5 客户机

~/TNV/src/05_client/01_conn.h

```
// 客户机
// 声明连接类
//
#pragma once

#include <string>
#include <lib_acl.hpp>
//
// 连接类
//
class conn_c: public acl::connect_client {
public:
    // 构造函数
    conn_c(char const* destaddr, int ctimeout = 30,
        int rtimeout = 60);
    // 析构函数
    ~conn_c(void);

    // 从跟踪服务器获取存储服务器地址列表
    int saddrs(char const* appid, char const* userid,
        char const* fileid, std::string& saddrs);
    // 从跟踪服务器获取组列表
    int groups(std::string& groups);

    // 向存储服务器上传文件
    int upload(char const* appid, char const* userid,
        char const* fileid, char const* filepath);
    // 向存储服务器上传文件
    int upload(char const* appid, char const* userid,
        char const* fileid, char const* filedata, long long filesize);
    // 向存储服务器询问文件大小
    int filesize(char const* appid, char const* userid,
        char const* fileid, long long* filesize);
    // 从存储服务器下载文件
    int download(char const* appid, char const* userid,
        char const* fileid, long long offset, long long size,
        char** filedata, long long* filesize);
    // 删除存储服务器上的文件
    int del(char const* appid, char const* userid, char const* fileid);

    // 获取错误号
    short errnumb(void) const;
    // 获取错误描述
    char const* errdesc(void) const;

protected:
    // 打开连接
    bool open(void);
    // 关闭连接
```

```
    void close(void);

private:
    // 构造请求
    int makerequ(char command, char const* appid,
        char const* userid, char const* fileid, char* requ);
    // 接收包体
    int recvbody(char** body, long long* bodylen);
    // 接收包头
    int recvhead(long long* bodylen);

    char*               m_destaddr; // 目的地址
    int                 m_ctimeout; // 连接超时
    int                 m_rtimeout; // 读写超时
    acl::socket_stream* m_conn;     // 连接对象
    short               m_errnumb;  // 错误号
    acl::string         m_errdesc;  // 错误描述
};
```

~/TNV/src/05_client/02_conn.cpp

```
// 客户机
// 实现连接类
//
#include <sys/sendfile.h>
#include <lib_acl.h>
#include "02_proto.h"
#include "03_util.h"
#include "01_conn.h"

// 构造函数
conn_c::conn_c(char const* destaddr, int ctimeout /* = 30 */,
    int rtimeout /* = 60 */): m_ctimeout(ctimeout),
    m_rtimeout(rtimeout), m_conn(NULL) {
    // 检查目的地址
    acl_assert(destaddr && *destaddr);
    // 复制目的地址
    m_destaddr = acl_mystrdup(destaddr);
}

// 析构函数
conn_c::~conn_c(void) {
    // 关闭连接
    close();
    // 释放目的地址
    acl_myfree(m_destaddr);
}

// 从跟踪服务器获取存储服务器地址列表
int conn_c::saddrs(char const* appid, char const* userid,
    char const* fileid, std::string& saddrs) {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|
    // |    8   |  1 |  1 |  16  |  256 |  128 |
    // 构造请求
    long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
```

```
    long long requlen = HEADLEN + bodylen;
    char requ[requlen];
    if (makerequ(CMD_TRACKER_SADDRS,
        appid, userid, fileid, requ) != OK)
        return ERROR;
    llton(bodylen, requ);

    if (!open())
        return SOCKET_ERROR;

    // 发送请求
    if (m_conn->write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        close();
        return SOCKET_ERROR;
    }

    char* body = NULL; // 包体指针

    // 接收包体
    int result = recvbody(&body, &bodylen);

    // 解析包体
    if (result == OK)
        // |包体长度|命令|状态|组名|存储服务器地址列表|
        // |    8   |  1 |  1 |16+1|  包体长度-(16+1) |
        saddrs = body + STORAGE_GROUPNAME_MAX + 1;
    else if (result == STATUS_ERROR) {
        // |包体长度|命令|状态|错误号|错误描述|
        // |    8   |  1 |  1 |   2  | <=1024 |
        m_errnumb = ntos(body);
        m_errdesc = bodylen > ERROR_NUMB_SIZE ?
            body + ERROR_NUMB_SIZE : "";
    }

    // 释放包体
    if (body) {
        free(body);
        body = NULL;
    }

    return result;
}

// 从跟踪服务器获取组列表
int conn_c::groups(std::string& groups) {
    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 构造请求
    long long bodylen = 0;
    long long requlen = HEADLEN + bodylen;
    char requ[requlen] = {};
```

```
    llton(bodylen, requ);
    requ[BODYLEN_SIZE] = CMD_TRACKER_GROUPS;
    requ[BODYLEN_SIZE+COMMAND_SIZE] = 0;

    if (!open())
        return SOCKET_ERROR;

    // 发送请求
    if (m_conn->write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        close();
        return SOCKET_ERROR;
    }

    char* body = NULL; // 包体指针

    // 接收包体
    int result = recvbody(&body, &bodylen);

    // 解析包体
    if (result == OK)
        // |包体长度|命令|状态| 组列表 |
        // |    8   |  1 |  1 |包体长度|
        groups = body;
    else if (result == STATUS_ERROR) {
        // |包体长度|命令|状态|错误号|错误描述|
        // |    8   |  1 |  1 |   2  | <=1024 |
        m_errnumb = ntos(body);
        m_errdesc = bodylen > ERROR_NUMB_SIZE ?
            body + ERROR_NUMB_SIZE : "";
    }

    // 释放包体
    if (body) {
        free(body);
        body = NULL;
    }

    return result;
}

// 向存储服务器上传文件
int conn_c::upload(char const* appid, char const* userid,
    char const* fileid, char const* filepath) {
    // |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|
    // |    8   |  1 |  1 |  16  |  256 |  128 |    8   |文件大小|
    // 构造请求
    long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
        BODYLEN_SIZE;
    long long requlen = HEADLEN + bodylen;
    char requ[requlen];
    if (makerequ(CMD_STORAGE_UPLOAD,
```

```
147         appid, userid, fileid, requ) != OK)
148         return ERROR;
149     acl::ifstream ifs;
150     if (!ifs.open_read(filepath)) {
151         logger_error("open file fail, filepath: %s", filepath);
152         return ERROR;
153     }
154     long long filesize = ifs.fsize();
155     llton(filesize, requ + HEADLEN +
156         APPID_SIZE + USERID_SIZE + FILEID_SIZE);
157     bodylen += filesize;
158     llton(bodylen, requ);
159 
160     if (!open())
161         return SOCKET_ERROR;
162 
163     // 发送请求
164     if (m_conn->write(requ, requlen) < 0) {
165         logger_error("write fail: %s, requlen: %lld, to: %s",
166             acl::last_serror(), requlen, m_conn->get_peer());
167         m_errnumb = -1;
168         m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
169             acl::last_serror(), requlen, m_conn->get_peer());
170         close();
171         return SOCKET_ERROR;
172     }
173 
174     // 发送文件
175     long long remain = filesize; // 未发送字节数
176     off_t offset = 0; // 文件读写位置
177     while (remain) { // 还有未发送数据
178         // 发送数据
179         long long bytes = std::min(remain,
180             (long long)STORAGE_RCVWR_SIZE);
181         long long count = sendfile(m_conn->sock_handle(),
182             ifs.file_handle(), &offset, bytes);
183         if (count < 0) {
184             logger_error(
185                 "send file fail, filesize: %lld, remain: %lld",
186                 filesize, remain);
187             m_errnumb = -1;
188             m_errdesc.format(
189                 "send file fail, filesize: %lld, remain: %lld",
190                 filesize, remain);
191             close();
192             return SOCKET_ERROR;
193         }
194         // 未发递减
195         remain -= count;
196     }
197     ifs.close();
198 
199     char* body = NULL; // 包体指针
200 
201     // 接收包体
202     int result = recvbody(&body, &bodylen);
```

```
203
204     // 解析包体
205     if (result == STATUS_ERROR) {
206         // |包体长度|命令|状态|错误号|错误描述|
207         // |    8   |  1 |  1 |   2  | <=1024 |
208         m_errnumb = ntos(body);
209         m_errdesc = bodylen > ERROR_NUMB_SIZE ?
210             body + ERROR_NUMB_SIZE : "";
211     }
212
213     // 释放包体
214     if (body) {
215         free(body);
216         body = NULL;
217     }
218
219     return result;
220 }
221
222 // 向存储服务器上传文件
223 int conn_c::upload(char const* appid, char const* userid,
224     char const* fileid, char const* filedata, long long filesize) {
225     // |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|
226     // |    8   |  1 |  1 |  16  |  256 |  128 |    8   |文件大小|
227     // 构造请求
228     long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
229         BODYLEN_SIZE;
230     long long requlen = HEADLEN + bodylen;
231     char requ[requlen];
232     if (makerequ(CMD_STORAGE_UPLOAD,
233             appid, userid, fileid, requ) != OK)
234         return ERROR;
235     llton(filesize, requ + HEADLEN +
236         APPID_SIZE + USERID_SIZE + FILEID_SIZE);
237     bodylen += filesize;
238     llton(bodylen, requ);
239
240     if (!open())
241         return SOCKET_ERROR;
242
243     // 发送请求
244     if (m_conn->write(requ, requlen) < 0) {
245         logger_error("write fail: %s, requlen: %lld, to: %s",
246             acl::last_serror(), requlen, m_conn->get_peer());
247         m_errnumb = -1;
248         m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
249             acl::last_serror(), requlen, m_conn->get_peer());
250         close();
251         return SOCKET_ERROR;
252     }
253
254     // 发送文件
255     if (m_conn->write(filedata, filesize) < 0) {
256         logger_error("write fail: %s, filesize: %lld, to: %s",
257             acl::last_serror(), filesize, m_conn->get_peer());
258         m_errnumb = -1;
```

```
        m_errdesc.format("write fail: %s, filesize: %lld, to: %s",
            acl::last_serror(), filesize, m_conn->get_peer());
        close();
        return SOCKET_ERROR;
    }

    char* body = NULL; // 包体指针

    // 接收包体
    int result = recvbody(&body, &bodylen);

    // 解析包体
    if (result == STATUS_ERROR) {
        // |包体长度|命令|状态|错误号|错误描述|
        // |    8   |  1 |  1 |   2  | <=1024 |
        m_errnumb = ntos(body);
        m_errdesc = bodylen > ERROR_NUMB_SIZE ?
            body + ERROR_NUMB_SIZE : "";
    }

    // 释放包体
    if (body) {
        free(body);
        body = NULL;
    }

    return result;
}

// 向存储服务器询问文件大小
int conn_c::filesize(char const* appid, char const* userid,
    char const* fileid, long long* filesize) {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|
    // |    8   |  1 |  1 |  16  |  256 |  128 |
    // 构造请求
    long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
    long long requlen = HEADLEN + bodylen;
    char requ[requlen];
    if (makerequ(CMD_STORAGE_FILESIZE,
        appid, userid, fileid, requ) != OK)
        return ERROR;
    llton(bodylen, requ);

    if (!open())
        return SOCKET_ERROR;

    // 发送请求
    if (m_conn->write(requ, requlen) < 0) {
        logger_error("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
            acl::last_serror(), requlen, m_conn->get_peer());
        close();
        return SOCKET_ERROR;
    }
```

```
315
316     char* body = NULL; // 包体指针
317
318     // 接收包体
319     int result = recvbody(&body, &bodylen);
320
321     // 解析包体
322     if (result == OK)
323         // |包体长度|命令|状态|文件大小|
324         // |    8   |  1 |  1 |    8   |
325         *filesize = ntoll(body);
326     else if (result == STATUS_ERROR) {
327         // |包体长度|命令|状态|错误号|错误描述|
328         // |    8   |  1 |  1 |   2  | <=1024 |
329         m_errnumb = ntos(body);
330         m_errdesc = bodylen > ERROR_NUMB_SIZE ?
331             body + ERROR_NUMB_SIZE : "";
332     }
333
334     // 释放包体
335     if (body) {
336         free(body);
337         body = NULL;
338     }
339
340     return result;
341 }
342
343 // 从存储服务器下载文件
344 int conn_c::download(char const* appid, char const* userid,
345     char const* fileid, long long offset, long long size,
346     char** filedata, long long* filesize) {
347     // |包体长度|命令|状态|应用ID|用户ID|文件ID|偏移|大小|
348     // |    8   |  1 |  1 |  16  |  256 |  128 |  8 |  8 |
349     // 构造请求
350     long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
351         BODYLEN_SIZE + BODYLEN_SIZE;
352     long long requlen = HEADLEN + bodylen;
353     char requ[requlen];
354     if (makerequ(CMD_STORAGE_DOWNLOAD,
355         appid, userid, fileid, requ) != OK)
356         return ERROR;
357     llton(bodylen, requ);
358     llton(offset, requ + HEADLEN +
359         APPID_SIZE + USERID_SIZE + FILEID_SIZE);
360     llton(size, requ + HEADLEN +
361         APPID_SIZE + USERID_SIZE + FILEID_SIZE + BODYLEN_SIZE);
362
363     if (!open())
364         return SOCKET_ERROR;
365
366     // 发送请求
367     if (m_conn->write(requ, requlen) < 0) {
368         logger_error("write fail: %s, requlen: %lld, to: %s",
369             acl::last_serror(), requlen, m_conn->get_peer());
370         m_errnumb = -1;
```

```
371         m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
372             acl::last_serror(), requlen, m_conn->get_peer());
373         close();
374         return SOCKET_ERROR;
375     }
376 
377     char* body = NULL; // 包体指针
378 
379     // 接收包体
380     int result = recvbody(&body, &bodylen);
381 
382     // 解析包体
383     if (result == OK) {
384         // |包体长度|命令|状态|文件内容|
385         // |    8   |  1 |  1 |内容大小|
386         *filedata = body;
387         *filesize = bodylen;
388         return result;
389     }
390     if (result == STATUS_ERROR) {
391         // |包体长度|命令|状态|错误号|错误描述|
392         // |    8   |  1 |  1 |   2  | <=1024 |
393         m_errnumb = ntos(body);
394         m_errdesc = bodylen > ERROR_NUMB_SIZE ?
395             body + ERROR_NUMB_SIZE : "";
396     }
397 
398     // 释放包体
399     if (body) {
400         free(body);
401         body = NULL;
402     }
403 
404     return result;
405 }
406 
407 // 删除存储服务器上的文件
408 int conn_c::del(char const* appid, char const* userid,
409     char const* fileid) {
410     // |包体长度|命令|状态|应用ID|用户ID|文件ID|
411     // |    8   |  1 |  1 |  16  |  256 |  128 |
412     // 构造请求
413     long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE;
414     long long requlen = HEADLEN + bodylen;
415     char requ[requlen];
416     if (makerequ(CMD_STORAGE_DELETE,
417         appid, userid, fileid, requ) != OK)
418         return ERROR;
419     llton(bodylen, requ);
420 
421     if (!open())
422         return SOCKET_ERROR;
423 
424     // 发送请求
425     if (m_conn->write(requ, requlen) < 0) {
426         logger_error("write fail: %s, requlen: %lld, to: %s",
```

```
427             acl::last_serror(), requlen, m_conn->get_peer());
428         m_errnumb = -1;
429         m_errdesc.format("write fail: %s, requlen: %lld, to: %s",
430             acl::last_serror(), requlen, m_conn->get_peer());
431         close();
432         return SOCKET_ERROR;
433     }
434 
435     char* body = NULL; // 包体指针
436 
437     // 接收包体
438     int result = recvbody(&body, &bodylen);
439 
440     // 解析包体
441     if (result == STATUS_ERROR) {
442         // |包体长度|命令|状态|错误号|错误描述|
443         // |    8   |  1 |  1 |   2  | <=1024 |
444         m_errnumb = ntos(body);
445         m_errdesc = bodylen > ERROR_NUMB_SIZE ?
446             body + ERROR_NUMB_SIZE : "";
447     }
448 
449     // 释放包体
450     if (body) {
451         free(body);
452         body = NULL;
453     }
454 
455     return result;
456 }
457 
458 // 获取错误号
459 short conn_c::errnumb(void) const {
460     return m_errnumb;
461 }
462 
463 // 获取错误描述
464 char const* conn_c::errdesc(void) const {
465     return m_errdesc.c_str();
466 }
467 
468 // 打开连接
469 bool conn_c::open(void) {
470     if (m_conn)
471         return true;
472 
473     // 创建连接对象
474     m_conn = new acl::socket_stream;
475 
476     // 连接目的主机
477     if (!m_conn->open(m_destaddr, m_ctimeout, m_rtimeout)) {
478         logger_error("open %s fail: %s",
479             m_destaddr, acl_last_serror());
480         delete m_conn;
481         m_conn = NULL;
482         m_errnumb = -1;
```

```
        m_errdesc.format("open %s fail: %s",
            m_destaddr, acl_last_serror());
        return false;
    }

    return true;
}

// 关闭连接
void conn_c::close(void) {
    if (m_conn) {
        delete m_conn;
        m_conn = NULL;
    }
}

// 构造请求
int conn_c::makerequ(char command, char const* appid,
    char const* userid, char const* fileid, char* requ) {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|
    // |    8   |  1 |  1 |  16  |  256 |  128 |
    requ[BODYLEN_SIZE] = command; // 命令
    requ[BODYLEN_SIZE+COMMAND_SIZE] = 0; // 状态

    // 应用ID
    if (strlen(appid) >= APPID_SIZE) {
        logger_error("appid too big, %lu >= %d",
            strlen(appid), APPID_SIZE);
        m_errnumb = -1;
        m_errdesc.format("appid too big, %lu >= %d",
            strlen(appid), APPID_SIZE);
        return ERROR;
    }
    strcpy(requ + HEADLEN, appid);

    // 用户ID
    if (strlen(userid) >= USERID_SIZE) {
        logger_error("userid too big, %lu >= %d",
            strlen(userid), USERID_SIZE);
        m_errnumb = -1;
        m_errdesc.format("userid too big, %lu >= %d",
            strlen(userid), USERID_SIZE);
        return ERROR;
    }
    strcpy(requ + HEADLEN + APPID_SIZE, userid);

    // 文件ID
    if (strlen(fileid) >= FILEID_SIZE) {
        logger_error("fileid too big, %lu >= %d",
            strlen(fileid), FILEID_SIZE);
        m_errnumb = -1;
        m_errdesc.format("fileid too big, %lu >= %d",
            strlen(fileid), FILEID_SIZE);
        return ERROR;
    }
    strcpy(requ + HEADLEN + APPID_SIZE + USERID_SIZE, fileid);
```

```
539
540     return OK;
541 }
542
543 // 接收包体
544 int conn_c::recvbody(char** body, long long* bodylen) {
545     // 接收包头
546     int result = recvhead(bodylen);
547
548     // 既非本地错误亦非套接字通信错误且包体非空
549     if (result != ERROR && result != SOCKET_ERROR && *bodylen) {
550         // 分配包体
551         if (!(*body = (char*)malloc(*bodylen))) {
552             logger_error("call malloc fail: %s, bodylen: %lld",
553                 strerror(errno), *bodylen);
554             m_errnumb = -1;
555             m_errdesc.format("call malloc fail: %s, bodylen: %lld",
556                 strerror(errno), *bodylen);
557             return ERROR;
558         }
559
560         // 接收包体
561         if (m_conn->read(*body, *bodylen) < 0) {
562             logger_error("read fail: %s, from: %s",
563                 acl::last_serror(), m_conn->get_peer());
564             m_errnumb = -1;
565             m_errdesc.format("read fail: %s, from: %s",
566                 acl::last_serror(), m_conn->get_peer());
567             free(*body);
568             *body = NULL;
569             close();
570             return SOCKET_ERROR;
571         }
572     }
573
574     return result;
575 }
576
577 // 接收包头
578 int conn_c::recvhead(long long* bodylen) {
579     if (!open())
580         return SOCKET_ERROR;
581
582     char head[HEADLEN]; // 包头缓冲区
583
584     // 接收包头
585     if (m_conn->read(head, HEADLEN) < 0) {
586         logger_error("read fail: %s, from: %s",
587             acl::last_serror(), m_conn->get_peer());
588         m_errnumb = -1;
589         m_errdesc.format("read fail: %s, from: %s",
590             acl::last_serror(), m_conn->get_peer());
591         close();
592         return SOCKET_ERROR;
593     }
594
```

```
    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 解析包头
    if ((*bodylen = ntoll(head)) < 0) { // 包体长度
        logger_error("invalid body length: %lld < 0, from: %s",
            *bodylen, m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("invalid body length: %lld < 0, from: %s",
            *bodylen, m_conn->get_peer());
        return ERROR;
    }
    int command = head[BODYLEN_SIZE]; // 命令
    int status = head[BODYLEN_SIZE+COMMAND_SIZE]; // 状态
    if (status) {
        logger_error("response status %d != 0, from: %s",
            status, m_conn->get_peer());
        return STATUS_ERROR;
    }
    logger("bodylen: %lld, command: %d, status: %d",
        *bodylen, command, status);

    return OK;
}
```

~/TNV/src/05_client/03_pool.h

```
// 客户机
// 声明连接池类
//
#pragma once

#include <lib_acl.hpp>
//
// 连接池类
//
class pool_c: public acl::connect_pool {
public:
    // 构造函数
    pool_c(char const* destaddr, int count, size_t index = 0);

    // 设置超时
    void timeouts(int ctimeout = 30, int rtimeout = 60,
        int itimeout = 90);
    // 获取连接
    acl::connect_client* peek(void);

protected:
    // 创建连接
    acl::connect_client* create_connect(void);

private:
    int m_ctimeout; // 连接超时
    int m_rtimeout; // 读写超时
    int m_itimeout; // 空闲超时
```

```
};
```

~/TNV/src/05_client/04_pool.cpp

```cpp
// 客户机
// 实现连接池类
//
#include "01_conn.h"
#include "03_pool.h"

// 构造函数
pool_c::pool_c(char const* destaddr, int count, size_t index /* = 0 */):
    connect_pool(destaddr, count, index),
    m_ctimeout(30), m_rtimeout(60), m_itimeout(90) {
}

// 设置超时
void pool_c::timeouts(int ctimeout /* = 30 */, int rtimeout /* = 60 */,
    int itimeout /* = 90 */) {
    m_ctimeout = ctimeout;
    m_rtimeout = rtimeout;
    m_itimeout = itimeout;
}

// 获取连接
acl::connect_client* pool_c::peek(void) {
    connect_pool::check_idle(m_itimeout);
    return connect_pool::peek();
}

// 创建连接
acl::connect_client* pool_c::create_connect(void) {
    return new conn_c(addr_, m_ctimeout, m_rtimeout);
}
```

~/TNV/src/05_client/05_mngr.h

```cpp
// 客户机
// 声明连接池管理器类
//
#pragma once

#include <lib_acl.hpp>
//
// 连接池管理器类
//
class mngr_c: public acl::connect_manager {
protected:
    // 创建连接池
    acl::connect_pool* create_pool(
        char const* destaddr, size_t count, size_t index);
};
```

~/TNV/src/05_client/06_mngr.cpp

```cpp
// 客户机
// 实现连接池管理器类
//
#include "03_pool.h"
#include "05_mngr.h"

// 创建连接池
acl::connect_pool* mngr_c::create_pool(
    char const* destaddr, size_t count, size_t index) {
    return new pool_c(destaddr, count, index);
}
```

~/TNV/src/05_client/07_client.h

```cpp
// 客户机
// 声明客户机类
//
#pragma once

#include <vector>
#include <string>
#include <lib_acl.hpp>
//
// 客户机类
//
class client_c {
public:
    // 初始化
    static int init(char const* taddrs,
        int tcount = 16, int scount = 8);
    // 终结化
    static void deinit(void);

    // 从跟踪服务器获取存储服务器地址列表
    int saddrs(char const* appid, char const* userid,
        char const* fileid, std::string& saddrs);
    // 从跟踪服务器获取组列表
    int groups(std::string& groups);

    // 向存储服务器上传文件
    int upload(char const* appid, char const* userid,
        char const* fileid, char const* filepath);
    // 向存储服务器上传文件
    int upload(char const* appid, char const* userid,
        char const* fileid, char const* filedata, long long filesize);
    // 向存储服务器询问文件大小
    int filesize(char const* appid, char const* userid,
        char const* fileid, long long* filesize);
    // 从存储服务器下载文件
    int download(char const* appid, char const* userid,
        char const* fileid, long long offset, long long size,
        char** filedata, long long* filesize);
```

```
    // 删除存储服务器上的文件
    int del(char const* appid, char const* userid, char const* fileid);

private:
    static acl::connect_manager* s_mngr; // 连接池管理器
    static std::vector<std::string> s_taddrs; // 跟踪服务器地址表
    static int s_scount; // 存储服务器连接数上限
};
```

~/TNV/src/05_client/08_client.cpp

```
// 客户机
// 实现客户机类
//
#include <lib_acl.h>
#include "01_types.h"
#include "03_util.h"
#include "01_conn.h"
#include "03_pool.h"
#include "05_mngr.h"
#include "07_client.h"

#define MAX_SOCKERRS 10 // 套接字通信错误最大次数

acl::connect_manager* client_c::s_mngr = NULL;
std::vector<std::string> client_c::s_taddrs;
int client_c::s_scount = 8;

// 初始化
int client_c::init(char const* taddrs,
    int tcount /* = 16 */, int scount /* = 8 */) {
    if (s_mngr)
        return OK;

    // 跟踪服务器地址表
    if (!taddrs || !*taddrs) {
        logger_error("tracker addresses is null");
        return ERROR;
    }
    split(taddrs, s_taddrs);
    if (s_taddrs.empty()) {
        logger_error("tracker addresses is empty");
        return ERROR;
    }

    // 跟踪服务器连接数上限
    if (tcount <= 0) {
        logger_error("invalid tracker connection pool count %d <= 0",
            tcount);
        return ERROR;
    }

    // 存储服务器连接数上限
    if (scount <= 0) {
        logger_error("invalid storage connection pool count %d <= 0",
```

```
            scount);
        return ERROR;
    }
    s_scount = scount;

    // 创建连接池管理器
    if (!(s_mngr = new mngr_c)) {
        logger_error("create connection pool manager fail: %s",
            acl_last_serror());
        return ERROR;
    }

    // 初始化跟踪服务器连接池
    s_mngr->init(NULL, taddrs, tcount);

    return OK;
}

// 终结化
void client_c::deinit(void) {
    if (s_mngr) {
        delete s_mngr;
        s_mngr = NULL;
    }

    s_taddrs.clear();
}

// 从跟踪服务器获取存储服务器地址列表
int client_c::saddrs(char const* appid, char const* userid,
    char const* fileid, std::string& saddrs) {
    if (s_taddrs.empty()) {
        logger_error("tracker addresses is empty");
        return ERROR;
    }

    int result = ERROR;

    // 生成有限随机数
    srand(time(NULL));
    int ntaddrs = s_taddrs.size();
    int nrand = rand() % ntaddrs;

    for (int i = 0; i < ntaddrs; ++i) {
        // 随机抽取跟踪服务器地址
        char const* taddr = s_taddrs[nrand].c_str();
        nrand = (nrand + 1) % ntaddrs;

        // 获取跟踪服务器连接池
        pool_c* tpool = (pool_c*)s_mngr->get(taddr);
        if (!tpool) {
            logger_warn("tracker connection pool is null, taddr: %s",
                taddr);
            continue;
        }

```

```
        for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
            // 获取跟踪服务器连接
            conn_c* tconn = (conn_c*)tpool->peek();
            if (!tconn) {
                logger_warn("tracker connection is null, taddr: %s",
                    taddr);
                break;
            }

            // 从跟踪服务器获取存储服务器地址列表
            result = tconn->saddrs(appid, userid, fileid, saddrs);

            if (result == SOCKET_ERROR) {
                logger_warn("get storage addresses fail: %s",
                    tconn->errdesc());
                tpool->put(tconn, false);
            }
            else {
                if (result == OK)
                    tpool->put(tconn, true);
                else {
                    logger_error("get storage addresses fail: %s",
                        tconn->errdesc());
                    tpool->put(tconn, false);
                }
                return result;
            }
        }
    }

    return result;
}

// 从跟踪服务器获取组列表
int client_c::groups(std::string& groups) {
    if (s_taddrs.empty()) {
        logger_error("tracker addresses is empty");
        return ERROR;
    }

    int result = ERROR;

    // 生成有限随机数
    srand(time(NULL));
    int ntaddrs = s_taddrs.size();
    int nrand = rand() % ntaddrs;

    for (int i = 0; i < ntaddrs; ++i) {
        // 随机抽取跟踪服务器地址
        char const* taddr = s_taddrs[nrand].c_str();
        nrand = (nrand + 1) % ntaddrs;

        // 获取跟踪服务器连接池
        pool_c* tpool = (pool_c*)s_mngr->get(taddr);
        if (!tpool) {
            logger_warn("tracker connection pool is null, taddr: %s",
```

```
                taddr);
            continue;
        }

        for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
            // 获取跟踪服务器连接
            conn_c* tconn = (conn_c*)tpool->peek();
            if (!tconn) {
                logger_warn("tracker connection is null, taddr: %s",
                    taddr);
                break;
            }

            // 从跟踪服务器获取组列表
            result = tconn->groups(groups);

            if (result == SOCKET_ERROR) {
                logger_warn("get groups fail: %s", tconn->errdesc());
                tpool->put(tconn, false);
            }
            else {
                if (result == OK)
                    tpool->put(tconn, true);
                else {
                    logger_error("get groups fail: %s",
                        tconn->errdesc());
                    tpool->put(tconn, false);
                }
                return result;
            }
        }
    }

    return result;
}

// 向存储服务器上传文件
int client_c::upload(char const* appid, char const* userid,
    char const* fileid, char const* filepath) {
    // 检查参数
    if (!appid || !*appid) {
        logger_error("appid is null");
        return ERROR;
    }
    if (!userid || !*userid) {
        logger_error("userid is null");
        return ERROR;
    }
    if (!fileid || !*fileid) {
        logger_error("fileid is null");
        return ERROR;
    }
    if (!filepath || !*filepath) {
        logger_error("filepath is null");
        return ERROR;
    }
```

```
213
214     // 从跟踪服务器获取存储服务器地址列表
215     int result;
216     std::string ssaddrs;
217     if ((result = saddrs(appid, userid, fileid, ssaddrs)) != OK)
218         return result;
219     std::vector<std::string> vsaddrs;
220     split(ssaddrs.c_str(), vsaddrs);
221     if (vsaddrs.empty()) {
222         logger_error("storage addresses is empty");
223         return ERROR;
224     }
225
226     result = ERROR;
227
228     for (std::vector<std::string>::const_iterator saddr =
229         vsaddrs.begin(); saddr != vsaddrs.end(); ++saddr) {
230         // 获取存储服务器连接池
231         pool_c* spool = (pool_c*)s_mngr->get(saddr->c_str());
232         if (!spool) {
233             s_mngr->set(saddr->c_str(), s_scount);
234             if (!(spool = (pool_c*)s_mngr->get(saddr->c_str()))) {
235                 logger_warn(
236                     "storage connection pool is null, saddr: %s",
237                     saddr->c_str());
238                 continue;
239             }
240         }
241
242         for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
243             // 获取存储服务器连接
244             conn_c* sconn = (conn_c*)spool->peek();
245             if (!sconn) {
246                 logger_warn("storage connection is null, saddr: %s",
247                     saddr->c_str());
248                 break;
249             }
250
251             // 向存储服务器上传文件
252             result = sconn->upload(appid, userid, fileid, filepath);
253
254             if (result == SOCKET_ERROR) {
255                 logger_warn("upload file fail: %s", sconn->errdesc());
256                 spool->put(sconn, false);
257             }
258             else {
259                 if (result == OK)
260                     spool->put(sconn, true);
261                 else {
262                     logger_error("upload file fail: %s",
263                         sconn->errdesc());
264                     spool->put(sconn, false);
265                 }
266                 return result;
267             }
268         }
```

```
    }

    return result;
}

// 向存储服务器上传文件
int client_c::upload(char const* appid, char const* userid,
    char const* fileid, char const* filedata, long long filesize) {
    // 检查参数
    if (!appid || !*appid) {
        logger_error("appid is null");
        return ERROR;
    }
    if (!userid || !*userid) {
        logger_error("userid is null");
        return ERROR;
    }
    if (!fileid || !*fileid) {
        logger_error("fileid is null");
        return ERROR;
    }
    if (!filedata || !filesize) {
        logger_error("filedata is null");
        return ERROR;
    }

    // 从跟踪服务器获取存储服务器地址列表
    int result;
    std::string ssaddrs;
    if ((result = saddrs(appid, userid, fileid, ssaddrs)) != OK)
        return result;
    std::vector<std::string> vsaddrs;
    split(ssaddrs.c_str(), vsaddrs);
    if (vsaddrs.empty()) {
        logger_error("storage addresses is empty");
        return ERROR;
    }

    result = ERROR;

    for (std::vector<std::string>::const_iterator saddr =
        vsaddrs.begin(); saddr != vsaddrs.end(); ++saddr) {
        // 获取存储服务器连接池
        pool_c* spool = (pool_c*)s_mngr->get(saddr->c_str());
        if (!spool) {
            s_mngr->set(saddr->c_str(), s_scount);
            if (!(spool = (pool_c*)s_mngr->get(saddr->c_str()))) {
                logger_warn(
                    "storage connection pool is null, saddr: %s",
                    saddr->c_str());
                continue;
            }
        }

        for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
            // 获取存储服务器连接
```

```
            conn_c* sconn = (conn_c*)spool->peek();
            if (!sconn) {
                logger_warn("storage connection is null, saddr: %s",
                    saddr->c_str());
                break;
            }

            // 向存储服务器上传文件
            result = sconn->upload(
                appid, userid, fileid, filedata, filesize);

            if (result == SOCKET_ERROR) {
                logger_warn("upload file fail: %s", sconn->errdesc());
                spool->put(sconn, false);
            }
            else {
                if (result == OK)
                    spool->put(sconn, true);
                else {
                    logger_error("upload file fail: %s",
                        sconn->errdesc());
                    spool->put(sconn, false);
                }
                return result;
            }
        }
    }

    return result;
}

// 向存储服务器询问文件大小
int client_c::filesize(char const* appid, char const* userid,
    char const* fileid, long long* filesize) {
    // 检查参数
    if (!appid || !*appid) {
        logger_error("appid is null");
        return ERROR;
    }
    if (!userid || !*userid) {
        logger_error("userid is null");
        return ERROR;
    }
    if (!fileid || !*fileid) {
        logger_error("fileid is null");
        return ERROR;
    }

    // 从跟踪服务器获取存储服务器地址列表
    int result;
    std::string ssaddrs;
    if ((result = saddrs(appid, userid, fileid, ssaddrs)) != OK)
        return ERROR;
    std::vector<std::string> vsaddrs;
    split(ssaddrs.c_str(), vsaddrs);
    if (vsaddrs.empty()) {
```

```
381         logger_error("storage addresses is empty");
382         return ERROR;
383     }
384 
385     result = ERROR;
386 
387     for (std::vector<std::string>::const_iterator saddr =
388         vsaddrs.begin(); saddr != vsaddrs.end(); ++saddr) {
389         // 获取存储服务器连接池
390         pool_c* spool = (pool_c*)s_mngr->get(saddr->c_str());
391         if (!spool) {
392             s_mngr->set(saddr->c_str(), s_scount);
393             if (!(spool = (pool_c*)s_mngr->get(saddr->c_str()))) {
394                 logger_warn(
395                     "storage connection pool is null, saddr: %s",
396                     saddr->c_str());
397                 continue;
398             }
399         }
400 
401         for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
402             // 获取存储服务器连接
403             conn_c* sconn = (conn_c*)spool->peek();
404             if (!sconn) {
405                 logger_warn("storage connection is null, saddr: %s",
406                     saddr->c_str());
407                 break;
408             }
409 
410             // 向存储服务器询问文件大小
411             result = sconn->filesize(appid, userid, fileid, filesize);
412 
413             if (result == SOCKET_ERROR) {
414                 logger_warn("get filesize fail: %s", sconn->errdesc());
415                 spool->put(sconn, false);
416             }
417             else {
418                 if (result == OK)
419                     spool->put(sconn, true);
420                 else {
421                     logger_error("get filesize fail: %s",
422                         sconn->errdesc());
423                     spool->put(sconn, false);
424                 }
425                 return result;
426             }
427         }
428     }
429 
430     return result;
431 }
432 
433 // 从存储服务器下载文件
434 int client_c::download(char const* appid, char const* userid,
435     char const* fileid, long long offset, long long size,
436     char** filedata, long long* filesize) {
```

```
    // 检查参数
    if (!appid || !*appid) {
        logger_error("appid is null");
        return ERROR;
    }
    if (!userid || !*userid) {
        logger_error("userid is null");
        return ERROR;
    }
    if (!fileid || !*fileid) {
        logger_error("fileid is null");
        return ERROR;
    }

    // 从跟踪服务器获取存储服务器地址列表
    int result;
    std::string ssaddrs;
    if ((result = saddrs(appid, userid, fileid, ssaddrs)) != OK)
        return ERROR;
    std::vector<std::string> vsaddrs;
    split(ssaddrs.c_str(), vsaddrs);
    if (vsaddrs.empty()) {
        logger_error("storage addresses is empty");
        return ERROR;
    }

    result = ERROR;

    for (std::vector<std::string>::const_iterator saddr =
        vsaddrs.begin(); saddr != vsaddrs.end(); ++saddr) {
        // 获取存储服务器连接池
        pool_c* spool = (pool_c*)s_mngr->get(saddr->c_str());
        if (!spool) {
            s_mngr->set(saddr->c_str(), s_scount);
            if (!(spool = (pool_c*)s_mngr->get(saddr->c_str()))) {
                logger_warn(
                    "storage connection pool is null, saddr: %s",
                    saddr->c_str());
                continue;
            }
        }

        for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
            // 获取存储服务器连接
            conn_c* sconn = (conn_c*)spool->peek();
            if (!sconn) {
                logger_warn("storage connection is null, saddr: %s",
                    saddr->c_str());
                break;
            }

            // 从存储服务器下载文件
            result = sconn->download(
                appid, userid, fileid, offset, size, filedata, filesize);

            if (result == SOCKET_ERROR) {
```

```
                logger_warn("download file fail: %s", sconn->errdesc());
                spool->put(sconn, false);
            }
            else {
                if (result == OK)
                    spool->put(sconn, true);
                else {
                    logger_error("download file fail: %s",
                        sconn->errdesc());
                    spool->put(sconn, false);
                }
                return result;
            }
        }
    }

    return result;
}

// 删除存储服务器上的文件
int client_c::del(char const* appid, char const* userid,
    char const* fileid) {
    // 检查参数
    if (!appid || !*appid) {
        logger_error("appid is null");
        return ERROR;
    }
    if (!userid || !*userid) {
        logger_error("userid is null");
        return ERROR;
    }
    if (!fileid || !*fileid) {
        logger_error("fileid is null");
        return ERROR;
    }

    // 从跟踪服务器获取存储服务器地址列表
    int result;
    std::string ssaddrs;
    if ((result = saddrs(appid, userid, fileid, ssaddrs)) != OK)
        return ERROR;
    std::vector<std::string> vsaddrs;
    split(ssaddrs.c_str(), vsaddrs);
    if (vsaddrs.empty()) {
        logger_error("storage addresses is empty");
        return ERROR;
    }

    result = ERROR;

    for (std::vector<std::string>::const_iterator saddr =
        vsaddrs.begin(); saddr != vsaddrs.end(); ++saddr) {
        // 获取存储服务器连接池
        pool_c* spool = (pool_c*)s_mngr->get(saddr->c_str());
        if (!spool) {
            s_mngr->set(saddr->c_str(), s_scount);
```

```
            if (!(spool = (pool_c*)s_mngr->get(saddr->c_str()))) {
                logger_warn(
                    "storage connection pool is null, saddr: %s",
                    saddr->c_str());
                continue;
            }
        }

        for (int sockerrs = 0; sockerrs < MAX_SOCKERRS; ++sockerrs) {
            // 获取存储服务器连接
            conn_c* sconn = (conn_c*)spool->peek();
            if (!sconn) {
                logger_warn("storage connection is null, saddr: %s",
                    saddr->c_str());
                break;
            }

            // 删除存储服务器上的文件
            result = sconn->del(appid, userid, fileid);

            if (result == SOCKET_ERROR) {
                logger_warn("delete file fail: %s", sconn->errdesc());
                spool->put(sconn, false);
            }
            else {
                if (result == OK)
                    spool->put(sconn, true);
                else {
                    logger_error("delete file fail: %s",
                        sconn->errdesc());
                    spool->put(sconn, false);
                }
                return result;
            }
        }
    }

    return result;
}
```

~/TNV/src/05_client/09_main.cpp

```
// 客户机
// 定义主函数
//
#include <unistd.h>
#include <lib_acl.h>
#include <lib_acl.hpp>
#include "01_types.h"
#include "07_client.h"

// 打印命令行用法
void usage(char const* cmd) {
    fprintf(stderr, "Groups  : %s <taddrs> groups\n", cmd);
    fprintf(stderr, "Upload  : %s <taddrs> upload   "
```

```
        "<appid> <userid> <filepath>\n", cmd);
    fprintf(stderr, "Filesize: %s <taddrs> filesize "
        "<appid> <userid> <fileid>\n", cmd);
    fprintf(stderr, "Download: %s <taddrs> download "
        "<appid> <userid> <fileid> <offset> <size>\n", cmd);
    fprintf(stderr, "Delete  : %s <taddrs> delete   "
        "<appid> <userid> <fileid>\n", cmd);
}

// 根据用户ID生成文件ID
std::string genfileid(char const* userid) {
    srand(time(NULL));

    struct timeval now;
    gettimeofday(&now, NULL);

    acl::string str;
    str.format("%s@%d@%lX@%d",
        userid, getpid(), acl_pthread_self(), rand());

    acl::md5 md5;
    md5.update(str.c_str(), str.size());
    md5.finish();

    char buf[33] = {};
    strncpy(buf, md5.get_string(), 32);
    memmove(buf, buf + 8, 16);
    memset(buf + 16, 0, 16);

    static int count = 0;
    if (count >= 8000)
        count = 0;

    acl::string fileid;
    fileid.format("%08lx%06lx%s%04d%02d", now.tv_sec,
        now.tv_usec, buf, ++count, rand() % 100);

    return fileid.c_str();
}

int main(int argc, char* argv[]) {
    char const* cmd = argv[0];

    if (argc < 3) {
        usage(cmd);
        return -1;
    }

    char const* taddrs = argv[1];
    char const* subcmd = argv[2];

    // 初始化ACL库
    acl::acl_cpp_init();

    // 日志打印到标准输出
    acl::log::stdout_open(true);
```

```
70
71      // 初始化客户机
72      if (client_c::init(taddrs) != OK)
73          return -1;
74
75      client_c client; // 客户机对象
76
77      // 从跟踪服务器获取组列表
78      if (!strcmp(subcmd, "groups")) {
79          std::string groups;
80          if (client.groups(groups) != OK) {
81              client_c::deinit();
82              return -1;
83          }
84          printf("%s\n", groups.c_str());
85      }
86      else
87      // 向存储服务器上传文件
88      if (!strcmp(subcmd, "upload")) {
89          if (argc < 6) {
90              client_c::deinit();
91              usage(cmd);
92              return -1;
93          }
94          char const* appid    = argv[3];
95          char const* userid   = argv[4];
96          std::string fileid   = genfileid(userid);
97          char const* filepath = argv[5];
98          if (client.upload(appid, userid,
99              fileid.c_str(), filepath) != OK) {
100             client_c::deinit();
101             return -1;
102         }
103         printf("Upload success: %s\n", fileid.c_str());
104     }
105     else
106     // 向存储服务器询问文件大小
107     if (!strcmp(subcmd, "filesize")) {
108         if (argc < 6) {
109             client_c::deinit();
110             usage(cmd);
111             return -1;
112         }
113         char const* appid    = argv[3];
114         char const* userid   = argv[4];
115         char const* fileid   = argv[5];
116         long long   filesize = 0;
117         if (client.filesize(appid, userid, fileid, &filesize) != OK) {
118             client_c::deinit();
119             return -1;
120         }
121         printf("Get filesize success: %lld\n", filesize);
122     }
123     else
124     // 从存储服务器下载文件
125     if (!strcmp(subcmd, "download")) {
```

```
        if (argc < 8) {
            client_c::deinit();
            usage(cmd);
            return -1;
        }
        char const* appid    = argv[3];
        char const* userid   = argv[4];
        char const* fileid   = argv[5];
        long long   offset   = atoll(argv[6]);
        long long   size     = atoll(argv[7]);
        char*       filedata = NULL;
        long long   filesize = 0;
        if (client.download(appid, userid,
            fileid, offset, size, &filedata, &filesize) != OK) {
            client_c::deinit();
            return -1;
        }
        printf("Download success: %lld\n", filesize);
        free(filedata);
    }
    else
    // 删除存储服务器上的文件
    if (!strcmp(subcmd, "delete")) {
        if (argc < 6) {
            client_c::deinit();
            usage(cmd);
            return -1;
        }
        char const* appid  = argv[3];
        char const* userid = argv[4];
        char const* fileid = argv[5];
        if (client.del(appid, userid, fileid) != OK) {
            client_c::deinit();
            return -1;
        }
        printf("Delete success: %s\n", fileid);
    }
    else {
        client_c::deinit();
        usage(cmd);
        return -1;
    }

    // 终结化客户机
    client_c::deinit();

    return 0;
}
```

~/TNV/src/05_client/Makefile

```
PROJ   = ../../bin/client
LPRJ   = ../../lib/libclient.a
OBJS   = $(patsubst %.cpp, %.o, $(wildcard ../01_common/*.cpp *.cpp))
LOBJ   = $(filter-out %main.o, $(OBJS))
```

```
CC     = g++
LINK   = g++
AR     = ar rv
RM     = rm -rf
CFLAGS = -c -Wall \
         -I/usr/include/acl-lib/acl -I/usr/include/acl-lib/acl_cpp \
         -I../01_common
LIBS   = -pthread -lacl_all

all: $(PROJ) $(LPRJ)

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

$(LPRJ): $(LOBJ)
    $(AR) $@ $^

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

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