

**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 | 常添 | | 院系 | 计算学部 | | |
| 班级 | 2203102 | | 学号 | 2022111699 | | |
| 任课教师 | 聂兰顺 | | 指导教师 |  | | |
| 实验地点 | G001 | | 实验时间 | 2024.10.10 | | |
| 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： |
| 本次实验的主要目的：  1. 熟悉并掌握 Socket 网络编程的过程与技术：通过构建 HTTP 代理服务器，学习如何使用 Socket 进行网络编程，了解 Socket 编程中的基本操作与流程。  2. 深入理解 HTTP 协议：在实验中，通过分析和处理 HTTP 请求和响应报文，增强对 HTTP 协议工作机制的理解，包括请求报文的结构、方法以及响应报文的内容。  3. 掌握 HTTP 代理服务器的工作原理：通过设计并实现 HTTP 代理服务器，学习代理服务器如何在客户端和目标服务器之间转发请求和响应，了解代理服务器的基本功能和数据流向。  4. 掌握 HTTP 代理服务器的设计与编程实现的基本技能：学习如何设计、实现并扩展一个 HTTP 代理服务器，使其具备基础代理功能以及缓存、过滤和引导等高级功能。 |
| 实验内容： |
| 概述本次实验的主要内容，包含的实验项等：  本次实验包含以下主要内容，通过实现不同功能模块，逐步扩展 HTTP 代理服务器的功能：  1. 基本 HTTP 代理服务器的设计与实现：  - 在指定端口（例如 8080）上接收来自客户端的 HTTP 请求。  - 从请求报文中解析出目标 URL 地址，并根据该地址访问对应的 HTTP 服务器（即原服务器）。  - 接收原服务器的响应报文并将其转发给客户端，从而实现代理功能。  2. 支持缓存功能的 HTTP 代理服务器：  - 在基本代理功能的基础上，增加缓存机制。将原服务器的响应对象进行缓存，以便后续请求可以直接从缓存中获取数据。  - 通过在请求报文中添加 If-Modified-Since头部，向原服务器确认缓存对象是否是最新版本。如果未修改，则直接从缓存中返回响应；否则，获取并更新缓存。  3. 扩展功能模块的实现：  - 网站过滤：配置代理服务器以允许或阻止访问某些特定网站，通过检查 URL 来决定是否转发请求。  - 用户过滤：根据用户的身份信息来限制其访问外部网站的权限。例如，可以通过 IP 地址或用户账户进行过滤。  - 网站引导（钓鱼网站）：代理服务器可将某些网站的访问请求引导至一个预设的模拟网站（钓鱼网站），实现访问重定向功能。 |
| 实验过程： |
| 以文字描述、实验结果截图等形式阐述实验过程，必要时可附相应的代码截图或以附件形式提交。  HTTP 代理服务器源代码：  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <winsock2.h>  #include <process.h>  #include <time.h>  #pragma comment(lib, "Ws2\_32.lib")  #define MAXSIZE 65507 // 最大数据包大小  #define HTTP\_PORT 80 // 目标服务器的 HTTP 端口  #define PROXY\_PORT 8080 // 代理服务器监听的端口  #define CACHE\_SIZE 10 // 缓存容量  // 定义 HTTP 请求头结构体  struct HttpHeader {  char method[8]; // 请求方法（GET, POST 等）  char url[1024]; // 请求的 URL  char host[1024]; // 请求的主机名  char cookie[1024 \* 10]; // 请求中的 Cookie 信息  };  // 定义缓存条目结构体  struct CacheEntry {  char url[1024]; // 缓存的 URL  char content[MAXSIZE]; // 缓存的内容  time\_t last\_modified; // 缓存的最后修改时间  };  // 全局缓存和黑名单  struct CacheEntry cache[CACHE\_SIZE]; // 缓存数组  char\* blacklist[] = { "example.com" }; // 禁止访问的网站  char\* redirect\_sites[] = { "www.cctv.cn" }; // 重定向的网站  char redirect\_url[] = "today.hit.edu.cn"; // 重定向目标 URL  // 用户 IP 黑名单  char\* user\_blacklist[] = { "127.0.1.1", "192.168.1.100" }; // 禁止访问的用户 IP  // 函数声明  BOOL InitSocket(); // 初始化代理服务器的套接字  void InitializeHttpHeader(struct HttpHeader\* httpHeader); // 初始化 HTTP 头  void ParseHttpHead(char\* buffer, struct HttpHeader\* httpHeader); // 解析 HTTP 头  BOOL ConnectToServer(SOCKET\* serverSocket, char\* host, struct HttpHeader\* httpHeader, struct CacheEntry\* cacheEntry); // 连接到目标服务器  unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter); // 处理客户端请求的线程  int check\_cache(char\* url, char\* content); // 检查缓存是否命中  void add\_to\_cache(char\* url, char\* content); // 将内容添加到缓存  BOOL is\_blacklisted(char\* host); // 检查主机是否在黑名单中  BOOL should\_redirect(char\* host); // 检查主机是否需要重定向  BOOL is\_user\_blacklisted(char\* ip); // 检查用户 IP 是否在黑名单中  // 全局变量  SOCKET ProxyServer;  struct sockaddr\_in ProxyServerAddr;  // 初始化代理服务器的套接字，设置为监听模式  BOOL InitSocket() {  WSADATA wsaData;  if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {  printf("Winsock 初始化失败\n");  return FALSE;  }  ProxyServer = socket(AF\_INET, SOCK\_STREAM, 0);  if (ProxyServer == INVALID\_SOCKET) {  printf("创建 Socket 失败\n");  return FALSE;  }  ProxyServerAddr.sin\_family = AF\_INET;  ProxyServerAddr.sin\_addr.s\_addr = INADDR\_ANY;  ProxyServerAddr.sin\_port = htons(PROXY\_PORT);  if (bind(ProxyServer, (struct sockaddr\*)&ProxyServerAddr, sizeof(ProxyServerAddr)) == SOCKET\_ERROR) {  printf("绑定 Socket 失败\n");  return FALSE;  }  if (listen(ProxyServer, SOMAXCONN) == SOCKET\_ERROR) {  printf("监听端口失败\n");  return FALSE;  }  printf("代理服务器正在监听端口 %d\n", PROXY\_PORT);  return TRUE;  }  // 初始化 HTTP 头部结构体  void InitializeHttpHeader(struct HttpHeader\* httpHeader) {  memset(httpHeader, 0, sizeof(struct HttpHeader));  }  // 解析 HTTP 请求头，将方法、URL、主机等信息存储到 HttpHeader 结构体中  void ParseHttpHead(char\* buffer, struct HttpHeader\* httpHeader) {  char\* p = strtok(buffer, "\r\n");  if (p) {  // 提取请求方法（GET, POST 等）  if (strncmp(p, "GET", 3) == 0) {  strcpy(httpHeader->method, "GET");  sscanf(p, "GET %s HTTP", httpHeader->url);  }  else if (strncmp(p, "POST", 4) == 0) {  strcpy(httpHeader->method, "POST");  sscanf(p, "POST %s HTTP", httpHeader->url);  }  }  // 提取主机名和 Cookie 信息  while (p) {  if (strncmp(p, "Host:", 5) == 0) {  sscanf(p, "Host: %s", httpHeader->host);  }  else if (strncmp(p, "Cookie:", 7) == 0) {  strcpy(httpHeader->cookie, p + 8);  }  p = strtok(NULL, "\r\n");  }  }  // 连接到目标服务器，如果缓存命中则添加 If-Modified-Since 头部进行请求  BOOL ConnectToServer(SOCKET\* serverSocket, char\* host, struct HttpHeader\* httpHeader, struct CacheEntry\* cacheEntry) {  struct sockaddr\_in serverAddr;  struct hostent\* server;  server = gethostbyname(host); // 根据主机名获取 IP 地址  if (server == NULL) {  printf("主机地址解析失败\n");  return FALSE;  }  \*serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);  if (\*serverSocket == INVALID\_SOCKET) {  printf("连接目标服务器失败\n");  return FALSE;  }  serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_port = htons(HTTP\_PORT);  memcpy(&serverAddr.sin\_addr.s\_addr, server->h\_addr\_list[0], server->h\_length);  // 尝试连接目标服务器  if (connect(\*serverSocket, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {  printf("连接目标服务器失败\n");  return FALSE;  }  printf("成功连接到目标服务器 %s\n", host);  // 构建 HTTP 请求  char request[MAXSIZE];  if (cacheEntry && cacheEntry->last\_modified) {  struct tm\* lastModTime = gmtime(&cacheEntry->last\_modified);  char lastModifiedStr[128];  strftime(lastModifiedStr, sizeof(lastModifiedStr), "%a, %d %b %Y %H:%M:%S GMT", lastModTime);  // 添加 If-Modified-Since 头部  snprintf(request, sizeof(request),  "%s %s HTTP/1.1\r\nHost: %s\r\nIf-Modified-Since: %s\r\nConnection: close\r\nUser-Agent: YourUserAgent\r\nAccept: \*/\*\r\n\r\n",  httpHeader->method, httpHeader->url, host, lastModifiedStr);  printf("Sending request with If-Modified-Since header:\n%s\n", lastModifiedStr);  }  else {  snprintf(request, sizeof(request),  "%s %s HTTP/1.1\r\nHost: %s\r\nConnection: close\r\nUser-Agent: YourUserAgent\r\nAccept: \*/\*\r\n\r\n",  httpHeader->method, httpHeader->url, host);  }  // 发送请求到目标服务器  printf("Sending request:\n%s\n", request);  if (send(\*serverSocket, request, strlen(request), 0) == SOCKET\_ERROR) {  printf("发送请求失败\n");  return FALSE;  }  return TRUE;  }  // 线程处理客户端请求，包括过滤、重定向和缓存检查  unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter) {  SOCKET clientSocket = ((SOCKET\*)lpParameter)[0];  SOCKET serverSocket;  char buffer[MAXSIZE];  int recvSize;  struct HttpHeader httpHeader;  InitializeHttpHeader(&httpHeader);  // 接收客户端请求  recvSize = recv(clientSocket, buffer, MAXSIZE, 0);  if (recvSize <= 0) {  goto error;  }  ParseHttpHead(buffer, &httpHeader);  // 获取客户端 IP 地址  struct sockaddr\_in clientAddr;  int clientAddrLen = sizeof(clientAddr);  getpeername(clientSocket, (struct sockaddr\*)&clientAddr, &clientAddrLen);  char\* clientIP = inet\_ntoa(clientAddr.sin\_addr);  // 检查用户 IP 是否在黑名单中  if (is\_user\_blacklisted(clientIP)) {  printf("访问被禁止的用户 IP：%s\n", clientIP);  send(clientSocket, "HTTP/1.1 403 Forbidden\r\n\r\nBlocked User", 39, 0);  goto error;  }  // 检查请求的主机是否在黑名单中  if (is\_blacklisted(httpHeader.host)) {  printf("访问被阻止：%s\n", httpHeader.host);  send(clientSocket, "HTTP/1.1 403 Forbidden\r\n\r\nBlocked Website", 39, 0);  goto error;  }  // 检查是否需要重定向到其他主机  if (should\_redirect(httpHeader.host)) {  strcpy(httpHeader.host, redirect\_url);  printf("重定向到：%s\n", redirect\_url);  }  // 缓存检查：如果命中缓存则直接返回缓存内容  if (check\_cache(httpHeader.url, buffer)) {  printf("缓存命中：%s\n", httpHeader.url);  send(clientSocket, buffer, strlen(buffer), 0);  goto error;  }  else {  printf("缓存未命中：%s\n", httpHeader.url);  }  // 连接到目标服务器  if (!ConnectToServer(&serverSocket, httpHeader.host, &httpHeader, cache)) {  goto error;  }  // 转发请求并添加到缓存  char request[MAXSIZE];  sprintf(request, "%s %s HTTP/1.1\r\nHost: %s\r\nConnection: close\r\nUser-Agent: YourUserAgent\r\nAccept: \*/\*\r\n\r\n",  httpHeader.method, httpHeader.url, httpHeader.host);  send(serverSocket, request, strlen(request), 0);  // 接收目标服务器的响应并转发给客户端  while ((recvSize = recv(serverSocket, buffer, MAXSIZE, 0)) > 0) {  send(clientSocket, buffer, recvSize, 0);  add\_to\_cache(httpHeader.url, buffer);  }  error:  closesocket(clientSocket);  closesocket(serverSocket);  \_endthreadex(0);  return 0;  }  // 主函数，初始化服务器并进入主循环等待客户端连接  int main() {  if (!InitSocket()) {  printf("服务器初始化失败\n");  return -1;  }  while (TRUE) {  SOCKET clientSocket;  struct sockaddr\_in clientAddr;  int clientAddrLen = sizeof(clientAddr);  clientSocket = accept(ProxyServer, (struct sockaddr\*)&clientAddr, &clientAddrLen);  if (clientSocket == INVALID\_SOCKET) {  printf("接受客户端连接失败\n");  continue;  }  printf("客户端已连接：%s\n", inet\_ntoa(clientAddr.sin\_addr));  \_beginthreadex(NULL, 0, ProxyThread, (void\*)&clientSocket, 0, NULL);  }  closesocket(ProxyServer);  WSACleanup();  return 0;  }  // 检查缓存是否存在指定 URL 的内容  int check\_cache(char\* url, char\* content) {  for (int i = 0; i < CACHE\_SIZE; i++) {  if (strcmp(cache[i].url, url) == 0) {  strcpy(content, cache[i].content);  return 1;  }  }  return 0;  }  // 添加内容到缓存中，环形缓冲机制  void add\_to\_cache(char\* url, char\* content) {  static int index = 0;  strcpy(cache[index].url, url);  strcpy(cache[index].content, content);  cache[index].last\_modified = time(NULL);  printf("已缓存 URL：%s，缓存索引：%d\n", url, index);  index = (index + 1) % CACHE\_SIZE;  }  // 检查主机是否在黑名单中  BOOL is\_blacklisted(char\* host) {  for (int i = 0; i < sizeof(blacklist) / sizeof(blacklist[0]); i++) {  if (strstr(host, blacklist[i]) != NULL) {  return TRUE;  }  }  return FALSE;  }  // 检查主机是否需要重定向  BOOL should\_redirect(char\* host) {  for (int i = 0; i < sizeof(redirect\_sites) / sizeof(redirect\_sites[0]); i++) {  if (strstr(host, redirect\_sites[i]) != NULL) {  return TRUE;  }  }  return FALSE;  }  // 检查用户 IP 是否在黑名单中  BOOL is\_user\_blacklisted(char\* ip) {  for (int i = 0; i < sizeof(user\_blacklist) / sizeof(user\_blacklist[0]); i++) {  if (strcmp(ip, user\_blacklist[i]) == 0) {  return TRUE;  }  }  return FALSE;  }  实验过程：  1. 环境搭建：  - 安装开发环境，使用 Windows 平台和 C++ 进行开发，并确保 Winsock 库已正确安装和配置。  - 熟悉 HTTP 协议的请求和响应报文格式，以便在后续实现中解析请求和处理响应。  2. 设计架构：  - 功能设计：代理服务器需具备接收客户端请求、解析并转发至目标服务器、接收并转发响应、实现缓存、用户和网站过滤等功能。  - 数据结构设计：为处理 HTTP 报文的头部信息、缓存条目等数据，设计适当的结构体以便高效存储和操作。    3. 实现 Socket 编程：  - Winsock 初始化：调用 WSAStartup 函数初始化 Winsock 库，以支持后续的网络操作。  - 创建和绑定套接字：通过 socket() 创建 TCP 套接字，并将其绑定到指定的端口（如 8080），使用 listen() 函数开始监听客户端连接请求。    4. 处理客户端连接：  - 接受连接并多线程处理：在主线程中循环接受客户端的连接请求，每个请求由单独的线程处理，以实现并发操作。  - 解析请求：在子线程中，接收并解析客户端的 HTTP 请求，提取 HTTP 方法、目标 URL、主机名等信息。    5. 请求处理：  - 用户和网站过滤：根据预设的黑名单，判断是否允许当前用户访问目标网站，并输出相应的过滤日志。  - 缓存管理：首先检查缓存中是否已有该请求的响应，如果命中缓存，则直接返回缓存的数据，否则转发请求至目标服务器。    6. 响应处理：  - 接收与转发响应：从目标服务器接收响应数据，转发给客户端的同时将其缓存，以便下次相同请求时直接返回缓存结果。  - 处理错误情况：若遇到连接失败、请求超时等异常，则返回适当的错误信息。    7. 资源清理：  - 关闭连接：完成请求后，关闭与客户端和目标服务器的连接，释放所占用的资源。  - 清理缓存：定期清理缓存中过期的条目，以节省内存并避免缓存污染。  代码的思路与逻辑：  1. 初始化和配置：  - Winsock 初始化：使用 WSAStartup() 函数初始化 Winsock，创建 TCP 套接字，并绑定到端口（如 8080），然后使用 listen() 开始监听。  - 多线程处理连接：为每个客户端连接创建线程，通过循环不断接收连接请求。  2. 接收和处理客户端请求：  - 提取并解析 HTTP 请求：从请求报文中解析出 HTTP 方法、目标 URL、主机名等关键信息。  - 过滤功能实现：根据主机名和客户端 IP 地址，检查是否符合过滤规则，以决定是否允许请求继续。  3. 缓存管理：  - 缓存查询与更新：在处理请求之前检查缓存是否命中，命中则直接返回缓存内容，未命中则从目标服务器获取响应并缓存。    4. 目标服务器连接：  - 解析与连接目标服务器：将主机名解析为 IP 地址，使用 `connect()` 建立与目标服务器的 TCP 连接，转发客户端请求。    5. 异常处理与资源清理：  - 错误处理与关闭连接：处理可能出现的错误情况，适时关闭连接，并使用 closesocket() 函数释放资源。 |
| 实验结果： |
| 采用演示截图、文字说明等方式，给出本次实验的实验结果。  首先修改系统（Windows）的网络代理设置，开启手动代理，并将端口修改为实验中的服务器端口8080：    在终端编译并运行，测试客户端的连接情况：    Invoke-WebRequest -Uri "http://example.com" -Proxy "http://127.0.0.1:8080"：这条请求中的URL是设置的黑名单，所以连接失败，以下是结果截图    Invoke-WebRequest -Uri "http://www.cctv.cn" -Proxy "http://127.0.0.1:8080"：这条请求实际上体现了http代理服务器的钓鱼功能，实际上连接到了today.hit.edu.cn      Invoke-WebRequest -Uri "http://jwes.hit.edu.cn/" -Proxy "http://127.0.0.1:8080"：这条请求连接到了jwes.hit.edu.cn，能缓存原服务器响应的对象，首次连接所以显示缓存miss，后续已经缓存了该URL      Invoke-WebRequest -Uri "http://httpbin.org" -Proxy "http://127.0.0.1:8080"：这里就是最基础的请求，服务器控制台的输出显示了使用"If-Modified-Since"头的修改      测试被封禁的用户ip为127.0.0.1，可以看到该用户被服务器阻止了连接，原本能访问的网站目前无法访问： |
| 问题讨论： |
| (1) Socket 编程的客户端和服务器端步骤  - 服务器端：  1. 创建套接字：用于监听客户端连接的服务器端套接字。  2. 绑定地址和端口：将服务器的 IP 和端口绑定到该套接字。  3. 监听连接请求：设置套接字为监听状态，等待客户端连接。  4. 接受客户端连接：处理来自客户端的连接请求。  5. 数据通信：与客户端之间进行数据收发。  6. 关闭连接：结束与客户端的连接，释放资源。  - 客户端：  1. 创建套接字：用于连接服务器的客户端套接字。  2. 连接服务器：将客户端套接字连接到指定服务器地址和端口。  3. 数据通信：与服务器之间进行数据收发。  4. 关闭连接：通信结束后关闭套接字，释放资源。  (2) HTTP 代理服务器的基本原理  1. 客户端请求代理服务器：客户端（如浏览器）将 HTTP 请求发送给代理服务器，请求中包含目标资源地址。    2. 代理服务器处理请求：代理服务器解析请求并决定是否处理，包括：  - 缓存处理：若请求的资源在缓存中，则直接返回缓存资源。  - 过滤和安全检查：阻止不符合策略的请求（如 URL 黑名单）。    3. 代理服务器请求目标服务器：若资源不在缓存中或通过过滤检查，则代理服务器向目标服务器发出新的请求。  4. 目标服务器响应代理服务器：目标服务器返回请求的资源数据（如网页内容）给代理服务器。  5. 代理服务器返回响应给客户端：代理服务器将目标服务器的响应转发给客户端，必要时缓存该响应以优化后续请求。  Caching proxy server state flow diagram (3) HTTP 代理服务器程序流程图  Caching proxy server states  **Figure 1: Caching proxy server states Figure 2: Caching proxy server state flow diagram**  (4) 实现 HTTP 代理服务器的关键技术和解决方案  1. 套接字编程：  - 关键技术：通过 Socket 建立与客户端和目标服务器的双向通信。  - 解决方案：代理服务器监听客户端请求，将请求转发到目标服务器并返回响应。  2. 请求与响应解析：  - 关键技术：解析 HTTP 请求和响应（如方法、URL、状态码）。  - 解决方案：解析客户端请求，提取目标服务器信息，并正确解析目标服务器的响应。  3. 连接管理：  - 关键技术：支持并发处理多个客户端请求，管理连接生命周期。  - 解决方案：使用多线程或异步 I/O 技术处理并发请求，或利用连接池技术提高性能。  4. 缓存机制：  - 关键技术：缓存常见请求的响应，减少直接访问目标服务器。  - 解决方案：存储静态资源或高频内容，控制缓存的有效性和更新策略。  5. 过滤与控制：  - 关键技术：基于策略管理请求和响应，支持 URL 黑名单等功能。  - 解决方案：实现安全策略控制，过滤不符合策略的请求内容。  6. 日志记录与监控：  - 关键技术：记录请求和响应数据，用于分析和审计。  - 解决方案：记录 IP 地址、请求 URL、响应时间等，并支持日志轮转。  7. 错误处理与超时机制：  - 关键技术：应对各种网络错误和超时情况。  - 解决方案：处理目标服务器不可达等情况，向客户端返回错误响应并支持超时控制和重试。 |
| 心得体会： |
| 在本次实验中，通过设计并实现一个 HTTP 代理服务器，我获得了以下体会和收获：  1. Socket 网络编程的掌握  实验让我深入理解了套接字编程的核心步骤，包括套接字的创建、绑定、监听、数据传输等操作。通过服务器和客户端间的通信实现，我加深了对网络编程基础的掌握，尤其是如何使用 Socket 与多个客户端和目标服务器进行连接与数据交互。  2. 对 HTTP 协议的深入理解  通过代理服务器的实现，我更好地理解了 HTTP 协议的工作原理。实验中解析并转发 HTTP 请求和响应的过程，使我学会了如何处理 HTTP 请求头和响应头，理解了不同的 HTTP 方法（如 GET 和 POST）的应用，以及请求从客户端发起到服务器返回的整个生命周期。  3. 缓存机制的应用  在实验中实现了支持缓存功能的代理服务器，我掌握了如何将服务器响应数据缓存下来，并利用 HTTP 的 If-Modified-Since 头部检查缓存的有效性。缓存机制不仅提升了代理服务器的性能，也有效减少了与目标服务器的交互，节省了网络资源。  4. 网站和用户过滤功能的实现  我实现了对特定网站和用户的访问控制，深入理解了如何基于请求内容（如 URL 或 IP 地址）进行过滤。这一功能使我了解了代理服务器在网络安全中的作用，并学会了如何通过访问控制来保护网络资源免受不合规访问。  5. 网站引导（重定向）的实现  实验中还涉及了网站引导功能，使我掌握了如何拦截用户对特定站点的请求并将其重定向到另一个站点。虽然此功能模拟了钓鱼攻击的实现，但我也看到了它在合法网络测试和流量管理中的应用潜力。  6. 网络安全的进一步理解  网站和用户过滤、重定向等扩展功能的实现让我意识到代理服务器在网络安全控制中的重要性。通过实验，我学会了如何利用代理服务器对网络访问进行控制，为将来设计更复杂的网络安全方案奠定了基础。  通过本次实验，我不仅巩固了对 Socket 编程和 HTTP 协议的理解，也对代理服务器在网络安全和流量管理中的应用有了更清晰的认识。 |