6 HTTP服务器

~/TNV/src/06_http/01_status.h

```
// HTTP服务器
// 响应状态码
//
#pragma once

#define STATUS_OK                  200
#define STATUS_BAD_REQUEST         400
#define STATUS_INTER_SERVER_ERROR  500
```

~/TNV/src/06_http/02_globals.h

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

#include <lib_acl.hpp>
//
// 配置信息
//
extern char* cfg_taddrs; // 跟踪服务器地址表
extern char* cfg_raddrs; // Redis地址表
extern acl::master_str_tbl cfg_str[]; // 字符串配置表

extern int cfg_maxthrds; // Redis最大线程
extern int cfg_ctimeout; // Redis连接超时
extern int cfg_rtimeout; // Redis读写超时
extern acl::master_int_tbl cfg_int[]; // 整型配置表

extern int cfg_rsession; // 是否使用Redis会话
extern acl::master_bool_tbl cfg_bool[]; // 布尔型配置表
```

~/TNV/src/06_http/03_globals.cpp

```
// HTTP服务器
// 定义全局变量
//
#include "02_globals.h"
//
// 配置信息
//
char* cfg_taddrs; // 跟踪服务器地址表
char* cfg_raddrs; // Redis地址表
acl::master_str_tbl cfg_str[] = { // 字符串配置表
    {"tnv_tracker_addrs", "127.0.0.1:21000", &cfg_taddrs},
    {"redis_addrs",       "127.0.0.1:6379",  &cfg_raddrs},
    {0, 0, 0}};
```

```

int cfg_maxthrds; // Redis最大线程
int cfg_ctimeout; // Redis连接超时
int cfg_rtimeout; // Redis读写超时
acl::master_int_tbl cfg_int[] = { // 整型配置表
    {"ioctl_max_threads",  250, &cfg_maxthrds, 0, 0},
    {"redis_conn_timeout",  10, &cfg_ctimeout, 0, 0},
    {"redis_rw_timeout",    10, &cfg_rtimeout, 0, 0},
    {0, 0, 0, 0, 0}};

int cfg_rsession; // 是否使用Redis会话
acl::master_bool_tbl cfg_bool[] = { // 布尔型配置表
    {"use_redis_session", 1, &cfg_rsession},
    {0, 0, 0}};
```

~/TNV/src/06_http/04_service.h

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

#include <lib_acl.hpp>
//
// 业务服务类
//
class service_c: public acl::HttpServlet {
public:
    service_c(acl::socket_stream* conn, acl::session* sess);

protected:
    bool doGet(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res);
    bool doPost(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res);
    bool doOptions(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res);
    bool doError(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res);
    bool doOther(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res, char const* method);

private:
    // 处理文件路由
    bool files(acl::HttpServletRequest& req,
        acl::HttpServletResponse& res);
};
```

~/TNV/src/06_http/05_service.cpp

```
// HTTP服务器
// 实现业务服务类
//
#include "01_types.h"
```

```cpp
#include "07_client.h"
#include "01_status.h"
#include "02_globals.h"
#include "04_service.h"

#define ROUTE_FILES "/files/"
#define APPID       "tnvideo"
#define USERID      "tnv001"
#define FILE_SLICE  1024*1024*8LL

service_c::service_c(acl::socket_stream* conn, acl::session* sess):
    HttpServlet(conn, sess) {
}

bool service_c::doGet(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res) {
    // 从请求中提取资源路径
    acl::string path = req.getPathInfo();

    if (!path.ncompare(ROUTE_FILES, strlen(ROUTE_FILES)))
        // 处理文件路由
        files(req, res);
    else {
        logger_error("unknown route, path: %s", path.c_str());
        res.setStatus(STATUS_BAD_REQUEST);
    }

    // 发送响应
    return res.write(NULL, 0);
}

bool service_c::doPost(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res) {
    // 发送响应
    return res.write(NULL, 0);
}

bool service_c::doOptions(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res) {
    res.setStatus(STATUS_OK)
       .setContentType("text/plain;charset=utf-8")
       .setContentLength(0)
       .setKeepAlive(req.isKeepAlive());

    // 发送响应
    return res.write(NULL, 0);
}

bool service_c::doError(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res) {
    // 发送响应头
    res.setStatus(STATUS_BAD_REQUEST)
       .setContentType("text/html;charset=");
    if (!res.sendHeader())
        return false;

```

```
    // 发送响应体
    acl::string body;
    body.format("<root error='some error happened' />\r\n");
    return res.getOutputStream().write(body);
}

bool service_c::doOther(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res, char const* method) {
    // 发送响应头
    res.setStatus(STATUS_BAD_REQUEST)
       .setContentType("text/html;charset=");
    if (!res.sendHeader())
        return false;

    // 发送响应体
    acl::string body;
    body.format("<root error='unknown request method %s' />\r\n",
        method);
    return res.getOutputStream().write(body);
}

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

// 处理文件路由
bool service_c::files(acl::HttpServletRequest& req,
    acl::HttpServletResponse& res) {
    // 以与请求相同的连接模式回复响应
    res.setKeepAlive(req.isKeepAlive());

    // 从请求的资源路径中提取文件ID并检查之
    acl::string path = req.getPathInfo();
    acl::string fileid = path.right(strlen(ROUTE_FILES) - 1);
    if (!fileid.c_str() || !fileid.size()) {
        logger_error("fileid is null");
        res.setStatus(STATUS_BAD_REQUEST);
        return false;
    }
    logger("fileid: %s", fileid.c_str());

    // 设置响应中的内容字段
    res.setContentType("application/octet-stream");
    acl::string filename;
    filename.format("attachment;filename=%s", fileid.c_str());
    res.setHeader("content-disposition", filename.c_str());

    client_c client; // 客户机对象

    // 向存储服务器询问文件大小
    long long filesize = 0;
    if (client.filesize(APPID, USERID, fileid, &filesize) != OK) {
        res.setStatus(STATUS_INTER_SERVER_ERROR);
        return false;
    }
    logger("filesize: %lld", filesize);

    // 从请求的头部信息中提取范围信息
```

```
    long long range_from, range_to;
    if (req.getRange(range_from, range_to)) {
        if (range_to == -1)
            range_to = filesize;
    }
    else {
        range_from = 0;
        range_to = filesize;
    }
    logger("range: %lld-%lld", range_from, range_to);

    // 从存储服务器下载文件并发送响应
    long long remain   = range_to - range_from;        // 未下载字节数
    long long offset   = range_from;                   // 文件偏移位置
    long long size     = std::min(remain, FILE_SLICE); // 期望下载大小
    char*     downdata = NULL;                         // 下载数据缓冲
    long long downsize = 0;                            // 实际下载大小
    while (remain) { // 还有未下载数据
        // 下载数据
        if (client.download(APPID, USERID, fileid.c_str(),
            offset, size, &downdata, &downsize) != OK) {
            res.setStatus(STATUS_INTER_SERVER_ERROR);
            return false;
        }
        // 发送响应
        res.write(downdata, downsize);
        // 继续下载
        remain -= downsize;
        offset += downsize;
        size    = std::min(remain, FILE_SLICE);
        free(downdata);
    }

    return true;
}
```

~/TNV/src/06_http/06_server.h

```
// HTTP服务器
// 声明服务器类
//
#pragma once

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

```
    bool proc_exit_timer(size_t nclients, size_t nthreads);
    // 进程退出前被调用
    void proc_on_exit(void);

    // 线程获得连接时被调用
    // 返回true，连接将被用于后续通信，否则
    // 函数返回后即关闭连接
    bool thread_on_accept(acl::socket_stream* conn);
    // 与线程绑定的连接可读时被调用
    // 返回true，保持长连接，否则
    // 函数返回后即关闭连接
    bool thread_on_read(acl::socket_stream* conn);
    // 线程读写连接超时时被调用
    // 返回true，继续等待下一次读写，否则
    // 函数返回后即关闭连接
    bool thread_on_timeout(acl::socket_stream* conn);
    // 与线程绑定的连接关闭时被调用
    void thread_on_close(acl::socket_stream* conn);

private:
    acl::redis_client_cluster* m_redis; // Redis集群
};
```

~/TNV/src/06_http/07_server.cpp

```
// HTTP服务器
// 实现服务器类
//
#include "01_types.h"
#include "07_client.h"
#include "02_globals.h"
#include "04_service.h"
#include "06_server.h"

// 进程切换用户后被调用
void server_c::proc_on_init(void) {
    // 初始化客户机
    client_c::init(cfg_taddrs);

    // 创建并初始化Redis集群
    m_redis = new acl::redis_client_cluster;
    m_redis->init(NULL, cfg_raddrs, cfg_maxthrds,
        cfg_ctimeout, cfg_rtimeout);

    // 打印配置信息
    logger("cfg_taddrs: %s, cfg_raddrs: %s, cfg_maxthrds: %d, "
        "cfg_ctimeout: %d, cfg_rtimeout: %d, cfg_rsession: %d",
        cfg_taddrs, cfg_raddrs, cfg_maxthrds,
        cfg_ctimeout, cfg_rtimeout, cfg_rsession);
}

// 子进程意图退出时被调用
// 返回true，子进程立即退出，否则
// 若配置项ioctl_quick_abort非0，子进程立即退出，否则
// 待所有客户机连接都关闭后，子进程再退出
```

```
bool server_c::proc_exit_timer(size_t nclients, size_t nthreads) {
    if (!nclients || !nthreads) {
        logger("nclients: %lu, nthreads: %lu", nclients, nthreads);
        return true;
    }

    return false;
}

// 进程退出前被调用
void server_c::proc_on_exit(void) {
    delete m_redis;

    // 终结化客户机
    client_c::deinit();
}

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

    // 设置读写超时
    conn->set_rw_timeout(cfg_rtimeout);

    // 创建会话
    acl::session* session = cfg_rsession ?
        (acl::session*)new acl::redis_session(*m_redis, cfg_maxthrds) :
        (acl::session*)new acl::memcache_session("127.0.0.1:11211");

    // 创建并设置业务服务对象
    conn->set_ctx(new service_c(conn, session));

    return true;
}

// 与线程绑定的连接可读时被调用
// 返回true，保持长连接，否则
// 函数返回后即关闭连接
bool server_c::thread_on_read(acl::socket_stream* conn) {
    service_c* service = (service_c*)conn->get_ctx();
    if (!service)
        logger_fatal("service is null");

    return service->doRun();
}

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

```

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

    service_c* service = (service_c*)conn->get_ctx();
    acl::session* session = &service->getSession();
    delete session;
    delete service;
}
```

~/TNV/src/06_http/08_main.cpp

```cpp
// HTTP服务器
// 定义主函数
//
#include "02_globals.h"
#include "06_server.h"

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

    // 创建并运行服务器
    server_c& server = acl::singleton2<server_c>::get_instance();
    server.set_cfg_str(cfg_str);
    server.set_cfg_int(cfg_int);
    server.set_cfg_bool(cfg_bool);
    server.run_alone("127.0.0.1:8080", "../etc/http.cfg");

    return 0;
}
```

~/TNV/src/06_http/Makefile

```
PROJ   = ../../bin/http
OBJS   = $(patsubst %.cpp, %.o, $(wildcard ../01_common/*.cpp *.cpp))
CC     = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -Wall \
         -I/usr/include/acl-lib/acl_cpp \
         -I../01_common -I../05_client
LIBS   = -pthread -L../../lib -lclient -lacl_all -lz

all: $(PROJ)

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

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

clean:
```

```
	$(RM) $(PROJ) $(OBJS)
```

~/TNV/etc/http.cfg

```
service http {
    # 跟踪服务器地址表
    tnv_tracker_addrs = 127.0.0.1:21000
    # Redis地址表
    redis_addrs = 127.0.0.1:6379
    # Redis最大线程
    ioctl_max_threads = 250
    # Redis连接超时
    redis_conn_timeout = 10
    # Redis读写超时
    redis_rw_timeout = 10
    # 是否使用Redis会话
    use_redis_session = 1
}
```