GlusterFS 分析报告

康国庆

---Version 1.3.0

引言

GlusterFS 是一个高层次的分布式文件系统解决方案。通过增加一个逻辑层，对上层使用者掩盖了下面的实现，使用者不用了解也不需知道，文件的存储形式、分布。

内部实现是整合了许多存储块（server）通过 Infiniband RDMA 或者 Tcp/Ip 方式互联的一个并行的网络文件系统，这样的许多存储块可以通过许多廉价的 x86 主机，通过网络搭建起来。

其相对于传统 NAS 、SAN、Raid 的优点就是：

1.容量可以按比例的扩展，且性能却不会因此而降低。

2.廉价且使用简单，完全抽象在已有的文件系统之上。

3.扩展和容错设计的比较合理，复杂度较低。扩展使用 translator 方式，扩展调度使用 scheduling 接口，容错交给了本地的文件系统来处理。

4.适应性强，部署方便，对环境依赖低，使用，调试和维护便利。

支持主流的 linux 系统发行版，包括 fc，ubuntu，debian，suse 等，并已有若干成功应用。

1.整体逻辑结构分析

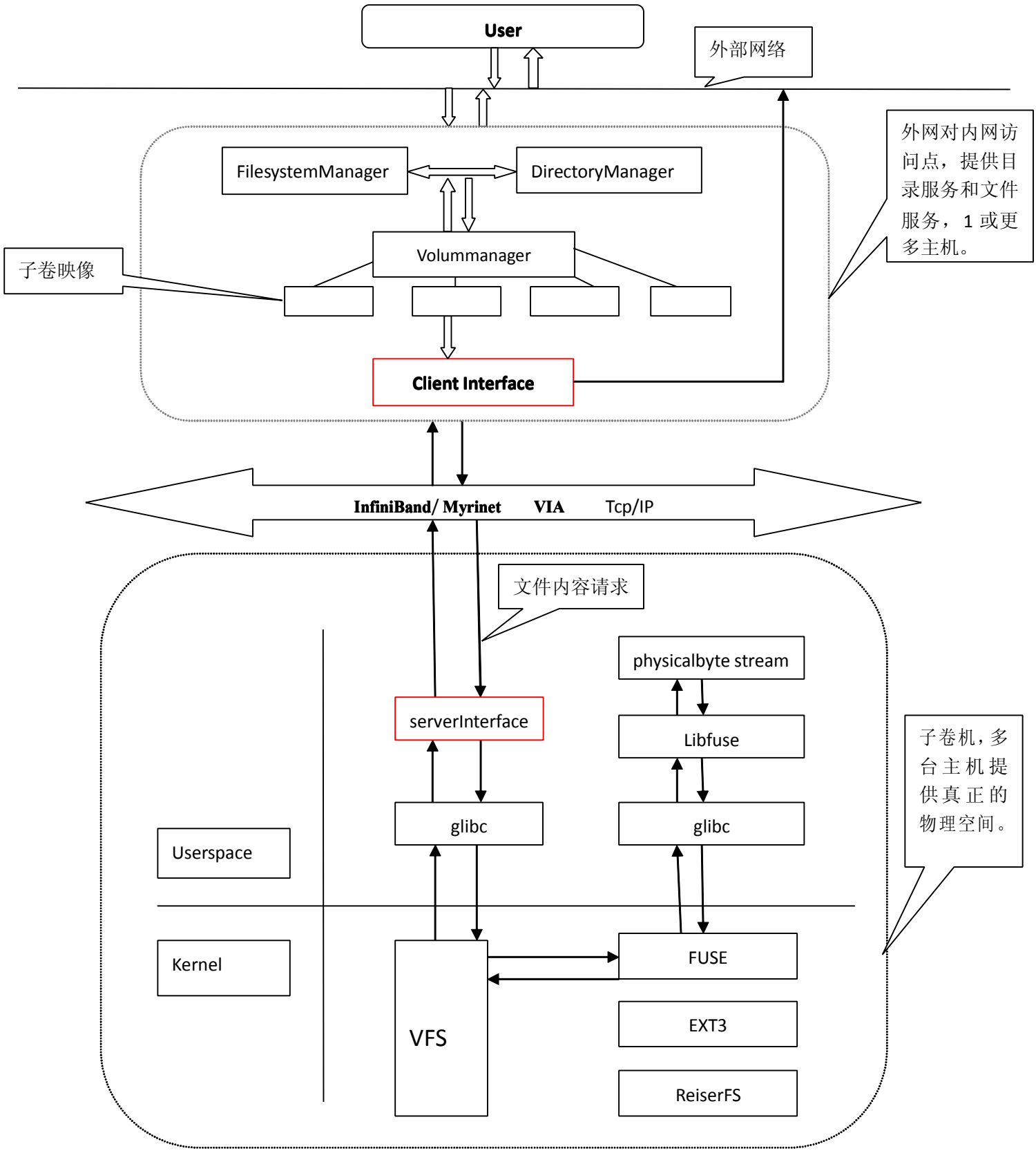
GlusterFS,整体来看分客户和服务端两部分，当然这是相对的。

客户端是对于提供数据中心整体来说的，它对外提供文件服务，目录服务，两个文件系统最重要的两个服务。（注 1：文件复制和共享的问题不知 GlusterFS 是怎么考虑的）。

客户机拥有一个卷管理器，和子卷的调度程序，在客户机中有的子卷映像和服务器主机是相对的，1 对 1。相当于一个卷集包含了若干逻辑卷，逻辑卷的物理位置是在服务器主机上的，该实现与 NFS 是有很大区别的。

服务器主机，上面拥有与客户机相应的通信接口，接口之间使用 GlusterFs protocol 来通信，服务器主机还应有自己的文件系统来提供文件服务和目录服务，GlusterFS 是构建在其上的。

当然客户和服务主机都有相应的配置文件，物理连接是通过 InfiniBand、Myrinet 或者 Gbit 以太网连接。下图为个人理解图：

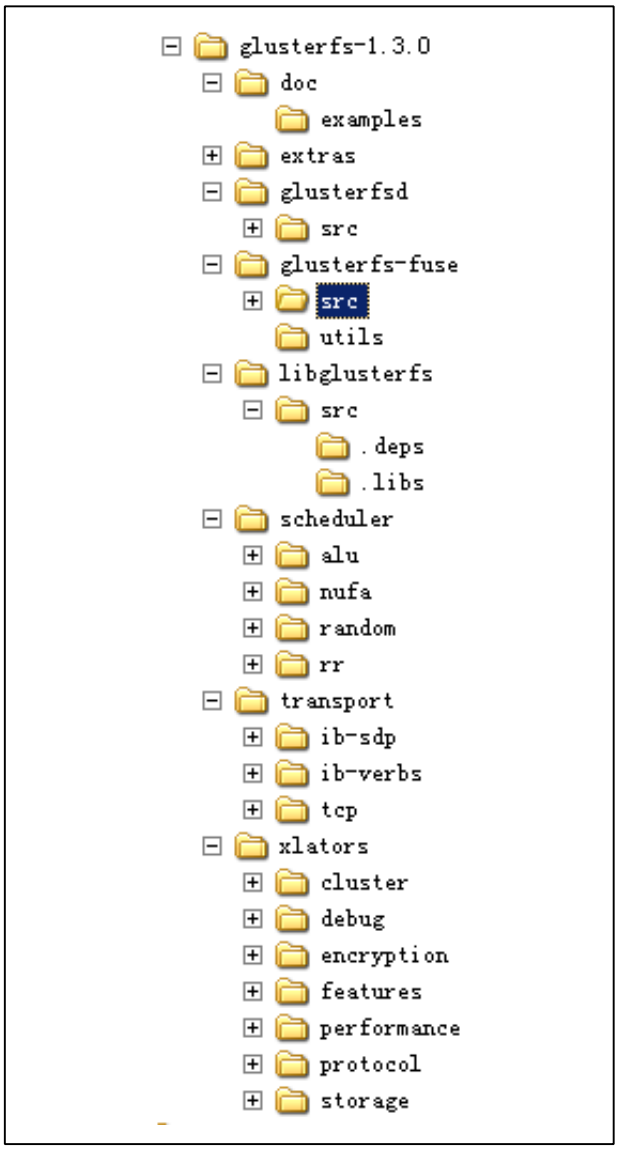

User
外部网络
外网对内网访问点，提供目录服务和文件服务，1 或更多主机。
FilesystemManager
DirectoryManager
Volummanager
子卷映像
Client Interface
InfiniBand/ Myrinet
VIA
Tcp/IP
文件内容请求
physicalbyte stream
serverInterface
Libfuse
glibc
glibc
Userspace
Kernel
VFS
FUSE
EXT3
ReiserFS
子卷机，多台主机提供真正的物理空间。

2.组件构成分析

GlusterFS，含有以下组件：

1.客户与服务器组件。这部分是复杂双方传输一个总的接口，服务器组件负责把自己的子卷发布出来，客户组件负责挂载 GlusterFS 到内核的 VFS 上。

2.翻译器模式， 一种来自 **GNU/Hurd** 的设计机制，（hurd 是借鉴 IBM vms 系统设计的，内核只有最简单的功能，其上层设计了模拟器可以模拟很多操作系统）该设计可以扩展



GlusterFS 的功能，包括调试器，性能调优的工具，加密验证等都是使用的该模式。 xlators 文件夹下的都是翻译器的实现。

3.传输模块，protocol translators 文件夹是其实现。

4.调度模块，Scheduler 文件夹下面有 4 种调度器实现，其作用是对子卷做负载均衡。

4 种调度器实现了 unify 翻译器。分别为

A）Adaptive Least Usage (ALU) 利用它的一种评价方式，把一些要素如磁盘利用率、文件访问频率（读、写分开）、响应速度等 综合起来考虑，做出的一种自适应的调度方式。其实现是 4 种调度中最复杂的。

B）Non-Uniform Filesystem Scheduler 这个有点象 NUMA ，一种结合 SMP 和 Mpp 以及 cluster 优点的内存管理方式，它的一个特点就是在创建文件时优先在本地创建文件。

C）Random scheduler 随机调度器，使用随机数发生器，选择子卷。

D）Round-Robin (RR) scheduler 螺旋线调度算法，它会将数据包均匀的分发给各台服务器，它把所有的服务器放在相等的地位上，而不会实际的去考虑各台服务器的差异，如负载，响应等等，如有 4 台服务器，调度序列可能就是 ABCDABCDABCDABCD。。。

4 种基本模块构成了 GlusterFS，另外还提供了一些扩展。

上图是源码解开后的结构。

3.源代码组成分析

定量分析：37 个头文件，55 个 c 源码文件，共有代码 59460 行（包括源码中的空行）。

文件最多的是 libglusterfs 文件夹，包含了 36 个文件。

3.1 启动过程

glusterfs_ctx_t （重要） GlusterFS 的环境类，里面包含日志文件，日志级别，定时器，poll 类型等等，使用 dict 实现。

服务端和客户端可以使用守护进程方式（deamon 精灵），也可以作为应用程序来启动，

下面剖析了两个部分的 main 函数。

server 部分（glusterfsd.c）初始化环境 ctx，初始化调用池？初始化链表，解析命令行参

数。

//main 部分

```
  FILE *fp;
  int32_t pidfd;
  glusterfs_ctx_t ctx = {                    //这个东西比较常见，环境提供者，使用字典
                                             或//map 实现。
    .logfile = DATADIR "/log/glusterfs/glusterfsd.log",
    .loglevel = GF_LOG_ERROR,            //日志级别，有点像 log4j
    .poll_type = SYS_POLL_TYPE_MAX,     //使用 poll 方式？奇怪为什么不使用 epoll
  };
  call_pool_t *pool;

  pool = ctx.pool = calloc (1, sizeof  (*pool));
  LOCK_INIT (&pool->lock);
  INIT_LIST_HEAD (&pool->all_frames);

  argp_parse (&argp, argc, argv, 0, 0, &ctx);
```

设定进程 pid，设定日志级别，文件路径。

```
  pidfd = pidfile_lock (pidfile);
  if (gf_log_init (ctx.logfile) < 0){
    return 1;
  }
  gf_log_set_loglevel (ctx.loglevel);//调用日志类来设定
```

接下来设定 系统资源限制，一般默认是 1024，比如打开文件数，此处设定为 65535.

软限制和硬限制设为相等。

```
  {
    struct rlimit lim;            //该结构体是系统内核提供 的，含有两个成员
    lim.rlim_cur = RLIM_INFINITY;
    lim.rlim_max = RLIM_INFINITY;

    if (setrlimit (RLIMIT_CORE, &lim) < 0) {
      gf_log ("glusterfsd",
              GF_LOG_ERROR,
              "main: failed to set RLIMIT_CORE, error string is %s",
              strerror (errno));
    }

    lim.rlim_cur = 65535; //RLIM_INFINITY;
    lim.rlim_max = 65535; //RLIM_INFINITY;

    if (setrlimit (RLIMIT_NOFILE, &lim) < 0) {
      gf_log ("glusterfsd",
              GF_LOG_ERROR,
              "main: failed to set RLIMIT_NOFILE, error string is %s",
              strerror (errno));
    }
  }
```

接下来读取卷配置文件 specfile 串，程序设定的是 CONFDIR "/glusterfs-server.vol"，

成功了如何如何.....失败了如何如何……

```
  if (specfile) {
    fp = fopen (specfile, "r");
    if (!fp) {
      gf_log ("glusterfsd",
```

```
          GF_LOG_ERROR,
          "FATAL: could not open specfile: '%s'",
          specfile);
      exit (1);
    }
  } else {
    gf_log ("glusterfsd",
        GF_LOG_DEBUG,
        "main: specfile not provided as command line arg");
    argp_help (&argp, stderr, ARGP_HELP_USAGE, argv[0]);
    exit (0);
  }
```

然后是判断环境 ctx 的 foreground 有值与否，没有值，就清空命令行的参数值，然后把 argv[0]填充成 `[glusterfsd]`，生成守候进程，更新前面锁定的 pidfile。

```
  if (!ctx.foreground) {
    int i;
    for (i=0;i<argc;i++)
      memset (argv[i], ' ', strlen (argv[i]));
    sprintf (argv[0], "[glusterfsd]");
    daemon (0, 0);        //生成守护进程
    pidfile_update (pidfd);
  }
```

初始化定时器，通过 specfile 串 对应文件构造出树来，关闭该文件，忽略管道信号 SIGPIPE

设置软中断处理函数为 `glusterfsd_cleanup_and_exit` 这个清理函数。

```
  gf_timer_registry_init (&ctx);

xlator_tree_node = get_xlator_graph (&ctx, fp);// 从文件中构造解释器树，
                                               //glusterfs.c 中有同名函数
  if (!xlator_tree_node) {//判断生成结果
    gf_log ("glusterfsd",
          GF_LOG_ERROR,
          "FATAL: could not create node graph");
    exit (1);
  }
  fclose (fp);

  /* Ignore SIGPIPE *///设置忽略的信号（sigpipe 管道）
  signal (SIGPIPE, SIG_IGN);

#if HAVE_BACKTRACE   //设置打印栈的信号处理
  /* Handle SIGABORT and SIGSEGV */
  signal (SIGSEGV, gf_print_trace);
  signal (SIGABRT, gf_print_trace);
#endif /* HAVE_BACKTRACE */

  signal (SIGTERM, glusterfsd_cleanup_and_exit);//设置软中断处理函数
```

最后进入循环，判断函数是 transport.c 中的 `poll_iteration` 函数。循环完毕关掉 pidfile。

```
  while (!poll_iteration (&ctx));

  close (pidfd);
```

观察 poll_iteration（&ctx）的实现，发现默认如果不设置 ctx 的异步进程通信模式的话，默认是使用 epoll 的，看代码：

//transport.c

```
int32_t
poll_iteration (glusterfs_ctx_t *ctx)
{
  int32_t ret = 0;

#ifdef HAVE_SYS_EPOLL_H
  switch (ctx->poll_type)
    {
    case SYS_POLL_TYPE_EPOLL:
      ret = sys_epoll_iteration (ctx);
      break;

    case SYS_POLL_TYPE_POLL:
      ret = sys_poll_iteration (ctx);
      break;

    default:
      ctx->poll_type = SYS_POLL_TYPE_EPOLL;
      ret = sys_epoll_iteration (ctx);
      if (ret == -1 && errno == ENOSYS) {
    ctx->poll_type = SYS_POLL_TYPE_POLL;
    ret = sys_poll_iteration (ctx);
      }
      break;
    }
```

```
#else
  ret = sys_poll_iteration (ctx);
#endif

  return ret;
}
```

关于 poll 和 epoll 相关函数的实现都在 poll.c 和 epoll.c 里面。poll、epoll、select 是网络编程和进程间通讯的三种模式。select 在 BSD Unix 中引入，而 poll 是 System V 的解决方案. epoll 调用添加在 2.5.45，作为使查询函数扩展到几千个文件描述符的方法。具体他们的区别可以查 man 手册页。

Client 客户端启动过程：（glusterfs-fuse/glusterfs.c）

初始化 glusterfs_ctx 环境中的日志文件位置，日志级别和 poll 类型，这个和 server 端一样，另外声明了 xlator 的图（树型），配置文件的指针，传输类型指针，还有就是系统资源限制，和调用池。

```
  xlator_t *graph = NULL;
  FILE *specfp = NULL;
  transport_t *mp = NULL;
  glusterfs_ctx_t ctx = {
    .logfile = DATADIR "/log/glusterfs/glusterfs.log",
    .loglevel = GF_LOG_ERROR,
    .poll_type = SYS_POLL_TYPE_MAX,
  };
  struct rlimit lim;
  call_pool_t *pool;
```

同样需要设定系统资源限制的值，此处还可以设定 debug 模式来使用 mtrace（）

```
#ifdef HAVE_MALLOC_STATS
```

```
#ifdef DEBUG
  mtrace ();
#endif
  signal (SIGUSR1, (sighandler_t)malloc_stats);
#endif
  lim.rlim_cur = RLIM_INFINITY;
  lim.rlim_max = RLIM_INFINITY;
  setrlimit (RLIMIT_CORE, &lim);
  setrlimit (RLIMIT_NOFILE, &lim);
```

初始化环境的调用池，解析参数等等这些和 server 是一样的。

```
pool = ctx.pool = calloc (1, sizeof (call_pool_t));
  LOCK_INIT (&pool->lock);
  INIT_LIST_HEAD (&pool->all_frames);
argp_parse (&argp, argc, argv, 0, 0, &ctx);
```

此处与 server 不同，但无非是一些判断和检测，如测试 ctx 的日志文件设置没，设置 glusertfs 的全局日志级别为环境的日志级别。

```
if (gf_log_init (ctx.logfile) == -1) {
    fprintf (stderr,
         "glusterfs: failed to open logfile \"%s\"\n",
         ctx.logfile);
    return -1;
  }
  gf_log_set_loglevel (ctx.loglevel);
```

针对解析的参数设定 mount_point，此处为判断设定与否。下面是设定端口号，和配置文件的地方等等，都是做检测用的。

```
  if (!mount_point) {
    fprintf (stderr, "glusterfs: MOUNT-POINT not specified\n");
    return -1;
  }
  if (!spec.where) {
    fprintf (stderr, "glusterfs: missing option --server=SERVER or --spec-file=VOLUME-SPEC-FILE\n");
    return -1;
  }

  if (spec.spec.server.port) {
    if (spec.where != SPEC_REMOTE_FILE)
      {
        fprintf (stderr, "glusterfs: -p|--port requires -s|--server option to be specified\n");
    exit (EXIT_FAILURE);
      }
  }
```

下面是通过配置文件 来设定 ctx，接下来就是一些中断屏蔽，中断处理函数的处理等等，这个和 server 是一样的。

```
specfp = get_spec_fp (&ctx);
  if (!specfp) {
    fprintf (stderr,
         "glusterfs: could not open specfile\n");
    return -1;
```

```
  }
  /* Ignore SIGPIPE */
  signal (SIGPIPE, SIG_IGN);
#if HAVE_BACKTRACE
  /* Handle SIGABORT and SIGSEGV */
  signal (SIGSEGV, gf_print_trace);
  signal (SIGABRT, gf_print_trace);
#endif /* HAVE_BACKTRACE */
```

接下来是当一切成员都初始化完毕时，此刻开始进行 glusterfs 的挂载，下面和 server 一样生成守护进程。然后注册定时器，生成 xlator 树并赋值给环境 ctx 中的 graph 成员，初始化 FUSE 的图，进入循环。后面的过程和 server 类似，少许不同。

所有的初始化过程都和 ctx 环境有关，从携带变量，赋值等等，都是操作的 ctx。

```
if (!(mp = glusterfs_mount (&ctx, mount_point))) {
    gf_log ("glusterfs", GF_LOG_ERROR, "Unable to mount glusterfs");
    return 1;
  }

  if (!ctx.foreground) {
    /* funky ps output */
    int i;
    for (i=0;i<argc;i++)
      memset (argv[i], ' ', strlen (argv[i]));
    sprintf (argv[0], "[glusterfs]");
    daemon (0, 0);
  }
```

```
  gf_timer_registry_init (&ctx);

  graph = get_xlator_graph (&ctx, specfp);
  if (!graph) {
    gf_log ("glusterfs", GF_LOG_ERROR,
        "Unable to get xlator graph");
    transport_disconnect (mp);
    return -1;
  }
  fclose (specfp);
  ctx.graph = graph;
  mp->xl = fuse_graph (graph);
  //  fuse_thread (&thread, mp);
  while (!poll_iteration (&ctx));
  return 0;
}
```

3.2 相关代码分析

3．2．1 传输协议代码

传输协议代码中体现了 3 种可以使用的方式：

- ib-verbs:使用 Infiniband verbs 层 为 RDMA(Remote Direct Memory Access) 通信. 这是最快的接口（1-4 ms）.
- ib-sdp: 使用 Infiniband SDP (sockets direct protocol) 为 RDMA 通信（70-90ms）.
- tcp:使用 普通 TCP/IP 或 IPoIB 内部连接. 假如有 4 节点的集群,每台主机的 NIC 带宽 1G byte/s ，那么组合起来的带宽也有 4G byte/s。

下面是 tcp 部分的分析：

共 4 个文件 tcp.h 是接口定义文件，里面定义了几个 tcp 操作的函数，接收，关闭连接等等，还定义了等待队列的结构体 wait_queue 和一个 tcp 状态的结构体 tcp_private。Tcp.c 是头文件里面定义的函数的实现。

Tcp-client.c:里面有关于建立连接的方法，tcp 客户端确认提交的方法，初始化传输和结束等方法。Tcp-server.c 里面也有相应的方法。

3 种方法都是按照 transport.h 定义的接口来实现的。这里用到了面向对象的思想和状态模式，transport.c 里面的相关方法对其多种“子类做了”dispatch。

例如：

```
//transport.c
int32_t
transport_notify (transport_t *this, int32_t event)
{
  int32_t ev = GF_EVENT_CHILD_UP;

  if ((event & POLLIN) || (event & POLLPRI))
    ev = GF_EVENT_POLLIN;
  if ((event & POLLERR) || (event & POLLHUP))
    ev = GF_EVENT_POLLERR;
  return this->notify (this->xl, ev, this);
}
```

该模块通过 transport_op 这个结构体实现了“多态”，该结构体在 transport.h 有接口定义（都是函数指针），而在每个连接方式的实现里面都声明一个实例，并按自己的方式初始化。

```
//transport.h
```

```
struct peer_info_t {              //连接属性结构体含有套接字信息
  struct sockaddr_in sockaddr;
};

struct transport {                //传输 interface
  struct transport_ops *ops;      //操作功能结构体指针，相当于类函数
  void *private;                  //私有信息，在子类中初始化
  void *xl_private;               //
  pthread_mutex_t lock;           //线程锁
  int32_t refcount;               //引用计数

  xlator_t *xl;
  void *dnscache;                         //dns 缓冲 ？作用不太明了
  data_t *buf;                            //数据缓冲
  int32_t (*init) (transport_t *this,     //初始化函数
           dict_t *options,
           event_notify_fn_t notify);

  struct peer_info_t peerinfo;            //包含了前面定义的套接字信息
  void (*fini) (transport_t *this);           //“析构”函数

  event_notify_fn_t notify;           //消息
};

struct transport_ops {                    //传输类的“成员函数“
  int32_t (*flush) (transport_t *this);

  int32_t (*recieve) (transport_t *this, char *buf, int32_t len);
  int32_t (*submit) (transport_t *this, char *buf, int32_t len);
  int32_t (*writev) (transport_t *this,
```

```
                const struct iovec *vector,
                int32_t count);
  int32_t (*readv) (transport_t *this,
                const struct iovec *vector,
                int32_t count);
  int32_t (*connect) (transport_t *this);
  int32_t (*disconnect) (transport_t *this);
  int32_t (*except) (transport_t *this);
  int32_t (*bail) (transport_t *this);
};
```

在真正的实现里面会初始化，传输类的成员函数的，如 Tcp-server.c

```
//Tcp-server.c

struct transport_ops transport_ops = { //此处初始化 transport 类的成员函数，这里是多态
  //  .flush = tcp_flush,              //每个其具体实现都可以灵活定义自己的操作细节。
  .recieve = tcp_recieve,
  .disconnect = tcp_disconnect,        //其具体实现也在本文件中。

  .submit = tcp_server_submit,
  .except = tcp_except,

  .readv = tcp_readv,
  .writev = tcp_server_writev
};
```

3．2．2 调度器剖析

调度器是给 unify 的接口用的，在集群文件系统中，分数据块时绑定一个相应的调度器，来进行数据的分布式存取。有四种调度器，前面有提到了。

调度操作接口比较简单，只要实现下列的接口函数的类就是一个调度器。

```
//scheduler.h
struct sched_ops { // 包括 初始化，清理费料，更新，调度器逻辑，通知。
  int32_t (*init) (xlator_t *this);
  void (*fini) (xlator_t *this);
  void (*update) (xlator_t *this);
  xlator_t *(*schedule) (xlator_t *this, int32_t size);
  void (*notify) (xlator_t *xl, int32_t event, void *data);
};
```

该文件还定义了一个获得调度器的方法 get_scheduler，具体实现在 scheduler.c 里面，被 unify.c 中调用（后面提 unify.c）.

RR 调度器是最简单的，和下面要写的 xlator 关系很密切。

结构体如下：

```
//rr.h
struct rr_sched_struct { //rr 调度需要的一些属性如刷新间隔，剩余磁盘空间，可用性等等。。
  xlator_t *xl;
  struct timeval last_stat_fetch;
  int64_t free_disk;
  int32_t refresh_interval;
  unsigned char eligible;
};
```

```
struct rr_struct {   //rr 结构体
  struct rr_sched_struct *array;  //包含 rr 调度结构体的一个数组
  struct timeval last_stat_fetch;
  int32_t refresh_interval;     //刷新时间间隔
  int64_t min_free_disk;      //最小的剩余空间

  pthread_mutex_t rr_mutex;  //线程锁
  int32_t child_count;            //节点计数
  int32_t sched_index;        //调度索引
};
```

与接口的联系方式和 transport 模块类似，实现了 sched_ops 定义的操作。

```
//rr.c
struct sched_ops sched = {
  .init     = rr_init,
  .fini     = rr_fini,
  .update   = rr_update,
  .schedule = rr_schedule,
  .notify   = rr_notify
};
```

3．2．3　xlator 与 xlator _list

Xlator 是一个有前驱和后继及父指针的节点类，其组成了 xlator_list 链表，另外系统使用它组成树结构来使用，schedulor 在初始化时是会遍历 xlator_list 的每个节点的。Xlator 里面还包括了相关 xlator 的操作符 xlator_fops、 xlator_mops，和构造函数与析构函数，以及一些必要的数据，节点表指针，消息，glusterfs 环境，配置选项字典（option）等

等。

```
//xlator.h
struct _xlator {
  char *name;
  char *type;
  xlator_t *next, *prev;
  xlator_t *parent;
  xlator_list_t *children;

  struct xlator_fops *fops;
  struct xlator_mops *mops;

  void (*fini) (xlator_t *this);
  int32_t (*init) (xlator_t *this);
  event_notify_fn_t notify;

  dict_t *options;
  glusterfs_ctx_t *ctx;
  inode_table_t *itable;
  char ready;
  void *private;
};
typedef struct xlator_list {
  xlator_t *xlator;
  struct xlator_list *next;
} xlator_list_t;
```

Xlator_list 是一个关于 xlator 的前向链表。

3．2．4 Translators 与 hurd/GNU

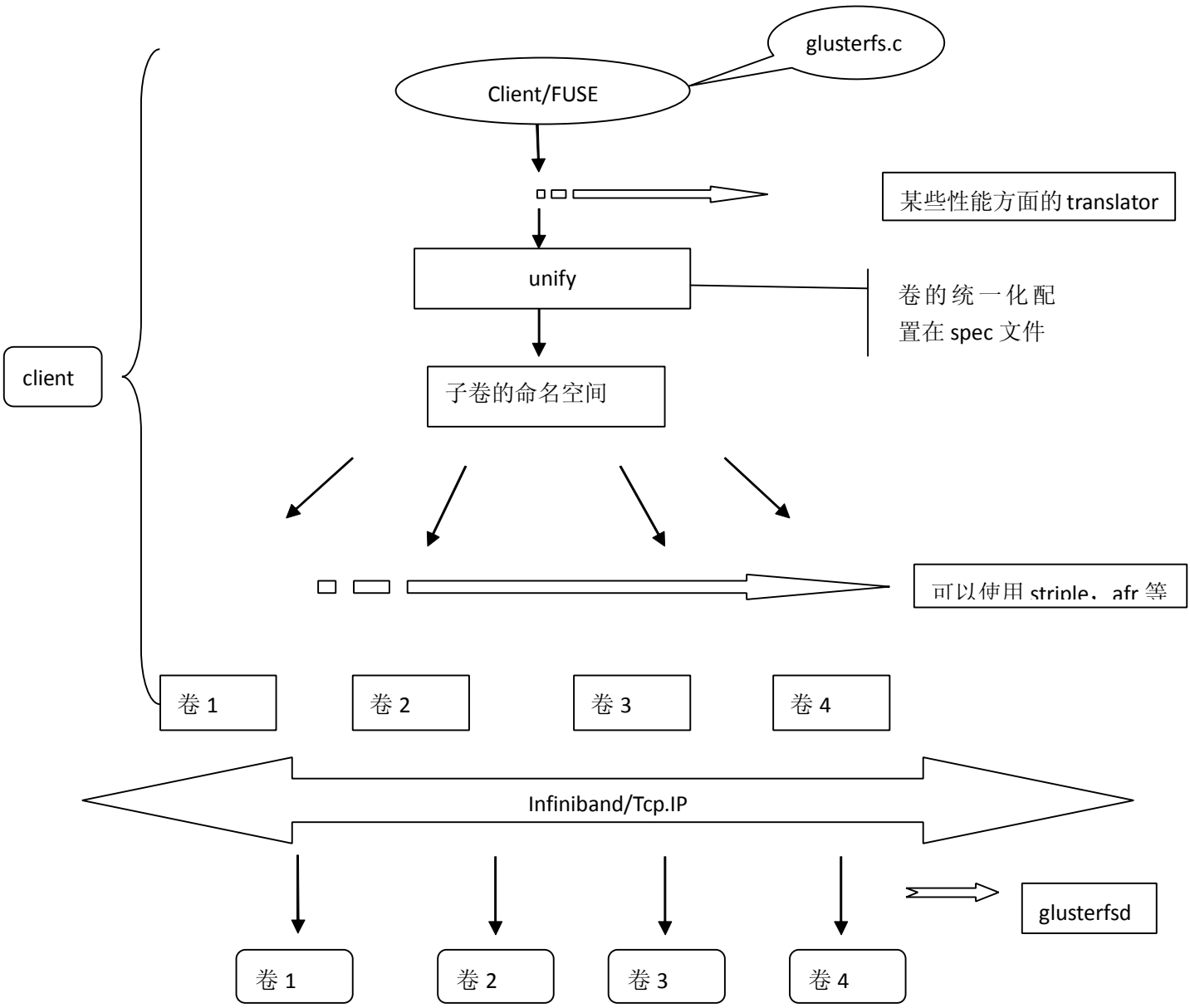
Translators 思路来自 GNU/hurd 微内核的虚拟文件系统设计部分，一个 translator 是一个用于目标服务器与 hurd 分布式文件系统之间的普通程序。作用是对外部的文件系统做操作时，转换成目标文件系统适当的调用。Translators 不需要特别的权限运行。

在传统的 unix 内核设计上，文件系统或虚拟文件系统都是在内核里面实现的，内核有绝对的机器访问权，在 hurd 上，使用的微内核设计，所以 translator 使用的协议相关操作不需要特别权限来执行，在用户空间上执行。

Translator 可以附着在节点上，每个 tanslator 都是针对相应的功能设计的，比如 rot-13 是加密用的，Trace 是追踪，调试用的，performance 文件夹下的 4 个 translator 是用来性能调优用的。

和 translator 有直接关系的是这么几个文件，xlator.c 、xlator.h 、default.h、,default.c 四个文件，后两个比较容易解释，就是其他 translator 在定义 fop 和 mop 时如果自己没有设定实现的话，便会使用默认实现，其实现就在这两个文件里面。也就是说这两个文件里面保含了所有的操作的定义。前两个文件在上节已经说过了。

下图是 glusterfs 使用的相关 translator 所处的位置。相对于客户端来说，服务端的任务真的是很简单的，大部分 translator 是工作在客户端的，比如，性能调优，调度器，合并器等等，而相反，服务端只需运行起来 glusterfsd 就可以了。



要设计一个 translator 也比较容易， 除了需要一个初始化函数

```
int32_t  init (xlator_t *this)
```

和收尾的函数

```
void    fini (xlator_t *this)
```

还需要对 xlator.h 中的 xlator_fops、 xlator_mops 两个结构体中，需要翻译的命令自己定义

```
//xlator_fops的定义，里面包括要求翻译的调用表，fops表示文件操作
struct xlator_fops {
```

```
  fop_lookup_t          lookup; //前者是一个指向函数的指针
  fop_forget_t          forget;
  fop_stat_t            stat;
        .............
  fop_lk_cbk_t              lk_cbk;
  fop_writedir_cbk_t        writedir_cbk;
};
```

结构体中每个属性都是个指向函数的指针，例：

```
typedef int32_t (*fop_lookup_t) (call_frame_t *frame,
                 xlator_t *this,
                 loc_t *loc);
```

xlator_mops的定义，里面包括要求翻译的调用表，mops表示管理操作

```
struct xlator_mops {
  mop_stats_t             stats;
  mop_fsck_t              fsck;
  mop_lock_t              lock;
  mop_unlock_t            unlock;
  mop_listlocks_t         listlocks;
  mop_getspec_t           getspec;

  mop_stats_cbk_t             stats_cbk;
  mop_fsck_cbk_t              fsck_cbk;
  mop_lock_cbk_t              lock_cbk;
  mop_unlock_cbk_t            unlock_cbk;
  mop_listlocks_cbk_t         listlocks_cbk;
  mop_getspec_cbk_t           getspec_cbk;
};
```

前面是需翻译的命令，翻译后的命令例子如下，以 rot-13 为例：

//rot-13.c 下面的赋值表示要翻译的两个调用，管理操作以默认不做更改

//（default.h,default.c）

```
struct xlator_fops fops = {
  .readv        = rot13_readv,
  .writev       = rot13_writev
}; //这里赋值结构体的方式是C99标准新出来 的，其在linux2.6内核源码中有较多的使用。


struct xlator_mops mops = {
};
```

同样你把自己需要翻译过来的调用自己实现 例如下：

```
static int32_t
rot13_writev (call_frame_t *frame,
              xlator_t *this,
              fd_t *fd,
              struct iovec *vector,
              int32_t count,
              off_t offset){  do something  }
```

这样扩展一个 translator 就完成了。

translator 操作是异步的，这样可以减少网络上调用的延时造成性能下降。

它使用 STACK_WIND 和'STACK_UNWIND 维护一个用户空间的调用栈。在桩文件 call-stub.h 文件中，里面有 call_stub_t 结构体的定义，结构体里面含有一个联合，另外头文件还有相关调用的桩，call-stub.c 里面是头文件接口的实现。

// call-stub.h

```
typedef struct {
  struct list_head list;
  char wind;
  call_frame_t *frame;
```

```
  glusterfs_fop_t fop;

  union {    //联合体里面包含了若干个结构体，其中每个结构体里面都是一个调用桩
             //是一个指向函数的指针，和相关需要传递或保存的属性或结构（像函数对象？）。
    /* lookup */
    struct {
      fop_lookup_t fn;
      loc_t loc;
    } lookup;
    ..........

  } args;
} call_stub_t;
```

// call-stub.c 头文件中示例的实现

```
call_stub_t *
fop_lookup_stub (call_frame_t *frame,
         fop_lookup_t fn,
         loc_t *loc)
{
  call_stub_t *stub = NULL;

  stub = stub_new (frame, 1, GF_FOP_LOOKUP);
  if (!stub)
    return NULL;

  stub->args.lookup.fn = fn;
  loc_copy (&stub->args.lookup.loc, loc);
```

```
  return stub;
}
```

3．2．5 server-protocol/client-protocol

Glusterfs 使用的协议是比较简单的，协议的定义可以在其官方网站有简短的描述。（来自代码注释 protocol.h）

```
  All value in bytes. '\n' is field seperator.
  Field:<field_length>
   ==================
  "Block Start\n":12
  callid:16
  Type:8
  Op:8
  Name:32
  BlockSize:32
  Block:<BlockSize>
  "Block End\n":10
  ==================
```

起始头 12 个字节，调用 id 16 个字节，操作类型 8 个字节，操作指令 8 个字节，操作名 32 个字节，数据块大小 32 个字节，然后是数据块，然后是包尾 10 个字节。

操作类型有四种：分别是请求和回应、对应当 fop（文件操作），mop（管理操作）

```
typedef enum {
  GF_OP_TYPE_FOP_REQUEST,
  GF_OP_TYPE_MOP_REQUEST,
  GF_OP_TYPE_FOP_REPLY,
  GF_OP_TYPE_MOP_REPLY
} glusterfs_op_type_t;
```

操作指令定义在 glusterfs.h 里面两个枚举类型，一个 fop，一个 mop。

服务端执行的是响应请求，所以收到的包中操作类型皆是 request 类型的 fop 或者 mop，然后将之交给一个解释器函数，函数负责分析是 fop 还是 mop，然后转换成 local 系统的执行序列。这其中还包括一些传输的错误处理，参数不正确等等。

该解释器通过调用相应的本地函数，处理完后返回的也是一个完整的协议数据包。

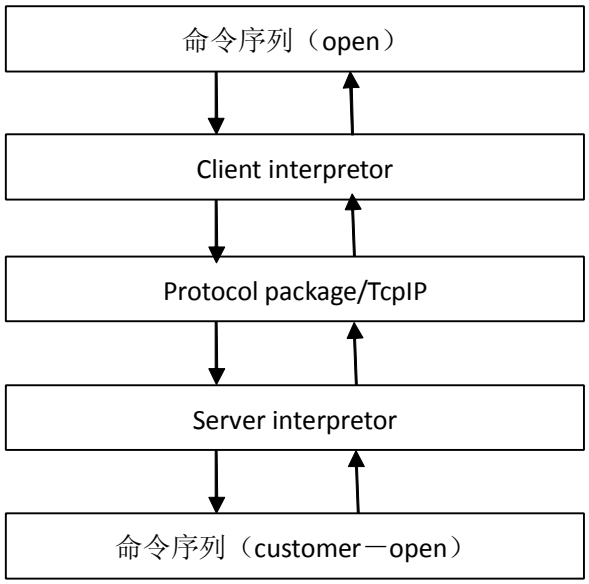
该函数在 server-protocol.c 文件中，声明如下：

```
static int32_t
server_protocol_interpret (transport_t *trans,
                gf_block_t *blk)
```

此外服务端还需要维护一个响应队列。

客户端就比较繁琐了，它需要负责连接，保持连接，握手等动作，另外它还和服务端一样，也有一个翻译器，负责解释收到的协议包的处理。

由于客户端对其上层应用来说还得提供文件服务，所以它所需要提供的 xlator_fops 对象的成员函数比服务端提供的更多一些。



该图是服务端与客户端建立连接之后，客户端请求一个命令，或传输一部分数据所需的过程示意图，最中间的部分是在网络中传输的协议包。顶部是客户端请求的指令，该指令一般是给更高层提供服务的。最下层是服务端处理请求的命令或数据给 local 调用处理。

3.3 相关的实用工具类

Util 类：dict 、stack、list，自己实现的容器和必要的操作。Stack 用的是 list 实现的。

hashFn： 实现的一个最快速 hash 算法，uint32_t SuperFastHash (const char * data, int32_t len)这个方法，来自网上一个实现。

Lock：加锁，解锁的类。

Logging：日志类，到文件的。

rot-13：简单的一个加密类，没有错误检查的。

Timer：定时器，记时器

3.4 性能，优化的部分代码

整体上，增加性能的优化方式就是利用缓冲，加之考虑其业务需求，比如频繁读写小文件，或是大量操作是读文件而很少写等等。利用其业务特点，适当的使用优化方式，源代码里面提供了 4 种优化器，都是利用了 translator 模式实现的。

Readhead：预读技术，这个在操作系统中内存技术，外存技术和 catch 技术中用到比较多了。大概的意思就是使用临近数据被访问的可能性较大的原理，做的预读的优化。

Writebehind：后写技术，就是当需要回写硬盘时，先做一个缓冲区，然后等缓冲区满了，一次性写回硬盘，这样减少了使用系统调用，网络等开销。

Io-cache：这个是利用多个服务端中多余内存来做缓冲用的，网站上有个性能测试，64 个服务端主机，每主机有 8GB mem，使用 io-catche 后，每个主机使用了 6GB mem 作

为 io 缓冲，共有 64×6GB =384GB 的数据缓冲区，这样可以大大减少外存的访问，提高了数据访问速度。

Io-threads：io 线程化，AIO 添加了异步读写功能，使用这个 translator，可以利用系统的 idle 进程堵塞时间来处理新到来的请求。当进入内核调用时系统会锁住资源，如 cpu mem 等等，这样就不能利用其做其他工作了，该 translator 可以更好的改进此模型，是之更有效率。看其 road map 是 1.3 版本新加进来的特性。

在 cluster 文件夹下面有两个与性能有关的三个 translator。

分片技术 strip：这个大意就是象 RAID0 那样，可以加速保存和读取，但风险加大，所以在 Stripe.c 文件的注释中，提示最好和下面的 translator 一起使用，以保证安全。

重复技术 afr：这个就是象 RAID1 那样，保存时写数据做双份。而且可以对相应的类型的文件做不同的设置。

Unify：组合了多个存储块到一个巨大的存储空间里面，前面介绍 translator 时有写 unify 在整个系统中所处的位置。

当然对单个主机中文件系统的优化也是需要的，比如对 ext3，reiserfs 的参数优化。

4.全局看整个系统

此处写的是个人体验，不一定都正确,供参考。

Glusterfs 是一个存储空间和访问效率都可以线性增加的一个分布式文件系统，网上资料除了 gluster.org 以外，几乎没有什么有关的介绍了。

通过对源码的审阅，个人感觉，比较主要的是把程序的整体结构理清，扩展方式弄明白在向下看具体的实现是比较好的。

该系统扩充的方式是使用了 translator 的模式，具体我还参考了《现代操作系统》中分布式文件系统章节和GNU/HURD 中解释 translator的部分，后者主要是在gnu的网站上。

数据结构上讲，整个文件系统中节点构成了一棵树，而且每个节点的操作是通过某个 translator 来工作的，一个节点可以附着很多的 translator。所有的 translator 都要实现 xlator 结构体和相关的 xlator_fops、xlator_mops 两个“成员函数的结构体”，从 xlator“继承”下来的操作如果不自己定义，那么就会使用默认的设置，这个在 default.c 里面定义。当然自己定义的操作并赋值，这个过程有些象子类覆盖父类的操作，平行来看也就是多态。当然这是从面向对象角度来看的，该系统很多地方都使用了面向对象的思想来设计的，这个和 linux 2.6 以后的内核模块设计是异曲同工的。

那么一般可以这样识别一个用 c 实现的 Class 关键字的类：

例（对源文件有些修改）：

```
struct A {
  char *name;
  char *type;                       //成员

  struct A  *this;                  //this 指针
  struct xlator_fops *fops;         //成员操作结构体 1
  struct xlator_mops *mops;         //成员操作结构体 2
  void (*fini) (struct A *this);    // 析构函数，垃圾清理
```

```
  int32_t (*init) (struct A *this);//构造函数，初始化
  event_notify_fn_t notify;        //成员。。

  dict_t *options;
  glusterfs_ctx_t *ctx;
  inode_table_t *itable;
  char ready;
  void *private;
};
```

1. 一个 struct 定义里面包含一个指针 该指针的类型是该 struct 定义的类型。

2. 上面的 struct 内部成员中含有其他结构体的指针，象 xlator_fops 就是这里提到的其他结构体的指针，该结构体里面全部都是指向函数的指针，也就是成员函数 了。

当然此处也可以把 xlator_fops 里面的成员都释放到 struct A 里面， 但是这样这个 struct 就显得有些臃肿了，，毕竟成员函数还是不少的。上面这个例子还有两个只有类才具备的析构函数，和构造函数。

glusterfs_ctx 控制了全局的信息，很多地方传输都是使用它来传递的，一个典型的环境类。初始化些东西也是针对它来做的。

Redhat GFS 和 Glusterfs 的目的类似，都是以全局在一个命名空间下面通过访问其他节点获取数据的。此处没有性能比较。

Lustre 也是一个开源基于 GNU lisence 的集群文件系统，网站资源比较丰富，开发者的资源也比较多，中文资料也不少，sun 公司收购了 clusterfs 公司，拥有了此技术。

下面地址显示的是 lustre 与 glusterfs 做相当命令所需时间的比较：

http://www.gluster.org/docs/index.php/GlusterFS_1.3.pre2-VERGACION_vs_Lustre-1.4.9.1_Various_Benchmarks

下面的地址是 NFS 与 glusterfs 性能的测试对比：

http://www.gluster.org/docs/index.php/GlusterFS_1.2.1-BENKI_Aggregated_I/O_vs_NFSv4_Benchmark

5. 附言

下载的源码为 1.3.0 版，使用 cygwin configure 失败，对 FUSE 配置也失败。
Window 下再没尝试。

另外时间较短，自己 c、api 方面经验不足，有些地方还不是很确定，比如：translator 到底能不能到处附着在节点上。后来看到性能测试时，client 和 server 还可以多对多...这个也是以前没想到的。
很多前面推测出的结论，后两天又由于深入的学习而推倒。这几天我收获比较大，人的认识过程就是这样，肯定到否定再到肯定。

2007-11-25 西安