2 跟踪服务器

~/TNV/src/02_tracker/01_globals.h

```
// 跟踪服务器
// 声明全局变量
//
#pragma once

#include <vector>
#include <string>
#include <map>
#include <list>
#include <lib_acl.hpp>
#include "01_types.h"
//
// 配置信息
//
extern char* cfg_appids; // 应用ID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_interval; // 存储服务器状态检查间隔秒数
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_appids; // 应用ID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 std::map<std::string,
    std::list<storage_info_t> > g_groups; // 组表
extern pthread_mutex_t g_mutex; // 互斥锁
```

~/TNV/src/02_tracker/02_globals.cpp

```
// 跟踪服务器
// 定义全局变量
//
#include "01_globals.h"
//
// 配置信息
//
char* cfg_appids; // 应用ID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_apps_id", "tnvideo",        &cfg_appids},
    {"mysql_addrs", "127.0.0.1",      &cfg_maddrs},
    {"redis_addrs", "127.0.0.1:6379", &cfg_raddrs},
    {0, 0, 0}};

int cfg_interval; // 存储服务器状态检测间隔秒数
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
    {"check_active_interval", 120, &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_appids; // 应用ID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
std::map<std::string,
    std::list<storage_info_t> > g_groups; // 组表
pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁
```

~/TNV/src/02_tracker/03_cache.h

```
// 跟踪服务器
// 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/02_tracker/04_cache.cpp

```cpp
// 跟踪服务器
// 实现缓存类
//
#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 tracker_key;
    tracker_key.format("%s:%s", TRACKER_REDIS_PREFIX, key);

    // 检查Redis连接池
    if (!g_rconns) {
        logger_warn("redis connection pool is null, key: %s",
            tracker_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",
            tracker_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(tracker_key.c_str(), value)) {
        logger_warn("get cache fail, key: %s", tracker_key.c_str());
        g_rconns->put(rconn, false);
        return ERROR;
    }

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

    logger("get cache ok, key: %s, value: %s",
        tracker_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 tracker_key;
    tracker_key.format("%s:%s", TRACKER_REDIS_PREFIX, key);

    // 检查Redis连接池
    if (!g_rconns) {
        logger_warn("redis connection pool is null, key: %s",
            tracker_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",
            tracker_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(tracker_key.c_str(), value, timeout)) {
        logger_warn("set cache fail, key: %s, value: %s, timeout: %d",
            tracker_key.c_str(), value, timeout);
        g_rconns->put(rconn, false);
        return ERROR;
    }
    logger("set cache ok, key: %s, value: %s, timeout: %d",
        tracker_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 tracker_key;
    tracker_key.format("%s:%s", TRACKER_REDIS_PREFIX, key);

    // 检查Redis连接池
    if (!g_rconns) {
        logger_warn("redis connection pool is null, key: %s",
            tracker_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",
            tracker_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客户机删除指定键值对
    if (!redis.del_one(tracker_key.c_str())) {
        logger_warn("delete cache fail, key: %s", tracker_key.c_str());
        g_rconns->put(rconn, false);
        return ERROR;
    }
    logger("delete cache ok, key: %s", tracker_key.c_str());
    g_rconns->put(rconn, true);

    return OK;
}
```

~/TNV/src/02_tracker/05_db.h

```
// 跟踪服务器
// 声明数据库访问类
//
#pragma once

#include <string>
#include <vector>
#include <mysql.h>
//
// 数据库访问类
//
class db_c {
public:
    // 构造函数
    db_c(void);
    // 析构函数
    ~db_c(void);

    // 连接数据库
    int connect(void);

    // 根据用户ID获取其对应的组名
    int get(char const* userid, std::string& groupname) const;
    // 设置用户ID和组名的对应关系
    int set(char const* appid, char const* userid,
        char const* groupname) const;
    // 获取全部组名
    int get(std::vector<std::string>& groupnames) const;

private:
    MYSQL* m_mysql; // MySQL对象
```

```
};
```

~/TNV/src/02_tracker/06_db.cpp

```
// 跟踪服务器
// 实现数据库访问类
//
#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_trackerdb", 0, NULL, 0)))
            return OK;

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

// 根据用户ID获取其对应的组名
int db_c::get(char const* userid, std::string& groupname) const {
    // 先尝试从缓存中获取与用户ID对应的组名
    cache_c cache;
    acl::string key;
    key.format("userid:%s", userid);
    acl::string value;
    if (cache.get(key, value) == OK) {
        groupname = value.c_str();
        return OK;
    }

    // 缓存中没有再查询数据库
```

```
52      acl::string sql;
53      sql.format("SELECT group_name FROM t_router WHERE userid='%s';",
54          userid);
55      if (mysql_query(m_mysql, sql.c_str())) {
56          logger_error("query database fail: %s, sql: %s",
57              mysql_error(m_mysql), sql.c_str());
58          return ERROR;
59      }
60
61      // 获取查询结果
62      MYSQL_RES* res = mysql_store_result(m_mysql);
63      if (!res) {
64          logger_error("result is null: %s, sql: %s",
65              mysql_error(m_mysql), sql.c_str());
66          return ERROR;
67      }
68
69      // 获取结果记录
70      MYSQL_ROW row = mysql_fetch_row(res);
71      if (!row)
72          logger_warn("result is empty: %s, sql: %s",
73              mysql_error(m_mysql), sql.c_str());
74      else {
75          groupname = row[0];
76
77          // 将用户ID和组名的对应关系保存在缓存中
78          cache.set(key, groupname.c_str());
79      }
80
81      return OK;
82  }
83
84  // 设置用户ID和组名的对应关系
85  int db_c::set(char const* appid, char const* userid,
86      char const* groupname) const {
87      // 插入一条记录
88      acl::string sql;
89      sql.format("INSERT INTO t_router SET "
90          "appid='%s', userid='%s', group_name='%s';",
91          appid, userid, groupname);
92      if (mysql_query(m_mysql, sql.c_str())) {
93          logger_error("insert database fail: %s, sql: %s",
94              mysql_error(m_mysql), sql.c_str());
95          return ERROR;
96      }
97
98      // 检查插入结果
99      MYSQL_RES* res = mysql_store_result(m_mysql);
100     if (!res && mysql_field_count(m_mysql)) {
101         logger_error("insert database fail: %s, sql: %s",
102             mysql_error(m_mysql), sql.c_str());
103         return ERROR;
104     }
105
106     return OK;
107 }
```

```
108
109 // 获取全部组名
110 int db_c::get(std::vector<std::string>& groupnames) const {
111     // 查询全部组名
112     acl::string sql;
113     sql.format("SELECT group_name FROM t_groups_info;");
114     if (mysql_query(m_mysql, sql.c_str())) {
115         logger_error("query database fail: %s, sql: %s",
116             mysql_error(m_mysql), sql.c_str());
117         return ERROR;
118     }
119
120     // 获取查询结果
121     MYSQL_RES* res = mysql_store_result(m_mysql);
122     if (!res) {
123         logger_error("result is null: %s, sql: %s",
124             mysql_error(m_mysql), sql.c_str());
125         return ERROR;
126     }
127
128     // 获取结果记录
129     int nrows = mysql_num_rows(res);
130     for (int i = 0; i < nrows; ++i) {
131         MYSQL_ROW row = mysql_fetch_row(res);
132         if (!row)
133             break;
134         groupnames.push_back(row[0]);
135     }
136
137     return OK;
138 }
```

~/TNV/src/02_tracker/07_service.h

```
1  // 跟踪服务器
2  // 声明业务服务类
3  //
4  #pragma once
5
6  #include <lib_acl.hpp>
7  #include "01_types.h"
8  //
9  // 业务服务类
10 //
11 class service_c {
12 public:
13     // 业务处理
14     bool business(acl::socket_stream* conn, char const* head) const;
15
16 private:
17     // 处理来自存储服务器的加入包
18     bool join(acl::socket_stream* conn, long long bodylen) const;
19     // 处理来自存储服务器的心跳包
20     bool beat(acl::socket_stream* conn, long long bodylen) const;
21     // 处理来自客户机的获取存储服务器地址列表请求
```

```
    bool saddrs(acl::socket_stream* conn, long long bodylen) const;
    // 处理来自客户机的获取组列表请求
    bool groups(acl::socket_stream* conn) const;

    // 将存储服务器加入组表
    int join(storage_join_t const* sj, char const* saddr) const;

    // 将存储服务器标为活动
    int beat(char const* groupname, char const* hostname,
        char const* saddr) const;

    // 响应客户机存储服务器地址列表
    int saddrs(acl::socket_stream* conn,
        char const* appid, char const* userid) const;
    // 根据用户ID获取其对应的组名
    int group_of_user(char const* appid,
        char const* userid, std::string& groupname) const;
    // 根据组名获取存储服务器地址列表
    int saddrs_of_group(char const* groupname,
        std::string& saddrs) const;

    // 应答成功
    bool ok(acl::socket_stream* conn) const;
    // 应答错误
    bool error(acl::socket_stream* conn, short errnumb,
        char const* format, ...) const;
};
```

~/TNV/src/02_tracker/08_service.cpp

```
// 跟踪服务器
// 实现业务服务类
//
#include <algorithm>
#include "02_proto.h"
#include "03_util.h"
#include "01_globals.h"
#include "05_db.h"
#include "07_service.h"

// 业务处理
bool service_c::business(acl::socket_stream* conn,
    char const* head) const {
    // |包体长度|命令|状态|  包体  |
    // |    8   |  1 |  1 |包体长度|
    // 解析包头
    long long bodylen = ntoll(head); // 包体长度
    if (bodylen < 0) {
        error(conn, -1, "invalid body length: %lld < 0", bodylen);
        return false;
    }
    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);
```

```
26
27      bool result;
28
29      // 根据命令执行具体业务处理
30      switch (command) {
31          case CMD_TRACKER_JOIN:
32              // 处理来自存储服务器的加入包
33              result = join(conn, bodylen);
34              break;
35
36          case CMD_TRACKER_BEAT:
37              // 处理来自存储服务器的心跳包
38              result = beat(conn, bodylen);
39              break;
40
41          case CMD_TRACKER_SADDRS:
42              // 处理来自客户机的获取存储服务器地址列表请求
43              result = saddrs(conn, bodylen);
44              break;
45
46          case CMD_TRACKER_GROUPS:
47              // 处理来自客户机的获取组列表请求
48              result = groups(conn);
49              break;
50
51          default:
52              error(conn, -1, "unknown command: %d", command);
53              return false;
54      }
55
56      return result;
57  }
58
59  ////////////////////////////////////////////////////////////////////////
60
61  // 处理来自存储服务器的加入包
62  bool service_c::join(acl::socket_stream* conn,
63      long long bodylen) const {
64      // |包体长度|命令|状态|storage_join_body_t|
65      // |    8   |  1 |  1 |     包体长度      |
66      // 检查包体长度
67      long long expected = sizeof(storage_join_body_t); // 期望包体长度
68      if (bodylen != expected) {
69          error(conn, -1, "invalid body length: %lld != %lld",
70              bodylen, expected);
71          return false;
72      }
73
74      // 接收包体
75      char body[bodylen];
76      if (conn->read(body, bodylen) < 0) {
77          logger_error("read fail: %s, bodylen: %lld, from: %s",
78              acl::last_serror(), bodylen, conn->get_peer());
79          return false;
80      }
81
```

```
    // 解析包体
    storage_join_t sj;
    storage_join_body_t* sjb = (storage_join_body_t*)body;
    // 版本
    strcpy(sj.sj_version, sjb->sjb_version);
    // 组名
    strcpy(sj.sj_groupname, sjb->sjb_groupname);
    if (valid(sj.sj_groupname) != OK) {
        error(conn, -1, "invalid groupname: %s", sj.sj_groupname);
        return false;
    }
    // 主机名
    strcpy(sj.sj_hostname, sjb->sjb_hostname);
    // 端口号
    sj.sj_port = ntos(sjb->sjb_port);
    if (!sj.sj_port) {
        error(conn, -1, "invalid port: %u", sj.sj_port);
        return false;
    }
    // 启动时间
    sj.sj_stime = ntol(sjb->sjb_stime);
    // 加入时间
    sj.sj_jtime = ntol(sjb->sjb_jtime);
    logger("storage join, version: %s, groupname: %s, "
        "hostname: %s, port: %u, stime: %s, jtime: %s",
        sj.sj_version, sj.sj_groupname,
        sj.sj_hostname, sj.sj_port,
        std::string(ctime(&sj.sj_stime)).c_str(),
        std::string(ctime(&sj.sj_jtime)).c_str());

    // 将存储服务器加入组表
    if (join(&sj, conn->get_peer()) != OK) {
        error(conn, -1, "join into groups fail");
        return false;
    }

    return ok(conn);
}

// 处理来自存储服务器的心跳包
bool service_c::beat(acl::socket_stream* conn,
    long long bodylen) const {
    // |包体长度|命令|状态|storage_beat_body_t|
    // |    8   |  1 |  1 |     包体长度      |
    // 检查包体长度
    long long expected = sizeof(storage_beat_body_t); // 期望包体长度
    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
    storage_beat_body_t* sbb = (storage_beat_body_t*)body;
    // 组名
    char groupname[STORAGE_GROUPNAME_MAX+1];
    strcpy(groupname, sbb->sbb_groupname);
    // 主机名
    char hostname[STORAGE_HOSTNAME_MAX+1];
    strcpy(hostname, sbb->sbb_hostname);
    logger("storage beat, groupname: %s, hostname: %s",
        groupname, hostname);

    // 将存储服务器标为活动
    if (beat(groupname, hostname, conn->get_peer()) != OK) {
        error(conn, -1, "mark storage as active fail");
        return false;
    }

    return ok(conn);
}

// 处理来自客户机的获取存储服务器地址列表请求
bool service_c::saddrs(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);

    // 响应客户机存储服务器地址列表
    if (saddrs(conn, appid, userid) != OK)
        return false;
```

```
194
195     return true;
196 }
197
198 // 处理来自客户机的获取组列表请求
199 bool service_c::groups(acl::socket_stream* conn) const {
200     // 互斥锁加锁
201     if ((errno = pthread_mutex_lock(&g_mutex))) {
202         logger_error("call pthread_mutex_lock fail: %s",
203             strerror(errno));
204         return false;
205     }
206
207     acl::string gps; // 全组字符串
208     gps.format("         COUNT OF GROUPS: %lu\n", g_groups.size());
209
210     // 遍历组表中的每一个组
211     for (std::map<std::string, std::list<storage_info_t> >::
212         const_iterator group = g_groups.begin();
213         group != g_groups.end(); ++group) {
214         acl::string grp; // 单组字符串
215         grp.format("               GROUPNAME: %s\n"
216             "       COUNT OF STORAGES: %lu\n"
217             "COUNT OF ACTIVE STORAGES: %s\n",
218             group->first.c_str(),
219             group->second.size(),
220             "%d");
221
222         int act = 0; // 活动存储服务器数
223
224         // 遍历该组中的每一台存储服务器
225         for (std::list<storage_info_t>::const_iterator si =
226             group->second.begin(); si != group->second.end(); ++si) {
227             acl::string stg; // 存储服务器字符串
228             stg.format("                 VERSION: %s\n"
229                 "               HOSTNAME: %s\n"
230                 "                ADDRESS: %s:%u\n"
231                 "           STARTUP TIME: %s"
232                 "              JOIN TIME: %s"
233                 "              BEAT TIME: %s"
234                 "                 STATUS: ",
235                 si->si_version,
236                 si->si_hostname,
237                 si->si_addr, si->si_port,
238                 std::string(ctime(&si->si_stime)).c_str(),
239                 std::string(ctime(&si->si_jtime)).c_str(),
240                 std::string(ctime(&si->si_btime)).c_str());
241
242             switch (si->si_status) {
243                 case STORAGE_STATUS_OFFLINE:
244                     stg += "OFFLINE";
245                     break;
246                 case STORAGE_STATUS_ONLINE:
247                     stg += "ONLINE";
248                     break;
249                 case STORAGE_STATUS_ACTIVE:
```

```
                    stg += "ACTIVE";
                    ++act;
                    break;
                default:
                    stg += "UNKNOWN";
                    break;
            }

            grp += stg + "\n";
        }

        gps += grp.format(grp, act);
    }

    gps = gps.left(gps.size() - 1);

    // 互斥锁解锁
    if ((errno = pthread_mutex_unlock(&g_mutex))) {
        logger_error("call pthread_mutex_unlock fail: %s",
            strerror(errno));
        return false;
    }

    // |包体长度|命令|状态| 组列表 |
    // |    8   |  1 |  1 |包体长度|
    // 构造响应
    long long bodylen = gps.size() + 1;
    long long resplen = HEADLEN + bodylen;
    char resp[resplen] = {};
    llton(bodylen, resp);
    resp[BODYLEN_SIZE] = CMD_TRACKER_REPLY;
    resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    strcpy(resp + HEADLEN, gps.c_str());

    // 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;
}

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

// 将存储服务器加入组表
int service_c::join(storage_join_t const* sj, char const* saddr) const {
    // 互斥锁加锁
    if ((errno = pthread_mutex_lock(&g_mutex))) {
        logger_error("call pthread_mutex_lock fail: %s",
            strerror(errno));
        return ERROR;
    }

    // 在组表中查找待加入存储服务器所隶属的组
```

```
306     std::map<std::string, std::list<storage_info_t> >::iterator
307         group = g_groups.find(sj->sj_groupname);
308     if (group != g_groups.end()) { // 若找到该组
309         // 遍历该组的存储服务器列表
310         std::list<storage_info_t>::iterator si;
311         for (si = group->second.begin();
312             si != group->second.end(); ++si)
313             // 若待加入存储服务器已在该列表中
314             if (!strcmp(si->si_hostname, sj->sj_hostname) &&
315                 !strcmp(si->si_addr, saddr)) {
316                 // 更新该列表中的相应记录
317                 strcpy(si->si_version, sj->sj_version); // 版本
318                 si->si_port   = sj->sj_port;            // 端口号
319                 si->si_stime  = sj->sj_stime;           // 启动时间
320                 si->si_jtime  = sj->sj_jtime;           // 加入时间
321                 si->si_btime  = sj->sj_jtime;           // 心跳时间
322                 si->si_status = STORAGE_STATUS_ONLINE;  // 状态
323                 break;
324             }
325         // 若待加入存储服务器不在该列表中
326         if (si == group->second.end()) {
327             // 将待加入存储服务器加入该列表
328             storage_info_t si;
329             strcpy(si.si_version,  sj->sj_version);  // 版本
330             strcpy(si.si_hostname, sj->sj_hostname); // 主机名
331             strcpy(si.si_addr,     saddr);           // IP地址
332             si.si_port   = sj->sj_port;             // 端口号
333             si.si_stime  = sj->sj_stime;            // 启动时间
334             si.si_jtime  = sj->sj_jtime;            // 加入时间
335             si.si_btime  = sj->sj_jtime;            // 心跳时间
336             si.si_status = STORAGE_STATUS_ONLINE;   // 状态
337             group->second.push_back(si);
338         }
339     }
340     else { // 若没有该组
341         // 将待加入存储服务器所隶属的组加入组表
342         g_groups[sj->sj_groupname] = std::list<storage_info_t>();
343         // 将待加入存储服务器加入该组的存储服务器列表
344         storage_info_t si;
345         strcpy(si.si_version,  sj->sj_version);  // 版本
346         strcpy(si.si_hostname, sj->sj_hostname); // 主机名
347         strcpy(si.si_addr,     saddr);           // IP地址
348         si.si_port   = sj->sj_port;             // 端口号
349         si.si_stime  = sj->sj_stime;            // 启动时间
350         si.si_jtime  = sj->sj_jtime;            // 加入时间
351         si.si_btime  = sj->sj_jtime;            // 心跳时间
352         si.si_status = STORAGE_STATUS_ONLINE;   // 状态
353         g_groups[sj->sj_groupname].push_back(si);
354     }
355
356     // 互斥锁解锁
357     if ((errno = pthread_mutex_unlock(&g_mutex))) {
358         logger_error("call pthread_mutex_unlock fail: %s",
359             strerror(errno));
360         return ERROR;
361     }
```

```
362
363     return OK;
364 }
365
366 // 将存储服务器标为活动
367 int service_c::beat(char const* groupname, char const* hostname,
368     char const* saddr) const {
369     // 互斥锁加锁
370     if ((errno = pthread_mutex_lock(&g_mutex))) {
371         logger_error("call pthread_mutex_lock fail: %s",
372             strerror(errno));
373         return ERROR;
374     }
375
376     int result = OK;
377
378     // 在组表中查找待标记存储服务器所隶属的组
379     std::map<std::string, std::list<storage_info_t> >::iterator
380         group = g_groups.find(groupname);
381     if (group != g_groups.end()) { // 若找到该组
382         // 遍历该组的存储服务器列表
383         std::list<storage_info_t>::iterator si;
384         for (si = group->second.begin();
385             si != group->second.end(); ++si)
386             // 若待标记存储服务器已在该列表中
387             if (!strcmp(si->si_hostname, hostname) &&
388                 !strcmp(si->si_addr, saddr)) {
389                 // 更新该列表中的相应记录
390                 si->si_btime  = time(NULL);             // 心跳时间
391                 si->si_status = STORAGE_STATUS_ACTIVE; // 状态
392                 break;
393             }
394         // 若待标记存储服务器不在该列表中
395         if (si == group->second.end()) {
396             logger_error("storage not found, groupname: %s, "
397                 "hostname: %s, saddr: %s", groupname, hostname, saddr);
398             result = ERROR;
399         }
400     }
401     else { // 若没有该组
402         logger_error("group not found, groupname: %s", groupname);
403         result = ERROR;
404     }
405
406     // 互斥锁解锁
407     if ((errno = pthread_mutex_unlock(&g_mutex))) {
408         logger_error("call pthread_mutex_unlock fail: %s",
409             strerror(errno));
410         return ERROR;
411     }
412
413     return result;
414 }
415
416 // 响应客户机存储服务器地址列表
417 int service_c::saddrs(acl::socket_stream* conn,
```

```
    char const* appid, char const* userid) const {
    // 应用ID是否合法
    if (valid(appid) != OK) {
        error(conn, -1, "invalid appid: %s", appid);
        return ERROR;
    }

    // 应用ID是否存在
    if (std::find(g_appids.begin(), g_appids.end(),
        appid) == g_appids.end()) {
        error(conn, -1, "unknown appid: %s", appid);
        return ERROR;
    }

    // 根据用户ID获取其对应的组名
    std::string groupname;
    if (group_of_user(appid, userid, groupname) != OK) {
        error(conn, -1, "get groupname fail");
        return ERROR;
    }

    // 根据组名获取存储服务器地址列表
    std::string saddrs;
    if (saddrs_of_group(groupname.c_str(), saddrs) != OK) {
        error(conn, -1, "get storage address fail");
        return ERROR;
    }

    logger("appid: %s, userid: %s, groupname: %s, saddrs: %s",
        appid, userid, groupname.c_str(), saddrs.c_str());

    // |包体长度|命令|状态|组名|存储服务器地址列表|
    // |    8   |  1 |  1 |        包体长度         |
    // 构造响应
    long long bodylen = STORAGE_GROUPNAME_MAX + 1 + saddrs.size() + 1;
    long long resplen = HEADLEN + bodylen;
    char resp[resplen] = {};
    llton(bodylen, resp);
    resp[BODYLEN_SIZE] = CMD_TRACKER_REPLY;
    resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;
    strncpy(resp + HEADLEN, groupname.c_str(), STORAGE_GROUPNAME_MAX);
    strcpy(resp + HEADLEN + STORAGE_GROUPNAME_MAX + 1, saddrs.c_str());

    // 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 ERROR;
    }

    return OK;
}

// 根据用户ID获取其对应的组名
int service_c::group_of_user(char const* appid,
    char const* userid, std::string& groupname) const {
```

```
474     db_c db; // 数据库访问对象
475 
476     // 连接数据库
477     if (db.connect() != OK)
478         return ERROR;
479 
480     // 根据用户ID获取其对应的组名
481     if (db.get(userid, groupname) != OK)
482         return ERROR;
483 
484     // 组名为空表示该用户没有组，为其随机分配一个
485     if (groupname.empty()) {
486         logger("groupname is empty, appid: %s, userid: %s, allocate one",
487             appid, userid);
488 
489         // 获取全部组名
490         std::vector<std::string> groupnames;
491         if (db.get(groupnames) != OK)
492             return ERROR;
493         if (groupnames.empty()) {
494             logger_error("groupnames is empty, appid: %s, userid: %s",
495                 appid, userid);
496             return ERROR;
497         }
498 
499         // 随机抽取组名
500         srand(time(NULL));
501         groupname = groupnames[rand() % groupnames.size()];
502 
503         // 设置用户ID和组名的对应关系
504         if (db.set(appid, userid, groupname.c_str()) != OK)
505             return ERROR;
506     }
507 
508     return OK;
509 }
510 
511 // 根据组名获取存储服务器地址列表
512 int service_c::saddrs_of_group(char const* groupname,
513     std::string& saddrs) const {
514     // 互斥锁加锁
515     if ((errno = pthread_mutex_lock(&g_mutex))) {
516         logger_error("call pthread_mutex_lock fail: %s",
517             strerror(errno));
518         return ERROR;
519     }
520 
521     int result = OK;
522 
523     // 根据组名在组表中查找特定组
524     std::map<std::string, std::list<storage_info_t> >::const_iterator
525         group = g_groups.find(groupname);
526     if (group != g_groups.end()) { // 若找到该组
527         if (!group->second.empty()) { // 若该组的存储服务器列表非空
528             // 在该组的存储服务器列表中，从随机位置开
529             // 始最多抽取三台处于活动状态的存储服务器
```

```
            srand(time(NULL));
            int nsis = group->second.size();
            int nrand = rand() % nsis;
            std::list<storage_info_t>::const_iterator si =
                group->second.begin();
            int nacts = 0;
            for (int i = 0; i < nsis + nrand; ++i, ++si) {
                if (si == group->second.end())
                    si = group->second.begin();
                logger("i: %d, nrand: %d, addr: %s, port: %u, "
                    "status: %d", i, nrand, si->si_addr, si->si_port,
                    si->si_status);
                if (i >= nrand && si->si_status ==
                    STORAGE_STATUS_ACTIVE) {
                    char saddr[256];
                    sprintf(saddr, "%s:%d", si->si_addr, si->si_port);
                    saddrs += saddr;
                    saddrs += ";";
                    if (++nacts >= 3)
                        break;
                }
            }
            if (!nacts) { // 若没有处于活动状态的存储服务器
                logger_error("no active storage in group %s",
                    groupname);
                result = ERROR;
            }
        }
        else { // 若该组的存储服务器列表为空
            logger_error("no storage in group %s", groupname);
            result = ERROR;
        }
    }
    else { // 若没有该组
        logger_error("not found group %s", groupname);
        result = ERROR;
    }

    // 互斥锁解锁
    if ((errno = pthread_mutex_unlock(&g_mutex))) {
        logger_error("call pthread_mutex_unlock fail: %s",
            strerror(errno));
        return ERROR;
    }

    return result;
}

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

// 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_TRACKER_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_TRACKER_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/02_tracker/09_status.h

```
// 跟踪服务器
// 声明存储服务器状态检查线程类
//
#pragma once

#include <lib_acl.hpp>
//
// 存储服务器状态检查线程类
//
class status_c: public acl::thread {
public:
    // 构造函数
    status_c(void);

    // 终止线程
    void stop(void);

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

private:
    // 检查存储服务器状态
    int check(void) const;

    bool m_stop; // 是否终止
};
```

~/TNV/src/02_tracker/10_status.cpp

```
// 跟踪服务器
// 实现存储服务器状态检查线程类
//
#include <unistd.h>
#include "01_globals.h"
#include "09_status.h"

// 构造函数
status_c::status_c(void): m_stop(false) {
}

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

// 线程过程
void* status_c::run(void) {
    for (time_t last = time(NULL); !m_stop; sleep(1)) {
        time_t now = time(NULL); // 现在

        // 若现在距离最近一次检查存储服务器状态已足够久
```

```
        if (now - last >= cfg_interval) {
            check(); // 检查存储服务器状态
            last = now; // 更新最近一次检查时间
        }
    }

    return NULL;
}

// 检查存储服务器状态
int status_c::check(void) const {
    time_t now = time(NULL); // 现在

    // 互斥锁加锁
    if ((errno = pthread_mutex_lock(&g_mutex))) {
        logger_error("call pthread_mutex_lock fail: %s",
            strerror(errno));
        return ERROR;
    }

    // 遍历组表中的每一个组
    for (std::map<std::string, std::list<storage_info_t> >::iterator
        group = g_groups.begin(); group != g_groups.end(); ++group)
        // 遍历该组中的每一台存储服务器
        for (std::list<storage_info_t>::iterator si =
            group->second.begin(); si != group->second.end(); ++si)
            // 若该存储服务器心跳停止太久
            if (now - si->si_btime >= cfg_interval)
                // 则将其状态标记为离线
                si->si_status = STORAGE_STATUS_OFFLINE;

    // 互斥锁解锁
    if ((errno = pthread_mutex_unlock(&g_mutex))) {
        logger_error("call pthread_mutex_unlock fail: %s",
            strerror(errno));
        return ERROR;
    }

    return OK;
}
```

~/TNV/src/02_tracker/11_server.h

```
// 跟踪服务器
// 声明服务器类
//
#pragma once

#include <lib_acl.hpp>
#include "09_status.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:
    status_c* m_status; // 存储服务器状态检查线程
};
```

~/TNV/src/02_tracker/12_server.cpp

```
// 跟踪服务器
// 实现服务器类
//
#include <unistd.h>
#include "02_proto.h"
#include "03_util.h"
#include "01_globals.h"
#include "07_service.h"
#include "11_server.h"

// 进程切换用户后被调用
void server_c::proc_on_init(void) {
    // 应用ID表
    if (!cfg_appids || !*cfg_appids)
        logger_fatal("application ids is null");
    split(cfg_appids, g_appids);
    if (g_appids.empty())
        logger_fatal("application ids 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;

    // 创建并启动存储服务器状态检查线程
    if ((m_status = new status_c)) {
        m_status->set_detachable(false);
        m_status->start();
    }

    // 打印配置信息
    logger("cfg_appids: %s, cfg_maddrs: %s, cfg_raddrs: %s, "
        "cfg_interval: %d, cfg_mtimeout: %d, cfg_maxconns: %d, "
        "cfg_ctimeout: %d, cfg_rtimeout: %d, cfg_ktimeout: %d",
        cfg_appids, cfg_maddrs, cfg_raddrs,
        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) {
    // 终止存储服务器状态检查线程
    m_status->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) {
    // 回收存储服务器状态检测线程
    if (!m_status->wait(NULL))
        logger_error("wait thread #%lu fail", m_status->thread_id());

    // 销毁存储服务器状态检查线程
    if (m_status) {
        delete m_status;
        m_status = NULL;
    }

    // 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/02_tracker/13_main.cpp

```
// 跟踪服务器
// 定义主函数
//
#include "01_globals.h"
#include "11_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("127.0.0.1:21000", "../etc/tracker.cfg");

    return 0;
}
```

~/TNV/src/02_tracker/Makefile

```
PROJ   = ../../bin/tracker
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/tracker.cfg

```
service tracker {
    # 应用ID表
    tnv_apps_id = tnvideo
    # MySQL地址表
    mysql_addrs = 127.0.0.1
    # Redis地址表
    redis_addrs = 127.0.0.1:6379
    # 存储服务器状态检查间隔秒数
    check_active_interval = 120
    # 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/tracker.sql

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

CREATE TABLE `t_groups_info` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `group_name` varchar(32) 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;

INSERT INTO `t_groups_info` (`group_name`) VALUES ('group001');

CREATE TABLE `t_router` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `appid` varchar(32) DEFAULT NULL,
  `userid` varchar(128) DEFAULT NULL,
  `group_name` varchar(32) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
```

```
23 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```