0.1版

**《Gluster分布式存储协助文档》**



深圳市瑞驰信息技术有限公司内部资料

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**前言：**

互联网有三大行业领域：计算、存储、应用。其中分布式存储是一个前景广阔的朝阳产业。

Glusterfs、Ceph是瑞驰公司目前使用的主流存储软件，两个不同框架的开源软件都是属于四大分布式存储，性能方面各有优缺点，业务领域重合度不激烈。

对于gluster的开发，遵循循序渐进的逻辑，分为两个分支来讲解：源码框架学习、功能模块开发。

Gluster3.2版本是一个被寄予厚望的经典版本，它的定位与linux2.6的定位是一致的：即代码框架已经相对稳定，功能相对完善，具备高度可扩展性。

公司使用的新版本是gluster3.8.15版本。新版本与3.2版本差别在与新的模块增多，本质上代码框架相似度极高，但是代码量比较多。用3.8.15去研究代码时会浪费比较多的时间去看很多冗余的代码，而3.2几乎冗余的代码比较少。 固本求源，去伪存真，当在学会了gluster3.2，再去理解新版本的代码自然没有问题。

由于本人能力有限，时间仓促，文档中有错误之处敬请原谅。欢迎读到本文档错误的工程师批评指正，教学相长、共同进步。

交流邮箱：[1650727278@qq.com](mailto:1650727278@qq.com)

jacky-------------2018.3.7

目录：--------------------------------------------------------------------------------------------------------------------

第一章-----------------------------gluster快速入门指南

1. ----------------------------安装指南
2. ----------------------------管理指南

由于时间关系，本文档中不再重新添加整理，前三章参考我制作的另外一个文档：《gluster管理手册》

1. ----------------------------gluster源码结构分析
2. ----------------------------字典、avl树、链表、哈希表数据结构
3. ----------------------------分布式存数文件系统模型
4. ----------------------------cli模型
5. ----------------------------rpc模型
6. ----------------------------内存池模型
7. ----------------------------加密算法
8. ----------------------------鉴权模块
9. -------------------------五种调度器
10. -------------------------转换器模块
11. -------------------------制作rpm发行包

---------------------------------------------------------------------------------------------------------------------------------

**第四章**

--------------glusterfs3.2源码目录结构解析

.

|-- argp-standalone

命令行参数解析器（Argument Parser）

|-- cli

gluster命令行（Command Line Interface）

|-- contrib

| |-- fuse-include

| |-- fuse-lib

| |-- fuse-util

用户空间文件系统（Filesystem in Userspace，简称FUSE）

| |-- ipaddr-py

ip地址验证工具

| |-- libgen

包含经过改写的dirname/basename命令，用于适配NetBSD系统

| |-- macfuse

fuse for Mac

| |-- rbtree

红黑树-数据结构（red-black tree）

| `-- uuid

通用唯一识别码（Universally Unique Identifier，简称UUID）

|-- doc

man文件

user-guide.pdf

卷配置文件.vol范例

|-- extras

性能测量：

测试性能的c工具、测试gluster安装完整性的脚本工具

OS相关的一些配置文件（如启动gluster的配置文件）

|-- glusterfsd

glusterfs daemon（命令解析器、任务构建）

|-- libglusterfs

glusterfs库-核心代码

|-- rpc

| |-- rpc-lib rpc接口，定义了大概35个服务程序（鉴权、握手、fops等）、服务过程

| |-- rpc-transport rpc传输层模块：socket、 rdma，二选一由卷配置文件确定

| `-- xdr 实现rpc消息的xdr编码

远程过程调用机制

|-- swift

对象存储相关的

`-- xlators

|-- bindings

| `-- python c语言调用python实现的一些高级文本处理功能的接口函数。

使得可以使用python写xlator

|-- cluster

| |-- afr

Automatic File Replication

副本机制

| |-- dht

Distributed Hash Table

使得很多分布式卷看起来是一个卷

也可以反过来说，将一个卷中的文件分散到很多分布式卷中

| |-- stripe

将一个文件分割成N份，存在N个地方

|-- debug

| |-- error-gen

生成错误的xlator，方便调试

| |-- io-stats

IO负载的相关信息

| `-- trace

打印各种调用信息

|-- encryption

| `-- rot-13 rot-13加密是对26个字母，每个字母与一半位置（13个）互为映射。

极简单的加密，就是将每个字符加13，仅仅是玩具。

|-- features

| |-- filter

| |-- index

| |-- locks

| |-- mac-compat

| |-- marker

| |-- path-convertor

| |-- quiesce

| |-- quota

设置磁盘/文件夹配额

| |-- read-only

让文件系统只读

| `-- trash

提供类似windows垃圾箱的功能

|-- lib

| `-- src

|-- meta

| `-- src

|-- mgmt

| `-- glusterd

|-- mount

| `-- fuse

|-- nfs

| `-- server

|-- performance 提高性能的一个模块

| |-- io-cache

| |-- io-threads

| |-- md-cache

| |-- quick-read

| |-- read-ahead

| |-- symlink-cache

| `-- write-behind

|-- protocol

| |-- auth

| |-- client

使得client可以访问远端的xlators

| `-- server

使得server可以让远端访问本地的xlators

|-- storage

| `-- posix

使用标准POSIX文件系统存储文件

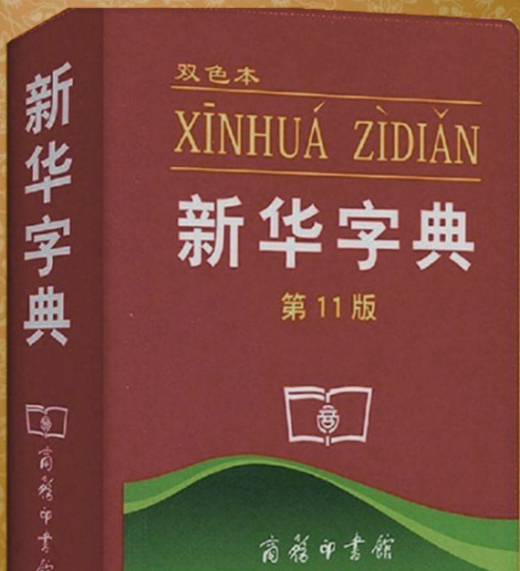
`-- system

`-- posix-acl

**第五章**

-------字典、红黑树、链表、哈希表数据结构

字典：

 **？**

艺术来源于生活，而又高于生活，技术也是一样的道理。现实生活中的很多事情，都可以抽象后形成理论，进而以理论为基础产生模型，形成技术。

观察上面的字典，首先看到它：

有一个封皮；

封皮里面有词汇；

每个词汇的格式是key:value；

它有如下特点：每个词汇的key值是一个字符串、对应的value也是一个字符串。所有字符串的长度不定（需要按需分配，你不能把有正页只写一个词汇，虽然那样写并不会出现错误。按需分配的理论类似于malloc() 、realloc()）

下面介绍一种编程语言中的数据结构，字典。

数据结构在“dict.h”头文件中定义：

//下面三个数据类型是为了在定义一个结构体之前就可以先引用它的类，防止编译器语法错误（gluster源码当中大量使用了这种定义方式）。

typedef struct \_data data\_t;

typedef struct \_dict dict\_t;

typedef struct \_data\_pair data\_pair\_t;

//词汇value对象

struct \_data {

unsigned char is\_static:1; // 1：词汇value存储在 全局变量，0：词汇value存储在 堆中(malloc)

unsigned char is\_const:1; // 词汇value是否有（被）修改的属性

unsigned char is\_stdalloc:1; // 1：malloc分配 0，acct分配

int32\_t len; // 词汇value字符串长度

struct iovec \*vec; // 词汇向量

char \*data; // 词汇value字符串

int32\_t refcount; // 词汇value被引用count

gf\_lock\_t lock; // 词汇value锁

};

//词汇结构对象

struct \_data\_pair {

struct \_data\_pair \*hash\_next; // 词汇哈希单链

struct \_data\_pair \*prev; // 词汇双向链

struct \_data\_pair \*next;

data\_t \*value; // 词汇value

char \*key; // 词汇key字符串

};

//字典对象

struct \_dict {

unsigned char is\_static:1; // 1：词汇存储在 全局变量，0：词汇存储在 堆中(malloc)

int32\_t hash\_size; // 字典的哈希边界

int32\_t count; // 词汇量

int32\_t refcount; // 本字典被引用count

data\_pair\_t \*\*members; // 本字典中，词汇数组首地址

data\_pair\_t \*members\_list; // 本字典中，词汇链表头

char \*extra\_free;

char \*extra\_stdfree;

gf\_lock\_t lock; //字典锁（注意结构体中出现的任何锁，都说明该结构体是 资源抢占性对象类。结构体类中出现的对象，自然是保护结构体中出现的成员 变量，也就是锁的作用域是整个所在类）

};

有了数据结构，对应一套实现机制在”dict.c”文件当中：

/\*\*

\*函数功能： 创建一个新的数据（词汇value）对象，返回其指针

\*/

data\_t \*get\_new\_data()

{

data\_t \*data = NULL;

data =

(data\_t \*) GF\_CALLOC(1, sizeof(data\_t), gf\_common\_mt\_data\_t);

if (!data) {

return NULL;

}

LOCK\_INIT(&data->lock);

return data;

}

/\*\*

\*函数功能： 创建一个新的数据对（词汇）对象，返回其指针

\*/

data\_pair\_t \*get\_new\_data\_pair()

{

data\_pair\_t \*data\_pair\_ptr = NULL;

data\_pair\_ptr = (data\_pair\_t \*) GF\_CALLOC(1, sizeof(data\_pair\_t),

gf\_common\_mt\_data\_pair\_t);

return data\_pair\_ptr;

}

/\*\*

\*函数功能： 创建一个有size\_hint条词汇的字典对象，返回其指针

\*/

dict\_t \*get\_new\_dict\_full(int size\_hint)

{

dict\_t \*dict = GF\_CALLOC(1, sizeof(dict\_t), gf\_common\_mt\_dict\_t);

if (!dict) {

return NULL;

}

dict->hash\_size = size\_hint;

dict->members = GF\_CALLOC(size\_hint, sizeof(data\_pair\_t \*),

gf\_common\_mt\_data\_pair\_t);

if (!dict->members) {

GF\_FREE(dict);

return NULL;

}

LOCK\_INIT(&dict->lock);

return dict;

}

/\*\*

\*函数功能： 创建有1条词汇的字典对象，返回其指针

\*/

dict\_t \*get\_new\_dict(void)

{

return get\_new\_dict\_full(1);

}

/\*\*

\*函数功能： 创建有1条词汇的字典对象，字典引用为1，返回对象指针

\*/

dict\_t \*dict\_new(void)

{

dict\_t \*dict = NULL;

dict = get\_new\_dict\_full(1);

if (dict)

dict\_ref(dict);

return dict;

}

/\*\*

\*函数功能： 比较参数one、two代表的字典词汇的value是否相等

参数： 1，值相等或是同一个对象

0，值不相等

\*/

int32\_t is\_data\_equal(data\_t \* one, data\_t \* two)

{

/\* LOG-TODO \*/

if (!one || !two || !one->data || !two->data)

return 1;

if (one == two)

return 1;

if (one->len != two->len)

return 0;

if (one->data == two->data)

return 1;

if (memcmp(one->data, two->data, one->len) == 0)

return 1;

return 0;

}

/\*\*

\*函数功能： 销毁词汇value对象

\*/

void data\_destroy(data\_t \* data)

{

if (data) {

LOCK\_DESTROY(&data->lock);

if (!data->is\_static) {

if (data->data) {

if (data->is\_stdalloc)

free(data->data);

else

GF\_FREE(data->data);

}

if (data->vec)

GF\_FREE(data->vec);

}

data->len = 0xbabababa;

if (!data->is\_const)

GF\_FREE(data);

}

}

/\*\*

\*函数功能： 新创建一个词汇对象，将old词汇的内容copy至新对象，返回对象的值

\*/

data\_t \*data\_copy(data\_t \* old)

{

if (!old) {

gf\_log\_callingfn("dict", GF\_LOG\_WARNING, "old is NULL");

return NULL;

}

data\_t \*newdata = (data\_t \*) GF\_CALLOC(1, sizeof(\*newdata),

gf\_common\_mt\_data\_t);

if (!newdata) {

return NULL;

}

if (old) {

newdata->len = old->len;

if (old->data) {

newdata->data = memdup(old->data, old->len);

if (!newdata->data)

goto err\_out;

}

if (old->vec) {

newdata->vec =

memdup(old->vec,

old->len \* (sizeof(void \*) +

sizeof(size\_t)));

if (!newdata->vec)

goto err\_out;

}

}

LOCK\_INIT(&newdata->lock);

return newdata;

err\_out:

if (newdata->data)

FREE(newdata->data);

if (newdata->vec)

FREE(newdata->vec);

GF\_FREE(newdata);

return NULL;

}

/\*\*

\*函数功能： 利用快速哈希功能在this字典中快速查找key所对应的词汇，返回其指针

\*/

static data\_pair\_t \*\_dict\_lookup(dict\_t \* this, char \*key)

{

if (!this || !key) {

gf\_log\_callingfn("dict", GF\_LOG\_WARNING,

"!this || !key (%s)", key);

return NULL;

}

int hashval = SuperFastHash(key, strlen(key)) % this->hash\_size;

data\_pair\_t \*pair;

for (pair = this->members[hashval]; pair != NULL;

pair = pair->hash\_next) {

if (pair->key && !strcmp(pair->key, key))

return pair;

}

return NULL;

}

字典数据结构的主要实现机制已经列除来：字典创建，字典中词汇查询，获取字典的指定词汇的值，删除子典中指定词汇的值……。其它未在本文明确列出的函数，都是见名知意的函数，如data\_t \*data\_from\_int64(int64\_t value)函数实现的是将参数指定的int64\_t 类型的值写入该函数新创建的一个词汇的value对象的data部分。（你观察返回值是一个结构指针，指针必然指向一个对象，而这个对象没有在函数参数中传入，一般来说全局变量不会让你用函数返回值来返回其地址，因此你就指定这个函数里面实现了什么东西，跑去一看果然就是。看上两个之后，你往source insight列表左边一看，好开心，因为字典的实现机制上面全部介绍清楚了，the other函数这样你想看，就能看懂。1500多行，谁也不是没事了在哪里瞎研究，意会就行了，你在一个公司领的是薪水，为公司付出的除了代码，还有你宝贵而又少得可怜的时间，所以冗余的可以直接跳过。）

本质上来说只要看懂了头文件中的三个结构体的内容，不用看任何资料也知道它就应该那样去实现。所有有时候我们看c代码会比看c++的代码乐观很多，当一个工程师的c代码达到一定功力，只需看头文件就一切不看自明。

至此，数据结构之->字典分析完毕。

分析它就是要用到它，在应用程序中有两种情况使用字典是最合适不过的：配置文件、命令行

因为配置文件里面的内容有一定的格式，都是字符串、不定长。我x，完全吻合呀有没有。

命令行就是你在终端里面输入的命令行，它传给应用程序时也是字符串，分成一片一片的，有的参数是成对儿出现（比如network=[tcp | udp]），有的是只有命令没有值。这个也很适合字典呀。没有值的你可以不写值嘛对不对，一个字典里面只有词汇没有说明，也不会有错误。

（例如字典里面有一个词汇叫： 人：），显然“人”这个词汇本身就不需要解释。

工程师一定要抽象能力，也需要有联想能力。这些是好代码的灵魂。

下面是一颗树：



上面的图片有根，有分支，有叶子。假如你有50颗糖果，还有50条小绳子，让你把糖果挂到树上。在假设你一伸手就能够到树顶的情况下，假设这颗树长得非常规律，每个树杈最多有两个分支。按照下面的步骤把所有糖果挂到树上：

1. 把50颗糖果和绳子全部放到衣服兜里。
2. 从树根开始，网上一路瞄，只要发现有树杈还没有挂东西，就用一个绳子绑一颗糖果到树杈的分支上。
3. 然后你不能东挂一个西挂一个，你遵循某种规律：占在一个位置把能够得着的树枝挂完，一般来说，你要么从上往下挂，要么从下网上挂。你不会去在一个树枝的头挂一个，尾挂一个，然后再这样循环。（潜意识里你会用最节省时间，距离最短的方式去挂满一个分支）
4. 当挂满一个分支之后，你发现手里还有42颗糖果，然后你决定将剩余的糖果挂载另外一个枝头，此时你不得不走几步，挪动到另外一个顺手一些的位置，去挂你的糖果
5. ......
6. 经过简单操作，你手里的糖果挂完了。 然而，你发现有3个挂满了糖果的树枝，很喜庆。但是还有两个树枝上面一个糖果也没有，此时接近元旦，孩子们正在屋外等候你的礼物，超时远在10公里之外。你甚至有些后悔，刚才应该多买一些的。然而，此情此景你必须当机立断，否则后果是一盘切碎了的榴莲。而榴莲未必是用来吃的……

想来细思极恐，此时刚好其中一个稍纤细一些的树枝，经不住重量，微微一下垂，一个糖果滑落，正好砸在你的脑袋上。有了，你决定将挂满了糖果的树写到，首先你数了一下树杈的数量，总共有80个分叉。你决定将糖果不要挂那么满，平均一下，于是你一遍观察，一边将糖果挪到其树杈上。经过一番转移，终于任务完成。

现在这颗树整个看起来很好看，没个树枝上都挂着一些糖果。

1. 虽然挂完了糖果，作为工程师的你依没有能愉快起来，因为你的老板刚才发了一条短信过来，让你实现一个可以快速查找数据，有层级关系、且可排序、可动态调整的体育类账号管理系统，要求这个系统可以按照不同的成绩依据去排名。

你想到了，把每一颗糖果看做一个对象（不仅可以挂糖，也可以挂其它任何东西，如：read()、clsoe()、write()、lseek()），它有评估依据（成绩和比较方法），也有层级关系（在哪一层分支上）

1. 想着这个系统的数据结构特性，你默默都走到了电脑桌前，打开浏览器输入《二叉树》、

《自平衡二叉树》、《红黑树》，终于搜到一篇写的很好的文章：

1. 红黑树本质上是一个二叉查找树（BST），但是它从根到最远叶子的长度不会超过到最近叶子长度的两倍，因此是近似平衡的。

2. 红黑树的节点不是黑的就是红的，不会有第三种颜色。

3. 树根必须是黑色。

4. 叶子所指的空节点必须是黑色。

5. 如果某个节点是红色，那么它的两个儿子必须都是黑色。

6. 从任意节点出发的所有向下的路径上包含相同个数的黑节点。这个个数我们称为黑高度Bh。

话不多说，言归正传。这份文档中gluster用到的是红黑树。

rb.c和rb.h

R代表red，b代表black，对应红黑树的名字。

上面举的例子说明了二叉树的结构特性和作用场景。本质上不用去再实现一颗红黑树。从内核里面copy即可。并且这种数据结构类的代码很容易忘却，但是你却能记住名字及其特性，正好你可以有很多途径下载到这些源码，重在应用不用过于担心。

前面我说过，有一些代码是不用去一行一行看的，只要你看懂了数据结构（结构模型和时间的c语言中的结构体），代码就是你心中所想的那样，世界原本就是这么神奇，充满规律。下面从gluster源码当中分析一下红黑树的数据结构

#ifndef RB\_H

#define RB\_H 1

#include <stddef.h>

/\* Function types. \*/

typedef int rb\_comparison\_func(const void \*rb\_a, const void \*rb\_b,

void \*rb\_param);

typedef void rb\_item\_func(void \*rb\_item, void \*rb\_param);

typedef void \*rb\_copy\_func(void \*rb\_item, void \*rb\_param);

#ifndef LIBAVL\_ALLOCATOR

#define LIBAVL\_ALLOCATOR

/\* Memory allocator. 自平衡二叉树内存分配器结构\*/

struct libavl\_allocator {

void \*(\*libavl\_malloc) (struct libavl\_allocator \*,

size\_t libavl\_size);

void (\*libavl\_free) (struct libavl\_allocator \*,

void \*libavl\_block);

};

#endif

/\* Default memory allocator. \*/

extern struct libavl\_allocator rb\_allocator\_default; // 自平衡二叉树内存分配器对象

void \*rb\_malloc(struct libavl\_allocator \*, size\_t); // 自平衡二叉树内存分配器malloc方法

void rb\_free(struct libavl\_allocator \*, void \*); // 自平衡二叉树内存分配器free方法

/\* Maximum RB height. \*/

#ifndef RB\_MAX\_HEIGHT

#define RB\_MAX\_HEIGHT 48 // gluster中红黑树（限定为）最高48层，2^48个节点足够用（实际上单颗数并没有这么多节点）

#endif

/\* Tree data structure. 整颗红黑树\*/

struct rb\_table {

struct rb\_node \*rb\_root; /\* Tree's root. 树根节点\*/

rb\_comparison\_func \*rb\_compare; /\* Comparison function. 查找函数\*/

void \*rb\_param; /\* Extra argument to |rb\_compare|. 红黑树万能参数\*/

struct libavl\_allocator \*rb\_alloc; /\* Memory allocator. 红黑树内存分配器结构指针\*/

size\_t rb\_count; /\* Number of items in tree. 整颗红黑树中的叶片/节点数量\*/

unsigned long rb\_generation; /\* Generation number. 流水号\*/

};

/\* Color of a red-black node. 红黑树颜色枚举体\*/

enum rb\_color {

RB\_BLACK, /\* Black. \*/

RB\_RED /\* Red. \*/

};

/\* A red-black tree node. 红黑树节点\*/

struct rb\_node {

struct rb\_node \*rb\_link[2]; /\* Subtrees. 子节点\*/

void \*rb\_data; /\* Pointer to data. 树的数据\*/

unsigned char rb\_color; /\* Color. 树的颜色\*/

};

/\* RB traverser structure. 红黑树（遍历相关）属性\*/

struct rb\_traverser {

struct rb\_table \*rb\_table; /\* Tree being traversed. 指向被遍历的树\*/

struct rb\_node \*rb\_node; /\* Current node in tree. 指向树中遍历到的当前节点\*/

struct rb\_node \*rb\_stack[RB\_MAX\_HEIGHT]; //缓存树节点的索引路径（最大等于树高：48层）

/\* All the nodes above |rb\_node|. \*/

size\_t rb\_height; /\* Number of nodes in |rb\_parent|. \*/

unsigned long rb\_generation; /\* Generation number. 流水号\*/

};

/\*\*

\*红黑树构建方法（构建、插入（绑糖果和绳子）、删除（从树杈上取下一个糖果吃了/扔了）、查询（去数一下上面写者大白兔的哪一个糖在哪里挂着，写者徐福记的糖在哪里挂着……）、交换（将大白兔和阿尔卑斯两颗糖互换位置））：

\*/

struct rb\_table \*rb\_create(rb\_comparison\_func \*, void \*,

struct libavl\_allocator \*);

struct rb\_table \*rb\_copy(const struct rb\_table \*, rb\_copy\_func \*,

rb\_item\_func \*, struct libavl\_allocator \*);

void rb\_destroy(struct rb\_table \*, rb\_item\_func \*);

void \*\*rb\_probe(struct rb\_table \*, void \*);

void \*rb\_insert(struct rb\_table \*, void \*);

void \*rb\_replace(struct rb\_table \*, void \*);

void \*rb\_delete(struct rb\_table \*, const void \*);

void \*rb\_find(const struct rb\_table \*, const void \*);

void rb\_assert\_insert(struct rb\_table \*, void \*);

void \*rb\_assert\_delete(struct rb\_table \*, void \*);

#define rb\_count(table) ((size\_t) (table)->rb\_count)

/\*\*

\*红黑树遍历方法：

\*/

void rb\_t\_init(struct rb\_traverser \*, struct rb\_table \*);

void \*rb\_t\_first(struct rb\_traverser \*, struct rb\_table \*);

void \*rb\_t\_last(struct rb\_traverser \*, struct rb\_table \*);

void \*rb\_t\_find(struct rb\_traverser \*, struct rb\_table \*, void \*);

void \*rb\_t\_insert(struct rb\_traverser \*, struct rb\_table \*, void \*);

void \*rb\_t\_copy(struct rb\_traverser \*, const struct rb\_traverser \*);

void \*rb\_t\_next(struct rb\_traverser \*);

void \*rb\_t\_prev(struct rb\_traverser \*);

void \*rb\_t\_cur(struct rb\_traverser \*);

void \*rb\_t\_replace(struct rb\_traverser \*, void \*);

#endif /\* rb.h \*/

红黑树的作用：

一个树杈上既可以挂糖果，也可以挂勺子，杯子（假设是一个塑料杯）

Gluster是使用栈回调的方式，将不同功能的组件（类似不同口味的糖果）有组织，有纪律的结合起来，构成更结构上简单，能够自适应，灵活的分布式文件系统。而承载这些不同功能组件的，正是红黑树。

内核链表：

Linux内核链表是一种非常经典的数据结构，它是面向对象和面向过程的完美结合。将链表结构 与 数据分离（上面介绍的红黑树也是如此）。

将链表内嵌在数据结构中，这样一条链表上的结构体未必完全一样。保证了逻辑上链表的结构，又允许节点多样性。

内核链是双向循环链表，它的每一个内联函数的实现都有详细的英文注释，直接去用，此处不再贴出源码注释。

哈希表：

哈希表是一个定长的数组，数组里面的每一个元素可以是内嵌了链表的结构指针，也可以是普通的结构体或普通变量。

数组的长度为index。哈希这两个字指的是给出一个下标，可以利用O(1)的查询速度去快速定位手柄。根据手柄去查询链表。哈希表与红黑树、内核链表一样，不需要自己写，也不需要记住怎么写（哪怕你上个月分析的清清楚楚，你甚至可以优化它，单再过两个月一定忘的干干净净，这是人类的特性）。还是经常强调的，对于数据结构这种源码，你需要记住它的特性以应用场景，对于应用程序这种源码，你需要分析清楚它的模型并且大概记住（记不住请写文档），对于算法这种源码需要知道输入和输出就可以了。

比如说我现在定义一个变量：

struct student\_list{

struct list\_head list; //链表结构

Int yuwen\_core;

Int shuxue\_core;

Int english\_core;

Int level; //年级

Char age; //年龄

Char \*province; //省份

Boolean married; //结婚否（注意是博士生哈，不是小学生，没有毛病）

Char \*Hobbies; //爱好

......

};

假如现在有100\*100个student\_list对象，让你用一种方式去管理这些学生的信息。

首先想到的是数组，定义一个10000个成员的数组，也是O(1)的速度，但是它有两个缺点：内容不能动态调整（不灵活）、大小不能动态调整（消耗资源）。

那你换了一个方式，我在上面的段落不是一直在鼓吹内核链表多么牛x嘛，那就用链表试一试，struct student\_list \*head; 内存消耗严重的问题解决了，然后出现了尴尬的事情：为了查询一个学生的信息，必须要从链表头找到尾（最严重情况需要遍历1万个节点）。这得消耗多少时间啊！！！

为了交换两个学生的顺序，得先查两个学的位置，再交换（最严重情况需要遍历2万次）。

尤其是当学生多了之后，比如让你查全院的学生信息（假设8000个学生），细思极恐……

显然，数组和链表都不是很合适。如果将数组和哈希的优点结合起来，就能解决上述问题。

恩，哈希表就是这种数据结构！！！

student\_list \*my\_hash[100];

my\_hash[x]每一个成员是一个结构体指针，由于student\_list 内嵌了链表，所以每个数组元素可以作为链表的头 或 链表中的任何一个节点。

这样，你有最大100个链表。索引一个学生只需要：找到链表头O(1)，找到链表中目标学生所在的节点O(1~100)，最多101次查询，就可以从10000个学生中查找出目标学生。速度虽然不及数组，但是也不慢，学生对象占用的内存也是按需分配。很合适。

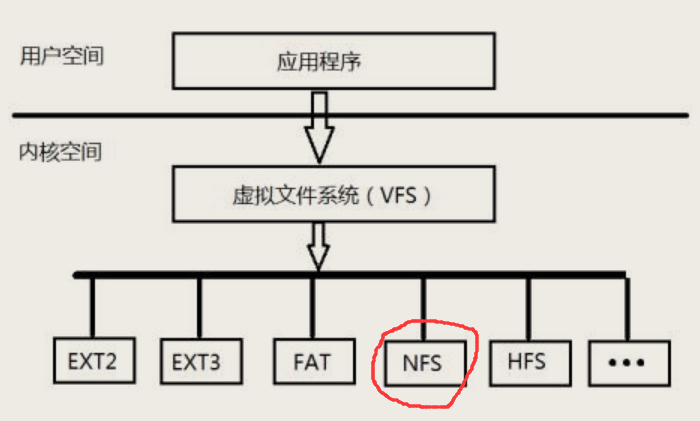
Gluster用到的数据结构就是以上五种（数组、字典、内核链表、红黑树、哈希表） 和 它的组合变体 以及数据结构的嵌套，先讲解数据结构是为了补全读者的基础知识（会的就可以直接跳过前面章节，直接读第六章的内容），让读者在看代码时知道这一块代码大概是什么意思，不至于迷失在代码里面。

**第六章**

----------------------------分布式文件系统模型

分布式存储文件系统抽象：

先把分布式两个字去掉，它的本相是一个文件系统。下面的图读者再熟悉不过。



注意，上面有一个文件系统我画上了红色圆圈。

Nfs是一个网络文件系统，网络文件系统是分布式文件系统的一种简化版本。当然技术实现上虽有相似之处，设计目标却是有区别的，不然不就是同一个产品了吗。

区别如下：

1. 网络文件系统（nfs）是单服务器，可以有多个客户端，其设计目标是让一台节点可以通过网络来看到另外一个远程节点。
2. 分布式文件系统是集群方式（多个服务器节点组成统一视图对外服务），其设计目标是让几千个客户端可以同时访问由几万个节点组成的服务器的存储空间，解决的是磁盘利用率不均衡、超大文件存放速度安全问题、海量文件统一视图问题。由于是并发访问，分布式存储文件系统还需要解决冲突问题。

共性：

1. 都是c/s模式，客户端访问服务器的目录。反之则不行。
2. 与本地文件系统相比，他们都是利用了rpc技术实现业务映射。

说到rpc技术，我们来先聊下电影。《大话西游》看过没有？里面的至尊宝大概是一个人从18岁到35岁之间的所有心理路程，经典的电影在于它包含的主题与大部分人的人生过往或多或少都有一丝关系……。当然我们是一个专业，严肃的技术问题，不是艺术人生教人欣赏电影的。 我们换一个话题：看《西游降魔》（我x，这不还是电影么！）……。淡定，还记得里面孙悟空捉猪八戒的片段吗，分为五个步骤：

引出猪八戒、由孙悟空抓住它、由唐僧教育引导它、观察教育效果、如果不行就杀死猪八戒，行则一起开取经大业新局面。

故事太长，写字太慢，我们只讲第一段儿：

听话符：道士经常用的那种

动作发起者：显然是个男的，而且有一定舞蹈基础那种，非唐三藏莫属。

动作跟随者：显然，应该是一个美女，不然猪八戒不会想起那段伤心的往事。

流程：

导演确定拍摄主题为：西游降魔第一集第38镜，第12次拍摄

演员就位、摄影就位、灯光师就位、道具师就位、助演（拿好你的提词器）就位、

导演喊action

唐三藏与舒淇分别贴上一张符咒（两位演员商量好动作流程），舒淇跑到靠近山洞的远处，与孙悟空保持距离。

唐三藏开始观察远处的舒淇的状态 舒淇站到指定位置，摆出pos等待指令到来

唐三藏看到ok，开始伸出右手 舒淇伸出右手

唐三藏开始弯腰 舒淇开始弯腰

唐三藏转身 舒淇转身

唐三藏xxx 舒淇xxx

唐三藏yyy 舒淇yyy

唐三藏www 舒淇www

…… ……

拍摄过程演员出现笑场，在拍摄现场紧张观看演员表演是否出现失误的导演这时候果断喊卡，与演员简单交流之后，重新喊action

…… ……

注意，重点来了！！！

猪八戒开始出现，孙悟空上场，以迅雷不及掩耳盗铃之势将猪八戒捉住，……后面的事情与本技术无关，不再详细描述。

终于西游降魔第一集第38镜第52次顺利拍摄完毕，导演喊卡。

上面的电影拍摄过程很清晰，有四个人来演这一镜：

孙悟空与唐僧是一个组的

舒淇与猪八戒是一个组的

唐僧可以通过一个听话符来远程控制舒淇的肢体动作。

舒淇搜索附近的土地，搜索出了猪八戒

舒淇通过听话符告诉唐三藏猪八戒来了

唐三藏告诉孙悟空猪八戒来了。

猪八戒被抓获。

由于猪八戒的特性、听话符的特性，理论上来说孙悟空只要想，他都可以通过这种看起来非常标准和丝滑版流畅的操作收发，捉到很多只猪八戒（假设八戒还有一些见死不救的酒肉兄弟也在附近）

一句话总结：实际上这一段的boss就是孙悟空，整个过程是围绕他想抓妖这一个需求来的。

孙悟空利用了唐三藏可以控制舒淇这一特点，引出猪八戒，捉之而后快。

言归正传，本文档第六章是将分布式存储模型的，提到了nfs技术，引出了rpc技术，rpc全拼是Remote Procedure Call，即远程过程调用。 见名知意:

一个节点是rpc客户端（把唐三藏叫做客户端）

一个节点是rpc服务器（把舒淇叫做服务器）

操作rpc客户端的叫nfs客户端

rpc服务器调用的对象叫nfs服务器

Nfs服务器导出的可操作目录里面由/etc/exports指定

很不巧，/etc/exports配置文件里面只写了一句话：我们只抓猪八戒，我们会继续保证其它妖怪幸福稳定的生活……。

上面的rpc的比喻。Rpc是一个固定格式的消息体+payload，来实现流程管控，数据收发。

我写了两个rpc的示例，存放在本文档同级文件夹rpc-demo下面

（1）其中rpc-dictionary子文件夹下面的例程是通过rpc客户端来远程读写rpc服务器的内存

（2）rpc-readfile子文件夹下面的历程是通过rpc客户端远程读取rpc服务器上面的文件

（文件绝对路径名，或相对路径名（相对于rpc服务器软件的pwd路径），由rpc客户端命令行输入）。运行效果就是，在客户端输入一个路径名，就可以在rpc客户端上打印rpc服务器上面指定的文件的文件内容。

既然可以远程read()、就可以实现write()、lseek()、truncate()、readdir()、fsync()、......

close()、open()这两个在read()中已实现（肯定是要先open成功才能read的，想都不要想）。

也就是说在rpc客户端上，完全可以操作rpc服务器上的文件！！！

这样来说，对rpc进行功能扩展，将每一个功能都用专门的功能模块去实现，不断的扩充，即可最终写成一个分布式存储文件系统。文件系统已经广泛存在，重点在与“分布式”这两个字。所有的源码都在实现以下功能：多节点的业务分工协作、单节点的业务模型搭建。

本章描述的是分布式文件系统模型，当然不是乱起名字。下面我们来聊一下，一个从框架层面开始设计的分布式文件系统需要哪些模块。

Cli模型：用于将终端输入的命令解析、添加缺省值、生成服务调用请求

Rpc模型：rpclib也就是rpc框架，讲道理的来说，linux节点本身是存在一个公共rpc的，你的ssh，telnet的核心机制都是rpc。但是这个rpc有个缺点就是如果实现一些私有化的东西比较困难。如果分布式文件系统都会实现自己的rpclib框架。Rpc-transport，rpc传输层，就是将网络包写成api接口形式，在需要rpc服务时给rpc对象实例化。（有socket、rdma等）。

Argp-standalone模块：某些版本的操作系统不自带参数解析器，一般的linux节点都有参数解析器，是一个开源的参数解析软件。在有的系统中没有的话，参数解析的需求不可避免，分布式软件会自带这个参数解析器。

Booster：分布式文件系统与应用程序的接口，分布式文件系统操作一个文件的api与其它文件系统一样，所有需要在fuse层截断后重新映射到分布式文件系统，然后到rpc，到网络包，到远程截节点，远程节点返回操作结果或数据。本文件就是一个壳，将vfs与分布式文件系统映射起来。

Contrib：

Fuse库，是分布式文件系统的公共实现部分，私有实现部分在libglusterfs中实现。具体操作可搜索网络资源，本次不再贴出描述。

MD5加密算法：一串字符串通过迭代，生成一个纯大小，或纯小写的32byte的字符串，其分段固定格式。

Uuid：uuid是保证唯一识别号的一个算法，会结合节点名、pid、时间戳等信息，生成一个唯一的编号。也是分段固定格式的。

Rbtree：红黑二叉树，前面有介绍。

Extras：集群与系统配置、管理、测试相关的脚本文件。

Glusterfsd:一个精灵进程，读取卷配置文件、初始化集群软件环境、打开或创建日志文件对象，嵌入到去全局状态管理对象中。解析cli命令，封装成事件分发出去。

Libglusterfs 文件系统的具体实现，与fuse接口挂钩。

lighttpd是一个为lighttpd编写的模块，用于绕过fuse，加速glusterfs上文件的访问。

调度器：计算节点权值的模块，一个文件是否存在这个节点上，或者另外一个节点上，要根据一个权值来比较，权值高的节点胜出承担业务。

转换器：负责消息传递和状态回调。

**第七章**

----------------cli模型

Cli模型目标：

简单应用程序的命令通过strcmp()，strcok\_r()函数就可以实现传递，复杂的程序其参数众多，需要专门的代码去将输入参数处理，然后传递给不同的精灵进程，因此出现cli模型。

Cli模型需要：将在控制台输入的所有命令检查格式是否合理：[] () {} 必须成对出现，一条大命令的参数部分需要用空格分割 或 用“，”分割。创建请求帧、检测执行结果打印到控制台等。

于是cli模型的源码就是有三部分构成：

检查命令格式、命令分片、命令调用

在本文档中贴出一部分cli源码的解析，详细文档请参考gluster源码中cli部分，有更为详细的注释。

|  |
| --- |
| #ifndef \_CONFIG\_H #define \_CONFIG\_H #include "config.h" #endif  #include <stdio.h> #include <string.h> #include <stdlib.h>  #include "cli.h" #include "cli-cmd.h"  /\*\* \*函数功能：        判断参数ch是否是空格 \*返回值： \*                返回1，ch字符是空格。                 \*                返回0，ch字符不是空格 \*/ static int \_\_is\_spc(int ch) {         if (ch == ' ')                 return 1;         return 0; }  /\*\* \*函数功能：        判断ch是否是参数分割符：'(' ')' '[' ']' '{' '}' '|' \*返回值：        1，是分割符        0，不是 \*/ static int \_\_is\_div(int ch) {         switch (ch) {         case '(':         case ')':         case '<':         case '>':         case '[':         case ']':         case '{':         case '}':         case '|':                 return 1;         }          return 0; }  /\*\* \*函数功能：        判断word指针所指向的字符是否是目标字符（命令行参数除分割符 和 空格之外的都是目标字符） \*返回值：                1，是目标字符（纯命令或参数片段）                        0，不是目标字符 \*/ static int \_\_is\_word(const char \*word) {         return (!\_\_is\_div(\*word) && !\_\_is\_spc(\*word)); }  /\*\* \*函数功能：        返回参数ch的对应字符 \*返回值：        -1，没有对应字符        other，对应字符 \*/ int counter\_char(int ch) {         switch (ch) {         case '(':                 return ')';         case '<':                 return '>';         case '[':                 return ']';         case '{':                 return '}';         }          return -1; }  /\*\* \*函数功能：        判断命令行参数中'[' ']' '{' '}' '(' ')'是否是成对出现 \*返回值：        NULL，不成对出现 或者["内容"] ("内容") {"内容"},内容中出现了空格        非NULL，']' '}' ')'左边的地址（指向被包含数据的尾巴） \*/ const char \*\_\_is\_template\_balanced(const char \*template) {         const char \*trav = NULL;         int ch = 0;          trav = template;          while (\*trav) {                 ch = \*trav;                  switch (ch) {                 case '<':                 case '(':                 case '[':                         trav = \_\_is\_template\_balanced(trav + 1);                         if (!trav)                                 return NULL;                         if (\*trav != counter\_char(ch))                                 return NULL;                         break;                 case '>':                 case ')':                 case ']':                         return trav;                 }                  trav++;         }          return trav; }  /\*\* \*函数功能：        封装\_\_is\_template\_balanced，对外提供api接口，判断template字符串是否是格式合法的命令行 \*返回值：        0，是格式合法命令行字符                -1，不是 \*/ int is\_template\_balanced(const char \*template) {         const char \*trav = NULL;          trav = \_\_is\_template\_balanced(template);         if (!trav || \*trav)                 return -1;          return 0; }  /\*\* \*函数功能：        统计命令行中，单元字符的数量 \*                （单位字符：'<' '>' '(' ')' '[' ']' '{' '}' '|'都是一个单元字符，而被这些字符包含的"content"整体算一个单元字符） \*/ int cli\_cmd\_token\_count(const char \*template) {         int count = 0;         const char \*trav = NULL;         int is\_alnum = 0;          for (trav = template; \*trav; trav++) {                 switch (\*trav) {                         case '<':                         case '>':                         case '(':                         case ')':                         case '[':                         case ']':                         case '{':                         case '}':                         case '|':                                 count++;                                 /\* fall through \*/                         case ' ':                                 is\_alnum = 0;                                 break;                         default:                                 if (!is\_alnum) {                                         is\_alnum = 1;                                         count++;                                 }                 }         }          return count + 1; }  /\*\* \*函数功能：        释放二级指针tokens执向的一级指针的所有内存段、释放二级指针tokens本身执行的内存段 \*例如：                         char \*string\_tbl[0];                 string\_tbl = (char \*\*)malloc(4);                 string\_tbl[0] = (char \*)malloc(12);        sprintf(string\_tbl[0], "%s", "hello world");                 string\_tbl[1] = (char \*)malloc(9);        sprintf(string\_tbl[1], "%s", "shenzhen");                 string\_tbl[2] = (char \*)malloc(8);        sprintf(string\_tbl[2], "%s", "cluster");                 string\_tbl[3] = NULL;                 现在想要释放string\_tbl，调用本函数即可完成。 \*/ void cli\_cmd\_tokens\_destroy(char \*\*tokens) {         char \*\*tokenp = NULL;          if (!tokens)                 return;          tokenp = tokens;         while (\*tokenp) {                 free(\*tokenp);                 tokenp++;         }          free(tokens); }  /\*\* \*函数功能：                将template命令行写入tokens中（底层操作，非API接口） 例如：                         template字符串为：        "gluster peer probe [server2]"，调用本函数后：         tokens[0] = "gluster";        tokens[1] = "peer";        tokens[2] = "probe";         tokens[3] = "[";        tokens[4] = "server2";        tokens[5] = "]"; \*/ int cli\_cmd\_tokens\_fill(char \*\*tokens, const char \*template) {         const char \*trav = NULL;         char \*\*tokenp = NULL;         char \*token = NULL;         int ret = 0;         int ch = 0;          tokenp = tokens;          for (trav = template; \*trav; trav++) {                 ch = \*trav;                  if (\_\_is\_spc(ch))                         continue;        //跳过template中的空格字符                  if (\_\_is\_div(ch)) {        //处理template中的参数分割字符                         token = calloc(2, 1);        //在内存的动态存储区中分配2个长度为1byte的连续空间                         if (!token)                                 return -1;                         token[0] = ch;                          \*tokenp = token;                         tokenp++;                          continue;                 }                  token = strdup(trav);                 \*tokenp = token;                 tokenp++;                  for (token++; \*token; token++) {                         if (\_\_is\_spc(\*token) || \_\_is\_div(\*token)) {                                 \*token = 0;                                 break;                         }                         trav++;                 }         }          return ret; }  /\*\* \*函数功能：                将template命令拆分成片段，将片段写入tokens中（api接口，对外提供服务） \*/ char \*\*cli\_cmd\_tokenize(const char \*template) {         char \*\*tokens = NULL;         int ret = 0;         int count = 0;          ret = is\_template\_balanced(template);                //命令行格式不合法（特殊字符不是成对出现）、返回NULL         if (ret)                 return NULL;          count = cli\_cmd\_token\_count(template);                //统计命令行template中单元字符的数量，若数量为0则返回NULL         if (count <= 0)                 return NULL;          tokens = calloc(count + 1, sizeof(char \*));        //申请单元数量的内存         if (!tokens)                 return NULL;          ret = cli\_cmd\_tokens\_fill(tokens, template);        //将命令行template写入二级指针         if (ret)                 goto err;          return tokens;                                        //返回二级指针（二级指针是将命令行拆分成片段后的结构）       err:         cli\_cmd\_tokens\_destroy(tokens);                        //若写入过程中出错，则销毁所有内存、并返回NULL         return NULL; }  /\*\* \*函数功能:                从word命令树枝（树叶）分支中，找到关键字为 \*token的叶片，并返回该叶片的指针 \*/ struct cli\_cmd\_word \*cli\_cmd\_nextword(struct cli\_cmd\_word \*word,                                       const char \*token) {         struct cli\_cmd\_word \*next = NULL;         struct cli\_cmd\_word \*\*trav = NULL;         int ret = 0;          if (!word->nextwords)                 return NULL;          for (trav = word->nextwords; (next = \*trav); trav++) {                 if (next->match) { //                        ret = next->match ();                 } else {                         ret = strcmp(next->word, token);        //遍历、字符串比较                 }                  if (ret == 0)                         break;         }          return next; }  /\*\* \*将token关键字插入word某一个分支的空叶片位置（新叶片采用的是增量方式） \*/ struct cli\_cmd\_word \*cli\_cmd\_newword(struct cli\_cmd\_word \*word,                                      const char \*token) {         struct cli\_cmd\_word \*\*nextwords = NULL;         struct cli\_cmd\_word \*nextword = NULL;          nextwords = realloc(word->nextwords,                             (word->nextwords\_cnt +                              2) \* sizeof(\*nextwords));                //为保存旧的所有叶片 申请空间nextwords\_cnt+2个数量的 size为cli\_cmd\_word结构体的内存空间         if (!nextwords)                 return NULL;          word->nextwords = nextwords;                                //word的分支          nextword = calloc(1, sizeof(\*nextword));                //为新的叶片创建空间         if (!nextword)                 return NULL;          nextword->word = strdup(token);                                //关键字拷贝给新叶片          if (!nextword->word) {                 free(nextword);                 return NULL;         }          nextword->tree = word->tree;                                //建立叶片层级关系树         nextwords[word->nextwords\_cnt++] = nextword;                //建立叶片现行关系树         nextwords[word->nextwords\_cnt] = NULL;                        //新叶片在升级为树枝之前，没有子叶片          return nextword;                //返回增量后（且写入了token）的树枝 }   /\*\* \*将命令行原始命令pattern、回调函数cbkfn、切片后的命令行参数tokens，desc字符串，写入命令行树tree合适的位置中 \*/ int cli\_cmd\_ingest(struct cli\_cmd\_tree \*tree, char \*\*tokens,                cli\_cmd\_cbk\_t \* cbkfn, const char \*desc,                const char \*pattern) {         int ret = 0;         char \*\*tokenp = NULL;         char \*token = NULL;         struct cli\_cmd\_word \*word = NULL;         struct cli\_cmd\_word \*next = NULL;          word = &tree->root;          //遍历、分别找到命令树中，tokens 纯参数/命令部分已经挂载的叶片，如果没有找到，则在树中新生成一片叶子出来，并设置其token         for (tokenp = tokens; (token = \*tokenp); tokenp++) {                 if (!\_\_is\_word(token))                         break;                  next = cli\_cmd\_nextword(word, token);                                 if (!next)                         next = cli\_cmd\_newword(word, token);                  word = next;                 if (!word)                         break;         }          if (!word)                 return -1;        //若没有一片叶子和树枝，则返回          if (word->cbkfn) {                 /\* warning - command already registered \*/        //若顶级命令已经注册了回调函数，3.2版本中不处理这种情况         }          word->cbkfn = cbkfn;        //覆盖顶级命令（本次命令）的回调函数、参数目的描述字符串、原始参数字符串         word->desc = desc;         word->pattern = pattern;          /\* end of static strings in command template \*/         /\* TODO: autocompletion beyond this point is just "nice to have" \*/          return ret; }  /\*\* \*注册命令：将gluster某子命令集合cmd 写入命令执行树（由控制台命令行输入所构建）中 \*/ int cli\_cmd\_register(struct cli\_cmd\_tree \*tree, struct cli\_cmd \*cmd) {         char \*\*tokens = NULL;         int ret = 0;          GF\_ASSERT(cmd)          if (cmd->reg\_cbk)                //执行回调函数，检测cmd命令是否应该被禁止执行（若是，则回调函数设置：cmd->disable = 0）                 cmd->reg\_cbk(cmd);          if (cmd->disable) {                //经过检测，发现cmd命令虽然格式合法，但是不应该执行，则跳出命令的注册。                 ret = 0;                 goto out;         }          tokens = cli\_cmd\_tokenize(cmd->pattern);        //将命令行写入分片后的二级指针         if (!tokens) {                 ret = -1;                 goto out;         }          ret =             cli\_cmd\_ingest(tree, tokens, cmd->cbk, cmd->desc,                            cmd->pattern);                //将命令行、分片后的二级指针（命令行）、回调函数写入命令行执行树         if (ret) {                 ret = -1;                 goto out;         }         ret = 0;        out:         if (tokens)                 cli\_cmd\_tokens\_destroy(tokens);                //tokens完成写入命令行树的任务后（数据已经拷贝到树中），释放其占用的内存          gf\_log("cli", GF\_LOG\_DEBUG, "Returning %d", ret);         return ret; } |
|  |

**第八章**

---------------------------内存池模型

Gluster的内存池有两种模式：

直接调用系统调用malloc() realloc() calloc()

gluster在系统调用基础上封装后的内存池。

显然封装后的内存池模型性能更好（通过减少系统调用内存分配函数的频率，来实现更迅速和减少碎片化的功能）

本部分源码并不长，我直接贴出来代码注释

先看头文件注释：mem-pool.h

/\*

Copyright (c) 2008-2011 Gluster, Inc. <http://www.gluster.com>

This file is part of GlusterFS.

GlusterFS is free software; you can redistribute it and/or modify

it under the terms of the GNU General Public License as published

by the Free Software Foundation; either version 3 of the License,

or (at your option) any later version.

GlusterFS is distributed in the hope that it will be useful, but

WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program. If not, see

<http://www.gnu.org/licenses/>.

\*/

#ifndef \_MEM\_POOL\_H\_

#define \_MEM\_POOL\_H\_

#include "list.h"

#include "locking.h"

#include "logging.h"

#include "mem-types.h"

#include <stdlib.h>

#include <inttypes.h>

#include <string.h>

#include <stdarg.h>

//acct内存分配对象

struct mem\_acct\_rec {

size\_t size; //转换器下，该类型的有效内存总size

size\_t max\_size; //>=size

uint32\_t num\_allocs; //转换器下，该类型的内存的单元数量

uint32\_t max\_num\_allocs; //>=num\_allocs

gf\_lock\_t lock;

};

//每一个转换器下面都会有一个本对象

struct mem\_acct {

uint32\_t num\_types; //rec总的对象数量

struct mem\_acct\_rec \*rec; //连续n个mem\_acct\_rec对象的启始地址，下标是type。

};

void \*\_\_gf\_calloc(size\_t cnt, size\_t size, uint32\_t type);

void \*\_\_gf\_malloc(size\_t size, uint32\_t type);

void \*\_\_gf\_realloc(void \*ptr, size\_t size);

int gf\_vasprintf(char \*\*string\_ptr, const char \*format, va\_list arg);

int gf\_asprintf(char \*\*string\_ptr, const char \*format, ...);

void \_\_gf\_free(void \*ptr);

/\*\*

\*内联函数，malloc()分配size个空间，函数返回一个指向分配起始地址的指针；如果分配不成功，生成日志，返回NULL。

\*/

static inline void \*\_\_gf\_default\_malloc(size\_t size)

{

void \*ptr = NULL;

ptr = malloc(size);

if (!ptr)

gf\_log\_nomem("", GF\_LOG\_ALERT, size);

return ptr;

}

/\*\*

\*内联函数，在内存动态存储区分配cnt个长度为size的连续空间，函数返回一个指向分配起始地址的指针；如果分配不成功，生成日志，返回NULL。

\*/

static inline void \*\_\_gf\_default\_calloc(int cnt, size\_t size)

{

void \*ptr = NULL;

ptr = calloc(cnt, size);

if (!ptr)

gf\_log\_nomem("", GF\_LOG\_ALERT, (cnt \* size));

return ptr;

}

/\*\*

\*内联函数，realloc()重新调整oldptr指向的内存空间为size大小，函数返回一个指向分配起始地址的指针；如果重新分配不成功，生成日志，返回NULL。

void \*realloc(void \*mem\_address, unsigned int newsize);

新的大小可大可小（如果新的大小大于原内存大小，则新分配部分不会被初始化；如果新的大小小于原内存大小，可能会导致数据丢失

1. realloc失败的时候，返回NULL

2. realloc失败的时候，原来的内存不改变，不会释放也不会移动

\*/

static inline void \*\_\_gf\_default\_realloc(void \*oldptr, size\_t size)

{

void \*ptr = NULL;

ptr = realloc(oldptr, size);

if (!ptr)

gf\_log\_nomem("", GF\_LOG\_ALERT, size);

return ptr;

}

//内存分配宏，内联函数封装系统调用、日志

#define MALLOC(size) \_\_gf\_default\_malloc(size)

#define CALLOC(cnt,size) \_\_gf\_default\_calloc(cnt,size)

#define REALLOC(ptr,size) \_\_gf\_default\_realloc(ptr,size)

#define FREE(ptr) \

if (ptr != NULL) { \

free ((void \*)ptr); \

ptr = (void \*)0xeeeeeeee; \

}

//acct内存分配宏，gluster详细封装了系统调用、日志

#define GF\_CALLOC(nmemb, size, type) \_\_gf\_calloc (nmemb, size, type)

#define GF\_MALLOC(size, type) \_\_gf\_malloc (size, type)

#define GF\_REALLOC(ptr, size) \_\_gf\_realloc (ptr, size)

#define GF\_FREE(free\_ptr) \_\_gf\_free (free\_ptr);

/\*\*

\*dump() src指向的字符串，返回拷贝后的字符串首地址

\*/

static inline char \*gf\_strdup(const char \*src)

{

char \*dup\_str = NULL;

size\_t len = 0;

len = strlen(src) + 1;

dup\_str = GF\_CALLOC(1, len, gf\_common\_mt\_strdup);

if (!dup\_str)

return NULL;

memcpy(dup\_str, src, len);

return dup\_str;

}

//gluster内存池类

struct mem\_pool {

struct list\_head list; //链

int hot\_count; //使用中的内存片数量

int cold\_count; //未使用的内存片数量

gf\_lock\_t lock; //锁

unsigned long padded\_sizeof\_type;

//数据类型 + GF\_MEM\_POOL\_PAD\_BOUNDARY(12byte: struct list + int)

void \*pool; //内存池有效数据首地址

void \*pool\_end; //内存池有效数据尾地址

int real\_sizeof\_type; //纯数据类型的size

};

struct mem\_pool \*mem\_pool\_new\_fn(unsigned long sizeof\_type,

unsigned long count);

#define mem\_pool\_new(type,count) mem\_pool\_new\_fn (sizeof(type), count)

void mem\_put(struct mem\_pool \*pool, void \*ptr);

void \*mem\_get(struct mem\_pool \*pool);

void \*mem\_get0(struct mem\_pool \*pool);

void mem\_pool\_destroy(struct mem\_pool \*pool);

int gf\_mem\_acct\_is\_enabled();

void gf\_mem\_acct\_enable\_set();

#endif /\* \_MEM\_POOL\_H \*/

，然后看内存池模型的实现方式mem-pool.c

**/\***

**Copyright (c) 2008-2011 Gluster, Inc. <http://www.gluster.com>**

**This file is part of GlusterFS.**

**GlusterFS is free software; you can redistribute it and/or modify**

**it under the terms of the GNU General Public License as published**

**by the Free Software Foundation; either version 3 of the License,**

**or (at your option) any later version.**

**GlusterFS is distributed in the hope that it will be useful, but**

**WITHOUT ANY WARRANTY; without even the implied warranty of**

**MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU**

**General Public License for more details.**

**You should have received a copy of the GNU General Public License**

**along with this program. If not, see**

**<http://www.gnu.org/licenses/>.**

**\*/**

**#include "mem-pool.h"**

**#include "logging.h"**

**#include "xlator.h"**

**#include <stdlib.h>**

**#include <stdarg.h>**

**#define GF\_MEM\_POOL\_LIST\_BOUNDARY (sizeof(struct list\_head))**

**#define GF\_MEM\_POOL\_PAD\_BOUNDARY (GF\_MEM\_POOL\_LIST\_BOUNDARY + sizeof(int))**

**#define mem\_pool\_chunkhead2ptr(head) ((head) + GF\_MEM\_POOL\_PAD\_BOUNDARY)**

**#define mem\_pool\_ptr2chunkhead(ptr) ((ptr) - GF\_MEM\_POOL\_PAD\_BOUNDARY)**

**#define is\_mem\_chunk\_in\_use(ptr) (\*ptr == 1)**

**#define GF\_MEM\_HEADER\_SIZE (4 + sizeof (size\_t) + sizeof (xlator\_t \*) + 4 + 8)**

**#define GF\_MEM\_TRAILER\_SIZE 8**

**//acct内存分配机制，内存头尾分配机制**

**#define GF\_MEM\_HEADER\_MAGIC 0xCAFEBABE**

**#define GF\_MEM\_TRAILER\_MAGIC 0xBAADF00D**

**//内存分配机制模式选择，环境变量key**

**#define GLUSTERFS\_ENV\_MEM\_ACCT\_STR "GLUSTERFS\_DISABLE\_MEM\_ACCT"**

**//1，使用gluster内存分配机制； 0，直接使用系统调用calloc分配内存**

**static int gf\_mem\_acct\_enable = 0;**

**/\*\***

**\*函数功能： 判断内存分配机制。**

**返回值： 1，使用acct内存分配机制(gluster封装的一个私有内存分配机制)**

**0，直接使用系统调用calloc分配内存**

**\*/**

**int gf\_mem\_acct\_is\_enabled()**

**{**

**return gf\_mem\_acct\_enable;**

**}**

**/\*\***

**\*函数功能： 读取环境变量"GLUSTERFS\_DISABLE\_MEM\_ACCT"，将环境变量第一个字符以十进制为单位转换为long变量，更新内存分配模式全局变量**

**\*/**

**void gf\_mem\_acct\_enable\_set()**

**{**

**char \*opt = NULL;**

**long val = -1;**

**#ifdef DEBUG**

**gf\_mem\_acct\_enable = 1;**

**return;**

**#endif**

**opt = getenv(GLUSTERFS\_ENV\_MEM\_ACCT\_STR);**

**if (!opt)**

**return;**

**val = strtol(opt, NULL, 0);**

**if (val)**

**gf\_mem\_acct\_enable = 0;**

**else**

**gf\_mem\_acct\_enable = 1;**

**}**

**/\*\***

**\*函数功能： 将alloc\_ptr内存区域格式化为acct对象**

**\*参数： alloc\_ptr，包含head tail 内存的指针**

**size，去掉head tail的内存size**

**type，内存的类型（每个xl->mem\_acct.rec[]下标）**

**Notes: 函数功能注释：**

**第0步： 更新转换器->acct内存对象->type类型对象->属性**

**内存格式化为acct对象的流程：**

**第1步：**

**alloc\_ptr**

**↓**

**-----------------------------------------**

**第2步：**

**alloc\_ptr**

**↓**

**-----------------------------------------**

**↑**

**ptr**

**第3步：**

**alloc\_ptr------→ptr = (d+8)**

**↓**

**--------------------------------------------**

**a b c d e f**

**↑ ↑ ↑ ↑ ↑ ↑**

**| | | | | ......ptr + size.....|**

**| | | | | |**

**| | | | | |**

**| | | | | 有效数据区域|**

**a:type--------------→-----| | | | | |**

**b:有效内存size------------→| | | | |**

**c:内存所属xlator\_t对象xl-→| | | |**

**d:GF\_MEM\_HEADER\_MAGIC--→-- | | |**

**e:8byte预留。--------------------------→-| |**

**f:GF\_MEM\_TRAILER\_MAGIC-----→----------------------------|**

**\*/**

**void**

**gf\_mem\_set\_acct\_info(xlator\_t \* xl, char \*\*alloc\_ptr,**

**size\_t size, uint32\_t type)**

**{**

**char \*ptr = NULL;**

**if (!alloc\_ptr)**

**return;**

**ptr = (char \*) (\*alloc\_ptr);**

**if (!xl) {**

**GF\_ASSERT(0);**

**}**

**if (!(xl->mem\_acct.rec)) {**

**GF\_ASSERT(0);**

**}**

**if (type > xl->mem\_acct.num\_types) {**

**GF\_ASSERT(0);**

**}**

**//更新转换器->acct内存对象->type类型对象->属性**

**LOCK(&xl->mem\_acct.rec[type].lock);**

**{**

**xl->mem\_acct.rec[type].size += size;**

**xl->mem\_acct.rec[type].num\_allocs++;**

**xl->mem\_acct.rec[type].max\_size = max(xl->mem\_acct.rec[type].max\_size,**

**xl->mem\_acct.rec[type].size);**

**xl->mem\_acct.rec[type].max\_num\_allocs = max(xl->mem\_acct.rec[type].max\_num\_allocs,**

**xl->mem\_acct.rec[type].num\_allocs);**

**}**

**UNLOCK(&xl->mem\_acct.rec[type].lock);**

**\*(uint32\_t \*) (ptr) = type;**

**ptr = ptr + 4;**

**memcpy(ptr, &size, sizeof(size\_t));**

**ptr += sizeof(size\_t);**

**memcpy(ptr, &xl, sizeof(xlator\_t \*));**

**ptr += sizeof(xlator\_t \*);**

**\*(uint32\_t \*) (ptr) = GF\_MEM\_HEADER\_MAGIC;**

**ptr = ptr + 4;**

**ptr = ptr + 8; //padding**

**\*(uint32\_t \*) (ptr + size) = GF\_MEM\_TRAILER\_MAGIC;**

**\*alloc\_ptr = (void \*) ptr;**

**return;**

**}**

**/\*\***

**\*函数功能： 调用calloc()，在内存动态存储区分配1个（cnt \* size + head + tail）字节的连续空间，函数返回一个指向**

**分配起始地址的指针；如果分配不成功，生成日志，返回NULL。将分配的内存格式化为acct对象**

**\*返回值： acct对象的有效内存区域的首地址**

**\*/**

**void \*\_\_gf\_calloc(size\_t nmemb, size\_t size, uint32\_t type)**

**{**

**size\_t tot\_size = 0;**

**size\_t req\_size = 0;**

**char \*ptr = NULL;**

**xlator\_t \*xl = NULL;**

**if (!gf\_mem\_acct\_enable)**

**return CALLOC(nmemb, size);**

**xl = THIS;**

**req\_size = nmemb \* size;**

**tot\_size = req\_size + GF\_MEM\_HEADER\_SIZE + GF\_MEM\_TRAILER\_SIZE;**

**ptr = calloc(1, tot\_size);**

**if (!ptr) {**

**gf\_log\_nomem("", GF\_LOG\_ALERT, tot\_size);**

**return NULL;**

**}**

**gf\_mem\_set\_acct\_info(xl, &ptr, req\_size, type);**

**return (void \*) ptr;**

**}**

**/\*\***

**\*函数功能： 调用malloc()，在内存动态存储区分配1个（size + head + tail）字节的连续空间，函数返回一个指向**

**分配起始地址的指针；如果分配不成功，生成日志，返回NULL。将分配的内存格式化为acct对象**

**\*返回值： acct对象的有效内存区域的首地址**

**\*/**

**void \*\_\_gf\_malloc(size\_t size, uint32\_t type)**

**{**

**size\_t tot\_size = 0;**

**char \*ptr = NULL;**

**xlator\_t \*xl = NULL;**

**if (!gf\_mem\_acct\_enable)**

**return MALLOC(size);**

**xl = THIS;**

**tot\_size = size + GF\_MEM\_HEADER\_SIZE + GF\_MEM\_TRAILER\_SIZE;**

**ptr = malloc(tot\_size);**

**if (!ptr) {**

**gf\_log\_nomem("", GF\_LOG\_ALERT, tot\_size);**

**return NULL;**

**}**

**gf\_mem\_set\_acct\_info(xl, &ptr, size, type);**

**return (void \*) ptr;**

**}**

**/\*\***

**\*函数功能： 调用realloc()，在内存动态存储区分配1个（size + head + tail）字节的连续空间，函数返回一个指向**

**分配起始地址的指针；如果分配不成功，生成日志，返回NULL。将分配的内存格式化为acct对象**

**\*返回值： acct对象的有效内存区域的首地址**

**\*/**

**void \*\_\_gf\_realloc(void \*ptr, size\_t size)**

**{**

**size\_t tot\_size = 0;**

**char \*orig\_ptr = NULL;**

**xlator\_t \*xl = NULL;**

**uint32\_t type = 0;**

**if (!gf\_mem\_acct\_enable)**

**return REALLOC(ptr, size);**

**tot\_size = size + GF\_MEM\_HEADER\_SIZE + GF\_MEM\_TRAILER\_SIZE;**

**orig\_ptr = (char \*) ptr - 8 - 4;**

**GF\_ASSERT(\*(uint32\_t \*) orig\_ptr == GF\_MEM\_HEADER\_MAGIC);**

**orig\_ptr = orig\_ptr - sizeof(xlator\_t \*);**

**xl = \*((xlator\_t \*\*) orig\_ptr);**

**orig\_ptr = (char \*) ptr - GF\_MEM\_HEADER\_SIZE;**

**type = \*(uint32\_t \*) orig\_ptr;**

**ptr = realloc(orig\_ptr, tot\_size);**

**if (!ptr) {**

**gf\_log\_nomem("", GF\_LOG\_ALERT, tot\_size);**

**return NULL;**

**}**

**gf\_mem\_set\_acct\_info(xl, (char \*\*) &ptr, size, type);**

**return (void \*) ptr;**

**}**

**/\*\***

**\*函数功能： 在当前转换器THIS下创建gf\_common\_mt\_asprintf类型的acct内存对象，向有效存储区域写入arg指向的数据流。string\_ptr**

**指向acct对象有效存储区域首地址**

**。**

**返回值： 写入成功的字节数**

**\*/**

**int gf\_vasprintf(char \*\*string\_ptr, const char \*format, va\_list arg)**

**{**

**va\_list arg\_save;**

**char \*str = NULL;**

**int size = 0;**

**int rv = 0;**

**if (!string\_ptr || !format)**

**return -1;**

**va\_copy(arg\_save, arg);**

**size = vsnprintf(NULL, 0, format, arg);**

**size++;**

**str = GF\_MALLOC(size, gf\_common\_mt\_asprintf);**

**if (str == NULL) {**

**/\* log is done in GF\_MALLOC itself \*/**

**return -1;**

**}**

**rv = vsnprintf(str, size, format, arg\_save);**

**\*string\_ptr = str;**

**return (rv);**

**}**

**/\*\***

**\*函数功能： 创建acct内存对象，数据流写入对象的有效存储区域。**

**\*返回值： string\_ptr指向acct对象的有效区域首地址**

**int，成功写入acct对象的字节数**

**\*/**

**int gf\_asprintf(char \*\*string\_ptr, const char \*format, ...)**

**{**

**va\_list arg;**

**int rv = 0;**

**va\_start(arg, format);**

**rv = gf\_vasprintf(string\_ptr, format, arg);**

**va\_end(arg);**

**return rv;**

**}**

**/\*\***

**\*函数功能： 释放参数指定的内存**

**\*/**

**void \_\_gf\_free(void \*free\_ptr)**

**{**

**size\_t req\_size = 0;**

**char \*ptr = NULL;**

**uint32\_t type = 0;**

**xlator\_t \*xl = NULL;**

**//若内存模式是直接系统api，则free(free\_ptr)完事儿，返回**

**if (!gf\_mem\_acct\_enable) {**

**FREE(free\_ptr);**

**return;**

**}**

**if (!free\_ptr)**

**return;**

**//检测free\_ptr是否符合 acct对象head魔法数？**

**ptr = (char \*) free\_ptr - 8 - 4;**

**if (GF\_MEM\_HEADER\_MAGIC != \*(uint32\_t \*) ptr) {**

**//Possible corruption, assert here**

**GF\_ASSERT(0);**

**}**

**\*(uint32\_t \*) ptr = 0;**

**//获取acct对象所属xl**

**ptr = ptr - sizeof(xlator\_t \*);**

**memcpy(&xl, ptr, sizeof(xlator\_t \*));**

**if (!xl) {**

**//gf\_free expects xl to be available**

**GF\_ASSERT(0);**

**}**

**if (!xl->mem\_acct.rec) {**

**ptr = (char \*) free\_ptr - GF\_MEM\_HEADER\_SIZE;**

**goto free;**

**}**

**//获取acct对象类型**

**ptr = ptr - sizeof(size\_t);**

**memcpy(&req\_size, ptr, sizeof(size\_t));**

**ptr = ptr - 4;**

**type = \*(uint32\_t \*) ptr;**

**//检索是否符合acct对象tail魔法数？**

**if (GF\_MEM\_TRAILER\_MAGIC != \*(uint32\_t \*)**

**((char \*) free\_ptr + req\_size)) {**

**// This points to a memory overrun**

**GF\_ASSERT(0);**

**}**

**\*(uint32\_t \*) ((char \*) free\_ptr + req\_size) = 0;**

**//减少xl下acct对象->对应类型的内存的size、num\_allocs属性**

**LOCK(&xl->mem\_acct.rec[type].lock);**

**{**

**xl->mem\_acct.rec[type].size -= req\_size;**

**xl->mem\_acct.rec[type].num\_allocs--;**

**}**

**UNLOCK(&xl->mem\_acct.rec[type].lock);**

**//释放内存**

**free:**

**FREE(ptr);**

**}**

**/\*\***

**\*函数功能： 创建参数指定的内存池对象，返回对象的指针**

**\*/**

**struct mem\_pool \*mem\_pool\_new\_fn(unsigned long sizeof\_type,**

**unsigned long count)**

**{**

**struct mem\_pool \*mem\_pool = NULL;**

**unsigned long padded\_sizeof\_type = 0;**

**void \*pool = NULL;**

**int i = 0;**

**struct list\_head \*list = NULL;**

**if (!sizeof\_type || !count) {**

**gf\_log("mem-pool", GF\_LOG\_ERROR, "invalid argument");**

**return NULL;**

**}**

**padded\_sizeof\_type = sizeof\_type + GF\_MEM\_POOL\_PAD\_BOUNDARY; //数据类型size + 12(struct list + int)**

**//1.分配一个内存池对象、初始化锁、链表、属性**

**mem\_pool = GF\_CALLOC(sizeof(\*mem\_pool), 1, gf\_common\_mt\_mem\_pool);**

**if (!mem\_pool)**

**return NULL;**

**LOCK\_INIT(&mem\_pool->lock);**

**INIT\_LIST\_HEAD(&mem\_pool->list);**

**mem\_pool->padded\_sizeof\_type = padded\_sizeof\_type;**

**mem\_pool->cold\_count = count;**

**mem\_pool->real\_sizeof\_type = sizeof\_type;**

**//2.分配count \* padded\_sizeof\_type个字节的acct对象，返回有效存储区域**

**pool = GF\_CALLOC(count, padded\_sizeof\_type, gf\_common\_mt\_long);**

**if (!pool) {**

**GF\_FREE(mem\_pool);**

**return NULL;**

**}**

**//3.将第2步创建的acct对象有效内存，以链表形式插入第1步创建的内存池对象链表尾**

**for (i = 0; i < count; i++) {**

**list = pool + (i \* (padded\_sizeof\_type));**

**INIT\_LIST\_HEAD(list);**

**list\_add\_tail(list, &mem\_pool->list);**

**}**

**//4.设置内存池对象数据有效范围**

**mem\_pool->pool = pool;**

**mem\_pool->pool\_end = pool + (count \* (padded\_sizeof\_type));**

**//5.返回内存池对象**

**return mem\_pool;**

**}**

**/\*\***

**\*函数功能： 从指定内存池获取一片内存，清0后返回这片内存的首地址**

**\*/**

**void \*mem\_get0(struct mem\_pool \*mem\_pool)**

**{**

**void \*ptr = NULL;**

**if (!mem\_pool) {**

**gf\_log("mem-pool", GF\_LOG\_ERROR, "invalid argument");**

**return NULL;**

**}**

**ptr = mem\_get(mem\_pool);**

**if (ptr)**

**memset(ptr, 0, mem\_pool->real\_sizeof\_type);**

**return ptr;**

**}**

**/\*\***

**\*函数功能： 从指定内存池获取一片内存，更新内存池中冷热内存片的数量**

**\*/**

**void \*mem\_get(struct mem\_pool \*mem\_pool)**

**{**

**struct list\_head \*list = NULL;**

**void \*ptr = NULL;**

**int \*in\_use = NULL;**

**if (!mem\_pool) {**

**gf\_log("mem-pool", GF\_LOG\_ERROR, "invalid argument");**

**return NULL;**

**}**

**LOCK(&mem\_pool->lock);**

**{**

**if (mem\_pool->cold\_count) {**

**list = mem\_pool->list.next;**

**list\_del(list);**

**mem\_pool->hot\_count++;**

**mem\_pool->cold\_count--;**

**ptr = list;**

**in\_use = (ptr + GF\_MEM\_POOL\_LIST\_BOUNDARY);**

**\*in\_use = 1;**

**goto fwd\_addr\_out;**

**}**

**/\* This is a problem area. If we've run out of**

**\* chunks in our slab above, we need to allocate**

**\* enough memory to service this request.**

**\* The problem is, these indvidual chunks will fail**

**\* the first address range check in \_\_is\_member. Now, since**

**\* we're not allocating a full second slab, we wont have**

**\* enough info perform the range check in \_\_is\_member.**

**\***

**\* I am working around this by performing a regular allocation**

**\* , just the way the caller would've done when not using the**

**\* mem-pool. That also means, we're not padding the size with**

**\* the list\_head structure because, this will not be added to**

**\* the list of chunks that belong to the mem-pool allocated**

**\* initially.**

**\***

**\* This is the best we can do without adding functionality for**

**\* managing multiple slabs. That does not interest us at present**

**\* because it is too much work knowing that a better slab**

**\* allocator is coming RSN.**

**\*/**

**ptr = MALLOC(mem\_pool->real\_sizeof\_type);**

**/\* Memory coming from the heap need not be transformed from a**

**\* chunkhead to a usable pointer since it is not coming from**

**\* the pool.**

**\*/**

**goto unlocked\_out;**

**}**

**fwd\_addr\_out:**

**ptr = mem\_pool\_chunkhead2ptr(ptr);**

**unlocked\_out:**

**UNLOCK(&mem\_pool->lock);**

**return ptr;**

**}**

**/\*\***

**\*函数功能： 判断ptr所指内存，是否在pool内存池范围内**

**\*/**

**static int \_\_is\_member(struct mem\_pool \*pool, void \*ptr)**

**{**

**if (!pool || !ptr) {**

**gf\_log("mem-pool", GF\_LOG\_ERROR, "invalid argument");**

**return -1;**

**}**

**if (ptr < pool->pool || ptr >= pool->pool\_end)**

**return 0;**

**if ((mem\_pool\_ptr2chunkhead(ptr) - pool->pool)**

**% pool->padded\_sizeof\_type)**

**return -1;**

**return 1;**

**}**

**/\*\***

**\*函数功能： 如果ptr指针在pool范围内，则回收。否则不处理。**

**\*参数： pool，目标内存池**

**ptr，acct对象->内存对象有效数据区首地址(ptr - 12 等于内存对象的链表地址)**

**\*/**

**void mem\_put(struct mem\_pool \*pool, void \*ptr)**

**{**

**struct list\_head \*list = NULL;**

**int \*in\_use = NULL;**

**void \*head = NULL;**

**if (!pool || !ptr) {**

**gf\_log("mem-pool", GF\_LOG\_ERROR, "invalid argument");**

**return;**

**}**

**LOCK(&pool->lock);**

**{**

**switch (\_\_is\_member(pool, ptr)) {**

**case 1:**

**list = head = mem\_pool\_ptr2chunkhead(ptr);**

**in\_use = (head + GF\_MEM\_POOL\_LIST\_BOUNDARY);**

**if (!is\_mem\_chunk\_in\_use(in\_use)) {**

**gf\_log\_callingfn("mem-pool",**

**GF\_LOG\_CRITICAL,**

**"mem\_put called on freed ptr %p of mem "**

**"pool %p", ptr, pool);**

**break;**

**}**

**pool->hot\_count--;**

**pool->cold\_count++;**

**\*in\_use = 0;**

**list\_add(list, &pool->list);**

**break;**

**case -1:**

**/\* For some reason, the address given is within**

**\* the address range of the mem-pool but does not align**

**\* with the expected start of a chunk that includes**

**\* the list headers also. Sounds like a problem in**

**\* layers of clouds up above us. ;)**

**\*/**

**abort();**

**break;**

**case 0:**

**/\* The address is outside the range of the mem-pool. We**

**\* assume here that this address was allocated at a**

**\* point when the mem-pool was out of chunks in mem\_get**

**\* or the programmer has made a mistake by calling the**

**\* wrong de-allocation interface. We do**

**\* not have enough info to distinguish between the two**

**\* situations.**

**\*/**

**FREE(ptr);**

**break;**

**default:**

**/\* log error \*/**

**break;**

**}**

**}**

**UNLOCK(&pool->lock);**

**}**

**/\*\***

**\*函数功能： 销毁指定内存池对象**

**\*/**

**void mem\_pool\_destroy(struct mem\_pool \*pool)**

**{**

**if (!pool)**

**return;**

**LOCK\_DESTROY(&pool->lock);**

**GF\_FREE(pool->pool);**

**GF\_FREE(pool);**

**return;**

**}**

**Gluster的内存池模型叫acct，**

**如果读者再添加代码时习惯直接系统调用，则用下面三个宏**

**//内存分配宏，内联函数封装系统调用、日志**

**#define MALLOC(size) \_\_gf\_default\_malloc(size)**

**#define CALLOC(cnt,size) \_\_gf\_default\_calloc(cnt,size)**

**#define REALLOC(ptr,size) \_\_gf\_default\_realloc(ptr,size)**

**如果想用gluster封装出来的内存池，则用下面四个宏**

**//acct内存分配宏，gluster详细封装了系统调用、日志**

**#define GF\_CALLOC(nmemb, size, type) \_\_gf\_calloc (nmemb, size, type)**

**#define GF\_MALLOC(size, type) \_\_gf\_malloc (size, type)**

**#define GF\_REALLOC(ptr, size) \_\_gf\_realloc (ptr, size)**

**#define GF\_FREE(free\_ptr) \_\_gf\_free (free\_ptr);**

**第十章**

----------------------------加密算法

Gluster最开始的加密方式是rot-13，现在比较新的加密方式也是一样的道理。只不过加密的方式更复杂，加密解密所需的时间更多。

因为比较简单，rot-13.h只定义了模式结构，并未专门定义加密流程管控结构类。

#ifndef \_\_ROT\_13\_H\_\_

#define \_\_ROT\_13\_H\_\_

#ifndef \_CONFIG\_H

#define \_CONFIG\_H

#include "config.h"

#endif

//rot13模式类

typedef struct {

gf\_boolean\_t encrypt\_write; //加密写

gf\_boolean\_t decrypt\_read; //解密读

} rot\_13\_private\_t;

#endif /\* \_\_ROT\_13\_H\_\_ \*/

rot-13.c文件

/\*\*

\*函数功能： rot13加密/解密函数，将buf中的英文字母，转换为对应偏移13位的字母

由于英文字母是26位，rot13是13位偏移，因此调用一次是加密数据，再调用一次解密数据

\*/

void rot13(char \*buf, int len)

{

int i;

for (i = 0; i < len; i++) {

if (buf[i] >= 'a' && buf[i] <= 'z')

buf[i] = 'a' + ((buf[i] - 'a' + 13) % 26);

else if (buf[i] >= 'A' && buf[i] <= 'Z')

buf[i] = 'A' + ((buf[i] - 'A' + 13) % 26);

}

}

//rot13模块支持功能查询表

struct volume\_options options[] = {

{.key = {"encrypt-write"},

.type = GF\_OPTION\_TYPE\_BOOL},

{.key = {"decrypt-read"},

.type = GF\_OPTION\_TYPE\_BOOL},

{.key = {NULL}},

};

核心的东西就是这些，嵌入到gluster集群软件，成文模块当然还会将以上过程封装成api，分别对应加密和解密。

注意options结构数组！每一个转换器模块都会有这样一个定义，在模块被加载后具体实例化全局管控对象，也就是说全局对象的接口函数是标准的，具体干事的这些模块功能不一样，根据options来知道这个模块支持的功能。就其这个作用。

其它的加密算法与此类似。如md5算法：

在gluster源码中有md5.c、md5.h两个文件，md5加密算法有两组API

md5\_begin();

md5\_update();

md5\_result();

这三个函数是一组api

get\_md5()是一个封装性更紧密的api，实际上内部调用的也是上面三个函数。

**第十一章**

----------------------------鉴权模块

Rpc在调用时，在write写服务上需要鉴权。

在比对rpc服务号、过程号、版本号之后就需要调用鉴权。

Gluster的鉴权有两种addr、login两种:

addr是一个卷配置文件当中，对于允许操作的客户端的ip和端口列表，也就是说没有经过rpc握手直接去调用卷文件是不能通过的。

auth\_result\_t gf\_auth(dict\_t \* input\_params, dict\_t \* config\_params)

Loging是账号密码列表，同样存在与一个卷的配置文件当中。

auth\_result\_t gf\_auth(dict\_t \* input\_params, dict\_t \* config\_params);

这两个鉴权都比较简单，函数接口是一样的，编译成模块。通过更上层来确定rpc客户端发过来的消息头，来确定具体用哪一个鉴权模块来实例化，读取配置文件和输入参数来确定鉴权结果：

typedef enum {

AUTH\_ACCEPT, //鉴权通过

AUTH\_REJECT, //鉴权拒绝

AUTH\_DONT\_CARE //服务器上调用该业务不需要鉴权

} auth\_result\_t;

**第十二章**

-----------------------五种调度器

scheduler/ 本文件夹是调度模块，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：

五种调度器实现了unify 翻译器。分别为: （1）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。

（2）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。

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

（4）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，响应速度等等，如有 4 台性能和磁盘容量不一样的服务器，调度序列可能就是ABCDABCDABCDABCD ……

（5）switch

五 种基本模块构成了GlusterFS的调度器，另外还提供了一些扩展。

五种调度方式，仅分析过其中一种：alu.c、alu.h，其它的调度方式在于计算的因素不同，源码框架一样。到第12章篇幅已经比较冗余，在没有规划好每个章节的情况下源码会显得比较冗余，因此Alu调度算法详细注释在源码中。

**第十三章**

----------------------转换器模块

哈希表在之前介绍的数据结构章节有较为详细的描述，介绍本章内容之前引出一个概念：

弹性哈希算法，有这样一个迭代函数：输入任意长度的文件名，返回一个int类型的对应值，只要文件名完全一直，返回的int值就唯一。文件名稍有不同，返回的int值就会有非常大的差距。

利用哈希算法的这一特性，可以确定以下功能的合理性：任意客户端都知道这个哈希算法、给任何客户端同样的文件名，算出来的int值都是相等的。比如一个集群有4个节点：

第一个节点分配哈希段为：0 ~ 1G

第二个节点分配哈希段为：1G ~ 2G

第二个节点分配哈希段为：2G ~ 3G

第二个节点分配哈希段为：3G ~ 4G

现在客户端有6个文件，则它们有如下关系：

文件名 哈希值 文件实际存储在的节点

file1 100 node1

file2 1G + 100 node2

file3 2G + 300 node3

file4 1G -1000 node1

file5 3G + 5000 node4

file6 4G -200 node4

如此，一个客户端根据文件名，就会计算出一个文件应该在哪一个节点上（文件名与所在节点是一一对应关系）。然后通过rpc远程调用，去访问该节点。节点本身有目录管理功能（vfs），因此最终找到了目标文件，是open、lseek、read、write、close，就看gluster客户端请求对文件进行何种操作。

之前讲过《西游降魔》，引出rpc模型，进而引出nfs是一个特殊化的分布式文件系统（单个节点作为服务器），此处引出弹性哈希算法，进一步弥完善了分布式文件系统的核心理论支撑。

Translators是GlusterFS提供的一种强大文件系统功能扩展机制，这一设计思想借鉴于GNU/Hurd微内核操作系统。GlusterFS中所有的功能都通过Translator机制实现，运行时以动态库方式进行加载，服务端和客户端相互兼容。GlusterFS 3.2中，主要包括以下几类Translator：

（1）  **Cluster**：存储集群分布，目前有AFR, DHT, Stripe三种方式

（2）  **Debug**：跟踪GlusterFS内部函数和系统调用

（3）  **Encryption**：简单的数据加密实现，也就是前面章节介绍的rot-13

（4）  **Features**：访问控制、锁、Mac os兼容、静默、配额、只读、回收站等

（5）  **Mgmt**：弹性卷管理

（6）  **Mount**：FUSE接口实现

（7）  **Nfs**：内部NFS服务器

（8）  **Performance**：io-cache, io-threads, quick-read, read-ahead, stat-prefetch, sysmlink-cache, write-behind等性能优化

（9）  **Protocol**：服务器和客户端协议实现

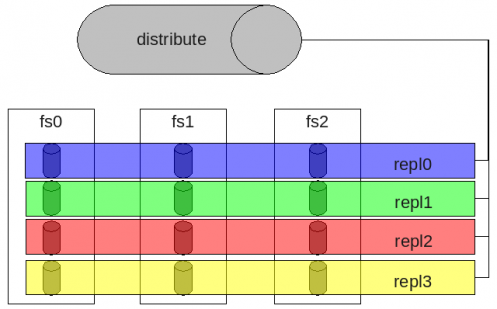
（10）**Storage**：底层文件系统POSIX接口实现

本章我们重点分析一下Cluster Translators，它是除rpc模型以外的另一个重要模型，是整个GlusterFS集群存储的核心，它包括AFR（Automatic File Replication）、DHT（Distributed Hash Table）和Stripe三种类型。

AFR相当于RAID1，同一文件在多个存储节点上保留多份，主要用于实现高可用性以及数据自动修复。AFR所有子卷上具有相同的名字空间，查找文件时从第一个节点开始，直到搜索成功或最后节点搜索完毕。读数据时，AFR会把所有请求调度到所有存储节点，进行负载均衡以提高系统性能。写数据时，首先需要在所有锁服务器上对文件加锁，默认第一个节点为锁服务器，可以指定多个。然后，AFR以日志事件方式对所有服务器进行写数据操作，成功后删除日志并解锁。AFR会自动检测并修复同一文件的数据不一致性，它判断change log来确定一个数据副本是否未被破坏。自动修复在文件目录首次访问时触发，如果是目录将在所有子卷上复制正确数据，如果文件不存则创建，文件信息不匹配则修复，日志指示更新则进行更新。

DHT即上面所介绍的弹性哈希算法，它采用hash方式进行数据分布，名字空间分布在所有节点上。查找文件时，通过弹性哈希算法进行，不依赖名字空间。但遍历文件目录时，则实现较为复杂和低效，需要搜索所有的存储节点。单一文件只会调度到唯一的存储节点，一旦文件被定位后，读写模式相对简单。DHT不具备容错能力，需要借助AFR实现高可用性, 如下图所示应用案例。

Stripe相当于RAID0，即分片存储，文件被划分成固定长度的数据分片以Round-Robin轮转方式存储在所有存储节点。Stripe所有存储节点组成完整的名字空间，查找文件时需要询问所有节点，这点非常低效。读写数据时，Stripe涉及全部分片存储节点，操作可以在多个节点之间并发执行，性能非常高。Stripe通常与AFR组合使用，构成RAID10/RAID01，同时获得高性能和高可用性，当然存储利用率会低于50%。（因此，Glustr做成大规模的商业产品，需要添加上纠删功能。既然提到了，本章是将转换器的，纠删不宜喧宾夺主，本章后面再加一章专门讲纠删功能吧）。



**GlusterFS应用案例：AFR+DHT**

1. 设计讨论

GlusterFS是一个具有高扩展性、高性能、高可用性、可横向扩展的弹性分布式文件系统，在架构设计上非常有特点，比如无元数据服务器设计（上面有一段文字已经描述清晰为什么不需要元数据服务器）、堆栈式架构等。然而，存储应用问题是很复杂的，GlusterFS也不可能满足所有的存储需求，设计实现上也一定有考虑不足之处，下面我们作简要分析。

l  **无元数据服务器 vs 元数据服务器**

无元数据服务器设计的好处是没有单点故障和性能瓶颈问题，可提高系统扩展性、性能、可靠性和稳定性。对于海量小文件应用，这种设计能够有效解决元数据的难点问题（不管多少海量小文件，哈希算法话费的时间永远是O(1)，而依赖元数据服务器的分布式则不同，更多的元数据服务器就会降低元数据服务区的性能）。

它的负面影响是，数据一致问题更加复杂，文件目录遍历操作效率低下，缺乏全局监控管理功能。同时也导致客户端承担了更多的职能，比如文件定位、名字空间缓存、逻辑卷视图维护等等，这些都增加了客户端的负载，占用相当的CPU和内存。

l  **用户空间 vs 内核空间文件系统**

用户空间实现起来相对要简单许多，对开发者技能要求较低，运行相对安全。用户空间效率低，数据需要多次与内核空间交换，另外GlusterFS借助FUSE来实现标准文件系统接口，性能上又有所损耗。内核空间实现可以获得很高的数据吞吐量，缺点是实现和调试非常困难，程序出错经常会导致系统崩溃，安全性低。纵向扩展上，内核空间要优于用户空间，GlusterFS有横向扩展能力来弥补。

Ceph是内核空间的文件系统（文件系统运行在3G~4G虚拟空间），gluster是fuse实现的用户空间文件系统（文件系统运行在0~3G虚拟空间）。

l  **堆栈式 vs 非堆栈式**

这有点像操作系统的微内核设计与单一内核设计之争。GlusterFS堆栈式设计思想源自GNU/Hurd微内核操作系统，具有很强的系统扩展能力，系统设计实现复杂性降低很多，基本功能模块的堆栈式组合就可以实现强大的功能。查看GlusterFS卷配置文件我们可以发现，translator功能树通常深达10层以上，一层一层进行调用，效率可见一斑。

非堆栈式设计可看成类似Linux的单一内核设计，系统调用通过中断实现，非常高效。后者的问题是系统核心臃肿，实现和扩展复杂，出现问题调试困难。

l  **原始存储格式 vs 私有存储格式**

GlusterFS使用原始格式存储文件或数据分片，可以直接使用各种标准的工具进行访问，数据互操作性好，迁移和数据管理非常方便。然而，数据安全成了问题，因为数据是以平凡的方式保存的，接触数据的人可以直接复制和查看。这对很多应用显然是不能接受的，比如云存储系统，用户特别关心数据安全，这也是影响公有云存储发展的一个重要原因。私有存储格式可以保证数据的安全性，即使泄露也是不可知的。GlusterFS要实现自己的私有格式，在设计实现和数据管理上相对复杂一些，也会对性能产生一定影响。

l  **大文件 vs 小文件**

GlusterFS适合大文件还是小文件存储？弹性哈希算法和Stripe数据分布策略，移除了元数据依赖，优化了数据分布，提高数据访问并行性，能够大幅提高大文件存储的性能。对于小文件，无元数据服务设计解决了元数据的问题。但GlusterFS并没有在I/O方面作优化，在存储服务器底层文件系统上仍然是大量小文件，本地文件系统元数据访问是一个瓶颈，数据分布和并行性也无法充分发挥作用。因此，GlusterFS适合存储大文件，小文件性能较差，还存在很大优化空间。

l  **可用性 vs 存储利用率**

GlusterFS使用复制技术来提供数据高可用性，复制数量没有限制，自动修复功能基于复制来实现。可用性与存储利用率是一个矛盾体，可用性高存储利用率就低，反之亦然。采用复制技术，存储利用率为1/复制数，镜像是50%，三路复制则只有33%。其实，可以有方法来同时提高可用性和存储利用率，比如RAID5的利用率是(n-1)/n，RAID6是(n-2)/n，而纠删码技术可以提供更高的存储利用率。但是，鱼和熊掌不可得兼，它们都会对性能产生较大影响。

在转换器源码路径下，我添加了一个null的转换器模块。没有写具体实现的功能，只是完成了框架搭建。新功能的添加（如纠删、用户审计），建议使用这个框架去实现。

由于前面分析了gluster的内存分配模型，为本处分析转换器的源码框架打下基础（也就是说如果没有对内存模型的详细分析，转换器的代码会非常难以看懂）。

下面用一个图来分布式存储模型：

alu.so

log.so

......

auth.so

encryption.so

posix-acl.so

Nfs-server.so

meta.so

posix.so

bdb.so

io-cache.so

fuse.so

io-threads.so

quto.so

Handler（使用帧、栈、内存池实现层级关系和传递关系），

Xlator（生长在红黑树上）通过wind()宏实现逻辑分级：业务单独、接口统一

Rpc服务器/客户端

Libglusterfs 基于fuse文件系统

fuse

vfs

控制台输入

Local filesystem

Gluster即cli模型

配置文件

rpc服务是分布式文件系统的核心，由上图可知转换器是rpc服务的统一接口，具体服务功能有不同的模块去实现。也就是说转换器起到的是支架的作用，rpc服务通过支架传输，再往下级传输，直到最后的业务实现函数。

转换器所处的位置就决定了它需要实现以下功能：

1. 消息准确传递
2. 消息执行结果回传
3. 忽略个模块之间的差异，实现统一接口

经过分析我们认为转换器应该这么去设计，那么我们去看源码发现确实如此：

稍后会拉出来一个讲转换器去讲一些具体如何实现一个！所谓工欲善其事，必先利其器。

现在我们去分析一下这个器。（任何能称得上模型的源码，都是有理论支撑的）

说白了就是一种可以准确传递消息的数据结构。

首先我们看三个结构体：

以下三组是gluster的惯用伎俩，用于先引用后定义。

struct \_call\_stack\_t;

typedef struct \_call\_stack\_t call\_stack\_t;

struct \_call\_frame\_t;

typedef struct \_call\_frame\_t call\_frame\_t;

struct \_call\_pool\_t;

typedef struct \_call\_pool\_t call\_pool\_t;

//帧、栈对象的内存池：0级

struct \_call\_pool\_t {

union {

struct list\_head all\_frames; //调用栈链表头

struct {

call\_stack\_t \*next\_call;

call\_stack\_t \*prev\_call;

} all\_stacks;

};

int64\_t cnt; //调用栈对象的数量

gf\_lock\_t lock; //all\_frames链表的保护锁

struct mem\_pool \*frame\_mem\_pool; // 用于帧对象调用 的内存池对象指针

struct mem\_pool \*stack\_mem\_pool; // 用于栈对象调用 的内存池对象指针

};

//栈对象：1级

struct \_call\_stack\_t {

union {

struct list\_head all\_frames; //调用栈对象的链表节点

struct {

call\_stack\_t \*next\_call;

call\_stack\_t \*prev\_call;

};

};

call\_pool\_t \*pool; //本栈对象所在的内存池

gf\_lock\_t stack\_lock; //栈锁

void \*trans;

uint64\_t unique;

void \*state; /\* pointer to request state \*/

uid\_t uid;

gid\_t gid;

pid\_t pid;

uint32\_t ngrps;

uint32\_t groups[GF\_REQUEST\_MAXGROUPS];

uint64\_t lk\_owner;

call\_frame\_t frames; //帧对象

int32\_t op; //fops操作名称gf\_fop\_list[]数组下标，可索引到本栈的文件操作对应的字符串

int8\_t type; //本栈的fops类型：GF\_OP\_TYPE\_FOP 或 GF\_OP\_TYPE\_MGMT

};

//帧对象：2级

struct \_call\_frame\_t {

call\_stack\_t \*root; /\* stack root 所属root栈对象指针\*/

call\_frame\_t \*parent; /\* previous BP 本帧的直接上级帧\*/

call\_frame\_t \*next;

call\_frame\_t \*prev; /\* maintenance list 调用帧链\*/

void \*local; /\* local variables 本帧私有的对象指针，卷数据结构cli\_local\_t、xl\_marker\_local\_t\*/

xlator\_t \*this; /\* implicit object 本帧所属转换器对象\*/

ret\_fn\_t ret; /\* op\_return address \*/

int32\_t ref\_count;

gf\_lock\_t lock; //本对象的保护锁

void \*cookie; /\* unique cookie 一般执行本帧对象\*/

gf\_boolean\_t complete;

glusterfs\_fop\_t op; //枚举体，帧对象的fops操作文件类型； -1，则无文件操作类型

struct timeval begin; /\* when this frame was created 帧对象的创建时间点\*/

struct timeval end; /\* when this frame completed 帧对象执行完成的时间点\*/

const char \*wind\_from; //调用宏将消息传递给本帧对象的那一行，所在的函数

const char \*wind\_to; //指定转换器要执行的函数

const char \*unwind\_from;//

const char \*unwind\_to; //回调函数，当下一个translator完成时会调用该函数

};

其中0级、1级、2级、是指的抽象层级，数字越小等级越高。自上而下，层层包含。

struct \_call\_pool\_t 是0级，包含了活动的栈（1级）链表、堆对象内存池、栈对象内存池，简单来说下面的所有堆对象、栈对象都是从此结构中的对应内存池构建和析构。

struct \_call\_stack\_t是1级，包含了帧对象（2级）、帧对象的fops类型等。

struct \_call\_frame\_t 是2级，fops类型、帧消息传递方向。传递消息时的执行函数、传递完成后的回调函数。

相应的，在stakc.h中封装了如上三个对象类的构建、析构函数组：

/\*\*

\*内联函数： 回收目标帧对象 到 目标帧对象所在内存池

\*/

static inline void FRAME\_DESTROY(call\_frame\_t \* frame)

/\*\*

\*内联函数： 回收目标栈对象 到 目标栈对象所在内存池

\*/

static inline void STACK\_DESTROY(call\_stack\_t \* stack)

/\*\*

\*函数功能： 从frame对象所属栈对象所在的内存池中，新创建一个栈对象。赋值frame对象所属栈对象的内容 到 新的栈对象

新的栈对象插入frame对象所属栈对象链表

返回值： 返回新的栈对象的指针

\*/

static inline call\_frame\_t \*copy\_frame(call\_frame\_t \* frame)

/\*\*

\*内联函数： 从指定的调用池的->栈池创建栈对象（新创建的栈对象已包含了帧对象），设置帧对象的所属root、所属转换器，

将栈对象插入指定内存池的栈链表头，初始化帧锁、栈锁

返回值： 返回栈对象的指针

\*/

static inline call\_frame\_t \*create\_frame(xlator\_t \* xl, call\_pool\_t \* pool)

以及消息传递宏：

/\*\*

make a call

宏功能： 当xlator 某个fops函数被调用，表示接受到一个请求，使用frame stack结构来表示。fops函数中执行相应操作，然后可把该请求使用

STACK\_WIND()宏传递给next 或 nexts translator

创建帧对象，填充层级关系。调用当前转换器的fops中的fn函数;注册回调函数rfn到新创建的帧对象，将该对象插入跟栈对象链表。

宏参数： frame： 表示请求帧对象的指针

rfn： 回调函数，当下一个translator完成时会调用该函数

obj： 正在控制的translator对象

fn： 从当前translator的fops table中指定要执行的函数

params：任何其他被调用函数的参数（比如，inodes, fd, offset, data buffer）

\*/

#define STACK\_WIND(frame, rfn, obj, fn, params ...)

/\* make a call with a cookie

宏定功能： 新创建帧对象，注册rfn回调函数到帧对象。将帧对象插入frame参数的root栈的帧对象链表。执行fn函数

宏参数： frame： 表示请求帧对象的指针

rfn： 回调函数，当下一个translator完成时会调用该函数

cky： cookie,这是一个opaque指针

obj： 正在控制的translator对象

fn： 从当前translator的fops table中指定要调用的translator函数

params：任何其他被调用函数的参数（比如，inodes, fd, offset, data buffer）

\*/

#define STACK\_WIND\_COOKIE(frame, rfn, cky, obj, fn, params ...)

/\*\*

return from function

宏功能： 当完成一个请求而不再需要调用下一个translator，或者当任务完成从回调函数中回到上一个translator时需要调用STACK\_UNWIND

宏参数： frame： 表示请求帧对象的指针

params：任何其他被调用函数的参数（比如，inodes, fd, offset, data buffer）

\*/

#define STACK\_UNWIND(frame, params ...)

/\*\*

return from function in type-safe way

宏功能： 根据op类型和帧指针，找到对应的fops函数指针，传入params参数执行该函数

宏参数：

op： frame对象期待业务的操作类型(lookup, stat, truncate)，用来在宏中创建对应的fops类型指针来索引frame->ret

frame： 表示请求帧对象的指针

params：任何其他被调用函数的参数（比如，inodes, fd, offset, data buffer）

\*/

#define STACK\_UNWIND\_STRICT(op, frame, params ...)

转换器通过上述宏和对象操作，来实现具体的fops调用，如：lseek()、lseek\_cbk()、

Truncate()、truncate\_cbk()、read()、read\_cbl()、......

每个转换有以下部分组成：

int32\_t init (xlator\_t \*this)；构造函数

Void fini (xlator\_t \*this)；析构函数

转换器业务手柄

struct xlator\_fops fops = {

.lookup = null\_lookup,

.stat = null\_stat,

.fstat = null\_fstat,

.truncate = null\_truncate,

.ftruncate = null\_ftruncate,

.access = null\_access,

......

}

转换器回调函数手柄

struct xlator\_cbks cbks={

.forget = null\_forget,

.release = null\_release,

.releasedir = null\_releasedir,

};

转换器模块字典选项

struct volume\_options options[] = {

{.key = {NULL}

},

};

转换器目录下，对应的就有xx\_fops.c、xx\_fops.h去实现手柄。

在libglusterfs/ 下面有两个文件defaults.c、defaults.h去实现默认的fops操作，也就是说一个转换器有私有的共有部分，共有部分即是defaults源码实现的部分。转换器可以不使用任何一个共有部分。

**第十四章**

----------------------制作gluster安装包

制作rpm安装包有四个步骤：

1. 将gluster源码编译成为rpm包

cd gluster/

./autogen.sh

./configure

make dist

cd extras/LinuxRPM

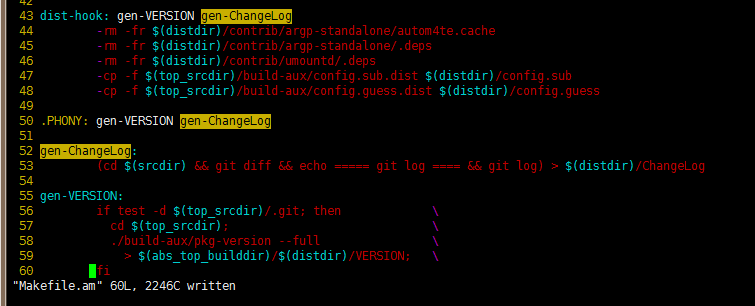
make glusterrpms

编译成功后ll extras/LinuxRPM 会看到如下的rpm包已经成功编译。

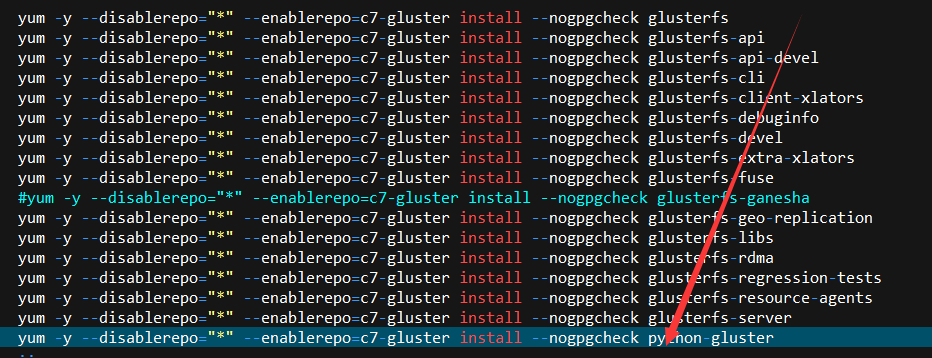


以上是从git获取的源码的编译过程。从gluster获得源码需要修改后才能编译成功：

将gluster源码中顶层的Makefile.am文件中，高亮部分去掉，即可正常编译通过。



1. 将依赖文件的rpm包全部缓存起来
2. 将1、2步骤的rpm做成本地yum源，由于是同一个Yum源，在安装gluster rpm软件时，会自动解决依赖关系。Gluster是一组软件，选择需要的进行安装。



1. 编写shell脚本，完成上述流程。

在gluster的发行包中的安装脚本，都有比较详细的注释。上述只是简单描述其整个过程。