4 存储服务器

~/TNV/src/04_storage/01_globals.h

```
// 存储服务器
// 声明全局变量
//
#pragma once

#include <vector>
#include <string>
#include <lib_acl.hpp>
//
// 配置信息
//
extern char* cfg_gpname; // 隶属组名
extern char* cfg_spaths; // 存储路径表
extern char* cfg_taddrs; // 跟踪服务器地址表
extern char* cfg_iaddrs; // ID服务器地址表
extern char* cfg_maddrs; // MySQL地址表
extern char* cfg_raddrs; // Redis地址表
extern acl::master_str_tbl cfg_str[]; // 字符串配置表

extern int cfg_bindport; // 绑定端口号
extern int cfg_interval; // 心跳间隔秒数
extern int cfg_mtimeout; // MySQL读写超时
extern int cfg_maxconns; // Redis连接池最大连接数
extern int cfg_ctimeout; // Redis连接超时
extern int cfg_rtimeout; // Redis读写超时
extern int cfg_ktimeout; // Redis键超时
extern acl::master_int_tbl cfg_int[]; // 整型配置表

extern std::vector<std::string> g_spaths; // 存储路径表
extern std::vector<std::string> g_taddrs; // 跟踪服务器地址表
extern std::vector<std::string> g_iaddrs; // ID服务器地址表
extern std::vector<std::string> g_maddrs; // MySQL地址表
extern std::vector<std::string> g_raddrs; // Redis地址表
extern acl::redis_client_pool*  g_rconns; // Redis连接池
extern std::string g_hostname; // 主机名
extern char const* g_version; // 版本
extern time_t g_stime; // 启动时间
```

~/TNV/src/04_storage/02_globals.cpp

```
// 存储服务器
// 定义全局变量
//
#include "01_globals.h"
//
// 配置信息
//
char* cfg_gpname; // 隶属组名
```

```
char* cfg_spaths; // 存储路径表
char* cfg_taddrs; // 跟踪服务器地址表
char* cfg_iaddrs; // ID服务器地址表
char* cfg_maddrs; // MySQL地址表
char* cfg_raddrs; // Redis地址表
acl::master_str_tbl cfg_str[] = { // 字符串配置表
    {"tnv_group_name",    "group001",        &cfg_gpname},
    {"tnv_store_paths",   "../data",         &cfg_spaths},
    {"tnv_tracker_addrs", "127.0.0.1:21000", &cfg_taddrs},
    {"tnv_ids_addrs",     "127.0.0.1:22000", &cfg_iaddrs},
    {"mysql_addrs",       "127.0.0.1",       &cfg_maddrs},
    {"redis_addrs",       "127.0.0.1:6379",  &cfg_raddrs},
    {0, 0, 0}};

int cfg_bindport; // 绑定端口号
int cfg_interval; // 心跳间隔秒数
int cfg_mtimeout; // MySQL读写超时
int cfg_maxconns; // Redis连接池最大连接数
int cfg_ctimeout; // Redis连接超时
int cfg_rtimeout; // Redis读写超时
int cfg_ktimeout; // Redis键超时
acl::master_int_tbl cfg_int[] = { // 整型配置表
    {"tnv_storage_port",        23000, &cfg_bindport, 0, 0},
    {"tnv_heart_beat_interval",    10, &cfg_interval, 0, 0},
    {"mysql_rw_timeout",           30, &cfg_mtimeout, 0, 0},
    {"redis_max_conn_num",        600, &cfg_maxconns, 0, 0},
    {"redis_conn_timeout",         10, &cfg_ctimeout, 0, 0},
    {"redis_rw_timeout",           10, &cfg_rtimeout, 0, 0},
    {"redis_key_timeout",          60, &cfg_ktimeout, 0, 0},
    {0, 0, 0, 0, 0}};

std::vector<std::string> g_spaths; // 存储路径表
std::vector<std::string> g_taddrs; // 跟踪服务器地址表
std::vector<std::string> g_iaddrs; // ID服务器地址表
std::vector<std::string> g_maddrs; // MySQL地址表
std::vector<std::string> g_raddrs; // Redis地址表
acl::redis_client_pool*  g_rconns; // Redis连接池
std::string g_hostname; // 主机名
char const* g_version = "1.0"; // 版本
time_t g_stime; // 启动时间
```

~/TNV/src/04_storage/03_cache.h

```
// 存储服务器
// 声明缓存类
//
#pragma once

#include <lib_acl.hpp>
//
// 缓存类
//
class cache_c {
public:
    // 根据键获取其值
```

```
    int get(char const* key, acl::string& value) const;

    // 设置指定键的值
    int set(char const* key, char const* value, int timeout = -1) const;

    // 删除指定键值对
    int del(char const* key) const;
};
```

~/TNV/src/04_storage/04_cache.cpp

```
// 存储服务器
// 实现缓存类
//
#include "01_types.h"
#include "01_globals.h"
#include "03_cache.h"

// 根据键获取其值
int cache_c::get(char const* key, acl::string& value) const {
    // 构造键
    acl::string storage_key;
    storage_key.format("%s:%s", STORAGE_REDIS_PREFIX, key);

    // 检查Redis连接池
    if (!g_rconns) {
        logger_warn("redis connection pool is null, key: %s",
            storage_key.c_str());
        return ERROR;
    }

    // 从连接池中获取一个Redis连接
    acl::redis_client* rconn = (acl::redis_client*)g_rconns->peek();
    if (!rconn) {
        logger_warn("peek redis connection fail, key: %s",
            storage_key.c_str());
        return ERROR;
    }

    // 持有此连接的Redis对象即为Redis客户机
    acl::redis redis;
    redis.set_client(rconn);

    // 借助Redis客户机根据键获取其值
    if (!redis.get(storage_key.c_str(), value)) {
        logger_warn("get cache fail, key: %s", storage_key.c_str());
        g_rconns->put(rconn, false);
        return ERROR;
    }

    // 检查空值
    if (value.empty()) {
        logger_warn("value is empty, key: %s", storage_key.c_str());
        g_rconns->put(rconn, false);
        return ERROR;
```

```
    }

    logger("get cache ok, key: %s, value: %s",
        storage_key.c_str(), value.c_str());
    g_rconns->put(rconn, true);

    return OK;
}

// 设置指定键的值
int cache_c::set(char const* key, char const* value,
    int timeout /* = -1 */) const {
    // 构造键
    acl::string storage_key;
    storage_key.format("%s:%s", STORAGE_REDIS_PREFIX, key);

    // 检查Redis连接池
    if (!g_rconns) {
        logger_warn("redis connection pool is null, key: %s",
            storage_key.c_str());
        return ERROR;
    }

    // 从连接池中获取一个Redis连接
    acl::redis_client* rconn = (acl::redis_client*)g_rconns->peek();
    if (!rconn) {
        logger_warn("peek Redis connection fail, key: %s",
            storage_key.c_str());
        return ERROR;
    }

    // 持有此连接的Redis对象即为Redis客户机
    acl::redis redis;
    redis.set_client(rconn);

    // 借助Redis客户机设置指定键的值
    if (timeout < 0)
        timeout = cfg_ktimeout;
    if (!redis.setex(storage_key.c_str(), value, timeout)) {
        logger_warn("set cache fail, key: %s, value: %s, timeout: %d",
            storage_key.c_str(), value, timeout);
        g_rconns->put(rconn, false);
        return ERROR;
    }
    logger("set cache ok, key: %s, value: %s, timeout: %d",
        storage_key.c_str(), value, timeout);
    g_rconns->put(rconn, true);

    return OK;
}

// 删除指定键值对
int cache_c::del(char const* key) const {
    // 构造键
    acl::string storage_key;
    storage_key.format("%s:%s", STORAGE_REDIS_PREFIX, key);
```

```
101
102     // 检查Redis连接池
103     if (!g_rconns) {
104         logger_warn("redis connection pool is null, key: %s",
105             storage_key.c_str());
106         return ERROR;
107     }
108
109     // 从连接池中获取一个Redis连接
110     acl::redis_client* rconn = (acl::redis_client*)g_rconns->peek();
111     if (!rconn) {
112         logger_warn("peek Redis connection fail, key: %s",
113             storage_key.c_str());
114         return ERROR;
115     }
116
117     // 持有此连接的Redis对象即为Redis客户机
118     acl::redis redis;
119     redis.set_client(rconn);
120
121     // 借助Redis客户机删除指定键值对
122     if (!redis.del_one(storage_key.c_str())) {
123         logger_warn("delete cache fail, key: %s", storage_key.c_str());
124         g_rconns->put(rconn, false);
125         return ERROR;
126     }
127     logger("delete cache ok, key: %s", storage_key.c_str());
128     g_rconns->put(rconn, true);
129
130     return OK;
131 }
```

~/TNV/src/04_storage/05_db.h

```
1  // 存储服务器
2  // 声明数据库访问类
3  //
4  #pragma once
5
6  #include <string>
7  #include <mysql.h>
8  //
9  // 数据库访问类
10 //
11 class db_c {
12 public:
13     // 构造函数
14     db_c(void);
15     // 析构函数
16     ~db_c(void);
17
18     // 连接数据库
19     int connect(void);
20
21     // 根据文件ID获取其对应的路径及大小
```

```
    int get(char const* appid, char const* userid, char const* fileid,
        std::string& filepath, long long* filesize) const;
    // 设置文件ID和路径及大小的对应关系
    int set(char const* appid, char const* userid, char const* fileid,
        char const* filepath, long long filesize) const;
    // 删除文件ID
    int del(char const* appid, char const*userid,
        char const* fileid) const;

private:
    // 根据用户ID获取其对应的表名
    std::string table_of_user(char const* userid) const;

    // 计算哈希值
    unsigned int hash(char const* buf, size_t len) const;

    MYSQL* m_mysql; // MySQL对象
};
```

~/TNV/src/04_storage/06_db.cpp

```
// 存储服务器
// 实现数据库访问类
//
#include "01_types.h"
#include "01_globals.h"
#include "03_cache.h"
#include "05_db.h"

// 构造函数
db_c::db_c(void): m_mysql(mysql_init(NULL)) { // 创建MySQL对象
    if (!m_mysql)
        logger_error("create dao fail: %s", mysql_error(m_mysql));
}

// 析构函数
db_c::~db_c(void) {
    // 销毁MySQL对象
    if (m_mysql) {
        mysql_close(m_mysql);
        m_mysql = NULL;
    }
}

// 连接数据库
int db_c::connect(void) {
    MYSQL* mysql = m_mysql;

    // 遍历MySQL地址表，尝试连接数据库
    for (std::vector<std::string>::const_iterator maddr =
        g_maddrs.begin(); maddr != g_maddrs.end(); ++maddr)
        if ((m_mysql = mysql_real_connect(mysql, maddr->c_str(),
            "root", "123456", "tnv_storagedb", 0, NULL, 0)))
            return OK;

```

```
    logger_error("connect database fail: %s",
        mysql_error(m_mysql = mysql));
    return ERROR;
}

// 根据文件ID获取其对应的路径及大小
int db_c::get(char const* appid, char const* userid, char const* fileid,
    std::string& filepath, long long* filesize) const {
    // 先尝试从缓存中获取与文件ID对应的路径及大小
    cache_c cache;
    acl::string key;
    key.format("uid:fid:%s:%s", userid, fileid);
    acl::string value;
    if (cache.get(key, value) == OK) {
        std::vector<acl::string> size_path = value.split2(";");
        if (size_path.size() == 2) {
            filepath = size_path[1].c_str();
            *filesize = atoll(size_path[0].c_str());
            if (!filepath.empty() && *filesize > 0) {
                logger("from cache, appid: %s, userid: %s, "
                    "fileid: %s, filepath: %s, filesize: %lld",
                    appid, userid, fileid, filepath.c_str(), *filesize);
                return OK;
            }
        }
    }

    // 缓存中没有再查询数据库
    std::string tablename = table_of_user(userid);
    if (tablename.empty()) {
        logger_error("tablename is empty, appid: %s, "
            "userid: %s, fileid: %s", appid, userid, fileid);
        return ERROR;
    }
    acl::string sql;
    sql.format("SELECT file_path, file_size FROM %s WHERE id='%s';",
        tablename.c_str(), fileid);
    if (mysql_query(m_mysql, sql.c_str())) {
        logger_error("query database fail: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    // 获取查询结果
    MYSQL_RES* res = mysql_store_result(m_mysql);
    if (!res) {
        logger_error("result is null: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    // 获取结果记录
    MYSQL_ROW row = mysql_fetch_row(res);
    if (!row) {
        logger_error("result is empty: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
```

```
        return ERROR;
    }
    filepath = row[0];
    *filesize = atoll(row[1]);
    logger("from database, appid: %s, userid: %s, "
        "fileid: %s, filepath: %s, filesize: %lld",
        appid, userid, fileid, filepath.c_str(), *filesize);

    // 将文件ID和路径及大小的对应关系保存在缓存中
    value.format("%lld;%s", *filesize, filepath.c_str());
    cache.set(key, value.c_str());

    return OK;
}

// 设置文件ID和路径及大小的对应关系
int db_c::set(char const* appid, char const* userid, char const* fileid,
    char const* filepath, long long filesize) const {
    // 根据用户ID获取其对应的表名
    std::string tablename = table_of_user(userid);
    if (tablename.empty()) {
        logger_error("tablename is empty, appid: %s, "
            "userid: %s, fileid: %s", appid, userid, fileid);
        return ERROR;
    }

    // 插入一条记录
    acl::string sql;
    sql.format("INSERT INTO %s SET id='%s', appid='%s', "
        "userid='%s', status=0, file_path='%s', file_size=%lld;",
        tablename.c_str(), fileid, appid, userid, filepath, filesize);
    if (mysql_query(m_mysql, sql.c_str())) {
        logger_error("insert database fail: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    // 检查插入结果
    MYSQL_RES* res = mysql_store_result(m_mysql);
    if (!res && mysql_field_count(m_mysql)) {
        logger_error("insert database fail: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    return OK;
}

// 删除文件ID
int db_c::del(char const* appid, char const*userid,
    char const* fileid) const {
    // 先从缓存中删除文件ID
    cache_c cache;
    acl::string key;
    key.format("uid:fid:%s:%s", userid, fileid);
    if (cache.del(key) != OK)
```

```
        logger_warn("delete cache fail: appid: %s, "
            "userid: %s, fileid: %s", appid, userid, fileid);

    // 再从数据库中删除文件ID
    std::string tablename = table_of_user(userid);
    if (tablename.empty()) {
        logger_error("tablename is empty, appid: %s, "
            "userid: %s, fileid: %s", appid, userid, fileid);
        return ERROR;
    }
    acl::string sql;
    sql.format("DELETE FROM %s WHERE id='%s';",
        tablename.c_str(), fileid);
    if (mysql_query(m_mysql, sql.c_str())) {
        logger_error("delete database fail: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    // 检查删除结果
    MYSQL_RES* res = mysql_store_result(m_mysql);
    if (!res && mysql_field_count(m_mysql)) {
        logger_error("delete database fail: %s, sql: %s",
            mysql_error(m_mysql), sql.c_str());
        return ERROR;
    }

    return OK;
}

// 根据用户ID获取其对应的表名
std::string db_c::table_of_user(char const* userid) const {
    char tablename[10];

    sprintf(tablename, "t_file_%02d",
        (hash(userid, strlen(userid)) & 0x7FFFFFFF) % 3 + 1);

    return tablename;
}

// 计算哈希值
unsigned int db_c::hash(char const* buf, size_t len) const {
    unsigned int h = 0;

    for (size_t i = 0; i < len; ++i)
        h ^= i&1 ? ~(h<<11^buf[i]^h>>5) : h<<7^buf[i]^h>>3;

    return h;
}
```

~/TNV/src/04_storage/07_file.h

```
// 存储服务器
// 声明文件操作类
//
```

```h
#pragma once

#include <sys/types.h>
//
// 文件操作类
//
class file_c {
public:
    // 构造函数
    file_c(void);
    // 析构函数
    ~file_c(void);

    // 打开文件
    int open(char const* path, char flag);
    // 关闭文件
    int close(void);

    // 读取文件
    int read(void* buf, size_t count) const;
    // 写入文件
    int write(void const* buf, size_t count) const;

    // 设置偏移
    int seek(off_t offset) const;
    // 删除文件
    static int del(char const* path);

    // 打开标志
    static char const O_READ   = 'r';
    static char const O_WRITE  = 'w';

private:
    int m_fd; // 文件描述符
};
```

~/TNV/src/04_storage/08_file.cpp

```cpp
// 存储服务器
// 实现文件操作类
//
#include <unistd.h>
#include <fcntl.h>
#include <lib_acl.hpp>
#include "01_types.h"
#include "07_file.h"

// 构造函数
file_c::file_c(void): m_fd(-1) {
}

// 析构函数
file_c::~file_c(void) {
    close();
}
```

```cpp

// 打开文件
int file_c::open(char const* path, char flag) {
    // 检查路径
    if (!path || !*path) {
        logger_error("path is null");
        return ERROR;
    }

    // 打开文件
    if (flag == O_READ)
        m_fd = ::open(path, O_RDONLY);
    else if (flag == O_WRITE)
        m_fd = ::open(path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    else {
        logger_error("unknown open flag: %c", flag);
        return ERROR;
    }

    // 打开失败
    if (m_fd == -1) {
        logger_error("call open fail: %s, path: %s, flag: %c",
            strerror(errno), path, flag);
        return ERROR;
    }

    return OK;
}

// 关闭文件
int file_c::close(void) {
    if (m_fd != -1) {
        // 关闭文件
        if (::close(m_fd) == -1) {
            logger_error("call close fail: %s", strerror(errno));
            return ERROR;
        }

        m_fd = -1;
    }

    return OK;
}

// 读取文件
int file_c::read(void* buf, size_t count) const {
    // 检查文件描述符
    if (m_fd == -1) {
        logger_error("invalid file handle");
        return ERROR;
    }

    // 检查文件缓冲区
    if (!buf) {
        logger_error("invalid file buffer");
        return ERROR;
```

```
    }

    // 读取给定字节数
    ssize_t bytes = ::read(m_fd, buf, count);
    if (bytes == -1) {
        logger_error("call read fail: %s", strerror(errno));
        return ERROR;
    }
    if ((size_t)bytes != count) {
        logger_error("unable to read expected bytes: %ld != %lu",
            bytes, count);
        return ERROR;
    }

    return OK;
}

// 写入文件
int file_c::write(void const* buf, size_t count) const {
    // 检查文件描述符
    if (m_fd == -1) {
        logger_error("invalid file handle");
        return ERROR;
    }

    // 检查文件缓冲区
    if (!buf) {
        logger_error("invalid file buffer");
        return ERROR;
    }

    // 写入给定字节数
    if (::write(m_fd, buf, count) == -1) {
        logger_error("call write fail: %s", strerror(errno));
        return ERROR;
    }

    return OK;
}

// 设置偏移
int file_c::seek(off_t offset) const {
    // 检查文件描述符
    if (m_fd == -1) {
        logger_error("invalid file handle");
        return ERROR;
    }

    // 检查文件偏移量
    if (offset < 0) {
        logger_error("invalid file offset");
        return ERROR;
    }

    // 设置文件偏移量
    if (lseek(m_fd, offset, SEEK_SET) == -1) {
```

```
        logger_error("call lseek fail: %s, offset: %ld",
            strerror(errno), offset);
        return ERROR;
    }

    return OK;
}

// 删除文件
int file_c::del(char const* path) {
    // 检查路径
    if (!path || !*path) {
        logger_error("path is null");
        return ERROR;
    }

    // 删除文件
    if (unlink(path) == -1) {
        logger_error("call unlink fail: %s, path: %s",
            strerror(errno), path);
        return ERROR;
    }

    return OK;
}
```

~/TNV/src/04_storage/09_id.h

```
// 存储服务器
// 声明ID客户机类
//
#pragma once
//
// ID客户机类
//
class id_c {
public:
    // 从ID服务器获取与ID键相对应的值
    long get(char const* key) const;

private:
    // 向ID服务器发送请求，接收并解析响应，从中获得ID值
    long client(char const* requ, long long requlen) const;
};
```

~/TNV/src/04_storage/10_id.cpp

```
// 存储服务器
// 实现ID客户机类
//
#include <lib_acl.hpp>
#include "02_proto.h"
#include "03_util.h"
#include "01_globals.h"
```

```cpp
#include "09_id.h"

// 从ID服务器获取与ID键相对应的值
long id_c::get(char const* key) const {
    // 检查ID键
    if (!key) {
        logger_error("key is null");
        return -1;
    }
    size_t keylen = strlen(key);
    if (!keylen) {
        logger_error("key is null");
        return -1;
    }
    if (keylen > ID_KEY_MAX) {
        logger_error("key too big: %lu > %d", keylen, ID_KEY_MAX);
        return -1;
    }

    // |包体长度|命令|状态|  ID键  |
    // |    8   |  1 |  1 |包体长度|
    // 构造请求
    long long bodylen = keylen + 1;
    long long requlen = HEADLEN + bodylen;
    char requ[requlen] = {};
    llton(bodylen, requ);
    requ[BODYLEN_SIZE] = CMD_ID_GET;
    requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    strcpy(requ + HEADLEN, key);

    // 向ID服务器发送请求，接收并解析响应，从中获得ID值
    return client(requ, requlen);
}

// 向ID服务器发送请求，接收并解析响应，从中获得ID值
long id_c::client(char const* requ, long long requlen) const {
    acl::socket_stream conn;

    // 从ID服务器地址表中随机抽取一台ID服务器尝试连接
    srand(time(NULL));
    int nids = g_iaddrs.size();
    int nrand = rand() % nids;
    for (int i = 0; i < nids; ++i)
        if (conn.open(g_iaddrs[nrand].c_str(), 0, 0)) {
            logger("connect id success, addr: %s",
                g_iaddrs[nrand].c_str());
            break;
        }
        else {
            logger("connect id fail, addr: %s",
                g_iaddrs[nrand].c_str());
            nrand = (nrand + 1) % nids;
        }
    if (!conn.alive()) {
        logger_error("connect id fail, addrs: %s", cfg_iaddrs);
        return -1;
```

```
    }

    // 向ID服务器发送请求
    if (conn.write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, conn.get_peer());
        conn.close();
        return -1;
    }

    // 从ID服务器接收响应
    long long resplen = HEADLEN + BODYLEN_SIZE;
    char resp[resplen] = {};
    if (conn.read(resp, resplen) < 0) {
        logger_error("read fail: %s, resplen: %lld, from: %s",
            acl::last_serror(), resplen, conn.get_peer());
        conn.close();
        return -1;
    }

    // |包体长度|命令|状态|ID值|
    // |    8   |  1 |  1 |  8 |
    // 从ID服务器的响应中解析出ID值
    long value = ntol(resp + HEADLEN);

    conn.close();

    return value;
}
```

~/TNV/src/04_storage/11_service.h

```
// 存储服务器
// 声明业务服务类
//
#pragma once

#include <lib_acl.hpp>
//
// 业务服务类
//
class service_c {
public:
    // 业务处理
    bool business(acl::socket_stream* conn, char const* head) const;

private:
    // 处理来自客户机的上传文件请求
    bool upload(acl::socket_stream* conn, long long bodylen) const;
    // 处理来自客户机的询问文件大小请求
    bool filesize(acl::socket_stream* conn, long long bodylen) const;
    // 处理来自客户机的下载文件请求
    bool download(acl::socket_stream* conn, long long bodylen) const;
    // 处理来自客户机的删除文件请求
    bool del(acl::socket_stream* conn, long long bodylen) const;
```

```
24
25      // 生成文件路径
26      int genpath(char* filepath) const;
27      // 将ID转换为512进制
28      long id512(long id) const;
29      // 用文件ID生成文件路径
30      int id2path(char const* spath, long fileid, char* filepath) const;
31      // 接收并保存文件
32      int save(acl::socket_stream* conn, char const* appid,
33          char const* userid, char const* fileid, long long filesize,
34          char const* filepath) const;
35
36      // 读取并发送文件
37      int send(acl::socket_stream* conn, char const* filepath,
38          long long offset, long long size) const;
39
40      // 应答成功
41      bool ok(acl::socket_stream* conn) const;
42      // 应答错误
43      bool error(acl::socket_stream* conn, short errnumb,
44          char const* format, ...) const;
45  };
```

~/TNV/src/04_storage/12_service.cpp

```
1   // 存储服务器
2   // 实现业务服务类
3   //
4   #include <linux/limits.h>
5   #include <algorithm>
6   #include "02_proto.h"
7   #include "03_util.h"
8   #include "01_globals.h"
9   #include "05_db.h"
10  #include "07_file.h"
11  #include "09_id.h"
12  #include "11_service.h"
13
14  // 业务处理
15  bool service_c::business(acl::socket_stream* conn,
16      char const* head) const {
17      // |包体长度|命令|状态|  包体  |
18      // |    8   |  1 |  1 |包体长度|
19      // 解析包头
20      long long bodylen = ntoll(head); // 包体长度
21      if (bodylen < 0) {
22          error(conn, -1, "invalid body length: %lld < 0", bodylen);
23          return false;
24      }
25      int command = head[BODYLEN_SIZE]; // 命令
26      int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
27      logger("bodylen: %lld, command: %d, status: %d",
28          bodylen, command, status);
29
30      bool result;
```

```
31
32      // 根据命令执行具体业务处理
33      switch (command) {
34          case CMD_STORAGE_UPLOAD:
35              // 处理来自客户机的上传文件请求
36              result = upload(conn, bodylen);
37              break;
38
39          case CMD_STORAGE_FILESIZE:
40              // 处理来自客户机的询问文件大小请求
41              result = filesize(conn, bodylen);
42              break;
43
44          case CMD_STORAGE_DOWNLOAD:
45              // 处理来自客户机的下载文件请求
46              result = download(conn, bodylen);
47              break;
48
49          case CMD_STORAGE_DELETE:
50              // 处理来自客户机的删除文件请求
51              result = del(conn, bodylen);
52              break;
53
54          default:
55              error(conn, -1, "unknown command: %d", command);
56              return false;
57      }
58
59      return result;
60  }
61
62  ////////////////////////////////////////////////////////////////////////
63
64  // 处理来自客户机的上传文件请求
65  bool service_c::upload(acl::socket_stream* conn,
66      long long bodylen) const {
67      // |包体长度|命令|状态|应用ID|用户ID|文件ID|文件大小|文件内容|
68      // |    8   |  1 |  1 |  16  |  256 |  128 |    8   |文件大小|
69      // 检查包体长度
70      long long expected = APPID_SIZE + USERID_SIZE +FILEID_SIZE +
71          BODYLEN_SIZE;
72      if (bodylen < expected) {
73          error(conn, -1, "invalid body length: %lld < %lld",
74              bodylen, expected);
75          return false;
76      }
77
78      // 接收包体
79      char body[expected];
80      if (conn->read(body, expected) < 0) {
81          logger_error("read fail: %s, expected: %lld, from: %s",
82              acl::last_serror(), expected, conn->get_peer());
83          return false;
84      }
85
86      // 解析包体
```

```
    char appid[APPID_SIZE];
    strcpy(appid, body);
    char userid[USERID_SIZE];
    strcpy(userid, body + APPID_SIZE);
    char fileid[FILEID_SIZE];
    strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
    long long filesize = ntoll(
        body + APPID_SIZE + USERID_SIZE + FILEID_SIZE);

    // 检查文件大小
    if (filesize != bodylen - expected) {
        logger_error("invalid file size: %lld != %lld",
            filesize, bodylen - expected);
        error(conn, -1, "invalid file size: %lld != %lld",
            filesize, bodylen - expected);
        return false;
    }

    // 生成文件路径
    char filepath[PATH_MAX+1];
    if (genpath(filepath) != OK) {
        error(conn, -1, "get filepath fail");
        return false;
    }

    logger("upload file, appid: %s, userid: %s, "
        "fileid: %s, filesize: %lld, filepath: %s",
        appid, userid, fileid, filesize, filepath);

    // 接收并保存文件
    int result = save(conn, appid, userid, fileid, filesize, filepath);
    if (result == SOCKET_ERROR)
        return false;
    else if (result == ERROR) {
        error(conn, -1, "receive and save file fail, fileid: %s",
            fileid);
        return false;
    }

    return ok(conn);
}

// 处理来自客户机的询问文件大小请求
bool service_c::filesize(acl::socket_stream* conn,
    long long bodylen) const {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|
    // |    8   |  1 |  1 |  16  |  256 |  128 |
    // 检查包体长度
    long long expected = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
    if (bodylen != expected) {
        error(conn, -1, "invalid body length: %lld != %lld",
            bodylen, expected);
        return false;
    }

    // 接收包体
```

```
    char body[bodylen];
    if (conn->read(body, bodylen) < 0) {
        logger_error("read fail: %s, bodylen: %lld, from: %s",
            acl::last_serror(), bodylen, conn->get_peer());
        return false;
    }

    // 解析包体
    char appid[APPID_SIZE];
    strcpy(appid, body);
    char userid[USERID_SIZE];
    strcpy(userid, body + APPID_SIZE);
    char fileid[FILEID_SIZE];
    strcpy(fileid, body + APPID_SIZE + USERID_SIZE);

    db_c db; // 数据库访问对象

    // 连接数据库
    if (db.connect() != OK)
        return false;

    std::string filepath; // 文件路径
    long long   filesize; // 文件大小

    // 根据文件ID获取其对应的路径及大小
    if (db.get(appid, userid, fileid, filepath, &filesize) != OK) {
        error(conn, -1, "read database fail, fileid: %s", fileid);
        return false;
    }

    logger("filesize, appid: %s, userid: %s, "
        "fileid: %s, filepath: %s, filesize: %lld",
        appid, userid, fileid, filepath.c_str(), filesize);

    // |包体长度|命令|状态|文件大小|
    // |    8   |  1 |  1 |    8   |
    // 构造响应
    bodylen = BODYLEN_SIZE;
    long long resplen = HEADLEN + bodylen;
    char resp[resplen] = {};
    llton(bodylen, resp);
    resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
    resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    llton(filesize, resp + HEADLEN);

    // 发送响应
    if (conn->write(resp, resplen) < 0) {
        logger_error("write fail: %s, resplen: %lld, to: %s",
            acl::last_serror(), resplen, conn->get_peer());
        return false;
    }

    return true;
}

// 处理来自客户机的下载文件请求
```

```
bool service_c::download(acl::socket_stream* conn,
    long long bodylen) const {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|偏移|大小|
    // |    8   |  1 |  1 |  16  |  256 |  128 |  8 |  8 |
    // 检查包体长度
    long long expected = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
        BODYLEN_SIZE + BODYLEN_SIZE;
    if (bodylen != expected) {
        error(conn, -1, "invalid body length: %lld != %lld",
            bodylen, expected);
        return false;
    }

    // 接收包体
    char body[bodylen];
    if (conn->read(body, bodylen) < 0) {
        logger_error("read fail: %s, bodylen: %lld, from: %s",
            acl::last_serror(), bodylen, conn->get_peer());
        return false;
    }

    // 解析包体
    char appid[APPID_SIZE];
    strcpy(appid, body);
    char userid[USERID_SIZE];
    strcpy(userid, body + APPID_SIZE);
    char fileid[FILEID_SIZE];
    strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
    long long offset = ntoll(
        body + APPID_SIZE + USERID_SIZE + FILEID_SIZE);
    long long size = ntoll(
        body + APPID_SIZE + USERID_SIZE + FILEID_SIZE + BODYLEN_SIZE);

    db_c db; // 数据库访问对象

    // 连接数据库
    if (db.connect() != OK)
        return false;

    std::string filepath; // 文件路径
    long long   filesize; // 文件大小

    // 根据文件ID获取其对应的路径及大小
    if (db.get(appid, userid, fileid, filepath, &filesize) != OK) {
        error(conn, -1, "read database fail, fileid: %s", fileid);
        return false;
    }

    // 检查位置
    if (offset < 0 || filesize < offset) {
        logger_error("invalid offset, %lld is not between 0 and %lld",
            offset, filesize);
        error(conn, -1, "invalid offset, %lld is not between 0 and %lld",
            offset, filesize);
        return false;
    }
```

```
255
256     // 大小为零表示下到文件尾
257     if (!size)
258         size = filesize - offset;
259
260     // 检查大小
261     if (size < 0 || filesize - offset < size) {
262         logger_error("invalid size, %lld is not between 0 and %lld",
263             size, filesize - offset);
264         error(conn, -1, "invalid offset, %lld is not between 0 and %lld",
265             size, filesize - offset);
266         return false;
267     }
268
269     logger("download file, appid: %s, userid: %s, fileid: %s, "
270         "offset: %lld, size: %lld, filepath: %s, filesize: %lld",
271         appid, userid, fileid, offset, size, filepath.c_str(), filesize);
272
273     // 读取并发送文件
274     int result = send(conn, filepath.c_str(), offset, size);
275     if (result == SOCKET_ERROR)
276         return false;
277     else if (result == ERROR) {
278         error(conn, -1, "read and send file fail, fileid: %s", fileid);
279         return false;
280     }
281
282     return true;
283 }
284
285 // 处理来自客户机的删除文件请求
286 bool service_c::del(acl::socket_stream* conn, long long bodylen) const {
287     // |包体长度|命令|状态|应用ID|用户ID|文件ID|
288     // |    8   |  1 |  1 |  16  |  256 |  128 |
289     // 检查包体长度
290     long long expected = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
291     if (bodylen != expected) {
292         error(conn, -1, "invalid body length: %lld != %lld",
293             bodylen, expected);
294         return false;
295     }
296
297     // 接收包体
298     char body[bodylen];
299     if (conn->read(body, bodylen) < 0) {
300         logger_error("read fail: %s, bodylen: %lld, from: %s",
301             acl::last_serror(), bodylen, conn->get_peer());
302         return false;
303     }
304
305     // 解析包体
306     char appid[APPID_SIZE];
307     strcpy(appid, body);
308     char userid[USERID_SIZE];
309     strcpy(userid, body + APPID_SIZE);
310     char fileid[FILEID_SIZE];
```

```
311     strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
312 
313     db_c db; // 数据库访问对象
314 
315     // 连接数据库
316     if (db.connect() != OK)
317         return false;
318 
319     std::string filepath; // 文件路径
320     long long   filesize; // 文件大小
321 
322     // 根据文件ID获取其对应的路径及大小
323     if (db.get(appid, userid, fileid, filepath, &filesize) != OK) {
324         error(conn, -1, "read database fail, fileid: %s", fileid);
325         return false;
326     }
327 
328     // 删除文件ID
329     if (db.del(appid, userid, fileid) != OK) {
330         error(conn, -1, "delete database fail, fileid: %s", fileid);
331         return false;
332     }
333 
334     // 删除文件
335     if (file_c::del(filepath.c_str()) != OK) {
336         error(conn, -1, "delete file fail, fileid: %s", fileid);
337         return false;
338     }
339 
340     logger("delete file success, appid: %s, userid: %s, "
341         "fileid: %s, filepath: %s, filesize: %lld",
342         appid, userid, fileid, filepath.c_str(), filesize);
343 
344     return ok(conn);
345 }
346 
347 ////////////////////////////////////////////////////////////////////////////
348 
349 // 生成文件路径
350 int service_c::genpath(char* filepath) const {
351     // 从存储路径表中随机抽取一个存储路径
352     srand(time(NULL));
353     int nspaths = g_spaths.size();
354     int nrand = rand() % nspaths;
355     std::string spath = g_spaths[nrand];
356 
357     // 以存储路径为键从ID服务器获取与之对应的值作为文件ID
358     id_c id;
359     long fileid = id.get(spath.c_str());
360     if (fileid < 0)
361         return ERROR;
362 
363     // 先将文件ID转换为512进制，再根据它生成文件路径
364     return id2path(spath.c_str(), id512(fileid), filepath);
365 }
366 
```

```
// 将ID转换为512进制
long service_c::id512(long id) const {
    long result = 0;

    for (int i = 1; id; i *= 1000) {
        result += (id % 512) * i;
        id /= 512;
    }

    return result;
}

// 用文件ID生成文件路径
int service_c::id2path(char const* spath, long fileid,
    char* filepath) const {
    // 检查存储路径
    if (!spath || !*spath) {
        logger_error("storage path is null");
        return ERROR;
    }

    // 生成文件路径中的各个分量
    unsigned short subdir1 = (fileid / 1000000000) % 1000; // 一级子目录
    unsigned short subdir2 = (fileid / 1000000) % 1000;    // 二级子目录
    unsigned short subdir3 = (fileid / 1000) % 1000;       // 三级子目录
    time_t         curtime = time(NULL);                   // 当前时间戳
    unsigned short postfix = (fileid / 1) % 1000;          // 文件名后缀

    // 格式化完整的文件路径
    if (spath[strlen(spath)-1] == '/')
        snprintf(filepath, PATH_MAX + 1, "%s%03X/%03X/%03X/%lX_%03X",
            spath, subdir1, subdir2, subdir3, curtime, postfix);
    else
        snprintf(filepath, PATH_MAX + 1, "%s/%03X/%03X/%03X/%lX_%03X",
            spath, subdir1, subdir2, subdir3, curtime, postfix);

    return OK;
}

// 接收并保存文件
int service_c::save(acl::socket_stream* conn, char const* appid,
    char const* userid, char const* fileid, long long filesize,
    char const* filepath) const {
    file_c file; // 文件操作对象

    // 打开文件
    if (file.open(filepath, file_c::O_WRITE) != OK)
        return ERROR;

    // 依次将接收到的数据块写入文件
    long long remain = filesize; // 未接收字节数
    char rcvwr[STORAGE_RCVWR_SIZE]; // 接收写入缓冲区
    while (remain) { // 还有未接收数据
        // 接收数据
        long long bytes = std::min(remain, (long long)sizeof(rcvwr));
        long long count = conn->read(rcvwr, bytes);
```

```
        if (count < 0) {
            logger_error("read fail: %s, bytes: %lld, from: %s",
                acl::last_serror(), bytes, conn->get_peer());
            file.close();
            return SOCKET_ERROR;
        }
        // 写入文件
        if (file.write(rcvwr, count) != OK) {
            file.close();
            return ERROR;
        }
        // 未收递减
        remain -= count;
    }

    // 关闭文件
    file.close();

    db_c db; // 数据库访问对象

    // 连接数据库
    if (db.connect() != OK)
        return ERROR;

    // 设置文件ID和路径及大小的对应关系
    if (db.set(appid, userid, fileid, filepath, filesize) != OK) {
        error(conn, -1, "insert database fail, fileid: %s", fileid);
        file.del(filepath);
        return ERROR;
    }

    return OK;
}

// 读取并发送文件
int service_c::send(acl::socket_stream* conn, char const* filepath,
    long long offset, long long size) const {
    file_c file; // 文件操作对象

    // 打开文件
    if (file.open(filepath, file_c::O_READ) != OK)
        return ERROR;

    // 设置偏移
    if (offset && file.seek(offset) != OK) {
        file.close();
        return ERROR;
    }

    // |包体长度|命令|状态|文件内容|
    // |    8   |  1 |  1 |内容大小|
    // 构造响应头
    long long bodylen = size;
    long long headlen = HEADLEN;
    char head[headlen] = {};
    llton(bodylen, head);
```

```
    head[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
    head[BODYLEN_SIZE+COMMAND_SIZE] = 0;

    // 发送响应头
    if (conn->write(head, headlen) < 0) {
        logger_error("write fail: %s, headlen: %lld, to: %s",
            acl::last_serror(), headlen, conn->get_peer());
        file.close();
        return SOCKET_ERROR;
    }

    // 依次将从文件中读取到的数据块作为响应体的一部分发送出去
    long long remain = size; // 未读取字节数
    char rdsnd[STORAGE_RDSND_SIZE]; // 读取发送缓冲区
    while (remain) { // 还有未读取数据
        // 读取文件
        long long count = std::min(remain, (long long)sizeof(rdsnd));
        if (file.read(rdsnd, count) != OK) {
            file.close();
            return ERROR;
        }
        // 发送数据
        if (conn->write(rdsnd, count) < 0) {
            logger_error("write fail: %s, count: %lld, to: %s",
                acl::last_serror(), count, conn->get_peer());
            file.close();
            return SOCKET_ERROR;
        }
        // 未读递减
        remain -= count;
    }

    // 关闭文件
    file.close();

    return OK;
}

////////////////////////////////////////////////////////////////////////

// 应答成功
bool service_c::ok(acl::socket_stream* conn) const {
    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 构造响应
    long long bodylen = 0;
    long long resplen = HEADLEN + bodylen;
    char resp[resplen] = {};
    llton(bodylen, resp);
    resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
    resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;

    // 发送响应
    if (conn->write(resp, resplen) < 0) {
        logger_error("write fail: %s, resplen: %lld, to: %s",
            acl::last_serror(), resplen, conn->get_peer());
```

```
        return false;
    }

    return true;
}

// 应答错误
bool service_c::error(acl::socket_stream* conn, short errnumb,
    char const* format, ...) const {
    // 错误描述
    char errdesc[ERROR_DESC_SIZE];
    va_list ap;
    va_start(ap, format);
    vsnprintf(errdesc, ERROR_DESC_SIZE, format, ap);
    va_end(ap);
    logger_error("%s", errdesc);
    acl::string desc;
    desc.format("[%s] %s", g_hostname.c_str(), errdesc);
    memset(errdesc, 0, sizeof(errdesc));
    strncpy(errdesc, desc.c_str(), ERROR_DESC_SIZE - 1);
    size_t desclen = strlen(errdesc);
    desclen += desclen != 0;

    // |包体长度|命令|状态|错误号|错误描述|
    // |    8   |  1 |  1 |   2  | <=1024 |
    // 构造响应
    long long bodylen = ERROR_NUMB_SIZE + desclen;
    long long resplen = HEADLEN + bodylen;
    char resp[resplen] = {};
    llton(bodylen, resp);
    resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
    resp[BODYLEN_SIZE+COMMAND_SIZE] = STATUS_ERROR;
    ston(errnumb, resp + HEADLEN);
    if (desclen)
        strcpy(resp + HEADLEN + ERROR_NUMB_SIZE, errdesc);

    // 发送响应
    if (conn->write(resp, resplen) < 0) {
        logger_error("write fail: %s, resplen: %lld, to: %s",
            acl::last_serror(), resplen, conn->get_peer());
        return false;
    }

    return true;
}
```

~/TNV/src/04_storage/13_tracker.h

```
// 存储服务器
// 声明跟踪客户机线程类
//
#pragma once

#include <lib_acl.hpp>
//
```

```
// 跟踪客户机线程类
//
class tracker_c: public acl::thread {
public:
    // 构造函数
    tracker_c(char const* taddr);

    // 终止线程
    void stop(void);

protected:
    // 线程过程
    void* run(void);

private:
    // 向跟踪服务器发送加入包
    int join(acl::socket_stream* conn) const;
    // 向跟踪服务器发送心跳包
    int beat(acl::socket_stream* conn) const;

    bool        m_stop;  // 是否终止
    acl::string m_taddr; // 跟踪服务器地址
};
```

~/TNV/src/04_storage/14_tracker.cpp

```
// 存储服务器
// 实现跟踪客户机线程类
//
#include <unistd.h>
#include "02_proto.h"
#include "03_util.h"
#include "01_globals.h"
#include "13_tracker.h"

// 构造函数
tracker_c::tracker_c(char const* taddr): m_stop(false), m_taddr(taddr) {
}

// 终止线程
void tracker_c::stop(void) {
    m_stop = true;
}

// 线程过程
void* tracker_c::run(void) {
    acl::socket_stream conn;

    while (!m_stop) {
        // 连接跟踪服务器
        if (!conn.open(m_taddr, 10, 30)) {
            logger_error("connect tracker fail, taddr: %s",
                m_taddr.c_str());
            sleep(2);
            continue; // 失败重连
```

```
        }

        // 向跟踪服务器发送加入包
        if (join(&conn) != OK) {
            conn.close();
            sleep(2);
            continue; // 失败重连
        }

        time_t last = time(NULL); // 上次心跳
        while (!m_stop) {
            time_t now = time(NULL); // 现在
            // 现在离上次心跳已足够久，再跳一次
            if (now - last >= cfg_interval) {
                // 向跟踪服务器发送心跳包
                if (beat(&conn) != OK) {
                    sleep(2);
                    break; // 失败重连
                }
                last = now;
            }
            sleep(1);
        }

        conn.close();
    }

    return NULL;
}

// 向跟踪服务器发送加入包
int tracker_c::join(acl::socket_stream* conn) const {
    // |包体长度|命令|状态|storage_join_body_t|
    // |    8   |  1 |  1 |     包体长度      |
    // 构造请求
    long long bodylen = sizeof(storage_join_body_t);
    long long requlen = HEADLEN + bodylen;
    char requ[requlen] = {};
    llton(bodylen, requ);
    requ[BODYLEN_SIZE] = CMD_TRACKER_JOIN;
    requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    storage_join_body_t* sjb = (storage_join_body_t*)(requ + HEADLEN);
    strcpy(sjb->sjb_version, g_version);           // 版本
    strcpy(sjb->sjb_groupname, cfg_gpname);        // 组名
    strcpy(sjb->sjb_hostname, g_hostname.c_str()); // 主机名
    ston(cfg_bindport, sjb->sjb_port);             // 端口号
    lton(g_stime, sjb->sjb_stime);                 // 启动时间
    lton(time(NULL), sjb->sjb_jtime);              // 加入时间

    // 发送请求
    if (conn->write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, conn->get_peer());
        return SOCKET_ERROR;
    }

```

```
    // 接收包头
    char head[HEADLEN];
    if (conn->read(head, HEADLEN) < 0) {
        logger_error("read fail: %s, from: %s",
            acl::last_serror(), conn->get_peer());
        return SOCKET_ERROR;
    }

    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 解析包头
    if ((bodylen = ntoll(head)) < 0) { // 包体长度
        logger_error("invalid body length: %lld < 0", bodylen);
        return ERROR;
    }
    int command = head[BODYLEN_SIZE]; // 命令
    int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
    logger("bodylen: %lld, command: %d, status: %d",
        bodylen, command, status);

    // 检查命令
    if (command != CMD_TRACKER_REPLY) {
        logger_error("unknown command: %d", command);
        return ERROR;
    }

    // 应答成功
    if (!status) return OK;

    // |包体长度|命令|状态|错误号|错误描述|
    // |    8   |  1 |  1 |   2  | <=1024 |
    // 检查包体长度
    long long expected = ERROR_NUMB_SIZE + ERROR_DESC_SIZE;
    if (bodylen > expected) {
        logger_error("invalid body length: %lld > %lld", bodylen,
expected);
        return ERROR;
    }

    // 接收包体
    char body[bodylen];
    if (conn->read(body, bodylen) < 0) {
        logger_error("read fail: %s, bodylen: %lld, from: %s",
            acl::last_serror(), bodylen, conn->get_peer());
        return SOCKET_ERROR;
    }

    // 解析包体
    short errnumb = ntos(body);
    char const* errdesc = "";
    if (bodylen > ERROR_NUMB_SIZE)
        errdesc = body + ERROR_NUMB_SIZE;

    logger_error("join fail, errnumb: %d, errdesc: %s",
        errnumb, errdesc);
    return ERROR;
```

```
}

// 向跟踪服务器发送心跳包
int tracker_c::beat(acl::socket_stream* conn) const {
    // |包体长度|命令|状态|storage_beat_body_t|
    // |    8   |  1 |  1 |     包体长度      |
    // 构造请求
    long long bodylen = sizeof(storage_beat_body_t);
    long long requlen = HEADLEN + bodylen;
    char requ[requlen] = {};
    llton(bodylen, requ);
    requ[BODYLEN_SIZE] = CMD_TRACKER_BEAT;
    requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    storage_beat_body_t* sbb = (storage_beat_body_t*)(requ + HEADLEN);
    strcpy(sbb->sbb_groupname, cfg_gpname);        // 组名
    strcpy(sbb->sbb_hostname, g_hostname.c_str()); // 主机名

    // 发送请求
    if (conn->write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, conn->get_peer());
        return SOCKET_ERROR;
    }

    // 接收包头
    char head[HEADLEN];
    if (conn->read(head, HEADLEN) < 0) {
        logger_error("read fail: %s, from: %s",
            acl::last_serror(), conn->get_peer());
        return SOCKET_ERROR;
    }

    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 解析包头
    if ((bodylen = ntoll(head)) < 0) { // 包体长度
        logger_error("invalid body length: %lld < 0", bodylen);
        return ERROR;
    }
    int command = head[BODYLEN_SIZE]; // 命令
    int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
    logger("bodylen: %lld, command: %d, status: %d",
        bodylen, command, status);

    // 检查命令
    if (command != CMD_TRACKER_REPLY) {
        logger_error("unknown command: %d", command);
        return ERROR;
    }

    // 应答成功
    if (!status) return OK;

    // |包体长度|命令|状态|错误号|错误描述|
    // |    8   |  1 |  1 |   2  | <=1024 |
    // 检查包体长度
```

```
    long long expected = ERROR_NUMB_SIZE + ERROR_DESC_SIZE;
    if (bodylen > expected) {
        logger_error("invalid body length: %lld > %lld",
            bodylen, expected);
        return ERROR;
    }

    // 接收包体
    char body[bodylen];
    if (conn->read(body, bodylen) < 0) {
        logger_error("read fail: %s, bodylen: %lld, from: %s",
            acl::last_serror(), bodylen, conn->get_peer());
        return SOCKET_ERROR;
    }

    // 解析包体
    short errnumb = ntos(body);
    char const* errdesc = "";
    if (bodylen > ERROR_NUMB_SIZE)
        errdesc = body + ERROR_NUMB_SIZE;

    logger_error("beat fail, errnumb: %d, errdesc: %s", errnumb, errdesc);
    return ERROR;
}
```

~/TNV/src/04_storage/15_server.h

```
// 存储服务器
// 声明服务器类
//
#pragma once

#include <list>
#include <lib_acl.hpp>
#include "13_tracker.h"
//
// 服务器类
//
class server_c: public acl::master_threads {
protected:
    // 进程切换用户后被调用
    void proc_on_init(void);
    // 子进程意图退出时被调用
    // 返回true，子进程立即退出，否则
    // 若配置项ioctl_quick_abort非0，子进程立即退出，否则
    // 待所有客户机连接都关闭后，子进程再退出
    bool proc_exit_timer(size_t nclients, size_t nthreads);
    // 进程退出前被调用
    void proc_on_exit(void);

    // 线程获得连接时被调用
    // 返回true，连接将被用于后续通信，否则
    // 函数返回后即关闭连接
    bool thread_on_accept(acl::socket_stream* conn);
    // 与线程绑定的连接可读时被调用
```

```
    // 返回true，保持长连接，否则
    // 函数返回后即关闭连接
    bool thread_on_read(acl::socket_stream* conn);
    // 线程读写连接超时时被调用
    // 返回true，继续等待下一次读写，否则
    // 函数返回后即关闭连接
    bool thread_on_timeout(acl::socket_stream* conn);
    // 与线程绑定的连接关闭时被调用
    void thread_on_close(acl::socket_stream* conn);

private:
    std::list<tracker_c*> m_trackers; // 跟踪客户机线程集
};
```

~/TNV/src/04_storage/16_server.cpp

```
// 存储服务器
// 实现服务器类
//
#include <unistd.h>
#include "02_proto.h"
#include "03_util.h"
#include "01_globals.h"
#include "11_service.h"
#include "15_server.h"

// 进程切换用户后被调用
void server_c::proc_on_init(void) {
    // 隶属组名
    if (strlen(cfg_gpname) > STORAGE_GROUPNAME_MAX)
        logger_fatal("groupname too big %lu > %d",
            strlen(cfg_gpname), STORAGE_GROUPNAME_MAX);

    // 绑定端口号
    if (cfg_bindport <= 0)
        logger_fatal("invalid bind port %d <= 0", cfg_bindport);

    // 存储路径表
    if (!cfg_spaths || !*cfg_spaths)
        logger_fatal("storage paths is null");
    split(cfg_spaths, g_spaths);
    if (g_spaths.empty())
        logger_fatal("storage paths is empty");

    // 跟踪服务器地址表
    if (!cfg_taddrs || !*cfg_taddrs)
        logger_fatal("tracker addresses is null");
    split(cfg_taddrs, g_taddrs);
    if (g_taddrs.empty())
        logger_fatal("tracker addresses is empty");

    // ID服务器地址表
    if (!cfg_iaddrs || !*cfg_iaddrs)
        logger_fatal("id addresses is null");
    split(cfg_iaddrs, g_iaddrs);
```

```
    if (g_iaddrs.empty())
        logger_fatal("id addresses is empty");

    // MySQL地址表
    if (!cfg_maddrs || !*cfg_maddrs)
        logger_fatal("mysql addresses is null");
    split(cfg_maddrs, g_maddrs);
    if (g_maddrs.empty())
        logger_fatal("mysql addresses is empty");

    // Redis地址表
    if (!cfg_raddrs || !*cfg_raddrs)
        logger_error("redis addresses is null");
    else {
        split(cfg_raddrs, g_raddrs);
        if (g_raddrs.empty())
            logger_error("redis addresses is empty");
        else {
            // 遍历Redis地址表，尝试创建连接池
            for (std::vector<std::string>::const_iterator raddr =
                g_raddrs.begin(); raddr != g_raddrs.end(); ++raddr)
                if ((g_rconns = new acl::redis_client_pool(
                    raddr->c_str(), cfg_maxconns))) {
                    // 设置Redis连接超时和读写超时
                    g_rconns->set_timeout(cfg_ctimeout, cfg_rtimeout);
                    break;
                }
            if (!g_rconns)
                logger_error("create redis connection pool fail,
cfg_raddrs: %s",
                    cfg_raddrs);
        }
    }

    // 主机名
    char hostname[256+1] = {};
    if (gethostname(hostname, sizeof(hostname) - 1))
        logger_error("call gethostname fail: %s", strerror(errno));
    g_hostname = hostname;

    // 启动时间
    g_stime = time(NULL);

    // 创建并启动连接每台跟踪服务器的客户机线程
    for (std::vector<std::string>::const_iterator taddr = g_taddrs.begin();
        taddr != g_taddrs.end(); ++taddr) {
        tracker_c* tracker = new tracker_c(taddr->c_str());
        tracker->set_detachable(false);
        tracker->start();
        m_trackers.push_back(tracker);
    }

    // 打印配置信息
    logger("cfg_gpname: %s, cfg_spaths: %s, cfg_taddrs: %s, "
        "cfg_iaddrs: %s, cfg_maddrs: %s, cfg_raddrs: %s, "
        "cfg_bindport: %d, cfg_interval: %d, cfg_mtimeout: %d, "
```

```
        "cfg_maxconns: %d, cfg_ctimeout: %d, cfg_rtimeout: %d, "
        "cfg_ktimeout: %d",
        cfg_gpname, cfg_spaths, cfg_taddrs,
        cfg_iaddrs, cfg_maddrs, cfg_raddrs,
        cfg_bindport, cfg_interval, cfg_mtimeout,
        cfg_maxconns, cfg_ctimeout, cfg_rtimeout,
        cfg_ktimeout);
}

// 子进程意图退出时被调用
// 返回true，子进程立即退出，否则
// 若配置项ioctl_quick_abort非0，子进程立即退出，否则
// 待所有客户机连接都关闭后，子进程再退出
bool server_c::proc_exit_timer(size_t nclients, size_t nthreads) {
    for (std::list<tracker_c*>::iterator tracker = m_trackers.begin();
        tracker != m_trackers.end(); ++tracker)
        // 终止跟踪客户机线程
        (*tracker)->stop();

    if (!nclients || !nthreads) {
        logger("nclients: %lu, nthreads: %lu", nclients, nthreads);
        return true;
    }

    return false;
}

// 进程退出前被调用
void server_c::proc_on_exit(void) {
    for (std::list<tracker_c*>::iterator tracker = m_trackers.begin();
        tracker != m_trackers.end(); ++tracker) {
        // 回收跟踪客户机线程
        if (!(*tracker)->wait(NULL))
            logger_error("wait thread #%lu fail", (*tracker)->thread_id());
        // 销毁跟踪客户机线程
        delete *tracker;
    }

    m_trackers.clear();

    // 销毁Redis连接池
    if (g_rconns) {
        delete g_rconns;
        g_rconns = NULL;
    }
}

// 线程获得连接时被调用
// 返回true，连接将被用于后续通信，否则
// 函数返回后即关闭连接
bool server_c::thread_on_accept(acl::socket_stream* conn) {
    logger("connect, from: %s", conn->get_peer());
    return true;
}

// 与线程绑定的连接可读时被调用
```

```
// 返回true，保持长连接，否则
// 函数返回后即关闭连接
bool server_c::thread_on_read(acl::socket_stream* conn) {
    // 接收包头
    char head[HEADLEN];
    if (conn->read(head, HEADLEN) < 0) {
        if (conn->eof())
            logger("connection has been closed, from: %s",
                conn->get_peer());
        else
            logger_error("read fail: %s, from: %s",
                acl::last_serror(), conn->get_peer());
        return false;
    }

    // 业务处理
    service_c service;
    return service.business(conn, head);
}

// 线程读写连接超时时被调用
// 返回true，继续等待下一次读写，否则
// 函数返回后即关闭连接
bool server_c::thread_on_timeout(acl::socket_stream* conn) {
    logger("read timeout, from: %s", conn->get_peer());
    return true;
}

// 与线程绑定的连接关闭时被调用
void server_c::thread_on_close(acl::socket_stream* conn) {
    logger("client disconnect, from: %s", conn->get_peer());
}
```

~/TNV/src/04_storage/17_main.cpp

```
// 存储服务器
// 定义主函数
//
#include "01_globals.h"
#include "15_server.h"

int main(void) {
    // 初始化ACL库
    acl::acl_cpp_init();
    acl::log::stdout_open(true);

    // 创建并运行服务器
    server_c& server = acl::singleton2<server_c>::get_instance();
    server.set_cfg_str(cfg_str);
    server.set_cfg_int(cfg_int);
    server.run_alone("127.0.0.1:23000", "../etc/storage.cfg");

    return 0;
}
```

~/TNV/src/04_storage/Makefile

```
PROJ   = ../../bin/storage
OBJS   = $(patsubst %.cpp, %.o, $(wildcard ../01_common/*.cpp *.cpp))
CC     = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -Wall \
         -I/usr/include/acl-lib/acl_cpp \
         `mysql_config --cflags` \
         -I../01_common
LIBS   = -pthread -lacl_all `mysql_config --libs`

all: $(PROJ)

$(PROJ): $(OBJS)
    $(LINK) $^ $(LIBS) -o $@

.cpp.o:
    $(CC) $(CFLAGS) $^ -o $@

clean:
    $(RM) $(PROJ) $(OBJS)
```

~/TNV/etc/storage.cfg

```
service storage {
    # 隶属组名
    tnv_group_name = group001
    # 存储路径表
    tnv_store_paths = ../data
    # 跟踪服务器地址表
    tnv_tracker_addrs = 127.0.0.1:21000
    # ID服务器地址表
    tnv_ids_addrs = 127.0.0.1:22000
    # MySQL地址表
    mysql_addrs = 127.0.0.1
    # Redis地址表
    redis_addrs = 127.0.0.1:6379
    # 绑定端口号
    tnv_storage_port = 23000
    # 心跳间隔秒数
    tnv_heart_beat_interval = 10
    # MySQL读写超时
    mysql_rw_timeout = 30
    # Redis连接池最大连接数
    redis_max_conn_num = 600
    # Redis连接超时
    redis_conn_timeout = 10
    # Redis读写超时
    redis_rw_timeout = 10
    # Redis键超时
    redis_key_timeout = 60
```

```
}
```

~/TNV/sql/storage.sql

```
DROP DATABASE IF EXISTS tnv_storagedb;
CREATE DATABASE tnv_storagedb;
USE tnv_storagedb;

CREATE TABLE `t_file_01` (
  `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
  `appid` varchar(32) DEFAULT NULL,
  `userid` varchar(128) DEFAULT NULL,
  `status` tinyint(4) DEFAULT NULL,
  `file_path` varchar(512) DEFAULT NULL,
  `file_size` bigint(20) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `t_file_02` (
  `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
  `appid` varchar(32) DEFAULT NULL,
  `userid` varchar(128) DEFAULT NULL,
  `status` tinyint(4) DEFAULT NULL,
  `file_path` varchar(512) DEFAULT NULL,
  `file_size` bigint(20) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `t_file_03` (
  `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
  `appid` varchar(32) DEFAULT NULL,
  `userid` varchar(128) DEFAULT NULL,
  `status` tinyint(4) DEFAULT NULL,
  `file_path` varchar(512) DEFAULT NULL,
  `file_size` bigint(20) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```