GLUSTERFS （3.2.5）

源 码 结 构 分 析

2012.3.26 于天津

刘建辉（huijianliu@gmail.com）QQ：94668419

目录

1 GlusterFS 之 GlusterFS 概括

1.1 glusterFS 源码目录分析

glusterFS 为一分布式文件系统，从源码组织分析其目录结构为：

a. argp-standalone 文件夹内容为命令行参数解析库，为调用应用程序传进来参数解析提供库函数
b. cli 文件夹为命令行接口（command line interface）应用程序，调用该程序执行 glusterfs 的相关操作命令。
c. contrib 文件夹里面包含用到得第三方库源码。例如 MD5，fuse，uuid 等。
d. doc 文件夹为 glusterFS 的一些文档和配置 example
e. extras 文件夹包含 glusterFS 配置的一些脚本信息。
f. glusterfsd 文件夹包含了 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。
g. libglusterfs 为一库工程，为 glusterfsd 以及 xlator 下的相关 translator 提供库函数。
h. xlators 文件夹包含所有用到的 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。例如 glusterd 应用程序只载入 mgmt。根据默认卷配置信息：/usr/local/etc/glusterfs/glusterd.vol。
i. rpc 文件夹包含 gluserFs 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。
j. configure.ac 文件为根据 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。
k. autogen.sh 为一脚本文件，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。

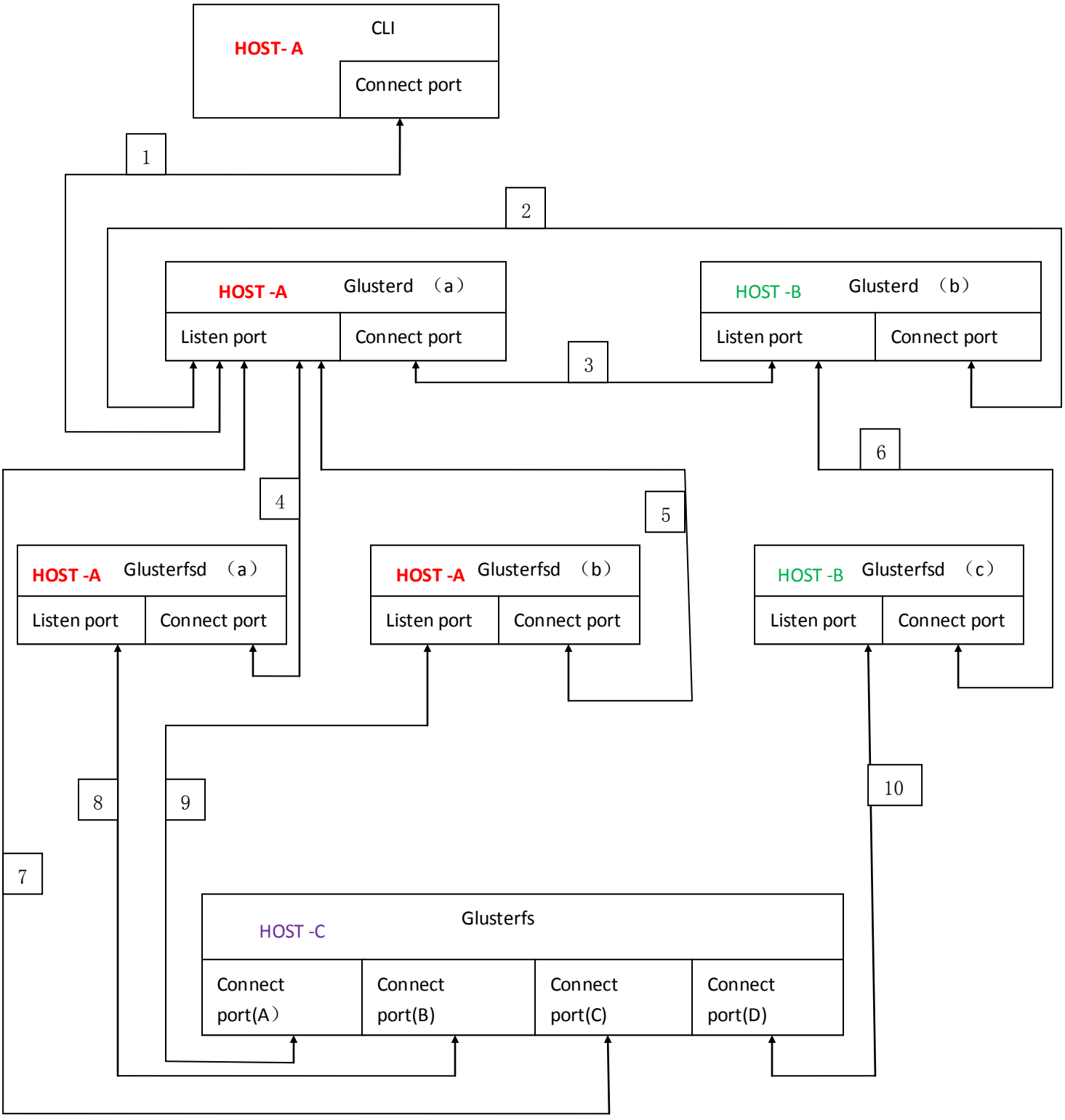
1.2 glusterFS 工作时各个应用程序之间联系分析

GlusterFS 工作时候共有四种应用程序需要运行，它们分别是 gluster，glusterd，glusterfsd，glusterfs，其中 glusterd，glusterfs 为 glusterfsd 的链接文件，通过修改该应用程序名称，来区分不同的功能（代码内部根据执行文件名称走不同函数流程）。

a. gluster 为 cli，即命令行执行工具，该应用负责把对 glusterFS 的操作请求发送到 glusterd 上去执行。
b. glusterd 为 damon 程序，该 damon 负责接收 gluster 发送过来的操作请求，并执行相关的操作，例如调用 glusterfsd 启动 brick 服务。
c. glusterfsd 为服务进程，由 glusterd 启动。根据卷配置信息执行由 glusterfs 发送过来的请求。通过 rpc 连接 glusterd 获取该服务的卷信息。
d. glusterfs 为客户端程序。根据卷配置信息将 fuse 过来的操作请求逐层专递到最底层

的 protocol/client translater 上，该 translater 通过 RPC 与 glusterfsd 连接，将请求发送到 glusterfsd 服务端继续执行。通过 rpc 连接 glusterd 获取该服务的卷信息

1.3 glusterFS（cli,deamon,server,client） 四方架构图



1.4 架构图注释

该构架图描述了 glusterFS 的通信结构图，每个应用程序通过 RPC（socket）与其他应用程序进行通信。图中 listen port 为应用程序创建的 sokcet 监听端口，connet port 为连接其他应用程序的 listen port 的端口。

1. 1 号线为 CLI 与 gluserd 之间 command 连接，通过该连接 cli 向 gluserd 发送命令，gluserd 执行譬如 peer，create，start 等命令。
2. 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。
3. 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。例如：hello.vm-pool-3.home-test1.vol
4. 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，例如：hello-fuse.vol
5. 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。

2 GlusterFS 之 damon

通过 ps -ef 查看进程项，可以发现 damon 启动项详细情况为：/usr/local/sbin/glusterd -p /var/run/glusterd.pid。即可执行文件名称为 glusterd。仅一个参数-p

该服务启动入口处为 glusterfsd/src/glusterfsd.c 的 main 函数，该入口也是 server 和 client 的启动入口。下面对启动代码分析一下。

1. ret = glusterfs_globals_init ();

   该函数主要初始化全局变量，里面主要包含了下面几个初始化函数：

   1.1 ret = glusterfs_ctx_init ();

   //该函数创建进程上下文 glusterfs_ctx 变量并初始化。

   1.2 ret = glusterfs_this_init ();

   //该函数初始化 global_xlator，并将上面创建的 glusterfs_ctx 赋值给 global_xlator。“THIS”目前指向该 translator。

   1.3 gf_mem_acct_enable_set ()

   //该函数读取参数或环境变量赋值全局变量 gf_mem_acct_enable。

2. ctx = glusterfs_ctx_get ();

   //获取上面创建的 lusterfs_ctx，无需解释。

3. ret = glusterfs_ctx_defaults_init (ctx);

   //初始化 ctx 众多参数

   3.1 xlator_mem_acct_init (THIS, gfd_mt_end);

   //根据 1.3，赋值 THIS（global_xlator）的 mem_acct 变量。

   其中#define THIS (*__glusterfs_this_location())。该宏通过__glusterfs_this_location（）里的函数“this_location = pthread_getspecific (this_xlator_key);”获取线程私有数据 this_xlator_key 的值。获取过程无需解释。

   3.2 ctx->process_uuid = generate_uuid ();

   //初始化 ctx->process_uuid 为由时间和主机等组成字符串。

3.3 ctx->page_size = 128 * GF_UNIT_KB;
ctx->iobuf_pool = iobuf_pool_new (8 * GF_UNIT_MB, ctx->page_size);
//创建一个总大小为 8 兆大小，每页 128 个字节的 io 内存池。创建过程如下：
3.3.1 iobuf_pool = GF_CALLOC (sizeof (*iobuf_pool), 1,gf_common_mt_iobuf_pool);
INIT_LIST_HEAD (&iobuf_pool->arenas.list);
INIT_LIST_HEAD (&iobuf_pool->filled.list);
INIT_LIST_HEAD (&iobuf_pool->purge.list);
iobuf_pool->arena_size = arena_size;
iobuf_pool->page_size = page_size;
//创建一个 iobuf_pool 结构体并初始化结构体变量，三个存储类型列表，和总内存池大小以及单页大小等。
3.3.2 iobuf_pool_add_arena (iobuf_pool);
//继续深入初始化 iobuf_pool，详情如下：
3.3.2.1 iobuf_arena = __iobuf_pool_add_arena (iobuf_pool);
//继续深入初始化 iobuf_pool，详情如下：
3.3.2.1.1 iobuf_arena = __iobuf_arena_unprune (iobuf_pool);
//尝试从 iobuf_pool->purge.list 列表中回收 iobuf_arena
3.3.2.1.2 iobuf_arena = __iobuf_arena_alloc (iobuf_pool);
//如果上步回收失败则创建一个 obuf_arena，创建过程如下：
3.3.2.1.2.1 iobuf_arena = GF_CALLOC (sizeof (*iobuf_arena), 1,
gf_common_mt_iobuf_arena);
//创建一个 iobuf_arena 结构体
3.3.1.1.2.2 INIT_LIST_HEAD (&iobuf_arena->list);
INIT_LIST_HEAD (&iobuf_arena->active.list);
INIT_LIST_HEAD (&iobuf_arena->passive.list);
//初始化 iobuf_arena 的 list
3.3.1.1.2.3 arena_size = iobuf_pool->arena_size;
iobuf_arena->mem_base = mmap (NULL, arena_size,
PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
//申请一快 arena_size（8M）大小的内存映射区域
3.3.1.1.2.4 __iobuf_arena_init_iobufs (iobuf_arena);
//对该内存区域进行配置分页，详情如下：
3.3.1.1.2.4.1 arena_size = iobuf_arena->iobuf_pool->arena_size;
page_size = iobuf_arena->iobuf_pool->page_size;
iobuf_cnt = arena_size / page_size;
//根据区域大小和一页大小计算可以分多少页
3.3.1.1.2.4.2 iobuf_arena->iobufs = GF_CALLOC (sizeof (*iobuf),
iobuf_cnt,gf_common_mt_iobuf);
//申请 iobuf_cnt 个 iobuf 用来存储分页信息。
3.3.1.1.2.4.3 iobuf = iobuf_arena->iobufs
//获取内存区域的首地址为一个页信息的指针
for (i = 0; i < iobuf_cnt; i++)
INIT_LIST_HEAD (&iobuf->list);

//初始化 iobuf->list
LOCK_INIT (&iobuf->lock);//加锁
iobuf->iobuf_arena = iobuf_arena;
//该 iobuf 所属于 iobuf_arena
iobuf->ptr = iobuf_arena->mem_base + offset;
//从 iobuf_arena->mem_base 分出
page_size 区域给 iobuf->ptr
list_add (&iobuf->list, &iobuf_arena->passive.list);
//将该页填入 obuf_arena->passive 列表,（未使用）
iobuf_arena->passive_cnt++;
//计数器加加
offset += page_size;
//重新计算分出页面首地址的偏移量
iobuf++;
//从 iobuf_arena->iobufs 获取下个 iobuf 的首指针
}

3.3.1.1.2.5 iobuf_pool->arena_cnt++;
//区域计数器加加

3.3.2.1.3 list_add_tail (&iobuf_arena->list, &iobuf_pool->arenas.list);
//将回收或创建的 iobuf_arena 添加到 iobuf_pool->arenas 列表中。

3.4 ctx->event_pool = event_pool_new (DEFAULT_EVENT_POOL_SIZE);
//创建 DEFAULT_EVENT_POOL_SIZE（16384）个事件池，创建过程如下：

3.4.1 event_pool = event_ops_epoll.new (count);
//调用 event_ops_epoll.new 函数创建 event_pool，详情如下：

3.4.1.1 event_pool = GF_CALLOC (1, sizeof (*event_pool),
gf_common_mt_event_pool);
event_pool->count = count;
event_pool->reg = GF_CALLOC (event_pool->count,
sizeof (*event_pool->reg),
gf_common_mt_reg);
//创建一个 event_pool 结构体并初始化其变量，并创建
count event_pool->reg 结构体

3.4.1.2 epfd = epoll_create (count);
event_pool->fd = epfd;
event_pool->count = count;
创建 epoll 变量，并将创建句柄赋值给 event_pool->fd，用来监听

3.4.1.3 pthread_mutex_init (&event_pool->mutex, NULL);
pthread_cond_init (&event_pool->cond, NULL);
//初始化线程锁和条件锁，完成 event_pool 初始化

3.4.2 event_pool->ops = &event_ops_epoll;
//函数指针结构体赋值操作，以后该 event_pool->ops 的对应的函数指针
均对应于 event_ops_epoll 结构体重的函数指针。

3.4.3 在创建监听端口时候，会调用 event_ops_register()注册事件处理函数。

3.5 pool = GF_CALLOC (1, sizeof (call_pool_t),gfd_mt_call_pool_t);

pool->frame_mem_pool = mem_pool_new (call_frame_t, 16384);
pool->stack_mem_pool = mem_pool_new (call_stack_t, 8192);
ctx->stub_mem_pool = mem_pool_new (call_stub_t, 1024);
//调用 mem_pool_new 函数创建相应类型的内存池 n 个， 该函数为一函数宏：
//#define mem_pool_new(type,count) mem_pool_new_fn (sizeof(type), count)，创
//建过程如下:

3.5.1 padded_sizeof_type = sizeof_type + GF_MEM_POOL_PAD_BOUNDARY;
//一个 type 类型的结构体所占内存（令额外加上链表结构体指针）

3.5.2 mem_pool = GF_CALLOC (sizeof (*mem_pool),
1, gf_common_mt_mem_pool);
mem_pool->padded_sizeof_type = padded_sizeof_type;
mem_pool->cold_count = count;
mem_pool->real_sizeof_type = sizeof_type;
//创建并初始化一个 mem_pool 结构体

3.5.3 pool = GF_CALLOC (count, padded_sizeof_type, gf_common_mt_long);
//创建一个大内存池，大小为 count*padded_sizeof_type

3.5.4 for (i = 0; i < count; i++) {
list = pool + (i * (padded_sizeof_type));
INIT_LIST_HEAD (list);
list_add_tail (list, &mem_pool->list);
}
//将该打内存池以 padded_sizeof_type 等分，并用 list_head *list 结构体将等分串到 mem_pool->list 变量上。（即每等份内存除包含 type 结构体大小内存外海需要包含 GF_MEM_POOL_PAD_BOUNDARY 大小的链表结构，对应 3.5.1）

3.6 cmd_args = &ctx->cmd_args;
cmd_args->log_level = DEFAULT_LOG_LEVEL;
cmd_args->fuse_direct_io_mode = GF_OPTION_DISABLE;
//设置缺省的明两行参数信息，如 log 等级，fuse 模式等。

3.7 lim.rlim_cur = RLIM_INFINITY;
lim.rlim_max = RLIM_INFINITY;
setrlimit (RLIMIT_CORE, &lim);
//设置进程资源限制，可以 google 一下。

4. ret = parse_cmdline (argc, argv, ctx);
//解析命令行参数
4.1 if (ENABLE_DEBUG_MODE == cmd_args->debug_mode)
//debug 模式则重新定向 log 的等级和输出。
4.2 process_mode = gf_get_process_mode (argv[0]);
//根据可执行程序的名称来判定该应用程序的工作模式。并设置相关卷文件路径该：
cmd_args->volfile = gf_strdup (DEFAULT_CLIENT_VOLFILE)。该 damon 模式为 GF_GLUSTERD_PROCESS，还有 GF_SERVER_PROCESS（server），GF_CLIENT_PROCESS（client）

5. ret = logging_init (ctx);
//log 初始化，打开 log 文件

5.1 ret = set_log_file_path (cmd_args);
//根据参数设置不同的 log 文件路径和名称
5.2 if (gf_log_init (cmd_args->log_file) == -1)
//打开 log 文件，获取文件句柄

6. ret = create_fuse_mount (ctx);
//该 damon 由于没有 mount point，所以会直接返回，我们将在 client 中详细分析。

7. ret = daemonize (ctx);
//根据是否 debug 模式等参数，来决定是否启动新进程重新定向输入输入，并关闭此 shell。

8. ret = glusterfs_volumes_init (ctx);
//该函数负责创建监听端口，与监听端口建立连接，或通过 RPC 从 deamon 上获取卷配置信息等工作。不通过的工作模式其工作流程亦不同。此 deamon 工作流程如下：

8.1 if (cmd_args->sock_file) //建立 brick 监听端口
if (cmd_args->volfile_server)//同 damon 监听端口建立连接
//damon 命令行不含该参数，所以不会执行两处 if 语句。

8.2 fp = get_volfp (ctx);
//打开卷配置文件，获取文件操作句柄，此卷文件为：
/usr/local/etc/glusterfs/glusterd.vol，参考 4.2

8.3 ret = glusterfs_process_volfp (ctx, fp);
//对卷文件进行解析，并执行该卷信息所对应的 translator 的操作。其详细过程如下：

8.3.1 graph = glusterfs_graph_construct (fp);
//利语法分析器 yyparse（）（google search）函数用解析卷配置文件构建一个 graph 结构体。 在解析过程中调用 xlator_set_type 已经将各个 translator 对应的动态库打开，并获取了相关函数和结构体的指针。参考：8.3.2.1.2 和 libglusterfs/src/graph.y 文件。

8.3.2 ret = glusterfs_graph_prepare (graph, ctx);
//对构建的 graph 结构预处理一下，其内部处理情况为：

8.3.2.1 ret = glusterfs_graph_settop (graph, ctx);
//设置 graph 的 top translator，默认卷配置文件中的最后一个 translator

8.3.2.1 ret = glusterfs_graph_readonly (graph, ctx);
ret = glusterfs_graph_acl (graph, ctx);
ret = glusterfs_graph_mac_compat (graph, ctx);
//根据 cmd_args 参数信息调用 glusterfs_graph_insert（）函数来决定是否额外添加一个 translator，并设置为 raph 的 top。其添加过程如下：

8.3.2.1.1 ixl = GF_CALLOC (1, sizeof (*ixl), gf_common_mt_xlator_t);
ixl->ctx = ctx;
ixl->graph = graph;
ixl->options = get_new_dict ();
ixl->name = gf_strdup (name);
//创建一个 translator 结构体，并初始化

8.3.2.1.2 if (xlator_set_type (ixl, type) == -1)
//设置 translator 的类型，并根据类型调用 xlator_dynload 载入相应动态库和函数，其载入细节如下：

8.3.2.1.2.1 handle = dlopen (name,

RTLD_NOW|RTLD_GLOBAL);
xl->dlhandle = handle;
//调用 dlopen 打开动态库句柄，并赋值。

8.3.2.1.2.2 if (!(xl->fops = dlsym (handle, "fops")))
if (!(xl->cbks = dlsym (handle, "cbks")))
if (!(xl->init = dlsym (handle, "init")))
if (!(vol_opt->given_opt = dlsym (handle, "options")))
//利用 dlsym 打开动态库中库函数，获取函数指针

8.3.2.1.2.3 fill_defaults (xl);
//设置 Xl translator 的其他未设置的选项为默认选项

8.3.2.1.3 if (glusterfs_xlator_link (ixl, graph->top) == -1)
//将新创建的 translator 与 graph->top 建立父子关系。建立过程如下：

8.3.2.1.3.1 xlparent = (void *) GF_CALLOC (1, sizeof (*xlparent), gf_common_mt_xlator_list_t);
xlchild = (void *) GF_CALLOC (1, sizeof (*xlchild), gf_common_mt_xlator_list_t);
//创建一个 parent 和一个 child

8.3.2.1.3.2 xlparent->xlator = pxl;
for (tmp = &cxl->parents; *tmp; tmp = &(*tmp)->next);
*tmp = xlparent;
//赋值，并将 graph->top 的兄弟的 parents 指针指向 xlparent

8.3.2.1.3.3 xlchild->xlator = cxl;
for (tmp = &pxl->children; *tmp; tmp = &(*tmp)->next);
*tmp = xlchild;
//赋值，并将新创建 translator 的兄弟的 child 指针指向 Xlchild，完成父子关系的建立

8.3.2.1.4 glusterfs_graph_set_first (graph, ixl);
graph->top = ixl;
//将新添加的 translator 设置为 graph 的 first 和 top，完成。

8.3.3 ret = glusterfs_graph_activate (graph, ctx);
//初始化 graph 中的各个 translator，创建 socket，建立事件监听函数。
其详细过程如下：

8.3.3.1 ret = glusterfs_graph_validate_options (graph);
//验证卷配置文件中的 options 参数的有效性。

8.3.3.2 ret = glusterfs_graph_init (graph);
//参考 8.3.1，自上而下调用 graph 中各个 translator 的 init()函数初始化。
此 damon 只会调用 mgmt 的 init 函数(xlators/mgmt/glusterd/src/glusterd.c)。
在初始化过程中建立/etc/glusterd/下的 vols，peers 等文件夹，创建监听端口，设置事件处理函数，恢复上次 deamon 退出的状态灯操作。简要分析

如下：

8.3.3.2.1 dir_data = dict_get (this->options, "working-directory");

//获取工作主目录

ret = gf_cmd_log_init (cmd_log_filename);

//设置 CLI 命令 log 文件

ret = mkdir (voldir, 0777);

//创建 vols，peers 等工作目录

ret = glusterd_rpcsvc_options_build (this->options);

//设置 PRC server 的选项配置信息 options

8.3.3.2.2 rpc = rpcsvc_init (this->ctx, this->options);

//创建一个 rpc server 结构体，并初始化一些参数信息。利用函数 ret = rpcsvc_program_register (svc, &gluster_dump_prog);添加 gluster_dump_prog 到 svc->programs 链表上。

8.3.3.2.3 ret = rpcsvc_register_notify (rpc, glusterd_rpcsvc_notify, this)

//主次 rpc server 一个 notify 处理函数。将该处理函数添加到 svc->notify 列表上。

8.3.3.2.4 ret = rpcsvc_create_listeners (rpc, this->options, this->name);

//利用 rpc 类型调用 ret = rpcsvc_create_listener (svc, options, transport_name);创建 listener，创建过程如下：

8.3.3.2.4.1 trans = rpcsvc_transport_create (svc, options, name);

//创建一个 rpc_transport_t 结构体，该结构体动态载入 soket 库以及其函数指针，创建一个 socket，其创建过程如下：

8.3.3.2.4.1.1 trans = rpc_transport_load (svc->ctx, options, name);

//动态载入相应类型的 RPC 库并调用库的 init 初始化。

8.3.3.2.4.1.2 ret = rpc_transport_listen (trans);

//调用 sokcet 库的 listen 建立监听端口，并调用 ctx->event_pool 的 event_register_epoll 函数将 sokcet 句柄利用 epoll_ctl 监听。详细见参考 soket 的 listen 函数的源码和 event_register_epoll 函数。

8.3.3.2.4.1.3 ret = rpc_transport_register_notify (trans, rpcsvc_notify, svc); rpcsvc_notify

//注册 trans 的 notify 处理函数为

8.3.3.2.4.2 listener = rpcsvc_listener_alloc (svc, trans);

//创建 listener，并将该 rpc server 和 trans 赋值给 listener。将该 listener 添加到 prc server 的 svc->listeners 链表上。

8.3.3.2.5 ret = glusterd_program_register (this, rpc, &glusterd1_mop_prog);

ret = glusterd_program_register (this, rpc, &gd_svc_cli_prog);

ret = glusterd_program_register (this, rpc, &gd_svc_mgmt_prog);

ret = glusterd_program_register (this, rpc, &gluster_pmap_prog);

ret = glusterd_program_register (this, rpc, &gluster_handshake_prog);

//注册 5 个事件处理结构体到 rpc->programs 列表上

8.3.3.2.6 ret = configure_syncdaemon (conf);

//a.定义 RUN_GSYNCD_CMD(prf)，该函数用来 damon 启动执行命令

//b.配置 geosync 信息。

8.3.3.2.7 ret = glusterd_restore ();

//从/etc/glusterd/目录获取获取以前操作的 peer，volume，bricks 等信息，保存到结构体中。

8.3.3.2.8 glusterd_restart_bricks (conf);

//根据上次 damon 运行状态针对每个 brick 启动一个 brick 服务

“glusterd_brick_start (volinfo, brickinfo);”该函数会调用

“ret = glusterd_volume_start_glusterfs (volinfo, brickinfo);”启动卷服务。该函数前面大部分工作在于设置命令行参数，最后调用

“ret = gf_system (cmd_str);”执行 ret = execvp (argv[0], argv)来创建新的进程启动服务。

最后并确定是否启动 nfs 服务：“glusterd_check_generate_start_nfs ();”

8.3.3.2.9 ret = glusterd_restart_gsyncds (conf);

//启动远程同步功能

8.3.3.2 ret = glusterfs_graph_parent_up (graph);

//调用卷配置文件中的各个 translator 的 notify 函数，由于 ctx->master 为空，所以不会执行 ret = xlator_notify (ctx->master, GF_EVENT_GRAPH_NEW, graph);调用 mgmt 的 notify 函数，发送 GF_EVENT_PARENT_UP 命令。该 mgmt 调用 default_notify (this, event, data); 没有做什么具体操作，可以忽略。

9. ret = event_dispatch (ctx->event_pool);

//监听事件池中注册的句柄，即 8.3.3.2.4.1.2 创建 soket 的句柄。并调用事件池中注册的函数处理，即 event_pool->ops->event_dispatch (event_pool);。详细过程如下：

9.1 ret = epoll_wait (event_pool->fd, event_pool->evcache, event_pool->evcache_size, -1);

//监听 socket 句柄，等待事件。

9.2 ret = event_dispatch_epoll_handler (event_pool,events, i);

//调用创建 socket 时候注册的事件处理函数处理事件。

3 GlusterFS 之 server

Server 在 host 上运行的应用程序为 glusterfsd，通过 ps –ef 查看进程信息可以看到一个 brick 对应一个服务，一个服务对应一个监听端口。其进程信息为：

```
/usr/local/sbin/glusterfsd --xlator-option nfs-volume-server.listen-port=24024
-s localhost
--volfile-id nfs-volume.vm-pool-1.root-export5
-p /etc/glusterd/vols/nfs-volume/run/vm-pool-1-root-export5.pid
-S /tmp/814c4d0ac6f6b853b760b10d51083429.socket
--brick-name /root/export5
--brick-port 24024 -l /usr/local/var/log/glusterfs/bricks/root-export5.log
```

该信息表明了该服务对应的 brick， port 以及命名的监听 sokcet 文件等信息。其启动由

damon 执行 ret = gf_system (cmd_str)来启动的，参考第二章的 8.3.3.2.8。该服务启动流程大致与 damon 类似，其详细流程如下：

1,2,3,4,5,6,7 与第二章的步骤大体一致，只是第四部运行模式已经变为 GF_SERVER_PROCESS，其配置文件为 glusterfsd.vol（为空），第五步的 log 记录文件名已改变。

8 ret = glusterfs_volumes_init (ctx);

//创建 brick 监听端口，同 damon 建立连接获取相关 brick 配置信息。其详细过程如下：

8.1 if (cmd_args->sock_file) {

ret = glusterfs_listener_init (ctx);}

//创建 brick 监听端口，注册事件处理函数，监听客户端发送过来的信息，详细过程如下：

8.1.1 ret = rpcsvc_transport_unix_options_build (&options,cmd_args->sock_file);

//设置 rpc-server 的参数信息 options，包含 sock_file,等信息。

8.1.2 rpc = rpcsvc_init (ctx, options);

//创建个 prc-server，并初始化，类似第二章的 8.3.3.2.2.

8.1.3 ret = rpcsvc_register_notify (rpc, glusterfs_rpcsvc_notify, THIS);

//注册 rpc-server 的通知处理函数，参考上章的相关信息

8.1.4 ret = rpcsvc_create_listeners (rpc, options, "glusterfsd");

//创建 listeners，参考上章的相关信息

8.1.5 ret = rpcsvc_program_register (rpc, &glusterfs_mop_prog);

//注册事件处理，参考上章相关信息

8.2 if (cmd_args->volfile_server) {

ret = glusterfs_mgmt_init (ctx);}

//同 damon 建立连接，获取 brick 配置信息。

8.2.1 ret = rpc_transport_inet_options_build (&options, host, port);

//设置字典 options 的 host 和 port 等信息。即连接本 host 的 deamon 的监听端口。

8.2.2 rpc = rpc_clnt_new (options, THIS->ctx, THIS->name);

//创建一个 RPC

8.2.2.1 rpc = GF_CALLOC (1, sizeof (*rpc), gf_common_mt_rpcclnt_t);

rpc->reqpool = mem_pool_new (struct rpc_req,
RPC_CLNT_DEFAULT_REQUEST_COUNT);

rpc->saved_frames_pool = mem_pool_new (struct saved_frame,
RPC_CLNT_DEFAULT_REQUEST_COUNT);

//创建一个 rpc 结构体，并初始化两个内存池，参考上章 3.5

8.2.2.2 ret = rpc_clnt_connection_init (rpc, ctx, options, name);

//初始化 rpc 的变量 rpc->conn

8.2.2.2.1 conn->trans = rpc_transport_load (ctx, options, name);

//初始化 conn 的 trans 变量。设置 trans 的函数指针等变量。

8.2.2.2.1.1 trans = GF_CALLOC (1, sizeof (struct rpc_transport),
gf_common_mt_rpc_trans_t);

//创建 trans，并初始化一些变量

8.2.2.2.1.2 handle = dlopen (name, RTLD_NOW|RTLD_GLOBAL);

trans->ops = dlsym (handle, "tops");

trans->init = dlsym (handle, "init");
trans->fini = dlsym (handle, "fini");
trans->reconfigure = dlsym (handle, "reconfigure");
//打开 rpc 库 socket，并获取 socket 的相关函数指针赋值给 trans 的函数指针变量。

8.2.2.2.1.3 ret = trans->init (trans);
//调用 socket 库的 init 函数初始化。

8.2.2.2.2 ret = rpc_transport_register_notify (conn->trans, rpc_clnt_notify,conn);
//注册 conn->trans 的通知调用函数为 rpc_clnt_notify

8.2.2.3 ret = rpc_clnt_register_notify (rpc, mgmt_rpc_notify, THIS);
//注册 rpc 的通知调用函数 mgmt_rpc_notify

8.2.3 ret = rpcclnt_cbk_program_register (rpc, &mgmt_cbk_prog);
//注册 rpc 的回调处理函数结构 mgmt_cbk_prog

8.2.4 ret = rpc_clnt_start (rpc);
//调用 rpc 的 conn 启动 trans

8.2.4.1 ret = rpc_transport_connect (trans, conn->config.remote_port);
//调用 trans 的变量 ops 的函数指针 connet 连接 damon 监听端口。this->ops->connect (this, port);注册事件句柄和事件处理函数，利用 epool 监听句柄。详情参考第 4 章的 8.2

9 ret = event_dispatch (ctx->event_pool);
//调用创建 socket 时候注册的事件处理函数处理事件。

4 GlusterFS 之 client

Client 端通过 mount server 上的一个卷启动。Client 端需要从 damon 端获取卷配置信息，并解析卷配置信息设置相应的 translator 树形图，最后根据利用 protocol/client 的 translator 利用 rpc 的 sokcet 连接不同 server 端对应的 protocol/server 的 translator 进行通信。

ps –ef 可以看到客户端对应的进程信息为：gluterfs –p

client 详细的执行流程如下：

1,2,3,4,5,7 基本与上章基本一致，不同在于此时 4 的运行模式为 GF_CLIENT_PROCESS。除此之外，此时的 client 第 6 步需要创建一个 fuse translator，该 translator 负责 while 监听 fuse 设备文件。

6. ret = create_fuse_mount (ctx);
//该根据 mount 时候的参数，判断是否有 mount point，如果有创建一个 translator 结构体，并初始化。

6.1 master = GF_CALLOC (1, sizeof (*master),gfd_mt_xlator_t);
master->name = gf_strdup ("fuse");
//创建一个 master translator 结构体。

6.2 if (xlator_set_type (master, "mount/fuse") == -1)
//调用 ret = xlator_dynload (xl) 动态载入 fuse 库，并载入函数地址赋值给 master 的函数指针等。详情参考第一章的 8.3.2.1.2 信息。

6.3 dict_set_static_ptr (master->options,ZR_MOUNTPOINT_OPT,

cmd_args->mount_point)
dict_set_double (master->options, ZR_ATTR_TIMEOUT_OPT,
cmd_args->fuse_attribute_timeout)
dict_set_int32 (master->options, "client-pid",
cmd_args->client_pid)
//设置 master->options 字典的不同类型的值。
6.4 ret = xlator_init (master);
//调用 fuse 库的 init 函数，主要初始化 master 的指针 priv 指针变量。
6.4.1 priv = GF_CALLOC (1, sizeof (*priv), gf_fuse_mt_fuse_private_t);
this_xl->private = (void *) priv
//申请一个 gf_fuse_mt_fuse_private_t 内存并赋值给 master
6.4.2 priv->mount_point = gf_strdup (value_string);
//设置 mount point
6.4.3 priv->fd = gf_fuse_mount (priv->mount_point, fsname,
"allow_other,"
"max_read=131072");
//设置 proc 监视句柄：fd = open ("/dev/fuse", O_RDWR);
6.4.5 priv->fuse_ops = fuse_std_ops;
//设置 fuse 的函数指针结构体
6.5 ctx->master = master;
//将创建的 fuse translator 作为 ctx 的 master。
7 ret = daemonize (ctx);
//damon 化该 client 端。
8 ret = glusterfs_volumes_init (ctx);
//利用 rpc 的 sokcet 库与 glusterfs 的 damon 建立连接，获取卷信息文件，初始化 client 端的 translator 并初始化，注册事件处理函数。
8.1 ret = rpc_transport_inet_options_build (&options, host, port);
rpc = rpc_clnt_new (options, THIS->ctx, THIS->name);
ret = rpc_clnt_register_notify (rpc, mgmt_rpc_notify, THIS);
ret = rpcclnt_cbk_program_register (rpc, &mgmt_cbk_prog);
//详情参考第二章的 8.2.
8.2 ret = rpc_clnt_start (rpc);
//调用上面第 socket ops 结构体中的 connet 连接函数启动连接。
8.2.1 ret = socket_client_get_remote_sockaddr (this, SA (&sockaddr),
&sockaddr_len, &sa_family);
priv->sock = socket (sa_family, SOCK_STREAM, 0);
//初始化连接地址类型和 sock 句柄，epoll 会监视该句柄的事件
8.2.2 ret = connect (priv->sock, SA (&this->peerinfo.sockaddr),
this->peerinfo.sockaddr_len);
//建立连接
8.2.3 priv->idx = event_register (ctx->event_pool, priv->sock,
socket_event_handler, this, 1, 1);
//注册 epool 监听句柄 priv->sock 和事件处理函数 socket_event_handler
其中 socket_event_handler 会处理三种类型事件：

A. ret = socket_connect_finish (this);

B. ret = socket_event_poll_out (this);

C. ret = socket_event_poll_in (this);

三种事件都会调用 conn->trans 中注册的 rpc_transport_notify (this, event, this)函数来处理事件。该函数会根据事件类型走不同的分支处理函数，如果是建立连接的事件则会走 8.1 中注册的：

ret = rpc_clnt_register_notify (rpc, mgmt_rpc_notify, THIS);

该函数会调用 ret = glusterfs_volfile_fetch (ctx);函数来获取卷文件信息并解析初始化和 activation。glusterfs_volfile_fetch 的执行流程如下：

8.2.3.1 ret = mgmt_submit_request (&req, frame, ctx,
&clnt_handshake_prog, GF_HNDSK_GETSPEC
xdr_from_getspec_req,mgmt_getspec_cbk);
//向 damon 提交 lnt_handshake_prog 中的 GF_HNDSK_GETSPEC 类型请求，并调用 mgmt_getspec_cbk 函数处理请求，其详情如下：

8.2.3.1.1 tmpfp = tmpfile ();
fwrite (rsp.spec, size, 1, tmpfp);
fflush (tmpfp);
//获取 damon 的卷数据流信息写到 tmpfp 文件中。

8.2.3.1.2 ret = glusterfs_process_volfp (ctx, tmpfp);
//解析卷配置文件，解析如下：

8.2.3.1.2.1 graph = glusterfs_graph_construct (fp);
//根据文件构建 translator 图

8.2.3.1.2.2 ret = glusterfs_graph_prepare (graph, ctx);
//预处理

8.2.3.1.2.3 ret = glusterfs_graph_activate (graph, ctx);
//激活该图，激活如下：

8.2.3.1.2.3.1 ret = xlator_notify (ctx->master,
GF_EVENT_GRAPH_NEW, graph);
//调用 master 的通知函数，通知类型 GF_EVENT_GRAPH_NEW，即调用 fuse 库的 nofity 函数处理 GF_EVENT_GRAPH_NEW 类型，什么也没由做，

8.2.3.1.2.3.2 ret = xlator_notify (trav,
GF_EVENT_PARENT_UP, trav);
//调用其他的 translator nofity 函数，通过分析各个 translator 的 notify（default_notify）源码可以知道，该函数最终会调用最底层的 nofity 函数，即 protocol/client 的 nofity 函数，而该函数会向上调用 default_notify 函数通知 GF_EVENT_CHILD_UP 函数，最终会执行 master 的 nofity 函数，即执行 fuse 的 notify 函数：

8.2.3.1.2.3.2.1 ret = pthread_create
(&private->fuse_thread, NULL,

fuse_thread_proc, this);
//启动一个新进程调用 fuse_thread_proc 处理函数，该处理函数在 for 循环中利用 Readv 读取上面 6.4.3 打开的设备句柄。从此进入工作流程。

9. ret = event_dispatch (ctx->event_pool);
//监听第 8.2.1 步注册事件处理函数的文件句柄。进入监听 socket 过程，首先监听 socket 建立连接事件，启动 fuse 的 while 循环进入客户端工作流程。

5. 总结

该源码分析只做参考，如果有疑问一同讨论分析。