**56运维二组网络框架二次开发说明**

**版本<1.0>**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本号** | **修订内容** | **修订人** |
| 20130812 | 1.0 | 创建文档 | 吕丁勤 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. **为什么需要框架开发**

我们运维二组目前的情况看来，接收到的任务比较多，如果没有一套规范的开发框架，每次都是按个人编码习惯从新编码，无疑会添加了自己的工作量以及后期维护多个系统的工作量，所以为了减轻大家的工作量，通过本文档，大家只要花一点时间学习一下从VFS抽取出来的网络框架，利用新框架进行开发，以后遇到新需求或者维护就能轻松一点，框架开发有以下优点：

1. 规范项目的代码结构，便于维护
2. 剥离公共部分，便于进行快速开发
3. **框架提供的功能**
4. 提供统一的日志系统
5. 提供统一的配置文件处理
6. 提供统一的网络句柄管理
7. 提供统一的网络前端事件处理
8. **插件开发说明**

插件和框架之间交互的数据结构 在global.h 定义：struct conn 该数据结构跟对应客户端网络fd关联。（初始化分配，避免运行时分配释放内存）

该数据结构有一个成员变量 user，是插件自己实现的关联数据地址，用于保存插件自己需要保存的数据结构，需要插件自己申请并释放内存。

**日志记录API**

//初始化日志线程

int init\_log();

//注册日志，返回日志fd

int registerlog(const char\* filename, int level, int rotatesize, int rotateintval, int filenum);

//打印日志到fd

void LOG(int fd, int level, const char\* fmt, ...);

//关闭日志fd

void logclose(int fd);

**配置文件读取API**

//初始化配置文件,读取配置信息到内存

int myconfig\_init(int argc, char \*\*argv);

//重新加载配置文件到内存，调用注册的回调函数

void myconfig\_reload();

//注册配置文件重新加载时调用函数

myconfig\_register\_reload(int (\*reload\_cb\_func)(void), char\*\* keys, int keynum);

//返回字符串型值

char\* myconfig\_get\_value(const char \*key);

//有多个配置值时，指定返回第index个字符串型值

char\* myconfig\_get\_multivalue(const char \*key, int index);

//返回数值型值，如果没有配置则返回def

int myconfig\_get\_intval(const char \*key, int def);

**框架提供给插件的函数介绍**

1. void add\_fd\_2\_efd(int fd); 插件主动发起一个连接后，把该句柄加到epoll监听事件。
2. void do\_close(int fd); 插件主动关闭一个网络句柄时调用。
3. void modify\_fd\_event(int fd, int events); 修改某个句柄网络监听事件类型。
4. int get\_client\_data(int fd, char \*\*data, uint32\_t \*len); 获取到该句柄关联的输入缓冲区的数据，成功返回0， len指示数据长度，data是数据地址
5. int consume\_client\_data(int fd, uint32\_t len); 插件处理一次请求后，告诉框架忽略掉的数据长度。
6. int set\_client\_data(int fd, char \*buf, uint32\_t len); 插件处理请求返回时，调用该函数通知框架copy数据到输出缓冲区，在下个网络事件来临时，会发送该数据。

int set\_client\_fd(int fd, int lfd, uint32\_t offset, uint32\_t len); 当需要通过sendfile传输文件时，调用该函数，fd是网络句柄，lfd是本地文件句柄，offset是起始位，len是发送长度。

**插件必需实现函数介绍**

A：int svc\_init() 该函数是插件全局初始化函数，只调用一次。（通常用来初始化插件日志，插件自有全局变量初始化）。返回0，成功，返回-1，失败，服务进程退出。

B：int svc\_initconn() 该函数是有新的网络句柄产生时，用来初始化这个网络句柄。返回0，成功，其他，失败， 关闭该句柄。

C：int svc\_recv() 当该句柄有网络读（有数据请求）时被调用。根据返回值，修改该网络监听事件或者关闭连接。

D：int svc\_send() 当该网络发送过数据后，该函数被调用，根据返回值修改该网络监听事件或者关闭连接。

E：void svc\_timeout() 该函数是定时被调用，用来处理一些插件的定时任务及其他。（例如，主动发起连接，心跳维护等）

F：void svc\_finiconn() 该函数在网络句柄被关闭时调用。

1. **框架各文件说明**

**atomic.h 原子操作工具，一般用于计数**

结构体 atomic\_t

操作操作api

#define ATOMIC\_INIT(i) { (i) }

#define atomic\_read(v) ((v)->counter)

#define atomic\_set(v,i) (((v)->counter) = (i))

static \_\_inline\_\_ void atomic\_add(int i, atomic\_t \*v)

static \_\_inline\_\_ void atomic\_sub(int i, atomic\_t \*v)

static \_\_inline\_\_ int atomic\_sub\_and\_test(int i, atomic\_t \*v)

static \_\_inline\_\_ void atomic\_inc(atomic\_t \*v)

static \_\_inline\_\_ void atomic\_dec(atomic\_t \*v)

static \_\_inline\_\_ int atomic\_dec\_and\_test(atomic\_t \*v)

static \_\_inline\_\_ int atomic\_inc\_and\_test(atomic\_t \*v)

static \_\_inline\_\_ int atomic\_add\_negative(int i, atomic\_t \*v)

**daemon.h 程序后台运行工具**

主要全局变量说明

volatile extern int stop; //1-服务器停止，0-服务器运行中

volatile extern int restart; //1-服务器异常退出后要自动重启，0-不自动重启

操作操作api

extern int daemon\_start(int, char \*\*); //设置程序后台运行

extern void daemon\_stop(); //停止或者重启程序

extern void daemon\_set\_title(const char \*title); //设置后台进程名

**fdinfo.h 程序网络连接使用fd、struct conn资源初步化工具**

操作操作api

extern struct conn \*acon; //全局网络连接fd资源指针

extern int maxfds; //初始化fd资源最大值，默认4096

extern int init\_fdinfo(void); //初始化全局使用的fd相关资源

extern void fini\_fdinfo(void); //释放fd使用的相关资源

**global.h 包含框架与插件交互结构体及api**

一般记录日志用到的宏定义，用于打印日志位置信息

#define ID \_\_FILE\_\_

#define LN \_\_LINE\_\_

#define FUNC \_\_FUNCTION\_\_

//global conn 全局网络连接fd资源指针

extern struct conn \*acon;

//global log fd 全局日志fd

extern int glogfd;

//connect object 连接信息结构体

struct conn {

char peerip[16]; //for debug print

int fd; //recheck avoid coredump

struct mybuff send\_buff; //send buffer for client struct mybuff recv\_buff; //recv buffer for client

int send\_len; //send len between call svc\_send void\* user; //user custom data，插件自定义结构体指针

};

#define RECV\_CLOSE 0x01 //do\_recv need to close socket

#define RECV\_ADD\_EPOLLIN 0x02 //do\_recv need to add fd EPOLLIN

#define RECV\_ADD\_EPOLLOUT 0x04 //do\_recv need to add fd EPOLLOUT

#define RECV\_ADD\_EPOLLALL 0x06 //do\_recv need to add fd EPOLLOUT and EPOLLIN

#define RECV\_SEND 0x08 //do\_recv need to send at once

#define SEND\_CLOSE 0x10 //do\_send need to close socket

#define SEND\_ADD\_EPOLLIN 0x20 //do\_send need to add fd EPOLLIN

#define SEND\_ADD\_EPOLLOUT 0x40 //do\_send need to add fd EPOLLOUT

#define SEND\_ADD\_EPOLLALL 0x80 //do\_send need to add fd EPOLLOUT and EPOLLIN

#define RET\_OK 300

#define RET\_SUCCESS 301

#define RET\_CLOSE\_HB 302 //4 detect hb

#define RET\_CLOSE\_MALLOC 303 //4 malloc err

#define RET\_CLOSE\_DUP 304 //dup connect

enum MODE {CON\_PASSIVE = 0, CON\_ACTIVE};

enum SERVER\_STAT {UNKOWN\_STAT = 0, WAIT\_SYNC, SYNCING, SYNCED, ON\_LINE};

**init.h 框架初始化，调用方法**

//mybuff init buff size 网络连接缓存大小设置

extern int init\_buff\_size;

//server self stat 服务器状态

uint8\_t self\_stat;

//global info struct

typedef struct {

uint16\_t sig\_port; //signal channel port

uint16\_t data\_port; //data channel port

uint16\_t timeout; //timeout to close connect

uint16\_t chktimeout; //timeout to invoke so timeout method

} t\_g\_config;

extern t\_g\_config g\_config; //全局配置信息结构体，用于读取配置信息

extern uint32\_t localip[64]; //服务器本地IP信息数组，uint32\_t格式

int init\_global(); //初始化配置信息

extern int init\_work\_thread(t\_thread\_arg \*name); //注册工作线程信息

**mybuff.h 网络连接缓存工具**

结构体

struct mybuff {

char\* data; //buffer指针

size\_t size; //buffer大小

size\_t len; //数据有效长度

int fd; //需要发送文件fd,使用sendfile64发送

off\_t foffset; //发送文件起始位置

size\_t flen; //发送文件长度

};

操作api

extern void mybuff\_init(struct mybuff\* mybuff); //初始缓冲区

mybuff\_setdata(struct mybuff\* mybuff, const char\* data, size\_t len); //写入数据

mybuff\_setfile(struct mybuff\* mybuff, int fd, off\_t offset, size\_t len); //写入文件信息

mybuff\_getdata(struct mybuff\* mybuff, char\*\* data, size\_t\* len); //取数据

mybuff\_skipdata(struct mybuff\* mybuff, size\_t en); //设置使用数据量（消耗缓存）

mybuff\_getfile(struct mybuff\* mybuff, int\* fd, off\_t\* offset, size\_t \* len); //取文件信息

mybuff\_skipfile(struct mybuff\* mybuff, size\_t len); //设置使用文件数据量

mybuff\_reinit(struct mybuff\* mybuff); //重新初始化缓存

mybuff\_fini(struct mybuff\* mybuff); //释放资源

**so.h 框架主要插件实现部分**

注册工作线程时需要设置的结构体信息

typedef struct {

char flag; //是否需要设置sockbuff

char name[255]; //插件名，用于打开so插件

int port; //如果port大于0则打开端口并进行监听

int maxevent; //epoll最大监听事件数

int threadcount; //暂时没用到

} t\_thread\_arg;

int log\_signalling\_thread(void \*arg); //线程初始化及主循环方法

void add\_fd\_2\_efd(int fd); //主动打开的网络连接fd添加到epoll管理

void do\_close(int fd); //关闭网络连接fd

void modify\_fd\_event(int fd, int events); //修改fd epoll事件类型

int get\_client\_data(int fd, char \*\*data, size\_t \*len); //从fd缓存取数据

int consume\_client\_data(int fd, size\_t len); //消费fd中缓存数据

int set\_client\_data(int fd, char \*buf, size\_t len); //添加数据到fd缓存

int set\_client\_fd(int fd, int lfd, size\_t offset, size\_t len); //添加文件数据fd缓存

**solib.h 插件信息工具**

typedef int (\*proc\_method) (int fd);

typedef int (\*proc\_init) (void);

typedef void (\*proc\_fini) (int fd);

typedef void (\*proc\_timeout) (void);

struct mylib {

void\* handle;

proc\_init svc\_init; //O

proc\_method svc\_initconn;//O

proc\_method svc\_recv; //M

proc\_method svc\_send; //M

proc\_fini svc\_finiconn;//O

proc\_timeout svc\_timeout; //O

};

**task.h 任务队列工具**

/\*任务队列类型，任务结点从不同队列类型中移动\*/

enum {TASK\_HOME = 0, COLLECT\_TASK, CANCEL\_TASK, TASK\_UNKNOWN};

/\*取任务队列状态\*/

enum {GET\_TASK\_ERR = -1, GET\_TASK\_OK, GET\_TASK\_NOTHING};

/\*遍历任务队列时是否移除任务结点\*/

enum {TASK\_DEL = 0, TASK\_HOLD};

/\*任务队列结点结构体\*/

typedef struct {

list\_head\_t llist; //不同队列类型链表结点

list\_head\_t hlist; //全局队列链表结点

int status; //任务所在队列类型

void \*user; //用户自定义结构体指针

}log\_tasklist;

typedef void (\*timeout\_task)(log\_tasklist \*task); //遍历时回调方法

int init\_task\_info(); //初始化任务队列

int log\_set\_task(log\_tasklist \*task, int status); //添加结点到status类型队列

int log\_get\_task(log\_tasklist \*\*task, int status); //从status队列取结点

int mv\_task\_to(log\_tasklist \*task, int status); //移动任务结点到status队列

int scan\_status\_task(int status, timeout\_task ob, int del); //遍历status队列，每个结点调用timeout\_task\_ob方法

**thread.h 线程工具**

线程结构体

struct thread {

char name[64]; //线程名字

int (\*entry)(void \*); //线程入口方法

void \*arg; //线程参数

pthread\_t tid; //当前线程ID

pid\_t pid; //进程ID

unsigned short id;

unsigned char state;

unsigned char inuse;

int rv;

};

enum {

THREAD\_UNUSED = 0, /\* slot not used \*/

THREAD\_STOPPED = 1, /\* not running \*/

THREAD\_STANDBY, /\* in standby list \*/

THREAD\_STARTING, /\* starting thread \*/

THREAD\_RUNNING, /\* busy running \*/

THREAD\_STOPPING, /\* stopping req \*/

THREAD\_EXITED, /\* exited \*/

THREAD\_KILLED /\* killed \*/

};

extern struct thread mainthread; //主线程信息

extern int init\_thread(void); //初始化线程运行环境

extern int register\_thread(const char \*name, int(\*entry)(void \*), void \*priv); //注册线程信息

extern int start\_threads(void); //启动已注册线程

extern void stop\_threads(void); //停止所有线程

extern void thread\_jumbo\_title(void); //设置线程名

extern struct thread \*get\_my\_thread(void); //取当前线程信息

extern int get\_thread\_id(); //取当前线程ID

**version.h 程序版本及编译时间**

//服务器版本描述字符串

extern char version\_string[];

//代码编译日期

extern char compiling\_date[];

**watchdog.h 看门狗工具，用于监听各线程心跳及重启程序**

//看门狗进程ID

extern int watchdog\_pid;

//进程心跳统计结构体

struct threadstat {

int tid; //线程ID

atomic\_t tickcnt; //tick数目

int badcnt; //线程被判定为停滞状态的次数

};

//线程定期心跳调用方法

static inline void thread\_reached(struct threadstat \*ts) {

if(ts)

atomic\_inc(&ts->tickcnt);

}

//取当前线程统计结构体

extern struct threadstat \*get\_threadstat(void);

//启动看门狗

extern int start\_watchdog(void);