GlusterFS 主函数工作流程

```
/************************************************
*  $Version:    glusterfs-2.0.3
*  $Draft:      2009.09
*  $Email:      NNing.a#gmail.com
*  $Link:       http://blog.chinaunix.net/u/33029
*
*  *** ~~~ 仅供学习交流，请勿用于其它用途！ ~~~ ***
************************************************/
```

---

glusterfs 主函数实现在<glusterfs>/glusterfsd/src/glusterfsd.c 文件中。主函数大概分两步，一是设置 glusterfs 进程运行的相关环境变量，另一个是转入事件处理部分。需要注意的几个模块：

1.开辟 IO 缓存

```
ctx->page_size   = 128 * 1024;     // 128k
ctx->iobuf_pool = iobuf_pool_new (8 * 1048576, ctx->page_size + 4096); // (8M, 132K)
```

开辟一个 IO 缓冲池(iobuf_pool)的过程涉及到主要三个数据结构：struct iobuf_pool，struct iobuf_arena，struct iobuf，其中从整体上理解就是 iobuf_pool 包含了 iobuf_arena，iobuf_arena 又包含了数个 iobuf，实际上他们是通过链表来相互关联起来的。

```
// iobuf 结构体 -- 最小的 io buffer 单元
struct iobuf {
        union {   // 链接同一个 arena 中其它 iobuf 的指针
                struct list_head        list;
                struct {
                        struct iobuf *next;
                        struct iobuf *prev;
                };
        };
        struct iobuf_arena   *iobuf_arena;  // 从属的 iobuf_arena

        gf_lock_t               lock; /* for ->ptr and ->ref */
        int                     ref;   /* 0 == passive, >0 == active */

        void                    *ptr;   /* usable memory region by the consumer */
                                // 初始化 iobuf_arena 块时，每个 iobuf 的地址会被映射
```

```
};

// iobuf_arena 结构体
struct iobuf_arena {
        union {   // 链接同一个 iobuf_pool 中其它 iobuf_arena 的指针
                struct list_head                list;
                struct {
                        struct iobuf_arena *next;
                        struct iobuf_arena *prev;
                };
        };
        struct iobuf_pool   *iobuf_pool;   //从属的 iobuf_pool

        void                *mem_base; // 这块 arena 的基地址，创建时由 mmap（）
                                       // 映射到内存
        struct iobuf        *iobufs;       /* allocated iobufs list */
                                       // 指向 arena 中 iobufs 链表的首节点

        int                 active_cnt;
        struct iobuf        active;       /* head node iobuf (unused by itself) */
        int                 passive_cnt;
        struct iobuf        passive;      /* head node iobuf(unused by itself) */
};

// iobuf_pool 结构体
struct iobuf_pool {
        pthread_mutex_t       mutex;
        size_t                page_size;    /* size of all iobufs in this pool */
        size_t                arena_size;    /* size of memory region in arena */

        int                   arena_cnt; // iobuf_pool 中 arena 块的数目
        struct iobuf_arena    arenas;       /* head node arena (unused by itself) */
                                       // 所有的 arena
        struct iobuf_arena    filled;       /* arenas without   free iobufs */
                                       // 没有剩余 iobuf 的 arena
        struct iobuf_arena    purge;        /* arenas which can be purged */
                                       // 可以被清空的 arena
};
```

新建一个 IO 缓冲池的过程大致为：开辟一个 IO 缓冲空间(新建一个 struct iobuf_pool 变量)，然后向这个池子(空间)添加 arena 块(iobuf_pool_add_arena()函数实现)。

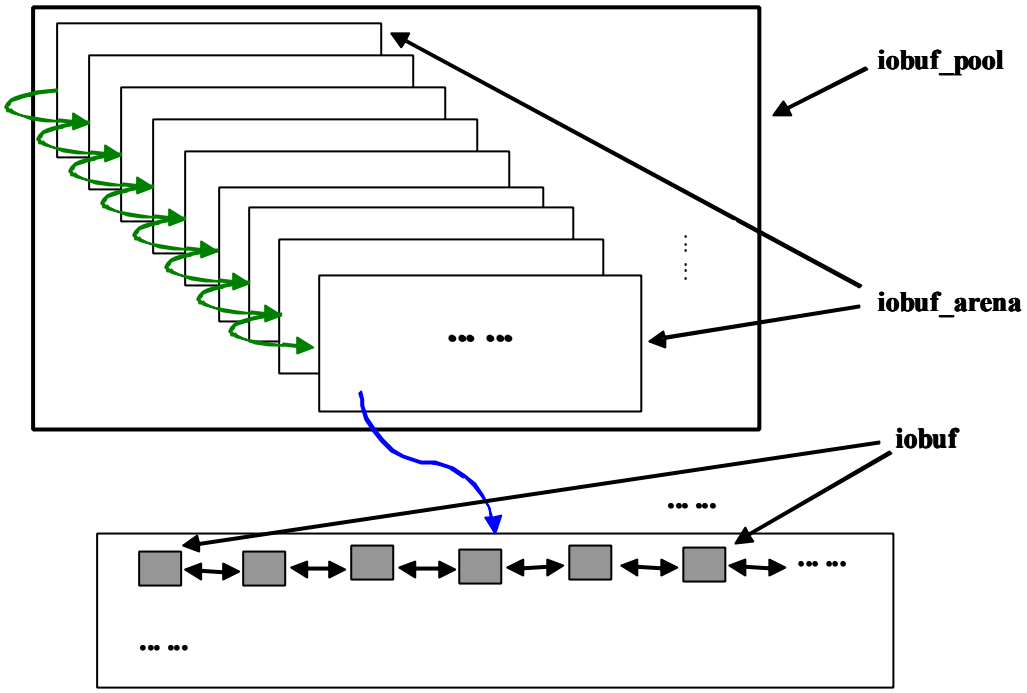
添加 arena 块的过程也可以再细分为两个步骤：(a)开辟一个 iobuf_arena 块空间，并用

mmap( )函数对这个块做内存映射；(b)初始化这个 iobuf_arena 。

初始化 iobuf_arena 块的的过程：在 arena 中划分出更小的块（连续分配的），即 iobuf，每一块的大小为 page_size (132K)，并把每一个 iobuf 小块设置为从属于这个 iobuf_arena（iobuf->iobuf_arena = iobuf_area;）；接下来就是确定每一个 iobuf 的地址，这个可以由 arena 的基地址通过偏移走位得到(offset += page_size;)；最后是把这些 iobuf 相互链接起来(struct iobuf_arena 结构体中的 iobufs 字段就是指向这个链表的)。

注：glusterfs 在创建 iobuf_pool 时，每一个 iobuf_arena 的大小为 8M (8 * 1048576,)，page_size 为 132K(ctx->page_size + 4096)，所以每个 iobuf_arena 中的 iobuf 个数 62 个(iobuf_cnt = iobuf_pool->arena_size / iobuf_pool->page_size)。

为便于理解，可大致画出 IO 缓冲池的示意图：



2.新建事件池

```
ctx->event_pool = event_pool_new (DEFAULT_EVENT_POOL_SIZE);   // 16384(16K)
```

其中 DEFAULT_EVENT_POOL_SIZE 为事件池的大小，在这里为 16384 。event_pool_new( ) 函数将会调用到 **event_ops_epoll.new( )** 。 event_ops_epoll 是 libglusterfs/src/event.h 中定义的一个 static 的变量(在 event.c 中定义) --- 在这里用到了 EPOLL 消息轮询机制，即

```
static struct event_ops event_ops_epoll = {
        .new             =     event_pool_new_epoll,
        .event_register  =     event_register_epoll,
```

```
        .event_select_on    =     event_select_on_epoll,
        .event_unregister   =     event_unregister_epoll,
        .event_dispatch     =     event_dispatch_epoll
    };
```

所以 new 指针实际上指向的是 event_pool_new_epoll( )函数的地址。从 event_pool_new_epoll( ) 函数的实现来看，新建的事件池可处理 DEFAULT_EVENT_POOL_SIZE（16384）个事件(即在事件池中可注册这么多个事件)，并为这个事件池里所有的事件创建一个 epoll_wait( ) 函数可以进行监听的句柄： epfd = epoll_create (count); 即 epoll_wait( )函数通过 epfd 即可监听事件池中所有的已注册的文件 fd 的状态。

创建了新的事件池后，还必须有相关的操作，即把 event_ops_epoll 作为这个事件池的应有的操作：

```
event_pool->ops = &event_ops_epoll;
```

附：

```
//struct event_pool 结构体定义
struct event_pool {
  struct event_ops *ops;       //事件池相关操作
  int fd;                 // epoll_create( ) 函数返回的句柄
  int breaker[2];         // used for pipe (in event_pool_new_epoll())

  int count;              // 事件池大小
  struct {
    int fd;               // 所关联的文件描述符
    int events;           // fd 关注的事件，如 EPOLLIN,EPOLLOUT 等
    void *data;
    event_handler_t handler;// 事件处理句柄
  } *reg;                 // 已注册事件的数组指针

  int used;               // 已经注册的文件描述符 fd 的个数
  int idx_cache;          //
  int changed;            // 当有新事件注册时，这个字段设置为 1

  pthread_mutex_t mutex;
  pthread_cond_t cond;

  void *evcache;          // event cache
  int evcache_size;       // event cache size
};

// 事件池相关操作
struct event_ops {
```

```
        struct event_pool * (*new) (int count);

        int (*event_register) (struct event_pool *event_pool, int fd,
                               event_handler_t handler,
                               void *data, int poll_in, int poll_out);

        int (*event_select_on) (struct event_pool *event_pool, int fd, int idx,
                                int poll_in, int poll_out);

        int (*event_unregister) (struct event_pool *event_pool, int fd, int idx);

        int (*event_dispatch) (struct event_pool *event_pool);
    };
```

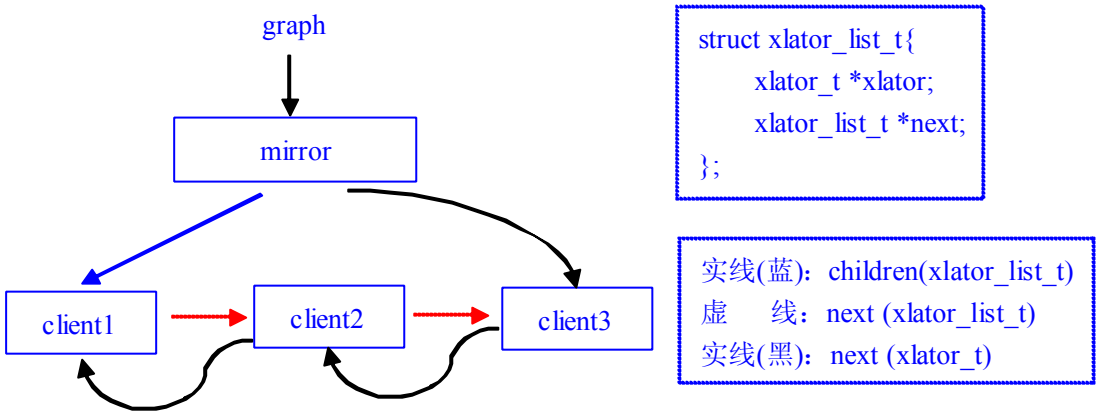
详细实现过程请自行查看<glusterfs>/libglusterfs/src/event.c 文件中的源码。

3.读取并分析配置文件

读取配置文件: specfp = _get_specfp(ctx);

分析配置文件：graph = _parse_specfp(ctx, specfp);

读取和分析配置文件的详细过程我们不深入，对其最终产生的解析器树(即 graph)才是重点。根据 gdb 显示的信息，可大致画出 graph 的结构图：



注：这里 volfile 中定义的是三个 mirror

这里涉及到两个链表，一个是 xlator *prev, *next; 一个是 xlator_list *parents, *children。对于"client1","client2"和"client3"，其 children 均为为空(即 0x0)，parents 均指向"mirror"。但链表 xlator_list_t 中，"mirror"的 children，其中 xlator 是"client2", next 指向的是"client2",即 next->xlator 为"client2"; 继续地，next->xlator 为"client3",如上图红虚线所示。

对于由 xlator_t *prev, *next 维护的链表，它只是把所有的解析器节点连接成一条"线",

没有体现上下层关系。大致顺序为：graph -> "mirror" -> "client3" -> "client2" -> "client1"。

需要注意的是，在分析配置文件生成节点树的过程中，各个节点对应的解析器模块(即.so 动态库文件)会被加载(dlopen()函数实现)，并将 init,fini,notify,fops,mops,cbks 等地址映射起来。 (这个可以在 gdb 跟调过程中设置断点 dlsym 观察到)

4.添加 fuse 解析器

这一过程主要是由函数_add_fuse_mount()实现，即语句：

graph = _add_fuse_mount (graph)

这一步主要是在原有解析器树上添加一个节点(A: xlator_t)，即"mount/fuse"，并把这个节点作为最顶层。同时还要维护 xlator_list_t 链表，即也添加一个新节点(B: xlator_list_t)，并把原链表连接到节点(A: xlator_t)的 children，即新节点处在原链表的上一层。

同时加载"fuse.so"动态库文件,在这里其完全路径为

"/usr/local/lib/glusterfs/2.0.3/xlator/mount/fuse.so"。

(a) 设置一些 fuse 相关的选项

```
……
ctx = graph->ctx;   // 关联程序运行环境
cmd_args = &ctx->cmd_args;
……
xlchild->xlator = graph;
……
top->children = xlchild;
top->ctx = graph->ctx;
top->next = gf_get_first_xlator(graph);        // 即 graph
top->options = get_new_dict();
ret = dict_set_static_ptr(top->options, ZR_MOUNTPOINT_OPT,
                          cmd_args->mount_point); // 设置字典中的"mountpoint"选项
……
ret=dict_set_static_ptr(top->options,ZR_DIRECT_IO_OPT,"enable");//"direct-io-mode"
```

若运行时参数中指定了"attribute-timeout"， "entry-timeout"， "strict-volfile-check"等时，也需要进行设置：

```
ret = dict_set_double(top->options, ZR_ATTR_TIMEOUT_OPT,
                      cmd_args->fuse_attribute_timeout);   //设置"attribute-timeout"
……
ret = dict_set_double(top->options, ZR_ENTRY_TIMEOUT_OPT,
                      cmd_args->fuse_entry_timeout);   //设置"entry_timeout"
……
ret = dict_set_int32(top->option, ZR_STRICT_VOLFILE_CHECK,
```

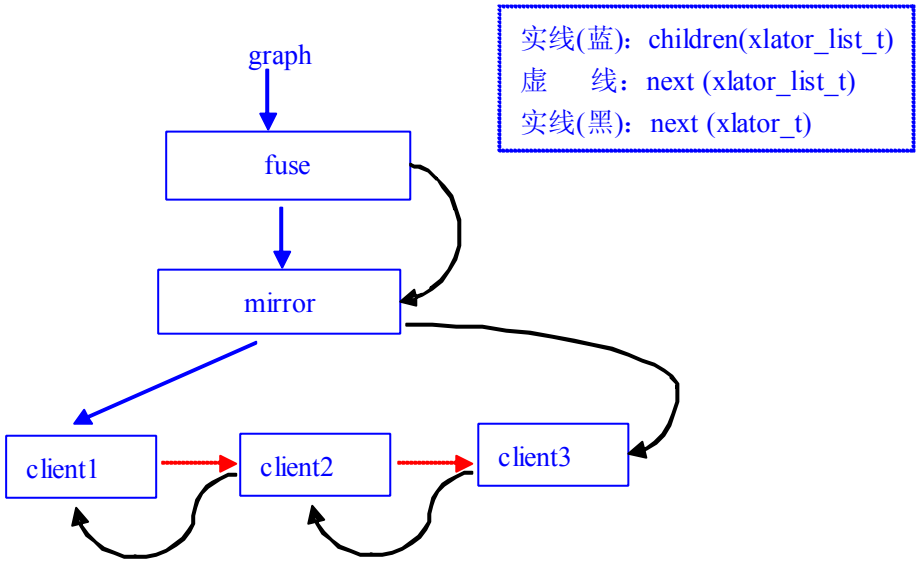
cmd_args->volfile-check); //设置"strict-volfile-check"

(b) xlator_set_type(top, ZR_XLATOR_FUSE);

这一步将新建的解析器节点与"mount/fuse"关联起来(从函数名上看，这个函数主要是设置该节点的类型…)，加载相应的库文件，并把动态库中的函数地址映射出来。流程大概如下：

```
asprintf(&name, "%s/%s.so", XLATORDIR,type);      // 获取动态库的完全路径
……
handle = dlopen(name, RTLD_NOW | RT_GLOBAL);      // 打开动态库文件
……
xl->fops = dlsym(handle, "fops"); // 链接--将动态库文件中的 fops 地址跟 xl->fops 对应起来
……
xl->mops = dlsym(handle, "mops");
……
xl->cbks = dlsym(handle, "cbks");
……
xl->init dlsym(handle, "init");
……
xl->fini = dlsym(handle, "fini");
……
xl->notify = dlsym(handle, "notify");
……
vol_opt ->given_opt = dlsym(handle, "options"); // 添加相关选项
list_add_tail (&vol_opt->list, &xl-.>volume_options);
```

添加"fuse"节点后的树结构为：



对于"fuse"节点，其 chidlren 指向"mirror"，即 graph->children->xlator 指向"mirror",而 chidlren 中的 next 指针为空，即 graph->children->next == 0x0，这说明"fuse"的子节点只有一

个，即"mirror"。而对于"mirror"，其 children 指向"client1"，即"mirror"->children->xlator 指向"client1"，且其 next(即"mirror"->children->next)指向"client2"; 同理"mirror"->chidlren->next->next 指向"client3"。

附 xlator_list_t 的定义：

```
typedef struct xlator_list {
        xlator_t            *xlator;
        struct xlator_list  *next;
} xlator_list_t;
```

5.解析初始化

glusterfs_graph_init(graph, fuse_volume_found);

这个函数实现了对树(graph)上各个节点的初始化。其初始化的过程中解析器的 init()函数是自下往上的，即一层一层往上进行。而 notify()函数的运行顺序是从树顶开始，逐步向下的。对于我们的配置，生成的树有三层，如上图所示。

其函数调用结构(过程)大概为：

glusterfs_graph_init( ) **--> (a)**
\-> _xlator_garph_init( ) **--> (b)**
\-> xlator_tree_init( ) **--> (c)**
\-> xlator_init_rec( ) **--> (d)**

(b) _xlator_graph_init( )的关键部分：

```
while(trav){
        if(!trav->ready){ // xlator_tree_init("fuse")后，其下的每个解析器的 ready 字段
                          // 都为 1，即此后不会再调用 xlator_tree_init( )
            if((ret = xlator_tree_init(trav)) < 0){
            ... ...
            return ret;
            }
        }
        trav = trav->next;  // 这一步在顶层有多个解析器时才显作用，如在服务器端
                            // 导出多个"存储块"(brick)。顶层的每一个解析器都有各自
                            // 的子节点。在 client 端最顶层是"fuse"，故只需调用一次。
}
```

(c) xlator_tree_init( )会调用到 xlator_init_rec( )函数，并发出通知(各解析器的 notify 的函数)，其关键部分：

```
ret = xlator_init_rec(top);
if(ret == 0 && top->notify){
        top->notify(top,GF_EVENT_PARENT_UP, NULL);          ---> (e)
}
```

(d) **xlator_init_rec( )**是一个递归函数，如下为其实现的关键步骤：

```
static int32_t xlator_init_rec(xlator_t *xl)
{
    xlator_list_t *trav;
    ... ...
    trav = xl->children; // 看是否有子节点；若有，则继续向下一层调用
                         // trav -> "mirror" -> "client1"
    while ( trav )
    {
        ret = 0;
        ret = xlator_init_rec(trav->xlator);
        ... ...
        trav = trav->next; // 这一个语句主要是用于初始化处在同一层的节点
                           // "client1" -> "client2" -> "client3"
    }
    if ( !ret && xl->ready ) {
        ret = -1;
        if ( xl->init ) {
            ret = xl->init ( xl );
            ... ...
        }
    }
}
```

(e) 由 xlator_tree_init( )中还可看到，只有对"fuse"解析器的调用了 notify( )函数。因为传递的消息是"GF_EVENT_PARENT_UP",故其对应的处理分支是：

```
case GF_EVENT_PARENT_UP:
{
    default_notify (this, GF_EVENT_PARENT_UP, data);
    break;
}
```

此分支只做默认处理，即调用了 default.c 中的 default_notify( )函数，其处理过程大致为：

```
case GF_EVENT_PARENT_UP:
{
    xlator_list_t *list = this->children;
    while (list)
    {
        list->xlator->notify (list->xlator, event, this);
        list = list->next;
    }
}
```

"fuse"解析器的 notify 做了默认处理，传给了其子节点"mirror"；参考 AFR 解析器的 notify

源码也发现采用了默认处理。再传给下一层就是各个"client"子节点了。"client"的 notify 过程完成了子卷映像(各个"client")与远程物理服务器建立连接。

对于函数调用结构和最下层的递归函数有个大致了解之后，我们可以知道各解析器节点的初始化过程主要包括运行 init()函数和 notify()函数两部分。根据如上所示的解析器节点结构图，我们可以大略描述整个节点树的 init()的顺序大致为："client1" -> "client2" -> "client3" -> "mirror" -> "fuse"，总共分为三层； notify()的顺序大致为： "fuse" -> "mirror" -> "client1" -> "client2" -> "client3"。下面依次对各层的初始化过程进行简单描述。

(1) 对"client"的初始化

(a) init

对"client"的初始化实质上是为 client 与各个远程服务器建立连接前做好准备，连接的过程，即完成远程存储块在 client 端本地的子卷映像的挂载(即一个远程存储块对应一个 client 的子卷映像)。

这个过程主要是调用到 client-protocol.c : init( )函数，其实现过程大概如下：

首先是从字典中读取"remote-subvolume"、"frame-timeout"、"ping-timeout"等参数值。

接下来是调用了 transport_load( )函数，这个函数主要有两部分：首先判断传输方式，比如 tcp 还是 ib-verb 连接方式；若是 tcp，则是 inet(ipv4)还是 inet6(ipv6)。明确好传输方式后，确定要加载的模块(即 transport/socket.so 模块)并加载；接下来是调用 socket.c : init( )函数，为完成 socket 的建立做好准备。

注：trans = transport_load(this->options, this); 这里 this->options 是选项字典，this 是 client 解析器 *（在这里，this 是 xlator_t 类型，其中有一个域是 option（即这里的 this->options），传一个 this 参数即可，为什么要传两个呢？）*。

在调用 transport_load( )部分，它建立起了两个 socket 连接：

```
for(i=0; i < CHANNEL_MAX; i++) { // here CHANNEL_MAX == 2
    trans = transport_load(this->options, this);
    ... ...
}
```

为什么要同时建立两个 socket 连接呢？一个数据管道一个命令管道？

接下来是 socket.c : init( )，其实这个函数主要是继续调用 socket_init( )函数。socket_init( )函数主要是完成建立 socket 通信通道前的相关参数设置，如检查 socket 解析器字典中"non-blocking-io"， "transport.socket.nodelay"， "windowsize"等参数是否已经设置，若已设置好则把这些相关的参数关联到 transport 参数中，即语句 this->private = priv;

(b) notify

notify( ) 根据得到的消息("GF_EVENT_PARENT_UP")，进入到 switch 的"GF_EVENT_PARENT_UP"分支，并两次调用到 client_protocol_reconnect()函数:

```
case GF_EVENT_PARENT_UP:
{
        client_conf_t *conf = NULL;
        int                   i = 0;
        transport_t     *trans = NULL;
        conf = this->private;
        for (i = 0; i < CHANNEL_MAX; i++) {     //一共要建立 2 个通道
                trans = conf->transport[i];
                if (!trans) {
                        gf_log (this->name, GF_LOG_DEBUG,
                                "transport init failed");
                        return -1;
                }
                conn = trans->xl_private;
                gf_log (this->name, GF_LOG_DEBUG,
                        "got GF_EVENT_PARENT_UP, attempting connect "
                        "on transport");

                client_protocol_reconnect (trans);
        }
}
```

对于 client_protocol_reconnect( ) 函数，其下一层的调用关系大概如下：

```
client_protocol_reconnect( )        ---> client_protocol.c
 \-> tranport_connect( )            ---> transport.c
     \-> socket_connect( )          ---> socket.c
         |-> socket_client_get_remote_sockaddr( )        ---> name.c
         |-> socket( )
         |-> setsockopt( )
         |-> _socket_nonblock( )
         |-> client_bind( )
         |-> connect( )
         \-> event_register( )
             \-> event_register_epoll( )
                 \-> socket_event_handler()
```

在 client_protocol_reconnect( ) 函数中设置了定时器，默认地，若还未连接成功，则每 10 秒钟尝试连接一次：

```
if (conn->connected == 0) {
        tv.tv_sec = 10;       // 默认的重连时间间隔
        gf_log (trans->xl->name, GF_LOG_TRACE, "attempting reconnect");
```

```
ret = transport_connect (trans);     // 尝试连接

conn->reconnect =    gf_timer_call_after (trans->xl->ctx, tv,
                                          client_protocol_reconnect, // 调用自身
                                          trans);
```

继续看 tranport_connect( )函数，它的参数是 transport_t *this。这个函数实际上是 transport_t 结构体操作列表中的 connect 行为(函数指针):

```
ret = this->ops->connect (this);
```

对于"tcp"连接方式， connect 指向的是 **socket_connect( )**函数(在 socket.c 中实现)。socket_connect( )函数主要完成了：

(a)套接口的创建。首先获取远程服务器的地址信息(主要是填充了 struct sockaddr 变量)，并通过 socket( )建立连接，获取套接口文件描述符 fd。 完成 socket 的创建后，利用设置这个套接口的接收、发送缓冲区大小、 TCP 的非延时特性等，并把这个 socket 设置为非阻塞模式。

```
... ...
ret = socket_client_get_remote_sockaddr (this, SA (&sockaddr), &sockaddr_len);
... ...
priv->sock = socket (SA (&sockaddr)->sa_family, SOCK_STREAM, 0);
... ...
setsockopt (priv->sock, SOL_SOCKET, SO_RCVBUF,
            &priv->windowsize, sizeof (priv->windowsize))
... ...
setsockopt (priv->sock, SOL_SOCKET, SO_SNDBUF,
             &priv->windowsize, sizeof (priv->windowsize))
... ...
setsockopt(priv->sock, IPPROTO_TCP, TCP_NODELAY, &on, sizeof(on))
// 非阻塞模式
... ...
ret = __socket_nonblock (priv->sock);
即：
                flags = fcntl (fd, F_GETFL);
                if (flags != -1)
                   ret = fcntl (fd, F_SETFL, flags | O_NONBLOCK);
```

(b)连接：首先是 client_bind( )对 sockaddr 和 fd 进行绑定(实际上用到 Socket 编程接口 bind()函数实现)，然后建立连接：

```
... ...
ret = client_bind (this, SA (&this->myinfo.sockaddr),
                   &this->myinfo.sockaddr_len, priv->sock);
... ...
ret = connect (priv->sock, SA (&this->peerinfo.sockaddr),   this->peerinfo.sockaddr_len);
```

(c) 注册事件：主要是调用到了事件池自身的注册操作(函数)。
priv->idx = **event_register** (ctx->event_pool, priv->sock, socket_event_handler, this, 1, 1);

event_register( )中对事件的处理要用到事件池自身的操作函数：
ret = event_pool->ops->event_register (event_pool, fd, handler, data, poll_in, poll_out);

在这里 poll_in : 1, poll_out : 1, 所以在 fd 上监听的是可读可写的状态；其处理句柄(fd 上事件触发的处理函数指针)为 socket_event_handler，即"client"层的事件转交给下一层(即 socket 层)处理。

在我们的配置方案中共有三个"client"子卷，即在三个"client"上做 mirror 镜像，每个子卷需要建立 2 个连接通道，总共要建立 6 个 socket 连接，并想事件池中注册 6 个事件。对于 socket 连接，每个事件的处理句柄都是 socket_event_handler，事件的处理交给了底层的 socket 来完成。

(2) 对"mirror"的初始化

afr 解析器的初始化主要是对 afr_private_t 结构体中相关参数的设置，最后把这个结构体变量和解析器自身关联起来。

afr_private_t 结构体的实现( )

```
#define AFR_XATTR_PREFIX "trusted.afr"      // AFR 解析器扩展属性前缀

typedef struct _afr_private {
        gf_lock_t lock;                 /* to guard access to child_count, etc */
        unsigned int child_count;       /* total number of children     */

        unsigned int read_child_rr;     /* round-robin index of the read_child    */
        gf_lock_t read_child_lock;      /* lock to protect above */

        xlator_t **children;            // 其子节点的链表(其实是一个数组，
                                        // 存储的是各个子节点的地址)

        unsigned char *child_up;        //

        char **pending_key;     // 数组，存放各个子节点扩展属性的键名
                                // pending_key[0] == "afr.trusted.client1"
                                // pending_key[1] == "afr.trusted.client2"
                                // pending_key[2] == "afr.trusted.client3"

        // 以下三个是 data、metadata 和 entry 自动修复功能的开关
        gf_boolean_t data_self_heal;            /* on/off */
        gf_boolean_t metadata_self_heal;        /* on/off */
```

```
        gf_boolean_t entry_self_heal;          /* on/off */

        // 以下三个是 data、metadata 和 entry 日志功能的开关
        gf_boolean_t data_change_log;          /* on/off */
        gf_boolean_t metadata_change_log;      /* on/off */
        gf_boolean_t entry_change_log;         /* on/off */

        int read_child;                        /* read-subvolume */
                                               // 优先读取的子卷
        unsigned int favorite_child;           /* subvolume to be preferred in resolving
                                                  split-brain cases */
                                               // 出现 split-brain 情况时，参考的子卷

        // 以下三个是 data、metadata 和 entry 锁服务器的计数
        unsigned int data_lock_server_count;
        unsigned int metadata_lock_server_count;
        unsigned int entry_lock_server_count;

        unsigned int wait_count;          /* # of servers to wait for success  */
    } afr_private_t;
```

AFR 解析器的初始化函数 init( )的主要工作如下：

(a) 从解析器("client1[/2/3]")中读取参数，主要有"read-subvolume"、"favorite-child"、"data-self-heal"、"metadata-self-heal"、"entry-self-heal"、"data-change-log"、"metadata-change-log"、"entry-change-log"、"data-lock-server-count"、"metadata-lock-server-count"、"entry-lock-server-count"

默认设置：

```
    /* Default values */
    priv->data_self_heal       = 1;
    priv->metadata_self_heal   = 1;
    priv->entry_self_heal      = 1;

    /* Change log options （default）*/
    priv->data_change_log      = 1;
    priv->metadata_change_log  = 0;
    priv->entry_change_log     = 1;

    /* Locking options （default） */
    priv->data_lock_server_count = 1;
    priv->metadata_lock_server_count = 0;
    priv->entry_lock_server_count = 1;
```

(b) 计数子节点

```
trav = this->children;
while (trav) {
    if (!read_ret && !strcmp (read_subvol, trav->xlator->name)) {
        ……
        priv->read_child = child_count;    // 优先读的子卷
    }

    if (fav_ret == 0 && !strcmp (fav_child, trav->xlator->name)) {
        ……
        priv->favorite_child = child_count;      // split-brain 情况下的参考子卷
    }

    child_count++;      // 子卷计数
    trav = trav->next;  //游走在子卷链表中的指针
}
……
priv->child_count = child_count;
```

(c) 设置各个子卷(解析器)扩展属性的键名

```
i = 0;
while (i < child_count) {
    priv->children[i] = trav->xlator;    // xlator_list_t trav;
        asprintf (&priv->pending_key[i], "%s.%s", AFR_XATTR_PREFIX,
                                trav->xlator->name);  // 设置扩展属性
                                // "trusted.afr.client3"
                                // "trusted.afr.client2"
                                // "trusted.afr.client1"
    trav = trav->next;
    i++;
}
```

(3) 对"fuse"的初始化

fuse 挂载的过程主要需要五个阶段：(1)挂载；(2)会话管理；(3)信号处理；(4)关联 channel 和 session；建立 inode 节点。

相关的数据结构：fuse_args、fuse_chan、fuse_session、fuse_private

```
// fuse 参数结构体
struct fuse_args{
    int argc;        // 参数个数
    char **argv;     // 参数列表
```

```
    int allocated;   // is 'argv' allocated?
};

//fuse channel
struct fuse_chan{
    size_t      bufsize;        //
    void        *data;          //
    int         fd;             // fuse 模块文件描述符(由 fuse_mount( )函数返回)
    fuse_chan_ops       op;    // "receive"、"send"、"destroy"
    fuse_session        *se;   //基于此 channel 的会话管理
};

// fuse session
struct fuse_session{
    fuse_chan       *ch;        // fuse channel
    void            *data;
    volatile int    exited;
    fuse_session_ops    op;  //
};

// fuse 模块自身的结构体
struct fuse_private {
        int                     fd;
        struct fuse             *fuse;   // the fuse handle
        struct fuse_session     *se;     // fuse 模块和底层文件系统的会话管理
        struct fuse_chan        *ch;     // fuse 模块与底层文件系统的交互,
                                         //  主要通过/dev/fuse
        char                    *volfile;
        size_t                  volfile_size;
        char                    *mount_point;
        struct iobuf            *iobuf;
        pthread_t               fuse_thread;
        char                    fuse_thread_started;
        uint32_t                direct_io_mode;
        double                  entry_timeout;
        double                  attribute_timeout;
        pthread_cond_t          first_call_cond;      //
        pthread_mutex_t         first_call_mutex;     //
        char                    first_call;           //
        gf_boolean_t            strict_volfile_check;
};
typedef struct fuse_private fuse_private_t;
```

"fuse"模块的初始化过程大致如下：

(a)挂载

```
priv->ch = fuse_mount(priv->mount_point, &args);
```

这个函数是把 glusterfs 挂载到 mount_point 目录下，其中 args (struct fuse_args)是一些相关的参数。

函数原型：

```
struct fuse_chan *fuse_mount(const char *mountpoint, struct fuse_args *args);
```

这个函数会创建并运行子进程 fusemount，并返回 fuse 模块文件 fd 给 fuse_main( )(这个可以在调试过程中看到)。

(b)会话管理

```
priv->se = fuse_lowlevel_new(&args, &fuse_ops, sizeof(fuse_ops), this_xl);
```

这将会创建一个会话管理句柄(通过 fuse_session_new()来生成)。

函数原型：

```
struct fuse_session *fuse_lowwlevel_new(struct fuse_args *args,
                                        const struct fuse_lowlevel_ops *op,
                                        size_t op_size,
                                        void *userdata);
```

args -- 相关参数(struct fuse_args)
fuse_ops -- fuse 模块与与底层文件系统交互的操作列表(一般为一个结构体)。
op_size -- fuse_ops 的大小
userdata -- 用户数据(在 fuse_session_new()中被使用)

(c)信号处理

```
ret = fuse_set_signal_handlers(priv->se);
```

这个函数设置信号的处理函数，包括 HUP，INT，TERM（忽略掉管道信号）。

fuse_set_signal_handlers()的原型为(在 fuse_signals.c 中)：

```
int fuse_set_signal_handlers(struct fuse_session *se)
{
    if (set_one_signal_handler(SIGHUP, exit_handler) == -1 ||
        set_one_signal_handler(SIGINT, exit_handler) == -1 ||
        set_one_signal_handler(SIGTERM, exit_handler) == -1 ||
        set_one_signal_handler(SIGPIPE, SIG_IGN) == -1)
        return -1;
```

```
        fuse_instance = se;        // static struct fuse_session *se;
        return 0;
    }
```

(d)关联 channel 和 session

channel 仅仅是建立了 fuse 模块和底层文件系统的交互接口(或者说是通道)，而 session 表示基于此类通道的会话管理对象，所以必须把它们关联起来。实现语句：

```
fuse_session_add_chan(priv->se, priv->ch);
```

函数原型(在 fuse_session.c 中)：

```
void fuse_session_add_chan(struct fuse_session *se, struct fuse_chan *ch)
{
        assert (se->ch == NULL);
        assert (ch->se == NULL);
        se->ch = ch;
        ch->se = se;
}
```

(d)建立 inode 节点表

```
this_xl->itable = inode_table_new (0, this_xl);
```

这一步主要是为"fuse"解析器创建一个 inode 表。首先定义一个 inode_table_t 变量: inode_table_t *new = NULL; 然后初始化这个 inode 表：__inode_table_init_root (new)，即新建一个 inode 节点(inode_t *root = NULL;)，设置相关信息，并把这个节点作为 inode 表的头节点。相关代码段如下：

```
        inode_t *root = NULL;
        struct stat stbuf = {0, };
        root = __inode_create (table);          // creating a new inode for root
        stbuf.st_ino = 1;              // 这里"1"是最上层节点的意思么？
        stbuf.st_mode = S_IFDIR|0755;    // directory
        __inode_link (root, NULL, NULL, &stbuf);  // 设置该 inode 的相关信息
        table->root = root;
```

6.事件处理

```
event_dispatch(ctx->event_pool);
```

这个函数的作用是对事件池中已注册的事件进行处理，它实际上调用了事件池自身的事件处理函数：

```
ret = event_pool->ops->event_dispatch (event_pool);
```

在这里，event_dispatch 函数指针指向的是 event_dispatch_epoll( )函数，这个就是整个程序事件处理的部分。

程序正常运行时为 daemon 进程，运行时的事件处理需要在 event_dispatch_epoll( )函数中的 while(1)循环中进行，其大致流程为：

(a) 若事件池中没有已注册事件，则条件等待：

```
while (event_pool->used == 0)
        pthread_cond_wait (&event_pool->cond, &event_pool->mutex);
```

(b) 若事件池中有注册事件在等待处理，则首先调用 epoll_wait( )函数轮询事件池，检查待处理事件文件 fd 的状态(epoll_wait( )函数正常返回的是需要处理的事件数)。然后逐个处理事件：

```
... ...
ret = epoll_wait (event_pool->fd, event_pool->evcache, event_pool->evcache_size, -1);
... ...
size = ret;
for (i = 0; i < size; i++) {
        if (!events[i].events)        // 没有关注事件消息
                continue;
                ret = event_dispatch_epoll_handler (event_pool, events, i);
}
```

在这里，event_dispatch_epoll_handler( )函数主要作用是调用事件池中的事件的处理句柄：

```
handler = event_pool->reg[idx].handler;
data = event_pool->reg[idx].data;
... ...
if(handler)
        ret = handler (event_data->fd, event_data->idx, data,
                        (event[i].events & (EPOLLIN | EPOLLPRI)), // fd 可读|有紧急数据可读
                        (event[i].events & (EPOLLOUT)),       // fd 可写
                        (event[i].events & (EPOLLERR | EPOLLHUP)));      // fd 错误|挂起
```

在前面 glusterfs_graph_init( )中，三个"client"子卷一共向事件池注册了 6 个事件(即要建立 6 个 socket 连接；每个"client"建立 2 个)，且每个事件的处理句柄都是 socket_event_handler。所以 glusterfs 运行后，在没有接收任何操作之前，需要处理 6 个已注册的事件。

对于 socket_event_handler( )函数：

```
int socket_event_handler(int fd,              // 事件要关联的文件 fd
                         int idx,             // 事件在事件池中的地址(索引下标)
                         void *data,          // 如 transport_t 等
```

```
        int poll_in,        // fd 是否可读
        int poll_out,       // fd 是否可写
        int poll_err) ;     // fd 是否发生错误
```

其主要流程：

(a) 检查是否建立，若否则新建一个连接；

```
if (!priv->connected) {     // socket_private_t *priv;
    ret = socket_connect_finish (this); // return 0 if success
```

(b) socket事件轮询：

```
if (!ret && poll_out) {
    ret = socket_event_poll_out (this);     // return 0 if success
}
if (!ret && poll_in) {
    ret = socket_event_poll_in (this);  // return 0 if success
}
if (ret < 0 || poll_err) {
    socket_event_poll_err (this);
    transport_unref (this);
}
```

这里需要注意的是三个函数的顺序及其 ret 值，是否调用则需要看上一个函数返回的 ret 值。

(对 socket_event_poll_xxx 部分的源码还未深入阅读……)

Ling Yi <lyi@0x55.cn>@ 2009 年 9 月