

node or“server”：指代trusted pool中的任意服务器

brick：存储设备

export：brick挂载路径，例如： /export/brick1

Global Namespace

Gluster volume：一个或者多个brick的集合

## 快速安装

1. 准备三台机器。这里我在vmware上创建了三台虚拟机并安装centos7.9系统，同时添加了一块5GB的磁盘单独用来GlusterFS存储
2. **格式化物理磁盘并挂载**：

|  |
| --- |
| # 输入n进行分区，一路回车，w写入修改并退出  fdisk /dev/sdb  # 格式化  mkfs.xfs -i size=512 /dev/sdb1  # 创建挂载目录  mkdir -p /data/brick1  # 加入启动挂载选项配置文件  echo '/dev/sdb1 /data/brick1 xfs defaults 1 2' >> /etc/fstab  # 挂载  mount -a && mount |

1. 各节点分别**安装并启动** glusterfs

|  |
| --- |
| yum -y install centos-release-gluster  yum -y install glusterfs glusterfs-server glusterfs-fuse --skip-broken  service glusterd start  service glusterd status |

1. **配置防火墙**，允许接收流量

|  |
| --- |
| iptables -I INPUT -p all -s <ip-address> -j ACCEPT  # 或者暴力点  iptables -F |

1. **配置信任池**

先编辑/etc/hosts文件，设置ip和hostname，举例：

|  |
| --- |
| echo -e "192.168.18.128 node1 \n192.168.18.129 node2 \n192.168.18.130 node3" >>/etc/hosts |

node1

|  |
| --- |
| gluster peer probe node2  gluster peer probe node3 |

node2

|  |
| --- |
| gluster peer probe node1 |

node3

|  |
| --- |
| gluster peer probe node1 |

查看状态

|  |
| --- |
| gluster peer status  node1    node2    node3 |

可以在任何一个主机进行节点删除

|  |
| --- |
| gluster peer detach node1 |

1. **配置GlusterFS卷**

所有节点

|  |
| --- |
| mkdir -p /data/brick1/gv0 |

任选节点

|  |
| --- |
| gluster volume create gv0 replica 3 node1:/data/brick1/gv0 node2:/data/brick1/gv0 node3:/data/brick1/gv0 |

查看刚刚创建好的卷

|  |
| --- |
| gluster volume info |

启动卷（start delete stop）

|  |
| --- |
| gluster volume start gv0 |

1. **测试**

取任一节点进行挂载

|  |
| --- |
| mkdir /mnt/gfs  mount -t glusterfs node1:/gv0 /mnt/gfs/ |

创建文件

|  |
| --- |
| for i in `seq -w 1 100`; do cp -rp /var/log/messages /mnt/gfs/copy-test-$i; done |

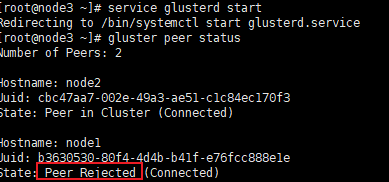
查看结果

|  |
| --- |
| 挂载节点：  ls -lA /mnt/gfs/copy\* | wc -l    其他节点：  ls -lA /data/brick1/gv0/copy\* | wc -l    由于创建卷选定的replicate类型，所以所有节点均有100个文件，否则约为33个 |

**遇见的问题：**

长时间后出现node1被排除到集群之外（或者需要更换节点）



、

此时需要修改node1的uuid：

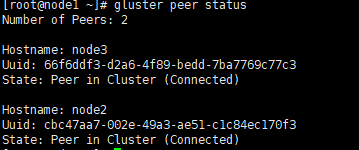
首先清空vim /var/lib/glusterd/目录下除了glusterd.info外的文件

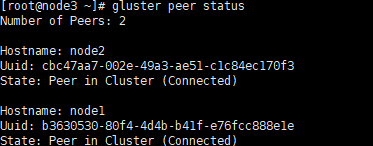
编辑vim /var/lib/glusterd/glusterd.info

将其他节点观测到的node1的uuid写入



然后重启服务，**配置信任池**





其他问题排查参考思路：

<https://docs.gluster.org/en/latest/Troubleshooting/troubleshooting-glusterd/>

## 源码安装

|  |
| --- |
| tar zxf glusterfs-3.4.1.tar.gz  cd glusterfs-3.4.1  ./autogen.sh  ./configure –prefix=/opt/soft/glusterfs –enable-systemtap  make -j4  make install |

# 重要特性

## 无元数据

传统分布式文件系统通常设置元数据服务器，进而管理文件与数据块间的存储映射关系。

而GlusterFS使用弹性哈希算法，去除元数据的使用，任何服务器和客户端都可以利用哈希算法、路径及文件名进行计算，从而定位数据位置。

这一改变提升了系统的可靠性，避免元数据服务器损坏导致的整个集群不可用，另外通过文件名查找位置将变得非常快，不过**只能是以文件为存储对象，且列出文件或目录的效率则会降低**

## 服务器对等

GlusterFS没有元数据，并且它将集群配置信息保留到每个节点服务器，不再用master这样的集中管理方式。

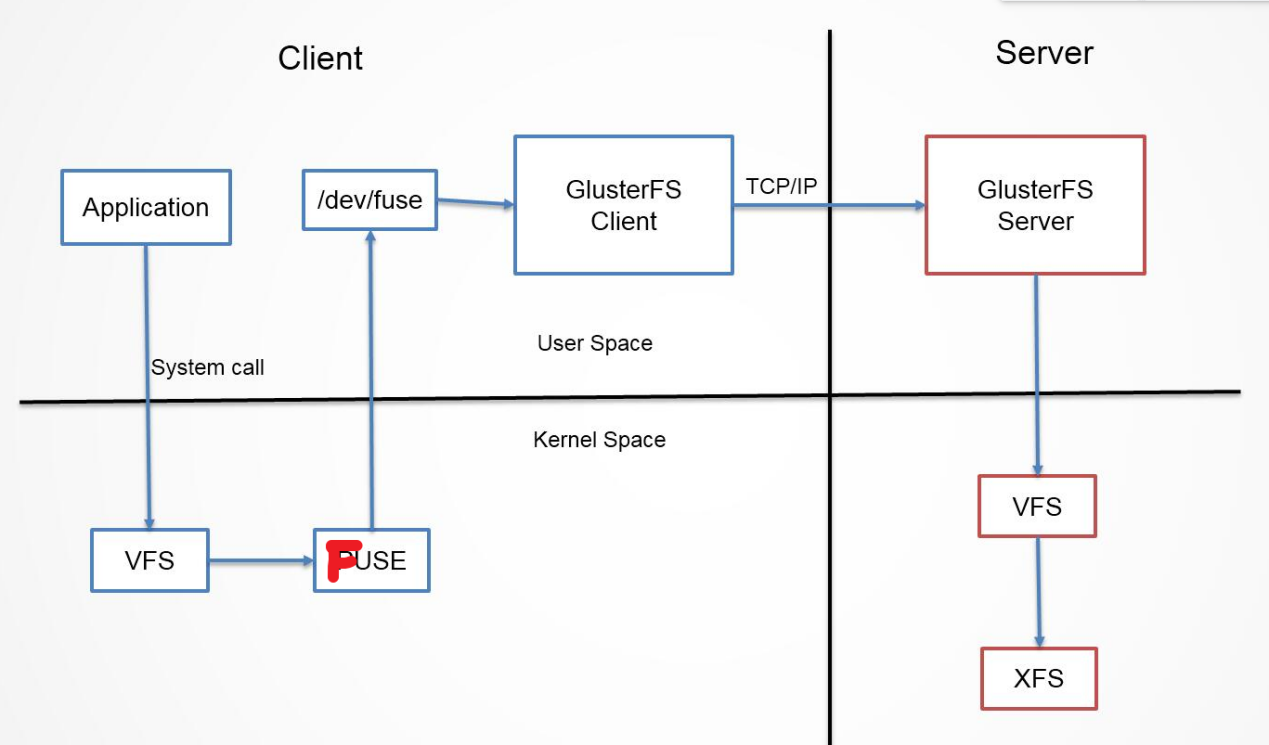
集群里每个服务器都是对等的，通过互相通信保持一致性，避免了master故障带来的整体不可用。

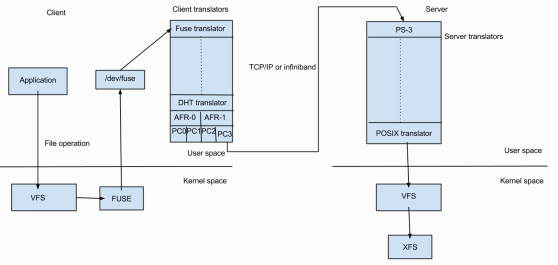
不过代价是**节点过多时信息同步效率下降**。

# 客户端访问流程

GlusterFS支持三种客户端类型。Gluster Native Client、NFS和CIFS。Gluster Native Client是在用户空间中运行的基于FUSE的客户端，官方推荐使用Native Client，可以使用GlusterFS的全部功能。

**下面是Native Client访问流程**：





# 源码阅读

## 源码目录

|  |
| --- |
| cli ：命令行接口  contrib ：包含用到得第三方库源码。例如 MD5， fuse， uuid 等  doc ： glusterFS 的一些文档和配置例子  extras ：包含 glusterFS 配置的一些脚本信息  glusterfsd ：包含了 glusterfsd 工程的相关源码和 Makefile 信息，其用到了文件夹 libglusterfs 提供的库函数，并根据卷配置信息动态载入（dlopen， dlsym） xlator文件夹提供的相关 translator 库  libglusterfs：为 glusterfsd 以及 xlator 下的相关 translator 提供库函数  xlators ：包含所有用到的 translator 库。下面根据类型又分为 cluster ， debug ， performance ， mgmt 等子文件夹。 glusterFS 各个应用程序根据自己的卷配置信息动态载入相关的 translator 库  rpc ：包含 glusterFs 应用程序之间通信用到的库函数。例如 socket 监听和连接等操作  configure.ac ：根据 autoscan 生成信息修改而成的配置信息。生成 configure 需用到该脚本。如果向该 glusterFS 添加子工程需要动态修改该配置信息  autogen.sh ：该脚本利用 automake 等工具生成 configure 执行脚本。 运行时候需要安装 pkg-config 应用程序 |

## 四个主要模块

**gluster（CLI）**：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。

**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。

**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。

**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。

**注意：服务端(glusterfsd)，客户端(glusterfs)，管理程序（glusterd）为同一main函数入口，该函数入口简化如下：**

|  |
| --- |
| 1 int main (int argc, char \*argv[])  2 {  3 glusterfs\_ctx\_t \*ctx = NULL;  4 int ret = -1;  5 ret = glusterfs\_globals\_init ();//初始化一些全局变量和参数  6 ctx = glusterfs\_ctx\_get ();  7 ret = glusterfs\_ctx\_defaults\_init (ctx);//初始化一些glusterfs的上下文默认信息  8 ret = parse\_cmdline (argc, argv, ctx);//解析命令行参数  9 ret = logging\_init (ctx);//初始化日志文件  10 gf\_proc\_dump\_init();//初始化代表程序的全局锁  11 ret = create\_fuse\_mount (ctx);//创建fuse的主（根）xlator：mount/fuse，并且初始化相关值  12 ret = daemonize (ctx);//设置守护进程运行模式  13 ret = glusterfs\_volumes\_init (ctx);//初始化卷服务，创建相应的xlator并且初始化  14 ret = event\_dispatch (ctx->event\_pool);//事件分发，将相应的事件交给相应的函数处理  15 } |

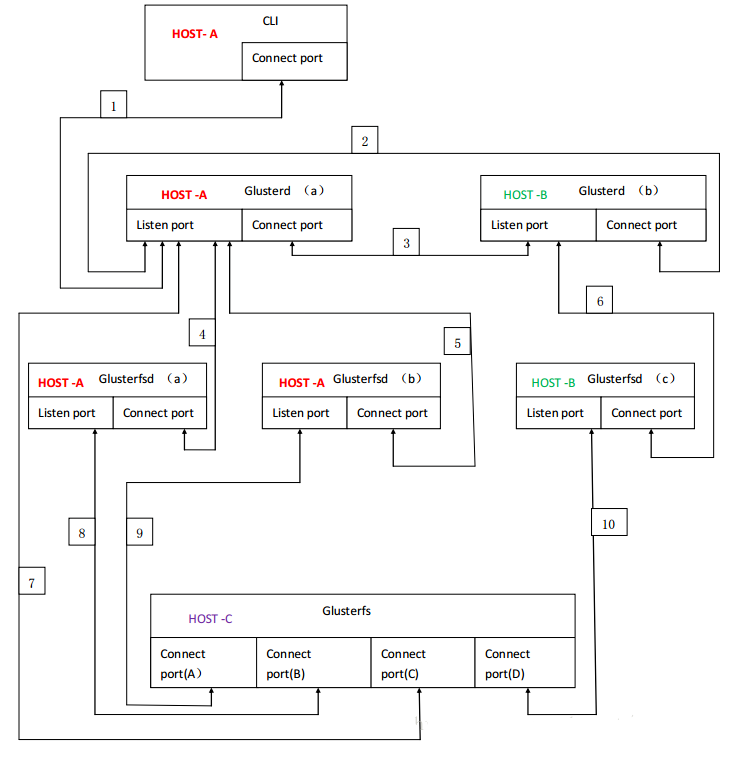
只要GlusterFS安装在服务器节点中，就会创建一个gluster管理守护程序（**glusterd**）二进制文件。此守护程序应该在群集中的所有参与节点中运行。启动glusterd后，可以创建一个包含所有存储服务器节点的可信服务器池（TSP）（TSP甚至可以包含单个节点）。现在可以将作为基本存储单元的块创建为这些服务器中的导出目录。来自这个TSP的任何数量的块都可以组合在一起形成一个卷。

创建卷后，**glusterfsd**进程将在每个参与的块中开始运行。除此之外，还将在/ var / lib / glusterd / vols /中生成称为vol文件的配置文件。卷中的每个块都有对应的配置文件。这将包含有关该特定块的所有详细信息。还将包含创建客户端进程所需的配置文件。现在我们的文件系统已经可以使用了。我们可以非常轻松地在客户端计算机上安装此卷，如下所示，并像使用本地存储一样使用它：

mount.glusterfs `<IP or hostname>`:`<volume\_name>` `<mount\_point>

IP或主机名可以是可信服务器池中创建所需卷的任何节点的IP或主机名。

当我们**在客户端中安装卷时，客户端glusterfs进程与服务器的glusterd进程通信**。服务器glusterd进程发送包含客户端转换器列表的配置文件（vol文件），另一个包含卷中每个块的信息，客户端glusterfs进程现在可以直接与每个块的glusterfsd进程通信。



1 号线为 CLI 与 gluserd 之间 command 连接，通过该连接 cli 向 glusterd 发送命令，

glusterd 执行譬如 peer， create， start 等命令。

2 号和 3 号线为为不同 host 端 gluserd 之间的连接，用来同步命令等相关操作。

4 号， 5 号， 6 号线为服务端 gluserfsd 与同一 host 端的 glusterd 的连接，用来从 glusterd获取该服务对应的服务端卷信息。

7 号线为客户端 glusterFS 与命令操作 host glusterd 之间的连接，用来获取该客户端

对应的客户端卷文件信息

8 号， 9 号， 10 号线为 protocol/client 和 server 的 translator 的连接，每个 brick 一

个连接，用来执行相应的文件操作

## 内存分配与使用

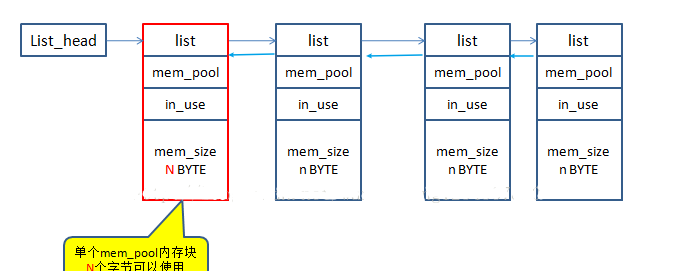
**内存分配主要有两种方式，一种是内存池分配，一种是普通内存分配**

对应文件libglusterfs\src\glusterfs\mem-pool.h，libglusterfs\src\mem-pool.c

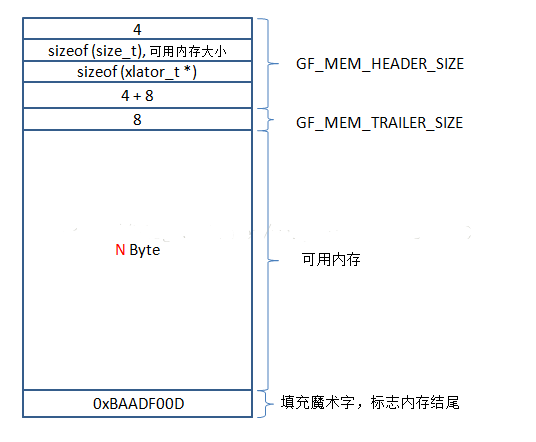
**内存池结构：**

|  |
| --- |
| struct mem\_pool {  /\* object size, without pooled\_obj\_hdr\_t \*/  unsigned long sizeof\_type;  unsigned long count; /\* requested pool size (unused) \*/  char \*name;  char \*xl\_name;  gf\_atomic\_t active; /\* current allocations \*/  #ifdef DEBUG  gf\_atomic\_t hit; /\* number of allocations served from pt\_pool \*/  gf\_atomic\_t miss; /\* number of std allocs due to miss \*/  #endif  struct list\_head owner; /\* glusterfs\_ctx\_t->mempool\_list \*/  glusterfs\_ctx\_t \*ctx; /\* take ctx->lock when updating owner \*/  struct mem\_pool\_shared \*pool; /\* the initial pool that was returned \*/  }; |

mempool链表以双向链表形式存储



**普通内存分配结构**(申请N字节内存时实际申请内存结构)：



**关键函数：**

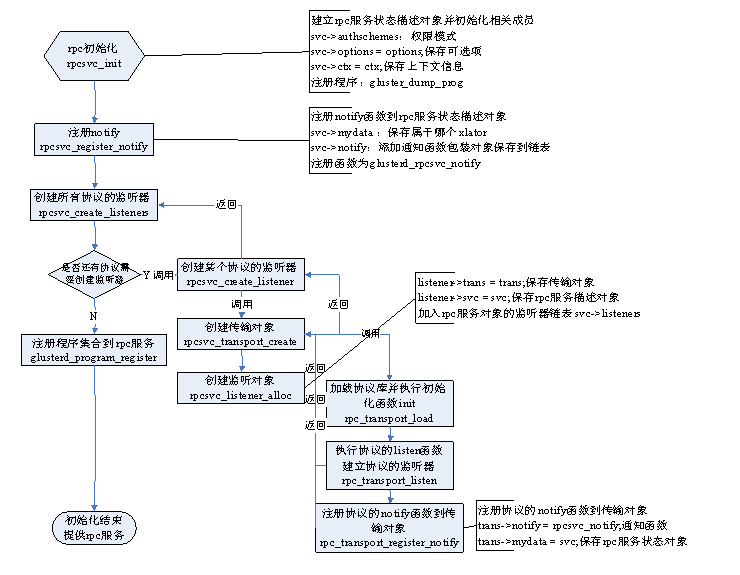
|  |
| --- |
| **mem\_pool\_new\_fn**  新建一个内存池对象，然后按照传递进来的内存的大小和个数分配内存  **mem\_get**  从内存池取出可用对象  **\_\_is\_member**  判断指针指向的内存是否是内存池的成员  **mem\_put**  将内存对象回收至内存池  **mem\_pool\_destroy**  销毁内存池 |

**封装后函数接口：**

|  |
| --- |
| #define mem\_pool\_new(type,count) mem\_pool\_new\_fn (sizeof(type), count, #type)  mem\_get0，mem-get0封装了mem\_get，做参数判断并且把需要使用的内存初始化为0  call\_stub\_destroy 封装mem\_put，摧毁存根，释放内存 |

## rpc模块

### 5.4.1 服务端实现原理



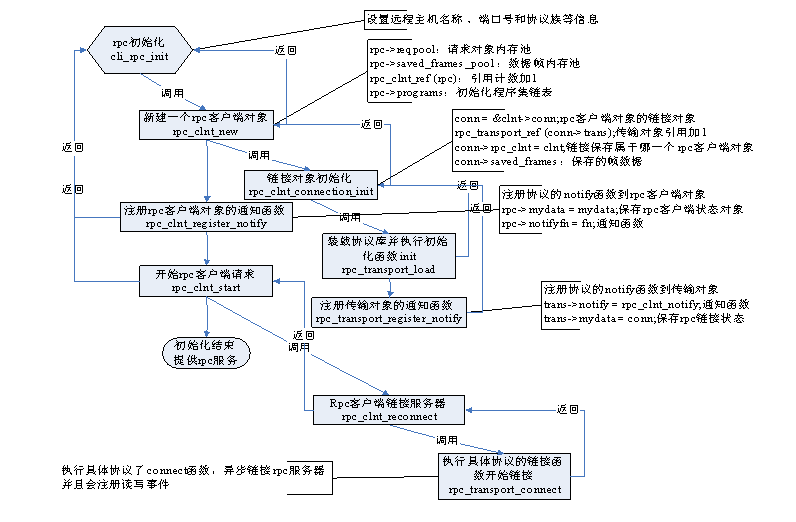
rpc\rpc-lib\src\rpcsvc.c：

|  |
| --- |
| rpcsvc\_init：为描述一个所有rpc服务的全局对象设置一些初始化的值，这些信息一直保存到整个rpc服务结束，也就是glusterfs集群管理程序等主进程结束  rpcsvc\_register\_notify：注册某些事件产生时的通知回调函数  rpcsvc\_create\_listeners：rpc服务建立，等待服务器连接，加载底层的通信协议，主要包括rdma和tcp  rpcsvc\_create\_listener：解析底层的传输类型，然后针对每一个解析出来的类型分别创建监听程序 |

rpc\rpc-lib\src\rpc-transport.c

|  |
| --- |
| rpc\_transport\_load：动态加载库函数到对应的对象中，最后执行这个传输对象的初始化函数init  rpc\_transport\_listen：执行了装载后具体的协议（rdma和tcp）的listen函数来开始监听客户端的请求  rpc\_transport\_register\_notify：注册的回调函数rpcsvc\_notify（把函数的地址保存到传输对象的notify成员中，把rpc状态描述对象保存到mydata中） |

### 5.4.2 客户端实现原理



cli\src\cli.c

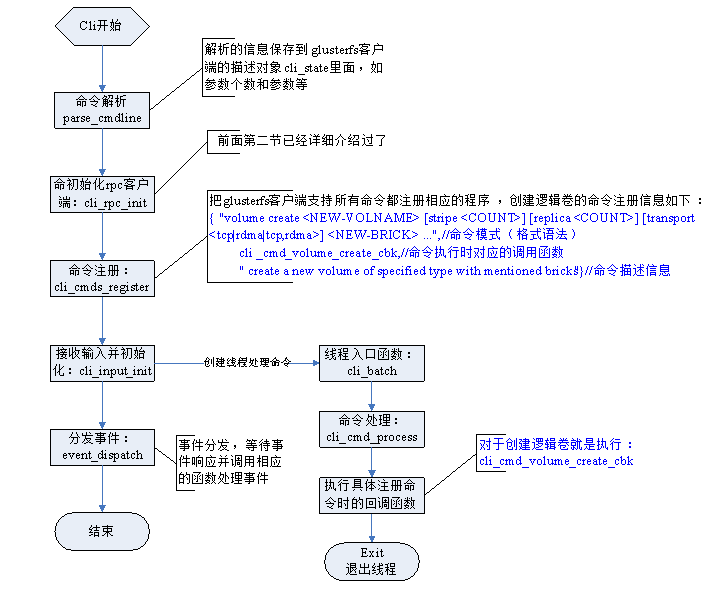
|  |
| --- |
| cli\_rpc\_init：客户端程序启动时建立rpc请求的初始化过程函数 |

rpc\rpc-lib\src\rpc-clnt.c

|  |
| --- |
| rpc\_clnt\_new：为客户端rpc对象分配内存，新建请求内存池，初始化rpc请求连接  rpc\_clnt\_connection\_init：主要完成协议库装载功能和链接选项的一些初始化功能  rpc\_clnt\_start：启动rpc  rpc\_clnt\_reconnect：此函数调用rpc\_transport\_connect 发起连接 |

### 5.4.3 rpc通信

客户端rpc请求提交准备过程流程图，以创建逻辑卷（volume）为例：



执行的函数是cli\_cmd\_volume\_create\_cbk，主要实现代码如下：

|  |
| --- |
| 1 proc = &cli\_rpc\_prog->proctable[GLUSTER\_CLI\_CREATE\_VOLUME];//从rpc程序表中选择对应函数  2  3 frame = create\_frame (THIS, THIS->ctx->pool);//创建帧  4  5 ret = cli\_cmd\_volume\_create\_parse (words, wordcount, &options);//创建逻辑卷的命令解析  6  7 if (proc->fn) {  8  9 ret = proc->fn (frame, THIS, options);//执行命令的回调函数  10  11 }  12  13 if (ret) {  14  15 cli\_cmd\_sent\_status\_get (&sent);//得到命令发送状态  16  17 if ((sent == 0) && (parse\_error == 0))  18  19 cli\_out ("Volume create failed");//如果失败，错误提示  20  21 } |

真正的处理函数：

|  |
| --- |
| [GLUSTER\_CLI\_CREATE\_VOLUME] = {"CREATE\_VOLUME", gf\_cli3\_1\_create\_volume} |

对于服务端，rpc服务启动的时候知道注册了监听事件，监听事件的处理函数：

|  |
| --- |
| socket\_server\_event\_handler：处理客户端的连接请求，然后在新的socket上注册可读事件（处理函数为socket\_event\_handler），并且执行通知函数做相应的处理 |

处理具体请求信息的实现是在\_\_socket\_proto\_state\_machine，分为几个阶段处理，中间不break

|  |
| --- |
| while (priv->incoming.record\_state != SP\_STATE\_COMPLETE) {//直到rpc服务记录状态完成为止  switch (priv->incoming.record\_state) {//根据现在rpc服务记录的状态做相应处理  case SP\_STATE\_NADA://开始状态  case SP\_STATE\_READING\_FRAGHDR://读取头部信息  case SP\_STATE\_READ\_FRAGHDR://处理已经读取的头部信息  case SP\_STATE\_READING\_FRAG://读取所有的数据  } |

处理完后调用通知函数rpcsvc\_notify（服务端传输对象初始化时注册）

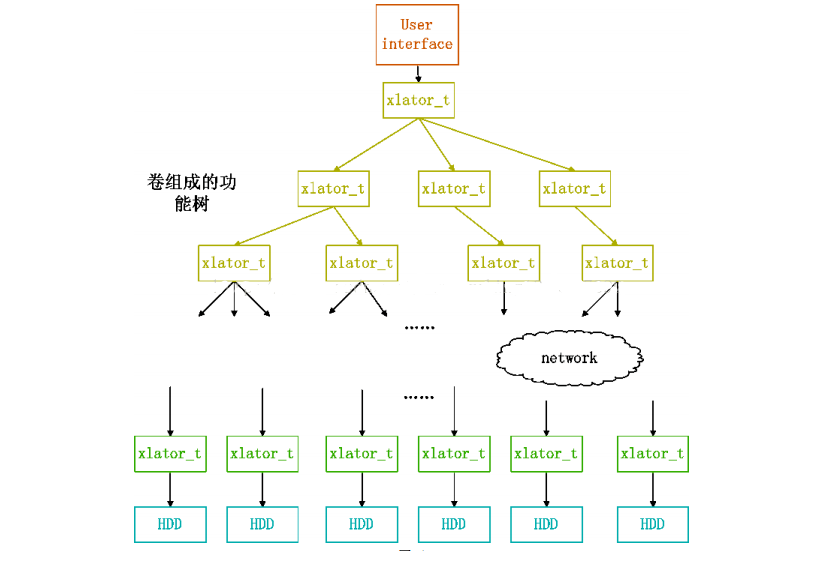
|  |
| --- |
| switch (event) {  case RPC\_TRANSPORT\_ACCEPT://rpc请求已经被接收处理  new\_trans = data;  ret = rpcsvc\_accept (svc, trans, new\_trans);//处理函数  break;  case RPC\_TRANSPORT\_DISCONNECT://断开连接消息  ret = rpcsvc\_handle\_disconnect (svc, trans);//处理函数  break;  case RPC\_TRANSPORT\_MSG\_RECEIVED://消息已经接收  msg = data;  ret = **rpcsvc\_handle\_rpc\_call** (svc, trans, msg);//rpc调用处理函数  break;    case RPC\_TRANSPORT\_MSG\_SENT://消息已经发生，不需要处理  break;    case RPC\_TRANSPORT\_CONNECT://已经连接  break;    case RPC\_TRANSPORT\_CLEANUP://清零消息  listener = rpcsvc\_get\_listener (svc, -1, trans->listener);//得到对应的监听器对象  rpcsvc\_program\_notify (listener, RPCSVC\_EVENT\_TRANSPORT\_DESTROY, trans);//通知上层  break;    case RPC\_TRANSPORT\_MAP\_XID\_REQUEST:  break;  } |

rpcsvc\_handle\_rpc\_call根据接收到的信息建立一个请求对象，然后根据建立的请求对象判断是都已经成功接纳此次rpc请求调用，如果是就继续执行函数rpcsvc\_program\_actor，这个函数会根据程序号、函数号等信息**查找对应的rpc请求的远程过程调用**，如果找到就执行相应的函数调用。

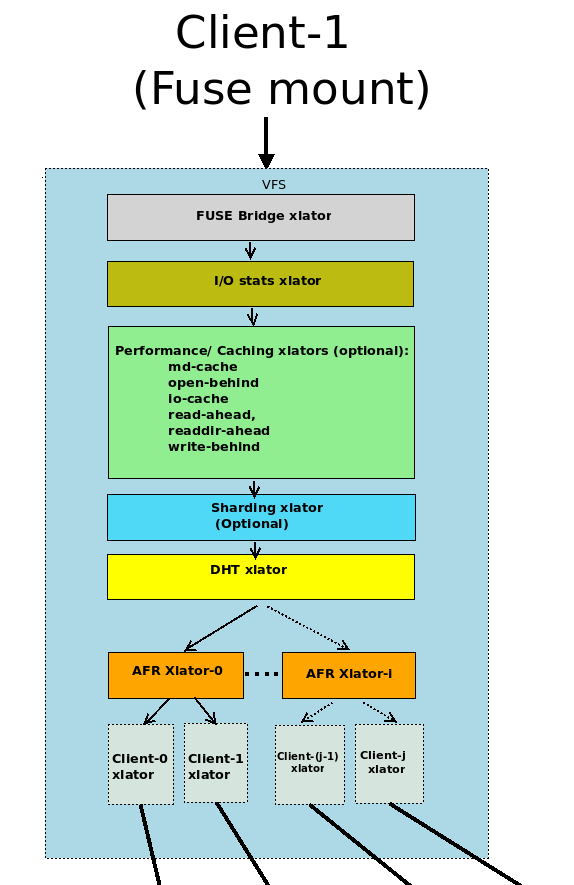
这里对应函数**glusterd\_handle\_create\_volume**，处理完后通知客户端。客户端会接收服务器端的回复，然后根据消息内容做相应的处理，如：创建成功等提示信息。这样一次完整的rpc通信就完成了。

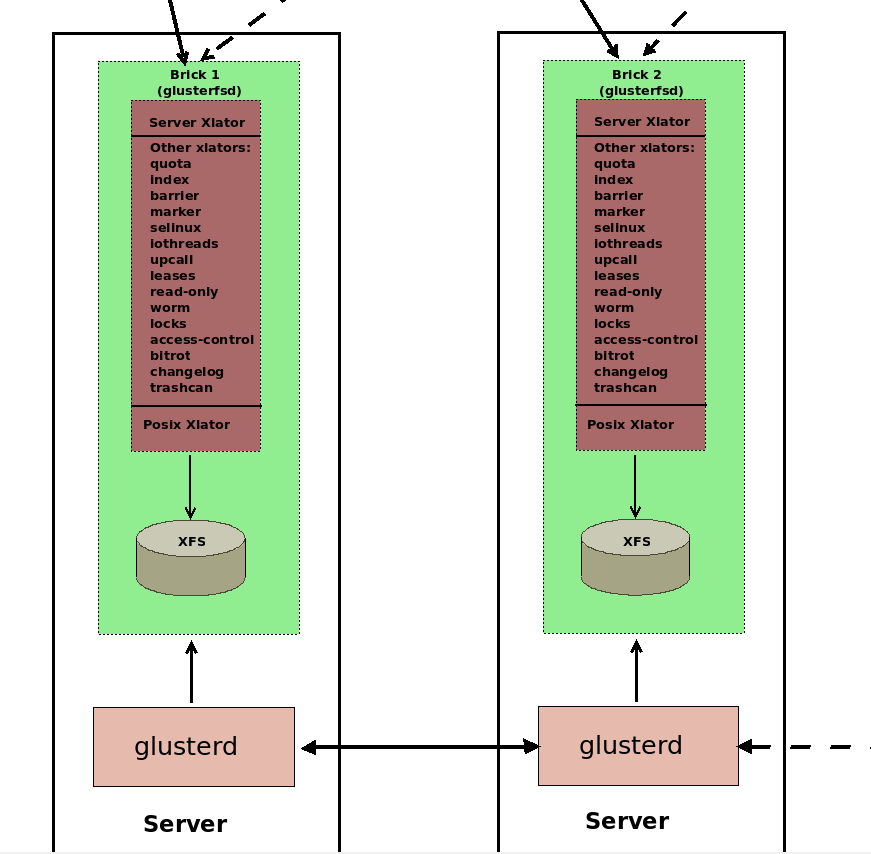
## xlator处理树

在 glusterfs 系统中引入了 Translator（翻译器）的处理机制，它在glusterfs的内部数据结构中称为 xlator\_t，多个Translator构成一颗功能树。



所有的子功能（io-cache, readahead, unify, stripe …）被以一个xlator\_t的结构表现在系统树中， 每一个 xlator\_t定义了自己的处理函数， 可以把 xlator\_t 理解为C++语言中的类。





xlator是glusterfs的核心，理解了xlator就能大致了解glusterfs的整体框架和数据流，接下来只需要针对各个xlator进行研究就可以了。

## write的调用流程（**xlator树的解析）**

本节以向 glusterfs 客户端的挂载点中写一个文件为例，**梳理xlator树的解析流程**：

1. linux 会向**VFS**传递这个写文件动作， VFS会将实际的处理交给**FUSE(kernel)文件系统**， 然后通过**/dev/fuse**这个设备文件， 最后将实际的写处理递交给了glusterfs功能树的**fuse\_xlator\_t** 这个根节点
2. 通过宏**STACK\_WIND**，根节点将这个写处理递交给他的子节点，子节点递归直到叶子节点停止，同时传递回调函数

|  |
| --- |
| STACK\_WIND (frame,  **writev\_cbk**,  child,  **child->fops->writev**,  fd,  vector,  count,  off); |

1. 叶子节点跟**socket**关联， 写操作就通过 socket 递交给glusterfs 服务器处理

**客户端处理结果**与上面的处理方向相反， 是由**子节点 -> 父节点**, 这是通过宏**STACK\_UNWIND**来实现的，处理的结果会一直被返回给 fuse\_xlators, 最后通过 FUSE(kernel)返回给用户

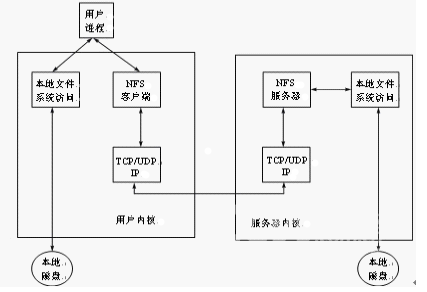
|  |
| --- |
| /\* return from function \*/  #define STACK\_UNWIND(frame, params ...) \  do { \  ret\_fn\_t fn = frame->ret; \  call\_frame\_t \*\_parent = frame->parent; \  \_parent->ref\_count--; \  fn (\_parent, frame->cookie, \_parent->this, params); \  } while (0) |

## NFS实现（**xlator服务加载示例）**

**本节作为xlator服务加载示例，**

### 5.7.1 NFS原理

NFS是一个使用SunRPC构造的客户端/服务器应用程序，其客户端通过向一台NFS服务器发送RPC请求来访问其中的文件。典型结构：



**NFS基于XDR/RPC协议**。**XDR（eXternal Data Representation，即外部数据表示法）**提供一种方法，把数据从一种格式转换成另一种标准数据格式表示法。**RPC（Remote Procedure Call，远程程序调用）**请求远程计算机给予服务。

NFS运用RPC传送数据的方法有以下几步：

（1）客户送出信息，请求服务。

（2）客户占位程序把客户送出的参数转换成XDR标准格式，并用系统调用把信息送到网络上。

（3）信息经过网络送达远程主机系统。

（4）远程主机将接受到的信息传给服务器占位程序。

（5）把XDR形式的数据，转换成符合主机端的格式，取出客户发出的服务请求参数，送给服务器。

（6）服务器给客户发送服务的逆向传送过程。

### 5.7.2 GlusterFS实现nfs

5.3节中说了服务以xlator表示，nfs也是如此。nfs translator运行在客户端模块。

nfs服务器启动流程如下：

|  |
| --- |
| 1. glusterfsd\src\glusterfsd.c，从入口函数main函数（5.2节有代码概述）开始，以glusterfs模式运行。对于nfs服务器，我们应该只关心nfs服务是如何加载的。 2. **glusterfs\_volumes\_init**被调用，初始化nfs卷服务 3. **glusterfs\_process\_volfp**被调用，构造graph并激活图结构的卷服务 4. **glusterfs\_graph\_activate**被调用，它是具体的激活函数 5. **glusterfs\_graph\_init** 被调用，初始化由整个配置文件中的各个卷组成的图结构，第一个节点为nfs类型，依次初始化每一个节点 6. **xlator\_init、\_\_xlator\_init、init**函数被依次调用，其中init为nfs真正的初始化函数 7. 调用**fini**函数，释放资源 |

其中init函数各个阶段如下，具体的nfs实现暂不细究：

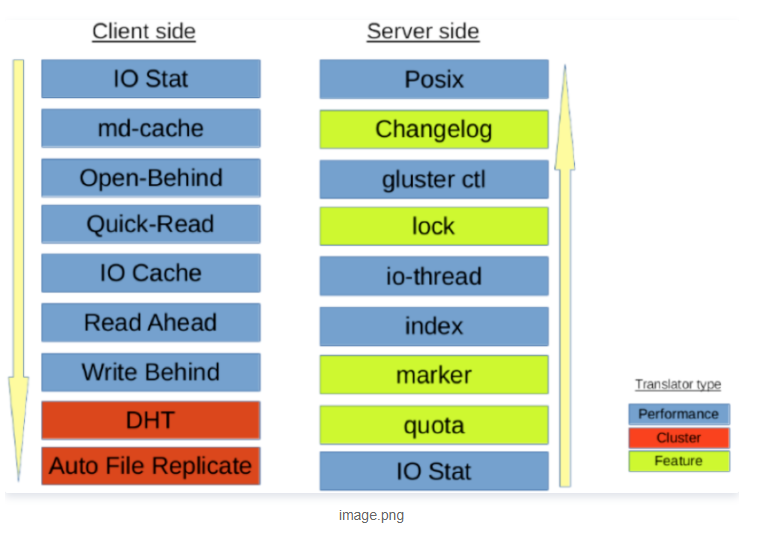
|  |
| --- |
| int init (xlator\_t \*this) {  struct nfs\_state \*nfs = NULL;  int ret = -1;  if (!this)  return -1;  nfs = nfs\_init\_state (this); //初始化一些nfs的选项参数  ret = nfs\_add\_all\_initiators (nfs); //添加所有协议的初始化器  ret = nfs\_init\_subvolumes (nfs, this->children); //初始化nfs的所有子卷  ret = nfs\_init\_versions (nfs, this); //初始化所有nfs协议的版本  return ret;  } |

## 重要xlator（translator）

下表为xlator目录下的所有内容

| Translator Type | Functional Purpose |
| --- | --- |
| Storage | 最底层的translator，存储、访问来自本地文件系统的数据 |
| Debug | 提供错误和调试的接口和统计信息 |
| Cluster | 处理数据的分发和复制，与bricks和节点的写入和读取有关 |
| Encryption | 扩展：存储数据的即时加密/解密 |
| Protocol | 扩展：客户端/服务器通信协议 |
| Performance | 调整translator以适应工作负载和 I/O 配置文件 |
| Bindings | 增加可扩展性，用于扩展与 GlusterFS 的 API 交互 |
| System | 系统访问translators,例如文件系统访问控制接口 |
| Scheduler | I/O调度器，决定如何分发新的写入操作到集群系统 |
| Features | 添加其他功能，例如配额、过滤器、锁等 |

默认的translators层次结构：



多个translator结合称为graph

其中最重要的两类为**Cluster and Performance**

### AFR（自动文件复制）

GlusterFS 中的自动文件复制 (AFR) 转换器使用扩展属性来跟踪文件操作。它负责跨bricks复制数据，属于同步复制策略

其职责包括：

1. 保持复制一致性（即两个brick上的数据应该相同，即使在多个应用程序/挂载点在同一文件/目录上并行发生操作的情况下，只要副本集中的所有bricks都已启动）。
2. 提供一种在发生故障时恢复数据的方法，只要至少有一个具有正确数据的brick即可。
3. 为read/stat/readdir 等提供新数据。

工作原理：

以含有两个副本A和B的DATA修复为例，Write的步骤可分解为：

1）下发Write操作。

2）加锁Lock。

3）向A，B副本的ChangeLog分别加1，记录到各个副本的扩展属性中。

4）对A，B副本进行写操作。

5）若该副本写成功则ChangeLog减1，若该副本写失败则ChangeLog值不变，记录到各个副本的扩展属性中。

6）解锁UnLock。

7）向上层返回，只要有一个副本写成功就返回成功。

afr.c中定义fops，整个fops包含后面要处理的对应事件的函数，在xlator初始化时注册

|  |
| --- |
| struct xlator\_fops fops = {  　　.lookup = afr\_lookup, 查询指定的目录（这里是将请求发向所有的节点）  　　.lk = afr\_lk,  　　.flush = afr\_flush, 将文件从内存写到磁盘  　　.statfs = afr\_statfs, 搜集磁盘的容量状态  　　.fsyncdir = afr\_fsyncdir,  　　.inodelk = afr\_inodelk,  　　.finodelk = afr\_finodelk,  　　.entrylk = afr\_entrylk,  .fentrylk = afr\_fentrylk,  .ipc = afr\_ipc,  .lease = afr\_lease,  　　/\* inode read \*/  　　.access = afr\_access, 检查调用进程是否可以对指定的文件执行某种操作  　　.stat = afr\_stat, 通过文件名获取文件的信息  　　.fstat = afr\_fstat, 以fd的方式来获取文件的信息  　　.readlink = afr\_readlink, 读取链接文件  　　.getxattr = afr\_getxattr, 以路径参数得到对应文件扩展属性  　　.fgetxattr = afr\_fgetxattr,以fd作为参数得到对应文件扩展属性  .readv = afr\_readv, 读取对应文件内容  .seek = afr\_seek, 跳转到指定偏移读取  　　/\* inode write \*/  　　.writev = afr\_writev, 写文件  　　.truncate = afr\_truncate, 以路径参数截断文件  　　.ftruncate = afr\_ftruncate,以fd参数截断文件  　　.setxattr = afr\_setxattr, 以路径参数设置文件的扩展属性  　　.fsetxattr = afr\_fsetxattr,以fd参数设置文件的扩展属性  　　.setattr = afr\_setattr, 以路径参数设置文件的属性  　　.fsetattr = afr\_fsetattr, 以fd参数设置文件的属性  　　.removexattr = afr\_removexattr,以路径参数删除文件的扩展属性  .fremovexattr = afr\_fremovexattr,以fd参数删除文件的扩展属性  .fallocate = afr\_fallocate,  .discard = afr\_discard,  .zerofill = afr\_zerofill,  .xattrop = afr\_xattrop,  .fxattrop = afr\_fxattrop,  .fsync = afr\_fsync,  /\*inode open\*/  .opendir = afr\_opendir, 打开目录  .open = afr\_open,  　　/\* dir read \*/  　　.readdir = afr\_readdir,读取目录  .readdirp = afr\_readdirp,读取目录  　　/\* dir write \*/  　　.create = afr\_create,以指定的格式创建文件  　　.mknod = afr\_mknod, 创建文件，主要是fifo文件  　　.mkdir = afr\_mkdir, 创建目录  　　.unlink = afr\_unlink, 删除目录项，并且减少一个链接数  　　.rmdir = afr\_rmdir, 删除目录  　　.link = afr\_link, 创建硬链接  　　.symlink = afr\_symlink, 创建软链接  　　.rename = afr\_rename, 重命名文件  　　}; |

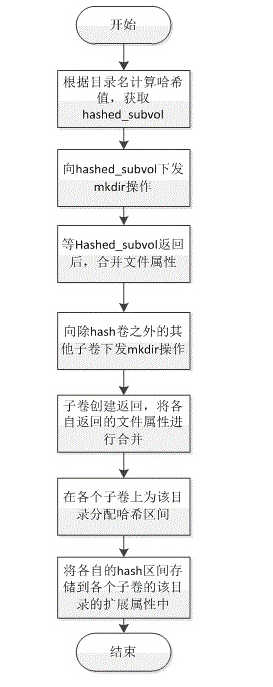
### DHT（分布式哈希表）

DHT 中使用的基本方法是**一致性哈希**。每个子卷（brick）被分配一个 32 位哈希空间的范围，覆盖整个范围，没有孔洞或重叠。然后通过散列文件名称，每个文件在同一空间中被分配一个值。

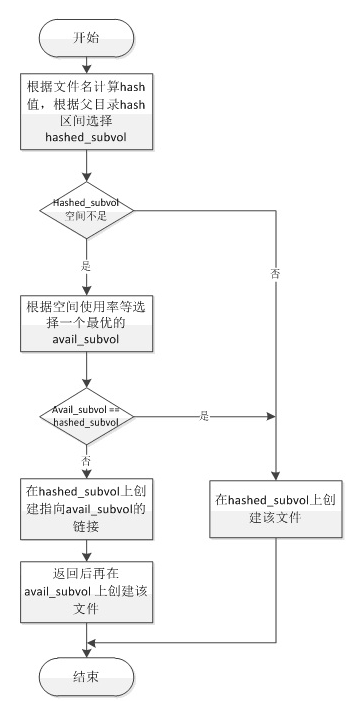
文件的哈希值如果在某一个brick的范围内，那么文件“应该”在该brick上。但是在很多情况下并非这么简单，例如自文件创建后，bricks（以及范围分配）发生了变化，或者brick几乎已满等等场景。

DHT 中的大部分复杂性都涉及这些特殊情况。 当 open() 一个文件时，分发translator提供文件名来查找文件，为了确定该文件的位置，translator通过散列算法处理文件名将该文件名转换为数字。

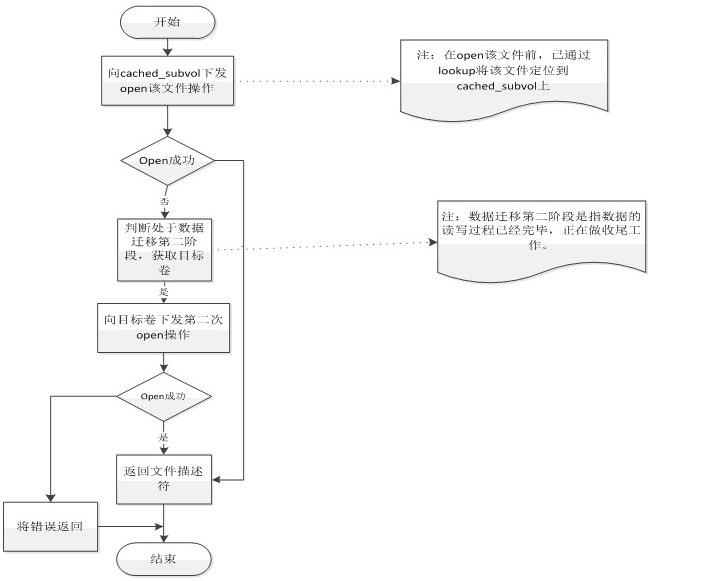
**创建目录流程**



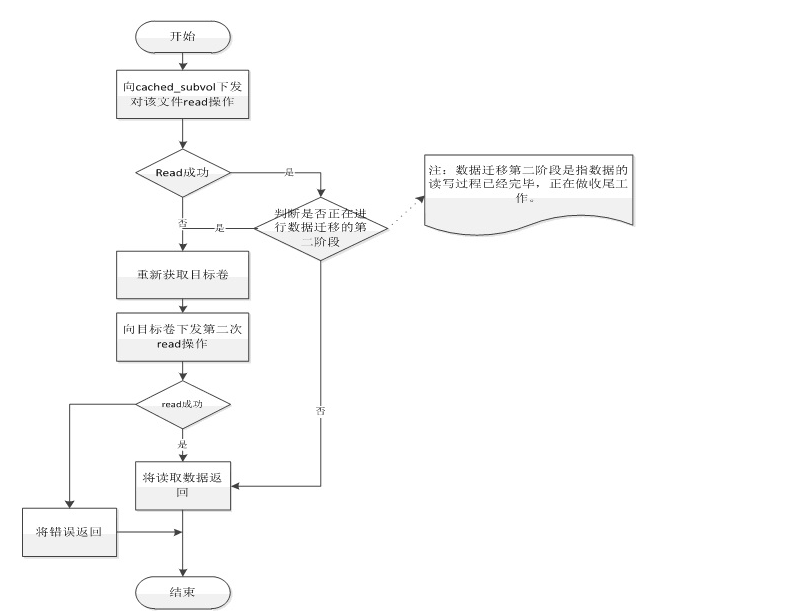
**创建文件流程**



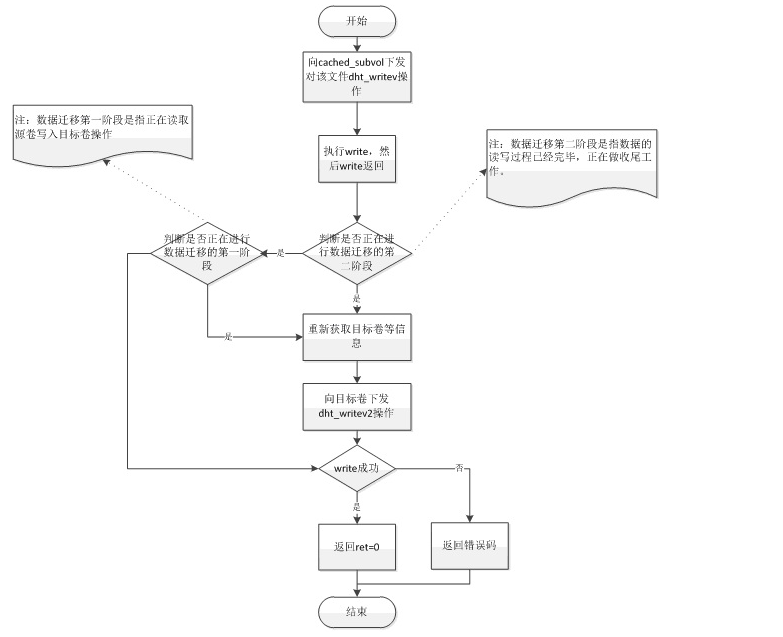
**打开文件**



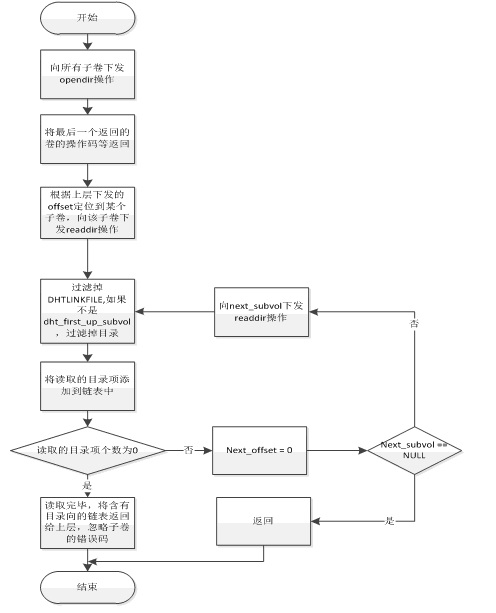
**读取文件**



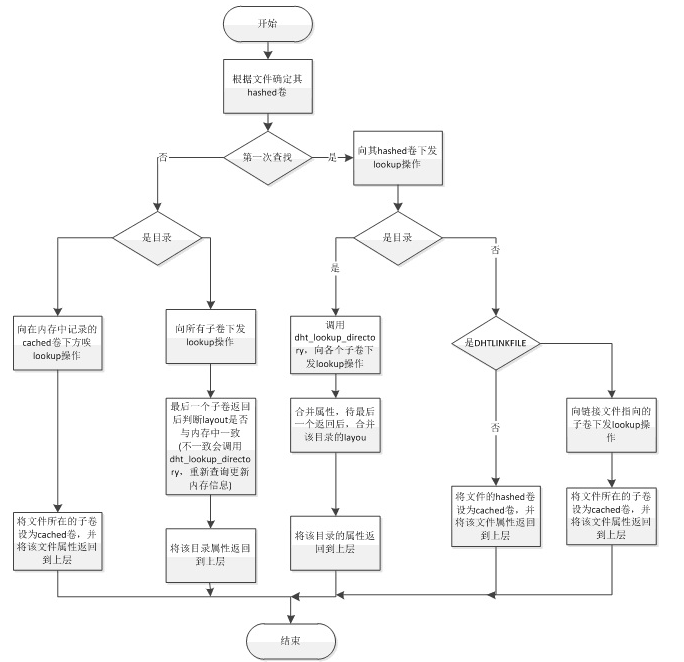
**写入数据**



读取目录



查找（lookup）



**添加卷后lookup**：

1)执行添加卷命令后，将会重新初始化。

2)lookup目录时，待各个子卷将目录信息返回后，都会调用dht\_layout\_merge,将各个子xlator指针、返回值等添加到layout中。

3)调用dht\_layout\_normalize，新添加的list.err会被置为ENOENT(start=stop=0，在检测是否有空洞和重叠时已按hash区间排序，所以新添加的卷没有空洞和重叠)。

4)dht\_layout\_normalize返回!=0, 然后进入目录修复。

5)调用dht\_selfheal\_dir\_mkdir在新添加的卷上创建该目录setattr(该目录没有分布区间信息,所以不需要setxattr)。

6)调用dht\_selfheal\_dir\_finish结束。

注：再次lookup时，在dht\_layout\_normalize中因为layout->list.err < 0(err ==-1)，所有该函数返回0(第一次该函数会返回ret>0)，不会触发目录修复动作。

后端**手动添加文件，再执行ls操作**，其主要流程有：

1)readdir时，其父目录会将该目录项返回给上层。

2)然后对该文件进行lookup。

3)若通过hashed\_subvol直接定位到了该文件，则将该文件属性返回给上层。

4)若没有，则会lookup\_everywhere，找到该文件，然后将该文件作为其cached\_subvol，并创建hashed\_subvol到cached\_subvol的链接文件。

后端**手动添加目录后，执行ls操作**，其主要流程有：

1)若该新添加的目录不是位于first\_up\_subvol，则该目录向在其父目录readdir时会被过滤，即在挂载点不会看到新添加的目录。

2)若新添加的目录位于first\_up\_subvol，则在readdir父目录时会向将该目录项返回给上层。

3)然后对该目录项进行lookup，在其hashed\_subvol找到该目录的话，执行looku\_directory（各个卷查找该目录）。若找不到，则会执行lookup\_everywhere.

4)在lookup\_diectory后，若需要修复，则在各子卷创建该目录，并分配hash区间。

5)在lookup\_everywhere时，找到该目录，然后再执行looku\_directory.

**修复目录layout**的主要流程有：

1)重新分配hash区间，hash区间按子卷个数划分，优先分配与原区间重叠最大的区间段。

2)将重新分配的hash区间，存储到其扩展属性中。

**数据迁移**的主要流程有：

1)首先lookup该目录。

2)遍历该目录下的DHT\_LINKFIFE.

3)如果该文件实际就是符号链接,则根据源文件信息在to上建立该符号链接，如果是设备文件，在to上mknode。然后将源文件unlink

4)如果是普通文件，则在其hash卷上create该文件。

5)然后打开源文件。

6)检测是否含有空洞文件。

7)进行读写。

8)读写完毕后，move扩展属性。

9)unlink源文件，truncate，然后清楚标志位等。

10)迁移该文件结束。

### io-cache

读文件（ioc\_readv）：

1、如果inode\_ctx\_get检查到inode在ctx中、未设置O\_DIRECT标志、fd\_ctx\_get检查到fd在ctx中则使用caching机制；

2、调用mem\_get0从内存池获取内存；

3、调用ioc\_dispatch\_requests，在缓存中查询请求对应的区域，并从cache中读取

查询操作（ioc\_lookup）：

1. 调用mem\_get0从内存池获取内存；
2. 调用loc\_copy获取对应路径的属性信息
3. 调用STACK\_WIND，回调设置为ioc\_lookup\_cbk（主要调用ioc\_inode\_update进行更新）

打开操作（ioc\_open）：

1. 调用mem\_get0从内存池获取内存；
2. 调用STACK\_WIND，回调设置为ioc\_open\_cbk
3. 回调函数通过inode\_ctx\_get检查到inode是否在ctx中，使用fd\_ctx\_set设置fd到ctx中

### io-threads（待阅读）

### md-cache

### open behind

### quick read

### read-ahead

### write-behind

## 5.9与FUSE的交互

glusterfs定义的fuse\_operations结构体：

|  |
| --- |
| static fuse\_handler\_t \*fuse\_std\_ops[FUSE\_OP\_HIGH] = {  [FUSE\_LOOKUP] = fuse\_lookup,  [FUSE\_RMDIR] = fuse\_rmdir,  [FUSE\_RENAME] = fuse\_rename,  [FUSE\_LINK] = fuse\_link,  [FUSE\_OPEN] = fuse\_open,  [FUSE\_READ] = fuse\_readv,  [FUSE\_WRITE] = fuse\_write,  [FUSE\_STATFS] = fuse\_statfs,  [FUSE\_SETXATTR] = fuse\_setxattr,  [FUSE\_GETXATTR] = fuse\_getxattr,  [FUSE\_ACCESS] = fuse\_access,  [FUSE\_CREATE] = fuse\_create,  ..........................  }; |

这些函数在glusterfs中可以简称为fop，glusterfs是Linux用户态的文件系统，会启动一个**fuse\_proc\_fuse**()**轮询读取/dev/fuse设备**，读取的数据出来保存在两个iov\_in结构体当中，一个保存fop类型，一个保存写入的数据。

例如本次操作为写操作，那么会从第二个iov\_in读取出数据，在相应的fop函数当中会执行解析文件名，gfid，构建inode等信息。准备好这些信息，调用STACK\_WIND下发到下一个xlator

# 调试方法

## 6.1 日志

**客户端调试方法**

可以添加 -o log-level=DEBUG参数，查看详细调试信息。

用法示例：

mount -t glusterfs -o log-level=DEBUG 192.168.1.120:/tank /mnt

glusterd.log（管理模块日志）

**服务器端调试方法**

可以更改日志级别：

diagnostics.brick-log-level # brick进程日志级别，默认为INFO

diagnostics.client-log-level # 客户端日志级别，默认为INFO

diagnostics.brick-sys-log-level # 默认值为CRITICAL，不用动，在gluster日志中也有

diagnostics.client-sys-log-level # 默认值为CRITICAL，不用动，在gluster日志中也有

用法示例：

gluster volume set diagnostics.brick-log-level DEBUG

glusterd服务默认启动进程为：

/usr/sbin/glusterd -p /var/run/glusterd.pid

调试时，也可以直接改变日志级别：

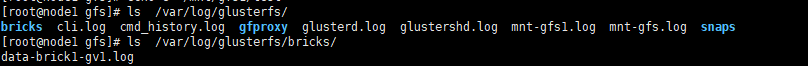
glusterd -p /var/run/glusterd.pid -L DEBUG

可选的日志级别为：

DEBUG, INFO, WARNING, ERROR, CRITICAL, TRACE and NONE，默认是INFO。

**主要日志位置说明：**

**目录/var/log/glusterfs**



**cli.log** 即**gluster cli命令执行工具**的日志

**cmd\_history.log** 为节点上执行的**GlusterFS命令**的日志（简单）

**bricks**目录下包含每个brick对应**glusterfsd服务端模块**的日志

**glusterd.log** 为**glusterd管理模块**日志文件

mnt-gfs1.log mnt-gfs.log 为**glusterfs客户端模块**的日志

## 6.2 statedump（状态转储）

获取转储文件存放路径

gluster --print-statedumpdir

如何生成转储文件

直接杀死进程：

kill -USR1 <pid-of-gluster-process>

kill -USR1 <pid-of-gluster-mount-process>

其他方法：

For bricks:

gluster volume statedump <volname>

For the NFS server:

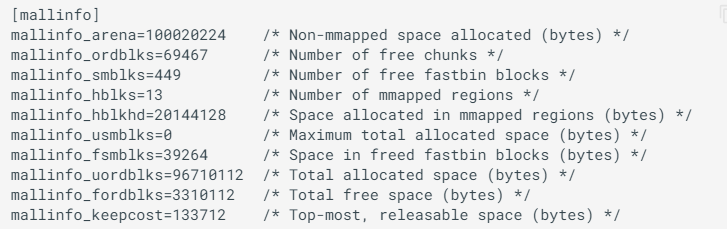
gluster volume statedump <volname> nfs

For quotad:

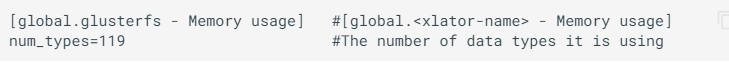
gluster volume statedump <volname> quotad

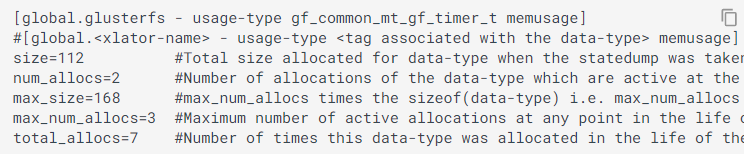
dump文件呈现方式

模板信息（mallinfo）

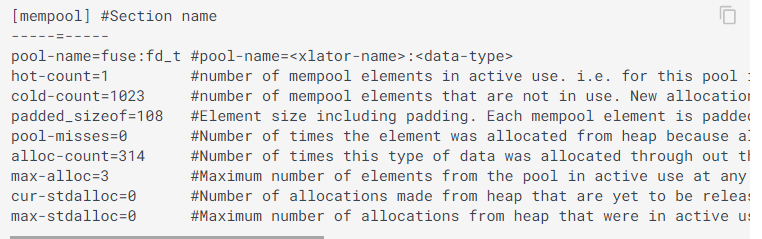


xlator内存信息

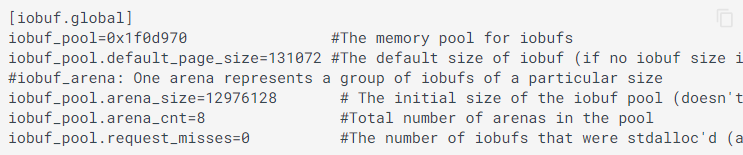




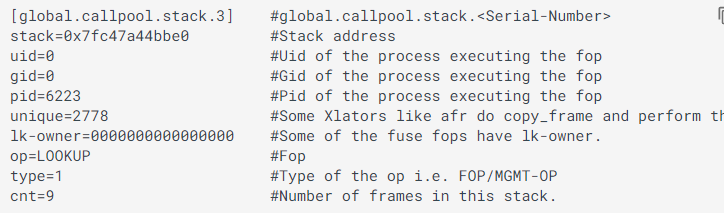
内存池



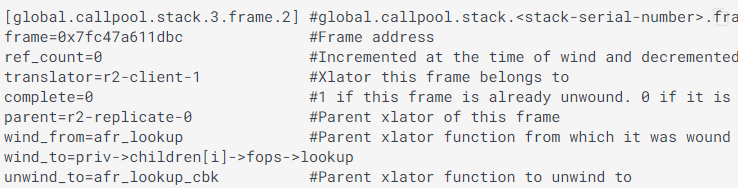
Iobufs（分为Arena list、Purge list、Filled list）



调用栈



调用帧（可看出属于哪个xlator，来自哪里，发往何处）



更多信息请参考：

<https://docs.gluster.org/en/latest/Troubleshooting/statedump/>

包含几个典型错误分析例子，如资源泄露

此外可参考

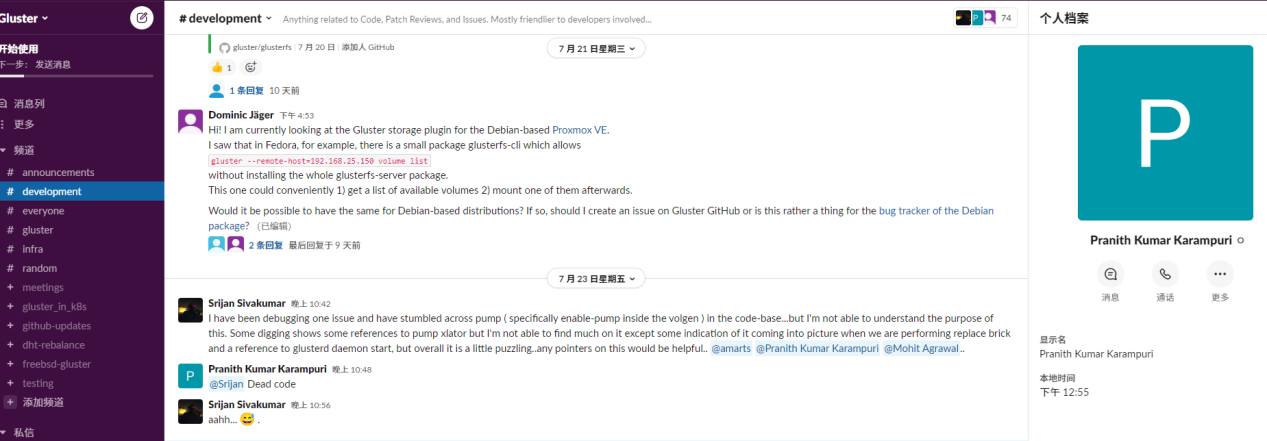
<https://docs.gluster.org/en/latest/Troubleshooting/troubleshooting-glusterd/>

# 社区

主页：<https://www.gluster.org/community/>

实时频道：

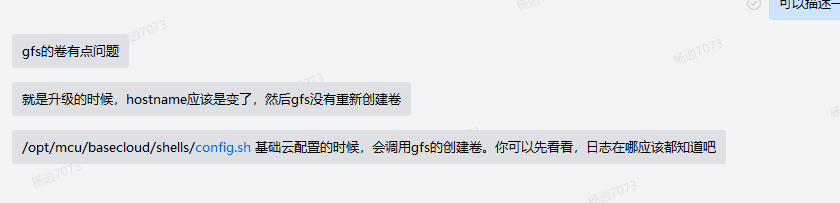
<https://join.slack.com/t/gluster/shared_invite/enQtODMwMDU5MTI0OTQ3LTNjMTA4NTJmMDY3OGM4YTA0ZDlhOGM5ZWYzNjRkYmQ0Mjg3NDUxODRjZGI4ZjQxNDczNTYxZWZjMmY5NDMyNzM>



# 实际排障记录

1）

**问题描述**

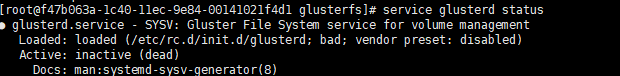


**日志位置**

/usr/local/glusterfsd/var/log/glusterfs/

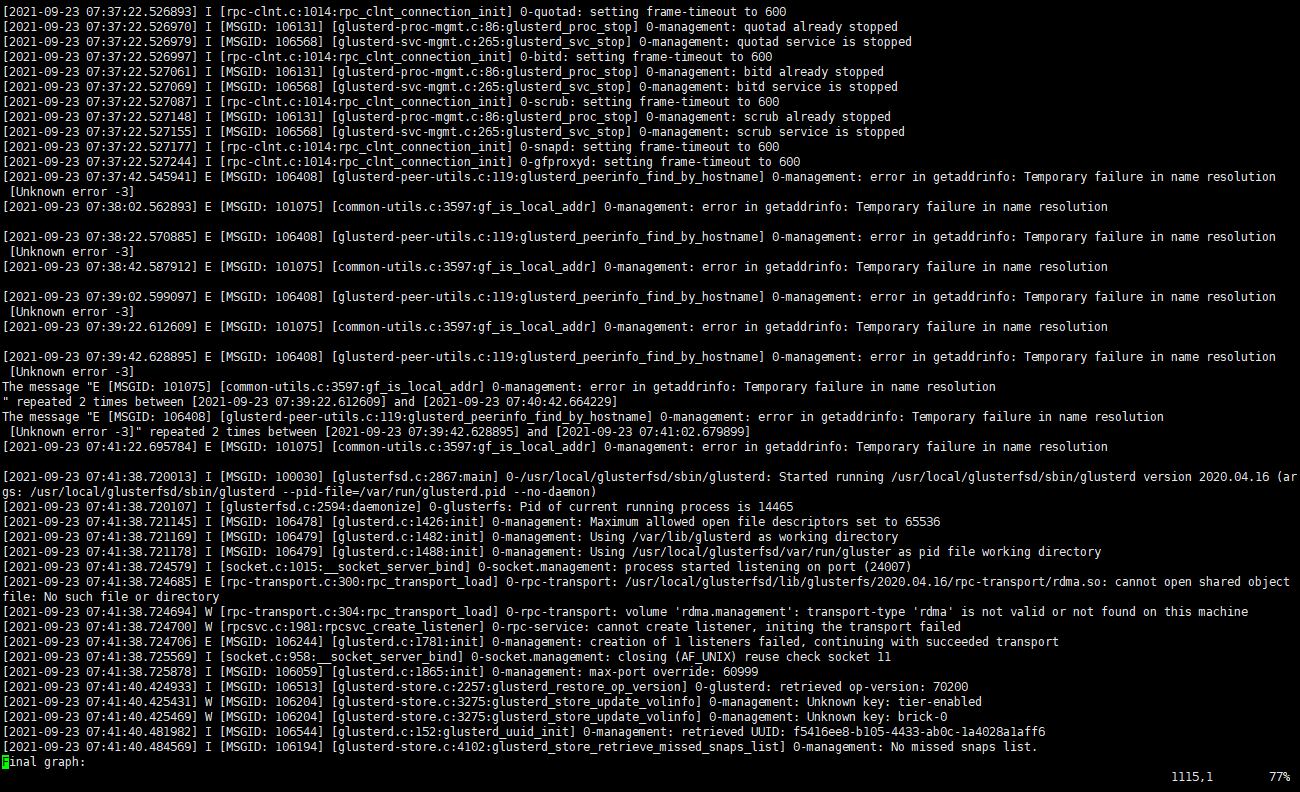
**查看管理模块状态**

**inactive**



**查看管理模块日志**

**/usr/local/glusterfsd/var/log/glusterfs/glusterd.log**

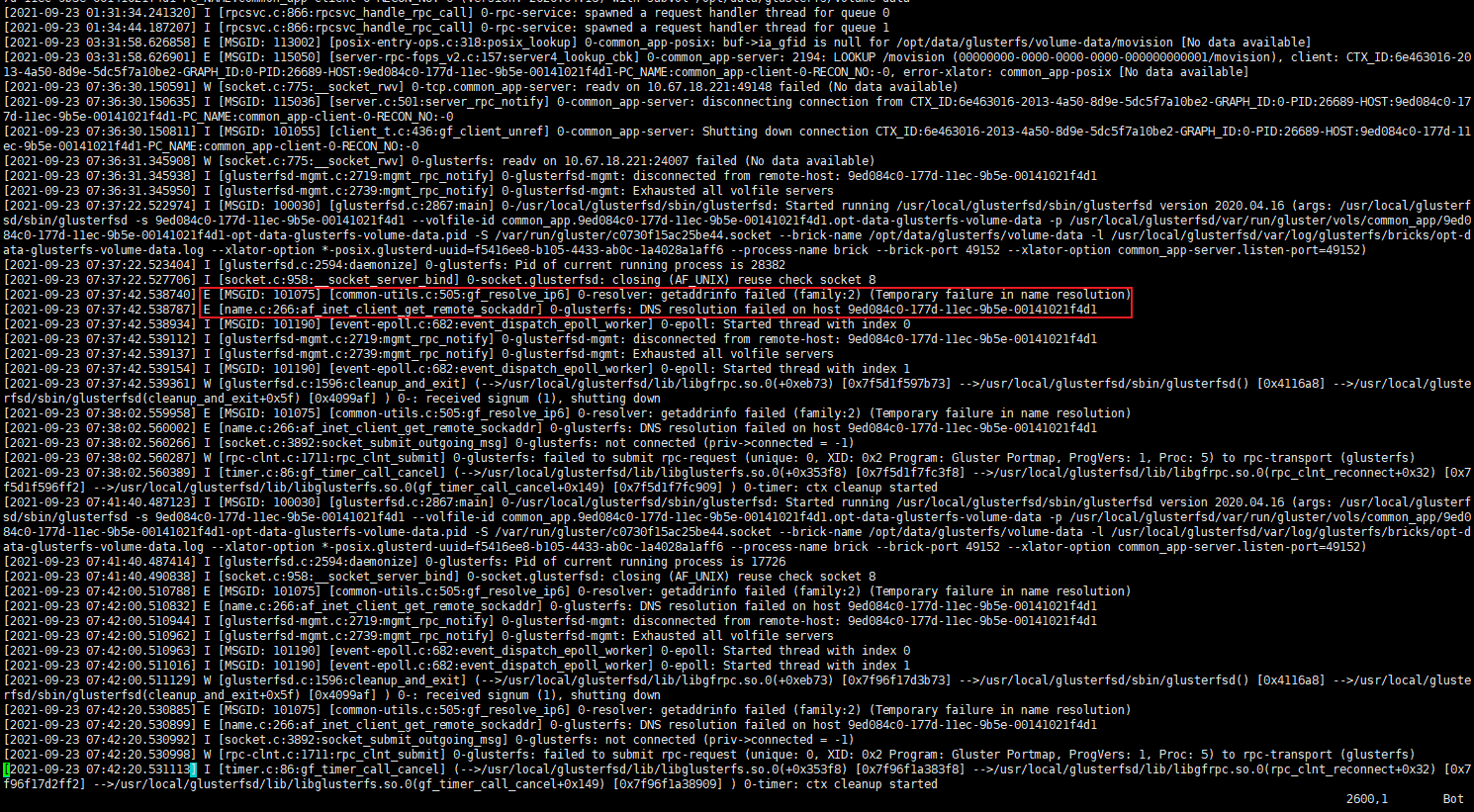


关键错误信息

|  |
| --- |
| The message "E [MSGID: 106408]  [glusterd-peer-utils.c:119:glusterd\_peerinfo\_find\_by\_hostname] 0-management: error in getaddrinfo: Temporary failure in name resolution  The message "E [MSGID: 101075]  [common-utils.c:3597:gf\_is\_local\_addr] 0-management: error in getaddrinfo: Temporary failure in name resolution |

看起来问题出在由hostname获取信息失败

**查看brick对应glusterfsd服务端模块的日志**

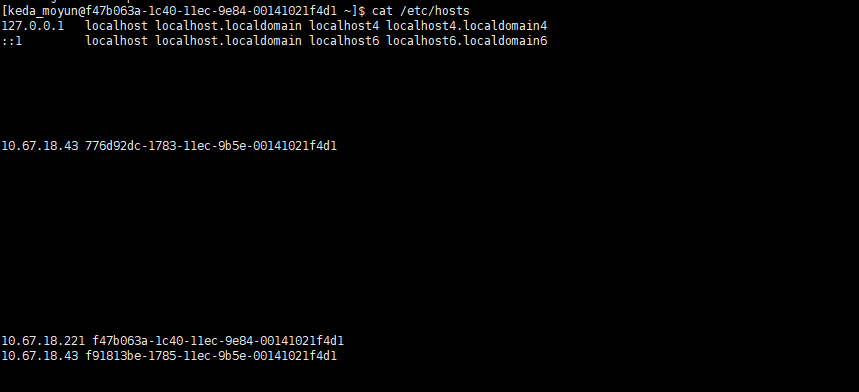


关键错误信息

|  |
| --- |
| [2021-09-23 07:37:42.538740] E [MSGID: 101075]  [common-utils.c:505:gf\_resolve\_ip6] 0-resolver: getaddrinfo failed (family:2) (Temporary failure in name resolution)  [2021-09-23 07:37:42.538787] E  [name.c:266:af\_inet\_client\_get\_remote\_sockaddr] 0-glusterfs: DNS resolution failed on host 9ed084c0-177d-11ec-9b5e-00141021f4d1 |

指出主机**host 9ed084c0-177d-11ec-9b5e-00141021f4d1**的DNS地址解析失败，推测不知道主机名和IP对应关系导致的

**查看主机ip映射/etc/hosts**



没有指明**host 9ed084c0-177d-11ec-9b5e-00141021f4d1的ip地址**