

6 HTTP服务器

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

```
1 // HTTP服务器
2 // 响应状态码
3 //
4 #pragma once
5
6 #define STATUS_OK 200
7 #define STATUS_BAD_REQUEST 400
8 #define STATUS_INTER_SERVER_ERROR 500
```

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

```
1 // HTTP服务器
2 // 声明全局变量
3 //
4 #pragma once
5
6 #include <lib_acl.hpp>
7 //
8 // 配置信息
9 //
10 extern char* cfg_taddrs; // 跟踪服务器地址表
11 extern char* cfg_raddrs; // Redis地址表
12 extern acl::master_str_tbl cfg_str[]; // 字符串配置表
13
14 extern int cfg_maxthrs; // Redis最大线程
15 extern int cfg_ctimeout; // Redis连接超时
16 extern int cfg_rtimeout; // Redis读写超时
17 extern acl::master_int_tbl cfg_int[]; // 整型配置表
18
19 extern int cfg_rsession; // 是否使用Redis会话
20 extern acl::master_bool_tbl cfg_bool[]; // 布尔型配置表
```

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

```
1 // HTTP服务器
2 // 定义全局变量
3 //
4 #include "02_globals.h"
5 //
6 // 配置信息
7 //
8 char* cfg_taddrs; // 跟踪服务器地址表
9 char* cfg_raddrs; // Redis地址表
10 acl::master_str_tbl cfg_str[] = { // 字符串配置表
11     {"tnv_tracker_addrs", "127.0.0.1:21000", &cfg_taddrs},
12     {"redis_addrs", "127.0.0.1:6379", &cfg_raddrs},
13     {0, 0, 0}};
```

```

14
15 int cfg_maxthrs; // Redis最大线程
16 int cfg_ctimeout; // Redis连接超时
17 int cfg_rtimeout; // Redis读写超时
18 acl::master_int_tbl cfg_int[] = { // 整型配置表
19     {"ioctl_max_threads", 250, &cfg_maxthrs, 0, 0},
20     {"redis_conn_timeout", 10, &cfg_ctimeout, 0, 0},
21     {"redis_rw_timeout", 10, &cfg_rtimeout, 0, 0},
22     {0, 0, 0, 0, 0}};
23
24 int cfg_rsession; // 是否使用Redis会话
25 acl::master_bool_tbl cfg_bool[] = { // 布尔型配置表
26     {"use_redis_session", 1, &cfg_rsession},
27     {0, 0, 0}};

```

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

```

1 // HTTP服务器
2 // 声明业务服务类
3 //
4 #pragma once
5
6 #include <lib_acl.hpp>
7 //
8 // 业务服务类
9 //
10 class service_c: public acl::HttpServlet {
11 public:
12     service_c(acl::socket_stream* conn, acl::session* sess);
13
14 protected:
15     bool doGet(acl::HttpServletRequest& req,
16               acl::HttpServletResponse& res);
17     bool doPost(acl::HttpServletRequest& req,
18               acl::HttpServletResponse& res);
19     bool doOptions(acl::HttpServletRequest& req,
20                  acl::HttpServletResponse& res);
21     bool doError(acl::HttpServletRequest& req,
22                 acl::HttpServletResponse& res);
23     bool doOther(acl::HttpServletRequest& req,
24                 acl::HttpServletResponse& res, char const* method);
25
26 private:
27     // 处理文件路由
28     bool files(acl::HttpServletRequest& req,
29               acl::HttpServletResponse& res);
30 };

```

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

```

1 // HTTP服务器
2 // 实现业务服务类
3 //
4 #include "01_types.h"

```

```

5 #include "07_client.h"
6 #include "01_status.h"
7 #include "02_globals.h"
8 #include "04_service.h"
9
10 #define ROUTE_FILES "/files/"
11 #define APPID      "tnvideo"
12 #define USERID     "tnv001"
13 #define FILE_SLICE 1024*1024*8LL
14
15 service_c::service_c(ac1::socket_stream* conn, ac1::session* sess):
16     HttpServlet(conn, sess) {
17 }
18
19 bool service_c::doGet(ac1::HttpServletRequest& req,
20     ac1::HttpServletResponse& res) {
21     // 从请求中提取资源路径
22     ac1::string path = req.getPathInfo();
23
24     if (!path.ncompare(ROUTE_FILES, strlen(ROUTE_FILES)))
25         // 处理文件路由
26         files(req, res);
27     else {
28         logger_error("unknown route, path: %s", path.c_str());
29         res.setStatus(STATUS_BAD_REQUEST);
30     }
31
32     // 发送响应
33     return res.write(NULL, 0);
34 }
35
36 bool service_c::doPost(ac1::HttpServletRequest& req,
37     ac1::HttpServletResponse& res) {
38     // 发送响应
39     return res.write(NULL, 0);
40 }
41
42 bool service_c::doOptions(ac1::HttpServletRequest& req,
43     ac1::HttpServletResponse& res) {
44     res.setStatus(STATUS_OK)
45         .setContentType("text/plain;charset=utf-8")
46         .setContentLength(0)
47         .setKeepAlive(req.isKeepAlive());
48
49     // 发送响应
50     return res.write(NULL, 0);
51 }
52
53 bool service_c::doError(ac1::HttpServletRequest& req,
54     ac1::HttpServletResponse& res) {
55     // 发送响应头
56     res.setStatus(STATUS_BAD_REQUEST)
57         .setContentType("text/html;charset=");
58     if (!res.sendHeader())
59         return false;
60

```

```

61 // 发送响应体
62 acl::string body;
63 body.format("<root error='some error happened' />\r\n");
64 return res.getOutputStream().write(body);
65 }
66
67 bool service_c::doOther(acl::HttpServletRequest& req,
68 acl::HttpServletResponse& res, char const* method) {
69 // 发送响应头
70 res.setStatus(STATUS_BAD_REQUEST)
71 .setContentType("text/html;charset=");
72 if (!res.sendHeader())
73     return false;
74
75 // 发送响应体
76 acl::string body;
77 body.format("<root error='unknown request method %s' />\r\n",
78     method);
79 return res.getOutputStream().write(body);
80 }
81
82 ///////////////////////////////////////////////////
83
84 // 处理文件路由
85 bool service_c::files(acl::HttpServletRequest& req,
86 acl::HttpServletResponse& res) {
87 // 以与请求相同的连接模式回复响应
88 res.setKeepAlive(req.isKeepAlive());
89
90 // 从请求的资源路径中提取文件ID并检查之
91 acl::string path = req.getPathInfo();
92 acl::string fileid = path.right(strlen(ROUTE_FILES) - 1);
93 if (!fileid.c_str() || !fileid.size()) {
94     logger_error("fileid is null");
95     res.setStatus(STATUS_BAD_REQUEST);
96     return false;
97 }
98 logger("fileid: %s", fileid.c_str());
99
100 // 设置响应中的内容字段
101 res.setContentType("application/octet-stream");
102 acl::string filename;
103 filename.format("attachment;filename=%s", fileid.c_str());
104 res.setHeader("content-disposition", filename.c_str());
105
106 client_c client; // 客户机对象
107
108 // 向存储服务询问文件大小
109 long long filesize = 0;
110 if (client.filesize(APPID, USERID, fileid, &filesize) != OK) {
111     res.setStatus(STATUS_INTER_SERVER_ERROR);
112     return false;
113 }
114 logger("filesize: %lld", filesize);
115
116 // 从请求的头部信息中提取范围信息

```

```

117     long long range_from, range_to;
118     if (req.getRange(range_from, range_to)) {
119         if (range_to == -1)
120             range_to = filesize;
121     }
122     else {
123         range_from = 0;
124         range_to = filesize;
125     }
126     logger("range: %lld-%lld", range_from, range_to);
127
128     // 从存储服务器下载文件并发送响应
129     long long remain    = range_to - range_from;           // 未下载字节数
130     long long offset    = range_from;                     // 文件偏移位置
131     long long size      = std::min(remain, FILE_SLICE);    // 期望下载大小
132     char*      downdata = NULL;                           // 下载数据缓冲
133     long long downsize = 0;                                // 实际下载大小
134     while (remain) { // 还有未下载数据
135         // 下载数据
136         if (client.download(APPID, USERID, fileid.c_str(),
137             offset, size, &downdata, &downsize) != OK) {
138             res.setStatus(STATUS_INTER_SERVER_ERROR);
139             return false;
140         }
141         // 发送响应
142         res.write(downdata, downsize);
143         // 继续下载
144         remain -= downsize;
145         offset += downsize;
146         size    = std::min(remain, FILE_SLICE);
147         free(downdata);
148     }
149
150     return true;
151 }

```

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

```

1  // HTTP服务器
2  // 声明服务器类
3  //
4  #pragma once
5
6  #include <lib_acl.hpp>
7  //
8  // 服务器类
9  //
10 class server_c: public acl::master_threads {
11 protected:
12     // 进程切换用户后被调用
13     void proc_on_init(void);
14     // 子进程意图退出时被调用
15     // 返回true, 子进程立即退出, 否则
16     // 若配置项ioctl_quick_abort非0, 子进程立即退出, 否则
17     // 待所有客户机连接都关闭后, 子进程再退出

```

```

18     bool proc_exit_timer(size_t nclients, size_t nthreads);
19     // 进程退出前被调用
20     void proc_on_exit(void);
21
22     // 线程获得连接时被调用
23     // 返回true, 连接将被用于后续通信, 否则
24     // 函数返回后即关闭连接
25     bool thread_on_accept(acl::socket_stream* conn);
26     // 与线程绑定的连接可读时被调用
27     // 返回true, 保持长连接, 否则
28     // 函数返回后即关闭连接
29     bool thread_on_read(acl::socket_stream* conn);
30     // 线程读写连接超时时被调用
31     // 返回true, 继续等待下一次读写, 否则
32     // 函数返回后即关闭连接
33     bool thread_on_timeout(acl::socket_stream* conn);
34     // 与线程绑定的连接关闭时被调用
35     void thread_on_close(acl::socket_stream* conn);
36
37 private:
38     acl::redis_client_cluster* m_redis; // Redis集群
39 };

```

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

```

1  // HTTP服务器
2  // 实现服务器类
3  //
4  #include "01_types.h"
5  #include "07_client.h"
6  #include "02_globals.h"
7  #include "04_service.h"
8  #include "06_server.h"
9
10 // 进程切换用户后被调用
11 void server_c::proc_on_init(void) {
12     // 初始化客户机
13     client_c::init(cfg_taddrs);
14
15     // 创建并初始化Redis集群
16     m_redis = new acl::redis_client_cluster;
17     m_redis->init(NULL, cfg_raddrs, cfg_maxthrs,
18         cfg_ctimeout, cfg_rtimeout);
19
20     // 打印配置信息
21     logger("cfg_taddrs: %s, cfg_raddrs: %s, cfg_maxthrs: %d, "
22         "cfg_ctimeout: %d, cfg_rtimeout: %d, cfg_rsession: %d",
23         cfg_taddrs, cfg_raddrs, cfg_maxthrs,
24         cfg_ctimeout, cfg_rtimeout, cfg_rsession);
25 }
26
27 // 子进程意图退出时被调用
28 // 返回true, 子进程立即退出, 否则
29 // 若配置项ioctl_quick_abort非0, 子进程立即退出, 否则
30 // 待所有客户机连接都关闭后, 子进程再退出

```

```

31 bool server_c::proc_exit_timer(size_t nclients, size_t nthreads) {
32     if (!nclients || !nthreads) {
33         logger("nclients: %lu, nthreads: %lu", nclients, nthreads);
34         return true;
35     }
36
37     return false;
38 }
39
40 // 进程退出前被调用
41 void server_c::proc_on_exit(void) {
42     delete m_redis;
43
44     // 终结化客户机
45     client_c::deinit();
46 }
47
48 // 线程获得连接时被调用
49 // 返回true, 连接将被用于后续通信, 否则
50 // 函数返回后即关闭连接
51 bool server_c::thread_on_accept(acl::socket_stream* conn) {
52     logger("connect, from: %s", conn->get_peer());
53
54     // 设置读写超时
55     conn->set_rw_timeout(cfg_rtimeout);
56
57     // 创建会话
58     acl::session* session = cfg_rsession ?
59         (acl::session*)new acl::redis_session(*m_redis, cfg_maxthrs) :
60         (acl::session*)new acl::memcache_session("127.0.0.1:11211");
61
62     // 创建并设置业务服务对象
63     conn->set_ctx(new service_c(conn, session));
64
65     return true;
66 }
67
68 // 与线程绑定的连接可读时被调用
69 // 返回true, 保持长连接, 否则
70 // 函数返回后即关闭连接
71 bool server_c::thread_on_read(acl::socket_stream* conn) {
72     service_c* service = (service_c*)conn->get_ctx();
73     if (!service)
74         logger_fatal("service is null");
75
76     return service->doRun();
77 }
78
79 // 线程读写连接超时时被调用
80 // 返回true, 继续等待下一次读写, 否则
81 // 函数返回后即关闭连接
82 bool server_c::thread_on_timeout(acl::socket_stream* conn) {
83     logger("read timeout, from: %s", conn->get_peer());
84     return false;
85 }
86

```

```

87 // 与线程绑定的连接关闭时被调用
88 void server_c::thread_on_close(acl::socket_stream* conn) {
89     logger("client disconnect, from: %s", conn->get_peer());
90
91     service_c* service = (service_c*)conn->get_ctx();
92     acl::session* session = &service->getSession();
93     delete session;
94     delete service;
95 }

```

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

```

1 // HTTP服务器
2 // 定义主函数
3 //
4 #include "02_globals.h"
5 #include "06_server.h"
6
7 int main(void) {
8     // 初始化ACL库
9     acl::acl_cpp_init();
10    acl::log::stdout_open(true);
11
12    // 创建并运行服务器
13    server_c& server = acl::singleton2<server_c>::get_instance();
14    server.set_cfg_str(cfg_str);
15    server.set_cfg_int(cfg_int);
16    server.set_cfg_bool(cfg_bool);
17    server.run_alone("127.0.0.1:8080", "../etc/http.cfg");
18
19    return 0;
20 }

```

~/TNV/src/06_http/Makefile

```

1 PROJ    = ../../bin/http
2 OBJS    = $(patsubst %.cpp, %.o, $(wildcard ../01_common/*.cpp *.cpp))
3 CC      = g++
4 LINK    = g++
5 RM      = rm -rf
6 CFLAGS  = -c -Wall \
7           -I/usr/include/acl-lib/acl_cpp \
8           -I../01_common -I../05_client
9 LIBS    = -pthread -L../lib -lclient -lacl_all -lz
10
11 all: $(PROJ)
12
13 $(PROJ): $(OBJS)
14     $(LINK) $^ $(LIBS) -o $@
15
16 .cpp.o:
17     $(CC) $(CFLAGS) $^ -o $@
18
19 clean:

```


~/TNV/etc/http.cfg

```
1 service http {
2     # 跟踪服务器地址表
3     tnv_tracker_addrs = 127.0.0.1:21000
4     # Redis地址表
5     redis_addrs = 127.0.0.1:6379
6     # Redis最大线程
7     ioctl_max_threads = 250
8     # Redis连接超时
9     redis_conn_timeout = 10
10    # Redis读写超时
11    redis_rw_timeout = 10
12    # 是否使用Redis会话
13    use_redis_session = 1
14 }
```