

4 存储服务器

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

```
1 // 存储服务器
2 // 声明全局变量
3 //
4 #pragma once
5
6 #include <vector>
7 #include <string>
8 #include <lib_acl.hpp>
9 //
10 // 配置信息
11 //
12 extern char* cfg_gpname; // 隶属组名
13 extern char* cfg_spaths; // 存储路径表
14 extern char* cfg_taddrs; // 跟踪服务器地址表
15 extern char* cfg_iaddrs; // ID服务器地址表
16 extern char* cfg_maddrs; // MySQL地址表
17 extern char* cfg_raddrs; // Redis地址表
18 extern acl::master_str_tbl cfg_str[]; // 字符串配置表
19
20 extern int cfg_bindport; // 绑定端口号
21 extern int cfg_interval; // 心跳间隔秒数
22 extern int cfg_mtimeout; // MySQL读写超时
23 extern int cfg_maxconns; // Redis连接池最大连接数
24 extern int cfg_ctimeout; // Redis连接超时
25 extern int cfg_rtimeout; // Redis读写超时
26 extern int cfg_ktimeout; // Redis键超时
27 extern acl::master_int_tbl cfg_int[]; // 整型配置表
28
29 extern std::vector<std::string> g_spaths; // 存储路径表
30 extern std::vector<std::string> g_taddrs; // 跟踪服务器地址表
31 extern std::vector<std::string> g_iaddrs; // ID服务器地址表
32 extern std::vector<std::string> g_maddrs; // MySQL地址表
33 extern std::vector<std::string> g_raddrs; // Redis地址表
34 extern acl::redis_client_pool* g_rconns; // Redis连接池
35 extern std::string g_hostname; // 主机名
36 extern char const* g_version; // 版本
37 extern time_t g_stime; // 启动时间
```

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

```
1 // 存储服务器
2 // 定义全局变量
3 //
4 #include "01_globals.h"
5 //
6 // 配置信息
7 //
8 char* cfg_gpname; // 隶属组名
```

```

9 char* cfg_spaths; // 存储路径表
10 char* cfg_taddrs; // 跟踪服务器地址表
11 char* cfg_iaddrs; // ID服务器地址表
12 char* cfg_maddrs; // MySQL地址表
13 char* cfg_raddrs; // Redis地址表
14 acl::master_str_tbl cfg_str[] = { // 字符串配置表
15     {"tnv_group_name", "group001", &cfg_gpname},
16     {"tnv_store_paths", "../data", &cfg_spaths},
17     {"tnv_tracker_addrs", "127.0.0.1:21000", &cfg_taddrs},
18     {"tnv_ids_addrs", "127.0.0.1:22000", &cfg_iaddrs},
19     {"mysql_addrs", "127.0.0.1", &cfg_maddrs},
20     {"redis_addrs", "127.0.0.1:6379", &cfg_raddrs},
21     {0, 0, 0}};
22
23 int cfg_bindport; // 绑定端口号
24 int cfg_interval; // 心跳间隔秒数
25 int cfg_mtimeout; // MySQL读写超时
26 int cfg_maxconns; // Redis连接池最大连接数
27 int cfg_ctimeout; // Redis连接超时
28 int cfg_rtimeout; // Redis读写超时
29 int cfg_ktimeout; // Redis键超时
30 acl::master_int_tbl cfg_int[] = { // 整型配置表
31     {"tnv_storage_port", 23000, &cfg_bindport, 0, 0},
32     {"tnv_heart_beat_interval", 10, &cfg_interval, 0, 0},
33     {"mysql_rw_timeout", 30, &cfg_mtimeout, 0, 0},
34     {"redis_max_conn_num", 600, &cfg_maxconns, 0, 0},
35     {"redis_conn_timeout", 10, &cfg_ctimeout, 0, 0},
36     {"redis_rw_timeout", 10, &cfg_rtimeout, 0, 0},
37     {"redis_key_timeout", 60, &cfg_ktimeout, 0, 0},
38     {0, 0, 0, 0, 0}};
39
40 std::vector<std::string> g_spaths; // 存储路径表
41 std::vector<std::string> g_taddrs; // 跟踪服务器地址表
42 std::vector<std::string> g_iaddrs; // ID服务器地址表
43 std::vector<std::string> g_maddrs; // MySQL地址表
44 std::vector<std::string> g_raddrs; // Redis地址表
45 acl::redis_client_pool* g_rconns; // Redis连接池
46 std::string g_hostname; // 主机名
47 char const* g_version = "1.0"; // 版本
48 time_t g_stime; // 启动时间

```

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

```

1 // 存储服务器
2 // 声明缓存类
3 //
4 #pragma once
5
6 #include <lib_acl.hpp>
7 //
8 // 缓存类
9 //
10 class cache_c {
11 public:
12     // 根据键获取其值

```

```

13     int get(char const* key, acl::string& value) const;
14
15     // 设置指定键的值
16     int set(char const* key, char const* value, int timeout = -1) const;
17
18     // 删除指定键值对
19     int del(char const* key) const;
20 };

```

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

```

1  // 存储服务器
2  // 实现缓存类
3  //
4  #include "01_types.h"
5  #include "01_globals.h"
6  #include "03_cache.h"
7
8  // 根据键获取其值
9  int cache_c::get(char const* key, acl::string& value) const {
10     // 构造键
11     acl::string storage_key;
12     storage_key.format("%s:%s", STORAGE_REDIS_PREFIX, key);
13
14     // 检查Redis连接池
15     if (!g_rconns) {
16         logger_warn("redis connection pool is null, key: %s",
17             storage_key.c_str());
18         return ERROR;
19     }
20
21     // 从连接池中获取一个Redis连接
22     acl::redis_client* rconn = (acl::redis_client*)g_rconns->peek();
23     if (!rconn) {
24         logger_warn("peek redis connection fail, key: %s",
25             storage_key.c_str());
26         return ERROR;
27     }
28
29     // 持有此连接的Redis对象即为Redis客户机
30     acl::redis redis;
31     redis.set_client(rconn);
32
33     // 借助Redis客户机根据键获取其值
34     if (!redis.get(storage_key.c_str(), value)) {
35         logger_warn("get cache fail, key: %s", storage_key.c_str());
36         g_rconns->put(rconn, false);
37         return ERROR;
38     }
39
40     // 检查空值
41     if (value.empty()) {
42         logger_warn("value is empty, key: %s", storage_key.c_str());
43         g_rconns->put(rconn, false);
44         return ERROR;

```

```

45     }
46
47     logger("get cache ok, key: %s, value: %s",
48         storage_key.c_str(), value.c_str());
49     g_rconns->put(rconn, true);
50
51     return OK;
52 }
53
54 // 设置指定键的值
55 int cache_c::set(char const* key, char const* value,
56     int timeout /* = -1 */) const {
57     // 构造键
58     acl::string storage_key;
59     storage_key.format("%s:%s", STORAGE_REDIS_PREFIX, key);
60
61     // 检查Redis连接池
62     if (!g_rconns) {
63         logger_warn("redis connection pool is null, key: %s",
64             storage_key.c_str());
65         return ERROR;
66     }
67
68     // 从连接池中获取一个Redis连接
69     acl::redis_client* rconn = (acl::redis_client*)g_rconns->peek();
70     if (!rconn) {
71         logger_warn("peek Redis connection fail, key: %s",
72             storage_key.c_str());
73         return ERROR;
74     }
75
76     // 持有此连接的Redis对象即为Redis客户机
77     acl::redis redis;
78     redis.set_client(rconn);
79
80     // 借助Redis客户机设置指定键的值
81     if (timeout < 0)
82         timeout = cfg_ktimeout;
83     if (!redis.setex(storage_key.c_str(), value, timeout)) {
84         logger_warn("set cache fail, key: %s, value: %s, timeout: %d",
85             storage_key.c_str(), value, timeout);
86         g_rconns->put(rconn, false);
87         return ERROR;
88     }
89     logger("set cache ok, key: %s, value: %s, timeout: %d",
90         storage_key.c_str(), value, timeout);
91     g_rconns->put(rconn, true);
92
93     return OK;
94 }
95
96 // 删除指定键值对
97 int cache_c::del(char const* key) const {
98     // 构造键
99     acl::string storage_key;
100    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

```

```

22     int get(char const* appid, char const* userid, char const* fileid,
23             std::string& filepath, long long* filesize) const;
24     // 设置文件ID和路径及大小的对应关系
25     int set(char const* appid, char const* userid, char const* fileid,
26             char const* filepath, long long filesize) const;
27     // 删除文件ID
28     int del(char const* appid, char const*userid,
29             char const* fileid) const;
30
31 private:
32     // 根据用户ID获取其对应的表名
33     std::string table_of_user(char const* userid) const;
34
35     // 计算哈希值
36     unsigned int hash(char const* buf, size_t len) const;
37
38     MYSQL* m_mysql; // MySQL对象
39 };

```

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

```

1 // 存储服务器
2 // 实现数据库访问类
3 //
4 #include "01_types.h"
5 #include "01_globals.h"
6 #include "03_cache.h"
7 #include "05_db.h"
8
9 // 构造函数
10 db_c::db_c(void): m_mysql(mysql_init(NULL)) { // 创建MySQL对象
11     if (!m_mysql)
12         logger_error("create dao fail: %s", mysql_error(m_mysql));
13 }
14
15 // 析构函数
16 db_c::~db_c(void) {
17     // 销毁MySQL对象
18     if (m_mysql) {
19         mysql_close(m_mysql);
20         m_mysql = NULL;
21     }
22 }
23
24 // 连接数据库
25 int db_c::connect(void) {
26     MYSQL* mysql = m_mysql;
27
28     // 遍历MySQL地址表，尝试连接数据库
29     for (std::vector<std::string>::const_iterator maddr =
30           g_maddrs.begin(); maddr != g_maddrs.end(); ++maddr)
31         if ((m_mysql = mysql_real_connect(mysql, maddr->c_str(),
32                                           "root", "123456", "tnv_storagedb", 0, NULL, 0)))
33             return OK;
34

```

```

35     logger_error("connect database fail: %s",
36         mysql_error(m_mysql = mysql));
37     return ERROR;
38 }
39
40 // 根据文件ID获取其对应的路径及大小
41 int db_c::get(char const* appid, char const* userid, char const* fileid,
42     std::string& filepath, long long* filesize) const {
43     // 先尝试从缓存中获取与文件ID对应的路径及大小
44     cache_c cache;
45     acl::string key;
46     key.format("uid:fid:%s:%s", userid, fileid);
47     acl::string value;
48     if (cache.get(key, value) == OK) {
49         std::vector<acl::string> size_path = value.split2(";");
50         if (size_path.size() == 2) {
51             filepath = size_path[1].c_str();
52             *filesize = atoll(size_path[0].c_str());
53             if (!filepath.empty() && *filesize > 0) {
54                 logger("from cache, appid: %s, userid: %s, "
55                     "fileid: %s, filepath: %s, filesize: %lld",
56                     appid, userid, fileid, filepath.c_str(), *filesize);
57                 return OK;
58             }
59         }
60     }
61
62     // 缓存中没有再查询数据库
63     std::string tablename = table_of_user(userid);
64     if (tablename.empty()) {
65         logger_error("tablename is empty, appid: %s, "
66             "userid: %s, fileid: %s", appid, userid, fileid);
67         return ERROR;
68     }
69     acl::string sql;
70     sql.format("SELECT file_path, file_size FROM %s WHERE id='%s';",
71         tablename.c_str(), fileid);
72     if (mysql_query(m_mysql, sql.c_str())) {
73         logger_error("query database fail: %s, sql: %s",
74             mysql_error(m_mysql), sql.c_str());
75         return ERROR;
76     }
77
78     // 获取查询结果
79     MYSQL_RES* res = mysql_store_result(m_mysql);
80     if (!res) {
81         logger_error("result is null: %s, sql: %s",
82             mysql_error(m_mysql), sql.c_str());
83         return ERROR;
84     }
85
86     // 获取结果记录
87     MYSQL_ROW row = mysql_fetch_row(res);
88     if (!row) {
89         logger_error("result is empty: %s, sql: %s",
90             mysql_error(m_mysql), sql.c_str());

```

```

91         return ERROR;
92     }
93     filepath = row[0];
94     *filesize = atoll(row[1]);
95     logger("from database, appid: %s, userid: %s, "
96           "fileid: %s, filepath: %s, filesize: %lld",
97           appid, userid, fileid, filepath.c_str(), *filesize);
98
99     // 将文件ID和路径及大小的对应关系保存在缓存中
100    value.format("%lld;%s", *filesize, filepath.c_str());
101    cache.set(key, value.c_str());
102
103    return OK;
104 }
105
106 // 设置文件ID和路径及大小的对应关系
107 int db_c::set(char const* appid, char const* userid, char const* fileid,
108               char const* filepath, long long filesize) const {
109     // 根据用户ID获取其对应的表名
110     std::string tablename = table_of_user(userid);
111     if (tablename.empty()) {
112         logger_error("tablename is empty, appid: %s, "
113                     "userid: %s, fileid: %s", appid, userid, fileid);
114         return ERROR;
115     }
116
117     // 插入一条记录
118     acl::string sql;
119     sql.format("INSERT INTO %s SET id='%s', appid='%s', "
120              "userid='%s', status=0, file_path='%s', file_size=%lld;",
121              tablename.c_str(), fileid, appid, userid, filepath, filesize);
122     if (mysql_query(m_mysql, sql.c_str())) {
123         logger_error("insert database fail: %s, sql: %s",
124                     mysql_error(m_mysql), sql.c_str());
125         return ERROR;
126     }
127
128     // 检查插入结果
129     MYSQL_RES* res = mysql_store_result(m_mysql);
130     if (!res && mysql_field_count(m_mysql)) {
131         logger_error("insert database fail: %s, sql: %s",
132                     mysql_error(m_mysql), sql.c_str());
133         return ERROR;
134     }
135
136     return OK;
137 }
138
139 // 删除文件ID
140 int db_c::del(char const* appid, char const*userid,
141               char const* fileid) const {
142     // 先从缓存中删除文件ID
143     cache_c cache;
144     acl::string key;
145     key.format("uid:fid:%s:%s", userid, fileid);
146     if (cache.del(key) != OK)

```



```

147         logger_warn("delete cache fail: appid: %s, "
148                     "userid: %s, fileid: %s", appid, userid, fileid);
149
150         // 再从数据库中删除文件ID
151         std::string tablename = table_of_user(userid);
152         if (tablename.empty()) {
153             logger_error("tablename is empty, appid: %s, "
154                         "userid: %s, fileid: %s", appid, userid, fileid);
155             return ERROR;
156         }
157         acl::string sql;
158         sql.format("DELETE FROM %s WHERE id='%s'",
159                  tablename.c_str(), fileid);
160         if (mysql_query(m_mysql, sql.c_str())) {
161             logger_error("delete database fail: %s, sql: %s",
162                         mysql_error(m_mysql), sql.c_str());
163             return ERROR;
164         }
165
166         // 检查删除结果
167         MYSQL_RES* res = mysql_store_result(m_mysql);
168         if (!res && mysql_field_count(m_mysql)) {
169             logger_error("delete database fail: %s, sql: %s",
170                         mysql_error(m_mysql), sql.c_str());
171             return ERROR;
172         }
173
174         return OK;
175     }
176
177     // 根据用户ID获取其对应的表名
178     std::string db_c::table_of_user(char const* userid) const {
179         char tablename[10];
180
181         sprintf(tablename, "t_file_%02d",
182                (hash(userid, strlen(userid)) & 0x7FFFFFFF) % 3 + 1);
183
184         return tablename;
185     }
186
187     // 计算哈希值
188     unsigned int db_c::hash(char const* buf, size_t len) const {
189         unsigned int h = 0;
190
191         for (size_t i = 0; i < len; ++i)
192             h ^= i&1 ? ~(h<<11^buf[i]^h>>5) : h<<7^buf[i]^h>>3;
193
194         return h;
195     }

```

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

```

1 // 存储服务器
2 // 声明文件操作类
3 //

```

```

4  #pragma once
5
6  #include <sys/types.h>
7  //
8  // 文件操作类
9  //
10 class file_c {
11 public:
12     // 构造函数
13     file_c(void);
14     // 析构函数
15     ~file_c(void);
16
17     // 打开文件
18     int open(char const* path, char flag);
19     // 关闭文件
20     int close(void);
21
22     // 读取文件
23     int read(void* buf, size_t count) const;
24     // 写入文件
25     int write(void const* buf, size_t count) const;
26
27     // 设置偏移
28     int seek(off_t offset) const;
29     // 删除文件
30     static int del(char const* path);
31
32     // 打开标志
33     static char const O_READ   = 'r';
34     static char const O_WRITE  = 'w';
35
36 private:
37     int m_fd; // 文件描述符
38 };

```

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

```

1  // 存储服务器
2  // 实现文件操作类
3  //
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <lib_acl.hpp>
7  #include "01_types.h"
8  #include "07_file.h"
9
10 // 构造函数
11 file_c::file_c(void): m_fd(-1) {
12 }
13
14 // 析构函数
15 file_c::~~file_c(void) {
16     close();
17 }

```

```
18
19 // 打开文件
20 int file_c::open(char const* path, char flag) {
21     // 检查路径
22     if (!path || !*path) {
23         logger_error("path is null");
24         return ERROR;
25     }
26
27     // 打开文件
28     if (flag == O_READ)
29         m_fd = ::open(path, O_RDONLY);
30     else if (flag == O_WRITE)
31         m_fd = ::open(path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
32     else {
33         logger_error("unknown open flag: %c", flag);
34         return ERROR;
35     }
36
37     // 打开失败
38     if (m_fd == -1) {
39         logger_error("call open fail: %s, path: %s, flag: %c",
40             strerror(errno), path, flag);
41         return ERROR;
42     }
43
44     return OK;
45 }
46
47 // 关闭文件
48 int file_c::close(void) {
49     if (m_fd != -1) {
50         // 关闭文件
51         if (::close(m_fd) == -1) {
52             logger_error("call close fail: %s", strerror(errno));
53             return ERROR;
54         }
55
56         m_fd = -1;
57     }
58
59     return OK;
60 }
61
62 // 读取文件
63 int file_c::read(void* buf, size_t count) const {
64     // 检查文件描述符
65     if (m_fd == -1) {
66         logger_error("invalid file handle");
67         return ERROR;
68     }
69
70     // 检查文件缓冲区
71     if (!buf) {
72         logger_error("invalid file buffer");
73         return ERROR;
```

```
74     }
75
76     // 读取给定字节数
77     ssize_t bytes = ::read(m_fd, buf, count);
78     if (bytes == -1) {
79         logger_error("call read fail: %s", strerror(errno));
80         return ERROR;
81     }
82     if ((size_t)bytes != count) {
83         logger_error("unable to read expected bytes: %ld != %lu",
84             bytes, count);
85         return ERROR;
86     }
87
88     return OK;
89 }
90
91 // 写入文件
92 int file_c::write(void const* buf, size_t count) const {
93     // 检查文件描述符
94     if (m_fd == -1) {
95         logger_error("invalid file handle");
96         return ERROR;
97     }
98
99     // 检查文件缓冲区
100    if (!buf) {
101        logger_error("invalid file buffer");
102        return ERROR;
103    }
104
105    // 写入给定字节数
106    if (::write(m_fd, buf, count) == -1) {
107        logger_error("call write fail: %s", strerror(errno));
108        return ERROR;
109    }
110
111    return OK;
112 }
113
114 // 设置偏移
115 int file_c::seek(off_t offset) const {
116     // 检查文件描述符
117     if (m_fd == -1) {
118         logger_error("invalid file handle");
119         return ERROR;
120     }
121
122     // 检查文件偏移量
123     if (offset < 0) {
124         logger_error("invalid file offset");
125         return ERROR;
126     }
127
128     // 设置文件偏移量
129     if (lseek(m_fd, offset, SEEK_SET) == -1) {
```

```

130         logger_error("call lseek fail: %s, offset: %ld",
131                       strerror(errno), offset);
132         return ERROR;
133     }
134
135     return OK;
136 }
137
138 // 删除文件
139 int file_c::del(char const* path) {
140     // 检查路径
141     if (!path || !*path) {
142         logger_error("path is null");
143         return ERROR;
144     }
145
146     // 删除文件
147     if (unlink(path) == -1) {
148         logger_error("call unlink fail: %s, path: %s",
149                       strerror(errno), path);
150         return ERROR;
151     }
152
153     return OK;
154 }

```

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

```

1 // 存储服务器
2 // 声明ID客户机类
3 //
4 #pragma once
5 //
6 // ID客户机类
7 //
8 class id_c {
9 public:
10     // 从ID服务器获取与ID键相对应的值
11     long get(char const* key) const;
12
13 private:
14     // 向ID服务器发送请求，接收并解析响应，从中获得ID值
15     long client(char const* requ, long long requlen) const;
16 };

```

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

```

1 // 存储服务器
2 // 实现ID客户机类
3 //
4 #include <lib_acl.hpp>
5 #include "02_proto.h"
6 #include "03_util.h"
7 #include "01_globals.h"

```

```

8  #include "09_id.h"
9
10 // 从ID服务器获取与ID键相对应的值
11 long id_c::get(char const* key) const {
12     // 检查ID键
13     if (!key) {
14         logger_error("key is null");
15         return -1;
16     }
17     size_t keylen = strlen(key);
18     if (!keylen) {
19         logger_error("key is null");
20         return -1;
21     }
22     if (keylen > ID_KEY_MAX) {
23         logger_error("key too big: %lu > %d", keylen, ID_KEY_MAX);
24         return -1;
25     }
26
27     // |包体长度|命令|状态| ID键 |
28     // | 8 | 1 | 1 |包体长度|
29     // 构造请求
30     long long bodylen = keylen + 1;
31     long long requlen = HEADLEN + bodylen;
32     char requ[requlen] = {};
33     htonl(bodylen, requ);
34     requ[BODYLEN_SIZE] = CMD_ID_GET;
35     requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
36     strcpy(requ + HEADLEN, key);
37
38     // 向ID服务器发送请求,接收并解析响应,从中获得ID值
39     return client(requ, requlen);
40 }
41
42 // 向ID服务器发送请求,接收并解析响应,从中获得ID值
43 long id_c::client(char const* requ, long long requlen) const {
44     acl::socket_stream conn;
45
46     // 从ID服务器地址表中随机抽取一台ID服务器尝试连接
47     srand(time(NULL));
48     int nids = g_iaddrs.size();
49     int nrand = rand() % nids;
50     for (int i = 0; i < nids; ++i)
51         if (conn.open(g_iaddrs[nrand].c_str(), 0, 0)) {
52             logger("connect id success, addr: %s",
53                 g_iaddrs[nrand].c_str());
54             break;
55         }
56     else {
57         logger("connect id fail, addr: %s",
58             g_iaddrs[nrand].c_str());
59         nrand = (nrand + 1) % nids;
60     }
61     if (!conn.alive()) {
62         logger_error("connect id fail, addr: %s", cfg_iaddrs);
63         return -1;

```

```

64     }
65
66     // 向ID服务器发送请求
67     if (conn.write(requ, requlen) < 0) {
68         logger_error("write fail: %s, requlen: %lld, to: %s",
69             acl::last_error(), requlen, conn.get_peer());
70         conn.close();
71         return -1;
72     }
73
74     // 从ID服务器接收响应
75     long long resplen = HEADLEN + BODYLEN_SIZE;
76     char resp[resplen] = {};
77     if (conn.read(resp, resplen) < 0) {
78         logger_error("read fail: %s, resplen: %lld, from: %s",
79             acl::last_error(), resplen, conn.get_peer());
80         conn.close();
81         return -1;
82     }
83
84     // |包体长度|命令|状态|ID值|
85     // | 8 | 1 | 1 | 8 |
86     // 从ID服务器的响应中解析出ID值
87     long value = ntohl(resp + HEADLEN);
88
89     conn.close();
90
91     return value;
92 }

```

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

```

1  // 存储服务器
2  // 声明业务服务类
3  //
4  #pragma once
5
6  #include <lib_acl.hpp>
7  //
8  // 业务服务类
9  //
10 class service_c {
11 public:
12     // 业务处理
13     bool business(acl::socket_stream* conn, char const* head) const;
14
15 private:
16     // 处理来自客户机的上传文件请求
17     bool upload(acl::socket_stream* conn, long long bodylen) const;
18     // 处理来自客户机的询问文件大小请求
19     bool filesize(acl::socket_stream* conn, long long bodylen) const;
20     // 处理来自客户机的下载文件请求
21     bool download(acl::socket_stream* conn, long long bodylen) const;
22     // 处理来自客户机的删除文件请求
23     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(ac1::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(ac1::socket_stream* conn, char const* filepath,
38         long long offset, long long size) const;
39
40 // 应答成功
41 bool ok(ac1::socket_stream* conn) const;
42 // 应答错误
43 bool error(ac1::socket_stream* conn, short errnumb,
44            char const* format, ...) const;
45 };

```

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

```

1 // 存储服务
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(ac1::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 = nto11(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_error(), expected, conn->get_peer());
83         return false;
84     }
85
86     // 解析包体

```

```

87     char appid[APPID_SIZE];
88     strcpy(appid, body);
89     char userid[USERID_SIZE];
90     strcpy(userid, body + APPID_SIZE);
91     char fileid[FILEID_SIZE];
92     strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
93     long long filesize = ntoll(
94         body + APPID_SIZE + USERID_SIZE + FILEID_SIZE);
95
96     // 检查文件大小
97     if (filesize != bodylen - expected) {
98         logger_error("invalid file size: %lld != %lld",
99             filesize, bodylen - expected);
100        error(conn, -1, "invalid file size: %lld != %lld",
101            filesize, bodylen - expected);
102        return false;
103    }
104
105    // 生成文件路径
106    char filepath[PATH_MAX+1];
107    if (genpath(filepath) != OK) {
108        error(conn, -1, "get filepath fail");
109        return false;
110    }
111
112    logger("upload file, appid: %s, userid: %s, "
113        "fileid: %s, filesize: %lld, filepath: %s",
114        appid, userid, fileid, filesize, filepath);
115
116    // 接收并保存文件
117    int result = save(conn, appid, userid, fileid, filesize, filepath);
118    if (result == SOCKET_ERROR)
119        return false;
120    else if (result == ERROR) {
121        error(conn, -1, "receive and save file fail, fileid: %s",
122            fileid);
123        return false;
124    }
125
126    return ok(conn);
127 }
128
129 // 处理来自客户机的询问文件大小请求
130 bool service_c::filesize(ac1::socket_stream* conn,
131     long long bodylen) const {
132     // |包体长度|命令|状态|应用ID|用户ID|文件ID|
133     // | 8 | 1 | 1 | 16 | 256 | 128 |
134     // 检查包体长度
135     long long expected = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
136     if (bodylen != expected) {
137         error(conn, -1, "invalid body length: %lld != %lld",
138             bodylen, expected);
139         return false;
140     }
141
142     // 接收包体

```

```

143     char body[bodylen];
144     if (conn->read(body, bodylen) < 0) {
145         logger_error("read fail: %s, bodylen: %lld, from: %s",
146             acl::last_serror(), bodylen, conn->get_peer());
147         return false;
148     }
149
150     // 解析包体
151     char appid[APPID_SIZE];
152     strcpy(appid, body);
153     char userid[USERID_SIZE];
154     strcpy(userid, body + APPID_SIZE);
155     char fileid[FILEID_SIZE];
156     strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
157
158     db_c db; // 数据库访问对象
159
160     // 连接数据库
161     if (db.connect() != OK)
162         return false;
163
164     std::string filepath; // 文件路径
165     long long    filesize; // 文件大小
166
167     // 根据文件ID获取其对应的路径及大小
168     if (db.get(appid, userid, fileid, filepath, &filesize) != OK) {
169         error(conn, -1, "read database fail, fileid: %s", fileid);
170         return false;
171     }
172
173     logger("filesize, appid: %s, userid: %s, "
174         "fileid: %s, filepath: %s, filesize: %lld",
175         appid, userid, fileid, filepath.c_str(), filesize);
176
177     // |包体长度|命令|状态|文件大小|
178     // | 8 | 1 | 1 | 8 |
179     // 构造响应
180     bodylen = BODYLEN_SIZE;
181     long long resplen = HEADLEN + bodylen;
182     char resp[resplen] = {};
183     htonl(bodylen, resp);
184     resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
185     resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;
186     htonl(filesize, resp + HEADLEN);
187
188     // 发送响应
189     if (conn->write(resp, resplen) < 0) {
190         logger_error("write fail: %s, resplen: %lld, to: %s",
191             acl::last_serror(), resplen, conn->get_peer());
192         return false;
193     }
194
195     return true;
196 }
197
198 // 处理来自客户机的下载文件请求

```

```

199 bool service_c::download(ac1::socket_stream* conn,
200     long long bodylen) const {
201     // |包体长度|命令|状态|应用ID|用户ID|文件ID|偏移|大小|
202     // | 8 | 1 | 1 | 16 | 256 | 128 | 8 | 8 |
203     // 检查包体长度
204     long long expected = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
205         BODYLEN_SIZE + BODYLEN_SIZE;
206     if (bodylen != expected) {
207         error(conn, -1, "invalid body length: %lld != %lld",
208             bodylen, expected);
209         return false;
210     }
211
212     // 接收包体
213     char body[bodylen];
214     if (conn->read(body, bodylen) < 0) {
215         logger_error("read fail: %s, bodylen: %lld, from: %s",
216             ac1::last_serror(), bodylen, conn->get_peer());
217         return false;
218     }
219
220     // 解析包体
221     char appid[APPID_SIZE];
222     strcpy(appid, body);
223     char userid[USERID_SIZE];
224     strcpy(userid, body + APPID_SIZE);
225     char fileid[FILEID_SIZE];
226     strcpy(fileid, body + APPID_SIZE + USERID_SIZE);
227     long long offset = nto11(
228         body + APPID_SIZE + USERID_SIZE + FILEID_SIZE);
229     long long size = nto11(
230         body + APPID_SIZE + USERID_SIZE + FILEID_SIZE + BODYLEN_SIZE);
231
232     db_c db; // 数据库访问对象
233
234     // 连接数据库
235     if (db.connect() != OK)
236         return false;
237
238     std::string filepath; // 文件路径
239     long long filesize; // 文件大小
240
241     // 根据文件ID获取其对应的路径及大小
242     if (db.get(appid, userid, fileid, filepath, &filesize) != OK) {
243         error(conn, -1, "read database fail, fileid: %s", fileid);
244         return false;
245     }
246
247     // 检查位置
248     if (offset < 0 || filesize < offset) {
249         logger_error("invalid offset, %lld is not between 0 and %lld",
250             offset, filesize);
251         error(conn, -1, "invalid offset, %lld is not between 0 and %lld",
252             offset, filesize);
253         return false;
254     }

```

```

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(ac1::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             ac1::last_error(), 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

```

```

367 // 将ID转换为512进制
368 long service_c::id512(long id) const {
369     long result = 0;
370
371     for (int i = 1; id; i *= 1000) {
372         result += (id % 512) * i;
373         id /= 512;
374     }
375
376     return result;
377 }
378
379 // 用文件ID生成文件路径
380 int service_c::id2path(char const* spath, long fileid,
381     char* filepath) const {
382     // 检查存储路径
383     if (!spath || !*spath) {
384         logger_error("storage path is null");
385         return ERROR;
386     }
387
388     // 生成文件路径中的各个分量
389     unsigned short subdir1 = (fileid / 1000000000) % 1000; // 一级子目录
390     unsigned short subdir2 = (fileid / 1000000) % 1000;    // 二级子目录
391     unsigned short subdir3 = (fileid / 1000) % 1000;        // 三级子目录
392     time_t curtime = time(NULL);                            // 当前时间戳
393     unsigned short postfix = (fileid / 1) % 1000;           // 文件名后缀
394
395     // 格式化完整的文件路径
396     if (spath[strlen(spath)-1] == '/')
397         snprintf(filepath, PATH_MAX + 1, "%s%03X/%03X/%03X/%1X_%03X",
398             spath, subdir1, subdir2, subdir3, curtime, postfix);
399     else
400         snprintf(filepath, PATH_MAX + 1, "%s/%03X/%03X/%03X/%1X_%03X",
401             spath, subdir1, subdir2, subdir3, curtime, postfix);
402
403     return OK;
404 }
405
406 // 接收并保存文件
407 int service_c::save(ac1::socket_stream* conn, char const* appid,
408     char const* userid, char const* fileid, long long filesize,
409     char const* filepath) const {
410     file_c file; // 文件操作对象
411
412     // 打开文件
413     if (file.open(filepath, file_c::O_WRITE) != OK)
414         return ERROR;
415
416     // 依次将接收到的数据块写入文件
417     long long remain = filesize; // 未接收字节数
418     char rcvwr[STORAGE_RCVWR_SIZE]; // 接收写入缓冲区
419     while (remain) { // 还有未接收数据
420         // 接收数据
421         long long bytes = std::min(remain, (long long)sizeof(rcvwr));
422         long long count = conn->read(rcvwr, bytes);

```

```

423         if (count < 0) {
424             logger_error("read fail: %s, bytes: %lld, from: %s",
425                 acl::last_serror(), bytes, conn->get_peer());
426             file.close();
427             return SOCKET_ERROR;
428         }
429         // 写入文件
430         if (file.write(rcvwr, count) != OK) {
431             file.close();
432             return ERROR;
433         }
434         // 未收递减
435         remain -= count;
436     }
437
438     // 关闭文件
439     file.close();
440
441     db_c db; // 数据库访问对象
442
443     // 连接数据库
444     if (db.connect() != OK)
445         return ERROR;
446
447     // 设置文件ID和路径及大小的对应关系
448     if (db.set(appid, userid, fileid, filepath, filesize) != OK) {
449         error(conn, -1, "insert database fail, fileid: %s", fileid);
450         file.del(filepath);
451         return ERROR;
452     }
453
454     return OK;
455 }
456
457 // 读取并发送文件
458 int service_c::send(acl::socket_stream* conn, char const* filepath,
459     long long offset, long long size) const {
460     file_c file; // 文件操作对象
461
462     // 打开文件
463     if (file.open(filepath, file_c::O_READ) != OK)
464         return ERROR;
465
466     // 设置偏移
467     if (offset && file.seek(offset) != OK) {
468         file.close();
469         return ERROR;
470     }
471
472     // |包体长度|命令|状态|文件内容|
473     // | 8 | 1 | 1 |内容大小|
474     // 构造响应头
475     long long bodylen = size;
476     long long headlen = HEADLEN;
477     char head[headlen] = {};
478     hton(bodylen, head);

```



```

479     head[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
480     head[BODYLEN_SIZE+COMMAND_SIZE] = 0;
481
482     // 发送响应头
483     if (conn->write(head, headlen) < 0) {
484         logger_error("write fail: %s, headlen: %lld, to: %s",
485             acl::last_serror(), headlen, conn->get_peer());
486         file.close();
487         return SOCKET_ERROR;
488     }
489
490     // 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
491     long long remain = size; // 未读取字节数
492     char rdsnd[STORAGE_RDSND_SIZE]; // 读取发送缓冲区
493     while (remain) { // 还有未读取数据
494         // 读取文件
495         long long count = std::min(remain, (long long)sizeof(rdsnd));
496         if (file.read(rdsnd, count) != OK) {
497             file.close();
498             return ERROR;
499         }
500         // 发送数据
501         if (conn->write(rdsnd, count) < 0) {
502             logger_error("write fail: %s, count: %lld, to: %s",
503                 acl::last_serror(), count, conn->get_peer());
504             file.close();
505             return SOCKET_ERROR;
506         }
507         // 未读递减
508         remain -= count;
509     }
510
511     // 关闭文件
512     file.close();
513
514     return OK;
515 }
516
517 ///////////////////////////////////////////////////
518
519 // 应答成功
520 bool service_c::ok(acl::socket_stream* conn) const {
521     // |包体长度|命令|状态|
522     // | 8 | 1 | 1 |
523     // 构造响应
524     long long bodylen = 0;
525     long long resplen = HEADLEN + bodylen;
526     char resp[resplen] = {};
527     hton(bodylen, resp);
528     resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
529     resp[BODYLEN_SIZE+COMMAND_SIZE] = 0;
530
531     // 发送响应
532     if (conn->write(resp, resplen) < 0) {
533         logger_error("write fail: %s, resplen: %lld, to: %s",
534             acl::last_serror(), resplen, conn->get_peer());

```

```

535         return false;
536     }
537
538     return true;
539 }
540
541 // 应答错误
542 bool service_c::error(acl::socket_stream* conn, short errnumb,
543     char const* format, ...) const {
544     // 错误描述
545     char errdesc[ERROR_DESC_SIZE];
546     va_list ap;
547     va_start(ap, format);
548     vsnprintf(errdesc, ERROR_DESC_SIZE, format, ap);
549     va_end(ap);
550     logger_error("%s", errdesc);
551     acl::string desc;
552     desc.format("[%s] %s", g_hostname.c_str(), errdesc);
553     memset(errdesc, 0, sizeof(errdesc));
554     strncpy(errdesc, desc.c_str(), ERROR_DESC_SIZE - 1);
555     size_t descrlen = strlen(errdesc);
556     descrlen += descrlen != 0;
557
558     // |包体长度|命令|状态|错误号|错误描述|
559     // |      8      | 1 | 1 | 2 | <=1024 |
560     // 构造响应
561     long long bodylen = ERROR_NUMB_SIZE + descrlen;
562     long long resplen = HEADLEN + bodylen;
563     char resp[resplen] = {};
564     htonl(bodylen, resp);
565     resp[BODYLEN_SIZE] = CMD_STORAGE_REPLY;
566     resp[BODYLEN_SIZE+COMMAND_SIZE] = STATUS_ERROR;
567     ston(errnumb, resp + HEADLEN);
568     if (descrlen)
569         strcpy(resp + HEADLEN + ERROR_NUMB_SIZE, errdesc);
570
571     // 发送响应
572     if (conn->write(resp, resplen) < 0) {
573         logger_error("write fail: %s, resplen: %lld, to: %s",
574             acl::last_serror(), resplen, conn->get_peer());
575         return false;
576     }
577
578     return true;
579 }

```

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

```

1 // 存储服务器
2 // 声明跟踪客户机线程类
3 //
4 #pragma once
5
6 #include <lib_acl.hpp>
7 //

```

```

8 // 跟踪客户机线程类
9 //
10 class tracker_c: public acl::thread {
11 public:
12     // 构造函数
13     tracker_c(char const* taddr);
14
15     // 终止线程
16     void stop(void);
17
18 protected:
19     // 线程过程
20     void* run(void);
21
22 private:
23     // 向跟踪服务器发送加入包
24     int join(acl::socket_stream* conn) const;
25     // 向跟踪服务器发送心跳包
26     int beat(acl::socket_stream* conn) const;
27
28     bool m_stop; // 是否终止
29     acl::string m_taddr; // 跟踪服务器地址
30 };

```

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

```

1 // 存储服务器
2 // 实现跟踪客户机线程类
3 //
4 #include <unistd.h>
5 #include "02_proto.h"
6 #include "03_util.h"
7 #include "01_globals.h"
8 #include "13_tracker.h"
9
10 // 构造函数
11 tracker_c::tracker_c(char const* taddr): m_stop(false), m_taddr(taddr) {
12 }
13
14 // 终止线程
15 void tracker_c::stop(void) {
16     m_stop = true;
17 }
18
19 // 线程过程
20 void* tracker_c::run(void) {
21     acl::socket_stream conn;
22
23     while (!m_stop) {
24         // 连接跟踪服务器
25         if (!conn.open(m_taddr, 10, 30)) {
26             logger_error("connect tracker fail, taddr: %s",
27                 m_taddr.c_str());
28             sleep(2);
29             continue; // 失败重连

```

```

30     }
31
32     // 向跟踪服务器发送加入包
33     if (join(&conn) != OK) {
34         conn.close();
35         sleep(2);
36         continue; // 失败重连
37     }
38
39     time_t last = time(NULL); // 上次心跳
40     while (!m_stop) {
41         time_t now = time(NULL); // 现在
42         // 现在离上次心跳已足够久，再跳一次
43         if (now - last >= cfg_interval) {
44             // 向跟踪服务器发送心跳包
45             if (beat(&conn) != OK) {
46                 sleep(2);
47                 break; // 失败重连
48             }
49             last = now;
50         }
51         sleep(1);
52     }
53
54     conn.close();
55 }
56
57 return NULL;
58 }
59
60 // 向跟踪服务器发送加入包
61 int tracker_c::join(ac1::socket_stream* conn) const {
62     // |包体长度|命令|状态|storage_join_body_t|
63     // | 8 | 1 | 1 | 包体长度 |
64     // 构造请求
65     long long bodylen = sizeof(storage_join_body_t);
66     long long requlen = HEADLEN + bodylen;
67     char requ[requlen] = {};
68     l1ton(bodylen, requ);
69     requ[BODYLEN_SIZE] = CMD_TRACKER_JOIN;
70     requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
71     storage_join_body_t* sjb = (storage_join_body_t*)(requ + HEADLEN);
72     strcpy(sjb->sjb_version, g_version); // 版本
73     strcpy(sjb->sjb_groupname, cfg_gpname); // 组名
74     strcpy(sjb->sjb_hostname, g_hostname.c_str()); // 主机名
75     ston(cfg_bindport, sjb->sjb_port); // 端口号
76     lton(g_stime, sjb->sjb_stime); // 启动时间
77     lton(time(NULL), sjb->sjb_jtime); // 加入时间
78
79     // 发送请求
80     if (conn->write(requ, requlen) < 0) {
81         logger_error("write fail: %s, requlen: %lld, to: %s",
82             ac1::last_serror(), requlen, conn->get_peer());
83         return SOCKET_ERROR;
84     }
85 }

```

```

86 // 接收包头
87 char head[HEADLEN];
88 if (conn->read(head, HEADLEN) < 0) {
89     logger_error("read fail: %s, from: %s",
90         acl::last_serror(), conn->get_peer());
91     return SOCKET_ERROR;
92 }
93
94 // |包体长度|命令|状态|
95 // | 8 | 1 | 1 |
96 // 解析包头
97 if ((bodylen = ntohl(head)) < 0) { // 包体长度
98     logger_error("invalid body length: %lld < 0", bodylen);
99     return ERROR;
100 }
101 int command = head[BODYLEN_SIZE]; // 命令
102 int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
103 logger("bodylen: %lld, command: %d, status: %d",
104     bodylen, command, status);
105
106 // 检查命令
107 if (command != CMD_TRACKER_REPLY) {
108     logger_error("unknown command: %d", command);
109     return ERROR;
110 }
111
112 // 应答成功
113 if (!status) return OK;
114
115 // |包体长度|命令|状态|错误号|错误描述|
116 // | 8 | 1 | 1 | 2 | <=1024 |
117 // 检查包体长度
118 long long expected = ERROR_NUMB_SIZE + ERROR_DESC_SIZE;
119 if (bodylen > expected) {
120     logger_error("invalid body length: %lld > %lld", bodylen,
expected);
121     return ERROR;
122 }
123
124 // 接收包体
125 char body[bodylen];
126 if (conn->read(body, bodylen) < 0) {
127     logger_error("read fail: %s, bodylen: %lld, from: %s",
128         acl::last_serror(), bodylen, conn->get_peer());
129     return SOCKET_ERROR;
130 }
131
132 // 解析包体
133 short errnumb = ntos(body);
134 char const* errdesc = "";
135 if (bodylen > ERROR_NUMB_SIZE)
136     errdesc = body + ERROR_NUMB_SIZE;
137
138 logger_error("join fail, errnumb: %d, errdesc: %s",
139     errnumb, errdesc);
140 return ERROR;

```

```

141 }
142
143 // 向跟踪服务器发送心跳包
144 int tracker_c::beat(acl::socket_stream* conn) const {
145     // |包体长度|命令|状态|storage_beat_body_t|
146     // | 8 | 1 | 1 | 包体长度 |
147     // 构造请求
148     long long bodylen = sizeof(storage_beat_body_t);
149     long long requlen = HEADLEN + bodylen;
150     char requ[requlen] = {};
151     htonl(bodylen, requ);
152     requ[BODYLEN_SIZE] = CMD_TRACKER_BEAT;
153     requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;
154     storage_beat_body_t* sbb = (storage_beat_body_t*)(requ + HEADLEN);
155     strcpy(sbb->sbb_groupname, cfg_gpname); // 组名
156     strcpy(sbb->sbb_hostname, g_hostname.c_str()); // 主机名
157
158     // 发送请求
159     if (conn->write(requ, requlen) < 0) {
160         logger_error("write fail: %s, requlen: %lld, to: %s",
161             acl::last_error(), requlen, conn->get_peer());
162         return SOCKET_ERROR;
163     }
164
165     // 接收包头
166     char head[HEADLEN];
167     if (conn->read(head, HEADLEN) < 0) {
168         logger_error("read fail: %s, from: %s",
169             acl::last_error(), conn->get_peer());
170         return SOCKET_ERROR;
171     }
172
173     // |包体长度|命令|状态|
174     // | 8 | 1 | 1 |
175     // 解析包头
176     if ((bodylen = ntohl(head)) < 0) { // 包体长度
177         logger_error("invalid body length: %lld < 0", bodylen);
178         return ERROR;
179     }
180     int command = head[BODYLEN_SIZE]; // 命令
181     int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
182     logger("bodylen: %lld, command: %d, status: %d",
183         bodylen, command, status);
184
185     // 检查命令
186     if (command != CMD_TRACKER_REPLY) {
187         logger_error("unknown command: %d", command);
188         return ERROR;
189     }
190
191     // 应答成功
192     if (!status) return OK;
193
194     // |包体长度|命令|状态|错误号|错误描述|
195     // | 8 | 1 | 1 | 2 | <=1024 |
196     // 检查包体长度

```

```

197     long long expected = ERROR_NUMB_SIZE + ERROR_DESC_SIZE;
198     if (bodylen > expected) {
199         logger_error("invalid body length: %lld > %lld",
200             bodylen, expected);
201         return ERROR;
202     }
203
204     // 接收包体
205     char body[bodylen];
206     if (conn->read(body, bodylen) < 0) {
207         logger_error("read fail: %s, bodylen: %lld, from: %s",
208             acl::last_serror(), bodylen, conn->get_peer());
209         return SOCKET_ERROR;
210     }
211
212     // 解析包体
213     short errnumb = ntos(body);
214     char const* errdesc = "";
215     if (bodylen > ERROR_NUMB_SIZE)
216         errdesc = body + ERROR_NUMB_SIZE;
217
218     logger_error("beat fail, errnumb: %d, errdesc: %s", errnumb, errdesc);
219     return ERROR;
220 }

```

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

```

1  // 存储服务器
2  // 声明服务器类
3  //
4  #pragma once
5
6  #include <list>
7  #include <lib_acl.hpp>
8  #include "13_tracker.h"
9  //
10 // 服务器类
11 //
12 class server_c: public acl::master_threads {
13 protected:
14     // 进程切换用户后被调用
15     void proc_on_init(void);
16     // 子进程意图退出时被调用
17     // 返回true, 子进程立即退出, 否则
18     // 若配置项ioctl_quick_abort非0, 子进程立即退出, 否则
19     // 待所有客户机连接都关闭后, 子进程再退出
20     bool proc_exit_timer(size_t nclients, size_t nthreads);
21     // 进程退出前被调用
22     void proc_on_exit(void);
23
24     // 线程获得连接时被调用
25     // 返回true, 连接将被用于后续通信, 否则
26     // 函数返回后即关闭连接
27     bool thread_on_accept(acl::socket_stream* conn);
28     // 与线程绑定的连接可读时被调用

```

```

29 // 返回true, 保持长连接, 否则
30 // 函数返回后即关闭连接
31 bool thread_on_read(ac1::socket_stream* conn);
32 // 线程读写连接超时时被调用
33 // 返回true, 继续等待下一次读写, 否则
34 // 函数返回后即关闭连接
35 bool thread_on_timeout(ac1::socket_stream* conn);
36 // 与线程绑定的连接关闭时被调用
37 void thread_on_close(ac1::socket_stream* conn);
38
39 private:
40     std::list<tracker_c*> m_trackers; // 跟踪客户机线程集
41 };

```

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

```

1 // 存储服务器
2 // 实现服务器类
3 //
4 #include <unistd.h>
5 #include "02_proto.h"
6 #include "03_util.h"
7 #include "01_globals.h"
8 #include "11_service.h"
9 #include "15_server.h"
10
11 // 进程切换用户后被调用
12 void server_c::proc_on_init(void) {
13     // 隶属组名
14     if (strlen(cfg_gpname) > STORAGE_GROUPNAME_MAX)
15         logger_fatal("groupname too big %lu > %d",
16                     strlen(cfg_gpname), STORAGE_GROUPNAME_MAX);
17
18     // 绑定端口号
19     if (cfg_bindport <= 0)
20         logger_fatal("invalid bind port %d <= 0", cfg_bindport);
21
22     // 存储路径表
23     if (!cfg_spaths || !*cfg_spaths)
24         logger_fatal("storage paths is null");
25     split(cfg_spaths, g_spaths);
26     if (g_spaths.empty())
27         logger_fatal("storage paths is empty");
28
29     // 跟踪服务器地址表
30     if (!cfg_taddrs || !*cfg_taddrs)
31         logger_fatal("tracker addresses is null");
32     split(cfg_taddrs, g_taddrs);
33     if (g_taddrs.empty())
34         logger_fatal("tracker addresses is empty");
35
36     // ID服务器地址表
37     if (!cfg_iaddrs || !*cfg_iaddrs)
38         logger_fatal("id addresses is null");
39     split(cfg_iaddrs, g_iaddrs);

```



```

40     if (g_iaddrs.empty())
41         logger_fatal("id addresses is empty");
42
43     // MySQL地址表
44     if (!cfg_maddrs || !*cfg_maddrs)
45         logger_fatal("mysql addresses is null");
46     split(cfg_maddrs, g_maddrs);
47     if (g_maddrs.empty())
48         logger_fatal("mysql addresses is empty");
49
50     // Redis地址表
51     if (!cfg_raddrs || !*cfg_raddrs)
52         logger_error("redis addresses is null");
53     else {
54         split(cfg_raddrs, g_raddrs);
55         if (g_raddrs.empty())
56             logger_error("redis addresses is empty");
57         else {
58             // 遍历Redis地址表, 尝试创建连接池
59             for (std::vector<std::string>::const_iterator raddr =
60                 g_raddrs.begin(); raddr != g_raddrs.end(); ++raddr)
61                 if ((g_rconns = new acl::redis_client_pool(
62                     raddr->c_str(), cfg_maxconns))) {
63                     // 设置Redis连接超时和读写超时
64                     g_rconns->set_timeout(cfg_ctimeout, cfg_rtimeout);
65                     break;
66                 }
67             if (!g_rconns)
68                 logger_error("create redis connection pool fail,
69 cfg_raddrs: %s",
70                             cfg_raddrs);
71         }
72     }
73
74     // 主机名
75     char hostname[256+1] = {};
76     if (gethostname(hostname, sizeof(hostname) - 1))
77         logger_error("call gethostname fail: %s", strerror(errno));
78     g_hostname = hostname;
79
80     // 启动时间
81     g_stime = time(NULL);
82
83     // 创建并启动连接每台跟踪服务器的客户机线程
84     for (std::vector<std::string>::const_iterator taddr = g_taddrs.begin();
85         taddr != g_taddrs.end(); ++taddr) {
86         tracker_c* tracker = new tracker_c(taddr->c_str());
87         tracker->set_detachable(false);
88         tracker->start();
89         m_trackers.push_back(tracker);
90     }
91
92     // 打印配置信息
93     logger("cfg_gpname: %s, cfg_spaths: %s, cfg_taddrs: %s, "
94           "cfg_iaddrs: %s, cfg_maddrs: %s, cfg_raddrs: %s, "
95           "cfg_bindport: %d, cfg_interval: %d, cfg_mtimeout: %d, "

```

```

95         "cfg_maxconns: %d, cfg_ctimeout: %d, cfg_rtimeout: %d, "
96         "cfg_ktimeout: %d",
97         cfg_gpname, cfg_spaths, cfg_taddrs,
98         cfg_iaddrs, cfg_maddrs, cfg_raddrs,
99         cfg_bindport, cfg_interval, cfg_mtimeout,
100         cfg_maxconns, cfg_ctimeout, cfg_rtimeout,
101         cfg_ktimeout);
102     }
103
104     // 子进程意图退出时被调用
105     // 返回true, 子进程立即退出, 否则
106     // 若配置项ioctl_quick_abort非0, 子进程立即退出, 否则
107     // 待所有客户机连接都关闭后, 子进程再退出
108     bool server_c::proc_exit_timer(size_t nclients, size_t nthreads) {
109         for (std::list<tracker_c*>::iterator tracker = m_trackers.begin();
110              tracker != m_trackers.end(); ++tracker)
111             // 终止跟踪客户机线程
112             (*tracker)->stop();
113
114         if (!nclients || !nthreads) {
115             logger("nclients: %lu, nthreads: %lu", nclients, nthreads);
116             return true;
117         }
118
119         return false;
120     }
121
122     // 进程退出前被调用
123     void server_c::proc_on_exit(void) {
124         for (std::list<tracker_c*>::iterator tracker = m_trackers.begin();
125              tracker != m_trackers.end(); ++tracker) {
126             // 回收跟踪客户机线程
127             if (!(*tracker)->wait(NULL))
128                 logger_error("wait thread #%lu fail", (*tracker)->thread_id());
129             // 销毁跟踪客户机线程
130             delete *tracker;
131         }
132
133         m_trackers.clear();
134
135         // 销毁Redis连接池
136         if (g_rconns) {
137             delete g_rconns;
138             g_rconns = NULL;
139         }
140     }
141
142     // 线程获得连接时被调用
143     // 返回true, 连接将被用于后续通信, 否则
144     // 函数返回后即关闭连接
145     bool server_c::thread_on_accept(ac1::socket_stream* conn) {
146         logger("connect, from: %s", conn->get_peer());
147         return true;
148     }
149
150     // 与线程绑定的连接可读时被调用

```

```

151 // 返回true，保持长连接，否则
152 // 函数返回后即关闭连接
153 bool server_c::thread_on_read(ac1::socket_stream* conn) {
154     // 接收包头
155     char head[HEADLEN];
156     if (conn->read(head, HEADLEN) < 0) {
157         if (conn->eof())
158             logger("connection has been closed, from: %s",
159                 conn->get_peer());
160         else
161             logger_error("read fail: %s, from: %s",
162                 ac1::last_serror(), conn->get_peer());
163         return false;
164     }
165
166     // 业务处理
167     service_c service;
168     return service.business(conn, head);
169 }
170
171 // 线程读写连接超时时被调用
172 // 返回true，继续等待下一次读写，否则
173 // 函数返回后即关闭连接
174 bool server_c::thread_on_timeout(ac1::socket_stream* conn) {
175     logger("read timeout, from: %s", conn->get_peer());
176     return true;
177 }
178
179 // 与线程绑定的连接关闭时被调用
180 void server_c::thread_on_close(ac1::socket_stream* conn) {
181     logger("client disconnect, from: %s", conn->get_peer());
182 }

```

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

```

1 // 存储服务器
2 // 定义主函数
3 //
4 #include "01_globals.h"
5 #include "15_server.h"
6
7 int main(void) {
8     // 初始化ACL库
9     ac1::acl_cpp_init();
10    ac1::log::stdout_open(true);
11
12    // 创建并运行服务器
13    server_c& server = ac1::singleton2<server_c>::get_instance();
14    server.set_cfg_str(cfg_str);
15    server.set_cfg_int(cfg_int);
16    server.run_alone("127.0.0.1:23000", "../etc/storage.cfg");
17
18    return 0;
19 }

```

~/TNV/src/04_storage/Makefile

```
1  PROJ    = ../../bin/storage
2  OBJS     = $(patsubst %.cpp, %.o, $(wildcard ../01_common/*.cpp *.cpp))
3  CC       = g++
4  LINK     = g++
5  RM       = rm -rf
6  CFLAGS   = -c -Wall \
7            -I/usr/include/acl-lib/acl_cpp \
8            `mysql_config --cflags` \
9            -I../01_common
10 LIBS     = -pthread -lacl_all `mysql_config --libs`
11
12 all: $(PROJ)
13
14 $(PROJ): $(OBJS)
15     $(LINK) $^ $(LIBS) -o $@
16
17 .cpp.o:
18     $(CC) $(CFLAGS) $^ -o $@
19
20 clean:
21     $(RM) $(PROJ) $(OBJS)
```

~/TNV/etc/storage.cfg

```
1  service storage {
2      # 隶属组名
3      tnv_group_name = group001
4      # 存储路径表
5      tnv_store_paths = ../data
6      # 跟踪服务器地址表
7      tnv_tracker_addrs = 127.0.0.1:21000
8      # ID服务器地址表
9      tnv_ids_addrs = 127.0.0.1:22000
10     # MySQL地址表
11     mysql_addrs = 127.0.0.1
12     # Redis地址表
13     redis_addrs = 127.0.0.1:6379
14     # 绑定端口号
15     tnv_storage_port = 23000
16     # 心跳间隔秒数
17     tnv_heart_beat_interval = 10
18     # MySQL读写超时
19     mysql_rw_timeout = 30
20     # Redis连接池最大连接数
21     redis_max_conn_num = 600
22     # Redis连接超时
23     redis_conn_timeout = 10
24     # Redis读写超时
25     redis_rw_timeout = 10
26     # Redis键超时
27     redis_key_timeout = 60
```

~/TNV/sql/storage.sql

```
1 DROP DATABASE IF EXISTS tnv_storagedb;
2 CREATE DATABASE tnv_storagedb;
3 USE tnv_storagedb;
4
5 CREATE TABLE `t_file_01` (
6   `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
7   `appid` varchar(32) DEFAULT NULL,
8   `userid` varchar(128) DEFAULT NULL,
9   `status` tinyint(4) DEFAULT NULL,
10  `file_path` varchar(512) DEFAULT NULL,
11  `file_size` bigint(20) DEFAULT NULL,
12  `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
13  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP,
14  PRIMARY KEY (`id`)
15 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
16
17 CREATE TABLE `t_file_02` (
18   `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
19   `appid` varchar(32) DEFAULT NULL,
20   `userid` varchar(128) DEFAULT NULL,
21   `status` tinyint(4) DEFAULT NULL,
22   `file_path` varchar(512) DEFAULT NULL,
23   `file_size` bigint(20) DEFAULT NULL,
24   `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
25   `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP,
26  PRIMARY KEY (`id`)
27 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
28
29 CREATE TABLE `t_file_03` (
30   `id` varchar(256) NOT NULL DEFAULT '' COMMENT '文件ID',
31   `appid` varchar(32) DEFAULT NULL,
32   `userid` varchar(128) DEFAULT NULL,
33   `status` tinyint(4) DEFAULT NULL,
34   `file_path` varchar(512) DEFAULT NULL,
35   `file_size` bigint(20) DEFAULT NULL,
36   `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
37   `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP,
38  PRIMARY KEY (`id`)
39 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```