

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | HTTP代理服务器的设计与实现 | | | | | |
| 姓名 | 何思远 | | 院系 | 计算学部 | | |
| 班级 | 2303103 | | 学号 | 2023212224 | | |
| 任课教师 | 聂兰顺 | | 指导教师 | 聂兰顺 | | |
| 实验地点 | 格物楼213 | | 实验时间 | 2025.10.14 | | |
| 实验课表现 | 出勤、表现得分(10) |  | 实验报告  得分(40) |  | 实验总分 |  |
| 操作结果得分(50) |  |
| 教师评语 | | | | | | |
|  | | | | | | |

****

|  |
| --- |
| 实验目的： |
| 熟悉并掌握Socket网络编程的过程与技术；深入理解HTTP协议，掌握HTTP代理服务器的基本工作原理；掌握HTTP代理服务器设计与编程实现的基本技能。 |
| 实验内容： |
| (1) 设计并实现一个基本HTTP代理服务器。  (2) 设计并实现一个支持Cache功能的HTTP代理服务器。  (3) 扩展HTTP代理服务器，支持网站过滤，用户过滤，网站引导。 |
| 实验过程： |
| 一、 Socket 编程的客户端和服务器端主要步骤  服务器端主要步骤：  1. 初始化Winsock库：调用WSAStartup()加载Winsock库，确保版本兼容性  2. 创建套接字：使用socket(AF\_INET, SOCK\_STREAM, 0)创建TCP套接字  3. 绑定地址：使用bind()将套接字绑定到指定IP地址和端口  4. 监听连接：使用listen()开始监听客户端连接请求  5. 接受连接：使用accept()接受客户端连接，返回新的套接字用于通信  6. 数据收发：使用recv()和send()进行数据接收和发送  7. 关闭套接字：使用closesocket()关闭套接字  8. 清理资源：调用WSACleanup()清理Winsock资源  客户端主要步骤：  1. 初始化Winsock库：调用WSAStartup()加载Winsock库  2. 创建套接字：使用socket(AF\_INET, SOCK\_STREAM, 0)创建TCP套接字  3. 连接服务器：使用connect()连接到指定的服务器IP和端口  4. 数据收发：使用send()发送请求，recv()接收响应  5. 关闭套接字：使用closesocket()关闭套接字  6. 清理资源：调用WSACleanup()清理Winsock资源  二、HTTP代理服务器的基本原理  HTTP代理服务器位于客户端和目标Web服务器之间，充当中介角色。其基本工作原理如下：  1. 请求拦截：代理服务器监听指定端口，接收客户端的HTTP请求  2. 请求解析：解析HTTP请求头部，提取目标服务器的主机名、端口、URL等信息  3. 连接目标服务器：根据解析出的信息，建立与目标Web服务器的TCP连接  4. 请求转发：将客户端的HTTP请求转发给目标服务器  5. 响应接收：接收目标服务器返回的HTTP响应  6. 响应转发：将响应转发回客户端  代理服务器通过这种方式实现了透明的网络请求转发，同时可以对请求和响应进行各种处理，如缓存、过滤、修改等。  三、HTTP代理服务器的程序流程图            完整矢量图链接：[Click\_me](https://raw.githubusercontent.com/ponder-j/img_bed/main/HTTP-server-diagram.svg)  四、实现HTTP代理服务器的关键技术及解决方案  4.1 多线程并发处理  • 技术：使用\_beginthreadex()创建子线程，每个客户端连接由独立线程处理  • 解决方案：避免单线程阻塞，实现多用户并发访问  4.2 HTTP协议解析  • 技术：字符串解析，提取请求行和头部字段  • 解决方案：  o 使用strtok\_s()分割HTTP头部  o 解析Host字段获取目标服务器信息  o 支持GET、POST、CONNECT方法  4.3 Socket网络编程  • 技术：Winsock API，TCP套接字编程  • 解决方案：  o 设置超时时间避免连接阻塞  o 错误处理和资源清理  o 支持IPv4地址解析  4.4 缓存机制  • 技术：文件系统缓存，基于URL哈希的键值存储  • 解决方案：  o 生成缓存键：host:port + url的哈希值  o 存储Last-Modified时间用于验证  o 添加If-Modified-Since头进行缓存验证  o 304响应时返回缓存数据  4.5 过滤功能  • 技术：配置文件加载，正则表达式匹配  • 解决方案：  o 网站过滤：支持主机名和URL路径匹配  o 用户过滤：基于客户端IP地址控制  o 网站引导：透明重定向到指定服务器  4.6 数据缓冲和转发  • 技术：动态内存分配，循环接收转发  • 解决方案：  o 使用MAXSIZE缓冲区处理大数据  o 分块接收和转发HTTP响应  o 内存管理和错误处理  完整带详细注释的代码：  #include <stdio.h>  #include <Windows.h>  #include <process.h>  #include <string.h>  #include <time.h>  #include <direct.h>  #include <sys/stat.h>  #include <vector>  #include <string>  #include <map>  #pragma comment(lib,"Ws2\_32.lib")  #define MAXSIZE 65507  #define HTTP\_PORT 80  #define CACHE\_DIR ".\\cache"  // 403 阻止页面  #define BLOCKED\_PAGE\_HTML "<!DOCTYPE html><html><head><meta charset='utf-8'><title>Access Blocked</title></head><body style='font-family: Arial, sans-serif; text-align: center; padding: 50px;'><h1 style='color: #d32f2f;'>403 Forbidden</h1><p style='font-size: 18px;'>This website has been blocked by the proxy administrator.</p><hr><p style='color: #666; font-size: 14px;'>Proxy Server v3.0</p></body></html>"  // HTTP重要头部数据结构  struct HttpHeader {      char method[8];      char url[1024];      char host[1024];      int port;      char cookie[1024 \* 10];      HttpHeader() {          ZeroMemory(this, sizeof(HttpHeader));          port = 80;      }  };  // HTTP响应头部数据结构  struct HttpResponse {      int statusCode;      char lastModified[128];      char contentType[128];      int contentLength;      bool hasLastModified;      HttpResponse() {          ZeroMemory(this, sizeof(HttpResponse));          statusCode = 0;          contentLength = -1;          hasLastModified = false;      }  };  // 缓存项结构  struct CacheItem {      char url[2048];      char lastModified[128];      char filePath[512];      int dataSize;      time\_t cacheTime;  };  // 过滤规则结构  struct FilterRule {      std::string clientIP;      // 客户端IP（用户过滤）      std::string hostname;      // 主机名（网站过滤）      std::string urlPath;       // URL路径（可选，用于细粒度过滤）      bool isAllowed;            // true=允许，false=阻止      std::string redirectTo;    // 重定向目标（为空则不重定向）  };  // 全局过滤规则  std::vector<FilterRule> websiteRules;  // 网站过滤规则  std::vector<FilterRule> userRules;     // 用户过滤规则  std::map<std::string, std::string> redirectRules; // 网站引导规则  // 函数声明  BOOL InitSocket();  void ParseHttpHead(char \*buffer, HttpHeader \*httpHeader);  void ParseHttpResponse(char \*buffer, int bufferSize, HttpResponse \*httpResponse);  BOOL ConnectToServer(SOCKET \*serverSocket, char \*host, int port);  unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter);  void GenerateCacheKey(const char \*host, int port, const char \*url, char \*cacheKey);  BOOL LoadCache(const char \*cacheKey, char \*\*data, int \*dataSize, char \*lastModified);  BOOL SaveCache(const char \*cacheKey, const char \*data, int dataSize, const char \*lastModified);  void ModifyRequestWithCache(char \*request, int \*requestSize, const char \*lastModified);  void InitCacheDirectory();  // 新增：过滤和引导相关函数  void LoadFilterRules();  BOOL CheckWebsiteAccess(const char \*hostname, int port, const char \*url);  BOOL CheckUserAccess(const char \*clientIP);  BOOL GetRedirectTarget(const char \*hostname, char \*redirectHost, int \*redirectPort);  void SendBlockedResponse(SOCKET clientSocket);  void SendRedirectResponse(SOCKET clientSocket, const char \*redirectUrl);  // 代理相关参数  SOCKET ProxyServer;  sockaddr\_in ProxyServerAddr;  const int ProxyPort = 10240;  // 代理参数结构  struct ProxyParam {      SOCKET clientSocket;      SOCKET serverSocket;      char clientIP[32];  // 客户端IP地址  };  int main()  {      printf("========================================\n");      printf("   HTTP 代理服务器 v3.0 (高级版)\n");      printf("   支持: 缓存 + 过滤 + 引导\n");      printf("========================================\n");      printf("代理服务器正在启动...\n");      // 初始化缓存目录      InitCacheDirectory();      // 加载过滤规则      LoadFilterRules();      printf("初始化Socket...\n");      if (!InitSocket()) {          printf("Socket初始化失败\n");          return -1;      }      printf("代理服务器启动成功!\n");      printf("监听端口: %d\n", ProxyPort);      printf("缓存目录: %s\n", CACHE\_DIR);      printf("网站过滤规则: %d 条\n", websiteRules.size());      printf("用户过滤规则: %d 条\n", userRules.size());      printf("网站引导规则: %d 条\n", redirectRules.size());      printf("等待客户端连接...\n");      printf("========================================\n\n");      SOCKET acceptSocket = INVALID\_SOCKET;      ProxyParam \*lpProxyParam;      HANDLE hThread;      SOCKADDR\_IN clientAddr;      int clientAddrLen = sizeof(SOCKADDR\_IN);      // 代理服务器循环监听      while (true) {          acceptSocket = accept(ProxyServer, (SOCKADDR\*)&clientAddr, &clientAddrLen);          if (acceptSocket == INVALID\_SOCKET) {              printf("接受连接失败\n");              continue;          }          lpProxyParam = new ProxyParam;          if (lpProxyParam == NULL) {              closesocket(acceptSocket);              continue;          }          lpProxyParam->clientSocket = acceptSocket;          lpProxyParam->serverSocket = INVALID\_SOCKET;          strcpy\_s(lpProxyParam->clientIP, sizeof(lpProxyParam->clientIP), inet\_ntoa(clientAddr.sin\_addr));          // 创建子线程处理代理请求          hThread = (HANDLE)\_beginthreadex(NULL, 0, &ProxyThread, (LPVOID)lpProxyParam, 0, 0);          if (hThread != NULL) {              CloseHandle(hThread);          }      }      // 资源释放      closesocket(ProxyServer);      WSACleanup();      return 0;  }  // 初始化缓存目录  void InitCacheDirectory() {      struct stat info;      if (stat(CACHE\_DIR, &info) != 0) {          \_mkdir(CACHE\_DIR);          printf("[缓存] 创建缓存目录: %s\n", CACHE\_DIR);      }  }  // 加载过滤规则  void LoadFilterRules() {      FILE \*file = NULL;      char line[512];      // 加载网站过滤规则（website\_filter.txt）      fopen\_s(&file, "website\_filter.txt", "r");      if (file != NULL) {          printf("[配置] 加载网站过滤规则...\n");          while (fgets(line, sizeof(line), file)) {              // 跳过注释和空行              if (line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;              // 去除换行符              line[strcspn(line, "\r\n")] = 0;              FilterRule rule;              char action[16];              char hostname[256];              char urlPath[512] = {0};              // 解析格式: allow/deny hostname [url\_path]              int parsed = sscanf\_s(line, "%s %s %s",                  action, (unsigned)sizeof(action),                  hostname, (unsigned)sizeof(hostname),                  urlPath, (unsigned)sizeof(urlPath));              if (parsed >= 2) {                  rule.hostname = hostname;                  rule.isAllowed = (strcmp(action, "allow") == 0);                  if (parsed == 3) {                      rule.urlPath = urlPath;                      printf("[配置]   %s %s%s\n", action, hostname, urlPath);                  } else {                      rule.urlPath = "";                      printf("[配置]   %s %s\n", action, hostname);                  }                  websiteRules.push\_back(rule);              }          }          fclose(file);      } else {          printf("[配置] 未找到 website\_filter.txt，跳过网站过滤\n");      }      // 加载用户过滤规则（user\_filter.txt）      fopen\_s(&file, "user\_filter.txt", "r");      if (file != NULL) {          printf("[配置] 加载用户过滤规则...\n");          while (fgets(line, sizeof(line), file)) {              if (line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;              line[strcspn(line, "\r\n")] = 0;              FilterRule rule;              char action[16];              char ip[32];              // 解析格式: allow/deny IP              if (sscanf\_s(line, "%s %s", action, (unsigned)sizeof(action), ip, (unsigned)sizeof(ip)) == 2) {                  rule.clientIP = ip;                  rule.isAllowed = (strcmp(action, "allow") == 0);                  userRules.push\_back(rule);                  printf("[配置]   %s %s\n", action, ip);              }          }          fclose(file);      } else {          printf("[配置] 未找到 user\_filter.txt，跳过用户过滤\n");      }      // 加载网站引导规则（redirect.txt）      fopen\_s(&file, "redirect.txt", "r");      if (file != NULL) {          printf("[配置] 加载网站引导规则...\n");          while (fgets(line, sizeof(line), file)) {              if (line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;              line[strcspn(line, "\r\n")] = 0;              char source[256];              char target[256];              // 解析格式: source\_host target\_host              if (sscanf\_s(line, "%s %s", source, (unsigned)sizeof(source), target, (unsigned)sizeof(target)) == 2) {                  redirectRules[source] = target;                  printf("[配置]   %s -> %s\n", source, target);              }          }          fclose(file);      } else {          printf("[配置] 未找到 redirect.txt，跳过网站引导\n");      }  }  // 检查网站访问权限（支持URL路径过滤和端口匹配）  BOOL CheckWebsiteAccess(const char \*hostname, int port, const char \*url) {      // 如果没有规则，默认允许      if (websiteRules.empty()) return TRUE;      // 构造完整的 host:port 字符串用于匹配      char fullHost[1280];      if (port != 80) {          sprintf\_s(fullHost, sizeof(fullHost), "%s:%d", hostname, port);      } else {          strcpy\_s(fullHost, sizeof(fullHost), hostname);      }      // 遍历规则查找匹配      for (size\_t i = 0; i < websiteRules.size(); i++) {          const char \*ruleHost = websiteRules[i].hostname.c\_str();          // 检查主机名是否匹配（支持子串匹配）          // 优先匹配完整的 host:port，如果失败则尝试只匹配 hostname          bool hostMatched = false;          // 情况1: 规则包含端口 (如 "165.99.42.83:30000")          if (strchr(ruleHost, ':') != NULL) {              // 必须完整匹配 host:port              hostMatched = (strstr(fullHost, ruleHost) != NULL);          }          // 情况2: 规则只有主机名 (如 "165.99.42.83" 或 "baidu.com")          else {              // 只匹配主机名部分              hostMatched = (strstr(hostname, ruleHost) != NULL);          }          if (hostMatched) {              // 如果规则指定了URL路径，需要同时匹配URL              if (!websiteRules[i].urlPath.empty()) {                  // 检查URL路径是否匹配                  if (strstr(url, websiteRules[i].urlPath.c\_str()) != NULL) {                      return websiteRules[i].isAllowed;                  }                  // URL不匹配，继续检查下一条规则                  continue;              }              // 只匹配主机名，不限制URL              return websiteRules[i].isAllowed;          }      }      // 默认行为：如果有规则但没匹配，允许访问      return TRUE;  }  // 检查用户访问权限  BOOL CheckUserAccess(const char \*clientIP) {      // 如果没有规则，默认允许      if (userRules.empty()) return TRUE;      // 遍历规则查找匹配      for (size\_t i = 0; i < userRules.size(); i++) {          if (strcmp(clientIP, userRules[i].clientIP.c\_str()) == 0) {              return userRules[i].isAllowed;          }      }      // 默认行为：如果有规则但没匹配，允许访问      return TRUE;  }  // 获取重定向目标  BOOL GetRedirectTarget(const char \*hostname, char \*redirectHost, int \*redirectPort) {      std::string host(hostname);      if (redirectRules.find(host) != redirectRules.end()) {          std::string target = redirectRules[host];          // 解析目标（支持 host:port 格式）          size\_t colonPos = target.find(':');          if (colonPos != std::string::npos) {              strcpy\_s(redirectHost, 1024, target.substr(0, colonPos).c\_str());              \*redirectPort = atoi(target.substr(colonPos + 1).c\_str());          } else {              strcpy\_s(redirectHost, 1024, target.c\_str());              \*redirectPort = 80;          }          return TRUE;      }      return FALSE;  }  // 发送阻止响应  void SendBlockedResponse(SOCKET clientSocket) {      char response[1024];      sprintf\_s(response, sizeof(response),          "HTTP/1.1 403 Forbidden\r\n"          "Content-Type: text/html; charset=utf-8\r\n"          "Content-Length: %d\r\n"          "Connection: close\r\n"          "\r\n"          "%s",          strlen(BLOCKED\_PAGE\_HTML),          BLOCKED\_PAGE\_HTML);      send(clientSocket, response, strlen(response), 0);  }  // 函数：初始化套接字  BOOL InitSocket() {      WORD wVersionRequested;      WSADATA wsaData;      int err;      wVersionRequested = MAKEWORD(2, 2);      err = WSAStartup(wVersionRequested, &wsaData);      if (err != 0) {          printf("加载winsock失败,错误代码为: %d\n", WSAGetLastError());          return FALSE;      }      if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {          printf("不能找到正确的winsock版本\n");          WSACleanup();          return FALSE;      }      ProxyServer = socket(AF\_INET, SOCK\_STREAM, 0);      if (INVALID\_SOCKET == ProxyServer) {          printf("创建套接字失败,错误代码为:%d\n", WSAGetLastError());          return FALSE;      }      ProxyServerAddr.sin\_family = AF\_INET;      ProxyServerAddr.sin\_port = htons(ProxyPort);      ProxyServerAddr.sin\_addr.S\_un.S\_addr = INADDR\_ANY;      if (bind(ProxyServer, (SOCKADDR\*)&ProxyServerAddr, sizeof(SOCKADDR)) == SOCKET\_ERROR) {          printf("绑定套接字失败\n");          closesocket(ProxyServer);          return FALSE;      }      if (listen(ProxyServer, SOMAXCONN) == SOCKET\_ERROR) {          printf("监听端口%d失败\n", ProxyPort);          closesocket(ProxyServer);          return FALSE;      }      return TRUE;  }  // 子线程执行的代理逻辑（核心逻辑 - 带过滤和引导）  unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter) {      char \*Buffer = new char[MAXSIZE];      char \*ResponseBuffer = new char[MAXSIZE \* 10];      ZeroMemory(Buffer, MAXSIZE);      ZeroMemory(ResponseBuffer, MAXSIZE \* 10);      SOCKADDR\_IN clientAddr;      int length = sizeof(SOCKADDR\_IN);      int recvSize;      int ret;      int totalBytes = 0;      int responseBufferSize = 0;      bool headerParsed = false;      HttpHeader\* httpHeader = NULL;      HttpResponse\* httpResponse = NULL;      char \*CacheBuffer = NULL;      char cacheKey[512];      char \*cachedData = NULL;      int cachedDataSize = 0;      char lastModified[128] = {0};      ProxyParam\* param = (ProxyParam\*)lpParameter;      // 获取客户端地址信息      getpeername(param->clientSocket, (SOCKADDR\*)&clientAddr, &length);      printf("[新连接] 客户端: %s:%d\n", inet\_ntoa(clientAddr.sin\_addr), ntohs(clientAddr.sin\_port));      // ========== 用户过滤检查 ==========      if (!CheckUserAccess(param->clientIP)) {          printf("[用户过滤] 阻止用户访问: %s\n", param->clientIP);          SendBlockedResponse(param->clientSocket);          goto error;      }      // 接收客户端请求      recvSize = recv(param->clientSocket, Buffer, MAXSIZE, 0);      if (recvSize <= 0) {          printf("[错误] 接收客户端请求失败\n");          goto error;      }      printf("[接收] 收到 %d 字节的HTTP请求\n", recvSize);      // 解析HTTP头部      httpHeader = new HttpHeader();      CacheBuffer = new char[recvSize + 1];      ZeroMemory(CacheBuffer, recvSize + 1);      memcpy(CacheBuffer, Buffer, recvSize);      ParseHttpHead(CacheBuffer, httpHeader);      delete[] CacheBuffer;      CacheBuffer = NULL;      if (strlen(httpHeader->host) == 0) {          printf("[错误] 无法解析目标主机\n");          delete httpHeader;          httpHeader = NULL;          goto error;      }      // 忽略 CONNECT 请求（HTTPS）      if (strcmp(httpHeader->method, "CONNECT") == 0) {          delete httpHeader;          httpHeader = NULL;          goto error;      }      printf("[解析] 方法: %s, 主机: %s:%d, URL: %s\n",             httpHeader->method, httpHeader->host, httpHeader->port, httpHeader->url);      // ========== 网站过滤检查 ==========      if (!CheckWebsiteAccess(httpHeader->host, httpHeader->port, httpHeader->url)) {          printf("[网站过滤] 阻止访问网站: %s:%d%s\n", httpHeader->host, httpHeader->port, httpHeader->url);          SendBlockedResponse(param->clientSocket);          delete httpHeader;          httpHeader = NULL;          goto error;      }      // ========== 网站引导（钓鱼）检查 ==========      char redirectHost[1024];      int redirectPort;      if (GetRedirectTarget(httpHeader->host, redirectHost, &redirectPort)) {          printf("[网站引导] %s -> %s:%d\n", httpHeader->host, redirectHost, redirectPort);          // 修改目标主机          strcpy\_s(httpHeader->host, sizeof(httpHeader->host), redirectHost);          httpHeader->port = redirectPort;      }      // ========== 缓存处理（仅GET请求）==========      if (strcmp(httpHeader->method, "GET") == 0) {          GenerateCacheKey(httpHeader->host, httpHeader->port, httpHeader->url, cacheKey);          if (LoadCache(cacheKey, &cachedData, &cachedDataSize, lastModified)) {              printf("[缓存] 找到缓存，大小: %d 字节, Last-Modified: %s\n",                     cachedDataSize, lastModified);              if (strlen(lastModified) > 0) {                  ModifyRequestWithCache(Buffer, &recvSize, lastModified);                  printf("[缓存] 添加 If-Modified-Since: %s\n", lastModified);              }          } else {              printf("[缓存] 未找到缓存\n");          }      }      // 连接目标服务器      if (!ConnectToServer(&param->serverSocket, httpHeader->host, httpHeader->port)) {          printf("[错误] 连接目标服务器 %s:%d 失败\n", httpHeader->host, httpHeader->port);          if (cachedData != NULL) {              printf("[缓存] 服务器连接失败，返回缓存数据\n");              send(param->clientSocket, cachedData, cachedDataSize, 0);              delete[] cachedData;              cachedData = NULL;          }          delete httpHeader;          httpHeader = NULL;          goto error;      }      printf("[连接] 成功连接到目标服务器: %s:%d\n", httpHeader->host, httpHeader->port);      // 转发客户端请求到目标服务器      ret = send(param->serverSocket, Buffer, recvSize, 0);      if (ret == SOCKET\_ERROR) {          printf("[错误] 转发请求失败\n");          delete httpHeader;          httpHeader = NULL;          goto error;      }      printf("[转发] 已转发请求到目标服务器 (%d 字节)\n", ret);      // 循环接收目标服务器响应      totalBytes = 0;      responseBufferSize = 0;      headerParsed = false;      httpResponse = new HttpResponse();      while (true) {          ZeroMemory(Buffer, MAXSIZE);          recvSize = recv(param->serverSocket, Buffer, MAXSIZE, 0);          if (recvSize <= 0) {              break;          }          totalBytes += recvSize;          if (responseBufferSize + recvSize < MAXSIZE \* 10) {              memcpy(ResponseBuffer + responseBufferSize, Buffer, recvSize);              responseBufferSize += recvSize;          }          if (!headerParsed && responseBufferSize > 0) {              ParseHttpResponse(ResponseBuffer, responseBufferSize, httpResponse);              headerParsed = true;              printf("[响应] 状态码: %d\n", httpResponse->statusCode);              if (httpResponse->hasLastModified) {                  printf("[响应] Last-Modified: %s\n", httpResponse->lastModified);              }              // 如果是304响应且有缓存，直接发送缓存，不再转发服务器响应              if (httpResponse->statusCode == 304 && cachedData != NULL && cachedDataSize > 0) {                  printf("[缓存] 服务器返回304，使用缓存替代响应\n");                  ret = send(param->clientSocket, cachedData, cachedDataSize, 0);                  if (ret == SOCKET\_ERROR) {                      printf("[错误] 发送缓存数据失败\n");                  } else {                      printf("[缓存] 成功发送缓存数据到客户端，大小: %d 字节\n", cachedDataSize);                  }                  // 跳出循环，不再转发服务器的304响应                  break;              }          }          ret = send(param->clientSocket, Buffer, recvSize, 0);          if (ret == SOCKET\_ERROR) {              printf("[错误] 转发响应到客户端失败\n");              break;          }      }      printf("[完成] 已转发响应到客户端 (总计 %d 字节)\n", totalBytes);      // 缓存处理      if (strcmp(httpHeader->method, "GET") == 0) {          if (httpResponse->statusCode == 304) {              // 304已在上面处理，这里只记录日志              printf("[缓存] 缓存仍然有效\n");          } else if (httpResponse->statusCode == 200 && responseBufferSize > 0) {              if (httpResponse->hasLastModified) {                  SaveCache(cacheKey, ResponseBuffer, responseBufferSize, httpResponse->lastModified);                  printf("[缓存] 已保存缓存，大小: %d 字节\n", responseBufferSize);              } else {                  printf("[缓存] 响应无Last-Modified头，不缓存\n");              }          }      }      // 清理      if (cachedData) {          delete[] cachedData;          cachedData = NULL;      }      if (httpHeader) {          delete httpHeader;          httpHeader = NULL;      }      if (httpResponse) {          delete httpResponse;          httpResponse = NULL;      }  error:      if (httpHeader) {          delete httpHeader;      }      if (httpResponse) {          delete httpResponse;      }      if (cachedData) {          delete[] cachedData;      }      delete[] Buffer;      delete[] ResponseBuffer;      if (param->clientSocket != INVALID\_SOCKET) {          closesocket(param->clientSocket);      }      if (param->serverSocket != INVALID\_SOCKET) {          closesocket(param->serverSocket);      }      delete param;      \_endthreadex(0);      return 0;  }  // 解析HTTP请求头部  void ParseHttpHead(char \*buffer, HttpHeader \*httpHeader) {      char \*p;      char \*ptr;      const char \*delim = "\r\n";      p = strtok\_s(buffer, delim, &ptr);      if (p == NULL) return;      if (strncmp(p, "GET", 3) == 0) {          memcpy(httpHeader->method, "GET", 3);          char \*url\_start = p + 4;          char \*url\_end = strstr(url\_start, " HTTP");          if (url\_end != NULL) {              int url\_len = url\_end - url\_start;              if (url\_len < 1024) {                  memcpy(httpHeader->url, url\_start, url\_len);              }          }      }      else if (strncmp(p, "POST", 4) == 0) {          memcpy(httpHeader->method, "POST", 4);          char \*url\_start = p + 5;          char \*url\_end = strstr(url\_start, " HTTP");          if (url\_end != NULL) {              int url\_len = url\_end - url\_start;              if (url\_len < 1024) {                  memcpy(httpHeader->url, url\_start, url\_len);              }          }      }      else if (strncmp(p, "CONNECT", 7) == 0) {          memcpy(httpHeader->method, "CONNECT", 7);      }      p = strtok\_s(NULL, delim, &ptr);      while (p) {          if (strncmp(p, "Host:", 5) == 0) {              char \*host\_start = p + 6;              while (\*host\_start == ' ') host\_start++;              char \*colon = strchr(host\_start, ':');              if (colon != NULL) {                  int host\_len = colon - host\_start;                  if (host\_len < 1024) {                      memcpy(httpHeader->host, host\_start, host\_len);                      httpHeader->host[host\_len] = '\0';                  }                  httpHeader->port = atoi(colon + 1);              } else {                  strcpy\_s(httpHeader->host, sizeof(httpHeader->host), host\_start);                  httpHeader->port = 80;              }          }          else if (strncmp(p, "Cookie:", 7) == 0) {              char \*cookie\_start = p + 8;              while (\*cookie\_start == ' ') cookie\_start++;              strcpy\_s(httpHeader->cookie, sizeof(httpHeader->cookie), cookie\_start);          }          p = strtok\_s(NULL, delim, &ptr);      }  }  // 解析HTTP响应头部  void ParseHttpResponse(char \*buffer, int bufferSize, HttpResponse \*httpResponse) {      char \*headerEnd = strstr(buffer, "\r\n\r\n");      if (headerEnd == NULL) return;      char \*tempBuffer = new char[headerEnd - buffer + 1];      memcpy(tempBuffer, buffer, headerEnd - buffer);      tempBuffer[headerEnd - buffer] = '\0';      char \*p;      char \*ptr;      const char \*delim = "\r\n";      p = strtok\_s(tempBuffer, delim, &ptr);      if (p != NULL) {          char \*statusStart = strchr(p, ' ');          if (statusStart != NULL) {              httpResponse->statusCode = atoi(statusStart + 1);          }      }      p = strtok\_s(NULL, delim, &ptr);      while (p) {          if (strncmp(p, "Last-Modified:", 14) == 0) {              char \*value = p + 15;              while (\*value == ' ') value++;              strcpy\_s(httpResponse->lastModified, sizeof(httpResponse->lastModified), value);              httpResponse->hasLastModified = true;          }          else if (strncmp(p, "Content-Type:", 13) == 0) {              char \*value = p + 14;              while (\*value == ' ') value++;              strcpy\_s(httpResponse->contentType, sizeof(httpResponse->contentType), value);          }          else if (strncmp(p, "Content-Length:", 15) == 0) {              char \*value = p + 16;              while (\*value == ' ') value++;              httpResponse->contentLength = atoi(value);          }          p = strtok\_s(NULL, delim, &ptr);      }      delete[] tempBuffer;  }  // 连接目标服务器  BOOL ConnectToServer(SOCKET \*serverSocket, char \*host, int port) {      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_port = htons(port);      HOSTENT \*hostent = gethostbyname(host);      if (!hostent) {          return FALSE;      }      in\_addr Inaddr = \*((in\_addr\*)\*hostent->h\_addr\_list);      serverAddr.sin\_addr.s\_addr = inet\_addr(inet\_ntoa(Inaddr));      \*serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);      if (\*serverSocket == INVALID\_SOCKET) {          return FALSE;      }      int timeout = 5000;      setsockopt(\*serverSocket, SOL\_SOCKET, SO\_RCVTIMEO, (char\*)&timeout, sizeof(timeout));      setsockopt(\*serverSocket, SOL\_SOCKET, SO\_SNDTIMEO, (char\*)&timeout, sizeof(timeout));      if (connect(\*serverSocket, (SOCKADDR \*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {          closesocket(\*serverSocket);          \*serverSocket = INVALID\_SOCKET;          return FALSE;      }      return TRUE;  }  // 生成缓存键  void GenerateCacheKey(const char \*host, int port, const char \*url, char \*cacheKey) {      char fullUrl[2048];      sprintf\_s(fullUrl, sizeof(fullUrl), "%s:%d%s", host, port, url);      unsigned int hash = 0;      for (int i = 0; fullUrl[i] != '\0'; i++) {          hash = hash \* 31 + fullUrl[i];      }      sprintf\_s(cacheKey, 512, "%s\\%u.cache", CACHE\_DIR, hash);  }  // 加载缓存  BOOL LoadCache(const char \*cacheKey, char \*\*data, int \*dataSize, char \*lastModified) {      FILE \*file = NULL;      fopen\_s(&file, cacheKey, "rb");      if (file == NULL) {          return FALSE;      }      char lm[128] = {0};      fread(lm, 1, 128, file);      strcpy\_s(lastModified, 128, lm);      fseek(file, 0, SEEK\_END);      long fileSize = ftell(file);      fseek(file, 128, SEEK\_SET);      int contentSize = fileSize - 128;      \*data = new char[contentSize];      \*dataSize = fread(\*data, 1, contentSize, file);      fclose(file);      return TRUE;  }  // 保存缓存  BOOL SaveCache(const char \*cacheKey, const char \*data, int dataSize, const char \*lastModified) {      FILE \*file = NULL;      fopen\_s(&file, cacheKey, "wb");      if (file == NULL) {          return FALSE;      }      char lm[128] = {0};      strcpy\_s(lm, sizeof(lm), lastModified);      fwrite(lm, 1, 128, file);      fwrite(data, 1, dataSize, file);      fclose(file);      return TRUE;  }  // 修改请求添加If-Modified-Since头  void ModifyRequestWithCache(char \*request, int \*requestSize, const char \*lastModified) {      char \*headerEnd = strstr(request, "\r\n\r\n");      if (headerEnd == NULL) return;      if (strstr(request, "If-Modified-Since:") != NULL) {          return;      }      char ifModifiedSince[256];      sprintf\_s(ifModifiedSince, sizeof(ifModifiedSince), "If-Modified-Since: %s\r\n", lastModified);      int headerLength = headerEnd - request;      int newSize = headerLength + strlen(ifModifiedSince) + 4;      char \*newRequest = new char[MAXSIZE];      memcpy(newRequest, request, headerLength + 2);      strcpy\_s(newRequest + headerLength + 2, MAXSIZE - headerLength - 2, ifModifiedSince);      strcat\_s(newRequest, MAXSIZE, "\r\n");      int bodyLength = \*requestSize - (headerEnd - request) - 4;      if (bodyLength > 0) {          memcpy(newRequest + newSize, headerEnd + 4, bodyLength);          newSize += bodyLength;      }      memcpy(request, newRequest, newSize);      \*requestSize = newSize;      delete[] newRequest;  } |
| 实验结果： |
| HTTP代理服务器实验验证过程以及实验结果  在远程服务器上启动一个 Alpine Nginx+php7.3 环境，在 /var/www/html 中传入 Test\_Server 文件夹中的代码，在 Firefox 浏览器中打开 ZeroOmega 配置本地 http 代理服务器进行本实验的验证：  1. 基础功能      可以看到能够正确显示页面内容，代表基础代理功能有效。  2. 缓存功能测试    可以看到页面加载过后再刷新，能够正确载入缓存，缓存功能有效。  3. 扩展功能测试  3.1 网站过滤  在 website\_filter.txt 中添加需要过滤的网站或 url 信息：    可以看到网页被正确过滤：    3.2 用户过滤  在 user\_filter.txt 中添加需要过滤的 ip：    可以看到本机已经无法再正常访问所有网站：    3.3 网站引导  如图所示，该网页原来指向的 url 是 http://baidu.com/phishing-demo.html：    实际上 Baidu 没有这个网页，但是我们在 redirect.txt 中配置了 baidu.com 165.99.42.83:30000：    因此，当用户点击这个链接的时候，会重定向到http:// 165.99.42.83:30000/phishing-demo.html（浏览器网址栏不会体现这一点）： |
| 问题讨论： |
| 测试过程中遇到一个问题：为什么用户过滤中填真实 ip 无法成功过滤？（即配置 deny 111.42.148.173 我们仍能正常使用代理服务器）    答：因为该 http 代理服务器配置在本地而不是公网，我们浏览器向它发出请求时该服务器接收到的也是一个本地的 ip，所以应该配置 127.0.0.1 而不是 111.42.148.173。如果这个代理服务器运行在公网，那么就应该配置 deny 111.42.148.173. |
| 心得体会： |
| 熟悉并掌握了Socket网络编程的过程与技术，深入理解了HTTP协议，能够正确处理报文，掌握了HTTP代理服务器的基本工作原理，对所谓“翻墙”技术有了更加深刻的理解。 |