buffer\buffer.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-26

3 \* @copyleft Apache 2.0

4 \*/

5 #include "buffer.h"

6

7 Buffer::Buffer(int initBuffSize) : buffer\_(initBuffSize), readPos\_(0), writePos\_(0) {}

8

9 size\_t Buffer::ReadableBytes() const {

10 return writePos\_ - readPos\_;

11 }

12 size\_t Buffer::WritableBytes() const {

13 return buffer\_.size() - writePos\_;

14 }

15

16 size\_t Buffer::PrependableBytes() const {

17 return readPos\_;

18 }

19

20 const char\* Buffer::Peek() const {

21 return BeginPtr\_() + readPos\_;

22 }

23

24 void Buffer::Retrieve(size\_t len) {

25 assert(len <= ReadableBytes());

26 readPos\_ += len;

27 }

28

29 void Buffer::RetrieveUntil(const char\* end) {

30 assert(Peek() <= end );

31 Retrieve(end - Peek());

32 }

33

34 void Buffer::RetrieveAll() {

35 bzero(&buffer\_[0], buffer\_.size());

36 readPos\_ = 0;

37 writePos\_ = 0;

38 }

39

40 std::string Buffer::RetrieveAllToStr() {

41 std::string str(Peek(), ReadableBytes());

42 RetrieveAll();

43 return str;

44 }

45

46 const char\* Buffer::BeginWriteConst() const {

47 return BeginPtr\_() + writePos\_;

48 }

49

50 char\* Buffer::BeginWrite() {

51 return BeginPtr\_() + writePos\_;

52 }

53

54 void Buffer::HasWritten(size\_t len) {

55 writePos\_ += len;

56 }

57

58 void Buffer::Append(const std::string& str) {

59 Append(str.data(), str.length());

60 }

61

62 void Buffer::Append(const void\* data, size\_t len) {

63 assert(data);

64 Append(static\_cast<const char\*>(data), len);

65 }

66

67 void Buffer::Append(const char\* str, size\_t len) {

68 assert(str);

69 EnsureWriteable(len);

70 std::copy(str, str + len, BeginWrite());

71 HasWritten(len);

72 }

73

74 void Buffer::Append(const Buffer& buff) {

75 Append(buff.Peek(), buff.ReadableBytes());

76 }

77

78 void Buffer::EnsureWriteable(size\_t len) {

79 if(WritableBytes() < len) {

80 MakeSpace\_(len);

81 }

82 assert(WritableBytes() >= len);

83 }

84

85 ssize\_t Buffer::ReadFd(int fd, int\* saveErrno) {

86 char buff[65535];

87 struct iovec iov[2];

88 const size\_t writable = WritableBytes();

89 /\* 分散读， 保证数据全部读完 \*/

90 iov[0].iov\_base = BeginPtr\_() + writePos\_;

91 iov[0].iov\_len = writable;

92 iov[1].iov\_base = buff;

93 iov[1].iov\_len = sizeof(buff);

94

95 const ssize\_t len = readv(fd, iov, 2);

96 if(len < 0) {

97 \*saveErrno = errno;

98 }

99 else if(static\_cast<size\_t>(len) <= writable) {

100 writePos\_ += len;

101 }

102 else {

103 writePos\_ = buffer\_.size();

104 Append(buff, len - writable);

105 }

106 return len;

107 }

108

109 ssize\_t Buffer::WriteFd(int fd, int\* saveErrno) {

110 size\_t readSize = ReadableBytes();

111 ssize\_t len = write(fd, Peek(), readSize);

112 if(len < 0) {

113 \*saveErrno = errno;

114 return len;

115 }

116 readPos\_ += len;

117 return len;

118 }

119

120 char\* Buffer::BeginPtr\_() {

121 return &\*buffer\_.begin();

122 }

123

124 const char\* Buffer::BeginPtr\_() const {

125 return &\*buffer\_.begin();

126 }

127

128 void Buffer::MakeSpace\_(size\_t len) {

129 if(WritableBytes() + PrependableBytes() < len) {

130 buffer\_.resize(writePos\_ + len + 1);

131 }

132 else {

133 size\_t readable = ReadableBytes();

134 std::copy(BeginPtr\_() + readPos\_, BeginPtr\_() + writePos\_, BeginPtr\_());

135 readPos\_ = 0;

136 writePos\_ = readPos\_ + readable;

137 assert(readable == ReadableBytes());

138 }

139 }

buffer\buffer.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-26

3 \* @copyleft Apache 2.0

4 \*/

5

6 #ifndef BUFFER\_H

7 #define BUFFER\_H

8 #include <cstring> //perror

9 #include <iostream>

10 #include <unistd.h> // write

11 #include <sys/uio.h> //readv

12 #include <vector> //readv

13 #include <atomic>

14 #include <assert.h>

15 class Buffer {

16 public:

17 Buffer(int initBuffSize = 1024);

18 ~Buffer() = default;

19

20 size\_t WritableBytes() const;

21 size\_t ReadableBytes() const ;

22 size\_t PrependableBytes() const;

23

24 const char\* Peek() const;

25 void EnsureWriteable(size\_t len);

26 void HasWritten(size\_t len);

27

28 void Retrieve(size\_t len);

29 void RetrieveUntil(const char\* end);

30

31 void RetrieveAll() ;

32 std::string RetrieveAllToStr();

33

34 const char\* BeginWriteConst() const;

35 char\* BeginWrite();

36

37 void Append(const std::string& str);

38 void Append(const char\* str, size\_t len);

39 void Append(const void\* data, size\_t len);

40 void Append(const Buffer& buff);

41

42 ssize\_t ReadFd(int fd, int\* Errno);

43 ssize\_t WriteFd(int fd, int\* Errno);

44

45 private:

46 char\* BeginPtr\_();

47 const char\* BeginPtr\_() const;

48 void MakeSpace\_(size\_t len);

49

50 std::vector<char> buffer\_;

51 std::atomic<std::size\_t> readPos\_;

52 std::atomic<std::size\_t> writePos\_;

53 };

54

55 #endif //BUFFER\_H

config\config.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-28

3 \* @copyleft Apache 2.0

4 \*/

http\httpconn.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-15

3 \* @copyleft Apache 2.0

4 \*/

5 #include "httpconn.h"

6 using namespace std;

7

8 const char\* HttpConn::srcDir;

9 std::atomic<int> HttpConn::userCount;

10 bool HttpConn::isET;

11

12 HttpConn::HttpConn() {

13 fd\_ = -1;

14 addr\_ = { 0 };

15 isClose\_ = true;

16 };

17

18 HttpConn::~HttpConn() {

19 Close();

20 };

21

22 void HttpConn::init(int fd, const sockaddr\_in& addr) {

23 assert(fd > 0);

24 userCount++;

25 addr\_ = addr;

26 fd\_ = fd;

27 writeBuff\_.RetrieveAll();

28 readBuff\_.RetrieveAll();

29 isClose\_ = false;

30 LOG\_INFO("Client[%d](%s:%d) in, userCount:%d", fd\_, GetIP(), GetPort(), (int)userCount);

31 }

32

33 void HttpConn::Close() {

34 response\_.UnmapFile();

35 if(isClose\_ == false){

36 isClose\_ = true;

37 userCount--;

38 close(fd\_);

39 LOG\_INFO("Client[%d](%s:%d) quit, UserCount:%d", fd\_, GetIP(), GetPort(), (int)userCount);

40 }

41 }

42

43 int HttpConn::GetFd() const {

44 return fd\_;

45 };

46

47 struct sockaddr\_in HttpConn::GetAddr() const {

48 return addr\_;

49 }

50

51 const char\* HttpConn::GetIP() const {

52 return inet\_ntoa(addr\_.sin\_addr);

53 }

54

55 int HttpConn::GetPort() const {

56 return addr\_.sin\_port;

57 }

58

59 ssize\_t HttpConn::read(int\* saveErrno) {

60 ssize\_t len = -1;

61 do {

62 len = readBuff\_.ReadFd(fd\_, saveErrno);

63 if (len <= 0) {

64 break;

65 }

66 } while (isET);

67 return len;

68 }

69

70 ssize\_t HttpConn::write(int\* saveErrno) {

71 ssize\_t len = -1;

72 do {

73 len = writev(fd\_, iov\_, iovCnt\_);

74 if(len <= 0) {

75 \*saveErrno = errno;

76 break;

77 }

78 if(iov\_[0].iov\_len + iov\_[1].iov\_len == 0) { break; } /\* 传输结束 \*/

79 else if(static\_cast<size\_t>(len) > iov\_[0].iov\_len) {

80 iov\_[1].iov\_base = (uint8\_t\*) iov\_[1].iov\_base + (len - iov\_[0].iov\_len);

81 iov\_[1].iov\_len -= (len - iov\_[0].iov\_len);

82 if(iov\_[0].iov\_len) {

83 writeBuff\_.RetrieveAll();

84 iov\_[0].iov\_len = 0;

85 }

86 }

87 else {

88 iov\_[0].iov\_base = (uint8\_t\*)iov\_[0].iov\_base + len;

89 iov\_[0].iov\_len -= len;

90 writeBuff\_.Retrieve(len);

91 }

92 } while(isET || ToWriteBytes() > 10240);

93 return len;

94 }

95

96 bool HttpConn::process() {

97 request\_.Init();

98 if(readBuff\_.ReadableBytes() <= 0) {

99 return false;

100 }

101 else if(request\_.parse(readBuff\_)) {

102 LOG\_DEBUG("%s", request\_.path().c\_str());

103 response\_.Init(srcDir, request\_.path(), request\_.IsKeepAlive(), 200);

104 } else {

105 response\_.Init(srcDir, request\_.path(), false, 400);

106 }

107

108 response\_.MakeResponse(writeBuff\_);

109 /\* 响应头 \*/

110 iov\_[0].iov\_base = const\_cast<char\*>(writeBuff\_.Peek());

111 iov\_[0].iov\_len = writeBuff\_.ReadableBytes();

112 iovCnt\_ = 1;

113

114 /\* 文件 \*/

115 if(response\_.FileLen() > 0 && response\_.File()) {

116 iov\_[1].iov\_base = response\_.File();

117 iov\_[1].iov\_len = response\_.FileLen();

118 iovCnt\_ = 2;

119 }

120 LOG\_DEBUG("filesize:%d, %d to %d", response\_.FileLen() , iovCnt\_, ToWriteBytes());

121 return true;

122 }

http\httpconn.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-15

3 \* @copyleft Apache 2.0

4 \*/

5

6 #ifndef HTTP\_CONN\_H

7 #define HTTP\_CONN\_H

8

9 #include <sys/types.h>

10 #include <sys/uio.h> // readv/writev

11 #include <arpa/inet.h> // sockaddr\_in

12 #include <stdlib.h> // atoi()

13 #include <errno.h>

14

15 #include "../log/log.h"

16 #include "../pool/sqlconnRAII.h"

17 #include "../buffer/buffer.h"

18 #include "httprequest.h"

19 #include "httpresponse.h"

20

21 class HttpConn {

22 public:

23 HttpConn();

24

25 ~HttpConn();

26

27 void init(int sockFd, const sockaddr\_in& addr);

28

29 ssize\_t read(int\* saveErrno);

30

31 ssize\_t write(int\* saveErrno);

32

33 void Close();

34

35 int GetFd() const;

36

37 int GetPort() const;

38

39 const char\* GetIP() const;

40

41 sockaddr\_in GetAddr() const;

42

43 bool process();

44

45 int ToWriteBytes() {

46 return iov\_[0].iov\_len + iov\_[1].iov\_len;

47 }

48

49 bool IsKeepAlive() const {

50 return request\_.IsKeepAlive();

51 }

52

53 static bool isET;

54 static const char\* srcDir;

55 static std::atomic<int> userCount;

56

57 private:

58

59 int fd\_;

60 struct sockaddr\_in addr\_;

61

62 bool isClose\_;

63

64 int iovCnt\_;

65 struct iovec iov\_[2];

66

67 Buffer readBuff\_; // 读缓冲区

68 Buffer writeBuff\_; // 写缓冲区

69

70 HttpRequest request\_;

71 HttpResponse response\_;

72 };

73

74

75 #endif //HTTP\_CONN\_H

http\httprequest.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-26

3 \* @copyleft Apache 2.0

4 \*/

5 #include "httprequest.h"

6 using namespace std;

7

8 const unordered\_set<string> HttpRequest::DEFAULT\_HTML{

9 "/index", "/register", "/login",

10 "/welcome", "/video", "/picture", };

11

12 const unordered\_map<string, int> HttpRequest::DEFAULT\_HTML\_TAG {

13 {"/register.html", 0}, {"/login.html", 1}, };

14

15 void HttpRequest::Init() {

16 method\_ = path\_ = version\_ = body\_ = "";

17 state\_ = REQUEST\_LINE;

18 header\_.clear();

19 post\_.clear();

20 }

21

22 bool HttpRequest::IsKeepAlive() const {

23 if(header\_.count("Connection") == 1) {

24 return header\_.find("Connection")->second == "keep-alive" && version\_ == "1.1";

25 }

26 return false;

27 }

28

29 bool HttpRequest::parse(Buffer& buff) {

30 const char CRLF[] = "\r\n";

31 if(buff.ReadableBytes() <= 0) {

32 return false;

33 }

34 while(buff.ReadableBytes() && state\_ != FINISH) {

35 const char\* lineEnd = search(buff.Peek(), buff.BeginWriteConst(), CRLF, CRLF + 2);

36 std::string line(buff.Peek(), lineEnd);

37 switch(state\_)

38 {

39 case REQUEST\_LINE:

40 if(!ParseRequestLine\_(line)) {

41 return false;

42 }

43 ParsePath\_();

44 break;

45 case HEADERS:

46 ParseHeader\_(line);

47 if(buff.ReadableBytes() <= 2) {

48 state\_ = FINISH;

49 }

50 break;

51 case BODY:

52 ParseBody\_(line);

53 break;

54 default:

55 break;

56 }

57 if(lineEnd == buff.BeginWrite()) { break; }

58 buff.RetrieveUntil(lineEnd + 2);

59 }

60 LOG\_DEBUG("[%s], [%s], [%s]", method\_.c\_str(), path\_.c\_str(), version\_.c\_str());

61 return true;

62 }

63

64 void HttpRequest::ParsePath\_() {

65 if(path\_ == "/") {

66 path\_ = "/index.html";

67 }

68 else {

69 for(auto &item: DEFAULT\_HTML) {

70 if(item == path\_) {

71 path\_ += ".html";

72 break;

73 }

74 }

75 }

76 }

77

78 bool HttpRequest::ParseRequestLine\_(const string& line) {

79 regex patten("^([^ ]\*) ([^ ]\*) HTTP/([^ ]\*)$");

80 smatch subMatch;

81 if(regex\_match(line, subMatch, patten)) {

82 method\_ = subMatch[1];

83 path\_ = subMatch[2];

84 version\_ = subMatch[3];

85 state\_ = HEADERS;

86 return true;

87 }

88 LOG\_ERROR("RequestLine Error");

89 return false;

90 }

91

92 void HttpRequest::ParseHeader\_(const string& line) {

93 regex patten("^([^:]\*): ?(.\*)$");

94 smatch subMatch;

95 if(regex\_match(line, subMatch, patten)) {

96 header\_[subMatch[1]] = subMatch[2];

97 }

98 else {

99 state\_ = BODY;

100 }

101 }

102

103 void HttpRequest::ParseBody\_(const string& line) {

104 body\_ = line;

105 ParsePost\_();

106 state\_ = FINISH;

107 LOG\_DEBUG("Body:%s, len:%d", line.c\_str(), line.size());

108 }

109

110 int HttpRequest::ConverHex(char ch) {

111 if(ch >= 'A' && ch <= 'F') return ch -'A' + 10;

112 if(ch >= 'a' && ch <= 'f') return ch -'a' + 10;

113 return ch;

114 }

115

116 void HttpRequest::ParsePost\_() {

117 if(method\_ == "POST" && header\_["Content-Type"] == "application/x-www-form-urlencoded") {

118 ParseFromUrlencoded\_();

119 if(DEFAULT\_HTML\_TAG.count(path\_)) {

120 int tag = DEFAULT\_HTML\_TAG.find(path\_)->second;

121 LOG\_DEBUG("Tag:%d", tag);

122 if(tag == 0 || tag == 1) {

123 bool isLogin = (tag == 1);

124 if(UserVerify(post\_["username"], post\_["password"], isLogin)) {

125 path\_ = "/welcome.html";

126 }

127 else {

128 path\_ = "/error.html";

129 }

130 }

131 }

132 }

133 }

134

135 void HttpRequest::ParseFromUrlencoded\_() {

136 if(body\_.size() == 0) { return; }

137

138 string key, value;

139 int num = 0;

140 int n = body\_.size();

141 int i = 0, j = 0;

142

143 for(; i < n; i++) {

144 char ch = body\_[i];

145 switch (ch) {

146 case '=':

147 key = body\_.substr(j, i - j);

148 j = i + 1;

149 break;

150 case '+':

151 body\_[i] = ' ';

152 break;

153 case '%':

154 num = ConverHex(body\_[i + 1]) \* 16 + ConverHex(body\_[i + 2]);

155 body\_[i + 2] = num % 10 + '0';

156 body\_[i + 1] = num / 10 + '0';

157 i += 2;

158 break;

159 case '&':

160 value = body\_.substr(j, i - j);

161 j = i + 1;

162 post\_[key] = value;

163 LOG\_DEBUG("%s = %s", key.c\_str(), value.c\_str());

164 break;

165 default:

166 break;

167 }

168 }

169 assert(j <= i);

170 if(post\_.count(key) == 0 && j < i) {

171 value = body\_.substr(j, i - j);

172 post\_[key] = value;

173 }

174 }

175

176 bool HttpRequest::UserVerify(const string &name, const string &pwd, bool isLogin) {

177 if(name == "" || pwd == "") { return false; }

178 LOG\_INFO("Verify name:%s pwd:%s", name.c\_str(), pwd.c\_str());

179 MYSQL\* sql;

180 SqlConnRAII(&sql, SqlConnPool::Instance());

181 assert(sql);

182

183 bool flag = false;

184 unsigned int j = 0;

185 char order[256] = { 0 };

186 MYSQL\_FIELD \*fields = nullptr;

187 MYSQL\_RES \*res = nullptr;

188

189 if(!isLogin) { flag = true; }

190 /\* 查询用户及密码 \*/

191 snprintf(order, 256, "SELECT username, password FROM user WHERE username='%s' LIMIT 1", name.c\_str());

192 LOG\_DEBUG("%s", order);

193

194 if(mysql\_query(sql, order)) {

195 mysql\_free\_result(res);

196 return false;

197 }

198 res = mysql\_store\_result(sql);

199 j = mysql\_num\_fields(res);

200 fields = mysql\_fetch\_fields(res);

201

202 while(MYSQL\_ROW row = mysql\_fetch\_row(res)) {

203 LOG\_DEBUG("MYSQL ROW: %s %s", row[0], row[1]);

204 string password(row[1]);

205 /\* 注册行为 且 用户名未被使用\*/

206 if(isLogin) {

207 if(pwd == password) { flag = true; }

208 else {

209 flag = false;

210 LOG\_DEBUG("pwd error!");

211 }

212 }

213 else {

214 flag = false;

215 LOG\_DEBUG("user used!");

216 }

217 }

218 mysql\_free\_result(res);

219

220 /\* 注册行为 且 用户名未被使用\*/

221 if(!isLogin && flag == true) {

222 LOG\_DEBUG("regirster!");

223 bzero(order, 256);

224 snprintf(order, 256,"INSERT INTO user(username, password) VALUES('%s','%s')", name.c\_str(), pwd.c\_str());

225 LOG\_DEBUG( "%s", order);

226 if(mysql\_query(sql, order)) {

227 LOG\_DEBUG( "Insert error!");

228 flag = false;

229 }

230 flag = true;

231 }

232 SqlConnPool::Instance()->FreeConn(sql);

233 LOG\_DEBUG( "UserVerify success!!");

234 return flag;

235 }

236

237 std::string HttpRequest::path() const{

238 return path\_;

239 }

240

241 std::string& HttpRequest::path(){

242 return path\_;

243 }

244 std::string HttpRequest::method() const {

245 return method\_;

246 }

247

248 std::string HttpRequest::version() const {

249 return version\_;

250 }

251

252 std::string HttpRequest::GetPost(const std::string& key) const {

253 assert(key != "");

254 if(post\_.count(key) == 1) {

255 return post\_.find(key)->second;

256 }

257 return "";

258 }

259

260 std::string HttpRequest::GetPost(const char\* key) const {

261 assert(key != nullptr);

262 if(post\_.count(key) == 1) {

263 return post\_.find(key)->second;

264 }

265 return "";

266 }

http\httprequest.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-25

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef HTTP\_REQUEST\_H

6 #define HTTP\_REQUEST\_H

7

8 #include <unordered\_map>

9 #include <unordered\_set>

10 #include <string>

11 #include <regex>

12 #include <errno.h>

13 #include <mysql/mysql.h> //mysql

14

15 #include "../buffer/buffer.h"

16 #include "../log/log.h"

17 #include "../pool/sqlconnpool.h"

18 #include "../pool/sqlconnRAII.h"

19

20 class HttpRequest {

21 public:

22 enum PARSE\_STATE {

23 REQUEST\_LINE,

24 HEADERS,

25 BODY,

26 FINISH,

27 };

28

29 enum HTTP\_CODE {

30 NO\_REQUEST = 0,

31 GET\_REQUEST,

32 BAD\_REQUEST,

33 NO\_RESOURSE,

34 FORBIDDENT\_REQUEST,

35 FILE\_REQUEST,

36 INTERNAL\_ERROR,

37 CLOSED\_CONNECTION,

38 };

39

40 HttpRequest() { Init(); }

41 ~HttpRequest() = default;

42

43 void Init();

44 bool parse(Buffer& buff);

45

46 std::string path() const;

47 std::string& path();

48 std::string method() const;

49 std::string version() const;

50 std::string GetPost(const std::string& key) const;

51 std::string GetPost(const char\* key) const;

52

53 bool IsKeepAlive() const;

54

55 /\*

56 todo

57 void HttpConn::ParseFormData() {}

58 void HttpConn::ParseJson() {}

59 \*/

60

61 private:

62 bool ParseRequestLine\_(const std::string& line);

63 void ParseHeader\_(const std::string& line);

64 void ParseBody\_(const std::string& line);

65

66 void ParsePath\_();

67 void ParsePost\_();

68 void ParseFromUrlencoded\_();

69

70 static bool UserVerify(const std::string& name, const std::string& pwd, bool isLogin);

71

72 PARSE\_STATE state\_;

73 std::string method\_, path\_, version\_, body\_;

74 std::unordered\_map<std::string, std::string> header\_;

75 std::unordered\_map<std::string, std::string> post\_;

76

77 static const std::unordered\_set<std::string> DEFAULT\_HTML;

78 static const std::unordered\_map<std::string, int> DEFAULT\_HTML\_TAG;

79 static int ConverHex(char ch);

80 };

81

82

83 #endif //HTTP\_REQUEST\_H

http\httpresponse.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-27

3 \* @copyleft Apache 2.0

4 \*/

5 #include "httpresponse.h"

6

7 using namespace std;

8

9 const unordered\_map<string, string> HttpResponse::SUFFIX\_TYPE = {

10 { ".html", "text/html" },

11 { ".xml", "text/xml" },

12 { ".xhtml", "application/xhtml+xml" },

13 { ".txt", "text/plain" },

14 { ".rtf", "application/rtf" },

15 { ".pdf", "application/pdf" },

16 { ".word", "application/nsword" },

17 { ".png", "image/png" },

18 { ".gif", "image/gif" },

19 { ".jpg", "image/jpeg" },

20 { ".jpeg", "image/jpeg" },

21 { ".au", "audio/basic" },

22 { ".mpeg", "video/mpeg" },

23 { ".mpg", "video/mpeg" },

24 { ".avi", "video/x-msvideo" },

25 { ".gz", "application/x-gzip" },

26 { ".tar", "application/x-tar" },

27 { ".css", "text/css "},

28 { ".js", "text/javascript "},

29 };

30

31 const unordered\_map<int, string> HttpResponse::CODE\_STATUS = {

32 { 200, "OK" },

33 { 400, "Bad Request" },

34 { 403, "Forbidden" },

35 { 404, "Not Found" },

36 };

37

38 const unordered\_map<int, string> HttpResponse::CODE\_PATH = {

39 { 400, "/400.html" },

40 { 403, "/403.html" },

41 { 404, "/404.html" },

42 };

43

44 HttpResponse::HttpResponse() {

45 code\_ = -1;

46 path\_ = srcDir\_ = "";

47 isKeepAlive\_ = false;

48 mmFile\_ = nullptr;

49 mmFileStat\_ = { 0 };

50 };

51

52 HttpResponse::~HttpResponse() {

53 UnmapFile();

54 }

55

56 void HttpResponse::Init(const string& srcDir, string& path, bool isKeepAlive, int code){

57 assert(srcDir != "");

58 if(mmFile\_) { UnmapFile(); }

59 code\_ = code;

60 isKeepAlive\_ = isKeepAlive;

61 path\_ = path;

62 srcDir\_ = srcDir;

63 mmFile\_ = nullptr;

64 mmFileStat\_ = { 0 };

65 }

66

67 void HttpResponse::MakeResponse(Buffer& buff) {

68 /\* 判断请求的资源文件 \*/

69 if(stat((srcDir\_ + path\_).data(), &mmFileStat\_) < 0 || S\_ISDIR(mmFileStat\_.st\_mode)) {

70 code\_ = 404;

71 }

72 else if(!(mmFileStat\_.st\_mode & S\_IROTH)) {

73 code\_ = 403;

74 }

75 else if(code\_ == -1) {

76 code\_ = 200;

77 }

78 ErrorHtml\_();

79 AddStateLine\_(buff);

80 AddHeader\_(buff);

81 AddContent\_(buff);

82 }

83

84 char\* HttpResponse::File() {

85 return mmFile\_;

86 }

87

88 size\_t HttpResponse::FileLen() const {

89 return mmFileStat\_.st\_size;

90 }

91

92 void HttpResponse::ErrorHtml\_() {

93 if(CODE\_PATH.count(code\_) == 1) {

94 path\_ = CODE\_PATH.find(code\_)->second;

95 stat((srcDir\_ + path\_).data(), &mmFileStat\_);

96 }

97 }

98

99 void HttpResponse::AddStateLine\_(Buffer& buff) {

100 string status;

101 if(CODE\_STATUS.count(code\_) == 1) {

102 status = CODE\_STATUS.find(code\_)->second;

103 }

104 else {

105 code\_ = 400;

106 status = CODE\_STATUS.find(400)->second;

107 }

108 buff.Append("HTTP/1.1 " + to\_string(code\_) + " " + status + "\r\n");

109 }

110

111 void HttpResponse::AddHeader\_(Buffer& buff) {

112 buff.Append("Connection: ");

113 if(isKeepAlive\_) {

114 buff.Append("keep-alive\r\n");

115 buff.Append("keep-alive: max=6, timeout=120\r\n");

116 } else{

117 buff.Append("close\r\n");

118 }

119 buff.Append("Content-type: " + GetFileType\_() + "\r\n");

120 }

121

122 void HttpResponse::AddContent\_(Buffer& buff) {

123 int srcFd = open((srcDir\_ + path\_).data(), O\_RDONLY);

124 if(srcFd < 0) {

125 ErrorContent(buff, "File NotFound!");

126 return;

127 }

128

129 /\* 将文件映射到内存提高文件的访问速度

130 MAP\_PRIVATE 建立一个写入时拷贝的私有映射\*/

131 LOG\_DEBUG("file path %s", (srcDir\_ + path\_).data());

132 int\* mmRet = (int\*)mmap(0, mmFileStat\_.st\_size, PROT\_READ, MAP\_PRIVATE, srcFd, 0);

133 if(\*mmRet == -1) {

134 ErrorContent(buff, "File NotFound!");

135 return;

136 }

137 mmFile\_ = (char\*)mmRet;

138 close(srcFd);

139 buff.Append("Content-length: " + to\_string(mmFileStat\_.st\_size) + "\r\n\r\n");

140 }

141

142 void HttpResponse::UnmapFile() {

143 if(mmFile\_) {

144 munmap(mmFile\_, mmFileStat\_.st\_size);

145 mmFile\_ = nullptr;

146 }

147 }

148

149 string HttpResponse::GetFileType\_() {

150 /\* 判断文件类型 \*/

151 string::size\_type idx = path\_.find\_last\_of('.');

152 if(idx == string::npos) {

153 return "text/plain";

154 }

155 string suffix = path\_.substr(idx);

156 if(SUFFIX\_TYPE.count(suffix) == 1) {

157 return SUFFIX\_TYPE.find(suffix)->second;

158 }

159 return "text/plain";

160 }

161

162 void HttpResponse::ErrorContent(Buffer& buff, string message)

163 {

164 string body;

165 string status;

166 body += "<html><title>Error</title>";

167 body += "<body bgcolor=\"ffffff\">";

168 if(CODE\_STATUS.count(code\_) == 1) {

169 status = CODE\_STATUS.find(code\_)->second;

170 } else {

171 status = "Bad Request";

172 }

173 body += to\_string(code\_) + " : " + status + "\n";

174 body += "<p>" + message + "</p>";

175 body += "<hr><em>TinyWebServer</em></body></html>";

176

177 buff.Append("Content-length: " + to\_string(body.size()) + "\r\n\r\n");

178 buff.Append(body);

179 }

http\httpresponse.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-25

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef HTTP\_RESPONSE\_H

6 #define HTTP\_RESPONSE\_H

7

8 #include <unordered\_map>

9 #include <fcntl.h> // open

10 #include <unistd.h> // close

11 #include <sys/stat.h> // stat

12 #include <sys/mman.h> // mmap, munmap

13

14 #include "../buffer/buffer.h"

15 #include "../log/log.h"

16

17 class HttpResponse {

18 public:

19 HttpResponse();

20 ~HttpResponse();

21

22 void Init(const std::string& srcDir, std::string& path, bool isKeepAlive = false, int code = -1);

23 void MakeResponse(Buffer& buff);

24 void UnmapFile();

25 char\* File();

26 size\_t FileLen() const;

27 void ErrorContent(Buffer& buff, std::string message);

28 int Code() const { return code\_; }

29

30 private:

31 void AddStateLine\_(Buffer &buff);

32 void AddHeader\_(Buffer &buff);

33 void AddContent\_(Buffer &buff);

34

35 void ErrorHtml\_();

36 std::string GetFileType\_();

37

38 int code\_;

39 bool isKeepAlive\_;

40

41 std::string path\_;

42 std::string srcDir\_;

43

44 char\* mmFile\_;

45 struct stat mmFileStat\_;

46

47 static const std::unordered\_map<std::string, std::string> SUFFIX\_TYPE;

48 static const std::unordered\_map<int, std::string> CODE\_STATUS;

49 static const std::unordered\_map<int, std::string> CODE\_PATH;

50 };

51

52

53 #endif //HTTP\_RESPONSE\_H

log\blockqueue.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-16

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef BLOCKQUEUE\_H

6 #define BLOCKQUEUE\_H

7

8 #include <mutex>

9 #include <deque>

10 #include <condition\_variable>

11 #include <sys/time.h>

12

13 template<class T>

14 class BlockDeque {

15 public:

16 explicit BlockDeque(size\_t MaxCapacity = 1000);

17

18 ~BlockDeque();

19

20 void clear();

21

22 bool empty();

23

24 bool full();

25

26 void Close();

27

28 size\_t size();

29

30 size\_t capacity();

31

32 T front();

33

34 T back();

35

36 void push\_back(const T &item);

37

38 void push\_front(const T &item);

39

40 bool pop(T &item);

41

42 bool pop(T &item, int timeout);

43

44 void flush();

45

46 private:

47 std::deque<T> deq\_;

48

49 size\_t capacity\_;

50

51 std::mutex mtx\_;

52

53 bool isClose\_;

54

55 std::condition\_variable condConsumer\_;

56

57 std::condition\_variable condProducer\_;

58 };

59

60

61 template<class T>

62 BlockDeque<T>::BlockDeque(size\_t MaxCapacity) :capacity\_(MaxCapacity) {

63 assert(MaxCapacity > 0);

64 isClose\_ = false;

65 }

66

67 template<class T>

68 BlockDeque<T>::~BlockDeque() {

69 Close();

70 };

71

72 template<class T>

73 void BlockDeque<T>::Close() {

74 {

75 std::lock\_guard<std::mutex> locker(mtx\_);

76 deq\_.clear();

77 isClose\_ = true;

78 }

79 condProducer\_.notify\_all();

80 condConsumer\_.notify\_all();

81 };

82

83 template<class T>

84 void BlockDeque<T>::flush() {

85 condConsumer\_.notify\_one();

86 };

87

88 template<class T>

89 void BlockDeque<T>::clear() {

90 std::lock\_guard<std::mutex> locker(mtx\_);

91 deq\_.clear();

92 }

93

94 template<class T>

95 T BlockDeque<T>::front() {

96 std::lock\_guard<std::mutex> locker(mtx\_);

97 return deq\_.front();

98 }

99

100 template<class T>

101 T BlockDeque<T>::back() {

102 std::lock\_guard<std::mutex> locker(mtx\_);

103 return deq\_.back();

104 }

105

106 template<class T>

107 size\_t BlockDeque<T>::size() {

108 std::lock\_guard<std::mutex> locker(mtx\_);

109 return deq\_.size();

110 }

111

112 template<class T>

113 size\_t BlockDeque<T>::capacity() {

114 std::lock\_guard<std::mutex> locker(mtx\_);

115 return capacity\_;

116 }

117

118 template<class T>

119 void BlockDeque<T>::push\_back(const T &item) {

120 std::unique\_lock<std::mutex> locker(mtx\_);

121 while(deq\_.size() >= capacity\_) {

122 condProducer\_.wait(locker);

123 }

124 deq\_.push\_back(item);

125 condConsumer\_.notify\_one();

126 }

127

128 template<class T>

129 void BlockDeque<T>::push\_front(const T &item) {

130 std::unique\_lock<std::mutex> locker(mtx\_);

131 while(deq\_.size() >= capacity\_) {

132 condProducer\_.wait(locker);

133 }

134 deq\_.push\_front(item);

135 condConsumer\_.notify\_one();

136 }

137

138 template<class T>

139 bool BlockDeque<T>::empty() {

140 std::lock\_guard<std::mutex> locker(mtx\_);

141 return deq\_.empty();

142 }

143

144 template<class T>

145 bool BlockDeque<T>::full(){

146 std::lock\_guard<std::mutex> locker(mtx\_);

147 return deq\_.size() >= capacity\_;

148 }

149

150 template<class T>

151 bool BlockDeque<T>::pop(T &item) {

152 std::unique\_lock<std::mutex> locker(mtx\_);

153 while(deq\_.empty()){

154 condConsumer\_.wait(locker);

155 if(isClose\_){

156 return false;

157 }

158 }

159 item = deq\_.front();

160 deq\_.pop\_front();

161 condProducer\_.notify\_one();

162 return true;

163 }

164

165 template<class T>

166 bool BlockDeque<T>::pop(T &item, int timeout) {

167 std::unique\_lock<std::mutex> locker(mtx\_);

168 while(deq\_.empty()){

169 if(condConsumer\_.wait\_for(locker, std::chrono::seconds(timeout))

170 == std::cv\_status::timeout){

171 return false;

172 }

173 if(isClose\_){

174 return false;

175 }

176 }

177 item = deq\_.front();

178 deq\_.pop\_front();

179 condProducer\_.notify\_one();

180 return true;

181 }

182

183 #endif // BLOCKQUEUE\_H

log\log.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-16

3 \* @copyleft Apache 2.0

4 \*/

5 #include "log.h"

6

7 using namespace std;

8

9 Log::Log() {

10 lineCount\_ = 0;

11 isAsync\_ = false;

12 writeThread\_ = nullptr;

13 deque\_ = nullptr;

14 toDay\_ = 0;

15 fp\_ = nullptr;

16 }

17

18 Log::~Log() {

19 if(writeThread\_ && writeThread\_->joinable()) {

20 while(!deque\_->empty()) {

21 deque\_->flush();

22 };

23 deque\_->Close();

24 writeThread\_->join();

25 }

26 if(fp\_) {

27 lock\_guard<mutex> locker(mtx\_);

28 flush();

29 fclose(fp\_);

30 }

31 }

32

33 int Log::GetLevel() {

34 lock\_guard<mutex> locker(mtx\_);

35 return level\_;

36 }

37

38 void Log::SetLevel(int level) {

39 lock\_guard<mutex> locker(mtx\_);

40 level\_ = level;

41 }

42

43 void Log::init(int level = 1, const char\* path, const char\* suffix,

44 int maxQueueSize) {

45 isOpen\_ = true;

46 level\_ = level;

47 if(maxQueueSize > 0) {

48 isAsync\_ = true;

49 if(!deque\_) {

50 unique\_ptr<BlockDeque<std::string>> newDeque(new BlockDeque<std::string>);

51 deque\_ = move(newDeque);

52

53 std::unique\_ptr<std::thread> NewThread(new thread(FlushLogThread));

54 writeThread\_ = move(NewThread);

55 }

56 } else {

57 isAsync\_ = false;

58 }

59

60 lineCount\_ = 0;

61

62 time\_t timer = time(nullptr);

63 struct tm \*sysTime = localtime(&timer);

64 struct tm t = \*sysTime;

65 path\_ = path;

66 suffix\_ = suffix;

67 char fileName[LOG\_NAME\_LEN] = {0};

68 snprintf(fileName, LOG\_NAME\_LEN - 1, "%s/%04d\_%02d\_%02d%s",

69 path\_, t.tm\_year + 1900, t.tm\_mon + 1, t.tm\_mday, suffix\_);

70 toDay\_ = t.tm\_mday;

71

72 {

73 lock\_guard<mutex> locker(mtx\_);

74 buff\_.RetrieveAll();

75 if(fp\_) {

76 flush();

77 fclose(fp\_);

78 }

79

80 fp\_ = fopen(fileName, "a");

81 if(fp\_ == nullptr) {

82 mkdir(path\_, 0777);

83 fp\_ = fopen(fileName, "a");

84 }

85 assert(fp\_ != nullptr);

86 }

87 }

88

89 void Log::write(int level, const char \*format, ...) {

90 struct timeval now = {0, 0};

91 gettimeofday(&now, nullptr);

92 time\_t tSec = now.tv\_sec;

93 struct tm \*sysTime = localtime(&tSec);

94 struct tm t = \*sysTime;

95 va\_list vaList;

96

97 /\* 日志日期 日志行数 \*/

98 if (toDay\_ != t.tm\_mday || (lineCount\_ && (lineCount\_ % MAX\_LINES == 0)))

99 {

100 unique\_lock<mutex> locker(mtx\_);

101 locker.unlock();

102

103 char newFile[LOG\_NAME\_LEN];

104 char tail[36] = {0};

105 snprintf(tail, 36, "%04d\_%02d\_%02d", t.tm\_year + 1900, t.tm\_mon + 1, t.tm\_mday);

106

107 if (toDay\_ != t.tm\_mday)

108 {

109 snprintf(newFile, LOG\_NAME\_LEN - 72, "%s/%s%s", path\_, tail, suffix\_);

110 toDay\_ = t.tm\_mday;

111 lineCount\_ = 0;

112 }

113 else {

114 snprintf(newFile, LOG\_NAME\_LEN - 72, "%s/%s-%d%s", path\_, tail, (lineCount\_ / MAX\_LINES), suffix\_);

115 }

116

117 locker.lock();

118 flush();

119 fclose(fp\_);

120 fp\_ = fopen(newFile, "a");

121 assert(fp\_ != nullptr);

122 }

123

124 {

125 unique\_lock<mutex> locker(mtx\_);

126 lineCount\_++;

127 int n = snprintf(buff\_.BeginWrite(), 128, "%d-%02d-%02d %02d:%02d:%02d.%06ld ",

128 t.tm\_year + 1900, t.tm\_mon + 1, t.tm\_mday,

129 t.tm\_hour, t.tm\_min, t.tm\_sec, now.tv\_usec);

130

131 buff\_.HasWritten(n);

132 AppendLogLevelTitle\_(level);

133

134 va\_start(vaList, format);

135 int m = vsnprintf(buff\_.BeginWrite(), buff\_.WritableBytes(), format, vaList);

136 va\_end(vaList);

137

138 buff\_.HasWritten(m);

139 buff\_.Append("\n\0", 2);

140

141 if(isAsync\_ && deque\_ && !deque\_->full()) {

142 deque\_->push\_back(buff\_.RetrieveAllToStr());

143 } else {

144 fputs(buff\_.Peek(), fp\_);

145 }

146 buff\_.RetrieveAll();

147 }

148 }

149

150 void Log::AppendLogLevelTitle\_(int level) {

151 switch(level) {

152 case 0:

153 buff\_.Append("[debug]: ", 9);

154 break;

155 case 1:

156 buff\_.Append("[info] : ", 9);

157 break;

158 case 2:

159 buff\_.Append("[warn] : ", 9);

160 break;

161 case 3:

162 buff\_.Append("[error]: ", 9);

163 break;

164 default:

165 buff\_.Append("[info] : ", 9);

166 break;

167 }

168 }

169

170 void Log::flush() {

171 if(isAsync\_) {

172 deque\_->flush();

173 }

174 fflush(fp\_);

175 }

176

177 void Log::AsyncWrite\_() {

178 string str = "";

179 while(deque\_->pop(str)) {

180 lock\_guard<mutex> locker(mtx\_);

181 fputs(str.c\_str(), fp\_);

182 }

183 }

184

185 Log\* Log::Instance() {

186 static Log inst;

187 return &inst;

188 }

189

190 void Log::FlushLogThread() {

191 Log::Instance()->AsyncWrite\_();

192 }

log\log.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-16

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef LOG\_H

6 #define LOG\_H

7

8 #include <mutex>

9 #include <string>

10 #include <thread>

11 #include <sys/time.h>

12 #include <string.h>

13 #include <stdarg.h> // vastart va\_end

14 #include <assert.h>

15 #include <sys/stat.h> //mkdir

16 #include "blockqueue.h"

17 #include "../buffer/buffer.h"

18

19 class Log {

20 public:

21 void init(int level, const char\* path = "./log",

22 const char\* suffix =".log",

23 int maxQueueCapacity = 1024);

24

25 static Log\* Instance();

26 static void FlushLogThread();

27

28 void write(int level, const char \*format,...);

29 void flush();

30

31 int GetLevel();

32 void SetLevel(int level);

33 bool IsOpen() { return isOpen\_; }

34

35 private:

36 Log();

37 void AppendLogLevelTitle\_(int level);

38 virtual ~Log();

39 void AsyncWrite\_();

40

41 private:

42 static const int LOG\_PATH\_LEN = 256;

43 static const int LOG\_NAME\_LEN = 256;

44 static const int MAX\_LINES = 50000;

45

46 const char\* path\_;

47 const char\* suffix\_;

48

49 int MAX\_LINES\_;

50

51 int lineCount\_;

52 int toDay\_;

53

54 bool isOpen\_;

55

56 Buffer buff\_;

57 int level\_;

58 bool isAsync\_;

59

60 FILE\* fp\_;

61 std::unique\_ptr<BlockDeque<std::string>> deque\_;

62 std::unique\_ptr<std::thread> writeThread\_;

63 std::mutex mtx\_;

64 };

65

66 #define LOG\_BASE(level, format, ...) \

67 do {\

68 Log\* log = Log::Instance();\

69 if (log->IsOpen() && log->GetLevel() <= level) {\

70 log->write(level, format, ##\_\_VA\_ARGS\_\_); \

71 log->flush();\

72 }\

73 } while(0);

74

75 #define LOG\_DEBUG(format, ...) do {LOG\_BASE(0, format, ##\_\_VA\_ARGS\_\_)} while(0);

76 #define LOG\_INFO(format, ...) do {LOG\_BASE(1, format, ##\_\_VA\_ARGS\_\_)} while(0);

77 #define LOG\_WARN(format, ...) do {LOG\_BASE(2, format, ##\_\_VA\_ARGS\_\_)} while(0);

78 #define LOG\_ERROR(format, ...) do {LOG\_BASE(3, format, ##\_\_VA\_ARGS\_\_)} while(0);

79

80 #endif //LOG\_H

main.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-18

3 \* @copyleft Apache 2.0

4 \*/

5 #include <unistd.h>

6 #include "server/webserver.h"

7

8 int main() {

9 /\* 守护进程 后台运行 \*/

10 //daemon(1, 0);

11

12 WebServer server(

13 1316, 3, 60000, false, /\* 端口 ET模式 timeoutMs 优雅退出 \*/

14 3306, "root", "root", "webserver", /\* Mysql配置 \*/

15 12, 6, true, 1, 1024); /\* 连接池数量 线程池数量 日志开关 日志等级 日志异步队列容量 \*/

16 server.Start();

17 }

18

pool\sqlconnpool.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-17

3 \* @copyleft Apache 2.0

4 \*/

5

6 #include "sqlconnpool.h"

7 using namespace std;

8

9 SqlConnPool::SqlConnPool() {

10 useCount\_ = 0;

11 freeCount\_ = 0;

12 }

13

14 SqlConnPool\* SqlConnPool::Instance() {

15 static SqlConnPool connPool;

16 return &connPool;

17 }

18

19 void SqlConnPool::Init(const char\* host, int port,

20 const char\* user,const char\* pwd, const char\* dbName,

21 int connSize = 10) {

22 assert(connSize > 0);

23 for (int i = 0; i < connSize; i++) {

24 MYSQL \*sql = nullptr;

25 sql = mysql\_init(sql);

26 if (!sql) {

27 LOG\_ERROR("MySql init error!");

28 assert(sql);

29 }

30 sql = mysql\_real\_connect(sql, host,

31 user, pwd,

32 dbName, port, nullptr, 0);

33 if (!sql) {

34 LOG\_ERROR("MySql Connect error!");

35 }

36 connQue\_.push(sql);

37 }

38 MAX\_CONN\_ = connSize;

39 sem\_init(&semId\_, 0, MAX\_CONN\_);

40 }

41

42 MYSQL\* SqlConnPool::GetConn() {

43 MYSQL \*sql = nullptr;

44 if(connQue\_.empty()){

45 LOG\_WARN("SqlConnPool busy!");

46 return nullptr;

47 }

48 sem\_wait(&semId\_);

49 {

50 lock\_guard<mutex> locker(mtx\_);

51 sql = connQue\_.front();

52 connQue\_.pop();

53 }

54 return sql;

55 }

56

57 void SqlConnPool::FreeConn(MYSQL\* sql) {

58 assert(sql);

59 lock\_guard<mutex> locker(mtx\_);

60 connQue\_.push(sql);

61 sem\_post(&semId\_);

62 }

63

64 void SqlConnPool::ClosePool() {

65 lock\_guard<mutex> locker(mtx\_);

66 while(!connQue\_.empty()) {

67 auto item = connQue\_.front();

68 connQue\_.pop();

69 mysql\_close(item);

70 }

71 mysql\_library\_end();

72 }

73

74 int SqlConnPool::GetFreeConnCount() {

75 lock\_guard<mutex> locker(mtx\_);

76 return connQue\_.size();

77 }

78

79 SqlConnPool::~SqlConnPool() {

80 ClosePool();

81 }

pool\sqlconnpool.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-16

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef SQLCONNPOOL\_H

6 #define SQLCONNPOOL\_H

7

8 #include <mysql/mysql.h>

9 #include <string>

10 #include <queue>

11 #include <mutex>

12 #include <semaphore.h>

13 #include <thread>

14 #include "../log/log.h"

15

16 class SqlConnPool {

17 public:

18 static SqlConnPool \*Instance();

19

20 MYSQL \*GetConn();

21 void FreeConn(MYSQL \* conn);

22 int GetFreeConnCount();

23

24 void Init(const char\* host, int port,

25 const char\* user,const char\* pwd,

26 const char\* dbName, int connSize);

27 void ClosePool();

28

29 private:

30 SqlConnPool();

31 ~SqlConnPool();

32

33 int MAX\_CONN\_;

34 int useCount\_;

35 int freeCount\_;

36

37 std::queue<MYSQL \*> connQue\_;

38 std::mutex mtx\_;

39 sem\_t semId\_;

40 };

41

42

43 #endif // SQLCONNPOOL\_H

pool\sqlconnRAII.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-19

3 \* @copyleft Apache 2.0

4 \*/

5

6 #ifndef SQLCONNRAII\_H

7 #define SQLCONNRAII\_H

8 #include "sqlconnpool.h"

9

10 /\* 资源在对象构造初始化 资源在对象析构时释放\*/

11 class SqlConnRAII {

12 public:

13 SqlConnRAII(MYSQL\*\* sql, SqlConnPool \*connpool) {

14 assert(connpool);

15 \*sql = connpool->GetConn();

16 sql\_ = \*sql;

17 connpool\_ = connpool;

18 }

19

20 ~SqlConnRAII() {

21 if(sql\_) { connpool\_->FreeConn(sql\_); }

22 }

23

24 private:

25 MYSQL \*sql\_;

26 SqlConnPool\* connpool\_;

27 };

28

29 #endif //SQLCONNRAII\_H

pool\threadpool.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-15

3 \* @copyleft Apache 2.0

4 \*/

5

6 #ifndef THREADPOOL\_H

7 #define THREADPOOL\_H

8

9 #include <mutex>

10 #include <condition\_variable>

11 #include <queue>

12 #include <thread>

13 #include <functional>

14 class ThreadPool {

15 public:

16 explicit ThreadPool(size\_t threadCount = 8): pool\_(std::make\_shared<Pool>()) {

17 assert(threadCount > 0);

18 for(size\_t i = 0; i < threadCount; i++) {

19 std::thread([pool = pool\_] {

20 std::unique\_lock<std::mutex> locker(pool->mtx);

21 while(true) {

22 if(!pool->tasks.empty()) {

23 auto task = std::move(pool->tasks.front());

24 pool->tasks.pop();

25 locker.unlock();

26 task();

27 locker.lock();

28 }

29 else if(pool->isClosed) break;

30 else pool->cond.wait(locker);

31 }

32 }).detach();

33 }

34 }

35

36 ThreadPool() = default;

37

38 ThreadPool(ThreadPool&&) = default;

39

40 ~ThreadPool() {

41 if(static\_cast<bool>(pool\_)) {

42 {

43 std::lock\_guard<std::mutex> locker(pool\_->mtx);

44 pool\_->isClosed = true;

45 }

46 pool\_->cond.notify\_all();

47 }

48 }

49

50 template<class F>

51 void AddTask(F&& task) {

52 {

53 std::lock\_guard<std::mutex> locker(pool\_->mtx);

54 pool\_->tasks.emplace(std::forward<F>(task));

55 }

56 pool\_->cond.notify\_one();

57 }

58

59 private:

60 struct Pool {

61 std::mutex mtx;

62 std::condition\_variable cond;

63 bool isClosed;

64 std::queue<std::function<void()>> tasks;

65 };

66 std::shared\_ptr<Pool> pool\_;

67 };

68

69

70 #endif //THREADPOOL\_H

readme.md

0 readme

server\epoller.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-19

3 \* @copyleft Apache 2.0

4 \*/

5

6 #include "epoller.h"

7

8 Epoller::Epoller(int maxEvent):epollFd\_(epoll\_create(512)), events\_(maxEvent){

9 assert(epollFd\_ >= 0 && events\_.size() > 0);

10 }

11

12 Epoller::~Epoller() {

13 close(epollFd\_);

14 }

15

16 bool Epoller::AddFd(int fd, uint32\_t events) {

17 if(fd < 0) return false;

18 epoll\_event ev = {0};

19 ev.data.fd = fd;

20 ev.events = events;

21 return 0 == epoll\_ctl(epollFd\_, EPOLL\_CTL\_ADD, fd, &ev);

22 }

23

24 bool Epoller::ModFd(int fd, uint32\_t events) {

25 if(fd < 0) return false;

26 epoll\_event ev = {0};

27 ev.data.fd = fd;

28 ev.events = events;

29 return 0 == epoll\_ctl(epollFd\_, EPOLL\_CTL\_MOD, fd, &ev);

30 }

31

32 bool Epoller::DelFd(int fd) {

33 if(fd < 0) return false;

34 epoll\_event ev = {0};

35 return 0 == epoll\_ctl(epollFd\_, EPOLL\_CTL\_DEL, fd, &ev);

36 }

37

38 int Epoller::Wait(int timeoutMs) {

39 return epoll\_wait(epollFd\_, &events\_[0], static\_cast<int>(events\_.size()), timeoutMs);

40 }

41

42 int Epoller::GetEventFd(size\_t i) const {

43 assert(i < events\_.size() && i >= 0);

44 return events\_[i].data.fd;

45 }

46

47 uint32\_t Epoller::GetEvents(size\_t i) const {

48 assert(i < events\_.size() && i >= 0);

49 return events\_[i].events;

50 }

server\epoller.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-15

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef EPOLLER\_H

6 #define EPOLLER\_H

7

8 #include <sys/epoll.h> //epoll\_ctl()

9 #include <fcntl.h> // fcntl()

10 #include <unistd.h> // close()

11 #include <assert.h> // close()

12 #include <vector>

13 #include <errno.h>

14

15 class Epoller {

16 public:

17 explicit Epoller(int maxEvent = 1024);

18

19 ~Epoller();

20

21 bool AddFd(int fd, uint32\_t events);

22

23 bool ModFd(int fd, uint32\_t events);

24

25 bool DelFd(int fd);

26

27 int Wait(int timeoutMs = -1);

28

29 int GetEventFd(size\_t i) const;

30

31 uint32\_t GetEvents(size\_t i) const;

32

33 private:

34 int epollFd\_;

35

36 std::vector<struct epoll\_event> events\_;

37 };

38

39 #endif //EPOLLER\_H

server\webserver.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-17

3 \* @copyleft Apache 2.0

4 \*/

5

6 #include "webserver.h"

7

8 using namespace std;

9

10 WebServer::WebServer(

11 int port, int trigMode, int timeoutMS, bool OptLinger,

12 int sqlPort, const char\* sqlUser, const char\* sqlPwd,

13 const char\* dbName, int connPoolNum, int threadNum,

14 bool openLog, int logLevel, int logQueSize):

15 port\_(port), openLinger\_(OptLinger), timeoutMS\_(timeoutMS), isClose\_(false),

16 timer\_(new HeapTimer()), threadpool\_(new ThreadPool(threadNum)), epoller\_(new Epoller())

17 {

18 srcDir\_ = getcwd(nullptr, 256);

19 assert(srcDir\_);

20 strncat(srcDir\_, "/resources/", 16);

21 HttpConn::userCount = 0;

22 HttpConn::srcDir = srcDir\_;

23 SqlConnPool::Instance()->Init("localhost", sqlPort, sqlUser, sqlPwd, dbName, connPoolNum);

24

25 InitEventMode\_(trigMode);

26 if(!InitSocket\_()) { isClose\_ = true;}

27

28 if(openLog) {

29 Log::Instance()->init(logLevel, "./log", ".log", logQueSize);

30 if(isClose\_) { LOG\_ERROR("========== Server init error!=========="); }

31 else {

32 LOG\_INFO("========== Server init ==========");

33 LOG\_INFO("Port:%d, OpenLinger: %s", port\_, OptLinger? "true":"false");

34 LOG\_INFO("Listen Mode: %s, OpenConn Mode: %s",

35 (listenEvent\_ & EPOLLET ? "ET": "LT"),

36 (connEvent\_ & EPOLLET ? "ET": "LT"));

37 LOG\_INFO("LogSys level: %d", logLevel);

38 LOG\_INFO("srcDir: %s", HttpConn::srcDir);

39 LOG\_INFO("SqlConnPool num: %d, ThreadPool num: %d", connPoolNum, threadNum);

40 }

41 }

42 }

43

44 WebServer::~WebServer() {

45 close(listenFd\_);

46 isClose\_ = true;

47 free(srcDir\_);

48 SqlConnPool::Instance()->ClosePool();

49 }

50

51 void WebServer::InitEventMode\_(int trigMode) {

52 listenEvent\_ = EPOLLRDHUP;

53 connEvent\_ = EPOLLONESHOT | EPOLLRDHUP;

54 switch (trigMode)

55 {

56 case 0:

57 break;

58 case 1:

59 connEvent\_ |= EPOLLET;

60 break;

61 case 2:

62 listenEvent\_ |= EPOLLET;

63 break;

64 case 3:

65 listenEvent\_ |= EPOLLET;

66 connEvent\_ |= EPOLLET;

67 break;

68 default:

69 listenEvent\_ |= EPOLLET;

70 connEvent\_ |= EPOLLET;

71 break;

72 }

73 HttpConn::isET = (connEvent\_ & EPOLLET);

74 }

75

76 void WebServer::Start() {

77 int timeMS = -1; /\* epoll wait timeout == -1 无事件将阻塞 \*/

78 if(!isClose\_) { LOG\_INFO("========== Server start =========="); }

79 while(!isClose\_) {

80 if(timeoutMS\_ > 0) {

81 timeMS = timer\_->GetNextTick();

82 }

83 int eventCnt = epoller\_->Wait(timeMS);

84 for(int i = 0; i < eventCnt; i++) {

85 /\* 处理事件 \*/

86 int fd = epoller\_->GetEventFd(i);

87 uint32\_t events = epoller\_->GetEvents(i);

88 if(fd == listenFd\_) {

89 DealListen\_();

90 }

91 else if(events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR)) {

92 assert(users\_.count(fd) > 0);

93 CloseConn\_(&users\_[fd]);

94 }

95 else if(events & EPOLLIN) {

96 assert(users\_.count(fd) > 0);

97 DealRead\_(&users\_[fd]);

98 }

99 else if(events & EPOLLOUT) {

100 assert(users\_.count(fd) > 0);

101 DealWrite\_(&users\_[fd]);

102 } else {

103 LOG\_ERROR("Unexpected event");

104 }

105 }

106 }

107 }

108

109 void WebServer::SendError\_(int fd, const char\*info) {

110 assert(fd > 0);

111 int ret = send(fd, info, strlen(info), 0);

112 if(ret < 0) {

113 LOG\_WARN("send error to client[%d] error!", fd);

114 }

115 close(fd);

116 }

117

118 void WebServer::CloseConn\_(HttpConn\* client) {

119 assert(client);

120 LOG\_INFO("Client[%d] quit!", client->GetFd());

121 epoller\_->DelFd(client->GetFd());

122 client->Close();

123 }

124

125 void WebServer::AddClient\_(int fd, sockaddr\_in addr) {

126 assert(fd > 0);

127 users\_[fd].init(fd, addr);

128 if(timeoutMS\_ > 0) {

129 timer\_->add(fd, timeoutMS\_, std::bind(&WebServer::CloseConn\_, this, &users\_[fd]));

130 }

131 epoller\_->AddFd(fd, EPOLLIN | connEvent\_);

132 SetFdNonblock(fd);

133 LOG\_INFO("Client[%d] in!", users\_[fd].GetFd());

134 }

135

136 void WebServer::DealListen\_() {

137 struct sockaddr\_in addr;

138 socklen\_t len = sizeof(addr);

139 do {

140 int fd = accept(listenFd\_, (struct sockaddr \*)&addr, &len);

141 if(fd <= 0) { return;}

142 else if(HttpConn::userCount >= MAX\_FD) {

143 SendError\_(fd, "Server busy!");

144 LOG\_WARN("Clients is full!");

145 return;

146 }

147 AddClient\_(fd, addr);

148 } while(listenEvent\_ & EPOLLET);

149 }

150

151 void WebServer::DealRead\_(HttpConn\* client) {

152 assert(client);

153 ExtentTime\_(client);

154 threadpool\_->AddTask(std::bind(&WebServer::OnRead\_, this, client));

155 }

156

157 void WebServer::DealWrite\_(HttpConn\* client) {

158 assert(client);

159 ExtentTime\_(client);

160 threadpool\_->AddTask(std::bind(&WebServer::OnWrite\_, this, client));

161 }

162

163 void WebServer::ExtentTime\_(HttpConn\* client) {

164 assert(client);

165 if(timeoutMS\_ > 0) { timer\_->adjust(client->GetFd(), timeoutMS\_); }

166 }

167

168 void WebServer::OnRead\_(HttpConn\* client) {

169 assert(client);

170 int ret = -1;

171 int readErrno = 0;

172 ret = client->read(&readErrno);

173 if(ret <= 0 && readErrno != EAGAIN) {

174 CloseConn\_(client);

175 return;

176 }

177 OnProcess(client);

178 }

179

180 void WebServer::OnProcess(HttpConn\* client) {

181 if(client->process()) {

182 epoller\_->ModFd(client->GetFd(), connEvent\_ | EPOLLOUT);

183 } else {

184 epoller\_->ModFd(client->GetFd(), connEvent\_ | EPOLLIN);

185 }

186 }

187

188 void WebServer::OnWrite\_(HttpConn\* client) {

189 assert(client);

190 int ret = -1;

191 int writeErrno = 0;

192 ret = client->write(&writeErrno);

193 if(client->ToWriteBytes() == 0) {

194 /\* 传输完成 \*/

195 if(client->IsKeepAlive()) {

196 OnProcess(client);

197 return;

198 }

199 }

200 else if(ret < 0) {

201 if(writeErrno == EAGAIN) {

202 /\* 继续传输 \*/

203 epoller\_->ModFd(client->GetFd(), connEvent\_ | EPOLLOUT);

204 return;

205 }

206 }

207 CloseConn\_(client);

208 }

209

210 /\* Create listenFd \*/

211 bool WebServer::InitSocket\_() {

212 int ret;

213 struct sockaddr\_in addr;

214 if(port\_ > 65535 && port\_ < 1024) {

215 LOG\_ERROR("Port:%d error!", port\_);

216 return false;

217 }

218 addr.sin\_family = AF\_INET;

219 addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

220 addr.sin\_port = htons(port\_);

221 struct linger optLinger = { 0 };

222 if(openLinger\_) {

223 /\* 优雅关闭: 直到所剩数据发送完毕或超时 \*/

224 optLinger.l\_onoff = 1;

225 optLinger.l\_linger = 1;

226 }

227

228 listenFd\_ = socket(AF\_INET, SOCK\_STREAM, 0);

229 if(listenFd\_ < 0) {

230 LOG\_ERROR("Create socket error!", port\_);

231 return false;

232 }

233

234 ret = setsockopt(listenFd\_, SOL\_SOCKET, SO\_LINGER, &optLinger, sizeof(optLinger));

235 if(ret < 0) {

236 close(listenFd\_);

237 LOG\_ERROR("Init linger error!", port\_);

238 return false;

239 }

240

241 int optval = 1;

242 /\* 端口复用 \*/

243 /\* 只有最后一个套接字会正常接收数据。 \*/

244 ret = setsockopt(listenFd\_, SOL\_SOCKET, SO\_REUSEADDR, (const void\*)&optval, sizeof(int));

245 if(ret == -1) {

246 LOG\_ERROR("set socket setsockopt error !");

247 close(listenFd\_);

248 return false;

249 }

250

251 ret = bind(listenFd\_, (struct sockaddr \*)&addr, sizeof(addr));

252 if(ret < 0) {

253 LOG\_ERROR("Bind Port:%d error!", port\_);

254 close(listenFd\_);

255 return false;

256 }

257

258 ret = listen(listenFd\_, 6);

259 if(ret < 0) {

260 LOG\_ERROR("Listen port:%d error!", port\_);

261 close(listenFd\_);

262 return false;

263 }

264 ret = epoller\_->AddFd(listenFd\_, listenEvent\_ | EPOLLIN);

265 if(ret == 0) {

266 LOG\_ERROR("Add listen error!");

267 close(listenFd\_);

268 return false;

269 }

270 SetFdNonblock(listenFd\_);

271 LOG\_INFO("Server port:%d", port\_);

272 return true;

273 }

274

275 int WebServer::SetFdNonblock(int fd) {

276 assert(fd > 0);

277 return fcntl(fd, F\_SETFL, fcntl(fd, F\_GETFD, 0) | O\_NONBLOCK);

278 }

279

280

server\webserver.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-17

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef WEBSERVER\_H

6 #define WEBSERVER\_H

7

8 #include <unordered\_map>

9 #include <fcntl.h> // fcntl()

10 #include <unistd.h> // close()

11 #include <assert.h>

12 #include <errno.h>

13 #include <sys/socket.h>

14 #include <netinet/in.h>

15 #include <arpa/inet.h>

16

17 #include "epoller.h"

18 #include "../log/log.h"

19 #include "../timer/heaptimer.h"

20 #include "../pool/sqlconnpool.h"

21 #include "../pool/threadpool.h"

22 #include "../pool/sqlconnRAII.h"

23 #include "../http/httpconn.h"

24

25 class WebServer {

26 public:

27 WebServer(

28 int port, int trigMode, int timeoutMS, bool OptLinger,

29 int sqlPort, const char\* sqlUser, const char\* sqlPwd,

30 const char\* dbName, int connPoolNum, int threadNum,

31 bool openLog, int logLevel, int logQueSize);

32

33 ~WebServer();

34 void Start();

35

36 private:

37 bool InitSocket\_();

38 void InitEventMode\_(int trigMode);

39 void AddClient\_(int fd, sockaddr\_in addr);

40

41 void DealListen\_();

42 void DealWrite\_(HttpConn\* client);

43 void DealRead\_(HttpConn\* client);

44

45 void SendError\_(int fd, const char\*info);

46 void ExtentTime\_(HttpConn\* client);

47 void CloseConn\_(HttpConn\* client);

48

49 void OnRead\_(HttpConn\* client);

50 void OnWrite\_(HttpConn\* client);

51 void OnProcess(HttpConn\* client);

52

53 static const int MAX\_FD = 65536;

54

55 static int SetFdNonblock(int fd);

56

57 int port\_;

58 bool openLinger\_;

59 int timeoutMS\_; /\* 毫秒MS \*/

60 bool isClose\_;

61 int listenFd\_;

62 char\* srcDir\_;

63

64 uint32\_t listenEvent\_;

65 uint32\_t connEvent\_;

66

67 std::unique\_ptr<HeapTimer> timer\_;

68 std::unique\_ptr<ThreadPool> threadpool\_;

69 std::unique\_ptr<Epoller> epoller\_;

70 std::unordered\_map<int, HttpConn> users\_;

71 };

72

73

74 #endif //WEBSERVER\_H

timer\heaptimer.cpp

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-17

3 \* @copyleft Apache 2.0

4 \*/

5 #include "heaptimer.h"

6

7 void HeapTimer::siftup\_(size\_t i) {

8 assert(i >= 0 && i < heap\_.size());

9 size\_t j = (i - 1) / 2;

10 while(j >= 0) {

11 if(heap\_[j] < heap\_[i]) { break; }

12 SwapNode\_(i, j);

13 i = j;

14 j = (i - 1) / 2;

15 }

16 }

17

18 void HeapTimer::SwapNode\_(size\_t i, size\_t j) {

19 assert(i >= 0 && i < heap\_.size());

20 assert(j >= 0 && j < heap\_.size());

21 std::swap(heap\_[i], heap\_[j]);

22 ref\_[heap\_[i].id] = i;

23 ref\_[heap\_[j].id] = j;

24 }

25

26 bool HeapTimer::siftdown\_(size\_t index, size\_t n) {

27 assert(index >= 0 && index < heap\_.size());

28 assert(n >= 0 && n <= heap\_.size());

29 size\_t i = index;

30 size\_t j = i \* 2 + 1;

31 while(j < n) {

32 if(j + 1 < n && heap\_[j + 1] < heap\_[j]) j++;

33 if(heap\_[i] < heap\_[j]) break;

34 SwapNode\_(i, j);

35 i = j;

36 j = i \* 2 + 1;

37 }

38 return i > index;

39 }

40

41 void HeapTimer::add(int id, int timeout, const TimeoutCallBack& cb) {

42 assert(id >= 0);

43 size\_t i;

44 if(ref\_.count(id) == 0) {

45 /\* 新节点：堆尾插入，调整堆 \*/

46 i = heap\_.size();

47 ref\_[id] = i;

48 heap\_.push\_back({id, Clock::now() + MS(timeout), cb});

49 siftup\_(i);

50 }

51 else {

52 /\* 已有结点：调整堆 \*/

53 i = ref\_[id];

54 heap\_[i].expires = Clock::now() + MS(timeout);

55 heap\_[i].cb = cb;

56 if(!siftdown\_(i, heap\_.size())) {

57 siftup\_(i);

58 }

59 }

60 }

61

62 void HeapTimer::doWork(int id) {

63 /\* 删除指定id结点，并触发回调函数 \*/

64 if(heap\_.empty() || ref\_.count(id) == 0) {

65 return;

66 }

67 size\_t i = ref\_[id];

68 TimerNode node = heap\_[i];

69 node.cb();

70 del\_(i);

71 }

72

73 void HeapTimer::del\_(size\_t index) {

74 /\* 删除指定位置的结点 \*/

75 assert(!heap\_.empty() && index >= 0 && index < heap\_.size());

76 /\* 将要删除的结点换到队尾，然后调整堆 \*/

77 size\_t i = index;

78 size\_t n = heap\_.size() - 1;

79 assert(i <= n);

80 if(i < n) {

81 SwapNode\_(i, n);

82 if(!siftdown\_(i, n)) {

83 siftup\_(i);

84 }

85 }

86 /\* 队尾元素删除 \*/

87 ref\_.erase(heap\_.back().id);

88 heap\_.pop\_back();

89 }

90

91 void HeapTimer::adjust(int id, int timeout) {

92 /\* 调整指定id的结点 \*/

93 assert(!heap\_.empty() && ref\_.count(id) > 0);

94 heap\_[ref\_[id]].expires = Clock::now() + MS(timeout);;

95 siftdown\_(ref\_[id], heap\_.size());

96 }

97

98 void HeapTimer::tick() {

99 /\* 清除超时结点 \*/

100 if(heap\_.empty()) {

101 return;

102 }

103 while(!heap\_.empty()) {

104 TimerNode node = heap\_.front();

105 if(std::chrono::duration\_cast<MS>(node.expires - Clock::now()).count() > 0) {

106 break;

107 }

108 node.cb();

109 pop();

110 }

111 }

112

113 void HeapTimer::pop() {

114 assert(!heap\_.empty());

115 del\_(0);

116 }

117

118 void HeapTimer::clear() {

119 ref\_.clear();

120 heap\_.clear();

121 }

122

123 int HeapTimer::GetNextTick() {

124 tick();

125 size\_t res = -1;

126 if(!heap\_.empty()) {

127 res = std::chrono::duration\_cast<MS>(heap\_.front().expires - Clock::now()).count();

128 if(res < 0) { res = 0; }

129 }

130 return res;

131 }

timer\heaptimer.h

0 /\*

1 \* @Author : mark

2 \* @Date : 2020-06-17

3 \* @copyleft Apache 2.0

4 \*/

5 #ifndef HEAP\_TIMER\_H

6 #define HEAP\_TIMER\_H

7

8 #include <queue>

9 #include <unordered\_map>

10 #include <time.h>

11 #include <algorithm>

12 #include <arpa/inet.h>

13 #include <functional>

14 #include <assert.h>

15 #include <chrono>

16 #include "../log/log.h"

17

18 typedef std::function<void()> TimeoutCallBack;

19 typedef std::chrono::high\_resolution\_clock Clock;

20 typedef std::chrono::milliseconds MS;

21 typedef Clock::time\_point TimeStamp;

22

23 struct TimerNode {

24 int id;

25 TimeStamp expires;

26 TimeoutCallBack cb;

27 bool operator<(const TimerNode& t) {

28 return expires < t.expires;

29 }

30 };

31 class HeapTimer {

32 public:

33 HeapTimer() { heap\_.reserve(64); }

34

35 ~HeapTimer() { clear(); }

36

37 void adjust(int id, int newExpires);

38

39 void add(int id, int timeOut, const TimeoutCallBack& cb);

40

41 void doWork(int id);

42

43 void clear();

44

45 void tick();

46

47 void pop();

48

49 int GetNextTick();

50

51 private:

52 void del\_(size\_t i);

53

54 void siftup\_(size\_t i);

55

56 bool siftdown\_(size\_t index, size\_t n);

57

58 void SwapNode\_(size\_t i, size\_t j);

59

60 std::vector<TimerNode> heap\_;

61

62 std::unordered\_map<int, size\_t> ref\_;

63 };

64

65 #endif //HEAP\_TIMER\_H