哈爾濱工業大學
Harbin Institute of Technology

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	杨明达		院系	未来技术学院	
班级	22R0311		学号	2022110829	
任课教师	李全龙		指导教师	李全龙、郭勇	
实验地点	G001		实验时间	周六 5、6 节	
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					

计算机科学与技术学院 SINCE 1956...
School of Computer Science and Technology

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。 要求在指定端口（例如 8080） 接收来自客户的 HTTP 请求并且根据其中的 URL 地址访问该地址所指向的 HTTP 服务器（原服务器）， 接收 HTTP 服务器的响应报文，并将响应报文转发给对应的客户进行浏览。

（2）设计并实现一个支持 Cache 功能的 HTTP 代理服务器。 要求能缓存原服务器响应的对象，并能够通过修改请求报文（添加 if-modified-since 头行），向原服务器确认缓存对象是否是最新版本。

（3）扩展 HTTP 代理服务器，支持如下功能：

a) 网站过滤：允许/不允许访问某些网站；

b) 用户过滤：支持/不支持某些用户访问外部网站；

c) 网站引导：将用户对某个网站的访问引导至一个模拟网站（钓鱼）。

实验过程：

一、实验大致步骤

（1）浏览器使用代理

为了使浏览器访问网址时通过代理服务器，必须进行相关设置，以 Google Chrome 浏览器设置为例：打开浏览器→设置→系统→打开你计算机的代理设置，具体过程如图 1-1 所示。

图 1-1 浏览器的代理服务器设置

（上左：浏览器设置，上右：计算机代理设置，下：浏览器系统）

（2）多线程使用

使用函数_beginthreadex 创建子线程，使用函数_endthreadex 结束线程。

（3）开发 HTTP 代理服务器并测试验证

编写 HTTP 代理服务器代码，部署运行，然后可以自己测试验证相关功能。

二、客户端和服务器在编程中的实现主要步骤

（1）客户端

a) 根据目的服务器 IP 地址和端口号创建套接字，连接服务器
b) 发送请求报文
c) 接收返回报文
d) 关闭链接服务器端

（2）服务器端

a) 创建套接字，绑定 IP 地址和端口号，监听端口
b) 从连接队列中取出一个连接请求，三次握手创建连接
c) 接受请求报文
d) 发送响应报文
e) 关闭当前连接，继续监听端口

三、HTTP 代理服务器基本原理及参考内容

代理服务器，俗称“翻墙软件”，允许一个网络终端（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接。如图 1-2 所示，为普通 Web 应用通信方式与采用代理服务器的通信方式的对比。

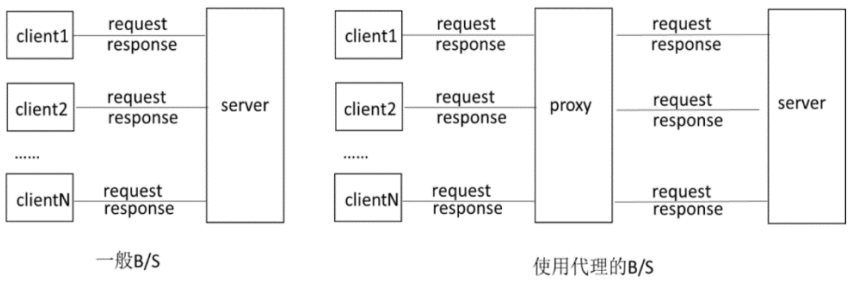

图 1-2 Web 应用通信方式对比

代理服务器在指定端口（例如 8080） 监听浏览器的访问请求（需要在客户端浏览器进行相应的设置）， 接收到浏览器对远程网站的浏览请求时，代理服务器开始在代理服务器的缓存中检索 URL 对应的对象（网页、图像等对象），找到对象文件后， 提取该对象文件的最新被修改时间； 代理服务器程序在客户的请求报文首部插入<If-Modified-Since: 对象文件的最新被修改时间>，并向原 Web 服务器转发修改后的请求报文。 如果代理服务器没有该对象的缓存，则会直接向原服务器转发请求报文， 并将原服务器返回的响应直接转发给客户端，同时将对象缓存到代理服务器中。 代理服务器程序会根据缓存的时间、大小和提取记录等对缓存进行清理。

HTTP 代理服务器程序流程如下图所示。

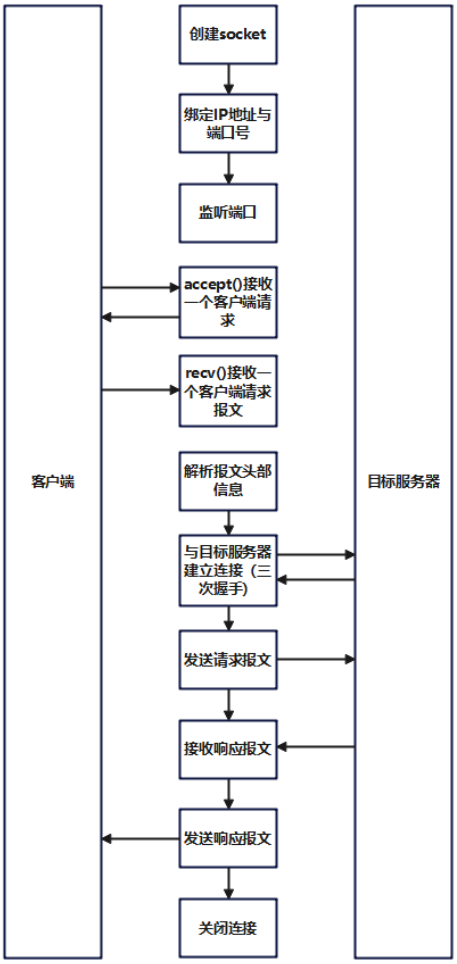

图 1-3 HTTP 代理服务器程序流程

四、HTTP 代理服务器的设计与实现中主要技术

（1）必做任务

InitSocket 函数用来初始化套接字，有 socket 创建，bind 绑定，listen 监听三部分组成。其中，Socket()根据选用的服务器建立套接字。Bind()将一个本地地址与一个创建好的套接字绑定，运行该函数代表代理服务器端的套接字可以开始准备工作。Listen()用来监听端口的连接请求，在运行函数之后套接字进入监听模式。

Accept()用来在一个套接字处接受/提取一个连接请求，在该函数运行后表示代理服务器和客户端之间的连接建立起来。

ProxyThread 函数用来执行线程，有 recv，parse，connect，send，recv，send 六部分组成。其中 recv()用来接受客户端报文数据。parse()用来解析 http 头。connect()用来根据主机创建目

标套接字并连接，即建立 socket 连线。send()用来发送报文数据，将客户端发送的 http 数据报文转发给目标服务器。recv()用来接受目标返回数据。send()用来将目标服务器的数据转发给客户端。

closeSocket()用来关闭套接字。

有了上述功能函数的定义，代理服务器的具体功能流程为：

a) 初始化套接字，使用 socket 函数创建一个套接字，使用 bind 函数将套接字和代理服务器的本地地址绑定，即将代理服务器的 IP 地址设置为 127.0.0.1，端口设置为 10240。套接字初始化完成后，使用 listen 函数将套接字进入监听环节，等待客户端发送连接请求。

b) 使用 accept 函数对每个到来的请求进行接受，对每一个连接都创建一个单独的进程来提供服务。

c) 使用如 recv 函数接受客户端发送的 HTTP 报文，解析目的服务器的相关信息，使用 connect 函数与目的服务器建立连接，使用 send 函数将 HTTP 请求报文发送给目的服务器。

d) 等待目的服务器返回响应报文，使用 send 函数将响应报文发送给客户端。

e) 关闭线程，清理缓存。

（2）选做任务——cache 功能

a) 创建一个 cache 数组存储用户访问过的服务器中的数据

b) 当客户第一次访问服务器中的数据时，代理服务器将返回的报文的内容缓存并保存到本地。

c) 如果 cache 数组中没有查找到代理服务器已经缓存的服务器的内容，则重新执行 b 操作。如果已经有缓存，要判断缓存是否为目的服务器上最新的内容。于是，解析客户端发送的 HTTP 请求报文，添加 if-modified-since，与缓存最后修改时间相加，判断目的服务器在改时间间隔中是否更改，如果代理服务器缓存是最新的，则目的服务器返回 304 状态码，将缓存返回给客户端；如果不是最新的，则缓存更新，目的服务器返回 200 状态码，将更新后的响应内容发送给客户端。

（3）选做任务——网站过滤

大致思想是将请求报文头部中的 host 与被禁止网站进行比较，如果出现相同的表示访问的网站被禁止访问。具体流程为在代理服务器段对于客户发送的请求报文进行解析，提取出其要访问的 url，如果和已知需要过滤的网站一致，直接关闭套接字，结束这次连接。

（4）选做任务——用户过滤

大致思想是主函数中，当建立起客户端和代理服务器的连接时，得到客户端的 IP 地址，与被禁用户 IP 比较，如果相同，则跳过。具体流程为在建立连接的时候比对请求连接的客户端 IP 地址是否和限定的 IP 地址一致，如果一致允许继续执行程序；如果不一致直接断开连接。

（5）选做任务——网站引导

大致思想是将请求报文头部中的 url 与被引导网站进行比较，如果相同则改为引导网站的 url。具体流程为检测客户端发送的 HTTP 请求报文，如果发现访问的网址是要被钓鱼的网址，则将该网址引导到其他网站（钓鱼目的网址），通过更改 HTTP 头部字段的 url 和主机名来实现。

实验结果：

（1）基础配置

为了使浏览器访问网址时通过代理服务器，必须进行相关设置，以 Google Chrome 浏览器设置为例：打开浏览器→设置→系统→打开你计算机的代理设置，具体过程见“实现过程”图 1-1 所示。在代理设置界面，手动设置代理，点击设置，更改代理 IP 地址为 127.0.0.1，端口号为 10240。

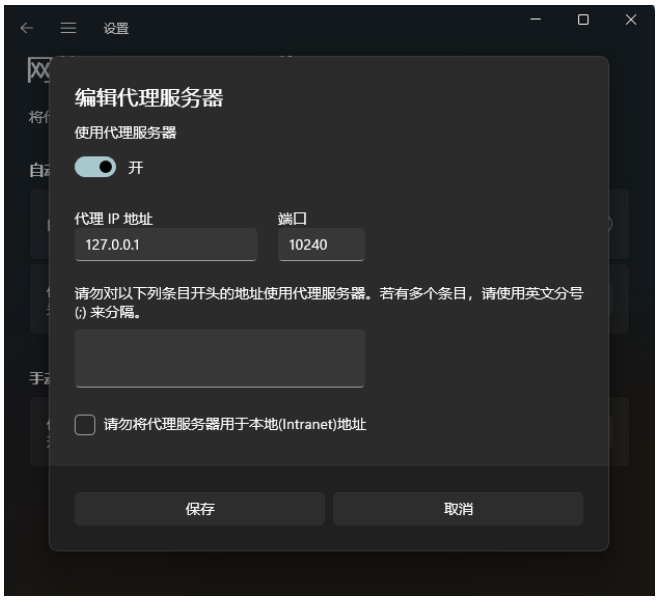

图 2-1 设置代理

（2）必做任务

button 选择 false 进入必做环节，运行程序，代理服务器开始运行。

图 2-2 代理服务器开始运行

在浏览器中访问 http://jwts.hit.edu.cn，终端显示结果，证明连接成功。

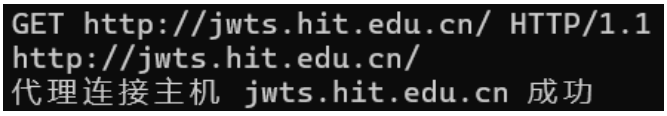

图 2-3 访问网站（终端显示）

观察浏览器中的界面，证明访问成功。

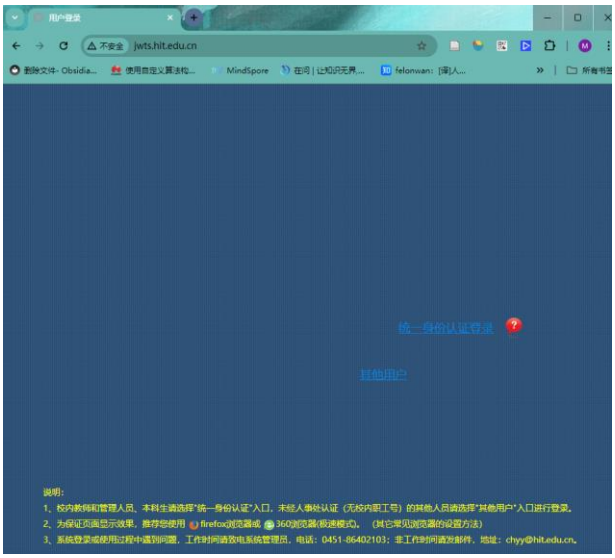

图 2-4 访问网站（浏览器显示）

（3）选做任务——cache

运行程序，代理服务器开始运行。

```
代理服务器正在启动
初始化...
代理服务器正在运行，监听端口 10240
```

图 2-5 代理服务器开始运行

在浏览器中第一次访问 http://jwts.hit.edu.cn，终端显示结果，证明连接成功。

```
GET http://jwts.hit.edu.cn/ HTTP/1.1
http://jwts.hit.edu.cn/
代理连接主机 jwts.hit.edu.cn 成功
```

图 2-6 第一次访问网站

在浏览器中第二次访问 http://jwts.hit.edu.cn，终端显示结果，观察目的服务器返回报文，访问的网站中的内容和缓存的内容是相同的，从而成功返回缓存索引，证明连接成功。

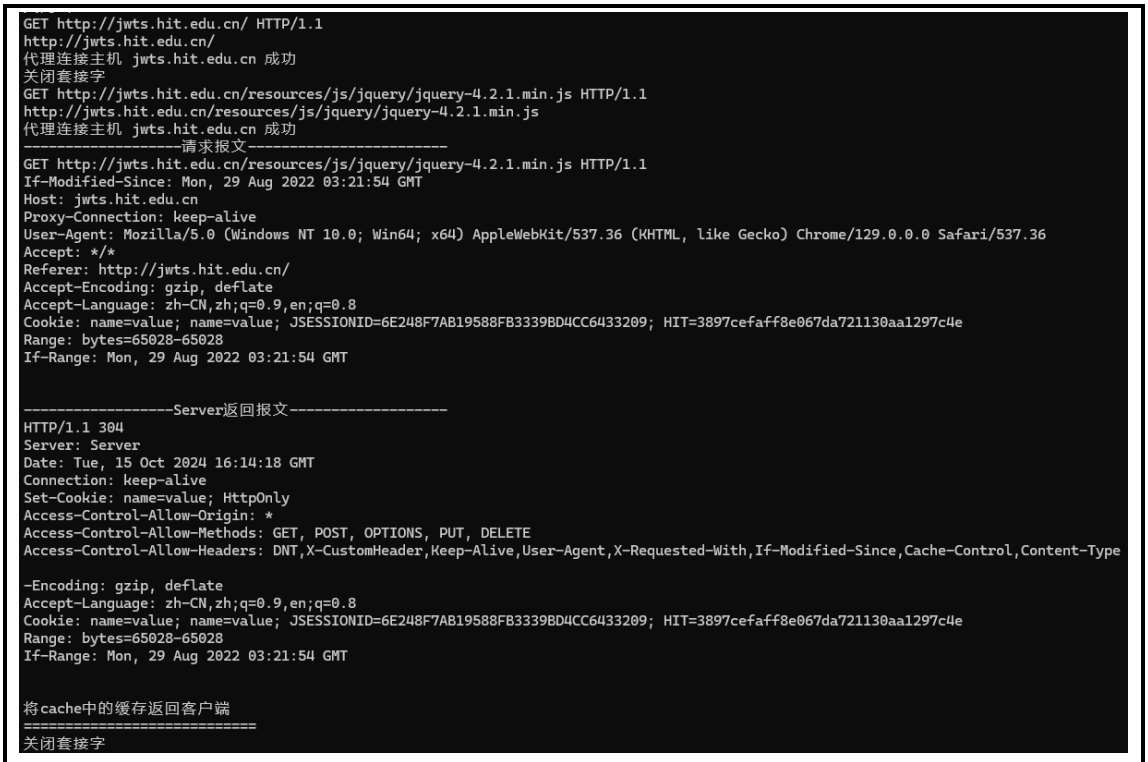

图 2-7 第二次访问网站

（4）选做任务——网站过滤

button 选择 true 进入选做环节，将过滤网站设置为 http://www.hit.edu.cn，运行程序，代理服务器开始运行。

图 2-8 代理服务器开始运行

在浏览器中访问 http://www.hit.edu.cn，终端显示结果，证明网址已经过滤，无法访问。

图 2-9 访问过滤网站（终端显示）

观察浏览器中的界面，证明网址已经过滤，无法访问。

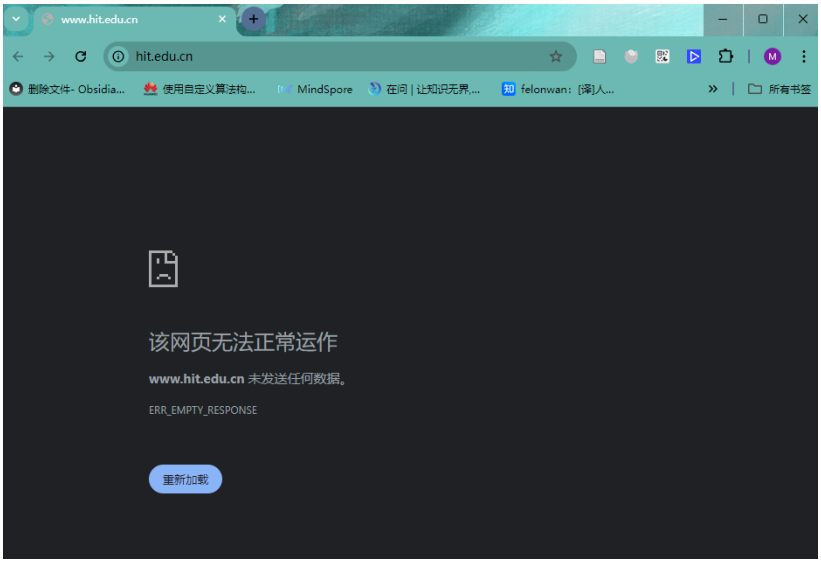

图 2-10 访问过滤网站（浏览器显示）

（5）选做任务——用户过滤

button 选择 true 进入选做环节，将过滤用户改为 127.0.0.1，运行程序，代理服务器开始运行，但是不断输出“该用户访问受限”，证明用户成功被过滤，无法访问任何网站。

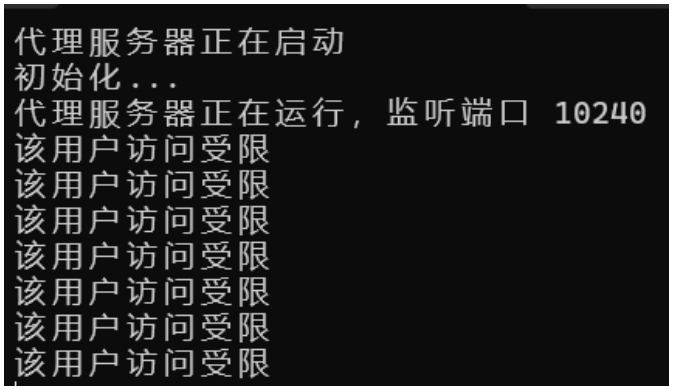

图 2-11 过滤用户

（6）选做任务——网站引导

button 选择 true 进入选做环节，将钓鱼网站原网址设置为 http://today.hit.edu.cn，钓鱼网站目标网址为 http://jwes.hit.edu.cn，运行程序，代理服务器开始运行。

图 2-12 代理服务器开始运行

在浏览器中访问 http://today.hit.edu.cn，终端显示结果，证明从 http://today.hit.edu.cn 跳转到 http://jwes.hit.edu.cn，实际访问的是 http://jwes.hit.edu.cn。

图 2-13 网站引导（终端显示）

观察浏览器中的界面，证明证明从 http://today.hit.edu.cn 跳转到 http://jwes.hit.edu.cn，实际访问的是 http://jwes.hit.edu.cn。

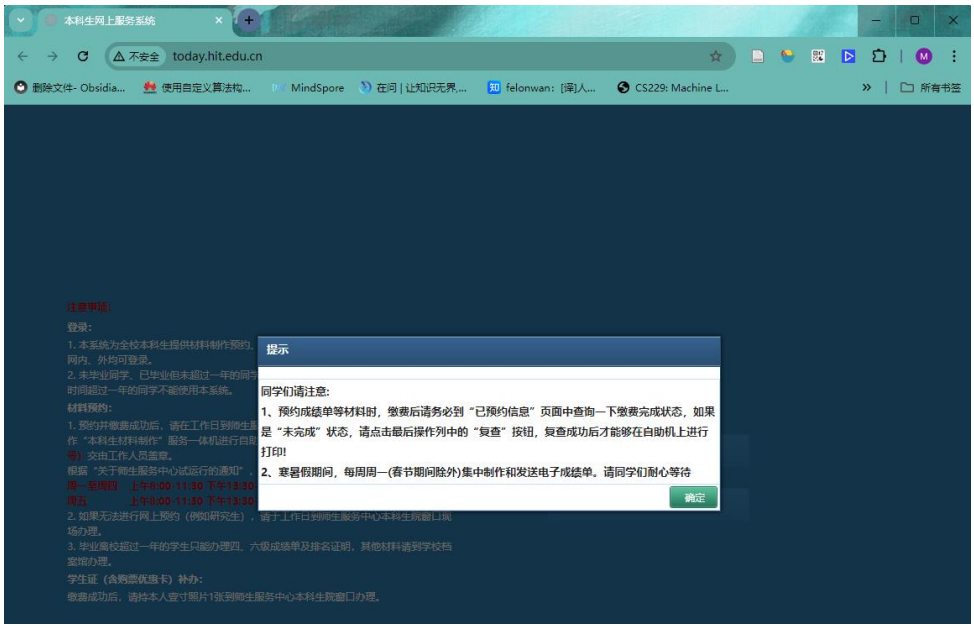

图 2-14 网站引导（浏览器显示）

问题讨论：

（1）#include "stdafx.h"代码报错，错误的原因是没有这个头文件，解决方式为直接去掉。该文件相关信息是Windows和MFC的include文件都非常大，即使有一个快速的处理程序，编译程序也要花费相当长的时间来完成工作。由于每个.CPP文件都包含相同的include文件，为每个.CPP文件都重复处理这些文件耗时很长，因此stdafx.h可以视为一个包含所有需要include的头文件，在编译开始的时候先把这些内容编译好，之后在编译每一个cpp文件的时候就阅读编译好的结果即可。

（2）用Visual Studio运行时，不支持 int _tmain(int argc, _TCHAR* argv[]) 的写法，改成 int main(int argc, char* argv[]) 后能运行成功。具体原因源自 main()和_tmain(int argc, _TCHAR* argv[]) 的详细区别和 c/c++ int _tmain(int argc, _TCHAR* argv[])

（3） goto 语句问题，代码一直报 goto 语句的问题，具体原因是 g++编译goto语句出现：[error:jump to label XXX]，也就是 goto 语句之后不能再定义新的变量。

（4）使用代理服务器后加载网页可能会有图片等条目无法显示的问题，因为在本实验中只进行了http协议的代理，并没有进行https的代理，两者相较而言少了一个connect方式，所以未能成功加载的条目是依据https协议进行的，才出现了这种问题。

心得体会：

通过本次实验，我实现了HTTP 代理服务器的设计与实现，进而熟悉并掌握 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，清楚客户端和服务器之间Socket通信过程，同时了解了钓鱼网站，禁止用户，禁止网站以及 Cache等的原理，有很大的收获。

具体实现代码

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>
#include <string.h>
#pragma comment(lib, "Ws2_32.lib")
#define MAXSIZE 65507  // 发送数据报文的最大长度
#define HTTP_PORT 80   // http 服务器端口
#define DATELENGTH 50  // 时间字节数
#define CACHE_NUM 50   // 定义最大缓存数量
// Http 重要头部数据
struct HttpHeader {
    char method[4];  // POST 或者 GET，注意有些为 CONNECT，本实验暂不考虑
    char url[1024];          // 请求的 url
    char host[1024];         // 目标主机
    char cookie[1024 * 10];  // cookie
    HttpHeader() { ZeroMemory(this, sizeof(HttpHeader)); }
};
// 因为不做外部存储，所以为了节省空间，cache 存储的时候
// 去掉 Http 头部信息中的 cookie
struct cacheHttpHead {
    char method[4];  // POST 或者 GET，注意有些为 CONNECT，本实验暂不考虑
    char url[1024];   // 请求的 url
    char host[1024];  // 目标主机
    cacheHttpHead() {
        ZeroMemory(this, sizeof(cacheHttpHead));
    } // 构造函数，将结构体内存清零，确保初始化为空。
};
// 代理服务器缓存技术
struct CACHE {
    cacheHttpHead httpHead;
    char buffer[MAXSIZE];   // 储存报文返回内容
    char date[DATELENGTH];  // 缓存内容的最后修改时间
    CACHE() {
        ZeroMemory(this->buffer, MAXSIZE);
        ZeroMemory(this->date, DATELENGTH);
    } // 构造函数，将结构体内存清零，确保初始化为空。
};
CACHE cache[CACHE_NUM];  // 缓存地址
int cache_index = 0;     // 记录当前应该将缓存放在哪个位置

```

```
41.BOOL InitSocket();  // 声明初始化套接字（socket）函数，返回值为布尔类型（成功或失败）。
42.// 声明 HTTP 请求头解析函数，将 buffer 中的内容解析到`HttpHeader`结构体中。
43.void ParseHttpHead(char* buffer, HttpHeader* httpHeader);
44.// 声明连接到目标服务器的函数，返回布尔值，指示连接是否成功。
45.BOOL ConnectToServer(SOCKET* serverSocket, char* host);
46.// 声明代理服务器线程函数，用于处理客户端请求。线程入口函数采用`__stdcall`调用约定。
47.unsigned int __stdcall ProxyThread(LPVOID lpParameter);
48.// 寻找缓存中是否存在，如果存在返回 index，不存在返回-1
49.int isInCache(CACHE* cache, HttpHeader httpHeader);
50.// 声明一个函数，用于比较两个 HTTP 报文是否相同，主要用于判断缓存是否匹配请求。
51.BOOL httpEqual(cacheHttpHead http1, HttpHeader http2);
52.// 声明一个函数，用于修改 HTTP 报文中的内容，如时间字段等。
53.void changeHTTP(char* buffer, char* date);
54.// 代理相关参数
55.// 定义代理服务器的套接字（Socket）。
56.SOCKET ProxyServer;
57.// 定义代理服务器的地址结构体，用于存储 IP 地址和端口号信息。
58.sockaddr_in ProxyServerAddr;
59.// 定义代理服务器监听的端口号为 10240。
60.const int ProxyPort = 10240;
61.// 由于新的连接都使用新线程进行处理，对线程的频繁的创建和销毁特别浪费资源
62.// 可以使用线程池技术提高服务器效率
63.// const int ProxyThreadMaxNum = 20;
64.// HANDLE ProxyThreadHandle[ProxyThreadMaxNum] = {0};
65.// DWORD ProxyThreadDW[ProxyThreadMaxNum] = {0};
66.// 定义一个结构体，用于传递客户端和服务器的套接字信息。
67.struct ProxyParam {
68.    SOCKET clientSocket;  // 客户端套接字。
69.    SOCKET serverSocket;  // 服务器套接字。
70.};
71.// --------------------选做功能参数定义--------------------
72.bool button = true;  // 取 true 的时候表示开始运行选做功能
73.// 禁止访问网站
74.char* invalid_website[10] = { (char*)"http://www.hit.edu.cn" };
75.const int invalid_website_num = 1;  // 有多少个禁止网站
76.// 钓鱼网站
77.char* fishing_src = (char*)"http://today.hit.edu.cn";  // 钓鱼网站原网址
```

```
78. char* fishing_dest = (char*)"http://jwes.hit.edu.cn";  // 钓鱼网站目标网址
79. char* fishing_dest_host = (char*)"jwes.hit.edu.cn";  // 钓鱼目的地址主机名
80. // 限制访问用户
81. char* restrict_host[10] = { (char*)"127.0.0.0" };
82. int main(int argc, char* argv[]) {
83.     printf("代理服务器正在启动\n");  // 输出启动信息。
84.     printf("初始化...\n");            // 输出初始化信息
85.     // 调用`InitSocket()`初始化套接字，如果失败则返回错误
86.     if (!InitSocket()) {
87.         printf("socket 初始化失败\n");
88.         return -1;
89.     }
90.     // 输出代理服务器运行和端口号信息。
91.     printf("代理服务器正在运行，监听端口 %d\n", ProxyPort);
92.     // 定义`acceptSocket`，表示客户端连接的套接字，初始值为无效套接字。
93.     SOCKET acceptSocket = INVALID_SOCKET;
94.     // 定义指向`ProxyParam`结构体的指针，用于传递客户端和服务器套接字信息
95.     ProxyParam* lpProxyParam;
96.     // 定义线程句柄，用于创建新的线程
97.     HANDLE hThread;
98.     // 定义线程 ID 变量（虽然未使用）
99.     DWORD dwThreadID;
100.     // 定义`addr_in`结构体，用于存储客户端的地址信息
101.     sockaddr_in addr_in;
102.     // 初始化地址长度变量`addr_len`
103.     int addr_len = sizeof(SOCKADDR);
104.     // 代理服务器不断监听
105.     while (true) {
106.         // 调用`accept()`函数等待客户端连接，成功后返回客户端的套接字
107.         acceptSocket = accept(ProxyServer, (SOCKADDR*)&addr_in, &(addr_len));
108.         // 为`ProxyParam`结构体分配内存
109.         lpProxyParam = new ProxyParam;
110.         // 如果分配失败，则跳过本次循环
111.         if (lpProxyParam == NULL) {
112.             continue;
113.         }
114.         // 受限用户,与列表中匹配上的都无法访问
```

```
115.         if (!strcmp(restrict_host[0], inet_ntoa(addr_in.sin_addr
     )) &&
116.             button)  // 注意比较之前将网络二进制的数字转换成网络地
     址
117.         {
118.             printf("该用户访问受限\n");
119.             continue;
120.         }
121.         // 将客户端的套接字存入`lpProxyParam`结构体
122.         lpProxyParam->clientSocket = acceptSocket;
123.         // 创建新线程调用`ProxyThread`函数，处理客户端请求
124.         hThread = (HANDLE)_beginthreadex(NULL, 0, &ProxyThread,
     
125.             (LPVOID)lpProxyParam, 0, 0);
126.         // 关闭线程句柄，避免资源泄露
127.         CloseHandle(hThread);
128.         // 休眠 200 毫秒，以防频繁创建线程
129.         Sleep(200);
130.     }
131.     // 关闭代理服务器的套接字
132.     closesocket(ProxyServer);
133.     // 释放 Winsock 资源
134.     WSACleanup();
135.     // 返回 0，表示程序正常结束
136.     return 0;
137. }
138. //************************************
139. // Method: InitSocket
140. // FullName: InitSocket
141. // Access: public
142. // Returns: BOOL
143. // Qualifier: 初始化套接字
144. //************************************
145. BOOL InitSocket() {
146.     // 加载套接字库（必须）
147.     // 定义请求的 Winsock 版本号
148.     WORD wVersionRequested;
149.     // 用于存储 Winsock 初始化信息的数据结构
150.     WSADATA wsaData;
151.     // 套接字加载时错误提示
152.     int err;
153.     // 版本 2.2
154.     wVersionRequested = MAKEWORD(2, 2);
155.     // 加载 dll 文件 Scoket 库
```

```
156.     err = WSAStartup(wVersionRequested, &wsaData);
157.     // 如果加载失败，输出错误信息并返回`FALSE`
158.     if (err != 0) {
159.         // 找不到 winsock.dll
160.         printf("加载 winsock 失败，错误代码
    为: %d\n", WSAGetLastError());
161.         return FALSE;
162.     }
163.     // if 中的语句主要用于比对是否是 2.2 版本
164.     if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion
    ) != 2) {
165.         printf("不能找到正确的 winsock 版本\n");
166.         WSACleanup();
167.         return FALSE;
168.     }
169.     // 创建的 socket 文件描述符基于 IPV4，TCP
170.     ProxyServer = socket(AF_INET, SOCK_STREAM, 0);
171.     if (INVALID_SOCKET == ProxyServer) {
172.         printf("创建套接字失败，错误代码
    为:  %d\n", WSAGetLastError());
173.         return FALSE;
174.     }
175.     // 设置地址族为 IPv4
176.     ProxyServerAddr.sin_family = AF_INET;
177.     ProxyServerAddr.sin_port = htons(
178.         ProxyPort);  // 整型变量从主机字节顺序转变成网络字节顺序,转
    换为大端法
179.     ProxyServerAddr.sin_addr.S_un.S_addr =
180.         INADDR_ANY;  // 泛指本机也就是表示本机的所有 IP，多网卡的情况
    下，这个就表示所有网卡 ip 地址的意思
181.     // 将套接字与指定的 IP 地址和端口绑定。如果失败，输出错误信息
182.     if (bind(ProxyServer, (SOCKADDR*)&ProxyServerAddr, sizeof(SO
    CKADDR)) ==
183.         SOCKET_ERROR) {
184.         printf("绑定套接字失败\n");
185.         return FALSE;
186.     }
187.     // 启动监听，允许服务器接收来自客户端的连接
188.     if (listen(ProxyServer, SOMAXCONN) == SOCKET_ERROR) {
189.         printf("监听端口%d 失败", ProxyPort);
190.         return FALSE;
191.     }
192.     return TRUE;
193. }
```

```
194.//***********************************
195.// Method: ProxyThread
196.// FullName: ProxyThread
197.// Access: public
198.// Returns: unsigned int __stdcall
199.// Qualifier: 线程执行函数
200.// Parameter: LPVOID lpParameter
201.//***********************************
202.// 该代码实现了一个 HTTP 代理服务器线程的核心逻辑，
203.// 包括缓存机制、钓鱼网站重定向、以及对服务器响应的处理。
204.// 通过检查 Last-Modified 字段，实现了缓存的更新与使用，从而提高访问
    效率
205.unsigned int __stdcall ProxyThread(LPVOID lpParameter) {
206.    // 缓存客户端发来的数据
207.    char Buffer[MAXSIZE];
208.    // 用于存储接收到的数据
209.    char* CacheBuffer;
210.    // 初始化 Buffer 为 0
211.    ZeroMemory(Buffer, MAXSIZE);
212.    // 客户端地址
213.    SOCKADDR_IN clientAddr;
214.    // 地址长度
215.    int length = sizeof(SOCKADDR_IN);
216.    // 接收到的数据大小
217.    int recvSize;
218.    // 发送/接收结果
219.    int ret;
220.    // 存储 HTTP 头部信息的结构体
221.    HttpHeader* httpHeader = new HttpHeader();
222.    // 用于处理服务器返回的缓存
223.    char* cacheBuffer2 = NULL;
224.    // 分隔符
225.    char* delim = NULL;
226.    // 存储报文中的日期
227.    char date[DATELENGTH];
228.    // 记录拆分字符串后的剩余部分
229.    char* nextStr;
230.    // 用于遍历分割后的字符串
231.    char* p = NULL;
232.    // 缓存中的索引
233.    int index = 0;
234.    // 标记是否找到修改时间
235.    bool flag;
```

```
236.     recvSize = recv(((ProxyParam*)lpParameter)->clientSocket, Buffer, MAXSIZE,
237.         0);  // 接收到报文
238.
239.     if (recvSize <= 0) {
240.         goto error;
241.     }
242.     // 动态分配缓存
243.     CacheBuffer = new char[recvSize + 1];
244.     // 初始化
245.     ZeroMemory(CacheBuffer, recvSize + 1);
246.     // 拷贝客户端数据到缓存
247.     memcpy(CacheBuffer, Buffer, recvSize);
248.     // 处理 HTTP 头部
249.     ParseHttpHead(CacheBuffer, httpHeader);
250.     // 处理禁止访问网站
251.     if (strstr(httpHeader->url, invalid_website[0]) != NULL && button) {
252.         printf("\n====================\n");
253.         printf("--------该网站已被屏蔽!----------\n");
254.         goto error;
255.     }
256.     // 处理钓鱼网站
257.     if (strstr(httpHeader->url, fishing_src) != NULL && button)
    {
258.         printf("\n====================\n");
259.         printf(
260.             "-------------已从源网址：%s 转到 目的网址 ： %s ----------\n",
261.             fishing_src, fishing_dest);
262.         // 修改 HTTP 报文
263.         memcpy(httpHeader->host, fishing_dest_host,
264.             strlen(fishing_dest_host) + 1);
265.         memcpy(httpHeader->url, fishing_dest, strlen(fishing_dest));
266.     }
267.     delete CacheBuffer;
268.     // 连接目标主机
269.     if (!ConnectToServer(&((ProxyParam*)lpParameter)->serverSocket,
270.         httpHeader->host)) {
271.         goto error;
272.     }
273.     printf("代理连接主机 %s 成功\n", httpHeader->host);
```

```
274.
275.    index = isInCache(cache, *httpHeader);
276.    // 如果在缓存中存在
277.    if (index > -1) {
278.        char* cacheBuffer;
279.        char Buf[MAXSIZE];
280.        ZeroMemory(Buf, MAXSIZE);
281.        memcpy(Buf, Buffer, recvSize);
282.        // 插入"If-Modified-Since: "
283.        changeHTTP(Buf, cache[index].date);
284.        printf("--------------------请求报文----------------------
   --\n%s\n",
285.            Buf);
286.        ret = send(((ProxyParam*)lpParameter)->serverSocket, Buf
   ,
287.            strlen(Buf) + 1, 0);
288.        recvSize =
289.            recv(((ProxyParam*)lpParameter)->serverSocket, Buf,
   MAXSIZE, 0);
290.        printf("------------------Server 返回报文----------------
   ---\n%s\n",
291.            Buf);
292.        if (recvSize <= 0) {
293.            goto error;
294.        }
295.        char* No_Modified = (char*)"304";
296.        // 没有改变，直接返回 cache 中的内容
297.        if (!memcmp(&Buf[9], No_Modified, strlen(No_Modified)))
   {
298.            ret = send(((ProxyParam*)lpParameter)->clientSocket,

299.                cache[index].buffer, strlen(cache[index].buffer)
    + 1, 0);
300.            printf("将 cache 中的缓存返回客户端\n");
301.            printf("============================\n");
302.            goto error;
303.        }
304.    }
305.    // 将客户端发送的 HTTP 数据报文直接转发给目标服务器
306.    ret = send(((ProxyParam*)lpParameter)->serverSocket, Buffer,

307.        strlen(Buffer) + 1, 0);
308.    // 等待目标服务器返回数据
309.    recvSize =
```

```
310.         recv(((ProxyParam*)lpParameter)->serverSocket, Buffer, M
   AXSIZE, 0);
311.     if (recvSize <= 0) {
312.         goto error;
313.     }
314.     // 以下部分将返回报文加入缓存
315.     // 从服务器返回报文中解析时间
316.     cacheBuffer2 = new char[MAXSIZE];
317.     ZeroMemory(cacheBuffer2, MAXSIZE);
318.     memcpy(cacheBuffer2, Buffer, MAXSIZE);
319.     delim = (char*)"\r\n";
320.     ZeroMemory(date, DATELENGTH);
321.     p = strtok_s(cacheBuffer2, delim, &nextStr);
322.     flag = false;  // 表示是否含有修改时间报文
323.     // 不断分行，直到分出具有修改时间的那一行
324.     while (p) {
325.         if (p[0] == 'L')  // 找到Last-Modified:那一行
326.         {
327.             if (strlen(p) > 15) {
328.                 char header[15];
329.                 ZeroMemory(header, sizeof(header));
330.                 memcpy(header, p, 14);
331.                 if (!(strcmp(header, "Last-Modified:"))) {
332.                     memcpy(date, &p[15], strlen(p) - 15);
333.                     flag = true;
334.                     break;
335.                 }
336.             }
337.         }
338.         p = strtok_s(NULL, delim, &nextStr);
339.     }
340.     if (flag) {
341.         if (index > -1)  // 说明已经有内容存在，只要改一下时间和内
   容
342.         {
343.             memcpy(&(cache[index].buffer), Buffer, strlen(Buffer
   ));
344.             memcpy(&(cache[index].date), date, strlen(date));
345.         }
346.         else  // 第一次访问，需要完全缓存
347.         {
348.             memcpy(&(cache[cache_index % CACHE_NUM].httpHead.hos
   t),
349.                 httpHeader->host, strlen(httpHeader->host));
```

```
350.             memcpy(&(cache[cache_index % CACHE_NUM].httpHead.met
   hod),
351.                 httpHeader->method, strlen(httpHeader->method));

352.             memcpy(&(cache[cache_index % CACHE_NUM].httpHead.url
   ),
353.                 httpHeader->url, strlen(httpHeader->url));
354.             memcpy(&(cache[cache_index % CACHE_NUM].buffer), Buf
   fer,
355.                 strlen(Buffer));
356.             memcpy(&(cache[cache_index % CACHE_NUM].date), date,
    strlen(date));
357.             cache_index++;
358.         }
359.     }
360.     // 将目标服务器返回的数据直接转发给客户端
361.     ret = send(((ProxyParam*)lpParameter)->clientSocket, Buffer,
    sizeof(Buffer),
362.         0);
363. error:  // 错误处理
364.     printf("关闭套接字\n");
365.     Sleep(200);
366.     closesocket(((ProxyParam*)lpParameter)->clientSocket);
367.     closesocket(((ProxyParam*)lpParameter)->serverSocket);
368.     delete lpParameter;
369.     _endthreadex(0);
370.     return 0;
371. }
372. //************************************
373. // Method: ParseHttpHead
374. // FullName: ParseHttpHead
375. // Access: public
376. // Returns: void
377. // Qualifier: 解析 TCP 报文中的 HTTP 头部
378. // Parameter: char * buffer
379. // Parameter: HttpHeader * httpHeader
380. //************************************
381. // ParseHttpHead 函数从 HTTP 请求头中提取关键字段，
382. // 如 GET/POST 方法、URL、Host
383. // 和 Cookie， 并将这些信息存储在 HttpHeader
384. // 结构体中。代码通过循环逐行解析请求头，
385. // 针对每个字段进行匹配和处理，确保正确提取信息
386. void ParseHttpHead(char* buffer, HttpHeader* httpHeader) {
387.     char* p;
```

```
388.     char* ptr;
389.     const char* delim = "\r\n";
390.     p = strtok_s(buffer, delim, &ptr);  // 提取第一行
391.     printf("%s\n", p);
392.     if (p[0] == 'G') {  // GET 方式
393.         memcpy(httpHeader->method, "GET", 3);
394.         memcpy(httpHeader->url, &p[4], strlen(p) - 13);
395.     }
396.     else if (p[0] == 'P') {  // POST 方式
397.         memcpy(httpHeader->method, "POST", 4);
398.         memcpy(httpHeader->url, &p[5], strlen(p) - 14);
399.     }
400.     printf("%s\n", httpHeader->url);
401.     p = strtok_s(NULL, delim, &ptr);
402.     while (p) {
403.         switch (p[0]) {
404.         case 'H':  // Host
405.             memcpy(httpHeader->host, &p[6], strlen(p) - 6);
406.             break;
407.         case 'C':  // Cookie
408.             if (strlen(p) > 8) {
409.                 char header[8];
410.                 ZeroMemory(header, sizeof(header));
411.                 memcpy(header, p, 6);
412.                 if (!strcmp(header, "Cookie")) {
413.                     memcpy(httpHeader->cookie, &p[8], strlen(p)
    - 8);
414.                 }
415.             }
416.             break;
417.         default:
418.             break;
419.         }
420.         p = strtok_s(NULL, delim, &ptr);
421.     }
422. }
423. //************************************
424. // Method: ConnectToServer
425. // FullName: ConnectToServer
426. // Access: public
427. // Returns: BOOL
428. // Qualifier: 根据主机创建目标服务器套接字，并连接
429. // Parameter: SOCKET * serverSocket
430. // Parameter: char * host
```

```
//***********************************
// 创建 TCP 套接字：使用 IPv4 和 TCP 协议创建套接字。
// 主机名解析：通过 gethostbyname 将主机名解析为 IP 地址。
// 建立连接：使用 connect
// 函数尝试连接到服务器。如果连接失败，则关闭套接字并返回失败状态；如果成功，返回
// TRUE。
BOOL ConnectToServer(SOCKET* serverSocket, char* host) {
    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(HTTP_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));  // 将一个将网络地址转换成一个长整数型数
    *serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (*serverSocket == INVALID_SOCKET) {
        return FALSE;
    }
    if (connect(*serverSocket, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) ==
        SOCKET_ERROR) {
        closesocket(*serverSocket);
        return FALSE;
    }
    return TRUE;
}
// 该函数逐一比较 请求方法、主机名 和 URL 是否相同。
// 如果任一字段不同，则立即返回 false；如果全部相同，则返回 true。
// 这是一个简单的比较逻辑，用于判断两个 HTTP 请求是否是同一个请求
BOOL httpEqual(cacheHttpHead http1, HttpHeader http2) {
    if (strcmp(http1.method, http2.method)) return false;
    if (strcmp(http1.host, http2.host)) return false;
    if (strcmp(http1.url, http2.url)) return false;
    return true;
}
// 该函数用于在缓存中查找某个 HTTP 请求头是否存在。
// 遍历逻辑：逐一遍历缓存数组中的每个项，使用 httpEqual 进行比较。
// 返回值：
// 如果找到匹配项，则返回该项的索引；
```

```
472.// 如果未找到，则返回 -1 表示未命中缓存。
473.// 这段代码实现了一个简单的 缓存查找功能。
474.int isInCache(CACHE* cache, HttpHeader httpHeader) {
475.    int index = 0;
476.    for (; index < CACHE_NUM; index++) {
477.        if (httpEqual(cache[index].httpHead, httpHeader)) return
    index;
478.    }
479.    return -1;
480.}
481.// 该函数的主要作用是在 HTTP 请求报文的 "Host" 字段之前插入 "If-
   Modified-Since:
482.// <日期>\r\n"。 逻辑流程： 查找 Host 字段的位置。 将 Host
483.// 及其后的内容保存到临时缓冲区 temp。 截断原报文，并插入 "If-
   Modified-Since"
484.// 字段及日期。 重新拼接 Host 及其后的原始内容。 这种操作常用于实
   现 HTTP
485.// 缓存机制，通过 "If-Modified-Since" 字段检查资源是否更新。
486.void changeHTTP(char* buffer, char* date) {
487.    // 此函数在 HTTP 中间插入"If-Modified-Since: "
488.    const char* strHost = "Host";
489.    const char* inputStr = "If-Modified-Since: ";
490.    char temp[MAXSIZE];
491.    ZeroMemory(temp, MAXSIZE);
492.    char* pos = strstr(buffer, strHost);  // 找到 Host 位置
493.    int i = 0;
494.    // 将 host 与之后的部分写入 temp
495.    for (i = 0; i < strlen(pos); i++) {
496.        temp[i] = pos[i];
497.    }
498.    *pos = '\0';
499.    while (*inputStr != '\0') {  // 插入 If-Modified-Since 字段
500.        *pos++ = *inputStr++;
501.    }
502.    while (*date != '\0') {
503.        *pos++ = *date++;
504.    }
505.    *pos++ = '\r';
506.    *pos++ = '\n';
507.    // 将 host 之后的字段复制到 buffer 中
508.    for (i = 0; i < strlen(temp); i++) {
509.        *pos++ = temp[i];
510.    }
511.}
```