5 客户机

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

```cpp
// 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,
```

```
        appid, userid, fileid, requ) != OK)
        return ERROR;
    acl::ifstream ifs;
    if (!ifs.open_read(filepath)) {
        logger_error("open file fail, filepath: %s", filepath);
        return ERROR;
    }
    long long filesize = ifs.fsize();
    llton(filesize, requ + HEADLEN +
        APPID_SIZE + USERID_SIZE + FILEID_SIZE);
    bodylen += filesize;
    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;
    }

    // 发送文件
    long long remain = filesize; // 未发送字节数
    off_t offset = 0; // 文件读写位置
    while (remain) { // 还有未发送数据
        // 发送数据
        long long bytes = std::min(remain,
            (long long)STORAGE_RCVWR_SIZE);
        long long count = sendfile(m_conn->sock_handle(),
            ifs.file_handle(), &offset, bytes);
        if (count < 0) {
            logger_error(
                "send file fail, filesize: %lld, remain: %lld",
                filesize, remain);
            m_errnumb = -1;
            m_errdesc.format(
                "send file fail, filesize: %lld, remain: %lld",
                filesize, remain);
            close();
            return SOCKET_ERROR;
        }
        // 未发递减
        remain -= count;
    }
    ifs.close();

    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;
}

// 向存储服务器上传文件
int conn_c::upload(char const* appid, char const* userid,
    char const* fileid, char const* filedata, long long filesize) {
    // |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,
            appid, userid, fileid, requ) != OK)
        return ERROR;
    llton(filesize, requ + HEADLEN +
        APPID_SIZE + USERID_SIZE + FILEID_SIZE);
    bodylen += filesize;
    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;
    }

    // 发送文件
    if (m_conn->write(filedata, filesize) < 0) {
        logger_error("write fail: %s, filesize: %lld, to: %s",
            acl::last_serror(), filesize, m_conn->get_peer());
        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;
    }
```

```

    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|文件大小|
        // |    8   |  1 |  1 |    8   |
        *filesize = ntoll(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;
}

// 从存储服务器下载文件
int conn_c::download(char const* appid, char const* userid,
    char const* fileid, long long offset, long long size,
    char** filedata, long long* filesize) {
    // |包体长度|命令|状态|应用ID|用户ID|文件ID|偏移|大小|
    // |    8   |  1 |  1 |  16  |  256 |  128 |  8 |  8 |
    // 构造请求
    long long bodylen = APPID_SIZE + USERID_SIZE + FILEID_SIZE +
        BODYLEN_SIZE + BODYLEN_SIZE;
    long long requlen = HEADLEN + bodylen;
    char requ[requlen];
    if (makerequ(CMD_STORAGE_DOWNLOAD,
        appid, userid, fileid, requ) != OK)
        return ERROR;
    llton(bodylen, requ);
    llton(offset, requ + HEADLEN +
        APPID_SIZE + USERID_SIZE + FILEID_SIZE);
    llton(size, requ + HEADLEN +
        APPID_SIZE + USERID_SIZE + FILEID_SIZE + BODYLEN_SIZE);

    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|文件内容|
        // |    8    |  1 |  1 |内容大小|
        *filedata = body;
        *filesize = bodylen;
        return result;
    }
    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;
}

// 删除存储服务器上的文件
int conn_c::del(char const* appid, char const* userid,
    char const* fileid) {
    // |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_DELETE,
        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(), reqlen, m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("write fail: %s, reqlen: %lld, to: %s",
            acl::last_serror(), reqlen, 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;
}

// 获取错误号
short conn_c::errnumb(void) const {
    return m_errnumb;
}

// 获取错误描述
char const* conn_c::errdesc(void) const {
    return m_errdesc.c_str();
}

// 打开连接
bool conn_c::open(void) {
    if (m_conn)
        return true;

    // 创建连接对象
    m_conn = new acl::socket_stream;

    // 连接目的主机
    if (!m_conn->open(m_destaddr, m_ctimeout, m_rtimeout)) {
        logger_error("open %s fail: %s",
            m_destaddr, acl_last_serror());
        delete m_conn;
        m_conn = NULL;
        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);
```

```
    return OK;
}

// 接收包体
int conn_c::recvbody(char** body, long long* bodylen) {
    // 接收包头
    int result = recvhead(bodylen);

    // 既非本地错误亦非套接字通信错误且包体非空
    if (result != ERROR && result != SOCKET_ERROR && *bodylen) {
        // 分配包体
        if (!(*body = (char*)malloc(*bodylen))) {
            logger_error("call malloc fail: %s, bodylen: %lld",
                strerror(errno), *bodylen);
            m_errnumb = -1;
            m_errdesc.format("call malloc fail: %s, bodylen: %lld",
                strerror(errno), *bodylen);
            return ERROR;
        }

        // 接收包体
        if (m_conn->read(*body, *bodylen) < 0) {
            logger_error("read fail: %s, from: %s",
                acl::last_serror(), m_conn->get_peer());
            m_errnumb = -1;
            m_errdesc.format("read fail: %s, from: %s",
                acl::last_serror(), m_conn->get_peer());
            free(*body);
            *body = NULL;
            close();
            return SOCKET_ERROR;
        }
    }

    return result;
}

// 接收包头
int conn_c::recvhead(long long* bodylen) {
    if (!open())
        return SOCKET_ERROR;

    char head[HEADLEN]; // 包头缓冲区

    // 接收包头
    if (m_conn->read(head, HEADLEN) < 0) {
        logger_error("read fail: %s, from: %s",
            acl::last_serror(), m_conn->get_peer());
        m_errnumb = -1;
        m_errdesc.format("read fail: %s, from: %s",
            acl::last_serror(), m_conn->get_peer());
        close();
        return SOCKET_ERROR;
    }

```

```
    // |包体长度|命令|状态|
    // |    8   |  1 |  1 |
    // 解析包头
    if ((*bodylen = ntoll(head)) < 0) { // 包体长度
        logger_error("invalid body length: %lld < 0, 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

```
// 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

```
// 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

```
// 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

```
// 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;
    }
```

```cpp

    // 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, filepath);

            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()) {
```

```
        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;
            }

            // 向存储服务器询问文件大小
            result = sconn->filesize(appid, userid, fileid, filesize);

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

    return result;
}

// 从存储服务器下载文件
int client_c::download(char const* appid, char const* userid,
    char const* fileid, long long offset, long long size,
    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);
```

```

    // 初始化客户机
    if (client_c::init(taddrs) != OK)
        return -1;

    client_c client; // 客户机对象

    // 从跟踪服务器获取组列表
    if (!strcmp(subcmd, "groups")) {
        std::string groups;
        if (client.groups(groups) != OK) {
            client_c::deinit();
            return -1;
        }
        printf("%s\n", groups.c_str());
    }
    else
    // 向存储服务器上传文件
    if (!strcmp(subcmd, "upload")) {
        if (argc < 6) {
            client_c::deinit();
            usage(cmd);
            return -1;
        }
        char const* appid    = argv[3];
        char const* userid   = argv[4];
        std::string fileid   = genfileid(userid);
        char const* filepath = argv[5];
        if (client.upload(appid, userid,
            fileid.c_str(), filepath) != OK) {
            client_c::deinit();
            return -1;
        }
        printf("Upload success: %s\n", fileid.c_str());
    }
    else
    // 向存储服务器询问文件大小
    if (!strcmp(subcmd, "filesize")) {
        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];
        long long   filesize = 0;
        if (client.filesize(appid, userid, fileid, &filesize) != OK) {
            client_c::deinit();
            return -1;
        }
        printf("Get filesize success: %lld\n", filesize);
    }
    else
    // 从存储服务器下载文件
    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)
```